**Clustered Index:**

* **Clustered Index** lưu trữ và sắp xếp dữ liệu vật lý trong table hoặc view dựa trên các giá trị khóa của chúng. Các cột khóa này được chỉ định trong định nghĩa index. Mỗi table hoặc view chỉ có duy nhất một Clusterd Index vì bản thân các dòng dữ liệu được lưu trữ và sắp xếp theo thứ tự vật lý dựa trên các cột trong loại Index này.
* Khi dữ liệu trong table hoặc view cần được lưu trữ và sắp xếp theo một thứ tự nhất định chính là lúc cần dùng đến Clustered Index. Khi một table có một Clusted Index thì khi đó table được gọi là Clustered Table. Nếu không, các dòng dữ liệu của table được lưu trong một cấu trúc không được sắp xếp gọi là HEAP.

**Non-Clustered Index:**

* **Non-Clustered** có một cấu trúc tách biệt với data row trong table hoặc view. Mỗi một index loại này chứa các giá trị của các cột khóa trong khai báo của index, và mỗi một bản ghi giá trị của key trong index này chứa một con trỏ tới dòng dữ liệu tương ứng của nó trong table.
* Mỗi con trỏ từ một dòng của **Non-Clustered index** tới một dòng dữ liệu trong table được gọi là  “row locator”. Cấu trúc của row locator phụ thuộc vào việc các trang dữ liệu được lưu trong HEAP hay trong một **Clustered Table** như đã diễn giải ở mục **Clustered Index** ở trên. Đối với HEAP, row locator là một con trỏ tới dòng dữ liệu, với clustered table, row locator chính là khóa index của **clustered index**.
* Bạn có thể thêm vào các cột không phải là khóa vào một index có sẵn với giới hạn là 900 bytes và 16 cột.

Cả **Clustered Index** và **Non-Clustered Index** đều có thể là unique. Điều này có nghĩa là không có hai dòng nào có sự trùng lặp giá trị trong các cột tạo ra index.

Index cũng được tự động bảo trì mỗi khi dữ liệu của table có sự thay đổi.

**Indexes và Constraints**

Index được tự động tạo ra mỗi khi chúng ta tạo Primary Key hoặc Unique Contraints. Ví dụ bạn tạo một cột ID trên table và set Primary key cho nó thì SQL Server tự động tạo ra Primary Key contraints và **index** cho nó.

**Index trong SQL Server được sử dụng như thế nào bởi trình Query Optimizer để tối ưu hóa câu lệnh SQL?**

Một database được thiết kế tốt sẽ làm giảm các hoạt động I/O trên ổ đĩa và tiêu tốn ít tài nguyên hệ thống hơn, do đó làm tăng hiệu suất của các câu lệnh SQL. Index rất hữu ích cho các loại câu lệnh khác nhau trong SQL như Inser, Update, Delete hoặc Merge... Hãy xem xét câu lệnh sau chạy trên database AdventureWorks2012:

SELECT Title, HireDate FROM HumanResources.Employee WHERE EmployeeID = 250

Khi câu lệnh này thực thi, trình query optimizer sẽ đánh giá các phương thức để lấy dữ liệu có sẵn và lựa chọn một phương thức tối ưu nhất. Các phương thức đó có thể là table scan, hoặc scan trên một hoặc nhiều Index nếu như chúng tồn tại.   
Nếu phương thức scan table được lựa chọn, query optimizer sẽ đọc tất cả các dòng trên table và trích xuất ra các dòng dữ liệu khớp với điều kiện trong mệnh đề where.   
  
Phương thức scan table là một phương thức tiêu tốn nhiều chi phí của hệ thống, nó thực hiện nhiều hoạt động I/O trên ổ cứng và dễ làm tài nguyên hệ thống trở nên quá tải nếu hành động scan table xảy ra thường xuyên với cường độ lớn. Tuy nhiên, phương thức scan table lại là phương thức tối ưu nhất nếu như số dòng trả về chiếm tỷ lệ cao trên tổng số dòng dữ liệu có trong table.  
  
Khi query optimizer sử dụng Index, nó sẽ tìm kiếm dữ liệu trên các cột đã được index trước, sau đó sẽ tìm nơi lưu trữ dòng dữ liệu tương ứng trên ổ cứng và trích xuất các dòng dữ liệu khớp với điều kiện. Thông thường, tìm kiếm trên index nhanh hơn nhiều so với tìm kiếm trong table, bởi vì không giống như table, index chỉ chứa thông tin của những cột được index và chúng được sắp xếp theo thứ tự.  
  
Trình query optimizer sẽ chọn phương thức tối ưu nhất để chạy câu query của bạn, tuy nhiên, nếu như không có **index** nào trên bảng thì nó sẽ dùng phương thức là scan table. Do đó, nhiệm vụ của bạn là phải lựa chọn các cột phù hợp trên table để tạo **index** cho chúng nhằm giúp trình query optimizer lựa chọn được phương pháp tối ưu nhất. **SQL Server** cung cấp công cụ  [Database Engine Tuning Advisor](http://msdn.microsoft.com/en-us/library/hh231122.aspx) để giúp chúng ta phân tích và gợi ý những **index**cần tạo trong môi trường **database**của chúng ta.

