# 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

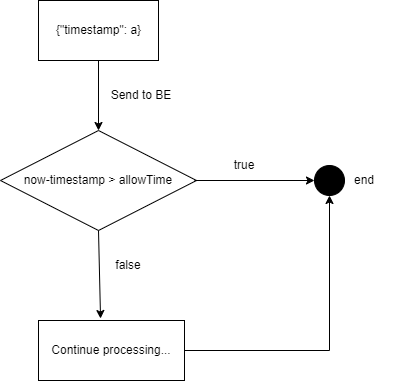
**[Thinking Clearly about Performance](https://queue.acm.org/detail.cfm?id=1854041)**

**[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.

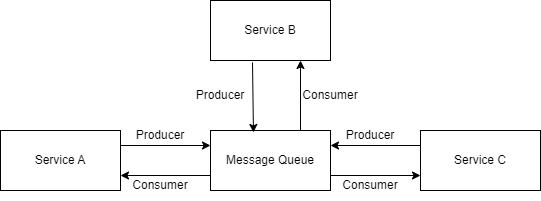
### Khóa bi quan

 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.

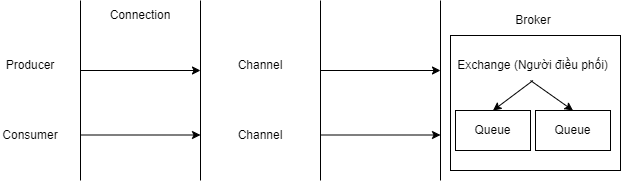
# Kiến trúc Message Queue

## 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

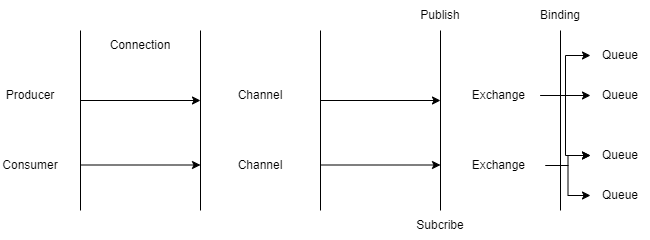
## Kiến trúc (RabbitMQ)



## 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

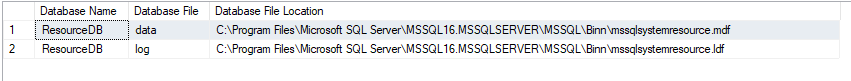
* Được tạo lại mỗi khi db khởi động:
  + Không cần recovery
  + Khôn cần backup
* Phục vụ chứa:
  + Temp table
  + Các công việc sử dụng dữ liệu tạm như SORT
* Tối ưu tempDb:
  + Mặc định SQL server chỉ tạo 1 temp file => Gây tắc nghẽn về I/O => Tạo nhiều tempfile
  + Ưu tiền đặt ở phần vùng có đọc ghi 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

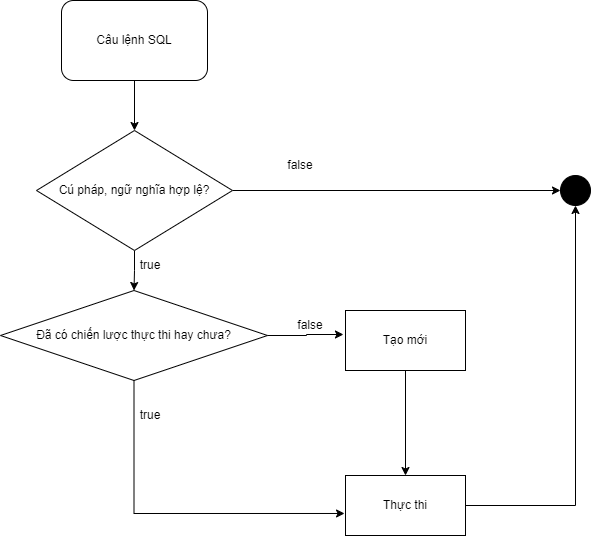
* Không nên lưu dữ liệu vào PRIMARY Filegroup

