# Định luật Amdahl về tối ưu hóa hiệu suất hệ thống

## Định nghĩa

Định luật Amdahl là một công thức toán học mô tả mối quan hệ giữa tốc độ thực thi của một chương trình và tỷ lệ phần trăm thời gian được dành cho một phần cụ thể của chương trình đó.

## Công thức

Tăng tốc tối đa = 1 / (1 - (Phần được tối ưu hóa / 100))

**Giải thích:**

* **Tăng tốc tối đa:** Tốc độ tối đa có thể đạt được sau khi tối ưu hóa.
* **Phần được tối ưu hóa:** Tỷ lệ phần trăm thời gian được dành cho phần chương trình được tối ưu hóa.

**Ví dụ:**

Giả sử một chương trình được thực thi trong 100 giây và 20% thời gian (20 giây) được dành cho một phần cụ thể của chương trình. Nếu chúng ta có thể tối ưu hóa phần này để nó chạy nhanh gấp 5 lần, thời gian thực thi sẽ giảm xuống còn 80 giây.

**Tăng tốc tối đa:**

Tăng tốc tối đa = 1 / (1 - (20 / 100)) = 1.25

**Thời gian thực thi sau khi tối ưu hóa:**

Thời gian thực thi mới = Thời gian thực thi ban đầu / Tăng tốc tối đa = 100 giây / 1.25 = 80 giây

**Lưu ý:**

* Định luật Amdahl chỉ là một ước tính và không phải lúc nào cũng chính xác.
* Việc tối ưu hóa một phần nhỏ của chương trình có thể không mang lại lợi ích đáng kể nếu phần đó chỉ chiếm một phần nhỏ thời gian thực thi.
* Cần xác định các phần chương trình chiếm nhiều thời gian thực thi nhất để tập trung tối ưu hóa.

## Ứng dụng định luật Amdahl vào tối ưu hóa hệ thống:

* **Xác định các phần mềm hoặc phần cứng chiếm nhiều thời gian nhất trong hệ thống.**
* **Tập trung tối ưu hóa các phần mềm hoặc phần cứng đó.**
* **Có thể sử dụng các công cụ profiling để xác định các phần mềm hoặc phần cứng chiếm nhiều thời gian nhất.**
* **Có thể áp dụng các kỹ thuật tối ưu hóa khác nhau như nâng cấp phần cứng, cài đặt phần mềm mới hoặc tối ưu hóa cấu hình hệ thống.**

**Ví dụ:**

* **Nếu hệ thống của bạn thường xuyên bị nghẽn mạng, bạn có thể nâng cấp băng thông mạng hoặc tối ưu hóa cấu hình mạng.**
* **Nếu hệ thống của bạn thường xuyên bị chậm khi truy cập cơ sở dữ liệu, bạn có thể tối ưu hóa truy vấn cơ sở dữ liệu hoặc nâng cấp phần cứng cơ sở dữ liệu.**

**Lưu ý:**

* Việc tối ưu hóa hệ thống là một quá trình liên tục và cần được thực hiện thường xuyên để đảm bảo hiệu suất tối ưu cho hệ thống.
* Nên tham khảo ý kiến của chuyên gia để được tư vấn và hỗ trợ tối ưu hóa hệ thống hiệu quả.

## Các yếu tố quyết định tối ưu hóa phần nào

Để quyết định cải thiện phần nào của hệ thống thì còn phải dựa vào:

* Vùng ảnh hưởng
* Chi phí
* Những lợi ích và cải tiến đạt được
* Rủi ro có thể gặp phải

## Các bước thực hiện

Lập bảng liệt kê tuần tự và xác định mỗi bước chiếm bao nhiêu thời gian

|  |  |  |
| --- | --- | --- |
| Bước | Thời gian thực thi (đơn vị: s) | % (tổng thời gian) |
| A |  |  |
| B |  |  |
| C |  |  |

Áp dụng công thức Amdahl + đánh giá vùng ảnh hưởng, chi phi, những lợi ích và cải tiến đạt được:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Bước | Tăng tốc tối đa | Thời gian thực thi mới | Vùng ảnh hưởng | Chi phí | Lợi ích và cải tiến đạt được | Rủi ro |
| A |  |  |  |  |  |  |
| B |  |  |  |  |  |  |
| C |  |  |  |  |  |  |

Dựa vào đó để quyết định sẽ tối ưu hóa phần nào

## Tài liệu tham khảo

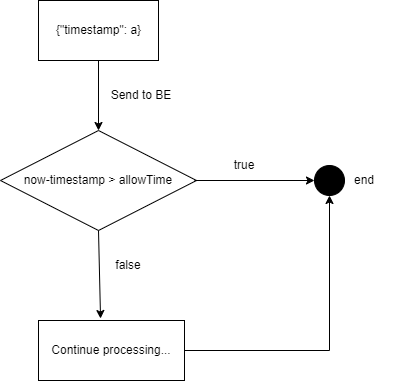
**[Thinking Clearly about Performance](https://queue.acm.org/detail.cfm?id=1854041)**

**[Improving the performance of complex software is difficult, but understanding some fundamental principles can make it easier](https://queue.acm.org/detail.cfm?id=1854041).**

# Các cách bảo mật API

## Tokens (JWT)

## Timestamp



## Chữ ký (signature generation)

Cơ chế:

* Bên gửi tạo public key, private key để ký và chỉ gửi đi public key
* Bên nhận: nhận public key để verify

## Nonce (number once – số dùng một lần)

Dùng để tránh reply attack

## Rate limits

## Ip access limits

## Blacklist

## Whitelist

# Xử lý Request Đồng thời

## Khóa lạc quan

### Giả định

Dữ liệu ít có khả năng bị xung đột khi truy cập đồng thời.

### Cách thức hoạt động

* Luồng truy cập dữ liệu sẽ **không** yêu cầu khóa trước khi truy cập.
* Luồng sẽ **kiểm tra** xem dữ liệu đã bị thay đổi bởi luồng khác hay chưa trước khi thực hiện thay đổi.
* Nếu dữ liệu **không** bị thay đổi, luồng sẽ thực hiện thay đổi và **cập nhật** dữ liệu.
* Nếu dữ liệu **bị** thay đổi, luồng sẽ **thử lại** hoặc **báo lỗi**.

### Ưu điểm

* Hiệu suất cao hơn vì không yêu cầu khóa trước khi truy cập dữ liệu.
* Tăng khả năng đồng thời vì nhiều luồng có thể truy cập dữ liệu cùng lúc.

### Nhược điểm

* Có thể xảy ra xung đột dữ liệu nếu nhiều luồng cố gắng thay đổi dữ liệu cùng lúc.
* Cần có cơ chế để giải quyết xung đột dữ liệu.

## Khóa bi quan

### Giả định

 Dữ liệu có khả năng cao bị xung đột khi truy cập đồng thời.

### Cách thức hoạt động

* Luồng truy cập dữ liệu sẽ **yêu cầu** khóa trước khi truy cập.
* Nếu khóa được cấp, luồng sẽ **truy cập** và **thay đổi** dữ liệu.
* Sau khi thay đổi xong, luồng sẽ **giải phóng** khóa.

### Ưu điểm

* Đảm bảo không xảy ra xung đột dữ liệu.
* Dễ dàng triển khai.

### Nhược điểm

* Hiệu suất thấp hơn vì luồng phải chờ đợi khóa trước khi truy cập dữ liệu.
* Giảm khả năng đồng thời vì chỉ có một luồng có thể truy cập dữ liệu cùng lúc.

## Lựa chọn phương pháp nào phù hợp

### Khóa lạc quan

Phù hợp cho các tình huống:

* Dữ liệu ít có khả năng bị xung đột.
* Hiệu suất cao là quan trọng.
* Có thể chấp nhận nguy cơ xung đột dữ liệu.

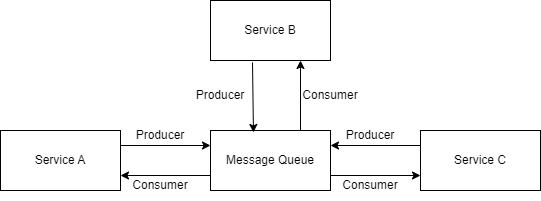
### Khóa bi quan

 Phù hợp cho các tình huống:

* Dữ liệu có khả năng cao bị xung đột.
* Tính toàn vẹn dữ liệu là quan trọng.
* Hiệu suất không phải là yếu tố quan trọng nhất.

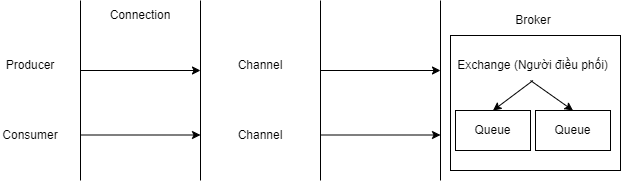
# Kiến trúc Message Queue

## Tại sao phải sử dụng message queue?



* Service gọi đến MQ => tách biệt các thành phần của hệ thống => giảm sự phụ thuộc giữa các service và tăng cường tính bảo mật.
* Xử lý bất đồng bộ => Cứ gửi lên MQ rồi không cần quan tâm => tăng hiệu suất
* Là nơi trung chuyển, điều phối => tăng khả năng chịu tải
* Message đẩy lên MQ => có các cơ chế bảo toàn và khôi phục => Đảm bảo hệ thống thông suốt
* Có thể sắp xếp độ ưu tiên của message => có thể lên lịch để thực hiện các message => linh hoạt và tăng hiệu suất

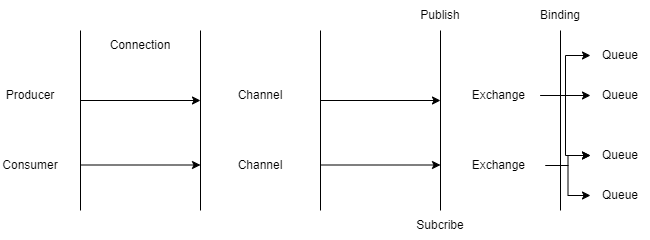
## Kiến trúc (RabbitMQ)



## Một số tham số

* Durable:
  + True: ghi queue vào rom, restart vẫn còn
  + False: ghi queue vào ram
* Persistant: True thì ghi vào rom
* noAck:
  + true: Gửi thì xóa luôn message
  + false: consumer phải gửi xác nhận thì mới xóa
* Ttl: Thời gian sống của một message

## Mô hình publish, subcribe



Exclusive: Nếu không có consumer nào subcribe thì clear hàng đợi

## Các loại exchange

### Direct: 1-1 (gửi đến queue có routingkey = routingkey)

* Định tuyến tin nhắn dựa trên **routing key** chính xác.
* Tin nhắn sẽ được gửi đến **queue** có **binding key** khớp chính xác với **routing key** của tin nhắn.
* **Ví dụ:**
  + **Routing key:** "news.sports"
  + **Binding key:** "news.sports"
  + **Queue:** "sports-news"

### Fanout: 1- All (không quan tâm routingkey)

* Gửi tin nhắn đến **tất cả** các **queue** đã được **binding** với **exchange**.
* **Routing key** không được sử dụng để định tuyến tin nhắn.
* **Ví dụ:**
  + **Routing key:** "any"
  + **Binding key:** "any"
  + **Queue:** "queue1"
  + **Binding key:** "any"
  + **Queue:** "queue2"

### Topic: 1- nhiều (routingkey và pattern)

* Định tuyến tin nhắn dựa trên **pattern matching** với **routing key**.
* Tin nhắn sẽ được gửi đến **queue** có **binding key** khớp với **routing key** của tin nhắn bằng **wildcard matching**.

#### \* (single wildcard)

* Phù hợp với bất kỳ ký tự nào tại vị trí được đặt.
* Ví dụ:
  + Routing key: "news.\*"
  + Binding key: "news.sports" -> Khớp
  + Binding key: "news.world" -> Khớp
  + Binding key: "news.sports.football" -> Không khớp

#### # (multi wildcard)

* Phù hợp với bất kỳ số lượng ký tự nào, bao gồm cả thư mục con.
* Ví dụ:
  + Routing key: "news.#"
  + Binding key: "news" -> Khớp
  + Binding key: "news.sports" -> Khớp
  + Binding key: "news.sports.football" -> Khớp

#### Kết hợp các wildcard

* Ví dụ:
  + Routing key: "\*.sports.#"
  + Binding key: "news.sports" -> Khớp
  + Binding key: "news.world" -> Không khớp
  + Binding key: "news.sports.football" -> Khớp

#### Lưu ý

* Wildcard chỉ được sử dụng trong binding key, không được sử dụng trong routing key.
* Wildcard không phân biệt hoa thường.

#### Bảng so sánh

|  |  |  |
| --- | --- | --- |
| Wildcard | Mô tả | Ví dụ |
| \* | Phù hợp với bất kỳ ký tự nào | news.\*, news.sports, news.world |
| # | Phù hợp với bất kỳ số lượng ký tự nào | news.#, news, news.sports, news.sports.football |

### Header: Dựa trên đặc tính của header

* Định tuyến tin nhắn dựa trên các **header** được đính kèm với tin nhắn.
* **Routing key** không được sử dụng để định tuyến tin nhắn.
* **Ví dụ:**
  + **Header:** "type": "news"
  + **Binding key:** {"type": "news"}
  + **Queue:** "news-queue"

# Tối ưu SQL

## “Kho hàng” - Database

### Kiến trúc

#### Các db khi tạo ra sẽ gồm

##### System Database

* Lưu trữ system catalogs;
  + Thông tin cấu hình
  + Thông tin các database trên instance
  + Thông tin DataFiles
  + Logins
  + …
* Bản chất dữ liệu không nằm ở System Database – nằm ở Resource Database

##### Model Database

* Là template để tạo db mới cho nhanh

##### TempDB Database

* Được tạo lại mỗi khi db khởi động:
  + Không cần recovery
  + Không cần backup
* Phục vụ chứa:
  + Temp table
  + Các công việc sử dụng dữ liệu tạm như SORT
* Tối ưu tempDb:
  + Mặc định SQL server chỉ tạo 1 temp file => Gây tắc nghẽn về I/O => Tạo nhiều tempfile
    - ALTER DATABASE tempdb

ADD FILE (

NAME = tempdev2,

FILENAME = 'path\_to\_file\_location\tempdb2.ndf',

SIZE = 500MB,

MAXSIZE = UNLIMITED,

FILEGROWTH = 100MB

);

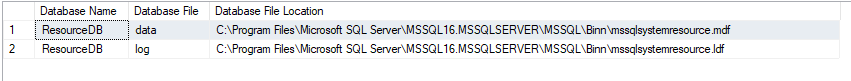
* + Ưu tiền đặt ở phần vùng có đọc ghi cao nhất (ổ ssd)
  + Temp Table cũng có thể tối ưu => Dùng index

##### MSDB

* Sử dụng cho SQL Server Agent service
* Sử dụng cho một số các service khác trong quản trị
* Ví dụ:
  + Cấu hình Scheduler
  + Cấu hình Log shipping
  + Cấu hình Database Email
  + …

##### Database “ẩn” – resource database

* Bản chất là nơi chứa các sytem objects – Các system store, fuction
* DBID = 32767
* Câu lệnh kiểm tra:
  + select 'ResourceDB' as 'Database Name', Name as [Database File], Filename as [Database File Location] from sys.sysaltfiles where DBID = 32767
  + Kết quả:



#### Cần lưu ý

Lưu ý thiết kế tempDb

Vấn đề thường xuất hiện trong các dự án chuyển đổi:

* Chuyển db sang SQL server instance khác
* Gốc của vấn đề: Một số objects của db cũ được lưu ở mức db instance, nên không chuyển theo được khi chúng ta chuyển ở mức db
  + Ví dụ:
    - Logins
    - Linked servers
    - Jobs
    - …

### Dữ liệu thực sự lưu vào đâu?

#### Data File (mục tiêu lưu trữ dữ liệu)

##### .MDF File (Master Database File)

* Đây là file chứa thông tin chính
  + Chứa thông tin INTERNAL CONFIGURATION
  + Chứa thông tin INTERNAL SYSTEM
* Chứa thông tin vị trí đến các datafile khác
* Bắt buộc có

##### .NDF File (Secondary Database File)

* Có thể có hoặc không
* Các datafile tạo mới thì đều tính là .NDF file

##### Cần lưu ý gì

Nhóm các files theo một logic lại cùng với nhau

* FileGroups
  + Mặc định hệ thống tạo ra PRIMARY FILEGROUP chứa .MDF
  + Chúng ta có thể tạo ra filegroup mới
    - Khuyến cáo sử dụng với hệ thống có dữ liệu lớn
    - Sử dụng trong chiến lược quy hoạch vòng đời dữ liệu

