Tài liệu PostgreSQL 9.0.13

Nhóm phát triển toàn cầu PostgreSQL

I. Sách chỉ dẫn

&

II. Ngôn ngữ SQL

Dịch sang tiếng Việt: Lê Trung Nghĩa, letrungnghia.foss@gmail.com

Dịch xong: 13/03/2014

Bản gốc tiếng Anh:

http://www.postgresql.org/files/documentation/pdf/9.0/postgresql-9.0-A4.pdf

---

PostgreSQL 9.0.13 Documentation

The PostgreSQL Global Development Group

I. Tutorial

&

II. The SQL Language

Lời người dịch

Tài liệu PostgreSQL 9.0.13 của nhóm phát triển toàn cầu PostgreSQL xuất bản năm 2013, như trong phần LỜI NÓI ĐẦU ở bên dưới giới thiệu, gồm 7 phần chính, đánh số từ I tới VII và phần dành cho các phụ lục, được đánh số VIII, tổng cộng dài 2.364 trang.

Bản dịch lần này gồm 2 phần đầu đã được dịch xong trước và đưa ra để các độc giả sử dụng sớm, bao gồm:

- Phần I: Sách chỉ dẫn, là một giới thiệu không chính thức cho những người mới sử dụng.
- Phần II: Ngôn ngữ SQL, viết về môi trường ngôn ngữ truy vấn SQL, bao gồm cả các dạng và các hàm dữ liệu, cũng như việc tinh chỉnh hiệu năng ở mức của người sử dụng. Mọi người sử dụng PostgreSQL đều nên đọc phần này.

Chính vì tài liệu không được dịch xong hoàn toàn tất cả các phần cùng một lúc, nên trong quá trình sử dụng, có một số nội dung tham chiếu ở các Phần I và Phần II tới các phần còn lại của tài liệu các độc giả chỉ có thể xem nội dung ở bản gốc tiếng Anh (như theo đường dẫn ở bìa trước của tài liệu) mà chưa có phần dịch tiếng Việt. Rất mong được các độc giả thông cảm.

Hy vọng các phần tiếp sau sẽ được dịch sang tiếng Việt sớm.

Việc dịch thuật chắc chắn không khỏi có những lỗi nhất định, rất mong các bạn độc giả, một khi phát hiện được, xin liên lạc với người dịch và đóng góp ý kiến về cách chỉnh sửa để tài liệu sẽ ngày một có chất lượng tốt hơn, phục vụ cho các bạn độc giả và mọi người có nhu cầu được tốt hơn. Xin chân thành cảm ơn trước các bạn độc giả về các đóng góp chỉnh sửa đó.

Mọi thông tin đóng góp chỉnh sửa cho bản dịch tiếng Việt, xin vui lòng gửi vào địa chỉ thư điện tử:

letrungnghia.foss@gmail.com

Chúc các độc giả thành công!

Hà Nội, ngày 13/03/2014

Lê Trung Nghĩa

Mục lục

Lời nói đầu

Cuốn sách này là tài liệu chính thức của PostgreSQL. Nó đã được các lập trình viên và những người tình nguyện khác của PostgreSQL viết song song với sự phát triển của phần mềm PostgreSQL. Nó mô tả tất cả các chức năng mà phiên bản hiện hành của PostgreSQL chính thức hỗ trợ.

Để làm cho số lượng lớn các thông tin về PostgreSQL có khả năng quản lý được, cuốn sách này đã được tổ chức trong vài phần. Mỗi phần có mục đích cho từng lớp người sử dụng khác nhau, hoặc cho các giai đoạn khác nhau của người sử dụng đối với kinh nghiệm về PostgreSQL của họ:

- Phần I là một giới thiệu không chính thức cho những người mới sử dụng.
- Phần II viết về môi trường ngôn ngữ truy vấn SQL, bao gồm cả các dạng và các hàm dữ liệu, cũng như việc tinh chỉnh hiệu năng ở mức của người sử dụng. Mọi người sử dụng PostgreSQL đều nên đọc phần này.
- Phần III mô tả sự cài đặt và quản trị máy chủ. Từng người mà quản lý một máy chủ PostgreSQL, dù là để sử dụng riêng hay vì những lý do khác, nên đọc phần này.
- Phần IV mô tả các giao diện lập trình cho các chương trình máy trạm PostgreSQL.
- Phần V bao gồm các thông tin cho những người sử dụng cao cấp về các khả năng mở rộng của máy chủ. Các chủ đề bao gồm các dạng và hàm dữ liệu do người sử dụng định nghĩa.
- Phần VI bao gồm các thông tin tham chiếu về các lệnh SQL, các chương trình máy trạm và máy chủ. Phần này hỗ trợ các phần khác với các thông tin được sắp xếp theo lệnh hoặc chương trình.
- Phần VII bao gồm các thông tin hỗn hợp có thể được sử dụng cho các lập trình viên của PostgreSQL.

1. PostgreSQL là gì?

PostgreSQL là một hệ quản trị cơ sở dữ liệu đối tượng - quan hệ - ORDBMS (object-relational database management system) dựa trên POSTGRES, phiên bản 4.2[1], được Phòng Khoa học Máy tính ở Berkeley của Đại học California phát triển. POSTGRES đã đi tiên phong trong nhiều khái niệm mà chỉ trở thành sẵn sàng trong một số hệ thống cơ sở dữ liệu thương mại lâu sau này.

PostgreSQL là một hậu bối nguồn mở với mã gốc ban đầu của Berkeley. Nó hỗ trợ một phần lớn tiêu chuẩn SQL và đưa ra nhiều tính năng hiện đại:

- các truy vấn phức tạp
- các khóa ngoại
- các trigger

1 http://db.cs.berkeley.edu/postgres.html

- các kiểu nhìn
- tính toàn vẹn của giao dịch
- kiểm soát đồng thời nhiều phiên bản

Hơn nữa, PostgreSQL còn có thể được người sử dụng mở rộng theo nhiều cách thức, ví dụ bằng việc bổ sung thêm mới:

- các dạng dữ liệu
- các hàm
- các toán tử
- các hàm tổng hợp
- các phương pháp đánh chỉ số
- các ngôn ngữ thủ tục

Và vì có giấy phép tự do, PostgreSQL có thể được bất kỳ ai sử dụng, sửa đổi và phân phối một cách miễn phí vì bất kỳ mục đích gì, dù là riêng tư, thương mại hay hàn lâm.

2. Ngắn gọn về lịch sử của PostgreSQL

Hệ quản trị cơ sở dữ liệu đối tượng - quan hệ – ORDBMS mà bây giờ được biết như là PostgreSQL có xuất xứ từ gói POSTGRES được Đại học California ở Berkeley viết. Với hơn 2 thập niên phát triển đã qua, PostgreSQL bây giờ là cơ sở dữ liệu nguồn mở tiến tiến nhất sẵn sàng ở khắp nơi.

2.1. Dự án POSTGRES của Berkeley

Dự án POSTGRES, do Giáo sư Michael Stonebraker lãnh đạo, đã được Cơ quan các Dự án Nghiên cứu Tiên tiến Quốc phòng - DARPA (Defense Advanced Research Projects Agency), Văn phòng Nghiên cứu Quân sự - ARO (Army Research Office), Quỹ Khoa học Quốc gia – NSF (National Science Foundation), và công ty ESL tài trợ. Sự triển khai của POSTGRES đã bắt đầu vào năm 1986. Các khái niệm ban đầu về hệ thống này đã được trình bày trong *Thiết kế POSTGRES*, và định nghĩa mô hình dữ liệu ban đầu đã xuất hiện trong *Mô hình dữ liệu POSTGRES*. Thiết kế hệ thống các qui tắc khi đó đã được mô tả trong *Thiết kế hệ thống các qui tắc của POSTGRES*. Nhân tố cơ bản và kiến trúc của trình quản lý lưu trữ đã được chi tiết hóa trong *Thiết kế hệ thống lưu trữ của POSTGRES*.

POSTGRES đã trải qua vài phiên bản chính kể từ đó. Hệ thống "phần mềm demo – demoware" đầu tiên đã được vận hành vào năm 1987 và từng được trình bày tại Hội nghị ACM-SIGMOD 1988. Phiên bản 1, được mô tả trong *Triển khai POSTGRES*, đã được phát hành cho một ít người sử dụng bên ngoài vào tháng 06/1989. Đáp lại một chỉ trích của hệ thống các qui tắc đầu tiên (*Bình luận về hệ thống các qui tắc của POSTGRES*), hệ thống các qui tắc đã được thiết kế lại (Về các qui tắc, thủ

tục, lưu trữ tạm và kiểu nhìn trong các hệ thống cơ sở dữ liệu), và Phiên bản 2 đã được phát hành vào tháng 06/1990 với hệ thống các qui tắc mới. Phiên bản 3 đã xuất hiện vào năm 1991 và đã bổ sung thêm sự hỗ trợ cho nhiều trình quản lý lưu trữ, một trình thực thi các truy vấn được cải tiến, và một hệ thống các qui tắc được viết lại. Phần lớn, các phiên bản tiếp sau cho tới Postgres95 (xem bên dưới) đã tập trung vào tính khả chuyển và độ tin cậy.

POSTGRES từng được sử dụng để triển khai nhiều ứng dụng sản xuất và nghiên cứu khác nhau. Chúng bao gồm: một hệ thống phân tích dữ liệu tài chính, một gói giám sát hiệu năng động cơ phản lực, một cơ sở dữ liệu theo dõi thiên văn, một cơ sở dữ liệu thông tin y tế và vài hệ thống thông tin địa lý. POSTGRES cũng từng được sử dụng như một công cụ giáo dục ở vài trường đại học. Cuối cùng, Illustra Information Technologies (sau này sát nhập vào Informix[2], và bây giờ IBM sở hữu[3]) đã lấy mã đó và thương mại hóa nó. Vào cuối năm 1992, POSTGRES đã trở thành trình quản lý dữ liệu hàng đầu cho dự án tính toán khoa học Sequoia 2000[4].

Kích thước của cộng đồng người sử dụng bên ngoài gần gấp đôi vào năm 1993. Đã trở nên ngày càng rõ ràng rằng sự duy trì mã bản mẫu và sự hỗ trợ đã chiếm lượng thời gian lớn mà nên được chuyên tâm cho sự nghiên cứu cơ sở dữ liệu. Trong một nỗ lực để giảm gánh nặng hỗ trợ này, dự án POSTGRES của Berkeley đã chính thức kết thúc với phiên bản 4.2.

2.2. Postgres95

Vào năm 1994, Andrew Yu và Jolly Chen đã bổ sung một trình biên dịch ngôn ngữ SLQ vào POSTGRES. Dưới một cái tên mới, Postgres95 đã liên tục được phát hành lên web để tìm kiếm con đường riêng của nó trên thế giới như là một hậu duệ nguồn mở của mã gốc ban đầu POSTGRES của Berkeley.

Mã Postgres95 từng hoàn toàn là ANSI C và được cắt tỉa tới 25% về kích cỡ. Nhiều thay đổi nội bộ đã cải thiện hiệu năng và khả năng duy trì. Postgres95 phiên bản 1.0.x đã chạy nhanh hơn khoảng 30-50% trên Wiscosin Benchmark so với POSTGRES phiên bản 4.2. Ngoài việc sửa lỗi, những cải tiến chính sau đây đã được thực hiện:

- Ngôn ngữ truy vấn PostQUEL đã được thay thế bằng SQL (được triển khai ở máy chủ). Các truy vấn con đã không được hỗ trợ cho tới PostgreSQL (xem bên dưới), nhưng chúng có thể được mô phỏng trong Postgres95 với các hàm SQL do người sử dụng định nghĩa. Các hàm tổng hợp đã được tái triển khai lại. Sự hỗ trợ cho mệnh đề truy vấn GROUP BY cũng đã được bổ sung thêm vào.
- Một chương trình mới (psql) đã được cung cấp cho các truy vấn SQL tương tác, nó đã sử dụng GNU Readline. Điều này đã thay thế phần lớn chương trình giám sát cũ.
- Một thư viện mới cho mặt tiền giao tiếp (front end), libpgtcl, đã hỗ trợ cho các máy trạm dựa vào TCL. Một trình biên dịch shell mẫu, pgtclsh, đã cung cấp các lệnh TCL mới để các

2 http://www.informix.com/
3 http://www.ibm.com/
4 http://meteora.ucsd.edu/s2k/s2k_home.html

chương trình TCL giao diện với máy chủ Postgres95.

- Một giao diện đối tượng lớn đã được xem xét lại kỹ lưỡng. Sự đảo ngược các đối tượng lớn từng là cơ chế duy nhất cho việc lưu trữ các đối tượng lớn. (Sự đảo ngược hệ thống tệp đã bị loại bỏ).
- Một hệ thống qui tắc mức sự việc đã bị loại bỏ. Các qui tắc từng vẫn sẵn sàng như các qui tắc được viết lại.
- Một sách chỉ dẫn ngắn giới thiệu các chức năng SQL thường xuyên cũng như các chức năng của Postgre95 đã được phân phối với mã nguồn.
- GNU make (thay cho BSD make) đã được sử dụng để xây dựng. Hơn nữa, Postgres95 có thể được biên dịch với một GCC không vá víu (căn chỉnh các đúp bản dữ liệu đã được sửa).

2.3. PostgreSQL

Tới năm 1996, đã trở nên rõ ràng rằng cái tên "Postgres95" có lẽ không phù hợp về thời gian. Chúng tôi đã chọn cái tên mới, PostgreSQL, để phản ánh mối quan hệ giữa POSTGRES gốc ban đầu và các phiên bản gần đó hơn với khả năng SQL. Cùng lúc, chúng tôi đã thiết lập việc đánh số phiên bản, bắt đầu với phiên bản 6.0, đặt các con số ngược về một cách tuần tự mà dự án POSTGRES của Berkeley đã bắt đầu.

Nhiều người tiếp tục tham chiếu tới PostgreSQL như là "Postgres" (bây giờ hiếm khi với tất cả các chữ hoa) vì truyền thống hoặc vì nó là dễ phát âm hơn. Sự sử dụng này được chấp nhận rộng rãi như một tên hiệu hoặc bí danh.

Sự nhấn mạnh trong quá trình phát triển của Postgres95 từng là vào việc xác định và hiểu các vấn đề trong mã của máy chủ. Với PostgreSQL, sự nhấn mạnh đã chuyển sang việc nâng cao các tính năng và khả năng, dù công việc tiếp tục trong tất cả các khía cạnh.

Các chi tiết về những gì đã xảy ra trong PostgreSQL kể từ đó có thể thấy trong Phụ lục E.

3. Các qui ước

Cuốn sách này sử dụng các qui ước in ấn sau đây để đánh dấu các phần nhất định của văn bản: các khái niệm mới, các cụm từ ngoại, và các đoạn văn bản quan trọng được nhấn mạnh bằng *chữ nghiêng*. Mọi thứ trình bày đầu vào hoặc đầu ra của máy tính, đặc biệt các lệnh, mã chương trình, và đầu ra màn hình, được biểu diễn theo một phông chữ đơn cách (ví dụ). Trong các đoạn văn bản như vậy, các chữ nghiêng (*ví dụ*) chỉ phần giữ chỗ; bạn phải chèn một giá trị thực tế vào thay cho phần giữ chỗ đó. Nhân đây, các phần mã chương trình cũng được nhấn mạnh bằng chữ đậm (**ví dụ**), nếu chúng từng được bổ sung thêm hoặc bị thay đổi kể từ ví dụ trước đó.

Những qui ước sau sẽ được sử dụng trong bảng tóm tắt của một lệnh: các dấu ngoặc vuông ([và]) chỉ các phần tùy ý chọn. (Trong các bảng tóm tắt một lệnh TCL, dấu hỏi (?) được sử dụng để thay thế, như thông thường trong TCL). Các dấu ngoặc nhọn ({và}) và các đường thẳng đứng (|) chỉ rằng

bạn phải chọn một lựa chọn. Các dấu chấm (…) có nghĩa là yếu tố trước đó có thể được lặp lại.

Ở những nơi cần sự làm sáng tỏ, các lệnh SQL được đặt trước bằng một dấu nhắc =>, và các lệnh biên dịch shell được đặt trước với một dấu nhắc $. Dù thông thường, các dấu nhắc sẽ không được thể hiện.

Người quản trị thường là một người mà có trách nhiệm trong việc cài đặt và quản lý máy chủ. Người sử dụng có thể là bất kỳ ai đang sử dụng, hoặc muốn sử dụng, bất kỳ phần nào của hệ thống PostgreSQL. Những khái niệm đó sẽ không được dịch nghĩa quá hẹp; cuốn sách này không có các giả thiết cố định về các thủ tục quản trị hệ thống.

4. Thông tin thêm

Ngoài cuốn sách này ra, có những tài nguyên khác về PostgreSQL:

Wiki.

> Wiki[5] của PostgreSQL bao gồm danh sách các câu hỏi đáp thường gặp (FAQ[6]) của dự án, danh sách các chỉ dẫn cách làm – TODO[7], và thông tin chi tiết hóa về nhiều chủ đề hơn nữa.

Website.

> Website[8] của PostgreSQL có các chi tiết về phiên bản mới nhất và các thông tin khác để thực hiện công việc của bạn hoặc chơi với PostgreSQL một cách có năng suất hơn.

Các danh sách thư.

> Các dánh sách thư là một nơi tốt để các câu hỏi của bạn được trả lời, để chia sẻ các kinh nghiệm với những người sử dụng khác, và để liên hệ với các lập trình viên. Hãy hỏi website của PostgreSQL để có được các chi tiết.

Bản thân bạn!

> PostgreSQL là một dự án nguồn mở. Vì thế, nó phụ thuộc vào cộng đồng những người sử dụng về sự hỗ trợ liên tục. Khi bạn bắt đầu sử dụng PostgreSQL, bạn sẽ dựa vào những người khác, hoặc thông qua tài liệu hoặc các danh sách thư để có được sự hỗ trợ. Hãy cân nhắc việc đóng góp tri thức của bạn ngược trở lại. Hãy đọc các danh sách thư và các câu hỏi đáp. Nếu bạn học được thứ gì đó mà không có trong tài liệu này, hãy viết ra và đóng góp nó. Nếu bạn bổ sung thêm các tính năng cho mã, hãy đóng góp chúng.

5. Các chỉ dẫn báo cáo lỗi

Khi bạn thấy có một lỗi trong PostgreSQL, chúng tôi mong muốn được nghe về nó. Các báo cáo lỗi của bạn sẽ đóng góp một phần quan trọng trong việc làm cho PostgreSQL tin cậy hơn vì thậm chí

5 http://wiki.postgresql.org
6 http://wiki.postgresql.org/wiki/Frequently_Asked_Questions
7 http://wiki.postgresql.org/wiki/Todo
8 http://www.postgresql.org

sự chăm sóc tột bực cũng không thể đảm bảo được rằng mỗi phần của PostgreSQL sẽ làm việc được trong mọi nền tảng theo mọi hoàn cảnh được.

Những gợi ý sau đây có ý định hỗ trợ cho bạn trong việc thông báo các báo cáo lỗi mà có thể được xử trí theo một cách thức có hiệu quả. Không ai bị yêu cầu phải tuân thủ chúng, nhưng làm như vậy có xu hướng tận dụng được ưu thế của mọi người.

Chúng tôi không thể hứa sẽ sửa được mọi lỗi ngay lập tức. Nếu lỗi đó là rõ ràng, có tính sống còn, hoặc ảnh hưởng tới nhiều người sử dụng, thì những cơ hội là tốt để ai đó sẽ xem xét nó. Cũng có thể xảy ra rằng chúng tôi nói cho bạn để cập nhật một phiên bản mới để xem liệu lỗi đó có xảy ra ở đó hay không. Hoặc chúng tôi có thể quyết định rằng lỗi đó không thể sửa được trước khi một vài sự viết lại chính yếu mà chúng tôi lên kế hoạch được thực hiện. Hoặc có thể đơn giản là quá khó và có nhiều điều quan trọng hơn trong chương trình nghị sự. Nếu bạn cần sự giúp đỡ ngay lập tức, hãy xem xét để có được một hợp đồng hỗ trợ thương mại.

5.1. Xác định các lỗi

Trước khi bạn báo cáo một lỗi, xin hãy đọc và đọc lại tài liệu này để kiểm tra xem bạn thực sự có thể làm bất kỳ điều gì mà bạn đang cố gắng hay không. Nếu còn chưa rõ từ tài liệu, liệu bạn có thể làm thứ gì đó hay không, xin cũng hãy báo cáo điều đó; đây là một lỗi trong tài liệu. Nếu hóa ra là một chương trình làm thứ gì đó khác với những gì tài liệu nói, thì đó là một lỗi. Điều đó có thể bao gồm, nhưng không bị giới hạn, tới các hoàn cảnh sau:

- Một chương trình kết thúc với một dấu hiệu chí tử hoặc một thông điệp lỗi hệ điều hành mà có thể chỉ tới một vấn đề trong chương trình. (Một phản ví dụ có thể là một thông điệp "đĩa đầy", khi bạn phải tự sửa điều đó).
- Một chương trình tạo ra đầu ra sai với bất kỳ đầu vào nào.
- Một chương trình từ chối chấp nhận đầu vào hợp lệ (như được xác định trong tài liệu).
- Một chương trình chấp nhận đầu vào không hợp lệ mà không có thông điệp lưu ý hoặc báo lỗi. Nhưng hãy nhớ trong đầu rằng ý tưởng của bạn về đầu vào không hợp lệ có thể là ý tưởng của chúng tôi về một mở rộng hoặc sự tương thích với thực tiễn theo truyền thống.
- PostgreSQL không biên dịch, xây dựng, hoặc cài đặt phù hợp với các lệnh trong các nền tảng được hỗ trợ.

Ở đây "chương trình" tham chiếu tới bất kỳ sự thực thi nào, chứ không chỉ cho máy chủ ở phần phụ trợ (backend).

Chạy chậm hoặc thiếu tài nguyên không phải là một lỗi. Hãy đọc tài liệu hoặc hỏi trên một trong các danh sách thư để được trợ giúp trong việc tinh chỉnh các ứng dụng của bạn. Không tuân thủ được với tiêu chuẩn SQL cũng không nhất thiết là một lỗi, trừ phi sự tuân thủ đối với tính năng đặc biết đó được nêu một cách rõ ràng.

Trước khi bạn tiếp tục, hãy kiểm tra danh sách TODO (hãy làm) và trong các câu hỏi đáp thường gặp - FAQ để xem liệu lỗi của bạn đã được biết rồi hay chưa. Nếu bạn không thể giải mã được thông tin trên danh sách TODO, hãy nêu vấn đề của bạn. Điều ít nhất chúng tôi có thể là làm cho danh sách TODO được rõ ràng hơn.

5.2. Báo cáo cái gì

Điều quan trọng nhất phải nhớ về việc báo cáo lỗi là nói lên tất cả các sự việc và chỉ các sự việc. Không đoán những gì bạn nghĩ sẽ là sai, những gì "nó dường như làm", hoặc phần nào của chương trình có một lỗi. Nếu bạn không quen với sự triển khai, thì bạn có thể đoán sai và không giúp được chúng tôi tí gì. Và thậm chí nếu bạn làm đúng, thì những giải thích có học là một sự bổ sung lớn chứ không phải là sự thay thế được cho các sự việc. Nếu chúng tôi định sẽ sửa lỗi đó thì chúng tôi sẽ phải xem nó xảy ra đối với bản thân chúng tôi trước. Việc báo cáo các sự việc trần trụi là khá thẳng thắn (bạn có thể sao chép và dán chúng từ màn hình) nhưng tất cả các chi tiết thường rất quan trọng sẽ bị đặt ra ngoài vì ai đó nghĩ nó không là vấn đề hoặc báo cáo có thể hiểu được bằng bất kỳ cách gì. Những điều sau đây nên có trong từng báo cáo lỗi:

- Tuần tự các bước chính xác từ khởi tạo chương trình, cần thiết để tái hiện lại vấn đề. Điều này nên là khép kín; không đủ để đưa vào một lệnh SELECT trần trụi mà không có các lệnh ở trước như CREATE TABLE và INSERT, nếu đầu ra phụ thuộc vào dữ liệu trong các bảng. Chúng tôi không có thời gian để thực hiện kỹ thuật nghịch đảo sơ đồ cơ sở dữ liệu của bạn, và nếu chúng tôi xử trí để tạo ra các dữ liệu riêng của chúng tôi thì chúng tôi có lẽ sẽ bỏ qua vấn đề đó.

  Định dạng tốt nhất cho một vụ kiểm thử các vấn đề có liên quan tới SQL là một tệp có thể được chạy qua mặt tiền psql mà chỉ ra được vấn đề đó. (Hãy chắc chắn không có bất kỳ thứ gì trong tệp khởi tạo ~/.psqlrc của bạn). Một cách dễ dàng để tạo tệp này là sử dụng pg_dump để đổ ra các khai báo bảng và các dữ liệu cần thiết để thiết lập kịch bản, sau đó bổ sung truy vấn của vấn đề. Bạn được khuyến khích để tối thiểu hóa kích cỡ ví dụ của bạn, nhưng điều này là không cần thiết một cách tuyệt đối. Nếu lỗi có khả năng tái tạo được, thì chúng tôi sẽ tìm được cách để làm thế.

  Nếu ứng dụng của bạn sử dụng một số giao diện máy trạm khác, như PHP, thì xin hãy cố cách li các truy vấn gây ra lỗi. Chúng tôi có thể sẽ không thiết lập một máy chủ web để tái tạo vấn đề của bạn. Trong mọi trường hợp, hãy ghi nhớ cung cấp các tệp đầu vào chính xác; không đoán rằng vấn đề xảy ra vì "các tệp lớn" hoặc "các cơ sở dữ liệu có các kích cỡ sai", … vì thông tin này là quá không chính xác để sử dụng.

- Đầu ra bạn có. Xin hãy không nói rằng nó "đã không làm việc" hoặc "đã bị hỏng". Nếu có một thông điệp lỗi, hãy chỉ nó ra, thậm chí nếu bạn không hiểu nó. Nếu chương trình kết thúc với một lỗi của hệ điều hành, hãy nói lỗi gì. Nếu không có gì xảy ra cả, hãy nói thế. Thậm chí nếu kết quả của vụ kiểm thử của bạn là một sự hỏng chương trình hoặc, nếu khác, rõ ràng nó có thể không xảy ra trong nền tảng của bạn. Điều dễ nhất là sao chép đầu ra từ

cửa số dòng lệnh – terminal, nếu có thể.

> **Lưu ý**: Nếu bạn đang báo cáo một thông điệp lỗi, xin hãy có mẫu dài dòng nhất của thông điệp đó. Trong psql, hãy nói sự dài dòng của \set VERBOSITY trước. Nếu bạn đang trích ra thông điệp từ lưu ký (log) của máy chủ, hãy đặt tham số thời gian chạy log_error_verbosity để nói dài sao cho tất cả các chi tiết sẽ được lưu ký lại.
>
> **Lưu ý**: Trong trường hợp có các lỗi chí tử, thông điệp lỗi được máy trạm nêu có lẽ không bao gồm tất cả các thông tin có sẵn. Xin cũng nhìn vào các kết quả đầu ra lưu ký của máy chủ cơ sở dữ liệu. Nếu bạn không giữ đầu ra lưu ý máy chủ của bạn, thì đây có thể là thời điểm tốt để bắt đầu làm thế.

- Kết quả đầu ra mà bạn mong đợi rất quan trọng để nêu. Nếu bạn chỉ viết "Lệnh này cho tôi kết quả đầu ra đó" hoặc "Đây không phải là điều tôi mong đợi", thì chúng tôi có thể tự làm nó, quét kết quả đầu ra, và nghĩ nó trông OK và chính xác là những gì chúng tôi đã mong đợi. Chúng tôi sẽ không mất thời gian để giải mã ngữ nghĩa chính xác đằng sau các lệnh của bạn. Đặc biệt hãy kiềm chế việc chỉ để nói rằng "Đây không phải là những gì SQL nói/Oracle làm". Việc đào sâu hành vi đúng từ SQL không phải là một việc định làm cho vui, cũng không phải chúng tôi biết tất cả cách tất cả các cơ sở dữ liệu quan hệ khác hành xử. (Nếu vấn đề của bạn là hỏng chương trình, thì bạn có thể rõ ràng quên đi khoản này).

- Bất kỳ lựa chọn dòng lệnh nào và các lựa chọn khởi tạo khác, bao gồm bất kỳ biến môi trường phù hợp nào hoặc các tệp cấu hình nào mà bạn đã thay đổi so với mặc định. Một lần nữa, xin cung cấp thông tin chính xác. Nếu bạn đang sử dụng một phân phối trước khi được đóng gói mà khởi động máy chủ cơ sở dữ liệu lúc khởi động, thì bạn nên cố tìm ra cách mà điều đó đã được thực hiện.

- Bất kỳ điều gì bạn đã làm trực tiếp từ các chỉ dẫn cài đặt.

- Phiên bản PostgreSQL. Bạn có thể chạy lệnh chọn phiên bản - SELECT version (); để tìm ra phiên bản của máy chủ mà bạn được kết nối tới. Hầu hết các chương trình thực thi cũng hỗ trợ một - lựa chọn phiên bản; ít nhất là postgres -- version và psql -- version sẽ làm việc. Nếu chức năng hoặc các lựa chọn này không tồn tại thì phiên bản của bạn đủ lớn hơn phiên bản cũ để đảm bảo một sự nâng cấp. Nếu bạn chạy một phiên bản trước khi được đóng gói, như RPM, giả sử thế, bao gồm bất kỳ phiên bản con nào mà gói đó có thể có. Nếu bạn đang nói về một hình chụp của Git, hãy nêu nó, bao gồm cả hàm băm đệ trình.

  Nếu phiên bản của bạn cũ hơn 9.0.13 thì chúng tôi hầu như chắc chắn sẽ nói bạn để nâng cấp. Có nhiều sửa lỗi và cải tiến trong từng phiên bản mới, nên hoàn toàn có khả năng là một lỗi bạn đã gặp trong một phiên bản cũ hơn của PostgreSQL đã được sửa rồi. Chúng tôi chỉ có thể cung cấp sự hỗ trợ có giới hạn cho các site sử dụng các phiên bản cũ hơn của PostgreSQL; nếu bạn yêu cầu nhiều hơn so với chúng tôi có thể cung cấp, hãy xem xét để có được một hợp đồng hỗ trợ thương mại.

- Thông tin nền tảng. Điều này bao gồm tên và phiên bản nhân, thư viện C, trình xử lý, thông

tin bộ nhớ, … Trong hầu hết các trường hợp là đủ để báo cáo về nhà cung cấp và phiên bản, nhưng không giả thiết bất kỳ ai cũng biết chính xác những gì "Debian" bao gồm hoặc từng người chạy trong i386s. Nếu bạn có các vấn đề về cài đặt thì thông tin về chuỗi công cụ trong máy của bạn (trình biên dịch, make, …) cũng là cần thiết.

Không sợ nếu báo cáo lỗi của bạn trở nên dài hơn. Đó là thực tế cuộc sống. Là tốt hơn để báo cáo mọi điều lần đầu tiên hơn là việc phải bóp lại các sự việc thoát khỏi bạn. Mặt khác, nếu các tệp đầu vào của bạn là khổng lồ, là hợp lý để yêu cầu trước hết liệu có ai đó có quan tâm trong việc nhìn vào đó hay không. Đây là một bài viết[9] mà phác thảo một số mẹo về báo cáo các lỗi.

Không bỏ ra tất cả thời gian của bạn để chỉ ra những thay đổi nào ở đầu vào làm cho vấn đề biến mất. Điều này có thể sẽ không giúp giải quyết được vấn đề. Nếu hóa ra là lỗi không thể sửa được ngay lập tức, thì bạn vẫn sẽ có thời gian để tìm và chia sẻ sự khắc phục của bạn. Hơn nữa, một lần nữa, không bỏ phí thời gian để đoán vì sao lỗi đó tồn tại. Chúng tôi sẽ tìm ra điều đó đủ sớm.

Khi viết một báo cáo lỗi, xin hãy tránh nhầm lẫn thuật ngữ. Gói phần mềm trong toàn bộ được gọi là "PostgreSQL", đôi khi ngắn gọn là "Postgres". Nếu bạn nói một cách đặc biệt về máy chủ phần phụ trợ (backend), hãy nêu điều đó, không chỉ nói "những hỏng hóc của PostgreSQL". Một sự hỏng hóc của một qui trình máy chủ phần phụ trợ là hoàn toàn khác với sự hỏng hóc của qui trình cha của "Postgres"; xin đừng nói "máy chủ bị hỏng" khi bạn ngụ ý một qui trình phần phụ trợ duy nhất không làm việc được, và cũng không ngược lại. Hơn nữa, các chương trình máy trạm như "psql" mặt tiền (front-end) tương tác là hoàn toàn tách biệt với phần phụ trợ. Xin hãy cố gắng phân biệt liệu vấn đề đó là ở phía máy trạm hay phía máy chủ.

5.3, Báo cáo các lỗi ở đâu

Nói chung, hãy gửi các báo cáo lỗi vào danh sách thư báo cáo lỗi tại <pgsql-bugs@postgresql.org>. Bạn được yêu cầu sử dụng một chủ đề mô tả cho thông điệp thư điện tử của bạn, có lẽ các phần của thông điệp lỗi.

Một phương pháp khác là điền vào mẫu có sẵn trên web báo cáo lỗi ở website của dự án[10]. Đưa vào báo cáo lỗi theo cách này sẽ làm cho nó được gửi bằng thư điện tử cho danh sách thư <pgsql-bugs@postgresql.org>.

Nếu báo cáo lỗi của bạn có những liên quan về an ninh và bạn thích điều đó không trở thành ngay lập tức được nhìn thấy trong các kho lưu trữ công khai, hãy đừng gửi nó cho pgsql-bugs. Các vấn đề về an ninh có thể được báo cáo riêng cho <security@postgresql.org>.

Không gửi các báo cáo lỗi tới bất kỳ danh sách thư nào của người sử dụng, như <pgsql-sql@postgresql.org> hoặc <pgsql-general@postgresql.org>. Các danh sách thư đó là dành cho việc trả lời các câu hỏi của người sử dụng, và những người đăng ký của chúng thường không muốn nhận các báo cáo lỗi. Quan trọng hơn, họ có lẽ sẽ không sửa được chúng.

9 http://www.chiark.greenend.org.uk/~sgtatham/bugs.html
10 http://www.postgresql.org

Hơn nữa, xin không gửi các báo cáo lỗi cho danh sách thư của các lập trình viên <pgsql-hackers@postgresql.org>. Danh sách này là để thảo luận về sự phát triển của PostgreSQL, và có thể là tốt nếu chúng tôi có thể giữ các báo cáo lỗi được tách biệt. Chúng tôi có thể chọn để tiến hành thảo luận về báo cáo lỗi của bạn trong pgsql-hackers, nếu vấn đề đó cần rà soát lại tiếp.

Nếu bạn có vấn đề với tài liệu, thì nơi tốt nhất để báo cáo nó là danh sách thư cho tài liệu <pgsql-docs@postgresql.org>. Xin chỉ ra phần nào của tài liệu mà bạn không thích.

Nếu lỗi của bạn là một vấn đề về tính khả chuyển trong một nền tảng không được hỗ trợ, hãy gửi thư điện tử tới <pgsql-hackers@postgresql.org>, sao cho chúng tôi (và bạn) có thể làm việc về việc chuyển PostgreSQL tới nền tảng của bạn.

> **Lưu ý**: Vì không may số lượng spam lớn đang xảy ra hiện nay, tất cả các địa chỉ thư điện tử ở trên đều là các danh sách thư đóng. Nghĩa là, bạn cần phải đăng ký vào một danh sách để được phép viết bài trong đó. (Tuy nhiên, bạn không cần phải đăng ký sử dụng mẫu báo cáo lỗi trên web). Nếu bạn muốn gửi thư nhưng không muốn nhận các thư của danh sách, thì bạn có thể đăng ký và thiết lập lựa chọn đăng ký của bạn thành nomail. Để có thêm thông tin, hãy gửi thư tới <majordomo@postgresql.org> với từ duy nhất help (giúp) ở thân của thông điệp.

I. Sách chỉ dẫn

Chào mừng bạn tới với Sách chỉ dẫn PostgreSQL. Các chương sau đây có ý định đưa ra một giới thiệu đơn giản về PostgreSQL, các khái niệm về cơ sở dữ liệu quan hệ, và ngôn ngữ SQL cho những ai còn mới đối với bất kỳ một trong những khía cạnh đó. Chúng tôi chỉ có một số tri thức chung về cách sử dụng các máy tính. Không có bất kỳ kinh nghiệm lập trình nào hay kinh nghiệm Unix cụ thể nào được yêu cầu. Phần này chủ yếu có ý định đưa ra cho bạn một số kinh nghiệm truyền tay với các khía cạnh quan trọng của hệ thống PostgreSQL. Nó không có ý định sẽ là sự xử lý đầy đủ hoặc tỉ mỉ các chủ đề mà nó bao trùm.

Sau khi bạn đã làm việc qua với sách chỉ dẫn này, bạn có thể muốn chuyển tới đọc Phần II để có được nhiều tri thức chính thống hơn về ngôn ngữ SQL, hoặc Phần IV để có thông tin về việc phát triển các ứng dụng cho PostgreSQL. Những người mà muốn thiết lập và quản trị máy chủ của riêng họ cũng nên đọc Phần III.

Chương 1. Làm quen

1.1. Cài đặt

Trước khi bạn có thể sử dụng PostgreSQL, bạn cần cài đặt nó, tất nhiên. Có khả năng là PostgreSQL được cài đặt rồi trên site của bạn, hoặc vì nó đã được đưa vào trong phát tán hệ điều hành của bạn rồi hoặc vì người quản trị hệ thống đã cài đặt nó rồi. Nếu đúng là như vậy, thì bạn nên có thông tin từ tài liệu hệ điều hành hoặc người quản trị hệ thống của bạn về cách để truy cập PostgreSQL.

Nếu bạn không chắc liệu PostgreSQL có sẵn sàng rồi hay liệu bạn có thể sử dụng nó cho thí nghiệm của bạn hay không thì tự bạn có thể cài đặt nó, rồi sau đó tham chiếu tới Chương 15 để có các chỉ dẫn về cài đặt, và quay lại với sách chỉ dẫn này khi việc cài đặt đã hoàn tất. Hãy chắc chắn tuân theo một cách sát sao phần về thiết lập các biến môi trường một cách thích hợp.

Nếu người quản trị site của bạn còn chưa thiết lập mọi điều theo cách thức mặc định, thì bạn có thể có một số công việc nữa phải làm. Ví dụ, nếu máy tính làm máy chủ cơ sở dữ liệu là một máy ở xa, thì bạn sẽ cần thiết lập biến môi trường PGHOST về tên của máy làm máy chủ cơ sở dữ liệu. Biến môi trường PGPORT cũng có thể phải được thiết lập. Vấn đề là thế này: nếu bạn cố gắng khởi động một chương trình ứng dụng và nó kêu rằng nó không thể kết nối được tới cơ sở dữ liệu, thì bạn nên hỏi người quản trị site của bạn hoặc, nếu đó là bạn, hãy tra cứu tài liệu để chắc chắn rằng môi trường của bạn được thiết lập một cách phù hợp. Nếu bạn đã không hiểu đoạn trước thì hãy đọc phần tiếp sau.

1.2. Cơ bản về kiến trúc

Trước khi chúng tôi xử lý, bạn nên hiểu kiến trúc cơ bản của hệ thống PostgreSQL. Việc hiểu cách mà các phần của PostgreSQL tương tác sẽ làm cho chương này rõ hơn một chút.

Trong biệt ngữ về cơ sở dữ liệu, PostgreSQL sử dụng mô hình máy trạm/máy chủ phục vụ. Một phiên làm việc của PostgreSQL bao gồm các tiến trình (các chương trình) kết hợp sau đây:

- Một tiến trình máy chủ, nó quản lý các tệp cơ sở dữ liệu, chấp nhận các kết nối tới cơ sở dữ liệu từ các ứng dụng máy trạm, và thực hiện các hành động của cơ sở dữ liệu nhân danh các máy trạm. Chương trình của máy chủ cơ sở dữ liệu được gọi là postgres.
- Ứng dụng máy trạm (giao diện mặt tiền – frontend) của người sử dụng mà muốn thực hiện các hoạt động cơ sở dữ liệu. Các ứng dụng máy trạm có thể rất đa dạng về bản chất tự nhiên: một máy trạm có thể là một công cụ hướng văn bản, một ứng dụng đồ họa, một máy chủ web mà truy cập cơ sở dữ liệu để hiển thị các trang web, hoặc một công cụ duy trì cơ sở dữ liệu được chuyên môn hóa. Một số ứng dụng máy trạm được cung cấp với phân phối PostgreSQL; hầu hết do những người sử dụng phát triển.

Như là điển hình đối với các ứng dụng máy trạm/máy chủ, máy trạm và máy chủ có thể nằm ở các

nơi đặt chỗ (host) khác nhau. Trong trường hợp đó chúng giao tiếp qua một kết nối mạng TCP/IP. Bạn nên nhớ điều này trong đầu, vì các tệp có thể được truy cập trong một máy tính trạm có thể không truy cập được (hoặc có thể chỉ truy cập được bằng việc sử dụng một tên tệp khác) trên máy tính chủ cơ sở dữ liệu.

Máy chủ PostgreSQL có thể xử trí nhiều kết nối đồng thời từ các máy trạm. Để đạt được điều này thì nó khởi động ("rẽ nhánh") một tiến trình mới cho từng kết nối. Từ thời điểm đó trở đi, máy trạm và tiến trình của máy chủ giao tiếp mà không có sự can thiệp của tiến trình postgres gốc ban đầu. Vì thế, tiến trình chủ đạo của máy chủ luôn chạy, chờ các kết nối của máy trạm, trong khi máy trạm và các tiến trình máy chủ có liên quan tới việc đến và đi. (Tất cả điều này tất nhiên là không nhìn thấy đối với người sử dụng. Chúng tôi chỉ nêu nó ở đây cho đầy đủ thôi).

1.3. Tạo một cơ sở dữ liệu

Bài tập đầu tiên để xem liệu bạn có thể truy cập máy chủ cơ sở dữ liệu hay không là thử tạo ra một cơ sở dữ liệu. Một máy chủ PostgreSQL đang chạy có thể quản lý nhiều cơ sở dữ liệu. Thông thường, một cơ sở dữ liệu riêng biệt được sử dụng cho từng dự án hoặc cho từng người sử dụng.

Có khả năng, người quản trị site của bạn đã tạo rồi một cơ sở dữ liệu để bạn sử dụng. Anh ta sẽ phải nói cho bạn tên của cơ sở dữ liệu đó là gì. Trong trường hợp đó bạn có thể bỏ qua bước này và nhảy tiếp sang phần tiếp sau.

Để tạo một cơ sở dữ liệu mới, trong ví dụ này tên là mydb, bạn sử dụng lệnh sau:

```
$ createdb mydb
```

Nếu điều này không đưa ra câu trả lời nào thì bước này đã thành công và bạn có thể bỏ qua phần còn lại của phần này.

Nếu bạn thấy một thông điệp tương tự như:

```
createdb: command not found
```

thì PostgreSQL đã được cài đặt không đúng. Hoặc nó đã không được cài đặt hoàn toàn, hoặc đường dẫn tìm kiếm trình biên dịch shell của bạn đã không được thiết lập để đưa nó vào. Hãy thay vào đó bằng việc cố gọi lệnh đó với một đường dẫn tuyệt đối:

```
$ /usr/local/pgsql/bin/createdb mydb
```

Đường dẫn ở site của bạn có thể là khác. Hãy liên hệ với người quản trị site của bạn hoặc kiểm tra các chỉ dẫn cài đặt để sửa lại tình trạng đó.

Một câu trả lời khác có thể như thế này:

```
createdb: could not connect to database postgres: could not connect to server: No such file Is the server
running locally and accepting connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

Điều này có nghĩa là máy chủ đã không được khởi động, hoặc nó đã không được khởi động ở nơi mà createdb đã mong đợi nó.

Một lần nữa, hãy kiểm tra các chỉ dẫn cài đặt hoặc hỏi người quản trị hệ thống.

Một câu trả lời khác có thể là thế này:

```
createdb: could not connect to database postgres: FATAL: role "joe" does not exist
```

trong đó tên đăng nhập của riêng bạn được nhắc tới. Điều này sẽ xảy ra nếu người quản trị đã không tạo ra một tài khoản PostgreSQL cho bạn. (Các tài khoản của người sử dụng PostgreSQL là khác biệt đối với các tài khoản người sử dụng của hệ điều hành). Nếu bạn là người quản trị, hãy xem Chương 20 để có sự trợ giúp cho việc tạo các tài khoản. Bạn sẽ cần trở thành người sử dụng của hệ điều hành theo đó PostgreSQL đã được cài đặt (thường là postgres) để tạo tài khoản người sử dụng đầu tiên. Cũng có thể bạn đã được chỉ định một cái tên người sử dụng PostgreSQL mà là khác với tên người sử dụng của hệ điều hành của bạn; trong trường hợp đó bạn cần sử dụng -U để chuyển hoặc thiết lập biến môi trường PGUSER để chỉ định tên người sử dụng PostgreSQL của bạn.

Nếu bạn có một tài khoản người sử dụng nhưng nó không có các quyền được yêu cầu để tạo một cơ sở dữ liệu, thì bạn hãy xem điều sau đây:

```
createdb: database creation failed: ERROR: permission denied to create database
```

Không phải mọi người sử dụng đều có quyền để tạo các cơ sở dữ liệu mới. Nếu PostgreSQL từ chối tạo các cơ sở dữ liệu đối với bạn, thì người quản trị site đó cần phải trao các quyền cho bạn để tạo các cơ sở dữ liệu. Hãy hỏi người quản trị site nếu điều này xảy ra. Nếu bạn đã tự cài PostgreSQL thì bạn nên đăng nhập vào vì các mục đích của sách chỉ dẫn này tuân theo tài khoản người sử dụng mà bạn đã khởi động máy chủ[1].

Bạn cũng có thể tạo các cơ sở dữ liệu với các tên khác. PostgreSQL cho phép bạn tạo bất kỳ số lượng nào các cơ sở dữ liệu ở một site. Các tên cơ sở dữ liệu phải có ký tự abc ở đầu tiên và bị giới hạn tới 63 bytes chiều dài. Một lựa chọn thuận tiện là tạo một cơ sở dữ liệu với cùng tên như tên người sử dụng hiện hành của bạn. Nhiều công cụ có tên cơ sở dữ liệu đó như là mặc định, nên nó có thể tiết kiệm cho bạn việc gõ chữ. Để tạo cơ sở dữ liệu, đơn giản gõ vào:

```
$ createdb
```

Nếu bạn không muốn sử dụng cơ sở dữ liệu của bạn hơn nữa thì bạn có thể loại bỏ nó. Ví dụ, nếu bạn là người chủ (người tạo ra) cơ sở dữ liệu mydb, thì bạn có thể hủy diệt nó bằng việc sử dụng lệnh sau:

```
$ dropdb mydb
```

(Đối với lệnh này, tên cơ sở dữ liệu không là mặc định đối với tên tài khoản của người sử dụng. Bạn luôn cần phải chỉ định nó). Hành động này về vật lý loại bỏ tất cả các tệp có liên quan tới cơ sở dữ liệu và không thể hoãn lệnh được, vì thế điều này chỉ nên được thực hiện với một suy nghĩ trước một cách hết sức thận trọng.

1 Như một sự giải thích vì sao điều này làm việc: Các tên người sử dụng của PostgreSQL là độc lập với tài khoản của người sử dụng hệ điều hành. Khi bạn kết nối tới một cơ sở dữ liệu, bạn có thể chọn tên người sử dụng PostgreSQL nào để kết nối; nếu bạn không muốn, nó sẽ mặc định tên y hệt như tài khoản hệ điều hành hiện hành của bạn. Khi điều này xảy ra, sẽ luôn có một tài khoản người sử dụng PostgreSQL mà có cùng tên như người sử dụng hệ điều hành đã khởi động máy chủ, và điều này cũng xảy ra rằng người sử dụng đó luôn có quyền tạo các cơ sở dữ liệu. Thay vì việc đăng nhập như người sử dụng đó, bạn cũng có thể chỉ định lựa chọn -U ở bất kỳ ở đâu để chọn một tên người sử dụng PostgreSQL để kết nối.

Nhiều thông tin hơn về createdb và dropdb có thể thấy ở createdb và dropdb một cách tương ứng.

1.4. Truy cập cơ sở dữ liệu

Một khi bạn đã tạo ra một cơ sở dữ liệu, bạn có thể truy cập nó bằng:

- Chạy một chương trình đầu cuối (terminal) tương tác, gọi là psql, cho phép bạn làm việc một cách tương tác với các lệnh SQL như vào, sửa và thực thi.
- Sử dụng một công cụ đồ họa mặt tiền như pgAdmin hoặc một bộ phần mềm văn phòng với sự hỗ trợ của ODBC hoặc JDBC để tạo và điều khiển một cơ sở dữ liệu. Các khả năng đó sẽ không được đề cập tới trong sách chỉ dẫn này.
- Viết một ứng dụng tùy ý, có sử dụng một trong vài ràng buộc ngôn ngữ có sẵn. Các khả năng đó được thảo luận xa hơn trong Phần IV.

Bạn có thể muốn khởi tạo psql để thử các ví dụ trong sách chỉ dẫn này. Nó có thể được kích hoạt cho cơ sở dữ liệu mydb bằng việc gõ lệnh:

```
$ psql mydb
```

Nếu bạn không cung cấp tên cơ sở dữ liệu thì nó sẽ mặc định sử dụng tên tài khoản người sử dụng của bạn. Bạn đã phát hiện rồi sơ đồ này trong phần trước bằng việc sử dụng createdb.

Trong psql, bạn sẽ được chào đón với thông điệp sau:

```
psql (9.0.13)
```

Hãy gõ “help” để xin trợ giúp.

```
mydb=>
```

Điều đó có thể có nghĩa là bạn là siêu người dùng (superuser) của một cơ sở dữ liệu, nó giống hệt trường hợp nếu bạn đã tự cài đặt PostgreSQL. Là siêu người sử dụng có nghĩa là bạn không phải tuân thủ các kiểm soát truy cập. Vì các mục đích của sách chỉ dẫn này, điều đó không quan trọng.

Nếu bạn gặp phải các vấn đề khởi động psql thì hãy quay ngược về phần trước. Các dự đoán của createdb và psql là tương tự nhau, và nếu cái trước đã làm việc được thì cái sau cũng sẽ làm việc.

Dòng cuối cùng được in ra từ psql là dấu nhắc, và nó chỉ ra rằng psql đang nghe bạn và rằng bạn có thể gõ các truy vấn SQL vào một không gian làm việc được psql duy trì. Hãy thử các lệnh đó:

```
mydb=> SELECT version();

version
----------------------------------------------------------------------------
PostgreSQL 9.0.13 on i586-pc-linux-gnu, compiled by GCC 2.96, 32-bit
(1 row)

mydb=> SELECT current_date;

date
------------
2002-08-31
(1 row)
```

```
mydb=> SELECT 2 + 2;

?column?
----------
4
(1 row)
```

Chương trình psql có một số lệnh nội bộ không phải là các lệnh SQL. Chúng bắt đầu với ký tự chéo ngược, "\". Ví dụ, bạn có thể có được sự trợ giúp trong cú pháp của các lệnh SQL khác nhau của PostgreSQL bằng việc gõ:

```
mydb=> \h
```

Để ra khỏi psql, hãy gõ:

```
mydb=> \q
```

và psql sẽ thoát ra và trả bạn về với trình biên dịch lệnh shell của bạn. (Để biết thêm về các lệnh nội bộ, hãy gõ \? ở dấu nhắc psql). Các khả năng đầy đủ của psql được viết thành tài liệu trong psql. Nếu PostgreSQL được cài đặt đúng thì bạn cũng có thể gõ `man psql` ở dấu nhắc shell của hệ điều hành để thấy tài liệu đó. Trong sách chỉ dẫn này chúng tôi sẽ không sử dụng các tính năng đó một cách rõ ràng, nhưng tự bạn có thể sử dụng chúng khi cần thiết.

Chương 2. Ngôn ngữ SQL

2.1 Giới thiệu

Chương này đưa ra tổng quan về cách sử dụng SQL để tiến hành các hoạt động đơn giản. Sách chỉ dẫn này chỉ có ý định giới thiệu cho bạn chứ không phải là một chỉ dẫn SQL hoàn chỉnh. Nhiều cuốn sách từng được viết về SQL, bao gồm *Hiểu SQL mới và Chỉ dẫn về Tiêu chuẩn SQL*. Bạn nên nhận thức được là một số tính năng của ngôn ngữ SQL là các mở rộng cho tiêu chuẩn đó.

Trong các ví dụ bên dưới, chúng tôi giả thiết là bạn đã tạo ra một cơ sở dữ liệu có tên là mydb, như được mô tả trong chương trước, và từng có khả năng để khởi tạo psql.

Các ví dụ trong sách chỉ dẫn này cũng có thể được thấy trong phân phối nguồn của PostgreSQL trong thư mục src/tutorial/. (Các phân phối nhị phân của PostgreSQL có thể không biên dịch các tệp đó). Để sử dụng các tệp đó, trước hết hãy thay đổi tới thư mục đó và chạy lệnh make:

```
$ cd ..../src/tutorial
$ make
```

Điều này tạo ra các script và biên dịch các tệp C có chứa các hàm và các dạng do người sử dụng định nghĩa. Sau đó, để khởi động sách chỉ dẫn, hãy làm như sau:

```
$ cd ..../tutorial
$ psql -s mydb
...
mydb=> \i basics.sql
```

Lệnh \i đọc trong các lệnh từ tệp được chỉ định. Lựa chọn psql đặt bạn vào chế độ một bước duy nhất và tạm ngừng trước khi gửi từng lệnh tới máy chủ. Các lệnh được sử dụng trong phần này là trong tệp basics.sql.

2.2. Các khái niệm

PostgreSQL là một *hệ quản trị cơ sở dữ liệu quan hệ* (RDBMS). Điều đó có nghĩa nó là một hệ thống cho việc quản lý các dữ liệu được lưu trữ theo *các quan hệ*. Mối quan hệ đó, về cơ bản, là khái niệm toán học cho *bảng*. Khái niệm của việc lưu trữ các dữ liệu trong các bảng là rất phổ biến ngày nay, có thể dường như là vốn dĩ rõ ràng vậy, nhưng có một số cách thức khác trong việc tổ chức các cơ sở dữ liệu. Các tệp và các thư mục trong các hệ điều hành giống Unix tạo nên ví dụ về một cơ sở dữ liệu có tôn ti trật tự. Một sự phát triển hiện đại hơn là *cơ sở dữ liệu hướng đối tượng*.

Mỗi bảng là một tập hợp được đặt tên của các *hàng*. Mỗi hàng của một bảng được đưa ra có cùng tập hợp các *cột* được đặt tên, và từng cột có dạng dữ liệu đặc thù. Trong khi các cột có một trật tự cố định theo từng hàng, thì điều quan trọng phải nhớ là SQL không đảm bảo trật tự của các hàng bên trong bảng theo bất kỳ cách gì (dù chúng có thể được sắp xếp một cách rõ ràng để hiển thị).

Các bảng được nhóm lại trong các cơ sở dữ liệu, và một bộ sưu tập các cơ sở dữ liệu được một cài đặt máy chủ PostgreSQL duy nhất quản lý tạo thành một *cụm* cơ sở dữ liệu.

2.3. Tạo một bảng mới

Bạn có thể tạo một bảng mới bằng việc chỉ định tên bảng, cùng với tất cả các tên cột và các dạng của chúng:

```
CREATE TABLE weather (
        city            varchar(80),
        temp_lo         int,           -- low temperature
        temp_hi         int,           -- high temperature
        prcp            real,          -- precipitation
        date date
);
```

Bạn có thể đưa điều này vào trong psql với các ngắt dòng. Psql sẽ nhận thức được rằng lệnh đó không bị dừng cho tới khi có dấu chấm phẩy.

Dấu trống (như, các dấu trống, các tab và các dòng mới) có thể được sử dụng một cách tự do trong các lệnh SQL. Điều đó có nghĩa là bạn có thể gõ lệnh được dóng hàng khác nhau so với ở trên, hoặc thậm chí tất cả chỉ trong một dòng. Hai dấu gạch ngang ("--") để giới thiệu các ghi chú bình luận. Bất kỳ thứ gì đi sau chúng sẽ bị bỏ qua cho tới kết thúc dòng đó. SQL phân biệt giữa chữ hoa và chữ thường đối với các từ khóa và các từ nhận dạng, ngoại trừ khi các từ nhận dạng được đặt trong các dấu ngoặc kép để giữ lại các chữ thường hoặc chữ hoa đó (không được thực hiện ở trên).

Varchar (80) chỉ định dạng dữ liệu có thể lưu trữ tùy ý các chuỗi ký tự cho tới 80 ký tự về độ dài. int là dạng số nguyên thông thường. real là một dạng để lưu trữ các số có dấu chấm động với độ chính xác duy nhất. date là để tự giải thích. (Vâng, cột dạng date cũng có tên là date. Điều này có thể là thuận tiện hoặc khó hiểu - tùy bạn chọn).

PostgreSQL hỗ trợ các dạng SQL tiêu chuẩn int, smallint, real, double precision, char(N), varchar(N), date, time, timestamp, và interval, cũng như các dạng tiện ích thông thường khác và một tập hợp giàu có các dạng địa lý. PostgreSQL có thể được tùy biến với một số lượng tùy ý các dạng dữ liệu do người sử dụng định nghĩa. Hệ quả là, các tên dạng không phải là các từ khóa trong cú pháp, ngoại trừ ở những nơi được yêu cầu để hỗ trợ các trường hợp đặc biệt trong tiêu chuẩn SQL.

Ví dụ thứ 2 sẽ lưu trữ các thành phố và vị trí địa lý có liên quan của chúng:

```
CREATE TABLE cities (
        name            varchar(80),
        location        point
);
```

Dạng point là một ví dụ về dạng dữ liệu đặc biệt của PostgreSQL.

Cuối cùng, được lưu ý rằng nếu bạn không cần một bảng nào nữa hoặc muốn tạo lại nó theo cách khác thì bạn có thể xóa nó bằng việc sử dụng lệnh:

```
DROP TABLE tablename;
```

2.4. Đưa dữ liệu vào bảng với các hàng

Lệnh chèn INSERT được sử dụng để đưa dữ liệu vào một bảng với các hàng:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Lưu ý rằng tất cả các dạng dữ liệu sử dụng khá rõ ràng các định dạng đầu vào. Các hằng mà không phải là các giá trị số đơn giản thường phải được đưa vào trong các dấu nháy đơn ('), như trong ví dụ trên. Dạng date thường khá là mềm dẻo theo những gì nó chấp nhận, nhưng đối với sách chỉ dẫn này thì chúng tôi sẽ gắn định dạng không mơ hồ được chỉ ra ở đây.

Dạng point yêu cầu một cặp tọa độ như là đầu vào, như được chỉ ra ở đây:

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

Cú pháp được sử dụng cho tới nay đòi hỏi bạn nhớ trật tự các cột. Một lựa chọn cú pháp khác cho phép bạn liệt kê các cột một cách rõ ràng:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

Bạn có thể liệt kê các cột theo một trật tự khác nếu bạn muốn hoặc thậm chí bỏ sót một số cột, nghĩa là, nếu sự hấp tấp chưa được nhận ra:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Nhiều lập trình viên coi việc liệt kê rõ ràng các cột là dạng tốt hơn so với việc dựa vào trật tự không rõ ràng.

Hãy vào tất cả các lệnh được chỉ ra ở trên sao cho bạn có một số dữ liệu để làm việc trong các phần tiếp sau.

Bạn cũng có thể đã sử dụng lệnh sao chép COPY để tải lượng dữ liệu lớn từ các tệp văn bản thô. Điều này thường xuyên là nhanh hơn vì lệnh COPY được tối ưu hóa cho ứng dụng này trong khi cho phép tính mềm dẻo ít hơn so với lệnh INSERT. Một ví dụ:

```
COPY weather FROM '/home/user/weather.txt';
```

trong đó tên tệp cho tệp nguồn phải là sẵn sàng đối với máy tính làm máy chủ phần phụ trợ (backend), không phải máy trạm, vì máy chủ phần phụ trợ đọc tệp đó một cách trực tiếp. Bạn có thể đọc nhiều hơn về lệnh COPY trong COPY.

2.5. Truy vấn bảng

Để truy xuất dữ liệu từ một bảng, bảng đó được truy vấn. Một lệnh chọn SELECT của SQL được sử dụng để làm điều này. Lệnh này được chia thành một danh sách chọn (phần liệt kê các cột sẽ được trả về), một danh sách bảng (phần liệt kê các bảng từ đó sẽ truy xuất các dữ liệu), và phần định tính tùy chọn (phần chỉ định bất kỳ giới hạn nào). Ví dụ, để trích xuất tất cả các hàng của bảng thời tiết, hãy gõ:

```
SELECT * FROM weather;
```

Ở đây * là viết tắt cho "tất cả các cột"[1]. Vì thế kết quả y hệt có thể có với:

1 Trong khi SELECT * là hữu dụng cho các truy vấn ít được chuẩn bị trước, thì nó được xem một cách rộng rãi như là dạng tồi trong việc tạo mã, vì việc bổ sung một cột vào bảng có thể làm thay đổi các kết quả.

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

Đầu ra sẽ là:

```
city          | temp_lo | temp_hi | prcp | date
--------------+---------+---------+------+------------
San Francisco |      46 |      50 | 0.25 | 1994-11-27
San Francisco |      43 |      57 |    0 | 1994-11-29
Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

Bạn có thể viết các biểu thức, không chỉ các tham chiếu cột đơn giản, trong danh sách chọn. Ví dụ, bạn có thể làm:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

Điều này sẽ đưa ra:

```
city          | temp_avg | date
--------------+----------+------------
San Francisco |       48 | 1994-11-27
San Francisco |       50 | 1994-11-29
Hayward       |       45 | 1994-11-29
(3 rows)
```

Lưu ý cách mà mệnh đề AS được sử dụng để gắn lại nhãn cho cột đầu ra. (Mệnh đề AS là tùy chọn).

Một truy vấn có thể "có hạn chế" bằng việc bổ sung thêm một mệnh đề nơi mà - WHERE chỉ định các hàng nào sẽ được mong muốn. Mệnh đề WHERE chứa một biểu thức Boolean (giá trị đúng), và chỉ các hàng mà biểu thức Boolean đó là đúng mới được trả về. Các toán tử Boolean thường dùng (AND, OR, và NOT) sẽ được phép trong phần định tính.

Ví dụ, thứ sau đây trích xuất thời tiết của San Francisco vào những ngày mưa:

```
SELECT * FROM weather
        WHERE city = 'San Francisco' AND prcp > 0.0;
```

Kết quả:

```
city          | temp_lo | temp_hi | prcp | date
--------------+---------+---------+------+------------
San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

Bạn có thể yêu cầu các kết quả của một truy vấn được trả về theo trật tự sắp xếp:

```
SELECT * FROM weather
        ORDER BY city;

city          | temp_lo | temp_hi | prcp | date
--------------+---------+---------+------+------------
Hayward       |      37 |      54 |      | 1994-11-29
San Francisco |      43 |      57 |    0 | 1994-11-29
San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

Trong ví dụ này, trật tự sắp xếp được chỉ định không đầy đủ, và vì thế bạn có thể có các hàng San Francisco theo cả 2 trật tự. Nhưng bạn có thể luôn có được các kết quả như ở trên nếu bạn làm:

```
SELECT * FROM weather
        ORDER BY city, temp_lo;
```

Bạn có thể yêu cầu các hàng đúp bản bị loại bỏ khỏi kết quả của một truy vấn:

```
SELECT DISTINCT city
        FROM weather;

city
---------------
Hayward
San Francisco
(2 rows)
```

Một lần nữa ở đây, trật tự các hàng kết quả có thể là khác nhau. Bạn có thể đảm bảo các kết quả nhất quán bằng việc sử dụng cùng một lúc cả DISTINCT và ORDER BY[2]:

```
SELECT DISTINCT city
        FROM weather
        ORDER BY city;
```

2.6. Liên kết giữa các bảng

Cho tới nay, các truy vấn của chúng ta đã chỉ truy cập tới 1 bảng ở một thời điểm. Các truy vấn có thể truy cập nhiều bảng cùng một lúc, hoặc truy cập bảng y hệt theo một cách thức mà nhiều hàng của bảng đang được xử lý cùng một lúc. Một truy vấn truy cập được nhiều hàng của cùng hoặc các bảng khác nhau cùng một lúc được gọi là một truy vấn liên kết. Ví dụ, bạn muốn liệt kê tất cả các bản ghi về thời tiết cùng với địa điểm của thành phố có liên quan. Để làm thế, bạn cần so sánh cột thành phố (city) của từng hàng trong bảng thời tiết (weather) với cột tên (name) của tất cả các hàng trong bảng các thành phố (cities), và chọn các cặp các hàng nơi mà các giá trị đó khớp nhau.

> **Lưu ý**: Đây chỉ là một mô hình khái niệm. Sự liên kết thường được thực hiện theo một cách thức có hiệu quả hơn so với việc thực sự so sánh cặp các hàng có khả năng, mà điều này là không nhìn thấy đối với người sử dụng.

Điều này có thể được thực hiện bằng truy vấn sau:

```
SELECT *
        FROM weather, cities
        WHERE city = name;

city          | temp_lo | temp_hi | prcp | date       | name          | location
--------------+---------+---------+------+------------+---------------+-----------
San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

Hãy quan sát 2 điều về tập các kết quả:

- Không có hàng kết quả nào cho thành phố Hayward. Điều này là vì không có khoản đầu vào nào khớp trong bảng các thành phố cho Hayward, nên sự liên kết bỏ qua các hàng không khớp nhau trong bảng thời tiết. Chúng ta sẽ thấy ngay cách mà điều này có thể được sửa.
- Có 2 cột chứa tên thành phố đó. Điều này là đúng vì các danh sách các cột từ các bảng thời

2 Trong một số hệ thống cơ sở dữ liệu, bao gồm cả các phiên bản cũ của PostgreSQL, sự triển khai của DISTINCT sẽ tự động sắp xếp các hàng và vì thế ORDER BY là không cần thiết. Nhưng điều này không được tiêu chuẩn SQL yêu cầu, và PostgreSQL hiện hành không đảm bảo rằng DISTINCT làm cho các hàng được sắp xếp theo trật tự được.

tiết và các thành phố được ghép lại với nhau. Trong thực tế điều này là không mong muốn, nên bạn có thể sẽ muốn liệt kê các cột đầu ra một cách rõ ràng hơn là việc sử dụng dấu *:

```
SELECT city, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;
```

Bài tập: Cố gắng xác định ngữ nghĩa của truy vấn này khi mệnh đề WHERE bị bỏ qua. Vì các cột tất cả đã có các tên khác nhau, nên trình phân tích cú pháp tự động thấy được bảng nào chúng thuộc về.

Nếu có các tên cột trùng nhau trong 2 bảng đó thì bạn cần phải định tính các tên cột để chỉ ra cột nào bạn ngụ ý, như trong:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
        FROM weather, cities
        WHERE cities.name = weather.city;
```

Được thừa nhận rộng rãi cách thức tốt để định tính tất cả các tên cột trong truy vấn liên kết, sao cho truy vấn đó sẽ không hỏng nếu một tên cột bị đúp bản sau này được thêm vào một trong các bảng.

Các truy vấn liên kết dạng đó tới nay vì thế được xem là cũng có thể được viết ở dạng lựa chọn thay thế này:

```
SELECT *
        FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

Cú pháp này không được sử dụng phổ biến như cú pháp ở trên, nhưng chúng tôi chỉ ra nó ở đây để giúp bạn hiểu các chủ đề tiếp sau.

Bây giờ chúng ta sẽ chỉ ra cách mà chúng ta có thể có các bản ghi có Hayward được đưa ngược vào. Những gì chúng ta muốn truy vấn đó thực hiện là quét bảng thời tiết và đối với mỗi hàng để tìm (các) hàng trùng khớp của bảng các thành phố (cities). Nếu không có hàng nào khớp được tìm thấy thì chúng ta muốn một số "giá trị rỗng" sẽ được thay thế cho các cột của bảng các thành phố. Dạng truy vấn này được gọi là *liên kết vòng ngoài*. (Các liên kết mà chúng ta đã từng thấy tới nay là các liên kết vòng trong). Lệnh đó trông giống thế này:

```
SELECT *
        FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);
```

```
city          | temp_lo | temp_hi | prcp | date       | name          | location
--------------+---------+---------+------+------------+---------------+----------
Hayward       |      37 |      54 |      | 1994-11-29 |               |
San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

Truy vấn này được gọi là *liên kết vòng ngoài bên trái* vì bảng được nhắc tới ở bên trái của toán tử liên kế sẽ có từng trong số các hàng của nó ở đầu ra ít nhất một lần, trong khi bảng ở bên phải sẽ chỉ có các hàng ở đầu ra mà khớp với một số hàng của bảng ở bên trái. Khi một hàng bảng ở bên trái đầu ra không có sự trùng khớp với bảng ở bên phải, thì các giá trị rỗng (null) sẽ được thay thế cho các cột của bảng ở bên phải.

Bài tập: Cũng có các liên kết vòng ngoài bên phải và các liên kết vòng ngoài đầy đủ. Hãy cố tìm ra

xem chúng làm được gì.

Chúng ta cũng có thể liên kết một bảng với chính nó. Điều này được gọi là *tự liên kết*. Ví dụ, giả sử chúng ta muốn thấy tất cả các bản ghi về thời tiết nằm trong dải nhiệt độ các bản ghi của bảng thời tiết khác. Vì thế chúng ta cần so sánh các cột temp_lo và temp_hi của từng bảng thời tiết với các cột temp_lo và temp_hi với tất cả các hàng khác của bảng thời tiết. Chúng ta có thể làm điều này bằng truy vấn sau:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
        W2.city, W2.temp_lo AS low, W2.temp_hi AS high
        FROM weather W1, weather W2
        WHERE W1.temp_lo < W2.temp_lo
        AND W1.temp_hi > W2.temp_hi;

city          | low | high | city          | low | high
--------------+-----+------+---------------+-----+------
San Francisco |  43 |   57 | San Francisco |  46 | 50
Hayward       |  37 |   54 | San Francisco |  46 | 50
(2 rows)
```

Ở đây chúng ta đã gắn lại nhãn cho bảng thời tiết như là W1 và W2 để có khả năng phân biệt được phía bên trái và bên phải của liên kết. Bạn cũng có thể sử dụng các dạng tên hiệu (aliase) đó trong các truy vấn khác để tiết kiệm việc gõ, như:

```
SELECT *
        FROM weather w, cities c
        WHERE w.city = c.name;
```

Bạn sẽ gặp dạng viết tắt này thường xuyên.

2.7. Các hàng tổng hợp

Giống như hầu hết các sản phẩm cơ sở dữ liệu quan hệ khác, PostgreSQL hỗ trợ các hàm tổng hợp. Một hàm tổng hợp tính toán một kết quả duy nhất từ nhiều hàng đầu vào. Ví dụ, có các tổng hợp để tính toán như count, sum, avg (average - trung bình), max (maximum - tối đa) và min (minimum - tối thiểu) đối với một tập hợp các hàng.

Ví dụ, chúng ta có thể tìm nhiệt độ thấp cao nhất ở đâu đó với:

```
SELECT max(temp_lo) FROM weather;

max
-----
46
(1 row)
```

Nếu chúng ta muốn biết thành phố (hoặc các thành phố nào đó) xảy ra điều trên, thì ta có thể thử:

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo); WRONG
```

nhưng điều này sẽ không làm việc vì tổng hợp max không thể được sử dụng trong mệnh đề WHERE. (Hạn chế này tồn tại vì mệnh đề WHERE xác định các hàng nào sẽ được đưa vào trong tính toán tổng hợp; nên rõ ràng nó phải được đánh giá trước khi các hàm tổng hợp được tính toán). Tuy nhiên, thường thì là trường hợp truy vấn có thể được tuyên bố lại để hoàn tất kết quả mong muốn, ở đây là

bằng việc sử dụng một *truy vấn con*:

```
SELECT city FROM weather
        WHERE temp_lo = (SELECT max(temp_lo) FROM weather);

city
---------------
San Francisco
(1 row)
```

Điều này là OK vì truy vấn con là một tính toán độc lập, nó tính tổng hợp của riêng nó một cách tách bạch với những gì đang xảy ra trong truy vấn vòng ngoài.

Các tổng hợp cũng rất hữu dụng trong sự kết hợp với các mệnh đề GROUP BY. Ví dụ, chúng ta có thể có được nhiệt độ thấp nhất được quan sát thấy trong từng thành phố với:

```
SELECT city, max(temp_lo)
        FROM weather
        GROUP BY city;

city                | max
--------------------+--------
Hayward             | 37
San Francisco       | 46
(2 rows)
```

mà nó trao cho chúng ta một hàng đầu ra cho mỗi thành phố. Mỗi kết quả tổng hợp được tính toán đối với các hàng của bảng khớp với thành phố đó. Chúng ta có thể lọc các hàng được nhóm lại đó bằng việc sử dụng HAVING:

```
SELECT city, max(temp_lo)
        FROM weather
        GROUP BY city
        HAVING max(temp_lo) < 40;

city                | max
--------------------+--------
Hayward             | 37
(1 row)
```

nó trao cho chúng ta các kết quả y hệt chỉ cho các thành phố có tất cả các giá trị temp_lo thấp hơn 40. Cuối cùng, nếu chúng ta chỉ quan tâm về các thành phố mà các tên bắt đầu với ký tự “S”, thì chúng ta có thể làm:

```
SELECT city, max(temp_lo)
        FROM weather
        WHERE city LIKE 'S%' ¶
        GROUP BY city
HAVING max(temp_lo) < 40;
```

¶ Toán tử LIKE thực hiện việc khớp mẫu và được giải thích trong phần 9.7.

Điều quan trọng phải hiểu sự tương tác giữa các tổng hợp và các mệnh đề WHERE và HAVING của SQL. Sự khác biệt cơ bản giữa WHERE và HAVING là điều này: WHERE chọn các hàng đầu vào trước khi các nhóm và các tổng hợp được tính toán (vì thế, nó kiểm soát các hàng nào đi vào trong tính toán tổng hợp đó), trong khi HAVING lựa chọn các hàng của nhóm trước khi các nhóm và các tổng hợp được tính toán. Vì thế mệnh đề WHERE phải không bao gồm các hàm tổng hợp; không có ý

nghĩa để thử sử dụng một tổng hợp để xác định các hàng nào sẽ là các đầu vào đối với các tổng hợp đó. Mặt khác, mệnh đề HAVING luôn bao gồm các hàm tổng hợp. (Nói một cách chặt chẽ, bạn được phép viết một mệnh đề HAVING mà không sử dụng các tổng hợp, nhưng điều này hiếm khi hữu dụng. Điều kiện tương tự có thể được sử dụng có hiệu quả hơn ở giai đoạn của WHERE).

Trong ví dụ trước, chúng ta có thể áp dụng sự hạn chế tên của thành phố trong WHERE, vì nó không cần tổng hợp. Điều này có hiệu quả hơn so với việc thêm hạn chế vào HAVING, vì chúng ta tránh thực hiện các tính toán tổng hợp và việc tạo nhóm đối với tất cả các hàng mà không kiểm tra được với WHERE.

2.8. Cập nhật

Bạn có thể cập nhật các hàng đang tồn tại bằng việc sử dụng lệnh cập nhật – UPDATE. Giả sử bạn phát hiện ra việc đọc nhiệt độ tất cả là lệch 2 độ sau ngày 28/11. Bạn có thể sửa các dữ liệu như sau:

```
UPDATE weather
    SET temp_hi = temp_hi - 2, temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

Hãy xem tình trạng mới của dữ liệu:

```
SELECT * FROM weather;
```

```
city          | temp_lo | temp_hi | prcp | date
--------------+---------+---------+------+------------
San Francisco |      46 |      50 | 0.25 | 1994-11-27
San Francisco |      41 |      55 |    0 | 1994-11-29
Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. Xóa

Các hàng có thể bị xóa khỏi một bảng bằng việc sử dụng lệnh xóa – DELETE. Giả sử bạn không còn quan tâm tới thời tiết của Hayward nữa. Sau đó bạn có thể làm điều sau đây để xóa các hàng đó khỏi bảng:

```
DELETE FROM weather WHERE city = 'Hayward';
```

Tất cả các bản ghi thời tiết thuộc về Hayward sẽ bị loại bỏ.

```
SELECT * FROM weather;
```

```
city          | temp_lo | temp_hi | prcp | date
--------------+---------+---------+------+------------
San Francisco |      46 |      50 | 0.25 | 1994-11-27
San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

Nên thận trọng đối với các tuyên bố dạng

```
DELETE FROM tablename;
```

Không có một sự thận trọng, DELETE sẽ xóa tất cả các hàng khỏi bảng được đưa ra đó, làm cho hàng sẽ rỗng. Hệ thống sẽ không hỏi khẳng định trước khi làm điều này!

Chương 3. Các tính năng cao cấp

3.1. Giới thiệu

Trong chương trước chúng ta đã đề cập tới những điều cơ bản của việc sử dụng SQL để lưu trữ và truy cập các dữ liệu của bạn trong PostgreSQL. Chúng ta bây giờ sẽ thảo luận một số tính năng cao cấp hơn của SQL mà đơn giản hóa quản lý và ngăn chặn mất mát hoặc hỏng các dữ liệu của chúng ta. Cuối cùng, chúng ta sẽ xem một số mở rộng của PostgreSQL.

Chương này sẽ nhân cơ hội tham chiếu tới các ví dụ được thấy trong Chương 2 để thay đổi hoặc cải tiến chúng, sao cho nó sẽ là hữu dụng để đọc chương đó. Một số ví dụ từ chương này cũng có thể được thấy trong tệp advanced.sql trong thư mục của sách chỉ dẫn. Tệp này cũng chứa một số dữ liệu mẫu để tải lên, nó sẽ không được lặp lại ở đây. (Tham chiếu tới Phần 2.1 về cách sử dụng tệp đó).

3.2. Các kiểu nhìn

Tham chiếu ngược về các truy vấn trong Phần 2.6. Giả thiết việc liệt kê kết hợp các bản ghi thời tiết và địa điểm của thành phố là sự quan tâm đặc biệt cho ứng dụng của bạn, nhưng bạn không muốn gõ truy vấn đó vào mỗi lần bạn cần nó. Bạn có thể tạo một kiểu nhìn (view) đối với truy vấn đó, nó trao một cái tên cho truy vấn mà bạn có thể tham chiếu tới như một bảng thông thường:

```
CREATE VIEW myview AS
        SELECT city, temp_lo, temp_hi, prcp, date, location
                FROM weather, cities
                WHERE city = name;
SELECT * FROM myview;
```

Tạo sự thông thoáng để sử dụng các kiểu nhìn là một khía cạnh chính của thiết kế tốt cơ sở dữ liệu SQL. Các kiểu nhìn cho phép bạn đóng gói các chi tiết cấu trúc các bảng của bạn, nó có thể thay đổi khi ứng dụng của bạn tiến hóa, đằng sau những giao diện nhất quán.

Các kiểu nhìn có thể được sử dụng trong hầu hết bất kỳ chỗ nào một bảng thực tế có thể được sử dụng. Việc xây dựng các kiểu nhìn dựa vào các kiểu nhìn khác không phải là phổ biến.

3.3. Các khóa ngoại

Nhớ lại các bảng thời tiết và các thành phố từ Chương 2. Xem xét vấn đề sau: Bạn muốn chắc chắn rằng không ai có thể chèn các hàng vào bảng thời tiết mà không có một khoản đầu vào khớp trong bảng các thành phố. Điều này được gọi là việc duy trì *tính toàn vẹn tham chiếu* các dữ liệu của bạn. Trong các hệ thống cơ sở dữ liệu giản dị thì điều này có thể được triển khai (nếu ở tất cả) bằng việc trước hết nhìn vào bảng các thành phố để kiểm tra xem liệu một bản ghi trùng khớp có tồn tại hay không, và sau đó chèn hoặc từ chối các bản ghi mới của bảng thời tiết. Tiếp cận này có một số vấn đề và là rất thuận tiện, nên PostgreSQL có thể làm điều này cho bạn.

Khai báo mới về các bảng có thể trông giống thế này:

```
CREATE TABLE cities (
        city varchar(80) primary key,
        location point
);


CREATE TABLE weather (
        city varchar(80) references cities(city),
        temp_lo int,
        temp_hi int,
        prcp real,
        date date
);
```

Bây giờ cố gắng chèn một bản ghi hợp lệ vào:

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
ERROR: insert or update on table "weather" violates foreign key constraint "weather_city_DETAIL: Key
(city)=(Berkeley) is not present in table "cities".
```

Hành vi của các khóa ngoại có thể cuối cùng được tinh chỉnh cho ứng dụng của bạn. Chúng ta sẽ không đi vượt ra khỏi ví dụ đơn giản này trong sách chỉ dẫn này, mà chỉ tham chiếu tới Chương 5 để có thêm thông tin. Việc làm cho sử dụng đúng các khóa ngoại chắc chắn sẽ cải thiện chất lượng các ứng dụng cơ sở dữ liệu của bạn, nên bạn được khuyến khích mạnh mẽ học về chúng.

3.4. Các giao dịch

Các giao dịch là một khái niệm cơ bản của tất cả các hệ thống cơ sở dữ liệu. Điểm cơ bản của một giao dịch là nó tập hợp nhiều bước trong một bước duy nhất, một hoạt động hoặc tất cả hoặc không có gì xảy ra. Các tình trạng ngay lập tức giữa các bước là không nhìn thấy đối với các giao dịch hiện hành khác, và nếu một số hỏng hóc xảy ra mà ngăn cản giao dịch đó hoàn tất, thì không có bước nào ảnh hưởng tới cơ sở dữ liệu cả.