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

Phải có chiến lược backup

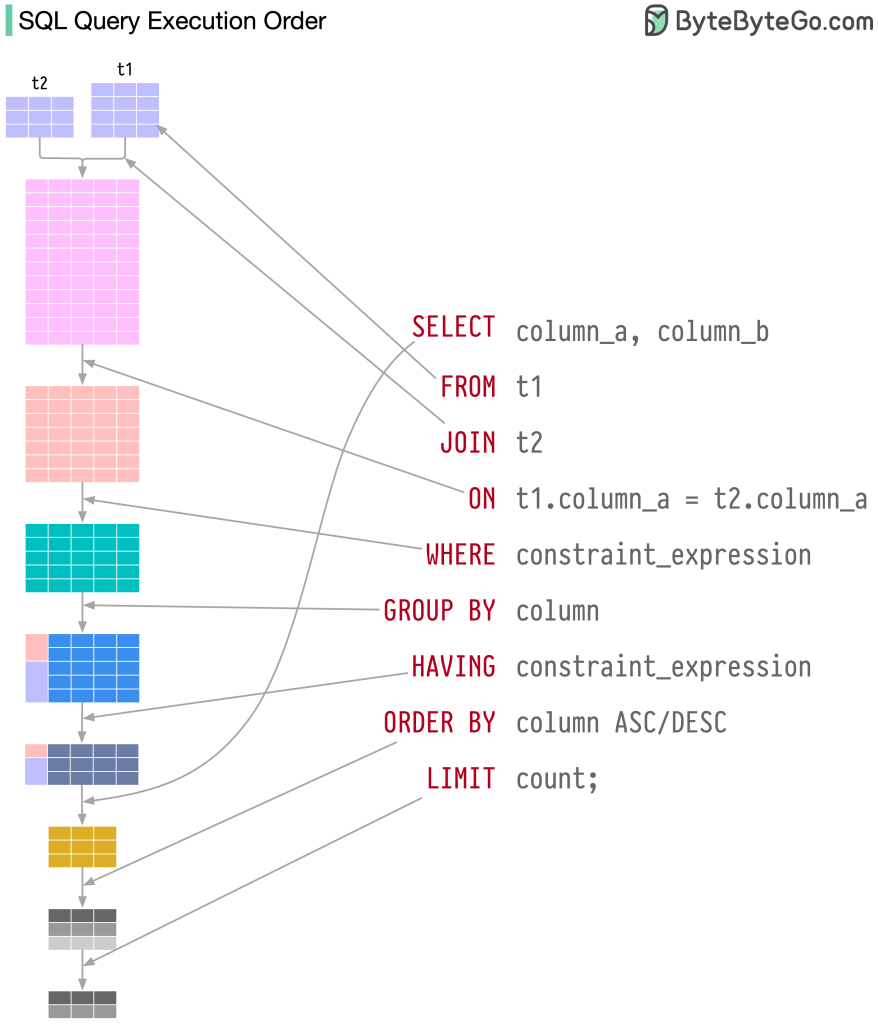
## “Công nhân giao hàng” – Câu lệnh SQL

### “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:



#### 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

#### 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 KPI – Chiến lực thực thi

#### Định nghĩa

Chiến lược thực thi là tập hợp các bước mà hệ thống quản trị cơ sở dữ liệu (DBMS) 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