Lựa chọn Default FileGroups

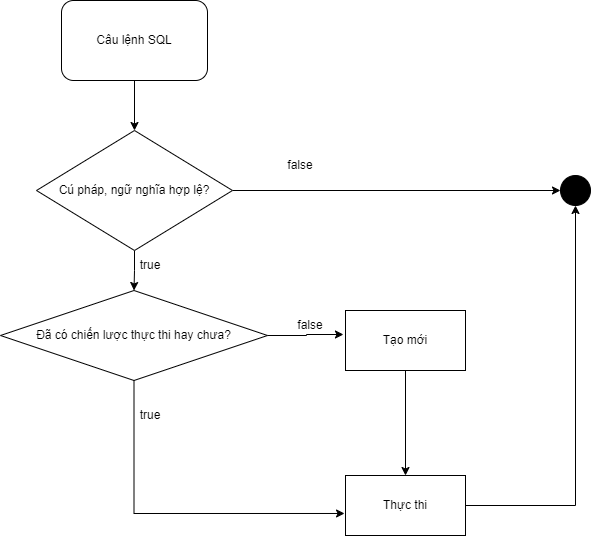
* Không nên lưu dữ liệu vào PRIMARY Filegroup

#### Transaction Log (mục tiêu là khôi phục dư liệu khi có sự cố)

Phải có chiến lược backup

## “Công nhân giao hàng” – Câu lệnh SQL

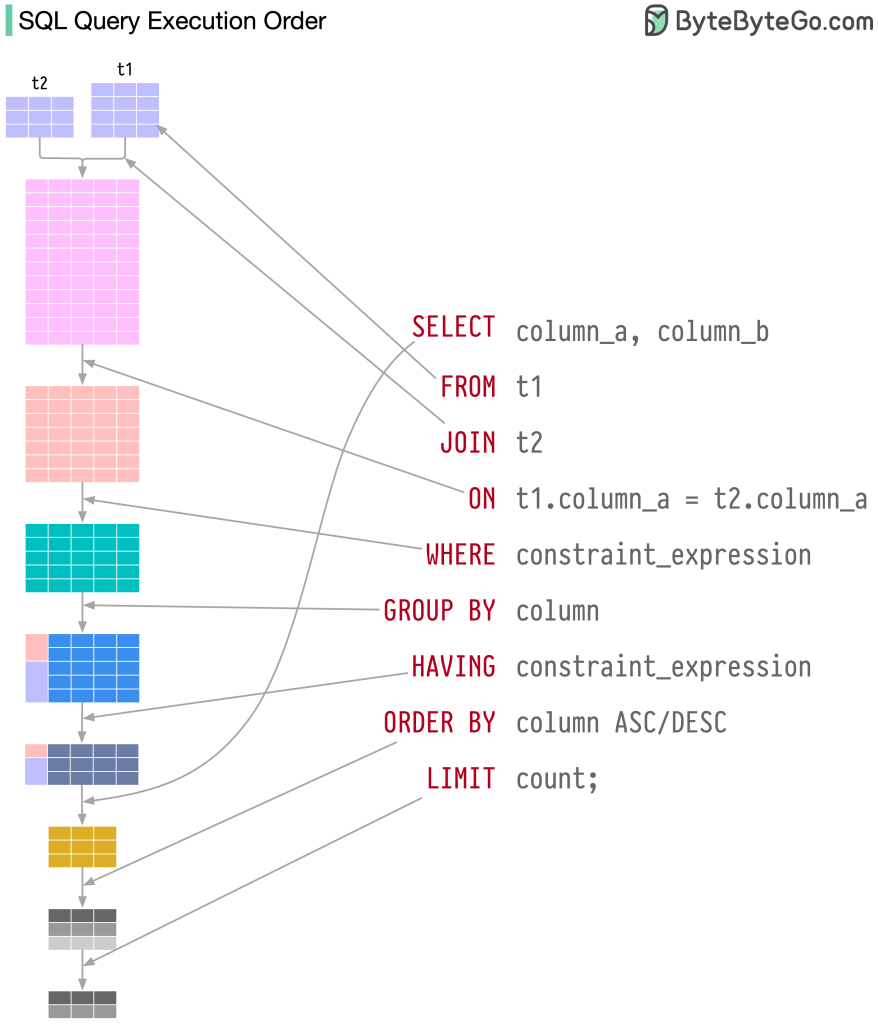
### “Công việc” - Cách thực hiện câu lệnh



#### Phân tích cú pháp, ngữ nghĩa

* Xác định xem câu lệnh SQL có tuân theo ngữ pháp SQL hay không.
* Kiểm tra xem các từ khóa, tên bảng, tên cột và các thành phần khác của câu lệnh SQL có hợp lệ hay không.
* Kiểm tra có quyền truy cập vào hay không.

##### Thứ tự chạy của cú pháp



##### Đổi chỗ các điều kiện ở select, join, where có ảnh hưởng đến hiệu năng hay không?

Câu trả lời là không (với điều kiện bình thường, bảng không có index composite. Nếu có composite thì đọc ở phần [Composite Index](#_Composite_Index)).

Kết quả thí nghiệm: Khác text => DB phân tích lại chiến lược thực thi => Cùng chiến lược thực thi

Giải thích: Chiến lược thực thi cũng như là một chỉ dẫn trên bản đồ, đi từ A đến B thì không khác từ B đến A.

#### Tạo chiến lực thực thi

“Công nhân” đến “kho” nhận hàng cần lựa chọn “đường đi”. Có nhiều “đường đi” để đến “kho”. Tại bước này, dựa vào cost – chi phí thực thi của chiến lược thực thi sẽ tìm ra đường đi ngắn nhất.

Ở bước này sẽ truy cập vào shared pool kiểm tra xem đã có chiến lược thực thi hay chưa, nếu có thì tái sử dụng (soft parse). Chưa có thì tạo chiến lược thực thi mới – hard parse (đây là bước tốn thời gian cần chú ý để tối ưu). Hard parse bao gồm:

* Lựa chọn phương pháp truy cập dữ liệu (như scan bảng, sử dụng chỉ mục)
* Lựa chọn thứ tự thực hiện các toán tử (như join, filter)
* Ước tính chi phí thực thi của truy vấn

##### Tối ưu truy vấn nhờ hạn chế hard parse

Ví dụ: Trong Oracle

###### Bài toán về tối ưu cơ sở dữ liệu đang gặp phải

* Hệ thống của chúng ta thường xuyên cần thực hiện câu lệnh tìm kiếm thông tin thông qua mã nhân viên (EMP\_ID).
* Thời điểm cao tải có thể có vài trăm nghìn session cùng thực hiện các câu lệnh tìm kiếm kiểu này.

*select \* from employees where emp\_id=1;*

*select \* from employees where emp\_id=2;*

*....*

*select \* from employees where emp\_id=1044;*

*select \* from employees where emp\_id=1045;*

*select \* from employees where emp\_id=1046;*

*….*

*select \* from employees where emp\_id=9400;*

Tôi sẽ giả lập việc chạy 100.000 câu lệnh này bằng cách chạy vòng lặp sau

*SQL> declare*

*emp\_rec employees%rowtype;*

*begin*

*for i in 1 .. 100000 loop*

*begin*

*execute immediate*

*'select \* from employees where emp\_id = ' || i*

*into emp\_rec;*

*exception*

*when no\_data\_found then null;*

*end;*

*end loop;*

*end;*

*/*

*PL/SQL procedure successfully completed.*

*Elapsed: 00:01:33.18*

Đoạn lệnh trên có ý nghĩa như sau:

* Thực thi lần lượt các câu lệnh dạng “select \* from employees where emp\_id = i” với i tăng dần từ 1 đến 100.000
* Vòng lặp for sẽ giúp chúng ta tạo ra 100.000 câu lệnh và thực thi chúng, giả lập đúng với bài toán đưa ra.

Thời gian thực thi của toàn bộ tiến trình này là 93.18 giây.

###### Phân tích các câu lệnh chạy trong thủ tục trên.

Chúng ta có thể thấy rằng 100.000 câu lệnh thực thi với ý nghĩa và chiến lược hoàn toàn giống nhau.

Bây giờ ta thử xem xét 1 câu lệnh riêng lẻ

SQL> select \* from employees where emp\_id=1044;

Elapsed: 00:00:00.00

Execution Plan

----------------------------------------------------------

Plan hash value: 2455998802

--------------------------------------------------------------------

| Id  | Operation     | Name   | Rows  | Bytes | Cost (%CPU)  | Time   |

--------------------------------------------------------------------

|   0 | SELECT STATEMENT     |   | 1 | 36 | 3   (0)  | 00:00:01 |

|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    | 1 | 36 | 3   (0)  | 00:00:01 |

|\*  2 |   INDEX UNIQUE SCAN     | SYS\_C0011152 | 1 |   | 2   (0)  | 00:00:01 |

Predicate Information (identified by operation id):

---------------------------------------------------

   2 - access("EMP\_ID"=1044)

Câu lệnh này chạy chỉ mấy vài ms (thời gian cực kỳ nhanh).

Chiến lược thực thi của câu lệnh đã là tối ưu:

* Đầu tiên hệ thống sẽ tìm kiếm thông tin theo UNIQUE INDEX (do cột EMP\_ID là PRIMARY KEY, hệ thống sẽ tự tạo 1 UNIQUE INDEX trên cột này).
* Do Index chỉ lưu thông tin của cột EMP\_ID, nên Oracle sẽ phải thực hiện truy cập vào TABLE để lấy nôt các thông tin của những cột khác (vì người dùng yêu cầu SELECT \* ). Đây chính là bước TABLE ACCESS BY INDEX ROWID
* Nếu bạn muốn tự mình có thể phân tích và hiểu tường tận về chiến lược thực thi của câu lệnh (Execution Plans) thì có thể đăng ký [chương trình tối ưu cơ sở dữ liệu cao cấp.](https://wecommit.com.vn/chuong-trinh-dao-tao-toi-uu-co-so-du-lieu-wecommit/)

Như vậy, mỗi câu lệnh SQL đã được tối ưu hết mức.

###### Phương án tối ưu trong trường hợp này là gì?

Trước khi đưa phương án, tôi muốn các bạn biết điều này.

Khi làm việc với Cơ sở dữ liệu, bạn có 2 cách thức.

* **Cách 1**: Truyền trực tiếp giá trị vào câu lệnh. Đây là cách trong ví dụ bên trên đã thực hiện. Chúng ta truyền luôn giá trị 1044 vào câu lệnh.

SQL> select \* from employees where emp\_id=1044;

* **Cách 2:** Truyền biến. Chúng ta sẽ đưa giá trị cần tìm kiếm vào 1 biến (ví dụ: biến :B1), khi thực hiện câu lệnh thì chúng ta sẽ khai báo cụ thể biến đó nhận giá trị bao nhiêu. Ví dụ như sau

SQL>  select \* from employees where emp\_id=:B1

Khi thực hiện sẽ cần nhập giá trị của biến :B1 sau.

**Hai cách thức trên đều trả ra cùng 1 kết quả.**

Bây giờ, thay vì sử dụng cách thức truyền giá trị thực tiếp trong vòng lặp FOR lúc đầu tiên, tôi sẽ chỉnh lại bằng cách sử dụng biến.

SQL> declare

  emp\_rec employees%rowtype;

begin

  for i in 1 .. 100000 loop

    begin

      execute immediate

        'select \* from employees where emp\_id = :B1'

        into emp\_rec using i;

    exception

      when no\_data\_found then null;

    end;

  end loop;

end;

/

PL/SQL procedure successfully completed.

Elapsed: 00:00:02.89

**Wow, thời gian hiện tại chỉ còn 2.89s (chỉ bằng ~3.1% so với thời gian lúc đầu là 93s).**

###### Giải thích

Đối với trường hợp truyền trực tiếp giá trị cho câu lệnh.

select \* from employees where emp\_id=1;

select \* from employees where emp\_id=2;

....

select \* from employees where emp\_id=1044;

select \* from employees where emp\_id=1045;

select \* from employees where emp\_id=1046;

….

select \* from employees where emp\_id=9400;

Do các câu lệnh có TEXT khác nhau (vì những giá trị truyền vào là khác nhau), vì vậy Cơ sở dữ liệu Oracle sẽ thực hiện như sau

* Tất cả các câu lệnh đều cần thực hiện bước số 1 – phân tích cú pháp. Cú pháp lệnh đúng nên sẽ được đi tiếp bước sau.
* Các câu lệnh đều thực hiện phân tích ngữ nghĩa. Tại đây Oracle thấy rằng ngữ nghĩa chuẩn xác, hệ thống tiếp tục đi sang bước thứ 3
* Tại bước kiểm tra thông tin trong Shared Pool, do đây là những câu lệnh khác nhau, **Oracle sẽ cần thực hiện 10.000 lần HARD PARSE.**
* Hệ thống phải thực hiện 10.000 lần bước số 4 và bước số 5 (đây là hai công việc cực kỳ tiêu tốn tài nguyên).
* Sau đó hệ thống thực hiện chạy câu lệnh theo chiến lược đã có ở bước số 5.

Đối với trường hợp chúng ta sử dụng phương pháp truyền biến

select \* from employees where emp\_id=:B1

Trong trường hợp này, Oracle sẽ nhìn thấy 10.000 câu lệnh truyền vào là giống nhau (vì đoạn text của câu lệnh giống nhau hoàn toàn). Do đó các công việc thực tế xảy ra như sau

* Hệ thống chỉ cần thực hiện phân tích câu lệnh 1 lần duy nhất (đối với câu lệnh đầu tiên).
* Với 9.999 câu lệnh còn lại, Oralce sẽ lấy luôn chiến lược thực thi (thực hiện SOFT PARSE – bỏ qua 2 bước tiêu tốn tài nguyên là bước số 4 và bước số 5).

Chính việc này dẫn đến kết quả tối ưu đã cải thiện hơn 97% (từ 93s còn ~3s).

#### Thực thi

Dựa vào kế hoạch truy vấn, máy chủ SQL sẽ truy cập dữ liệu từ cơ sở dữ liệu.

Quá trình này bao gồm:

* Truy cập dữ liệu từ các bảng, view hoặc derived table
* Áp dụng các điều kiện lọc và toán tử
* Sắp xếp dữ liệu theo yêu cầu

### “Công cụ kiểm tra công việc- Jira của sql” – Chiến lực thực thi

#### Định nghĩa

Chiến lược thực thi là tập hợp các bước mà hệ thống quản trị cơ sở dữ liệu (RDBMS) thực hiện để xử lý một câu lệnh SQL. Chiến lược này quyết định cách thức truy vấn dữ liệu, sử dụng các chỉ mục, sắp xếp dữ liệu và thực hiện các phép toán.