Ví dụ, hãy cân nhắc một cơ sở dữ liệu trắng mà bao gồm bảng quyết toán cân bằng thu chi cho các tài khoản khác nhau của người sử dụng, cũng như tổng cân bằng tiền gửi đối với các chi nhánh. Giả sử là chúng ta muốn ghi lại thanh toán của 100.00 USD từ tài khoản của Alice cho tài khoản của Bob. Quá đơn giản, các lệnh SQL cho việc này có thể là:

```
UPDATE accounts SET balance = balance - 100.00
        WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
        WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
        WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
        WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

Các chi tiết của các lệnh đó là không quan trọng ở đây; điều quan trọng là có vài bản cập nhật riêng rẽ có liên quan để hoàn tất điều này hơn là hoạt động đơn giản. Các nhân viên ngân hàng của chúng ta sẽ muốn được đảm bảo rằng tất cả các bản cập nhật đó hoặc xảy ra, hoặc không điều gì trong số chúng xảy ra cả. Có lẽ chắc chắn không làm vì một sự hỏng hóc hệ thống gây ra trong việc Bob nhận 100.00 USD sẽ không được ghi nợ từ Alice. Alice có lẽ không thấy mình là một khách hàng hạnh phúc nếu cô ta đã được ghi nợ mà Bob không nhận được. Chúng ta cần một sự đảm bảo rằng

nếu thứ gì đó sai giữa đường đối với hoạt động đó, thì không có bước nào được thực hiện cho tới lúc đó sẽ có hiệu lực cả. Việc nhóm các bản cập nhật vào một giao dịch trao cho chúng ta sự đảm bảo này. Một giao dịch được nói sẽ là một hạt nhân: từ quan điểm của các giao dịch khác, nó hoặc xảy ra hoàn chỉnh hoặc hoàn toàn không xảy ra.

Chúng ta cũng muốn một sự đảm bảo rằng một khi một giao dịch được hoàn tất và được hệ thống cơ sở dữ liệu thừa nhận, thì nó quả thực được ghi lại vĩnh viễn và sẽ không bị mất thậm chí nếu một sự hỏng hóc xảy ra sau đó ngay lập tức. Ví dụ, nếu chúng ta đang ghi một sự rút tiền của Bob, thì chúng ta không muốn bất kỳ tình huống nào mà sự ghi nợ cho tài khoản của anh ta sẽ biến mất vì một sự hỏng hóc ngay sau khi anh ta đi ra khỏi cửa ngân hàng. Cơ sở dữ liệu của một giao dịch đảm bảo rằng tất cả các bản cập nhật được một giao dịch thực hiện bị khóa trong lưu trữ vĩnh cửu (như, trên đĩa) trước khi giao dịch đó được nói là hoàn tất.

Một đặc tính quan trọng khác của các cơ sở dữ liệu giao dịch có liên quan mật thiết với khái niệm các bản cập nhật hạt nhân: khi nhiều giao dịch đang chạy đồng thời, mỗi giao dịch nên có khả năng thấy những thay đổi không hoàn chỉnh do những người khác thực hiện. Ví dụ, nếu một giao dịch đang bận tính tổng của các bản quyết toán của tất cả các chi nhánh, thì nó có thể không nên làm điều đó để đưa vào sự ghi nợ từ chi nhánh của Alice mà không làm sự cho nợ đối với chi nhánh của Bob, và ngược lại cũng không nên. Vì thế các giao dịch phải là hoặc tất cả - hoặc không có gì, không chỉ về các khía cạnh hiệu quả vĩnh cửu của chúng trong cơ sở dữ liệu, mà còn trong các khía cạnh về tính trực quan có thể nhìn thấy được của chúng khi chúng xảy ra. Các bản cập nhật được một giao dịch mở thực hiện cho tới nay là không nhìn thấy đối với các giao dịch khác cho tới khi giao dịch đó hoàn tất, ngay lúc đó tất cả các bản cập nhật trở nên trực quan một cách đồng thời.

Trong PostgreSQL, một giao dịch được thiết lập bằng các lệnh SQL bao quanh giao dịch đó với các lệnh bắt đầu - BEGIN và thực hiện – COMMIT. Vì thế giao dịch ngân hàng của chúng ta có lẽ thực sự trông giống như:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
        WHERE name = 'Alice';
-- etc etc
COMMIT;
```

Nếu, giữa đường của giao dịch, chúng ta quyết định chúng ta không muốn thực hiện (có thể chúng ta đã chỉ lưu ý rằng bản quyết toán của Alice là không tích cực), thì chúng ta có thể đưa ra lệnh ROLLBACK thay cho lệnh COMMIT, và tất cả các bản cập nhật của chúng ta cho tới lúc đó sẽ bị hoãn.

PostgreSQL thực sự đối xử với từng lệnh SQL như đang được thực thi bên trong một giao dịch. Nếu bạn không đưa ra lệnh BEGIN, thì từng lệnh riêng rẽ sẽ có một BEGIN và (nếu thành công) COMMIT được bao bọc xung quanh nó. Một nhóm các lệnh được BEGIN và COMMIT bao bọc xung quanh đôi khi được gọi là một khối giao dịch.

> **Lưu ý**: Một số thư viện máy trạm đưa ra các lệnh BEGIN và COMMIT một cách tự động, sao cho bạn có thể có được hiệu quả của các khối giao dịch mà không phải hỏi. Hãy kiểm tra tài liệu cho giao diện mà bạn đang sử dụng.

Có khả năng để kiểm soát các lệnh trong một giao dịch theo một cách thức có trọng tâm hơn bằng việc sử dụng các *điểm an toàn*. Các điểm an toàn cho phép bạn hủy bỏ một cách có lựa chọn các phần của giao dịch, trong khi thực hiện được phần còn lại. Sau việc xác định một điểm an toàn với SAVEPOINT, bạn có thể, nếu cần, quay trở lại tới điểm an toàn đó với lệnh ROLLBACK TO. Tất cả những thay đổi của cơ sở dữ liệu giao dịch giữa việc xác định điểm an toàn và việc quay ngược lại về nó sẽ được hủy bỏ, nhưng những thay đổi trước điểm an toàn đó sẽ được giữ lại.

Sau khi quay ngược trở lại tới một điểm an toàn, nó tiếp tục sẽ được nhận diện, sao cho bạn có thể quay ngược trở lại về nó vài lần. Ngược lại, nếu bạn chắc chắn bạn sẽ không cần quay ngược trở về một điểm an toàn đặc biệt một lần nữa, thì nó có thể được giải phóng, sao cho hệ thống có thể giải phóng một số tài nguyên. Hãy nhớ trong đầu rằng hoặc việc thoát ra hoặc quay ngược trở về một điểm an toàn sẽ tự động thoát ra khỏi tất cả các điểm an toàn mà đã từng được xác định sau nó.

Tất cả điều này đang xảy ra bên trong khối giao dịch, nên không có thứ gì là nhìn thấy được đối với các phiên khác của cơ sở dữ liệu. Khi và nếu bạn thực hiện khối giao dịch, các hành động được thực hiện trở nên nhìn thấy được như một đơn vị đối với các phiên khác, trong khi các hành động được quay ngược trở lại sẽ không bao giờ trở nên nhìn thấy được cả.

Ghi nhớ cơ sở dữ liệu trống, giả thiết chúng ta ghi nợ 100.00 USD từ tài khoản của Alice, và ghi có cho tài khoản của Bob, sẽ chỉ thấy sau này rằng chúng ta nên có tài khoản tin cậy của Wally. Chúng ta có thể làm điều này bằng việc sử dụng các điểm an toàn giống thế này:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
        WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
        WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
        WHERE name = 'Wally';
COMMIT;
```

Tất nhiên, ví dụ này là quá đơn giản, nhưng có nhiều khả năng kiểm soát trong một khối giao dịch thông qua sử dụng các điểm an toàn. Hơn nữa, ROLLBACK TO chỉ là cách để giành lại sự kiểm soát của một khối giao dịch mà đã được hệ thống đặt trong tình trạng bị hỏng vì một lỗi, ngắn gọn là quay nó ngược trở lại hoàn toàn và bắt đầu lại một lần nữa.

3.5. Hàm cửa sổ

Một *hàm cửa sổ* thực hiện một tính toán qua một tập hợp các hàng của bảng mà bằng cách nào đó có liên quan tới hàng hiện hành. Điều này có khả năng so sánh được với dạng tính toán mà có thể được thực hiện với một hàm tổng hợp. Nhưng không giống như các hàm tổng hợp thông thường, sử dụng một hàm cửa sổ không làm cho các hàng trở nên bị nhóm thành một hàng đầu ra duy nhất - các hàng vẫn giữ lại các định danh riêng biệt của chúng. Ở phía đằng sau, hàm cửa sổ đó có khả năng truy cập nhiều hơn là chỉ hàng hiện hành của kết quả truy vấn đó.

Đây là một ví dụ chỉ ra cách để so sánh từng khoản lương của nhân viên với lương trung bình trong phòng của anh hoặc chị ta:

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
depname   | empno | salary | avg
----------+-------+--------+-----------------------------------
develop   |    11 |   5200 | 5020.0000000000000000
develop   |     7 |   4200 | 5020.0000000000000000
develop   |     9 |   4500 | 5020.0000000000000000
develop   |     8 |   6000 | 5020.0000000000000000
develop   |    10 |   5200 | 5020.0000000000000000
personnel |     5 |   3500 | 3700.0000000000000000
personnel |     2 |   3900 | 3700.0000000000000000
sales     |     3 |   4800 | 4866.6666666666666667
sales     |     1 |   5000 | 4866.6666666666666667
sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

3 cột đầu ra đầu tiên tới trực tiếp từ bảng lương của nhân viên – empsalary, và có một hàng đầu ra cho từng hàng trong bảng đó. Cột thứ 4 đại diện cho lương trung bình được lấy ra từ tất cả các hàng của bảng mà có giá trị tên phòng - depname y hệt như hàng hiện hành. (Đây thực sự là hàm y hệt như hàm tổng thông thường avg, nhưng mệnh đề OVER làm cho nó được xử lý như một hàm cửa sổ và được tính toán xuyên khắp một tập hợp thích hợp các hàng).

Một lời gọi hàm cửa sổ luôn bao gồm một mệnh đề OVER đi sau tên và (các) biến của hàm cửa sổ đó. Đây là, theo cú pháp, những gì phân biệt nó với một hàm thông thường hoặc một hàm tổng hợp. Mệnh đề OVER xác định chính xác cách mà các hàng của truy vấn được chia tách cho việc xử lý của hàm cửa sổ. Danh sách PARTITION BY trong OVER chỉ định việc phân chia các hàng thành các nhóm, hoặc các phân vùng, mà chia sẻ cùng các giá trị của (các) biểu thức PARTITION BY. Đối với mỗi hàng, hàm cửa sổ được tính toán khắp các hàng mà rơi vào trong cùng phân vùng như hàng hiện hành.

Dù avg sẽ sản sinh ra cùng kết quả bất luận trật tự mà nó xử lý các hàng của phân vùng đó ra sao, thì điều này là không đúng đối với tất cả các hàm cửa sổ. Khi cần, bạn có thể kiểm soát trật tự đó bằng việc sử dụng ORDER BY trong OVER. Đây là một ví dụ:

```
SELECT depname, empno, salary, rank() OVER (PARTITION BY depname ORDER BY salary DESC) FROM
```

```
depname   | empno  | salary | rank
----------- +-------+--------+------
develop   |     8 |   6000 | 1
develop   |    10 |   5200 | 2
develop   |    11 |   5200 | 2
develop   |     9 |   4500 | 4
develop   |     7 |   4200 | 5
personnel |     2 |   3900 | 1
personnel |     5 |   3500 | 2
sales     |     1 |   5000 | 1
sales     |     4 |   4800 | 2
sales     |     3 |   4800 | 2
(10 rows)
```

Như được chỉ ra ở đây, hàm xếp hàng - rank tạo ra một hằng số bên trong phân vùng của hàng hiện hành cho từng giá trị độc nhất của ORDER BY, để được mệnh đề ORDER BY xác định. rank không cần

tham số rõ ràng, vì hành vi của nó hoàn toàn được xác định bằng mệnh đề OVER.

Các hàng mà một hàm cửa sổ xem xét là các hàng của "bảng ảo" được tạo ra từ mệnh đề FROM của truy vấn khi được các mệnh đề WHERE, GROUP BY và HAVING của nó lọc, nếu có. Ví dụ, một hàng bị loại bỏ vì nó không đáp ứng được các điều kiện của WHERE sẽ được bất kỳ hàm cửa sổ nào nhìn thấy. Một truy vấn có thể bao gồm nhiều hàm cửa sổ mà cắt lát các dữ liệu theo các cách thức khác nhau bằng những mệnh đề OVER khác nhau, nhưng tất cả chúng hành động trong cùng một bộ sưu tập các hàng được bảng ảo này xác định.

Chúng ta đã thấy rồi rằng ORDER BY có thể bị bỏ qua nếu việc sắp xếp các hàng là không quan trọng. Cũng có khả năng để bỏ qua PARTITION BY, trong trường hợp đó chỉ có một phân vùng có chứa tất cả các hàng.

Có một khái niệm quan trọng khác có liên quan tới các hàm cửa sổ: đối với từng hàng, có một tập hợp các hàng trong phân vùng của nó được gọi là khung cửa sổ của nó. Nhiều (nhưng không phải tất cả) các hàm cửa sổ hành động chỉ trong các hàng của khung cửa sổ, thay vì của toàn bộ phân vùng đó. Mặc định, nếu ORDER BY được cung cấp thì khung đó bao gồm tất cả các hàng từ đầu của phân vùng cho tới hàng hiện hành, cộng với bất kỳ hàng theo sau nào mà ngang bằng với hàng hiện hành theo mệnh đề ORDER BY. Khi ORDER BY bị bỏ qua thì khung mặc định bao gồm tất cả các hàng trong phân vùng[1]. Đây là một ví dụ sử dụng tổng:

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

```
salary  | sum
----------+-------
  5200 | 47100
  5000 | 47100
  3500 | 47100
  4800 | 47100
  3900 | 47100
  4200 | 47100
  4500 | 47100
  4800 | 47100
  6000 | 47100
  5200 | 47100
(10 rows)
```

Ở trên, khi không có ORDER BY trong mệnh đề OVER, thì khung cửa sổ là y hệt như phân vùng đó, thiếu PARTITION BY là bảng tổng thể; nói cách khác, từng tổng số được thực hiện cho toàn bộ bảng và vì thế chúng ta có kết quả y hệt cho từng hàng đầu ra. Nhưng nếu chúng ta bổ sung thêm một mệnh đề ORDER BY, thì chúng ta có các kết quả rất khác nhau:

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
salary  | sum
----------+----------
  3500 |   3500
  3900 |   7400
  4200 | 11600
  4500 | 16100
  4800 | 25700
```

1 Có những lựa chọn để xác định khung cửa sổ theo các cách khác, nhưng tài liệu chỉ dẫn này không đề cập tới chúng. Xem Phần 4.2.8 để có các chi tiết.

```
  4800 | 25700
  5000 | 30700
  5200 | 41100
  5200 | 41100
  6000 | 47100
(10 rows)
```

Ở đây tổng này được thực hiện từ lương đầu tiên (thấp nhất) cho tới hiện hành, bao gồm cả bất kỳ sự đúp bản nào của lương hiện hành (lưu ý các kết quả cho các lương bị đúp bản).

Các hàm cửa sổ được phép chỉ trong danh sách SELECT và mệnh đề ORDER BY của truy vấn đó. Chúng bị cấm ở đâu đó khác nữa, như trong các mệnh đề GROUP BY, HAVING và WHERE. Điều này là vì chúng, về logic, thực thi sau việc xử lý các mệnh đề đó. Hơn nữa, các hàm cửa sổ thực thi sau các hàm tổng hợp thông thường. Điều này có nghĩa rằng, là hợp lệ để đưa vào một lời gọi hàm tổng hợp vào trong các tham số của một hàm cửa sổ, nhưng không làm ngược lại được.

Nếu có một nhu cầu phải lọc hoặc tạo thành nhóm các hàng sau khi các tính toán cửa sổ được thực hiện, thì bạn có thể sử dụng một lựa chọn con (phụ). Ví dụ:

```
SELECT depname, empno, salary, enroll_date
FROM
        (SELECT depname, empno, salary, enroll_date,
                rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
        FROM empsalary
        ) AS ss
WHERE pos < 3;
```

Truy vấn ở trên chỉ đưa ra các hàng từ truy vấn vòng trong có rank ít hơn 3.

Khi một truy vấn có liên quan tới nhiều hàm cửa sổ, có khả năng để viết ra từng hàm với một mệnh đề OVER, nhưng điều này là đúp bản và lỗi - hỏng nếu hành vi tạo cửa sổ y hệt được mong muốn đối với vài hàm. Thay vào đó, từng hành vi tạo cửa sổ có thể được đặt tên trong một mệnh đề WINDOW và sau đó được tham chiếu trong OVER. Ví dụ:

```
SELECT sum(salary) OVER w, avg(salary) OVER w
        FROM empsalary
        WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

Chi tiết hơn về các hàm cửa sổ có thể thấy trong Phần 4.2.8, Phần 9.19, Phần 7.2.4, và trang tham chiếu của SELECT.

3.6. Sự kế thừa

Sự kế thừa là một khái niệm từ các cơ sở dữ liệu hướng đối tượng. Nó mở ra các khả năng mới thú vị của thiết kế cơ sở dữ liệu.

Hãy tạo 2 bảng. Một bảng các thành phố - cities và một bảng các thủ phủ - capitals. Một cách tự nhiên, các thủ phủ cũng là các thành phố, nên bạn muốn một số cách để trình bày các thủ phủ một cách ẩn khi bạn liệt kê tất cả các thành phố. Nếu bạn thực sự khôn ngoan thì bạn có thể sáng tạo một số sơ đồ giống thế này:

```
CREATE TABLE capitals (
        name text,
        population real,
        altitude int, -- (in ft)
        state char(2)
);

CREATE TABLE non_capitals (
        name text,
        population real,
        altitude int -- (in ft)
);

CREATE VIEW cities AS
        SELECT name, population, altitude FROM capitals
                UNION
        SELECT name, population, altitude FROM non_capitals;
```

Điều này làm việc OK với việc truy vấn, nhưng nó trở nên xấu xí khi bạn cần cập nhật vài hàng, vì một điều.

Một giải pháp tốt hơn là:

```
CREATE TABLE cities (
        name text,
        population real,
        altitude int -- (in ft)
);

CREATE TABLE capitals (
        state char(2)
) INHERITS (cities);
```

Trong trường hợp này, một hàng của bảng các thủ phủ kế thừa tất cả các cột (`name`, `population`, và `altitude`) từ bảng cha của nó, bảng các thành phố. Dạng của cột `name` là văn bản, một dạng bẩm sinh của PostgreSQL cho các chuỗi ký tự độ dài các biến. Các thủ phủ bang có một cột dôi ra, `state`, chỉ bang của chúng. Trong PostgreSQL, một bảng có thể kế thừa từ 0 hoặc nhiều hơn các bảng khác.

Ví dụ, truy vấn sau đây tìm thấy các tên của tất cả các thành phố, bao gồm cả các thủ phủ của các bang, mà nằm ở độ cao hơn 500 feet:

```
SELECT name, altitude
        FROM cities
        WHERE altitude > 500;
```

nó sẽ trả về:

```
name            | altitude
----------------+----------
Las Vegas       | 2174
Mariposa        | 1953
Madison         |  845
(3 rows)
```

Mặt khác, truy vấn sau đây tìm thấy tất cả các thành phố mà không phải là các thủ phủ bang và nằm ở độ cao 500 feet hoặc cao hơn:

```
SELECT name, altitude
        FROM ONLY cities
        WHERE altitude > 500;
```

```
name            | altitude
----------------+----------
Las Vegas       | 2174
Mariposa        | 1953
(2 rows)
```

Ở đây ONLY trước cities chỉ rằng truy vấn nên được chạy chỉ đối với bảng cities, và không với các bảng bên dưới cities trong tôn ti trật tự kế thừa. Nhiều lệnh mà chúng tôi đã thảo luận rồi như SELECT, UPDATE, và DELETE hỗ trợ cú pháp ONLY này.

> **Lưu ý**: Dù sự kế thừa thường là hữu dụng, nó còn chưa được tích hợp vào với các hằng duy nhất hoặc các khóa ngoại, chúng hạn chế tính hữu dụng của nó. Xem Phần 5.8 để có thêm chi tiết.

3.7. Kết luận

PostgreSQL có nhiều tính năng không được đề cập tới trong giới thiệu của sách chỉ dẫn này, nó từng được hướng tới những người sử dụng mới hơn của SQL. Các tính năng đó được thảo luận chi tiết hơn trong phần còn lại của sách chỉ dẫn này.

Nếu bạn cảm thấy bạn cần nhiều tư liệu giới thiệu hơn, xin hãy thăm website PostgreSQL[2] để có các đường liên kết tới nhiều tài nguyên hơn.

2 http://www.postgresql.org/

II. Ngôn ngữ SQL

Phần này mô tả sử dụng ngôn ngữ SQL trong PostgreSQL. Chúng ta bắt đầu với việc mô tả cú pháp chung của SQL, rồi giải thích cách để tạo ra các cấu trúc để lưu trữ dữ liệu, cách để đưa dữ liệu vào cơ sở dữ liệu, và cách để truy vấn nó. Phần giữa liệt kê các dạng và các hàm dữ liệu có sẵn để sử dụng trong các lệnh SQL. Phần còn lại đề cập tới vài khía cạnh quan trọng cho việc tinh chỉnh một cơ sở dữ liệu để có hiệu năng tối ưu.

Thông tin trong phần này được sắp xếp sao cho người sử dụng mới có thể đi theo từ đầu chí cuối để có được một sự hiểu biết đầy đủ các chủ đề mà không phải tham chiếu tới quá nhiều lần. Các chương có ý định sẽ là khép kín, sao cho những người sử dụng tiên tiến có thể đọc được các chương một cách riêng rẽ khi họ chọn. Thông tin trong phần này được trình bày theo cách thức kể chuyện theo các đơn vị chủ đề. Các độc giả tìm kiếm một mô tả hoàn chỉnh của một lệnh đặc biệt sẽ xem Phần VI.

Các độc giả của phần này sẽ biết cách để kết nối tới một cơ sở dữ liệu PostgreSQL và đưa ra các lệnh SQL. Các độc giả chưa quen với các vấn đề đó được khuyến khích đọc Phần I trước. Các lệnh SQL được đưa vào một cách điển hình bằng việc sử dụng trình đầu cuối (terminal) tương tác psql của PostgreSQL, nhưng các chương trình khác có chức năng tương tự cũng có thể được sử dụng.

Chương 4. Cú pháp SQL

Chương này mô tả cú pháp của SQL. Nó tạo thành nền tảng để hiểu các chương sau mà sẽ đi vào chi tiết về cách mà các lệnh SQL được áp dụng để xác định và sửa đổi các dữ liệu.

Chúng tôi cũng khuyến cáo những người sử dụng mà đã quen rồi với SQL đọc chương này cẩn thận vì nó có vài qui tắc và khái niệm được triển khai không nhất quán trong các cơ sở dữ liệu SQL hoặc là đặc biệt đối với PostgreSQL.

4.1. Cấu trúc từ vựng

Đầu vào SQL bao gồm một sự tuần tự các *lệnh*. Một lệnh được cấu tạo từ một sự tuần tự các thẻ token, kết thúc bằng một dấu chấm phẩy (";"). Kết thúc của dòng đầu vào cũng kết thúc một lệnh. Thẻ token nào là hợp lệ phụ thuộc vào cú pháp của lệnh đặc biệt đó.

Một thẻ token có thể là một *từ khóa*, một *mã định danh*, một *mã định danh trong ngoặc (quoted identifier)*, một *hằng số*, hoặc một biểu tượng ký tự đặc biệt. Các thẻ token thường được cách nhau bằng các dấu trắng (khoảng trống, các tab, dòng mới), nhưng sẽ là không cần nếu không có sự tối nghĩa (nó thường chỉ là trường hợp nếu một ký tự đặc biệt liền kề với một số dạng thẻ token khác).

Ví dụ, sau đây là đầu vào SQL hợp lệ (theo cú pháp):

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

Đây là sự tuần tự của 3 lệnh, mỗi lệnh một dòng (dù điều này không là bắt buộc; hơn 1 lệnh có thể nằm trên một dòng, và các lệnh có thể được chia tách một cách hữu dụng trên các dòng).

Hơn nữa, các *ghi chú (bình luận)* có thể xảy ra ở đầu vào SQL. Chúng không phải là các thẻ token, chúng tương đương một cách có hiệu quả với các khoảng trắng.

Cú pháp SQL không thật nhất quán đối với những gì các thẻ token nhận diện các lệnh, toán hạng và tham số. Vài thẻ token đầu tiên thường là tên lệnh, nên trong ví dụ ở trên chúng ta thường có thể nói về một lệnh chọn "SELECT", một lệnh cập nhật - "UPDATE", và một lệnh chèn - "INSERT". Nhưng ví dụ lệnh UPDATE luôn đòi hỏi một thẻ token SET xuất hiện ở một vị trí nhất định, và điều khác nhau đặc biệt này của INSERT cũng đòi hỏi một VALUES để hoàn chỉnh. Các qui tắc cú pháp chính xác cho từng lệnh được mô tả ở Phần VI.

4.1.1. Mã định danh và các từ khóa

Các thẻ token như SELECT, UPDATE, hoặc VALUES trong ví dụ ở trên là những ví dụ về các *từ khóa*, đó là, các từ có một nghĩa cố định trong ngôn ngữ SQL. Các thẻ token MY_TABLE và A là những ví dụ về các *mã định danh*. Chúng xác định các tên bảng, cột, hoặc các đối tượng khác của cơ sở dữ liệu, phụ thuộc vào lệnh mà chúng sẽ được sử dụng trong đó. Vì thế chúng đôi khi được gọi đơn giản là "các tên". Các từ khóa và các mã định danh có cùng cấu trúc từ vựng, nghĩa là người ta không thể biết liệu một thẻ token có phải là một mã định danh hay là một từ khóa mà không cần biết tới ngôn

ngữ. Một danh sách đầy đủ các từ khóa có thể thấy trong Phụ lục C.

Các mã định danh SQL và các từ khóa phải bắt đầu với một ký tự (a-z, nhưng cũng là các ký tự với các dấu đặc biệt và các ký tự không phải là Latin) hoặc một dấu gạch dưới (_). Các ký tự tiếp sau trong một mã định danh hoặc từ khóa có thể là các ký tự, các dấu gạch dưới, các con số (0-9), hoặc các dấu $. Lưu ý rằng các dấu $ không được phép trong các mã định danh theo ký tự của tiêu chuẩn SQL, nên sự sử dụng của chúng có thể trả về các ứng dụng ít khả chuyển hơn. Tiêu chuẩn SQL sẽ không xác định một từ khóa có chứa các chữ số hoặc bắt đầu hoặc kết thúc với một dấu gạch chân, sao cho các mã định danh của dạng này là an toàn đối với xung đột có khả năng với các mở rộng trong tương lai của tiêu chuẩn đó.

Hệ thống sử dụng không nhiều hơn NAMEDATALEN -1 byte đối với một mã định danh; các tên dài hơn có thể được viết trong các lệnh, nhưng chúng sẽ bị cắt ngắn bớt. Mặc định, NAMEDATALEN là 64 byte so với độ dài tối đa của mã định danh là 63 byte. Nếu giới hạn này là có vấn đề, thì nó có thể sẽ nảy sinh bằng việc thay đổi hằng số NAMEDATALEN trong src/include/pg_config_manual.h.

Các từ khóa và các mã định danh không nằm trong dấu ngoặc sẽ phân biệt chữ hoa với chữ thường. Vì thế:

```
UPDATE MY_TABLE SET A = 5;
```

có thể tương đương được viết như là:

```
uPDaTE my_TabLE SeT a = 5;
```

Một qui ước thường được sử dụng là để viết các từ khóa theo chữ hoa và các tên theo chữ thường, nghĩa là:

```
UPDATE my_table SET a = 5;
```

Có dạng mã định danh thứ 2: *mã định danh giới hạn (delimited identifier)* hoặc *mã định dạng trong ngoặc (quoted identifier)*. Nó được hình thành bằng việc kèm theo một tuần tự tùy ý các ký tự trong các dấu ngoặc kép (“). Một mã định danh giới hạn luôn là một mã định danh, không bao giờ là một từ khóa. Nên “select” có thể được sử dụng để tham chiếu tới một cột hoặc bảng có tên là “select”, trong khi một select không trong các dấu ngoặc kép có thể được coi như một từ khóa và có thể vì thế được gợi ý là một lỗi phân tích khi được sử dụng ở nơi mà một tên bảng hoặc cột được mong đợi. Ví dụ có thể được viết với các mã định danh trong ngoặc giống như thế này:

```
UPDATE "my_table" SET "a" = 5;
```

Các mã định danh trong ngoặc có thể chứa bất kỳ ký tự nào, ngoại trừ ký tự với mã 0. (Để đưa vào một dấu ngặc kép, hãy viết 2 dấu ngặc kép). Điều này cho phép việc xây dựng các tên bảng hoặc cột mà nếu khác đi có thể là không thể, như tên có chứa các dấu trống hoặc ký hiệu &. Giới hạn độ dài vẫn áp dụng.

Một biến thể của các mã định danh nằm trong dấu ngoặc cho phép bao gồm các ký tự Unicode thoát ly được các điểm mã của chúng xác định. Các biến thể này bắt đầu với U& (chữ U hoa hoặc thường đi sau là ký hiệu &) ngay trước khi mở dấu ngoặc kép, mà không có bất kỳ chỗ trống nào ở giữa, ví

dụ U&”foo”. (Lưu ý là điều này tạo ra một sự không rõ ràng với toán tử &. Hãy sử dụng ký tự trống xung quanh toán tử đó để tránh vấn đề này). Bên trong các dấu ngoặc, các ký tự Unicode có thể được chỉ định ở dạng thoát ly bằng việc viết một dấu chéo ngược đi sau là dấu cộng (+), đi sau là một số điểm mã có 6 chữ số theo hệ 16 (hexadecimal). Ví dụ, “dữ liệu” của mã định danh có thể được viết như là

```
U&"d\0061t\+000061"
```

Ví dụ ít tầm thường hơn sau đây viết từ tiếng Nga “slon” (con voi) theo các ký tự Cyrillic:

```
U&"\0441\043B\043E\043D"
```

Nếu một ký tự thoát khác với dấu chéo ngược được mong muốn, thì nó có thể được chỉ định bằng việc sử dụng mệnh đề UESCAPE sau chuỗi đó, ví dụ:

```
U&"d!0061t!+000061" UESCAPE '!'
```

Ký tự thoát có thể là bất kỳ ký tự nào khác với chữ số theo hệ 16, dấu cộng (+), dấu nháy đơn ('), dấu nháy kép (“), hoặc một ký tự trắng. Lưu ý rằng ký tự thoát được viết trong dấu nháy đơn, không phải trong dấu nháy kép.

Để đưa vào ký tự thoát trong mã định danh theo nghĩa đen, hãy viết nó 2 lần.

Cú pháp thoát Unicode chỉ làm việc khi mã máy chủ là UTF8. Khi các mã máy chủ khác được sử dụng, thì chỉ các điểm mã trong dải ASCII (tới \007F) có thể được chỉ định. Cả dạng 4 chữ số và 6 chữ số có thể được sử dụng để chỉ định các cặp thay thế UTF16 để soạn các ký tự với các điểm mã lớn hơn U+FFFF, dù tính sẵn sàng của dạng 6 chữ số về mặt kỹ thuật làm cho điều này không cần thiết. (Khi các cặp thay thế được sử dụng khi mã máy chủ là UTF8, trước hết chúng được kết hợp trong một điểm mã duy nhất mà sau đó được mã hóa theo UTF-8).

Việc đưa vào dấu ngoặc một mã định danh cũng làm cho nó phân biện chữ hoa và chữ thường, trong khi các tên không được đưa vào dấu ngoặc luôn được viết với chữ thường. Ví dụ, các mã định danh FOO, foo, và “foo” được xem là y hệt nhau với PostgreSQL, nhưng “Foo” và “FOO” là khác nhau so với 3 cái đó và khác với nhau. (Việc viết các tên không nằm trong các dấu ngoặc theo chữ thường trong PostgreSQL là không tương thích với tiêu chuẩn SQL, tiêu chuẩn nói rằng các tên không trong dấu ngoặc sẽ được viết theo chữ hoa. Vì thế, foo sẽ là tương đương với “FOO” chứ không tương đương với “foo” theo tiêu chuẩn đó. Nếu bạn muốn viết các ứng dụng khả chuyển được thì bạn được khuyến cáo luôn đưa vào dấu ngoặc một tên đặc biệt hoặc không bao giờ đưa nó vào ngoặc cả).

4.1.2. Hằng số

Có 3 dạng *hằng số ám chỉ dạng* trong PostgreSQL: các chuỗi, các chuỗi bit và các số. Các hằng cũng có thể được chỉ định với các dạng ẩn, chúng có thể cho phép sự trình bày lại chính xác hơn và có hiệu quả hơn bằng việc xử lý của hệ thống. Các lựa chọn thay thế đó được thảo luận trong các tiểu phần bên dưới.

4.1.2.1. Hằng số chuỗi (hằng chuỗi)

Một hằng chuỗi trong SQL là một sự tuần tự tùy ý các ký tự nằm trong các dấu nháy đơn ('), ví dụ 'Đây là một chuỗi'. Để đưa vào một ký tự dấu nháy đơn trong một hằng chuỗi, hãy viết 2 dấu nháy đơn liền nhau, như: 'Dianne'' horse' ('con ngựa của Dianne'). Lưu ý là điều này không là y hệt như với một ký tự nháy kép (“).

Hai hằng chuỗi chỉ được cách biệt nhau bằng dấu trắng với ít nhất một dòng mới sẽ được ghép và được đối xử một cách có hiệu lực dường như chuỗi đó từng được viết như một hằng. Ví dụ:

```
SELECT 'foo'
'bar';
```

là tương đương với:

```
SELECT 'foobar';
```

nhưng

```
SELECT 'foo' 'bar';
```

là cú pháp không hợp lệ (Hành vi khá kỳ lạ này đặc biệt là đối với SQL; PostgreSQL tuân theo tiêu chuẩn).

4.1.2.2. Hằng chuỗi với các thoát dạng C

PostgreSQL cũng chấp nhận các hằng chuỗi “thoát”, chúng là một mở rộng đối với tiêu chuẩn SQL.

Một hằng chuỗi thoát được đặc trưng bằng việc viết ký tự E (chữ hoa hoặc chữ thường) ngay trước dấu nháy đơn, nghĩa là, E'foo'. (Khi tiếp tục một hằng chuỗi thoát xuyên khắp các dòng, hãy viết E chỉ trước dấu nháy mở đầu tiên). Trong một chuỗi thoát, một ký tự chéo ngược (\) bắt đầu một sự tuần tự thoát chéo ngược giống C, trong đó sự kết hợp của dấu chéo ngược và (các) ký tự theo sau thể hiện một giá trị byte đặc biệt, như chỉ ra trong Bảng 4-1.

Bảng 4-1. Tuần tự thoát của dấu chéo ngược

Tuần tự thoát của dấu chéo ngược	Giải nghĩa
\b	dấu xóa ngược (backspace)
\f	mẫu cấp dữ liệu (form feed)
\n	dòng mới (newline)
\r	carriage return
\t	tab
\o, \oo, \ooo (o = 0 - 7)	giá trị byte theo hệ số 8 (octal byte value)
\xh, \xhh (h = 0 - 9, A - F)	giá trị byte theo hệ số 16 (hexadecimal byte value)
\uxxxx, \Uxxxxxxxx (x = 0 - 9, A - F)	giá trị ký tự Unicode 16 hoặc 32 bit theo hệ số 16 (16 or 32-bit hexadecimal Unicode character value).

Bất kỳ ký tự nào khác theo sau một dấu chéo ngược sẽ được lấy theo nghĩa đen. Vì thế, để đưa vào một ký tự dấu chéo ngược, hãy viết 2 dấu chéo ngược (\\). Hơn nữa, dấu nháy đơn có thể được đưa vào trong một chuỗi thoát bằng việc viết \', bổ sung thêm vào cách thức thông thường của dấu nháy

đúp (”).

Là trách nhiệm của bạn rằng những tuần tự theo byte mà bạn tạo ra, đặc biệt khi sử dụng các thoát theo các hệ 8 hoặc 16, tạo nên các ký tự hợp lệ trong việc mã hóa tập các ký tự trên máy chủ. Khi việc mã hóa máy chủ là UTF-8, thì các thoát Unicode hoặc cú pháp thoát Unicode cho lựa chọn thay thế, như trong Phần 4.1.2.3, sẽ được sử dụng thay. (Lựa chọn thay thế có thể là việc mã hóa UTF-8 bằng tay và viết ra các bytes, nó có thể là rất nặng nhọc).

Cú pháp thoát Unicode làm việc đầy đủ chỉ khi việc mã hóa máy chủ là UTF-8. Khi các mã hóa máy chủ khác được sử dụng, thì chỉ các điểm mã trong dải ASCII (tới `\u007F`) có thể được chỉ định. Cả mẫu 4 chữ số và 8 chữ số có thể được sử dụng để chỉ định các cặp thay thế UTF-16 để soạn ra các ký tự với các điểm mã lớn hơn U+FFFF, dù tính sẵn sàng của mẫu 8 chữ số, về mặt kỹ thuật, làm cho điều này là không cần thiết. (Khi các cặp thay thế được sử dụng khi việc mã hóa máy chủ là UTF-8, thì chúng trước hết được kết hợp trong một điểm mã duy nhất mà sau đó được mã hóa theo UTF-8).

Thận trọng

Nếu tham số cấu hình standard_conforming_strings (các chuỗi tuân thủ tiêu chuẩn) mà là tắt (off), thì PostgreSQL nhận các thoát dấu chéo ngược theo cả các hằng chuỗi thoát và thông thường. Đây là sự tương thích ngược với hành vi lịch sử, nơi mà các thoát dấu chéo ngược từng luôn được thừa nhận. Dù standard_conforming_strings hiện mặc định là tắt, thì mặc định này sẽ thay đổi thành bật (on) trong một phiên bản trong tương lai vì sự tuân thủ các tiêu chuẩn được cải thiện. Các ứng dụng vì thể được khuyến khích để chuyển đổi khỏi việc sử dụng các thoát chéo ngược. Nếu bạn cần sử dụng một thoát chéo ngược để thể hiện một ký tự đặc biệt, hãy viết hằng chuỗi đó với một chữ E để đảm bảo nó sẽ được điều khiển cùng một cách như trong các phiên bản trong tương lai.

Bổ sung thêm vào standard_conforming_strings, các tham số cấu hình của escape_string_warning (cảnh báo chuỗi thoát) và backslash_quote (dấu chéo trong ngoặc) điều chỉnh đối xử của các dấu chéo ngược trong các hằng chuỗi.

Ký tự với mã 0 không thể nằm trong một hằng chuỗi.

4.1.2.3. Hằng chuỗi với các thoát Unicode

PostgreSQL cũng hỗ trợ dạng cú pháp thoát khác cho các chuỗi mà cho phép việc chỉ định các ký tự Unicode tùy ý bằng điểm mã. Một hằng chuỗi thoát Unicode bắt đầu với U& (U là chữ hoa hoặc chữ thường và theo sau là dấu và &) ngay trước khi mở dấu nháy, mà không có bất kỳ dấu trống nào ở giữa, ví dụ, U&'foo'. (Lưu ý rằng điều này tạo ra một sự tù mù với toán tử &. Hãy sử dụng các dấu trống xung quanh toán tử đó để tránh vấn đề này). Bên trong các dấy nháy, các ký tự Unicode có thể được chỉ định ở dạng được thoát bằng việc viết một dấu chéo ngược đi sau là số

điểm mã 4 chữ số theo hệ 16 hoặc lựa chọn tùy ý một dấu chéo ngược đi sau là một dấu cộng (+) sau đó là một số điểm mã 6 chữ số hệ 16. Ví dụ, 'sữ liệu' chuỗi có thể được viết như là

```
U&'d\0061t\+000061'
```

Ví dụ ít thông thường hơn sau đây viết từ tiếng Nga “slon” (con voi) theo các ký tự Cyrillic:

```
(U&'\0441\043B\043E\043D')
```

Nếu một ký tự thoát khác với dấu chéo ngược là mong muốn, thì nó có thể được chỉ định bằng việc sử dụng mệnh đề UESCAPE sau chuỗi đó, ví dụ:

```
U&'d!0061t!+000061' UESCAPE '!'
```

Ký tự thoát có thể là bất kỳ ký tự đơn nào khác với một chữ số hệ 16, dấu cộng, dấu nháy đơn, dấu nháy kép hoặc một ký tự dấu trắng.

Cú pháp thoát Unicode chỉ làm việc khi mã hóa máy chủ là UTF-8. Khi các mã hóa máy chủ khác được sử dụng, thì chỉ các điểm mã trong dải ASCII (tới \007F) có thể được chỉ định. Cả 2 dạng 4 chữ số và 6 chữ số đều có thể được sử dụng để chỉ định các cặp thay thế UTF-16 để soạn ra các ký tự với các điểm mã lớn hơn so với U+FFFF, dù sự sẵn sàng của mẫu 6 chữ số, về mặt kỹ thuật, làm cho điều này là không cần thiết. (Khi các cặp thay thế được sử dụng khi mã hóa máy chủ là UTF-8, thì chúng trước hết được kết hợp vào trong điểm mã duy nhất mà sau đó được mã hóa theo UTF-8).

Hơn nữa, cú pháp thoát Unicode cho các hằng chuỗi chỉ làm việc khi tham số cấu hình (standard_conforming_strings) được bật. Điều này là vì nếu khác thì cú pháp này có thể gây lẫn lộn cho các máy trạm mà phân tích cú pháp các lệnh SQL tới điểm mà nó có thể dẫn tới các sự tiêm SQL (SQL injections) và các vấn đề an ninh tương tự. Nếu tham số đó là tắt (off), thì cú pháp này sẽ bị từ chối với một thông điệp lỗi.

Để đưa vào ký tự thoát trong chuỗi theo nghĩa đen, hãy viết nó 2 lần.

4.1.2.4. Hằng chuỗi trong các dấu $

Trong khi cú pháp tiêu chuẩn cho việc chỉ định các hằng chuỗi thường là thuận tiện, thì nó có thể là khó để hiểu khi các chuỗi mong muốn có chứa nhiều dấu nháy hoặc dấu chéo ngược, vì từng dấu đó phải được đúp bản. Để cho phép các truy vấn có khả năng đọc được nhiều hơn trong những tình huống như vậy, PostgreSQL đưa ra cách khác, gọi là “đưa vào trong các dấu $”, để viết các hằng chuỗi. Một hằng chuỗi trong các dấu $ bao gồm một dấu $, một “thẻ” tùy chọn của 0 hoặc các ký tự, một dấu $ nữa, một sự tuần tự tùy ý các ký tự tạo nên nội dung chuỗi, một dấu $, thẻ y hệt bắt đầu dấu $ đó, và một dấu $. Ví dụ, đây là 2 cách khác nhau để thể hiện chuỗi “Dianne's horse” bằng việc sử dụng các dấu $:

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

Lưu ý là bên trong chuỗi được đưa vào các dấu $, các nháy đơn có thể được sử dụng mà không cần phải được thoát. Quả thực, không ký tự nào bên trong một chuỗi được đưa vào các dấu $ được thoát ra bao giờ cả: nội dung của chuỗi luôn được viết theo nghĩa đen. Các dấu chéo ngược không phải là

đặc biệt, và chúng không là các dấu $, trừ phi chúng là một phần của một sự trùng khớp tuần tự với thẻ mở đầu.

Có khả năng lồng các hằng chuỗi được đưa vào các dấu $ bằng việc chọn các thẻ khác ở từng mức lồng. Điều này được sử dụng phổ biến nhất trong khi viết các định nghĩa hàm. Ví dụ:

```
$function$
BEGIN
        RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Ở đây, tuần tự `$q$[\t\r\n\v\\]$q$` thể hiện một chuỗi theo nghĩa đen được đưa vào trong các dấu $ là `[\t\r\n\v\\]`, nó sẽ được thừa nhận khi thân của hàm được PostgreSQL thực thi. Nhưng vì sự tuần tự không khớp với dấu phân cách các dấu $ vòng ngoài `$function$`, nên chỉ một số ký tự bên trong hằng đó cho tới nay như là chuỗi vòng ngoài được quan tâm.

Thẻ, nếu có, của một chuỗi được đưa vào trong các dấu $ tuân theo cùng các qui ước như một mã định danh không nằm trong các dấu, ngoại trừ là nó không thể có chứa một dấu $. Các thẻ là phân biệt chữ hoa và chữ thường, nên `$tag$String content$tag$` là đúng, nhưng `$TAG$String content$tag$` thì không.

Chuỗi trong các dấu $ mà đi theo một từ khóa hoặc mã định danh phải được tách bạch khỏi nó bằng dấu trắng; nếu không thì dấu phân cách của dấu $ có thể được coi như một phần của mã định danh đi trước.

Việc đưa vào trong các dấu $ không phải là một phần của tiêu chuẩn SQL, nhưng nó thường là một cách thức thuận tiện để viết các hằng chuỗi phức tạp hơn là cú pháp các dấu nháy đơn tuân thủ chuẩn. Nó đặc biệt hữu dụng khi thể hiện các hằng chuỗi bên trong các hằng khác, như thường là cần thiết trong các định nghĩa hàm thủ tục. Với cú pháp dấu nháy đơn, từng dấu chéo ngược trong ví dụ ở trên có thể phải được viết như 4 dấu chéo ngược, nó có thể được giảm tới 2 dấu chéo ngược trong việc phân tích cú pháp hằng chuỗi gốc ban đầu, và sau đó giảm về 1 dấu chéo ngược khi hằng chuỗi vòng trong được tái phân tích cú pháp trong quá trình thực thi hàm.

4.1.2.5. Hằng chuỗi bit

Các hằng chuỗi bit trông giống như các hằng chuỗi thông thường với một ký tự B (chữ hoa hoặc chữ thường) ngay lập tức trước khi mở ngoặc (không có các dấu trắng xen giữa), như, B'1001'. Các ký tự duy nhất được phép bên trong các hằng chuỗi bit là 0 và 1.

Một cách lựa chọn, các hằng chuỗi bit có thể được chỉ định trong ký hiệu theo hệ 16, bằng việc sử dụng một ký tự X đi đầu (chữ thường hoặc chữ hoa), như, X'1FF'. Ký hiệu này là tương đương với một hằng chuỗi bit với 4 chữ số nhị phân cho từng chữ số hệ 16.

Cả 2 dạng hằng chuỗi bit đều có thể được tiếp tục trên các dòng theo cùng cách thức như các hằng chuỗi thông thường. Việc đưa vào trong các dấu $ không thể được sử dụng trong hằng chuỗi bit.

4.1.2.6. Hằng là số

Các hằng là số được chấp nhận theo các mẫu chung:

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

trong đó các chữ số - digits là một hoặc nhiều hơn các chữ số thập phân (từ 0 đến 9). Ít nhất một chữ số phải đứng trước hoặc sau dấu chấm thập phân, nếu một dấu chấm thập phân được sử dụng. Ít nhất một chữ số phải đi theo dấu mũ (e), nếu có một dấu mũ hiện diện. Không thể có bất kỳ dấu trống hay các ký tự nào khác được nhúng trong hằng đó. Lưu ý là bất kỳ dấu cộng hoặc trừ nào đứng trước cũng thực sự không được coi là một phần của hằng số đó; đây là một toán tử được áp dụng cho hằng số.

Một số ví dụ về các hằng là số hợp lệ:

```
42
3.5
4.
.001
5e2
1.925e-3
```

Một hằng là số không bao gồm dấu thập phân, cũng không số mũ ban đầu được giả thiết là dạng số nguyên nếu giá trị của nó khớp theo dạng số nguyên (32 bit); nếu không thì nó được giả thiết sẽ là dạng bigint nếu giá trị của nó khớp theo dạng bigint (64 bit); nếu không thì nó được lấy như là dạng số – numeric. Các hằng mà có các dấu thập phân và/hoặc dấu mũ luôn được giả thiết từ đầu là dạng số – numeric.

Dạng các hằng số dữ liệu được chỉ định ban đầu chỉ là điểm khởi đầu cho các thuật toán qui định dạng. Trong hầu hết các trường hợp hằng đó sẽ được tự động ép vào dạng phù hợp nhất, phụ thuộc vào ngữ cảnh. Khi cần, bạn có thể ép một giá trị số sẽ được biên dịch như một dạng dữ liệu đặc thù bằng việc đưa nó ra. Ví dụ, bạn có thể ép một giá trị số để được đối xử như là dạng real (float4) bằng việc viết:

```
REAL '1.23' -- kiểu chuỗi
1.23::REAL -- kiểu PostgreSQL (lịch sử)
```

Chúng thực sự chỉ là các trường hợp đặc biệt của các ký hiệu đưa ra chung được thảo luận tiếp sau.

4.1.2.7. Hằng các dạng khác

Một hằng của một dạng tùy ý có thể được đưa vào bằng việc sử dụng bất kỳ một trong những ký hiệu nào sau đây:

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

Văn bản của hằng chuỗi được truyền tới thủ tục hoán đổi đầu vào cho dạng có tên là type. Kết quả là một hằng dạng đó được chỉ ra. Sự đưa ra dạng rõ ràng có thể bị bỏ qua nếu không có sự mù mờ

như đối với dạng mà hằng đó phải là (ví dụ, khi nó được chỉ định trực tiếp tới một cột của bảng), trong trường hợp đó nó tự động bị ép buộc.

Hằng chuỗi có thể được viết bằng việc sử dụng hoặc ký hiệu SQL thông thường hoặc đưa vào trong các dấu $.

Cũng có khả năng để chỉ định một dạng ép buộc bằng việc sử dụng cú pháp giống hàm:

```
typename ( 'string' )
```

nhưng không phải tất cả các tên dạng có thể được sử dụng theo cách này; xem Phần 4.2.9 để có thêm chi tiết.

Các cú pháp gọi hàm và ::, CAST(), cũng có thể được sử dụng để chỉ định những hoán đổi dạng thời gian chạy (run-time) của các biểu thức tùy ý, như được thảo luận trong Phần 4.2.9. Để tránh sự tù mù về cú pháp, thì cú pháp dạng 'chuỗi' - type 'string' duy nhất có thể được sử dụng để chỉ định dạng hằng đơn giản theo nghĩa đen. Một giới hạn khác trong cú pháp type 'string' là nó không làm việc đối với các dạng mảng (array); hãy sử dụng :: hoặc CAST() để chỉ định dạng của một hằng mảng.

Cú pháp CAST() tuân thủ SQL. Cú pháp type 'string' là một sự tổng quát hóa tiêu chuẩn đó: SQL chỉ định cú pháp này chỉ cho một ít dạng dữ liệu, nhưng PostgreSQL cho phép nó đối với tất cả các dạng. Cú pháp với :: là sử dụng theo lịch sử của PostgreSQL, như là cú pháp gọi hàm.

4.1.3. Toán tử

Tên của một toán tử là một sự tuần tự cho tới NAMEDATALEN-1 ký tự (63 là mặc định) từ danh sách sau đây:

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

Tuy nhiên, có một ít giới hạn trong các tên toán tử:

- -- và /* không thể xuất hiện ở bất cứ đâu trong tên một toán tử, vì chúng sẽ được coi như là bắt đầu của một chú giải.
- Tên của một toán tử nhiều ký tự không thể kết thúc ở dấu cộng + hoặc dấu trừ -, trừ phi tên đó cũng có ít nhất một trong các ký tự sau: ~ ! @ # % ^ & | ` ?

  Ví dụ, @- là một tên toán tử được phép, nhưng *- thì không. Hạn chế này cho phép PostgreSQL phân tích cú pháp các truy vấn tuân thủ SQL mà không có các khoảng trống giữa các thẻ token.

Khi làm việc với các tên toán tử không theo tiêu chuẩn SQL, bạn sẽ thường cần phải tách bạch các toán tử liền kề bằng các dấu trống để tránh sự mù mờ. Ví dụ, nếu bạn đã xác định được một toán tử một toán hạng bên trái có tên là @, thì bạn không thể viết X* @Y; bạn phải viết X* @Y để đảm bảo rằng PostgreSQL đọc được nó như là 2 tên toán tử chứ không phải là một.

4.1.4. Ký tự đặc biệt

Một số ký tự mà không phải thao abc có một ý nghĩa đặc biệt và là khác với việc là một toán tử. Các chi tiết về sử dụng có thể được thấy ở vị trí nơi mà yếu tố cú pháp tương ứng được mô tả. Phần này chỉ tồn tại để tư vấn cho sự tồn tại và tóm tắt các mục tiêu của các ký tự đó.

- Dấu $ theo sau là các chữ số được sử dụng để thể hiện một tham số vị trí trong thân của một định nghĩa hàm hoặc một khai báo được chuẩn bị. Trong các ngữ cảnh khác thì dấu $ có thể là một phần của một mã định danh hoặc một hằng chuỗi trong các dấu $.
- Các dấu ngoặc đơn (()) có ý nghĩa thông thường của chúng để nhóm các biểu thức và tăng cường quyền ưu tiên trước. Trong một số trường hợp các dấu ngoặc đơn được yêu cầu như một phần của cú pháp cố định của một lệnh SQL đặc biệt.
- Các dấu ngoặc vuông ([]) được sử dụng để lựa chọn các phần tử của một mảng. Xem Phần 8.14 để có thêm thông tin về các mảng.
- Dấu phẩy (,) được sử dụng trong một số cấu trúc cú pháp để tách bạch các yếu tố của một danh sách.
- Dấu chấm phẩy (;) kết thúc một lệnh SQL. Nó không thể xuất hiện ở bất kỳ đâu trong một lệnh, ngoại trừ bên trong một hằng chuỗi hoặc mã định danh trong ngoặc.
- Dấu 2 chấm (:) được sử dụng để chọn "các lát cắt" từ mảng. (Xem Phần 8.14). Trong các biến thể SQL nhất định (như SQL nhúng), dấu 2 chấm được sử dụng cho các tên biến tiền tố.
- Dấu sao (*) được sử dụng trong một số ngữ cảnh để biểu thị tất cả các trường của một hàng của bảng hoặc giá trị kết hợp. Nó cũng có một ý nghĩa đặc biệt khi được sử dụng như biến số của một hàm tổng hợp, ấy là sự tổng hợp không đòi hỏi bất kỳ tham số rõ ràng nào.
- Dấu chấm (.) được sử dụng trong các hằng là số, và để tách biệt các tên sơ đồ, bảng và cột.

4.1.5. Các chú giải

Một chú giải là một tuần tự các ký tự bắt đầu với 2 dấu gạch ngang (--) và mở rộng cho tới cuối dòng, như:

```
-- Đây là một chú giải SQL tiêu chuẩn
```

Như một sự lựa chọn, các chú giải khối dạng C có thể được sử dụng:

```
/* multiline comment
* with nesting: /* nested block comment */
*/
```

trong đó chú giải bắt đầu với /* và mở rộng để khớp với dấu */ nữa. Khối chú giải lồng đó, như được chỉ định trong tiêu chuẩn SQL nhưng không giống C, vì thế một người có thể đưa ra chú giải các khối mã lớn hơn mà có thể chứa các chú giải khối đang tồn tại.

Một chú giải bị loại bỏ khỏi luồng đầu vào trước phân tích cú pháp tiếp và được dấu trắng thay thế một cách có hiệu quả.

4.1.6. Quyền ưu tiên trước của từ vựng

Bảng 4-2 chỉ ra *quyền ưu tiên trước* và tính liên kết của các toán tử trong PostgreSQL. Hầu hết các toán tử có cùng quyền ưu tiên trước và là liên kết bên trái. Quyền ưu tiên trước và tính liên kết của các toán tử được liên kết chặt chẽ trong trình phân tích cú pháp. Điều này có thể dẫn tới hành vi không trực giác; ví dụ các toán tử Boolean <and> có một quyền ưu tiên trước khác so với các toán tử Boolean <= and >=. Hơn nữa, đôi khi bạn sẽ cần phải bổ sung thêm các dấu ngoặc đơn khi sử dụng các tổ hợp nhị phân và các toán tử một toán hạng. Ví dụ:

```
SELECT 5 ! - 6;
```

sẽ được phân tích như là:

```
SELECT 5 ! (- 6);
```

vì trình phân tích cú pháp không có ý tưởng – cho tới khi là quá muộn – nên dấu chấm than ! được xác định như là một toán tử hậu tố, không phải là một trung tố. Để có được hành vi mong muốn trong trường hợp này, bạn phải viết:

```
SELECT (5 !) - 6;
```

Đây là cái giá phải trả cho tính có thể mở rộng.

Bảng 4-2. Quyền ưu tiên trước của toán tử (theo chiều thấp dần)

Toán tử/ yếu tố	Tính liên kết	Mô tả
.	trái (left)	phân cách tên bảng/cột
::	trái	Dạng cast theo kiểu của PostgreSQL
[ ]	trái	lựa chọn phần tử mảng
-	phải (right)	các dấu trừ một toán hạng
^	trái	dấu mũ
* / %	trái	dấu nhân, chia và phần trăm
+ -	trái	dấu cộng, dấu trừ
IS		IS TRUE , IS FALSE , IS UNKNOWN , IS NULL (LÀ ĐÚNG, LÀ SAI, LÀ KHÔNG BIẾT, LÀ BẰNG 0)
ISNULL		kiểm thử xem có là null
NOTNULL		kiểm thử xem có là không phải null
(any other - bất kỳ gì khác)	trái	tất cả các toán tử khác bẩm sinh và do người sử dụng định nghĩa
IN		thiết lập quan hệ thành viên
BETWEEN		nằm trong dải
OVERLAPS		chồng lấn theo khoảng thời gian
LIKE ILIKE SIMILAR		khớp mẫu chuỗi
< >		nhỏ hơn, lớn hơn
=	phải	bằng nhau, chỉ định
NOT	phải	phủ định theo logic
AND	trái	và theo logic

Toán tử/ yếu tố	Tính liên kết	Mô tả
OR	trái	hoặc theo logic

Lưu ý rằng các qui định ưu tiên trước của các toán tử cũng áp dụng cho các toán tử do người sử dụng định nghĩa mà cũng có cùng các tên như các toán tử được xây dựng sẵn, được nhắc tới ở trên. Ví dụ, nếu bạn định nghĩa một toán tử "+" cho một số dạng dữ liệu tùy biến thì nó sẽ có cùng ưu tiên trước như toán tử "+" được xây dựng sẵn, bất kể của bạn là thế nào.

Khi một tên toán tử đủ điều kiện theo một sơ đồ nào đó được sử dụng trong cú pháp toán tử OPERATOR, như ví dụ trong:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

thì cấu trúc của OPERATOR được lấy để có sự ưu tiên trước mặc định được chỉ ra trong Bảng 4-2 cho toán tử "bất kỳ gì khác". Điều này là đúng bất kể toán tử đặc biệt nào xuất hiện trong OPERATOR().

4.2. Biểu thức giá trị

Các biểu thức giá trị sẽ được sử dụng trong các ngữ cảnh khác nhau, như trong danh sách đích của lệnh SELECT, khi các giá trị cột mới trong INSERT hoặc UPDATE hoặc trong các điều kiện trong một số lệnh. Kết quả của một biểu thức giá trị đôi khi được gọi là một lượng vô hướng, để phân biệt nó với kết quả của một biểu thức bảng (nó là một bảng). Các biểu thức giá trị vì thế cũng được gọi là các biểu thức vô hướng (hoặc thậm chí đơn giản là các biểu thức). Cú pháp của biểu thức cho phép tính toán các giá trị từ các phần nguyên sơ bằng việc sử dụng tính toán số học, logic, tập hợp và các hoạt động khác.

Một biểu thức giá trị là một biểu thức dạng sau đây:

- Một giá trị hằng
- Một tham chiếu cột
- Một tham chiếu tham số vị trí, trong thân của định nghĩa hàm hoặc khai báo được chuẩn bị
- Một biểu thức có đánh chỉ số dưới
- Một biểu thức lựa chọn trường
- Một viện dẫn toán tử
- Một lời gọi hàm
- Một biểu thức tổng hợp
- Một lời gọi hàm cửa sổ
- Một dạng phát hành cast
- Một truy vấn con vô hướng

- Một cấu trúc mảng
- Một cấu trúc hàng
- Biểu thức giá trị khác trong các dấu ngoặc đơn (được sử dụng để tạo nhóm các biểu thức con và ghi đè ưu tiên trước)

Bổ sung vào danh sách này, có một số cấu trúc mà có thể được phân loại như một biểu thức nhưng không tuân theo bất kỳ qui tắc cú pháp chung nào. Chúng thường có ngữ nghĩa của một hàm hoặc toán tử và được giải thích ở vị trí phù hợp trong Chương 9. Một ví dụ là mệnh đề IS NULL.

Chúng ta đã thảo luận các hằng trong Phần 4.1.2. Các phần sau đây thảo luận các lựa chọn còn lại.

4.2.1. Các tham chiếu cột

Một cột có thể được tham chiếu ở dạng:

```
correlation.columnname
```

correlation là tên của một bảng (có khả năng đủ điều kiện với một tên sơ đồ), hoặc một tên hiệu (alias) đối với một bảng được xác định bằng một mệnh đề FROM. Tên correlation và dấu chấm có thể được bỏ qua nếu tên cột là duy nhất xuyên tất cả các bảng đang được sử dụng trong truy vấn hiện hành. (Xem thêm Chương 7).

4.2.2. Tham số vị trí

Một tham chiếu tham số vị trí được sử dụng để chi ra một giá trị được cung cấp từ bên ngoài cho một lệnh SQL. Các tham số được sử dụng trong các định nghĩa hàm SQL và trong các truy vấn được chuẩn bị. Một số thư viện máy trạm cũng hỗ trợ việc chỉ định các giá trị dữ liệu tách biệt khỏi chuỗi lệnh SQL, trong trường hợp đó các tham số được sử dụng để tham chiếu tới các giá trị dữ liệu nằm ngoài dòng. Mẫu của một tham chiếu tham số là:

```
$number
```

Ví dụ, hãy xem xét định nghĩa của một hàm, dept, như:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Ở đây $1 tham chiếu tới giá trị của đối số hàm đầu tiên bất kỳ khi nào hàm đó được gọi.

4.2.3. Chỉ số dưới - Subscript

Nếu một biểu thức có một giá trị ở dạng mảng, thì một phần tử đặc thù của giá trị mảng đó có thể được trích xuất bằng việc viết

```
expression[subscript]
```

hoặc nhiều phần tử liền kề (một "lát cắt mảng") có thể được trích xuất bằng việc viết

```
expression[lower_subscript:upper_subscript]
```

(Ở đây, các dấu ngoặc vuông [ ] có nghĩa là sẽ xuất hiện theo nghĩa đen). Mỗi subscript bản thân nó là một biểu thức, nó phải có một giá trị nguyên.

Nói chung biểu thức mảng phải nằm trong các dấu ngoặc đơn, nhưng các dấu ngoặc đơn có thể bị bỏ qua khi biểu thức đó sẽ được viết theo chỉ số dưới chỉ là một tham chiếu cột hoặc tham số vị trí. Hơn nữa, nhiều chỉ số dưới có thể được ghép khi mảng gốc ban đầu là đa chiều. Ví dụ:

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

Các dấu ngoặc đơn trong ví dụ cuối được yêu cầu. Xem phần 8.14 để biết thêm về các mảng.

4.2.4. Chọn trường

Nếu một biểu thức có một giá trị ở dạng tổng hợp (dạng hàng), thì một trường đặc biệt của hàng đó có thể được trích xuất bằng cách viết

```
expression.fieldname
```

Nói chung biểu thức hàng phải nằm trong các dấu ngoặc đơn, nhưng các dấu ngoặc đơn có thể bị bỏ qua khi biểu thức đó được chọn từ chỉ một tham chiếu bảng hoặc tham số vị trí. Ví dụ:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

(Vì thế, một tham chiếu cột đủ tiêu chuẩn thực sự chỉ là một trường hợp đặc biệt của cú pháp chọn trường). Một trường hợp đặc biệt quan trọng là việc trích xuất một trường từ một cột của bảng ở dạng tổng hợp:

```
(compositecol).somefield
(mytable.compositecol).somefield
```

Các dấu ngoặc đơn được yêu cầu ở đây để chỉ ra rằng compositecol là một tên cột chứ không phải là tên bảng, hoặc rằng mytable là một tên bảng chứ không phải là tên sơ đồ trong trường hợp thứ 2.

4.2.5. Viện dẫn toán tử

Có 3 khả năng cú pháp cho một sự viện dẫn toán tử:

expression operator expression (toán tử trung tố nhị phân)

operator expression (toán tử tiền tố một toán hạng)

expression operator (toán tử hậu tố một toán hạng)

trong đó thẻ toán tử operator đi sau các qui tắc cú pháp của Phần 4.1.3, hoặc là một trong những từ khóa AND, OR, và NOT, hoặc là một tên toán tử đủ điều kiện ở dạng:

```
OPERATOR(schema.operatorname)
```

Những toán tử đặc biệt nào tồn tại và liệu chúng có là một toán hạng hay nhị phân sẽ phụ thuộc vào các toán tử nào từng được hệ thống hoặc người sử dụng định nghĩa. Chương 9 mô tả các toán tử được xây dựng sẵn.

4.2.6. Lời gọi hàm

Cú pháp một lời gọi hàm là tên của hàm (có thể đủ điều kiện với một tên sơ đồ), theo sau là danh sách các đối số được đưa vào trong các dấu ngoặc đơn:

```
function_name ([expression [, expression ... ]] )
```

Ví dụ, thứ sau đây tính toán căn bậc 2:

```
sqrt(2)
```

Danh sách các hàm xây dựng sẵn là trong Chương 9. Các hàm khác có thể được người sử dụng bổ sung thêm. Các đối số có thể có các tên được tùy ý gắn vào. Xem Phần 4.3 để có thêm các chi tiết.

4.2.7. Biểu thức tổng hợp

Một biểu thức tổng hợp đại diện cho ứng dụng của một hàm tổng hợp khắp các hàng được một truy vấn lựa chọn. Một hàm tổng hợp làm giảm nhiều đầu vào tới một giá trị đầu vào duy nhất, như tổng hoặc trung bình các đầu vào. Cú pháp của một biểu thức tổng hợp là một trong những thứ sau:

```
aggregate_name (expression [ , ... ] [ order_by_clause ] )
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] )
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] )
aggregate_name ( * )
```

trong đó aggregate_name là một tổng hợp được định nghĩa trước (có khả năng đủ điều kiện với một tên sơ đồ), expression là bất kỳ biểu thức nào mà bản thân nó không chứa một biểu thức tổng hợp hoặc một lời gọi hàm cửa sổ, và order_by_clause là một mệnh đề ORDER BY tùy chọn như được mô tả bên dưới.

Mẫu ban đầu của biểu thức tổng hợp gọi sự tổng hợp một lần cho từng hàng đầu vào. Mẫu thứ 2 là y hệt như mẫu đầu, vì ALL là mặc định. Mẫu thứ 3 gọi tổng hợp một lần cho từng giá trị duy nhất của biểu thức (hoặc tập hợp duy nhất các giá trị, cho nhiều biểu thức) được thấy trong các hàng đầu vào. Mẫu cuối cùng gọi tổng hợp một lần cho từng hàng đầu vào; vì không có giá trị đầu vào cụ thể nào được chỉ định, nó thường chỉ hữu dụng cho hàm tổng hợp đếm count(*).

Hầu hết các hàm tổng hợp bỏ qua các đầu vào null, nên các hàng trong đó một hoặc nhiều biểu thức hơn có null sẽ bị bỏ qua. Điều này có thể được giả thiết là đúng, trừ phi điều khác được chỉ định, cho tất cả các tổng hợp được xây dựng sẵn.

Ví dụ, count(*) cho tổng số các hàng đầu vào; count(f1) cho số các hàng đầu vào theo đó f1 không là null, vì count bỏ qua null; và count(distinct f1) cho số các giá trị duy nhất không là null của f1.

Thông thường, các hàng đầu vào được nuôi dưỡng cho hàm tổng hợp theo một trật tự không được chỉ định trước. Trong nhiều trường hợp điều này không là vấn đề; ví dụ, min tạo ra kết quả y hệt bất kể trật tự nào nó nhận được các đầu vào. Tuy nhiên, một số hàm tổng hợp (như array_agg và string_agg) tạo ra các kết quả phụ thuộc vào thứ tự các hàng đầu vào. Khi sử dụng một tổng hợp như vậy, tùy chọn order_by_clause có thể được sử dụng để chỉ định trật tự mong muốn. Tùy chọn order_by_clause có cú pháp y hệt như đối với một mệnh đề mức truy vấn ORDER BY, như được mô tả trong Phần 7.5, ngoại trừ là các biểu thức của nó luôn chỉ là các biểu thức và không thể là các tên

cột đầu ra hoặc các con số. Ví dụ:

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

Khi làm việc với nhiều hàm tổng hợp nhiều đối số, lưu ý là mệnh đề ORDER BY đi sau tất cả các đối số tổng hợp. Ví dụ, hãy viết thế này:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

không viết thế này:

```
SELECT string_agg(a ORDER BY a, ',') FROM table; -- incorrect
```

Cái sau là đúng về cú pháp, nhưng nó thể hiện một lời gọi của một hàm tổng hợp một đối số duy nhất với 2 khóa ORDER BY (khóa thứ 2 khá là vô dụng vì nó là một hằng).

Nếu DISTINCT được chỉ định thêm vào một tùy chọn order_by_clause, thì tất cả các biểu thức ORDER BY phải khớp với các đối số thông thường của tổng hợp đó; đó là, bạn không thể sắp xếp trong một biểu thức mà không được đưa vào trong danh sách DISTINCT.

> **Lưu ý**: Khả năng để chỉ định cả DISTINCT và ORDER BY trong một hàm tổng hợp là một mở rộng của PostgreSQL.

Các hàm tổng hợp được xác định trước được mô tả trong Phần 9.18. Các hàm tổng hợp khác có thể được người sử dụng bổ sung thêm vào.

Một biểu thức tổng hợp chỉ có thể xuất hiện trong danh sách kết quả hoặc mệnh đề HAVING của một lệnh SELECT. Là cấm kỵ trong các mệnh đề khác, như WHERE, vì các mệnh đề đó được đánh giá về logic trước khi các kết quả của các tổng hợp được hình thành.

Khi một biểu thức tổng hợp xuất hiện trong một truy vấn con (xem Phần 4.2.10 và Phần 9.20), thì tổng hợp đó thường được đánh giá đối với các hàng của truy vấn phụ đó. Nhưng một ngoại lệ xảy ra nếu các đối số của tổng hợp đó chỉ có các biến mức vòng ngoài: thì tổng hợp đó sau đó thuộc về mức vòng ngoài gần nhất, và được đánh giá đối với các hàng của truy vấn đó. Biểu thức tổng hợp như một tổng thể sau đó là một tham chiếu vòng ngoài cho truy vấn con mà nó xuất hiện trong đó, và hành động như một hằng đối với bất kỳ sự đánh giá nào đối với truy vấn con đó. Hạn chế về việc xuất hiện này chỉ trong danh sách kết quả hoặc mệnh đề HAVING áp dụng với lưu ý đối với mức truy vấn mà tổng hợp đó thuộc về.

4.2.8. Lời gọi hàm cửa sổ

Một lời gọi hàm cửa sổ đại diện cho ứng dụng của một hàm dạng tổng hợp đối với một số phần của các hàng được một truy vấn lựa chọn. Không giống như các lời gọi hàm tổng hợp thông thường, điều này không bị trói vào việc tạo nhóm các hàng được lựa chọn vào một hàng đầu ra duy nhất - từng hàng vẫn giữ là tách biệt nhau ở đầu ra của truy vấn. Tuy nhiên hàm cửa sổ có khả năng quét tất cả các hàng mà có thể là một phần của nhóm các hàng hiện hành mà tuân theo đặc tả tạo thành nhóm (danh sách PARTITION BY) của lời gọi hàm cửa sổ. Cú pháp của một lời gọi hàm cửa sổ là cú pháp như sau:

function_name ([expression [, expression ...]]) OVER (window_definition)
function_name ([expression [, expression ...]]) OVER window_name
*function_name (*) OVER (window_definition)*
*function_name (*) OVER window_name*

trong đó window_definition có cú pháp:

[existing_window_name]
[PARTITION BY expression [, ...]]
[ORDER BY expression [ASC | DESC | USING operator] [NULLS { FIRST | LAST }] [, ...]]
[frame_clause]

và tùy chọn frame_clause có thể là một trong số

[RANGE | ROWS] frame_start
[RANGE | ROWS] BETWEEN frame_start AND frame_end

trong đó frame_start và frame_end có thể là một trong số

UNBOUNDED PRECEDING
value PRECEDING
CURRENT ROW
value FOLLOWING
UNBOUNDED FOLLOWING

Ở đây biểu thức - expression thể hiện bất kỳ biểu thức giá trị nào mà bản thân nó không có các lời gọi hàm cửa sổ. Các danh sách PARTITION BY và ORDER BY về cơ bản có cùng cú pháp và ngữ nghĩa y hệt như các mệnh đề GROUP BY và ORDER BY của toàn bộ truy vấn, ngoại trừ là các biểu thức của chúng luôn chỉ là các biểu thức và không thể là các tên cột đầu ra hoặc các con số. window_name là một tham chiếu tới một đặc tả cửa sổ được đặt tên được xác định trong mệnh đề WINDOW của truy vấn. Các đặc tả cửa sổ được đặt tên thường được tham chiếu với chỉ OVER window_name, nhưng nó cũng có khả năng để viết tên một cửa sổ vào trong các dấu ngoặc đơn và sau đó cung cấp tùy ý cho một mệnh đề sắp xếp và/hoặc mệnh đề khung (cửa sổ được tham chiếu phải không có các mệnh đề đó, nếu chúng được cung cấp ở đây). Cú pháp sau ở đây tuân theo cùng các qui tắc y hệt như việc sửa đổi tên của một cửa sổ đang tồn tại bên trong mệnh đề WINDOW; xem trang tham chiếu SELECT để có thêm các chi tiết.

frame_clause chỉ định tập hợp các hàng tạo thành khung cửa sổ, đối với các hàm cửa sổ mà hành động trong khung (frame) thay vì toàn bộ phân vùng. Nếu frame_end bị bỏ qua thì các mặc định là cho hàng hiện hành CURRENT ROW. Những hạn chế là việc frame_start không thể là tuân theo vô giới hạn UNBOUNDED FOLLOWING, frame_end không thể là có trước không giới hạn UNBOUNDED PRECEDING, và lựa chọn frame_end không thể xuất hiện sớm hơn trong danh sách ở trên so với lựa chọn frame_start - ví dụ RANGE BETWEEN CURRENT ROW AND value PRECEDING là không được phép. Tùy chọn tạo khung mặc định là RANGE UNBOUNDED PRECEDING, nó là y hệt như RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW; nó thiết lập khung để tất cả các hàng từ phân vùng đó khởi tạo qua điểm ngang hàng cuối cùng của hàng hiện hành trong trật tự ORDER BY (nó có nghĩa là tất cả các hàng nếu không có ORDER BY). Nói chung, UNBOUNDED PRECEDING có nghĩa là khung đó bắt đầu với hàng đầu tiên của phân vùng, và tương tự UNBOUNDED FOLLOWING có nghĩa là khung kết thúc với hàng cuối cùng của phân vùng (bất kể chế độ RANGE hay ROWS). Trong chế độ ROWS, CURRENT ROW có nghĩa là khung bắt đầu hoặc kết thúc với hàng hiện hành; nhưng trong chế độ RANGE thì nó có nghĩa là khung bắt đầu và kết thúc với điểm ngang hàng đầu tiên hoặc cuối cùng

của hàng hiện hành theo trật tự ORDER BY. Các trường hợp giá trị PRECEDING và giá trị FOLLOWING hiện chỉ được phép trong chế độ ROWS. Chúng chi ra rằng khung bắt đầu hoặc kết thúc với hàng mà có nhiều hàng trước hoặc sau hàng hiện hành. Giá trị value phải là một biểu thức số nguyên không chứa bất kỳ biến, các hàm tổng hợp hoặc các hàm cửa sổ nào. Giá trị đó phải không là null hoặc âm; nhưng nó có thể là 0, khi chọn bản thân hàng hiện hành.

Các hàm cửa sổ được xây dựng sẵn được mô tả trong Bảng 9-44. Các hàm cửa sổ khác có thể được người sử dụng cho thêm vào. Hơn nữa, bất kỳ hàm tổng hợp được xây dựng sẵn hay do người sử dụng định nghĩa cũng có thể được sử dụng như một hàm cửa sổ.

Các cú pháp sử dụng dấu * được sử dụng để gọi các hàm tổng hợp ít tham số như các hàm cửa sổ, ví dụ count(*) OVER (PARTITION BY x ORDER BY y). * không được sử dụng một cách thông thường cho các hàm cửa sổ không tổng hợp. Các hàm cửa sổ tổng hợp, không giống như các hàm tổng hợp, không cho phép DISTINCT hoặc ORDER BY được sử dụng bên trong danh sách đối số hàm.

Các lời gọi hàm cửa sổ chỉ được phép trong danh sách SELECT và mệnh đề ORDER BY của truy vấn.

Nhiều thông tin hơn về các hàm cửa sổ có thể thấy trong Phần 3.5, Phần 9.19 và Phần 7.2.4.

4.2.9. Cast dạng

Một cast dạng chỉ định một sự chuyển đổi từ dạng dữ liệu này sang dạng dữ liệu khác. PostgreSQL chấp nhận 2 cú pháp tương đương nhau cho các cast dạng:

```
CAST ( expression AS type )
expression::type
```

Cú pháp của CAST tuân thủ SQL; cú pháp với :: là sử dụng theo lịch sử của PostgreSQL.

Khi một cast được áp dụng cho một biểu thức giá trị của một dạng được biết, nó thể hiện một biến đổi dạng thời gian thực. Cast sẽ chỉ thành công nếu một hoạt động biến đổi dạng phù hợp từng được xác định. Lưu ý điều này là hơi khác với sử dụng các cast với các hằng, được chỉ ra ở Phần 4.1.2.7. Một cast được áp dụng cho một hằng chuỗi để tự nhiên thể hiện sự chỉ định trong nội bộ một dạng đối với giá trị hằng theo nghĩa đen, và vì thế nó sẽ thành công đối với bất kỳ dạng nào (nếu các nội dung của hằng chuỗi là cú pháp đầu vào chấp nhận được cho dạng dữ liệu đó).

Một cast dạng rõ ràng có thể thường được làm mờ đi nếu không có sự mù mờ như đối với dạng mà một biểu thức giá trị phải sinh ra (ví dụ, khi nó được chỉ định tới một cột của bảng); hệ thống sẽ tự động áp dụng một cast dạng trong các trường hợp như vậy. Tuy nhiên, việc cast tự động chỉ được thực hiện cho các cast được đánh dấu "OK to apply implicitly" ("OK để áp dụng một cách ẩn") trong các catalog hệ thống. Các cast khác phải được gọi với cú pháp của việc cast rõ ràng. Hạn chế này được mong đợi để ngăn chặn sự biến đổi gây ngạc nhiên khi được áp dụng một cách âm thầm.

Cũng có khả năng để chỉ định một cast dạng bằng việc sử dụng một cú pháp giống hàm:

```
typename ( expression )
```

Tuy nhiên, điều này chỉ làm việc cho các dạng mà các tên của chúng cũng là hợp lệ như các tên

hàm. Ví dụ, độ chính xác đúp double precision không thể được sử dụng theo cách này, nhưng thứ tương tự là float8 thì có thể. Hơn nữa, các tên interval, time và timestamp chỉ có thể được sử dụng theo cách thức này nếu chúng nằm trong các dấu ngoặc kép, vì các xung đột về cú pháp. Vì thế, sử dụng cú pháp cast giống hàm dẫn tới những sự không nhất quán và có thể nên tránh.

> **Lưu ý**: Cú pháp giống như hàm trong thực tế chỉ là lời gọi hàm. Khi một trong 2 cú pháp cast tiêu chuẩn được sử dụng để thực hiện một biến đổi thời gian thực, thì nó sẽ gọi trong nội bộ một hàm được đăng ký để thực hiện sự biến đổi đó. Theo qui ước, các hàm biến đổi đó có tên y hệt như dạng đầu ra của chúng, và vì thế "cú pháp giống như hàm" không là gì hơn một lời gọi trực tiếp hàm biến đổi nằm bên dưới. Rõ ràng, điều này là thứ gì đó mà một ứng dụng khả chuyển sẽ dựa vào. Để có thêm các chi tiết, xem CREATE CAST.

4.2.10. Truy vấn con vô hướng

Một truy vấn con vô hướng là một truy vấn SELECT thông thường trong các dấu ngoặc đơn mà trả về chính xác một hàng với một cột. (Xem Chương 7 để có thông tin về việc viết các truy vấn). Truy vấn SELECT được thực thi và giá trị được trả về duy nhất được sử dụng trong biểu thức giá trị xung quanh. Là một lỗi nếu sử dụng một truy vấn mà trả về hơn một hàng hoặc hơn một cột như một truy vấn con vô hướng. (Mà nếu, trong quá trình thực thi đặc biệt, truy vấn con không trả về hàng nào, thì không có lỗi; kết quả vô hướng được nhận như là null). Truy vấn con có thể tham chiếu tới các biến từ truy vấn xung quanh, nó sẽ hành động như các hằng trong bất kỳ đánh giá nào của truy vấn con. Cũng xem Phần 9.20 cho các biểu thức khác có liên quan tới các truy vấn con.

Ví dụ, thứ sau đây tìm kiếm dân số của thành phố lớn nhất trong từng bang:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
FROM states;
```

4.2.11. Các cấu trúc mảng

Một cấu trúc mảng là một biểu thức mà xây dựng một giá trị mảng bằng việc sử dụng các giá trị cho các phần tử thành viên của nó.

Một cấu trúc mảng đơn giản bao gồm từ khóa ARRAY, một dấu ngoặc vuông bên trái [, một danh sách các biểu thức (cách nhau bằng dấu phẩy) cho các giá trị phần tử mảng, và cuối cùng một dấu ngoặc vuông phải]. Ví dụ:

```
SELECT ARRAY[1,2,3+4];
array
---------
{1,2,7}
(1 row)
```

Mặc định, dạng phần tử mảng là dạng chung của các biểu thức thành phần, được xác định bằng việc sử dụng cùng các qui tắc y hệt như các cấu trúc UNION hoặc CASE (xem Phần 10.5). Bạn có thể viết đè điều này bằng việc cast rõ ràng cấu trúc mảng tới dạng mong muốn, ví dụ:

```
SELECT ARRAY[1,2,22.7]::integer[];
array
```

```
----------
{1,2,23}
(1 row)
```

Điều này có hiệu ứng y hệt như việc cast từng biểu thức tới dạng phần tử mạng một cách riêng rẽ. Để có nhiều hơn về việc cast, xem Phần 4.2.9.

Các giá trị mảng đa chiều có thể được xây dựng bằng việc lồng các cấu trúc mảng. Trong các cấu trúc vòng trong, từ khóa mảng ARRAY có thể bị bỏ qua. Ví dụ, điều này tạo ra kết quả y hệt:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
array
---------------
{{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
array
---------------
{{1,2},{3,4}}
(1 row)
```

Vì các mảng đa chiều phải là hình chữ nhật, nên các cấu trúc vòng trong ở mức y hệt phải tạo ra các mạng con có các chiều y hệt. Bất kỳ cast nào được áp dụng cho cấu trúc mảng ARRAY vòng ngoài truyền giống một cách tự động cho tất cả cấc cấu trúc vòng trong.