Để hiểu index là gì, ta hãy thử đặt ra một kịch bản truy vấn đơn giản, giả sử ta muốn tìm trong bảng users, lấy ra thông tin của tất cả những user có name là DungNQ.

**Default**  
SELECT \* FROM USER WHERE NAME = 'DungNQ';

Trong tình huống này, database của chúng ta sẽ phải đọc ra từng row trong bảng users, so sánh giá trị của trường name trong row đó với giá trị cần tìm, và thực hiện từ row đầu tiên cho tới row cuối cùng của bảng. Cách làm này gọi là full table search. Dĩ nhiên, ta có thể thấy đây là cách làm chậm nhất, vì phải lọc qua tất cả dữ liệu hiện hữu trong bảng. Từ đó đặt ra vấn đề, muốn tăng tốc độc truy vấn, phải làm sao giảm thiểu số lượng bản ghi/trường cần phải đọc khi thực hiện truy vấn.

Tăng tốc truy vấn.

Hãy tưởng tượng, cũng trong kịch bản truy vấn trên, nhưng trong tay ta đang có sẵn một chuỗi, bao gồm tất cả các giá trị của trường name có trong bảng users, và các giá trị này đã được sắp xếp theo thứ tự tăng dần.Lúc này, để tìm ra row có giá trì name = 'DungN', thay vì lôi lần lượt từ giá trị đầu tiên đến giá trị thứ n cuối cùng ra để so sánh (cần thực hiện n phép so sánh ), ta có thể thực hiện một [phép tìm kiếm nhị phân](https://en.wikipedia.org/wiki/Binary_search_algorithm). Nếu bạn ko hiểu khái niệm Binary Search là gì (hàng hiếm, giống mình :v), và cũng lười đọc link wiki tiếng Anh, thì có thể xem thêm tại bài viết [này](https://viblo.asia/nguyenthingoc/posts/n6BkGypLM5aV) của bạn [@nguyenthingoc](https://viblo.asia/u/nguyenthingoc) , trong đó có một đoạn giải thích khá trực quan về Binary Search (độ dễ hiểu đến đâu thì tùy cảm nhận từng người :v). Lười đọc nữa thì có thể hiểu nôm na, ta lần lượt băm đôi dãy cần tìm kiếm, lấy giá trị ở giữa ra so sánh, vì dãy này đã được sắp xếp theo thứ tự, nên trong trường hợp giá trị ở giữa này khác giá trị cần tìm, ta có thể biết được giá trị cần tìm nếu có sẽ phải nằm ở nửa nào của dãy. Lặp lại thao tác cho đến hết, tối đa ta sẽ chỉ cần Log cơ số 2 của n phép tìm kiếm để tìm ra giá trị cần tìm.

Quay trở lại vấn đề chính, nếu trong tay ta có một dãy như trên, rõ ràng việc thực hiện phép tìm kiếm ban đầu đã trở nên dễ dàng hơn rất nhiều. Không chỉ số lượng row ta phải đọc ít đi, mà trong mỗi row, ta cũng chỉ cần đọc giá trị của đúng trường name mà thôi. Và giả sử, ứng với mỗi một phần tử trong dãy như trên, ta có thể gán ứng với 1 giá trị con trỏ, trỏ để row tương ứng trong bảng users, ta hoàn toàn có thể truy vấn tất cả các thông tin của user cần tìm, chứ không chỉ giá trị trường name được lưu trong dãy. Dạng cấu trúc dữ liệu như dãy đc mô tả trên, chính là một dạng index, cũng thuộc loại phổ biến bậc nhất , B-tree index.

Một kịch bản khác.

Cũng trong kịch bản truy vấn trên, ta không có một dãy được sắp xếp theo thứ tự sẵn như trên nữa. Thay vào đó, ta có một array các cặp key => value dạng như 'DungNQ' => 0x28939 trong đó, key chính là giá trị của trường name trong bảng users, còn value là biến con trỏ chỉ đến row tương ứng trong bảng. Rõ ràng, trong trường hợp này, truy vấn đầu tiên, nếu được thực hiện trong array kia, sẽ nhanh chóng hơn rất nhiều so với thực hiện full table scan trên bảng users . Kiểu cấu trúc dữ liệu giống như mô tả trên cũng là một dạng index tương đối phổ biến , hash table index.