#### Cách đọc

##### SQL server

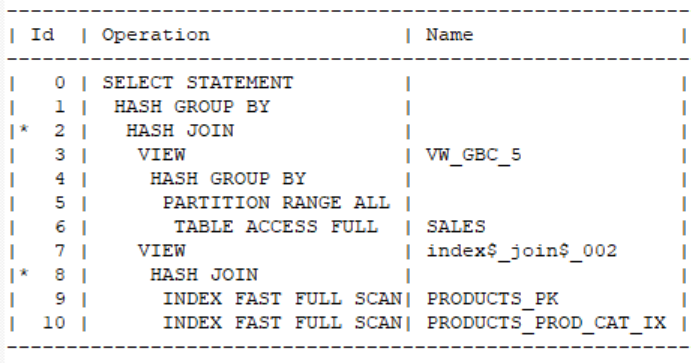
* Đọc từ Phải sang trái, từ trên xuống dưới

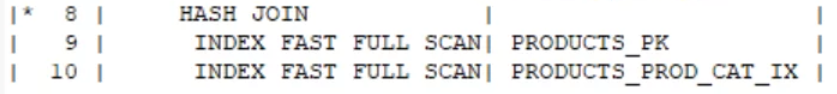
##### Mysql

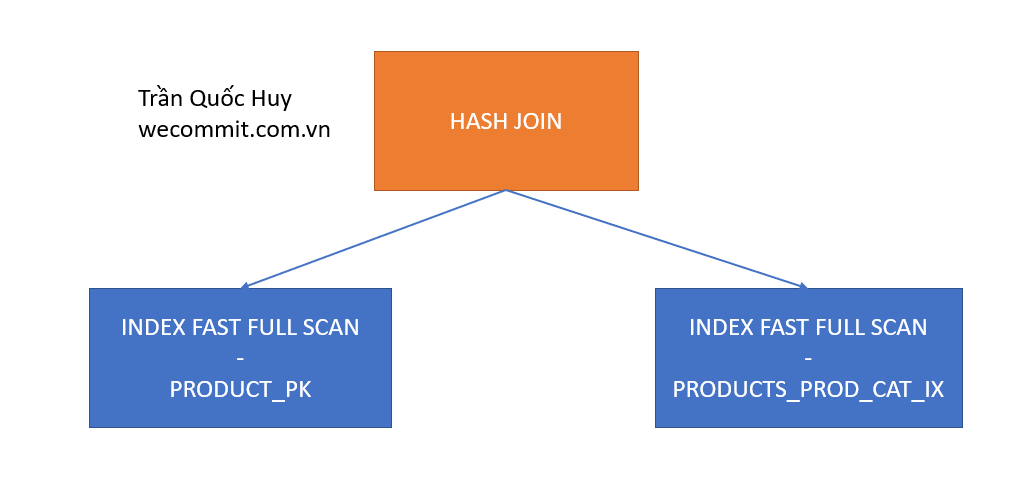
* (Dùng **explain analyze** ở đầu câu lệnh để kiểm tra chiến lược thực thi)
* Ví dụ về một SQL Exeuction Plans

Đây là chiến lược thực thi của câu lệnh

SELECT prod\_category, AVG (amount\_sold)  
FROM wecommit.sales s, wecommit.products p  
WHERE p.prod\_id = s.prod\_id  
GROUP BY prod\_category



* Mỗi một dòng trong biểu đồ SQL Execution Plans tương ứng với một hành động nào đó (Operation). Các hành động này có liên kết với nhau theo cấp bậc CHA – CON. Về bản chất chiến lược thực thi có hình dạng là một CÂY CHA – CON.
* Cột bên tay phải với tên Name cho chúng ta biết các hành động tương ứng được thực hiện trên đối tượng nào (Table tên là gì hoặc Index tên là gì).
* Ví dụ:
  + Ở hành động có ID=6, hệ thống thực hiện công việc là TABLE ACCESS FULL, công việc này thực hiện trên đối tượng là SALES (bảng SALES trong Cơ sở dữ liệu)
  + Ở hành động có ID=9, hệ thống thực hiện công việc là INDEX FAST FULL SCAN trên đối tượng tên là PRODUCTS\_PK (đây chính là Index Primary Key của bảng Products, tên Index là PRODUCTS\_PK)
  + Nếu để ý thì anh em sẽ thấy trong biểu đồ SQL Execution Plans bên trên có những sự “THÒ THỤT ở LỀ TRÁI” một cách rất đẹp, đều, phần này không phải được bố trí một cách ngẫu nhiên, mà sự THÒ THỤT LỀ này có ý nghĩa rất quan trọng trong quá trình chúng ta phân tích.
  + Tất cả những hành động nào cùng cấp bậc thì sẽ có mức độ CĂN LỀ TRÁI giống nhau
  + Ví dụ: trong ví dụ trên, chúng ta xét cụm các hành động số 8,9,10
  + 
  + Hành động số 9 và số 10 được tính là cùng cấp bậc (do có độ CĂN LỀ TRÁI giống nhau)
  + Hành động số 8 là CHA của hành động số 9 và số 10: vì trong SQL Execution Plans, bước số 8 ở gần bên trái hơn và ở ngay bên trên của bước con số 9.
  + Có thể hình dung nếu ở dạng cây thì cụm hành động số 8, 9, 10 sẽ có hình dáng như sau



* NGUYÊN LÝ của việc đọc hiểu chiến lược thự thi đó là: biết cách xây dựng lên SQL Execution Plans Tree như tôi vừa miêu tả bên trên.
* Sau khi đã có được mô hình dạng cây rồi, chúng ta sẽ biết được thứ tự thực hiện bằng cách: Bắt đầu từ NODE CON bên TRÁI và lần lượt dò qua toàn bộ cây.

#### Ý nghĩa các thành phần trong execute plan (SQL server)

##### SQL server

###### Scan/Seek Operators

* **Scan**: Đọc toàn bộ hoặc một phần lớn dữ liệu từ một bảng hoặc chỉ mục.
* **Seek**: Tìm kiếm dữ liệu cụ thể trong một chỉ mục.
* Ví dụ: Trong một câu lệnh SELECT, một Scan Operator có thể chỉ ra rằng cơ sở dữ liệu đang quét toàn bộ bảng để trả về kết quả.

###### Join Operators

* **Nested Loop Join**: Thực hiện một vòng lặp qua từng hàng của bảng ngoài và thực hiện một tìm kiếm cho mỗi hàng trong bảng trong.
* **Merge Join**: Sắp xếp các bảng theo các cột kết nối và sau đó thực hiện việc kết hợp từng cặp hàng có giá trị kết nối giống nhau.
* **Hash Join**: Tạo một bảng băm trong bộ nhớ cho mỗi bảng tham gia, sau đó thực hiện việc kết hợp các hàng dựa trên giá trị băm.
* Ví dụ: Một execution plan có thể chứa một Nested Loop Join để kết hợp hai bảng dựa trên một điều kiện.

###### Filter Operators

* Lọc các hàng dữ liệu dựa trên điều kiện được chỉ định.
* Ví dụ: Một Filter Operator có thể được sử dụng để lọc các hàng không thỏa mãn một điều kiện trong câu lệnh WHERE.

###### Sort Operators

* Sắp xếp các hàng dữ liệu theo một hoặc nhiều cột.
* Ví dụ: Nếu câu lệnh SELECT yêu cầu kết quả được sắp xếp theo một cột cụ thể, một Sort Operator có thể được thấy trong execution plan.

###### Aggregate Operators

* Thực hiện các phép tổng hợp như SUM, AVG, MAX, MIN trên một tập hợp các giá trị. Ví dụ: Nếu câu lệnh SELECT chứa một hàm tổng hợp như SUM(), một Aggregate Operator sẽ được thấy trong execution plan.

###### Access Methods

* Chỉ ra cách cơ sở dữ liệu truy cập dữ liệu, chẳng hạn như sử dụng quét toàn bộ bảng, quét theo chỉ mục, hoặc tìm kiếm theo hàng.
* Ví dụ: Một Access Method có thể chỉ ra rằng cơ sở dữ liệu sử dụng một chỉ mục để thực hiện một Seek hoặc Scan.

###### Tài liệu tham khảo cách đọc chiến lược thực thi

<https://www.sqlshack.com/sql-server-query-execution-plan-beginners-types-options/>

##### MySql

###### EXPLAIN [option] SELECT ...

Option:

* PARTITIONS: Hiển thị thông tin về cách phân vùng của bảng.
* EXTENDED: Hiển thị thông tin mở rộng về cách thực thi câu truy vấn.
* FORMAT=JSON: Hiển thị kết quả dưới dạng JSON.

###### Ý nghĩa của các trường thông tin trả ra

EXPLAIN SELECT \* FROM categories\G

Explain

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: categories

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 4

filtered: 10

Extra:

1 row in set (0.00 sec)

Ý nghĩa các trường trả ra:

* id - Số thứ tự cho mỗi câu SELECT trong truy vấn của bạn (trường hợp bạn sử dụng các truy vấn lồng nhau (nested subqueries).
* select\_type - Loại của câu SELECT. Có thể có các giá trị sau.
  + SIMPLE - Truy vấn là một câu SELECT cơ bản, không có bất cứ truy vấn con (subqueries) hay câu lệnh hợp (UNION) nào.
  + PRIMARY - Truy vấn là câu SELECT ngoài cùng của một lệnh JOIN.
  + DERIVED - Truy vấn là một truy vấn con của truy vấn khác, nằm trong lệnh FROM.
  + SUBQUERY - Truy vấn đầu tiên của một truy vấn con.
  + DEPENDENT SUBQUERY - Truy vấn con, phụ thuộc vào một truy vấn khác bên ngoài nó.
  + UNCACHEABLE SUBQUERY - Truy vấn không thể lưu lại được (có quy định điều kiện cụ thể, thế nào là một truy vấn có thể lưu lại được).
  + UNION - Truy vấn là câu SELECT thứ hai của lệnh UNION.
  + DEPENDENT UNION - Truy vấn thứ hai hoặc các truy vấn tiếp theo của lệnh UNION phụ thuộc vào một truy vấn bên ngoài.
  + UNION RESULT - Truy vấn là kết quả của lệnh UNION.
* table - Bảng liên quan đến câu truy vấn.
* type - Cách MySQL join các bảng lại với nhau. Đây là một trong những trường quan trọng nhất của kết quả trả về, nó chỉ ra đâu là nơi thiếu chỉ mục (index) và làm cách nào truy vấn của bạn cần phải xem xét lại. Các giá trị trả về có thể là.
  + system - Bảng không có hoặc chỉ có 1 dòng.
  + const - Bảng chỉ có duy nhất 1 dòng đã được đánh chỉ mục mà khớp với điều kiện tìm kiếm. Đây là loại join nhanh nhất, bởi bảng chỉ cần đọc một lần duy nhất và giá trị của cột được xem như là hằng số khi join với các bảng khác.
  + eq\_ref - Tất cả các thành phần của index được sử dụng bởi lệnh join và index thuộc loại PRIMARY KEY hoặc UNIQUE NOT NULL. Đây là loại join tốt thứ hai (chỉ sau const).
  + ref - Tất cả các dòng khớp với điều kiện tìm kiếm và chưa cột đã được index đều được đọc cho mỗi sự kết hợp với các dòng của bảng trước đó. Loại join này có thể thấy khi so sánh cột với điều kiện = hoặc <=>.
  + fulltext - join sử dụng chỉ mục dạng FULLTEXT.
  + ref\_or\_null - Gần giống như ref nhưng chứa cả các dòng với cột mang giá trị null.
  + index\_merge - join sử dụng một danh sách các chỉ mục để tạo ra tập kết quả. Cột key trong kết quả của EXPLAIN sẽ liệt kê các khóa được sử dụng.
  + unique\_subquery - Truy vấn con với lệnh IN trả về duy nhất một kết quả và sử dụng primary key.
  + index\_subquery - Gần giống như unique\_subquery nhưng trả về nhiều hơn một dòng.
  + range - Chỉ mục được dùng để tìm ra các dòng thỏa mãn điều kiện tìm kiếm, cụ thể là khi khóa được so sánh với hằng số thông qua các toán tử BETWEEN, IN, >, >=,...
  + index - Toàn bộ cây chỉ mục được duyệt để tìm ra dòng thỏa mãn điều kiện.
  + all - Toàn bộ bảng được duyệt để tìm dòng cho join. Đây là loại join tồi tệ nhất và thường cho thấy sự thiếu xót trong việc đánh chỉ mục.
* possible\_keys - Hiển thị keys có thể được dùng bởi MySQL để tìm dòng trong bảng, tuy nhiên nó có thể hoặc không thể được dùng. Trong thực tế, cột này đôi khi giúp cho việc tối ưu truy vấn, bởi nếu cột này trống (NULL), nó thường cho thấy không có chỉ mục liên quan được định nghĩa trong bảng.
* *key* - Khóa thực sự được sử dụng bởi MySQL. Cột này có thể chứa khóa không được liệt kê ở cột possible\_keys. Trình tối ưu của MySQL luôn cố gắng tìm kiếm khóa tối ưu nhất cho truy vấn. Khi kết hợp nhiều bảng, nó có thể dùng khóa không nằm trong danh sách possible\_keys nhưng lại đem về hiệu quả cao hơn.
* key\_len - Độ dài của key (số byte được đọc từ index). ref - Tên cột hoặc hằng số được dùng để so sánh với chỉ mục được nêu ra ở cột key. MySQL có thể lấy ra một hằng số, hoặc một cột cho quá trình thực hiện truy vấn. Bạn có thể thấy trong ví dụ sẽ được liệt kê dưới đây.
* rows - Số hàng được dự đoán sẽ được đọc hoặc đánh giá. Đây cũng là một cột hết sức quan trọng cho việc tối ưu truy vấn, nhất là khi bạn dùng JOIN hoặc truy vấn con.
* filtered:
  + Tỷ lệ phần trăm số hàng được đọc sau khi áp dụng điều kiện WHERE. Các giá trị gần 100% tốt hơn.
* Extra - Các thông tin bổ sung liên quan đến quá trình thực hiện truy vấn. Các giá trị kiểu như Using Temporary (dùng tạm thời), Using filesort (dùng sắp xếp file),... của cột này có thể cho thấy một truy vấn không thực sự tốt. Danh sách đầy đủ của các giá trị có thể có ở cột này có thể xem tại [Tài liệu MySQL](http://dev.mysql.com/doc/refman/5.6/en/explain-output.html#explain-extra-information)
  + Using index: Đọc toàn bộ dữ liệu từ chỉ mục.
  + Using where: Sử dụng điều kiện WHERE sau khi đọc dữ liệu từ chỉ mục.
  + Using temporary: Sử dụng bảng tạm thời để lưu trữ kết quả trung gian.
  + Using filesort: Sắp xếp dữ liệu bằng cách đọc từ đĩa cứng.

Bạn có thể bổ sung thêm từ khóa EXTENDED sau EXPLAIN và MySQL sẽ đưa ra các thông tin bổ sung về quá trình thực hiện truy vấn. Để xem chi tiết, thực hiện lệnh SHOW WARNINGS ngay sau lệnh EXPLAIN. Nó thường được dùng để xem các câu truy vấn được thực hiện sau bất cứ thay đổi nào được tạo ra bởi Query Optimizer.

EXPLAIN EXTENDED SELECT City.Name FROM City

JOIN Country ON (City.CountryCode = Country.Code)

WHERE City.CountryCode = 'IND' AND Country.Continent = 'Asia'\G

Explain

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: Country

type: const

possible\_keys: PRIMARY

key: PRIMARY

key\_len: 3

ref: const

rows: 1

filtered: 100.00

Extra:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: City

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 4079

filtered: 100.00

Extra: Using where

2 rows in set, 1 warning (0.00 sec)

SHOW WARNINGS\G

Explain

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Level: Note

Code: 1003

Message: select `World`.`City`.`Name` AS `Name` from `World`.`City` join `World`.`Country` where ((`World`.`City`.`CountryCode` = 'IND'))

1 row in set (0.00 sec)

#### Hint

##### Định nghĩa

Nếu không muốn dùng chiến lược thực thi mà hệ thống chọn thì dùng hint để chỉ định chiến lược thực thi

##### Phân loại

* Join Hints: Được sử dụng để chỉ định cách thức thực hiện join giữa các bảng.
* Query Hints: Gợi ý cho trình tối ưu hóa thực hiện các biến đổi truy vấn nhất định.
* Table Hints: Cung cấp gợi ý về cách truy cập hoặc sử dụng bảng trong truy vấn.

##### Giới thiệu một số hint

###### INDEX hint

* **Định nghĩa:** INDEX hint được sử dụng để gợi ý cơ sở dữ liệu sử dụng một chỉ mục cụ thể cho một truy vấn, thay vì để hệ thống quyết định mặc định.
* **Ý nghĩa:** Hint này có thể được sử dụng khi bạn muốn chắc chắn rằng một chỉ mục nhất định được sử dụng trong quá trình thực thi của truy vấn, thường là khi hệ thống không chọn một chỉ mục mà bạn cho là tối ưu.
* **Ví dụ:**

SELECT \* FROM employees WHERE salary > 5000 /\*+ INDEX(emp\_salary\_idx)\*/;

Trong trường hợp này, chúng ta sử dụng INDEX hint để gợi ý hệ thống sử dụng chỉ mục emp\_salary\_idx cho truy vấn, giả sử rằng hệ thống có thể không chọn chỉ mục này mặc định.

###### FORCE INDEX hint

* **Định nghĩa:** FORCE INDEX hint buộc cơ sở dữ liệu sử dụng một chỉ mục cụ thể cho một truy vấn, thay vì để hệ thống quyết định mặc định.
* **Ý nghĩa:** Hint này tương tự như INDEX hint, nhưng nó bắt cơ sở dữ liệu sử dụng chỉ mục chỉ định và không xem xét các phương pháp thực thi khác.
* **Ví dụ:**

SELECT \* FROM products FORCE INDEX(product\_name\_idx) WHERE product\_name LIKE 'A%';

Trong ví dụ này, FORCE INDEX hint yêu cầu cơ sở dữ liệu sử dụng chỉ mục product\_name\_idx cho truy vấn, bất kể phương pháp thực thi nào khác có thể được ưu tiên.

###### USE INDEX hint

* **Định nghĩa:** USE INDEX hint chỉ định cơ sở dữ liệu sử dụng một hoặc nhiều chỉ mục cụ thể cho việc thực thi truy vấn.
* **Ý nghĩa:** Hint này cho phép bạn lựa chọn cụ thể các chỉ mục sẽ được sử dụng trong quá trình thực thi truy vấn, nhưng hệ thống vẫn có thể quyết định sử dụng các chỉ mục khác nếu nó cho rằng chúng tốt hơn.
* **Ví dụ:**

SELECT \* FROM orders USE INDEX(order\_date\_idx) WHERE order\_date BETWEEN '2023-01-01' AND '2023-12-31';

Trong trường hợp này, USE INDEX hint chỉ định sử dụng chỉ mục order\_date\_idx cho truy vấn, nhưng hệ thống vẫn có thể chọn một chỉ mục khác nếu nó hiệu quả hơn.