Các phần tử cấu trúc mảng đa chiều có thể là bất kỳ thứ gì có bất kỳ mảng nào có dạng phù hợp hơn, không chỉ một cấu trúc mảng con (sub-ARRAY). Ví dụ:

```
CREATE TABLE arr(f1 int[], f2 int[]);
INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);
SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
array
------------------------------------------------
(1 row)
```

Bạn có thể xây dựng một mảng rỗng, nhưng vì không có khả năng để có một mảng mà không có dạng nào, nên bạn phải cast rõ ràng mảng rỗng của bạn vào dạng mong muốn. Ví dụ:

```
SELECT ARRAY[]::integer[];
array
-------
{}
(1 row)
```

Cũng có khả năng để xây dựng một mảng từ các kết quả của một truy vấn con. Ở dạng này, cấu trúc mảng được viết với từ khóa ARRAY đi theo sau là một truy vấn con nằm trong các dấu ngoặc đơn (không phải các dấu ngoặc vuông). Ví dụ:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
?column?
----------------------------------------------------------
{2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31}
(1 row)
```

Truy vấn con phải trả về một cột duy nhất. Mảng kết quả một chiều sẽ có một phần tử cho từng hàng trong kết quả của truy vấn con, với một dạng phần tử khớp với dạng của cột đầu ra của truy

vấn con đó.

Chỉ số dưới của một giá trị mảng được xây dựng với ARRAY luôn bắt đầu với 1. Để có thêm thông tin về các mảng, xem Phần 8.14.

4.2.12. Cấu trúc hàng

Một cấu trúc hàng là một biểu thức xây dựng một giá trị hàng (cũng được gọi là một giá trị tổng hợp) bằng việc sử dụng các giá trị cho các trường các thành phần của nó. Một cấu trúc hàng bao gồm từ khóa hàng ROW, một dấu ngoặc đơn trái, không hoặc nhiều biểu thức hơn (cách nhau bằng các dấu phẩy cho các giá trị trường hàng, và cuối cùng một dấu ngoặc đơn phải. Ví dụ:

```
SELECT ROW(1,2.5,’this is a test’);
```

Từ khóa hàng ROW là tùy chọn khi có nhiều hơn một biểu thức trong danh sách.

Một cấu trúc hàng có thể bao gồm cú pháp rowvalue.*, nó sẽ được mở rộng tới một danh sách các phần tử của giá trị hàng đó, hệt như xảy ra khi cú pháp .* được sử dụng ở mức đỉnh của một danh sách SELECT. Ví dụ, nếu bảng t có các cột f1 và f2, những thứ này là y hệt:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **Lưu ý**: Trước PostgreSQL phiên bản 8.2, cú pháp .* từng không được mở rộng, vì thế việc viết ROW(t.*, 42) đã tạo ra một hàng 2 trường mà trường đầu tiên của nó từng có giá trị hàng khác. Cách hành xử mới thường hữu dụng hơn. Nếu bạn cần hành xử cũ các giá trị hàng lồng nhau, hãy viết giá trị hàng vòng trong mà không có .*, ví dụ ROW(t, 42).

Mặc định, giá trị được biểu thức ROW tạo ra là ở dạng bản ghi nặc danh. Nếu cần, nó có thể là cast tới dạng tổng hợp được đặt tên - hoặc dạng hàng của một bảng, hoặc dạng tổng hợp được tạo ra với CREATE TYPE AS . Một cast rõ ràng có thể là cần thiết để tránh sự mù mờ. Ví dụ:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS ’SELECT $1.f1’ LANGUAGE SQL;

-- No cast needed since only one getf1() exists (Không cast nào cần thiết vì chỉ một getf1() tồn tại)

SELECT getf1(ROW(1,2.5,’this is a test’));
getf1
-------
1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS ’SELECT $1.f1’ LANGUAGE SQL;
-- Now we need a cast to indicate which function to call: (Bây giờ chúng ta cần một cast để chỉ định hàm nào để gọi:)

SELECT getf1(ROW(1,2.5,’this is a test’));
ERROR: function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,’this is a test’)::mytable);
getf1
```

```
-------
1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
getf1
-------
11
(1 row)
```

Các cấu trúc hàng có thể được sử dụng để xây dựng các giá trị tổng hợp sẽ được lưu trữ trong một cột của bảng dạng tổng hợp, hoặc sẽ được chuyển tới một hàm chấp nhận một tham số tổng hợp. Hơn nữa, có khả năng so sánh 2 giá trị hàng hoặc kiểm thử một hàng với IS NULL hoặc IS NOT NULL, ví dụ:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');


SELECT ROW(table.*) IS NULL FROM table; -- detect all-null rows
```

Để có thêm chi tiết, xem Phần 9.21. Các cấu trúc hàng cũng có thể được sử dụng trong sự kết nối với các truy vấn con, như được thảo luận trong Phần 9.20.

4.2.13. Các qui tắc đánh giá biểu thức

Trật tự đánh giá các biểu thức con không được xác định. Đặc biệt, các đầu vào của một toán tử hoặc hàm không nhất thiết được đánh giá từ trái qua phải hoặc theo bất kỳ trật tự cố định nào khác.

Hơn nữa, nếu kết quả của một biểu thức có thể được xác định bằng sự đánh giá chỉ một số phần của nó, thì các biểu thức con khác có thể sẽ không được đánh giá hoàn toàn. Ví dụ, nếu một người viết:

```
SELECT true OR somefunc();
```

thì somefunc() có thể (có lẽ) không được gọi hoàn toàn. Điều y hệt có thể là trường hợp nếu viết:

```
SELECT somefunc() OR true;
```

Lưu ý rằng điều này không là y hệt nhau như "việc đi đường tắt" từ trái qua phải của các toán tử Boolean được thấy trong một số ngôn ngữ lập trình.

Như là một hệ quả, là không khôn ngoan để sử dụng các hàm với các hiệu ứng phụ như một phần của các biểu thức phức tạp. Đặc biệt nguy hiểm để dựa vào các hiệu ứng phụ hoặc trật tự đánh giá trong các mệnh đề WHERE và HAVING, vì các mệnh đề đó được tái xử lý một cách rộng rãi như một phần của việc phát triển một kế hoạch thực thi. Các biểu thức Boolean (kết hợp của AND /OR /NOT) trong các mệnh đề đó có thể được tổ chức lại theo bất kỳ cách gì mà luật số học Boolean cho phép.

Khi điều cơ bản để ép trật tự đánh giá, thì một cấu trúc CASE (xem Phần 9.16) có thể được sử dụng. Ví dụ, đây là một cách không tin cậy của việc cố gắng tránh chia cho 0 trong một mệnh đề WHERE:

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

Nhưng điều này là an toàn:

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

Một cấu trúc CASE được sử dụng theo cách này sẽ đánh bại các cố gắng tối ưu hóa, vì thế nó nên chỉ được thực hiện khi cần thiết. (Trong ví dụ đặc biệt này, có thể là tốt hơn để đặt vấn đề đó sang bên bằng việc thay vào đó viết y > 1.5*x).

4.3. Gọi hàm

PostgreSQL cho phép các hàm có các tham số có tên sẽ được gọi hoặc bằng ký hiệu vị trí hoặc ký hiệu có tên. Ký hiệu có tên là đặc biệt hữu dụng cho các hàm có một số lượng lớn các tham số, vì nó làm cho các mối liên quan giữa các tham số và các đối số thực rõ ràng và đáng tin cậy hơn. Theo ký hiệu vị trí, một lời gọi hàm được viết với các giá trị đối số của nó theo trật tự y hệt như chúng được định nghĩa trong khai báo hàm. Theo ký hiệu được đặt tên, các đối số sẽ được khớp với các tham số hàm bằng tên và có thể được viết theo bất kỳ trật tự nào.

Theo bất kỳ ký hiệu nào, thì các tham số mà có các giá trị mặc định được đưa ra trong khai báo hàm cũng cần hoàn toàn không được viết trong lời gọi. Nhưng điều này đặc biệt hữu dụng theo ký hiệu được đặt tên, vì bất kỳ sự kết hợp các tham số nào cũng có thể bị bỏ qua; trong khi theo ký hiệu vị trí thì các tham số chỉ có thể bị bỏ qua từ phải qua trái.

PostgreSQL cũng hỗ trợ ký hiệu pha trộn, nó kết hợp ký hiệu vị trí và được đặt tên. Trong trường hợp này, các tham số vị trí được viết trước và các tham số được đặt tên xuất hiện sau chúng.

Các ví dụ sau sẽ minh họa sự sử dụng của tất cả 3 ký hiệu đó, sử dụng định nghĩa hàm sau:

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

Hàm concat_lower_or_upper có 2 tham số bắt buộc, a và b. Bổ sung thêm, có một tham số tùy chọn chữ hoa uppercase mà mặc định là sai – false. Các đầu vào a và b sẽ được ghép nối, và bị ép vào hoặc chữ thường hoặc chữ hoa, phụ thuộc vào tham số uppercase. Các chi tiết còn lại của định nghĩa hàm này là không quan trọng ở đây (xem Chương 35 để có thêm thông tin).

4.3.1. Sử dụng ký hiệu vị trí

Ký hiệu vị trí là cơ chế truyền thống cho việc truyền các đối số tới các hàm trong PostgreSQL. Một ví dụ là:

```
SELECT concat_lower_or_upper('Hello', 'World', true);
concat_lower_or_upper
-----------------------
HELLO WORLD
(1 row)
```

Tất cả các đối số được chỉ định theo trật tự. Kết quả là chữ hoa vì uppercase được chỉ định là đúng –

true. Ví dụ khác là:

```
SELECT concat_lower_or_upper('Hello', 'World');
concat_lower_or_upper
----------------------
hello world
(1 row)
```

Ở đây, tham số uppercase bị bỏ qua, nên nó nhận giá trị mặc định của nó là sai - false, trả về chữ thường ở đầu ra. Theo ký hiệu vị trí thì các đối số có thể bị bỏ qua từ phải qua trái cho tới khi chúng có được các mặc định.

4.3.2. Sử dụng ký hiệu được đặt tên

Theo ký hiệu được đặt tên, từng tên đối số được chỉ định bằng việc sử dụng := để ngăn cách nó với biểu thức đối số. Ví dụ:

```
SELECT concat_lower_or_upper(a := 'Hello', b := 'World');
concat_lower_or_upper
----------------------
hello world
(1 row)
```

Một lần nữa, đối số uppercase đã bị bỏ qua nên nó được thiết lập về false một cách ẩn. Một ưu tiên của sử dụng ký hiệu được đặt tên là các đối số có thể được chỉ định theo bất kỳ trật tự nào, ví dụ:

```
SELECT concat_lower_or_upper(a := 'Hello', b := 'World', uppercase := true);
concat_lower_or_upper
----------------------
HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
concat_lower_or_upper
----------------------
HELLO WORLD
(1 row)
```

4.3.3. Sử dụng ký hiệu pha trộn

Ký hiệu pha trộn kết hợp ký hiệu vị trí và được đặt tên. Tuy nhiên, như đã được nhắc tới, các đối số được đặt tên không thể đi trước các đối số vị trí. Ví dụ:

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase := true);
concat_lower_or_upper
----------------------
HELLO WORLD
(1 row)
```

Trong truy vấn ở trên, các đối số a và b được chỉ định theo vị trí, trong khi uppercase được chỉ định theo tên. Trong ví dụ này, điều đó bổ sung thêm được ít, ngoại trừ tài liệu. Với một hàm phức tạp hơn có nhiều tham số mà có các giá trị mặc định, thì ký hiệu được đặt tên hoặc pha trộn có thể tiết kiệm được nhiều việc viết và làm giảm các cơ hội sinh lỗi.

Chương 5. Định nghĩa dữ liệu

Chương này đề cập tới cách tạo các cấu trúc cơ sở dữ liệu mà sẽ lưu trữ các dữ liệu. Trong cơ sở dữ liệu quan hệ, các dữ liệu thô được lưu trữ trong các bảng, vì thế phần lớn chương này dành cho việc giải thích cách mà các bảng sẽ được tạo ra và được sửa đổi như thế nào và các tính năng nào là sẵn sàng để kiểm soát những dữ liệu gì được lưu trữ trong các bảng đó. Tiếp đến, chúng ta sẽ thảo luận về cách mà các bảng có thể được tổ chức trong các sơ đồ, và cách mà các quyền ưu tiên có thể được chỉ định cho các bảng. Cuối cùng, chúng ta sẽ nhìn nhanh vào các tính năng khác mà có ảnh hưởng tới lưu trữ cơ sở dữ liệu, như tính kế thừa, các kiểu nhìn, các hàm, và các trigger.

5.1. Cơ bản về bảng

Một bảng trong cơ sở dữ liệu quan hệ rất giống một bảng trên giấy: nó bao gồm các hàng và các cột. Số lượng và thứ tự các cột là cố định, và từng cột đều có tên. Số hàng là biến đổi - nó phản ánh có bao nhiêu dữ liệu được lưu trữ ở thời điểm được đưa ra. SQL không có bất kỳ đảm bảo nào về trật tự của các hàng trong một bảng. Khi một bảng được đọc, các hàng sẽ xuất hiện theo một trật tự không được xác định, trừ phi việc sắp xếp được yêu cầu rõ ràng. Điều này được đề cập tới trong Chương 7. Hơn nữa, SQL không chỉ định các mã định danh duy nhất cho các hàng, nên có khả năng sẽ có vài hàng hoàn toàn giống hệt như nhau trong một bảng. Đây là hệ quả của mô hình toán học nằm bên dưới SQL nhưng thường là không mong muốn. Phần sau của chương này chúng ta sẽ xem cách làm việc với vấn đề này.

Từng cột đều có một dạng dữ liệu. Dạng dữ liệu ràng buộc tập hợp các giá trị có khả năng mà có thể được sử dụng để tính toán. Ví dụ, một cột được khai báo sẽ là dạng số thì sẽ không chấp nhận các chuỗi văn bản tùy ý, và các dữ liệu được lưu trữ trong một cột như vậy có thể được sử dụng cho các tính toán toán học. Ngược lại, một cột được khai báo ở dạng chuỗi ký tự sẽ chấp nhận hầu hết bất kỳ dạng dữ liệu nào nhưng tự nó không sử dụng được để thực hiện các phép tính toán học, dù các hoạt động khác như ghép nối chuỗi là sẵn sàng.

PostgreSQL bao gồm một tập hợp lớn các dạng dữ liệu được xây dựng sẵn mà phù hợp cho nhiều ứng dụng. Người sử dụng cũng có thể định nghĩa các dạng dữ liệu của riêng họ. Hầu hết các dạng dữ liệu được xây dựng sẵn có các tên và ngữ nghĩa rõ ràng, nên chúng ta sẽ hoãn giải thích chi tiết cho Chương 8. Một số dạng dữ liệu được sử dụng thường xuyên là số nguyên integer cho toàn bộ các số, numeric cho các số thập phân có thể, text cho các chuỗi ký tự, date cho ngày tháng, time cho các giá trị thời gian trong ngày và timestamp cho các giá trị chứa cả ngày tháng và thời gian.

Để tạo một bảng, bạn sử dụng khéo léo lệnh có tên là tạo bảng CREATE TABLE. Trong lệnh này bạn chỉ định ít nhất một tên cho bảng mới, các tên của các cột và dạng dữ liệu của từng cột. Ví dụ:

```
CREATE TABLE my_first_table (
        first_column text,
        second_column integer
);
```

Điều này tạo ra một bảng có tên là my_first_table với 2 cột. Cột đầu có tên là first_column và có dạng

dữ liệu là text; cột thứ 2 có tên là second_column và dạng số nguyên integer. Các tên bảng và cột tuân theo cú pháp của mã định danh trong Phần 4.1.1. Các tên đó thường cũng là các mã định danh, nhưng có một số ngoại lệ. Lưu ý rằng danh sách các cột được tách bạch nhau bằng dấu phẩy và được các dấu ngoặc đơn bao bọc.

Tất nhiên, ví dụ trước đó đã được trù tính nhiều. Thông thường, bạn có lẽ đưa ra các cái tên cho các bảng và các cột của bạn mà truyền đạt được dạng dữ liệu chúng lưu trữ. Nên hãy xem ví dụ thực tiễn hơn:

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric
);
```

(Dạng numeric có thể lưu trữ các thành phần thập phân, điển hình như là số tiền).

> **Mẹo**: Khi bạn tạo ra nhiều bảng có liên quan với nhau, là khôn ngoan để chọn một mẫu đặt tên nhất quán cho các bảng và cột. Ví dụ, có một lựa chọn sử dụng các danh từ số ít hoặc số nhiều cho các tên bảng, cả 2 dạng đều được một số nhà lý thuyết ưa thích.

Có một giới hạn về số lượng cột mà một bảng có thể có. Phụ thuộc vào các dạng cột, nó vào khoảng từ 250 cho tới 1600. Tuy nhiên, việc xác định một bảng với bất kỳ số lượng cột nào gần với các con số đó là vô dụng cao độ và thường là một thiết kế đáng ngờ.

Nếu bạn không còn cần một bảng nữa, bạn có thể loại bỏ nó bằng việc sử dụng lệnh bỏ bảng DROP TABLE. Ví dụ:

```
DROP TABLE my_first_table;
DROP TABLE products;
```

Việc cố bỏ một bảng không tồn tại là một lỗi. Dù vậy, là phổ biến trong các tệp script SQL để cố gắng một cách vô điều kiện bỏ từng bảng trước khi tạo ra nó, bỏ qua bất kỳ thông điệp lỗi nào, sao cho script đó làm việc bất kể bảng đó có hay không tồn tại. (Nếu bạn thích, bạn có thể sử dụng biến bỏ bảng nếu tồn tại DROP TABLE IF EXISTS để tránh các thông điệp lỗi, nhưng điều này không phải là tiêu chuẩn SQL).

Nếu bạn cần sửa đổi một bảng mà đã tồn tại rồi, xem Phần 5.5 ở sau trong chương này.

Với các công cụ được thảo luận cho tới nay, bạn có thể hoàn toàn tạo ra được các bảng hoạt động đầy đủ. Phần còn lại của chương này quan tâm tới việc bổ sung thêm các tính năng cho định nghĩa bảng để đảm bảo tính toàn vẹn dữ liệu, an ninnh hoặc sự thuận tiện. Nếu bạn hăng hái điền các dữ liệu vào bảng bây giờ thì bạn có thể bỏ qua Chương 6 và đọc phần còn lại của chương này sau.

5.2. Các giá trị mặc định

Một cột có thể được chỉ định một giá trị mặc định. Khi một hàng mới được tạo ra và không giá trị nào được chỉ định cho một số cột, thì các cột đó sẽ được điền với các giá trị mặc định tương ứng với chúng. Một lệnh điều khiển dữ liệu cũng có thể yêu cầu rõ ràng là một cột sẽ được thiết lập với giá trị mặc định của nó, không cần biết giá trị đó là gì. (Các chi tiết về các lệnh điều khiển dữ liệu nằm

ở Chương 6).

Nếu không giá trị mặc định nào được khai báo rõ ràng, thì giá trị mặc định là giá trị null. Điều này thường có ý nghĩa vì một giá trị null có thể được xem xét để trình bày các dữ liệu không biết.

Trong định nghĩa một bảng, các giá trị mặc định có thể được liệt kê sau dạng dữ liệu cột. Ví dụ:

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric DEFAULT 9.99
);
```

Giá trị mặc định có thể là một biểu thức, nó sẽ được đánh giá bất kỳ khi nào giá trị mặc định được chèn vào (không phải khi nào bảng được tạo ra). Một ví dụ phổ biến là đối với một cột timestamp để có một mặc định của dấu thời gian hiện hành CURRENT_TIMESTAMP, sao cho nó có thiết lập thời gian khi chèn hàng vào. Ví dụ phổ biến khác đang tạo ra một "số tuần tự" cho từng hàng. Trong PostgreSQL điều này thường được thực hiện bằng một số thứ giống như:

```
CREATE TABLE products (
        product_no integer DEFAULT nextval('products_product_no_seq'),
        ...
);
```

trong đó hàm nextval() cung cấp các giá trị thành công từ một đối tượng tuần tự (xem Phần 9.15). Đối số này là đủ phổ biến để có được sự tốc ký đặc biệt cho nó.

```
CREATE TABLE products (
        product_no SERIAL,
        ...
);
```

Tốc ký tuần tự SERIAL được thảo luận xa hơn ở Phần 8.1.4.

5.3. Ràng buộc

Các dạng dữ liệu là cách để giới hạn kiểu dữ liệu có thể được lưu trữ trong một bảng. Tuy nhiên, đối với nhiều ứng dụng, sự ràng buộc mà chúng đưa ra là quá kém. Ví dụ, một cột chứa giá một sản phẩm có lẽ chỉ nên có các giá trị dương. Nhưng không có dạng dữ liệu tiêu chuẩn nào chấp nhận chỉ các số dương cả. Vấn đề khác như bạn có thể muốn ràng buộc các dữ liệu của cột với lưu ý về các cột và hàng khác. Ví dụ, trong một bảng có thông tin sản phẩm, sẽ chỉ nên có một hàng cho từng số lượng sản phẩm.

Vì điều này, SQL cho phép bạn định nghĩa các ràng buộc trong các cột và bảng. Các ràng buộc cho bạn sự kiểm soát càng lớn càng tốt theo bạn muốn đối với các dữ liệu trong các bảng của bạn. Nếu một người sử dụng định lưu trữ các dữ liệu trong một cột mà có thể vi phạm một ràng buộc, thì lỗi sẽ nảy sinh. Điều này áp dụng thậm chí nếu giá trị đó tới từ định nghĩa giá trị mặc định.

5.3.1. Ràng buộc kiểm tra

Một ràng buộc kiểm tra là dạng ràng buộc phổ biến nhất. Nó cho phép bạn chỉ định rằng giá trị đó

trong một cột nhất định phải thỏa mãn một biểu thức Boolean (giá trị đúng – true). Ví dụ, để yêu cầu các giá thành sản phẩm phải dương, bạn có thể sử dụng:

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric CHECK (price > 0)
);
```

Như bạn thấy, định nghĩa ràng buộc tới sau dạng dữ liệu, hệt như các định nghĩa giá trị mặc định. Các giá trị và các ràng buộc mặc định có thể được liệt kê theo bất kỳ trật tự nào. Một ràng buộc kiểm tra bao gồm từ khóa kiểm tra CHECK theo sau là một biểu thức trong các dấu ngoặc đơn. Biểu thức ràng buộc kiểm tra sẽ có liên quan tới cột có ràng buộc, nếu không thì ràng buộc đó có thể cũng sẽ không có nhiều ý nghĩa.

Bạn cũng có thể trao cho ràng buộc đó một cái tên. Điều này làm rõ thêm cho các thông điệp lỗi và cho phép bạn tham chiếu tới ràng buộc đó khi bạn cần thay đổi nó. Cú pháp là:

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

Vì thế, để chỉ định một ràng buộc có tên, hãy sử dụng từ khóa CONSTRAINT theo sau là một mã định danh, theo sau nữa là định nghĩa ràng buộc đó. (Nếu bạn không chỉ định tên một ràng buộc theo cách này, thì hệ thống chọn một tên cho bạn).

Một ràng buộc kiểm tra cũng có thể tham chiếu tới vài cột. Hãy nói bạn lưu trữ một giá thông thường và một giá có triết khấu, và bạn muốn đảm bảo rằng giá triết khấu là thấp hơn so với giá thông thường.

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric CHECK (price > 0),
        discounted_price numeric CHECK (discounted_price > 0),
        CHECK (price > discounted_price)
);
```

Hai ràng buộc đầu tiên sẽ trông quen. Ràng buộc thứ 3 sử dụng một cú pháp mới. Nó không được gắn vào cột cụ thể nào, thay vào đó nó xuất hiện như một khoản riêng biệt trong danh sách cột được cách nhau bằng dấu phẩy. Các định nghĩa cột và các định nghĩa ràng buộc đó có thể được liệt kê theo trật tự pha trộn.

Chúng tôi nói rằng 2 ràng buộc đầu tiên là các ràng buộc cột, trong khi ràng buộc thứ 3 là một ràng buộc bảng vì nó được viết tách biệt với bất kỳ định nghĩa cột nào. Các ràng buộc cột cũng có thể được viết như các ràng buộc bảng, trong khi ngược lại có khả năng là không cần thiết, vì một ràng buộc cột được cho là để tham chiếu tới chỉ cột mà nó được gắn tới. (PostgreSQL không ép tuân thủ qui tắc đó, nhưng bạn nên tuân theo nó nếu bạn muốn các định nghĩa bảng của bạn làm việc được với các hệ thống bảng khác). Ví dụ ở trên cũng có thể được viết là:

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric,
        CHECK (price > 0),
        discounted_price numeric,
        CHECK (discounted_price > 0),
        CHECK (price > discounted_price)
);
```

hoặc thậm chí:

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric CHECK (price > 0),
        discounted_price numeric,
        CHECK (discounted_price > 0 AND price > discounted_price)
);
```

Điều đó chỉ là vấn đề sở thích.

Các tên có thể được chỉ định cho các ràng buộc bảng theo cách y hệt như các ràng buộc cột:

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric,
        CHECK (price > 0),
        discounted_price numeric,
        CHECK (discounted_price > 0),
        CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

Đáng lưu ý rằng một ràng buộc kiểm tra được thỏa mãn nếu biểu thức kiểm tra đánh giá thành giá trị đúng hoặc null. Vì hầu hết các biểu thức sẽ đánh giá về giá trị null nếu bất kỳ toán hạng nào là null, chúng sẽ không ngăn cản các giá trị null trong các cột bị ràng buộc. Để đảm bảo rằng một cột không có các giá trị null, thì các ràng buộc không null (not null) được mô tả trong phần tiếp sau có thể được sử dụng.

5.3.2. Ràng buộc không null

Một ràng buộc không null đơn giản chỉ định rằng một cột phải không giả thiết giá trị null. Ví dụ:

```
CREATE TABLE products (
        product_no integer NOT NULL,
        name text NOT NULL,
        price numeric
);
```

Một ràng buộc không null luôn được viết như một ràng buộc cột. Một ràng buộc không null, về chức năng, tương đương với việc tạo ra một ràng buộc kiểm tra CHECK (column_name IS NOT NULL), nhưng trong PostgreSQL việc tạo ra một ràng buộc rõ ràng không null là hiệu quả hơn. Yếu điểm là bạn không thể trao các tên rõ ràng cho các ràng buộc không null được tạo ra theo cách này.

Tất nhiên, một cột có thể có nhiều hơn một ràng buộc. Hãy viết các ràng buộc từng cái một, cái này sau cái kia:

```
CREATE TABLE products (
        product_no integer NOT NULL,
        name text NOT NULL,
        price numeric NOT NULL CHECK (price > 0)
);
```

Trật tự không là vấn đề. Không nhất thiết phải xác định với trật tự nào các ràng buộc được kiểm tra.

Ràng buộc NOT NULL có một sự nghịch đảo: ràng buộc NULL. Điều này không có nghĩa là cột phải là null, mà có thể chắc chắn là vô dụng. Thay vào đó, điều này đơn giản chọn hành vi mặc định mà cột có thể là null. Ràng buộc NULL không thể hiện trong tiêu chuẩn SQL và không nên sử dụng trong các ứng dụng khả chuyển. (Nó chỉ được đưa vào PostgreSQL để tương thích được với một số hệ thống cơ sở dữ liệu khác). Tuy nhiên, một số người sử dụng, thích điều này vì nó làm dễ dàng để hoán đổi ràng buộc trong tệp script.

Ví dụ, bạn có thể bắt đầu với:

```
CREATE TABLE products (
        product_no integer NULL,
        name text NULL,
        price numeric NULL
);
```

và sau đó chèn từ khóa NOT vào bất kỳ đâu bạn muốn.

> **Mẹo**: Trong hầu hết các thiết kế cơ sở dữ liệu, đa số các cột sẽ được đánh dấu là không null.

5.3.3. Ràng buộc độc nhất

Các ràng buộc độc nhất đảm bảo rằng các dữ liệu có trong một cột hoặc một nhóm các cột là độc nhất với lưu ý tới tất cả các hàng trong bảng. Cú pháp là:

```
CREATE TABLE products (
        product_no integer UNIQUE,
        name text,
        price numeric
);
```

khi được viết như một ràng buộc cột, và:

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric,
        UNIQUE (product_no)
);
```

khi được viết như một ràng buộc bảng.

Nếu một ràng buộc độc nhất tham chiếu tới một nhóm các cột, thì các cột đó được liệt kê và được tách biệt nhau bằng các dấu phẩy:

```
CREATE TABLE example (
        a integer,
        b integer,
        c integer,
        UNIQUE (a, c)
);
```

Điều này chỉ định rằng sự kết hợp các giá trị trong các cột được chỉ định là độc nhất xuyên khắp toàn bộ bảng, dù bất kỳ một trong số các cột nào cần không (và thường sẽ không) là độc nhất.

Bạn có thể chỉ định tên của riêng bạn cho một ràng buộc độc nhất, theo cách thông thường:

```
CREATE TABLE products (
        product_no integer CONSTRAINT must_be_different UNIQUE,
        name text,
        price numeric
);
```

Việc bổ sung thêm một ràng buộc độc nhất sẽ tự động tạo ra một chỉ số B-tree độc nhất trong cột hoặc nhóm các cột được sử dụng trong ràng buộc đó.

Nói chung, một ràng buộc độc nhất bị vi phạm khi có hơn một hàng trong bảng nơi mà các giá trị của tất cả các cột được đưa vào trong ràng buộc đó là bằng như nhau. Tuy nhiên, 2 giá trị null không được xem là bằng nhau trong so sánh này. Điều đó có nghĩa là thậm chí trong sự hiện diện của một ràng buộc độc nhất thì vẫn có khả năng để lưu trữ các hàng đúp bản mà có một giá trị null trong ít nhất một trong các cột bị ràng buộc. Hành vi này khẳng định đối với tiêu chuẩn SQL, nhưng chúng ta có nghe rằng các cơ sở dữ liệu SQL khác có thể không tuân theo qui tắc này. Vì thế hãy thận trọng khi phát triển các ứng dụng mà có ý định sẽ là khả chuyển.

5.3.4. Khóa chủ

Về mặt kỹ thuật, một ràng buộc khóa chủ (Primary Key) đơn giản là một sự kết hợp của một ràng buộc độc nhất và một ràng buộc không null. Vì thế, 2 định nghĩa bảng sau đây chấp nhận các dữ liệu hệt nhau:

```
CREATE TABLE products (
        product_no integer UNIQUE NOT NULL,
        name text,
        price numeric
);
CREATE TABLE products (
        product_no integer PRIMARY KEY,
        name text,
        price numeric
);
```

Các khóa chủ cũng có thể ràng buộc nhiều hơn một cột; cú pháp là tương tự như đối với các ràng buộc độc nhất:

```
CREATE TABLE example (
        a integer,
        b integer,
        c integer,
        PRIMARY KEY (a, c)
);
```

Một khóa chủ chỉ định rằng một cột hoặc nhóm các cột có thể được sử dụng như một mã định danh độc nhất cho các hàng trong bảng. (Đây là hệ quả trực tiếp của định nghĩa một khóa chủ. Lưu ý là một ràng buộc độc nhất sẽ không, tự bản thân nó, cung cấp một mã định danh độc nhất vì nó không loại trừ các giá trị null). Điều này là hữu dụng cả cho các mục đích làm tài liệu và cho các ứng dụng

máy trạm. Ví dụ, một ứng dụng giao diện đồ họa cho người sử dụng – GUI mà cho phép sửa đổi các giá trị hàng có khả năng cần biết khóa chủ của một bảng để có khả năng xác định một cách độc nhất các hàng.

Bổ sung thêm vào một khóa chủ sẽ tự động tạo ra một chỉ số B-tree độc nhất trong cột hoặc nhóm các cột được sử dụng trong khóa chủ đó.

Một bảng có thể có nhiều nhất một khóa chủ. (Có thể là bất kỳ số nào trong các ràng buộc độc nhất và không null, về chức năng, nó là điều y hệt, nhưng chỉ một có thể được xác định như là khóa chủ). Lý thuyết cơ sở dữ liệu quan hệ chỉ ra rằng mỗi bảng phải có một khóa chủ. Qui tắc này không bị PostgreSQL ép tuân thủ, nhưng thường tốt nhất phải tuân theo nó.

5.3.5. Khóa ngoại

Một ràng buộc khóa ngoại (Foreign Key) chỉ định rằng các giá trị trong một cột (hoặc một nhóm các cột) phải khớp với các giá trị xuất hiện trong một số hàng của bảng khác. Chúng ta nói điều này duy trì tính toàn vẹn tham chiếu giữa 2 bảng có liên quan.

Nói bạn có bảng sản phẩm (product) mà chúng ta đã sử dụng vài lần rồi:

```
CREATE TABLE products (
        product_no integer PRIMARY KEY,
        name text,
        price numeric
);
```

Hãy cũng giả thiết bạn có một bảng lưu trữ các đơn hàng của các sản phẩm đó. Chúng ta muốn đảm bảo rằng bảng các đơn hàng chỉ chứa các đơn hàng của các sản phẩm mà thực sự tồn tại. Vì thế chúng ta định nghĩa một ràng buộc khóa ngoại trong bảng các đơn hàng mà tham chiếu tới bảng các sản phẩm:

```
CREATE TABLE orders (
        order_id integer PRIMARY KEY,
        product_no integer REFERENCES products (product_no),
        quantity integer
);
```

Bây giờ không có khả năng để tạo các đơn hàng với các khoản `product_no` mà không xuất hiện trong bảng các sản phẩm.

Chúng ta nói rằng trong tình huống này thì bảng các đơn hàng là *bảng tham chiếu* và bảng các sản phẩm là *bảng được tham chiếu* tới. Tương tự, có các *cột tham chiếu* và *cột được tham chiếu* tới.

Bạn cũng có thể rút gọn lệnh ở trên thành:

```
CREATE TABLE orders (
        order_id integer PRIMARY KEY,
        product_no integer REFERENCES products,
        quantity integer
);
```

vì thiếu một danh sách cột thì khóa chủ của bảng được tham chiếu sẽ được sử dụng như là (các) cột được tham chiếu.

Một khóa chủ cũng có thể ràng buộc và tham chiếu tới một nhóm các cột. Như thường lệ, nó sau đó cần phải được viết ở dạng ràng buộc bảng. Đây là một ví dụ cú pháp được trù tính trước:

```
CREATE TABLE t1 (
        a integer PRIMARY KEY,
        b integer,
        c integer,
        FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Tất nhiên, số lượng và dạng các cột bị ràng buộc cần phải khớp với số lượng và dạng của các cột được tham chiếu.

Bạn có thể chỉ định tên của riêng bạn cho một ràng buộc khóa ngoại, theo cách thông thường.

Một bảng có thể gồm nhiều hơn một ràng buộc khóa ngoại. Điều này được sử dụng để triển khai các mối quan hệ nhiều - nhiều giữa các bảng. Nói bạn có các bảng về các sản phẩm và các đơn hàng, nhưng bây giờ bạn muốn cho phép một đơn hàng có khả năng có nhiều sản phẩm (mà cấu trúc ở trên đã không cho phép). Bạn có thể sử dụng cấu trúc bảng này:

```
CREATE TABLE products (
        product_no integer PRIMARY KEY,
        name text,
        price numeric
);
CREATE TABLE orders (
        order_id integer PRIMARY KEY,
        shipping_address text,
        ...
);
CREATE TABLE order_items (
        product_no integer REFERENCES products,
        order_id integer REFERENCES orders,
        quantity integer,
        PRIMARY KEY (product_no, order_id)
);
```

Lưu ý rằng khóa chủ chồng lấn với các khóa ngoại trong bảng cuối cùng.

Chúng ta biết rằng các khóa ngoại không cho phép tạo các đơn hàng mà không có liên quan tới bất kỳ sản phẩm nào.

Nhưng điều gì sẽ xảy ra nếu một sản phẩm bị loại bỏ sau khi một đơn hàng đã được tạo ra mà tham chiếu tới nó? SQL cũng cho phép bạn điều khiển điều đó. Bằng trực giác, chúng ta có vài lựa chọn:

- Không cho phép xóa một sản phẩm được tham chiếu
- Cũng xóa được các đơn hàng
- Thứ gì nữa đây?

Để minh họa điều này, hãy triển khai chính sách sau trong ví dụ về mối quan hệ nhiều - nhiều ở trên: khi ai đó muốn loại bỏ một sản phẩm mà vẫn còn được tham chiếu tới từ một đơn hàng (thông qua order_items), chúng ta không cho phép điều này. Nếu ai đó loại bỏ một đơn hàng, các khoản của đơn hàng đó cũng sẽ bị loại bỏ:

```
CREATE TABLE products (
        product_no integer PRIMARY KEY,
        name text,
        price numeric
);
CREATE TABLE orders (
        order_id integer PRIMARY KEY,
        shipping_address text,
        ...
);
CREATE TABLE order_items (
        product_no integer REFERENCES products ON DELETE RESTRICT,
        order_id integer REFERENCES orders ON DELETE CASCADE,
        quantity integer,
        PRIMARY KEY (product_no, order_id)
);
```

Việc hạn chế và xóa theo kiểu thác nước (cascade) là 2 lựa chọn phổ biến nhất. RESTRICT ngăn cản sự xóa một hàng được tham chiếu. NO ACTION có nghĩa là nếu bất kỳ các hàng tham chiếu nào vẫn còn tồn tại khi ràng buộc đó được kiểm tra, thì một lỗi sẽ nảy sinh; đây là hành vi mặc định nếu bạn không chỉ định bất kỳ điều gì. (Sự khác biệt cơ bản giữa 2 lựa chọn đó là NO ACTION cho phép sự kiểm tra sẽ được trì hoãn cho tới sau đó trong giao dịch, trong khi RESTRICT thì không). CASCADE chỉ định rằng khi một hàng được tham chiếu bị xóa, thì (các) hàng tham chiếu tới nó cũng sẽ tự động bị xóa. Có 2 lựa chọn khác: SET NULL và SET DEFAULT. Chúng làm cho các cột tham chiếu sẽ được thiết lập về null hoặc về các giá trị mặc định, một cách tương ứng, khi hàng được tham chiếu bị xóa. Lưu ý rằng chúng không tha thứ cho bạn đối với việc quan sát thấy bất kỳ các ràng buộc nào. Ví dụ, nếu một hành động chỉ định SET DEFAULT nhưng giá trị mặc định có thể không làm thỏa mãn khóa ngoại, thì hành động đó sẽ hỏng.

Tương tự như với ON DELETE cũng có ON UPDATE mà nó được gọi ra khi một cột được tham chiếu được thay đổi (được cập nhật). Các hành động có khả năng là y hệt như nhau.

Vì một lệnh DELETE một hàng khỏi bảng được tham chiếu hoặc một lệnh UPDATE của một cột được tham chiếu sẽ đòi hỏi một sự quét bảng tham chiếu đối với các hàng khớp với giá trị cũ đó, nên thường một ý tưởng tốt là đánh chỉ số cho các cột tham chiếu. Vì điều này không luôn là cần thiết, và có nhiều sự lựa chọn có sẵn về cách để đánh chỉ số, sự khai báo một ràng buộc khóa ngoại không tự động tạo ra một chỉ số trong các cột tham chiếu.

Nhiều thông tin hơn về việc cập nhật và xóa dữ liệu có trong Chương 6.

Cuối cùng, chúng ta nên lưu ý rằng một khóa ngoại phải tham chiếu tới các cột mà hoặc là một khóa chủ hoặc tạo thành một ràng buộc độc nhất. Nếu khóa ngoại đó tham chiếu tới một ràng buộc độc nhất, thì sẽ có một số khả năng bổ sung liên quan tới việc các giá trị null sẽ khớp được như thế nào. Chúng sẽ được giải thích trong tài liệu tham chiếu của lệnh tạo bảng CREATE TABLE.

5.3.6. Ràng buộc loại trừ

Các ràng buộc loại trừ đảm bảo rằng nếu bất kỳ 2 hàng nào được so sánh trong các cột đặc biệt hoặc

các biểu thức có sử dụng các toán tử đặc biệt, thì ít nhất một trong các so sánh toán tử đó sẽ trả về sai hoặc null.

Cú pháp là:

```
CREATE TABLE circles (
        c circle,
        EXCLUDE USING gist (c WITH &&)
);
```

Xem thêm CREATE TABLE ... CONSTRAINT ... EXCLUDE để có thêm chi tiết.

Việc bổ sung thêm một ràng buộc loại trừ sẽ tự động tạo ra một chỉ số của dạng được chỉ định trong khai báo ràng buộc đó.

5.4. Cột hệ thống

Mỗi bảng có vài cột hệ thống được hệ thống định nghĩa theo một cách ẩn. Vì thế, các tên đó không thể được sử dụng như là tên các cột do người sử dụng định nghĩa. (Lưu ý rằng những hạn chế đó là tách bạch với việc liệu tên đó có là một từ khóa hay không; đưa vào ngoặc một cái tên sẽ không cho phép bạn thoát khỏi những giới hạn đó). Bạn thực sự không cần lo ngại về các cột đó; chỉ biết chúng có tồn tại.

oid

Mã định danh đối tượng (ID đối tượng) của một hàng. Cột này chỉ được trình bày nếu bảng đã được tạo ra bằng việc sử dụng WITH OIDS, hoặc nếu biến cấu hình default_with_oids đã được thiết lập khi đó. Cột này là oid dạng (tên y hệt như cột); xem Phần 8.16 để có thêm thông tin về dạng đó.

tableoid

OID của bảng có chứa hàng này. Cột này đặc biệt thuận tiện cho các truy vấn mà chọn từ các tôn ti trật tự cấp bậc thừa kế (xem Phần 5.8), vì không có nó, khó để nói một hàng tới từ bảng riêng rẽ nào. tableoid có thể được kết nối đối với cột oid của pg_class để có được tên bảng.

xmin

Sự định danh (ID giao dịch) của giao dịch chèn cho phiên bản hàng. (Một phiên bản hàng là tình trạng một hàng riêng rẽ; từng cập nhật của một hàng tạo ra một phiên bản hàng mới cho hàng logic y hệt).

cmin

Mã định danh lệnh (bắt đầu từ 0) trong giao dịch chèn.

xmax

Sự định danh (ID giao dịch) của giao dịch xóa, hoặc 0 cho một phiên bản hàng bị xóa. Có khả năng đối với cột này sẽ là không bằng 0 trong một phiên bản hàng nhìn thấy được. Điều đó thường chỉ ra rằng giao dịch xóa còn chưa được thực hiện, hoặc một sự xóa có ý định đã

được hoàn ngược lại.

cmax

Mã định dạng lệnh trong giao dịch xóa, hoặc 0.

ctid

Vị trí vật lý của phiên bản hàng trong bảng của nó. Lưu ý rằng dù ctid có thể được sử dụng để định vị phiên bản hàng rất nhanh, một ctid sẽ thay đổi nếu nó được cập nhật hoặc loạt bỏ bằng VACUUM FULL. Vì thế ctid là vô dụng như một mã định danh hàng trong dài hạn. OID, hoặc thậm chí tốt hơn một số thứ tự do người sử dụng định nghĩa, sẽ được sử dụng để xác định các hàng logic.

Các OID là các lượng 32 bit và được chỉ định từ một con tính theo bó rộng duy nhất. Trong một cơ sở dữ liệu lớn hoặc để sống dài lâu, có khả năng đối với con tính để bọc xung quanh. Kể từ đây, là thực tế tồi để giả thiết rằng các OID là độc nhất, trừ phi bạn nắm lấy các bước để đảm bảo rằng đây chính là vấn đề. Nếu bạn cần xác định các hàng trong một bảng, việc sử dụng một bộ phát tuần tự được khuyến cáo mạnh mẽ. Tuy nhiên, các OID cũng có thể được sử dụng, miễn là một ít đề phòng bổ sung được thực hiện:

- Ràng buộc độc nhất sẽ được tạo ra trong cột OID của từng bảng mà theo đó OID sẽ được sử dụng để xác định các hàng. Khi một ràng buộc độc nhất như vậy (hoặc chỉ số độc nhất) tồn tại, thì hệ thống chăm sóc không phải tạo ra một OID khớp với một hàng đang tồn tại sẵn rồi. (Tất nhiên, điều này chỉ có thể nếu bảng đó chứa ít hơn 2^{32} (4 tỷ) hàng, và trong thực tế kích cỡ bảng tốt hơn nếu nhỏ hơn thế nhiều, hoặc hiệu năng có thể bị ảnh hưởng).
- Các OID sẽ không bao giờ được giả thiết là độc nhất xuyên khắp các bảng; hãy sử dụng sự kết hợp của tableoid và OID hàng nếu bạn cần một mã định danh cơ sở dữ liệu rộng lớn.
- Tất nhiên, các bảng theo yêu cầu phải được tạo ra bằng WITH OIDS. Từ PostgreSQL 8.1, WITHOUT OIDS là mặc định.

Các mã định danh của giao dịch cũng là các lượng 32 bit. Trong một cơ sở dữ liệu sống dài lâu thì có khả năng cho các ID giao dịch phải bọc xung quanh. Đây không phải là một vấn đề nghiêm trọng, biết rằng có các thủ tục duy trì phù hợp; xem Chương 23 để có thêm chi tiết. Tuy nhiên, điều đó là không khôn ngoan, để phụ thuộc vào sự độc nhất của các ID giao dịch về lâu dài (hơn 1 tỷ giao dịch).

Các mã định danh lệnh cũng là các lượng 32 bit. Điều này tạo ra một giới hạn cứng 2^{32} (4 tỷ) lệnh SQL trong một giao dịch duy nhất. Trong thực tế giới hạn này không là vấn đề - lưu ý rằng giới hạn này là trong số các lệnh SQL, không phải trong số các hàng được xử lý. Hơn nữa, như từ PostgreSQL 8.3, chỉ các lệnh mà thực sự sửa đổi các nội dung của cơ sở dữ liệu sẽ sử dụng một mã định danh lệnh.

5.5. Sửa đổi bảng

Khi bạn tạo một bảng và bạn nhận thấy rằng bạn đã tạo một sai lầm, hoặc các yêu cầu của ứng dụng thay đổi, bạn có thể bỏ bảng đó và tạo nó lại. Nhưng điều này không phải là một lựa chọn thuận tiện nếu bảng đó được điền các dữ liệu rồi, hoặc nếu bảng đó được các đối tượng cơ sở dữ liệu khác tham chiếu tới (ví dụ một ràng buộc khóa ngoại). Vì thế PostgreSQL cung cấp một họ các lệnh để tiến hành các sửa đổi đối với các bảng đang tồn tại. Lưu ý là điều này, về mặt khái niệm, là khác với việc tùy biến các dữ liệu nằm trong bảng đó: ở đây chúng ta có quan tâm trong việc tùy biến định nghĩa, hoặc cấu trúc của bảng.

Bạn có thể:

- Thêm các cột
- Loại bỏ các cột
- Thêm các ràng buộc
- Loại bỏ các ràng buộc
- Thay đổi các giá trị mặc định
- Thay đổi các dạng dữ liệu cột
- Đổi tên các cột
- Đổi tên các bảng

Tất cả các hành động đó được thực hiện bằng việc sử dụng lệnh ALTER TABLE, trang tham chiếu của nó có các chi tiết vượt ra khỏi những gì được nêu ở đây.

5.5.1. Thêm cột

Để thêm một cột, hãy sử dụng một lệnh giống như:

```
ALTER TABLE products ADD COLUMN description text;
```
(Sửa bảng sản phẩm và thêm cột văn bản mô tả)

Cột mới ban đầu được điền với giá trị mặc định bất kể thế nào được đưa ra (null nếu bạn không chỉ định một mệnh đề mặc định DEFAULT).

Bạn cũng có thể định nghĩa các ràng buộc trong cột cùng một lúc, bằng việc sử dụng cú pháp thông thường:

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

Trong thực tế tất cả các lựa chọn có thể được áp dụng cho một mô tả cột trong CREATE TABLE có thể được sử dụng ở đây. Tuy nhiên hãy nhớ trong đầu rằng giá trị mặc định phải thỏa mãn các ràng buộc được đưa ra, hoặc lệnh ADD sẽ hỏng. Như một lựa chọn, bạn có thể thêm các ràng buộc sau (xem bên dưới) sau khi bạn đã điền vào cột mới đó một cách đúng đắn.

> **Mẹo**: Việc thêm một cột với mặc định đòi hỏi việc cập nhật từng hàng của bảng (để lưu giữ

giá trị cột mới). Tuy nhiên, nếu không mặc định nào được chỉ định, thì PostgreSQL có khả năng tánh sự cập nhật vật lý. Vì thế nếu bạn định điền đầy cột với các giá trị hầu hết không mặc định, thì tốt nhất hãy thêm cột với không mặc định, hãy chèn các giá trị đúng bằng việc sử dụng UPDATE, và sau đó thêm bất kỳ mặc định mong muốn nào như được mô tả ở dưới.

5.5.2. Loại bỏ cột

Để loại bỏ một cột, hãy sử dụng một lệnh giống như là:

```
ALTER TABLE products DROP COLUMN description;
```

Bất kể dữ liệu nào từng có trong cột sẽ biến mất. Các ràng buộc bảng có liên quan tới cột cũng sẽ bị xóa nốt. Tuy nhiên, nếu cột đó được một ràng buộc khóa ngoài của một bảng khác tham chiếu tới, thì PostgreSQL sẽ không loại bỏ một cách âm thầm ràng buộc đó. Bạn có thể cho phép việc loại bỏ bất kỳ thứ gì mà phụ thuộc vào cột đó bằng việc thêm vào CASCADE:

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

Xem Phần 5.11 để có sự mô tả cơ chế chung đằng sau điều này.

5.5.3. Thêm ràng buộc

Để thêm một ràng buộc, cú pháp ràng buộc bảng được sử dụng. Ví dụ:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

Để thêm một ràng buộc không null, mà nó không thể được viết như một ràng buộc bảng, hãy sử dụng cú pháp này:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

Ràng buộc đó sẽ được kiểm tra ngay lập tức, nên dữ liệu của bảng phải làm thỏa mãn ràng buộc đó trước khi nó có thể được thêm vào.

5.5.4. Loại bỏ ràng buộc

Để loại bỏ một ràng buộc thì bạn cần biết tên của nó. Nếu bạn đã cho nó một cái tên thì điều đó là dễ dàng. Nếu không thì hệ thống đã chỉ định một tên được sinh ra, mà bạn cần tìm ra nó. Lệnh psql \d tablename có thể là hữu ích ở đây; các giao diện khác có thể cũng cung cấp một cách thức để kiểm tra các chi tiết bảng. Lệnh đó là:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

(Nếu bạn đang làm việc với một tên ràng buộc được sinh ra như là $2, đừng quên rằng bạn sẽ cần đưa nó vào các dấu ngoặc kép để làm cho nó thành một mã định danh hợp lệ).

Như với việc loại bỏ một cột, bạn cần phải thêm CASCADE nếu bạn muốn loại bỏ một ràng buộc mà thứ gì đó khác phụ thuộc vào nó. Một ví dụ là việc một ràng buộc khóa ngoại phụ thuộc vào một ràng buộc khóa chủ hoặc độc nhất trong (các) cột được tham chiếu.

Điều này làm việc y hệt đối với tất cả các dạng ràng buộc ngoại trừ các ràng buộc không null. Để loại bỏ một ràng buộc không null, hãy sử dụng:

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

(Nhớ lại rằng các ràng buộc không null không có các tên).

5.5.5. Thay đổi giá trị mặc định của cột

Để thiết lập một mặc định mới cho một cột, hãy sử dụng một lệnh như:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Lưu ý rằng điều này không tác động tới bất kỳ hàng đang tồn tại nào trong bảng, nó chỉ làm thay đổi mặc định đối với các lệnh chèn INSERT trong tương lai.

Để loại bỏ bất kỳ giá trị mặc định nào, hãy sử dụng:

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

Điều này có hiệu quả y hệt như việc thiết lập mặc định về null. Như là một hệ quả, đây không phải là một lỗi để loại bỏ một mặc định trong đó một giá trị mặc định từng không được định nghĩa, vì mặc định đó là giá trị null ẩn.

5.5.6. Thay đổi dạng dữ liệu của một cột

Để chuyển đổi một cột tới một dạng dữ liệu khác, hãy sử dụng lệnh sau:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

điều này sẽ chỉ thành công nếu từng khoản đầu vào đang tồn tại trong cột có thể được chuyển đổi sang dạng mới bằng một cast ẩn. Nếu một sự biến đổi phức tạp hơn là cần thiết, thì bạn có thể thêm một mệnh đề USING mà chỉ định cách tính toán các giá trị mới từ giá trị cũ.

PostgreSQL sẽ cố gắng biến đổi giá trị mặc định của cột (nếu có) sang dạng mới đó, cũng như bất kỳ ràng buộc nào có liên quan tới cột đó. Nhưng những biến đổi đó có thể hỏng, hoặc có thể tạo ra các kết quả gây ngạc nhiên. Thường tốt nhất hãy loại bỏ bất kỳ ràng buộc nào trong cột đó trước khi tùy biến dạng của nó, và sau đó thêm trở lại các ràng buộc được sửa đổi phù hợp sau.

5.5.7. Đổi tên cột

Để đổi tên một cột:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.5.8. Đổi tên bảng

Để đổi tên bảng:

```
ALTER TABLE products RENAME TO items;
```

5.6. Các quyền ưu tiên

Khi bạn tạo một đối tượng cơ sở dữ liệu, bạn trở thành chủ của nó. Mặc định, chỉ người chủ của một đối tượng mới có thể làm bất kỳ điều gì với đối tượng đó. Để cho phép những người sử dụng khác sử dụng nó, các *quyền ưu tiên* phải được trao. (Tuy nhiên, những người sử dụng mà có thuộc tính siêu người sử dụng - superuser luôn có thể truy cập bất kỳ đối tượng nào).

Có vài quyền ưu tiên khác nhau: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, và USAGE. Các quyền ưu tiên được áp dụng cho một đối tượng đặc biệt sẽ khác nhau, phụ thuộc vào dạng đối tượng (bảng, hàm, …). Để có thông tin hoàn chỉnh về các dạng quyền ưu tiên khác nhau được PostgreSQL hỗ trợ, hãy tham khảo trang tham chiếu trao quyền GRANT. Các phần và các chương sau đây cũng sẽ chỉ cho bạn cách mà các quyền ưu tiên đó được sử dụng.

Quyền sửa đổi hoặc phá hủy một đối tượng luôn là quyền của chỉ người chủ.

> **Lưu ý**: Để thay đổi người chủ của một bảng, chỉ số, sự tuần tự hoặc kiểu nhìn, hãy sử dụng lệnh tùy biến bảng ALTER TABLE. Có các lệnh ALTER tương ứng cho các dạng đối tượng khác.

Để chỉ định các quyền ưu tiên, lệnh GRANT được sử dụng. Ví dụ, nếu joe là một người sử dụng đang tồn tại, và accounts (các tài khoản) là một bảng đang tồn tại, thì quyền ưu tiên để cập nhật bảng đó có thể được trao bằng:

```
GRANT UPDATE ON accounts TO joe;
```

Việc viết ALL vào chỗ của một quyền ưu tiên đặc biệt trao tất cả các quyền ưu tiên là phù hợp cho dạng đối tượng đó.

Tên của "người sử dụng" đặc biệt PUBLIC có thể được sử dụng để trao một quyền ưu tiên cho từng người sử dụng trong hệ thống. Hơn nữa, các vai trò "nhóm" có thể được thiết lập để giúp quản lý các quyền ưu tiên khi có nhiều người sử dụng của một cơ sở dữ liệu - xem các chi tiết ở Chương 20.

Để thu hồi một quyền ưu tiên, hãy sử dụng lệnh thu hồi REVOKE có tên thích hợp:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Các quyền ưu tiên đặc biệt của người chủ đối tượng (như, quyền thực hiện DROP, GRANT, REVOKE, …) luôn là ẩn khi là người chủ, và không thể được trao hoặc bị thu hồi. Nhưng người chủ đối tượng có thể chọn để thu hồi các quyền ưu tiên thông thường của anh ta, ví dụ để làm cho một bảng chỉ đọc đối với bản thân anh ta cũng như những người khác.

Thông thường, chỉ người chủ đối tượng (hoặc một siêu người sử dụng – superuser) có thể trao hoặc thu hồi các quyền ưu tiên đối với một đối tượng. Tuy nhiên, có khả năng để trao một quyền ưu tiên "với lựa chọn trao", nó trao cho người nhận quyền để tới lượt trao nó cho những người khác. Nếu lựa chọn trao bị thu hồi sau đó thì tất cả những ai đã nhận được quyền ưu tiên từ người nhận đó (một cách trực tiếp hoặc thông qua một chuỗi các cuộc trao) sẽ mất quyền ưu tiên đó. Các chi tiết thêm có trong các trang tham chiếu của GRANT và REVOKE.

5.7. Sơ đồ (schema)

Một bó cơ sở dữ liệu PostgreSQL bao gồm một hoặc nhiều cơ sở dữ liệu có tên. Những người sử dụng và các nhóm người sử dụng được chia sẻ qua toàn bộ bó, nhưng không dữ liệu nào khác được chia sẻ qua các cơ sở dữ liệu. Bất kỳ kết nối máy trạm tới máy chủ nào được đưa ra cũng chỉ có thể truy cập được các dữ liệu trong một cơ sở dữ liệu duy nhất, cơ sở dữ liệu được chỉ định theo yêu cầu kết nối.

> **Lưu ý**: Người sử dụng của một bó máy không nhất thiết phải có quyền ưu tiên để truy cập bất kỳ cơ sở dữ liệu nào trong bó máy đó. Việc chia sẻ các tên người sử dụng có nghĩa là không thể có những người sử dụng khác được đặt tên, ví dụ, joe trong 2 cơ sở dữ liệu trong cùng một bó máy; nhưng hệ thống có thể được thiết lập cấu hình để cho phép joe truy cập tới chỉ một số cơ sở dữ liệu đó.

Một cơ sở dữ liệu bao gồm một hoặc nhiều sơ đồ, tới lượt chúng bao gồm các bảng. Các sơ đồ cũng bao gồm các dạng khác các đối tượng có tên, bao gồm các dạng dữ liệu, các hàm và các toán tử. Tên đối tượng y hệt có thể được sử dụng trong các sơ đồ khác nhau mà không có xung đột; ví dụ, cả schema1 và myschema đều có thể có các bảng có tên là mytable. Không giống như các cơ sở dữ liệu, các sơ đồ không được tách bạch nhau một cách cứng nhắc: một người sử dụng có thể truy cập các đối tượng trong bất kỳ sơ đồ nào trong cơ sở dữ liệu mà anh ta được kết nối tới, nếu anh ta có các quyền ưu tiên để làm thế.

Có vài lý do vì sao một người có thể muốn sử dụng các sơ đồ:

- Để cho phép nhiều người sử dụng dùng một cơ sở dữ liệu mà không quấy rầy lẫn nhau.
- Để tổ chức các đối tượng cơ sở dữ liệu trong các nhóm logic làm cho chúng có khả năng quản lý được nhiều hơn.
- Các ứng dụng của bên thứ 3 có thể được đặt vào các sơ đồ tách bạch nhau sao cho chúng không xung đột với các tên của các đối tượng khác.

Các sơ đồ là tương tự với các thư mục ở mức hệ điều hành, ngoại trừ là các sơ đồ không thể lồng nhau được.

5.7.1. Tạo sơ đồ

Để tạo một sơ đồ, hãy sử dụng lệnh tạo sơ đồ CREATE SCHEMA. Hãy đặt tên cho sơ đồ theo ý bạn. Ví dụ:

```
CREATE SCHEMA myschema;
```

Để tạo hoặc truy cập các đối tượng trong một sơ đồ, hãy viết một cái tên đủ điều kiện được cấu tạo từ tên sơ đồ và tên bảng được cách nhau bằng một dấu chấm:

```
schema.table
```

Điều này làm việc ở bất kỳ đâu mà tên một bảng được mong đợi, bao gồm cả các lệnh sửa đổi bảng

và các lệnh truy cập dữ liệu được thảo luận trong các chương sau. (Vì sự khúc triết chúng tôi sẽ chỉ nói về các bảng, nhưng các ý tưởng y hệt áp dụng cho các dạng khác các đối tượng có tên, như các dạng và các hàm).

Thực sự, thậm chí cú pháp thông thường hơn

```
database.schema.table
```

có thể cũng được sử dụng, nhưng hiện tại điều này chỉ cho sự tuân thủ chính thống với tiêu chuẩn SQL. Nếu bạn viết một tên cơ sở dữ liệu, nó phải là tên như cơ sở dữ liệu bạn được kết nối tới.

Vì thế để tạo một bảng trong một sơ đồ mới, hãy sử dụng:

```
CREATE TABLE myschema.mytable (
        ...
);
```

Để bỏ một sơ đồ nếu nó rỗng (tất cả các đối tượng trong nó đã bị bỏ), hãy sử dụng:

```
DROP SCHEMA myschema CASCADE;
```

Để bỏ một sơ đồ bao gồm tất cả các đối tượng bên trong, hãy sử dụng:

```
DROP SCHEMA myschema CASCADE;
```

Xem Phần 5.11 để có mô tả cơ chế chung đằng sau điều này.

Thường thì bạn sẽ muốn tạo một sơ đồ mà ai đó khác là chủ (vì điều này là một trong các cách thức để hạn chế các hoạt động của những người sử dụng của bạn đối với các không gian tên được định nghĩa tốt). Cú pháp cho điều đó là:

```
CREATE SCHEMA schemaname AUTHORIZATION username;
```

Bạn có thể thậm chí bỏ qua tên sơ đồ, trong trường hợp đó thì tên sơ đồ sẽ là y hệt như tên của người sử dụng. Xem Phần 5.7.6 để biết cách mà điều này có thể là hữu dụng.

Các tên sơ đồ bắt đầu với `pg_` được dành cho các mục đích của hệ thống và không thể được người sử dụng tạo ra.

5.7.2. Sơ đồ công khai

Trong các phần trước chúng ta đã tạo ra các bảng mà không có việc chỉ định bất kỳ tên sơ đồ nào. Mặc định thì các bảng như vậy (và các đối tượng khác) tự động được đặt vào trong sơ đồ có tên là "public" ("công cộng"). Từng cơ sở dữ liệu mới đều có một sơ đồ. Vì thế, thứ sau là tương đương:

```
CREATE TABLE products ( ... );
```

và

```
CREATE TABLE public.products ( ... );
```

5.7.3. Đường tìm kiếm sơ đồ

Các cái tên có đủ điều kiện là nặng nhọc để viết, và thường tốt nhất là không viết một tên sơ đồ cụ thể nào vào các ứng dụng cả. Vì thế các bảng thường được các tên không đủ điều kiện tham chiếu

tới, nó bao gồm chỉ tên bảng. Hệ thống xác định bảng nào là có nghĩa bằng cách đi theo một đường tìm kiếm, nó là một danh sách các sơ đồ để tra cứu. Bảng khớp đầu tiên trong đường tìm kiếm được lấy là một bảng mong muốn. Nếu không có sự trùng khớp nào trong đường tìm kiếm, thì một lỗi được báo, thậm chí nếu các tên bảng trùng khớp tồn tại trong các sơ đồ khác trong cơ sở dữ liệu đó.

Sơ đồ đầu tiên có tên trong đường tìm kiếm được gọi là sơ đồ hiện hành. Bên cạnh là sơ đồ đầu tiên được tìm thấy, nó cũng là sơ đồ mà trong đó các bảng mới sẽ được tạo ra nếu lệnh tạo bảng CREATE TABLE không chỉ định một tên sơ đồ.

Để chỉ ra đường tìm kiếm hiện hành, hãy sử dụng lệnh sau đây:

```
SHOW search_path;
```

Trong thiết lập mặc định thì điều này trả về:

```
search_path
--------------
"$user",public
```

Yếu tố đầu tiên chỉ ra rằng một sơ đồ với tên cùng y hệt như của người sử dụng hiện hành sẽ được tìm. Nếu không có sơ đồ nào như vậy tồn tại, thì khoản đầu vào đó bị bỏ qua. Yếu tố thứ 2 tham chiếu tới sơ đồ công khai mà chúng ta đã thấy rồi.

Sơ đồ đầu trong đường tìm kiếm mà tồn tại là vị trí mặc định cho việc tạo các đối tượng mới. Đó là lý do giải thích vì sao theo mặc định thì các đối tượng sẽ được tạo ra trong sơ đồ công khai. Khi các đối tượng được tham chiếu theo bất kỳ ngữ cảnh nào mà không có sự định phẩm chất sơ đồ (sửa đổi bảng, sửa đổi dữ liệu, hoặc các lệnh truy vấn) thì đường tìm kiếm được đi ngang cho tới khi một đối tượng trùng khớp được tìm thấy. Vì thế, theo cấu hình mặc định, bất kỳ sự truy cập không đủ tiêu chuẩn nào cũng chỉ có thể tham chiếu tới sơ đồ công khai mà thôi.

Để đặt sơ đồ mới của chúng ta vào đường đó, chúng ta sử dụng:

```
SET search_path TO myschema,public;
```

(Chúng ta bỏ qua $user ở đây vì chúng ta chưa cần ngay đối với nó). Và sau đó chúng ta có thể truy cập bảng đó mà không có sự định phẩm chất sơ đồ:

```
DROP TABLE mytable;
```

Hơn nữa, vì myschema là yếu tố đầu tiên trong đường đó, nên các đối tượng mới có thể theo mặc định được tạo ra trong nó.

Chúng ta cũng có thể đã viết:

```
SET search_path TO myschema;
```

Sau đó chúng ta không còn có được sự truy cập tới sơ đồ công khai mà không có sự định phẩm chất rõ ràng nữa. Không có gì đặc biệt về sơ đồ công khai ngoại trừ là nó tồn tại theo mặc định. Nó cũng có thể bị bỏ đi.

Xem Phần 9.23 để có cách cách thức khác điều khiển đường tìm kiếm sơ đồ.

Đường tìm kiếm làm việc theo y hệt cách đối với các tên dạng dữ liệu, các tên hàm và các tên toán

tử như nó làm đối với các tên bảng. Nếu bạn cần viết một tên toán tử đủ điều kiện trong một biểu thức, thì có một cách đặc biệt: bạn phải viết

```
OPERATOR(schema.operator)
```

Điều này là cần thiết để tránh sự mù mờ về cú pháp. Một ví dụ:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

Trong thực tế người ta thường dựa vào đường tìm kiếm đối với các toán tử, vì thế không phải viết bất kỳ điều gì quá xấu xí như thế.

5.7.4. Sơ đồ và quyền ưu tiên

Mặc định, người sử dụng không thể truy cập bất kỳ đối tượng nào trong các sơ đồ mà họ không là chủ. Để cho phép điều đó, người chủ sơ đồ phải trao quyền ưu tiên sử dụng USAGE trong sơ đồ đó. Để cho phép những người sử dụng dùng các đối tượng trong sơ đồ, các quyền ưu tiên bổ sung có thể cần phải được trao, phù hợp với dự án.

Một người sử dụng cũng có thể được phép tạo ra các đối tượng trong sơ đồ của ai đó khác. Để cho phép điều đó, quyền ưu tiên CREATE trong sơ đồ cần phải được trao. Lưu ý là theo mặc định, từng người có các quyền ưu tiên CREATE và USAGE trong sơ đồ công khai public. Điều này cho phép tất cả những người sử dụng mà có khả năng kết nối tới một cơ sở dữ liệu được đưa ra để tạo các đối tượng trong sơ đồ public của mình. Nếu bạn không muốn cho phép điều đó, thì bạn có thể thu hồi quyền ưu tiên đó:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

(“Public” (“Công khai”) đầu tiên trong sơ đồ, bản ghi “public” có nghĩa là “từng người sử dụng”. Theo nghĩa đầu tiên thì nó là một mã định danh, theo nghĩa thứ 2 thì nó là một từ khóa, vì thế viết chữ hoa hay chữ thường là khác nhau; hãy nhớ lại các chỉ dẫn từ Phần 4.1.1).

5.7.5. Sơ đồ catalog hệ thống

Bổ sung thêm vào các sơ đồ public và do người sử dụng tạo ra, từng cơ sở dữ liệu bao gồm một sơ đồ pg_catalog, nó bao gồm các bảng hệ thống và tất cả các dạng dữ liệu và các toán tử được xây dựng sẵn. pg_catalog luôn là phần có hiệu quả của đường tìm kiếm. Nếu nó không được đặt tên một cách rõ ràng trong đường đó thì nó sẽ được tìm kiếm ẩn trước khi tìm kiếm các sơ đồ của đường đó. Điều này đảm bảo rằng các tên được xây dựng sẵn sẽ luôn có khả năng tìm thấy được. Tuy nhiên, bạn có thể rõ ràng đặt pg_catalog ở cuối đường tìm kiếm của bạn nếu bạn thích các tên do người sử dụng định nghĩa ghi đè lên các tên được xây dựng sẵn.

Trong PostgreSQL phiên bản trước 7.3, các tên bảng bắt đầu bằng pg_ were được giữ lại. Điều này không còn đúng nữa: bạn có thể tạo một tên bảng như vậy nếu bạn muốn, trong bất kỳ thứ gì không phải hệ thống. Tuy nhiên, tốt nhất hãy tiếp tục tránh các tên như vậy, để đảm bảo rằng bạn sẽ không chịu một xung đột nếu có phiên bản trong tương lai định nghĩa một bảng hệ thống có tên y hệt như bảng của bạn. (Với đường tìm kiếm mặc định, một tham chiếu không đủ điều kiện tới tên bảng của

bạn có thể sau đó được giải quyết như là bảng hệ thống). Các bảng hệ thống sẽ tiếp tục tuân theo qui ước có các tên bắt đầu với pg_, sao cho chúng sẽ không xung đột với các tên bảng của người sử dụng không đủ điều kiện, miễn là những người sử dụng tránh tiền tố pg_.

5.7.6. Sử dụng các mẫu

Các sơ đồ có thể được sử dụng để tổ chức các dữ liệu của bạn theo nhiều cách. Có một ít các mẫu sử dụng được khuyến cáo và dễ dàng được cấu hình mặc định hỗ trợ.

- Nếu bạn không tạo ra bất kỳ sơ đồ nào thì sau đó tất cả những người sử dụng truy cập ẩn sơ đồ công khai. Điều này khuyến khích tình huống nơi mà các sơ đồ hoàn toàn không sẵn sàng. Thiết lập này chủ yếu được khuyến cáo khi chỉ có một người sử dụng duy nhất hoặc một ít người sử dụng cộng tác trong một cơ sở dữ liệu. Thiết lập này cũng cho phép biến đổi trơn tru từ thế giới thừa nhận không sơ đồ.
- Bạn có thể tạo một sơ đồ cho từng người sử dụng với cùng tên như người sử dụng đó. Hãy nhớ lại rằng đường tìm kiếm mặc định với $user, nó giải quyết cho tên người sử dụng đó. Vì thế nếu từng người sử dụng có một sơ đồ riêng rẽ, thì họ truy cập các sơ đồ riêng của họ theo mặc định.

  Nếu bạn sử dụng thiết lập này thì bạn cũng có thể muốn thu hồi sự truy cập tới sơ đồ công khai (hoặc bỏ nó cùng), sao cho những người sử dụng thực sự có ràng buộc với các sơ đồ của riêng họ.
- Để cài đặt các ứng dụng được chia sẻ (các bảng được từng người sử dụng, các hàm bổ sung được các bên thứ 3 cung cấp, …), đặt chúng vào các sơ đồ riêng rẽ. Hãy nhớ phải trao các quyền ưu tiên phù hợp để cho phép những người sử dụng khác truy cập chúng. Những người sử dụng có thể sau đó tham chiếu tới các đối tượng bổ sung thêm đó bằng việc kiểm tra phẩm chất các tên với một tên sơ đồ, hoặc họ có thể đặt các sơ đồ bổ sung thêm đó vào đường tìm kiếm của họ, khi họ chọn.

5.7.7. Tính khả chuyển

Theo tiêu chuẩn SQL, ký hiệu các đối tượng trong cùng sơ đồ được làm chủ bởi những người sử dụng khác nhau không tồn tại. Hơn nữa, một số triển khai không cho phép bạn tạo các sơ đồ mà có một tên khác với tên của người chủ của họ. Trong thực tế, các khái niệm về sơ đồ và người sử dụng gần như là tương đương trong một hệ thống cơ sở dữ liệu mà triển khai hỗ trợ chỉ sơ đồ cơ bản được chỉ định theo tiêu chuẩn. Vì thế, nhiều người sử dụng coi các tên có đủ điều kiện gần như bao gồm username.tablename. Đây là cách mà PostgreSQL sẽ hành xử có hiệu lực nếu bạn tạo một sơ đồ cho từng người sử dụng.

Hơn nữa, không có khái niệm một sơ đồ public theo tiêu chuẩn SQL. Tuân thủ tối đa đối với tiêu chuẩn đó, bạn không nên sử dụng (có lẽ thậm chí loại bỏ) sơ đồ public đó.

Tất nhiên, một số hệ thống cơ sở dữ liệu SQL có thể không triển khai các sơ đồ đó hoàn toàn, hoặc

cung cấp sự hỗ trợ không gian tên bằng việc cho phép (có thể có giới hạn) sự truy cập liên các cơ sở dữ liệu. Nếu bạn cần làm việc với các hệ thống đó, thì tính khả chuyển cực đại có thể đạt được bằng việc hoàn toàn không sử dụng các sơ đồ.

5.8. Kế thừa

PostgreSQL triển khai sự kế thừa bảng, nó có thể là công cụ hữu dụng cho các nhà thiết kế bảng. (SQL:1999 và sau này định nghĩa một tính năng kế thừa dạng, nó khác theo nhiều khía cạnh với các tính năng được mô tả ở đây).

Hãy bắt đầu với một ví dụ: giả sử chúng ta đang cố gắng xây dựng một mô hình dữ liệu cho các thành phố. Từng bang có nhiều thành phố, nhưng chỉ có một thủ phủ. Chúng ta muốn có khả năng nhanh chóng truy xuất thành phố thủ phủ cho bất kỳ bang đặc biệt nào. Điều này có thể được thực hiện bằng việc tạo ra 2 bảng, một cho các thủ phủ bang và một cho các thành phố mà không phải là các thủ phủ. Tuy nhiên, điều gì xảy ra khi chúng ta muốn yêu cầu các dữ liệu về một thành phố, bất kể liệu nó có là thủ phủ hay không? Tính năng kế thừa có thể giúp giải quyết vấn đề này. Chúng ta định nghĩa các bảng thủ phủ - capitals sao cho nó kế thừa từ bảng các thành phố - cities:

```
CREATE TABLE cities (
        name text,
        population float,
        altitude int -- in feet
);

CREATE TABLE capitals (
        state char(2)
) INHERITS (cities);
```

Trong trường hợp này, bảng capitals kế thừa tất cả các cột của bảng cha của nó, bảng cities. Các thủ phủ bang cũng có một cột thêm ra, cột bang - state, mà chỉ ra bang của chúng.

Trong PostgreSQL, một bảng có thể kế thừa từ 0 hoặc nhiều bảng khác, và một truy vấn có thể tham chiếu hoặc tất cả các hàng của một bảng, hoặc tất cả các hàng của một bảng cộng với tất cả các bảng con của nó. Hành vi sau là mặc định. Ví dụ, truy vấn sau đây tìm các tên của tất cả các thành phố, bao gồm cả các thủ phủ bang, mà nằm ở một độ cao hơn 500 feet:

```
SELECT name, altitude
        FROM cities
        WHERE altitude > 500;
```

Đưa ra các dữ liệu mẫu từ sách chỉ dẫn PostgreSQL (xem Phần 2.1), điều này trả về:

```
name            | altitude
----------------+----------
Las Vegas       | 2174
Mariposa        | 1953
Madison         |  845
```

Mặt khác, truy vấn sau đây tìm tất cả các thành phố mà không phải là các thủ phủ bang và nằm ở một độ cao hơn 500 feet:

```
SELECT name, altitude
        FROM ONLY cities
        WHERE altitude > 500;

name            | altitude
----------------------+----------
Las Vegas       | 2174
Mariposa        | 1953
```

Ở đây từ khóa ONLY chỉ rằng truy vấn chỉ nên áp dụng cho bảng cities, và không cho bất kỳ bảng nào bên dưới cities trong tôn ti trật tự kế thừa đó. Nhiều lệnh mà chúng ta đã thảo luận rồi - SELECT, UPDATE và DELETE - hỗ trợ cho từ khóa ONLY.

Bạn cũng có thể viết tên bảng với một cái đuôi * để chỉ định rõ ràng rằng các bảng con có bao gồm:

```
SELECT name, altitude
        FROM cities*
        WHERE altitude > 500;
```

Việc viết * là không cần thiết, vì hành vi này là mặc định (trừ phi bạn đã thay đổi thiết lập của tùy chọn cấu hình sql_inheritance). Tuy nhiên việc viết * có thể là hữu dụng để nhấn mạnh rằng các bảng bổ sung sẽ được tìm kiếm.

