

Everything developers need to know about SQL performance

Thuật ngữ chuyên ngành

Từ chuyên ngành	Giải thích
Obfuscate:	Làm mờ, làm khó hiểu
Function-based index	Tạo index trực tiếp trên hàm với tham số là một số cột trong bảng
Concatenated index	Tạo index trên nhiều cột
Execution plan	Kế hoạch thực thi
Smart logic	Logic thông minh

1. Chương 1	6
1.1. Cấu trúc của một chỉ mục (index)	6
1.2. Nút lá chỉ mục (The Index Leaf Nodes)	7
1.3. Cây tìm kiếm (B-tree)	8
1.4. Vấn đề sử dụng index nhưng vẫn chậm, phần 1	10
2. Chương 2	14
2.1. Mệnh đề WHERE	14
2.1.1. Toán tử bằng	14
2.1.2. Khóa chính	14
2.1.3. Chỉ mục kết hợp (concatenated index)	16
2.1.4. Sử dụng chỉ mục nhưng truy vấn vẫn chậm, phần 2	21
2.2. Hàm	25
2.2.1. Tìm kiếm không phân biệt chữ hoa hay chữ thường sử dụng UPPER hoặc LOWER	25
2.2.2. Các hàm định nghĩa bởi người dùng	29
2.2.3. Over-indexing	31
2.3. Truy vấn có tham số	31
TÍNH BẢO MẬT	32
HIỆU NĂNG	32
2.4. Tìm kiếm theo khoảng	40
2.4.1. Lớn hơn, nhỏ hơn và trong khoảng (between)	40
2.4.2. Indexing LIKE Filters	45
2.4.3. Chỉ mục gộp (index merge)	48
2.5. Chỉ mục một phần	50
2.6. NULL trong cơ sở dữ liệu Oracle	52
2.6.1. Đánh chỉ mục giá trị NULL	53
2.6.2. Ràng buộc NOT NULL	56
2.6.3. Mô phỏng chỉ mục một phần	60
2.7. Những điều kiện bị mờ (obfuscated conditions)	63
2.7.1. Các kiểu dữ liệu	63
2.7.2. Kiểu chuỗi số	68
2.7.3. Các cột kết hợp	70
2.7.4. SMART LOGIC	72
2.7.5. Biểu thức toán học (MATH)	76
3. HIỆU NĂNG VÀ KHẢ NĂNG MỞ RỘNG	77
3.1. Khối lượng dữ liệu ảnh hưởng đến hiệu năng	77
3.2. Những tác động tới hiệu năng của tải trên hệ thống	81
3.3. Thời gian phản hồi và thông lượng	83
4. Phép toán kết nối	87
4.1. NESTED LOOPS (VÒNG LẶP LỒNG NHAU)	88
4.2. HASH JOIN (KẾT NỐI BĂM)	101

4.3. SORT MERGE JOIN	109
5. Gom cụm dữ liệu: Sức mạnh thứ hai của chỉ mục	110
5.1. Bộ lọc chỉ mục được sử dụng một cách có chủ đích	111
5.2. Chỉ mục bao phủ truy vấn, Index-only scan (index covers query)	114
5.3. Index-Organized Tables	119
6. Sorting and Grouping	124
6.1. INDEXING ORDER BY	124
6.2. INDEXING ASC , DESC và NULLS FIRST/LAST:	128
6.3. INDEXING GROUP BY:	132
7. CHAPTER7: PARTIAL RESULT	135
7.1. LẤY VỀ TOP-N BẢN GHI (hoặc DÒNG - ROWS)	135
7.2. PHÂN TRANG KẾT QUẢ	140
7.3. SỬ DỤNG WINDOW FUNCTIONS CHO PHÂN TRANG.	148
8. Modifying Data - Cập nhật dữ liệu	150
8.1. Insert	150
8.2. Delete	152
8.3. Update	153

Hình 1 Chỉ mục kết hợp	19
Hình 2: Truy vấn trên nhánh nhỏ	34
Hình 3: Truy vấn trên nhánh lớn	35
Hình 4 : C# không sử dụng tham số ràng buộc	37
Hình 5: C# sử dụng tham số ràng buộc	37
Hình 6: Java không sử dụng tham số ràng buộc	37
Hình 7: Java có sử dụng tham số ràng buộc	38
Hình 8: Perl không sử dụng tham số ràng buộc	38
Hình 9: Perl có sử dụng tham số ràng buộc	38
Hình 10: PHP không sử dụng tham số ràng buộc	38
Hình 11: PHP sử dụng tham số ràng buộc	39
Hình 12: Ruby không sử dụng tham số ràng buộc	39
Hình 13: Ruby sử dụng tham số ràng buộc	39
Hình 14: Ví dụ tham số ràng buộc không hoạt động	40
Hình 15: Truy vấn date_of_birth có chặn đầu trên và đầu dưới	41
Hình 16: Truy vấn khoảng date_of_birth có thêm điều kiện ngoài	41
Hình 17: Phạm vi quét index với date_of_birth đứng trước	42

Hình 18: Phạm vi quét của index với subsidiary_id đứng trước	43
Hình 19: Kế hoạch thực hiện rõ ràng index	44
Hình 20: Sử dụng toàn bộ vị trí truy cập	45
Hình 21: Truy vấn sử dụng between	45
Hình 22: Truy vấn between sử dụng giống truy vấn sử dụng (\leq) và (\geq)	45
Hình 23: Truy vấn LIKE khi đặt kí tự đại diện ở giữa	46
Hình 24: Các phương pháp tìm kiếm LIKE khác nhau	47
Hình 25: Truy vấn sử dụng hai điều kiện độc lập	49

1. Chương 1

2. Cấu trúc của một chỉ mục (index)

"chỉ mục (index) làm tăng tốc độ của câu truy vấn" đó là một cách giải thích đơn giản nhất về chỉ mục mà tôi từng nghe trước đây. Mặc dù nó đã diễn tả khía cạnh quan trọng nhất của một chỉ mục nhưng thật không may nó không đủ cho cuốn sách này. Chương này sẽ mô tả cấu trúc của chỉ mục nhưng cũng không đi quá sâu vào chi tiết. Nó sẽ cung cấp đủ kiến thức để bạn có thể hiểu các phần liên quan đến hiệu suất của SQL được nhắc đến trong các phần tiếp theo của cuốn sách.

Một chỉ mục là một cấu trúc riêng biệt trong cơ sở dữ liệu, nó được tạo ra bằng câu lệnh **create index**. Nó cần có không gian lưu trữ riêng trên thiết bị lưu trữ (đĩa cứng) và có một phần bản sao của dữ liệu của bảng được lập chỉ mục. Điều này có nghĩa rằng việc tạo ra một chỉ mục là có sự dự thừa về dữ liệu. Tạo một chỉ mục không thay đổi dữ liệu của các bảng; nó chỉ tạo một cấu trúc dữ liệu mới và nó trỏ đến bảng. Tóm lại một chỉ mục giống như phần mục lục của một quyển sách, nó có không gian riêng (cần phải lưu trữ như dữ liệu) và có chỉ dẫn đến các thông tin (dòng dữ liệu trong bảng) thực sự được lưu trữ tại các nơi khác nhau trên thiết bị lưu trữ vật lý (thường qua hệ thống quản lý tập tin, lưu trên đĩa cứng).

Các chỉ mục nhóm cụm – Clustered indexes

Khi nhắc đến chỉ mục, thông thường ta hay nghĩ ngay đến cấu trúc dữ liệu nằm ngoài bảng dữ liệu như trình bày ở trên. Các chỉ mục này được gọi là chỉ mục không nhóm cụm – nonclustered index. Tuy nhiên thực tế như trong SQL Server và MySQL còn một dạng tổ chức chỉ mục khác gọi là chỉ mục nhóm cụm – clustered index. Với chỉ mục nhóm cụm, các bảng dữ liệu thực sự được tổ chức, lưu trữ sắp xếp theo thuộc tính được đánh chỉ mục. Các bảng này được gọi là Index-Organized Tables (IOT) trong Oracle database.

Chương 5, “Clustering Data” sẽ mô tả chi tiết hơn và giải thích các ưu điểm, nhược điểm của chỉ mục dạng này. Trong chương này, khi nhắc tới chỉ mục là nói tới chỉ mục không nhóm cụm – nonclustered index.

Tìm kiếm trong chỉ mục giống như là tìm kiếm trong một danh mục điện thoại. Điểm mấu chốt ở đây là tất cả các mục được sắp xếp theo một thứ tự được định nghĩa trước. Thực hiện tìm dữ liệu trong một tập dữ liệu đã được sắp xếp là nhanh và dễ dàng hơn bởi vì đã có thứ hạng sắp xếp cho các phần tử.

Tuy nhiên chỉ mục phức tạp hơn so với một danh mục điện thoại vì nó phải thay đổi liên tục. Việc cập nhật một danh mục điện thoại cho mọi thay đổi là không thể vì một lý do đơn giản là không có chỗ trống giữa các phần tử để có thể thêm một phần tử mới. Nó chỉ có thể chỉnh sửa trong lần xuất bản tiếp theo. Ngược lại một hệ quản trị CSDL cần phải luôn cập nhật các chỉ mục mà vẫn đảm bảo thời gian xử lý truy vấn

không bị ảnh hưởng nghiêm trọng (quá lâu). Nó cần phải phải thực hiện các câu lệnh **insert**, **delete** và **update** nhanh nhất có thể trong khi cố gắng cập nhật cấu trúc chỉ mục index mà không cần phải di chuyển quá nhiều dữ liệu.

Một cơ sở dữ liệu (CSDL) thông thường sử dụng kết hợp hai cấu trúc dữ liệu để làm nên chỉ mục: danh sách liên kết đôi (a doubly linked list) và một cây tìm kiếm (a search tree). Hai cấu trúc dữ liệu này giải quyết hầu như tất cả các vấn đề về hiệu suất của cơ sở dữ liệu.

3. Nút lá chỉ mục (The Index Leaf Nodes)

Mục đích chính của index là trình diễn bảng dữ liệu được đánh chỉ mục theo thứ tự sắp xếp. Tuy nhiên, chỉ mục không nhóm cụm không thay đổi trật tự lưu trữ các dòng dữ liệu trong bảng theo thứ tự của cột được đánh chỉ mục. Hình dung là nếu các dòng được lưu trữ theo thứ tự sắp xếp, câu lệnh **insert** sẽ cần phải di chuyển các phần tử phía sau để có thể cho thêm vào phần tử mới. Việc di chuyển nhiều dữ liệu sẽ mất rất nhiều chi phí vì thế mà câu lệnh **insert** sẽ rất chậm. Cách giải quyết là sử dụng một thứ tự logic không phụ thuộc vào thứ tự vật lý trên bộ nhớ.

Thứ tự logic có thể được thiết lập thông qua danh sách liên kết đôi (doubly linked list). Mọi nút sẽ có liên kết tới hai nút lân cận giống như một chuỗi. Nút mới được chèn vào giữa hai nút đã tồn tại bằng cách cập nhật các liên kết. Vị trí lưu trữ của các nút mới trên thiết bị lưu trữ (đĩa cứng) không quan trọng vì danh sách liên kết đôi đã lưu trữ thứ tự logic của các nút.

Cấu trúc dữ liệu được gọi là danh sách liên kết đôi bởi vì mỗi nút có thể trỏ tới hai nút lân cận của nó (nút trước và nút sau). Nó cho phép cơ sở dữ liệu đọc các nút tiếp hoặc quay lại nút trước nếu cần. Danh sách liên kết đôi cho phép chèn thêm các nút mới vào giữa 2 nút trước đó mà không phải di chuyển một số lượng lớn dữ liệu – nó chỉ cần thay đổi một số con trỏ.

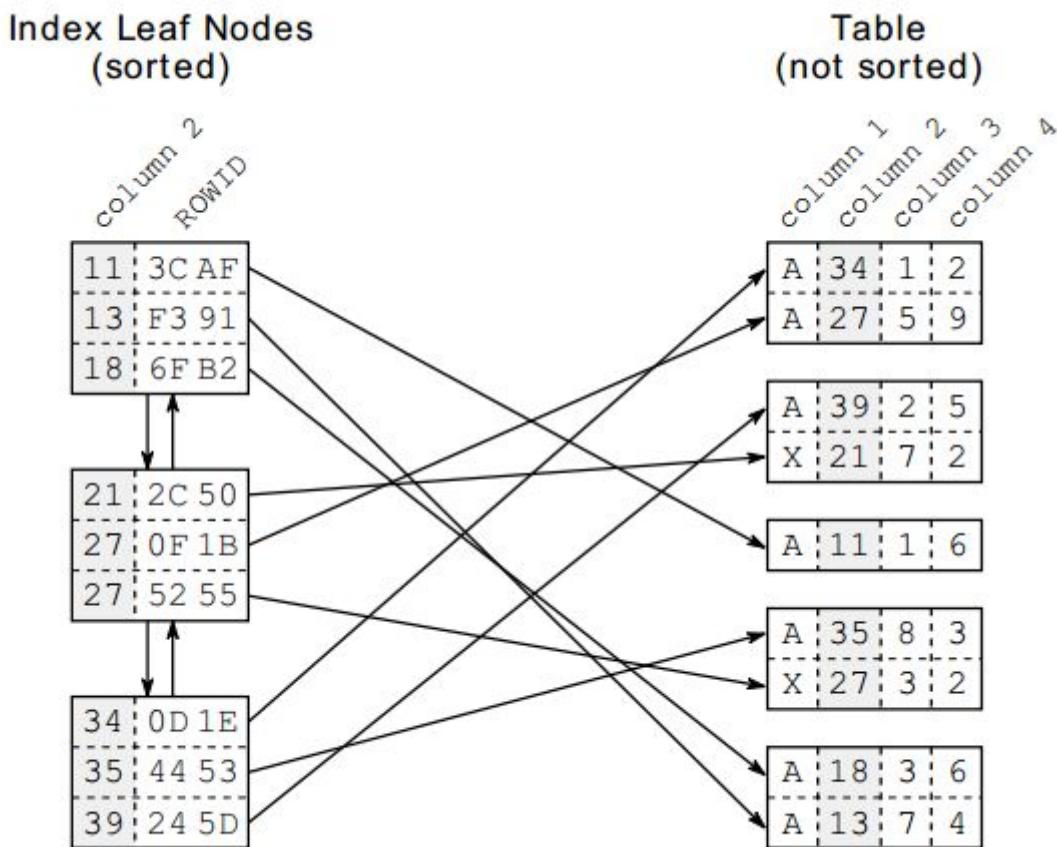
Danh sách liên kết đôi cũng được sử dụng trong nhiều ngôn ngữ lập trình (Hình 1)

Programming Language	Name
Java	<code>java.util.LinkedList</code>
.NET Framework	<code>System.Collections.Generic.LinkedList</code>
C++	<code>std::list</code>

Hình 1 Danh sách liên kết trong các ngôn ngữ lập trình

Cơ sở dữ liệu sử dụng danh sách liên kết đôi để liên kết các nút lá chỉ mục (index leaf nodes). Mỗi nút lá được lưu trữ trong một đơn vị lưu trữ gọi là database *block* hay *page*. Tất cả các block có cùng kích thước và chỉ khoảng một vài Kilobytes (thông thường là 4KB, bằng với kích thước trang lưu trữ page trên đĩa cứng). Cơ sở dữ liệu sử dụng không gian trong mỗi block để lưu trữ càng nhiều phần tử chỉ mục càng tốt. Điều này có nghĩa rằng trật tự sắp xếp của chỉ mục được duy trì trên hai mức khác

nhau: thứ tự sắp xếp của các phần tử chỉ mục trong mỗi nút lá và giữa các nút lá bằng cách sử dụng danh sách liên kết đôi.

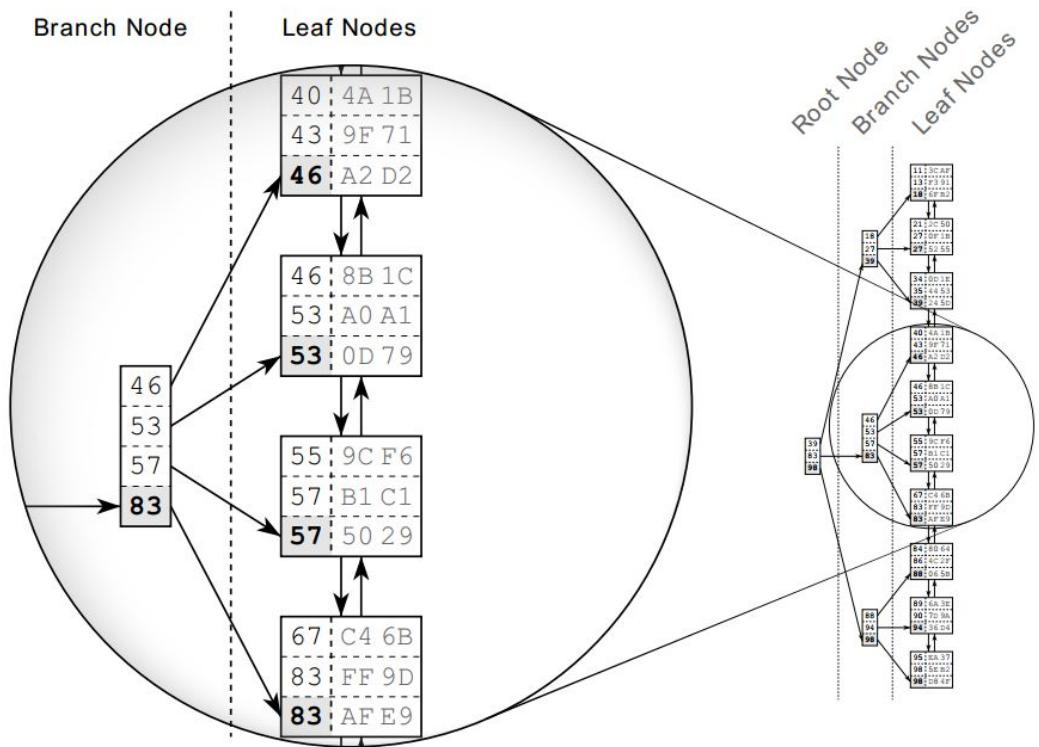


Hình 2 Nút lá chỉ mục và dữ liệu bảng tương ứng

Hình 2 mô tả các nút lá chỉ mục và các kết nối của chúng tới bảng dữ liệu. Mỗi phần tử chỉ mục bao gồm một cột được đánh chỉ mục (*column 2*) và một tham chiếu tới bản ghi tương ứng trên bảng dữ liệu (*ROWID* hay *RID*). Không giống như chỉ mục, bảng dữ liệu được lưu trữ bằng cấu trúc dữ liệu *heap* và không được sắp xếp. Không hề có một liên kết nào giữa các hàng được lưu trữ trong cùng một block hay bất kỳ kết nối nào giữa các block.

4. Cây tìm kiếm (B-tree)

Các nút lá chỉ mục được lưu trữ theo một thứ tự tùy ý tức là vị trí trên ổ đĩa không tương ứng với vị trí logic khi đánh chỉ mục. Nó giống như cuốn sổ danh bạ điện thoại với các trang bị xáo trộn. Nếu bạn muốn tìm kiếm “Smith” nhưng khi mở quyển danh bạ bạn thấy “Robinson” thì điều đó không đảm bảo rằng nếu bạn tiếp tục tìm kiếm tuần tự sẽ thấy được “Smith”. Cơ sở dữ liệu cần một cấu trúc dữ liệu thứ hai để có thể tìm được một mục trong các trang bị xáo trộn một cách nhanh nhất đó là một *cây tìm kiếm cân bằng* – gọi tắt là *B-tree*.



Hình 3 Cấu trúc B-tree

Hình 3 là ví dụ về một index với 30 phần tử. Danh sách liên kết đôi thiết lập các thứ tự logic giữa các nút lá. Các nút gốc (root nodes) và các nút nhánh (branch nodes) hỗ trợ tìm kiếm nhanh chóng các nút lá.

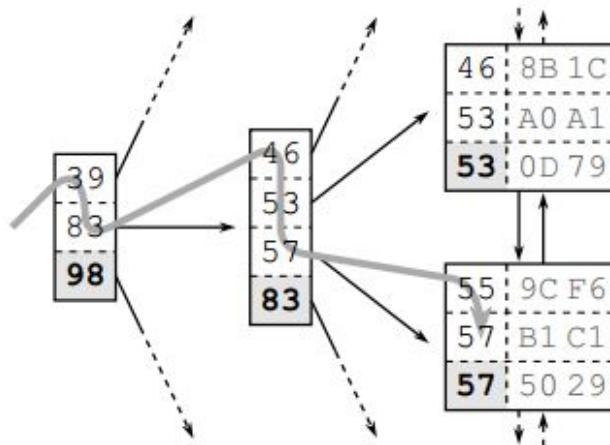
Phần phóng to của hình mô tả một nút nhánh và các nút lá mà nó trỏ tới. Mỗi phần tử của nút nhánh tương ứng với giá trị lớn nhất trên nút lá. Do đó, với nút lá đầu tiên, giá trị lớn nhất là 46 vì thế phần tử đầu tiên trên nút nhánh chính là 46. Điều này cũng đúng với các nút lá còn lại nên nút nhánh có các giá trị là 46, 53, 57, 83. Các nút nhánh được tạo ra cho đến khi mọi nút lá đều có một nút nhánh trỏ tới.

Lớp (layer) tiếp theo được xây dựng tương tự nhưng dựa trên các nút nhánh đã xây dựng. Quá trình lặp đi lặp lại cho đến khi chỉ còn 1 nút – chính là nút gốc. Cấu trúc này được gọi là *cây cân bằng* vì độ sâu của cây bằng nhau tại mọi vị trí; khoảng cách giữa nút gốc và nút lá là giống nhau tại mọi nút lá

Chú ý

Một B-tree là một cây cân bằng chứ không phải cây nhị phân

Sau khi được tạo, cơ sở dữ liệu phải duy trì index một cách tự động. Mọi thao tác insert, delete, update đều phải cập nhật index và giữ cho cây luôn ở trạng thái cân bằng. Tức là, quá trình duy trì sự cân bằng của cây luôn xảy ra khi có thao tác ghi. Chương 8 sẽ trình bày về vấn đề này.



Hình 4 Quá trình duyệt B-tree

Hình 4 mô tả quá trình tìm kiếm cho từ khoá là “57”. Quá trình duyệt cây bắt đầu từ nút gốc phía bên trái. Khoá sẽ được so sánh lần lượt với các phần tử trong nút theo thứ tự tăng dần cho đến khi có phần tử lớn hơn hoặc bằng (\geq) với khoá (57). Như trong hình thì phần tử đó chính là 83. Từ đó cơ sở dữ liệu sẽ được trả tới nút nhánh tương ứng và lặp lại quá trình như trên cho đến khi tới nút lá.

Duyệt cây là một thao tác khá hiệu quả, nó thể hiện quyền năng của cấu trúc đánh chỉ mục. Sử dụng index trả về kết quả rất nhanh ngay cả khi với một tập dữ liệu rất lớn. Điều này bởi vì cây cân bằng cho phép truy cập tất cả các phần tử với số lượng các bước như nhau và độ sâu của cây chỉ tăng theo hàm logarit. Độ sâu của cây tăng rất chậm so với số lượng các nút lá. Trong thực tế quá trình đánh chỉ mục thực hiện trên hàng triệu bản ghi thì độ sâu của cây cũng chỉ là 4 hoặc 5. Độ sâu là 6 cực kỳ hiếm gặp. Mục “Khả năng mở rộng của hàm Logarit” phía dưới sẽ diễn tả chi tiết hơn về điều này.

5. Vấn đề sử dụng index nhưng vẫn chậm, phần 1

Mặc dù quá trình duyệt cây là rất hiệu quả, tuy nhiên vẫn còn một trường hợp index không làm việc như mong đợi. Điều này đã đưa ra một tranh cãi về “sự thoái hoá của chỉ mục – degenerated index” trong một thời gian dài mà cách giải quyết (gây tranh cãi) đơn giản là đánh chỉ mục lại từ đầu. Lý do các câu truy vấn chậm ngay cả khi được đánh chỉ mục có thể được giải thích dựa trên các nguyên lý đã trình bày ở phần trước.

Điều đầu tiên khiến cho một tra cứu chỉ mục chậm đi là do chuỗi các nút lá. Ta xem xét lại ví dụ trong Hình 4 với quá trình tìm kiếm “57”. Rõ ràng ta thấy có hai phần tử khớp trong chỉ mục. Có ít nhất 2 phần tử giống nhau hay nói chính xác hơn là: có thể có nhiều phần tử ở các nút lá tiếp theo cũng có giá trị là “57”. Cơ sở dữ liệu phải đọc các nút lá tiếp theo để xem nếu có bất kỳ phần tử nào khớp với giá trị cần tìm kiếm. Điều này có nghĩa rằng một quá trình tra cứu chỉ mục không chỉ cần thực hiện việc duyệt cây mà cũng cần thực hiện việc duyệt trên chuỗi các nút lá.

Điều thứ hai có thể làm một tra cứu chỉ mục chậm đi là quá trình truy cập bảng. Một nút lá bao gồm nhiều nút chỉ mục, thường hàng trăm, mỗi nút chỉ mục trả tới một dòng dữ liệu trong bảng ở những vị trí lưu trữ khác nhau. Bảng dữ liệu tương ứng thường nằm rải rác trên nhiều block (Hình 2).

Một quá trình tra cứu chỉ mục gồm 3 bước: (1) quá trình duyệt cây; (2) duyệt các nút lá kế tiếp; (3) lấy dữ liệu từ bảng. Quá trình duyệt cây là bước duy nhất có giới hạn trên cho số lần truy cập các block – độ sâu của cây. Hai bước còn lại có thể truy cập nhiều block – chúng là nguyên nhân tại sao quá trình tra cứu chỉ mục chậm đi.

Khả năng mở rộng của hàm Logarit

Trong toán học, logarit của một số với một cơ số cho trước là số mũ của cơ số dùng trong phép luỹ thừa để có thể tạo ra số ấy.

Trong một cây tìm kiếm, cơ số tương ứng với số phần tử trên một nút nhánh và mũ chính là độ sâu của cây. Ví dụ, chỉ mục trong Hình 3 có 4 phần tử trong một nút và có độ sâu là 3. Điều này có nghĩa rằng chỉ mục có thể chứa $64 (4^3)$ phần tử. Nếu nó được tăng lên một mức nữa thì nó có thể chứa được 256 phần tử (4^4). Mỗi lần có một mức mới được thêm vào thì số lượng các phần tử có thể được đánh chỉ mục tăng 4 lần. Còn độ sâu của cây chỉ là $\log_4(\text{số-phần-tử}-\text{được}-\text{đánh}-\text{chỉ}-\text{mục})$

Sự tăng chậm của hàm logarit cho phép chúng ta có thể đánh chỉ mục cho hơn một triệu bản ghi nhưng chỉ với độ sâu của cây là 10, trong thực tế còn có thể hiệu quả hơn. Yếu tố chính ảnh hưởng đến độ sâu của cây và hiệu quả của quá trình tra cứu chỉ mục chính là số lượng phần tử có trong mỗi nút của cây. Đó chính là cơ số trong hàm logarit. Cơ số càng cao thì cây càng thấp và duyệt càng nhanh.

Độ sâu của cây	Số phần tử được đánh chỉ mục
3	64
4	256
5	1024
6	4096
7	16384
8	65536
9	262144
10	1048576

Cơ sở dữ liệu khai thác khái niệm này ở mức tối đa và đưa nhiều phần tử nhất có thể vào mỗi nút – thường là hàng trăm. Điều này có nghĩa rằng mỗi khi độ sâu tăng 1 thì số phần tử lưu trữ tăng lên hàng trăm lần.

Nguồn gốc của câu chuyện “chỉ mục chậm” do người ta nghĩ rằng quá trình tra cứu chỉ mục chỉ là quá trình duyệt cây, vì vậy mà dẫn đến lầm tưởng rằng quá trình tra cứu chỉ mục chậm là do cây bị “hỏng” hoặc “mất cân bằng” gây ra. Trong thực tế bạn có thể yêu cầu các cơ sở dữ liệu đưa ra cách mà chúng sử dụng index. Oracle database khá rõ ràng về khía cạnh này. Nó có 3 thao tác mô tả một quá trình tra cứu chỉ mục cơ bản

INDEX UNIQUE SCAN

INDEX UNIQUE SCAN chỉ thực hiện duyệt cây. Oracle database sử dụng thao tác này nếu có một ràng buộc duy nhất chắc chắn rằng các tiêu chí tìm kiếm sẽ chỉ khớp với duy nhất một phần tử

INDEX RANGE SCAN

INDEX RANGE SCAN thực hiện duyệt cây và duyệt tiếp các nút lá theo danh sách liên kết đôi để tìm tất cả các phần tử khớp với yêu cầu tìm kiếm. Đây là thao tác được thực hiện nếu có nhiều phần tử khớp với các tiêu chí tìm kiếm

TABLE ACCESS BY INDEX ROWID

TABLE ACCESS BY INDEX ROWID truy xuất vào các dòng trong bảng dữ liệu với ROWID. Thao tác này được thực hiện sau khi tìm được ROWID của các bản ghi phù hợp từ các thao tác INDEX SCAN trước đó (UNIQUE và RANGE).

Điều quan trọng là một INDEX RANGE SCAN có nguy cơ phải đọc một phần khá lớn của index. Nếu có một hoặc nhiều lần truy cập bảng cho mỗi dòng kết quả trả về thì câu truy vấn có thể rất chậm ngay cả khi sử dụng chỉ mục.

6. Chương 2

7. Mệnh đề WHERE

Chương trước mô tả cấu trúc của chỉ mục (index) và giải thích khả năng xuất hiện hiện tượng sử dụng index mà truy vấn vẫn chậm. Trong phần tiếp theo, chúng ta sẽ chỉ ra cách phát hiện và tránh những vấn đề trong câu truy vấn SQL. Chúng ta bắt đầu từ việc tìm hiểu mệnh đề Where.

Mệnh đề Where định nghĩa điều kiện tìm kiếm của câu truy vấn SQL. Do đó, mệnh đề Where liên quan khá chặt chẽ đến vai trò của chỉ mục: giúp CSDL tìm dữ liệu một cách nhanh chóng. Mặc dù mệnh đề Where có một sự ảnh hưởng lớn đến hiệu năng thực thi truy vấn, nhưng nó thường không được để ý, có nhiều trường hợp mà mệnh đề Where trong truy vấn khiến hệ quản trị CSDL phải quét phần lớn cấu trúc chỉ mục. Kết quả: một cách viết mệnh đề Where kém hiệu quả làm chậm tốc độ truy vấn.

Chương này giải thích sự khác nhau của các toán tử ảnh hưởng đến việc dùng chỉ mục và cách dùng chỉ mục cho nhiều câu truy vấn nhất có thể. Phần cuối trình bày một vài cách viết mệnh đề Where không hiệu quả và cách viết lại một mệnh đề tốt hơn.

8. Toán tử bằng

Toán tử so sánh bằng được sử dụng thường xuyên nhất trong các toán tử SQL. Việc tạo ra và sử dụng chỉ mục không phù hợp ảnh hưởng nhiều đến hiệu năng và mệnh đề Where bao gồm nhiều điều kiện thì dễ cho kết quả không mong muốn.

Phần này sẽ chỉ ra cách kiểm chứng câu truy vấn có sử dụng index hay không và giải thích công dụng của chỉ mục và công dụng của chỉ mục kết hợp – concatenated index có khả năng tối ưu hóa các mệnh đề điều kiện kết hợp. Để hiểu hơn, chúng ta sẽ phân tích những câu truy vấn chậm để thấy sự các nhân tố ảnh hưởng tới tốc độ truy vấn sử dụng index trong Chương 1.

9. Khóa chính

Chúng ta bắt đầu với mệnh đề Where đơn giản nhất: chứa khóa chính. Chúng ta sử dụng bảng EMPLOYEES được định nghĩa dưới đây cho những ví dụ của chương này:

```
CREATE TABLE employees (
    employee_id NUMBER           NOT NULL,
    first_name   VARCHAR2(1000)  NOT NULL,
    last_name    VARCHAR2(1000)  NOT NULL,
    date_of_birth DATE            NOT NULL,
    phone_number VARCHAR2(1000)  NOT NULL,
    CONSTRAINT employees_pk PRIMARY KEY (employee_id)
);
```

Thông thường, CSDL tự động tạo một chỉ mục index cho khóa chính. Điều đó có nghĩa là có một chỉ mục (index) trong cột EMPLOYEE_ID, mặc dù không có câu lệnh tạo chỉ mục (index) nào

Câu truy vấn dưới đây sử dụng khóa chính để lấy ra tên của employees

```
SELECT first_name, last_name  
  FROM employees  
 WHERE employee_id = 123
```

Mệnh đề Where ở đây sẽ không truy xuất nhiều bản ghi, vì ràng buộc khóa chính đảm bảo sự duy nhất của giá trị EMPLOYEE_ID. CSDL không cần duyệt theo danh sách liên kết trên các nút lá - nó chỉ cần duyệt qua cây chỉ mục là đủ. Chúng ta có thể xem kế hoạch thực thi (execution plan) của câu truy vấn này:

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	2
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	2
*2	INDEX UNIQUE SCAN	EMPLOYEES_PK	1	1

Predicate Information (identified by operation id):

2 - access("EMPLOYEE_ID"=123)

“execution plan” của Oracle chỉ ra một INDEX UNIQUE SCAN - là phép duyệt cây chỉ mục. Số lượng các nút chỉ mục cần duyệt là chiêu cao của cây – tốc độ thực thi hầu như không phụ thuộc vào kích thước bảng.

Mẹo

Kế hoạch thực thi - execution plan chỉ ra các bước cơ sở dữ liệu thực hiện một câu truy vấn SQL. ~~Phụ lục A ở trang 165 giải thích làm thế nào để lấy và đọc những “Execution plan” với cơ sở dữ liệu khác.~~

Sau khi truy cập vào chỉ mục, cơ sở dữ liệu phải làm thêm một bước để lấy dữ liệu cần truy vấn (FIRST_NAME, LAST_NAME) từ bảng dữ liệu lưu trữ: toán tử TABLE ACCESS BY INDEX ROWID. Toán tử này có thể trở thành điểm thắt cổ chai (tắc nghẽn về hiệu năng) như giải thích trong “Vấn đề sử dụng chỉ mục nhưng chậm, phần 1”. Nhưng trong câu truy vấn cụ thể ở trên, vấn đề truy vấn chậm sẽ không xảy ra đi cùng với INDEX UNIQUE SCAN. Vì INDEX UNIQUE SCAN không thể trả về nhiều hơn một phần tử trong chỉ mục, do đó không thể truy cập nhiều hơn một dòng trong bảng. Điều đó có nghĩa rằng INDEX UNIQUE SCAN không xuất hiện trong kế hoạch thực thi của một câu truy vấn chậm.

Khóa chính không cùng với chỉ mục ràng buộc đơn nhất (unique index):

Một khóa chính không cần thiết sử dụng một chỉ mục ràng buộc đơn nhất (unique index) - bạn có thể sử dụng chỉ mục không có ràng buộc đơn nhất (non-unique index). Trong trường hợp đó cơ sở dữ liệu Oracle không sử dụng INDEX UNIQUE SCAN nhưng thay vào đó là phép INDEX RANGE SCAN. Tuy nhiên, ràng buộc khóa chính vẫn đảm bảo tính duy nhất do đó việc tìm kiếm trên chỉ mục luôn luôn trả về nhiều nhất một bản ghi (dòng).

Một trong những lý do cho việc sử dụng những chỉ mục non-unique cho khóa chính là những ràng buộc trì hoãn (**defferable constraints**). Khác với những ràng buộc thông thường, được kiểm tra thường xuyên trong suốt quá trình thực thi câu truy vấn, cơ sở dữ liệu hoãn lại việc kiểm tra những ràng buộc trì hoãn cho đến khi giao dịch được thiết lập. Những ràng buộc trì hoãn là bắt buộc cho dữ liệu chèn vào trong bảng với phụ thuộc vòng (circular dependencies).

10. Chỉ mục kết hợp (concatenated index)

Mặc dù cơ sở dữ liệu tạo chỉ mục cho khóa chính một cách tự động, chúng ta vẫn có thể tạo chỉ mục tối ưu hơn thay thế cho chỉ mục được tạo ra tự động. Trong trường hợp, khóa chính gồm nhiều cột, cơ sở dữ liệu tạo một chỉ mục kết hợp cho toàn bộ cột trong khóa chính - được gọi là chỉ mục kết hợp (concatenated, multicolumn, composite, combined index). Thứ tự cột của chỉ mục kết hợp có sự ảnh hưởng lớn đến tính hiệu quả hay khả năng sử dụng của chỉ mục đó, do đó nó phải được lựa chọn cẩn thận.

Để chứng minh cho điều này, hãy xem xét dữ liệu một cuộc sáp nhập công ty. Nhân viên từ các công ty khác được thêm vào bảng EMPLOYEES trong CSDL của công ty chính, và bảng này trở nên lớn gấp 10 lần. Vấn đề xảy ra khi EMPLOYEE_ID không còn là khóa chính để phân biệt các nhân viên (EMPLOYEE_ID trùng nhau đối với các công ty khác nhau). Chúng ta cần mở rộng khóa chính bởi một định danh mở rộng. Vd: một khóa mới có 2 cột: EMPLOYEE_ID và SUBSIDIARY_ID để thiết lập lại ràng buộc duy nhất.

Chỉ mục cho một khóa chính mới được định nghĩa như cách dưới đây:

```
CREATE UNIQUE INDEX employee_pk  
    ON employees (employee_id, subsidiary_id);
```

Một truy vấn cho một nhân viên bất kỳ phải sử dụng đầy đủ khóa chính, ngoài EMPLOYEE_ID, SUBSIDIARY_ID cũng phải được sử dụng.

```
SELECT first_name, last_name  
    FROM employees  
   WHERE employee_id = 123  
     AND subsidiary_id = 30;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	2
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	2
*2	INDEX UNIQUE SCAN	EMPLOYEES_PK	1	1

Predicate Information (identified by operation id):

2 - access("EMPLOYEE_ID"=123 AND "SUBSIDIARY_ID"=30)

Khi một câu truy vấn chứa đầy đủ khóa chính, CSDL có thể dùng phép duyệt chỉ mục duy nhất (INDEX UNIQUE SCAN) mà không quan tâm đến số lượng cột nằm trong chỉ mục. Điều gì sẽ xảy ra nếu truy vấn chỉ dùng một cột của khóa chính? Ví dụ lấy ra tất cả nhân viên của một chi nhánh, câu truy vấn sẽ là:

```
SELECT first_name, last_name
  FROM employees
 WHERE subsidiary_id = 20;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		106	478
* 1	TABLE ACCESS FULL	EMPLOYEES	106	478

Predicate Information (identified by operation id):

1 - filter("SUBSIDIARY_ID"=20)

Kế hoạch thực thi chỉ ra rằng CSDL sẽ không sử dụng chỉ mục. Thay vào đó là thực hiện quét toàn bộ bảng (FULL TABLE SCAN). Kết quả là CSDL sẽ đọc cả bảng như đầu vào và đổi chiều từng bản ghi với điều kiện trong WHERE. Thời gian thực thi tương ứng với kích thước của bảng, trong ví dụ này sẽ là gấp 10 lần. Phép quét chạy đủ nhanh đối với môi trường lập trình phát triển, nhưng sẽ gây ra những vấn đề nghiêm trọng về hiệu năng trong môi trường chạy thực tế (production).

QUÉT TOÀN BẢNG (FULL TABLE SCAN):

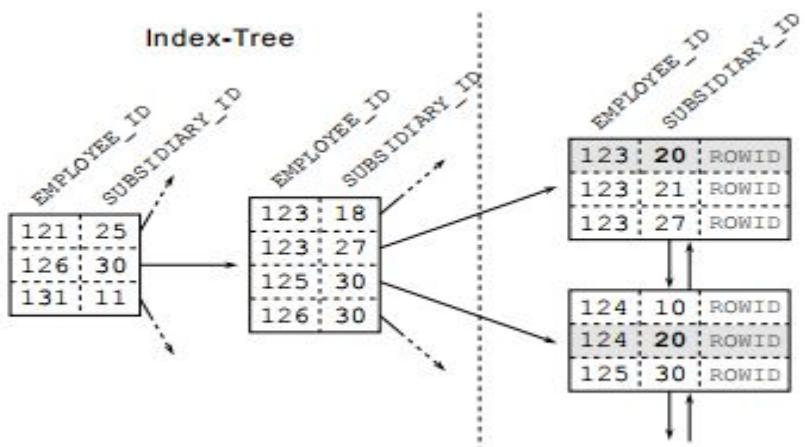
Phép quét toàn bảng có thể rất hiệu quả trong một số trường hợp nhất định. Cụ thể khi cần truy xuất lượng lớn dữ liệu trong bảng, việc tìm kiếm dựa trên chỉ mục có thể sẽ tồi hơn rất nhiều so với phép quét toàn bảng. Lý do là phép duyệt trên chỉ mục sẽ phải đọc các khối dữ liệu (tuần tự logic theo thứ tự đánh chỉ mục), các khối có vị trí rất khác nhau trên thiết bị lưu trữ, yêu cầu nhiều thao tác di chuyển đầu đọc (đĩa cứng). Quét toàn bảng duyệt tuần tự hết cả bảng do đó CSDL sẽ hạn chế thao tác di chuyển đầu đọc, dù cho CSDL phải đọc nhiều dữ liệu hơn nhưng lại thực thi ít toán tử đọc hơn.

Từ ví dụ ở trên ta thấy CSDL sẽ không dùng chỉ mục nếu truy vấn sử dụng một cách tùy ý cột đơn trong chỉ mục kết hợp. Cấu trúc chỉ mục kết hợp sẽ giúp điều này sáng tỏ hơn.

Một chỉ mục kết hợp là một chỉ mục có cấu trúc B-tree (cây cân bằng), cũng giống như các chỉ mục khác, nó sẽ sắp xếp các dữ liệu được đánh chỉ mục. Việc sắp xếp dữ liệu này dựa trên thứ tự các cột được định nghĩa trong chỉ mục kết hợp. Đầu tiên CSDL sắp xếp các bản ghi theo cột thứ nhất, khi có 2 bản ghi cùng giá trị ở cột thứ nhất thì sẽ sắp xếp theo cột thứ 2, cứ như vậy cho đến hết các cột của chỉ mục kết hợp.

Quan trọng

Một chỉ mục kết hợp là một chỉ mục trên nhiều cột. Thự tự các bản ghi của chỉ mục kết hợp gồm 2 cột giống như thứ tự trong danh bạ điện thoại: đầu tiên các mục được xếp theo tên, nếu cùng tên sẽ sắp xếp theo họ. Điều đó cũng có nghĩa chỉ mục kết hợp này sẽ không hỗ trợ nếu chỉ tìm theo họ, mà sẽ tiện lợi khi tìm theo tên.



Hình 5 Chỉ mục kết hợp

Chỉ mục trên Hình 5 chỉ ra rằng các bản ghi cho mã chi nhánh SUBSIDIARY_ID= 20 không được sắp xếp liền nhau. Và hiển nhiên rằng cây cũng không có đầu vào với SUBSIDIARY = 20, dù cho chúng nằm ở trên các nút lá (lưu ý đây là cây chỉ mục thưa (spare index), một con trỏ trả tới 1 trang lưu trữ, không trả tới một dòng dữ liệu cụ thể). Cây chỉ mục không có tác dụng đối với truy vấn này.

Mẹo

Bạn có thể hiển thị trực quan cách chỉ mục sắp xếp dữ liệu bằng cách thực thi truy vấn các cột được đánh chỉ mục, sắp xếp theo thứ tự mặc định theo thứ tự index.

```
SELECT < Các cột trong chỉ mục cần xem xét >
FROM < TABLE >
ORDER BY < Thứ tự các cột trong index đang xep xet >
FETCH FIRST 100 ROWS ONLY;
```

Với kết quả trả về, bạn có thể xác định là với câu truy vấn cụ thể mà các bản ghi thỏa mãn không nằm liên tiếp theo cụm thì tức là việc sử dụng chỉ mục cho câu truy vấn cụ thể đó có thể không thực sự hiệu quả.

Trong ví dụ trên, tất nhiên có thể tạo thêm 1 chỉ mục khác trên cột SUBSIDIARY_ID để cải thiện tốc độ truy vấn. Tuy nhiên có một giải pháp tốt hơn, giả định rằng việc tìm kiếm sử dụng duy nhất thuộc tính EMPLOYEE_ID là không có ý nghĩa.

Chúng ta có thể tận dụng một điều chắc chắn rằng cột đầu tiên trong chỉ mục kết hợp luôn hữu dụng cho việc truy vấn. Quay lại ví dụ về danh bạ điện thoại: bạn không cần phải biết tên mà chỉ cần tìm kiếm theo họ. Từ đó ta có thể đặt SUBSIDIARY_ID lên vị trí đầu tiên trong chỉ mục:

```
CREATE UNIQUE INDEX EMPLOYEES_PK
ON EMPLOYEES (SUBSIDIARY_ID, EMPLOYEE_ID);
```

Hai cột kết hợp theo thứ tự này vẫn đảm bảo tính duy nhất nên truy vấn trên đầy đủ khóa chính có thể dùng tới INDEX UNIQUE SCAN nhưng thứ tự thuộc tính trong chỉ mục đã thay đổi. SUBSIDIARY_ID trở thành thuộc tính đầu tiên cho việc sắp xếp dữ

liệu. Điều này có nghĩa là tất cả các dòng dữ liệu thuộc cùng một mã chi nhánh SUBSIDIARY_ID sẽ được sắp xếp gần nhau thành nhóm, thứ tự trong mỗi nhóm được xác định bởi cột thứ 2 trong chỉ mục là EMPLOYEE_ID.