###### IGNORE INDEX hint

* **Định nghĩa:** IGNORE INDEX hint gợi ý cơ sở dữ liệu bỏ qua việc sử dụng một hoặc nhiều chỉ mục cụ thể cho truy vấn.
* **Ý nghĩa:** Hint này được sử dụng khi bạn muốn hệ thống không sử dụng một số chỉ mục trong quá trình thực thi truy vấn, thường là khi các chỉ mục đó không hiệu quả hoặc gây ra vấn đề.
* **Ví dụ:**

SELECT \* FROM products IGNORE INDEX(product\_name\_idx) WHERE product\_category = 'Electronics';

Trong ví dụ này, IGNORE INDEX hint yêu cầu cơ sở dữ liệu bỏ qua chỉ mục product\_name\_idx cho truy vấn, có thể do chỉ mục này không được tối ưu cho truy vấn cụ thể này.

###### JOIN Hint

* **Định nghĩa:** Hint này chỉ định cách thức thực hiện phép nối (join) trong truy vấn.
* **Ý nghĩa:** Sử dụng để kiểm soát cách thức thực hiện phép nối giữa các bảng.
* **Ví dụ:**

SELECT \* FROM orders o STRAIGHT\_JOIN customers c ON o.customer\_id = c.customer\_id;

Trong ví dụ trên, hint **STRAIGHT\_JOIN** yêu cầu thực hiện phép nối theo thứ tự được chỉ định, bắt đầu từ bảng **orders**.

###### USE\_NL Hint

* **Định nghĩa:** Hint này chỉ định sử dụng phép nối theo thuật toán Nested Loop.
* **Ý nghĩa:** Sử dụng để bắt buộc trình tối ưu hóa sử dụng phép nối theo Nested Loop.
* **Ví dụ:**

SELECT /\*+ USE\_NL(e) \*/ \* FROM employees e, departments d WHERE e.department\_id = d.department\_id;

Trong ví dụ trên, hint **USE\_NL** yêu cầu sử dụng phép nối theo Nested Loop giữa bảng **employees** và **departments**.

###### USE\_MERGE Hint

* **Định nghĩa:** Hint này chỉ định sử dụng phép nối theo thuật toán Merge Join.
* **Ý nghĩa:** Sử dụng để bắt buộc trình tối ưu hóa sử dụng phép nối theo Merge Join.
* **Ví dụ:**

SELECT /\*+ USE\_MERGE(e) \*/ \* FROM employees e, departments d WHERE e.department\_id = d.department\_id;

Trong ví dụ trên, hint **USE\_MERGE** yêu cầu sử dụng phép nối theo Merge Join giữa bảng **employees** và **departments**.

##### Chú ý

Cần cẩn thận vì việc dùng hint giống như ta chỉ định cách thực thi câu lệnh SQL một cách cố định. Khi môi trường (statistic, object, bản ghi) thay đổi thì chiến lược sẽ không thay đổi mà vẫn giữ nguyên chiến lược chỉ định trong hint. Nếu ta không đánh giá lại thì câu lệnh sẽ chậm.

Ví dụ: Con đường quốc lộ mới đâm thẳng qua núi so với con đường đèo cũ quanh co. Nếu chúng ta cứng nhắc đi theo con đường đèo cũ mặc dù có con đường mới nhanh hơn, thì hiệu suất sẽ bị ảnh hưởng.

### “Đối tượng làm việc”

#### Table

##### Thiết kế bảng tối ưu

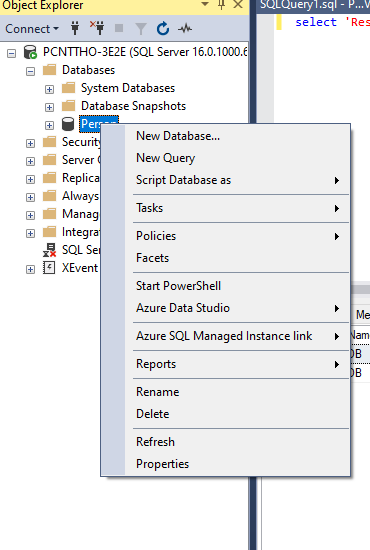
* **Tính đơn giản và hiệu quả:**
  + Mỗi bảng nên chứa thông tin về một thực thể duy nhất hoặc một quan hệ duy nhất giữa các thực thể.
  + Tránh việc tạo ra các bảng quá phức tạp hoặc chứa quá nhiều thông tin không liên quan.
* **Tối ưu hóa dung lượng lưu trữ:**
  + Sử dụng kiểu dữ liệu phù hợp cho mỗi cột để giảm thiểu dung lượng lưu trữ cần thiết.
  + Tránh lặp lại dữ liệu bằng cách phân chia dữ liệu thành các bảng và sử dụng khóa ngoại để tham chiếu tới dữ liệu chung.
* **Tối ưu hóa cấu trúc bảng:**
  + Tránh sử dụng quá nhiều các cột không cần thiết trong một bảng để giảm dung lượng lưu trữ và tăng tốc độ truy xuất.
  + Phân bố dữ liệu một cách hợp lý giữa các bảng để tránh tình trạng quá nhiều dữ liệu lưu trữ trong một bảng và ít dữ liệu trong các bảng khác.
* **Tối ưu hóa truy xuất dữ liệu:**
  + Xác định các trường mà bạn sẽ thường xuyên truy xuất và tạo chỉ mục cho chúng để tăng tốc độ truy xuất dữ liệu.
  + Tối ưu hóa câu truy vấn để sử dụng chỉ mục một cách hiệu quả.
* **Tối ưu hóa quan hệ giữa các bảng:**
  + Xác định và thiết lập các quan hệ giữa các bảng một cách rõ ràng và logic, sử dụng khóa ngoại.
  + Đảm bảo rằng các quan hệ được thiết lập một cách hợp lý để giảm thiểu việc thực hiện các phép nối (join) phức tạp trong các truy vấn.
* **Đảm bảo tính toàn vẹn dữ liệu:**
  + Sử dụng ràng buộc (constraints) như NOT NULL, UNIQUE, hoặc CHECK để đảm bảo tính toàn vẹn của dữ liệu.
  + Xác định các quy tắc và điều kiện cần thiết để ngăn chặn dữ liệu không hợp lệ được lưu trữ trong cơ sở dữ liệu.
* **Tối ưu hóa cho việc thêm, sửa, xóa dữ liệu:**
  + Thiết kế các bảng sao cho việc thêm, sửa, xóa dữ liệu có thể được thực hiện một cách dễ dàng và hiệu quả.
  + Tránh tình trạng phụ thuộc quá mức vào các thao tác cập nhật (update) hoặc xóa (delete) để tránh gây ra các vấn đề về hiệu suất.

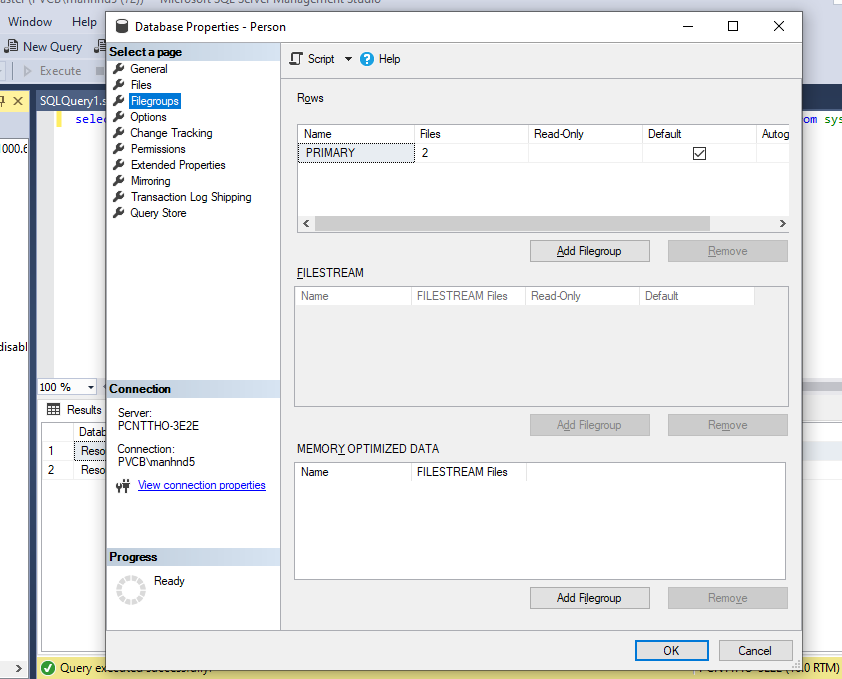
##### NDF

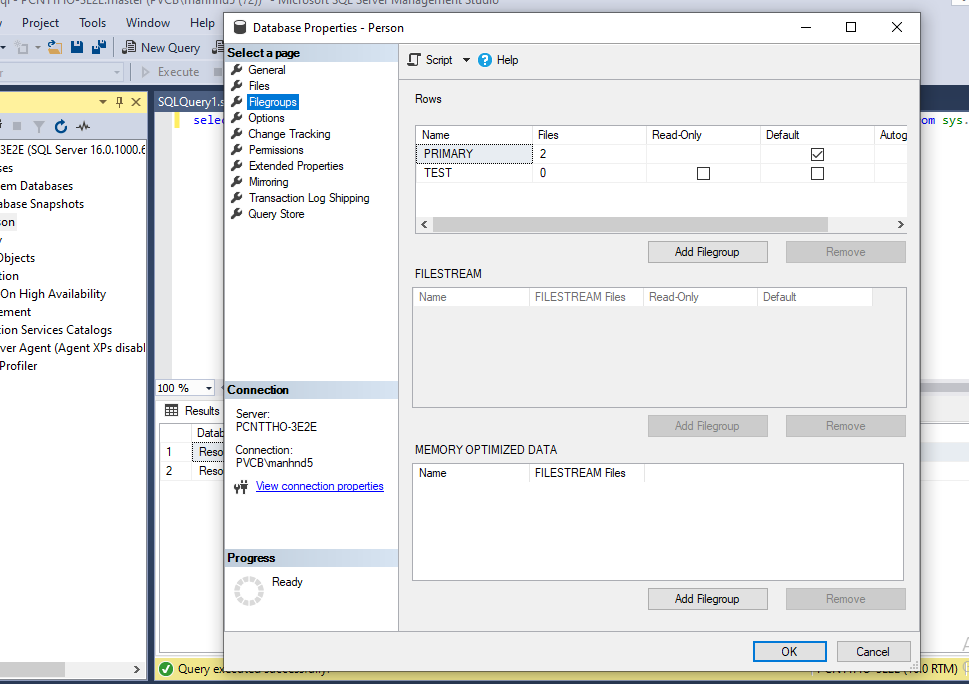
Tạo ra các file ndf riêng để lưu các data có cùng nhóm

Cách làm:

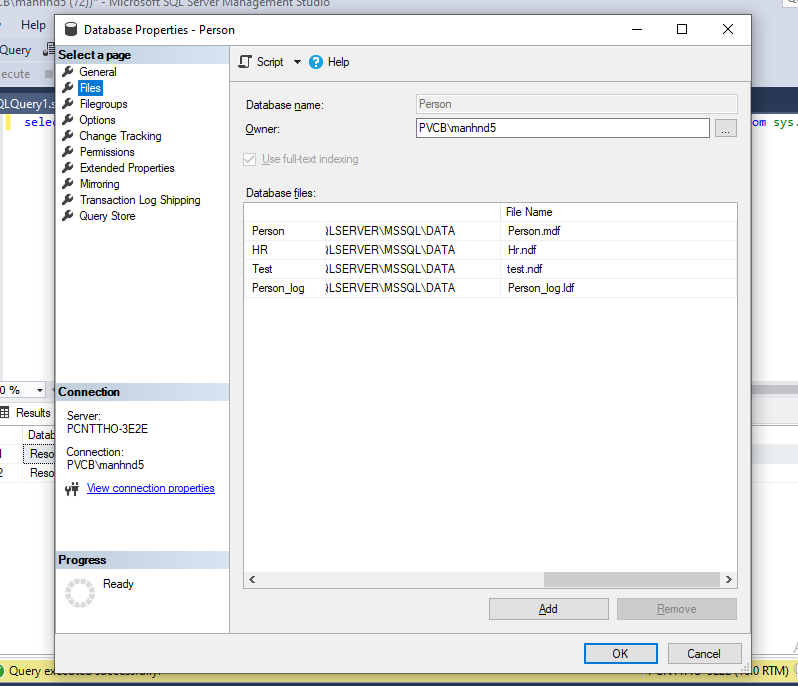
* Tạo filegroup: chuột phải vào db muốn tạo filegroup chọn properties > FilesGroups > Add Filegroup > Điền tên > OK







* Tạo file ndf và gắn vào filegroup: Chuột phải vào db > Properties > Files > Add > Điền tên ở cột Logical Name, chọn FileGroups ở cột Filegroup và đặt tên file ở cột File Name (nhớ có đuôi .ndf) > Ok



* Khi tạo bảng chỉ định file ndf để lưu: thêm on FILEGROUP\_NAME
  + Ví dụ: Create table test (id int, name varchar) on TEST

##### Partition

###### Định nghĩa

Partition trong SQL là quá trình chia dữ liệu của một bảng hoặc chỉ mục thành các phần nhỏ hơn, gọi là các partition, dựa trên một hoặc nhiều cột hoặc biểu thức. Mỗi partition có thể được lưu trữ và quản lý độc lập, giúp tăng hiệu suất truy vấn, quản lý dữ liệu và thực hiện các thao tác bảo trì.

Bạn hãy hình dung: ngôi nhà của bạn có một TỦ SÁCH và nhiệm vụ của bạn cần thiết kế làm sao khi chúng ta tìm kiếm thông tin một quyển sách và lấy ra được quyển sách đó với thời gian ngắn nhất.

Thông thường sẽ có 2 cách:

* Cách thứ nhất: Các quyển sách được xếp chồng lên nhau một cách lộn xộn, không có bất kỳ nguyên tắc, quy củ gì cả.
* Cách thứ hai: Chúng ta chia nhỏ tủ sách thành các ô riêng biệt, mỗi ô sẽ chưa một loại sách đặc thù (sách văn học, sách kinh tế, sách lịch sử…)

Nếu bây giờ chúng ta cần tìm tất cả quyển sách kinh tế và có tên bắt đầu bằng chữ H.

Bạn hãy tưởng tượng với hai cách thiết kế tủ sách bên trên, cách nào sẽ đem lại thời gian tìm kiếm nhanh hơn nhé.

* Cách 1: Quyển sách mà bạn tìm có thể ở bất kỳ chỗ nào trong đống “hổ lốn” sách
* Cách 2: Quyển sách mà bạn tìm chắc chắn chỉ ở trong “ô sách được dán nhãn Kinh tế”

Đây chính là ý tưởng chính của kỹ thuật Partition, và việc làm tăng hiệu năng câu lệnh, bản chất là do “số lượng block cần tìm kiếm” đã được khoanh vùng, do đó khối lượng cần thực hiện của câu lệnh đã giảm đi rất nhiều.

###### Ý nghĩa và tầm quan trọng

* **Tăng hiệu suất truy vấn:** Partition cho phép cơ sở dữ liệu thực hiện các thao tác truy vấn trên các phần nhỏ của dữ liệu, giúp giảm thời gian thực thi và tăng hiệu suất.
* **Quản lý dữ liệu hiệu quả:** Bằng cách chia dữ liệu thành các partition, bạn có thể dễ dàng quản lý và duy trì dữ liệu, đặc biệt là khi làm việc với các bảng có dung lượng lớn.
* **Tối ưu hóa bảo trì:** Partition cung cấp khả năng thực hiện các thao tác bảo trì như sao lưu, phục hồi và chỉnh sửa dữ liệu một cách linh hoạt và hiệu quả hơn.

###### Các loại partition và cú pháp

* **Partitioning theo Range (Range Partitioning):**
  + Dữ liệu được chia thành các partition dựa trên giá trị của một cột trong bảng. Mỗi partition đại diện cho một phạm vi giá trị.
  + Ví dụ: Chia dữ liệu của một bảng Orders thành các partition dựa trên ngày đặt hàng.

CREATE TABLE Orders (

OrderID INT,

OrderDate DATE,

-- Other columns

)

PARTITION BY RANGE (YEAR(OrderDate)) (

PARTITION p1 VALUES LESS THAN (2010),

PARTITION p2 VALUES LESS THAN (2020),

PARTITION p3 VALUES LESS THAN MAXVALUE

);

* **Partitioning theo Hash (Hash Partitioning):**
  + Dữ liệu được chia thành các partition dựa trên giá trị được băm của một hoặc nhiều cột.
  + Phân phối dữ liệu đồng đều giữa các partition, không cần phải quan tâm đến giá trị cụ thể.
  + Ví dụ: Chia dữ liệu của một bảng Users thành các partition dựa trên giá trị băm của cột UserID.