Trong một số trường hợp bạn có thể muốn biết bảng nào một hàng đặc biệt là từ đó ra. Có một cột hệ thống gọi là tableoid trong từng bảng mà có thể nói cho bạn bảng gốc gác:

```
SELECT c.tableoid, c.name, c.altitude
        FROM cities c
        WHERE c.altitude > 500;
```

nó trả về:

```
tableoid | name          | altitude
-----------+------------------+----------
 139793 | Las Vegas     | 2174
 139793 | Mariposa      | 1953
 139798 | Madison       |  845
```

(Nếu bạn cố gắng tái tạo ví dụ này, thì bạn sẽ có khả năng có các số OID khác nhau). Bằng việc thực hiện liên kết với pg_class bạn có thể thấy các tên bảng thực sự:

```
SELECT p.relname, c.name, c.altitude
        FROM cities c, pg_class p
        WHERE c.altitude > 500 AND c.tableoid = p.oid;
```

nó trả về:

```
relname | name          | altitude
------------+------------------+------------
cities     | Las Vegas     | 2174
cities     | Mariposa      | 1953
capitals   | Madison       |  845
```

Sự kế thừa không tự động truyền giống các dữ liệu từ các lệnh INSERT hoặc COPY tới các bảng khác trong tôn ti trật tự kế thừa. Trong ví dụ của chúng ta, lệnh INSERT sau đây sẽ hỏng:

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('New York', NULL, NULL, 'NY');
```

Chúng ta có thể hy vọng rằng các dữ liệu có thể bằng cách nào đó được định tuyến tới bảng capitals, nhưng điều này không xảy ra: INSERT luôn chèn vào chính xác bảng được chỉ định. Trong một số trường hợp có khả năng để tái định tuyến sự chèn bằng việc sử dụng một qui tắc (xem Chương 37).

Tuy nhiên điều đó không giúp được đối với trường hợp ở trên vì bảng cities không chứa cột state, và vì thế lệnh đó sẽ bị từ chối trước khi qui tắc đó có thể được áp dụng.

Tất cả các ràng buộc kiểm tra và các ràng buộc không null trong một bảng cha tự động được các con của nó kế thừa. Các dạng ràng buộc khác (các ràng buộc độc nhất, khóa chủ và khóa ngoại) không được kế thừa.

Một bảng có thể kế thừa từ nhiều hơn 1 bảng cha, trong trường hợp đó nó có sự hợp nhất các cột được các bảng cha định nghĩa. Bất kỳ cột nào được khai báo trong định nghĩa của bảng con cũng sẽ được bổ sung vào đó. Nếu tên cột y hệt xuất hiện trong nhiều bảng cha, hoặc trong cả một bảng cha và định nghĩa của bảng con, thì các cột đó được "trộn" sao cho chỉ có một cột như vậy trong bảng con. Để được trộn, các cột phải có cùng các dạng dữ liệu, nếu không thì một lỗi sẽ phát sinh. Cột được trộn sẽ có các bản sao của tất cả các ràng buộc kiểm tra tới từ bất kỳ một trong số các định nghĩa cột nào mà nó từ đó tới, và sẽ được đánh dấu là không null.

Sự kế thừa bảng điển hình được thiết lập khi bảng con được tạo ra, bằng việc sử dụng mệnh đề kế thừa INHERITS của lệnh tạo bảng CREATE TABLE. Như một sự lựa chọn, một bảng mà được xác định rồi theo một cách tương thích có thể có một mối quan hệ cha mới được bổ sung vào, bằng việc sử dụng phương án INHERIT của ALTER TABLE. Để làm điều này thì bảng con mới phải đưa vào rồi các cột với cùng tên và dạng như các cột của bảng cha. Nó cũng phải đưa vào các ràng buộc kiểm tra với các tên y hệt và các biểu thức kiểm tra như các biểu thức của bảng cha. Tương tự một liên kết kế thừa có thể bị loại bỏ khỏi một bảng con bằng việc sử dụng phương án NO INHERIT của ALTER TABLE. Việc bổ sung và loại bỏ động các liên kết kế thừa giống thế này có thể là hữu dụng khi mối quan hệ kế thừa đang được sử dụng cho việc phân vùng bảng (xem Phần 5.9).

Một cách thuận tiện để tạo một bảng tương thích mà sau đó sẽ được làm thành một bảng con mới là sử dụng mệnh đề LIKE trong CREATE TABLE. Điều này tạo ra một bảng mới với các cột y hệt như bảng nguồn. Nếu có bất kỳ ràng buộc CHECK nào được định nghĩa trong bảng nguồn, thì tùy chọn INCLUDING CONSTRAINTS đối với LIKE sẽ được chỉ định, khi bảng con mới phải có các ràng buộc khớp với bảng cha được cho là tương thích.

Một bảng cha không thể bị bỏ trong khi bất kỳ bảng con nào của nó vẫn còn. Các cột hoặc các ràng buộc kiểm tra của các bảng con đều không thể bị bỏ hoặc được tùy biến nếu chúng được kế thừa từ bất kỳ bảng cha nào. Nếu bạn muốn loại bỏ một bảng và tất cả các con cháu của nó, một cách dễ dàng để bỏ bảng cha với tùy chọn CASCADE.

ALTER TABLE sẽ nhân giống bất kỳ sự thay đổi nào trong các định nghĩa dữ liệu cột và các ràng buộc kiểm tra xuống tôn ti trật tự kế thừa. Một lần nữa, việc bỏ các cột mà bị phụ thuộc vào các bảng khác chỉ có khả năng khi sử dụng tùy chọn CASCADE. ALTER TABLE tuân theo cùng các qui tắc đối với việc trộn và từ chối các cột đúp bản mà áp dụng trong quá trình CREATE TABLE.

Lưu ý cách mà các quyền truy cập bảng được điều khiển. Việc truy vấn một bảng cha có thể tự động truy cập các dữ liệu trong các bảng con mà không có việc kiểm tra quyền ưu tiên truy cấp tiếp. Điều này giữ lại sự xuất hiện mà dữ liệu là (cũng) có trong bảng cha. Tuy nhiên, việc truy cập các bảng

con một cách trực tiếp là không tự động được phép và có thể đòi hỏi các quyền ưu tiên xa hơn phải được trao.

5.8.1. Các vấn đề còn tồn tại

Lưu ý rằng không phải tất cả các lệnh SQL có khả năng làm việc trong các tôn ti trật tự kế thừa. Các lệnh mà được sử dụng cho việc truy vấn dữ liệu, sửa đổi dữ liệu hoặc sửa đổi sơ đồ (như, SELECT, UPDATE, DELETE, hầu hết các phương án của ALTER TABLE, nhưng không INSERT hoặc ALTER TABLE ... RENAME) thường mặc định đưa vào các bảng con và hỗ trợ ký hiệu ONLY để loại bỏ chúng. Các lệnh mà thực hiện duy trì và tinh chỉnh cơ sở dữ liệu (như, REINDEX, VACUUM) thường chỉ làm việc với các bảng riêng rẽ, vật lý và không hỗ trợ việc lặp đối với các tôn ti trật tự kế thừa. Hành vi tương ứng của từng lệnh riêng rẽ được làm thành tài liệu trong trang tham chiếu của nó (Tham chiếu I, Các lệnh SQL).

Hạn chế nghiêm trọng của tính năng kế thừa là các chỉ số (bao gồm cả các ràng buộc độc nhất) và các ràng buộc khóa ngoại chỉ áp dụng cho các bảng đơn, không cho các bảng con kế thừa. Điều này là đúng cho cả bên tham chiếu và bên được tham chiếu của một ràng buộc khóa ngoại. Vì thế về ví dụ ở trên:

- Nếu chúng ta khai báo cities.name sẽ là UNIQUE hoặc một PRIMARY KEY, thì điều này có thể không dừng bảng capitals khỏi việc có các hàng với các tên đúp bản các hàng trong bảng cities. Và các hàng đúp bản đó có thể mặc định chỉ ra trong các truy vấn từ bảng cities. Trong thực tế, mặc định bảng capitals có thể không có ràng buộc độc nhất nào cả, và vì thế có thể bao gồm nhiều hàng với cùng tên. Bạn có thể thêm một ràng buộc độc nhất vào bảng capitals, nhưng điều này có thể không ngăn được sự đúp bản khi so sánh với bảng cities.
- Tương tự, nếu chúng ta từng phải chỉ định rằng cities.name REFERENCES (THAM CHIẾU) tới một số bảng khác, thì ràng buộc này có thể không tự động nhân giống cho bảng capitals. Trong trường hợp này bạn có thể khắc phục nó bằng việc thêm bằng tay cùng y hệt ràng buộc REFERENCES tới bảng capitals.
- Việc chỉ định rằng cột của bảng khác REFERENCES cities(name) có thể cho phép bảng khác đó ràng buộc các tên thành phố, nhưng không ràng buộc được các tên thủ phủ. Không có sự khắc phục tốt cho trường hợp này.

Những phụ thuộc đó có thể sẽ được sửa trong một số phiên bản trong tương lai, nhưng trong khi chờ đợi thì sự chú ý đáng kể là cần thiết trong việc quyết định liệu sự kế thừa có là hữu dụng hay không cho ứng dụng của bạn.

5.9. Phân vùng

PostgreSQL hỗ trợ việc phân vùng của bảng cơ bản. Phần này mô tả vì sao và bằng cách nào để triển khai việc phân vùng như một phần của thiết kế cơ sở dữ liệu của bạn.

5.9.1. Tổng quan

Việc phân vùng tham chiếu tới việc chia tách những gì về logic là một bảng lớn thành các mẩu nhỏ hơn về mặt vật lý. Việc phân vùng có thể cung cấp vài lợi ích:

- Hiệu năng truy vấn có thể được cải thiện đáng kể trong những tình huống nhất định, đặc biệt khi hầu hết các hàng được truy cập nhiều của bảng là trong một phân vùng duy nhất hoặc một số lượng nhỏ các phân vùng. Việc phân vùng thay thế cho việc dẫn dắt các cột chỉ số, làm giảm kích thước chỉ số và làm cho có khả năng hơn đối với các phần được sử dụng nhiều của các chỉ số phù hợp trong bộ nhớ.
- Khi các truy vấn hoặc các cập nhật truy cập một số phần trăm lớn đối với một phân vùng duy nhất, hiệu năng có thể được cải thiện bằng việc tận dụng sự quét tuần tự phân vùng đó thay vì sử dụng một chỉ số và đọc truy cập ngẫu nhiên bị phân tán khắp toàn bộ bảng.
- Các tải và các sự xóa theo bó có thể được hoàn tất bằng việc thêm hoặc bớt các phân vùng, nếu yêu cầu đó có kế hoạch trong thiết kế phân vùng. ALTER TABLE NO INHERIT và DROP TABLE cả 2 đều nhanh hơn nhiều so với một hoạt động theo bó. Các lệnh đó cũng hoàn toàn tránh được chi phí tổng VACUUM sinh ra vì một bó các lệnh DELETE.
- Các dữ liệu hiếm khi được sử dụng có thể được chuyển đổi sang các phương tiện lưu trữ rẻ hơn và chậm hơn.

Những lợi ích sẽ thường đáng kể chỉ khi một bảng có thể nếu khác đi thì rất lớn. Điểm chính xác mà ở đó một bảng sẽ có lợi từ việc phân vùng phụ thuộc vào ứng dụng, dù một qui tắc ngón tay cái là kích cỡ của bảng sẽ vượt quá bộ nhớ vật lý của máy chủ cơ sở dữ liệu.

Hiện hành, PostgreSQL hỗ trợ việc phân vùng thông qua sự kế thừa của bảng. Từng phân vùng phải được tạo ra như một bảng con của một bảng cha duy nhất. Bản thân bảng cha thường là rỗng; nó tồn tại chỉ để thể hiện toàn bộ tập dữ liệu. Bạn nên quen với sự kế thừa (xem Phần 5.8) trước khi cố gắng thiết lập việc phân vùng.

Các mẫu sau của việc phân vùng có thể được triển khai trong PostgreSQL:

Phân vùng theo khoảng

Bảng được phân vùng thành "các khoảng" được một cột hoặc tập hợp các cột khóa xác định, không có sự chồng lấn giữa các khoảng giá trị được chỉ định cho các phân vùng khác. Ví dụ một người có thể phân vùng theo các khoảng ngày tháng, hoặc theo các khoảng của các mã định danh đối với các đối tượng nghiệp vụ

Phân vùng liệt kê

Bảng được phân vùng bằng việc liệt kê rõ ràng các giá trị chủ chốt nào xuất hiện trong từng phân vùng đó.

5.9.2. Triển khai phân vùng

Để thiết lập một bảng có phân vùng, hãy làm những điều sau:

1. Tạo bảng "chủ" ("master"), từ đó tất cả các phân vùng sẽ kế thừa.

   Bảng này sẽ không chứa dữ liệu. Không định nghĩa bất kỳ ràng buộc kiểm tra nào trong bảng này, trừ phi bạn có ý định chúng sẽ được áp dụng y hệt cho tất cả các phân vùng. Không có nghĩa trong việc định nghĩa bất kỳ chỉ số hoặc hằng số độc nhất nào trong nó.

2. Tạo vài bảng "con" mà từng bảng kế thừa từ bảng chủ. Thông thường, các bảng đó sẽ không thêm bất kỳ cột nào vào tập hợp được kế thừa từ bảng chủ.

   Chúng ta sẽ tham chiếu tới các bảng con như là các phân vùng, dù chúng theo mọi cách là những bảng PostgreSQL thông thường.

3. Thêm các ràng buộc bảng vào các bảng phân vùng để định nghĩa các giá trị khóa được phép trong từng phân vùng.

   Các ví dụ điển hình có thể là:

   ```
   CCHECK ( x = 1 )
   CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
   CHECK ( outletID >= 100 AND outletID < 200 )
   ```

   Hãy chắc chắn các ràng buộc đảm bảo rằng không có sự chồng lấn giữa các giá trị khóa được phép trong các phần khác nhau. Một sai sót phổ biến là thiết lập dải các ràng buộc giống như:

   ```
   CHECK ( outletID BETWEEN 100 AND 200 )
   CHECK ( outletID BETWEEN 200 AND 300 )
   ```

   Điều này là sai vì không rõ phân vùng nào giá trị khóa 200 nằm trong đó.

   Lưu ý là không có sự khác biệt trong cú pháp giữa việc phân vùng theo khoảng và phân vùng liệt kê; những khái niệm đó chỉ là diễn tả.

4. Đối với từng phân vùng, hãy tạo một chỉ số trong (các) cột khóa, cũng như bất kỳ các chỉ số khác mà bạn có thể muốn. (Chỉ số chính không nhất thiết là khắt khe, mà trong hầu hết các kịch bản là hữu dụng. Nếu bạn định để cho các giá trị khóa là độc nhất thì bạn nên luôn tạo một ràng buộc độc nhất hoặc khóa chủ cho từng phân vùng).

5. Như một sự lựa chọn, hãy định nghĩa một trigger hoặc qui tắc để tái định tuyến các dữ liệu được chèn vào bảng chủ tới phân vùng phù hợp.

6. Hãy đảm bảo tham số cấu hình constraint_exclusion không bị vô hiệu hóa trong postgresql.conf. Nếu là thế, các truy vấn sẽ không được tối ưu hóa như mong đợi.

Ví dụ, giả sử chúng ta đang xây dựng một cơ sở dữ liệu cho một công ty làm kem lớn. Công ty đo các nhiệt độ lúc cao điểm mỗi ngày cũng như lượng kem bán trong từng vùng. Về nguyên tắc, chúng ta muốn một bảng giống như:

```
CREATE TABLE measurement (
        city_id int not null,
```

```
        logdate date not null,
        peaktemp int,
        unitsales int
);
```

Chúng ta biết rằng hầu hết các truy vấn sẽ truy cập chỉ các dữ liệu của tuần, tháng, quý trước, vì sử dụng chính bảng này sẽ là để chuẩn bị cho các báo cáo trực tuyến để quản lý. Để giảm số lượng các dữ liệu cũ mà cần phải được lưu trữ, chúng tôi quyết định chỉ giữ các dữ liệu 3 nằm gần đây nhất. Vào đầu mỗi tháng chúng tôi sẽ loại bỏ các dữ liệu của tháng cũ nhất.

Trong tình huống này chúng ta có thể sử dụng việc phân vùng để giúp chúng ta đáp ứng được tất cả các yêu cầu khác nhau của chúng ta đối với những đo đếm cho bảng. Tuân theo các bước được phác họa ở trên, việc phân vùng có thể được thiết lập như sau:

1. Bảng chủ là bảng đo đếm, được khai báo chính xác như ở trên.
2. Tiếp theo chúng ta tạo một phân vùng cho từng tháng hoạt động:

   ```
   CREATE TABLE measurement_y2006m02 ( ) INHERITS (measurement);
   CREATE TABLE measurement_y2006m03 ( ) INHERITS (measurement);
   ...
   CREATE TABLE measurement_y2007m11 ( ) INHERITS (measurement);
   CREATE TABLE measurement_y2007m12 ( ) INHERITS (measurement);
   CREATE TABLE measurement_y2008m01 ( ) INHERITS (measurement);
   ```

   Mỗi trong số các phân vùng là các bảng hoàn chỉnh theo quyền của riêng chúng, nhưng chúng kế thừa các định nghĩa của chúng từ bảng đo đếm.

   Điều này giải quyết một trong những vấn đề của chúng ta: xóa các dữ liệu cũ. Mỗi tháng, tất cả điều chúng ta sẽ cần phải làm là thực hiện một lệnh bỏ bảng DROP TABLE trong bảng con cũ nhất và tạo ra một bảng con mới cho các dữ liệu của tháng mới.

3. Chúng ta phải đưa ra các ràng buộc bảng không chồng lấn. Thay vì chỉ tạo các bảng phân vùng như ở trên, script tạo bảng thực sự sẽ là:

   ```
   CREATE TABLE measurement_y2006m02 (
           CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
   ) INHERITS (measurement);
   CREATE TABLE measurement_y2006m03 (
           CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
   ) INHERITS (measurement);
   ...
   CREATE TABLE measurement_y2007m11 (
           CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
   ) INHERITS (measurement);
   CREATE TABLE measurement_y2007m12 (
           CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
   ) INHERITS (measurement);
   CREATE TABLE measurement_y2008m01 (
           CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
   ) INHERITS (measurement);
   ```

4. Chúng ta có lẽ cũng cần các chỉ số trong các cột khóa:

   ```
   CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
   CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
   ...
   CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
   ```

```
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

Chúng ta chọn không thêm các chỉ số nữa vào thời điểm này.

5. Chúng ta muốn ứng dụng của chúng ta sẽ có khả năng nói INSERT INTO measurement … và có các dữ liệu được tái định tuyến vào trong bảng phân vùng phù hợp. Chúng ta có thể dàn xếp bằng việc gắn một hàm trigger phù hợp tới bảng chủ.

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
        RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Sau khi tạo hàm đó, chúng ta tạo một trigger gọi hàm trigger đó:

```
CREATE TRIGGER insert_measurement_trigger
        BEFORE INSERT ON measurement
        FOR EACH ROW EXECUTE PROCEDURE measurement_insert_trigger();
```

Chúng ta phải tái định nghĩa hàm trigger mỗi tháng sao cho nó luôn chỉ tới phân vùng hiện hành. Tuy nhiên, định nghĩa trigger không cần phải được cập nhật.

Chúng ta có thể muốn chèn các dữ liệu và để máy chủ tự động định vị phân vùng vào hàng nào mà sẽ được thêm vào. Chúng ta có thể làm điều này bằng một hàm trigger phức tạp hơn, ví dụ:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
        IF ( NEW.logdate >= DATE '2006-02-01' AND
                NEW.logdate < DATE '2006-03-01' ) THEN
                INSERT INTO measurement_y2006m02 VALUES (NEW.*);
        ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
                NEW.logdate < DATE '2006-04-01' ) THEN
                INSERT INTO measurement_y2006m03 VALUES (NEW.*);
                ...
        ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
                NEW.logdate < DATE '2008-02-01' ) THEN
                INSERT INTO measurement_y2008m01 VALUES (NEW.*);
        ELSE
                RAISE EXCEPTION 'Date out of range. Fix the measurement_insert_trigger()
                function!';
                END IF;
                RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Định nghĩa trigger là y hệt như trước. Lưu ý rằng từng kiểm thử IF phải chính xác khớp với ràng buộc kiểm tra CHECK cho phân vùng của nó.

Trong khi hàm này là phức tạp hơn so với trường hợp tháng duy nhất, thì không cần phải được cập nhật thường xuyên, vì các nhánh có thể được thêm vào trước khi cần thiết.

> **Lưu ý**: Trong thực tế có thể là tốt nhất để kiểm tra phân vùng mới nhất trước, nếu hầu hết các vụ chèn đi vào phân vùng đó. Để đơn giản, chúng tôi đã chỉ ra các kiểm thử trigger theo cùng trật tự như trong các phần khác của ví dụ này.

Như chúng ta có thể thấy, một sơ đồ phân vùng phức tạp có thể đòi hỏi một số lượng các DDL đáng kể. Trong ví dụ ở trên chúng ta có thể đang tạo ra một phân vùng mới mỗi tháng, nên có lẽ là khôn ngoan để viết một script mà tạo ra DDL theo yêu cầu một cách tự động.

5.9.3. Quản lý phân vùng

Thông thường tập hợp các phân vùng được thiết lập khi việc định nghĩa bảng ban đầu không có ý định sẽ là tĩnh. Là phổ biến để muốn loại bỏ các phân vùng các dữ liệu cũ và định kỳ thêm các phân vùng mới cho các dữ liệu mới. Một trong những ưu điểm quan trọng nhất của việc phân vùng chính xác là nó cho phép điều này nếu khác đi sẽ là tác vụ đau đớn phải được thực hiện gần như cùng một lúc bằng việc điều khiển bằng tay cấu trúc phân vùng, thay vì việc loại bỏ một cách vật lý lượng lớn các dữ liệu.

Tùy chọn đơn giản nhất cho việc loại bỏ các dữ liệu cũ đơn giản là bỏ phân vùng mà không còn cần thiết nữa:

```
DROP TABLE measurement_y2006m02;
```

Điều này có thể xóa rất nhanh hàng triệu bản ghi vì nó không phải xóa riêng lẻ từng bản ghi.

Lựa chọn khác thường được ưu tiên là loại bỏ phân vùng từ bảng được phân vùng nhưng giữ lại sự truy cập tới nó như là một bảng theo quyền của riêng nó:

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

Điều này cho phép các hoạt động xa hơn sẽ được thực hiện trong cơ sở dữ liệu trước khi nó bị bỏ. Ví dụ, đây thường là thời điểm hữu dụng để sao lưu các dữ liệu bằng lệnh COPY, pg_dump hoặc các công cụ tương tự. Có lẽ cũng là thời điểm hữu dụng để tổng hợp dữ liệu trong các định dạng nhỏ hơn, thực hiện những điều khiển dữ liệu khác, hoặc chạy các báo cáo. Tương tự chúng ta có thể thêm một phân vùng mới để điều khiển các dữ liệu mới. Chúng ta có thể tạo một phân vùng rỗng trong bảng được phân vùng hệt như các phân vùng ban đầu đã được tạo ra ở trên:

```
CREATE TABLE measurement_y2008m02 (
        CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

Như một lựa chọn, đôi lúc là thuận tiện hơn để tạo ra bảng mới bên ngoài cấu trúc phân vùng đó, và làm cho nó thành một phân vùng phù hợp hơn sau này. Điều này cho phép các dữ liệu sẽ được tải lên, được kiểm tra và được biến đổi trước khi nó xuất hiện trong bảng được phân vùng:

```
CREATE TABLE measurement_y2008m02
        (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
        CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work (có khả năng một số công việc chuẩn bị dữ liệu khác)
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.9.4. Loại trừ ràng buộc và phân vùng

Loại trừ ràng buộc là một kỹ thuật tối ưu hóa truy vấn mà cải thiện hiệu năng cho các bảng được phân vùng được định nghĩa theo cách thức được mô tả ở trên. Như một ví dụ:

```
SET constraint_exclusion = on;

SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

Không có sự loại trừ ràng buộc, thì truy vấn ở trên có thể quét từng trong các phân vùng của bảng đo đếm. Với sự loại trừ ràng buộc được kích hoạt, trình hoạch định (planner) sẽ xem xét các ràng buộc của từng phân vùng và cố gắng chứng minh rằng phân vùng cần không được quét vì nó có thể không có bất kỳ hàng nào đáp ứng được mệnh đề WHERE của truy vấn đó. Khi trình hoạch định có thể chứng minh được điều này, nó loại trừ phân vùng khỏi kế hoạch truy vấn.

Bạn có thể sử dụng lệnh EXPLAIN để chỉ ra sự khác biệt giữa một kế hoạch với constraint_exclusion được bật (kích hoạt) và một kế hoạch mà nó bị tắt (giải hoạt). Một kế hoạch điển hình không được tối ưu hóa cho dạng thiết lập này của bảng là:

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
QUERY PLAN
---------------------------------------------------------------------------------------------
Aggregate (cost=158.66..158.68 rows=1 width=0)
        -> Append (cost=0.00..151.88 rows=2715 width=0)
                -> Seq Scan on measurement (cost=0.00..30.38 rows=543 width=0)
                Filter: (logdate >= '2008-01-01'::date)
                -> Seq Scan on measurement_y2006m02 measurement (cost=0.00..30.38 rows=543
                width=Filter: (logdate >= '2008-01-01'::date)
                -> Seq Scan on measurement_y2006m03 measurement (cost=0.00..30.38 rows=543
                width=Filter: (logdate >= '2008-01-01'::date)
...
                -> Seq Scan on measurement_y2007m12 measurement (cost=0.00..30.38 rows=543
                width=Filter: (logdate >= '2008-01-01'::date)
                -> Seq Scan on measurement_y2008m01 measurement (cost=0.00..30.38 rows=543
                width=Filter: (logdate >= '2008-01-01'::date)
```

Một số hoặc tất cả các phân vùng có thể sử dụng các vụ quét chỉ số thay vì các vụ quét tuần tự toàn bộ bảng, nhưng điểm mấu chốt ở đây là hoàn toàn không cần phải quét các phân vùng cũ hơn để trả lời cho truy vấn này. Khi chúng ta kích hoạt loại trừ ràng buộc, chúng ta có một kế hoạch rẻ hơn đáng kể mà sẽ đưa ra câu trả lời y hệt:

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
QUERY PLAN
---------------------------------------------------------------------------------------------
Aggregate (cost=63.47..63.48 rows=1 width=0)
        -> Append (cost=0.00..60.75 rows=1086 width=0)
                -> Seq Scan on measurement (cost=0.00..30.38 rows=543 width=0)
                Filter: (logdate >= '2008-01-01'::date)
                -> Seq Scan on measurement_y2008m01 measurement (cost=0.00..30.38 rows=543
                width=Filter: (logdate >= '2008-01-01'::date)
```

Lưu ý rằng sự loại trừ ràng buộc chỉ được dẫn dắt bằng các ràng buộc kiểm tra CHECK, không bằng sự hiện diện của các chỉ số. Vì thế không cần thiết phải định nghĩa các chỉ số trong các cột khóa.

Liệu một chỉ số có cần phải được tạo ra cho một phân vùng được đưa ra hay không sẽ phụ thuộc vào việc liệu bạn có mong đợi rằng các truy vấn quét phân vùng đó sẽ, nói chung, quét một phần lớn phân vùng đó hay chỉ một phân vùng con. Một chỉ số sẽ hữu dụng trong trường hợp sau nhưng không trong trường hợp trước.

Thiết lập mặc định (và được khuyến cáo) của constraint_exclusion thường hoặc không bật (on) cũng không tắt (off), nhưng một thiết lập trung gian được gọi là partition, nó làm cho kỹ thuật đó sẽ chỉ được áp dụng cho các truy vấn mà có khả năng sẽ làm việc được trong các bảng được phân vùng. Thiết lập bật (on) làm cho trình hoạch định kiểm tra các ràng buộc CHECK trong tất cả các truy vấn, thậm chí cả các truy vấn đơn giản mà khó có thể có lợi.

5.9.5. Phương pháp phân vùng khác

Một tiếp cận khác để tái định tuyến các vụ chèn vào bảng phân vùng phù hợp là thiết lập các qui tắc, thay vì một trigger, trong bảng chủ. Ví dụ:

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
        ( logdate >= DATE ’2006-02-01’ AND logdate < DATE ’2006-03-01’ )
DO INSTEAD
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
        ( logdate >= DATE ’2008-01-01’ AND logdate < DATE ’2008-02-01’ )
DO INSTEAD
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

Một qui tắc có chi phí tổng nhiều hơn đáng kể so với một trigger, nhưng tổng chi phí được trả một lần theo từng truy vấn thay vì một lần theo từng hàng, nên phương pháp này có thể là có ưu thế cho những tình huống chèn theo bó. Tuy nhiên, trong hầu hết các trường hợp, phương pháp trigger sẽ đưa ra hiệu năng tốt hơn.

Hãy hiểu rằng lệnh COPY sẽ bỏ qua các qui tắc. Nếu bạn muốn sử dụng COPY để chèn dữ liệu, thì bạn sẽ cần phải sao chép vào bảng phân vùng đúng thay vì vào bảng chủ. COPY sẽ đốt trigger, nên bạn có thể sử dụng nó bình thường nếu bạn sử dụng tiếp cận trigger.

Nhược điểm khác của tiếp cận qui tắc này là không có cách thức đơn giản nào để ép một lỗi nếu tập hợp các qui tắc không bao trùm ngày tháng chèn; ngày tháng sẽ âm thầm đi vào bảng chủ.

Việc phân vùng cũng có thể được dàn xếp bằng việc sử dụng một kiểu nhìn UNION ALL, thay vì kế thừa bảng. Ví dụ,

```
CREATE VIEW measurement AS
        SELECT * FROM measurement_y2006m02
UNION ALL SELECT * FROM measurement_y2006m03
...
UNION ALL SELECT * FROM measurement_y2007m11
UNION ALL SELECT * FROM measurement_y2007m12
UNION ALL SELECT * FROM measurement_y2008m01;
```

Tuy nhiên, sự cần thiết phải tái tạo kiểu nhìn sẽ thêm một bước nữa vào việc bổ sung và loại bỏ các

phân vùng riêng rẽ của tập hợp dữ liệu. Trong thực tế phương pháp này có ít điều để khuyến cáo nó so với việc sử dụng sự kế thừa.

5.9.6. Các vấn đề còn tồn tại

Các vấn đề còn tồn tại sau đây áp dụng cho các bảng được phân vùng:

- Không có cách nào để kiểm tra tất cả các ràng buộc CHECK có là loại trừ lẫn nhau hay không. Là an toàn hơn để tạo mã mà sinh ra các phân vùng và tạo ra và/hoặc sửa đổi các đối tượng có liên quan so với để viết từng thứ bằng tay.
- Các sơ đồ chỉ ra ở đây giả thiết rằng (các) cột khóa phân vùng của một hàng không bao giờ thay đổi, hoặc ít nhất không thay đổi đủ để yêu cầu nó phải chuyển tới phân vùng khác. Một lệnh UPDATE có ý định thực hiện điều đó sẽ hỏng vì các ràng buộc CHECK. Nếu bạn cần điều khiển các trường hợp như vậy, thì bạn có thể đặt các trigger cập nhật phù hợp trong các bảng phân vùng đó, nhưng nó quản lý cấu trúc đó phức tạp hơn nhiều.
- Nếu bạn đang sử dụng bằng tay các lệnh VACUUM hoặc ANALYZE, đừng quên rằng bạn cần chạy chúng trong từng phân vùng một cách riêng rẽ. Một lệnh giống như:

  ```
  ANALYZE measurement;
  ```

  sẽ chỉ xử lý bảng chủ.

Các vấn đề còn tồn tại sau đây áp dụng cho sự loại trừ các ràng buộc:

- Loại trừ ràng buộc chỉ làm việc khi mệnh đề WHERE của truy vấn có các ràng buộc. Một truy vấn có tham số sẽ không được tối ưu hóa, vì trình hoạch định không thể biết các phân vùng nào giá trị tham số có thể chọn ở lúc chạy. Vì cùng lý do đó, các hàm "ổn định" như CURRENT_DATE phải được tránh.
- Hãy giữ cho việc phân vùng các ràng buộc là đơn giản, nếu không thì trình hoạch định có thể không có khả năng để chứng minh rằng các phân vùng không cần phải được thăm viếng. Hãy sử dụng các điều kiện ngang bằng đơn giản cho việc phân vùng theo liệt kê, hoặc các kiểm thử khoảng đơn giản cho việc phân vùng theo khoảng, như được minh họa trong các ví dụ ở trước. Một qui tắc ngón tay cái tốt là việc phân vùng các ràng buộc nên chỉ bao gồm những so sánh của việc phân vùng (các) cột với các hằng số bằng việc sử dụng các toán tử đánh chỉ số được theo Btree.
- Tất cả các ràng buộc trong tất cả các phân vùng của bảng chủ được kiểm tra trong quá trình loại trừ ràng buộc, sao cho các số phân vùng lớn có khả năng làm gia tăng thời gian lên kế hoạch truy vấn một cách đáng kể. Việc phân vùng bằng việc sử dụng các kỹ thuật đó sẽ làm việc tốt với, có lẽ, hàng trăm phân vùng; không cố sử dụng nhiều ngàn phân vùng.

5.10. Đối tượng cơ sở dữ liệu khác

Các bảng là các đối tượng trọng tâm trong cấu trúc một cơ sở dữ liệu quan hệ, vì chúng lưu trữ các

dữ liệu của bạn. Nhưng chúng không chỉ là các đối tượng mà tồn tại trong một cơ sở dữ liệu. Nhiều dạng đối tượng khác có thể được tạo ra để sử dụng và quản lý các dữ liệu có hiệu quả và thuận tiện hơn. Chúng không được thảo luận trong chương này, nhưng chúng tôi cho bạn một danh sách ở đây để bạn nhận biết được về những gì có khả năng:

- Các kiểu nhìn
- Các hàm và toán tử
- Các dạng và miền dữ liệu
- Các trigger và các qui tắc viết lại

Thông tin chi tiết về các chủ đề đó có trong Phần V.

5.11. Theo dõi sự phụ thuộc

Khi bạn tạo các cấu trúc cơ sở dữ liệu phức tạp có liên quan tới nhiều bảng với các ràng buộc khóa ngoại, kiểu nhìn, trigger, hàm, …, bạn ngầm tạo ra một mạng các phụ thuộc giữa các đối tượng đó. Ví dụ, một bảng với một ràng buộc khóa ngoại phụ thuộc vào bảng mà nó tham chiếu.

Để đảm bảo tính toàn vẹn của toàn bộ cấu trúc cơ sở dữ liệu, PostgreSQL chắc chắn là bạn không thể bỏ các đối tượng mà các đối tượng khác còn phụ thuộc vào. Ví dụ, việc định bỏ bảng các sản phẩm mà chúng ta đã xem xét trong Phần 5.3.5, với bảng các đơn hàng phụ thuộc vào nó, có thể tạo ra một thông điệp lỗi như thế này:

```
DROP TABLE products;
NOTICE: constraint orders_product_no_fkey on table orders depends on table products
ERROR: cannot drop table products because other objects depend on it
HINT: Use DROP ... CASCADE to drop the dependent objects too.
```

Thông điệp lỗi có một gợi ý hữu dụng: nếu bạn không muốn làm phiền việc xóa tất cả các đối tượng phụ thuộc một các riêng rẽ, bạn có thể chạy:

```
DROP TABLE products CASCADE;
```

và tất cả các đối tượng phụ thuộc sẽ bị loại bỏ. Trong trường hợp này nó không loại bỏ bảng orders, nó chỉ loại bỏ ràng buộc khóa ngoại. (nếu bạn muốn kiểm tra xem DROP ... CASCADE sẽ làm được gì, hãy chạy DROP mà không có CASCADE và đọc các thông điệp lưu ý NOTICE).

Tất cả các lệnh DROP trong PostgreSQL hỗ trợ việc chỉ định CASCADE. Tất nhiên, bản chất tự nhiên của các phụ thuộc có khả năng là khác nhau với dạng đối tượng. Bạn cũng có thể viết RESTRICT thay vì CASCADE để có được hành vi mặc định, nó là để ngăn chặn việc bỏ các đối tượng mà các đối tượng khác còn phụ thuộc vào.

> **Lưu ý**: Theo tiêu chuẩn SQL, việc chỉ đỉnh hoặc RESTRICT hoặc CASCADE được yêu cầu. Không hệ thống cơ sở dữ liệu nào thực sự bắt tuân theo qui tắc đó, nhưng liệu hành vi mặc định là RESTRICT hay CASCADE sẽ là khác nhau với các hệ thống khác nhau.
>
> **Lưu ý**: Các phụ thuộc ràng buộc khóa ngoại và các phụ thuộc cột liên tục từ PostgreSQL

phiên bản trước 7.3 sẽ không được duy trì hoặc tạo ra trong quá trình nâng cấp. Tất cả các dạng phụ thuộc khác sẽ được tạo ra một cách phù hợp trong một bản nâng cấp từ cơ sở dữ liệu trước 7.3.

Chương 6. Điều khiển dữ liệu

Chương trước đã thảo luận cách tạo ra các bảng và các cấu trúc khác để lưu trữ dữ liệu của bạn. Bây giờ đã tới lúc điền dữ liệu vào các bảng. Chương này đề cập tới cách chèn, cập nhật và xóa các dữ liệu của bảng. Chương sau chương này cuối cùng sẽ giải thích cách trích xuất dữ liệu bền lâu từ cơ sở dữ liệu.

6.1. Chèn dữ liệu

Khi một bảng được tạo ra, nó không có dữ liệu. Điều đầu tiên phải làm trước khi một cơ sở dữ liệu có thể được sử dụng nhiều là chèn dữ liệu vào. Dữ liệu, về khái niệm, được chèn vào từng hàng một theo thời gian. Tất nhiên bạn cũng có thể chèn nhiều hơn một hàng, nhưng không có cách nào để chèn ít hơn một hàng cả. Thậm chí nếu bạn biết chỉ một số giá trị cột, thì một hàng hoàn chỉnh phải được tạo ra.

Để tạo ra một hàng mới, hãy sử dụng lệnh chèn INSERT. Lệnh này đòi hỏi tên bảng và các giá trị cột. Ví dụ, hãy xem xét bảng các sản phẩm từ Chương 5:

```
CREATE TABLE products (
        product_no integer,
        name text,
        price numeric
);
```

Một lệnh ví dụ để chèn một hàng có thể là:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

Các giá trị dữ liệu được liệt kê theo trật tự trong đó các cột xuất hiện trong bảng, cách nhau bằng các dấu phẩy.

Thông thường, các giá trị dữ liệu sẽ là các hằng, nhưng các biểu thức vô hướng cũng được phép. Cú pháp ở trên có nhược điểm là bạn cần biết thứ tự của các cột trong bảng. Để tránh điều này bạn cũng có thể liệt kê các cột một cách rõ ràng. Ví dụ, cả 2 lệnh sau có cùng hiệu quả như lệnh ở trên:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

Nhiều người sử dụng coi nó là thực tiễn tốt để luôn liệt kê các tên cột.

Nếu bạn không có các giá trị cho tất cả các cột, thì bạn có thể bỏ qua một số chúng. Trong trường hợp đó, các cột sẽ được điền với các giá trị mặc định. Ví dụ:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

Mẫu thứ 2 là một mở rộng của PostgreSQL. Nó điền các cột từ trái qua với càng nhiều giá trị càng tốt, và phần còn lại sẽ được mặc định.

Để làm rõ, bạn cũng có thể yêu cầu các giá trị mặc định một cách rõ ràng, cho các cột riêng rẽ hoặc cho toàn bộ hàng:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

Bạn có thể chèn nhiều hàng trong một lệnh duy nhất;

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

Mẹo: Khi chèn nhiều dữ liệu cùng một lúc, xem xét sử dụng lệnh COPY. Không mềm dẻo như lệnh INSERT, nhưng là có hiệu quả hơn. Hãy tham chiếu tới Phần 14.4 để có thêm thông tin về việc cải thiện hiệu năng tải theo bó.

6.2. Cập nhật dữ liệu

Sửa đổi các dữ liệu đã có rồi trong cơ sở dữ liệu được tham chiếu tới như là việc cập nhật. Bạn có thể cập nhật các hàng riêng rẽ, tất cả các hàng trong một bảng, hoặc một tập con của tất cả các hàng. Từng cột có thể được cập nhật một cách riêng rẽ; các cột khác sẽ không bị ảnh hưởng.

Để cập nhật các hàng đang tồn tại, hãy sử dụng lệnh cập nhật UPDATE. Lệnh này đòi hỏi 3 mẩu tin:

1. Tên bảng và cột để cập nhật
2. Giá trị mới của cột
3. (Các) hàng nào để cập nhật

Nhớ lại từ Chương 5 rằng SQL, nói chung, không cung cấp một mã định danh độc nhất cho các hàng. Vì thế không luôn có khả năng để chỉ định trực tiếp hàng nào để cập nhật. Thay vào đó, bạn chỉ định các điều kiện nào một hàng phải đáp ứng để được cập nhật. Chỉ nếu bạn có một khóa chủ trong bảng (độc lập với việc liệu bạn đã khai báo nó hay chưa) có thể bạn đề cập một cách tin cậy các hàng riêng rẽ bằng việc chọn một điều kiện mà trùng khớp với khóa chủ. Các công cụ truy cập cơ sở dữ liệu đồ họa dựa vào thực tế này để cho phép bạn cập nhật các hàng một cách riêng rẽ.

Ví dụ, lệnh này cập nhật tất cả các sản phẩm mà có giá là 5 sẽ có giá lên là 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

Điều này có thể làm cho 0, 1 hoặc nhiều hàng sẽ được cập nhật. Không là lỗi để cố gắng một cập nhật mà không khớp với hàng nào cả.

Hãy nhìn vào lệnh đó một cách chi tiết. Trước hết là từ khóa UPDATE theo sau là tên bảng. Thông thường, tên bảng có thể có đủ điều kiện như là sơ đồ, nếu không thì nó được tra trong đường dẫn. Cạnh từ khóa SET đi theo sau là tên cột, một dấu bằng (=), và giá trị cột mới. Giá trị cột mới đó có thể là bất kỳ biểu thức vô hướng nào, không chỉ là một hằng số. Ví dụ, nếu bạn muốn nâng giá thành của tất cả các sản phẩm lên 10% thì bạn có thể sử dụng:

```
UPDATE products SET price = price * 1.10;
```

Như bạn thấy, biểu thức cho giá trị mới có thể tham chiếu tới (các) giá trị đang tồn tại trong hàng. Chúng ta cũng để lại mệnh đề WHERE. Nếu nó bị bỏ qua, thì có nghĩa là tất cả các hàng trong bảng

được cập nhật. Nếu nó hiện diện, thì chỉ những hàng nào mà khớp với điều kiện WHERE sẽ được cập nhật. Lưu ý rằng dấu bằng trong mệnh đề SET là một chỉ định trong khi dấu hiệu trong mệnh đề WHERE là một so sánh, nhưng điều này không tạo ra bất kỳ sự tù mù nào. Tất nhiên, điều kiện WHERE không phải là một kiểm thử về sự ngang bằng nhau. Nhiều toán tử khác cũng sẵn sàng (xem Chương 9). Nhưng biểu thức đó cần phải đánh giá theo một kết quả Boolean.

Bạn có thể cập nhật nhiều hơn một cột trong một lệnh UPDATE bằng việc liệt kê nhiều hơn một chỉ định trong mệnh đề SET. Ví dụ:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. Xóa dữ liệu

Cho tới nay chúng ta đã giải thích cách để thêm dữ liệu vào các bảng và cách để thay đổi dữ liệu. Điều còn lại là thảo luận cách để loại bỏ dữ liệu mà không còn cần thiết nữa. Hệt như việc thêm dữ liệu chỉ có khả năng trong toàn bộ các hàng, bạn chỉ có thể loại bỏ toàn bộ các hàng khỏi một bảng. Trong phần trước chúng ta đã giải thích rằng SQL không đưa ra cách thức để trực tiếp đề cập tới các hàng riêng rẽ. Vì thế, việc loại bỏ các hàng chỉ có thể được thực hiện bằng việc chỉ định các điều kiện mà các hàng sẽ được loại bỏ phải trùng khớp. Nếu bạn có một khóa chủ trong bảng thì bạn có thể chỉ định hàng chính xác. Nhưng bạn cũng có thể loại bỏ nhóm các hàng khớp với một điều kiện, hoặc bạn có thể loại bỏ tất cả các hàng trong một bảng cùng một lần.

Bạn hãy sử dụng lệnh xóa DELETE để loại bỏ các hàng; cú pháp rất tương tự như với lệnh UPDATE.

Ví dụ, để loại bỏ tất cả các hàng khỏi bảng các sản phẩm mà có giá là 10, hãy sử dụng:

```
DELETE FROM products WHERE price = 10;
```

Nếu bạn đơn giản viết:

```
DELETE FROM products;
```

thì tất cả các hàng trong bảng đó sẽ bị xóa! Vấn đề còn lại là của lập trình viên.

Chương 7. Truy vấn

Các chương trước đã giải thích cách để tạo các bảng, cách đề điền đầy dữ liệu cho chúng, và cách để điều khiển các dữ liệu đó. Bây giờ chúng ta cuối cùng thảo luận cách để truy xuất dữ liệu từ cơ sở dữ liệu.

7.1. Tổng quan

Qui trình của việc truy xuất hoặc lệnh để truy xuất (retrieve) dữ liệu từ một cơ sở dữ liệu được gọi là một *truy vấn* (*query*). Trong SQL thì lệnh SELECT được sử dụng để chỉ định các truy vấn. Cú pháp thông thường của lệnh SELECT là

[WITH with_queries] SELECT select_list FROM table_expression [sort_specification]

Các phần sau đây mô tả các chi tiết của danh sách chọn, biểu thức của bảng, và đặc tả các kiểu. Các truy vấn WITH được đề cập cuối cùng vì chúng là một tính năng cao cấp.

Một dạng truy vấn đơn giản có dạng:

```
SELECT * FROM table1;
```

Giả thiết rằng có một bảng có tên là table1, thì lệnh này có thể truy xuất tất cả các hàng và tất cả các cột từ table1. (Phương pháp truy xuất phụ thuộc vào ứng dụng máy trạm. Ví dụ, chương trình psql với hiển thị một bảng dạng ASCII trên màn hình, trong khi các thư viện máy trạm sẽ đưa ra các hàm để truy xuất các giá trị riêng rẽ từ kết quả của truy vấn đó). Đặc tả của danh sách chọn * nghĩa là tất cả các cột mà biểu thức bảng ngẫu nhiên đưa ra. Một danh sách chọn cũng có thể chọn một tập con của các cột có sẵn hoặc thực hiện các tính toán bằng việc sử dụng các cột. Ví dụ, nếu table1 có các cột có tên là a, b, và c (và có lẽ các tên khác nữa) thì bạn có thể làm truy vấn sau:

```
SELECT a, b + c FROM table1;
```

(giả thiết rằng a, b và c là dạng dữ liệu số). Xem Phần 7.3 để có thêm các chi tiết.

FROM table1 là một dạng biểu thức bảng đơn giản: nó chỉ đọc một bảng. Nói chung, các biểu thức bảng có thể là các cấu trúc phức tạp của các bảng cơ bản, các ghép nối liên kết và các truy vấn con. Nhưng bạn cũng có thể bỏ qua toàn bộ biểu thức bảng và sử dụng lệnh SELECT như một máy tính:

```
SELECT 3 * 4;
```

Điều này là hữu dụng hơn nếu các biểu thức trong danh sách chọn trả về các kết quả khác nhau. Ví dụ, bạn có thể gọi một hàm theo cách này:

```
SELECT random();
```

7.2. Biểu thức bảng

Một *biểu thức bảng* tính toán một bảng. Biểu thức bảng bao gồm một mệnh đề FROM mà tùy chọn đi theo là các mệnh đề WHERE, GROUP BY, và HAVING. Các biểu thức bảng thông thường đơn giản tham chiếu tới một bảng trên đĩa, một cái gọi là bảng cơ sở, nhưng nhiều biểu thức phức tạp hơn có thể

được sử dụng để sửa đổi hoặc kết hợp các bảng cơ sở theo các cách thức khác nhau.

Các mệnh đề tùy chọn WHERE, GROUP BY,và HAVING trong biểu thức bảng chỉ định một đường tới các biến đổi kế tiếp nhau được thực hiện trong bảng được dẫn xuất trong mệnh đề FROM. Tất cả những biến đổi đó tạo ra một bảng ảo cung cấp các hàng mà được truyền tới danh sách chọn để tính toán các hàng đầu ra của truy vấn.

7.2.1. Mệnh đề FROM

Mệnh đề FROM xuất phát từ một hoặc nhiều bảng được đưa ra theo một danh sách tham chiếu các bảng cách biệt nhau bằng dấu phẩy.

FROM *table_reference [, table_reference [, ...]]*

Một tham chiếu bảng có thể là một tên bảng (có khả năng là sơ đồ đủ điều kiện), hoặc một bảng dẫn xuất như một truy vấn con, một liên kết bảng hoặc các tổ hợp phức tạp của chúng. Nếu nhiều hơn một tham chiếu bảng được liệt kê trong mệnh đề FROM thì chúng được liên kết chéo (xem bên dưới) để tạo thành bảng trung gian ảo mà có thể sau đó tuân theo những biến đổi của các mệnh đề WHERE, GROUP BY, và HAVING và cuối cùng là kết quả của toàn bộ biểu thức bảng.

Khi một tham chiếu bảng gọi tên một bảng là cha của tôn ti trật tự kế thừa của một bảng, thì từ khóa ONLY đi trước tên bảng đó. Tuy nhiên, tham chiếu đó chỉ tạo ra các cột mà xuất hiện trong bảng có tên đó - bất kỳ cột nào được bổ sung trong các bảng con cũng sẽ bị bỏ qua.

Thay vì viết ONLY trước tên bảng, bạn có thể viết * sau tên bảng đó để chỉ định một cách rõ ràng rằng các bảng con đã được đưa vào. Việc viết * là không nhất thiết vì hành vi đó là mặc định (trừ phi bạn đã thay đổi thiết lập lựa chọn cấu hình của sql_inheritance). Tuy nhiên việc viết * có thể là hữu dụng để nhấn mạnh rằng các bảng bổ sung thêm sẽ được tìm kiếm.

7.2.1.1. Bảng kết nối

Một bảng được kết nối là một bảng được dẫn xuất từ 2 bảng khác (thực hoặc được dẫn xuất) theo các qui tắc của dạng kết nối đặc biệt, vòng ngoài, và các liên kết chéo là sẵn sàng.

Các dạng liên kết

Liên kết chéo

T1 CROSS JOIN T2

Đối với từng sự kết hợp có khả năng của các hàng từ T1 và T2 (như, một sản phẩm Cartesian [Đề các]), bảng được kết nối sẽ có một hàng gồm tất cả các cột trong T1 theo sau là tất cả các cột trong T2. Nếu các bảng có N và M hàng một cách tương ứng, thì bảng được liên kết sẽ có N * M hàng.

FROM T1 CROSS JOIN T2 is equivalent to FROM T1, T2 . It is also equivalent to FROM T1 INNER JOIN T2 ON TRUE (see below).

Các liên kết có đủ điều kiện

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

Các từ vòng trong INNER và vòng ngoài OUTER là tùy chọn trong tất cả các mẫu. INNER là mặc định; LEFT, RIGHT, và FULL ngụ ý một liên kết vòng ngoài.

Điều kiện liên kết được chỉ định trong mệnh đề ON hoặc USING, hoặc ẩn với từ NATURAL.

Điều kiện liên kết xác định các hàng nào từ 2 bảng nguồn được xem là "khớp", như được giải thích chi tiết bên dưới.

Mệnh đề ON là dạng điều kiện liên kết chung nhất: nó lấy một biểu thức giá trị Boolean của dạng y hệt như được sử dụng trong mệnh đề WHERE. Một cặp các hàng từ T1 và T2 khớp nếu biểu thức ON đánh giá là đúng đối với chúng.

USING là ký hiệu tốc ký: nó lấy một danh sách các tên cột cách nhau bằng một dấu phẩy mà các bảng được liên kết phải có, nói chung, và tạo nên một điều kiện liên kết chỉ định sự ngang bằng của từng trong các cặp các cột đó. Hơn nữa, đầu ra của JOIN USING có một cột cho từng trong các cặp ngang bằng của các cột đầu vào, theo sau là các cột còn lại từ từng bảng. Vì thế, USING (a, b, c) là tương đương với ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c) với ngoại lệ là nếu ON được sử dụng thì sẽ là 2 cột a, b, và c trong kết quả, trong đó với USING sẽ chỉ là một của từng bảng (và chúng sẽ xuất hiện trước nếu SELECT * được sử dụng). Cuối cùng, NATURAL là một dạng tốc ký của USING: nó hình thành một danh sách USING bao gồm tất cả các tên cột xuất hiện trong cả 2 bảng đầu vào. Như với USING, các cột đó chỉ xuất hiện một lần trong bảng đầu ra.

Các dạng có khả năng của liên kết đủ điều kiện là:

INNER JOIN (Liên kết bên trong)

Đối với từng hàng R1 của bảng T1, bảng được liên kết có một hàng cho từng hàng trong bảng T2 thỏa mãn điều kiện liên kết với R1.

LEFT OUTER JOIN (Liên kết trái bên ngoài)

Trước tiên, một liên kết bên trong được thực hiện. Sau đó, đối với từng hàng trong T1 mà không làm thỏa mãn điều kiện liên kết với bất kỳ hàng nào trong T2, thì một hàng được liên kết được thêm vào với các giá trị null trong các cột của T2.

RIGHT OUTER JOIN (Liên kết phải bên ngoài)

Trước tiên, một liên kết bên trong được thực hiện. Sau đó, đối với từng hàng trong T2 mà không làm thỏa mãn điều kiện liên kết với bất kỳ hàng nào trong T1, thì một hàng được liên kết được thêm vào với các giá trị null trong các cột của T1.

FULL OUTER JOIN (Liên kết đầy đủ bên ngoài)

Trước hết, một liên kết bên trong được thực hiện. Sau đó, đối với từng hàng trong T1 mà không làm thỏa mãn điều kiện liên kết với bất kỳ hàng nào trong T2, thì một hàng được liên kết được thêm vào với các giá trị null trong các cột T2. Hơn nữa, đối với từng hàng của T2

mà không làm thỏa mãn điều kiện liên kết với bất kỳ hàng nào trong T1, thì một hàng được liên kết với các giá trị null trong các cột T1 sẽ được thêm vào.

Các liên kết của tất cả các dạng có thể được nối thành chuỗi cùng nhau hoặc được lồng: cả T1 và T2 đều có thể là các bảng được liên kết. Các dấu ngoặc đơn có thể được sử dụng xung quanh các mệnh đề JOIN để kiểm soát trật tự liên kết. Thiếu các dấu ngoặc đơn thì các mệnh đề JOIN lồng nhau từ trái qua phải.

Để đặt điều này cùng nhau, giả thiết chúng ta có các bảng t1:

```
num | name
------+------
    1 | a
    2 | b
    3 | c
```

và t2:

```
num | value
------+-------
    1 | xxx
    3 | yyy
    5 | zzz
```

sau đó chúng ta có các kết quả sau cho các liên kết khác nhau:

```
=> SELECT * FROM t1 CROSS JOIN t2;

num | name | num | value
------+--------+-------+----------
    1 |      a |     1 | xxx
    1 |      a |     3 | yyy
    1 |      a |     5 | zzz
    2 |      b |     1 | xxx
    2 |      b |     3 | yyy
    2 |      b |     5 | zzz
    3 |      c |     1 | xxx
    3 |      c |     3 | yyy
    3 |      c |     5 | zzz
(9 rows)
=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;

num | name | num | value
------+--------+-------+----------
    1 |      a |     1 | xxx
    3 |      c |     3 | yyy
(2 rows)
=> SELECT * FROM t1 INNER JOIN t2 USING (num);

num | name | value
------+--------+----------
    1 |      a | xxx
    3 |      c | yyy
(2 rows)
=> SELECT * FROM t1 NATURAL INNER JOIN t2;

num | name | value
------+---------+----------
    1 |      a | xxx
    3 |      c | yyy
(2 rows)
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
```

```
num | name | num | value
------+--------+-------+----------
   1 |      a |     1 | xxx
   2 |      b |       |
   3 |      c |     3 | yyy
(3 rows)
=> SELECT * FROM t1 LEFT JOIN t2 USING (num);

num | name | value
------+--------+----------
   1 |      a | xxx
   2 |      b |
   3 |      c | yyy
(3 rows)
=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;


num | name | num | value
------+--------+-------+----------
   1 |      a |     1 | xxx
   3 |      c |     3 | yyy
     |        |     5 | zzz
(3 rows)
=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;

num | name | num | value
------+--------+-------+----------
   1 |      a |     1 | xxx
   2 |      b |       |
   3 |      c |     3 | yyy
     |        |     5 | zzz
(4 rows)
```

Điều kiện liên kết được chỉ định với ON cũng có thể có các điều kiện mà không liên quan trực tiếp tới liên kết đó.

Điều này có thể chứng minh là hữu dụng đối với một số truy vấn nhưng cần phải được suy nghĩ cẩn thận. Ví dụ:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';

num | name | num | value
------+--------+-------+----------
   1 |      a |     1 | xxx
   2 |      b |       |
   3 |      c |       |
(3 rows)
```

Lưu ý rằng việc đặt ra hạn chế trong mệnh đề WHERE tạo ra một kết quả khác:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';

num | name | num | value
------+--------+-------+----------
   1 |      a |     1 | xxx
(1 row)
```

Điều này là vì một hạn chế được đặt trong mệnh đề ON được xử lý *trước* liên kết, trong khi một hạn chế được đặt trong mệnh đề WHERE được xử lý *sau* liên kết.

7.2.1.2. Tên hiệu của bảng và cột

Một tên tạm thời có thể được trao cho các bảng và các tham chiếu bảng phức tạp để được sử dụng cho các tham chiếu tới bảng dẫn xuất trong phần còn lại của truy vấn. Điều này được gọi là một tên hiệu (alias) của một bảng.

Để tạo tên hiệu của một bảng, hãy viết

```
FROM table_reference AS alias
```

hoặc

```
FROM table_reference alias
```

Từ khóa AS là nhiễu tùy chọn, alias có thể là bất kỳ mã định danh nào.

Ứng dụng điển hình của các tên hiệu bảng là để chỉ định các mã định danh ngắn cho các tên bảng dài để giữ cho các mệnh đề liên kết dễ đọc. Ví dụ:

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num;
```

Tên hiệu trở thành tên mới của tham chiếu bảng cho tới nay như truy vấn hiện hành được quan tâm - không được phép tham chiếu tới bảng bằng tên gốc ở bất kỳ đâu khác trong truy vấn đó. Vì thế, điều này là không hợp lệ:

```
SELECT * FROM my_table AS m WHERE my_table.a > 5; -- sai
```

Các tên hiệu của bảng chủ yếu là vì sự thuận tiện, nhưng là cần thiết để sử dụng chúng khi liên kết một bảng tới bản thân nó, như:

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

Hơn nữa, một tên hiệu được yêu cầu nếu tham chiếu bảng là một truy vấn con (xem Phần 7.2.1.3).

Các dấu ngoặc đơn sẽ được sử dụng để giải quyết những tù mù. Trong ví dụ sau, lệnh đầu tiên chỉ định tên hiệu b cho sự xuất hiện thứ 2 của my_table, nhưng lệnh thứ 2 chỉ định tên hiệu cho kết quả của liên kết:

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

Dạng khác tạo tên hiệu đưa ra các tên tạm thời cho các cột của bảng, cũng như bản thân bảng:

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

Nếu ít tên hiệu cột hơn được chỉ định so với bảng thực tế có các cột, thì các cột còn lại sẽ không được đổi tên. Cú pháp này đặc biệt hữu dụng cho các liên kết tự thân hoặc các truy vấn con.

Khi một tên hiệu được áp dụng cho đầu ra của một mệnh đề liên kết JOIN, thì tên hiệu ẩn đi (các) tên gốc trong JOIN. Ví dụ:

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

là SQL hợp lệ, nhưng:

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

là không hợp lệ; tên hiệu bảng a là không nhìn thấy được bên ngoài tên hiệu c.

7.2.1.3. Truy vấn con

Các truy vấn con (hoặc phụ) chỉ định một bảng dẫn xuất phải nằm trong các dấu ngoặc đơn và phải được chỉ định cho một tên hiệu bảng. (Xem Phần 7.2.1.2). Ví dụ:

```
FROM (SELECT * FROM table1) AS alias_name
```

Ví dụ này là tương đương với FROM table1 AS alias_name. Các trường hợp thú vị hơn, chúng không thể được giảm về một liên kết thường, làm nảy sinh truy vấn con có liên quan tới việc tạo nhóm hoặc tổng hợp.

Một truy vấn con cũng có thể là một danh sách các giá trị VALUES:

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
        AS names(first, last)
```

Một lần nữa, một tên hiệu bảng được yêu cầu. Việc chỉ định các tên hiệu cho các cột của danh sách VALUES là tùy chọn, nhưng là thực tiễn tốt. Để có thêm thông tin, xem Phần 7.7.

7.2.1.4. Hàm bảng

Các hàm bảng là các hàm tạo ra một tập hợp các hàng, được làm hoặc từ các dạng dữ liệu cơ bản (các dạng vô hướng) hoặc các dạng dữ liệu tổng hợp (các hàng của bảng). Chúng được sử dụng giống như một bảng, kiểu nhìn hoặc truy vấn con trong mệnh đề FROM của một truy vấn. Các cột được các hàm bảng trả về có thể được đưa vào trong các mệnh đề SELECT, JOIN, hoặc WHERE theo cách thức y hệt như một bảng, kiểu nhìn hoặc cột của truy vấn con.

Nếu một hàm bảng trả về một dạng dữ liệu cơ bản, thì tên cột kết quả duy nhất sẽ khớp với tên hàm. Nếu hàm trả về một dạng tổng hợp, thì các cột kết quả có các tên y hệt như các thuộc tính riêng rẽ của dạng đó.

Một hàm bảng có thể được gắn tên hiệu trong mệnh đề FROM, nhưng nó cũng có thể không có tên hiệu. Nếu một hàm được sử dụng trong mệnh đề FROM không có tên hiệu, thì tên hàm được sử dụng như là tên bảng kết quả. Một số ví dụ:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
        SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
SELECT * FROM getfoo(1) AS t1;
SELECT * FROM foo
        WHERE foosubid IN      (
                               SELECT foosubid
                               FROM getfoo(foo.fooid) z
                               WHERE z.fooid = foo.fooid
                               );
CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);
SELECT * FROM vw_getfoo;
```

Trong một số trường hợp, là hữu dụng để định nghĩa các hàm bảng mà có thể trả về các tập hợp các

cột khác nhau, phụ thuộc vào cách mà chúng được gọi. Để hỗ trợ điều này, hàm bảng có thể được khai báo như là việc trả về dạng giả (pseudotype) record. Khi một hàm như vậy được sử dụng trong một truy vấn, cấu trúc hàng được mong đợi phải được chỉ định trong bản thân truy vấn đó, sao cho hệ thống có thể biết cách để phân tích cú pháp và lên kế hoạch cho truy vấn. Hãy xem xét ví dụ này:

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
        AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

Hàm dblink thực thi một truy vấn ở xa (xem contrib/dblink). Nó được khai báo để trả về record vì nó có lẽ được sử dụng cho bất kỳ dạng truy vấn nào. Tập hợp các cột thực sự phải được chỉ định trong việc gọi truy vấn sao cho trình phân tích cú pháp biết được, ví dụ, * sẽ mở rộng cái gì.

7.2.2. Mệnh đề WHERE

Cú pháp của mệnh đề WHERE là

```
WHERE search_condition
```

trong đó search_condition là bất kỳ biểu thức giá trị nào (xem Phần 4.2) mà trả về một giá trị dạng boolean.

Sau khi xử lý mệnh đề FROM, mỗi hàng của bảng ảo dẫn xuất được kiểm tra đối với điều kiện tìm kiếm. Nếu kết quả của điều kiện là đúng, thì hàng đó được giữ trong bảng đầu ra, nếu không (như, nếu kết quả là sai hoặc null) thì sẽ bị bỏ. Điều kiện tìm kiếm thường tham chiếu ít nhất tới một cột của bảng được tạo ra trong mệnh đề FROM; Điều này không bị yêu cầu, nhưng nếu không thì mệnh đề WHERE sẽ khá là vô dụng.

> **Lưu ý**: Điều kiện liên kết của một liên kết vòng trong có thể được viết hoặc trong mệnh đề WHERE hoặc trong mệnh đề JOIN. Ví dụ, các biểu thức bảng sau là tương đương:
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> và:
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> hoặc thậm chí có thể:
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> Cái nào trong số này bạn sử dụng chỉ là vấn đề chọn kiểu. Cú pháp JOIN trong mệnh đề FROM có lẽ không khả chuyển được như với các hệ quản trị cơ sở dữ liệu SQL khác, thậm chí dù nó là theo tiêu chuẩn SQL. Đối với các liên kết bên ngoài thì không có sự lựa chọn: chúng phải được thực hiện theo mệnh đề FROM. Mệnh đề ON hoặc USING của liên kết vòng ngoài là không tương đương với điều kiện WHERE, vì nó cho kết quả là sự bổ sung các hàng (đối với các hàng đầu vào không khớp) cũng như sự loại bỏ các hàng trong kết quả cuối cùng.

Đây là một số ví dụ của các mệnh đề WHERE:

```
SELECT ... FROM fdt WHERE c1 > 5
SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)
```

```
SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)
SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100
SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

fdt là bảng dẫn xuất trong mệnh đề FROM. Các hàng mà không đáp ứng được điều kiện tìm kiếm của mệnh đề WHERE bị loại bỏ khỏi fdt. Lưu ý sử dụng các truy vấn con vô hướng như các biểu thức giá trị. Hệt như bất kỳ truy vấn nào khác, các truy vấn con có thể sử dụng các biểu thức bảng phức tạp. Cũng lưu ý cách mà fdt được tham chiếu trong các truy vấn con. Việc định phẩm chất c1 như fdt.c1 chỉ nhất thiết nếu c1 cũng là tên của một cột trong bảng đầu vào dẫn xuất của truy vấn con. Nhưng việc định phẩm chất tên cột bổ sung thêm sự rõ ràng minh bạch thậm chí khi nó là không cần thiết. Ví dụ này chỉ ra cách mà cột nêu phạm vi của một truy vấn vòng ngoài mở rộng trong các truy vấn vòng trong của nó.

7.2.3. Các mệnh đề GROUP BY và HAVING

Sau khi đi qua bộ lọc WHERE, bảng đầu vào dẫn xuất có thể tuân theo việc tạo nhóm, bằng việc sử dụng mệnh đề GROUP BY, và loại bỏ các hàng của nhóm bằng việc sử dụng mệnh đề HAVING.

```
SELECT select_list
        FROM ...
        [WHERE ...]
        GROUP BY grouping_column_reference [, grouping_column_reference]...
```

Mệnh đề GROUP BY được sử dụng để tạo nhóm cùng các hàng trong một bảng mà có cùng y hệt các giá trị trong tất cả các cột được liệt kê. Trật tự theo đó các cột được liệt kê không là vấn đề. Hiệu ứng là để kết hợp từng tập hợp các hàng có cùng các giá trị vào trong một hàng của nhóm mà đại diện cho tất cả các hàng trong nhóm đó. Điều này được thực hiện để loại bỏ sự dư thừa ở đầu ra và/hoặc tính toán các tổng mà áp dụng cho các nhóm đó.

Ví dụ:

```
=> SELECT * FROM test1;

x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;

x
---
a
b
c
(3 rows)
```

Trong truy vấn thứ 2, chúng ta có thể đã không viết SELECT * FROM test1 GROUP BY x, vì không có giá trị duy nhất cho cột y mà có thể có liên kết với từng nhóm. Các cột được nhóm có thể được tham chiếu trong danh sách chọn vì chúng có một giá trị duy nhất trong từng nhóm.

Nói chung, nếu một bảng được nhóm, thì các cột mà không được liệt kê trong GROUP BY không thể được tham chiếu ngoại trừ trong các biểu thức tổng hợp. Một ví dụ với các biểu thức tổng hợp là:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;

 x | sum
---+-----
 a | 4
 b | 5
 c | 2
(3 rows)
```

Ở đây tổng (sum) là một hàm tổng hợp mà tính một giá trị duy nhất đối với toàn bộ nhóm. Thông tin nhiều hơn về các hàm tổng hợp có sẵn có thể thấy trong Phần 9.18.

> **Mẹo**: Việc tạo nhóm mà không có các biểu thức tổng hợp tính toán có hiệu quả tập hợp các giá trị độc nhất trong một cột. Điều này cũng có thể đạt được bằng việc sử dụng mệnh đề DISTINCT (xem Phần 7.3.3).

Đây là một ví dụ khác: nó tính toán tổng tiền bán hàng cho từng sản phẩm (thay vì tổng tiền bán hàng của tất cả các sản phẩm):

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
        FROM products p LEFT JOIN sales s USING (product_id)
        GROUP BY product_id, p.name, p.price;
```

Trong ví dụ này, các cột product_id, p.name, và p.price phải nằm trong mệnh đề GROUP BY vì chúng được tham chiếu trong danh sách chọn của truy vấn. (Phụ thuộc vào cách mà bảng các sản phẩm được thiết lập, tên và giá có thể hoàn toàn phụ thuộc vào mã ID sản phẩm, nên việc tạo các nhóm bổ sung có thể về lý thuyết là không cần thiết, dù điều này không được triển khai). Cột s.units không phải nằm trong danh sách GROUP BY vì nó chỉ được sử dụng trong một biểu thức tổng hợp (sum(...)), nó thể hiện tiền bán hàng của một sản phẩm. Đối với từng sản phẩm, truy vấn đó trả về một hàng tổng tất cả tiền bán hàng của sản phẩm đó.

Trong SQL khắt khe, GROUP BY chỉ có thể nhóm theo các cột của bảng nguồn nhưng PostgreSQL mở rộng điều này để cũng cho phép GROUP BY đối với nhóm theo các cột trong danh sách chọn. Việc nhóm theo các biểu thức giá trị thay vì các tên cột đơn giản cũng được phép.

Nếu một bảng đã từng được nhóm bằng việc sử dụng GROUP BY, nhưng chỉ các nhóm nhất định có quan tâm, thì mệnh đề HAVING có thể được sử dụng, rất giống một mệnh đề WHERE, để loại bỏ các nhóm khỏi kết quả đó. Cú pháp là:

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

Các biểu thức trong mệnh đề HAVING có thể tham chiếu tới cả các biểu thức được nhóm và các biểu thức không được nhóm (chúng nhất thiết liên quan tới một hàm tổng hợp). Ví dụ:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a | 4
 b | 5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
```

```
 x | sum
---+-----
 a | 4
 b | 5
(2 rows)
```

Một lần nữa, một ví dụ thực tiễn hơn:

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
        FROM products p LEFT JOIN sales s USING (product_id)
        WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
        GROUP BY product_id, p.name, p.price, p.cost
        HAVING sum(p.price * s.units) > 5000;
```

Trong ví dụ ở trên, mệnh đề WHERE đang chọn các hàng theo một cột mà không được nhóm (biểu thức chỉ đúng cho các bán hàng trong 4 tuần cuối), trong khi mệnh đề HAVING hạn chế đầu ra đối với các nhóm với tổng lượng bán hàng hơn 5.000. Lưu ý rằng các biểu thức tổng hợp không nhất thiết cần phải là y hệt trong tất cả các phần của truy vấn.

Nếu một truy vấn gồm các lời gọi hàm tổng hợp, nhưng không có mệnh đề GROUP BY, thì việc nhóm vẫn xảy ra; kết quả là một hàng nhóm duy nhất (hoặc có lẽ không hàng nào cả, nếu hàng duy nhất sau đó bị HAVING loại bỏ). Y hệt là đúng nếu nó có mệnh đề HAVING, thậm chí không có bất kỳ lời gọi hàm tổng hợp nào hoặc mệnh đề GROUP BY nào.

7.2.4. Xử lý hàm cửa sổ

Nếu truy vấn có bất kỳ hàm cửa sổ nào (xem Phần 3.5, Phần 9.19 và Phần 4.2.8), thì các hàm đó được đánh giá sau bất kỳ việc tạo nhóm, tổng hợp và lọc HAVING nào được thực hiện. Đó là, nếu truy vấn sử dụng bất kỳ tổng hợp GROUP BY hoặc HAVING nào thì các hàng được các hàm cửa sổ xem là các hàng của nhóm thay vì các hàng của bảng gốc từ FROM /WHERE.

Khi nhiều hàm cửa sổ được sử dụng, tất cả các hàm cửa sổ, về mặt cú pháp tương đương với các mệnh đề PARTITION BY và ORDER BY trong các định nghĩa cửa sổ được đảm bảo sẽ được đánh giá theo một sự truyền dữ liệu duy nhất. Vì thế chúng sẽ thấy cùng y hệt việc sắp xếp trật tự, thậm chí nếu ORDER BY không xác định một cách duy nhất một trật tự. Tuy nhiên, không đảm bảo nào được thực hiện đối với sự đánh giá các hàm có các đặc tả khác nhau về PARTITION BY hoặc ORDER BY. (Trong các trường hợp như vậy một bước sắp xếp điển hình được yêu cầu giữa các việc truyền các đánh giá hàm cửa sổ, và sự sắp xếp không được đảm bảo để giữ lại trật tự các hàng mà ORDER BY của nó coi như là tương đương).

Hiện hành, các hàm cửa sổ luôn yêu cầu các dữ liệu trước khi được sắp xếp, và vì thế đầu ra của truy vấn sẽ có trật tự theo cách này hay cách khác của các mệnh đề PARTITION BY /ORDER BY đối với các hàm cửa sổ. Tuy nhiên, không được khuyến cáo để dựa vào điều này. Hãy sử dụng một mệnh đề ORDER BY mức đỉnh rõ ràng nếu bạn muốn chắc chắn các kết quả được sắp xếp theo một cách thức đặc biệt.

7.3. Danh sách chọn

Như được chỉ ra trong phần trước, biểu thức bảng trong lệnh SELECT tạo ra một bảng ảo trung gian

bằng việc có khả năng kết hợp các bảng, các kiểu nhìn, việc loại bỏ các hàng, việc tạo nhóm, …

Bảng này cuối cùng được truyền tới cho việc xử lý bằng danh sách chọn. Danh sách chọn xác định các cột nào của bảng trung gian thực sự sẽ là đầu ra.

7.3.1. Các khoản của danh sách chọn

Dạng đơn giản nhất của danh sách chọn là * mà nó đưa ra tất cả các cột mà biểu thức bảng tạo ra. Nếu không, một danh sách chọn là một danh sách các biểu thức giá trị cách nhau bằng dấu phẩy (như được xác định trong Phần 4.2). Ví dụ, nó có thể là một danh sách các tên cột:

```
SELECT a, b, c FROM ...
```

Các tên cột a, b, và c hoặc là các tên thực của các cột của các bảng được tham chiếu trong mệnh đề FROM, hoặc là các tên hiệu được đưa ra cho chúng như được giải thích trong Phần 7.2.1.2. Không gian tên sẵn sàng trong danh sách chọn là y hệt như trong mệnh đề WHERE, trừ phi việc tạo nhóm được sử dụng, trong trường hợp đó nó là y hệt như mệnh đề HAVING.

Nếu hơn một bảng có một cột với tên y hệt, thì tên bảng cũng phải được đưa ra, như trong:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

Khi làm việc với nhiều bảng, cũng có thể hữu dụng để yêu cầu tất cả các cột của một bảng đặc biệt:

```
SELECT tbl1.*, tbl2.a FROM ..
```

(Xem thêm Phần 7.2.2).

Nếu một biểu thức giá trị tùy chọn được sử dụng trong danh sách chọn, theo nguyên tắc, nó bổ sung thêm một cột ảo mới vào bảng được trả về đó. Biểu thức giá trị được đánh giá mỗi lần cho từng hàng kết quả, với các giá trị hàng được thay thế cho bất kỳ tham chiếu cột nào. Nhưng biểu thức đó trong danh sách chọn không phải tham chiếu tới bất kỳ cột nào trong biểu thức bảng của mệnh đề FROM; chúng có thể là các biểu thức tính số học các hằng số, ví dụ thế.

7.3.2. Nhãn cột

Các khoản đầu vào trong danh sách chọn có thể là các tên được chỉ định cho việc xử lý sau, như để sử dụng trong một mệnh đề ORDER BY hoặc để hiển thị đối với ứng dụng máy trạm. Ví dụ:

```
SELECT a AS value, b + c AS sum FROM ...
```

Nếu không có tên cột đầu ra nào được chỉ định bằng việc sử dụng AS, thì hệ thống chỉ định một tên cột mặc định. Đối với các tham chiếu cột đơn giản, đây là tên của cột được tham chiếu. Đối với các lời gọi hàm, đây là tên của hàm. Đối với các biểu thức phức tạp, hệ thống sẽ sinh ra tên chung.

Từ khóa AS là tùy chọn, nhưng chỉ nếu tên cột mới không khớp với bất kỳ từ khóa PostgreSQL nào (xem Phụ lục C). Để tránh một sự trùng khớp ngẫu nhiên với một từ khóa, bạn có thể đưa vào trong các dấu ngoặc kép tên cột đó. Ví dụ, VALUE là một từ khóa, nên điều này không làm việc:

```
SELECT a value, b + c AS sum FROM ...
```

nhưng điều này thì làm việc:

```
SELECT a "value", b + c AS sum FROM ...
```

Để bảo vệ chống lại việc có thêm các từ khóa có khả năng trong tương lai, được khuyến cáo rằng bạn luôn hoặc viết AS hoặc cho vào trong ngoặc kép tên cột đầu ra.

> **Lưu ý**: Việc đặt tên các cột đầu ra ở đây là khác so với được làm trong mệnh đề FROM (xem Phần 7.2.1.2). Có khả năng để đổi tên cột y hệt 2 lần, nhưng tên được chỉ định trong danh sách chọn là tên mà sẽ được chuyển đi.

7.3.3. Khác biệt - DISTINCT

Sau khi danh sách chọn được xử lý, bảng kết quả có thể tùy ý tuân theo sự loại trừ các hàng đúp bản. Từ khóa DISTINCT được viết trực tiếp sau SELECT để chỉ định điều này:

```
SELECT DISTINCT select_list ...
```

(Thay vì DISTINCT, từ khóa ALL có thể được sử dụng để chỉ định hành vi mặc định của việc giữ lại tất cả các hàng).

Rõ ràng, 2 hàng được xem là phân biệt rõ ràng nếu chúng khác nhau ít nhất 1 giá trị cột. Các giá trị null được coi là bằng nhau trong so sánh này.

Như một sự lựa chọn, một biểu thức tùy ý có thể xác định các hàng nào sẽ được coi là phân biệt:

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

Biểu thức ở đây là một biểu thức giá trị tùy chọn được đánh giá cho tất cả các hàng. Một tập hợp các hàng theo đó tất cả các biểu thức là bằng nhau được xem là đúp bản, và chỉ hàng đầu tiên của tập hợp được sắp xếp trong các cột đủ để đảm bảo việc sắp xếp trật tự độc nhất hoặc các hàng đi đến được sự lọc DISTINCT. (Việc xử lý DISTINCT ON xảy ra sau việc sắp xếp ORDER BY).

Mệnh đề DISTINCT ON không phải là một phần của tiêu chuẩn SQL và đôi khi được xem là kiểu tồi tệ vì bản chất tự nhiên trung gian tiềm tàng các kết quả của nó. Với việc sử dụng thận trọng mệnh đề GROUP BY và các truy vấn phụ trong FROM, cấu trúc này có thể tránh được, nhưng nó thường là lựa chọn thuận tiện nhất.

7.4. Kết hợp các truy vấn

Các kết quả của 2 truy vấn có thể được kết hợp bằng việc sử dụng sự kết hợp tập hợp các hoạt động, sự giao nhau và sự khác nhau.

Cú pháp là

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

query1 và *query2* là các truy vấn có thể sử dụng bất kỳ tính năng nào được thảo luận cho tới thời điểm này. Tập hợp các hoạt động cũng có thể được lồng nhau và xâu thành chuỗi, ví dụ

```
query1 UNION query2 UNION query3
```

nó được thực thi như:

```
(query1 UNION query2) UNION query3
```

UNION nối thêm một cách hiệu quả kết quả của query2 cho kết quả của query1 (dù không có sự đảm bảo nào rằng điều này là trật tự theo đó các hàng thực sự được trả về). Hơn nữa, nó loại bỏ các hàng đúp bản khỏi kết quả của nó, theo cách y hệt như DISTINCT, trừ phi UNION ALL được sử dụng.

INTERSECT trả về tất cả các hàng nằm ở trong kết quả của query1 và trong kết quả của query2. Đúp bản các hàng bị loại trừ, trừ phi INTERSECT ALL được sử dụng.

EXCEPT trả về tất cả các hàng nằm trong kết quả của query1 nhưng không nằm trong kết quả của query2. (Điều này đôi khi được gọi là *sự khác biệt* giữa 2 truy vấn). Một lần nữa, các đúp bản bị loại trừ, trừ phi EXCEPT ALL được sử dụng.

Để tính toán sự kết hợp, sự giao nhau hoặc sự khác nhau của 2 truy vấn, 2 truy vấn đó phải là "tương thích kết hợp", có nghĩa là chúng trả về số cột y hệt và các cột tương ứng có các dạng dữ liệu tương thích, như được mô tả trong Phần 10.5.

7.5. Sắp xếp hàng

Sau khi một truy vấn đã sản sinh ra một bảng đầu ra (sau khi danh sách chọn được xử lý), nó có thể được sắp xếp tùy ý. Nếu việc sắp xếp không được chọn, thì các hàng sẽ được trả về theo một trật tự không được chỉ định. Trật tự thực sự trong trường hợp đó sẽ phụ thuộc vào các dạng quét và liên kết và trật tự trên đĩa, nhưng nó phải không được dựa vào. Một trật tự sắp xếp đầu ra đặc biệt chỉ có thể được đảm bảo nếu bước sắp xếp được chọn rõ ràng.

Mệnh đề ORDER BY chỉ định trật tự sắp xếp:

```
SELECT select_list
        FROM table_expression
        ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
                [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

(Các) biểu thức sắp xếp có thể là bất kỳ biểu thức nào có thể là hợp lệ trong danh sách chọn của truy vấn. Một ví dụ là:

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

Khi nhiều hơn một biểu thức được chỉ định, các giá trị sau đó sẽ được sử dụng để sắp xếp các hàng bằng nhau theo các giá trị trước đó. Mỗi biểu thức có thể có ở đằng sau một từ khóa tùy chọn ASC hoặc DESC để thiết lập hướng sắp xếp tăng hoặc giảm. Trật tự ASC là mặc định. Trật tự tăng đặt các giá trị nhỏ hơn ở trước, nơi mà "nhỏ hơn" được xác định theo toán tử nhỏ hơn (<). Tương tự, trật tự giảm được xác định với toán tử lớn hơn (>)[1].

Các lựa chọn NULLS FIRST và NULLS LAST có thể được sử dụng để xác định liệu các null có xuất hiện

1 Thực sự, PostgreSQL sử dụng lớp toán tử mặc định B-tree cho dạng dữ liệu của biểu thức để xác định trật tự sắp xếp ASC và DESC. Theo qui ước, các dạng dữ liệu sẽ được thiết lập sao cho các toán tử < and > tương ứng với trật tự sắp xếp này, nhưng một nhà thiết kế dạng dữ liệu do người sử dụng định nghĩa có thể chọn làm thứ gì đó khác.

trước hay sau các giá trị không null trong trật tự sắp xếp hay không. Mặc định, các giá trị null sắp xếp dường như lớn hơn so với bất kỳ giá trị không null (non-null) nào; đó là, NULLS FIRST là mặc định cho trật tự DESC, nếu khác thì là NULLS LAST.

Lưu ý rằng các lựa chọn sắp xếp trật tự được xem là độc lập đối với từng cột sắp xếp. Ví dụ ORDER BY x, y DESC có nghĩa là ORDER BY x ASC, y DESC, nó không là y hệt như ORDER BY x DESC, y DESC.

Một sort_expression cũng có thể là nhãn cột hoặc số của một cột đầu ra, như trong:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

cả 2 cách đó đều sắp xếp theo cột đầu ra đầu tiên. Lưu ý rằng tên một cột đầu ra phải đứng một mình, đó là, nó không thể được sử dụng trong một biểu thức - ví dụ, điều này là *không* đúng:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c; -- wrong
```

Hạn chế này được thực hiện để làm giảm sự tù mù. Vẫn có sự tù mù nếu một khoản ORDER BY là một tên đơn giản có thể khớp hoặc một tên cột đầu ra hoặc một cột từ biểu thức bảng. Cột đầu ra được sử dụng trong các trường hợp như vậy. Điều này chỉ có thể gây ra sự lúng túng nếu bạn sử dụng AS để đổi tên một cột đầu ra để khớp với một số tên cột của bảng khác.

ORDER BY có thể được áp dụng cho kết quả của một sự kết hợp UNION, INTERSECT, và EXCEPT, nhưng trong trường hợp này nó chỉ được phép sắp xếp theo các tên hoặc các số cột đầu ra, không theo các biểu thức.

7.6. LIMIT và OFFSET

LIMIT và OFFSET cho phép truy xuất chỉ một phần các hàng được phần còn lại của truy vấn tạo ra:

```
SELECT select_list
        FROM table_expression
        [ ORDER BY ... ]
        [ LIMIT { number | ALL } ] [ OFFSET number ]
```

Nếu một sự tính toán hạn chế được đưa ra, không nhiều hơn việc nhiều hàng sẽ được trả về (nhưng có thể là ít hơn, nếu bản thân truy vấn có ít hàng hơn). LIMIT ALL là y hệt như việc bỏ qua mệnh đề LIMIT.

OFFSET nói để bỏ qua nhiều hàng đó trước khi bắt đầu trả về các hàng. OFFSET 0 là y hệt như việc bỏ qua mệnh đề OFFSET, và LIMIT NULL là y hệt như việc bỏ qua mệnh đề LIMIT. Nếu cả OFFSET và LIMIT đều xuất hiện, thì các hàng OFFSET được bỏ qua trước khi bắt đầu tính các hàng LIMIT sẽ được trả về.

Khi sử dụng LIMIT, điều quan trọng phải sử dụng một mệnh đề ORDER BY mà ràng buộc các hàng kết quả vào một trật tự độc nhất. Nếu không thì bạn sẽ có một tập con các hàng của truy vấn đó một cách không thể đoán trước được. Bạn có thể được yêu cầu từ 10-20 hàng, nhưng 10-20 theo trật tự sắp xếp nào? Trật tự sắp xếp là không rõ, trừ phi bạn chỉ định ORDER BY.

Trình tối ưu hóa truy vấn tính tới LIMIT khi tạo các kế hoạch truy vấn, nên bạn rất có khả năng có các kế hoạch khác nhau (có các trật tự hàng khác nhau) phụ thuộc vào những gì bạn trao cho LIMIT và OFFSET. Vì thế, việc sử dụng các giá trị khác nhau của LIMIT/OFFSET để chọn các tập con khác nhau

kết quả của một truy vấn sẽ đưa ra các kết quả không nhất quán, trừ phi bạn ép tuân thủ một trật tự sắp xếp các kết quả có khả năng đoán trước được với ORDER BY. Đây không phải là một lỗi; nó là một hệ quả kế thừa của thực tế rằng SQL không hứa hẹn phân phối các kết quả của một truy vấn theo bất kỳ trật tự đặc biệt nào, trừ phi ORDER BY được sử dụng để ràng buộc trật tự đó.

Các hàng bị mệnh đề OFFSET bỏ qua vẫn phải được tính toán bên trong máy chủ; vì thế một OFFSET lớn có lẽ là không hiệu quả.

7.7. Danh sách giá trị

VALUES đưa ra một cách thức để tạo một "bảng các hằng số" mà có thể được sử dụng trong một truy vấn mà thực sự không phải tạo ra và đưa dữ liệu vào một bảng trên đĩa. Cú pháp là:

```
VALUES ( expression [, ...] ) [, ...]
```

Từng danh sách các biểu thức trong các dấu ngoặc đơn sinh ra một hàng trong bảng. Các danh sách tất cả phải có cùng y hệt số các phần tử (như, số cột trong bảng), và các khoản đầu vào tương ứng trong từng danh sách phải có các dạng dữ liệu tương thích. Dạng dữ liệu thực sự được chỉ định cho từng cột kết quả được xác định bằng việc sử dụng các qui tắc y hệt như đối với UNION (xem Phần 10.5). Như một ví dụ:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

sẽ trả về một bảng có 2 cột và 3 hàng. Nó tương đương một cách có hiệu lực với:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

Mặc định, PostgreSQL chỉ định các tên column1, column2, … cho các cột của bảng VALUES. Các tên cột không được tiêu chuẩn SQL chỉ định và các hệ cơ sở dữ liệu khác nhau làm khác nhau, nên thường là tốt hơn để ghi đè các tên mặc định với danh sách các tên hiệu của một bảng.

Về cú pháp, theo sau VALUES có các danh sách biểu thức được đối xử tương đương với:

```
SELECT select_list FROM table_expression
```

và có thể xuất hiện ở bất cứ đâu mà một SELECT có thể. Ví dụ, bạn có thể sử dụng nó như một phần của một UNION, hoặc gắn một sort_specification (ORDER BY, LIMIT, và/hoặc OFFSET) cho nó. VALUES được sử dụng phổ biến nhất như là nguồn dữ liệu trong lệnh INSERT, và tiếp sau phổ biến nhất như một truy vấn con.

Để có thêm thông tin, xem VALUES.

7.8. Truy vấn với WITH (Biểu thức bảng chung)

WITH đưa ra một cách thức để viết các truy vấn con để sử dụng trong một truy vấn SELECT lớn hơn. Các truy vấn con, chúng thường được tham chiếu tới như là các Biểu thức Bảng Chung - CTE (Common Table Expressions), có thể được xem như việc định nghĩa các bảng tạm thời đang tồn tại

chỉ vì truy vấn này. Người ta sử dụng tính năng này là để chia các truy vấn phức tạp thành các phần đơn giản hơn. Một ví dụ là:

```
WITH regional_sales AS (
        SELECT region, SUM(amount) AS total_sales
        FROM orders
        GROUP BY region
     ), top_regions AS (
        SELECT region
        FROM regional_sales
        WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
     )
SELECT region,
        product,
        SUM(quantity) AS product_units,
        SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

nó hiển thị tổng bán hàng theo từng sản phẩm chỉ trong các khu vực bán hàng hàng đầu. Ví dụ này có thể được viết mà không có WITH, nhưng chúng ta có thể đã cần tới 2 mức lồng nhau các lệnh SELECT con. Dễ dàng hơn để tuân theo cách này.