Quan Trọng:

Điều cực kì quan trọng khi định nghĩa một chỉ mục kết hợp là việc chọn thứ tự của các cột để đảm bảo chỉ mục có thể được tận dụng một cách tối đa.

Bảng kế hoạch thực dưới đây xác nhận việc thực thi truy vấn có sử dụng chỉ mục. Cột SUBSIDIARY_ID không mang tính duy nhất nên CSDL phải duyệt thêm trên các nút lá để tìm ra bản ghi phù hợp: sử dụng phép quét vùng chỉ mục (INDEX RANGE SCAN)

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT			
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	106	75
*2	INDEX RANGE SCAN	EMPLOYEE_PK	106	2

Predicate Information (identified by operation id):

2 - access("SUBSIDIARY_ID"=20)

Nhận xét chung, CSDL có thể sử dụng chỉ mục kết hợp khi tìm kiếm trên cột đầu tiên của chỉ mục kết hợp. Một chỉ mục kết hợp trên 3 cột có thể sử dụng khi truy vấn chưa cột đầu hoặc 2 cột đầu hoặc tất cả các cột.

Cho dù giải pháp dùng chỉ mục kết hợp khá hiệu quả trong câu truy vấn SELECT, thế nhưng chỉ mục đơn có 1 cột vẫn được ưa thích hơn. Điều này không chỉ tiết kiệm bộ nhớ, mà còn tránh việc quá tải cho chỉ mục thứ 2. Số lượng chỉ mục càng ít, hiệu năng của các lệnh INSERT, DELETE và UPDATE càng tốt.

Để định nghĩa một chỉ mục tốt, chúng ta không chỉ cần hiểu rõ cách chỉ mục hoạt động mà còn phải hiểu cách ứng dụng truy vấn dữ liệu. Có nghĩa rằng bạn phải biết mối liên hệ giữa các cột trong điều kiện WHERE.

Định nghĩa một chỉ mục tốt là việc khó khăn đối với người ngoài (không tham gia phát triển ứng dụng khai thác CSDL), bởi vì họ không có cái nhìn tổng hợp về cách mà ứng dụng tiếp cận dữ liệu. Họ có thể luôn sử dụng chỉ một truy vấn. Họ cũng không quan tâm nhiều về hiệu quả của chỉ mục đem lại đối với những câu truy vấn khác. Người quản trị CSDL có thể biết rõ về lược đồ CSDL nhưng không nắm rõ được cách tiếp cận dữ liệu của ứng dụng.

Phòng phát triển ứng dụng có lẽ là nơi duy nhất mà có cả kiến thức về kĩ thuật trong CSDL và kiến thức về ứng dụng sử dụng CSDL đó. Lập trình viên biết cách sử dụng dữ liệu cũng như cách tiếp cận chúng. Họ có thể tận dụng lợi ích của chỉ mục một cách tối ưu và khôn ngoan.

11. Sử dụng chỉ mục nhưng truy vấn vẫn chậm, phần 2

Phần trước đã giải thích phương pháp để khai thác tối đa hiệu quả một chỉ mục kết hợp đã có bằng việc thay đổi thứ tự cột của nó, nhưng ví dụ được nhắc đến chỉ là 2 câu lệnh SQL. Tuy nhiên, sự thay đổi của một chỉ mục có thể tác động đến toàn bộ các truy vấn trên bảng sử dụng nó. Phần này sẽ giải thích xem cách các CSDL chọn lựa chỉ mục như thế nào, đồng thời làm rõ những tác dụng phụ có thể gặp phải khi thay đổi các chỉ mục hiện tại.

Việc sửa lại chỉ mục EMPLOYEE_PK sẽ cải thiện hiệu suất của tất cả các câu truy vấn được tìm kiếm theo chi nhánh– subsidiary. Tuy nhiên, nó cũng có thể dùng cho các câu truy vấn tìm kiếm theo SUBSIDIARY_ID – kết hợp với các điều kiện khác trong mệnh đề WHERE. Do đó, bộ tối ưu truy vấn sẽ có thêm các phương án thực thi khi có thêm chỉ mục mới hoặc chỉ mục cũ được cập nhật, vai trò của bộ tối ưu là tìm ra phương án thực thi tối ưu.

Xử lý truy vấn, bộ tối ưu hoá câu truy vấn

Thành phần xử lý truy vấn, hay người lập kế hoạch truy vấn, là thành phần cơ sở dữ liệu chuyển đổi một câu lệnh SQL thành một kế hoạch thực thi. Quá trình này cũng được gọi là *bien dịch* hay *phân tích cú pháp* (*tránh nhầm lẫn với bước phân tích cú pháp là một bước con trong xử lý truy vấn*). Có 2 loại trình tối ưu khác nhau.

Trình tối ưu hoá dựa trên chi phí (CBO) tạo ra nhiều biến thể của kế hoạch thực thi và tính toán một giá trị chi phí cho mỗi kế hoạch. Việc tính toán chi phí dựa trên việc thực hiện sử dụng và ước lượng số lượng các hàng. Cuối cùng, giá trị chi phí dùng làm mốc để chọn lựa kế hoạch thực thi “tốt nhất”.

Trình tối ưu hoá dựa trên quy tắc (RBO) tạo ra kế hoạch thực thi bằng việc sử dụng một bộ quy tắc cứng (theo kinh nghiệm). Ngày nay, trình tối ưu hoá này kém linh hoạt và cũng hiếm khi được sử dụng hơn.

Việc thay đổi một chỉ mục có thể gặp những tác dụng phụ. Tiếp tục xét trong ví dụ trên, ứng dụng danh bạ điện thoại nội bộ đã trở nên rất chậm từ sau khi sáp nhập. Phân tích đầu tiên đã xác định truy vấn dưới đây là nguyên nhân dẫn đến giảm tốc độ:

```
SELECT first_name, last_name, subsidiary_id, phone_number
  FROM employees
 WHERE last_name = 'WINAND'
   AND subsidiary_id = 30;
```

Kế hoạch thực thi của truy vấn này là:

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	30
*1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	30
*2	INDEX RANGE SCAN	EMPLOYEES_PK	40	2

Predicate Information (identified by operation id):

- 1 - filter("LAST_NAME"='WINAND')
- 2 - access("SUBSIDIARY_ID"=30)

Ví dụ 1 Kế hoạch thực thi với chỉ mục trên khóa chính thay đổi

Kế hoạch thực thi trên sử dụng chỉ mục, có tổng giá trị chi phí là 30. Tính đến thời điểm hiện tại thì mọi việc đều tốt đẹp. Tuy nhiên, nó đang sử dụng chỉ mục mà chúng ta vừa mới thay đổi – do đó chúng ta có thể nghi ngờ rằng sự thay đổi của chỉ mục gây ra vấn đề về hiệu năng, đặc biệt là khi ta nhớ lại việc chọn chỉ mục cũ – nó bắt đầu với cột EMPLOYEE_ID, không phải là một phần của bất kì mệnh đề WHERE nào trong câu truy vấn trên (câu truy vấn trên đã không thể sử dụng chỉ mục này trước đó)

Để phân tích sâu hơn, tốt nhất là ta nên so sánh kế hoạch thực thi trước và sau khi thay đổi. Để có được kế hoạch thực thi ban đầu, chúng ta có thể chỉ cần triển khai lại việc xác định chỉ mục cũ, tuy nhiên hầu hết các cơ sở dữ liệu cung cấp một phương thức đơn giản hơn để ngăn ngừa sử dụng một chỉ mục cho một truy vấn cá biệt. Ví dụ dưới đây sử dụng một gợi ý tối ưu của Oracle cho mục đích đó.

```
SELECT /*+ NO_INDEX(EMPLOYEES EMPLOYEE_PK) */
       first_name, last_name, subsidiary_id, phone_number
    FROM employees
   WHERE last_name = 'WINAND'
     AND subsidiary_id = 30;
```

Kế hoạch thực thi tương ứng trước khi sử dụng chỉ mục:

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	477
* 1	TABLE ACCESS FULL	EMPLOYEES	1	477

Predicate Information (identified by operation id):

- 1 - filter("LAST_NAME"='WINAND' AND "SUBSIDIARY_ID"=30)

Mặc dù TABLE ACCESS FULL phải đọc và xử lý toàn bộ bảng nhưng nó nhanh hơn việc sử dụng chỉ mục trong trường hợp này. Điều này đặc biệt bất thường vì truy vấn chỉ khớp với một hàng duy nhất. Sử dụng một chỉ mục để tìm ra một hàng đơn lẻ đáng ra phải nhanh hơn nhiều so với quét toàn bộ bảng, nhưng ở đây thì không. Chỉ mục có vẻ chậm (mặc dù chi phí được tính cost thấp hơn nhưng chi phí thực thi thực tế lại nhiều hơn).

Trong những trường hợp như vậy tốt nhất ta nên thực hiện từng bước của kế hoạch thực thi phức tạp. Bước đầu tiên, INDEX RANGE SCAN trên chỉ mục EMPLOYEES_PK. Chỉ mục này không bao phủ cột LAST_NAME – INDEX RANGE SCAN chỉ có thể xét đến bộ SUBSIDIARY_ID; Cơ sở dữ liệu Oracle chỉ ra điều này trong phần “Thông tin Dự đoán – predicate information” – mục “2” của kế hoạch thực thi. Ở đó bạn có thể thấy các điều kiện được áp dụng cho mỗi hoạt động.

Mẹo

Phụ lục A nói đến “Kế hoạch Thực thi”, giải thích cách tìm “Thông tin dự đoán” cho các cơ sở dữ liệu khác.

INDEX RANGE SCAN với ID hoạt động = 2 chỉ áp dụng bộ lọc SUBSIDIARY_ID=30. Điều đó có nghĩa là nó đi qua cây chỉ mục để tìm mục đầu tiên có SUBSIDIARY_ID bằng 30. Tiếp đó, nó đi theo chuỗi nút lá để tìm tất cả các mục khác cho chi nhánh đó. Kết quả của INDEX RANGE SCAN là một danh sách các ROWID đáp ứng điều kiện SUBSIDIARY_ID: tuỳ thuộc vào kích thước của chi nhánh Subsidiary, có thể chỉ có một vài cũng có thể có tới hàng trăm.

Bước tiếp theo là thực hiện TABLE ACCESS BY INDEX ROWID. Nó sử dụng các ROWID từ bước trước để lấy ra các hàng – tất cả các cột – từ bảng. Khi có thể sử dụng cột LAST_NAME, cơ sở dữ liệu có thể ước lượng phần còn lại của mệnh đề WHERE. Tức, nó có thể lấy ra được tất cả các hàng có SUBSIDIARY=30 trước khi áp dụng bộ lọc LAST_NAME.

Thời gian phản hồi của truy vấn không phụ thuộc vào kích thước bộ kết quả nhưng lại phụ thuộc vào số lượng nhân viên trong chi nhánh đang xét. Nếu chi nhánh SUBSIDIARY = 30 có ít nhân viên, INDEX RANGE SCAN sẽ có hiệu xuất tốt hơn. Tuy nhiên một TABLE ACCESS FULL - truy cập bảng đầy đủ lại nhanh hơn cho một phép duyệt rời rạc tất cả các nhân viên trong một tập chi nhánh lớn bởi vì nó có thể đọc tuần tự từ đầu bảng (giảm thiểu thao tác di chuyển đầu đọc trên đĩa cứng, nguyên nhân dẫn đến truy xuất dữ liệu bị chậm) (xem lại "Full Table Scan" ở trang ?).

Truy vấn chậm vì tra cứu chỉ mục trả về nhiều ROWIDs – mỗi cho một nhân viên thuộc chi nhánh là một công ty trước khi sát nhập và bộ thực thi truy vấn phải duyệt lần lượt, rồi rạc từng dòng bản ghi. Đó là sự kết hợp hoàn hảo của hai thành phần làm cho một chỉ mục chậm: cơ sở dữ liệu đọc một dải chỉ mục rộng và phải tìm nạp nhiều dòng bản ghi riêng lẻ.

Việc lựa chọn kế hoạch thực thi tốt nhất phụ thuộc vào việc phân phối (lưu trữ vật lý) dữ liệu của bảng cũng như cách bộ tối ưu sử dụng các thông tin thống kê về nội dung

của bảng cơ sở dữ liệu. Trong ví dụ trên, một histogram có chứa phân phối các thành viên của các chi nhánh có thể được sử dụng, cho phép trình tối ưu hóa ước lượng số lượng hàng được trả về từ tra cứu chỉ mục - kết quả được sử dụng cho ước lượng chi phí.

Các thông tin thống kê:

Trình tối ưu hóa dựa trên chi phí, sử dụng số liệu thống kê về bảng, cột và chỉ mục. Hầu hết số liệu thống kê đều được thu thập ở cấp độ cột: số các giá trị khác biệt, giá trị nhỏ nhất và lớn nhất (dải dữ liệu), số lần xuất hiện NULL và biểu đồ cột (dữ liệu phân phối). Giá trị thống kê quan trọng nhất của một bảng là giá trị kích thước (theo hàng và khối (block hay page lưu trữ)).

Các chỉ số thống kê quan trọng nhất đối với chỉ mục là độ sâu cây, số lượng các nút lá, số lượng các khóa riêng biệt và các nhân tố phân nhóm clustering factor (xem Chương 5, "Clustering Data").

Trình tối ưu hóa sử dụng các giá trị này để ước tính độ chọn lọc mệnh đề Where

Nếu không có sẵn số liệu thống kê - ví dụ như bởi vì chúng đã bị xóa, hay trình tối ưu hóa sử dụng các giá trị mặc định. Các số liệu thống kê mặc định của CSDL Oracle để xuất một chỉ số nhỏ với độ chọn lọc trung bình. Chúng dẫn đến ước tính rằng INDEX RANGE SCAN sẽ trả lại 40 hàng. Kế hoạch thực hiện cho thấy ước tính này trong cột Rows (xem ví dụ trước trang ?). Rõ ràng đây là một đánh giá thấp, vì có 1000 thành phần trong tập con này.

Nếu chúng ta cung cấp số liệu thống kê chính xác, trình tối ưu hóa sẽ làm tốt hơn. Các kế hoạch thực hiện sau này cho thấy ước tính mới: 1000 hàng cho INDEX RANGE SCAN. Do đó, nó đã tính giá trị chi phí cao hơn cho truy cập bảng con.

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	680
*1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	680
*2	INDEX RANGE SCAN	EMPLOYEES_PK	1000	4

Predicate Information (identified by operation id):

```
1 - filter("LAST_NAME"='WINAND')
2 - access("SUBSIDIARY_ID"=30)
```

Giá trị chi phí 680 thậm chí cao hơn giá trị chi phí cho việc thực sử dụng FULL TABLE SCAN (477, xem trang ?). Trình tối ưu hóa do đó sẽ tự động chọn FULL TABLE SCAN.

Chúng ta vừa xem xét một ví dụ tiếp theo về sử dụng chỉ mục nhưng chậm, điều này không làm xáu đi những lợi ích về hiệu năng mà việc sử dụng chỉ mục đúng đắn đem

lại. Dĩ nhiên, tìm kiếm trên LASTNAME được hỗ trợ tốt nhất bởi một chỉ mục trên LAST_NAME:

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

Sử dụng chỉ mục mới, trình tối ưu tính toán giá trị chi phí là 3.

Id Operation	Name	Rows	Cost
0 SELECT STATEMENT		1	3
* 1 TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	3
* 2 INDEX RANGE SCAN	EMP_NAME	1	1

Predicate Information (identified by operation id):

```
1 - filter("SUBSIDIARY_ID"=30)
2 - access("LAST_NAME"='WINAND')
```

Ví dụ 2 Kế hoạch thực thi với chỉ mục trên LAST_NAME

Thao tác 2, INDEX RANGE SCAN, theo bộ tối ưu đánh giá chỉ trả về 1 dòng dữ liệu. Việc thực thi chỉ cần đọc 1 dòng trong bảng dữ liệu, điều này chắc chắn là nhanh hơn nhiều so với FULL TABLE SCAN. Một chỉ mục hợp lý chắc chắn tốt hơn là duyệt toàn bộ bảng.

Hai kế hoạch thực hiện từ Ví dụ 1 và Ví dụ 2 gần như giống hệt nhau. Cơ sở dữ liệu thực hiện các hoạt động tương tự và trình tối ưu hóa tính toán các giá trị chi phí tương tự, tuy nhiên ở kế hoạch thứ hai, thực hiện tốt hơn nhiều. Hiệu quả của một INDEX RANGE SCAN có thể thay đổi rất lớn - đặc biệt là khi tiếp theo là một truy cập bảng. Sử dụng index không tự động nghĩa là một câu lệnh được thực hiện theo cách tốt nhất có thể.

12. Hàm

Việc đánh chỉ mục trên trường LAST_NAME đã cải thiện hiệu năng đáng kể, nhưng nó đòi hỏi bạn phải sử dụng cùng kiểu chữ (hoa/thường) như được lưu trữ trong cơ sở dữ liệu khi muốn tìm kiếm. Phần này sẽ làm rõ cách thức loại bỏ hạn chế này mà không làm suy giảm hiệu năng.

Lưu ý

MySQL 5.6 không hỗ trợ lập chỉ mục dựa trên hàm như mô tả dưới đây. Để thay thế, các cột ảo đã được định nghĩa trong MySQL 6.0 nhưng cuối cùng chỉ được giới thiệu trong MariaDB 5.2.

13. Tìm kiếm không phân biệt chữ hoa hay chữ thường sử dụng UPPER hoặc LOWER

Bỏ qua sự phân biệt hoa thường trong mệnh đề WHERE là rất đơn giản. Bạn có thể biến đổi cả 2 vế của biểu thức so sánh thành các kí tự in hoa:

```
SELECT first_name, last_name, phone_number  
FROM employees  
WHERE UPPER(last_name) = UPPER('winand');
```

Bất kể các ký tự cái viết hoa hay viết thường ở trong cụm từ tìm kiếm hoặc trong cột LAST_NAME, hàm UPPER biến đổi chúng thành cùng dạng để so sánh như mong muốn.

Lưu ý

Một cách khác để so sánh không phân biệt chữ hoa hay chữ thường là sử dụng một “bộ đối chiếu”. Các “bộ đối chiếu” mặc định được sử dụng bởi SQL Server và MySQL không phân biệt giữa chữ hoa và chữ thường - chúng đã được mặc định như vậy.

Câu truy vấn trên là hoàn toàn hợp lý về mặt logic nhưng về mặt thực thi thì không như vậy:

<i>Id</i>	<i>Operation</i>	<i>Name</i>	<i>Rows</i>	<i>Cost</i>
0	SELECT STATEMENT		10	477
* 1	TABLE ACCESS FULL	EMPLOYEES	10	477

Predicate Information (identified by operation id):

```
1 - filter(UPPER("LAST_NAME")='WINAND')
```

Chúng ta lại bị quay trở lại với việc quét toàn bộ bảng. Mặc dù có chỉ mục trên trường LAST_NAME, nhưng nó không thể sử dụng được - bởi vì việc tìm kiếm không tìm trên trường LAST_NAME mà tìm trên UPPER(LAST_NAME). Trên góc nhìn từ phía cơ sở dữ liệu, đó là một sự khác biệt hoàn toàn.

Đây là một cái bẫy mà chúng ta rất dễ rơi vào. Chúng ta nhìn thấy có mối liên hệ giữa LAST_NAME và UPPER(LAST_NAME) và mong muốn rằng CSDL cũng ‘nhìn thấy’ điều đó. Trên thực tế, góc nhìn của trình tối ưu hóa có thể hiểu như sau:

```

SELECT first_name, last_name, phone_number
FROM employees
WHERE BLACKBOX(...) = 'WINAND';

```

Hàm UPPER có thể xem chỉ như một hộp đen. Các tham số truyền vào hàm không liên quan vì không có mối liên hệ chung nào giữa các tham số và kết quả.

Tip

Thay thế tên hàm bằng BLACKBOX để hiểu cách nhìn của trình tối ưu hóa.

Ước lượng trong thời gian biên dịch

Trình tối ưu hóa có thể ước lượng trước biểu thức phía bên phải trong “thời gian biên dịch” bởi vì nó có tất cả các tham số đầu vào. Vì thế, kịch bản thực thi của Oracle (phần “Thông tin dự đoán”) chỉ hiển thị viết hoa của cụm từ tìm kiếm. Hành vi này tương đương với 1 trình biên dịch đánh giá các biểu thức hằng số trong thời gian biên dịch.

Để hỗ trợ câu truy vấn như trên, chúng ta cần một chỉ mục bao trùm lên cụm tìm kiếm thực sự. Điều đó có nghĩa là chúng ta không cần chỉ mục trên LAST_NAME mà chỉ cần chỉ mục trên UPPER(LAST_NAME):

```

CREATE INDEX emp_up_name
ON employees (UPPER(last_name));

```

Chỉ mục mà định nghĩa của nó chứa các hàm hoặc biểu thức được gọi là chỉ mục dựa-trên-hàm (function-based index FBI). Thay vì phải sao chép dữ liệu trên cột trực tiếp vào các chỉ mục, một chỉ mục dựa-trên-hàm thực thi hàm và đặt kết quả vào chỉ mục. Kết quả là, các chỉ mục lưu trữ nội dung được viết hoa hoàn toàn.

Cơ sở dữ liệu có thể sử dụng một chỉ mục dựa-trên-hàm nếu biểu thức định nghĩa của chỉ mục xuất hiện trong lệnh SQL, giống như ví dụ trên. Có thể nhận thấy điều này trong kịch bản thực thi:

<i>Id</i>	<i>Operation</i>	<i>Name</i>	<i>Rows</i>	<i>Cost</i>
0 <i>SELECT STATEMENT</i>			100	41
1 <i>TABLE ACCESS BY INDEX ROWID</i>	EMPLOYEES		100	41
*2 <i>INDEX RANGE SCAN</i>	EMP_UP_NAME	40	1	

Predicate Infomation (identified by operation id):

2 - access(UPPER("LAST_NAME")='WINAND')

Nó đơn thuần là việc thực hiện INDEX RANGE SCAN như mô tả trong chương 1. Cơ sở dữ liệu đi theo cây B-tree và đi theo các chuỗi nút lá. Không có bất kỳ toán tử hay từ khóa nào dành riêng cho chỉ mục dựa trên hàm.

Chú ý

Đôi khi các công cụ object-relational mapping ORM sử dụng UPPER hoặc LOWER một cách ngầm định mà nhà phát triển không biết. Ví dụ như Hibernate, sử dụng các LOWER ngầm cho việc tìm kiếm phân biệt chữ hoa-thường.

Kịch bản thực thi vẫn chưa giống như phần trước khi không có UPPER; số lượng hàng ước lượng vẫn còn rất cao. Điều này là đặc biệt kì lạ khi mà trình tối ưu luôn muốn lấy được số lượng hàng nhiều hơn việc quét dải chỉ mục trả về. Làm sao để có thể lấy ra 100 hàng từ bảng khi mà việc quét chỉ mục trước đó chỉ trả về 40 hàng? Câu trả lời là không thể. Các mâu thuẫn về ước tính như trên thường chỉ ra các vấn đề liên quan đến số liệu thống kê. Trong trường hợp cụ thể này là do cơ sở dữ liệu của Oracle không cập nhập các thống kê của bảng khi tạo ra các index mới (xem “Thống kê của Oracle về chỉ mục dựa-trên-hàm” - trang ?).

Sau khi các số liệu thống kê được cập nhập, trình tối ưu sẽ tính toán các ước tính chính xác hơn:

<i>Id</i>	<i>Operation</i>	<i>Name</i>	<i>Rows</i>	<i>Cost</i>
-----------	------------------	-------------	-------------	-------------

0	<i>SELECT STATEMENT</i>		1	3
---	-------------------------	--	---	---

1	<i>TABLE ACCESS BY INDEX ROWID EMPLOYEES</i>		1	3
---	---	--	---	---

*2	<i>INDEX RANGE SCAN</i>	<i>EMP_UP_NAME</i>	1	1
----	-------------------------	--------------------	---	---

Predicate Infomation (identified by operation id):

2 - access(UPPER("LAST_NAME")='WINAND')

Lưu ý

“Thống kê mở rộng – extended statistics” trên các biểu thức và các nhóm cột đã được giới thiệu trong bản phát hành Oracle 11g.

Mặc dù những số liệu thống kê đã cập nhập không cải thiện được hiệu suất trong trường hợp này - dù chỉ mục cũng đã được sử dụng hợp lý - việc kiểm tra những ước tính của trình tối ưu hóa luôn luôn là một ý tưởng tốt. Số lượng các hàng được xử lí cho mỗi hành động (ước lượng các hành động chính) là một giá trị đặc biệt quan trọng mà luôn có mặt trong kế hoạch thực thi của SQL Server và PostgreSQL.

Tip

Phụ lục A, "Các kịch bản thực thi", mô tả các ước tính về số lượng hàng trong các kịch bản thực thi của SQL Server và PostgreSQL.

SQL Server không hỗ trợ các chỉ mục dựa-trên-hàm như đã mô tả nhưng nó cung cấp các cột-tính-toán-trước có thể sử dụng thay thế. Để sử dụng cái này, trước tiên bạn phải thêm một cột-tính-toán-trước vào bảng để có thể được đánh chỉ mục sau đó:

```
ALTER TABLE employees ADD last_name_up AS UPPER(last_name);  
CREATE INDEX emp_up_name ON employees (last_name_up);
```

SQL Server có thể sử dụng các chỉ mục này bất cứ khi nào biểu thức được đánh chỉ mục xuất hiện trong câu truy vấn. Bạn không cần phải viết lại các truy vấn để có thể sử dụng các cột-tính-toán-trước.

Thống kê của Oracle về chỉ mục dựa-trên-hàm

Cơ sở dữ liệu của Oracle duy trì thông tin về số lượng các giá trị khác biệt của cột như một phần của bảng thống kê. Những số liệu đó sẽ được sử dụng nếu như một cột là một phần của nhiều chỉ mục.

Những số liệu thống kê về chỉ mục dựa-trên-hàm (FBI) cũng được giữ ở mức bảng dữ liệu như là các cột ảo. Mặc dù cơ sở dữ liệu của Oracle thu thập các số liệu thống kê về các chỉ mục mới một cách tự động (từ bản phát hành 10g), nó không hề cập nhập các thống kê về bảng dữ liệu. Cũng vì lí do này, các tài liệu của Oracle khuyến nghị cập nhập các thống kê về bảng dữ liệu sau khi tạo chỉ mục dựa-trên-hàm.

Sau khi tạo ra một chỉ mục dựa-trên-hàm, việc thu thập các số liệu thống kê về cả chỉ mục và bảng cơ sở của nó sử dụng DBMS_STATS. Những số lượng thống kê như vậy sẽ giúp cho cơ sở dữ liệu của Oracle quyết định

được chính xác khi nào thì cần sử dụng các chỉ mục. -- Theo hội thảo ngôn ngữ SQL cho cơ sở dữ liệu của Oracle

Khuyến cáo của cá nhân tôi còn xa hơn nữa: sau đó, mỗi khi các chỉ mục thay đổi, hãy cập nhập các số liệu thống kê trong bảng cơ sở và tất cả các chỉ mục của nó. Điều đó là hoàn toàn có thể, tuy nhiên, điều này cũng dẫn đến những hệ quả không mong muốn. Hãy phối hợp các hoạt động này với các quản trị viên cơ sở dữ liệu (DBAs) và sao lưu lại các số liệu thống kê gốc.

14. Các hàm định nghĩa bởi người dùng

Chỉ mục dựa trên hàm là một cách tiếp cận rất chung chung. Bên cạnh các hàm như UPPER, bạn có thể sử dụng các biểu thức chỉ mục như A+B và thậm chí sử dụng các hàm do người dùng tự mình định nghĩa trong việc xác lập chỉ mục.

Có một ngoại lệ quan trọng. Đó là, ví dụ, không thể tham vấn đến thời gian hiện tại trong một định nghĩa chỉ mục, không thể trực tiếp hoặc gián tiếp, như trong ví dụ sau đây:

```
CREATE FUNCTION get_age(date_of_birth DATE)
  RETURN NUMBER
AS
BEGIN
  RETURN
    TRUNC(MONTHS_BETWEEN(SYSDATE, date_of_birth)/12);
END;
/
```

Hàm GET_AGE được sử dụng để lấy thời gian hiện tại (SYSDATE) để tính toán tuổi từ ngày sinh được cung cấp. Bạn có thể sử dụng hàm này trong mọi phần của một câu truy vấn SQL, ví dụ trong mệnh đề select và where:

```
SELECT first_name, last_name, get_age(date_of_birth)
  FROM employees
 WHERE get_age(date_of_birth) = 42;
```

Cây truy vấn liệt kê mọi nhân viên 42 tuổi. Sử dụng một chỉ mục dựa trên hàm là một ý tưởng hiển nhiên để tối ưu truy vấn này, tuy vậy, bạn không thể sử dụng hàm GET_AGE trong một định nghĩa index bởi vì nó không xác định. Có nghĩa là kết quả của lời gọi hàm không hoàn toàn xác định được qua các tham số của nó. Chỉ có các hàm luôn luôn trả lại cùng kết quả cho cùng các tham số - các hàm xác định được - mới có thể được lập chỉ mục.

Lý do đằng sau sự giới hạn này rất đơn giản. Khi chèn thêm một hàng mới, cơ sở dữ liệu gọi đến hàm và lưu trữ kết quả trong chỉ mục và nó sẽ ở yên đó, không thay đổi. Không có tiến trình định kì nào để cập nhật các chỉ mục. Cơ sở dữ liệu chỉ cập nhật

các chỉ mục về tuổi khi ngày sinh bị thay đổi bởi một câu lệnh cập nhập. Sau ngày sinh nhật tiếp theo, tuổi được lưu trữ trong chỉ mục sẽ bị sai.

Bên cạnh tính xác định, cơ sở dữ liệu PostgreSQL và Oracle yêu cầu các hàm phải được khai báo là xác định khi sử dụng bên trong một chỉ mục bằng việc sử dụng từ khóa DETERMINISTIC (Oracle) hoặc IMMUTABLE (PostgreSQL).

Cảnh báo

Cơ sở dữ liệu PostgreSQL và Oracle tin tưởng vào các khai báo DETERMINISTIC hoặc IMMUTABLE - điều đó có nghĩa là họ tin tưởng vào các nhà phát triển.

Bạn có thể khai báo hàm GET_AGE là xác định và sử dụng nó trong một định nghĩa chỉ mục. Tuy nhiên, bất kể những khai báo đó, nó sẽ không hoạt động như dự định bởi vì độ tuổi được lưu trữ trong chỉ mục không tăng theo năm trôi qua; các nhân viên sẽ không già đi- ít nhất là trong chỉ mục.

Một ví dụ khác về các hàm không thể lập chỉ mục đó là bộ sinh số ngẫu nhiên và các hàm phụ thuộc vào các biến môi trường.

Góc suy ngẫm

Làm thế nào bạn vẫn có thể sử dụng chỉ mục để tối ưu hóa truy vấn cho tất cả nhân viên 42 tuổi?

15. Over-indexing

Nếu như khái niệm về chỉ mục dựa trên hàm là mới đối với bạn, bạn có thể bị cảm dỗ lập chỉ mục cho tất cả mọi thứ , tuy nhiên đây thực sự là điều cuối cùng bạn nên làm. Lý do là vì mọi chỉ mục đều cần sự duy trì liên tục. Các chỉ mục dựa trên hàm là đặc biệt phiền phức bởi vì chúng rất dễ dàng để tạo các chỉ mục dư thừa.

Việc tìm kiếm không phân biệt chữ hoa chữ thường ở trên cũng có thể được thực hiện với hàm LOWER như sau:

```
SELECT first_name, last_name, phone_number  
FROM employees  
WHERE LOWER(last_name) = LOWER('winand');
```

Một chỉ mục không thể hỗ trợ cả hai phương thức biến đổi khuôn dạng kí tự. Tất nhiên chúng ta có thể tạo ra một chỉ mục thứ 2 trên LOWER(last_name) cho truy vấn này, tuy nhiên điều này có nghĩa là cơ sở dữ liệu phải duy trì 2 chỉ mục cho mỗi câu lệnh insert, update, và delete (xem thêm trong chương 8, “Sửa đổi dữ liệu”). Để chỉ phải tạo một chỉ mục, bạn nên luôn sử dụng cùng một hàm trong cả ứng dụng của mình.

Tip

Thống nhất về đường dẫn truy cập để một chỉ mục có thể được sử dụng bởi nhiều câu truy vấn.

Tip

Luôn luôn hướng đến việc lập chỉ mục với dữ liệu gốc vì đó thường là thông tin hữu ích nhất mà bạn có thể đưa vào một chỉ mục.

16. Truy vấn có tham số

Phần này giới thiệu về một chủ đề được bỏ qua trong hầu hết các sách giáo khoa SQL: *truy vấn có tham số và tham số trói buộc (bind parameters)*.

Tham số liên kết, còn được gọi là các tham số động, hay biến ràng buộc là một cách khác để truyền dữ liệu tới cơ sở dữ liệu. Thay vì đưa giá trị trực tiếp vào câu lệnh SQL, bạn chỉ cần sử dụng một placeholder như ?, :name hoặc @name và truyền vào các giá trị bằng cách sử dụng một lời gọi API riêng biệt.

Không có vấn đề gì khi viết các giá trị trực tiếp vào các câu truy vấn, tuy nhiên có hai lý do để sử dụng tham số liên kết trong các chương trình:

TÍNH BẢO MẬT

- Tham số liên kết là cách tốt nhất để ngăn chặn SQL injection (*một kỹ thuật cho phép những kẻ tấn công lợi dụng lỗ hổng của việc kiểm tra dữ liệu đầu vào trong các ứng dụng web và các thông báo lỗi của hệ quản trị cơ sở dữ liệu trả về để inject (tiêm vào) và thi hành các câu lệnh SQL bất hợp pháp*).

HIỆU NĂNG

- Cơ sở dữ liệu với bộ nhớ cache lưu kế hoạch thực thi như SQL Server và cơ sở dữ liệu Oracle có thể sử dụng lại kế hoạch thực thi khi thực hiện cùng một câu lệnh nhiều lần. Nó tiết kiệm công sức trong việc xây dựng lại kế hoạch thực thi nhưng chỉ hoạt động được nếu câu lệnh SQL là giống hệt nhau. Nếu bạn đưa các giá trị khác vào câu lệnh SQL, cơ sở dữ liệu sẽ xử lý nó như một câu lệnh khác và sẽ tạo lại kế hoạch thực thi.
- Khi sử dụng các tham số ràng buộc, bạn không ghi các giá trị thực tế nhưng thay vào đó chèn các placeholder vào câu lệnh SQL. Bằng cách đó, các câu lệnh không thay đổi khi thực hiện chúng với các giá trị khác nhau.

Đương nhiên sẽ có những ngoại lệ, ví dụ như nếu dữ liệu bị ảnh hưởng phụ thuộc vào giá trị thực tế:

```
99 rows selected.
```

```
SELECT first_name, last_name  
  FROM employees  
 WHERE subsidiary_id = 20;
```

Id Operation	Name	Rows	Cost
0 SELECT STATEMENT		99	70
1 TABLE ACCESS BY INDEX ROWID	EMPLOYEES	99	70
*2 INDEX RANGE SCAN	EMPLOYEE_PK	99	2

```
Predicate Information (identified by operation id):
```

```
2 - access("SUBSIDIARY_ID"=20)
```

Hình 2: Truy vấn trên nhánh nhỏ

Một index mang lại hiệu suất tốt nhất cho chi nhánh nhỏ, có số nhân viên ít, nhưng một TABLE ACCESS FULL có thể tốt hơn index cho chi nhánh lớn:

```
1000 rows selected.
```

```
SELECT first_name, last_name  
  FROM employees  
 WHERE subsidiary_id = 30;
```

Id Operation	Name	Rows	Cost
0 SELECT STATEMENT		1000	478
* 1 TABLE ACCESS FULL	EMPLOYEES	1000	478

```
Predicate Information (identified by operation id):
```

```
1 - filter("SUBSIDIARY_ID"=30)
```

Hình 3: Truy vấn trên nhánh lớn

Trong trường hợp này, thông tin histogram trên SUBSIDIARY_ID là rất hữu dụng. Bộ tối ưu hóa sử dụng nó để xác định tần số của ID chi nhánh được đề cập trong truy vấn SQL. Do đó nó có 2 cách tính số lần truy cập khác nhau cho cả hai truy vấn.

Việc tính chi phí sau đó sẽ dẫn đến kết quả hai giá trị chi phí khác nhau. Khi bộ tối ưu hóa chọn một kế hoạch thực thi, nó sẽ chọn kế hoạch với giá trị chi phí thấp nhất. Đối với chi nhánh nhỏ hơn, thì đó là sử dụng index.

Chi phí của sử dụng TABLE ACCESS BY INDEX ROWID là rất nhạy cảm với ước tính số lượng hàng. Chọn mười lần nhiều hơn số các hàng sẽ làm tăng chi phí bởi yếu tố đó. Chi phí tổng thể bằng cách sử dụng index thậm chí còn cao hơn so với quét toàn bộ bảng. Bộ tối ưu hóa do đó sẽ chọn kế hoạch thực thi khác cho chi nhánh lớn hơn.

Khi sử dụng các tham số liên kết, bộ tối ưu hóa không có giá trị cụ thể có sẵn để xác định tần số của chúng. Nó về sau sẽ giả định một sự phân bố ngang bằng và luôn nhận được các giá trị ước tính và giá trị chi phí như nhau. Cuối cùng, nó sẽ luôn luôn chọn cùng một kế hoạch thực thi.

Chú ý:

Histogram của các giá trị của cột rất hữu ích nếu các giá trị không được phân bổ đều. Đối với các cột có sự phân bố đều thì thường phân chia số lượng các giá trị riêng biệt bằng số hàng trong bảng. Phương pháp này cũng hoạt động khi sử dụng các tham số ràng buộc.

Nếu chúng ta so sánh bộ tối ưu hóa với trình biên dịch, các biến trói buộc giống như các biến chương trình, nhưng nếu bạn viết các giá trị trực tiếp vào câu lệnh, chúng giống như các hằng số. Cơ sở dữ liệu có thể sử dụng các giá trị từ câu lệnh SQL trong quá trình tối ưu giống như trình biên dịch có thể đánh giá các biểu thức hằng trong quá trình biên dịch. Các tham số liên kết không hiển thị với bộ tối ưu hóa cũng giống như các giá trị thời gian chạy của biến không được biết đến trình biên dịch.

Từ quan điểm này, có một chút nghịch lý rằng các tham số trói buộc có thể cải thiện hiệu suất hay không nếu việc sử dụng tham số trói buộc không cho phép bộ tối ưu hóa lựa chọn kế hoạch thực hiện tốt nhất. Tuy nhiên, câu hỏi đặt ra là mức giá nào? Việc tạo ra và đánh giá tất cả các biến thể của kế hoạch thực hiện là một nỗ lực rất lớn không tương xứng nếu cuối cùng bạn nhận được cùng một kết quả.

Chú ý:

Không sử dụng các tham số trói buộc giống như luôn luôn biến dịch lại một chương trình bất cứ khi nào có truy vấn.

Việc quyết định xây dựng một kế hoạch thực thi riêng biệt hoặc tổng quát là một vấn đề khó quyết cho bộ tối ưu. Hoặc là bộ tối ưu đánh giá tất cả các biến thể của kế hoạch thực thi để luôn có kế hoạch thực thi tốt nhất hoặc sử dụng một kế hoạch thực thi đã lưu trữ trong bộ nhớ cache được sử dụng bất cứ khi nào có thể, chấp nhận nguy cơ sử dụng một kế hoạch thực thi chưa tối ưu nhất. Vấn đề mấu chốt là bộ tối ưu không thể biết là liệu thực thi lại quá trình đánh giá các phương án thực thi có đưa ra một kế hoạch thực thi khác thực sự thực hiện tối ưu hóa đầy đủ hay không. Các nhà phát triển hệ quản trị cơ sở dữ liệu cố gắng giải quyết vấn đề tiến thoái lưỡng nan này với các phương pháp dựa theo kinh nghiệm (heuristics) - nhưng thành công vẫn rất hạn chế.

Là nhà phát triển phần mềm, bạn có thể cân nhắc sử dụng các tham số trói buộc để giúp giải quyết tình trạng tiến thoái lưỡng nam này. Đó là, bạn nên luôn luôn sử dụng các tham số trói buộc, ngoại trừ trường hợp các giá trị đầu vào đưa vào tham số trói buộc có khả năng cao đưa đến kế hoạch thực thi tồi.

Xét ví dụ cột trong bảng dữ liệu lưu các mã trạng thái phân phôi không đồng đều như "todo" và "done" là một ví dụ điển hình. Số mục "done" thường vượt quá các bản ghi "todo" theo thứ tự độ lớn. Sử dụng một chỉ mục chỉ có ý nghĩa khi tìm kiếm các giá trị "todo" trong trường hợp đó. Phân vùng (partitionning) trong cơ sở dữ liệu là một ví dụ khác-nghĩa là, nếu bảng dữ liệu được phân mảnh và đánh chỉ mục trên nhiều thiết bị lưu trữ độc lập. Các giá trị thực tế trong bảng dữ liệu sẽ ảnh hưởng đến những phân vùng nào cần được quét khi thực thi truy vấn SELECT. Hiệu năng, tốc độ thực hiện các truy vấn LIKE sẽ biến thiên rất lớn khi sử dụng với các tham số trói buộc, chúng ta sẽ thấy trong phần tiếp theo.

Chú ý:

Trong thực tế, chỉ có một vài trường hợp, trong đó các giá trị thực tế ảnh hưởng đến kế hoạch thực thi. Do đó, bạn nên sử dụng các tham số trói buộc – ít nhất để ngăn chặn SQL injection.

Các đoạn mã sau đây hướng dẫn sử dụng các tham số trói buộc trong các ngôn ngữ lập trình.

C#

Không có tham số ràng buộc:

```
int subsidiary_id;
SqlCommand cmd = new SqlCommand(
    "select first_name, last_name"
    + " from employees"
    + " where subsidiary_id = " + subsidiary_id
    , connection);
```

Hình 4 : C# không sử dụng tham số ràng buộc

Có tham số ràng buộc:

```
int subsidiary_id;
SqlCommand cmd =
    new SqlCommand(
        "select first_name, last_name"
        + " from employees"
        + " where subsidiary_id = @subsidiary_id"
        , connection);
cmd.Parameters.AddWithValue("@subsidiary_id", subsidiary_id);
```

Hình 5: C# sử dụng tham số ràng buộc

Xem thêm: *SqlParameterCollection class documentation*.

Java:

Không có tham số ràng buộc:

```
int subsidiary_id;
Statement command = connection.createStatement(
    "select first_name, last_name"
    + " from employees"
    + " where subsidiary_id = " + subsidiary_id
);
```

Hình 6: Java không sử dụng tham số ràng buộc

Có tham số ràng buộc:

```
int subsidiary_id;
PreparedStatement command = connection.prepareStatement(
        "select first_name, last_name"
        + " from employees"
        + " where subsidiary_id = ?"
        );
command.setInt(1, subsidiary_id);
```

Hình 7: Java có sử dụng tham số ràng buộc

Xem thêm: *PreparedStatement class documentation*.

Perl:

Không có tham số ràng buộc:

```
my $subsidiary_id;
my $sth = $dbh->prepare(
        "select first_name, last_name"
        . " from employees"
        . " where subsidiary_id = $subsidiary_id"
        );
$sth->execute();
```

Hình 8: Perl không sử dụng tham số ràng buộc

Có tham số ràng buộc:

```
my $subsidiary_id;
my $sth = $dbh->prepare(
        "select first_name, last_name"
        . " from employees"
        . " where subsidiary_id = ?"
        );
$sth->execute($subsidiary_id);
```

Hình 9: Perl có sử dụng tham số ràng buộc

Xem thêm: *Programming the Perl DBI*.

PHP:

Không có tham số ràng buộc:

```
$mysqli->query("select first_name, last_name"
    . " from employees"
    . " where subsidiary_id = " . $subsidiary_id);
```

Hình 10: PHP không sử dụng tham số ràng buộc

Có tham số ràng buộc:

```
if ($stmt = $mysqli->prepare("select first_name, last_name"
    . " from employees"
    . " where subsidiary_id = ?"))
{
    $stmt->bind_param("i", $subsidiary_id);
    $stmt->execute();
} else {
    /* handle SQL error */
}
```

Hình 11: PHP sử dụng tham số ràng buộc

Xem thêm: *mysqli_stmt::bind_param class documentation and “Prepared statements and stored procedures” in the PDO documentation.*

Ruby:

Không có tham số ràng buộc:

```
dbh.execute("select first_name, last_name"
    + " from employees"
    + " where subsidiary_id = #{subsidiary_id}");
```

Hình 12: Ruby không sử dụng tham số ràng buộc

Có tham số ràng buộc:

```
dbh.prepare("select first_name, last_name"
    + " from employees"
    + " where subsidiary_id = ?");
dbh.execute(subsidiary_id);
```

Hình 13: Ruby sử dụng tham số ràng buộc

Xem thêm: “*Quoting, Placeholders, and Parameter Binding*” in the Ruby DBI Tutorial.

Dấu chấm hỏi (?) là ký tự giữ chỗ (placeholder) duy nhất mà tiêu chuẩn SQL định nghĩa. Dấu chấm hỏi là các tham số có sắp thứ tự. Các dấu chấm hỏi được đánh số từ trái sang phải. Để trói buộc một giá trị cho một dấu chấm hỏi cụ thể, bạn phải chỉ định số thứ tự của nó. Tuy nhiên, điều đó có thể rất không thiết thực vì vị trí tham số thay đổi khi thêm hoặc xoá các placeholder. Nhiều cơ sở dữ liệu cung cấp tính năng mở rộng cho phép sử dụng các tham số được đặt tên để giải quyết vấn đề này-ví dụ: sử dụng ký hiệu "at" (@name) hoặc dấu hai chấm (:name).