CREATE TABLE Users (

UserID INT,

Username VARCHAR(50),

-- Other columns

)

PARTITION BY HASH(UserID) (

PARTITION p1,

PARTITION p2,

PARTITION p3

);

* **Partitioning theo List (List Partitioning):**
  + Dữ liệu được chia thành các partition dựa trên giá trị cụ thể của một hoặc nhiều cột.
  + Mỗi partition được gán một tập hợp các giá trị cụ thể.
  + Ví dụ: Chia dữ liệu của một bảng Products thành các partition dựa trên danh sách các danh mục sản phẩm.

CREATE TABLE Products (

ProductID INT,

ProductName VARCHAR(100),

Category VARCHAR(50),

-- Other columns

)

PARTITION BY LIST(Category) (

PARTITION pElectronics VALUES IN ('Phone', 'Laptop', 'Tablet'),

PARTITION pClothing VALUES IN ('Shirt', 'Pants', 'Dress'),

PARTITION pOther VALUES IN (DEFAULT)

);

* **Partitioning theo Cột (Column Partitioning):**
  + Dữ liệu được chia thành các partition dựa trên giá trị của một cột đặc biệt, thường là một cột khóa.
  + Mỗi partition được gán cho một phạm vi giá trị hoặc một tập hợp giá trị.
  + Ví dụ: Chia dữ liệu của một bảng Customers thành các partition dựa trên giá trị của cột CustomerID.

CREATE TABLE Customers (

CustomerID INT,

CustomerName VARCHAR(100),

-- Other columns

)

PARTITION BY COLUMN (CustomerID) (

PARTITION p1 VALUES LESS THAN (1000),

PARTITION p2 VALUES LESS THAN (2000),

PARTITION p3 VALUES LESS THAN MAXVALUE

);

###### Bảng như thế nào thì nên cân nhắc thiết kế dạng Partition?

Các table có dung lượng lớn hơn 2GB, chúng ta đều nên cân nhắc thực hiện partition.

Ngoài ra, một yếu tố vô cùng quan trọng mà bạn phải có trước khi quyết định áp dụng kỹ thuật partition: **Phải chọn được PARTITION KEY.**

Việc lựa chọn **PARTITION KEY chính là yếu tố then chốt mang lại hiệu quả của kỹ thuật Partition**.

Hãy quay lại ví dụ tìm kiếm sách trong tủ sách bên trên. Chúng ta quyết định chia tủ sách ban đầu thành các ô riêng biệt theo “Loại sách”, tại đây “Loại sách” chính là “PARTITION KEY”.

**Ví dụ về đánh partition không hiệu quả**: Ta đi tìm sách kinh tế nhưng phân biệt các ô theo “Năm xuất bản”. Quyển sách KINH TẾ có thể nằm trong bất kỳ một ngăn tủ nào bên trên:

* Sẽ có quyển sách KINH TẾ xuất bản vào năm 2013, 2014, cũng có thể có quyển sách xuất bản vào năm 2022

Như vậy trong trường hợp này, việc bạn phân chia sách theo năm xuất bản KHÔNG ĐEM LẠI bất kỳ giá trị nào về mặt hiệu năng cả. Ngược lại công sức tìm kiếm thực tế CÒN TĂNG LÊN.

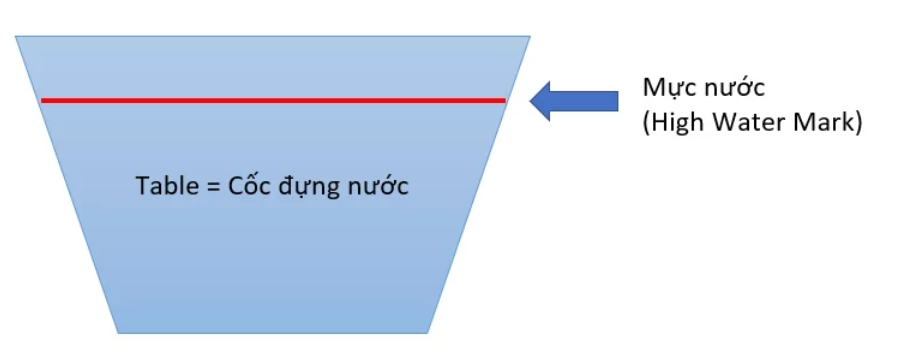
##### High Water Mark

###### Định nghĩa

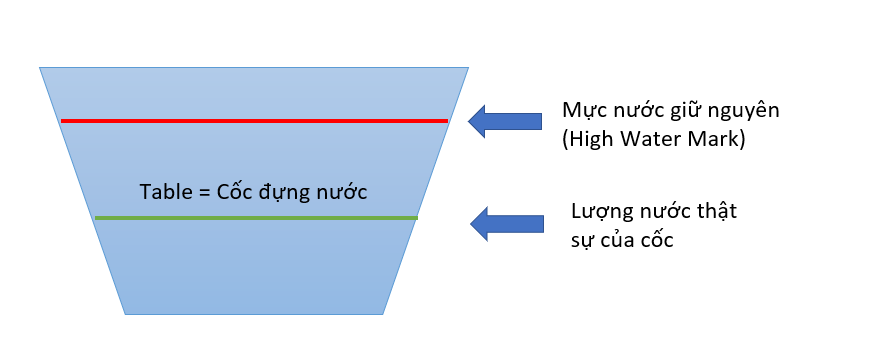
Bản chất của việc Cơ sở dữ liệu thực hiện lấy dữ liệu không phải dựa vào số lượng bản ghi.

Bạn có thể hình dung các Table giống như những cốc đựng nước.

Hành động thêm dữ liệu vào Table, cũng giống như chúng ta đổ nước vào trong cốc.



Bây giờ nếu chúng ta muốn đổ bớt nước trong cốc ra ngoài, chuyện gì sẽ xảy ra với “Mực nước” đã được đánh dấu ở phía trước?



Tại trường hợp này:

* Mặc dù lượng nước thật sự của cốc đã giảm đi, tuy nhiên “Mốc đánh dấu” ban đầu thì không hề giảm (High Water Mark vẫn còn như cũ)

Vấn đề này cũng giống hệt như việc khi bạn thực hiện lệnh DELETE từ TABLE:

* Số lượng bản ghi trong bảng giảm đi
* High water mark của phần lưu trữ thì không hề giảm

Khi cơ sở dữ liệu tìm kiếm thông tin, nó cần tìm toàn bộ vùng dữ liệu từ block đầu tiên cho tới block được đánh dấu bởi high water mark.

Chính vì vây mặc dù số lượng bản ghi của bảng cực kỳ nhỏ (0 bản ghi), nhưng nếu High Water Mark rất lớn thì câu lệnh vẫn chạy cực kỳ lâu.

###### Kiểm tra thông tin High Water Mark của một bảng

Trong Oracle:

SET SERVEROUT ON

DECLARE

CURSOR cu\_tables IS

  SELECT a.owner,

         a.table\_name

  FROM   dba\_tables a

  WHERE  a.table\_name = '&TABLE\_NAME'

  AND    a.owner      = '&OWNER';

op1  NUMBER;

op2  NUMBER;

op3  NUMBER;

op4  NUMBER;

op5  NUMBER;

op6  NUMBER;

op7  NUMBER;

BEGIN

Dbms\_Output.Enable(1000000);

Dbms\_Output.Put\_Line('TABLE                             UNUSED BLOCKS     TOTAL BLOCKS  HIGH WATER MARK');

Dbms\_Output.Put\_Line('------------------------------  ---------------  ---------------  ---------------');

FOR cur\_rec IN cu\_tables LOOP

  Dbms\_Space.Unused\_Space(cur\_rec.owner,cur\_rec.table\_name,'TABLE',op1,op2,op3,op4,op5,op6,op7);

  Dbms\_Output.Put\_Line(RPad(cur\_rec.table\_name,30,' ') ||

                       LPad(op3,15,' ')                ||

                       LPad(op1,15,' ')                ||

                       LPad(Trunc(op1-op3-1),15,' '));

END LOOP;

END;

* Trong đoạn lệnh sử dụng 2 biến:
* Biến &TABLE\_NAME: bạn nhập tên của Table muốn kiểm tra (nhớ là nhập IN HOA giá trị này)
* Biến &OWNER: bạn nhập tên của uesr sở hữu table (nhớ là nhập IN HOA giá trị này)

Nhập giá trị cần tìm kiếm như sau:

Enter value for table\_name: WECOMMIT\_TEST

old   6:     WHERE  a.table\_name = '&TABLE\_NAME'

new   6:     WHERE  a.table\_name = 'WECOMMIT\_TEST'

Enter value for owner: HUYTQ

old   7:     AND    a.owner = '&OWNER';

new   7:     AND    a.owner = 'HUYTQ';

* Kết quả

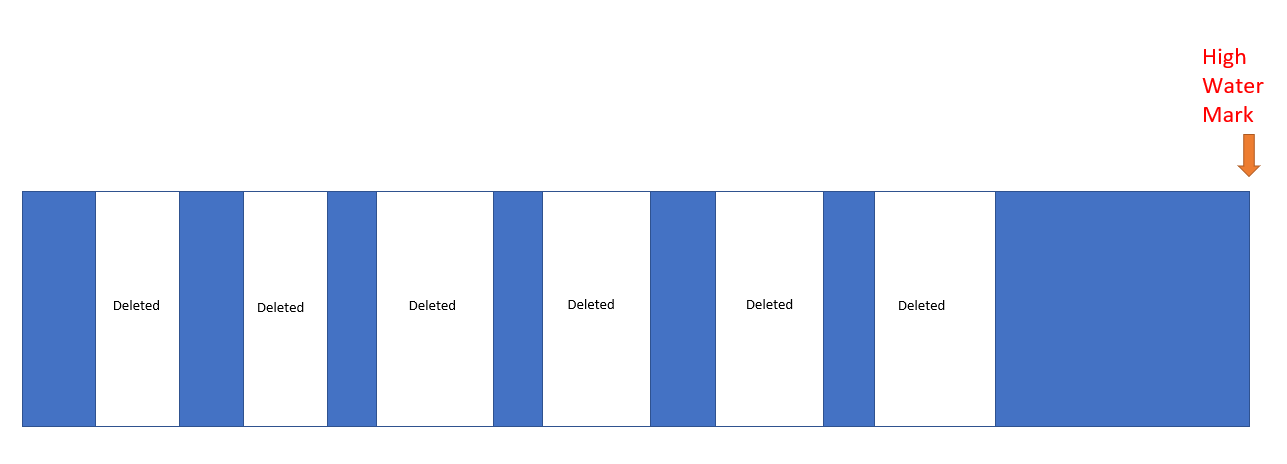
**TABLE                      UNUSED  BLOCKS     TOTAL BLOCKS    HIGH WATER MARK**

**WECOMMIT\_TEST           238                                22528                  22289**

**Ghi chú:**

* Một block dữ liệu có thể chứa nhiều bản ghi, do đó số block chứa dữ liệu sẽ khác với số lượng bản ghi!

Hình ảnh sử dụng block segment của table như sau:



Theo kết quả phân tích bên trên:

* Tổng số block dữ liệu màu trắng (dữ liệu đã bị xóa): chiếm 238 Block
* Tổng số dữ liệu màu xanh: 22528 block
* High Water Mark đánh dấu của table: 22289.

###### Việc hiểu về High Water Mark có thể giúp bạn tối ưu trong thực tế ra sao

Do trong quá trình Insert dữ liệu, cơ sở dữ liệu sẽ thực hiện đưa dữ liệu lấp đầy các khoảng trống bên dưới High Water Mark (việc này sẽ khiến công việc bị chậm), trong những hệ thống yêu cầu việc Insert dữ liệu nhanh chóng, chúng ta có kỹ thuật bắt buộc tiến trình sẽ Insert ở phía bên trên High Water Mark, do đó tăng tốc đáng kể các job Insert.

Với những Table bị phân mảnh quá nhiều, chúng ta có thể thực hiện tối ưu lại, giúp dữ liệu được sắp xếp liền mạch hơn, từ đó giảm thiểu số lượng block cần phải thực hiện khi SELECT.

##### Hiện tượng bảng bị phân mảnh (Defragment Table | Shrink Table)

###### Định nghĩa

Phân mảnh (Fragmentation) xảy ra khi dữ liệu trong một bảng không được lưu trữ liên tục trên các trang và cụm (extents) của đĩa cứng. Điều này có thể xảy ra do các hoạt động thêm, xóa, cập nhật dữ liệu trong bảng. Khi dữ liệu bị phân tán trên nhiều vùng khác nhau của đĩa cứng, việc truy xuất dữ liệu sẽ trở nên chậm hơn vì đầu đọc đĩa phải di chuyển nhiều lần để tìm và đọc các phần dữ liệu rời rạc.

Các bạn có thể hình dùng Table của bạn giống như một tòa chung cư. Các dữ liệu của bạn giống như những người dân sống trong chung cư đó.

Lúc đầu bạn đưa dữ liệu vào bảng, giống như việc những người dân ồ ạt kéo đến ở trong tòa chung cư.

Lúc ban đầu chung cư rất đông vui nhộn nhịp, rất cả các phòng đều có người ở (dữ liệu được đưa vào các block). Hình ảnh dưới đây chính là chung cư của các bạn: có 14 tầng, mỗi tầng có 18 căn hộ (mỗi căn hộ chính là 1 block chứa dữ liệu đó).

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tầng 14 | P1401 | P1402 | P1403 | P1404 | P1405 | P1406 | P1407 | P1408 |
| Tầng 13 | P1301 | P1302 | P1303 | P1304 | P1305 | P1306 | P1307 | P1308 |
| Tầng 12 | P1201 | P1202 | P1203 | P1204 | P1205 | P1206 | P1207 | P1208 |
| Tầng 11 | P1101 | P1102 | P1103 | P1104 | P1105 | P1106 | P1107 | P1108 |
| Tầng 10 | P1001 | P1002 | P1003 | P1004 | P1005 | P1006 | P1007 | P1008 |
| Tầng 9 | P901 | P902 | P903 | P904 | P905 | P906 | P907 | P908 |
| Tầng 8 | P801 | P802 | P803 | P804 | P805 | P806 | P807 | P808 |
| Tầng 7 | P701 | P702 | P703 | P704 | P705 | P706 | P707 | P708 |
| Tầng 6 | P601 | P602 | P603 | P604 | P605 | P606 | P607 | P608 |
| Tầng 5 | P501 | P502 | P503 | P504 | P505 | P506 | P507 | P508 |
| Tầng 4 | P401 | P402 | P403 | P404 | P405 | P406 | P407 | P408 |
| Tầng 3 | P301 | P302 | P303 | P304 | P305 | P306 | P307 | P308 |
| Tầng 2 | P201 | P202 | P203 | P204 | P205 | P206 | P207 | P208 |
| Tầng 1 | P101 | P102 | P103 | P104 | P105 | P106 | P107 | P108 |

Một ngày nọ, chung cư buộc phải di dời người dân ra địa điểm khác (vì lý do quy hoạch), tất cả mọi người đều rời khỏi căn hộ của mình, trừ ông Mạnh ND vẫn bám trụ ở lại.

Hình ảnh căn hộ của bạn lúc này như sau:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tầng 14 | P1401 | P1402 | P1403 | P1404 | P1405 | P1406 | P1407 | P1408 |
| Tầng 13 | P1301 | P1302 | P1303 | P1304 | P1305 | P1306 | P1307 | P1308 |
| Tầng 12 | P1201 | P1202 | P1203 | P1204 | P1205 | P1206 | P1207 | P1208 |
| Tầng 11 | P1101 | P1102 | P1103 | P1104 | P1105 | **P1106** | P1107 | P1108 |
| Tầng 10 | P1001 | P1002 | P1003 | P1004 | P1005 | P1006 | P1007 | P1008 |
| Tầng 9 | P901 | P902 | P903 | P904 | P905 | P906 | P907 | P908 |
| Tầng 8 | P801 | P802 | P803 | P804 | P805 | P806 | P807 | P808 |
| Tầng 7 | P701 | P702 | P703 | P704 | P705 | P706 | P707 | P708 |
| Tầng 6 | P601 | P602 | P603 | P604 | P605 | P606 | P607 | P608 |
| Tầng 5 | P501 | P502 | P503 | P504 | P505 | P506 | P507 | P508 |
| Tầng 4 | P401 | P402 | P403 | P404 | P405 | P406 | P407 | P408 |
| Tầng 3 | P301 | P302 | P303 | P304 | P305 | P306 | P307 | P308 |
| Tầng 2 | P201 | P202 | P203 | P204 | P205 | P206 | P207 | P208 |
| Tầng 1 | P101 | P102 | P103 | P104 | P105 | P106 | P107 | P108 |

Các căn hộ đánh dấu màu đen là những căn hộ trống.

Căn hộ được bôi màu đỏ (phòng 1106) là căn hộ có người ở (ông Mạnh ND vẫn ở đó).

Hiện tượng này tương đương với việc rất nhiều block dữ liệu đang không được sử dung, nhưng không được thu hồi (hiện tượng phân mảnh của bảng).