Trình sửa đổi tùy chọn RECURSIVE làm thay đổi WITH từ chỉ sự thuận tiện về cú pháp trong một tính năng hoàn thành các điều mà nếu khác đi thì không có khả năng trong SQL tiêu chuẩn. Sử dụng RECURSIVE, một truy vấn WITH có thể tham chiếu tới đầu ra của riêng nó. Một ví dụ rất đơn giản là truy vấn này để tính tổng các số nguyên từ 1 tới 100:

```
WITH RECURSIVE t(n) AS (
        VALUES (1)
    UNION ALL
        SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Dạng chung của một truy vấn đệ qui WITH luôn là một khoản không đệ qui, rồi UNION (hoặc UNION ALL), rồi một khoản *đệ qui*, nơi mà chỉ khoản đệ qui có thể có một tham chiếu tới đầu ra của riêng truy vấn đó. Một truy vấn như vậy được thực thi như sau:

Đánh giá truy vấn đệ qui

1. Hãy đánh giá khoản không đệ qui. Đối với UNION (nhưng không với UNION ALL), hãy bỏ các hàng đúp bản. Hãy đưa vào tất cả các hàng còn lại trong kết quả của truy vấn đệ qui, và cũng đặt chúng vào một *bảng làm việc tạm thời*.
2. Miễn là bảng làm việc không rỗng, hãy lặp lại các bước:
   a) Đánh giá khoản đệ qui, thay thế các nội dung của bảng làm việc đối với tự tham chiếu đệ qui. Đối với UNION (nhưng không với UNION ALL), hãy bỏ các hàng đúp bản và các hàng đúp bất kỳ hàng kết quả nào trước đó. Đưa vào tất cả các hàng còn lại vào trong kết quả của truy vấn đệ qui, và cũng đặt chúng vào một *bảng trung gian tạm*.
   b) Thay thế các nội dung của bảng làm việc bằng các nội dung của bảng trung gian, rồi làm

rỗng bảng trung gian.

Lưu ý: Nói một cách nghiêm ngặt, qui trình này là lặp đi lặp lại không phải sự đệ qui, mà RECURSIVE là thuật ngữ được ủy ban các tiêu chuẩn SQL lựa chọn.

Trong ví dụ ở trên, bảng làm việc chỉ có 1 hàng duy nhất trong từng bước, và nó lấy các giá trị từ 1 đến 100 theo các bước kế tiếp. Trong bước thứ 100, không có đầu ra nào vì mệnh đề WHERE, và vì thế truy vấn kết thúc.

Các truy vấn đệ qui thường được sử dụng để làm việc với các dữ liệu kế thừa hoặc có cấu trúc hình cây. Một ví dụ hữu dụng là truy vấn này thấy tất cả các phần con trực tiếp và gián tiếp của một sản phẩm, biết rằng chỉ một bảng chỉ ra được những chèn thêm ngay tức thì:

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
        SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
    UNION ALL
        SELECT p.sub_part, p.part, p.quantity
        FROM included_parts pr, parts p
        WHERE p.part = pr.sub_part
    )
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

Khi làm việc với các truy vấn đệ qui, điều quan trọng phải chắc chắn rằng phần đệ qui của truy vấn cuối cùng sẽ không trả về bộ số liệu, nếu không thì truy vấn sẽ lặp vô tận. Đôi khi, việc sử dụng UNION thay cho UNION ALL có thể hoàn tất được điều này bằng việc bỏ các hàng mà đúp bản các hàng đầu ra trước đó. Tuy nhiên, thường thì một chu kỳ không liên quan tới các hàng đầu ra mà hoàn toàn đúp bản: có thể là cần thiết để kiểm tra chỉ một hoặc một ít các trường để thấy liệu điểm y hệt có đạt được trước hay không. Phương pháp tiêu chuẩn cho việc điều khiển các tình huống như vậy là để tính toán bất kỳ mảng giá trị nào đã được thăm viếng rồi. Ví dụ, xem xét truy vấn sau đây tìm kiếm đồ họa một bảng bằng việc sử dụng một trường liên kết:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
        SELECT g.id, g.link, g.data, 1
        FROM graph g
    UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1
        FROM graph g, search_graph sg
        WHERE g.id = sg.link
    )
SELECT * FROM search_graph;
```

Truy vấn này sẽ lặp nếu các mối quan hệ liên kết có các chu kỳ. Vì chúng ta yêu cầu một đầu ra "sâu", nên chỉ việc thay đổi UNION ALL thành UNION cũng có thể không loại trừ được việc lặp. Thay vào đó chúng ta cần nhận thức được liệu chúng ta đã tới được hàng y hệt một lần nữa hay chưa trong khi tuân theo một đường các liên kết đặc biệt. Chúng ta thêm 2 cột path và cycle vào truy vấn lặp - dễ hỏng:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
```

```
    SELECT g.id, g.link, g.data, 1,
        ARRAY[g.id],
        false
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
        path || g.id,
        g.id = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Ngoài việc ngăn các chu kỳ, giá trị mảng thường hữu dụng theo quyền của riêng nó như việc đại diện cho "đường" ("path") được lấy để với tới được bất kỳ hàng đặc biệt nào.

Trong trường hợp chung nơi mà nhiều hơn một trường cần phải được kiểm tra để nhận thức được một chu kỳ, hãy sử dụng một mảng các hàng. Ví dụ, nếu chúng ta cần so sánh các trường f1 và f2:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
        ARRAY[ROW(g.f1, g.f2)],
        false
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
        path || ROW(g.f1, g.f2),
        ROW(g.f1, g.f2) = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

> **Mẹo**: Bỏ qua cú pháp ROW() trong trường hợp chung nơi mà chỉ một trường cần phải được kiểm tra để nhận ra được một chu kỳ. Điều này cho phép một mảng đơn giản hơn là một mảng dạng tổng hợp sẽ được sử dụng, giành được sự hiệu quả.

> **Mẹo**: Thuật toán đánh giá truy vấn đệ qui tạo ra đầu ra của nó theo trật tự tìm kiếm theo độ rộng trước. Bạn có thể hiển thị các kết quả theo trật tự tìm kiếm độ sâu trước bằng cách làm cho truy vấn vòng ngoài ORDER BY thành một cột "đường" ("path") được xây theo cách này.

Một mẹo hữu dụng cho việc kiểm thử các truy vấn khi bạn không chắc chắn nếu chúng có thể lặp là hãy đặt một LIMIT vào truy vấn cha. Ví dụ, truy vấn này có thể lắp bất tận mà không có LIMIT:

```
WITH RECURSIVE t(n) AS (
        SELECT 1
    UNION ALL
        SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

Điều này làm việc vì sự triển khai PostgreSQL chỉ đánh giá càng nhiều hàng của một truy vấn WITH như thực sự được truy vấn cha lấy. Sử dụng mẹo này trong sản xuất không được khuyến cáo, vì các hệ thống khác có thể làm việc khác nhau. Hơn nữa, nó thực sự không làm việc nếu bạn để truy vấn vòng ngoài sắp xếp các kết quả truy vấn đệ qui hoặc liên kết chúng với một số bảng khác.

Một đặc tính hữu dụng của các truy vấn WITH là chúng chỉ được đánh giá một lần cho từng sự thực

thi của truy vấn cha, thậm chí nếu chúng được tham chiếu tới nhiều hơn một lần đối với truy vấn cha hoặc các truy vấn anh em WITH. Vì thế, những tính toán đắt giá cần thiết ở nhiều nơi có thể được đặt trong một truy vấn WITH để tránh công việc dư thừa. Ứng dụng có khả năng khác là để ngăn ngừa nhiều đánh giá các hàm không mong muốn với các hiệu ứng phụ. Tuy nhiên, mặt kia của đồng xu này là trình tối ưu hóa ít có khả năng hơn để đẩy những hạn chế từ truy vấn cha xuống vào trong một truy vấn WITH so với một truy vấn con (phụ) thông thường. Với truy vấn WITH thường sẽ được đánh giá như được nêu, mà không ép các hàng mà truy vấn cha có thể bỏ sau đó. (Mà, như được nhắc tới ở trên, sự đánh giá có thể dừng sớm nếu (các) tham chiếu cho truy vấn chỉ đòi hỏi một số lượng hạn chế các hàng).

Chương 8. Dạng dữ liệu

PostgreSQL có một tập hợp giàu có các dạng dữ liệu bẩm sinh sẵn có cho người sử dụng. Người sử dụng có thể thêm các dạng mới cho PostgreSQL bằng việc sử dụng lệnh tạo dạng CREATE TYPE.

Bảng 8-1 chỉ ra tất cả các dạng dữ liệu mục đích chung được xây dựng sẵn. Hầu hết các tên lựa chọn thay thế được liệt kê trong cột "Aliases" ("Các tên hiệu") là các tên được PostgreSQL sử dụng nội bộ vì các lý do lịch sử. Hơn nữa, một số dạng không được tán thành hoặc được sử dụng nội bộ là sẵn sàng, nhưng không được liệt kê ở đây.

Bảng 8-1. Các dạng dữ liệu

Tên	Tên hiệu	Mô tả
bigint	int8	số nguyên 8 byte được ký
bigserial	serial8	số nguyên 8 byte tự động tăng
bit [(n)]		chuỗi bit độ dài cố định
bit varying [(n)]	varbit	chuỗi bit độ dài biến đổi
boolean	bool	Boolean logic (đúng/sai - true/false)
box	-	hộp chữ nhật trên một mặt phẳng
bytea		dữ liệu nhị phân ("mảng theo byte")
character varying [(n)]	varchar [(n)]	chuỗi ký tự độ dài biến đổi
character [(n)]	char [(n)]	chuỗi ký tự độ dài cố định
cidr		địa chỉ mạng IPv4 hoặc IPv6
circle		mạch trên mặt phẳng
date		ngày tháng theo lịch (năm, tháng, ngày)
double precision	float8	số các chấm động chính xác đúp (8 byte)
inet		địa chỉ máy chủ theo IPv4 hoặc IPv6
integer	int, int4	số nguyên 4 byte được ký
interval [fields] [(p)]		khoảng thời gian
line		đường vô cực trên một mặt phẳng
lseg		đoạn thẳng trên mặt phẳng
macaddr		địa chỉ MAC (Kiểm soát Truy cập Phương tiện - Media Access Control)
money		lượng hiện hành
numeric [(p, s)]	decimal [(p, s)]	số chính xác của độ chính xác có khả năng chọn được
path		đường địa lý trên mặt phẳng
point		điểm địa lý trên mặt phẳng
polygon		đường địa lý khép kín trên mặt phẳng
real	float4	điểm chấm động chính xác duy nhất (4 byte)
smallint	int2	số nguyên 2 byte được ký
serial	serial4	số nguyên 4 byte tự động tăng

Tên	Tên hiệu	Mô tả
text		chuỗi ký tự độ dài biến đổi
time [(p)] [without time zone]		thời gian trong ngày (không có vùng thời gian)
time [(p)] with time zone	timetz	thời gian trong ngày, có vùng thời gian
timestamp [(p)] [without time zone]		ngày tháng và thời gian (không có vùng thời gian)
timestamp [(p)] with time zone	timestamptz	ngày tháng và thời gian, có vùng thời gian
tsquery		truy vấn tìm kiếm văn bản
tsvector		tài liệu tìm kiếm văn bản
txid_snapshot		hình chụp ID giao dịch mức người sử dụng
uuid		mã định danh độc nhất vạn năng
xml		dữ liệu XML

Tính tương thích: Các dạng sau đây (hoặc sự đánh vần của chúng) được SQL chỉ định: bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (có hoặc không có vùng thời gian), timestamp (có hoặc không có vùng thời gian), xml.

Mỗi dạng dữ liệu có một trình bày bên ngoài được các hàm đầu vào và đầu ra của nó xác định. Nhiều trong số các dạng được xây dựng sẵn có các định dạng bên ngoài rõ ràng. Tuy nhiên, vài dạng cũng là độc nhất đối với PostgreSQL, như các đường địa lý, hoặc có vài định dạng có khả năng, như các dạng ngày tháng và thời gian. Một số hàm đầu vào và đầu ra không nghịch đảo được, như, kết quả của hàm đầu ra có thể mất độ chính xác khi được so sánh với đầu vào gốc ban đầu.

8.1. Dạng số

Các dạng số bao gồm các số nguyên 2, 4 và 8 byte, các số dấu chấm động 4 và 8 byte, và các số thập phân độ chính xác có khả năng chọn được. Bảng 8-2 liệt kê các dạng có sẵn.

Bảng 8-2. Các dạng số

Tên	Kích cỡ lưu trữ	Mô tả	Dải
smallint	2 bytes	số nguyên dãy nhỏ	-32768 tới +32767
integer	4 bytes	lựa chọn điển hình cho số nguyên	-2147483648 tới +2147483647
bigint	8 bytes	số nguyên dãy lớn	- 9223372036854775808 tới 9223372036854775807
decimal	variable	chính xác, độ chính xác do người sử dụng chỉ định	không giới hạn
numeric	variable	chính xác, độ chính xác do người sử dụng chỉ định	không giới hạn

Tên	Kích cỡ lưu trữ	Mô tả	Dải
real	4 bytes	không chính xác, độ chính xác biến đổi	độ chính xác 6 chữ số thập phân
double precision	8 bytes	không chính xác, độ chính xác biến đổi	độ chính xác 15 chữ số thập phân
serial	4 bytes	số nguyên tự động tăng	1 tới 2147483647
bigserial	8 bytes	số nguyên tự động tăng lớn	1 tới 9223372036854775807

Cú pháp của các hàng đối với các dạng số được mô tả trong Phần 4.1.2. Các dạng số có một tập hợp đầy đủ các toán tử và các hàm đại số tương ứng. Tham chiếu tới Chương 9 để có thêm thông tin.

Các phần sau đây mô tả các dạng đó một cách chi tiết.

8.1.1. Dạng số nguyên

Các dạng và lưu trữ toàn bộ các số, đó là, các số không có các thành phần thập phân, của các dãy khác nhau. Những cố gắng để lưu trữ các giá trị bên ngoài của dãy được phép sẽ gây ra một lỗi.

Dạng integer là lựa chọn chung, khi nó đưa ra sự cân bằng tốt nhất giữa dãy, kích cỡ lưu trữ và hiệu năng. Dạng smallint thường chỉ được sử dụng nếu không gian đĩa khan hiếm. Dạng bigint chỉ nên được sử dụng nếu dãy integer là không đủ, vì cái sau là nhanh hơn chắc chắn.

Trên các hệ điều hành rất tối thiểu thì dạng bigint có thể không chạy tốt được, vì nó dựa vào sự hỗ trợ của trình biên dịch cho các số nguyên 8 byte. Trên các máy như vậy, bigint hành động y hệt như integer, nhưng vẫn lấy 8 byte lưu trữ. (Chúng tôi không biết về bất kỳ nền tảng hiện đại nào có vấn đề này).

SQL chỉ chỉ định các dạng số nguyên integer (hoặc int), smallint và bigint. Các tên dạng int2, int4, và int8 là các mở rộng, chúng cũng được một số hệ cơ sở dữ liệu SQL sử dụng.

8.1.2. Số chính xác tùy ý

Dạng numeric có thể lưu trữ các số tới 1.000 chữ số chính xác và thực hiện các tính toán một cách chính xác. Đặc biệt được khuyến cáo cho việc lưu trữ các lượng tiền và các lượng khác nơi mà tính chính xác được yêu cầu. Tuy nhiên, tính toán số học trên các giá trị numeric là rất chậm so với các dạng số nguyên, hoặc với các dạng dấu chấm động được mô tả trong phần sau.

Chúng ta sử dụng các khái niệm sau ở bên dưới: *Phạm vi* của một numeric là sự tính toán các chữ số thập phân trong phần phân số, bên phải của dấu thập phân. *Độ chính xác* của numeric là tính tổng của các chữ số có nghĩa trong toàn bộ số đó, đó là, số các chữ số ở cả 2 phía của dấu thập phân. Vì thế số 23.5141 có độ chính xác của 6 và thang số 4. Các số nguyên có thể được coi là có thang số 0.

Cả độ chính xác tối đa và thang tối đa của một cột numeric có thể được cấu hình. Để khai báo một cột dạng numeric, hãy sử dụng cú pháp:

```
NUMERIC(precision, scale)
```

Độ chính xác phải là dương, thang 0 hoặc dương. Như một sự lựa chọn:

NUMERIC(precision)

chọn một thang 0. Chỉ định:

NUMERIC

không với bất kỳ độ chính xác hoặc thang nào tạo ra một cột trong đó các giá trị của bất kỳ độ chính xác và thang nào cũng có thể được lưu trữ, cho tới giới hạn triển khai về độ chính xác. Một cột dạng này sẽ không ép các giá trị đầu vào tới bất kỳ thang đặc biệt nào, trong khi các cột numeric với một thang được khai báo sẽ ép các giá trị đầu vào tới thang đó. (Tiêu chuẩn SQL đòi hỏi một thang 0 mặc định, như, ép tới độ chính xác là số nguyên. Chúng ta thấy điều này hơi vô dụng. Nếu bạn có quan tâm về tính khả chuyển, hãy luôn chỉ định độ chính xác và thang một cách rõ ràng).

Nếu thang của một giá trị được lưu trữ là lớn hơn so với thang được khai báo của cột, thì hệ thống sẽ làm tròn giá trị tới số các chữ số thập phân được chỉ định. Sau đó, nếu số các chữ số về bên trái của dấu chấm thập phân vượt quá độ chính xác được khai báo trừ đi thang được khai báo, thì một lỗi sẽ nảy sinh.

Các giá trị số được lưu trữ một cách vật lý không với bất kỳ các số 0 thừa nào ở đầu hoặc ở sau. Vì thế, độ chính xác và thang được khai báo của một cột là tối đa, không phải là những phân bổ cố định. (Theo nghĩa này thì dạng numeric là khá giống với varchar(n) hơn là char(n)). Yêu cầu lưu trữ thực tế là 2 byte cho từng nhóm 4 chữ số thập phân, cộng với 5 tới 8 byte tổng thể.

Bổ sung thêm vào các giá trị số thông thường, dạng cho phép giá trị đặc biệt NaN, nghĩa là "không phải là một số". Bất kỳ hoạt động nào trong NaN cũng cho ra NaN khác. Khi viết giá trị này như một hằng trong lệnh SQL, bạn phải đặt các dấu nháy xung quanh nó, ví dụ UPDATE table SET x = 'NaN'. Ở đầu vào, chuỗi NaN được thừa nhận theo một cách thức không phân biệt chữ hoa chữ thường.

> **Lưu ý**: Trong hầu hết các triển khai của khái niệm "không phải là một số", NaN không được xem là ngang bằng với bất kỳ giá trị số khác nào (bao gồm cả NaN). Để cho phép các giá trị numeric sẽ được sắp xếp và sử dụng trong các chỉ số dựa vào cây, PostgreSQL đối xử với các giá trị NaN ngang bằng như nhau, và lớn hơn so với tất cả các giá trị không là NaN.

Các dạng decimal và numeric là tương đương nhau. Cả 2 dạng là một phần của tiêu chuẩn SQL.

8.1.3. Dạng dấu chấm động

Các dạng dữ liệu real và double precision là không chính xác, các dạng số có độ chính xác biến đổi. Trong thực tế, các dạng đó thường là các triển khai của tiêu chuẩn 754 của IEEE cho Số học Chấm Thập phân Nhị phân - Binary Floating Point Arithmetic (độ chính xác đơn và đúp, một cách tương ứng), ở mức độ mà trình xử lý nằm bên dưới, hệ điều hành và trình biên dịch hỗ trợ nó.

Không chính xác có nghĩa là một số giá trị không thể chuyển đổi được chính xác sang định dạng nội bộ và được lưu trữ như là những xấp xỉ gần đúng, sao cho việc lưu trữ và truy xuất một giá trị có thể chỉ ra khá khác nhau. việc quản lý các lỗi đó và cách mà chúng nhân giống thông qua các tính toán là chủ đề của toàn bộ một nhánh khoa học toán học và máy tính và sẽ không được thảo luận ở đây, ngoại trừ các điểm sau đây:

- Nếu bạn yêu cầu lưu trữ các tính toán chính xác (như đối với các tài khoản tiền tệ), hãy sử dụng dạng numeric.
- Nếu bạn muốn thực hiện các tính toán phức tạp với các dạng đó vì bất kỳ điều gì quan trọng, đặc biệt nếu bạn dựa vào hành vi nhất định trong các trường hợp biên (vô cực, tràn dưới), thì bạn nên đánh giá sự triển khai một cách cẩn thận.
- So sánh 2 giá trị chấm thập phân cho sự bằng nhau có thể không phải lúc nào cũng làm việc được như mong đợi.

Trong hầu hết các nền tảng, dạng real có một dải ít nhất là 1E-37 tới 1E+37 với độ chính xác ít nhất 6 chữ số thập phân. Dạng double precision thường có một dải khoảng 1E-307 tới 1E+308 với độ chính xác ít nhất 15 chữ số. Các giá trị mà là quá lớn hoặc quá nhỏ sẽ gây ra một lỗi. Việc làm tròn có thể diễn ra nếu độ chính xác của một số đầu vào là quá cao. Các số quá gần tới 0 mà không thể hiện được sự khác biệt với 0 sẽ gây ra lỗi tràn dưới.

Bổ sung vào các giá trị số thông thường, các dạng dấu chấm thập phân có vài giá trị đặc biệt:

```
Infinity
-Infinity
NaN
```

Chúng thể hiện các giá trị "vô cực", "âm vô cực" và "không phải là số" đặc biệt của IEEE 754, một cách tương ứng.

(Trên một máy thì tính toán số học dấu chấm thập phân không tuân theo IEEE 754, các giá trị đó có thể sẽ không làm việc như mong đợi). Khi viết các giá trị đó như các hằng số trong một lệnh SQL, bạn phải đặt các dấu nháy xung quanh chúng, ví dụ UPDATE table SET x = 'Infinity'. Ở đầu vào, các chuỗi đó được nhận theo cách phân biệt chữ hoa và chữ thường.

> **Lưu ý**: IEEE 754 chỉ định rằng NaN sẽ không so sánh ngang bằng với bất kỳ giá trị điểm chấm thập phân nào (bao gồm cả NaN). Để cho phép các giá trị điểm chấm thập phân được sắp xếp và được sử dụng trong các chỉ số dựa vào cây, PostgreSQL đối xử với các giá trị NaN ngang bằng nhau, và lớn hơn tất cả các giá trị không phải là NaN.

PostgreSQL cũng hỗ trợ các ký hiệu tiêu chuẩn SQL float và float(p) cho việc chỉ định các dạng số không chính xác. Ở đây, p chỉ định độ chính xác tối thiểu chấp nhận được theo các chữ số nhị phân. PostgreSQL chấp nhận float(1) tới float(24) khi lựa chọn dạng real, trong khi float(25) tới float(53) lựa chọn double precision. Các giá trị của p nằm ngoài dải được phép sẽ dẫn tới một lỗi. float với không có độ chính xác được chỉ định sẽ được lấy để ngụ ý double precision.

> **Lưu ý**: Trước phiên bản PostgreSQL 7.4, độ chính xác trong float(p) đã được lấy để ngụ ý quá nhiều các số thập phân. Điều này đã được sửa để khớp với tiêu chuẩn SQL, nó chỉ định rằng độ chính xác được đo đếm theo chữ số nhị phân. Giả thiết là real và double precision có chính xác 24 và 53 bit trong phần định trị một cách tương ứng là đúng cho những triển khai dấu chấm thập phân theo tiêu chuẩn IEEE. Trên các nền tảng không phải của IEEE thì nó có thể khác một chút, nhưng để đơn giản thì các dải y hệt của p sẽ được sử dụng trong tất cả các

nền tảng.

8.1.4. Dạng tuần tự (Serial)

Các dạng dữ liệu serial và bigserial không phải là các dạng đúng, mà chỉ là sự thuận tiện ký hiệu cho việc tạo ra các cột mã định danh duy nhất (tương tự như tính chất tự động tăng AUTO_INCREMENT được một số cơ sở dữ liệu khác hỗ trợ). Trong triển khai hiện hành, việc chỉ định:

```
CREATE TABLE tablename (
        colname SERIAL
);
```

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
        colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

vì thế, chúng ta đã tạo ra một cột số nguyên và đã dàn xếp cho các giá trị mặc định của nó sẽ được chỉ định từ một máy tạo sự tuần tự. Một ràng buộc NOT NULL được áp dụng để đảm bảo rằng một giá trị null không thể được chèn vào. (Trong hầu hết các trường hợp bạn cũng có thể muốn gắn một ràng buộc UNIQUE hoặc PRIMARY KEY để ngăn ngừa các giá trị đúp bản khỏi bị chèn vào ngẫu nhiên, nhưng điều này là không tự động). Cuối cùng, sự tuần tự đó được đánh dấu như là "được sở hữu bởi" cột, sao cho nó sẽ bị bỏ đi nếu cột hoặc bảng đó bị bỏ đi.

> **Lưu ý**: Trước bản PostgreSQL 7.3, serial ngụ ý là UNIQUE. Điều này không còn tự động nữa. Nếu bạn muốn một cột tuần tự có một ràng buộc độc nhất hoặc sẽ là một khóa chủ, thì bây giờ nó phải được chỉ định, hệt như bất kỳ dạng dữ liệu nào khác.

Để chèn giá trị tiếp sau của tuần tự đó vào cột serial, hãy chỉ định rằng cột serial đó sẽ được chỉ định giá trị mặc định của nó. Điều này có thể được thực hiện hoặc bằng việc loại bỏ cột khỏi danh sách các cột trong lệnh INSERT, hoặc thông qua sử dụng từ khóa DEFAULT.

Các tên dạng serial và serial4 là tương đương nhau: cả 2 tạo ra các cột integer. Các tên dạng bigserial và serial8 làm việc theo cách y hệt, ngoại trừ là chúng tạo ra một cột bigint. bigserial sẽ được sử dụng nếu bạn biết sử dụng trước hơn 2^{31} mã định dạng qua vòng đời của bảng.

Sự tuần tự được tạo ra cho một cột serial được bỏ tự động khi cột chủ bị bỏ. Bạn có thể bỏ sự tuần tự mà không bỏ cột, nhưng điều này sẽ ép loại bỏ biểu thức mặc định của cột.

8.2. Dạng tiền tệ

Dạng tiền tệ lưu trữ một lượng tiền tệ với một độ chính xác thập phân cố định; xem Bảng 8-3. Độ chính xác thập phân được thiết lập lc_monetary của cơ sở dữ liệu xác định. Đầu vào được chấp nhận trong các định dạng khác nhau, bao gồm cả các hằng số nguyên và dấu chấm thập phân, cũng như việc định dạng tiền tệ thông thường, như '$1.000.00'. Đầu ra thường ở mẫu sau nhưng phụ thuộc vào miền địa phương. Các giá trị số không nằm trong ngoặc có thể được chuyển đổi sang money bằng việc chuyển giá trị số sang text và sau đó money, ví dụ:

```
SELECT 1234::text::money;
```

Không có con đường đơn giản để làm ngược lại theo một cách thức độc lập với một miền địa phương, ấy là việc biến một giá trị money thành một dạng số. Nếu bạn biết ký hiệu tiền tệ và dấu phân cách hàng ngàn thì bạn có thể sử dụng regexp_replace():

```
SELECT regexp_replace('52093.89'::money::text, '[$,]', '', 'g')::numeric;
```

Vì đầu ra của dạng dữ liệu này là phân biệt theo miền địa phương, nó có thể không làm việc để tải các dữ liệu tiền tệ money vào trong một cơ sở dữ liệu mà có một thiết lập khác của lc_monetary. Để tránh các vấn đề đó, trước khi phục hồi một đống trong một cơ sở dữ liệu mới, hãy chắc chắn lc_monetary có giá trị y hệt hoặc tương tự như trong cơ sở dữ liệu mà đã được đáng đống.

Bảng 8-3. Các dạng tiền tệ

Tên	Kích cỡ lưu trữ	Mô tả	Dải
money	8 bytes	lượng tiền	- 92233720368547758.08 tới +92233720368547758.07

8.3. Dạng ký tự

Bảng 8-4. Các dạng ký tự

Tên	Mô tả
character varying(n) , varchar(n)	độ dài biến đổi với hạn chế
character(n) , char(n)	độ dài cố định, được lót bằng ký tự trống
text	độ dài không hạn chế và biến đổi

Bảng 8-4 chỉ ra các dạng ký tự mục đích chung trong PostgreSQL.

SQL xác định 2 dạng ký tự ban đầu: character varying(n) và character(n), trong đó n là một số nguyên dương. Cả 2 dạng đó có thể lưu trữ các chuỗi cho tới n ký tự (không phải các byte) độ dài. Một cố gắng để lưu trữ một chuỗi dài hơn trong một cột của các dạng đó sẽ gây ra một lỗi, trừ phi các ký tự vượt quá tất cả đều là các khoảng trống, trong trường hợp đó chuỗi sẽ bị cắt ngắn về độ dài cực đại. (Đây là một ngoại lệ khá kỳ lạ được tiêu chuẩn SQL yêu cầu). Nếu chuỗi sẽ được lưu trữ là ngắn hơn so với độ dài được khai báo, thì các giá trị của dạng character sẽ được lót bằng các ký tự trống; các giá trị của dạng character varying đơn giản sẽ lưu trữ chuỗi ngắn hơn.

Nếu một chuỗi đưa ra rõ ràng một giá trị tới character varying(n) hoặc character(n), thì giá trị dài quá sẽ bị cắt về n ký tự mà không làm sinh ra một lỗi. (Điều này cũng được tiêu chuẩn SQL yêu cầu).

Các ký hiệu varchar(n) và char(n) là các tên hiệu cho varying(n) và character(n), một cách tương ứng. character không có độ dài chỉ định là tương đương với character(1). Nếu character varying được sử dụng mà không có chỉ định độ dài, thì dạng đó chấp nhận các chuỗi kích cỡ bất kỳ. Cái sau là một mở rộng của PostgreSQL.

Hơn nữa, PostgreSQL đưa ra dạng text, nó lưu trữ các chuỗi độ dài bất kỳ. Dù dạng text là không

theo tiêu chuẩn SQL, thì vài hệ quản trị cơ sở dữ liệu SQL khác cũng có nó.

Các giá trị dạng character về mặt vật lý được lót với các ký tự trống tới độ rộng n được chỉ định, và được lưu trữ và được hiển thị theo cách đó. Tuy nhiên, các ký tự trống lót đó được đối xử đáng kể theo ngữ nghĩa. Các ký tự trống đi sau đuôi không được để ý khi so sánh 2 giá trị dạng character, và chúng sẽ bị loại bỏ khi chuyển đổi một giá trị character sang một giá trị của các dạng chuỗi khác. Lưu ý rằng các ký tự trống ở đuôi là đáng kể về ngữ nghĩa trong các giá trị character varying và text.

Yêu cầu lưu trữ cho một chuỗi ngắn (tới 126 byte) là 1 byte cộng với chuỗi thực tế đó, nó bao gồm việc lót ký tự trống trong trường hợp của character. Các chuỗi dài hơn có 4 byte tổng thể thay vì 1. Các chuỗi dài được hệ thống nén một cách tự động, nên yêu cầu vật lý trên đĩa có thể ít hơn. Các giá trị rất dài cũng được lưu trữ trong các bảng nền sao cho chúng không can thiệp với sự truy cập nhanh tới các giá trị cột ngắn hơn. Trong mọi trường hợp, chuỗi ký tự có khả năng dài nhất mà có thể được lưu trữ là khoảng 1 GB. (Giá trị cực đại sẽ được phép cho n trong khai báo dạng dữ liệu là ít hơn thế. Có lẽ là hữu dụng để thay đổi điều này vì với những mã hóa ký tự nhiều byte thì số các ký tự và các byte có thể hoàn toàn khác nhau. Nếu bạn muốn lưu trữ các chuỗi dài mà không có giới hạn trên cụ thể, hãy sử dụng text hoặc character varying mà không có chỉ định độ dài, thay vì tạo một giới hạn độ dài tùy ý).

> **Mẹo**: Không có sự khác biệt hiệu năng giữa 3 dạng đó, ngoại trừ khoảng trống lưu trữ gia tăng khi sử dụng dạng được lót bằng ký tự trống, và một ít các chu kỳ dư thừa của CPU để kiểm tra độ dài khi việc lưu trữ trong một cột bị ràng buộc về độ dài. Trong khi character(n) có các ưu thế hiệu năng trong một số hệ thống cơ sở dữ liệu khác, thì không có ưu thế như vậy trong PostgreSQL; trong thực tế character(n) thường là chậm nhất trong 3 dạng vì các chi phí lưu trữ bổ sung của nó. Trong hầu hết các tình huống, nên thay bằng việc sử dụng text hoặc character varying.

Tham chiếu tới Phần 4.1.2.1 để có thông tin về cú pháp các hằng chuỗi, và tới Chương 9 để có thông tin về các toán tử và các hàm có sẵn. Tập hợp các ký tự cơ sở dữ liệu xác định tập hợp các ký tự được sử dụng để lưu trữ các giá trị văn bản; để có thêm thông tin về sự hỗ trợ tập hợp các ký tự, hãy tham chiếu tới Phần 22.2.

Ví dụ 8-1. Sử dụng các dạng ký tự

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- [1]

a    | char_length
-----+-------------
ok   | 2

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good ');
INSERT INTO test2 VALUES ('too long');
ERROR: value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit truncation
SELECT b, char_length(b) FROM test2;
```

```
b          | char_length
-----------+-------------
ok         | 2
good       | 5
too l      | 5
```

[1] Hàm char_length được thảo luận trong Phần 9.4.

Có 2 dạng ký tự độ dài cố định khác trong PostgreSQL, được chỉ ra trong Bảng 8-5. Dạng name chỉ tồn tại cho việc lưu trữ các mã định danh trong các catalog hệ thống nội bộ và không có ý định để sử dụng đối với người sử dụng thông thường. Chiều dài của nó hiện được xác định như là 64 byte (63 ký tự dùng được cộng với ký tự kết thúc) nhưng sẽ được tham chiếu bằng việc sử dụng hằng NAMEDATALEN trong mã nguồn C. Độ dài này được thiết lập vào thời điểm biên dịch (và vì thế có khả năng tinh chỉnh được cho các sử dụng đặc biệt); độ dài cực đại mặc định có thể thay đổi trong một phiên bản trong tương lai. Dạng "char" (lưu ý các dấu ngoặc kép) là khác với char(1) theo đó nó chỉ sử dụng một byte lưu trữ. Nó được sử dụng nội bộ trong các catalog hệ thống như một dạng đếm liệt kê đơn giản.

Bảng 8-5. Các dạng ký tự đặc biệt

Tên	Kích cỡ lưu trữ	Mô tả
"char"	1 byte	dạng nội bộ 1 byte
name	64 bytes	dạng nội bộ cho các tên đối tượng

8.4. Dạng dữ liệu nhị phân

Dạng dữ liệu bytea cho phép lưu trữ các chuỗi nhị phân; xem Bảng 8-6.

Bảng 8-6. Các dạng dữ liệu nhị phân

Tên	Kích cỡ lưu trữ	Mô tả
bytea	1 hoặc 4 byte cộng với chuỗi nhị phân thực tế	chuỗi nhị phân độ dài biến đổi

Chuỗi nhị phân là một sự tuần tự của bộ từng 8 bit một (hoặc các byte). Các chuỗi nhị phân được phân biệt với các chuỗi ký tự theo 2 cách. Trước hết, các chuỗi nhị phân đặc biệt cho phép lưu trữ các byte giá trị 0 và các byte khác "không in được" (thường là, các byte bên ngoài dãy 32 tới 126). Các chuỗi ký tự không cho phép các byte 0, và cũng không cho phép bất kỳ giá trị byte nào khác và các tuần tự của các giá trị byte mà không hợp lệ theo bộ mã ký tự được chọn của cơ sở dữ liệu. Thứ 2, các hoạt động trong các chuỗi nhị phân xử lý các byte thực sự, trong khi việc xử lý các chuỗi ký tự phụ thuộc vào các thiết lập theo miền địa phương. Ngắn gọn, các chuỗi nhị phân là phù hợp cho việc lưu trữ các dữ liệu mà người lập trình nghĩ như các "byte thô", trong khi các chuỗi ký tự là phù hợp cho việc lưu trữ văn bản.

Dạng bytea hỗ trợ 2 định dạng bên ngoài cho đầu vào và đầu ra: định dạng "thoát" ("escape") theo lịch sử của PostgreSQL, và định dạng "hex". Cả 2 chúng đều luôn được chấp nhận ở đầu vào. Định

dạng đầu ra phụ thuộc vào tham số cấu hình bytea_output; mặc định là hex. (Lưu ý rằng định dạng hex đã được giới thiệu trong PostgreSQL 9.0; các phiên bản sớm và một số công cụ không hiểu nó).

Tiêu chuẩn SQL xác định một dạng chuỗi nhị phân khác, gọi là BLOB hoặc BINARY LARGE OBJECT. Định dạng đầu vào là khác với bytea, nhưng các hàm và toán tử được cung cấp hầu hết như nhau.

8.4.1. Định dạng bytea hex

Định dạng "hex" mã hóa các dữ liệu nhị phân như là 2 chữ số hệ 16 (hexadecimal) cho 1 byte, đáng kể nhất là phần đầu. Toàn bộ chuỗi được đi đầu bằng tuần tự \x (phân biệt nó với định dạng thoát). Trong một số ngữ cảnh, dấu chéo ngược ban đầu có thể cần phải được thoát bằng việc viết nó 2 lần, trong các trường hợp y hệt theo đó các dấu chéo ngược phải được đúp bản ở định dạng thoát; các chi tiết ở bên dưới. Các chữ số theo hệ 16 có thể hoặc chữ hoa hoặc chữ thường, và dấu trắng được phép giữa các cặp chữ số (nhưng không nằm trong một cặp chữ số cũng không nằm ở đầu \x tuần tự). Định dạng hex là tương thích với một dải rộng lớn các ứng dụng và các giao thức bên ngoài, và nó có xu hướng sẽ là nhanh hơn để chuyển đổi hơn là định dạng thoát, nên việc sử dụng nó được ưu tiên. Ví dụ:

```
SELECT E'\\xDEADBEEF';
```

8.4.2. Định dạng thoát bytea

Định dạng "thoát" ("escape") là định dạng truyền thống của PostgreSQL đối với dạng bytea. Nó lấy tiếp cận thể hiện một chuỗi nhị phân như một sự tuần tự các ký tự ASCII, trong khi chuyển đổi các byte mà không thể được thể hiện như một ký tự ASCII trong các tuần tự thoát đặc biệt. Nếu, từ quan điểm ứng dụng việc thể hiện các byte như các ký tự là có ý nghĩa, thì sau đó sự thể hiện này có thể là thuận tiện. Nhưng trong thực tế thường lúng túng vì nó đưa ra sự khác biệt giữa các chuỗi nhị phân và các chuỗi ký tự, và cả cơ chế thoát đặc biệt đã được chọn là thứ gì đó khó sử dụng. Vì thế định dạng này có lẽ nên được tránh đối với hầu hết các ứng dụng mới.

Khi vào các dữ liệu bytea trong định dạng thoát, các byte của các giá trị nhất định phải được thoát, trong khi tất cả các giá trị byte có thể được thoát. Nói chung, để thoát một byte, hãy chuyển đổi nó thành giá trị byte 3 ký tự số của nó và đặt trước nó bằng một dấu chéo ngược (hoặc 2 dấu chéo ngược, nếu việc viết giá trị đó như một hằng bằng việc sử dụng cú pháp chuỗi thoát). Bản thân dấu chéo ngược (byte giá trị 92) có thể như một lựa chọn được thể hiện bằng 2 dấu chéo ngược. Bảng 8-7 chỉ ra các ký tự phải được thoát, và đưa ra các tuần tự thoát lựa chọn thay thế ở những nơi áp dụng được.

Bảng 8-7. Các byte thoát hằng bytea

Giá trị byte thập phân	Mô tả	Trình bày đầu vào được thoát	Ví dụ	Trình bày đầu ra
0	zero octet	E'\\000'	SELECT E'\\000'::bytea;	\000
39	nháy đơn	'''' hoặc E'\\047'	SELECT E'\''::bytea;	'
92	dấu chéo ngược	E'\\\\' hoặc E'\\134'	SELECT E'\\\\'::bytea;	\\

Giá trị byte thập phân	Mô tả	Trình bày đầu vào được thoát	Ví dụ	Trình bày đầu ra
0 tới 31 và 127 tới 255	Bộ 8 bit (byte) “không in được”	E’\\xxx’ (giá trị hệ 8)	SELECT E’\\001’::bytea;	\001

Yêu cầu để thoát các byte không in được khác nhau, phụ thuộc vào các thiết lập miền địa phương. Trong một số trường hợp bạn có thể đi mà vẫn để lại chúng không được thoát. Lưu ý là kết quả trong từng trong số các ví dụ trong Bảng 8-7 chính xác là một byte độ dài, thậm chí dù trình bày đầu ra đôi khi hơn một ký tự.

Lý do nhiều dấu chéo ngược được yêu cầu, như được chỉ ra trong Bảng 8-7, là một chuỗi đầu vào được viết như một hằng chuỗi phải truyền qua 2 pha phân tích cú pháp trong máy chủ PostgreSQL. Dấu chéo ngược đầu tiên của từng đôi được hiểu như là một ký tự thoát bằng trình phân tích cú pháp hằng chuỗi (giả thiết cú pháp chuỗi thoát được sử dụng) và vì thế được sử dụng, để lại dấu chéo ngược thứ 2 của đôi. (Các chuỗi trong các dấu $ có thể được sử dụng để tránh mức thoát này). Dấu chéo ngược còn lại sau đó được các hàm đầu vào `bytea` nhận như là việc khởi đầu hoặc một giá trị byte 3 chữ số, hoặc việc thoát dấu chéo ngược khác. Ví dụ, một hằng chuỗi đã truyền qua tới máy chủ như là `E'\\001'` trở thành `\001` sau khi truyền qua trình phân tích cú pháp chuỗi thoát. `\001` sau đó được gửi tới hàm đầu vào `bytea`, nơi mà nó được chuyển đổi thành một byte duy nhất với một giá trị thập phân của 1. Lưu ý là ký tự nháy đơn không được `bytea` đối xử đặc biệt, nên nó tuân theo các qui tắc cho các hằng chuỗi. (Xem Phần 4.1.2.1.).

Các byte `bytea` đôi khi được thoát khi có đầu ra. Nói chung, mỗi byte “không in được” được chuyển đổi thành giá trị byte 3 chữ số tương đương và đi trước với một dấu chéo ngược. Hầu hết các byte “in được” được trình bày bằng sự trình bày tiêu chuẩn của chúng trong tập ký tự máy trạm. Byte với giá trị thập phân 92 (dấu chéo ngược) được đúp bản ở đầu ra. Các chi tiết ở trong Bảng 8-8.

Bảng 8-8. Các byte thoát đầu ra bytea

Giá trị byte thập phân	Mô tả	Trình bày đầu ra được thoát	Ví dụ	Kết quả đầu ra
92	dấu chéo ngược	\\	SELECT E’\\134’::bytea;	\\
0 tới 31 và 127 tới 255	các byte “không in được”	\xxx (giá trị hệ 8)	SELECT E’\\001’::bytea;	\001
32 tới 126	các byte “in được”	đại diện bộ ký tự máy trạm	SELECT E’\\176’::bytea;	~

Phụ thuộc vào mặt tiền (front end) đối với PostgreSQL mà bạn sử dụng, bạn có thể có công việc bổ sung để làm trong các chuỗi thoát và không thoát `bytea`. Ví dụ, bạn có thể cũng phải thoát các đường nuôi và vận chuyển ngược trở về nếu giao diện của bạn tự động dịch chúng.

8.5. Dạng Date/Time (Ngày tháng/Thời gian)

PostgreSQL hỗ trợ một tập hợp đầy đủ các dạng dữ liệu về thời gian, như trong Bảng 8-9. Các hoạt

động sẵn sàng trong các dạng dữ liệu đó được mô tả trong Phần 9.9.

Bảng 8-9. Các dạng ngày tháng/thời gian

Tên	Kích cỡ lưu trữ	Mô tả	Giá trị thấp	Giá trị cao	Giải pháp
timestamp [(p)] [without time zone]	8 bytes	cả ngày tháng và thời gian (không có vùng thời gian)	4713 BC	294276 AD	1 microsecond / 14 chữ số
timestamp [(p)] with time zone	8 bytes	cả ngày tháng và thời gian, có vùng thời gian	4713 BC	294276 AD	1 microsecond / 14 chữ số
date	4 bytes	ngày tháng (không có thời gian của ngày)	4713 BC	5874897 AD	1 ngày
time [(p)] [without time zone]	8 bytes	thời gian của ngày (không có ngày tháng)	00:00:00	24:00:00	1 microsecond / 14 chữ số
time [(p)] with time zone	12 bytes	chỉ có thời gian của ngày, có vùng thời gian	00:00:00+1459	24:00:00-1459	1 microsecond / 14 chữ số
interval [fields][(p)]	12 bytes	khoảng thời gian	-178000000 năm	178000000 năm	1 microsecond / 14 chữ số

> **Lưu ý**: Tiêu chuẩn SQL yêu cầu là viết chỉ timestamp thì sẽ tương đương với timestamp without time zone, và PostgreSQL trung thành với hành vi đó. (Các phiên bản trước 7.3 đã đối xử với nó như là timestamp with time zone).

time, timestamp và interval chấp nhận một giá trị với độ chính xác tùy chọn p, nó chỉ định số các chữ số thập phân còn được giữ trong trường giây. Mặc định, không có ràng buộc rõ ràng nào về độ chính xác. Dải được phép của p là từ 0 tới 6 cho các dạng timestamp và interval.

> **Lưu ý**: Khi các giá trị timestamp được lưu trữ như là các số nguyên 8 byte (hiện là mặc định), thì độ chính xác microsecond (micro giây) là sẵn sàng đối với toàn bộ dải các dữ liệu. Khi các giá trị timestamp được lưu giữ như là các số dấu chấm thập phân độ chính xác đúp (lựa chọn biên dịch thời gian đối nghịch), thì giới hạn độ chính xác có hiệu quả có thể ít hơn 6, các giá trị timestamp được lưu trữ như là giây trước hoặc sau nửa đêm 2000-01-01. Khi các giá trị timestamp được triển khai bằng việc sử dụng các số dấu chấm thập phân, thì độ chính xác microsecond sẽ đạt được cho các ngày tháng trong một ít năm của 2000-01-01, nhưng độ chính xác giảm cho ngày tháng xa hơn. Lưu ý rằng việc sử dụng datetimes dấu chấm thập phân cho phép một dải rộng lớn hơn các giá trị timestamp sẽ được trình bày nhiều hơn so với ở trên: từ 4713 BC cho tới 5874897 AD.
>
> Lựa chọn biên dịch thời gian y hệt cũng xác định liệu các giá trị time và interval có được lưu trữ như là các số dấu chấm thập phân hay các số nguyên 8 byte hay không. Trong trường hợp dấu chấm thập phân, các giá trị interval lớn sẽ giảm độ chính xác khi kích cỡ của khoảng (interval) gia tăng.

Đối với các dạng time, dải được phép của p là từ 0 tới 6 khi lưu trữ số nguyên 8 byte được sử dụng,

hoặc từ 0 tới 10 khi lưu trữ dấu thập phân được sử dụng.

Dạng interval có một lựa chọn bổ sung để hạn chế tập hợp các trường được lưu trữ bằng việc viết một trong các cách sau:

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

Lưu ý là nếu cả fields và p được chỉ định, thì fields phải bao gồm SECOND, vì độ chính xác áp dụng chỉ cho các giây.

Dạng thời gian với vùng thời gian (time with time zone) được tiêu chuẩn SQL xác định, nhưng định nghĩa đưa ra các đặc tính dẫn tới sự hữu dụng đáng ngờ. Trong hầu hết các trường hợp, sự kết hợp của date, time, timestamp without time zone và timestamp with time zone sẽ đưa ra một dải hoàn chỉnh chức năng ngày tháng/thời gian được bất kỳ ứng dụng nào yêu cầu.

Các dạng abstime và reltime là các dạng có độ chính xác thấp hơn và được sử dụng trong nội bộ. Bạn không được khuyến khích sử dụng các dạng đó trong các ứng dụng; các dạng nội bộ đó có thể biến mất trong một phiên bản trong tương lai.

8.5.1. Đầu vào ngày tháng/thời gian

Đầu vào ngày tháng và thời gian được chấp nhận trong hầu hết bất kỳ định dạng hợp lý nào, bao gồm cả ISO 8601, tương thích SQL, POSTGRES truyền thống và các dịnh dạng khác. Đối với một số định dạng, trật tự ngày, tháng và năm trong đầu vào ngày tháng là không xác định và có sự hỗ trợ cho việc xác định trật tự được mong đợi của các trường đó. Thiết lập tham số DateStyle về MDY để chọn tháng-ngày-năm, DMY để chọn ngày-tháng-năm, hoặc YDM để chọn năm-tháng-ngày.

PostgreSQL mềm dẻo hơn trong việc điều khiển đầu vào ngày tháng/thời gian so với tiêu chuẩn SQL yêu cầu. Xem Phụ lục B để có các qui tắc chính xác phân tích cú pháp đầu vào ngày tháng/thời gian và cho các trường văn bản được thừa nhận bao gồm tháng, ngày của tuần và vùng thời gian.

Nhớ rằng bất kỳ đầu vào hằng ngày tháng hoặc thời gian nào cũng cần phải được đưa vào các nháy đơn, giống như các chuỗi văn bản. Hãy tham chiếu tới Phần 4.1.2.7 để có thêm thông tin. SQL yêu cầu cú pháp sau:

```
type [ (p) ] 'value'
```

trong đó p là một đặc tả độ chính xác tùy chọn đưa ra số các chữ số thập phân trong trường giây seconds. Độ chính xác có thể được chỉ định cho các dạng time, timestamp và interval. Các giá trị được phép được nhắc tới ở trên. Nếu không có độ chính xác nào được chỉ định trong một đặc tả

hằng, thì nó mặc định cho độ chính xác của giá trị hằng đó.

8.5.1.1. Ngày tháng

Bảng 8-10 chỉ một số đầu vào có khả năng cho dạng ngày tháng date.

Bảng 8-10. Đầu vào ngày tháng

Ví dụ	Mô tả
1999-01-08	ISO 8601; Ngày 8 tháng 1 ở mọi chế độ (định dạng được khuyến cáo)
January 8, 1999	mập mờ trong mọi chế độ đầu vào dạng ngày tháng datestyle
1/8/1999	Ngày 8 tháng 1 theo chế độ MDY; ngày 1 tháng 8 theo chế độ DMY
1/18/1999	Ngày 18 tháng 1 theo chế độ MDY; bị từ chối trong các chế độ khác
01/02/03	Ngày 2 tháng 1 năm 2003 theo chế độ MDY, ngày 1 tháng 2 năm 2003 theo chế độ DMY; ngày 3 tháng 2 năm 2001 theo chế độ YMD
1999-Jan-08	Ngày 8 tháng 1 trong bất kỳ chế độ nào
Jan-08-1999	Ngày 8 tháng 1 trong bất kỳ chế độ nào
08-Jan-1999	Ngày 8 tháng 1 trong bất kỳ chế độ nào
99-Jan-08	Ngày 8 tháng 1 theo YMD, còn lại thì lỗi
08-Jan-99	Ngày 8 tháng 1, ngoại trừ có lỗi theo chế độ YMD
Jan-08-99	Ngày 8 tháng 1, ngoại trừ có lỗi theo chế độ YMD
19990108	ISO 8601; ngày 8 tháng 1 năm 1999 theo bất kỳ chế độ nào
990108	ISO 8601; ngày 8 tháng 1 năm 1999 theo bất kỳ chế độ nào
1999.008	năm và ngày của năm
J2451187	ngày theo lịch Julian
January 8, 99 BC	Năm 99 BC

8.5.1.2. Thời gian

Các dạng thời gian của ngày (time-of-day) là time [(p)] without time zone và time [(p)] with time zone. time là tương đương một mình với time without time zone.

Đầu vào hợp lệ cho các dạng đó bao gồm thời gian của ngày theo sau là một vùng thời gian tùy chọn. (Xem Bảng 8-11 và Bảng 8-12). Nếu một vùng thời gian được chỉ định ở đầu vào cho time without time zone, thì nó âm thầm được bỏ qua. Bạn cũng có thể chỉ định một ngày tháng nhưng nó sẽ bị bỏ qua, ngoại trừ khi bạn sử dụng một tên vùng thời gian có liên quan tới một qui tắc tiết kiệm ánh sáng ban ngày như America/New_York. Trong trường hợp này việc chỉ định ngày tháng được yêu cầu để xác định liệu tiêu chuẩn hoặc thời gian tiết kiệm ánh sáng ban ngày có được áp dụng hay không. Sự bù trừ vùng thời gian phù hợp được ghi nhận trong giá trị time with time zone.

Bảng 8-11. Đầu vào thời gian

Ví dụ	Mô tả
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	hệt như 04:05; AM không ảnh hưởng tới giá trị
04:05 PM	hệt như 16:05; giờ đầu vào phải là <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	vùng thời gian được chỉ định viết tắt
2003-04-12 04:05:06 America/New_York	vùng thời gian được tên đầy đủ chỉ định

Bảng 8-12. Đầu vào vùng thời gian

Ví dụ	Mô tả
PST	Viết tắt (cho thời gian tiêu chuẩn Thái Bình dương)
America/New_York	Tên đầy đủ của vùng thời gian
PST8PDT	Đặc tả vùng thời gian kiểu POSIX
-8:00	ISO-8601 phần bù cho PST
-800	ISO-8601 phần bù cho PST
-8	ISO-8601 phần bù cho PST
zulu	Viết tắt kiểu quân sự cho UTC
z	Dạng ngắn gọn của zulu

Tham khảo Phần 8.5.3 để có thêm thông tin về cách chỉ định các vùng thời gian.

8.5.1.3. Dấu thời gian

Đầu vào hợp lệ cho các dạng dấu thời gian cấu tạo từ sự ghép một ngày tháng và một thời gian, theo sau là một vùng thời gian tùy chọn, theo sau là một lựa chọn AD hoặc BC. (Như một lựa chọn, AD/BC có thể xuất hiện trước vùng thời gian, nhưng đây không là trật tự được ưu tiên). Vì thế:

```
1999-01-08 04:05:06
```

và

```
1999-01-08 04:05:06 -8:00
```

là các giá trị hợp lệ, chúng tuân theo tiêu chuẩn ISO 8601. Hơn nữa, định dạng chung:

January 8 04:05:06 1999 PST

được hỗ trợ.

Tiêu chuẩn SQL phân biệt các hằng timestamp without time zone và timestamp with time zone bằng sự hiện diện của một biểu tượng dấu "+" hoặc "-" và bù trừ vùng thời gian sau thời gian đó. Vì thế, theo tiêu chuẩn,

TIMESTAMP '2004-10-19 10:23:54'

là một timestamp without time zone, trong khi

TIMESTAMP '2004-10-19 10:23:54+02'

là một timestamp with time zone. PostgreSQL không bao giờ kiểm tra nội dung của một chuỗi hằng trước khi xác định dạng của nó, và vì thế sẽ đối xử với cả 2 ở trên như là timestamp without time zone. Để đảm bảo rằng một hằng được đối xử như là timestamp with time zone, hãy trao cho nó dạng rõ ràng đúng:

TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'

Trong một hằng mà từng được xác định sẽ là timestamp without time zone, PostgreSQL sẽ âm thầm bỏ qua bất kỳ chỉ định vùng thời gian nào. Đó là, giá trị kết quả được dẫn xuất từ các trường ngày tháng/thời gian ở giá trị đầu vào, và không được tinh chỉnh cho vùng thời gian.

Đối với timestamp with time zone, giá trị ban đầu được lưu trữ luôn là theo Thời gian Được điều phối Vạn năng - UTC (Universal Coordinated Time), theo truyền thống được biết tới như là Giờ mặt trời – GMT (Greenwich Mean Time). Một giá trị đầu vào có một vùng thời gian rõ ràng được chỉ định được chuyển đổi thành UTC bằng việc sử dụng bù trừ phù hợp cho vùng thời gian đó. Nếu không vùng nào được công bố trong chuỗi đầu vào, thì được giả thiết sẽ là trong vùng thời gian được tham số vùng thời gian (timezone) của hệ thống chỉ định, và được chuyển đổi thành UTC bằng việc sử dụng bù trừ cho vùng timezone.

Khi một giá trị timestamp with time zone là đầu ra, nó luôn được chuyển đổi từ UTC sang vùng timezone hiện hành, và được hiển thị như là thời gian địa phương theo vùng đó. Để xem thời gian trong vùng thời gian khác, hoặc thay đổi timezone hoặc sử dụng cấu trúc AT TIME ZONE (xem Phần 9.9.3).

Những biến đổi giữa timestamp without time zone và timestamp with time zone thường giả thiết rằng giá trị timestamp without time zone sẽ được lấy hoặc được đưa ra như là timezone thời gian địa phương. Một vùng thời gian khác có thể được chỉ định cho sự biến đổi đó bằng việc sử dụng AT TIME ZONE.

8.5.1.4. Giá trị đặc biệt

PostgreSQL hỗ trợ vài giá trị đầu vào ngày tháng/thời gian đặc biệt vì sự tiện lợi, như được nêu ở Bảng 8-13. Giá trị infinity và -infinity được trình bày đặc biệt bên trong hệ thống và sẽ được hiển thị không thay đổi; nhưng những giá trị khác đơn giản là những tốc ký sẽ được biến đổi thành các giá trị ngày tháng/thời gian thông thường khi đọc. (Đặc biệt, now và các chuỗi có liên quan được

chuyển đổi thành một giá trị thời gian đặc biệt ngay khi chúng được đọc). Tất cả các giá trị đó cần phải được đưa vào các nháy đơn khi được sử dụng như những hằng số trong các lệnh SQL.

Bảng 8-13. Các đầu vào ngày tháng/thời gian đặc biệt

Chuỗi đầu vào	Các dạng hợp lệ	Mô tả
epoch	date , timestamp	1970-01-01 00:00:00+00 (thời gian 0 theo hệ thống Unix)
infinity	date , timestamp	chậm hơn tất cả các dấu thời gian khác
-infinity	date , timestamp	sớm hơn tất cả các dấu thời gian khác
now	date , time , timestamp	thời gian bắt đầu giao dịch hiện hành
today	date , timestamp	nửa đêm hôm nay
tomorrow	date , timestamp	nửa đêm ngày mai
yesterday	date , timestamp	nửa đêm hôm qua
allballs	time	00:00:00.00 UTC

Các hàm tương thích SQL sau đây cũng có thể được sử dụng để giành được giá trị thời gian hiện hành đối với dạng dữ liệu tương ứng: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP. 4 dạng sau chấp nhận một đặc tả độ chính xác phần giây tùy chọn. (Xem Phần 9.9.4). Lưu ý chúng là các hàm SQL và không được thừa nhận trong các chuỗi đầu vào dữ liệu.

8.5.2. Đầu ra ngay tháng/thời gian

Định dạng đầu ra của các dạng ngày tháng/thời gian có thể được thiết lập cho 1 trong 4 dạng ISO 8601, SQL (Ingres), POSTGRES truyền thống (định dạng dữ liệu Unix), hoặc German (Đức). Mặc định là định dạng ISO. (Tiêu chuẩn SQL yêu cầu sử dụng định dạng ISO 8601. Tên của định dạng đầu ra "SQL" là một sự việc lịch sử). Bảng 8-14 chỉ các ví dụ của từng dạng đầu ra. Đầu ra của các dạng date và time tất nhiên chỉ là phần ngày tháng hoặc thời gian tuân theo các ví dụ được đưa ra.

Bảng 8-14. Các kiểu đầu ra ngày tháng/thời gian

Đặc tả kiểu	Mô tả	Ví dụ
ISO	tiêu chuẩn ISO 8601/SQL	1997-12-17 07:37:16-08
SQL	kiểu truyền thống	12/17/1997 07:37:16.00 PST
POSTGRES	kiểu gốc ban đầu	Wed Dec 17 07:37:16 1997 PST
German	kiểu vùng	17.12.1997 07:37:16.00 PST

Trong các kiểu SQL và POSTGRES, ngày xuất hiện trước tháng nếu trật tự trường DMY từng được chỉ định, nếu khác thì tháng sẽ xuất hiện trước ngày. (Xem Phần 8.5.1 để biết cách điều này thiết lập cũng ảnh hưởng tới sự nhận biết các giá trị đầu vào). Bảng 8-15 chỉ ra một ví dụ.

Bảng 8-15. Các biến đổi trật tự ngày tháng

Thiết lập datestyle	Trật tự đầu vào	Đầu ra
SQL, DMY	day/month/year	17/12/1997 15:37:16.00 CET
SQL, MDY	month/day/year	12/17/1997 07:37:16.00 PST
Postgres, DMY	day/month/year	Wed 17 Dec 07:37:16 1997 PST

Các kiểu ngày tháng/thời gian có thể được người sử dụng chọn bằng lệnh SET datestyle, tham số DateStyle trong tệp cấu hình postgresql.conf, hoặc biến môi trường PGDATESTYLE trên máy chủ hoặc máy trạm. Hàm định dạng to_char (xem Phần 9.8) cũng sẵn sàng như một cách thức mềm dẻo hơn để định dạng kết quả đầu ra ngày tháng/thời gian.

8.5.3. Vùng thời gian

Các vùng thời gian, và các qui ước vùng thời gian chịu ảnh hưởng bởi các quyết định chính trị, chứ không chỉ của địa lý trái đất. Vùng thời gian trên thế giới đã trở thành thứ gì đó được tiêu chuẩn hóa vào những năng 1990, nhưng tiếp tục hỏng đối với những thay đổi tùy ý, đặc biệt về các qui định tiết kiệm ánh sáng ban ngày. PostgreSQL sử dụng cơ sở dữ liệu vùng thời gian được sử dụng rộng rãi đối với thông tin về các qui tắc vùng thời gian theo lịch sử. Đối với thời gian trong tương lai, giả thiết là các qui tắc được biết tới mới nhất cho một vùng thời gian được đưa ra sẽ tiếp tục được quan sát vô hạn định trong tương lai.

Những cố gắng xây dựng PostgreSQL tương thích được với các định nghĩa tiêu chuẩn SQL để sử dụng điển hình. Tuy nhiên, tiêu chuẩn SQL có một sự pha trộn kỳ lạ các dạng và khả năng của dữ liệu ngày tháng và thời gian. Có 2 vấn đề rõ ràng là:

- Dù dạng date không thể có một vùng thời gian có liên quan, thì dạng time lại có thể. Các vùng trong thế giới thực có ít ý nghĩa, trừ phi có liên quan tới ngày tháng cũng như thời gian, vì sự bù trừ có thể khác nhau trong năm với các biên thời gian tiết kiệm ánh sáng ban ngày.
- Vùng thời gian mặc định được chỉ định như một bù trừ số không đổi đối với UTC. Vì thế không có khả năng để thích nghi cho thời gian tiết kiệm ánh sáng ban ngày khi thực hiện tính toán số học cho ngày tháng/thời gian khắp các biên DST.

Để giải quyết các khó khăn đó, chúng tôi khuyến cáo sử dụng các dạng ngày tháng/thời gian có chứa cả ngày tháng và thời gian khi sử dụng các vùng thời gian. Chúng tôi không khuyến cáo sử dụng dạng time with time zone (dù nó được PostgreSQL hỗ trợ cho các ứng dụng đã có trước rồi và vì sự thuận tiện với tiêu chuẩn SQL). PostgreSQL giả thiết vùng thời gian bản địa của bạn cho bất kỳ dạng nào chỉ chứa ngày tháng hoặc thời gian.

Tất cả các ngày tháng và thời gian mà vùng thời gian hiểu được sẽ được lưu nội bộ trong UTC. Chúng được chuyển đổi sang thời gian bản địa trong vùng được tham số cấu hình timezone chỉ định trước khi được hiển thị cho máy khách.

PostgreSQL cho phép bạn chỉ định các vùng thời gian ở 3 dạng khác nhau:

- Tên vùng thời gian đầy đủ, ví dụ America/New_York. Các tên vùng thời gian được thừa nhận sẽ được liệt kê trong kiểu nhìn pg_timezone_names (xem Phần 45.60). PostgreSQL sử dụng dữ liệu vùng thời gian được sử dụng rộng rãi zoneinfo cho mục đích này, nên các tên y hệt cũng được nhiều phần mềm khác hiểu được.
- Một viết tắt vùng thời gian, ví dụ PST. Một đặc tả như vậy chỉ xác định một bù trừ đặc biệt từ UTC, ngược lại với các tên vùng thời gian đầy đủ có thể cũng ngụ ý một tập hợp các qui tắc ngày tháng biến đổi (transitiondate) của việc tiết kiệm ánh sáng ban ngày. Những viết tắt được thừa nhận được liệt kê trong kiểu nhìn pg_timezone_abbrevs (xem Phần 45.59). Bạn không thể thiết lập các tham số cấu hình timezone hoặc log_timezone thành một sự viết tắt vùng thời gian, nhưng bạn có thể sử dụng những viết tắt đó trong các giá trị đầu vào ngày tháng/thời gian và với toán tử AT TIME ZONE.
- Bổ sung thêm vào các tên và các viết tắt vùng thời gian, PostgreSQL sẽ chấp nhận các đặc tả vùng thời gian dạng POSIX đối với dạng STDoffset hoặc STDoffsetDST, trong đó STD là một viết tắt vùng tiết kiệm ánh sáng ban ngày tùy chọn, được giả thiết thay cho một giờ trước sự bù trừ được đưa ra. Ví dụ, nếu EST5EDT còn chưa là một tên vùng được thừa nhận, thì có thể được chấp nhận và có thể, về chức năng, tương đương với thời gian Bờ Đông nước Mỹ (United State East Coast time). Khi một tên vùng tiết kiệm ánh sáng ban ngày là hiện diện, được giả thiết sẽ được sử dụng theo cùng các qui tắc biến đổi tiết kiệm ánh sáng ban ngày được sử dụng ở khoản đầu vào posixrules của cơ sở dữ liệu vùng thời gian zoneinfo. Trong một cài đặt PostgreSQL tiêu chuẩn, posixrules là hệt như US/Eastern, sao cho các đặc tả vùng thời gian dạng POSIX tuân theo các qui định tiết kiệm ánh sáng ban ngày của Mỹ. Nếu cần, bạn có thể tinh chỉnh hành vi này bằng việc thay thế tệp posixrules.

Ngắn gọn, đây là sự khác biệt giữa những viết tắt và các tên đầy đủ: các viết tắt luôn đại diện cho một sự bù trừ cố định từ UTC, trong khi hầu hết các tên đầy đủ ngụ ý một qui định thời gian địa phương về tiết kiệm ánh sáng ban ngày, và vì thế có 2 bù trừ có khả năng theo UTC.

Bạn có thể lo rằng tính năng vùng thời gian dạng POSIX có thể âm thầm dẫn tới việc chấp nhận các đầu vào giả tạo, vì không có sự kiểm tra về tính hợp lý của các từ viết tắt vùng thời gian. Ví dụ, SET TIMEZONE TO FOOBAR0 sẽ làm việc, làm cho hệ thống sử dụng một cách có hiệu quả hơn là sự viết tắt kỳ dị khác thường theo UTC.

Một vấn đề khác phải nhớ trong đầu là theo các tên vùng thời gian POSIX, các phần bù dương được sử dụng cho các vị trí ở phía tây của Greenwich. Ở những nơi khác, PostgreSQL tuân theo qui ước chuẩn ISO – 8601 mà các phần bù vùng thời gian dương là phía đông của Greenwich.

Trong mọi trường hợp, các tên vùng thời gian được thừa nhận là phân biệt chữ hoa và chữ thường. (Đây là sự thay đổi với các phiên bản PostgreSQL trước 8.2, nó từng là phân biệt chữ hoa và chữ thường trong một số ngữ cảnh này nhưng lại không trong các ngữ cảnh khác). Các tên đầy đủ và các tên viết tắt đều không được gắn cứng vào máy chủ; chúng có được từ các tệp cấu hình được lưu trữ

trong .../share/timezone/ và .../share/timezonesets/ của thư mục cài đặt (xem Phần B.3).

Tham số cấu hình vùng thời gian có thể được thiết lập trong tệp postgresql.conf, hoặc trong bất kỳ cách thức tiêu chuẩn nào khác được mô tả trong Chương 18. Cũng có vài cách thức đặc biệt để thiết lập nó:

- Nếu vùng thời gian không được chỉ định trong postgresql.conf hoặc như một lựa chọn dòng lệnh của một máy chủ, thì máy chủ đó sẽ cố gắng sử dụng giá trị biến môi trường TZ như là vùng thời gian mặc định. Nếu TZ không được định nghĩa hoặc không phải là bất kỳ tên nào của vùng thời gian được biết tới đối với PostgreSQL, thì máy chủ sẽ cố gắng xác định vùng thời gian mặc định của hệ điều hành bằng việc kiểm tra hành vi của hàm localtime() của thư viện C. Vùng thời gian mặc định được lựa chọn như là sự trùng khớp gần nhất trong các vùng thời gian của PostgreSQL. (Các qui tắc đó cũng được sử dụng để chọn giá trị mặc định của log_timezone, nếu không được chỉ định).
- Lệnh SET TIME ZONE của SQL thiết lập vùng thời gian cho phiên làm việc. Đây là một lựa chọn thay thế nói SET TIMEZONE TO với một cú pháp tương thích đặc tả của SQL.
- Biến môi trường PGTZ được libpq các máy trạm sử dụng để gửi đi một lệnh SET TIME ZONE tới máy chủ khi được kết nối.

8.5.4. Đầu vào khoảng

Các giá trị khoảng interval có thể được viết bằng việc sử dụng cú pháp rườm rà sau:

[@] quantity unit [quantity unit...] [direction]

trong đó quantity là một số (có khả năng có ký hiệu); unit là microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium (micro giây, milli giây, giây, phút, giờ, ngày, tuần, tháng, năm, thập niên, thế kỷ, ngàn năm), hoặc những chữ viết tắt hoặc số nhiều của các đơn vị đó; direction có thể là ago hoặc rỗng. Ký hiệu @ (ở) là nhiễu tùy chọn. Số lượng các đơn vị khác được bổ sung vào một cách ẩn với việc tính tới các dấu hiệu phù hợp. Từ ago phủ định tất cả các trường đó. Cú pháp này cũng được sử dụng cho đầu ra của khoảng, nếu IntervalStyle được thiết lập cho postgres_verbose.

Số lượng các ngày, giờ, phút và giây có thể được chỉ định mà không có những đánh dấu đơn vị rõ ràng. Ví dụ, '1 12:59:10' được đọc y hệt như '1 ngày 12 giờ 59 phút 10 giây'. Hơn nữa, một sự kết hợp của các năm và tháng có thể được chỉ định với một dấu gạch nối; ví dụ '200-10' được đọc y hệt như '200 năm 10 tháng'. (Các dạng ngắn gọn đó trong thực tế chỉ là các dạng được tiêu chuẩn SQL cho phép, và được sử dụng cho đầu ra khi IntervalStyle được thiết lập theo sql_standard).

Các giá trị khoảng cũng có thể được viết như các vùng thời gian theo ISO 8601, bằng việc sử dụng hoặc "định dạng với các bộ chỉ định" của tiêu chuẩn ở phần 4.4.3.2 hoặc "định dạng lựa chọn thay thế" của phần 4.4.3.3. Định dạng với các bộ chỉ định trông giống thế này:

P quantity unit [quantity unit ...] [T [quantity unit ...]]

Chuỗi đó phải bắt đầu với một chữ P, và có thể có cả chữ T mà giới thiệu các đơn vị thời gian của ngày time-of-day. Những chữ viết tắt đơn vị có sẵn được đưa ra trong Bảng 8-16. Các đơn vị có thể bị bỏ qua, và có thể được chỉ định theo bất kỳ trật tự nào, nhưng các đơn vị nhỏ hơn so với một ngày phải xuất hiện sau chữ T. Đặc biệt, ý nghĩa của chữ M phụ thuộc vào việc liệu nó là đứng trước hay sau chữ T.

Bảng 8-16. Các chữ viết tắt đơn vị khoảng theo ISO 8601

Viết tắt	Y	M	W	D	H	M	S
Nghĩa tiếng Anh	Years	Months	Weeks	Days	Hours	Minutes	Seconds
Nghĩa tiếng Việt	Năm	Tháng	Tuần	Ngày	Giờ	Phút	Giây

Theo định dạng lựa chọn thay thế:

```
P [ years-months-days ] [ T hours:minutes:seconds ]
```

chuỗi đó phải bắt đầu bằng chữ P, và một chữ T tách các phần ngày tháng và thời gian của khoảng đó. Các giá trị được đưa ra như là các số tương tự với ngày tháng theo ISO 8601.

Khi viết một hằng khoảng với một đặc tả các *trường fields*, hoặc khi chỉ định một chuỗi cho một cột khoảng mà đã được xác định với một đặc tả *fields*, ý nghĩa của số lượng không được đánh dấu phụ thuộc vào *fields*. Ví dụ INTERVAL '1' YEAR được đọc như là 1 năm, trong khi INTERVAL '1' có nghĩa là 1 giây. Hơn nữa, các giá trị trường "ở bên phải" của trường có ý nghĩa nhất được đặc tả fields cho phép sẽ bị loại bỏ một cách âm thầm. Ví dụ, viết INTERVAL '1 day 2:03:04' HOUR TO MINUTE sẽ có kết quả loại bỏ trường giây, nhưng không loại bỏ trường ngày.

Theo tiêu chuẩn SQL thì tất cả các trường của một giá trị khoảng phải có dấu y hệt, nên một dấu âm ở đầu sẽ áp dụng cho tất cả các trường; ví dụ dấu âm trong hằng khoảng '-1 2:03:04' sẽ áp dụng cho cả các ngày và các phần giờ/phút/giây. PostgreSQL cho phép các trường có các dấu khác nhau, và theo truyền thống đối xử với từng trường theo sự trình bày theo ngữ cảnh như có ký hiệu một cách độc lập, sao cho phần giờ/phút/giây được xem là dương trong ví dụ này. Nếu IntervalStyle được thiết lập cho sql_standard thì một dấu ở đầu được coi là sẽ áp dụng cho tất cả các trường (mà chỉ khi không có dấu bổ sung nào xuất hiện). Nếu không thì diễn giải theo truyền thống của PostgreSQL sẽ được sử dụng. Để tránh sự nhầm lẫn, được khuyến cáo gắn một dấu rõ ràng cho từng trường nếu có bất kỳ trường nào là âm.

Về nội bộ, các giá trị interval được lưu trữ như là các tháng, ngày và giây. Điều này được làm vì số ngày trong tháng là khác nhau, và một ngày có thể có 23 hoặc 25 giờ nếu sự chỉnh sửa thời gian tiết kiệm ánh sáng ban ngày có liên quan. Các trường tháng và ngày là số nguyên, trong khi trường giây có thể là phân số. Vì các khoảng thường được tạo ra từ các chuỗi hằng hoặc từ phép trừ timestamp, nên phương pháp lưu trữ này làm việc tốt trong hầu hết các trường hợp. Các hàm justify_days và justify_hours là sẵn sàng cho việc chỉnh sửa ngày và giờ mà tràn khỏi dải thông thường của chúng.

Ở định dạng đầu vào dài dòng, và trong một số trường của các định dạng đầu vào gọn chặt hơn, các giá trị trường có thể có các phần thập phân; ví dụ '1.5 week' hoặc '01:02:03.45'. Đầu vào như vậy

được chuyển đổi thành số lượng phù hợp của các tháng, ngày và giây để lưu trữ. Khi điều này có thể gây ra một số thập phân của các tháng hoặc ngày, phần thập phân được bổ sung vào các trường theo trật tự thấp hơn bằng việc sử dụng các yếu tố chuyển đổi 1 tháng = 30 ngày và 1 ngày = 24 giờ. Ví dụ '1.5 month' ('1.5 tháng') trở thành 1 tháng và 15 ngày. Chỉ các giây sẽ luôn được chỉ ra như là phần lẻ ở đầu ra.

Bảng 8-17 chỉ một số ví dụ đầu vào interval hợp lệ.

Bảng 8-17. Đầu vào khoảng

Ví dụ	Mô tả
1-2	Định dạng tiêu chuẩn SQL: 1 năm 2 tháng
3 4:05:06	Định dạng tiêu chuẩn SQL: 3 ngày 4 giờ 5 phút 6 giây
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	Định dạng PostgreSQL truyền thống: 1 năm 2 tháng 3 ngày 4 giờ 5 phút 6 giây
P1Y2M3DT4H5M6S	ISO 8601 “định dạng có bộ chỉ định”: ý nghĩa hệt như trên
P0001-02-03T04:05:06	ISO 8601 “định dạng tùy chọn”: ý nghĩa hệt như trên

8.5.5. Đầu ra khoảng

Định dạng đầu ra của dạng khoảng có thể được thiết lập về một trong 4 dạng sql_standard, postgres, postgres_verbose hoặc iso_8601, bằng việc sử dụng lệnh SET intervalstyle. Mặc định là định dạng postgres. Bảng 8-18 chỉ các ví dụ của từng dạng đầu ra.

Dạng sql_standard tạo đầu ra mà tuân thủ với đặc tả tiêu chuẩn SQL cho các chuỗi hằng khoảng, nếu giá trị khoảng đáp ứng được những hạn chế của tiêu chuẩn (hoặc chỉ năm-tháng hoặc chỉ ngày-thời gian, không pha trộn các thành phần dương và âm). Nếu không đầu ra sẽ trông giống như một chuỗi hằng tiêu chuẩn năm-tháng đi theo sau là một chuỗi hằng ngày-thời gian, với các dấu rõ ràng được thêm vào để tránh nhầm lẫn các khoảng dấu bị pha trộn.

Đầu ra dạng postgres khớp với đầu ra các phiên bản PostgreSQL trước 8.4 khi mà tham số DateStyle được thiết lập theo ISO.

Đầu ra của dạng postgres_verbose khớp với đầu ra của các phiên bản PostgreSQL trước 8.4 khi tham số DateStyle từng được thiết lập cho đầu ra không theo ISO.

Đầu ra của dạng iso_8601 khớp với “định dạng có các bộ chỉ định” được mô tả ở phần 4.4.3.2 của tiêu chuẩn ISO 8601.

Bảng 8-18 Các ví dụ dạng đầu ra khoảng

Đặc tả dạng	Khoảng năm-tháng	Khoảng ngày-thời gian	Khoảng pha trộn
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06
postgres	1 năm 2 tháng	3 ngày 04:05:06	-1 năm -2 tháng +3 ngày -04:05:06
postgres_verbose @	1 năm 2 tháng	@ 3 ngày 4 giờ 5 phút 6 giây	@ 1 năm 2 tháng -3 ngày 4 giờ 5 phút 6 giây trước đó

Đặc tả dạng	Khoảng năm-tháng	Khoảng ngày-thời gian	Khoảng pha trộn
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.5.6. Chi tiết bên trong

PostgreSQL sử dụng ngày Julian cho tất cả các tính toán ngày tháng/thời gian. Điều này có đặc điểm hữu dụng đối với các ngày tính toán đúng từ năm 4713 BC cho tới xa trong tương lai, bằng việc sử dụng giả thiết rằng độ dài của năm là 365.2425 ngày.

Những chuyển đổi ngày trước thế kỷ 19 làm cho việc đọc là thú vị, nhưng không nhất quán đủ để đảm bảo việc lập trình trong trình điều khiển ngày tháng/thời gian.

8.6. Dạng boolean

PostgreSQL cung cấp dạng boolean theo tiêu chuẩn SQL; xem Bảng 8-19. Dạng boolean có thể có 1 trong chỉ 2 trạng thái: “true” (“đúng”) hoặc “false” (“sai”). Trạng thái thứ 3, “unknown” (“không biết”), được thể hiện bằng giá trị null theo SQL.

Bảng 8-19. Dạng dữ liệu boolean

Tên	Kích cỡ lưu trữ	Mô tả
boolean	1 byte	trạng thái đúng hoặc sai

Các giá trị hằng hợp lệ đối với trạng thái “true” là:

```
TRUE
't'
'true'
'y'
'yes'
'on'
'1'
```

Đối với trạng thái “false”, các giá trị sau đây có thể được sử dụng:

```
FALSE
'f'
'false'
'n'
'no'
'off'
'0'
```

Dấu trắng ở đầu hoặc ở sau sẽ bị bỏ qua, và không phân biệt chữ hoa và chữ thường. Các từ khóa TRUE và FALSE được ưu tiên (tuân theo SQL) sử dụng.

Ví dụ 8-2 chỉ ra rằng các giá trị boolean là các đầu ra sử dụng các ký tự t và f.