Chú ý:

Tham số trói buộc không thể thay đổi cấu trúc của câu lệnh SQL.

Điều đó có nghĩa là bạn không thể sử dụng tham số trói buộc cho các tên bảng hoặc cột. Các tham số ràng buộc sau đây không hoạt động:

```
String sql = prepare("SELECT * FROM ? WHERE ?");  
sql.execute('employees', 'employee_id = 1');
```

Hình 13: Ví dụ tham số ràng buộc không hoạt động

Nếu bạn cần thay đổi cấu trúc câu lệnh SQL trong thời gian chạy, hãy sử dụng SQL động.

Con trỏ chia sẻ và tự động hóa tham số:

Bộ tối ưu hóa và truy vấn SQL càng phức tạp, kỹ thuật cache kế hoạch thực thi ở bộ nhớ đệm càng quan trọng hơn. Cơ sở dữ liệu SQL Server và Oracle có các tính năng tự động thay thế các giá trị trong một chuỗi SQL thành các tham số trói buộc. Các tính năng này được gọi là CURSOR_SHARING (Oracle) hoặc Forced parameterization (SQL Server).

Cả hai tính năng là cách giải quyết cho các ứng dụng mà không sử dụng các tham số trói buộc. Bật các tính năng này ngăn các nhà phát triển cố ý sử dụng các giá trị cụ thể đưa vào tham số trong mệnh đề WHERE.

17. Tìm kiếm theo khoảng

Các toán tử bắt đầu như <, > và **between** có thể sử dụng các index giống như toán tử bằng, như đã giải thích ở trên. Trong những trường hợp nhất định, với lệnh lọc “LIKE” cũng có thể sử dụng index giống như với điều kiện phạm vi (truy vấn theo khoảng).

Sử dụng các phép toán này giới hạn sự lựa chọn thứ tự cột trong các index nhiều cột. Sự hạn chế đó thậm chí có thể loại trừ tất cả các tùy chọn chỉ mục tối ưu - Có những câu truy vấn mà bạn đơn giản không xác định được “chính xác” thứ tự các cột.

18. Lớn hơn, nhỏ hơn và trong khoảng (between)

Rủi ro hiệu năng lớn nhất của INDEX RANGE SCAN đó là việc di chuyển qua nút lá (trong phần B-Tree Traversal - Chương 1). Vì vậy, quy tắc vàng của việc tạo index là khiến cho việc quét index có phạm vi nhỏ nhất có thể. Bạn có thể kiểm tra bằng cách tự xem xét nơi quét index bắt đầu và nơi nó kết thúc.

Câu hỏi thật dễ dàng để trả lời nếu câu lệnh SQL đề cập đến điều kiện bắt đầu và điều kiện dừng rõ ràng:

```
SELECT first_name, last_name, date_of_birth  
      FROM employees  
 WHERE date_of_birth >= TO_DATE(?:, 'YYYY-MM-DD')  
   AND date_of_birth <= TO_DATE(?:, 'YYYY-MM-DD')
```

Hình 15: Truy vấn date_of_birth có chẵn đầu trên và đầu dưới

Chỉ mục trên DATE_OF_BIRTH chỉ được quét trong phạm vi được chỉ định. Việc quét bắt đầu từ ngày đầu tiên và kết thúc ở lần thứ hai. Chúng ta không thể thu hẹp phạm vi quét index được nữa.

Các điều kiện bắt đầu và điều kiện dừng sẽ kém rõ ràng nhất nếu có cột thứ hai trở nên được đưa vào:

```
SELECT first_name, last_name, date_of_birth  
      FROM employees  
 WHERE date_of_birth >= TO_DATE(?:, 'YYYY-MM-DD')  
   AND date_of_birth <= TO_DATE(?:, 'YYYY-MM-DD')  
   AND subsidiary_id = ?
```

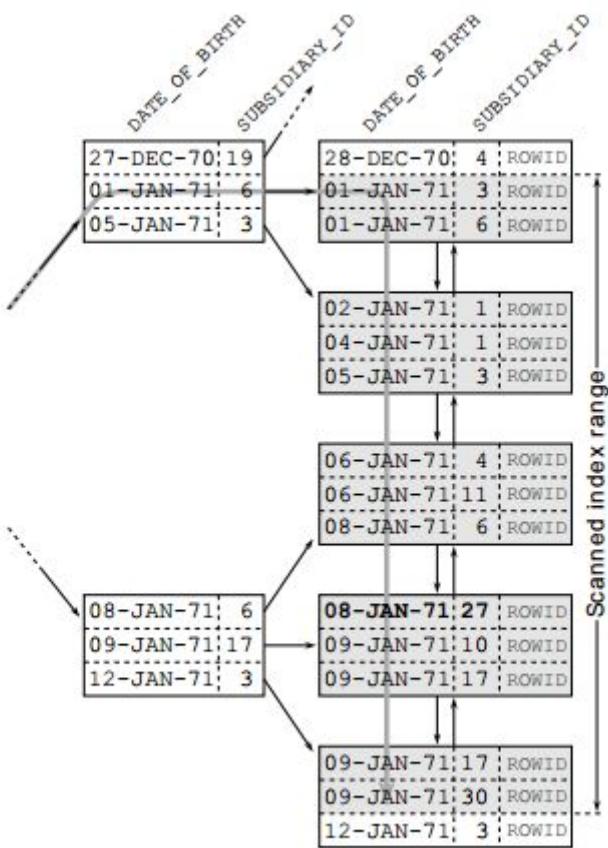
Hình 16: Truy vấn khoảng date_of_birth có thêm điều kiện ngoài

Tất nhiên một index lý tưởng đã bao gồm cả hai cột, nhưng câu hỏi là trong thứ tự nào?

Các con số sau đây cho thấy hiệu quả của thứ tự cột khi duyệt index theo khoảng. Để minh họa, chúng ta tìm kiếm tất cả nhân viên của chi nhánh số 27 có ngày sinh từ ngày 1 tháng 1 đến ngày 9 tháng 1 năm 1971.

Hình ? hiển thị một thẻ hiện chi tiết của index trên DATE_OF_BIRTH và SUBSIDIARY_ID, theo thứ tự đó. CSDL sẽ bắt đầu duyệt từ nút là nào, nói cách khác: việc duyệt cây kết thúc ở đâu?

Chúng ta thấy index này được sắp xếp theo ngày sinh trước tiên. Chỉ khi hai nhân viên ra đời trong cùng một ngày thì SUBSIDIARY_ID được sử dụng để sắp xếp các bản ghi này. Tuy nhiên, câu truy vấn bao gồm một phạm vi ngày. Thứ tự của SUBSIDIARY_ID vì thế không có ích trong suốt quá trình di chuyển cây. Điều đó trở nên rõ ràng nếu bạn nhận ra rằng không có nút chỉ mục cho chi nhánh có số 27 trong các nút nhánh - mặc dù có một trong các nút lá. Bộ lọc vào DATE_OF_BIRTH do đó là điều kiện duy nhất giới hạn phạm vi index được quét. Nó bắt đầu từ nút lá đầu tiên phù hợp với phạm vi ngày và kết thúc ở nút lá cuối cùng - tất cả năm nút lá thể hiện trong hình ?.

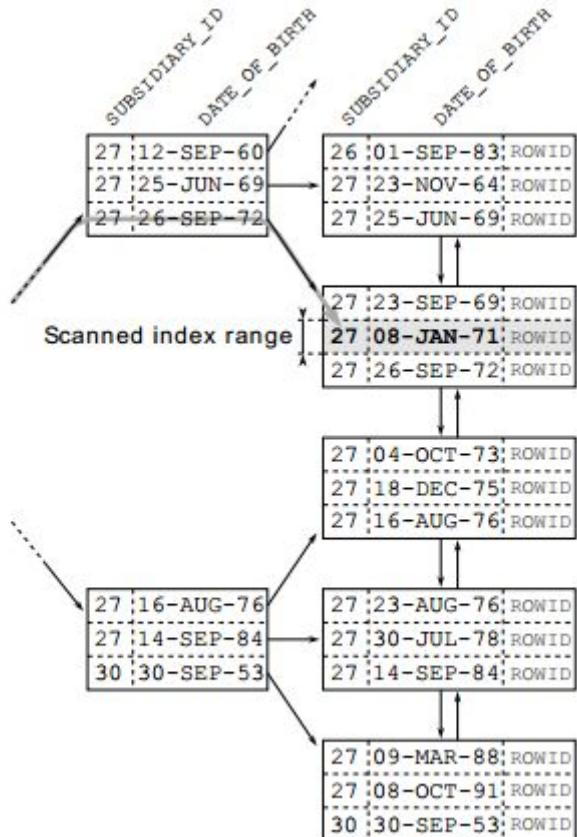


Hình 17: Phạm vi quét index với `date_of_birth` đứng trước

Trong trường hợp đảo ngược thứ tự các cột trong index, hình ảnh có vẻ hoàn toàn khác. Hình ? minh họa việc quét nếu index bắt đầu bằng cột `SUBSIDIARY_ID`.

Sự khác biệt là toán tử bằng giới hạn cột chỉ mục đầu tiên với một giá trị. Trong phạm vi cho giá trị này (`SUBSIDIARY_ID 27`) index được sắp xếp theo cột thứ hai - `DATE_OF_BIRTH` - do đó không cần phải ghé thăm nút lá đầu tiên vì nút nhánh đã chỉ ra rằng không có nhân viên nào ở chi nhánh 27 sinh sau ngày 25 tháng 6 năm 1969 ở nút lá đầu tiên.

Figure 2.3. Range Scan in `SUBSIDIARY_ID, DATE_OF_BIRTH Index`



Hình 18: Phạm vi quét của index với `subsidiary_id` đúng trước

Việc đi qua cây trực tiếp dẫn đến nút lá thứ hai. Trong trường hợp này, tất cả các điều kiện mệnh đề `where` hạn chế phạm vi index để quét chấm dứt tại nút cùng một lá.

Chú ý:

*Nguyên tắc chung: **index for equality first—then for ranges** (index cho điều kiện bằng trước, cho điều kiện khoảng sau).*

Sự khác biệt về hiệu suất thực tế phụ thuộc vào dữ liệu và tiêu chí tìm kiếm. Sự khác biệt có thể không đáng kể nếu bộ lọc trên `DATE_OF_BIRTH` có tính chọn lọc cao. Phạm vi ngày càng lớn thì sự khác biệt về hiệu suất sẽ lớn hơn.

Với ví dụ này, chúng ta cũng có thể thấy sai lầm rất phổ biến là luôn chọn cột có tính chọn lọc nhất nằm ở vị trí bên trái nhất có thể của index kết hợp. Nếu chúng ta nhìn vào các con số và chỉ xem xét tính chọn lọc của cột đầu tiên, chúng ta thấy rằng cả hai phương án đều có 13 bản ghi thỏa mãn. Bất kể trường hợp nào khi chúng ta chỉ lọc theo `DATE_OF_BIRTH` hoặc chỉ theo `SUBSIDIARY_ID`. Sự chọn lọc không sử dụng ở đây, nhưng thứ tự của cột dẫn tới một phương án index tốt hơn phương án còn lại.

Để tối ưu hóa hiệu suất, điều quan trọng là phải biết phạm vi index được quét. Với hầu hết các cơ sở dữ liệu bạn thậm chí có thể thấy điều này trong kế hoạch thực thi - bạn chỉ cần biết mình cần tìm cái gì. Kế hoạch thực thi sau từ cơ sở dữ liệu Oracle chỉ rõ ràng rằng index EMP_TEST bắt đầu với cột DATE_OF_BIRTH.

Id Operation	Name	Rows	Cost
0 SELECT STATEMENT			
*1 FILTER			
2 TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	4
*3 INDEX RANGE SCAN	EMP_TEST	2	2

Predicate Information (identified by operation id):

```

1 - filter(:END_DT >= :START_DT)
3 - access(DATE_OF_BIRTH >= :START_DT
           AND DATE_OF_BIRTH <= :END_DT)
      filter(SUBSIDIARY_ID = :SUBS_ID)

```

Hình 19: Kế hoạch thực hiện rõ ràng index

Phản thông tin dự báo (predicate information) cho INDEX RANGE SCAN đưa ra những gợi ý quan trọng. Nó chỉ ra các điều kiện của lệnh **where** hoặc là **access** hoặc **filter**. Đây là cách mà cơ sở dữ liệu cho chúng ta các điều kiện được thực thi theo cách nào.

Chú ý:

Kế hoạch thực thi trên đã được đơn giản hóa cho rõ ràng. Phụ lục ở trang ? giải thích các chi tiết của phần "Predicate infomation" trong một kế hoạch thực thi của Oracle.

Các điều kiện trên cột DATE_OF_BIRTH là điều kiện của vị từ truy cập, chúng giới hạn phạm vi index sẽ quét. DATE_OF_BIRTH là cột đầu tiên trong index EMP_TEST. Cột SUBSIDIARY_ID chỉ được sử dụng làm bộ lọc.

Điểm quan trọng:

Các vị từ truy cập (access predicates) là các điều kiện bắt đầu và dùng cho một truy cập trên index. Chúng xác định phạm vi index được quét.

Các vị từ lọc (filter predicates) index được áp dụng trong suốt quá trình di chuyển nút lá. Chúng không làm giảm phạm vi index được quét.

Phụ lục A giải thích làm thế nào để nhận ra các vị trí truy cập trong các cơ sở dữ liệu khác.

Cơ sở dữ liệu có thể sử dụng tất cả các điều kiện như vị trí truy cập nếu chúng ta đảo thứ tự cột trong index kết hợp:

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	3
* 1	FILTER			
2	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	3
* 3	INDEX RANGE SCAN	EMP_TEST2	1	2

Predicate Information (identified by operation id):

```
1 - filter(:END_DT >= :START_DT)
3 - access(SUBSIDIARY_ID = :SUBS_ID
           AND DATE_OF_BIRTH >= :START_DT
           AND DATE_OF_BIRTH <= :END_T)
```

Hình 20: Sử dụng toàn bộ vị trí truy cập

Dưới đây chúng ta xem xét toán tử **between**. Nó cho phép bạn chỉ định giới hạn trên và dưới trong một điều kiện duy nhất:

```
DATE_OF_BIRTH BETWEEN '01-JAN-71'
                  AND '10-JAN-71'
```

Hình 21: Truy vấn sử dụng between

Lưu ý rằng toán tử **between** luôn bao gồm các giá trị được chỉ định, giống như sử dụng nhỏ hơn hoặc bằng (\leq) và lớn hơn hoặc bằng các toán tử (\geq):

```
DATE_OF_BIRTH >= '01-JAN-71'
AND DATE_OF_BIRTH <= '10-JAN-71'
```

Hình 22: Truy vấn between sử dụng giống truy vấn sử dụng (\leq) và (\geq)

19. Indexing LIKE Filters

Toán tử SQL LIKE thường xuyên kéo theo tác động không lường trước được tới hiệu suất truy vấn bởi vì có thể tồn tại từ khóa tìm kiếm mà hạn chế việc sử dụng index

hiệu quả. Điều này có nghĩa là nhiều từ khóa tìm kiếm có thể sử dụng index rất hiệu quả nhưng một số từ khóa khác thì không. Vị trí của kí tự đại diện sẽ tạo ra sự khác biệt trong truy vấn.

Ví dụ dưới đây sử dụng kí tự % đặt ở giữa từ khóa tìm kiếm:

```
SELECT first_name, last_name, date_of_birth  
  FROM employees  
 WHERE UPPER(last_name) LIKE 'WIN%D'
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	4
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	4
*2	INDEX RANGE SCAN	EMP_UP_NAME	1	2

Hình 23: Truy vấn LIKE khi đặt kí tự đại diện ở giữa

Bộ lọc LIKE chỉ có thể sử dụng các kí tự nằm trước kí tự đại diện đầu tiên (kí tự % trong ví dụ trên) trong suốt quá trình duyệt cây. Các kí tự còn lại chỉ là các vị từ lọc mà không giúp thu nhỏ phạm vi quét index. Một biểu thức LIKE có thể chứa hai loại vị từ sau: (1) phần trước kí tự đại diện có tư cách làm vị từ truy cập; (2) Phần kí tự còn lại có tư cách làm vị từ lọc.

Lưu ý :

Đối với cơ sở dữ liệu PostgreSQL, bạn có thể phải chỉ định rõ ràng một lớp toán tử (Ví dụ : varchar_pattern_ops) để sử dụng được các biểu thức LIKE làm các vị từ truy cập. Tham khảo “Operator Classes and Operator Families ” trong tài liệu PostgreSQL để biết thêm chi tiết.

Tiền tố có nhiều tính chọn lọc nằm trước kí tự đại diện đầu tiên làm cho phạm vi được quét bởi index nhỏ hơn. Điều này làm cho việc tìm kiếm của index nhanh hơn. Hình ? minh họa cho mối quan hệ này bằng cách sử dụng ba biểu thức LIKE khác nhau. Cả ba lựa chọn cùng một hàng, nhưng phạm vi được index quét - và do đó hiệu suất là rất khác nhau.

LIKE 'WI%ND'	LIKE 'WIN%D'	LIKE 'WINA%'
WIAW	WIAW	WIAW
WIBLQQNPUA	WIBLQQNPUA	WIBLQQNPUA
WIBYHSNZ	WIBYHSNZ	WIBYHSNZ
WIFMDWUQMB	WIFMDWUQMB	WIFMDWUQMB
WIGLZX	WIGLZX	WIGLZX
WIH	WIH	WIH
WIHTFVZNLC	WIHTFVZNLC	WIHTFVZNLC
WIJYAXPP	WIJYAXPP	WIJYAXPP
WINAND	WINAND	WINAND
WINBKYDSKW	WINBKYDSKW	WINBKYDSKW
WIPOJ	WIPOJ	WIPOJ
WISRGPK	WISRGPK	WISRGPK
WITJIVQJ	WITJIVQJ	WITJIVQJ
WIW	WIW	WIW
WIWGPJMQGG	WIWGPJMQGG	WIWGPJMQGG
WIWKHLBJ	WIWKHLBJ	WIWKHLBJ
WIYETHN	WIYETHN	WIYETHN
WIYJ	WIYJ	WIYJ

Hình 24: Các phương pháp tìm kiếm LIKE khác nhau

Biểu thức đầu tiên có hai ký tự trước ký tự đại diện. Nó giới hạn phạm vi index quét đến 18 hàng. Chỉ có 1 dòng trong đó khớp với toàn bộ cụm từ được tìm kiếm bởi toán tử LIKE, vì vậy 17 dòng được nạp vào nhưng vẫn bị loại. Biểu thức thứ hai có tiền tố dài hơn thu hẹp phạm vi index được quét xuống hai hàng. Với biểu thức này, cơ sở dữ liệu chỉ phải đọc một dòng phụ không liên quan đến kết quả. Biểu thức cuối cùng không có vị từ lọc: cơ sở dữ liệu chỉ đọc thực thể phù hợp với toàn bộ cụm từ được tìm kiếm trong toán tử LIKE.

Quan trọng:

Chỉ phần nằm trước ký tự đại diện đầu tiên đóng vai trò một vị từ truy cập. Các ký tự còn lại không thu hẹp phạm vi được quét bởi index – các dòng, bản ghi không phù hợp được loại bỏ khỏi kết quả tìm kiếm.

Trường hợp ngược lại cũng có thể xảy ra: biểu thức LIKE bắt đầu bằng ký tự đại diện. Biểu thức LIKE như vậy không thể phục vụ như một vị từ truy cập. Cơ sở dữ liệu phải quét toàn bộ bảng nếu không có các điều kiện khác cung cấp các vị từ truy cập.

Chú ý:

Tránh biểu thức LIKE với ký tự đầu tiên là ký tự đại diện (ví dụ: "% TERM").

Vị trí của các ký tự đại diện sẽ ảnh hưởng đến việc sử dụng index - ít nhất theo lý thuyết. Trong thực tế, bộ tối ưu hóa tạo ra một kế hoạch thực thi chung khi cụm từ tìm kiếm được cung cấp thông qua các tham số trói buộc (bind parameters). Trong trường hợp đó bộ tối ưu hóa phải dự đoán xem liệu phần lớn các cách thực thi có một ký tự đại diện % ở đầu hay không.

Hầu hết các cơ sở dữ liệu giả định rằng không có ký tự đại diện ở đầu khi tối ưu hóa điều kiện LIKE với tham số trói buộc, nhưng giả định này là sai nếu biểu thức LIKE được sử dụng cho full-text search. Không may là các lập trình viên hay sử dụng điều kiện LIKE cho full-text search. Lưu ý “Labeling Full-Text LIKE Expressions” dưới đây đưa ra một ví dụ cách dùng mà không hiệu quả. Chỉ định cụm từ tìm kiếm mà không sử dụng tham số trói buộc là phương pháp rõ ràng nhất, nhưng nó làm tăng chi phí tối ưu hóa và mở ra một số lỗi SQL injection. Một giải pháp hiệu quả nhưng vẫn an toàn và tiện là có ý làm mờ điều kiện LIKE. “Combining Columns” ở trang ? giải thích chi tiết.

Labeling Full-Text LIKE Expressions

Khi sử dụng toán tử LIKE cho full-text search, chúng ta có thể tách biệt các ký tự đại diện khỏi thuật ngữ tìm kiếm:

WHERE text_column LIKE '%' || ? || '%'

Các ký tự đại diện được viết tiếp vào câu lệnh SQL, nhưng chúng ta sử dụng tham số trói buộc cho cụm từ tìm kiếm. Biểu thức LIKE cuối cùng được xây dựng bởi chính cơ sở dữ liệu bằng cách sử dụng toán tử kết nối chuỗi || (Oracle, PostgreSQL). Mặc dù sử dụng một tham số trói buộc, biểu thức cuối cùng LIKE sẽ luôn luôn bắt đầu với một ký tự đại diện. Thật không may cơ sở dữ liệu không nhận ra điều đó.

Đối với cơ sở dữ liệu PostgreSQL, vấn đề có sự khác biệt vì PostgreSQL giả định có một ký tự đại diện ở đầu khi sử dụng tham số trói buộc cho một biểu thức LIKE. Trong trường hợp này, PostgreSQL quyết định không sử dụng. Cách duy nhất để sử dụng index cho một biểu thức LIKE là làm bộ tối ưu hóa nhìn thấy được cụm từ tìm kiếm thực tế. Nếu bạn không sử dụng tham số liên kết nhưng đặt cụm từ tìm kiếm trực tiếp vào câu lệnh SQL, bạn phải thực hiện các biện pháp phòng ngừa khác đối với các cuộc tấn công SQL injection!

Ngay cả khi cơ sở dữ liệu tối ưu hóa kế hoạch thực thi truy vấn có điều kiện LIKE với một ký tự đại diện % ở đầu, nó vẫn không đạt được hiệu suất tối ưu. Bạn có thể sử dụng một điều kiện khác trong mệnh đề **where** để truy cập dữ liệu hiệu quả trong trường hợp đó - xem thêm "Index Filter Predicates Used Intentionally" ở trang ?.

Nếu không có đường dẫn truy cập khác, bạn có thể sử dụng một trong các giải pháp full-text index thừa riêng sau đây.

MySQL

MySQL cung cấp từ khóa match và against cho tìm kiếm toàn văn (full-text search). Bắt đầu với MySQL 5.6, bạn có thể tạo những index full-text cho các bảng InnoDB - trước đây, điều này chỉ có thể với các bảng MyISAM. Tham khảo “Full-Text Search Functions” trong tài liệu MySQL.

Oracle Database

Cơ sở dữ liệu Oracle cung cấp từ khóa **contains**. Xem “Oracle Text Application Developer’s Guide.”

PostgreSQL

PostgreSQL cung cấp toán tử **@@** để thực hiện full-text searches. Xem “Full-Text Search” trong tài liệu PostgreSQL.

Một tùy chọn khác là sử dụng phần mở rộng WildSpeed để tối ưu hóa trực tiếp biểu thức LIKE. Phần mở rộng lưu văn bản ở tất cả các phép quay có thể để mỗi ký tự đều có 1 lần đứng đầu. Điều đó có nghĩa là nội dung văn bản được đánh chỉ mục không chỉ được lưu trữ một lần mà nhiều lần bằng số các ký tự trong chuỗi - do đó nó cần rất nhiều không gian lưu trữ.

SQL Server

SQL Server cung cấp từ khóa **contains**. Xem “Full-Text Search” trong tài liệu SQL Server.

Hãy suy nghĩ về điều này:

Làm thế nào bạn có thể lập index cho phép tìm kiếm LIKE mà chỉ có một ký tự đại diện ở đầu của cụm từ tìm kiếm ('% TERM')?

20. Chỉ mục gộp (index merge)

Đây là một trong những câu hỏi phổ biến nhất về lập index: tốt hơn là nên tạo ra một index cho mỗi cột hay một index duy nhất cho tất cả các cột của mệnh đề **where**? Câu trả lời rất đơn giản trong hầu hết các trường hợp: một index với nhiều cột là tốt hơn.

Thực tế, có những truy vấn mà một index đơn không thể đem lại hiệu năng hoàn hảo, dù bạn có lựa chọn index như thế nào chăng nữa; ví dụ: các truy vấn có hai hoặc nhiều điều kiện phạm vi độc lập như trong ví dụ sau:

```
SELECT first_name, last_name, date_of_birth  
  FROM employees  
 WHERE UPPER(last_name) < ?  
   AND date_of_birth < ?
```

Hình 25: Truy vấn sử dụng hai điều kiện độc lập

Không thể định nghĩa một B-tree index có thể hỗ trợ truy vấn này mà không sử dụng bộ lọc vị từ. Để giải thích, bạn chỉ cần nhớ rằng một index, thông thường chuỗi nút lá là một danh sách liên kết.

Nếu bạn xác định index kết hợp (index gộp) là UPPER (LAST_NAME), DATE_OF_BIRTH (theo thứ tự đó), danh sách bắt đầu bằng chữ A và kết thúc bằng chữ Z. Ngày sinh chỉ được xem là khi có hai nhân viên có cùng tên. Nếu bạn xác định index theo cách khác, nó sẽ bắt đầu với các nhân viên lớn tuổi và kết thúc bằng người trẻ nhất. Trong trường hợp đó, tên chỉ có tác động nhỏ đến thứ tự sắp xếp.

Cho dù bạn có làm mọi cách để định nghĩa một index, các bản ghi luôn luôn được sắp xếp theo một thứ tự logic. Ở một đầu, bạn có các mục nhỏ và ở đầu kia là những cái lớn. Một index có thể chỉ hỗ trợ một phạm vi điều kiện như một vị từ truy cập. Hỗ trợ hai dải độc lập điều kiện đòi hỏi một trực thứ hai, ví dụ như một bàn cờ vua. Truy vấn ở trên sau đó sẽ khớp với tất cả các mục từ một góc của bàn cờ vua, nhưng một index không giống như một chiếc bàn cờ-nó giống như một chuỗi.

Tất nhiên bạn có thể chấp nhận sử dụng các vị từ lọc và sử dụng một index nhiều cột. Đó là giải pháp tốt nhất trong nhiều trường hợp. Định nghĩa index nên đề cập đến cột có tính chọn lọc hơn đầu tiên để có thể được sử dụng làm một vị từ truy cập. Đó có thể là nguồn gốc của kinh nghiệm "sử dụng chọn lọc cao nhất đầu tiên" nhưng quy tắc này chỉ đúng nếu bạn không thể tránh được việc sử dụng vị từ lọc.

Tùy chọn khác là sử dụng hai index riêng biệt, một cho mỗi cột. Tiếp đó cơ sở dữ liệu phải quét cả hai index và rồi kết hợp các kết quả. Việc tra cứu từng index độc lập lặp lại đã mất nhiều công sức hơn vì cơ sở dữ liệu phải đi qua hai cây index. Ngoài ra, cơ sở dữ liệu cần rất nhiều bộ nhớ và thời gian CPU để kết hợp các kết quả trung gian.

Chú ý:

Quét một chỉ mục nhanh hơn quét hai chỉ mục.

Cơ sở dữ liệu sử dụng hai phương pháp để kết hợp các index. Thứ nhất là phép kết nối sử dụng index. Chương 4, "The Join Operation" giải thích các thuật toán liên quan một cách chi tiết. Cách tiếp cận thứ hai sử dụng phương pháp kỹ thuật trong lĩnh vực quản trị kho dữ liệu.

Kho dữ liệu là CSDL mà phải hỗ trợ nhiều các truy vấn ngẫu nhiên (ad-hoc). Chỉ cần một vài nhấp chuột là bạn có thể kết hợp điều kiện tùy ý vào truy vấn. Trong trường hợp này, sẽ là không thể dự đoán các kết hợp cột có thể xuất hiện trong mệnh đề **where** và dẫn tới việc tạo index phù hợp, như đã được giải thích trước, gần như không khả thi.

Kho dữ liệu sử dụng một loại index chỉ mục đặc biệt để giải quyết vấn đề đó: gọi là *bitmap index*. Ưu điểm của các bitmap index là chúng có thể được kết hợp khá dễ dàng. Điều đó có nghĩa là bạn có được hiệu suất truy vấn tốt ngay cả khi tạo index từng cột riêng lẻ. Ngược lại nếu bạn biết truy vấn trước, để bạn có thể tạo ra một B-tree index đa cột phù hợp, nó sẽ vẫn nhanh hơn việc kết hợp nhiều bitmap index.

Điểm yếu lớn nhất của bitmap index là việc hỗ trợ tối ưu cho **insert**, **update** và **delete**. Thao tác ghi đồng thời hầu như là không thể. Đó không phải là vấn đề trong kho dữ liệu vì dữ liệu thường được thu thập định kỳ, ít cập nhật. Trong các ứng dụng trực tuyến, bitmap index thường vô dụng, tốt thời gian duy trì lớn.

Điểm quan trọng:

Các bitmap index hầu như không sử dụng được cho xử lý giao dịch trực tuyến (OLTP).

Nhiều sản phẩm hệ quản trị cơ sở dữ liệu cung cấp một giải pháp lai giữa lập B-tree index và bitmap index. Trong trường hợp không có một phương án thực thi tốt hơn, chúng chuyên đổi các kết quả của nhiều thao tác quét B-tree vào cấu trúc bitmap trong bộ nhớ. Các cấu trúc bitmap không được lưu trữ lâu dài và được loại bỏ sau khi thực hiện các câu truy vấn, do đó bỏ qua vấn đề không có khả năng mở rộng của bitmap index. Nhược điểm là nó cần rất nhiều bộ nhớ và thời gian chạy CPU. Phương pháp này, sau tất cả là một hành động tuyệt vọng của bộ tối ưu.

21. Chỉ mục một phần

Cho tới hiện tại chúng ta chỉ mới thảo luận về lựa chọn các cột để tạo chỉ mục. Với chỉ mục một phần (*partial index* – PostgreSQL, *filtered index* – SQL Server), bạn cũng có thể lựa chọn các hàng để thêm vào chỉ mục.

Chú ý: Oracle database có cách tiếp cận riêng với chỉ mục một phần. Phần tiếp theo sẽ giải thích vấn đề đó dựa trên những gì đã được bàn luận tại đây.

Chỉ mục một phần thường được sử dụng trong các trường hợp mà mệnh đề where sử dụng các hằng số - giống như mã trạng thái trong ví dụ sau (thuộc tính processed):

```
SELECT message  
      FROM messages  
     WHERE processed = 'N'  
       AND receiver = ?
```

Những câu truy vấn như trên rất hay được sử dụng trong các hệ thống hàng đợi (queuing systems). Câu truy vấn này lấy ra tất cả các thông điệp chưa được xử lý cho một người nhận cụ thể. Những tin nhắn mà đã được xử lý thì hiếm khi được cần đến. Nếu được dùng đến thì chúng được truy xuất bởi những điều kiện cụ thể hơn như là khóa chính.

Chúng ta có thể tối ưu hóa câu truy vấn này bằng chỉ mục kết hợp trên hai cột. Nếu chỉ xem xét câu truy vấn này, thứ tự các cột là không quan trọng bởi vì không có điều kiện chọn theo khoảng.

```
CREATE INDEX messages_todo  
      ON messages (receiver, processed)
```

Chỉ mục này đáp ứng hoàn toàn yêu cầu, nhưng nó chứa nhiều hàng mà không bao giờ được tìm kiếm, cụ thể là những thông điệp mà đã được xử lý. Bởi vì độ phức tạp xử lý chỉ là hàm log nên dù sao thì chỉ mục này vẫn làm cho câu truy vấn chạy rất nhanh mặc dù nó lãng phí nhiều không gian lưu trữ.

Với chỉ mục một phần, bạn có thể giới hạn để chỉ mục chỉ bao gồm các thông điệp chưa được xử lý. Cú pháp của nó cực kỳ đơn giản: chỉ một mệnh đề where.

```
CREATE INDEX messages_todo  
      ON messages(receiver)  
     WHERE processed = 'N'
```

Chỉ mục này chỉ bao gồm những hàng mà thỏa mãn điều kiện where. Trong trường hợp này chúng ta thậm chí còn loại bỏ được cột PROCESSED bởi vì nó luôn luôn là 'N'. Có nghĩa là chỉ mục đã được giảm kích thước theo cả hai chiều: theo chiều dọc, vì nó chứa ít hàng hơn; theo chiều ngang, bởi vì cột được loại bỏ.

Do đó chỉ mục này rất nhỏ. Với một hàng đợi, nó thậm chí có nghĩa rằng kích thước của chỉ mục không đổi mặc dù kích thước bảng tăng rất nhanh mà không có giới hạn. Chỉ mục không bao gồm tất cả các thông điệp mà chỉ bao gồm các thông điệp chưa được xử lý.

Mệnh đề where trong chỉ mục một phần có thể phức tạp tùy ý. Chỉ có giới hạn duy nhất là về các hàm: bạn chỉ có thể sử dụng những hàm xác định như là một tiêu chuẩn chung trong mọi định nghĩa chỉ mục của các hệ quản trị cơ sở dữ liệu. Tuy nhiên, SQL Server có những quy định chặt chẽ hơn và không cho phép các hàm cũng như toán tử OR trong những điều kiện chỉ mục.

Một cơ sở dữ liệu có thể sử dụng chỉ mục một phần bất cứ khi nào có mệnh đề where trong câu truy vấn.

Câu hỏi: chỉ mục nhỏ nhất có thể trong câu truy vấn sau?

```
SELECT message  
FROM messages  
WHERE processed = 'N';
```

22. NULL trong cơ sở dữ liệu Oracle

Giá trị NULL trong SQL thường gây ra những nhấp nhằng. Mặc dù ý tưởng cơ bản của giá trị NULL – để biểu diễn dữ liệu bị thiếu – là khá đơn giản, có một vài dị thường. Ví dụ, bạn phải sử dụng IS NULL thay vì = NULL. Hơn thế nữa cơ sở dữ liệu Oracle có thêm những dị thường về giá trị NULL, một mặt bởi vì nó không phải lúc nào cũng xử lý giá trị NULL theo tiêu chuẩn, mặt khác nó có cách xử lý rất “đặc biệt” đối với giá trị NULL trong cấu trúc chỉ mục.

SQL tiêu chuẩn không định nghĩa NULL như là một giá trị mà thay vào đó là một sự thay thế cho những giá trị bị thiếu hoặc không biết. Do đó, không có giá trị nào có thể là NULL. Tuy nhiên, cơ sở dữ liệu Oracle lại coi một xâu rỗng là NULL:

```
SELECT '0 IS NULL???' AS "what is NULL?" FROM dual  
WHERE 0 IS NULL  
UNION ALL  
SELECT '0 is not null' FROM dual  
WHERE 0 IS NOT NULL  
UNION ALL  
SELECT """ IS NULL??? FROM dual  
WHERE " IS NULL  
UNION ALL  
SELECT """ is not null' FROM dual  
WHERE " IS NOT NULL;  
what is NULL?  
-----  
) is not null
```

" IS NULL???

Rắc rối hơn, thậm chí có trường hợp cơ sở dữ liệu Oracle còn xử lý giá trị NULL như là một xâu rỗng:

```
SELECT dummy ,  
       dummy || '' ,  
       dummy || NULL  
FROM dual;
```

```
D D D  
- - -  
X X X
```

Ghép nối DUMMY (luôn luôn chứa ‘X’) với NULL đáng lẽ ra phải trả về giá trị NULL.

Khái niệm NULL được sử dụng trong nhiều ngôn ngữ lập trình. Bạn có thể nhìn thấy ở bất kỳ đâu, một xâu rỗng không bao giờ là một giá trị NULL... ngoại trừ trong cơ sở dữ liệu Oracle. Nghĩa là, trong thực tế, bạn không thể nào lưu trữ một giá trị xâu rỗng trong một trường VARCHAR2. Nếu bạn làm thế, cơ sở dữ liệu Oracle chỉ lưu trữ giá trị NULL.

Điều kỳ dị này không chỉ lạ, mà nó còn nguy hiểm. Thêm vào đó, vấn đề giá trị NULL khác thường của cơ sở dữ liệu Oracle không dừng lại ở đây mà nó còn tiếp tục với việc đánh chỉ mục.

23. Đánh chỉ mục giá trị NULL

Cơ sở dữ liệu Oracle không thêm các hàng vào trong chỉ mục nếu mà các cột được sử dụng đánh chỉ mục đó đều nhận giá trị NULL. Điều đó có nghĩa là mọi chỉ mục đều là chỉ mục một phần – giống như có một mệnh đề WHERE:

```
CREATE INDEX idx  
    ON tbl (A, B, C, ...)  
WHERE A IS NOT NULL  
OR B IS NOT NULL  
OR C IS NOT NULL  
...;
```

Xem xét chỉ mục EMP_DOB. Nó chỉ có duy nhất một cột: DATE_OF_BIRTH. Một hàng mà không có giá trị DATE_OF_BIRTH sẽ không được thêm vào trong chỉ mục này.

```
INSERT INTO employees ( subsidiary_id, employee_id , first_name ,  
last_name , phone_number)  
VALUES ( ?, ?, ?, ?, ? );
```

Câu lệnh insert không xác định giá trị DATE_OF_BIRTH do đó giá trị mặc định là NULL – vì thế nó không được thêm vào trong chỉ mục EMP_DOB. Kết quả là chỉ mục không thể hỗ trợ một câu truy vấn cho những bản ghi mà có điều kiện DATE_OF_BIRTH = NULL:

```
SELECT first_name, last_name  
FROM employees  
WHERE date_of_birth IS NULL;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	477
* 1	TABLE ACCESS FULL	EMPLOYEES	1	477

Predicate Information (identified by operation id):

```
1 - filter("DATE_OF_BIRTH" IS NULL)
```

Tuy vậy, một bản ghi được thêm vào chỉ mục nếu có ít nhất một cột trong chỉ mục có giá trị khác NULL:

```
CREATE INDEX demo_null  
    ON employees (subsidiary_id, date_of_birth);
```

Hàng đã tạo bên trên sẽ được thêm vào chỉ mục bởi vì SUBSIDIARY_ID là khác NULL. Chỉ mục này có thể hỗ trợ truy vấn tất cả các nhân viên có giá trị trường subsidiary_id cụ thể và không có giá trị DATE_OF_BIRTH.

```
SELECT first_name, last_name  
FROM employees  
WHERE subsidiary_id = ? AND date_of_birth IS NULL;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	2
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	2
* 2	INDEX RANGE SCAN	DEMO_NULL	1	1

Predicate Information (identified by operation id):

```
2 - access("SUBSIDIARY_ID"=TO_NUMBER(?)  
AND "DATE_OF_BIRTH" IS NULL)
```

Chú ý rằng chỉ mục phủ toàn bộ mệnh đề where (index cover query); tất cả các điều kiện tìm kiếm đều được sử dụng trong INDEX RANGE SCAN.

Chúng ta có thể mở rộng ý tưởng này cho câu truy vấn lúc đầu để tìm kiếm tất cả các bản ghi mà có điều kiện DATE_OF_BIRTH IS NULL. Trong đó, cột DATE_OF_BIRTH phải nằm ở vị trí trái nhất trong chỉ mục để có thể sử dụng như điều kiện tìm kiếm trong chỉ mục (access predicate). Mặc dù chúng ta không dùng cột chỉ mục thứ hai cho bản thân câu truy vấn, chúng ta thêm một cột khác đảm bảo rằng cột này không bao giờ nhận giá trị NULL, do đó chỉ mục chứa tất cả các hàng.

Chúng ta có thể sử dụng bất kỳ cột nào có ràng buộc NOT NULL, như SUBSIDIARY_ID, cho mục đích đó.

Một cách khác chúng ta có thể dùng một biểu thức hằng, cái mà không bao giờ NULL. Điều này đảm bảo rằng chỉ mục chứa tất cả các hàng mặc dù cột DATE_OF_BIRTH có thể NULL.

```
DROP INDEX emp_dob;
```

```
CREATE INDEX emp_dob ON employees (date_of_birth, '1');
```

Về mặt kỹ thuật thì chỉ mục này được gọi là chỉ mục dựa trên hàm (function-based index). Ví dụ này phản bác quan điểm sai lầm rằng cơ sở dữ liệu Oracle không thể đánh chỉ mục giá trị NULL.

Mẹo: thêm một cột mà không bao giờ nhận giá trị NULL để đánh chỉ mục giá trị NULL như bất kỳ giá trị nào khác.

24. Ràng buộc NOT NULL

Để đánh chỉ mục cho một điều kiện IS NULL trong cơ sở dữ liệu Oracle, chỉ mục phải có ít nhất một cột không bao giờ nhận giá trị NULL.

Do đó, giá trị của cột không có mục nào NULL vẫn là chưa đủ. Cơ sở dữ liệu phải đảm bảo rằng cột đó không bao giờ có bất cứ một mục dữ liệu nào có thể nhận giá trị NULL, nếu không thì CSDL buộc phải chấp nhận rằng có thể có một hàng nào đó không có trong chỉ mục.

Chỉ mục sau chỉ hỗ trợ câu truy vấn chỉ khi cột LAST_NAME có ràng buộc NOT NULL.

```
DROP INDEX emp_dob;  
CREATE INDEX emp_dob_name  
    ON employees (date_of_birth, last_name);  
SELECT *  
    FROM employees  
    WHERE date_of_birth IS NULL;
```

[id]	Operation		Name	Rows	Cost
0 SELECT STATEMENT				1	3
1 TABLE ACCESS BY INDEX ROWID	EMPLOYEES			1	3
*2 INDEX RANGE SCAN	EMP_DOB_NAME	1		2	

Predicate Information (identified by operation id):

```
- access("DATE_OF_BIRTH" IS NULL)
```

Loại bỏ ràng buộc NOT NULL dẫn đến chỉ mục không thể được sử dụng cho câu truy vấn này.

```

ALTER TABLE employees MODIFY last_name NULL;
SELECT *
  FROM employees
 WHERE date_of_birth IS NULL;

```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	477
* 1	TABLE ACCESS FULL	EMPLOYEES	1	477

Mẹo: việc thiếu ràng buộc NOT NULL có thể ngăn cản việc sử dụng chỉ mục trong cơ sở dữ liệu Oracle – đặc biệt với các câu truy vấn count(*)).

Bên cạnh ràng buộc NOT NULL, cơ sở dữ liệu cũng biết rằng một biểu thức hằng giống như ở phần trước là không bao giờ NULL.

Tuy nhiên một chỉ mục trên hàm do người dùng định nghĩa không đảm bảo một ràng buộc NOT NULL trong biểu thức tạo chỉ mục.

```

CREATE OR REPLACE FUNCTION blackbox(id IN NUMBER) RETURN
NUMBER DETERMINISTIC
S BEGIN
  RETURN id;
END;
DROP INDEX emp_dob_name;
CREATE INDEX emp_dob_bb
  ON employees (date_of_birth, blackbox(employee_id));

```

```

SELECT *
  FROM employees
 WHERE date_of_birth IS NULL;

```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	477
* 1	TABLE ACCESS FULL	EMPLOYEES	1	477

Tên hàm BLACKBOX nhấn mạnh thực tế rằng bộ tối ưu hóa không biết về những gì được thực hiện trong hàm. Chúng ta có thể nhìn thấy hàm truyền thẳng giá trị đầu vào đến đầu ra, nhưng đối với cơ sở dữ liệu đó chỉ là một hàm mà trả về giá trị là một số. Thuộc tính NOT NULL của tham số bị mất. Mặc dù chỉ mục chứa tất cả các hàng, cơ sở dữ liệu không biết điều đó và do đó không thể dùng cho câu truy vấn.

Nếu bạn biết hàm không bao giờ NULL, như trong ví dụ này, bạn nên thay đổi câu truy vấn để thể hiện điều đó:

```

SELECT *
  FROM employees
 WHERE date_of_birth IS NULL
   AND blackbox(employee_id) IS NOT NULL;

```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	3
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	3
*2	INDEX RANGE SCAN	EMP_DOB_BB	1	2

Điều kiện bổ sung trong mệnh đề where luôn luôn đúng và do đó không thay đổi kết quả. Dù sao đi nữa thì cơ sở dữ liệu Oracle cũng nhận ra rằng bạn chỉ truy vấn trên những hàng mà phải nằm trong chỉ mục theo định nghĩa.

Đáng tiếc là không có cách nào để gán nhãn một hàm là không bao giờ trả về giá trị NULL, nhưng bạn có thể di chuyển lời gọi hàm đến một cột ảo (kể từ phiên bản 11g) và đặt ràng buộc NOT NULL trên nó.