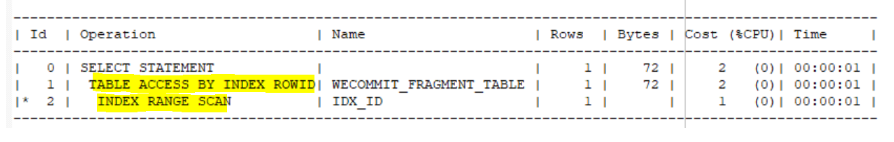
Bây giờ, chúng ta xét 2 trường hợp

Trường hợp 1: Chung cư này không có danh sách thông tin từng người dân đang ở căn hộ số bao nhiêu. Nếu bạn cần đi tìm ông Mạnh ND để yêu cầu ông ấy di dời khỏi tòa chung cư thì phải làm thế nào?

* Do bạn không biết chính xác ông ấy đang ở phòng số bao nhiêu, do đó bạn sẽ buộc phải đi từng tầng, đối với mỗi tầng sẽ ghé qua từng phòng để tìm. Việc này chính là giải thuật FULL TABLE SCAN (quét toàn bộ block dữ liệu).

Trường hợp 2: Nếu ban quản lý của chung cư ấy có một danh sách cho biết thông tin của người dân và căn hộ tương ứng thì sao.

* Để cần tìm ông Mạnh ND, bạn sẽ phải mở danh sách đó ra và tra xem anh ta đang ở phòng số bao nhiêu. Tại đây bạn biết được anh ấy ở tầng 11, phòng 1106. Bước này tương ứng với giải thuật INDEX RANGE SCAN để quét thông tin ROWID- số phòng)
* Bước số 2: Đi tới phòng 1106 để gặp ông Mạnh ND. Bước này chính là bước TABLE ACCES BY INDEX ROWID



Như vậy, không phải lúc nào bảng bị phân mảnh nhiều thì cũng bị chậm, treo.

###### Câu lệnh kiểm tra xem bảng có bị phân mảnh hay không?

SQL server

SELECT

DB\_NAME(database\_id) AS 'Tên database',

OBJECT\_NAME(object\_id) AS 'Tên bảng',

index\_id,

index\_type\_desc,

avg\_fragmentation\_in\_percent As 'Tỉ lệ bị phân mảnh (%s)'

FROM sys.dm\_db\_index\_physical\_stats(DB\_ID(), NULL, NULL, NULL, 'LIMITED')

WHERE avg\_fragmentation\_in\_percent > 30 -- Ngưỡng phân mảnh cao

And index\_id = 0 -- Chỉ lấy ra bảng bảng/ index\_id > 0 : lấy ra index

ORDER BY avg\_fragmentation\_in\_percent DESC;

Oracle

**Bước 1:** Thực hiện cập nhật lại thông tin thống kê của bảng

SQL> EXEC DBMS\_STATS.gather\_table\_stats('HUYTQ', 'WECOMMIT\_FRAGMENT\_TABLE', estimate\_percent => 100, cascade => TRUE);

PL/SQL procedure successfully completed.

**Bước 2:**  Câu lệnh kiểm tra chi tiết tình trạng phân mảnh

 SELECT owner,

        table\_name,

        ROUND ((blocks \* 8) / 1024, 2)

            "Fragmented size (mb)",

        ROUND ((num\_rows \* avg\_row\_len / 1024 / 1024), 2)

            "Actual size (mb)",

          ROUND ((blocks \* 8) / 1024, 2)

        - ROUND ((num\_rows \* avg\_row\_len / 1024 / 1024), 2)

            "Reclaim size (mb)",

            (  (  ROUND ((blocks \* 8 / 1024), 2)

                - ROUND ((num\_rows \* avg\_row\_len / 1024 / 1024), 2))

             / ROUND ((blocks \* 8 / 1024), 2))

          \* 100

        - 10

            "Reclaimable space % "

   FROM dba\_tables

  WHERE     table\_name = '&TABLE\_NAME'

        AND ROUND ((blocks \* 8 / 1024), 2) > 0

        AND     (  (  ROUND ((blocks \* 8 / 1024), 2)

                    - ROUND ((num\_rows \* avg\_row\_len / 1024 / 1024), 2))

                 / ROUND ((blocks \* 8 / 1024), 2))

              \* 100

            - 10 >

            10

        AND   ROUND ((blocks \* 8) / 1024, 2)

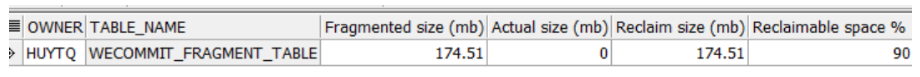
            - ROUND ((num\_rows \* avg\_row\_len / 1024 / 1024), 2) >

            100

        AND owner NOT IN ('SYS')

ORDER BY 1, 6 DESC;

Enter value for table\_name: WECOMMIT\_FRAGMENT\_TABLE

**Kết quả cho thấy bảng đã bị phân mảnh hơn 90%**

#### Chỉ mục (Index)

##### Định nghĩa

Trong cơ sở dữ liệu, index là một cấu trúc dữ liệu đặc biệt được sử dụng để nhanh chóng tìm kiếm, truy cập và sắp xếp dữ liệu trong bảng. Index giúp cải thiện hiệu suất của các truy vấn SQL bằng cách giảm thời gian tìm kiếm và sắp xếp dữ liệu.

Hình dung table là quyển sách thì index là mục lục.

##### Các loại index

###### Theo Cấu trúc Dữ liệu Lưu Trữ

B-tree Index

* Định nghĩa: Sử dụng cấu trúc dữ liệu cây nhị phân cân bằng (B-tree) để lưu trữ các giá trị khóa đã sắp xếp.
* Cú pháp tạo (SQL Server):

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index\_name

ON table\_name (column1 [ASC | DESC] [, column2, ...]);

* Cách sử dụng: Được sử dụng rộng rãi cho hầu hết các truy vấn có điều kiện dựa trên phạm vi giá trị hoặc điểm truy nhập.
* Khi nào nên sử dụng: Phù hợp cho các cột thường xuyên được truy vấn, sắp xếp hoặc tham gia vào bắt cặp.
* Lưu ý: Việc thêm/xóa hàng trong bảng sẽ chậm hơn do phải duy trì cấu trúc B-tree.

Bitmap Index

* Định nghĩa: Lưu trữ bit vector thể hiện sự hiện diện/vắng mặt của mỗi giá trị duy nhất trong một cột.
* Cú pháp tạo (Oracle):

CREATE BITMAP INDEX index\_name

ON table\_name (column1 [, column2, ...]);

* Cách sử dụng: Thích hợp cho các truy vấn có điều kiện OR, đặc biệt với số lượng hàng lớn.
* Khi nào nên sử dụng: Phù hợp cho các cột có số lượng giá trị khác nhau không quá lớn và có nhiều giá trị trùng lặp.
* Lưu ý: Không hiệu quả cho các truy vấn dựa trên phạm vi giá trị do không đảm bảo thứ tự.

Hash Index

* Định nghĩa: Sử dụng hàm băm (hash function) để ánh xạ các giá trị khóa thành các vị trí lưu trữ vật lý.
* Cú pháp tạo (Oracle):

CREATE [UNIQUE] HASH INDEX index\_name

ON table\_name (column1 [, column2, ...])

[HASH IS expr [, expr ...]] [PARTITION ...];

* Cách sử dụng: Hiệu quả cho truy vấn điểm trực tiếp, không yêu cầu sắp xếp.
* Khi nào nên sử dụng: Phù hợp cho các truy vấn tìm kiếm nhanh trên các cột có số lượng giá trị lớn và khác nhau.
* Lưu ý: Không hiệu quả cho truy vấn dựa trên phạm vi, không đảm bảo thứ tự. Có thể gặp va chạm (collision).

###### Theo Đối tượng Index

Cột Index

Clustered Index

* Định nghĩa: Một loại index đặc biệt trong đó các hàng trong bảng được sắp xếp và lưu trữ trên đĩa dựa trên khóa clustered index.
* Cú pháp tạo (SQL Server):

CREATE CLUSTERED INDEX index\_name

ON table\_name (column1 [ASC | DESC] [, column2, ...]);

* Cách sử dụng: Được sử dụng làm index chính cho bảng, đảm bảo hiệu suất cao cho các truy vấn phổ biến.
* Khi nào nên sử dụng: Nên tạo clustered index trên cột/cột có tính duy nhất cao và thường được truy vấn.
* Lưu ý: Mỗi bảng chỉ có thể có một clustered index, việc thay đổi clustered index có thể chậm.

Non-Clustered Index

* Định nghĩa: Một loại index phổ biến, trong đó các mục lục (điểm nhập) được lưu trữ riêng biệt với các hàng dữ liệu.
* Cú pháp tạo:

CREATE NONCLUSTERED INDEX index\_name

ON table\_name (column1 [ASC | DESC] [, column2, ...]);

* Cách sử dụng: Được sử dụng phổ biến để nâng cao hiệu suất truy vấn trên các cột không được đưa vào clustered index.
* Khi nào nên sử dụng: Nên tạo non-clustered index cho các cột thường xuyên được truy vấn nhưng không phải là clustered index.
* Lưu ý: Mỗi bảng có thể có nhiều non-clustered index, nhưng quá nhiều có thể làm chậm các hoạt động thêm/xóa/sửa.

Unique Index

* Định nghĩa: Một loại index đảm bảo rằng các giá trị trong cột hoặc tập hợp các cột được chỉ định là duy nhất.
* Cú pháp tạo:

CREATE UNIQUE [CLUSTERED | NONCLUSTERED] INDEX index\_name

ON table\_name (column1 [ASC | DESC] [, column2, ...]);

* Cách sử dụng: Được sử dụng để đảm bảo tính duy nhất của dữ liệu và nâng cao hiệu suất truy vấn.
* Khi nào nên sử dụng: Nên tạo unique index trên các cột hoặc tập hợp cột có yêu cầu tính duy nhất.
* Lưu ý: Unique index có thể được tạo dựa trên clustered hoặc non-clustered index.

Composite Index

* Định nghĩa: Một index tạo trên nhiều hơn một cột trong bảng.
* Cú pháp tạo: Tương tự như B-tree index, chỉ cần liệt kê nhiều cột hơn.
* Cách sử dụng: Được sử dụng để nâng cao hiệu suất cho các truy vấn có điều kiện trên nhiều cột.
* Khi nào nên sử dụng: Nên tạo composite index khi các truy vấn thường xuyên sử dụng nhiều cột trong điều kiện WHERE, JOIN hoặc ORDER BY.
* **Thay đổi thứ tự các cột trong index có ảnh hưởng tới hiệu năng?**
  + Nếu trong điều kiện where có đủ các cột trong index thì không quan trọng thứ tự.
  + Nếu có ít cột hơn thì sẽ có sự khác biệt về hiệu năng => Phải có chiến lược chọn index
  + Khuyến nghị : nên sắp xếp theo thứ tự quan trọng giảm dần.
* **Nếu chỉ đánh index trên một cột (mổ cò) và đánh nhiều index thì được không?**
  + Nhiều index thì hiệu năng kém => giảm tốc độ ghi và chưa chắc tăng thời gian truy xuất

Covered Index

* Định nghĩa: Một loại non-clustered index đặc biệt trong đó tất cả các cột cần thiết để truy vấn được lưu trữ trong chính index.
* Cách sử dụng: Được sử dụng để tránh truy cập vào bảng dữ liệu, nâng cao hiệu suất truy vấn.
* Khi nào nên sử dụng: Nên tạo covered index cho các truy vấn phức tạp, tham chiếu nhiều cột từ nhiều bảng khác nhau.
* Lưu ý: Covered index có thể chiếm nhiều không gian hơn so với non-clustered index thông thường.

Dữ liệu Đặc Biệt Index

Full-Text Index

* Định nghĩa: Được sử dụng để tìm kiếm văn bản tự do trong các cột dữ liệu có kích thước lớn như văn bản hay tài liệu.
* Cú pháp tạo (SQL Server):

CREATE FULLTEXT INDEX ON table\_name

(

column1 [LANGUAGE language\_term] [,...n]

)

KEY INDEX key\_index\_name

[WITH <fulltext\_index\_option> [,...n]]

* Cách sử dụng: Cho phép tìm kiếm toàn văn bản, tương tự như công cụ tìm kiếm trực tuyến.
* Khi nào nên sử dụng: Nên tạo full-text index cho các cột chứa dữ liệu văn bản lớn và cần tìm kiếm theo từ khóa.
* Lưu ý: Full-text index không hiệu quả cho các truy vấn điểm hoặc phạm vi giá trị.

Spatial Index

* Định nghĩa: Được sử dụng để tăng tốc các truy vấn liên quan đến dữ liệu không gian như địa lý hoặc hình học.
* Cú pháp tạo (SQL Server):

CREATE SPATIAL INDEX index\_name

ON table\_name (column\_name)

[USING <geography\_or\_geometry\_data\_type>]

[WITH <spatial\_index\_option> [,...n]]

* Cách sử dụng: Hỗ trợ các truy vấn không gian, chẳng hạn như tìm các điểm trong một khu vực nhất định hoặc kiểm tra sự giao nhau của các đối tượng không gian.
* Khi nào nên sử dụng: Nên tạo spatial index cho các cột chứa dữ liệu địa lý hoặc hình học và cần thực hiện các phép toán không gian.
* Lưu ý: Spatial index có thể chiếm nhiều không gian lưu trữ và yêu cầu quản lý phức tạp hơn.

XML Index

* Định nghĩa: Được sử dụng để tăng tốc truy vấn trên dữ liệu XML.
* Cú pháp tạo (SQL Server):

CREATE XML INDEX index\_name

ON table\_name (xml\_column)

[USING XML INDEX xml\_index\_option [,...n]]

* Cách sử dụng: Hỗ trợ các truy vấn và phép toán trên dữ liệu XML, chẳng hạn như tìm kiếm, điều kiện, trích xuất.
* Khi nào nên sử dụng: Nên tạo XML index cho các cột chứa dữ liệu XML và cần thực hiện các truy vấn phức tạp trên dữ liệu đó.
* Lưu ý: XML index có thể chiếm nhiều không gian lưu trữ và yêu cầu quản lý phức tạp hơn.

Hàm Index

Function-based Index

* Định nghĩa: Một loại index được tạo không chỉ trên cột mà còn trên kết quả của một hàm tính toán trên một hoặc nhiều cột.
* Cú pháp tạo (SQL Server):

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index\_name

ON table\_name (function\_expression)

[INCLUDE (column\_name [,...n])]

* Cách sử dụng: Hỗ trợ tối ưu hóa các truy vấn dựa trên kết quả của hàm ứng dụng trên dữ liệu.
* Khi nào nên sử dụng: Nên tạo function-based index khi các truy vấn thường xuyên sử dụng hàm tính toán trên dữ liệu, chẳng hạn như UPPER, REVERSE, SUBSTRING.
* Lưu ý: Function-based index có thể chiếm nhiều không gian lưu trữ hơn và không thể được tạo trên các hàm không phù hợp.

###### Theo Cách Sắp xếp

Reverse Index (Key Reverse Index)

* Định nghĩa: Lưu trữ giá trị của khóa theo thứ tự đảo ngược, giúp tối ưu hóa các truy vấn có ORDER BY DESC.
* Cách sử dụng: Được sử dụng để nâng cao hiệu suất cho các truy vấn có sắp xếp giảm dần trên cột được đảo ngược.
* Khi nào nên sử dụng: Nên tạo reverse index khi có nhiều truy vấn sử dụng ORDER BY DESC trên một cột nhất định.
* Lưu ý: Reverse index chỉ phục vụ mục đích sắp xếp, không hỗ trợ tìm kiếm dựa trên phạm vi giá trị.

Descending Index:

* Định nghĩa: Sắp xếp các giá trị khóa theo thứ tự giảm dần thay vì tăng dần.
* Cách sử dụng: Được sử dụng để nâng cao hiệu suất cho các truy vấn có sắp xếp giảm dần trên cột được index.
* Khi nào nên sử dụng: Nên tạo descending index khi có nhiều truy vấn sử dụng ORDER BY DESC trên một cột nhất định.
* Lưu ý: Descending index cũng hỗ trợ tìm kiếm dựa trên phạm vi giá trị, nhưng phạm vi sẽ đảo ngược so với thứ tự tăng dần.

###### Theo phạm vi áp dụng

Filtered Index

* Định nghĩa: Một loại index chỉ bao gồm các hàng nào thỏa mãn một điều kiện lọc nhất định, làm giảm kích thước index và tăng hiệu suất cho các truy vấn áp dụng điều kiện lọc tương tự.
* Cú pháp tạo (SQL Server):

CREATE UNIQUE NONCLUSTERED INDEX index\_name

ON table\_name (column1, column2)

WHERE column3 > value;

* Cách sử dụng: Được sử dụng để tối ưu hóa hiệu suất cho các truy vấn có điều kiện lọc tương tự.