Đọc từ Phải sang trái, từ trên xuống dưới

#### Ý nghĩa các thành phần trong execute plan

##### 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.

#### Hint

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

##### 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**.

### “Đối tượng làm việc”

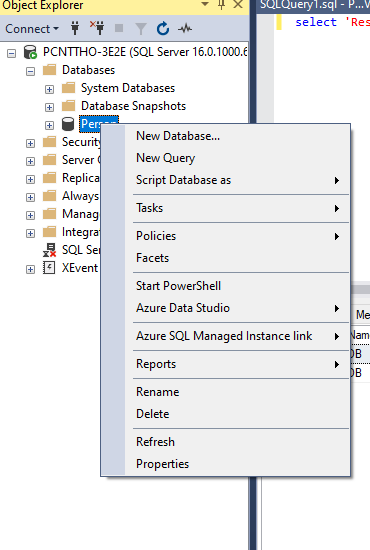
#### Table

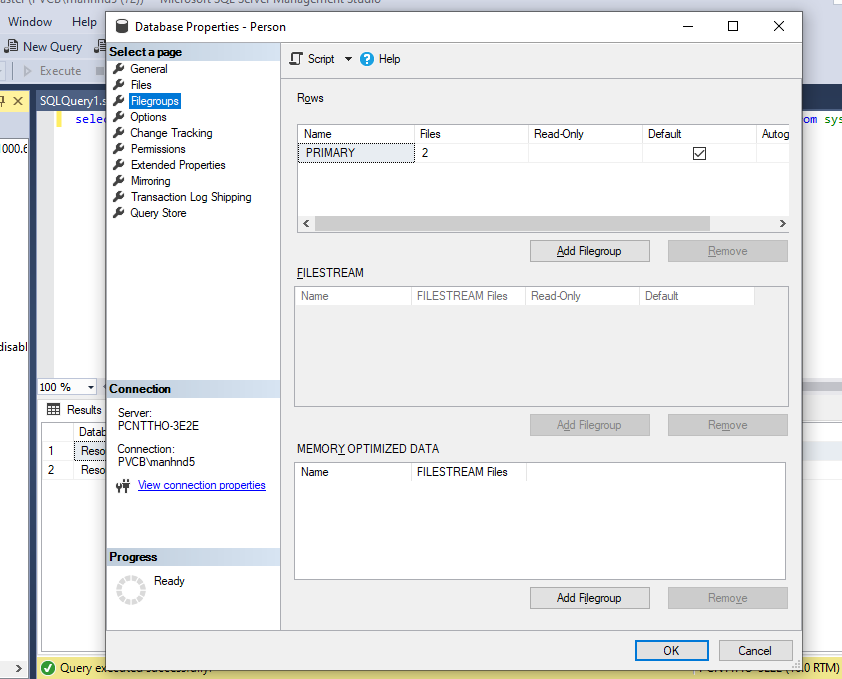
##### NDF

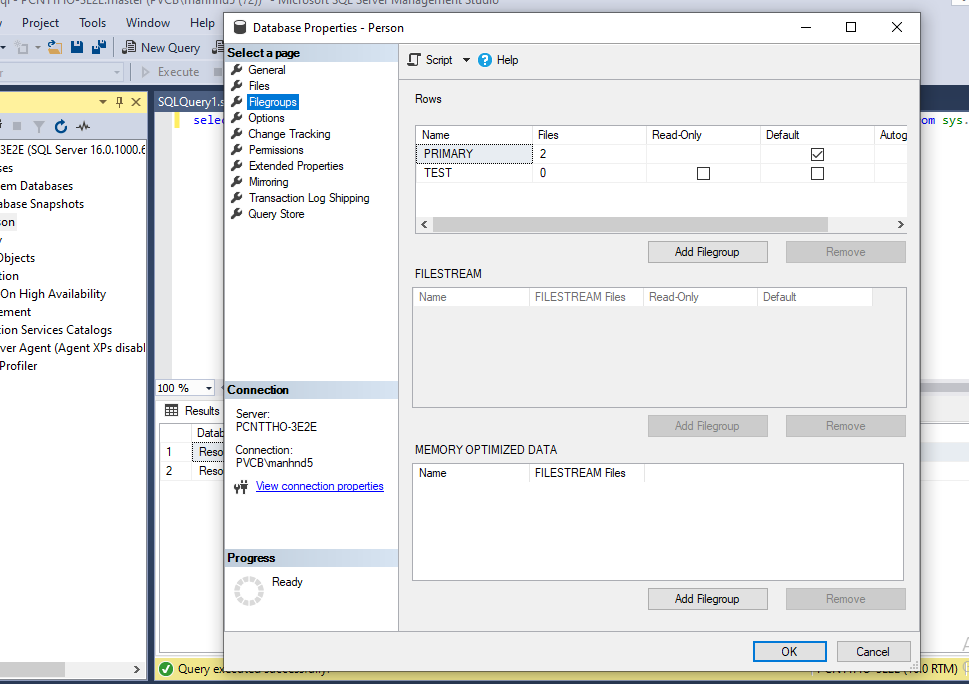
Tạo ra các file ndf riêng để lưu các data có cùng nhóm

Cách làm:

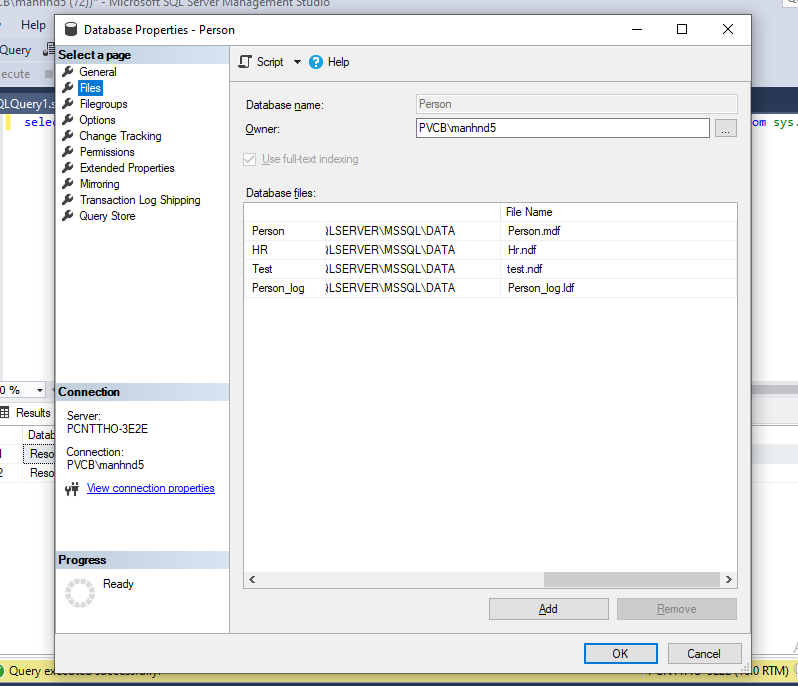
* Tạo filegroup: chuột phải vào db muốn tạo filegroup chọn properties > FilesGroups > Add Filegroup > Điền tên > OK







* 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.

###### 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

);

#### 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.

#### 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.

##### Các loại index

* **Primary Key Index:** Index được tạo tự động cho cột primary key, đảm bảo mỗi giá trị trong cột là duy nhất và không null.
* **Unique Index:** Index được tạo cho cột có ràng buộc unique, đảm bảo mỗi giá trị trong cột là duy nhất.
  + **Cú pháp:** CREATE UNIQUE INDEX index\_name ON table\_name (column1);
* **Composite Index:** Index được tạo cho một nhóm các cột, cho phép tìm kiếm dựa trên các cột kết hợp.
  + **Cú pháp:** CREATE INDEX index\_name

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

* **Partial Index:** là một loại index trong SQL mà chỉ lưu trữ một phần của dữ liệu trong bảng thay vì toàn bộ dữ liệu. Việc này cho phép tối ưu hóa truy vấn bằng cách chỉ tập trung vào một phần cụ thể của dữ liệu mà thường xuyên được truy cập hoặc cần được tìm kiếm.
  + **Cú pháp:** CREATE INDEX index\_name

ON table\_name (column1)

WHERE condition;

* **Clustered Index:** Trong một số hệ quản trị cơ sở dữ liệu, dữ liệu trong bảng được tổ chức theo thứ tự của index. Trong trường hợp này, index được gọi là clustered index.
  + **Cú pháp:** CREATE CLUSTERED INDEX index\_name

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

* **Non-Clustered Index:** Index không sắp xếp lại dữ liệu trong bảng, mà chỉ tạo ra một bản sao của các cột index và các con trỏ đến dữ liệu thực.
  + **Cú pháp**: CREATE NONCLUSTERED INDEX index\_name
  + ON table\_name (column1, column2, ...);
* **Full-Text Index**:
  + Định nghĩa: Full-Text Index được sử dụng để tối ưu hóa việc tìm kiếm văn bản hoặc dữ liệu văn bản dựa trên từ khoá hoặc cụm từ.
  + Ý nghĩa: Index này cho phép truy vấn văn bản theo nội dung thực sự của nó, thay vì chỉ dựa vào các điều kiện tìm kiếm truyền thống.
  + Ví dụ: Tạo full-text index trên cột content trong bảng articles:

CREATE FULLTEXT INDEX idx\_content ON articles(content);

Sau đó, thực hiện truy vấn full-text search:

SELECT \* FROM articles WHERE MATCH(content) AGAINST('keyword');

* **Spatial Index**:
  + Định nghĩa: Spatial Index được sử dụng để tối ưu hóa việc truy vấn dữ liệu không gian, chẳng hạn như địa lý hoặc hình học.
  + Ý nghĩa: Index này cho phép thực hiện các truy vấn không gian như tìm kiếm các đối tượng trong một khu vực cụ thể một cách hiệu quả.
  + Ví dụ: Tạo spatial index trên cột location trong bảng places:

CREATE SPATIAL INDEX idx\_location ON places(location);

Sau đó, thực hiện truy vấn spatial search:

SELECT \* FROM places WHERE ST\_CONTAINS(area, POINT(x, y));

* **Bitmap Index**:
  + Định nghĩa: Bitmap Index được sử dụng cho các cột có ít giá trị duy nhất hoặc có phân phối đồng đều.
  + Ý nghĩa: Index này sử dụng một bitmap cho mỗi giá trị có thể có của cột, giúp tìm kiếm nhanh chóng và hiệu quả trong các truy vấn với các điều kiện lọc.
  + Ví dụ: Tạo bitmap index trên cột gender trong bảng customers:

CREATE BITMAP INDEX idx\_gender ON customers(gender);

Sau đó, thực hiện truy vấn với điều kiện lọc:

SELECT \* FROM customers WHERE gender = 'Male';

##### Lưu ý

* **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.
* **Cập Nhật Thống Nhất:** Đảm bảo rằng dữ liệu được đồng bộ hóa với index để tránh sự không nhất quán.