```
ALTER TABLE employees ADD bb_expression
```

```

GENERATED ALWAYS AS (blackbox(employee_id)) NOT NULL;

DROP INDEX emp_dob_bb;
CREATE INDEX emp_dob_bb
    ON employees (date_of_birth, bb_expression);
SELECT *
    FROM employees
    WHERE date_of_birth IS NULL;

```

Id	Operation		Name	Rows	Cost
0	SELECT STATEMENT			1	3
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES		1	3
*2	INDEX RANGE SCAN	EMP_DOB_BB		1	2

Cơ sở dữ liệu Oracle biết rằng một số hàm nội tại chỉ trả về NULL nếu giá trị đầu vào là NULL.

```

DROP INDEX emp_dob_bb;
CREATE INDEX emp_dob_upname
    ON employees (date_of_birth, upper(last_name));
SELECT *
    FROM employees
    WHERE date_of_birth IS NULL;

```

Id	Operation	Name	Cost
0	SELECT STATEMENT		3
1	TABLE ACCESS BY INDEX ROWID EMPLOYEES		3
*2	INDEX RANGE SCAN	EMP_DOB_UPNAME	2

Hàm UPPER giữ nguyên thuộc tính NOT NULL của cột LAST_NAME. Tuy nhiên nếu loại bỏ ràng buộc trên cột này, dẫn đến không dùng được chỉ mục.

```
ALTER TABLE employees MODIFY last_name NULL;
SELECT *
    FROM employees
    WHERE date_of_birth IS NULL;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	477
* 1	TABLE ACCESS FULL EMPLOYEES 	1	477	

25. Mô phỏng chỉ mục một phần

Cách xử lý giá trị NULL khác thường của cơ sở dữ liệu Oracle có thể được sử dụng để mô phỏng chỉ mục một phần. Để làm được điều đó, chúng ta chỉ cần sử dụng giá trị NULL cho những hàng mà không nên nằm trong chỉ mục.

Để minh họa chúng ta mô phỏng chỉ mục một phần sau:

```
CREATE INDEX messages_todo
    ON messages (receiver)
    WHERE processed = 'N'
```

Đầu tiên chúng ta cần một hàm mà trả về giá trị cột RECEIVER chỉ khi cột PROCESSED là 'N'.

```
CREATE OR REPLACE
FUNCTION pi_processed(processed CHAR, receiver NUMBER)
RETURN NUMBER
DETERMINISTIC
```

```
AS BEGIN
    IF processed IN ('N') THEN
        RETURN receiver;
    ELSE
        RETURN NULL;
    END IF;
END;
```

/

Hàm này phải trả về giá trị xác định (deterministic) để có thể được sử dụng trong khai báo chỉ mục.

Bây giờ chúng ta có thể tạo một chỉ mục mà chỉ chứa những hàm có PROCESSED bằng 'N'.

```
CREATE INDEX messages_todo
    ON messages (pi_processed(processed, receiver));
```

Để sử dụng chỉ mục bạn phải sử dụng hàm đã được định nghĩa ở trên trong câu truy vấn:

```
SELECT message
      FROM messages
     WHERE pi_processed(processed, receiver) = ?
```

Id	Operation	Name	Cost
0	SELECT STATEMENT		5330
1	TABLE ACCESS BY INDEX ROWID	MESSAGES	5330
*2	INDEX RANGE SCAN	MESSAGES_TODO	5303

Predicate Information (identified by operation id):

2 - access("PI_PROCESSED"("PROCESSED","RECEIVER")=:X)

Chỉ mục một phần, phần II

Kể từ phiên bản 11g, có một cách tiếp cận thứ hai – phức tạp tương đương – để mô phỏng chỉ mục một phần trong cơ sở dữ liệu Oracle bằng việc sử dụng chỉ mục phân vùng (partitioned index) có chủ định và tham số SKIP_UNUSABLE_INDEX.

26. Những điều kiện bị mờ (obfuscated conditions)

Các phần dưới đây mô tả một vài phương pháp phổ biến cho những điều kiện bị mờ. Những điều kiện bị mờ là những mệnh đề where mà được trình bày theo một cách mà ngăn cản việc sử dụng chỉ mục đúng đắn. Phần này là một tập các khuôn mẫu sai lầm (anti-patterns) mà mọi developer nên biết và tránh.

27. Các kiểu dữ liệu

Hầu hết ofucation có liên quan đến các kiểu dữ liệu DATE. Cơ sở dữ liệu Oracle dễ bị tấn công về mặt này bởi vì nó chỉ có một kiểu dữ liệu DATE luôn luôn bao gồm thành phần thời gian đi cùng.

Thông thường, chúng ta sử dụng hàm TRUNC nhằm mục đích loại bỏ thành phần thời gian. Trong thực tế, TRUNC không loại bỏ thời gian nhưng thay vào đó là đặt lại nó vào lúc nửa đêm bởi vì cơ sở dữ liệu Oracle không có kiểu DATE thuận túy. Để bỏ qua thành phần thời gian cho một tìm kiếm, bạn có thể sử dụng hàm TRUNC trong cả hai phía của biểu thức so sánh – ví dụ, để tìm kiếm doanh số bán hàng ngày hôm qua:

```
SELECT ...
  FROM sales
 WHERE TRUNC(sale_date) = TRUNC(sysdate - INTERVAL '1' DAY)
```

Câu truy vấn trên hoàn toàn hợp lệ và chính xác nhưng nó thực sự không thể sử dụng được một index trên cột SALE_DATE. Điều này cũng được giải thích trong ‘Case-Insensitive Search Using UPPER or LOWER’ trong trang ?; TRUNC(sale_date) là hoàn toàn khác SALE_DATE – Hàm này là hộp đen đối với cơ sở dữ liệu.

Một cách đơn giản hơn để giải quyết vấn đề này là sử dụng một function-based index.

```
CREATE INDEX index_name
  ON table_name (TRUNC(sale_date))
```

Nhưng sau đó bạn luôn phải sử dụng TRUNC (date_column) trong mệnh đề **where**. Nếu bạn sử dụng nó không nhất quán – đôi khi sử dụng, đôi khi không sử dụng TRUNC – thì bạn cần hai index!

Vấn đề cũng có thể xảy ra với cơ sở dữ liệu có kiểu date rõ ràng nếu bạn tìm kiếm cho một khoảng thời gian dài hơn như trong câu truy vấn MySQL dưới đây:

```
SELECT ...
  FROM sales
 WHERE DATE_FORMAT(sale_date, "%Y-%M")
   = DATE_FORMAT(now()      , "%Y-%M")
```

Câu truy vấn sử dụng một định dạng date chỉ bao gồm năm và tháng: một lần nữa, câu truy vấn này hoàn toàn đúng với cùng một vấn đề như trước. Tuy nhiên, giải pháp ở trên không được áp dụng ở đây vì MySQL không có function-based index.

Sử dụng một khoảng điều kiện như một giải pháp thay thế. Đây chính là một phương pháp chung để làm việc với tất cả cơ sở dữ liệu.

```
SELECT ...
  FROM sales
 WHERE sale_date BETWEEN quarter_begin(?)
                      AND quarter_end(?)
```

Nếu bạn đã làm xong các bài tập về nhà của bạn, bạn chắc chắn nhận ra kiểu áp dụng từ bài tập cho câu truy vấn tìm tất cả nhân viên đang ở tuổi 42.

Một index trực tiếp trên cột SALE_DATE là đủ để tối ưu hóa câu truy vấn này. Các hàm QUARTER_BEGIN và QUARTER_END tính toán các giá trị biên. Sự tính toán có thể phức tạp hơn một chút bởi vì toán tử **between** luôn bao gồm các giá trị biên. Do đó, hàm QUARTER_END phải trả về một timestamp trước ngày đầu tiên của quý tiếp theo nếu SALE_DATE có thành phần thời gian. Logic này có thể bị ẩn đi trong chức năng này.

Các ví dụ tiếp theo chỉ ra sự thực thi các hàm QUARTER_BEGIN và QUARTER_END cho các cơ sở dữ liệu khác.

MySQL

```
CREATE FUNCTION quarter_begin(dt DATETIME)
RETURNS DATETIME DETERMINISTIC
RETURN CONVERT
(
    CONCAT
    ( CONVERT(YEAR(dt),CHAR(4))
    , '-'
    , CONVERT(QUARTER(dt)*3-2,CHAR(2))
    , '-01'
    )
    , datetime
);

CREATE FUNCTION quarter_end(dt DATETIME)
RETURNS DATETIME DETERMINISTIC
RETURN DATE_ADD
( DATE_ADD ( quarter_begin(dt), INTERVAL 3 MONTH )
, INTERVAL -1 MICROSECOND);
```

ORACLE DATABASE

```
CREATE FUNCTION quarter_begin(dt IN DATE)
RETURN DATE
AS
BEGIN
    RETURN TRUNC(dt, 'Q');
END;
/

CREATE FUNCTION quarter_end(dt IN DATE)
RETURN DATE
AS
BEGIN
    -- the Oracle DATE type has seconds resolution
    -- subtract one second from the first
    -- day of the following quarter
    RETURN TRUNC(ADD_MONTHS(dt, +3), 'Q')
        - (1/(24*60*60));
END;
/
```

POSTGRESQL

```
CREATE FUNCTION quarter_begin(dt timestamp with time zone)
RETURNS timestamp with time zone AS $$$
BEGIN
    RETURN date_trunc('quarter', dt);
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION quarter_end(dt timestamp with time zone)
RETURNS timestamp with time zone AS $$
BEGIN
    RETURN date_trunc('quarter', dt)
        + interval '3 month'
        - interval '1 microsecond';
END;
$$ LANGUAGE plpgsql;
```

SQL SERVER

```
CREATE FUNCTION quarter_begin (@dt DATETIME )
RETURNS DATETIME
BEGIN
    RETURN DATEADD (qq, DATEDIFF (qq, 0, @dt), 0)
END
GO

CREATE FUNCTION quarter_end (@dt DATETIME )
RETURNS DATETIME
BEGIN
    RETURN DATEADD
    (
        ms
        , -3
        , DATEADD(mm, 3, dbo.quarter_begin(@dt))
    );
END
GO
```

Bạn có thể sử dụng các hàm hỗ trợ tương tự cho các khoảng thời gian khác – hầu hết chúng sẽ ít phức tạp hơn các ví dụ ở trên, đặc biệt sử dụng điều kiện lớn hơn hoặc bằng (\geq) và nhỏ hơn ($<$) thay vì sử dụng **between**. Đương nhiên là bạn có thể tính toán các giá trị biên cho ứng dụng của bạn nếu muốn.

Mẹo

Viết câu truy vấn cho các khoảng thời gian liên tục với phạm vi điều kiện rõ ràng. Để làm được điều này dù chỉ một ngày – ví dụ, trong cơ sở dữ liệu Oracle:

```
sale_date >= TRUNC(sysdate)
AND sale_date < TRUNC(sysdate + INTERVAL '1' DAY)
```

Một obfuscation chung phổ biến khác là so sánh date như string được chỉ ra trong ví dụ trong PostgreSQL dưới đây:

```
SELECT ...
FROM sales
WHERE TO_CHAR(sale_date, 'YYYY-MM-DD') = '1970-01-01'
```

Lại một vấn đề về chuyển đổi DATE_COLUMN. Các điều kiện như vậy thường được tạo ra với niềm tin rằng không thể truyền các kiểu dữ liệu khác ngoài kiểu number và string trong cơ sở dữ liệu. Tuy nhiên, các tham số trói buộc hỗ trợ tất cả các kiểu dữ liệu. Điều đó có nghĩa là có thể sử dụng một đối tượng **java.util.Date** như một tham số ràng buộc. Đây chính là một lợi ích khác của các tham số ràng buộc.

Nếu không thể làm được điều đó, bạn có thể chuyển đổi điều kiện tìm kiếm thay cho cột của bảng:

```
SELECT ...
  FROM sales
 WHERE sale_date = TO_DATE('1970-01-01', 'YYYY-MM-DD')
```

Câu truy vấn này sử dụng đúng một index trực tiếp trên cột SALE_DATE. Hơn nữa, nó chuyển đổi chuỗi đầu vào từ chuỗi qua kiểu DATE chỉ một lần duy nhất. Trong câu truy vấn trước phải chuyển đổi tất cả các ngày được lưu trong bảng sang kiểu char trước khi chúng có thể so sánh với điều kiện tìm kiếm.

Bất cứ khi nào thực hiện thay đổi – sử dụng một tham số trói buộc hoặc chuyển đổi sang dạng so sánh khác – bạn có thể gặp phải một số lỗi nếu SALE_DATE có chứa thời gian. Bạn phải sử dụng một phạm vi điều kiện rõ ràng trong trường hợp đó:

```
SELECT ...
  FROM sales
 WHERE sale_date >= TO_DATE('1970-01-01', 'YYYY-MM-DD')
   AND sale_date <  TO_DATE('1970-01-01', 'YYYY-MM-DD')
                  + INTERVAL '1' DAY
```

Luôn luôn chú ý xem xét sử dụng một phạm vi điều kiện rõ ràng khi so sánh các ngày.

LIKE trong kiểu dữ liệu DATE

Obfuscation dưới đây đặc biệt phức tạp:

```
sale_date LIKE SYSDATE
```

Nó không giống obfuscation lăm bởi vì nó không sử dụng bất kì một hàm nào

Tuy nhiên, toán tử LIKE bắt buộc phải có một sự so sánh string. Phụ thuộc vào cơ sở dữ liệu, điều đó có thể gây ra lỗi hoặc gây ra một sự chuyển đổi hoàn toàn kiểu

trong cả hai bên. Phần “Predicate Information” của kế hoạch thực thi chỉ ra cái mà cơ sở dữ liệu Oracle làm:

```
filter (INTERNAL_FUNCTION (SALE_DATE)
       LIKE TO_CHAR (SYSDATE @!))
```

Chức năng của INTERNAL_FUNCTION là chuyển đổi kiểu dữ liệu của cột SALE_DATE. Hàm này một phần cũng ảnh hưởng đến việc ngăn chặn sử dụng một index trên DATE_COLUMN giống như bất kì hàm khác.

28. Kiểu chuỗi số

Chuỗi số là các số mà được lưu trữ trong các cột kiểu text. Mặc dù đó là một thói quen không tốt, nó chỉ không sử dụng index nếu chúng ta không tiếp tục coi nó như string.

```
SELECT ...
      FROM ...
 WHERE numeric_string = '42'
```

Đương nhiên, câu lệnh này có thể sử dụng một index trên NUMERIC_STRING. Tuy nhiên, nếu bạn dùng một số để so sánh chúng, cơ sở dữ liệu không còn có thể sử dụng điều kiện này như một vị từ truy cập.

```
SELECT ...
      FROM ...
 WHERE numeric_string = 42
```

Lưu ý các dấu trích dẫn bị mất. Mặc dù một số cơ sở dữ liệu gây ra lỗi (như PostgreSQL) nhiều cơ sở dữ liệu tự động thêm chuyển đổi kiểu dữ liệu ẩn.

```
SELECT ...
      FROM ...
 WHERE TO_NUMBER(numeric_string) = 42
```

Tương tự như vấn đề trước. Một index trên NUMERIC_STRING không được sử dụng bởi lời gọi hàm. Giải pháp xử lý vấn đề này cũng giống như trước: chuyển đổi điều kiện tìm kiếm thay vì chuyển đổi giá trị của cột trong bảng.

```
SELECT ...
  FROM ...
 WHERE numeric_string = TO_CHAR(42)
```

Có lẽ bạn muốn hỏi tại sao cơ sở dữ liệu không làm điều này một cách tự động. Đó là bởi vì khi chuyển một string sang một number luôn trả về một kết quả rõ ràng. Điều này không đúng theo chiều ngược lại. Một số được định dạng như là text có thể chứa khoảng trắng, các dấu câu và nhiều chữ số 0 đứng đầu. Một giá trị đơn lẻ có thể viết theo nhiều cách sau:

```
42
042
0042
00042
...
...
```

Cơ sở dữ liệu không biết được định dạng số đã dùng trong cột NUMERIC_STRING, vì thế nó có cách khác: cơ sở dữ liệu chuyển đổi string sang number – phép chuyển đổi này rất rõ ràng.

Hàm TO_CHAR trả về chỉ một chuỗi biểu diễn của số. Do đó, nó sẽ chỉ khớp với chuỗi đầu tiên được nêu ở trên (chuỗi 42). Nếu sử dụng TO_NUMBER, nó sẽ khớp với tất cả các giá trị. Điều đó có nghĩa là không những có sự khác nhau về hiệu năng mà hai biến thể này còn có một ngữ nghĩa khác!

Thông thường sử dụng chuỗi số sinh ra một số ván đề rắc rối: quan trọng nhất là nó gây ra một số ván đề về hiệu năng bởi sự chuyển đổi kiểu dữ liệu ẩn và cũng dẫn đến một rủi ro khi số không hợp lệ. Thậm chí câu truy vấn thông thường không sử dụng bất kì hàm nào trong mệnh đề **where** có thể gây ra lỗi trong quá trình chuyển đổi nếu có một số không hợp lệ được lưu trữ trong bảng.

MẸO

Sử dụng kiểu dữ liệu numeric để lưu trữ các số

Chú ý rằng ván đề không tồn tại trong trường hợp ngược lại:

```
SELECT ...
  FROM ...
 WHERE numeric_number = '42'
```

Cơ sở dữ liệu sẽ chuyển đổi nhất quán kiểu string sang number. Nó không áp dụng một hàm trên cột đánh chỉ mục index: do đó một index trên cột này là hưu dụng. Tuy nhiên rất có thể nếu chúng ta tự chuyển đổi, chúng ta có thể tạo ra một truy vấn tồi:

```
SELECT ...
FROM ...
WHERE TO_CHAR(numeric_number) = '42'
```

29. Các cột kết hợp

Phần này nói về một obfuscation phổ biến ảnh hưởng đến concatenated index.

Ví dụ đầu tiên này cũng nói về kiểu date và time nhưng theo một cách khác. Câu truy vấn MySQL dưới đây kết hợp một cột date và một cột time để áp dụng phạm vi lọc của chúng.

```
SELECT ...
FROM ...
WHERE ADDTIME(date_column, time_column)
    > DATE_ADD(now(), INTERVAL -1 DAY)
```

Câu truy vấn chọn tất cả các bản ghi trong 24 giờ qua. Nó thực sự không sử dụng một concatenated index trên (DATE_COLUMN, TIME_COLUMN) một cách đúng đắn bởi vì quá trình tìm kiếm không sử dụng các cột index nhưng sử dụng được dữ liệu đã được biến đổi.

Có thể phòng tránh vấn đề này bằng cách sử dụng một kiểu dữ liệu date bao gồm cả thành phần date và time (ví dụ như kiểu DATETIME trong MySQL). Có thể sử dụng cột datetime này mà không cần lời gọi hàm.

```
SELECT ...
FROM ...
WHERE datetime_column
    > DATE_ADD(now(), INTERVAL -1 DAY)
```

Thật không may mắn, thông thường chúng ta ít khi thay đổi cấu trúc của bảng dữ liệu.

Sự lựa chọn tiếp theo là một function-based index nếu cơ sở dữ liệu hỗ trợ nó – mặc dù điều này có tất cả các mặt hạn chế như đã thảo luận trước đó. Khi sử dụng MySQL, function-based indexes không là phương án lựa chọn vì MySQL không cung cấp.

Chúng ta vẫn có thể viết câu truy vấn để cơ sở dữ liệu có thể sử dụng một concatenated index trên DATE_COLUMN, TIME_COLUMN với một vị trí truy cập – ít nhất là một phần. Để làm được điều này, chúng ta bổ sung thêm điều kiện trên DATE_COLUMN.

```
WHERE ADDTIME(date_column, time_column)
    > DATE_ADD(now(), INTERVAL -1 DAY)
AND date_column
    >= DATE(DATE_ADD(now(), INTERVAL -1 DAY))
```

Điều kiện mới thêm vào này là hoàn toàn dư thừa nhưng nó là một bộ lọc trực tiếp trên DATE_COLUMN để có thể được sử dụng như một vị trí truy cập. Mặc dù kỹ thuật này không được tốt, nó cũng là một xấp xỉ đủ tốt.

MẸO

Sử dụng một điều kiện dư thừa trên hầu hết cột quan trọng khi trong mệnh đề WHERE có điều kiện khoảng kết hợp nhiều cột.

Trong PostgreSQL, chúng ta có thể sử dụng cú pháp giá trị hàng (row values syntax) được mô tả trong trang ?

Chúng ta cũng có thể sử dụng kỹ thuật này khi lưu date và time trong các cột kiểu text, nhưng vẫn phải sử dụng các định dạng date và time để tạo ra một trình tự thời gian khi sắp xếp. Ví dụ như được gọi ý bởi ISO 8601 (YYYY-MM-DD HH:MM:SS). Ví dụ dưới đây sử dụng hàm **TO_CHAR** trong cơ sở dữ liệu Oracle cho mục đích này:

```
SELECT ...
FROM ...
WHERE date_string || time_string
    > TO_CHAR(sysdate - 1, 'YYYY-MM-DD HH24:MI:SS')
AND date_string
    >= TO_CHAR(sysdate - 1, 'YYYY-MM-DD')
```

Chúng ta sẽ đối mặt với vấn đề áp dụng điều kiện khoảng trên nhiều cột ở trong toàn bộ phần “Paging Through Results”. Phương pháp lấy xấp xỉ cũng được sử dụng để làm giảm bớt điều này.

Đôi lúc chúng ta phải đảo ngược trường hợp và cố ý làm mờ (obfuscate) điều kiện lên để nó không thể được sử dụng như vị trí truy cập nữa. Chúng ta đã xem xét vấn đề đó khi thảo luận về sự ảnh hưởng của các tham số ràng buộc trong các điều kiện có LIKE. Xem xét ví dụ sau đây:

```
SELECT last_name, first_name, employee_id  
  FROM employees  
 WHERE subsidiary_id = ?  
   AND last_name LIKE ?
```

Giả sử rằng có một index trên SUBSIDIARY_ID và một index khác trên LAST_NAME, lựa chọn index trên cột nào thì tốt hơn cho truy vấn này?

Do chúng ta không biết vị trí của wildcard's trong điều kiện tìm kiếm, chúng ta không thể đưa ra được một đáp án chất lượng. Bộ tối ưu hóa không có sự lựa chọn khác hơn là phỏng đoán. Nếu bạn biết rằng luôn có một wildcard quan trọng, bạn có thể cố tình obfuscate làm cho điều kiện có **LIKE** trở nên phức tạp hơn để bộ tối ưu hóa không thể cân nhắc chọn index trên LAST_NAME.

```
SELECT last_name, first_name, employee_id  
  FROM employees  
 WHERE subsidiary_id = ?  
   AND last_name || '' LIKE ?
```

Chỉ cần thêm vào một chuỗi rỗng sau cột LAST_NAME như trên là đủ. Tuy nhiên, đây là sự lựa chọn cuối cùng. Chỉ làm nó khi thực sự cần thiết.

30. SMART LOGIC

Một số đặc chức năng quan trọng của các cơ sở dữ liệu SQL là nó hỗ trợ các câu truy vấn ad-hoc (ngắn nhanh): các câu truy vấn mới có thể được thực thi tại bất kỳ thời điểm nào. Điều này thực hiện được bởi vì bộ tối ưu hóa truy vấn làm việc khi chạy; nó phân tích mỗi câu truy vấn khi được nhận và ngay lập tức tạo ra một lược đồ thực thi thích hợp. Chi phí phát sinh do phải tính toán tối ưu có thể được giảm thiểu bằng cách sử dụng truy vấn với những tham số trói buộc.

Tóm lại, hệ quản trị cơ sở dữ liệu có kỹ thuật tối ưu hóa cho câu truy vấn SQL – vì thế, bạn không nhất thiết sử dụng tham số trói buộc nếu không cần.

Tuy nhiên có một thói quen được sử dụng rộng rãi để tránh SQL động, sử dụng SQL tĩnh – bởi vì sai lầm “SQL động thường chậm hơn so với SQL tĩnh”. Thói quen này có hại nhiều hơn có lợi nếu cơ sở dữ liệu có cơ chế lưu lại các lược đồ thực thi trên bộ nhớ đệm, được chia sẻ giữa các truy vấn tương tự nhau giống như DB2, cơ sở dữ liệu Oracle hoặc SQL Server.

Để chứng minh điều này, tưởng tượng một ứng dụng truy vấn trên bảng EMPLOYEES. Ứng dụng cho phép tìm kiếm trên subsidiary id, employee id và last name (không phân biệt chữ hoa hoặc chữ thường) trong bất kỳ sự kết hợp nào. Vẫn có

thể viết một câu truy vấn đơn mà bao phủ hết các trường hợp bằng cách sử dụng “smart” logic.

```
SELECT first_name, last_name, subsidiary_id, employee_id
  FROM employees
 WHERE ( subsidiary_id      = :sub_id OR :sub_id IS NULL )
   AND ( employee_id       = :emp_id OR :emp_id IS NULL )
   AND ( UPPER(last_name) = :name    OR :name    IS NULL )
```

Câu truy vấn sử dụng các biến trói buộc có tên để đọc tốt hơn. Tất cả các biểu thức lọc được mã hóa tĩnh trong câu lệnh. Bất cứ khi nào bộ lọc không cần thiết, sử dụng giá trị NULL thay vì điều kiện tìm kiếm: nó vô hiệu hóa điều kiện qua logic OR.

Câu truy vấn hoàn toàn hợp lý. Việc sử dụng NULL là phù hợp với định nghĩa ba giá trị logic của SQL. Tuy nhiên, nó cũng là một trong số các mô hình anti-pattern mang lại hiệu suất tồi nhất.

Cơ sở dữ liệu không thể tối ưu hóa cho một bộ lọc cụ thể bởi vì bất kì execution plan nào cũng có thể bị hủy bỏ trong thời gian chạy. Cơ sở dữ liệu cần chuẩn bị cho trường hợp xấu nhất – nếu tất cả các bộ lọc đều bị vô hiệu hóa.

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		2	478
* 1	TABLE ACCESS FULL	EMPLOYEES	2	478

Predicate Information (identified by operation id):

```
1 - filter(:NAME IS NULL OR UPPER("LAST_NAME")=:NAME)
      AND (:EMP_ID IS NULL OR "EMPLOYEE_ID"=:EMP_ID)
      AND (:SUB_ID IS NULL OR "SUBSIDIARY_ID"=:SUB_ID))
```

Như vậy, cơ sở dữ liệu quét hết bảng mặc dù mỗi cột đều có một index.

Điều đó không phải là do cơ sở dữ liệu không thể giải quyết được theo “smart” logic. Mà cơ sở dữ liệu tạo ra lược đồ thực thi chung do sử dụng các tham số ràng buộc, vì thế nó có thể được lưu và tái sử dụng với các giá trị khác sau đó. Nếu không sử dụng các tham số ràng buộc nhưng vẫn viết các giá trị thực tế trong câu lệnh SQL, bộ tối ưu hóa chọn index thích hợp hơn cho bộ lọc đang hoạt động:

```
SELECT first_name, last_name, subsidiary_id, employee_id
  FROM employees
 WHERE( subsidiary_id      = NULL      OR NULL IS NULL )
   AND( employee_id       = NULL      OR NULL IS NULL )
   AND( UPPER(last_name) = 'WINAND' OR 'WINAND' IS NULL )
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT			
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	1	2
*2	INDEX RANGE SCAN	EMP_UP_NAME	1	1

Predicate Information (identified by operation id):

2 - access(UPPER("LAST_NAME")='WINAND')

Tuy nhiên, đây không phải là giải pháp. Nó chỉ mới chứng minh rằng cơ sở dữ liệu có thể giải quyết các điều kiện này.

CẢNH BÁO

Sử dụng các giá trị văn bản làm cho ứng dụng dễ bị tấn công SQL injection và có thể gây ra các vấn đề về hiệu suất do tổng chi phí tối ưu tăng.

Rõ ràng giải pháp cho các truy vấn động là SQL động. Theo nguyên lý KISS (KISS principle wikipedia), chỉ cần đưa ra truy vấn cụ thể.

```
SELECT first_name, last_name, subsidiary_id, employee_id
  FROM employees
 WHERE UPPER(last_name) = :name
```

Lưu ý rằng câu truy vấn sử dụng một tham số ràng buộc.

MẸO

Sử dụng SQL động nếu cần mệnh đề **where** động.

Vẫn sử dụng các tham số trói buộc khi tạo ra SQL động – nếu không thì sai lầm “SQL động là chậm” thành thật.

Vấn đề được mô tả trong phần này rất phổ biến. Tất cả các cơ sở dữ liệu cache lại kế hoạch thực thi để chia sẻ giữa các truy vấn có chức năng để đối phó với các vấn đề này – thường xuyên gây ra các vấn đề mới và các lỗi mới.

MySQL

MySQL không xảy ra vấn đề này bởi vì nó không có bộ nhớ cache kế hoạch thực thi. Năm 2009, các nhà phát triển đã thảo luận về tác động của việc sử dụng lại các kế hoạch thực thi đã được cache. Có vẻ như bộ tối ưu hóa của MySQL đủ đơn giản để khiến cho việc cache các kế hoạch thực thi không đem lại hiệu quả rõ rệt.

ORACLE DATABASE

Cơ sở dữ liệu Oracle sử dụng một bộ nhớ đệm cho kế hoạch thực thi được chia sẻ và hoàn toàn bị tác động bởi vấn đề được mô tả trong phần này. Oracle giới thiệu blind peeking với phiên bản 9i. Blind peeking cho phép bộ tối ưu hóa sử dụng giá trị ràng buộc thực tế của lần thực thi đầu tiên khi chuẩn bị một kế hoạch thực thi. Vấn đề với cách tiếp cận này nằm ở tính không xác định ở hệ quả của tác động: các giá trị từ lần thực hiện đầu tiên ảnh hưởng đến tất cả các lần thực thi khác. Kế hoạch thực thi có thể thay đổi bất cứ khi nào cơ sở dữ liệu được khởi động lại hoặc ít nhất có thể đoán trước được, khi kế hoạch được lưu trong bộ nhớ cache hết hạn và bộ tối ưu hóa tạo lại một kế hoạch thực thi mới sử dụng các giá trị khác trong lần tiếp theo thực thi câu lệnh.

Phiên bản 11g đã giới thiệu tính năng con trỏ chia sẻ thích nghi (adaptive cursor sharing) để cải thiện hiệu năng hơn nữa. Đặc tính này cho phép cơ sở dữ liệu cache nhiều kế hoạch thực thi cho cùng một câu lệnh SQL. Hơn nữa, bộ tối ưu hóa có cơ chế xem lướt (nhìn nhanh) các tham số trói buộc và lưu lại độ chọn lọc ước lượng (estimated selectivity) cùng với kế hoạch thực thi. Khi bộ nhớ cache được truy cập sau đó, độ chọn lọc của các giá trị trói buộc hiện tại phải nằm trong khoảng chọn lọc của kế hoạch thực thi được lựa chọn tái sử dụng. Nếu không, bộ tối ưu hóa tạo ra một kế hoạch thực thi mới và so sánh nó với kế hoạch thực thi đã được lưu trong bộ nhớ đệm cho câu truy vấn này. Nếu đã có một kế hoạch thực thi như vậy, cơ sở dữ liệu thay thế nó với kế hoạch thực thi mới mà bao phủ độ chọn lọc ước lượng cho các giá trị tham số trói buộc hiện tại. Nếu không, nó lưu trữ một biến thể kế hoạch thực thi mới cho câu truy vấn này – cùng với các ước lượng chọn lọc.

POSTGRESQL

Bộ nhớ cache của kế hoạch truy vấn chỉ hoạt động cho các câu lệnh mở - miễn là bạn vẫn tiếp tục mở preparedStatement. Vấn đề đã mô tả ở trên chỉ xảy ra khi tái sử dụng một statement. Lưu ý rằng trình điều khiển JDBC của PostgreSQL cho phép bộ nhớ cache hoạt động chỉ sau 5 lần thực thi.

SQL SERVER

SQL Server sử dụng parameter sniffing. Tham số này cho phép bộ tối ưu hóa sử dụng các giá trị ràng buộc cho lần thực thi đầu tiên trong suốt quá trình phân tích cú pháp. Vấn đề tiếp cận theo cách này là hệ quả của nó có tính chất không

xác định. Kế hoạch thực thi có thể thay đổi bất cứ khi nào cơ sở dữ liệu được khởi động lại hoặc ít nhất có thể đoán trước được, hoặc kế hoạch được lưu trong bộ nhớ cache hết hạn và bộ tối ưu hóa tái tạo lại nó sử dụng các giá trị khác trong lần tiếp theo thực thi câu lệnh.

SQL Server 2005 đã bổ sung thêm các gợi ý truy vấn mới để có thể kiểm soát nhiều hơn qua các tham số parameter sniffing và biên dịch lại (compiling). Gợi ý truy vấn RECOMPILE bỏ qua bộ nhớ đệm kế hoạch cho một câu lệnh được chọn. OPTIMIZE FOR cho phép xác định các giá trị tham số thực tế được sử dụng chỉ cho sự tối ưu hóa truy vấn. Cuối cùng, chúng ta có thể cung cấp toàn bộ kế hoạch thực thi với gợi ý USE PLAN.

Cài đặt ban đầu của gợi ý OPTION (RECOMPILE) có một lỗi là nó không xem xét hết tất cả các biến trói buộc. Cách mới được sử dụng trong SQL Server 2008 có một số lỗi khác làm cho tình huống trở nên rất khó hiểu. ~~Erlend Sommarskog đã thử hợp tất cả thông tin liên qua bao gồm các bản phát hành của SQL Server~~.

Mặc dù phương pháp dựa theo kinh nghiệm (heuristic) có thể cải thiện vấn đề “smart logic” đến mức độ nhất định, nhưng thực tế chúng được xây dựng để giải quyết các vấn đề của tham số trói buộc liên quan đến histogram giá trị cả cột và các biểu thức có LIKE.

Phương pháp đáng tin cậy nhất để đạt được kế hoạch thực thi tốt nhất là tránh sử dụng các bộ lọc không cần thiết trong câu lệnh SQL.

31. Biểu thức toán học (MATH)

Có một lớp khác của vấn đề obfuscation này cũng ngăn chặn sử dụng index thích hợp. Đó là sử dụng biểu thức tính toán thay vì sử dụng các biểu thức logic.

Xem xét câu lệnh dưới. Có thể sử dụng index trên NUMERIC_NUMBER hay không?

```
SELECT numeric_number
      FROM table_name
 WHERE numeric_number - 1000 > ?
```

Tương tự, ví dụ dưới đây có thể sử dụng một index A và B – bạn chọn thứ tự nào?

```
SELECT a, b
      FROM table_name
 WHERE 3*a + 5 = b
```

Hãy đưa các câu hỏi cho các truy vấn này trong một góc nhìn khác; nếu bạn đang phát triển cơ sở dữ liệu SQL, bạn sẽ muốn bổ sung một phương trình không? Đa số các nhà

cung cấp cơ sở dữ liệu nói rằng “không!” và do đó, hai ví dụ trên đều không sử dụng index.

Chúng ta vẫn có thể sử dụng các phép toán học để làm rối obfuscate điều kiện – như đã làm trước đó với ví dụ tìm kiếm với LIKE. Chỉ cần thêm số 0 vào là đủ, ví dụ:

```
SELECT numeric_number  
      FROM table_name  
     WHERE numeric_number + 0 = ?
```

Tuy nhiên, chúng ta có thể lập chỉ mục các biểu thức này với function-based index nếu sử dụng tính toán một cách thông minh và biến đổi mệnh đề **where** như một phương trình:

```
SELECT a, b  
      FROM table_name  
     WHERE 3*a - b = -5
```

Rõ ràng chúng ta đã di chuyển các tham chiếu của bảng về một phía của phương trình và các hằng số về phía còn lại. Sau đó có thể tạo một function-based index cho phía bên trái của phương trình:

```
CREATE INDEX math ON table_name (3*a - b)
```

32. HIỆU NĂNG VÀ KHẢ NĂNG MỞ RỘNG

Chương này nói về hiệu năng và khả năng mở rộng của cơ sở dữ liệu.

Trong tình huống này, tôi sử dụng định nghĩa dưới đây cho khả năng mở rộng:

Khả năng mở rộng là khả năng của hệ thống, mạng, hoặc tiến trình của để xử lý một lượng công việc ngày càng tăng lên theo một phương pháp khả thi hoặc khả năng của nó được mở rộng để đáp ứng sự tăng trưởng đó. -Wikipedia

Bạn nhận ra rằng thực sự có hai định nghĩa về khả năng mở rộng. Cái thứ nhất là về tác động của việc gia tăng lượng tải trên hệ thống và cái thứ hai là phát triển hệ thống để xử lý nhiều lượng tải hơn.

Định nghĩa thứ hai được sử dụng phổ biến hơn định nghĩa thứ nhất. Bất cứ khi nào mọi người nói về khả năng mở rộng, thì hầu như luôn luôn được nói về việc sử dụng phần cứng nhiều hơn. Scale-up và scale-out là hai từ khóa riêng biệt được bổ sung bởi những thuật ngữ thông dụng mới như là web scale.

Nói rộng ra, khả năng mở rộng này nói về ảnh hưởng của hiệu năng từ những thay đổi môi trường. Phần cứng chỉ là tham số môi trường có thể thay đổi. Chương này bao gồm những thông số khác như là khối lượng dữ liệu và tải hệ thống.

33. Khối lượng dữ liệu ảnh hưởng đến hiệu năng

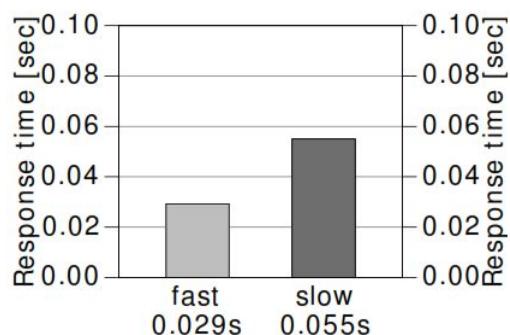
Lượng dữ liệu được lưu trữ trong cơ sở dữ liệu có ảnh hưởng lớn tới hiệu năng của hệ thống. Thông thường, một câu truy vấn trở nên chậm hơn khi tăng dữ liệu được thêm vào trong cơ sở dữ liệu. Những ảnh hưởng của hiệu năng sẽ lớn như thế nào nếu tập dữ liệu được tăng lên gấp đôi? Và làm thế nào chúng ta có thể cải thiện tỉ lệ này? Đây là những câu hỏi quan trọng khi thảo luận về sự mở rộng cơ sở dữ liệu.

Ví dụ, chúng tôi phân tích thời gian phản hồi của truy vấn sau đây khi sử dụng hai cách đánh chỉ mục khác nhau. Các định nghĩa về chỉ mục sẽ vẫn chưa được biết đến trong thời gian này - chúng sẽ được tiết lộ trong quá trình thảo luận.

```
SELECT count(*)
  FROM scale_data
 WHERE section = ?
   AND id2 = ?
```

Cột SECTION có một mục đích đặc biệt trong câu truy vấn này: nó quyết định khối lượng dữ liệu. Số lượng SECTION càng lớn thì càng có nhiều hàng truy vấn. Hình 3.1 cho biết thời gian trả lời của một SECTION nhỏ.

Hình 3.1 Sự so sánh hiệu năng



Có một sự khác biệt hiệu năng đáng kể giữa hai loại chỉ mục. Cả hai thời gian trả lời vẫn luôn giữ ở mức dưới 1/10 giây vì vậy thậm chí câu truy vấn được trả lời chậm hơn có lẽ vẫn đủ nhanh trong mọi trường hợp. Tuy nhiên biểu đồ hiệu năng chỉ cho thấy

một điểm cần kiểm tra. Khả năng mở rộng đang thảo luận có nghĩa là xem xét hiệu

QUAN TRỌNG

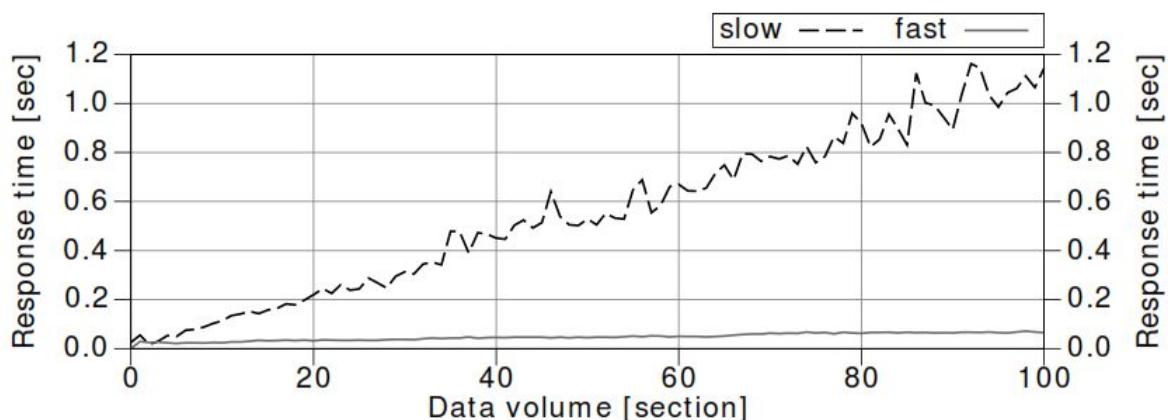
Khả năng mở rộng chỉ ra sự phụ thuộc của hiệu năng vào các yếu tố như là tập dữ liệu.

Một giá trị hiệu năng (performance value) chỉ là một điểm dữ liệu duy nhất trên biểu đồ thể hiện khả năng mở rộng.

quả hoạt động khi thay đổi tham số môi trường – chẳng hạn như khối lượng dữ liệu.

Hình 3.2 chỉ ra thời gian trả lời truy vấn theo số SECTION – liên quan đến sự gia tăng dữ liệu.

Hình 3.2 Khả năng mở rộng bởi tập dữ liệu



Biểu đồ cho thấy thời gian phản hồi ngày càng tăng đối với cả hai phương án đánh chỉ mục. Ở phía bên phải của biểu đồ, khi mà tập dữ liệu tăng lên 100 lần, thì câu truy vấn nhanh cần nhiều gấp đôi thời gian so với lúc đầu trong khi thời gian trả lời của câu truy vấn chậm tăng lên 20 lần, nhiều hơn một giây.

Thời gian phản hồi của câu truy vấn SQL phụ thuộc vào nhiều yếu tố. Khối lượng dữ liệu là một trong số đó. Nếu một câu truy vấn đủ nhanh dưới một điều kiện kiểm tra nhất định, thì không có nghĩa nó sẽ nhanh trong môi trường thực tế (production). Đó chỉ là một trường hợp đặc biệt trong môi trường phát triển chỉ có một phần dữ liệu của hệ thống sản xuất chạy thực tế.

Tuy nhiên, không ngạc nhiên rằng những câu truy vấn sẽ trở nên chậm hơn khi mà tập dữ liệu tăng. Nhưng sự khác biệt nổi bật giữa hai chỉ mục này là hơi bất ngờ. Đâu là lý do cho tỉ lệ khác nhau lớn như vậy?

Dễ dàng để nhận ra lý do đó bằng việc so sánh hai những phương án thực thi:

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	972
1	SORT AGGREGATE		1	
* 2	INDEX RANGE SCAN	SCALE_SLOW	3000	972

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	13
1	SORT AGGREGATE		1	
* 2	INDEX RANGE SCAN	SCALE_FAST	3000	13

Hai phương án thực thi này gần như giống nhau – chúng chỉ sử dụng một cách đánh chỉ mục khác nhau. Mặc dù các giá trị chi phí phản ánh sự khác biệt về tốc độ, nhưng lý do không được hiển thị trong kế hoạch thực thi.

Dường như chúng ta đang phải đối mặt với “trải nghiệm chỉ mục chậm”; câu truy vấn chậm mặc dù nó sử dụng một chỉ mục. ~~Tuy nhiên chúng tôi không tin tưởng vào huyền thoại của “chỉ số đầy” nữa.~~ Chúng ta nên nghi vấn vào hai thành phần làm cho việc tra cứu chỉ mục bị chậm: (1) truy cập bảng, và (2) việc quét một phạm vi rộng trên chỉ mục.

Nếu phương án thực thi không chỉ ra một thao tác TABLE ACCESS BY INDEX ROWID thì các phương án thực thi cần phải quét một phạm vi chỉ mục rộng hơn. Vậy kế hoạch thực thi sẽ hiển thị phạm vi chỉ mục được quét ở đâu? Tất nhiên là trong phần thông tin dự tính (predicate information).

LỜI KHUYÊN

Chú ý vào những thông tin dự tính.

Các thông tin dự tính không phải là một chi tiết không quan trọng mà bạn có thể bỏ qua. Một phương án thực thi mà không có thông tin dự tính là không hoàn thiện. Điều đó có nghĩa là bạn không thể nhận ra lý do cho sự khác biệt về hiệu suất trong các biểu đồ được trình bày ở trên. Nếu chúng ta nhìn vào kế hoạch thực thi đầy đủ, chúng ta có thể thấy sự khác biệt.

Chú ý

Phương án thực thi được đơn giản hóa để cho rõ ràng hơn. Phụ lục ở trang ? giải thích chi tiết cho phần “Predicate Information” trong một phương án thực thi của Oracle.

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	972
1	SORT AGGREGATE		1	
* 2	INDEX RANGE SCAN	SCALE_SLOW	3000	972

Predicate Information (identified by operation id):

```
2 - access("SECTION"=TO_NUMBER(:A))
      filter("ID2"=TO_NUMBER(:B))
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1	13
1	SORT AGGREGATE		1	
* 2	INDEX RANGE SCAN	SCALE_FAST	3000	13

Predicate Information (identified by operation id):

```
2 - access("SECTION"=TO_NUMBER(:A) AND "ID2"=TO_NUMBER(:B))
```

Bây giờ, sự khác biệt rõ ràng: chỉ có điều kiện trên SECTION là một thuộc tính truy cập khi sử dụng chỉ mục **SCALE_SLOW**. Cơ sở dữ liệu đọc tất cả các hàng từ section và loại bỏ những cái không khớp với vị trí lọc (filter predicate) trên ID2. Thời gian phản hồi tăng với số hàng trong section. Với chỉ mục **SCALE_FAST**, cơ sở dữ liệu sử dụng tất cả các điều kiện như các vị trí truy cập. Thời gian phản hồi tăng với số hàng được chọn.