Ví dụ 8-2. Sử dụng dạng boolean

```
CREATE TABLE test1 (a boolean, b text);
```

```
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;

 a | b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;

 a | b
---+---------
 t | sic est
```

8.7. Các dạng đánh số

Các dạng đánh số (enum) là các dạng dữ liệu bao gồm một tập hợp các giá trị tĩnh, được sắp xếp theo trật tự. Chúng là tương đương với các dạng enum được hỗ trợ trong một số ngôn ngữ lập trình. Một ví dụ dạng enum có thể là các ngày của tuần, hoặc một tập hợp các giá trị trạng thái cho một mẩu dữ liệu.

8.7.1. Khai báo các dạng đánh số

Các dạng đánh số được tạo ra bằng việc sử dụng lệnh tạo dạng CREATE TYPE, ví dụ:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Một khi được tạo ra, dạng enum có thể được sử dụng trong các định nghĩa bảng hoặc hàm giống hệt như bất kỳ dạng nào khác:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
        name text,
        current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';

name | current_mood
--------+--------------
 Moe | happy
(1 row)
```

8.7.2. Xếp thứ tự

Việc xếp thứ tự các giá trị trong một dạng enum là trật tự theo đó các giá trị được liệt kê khi dạng đó đã được tạo ra. Tất cả các toán tử so sánh tiêu chuẩn và các hàm tổng hợp có liên quan được hỗ trợ cho enum. Ví dụ:

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
```

```
name | current_mood
--------+--------------
Moe   | happy
Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
name | current_mood
--------+--------------
Curly  | ok
Moe    | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);

name
-------
Larry
(1 row)
```

8.7.3. An toàn dạng

Mỗi dạng dữ liệu liệt kê là tách bạch nhau và không thể được so sánh với các dạng được liệt kê khác. Xem ví dụ này:

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
        num_weeks integer,
        happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR: invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
        WHERE person.current_mood = holidays.happiness;
ERROR: operator does not exist: mood = happiness
```

Nếu bạn thực sự cần làm thứ gì đó giống điều đó, thì bạn có thể hoặc viết một toán tử tùy biến hoặc thêm các cast rõ ràng vào truy vấn của bạn:

```
SELECT person.name, holidays.num_weeks FROM person, holidays
WHERE person.current_mood::text = holidays.happiness::text;
name | num_weeks
--------+-----------
  Moe | 4
(1 row)
```

8.7.4. Các chi tiết triển khai

Một giá trị enum chiếm 4 byte trên đĩa. Độ dài của một nhãn văn bản giá trị enum được giới hạn bằng thiết lập NAMEDATALEN được biên dịch trong PostgreSQL; theo xây dựng tiêu chuẩn thì điều này có nghĩa là nhiều nhất 63 byte.

Các nhãn enum phân biệt chữ hoa và chữ thường, nên 'happy' là không giống với 'HAPPY'. Ký tự

trắng trong các nhãn cũng là có ý nghĩa.

Các bản dịch từ các giá trị enum nội bộ sang các nhãn văn bản được giữ trong catalog hệ thống pg_enum. Việc truy vấn catalog này trực tiếp có thể là hữu dụng.

8.8. Dạng địa lý

Các dạng dữ liệu địa lý thể hiện các đối tượng không gian 2 chiều. Bảng 8-20 chỉ ra các dạng địa lý có sẵn trong PostgreSQL. Dạng cơ bản nhất, điểm, tạo nên cơ sở cho tất cả các dạng khác.

Bảng 8-20. Các dạng địa lý

Tên	Kích cỡ lưu trữ	Trình bày	Mô tả
point (điểm)	16 bytes	Điểm trên một mặt phẳng	(x,y)
line (đường thẳng)	32 bytes	Đường vô tận (được triển khai không đầy đủ)	((x1,y1),(x2,y2))
lseg (đoạn thẳng)	32 bytes	Đoạn thẳng có giới hạn	((x1,y1),(x2,y2))
box (hộp)	32 bytes	Hộp chữ nhật	((x1,y1),(x2,y2))
path (đường)	16+16n bytes	Đường khép kín (tương tự như đa giác)	((x1,y1),...)
path (đường)	16+16n bytes	Đường mở	[(x1,y1),...]
polygon (đa giác)	40+16n bytes	Đa giác (tương tự như đường khép kín)	((x1,y1),...)
circle (đường tròn)	24 bytes	Đường tròn	<(x,y),r> (tâm và bán kính)

Một tập hợp giàu có các hàm và toán tử là sẵn có để thực thi các hoạt động địa lý khác nhau như chia tỷ lệ, tịnh tiến, xoay và xác định các điểm giao cắt. Chúng được giải thích trong Phần 9.11.

8.8.1. Điểm

Các điểm là khối xây dựng 2 chiều cơ bản cho các dạng địa lý. Các giá trị dạng point được chỉ định bằng việc sử dụng các cú pháp sau:

```
( x , y )
  x , y
```

trong đó x và y là các tọa độ tương ứng, như các số có các dấu thập phân.

Các điểm là đầu ra có sử dụng cú pháp đầu tiên.

8.8.2. Các đoạn thẳng

Các đoạn thẳng (lseg) được trình bày theo cặp các điểm. Các giá trị dạng lseg được chỉ định bằng việc sử dụng bất kỳ cú pháp nào sau đây:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1 , x2 , y2
```

trong đó (x1,y1) và (x2,y2) là các điểm cuối của đoạn thẳng đó.

Các đoạn thẳng là đầu ra có sử dụng cú pháp đầu.

8.8.3. Các hộp

Các hộp được thể hiện bằng cặp các điểm mà chúng nằm ở các góc đối diện của hộp đó. Các giá trị dạng box được chỉ định bằng việc sử dụng bất kỳ cú pháp nào sau đây:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1 , x2 , y2
```

trong đó (x1,y1) và (x2,y2) là bất kỳ 2 góc đối diện nào của hộp đó.

Các hộp là các kết quả đầu ra có sử dụng cú pháp thứ 2.

Bất kỳ 2 góc đối diện nhau nào cũng có thể được cung cấp ở đầu vào, nhưng các giá trị sẽ được tái lập trật tự như cần thiết để lưu trữ các góc phải trên và trái dưới, theo trật tự đó.

8.8.4. Đường

Các đường được các danh sách các điểm kết nối thể hiện. Các đường có thể là mở, nơi mà các điểm đầu và cuối trong danh sách được xem là không nối với nhau, hoặc đóng, nơi mà các điểm đầu và cuối được thấy có nối với nhau.

Các giá trị dạng path được chỉ định bằng việc sử dụng bất kỳ cú pháp nào sau đây:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
    ( x1 , y1 , ... , xn , yn )
      x1 , y1 , ... , xn , yn
```

trong đó các điểm là các điểm cuối của các đoạn thẳng tạo nên đường đó. Các ngoặc vuông ([]) chỉ một đường mở, trong khi các ngoặc đơn (()) chỉ một đường khép kín. Khi các dấu ngoặc đơn vòng ngoài cùng bị bỏ qua, như trong các cú pháp thứ 3 và 5, thì một đường khép kín được giả thiết.

Các đường là đầu ra có sử dụng cú pháp 1 hoặc 2 một cách phù hợp.

8.8.5. Đa giác

Các đa giác được các danh sách các điểm (các đỉnh của đa giác) thể hiện. Các đa giác rất giống với các đường khép kín, nhưng được lưu giữ khác và có tập các thủ tục hỗ trợ riêng của chúng.

Các giá trị dạng polygon được chỉ định bằng việc sử dụng bất kỳ cú pháp nào sau đây:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
    ( x1 , y1 , ... , xn , yn )
      x1 , y1 , ... , xn , yn
```

trong đó các điểm là các điểm cuối của các đoạn thẳng tạo nên đường bao của đa giác đó.

Các đa giác là đầu ra có sử dụng cú pháp số 1.

8.8.6. Đường tròn

Các đường tròn được một điểm tâm và bán kính thể hiện. Các giá trị dạng circle được chỉ định bằng việc sử dụng bất kỳ cú pháp nào sau đây:

```
< ( x , y ) , r >
 ( ( x , y ) , r )
   ( x , y ) , r
     x , y , r
```

trong đó (x,y) là tâm và r là bán kính của đường tròn.

Các đường tròn là đầu ra có sử dụng cú pháp số 1.

8.9. Dạng địa chỉ mạng

PostgreSQL đưa ra các dạng dữ liệu để lưu trữ các địa chỉ IPv4, IPv6 và MAC, như được nêu trong Bảng 8-21. Là tốt hơn để sử dụng các dạng đó thay vì các dạng văn bản thô để lưu trữ các địa chỉ mạng, vì các dạng đó đưa ra việc kiểm tra lỗi đầu vào và các toán tử và hàm chuyên biệt (xem Phần 9.12).

Bảng 8-21. Các dạng địa chỉ mạng

Tên	Kích cỡ lưu trữ	Mô tả
cidr	7 hoặc 19 bytes	Các mạng IPv4 và IPv6
inet	7 hoặc 19 bytes	Các máy chủ và các mạng IPv4 và IPv6
macaddr	6 bytes	Các địa chỉ MAC

Khi sắp xếp các dạng dữ liệu inet hoặc cidr, các địa chỉ IPv4 sẽ luôn sắp xếp trước các địa chỉ IPv6, bao gồm cả các địa chỉ IPv4 được gói hoặc được ánh xạ tới các địa chỉ IPv6, như ::10.2.3.4 hoặc ::ffff:10.4.3.2.

8.9.1. inet

Dạng inet giữ một địa chỉ máy chủ IPv4 hoặc IPv6, và mạng con (subnet) của nó một cách tùy chọn, tất cả trong một trường. Mạng con đó được thể hiện bằng số các bit địa chỉ mạng thể hiện trong địa chỉ máy chủ đó ("mặt nạ mạng" ["netmask"]). Nếu mặt nạ mạng là 32 bit và địa chỉ là IPv4, thì giá trị đó không chỉ một mạng con, mà chỉ là một máy chủ (host) duy nhất. Trong IPv6, độ dài địa chỉ là 128 bit, nên 128 bit chỉ định một địa chỉ máy chủ duy nhất. Lưu ý là nếu bạn muốn chấp nhận chỉ các mạng, thì bạn nên sử dụng dạng cidr hơn là dạng inet.

Định dạng đầu vào cho dạng này là address/y trong đó address là một địa chỉ IPv4 hoặc IPv6 và y là số các bit trong mặt nạ mạng đó. Nếu phần /y không có, thì mặt nạ mạng là 32 bit cho IPv4 và 128 bit cho IPv6, nên giá trị đó chỉ thể hiện một máy chủ host duy nhất. Để hiển thị, phần /y được nén nếu mặt nạ mạng chỉ định một máy chủ host duy nhất.

8.9.2. cidr

Dạng cidr giữ một đặc tả mạng IPv4 hoặc IPv6. Các định dạng đầu vào và đầu ra tuân theo các qui ước Định tuyến Miền Internet Không có lớp (Classless Internet Domain Routing). Định dạng đó cho việc chỉ định các mạng là address/y trong đó address là mạng được thể hiện như một địa chỉ IPv4 hoặc IPv6, và y là số các bit trong mặt nạ mạng. Nếu y bị bỏ qua, thì nó được tính bằng việc sử dụng những giả thiết từ hệ thống đánh số mạng đủ lớp cũ hơn, ngoại trừ nó sẽ ít nhất là đủ lớn để bao gồm tất cả các byte được viết ở đầu vào. Là một lỗi để chỉ định một địa chỉ mạng mà có các bit được thiết lập ở bên phải của mặt nạ mạng được chỉ định.

Bảng 8-22 chỉ một số ví dụ.

Bảng 8-22. Các ví dụ đầu vào dạng cidr

Đầu vào cidr	Đầu ra cidr	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1	/2102081:4f8:3:ba:2e0:81ff:fe22:d1f1	/2102081:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. inet so với cidr

Sự khác biệt cơ bản giữa các dạng dữ liệu inet và cidr là inet chấp nhận các giá trị với các bit không bằng 0 ở bên phải của mặt nạ mạng, trong khi cidr thì không.

> **Mẹo**: Nếu bạn không thích định dạng đầu ra đối với các giá trị inet hoặc cidr, hãy thử các hàm host, text và abbrev.

8.9.4. macaddr

Dạng macaddr lưu trữ các địa chỉ MAC, được biết, ví dụ, từ các địa chỉ phần cứng card mạng

Ethernet (dù các địa chỉ MAC được sử dụng cho cả những mục đích khác nữa). Đầu vào được chấp nhận trong các định dạng sau:

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08002b010203'
```

Các ví dụ đó tất cả có thể chỉ định cùng y hệt địa chỉ. Chữ hoa và chữ thường được chấp nhận cho các ký tự số từ a tới f. Đầu ra luôn là mẫu được chỉ ra đầu tiên.

Tiêu chuẩn 802-2001 của IEEE chỉ định mẫu được chỉ ra thứ 2 (với các dấu nối) như mẫu hợp chuẩn cho các địa chỉ MAC, và chỉ định mẫu đầu tiên (với các dấu 2 chấm) như là ký hiệu bit nghịch đảo, sao cho 08-00-2b-01-02-03 = 01:00:4D:08:04:0C. Qui ước này ngày nay bị bỏ qua một cách rộng rãi, và chỉ phù hợp cho các giao thức mạng đã lỗi thời (như Token Ring). PostgreSQL không đưa ra điều khoản nào cho sự nghịch đảo các bit, và tất cả các định dạng được chấp nhận sử dụng trật tự LSB hợp chuẩn.

4 định dạng đầu vào còn lại không phải là một phần của bất kỳ tiêu chuẩn nào.

8.10. Dạng chuỗi bit

Các chuỗi bit là các chuỗi các số 1 và 0. Chúng có thể được sử dụng để lưu trữ hoặc trực quan hóa các mặt nạ bit. Có 2 dạng bit theo SQL: bit(n) và bit varying(n), trong đó bit n là số nguyên dương.

Dữ liệu dạng bit phải khớp chính xác với độ dài n; sẽ là một lỗi cố gắng để lưu trữ các chuỗi bit ngắn hơn hoặc dài hơn. Dữ liệu bit varying là độ dài biến phụ thuộc vào độ dài cực đại n; các chuỗi dài hơn sẽ bị từ chối. Việc viết bit mà không có một độ dài là tương đương với bit(1), trong khi bit varying không có chỉ định độ dài có nghĩa là độ dài không giới hạn.

> **Lưu ý**: Nếu một người rõ ràng đưa ra một giá trị chuỗi bit cho bit(n), thì nó sẽ bị cắt bớt hoặc được đệm thêm số 0 vào bên phải để trở thành chính xác n bit, không nảy sinh một lỗi nào. Tương tự, nếu một người đưa ra rõ ràng một giá trị chuỗi bit cho bit varying(n), thì nó sẽ bị cắt bớt ở bên phải nếu nó lớn hơn n bit.

Hãy tham chiếu tới Phần 4.1.2.5 để có thông tin về cú pháp của các hằng chuỗi bit. Các toán tử logic về bit và các hàm điều khiển chuỗi là có sẵn; xem Phần 9.6.

Ví dụ 8-3. Sử dụng các dạng chuỗi bit

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
ERROR: bit string length 2 does not match type bit(3)
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

  a | b
-----+-------
101 |  00
100 | 101
```

Một giá trị chuỗi bit đòi hỏi 1 byte cho từng nhóm 8 bit, cộng 5 hoặc 8 byte tổng thể phụ thuộc vào độ dài của chuỗi đó (nhưng các giá trị độ dài có thể được nén hoặc được dịch chuyển vượt ra khỏi dòng, như được giải thích ở Phần 8.3 cho các chuỗi ký tự).

8.11. Dạng tìm kiếm văn bản

PostgreSQL cung cấp 2 dạng dữ liệu được thiết kế để hỗ trợ tìm kiếm toàn văn, nó là hoạt động tìm kiếm qua một bộ sưu tập các tài liệu ngôn ngữ tự nhiên để định vị các tài liệu mà trùng khớp tốt nhất với một truy vấn. Dạng tsvector thể hiện một tài liệu ở dạng được tối ưu hóa cho tìm kiếm văn bản; dạng tsquery thể hiện tương tự như một truy vấn văn bản. Chương 12 đưa ra một giải thích chi tiết cơ sở này, và Phần 9.13 tóm tắt các hàm và toán tử có liên quan.

8.11.1. tsvector

Một giá trị tsvector là một danh sách được sắp xếp các từ vị khác biệt nhau, chúng là các từ đã được bình thường hóa để trộn các biến thể khác nhau của cùng từ y hệt đó (xem Chương 12 để có các chi tiết). Việc sắp xếp và loại bỏ đúp bản được thực hiện tự động ở đầu vào, như trong ví dụ này:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;

                    tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

Để thể hiện các từ vị với các dấu trắng hoặc các dấu chấm ngắt, hãy đưa chúng vào các dấu nháy:

```
SELECT $$the lexeme ' ' contains spaces$$::tsvector;

                  tsvector
-------------------------------------------
 ' ' 'contains' 'lexeme' 'spaces' 'the'
```

(Chúng ta sử dụng các hằng chuỗi được đưa vào các dấu $ trong ví dụ này và ví dụ tiếp sau để tránh sự nhầm lẫn phải đúp bản các dấu nháy trong các hằng đó). Các dấu chéo ngược và dấu nháy nhúng phải được đúp bản:

```
SELECT $$the lexeme 'Joe"s' contains a quote$$::tsvector;

                    tsvector
-------------------------------------------------
 'Joe"s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Một cách tùy chọn, các *vị trí* số nguyên có thể được gắn vào các từ vị:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

Một vị trí thường chỉ ra vị trí từ nguồn gốc trong tài liệu. Thông tin vị trí có thể được sử dụng cho việc *xếp hạng sự tiệm cận*. Các giá trị vị trí có thể trải từ 1 tới 16383; các số lớn hơn được thiết lập âm thầm tới 16383. Các vị trí đúp bản đối với cùng từ vị sẽ bị loại bỏ.

Các từ vị có các vị trí sau đó có thể được gắn nhãn với một trọng số, nó có thể là A, B, C hoặc D. D

là mặc định và vì thế không được hiển thị ở đầu ra:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;

tsvector
----------------------------
'a':1A 'cat':5 'fat':2B,4C
```

Các trọng số thường được sử dụng để phản ánh cấu trúc tài liệu, ví dụ bằng việc tạo các từ đầu đề khác với các từ của thân tài liệu. Các hàm xếp hạng tìm kiếm văn bản có thể chỉ định các ưu tiên khác nhau cho các dấu trọng số khác nhau.

Điều quan trọng phải hiểu rằng bản thân dạng tsvector không thực hiện bất kỳ sự bình thường hóa nào; nó giả thiết các từ được đưa ra được bình thường hóa phù hợp cho ứng dụng. Ví dụ:

```
select 'The Fat Rats'::tsvector;

tsvector
--------------------
'Fat' 'Rats' 'The'
```

Đối với các ứng dụng tìm kiếm văn bản tiếng Anh thì các từ ở trên có thể được xem là không được bình thường hóa, nhưng tsvector không quan tâm. Văn bản tài liệu thô thường sẽ được chuyển qua to_tsvector để bình thường hóa các từ phù hợp cho việc tìm kiếm:

```
SELECT to_tsvector('english', 'The Fat Rats');

to_tsvector
-----------------
'fat':2 'rat':3
```

Một lần nữa, xem Chương 12 để có thêm chi tiết.

8.11.2. tsquery

Một giá trị tsquery lưu trữ các từ vị sẽ được tìm kiếm, và kết hợp chúng với các toàn tử Boolean & (AND), | (OR) và ! (NOT). Các dấu ngoặc đơn có thể được sử dụng để ép tạo nhóm các toán tử:

```
SELECT 'fat & rat'::tsquery;

tsquery
---------------
'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;

tsquery
---------------------------
'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;

tsquery
------------------------
'fat' & 'rat' & !'cat'
```

Không có các dấu ngoặc đơn, ! (NOT) ràng buộc chặt chẽ nhất, và & (AND) ràng buộc chặt chẽ hơn

so với | (OR).

Một cách tùy chọn, các từ vị trong một tsquery có thể được gắn nhãn với một hoặc nhiều hơn các ký tự trọng số, hạn chế chúng trùng khớp chỉ các từ vị của tsvector với các trọng số trùng khớp:

```
SELECT 'fat:ab & cat'::tsquery;

tsquery
------------------
'fat':AB & 'cat'
```

Hơn nữa, các từ vị trong một tsquery có thể được gắn nhãn bằng * để chỉ định sự khớp tiền tố:

```
SELECT 'super:*'::tsquery;

tsquery
-----------
'super':*
```

Truy vấn này sẽ khớp với bất kỳ từ nào trong một tsvector mà bắt đầu với "super".

Các qui tắc tạo dấu ngoặc cho các từ vị là y hệt như được mô tả trước đó cho các từ vị trong tsvector; và, như với tsvector, bất kỳ sự bình thường hóa các từ nào theo yêu cầu cũng phải được thực hiện trước khi chuyển đổi sang dạng tsquery.

Hàm to_tsquery là thuận tiện cho việc thực thi sự bình thường hóa như vậy:

```
SELECT to_tsquery('Fat:ab & Cats');

to_tsquery
------------------
'fat':AB & 'cat'
130
```

8.12. Dạng UUID

Dạng dữ liệu uuid lưu trữ các mã định danh duy nhất vạn năng - UUID (Universally Unique Identifier) như được RFC 4122, ISO/IEC 9834-8:2005, và các tiêu chuẩn liên quan xác định. (Một số hệ thống tham chiếu tới dạng dữ liệu này như một mã định danh duy nhất tổng thể, hoặc GUID [Global Unique Identifier]). Mã định danh này là một lượng 128 bit được một thuật toán sinh ra và được chọn để làm cho nó không có chắc rằng mã định danh y hệt sẽ được bất kỳ ai khác sinh ra trong vũ trụ được biết tới bằng việc sử dụng thuật toán y hệt. Vì thế, đối với các hệ thống phân tán, các mã định danh đó đưa ra một đảm bảo về tính duy nhất tốt hơn so với các bộ sinh tuần tự, chúng chỉ là duy nhất trong một cơ sở dữ liệu đơn nhất.

Một UUID được viết như một sự tuần tự của các ký tự số hệ 16 chữ thường, trong vài nhóm được cách nhau bằng dấu nối, đặc biệt một nhóm 8 ký tự số theo sau là 3 nhóm với 4 ký tự số rồi lại theo sau là 1 nhóm với 12 ký tự số, tổng cộng 32 ký tự số đại diện cho 128 bit. Một ví dụ của một UUID ở dạng tiêu chuẩn này là:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

PostgreSQL cũng chấp nhận các mẫu lựa chọn thay thế sau cho đầu vào: sử dụng các ký tự số chữ hoa, định dạng tiêu chuẩn trong các dấu ngoặc nhọn, bỏ qua một số dấu gạch ngang, bổ sung thêm

một dấu gạch ngang sau bất kỳ nhóm 4 ký tự số nào. Các ví dụ là:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

Đầu ra luôn ở trong dạng tiêu chuẩn.

PostgreSQL cung cấp các hàm lưu trữ và so sánh cho các UUID, nhưng cơ sở dữ liệu lõi không bao gồm bất kỳ hàm nào cho việc sinh các UUID, vì không thuật toán đơn nhất nào phù hợp tốt cho mọi ứng dụng. Module contrib contrib/uuid-ossp cung cấp các hàm triển khai vài thuật toán tiêu chuẩn. Như một sự lựa chọn, các UUID có thể được sinh ra từ các ứng dụng hoặc các thư viện khác bị thu hồi thông qua một hàm ở phía máy chủ.

8.13. Dạng XML

Dạng dữ liệu xml có thể được sử dụng để lưu trữ các dữ liệu XML. Ưu điểm của nó so với việc lưu trữ các dữ liệu XML trong một trường văn bản là nó kiểm tra các giá trị đầu vào về sự thiết lập tốt, và có các hàm hỗ trợ để tiến hành các hoạt động dạng an toàn trong nó; xem Phần 9.14. Việc sử dụng dữ liệu này đòi hỏi sự cài đặt phải được xây dựng với configure –with-libxml.

Dạng xml có thể lưu trữ "các tài liệu" được thiết lập tốt, được được tiêu chuẩn XML định nghĩa, cũng như các phân mảnh "nội dung" được xác định bằng sự đưa ra XMLDecl? content trong tiêu chuẩn XML. Đại thể, điều này có nghĩa là các phân mảnh nội dung có thể có hơn một nút ký tự hoặc yếu tố mức đỉnh. Biểu thức xmlvalue IS DOCUMENT có thể được sử dụng để đánh giá liệu một giá trị nhất định xml có là một tài liệu đầy đủ hay chỉ là một phân đoạn nội dung.

8.13.1. Tạo giá trị XML

Để tạo một giá trị dạng xml từ các dữ liệu ký tự, hãy sử dụng hàm xmlparse:

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Các ví dụ:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

Trong khi đây chỉ là cách để biến đổi các chuỗi ký tự thành các giá trị XML theo tiêu chuẩn SQL, thì các cú pháp đặc thù PostgreSQL là:

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

cũng có thể được sử dụng.

Dạng xml không kiểm tra hợp lệ các giá trị đầu vào đối với một khai báo dạng tài liệu - DTD (Document Type Declaration), thậm chí khi giá trị đầu vào chỉ định một DTD. Hiện cũng không có sự hỗ trợ nào được xây dựng sẵn cho việc kiểm tra hợp lệ đối với các ngôn ngữ sơ đồ khác XML như XML Schema.

Thao tác ngược lại, việc tạo ra một giá trị chuỗi ký tự từ xml, hãy sử dụng hàm xmlserialize:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

dạng có thể là character, character varying hoặc text (hoặc một tên hiệu cho một trong số đó). Một lần nữa, theo tiêu chuẩn SQL, đây chỉ là cách để biến đổi giữa dạng xml và các dạng ký tự, nhưng PostgreSQL cũng cho phép bạn đơn giản đưa ra giá trị đó.

Khi một giá trị chuỗi ký tự được đưa ra hoặc từ dạng xml mà không đi qua XMLPARSE hoặc XMLSERIALIZE, một cách tương ứng, thì sự lựa chọn của DOCUMENT so với CONTENT được xác định bằng tham số cấu hình phiên làm việc của "lựa chọn XML", nó có thể được thiết lập bằng việc sử dụng lệnh tiêu chuẩn:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

hoặc cú pháp giống PostgreSQL hơn

```
SET xmloption TO { DOCUMENT | CONTENT };
```

Mặc định là CONTENT, nên tất cả các dạng dữ liệu XML đều được phép.

> **Lưu ý**: Với việc thiết lập lựa chọn XML mặc định, bạn không thể trực tiếp đưa ra các chuỗi ký tự tới dạng xml nếu chúng chứa một khai báo dạng tài liệu, vì định nghĩa đoạn nội dung XML không chấp nhận chúng. Nếu bạn cần làm thế, hoặc hãy sử dụng XMLPARSE hoặc thay đổi lựa chọn XML.

8.13.2. Điều khiển việc mã hóa

Cần phải cẩn thận khi làm việc với nhiều việc mã hóa ký tự trên máy trạm, máy chủ và trong các dữ liệu XML được truyền qua chúng. Khi sử dụng chế độ văn bản để truyền các truy vấn tới máy chủ và các kết quả truy vấn tới máy trạm (mà là ở chế độ thông thường), PostgreSQL chuyển đổi tất cả các dữ liệu ký tự được truyền giữa máy trạm và máy chủ và ngược lại tới việc mã hóa ký tự của đầu tương ứng; xem Phần 22.2. Điều này bao gồm các trình diễn chuỗi các giá trị XML, như trong các ví dụ ở trên. Điều này có thể thông thường có nghĩa là các khai báo việc mã hóa có trong các dữ liệu XML có thể trở thành không hợp lệ khi các dữ liệu ký tự được chuyển đổi sang việc mã hóa trong khi truyền giữa máy trạm và máy chủ, vì khai báo mã hóa được nhúng không được thay đổi. Để vượt qua hành vi này, các khai báo mã hóa có trong các chuỗi ký tự được trình diễn cho đầu vào đối với dạng xml sẽ bị bỏ qua, và nội dung được giả thiết sẽ ở trong mã hóa máy chủ hiện hành. Hệ quả là, đối với việc xử lý đúng, các chuỗi ký tự của dữ liệu XML phải được gửi từ máy trạm đang trong mã hóa máy trạm hiện hành. Là trách nhiệm của máy trạm để hoặc biến đổi các tài liệu sang mã hóa máy trạm trước khi gửi chúng tới máy chủ, hoặc tinh chỉnh việc mã hóa máy trạm một cách phù hợp. Ở đầu ra, các giá trị của dạng xml sẽ không có khai báo mã hóa, và các máy trạm sẽ giả thiết tất cả các dữ liệu là trong việc mã hóa máy trạm hiện hành.

Khi sử dụng chế độ nhị phân để truyền các tham số truy vấn tới máy chủ và các kết quả truy vấn ngược về máy trạm, không biến đổi tập hợp ký tự nào được thực hiện, nên tình huống là khác. Trong trường hợp này, một khai báo mã hóa trong dữ liệu XML sẽ được quan sát thấy, và nếu nó là

không có, thì dữ liệu sẽ được giả thiết ở UTF-8 (như được tiêu chuẩn XML yêu cầu; lưu ý là PostgreSQL không hỗ trợ UTF-16). Ở đầu ra, dữ liệu sẽ có một khai báo mã hóa chỉ định mã hóa máy trạm, trừ phi mã hóa máy trạm là UTF-8, trong trường hợp đó nó sẽ bị bỏ qua.

Không cần phải nói, việc xử lý dữ liệu XML với PostgreSQL sẽ ít lỗi hỏng và hiệu quả hơn nếu mã hóa dữ liệu XML, mã hóa máy trạm và mã hóa máy chủ là y hệt nhau. Vì dữ liệu XML được xử lý nội bộ trong UTF-8, các tính toán sẽ hiệu quả nhất nếu mã hóa máy chủ là UTF-8.

Thận trọng

Một số hàm có liên quan tới XML có thể không làm việc hoàn toàn trong các dữ liệu không phải ASCII khi mã hóa máy chủ không phải là UTF-8. Điều này được biết tới như là một vấn đề đặc biệt đối với `xpath()`.

8.13.3. Truy cập các giá trị XML

Dạng dữ liệu `xml` là không bình thường theo đó nó không cung cấp bất kỳ toán tử so sánh nào. Điều này là vì không có toán tử so sánh hữu dụng vạn năng và được xác định tốt nào cho dữ liệu XML. Một hậu quả của điều này là bạn không thể truy xuất các hàng bằng việc so sánh một cột `xml` với một giá trị tìm kiếm. Các giá trị XML vì thế sẽ thường được đi với một trường khóa riêng rẽ như một ID. Một giải pháp lựa chọn thay thế cho việc so sánh các giá trị XML là phải biến đổi chúng thành các chuỗi ký tự trước, nhưng lưu ý là sự so sánh các chuỗi ký tự có ít điều phải làm với một phương pháp so sánh XML hữu dụng.

Vì không có các toán tử so sánh cho dạng dữ liệu `xml`, không có khả năng để tạo một chỉ số trực tiếp trong một cột dạng này. Nếu các tìm kiếm nhanh trong dữ liệu XML là mong muốn, thì những khắc phục có khả năng bổ sung bằng việc đưa ra biểu thức cho một dạng chuỗi ký tự và đánh chỉ số nó, hoặc đánh chỉ số một biểu thức Xpath. Tất nhiên, truy vấn thực sự có thể phải được tinh chỉnh để tìm kiếm theo biểu thức được đánh chỉ số đó.

Chức năng tìm kiếm văn bản trong PostgreSQL cũng có thể được sử dụng để tăng nhanh các tìm kiếm tài liệu toàn văn các dữ liệu XML. Tuy nhiên, sự hỗ trợ xử lý cần thiết còn chưa sẵn sàng trong phân phối PostgreSQL.

8.14. Mảng

PostgreSQL cho phép các cột của một bảng được xác định như là các mảng đa chiều độ dài biến thiên. Các mảng dạng cơ bản được xây dựng sẵn hoặc do người sử dụng định nghĩa, hoặc dạng tổng hợp có thể được tạo ra. Các mảng của các miền còn chưa được hỗ trợ.

8.14.1. Khai báo dạng mảng

Để minh họa sử dụng các dạng mảng, chúng ta tạo bảng này:

```
CREATE TABLE sal_emp (
        name text,
        pay_by_quarter integer[],
        schedule text[][]
);
```

Như được chỉ ra, một dạng dữ liệu mảng được đặt tên bằng việc nối thêm các dấu ngoặc vuông ([]) vào tên các phần tử mảng của dạng dữ liệu đó. Lệnh ở trên sẽ tạo một bảng có tên là sal_emp với một cột dạng text (name), một mảng một chiều dạng số nguyên integer (pay_by_quarter - trả theo quý), nó thể hiện lương nhân viên theo quý, và một mảng văn bản (schedule - lịch) 2 chiều, nó thể hiện lịch tuần của nhân viên đó.

Cú pháp đối với CREATE TABLE cho phép kích cỡ chính xác của các mảng sẽ được chỉ định, ví dụ:

```
CREATE TABLE tictactoe (
        squares integer[3][3]
);
```

Tuy nhiên, sự triển khai hiện hành bỏ qua bất kỳ các giới hạn kích cỡ mảng được cung cấp nào, nghĩa là, hành vi đó là y hệt như đối với các mảng độ dài không được chỉ định.

Triển khai hiện hành cũng không ép tuân thủ số chiều được khai báo. Các mảng của một dạng yếu tố đặc biệt tất cả được xem xét là dạng y hệt, bất chấp kích cỡ hoặc số chiều. Vì thế, việc khai báo kích cỡ hoặc số lượng chiều của mảng trong lệnh tạo bảng CREATE TABLE đơn giản là tài liệu; nó không tác động tới hành vi trong thời gian thực.

Một cú pháp lựa chọn thay thế tuân thủ tiêu chuẩn SQL bằng việc sử dụng từ khóa mảng ARRAY, có thể được sử dụng cho các mảng 1 chiều. pay_by_quarter có thể đã được định nghĩa như là:

```
pay_by_quarter integer ARRAY[4],
```

Hoặc, nếu không kích cỡ mảng nào được chỉ định:

```
pay_by_quarter integer ARRAY,
```

Tuy nhiên, như trước, PostgreSQL không ép tuân thủ hạn chế kích cỡ trong bất kỳ trường hợp nào.

8.14.2. Đầu vào giá trị mảng

Để viết một giá trị mảng như một hằng theo nghĩa đen, hãy đưa các giá trị phần tử vảo trong các dấu ngoặc nhọn và tách bạch chúng bằng dấu phẩy. (Nếu bạn biết C, thì điều này không giống cú pháp của C vì các cấu trúc khởi tạo). Bạn có thể đặt các dấu ngoặc kép xung quanh bất kỳ giá trị phần tử mảng nào, và phải làm thế nếu nó có chứa các dấu phẩy hoặc các dấu ngoặc nhọn. (Chi tiết hơn xuất hiện ở bên dưới). Vì thế, định dạng chung của một hằng mảng là như sau:

```
'{ val1 delim val2 delim ... }'
```

trong đó delim là ký tự phân cách cho dạng này, như được ghi lại trong khoản đầu vào its pg_type. Trong số các dạng dữ liệu tiêu chuẩn được cung cấp trong phân phối của PostgreSQL, tất cả đều sử dụng dấu phẩy (,), ngoại trừ dạng hộp box mà nó sử dụng một dấu chấm phẩy (;). Từng giá trị val hoặc là một hằng số của dạng phần tử mảng, hoặc là một mảng phụ. Ví dụ của một hằng mảng là:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Hằng này là một mảng 2 chiều, 3x3, bao gồm 3 mảng phụ các số nguyên.

Để thiết lập một phần tử của một hằng mảng thành NULL, hãy viết NULL cho giá trị phần tử đó. (bất kỳ chữ hoa hay chữ thường cho NULL đó). Nếu bạn muốn một giá trị chuỗi thực sự "NULL", thì bạn phải đặt xung quanh nó các dấu ngoặc kép.

(Các dạng hằng mảng thực sự chỉ là một trường hợp đặc biệt của các hằng dạng chung được thảo luận trong Phần 4.1.2.7. Hằng ban đầu được đối xử như một chuỗi và được truyền tới thủ tục biến đổi đầu vào của mảng. Một đặc tả dạng rõ ràng có thể là cần thiết).

Bây giờ chúng ta có thể chỉ ra một số lệnh chèn INSERT:

```
INSERT INTO sal_emp
        VALUES ('Bill',
        '{10000, 10000, 10000, 10000}',
        '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
        VALUES ('Carol',
        '{20000, 25000, 25000, 25000}',
        '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

Kết quả của 2 lệnh chèn trước trông giống thế này:

```
SELECT * FROM sal_emp;

name   | pay_by_quarter                  | schedule
-------+---------------------------------+-------------------------------------------------------------
Bill   | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
Carol  | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

Các mảng đa chiều phải có sự trùng khớp mở rộng cho từng chiều. Một sự không trùng khớp gây ra một lỗi, ví dụ:

```
INSERT INTO sal_emp
        VALUES ('Bill',
        '{10000, 10000, 10000, 10000}',
        '{{"meeting", "lunch"}, {"meeting"}}');
ERROR: multidimensional arrays must have array expressions with matching dimensions
```

Cú pháp cấu trúc mảng ARRAY cũng có thể được sử dụng:

```
INSERT INTO sal_emp
        VALUES ('Bill',
        ARRAY[10000, 10000, 10000, 10000],
        ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
        VALUES ('Carol',
        ARRAY[20000, 25000, 25000, 25000],
        ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

Lưu ý rằng các phần tử mảng là các hằng hoặc các biểu thức SQL thông thường; ví dụ, các hằng chuỗi sẽ nằm trong các dấu ngoặc đơn, thay vì trong các dấu ngoặc kép như chúng có thể ở trong một hằng mảng. Cú pháp cấu trúc mảng ARRAY được thảo luận chi tiết hơn trong Phần 4.2.11.

8.14.3. Truy cập mảng

Bây giờ, chúng ta có thể chạy một số truy vấn trong bảng. Trước hết, chúng ta chỉ ra cách để truy cập một phần tử duy nhất của một mảng. Truy vấn này truy xuất các tên của các nhân viên mà lương (pay) của họ bị thay đổi trong quý II:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

name
-------
Carol
(1 row)
```

Các số chỉ số dưới của mảng được viết bên trong các dấu ngoặc vuông. Mặc định thì PostgreSQL sử dụng một qui ước đánh số một cơ sở cho các mảng, đó là, một mảng của n phần tử bắt đầu bằng array[1] và kết thúc bằng array[n].

Truy vấn này truy xuất lương quý III của tất cả các nhân viên:

```
SELECT pay_by_quarter[3] FROM sal_emp;

pay_by_quarter
----------------
10000
25000
(2 rows)
```

Chúng ta cũng có thể truy cập tùy ý các lát cắt chữ nhật của một mảng, hoặc mảng phụ. Một lát cắt mảng được biểu hiện bằng việc viết lower-bound:upper-bound cho một hoặc nhiều chiều của mảng. Ví dụ, truy vấn này truy xuất khoản đầu tiên trong thời gian biểu của Bill cho 2 ngày đầu của tuần:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

schedule
------------------------
{{meeting},{training}}
(1 row)
```

Nếu bất kỳ chiều nào được viết như một lát cắt, nghĩa là bao gồm một dấu 2 chấm, thì tất cả các chiều được đối xử như là các lát cắt. Bất kỳ chiều nào mà chỉ có một số duy nhất (không có dấu 2 chấm) sẽ được đối xử như đang từ 1 tới số được chỉ định. Ví dụ, [2] được đối xử như là [1:2], như trong ví dụ này:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

schedule
-------------------------------------------
{{meeting,lunch},{training,presentation}}
(1 row)
```

Để tránh lẫn lộn với trường hợp không có lát cắt, tốt nhất hãy sử dụng cú pháp lát cắt cho tất cả các chiều, nghĩa là, [1:2][1:1], not [2][1:1].

Một biểu thức chỉ số dưới sẽ trả về null nếu hoặc bản thân mảng hoặc bất kỳ biểu thức chỉ số dưới nào mà là null. Hơn nữa, null được trả về nếu một chỉ số dưới nằm ngoài các giới hạn mảng (trường hợp này không sinh ra lỗi). Ví dụ, nếu thời gian biểu hiện hành có các chiều [1:3][1:2] thì việc tham

chiếu tới schedule[3][3] sẽ là NULL. Tương tự, một tham chiếu mảng với số các chỉ số dưới sai cũng cho null hơn là một lỗi.

Một biểu thức lát cắt mảng cũng có thể cho null nếu bản thân mảng đó hoặc bất kỳ biểu thức chỉ số dưới nào là null. Tuy nhiên, trong các trường hợp khác như việc chọn một lát cắt mảng mà hoàn toàn nằm ngoài các giới hạn mảng hiện hành, một biểu thức lát cắt có một mảng rỗng (0 chiều) thay vì null. (Điều này không khớp với hành vi không lát cắt và được thực hiện vì các lý do lịch sử). Nếu lát cắt được yêu cầu một phần chèn lên các giới hạn của mảng, thì nó âm thầm bị giảm xuống tới chỉ vùng chồng lấn đó thay vì trả về null.

Các chiều hiện hành của bất kỳ giá trị mảng nào cũng có thể được truy xuất với hàm array_dims:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

array_dims
------------
[1:2][1:2]
(1 row)
```

array_dims tạo ra một kết quả văn bản text, nó là tiện lợi cho mọi người để đọc nhưng có lẽ không tiện cho các chương trình. Các chiều cũng có thể được truy xuất bằng array_upper and array_lower, nó trả về giới hạn cao hơn hoặc thấp hơn một chiều mảng được chỉ định, một cách tương ứng:

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

array_upper
-------------
2
(1 row)
```

array_length sẽ trả về độ dài của chiều mảng được chỉ định:

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

array_length
--------------
2
(1 row)
```

8.14.4. Sửa đổi mảng

Một giá trị mảng có thể được thay thế hoàn toàn:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
        WHERE name = 'Carol';
```

Hoặc sử dụng cú pháp biểu thức mảng ARRAY:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
        WHERE name = 'Carol';
```

Một mảng cũng có thể được cập nhật ở một phần tử duy nhất:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
        WHERE name = 'Bill';
```

hoặc được cập nhật trong một lát cắt:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
        WHERE name = 'Carol';
```

Một giá trị mảng được lưu trữ có thể được mở rộng bằng việc chỉ định tới các phần tử còn chưa hiện diện. Bất kỳ vị trí nào giữa chúng trước đó hiện diện và các phần tử được chỉ định mới cũng sẽ được điền bằng null. Ví dụ, nếu mảng myarray hiện có 4 phần tử, thì nó sẽ có 6 phần tử sau một cập nhật mà chỉ định tới myarray[6]; myarray[5] sẽ có chứa null. Hiện hành, sự mở rộng theo cách thức này chỉ được phép đối với các mảng 1 chiều, không cho các mảng nhiều chiều.

Sự chỉ định có chỉ số dưới cho phép tạo các mảng không sử dụng các chỉ số dưới dựa vào 1. Ví dụ một người có thể chỉ định tới myarray[-2:7] để tạo một mảng với các giá trị chỉ số dưới từ -2 tới 7.

Các giá trị mảng mới cũng có thể được cấu trúc bằng việc sử dụng toán tử ghép nối, ||:

```
SELECT ARRAY[1,2] || ARRAY[3,4];

?column?
-----------
{1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];

?column?
---------------------
{{5,6},{1,2},{3,4}}
(1 row)
```

Toán tử ghép nối cho phép một phần tử duy nhất sẽ được đẩy vào đầu hoặc cuối một mảng 1 chiều. Nó cũng chấp nhận 2 mảng N chiều, hoặc một mảng N chiều và một mảng N+1 chiều.

Khi một phần tử duy nhất được đẩy vào hoặc đầu hoặc cuối của một mảng 1 chiều, thì kết quả là một mảng với chỉ số dưới giới hạn thấp hơn y hệt như toán hạng của mảng. Ví dụ:

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);

array_dims
------------
[0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);

array_dims
------------
[1:3]
(1 row)
```

Khi 2 mảng với số chiều bằng nhau được ghép nối, thì kết quả vẫn là chỉ số dưới giới hạn thấp hơn của chiều bên ngoài toán hạng bên tay trái. Kết quả là một mảng bao gồm mọi phần tử của toán hạng bên tay trái rồi theo sau là mọi phần tử của toán hạng bên tay phải. Ví dụ:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
array_dims
------------
[1:5]
(1 row)
```

```
SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);

array_dims
------------
[1:5][1:2]
(1 row)
```

Khi một mảng N chiều được đẩy vào đầu hoặc cuối của một mảng N+1 chiều, thì kết quả là tương tự như trường hợp mảng các phần tử ở trên. Từng mảng phụ N chiều về cơ bản là phần tử của chiều bên ngoài mảng N+1 chiều. ví dụ:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);

array_dims
------------
[1:3][1:2]
(1 row)
```

Một mảng cũng có thể được cấu thành bằng việc sử dụng các hàm array_prepend, array_append hoặc array_cat. 2 hàm đầu chỉ hỗ trợ các mảng 1 chiều, nhưng array_cat hỗ trợ các mảng nhiều chiều. Lưu ý là toán tử ghép nối được thảo luận ở trên được ưu tiên hơn là sử dụng trực tiếp các hàm đó.

Trên thực tế, các hàm đó trước hết tồn tại để sử dụng trong việc triển khai toán tử ghép nối. Tuy nhiên, chúng có thể là trực tiếp hữu dụng trong việc tạo các ghép nối do người sử dụng định nghĩa. Một số ví dụ:

```
SELECT array_prepend(1, ARRAY[2,3]);

array_prepend
---------------
{1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);

array_append
--------------
{1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);

array_cat
-----------
{1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);

array_cat
---------------------
{{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);

array_cat
---------------------
{{5,6},{1,2},{3,4}}
```

8.14.5. Tìm kiếm trong mảng

Để tìm một giá trị trong một mảng, mỗi giá trị phải được kiểm tra. Điều này có thể được thực hiện bằng tay, nếu bạn biết kích cỡ mảng. Ví dụ:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

Tuy nhiên, điều này nhanh chóng trở nên nặng nhọc cho những mảng lớn, và không hữu dụng nếu không biết kích cỡ của mảng. Một phương pháp thay thế được mô tả trong Phần 9.21. Truy vấn ở trên có thể được thay thế bằng:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

Hơn nữa, bạn có thể thấy các hàng nơi mà mảng có tất cả các giá trị bằng 10.000 với:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

Như một lựa chọn, hàm generate_subscripts có thể được sử dụng. Ví dụ:

```
SELECT * FROM
    (SELECT pay_by_quarter,
        generate_subscripts(pay_by_quarter, 1) AS s
    FROM sal_emp) AS foo
WHERE pay_by_quarter[s] = 10000;
```

Hàm này được mô tả trong Bảng 9-46.

> **Mẹo**: Các mảng không phải là các tập hợp; việc tìm kiếm các phần tử mảng đặc biệt có thể là dấu hiệu của sự thiết kế sai cơ sở dữ liệu. Hãy xem xét sử dụng một bảng riêng biệt với một hàng cho từng khoản mà có thể là một phần tử mảng. Điều này sẽ là dễ dàng hơn để tìm kiếm, và có khả năng co dãn tốt hơn đối với một số lượng lớn các phần tử.

8.14.6. Cú pháp đầu vào và đầu ra của mảng

Miêu tả bên ngoài bằng văn bản của một giá trị mảng bao gồm các khoản được biên dịch theo các qui tắc qui ước của I/O cho dạng phần tử mảng, cộng với sự trang trí chỉ ra cấu trúc mảng. Sự trang trí đó bao gồm các dấu ngoặc nhọn ({and}) xung quanh giá trị mảng đó cộng với các ký tự phân cách giữa các khoản liền kề nhau. Ký tự phân cách thường là một dấu phẩy (,) nhưng có thể là thứ gì đó khác: được xác định bằng việc thiết lập typdelim cho dạng phần tử mảng. Trong số các dạng dữ liệu tiêu chuẩn được phân phối PostgreSQL cung cấp, tất cả sử dụng một dấu phẩy, ngoại trừ dạng hộp box, nó sử dụng một dấu chấm phẩy (;). Trong một mảng nhiều chiều, từng chiều (hàng, mặt phẳng, khối lập phương, …) có mức các dấu ngoặc nhọn của riêng nó, và các dấu phân cách phải được viết giữa các thực thể có cùng mức trong các dấu ngoặc nhọn liền kề nhau.

Thủ tục đầu vào của mảng sẽ đặt các dấu ngoặc kép xung quanh các giá trị phần tử nếu chúng là các chuỗi rỗng, chứa các dấu ngoặc nhọn, các ký tự phân cách, các dấu ngoặc kép, các dấu chéo ngược hoặc các ký tự trống, hoặc khớp với từ NULL. Các dấu ngoặc kép và các dấu chéo ngược được nhúng trong các giá trị phần tử sẽ có thoát bằng dấu chéo ngược. Đối với các dạng dữ liệu số thì an toàn để giả thiết rằng các dấu ngoặc đơn sẽ không bao giờ xuất hiện, nhưng đối với các dạng dữ liệu

văn bản thì sẽ được chuẩn bị để vượt qua được hoặc sự tồn tại hoặc sự thiếu vắng các dấu ngoặc.

Mặc định, giá trị chỉ số giới hạn thấp hơn của một chiều của mảng được thiết lập về 1. Để trình bày các mảng với các giới hạn thấp hơn khác, thì các dãy chỉ số dưới của mảng có thể được chỉ định rõ ràng trước khi viết các nội dung của mảng. Sự trang trí này bao gồm các dấu ngoặc vuông ([]) xung quanh từng giới hạn thấp hơn và cao hơn chiều của mảng, với một ký tự dấu phân cách dấu 2 chấm (:) ở giữa. Sự trang trí chiều của mảng được theo sau với một dấu bằng (=). Ví dụ:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
        FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

e1 | e2
----+----
  1 | 6
(1 row)
```

Thủ tục đầu ra của mảng sẽ bao gồm các chiều rõ ràng trong kết quả của nó chỉ khi có một hoặc nhiều hơn các giới hạn khác với nhau.

Nếu giá trị được viết cho một phần tử là NULL (trong bất kỳ trường hợp nào), thì phần tử đó sẽ được lấy là NULL. Sự hiện diện của bất kỳ dấu ngoặc hoặc dấu chéo ngược nào cũng vô hiệu hóa điều này và cho phép giá trị chuỗi của hằng "NULL" sẽ được đưa vào. Hơn nữa, vì tính tương thích ngược với các phiên bản PostgreSQL trước 8.2, tham số cấu hình array_nulls có thể sẽ được tắt off để ép nhận NULL như là NULL.

Như được chỉ ra trước đó, khi viết một giá trị mảng thì bạn có thể sử dụng các dấu ngoặc kép xung quanh bất kỳ phần tử mảng riêng rẽ nào. Bạn phải làm thế nếu giá trị phần tử đó có thể nếu khác sẽ làm lúng túng cho trình phân tích giá trị mảng. Ví dụ, các phần tử chứa các dấu ngoặc nhọn, các dấu phẩy (hoặc ký tự phân cách dạng dữ liệu), các dấu ngoặc kép, các dấu chéo ngược hoặc các ký tự trống ở trước hoặc sau phải được đưa vào trong ngoặc kép. Các chuỗi rỗng và chuỗi trùng từ NULL cũng phải được đưa vào ngoặc. Để đặt một dấu ngoặc kép hoặc dấu chéo ngược vào trong một giá trị phần tử mảng trong dấu ngoặc, hãy sử dụng cú pháp chuỗi thoát và đặt trước nó một dấu chéo ngược. Như một sự lựa chọn, bạn có thể tránh các dấu ngoặc và sử dụng thoát dấu chéo ngược để bảo vệ tất cả các ký tự dữ liệu mà có thể nếu khác sẽ được lấy như là cú pháp mảng.

Bạn có thể thêm dấu trắng trước một dấu ngoặc nhọn trái hoặc sau một dấu ngoặc nhọn phải. Bạn cũng có thể thêm dấu trắng trước hoặc sau bất kỳ khoản chuỗi riêng rẽ nào. Trong tất cả các trường hợp đó thì dấu trắng sẽ bị bỏ qua. Tuy nhiên, dấu trắng trong các phần tử trong các dấu ngoặc kép, hoặc được bao quanh cả 2 phía với các ký tự không trắng của một phần tử, sẽ không bị bỏ qua.

> Lưu ý: Hãy nhớ rằng những gì bạn viết trong một lệnh SQL trước hết sẽ được biên dịch như một hằng chuỗi, và sau đó như một mảng. Điều này làm gấp đôi số lượng các dấu chéo ngược bạn cần. Ví dụ, để chèn một giá trị mảng văn bản có chứa một dấu chéo ngược và một dấu ngoặc kép, bạn cần phải viết:
>
> ```
> INSERT ... VALUES (E'{"\\\\","\\""}');
> ```
>
> Trình xử lý chuỗi thoát loại bỏ một mức các dấu chéo ngược, sao cho những gì tới trong

trình phân tích giá trị bất kỳ nào của mảng trong cũng giống như {"\\","\""}. Đổi lại, các chuỗi đối với thủ tục đầu vào dạng dữ liệu văn bản text sẽ trở thành \ và " một cách tương ứng. (Nếu chúng ta từng làm việc với một dạng dữ liệu mà thủ tục đầu vào của nó cũng đối xử với các dấu chéo ngược một cách đặc biệt, ví dụ bytea, thì chúng ta có thể cần tới 8 dấu chéo ngược trong lệnh để có được một dấu chéo ngược trong phần tử mảng được lưu trữ). Việc đưa vào các dấu $ (xem Phần 4.1.2.4) có thể được sử dụng để tránh nhu cầu phải nhân đôi các dấu chéo ngược.

Mẹo: Cú pháp cấu trúc mảng ARRAY (xem Phần 4.2.11) thường dễ hơn để làm việc so với cú pháp hằng mảng khi viết các giá trị mảng trong các lệnh SQL. Trong ARRAY, các giá trị phần tử riêng rẽ được viết theo cách y hệt mà chúng có thể được viết khi không là các thành viên của một mảng.

8.15. Dạng tổng hợp

Một dạng tổng hợp đại diện cho cấu trúc của một hàng hoặc bản ghi; về cơ bản nó chỉ là một danh sách các tên trường và các dạng dữ liệu của chúng. PostgreSQL cho phép các dạng tổng hợp được sử dụng theo nhiều cách thức mà các dạng đơn giản có thể được sử dụng. Ví dụ, một cột của một bảng có thể được khai báo là một dạng tổng hợp.

8.15.1. Khai báo dạng tổng hợp

Đây là 2 ví dụ định nghĩa các dạng tổng hợp:

```
CREATE TYPE complex AS (
        r double precision,
        i double precision
);

CREATE TYPE inventory_item AS (
        name text,
        supplier_id integer,
        price numeric
);
```

Cú pháp có thể so sánh được với lệnh tạo bảng CREATE TABLE, ngoại trừ chỉ các tên và dạng của trường có thể được chỉ định; không ràng buộc nào (như NOT NULL) có thể được đưa vào. Lưu ý là từ khóa AS là cơ bản; không có nó, hệ thống sẽ nghĩ là một dạng khác của lệnh tạo dạng CREATE TYPE, và bạn sẽ có các lỗi cú pháp kỳ lạ.

Khi đã xác định được các dạng, chúng ta có thể sử dụng chúng để tạo các bảng:

```
CREATE TABLE on_hand (
        item inventory_item,
        count integer
);
INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

hoặc các hàm:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
```

```
AS 'SELECT $1.price * $2' LANGUAGE SQL;
SELECT price_extension(item, 10) FROM on_hand;
```

Bất kỳ khi nào bạn tạo một bảng, một dạng tổng hợp cũng được tự động tạo ra, với tên y hệt như bảng đó, để đại diện cho dạng hàng của bảng. Ví dụ, ta đã nói:

```
CREATE TABLE inventory_item (
        name text,
        supplier_id integer REFERENCES suppliers,
        price numeric CHECK (price > 0)
);
```

rồi sau đó dạng tổng hợp y hệt inventory_item được chỉ ra ở trên có thể tới như là một bán thành phẩm, và có thể được sử dụng hệt như ở trên. Tuy nhiên hãy lưu ý một hạn chế quan trọng của triển khai hiện hành: vì không ràng buộc nào có liên quan tới một dạng tổng hợp, các ràng buộc được chỉ ra trong định nghĩa bảng *không áp dụng* được cho các giá trị của dạng tổng hợp bên ngoài bảng đó. (Cách giải quyết một phần là hãy sử dụng các dạng miền như các thành viên của dạng tổng hợp).

8.15.2. Đầu vào giá trị tổng hợp

Để viết một giá trị tổng hợp như một hằng số theo nghĩa đen, hãy đưa các giá trị của trường đó vào các dấu ngoặc đơn và phân cách chúng bằng các dấu phẩy. Bạn có thể đặt các dấu ngoặc kép xung quanh bất kỳ giá trị trường nào, và phải làm thế nếu nó có chứa các dấu phẩy hoặc các dấu ngoặc đơn. (Chi tiết hơn xem bên dưới). Vì thế, định dạng chung của một hằng tổng hợp là như sau:

```
'( val1 , val2 , ... )'
```

Một ví dụ là:

```
'("fuzzy dice",42,1.99)'
```

mà có thể là một giá trị hợp lệ của dạng inventory_item được xác định ở trên. Để làm cho một trường là NULL, hãy không viết các ký tự nào cả trong vị trị của nó trong danh sách. Ví dụ, hằng này chỉ định một NULL cho trường thứ 3:

```
'("fuzzy dice",42,)'
```

Nếu bạn muốn một chuỗi rỗng hơn là NULL, hãy viết các dấu ngoặc kép:

```
'("",42,)'
```

Ở đây trường đầu tiên là một chuỗi rỗng không NULL, trường thứ 3 là NULL.

(Các hằng đó thực sự chỉ là một trường hợp đặc biệt của các hằng dạng chung được thảo luận trong Phần 4.1.2.7. Hằng đó ban đầu được đối xử như là một chuỗi và được truyền tới thủ tục qui ước đầu vào dạng tổng hợp. Một đặc tả dạng rõ ràng có thể là cần thiết).

Cú pháp biểu thức hàng ROW cũng có thể được sử dụng để tạo ra các giá trị tổng hợp. Trong hầu hết các trường hợp điều này được cho là đơn giản hơn để sử dụng so với cú pháp hằng chuỗi vì bạn không phải lo lắng về nhiều lớp các dấu ngoặc. Chúng tôi đã sử dụng phương pháp này ở trên:

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

Từ khóa hàng ROW thực sự là tùy chọn miễn là bạn có nhiều hơn 1 trường trong biểu thức, sao cho chúng có thể đơn giản hóa thành:

('fuzzy dice', 42, 1.99)
('', 42, NULL)

Cú pháp biểu thức hàng ROW được thảo luận chi tiết hơn trong Phần 4.2.12.

8.15.3. Truy cập dạng tổng hợp

Để truy cập một trường của một cột tổng hợp, một người viết một dấu chấm và tên trường đó, giống hệt như việc chọn một trường từ một tên bảng. Trên thực tế, nó rất giống việc chọn từ một tên bảng mà bạn thường phải sử dụng các dấu ngoặc đơn để giữ cho khỏi nhầm cho trình phân tích. Ví dụ, bạn có thể thử chọn một số trường phụ từ bảng ví dụ on_hand của chúng tôi với thứ gì đó giống như:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

Điều này sẽ không làm việc vì tên khoản item được lấy là tên của một bảng, không phải tên của một cột của bảng on_hand, theo các qui định cú pháp của SQL. Bạn phải viết nó như thế này:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

hoặc nếu bạn vẫn cần phải sử dụng tên bảng đó (ví dụ trong một truy vấn nhiều bảng), như:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

Bây giờ đối tượng trong dấu ngoặc đơn được diễn giải trực tiếp như một tham chiếu tới cột item, và sau đó trường con (subfield) có thể được lựa chọn từ đó.

Tương tự các vấn đề cú pháp áp dụng bất kỳ khi nào bạn chọn một trường từ một giá trị tổng hợp. Ví dụ, để lựa chọn một trường từ kết quả của một hàm mà trả về một giá trị tổng hợp, bạn có thể cần viết thứ gì đó giống như:

```
SELECT (my_func(...)).field FROM ...
```

Không có các dấu ngoặc đơn thừa, điều này sẽ làm sinh ra một lỗi cú pháp.

8.15.4. Sửa đổi dạng tổng hợp

Đây là một số ví dụ về cú pháp phù hợp cho việc chèn và cập nhật các cột tổng hợp.

Trước hết, việc chèn hoặc cập nhật toàn bộ một cột:

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));
UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

Ví dụ đầu bỏ qua hàng ROW, ví dụ 2 sử dụng nó; chúng ta có thể thực hiện nó bất kỳ cách nào.

Chúng ta có thể cập nhật một trường phụ riêng rẽ của một cột tổng hợp:

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

Lưu ý ở đây rằng chúng ta không cần (và quả thực không thể) đặt các dấu ngoặc đơn xung quanh tên cột xuất hiện chỉ sau SET, nhưng chúng ta cần các dấu ngoặc đơn khi tham chiếu tới cột y hệt trong biểu thức ở bên phải của dấu bằng.

Và chúng ta cũng có thể chỉ định các trường phụ như là các đích cho lệnh chèn INSERT:

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

Chúng ta đã không cung cấp các giá trị cho tất cả các trường phụ của cột, các trường phụ còn lại có thể được điền với các giá trị null.

8.15.5 Cú pháp đầu và và ra của dạng tổng hợp

Sự trình bày ngoài bằng văn bản của một giá trị tổng hợp bao gồm các khoản sẽ được biên dịch theo các qui tắc qui ước cho các dạng trường riêng rẽ, cộng với sự trang trí mà chỉ ra cấu trúc tổng hợp đó. Trang trí bao gồm các dấu ngoặc đơn (( )) xung quanh toàn bộ giá trị đó, cộng với dấu phẩy (,) giữa các khoản liền kề. Dấu trắng bên ngoài các dấu ngoặc đơn sẽ bị bỏ qua, nhưng bên trong các dấu ngoặc đơn thì nó được xem như là một phần của giá trị trường đó, và có thể có hoặc không phụ thuộc đáng kể vào các qui tắc qui ước đầu vào cho dạng dữ liệu trường. Ví dụ, trong

```
'( 42)'
```

thì dấu trắng sẽ bị bỏ qua nếu dạng của trường đó là số nguyên, nhưng sẽ không bỏ qua nếu dạng trường đó là văn bản text.

Như được chỉ ra trước đó, khi viết một giá trị tổng hợp bạn có thể viết các dấu ngoặc kép xung quanh bất kỳ giá trị trường riêng rẽ nào. Bạn phải làm thế nếu giá trị trường đó có thể nếu khác làm lúng túng cho trình phân tích giá trị tổng hợp. Đặc biệt, các trường có chứa các dấu ngoặc đơn, dấu phẩy, dấu ngoặc kép hoặc dấu chéo ngược phải được đưa vào các dấu ngoặc kép. Để đặt một dấu ngoặc đơn hoặc chéo ngược trong một giá trị trường tổng hợp trong dấu, hãy đặt trước nó một dấu chéo ngược. (Hơn nữa, một cặp dấu ngoặc đơn trong một giá trị trường các dấu ngoặc kép được lấy để đại diện cho một ký tự dấu ngoặc kép, tương tự như các qui tắc cho các dấu ngoặc đơn trong các chuỗi hằng SQL). Như một sự lựa chọn, bạn có thể tránh đưa vào trong các dấu và sử dụng ký tự thoát dấu chéo ngược để bảo vệ tất cả các ký tự dữ liệu mà có thể nếu khác thì sẽ được lấy như là cú pháp tổng hợp.

Một giá trị trường rỗng hoàn toàn (hoàn toàn không có các ký tự giữa các dấu phẩy và các dấu ngoặc đơn) đại diện cho một NULL. Để viết một giá trị mà là chuỗi rỗng hơn là NULL, hãy viết "".

Thủ tục đầu ra tổng hợp sẽ đặt các dấu ngoặc kép xung quanh các giá trị trường nếu chúng là các chuỗi rỗng hoặc có các dấu ngoặc đơn, dấu phẩy, dấu ngoặc kép, dấu chéo ngược, hoặc dấu trống. (Làm như vậy cho dấu trắng là không cơ bản, nhưng giúp cho dễ hiểu). Các dấu ngoặc kép hoặc chéo ngược được nhúng trong các giá trị trường sẽ được nhân đôi.

> **Lưu ý**: Hãy ghi nhớ rằng những gì bạn viết trong một lệnh SQL trước hết sẽ được hiểu như là một hằng chuỗi, và sau đó như một sự tổng hợp. Điều này nhân đôi số các dấu chéo ngược mà bạn cần (giả thiết cú pháp chuỗi thoát được sử dụng). Ví dụ, để chèn một trường văn bản text có chứa một dấu ngoặc kép và một dấu chéo ngược vào một giá trị tổng hợp, bạn cần phải viết:
>
> ```
> INSERT ... VALUES (E'("\\"\\\\")');
> ```

Trình xử lý hằng chuỗi loại bỏ một mức các dấu chéo ngược, sao cho những gì tới trình phân tích giá trị tổng hợp trông giống như ("\"\\"). Đổi lại, chuỗi được dùng cho thủ tục đầu vào dạng dữ liệu văn bản text sẽ trở thành "\. (nếu chúng ta từng làm việc với một dạng dữ liệu mà thủ tục đầu vào của nó cũng đối xử với các dấu chéo ngược một cách đặc biệt, ví dụ bytea, thì chúng ta có thể cần tới 8 dấu chéo ngược trong lệnh để có được một dấu chéo ngược trong trường tổng hợp được lưu trữ). Việc cho vào ngoặc các dấu $ (xem Phần 4.1.2.4) có thể được sử dụng để tránh cần phải nhân đôi các dấu chéo ngược.

Mẹo: Cú pháp cấu trúc hàng ROW thường dễ dàng hơn để làm việc so với cú pháp hằng tổng hợp khi viết các giá trị tổng hợp trong các lệnh SQL. Trong ROW, các giá trị trường riêng rẽ được viết theo cách y hệt như chúng có thể được viết khi không có các thành viên của một sự tổng hợp.

8.16. Dạng mã định danh đối tượng

Mã định danh đối tượng - OID (Object Indentifier) được sử dụng nội bộ trong PostgreSQL như là các khóa chính cho các bảng của các hệ thống khác nhau. Các OID không được thêm vào các bảng do người sử dụng tạo ra, trừ phi WITH OIDS được chỉ định khi bảng được tạo ra, hoặc biến cấu hình default_with_oids được kích hoạt. Dạng oid thể hiện một mã định danh đối tượng. Cũng có vài dạng tên hiệu cho oid: regproc, regprocedure, regoper, regoperator, regclass, regtype, regconfig, và regdictionary. Bảng 8-23 chỉ ra sự tổng quan.

Dạng oid hiện được triển khai như một số nguyên 4 byte không ký. Vì thế, nó không đủ lớn để cung cấp tính độc nhất rộng khắp cơ sở dữ liệu trong các cơ sở dữ liệu lớn, hoặc thậm chí trong các bảng lớn riêng rẽ. Vì thế, sử dụng một cột OID trong bảng do người sử dụng tạo ra như một khóa chính không được khuyến khích. Các OID sử dụng tốt nhất chỉ cho các tham chiếu tới các bảng hệ thống.

Bản thân dạng oid có ít hoạt động ngoài sự so sánh. Tuy nhiên, nó có thể đưa ra cho số nguyên và sau đó được điều khiển bằng việc sử dụng các toán tử số nguyên tiêu chuẩn. (Chú ý về sự lộn xộn ký với không ký có thể nếu bạn làm điều này).

Các dạng tên hiệu OID không có các hoạt động của riêng chúng ngoại trừ các thủ tục đầu vào và đầu ra được chuyên biệt hóa. Các thủ tục đó có khả năng chấp nhận và hiển thị các tên biểu tượng cho các đối tượng hệ thống, hơn là các giá trị cho các đối tượng. Ví dụ, để xem xét các hàng pg_attribute có liên quan tới bảng mytable, một người có thể viết:

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

hơn là

```
SELECT * FROM pg_attribute
    WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

Trong khi bản thân điều đó tất cả không thật là xấu, thì nó vẫn quá được đơn giản hóa. Sự lựa chọn phụ thêm (subselect) tinh vi phức tạp hơn nhiều có thể cần phải chọn OID đúng nếu có nhiều bảng được đặt tên mytable trong các sơ đồ khác. Trình chuyển đổi đầu vào regclass điều khiển sự tra cứu

bảng theo thiết lập đường sơ đồ, và vì thế nó thực hiện "điều đúng đắn" một cách tự động. Tương tự, việc đưa ra một OID của bảng cho regclass là thuận tiện cho hiển thị biểu tượng của một OID số.

Bảng 8-23. Dạng định danh đối tượng

Tên	Tham chiếu	Mô tả	Ví dụ giá trị
oid	any	định danh đối tượng số	564182
regproc	pg_proc	tên hàm	sum
regprocedure	pg_proc	hàm với các dạng đối số	sum(int4)
regoper	pg_operator	tên toán tử	+
regoperator	pg_operator	toán tử với các dạng đối số	*(integer,integer) hoặc -(NONE,integer)
regclass	pg_class	tên quan hệ	pg_type
regtype	pg_type	tên dạng dữ liệu	integer
regconfig	pg_ts_config	cấu hình tìm kiếm văn bản	english
regdictionary	pg_ts_dict	từ điển tìm kiếm văn bản	simple

Tất cả các dạng tên hiệu OID chấp nhận các tên đủ điều kiện theo sơ đồ, và sẽ hiển thị các tên đủ điều kiện theo sơ đồ ở đầu ra nếu đối tượng đó có thể không được tìm thấy trong đường tìm kiếm hiện hành mà không đang có đủ điều kiện. Các dạng tên hiệu regproc và regoper sẽ chỉ chấp nhận các tên đầu vào mà là độc nhất (không bị quá tải), nên chúng được sử dụng hạn chế; sử dụng nhiều nhất regprocedure hoặc regoperator là phù hợp hơn. Đối với regoperator, các toán tử một yếu tố được nhận diện bằng việc viết NONE cho toán hạng không được sử dụng.

Một thuộc tính bổ sung của các dạng tên hiệu OID là sự tạo ra các phụ thuộc. Nếu một hằng số của một trong các đạng đó xuất hiện trong một biểu thức được lưu trữ (như một biểu thức mặc định cột hoặc kiểu nhìn), thì nó tạo ra một sự phụ thuộc vào đối tượng được tham chiếu đó. Ví dụ, nếu một cột có một biểu thức mặc định nextval('my_seq'::regclass), thì PostgreSQL hiểu rằng biểu thức mặc định phụ thuộc vào tuần tự my_seq; hệ thống đó sẽ không để cho sự tuần tự đó bị bỏ đi mà không có việc trước đó loại bỏ biểu thức mặc định đó.

Dạng định dạng khác được hệ thống sử dụng là xid, hoặc định dạng giao dịch (ngắn gọn là xact). Đây là dạng dữ liệu của các cột hệ thống xmin và xmax. Các định danh giao dịch là các lượng 32 bit.

Dạng định danh thứ 3 được hệ thống sử dụng là cid, hoặc định danh lệnh. Đây là dạng dữ liệu của các cột hệ thống cmin và cmax. Định danh lệnh cũng là các lượng 32 bit.

Dạng định danh cuối cùng được hệ thống sử dụng là tid, hoặc định danh bộ (định danh hàng). Đây là dạng dữ liệu của cột hệ thống ctid. Một ID hàng là một cặp (số khối, chỉ số bộ trong khối) mà định danh vị trí vật lý của hàng bên trong bảng của nó.

(Các cột hệ thống được giải thích xa hơn trong Phần 5.4).

8.17. Dạng bí danh

Hệ thống dạng PostgreSQL gồm một số khoản đầu vào có mục đích đặc biệt mà nói chung được gọi là các *dạng bí danh*. Một dạng bí danh không thể được sử dụng như một dạng dữ liệu cột, nhưng nó có thể được sử dụng để khai báo một đối số hàm hoặc dạng kết quả. Từng trong số các dạng bí danh có sẵn được sử dụng trong các tình huống nơi mà hành vi của một hàm không tương xứng một cách đơn giản với việc lấy hoặc trả về một giá trị của một dạng dữ liệu SQL đặc biệt. Bảng 8-24 liệt kê các dạng bí danh đang tồn tại.

Bảng 8-24. Các dạng bí danh

Tên	Mô tả
any	Chỉ rằng một hàm chấp nhận bất kỳ dạng dữ liệu đầu vào nào.
anyarray	Chỉ rằng một hàm chấp nhận bất kỳ dạng dữ liệu mảng nào (xem Phần 32.2.5).
anyelement	Chỉ rằng một hàm chấp nhận bất kỳ dạng dữ liệu nào (xem Phần 32.2.5).
anyenum	Chỉ rằng một hàm chấp nhận bất kỳ dạng dữ liệu enum nào (xem Phần 35.2.5 và Phần 8.7).
anynonarray	Chỉ rằng một hàm chấp nhận bất kỳ dạng dữ liệu không là mảng nào (xem Phần 35.2.5).
cstring	Chỉ rằng một hàm chấp nhận hoặc trả về một chuỗi C được kết thúc bằng null.
internal	Chỉ rằng một hàm chấp nhận hoặc trả về một dạng dữ liệu nội bộ của máy chủ.
language_handler	Một điều khiển lời gọi ngôn ngữ thủ tục được khai báo để trả về language_handler.
record	Nhận diện một hàm trả về một dạng hàng không được xác định.
trigger	Một hàm trigger được khai báo để trả về trigger.
void	Chỉ rằng một hàm trả về không giá trị nào cả.
opaque	Một tên dạng lỗi thời được phục vụ trước đó cho tất cả các mục đích ở trên.

Các hàm được viết mã theo ngôn ngữ C (hoặc được tải lên sẵn hoặc động) có thể được khai báo để chấp nhận hoặc trả về bất kỳ dạng dữ liệu bí danh nào. Điều đó phụ thuộc vào tác giả của hàm để đảm bảo rằng hàm đó sẽ hành xử an toàn khi một dạng bí danh được sử dụng như một dạng đối số.

Các hàm được viết mã trong các ngôn ngữ thủ tục có thể sử dụng các dạng tên hiệu chỉ khi được các ngôn ngữ triển khai của chúng cho phép. Hiện hành các ngôn ngữ thủ tục tất cả đều cấm sử dụng một dạng bí danh như là dạng đối số, và chỉ cho phép void và record như một dạng kết quả (cộng với trigger khi hàm đó được sử dụng như một trigger). Một số cũng hỗ trợ các hàm đa hình bằng việc sử dụng các dạng anyarray, anyelement, anyenum, và anynonarray.

Dạng bí danh internal được sử dụng để khai báo các hàm mà chỉ có nghĩa để được hệ thống cơ sở dữ liệu gọi nội bộ, và không trực tiếp viện tới trong một truy vấn SQL. Nếu một hàm có ít nhất một đối số dạng internal thì nó không thể được gọi từ SQL. Để giữ an toàn dạng của hạn chế này thì điều quan trọng phải tuân theo qui tắc lập trình này: không tạo ra bất kỳ hàm nào mà được khai báo để trả về internal trừ phi nó có ít nhất một đối số internal.

Chương 9. Hàm và toán tử

PostgreSQL đưa ra một số lượng lớn các hàm và toán tử cho các dạng dữ liệu được xây dựng sẵn. Người sử dụng cũng có thể định nghĩa các hàm và toán tử của riêng mình, như được mô tả trong Phần V. Các lệnh psql \df và \do có thể được sử dụng để liệt kê tất cả các hàm và toán tử có sẵn một cách tương ứng.

Nếu bạn có quan tâm về tính khả chuyển thì hãy lưu ý rằng hầu hết các hàm và toán tử được mô tả trong chương này, với ngoại lệ của hầu hết các toán tử số học và so sánh thông thường và một số hàm được lưu ý rõ ràng, sẽ không được tiêu chuẩn SQL xác lập. Một số trong chức năng được mở rộng đó có hiện diện trong các hệ thống quản lý cơ sở dữ liệu SQL khác, và trong nhiều trường hợp chức năng đó là tương thích và nhất quán giữa các triển khai khác nhau. Chương này cũng không phải là toàn diện mọi khía cạnh; các hàm bổ sung thêm sẽ có trong các phần phù hợp của sách chỉ dẫn này.

9.1. Các toán tử logic

Các toán tử logic thông thường là sẵn sàng:

AND
OR
NOT

SQL sử dụng một hệ thống logic 3 giá trị với true (đúng), false (sai) và null (không biết). Hãy quan sát các bảng đúng sau đây:

a	b	a AND b	a OR b
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Các toán tử AND và OR là giao hoán được, đó là, bạn có thể hoán đổi toán hạng bên trái và bên phải mà không ảnh hưởng tới kết quả. Xem Phần 4.2.13 để có thêm thông tin về trật tự đánh giá các biểu thức phụ.

9.2. Các toán tử so sánh

Các toán tử so sánh thông thường là có sẵn, được nêu trong Bảng 9-1.

Bảng 9-1. Các toán tử so sánh

Toán tử	Mô tả
<	nhỏ hơn
>	lớn hơn
<=	nhỏ hơn hoặc bằng
>=	lớn hơn hoặc bằng
=	bằng
<> hoặc !=	không bằng

> **Lưu ý**: Toán tử != được chuyển thành <> trong pha của trình phân tích cú pháp. Không có khả năng để triển khai các toán tử != và <> mà làm các điều khác nhau.

Các toán tử so sánh là có sẵn cho tất cả các dạng dữ liệu phù hợp. Tất cả các toán tử đều là các toán tử nhị phân trả về các giá trị dạng boolean; các biểu thức như 1 < 2 < 3 là không hợp lệ (vì không có toán tử < để so sánh một giá trị Boolean với 3).

Hơn nữa đối với các toán tử so sánh, cấu trúc BETWEEN đặc thù là có sẵn:

```
a BETWEEN x AND y
```

là tương đương với

```
a >= x AND a <= y
```

Lưu ý là BETWEEN đối xử với các dữ liệu đầu cuối như được đưa vào trong dải đó. Còn NOT BETWEEN thực hiện sự so sánh ngược lại:

```
a NOT BETWEEN x AND y
```

là tương đương với

```
a < x OR a > y
```

BETWEEN SYMMETRIC là y hệt như BETWEEN ngoại trừ không có yêu cầu rằng đối số ở bên trái của AND sẽ là ít hơn hoặc bằng đối số ở bên phải. Nếu không phải thế, thì 2 đối số đó tự động được hoán đổi cho nhau, sao cho một dải không rỗng luôn được ngụ ý.

Để kiểm tra liệu một giá trị có hay không là null, hãy sử dụng các cấu trúc:

```
expression IS NULL
expression IS NOT NULL
```

hoặc tương tự, nhưng phi tiêu chuẩn, các cấu trúc:

```
expression ISNULL
expression NOTNULL
```

Hãy *không* viết expression = NULL vì NULL là không "bằng với" NULL. (Giá trị null đại diện cho một

giá trị không được biết, và không rõ liệu 2 giá trị không được biết có bằng nhau hay không). Hành vi này tuân thủ theo tiêu chuẩn SQL.

> **Mẹo**: Một số ứng dụng có thể mong đợi rằng expression = NULL trả về đúng (true) nếu expression ước tính giá trị null. Được khuyến cáo mạnh mẽ rằng các ứng dụng đó sẽ được sửa đổi để tuân thủ với tiêu chuẩn SQL. Tuy nhiên, nếu điều đó không thể được thực hiện thì biến cấu hình transform_null_equals là sẵn sàng. Nếu điều đó được kích hoạt, thì PostgreSQL sẽ biến đổi các mệnh đề x = NULL thành x IS NULL.
>
> **Lưu ý**: Nếu expression là có giá trị hàng, thì IS NULL là đúng (true) khi bản thân biểu thức hàng đó là null hoặc khi tất cả các trường của hàng đó là null, trong khi IS NOT NULL là đúng khi bản thân biểu thứ đó là non-null và tất cả các trường của hàng đó là non-null. Vì hành vi này, IS NULL và IS NOT NULL không luôn trả về các kết quả ngược nhau đối với các biểu thức có giá trị hàng, nghĩa là một biểu thức có giá trị hàng mà có chứa cả các giá trị null và not-null sẽ trả về sai (false) cho cả 2 kiểm thử. Định nghĩa này tuân thủ tiêu chuẩn SQL, và là một thay đổi từ hành vi không nhất quán có trong PostgreSQL các phiên bản trước 8.2.

Các toán tử so sánh thông thường mang lại null (biểu thị "không biết"), không là đúng (true) hoặc sai (false), khi đầu vào là null. Ví dụ, 7 = NULL mang lại null. Khi hành vi này là không phù hợp, hãy sử dụng các cấu trúc IS [NOT] DISTINCT FROM:

expression IS DISTINCT FROM expression
expression IS NOT DISTINCT FROM expression

Đối với các đầu vào non-null, IS DISTINCT FROM là y hệt như toán tử <>. Tuy nhiên, nếu cả 2 đầu vào đều là null thì nó trả về false, và nếu chỉ một đầu vào là null thì nó trả về true. Tương tự, IS NOT DISTINCT FROM là y hệt với = đối với các đầu vào non-null, nhưng nó trả về true khi cả 2 đầu vào là null, và false khi chỉ một đầu vào là null. Vì thế, các cấu trúc đó hành động có hiệu lực như thể null là một giá trị dữ liệu thông thường, chứ không phải là "không biết".

Các giá trị Boolean cũng có thể được kiểm thử bằng việc sử dụng các cấu trúc

expression IS TRUE
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN

Chúng sẽ luôn trả về true hoặc false, không bao giờ trả về một giá trị null, thậm chí khi toán tử là null. Một đầu vào null được đối xử như là giá trị logic "không biết". Lưu ý rằng IS UNKNOWN và IS NOT UNKNOWN có hiệu lực như nhau như IS NULL và IS NOT NULL một cách tương ứng, ngoại trừ là biểu thức đầu vào phải ở dạng Boolean.

9.3. Hàm và toán tử toán học

Các toán tử toán học được cung cấp cho nhiều dạng PostgreSQL. Đối với các dạng không có các qui ước toán học tiêu chuẩn (như các dạng ngày tháng/thời gian – date/time) chúng ta mô tả hành vi thực sự trong các phần tiếp sau.

Bảng 9-2 chỉ ra các toán tử toán học có sẵn.

Bảng 9-2. Các toán tử toán học

Toán tử	Mô tả	Ví dụ	Kết quả
+	cộng	2 + 3	5
-	trừ	2 - 3	-1
*	nhân	2 * 3	6
/	chia (chia số nguyên sẽ cắt bớt kết quả)	4 / 2	2
%	modulo (phần còn lại)	5 % 4	1
^	lũy thừa	2.0 ^ 3.0	8
|/	căn bậc 2	|/ 25.0	5
||/	căn bậc 3	||/ 27.0	3
!	giai thừa	5 !	120
!!	giai thừa (toán tử tiếp đầu ngữ)	!! 5	120
@	giá trị tuyệt đối	@ -5.0	5
&	bitwise	AND 91 & 15	11
|	bitwise	OR 32 | 3	35
#	bitwise	XOR 17 # 5	20
~	bitwise	NOT ~1	-2
<<	bitwise dịch trái	1 << 4	16
>>	bitwise dịch phải	8 >> 2	2

Các toán tử bitwise chỉ làm việc trong các dạng dữ liệu tích phân, trong khi các toán tử khác là sẵn sàng cho tất cả các dạng dữ liệu số. Các toán tử bitwise cũng sẵn sàng cho các dạng chuỗi bit bit và bit varying, như được chỉ ra trong bảng 9-10.

Bảng 9-3 chỉ ra các hàm toán học có sẵn. Trong bảng, dp chỉ độ chính xác gấp đôi double precision. Nhiều trong số các hàm đó được đưa ra ở nhiều dạng với các dạng đối số khác nhau. Ngoại trừ ở những nơi được lưu ý, bất kỳ dạng nào của một hàm được đưa ra cũng trả về dạng dữ liệu y hệt như đối số của nó. Các hàm làm việc với các dữ liệu double precision hầu hết được triển khai trên đỉnh của thư viện C hệ thống chủ (host system); độ chính xác và hành vi trong các trường hợp biên vì thế có thể biến đổi, phụ thuộc vào hệ thống chủ đó.

Bảng 9-3. Các hàm toán học

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
abs(x)	(y hệt như đầu vào)	giá trị tuyệt đối	abs(-17.4)	17.4
cbrt(dp)	dp	căn bậc 3	cbrt(27.0)	3
ceil(dp or numeric)	(y hệt như đầu vào)	số nguyên nhỏ nhất không nhỏ hơn đối số	ceil(-42.8)	-42
ceiling(dp or numeric)	(y hệt như đầu vào)	số nguyên nhỏ nhất không	ceiling(-95.3)	-95

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
		nhỏ hơn đối số (tên hiệu cho ceil)		
degrees(dp)	dp	từ radians sang độ	degrees(0.5)	28.6478897565412
div(y numeric, x numeric)	numeric	thương số số nguyên của y/x	div(9,4)	2
exp(dp or numeric)	(y hệt như đầu vào)	số mũ	exp(1.0)	2.71828182845905
floor(dp or numeric)	(y hệt như đầu vào)	số nguyên lớn nhất không lớn hơn đối số	floor(-42.8)	-43
ln(dp or numeric)	(y hệt như đầu vào)	logarithm tự nhiên	ln(2.0)	0.693147180559945
log(dp or numeric)	(y hệt như đầu vào)	logarithm cơ số 10	log(100.0)	2
log(b numeric, x numeric)	numeric	Logarithm cơ số b	Log(2.0, 64.0)	6.0000000000
mod(y, x)	(y hệt dạng đối số)	phần còn lại của y/x	mod(9,4)	1
pi()	dp	hằng số “π”	pi()	3.14159265358979
power(a dp, b dp)	dp	a lũy thừa b	power(9.0, 3.0)	729
power(a numeric, b numeric)	numeric	a lũy thừa b	Power(9.0, 3.0)	729
radians(dp)	dp	từ độ sang radians	radians(45.0)	0.785398163397448
random()	dp	giá trị ngẫu nhiên trong dãy 0.0 <= x < 1.0	random()	
round(dp or numeric)	(y hệt như đầu vào)	làm tròn tới số nguyên gần nhất	round(42.4)	42
round(v numeric, s int)	numeric	làm tròn tới s dấu thập phân	round(42.4382, 2)	42.44
setseed(dp)	void	đặt hạt giống cho các lời gọi random () tiếp sau (giá trị giữa -1.0 và 1.0, bao gồm 2 số đó)	setseed(0.54823)	
sign(dp or numeric)	(y hệt như đầu vào)	dấu của đối số (-1, 0, +1)	sign(-8.4)	-1
sqrt(dp or numeric)	(y hệt như đầu vào)	căn bậc 2	sqrt(2.0)	1.4142135623731
trunc(dp or numeric)	(y hệt như đầu vào)	cắt ngắn tới 0	trunc(42.8)	42
trunc(v numeric, s int)	numeric	cắt ngắn tới s dấu thập phân	trunc(42.4382, 2)	42.43
width_bucket(op numeric, b1 numeric, b2 numeric, count int)	int	trả về khoảng theo đó toán hạng có thể được chỉ định theo biểu đồ cùng mức với các khoảng tính toán, trong dải b1 tới b2	width_bucket(5.35, 0.024, 10.06, 5)	
width_bucket(op dp, b1 dp, b2 dp, count int)	int	trả về khoảng theo đó toán hạng có thể được chỉ định theo biểu đồ cùng mức với các khoảng tính toán, trong dải b1 tới b2	width_bucket(5.35, 0.024, 10.06, 5)	

Cuối cùng, Bảng 9-4 chỉ ra các hàm lượng giác có sẵn. Tất cả các hàm lượng giác lấy các đối số và trả về các giá trị dạng double precision. Các đối số hàm lượng giác được thể hiện bằng radian. Các hàm ngược trả về các giá trị ở dạng radian. Xem các hàm biến đổi đơn vị radians() và degrees() ở trên.

Bảng 9-4. Các hàm lượng giác

Hàm	acos(x)	asin(x)	atan(x)	atan2(y, x)	cos(x)	cot(x)	sin(x)	tan(x)
Mô tả	ngược với cosine	ngược với sine	ngược với tangent	ngược với tangent của y/x	cosine	cotangent	sine	tangent

9.4. Hàm và toán tử chuỗi

Phần này mô tả các hàm và toán tử cho việc kiểm tra và điều khiển các giá trị chuỗi. Các chuỗi trong ngữ cảnh này bao gồm các giá trị các dạng character, character varying, và text. Trừ phi được lưu ý khác đi, tất cả các hàm được liệt kê bên dưới làm việc trong tất cả các dạng, nhưng là khác nhau về các hiệu ứng tiềm năng của việc chêm vào chỗ trống một cách tự động khi sử dụng dạng character. Một số hàm cũng tồn tại bẩm sinh đối với các dạng chuỗi bit.

SQL xác định một số hàm chuỗi mà sử dụng các từ khóa, thay vì các dấu phẩy, để tách các đối số. Các chi tiết ở trong Bảng 9-5. PostgreSQL cũng đưa ra các phiên bản các hàm mà sử dụng cú pháp gọi hàm thông thường (xem Bảng 9-6).

> **Lưu ý**: Trước PostgreSQL v8.3, các hàm đó có thể cũng âm thầm chấp nhận các giá trị của vài dạng dữ liệu không phải là chuỗi, vì sự hiện diện của sự ép buộc ngấm ngầm từ các dạng dữ liệu đó đối với text. Những ép buộc đó từng được loại bỏ vì chúng thường xuyên gây ra các hành vi gây ngạc nhiên. Tuy nhiên, toán tử ghép chuỗi (||) vẫn chấp nhận đầu vào không phải là chuỗi, nên miễn là ít nhất một đầu vào là ở dạng chuỗi, như được nêu trong Bảng 9-5. Đối với các trường hợp khác, hãy chèn một sự cưỡng bức rõ ràng vào text nếubạn cần đúp bản hành vi trước đó.

Bảng 9-5. Hàm và toán tử chuỗi SQL

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
string || string	text	Ghép chuỗi	'Post' || 'greSQL'	PostgreSQL
String || non-string or non-string || string	text	Ghép chuỗi với một đầu vào không phải là chuỗi	'Value: ' || 42	Value: 42
bit_length(string)	int	Số bit trong chuỗi	bit_length('jose3'2)	
char_length(string) hoặc character_length(string)	int	Số ký tự trong chuỗi	char_length('jos4e')	
lower(string)	text	Chuyển chuỗi sang chữ thường	lower('TOM')	tom
octet_length(string)	int	Số byte trong chuỗi	octet_length('jo4se')	
overlay(string placing string from int [for int])	text	Thay chuỗi con	overlay('Txxxxas' đặt 'hom' từ 2 với 4)	Thomas

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
position(substring in string)	int	Ví trí chuỗi con được chỉ định	position(’om’ trong ’Thomas’)	3
substring(string [from int] [for int])	text	Trích ra chuỗi con	substring(’Thomas’ từ 2 với 3)	hom
substring(string from pattern)	text	Trích chuỗi con khớp với biểu thức POSIX thông thường. Xem Phần 9.7 để có thêm thông tin về khớp mẫu.	substring(’Thomas’ từ ’... $’)	mas
substring(string from pattern for escape)	text	Trích chuỗi con khớp với biểu thức SQL thông thường. Xem Phần 9.7 để có thêm thông tin về khớp mẫu.	substring(’Thomas’ từ ’%#"o_a#"_’ với ’#’)	oma
trim([leading | trailing | both] [characters] from string)	text	Loại bỏ chuỗi dài nhất có chứa chỉ các ký tự (một dấu trống mặc định) từ điểm đầu/cuối/2 đầu của chuỗi	trim(both ’x’ từ ’xTomxx’)	Tom
upper(string)	text	Chuyển chuỗi thành chữ hoa	upper(’tom’)	TOM

Các hàm bổ sung để điều khiển chuỗi là sẵn có và được liệt kê trong Bảng 9-6. Một số chúng được sử dụng nội bộ để triển khai các hàm chuỗi theo tiêu chuẩn SQL được liệt kê trong Bảng 9-5.

Bảng 9-6. Các hàm chuỗi khác

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
ascii(string)	int	Mã ASCII ký tự đầu tiên của đối số. UTF8 trả về điểm mã ký tự Unicode. Những mã hóa nhiều byte khác thì đối số phải là một ký tự ASCII.	ascii(’x’)	120
btrim(string text [, characters text])	text	Loại bỏ chuỗi dài nhất chỉ bao gồm các ký tự trong các ký tự (một dấu trống là mặc định) từ đầu và cuối của chuỗi.	btrim(’xyxtrimyyx’, ’xy’)	trim
chr(int)	text	Ký tự với mã được đưa ra. Với UTF8 thì đối số được theo dõi như một điểm mã Unicode, đối với các mã hóa nhiều byte khác thì đối số phải chỉ định một ký tự ASCII. Ký tự NULL (0) là không được phép vì các dạng dữ liệu văn bản không thể lưu trữ các byte như vậy.	chr(65)	A
convert(string bytea, src_encoding name, dest_encoding name)	bytea	Biến đổi chuỗi sang dest_encoding. Việc mã hóa ban đầu được src_encoding	convert(’text_in_utf8’, ’UTF8’, ’LATIN1’)	text_in_utf8 represented in Latin-1 encoding

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
		chỉ định. Chuỗi đó phải là hợp lệ trong việc mã hóa này. Các biến đổi có thể do CREATE CONVERSION xác định. Hơn nữa sẽ có một số biến đổi được xác định trước. Xem Bảng 9-7 cho các biến đổi có sẵn.		(ISO 8859-1)
convert_from(string bytea, src_encoding name)	text	Biến đổi chuỗi sang mã cơ sở dữ liệu. Mã gốc ban đầu được src_encoding chỉ định. Chuỗi đó phải hợp lệ trong mã này.	convert_from('text_in_utf8', 'UTF8')	text_in_utf8 represented in the current database encoding
convert_to(string text, dest_encoding name)	bytea	Biến đổi chuỗi sang dest_encoding.	convert_to('some text', 'UTF8')	some text represented in the UTF8 encoding
decode(string text, format text)	bytea	Giải mã các dữ liệu nhị phân từ trình diễn văn bản trong chuỗi. Các lựa chọn cho định dạng là y hệt như trong mã hóa.	decode('MTIzAAE=', 'base64')	x3132330001
encode(data bytea, format text)	text	Mã hóa dữ liệu nhị phân thành một trình diễn văn bản. Các định dạng được hỗ trợ gồm: base64, hex, escape. escape biến đổi các byte 0 và các byte trong thiết lập bit cao sang các tuần tự cơ số 8 (\nnn) và đúp bản các dấu chéo ngược.	encode(E'123\\000\\001', 'base64')	MTIzAAE=
initcap(string)	text	Biến đổi ký tự đầu của từng từ thành chữ hoa và phần còn lại thành chữ thường. Các từ là tuần tự của các ký tự abc được tách bạch nhau bằng các ký tự không phải abc.	initcap('hi THOMAS')	Hi Thomas
length(string)	int	Số lượng các ký tự trong chuỗi	length('jose')	4
length(stringbytea, encoding name)	int	Số lượng các ký tự trong chuỗi trong mã được đưa ra. Chuỗi đó phải là hợp lệ trong mã này.	length('jose', 'UTF8')	4
lpad(string text, length int [, fill text])	text	Điền chuỗi vào độ dài bằng việc treo trước ký tự điền (mặc định một dấu trắng). Nếu chuỗi đó dài hơn rồi so với độ dài thì nó bị cắt bớt (ở bên phải).	lpad('hi', 5, 'xy')	xyxhi

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
ltrim(string text [, characters text])	text	Loại bỏ chuỗi dài nhất chỉ chứa các ký tự từ các ký tự (mặc định một dấu trắng) từ đầu chuỗi.	ltrim(’zzzytrim’, ’xyz’)	trim
md5(string)	text	Tính hàm băm MD5 của chuỗi, trả về kết quả dạng cơ số 16	md5(’abc’)	900150983cd24fb0 d6963f7d28e17f72
pg_client_encoding()	name	Tên mã hóa trạm hiện hành.	pg_client_encoding	SQL()ASCII
quote_ident(string text)	text	Trả về chuỗi được đưa ra được đưa vào dấu ngoặc kép phù hợp để sử dụng như một mã định danh trong một chuỗi lệnh SQL. Các dấu ngoặc kép được bổ sung chỉ nếu cần thiết (như, nếu chuỗi đó gồm các ký tự không phải của mã định danh hoặc có thể là chữ gộp). Các dấu ngoặc kép được nhúng được đúp bản phù hợp. Xem thêm Ví dụ 39-1.	quote_ident(’Foo bar’)	"Foo bar"
quote_literal(string text)	text	Trả về chuỗi được đưa ra phù hợp được đưa vào các dấu ngoặc kép để sử dụng như một hằng chuỗi trong một chuỗi lệnh SQL. Các dấu ngoặc kép đơn được nhúng và các dấu chéo ngược được đúp bản phù hợp. Lưu ý quote_literal trả về null ở đầu vào null; nếu đối số có thể là null, quote_nullable thường phù hợp hơn. Xem thêm Ví dụ 39-1.	quote_literal(E’O\”’RR eeiilllyy’’)	
quote_literal(value anyelement)	text	Ép giá trị được đưa ra thành văn bản và sau đó đưa nó vào dấu ngoặc kép như một hằng. Các dấu ngoặc kép đơn và các dấu chéo ngược được đúp bản phù hợp.	quote_literal(42’.452).5 ’	
quote_nullable(string text)	text	Trả về chuỗi được đưa ra được đưa vào các dấu ngoặc kép phù hợp để sử dụng như một hằng chuỗi trong một chuỗi lệnh SQL hoặc, nếu đối số là null, thì trả về NULL. Các dấu ngoặc đơn được nhúng và các dấu chéo ngược được đúp bản phù hợp. Xem	quote_nullable(NULL)	

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
		thêm Ví dụ 39-1.		
quote_nullable(value anyelement)	text	Ép giá trị được đưa ra thành văn bản và sau đó đưa nó vào các dấu ngoặc kép như một hằng; hoặc, nếu đối số là null, thì trả về NULL. Các dấu ngoặc kép được nhúng và các dấu chéo ngược được đúp bản phù hợp.	quote_nullable(4’24.25.)5’	
regexp_matches(string text, pattern text [, flags text]) setof text[]		Trả về tất cả các chuỗi con bắt được có được từ việc trùng với một biểu thức POSIX thông thường đối với chuỗi đó. Xem phần 9.7.3 để có thêm thông tin.	regexp_matches(’foobar bequebaz’, ’(bar) (beque)’) {bar,beque}	
regexp_replace(string text, pattern text, replacement text [, flags text])	text	Trả về (các) chuỗi con khớp với một biểu thức POSIX thông thường. Xem Phần 9.7.3 để có thêm thông tin.	regexp_replace(’Thomas’, ’.[mN]a.’, ’M’)	ThM
regexp_split_to_array(string text, pattern text [, flags text]) text[]		Chia chuỗi bằng việc sử dụng một biểu thức POSIX thông thường như là dấu tách. Xem Phần 9.7.3 để có thêm thông tin.	regexp_split_to_array(’hello world’, E’\\s+’)	{hello,world}
regexp_split_to_table(string text, pattern text [, flags text]) setof text		Chia chuỗi bằng việc sử dụng một biểu thức POSIX thông thường như là dấu tách. Xem Phần 9.7.3 để có thêm thông tin.	regexp_split_to_table(’hello world’, E’\\s+’)	Helloworld (2 rows)
repeat(string text, number int)	text	Lặp lại chuỗi với số lần được chỉ định.	repeat(’Pg’, 4)	PgPgPgPg
replace(string text, from text, to text)	text	Thay thế tất cả các trường hợp trong chuỗi của chuỗi con từ bằng chuỗi con tới	replace(’abcdefabcdef’, ’cd’, ’XX’)	abXXefabXXef
rpad(string text, length int [, fill text])	text	Điền chuỗi vào độ dài bằng việc nối thêm các ký tự điền (mặc định một ký tự trắng). Nếu chuỗi đó dài hơn rồi so với chiều dài đó thì nó sẽ bị cắt bớt.	rpad(’hi’, 5, ’xy’)	hixyx
rtrim(string text [, characters text])	text	Loại bỏ chuỗi dài nhất chỉ chứa các ký tự từ các ký tự (mặc định là một dấu trắng) từ cuối của chuỗi.	rtrim(’trimxxxx’, ’x’)	trim
split_part(string text, delimiter text, field int)	text	Chia chuỗi theo dấu phân cách và trả về trường được đưa ra (tính từ 1).	split_part(’abc~@~def~@~ghi’, ’~@~’, 2)	def
strpos(string, substring)	int	Địa điểm của chuỗi con được chỉ định (y hệt như vị	strpos(’high’, ’ig’)	2

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
		trí (chuỗi con trong chuỗi), như lưu ý trật tự ngược của đối số)		
substr(string, from [, count])	text	Trích chuỗi con (y hệt như chuỗi con (chuỗi từ đó để đếm) substr('alphabet', 3, 2) ph to_ascii(string text [, encoding text]) text. Biến đổi chuỗi sang ASCII từ việc mã hóa khác (chỉ hỗ trợ biến đổi từ các mã hóa LATIN1, LATIN2, LATIN9, và WIN1250)	to_ascii('Karel'K)arel	
to_hex(number int or bigint)	text	Biến đổi số thành các trình bày tương đương của nó ở cơ số 16	to_hex(21474836477f)	ffffff
translate(string text, from text, to text)	text	Bất kỳ ký tự nào dạng chuỗi mà khớp một ký tự bên trong tập hợp từ được thay thế bằng ký tự tương ứng trong tập hợp đến.	translate('12345', '14', 'ax')	a23x5

Xem thêm hàm tổng hợp string_agg ở Phần 9.18.

Bảng 9-7. Các chuyển đổi được xây dựng sẵn

Tên chuyển đổi a	Mã nguồn	Mã đích
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8

Tên chuyển đổi a	Mã nguồn	Mã đích
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1

Tên chuyển đổi a	Mã nguồn	Mã đích
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
tcvn_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB

Tên chuyển đổi a	Mã nguồn	Mã đích
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_tcvn	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_8	W6I6N1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_12	W5I1N866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
ut8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
ut8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis	EU2C00J4IS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis	SH2I0F0T4_JIS_2004	EUC_JIS_2004

Tên chuyển đổi a	Mã nguồn	Mã đích
Lưu ý: a. Các tên chuyển đổi theo một sơ đồ gọi tên tiêu chuẩn: tên chính thức của mã nguồn với tất cả các ký tự không phải là abc được các dấu gạch chân thay thế, được theo sau bằng _to_, được theo sau bằng tên mã đích được xử lý tương tự. Vì thế, các tên có thể khác nhau với các tên mã hóa theo thói thường.		

9.5. Hàm và toán tử chuỗi nhị phân

Phần này mô tả các hàm và toán tử cho việc kiểm tra và điều khiển các giá trị dạng bytea.

SQL định nghĩa một số hàm chuỗi sử dụng các từ khóa, thay vì dấu phẩy, để tách biệt các đối số. Các chi tiết nằm trong Bảng 9-8. PostgreSQL cũng đưa ra các phiên bản của các hàm có sử dụng cú pháp gọi hàm thông thường (xem Bảng 9-9).

> **Lưu ý**: Các kết quả ví dụ trong trang này giả thiết rằng tham số máy chủ bytea_output được thiết lập để thoát (định dạng PostgreSQL truyền thống).

Bảng 9-8. Các hàm và toán tử chuỗi nhị phân SQL

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
string || string	bytea	Ghép chuỗi	E'\\\\Post'::bytea || E'\\047gres\\000'::bytea	\\Post'gres\000
octet_length(string)	int	Số lượng byte trong chuỗi nhị phân	octet_length(E'jo\\000se'::bytea)	5
overlay(string placing string from int [for int])	bytea	Thay thế chuỗi con	overlay(E'Th\\000omas'::bytea placing E'\\002\\003'::bytea from 2 for 3)	T\\002\\003mas
position(substring in string)	int	Vị trí chuỗi con được chỉ định	position(E'\\000om'::bytea in E'Th\\000omas'::bytea)	3
substring(string [from int] [for int])	bytea	Trích ra chuỗi con	substring(E'Th\\000omas'::bytea from 2 for 3)	h\000o
trim([both] bytes from string)	bytea	Loại bỏ chuỗi dài nhất chỉ có các byte trong các byte từ đầu tới cuối chuỗi	trim(E'\\000'::bytea from E'\\000Tom\\000'::bytea)	Tom

Các hàm điều khiển chuỗi nhị phân bổ sung là sẵn sàng và được liệt kê trong Bảng 9-9. Một số trong số chúng được sử dụng nội bộ để triển khai các hàm chuỗi tiêu chuẩn SQL được liệt kê trong Bảng 9-8.

Bảng 9-9. Các hàm chuỗi nhị phân khác

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
btrim(string bytea, bytes bytea)	bytea	Loại bỏ chuỗi dài nhất chỉ có các byte trong các byte từ đầu tới cuối chuỗi	btrim(E'\\000trim::bytea, E'\\000'::bytea)	\\000'
decode(string text, format text)	bytea	Giải mã dữ liệu nhị phân từ trình bày văn bản theo chuỗi. Các lựa	decode(E'123\\000456', 'escape')	123\000456

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
		chọn về định dạng là y hệt như trong mã hóa.		
encode(data bytea, format text)	text	Dữ liệu mã hóa nhị phân trong trình bày văn bản. Các định dạng được hỗ trợ là: base64, hex, escape. escape chuyển đổi các byte 0 và các byte theo tập hợp bit thành các tuần tự hệ số 8 (\nnn) và đúp bản các dấu chéo ngược.	encode(E'123\\000456'::bytea, 'escape')	123\000456
get_bit(string, offset)	int	Trích bit từ chuỗi	get_bit(E'Th\\000omas'::bytea, 45)	1
get_byte(string, offset)	int	Trích byte từ chuỗi	get_byte(E'Th\\000omas'::bytea, 4)	109
length(string)	int	Độ dài chuỗi nhị phân	length(E'jo\\000se'::bytea)	5
md5(string)	text	Tính băm MD5 của chuỗi, trả về kết quả ở hệ số 16 (hexadecimal)	md5(E'Th\\000oma8sa'b:2:db3yct9e6a8)	9aaf18 b4958c334c82d8b1
set_bit(string, offset, newvalue)	bytea	Thiết lập bit trong chuỗi	set_bit(E'Th\\000omas'::bytea, 45, 0)	Th\000omAs
set_byte(string, offset, newvalue)	bytea	Thiết lập byte trong chuỗi	set_byte(E'Th\\000omas'::bytea, 4, 64)	Th\000o@as

get_byte và set_byte số byte đầu của một chuỗi nhị phân như byte 0. get_bit và set_bit số bit từ phải qua bên trong từng byte; ví dụ bit 0 là bit ít đáng kể nhất của byte đầu tiên, và bit 15 là bit đáng kể nhất của byte thứ 2.

9.6. Hàm và toán tử chuỗi bit

Phần này mô tả các hàm và toán tử cho việc xem xét và điều khiển các chuỗi bit, đó là các giá trị của các dạng bit và bit varying. Ngoài các toán tử so sánh thông thường, các toán tử được nêu trong Bảng 9-10 có thể được sử dụng. Các toán hạng chuỗi bit &, | và # phải có độ dài bằng nhau. Khi dịch chuyển bit, độ dài ban đầu của chuỗi được giữ lại, như được chỉ ra trong các ví dụ.

Bảng 9-10. Các toán tử chuỗi bit

Toán tử	Mô tả	Ví dụ	Kết quả
||	ghép (concatenation)	B'10001' || B'011'	10001011
&	VÀ bitwise (bitwise AND)	B'10001' & B'01101'	00001
|	HOẶC bitwise (bitwise OR)	B'10001' | B'01101'	11101
#	bitwise XOR	B'10001' # B'01101'	11100
~	bitwise NOT	~ B'10001'	01110

Toán tử	Mô tả	Ví dụ	Kết quả
<<	dịch sang trái bitwise (bitwise shift left)	B'10001' << 3	01000
>>	dịch sang phải bitwise (bitwise shift right)	B'10001' >> 2	00100

Các hàm tiêu chuẩn SQL sau đây làm việc trong cả các chuỗi bit cũng như các chuỗi ký tự: length, bit_length, octet_length, position, substring, overlay.

Các hàm sau làm việc trong các chuỗi bit cũng như các chuỗi nhị phân: get_bit, set_bit. Khi làm việc với một chuỗi bit, các hàm đó đánh số cho bit đầu tiên (bên trái nhất) của chuỗi như là bit 0. Hơn nữa, có khả năng để đưa ra các giá trị tích phân tới và từ dạng bit. Một số ví dụ:

```
44::bit(10)             0000101100
44::bit(3)              100
cast(-44 as bit(12))    111111010100
'1110'::bit(4)::integer 14
```

Lưu ý rằng việc đưa ra chỉ "bit" nghĩa là đưa ra cho bit(1), và vì thế sẽ phân phối chỉ bit ít đáng kể nhất của số interger đó.

> **Lưu ý**: Trước PostgreSQL 8.0, việc đưa ra một số interger cho bit(n) có thể sao chép n bit trái nhất của số interger đó, trong khi bây giờ nó sao chép n bit phải nhất. Hơn nữa, việc đưa ra một số integer tới một chuỗi bit độ rộng lớn hơn so với bản thân số integer đó sẽ mở rộng dấu hiệu về bên trái.

9.7. Khớp mẫu

Có 3 tiếp cận riêng rẽ để khớp mẫu được PostgreSQL cung cấp: toán tử LIKE của SQL truyền thống, toán tử gần đây hơn SIMILAR TO (được bổ sung vào SQL:1999), và các biểu thức dạng POSIX thông thường. Ngoài các toán tử cơ bản "làm cho chuỗi này khớp với mẫu này?", các hàm là sẵn sàng để mở rộng hoặc thay thế các chuỗi trùng khớp và để chia một chuỗi ở các vị trí khớp.

> **Mẹo**: Nếu bạn có các nhu cầu khớp mẫu mà vượt ra khỏi điều này, hãy cân nhắc việc viết một hàm do người sử dụng định nghĩa trong Perl hoặc TCL.

9.7.1. LIKE

string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]

Biểu thức LIKE trả về đúng - true nếu *chuỗi* khớp với *mẫu pattern* được cung cấp. (Như được kỳ vọng, biểu thức NOT LIKE trả về sai - false nếu LIKE trả về đúng, và ngược lại. Một biểu thức tương đương là NOT (*string* LIKE *pattern*)).

Nếu *mẫu pattern* không chứa các dấu % hoặc các dấu gạch chân, thì mẫu đó chỉ đại diện cho bản thân chuỗi đó; trong trường hợp đó thì LIKE hành động như là toán tử bằng. Một dấu gạch chân (_) trong pattern sẽ đại diện cho (các trùng khớp) bất kỳ ký tự nào; một dấu % khớp với bất kỳ tuần tự nào của số 0 hoặc nhiều hơn các ký tự. Một số ví dụ:

```
'abc' LIKE 'abc'        true
'abc' LIKE 'a%'         true
'abc' LIKE '_b_'        true
'abc' LIKE 'c'          false
```

Khớp mẫu LIKE luôn bao trùm toàn bộ chuỗi. Vì thế, để khớp một tuần tự bất kỳ ở đâu trong một chuỗi, thì mẫu đó phải bắt đầu và kết thúc với một dấu %.

Để khớp một dấu gạch chân hoặc % của hằng mà không có việc khớp với các ký tự khác, ký tự tương ứng trong mẫu pattern phải có ký tự thoát đi trước. Ký tự thoát mặc định là dấu chéo ngược nhưng một cách thoát khác cũng có thể được chọn bằng việc sử dụng mệnh đề thoát ESCAPE. Để khớp bản thân ký tự thoát, hãy viết 2 ký tự thoát.

Lưu ý rằng dấu chéo ngược có rồi một ý nghĩa đặc biệt trong các hằng chuỗi, nên để viết một hằng mẫu có chứa một dấu chéo ngược thì bạn phải viết 2 dấu chéo ngược trong một lệnh SQL (giả thiết cú pháp chuỗi thoát được sử dụng, xem Phần 4.1.2.1). Vì thế, việc viết một mẫu thực sự khớp với một dấu chéo ngược nghĩa là viết 4 dấu chéo ngược trong lệnh. Bạn có thể tránh điều này bằng việc chọn một ký tự thoát khác với ESCAPE; rồi một dấu chéo ngược sẽ không là đặc biệt với LIKE nữa. (Nhưng dấu chéo ngược vẫn là đặc biệt với trình phân tích hằng chuỗi, nên bạn vẫn cần 2 dấu đó để khớp với một dấu chéo ngược).

Cũng có khả năng không chọn ký tự thoát nào bằng việc viết ESCAPE ''. Điều này sẽ vô hiệu hóa có hiệu quả cơ chế thoát, nó làm cho có khả năng tắt ý nghĩa đặc biệt của các dấu % và gạch chân trong mẫu đó.

Từ khóa ILIKE có thể được sử dụng thay cho LIKE để tạo sự khớp phân biệt chữ hoa chữ thường theo ngôn ngữ bản địa đang hiện diện. Đây không phải là tiêu chuẩn SQL nhưng là một mở rộng của PostgreSQL.

Toán tử ~ ~ là tương đương với LIKE, còn ~ ~ * là tương đương với ILIKE. Cũng có các toán tử !~ ~ và !~ ~* đại diện cho NOT LIKE và NOT ILIKE, một cách tương ứng. Tất cả các toán tử đó là đặc thù cho PostgreSQL.

9.7.2. Biểu thức thông dụng SIMILAR TO

string SIMILAR TO *pattern* [ESCAPE *escape-character*]
string NOT SIMILAR TO *pattern* [ESCAPE *escape-character*]

Toán tử SIMILAR TO trả về đúng - true hoặc sai - false phụ thuộc vào việc liệu mẫu đó có khớp với chuỗi được đưa ra hay không. Tương tự như LIKE, ngoại trừ là nó diễn giải mẫu đó bằng việc sử dụng định nghĩa tiêu chuẩn SQL của một biểu thức thông dụng. Các biểu thức thông dụng SQL là một sự tò mò giữa ký hiệu LIKE và ký hiệu biểu thức thông thường.

Giống như LIKE, toán tử SIMILAR TO chỉ thành công nếu mẫu của nó khớp với toàn bộ chuỗi; đây có lẽ không là hành vi của biểu thức thông thường chung ở những nơi mà mẫu đó có thể khớp với bất kỳ phần nào của chuỗi. Hơn nữa, giống như LIKE , SIMILAR TO sử dụng các dấu _và % như là các ký tự * biểu thị bất kỳ ký tự duy nhất nào và bất kỳ chuỗi nào, một cách tương ứng (có sự so sánh dấu

chấm (.) với dấu chấm đi với ký tự sao (.*) trong các biểu thức POSIX thông thường).

Bổ sung thêm vào các cơ sở được mượn từ LIKE , SIMILAR TO hỗ trợ cho các ký tự khớp mẫu đó được mượn từ các biểu thức POSIX thông thường:

- | biểu thị sự xoay chiều (hoặc 2 lựa chọn thay thế cho nhau).
- * biểu thị sự lặp lại khoản trước đó 0 hoặc nhiều lần.
- + biểu thị sự lặp lại khoản trước đó 1 hoặc nhiều lần.
- ? biểu thị sự lặp lại khoản trước đó 0 hoặc 1 lần.
- {m} biểu thị sự lặp lại khoản trước đó chính xác m lần.
- {m,} biểu thị sự lặp lại khoản trước đó m hoặc nhiều hơn lần.
- {m,n} biểu thị sự lặp lại khoản trước đó ít nhất m và không lớn hơn n lần.
- Các dấu ngoặc đơn ( ) có thể được sử dụng để nhóm các khoản trong một khoản logic duy nhất.
- Một biểu thức trong các dấu ngoặc vuông […] chỉ định một lớp ký tự, hệt như trong các biểu thức POSIX thông thường.

Lưu ý rằng dấu chấm (.) không phải là một ký tự cho SIMILAR TO.

Như với LIKE, một dấu chéo ngược vô hiệu hóa ý nghĩa đặc biệt của bất kỳ siêu ký tự nào; hoặc một ký tự thoát khác có thể được chỉ định với ESCAPE.

Một số ví dụ:

```
'abc' SIMILAR TO 'abc'          true
'abc' SIMILAR TO 'a'            false
'abc' SIMILAR TO '%(b|d)%'      true
'abc' SIMILAR TO '(b|c)%'       false
```

Hàm substring với 3 tham số, substring (*string* from *pattern* for *escape-character*), đưa ra sự trích một chuỗi khớp với một mẫu biểu thức thông thường SQL. Như với SIMILAR TO, mẫu được chỉ định phải khớp với toàn bộ chuỗi dữ liệu, hoặc nếu không thì hàm thất bại và trả về null. Để chỉ ra phần của mẫu mà sẽ được trả về thành công, mẫu đó phải có 2 lần ký tự thoát theo sau là một dấu ngoặc kép ("). Các chữ khớp với phần mẫu giữa các đánh dấu đó được trả về.

Một số ví dụ, với #" phân định chuỗi trả về:

```
substring('foobar' from '%#"o_b#"%' for '#')          oob
substring('foobar' from '#"o_b#"%' for '#')           NULL
```

9.7.3. Các biểu thức POSIX thông thường

Bảng 9-11 liệt kê các toán tử có sẵn cho việc khớp mẫu bằng việc sử dụng các biểu thức POSIX thông thường.

Bảng 9-11. Các toán tử khớp biểu thức thông thường

Toán tử	Mô tả	Ví dụ
~	Khớp biểu thức thông thường, phân biệt chữ hoa chữ thường	'thomas' ~ '.*thomas.*'
~*	Khớp biểu thức thông thường, không phân biệt chữ hoa chữ thường	'thomas' ~* '.*Thomas.*'
!~	Không khớp biểu thức thông thường, phân biệt chữ hoa chữ thường	'thomas' !~ '.*Thomas.*'
!~*	Không khớp biểu thức thông thường, không phân biệt chữ hoa chữ thường	'thomas' !~* '.*vadim.*'

Các biểu thức POSIX thông thường đưa ra một phương tiện mạnh hơn cho việc khớp mẫu so với các toán tử LIKE và SIMILAR TO. Nhiều công cụ Unix như egrep, sed hoặc awk sử dụng một ngôn ngữ khớp mẫu là tương tự với thứ được mô tả ở đây.

Một biểu thức thông thường là một sự tuần tự các ký tự mà là một định nghĩa ngắn gọn của một tập hợp các chuỗi (một tập hợp thông thường). Một chuỗi được nói là khớp với một biểu thức thông thường nếu nó là một thành viên của tập thông thường được biểu thức thông thường đó mô tả. Như với LIKE, các ký tự mẫu khớp các ký tự chuỗi một cách chính xác trừ phi chúng là những ký tự đặc biệt trong ngôn ngữ của biểu thức thông thường - nhưng các biểu thức thông thường sử dụng các ký tự đặc biệt khác với LIKE làm. Không giống như mẫu LIKE, một biểu thức thông thường được phép khớp ở bất kỳ đâu trong một chuỗi, trừ phi biểu thức thông thường đó rõ ràng neo đậu ở đầu hoặc cuối của chuỗi. Một vài ví dụ

```
'abc' ~ 'abc'           true
'abc' ~ '^a'            true
'abc' ~ '(b|d)'         true
'abc' ~ '^(b|c)'        false
```

Ngôn ngữ POSIX mẫu được mô tả chi tiết hơn nhiều bên dưới. Hàm trích chuỗi với 2 tham số, substring(string from pattern), đưa ra sự trích xuất một chuỗi con khớp với mẫu biểu thức POSIX thông thường. Nhưng nếu mẫu đó có chứa bất kỳ dấu ngoặc kép nào, thì tỷ lệ văn bản và khớp với các biểu thức con đầu tiên trong các dấu ngoặc đơn (biểu thức con mà dấu ngoặc trái của nó đến trước) được trả về. Bạn có thể đặt các dấu ngoặc đơn xung quanh toàn bộ biểu thức nếu bạn muốn sử dụng các dấu ngoặc đơn bên trong nó mà không động chạm tới ngoại lệ này. Nếu bạn cần các dấu ngoặc đơn trong mẫu đó trước biểu thức con bạn muốn trích, hãy xem các dấu ngoặc đơn không bắt được được mô tả bên dưới. Một số ví dụ:

```
substring('foobar' from 'o.b')          oob
substring('foobar' from 'o(.)b')        o
```

Hàm regexp_replace đưa ra sự thay thế văn bản mới cho các chuỗi con khớp với các mẫu biểu thức POSIX thông thường. Nó có cú pháp regexp_replace(source, pattern, replacement [, flags]). Chuỗi nguồn được trả về không thay đổi nếu không có sự trùng khớp nào với mẫu pattern. Nếu có sự trùng khớp, thì chuỗi nguồn được trả về với chuỗi thay thế replacement được thay thế cho chuỗi con trùng khớp. Chuỗi thay thế replacement có thể chứa \n, trong đó n là 1 tới 9, để chỉ ra rằng chuỗi con nguồn khớp với biểu thức con trong dấu ngoặc thứ n của mẫu sẽ được chèn vào, và nó có thể gồm \& để chỉ rằng chuỗi con khớp với toàn bộ mẫu nên được chèn vào. Hãy viết \ nếu bạn cần phải đặt một dấu chéo ngược hằng trong văn bản thay thế. (Như thường lệ, hãy nhớ đúp bản các dấu chéo ngược được viết trong các chuỗi hằng theo nghĩa đen, giả thiết cú pháp chuỗi thoát được sử dụng).

Tham số flags là một chuỗi văn bản tùy ý có chứa các cờ 0 hoặc nhiều ký tự duy nhất hơn mà thay đổi hành vi của hàm. Cờ i chỉ việc trùng khớp có phân biệt chữ hoa chữ thường, trong khi cờ g chỉ sự thay thế của từng chuỗi con trùng khớp thay vì chỉ chuỗi con đầu tiên trùng khớp. Các cờ được hỗ trợ khác được mô tả trong Bảng 9-19. Một số ví dụ:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                    fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                    fooXX
regexp_replace('foobarbaz', 'b(..)', E'X\\1Y', 'g')
                                    fooXarYXazY
```

Hàm regexp_matches trả về một mảng văn bản của tất cả các chuỗi con lấy được là kết quả từ việc khớp mẫu của một biểu thức POSIX thông thường. Nó có cú pháp regexp_matches (string , pattern [, flags]). Chức năng đó có thể không trả về hàng nào, trả về 1 hàng, hoặc nhiều hàng (xem cờ g bên dưới). Nếu mẫu pattern không trùng khớp, thì hàm đó không trả về hàng nào. Nếu mẫu đó không chứa các biểu thức con trong các dấu ngoặc đơn, thì từng hàng được trả về là một mảng văn bản có 1 yếu tố duy nhất, có chứa chuỗi con trùng khớp với toàn bộ mẫu. Nếu mẫu đó có chứa các biểu thức con trong các dấu ngoặc đơn, thì hàm trả về một mảng văn bản mà phần tử thứ n của nó là chuỗi con trùng khớp với biểu thức con nằm trong các dấu ngoặc đơn thứ n của mẫu đó (không tính tới các dấu ngoặc đơn "không bắt được"; xem bên dưới để có thêm chi tiết). Tham số cờ flags là một chuỗi văn bản tùy chọn có chứa các cờ 0 hoặc nhiều hơn 1 ký tự mà thay đổi hành vi của hàm. Cờ g làm cho hàm tìm từng sự trùng khớp trong chuỗi đó, không chỉ sự trùng khớp đầu tiên, và trả về một hàng cho từng sự trùng khớp như vậy. Các cờ được hỗ trợ khác được mô tả trong Bảng 9-19. Một số ví dụ:

```
SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');

regexp_matches
----------------
{bar,beque}
(1 row)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');

regexp_matches
----------------
{bar,beque}
{bazil,barf}
(2 rows)

SELECT regexp_matches('foobarbequebaz', 'barbeque');

regexp_matches
----------------
{barbeque}
(1 row)
```

Có khả năng để ép regexp_matches() luôn trả về một hàng bằng việc sử dụng một lựa chọn phụ (con); điều này là hữu dụng một phần trong một danh sách chọn đích SELECT khi bạn muốn tất cả các hàng được trả về, thậm chí các hàng không trùng khớp.

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

Hàm regexp_split_to_table chia một chuỗi bằng việc sử dụng một mẫu biểu thức POSIX thông thường như một ký tự phân cách. Nó có cú pháp regexp_split_to_table (string , pattern [, flags]). Nếu không có sự trùng khớp nào đối với mẫu pattern, thì hàm đó trả về chuỗi string. Nếu có ít nhất một sự trùng khớp, đối với mỗi sự trùng khớp nó trả về văn bản từ cuối của sự trùng khớp cuối cùng (hoặc đầu của chuỗi đó) tới đầu của sự trùng khớp. Khi không có các trùng khớp khác nữa, thì nó trả về văn bản từ cuối của sự trùng khớp cuối cùng tới cuối của chuỗi đó. Tham số flags là một chuỗi văn bản tùy chọn có chứa các cờ 0 hoặc nhiều hơn 1 ký tự duy nhất mà thay đổi hành vi của hàm đó. regexp_split_to_table hỗ trợ các cờ được mô tả trong Bảng 9-19.

Hàm regexp_split_to_array hành xử y hệt như regexp_split_to_table, ngoại trừ là regexp_split_to_array trả về kết quả của nó như một mảng văn bản text. Nó có cú pháp regexp_split_to_array (string , pattern [, flags]). Các tham số là y hệt như đối với regexp_split_to_table. Một số ví dụ:

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumped over the lazy dog', E'\\

foo
--------
the
quick
brown
fox
jumped
over
the
lazy
dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumped over the lazy dog', E'\\s+');

regexp_split_to_array
-----------------------------------------------
{the,quick,brown,fox,jumped,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', E'\\s*') AS foo;
foo
-----
t
h
e
q
u
i
c
k
b
r
o
w
n
f
o
x
(16 rows)
```

Như ví dụ cuối cùng thể hiện, các hàm chia regexp bỏ qua các trùng khớp độ dài 0 mà diễn ra ở đầu

hoặc cuối của chuỗi hoặc ngay sau một sự trùng khớp trước đó. Điều này là ngược lại với định nghĩa khắt khe của việc trùng khớp regexp mà được regexp_matches triển khai, nhưng thường là hành vi tiện lợi nhất trong thực tiễn. Các hệ thống phần mềm khác như Perl sử dụng các định nghĩa tương tự.

9.7.3.1. Chi tiết của biểu thức thông thường

Các biểu thức thông thường của PostgreSQL được triển khai bằng việc sử dụng một gói phần mềm được Henry Spencer viết. Nhiều mô tả các biểu thức thông thường bên dưới được sao chép nguyên bản từ sách chỉ dẫn của ông.

Các biểu thức thông thường - RE (Regular Expressions), như được định nghĩa trong POSIX 1003.2, có ở 2 dạng: các RE được mở rộng hoặc các ERE (thô thiển như các biểu thức egrep), và các RE hoặc BRE cơ bản (thô thiển như các dạng ed). PostgreSQL hỗ trợ cả 2 dạng, và cũng triển khai một số mở rộng mà không có trong tiêu chuẩn POSIX, nhưng đã được sử dụng rộng rãi vì tính sẵn sàng của chúng trong các ngôn ngữ lập trình như Perl và TCL. Các RE sử dụng các mở rộng phi POSIX đó được gọi là các RE hoặc ARE tiên tiến trong tài liệu này. Các ARE hầu như chính xác là một siêu tập hợp các ERE, nhưng các BRE có vài sự không tương thích đáng lưu ý (cũng như bị hạn chế hơn nhiều). Chúng tôi lần đầu mô tả các dạng ARE và ERE, lưu ý các tính chất áp dụng chỉ cho các ARE, và sau đó mô tả cách mà các BRE khác biệt.

> **Lưu ý**: PostgreSQL luôn ngay từ đầu giả định rằng một biểu thức thông thường tuân theo các qui tắc của ARE. Tuy nhiên, các qui tắc của ERE hoặc BRE bị hạn chế hơn có thể được lựa chọn bằng việc thêm vào trước một lựa chọn được nhúng vào mẫu RE, như được mô tả trong Phần 9.7.3.4. Điều này có thể là hữu dụng cho tính tương thích với các ứng dụng mà kỳ vọng chính xác các qui tắc của POSIX 1003.2.

Một biểu thức thông thường được xác định như một hoặc nhiều *nhánh* hơn, được phân tách nhau bằng dấu |. Nó khớp với bất kỳ điều gì mà khớp được với một trong các nhánh đó.

Một nhánh là 0 hoặc các *nguyên tử* hoặc *ràng buộc* định lượng được hơn, được ghép vào nhau. Nó trùng khớp với một sự trùng khớp cho cái đầu, tiếp sau là sự trùng khớp cho cái thứ 2...; một nhánh rỗng trùng khớp với chuỗi rỗng.

Một nguyên tử đủ điều kiện là một *nguyên tử* có khả năng sau đó có một *trình định lượng* duy nhất. Không có một trình định lượng, nó khớp với một sự trùng khớp cho nguyên tử đó. Với một trình định lượng, nó có thể khớp với vài sự trùng khớp của nguyên tử đó. Một nguyên tử có thể là bất kỳ trong các khả năng được chỉ ra trong Bảng 9-12. Các trình định lượng có khả năng và các ý nghĩa của chúng được chỉ ra trong Bảng 9-13.

Một *ràng buộc* khớp với một chuỗi rỗng, nhưng chỉ trùng khớp khi các điều kiện đặc biệt được đáp ứng. Một ràng buộc có thể được sử dụng ở nơi mà một nguyên tử có thể được sử dụng, ngoại trừ nó không thể được đi theo với một trình định lượng. Các ràng buộc đơn giản được chỉ ra trong Bảng 9-14; một số ràng buộc hơn được mô tả sau.

Bảng 9-12. Các nguyên tử của biểu thức thông thường

Nguyên tử	Mô tả
(re)	(ở những nơi mà re là bất kỳ biểu thức nào) khớp với một trùng khớp cho re, với sự trùng khớp được lưu ý về việc báo cáo có khả năng
(?:re)	như ở trên, nhưng sự trùng khớp không được lưu ý cho việc báo cáo (một tập hợp các dấu ngoặc đơn "không bắt được") (chỉ các ARE)
.	khớp với bất kỳ ký tự đơn nào
[chars]	biểu thức trong dấu ngoặc vuông, khớp với bất kỳ một trong số các ký tự (xem Phần 9.7.3.2 để có thêm chi tiết)
\k	(ở những nơi k là ký tự không phải abc) các trùng khớp ký tự đó được lấy như một ký tự thông thường, như \\ khớp với một ký tự dấu chéo ngược
\c	ở những nơi mà c là ký tự abc (có khả năng có các ký tự khác theo sau) là một sự thoát, xem Phần 9.7.3.3 (chỉ các ARE; trong các ERE và BRE, điều này trùng khớp với c)
{	khi đi theo sau là một ký tự khác với một ký tự số, trùng khớp ký tự đúp trái {; khi theo sau là một ký tự số, đây là đầu của một ràng buộc (xem bên dưới)
x	ở những nơi mà x là một ký tự duy nhất mà không có ý nghĩa khác, khớp với ký tự đó

Một RE không thể kết thúc bằng dấu chéo ngược \.

> **Lưu ý**: Hãy nhớ là dấu chéo ngược (\) có rồi một ý nghĩa đặc biệt trong các hằng chuỗi PostgreSQL. Để viết một hằng mẫu có chứa một dấu chéo ngược, bạn phải viết 2 dấu chéo ngược trong lệnh, giả thiết cú pháp chuỗi thoát được sử dụng (xem Phần 4.1.2.1).

Bảng 9-13. Các trình định lượng biểu thức thông thường

Trình định lượng	Các trùng khớp
*	sự tuần tự của 0 hoặc nhiều sự trùng khớp hơn của hạt nhân
+	sự tuần tự của 1 hoặc nhiều sự trùng khớp hơn của hạt nhân
?	sự tuần tự của 0 hoặc 1 sự trùng khớp hơn của hạt nhân
{m}	sự tuần tự của chính xác m sự trùng khớp của hạt nhân
{m,}	sự tuần tự của m hoặc nhiều sự trùng khớp hơn của hạt nhân
{m,n}	sự tuần tự của m tới n (bao gồm cả) các sự trùng khớp của hạt nhân; m không thể vượt n
*?	phiên bản không tham của *
+?	phiên bản không tham của +
??	phiên bản không tham của ?
{m}?	phiên bản không tham của {m}
{m,}?	phiên bản không tham của {m,}
{m,n}?	phiên bản không tham của {m,n}

Các mẫu sử dụng {...} được biết như là các giới hạn. Các số m và n trong một giới hạn là các số nguyên thập phân không được ký với các giá trị cho phép từ >= 0 tới <= 255.

Các trình định lượng không tham (chỉ sẵn sàng ở các ARE) khớp với cùng các khả năng như các đối tác (tham) thông thường tương ứng của chúng, nhưng ưu tiên hơn số nhỏ nhất thay vì số lớn nhất các trùng khớp. Xem Phần 9.7.3.5 để có thêm chi tiết.

> **Lưu ý**: Một trình định lượng không thể ngay lập tức theo sau một trình định lượng khác, như, ** là không hợp lệ. Một trình định lượng không thể bắt đầu một biểu thức hoặc biểu thức con hoặc đi theo dấu mũ ^ hoặc dấu |.

Bảng 9-14. Các ràng buộc biểu thức thông thường

Ràng buộc	Mô tả
^	khớp ở đầu của chuỗi
$	khớp ở cuối của chuỗi
(?=re)	cái nhìn tích cực về phía trước trùng khớp với thời điểm bất kỳ nơi mà một chuỗi con khớp với RE bắt đầu (chỉ các ARE)
(?!re)	cái nhìn tiêu cực về phía trước khớp với bất kỳ điểm nào nơi mà không chuỗi con nào trùng khớp với RE bắt đầu (chỉ các ARE)

Các ràng buộc hướng về phía trước không thể có các tham chiếu ngược (xem Phần 9.7.3.3), và tất cả các dấu ngoặc đơn bên trong chúng được xem như là không bắt được.

9.7.3.2. Các biểu thức dấu ngoặc vuông

Một *biểu thức dấu ngoặc vuông* là một danh sách các ký tự đặt trong các dấu ngoặc vuông [ ]. Nó thường khớp với bất kỳ ký tự nào từ danh sách (xem bên dưới). Nếu danh sách đó bắt đầu bằng ^, thì nó khớp với bất kỳ ký tự đơn nào không từ phần còn lại của danh sách đó. Nếu 2 ký tự trong danh sách tách biệt nhau bằng dấu gạch ngang (-), thì đây là viết tắt cho cả dãy đầy đủ các ký tự giữa 2 ký tự đó (tính cả 2 ký tự đó) trong tuần tự so sánh đó, nghĩa là, [0-9] theo ASCII khớp với bất kỳ ký tự số thập phân nào. Là không hợp lệ cho 2 dãy để chia sẻ một điểm kết thúc, như, a-c-e. Các dãy là rất phụ thuộc vào tuần tự đối chiếu so sánh, vì thế các chương trình khả chuyển nên tránh dựa vào chúng.

Để đưa vào một dấu] trong danh sách, hãy làm cho nó thành ký tự đầu tiên (sau dấu ^, nếu điều đó được sử dụng). Để đưa vào một dấu -, hãy làm cho nó thành ký tự đầu hoặc cuối, hoặc điểm kết thúc thứ 2 của dãy. Để sử dụng một dấu - như là điểm kết thúc đầu tiên của một dãy, hãy đưa nó vào trong [. and .] để làm cho nó thành một phần tử đối chiếu (xem bên dưới). Với ngoại lệ của các ký tự đó, một số kết hợp bằng việc sử dụng [(xem các đoạn tiếp sau), và các ký tự thoát (chỉ các ARE), tất cả các ký tự đặc biệt khác đánh mất ý nghĩa đặc biệt của chúng bên trong một biểu thức dấu ngoặc vuông. Đặc biệt, dấu chéo ngược \ không là đặc biệt khi tuân theo các qui tắc ERE hoặc BRE, dù nó là đặc biệt (như giới thiệu một ký tự thoát) trong các ARE.

Trong một biểu thức dấu ngoặc vuông, một phần tử đối chiếu (một ký tự, một tuần tự nhiều ký tự mà đối sánh dường như nó là một ký tự duy nhất, hoặc một tên tuần tự đối sánh) được đưa vào trong [. and .] nghĩa là sự tuần tự các ký tự của phần tử đối sánh đó. Sự tuần tự được ứng xử như

một phần tử duy nhất của danh sách các biểu thức dấu ngoặc vuông. Điều này cho phép một biểu thức dấu ngoặc vuông chứa một phần tử đối sánh có nhiều ký tự khớp nhiều hơn so với một ký tự, nghĩa là, nếu tuần tự đối sánh đó bao gồm một phần tử đối sánh ch, thì RE [[.ch.]]*c sẽ khớp với 5 ký tự đầu tiên của chchcc.

> **Lưu ý**: PostgreSQL hiện hành không hỗ trợ các phần tử đối sánh nhiều ký tự. Thông tin này mô tả hành vi có khả năng trong tương lai.

Trong một biểu thức dấu ngoặc vuông, một phần tử đối sánh được đưa vào trong [= and =] là lớp tương đương, thay cho các tuần tự các ký tự của tất cả các phần tử đối sánh tương ứng cho tuần tự đó, bao gồm chính nó. (Nếu không có các phần tử đối sánh tương ứng khác, thì đối xử dường như là việc đưa vào các dấu ngắt là [.and.]). Ví dụ, nếu o và ^ là các thành viên của một lớp tương ứng, thì [[=o=]], [[=^=]], và [o^] là tất cả các đồng nghĩa. Một lớp tương ứng không thể là một điểm kết thúc của một dãy.

Trong một biểu thức dấu ngoặc vuông, tên của một lớp ký tự được đưa vào trong [: and :] các chỗ cho danh sách tất cả các ký tự thuộc về lớp đó. Các tên lớp ký tự tiêu chuẩn là: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit. Các chỗ cho các lớp ký tự đó được xác định theo ctype. Một miền địa phương có thể đưa ra các thứ khác. Một lớp ký tự không thể được sử dụng như một điểm cuối của một dãy .

Có 2 trường hợp đặc biệt các biểu thức dấu ngoặc vuông: các biểu thức dấu ngoặc vuông [[:<:]] và [[:>:]] là các ràng buộc, khớp với các chuỗi rỗng ở đầu và cuối của một từ một cách tương ứng. Một từ được xác định như một sự tuần tự các ký tự của từ mà không được đứng trước cũng không đứng đằng sau các ký tự của từ. Một ký tự của từ là một ký tự alnum (như được ctype xác định) hoặc một dấu gạch chân. Đây là một mở rộng, tương thích với nhưng không do POSIX 1003.2 chỉ định, và nên được sử dụng với sự thận trọng trong phần mềm có ý định sẽ là khả chuyển đổi với các hệ thống khác. Các sự thoát ràng buộc được mô tả bên dưới thường được ưu tiên; chúng không còn là tiêu chuẩn nữa, mà dễ dàng hơn để gõ.

9.7.3.3. Các thoát biểu thức thông thường

Các *thoát* là những tuần tự đặc biệt bắt đầu với dấu chéo ngược \ đi theo sau với một ký tự abc. Các thoát có vài biến thể khác nhau: khoản vào của ký tự, các tốc ký lớp, các thoát ràng buộc và các tham chiếu ngược. Một dấu chéo ngược theo sau là một ký tự abc nhưng không cấu thành một thoát hợp lý là không hợp lệ trong các ARE. Trong các ERE, không có các thoát: ngoài một biểu thức dấu ngoặc vuông, thì một dấu chéo ngược \ đi sau một ký tự abc chỉ thay cho ký tự đó như một ký tự thông thường, và bên trong một biểu thức dấu ngoặc vuông, thì dấu chéo ngược \ là một ký tự thông thường. (Cái sau là cái thực sự không tương thích giữa các ERE và ARE).

Các *thoát khoản vào của ký tự (character-entry escape)* tồn tại để làm cho dễ dàng hơn để chỉ định các ký tự không in được và các ký tự không thuận tiện khác trong các RE. Chúng được chỉ ra trong Bảng 9-15.

Các *thoát tốc ký lớp (Class-shorthand escapes)* đưa ra các tốc ký cho các lớp ký tự nhất định được sử dụng phổ biến. Chúng được chỉ ra trong Bảng 9-16.

Một *thoát ràng buộc* là một ràng buộc, khớp với chuỗi rỗng nếu các điều kiện đặc biệt được đáp ứng, được viết như một thoát. Chúng được chỉ ra trong Bảng 9-17.

Một *tham chiếu ngược* (`\n`) khớp với cùng y hệt chuỗi được biểu thức phụ trong các dấu ngoặc đơn trước đó khớp được số n chỉ định (xem Bảng 9-18). Ví dụ, `([bc])\1` khớp với `bb` hoặc `cc` nhưng không với `bc` hoặc `cb`. Biểu thức con phải hoàn toàn đứng trước tham chiếu ngược trong RE. Các biểu thức con được đánh số theo trật tự các dấu ngoặc đơn dẫn dắt của chúng. Các dấu ngoặc đơn không bắt được sẽ không xác định các biểu thức con.

> **Lưu ý**: Hãy ghi nhớ rằng một dấu chéo ngược \ dẫn dắt của sự thoát sẽ cần phải được đúp bản khi vào mẫu như một hằng chuỗi SQL. Ví dụ:
>
> ```
> '123' ~ E'^\\d{3}' true
> ```

Bảng 9-15. Các thoát khoản vào của ký tự của biểu thức thông thường

Thoát	Mô tả
\a	ký tự cảnh báo (chuông), như trong C
\b	dấu ngược backspace, như trong C
\B	đồng nghĩa cho dấu chéo ngược (\) để giúp làm giảm nhu cầu về đúp bản dấu chéo ngược
\cX	(trong đó X là ký tự bất kỳ) ký tự mà 5 bit trật tự thấp của nó là y hệt như các bit của X, và các bit khác của nó tất cả đều là 0
\e	ký tự và tên trật tự đối sánh của nó là ESC, hoặc ký tự với giá trị 033 theo hệ số 8
\f	mẫu nuôi, như trong C
\n	dòng mới, như trong C
\r	trả về sự thi hành, như trong C
\t	thẻ tab nằm ngang, như trong C
\uwxyz	(trong đó wxyz chính xác là 4 ký tự số hệ 16) ký tự UTF16 (Unicode, 16 bit) U+wxyz trong trật tự byte cục bộ
\Ustuvwxyz	(trong đó stuvwxyz chính xác là 8 ký tự số hệ 16) được giữ lại cho một mở rộng Unicode giả cho 32 bit
\v	thẻ tab thẳng đứng, như trong C
\xhhh	(trong đó hhh là tuần tự bất kỳ của các ký tự số hệ 16) ký tự mà giá trị theo hệ 16 của nó là 0xhhh (một ký tự duy nhất bất kể có bao nhiêu ký tự số hệ 16 được sử dụng)
\0	ký tự mà giá trị của nó là 0 (byte null)
\xy	(trong đó xy chính xác là 2 ký tự số hệ 8, và không phải là một tham chiếu ngược) ký tự mà giá trị theo hệ 8 của nó là 0xy
\xyz	(trong đó xyz chính xác là 3 ký tự số hệ 8, và không phải là một tham chiếu ngược) ký tự mà giá trị theo hệ 8 của nó là 0xyz

Các ký tự số hệ 16 là 0-9, a-f, và A-F. Các ký tự số hệ 8 là 0-7.

Các thoát khoản vào ký tự luôn được lấy như là các ký tự thông thường. Ví dụ, \135 là] trong

ASCII, nhưng \135 không kết thúc một biểu thức dấu ngoặc vuông.

Bảng 9-16. Các thoát tốc ký lớp của biểu thức thông thường

Thoát	Mô tả
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_] (lưu ý dấu gạch chân được đưa vào)
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_] (lưu ý dấu gạch chân được đưa vào)

Trong các biểu thức dấu ngoặc vuông, \d, \s, và \w đánh mất các dấu ngoặc vuông bên ngoài hơn, và \D, \S và \W là không hợp lệ. (Vì thế, ví dụ, `[a-c\d]` là tương đương với `[a-c[:digit:]]`. Hơn nữa, `[a-c\D]`, nó là tương đương với `[a-c^[:digit:]]`, là không hợp lệ).

Bảng 9-17. Các thoát ràng buộc biểu thức thông thường

Thoát	Mô tả
\A	chỉ khớp ở đầu của chuỗi (xem Phần 9.7.3.5 để thấy điều này khác thế nào, so với ^)
\m	chỉ khớp ở đầu của một từ
\M	chỉ khớp ở cuối của một từ
\y	chỉ khớp ở đầu hoặc cuối của một từ
\Y	chỉ khớp ở một điểm không phải là bắt đầu hoặc kết thúc của một từ
\Z	chỉ khớp ở cuối của chuỗi (xem Phần 9.7.3.5 để thấy điều này khác thế nào so với $)

Một từ được định nghĩa như trong đặc tả của `[[:<:]]` and `[[:>:]]` ở trên. Các thoát ràng buộc là không hợp lệ trong các biểu thức dấu ngoặc vuông.

Bảng 9-18. Các tham chiếu ngược của biểu thức thông thường

Thoát	Mô tả
\m	(trong đó m là một ký tự số không phải là 0) một tham chiếu ngược tới biểu thức con thứ m
\mnn	(trong đó m là một ký tự số không phải là 0, và nn là nhiều hơn vài ký tự số, và giá trị thập phân mnn không lớn hơn so với số lượng các dấu ngoặc đơn đóng bắt được được thấy cho tới nay) một tham chiếu ngược tới biểu thức con thứ mnn

> **Lưu ý**: Có một sự tù mù vốn dĩ giữa các thoát khoản vào của ký tự hệ 8 và các tham chiếu ngược, nó được giải quyết bằng công nghệ tự động sau, như được gợi ý ở trên. Số 0 dẫn trước luôn chỉ ra một thoát hệ 8. Ký tự số không phải là 0, không có ký tự số nào khác theo sau, luôn là một tham chiếu ngược. Sự tuần tự nhiều ký tự số không bắt đầu bằng số 0 được lấy như một tham chiếu ngược nếu nó đi sau một biểu thức con phù hợp (nghĩa là, số đó là trong dãy hợp hệ cho một tham chiếu ngược), và nếu không thì nó được lấy như là hệ 8.

9.7.3.4. Siêu cú pháp của biểu thức thông thường

Bổ sung vào cú pháp chính được mô tả ở trên, có một số dạng đặc biệt và các cơ sở thuộc về cú pháp hỗn hợp có sẵn.

Một RE có thể bắt đầu với 1 trong 2 tiếp đầu ngữ dẫn dắt. Nếu một RE bắt đầu bằng ***:, thì phần còn lại của RE được lấy như là một ARE. (Điều này thường không có tác động trong PostgreSQL, vì các RE được giả thiết sẽ là các ARE; nhưng nó có một hiệu ứng nếu chế độ ERE hoặc BRE từng được các tham số cờ flags chỉ định). Nếu một RE bắt đầu bằng ***=, thì phần còn lại của RE được lấy sẽ là một chuỗi hằng, với tất cả các ký tự được xem là các ký tự thông thường.

Một ARE có thể bắt đầu với các lựa chọn được nhúng: một sự tuần tự (?xyz) (trong đó xyz là một hoặc nhiều hơn các ký tự abc) chỉ định các lựa chọn có tác động tới phần còn lại của RE. Các lựa chọn đó đè lên bất kỳ lựa chọn nào trước đó được xác định - đặc biệt, chúng có thể đè lấn hành vi phân biệt chữ hoa chữ thường được một toán tử regex ngụ ý, hoặc tham số cờ flags đối với một hàm regex. Các ký tự lựa chọn sẵn sàng được chỉ ra trong Bảng 9-19. Lưu ý rằng các ký tự lựa chọn y hệt đó được sử dụng trong các tham số *cờ flags* của các hàm regex.

Bảng 9-19. Các ký tự lựa chọn nhúng ARE

Lựa chọn	Mô tả
b	phần còn lại của RE là một BRE
c	khớp phân biệt chữ hoa chữ thường (đè dạng toán tử)
e	phần còn lại của RE là một ERE
i	khớp phân biệt chữ hoa chữ thường (xem Phần 9.7.3.5) (đè dạng toán tử)
m	đồng nghĩa về lịch sử đối với n
n	khớp dòng mới phân biệt chữ hoa chữ thường (xem Phần 9.7.3.5)
p	khớp dòng mới phân biệt chữ hoa chữ thường một phần (xem Phần 9.7.3.5)
q	phần còn lại của RE là một chuỗi hằng (“được trích”), tất cả các ký tự thông thường
s	khớp dòng mới không phân biệt chữ hoa chữ thường (mặc định)
t	cú pháp chặt chẽ (mặc định; xem bên dưới)
w	khớp phân biệt chữ hoa chữ thường dòng mới ngược một phần (“kỳ lạ”) (xem Phần 9.7.3.5)
x	cú pháp được mở rộng (xem bên dưới)

Các lựa chọn nhúng có hiệu quả trong) kết thúc sự tuần tự. Chúng có thể chỉ xuất hiện ở đầu của một ARE (sau ***: dẫn dắt nếu có).

Bổ sung thêm vào cú pháp thông thường RE (chặt), theo đó tất cả các ký tự là có nghĩa, có một cú pháp được mở rộng, sẵn sàng bằng việc chỉ định lựa chọn x được nhúng. Trong cú pháp được mở rộng, các ký tự dấu trắng trong RE được bỏ qua, như là tất cả các ký tự giữa a# và dòng mới theo sau (hoặc kết thúc của RE). Điều này cho phép tạo đoạn và ghi chú cho một RE phức tạp. Có 3 ngoại lệ cho qui tắc cơ bản đó:

- một ký tự trắng hoặc \ đi trước được giữ lại
- ký tự trắng hoặc # trong một biểu thức dấu ngoặc vuông được giữ lại
- ký tự trắng và các ghi chú không thể xuất hiện trong các ký hiệu nhiều ký tự, như (?:

Vì mục đích này, các ký tự dấu trắng là dấu trắng, các thẻ tab, dòng mới, và bất kỳ ký tự nào thuộc vệ lớp các ký tự trắng space.

Cuối cùng, trong một ARE, bên ngoài các biểu thức dấu ngoặc vuông, tuần tự (?#ttt) (trong đó ttt là bất kỳ văn bản nào không chứa một dấu ngoặc đơn)) là một bình luận, hoàn toàn bị bỏ qua. Một lần nữa, điều này không được phép giữa các ký tự của các ký hiệu nhiều ký tự, như (?:. Các ghi chú như vậy là nhiều hơn một chế tác lịch sử so với một cơ sở hữu dụng, và sử dụng chúng bị phản đối; sử dụng cú pháp được mở rộng thay vào đó.

Không sự mở rộng siêu cú pháp nào là sẵn sàng nếu một dẫn dắt ***= đã chỉ định rằng đầu vào của người sử dụng được đối xử như một chuỗi hằng thay vì như một RE.

9.7.3.5. Các qui tắc khớp của biểu thức thông thường

Trong sự kiện một RE có thể khớp nhiều hơn một chuỗi con của một chuỗi được đưa ra, thì RE khớp với chuỗi con bắt đầu sớm nhất trong chuỗi đó. Nếu RE có thể khớp nhiều hơn một chuỗi con bắt đầu ở điểm đó, thì hoặc sự khớp có khả năng dài nhất hoặc sự khớp có khả năng ngắn nhất sẽ xảy ra, phụ thuộc vào việc liệu RE đó có là *tham* hay *không tham*.

Liệu một RE có là tham hay không được các qui tắc sau đây xác định:

- Hầu hết các nguyên tử, và tất cả các ràng buộc, không có thuộc tính tham (vì chúng không thể khớp lượng biến của văn bản bằng mọi cách).
- Bổ sung thêm các dấu ngoặc đơn xung quanh một RE không làm thay đổi tính tham của nó.
- Nguyên tử có đủ điều kiện với một sự lượng hóa cố định lặp đi lặp lại ({m} hoặc {m}?) có cùng độ tham (có thể là không) như bản thân nguyên tử đó.
- Nguyên tử của một sự lượng hóa với các sự lượng hóa thông thường khác (bao gồm cả {m,n}? với m bằng n) là tham (ưu tiên sự khớp dài nhất).
- Nguyên tử của một sự lượng hóa với một sự lượng hóa không tham (bao gồm cả {m,n}? với m bằng n) là không tham (ưu tiên khớp ngắn nhất).
- Một nhánh - đó là, một RE không có toán tử | mức đỉnh - có độ tham y hệt như hạt nhân đầu tiên có đủ điều kiện trong nó mà có một thuộc tính tham.
- Một RE bao gồm 2 hoặc nhiều nhánh hơn được toán tử | kết nối luôn là tham.

Các qui tắc ở trên có liên quan tới các thuộc tính tham không chỉ với các nguyên tử định lượng cá nhân, mà còn với các chi nhánh và toàn bộ các RE chứa các nguyên tử được định lượng đó. Những gì điều đó có nghĩa là việc khớp được thực hiện theo một cách thức mà nhánh đó, hoặc toàn bộ RE,

khớp với chuỗi con có khả năng dài nhất hoặc ngắn nhất như một tổng thể. Một khi chiều dài của toàn bộ sự khớp được xác định, thì phần của nó mà khớp với bất kỳ biểu thức con cụ thể nào được xác định trên cơ sở của thuộc tính tham của biểu thức phụ đó, với các biểu thức phụ bắt đầu sớm hơn trong RE lấy ưu tiên hơn các biểu thức phụ bắt đầu muộn hơn.

Một ví dụ của những gì điều này có nghĩa:

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Result: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Result: 1
```

Trong trường hợp đầu tiên, RE như một tổng thể là tham vì `Y*` là tham. Nó có thể khớp với đầu ở `Y`, và nó khớp với chuỗi có khả năng dài nhất bắt đầu ở đó, nghĩa là, `Y123`. Kết quả đầu ra được đưa vào các dấu ngoặc đơn phần của nó, hay `123`. Trong trường hợp thứ 2, RE như một tổng thể là không tham vì `Y*?` là không tham.

Nó có thể khớp với đầu ở `Y`, và nó khớp với chuỗi có khả năng ngắn nhất bắt đầu ở đó, nghĩa là, `Y1`. Biểu thức con `[0-9]{1,3}` là tham nhưng nó không thể thay đổi quyết định đối với toàn bộ chiều dài khớp được; vì thế nó bị ép phải khớp chỉ với 1.

Ngắn gọn, khi một RE chứa cả các biểu thức phụ tham và không tham, tổng chiều dài khớp hoặc dài hoặc ngắn nhất có thể, tùy theo thuộc tính được chỉ định cho toàn bộ RE. Các thuộc tính được chỉ định cho các biểu thức phụ chỉ tác động tới bao nhiêu sự khớp đó chúng được phép để "ăn" lẫn nhau một cách tương đối.

Các định lượng `{1,1}` và `{1,1}?` có thể được sử dụng để ép tính tham hoặc không tham, một cách tương ứng, trong một biểu thức phụ hoặc toàn bộ RE.

Các độ dài trùng khớp được đo đếm theo các ký tự, không theo các phần tử đối sánh. Một chuỗi rỗng được xem là dài hơn so với không khớp hoàn toàn. Ví dụ: `bb*` khớp 3 ký tự giữa của `abbbc`; `(week|wee)(night|knights)` khớp tất cả 10 ký tự của các đêm cuối tuần (weeknights); khi (.*).* là khớp được đối với abc biểu thức trong dấu ngoặc đơn khớp tất cả 3 ký tự; và khi (a*)* khớp được đối với bc cả toàn bộ RE và biểu thức con trong dấu ngoặc đơn khớp một chuỗi rỗng.

Nếu việc khớp độc lập với chữ hoa chữ thường được chỉ định, thì hiệu ứng là nhiều nếu tất cả các khác biệt chữ hoa chữ thường đã biến khỏi abc. Khi một abc tồn tại trong nhiều trường hợp xuất hiện như một ký tự thông thường bên ngoài một biểu thức dấu ngoặc vuông, thì nó được biến đổi có hiệu quả thành một biểu thức dấu ngoặc vuông có chứa cả 2 dạng chữ, nghĩa là, x trở thành [xX]. Khi nó xuất hiện bên trong một biểu thức dấu ngoặc vuông, thì tất cả các đối tác chữ hoa chữ thường của nó được bổ sung thêm vào biểu thức dấu ngoặc vuông đó, nghĩa là, [x] trở thành [xX] và [^x] trở thành [^xX].

Nếu việc khớp dòng mới phân biệt chữ hoa chữ thường được chỉ định, thì dấu chấm . và các biểu thức dấu ngoặc vuông có sử dụng dấu mũ ^ sẽ không bao giờ khớp ký tự dòng mới đó (sao cho các sự khớp sẽ không bao giờ xuyên các dòng mới trừ phi RE rõ ràng dàn xếp nó) và ^ và $ sẽ khớp chuỗi rỗng sau và trước một dòng mới một cách tương ứng, bổ sung thêm vào việc khớp ở đầu và

cuối của chuỗi một cách tương ứng. Nhưng thoát ARE \A và \Z tiếp tục chỉ khớp đầu hoặc cuối của chuỗi.

Nếu việc khớp một phần dòng mới phân biệt chữ hoa chữ thường được chỉ định, và các biểu thức dấu ngoặc vuông như với việc khớp phân biệt chữ hoa chữ thường dòng mới, nhưng không phải ^ và $.

Nếu việc khớp phân biệt chữ hoa chữ thường dòng mới một phần ngược lại được chỉ định, thì điều này tác động tới ^ và $ như với việc khớp phân biệt chữ hoa chữ thường dòng mới, nhưng không là dấu chấm (.) và các biểu thức dấu ngoặc vuông. Điều này rất không hữu dụng nhưng được đưa ra vì sự đối xứng.

9.7.3.6. Giới hạn và tính tương thích

Không có giới hạn đặc biệt nào được áp đặt lên chiều dài của các RE trong triển khai này. Tuy nhiên, các chương trình có ý định sẽ có tính khả chuyển cao nên không triển khai các RE dài hơn 256 byte, vì một triển khai tuân thủ POSIX có thể từ chối chấp nhận các RE như vậy.

Tính năng duy nhất của các ARE mà thực sự không tương thích với các ERE POSIX là việc dấu chéo ngược \ không đánh mất tầm quan trọng đặc biệt của nó bên trong các biểu thức dấu ngoặc vuông. Tất cả các tính năng khác của ARE sử dụng cú pháp không hợp lệ hoặc có các hiệu ứng chưa được định nghĩa hoặc chưa được chỉ định trong các ERE POSIX; cú pháp *** của các chỉ dẫn giống như là nằm bên ngoài cú pháp POSIX cho cả các BRE và ERE.

Nhiều mở rộng ARE được mượn từ Perl, nhưng một số từng bị thay đổi để làm rõ thêm chúng, và một ít mở rộng Perl không hiện diện. Các chỗ lưu ý không tương thích bao gồm \b, \B sự thiếu đối xử đặc biệt cho một dòng mới đi sau, bổ sung thêm các biểu thức dấu ngoặc vuông vào những điều bị tác động từ việc khớp các dòng mới có phân biệt chữ hoa chữ thường, và các ngữ nghĩa của việc khớp đối với khớp dài nhất/ngắn nhất (thay vì khớp cái đầu tiên).

2 sự không tương thích đáng kể tồn tại giữa cú pháp của các ARE và ERE được thừa nhận trong các phiên bản trước 7.4 của PostgreSQL.

- Trong các ARE, dấu chéo ngược \ đi theo sau là một ký tự abc hoặc là một ký tự thoát hoặc là một lỗi, trong khi trong các phiên bản trước đó, nó từng chỉ là một cách viết khác cho abc. Điều này sẽ không là một vấn đề gì lớn vì đã không có lý do để viết một sự tuần tự như vậy trong các phiên bản trước đó.
- Trong các ARE, các dấu chéo ngược \ vẫn là một ký tự đặc biệt trong các dấu ngoặc vuông [], vì thế một dấu chéo ngược hằng \ trong một biểu thức dấu ngoặc vuông phải được viết là \\.

9.7.3.7. Biểu thức cơ bản thông thường

Các BRE khác với các ERE trong vài khía cạnh. Trong các BRE, dấu |, + và ? là các ký tự thông thường và không tương đương với chức năng của chúng. Các dấu phân cách cho các ràng buộc là \{

and \}, với { and } tự chúng là các ký tự thông thường. Các dấu ngoặc cho các biểu thức phụ lồng nhau là \(and \), với (and) bản thân chúng là các ký tự thông thường. ^ là một ký tự thông thường ngoại trừ ở đầu của RE hoặc ở đầu của một biểu thức phụ trong các dấu ngoặc, $ là một ký tự thông thường ngoại trừ ở cuối của RE hoặc ở cuối của một biểu thức phụ trong các dấu ngoặc, và * là một ký tự thông thường nếu nó xuất hiện ở đầu của RE hoặc ở đầu của một biểu thức phụ trong các dấu ngoặc (sau một dấu ^ đi trước có thể). Cuối cùng, các tham chiếu ngược một ký tự số duy nhất là sẵn sàng, và \< và \> là đồng nghĩa với [[:<:]] và [[:>:]] một cách tương ứng; không ký tự thoát nào khác là sẵn sàng trong các BRE.

9.8. Hàm định dạng dạng dữ liệu

Các hàm định dạng của PostgreSQL đưa ra một tập hợp các công cụ mạnh để biến đổi các dạng dữ liệu khác nhau (ngày tháng/thời gian, số nguyên, chấm thập phân, số) sang các chuỗi được định dạng và cho việc biến đổi từ các chuỗi được định dạng sang các dạng dữ liệu đặc thù. Bảng 9-20 liệt kê chúng. Các hàm đó tất cả tuân theo một qui ước gọi chung: đối số đầu tiên là giá trị sẽ được định dạng và đối số thứ 2 là một mẫu template xác định định dạng đầu ra hoặc đầu vào.

Một hàm đối số duy nhất to_timestamp cũng sẵn sàng; nó chấp nhận một đối số độ chính xác gấp đôi double precision và biến đổi từ thời Unix (các giây kể từ 1970-01-01 00:00:00+00) thành timestamp với vùng thời gian time zone. (Số nguyên Interger thời kỳ Unix được đưa ra một cách ngầm định cho độ chính xác gấp đôi double precision).

Bảng 9-20. Các hàm định dạng

Hàm	Dạng trả về	Mô tả	Ví dụ
to_char(timestamp, text)	text	biến đổi dấu thời gian sang chuỗi	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	biến đổi khoảng thời gian sang chuỗi	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	biến đổi số nguyên sang chuỗi	to_char(125, '999')
to_char(double precision, text)	text	biến đổi độ chính xác thực/đúp (real/double) sang chuỗi	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	biến đổi số sang chuỗi	to_char(-125.8, '999D99S')
to_date(text, text)	date	biến đổi chuỗi sang ngày tháng	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	biến đổi chuỗi sang số	to_number('12,454.8-', '99G999D9S')
to_timestamp(text, text)	timestamp with time zone	biến đổi chuỗi sang dấu thời gian	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(double precision)	timestamp with time zone	biến đổi từ thời kỳ Unix sang dấu thời gian	to_timestamp(1284352323)

Trong một chuỗi mẫu temlate đầu ra to_char, có các mẫu nhất định được thừa nhận và được thay thế bằng các dữ liệu được định dạng phù hợp dựa vào giá trị được đưa ra. Bất kỳ văn bản nào mà không

phải là một mẫu template đơn giản sẽ được sao chép y nguyên. Tương tự, trong một chuỗi mẫu template đầu vào (đối với các hàm khác), các mẫu template xác định các giá trị sẽ được chuỗi dữ liệu đầu vào cung cấp.

Bảng 9-21 chỉ ra các mẫu template có sẵn cho việc định dạng các giá trị ngày tháng và thời gian.