Khi nào nên sử dụng: Nên tạo filtered index khi có nhiều truy vấn sử dụng cùng một điều kiện lọc và dữ liệu thỏa mãn điều kiện đó chiếm tỷ lệ nhỏ trong bảng.

* Lưu ý: Filtered index chỉ hữu ích cho các truy vấn có điều kiện tương tự như điều kiện lọc được sử dụng khi tạo index.

Partial Index

* Định nghĩa: Tương tự như filtered index, chỉ bao gồm một tập con hàng dựa trên một điều kiện hoặc biểu thức nhất định.
* Cú pháp:

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index\_name

ON table\_name (column1 [ASC | DESC] [, column2, ...])

WHERE column\_filter\_expression

[WITH (partition\_option\_data)]

* Cách sử dụng: Được sử dụng để nâng cao hiệu suất cho các truy vấn chỉ làm việc với một phần dữ liệu trong bảng.
* Khi nào nên sử dụng: Nên tạo partial index khi có nhiều truy vấn chỉ làm việc với một phần dữ liệu trong bảng và phần dữ liệu đó chiếm tỷ lệ nhỏ.
* Lưu ý: Partial index chỉ hữu ích cho các truy vấn làm việc với phần dữ liệu được bao gồm trong index.

###### Theo tổ chức dữ liệu

Index-Organized Table (IOT):

* Định nghĩa: Một kỹ thuật đặc biệt trong Oracle, nơi dữ liệu được lưu trữ theo thứ tự index thay vì cấu trúc bảng riêng biệt.
* Cách sử dụng: Được sử dụng để tối ưu hóa hiệu suất cho các truy vấn thường xuyên trên các cột được index, vì dữ liệu đã được sắp xếp sẵn.
* Khi nào nên sử dụng: Nên sử dụng IOT khi có nhiều truy vấn thường xuyên trên một số cột nhất định và yêu cầu hiệu suất cao.
* Lưu ý: IOT yêu cầu quản lý và bảo trì phức tạp hơn so với bảng thông thường, và không phù hợp cho các trường hợp có nhiều cập nhật dữ liệu.

###### Theo không gian lưu trữ:

Compressed Index

* Định nghĩa: Các giá trị index được nén để tiết kiệm không gian lưu trữ đĩa, nhưng có thể làm giảm hiệu suất một chút.
* Cách sử dụng: Được sử dụng để tiết kiệm không gian lưu trữ cho các index lớn.
* Khi nào nên sử dụng: Nên sử dụng compressed index khi không gian lưu trữ là vấn đề quan trọng và hiệu suất không phải là ưu tiên hàng đầu.
* Lưu ý: Compressed index có thể làm giảm hiệu suất truy vấn một chút do quá trình giải nén, nhưng tiết kiệm đáng kể không gian lưu trữ.

###### Theo Quản lý Phân vùng

Partitioned Index

* Định nghĩa: Index được chia thành nhiều phân vùng dựa trên một hoặc nhiều cột khóa, giúp tăng hiệu suất cho các truy vấn trên một phân vùng cụ thể.
* Cú pháp:

CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index\_name

ON table\_name (column1 [ASC | DESC] [, column2, ...])

ON partition\_scheme\_name (column)

[WITH (partition\_option\_data)]

* Cách sử dụng: Được sử dụng để tối ưu hóa hiệu suất cho các truy vấn chỉ làm việc với một phần dữ liệu trong bảng.
* Khi nào nên sử dụng: Nên tạo partitioned index khi có nhiều truy vấn chỉ làm việc với một phần dữ liệu trong bảng và phần dữ liệu đó có thể được phân vùng dựa trên một hoặc nhiều cột khóa.
* Lưu ý: Partitioned index yêu cầu quản lý và bảo trì phức tạp hơn so với index thông thường, nhưng có thể mang lại hiệu suất đáng kể cho các truy vấn làm việc với phân vùng cụ thể.

##### Làm thế nào để đánh index hiệu quả?

* Để đánh được index thì ta phải có chiến lược, có cái nhìn toàn cảnh về hệ thống sau khi nó phát triển thì dữ liệu sẽ tăng trưởng như nào, người dùng tìm kiếm trên những cột nào, đâu là cột tìm kiếm nhiều, đâu là cột được tìm kiếm cùng nhau.
* Đọc chiến lược thực thi để kiểm tra index có được dùng vào trong câu lệnh hay không, có hiệu quả hay không.

##### Lưu ý và tips

* Index đánh trên cột nào thì lưu thông tin của cột đó => Khi đánh index trên một cột mà select/where cả những cột khác => không đủ thông tin nên db phải seq scan (quét cả bảng) => Bản chất db lưu data theo row nên khi (full table scan)thì nó lấy thông tin tất cả các cột chứ không phải chỉ lấy thông tin của các cột select => index không có tác dụng => Chú ý: Một câu lệnh đang chạy nhanh mà chỉ cần thêm một cột ở select/where là có thể bị chậm.
* **Tạo index vào foreign key**: giải quyết lock và giúp các câu lệnh làm việc trên bảng cha nhanh hơn.
* **Chỉ Tạo Index Khi Cần Thiết:** Tạo index cho các cột thường xuyên được sử dụng trong điều kiện WHERE, JOIN, ORDER BY để cải thiện hiệu suất truy vấn.
* **Hạn Chế Số Lượng Index:** Quá nhiều index có thể làm chậm quá trình cập nhật và làm tăng kích thước cơ sở dữ liệu.
* Mặc định index là dạng B-TREE. Tuy nhiên, B-TREE INDEX không thể lưu trữ được giá trị NULL. Vì vậy, nếu tìm kiếm trên cột có giá trị null thì index sẽ không có hiệu quả.
  + Cách giải quyết:
    1. Có tạo bitmap index thay vì b-tree (Nhưng cần kiểm tra lại chiến lược thực thi vì không phải lúc nào cũng hiệu quả)
    2. Tạo composite index với constant (ví dụ trường salary có giá trị null thì sẽ tạo create index idx\_salary\_1\_wecommit on wecommit\_test(salary,1))
* Tránh dùng functions trong predicates (các điều kiện sau WHERE, JOIN, ORBER BY và GROUP BY)
  + Database không sử dụng index nếu nó có một số function predefined trong column.
  + Ví dụ: SELECT \* FROM TABLE1 WHERE UPPER(COL1)='ABC'Copy
  + Do function UPPER(), database không sử dụng index trên COL1. Nếu có bất kỳ cách nào để tránh function đó trong SQL, bạn phải tạo một index dựa trên function mới hoặc phải custom các column trong database để cải thiện performance.
* Tránh sử dụng các ký tự % ở đầu câu
  + Ví dụ trong trường hợp sau LIKE '%abc' gây quét toàn bộ table: SELECT \* FROM TABLE1 WHERE COL1 LIKE '%ABC'Copy
  + Trong hầu hết các trường hợp, việc sử dụng ký tự % này gây giới hạn performance lớn.
* Trong một câu sql có nhiều điều kiện trong where:
  + Đánh index ở cột có độ phân bổ dữ liệu càng nhỏ (cột có ít các giá trị khác nhau) thì hiệu suất càng cao.
    - Ví dụ: Nếu mỗi bản ghi có giá trị trường field là duy nhất, thì tương đương với tìm kiếm bằng id.
  + Khi đánh composite index (chỉ mục kết hợp), hãy đặt cột có miền phân bổ giá trị càng ít lên đầu tiên, thì hiệu suất càng cao.
    - Ví dụ: Chỉ mục kết hợp (ft, createdts) sẽ nhanh hơn (createdts, ft) trong trường hợp ft có miền giá trị phân bổ nhỏ hơn.
  + Nếu có cột có miền giá trị là duy nhất (ví dụ ở đây là transfer\_code và ft), thì chỉ cần đánh single index (chỉ mục đơn) mà không cần composite index.
  + Với trường hợp các cột có miền phân bổ giá trị gần giống nhau, cần đọc kế hoạch thực thi (execution plan) để tìm ra chỉ mục tối ưu nhất.

#### Join các bảng

Tối ưu hóa các câu lệnh SQL JOIN nhiều bảng

* Sử dụng loại JOIN phù hợp:
  + INNER JOIN: Chỉ trả về các bản ghi có trong cả hai bảng.
  + LEFT JOIN: Trả về tất cả các bản ghi từ bảng bên trái, cùng với các bản ghi khớp từ bảng bên phải.
  + RIGHT JOIN: Trả về tất cả các bản ghi từ bảng bên phải, cùng với các bản ghi khớp từ bảng bên trái.
  + FULL JOIN: Trả về tất cả các bản ghi từ cả hai bảng, bao gồm cả các bản ghi không khớp.

Lựa chọn loại JOIN phù hợp sẽ giúp giảm thiểu số lượng bản ghi cần xử lý, từ đó tăng hiệu suất truy vấn.

* Viết truy vấn rõ ràng:
  + Sử dụng tên cột thay vì \*.
  + Sử dụng điều kiện WHERE để lọc dữ liệu trước khi JOIN.
  + Sử dụng mệnh đề ORDER BY để sắp xếp kết quả.
* Tránh sử dụng các hàm và toán tử phức tạp trong điều kiện JOIN:
  + Sử dụng các hàm và toán tử phức tạp trong điều kiện JOIN có thể làm giảm hiệu suất truy vấn.
* Sử dụng EXPLAIN để kiểm tra kế hoạch thực thi của truy vấn:
  + EXPLAIN giúp bạn hiểu cách thức MySQL thực thi truy vấn. Sử dụng EXPLAIN để xác định các điểm cần tối ưu hóa.
* Sử dụng các công cụ tối ưu hóa truy vấn:
  + Có nhiều công cụ có thể giúp bạn tối ưu hóa các câu lệnh SQL JOIN.
  + Ví dụ:
    - MySQL Workbench: Cung cấp giao diện đồ họa để tạo và tối ưu hóa truy vấn.
    - dbForge Studio for MySQL: Cung cấp nhiều tính năng để tối ưu hóa truy vấn, bao gồm gợi ý chỉ mục, kiểm tra kế hoạch thực thi và refactor truy vấn.

### “Quy trình chữa bệnh khi công nhân bị bệnh” - Tối ưu hóa câu lệnh SQL bị chậm

#### Insert/Update

Khi thấy câu lệnh châm ta cần lưu ý một số thứ sau đây:

* [Kiểm tra và xử lý lock](#_Công_cụ_kiểm)
* Kiểm tra trigger
* Tần suất commit: Nếu thực hiện commit quá thường xuyên, điều này sẽ làm giảm hiệu suất vì việc commit đòi hỏi phải ghi dữ liệu vào đĩa cứng.
* Tối ưu subquery (tách nhỏ câu sql để tối ưu từng phần)
  + ví dụ: insert into A select \* from B where id=1, chậm có thể ở select => tối ưu câu select bằng index.
* Chậm vì quá nhiều index: vì khi insert bản ghi thì cũng phải ghi thêm vào index
* Chậm vì constrain (ràng buộc): Các ràng buộc như khóa ngoại (FOREIGN KEY), khóa duy nhất (UNIQUE KEY) cũng có thể làm giảm hiệu suất nếu không được thiết kế tối ưu.
* Dùng hint để tăng tốc
  + VD trong Oracle:
    - Hint APPEND:
      * Khi sử dụng hint APPEND, trình tối ưu hóa câu truy vấn sẽ thêm các bản ghi vào bên trên Hight water mark. Điều này giúp tránh phân mảnh dữ liệu và khóa dữ liệu, từ đó tăng hiệu suất cho thao tác INSERT.
      * Lượng Redo log sinh ra khi thực hiện INSERT APPEND chỉ bằng 0.2% so với lượng Redo Log sinh ra khi thực hiện câu lệnh Insert thông thường.
      * Do lượng sinh Redo log sinh ra ít hơn rất nhiều, nên mặc dùng có cùng chi phí thực thi, nhưng hiệu năng và thời gian thực hiện của câu lệnh INSERT APPEND sẽ tốt hơn câu lệnh INSERT thông thường
      * Ghi chú: Môi trường thực hiện việc đánh giá này

– Oracle Database 19c

– Table test\_insert cấu hình NOLOGGING

* + - Hint RESULT\_CACHE: Hint này cho phép lưu trữ kết quả của một câu truy vấn trong bộ nhớ cache. Khi câu truy vấn đó được thực hiện lần tiếp theo, kết quả sẽ được lấy trực tiếp từ bộ nhớ cache thay vì phải tính toán lại, giúp tăng tốc hiệu suất lên 0 giây. Tuy nhiên, cần cân nhắc vấn đề bộ nhớ và dữ liệu không được cập nhật kịp thời khi sử dụng hint này.
      * Nếu một trong các bảng emp hoặc dept bị thay đổi dữ liệu (bởi các câu lệnh DML), hệ thống sẽ tự động phát hiện được việc này và đánh dấu kết quả trong RESULT\_CACHE MEMORY là không còn đúng nữa.
      * Chính vì thế sau khi dữ liệu của bảng bị thay đổi, hệ thống sẽ cần TÍNH TOÁN LẠI chứ không lấy ngay kết quả trong RESULT CACHE MEMORY để trả ra cho người dùng được.

#### Delete

* Delete có điều kiện Where bị chậm => dùng index
* Delete không có điều kiện where thì dùng TRUNCATE vì:
* DELETE không có mệnh đề WHERE sẽ cần quét FULL toàn bộ các block dữ liệu của bảng.
* DELETE thực hiện sinh ra UNDO (đây là dữ liệu cần thiết để có thể thực hiện ROLLBACK).
* Cơ chế của TRUNCATE không cần thiết tạo ra UNDO (do đó TRUNCATE không thể ROLLBACK).

## “Nhiều công nhân

### “Va chạm- tranh chấp” - Lock Conflict

Hãy tưởng tượng Database giống như một ngôi nhà và những session kết nối vào database giống như những người đang sống trong căn nhà đó.

Trong căn nhà này có một nguyên tắc:

* Tại một thời điểm, mỗi đồ vật trong nhà chỉ được sử dụng bởi 1 người duy nhất.
* Nếu nhiều người đều muốn sử dụng chung một đồ vật, chúng ta sẽ áp dụng nguyên tắc “người nào tới trước thì được sử dụng trước”, những người đằng sau sẽ phải đợi.

Ví dụ:

* Hai người Huy và Tuân cùng có mặt trong tòa nhà Wecommit (Wecommit Building)
* Cả hai người đều muốn sử dụng máy chấm công được đặt ở cửa ra vào.
* Ông Huy là người đến lúc 9 giờ và Tuân đến lúc 9 giờ 5 phút.
* Theo nguyên tắc của “người vào tới trước thì được sử dụng trước”, lúc này mọi chuyện sẽ xảy ra như sau:
  + Huy sẽ được sử dụng máy chấm công trước
  + Tuân sẽ buộc phải đợi khi Huy sử dụng xong thì mới được dùng. Việc đợi này có thể diễn ra LÂU hay CHÓNG là do Huy quyết định.

Hiện tượng tôi vừa mô tả bên trên chính là hình ảnh của SQL Lock Conflict (hay một số anh em còn gọi đơn giản là SQL Lock hoặc Transaction Lock hoặc Lock Conflict, một số người có thể gọi nhanh là Lock, dù gọi thế nào đi nữa tôi muốn mọi người hãy hiểu về bản chất của nó.).

**SQL Lock Conflict xảy ra khi có nhiều transaction cùng thay đổi một bản ghi trong cùng TABLE.**

Transaction nào tới trước sẽ thực hiện “KHÓA” (thuật ngữ gọi là LOCK) các bản ghi (ROWS) bị thay đổi lại, bản chất phải có cơ chế này để đảm bảo tính toàn vẹn cho dữ liệu.

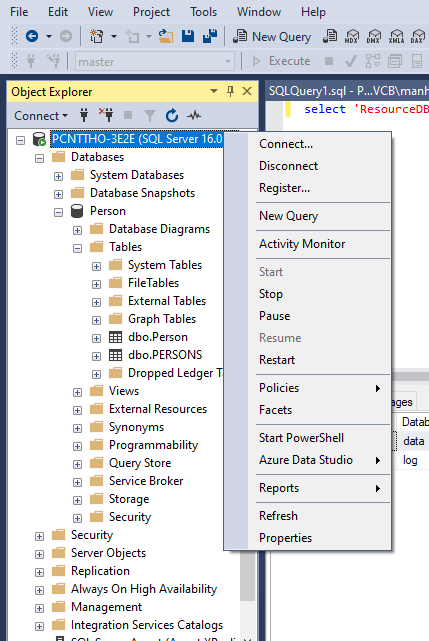
Tất cả những transaction tới sau sẽ không thể thay đổi các bản ghi (ROWS) đã bị khóa.

Các “khóa” này sẽ được giải phóng khi transaction đang giữ “khóa” thực hiện COMMIT hoặc ROLLBACK.