#### 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.

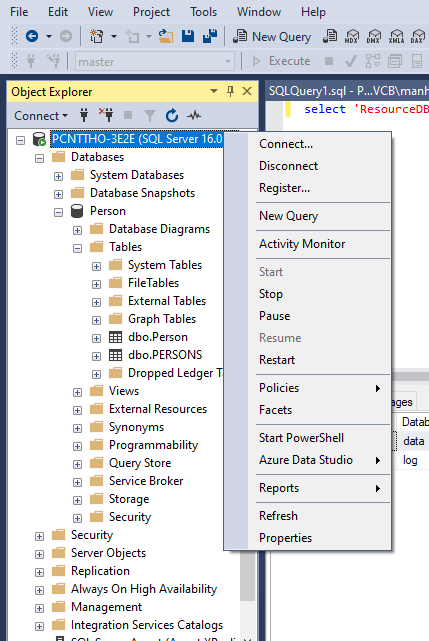
* Sử dụng index:
  + Chỉ mục giúp truy cập dữ liệu nhanh hơn. Tạo chỉ mục cho các cột được sử dụng trong điều kiện JOIN sẽ giúp tăng hiệu suất truy vấn.
* Viết truy vấn rõ ràng:
  + Sử dụng tên cột thay vì \*.
  + Sử dụng điều kiện WHERE để lọc dữ liệu trước khi JOIN.
  + Sử dụng mệnh đề ORDER BY để 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:
    1. MySQL Workbench: Cung cấp giao diện đồ họa để tạo và tối ưu hóa truy vấn.
    2. 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.

## “Nhiều công nhân”

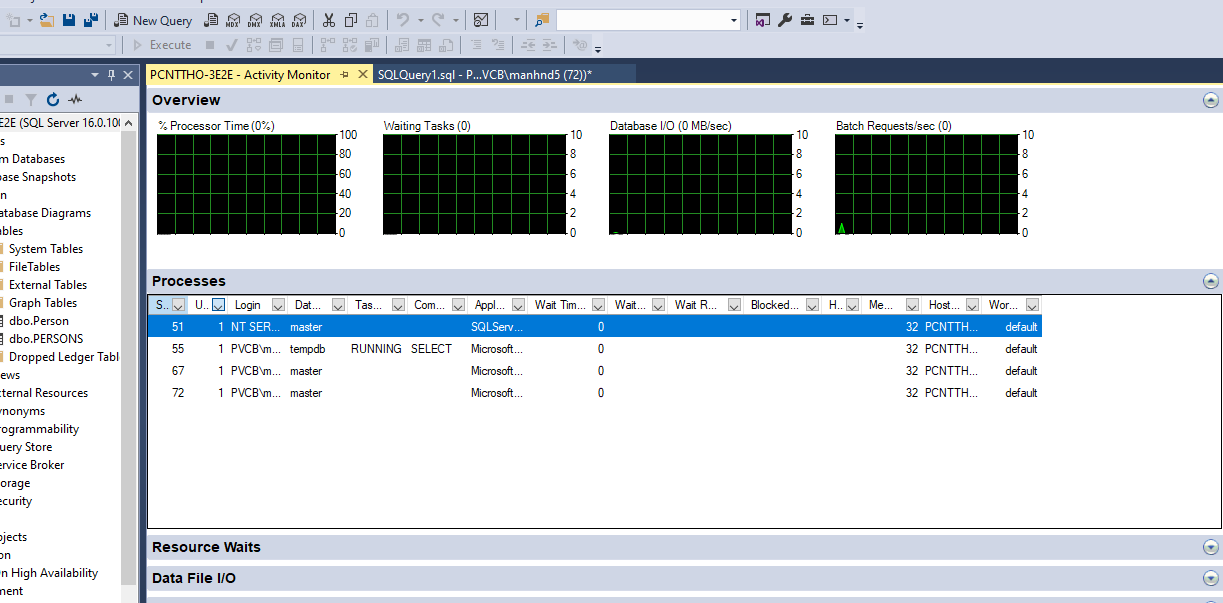
### Công cụ kiểm tra khi có va chạm (deadlock)