QUAN TRỌNG

Các vị trí lọc giống như các thiết bị bom mìn chưa nổ. Chúng có thể phát nổ bất cứ lúc nào.

Những phần cuối cùng còn thiếu trong phần này là các định danh chỉ mục. Chúng ta có thể cấu trúc lại định nghĩa chỉ mục từ các phương án thực thi hay không?

Định nghĩa của chỉ mục **SCALE_SLOW** phải bắt đầu với cột SECTION – nếu không nó không thể được sử dụng như vị trí truy cập. Điều kiện trên ID2 không phải một vị trí truy cập vì nó không thể thực hiện theo SECTION trong định danh chỉ mục. Điều đó có nghĩa rằng chỉ mục **SCALE_SLOW** phải có tối thiểu 3 cột trong đó SECTION là phần thứ nhất và ID2 không phải phần thứ 2. Đó chính xác là cách chỉ mục được định nghĩa trong phân tích này:

- **CREATE INDEX scale_slow ON scale_data (section, id1, id2)**

Cơ sở dữ liệu không sử dụng ID2 như thuộc tính truy cập vì cột ID1 nằm ở vị trí thứ 2.

Định danh của chỉ mục SCALE_FAST phải có các cột SECTION và ID2 trong 2 vị trí đầu tiên bởi vì cả 2 được sử dụng như các vị từ truy cập. Dù sao chúng tôi cũng không thể nói bất cứ điều gì về câu lệnh của họ. Chỉ mục đã được sử dụng để phân tích trong ví dụ bắt đầu với cột SECTION và có bổ sung thêm cột ID1 vào vị trí thứ 3:

- **CREATE INDEX scale_fast ON scale_data (section, id2, id1)**

Cột ID1 được thêm vào để chỉ mục này có cùng kích thước như SCALE_SLOW- nếu không bạn có thể có ẩn tượng rằng kích thước chỉ mục khác nhau tạo ra ảnh hưởng khác nhau.

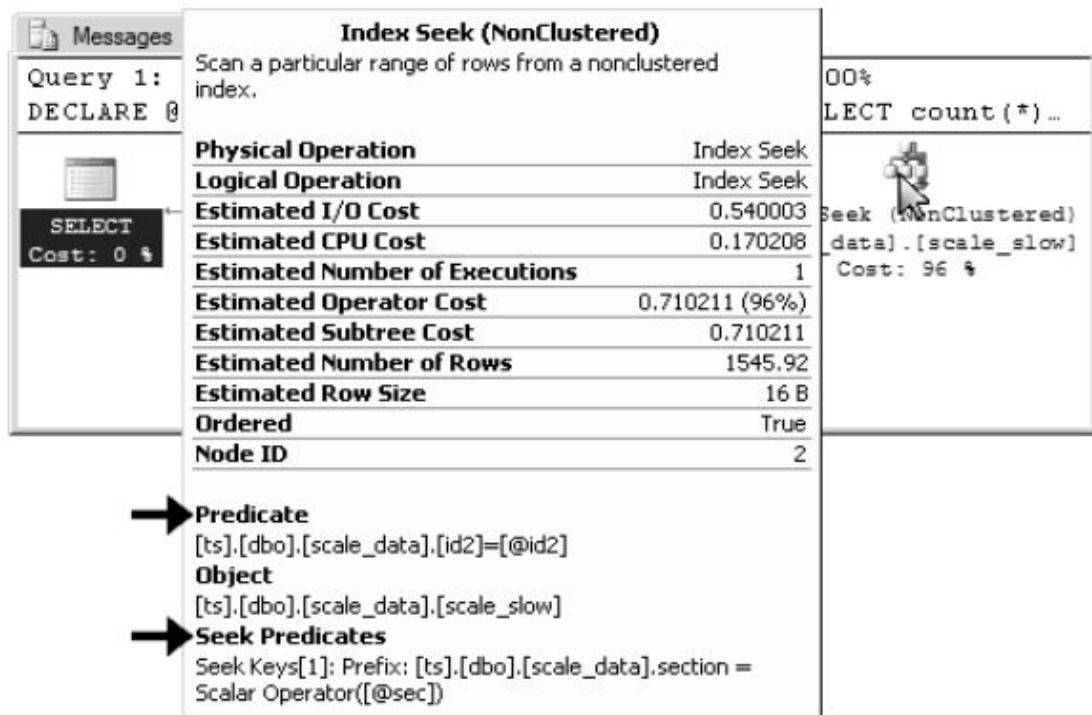
34. Những tác động tới hiệu năng của tải trên hệ thống

Việc xem xét làm thế nào để định nghĩa một chỉ mục đa cột thường dừng lại ngay khi chỉ mục sử dụng cho truy vấn được điều chỉnh. Tuy nhiên, bộ tối ưu hóa sử dụng chỉ mục không phải vì đó là phương án đúng, mà bởi vì nó là phương án hiệu quả hơn so với việc quét toàn bộ một bảng. Điều đó không có nghĩa là chỉ mục được sử dụng là tối ưu nhất cho câu truy vấn.

Ví dụ trước đã cho thấy những khó khăn khi chỉ ra sự sắp xếp không đúng của các cột trong một kế hoạch thực thi. Thường thì các thông tin dự tính hay bị ẩn đi vì vậy bạn phải tìm kiếm nó một cách cụ thể để xác nhận việc sử dụng chỉ mục là tối ưu.

Ví dụ khi sử dụng SQL Server Management Studio, ta có một cách để hiển thị các thông tin dự tính, đó là di chuyển con trỏ chuột ("hover") quanh vùng hoạt động của chỉ mục. Kế hoạch thực thi sau đây là sử dụng chỉ mục SCALE_SLOW, nó hiện thị các điều kiện trên ID2 như vị từ lọc (chỉ là "Predicate", mà không phải là tìm kiếm).

Figure 3.3. Predicate Information as a Tool Tip

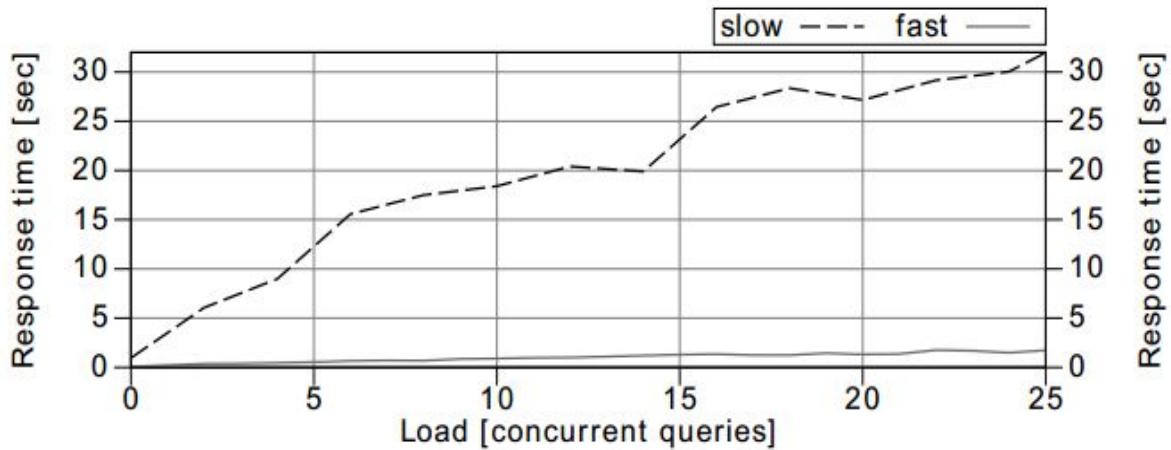


Việc thu thập thông tin dự tính từ một kế hoạch thực thi của MySQL hoặc PostgreSQL thì thậm chí còn kỳ cục hơn. *Xem thêm trang ? để biết thêm chi tiết.*

Mặc dù việc các thông tin dự tính xuất hiện trong kế hoạch thực thi không đáng kể, nhưng nó lại có tác động lớn đến hiệu suất, đặc biệt là khi hệ thống phát triển gia tăng. Nên nhớ rằng không chỉ có sự gia tăng của khối lượng dữ liệu mà còn có sự gia tăng của tỷ lệ truy cập. Đây là một tham số khác ảnh hưởng tới khả năng mở rộng (scalability).

Hình 3.4 biểu diễn thời gian phản hồi dưới dạng đồ thị hàm của tốc độ truy cập với khối lượng dữ liệu không đổi. Nó thể hiện thời gian thực hiện câu truy vấn với lượng dữ liệu lớn nhất. Điều đó có nghĩa là điểm cuối cùng từ hình 3.2 trang ? tương ứng với điểm đầu tiên trong biểu đồ này.

Figure 3.4. Scalability by System Load



Các đường nét đứt thể hiện thời gian phản hồi khi sử dụng chỉ mục SCALE_SLOW, nó đạt tới 32 giây khi thực hiện 25 câu truy vấn đồng thời. Đây là tính trong trường hợp không có load background, còn nếu trong môi trường chạy thực - thời gian có thể tăng gấp 30 lần. Ngay cả khi bạn có một bản sao chép đầy đủ cơ sở dữ liệu thì các tải nền trên server thực vẫn làm câu truy vấn chậm đi rất nhiều.

Các đường nét liền thì thể hiện thời gian phản hồi khi sử dụng chỉ mục SCALE_FAST, nó không có bất kỳ một vị từ lọc nào. Thời gian phản hồi vẫn giữ tốt dưới 2 giây kể cả khi chạy đồng thời 25 câu truy vấn.

Thời gian phản hồi đáng ngờ thường được xem nhẹ trong quá trình phát triển. Điều đó phần lớn là bởi vì chúng ta mong muốn "phần cứng mạnh mẽ hơn" mang lại hiệu suất tốt hơn. Nhưng thường thì đó không phải là việc cần được chú trọng vì môi trường chạy thực tế rất phức tạp và độ trễ tích lũy thường xảy ra trong môi trường phát triển. Ngay cả khi kiểm thử trên một cơ sở hạ tầng tương đương, các tải nền load background vẫn có thể gây ra thời gian phản hồi khác nhau. Trong phần tiếp theo chúng ta sẽ thấy rõ hơn lý do không nên mong đợi phản hồi nhanh hơn từ việc có phần cứng tốt hơn.

35. Thời gian phản hồi và thông lượng

Phần cứng lớn hơn không phải lúc nào cũng giúp nhanh hơn - nhưng nó có thể xử lý nhiều tải hơn. Phần cứng lớn hơn thì giống như bạn đi trên một con đường rộng hơn, chứ không phải lái một chiếc ô tô nhanh hơn. Bạn không thể lái nhanh hơn, vì tuy đường rộng hơn nhưng lại có nhiều làn hơn. Đó là lý do tại sao phần cứng không thể cải thiện tốc độ xử lý chậm của câu truy vấn.

Chúng ta không còn ở thời kỳ vào những năm 90. Khi sức mạnh tính toán của một core CPU cá nhân tăng trưởng một cách nhanh chóng. Nó giống như một chiếc ô tô

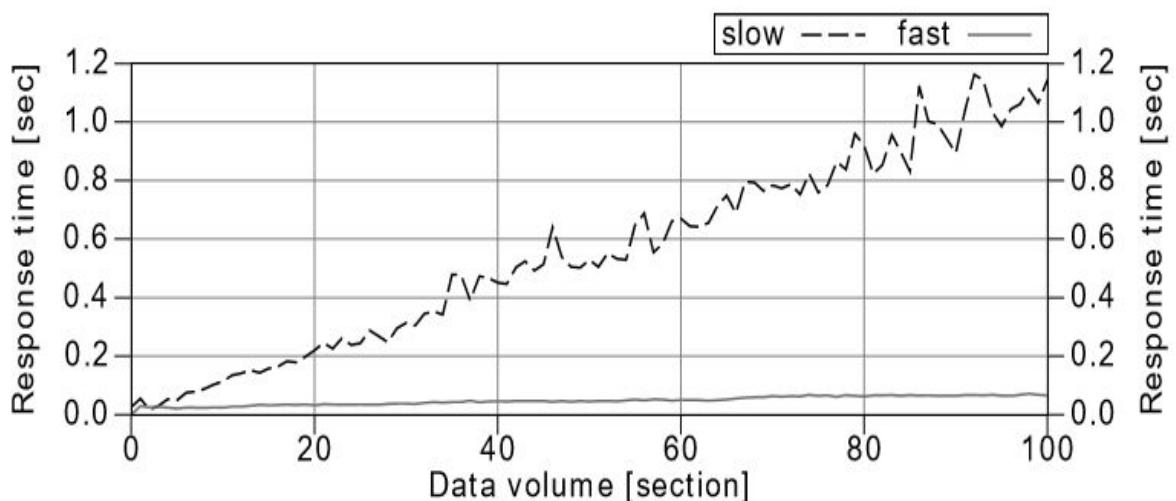
thế hệ mới chạy nhanh gấp hai lần thế hệ cũ - mỗi năm. Tuy nhiên, sức mạnh của CPU cá nhân đã chạm tới giới hạn trong những năm đầu của thế kỷ 21. Hầu như đã không còn sự phát triển trên phương diện này nữa. Nếu muốn tiếp tục phát triển sức mạnh của CPU, các nhà cung cấp phải hướng tới việc phát triển chiến lược đa lõi. Thậm chí khi có thể chạy đồng thời nhiều task, thì điều đó cũng không cải thiện hiệu suất khi chỉ có duy nhất một task. Hiệu suất không chỉ quyết định bởi duy nhất một tác nhân.

Mở rộng theo chiều ngang (gắn thêm nhiều server) cũng bị giới hạn tương tự. Mặc dù nhiều server có thể xử lý nhiều yêu cầu hơn, thì chúng cũng không cải thiện thời gian phản hồi của từng câu truy vấn. Để tăng tốc độ tìm kiếm, bạn cần một cây tìm kiếm tối ưu – thậm chí ngay cả trong hệ CSDL phi quan hệ như CouchDB và MongoDB.

Lưu ý: Lập chỉ mục phù hợp là cách tốt nhất để giảm thiểu thời gian phản hồi của truy vấn - trong các hệ cơ sở dữ liệu quan hệ SQL cũng như các hệ CSDL phi quan hệ.

Lập chỉ mục hợp lý nhằm khai thác tối đa hiệu quả của mở rộng logarit của chỉ mục B-tree. Không may việc lập chỉ mục thường được làm một cách cầu thả. Đồ thị ở phần "Performance Impacts of Data Volume" cho thấy rõ hiệu ứng của việc lập chỉ mục cầu thả, không xem xét kĩ càng các nhân tố.

Figure 3.5. Response Time by Data Volume



Thời gian phản hồi khác nhau giữa một chỉ mục cầu thả và một chỉ mục phù hợp. Thực khó để bù đắp sự khác biệt này bằng việc tăng thêm phần cứng. Thậm chí khi bạn giảm thiểu được thời gian phản hồi bằng phần cứng, thì đó chưa chắc đó là giải pháp tốt nhất cho vấn đề này.

Nhiều hệ thống NoSQL vẫn yêu cầu giải quyết mọi vấn đề hiệu suất bằng việc mở rộng chiều ngang. Tuy nhiên sự mở rộng này hầu hết chỉ giới hạn trong hoạt động ghi dữ liệu và áp dụng mô hình nhất quán sau cùng (eventual consistency model). Những cơ sở dữ liệu quan hệ sử dụng một mô hình nhất quán chặt (strong consistency model) mà có thể làm giảm tốc độ của tiến trình ghi nhưng điều đó không nhất thiết tác động xấu tới thông lượng. Hãy đọc phần "Eventual Consistency and the CAP Theorem" để biết thêm chi tiết.

Nhiều phần cứng hơn thường không cải thiện thời gian phản hồi. Trên thực tế, nó có thể khiến hệ thống chậm hơn vì sự phức tạp thêm có thể tích lũy thêm độ trễ. Độ trễ mạng sẽ không phải là vấn đề nếu ứng dụng và cơ sở dữ liệu chạy trên cùng một máy tính, nhưng kiểu cài đặt này thường không phổ biến trong các môi trường thực nơi chúng thường được cài đặt ở những phần cứng riêng biệt. Các chính sách bảo mật có thể yêu cầu tường lửa giữa server ứng dụng và cơ sở dữ liệu - điều đó thường làm tăng gấp đôi độ trễ mạng. Sự phức tạp hơn do cơ sở hạ tầng gây ra, tích lũy nhiều độ trễ hơn thì thời gian phản hồi sẽ chậm hơn. Điều này thường gây ra hiểu nhầm rằng phần cứng sản xuất đắt tiền chậm hơn môi trường PC rẻ tiền được sử dụng khi phát triển.

Mô hình nhất quán sau cùng và định lý CAP

Duy trì sự nhất quán (strong consistency) trong một hệ thống phân tán yêu cầu một sự điều phối đồng bộ của tất cả các thao tác ghi giữa các nút. Nguyên tắc này có 2 ảnh hưởng không tốt: (1) nó làm tăng độ trễ và tăng thời gian phản hồi; (2) nó làm giảm tính sẵn sàng toàn cục bởi vì nhiều thành viên phải hợp vào cùng một thời điểm để hoàn thành một thao tác ghi dữ liệu.

Một cơ sở dữ liệu SQL phân tán thường bị lẫn lộn với các cụm máy tính sử dụng một hệ thống lưu trữ chia sẻ hoặc một bản sao master-slave. Trong thực tế một cơ sở dữ liệu phân tán giống như một cửa hàng trực tuyến được tích hợp với một hệ thống ERP - thường là hai sản phẩm khác nhau từ các nhà cung cấp khác nhau. Sự nhất quán giữa cả hai hệ thống vẫn là một mục tiêu mong muốn, thường đạt được bằng cách sử dụng giao thức xác nhận hai bước (2PC). Giao thức này đã thiết lập các giao dịch toàn cục mà thực thi cách xử lý "all-or-nothing" nổi tiếng trên nhiều cơ sở dữ liệu. Việc hoàn tất giao dịch toàn cục chỉ có thể thực hiện nếu tất cả các thành viên đều sẵn sàng. Do đó làm giảm tính sẵn sàng toàn cục.

Một hệ thống phân tán có càng nhiều nút, sự nhất quán chặt càng trở nên rắc rối hơn. Bảo trì sự nhất quán chặt là gần như không thể nếu hệ thống có nhiều hơn một vài nút. Tuy nhiên, giảm đi sự nhất quán chặt chẽ, giải quyết được vấn đề tính sẵn sàng và có thể giảm thời gian phản hồi. Ý tưởng cơ bản là thiết lập lại sự nhất quán toàn cục sau khi hoàn thành hoạt động ghi trên mạng con của các nút. Cách tiếp cận này chỉ để lại một vấn đề chưa được giải quyết: không thể ngăn ngừa xung đột nếu hai nút chấp nhận các thay đổi mâu thuẫn. Sự nhất quán sau cùng đạt được bằng cách giải quyết các xung đột, chứ không phải bằng cách ngăn chặn chúng. Trong phạm vi đó, tính

nhất quán có nghĩa là tất cả các nút có cùng một dữ liệu - không nhất thiết là dữ liệu chính xác hoặc tốt nhất.

Định lý CAP của Brewer mô tả các phụ thuộc chung giữa Tính nhất quán, Tính sẵn sang và Chống phân mảnh.

Một độ trễ quan trọng khác là thời gian tìm kiếm trên ổ đĩa. Các ổ đĩa cứng quay (HDD) cần một khoảng thời gian khá dài để các bộ phận có thể đọc dữ liệu - thường là một vài mili giây. Độ trễ này xảy ra 4 lần khi duyệt cây B tree 4 cấp – tổng cộng: một vài chục mili giây. Độ trễ này là tương đối cao và dưới ngưỡng mong đợi khi thực hiện các truy vấn. Hơn nữa, rất dễ kích hoạt hàng trăm hoặc hàng ngàn đĩa tìm kiếm với một câu lệnh SQL duy nhất, đặc biệt khi liên kết nhiều bảng với câu lệnh kết nối.

Ổ ĐĨA TRẠNG THÁI RẮN VÀ BỘ NHỚ ĐỆM

Ổ đĩa trạng thái rắn (SSD) là một công nghệ lưu trữ hàng loạt mà không sử dụng các bộ phận chuyển động. Thời gian tìm kiếm đặc thù của SSD nhanh hơn thời gian tìm kiếm của ổ đĩa cứng (HDD). SSD đã được cung cấp từ năm 2010 nhưng do chi phí cao và thời gian sống ngắn nên không được sử dụng phổ biến cho lưu trữ cơ sở dữ liệu vào thời điểm đó.

Tuy nhiên, CSDL thường xuyên cache dữ liệu trên bộ nhớ RAM. Điều này đặc biệt hữu ích cho dữ liệu mà khi cần truy cập chỉ mục mỗi lần – ví dụ các nút gốc của chỉ mục. Cơ sở dữ liệu có thể hoàn toàn sử dụng bộ nhớ cache thường xuyên để tra chỉ mục mà không cần một đĩa tìm kiếm.

36. Phép toán kết nối



An SQL query walk into a bar and sees two tables.

He walks up to them and asks ‘Can I join you?’

- Nguồn: Internet –

Trong chương trước, chúng ta đã cùng nhau tìm hiểu qua về hiệu suất và khả năng mở rộng của cơ sở dữ liệu. Quay trở lại một chút với SQL, ở chương này, chúng ta sẽ nói về một phép toán hết sức quan trọng và cơ bản của SQL: Phép toán Join.

Trong SQL, để tránh việc bị trùng lặp dữ liệu, ta thiết kế cơ sở dữ liệu dưới các dạng chuẩn khác nhau. Dạng chuẩn càng cao, càng phải tách ra nhiều bảng. Tuy nhiên, khi làm các ứng dụng thực tế, ta vẫn thường phải tổ hợp dữ liệu từ nhiều bảng lại với nhau mới có được kết quả mong muốn. Để thực hiện điều này ta cần dùng phép toán Join kết nối các bảng lại với nhau.

Vậy, về cơ bản, phép toán Join là một phép toán chuyển đổi dữ liệu từ các trạng thái chuẩn sang trạng thái dữ liệu không chuẩn hóa, phục vụ cho mục đích xử lý cụ thể nào đó. Phép Join là một phép toán thường xuyên sử dụng trong SQL. Tốc độ thực hiện cũng phụ thuộc lớn vào cách phân bố và chiến lược tìm kiếm dữ liệu trên ô đĩa vì nó phải kết hợp nhiều đoạn dữ liệu trên các ngăn nhớ khác nhau của đĩa. Vì vậy, việc đánh chỉ mục (index) hợp lý sẽ là giải pháp tốt để tăng tốc độ câu truy vấn. Một chỉ mục được coi là hợp lý phụ thuộc vào một trong ba giải thuật Join phổ biến: Nested Loop, Hash Join và Sort Merge. Những giải thuật này sẽ được nêu ra ở phần sau.

Nói chung, các giải thuật Join trên đều chỉ thực hiện phép kết nối chỉ trên 2 bảng trong một thời điểm. Nếu phải kết nối nhiều hơn hai bảng, chúng sẽ phải thực hiện nhiều bước kết nối. Đầu tiên, SQL xây dựng một tập các kết quả trung gian sau khi join các bảng trước, bảng này sẽ được kết nối với bảng tiếp theo và cứ như vậy cho đến khi kết nối hết các bảng lại với nhau.

Mặc dù thứ tự các phép kết nối không ảnh hưởng tới kết quả cuối cùng, nhưng nó lại ảnh hưởng lớn tới hiệu năng của câu truy vấn. Bộ tối ưu sẽ tính toán tất cả các thứ tự kết nối có thể có và chọn ra thứ tự tốt nhất. Vậy làm thế nào để ta có thể chọn được thứ tự kết nối tối ưu?

Cách suy nghĩ đơn giản nhất là ta sẽ đánh giá tất cả các thứ tự kết hợp. Tuy nhiên, làm như vậy sẽ khó áp dụng cho các tình huống phức tạp, có nhiều phép kết nối, bởi lẽ ta sẽ phải tính tới $n!$ các cách kết nối khác nhau. Chi phí bỏ ra cho việc đánh giá như vậy là không cần thiết. Trên thực tế, ta sử dụng các tham số trói buộc (Bind Parameters) để giải quyết vấn đề này.



QUAN TRỌNG

Câu truy vấn càng phức tạp thì việc sử dụng các tham số trói buộc càng tỏ ra hiệu quả.

Không sử dụng tham số trói buộc để đánh giá chẳng khác gì việc phải biên dịch lại chương trình mỗi lần chạy.

Sử dụng cơ chế đường ống (pipelining) cho các kết quả trung gian

Trong giải thuật Join nhiều hơn 2 bảng, ta phải sử dụng các kết quả trung gian từ các phép Join trước để thực hiện phép Join với bảng tiếp theo. Mặc dù thuật ngữ “kết quả trung gian” diễn giải thuật toán tương đối tốt, tuy nhiên cơ sở dữ liệu không nhất thiết phải thực hiện hóa chúng một cách cụ thể rồi lưu trữ chúng để sử

dụng cho lần Join tiếp theo. Trên thực tế, ta sử dụng cơ chế đường ống lệnh nhằm giảm thiểu việc sử dụng bộ nhớ. Mỗi hàng từ kết quả trung gian sẽ ngay lập tức được chuyển đến phép Join kế tiếp, tránh việc lưu tập kết quả trung gian.

37. NESTED LOOPS (VÒNG LẶP LỒNG NHAU)

Trong các thuật toán Join thì Nested Loops Join là thuật toán cơ bản nhất. Cơ chế hoạt động của nó giống như việc sử dụng hai câu truy vấn lồng nhau: Câu truy vấn ngoài (Outer Query/ Driving Query) và câu truy vấn trong (Inner Query). Câu truy vấn ngoài sẽ lấy toàn bộ dữ liệu từ bảng thứ nhất. Ta tạm gọi kết quả của câu truy vấn ngoài là Outer Table. Với mỗi hàng của Outer Table, câu truy vấn trong sẽ quét từng dòng ở bảng còn lại để tìm dữ liệu tương ứng ghép nối.

for each row R1 in the outer table

for each row R2 in the inner table

if R1.join_column = R2.join_column

return (R1, R2)

Ta có thể sử dụng “Câu lệnh Selects lồng nhau” (**Nested Selects**) để cài đặt thuật toán Nested Loops. Tuy nhiên cách tiếp cận này không hiệu quả cho lắm do độ trễ trên ổ đĩa sẽ ảnh hưởng lớn tới thời gian thực hiện câu truy vấn. Trên thực tế, Nested Selects vẫn được sử dụng khá phổ biến bởi nó dễ dàng cài đặt.

Nested Selects gặp một vấn đề gọi là “N+1 Selects Problem” hay nói một cách ngắn gọn “N+1 Problem”. Để diễn giải vấn đề này, ta có thể sử dụng công cụ ánh xạ đối tượng – quan hệ ORM (Object relational mapping).

Ví dụ dưới đây chỉ ra chỉ ra cho chúng ta thấy vấn đề gặp phải khi sử dụng “Nested Select” với công cụ ORM.

Giả sử ta phải đưa ra tất cả các nhân viên có trong bảng employees với tên bắt đầu bằng “WIN” và lấy toàn bộ dữ liệu về SALES của những nhân viên này.

ORM không triển khai SQL Join. Thay vào đó, nó truy vấn bảng SALES với câu lệnh Select lồng nhau. Khi câu truy vấn Select ngoài (Outer Select) trả về kết quả N hàng, thì ta phải thực hiện thêm N câu lệnh Select cho mỗi hàng để tìm ra hàng cần ghép nối. Như vậy, ta sẽ cần toàn bộ N+1 câu lệnh Select nếu câu lệnh của Outer Select trả về N hàng.

Ví dụ về JPA sử dụng CriteriaBuilder interface

```
CriteriaBuilder queryBuilder = em.getCriteriaBuilder();
CriteriaQuery<Employees>
```

```
query = queryBuilder.createQuery(Employees.class);
Root<Employees> r = query.from(Employees.class); query.where(
queryBuilder.like(
    queryBuilder.upper(r.get(Employees_.lastName)),
    "WIN%"
)
);
```

```
List<Employees> emp = em.createQuery(query).getResultList();
```

```
for (Employees e: emp) {
    // process Employee
    for (Sales s: e.getSales()) {
        // process sale for Employee
    }
}
```

Hibernate JPA 3.6.0 sinh ra N+1 câu truy vấn Select

```
select employees0_.subsidiary_id as subsidiary1_0_
-- MORE COLUMNS
from employees employees0_
where upper(employees0_.last_name) like ?
```

```
select sales0_.subsidiary_id as subsidiary4_0_1_
    -- MORE COLUMNS
    from sales sales0_
where sales0_.subsidiary_id=?
and sales0_.employee_id=?
```

```
select sales0_.subsidiary_id as subsidiary4_0_1_
    -- MORE COLUMNS
    from sales sales0_
where sales0_.subsidiary_id=?
and sales0_.employee_id=?
```

Perl

Ví dụ dưới đây mô tả Perl's DBIx::Class framework:

```
my @employees =
    $schema->resultset('Employees')
        ->search({'UPPER(last_name)' => {-like=>'WIN%'}}); foreach my
$employee (@employees) {
    process Employee

    foreach my $sale ($employee->sales) { process Sale for
Employee
    }
}
```

DBIx::Class 0.08192 sinh ra N+1 câu truy vấn select:

```
SELECT me.employee_id, me.subsidiary_id
```

```
,    me.last_name, me.first_name, me.date_of_birth  
FROM employees me  
WHERE ( UPPER(last_name) LIKE ? )
```

```
SELECT me.sale_id, me.employee_id, me.subsidiary_id  
    , me.sale_date, me.eur_value FROM sales me  
WHERE ( ( me.employee_id = ?  
    AND me.subsidiary_id = ? ) )
```

```
SELECT me.sale_id, me.employee_id, me.subsidiary_id  
    , me.sale_date, me.eur_value  
FROM sales me  
WHERE ( ( me.employee_id = ?  
    AND me.subsidiary_id = ? ) )
```

PHP

Ví dụ mẫu về Doctrine sử dụng truy vấn builder interface:

```
$qb = $em->createQueryBuilder();  
$qb->select('e')  
    ->from('Employees', 'e')  
    ->where("upper(e.last_name) like :last_name")  
    ->setParameter('last_name', 'WIN%');  
$r = $qb->getQuery()->getResult();
```

```
foreach ($r as $row) {  
// process Employee  
    foreach ($row->getSales() as $sale) { // process Sale for  
        Employee
```

```
 }  
 }
```

Doctrine 2.0.5 sinh ra N+1 câu truy vấn select:

```
SELECT e0_.employee_id AS employee_id0 -- MORE COLUMNS  
FROM employees e0_  
WHERE UPPER(e0_.last_name) LIKE ?
```

```
SELE t0.sale_id AS SALE_ID1 --  
CT MORE COLUMNS  
FRO  
M sales t0  
WHE t0.subsidiary_i  
RE d = ?  
t0.employee_i  
AND d = ?  
SELE t0.sale_id AS SALE_ID1 --  
CT MORE COLUMNS  
FRO  
M sales t0  
WHE t0.subsidiary_i  
RE d = ?  
t0.employee_i  
AND d = ?
```



Bật
chế
độ
SQL

Logging

Trong quá trình phát triển phần mềm hay dự án, ta nên bật chế độ SQL Logging để xem câu lệnh SQL được tạo ra là gì, từ đó đưa ra phương án chỉnh sửa thích hợp.

DBIx::Class

`export BDIC_TRACE=1` trên cửa sổ dòng lệnh

DOCTRINE

```
$logger = new \Doctrine\DBAL\Logging\EchoSqlLogger;  
$config->setSQLLogger($logger);
```

HIBERNATE (NATIVE)

```
<property name="show_sql">true</property>
```

Câu hình trong App.config hoặc hibernate.cfg.xml

Cấu hình trong persistence.xml

```
<property name="eclipselink.logging.level" value="FINE"/>
<property name="hibernate.show_sql" value="TRUE"/>
<property name="openjpa.Log" value="SQL=TRACE"/>
```

Mặc dù cách tiếp cận theo “Nested Selects” chưa thực sự chuẩn xác, nhưng nó vẫn diễn giải khá tốt về kết nối bảng theo Nested Loops. Cơ sở dữ liệu thực thi phép Join giống như cách mà ORM đã làm như trên. Việc tạo Index cho Nested Loops join cũng giống như tạo Index trên các trường mà câu lệnh Select đưa ra ở ví dụ trên.

Ở đây, ta có thể đánh chỉ mục trên bảng EMPLOYEES và đánh 1 chỉ mục nối cho phép Join trên bảng SALES như sau:

```
CREATE INDEX emp_up_name ON employees (UPPER(last_name));
CREATE INDEX sales_emp ON sales (subsidiary_id, employee_id);
```

Công bằng mà nói, một phép kết nối SQL vẫn hiệu quả hơn cách tiếp cận theo kiểu truy vấn lồng (Nested Selects) kể cả khi thực hiện tìm kiếm trên cùng một Index. Bởi lẽ phép kết nối SQL giảm thiểu được trễ đường truyền. Chúng ta sẽ thấy tốc độ tăng rõ rệt khi dữ liệu cho câu truy vấn lớn vì càng có nhiều bản ghi Employee bị trùng lặp cho mỗi trường Sale.

Khi nói tới hiệu năng câu truy vấn, chúng ta cần quan tâm hai chiều: Thời gian phản hồi (Response time) và thông lượng (Throughput). Trong Mạng máy tính, ta gọi chúng là Độ trễ (latency) và Băng thông (Bandwidth). Băng thông có ảnh hưởng không đáng kể tới thời gian phản hồi, tuy nhiên độ trễ thì lại ảnh hưởng rất lớn. Như vậy, số lần tương tác với cơ sở dữ liệu ảnh hưởng tới thời gian phản hồi hơn so với lượng dữ liệu phải truyền đi. Ta cần giảm thiểu tối đa số lần tương tác qua lại với cơ sở dữ liệu.



MẸO

Thực thi phép Join ở trong cơ sở dữ liệu!

Hầu hết các công cụ ORM thường có những phương pháp nhất định để tạo phép kết nối SQL. Một trong những phương pháp quan trọng và phổ biến nhất là *eager-fetching*. Nó được triển khai ở mức thuộc tính trong ánh xạ thực thể (Entity mappings). Quay trở lại với ví dụ trước, ta triển khai đối với thuộc tính Employees trong lớp Sales. ORM sẽ tự động kết nối bảng EMPLOYEES khi có truy vấn tới bảng SALES. Eager-fetching sẽ chỉ thực sự có ý nghĩa nếu bạn luôn cần truy vấn thông tin chi tiết của Employee trên bảng Sale.

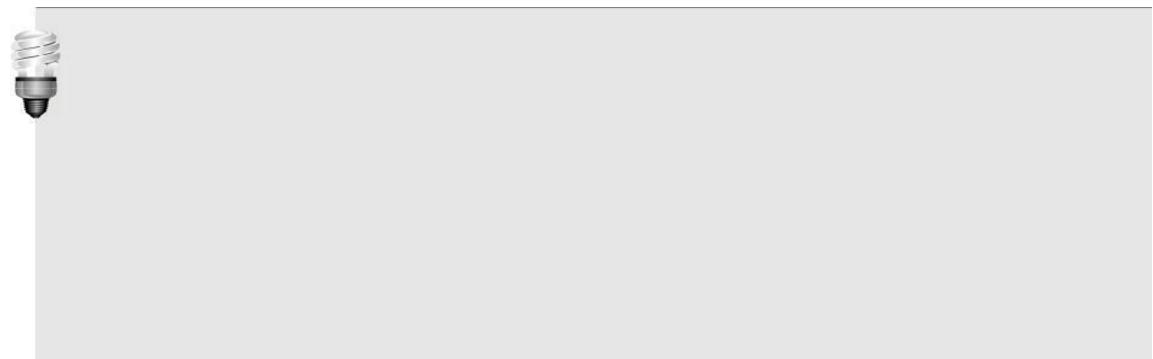
Eager fetching sẽ phản tác dụng trong trường hợp không nhất thiết phải lấy tất cả bản ghi con mỗi lần truy cập vào đối tượng cha. Chẳng hạn như ở ví dụ trên, nếu ta chỉ cần tạo ra một danh bạ điện thoại của các nhân viên mà lại phải lấy tất cả dữ liệu về SALES khi xem chi tiết về nhân viên đó thì thật không cần thiết. Chỉ một vài trường hợp ta cần lấy dữ liệu liên quan đến Sales của nhân viên. Cách cấu hình tĩnh như vậy chưa phải là một giải pháp tốt.

Để tối ưu hóa hiệu năng, bạn cần hiểu rõ phép kết nối. Sau đây sẽ là một số ví dụ cho chúng ta thấy cách tinh chỉnh phép kết nối một cách linh hoạt nhất tại thời điểm runtime.

JAVA

Trong Java, JPA CriteriaBuilder interface cung cấp phương thức *Root<>fetch()* giúp tinh chỉnh các phép kết nối. Phương thức này cho phép bạn cấu hình thời điểm, cách thức kết nối với các đối tượng được tham chiếu tới câu truy vấn chính.

Ví dụ sau, chúng ta sử dụng Left Join để lấy ra danh sách tất cả các nhân viên và dữ liệu Sales của họ kể cả những nhân viên không có dữ liệu Sales.



CẢNH BÁO

JPA và Hibernate trả về nhiều danh sách *employees cho mỗi dữ liệu Sale*.

Giả sử một nhân viên với 30 dữ liệu sale thì nhân viên đó sẽ xuất hiện 30 lần trong kết quả, mỗi dòng ứng với mỗi dữ liệu Sale của nhân viên đó. Điều này làm cho kết quả hiện lên khá rối mắt. Bạn có thể loại bỏ thủ công các bản sao ở quan hệ cha hoặc sử dụng hàm *distinct()* như trong ví dụ sau.

```

CriteriaBuilder qb = em.getCriteriaBuilder(); CriteriaQuery<Employees> q = qb.createQuery(Employees.class); Root<Employees> r = q.from(Employees.class); q.where(queryBuilder.like(
    queryBuilder.upper(r.get(Employees_.lastName)), "WIN%")
);

r.fetch("sales", JoinType.LEFT);
// needed to avoid duplication of Employee records
query.distinct(true);

```

List<Employees> emp = em.createQuery(query).getResultList();

Với Hibernate 3.6.0 :

```

select distinct
    employees0_.subsidiary_id as subsidiary1_0_0_
    , employees0_.employee_id as employee2_0_0_
    -- MORE COLUMNS
    , sales1_.sale_id as sale1_0_0_
from employees employees0_
left outer join sales sales1_
on employees0_.subsidiary_id=sales1_.subsidiary_id      and
employees0_.employee_id=sales1_.employee_id
where upper(employees0_.last_name) like ?

```

Để có thể lấy hết danh sách tất cả các nhân viên, câu truy vấn cần dùng kết nối trái (Left Join), và để loại bỏ các bản ghi trùng lặp, ta dùng thêm từ khóa *distinct*. Khi kết nối hai bảng có quan hệ một – nhiều (*one to many*) sẽ xảy ra hiện tượng trùng lặp 1 số bản ghi ở bảng có liên kết một (*one*) nhưng lại không bị trùng ở bảng liên kết nhiều (*many*). Tuy nhiên, JPA lại không cung cấp API nào thực sự tốt để lọc các bản ghi trùng lặp này. Khi dùng từ khóa *distinct*, chỉ một vài cơ sở dữ liệu có khả năng lọc trùng lặp bản ghi trên khóa chính, còn hầu hết chúng sẽ chỉ lọc các bản ghi thực sự trùng lặp hoàn toàn trên tất cả các trường. Vì thế, ta cần cân nhắc kỹ lưỡng trước khi dùng *distinct*.

Native Hibernate API giải quyết vấn đề này bên phía client bằng cách biến đổi tập kết quả thông qua phương thức *setResultTransformer()* :

```
Criteria c = session.createCriteria(Employees.class);
c.add(Restrictions.ilike("lastName", 'Win%'));

c.setFetchMode("sales", FetchMode.JOIN);
c.setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY);

List<Employees> result = c.list();
```

Nó sinh ra câu truy vấn sau:

```
select this_.subsidiary_id as subsidiary1_0_1_
, this_.employee_id as employee2_0_1_
-- MORE this_ columns on employees
, sales2_.sale_id as sale1_3_
-- MORE sales2_ columns on sales from employees
this_
```

```
left outer join sales sales2_
on this_.subsidiary_id=sales2_.subsidiary_id
and this_.employee_id=sales2_.employee_id      where
lower(this_.last_name) like ?
```

Phương thức này cung cấp trực tiếp cho SQL mà không cần bắt cú mệnh đề mở rộng nào. Ta cần lưu ý thêm rằng Hibernate sử dụng phương thức *lower()* dùng cho những câu truy vấn trả về kết quả không phân biệt chữ hoa chữ thường. Đây là cơ sở quan trọng trong việc đánh chỉ mục theo kiểu function-based.

PERL

Với Perl's DBIx::Class Framework:

```
my @employees =
$schema->resultset('Employees')
->search({ 'UPPER(last_name)' => {-like => 'WIN%'}}
```

```
,      {prefetch => ['sales']} );
```

DBIx::Class 0.08192 sinh ra câu lệnh SQL:

```
SELECT me.employee_id, me.subsidiary_id, me.last_name  
      --MORE COLUMNS  
FROM employees me  
LEFT JOIN sales sales  
ON (sales.employee_id      = me.employee_id  
AND      sales.subsidiary_id = me.subsidiary_id)  
WHERE ( UPPER(last_name) LIKE ? )  
ORDER BY sales.employee_id, sales.subsidiary_id
```

Lưu ý rằng trong ví dụ này, ta không nhất thiết phải sử dụng mệnh đề ORDER BY. Việc sắp xếp kết quả chỉ nhằm làm cho tập kết quả sinh ra dễ quan sát hơn mà thôi.

PHP

Ví dụ sau sử dụng Doctrine framework của PHP:

```
$qb = $em->createQueryBuilder();  
$qb->select('e,s')  
      ->from('Employees', 'e')  
      ->leftJoin('e.sales', 's')  
      ->where("upper(e.last_name) like :last_name")  
      ->setParameter('last_name', 'WIN%');  
$r = $qb->getQuery()->getResult();
```

Doctrine 2.0.5 sinh ra câu lệnh SQL sau:

```
SELECT e0_.employee_id AS employee_id0  
      -- MORE COLUMNS
```

```
FROM employees e0_ LEFT JOIN  
sales s1_  
  
ON e0_.subsidiary_id = s1_.subsidiary_id AND  
e0_.employee_id = s1_.employee_id  
  
WHERE UPPER(e0_.last_name) LIKE ?
```

Kế hoạch thực thi chỉ ra cách thức NESTED LOOPS OUTER hoạt động:

Id	Operation	Name	Rows	Cost
<hr/>				
0	SELECT STATEMENT		822	38
1	NESTED LOOPS		822	38
	OUTER			

	TABLE	BY	INDEX	EMPLOY			
2	ACCESS	ROWID		EES		1	4
	INDEX			EMP_UP_NA			
*3	RANGE	SCAN		ME		1	
	TABLE	BY	INDEX				
4	ACCESS	ROWID		SALES		821	34
	INDEX			SALES_E			
*5	RANGE	SCAN		MP		31	

Predicate Information (identified by operation id):

```
3 - access(UPPER("LAST_NAME") LIKE 'WIN%')  
filter(UPPER("LAST_NAME") LIKE 'WIN%')
```

```
5 - access("E0_"."SUBSIDIARY_ID"="S1_"."SUBSIDIARY_ID"(+)
          AND "E0_"."EMPLOYEE_ID" ="S1_"."EMPLOYEE_ID"(+))
```

Ta thấy, trên bảng EMPLOYEES có đánh chỉ mục cho Last_Name. Cơ sở dữ liệu sẽ lấy kết quả của bảng EMPLOYEES thông qua chỉ mục EMP_UP_NAME trước, sau đó nó tìm trên bảng SALES dữ liệu tương ứng cho mỗi dữ liệu của Employee tìm được.



MẸO

Hãy tìm hiểu về ORM và cách để điều khiển các phép kết nối thông qua nó.

Nested loops join sẽ thực sự mang lại hiệu quả nếu như câu truy vấn ngoài (Driving query) trả về một tập kết quả nhỏ. Nếu tập kết quả trả về lớn, bộ tối ưu sẽ cân nhắc lựa chọn một giải thuật kết nối khác, như Hash Join chẳng hạn. Các giải thuật này sẽ được mô tả kỹ hơn ở phần sau.

38. HASH JOIN (KẾT NỐI BĂM)

Thuật toán kết nối băm (hash join) nhằm khắc phục những điểm yếu của phép kết nối lồng gây ra: phải duyệt qua nhiều cây B-tree khi thực hiện câu truy vấn con. Thay vào đó hash join sẽ lấy các bản ghi từ một bảng của phép kết nối cho vào bảng băm (các bản ghi được băm dựa vào trường được join), nhờ đó có thể duyệt rất nhanh mỗi hàng trong bảng còn lại của phép kết nối. Thiết kế hash join yêu cầu phương pháp đánh chỉ mục khác hoàn toàn so với phép kết nối lồng (nested loops join). Ngoài ra, ta có thể cải thiện hiệu suất của hash join bằng cách chọn ít cột hơn để băm – một thách thức đối với hầu hết công cụ ORM.