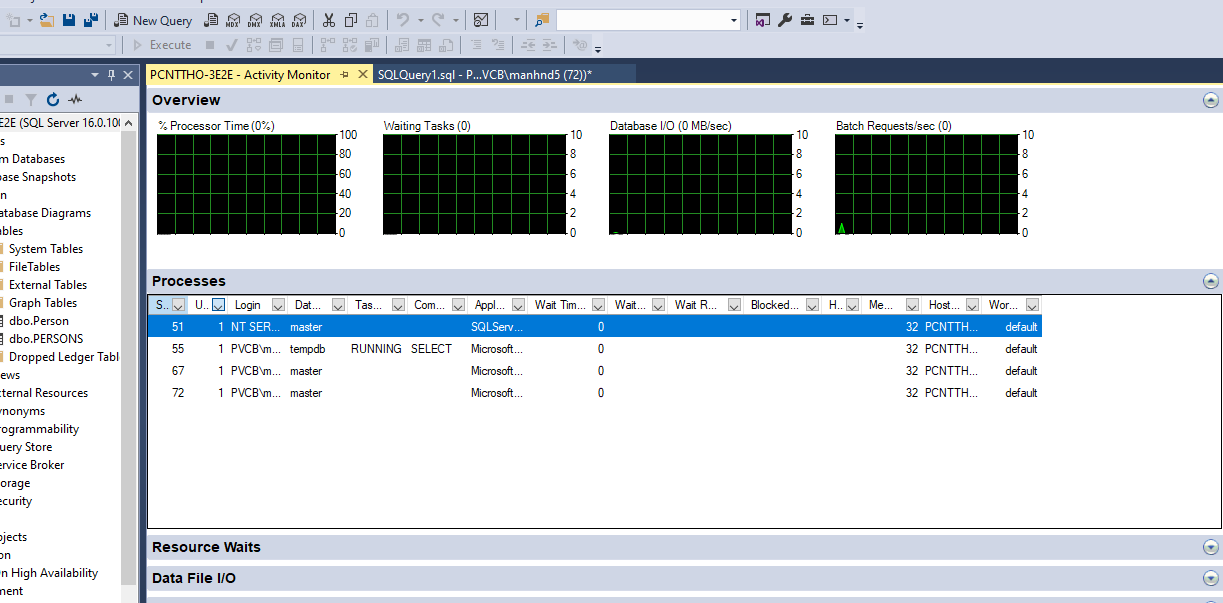
### Công cụ kiểm tra khi có va chạm (lock)

#### Activity Monitor

* Chuột phải vào server > Activity Monitor > Process



* Cột Blocked by thể hiện tiến trình đang bị block bởi tiến trình khác



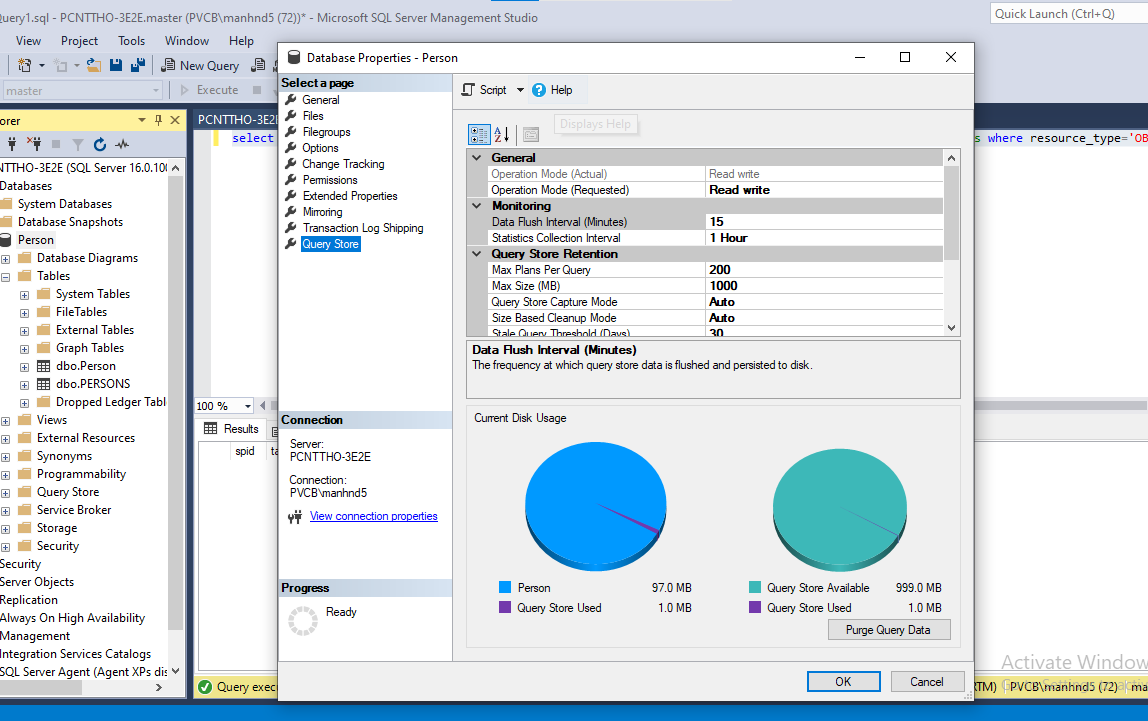
* Chuột phải vào row > details để xem là câu lệnh nào
* Nếu có lock thì kill đi tiến trình ít quan trọng hơn

#### Lệnh sql

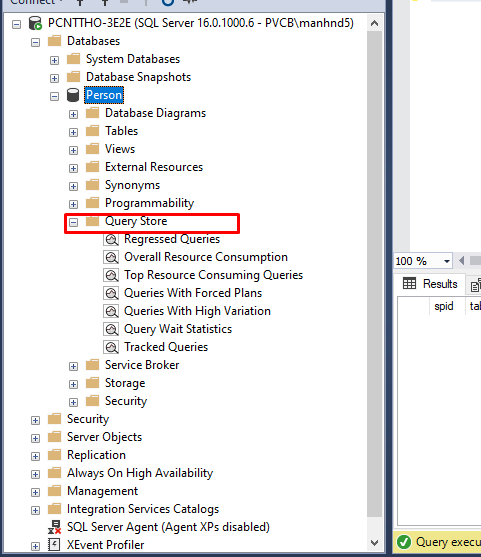
* Câu lệnh sql kiểm tra xem lock:
  + select session\_id 'Id tiến trình bị lock, phải đợi', command, blocking\_session\_id 'Id tiến trình chiếm lock' from sys.dm\_exec\_requests where blocking\_session\_id <> 0;
* Kiểm tra xem tiến trình thực thi là câu nào:
  + DBCC INPUTBUFER(ID\_tien\_trinh)
* Xóa tiến trình nào đó:
  + kill ID\_tien\_trinh
* Kiểm tra xem table đang nào bị lock
  + select request\_session\_id spid, OBJECT\_NAME(resource\_associated\_entity\_id) tableName from sys.dm\_tran\_locks where resource\_type='OBJECT';

### Giám sát hiệu năng – Query store

* Lưu được thông tin lịch sử khi thực hiện câu lệnh SQL
* Có từ phiên bản SQL Server 2016
  + Tuy nhiên mặc định không được enable
  + Từ sql server 2022 mặc định được enable
* Cách kiểm tra xem đã bật hay chưa:
  + Câu lệnh: select name as dbName, is\_query\_store\_on from sys.databases;
* Cách bật:
  + Alter database db\_name set QUERY\_STORE = ON (OPERATION\_MODE = READ\_WRITE);
  + Trong đó db\_name là tên của database
* Cấu hình query store
  + Chuột phải vào db > Properties> Query Store



* + Cấu hình:
    - Monitoring > Data Flush Interval (Minutes): Bao nhiêu phút thì lưu thông tin (nên để 5)
    - Monitoring > Statistics Collection Interval: Bao lâu thì capture lại báo cáo của các câu lệnh (nên để 30p)
    - Query Store Retention > Stale Query Threshold (Days): Lưu thông tin của query store trong bao nhiêu ngày (nên để 90)
    - Query Store Retention > Max Size
  + Xem báo cáo:



* Câu lệnh để xem toàn bộ các câu lệnh chạy trên db:

use db\_name

select qt.query\_sql\_text, q.query\_id,

qt.query\_text\_id, p.plan\_id, rs.last\_execution\_time

from sys.query\_store\_query\_text as qt

join sys.query\_store\_query as q

on qt.query\_text\_id = q.query\_text\_id

join sys.query\_store\_plan as p

on q.query\_id = p.query\_id

join sys.query\_store\_runtime\_stats as rs

on p.plan\_id = rs.plan\_id

order by rs.last\_execution\_time desc

* + Trong đó: db\_name là tên db
  + Từ câu lệnh trên, thêm các điều kiện để tạo thành báo cáo tùy chỉnh. Ví dụ: top 10 câu lệnh chiếm nhiều nhất physical\_io\_reads (đọc ghi từ đĩa) trong vòng 24h:

use dbname

select top 10 rs.avg\_physical\_io\_reads, qt.query\_sql\_text, q.query\_id,

select top 10 rs.avg\_physical\_io\_reads, qt.query\_sql\_text, q.query\_id, qt.query\_text\_id, p.plan\_id,

rs.runtime\_stats\_id, rsi.start\_time, rsi.end\_time, rs.avg\_rowcount, rs.count\_executions

from sys.query\_store\_query\_text as qt

join sys.query\_store\_query as q

on qt.query\_text\_id = q.query\_text\_id

join sys.query\_store\_plan as p

on q.query\_id = p.query\_id

join sys.query\_store\_runtime\_stats as rs

on p.plan\_id = rs.plan\_id

join sys.query\_store\_runtime\_stats\_interval as rsi

on rsi.runtime\_stats\_interval\_id = rs.runtime\_stats\_interval\_id

where rsi.start\_time >= DATEADD(hour, -24, GETUTCDATE())

order by rs.avg\_physical\_io\_reads desc;

# Hiểu về Bloom Filter

## Định nghĩa Bloom Filter

- Bloom Filter là một cấu trúc dữ liệu xác suất, được sử dụng để kiểm tra sự tồn tại của một phần tử trong một tập hợp.

- Nó sử dụng một số hàm băm để ghi nhớ các phần tử được thêm vào, và có thể kiểm tra nhanh chóng xem một phần tử có nằm trong tập hợp hay không.

## Ưu điểm của Bloom Filter

- Tiết kiệm bộ nhớ: Bloom Filter sử dụng ít bộ nhớ hơn so với các cấu trúc dữ liệu khác để lưu trữ cùng một tập hợp.

- Truy xuất nhanh chóng: Kiểm tra sự tồn tại của một phần tử trong Bloom Filter chỉ mất thời gian O(k), với k là số hàm băm được sử dụng.

- Không có giới hạn về kích thước tập hợp: Bloom Filter có thể lưu trữ một tập hợp không giới hạn các phần tử, mặc dù độ chính xác sẽ giảm khi tập hợp trở nên quá lớn.

## Triển khai Bloom Filter

### Cấu trúc dữ liệu Bloom Filter

- Bloom Filter sử dụng một mảng bit để lưu trữ các phần tử. Mảng này được khởi tạo tất cả bằng 0.

- Khi thêm một phần tử vào Bloom Filter, các hàm băm được sử dụng để xác định các vị trí trong mảng bit cần được thiết lập thành 1.

- Khi kiểm tra sự tồn tại của một phần tử, các hàm băm được sử dụng để xác định các vị trí tương ứng trong mảng bit. Nếu tất cả các vị trí đều là 1, thì phần tử được coi là tồn tại trong Bloom Filter (mặc dù có thể có sai lệch).

### Triển khai hàm add() và contains()

- `add(element)`: Sử dụng các hàm băm để xác định các vị trí trong mảng bit và thiết lập chúng thành 1.

- `contains(element)`: Sử dụng các hàm băm để xác định các vị trí trong mảng bit. Nếu tất cả các vị trí đều là 1, trả về `true`, ngược lại trả về `false`.

### Lựa chọn số hàm băm và kích thước mảng bit

- Số hàm băm k ảnh hưởng đến xác suất âm tính giả, với k càng lớn thì xác suất càng thấp.

- Kích thước mảng bit m ảnh hưởng đến bộ nhớ sử dụng và xác suất âm tính giả, với m càng lớn thì xác suất càng thấp.

- Có các công thức toán học để tính toán số hàm băm k và kích thước mảng bit m tối ưu dựa trên số lượng phần tử n cần lưu trữ và xác suất âm tính giả mong muốn.

## Kết hợp Bloom Filter với Redis

### Tại sao kết hợp Bloom Filter với Redis

- Redis là một database in-memory nhanh chóng, được sử dụng rộng rãi trong các ứng dụng hiện đại.

- Bloom Filter là cấu trúc dữ liệu hiệu quả để kiểm tra sự tồn tại của phần tử, phù hợp để kết hợp với Redis.

- Sự kết hợp này giúp tăng tốc độ và hiệu quả của các ứng dụng, đặc biệt là trong các trường hợp cần kiểm tra sự tồn tại của phần tử thường xuyên.

### Cách triển khai Bloom Filter trên Redis

- Redis cung cấp module `redis-cell` cho phép triển khai Bloom Filter trực tiếp trên Redis.

- Bạn có thể cài đặt và cấu hình module `redis-cell` trên Redis server.

- Sử dụng các lệnh như `CF.ADD`, `CF.EXISTS` và `CF.INFO` để thao tác với Bloom Filter trên Redis.

### Ví dụ về cách sử dụng Bloom Filter trên Redis

# Tạo một Bloom Filter với tên "my\_bloom\_filter"

CF.ADD my\_bloom\_filter "element1"

CF.ADD my\_bloom\_filter "element2"

CF.ADD my\_bloom\_filter "element3"

# Kiểm tra sự tồn tại của các phần tử

CF.EXISTS my\_bloom\_filter "element1" # Trả về 1 (true)

CF.EXISTS my\_bloom\_filter "element4" # Trả về 0 (false)

# Lấy thông tin về Bloom Filter

CF.INFO my\_bloom\_filter

```

Trong ví dụ trên, chúng ta đã tạo một Bloom Filter với tên "my\_bloom\_filter" trên Redis, thêm một số phần tử vào, và kiểm tra sự tồn tại của các phần tử. Cuối cùng, chúng ta lấy thông tin về Bloom Filter đang sử dụng.

### Lưu ý khi sử dụng trong dự án thực tế

#### Xác định nhu cầu

Trước tiên, hãy xác định rõ nhu cầu của ứng dụng và xem Bloom Filter có phù hợp không. Bloom Filter thường được sử dụng khi cần kiểm tra sự tồn tại của phần tử một cách nhanh chóng, với khả năng chấp nhận một số lượng nhỏ âm tích giả.

#### Chọn thư viện phù hợp

Trong Java, bạn có thể sử dụng thư viện Guava của Google hoặc thư viện Caffeine để triển khai Bloom Filter. Ngoài ra, bạn cũng có thể sử dụng các thư viện khác như Bloom4J hoặc Fastbloom.

#### Cấu hình tối ưu

Khi tạo Bloom Filter, bạn cần tính toán các thông số như kích thước mảng bit, số lượng hàm băm để đạt được xác suất âm tính giả mong muốn. Việc cấu hình không chính xác có thể ảnh hưởng đến hiệu suất và độ chính xác của Bloom Filter.

#### Tích hợp với Redis

Nếu ứng dụng của bạn sử dụng Redis, bạn có thể kết hợp Bloom Filter với Redis để tăng tốc độ và hiệu quả của các thao tác kiểm tra sự tồn tại. Redis cung cấp module `redis-cell` cho phép triển khai Bloom Filter trực tiếp trên Redis.

#### Cân bằng giữa bộ nhớ và độ chính xác

Bloom Filter sử dụng ít bộ nhớ hơn so với các cấu trúc dữ liệu khác, nhưng nó có thể có âm tính giả. Bạn cần cân bằng giữa bộ nhớ sử dụng và xác suất âm tính giả dựa trên nhu cầu của ứng dụng.

#### Sử dụng trong các trường hợp phù hợp

Bloom Filter thường được sử dụng trong các trường hợp như:

- Lọc nhanh các yêu cầu trùng lặp.

- Kiểm tra sự tồn tại của người dùng, sản phẩm, hoặc các thực thể khác trong hệ thống.

- Tối ưu hóa các truy vấn database bằng cách loại bỏ trước các yêu cầu không có trong hệ thống.

#### Kết hợp với các kỹ thuật khác

Bloom Filter có thể được kết hợp với các kỹ thuật khác như cache, phân trang, hoặc các cơ chế xử lý ngoại lệ để tăng cường hiệu suất và độ tin cậy của ứng dụng.

#### Giám sát và đo lường hiệu suất

Theo dõi và đo lường hiệu suất của Bloom Filter trong ứng dụng là rất quan trọng. Bạn có thể sử dụng các công cụ giám sát để theo dõi các chỉ số như số lượng phần tử, xác suất âm tính giả, và thời gian truy xuất.