#### Activity Monitor

* Chuột phải vào server > Activity Monitor > Process



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



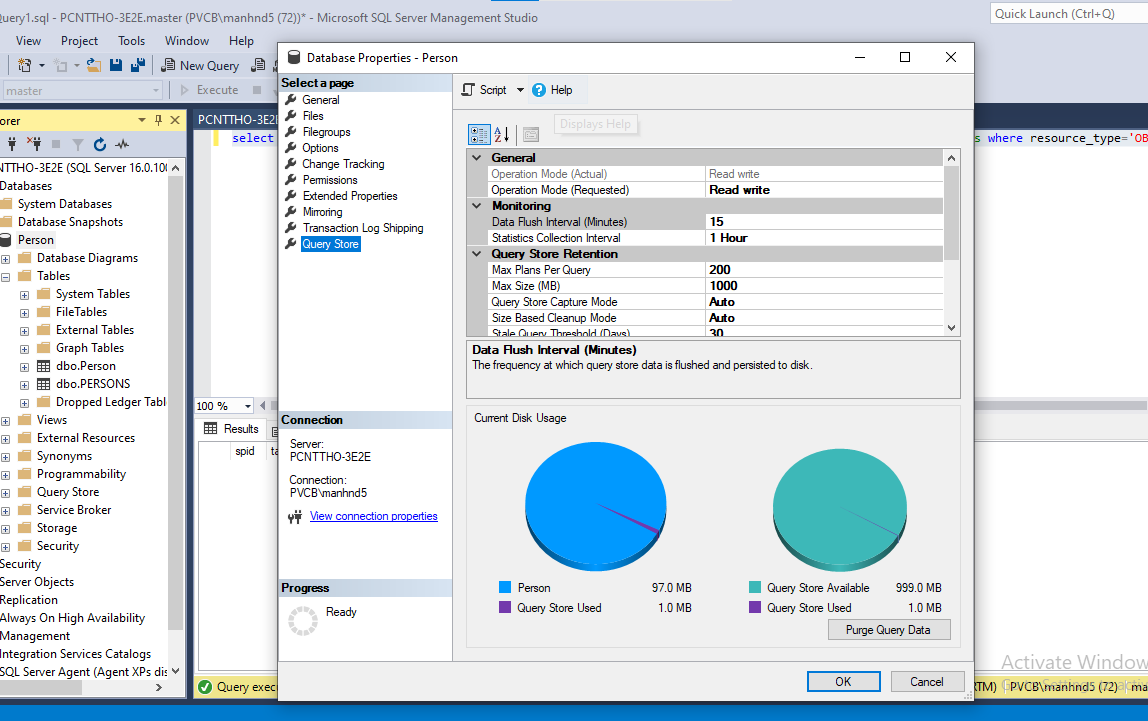
* Chuột phải vào row > details để xem là câu lệnh nào
* Nếu có lock thì kill đi tiến trình ít quan trọng hơn

#### Lệnh sql

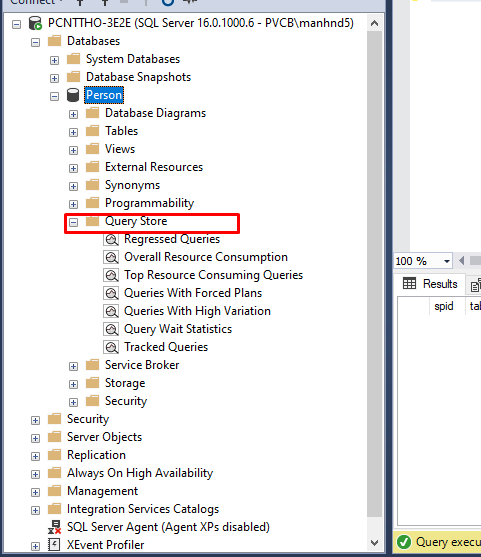
* Câu lệnh sql kiểm tra xem lock:
  + select session\_id 'Id tiến trình bị lock, phải đợi', command, blocking\_session\_id 'Id tiến trình chiếm lock' from sys.dm\_exec\_requests where blocking\_session\_id <> 0;
* Kiểm tra xem tiến trình thực thi là câu nào:
  + DBCC INPUTBUFER(ID\_tien\_trinh)
* Xóa 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;