Chiến lược đánh index của hash join rất khác vì nó không cần đánh index đối với các cột được join mà đánh index cho vị từ độc lập trong mệnh đề where sẽ cải thiện hiệu năng của hash join.



MẸO

Đánh index cho vị từ độc lập trong mệnh đề where giúp cải thiện hiệu năng của hash join.

Xem xét ví dụ sau: Lấy tất cả các doanh số bán hàng trong 6 tháng gần đây tương ứng với từng nhân viên.

```
SELECT *
  FROM sales s
  JOIN employees e ON (s.subsidiary_id = e.subsidiary_id
                      AND s.employee_id = e.employee_id )
 WHERE s.sale_date > trunc(sysdate) - INTERVAL '6' MONTH
```

Điều kiện chọn SALE_DATE là vị từ độc lập trong mệnh đề **where**, nghĩa là nó chỉ tham chiếu đến một bảng (bảng SALES) và không nằm trong các vị từ nối (subsidiary_id, employee_id).

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT		49244	59M	12049
* 1	HASH JOIN		49244	59M	12049
2	TABLE ACCESS FULL	EMPLOYEES	10000	9M	478
* 3	TABLE ACCESS FULL	SALES	49244	10M	10521

Predicate Information (identified by operation id):

```

1 - access("S"."SUBSIDIARY_ID"="E"."SUBSIDIARY_ID"
           AND "S"."EMPLOYEE_ID" = "E"."EMPLOYEE_ID")
3 - filter("S"."SALE_DATE">>TRUNC(SYSDATE@!)
           -INTERVAL '+00-06' YEAR(2) TO MONTH)

```

Bước thực thi đầu tiên là duyệt toàn bộ bảng EMPLOYEES để lấy tất cả các bản ghi nhân viên vào một bảng băm (plan id 2). Bảng băm sử dụng vị từ nối như là khóa để băm. Ở bước tiếp theo, quét toàn bộ bảng SALES và loại bỏ tất cả các bản ghi SALES không thỏa mãn điều kiện trên SALE_DATE (plan id 3). Đối với các bản ghi SALES còn lại ta sẽ truy cập bảng băm để đưa ra các nhân viên tương ứng với bản ghi đó.

Mục đích duy nhất của bảng băm là hoạt động như một cấu trúc lưu trữ tạm thời cho bộ nhớ để tránh truy cập vào bảng EMPLOYEE nhiều lần. Vì tất cả bản ghi (trong một bảng của phép join) đều được nạp vào bảng băm nên ta không cần đánh index cho các bản ghi đó. Trong mệnh đề where không có điều kiện lọc nào được áp dụng trên bảng EMPLOYEES (plant id 2). Và câu truy vấn cũng không có bất kỳ vị từ độc lập nào trên bảng này.



QUAN TRỌNG

Việc đánh index cho các vị từ kết nối không cải thiện hiệu năng của hash join.

Điều này không có nghĩa sẽ là không thể để đánh index trong hash join. Các vị từ độc lập có thể được đánh index. Đây là điều kiện quyết định bảng nào trong hai bảng được đánh index. Trong ví dụ trên có thể đánh index trên trường SALE_DATE trong bảng SALES. Điều này giúp lọc các bản ghi không thỏa mãn điều kiện SALE_DATE hiệu quả hơn.

CREATE INDEX sales_date ON sales (sale_date);

Bên dưới là execution plan khi sử dụng index (cost khi duyệt bảng SALES giảm từ 10521 xuống 1724). Tuy nhiên ta vẫn sẽ duyệt toàn bộ bảng EMPLOYEES vì câu truy vấn không có vị trí độc lập nào trong mệnh đề where và nằm trên bảng EMPLOYEES.

Id	Operation		Name	Bytes	Cost
		SELECT		59M	
0	STATEMENT			3252	
		HASH		59M	
* 1	JOIN			3252	
		TABACC	EMPLOY		
2	LE ESS FULL	EES	9M	478	
	TABACC BY INDEX SALE		10M		
3	LE ESS ROWID S		1724		
	INDEX	SALES_D			
* 4	RANGE SCAN ATE				

Predicate Information (identified by operation id):

- 1 - access("S"."SUBSIDIARY_ID"="E"."SUBSIDIARY_ID" AND "S"."EMPLOYEE_ID"="E"."EMPLOYEE_ID")
- 4 - access("S"."SALE_DATE" > TRUNC(SYSDATE@!) - INTERVAL '+00-06' YEAR(2) TO MONTH)

Đánh index trong hash join trái ngược với nested loops join. Bởi vì thứ tự kết nối không ảnh hưởng đến việc đánh index. Index trên trường SALES_DATE có thể sử dụng để tải bảng băm ngay cả khi thứ tự join bị đảo ngược.



CHÚ Ý

Đánh index trong hash join không phụ thuộc vào thứ tự join.

Một phương pháp khác để tối ưu hiệu năng của hash join là giảm kích thước của bảng băm. Hash join là tối ưu khi toàn bộ bảng băm được đưa vừa vặn vào bộ nhớ. Khi join giữa các bảng, bộ tối ưu sẽ chọn bảng có kích thước nhỏ để đưa vào bảng băm. Công cụ Oracle execution plan cho ta thấy ước lượng bộ nhớ cần thiết trong cột “Bytes”. Trong quá trình thực thi câu truy vấn trên, bảng EMPLOYEES cần 9MB, nhỏ hơn so với bảng SALES.

Cũng có thể giảm kích thước của bảng băm bằng cách thay đổi câu truy vấn SQL, ví dụ bằng cách thêm các điều kiện bổ sung để cơ sở dữ liệu tải ít bản ghi vào bảng băm hơn. Tiếp tục với ví dụ trên, việc thêm một điều kiện trên thuộc tính DEPARTMENT trong bảng EMPLOYEES để chỉ nhân viên bán hàng mới được xét (department = “sales staff”). Hiệu năng của hash join được cải thiện ngay cả khi không có index trên thuộc tính DEPARTMENT vì cơ sở dữ liệu không cần lưu các nhân viên mà không phải là nhân viên bán hàng trong bảng băm. Khi làm như vậy ta phải chắc chắn không có bất kỳ bản ghi SALES nào mà nhân viên không thuộc một bộ phận nào cả. Ta cần sử dụng các ràng buộc để đảm bảo vấn đề trên không xảy ra.

Khi thu gọn kích thước bảng băm, yếu tố quan trọng không phải số lượng bản ghi mà là dung lượng bộ nhớ nó chiếm. Trên thực tế, vẫn có thể giảm kích thước bảng băm bằng cách chọn ít cột hơn - chỉ những thuộc tính thực sự cần thiết:

```
SELECT s.sale_date, s.eur_value  
      , e.last_name, e.first_name  
      , FROM sales s  
  
JOIN employees e ON (s.subsidiary_id = e.subsidiary_id  
AND s.employee_id = e.employee_id )  
WHERE s.sale_date > trunc(sysdate) - INTERVAL '6' MONTH
```

Phương pháp này rất có thể gây lỗi vì chỉ cần bỏ đi nhầm cột (SELECT không đúng hoặc thiếu cột) thì sẽ nhận ngay được thông báo lỗi truy vấn. Kết quả giúp thu gọn kích cỡ của bảng băm một cách đáng kể, trong trường hợp trên giảm từ 9 MB xuống còn 234 KB – giảm đi 97%

Id Operation	Name	Bytes Cost
----------------	------	--------------

		SELECT		2067K
0	STATEMENT		2202	
	HASH			2067K
* 1	JOIN		2202	
	TABLE	EMPLO		
2	ACCESS	FULL YEEs	234K 478	
	TABLE	BY INDEX	913K	
3	ACCESS	ROWID SALES	1724	
	INDEX	SALES_D		
* 4	RANGE	SCAN ATE		133



MẸO

Chọn ít cột hơn sẽ giúp cải thiện hiệu năng của hash join.

Mặc dù có vẻ như đơn giản khi loại bỏ một vài cột trong câu lệnh SQL. Nhưng đó là một thách thức lớn khi sử dụng công cụ ánh xạ quan hệ đối tượng (Object Relational Mapping - ORM). Vì khả năng xử lý dữ liệu trên từng phần của đối tượng rất hạn chế. Ví dụ tiếp theo cho thấy một số khả năng.

JAVA

JPA định nghĩa FetchType.LAZY trong nhãn @Basic. Nó có thể áp dụng trên các lớp thuộc tính:

```

@Column(name="junk")
@Basic(fetch=FetchType.LAZY)
private String junk;

```

Chiến lược LAZY (lazily fetch): Trong lần lấy dữ liệu đầu tiên (ví dụ: nạp một thực thể) chỉ một phần dữ liệu được nạp (một số thuộc tính của thực thể), phần dữ liệu còn lại sẽ được nạp sau đó khi cần thiết bằng cách gọi phương thức tương ứng của đối tượng (các thuộc tính còn lại của thực thể).

Chiến lược EAGER (eagerly fetch): Toàn bộ dữ liệu sẽ được nạp (khi nạp thực thể thì toàn bộ thuộc tính của nó sẽ được nạp nên tiêu tốn tài nguyên hơn chiến lược LAZY).

Hibernate 3.6 thực hiện nạp dữ liệu theo kiểu Lazy trong thời gian trình điều khiển biên dịch. Trình điều khiển bổ sung code vào các lớp được biên dịch, các code này là mặc định sẽ nạp dữ liệu theo kiểu Lazy. Phương pháp này là hoàn toàn trong suốt đối với ứng dụng nhưng nó nảy sinh vấn đề “N+1 problems” (đã được định nghĩa ở phần Nester Loops Join): mất một lệnh select cho mỗi bản ghi và thuộc tính. Sẽ vô cùng nguy hiểm vì JPA không kiểm soát code lúc chạy nên nó không thể nạp thêm dữ liệu bổ sung khi cần thiết (do Lazy chỉ nạp một phần).

Ngôn ngữ HQL (Hibernate’s native query language) có thể giải quyết được vấn đề này với mệnh đề **FETCH ALL PROPERTIES**:

```
select s from Sales s FETCH ALL PROPERTIES
          s.employee e FETCH ALL
inner join fetch PROPERTIES
where s.saleDate >:dt
```

Mệnh đề **FETCH ALL PROPERTIES** bắt buộc Hibernate nạp toàn bộ dữ liệu của thực thể ngay cả khi sử dụng nhãn **LAZY**.

Một lựa chọn khác là chỉ tải duy nhất các cột đã chọn bằng cách sử dụng DTOs (data transport objects) thay cho các thực thể. Phương pháp này có cách thức hoạt động như nhau trong HQL và JPQL, bạn cần cài đặt một đối tượng trong câu truy vấn.

```
select new SalesHeadDTO(s.saleDate , s.eurValue
                         ,e.firstName, e.lastName)
from Sales s
join s.employee e
where s.saleDate > :dt
```

Câu truy vấn select trả về một đối tượng **SalesHeadDTO** – đối tượng đơn giản trong Java (POJO), đây không phải là một thực thể (thực thể được thể hiện là một bảng trong database còn đối tượng dùng để chỉ các cấu trúc dữ liệu trong bộ nhớ).

PERL

Framework DBIx::Class được dùng để ánh xạ các hàng trong cơ sở dữ liệu quan hệ thành đối tượng. Việc kế thừa từ lớp này không gây ra vấn đề aliasing (dữ liệu trong bộ nhớ có thể được truy cập bởi các tên biểu tượng – symbolic name khác nhau gây

khó khăn trong việc phân tích và tối ưu chương trình). Lớp cookbook nằm trong framework DBIx::Class hỗ trợ phương pháp này. Lược đồ sau đây định nghĩa lớp Sales trên hai cấp độ:

```
Package UseTheIndexLuke::Schema::Result::SalesHead;
use base qw/DBIx::Class::Core/;

__PACKAGE__->table('sales');
__PACKAGE__->add_columns(qw/sale_id employee_id subsidiary_id
                           sale_date eur_value/);
__PACKAGE__->set_primary_key(qw/sale_id/);

__PACKAGE__->belongs_to('employee', 'Employees',
                        {'foreign.employee_id' => 'self.employee_id'
                         , 'foreign.subsidiary_id' => 'self.subsidiary_id'});
```

```
package UseTheIndexLuke::Schema::Result::Sales;
use base qw/UseTheIndexLuke::Schema::Result::SalesHead/;
```

```
__PACKAGE__->table('sales');
__PACKAGE__->add_columns(qw/junk/);
```

Lớp Sales được phát sinh từ lớp SalesHead và bổ sung thêm các thuộc tính bị thiếu. Ta có thể sử dụng cả 2 lớp nếu cần thiết. Hãy lưu ý rằng bảng tạo ra cũng phải được sinh ra như trên.

Ta có thể lấy toàn bộ thông tin chi tiết của Employee bằng cách chọn các cột như đoạn code bên dưới:

```
my @sales =
  $schema->resultset('SalesHead')
    ->search($cond
              , { join => 'employee'
```

```

        ,'+columns' => ['employee.first_name'
                      , 'employee.last_name']
    }
);

```

Không khả thi khi chỉ tải duy nhất các cột được chọn từ bảng gốc – SalesHead trong trường hợp này.

DBIx::Class 0.008192 tạo ra mệnh đề SQL sau. Nó lấy tất cả các cột từ bảng SALES và lấy các thuộc tính từ EMPLOYEES:

```

SELECT me.sale_id,
       me.employee_id,
       me.subsidiary_id,
       me.sale_date,
       me.eur_value,
       employee.first_name,
       employee.last_name
  FROM sales me
 JOIN employees employee
    ON( employee.employee_id = me.employee_id
      AND employee.subsidiary_id = me.subsidiary_id)
 WHERE(sale_date > ?)

```

PHP

Version 2 của framework Doctrine hỗ trợ việc chọn thuộc tính trong thời điểm runtime. Tuy nhiên bạn cần phải chọn cột khóa chính một cách rõ ràng:

```

$qb = $em->createQueryBuilder();
$qb->select('partial s.{sale_id, sale_date, eur_value}', '
            . 'partial e.{employee_id, subsidiary_id, '
            . 'first_name , last_name}')
->from('Sales', 's')
->join('s.employee', 'e')
->where("s.sale_date > :dt")

```

```
->setParameter('dt', $dt, Type::DATETIME);
```

Câu lệnh SQL được tạo ra chứa các cột yêu cầu cùng SUBSIDIARY_ID và EMPLOYEE_ID từ bảng SALES.

```
SEL          A
ECT s0_sale_id    S sale_id0,
          A
          s0_sale_date   S sale_date1,
          A
          s0_eur_value   S eur_value2,
          e1_employee_i  A
          d              S employee_id3,
          e1_subsidiary_id
          AS             subsidiary_id4,
          A
          e1_first_name  S first_name5,
          A
          e1_last_name   S last_name6,
          s0_subsidiary_id
          AS             subsidiary_id7,
          s0_employee_id
          AS             employee_id8

FROM sales s0_
INNER      JOIN
employees e1_
```

```
ON      s0_.subsidiary_id      =
e1_.subsidiary_id
AND      =
s0_.employee_id    e1_.employee_id
WHE s0_.sale_date >
RE ?
```

Các đối tượng trả về tương tự như các đối tượng được nạp đầy đủ, mặc dù một số cột bị thiếu sẽ không được khởi tạo nhưng việc truy cập chúng không gây ra ngoại lệ.



CHÚ
Ý:

MySQL không hỗ trợ hash join.

39. SORT MERGE JOIN

Kết nối sắp xếp trộn (sort-merge join) hợp nhất 2 danh sách đã sắp xếp. Cả 2 bảng của phép kết nối đều phải được sắp xếp.

Sort-merge join có thể đánh index giống như hash join, đó là index cho vị trí độc lập để tăng hiệu quả trong việc chọn các bản ghi thỏa mãn điều kiện trong mệnh đề where. Việc đánh chỉ mục cho các vị trí nối là không cần thiết. Mọi thứ dường như giống với hash join. Tuy nhiên có một điểm cần lưu ý đối với sort-merge join đó là: đối xứng tuyệt đối (absolute symmetry). Thứ tự kết nối không tạo ra bất kỳ khác biệt nào – ngay cả đối với hiệu năng. Tính chất này rất hữu dụng cho các phép outer joins. Đối với các phương pháp khác vị trí của outer join (trái hoặc phải) sẽ ảnh hưởng đến thứ tự kết nối – nhưng với sort-merge join thì không. Sort-merge join thậm chí có thể kết nối ngoài trái và phải tại cùng một thời điểm nên được gọi là kết nối ngoài đầy đủ (full outer join).

Mặc dù hiệu quả của sort-merge join rất tốt khi đầu vào đã được sắp xếp, nhưng nó trở nên khó khăn vì việc sắp xếp cả 2 bảng trước khi kết nối tốn rất nhiều chi phí. Kết nối bảng băm mặt khác chỉ cần tiền xử lý duy nhất một bên.

Sort-merge join thực sự mạnh mẽ khi đầu vào của nó đã được sắp xếp. Điều này có thể thực hiện trên các thuộc tính đã có chỉ mục nhóm cụm (clustering index). Chương 6, “Sắp xếp và phân nhóm”, giải thích chi tiết về khái niệm này. Thực tế, Hash join được sử dụng phổ biến hơn cả.

40. Gom cụm dữ liệu: Sức mạnh thứ hai của chỉ mục

Thuật ngữ *cluster* (cụm) được sử dụng trong nhiều lĩnh vực khác nhau. Ví dụ, một cụm sao là một nhóm các ngôi sao. Một cụm máy tính là một nhóm các máy tính làm việc chật chẽ với nhau, để giải quyết một vấn đề phức tạp (cụm máy tính hiệu năng cao), hoặc để tăng tính sẵn sàng cho hệ thống (cụm chuyển đổi dự phòng). Nói chung, thuật ngữ *cluster* là chỉ các cụm, các nhóm chứa những thứ liên quan xuất hiện cùng với nhau.

Trong lĩnh vực máy tính có thêm một loại cụm khác, mà khi nhắc tới thường hay bị hiểu nhầm là cụm dữ liệu. Clustering data có nghĩa là lưu trữ những dữ liệu truy cập mà có liên kết chặt chẽ với nhau một cách liên tiếp, để khi truy cập chúng đòi hỏi ít các hoạt động xử lí vào ra hơn. Các cụm dữ liệu là rất quan trọng trong các thuật ngữ của tối ưu và cải thiện cơ sở dữ liệu. Mặt khác, các cụm máy tính cũng rất phổ biến trong lĩnh vực cơ sở dữ liệu, do đó, thuật ngữ *cụm* là một cụm từ rất mơ hồ. Ví dụ, khi nói: "Hãy sử dụng một cụm để cải thiện hiệu suất cơ sở dữ liệu", nó có thể đề cập đến một cụm máy tính nhưng cũng có thể có nghĩa là một cụm dữ liệu. Trong chương này, cụm thường đề cập đến các cụm dữ liệu.

Cụm dữ liệu đơn giản nhất trong một cơ sở dữ liệu quan hệ (SQL database) là hàng. Nếu có thể, cơ sở dữ liệu sẽ lưu trữ tất cả các cột của một hàng trong cùng một khối cơ sở dữ liệu. Ngoại lệ xảy ra khi một hàng không chỉ thuộc một khối duy nhất, ví dụ: các kiểu dữ liệuLOB (Large Object byte).

Lưu trữ theo cột

Cơ sở dữ liệu hướng cột (column oriented databases), lưu trữ theo cột, hay nói cách khác là tổ chức các bảng theo cách phân cột. Mô hình này có lợi khi truy cập nhiều hàng nhưng chỉ một vài cột - một mô hình rất phổ biến trong kho dữ liệu (OLAP).

Chỉ mục cho phép xác định một cụm dữ liệu. Cơ sở cho điều này đã được giải thích trong Chương 1: “Anatomy of an Index”, các nút lá chỉ mục lưu trữ các phần tử của cột được đánh chỉ mục theo một cách thức đánh thứ tự để các giá trị tương tự được lưu trữ cạnh nhau. Điều đó có nghĩa là các chỉ mục xây dựng các cụm hàng có giá trị tương tự nhau. Khả năng phân cụm dữ liệu là rất quan trọng, điều mà tôi xem nó như là sức mạnh thứ hai của việc lập chỉ mục.

Các phần tiếp theo sẽ giải thích cách sử dụng chỉ mục để phân cụm dữ liệu và cải thiện hiệu suất của câu truy vấn.

41. Bộ lọc chỉ mục được sử dụng một cách có chủ đích

Thường thì các vị từ lọc cho thấy việc sử dụng chỉ mục không đúng gây ra bởi một trật tự cột không chính xác trong một chỉ mục kết hợp (concatenated index). Tuy nhiên, các vị từ lọc cũng có thể được sử dụng vì một lý do tốt - không phải để cải thiện hiệu năng quét dài (range scan) nhưng để nhóm dữ liệu truy cập liên tiếp với nhau.

Các vị từ của mệnh đề **where** không thể dùng làm vị từ truy cập là ứng viên tốt cho kỹ thuật này:

```
SELECT first_name, last_name, subsidiary_id, phone_number
  FROM employees
 WHERE subsidiary_id = ?
   AND UPPER(last_name) LIKE '%INA%';
```

Hãy nhớ rằng các biểu thức LIKE với ký tự đại diện ở đầu (%) không thể sử dụng cây chỉ mục. Điều đó có nghĩa là lập chỉ mục LAST_NAME không thu hẹp được phạm vi chỉ mục được quét - dù bạn lập chỉ mục LAST_NAME hay UPPER(last_name). Điều kiện này không phải điều kiện tốt để lập chỉ mục. Tuy nhiên, điều kiện về SUBSIDIARY_ID rất phù hợp cho lập chỉ mục. Chúng ta thậm chí không cần phải thêm một chỉ mục mới bởi vì SUBSIDIARY_ID đã là cột đầu trong chỉ mục cho khóa chính.

Id Operation	Name	Rows	Cost
0 SELECT STATEMENT		17	230
*1 TABLE ACCESS BY INDEX ROWID	EMPLOYEES	17	230
*2 INDEX RANGE SCAN	EMPLOYEE_PK	333	2

Predicate Information (identified by operation id):

```
1 - filter(UPPER("LAST_NAME") LIKE '%INA%')
2 - access("SUBSIDIARY_ID"=TO_NUMBER(:A))
```

Trong kế hoạch thực hiện ở trên, giá trị **Cost** tăng lên gấp hàng trăm lần từ hoạt động INDEX RANGE SCAN đến TABLE BY INDEX ROWID. Nói cách khác, truy cập bảng làm việc nhiều nhất. Nó thực sự là một hiện tượng phổ biến và không phải là một vấn đề của chính nó. Tuy nhiên, truy cập bảng là đóng góp quan trọng nhất cho thời gian thực hiện tổng thể của truy vấn này.

Truy cập bảng không nhất thiết là nút cổ chai nếu các hàng được truy cập và lưu trữ gom nhóm theo cụm vì cơ sở dữ liệu có thể lấy tất cả các hàng với một thao tác đọc tuần tự (giảm thiểu thời gian di chuyển đầu đọc). Trái lại, nếu các hàng tương tự nhau được trải trên nhiều khối, block dữ liệu trên đĩa cứng rải rác không tuần tự, việc truy

cập bảng có thể trở thành một vấn đề hiệu suất nghiêm trọng vì cơ sở dữ liệu phải lấy nhiều khối ở các vị trí khác nhau trên thiết bị lưu trữ để lấy tất cả các hàng. Điều đó có nghĩa là hiệu suất phụ thuộc vào sự phân bố vật lý của các hàng được truy cập - nói cách khác: nó phụ thuộc vào sự phân cụm hàng.

Chú ý:

Mỗi tương quan về trật tự chỉ mục và thứ tự bảng là một chuẩn hiệu năng - cái gọi là nhân tố phân cụm chỉ mục.

Trên thực tế, có thể cải thiện hiệu suất truy vấn bằng cách sắp xếp lại các hàng trong bảng sao cho chúng tương thích với thứ tự chỉ mục. Tuy nhiên, phương pháp này rất hiếm khi áp dụng vì bạn chỉ có thể lưu các hàng của bảng theo một thứ tự nhất định duy nhất. Điều đó có nghĩa là bạn có thể tối ưu hóa trật tự lưu trữ các hàng theo một chỉ mục gom cụm mà thôi. Thậm chí nếu bạn có thể chọn một chỉ mục gom cụm mà bạn muốn tối ưu hóa bảng, nó vẫn là một nhiệm vụ khó khăn vì hầu hết các cơ sở dữ liệu chỉ cung cấp công cụ thô sơ cho nhiệm vụ này. Sau tất cả cái gọi là sắp xếp hàng là một cách tiếp cận không thực tế.

Hệ số phân cụm chỉ mục

Hệ số phân cụm chỉ mục là một cách đo lường gián tiếp của xác suất rằng hai phần tử của chỉ mục kế tiếp nhau tham chiếu đến cùng một khối lưu trữ. Trình tối ưu hóa tính xác suất này khi tính giá trị chi phí của hoạt động TABLE ACCESS BY INDEX ROWID.

Đây chính xác là sức mạnh thứ hai của việc lập chỉ mục – gom cụm dữ liệu. Bạn có thể chọn nhiều cột vào một chỉ mục gom cụm mục sao cho chúng được lưu trữ tự động theo thứ tự được xác định rõ ràng. Điều đó làm cho chỉ mục là một công cụ mạnh mẽ nhưng đơn giản để gom cụm dữ liệu.

Để áp dụng khái niệm này cho truy vấn ở trên, chúng ta phải mở rộng chỉ mục để bao gồm tất cả các cột từ mệnh đề where - ngay cả khi chúng không thu hẹp phạm vi chỉ mục được quét.

```
CREATE INDEX empsubupnam ON employees  
    (subsidiary_id, UPPER(last_name));
```

Cột SUBSIDIARY_ID là cột chỉ mục đầu tiên để nó có thể được sử dụng làm vị từ truy cập. Các thẻ hiện UPPER(last_name) bao phủ bộ lọc LIKE như bộ lọc vị từ chỉ mục. Đánh chỉ mục biểu diễn chữ hoa sẽ tiết kiệm một vài chu kì CPU trong quá trình thực hiện, nhưng một chỉ mục thẳng trên LAST_NAME cũng sẽ hoạt động tốt. Bạn sẽ tìm hiểu thêm điều này trong phần tiếp theo.

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		17	20
1	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	17	20
*2	INDEX RANGE SCAN	EMPSUBUPNAM	17	3

Predicate Information (identified by operation id):

```
2 - access("SUBSIDIARY_ID"=TO_NUMBER(:A))
      filter(UPPER("LAST_NAME") LIKE '%INA%')
```

Kế hoạch thực hiện mới cho thấy các hoạt động tương tự như trước. Giá trị chi phí giảm đáng kể. Trong thông tin predicate chúng ta có thể thấy rằng bộ lọc LIKE đã được áp dụng trong INDEX RANGE SCAN. Các hàng không hoàn thành bộ lọc LIKE sẽ bị loại bỏ ngay lập tức. Truy cập bảng không có bất kỳ vị từ lọc nào nữa. Điều đó có nghĩa là nó không tải về các hàng không hoàn thành mệnh đề where.

Sự khác biệt giữa hai kế hoạch thực hiện được hiển thị rõ ràng trong cột **Rows**. Theo ước tính của trình tối ưu hóa, truy vấn cuối cùng sẽ khớp với 17 bản ghi. Chỉ số quét trong kế hoạch thực hiện lần đầu tiên cung cấp 333 hàng. Cơ sở dữ liệu phải tải 333 hàng từ bảng để áp dụng bộ lọc LIKE làm giảm kết quả xuống còn 17 hàng. Trong kế hoạch thực hiện thứ hai, truy cập chỉ mục không cung cấp những hàng đó ở vị trí đầu tiên nên cơ sở dữ liệu chỉ cần thực hiện TABLE ACCESS BY INDEX ROWID thao tác này 17 lần.

Bạn cũng nên lưu ý rằng giá trị chi phí của hoạt động INDEX RANGE SCAN tăng từ hai đến ba vì cột bổ sung làm cho chỉ số lớn hơn. Theo quan điểm của việc đạt được hiệu suất, nó là một thỏa hiệp chấp nhận được.

Cảnh báo:

Không sử dụng một chỉ mục mới chỉ vì mục đích duy nhất là làm các vị từ lọc. Thay vào đó, mở rộng một chỉ mục đang tồn tại và duy trì nỗ lực bảo trì ở mức thấp. Với một số cơ sở dữ liệu bạn thậm chí còn có thể thêm các cột vào chỉ mục cho khóa chính, mà cột đó không phải một phần của khóa chính.

Kích thước chỉ mục tăng lên cùng với số lượng cột - đặc biệt là khi thêm các cột văn bản. Tuy nhiên hiệu suất không thể tốt hơn cho một chỉ mục lớn hơn mặc dù khả năng mở rộng logarithmic hạn chế tác động không đáng kể. Bạn không nên thêm tất cả các cột tồn tại trong mệnh đề where vào một chỉ mục mà thay vào đó chỉ vị từ lọc trong chỉ mục một cách có chủ đích để giảm khối lượng dữ liệu trong một bước thực hiện trước đó.

42. Chỉ mục bao phủ truy vấn, Index-only scan (index covers query)

Index-only scan: chỉ quét trên chỉ mục (không phải truy cập trực tiếp vào bảng).

Index-only scan là 1 trong những phương pháp tinh chỉnh hiệu quả nhất. Nó không chỉ tránh việc truy cập bảng để đánh giá mệnh đề "where", mà còn tránh truy cập bảng 1 cách hoàn toàn nếu CSDL có thể trả lời được truy vấn chỉ dựa vào các cột đã được đánh chỉ mục.

Để bao phủ toàn bộ một câu truy vấn, 1 chỉ mục phải chứa tất cả các cột từ câu lệnh SQL - đặc biệt những cột từ mệnh đề "select", biểu diễn theo ví dụ sau:

```
CREATE INDEX sales_sub_eur
  ON sales
  ( subsidiary_id, eur_value );

SELECT SUM(eur_value)
  FROM sales
 WHERE subsidiary_id = ?;
```

Tất nhiên việc lập chỉ mục cho mệnh đề "where" được ưu tiên so với các loại mệnh đề khác. Do đó, cột SUBSIDIARY_ID ở ưu tiên hàng đầu vì vậy nó được coi là 1 vị từ truy cập. (***)

Kế hoạch thực thi cho thấy quét chỉ mục không cần truy cập bảng tiếp theo (bảng được truy cập bởi chỉ số dòng).

Id Operation	Name	Rows Cost
0 SELECT STATEMENT		1 104
1 SORT AGGREGATE		1 104
* 2 INDEX RANGE SCAN	SALES_SUB_EUR	40388 104

Predicate Information (identified by operation id):

```
2 - access("SUBSIDIARY_ID"=TO_NUMBER(:A))
```

Việc đánh chỉ mục bao hàm toàn bộ câu truy vấn bởi vậy nó còn được gọi là "converging index".

Chú ý:

Nếu một chỉ mục ngăn chặn việc truy cập bảng, nó cũng được gọi là 1 "converging index".

Tuy nhiên, thuật ngữ này dễ gây hiểu nhầm bởi vì nó nghe giống với một thuộc tính của chỉ mục. Cụm từ index-only scan có thể diễn đạt một cách chính xác rằng nó là một phép toán trên sơ đồ thực thi.

Chỉ mục có 1 bản sao chép của cột EUR_VALUE vì vậy cơ sở dữ liệu có thể sử dụng giá trị được lưu trữ trong chỉ mục. Việc truy nhập bảng là không cần thiết bởi vì chỉ mục có tất cả thông tin để đáp ứng câu truy vấn.

Một "index-only scan" có thể cải thiện hiệu năng một cách đáng kinh ngạc. Chỉ cần nhìn vào ước tính số lượng hàng trong kế hoạch thực thi: trình tối ưu hóa dự kiến sẽ tổng hợp hơn 40000 hàng. Nghĩa là việc chỉ quét chỉ mục ngăn việc 40000 bảng tìm nạp - nếu mỗi dòng nằm trong các khối bảng khác nhau (đọc khối lưu trữ khác nhau thì cần di chuyển đầu đọc nhiều lần). Nếu chỉ mục có hệ số phân cụm hợp lý, có nghĩa là, nếu các hàng tương ứng được gom lại trên tổ chức lưu trữ vật lý thì lợi ích có thể giảm đi 1 cách đáng kể.

Ngoài hệ số phân cụm, số lượng các hàng được chọn hạn chế việc tăng hiệu năng tiềm năng đạt được. Ví dụ, nếu bạn chọn 1 hàng, bạn chỉ có thể lưu một truy cập bảng duy nhất. Xem xét cách duyệt cây cần nạp chỉ 1 vài khối dữ liệu, truy cập bảng lưu trữ có thể trở nên không đáng kể.

Quan trọng:

Lợi thế về hiệu năng của một "index-only scan" phụ thuộc vào số lượng hàng truy cập và hệ số phân cụm dữ liệu.

"Index-only scan" là một chiến lược đánh chỉ mục tích cực. Không tạo 1 chỉ mục cho "index-only scan" chỉ vì nghi ngờ bởi vì nó tăng sử dụng bộ nhớ và làm tăng chi phí bảo trì cho câu lệnh "UPDATE". Xem chương 8, "Modifying Data". Trong thực tế, bạn nên đánh chỉ mục mà không cần xem xét các cột trong mệnh đề "select" và chỉ khi thật cần thiết mới mở rộng nó.

"Index-only scan" còn tồn tại khuyết điểm. Trong trường hợp nếu chúng ta mở rộng thêm câu truy vấn bằng việc giới hạn để tìm những hàng hóa bán gần đây:

```
SELECT SUM(eur_value)
  FROM sales
 WHERE subsidiary_id = ?
   AND sale_date > ?;
```

Bằng trực quan, ta có thể cho rằng câu truy vấn thực hiện nhanh hơn vì chọn ít dòng hơn. Tuy nhiên không phải như vậy, lúc này chúng ta đang sử dụng mệnh đề "where" với một cột không được đánh chỉ mục, vì vậy phải truy cập bảng và tái cột đó về.

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT			1 371
1	SORT AGGREGATE		1	
*2	TABLE ACCESS BY INDEX ROWID	SALES	2019	371
*3	INDEX RANGE SCAN	SALES_DATE	10541	30

Predicate Information (identified by operation id):

```
2 - filter("SUBSIDIARY_ID"=TO_NUMBER(:A))
3 - access("SALE_DATE">>:B)
```

Truy cập bảng làm tăng thời gian phản hồi mặc dù câu truy vấn chọn ít hàng hơn. Vẫn đề không phải là số hàng câu truy vấn phải trả về mà là số hàng trong cơ sở dữ liệu phải duyệt để tìm ra chúng.

Cảnh báo:

Mở rộng mệnh đề "where" có thể là gây ra bất lợi về hiệu năng. Vì vậy chúng ta nên kiểm tra kế hoạch thực thi trước khi mở rộng câu truy vấn.

Nếu “index-only scan” không còn được sử dụng, trình tối ưu hóa sẽ chọn kế hoạch thực thi kế tiếp tốt nhất. Có nghĩa là trình tối ưu chọn một kế hoạch thực thi hoàn toàn khác hoặc, như trên, một kế hoạch thực thi tương tự với một chỉ mục khác. Trong trường hợp này, nó sử dụng chỉ mục trên trường SALE_DATE.

Tùy cách lựa chọn kế hoạch thực thi của trình tối ưu, chỉ mục này có hai lợi thế so với chỉ mục SALE_SUB_EUR. Lợi thế thứ nhất, trình tối ưu chỉ ra rằng tính chọn lọc trên SALE_DATE tốt hơn trên trường SUBSIDIARY_ID. Bạn có thể thấy rằng trong cột Rows hàng cuối cùng tương ứng của hai kế hoạch thực hiện (khoảng 10000 hàng ứng với SALE_DATE so với 40000 hàng ứng với SUBSIDIARY_ID).

Tuy nhiên sự ước lượng này chỉ mang tính cục bộ vì với mỗi câu truy vấn lại gắn với những tham số khác nhau. Điều kiện trên trường SALE_DATE có thể khiến phải chọn toàn bộ bảng khi điều kiện là chọn ngày bán hàng đầu tiên.

Lợi thế thứ hai của chỉ mục SALES_DATE là có một hệ số phân cụm tốt hơn. Điều này có căn cứ bởi bảng SALES chỉ phát triển theo thứ tự thời gian. Những hàng mới luôn được thêm vào cuối bảng miễn là không có những hàng bị xóa đi. Thứ tự trong bảng tương ứng với thứ tự của chỉ mục vì cả hai đều được sắp xếp theo thứ tự thời gian - chỉ mục này có 1 hệ số phân cụm tốt.

Khi sử dụng một chỉ mục với một hệ số phân cụm tốt, các hàng được chọn sẽ được lưu trữ gần nhau trong cơ sở dữ liệu, chỉ cần đọc tuần tự sẽ lấy được toàn bộ các hàng. Sử dụng chỉ mục này, câu truy vấn có thể đủ nhanh mà không cần “index-only scan”.

Trong trường hợp này chúng ta nên loại bỏ những cột không cần thiết mà đã đánh chỉ mục trước đó.

Chú ý:

Những chỉ mục có hệ số phân cụm tốt cho thấy lợi thế về hiệu năng của "index-only scan" là nhỏ nhất.

Trong ví dụ cụ thể này, có một sự trùng hợp ngẫu nhiên. Bộ lọc mới trên SALE_DATE không chỉ ngăn việc "index-only scan" mà còn mở ra một cách thực thi truy vấn mới. Do đó, trình tối ưu hóa có thể hạn chế tác động tới hiệu năng của sự thay đổi này. Tuy nhiên, cũng có thể ngăn "index-only scan" bằng cách select thêm một vài cột khác. Tuy nhiên, thêm 1 cột trong mệnh đề "select" có thể không bao giờ mở ra một cách truy vấn mới, điều đó có thể hạn chế ảnh hưởng của việc mất đi "index-only scan".

Gợi ý:

Duy trì "index-only scan" của bạn.

Thêm các chú thích để nhắc bạn về "index-only scan" và tham chiếu tới trang đó để mọi người có thể đọc về nó.

Các chỉ mục dựa trên hàm cũng có thể gây ra những sự khó chịu khi kết nối với "index-only scan". Một chỉ mục trên UPPER(last_name) không thể được sử dụng cho "index-only scan" khi select cột LAST_NAME. Trong phần trước, đáng lẽ chúng ta đã phải lập chỉ mục cho cột LAST_NAME để hỗ trợ bộ lọc LIKE và cho phép nó được sử dụng cho "index-only scan" khi chọn cột LAST_NAME.

Gợi ý:

Luôn luôn tập trung vào lập chỉ mục trên dữ liệu ban đầu vì đó thường là thông tin hữu ích nhất mà bạn có thể đưa vào một chỉ mục.

Tránh lập chỉ mục dựa trên hàm cho các biểu thức không thể được sử dụng như vị từ truy cập (VD: với hàm UPPPER ở trên).

Các truy vấn kết tập giống như trình bày là các ví dụ tốt mà có thể sử dụng index-only scan. Chúng truy vấn nhiều hàng nhưng chỉ có một vài cột, tạo một chỉ mục bé cũng đủ để hỗ trợ "index-only scan". Bạn truy vấn càng nhiều cột thì càng nhiều cột bạn phải thêm vào chỉ mục để có chỉ mục phù hợp. Với tư cách là nhà phát triển, bạn nên chỉ chọn các cột mà bạn thực sự cần.

Gợi ý:

Tránh dùng "SELECT * " và chỉ nên lấy các cột mà bạn cần.

Việc lập chỉ mục cho nhiều hàng cần rất nhiều không gian lưu trữ, bạn có thể đạt đến các giới hạn của cơ sở dữ liệu của bạn. Hầu hết các cơ sở dữ liệu áp đặt các giới hạn khá cứng nhắc về số cột trên một chỉ mục và tổng kích thước của một chỉ mục. Điều đó có nghĩa là bạn không thể lập chỉ mục một số cột tùy ý cũng như các cột dài tùy tiện. Tổng quan sau đây liệt kê những hạn chế quan trọng nhất. Tuy nhiên có những chỉ mục mà bao hàm toàn bộ một bảng như chúng ta thấy trong phần tới.

Think about it

Truy vấn không chọn bất kỳ cột nào trong bảng thường được thực hiện bằng quét chỉ mục. Hãy lấy một ví dụ có ý nghĩa?

MySQL

MySQL 5.6 với InnoDB giới hạn mỗi cột đơn là 767 byte và tất cả các cột với nhau là 3072 byte. Các chỉ mục MyISAM được giới hạn trong 16 cột và chiều dài khóa lớn nhất là 1000 byte.

MySQL có một tính năng duy nhất được gọi là "prefix indexing" (đôi khi còn được gọi là "partial indexing" (chỉ mục một phần)). Điều này có nghĩa là lập chỉ mục các ký tự đầu tiên của một cột - vì vậy nó không liên quan gì đến các chỉ mục phần được mô tả trong Chương 2. Nếu bạn lập chỉ mục một cột vượt quá chiều dài cột cho phép (767 byte cho InnoDB), MySQL sẽ tự động cắt cột theo. Đây là lý do khiến việc lập chỉ mục đã thành công với cảnh báo "Khóa đã xác định quá dài; độ dài khóa tối đa là 767 byte "nếu bạn vượt quá giới hạn. Điều đó có nghĩa là chỉ mục không chứa một bản sao đầy đủ của cột nữa và do đó chỉ sử dụng hạn chế cho một index-only scan (tương tự như một chỉ mục dựa trên hàm). Bạn có thể sử dụng prefix indexing của MySQL một cách rõ ràng để tránh vượt quá giới hạn độ dài tổng số khóa nếu bạn nhận được thông báo lỗi "Khóa đã xác định quá dài; chiều dài khóa tối đa là [1000/3072] byte". Ví dụ sau chỉ lập chỉ mục 10 ký tự đầu tiên của cột LAST_NAME:

```
CREATE INDEX ... ON employees (last_name(10));
```

Oracle Database

Chiều dài khóa chỉ mục tối đa phụ thuộc vào kích thước khối và các thông số lưu trữ (thông thường tối đa 75% kích thước khối (8KB) trừ một số chi phí). Chỉ mục B-tree được giới hạn trong 32 cột. Khi sử dụng Oracle 11g với tất cả mặc định tại chỗ (8k blocks), chiều dài khóa chỉ mục tối đa là 6398 byte. Vượt quá giới hạn này gây ra thông báo lỗi "ORA-01450: đã vượt quá độ dài khóa (6398)".

PostgreSQL

Cơ sở dữ liệu PostgreSQL hỗ trợ index-only scans chỉ từ phiên bản 9.2 trở lên.

Chiều dài khóa của chỉ mục "B-tree" được giới hạn trong 2713 byte (mã hóa có sẵn, khoảng BLCKSZ / 3). Thông báo lỗi tương ứng "chỉ số kích thước hàng ... vượt quá

btree tối đa, 2713" xuất hiện chỉ khi thực hiện "insert" hoặc "update" vượt quá giới hạn. Chỉ mục "B-tree" có thể chứa tối đa 32 cột.

SQL Server

SQL Server hạn chế chiều dài khóa tối 900 byte và 16 cột khóa. Tuy nhiên, SQL Server có một tính năng cho phép bạn thêm các cột dài tùy ý vào một chỉ mục với mục đích duy nhất là hỗ trợ index-only scan. Do đó, SQL Server phân biệt giữa các cột khóa và các cột không phải khóa.

Các cột khóa là các cột được lập chỉ mục như chúng đã được thảo luận cho đến thời điểm này. Các cột không phải khóa là các cột bổ sung chỉ được lưu trữ trong các nút lá chỉ mục. Các cột không phải khóa có thể kéo dài một cách tùy ý nhưng không thể được sử dụng như các vị từ truy cập (vị từ để tìm kiếm). Các cột không phải khóa được định nghĩa bằng từ khóa include của lệnh create index:

```
CREATE INDEX empsubupnam
  ON employees
    (subsidiary_id, last_name)
INCLUDE(phone_number, first_name);
```

43. Index-Organized Tables

Việc chỉ quét trên chỉ mục (Index-only scan) thực hiện một câu lệnh SQL chỉ sử dụng những dữ liệu được lưu trữ ở trong chỉ mục, những dữ liệu ban đầu được lưu giữ ở trong bảng heap không được yêu cầu để sử dụng. Nếu chúng ta đưa khái niệm đó lên cấp độ cao hơn và lưu tất cả giá trị của các cột vào trong chỉ mục, rất có thể bạn sẽ muốn biết rằng tại sao chúng ta lại cần tới bảng heap.