Index là gì

Ngoài B-tree index hay hash table index ra, còn có những loại index tương đối kém phổ biến hơn , như R-tree index hay bitmap index. Đi sâu vào giới thiệu chi tiết từng loại index tồn tại, có lẽ sẽ cần phải khuôn khổ của một bài viết khác. Trong bài viết này, ta hãy chỉ tạm nói với nhau rằng, có rất nhiều loại index, mỗi loại đều có một cấu trúc riêng, đem lại ưu thế riêng cho từng hoàn cảnh truy vấn. Ví dụ như hash table index có lợi thế rất nhanh khi thực hiện các phép truy vấn với toán tử = hay <> , tuy nhiên lại bó tay trước các phép truy vấn với toán tử < hay >. B-tree index là loại index phổ biến nhất, đa số các phần mềm database đều dùng đây là loại index mặc định. Tuy index loại này rất mềm dẻo, hữu ích trong rất nhiều tình huống so với hash table index, nhưng lại chịu chết với các phép so sánh <> ( những điều này hoàn toàn dễ hiểu, khi ta nhìn lại chi tiết cấu tạo của từng loại index giới thiệu ở trên ). Tổng kết chung lại , ta có thể có một định nghĩa khá chung và toàn diện cho index

**Default**  
Index là một dạng cấu trúc dữ liệu, trong đó chứa giá trị của một trường nhất định trong một bảng dữ liệu, đồng thời trỏ đến row dữ liệu tương ứng.

(Khái niệm tự chém, mấy em học sinh sinh viên lấy chép vào bài kiểm tra, thầy cho bao nhiêu điểm thì tự chịu :v)

Còn lại gì sau cơn mưa

Lan man một hồi, có lẽ cũng là gọi là tạm đủ để trả lời mấy câu hỏi đặt ra lúc ban đầu : index là gì (khái niệm tự chém kia), index hoạt động như thế nào (giải thích chi tiết 2 loại cơ bản nhất là đủ), tại sao đánh index lại làm truy vấn nhanh hơn(xem phần hoạt động). Tuy nhiên, nếu chỉ có nhiêu đó, thì chắc chỉ có ích cho hoạt động duy nhất , đấy là đi phỏng vấn( hỏi thằng khác, nếu bạn may mắn, or chém cho thằng phỏng vấn nghe, nếu bạn là phó thường dân ). Nếu bạn là một lập trình viên bình thường, quần đùi chân đất, đọc xong bằng kia chữ, rốt cuộc là ta được cái gì :

* Thứ nhất là khi nào cần đánh index, và đánh như nào : Qua giải thích chi tiết như trên kia, thì ta có thể thấy là index chỉ hữu ích với những truy vấn có điều kiện tìm kiếm liên quan đến cột ta đánh index. Thế thì xét xem cột nào hay dùng nhất thì đánh index vào. Ví dụ như name trong users, tối ngày cần đến nó, vì lấy ra thông tin kiểu gì chả phải lấy cho nó cái tên, rồi còn check xem có username này chưa khi đăng kí, rồi lúc login .. blah blah. vậy thì tạo cho nó cái index.
* Index hữu dụng thế, nên trường id, đa số các phần mềm database đều tự động tạo sẵn cho ta rồi, khỏi mất công tạo lại. Như trong Mysql thì có thể dùng SHOW INDEX FROM table\_name FROM db\_name; để kiểm tra xem bảng đã có những index nào rồi, khỏi phải tạo lại.( Cái này bên trên ko nói đến tí nào, giấu hàng giờ mới nói ).
* Tạo index cho một trường, là tạo ra một cấu trúc lưu dữ liệu của trường đó, kèm theo một số thứ khác nữa. Thế nên cũng đừng thấy, ví dụ bảng posts có trường contents hay được search, lại hăm hở đánh index cho cái trường đấy, mà biết đâu cái index to gần bằng nguyên xi cả bảng.
* Quan trọng nhất là , hiều được cách hoạt động của từng loại index rồi, thì dùng sao cho hợp lí. Cụ thể hơn, các trường hợp hay gặp nhất là :
  + B-tree index dựa vào việc sắp xếp các value theo thứ tự. Thế nên chẳng hạn đánh index ( chỉ khai báo index không, thì database sẽ tự hiểu là mặc định, dùng B-tree index ) cho trường name, nhưng lại thực hiện toàn các truy vấn kiểu WHERE NAME NOT IN ('blah', 'bloh', 'bleh') hay WHERE NAME <> 'SPARTANNNNNNNN' thì index cũng như không.
  + Cũng về B-tree index, ở trên không nói rõ, tại mình cũng ko nắm chắc 100%, nhưng có vẻ như thứ tự sắp xếp trong index là thứ tự tăng dần. Nên nếu dùng index loại này, các truy vấn kiểu như WHERE NAME LIKE 'DUNG%' hay thậm chí là WHERE NAME LIKE 'D%G%' thì index đều có tác dụng, nhưng với WHERE NAME LIKE '%UNG' thì index có cũng như ko.
  + Cuối cùng là vì đặc điểm cấu tạo của B-tree index, index dạng này có tác dụng tăng tốc với các phép truy vấn ORDER BY.
  + Về hash table index, do cấu tạo như đã nói, index dạng này chỉ có tác dụng với các truy vấn = và <>hay NOT IN, nghĩa là các truy vấn chính xác, giống hay ko giống. Các truy vấn dạng như WHERE NAME LIKE 'A%' , index dạng này cũn không có tác dụng, chứ đừng nói đến các toán tử khác như > hay < . Bù lại thì đúng theo tinh thần chuyên môn hóa cao, các phép truy vấn như ví dụ đầu tiên WHERE NAME = 'DungNQ' thực hiện trên index dạng này lại có tốc độ vô đối. Tùy theo nhu cầu mà thiết kế index cho hợp lí.