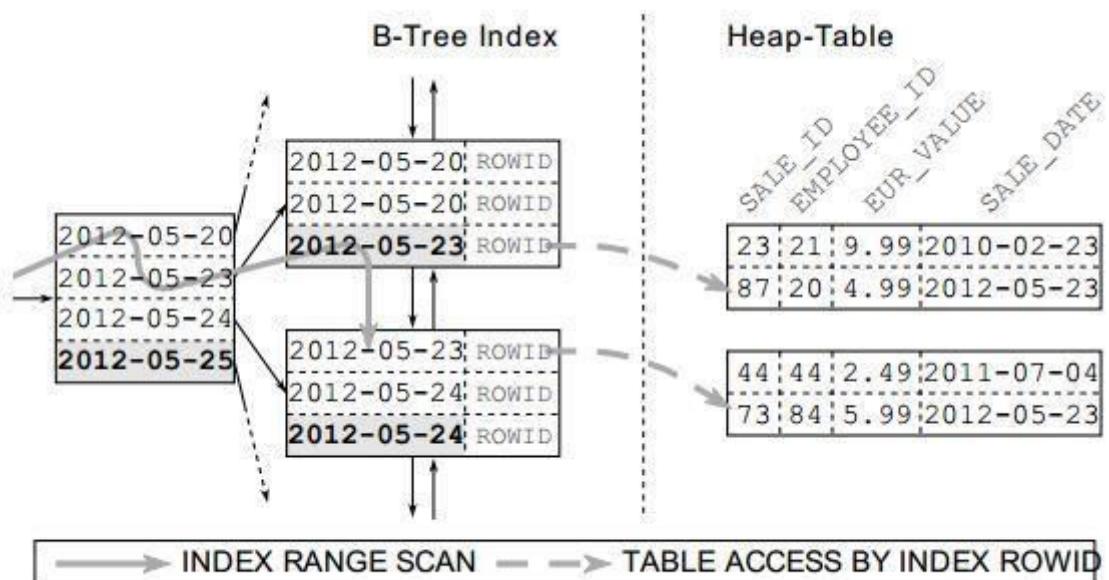
Một số cơ sở dữ liệu có thể sử dụng một chỉ mục như một bảng lưu trữ chính. Cơ sở dữ liệu Oracle gọi khái niệm này là *index-organized tables* (IOT). Các cơ sở dữ liệu khác thì sử dụng thuật ngữ *clustered index*. Trong phần này, cả hai thuật ngữ đều được sử dụng để nhấn mạnh vào bảng hoặc các đặc trưng chỉ mục khi cần thiết.

Một *index-organized table* chính là một *B-tree index* mà không có bảng heap. Có thể hiểu đơn giản rằng, đó là một loại index mà theo đó các bản ghi trong bảng được sắp thứ tự theo trường được index. Khi một bảng được tạo một index loại này thì bản thân nó sẽ trở thành một cây index, với các node lá chứa khóa là các trường được index và cũng đồng thời chứa tất cả các trường còn lại của bảng. Điều này mang lại hai lợi ích: (1) Tiết kiệm được không gian cho cấu trúc heap; (2) Mọi truy cập trên một *clustered index* chính là một index-only scan tự động. Cả hai lợi ích đều có vẻ hứa hẹn nhưng trên thực tế đều rất khó để có thể thực hiện được.

Khi một bảng đã có *clustered index* thì các index khác (non-clustered) sẽ dùng khóa của trường *clustered index* làm con trỏ để trỏ về bản ghi tương ứng (nếu bảng không có *clustered index* thì một giá trị ROWID nội bộ được dùng). Chính lúc này, những

mặt hạn chế của một index-organized table sẽ trở nên rõ ràng khi ta tạo một chỉ mục khác ở trên cùng một bảng. Tương tự với một chỉ mục thông thường, một chỉ mục – mà được gọi là chỉ mục thứ cấp (*secondary index*) để cập đến dữ liệu bảng ban đầu – cái mà được lưu trữ ở trong *clustered index*. Tại đó, dữ liệu không được lưu trữ cố định như trong bảng heap mà có thể bị di chuyển vào bất kỳ lúc nào cần thiết để duy trì thứ tự chỉ mục. Vì vậy, không thể lưu trữ địa chỉ vật lý của các hàng trong index-organized table vào trong chỉ mục thứ cấp. Cơ sở dữ liệu phải sử dụng một khóa logic (logical key) để thay thế.

Những con số sau đây cho thấy quá trình tìm kiếm chỉ mục cho việc tìm kiếm tất cả doanh số bán hàng vào ngày 23 tháng 5 năm 2012. Để so sánh, trước tiên chúng ta sẽ nhìn vào Hình 5.1, cho thấy quá trình tìm kiếm khi ta sử dụng một bảng heap. Việc thực hiện bao gồm hai bước: (1) INDEX RANGE SCAN; (2) TABLE ACCESS BY INDEX ROWID.



Hình 5.1. Truy cập dựa trên chỉ mục trên một bảng heap

Mặc dù việc truy cập bảng có thể trở thành một nút cỏ chai, nhưng nó được giới hạn cho một thao tác đọc 1 hàng, bởi vì chỉ mục có ROWID như một con trỏ trực tiếp tới hàng của bảng. Cơ sở dữ liệu có thể ngay lập tức tải hàng từ bảng heap thông qua ROWID, vì chỉ mục có vị trí chính xác của nó. Tuy nhiên, khi sử dụng một chỉ mục thứ cấp trên một index-organized table thì ta sẽ có một hình ảnh khác. Một chỉ mục thứ cấp không lưu trữ một con trỏ vật lý (ROWID) mà chỉ có các giá trị khóa của clustered index - cái mà được gọi là *clustering key* (hay khóa phân cụm). Thông thường đó là khoá chính của một index-organized table).

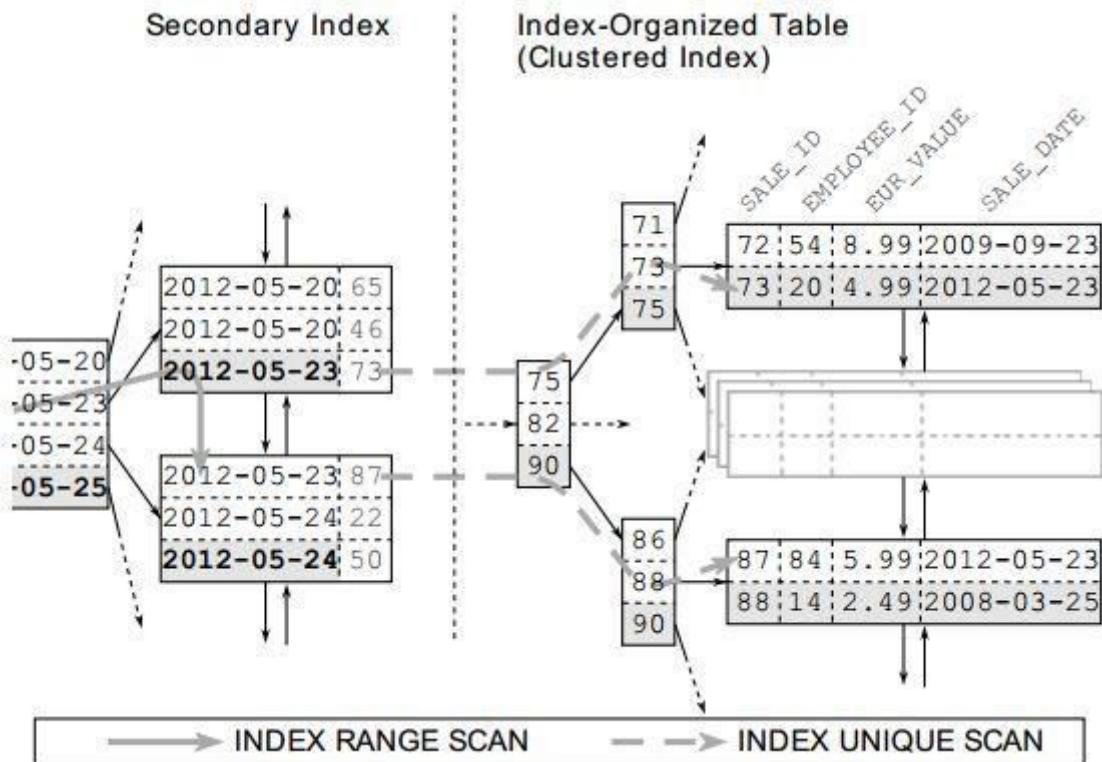
Tại sao các chỉ mục thứ cấp không có ROWID:

Một con trỏ trực tiếp tới hàng của bảng cũng sẽ là mong muốn cho một chỉ mục thứ cấp. Nhưng điều đó chỉ có thể thực hiện nếu hàng của bảng nằm ở vị trí lưu trữ cố định. Tuy nhiên điều này là không thể nếu hàng là một phần của một cấu trúc chỉ mục

và được giữ theo thứ tự. Để giữ thứ tự chỉ mục thì đôi khi ta cần phải di chuyển các hàng. Điều này cũng đúng cho các hoạt động không ảnh hưởng đến bản thân hàng này. Ví dụ, với một câu lệnh *insert*, có thể chia tách một nút lá để giành không gian cho mục mới. Điều đó có nghĩa là, một số mục được di chuyển đến một khối dữ liệu mới ở một nơi khác.

Mặt khác, một bảng heap không lưu trữ các hàng theo bất kỳ thứ tự nào. Cơ sở dữ liệu lưu các mục nhập mới ở bất cứ nơi nào nó tìm thấy đủ không gian. Sau một lần ghi thì dữ liệu sẽ không còn di chuyển trong các bảng heap.

Truy cập vào một chỉ mục thứ cấp không trả lại một ROWID nhưng thay vào đó, ta có một khóa logic để tìm kiếm trên clustered index. Tuy nhiên, nếu chỉ sử dụng một truy cập thì sẽ không đủ để tìm kiếm trên clustered index - nó đòi hỏi phải duyệt trên toàn bộ cây. Điều đó có nghĩa là việc truy cập một bảng thông qua một chỉ mục thứ cấp là tìm kiếm trên hai chỉ mục: lần 1 là tìm trên chỉ mục thứ cấp (INDEX RANGE SCAN), sau đó là tìm trên clustered index cho mỗi hàng đã tìm được trong chỉ mục thứ cấp (INDEX UNIQUE SCAN).



Hình 5.2: Chỉ mục thứ cấp trên một IOT

Hình 5.2 cho thấy rõ ràng rằng B-tree của *clustered index* nằm giữa chỉ mục thứ cấp và dữ liệu bảng.

Việc truy cập một index-organized table thông qua một chỉ mục thứ cấp là rất không hiệu quả, và điều này có thể được ngăn ngừa theo cách tương tự để tránh việc truy cập bảng trên một bảng heap: bằng cách sử dụng index-only scan, trong trường hợp này

mô tả tốt hơn là "secondary-index-only scan". Lợi thế về hiệu suất của một index-only scan thậm chí còn lớn hơn bởi vì nó không chỉ ngăn cản một lần truy cập đơn lẻ mà còn là toàn bộ INDEX UNIQUE SCAN.

Chú ý quan trọng:

Truy cập vào một index-organized table thông qua một chỉ mục thứ cấp (secondary index) là rất không hiệu quả.

Sử dụng ví dụ này, chúng ta cũng có thể thấy rằng các cơ sở dữ liệu khai thác tất cả các thông tin mà chúng có. Lưu ý rằng, một chỉ mục thứ cấp lưu trữ *clustering key* cho mỗi phần tử chỉ mục. Do đó, chúng ta có thể truy vấn *clustering key* từ một chỉ mục thứ cấp mà không cần truy cập vào index-organized table:

```
SELECT sale_id
  FROM sales_iot
 WHERE sale_date = ?;
```

Id	Operation	Name	Cost
0	SELECT STATEMENT		4
* 1	INDEX RANGE SCAN	SALES_IOT_DATE	4

Predicate Information (identified by operation id):

```
1 - access("SALE_DATE"=:DT)
```

Bảng SALES_IOT là một index-organized table sử dụng SALE_ID làm clustering key. Mặc dù chỉ mục SALE_IOT_DATE chỉ có trên cột SALE_DATE, nhưng nó vẫn có một bản sao của clustering key SALE_ID, do đó, nó có thể đáp ứng truy vấn khi chỉ sử dụng chỉ mục thứ cấp.

Khi chọn các cột khác, cơ sở dữ liệu phải chạy một INDEX UNIQUE SCAN trên clustered index cho mỗi hàng:

```

SELECT eur_value
  FROM sales_iot
 WHERE sale_date = ?;

```

Id Operation	Name	Cost
0 SELECT STATEMENT		13
* 1 INDEX UNIQUE SCAN	SALES_IOT_PK	13
* 2 INDEX RANGE SCAN	SALES_IOT_DATE	4

Predicate Information (identified by operation id):

- 1 - access("SALE_DATE"=:DT)
- 2 - access("SALE_DATE"=:DT)

Index-organized tables và *clustered indexes*, xét cho cùng, được đánh giá là không hữu ích như ngay từ cái nhìn đầu tiên. Sự cải thiện về hiệu suất trên clustered index dễ dàng bị mất tác dụng khi sử dụng thêm một chỉ mục thứ cấp. Clustering key thường dài hơn một ROWID, do đó các chỉ mục thứ cấp thường lớn, làm giảm đi ưu điểm của việc bỏ qua không lưu dữ liệu trên heap. Sức mạnh của các index-organized tables và clustered indexes hầu hết được giới hạn ở các bảng không cần chỉ mục thứ hai. Các bảng trên heap có lợi thế từ việc cung cấp một bản gốc cố định có thể được dễ dàng tham chiếu tới.

Chú ý quan trọng:

Các bảng chỉ có một chỉ mục được thực hiện tốt nhất dưới dạng các clustered indexes hay index-organized tables.

Các bảng có nhiều chỉ mục thường có thể có lợi từ các bảng trên heap. Bạn vẫn có thể sử dụng index-only scan để tránh truy cập bảng. Điều này cho phép bạn có được hiệu năng tốt như của một clustered index mà không làm chậm các chỉ mục khác.

Cơ sở dữ liệu hỗ trợ cho các index-organized tables và clustered index là rất không nhất quán. Sự khai quát dưới đây giải thích các đặc trưng quan trọng nhất:

MySQL

Công cụ MyISAM chỉ sử dụng các bảng heap trong khi công cụ InnoDB thì luôn sử dụng các clustered indexes. Điều đó có nghĩa là bạn không trực tiếp có lựa chọn.

Cơ sở dữ liệu Oracle

Cơ sở dữ liệu Oracle sử dụng các bảng heap theo mặc định. Các Index-organized tables có thể được tạo ra bằng cách sử dụng mệnh đề ORGANIZATION INDEX như sau:

```
CREATE TABLE (
    id      NUMBER NOT NULL PRIMARY KEY,
    [...]
) ORGANIZATION INDEX;
```

Cơ sở dữ liệu Oracle luôn sử dụng khóa chính làm clustering key.

PostgreSQL

PostgreSQL chỉ sử dụng bảng heap. Tuy nhiên, bạn có thể sử dụng mệnh đề CLUSTER để sắp xếp nội dung của bảng heap với một chỉ mục.

SQL Server

Theo mặc định, SQL Server sử dụng các clustered indexes (index-organized tables) bằng cách sử dụng khóa chính làm clustering key. Tuy nhiên, bạn có thể sử dụng các cột tùy ý làm clustering key, thậm chí các cột đó không phải là các *unique columns*.

Để tạo một bảng heap bạn phải sử dụng mệnh đề NONCLUSTERED trong định nghĩa khóa chính:

```
CREATE TABLE (
    id      NUMBER NOT NULL,
    [...]
    CONSTRAINT pk PRIMARY KEY NONCLUSTERED (id)
);
```

Việc xóa một clustered index sẽ biến đổi một bảng thành một bảng heap.

Hành vi mặc định của SQL Server thường gây ra các vấn đề về hiệu suất khi sử dụng các chỉ mục thứ cấp.

44. Sorting and Grouping

Dịch thiếu

45. INDEXING ORDER BY

Truy vấn SQL cùng mệnh đề order by không cần sắp xếp kết quả nếu index liên quan đã đưa ra những bản ghi đã được sắp xếp theo yêu cầu. Có nghĩa là index được sử dụng trong mệnh đề where cũng phải phủ (cover) mệnh đề order by.

Như một ví dụ, hãy xem xét truy vấn sau đây chọn các đơn hàng ngày hôm qua sắp xếp theo sale_date và product ID:

```
SELECT sale_date, product_id, quantity
  FROM sales
 WHERE sale_date = TRUNC(sysdate) - INTERVAL '1' DAY
 ORDER BY sale_date, product_id;
```

Đã có index trên trường sale_date và được sử dụng trong mệnh đề where. Ở đây index đã đưa ra các bản ghi được sắp xếp theo trường sale_date nhưng mệnh đề orderby cần phải sắp xếp theo cả trường product_id bởi vậy database vẫn phải thực hiện 1 hoạt

động sắp xếp.

Id Operation	Name	Rows	Cost
0 SELECT STATEMENT		320	18
1 SORT ORDER BY		320	18
2 TABLE ACCESS BY INDEX ROWID	SALES	320	17
*3 INDEX RANGE SCAN	SALES_DATE	320	3

Một **INDEX RANGE SCAN** đưa ra những bản ghi đã được sắp xếp chỉ theo trường sale_date. Và để tối ưu cho mệnh đề order by thì ta sẽ mở rộng index trên 2 trường sale_date và product_id, và khi đó việc truy vấn theo index đã đưa ra những bản ghi sắp xếp theo trường sale_date và product_id, khi đó database sẽ không cần 1 hoạt động sắp xếp nữa.

```
DROP INDEX sales_date;
CREATE INDEX sales_dt_pr ON sales (sale_date, product_id);
```

Id Operation	Name	Rows	Cost
0 SELECT STATEMENT		320	300
1 TABLE ACCESS BY INDEX ROWID	SALES	320	300
*2 INDEX RANGE SCAN	SALES_DT_PR	320	4

Hoạt động sắp xếp đã được loại bỏ sau khi tạo 1 index trên 2 trường sale_date và product_id mặc dù câu query vẫn có mệnh đề order by. Database đã khai thác việc index trên 2 trường sale_date và product_id đã đưa ra các bản ghi sắp thứ tự theo sale_date và product_id nên đã bỏ qua việc sắp xếp.

Quan trọng

Nếu thứ tự các trường trong index phù hợp với thứ tự các trường trong mệnh đề order by thì database sẽ bỏ qua việc sắp xếp.

Mặc dù việc tạo ra index trên 2 trường sale_date và product_id loại bỏ được việc sắp xếp nhưng tổng chi phí thực thi cho cả câu truy vấn lại tăng lên, nguyên nhân là do chỉ số nhóm cụm (clustering factor) trở nên tồi tệ sẽ được giải thích trong trang 133: "Tự động tối ưu chỉ số nhóm cụm").

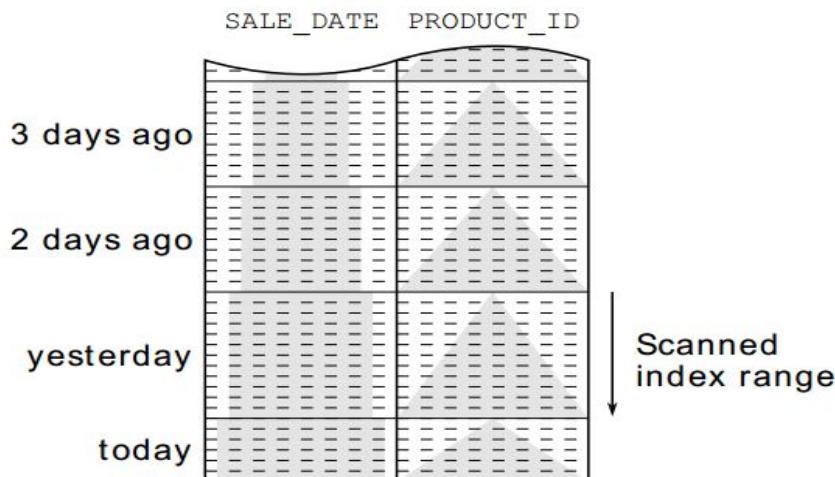
Sự tối ưu của index phù hợp với mệnh đề order by còn được thể hiện trong ví dụ sau:

```
SELECT sale_date, product_id, quantity
  FROM sales
 WHERE sale_date = TRUNC(sysdate) - INTERVAL '1' DAY
 ORDER BY product_id;
```

Ở đây thì hoạt động sắp xếp theo product_id trong mệnh đề order by cũng sẽ được database bỏ qua.

Trong hình 6.1 thì ta thấy phạm vi duyệt index chỉ là trong ngày hôm qua vì vậy index trên 2 trường sale_date và product_id sẽ đưa ra các sản phẩm đã sắp xếp theo trường product_id vì vậy database sẽ có cơ sở để bỏ qua việc sắp xếp.

Figure 6.1. Sort Order in the Relevant Index Range



Sự tối ưu khi sử dụng index để tránh việc sắp xếp cho mệnh đề order by có thể gây ra kết quả ngoài sự mong muốn khi mở rộng việc duyệt index.

```
SELECT sale_date, product_id, quantity
  FROM sales
 WHERE sale_date >= TRUNC(sysdate) - INTERVAL '1' DAY
 ORDER BY product_id;
```

Câu query trên không chỉ trả về duy nhất những mặt hàng bán ngày hôm trước. Điều đó có nghĩa là câu truy vấn lấy về những mặt hàng bán trong nhiều ngày và việc duyệt index thì không phải chỉ sắp xếp theo trường product_id mà sắp xếp theo cả 2 trường

sale_date và product_id. Bởi vậy index sẽ không được sử dụng trong trường hợp này mà database sẽ phải thực hiện 1 hoạt động sắp xếp để làm thỏa mãn mệnh đề order by.

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		320	301
1	SORT ORDER BY		320	301
2	TABLE ACCESS BY INDEX ROWID	SALES	320	300
*3	INDEX RANGE SCAN	SALES_DT_PR	320	4

Nếu cơ sở dữ liệu sử dụng 1 hoạt động sắp xếp mặc dù bạn mong muốn thực thi theo cơ chế pipelined (tức là không có hoạt động sắp xếp) thì có 2 lý do : (1) kịch bản thực thi cùng với hoạt động sắp xếp có chi phí là nhỏ nhất; (2) thứ tự các trường trong mệnh đề order by không giống với thứ tự của các trường trong index.

Để đơn giản ta sẽ định nghĩa 1 index đầy đủ, tức là 1 index gồm các trường theo thứ tự các trường trong mệnh đề order by để loại bỏ giả thiết số 2: “thứ tự các trường trong mệnh đề order by không giống thứ tự các trường trong index”.

Nếu cơ sở dữ liệu vẫn sử dụng 1 hoạt động sắp xếp thì là do kế hoạch thực thi này có chi phí nhỏ hoặc là cơ sở dữ liệu không sử dụng index cho mệnh đề order by.

Ý tưởng

Định nghĩa 1 index đầy đủ cho mệnh đề order by để tìm ra nguyên nhân của hoạt động sắp xếp.

Khi thực thi câu truy vấn cùng 1 index đầy đủ và xem xét kĩ kết quả của nó thì ta sẽ nhận thấy được sự sai lầm trong việc hiểu về index rằng thứ tự của index trong order by là không cần thiết. Sai lầm này dẫn tới cơ sở dữ liệu sẽ không thể sử dụng index để tránh việc sắp xếp.

Còn nếu trình tối ưu thực hiện hoạt động sắp xếp do chi phí của việc thực thi thì là bởi vì kịch bản thực thi cùng với hoạt động sắp xếp là tốt nhất cho việc đưa ra toàn bộ kết quả truy vấn hay trả về kết quả bản ghi cuối cùng nhanh nhất. Còn nếu truy vấn chỉ cần đưa ra một vài bản ghi đầu nhanh nhất thì trình tối ưu sẽ lựa chọn việc sử dụng indexed order by(???).

Tự động tối ưu tham số gom cụm:

Cơ sở dữ liệu của oracle sẽ cố gắng giữ tham số gom cụm là nhỏ nhất bằng việc xem xét ROWID khi lựa chọn thứ tự trong index. Vì vậy 2 phần tử index mà có chung khóa giá trị thì ROWID sẽ quyết định thứ tự cuối cùng của chúng. Vì vậy chỉ mục

được xây dựng cố gắng theo thứ tự các hàng trong bảng, bảng sẽ có hệ số phân cụm là nhỏ nhất (thứ tự index sẽ gần với thứ tự của bảng nhất) bởi vì ROWID là địa chỉ vật lý của dòng.

Khi thêm 1 cột vào 1 index, chúng ta thêm 1 tiêu chí sắp xếp trước ROWID thì database sẽ có ít lựa chọn hơn cho việc sắp xếp 2 phần tử được index theo thứ tự lưu trữ vật lý bảng, vì vậy hệ số phân cụm có thể sẽ tồi tệ hơn (tức là thứ tự index chênh lệnh nhiều so với thứ tự bảng).

Thực tế, thứ tự index có thể chênh lệch nhiều so với thứ tự bảng. Ví dụ, các sản phẩm bán trong 1 ngày sẽ được phân cụm cùng nhau ở trong bảng cũng như trong index, thậm chí thứ tự của chúng là không thật sự giống nhau ở mọi nơi. Cơ sở dữ liệu phải đọc các khối lưu trữ nhiều lần khi sử dụng index SALE_DT_PR dù những khối bảng là giống nhau lần đọc trước. Do thường xuyên eaching dữ liệu nên chi phí của việc đọc ghi cũng sẽ giảm đáng kể.

46. INDEXING ASC , DESC và NULLS FIRST/LAST:

Các cơ sở dữ liệu có thể đọc chỉ mục (index) theo 2 hướng, do đó lệnh **order by** cũng có thể làm được. Mặc dù **ASC** và **DESC** trong lệnh **order by** có thể ngăn chặn câu lệnh thực thi. Tuy nhiên, hầu hết các cơ sở dữ liệu có thể đưa ra một cách đơn giản để thay đổi thứ tự index, vì vậy index có thể dùng được trong lệnh **order by**.

Sau đây là xét các trường hợp có thể xảy ra đối với lệnh **order by**:

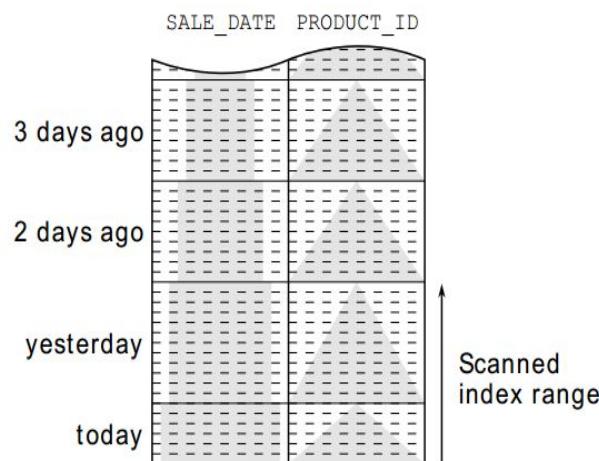
Ví dụ sau được sử dụng index để đưa ra doanh thu của ngày hôm trước được sắp xếp giảm dần theo thời gian và giảm dần theo product_id.

```
SELECT sale_date, product_id, quantity
  FROM sales
 WHERE sale_date >= TRUNC(sysdate) - INTERVAL '1' DAY
 ORDER BY sale_date DESC, product_id DESC;
```

Sử dụng execution plan đưa ra chi phí của câu truy vấn khi sử dụng index.

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		320	300
1	TABLE ACCESS BY INDEX ROWID	SALES	320	300
*2	INDEX RANGE SCAN DESCENDING	SALES_DT_PR	320	4

Trong trường hợp này, cơ sở dữ liệu sử dụng cây index để tìm đến node lá. Từ đó, nó sẽ theo chuỗi node lá đi ngược lên trên →xem hình bên dưới.

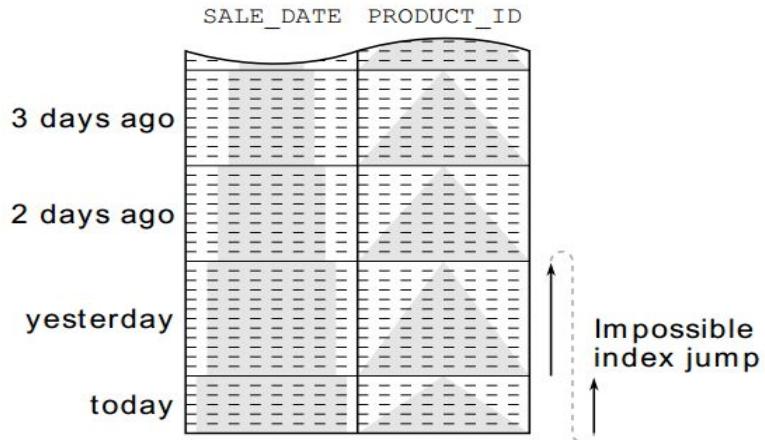


Đây là lý do mà cơ sở dữ liệu sử dụng danh sách liên kết đôi để xây dựng lên chuỗi node lá.

Ví dụ tiếp theo là sử dụng **DESC** và **ASC** trong cùng một câu lệnh **order by**:

```
SELECT sale_date, product_id, quantity
  FROM sales
 WHERE sale_date >= TRUNC(sysdate) - INTERVAL '1' DAY
 ORDER BY sale_date ASC, product_id DESC;
```

Câu truy vấn trên sẽ lấy ra doanh thu của ngày hôm qua và được sắp xếp giảm dần theo product_id, sau đó mới đến ngày hôm nay và cũng được sắp xếp giảm dần theo product_id. Tuy nhiên để lấy được doanh thu được sắp xếp theo yêu cầu, database sẽ phải “nhảy” trong quá trình quét chỉ mục (hình bên dưới).



Hình vẽ trên thể hiện rằng, sau khi sắp xếp doanh thu của ngày hôm trước ('yesterday') giảm dần theo product_id thì sẽ liên kết với danh thu của ngày hôm nay (today) giảm dần theo product_id. Tuy nhiên chỉ mục index không có liên kết từ sản phẩm có PRODUCT_ID nhỏ nhất bán ngày hôm qua tới sản phẩm có PRODUCT_ID lớn nhất bán ngày hôm nay, do đó không thể sử dụng index trong trường hợp này.

Đối với trường hợp như vậy, cơ sở dữ liệu đưa ra một phương pháp đơn giản để chỉnh sửa thứ tự index đối với thứ tự trong **order by**. cụ thể là ta sẽ sử dụng **ASC** và **DESC** trong phần khai báo chỉ mục:

```

DROP INDEX sales_dt_pr;

CREATE INDEX sales_dt_pr
    ON sales (sale_date ASC, product_id DESC);

```

Chú ý: Đối với MySql, cơ sở dữ liệu sẽ bỏ qua **ASC** và **DESC** đối với việc định nghĩa chỉ mục

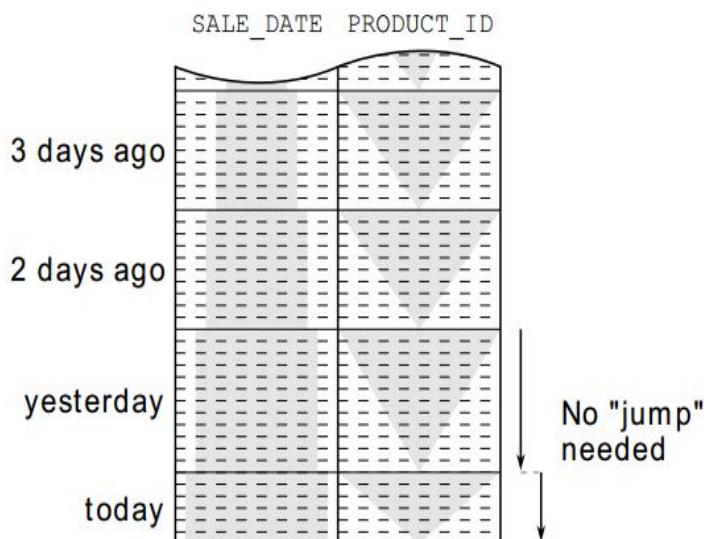
Bây giờ thứ tự chỉ mục tương ứng với thứ tự cần sắp xếp trong mệnh đề **order by** vì vậy cơ sở dữ liệu có thể bỏ qua hoạt động sắp xếp:

Id Operation	Name	Rows	Cost
0 SELECT STATEMENT			
1 TABLE ACCESS BY INDEX ROWID SALES	320 301		
*2 INDEX RANGE SCAN	SALES_DT_PR 320 4		

Hình 6.4 thể hiện một cấu trúc index mới, có thay đổi sự sắp xếp của cột thứ 2 (product_id) khi thực hiện câu lệnh khai báo chỉ mục trên:

Quan trọng: Khi sử dụng hỗn hợp **ASC** và **DESC** trong mệnh đề **order by** bạn phải tạo các chỉ mục tương tự như vậy để có thể thử dụng (pipelined) **order by**. Điều này không làm ảnh hưởng tới khả năng sử dụng của chỉ mục trong mệnh đề **where**

Figure 6.4. Mixed-Order Index



Việc **ASC/DESC** cho chỉ mục chỉ cần thiết cho việc phân loại các cột riêng lẻ trong các hướng ngược nhau. Không cần thiết phải đảo ngược thứ tự của tất cả các cột vì cơ sở dữ liệu vẫn có thể đọc các chỉ mục theo thứ tự dăm dặm nếu cần thiết. Các chỉ mục thứ cấp trên bảng được tổ chức theo chỉ mục (Index organized table) là ngoại lệ duy nhất. Các chỉ mục thứ cấp ngầm bổ sung khóa, thuộc tính được gom cụm vào chỉ mục mà không cung cấp khả năng chỉ ra thứ tự sắp xếp. Nếu bạn cần sắp xếp khóa gom cụm theo thứ tự dăm dặm, bạn buộc phải sắp xếp tất cả các cột theo thứ tự dăm dặm. Sau đó cơ sở dữ liệu mới có thể đọc chỉ mục theo chiều ngược lại để có được thứ tự mong muốn.

Bên cạnh **ASC** và **DESC**, SQL chuẩn còn định nghĩa hai sửa đổi ít được sử dụng trong mệnh đề **order by**: **NULLS FIRST** và **NULLS LAST**. Việc điều khiển ràng buộc mạnh hơn với sắp xếp có NULL mới được giới thiệu như một tùy chọn mở rộng trong SQL 2003. Do đó, tính năng hỗ trợ là không phổ biến trong các hệ quản trị cơ sở dữ liệu. Điều này rất đáng lo ngại vì không có tiêu chuẩn xác định rõ ràng thứ tự sắp xếp

của **NULL**. Nó chỉ cho biết là tất cả các **NULL** phải xuất hiện cùng nhau sau khi sắp xếp. Tuy nhiên không chỉ rõ là chúng nên xuất hiện trước hay sau các dòng dữ liệu khác trả về. Một cách nghiêm túc, bạn thực sự cần phải chỉ định rõ cách sắp xếp **NULL** cho tất cả các cột có thể **NULL** trong mệnh đề **order by** để có thứ tự trả về xác định rõ ràng. Tuy nhiên thực tế là phần tùy chọn mở rộng không được thực hiện trong SQL Server 2012 hay trong MySQL 5.6. Ngược lại Oracle hỗ trợ việc sắp xếp **NULL** ngay cả khi chưa có tiêu chuẩn đưa ra, nhưng nó vẫn chưa chấp nhận điều đó trong bảng phát hành 11g (hình 6.5). Do đó cơ sở dữ liệu của Oracle không thể thực hiện một pipeline (đường ống) **order by** khi sắp xếp với **NULLS FIRST**. Chỉ có PostgreSQL (kể từ phiên bản 8.3) là hỗ trợ công cụ tùy chỉnh **NULLS** trong cả hai lệnh : **order by** và **define** nghĩa chỉ mục.

*) **Tóm tắt tính năng được cung cấp bởi các hệ quản trị cơ sở dữ liệu:**

Figure 6.5. Database/Feature Matrix

	MySQL	Oracle	PostgreSQL	SQL Server
Read index backwards	✓	✓	✓	✓
Order by ASC/DESC	✓	✓	✓	✓
Index ASC/DESC	✗	✓	✓	✓
Order by NULLS FIRST/LAST	✗	✓	✓	✗
Default NULLS order	First	Last	Last	First
Index NULLS FIRST/LAST	✗	✗	✓	✗

47. INDEXING GROUP BY:

Cơ sở dữ liệu sql sử dụng 2 thuật toán để **GROUP BY** và 2 thuật toán này hoàn toàn khác nhau.

- Thuật toán đầu tiên, thuật toán băm, nhóm các bản ghi vào một bảng băm tạm thời. Khi tất cả các bản ghi đã được xử lý, bảng băm sẽ chính là kết quả được trả về.

- Thuật toán thứ hai, thuật toán sort/group, trước tiên sẽ sắp xếp các dữ liệu đầu vào bằng cách nhóm các key để các bản ghi của mỗi nhóm đứng liên tiếp nhau. Sau đó, cơ sở dữ liệu chỉ cần gộp chúng lại.

Nói chung, cả 2 thuật toán đều bao gồm 1 trạng thái tạm thời, vì vậy chúng không được thực thi theo kiểu pipeline (đường ống). Mặc dù vậy, thuật toán sort/group có thể sử dụng index để tránh việc sắp xếp, và nhờ đó nó có thể giúp **GROUP BY** theo kiểu pipeline.

* Note: MySQL 5.6 không sử dụng thuật toán băm. Dù vậy, việc tối ưu thuật toán sort/group hoạt động như sau

Xét câu truy vấn sau: câu truy vấn trả về doanh thu của ngày hôm qua và được **GROUP BY** product_id:

```
SELECT product_id, sum(eur_value)
  FROM sales
 WHERE sale_date = TRUNC(sysdate) - INTERVAL '1' DAY
 GROUP BY product_id;
```

Khi đã có index ở cột sale_date và product_id, thuật toán sort/group sẽ hợp lý hơn vì **INDEX RANGE SCAN** sẽ tự động trả về các bản ghi theo thứ tự mong muốn. Điều này có nghĩa là cơ sở dữ liệu không cần phải thực hiện việc sắp xếp - **GROUP BY** có thể được thực hiện theo kiểu pipeline.

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		17	192
1	SORT GROUP BY NOSORT		17	192
2	TABLE ACCESS BY INDEX ROWID	SALES	321	192
*3	INDEX RANGE SCAN	SALES_DT_PR	321	3

Kế hoạch thực hiện câu lệnh của cơ sở dữ liệu Oracle sẽ đánh dấu một quá trình **SORT GROUP BY** (theo kiểu pipeline) bằng **NOSORT**. Kế hoạch thực hiện của các cơ sở dữ liệu khác không đề cập đến bất kì quá trình sắp xếp nào.

Group by theo kiểu pipeline cần những điều kiện tương tự như order by theo kiểu pipeline, ngoại trừ việc không có tùy chọn **ASC** và **DESC**. Điều này có nghĩa là việc tạo ra 1 index với sự tùy chọn **ASC/DESC** sẽ không ảnh hưởng đến sự thực thi **GROUP BY**. Điều tương tự cũng đúng cho **NULL FIRST/LAST**. Dù vậy vẫn có

những cơ sở dữ liệu không thể sử dụng ASC/DESC index cho **GROUP BY** theo kiểu pipeline.

***Chú Ý:** Với PostgreSQL, phải có mệnh đề **ORDER BY** kiểu **NULLS LAST** để có thể được sử dụng trong **GROUP BY** theo kiểu pipeline. Cơ sở dữ liệu Oracle không thể đọc ngược thứ tự index, vì vậy không thể thực thi **GROUP BY** theo kiểu pipeline mà theo sau bằng mệnh đề **ORDER BY**.

Giả sử muốn mở rộng câu truy vấn để lấy được tất cả các mua bán kể từ ngày hôm qua, điều này sẽ ngăn chặn **GROUP BY** theo kiểu pipeline vì **INDEX RANGE SCAN** không đưa ra các rows được sắp xếp bằng khóa nhóm.

```
SELECT product_id, sum(eur_value)
  FROM sales
 WHERE sale_date >= TRUNC(sysdate) - INTERVAL '1' DAY
 GROUP BY product_id;
```

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		24	356
1	HASH GROUP BY		24	356
2	TABLE ACCESS BY INDEX ROWID	SALES	596	355
*3	INDEX RANGE SCAN	SALES_DT_PR	596	4

Thay vào đó, cơ sở dữ liệu Oracle sẽ sử dụng thuật toán băm. Lợi thế của thuật toán băm là nó chỉ cần lưu lại kết quả sau khi đã gộp, còn thuật toán sort/group phải lưu lại toàn bộ đầu vào. Nói cách khác, thuật toán băm tốn ít bộ nhớ hơn.

Giống như đối với **ORDER BY** theo kiểu pipeline, thực thi nhanh không phải là điều quan trọng nhất trong việc thực thi **GROUP BY** theo kiểu pipeline. Điều quan trọng hơn là cơ sở dữ liệu thực thi theo kiểu pipeline và trả về kết quả đầu tiên trước khi đọc xong toàn bộ đầu vào. Đây là điều kiện bắt buộc cho các phương pháp tối ưu được giải thích trong các chương sau.

Bạn có thể suy nghĩ bất kỳ một hoạt động nào khác của cơ sở dữ liệu, bên cạnh việc sắp xếp và gom nhóm, mà có thể là sử dụng chỉ mục để tránh chi phí sắp xếp.

48. CHAPTER7: PARTIAL RESULT

Truy vấn một phần (partial result): Một tập nhỏ các bản ghi thuộc kết quả trả về từ một câu truy vấn SQL.

Đôi khi bạn không cần kết quả đầy đủ của một truy vấn SQL mà chỉ là vài hàng đầu tiên - ví dụ, chỉ hiển thị 10 tin nhắn gần đây. Trong trường hợp này, cách mà ta thường cho phép người dùng duyệt qua các tin nhắn cũ hơn - hoặc là sử dụng phương pháp truyền thống là điều hướng bằng phân trang hoặc hiện đại hơn là dùng “thanh cuộn vô hạn”. Các truy vấn SQL có thể được sử dụng cho chức năng này, tuy nhiên lại gây ra các vấn đề về hiệu năng nghiêm trọng nếu *tất cả* các tin nhắn phải được sắp xếp để tìm những cái mới nhất. Một đường ống lệnh (pipelined) **order by** (xem “Indexing Order By” ở Chương 6) tối ưu rất mạnh cho những truy vấn như vậy.

Chương này chỉ ra làm thế nào để sử dụng một đường ống lệnh **order by** hiệu quả để lấy một phần kết quả. Mặc dù cú pháp của các truy vấn này thay đổi với mỗi CSDL, chúng vẫn thực hiện các truy vấn một cách rất giống nhau.

49. LẤY VỀ TOP-N BẢN GHI (hoặc DÒNG - ROWS)

Câu lệnh TOP-N là câu lệnh để giới hạn số lượng các kết quả lấy về. Đây là câu lệnh thường được sử dụng khi người dùng muốn lấy về các kết quả tốt nhất theo một tiêu chí nào đó trong 1 tập kết quả. Để thực hiện câu lệnh có hiệu quả, việc xếp hạng bản ghi phải được hoàn thành bằng cách thực hiện order by theo phương pháp đường ống.

TIP

Hãy cho CSDL biết rằng bạn không cần tất cả bản ghi.

Cách đơn giản nhất để chỉ trả về các hàng đầu tiên của truy vấn là tìm nạp các hàng được yêu cầu và sau đó đóng câu truy vấn. Thật không may, bộ tối ưu hóa không thể đoán trước rằng khi nào cần chuẩn bị kế hoạch thực thi. Để chọn kế hoạch thực thi tốt nhất, bộ tối ưu hóa phải biết ứng dụng sẽ nạp tất cả các dòng trả về hay không. Trong trường truy vấn lấy toàn bộ dữ liệu trả về, việc quét toàn bộ bảng với toán tử sắp xếp rõ ràng có thể hoạt động tốt nhất, mặc dù một đường ống lệnh **order by** có thể tốt hơn khi tìm nạp chỉ mười hàng - ngay cả khi CSDL phải lấy từng hàng riêng lẻ. Điều đó có nghĩa là trình tối ưu hóa phải biết bạn có muốn dùng câu lệnh trước khi tìm nạp tất cả các hàng, sao cho nó có thể chọn kế hoạch thực hiện tốt nhất hay không.

Các chuẩn SQL loại trừ yêu cầu này trong một thời gian dài. Tiện ích **fetch first** đã được giới thiệu trong SQL:2008 và hiện chỉ có trong IBM DB2, PostgreSQL và SQL Server 2012. Một mặt, đó là vì tính năng này không phải là một phần mở rộng cốt lõi, và mặt khác nó là bởi vì mỗi CSDL đã được cung cấp các giải pháp độc quyền của riêng mình trong nhiều năm.

Các ví dụ sau cho thấy việc sử dụng các tiện ích mở rộng nổi tiếng này bằng cách truy vấn 10 lần bán hàng gần đây nhất. Bản chất là như nhau: lấy tất cả các doanh số bán hàng, bắt đầu với một trong những lần bán gần đây nhất. Cú pháp top-N tương ứng chỉ hủy bỏ việc thực thi sau khi tìm nạp đủ 10 hàng.

MySQL:

MySQL và PostgreSQL dùng từ khóa **limit** để giới hạn số bản ghi lấy về.

```
SELECT *
  FROM sales
 ORDER BY sale_date DESC
 LIMIT 10;
```

ORACLE DATABASE:

Oracle cung cấp một cột giả tên là *ROWNUM* để đếm số bản ghi một cách tự động. Để dùng cột này, chúng ta sẽ đặt câu lệnh cần thực thi vào trong một câu lệnh lồng:

```
SELECT *
  FROM (
    SELECT *
      FROM sales
     ORDER BY sale_date DESC
  )
 WHERE rownum <= 10;
```

PostgreSQL

Gần đây Postgre cung cấp từ câu lệnh **fetchfirst** từ phiên bản 8.4. Trước đó hệ quản trị này sử dụng **limit** giống MySQL.

```
SELECT *
  FROM sales
 ORDER BY sale_date DESC
FETCH FIRST 10 ROWS ONLY;
```

SQLServer

SQL Server cung cấp từ khóa **top** để giới hạn số bản ghi lấy về.

```
SELECT TOP 10 *
  FROM sales
 ORDER BY sale_date DESC;
```

Bắt đầu từ bản phát hành 2012, SQL Server cũng hỗ trợ tiện ích mở rộng **fetch first**. Những câu lệnh trên đều được hệ quản trị CSDL nhìn nhận như một câu lệnh top-N query.

IMPORTANT

CSDL chỉ có thể tối ưu hóa truy vấn cho một câu truy vấn trả về kết quả một phần nếu nó biết điều này ngay từ đầu.

Nếu trình tối ưu hóa nhận thức được thực tế kết quả trả về chỉ cần mười bản ghi, nó sẽ ưu tiên sử dụng một đường ống lệnh **orderby** nếu áp dụng:

Operation	Name	Rows	Cost
SELECT STATEMENT		10	9
COUNT STOPKEY			
VIEW		10	9
TABLE ACCESS BY INDEX ROWID	SALES	1004K	9
INDEX FULL SCAN DESCENDING	SALES_DT_PR	10	3

Kế hoạch thực thi câu truy vấn của Oracle cho thấy nó kết thúc với thao tác COUNT STOPKEY. Điều đó có nghĩa là CSDL đã nhận ra cú pháp của câu truy vấn top-N.

TIP

Phụ lục A, “Excution Plans”, tóm tắt các thao tác tương ứng cho MySQL, Oracle, PostgreSQL và SQL Server.

Sử dụng đúng cú pháp chỉ là một phần bởi vì việc kết thúc thực thi câu truy vấn một cách hiệu quả còn yêu cầu sự thực thi của các thao tác nằm phía dưới trong một phương pháp đường ống lệnh. Có nghĩa là mệnh đề **order by** phải được bao phủ bởi một chỉ mục (covered by an index) - ví dụ chỉ mục SALE_DT_PR trên cột SALE_DATE và PRODUCT_ID trong ví dụ này. Bằng việc sử dụng chỉ mục, CSDL có thể tránh một hoạt động sắp xếp tường minh và do đó ngay lập tức có thể gửi các bản ghi đến cho ứng dụng thông qua việc đọc từ chỉ mục. Việc thực thi bị hủy bỏ sau khi lấy ra mười bản ghi nên CSDL không đọc nhiều bản ghi hơn số bản ghi cần lấy.

IMPORTANT

Truy vấn top-N sử dụng đường ống không cần đọc và sắp xếp toàn bộ tập hợp kết quả.

Nếu không có chỉ mục thích hợp trên bảng SALE_DATE cho lệnh **order by**, CSDL phải đọc và sắp xếp toàn bộ bảng. Bản ghi đầu tiên chỉ lấy được sau khi đọc bản ghi cuối cùng từ bảng.

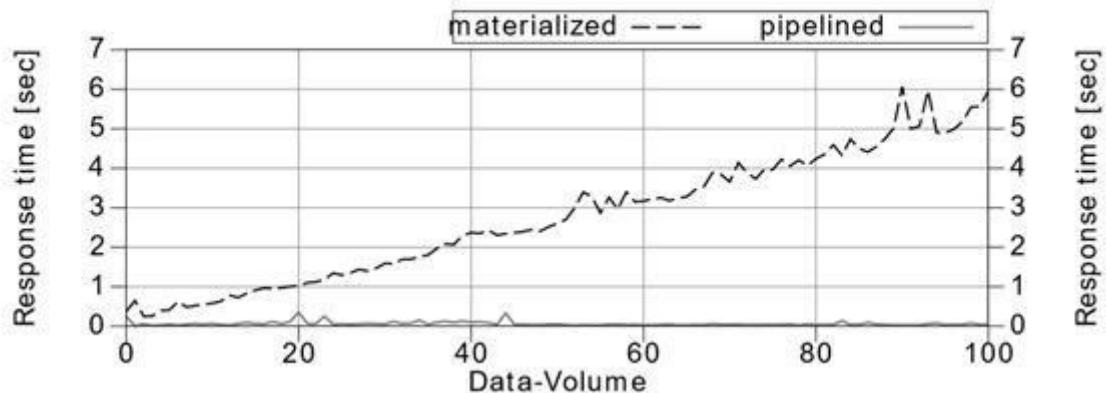
Operation	Name	Rows	Cost
SELECT STATEMENT		10	59558
COUNT STOPKEY			
VIEW		1004K	59558
SORT ORDER BY STOPKEY		1004K	59558
TABLE ACCESS FULL	SALES	1004K	9246

Kế hoạch thực thi câu truy vấn không có **order by** bằng đường ống chậm như việc hủy bỏ thực thi từ phía client (sau khi mà CSDL đã làm mọi công việc và trả về mọi kết quả, client lọc lấy kết quả mong muốn). Sử dụng cú pháp top-N vẫn là tốt hơn bởi vì CSDL không cần thực hiện đầy đủ kết quả mà chỉ trên mười bản ghi gần nhất. Điều này đòi hỏi ít bộ nhớ hơn đáng kể. Kế hoạch thực thi của Oracle cho thấy sự tối ưu hóa này với sự sửa đổi STOPKEY trên thao tác SORT ORDERBY.

Những ưu điểm của một câu đường ống lệnh truy vấn top-N bao gồm: không chỉ tăng hiệu suất ngay lập tức mà còn tăng cường khả năng mở rộng. Nếu không sử dụng phương pháp thực thi theo đường ống, thời gian phản hồi của truy vấn top-N sẽ tăng lên phụ thuộc vào kích thước bảng. Trong khi đó, thời gian phản hồi khi sử dụng một thực thi trên đường ống lệnh chỉ tăng lên phụ thuộc vào số lượng các bản ghi cần lấy. Nói cách khác, thời gian đáp ứng của một truy vấn top-N trên đường ống là luôn luôn là như nhau; điều này hầu như là độc lập với kích thước của bảng. Chỉ khi độ sâu của cây B-tree tăng thì truy vấn trở nên chậm hơn một chút.

Hình 7.1 cho thấy khả năng mở rộng của cả hai biến thể (materialized và pipelined) đối với sự gia tăng của khối lượng dữ liệu. Để thấy sự tăng lên của thời gian phản hồi là tuyến tính đối với việc thực thi mà không có **order by** qua đường ống. Thời gian phản hồi cho việc thực thi trên đường ống lệnh vẫn không đổi.

Figure 7.1. Scalability of Top-N Queries



Mặc dù thời gian phản hồi của một truy vấn top-N không phụ thuộc vào kích thước bảng, nó vẫn tăng lên nếu số lượng bản ghi cần lấy tăng lên. Thời gian phản hồi do đó sẽ tăng gấp đôi khi chọn hai lần trên nhiều bản ghi. Điều này đặc biệt đáng kể đối với các truy vấn "phân trang", các truy vấn dạng này thường bắt đầu ở dòng đầu tiên của trang đầu tiên khi cần lấy dữ liệu cho một trang mới; chúng sẽ đọc các bản ghi đã được hiển thị trên trang trước và loại bỏ các bản ghi đó trước khi cuối cùng đưa ra kết quả cho trang thứ hai. Tuy nhiên, có một giải pháp cho vấn đề này mà chúng ta sẽ thấy trong phần tiếp theo.

50. PHÂN TRANG KẾT QUẢ

Sau khi thực hiện một top-N truy vấn để nhận trang đầu tiên một cách hiệu quả, bạn thường sẽ cần một câu truy vấn khác để lấy các trang tiếp theo. Thách thức ở đây là CSDL phải bỏ qua các hàng từ các trang trước đó. Có hai phương pháp khác nhau để vượt qua thách thức này: thứ nhất là dùng *offset*, bằng cách đánh số các dòng từ đầu và sử dụng một bộ lọc cho số dòng để loại bỏ các hàng trước khi gửi yêu cầu lấy một trang. Cách thứ hai, mà tôi gọi là phương pháp *seek*, tìm kiếm đầu vào (entry) cuối cùng của trang trước và chỉ lấy các dòng sau đó.

Các ví dụ sau cho thấy sự sử dụng rộng rãi của phương pháp dùng offset. Lợi thế chính của nó là dễ xử lý - đặc biệt với các CSDL mà có từ khóa dành riêng cho nó (**offset**). Từ khóa này đã được đưa vào chuẩn SQL như là một phần của tiện ích **fetch first**.

MySQL

MySQL và PostgreSQL cung cấp biểu thức **offset** cho việc loại bỏ các hàng cụ thể khi bắt đầu câu truy vấn top-N. Mệnh đề **limit** được áp dụng sau đó.

```
SELECT *
  FROM sales
 ORDER BY sale_date DESC
 LIMIT 10 OFFSET 10;
```

CSDL ORACLE

CSDL Oracle cung cấp cột giả ROWNUM mà số hàng trong kết quả được đặt tự động. Tuy nhiên, không thể áp dụng bộ lọc lớn hơn hoặc bằng (\geq) trên cột giả này. Để sử dụng cột giả này, trước hết bạn cần phải "materialize" số hàng bằng cách đổi tên cột bằng một bí danh.

```
SELECT *
  FROM ( SELECT tmp.*, rownum rn
    FROM ( SELECT *
            FROM sales
           ORDER BY sale_date DESC
        ) tmp
   WHERE rownum <= 20
  )
 WHERE rn > 10;
```

Lưu ý việc sử dụng bí danh RN cho các ràng buộc thấp hơn và cột giả ROWNUM cho các ràng buộc cao hơn.

POSTGRESQL

Tiện ích **fetch first** cũng định nghĩa một mệnh đề **offset ... rows**. Tuy nhiên PostgreSQL chỉ dùng **offset** mà không có từ khóa **rows**. Cú pháp **limit/offset** sử dụng trong ví dụ MySQL dùng được trong PostgreSQL.

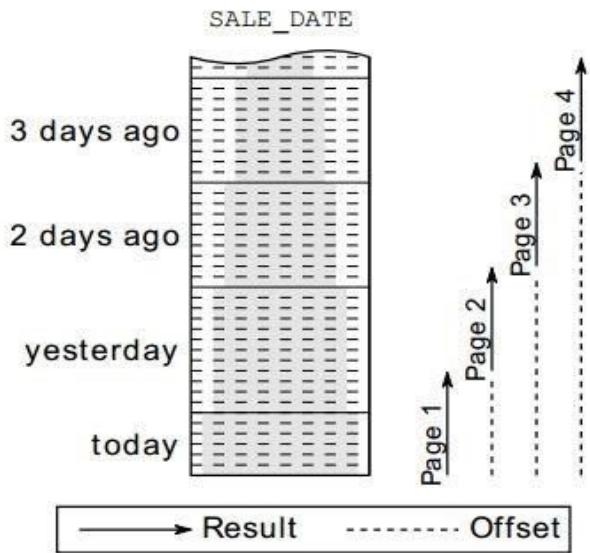
```
SELECT *
  FROM sales
 ORDER BY sale_date DESC
OFFSET 10
FETCH NEXT 10 ROWS ONLY;
```

SQLSERVER

SQL Server không có tiện ích “**offset**” cho **top** dữ liệu mà sử dụng tiện ích **fetch first** với SQL Server 2012 (“Denali”). Mệnh đề **offset** là bắt buộc mặc dù trong định nghĩa chuẩn nó là tùy chọn.

```
SELECT *
  FROM sales
 ORDER BY sale_date DESC
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY;
```

Bên cạnh sự đơn giản, một lợi thế khác của phương pháp này là bạn chỉ cần bỏ qua số hàng không cần để lấy một trang tùy ý. Tuy nhiên, CSDL phải đếm tất cả các hàng từ đầu cho đến khi tới trang được yêu cầu. Hình 7.2 cho thấy phạm vi chỉ mục được quét trở nên lớn hơn khi nạp nhiều trang hơn.



Hình 7.2. Truy cập sử dụng phương pháp dùng Offset

Phương pháp **offset** này có hai nhược điểm: (1) các trang trôi dạt khi chèn thêm **sales** mới bởi vì việc đánh số luôn luôn được thực hiện lại từ đầu; (2) thời gian đáp ứng tăng khi duyệt ngược lại.

Phương pháp **seek** tránh được cả hai vấn đề vì nó sử dụng các giá trị của trang trước như một mốc phân cách. Điều đó có nghĩa là nó tìm kiếm các giá trị phải nằm sau mục cuối cùng từ trang trước đó. Điều này có thể được thể hiện bằng một mệnh đề **where** đơn giản. Nói theo cách khác: phương pháp **seek** chỉ đơn giản là không chọn các giá trị đã được hiển thị.

Ví dụ tiếp theo giới thiệu phương thức **seek**. Để thuận tiện cho việc diễn giải, chúng ta sẽ bắt đầu với giả định rằng chỉ có một lần bán mỗi ngày. Điều này làm cho **SALE_DATE** là khóa duy nhất. Để chọn các lần bán sau một ngày cụ thể bạn phải sử dụng một điều kiện ít hơn (**<**) do thứ tự sắp xếp giảm dần của thời gian. Đối với thứ tự tăng dần, bạn sẽ phải sử dụng điều kiện lớn hơn (**>**). Mệnh đề **fetch first** chỉ được sử dụng để giới hạn kết quả đến mười hàng.

```

SELECT *
  FROM sales
 WHERE sale_date < ?
 ORDER BY sale_date DESC
 FETCH FIRST 10 ROWS ONLY;
  
```

Thay vì dùng số hàng, bạn sử dụng giá trị cuối cùng của trang trước để xác định giới hạn dưới.

Điều này có một lợi ích rất lớn về mặt hiệu suất vì CSDL có thể sử dụng điều kiện **SALE_DATE < ?** để truy cập index. Điều đó có nghĩa là CSDL thực sự có thể bỏ

qua các hàng từ trang trước đó. Trên hết, bạn luôn luôn nhận được dữ liệu trả về theo một trật tự nhất quán, ổn định ngay cả khi có các hàng mới chèn vào.

Tuy nhiên, phương pháp này không dùng được nếu có nhiều hơn một lần bán hàng được thực hiện mỗi ngày - như thể hiện trong hình 7.2 - vì sử dụng ngày cuối cùng từ trang đầu tiên ("Yesterday") sẽ bỏ qua *tất cả* các kết quả từ ngày hôm qua - không chỉ là bỏ qua những bản ghi đã được hiển thị trên trang đầu tiên.

Nếu không có mệnh đề **order by** xác định một thứ tự nhất định, theo định nghĩa CSDL không trả về thứ tự các dòng cũng theo một thứ tự nhất định. Lý do duy nhất bạn thường nhận được một dãy các hàng thống nhất theo thứ tự khi thực hiện truy vấn là do CSDL thường thực hiện truy vấn theo cách tương tự nhau. Tuy nhiên, CSDL trên thực tế có thể xáo trộn các hàng có cùng SALE_DATE và vẫn thực hiện theo mệnh đề **order by**. Trong các bản phát hành gần đây của CSDL bạn có thể nhận được kết quả theo một thứ tự khác nhau mỗi khi bạn chạy câu truy vấn, không phải vì CSDL xáo trộn kết quả một cách cố ý mà vì CSDL có thể truy vấn một cách song song. Điều đó có nghĩa là với cùng một kế hoạch thực thi có thể cho ra một kết quả là một dãy hàng khác nhau vì các luồng thực thi kết thúc theo thứ tự không xác định.

IMPORTANT

Việc đánh số trang cần xác định một thứ tự nào đó.

Ngay cả khi các hàm đặc tả chỉ yêu cầu sắp xếp "theo ngày, mới nhất đầu tiên", chúng ta phải đảm bảo rằng mệnh đề **order by** sinh ra một trình tự hàng xác định. Với mục đích này, chúng ta có thể cần mở rộng mệnh đề **order by** với các cột tùy ý chỉ để đảm bảo rằng chúng ta có thể nhận được kết quả thống nhất theo một thứ tự nhất định. Nếu chỉ mục được sử dụng cho đường ống lệnh **order by** có thêm các cột khác, đó là một khởi đầu tốt để thêm chúng vào mệnh đề **order by** để chúng ta có thể tiếp tục sử dụng chỉ mục này cho các đường ống lệnh **order by**. Nếu điều này vẫn không mang lại một thứ tự sắp xếp xác định, chỉ cần thêm bất kỳ (các) cột unique (duy nhất) và mở rộng chỉ mục cho phù hợp.

Trong ví dụ sau, chúng ta mở rộng mệnh đề **order by** bằng chỉ mục cho khóa chính **SALE_ID** để có được dữ liệu trả về theo thứ tự nhất quán. Hơn nữa chúng ta phải áp dụng logic "comes after" cho cả hai cột với nhau để có được kết quả như ý:

```
CREATE INDEX sl_dtid ON sales (sale_date, sale_id);

SELECT *
  FROM sales
 WHERE (sale_date, sale_id) < (?, ?)
 ORDER BY sale_date DESC, sale_id DESC
 FETCH FIRST 10 ROWS ONLY;
```

Mệnh đề **where** sử dụng cú pháp "row values" ít được biết đến (xem "SQL Row Values"). Nó kết hợp nhiều giá trị vào một đơn vị logic áp dụng cho các toán tử so sánh thông thường. Như với giá trị vô hướng, điều kiện " $<=$ " tương ứng với "comes after" khi phân loại theo thứ tự giảm dần. Điều đó có nghĩa là truy vấn chỉ xem xét doanh thu đến sau một cặp **SALE_DATE**, **SALE_ID**.

Mặc dù cú pháp các giá trị dòng (row values) là một phần của tiêu chuẩn SQL, chỉ có một vài CSDL hỗ trợ nó. SQL Server 2012 ("Denali") không hỗ trợ các giá trị dòng. Về nguyên tắc CSDL Oracle hỗ trợ các giá trị dòng, nhưng không thể áp dụng các toán tử điều khiển trên chúng (ORA-01796). MySQL đánh giá các biểu thức giá trị dòng một cách chính xác nhưng không thể sử dụng chúng làm vị từ truy cập trong quá trình truy cập chỉ mục. Tuy nhiên, PostgreSQL hỗ trợ cú pháp giá trị dòng và sử dụng chúng để truy cập vào chỉ mục nếu có một chỉ mục tương ứng có sẵn.

Tuy nhiên, có thể sử dụng một phương pháp khác gần giống phương pháp **seek** với các CSDL không hỗ trợ đúng các giá trị dòng - mặc dù phương pháp này hay và hiệu quả như các giá trị dòng trong PostgreSQL. Với phương pháp này, chúng ta phải sử dụng các so sánh "thông thường" để sử dụng logic được áp dụng như trong ví dụ của Oracle:

```
SELECT *
  FROM (
    SELECT *
      FROM sales
     WHERE sale_date <= ?
       AND NOT (sale_date = ? AND sale_id >= ?)
     ORDER BY sale_date DESC, sale_id DESC
  )
 WHERE rownum <= 10;
```

Mệnh đề **where** bao gồm hai phần. Phần đầu tiên chỉ sử dụng **SALE_DATE** và sử dụng điều kiện dưới (\leq) - nó sẽ chọn nhiều hàng hơn nếu cần. Phần này của mệnh đề **where** là đủ đơn giản để tất cả các CSDL có thể sử dụng nó để truy cập vào chỉ mục. Phần thứ hai của mệnh đề **where** là loại bỏ các hàng thừa đã được hiển thị trên trang trước. Mục có tiêu đề "Indexing Equivalent Logic" phía dưới giải thích lý do tại sao mệnh đề **where** được dùng theo cách này.

Kế hoạch thực thi cho thấy rằng CSDL sử dụng phần đầu tiên của mệnh đề **where** để truy cập.

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		10	4
*1	COUNT STOPKEY			
2	VIEW		10	4
3	TABLE ACCESS BY INDEX ROWID	SALES	50218	4
*4	INDEX RANGE SCAN DESCENDING	SL_DTIT	2	3

Predicate Information (identified by operation id):

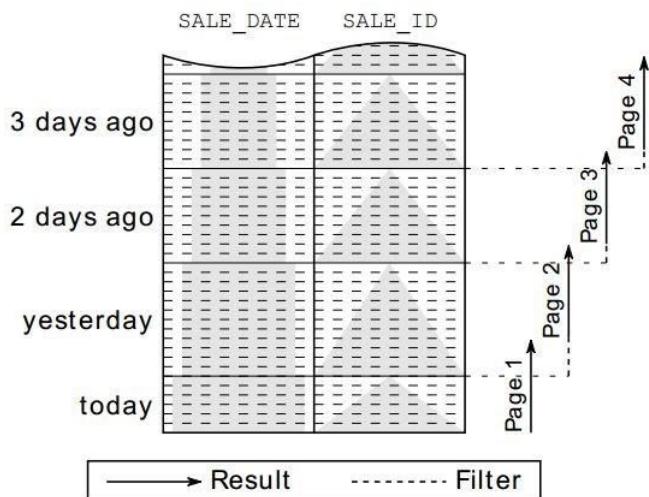
```

1 - filter(ROWNUM<=10)
4 - access("SALE_DATE" <= :SALE_DATE)
      filter("SALE_DATE" <> :SALE_DATE
             OR "SALE_ID" < TO_NUMBER(:SALE_ID))

```

Việc tìm và duyệt trên các nút lá (access predicate) sử dụng SALE_DATE cho phép CSDL bỏ qua những ngày đã được hiển thị đầy đủ trên các trang trước đó. Phần thứ hai của mệnh đề **where** là filter predicate. Điều đó có nghĩa là CSDL kiểm tra một vài mục dòng trang trước đó, nhưng sẽ bỏ chúng ngay lập tức.

Figure 7.3. Access Using the Seek Method



Hình 7.3 cho thấy đường dẫn truy cập tương ứng

Giá trị Dòng SQL

Bên cạnh các giá trị vô hướng thông thường, chuẩn SQL cũng định nghĩa các *row value constructors*. Tức "Chỉ định một bộ các giá trị phải được xây dựng thành một dòng hàng hoặc một phần của hàng" [SQL: 92, §7.1: <row value constructor>]. Về mặt cú pháp, các giá trị hàng là các danh sách trong ngoặc đơn. Cú pháp này được biết đến nhiều nhất vì nó được sử dụng trong câu lệnh **insert**.

Sử dụng các hàm tạo giá trị giá trị trong mệnh đề **where** ít được biết đến hơn nhưng vẫn hoàn toàn hợp lệ. Các chuẩn SQL thực sự xác định tất cả các toán tử so sánh cho row value constructor. Ví dụ, định nghĩa cho toán tử nhỏ hơn ($<$) như sau:

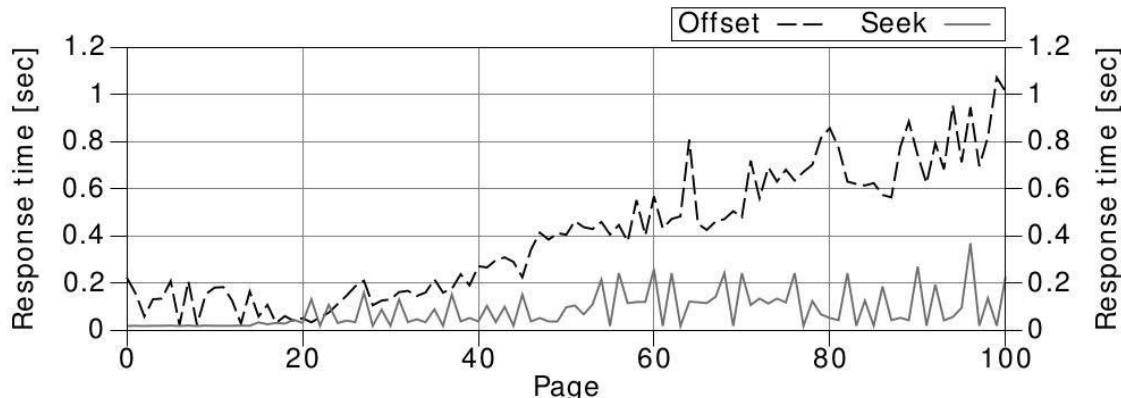
" $Rx < Ry$ " là đúng khi và chỉ khi $RXi = RYi$ cho tất cả $i < n$ và
 $RXn < RYn$ cho một số n .

-SQL: 92, §8.2.7.2

Trong đó i và n phản ánh các vị trí chỉ mục trong danh sách. Điều đó có nghĩa là giá trị dòng hàng RX nhỏ hơn RY nếu giá trị RXn nhỏ hơn giá trị RYn tương ứng và tất cả các cặp giá trị trước đó bằng nhau ($RXi = RYi$, với $i < n$).

Định nghĩa này làm cho biểu thức $RX < RY$ đồng nghĩa với "RX loại trước RY" đó là chính xác logic chúng ta cần cho phương pháp tìm kiếm.

Hình 7.4 so sánh hiệu năng của phương pháp **offset** và **seek**. Các thông số của phép đo không cho thấy sự khác biệt từ phía bên trái của biểu đồ, tuy nhiên có sự khác biệt rõ ràng có thể nhìn thấy được từ khoảng 20 trang trở đi.



Hình 7.4 khả năng mở rộng khi tìm nạp các trang kế tiếp

Tuy nhiên, phương pháp **seek** cũng có những hạn chế, sự khó khăn trong việc sử dụng nó là một trong những vấn đề quan trọng nhất. Bạn không chỉ phải xác định mệnh đề **where** một cách cẩn thận - bạn cũng không thể lấy các trang một cách tùy ý. Hơn nữa bạn cần phải đảo ngược tất cả các phép so sánh và sắp xếp để thay đổi hướng duyệt qua. Nói một cách chính xác, hai phương thức **bỏ qua các trang (skipping pages)** và **duyệt** là không cần thiết khi sử dụng cơ chế cuộn vô tận cho giao diện người dùng.

Những lập luận logic tương đương (indexing equivalent logic)

Một điều kiện logic có thể được diễn đạt bằng nhiều cách khác nhau. Bạn có thể thực hiện lại những câu lệnh logic ở trên theo cách sau:

```
WHERE (
    (sale_date < ?)
    OR
    (sale_date = ? AND sale_id < ?)
)
```

Câu lệnh này bao gồm nhiều điều kiện và nó ít nhất dễ hiểu hơn cho con người. CSDL lại có một cách nhìn khác. Nó không nhận ra mệnh đề **where** chọn ra tất cả các bản ghi bắt đầu từ một cặp tương ứng SALE_DATE/SALE_ID với điều kiện SALE_DATE của cả hai điều kiện so sánh là giống nhau. Thay vào đó, CSDL sử dụng toàn bộ mệnh đề **where** như là một vị tử lọc. Chúng ta có thể mong muốn trình tối ưu đặt điều kiện SALE_DATE <= ? ra ngoài hai vế so sánh, nhưng không có CSDL nào làm việc này. Dù vậy, chúng ta có thể tự thêm các điều kiện dư thừa này - mặc dù việc đó không làm tăng thêm khả năng dễ đọc:

```
WHERE sale_date <= ?
AND (
    (sale_date < ?)
    OR
    (sale_date = ? AND sale_id < ?)
)
```

May mắn là tất cả CSDL có thể sử dụng mệnh đề **where** này để truy cập. Tuy nhiên thì mệnh đề này khó hiểu hơn câu logic trên. Hơn nữa, câu logic ban đầu tránh được rủi ro do phần không cần thiết được xóa khỏi mệnh đề **where** sau đó.

51. SỬ DỤNG WINDOW FUNCTIONS CHO PHÂN TRANG.

Window functions cung cấp những cách khác để thực thi việc phân trang trong SQL. Đây là một điểm linh hoạt, và trên hết, một phương thức phù hợp với tiêu chuẩn. Tuy nhiên, chỉ SQL Server và Oracle database có thể sử dụng chúng cho câu truy vấn đường ống lệnh top-N. PostgreSQL không hủy việc duyệt chỉ mục sau khi lấy đủ các hàng và do đó thực thi những câu truy vấn đó không thực sự hiệu quả. MySQL không hỗ trợ các chức năng của cửa sổ.

Ví dụ sau đây sử dụng chức năng cửa sổ ROW_NUMBER cho một truy vấn phân trang:

```
SELECT *
  FROM ( SELECT sales.*,
               , ROW_NUMBER() OVER (ORDER BY sale_date DESC
                                      , sale_id DESC) rn
        FROM sales
      ) tmp
 WHERE rn between 11 and 20
 ORDER BY sale_date DESC, sale_id DESC;
```

Hàm ROW_NUMBER liệt kê các hàng theo thứ tự sắp xếp được xác định sẵn trong mệnh đề **over**. Mệnh đề **where** bên ngoài sử dụng danh sách liệt kê này để giới hạn kết quả của trang thứ 2 (từ dòng 11 đến 20).

Oracle database sắp xếp điều kiện loại trừ và sử dụng đánh chỉ mục trên SALE_DATE và SALE_ID để tạo ra truy vấn đường ống lệnh top-N:

Id	Operation	Name	Rows	Cost
0	SELECT STATEMENT		1004K	36877
*1	VIEW		1004K	36877
*2	WINDOW NOSORT STOPKEY		1004K	36877
3	TABLE ACCESS BY INDEX ROWID	SALES	1004K	36877
4	INDEX FULL SCAN DESCENDING	SL_DTID	1004K	2955

Predicate Information (identified by operation id):

```

1 - filter("RN">>=11 AND "RN"<=20)
2 - filter(ROW_NUMBER() OVER (
    ORDER BY "SALE_DATE" DESC, "SALE_ID" DESC )<=20)

```

Phép toán **WINDOW NOSORT STOPKEY** diễn tả việc không có sự sắp xếp nào (**NOSORT**) và những CSDL hủy bỏ việc thực thi khi đạt đến ngưỡng trên (**STOPKEY**). Do đó việc hủy bỏ các hoạt động được thực theo cách thức đường ống, nghĩa là những câu truy vấn này hiệu quả như phương thức **offset** được giải thích ở phần trước.

Sức mạnh của chức năng cửa sổ không chỉ là phân trang mà bao gồm cả phân tích tính toán. Nếu bạn không sử dụng các chức năng cửa sổ trước đây, bạn nên dành một ít thời gian để học các tài liệu liên quan.

52. Modifying Data - Cập nhật dữ liệu

Từ đầu sách đến giờ, chúng ta chỉ bàn về phần hiệu xuất, tối ưu truy vấn, nhưng SQL không chỉ là về truy vấn. Nó cũng hỗ trợ thao tác dữ liệu. Các lệnh tương ứng là **insert**, **delete**, và **update**, cái gọi là “data manipulation language- ngôn ngữ thao tác dữ liệu” (DML) - một phần của chuẩn SQL. Hiệu suất của các lệnh trên đa phần bị ảnh hưởng tiêu cực bởi việc đánh chỉ mục.

Một chỉ mục đơn thuần nó là sự rườm rà, dư thừa. Nó chỉ chứa dữ liệu mà cũng được cất trữ trong bảng. Trong hoạt động ghi, cơ sở dữ liệu phải giữ các dư thừa đó luôn nhất quán (giống nhau trong index và trong bảng). Cụ thể, nó có nghĩa là insert, delete hay update không chỉ ảnh hưởng đến bản chính mà các chỉ mục cũng bị ảnh hưởng.

53. Insert

Số các chỉ mục trên một bảng là cái hệ số chi phối nhiều nhất tới hiệu suất của insert. Các chỉ mục của một bảng càng nhiều, việc thực thi càng chậm. Câu lệnh INSERT là hoạt động duy nhất mà không thể trực tiếp hưởng lợi từ việc đánh chỉ mục vì nó không có mệnh đề WHERE.

Thêm một hàng mới vào một bảng bao gồm một vài bước. Trước tiên, database phải tìm nơi để cất trữ hàng. Đối với các bảng **heap** thông thường - không có thứ tự hàng cụ thể nào - cơ sở dữ liệu có thể lấy bất kì **block** nào có đủ không gian trống cho bảng này. Đây là một quá trình rất đơn giản và nhanh chóng, hầu hết được hoàn thành trong main memory. Tất cả những việc database làm sau đó là thêm dữ liệu mới vào khối dữ liệu tương ứng.

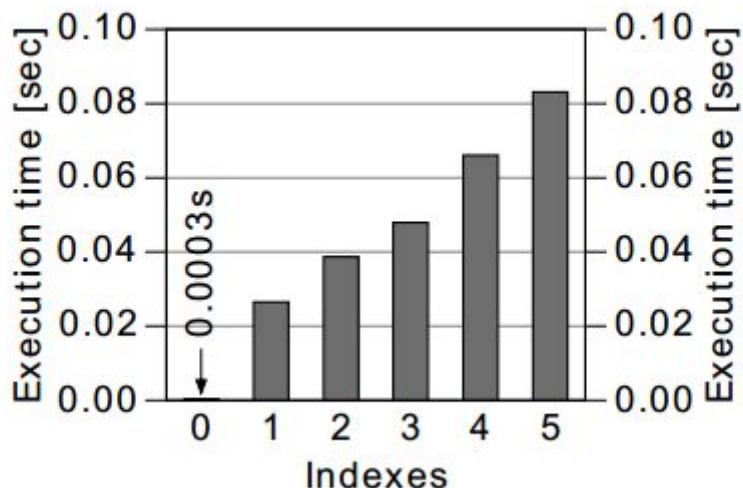
Nếu có nhiều index trên một bảng, CSDL phải đảm bảo các hàng mới thêm vào cũng được tìm thấy thông qua các chỉ mục. Vì lý do này mà nó phải thêm phần tử chỉ mục cho hàng mới cho mọi index. Do đó mà số chỉ mục càng nhiều sẽ khuếch đại chi phí cho mỗi thao tác insert.

Hơn nữa, thêm một phần tử mới vào một chỉ mục đắt hơn nhiều do với việc việc thêm vào một cấu trúc heap bởi vì CSDL phải giữ thứ tự các chỉ mục và cây cân bằng. Điều đó nghĩa là phần tử mới không thể được ghi cho bất kì khối nào - nó thuộc về một nút lá cụ thể. Mặc dù CSDL sử dụng cây chỉ mục để tìm đúng nút lá, nó vẫn phải đọc một vài các khối chỉ mục cho việc duyệt cây.

Chỉ khi nút lá đúng được xác định, database sẽ xác nhận rằng có đủ không gian trống trong nút này. Nếu không, cơ sở dữ liệu sẽ chia tách nút lá và phân phối lại các phần tử giữa nút cũ và nút mới. Quá trình này cũng ảnh hưởng tới cả node mới và node cũ, tức ảnh hưởng bị nhân đôi. **Xa hơn**, nút nhánh có thể cũng bị dày vì vậy nó có thể bị phân tách. Trong trường hợp xấu nhất, cơ sở dữ liệu tách tất cả

các node, tới node gốc. Đây là trường hợp duy nhất mà cây được thêm tầng và lớn hơn về độ sâu.

Sau tất cả, việc duy trì index là phần đắt nhất của hoạt động insert. Điều này cũng có thể thấy ở trong bảng 8.1 “hiệu xuất của insert theo số chỉ mục”. Việc thêm một chỉ mục đơn là đủ để tăng thời gian thực hiện. Mỗi index được thêm lại làm chậm các việc thực thi hơn nữa.



Hình 8.1: hiệu xuất của insert theo số chỉ mục

- Chú ý: Các chỉ mục đầu tiên tạo ra sự khác biệt lớn nhất.
Để tối ưu hóa hiệu xuất insert thì rất rât quan trọng việc giữ số index nhỏ.
- Mẹo: Sử dụng indexes cần xem xét kĩ lưỡng và cần tiết kiệm số index, và tránh các index rườm rà bất cứ khi nào có thể. Điều này cũng có lợi với các câu lệnh delete hay update.

Xem xét riêng câu lệnh INSERT, tốt nhất nên tránh hoàn toàn các chỉ mục - điều này có ảnh hưởng tốt nhất cho hiệu suất insert. Tuy nhiên các bảng không có chỉ mục là không thực tế trong các ứng dụng trong thế giới thực. Bạn luôn cần truy vấn dữ liệu vì vậy bạn cần đánh chỉ mục để cải thiện tốc độ truy vấn. Ngay đến các bảng log chỉ ghi thường có một khóa chính và một chỉ mục riêng cho mỗi bảng.

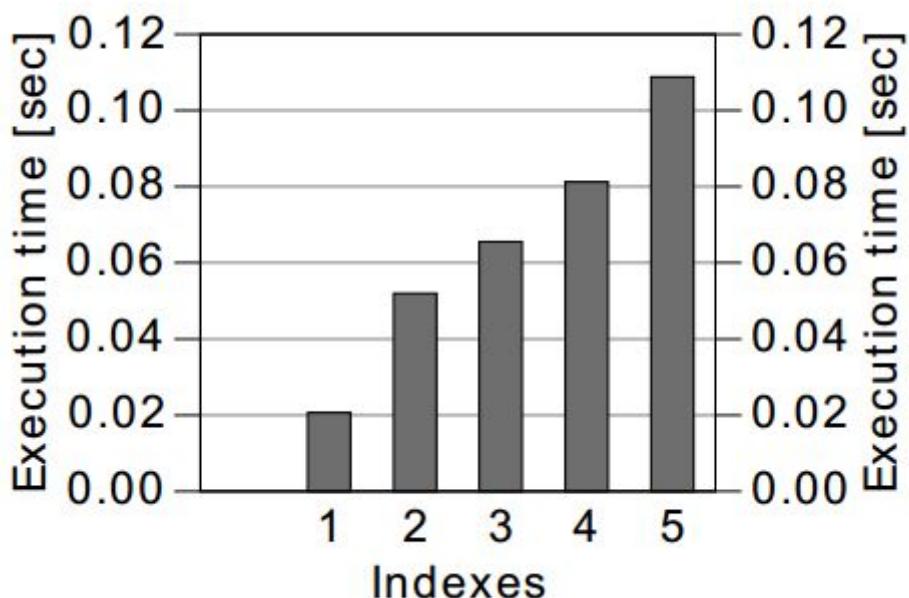
Tuy nhiên, hiệu suất khi không đánh chỉ chỉ mục là rât tốt mà chúng ta có thể nghĩ đến việc tạm thời drop tất cả các index trong khi load một lượng dữ liệu lớn - vì cung cấp index là không cần thiết cho bất kì câu lệnh SQL nào trong quá trình chờ đợi. Điều này có mang lại hiệu năng cao hơn nhiều mà có thể được nhìn thấy trên bảng so sánh trong thực tế, đây là thực hành phổ biến trong các kho dữ liệu.

Câu hỏi đặt ra là bảng ở hình 8.1 sẽ thay đổi thế nào khi sử dụng **index organized table** hay **clustered index**? Và có hay không một cách nào đó mà đánh index để làm cho câu lệnh insert nhanh hơn?

54. Delete

Không giống như câu lệnh insert, câu lệnh delete có mệnh đề where mà có thể sử dụng tất cả các phương pháp mô tả ở chương 2 “The where clause” để hưởng lợi trực tiếp từ việc đánh chỉ mục. Trên thực tế, lệnh delete làm việc giống như một việc lựa chọn các hàng xác định để xóa.

Việc xóa một hàng thực tế là một quá trình tương tự với việc insert một hàng - đặc biệt là về việc loại bỏ các tham chiếu/con trỏ của index và các hoạt động để giữ cho cột index cân bằng. Do đó mà biểu đồ hiệu xuất hiện trong hình 8.2 là gần giống với một biểu đồ hiệu xuất cho hành động insert.



Hình 8.2: Hiệu xuất lệnh delete theo số chỉ mục

Về lý thuyết, chúng ta mong đợi hiệu xuất của lệnh delete là tốt nhất trong trường hợp không có index - như với lệnh insert. Tuy nhiên, nếu không có index, cơ sở dữ liệu phải đọc toàn bộ bảng để tìm ra hàng cần xóa. Điều này có nghĩa rằng việc xóa thì nhanh hơn nhưng việc tìm ra hàng để xóa thì lại rất chậm. Trường hợp này do đó không được thể hiện trong hình 8.2.

Tuy nhiên có thể có khả năng là câu lệnh delete thực thi tốt hơn khi không có index như là việc hiểu được có thể có khả năng lệnh insert thực thi khi không có index tốt hơn nếu truy vấn tác động đến phần lớn của bảng.

Mẹo: Thậm chí các lệnh delete và update cũng có các bước thực thi.

Một lệnh delete không có mệnh đề where là một ví dụ rõ ràng mà database không sử dụng index, mặc dù đây là trường hợp đặc biệt mà có lệnh SQL riêng: **truncate table**.

Lệnh này có ảnh hưởng giống như lệnh delete mà không có mệnh đề where ngoại trừ rằng nó xóa tất cả các hàng trong một lần. Nó là rất nhanh nhưng có hai tác dụng phụ quan trọng: (1) nó thực hiện một cam kết ngầm (ngoại trừ: PostgreSQL); (2) nó không thực thi các **triggers**.

Tác dụng phụ của MVCC.

MVCC (Multiversion concurrency control) là điều khiển tương tranh dựa trên phiên bản- là một cơ chế cho phép truy cập tương tranh không chặn và khung nhìn giao dịch nhất quán. Tuy nhiên việc hoàn thành là khác nhau với từng database và thậm chí có thể ảnh hưởng lớn đến hiệu xuất.

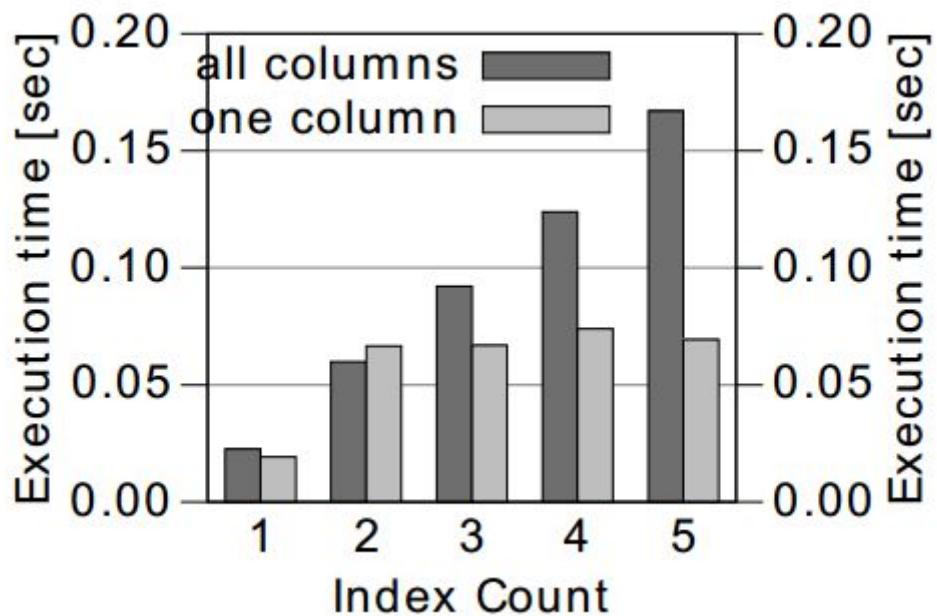
Ví dụ, PostgreSQL database (cơ sở dữ liệu quan hệ) chỉ dũ lại thông tin phiên bản (= thông tin thấy được) trên table level: xóa một hàng chỉ cần đặt cờ “delete”: trong block của bảng. Do đó hiệu xuất lệnh delete với cơ sở dữ liệu quan hệ không phụ thuộc vào số index trên một bảng. Việc xóa vật lý các hàng của bảng và liên quan tới bảo trì các chỉ mục chỉ được tiến hành trong quá trình **VACUUM**.

55. Update

Một lệnh update phải di rời các phần tử chỉ mục thay đổi để duy trì thứ tự index. Để làm điều này thì cơ sở dữ liệu phải xóa phần tử chỉ mục cũ ban đầu và thêm vào phần tử chỉ mục mới tại vị trí mới. Thời gian phản hồi cơ bản là giống như là riêng từng lệnh delete và insert cùng thực hiện vậy.

Như lệnh insert hay delete thì hiệu xuất của lệnh update cũng phụ thuộc vào số index trên bảng. Sự khác biệt duy nhất là câu lệnh update không nhất thiết phải ảnh hưởng tới tất cả các cột vì chúng chỉ sửa đổi một số cột được chọn. Kết quả là lệnh index không nhất thiết ảnh hưởng đến tất cả các index trên bảng nhưng vẫn ảnh hưởng tới các index có chứa trong các cột được cập nhật.

Hình 8.3 chỉ ra thời gian phản hồi cho hai lệnh update: một lệnh update tất cả các cột và ảnh hưởng tới tất cả các index, một lệnh chỉ update một cột và chỉ ảnh hưởng tới một index.



Hình 8.3: Hiệu xuất lệnh update theo các chỉ mục và số lượng cột.

Lệnh update trên tất cả các cột cho thấy cùng một pattern ta đã quan sát được trong các phần trước: thời gian phản ứng tăng lên với mỗi một index được thêm, thời gian phản hồi với update trên một cột không tăng nhiều bởi hầu hết các chỉ mục không phải cập nhật.

Để tối ưu hóa hiệu xuất của lệnh update, bạn chỉ cần lưu tâm tới những cột bị cập nhật. Điều này là hiển nhiên nếu chạy các câu lệnh cập nhật thủ công. Tuy nhiên nếu sử dụng công cụ ORM thì câu lệnh UPDATE có thể được tạo ra luôn luôn cập nhật tất cả các cột. Hibernate là một ví dụ, luôn cập nhật tất cả các cột nếu tắt tùy chọn dynamic-update mode. Từ phiên bản 4.0, chế độ này được bật mặc định.

Khi sử dụng công cụ ORM, một thói quen tốt là luôn luôn bật chế độ ghi nhật ký truy vấn trong môi trường phát triển để có thể xem xét các câu truy vấn được sinh ra tự động. Mẹo “Enabling SQL Logging” ở trang 95 có một giới thiệu ngắn về cách kích hoạt tính năng SQL logging trong một số công cụ ORM được sử dụng rộng rãi.

Câu hỏi đặt ra là: Bạn có thể nghĩ ra trường hợp nào mà câu lệnh insert hoặc delete không ảnh hưởng tới tất cả các index của một bảng.