Bảng 9-21. Các mẫu template cho việc định dạng ngày tháng / thời gian

Mẫu	Mô tả
HH	giờ của ngày (01-12)
HH12	giờ của ngày (01-12)
HH24	giờ của ngày (00-23)
MI	phút (00-59)
SS	giây (00-59)
MS	mili giây (000-999)
US	mili giây (000000-999999)
SSSS	các giây đi qua nửa đêm (0-86399)
AM, am, PM hoặc pm	chỉ số meridiem, chỉ nửa ngày đầu và nửa ngày sau (không có các dấu chấm)
A.M. , a.m. , P.M. hoặc p.m.	chỉ số meridiem, chỉ nửa ngày đầu và nửa ngày sau (có các dấu chấm)
Y,YYY	năm (4 hoặc nhiều hơn ký tự số) với dấu phẩy
YYYY	Năm (4 và nhiều hơn các ký tự số)
YYY	3 ký tự cuối của năm
YY	2 ký tự số cuối của năm
Y	ký tự số cuối của năm
IYYY	năm theo ISO (4 và nhiều hơn ký tự số)
IYY	3 ký tự số cuối của năm theo ISO
IY	2 ký tự số cuối của năm theo ISO
I	ký tự số cuối của năm theo ISO
BC , bc , AD hoặc ad	chỉ số kỷ nguyên (không có các dấu chấm)
B.C. , b.c. , A.D. hoặc a.d.	chỉ số kỷ nguyên (có các dấu chấm)
MONTH	tên tháng chữ hoa đầy đủ (dấu trắng - được thêm vào tới 9 ký tự)
Month	tên tháng chữ hoa đầy đủ (dấu trắng - được thêm vào tới 9 ký tự)
month	tên tháng chữ thường đầy đủ (dấu trắng - được thêm vào tới 9 ký tự)
MON	tên tháng chữ hoa viết tắt (3 ký tự tiếng Anh, độ dài được bản địa hóa khác nhau)
Mon	tên tháng chữ hoa viết tắt (3 ký tự tiếng Anh, độ dài được bản địa hóa khác nhau)
mon	tên tháng chữ thường viết tắt (3 ký tự tiếng Anh, độ dài được bản địa hóa khác nhau)
MM	số tháng (01-12)
DAY	tên ngày chữ hoa đầy đủ (dấu trắng - được thêm vào cho tới 9 ký tự)
Day	tên ngày chữ hoa đầy đủ (dấu trắng - được thêm vào cho tới 9 ký tự)
day	tên ngày chữ thường đầy đủ (dấu trắng - được thêm vào cho tới 9 ký tự)

Mẫu	Mô tả
DY	tên ngày chữ hoa viết tắt (3 ký tự tiếng Anh, độ dài được bản địa hóa khác nhau)
Dy	tên ngày chữ hoa viết tắt (3 ký tự tiếng Anh, độ dài được bản địa hóa khác nhau)
dy	tên ngày chữ thường viết tắt (3 ký tự tiếng Anh, độ dài được bản địa hóa khác nhau)
DDD	ngày của năm (001-366)
IDDD ISO	ngày của năm (001-371; ngày 1 của năm là Thứ hai của tuần đầu tiên theo ISO).
DD	ngày của tháng (01-31)
D	ngày của tuần, Chủ nhật (1) tới Thứ bảy (7)
ID ISO	ngày của tuần, Thứ hai (1) tới Thứ bảy (7)
W	tuần của tháng (1-5) (tuần đầu tiên bắt dầu vào ngày đầu tiên của tháng)
WW	số tuần của năm (1-53) (tuần đầu tiên bắt đầu vào ngày đầu tiên của năm).
IW ISO	số tuần của năm (01-53; Thứ năm đầu tiên của năm mới là trong tuần đầu tiên)
CC	thế kỷ (2 ký tự số) (thế kỷ 21 bắt đầu vào 01/01/2001)
J	Ngày theo lịch Julian (các ngày kể từ 24/11 năm 4714 BC vào lúc nửa đêm)
Q	Quý (bị bỏ qua bởi to_date và to_timestamp)
RM	tháng viết chữ hoa theo các số La mã (I-XII; I = Tháng 1)
rm	tháng viết chữ thường theo các số La mã (i-xii; i = Tháng 1)
TZ	tên vùng thời gian chữ hoa
tz	tên vùng thời gian chữ thường

Các sửa đổi có thể được áp dụng cho bất kỳ mẫu template nào để chỉnh sửa hành vi của nó. Ví dụ, FMMonth là mẫu Month với sửa đổi FM. Bảng 9-22 chỉ ra các mẫu sửa đổi cho định dạng ngày tháng / thời gian.

Bảng 9-22. Các sửa đổi mẫu template cho việc định dạng ngày tháng / thời gian

Sửa đổi	Mô tả	Ví dụ
Tiền tố FM	chế độ điền (ngăn chặn các khoảng trống thêm vào và các số 0)	FMMonth
Hậu tố TH	hậu tố số thông thường chữ hoa	DDTH, e.g., 12TH
Hậu tố th	hậu tố số thông thường chữ thường	DDth, e.g., 12th
Tiền tố FX	lựa chọn tổng thể định dạng cố định (xem các lưu ý sử dụng)	FX Month DD Day
Tiền tố TM	chế độ dịch (in các tên ngày và tháng được bản địa hóa dựa vào lc_time)	TMMonth
Hậu tố SP	chế độ chính tả (không được triển khai)	DDSP

Sử dụng các lưu ý cho việc định dạng ngày tháng / thời gian:

- FM ngăn chặn các số 0 đi đầu và các dấu trắng đi sau mà có thể nếu không sẽ được bổ sung thêm vào để làm thành đầu ra của một mẫu có độ rộng cố định. Trong PostgreSQL, FM chỉ sửa đổi đặc tả tiếp theo, trong khi trong Oracle thì FM tác động tới tất cả các đặc tả tiếp sau, và các sửa đổi FM được lặp đi lặp lại, chuyển qua lại bật / tắt chế độ điền.

- TM không bao gồm các khoảng trắng đi sau.
- Nhiều khoảng trắng to_timestamp và to_date skip trong chuỗi đầu vào trừ phi lựa chọn FX được sử dụng. Ví dụ, to_timestamp('2000 JUN', 'YYYY MON') làm việc, nhưng to_timestamp('2000 JUN', 'FXYYYY MON') trả về một lỗi vì to_timestamp kỳ vọng một khoảng trống duy nhất. FX phải được chỉ định như là khoản đầu tiên trong mẫu template.
- Văn bản thông thường được phép trong các mẫu template to_char và sẽ là đầu ra theo nghĩa đen. Bạn có thể đặt một chuỗi con trong các dấu ngoặc kép để ép nó sẽ được dịch như là văn bản hằng thậm chí nếu nó có chứa các từ khóa mẫu. Ví dụ trong '"Hello Year "YYYY', YYYY sẽ được thay thế bằng dữ liệu năm, nhưng Y duy nhất trong Year thì sẽ không. Trong to_date, to_number, và to_timestamp, các chuỗi trong các dấu ngoặc kép bỏ qua một số ký tự đầu vào có trong chuỗi đó, nghĩa là "XX" bỏ qua 2 ký tự đầu vào.
- Nếu bạn muốn có một dấu ngoặc đơn ở đầu ra thì bạn phải đặt trước nó một dấu chéo ngược, ví dụ E'\\"YYYY Month\\"'. (2 dấu chéo ngược là cần thiết vì dấu chéo ngược có ý nghĩa đặc biệt khi sử dụng cú pháp chuỗi thoát).
- Biến đổi YYYY từ chuỗi sang timestamp hoặc date có một hạn chế khi việc xử lý các năm với hơn 4 ký tự số. Bạn phải sử dụng một số ký tự không phải ký tự số hoặc mẫu template sau YYYY, nếu không thì năm sẽ luôn được dịch như là 4 ký tự số. Ví dụ (với năm 20000): to_date('200001131', 'YYYYMMDD') sẽ được dịch như là một năm 4 ký tự số; thay vì sử dụng một sự phân tách không phải ký tự số sau năm, như to_date('20000-1131', 'YYYY-MMDD') hoặc to_date('20000Nov31', 'YYYYMonDD').
- Trong biến đổi từ chuỗi sang timestamp hoặc date, trường CC (thế kỷ) bị bỏ qua nếu có một trường YYY, YYYY hoặc Y,YYY. Nếu CC được sử dụng với YY hoặc Y thì năm được tính như là (CC-1)*100+YY.
- Một ngày của tuần theo ISO (khác với ngày theo Gregorian) có thể được chỉ định cho to_timestamp và to_date theo 1 trong 2 cách:
  - Năm, tuần và ngày trong tuần: ví dụ to_date('2006-42-4', 'IYYY-IW-ID') sẽ trả về ngày 19/10/2006. Nếu bạn làm mờ ngày trong tuần thì nó được giả thiết sẽ là 1 (Thứ hai).
  - Năm và ngày của năm: ví dụ to_date('2006-291', 'IYYY-IDDD') cũng sẽ trả về 19/10/2006.

  Việc cố gắng xây dựng một ngày bằng việc sử dụng một sự pha trộn các trường tuần theo ISO và ngày tháng theo Gregorian là không có ý nghĩa, và sẽ gây ra lỗi. Trong ngữ cảnh của một năm theo ISO, khái niệm của một "tháng" hoặc "ngày của tháng" không có ý nghĩa. Trong ngữ cảnh của một năm theo Gregoria, thì tuần theo ISO không có ý nghĩa. Những người sử dụng nên tránh việc trộn các đặc tả ngày tháng theo Gregorian và ISO.
- Trong một biến đổi từ chuỗi sang timestamp, các giá trị mili giây (MS) hoặc mili giây (US) sẽ được sử dụng như là các ký tự số cho các giây sau dấu chấm thập phân. Ví dụ

to_timestamp('12:3', 'SS:MS') không phải là 3 mili giây, mà là 300, vì biến đổi đó tính nó như là 12 + 0.3 giây. Điều này có nghĩa là đối với định dạng SS:MS, các giá trị đầu vào 12:3 , 12:30 và 12:300 chỉ định số các mili giây y hệt. Để có 3 mili giây, phải sử dụng 12:003, để biến đổi tính như là 12 + 0.003 = 12.003 giây.

Đây là một ví dụ phức tạp hơn: to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US') là 15 giờ, 12 phút và 2 giây + 20 mili giây + 1230 micro giây = 2.021230 giây.

- Việc đánh số ngày của tuần của to_char(..., 'ID') khớp với hàm extract(isodow from ...), nhưng to_char(..., 'D') không khớp với việc đánh số ngày của extract(dow from ...).
- to_char(interval) định dạng HH và HH12 như được chỉ ra trong một chiếc đồng hồ 12 giờ, nghĩa là các giờ 0 và 36 cho ra như là 12, trong khi HH24 cho đầu ra giá trị giờ đầy đủ, nó có thể vượt qua 23 trong một khoảng thời gian.

Bảng 9-23 chỉ ra các mẫu template sẵn sàng cho việc định dạng các giá trị số.

Bảng 9-23. Các mẫu template cho việc định dạng số

Mẫu	Mô tả
9	giá trị với số các ký tự số được chỉ định
0	giá trị với các số 0 dẫn đầu
. (dấu chấm)	dấu thập phân
, (dấu phẩy)	dấu phân cách nhóm (hàng ngàn)
PR	giá trị âm trong các dấu ngoặc nhọn
S	dấu được neo cho số (sử dụng bản địa)
L	ký hiệu hiện hành (sử dụng bản địa)
D	dấu chấm thập phân (sử dụng bản địa)
G	phân cách nhóm (sử dụng bản địa)
MI	dấu trừ ở vị trí được chỉ định (nếu số đó nhỏ hơn 0)
PL	dấu cộng ở vị trí được chỉ định (nếu số đó lớn hơn 0)
SG	dấu cộng/trừ ở vị trí được chỉ định
RN	số La mã (đầu vào giữa 1 và 3999)
TH hoặc th	hậu tố của số thông thường
V	số các ký tự số dịch chuyển được chỉ định (xem các lưu ý)
EEEE	số mũ cho các ký hiệu khoa học

Sử dụng các lưu ý cho việc định dạng số:

- Một ký hiệu được định dạng bằng việc sử dụng Sài Gòn, PL, hoặc MI không được neo cho số; ví dụ, to_char(-12,'MI9999') tạo ra '- 12' nhưng to_char(-12, 'S9999') tạo ra ' -12'. Triển khai của Oracle không cho phép sử dụng MI trước 9, nhưng thay vào đó yêu cầu là 9 đi trước MI.
- 9 dẫn tới một giá trị với số các ký tự số y hệt như có 9 s. Nếu một ký tự số không sẵn sàng

thì nó đưa ra một dấu trắng.

- TH không biến đổi các giá trị ít hơn 0 và không biến đổi các số thập phân.
- PL, SG, và TH là các mở rộng của PostgreSQL.
- V nhân có hiệu lực các giá trị đầu vào với 10^n, trong đó n là số các ký tự số sau V. to_char không hỗ trợ sử dụng V được kết hợp với một dấu thập phân (nghĩa là, 00.9V99 là không được phép).
- EEEE (ký hiệu khoa học) không thể được sử dụng trong sự kết hợp với bất kỳ mẫu hoặc sửa đổi định dạng nào khác ngoài các mẫu ký tự số và dấu thập phân, và phải đặt ở cuối của chuỗi định dạng (như, 9.99EEEE là một mẫu hợp lệ).

Các sửa đổi nhất định có thể được áp dụng cho bất kỳ mẫu template nào để sửa hành vi của nó. Ví dụ, FM9999 là mẫu 9999 với sửa đổi FM. Bảng 9-24 chỉ ra các mẫu sửa đổi cho việc định dạng số.

Bảng 9-24. Các sửa đổi mẫu template cho việc định dạng số

Sửa đổi	Mô tả	Ví dụ
Tiền tố FM	chế độ điền (ngăn chặn các dấu trắng và 0 thêm vào)	FM9999
Hậu tố TH	hậu tố số thông thường chữ hoa	999TH
Hậu tố th	hậu tố số thông thường chữ thường	999th

Bảng 9-25 chỉ ra một số ví dụ sử dụng hàm to_char.

Bảng 9-25. Các ví dụ hàm to_char

Biểu thức	Kết quả
to_char(current_timestamp, ’Day, DD HH12:MI:SS’)	’Tuesday , 06 05:39:18’
to_char(current_timestamp, ’FMDay, FMDD HH12:MI:SS’)	’Tuesday, 6 05:39:18’
to_char(-0.1, ’99.99’)	’ -.10’
to_char(-0.1, ’FM9.99’)	’-.1’
to_char(0.1, ’0.9’)	’ 0.1’
to_char(12, ’9990999.9’)	’ 0012.0’
to_char(12, ’FM9990999.9’)	’0012.’
to_char(485, ’999’)	’ 485’
to_char(-485, ’999’)	’-485’
to_char(485, ’9 9 9’)	’ 4 8 5’
to_char(1485, ’9,999’)	’ 1,485’
to_char(1485, ’9G999’)	’ 1 485’
to_char(148.5, ’999.999’)	’ 148.500’
to_char(148.5, ’FM999.999’)	’148.5’
to_char(148.5, ’FM999.990’)	’148.500’

Biểu thức	Kết quả
to_char(148.5, ’999D999’)	’ 148,500’
to_char(3148.5, ’9G999D999’)	’ 3 148,500’
to_char(-485, ’999S’)	’485-’
to_char(-485, ’999MI’)	’485-’
to_char(485, ’999MI’)	’485 ’
to_char(485, ’FM999MI’)	’485’
to_char(485, ’PL999’)	’+485’
to_char(485, ’SG999’)	’+485’
to_char(-485, ’SG999’)	’-485’
to_char(-485, ’9SG99’)	’4-85’
to_char(-485, ’999PR’)	’<485>’
to_char(485, ’L999’)	’DM 485
to_char(485, ’RN’)	’ CDLXXXV’
to_char(485, ’FMRN’)	’CDLXXXV’
to_char(5.2, ’FMRN’)	’V’
to_char(482, ’999th’)	’ 482nd’
to_char(485, ’"Good number:"999’)	’Good number: 485’
to_char(485.8, ’"Pre:"999" Post:" .999’)	’Pre: 485 Post: .800’
to_char(12, ’99V999’)	’ 12000’
to_char(12.4, ’99V999’)	’ 12400’
to_char(12.45, ’99V9’)	’ 125’
to_char(0.0004859, ’9.99EEEE’)	’ 4.86e-04’

9.9. Hàm và toán tử ngày tháng / thời gian

Bảng 9-27 chỉ ra các hàm có sẵn cho việc xử lý giá trị ngày tháng / thời gian, với các chi tiết xuất hiện trong các tiểu phần sau đây. Bảng 9-26 minh họa các hành vi của các toán tử số học cơ bản (+, *, …). Đối với các hàm định dạng, hãy tham chiếu tới Phần 9.8. Bạn nên làm quen với thông tin cơ bản về các dạng dữ liệu ngày tháng / thời gian từ Phần 8.5.

Tất cả các hàm và toán tử được mô tả bên dưới mà lấy các đầu vào time hoặc timestamp thực sự có 2 phương án: một là lấy time với time zone hoặc timestamp với time zone, và hai là lấy time không với time zone hoặc timestamp không với time zone. Vì tính không bền, các phương án đó không được chỉ ra một cách riêng rẽ. Hơn nữa, các toán tử + và * đi theo các cặp giao hoán (ví dụ cả ngày tháng + số nguyên và số nguyên + ngày tháng), chúng ta sẽ chỉ đưa ra một trong các cặp như vậy.

Bảng 9-26. Các toán tử ngày tháng / thời gian

Toán tử	Ví dụ	Kết quả
+	date ’2001-09-28’ + integer ’7’	date ’2001-10-05’
+	date ’2001-09-28’ + interval ’1 hour’	timestamp ’2001-09-28 01:00:00’
+	date ’2001-09-28’ + time ’03:00’	timestamp ’2001-09-28 03:00:00’
+	interval ’1 day’ + interval ’1 hour’	interval ’1 day 01:00:00’
+	timestamp ’2001-09-28 01:00’ + interval ’23 hours’	timestamp ’2001-09-29 00:00:00’
+	time ’01:00’ + interval ’3 hours’	time ’04:00:00’
-	- interval ’23 hours’	interval ’-23:00:00’
-	date ’2001-10-01’ - date ’2001-09-28’	integer ’3’ (days)
-	date ’2001-10-01’ - integer ’7’	date ’2001-09-24’
-	date ’2001-09-28’ - interval ’1 hour’	timestamp ’2001-09-27 23:00:00’
-	time ’05:00’ - time ’03:00’	interval ’02:00:00’
-	time ’05:00’ - interval ’2 hours’	time ’03:00:00’
-	timestamp ’2001-09-28 23:00’ - interval ’23 hours’	timestamp ’2001-09-28 00:00:00’
-	interval ’1 day’ - interval ’1 hour’	interval ’1 day -01:00:00’
-	timestamp ’2001-09-29 03:00’ - timestamp ’2001-09-27 12:00’	interval ’1 day 15:00:00’
*	900 * interval ’1 second’	interval ’00:15:00’
*	21 * interval ’1 day’	interval ’21 days’
*	double precision ’3.5’ * interval ’1 hour’	interval ’03:30:00’
/	interval ’1 hour’ / double precision ’1.5’	interval ’00:40:00’

Bảng 9-27. Các hàm ngày tháng / thời gian

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
age(timestamp, timestamp)	interval	Đối số phép trừ, tạo ra một kết quả “biểu tượng” mà sử dụng các năm và tháng	age(timestamp ’2001-04-10’, timestamp ’1957-06-13’)	43 năm 9 tháng 27 ngày
age(timestamp)	interval	Trừ đi từ ngày hiện hành current_date (lúc nửa đêm)	age(timestamp ’1957-06-13’)	43 năm 8 tháng 3 ngày
clock_timestamp()	timestamp with time zone	Ngày tháng và thời gian hiện hành (các thay đổi khi thực hiện lệnh); xem Phần 9.9.4		
current_date	date	Ngày tháng hiện hành; xem Phần 9.9.4		
current_time	time with time zone	Thời gian hiện hành của ngày; xem Phần 9.9.4		
current_timestamp	timestamp with time zone	Ngày tháng và thời gian hiện hành (bắt đầu giao dịch hiện hành); xem Phần 9.9.4		
date_part(text,	double	Lấy trường con (giống như	date_part(’hour’, timestamp	20

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
timestamp)	precision	phép trừ); xem Phần 9.9.1	’2001-02-16 20:38:40’)	
date_part(text, interval)	double precision	Lấy trường con (giống như phép trừ); xem Phần 9.9.1	date_part(’month’, interval ’2 years 3 months’)	3
date_trunc(text, timestamp)	timestamp	Cắt bớt về độ chính xác được chỉ định; xem Phần 9.9.2	date_trunc(’hour’, timestamp ’2001-02-16 20:38:40’)	2001-02-16 20:00:00
extract(field from timestamp)	double precision	Lấy trường con; xem Phần 9.9.1	extract(hour from timestamp ’2001-02-16 20:38:40’)	20
extract(field from interval)	double precision	Lấy trường con; xem Phần 9.9.1	extract(month from interval ’2 years 3 months’)	3
isfinite(date)	boolean	Kiểm thử ngày tháng xác định (not +/-infinity)	isfinite(date ’2001-02-16’)	true
isfinite(timestamp)	boolean	Kiểm thử ngày tháng xác định (not +/-infinity)	isfinite(timestamp ’2001-02-16 21:28:30’)	true
isfinite(interval)	boolean	Kiểm thử khoảng xác định	isfinite(interval ’4 hours’)	true
justify_days(interval)	interval	Tinh chỉnh khoảng sao cho giai đoạn thời gian 30 ngày được thể hiện như các tháng	justify_days(interval ’35 days’)	1 mon 5 days
justify_hours(interval)	interval	Chỉnh khoảng sao cho giai đoạn 24 giờ được thể hiện như các ngày	justify_hours(interval ’27 hours’)	1 day 03:00:00
justify_interval(interval)	interval	Tinh chỉnh khoảng bằng việc sử dụng justify_days và justify_hours, với các tinh chỉnh ký hiệu bổ sung	justify_interval(interval ’1 mon -1 hour’)	29 days 23:00:00
localtime	time	Thời gian của ngày hiện hành; xem Phần 9.9.4		
localtimestamp	timestamp	Ngày tháng và thời gian hiện hành (bắt đầu giao dịch hiện hành); xem Phần 9.9.4		
now()	timestamp with time zone	Ngày tháng và thời gian hiện hành (bắt đầu giao dịch hiện hành); xem Phần 9.9.4		
statement_timestamp()	timestamp with time zone	Ngày tháng và thời gian hiện hành (bắt đầu giao dịch hiện hành); xem Phần 9.9.4		
timeofday()	text	Ngày tháng và thời gian hiện hành (như clock_timestamp, nhưng như một chuỗi văn bản); xem Phần 9.9.4		

Bổ sung thêm vào các hàm đó, toán tử OVERLAPS của SQL được hỗ trợ:

(*start1*, *end1*) OVERLAPS (*start2*, *end2*)
(*start1*, *length1*) OVERLAPS (*start2*, *length2*)

Biểu thức này là đúng khi 2 giai đoạn thời gian (được các điểm cuối của chúng xác định) chồng lấn

lên nhau, là sai khi chúng không chồng lấn nhau. Các điểm cuối có thể được chỉ định như các cặp ngày tháng, thời gian hoặc các dấu thời gian; hoặc như một ngày tháng, thời gian, hoặc dấu thời gian đi sau là một khoảng thời gian. Khi một cặp các giá trị được cung cấp, hoặc bắt đầu hoặc kết thúc có thể được viết trước; OVERLAPS tự động lấy giá trị sớm nhất của cặp đó như là bắt đầu. Mỗi giai đoạn thời gian được xem xét để thể hiện khoảng nửa mở bắt đầu start <= time < end (kết thúc), trừ phi start và end là như nhau trong trường hợp đó nó thể hiện thời gian duy nhất ngay lập tức đó. Điều này có nghĩa là sự việc mà 2 giai đoạn thời gian chỉ một điểm kết thúc nói chung không chồng lấn nhau.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: true

SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: false

SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: false

SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: true
```

Khi thêm một giá trị interval vào (hoặc bớt đi một giá trị interval khỏi) một giá trị dấu thời gian với vùng thời gian (timestamp with time zone), thành phần các ngày tăng (hoặc giảm) ngày của timestamp with time zone bằng số các ngày được chỉ định. Khắp những thay đổi tiết kiệm ánh sáng ban ngày (với vùng thời gian các phiên được thiết lập tới một vùng thời gian mà nhận biết được DST), điều này có nghĩa là interval '1 day' (khoảng thời gian 1 ngày) không nhất thiết bằng với interval '24 hours' (khoảng thời gian 24 giờ). Ví dụ, với vùng thời gian các phiên được thiết lập về CST7DT, thì timestamp with time zone '2005-04-02 12:00-07' + interval '1 day' sẽ tạo ra timestamp with time zone '2005-04-03 12:00-06', trong khi việc thêm interval '24 hours' vào timestamp with time zone y hệt ban đầu sẽ tạo ra timestamp with time zone '2005-04-03 13:00-06', như có một sự thay đổi trong thời gian tiết kiệm ánh sáng ban ngày lúc 2005-04-03 02:00 theo vùng thời gian CST7CDT.

Lưu ý có thể sẽ có sự mù mờ trong các tháng được age trả về vì các tháng khác nhau có các số ngày khác nhau. Tiếp cận của PostgreSQL sử dụng tháng từ đầu của 2 ngày khi tính toán phần của các tháng. Ví dụ, age('2004-06-01', '2004-04-30') sử dụng tháng Tư để có 1 mon 1 day, trong khi sử dụng tháng Năm có thể có 1 mon 2 day vì tháng Năm có 31 ngày, trong khi tháng Tư chỉ có 30 ngày.

9.9.1. EXTRACT, date_part

EXTRACT(field FROM source)

Hàm extract truy xuất các trường con như năm hoặc giờ từ các giá trị ngày tháng / thời gian. Trong đó source phải là một biểu thức giá trị dạng timestamp, time hoặc interval. (Các biểu thức dạng date đưa ra cho timestamp và có thể vì thế cũng được sử dụng). Trường field là một mã định danh hoặc

chuỗi mà chọn trường nào để trích xuất từ giá trị nguồn. Hàm extract trả về các giá trị dạng double precision. Sau đây là các tên trường hợp lệ:

century

Thế kỷ (century) sẽ

SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Result: 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 21

Thế kỷ đầu tiên bắt đầu lúc 0001-01-01 00:00:00 AD, dù chúng đã không biết nó khi đó. Định nghĩa này áp dụng cho tất cả các nước có lịch Gregorian. Không có thế kỷ 0, bạn đi từ thế kỷ -1 sang thế kỷ 1. Nếu bạn không đồng ý với điều này, xin viết yêu cầu của bạn cho: Giáo hoàng, Nhà thờ Saint-Peter ở Roma, Vatican.

Các phiên bản PostgreSQL trước 8.0 đã không tuân theo việc đánh số thế kỷ như qui ước, mà chỉ trả về trường năm được chia cho 100.

day

Trường ngày (của tháng) (1-31)

SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16

decade

Trường năm chia cho 10

SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200

dow

Ngày của tuần là từ Chủ nhất (0) tới thứ Bảy (6)

SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5

Lưu ý rằng việc đánh số ngày của tuần của hàm extract khác với việc đánh số của hàm to_char(..., 'D').

doy

Ngày của năm (1-365/366)

SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47

epoch

Đối với các giá trị date và timestamp, số các giây từ 1970-01-01 00:00:00 UTC (có thể là số âm); đối với các giá trị khoảng interval, tổng số các giây trong khoảng

SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
Result: 982384720.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800

Đây là cách mà bạn có thể biến đổi một giá trị thời đại ngược trở về một dấu thời gian:

SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720.12 * INTERVAL '1 second';

(Hàm to_timestamp tuân theo qui ước bên trên).

hour

Trường giờ (0-23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP ’2001-02-16 20:38:40’);
Result: 20
```

isodow

Ngày của tuần như là thứ Hai (1) tới thứ Bảy (7)

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP ’2001-02-18 20:38:40’);
Result: 7
```

Điều này là đúng hệt nhưng ngoại trừ thứ Bảy. Điều này khớp với việc đánh số ngày của tuần theo ISO 8601.

isoyear

Năm theo ISO 8601 có ngày tháng nằm trong (không áp dụng được cho các khoảng)

```
SELECT EXTRACT(ISOYEAR FROM DATE ’2006-01-01’);
Result: 2005

SELECT EXTRACT(ISOYEAR FROM DATE ’2006-01-02’);
Result: 2006
```

Mỗi năm theo ISO bắt đầu vào thứ Hai của tuần gồm tuần thứ 4 của tháng 1, nên trong tháng 1 sớm hoặc tháng 12 muộn thì năm theo ISO có thể khác với năm theo Gregorian. Xem trường tuần week để có thêm thông tin.

Trường này không có trong các phiên bản PostgreSQL trước 8.3.

microseconds

Trường giây, bao gồm cả các phần thập phân, được nhân lên với 1.000.000; lưu ý rằng điều này bao gồm cả các giây đầy đủ

```
SELECT EXTRACT(MICROSECONDS FROM TIME ’17:12:28.5’);
Result: 28500000
```

millennium

Thiên niên kỷ

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP ’2001-02-16 20:38:40’);
Result: 3
```

Các năm trong những năm 1900 là thuộc thiên niên kỷ thứ 2. Thiên niên kỷ thứ 3 đã bắt đầu vào ngày 01/01/2001.

Các phiên bản PostgreSQL trước 8.0 đã không tuân theo việc đánh số thiên niên kỷ này, mà chỉ trả về trường năm được chia cho 1.000.

milliseconds

Trường các giây, bao gồm cả các phần thập phân, được nhân với 1.000. Lưu ý là điều này bao gồm các giây đầy đủ.

```
SELECT EXTRACT(MILLISECONDS FROM TIME ’17:12:28.5’);
```

Result: 28500

minute

Trường phút (0-59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

month

Đối với các giá trị timestamp, số tháng trong năm (1-12); đối với các giá trị khoảng interval số các tháng, modulo 12 (0-11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Result: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1
```

quarter

Quý của năm (1-4) mà ngày tháng là trong

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

second

Trường giây, bao gồm các phần thập phân (0-59)

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.5
```

timezone

Bù trừ vùng thời gian từ UTC, được đo bằng giây. Các giá trị dương tương ứng với các vùng thời gian phía đông của UTC, các giá trị âm cho các vùng phía tây của UTC.

timezone_hour

Thành phần giờ của bù trừ vùng thời gian

timezone_minute

Thành phần phút của bù trừ vùng thời gian

week

Số tuần của năm mà ngày trong đó. Theo định nghĩa của ISO 8601, các tuần bắt đầu vào thứ Hai và tuần đầu của năm có ngày mồng 4 tháng 1 của năm đó. Nói cách khác, thứ Năm đầu tiên của một năm là trong tuần số 1 của năm đó.

Định nghĩa của ISO, khả năng của nó cho các ngày tháng của tháng 1 sớm sẽ là một phần của tuần thứ 52 hoặc 53 của năm trước đó, và đối với các ngày tháng của tháng 12 muộn sẽ là một phần của tuần đầu tiên của năm tiếp sau. Ví dụ, 2005-01-01 là một phần của tuần thứ 53 của năm 2004, và 2006-01-01 là một phần của tuần thứ 52 của năm 2005, trong khi 2012-12-31 là một phần của tuần đầu tiên của năm 2013. Được khuyến cáo sử dụng trường isoyear cùng với week để có được các kết quả nhất quán.

SELECT EXTRACT(WEEK FROM TIMESTAMP ’2001-02-16 20:38:40’);
Result: 7

year

Trường năm. Hãy ghi nhớ trong đầu không có AD 0, nên việc trừ các năm BC từ các năm AD nên được thực hiện thận trọng.

SELECT EXTRACT(YEAR FROM TIMESTAMP ’2001-02-16 20:38:40’);
Result: 2001

Hàm extract trước hết có ý định cho việc xử lý tính toán. Đối với các giá trị ngày tháng / thời gian để hiển thị, xem Phần 9.8.

Hàm date_part tạo thành mô hình tương đương với Ingres truyền thống với hàm tiêu chuẩn SQL.

extract:
date_part(’field’, source)

Lưu ý rằng ở đây tham số trường field cần phải là một giá trị chuỗi, không phải là một tên. Các tên trường hợp lệ cho date_part là y hệt như với extract.

Là 60 nếu các giây của năm nhuận được hệ điều hành triển khai

SELECT date_part(’day’, TIMESTAMP ’2001-02-16 20:38:40’);
Result: 16

SELECT date_part(’hour’, INTERVAL ’4 hours 3 minutes’);
Result: 4

9.9.2. date_trunc

Hàm date_trunc về khái niệm là tương tự như hàm trunc cho các số.

date_trunc(’field’, source)

Trong đó source là một biểu thức giá trị dạng timestamp hoặc interval. (Các giá trị dạng date và time được đưa ra một cách tự động cho timestamp hoặc interval, một cách tương ứng). Trường field sẽ chọn độ chính xác nào đó để cắt bớt giá trị đầu vào. Giá trị trả về là dạng timestamp hoặc interval với tất cả các trường ít quan trọng hơn so với được chọn và được thiết lập về 0 (hoặc 1, cho ngày và tháng).

Các giá trị hợp lệ cho field là:

microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade
century
millennium

Các ví dụ:

SELECT date_trunc(’hour’, TIMESTAMP ’2001-02-16 20:38:40’);

```
Result: 2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00
```

9.9.3. AT TIME ZONE

Cấu trúc AT TIME ZONE cho phép các biến đổi các dấu thời gian cho các vùng thời gian khác nhau. Bảng 9-28 chỉ ra các biến thể của nó.

Bảng 9-28. Các phương án AT TIME ZONE

Biểu thức	Dạng trả về	Mô tả
timestamp without time zone AT TIME ZONE zone	dấu thời gian có vùng thời gian	Đối xử với dấu thời gian được đưa ra *không có vùng thời gian* được nằm trong vùng thời gian được chỉ định
timestamp with time zone AT TIME ZONE zone	dấu thời gian không có vùng thời gian	Biến đổi dấu thời gian được đưa ra *có vùng thời gian* cho vùng thời gian mới, không có chỉ định vùng thời gian
time with time zone AT TIME ZONE zone	Thời gian có vùng thời gian	Biến đổi thời gian cho trước *có vùng thời gian* thành vùng thời gian mới

Trong các biểu thức đó, vùng thời gian mong muốn zone có thể được chỉ định hoặc như một chuỗi văn bản (như, 'PST') hoặc như một khoảng (như, INTERVAL '-08:00'). Trong trường hợp văn bản, một tên vùng thời gian có thể được chỉ định theo bất kỳ cách nào được mô tả trong Phần 8.5.3.

Các ví dụ (giả thiết vùng thời gian địa phương là PST8PDT):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Result: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Result: 2001-02-16 18:38:40
```

Ví dụ đầu lấy một dấu thời gian mà không có vùng thời gian và dịch nó thành thời gian MST (UTC-7), mà sau đó được biến đổi thành PST (UTC-8) để hiển thị. Ví dụ 2 lấy một dấu thời gian được chỉ định trong EST (UTC-5) và biến đổi nó thành thời gian địa phương theo MST (UTC-7).

Hàm timezone(zone, timestamp) là tương đương với cấu trúc tuân thủ SQL timestamp AT TIME ZONE zone.

9.9.4. Date/Time hiện hành

PostgreSQL đưa ra một số hàm trả về các giá trị có liên quan tới ngày tháng và giờ hiện hành.

Các hàm tiêu chuẩn SQL đó tất cả đều trả về các giá trị dựa vào thời điểm đầu của giao dịch hiện hành:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
```

```
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

CURRENT_TIME và CURRENT_TIMESTAMP đưa ra các giá trị với vùng thời gian; LOCALTIME và LOCALTIMESTAMP đưa ra các giá trị không có vùng thời gian.

CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME và LOCALTIMESTAMP có thể tùy ý lấy một tham số chính xác, nó làm cho việc làm tròn số cho nhiều ký tự số thập phân trong trường giây. Không có một tham số chính xác, thì kết quả đó được đưa ra với độ chính xác đầy đủ có sẵn. Một số ví dụ:

```
SELECT CURRENT_TIME;
Result: 14:39:53.662522-05

SELECT CURRENT_DATE;
Result: 2001-12-23

SELECT CURRENT_TIMESTAMP;
Result: 2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Result: 2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
Result: 2001-12-23 14:39:53.662522
```

Vì các hàm đó trả về thời điểm đầu của giao dịch hiện hành, các giá trị của chúng không thay đổi trong quá trình giao dịch đó. Điều này được xem như là một chức năng: ý định là để cho phép một giao dịch duy nhất sẽ có một ký hiệu nhất quán về thời điểm "hiện hành", sao cho nhiều sửa đổi bên trong cùng giao dịch có được dấu thời gian y hệt.

Lưu ý: Các hệ cơ sở dữ liệu khác có thể ưu tiên các giá trị đó thường xuyên hơn.

PostgreSQL cũng đưa ra các hàm trả về thời điểm đầu của lệnh hiện hành, cũng như thời điểm hiện hành thực sự trong sự việc khi hàm đó được gọi. Danh sách đầy đủ các hàm thời gian không theo tiêu chuẩn SQL là:

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

transaction_timestamp() là tương đương với CURRENT_TIMESTAMP, nhưng được đặt tên để phản ánh rõ ràng những gì nó trả về. statement_timestamp() trả về giá trị y hệt trong quá trình lệnh đầu tiên của một giao dịch, mà có thể khác trong quá trình các lệnh tiếp theo. clock_timestamp() trả về thời gian hiện hành thực sự, và vì thế giá trị của nó thay đổi thậm chí trong một lệnh SQL duy nhất. timeofday() là một hàm lịch sử của PostgreSQL. Giống như clock_timestamp(), nó trả về thời gian hiện thành thực tế, nhưng như một chuỗi text được định dạng thay vì một giá trị dấu thời gian với vùng thời gian. now() là một tương đương truyền thống của PostgreSQL cho transaction_timestamp().

Tất cả các dạng dữ liệu ngày tháng / thời gian cũng chấp nhận giá trị hằng đặc biệt now để chỉ định ngày tháng và thời gian hiện hành (một lần nữa, được dịch như là thời điểm bắt đầu của giao dịch). Vì thế cả 3 thứ sau đều trả về cùng một kết quả:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now'; -- không đúng để sử dụng với DEFAULT
```

> **Mẹo**: Bạn không muốn sử dụng dạng thứ 3 khi chỉ định một mệnh đề mặc định DEFAULT trong khi tạo một bảng. Hệ thống sẽ biến đổi now thành một timestamp ngay khi hằng số đó được phân tích, sao cho khi giá trị mặc định là cần thiết, thì thời gian tạo bảng có thể được sử dụng! 2 dạng đầu sẽ không được đánh giá cho tới khi giá trị mặc định được sử dụng, vì chúng là các lời gọi hàm. Vì thế chúng sẽ đưa ra hành vi mong muốn của việc mặc định để chèn thời gian vào hàng.

9.9.5. Thực thi trễ

Hàm sau đây là sẵn sàng để làm trễ sự thực thi tiến trình của máy chủ:

```
pg_sleep(seconds)
```

pg_sleep tạo ra sự ngủ tiến trình của phiên làm việc hiện hành cho tới các giây seconds trôi qua. seconds là một giá trị dạng độ chính xác gấp đôi double precision, nên các trễ phần thập phân của giây có thể được chỉ định. Ví dụ:

```
SELECT pg_sleep(1.5);
```

> **Lưu ý**: Giải pháp hiệu quả của khoảng ngủ là đặc thù theo nền tảng; 0.01 giây là một giá trị chung. Trễ ngủ sẽ ít nhất miễn là được chỉ định. Nó có thể là dài hơn, phụ thuộc vào các yếu tố như tải của các máy chủ.

> **Cảnh báo**
>
> Hãy chắc chắn là phiên làm việc của bạn không giữ các khóa nhiều hơn so với cần thiết khi gọi pg_sleep. Nếu không thì các phiên khác có thể phải chờ tiến trình ngủ của bạn, làm chậm toàn bộ hệ thống.

9.10. Hàm hỗ trợ đánh số Enum

Đối với các dạng đánh số (được mô tả trong Phần 8.7), có vài hàm cho phép việc lập trình sạch sẽ hơn mà không có các giá trị đặc biệt mã cứng (hard code) của một dạng đánh số enum. Chúng được liệt kê trong Bảng 9-29. Các ví dụ giả thiết một dạng đánh số enum được tạo ra như:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

Bảng 9-29. Các hàm hỗ trợ đánh số Enum

Hàm	Mô tả	Ví dụ	Kết quả ví dụ
enum_first(anyenum)	Trả về giá trị đầu của dạng enum đầu vào	enum_first(null::rainbow)	red
enum_last(anyenum)	Trả về giá trị cuối cùng của đạng enum đầu vào	enum_last(null::rainbow)	purple

Hàm	Mô tả	Ví dụ	Kết quả ví dụ
enum_range(anyenum)	Trả về tất cả các giá trị của dạng enum đầu vào trong một mảng có trật tự	enum_range(null::rainbow)	{red, orange, yellow, green, blue, purple}
enum_range(anyenum, anyenum)	Trả về dải giữa 2 giá trị enum được đưa ra, như một mảng có trật tự. Các giá trị đó phải từ cùng một dạng enum y hệt. Nếu tham số đầu là null, thì kết quả sẽ bắt đầu với giá trị đầu tiên của dạng enum. Nếu tham số thứ 2 là null, thì kết quả sẽ kết thúc với giá trị cuối của dạng enum.	enum_range('orange'::rainbow, 'green'::rainbow)	{orange, yellow, green}
		enum_range(NULL, 'green'::rainbow)	{red, orange, yellow, green}
		enum_range('orange'::rainbow, NULL)	{orange, yellow, green, blue, purple}

Lưu ý rằng ngoại trừ dạng 2 đối số của enum_range, các hàm đó không quan tâm tới giá trị đặc thù được truyền tới chúng; chúng chỉ quan tâm về dạng dữ liệu được khai báo của nó. Hoặc null hoặc một giá trị đặc thù của dạng đó có thể được truyền qua, với cùng y hệt kết quả. Phổ biến hơn để áp dụng các hàm đó cho một cột của bảng hoặc đối số của hàm hơn là cho một tên dạng được viết tay như được các ví dụ gợi ý.

9.11. Hàm và toán tử hình học

Các dạng point, box, lseg, line, path, polygon và circle có một tập hợp lớn các hàm và toán tử bẩm sinh hỗ trợ, được chỉ ra trong Bảng 9-30, và Bảng 9-32.

> **Chú ý**
>
> Lưu ý rằng toán tử "hệt như", ~ =, thể hiện ký hiệu bằng nhau thông thường cho các dạng the point, box, polygon và circle. Một số dạng đó cũng có một toán tử =, nhưng = so với chỉ các lĩnh vực bằng nhau. Các toán tử so sánh tuyến tính khác (<= và …) hơn nữa so sánh các lĩnh vực cho các dạng đó.

Bảng 9-30. Các toán tử địa lý

Toán tử	Mô tả	Ví dụ
+	Chuyển đổi	box '((0,0),(1,1))' + point '(2.0,0)'
-	Chuyển đổi	box '((0,0),(1,1))' - point '(2.0,0)'
*	Co dãn / Xoay	box '((0,0),(1,1))' * point '(2.0,0)'
/	Co dãn / Xoay	box '((0,0),(2,2))' / point '(2.0,0)'
#	Điểm hoặc hộp giao nhau	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Số các điểm trên đường hoặc đa giác	# '((1,0),(0,1),(-1,0))'
@-@	Chiều dài hoặc chu vi	@-@ path '((0,0),(1,0))'
@@	Tâm	@@ circle '((0,0),10)'
##	Điểm gần nhất với toán hạng đầu tiên trong toán hạng thứ 2	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	Khoảng giữa	circle '((0,0),1)' <-> circle '((5,0),1)'

Toán tử	Mô tả	Ví dụ
&&	Chồng lấn? (Một điểm chung làm điều này thành đúng).	box '((0,0),(1,1))' && box '((0,0),(2,2))'
<<	Nghiêm trái?	circle '((0,0),1)' << circle '((5,0),1)'
>>	Nghiêm phải?	circle '((5,0),1)' >> circle '((0,0),1)'
&<	Không mở rộng sang phải?	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	Không mở rộng sang trái?	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<|	Nghiêm dưới?	box '((0,0),(3,3))' <<| box '((3,4),(5,5))'
|>>	Nghiêm trên?	box '((3,4),(5,5))' |>> box '((0,0),(3,3))'
&<|	Không mở rộng lên trên?	box '((0,0),(1,1))' &<| box '((0,0),(2,2))'
|&>	Không mở rộng xuống dưới?	box '((0,0),(3,3))' |&> box '((0,0),(2,2))'
<^	Ở dưới (cho phép động tới)?	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	Ở trên (cho phép động tới)?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Giao nhau?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	Nằm ngang?	?- lseg '((-1,0),(1,0))'
?-	Dóng hàng nằm ngang?	point '(1,0)' ?- point '(0,0)'
?|	Thẳng đứng?	?| lseg '((-1,0),(1,0))'
?|	Dóng hàng thẳng đứng?	point '(0,1)' ?| point '(0,0)'
?-|	Vuông góc?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
?||	Song song?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@>	Bao gồm?	circle '((0,0),2)' @> point '(1,1)'
<@	Nằm trong hoặc nằm trên?	point '(1,1)' <@ circle '((0,0),2)'
~=	Hệt như?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Lưu ý: Trước PostgreSQL 8.2, các toán tử chứa @> và <@ từng được gọi một cách tương ứng là ~ và @. Các tên đó vẫn sẵn sàng, nhưng được cắt ngắn và cuối cùng sẽ bị loại bỏ.

Bảng 9-31. Các hàm địa lý

Hàm	Dạng trả về	Mô tả	Ví dụ
area(object)	double precision	vùng diện tích	area(box '((0,0),(1,1))')
center(object)	point	tâm	center(box '((0,0),(1,2))')
diameter(circle)	double precision	đường kính vòng tròn	diameter(circle '((0,0),2.0)')
height(box)	double precision	kích thước chiều dọc của hộp	height(box '((0,0),(1,1))')
isclosed(path)	boolean	đường khép kín?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	đường mở?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	double precision	độ dài	length(path '((-1,0),(1,0))')
npoints(path)	int	số các điểm	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	số các điểm	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	biến đổi đường thành khép kín	pclose(path '[(0,0),(1,1),(2,0)]')

Hàm	Dạng trả về	Mô tả	Ví dụ
popen(path)	path	biến đổi đường thành mở	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	bán kính vòng tròn	radius(circle '((0,0),2.0)')
width(box)	double precision	kích thước nằm ngang của hộp	width(box '((0,0),(1,1))')

Bảng 9-32. Các hàm biến đổi dạng hình học

Hàm	Dạng trả về	Mô tả	Ví dụ
box(circle)	box	vòng tròn thành hộp	box(circle '((0,0),2.0)')
box(point, point)	box	các điểm thành hộp	box(point '(0,0)', point '(1,1)')
box(polygon)	box	đa giác thành hộp	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	hộp thành vòng tròn	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	tâm và bán kính đường tròn	circle(point '(0,0)', 2.0)
circle(polygon)	circle	đa giác thành đường tròn	circle(polygon '((0,0),(1,1),(2,0))')
lseg(box)	lseg	đường chéo hộp thành đoạn thẳng	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	các điểm thành đoạn thẳng	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	path	đa giác thành đường	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	xây dựng điểm	point(23.4, -44.5)
point(box)	point	tâm của hộp	point(box '((-1,0),(1,0))')
point(circle)	point	tâm đường tròn	point(circle '((0,0),2.0)')
point(lseg)	point	tâm của đoạn thẳng	point(lseg '((-1,0),(1,0))')
point(polygon)	point	tâm của đa giác	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	hộp thành đa giác 4 điểm	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	đường tròn thành đa giác 12 điểm	polygon(circle '((0,0),2.0)')
polygon(npts, circle)	polygon	đường tròn thành đa giác npts điểm	Polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	đường thành đa giác	polygon(path '((0,0),(1,1),(2,0))')

Có khả năng truy cập 2 số thành phần của một điểm dù điểm đó từng là một mảng với các chỉ số 0 và 1. Ví dụ, nếu t.p là một cột các điểm point thì SELECT p[0] FROM t truy xuất tọa độ X và UPDATE t SET p[1] = ... thay đổi tọa độ Y. Theo cách y hệt, một giá trị dạng box hoặc lseg có thể được truy xuất như một mảng của 2 giá trị điểm point.

Hàm area làm việc cho các dạng box, circle và path. Hàm area chỉ làm việc ở dạng dữ liệu path nếu các điểm trong path là không giao nhau. Ví dụ, path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH sẽ không làm việc; tuy nhiên, path '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH sẽ làm việc. Nếu khái niệm của việc giao nhau đối lại với path không giao nhau là dễ lẫn, thì hãy vẽ cả 2 đường path ở trên cạnh nhau trong một mẩu giấy đồ họa.

9.12. Hàm và toán tử địa chỉ mạng

Bảng 9-33 chỉ các toán tử có sẵn cho các dạng cidr và inet. Các toán tử <<, <<=, >> và >>= kiểm thử sự đưa vào mạng con (subnet). Chúng chỉ xem các phần mạng của 2 địa chỉ (bỏ qua bất kỳ phần chủ (host) nào) và xác định liệu một mạng có là y hệt với hoặc mạng con của mạng khác hay không.

Bảng 9-33. Các toán tử cidr và inet

Toán tử	Mô tả	Ví dụ
<	nhỏ hơn	inet ’192.168.1.5’ < inet ’192.168.1.6’
<=	nhỏ hơn hoặc bằng	inet ’192.168.1.5’ <= inet ’192.168.1.5’
=	bằng	inet ’192.168.1.5’ = inet ’192.168.1.5’
>=	lớn hơn hoặc bằng	inet ’192.168.1.5’ >= inet ’192.168.1.5’
>	lớn hơn	inet ’192.168.1.5’ > inet ’192.168.1.4’
<>	không bằng	inet ’192.168.1.5’ <> inet ’192.168.1.4’
<<	nằm trong	inet ’192.168.1.5’ << inet ’192.168.1/24’
<<=	nằm trong hoặc bằng	inet ’192.168.1/24’ <<= inet ’192.168.1/24’
>>	bao gồm	inet ’192.168.1/24’ >> inet ’192.168.1.5’
>>=	bao gồm hoặc bằng	inet ’192.168.1/24’ >>= inet ’192.168.1/24’
~	KHÔNG (NOT) theo bit	~ inet ’192.168.1.6’
&	VÀ (AND) theo bit	inet ’192.168.1.6’ & inet ’0.0.0.255’
|	HOẶC (OR) theo bit	itwise OR inet ’192.168.1.6’ | inet ’0.0.0.255’
+	cộng	inet ’192.168.1.6’ + 25
-	trừ	inet ’192.168.1.43’ - 36
-	trừ	inet ’192.168.1.43’ - inet ’192.168.1.19’

Bảng 9-34 chỉ các hàm sẵn sàng để sử dụng với các dạng cidr và inet. Các hàm abbrev, host và text ban đầu có ý định đưa ra các định dạng hiển thị tùy chọn thay thế.

Bảng 9-34. Các hàm cidr và inet

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
abbrev(inet)	text	định dạng hiển thị rút gọn như văn bản	abbrev(inet ’10.1.0.0/16’)	10.1.0.0/16
abbrev(cidr)	text	định dạng hiển thị rút gọn như văn bản	abbrev(cidr ’10.1.0.0/16’)	10.1/16
broadcast(inet)	inet	địa chỉ phát cho mạng	broadcast(’192.168.168/24’)	192.255/24
family(inet)	int	họ địa chỉ chính xác; 4 cho IPv4, 6 cho IPv6	family(’::1’)	6
host(inet)	text	địa chỉ IP chính xác như văn bản	host(’192.168.1.5/24’)	192.168.1.5
hostmask(inet)	inet	xây dựng mặt nạ chủ cho mạng	hostmask(’192.168.2.30/30’)	0.0.2.30
masklen(inet)	int	độ dài chính xác của mặt nạ	masklen(’192.168.1.5/24’)	24
netmask(inet)	inet	xây dựng mặt nạ cho mạng	netmask(’192.168.1.5/24’)	255.255.255.0

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
network(inet)	cidr	phần địa chỉ mạng chính xác	network(’192.168.1.5/24’)	192.168.0.1/24
set_masklen(inet, int)	inet	thiết lập độ dài mặt nạ cho giá trị inet	set_masklen(’192.168.1.5/24’, 16)	192.168.1.5/16
set_masklen(cidr, int)	cidr	thiết lập độ dài mặt nạ cho giá trị cidr	set_masklen(’192.168.1.0/24’::cidr, 16)	192.168.0.0/16
text(inet)	text	trích địa chỉ IP và độ dài mặt nạ như văn bản	text(inet ’192.168.1.5’)	192.168.1.5/32

Bất kỳ giá trị cidr nào cũng có thể đưa ra cho inet sự ngầm định hoặc rõ ràng; vì thế các hàm được chỉ ra ở trên như việc vận hành trong inet cũng làm việc được trong các giá trị của cidr. (Ở những nơi có các hàm riêng rẽ cho intet và cidr, thì đó là vì hành vi phải là khác đối với 2 trường hợp đó). Hơn nữa, được cho phép đưa ra một giá trị inet cho cidr. Khi điều này được thực hiện, thì bất kỳ bit nào ở bên phải của mặt nạ cũng âm thầm biến thành 0 để tạo ra một giá trị cidr hợp lệ. Hơn nữa, bạn có thể đưa ra một giá trị văn bản cho inet hoặc cidr bằng việc sử dụng cú pháp đưa ra thông thường, ví dụ: inet(expression) hoặc colname::cidr.

Bảng 9-35 chỉ ra các hàm sẵn sàng để sử dụng với dạng macaddr. Hàm trunc(macaddr) trả về một địa chỉ MAC với 3 byte cuối cùng được đặt về 0. Điều này có thể được sử dụng để liên kết tiền tố còn lại với một nhà sản xuất.

Bảng 9-35. Các hàm macaddr

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
trunc(macaddr)	macaddr	thiết lập 3 byte cuối về 0	trunc(macaddr ’12:34:56:78:90:ab’)	12:34:56:00:00:00

Dạng macaddr cũng hỗ trợ các toán tử quan hệ tiêu chuẩn (>, <=, …) theo trật tự từ điển.

9.13. Hàm và toán tử tìm kiếm văn bản

Bảng 9-36, Bảng 9-37 và Bảng 9-38 tóm tắt các hàm và toán tử sẽ được cung cấp cho việc tìm kiếm toàn văn. Xem Chương 12 để có giải thích chi tiết về cơ sở tìm kiếm toàn văn của PostgreSQL.

Bảng 9-36. Các toán tử tìm kiếm văn bản

Toán tử	Mô tả	Ví dụ	Kết quả
@@	tsvector có khớp tsquery ?	to_tsvector(’fat cats ate rats’) @@ to_tsquery(’cat & rat’)	t
@@@	từ đồng nghĩa được yêu cầu cho @@	to_tsvector(’fat cats ate rats’) @@@ to_tsquery(’cat & rat’)	t
||	ghép nối tsvectors	’a:1 b:2’::tsvector || ’c:1 d:2 b:3’::tsvector	’a’:1 ’b’:2,5 ’c’:3 ’d’:4
&&	AND tsquerys cùng nhau	’fat | rat’::tsquery && ’cat’::tsquery	(’fat’ | ’rat’) & ’cat’
||	OR tsquerys cùng nhau	’fat | rat’::tsquery || ’cat’::tsquery	(’fat’ | ’rat’) | ’cat’

Toán tử	Mô tả	Ví dụ	Kết quả
!!	phủ định một tsquery	!! 'cat'::tsquery	!'cat'
@>	tsquery chứa thứ khác?	'cat'::tsquery @> 'cat & rat'::tsquery	f
<@	tsquery nằm bên trong?	'cat'::tsquery <@ 'cat & rat'::tsquery	t

Lưu ý: Các toán tử chứa tsquery chỉ xem xét các từ được liệt kê trong 2 truy vấn, bỏ qua các toán tử kết hợp.

Bổ sung thêm vào các toán tử được chỉ ra trong bảng, các toán tử so sánh B-tree (cây-B) thông thường (=, <, …) được xác định cho các dạng tsvector và tsquery. Chúng không thật hữu dụng để tìm kiếm văn bản nhưng cho phép, ví dụ, các từ độc nhất được xây dựng trong các cột các dạng đó.

Bảng 9-37. Các hàm tìm kiếm văn bản

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
to_tsvector([config regconfig ,] document text)	tsvector	giảm văn bản tài liệu về tsvector	to_tsvector('english', 'The Fat Rats')	'fat':2 'rat':3
length(tsvector)	integer	số các từ trong tsvector	length('fat:2,4 cat:3 rat:5A'::tsvector)	3
setweight(tsvector, "char")	tsvector	chỉ định trọng số cho từng yếu tố của tsvector	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')	'cat':3A 'fat':2A,4A 'rat':5A
strip(tsvector)	tsvector	loại bỏ các vị trí và trọng số khỏi tsvector	strip('fat:2,4 cat:3 rat:5A'::tsvector)	'cat' 'fat' 'rat'
to_tsquery([config regconfig ,] query text)	tsquery	bình thường hóa các từ và biến đổi sang tsquery	to_tsquery('english', 'The & Fat & Rats')	'fat' & 'rat'
plainto_tsquery([config regconfig ,] query text)	tsquery	tạo tsquery bỏ qua các dấu ngắt từ	plainto_tsquery('english ', 'The Fat Rats')	'fat' & 'rat'
numnode(tsquery)	integer	số từ cộng các toán tử trong tsquery	numnode('(fat & rat) | cat'::tsquery)	5
querytree(query tsquery)	text	lấy phần tạo được chỉ số của một tsquery	querytree('foo & ! bar'::tsquery)	'foo'
ts_rank([weights float4[],] vector tsvector, query tsquery [, normalization integer])	float4	xếp hạng tài liệu cho truy vấn	ts_rank(textsearch, query)	0.818
ts_rank_cd([weights float4[],] vector tsvector, query tsquery [, normalization integer])	float4	xếp hạng tài liệu cho truy vấn bằng việc sử dụng mật độ bao trùm	ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)	2.01317
ts_headline([config regconfig,] document text, query tsquery [, options text])	text	hiển thị một sự khớp truy vấn	ts_headline('x y z', 'z'::tsquery)	x y z
ts_rewrite(query tsquery, target tsquery, substitute tsquery)	tsquery	thay thế đích bằng cụm thay thế trong truy vấn	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)	'b' & ('foo' | 'bar')
ts_rewrite(query tsquery, select	tsquery	thay thế bằng việc sử	SELECT ts_rewrite('a	'b' & ('foo' | 'bar')

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
text)		dụng các đích và thay thế từ một lệnh SELECT	& b’::tsquery, ’SELECT t,s FROM aliases’)	
get_current_ts_config regconfig()	text	lấy cấu hình tìm kiếm mặc định	get_current_ts_config()	english
tsvector_update_trigger()	trigger	hàm trigger cho cập nhật cột tsvector tự động CREATE TRIGGER ...	tsvector_update_trigger(tsvcol, ’pg_catalog.swedish’, title, body)	
tsvector_update_trigger_column()	trigger	hàm trigger cho cập nhật cột tsvector tự động CREATE TRIGGER ...	tsvector_update_trigger _column(tsvcol, configcol, title, body)	

Lưu ý: Tất cả các hàm tìm kiếm văn bản chấp nhận một đối số tùy chọn regconfig sẽ sử dụng cấu hình được default_text_search_config chỉ định khi đối số đó bị làm mờ đi.

Các hàm trong Bảng 9-38 được liệt kê riêng rẽ vì chúng thường không được sử dụng trong các hoạt động tìm kiếm văn bản hàng ngày. Chúng là hữu dụng cho sự phát triển và việc dò lỗi các cấu hình tìm kiếm mới các văn bản.

Bảng 9-38. Các hàm gỡ lỗi tìm kiếm văn bản

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
ts_debug([config regconfig,] document text, OUT alias text, OUT description text, OUT token text, OUT dictionaries regdictionary[], OUT dictionary regdictionary, OUT lexemes text[])	setof record	kiểm thử một cấu hình	ts_debug(’english’, ’The Brightest supernovaes’)	(asciiword,"Word, all ASCII",The, {english_stem},eng
ts_lexize(dict regdictionary, token text)	text[]	kiểm thử một từ điển	ts_lexize(’english_stem’, ’stars’)	{star}
ts_parse(parser_name text, document text, OUT tokid integer, OUT token text)	setof record	kiểm thử một trình phân tích cú pháp	ts_parse(’default’, ’foo - bar’)	(1,foo) ...
ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)	setof record	kiểm thử một trình phân tích cú pháp	ts_parse(3722, ’foo - bar’)	(1,foo) ...
ts_token_type(parser_name text, OUT tokid integer, OUT alias text, OUT description text)	setof record	lấy các dạng token được trình phân tích cú pháp xác định	ts_token_type(’default’)	(1, ascii word,"Word, all ASCII") ...
ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text, OUT description text)	setof record	lấy các dạng token được trình phân tích cú pháp xác định	ts_token_type(3722)	(1, ascii word,"Word, all ASCII") ...

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
ts_stat(sqlquery text, [weights text,] OUT word text, OUT ndoc integer, OUT nentry integer)	setof record	lấy số thống kể của một cột tsvector	ts_stat('SELECT vector from apod')	(foo,10,15) ...

9.14. Hàm XML

Các hàm và các biểu thức giống hàm được mô tả trong phần này vận hành với các giá trị dạng xml. Hãy kiểm tra Phần 8.13 để có thông tin về dạng xml. Các biểu thức giống hàm xmlparse và xmlserialize cho việc biến đổi sang và từ dạng xml không được lặp lại ở đây. Sử dụng nhiều hàm đó đòi hỏi cài đặt phải được xây dựng cấu hình - với libxml.

9.14.1. Tạo nội dung XML

Một tập hợp các hàm và các biểu thức giống hàm là sẵn sàng cho việc tạo ra nội dung XML từ các dữ liệu SQL. Như vậy, chúng đặc biệt phù hợp cho việc định dạng các kết quả truy vấn trong các tài liệu XML cho việc xử lý trong các ứng dụng máy trạm.

9.14.1.1. xmlcomment

xmlcomment(text)

Hàm xmlcomment tạo ra một giá trị XML có chứa một bình luận XML với văn bản được chỉ định như là nội dung. Văn bản đó không thể có "--" hoặc kết thúc với một "-" sao cho cấu trúc kết quả là một bình luận XML hợp lệ. Nếu đối số là null, thì kết quả là null.

Ví dụ:

```
SELECT xmlcomment('hello');

xmlcomment
--------------
<!--hello-->
```

9.14.1.2. xmlconcat

xmlconcat(xml[, ...])

Hàm xmlconcat ghép nối một danh sách các giá trị XML riêng rẽ để tạo một giá trị duy nhất có chứa một phân mảnh nội dung XML. Các giá trị null được làm mờ; kết quả chỉ là null nếu không có các đối số nonnull.

Ví dụ:

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

xmlconcat
----------------------
<abc/><bar>foo</bar>
```

Các khai báo XML, nếu có, được kết hợp như sau. Nếu tất cả các giá trị đối số có cùng khai báo phiên bản XML, thì phiên bản đó được sử dụng trong kết quả đó, nếu không thì không phiên bản nào được sử dụng. Nếu tất cả các giá trị đối số có giá trị khai báo đứng một mình là “yes” (có), thì giá trị đó được sử dụng trong kết quả. Nếu tất cả các giá trị đối số có một giá trị khai báo đứng một mình và ít nhất một giá trị là “no” (không), thì nó được sử dụng trong kết quả đó. Kết quả khác sẽ không có khai báo đứng một mình. Nếu kết quả được xác định phải yêu cầu một khai báo đứng một mình nhưng không khai báo phiên bản nào có, thì một khai báo phiên bản với phiên bản 1.0 sẽ được sử dụng vì XML đòi hỏi một khai báo XML có chứa một khai báo phiên bản. Việc mã hóa các khai báo bị bỏ qua và bị loại bỏ trong tất cả các trường hợp.

Ví dụ:

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>');

xmlconcat
-----------------------------------
<?xml version="1.1"?><foo/><bar/>
```

9.14.1.3. xmlelement

```
xmlelement(name name [, xmlattributes(value [AS attname] [, ... ])] [, content, ...])
```

Biểu thức xmlelement tạo ra một phần tử XML, được trao tên, các thuộc tính và nội dung. Ví dụ:

```
SELECT xmlelement(name foo);

xmlelement
------------
<foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

xmlelement
------------------
<foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

xmlelement
-------------------------------------
<foo bar="2007-01-26">content</foo>
```

Các tên phần tử và thuộc tính mà không phải là các tên XML hợp lệ được thoát bằng việc thay thế các ký tự có lỗi bằng sự tuần tự _xHHHH_, trong đó HHHH là điểm mã Unicode của ký tự theo ký hiệu hexadecimal (hệ 16). Ví dụ:

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

xmlelement
----------------------------------
<foo_x0024_bar a_x0026_b="xyz"/>
```

Một tên thuộc tính rõ ràng cần không được chỉ định nếu giá trị thuộc tính đó là một tham chiếu cột, trong trường hợp đó tên cột sẽ được sử dụng như là tên thuộc tính một cách mặc định. Trong các trường hợp khác, thuộc tính đó phải được đưa ra một cái tên rõ ràng. Vì thế ví dụ này là hợp lệ:

```
CREATE TABLE test (a xml, b xml);
```

```
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

Nhưng các ví dụ này thì không:

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

Nội dung của phần tử, nếu được chỉ định, sẽ được định dạng theo dạng dữ liệu của nó. Nếu nội dung là bản thân dạng xml, thì các tài liệu XML phức tạp có thể được cấu thành. Ví dụ:

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
        xmlelement(name abc),
        xmlcomment('test'),
        xmlelement(name xyz));

xmlelement
----------------------------------------------
<foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Nội dung của các dạng khác sẽ được định dạng trong các dữ liệu ký tự XML hợp lệ. Điều này có nghĩa đặc biệt rằng các ký tự <, > và & sẽ được biến đổi thành các thực thể. Các dữ liệu nhị phân (dạng dữ liệu bytea) sẽ được thể hiện ở mã base64 hoặc hex, phụ thuộc vào thiết lập của tham số cấu hình xmlbinary. Hành vi đặc biệt đối với các dạng dữ liệu cá nhân được kỳ vọng sẽ tiến hóa để phù hợp với các dạng dữ liệu của SQL và PostgreSQL với đặc tả Sơ đồ XML, ở điểm mà một mô tả chính xác hơn sẽ xuất hiện.

9.14.1.4. xmlforest

```
xmlforest(content [AS name] [, ...])
```

Biểu thức xmlforest tạo ra một rừng (tuần tự) XML các phần tử bằng việc sử dụng các tên và nội dung được đưa ra.

Các ví dụ:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

xmlforest
------------------------------
<foo>abc</foo><bar>123</bar>

SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

xmlforest
-------------------------------------------------------------------------------------------
<table_name>pg_authid</table_name><column_name>rolname</column_name>
<table_name>pg_authid</table_name><column_name>rolsuper</column_name>
...
```

Như được thấy trong ví dụ thứ 2, tên phần tử có thể làm mờ nếu giá trị nội dung là một tham chiếu cột, trong trường hợp đó thì tên cột được sử dụng mặc định. Nếu không, một tên phải được chỉ định.

Các tên phần tử mà không là các tên XML hợp lệ sẽ được thoát như được chỉ ra cho xmlelement ở trên. Tương tự, dữ liệu nội dung được thoát để làm cho nội dung XML hợp lệ, trừ phi nó là dạng xml rồi.

Lưu ý rằng các rừng XML không là các tài liệu XML hợp lệ nếu chúng bao gồm nhiều hơn một phần tử, nên có thể là hữu dụng để bọc các biểu thức xmlforest trong xmlelement.

9.14.1.5. xmlpi

xmlpi(name target [, content])

Biểu thức xmlpi tạo ra một lệnh xử lý XML. Nội dung, nếu có, phải không chứa sự tuần tự ký tự ?>.

Ví dụ:

```
SELECT xmlpi(name php, 'echo "hello world";');

xmlpi
-----------------------------
<?php echo "hello world";?>
```

9.14.1.6. xmlroot

xmlroot(xml, version text | no value [, standalone yes|no|no value])

Biểu thức xmlroot chỉnh các thuộc tính nút gốc root của một giá trị XML. Nếu một phiên bản được chỉ định, nó thay thế giá trị trong khai báo phiên bản nút gốc đó; nếu một thiết lập đứng một mình được chỉ định, thì nó thay thế giá trị trong khai báo đứng một mình của nút gốc đó.

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),

version '1.0', standalone yes);
xmlroot
----------------------------------------
<?xml version="1.0" standalone="yes"?>
<content>abc</content>
```

9.14.1.7. xmlagg

xmlagg(xml)

Hàm xmlagg là, không giống như các hàm khác được mô tả ở đây, một hàm tổng hợp. Nó ghép nối các giá trị đầu vào tới lời gọi của hàm tổng hợp, rất giống xmlconcat làm, ngoại trừ là sự ghép nối xảy ra khắp các hàng thay vì khắp các biểu thức trong một hàng duy nhất. Xem Phần 9.18 để có thêm thông tin về các hàm tổng hợp.

Ví dụ:

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;

xmlagg
----------------------
<foo>abc</foo><bar/>
```

Để xác định trật tự ghép nối, một mệnh đề ORDER BY có thể được thêm vào lời gọi tổng hợp như được mô tả trong Phần 4.2.7. Ví dụ:

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;

xmlagg
----------------------
<bar/><foo>abc</foo>
```

Tiếp cận phi tiêu chuẩn sau đây được sử dụng để khuyến cáo trong các phiên bản trước, và có thể vẫn còn hữu dụng trong các trường hợp đặc biệt:

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;

xmlagg
----------------------
<bar/><foo>abc</foo>
```

9.14.1.8. Từ vị XML

xml IS DOCUMENT

Biểu thức IS DOCUMENT trả về đúng (true) nếu giá trị đối số XML là một tài liệu XML phù hợp, sai (false) nếu nó không (đó là, đây là một sự phân mảnh nội dung), hoặc null nếu đối số là null. Xem Phần 8.13 về sự khác biệt giữa các tài liệu và các phân mảnh nội dung.

9.14.2. Xử lý XML

Để xử lý các giá trị dạng dữ liệu xml, PostgreSQL đưa ra hàm xpath, nó đánh giá các biểu thức Xpath 1.0.

xpath(xpath, xml[, nsarray])

Hàm xpath đánh giá biểu thức XPath xpath đối với giá trị XML xml. Nó trả về một mảng các giá trị XML tương ứng với tập hợp các nút được biểu thức XPath tạo ra.

Đối số thứ 2 phải là một tài liệu XML được hình thành tốt. Đặc biệt, nó phải có một yếu tố nút gốc root duy nhất.

Đối số thứ 3 của hàm là một mảng các ánh xạ không gian tên. Mảng này nên là một mảng 2 chiều với độ dài trục thứ 2 bằng 2 (nghĩa là, nó nên là một mảng của các mảng, mỗi mảng của nó bao gồm chính xác 2 phần tử). Phần tử đầu của từng đầu vào mảng là tên không gian tên (alias), phần tử thứ 2 là URL không gian tên. Không được yêu cầu rằng các alias được cung cấp trong mảng này là y hệt như các alias đang được sử dụng trong bản thân tài liệu XML (nói cách khác, cả trong tài liệu XML và trong ngữ cảnh của hàm xpath, các alias là cục bộ). Ví dụ:

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
        ARRAY[ARRAY['my', 'http://example.com']]);

xpath
--------
{test}
(1 row)
```

Làm việc thế nào với các không gian tên mặc định (nặc danh):

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
```

```
        ARRAY[ARRAY['mydefns', 'http://example.com']]);

xpath
--------
{test}
(1 row)
```

9.14.3. Bảng ánh xạ tới XML

Các hàm sau đây ánh xạ các nội dung các bảng quan hệ tới các giá trị XML. Chúng có thể được nghĩ như là chức năng xuất XML:

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
        tableforest boolean, targetns text)
```

Dạng trả về của từng hàm là xml.

table_to_xml ánh xạ nội dung của bảng có tên, truyền như tham số tbl. Dạng regclass chấp nhận các chuỗi xác định các bảng bằng việc sử dụng ký hiệu thông thường, bao gồm chất lượng sơ đồ tùy chọn và các dấu ngoặc kép, query_to_xml thực thi truy vấn mà văn bản của nó được truyền như tham số query và ánh xạ tập các kết quả. cursor_to_xml đem đến số các hàng được đánh chỉ số từ con trỏ được tham số cursor chỉ định. Phương án này được khuyến cáo nếu các bảng lớn phải được ánh xạ, vì giá trị kết quả được xây dựng trong bộ nhớ từ từng hàm.

Nếu tableforest là sai (false), thì tài liệu XML kết quả trông như thế này:

```
<tablename>
        <row>
                <columnname1>data</columnname1>
                <columnname2>data</columnname2>
        </row>

        <row>
        ...
        </row>
        ...
</tablename>
```

Nếu là đúng, thì kết quả là một phân mảnh nội dung XML mà trông như thế này:

```
<tablename>
        <columnname1>data</columnname1>
        <columnname2>data</columnname2>
</tablename>

<tablename>
...
</tablename>
...
```

Nếu không tên bảng nào là sẵn sàng, đó là, khi việc ánh xạ một truy vấn hoặc một con trỏ, chuỗi table được sử dụng trong định dạng trước, còn row trong định dạng thứ 2.

Sự lựa chọn giữa các định dạng đó phụ thuộc vào người sử dụng. Định dạng đầu tiên là một tài liệu

XML phù hợp, nó sẽ là quan trọng trong nhiều ứng dụng. Định dạng thứ 2 có xu hướng sẽ là hữu dụng hơn trong hàm cursor_to_xml nếu các giá trị kết quả sẽ được tập hợp lại trong tài liệu sau đó.

Các hàm cho việc tạo ra nội dung XML được thảo luận ở trên, đặc biệt xmlelement, có thể được sử dụng để tùy biến các kết quả theo ý muốn.

Các giá trị dữ liệu được ánh xạ theo cách thức y hệt như được mô tả cho hàm xmlelement ở trên. tham số nulls xác định liệu các giá trị null có nên được đưa vào ở đầu ra hay không. Nếu đúng, thì các giá trị null trong các cột được thể hiện như:

```
<columnname xsi:nil="true"/>
```

trong đó xsi là tiền tố không gian tên XML cho trường hợp Sơ đồ XML. Một khai báo không gian tên phù hợp sẽ được thêm vào giá trị kết quả. Nếu sai, các cột có chứa các giá trị null đơn giản sẽ bị mờ khỏi đầu ra.

Tham số targetns chỉ định không gian tên XML mong muốn của kết quả. Nếu không có không gian tên nào đặc biệt mong muốn, thì một chuỗi rỗng sẽ được truyền.

Các hàm sau trả về các tài liệu Sơ đồ XML, mô tả việc ánh xạ được các hàm tương ứng ở trên thực hiện:

```
table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)
```

Là cơ bản rằng các tham số y hệt được truyền để giành được việc khớp các ánh xạ dữ liệu XML và các tài liệu Sơ đồ XML.

Các hàm sau tạo ra các ánh xạ dữ liệu XML và Sơ đồ XML tương ứng trong một tài liệu (hoặc rừng), được liên kết với nhau. Chúng có thể là hữu dụng ở những nơi các kết quả khép kín và tự mô tả được mong muốn:

```
table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
```

Hơn nữa, các hàm sau là sẵn sàng để tạo ra các ánh xạ tương tự của toàn bộ các sơ đồ hoặc toàn bộ cơ sở dữ liệu hiện hành:

```
schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)
```

Lưu ý rằng các ánh xạ nội dung của các sơ đồ và cơ sở dữ liệu lớn, có thể đáng để xem xét việc ánh xạ các bảng một cách tách bạch thay vào đó, thậm chí có thể thông qua một con trỏ.

Kết quả của một ánh xạ nội dung sơ đồ trông giống như thế này:

```
<schemaname>
table1-mapping
table2-mapping
```

```
...
</schemaname>
```

trong đó định dạng một ánh xạ bảng phụ thuộc vào tham số rừng bảng như được giải thích ở trên. Kết quả của một ánh xạ nội dung cơ sở dữ liệu trông giống thế này:

```
<dbname>
<schema1name>
...
</schema1name>
<schema2name>
...
</schema2name>
...
</dbname>
```

trong đó ánh xạ sơ đồ là như ở trên.

Như một ví dụ sử dụng đầu ra được các hàm đó tạo ra, Hình 9-1 chỉ một bảng kiểu XSLT mà biến đổi đầu ra của table_to_xml_and_xmlschema thành một tài liệu HTML có chứa một sự trả về dạng bảng các dữ liệu bảng đó. Theo cách thức tương tự, các kết quả từ các hàm đó có thể được biến đổi thành các định dạng khác dựa vào XML.

Hình 9-1. Bảng XSLT cho việc biến đổi đầu ra SQL/XML thành HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns="http://www.w3.org/1999/xhtml"
>
        <xsl:output method="xml"
                doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
                doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
                indent="yes"/>
        <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
        select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
        select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/xsd:<html>
    <head>
        <title><xsl:value-of select="name(current())"/></title>
    </head>
    <body>
        <table>
        <tr>
            <xsl:for-each select= "$schema/xsd:complexType[@name=$rowtypename]/xsd:sequence/
                <th> <xsl:value-of select="."/></th>
            </xsl:for-each>
        </tr>

        <xsl:for-each select="row">
            <tr>
                <xsl:for-each select="*">
                    <td><xsl:value-of select="."/></td>
                </xsl:for-each>
            </tr>
            </xsl:for-each>
        </table>
    </body>
```

```
        </html>
    </xsl:template>
</xsl:stylesheet>
```

9.15. Hàm điều khiển sự tuần tự

Phần này mô tả các hàm PostgreSQL để vận hành trong các đối tượng tuần tự. Các đối tượng tuần tự (còn được gọi là các bộ sinh tuần tự hoặc chỉ đơn thuần là các tuần tự) là các bảng đặc biệt 1 hàng duy nhất được tạo ra bằng lệnh CREATE SEQUENCE. Một đối tượng tuần tự thường được sử dụng để sinh ra các mã định danh duy nhất cho các hàng của một bảng. Các hàm tuần tự, được liệt kê trong Bảng 9-39, đưa ra các phương pháp đơn giản, an toàn cho người sử dụng để có được các giá trị tuần tự thành công từ các đối tượng tuần tự.

Bảng 3-39. Các hàm tuần tự

Hàm	Dạng trả về	Mô tả
currval(regclass)	bigint	Trả về giá trị có gần đây nhất với nextval cho tuần tự được chỉ định
lastval()	bigint	Trả về giá trị có gần đây nhất với nextval cho bất kỳ tuần tự nào
nextval(regclass)	bigint	Tuần tự tiên tiến và trả về giá trị mới
setval(regclass, bigint)	bigint	Thiết lập giá trị hiện hành của tuần tự
setval(regclass, bigint, boolean)	bigint	Thiết lập giá trị hiện hành của tuần tự và cờ is_called

Tuần tự sẽ được vận hành bằng một hàm tuần tự được một đối số regclass chỉ định, nó đơn giản là một OID của sự tuần tự trong catalog hệ thống pg_class. Tuy nhiên, bạn không phải tra cứu bằng tay OID vì trình biến đổi đầu vào dạng dữ liệu regclass sẽ thực hiện công việc đó cho bạn. Chỉ viết tên sự tuần tự được đưa vào trong các dấu ngoặc đơn sao cho nó trông giống như một hằng ký tự. Đối với tính tương thích với việc điều khiển các tên SQL thông thường, chuỗi đó sẽ được biến đổi thành các chữ thường, trừ phi nó có các dấu ngoặc kép xung quanh tên tuần tự. Vì thế:

```
nextval('foo') operates on sequence foo
nextval('FOO') operates on sequence foo
nextval('"Foo"') operates on sequence Foo
```

Tên của sự tuần tự có thể được định tính theo sơ đồ nếu cần thiết:

```
nextval('myschema.foo') operates on myschema.foo
nextval('"myschema".foo') same as above
nextval('foo') searches search path for foo
```

Xem Phần 8.16 để có thêm thông tin về regclass.

> **Lưu ý**: Trước phiên bản PostgreSQL 8.1, các đối số của các hàm tuần tự từng là dạng text, không phải regclass, và biến đổi được mô tả ở trên từ một chuỗi văn bản thành một giá trị OID có thể xảy ra trong thời gian thực trong từng lời gọi. Về tính tương hợp ngược, cơ sở này vẫn tồn tại, nhưng trong nội bộ nó bây giờ được điều khiển như một sự cưỡng ép ngầm từ text tới regclass trước khi hàm đó được gọi.
>
> Khi bạn viết đối số của một hàm tuần tự như một chuỗi ký tự không có trang trí, thì nó trở thành một hằng dạng regclass. Vì điều này thực sự chỉ là một OID, nó sẽ dõi theo sự tuần tự

được nhận diện ban đầu bất chấp việc đổi tên, tái chỉ định sơ đồ sau này... Hành vi "ràng buộc sớm" này thường là mong muốn cho các tham chiếu tuần tự trong các mặc định cột và các kiểu nhìn. Nhưng đôi khi bạn có thể muốn "ràng buộc sau" ở những nơi tham chiếu tuần tự được giải quyết trong thời gian thực. Để có được hành vi ràng buộc sau, hãy ép hằng số đó phải được lưu giữ như một hằng số text thay vì regclass:

nextval('foo'::text) foo is looked up at runtime

Lưu ý rằng ràng buộc sau từng chỉ là hành vi được các phiên bản PostgreSQL trước 8.1 hỗ trợ, nên bạn có thể cần phải làm điều này để giữ lại các ngữ nghĩa của các ứng dụng cũ.

Tất nhiên, đối số của một hàm tuần tự có thể là một biểu thức cũng như một hằng số. Nếu nó là một biểu thức văn bản thì sự cưỡng ép ngầm sẽ gây ra một sự tra cứu thời gian thực.

Các hàm tuần tự có sẵn là:

nextval

Thúc đẩy đối tượng tuần tự tới giá trị tiếp sau của nó và trả về giá trị đó. Điều này được thực hiện một cách tự động: thậm chí nếu nhiều phiên thực thi nextval cùng một lúc, từng phiên sẽ an toàn nhận được một giá trị tuần tự riêng biệt.

currval

Trả về giá trị gần đây nhất có được bằng nextval cho tuần tự này trong phiên hiện hành. (Một lỗi được báo cáo nếu nextval chưa bao giờ từng được gọi cho tuần tự trong phiên này). Vì điều này đang trả về một giá trị phiên cục bộ, nó trao một câu trả lời có thể đoán trước được liệu có hay không các phiên khác đã thực thi nextval kể từ khi phiên hiện hành đã thực hiện.

lastval

Trả về giá trị gần đây nhất được trả về bằng nextval trong phiên hiện hành. Hàm này là y hệt với currval, ngoại trừ rằng thay vì lấy tên tuần tự như một đối số thì nó lấy giá trị của tuần tự cuối cùng được nextval sử dụng trong phiên hiện hành. Đây là một lỗi gọi lastval nếu nextval còn chưa được gọi trong phiên hiện hành.

setval

Thiết lập lại giá trị tính đối tượng tuần tự. Dạng 2 tham số thiết lập trường last_value của sự tuần tự cho giá trị được chỉ định và thiết lập trường is_called của nó thành đúng (true), nghĩa là nextval tiếp theo sẽ thúc đẩy sự tuần tự trước việc trả về một giá trị. Giá trị được currval nêu cũng được thiết lập tới giá trị được chỉ định. Ở dạng 3 tham số, is_called có thể được thiết lập hoặc về đúng (true) hoặc về sai (false). true có cùng hiệu ứng như dạng 2 tham số. Nếu nó được thiết lập thành sai (false), thì nextval tiếp sau sẽ trả về chính xác giá trị được chỉ định, và sự thúc đẩy tuần tự bắt đầu bằng nextval sau. Hơn nữa, giá trị được currval nêu không bị thay đổi trong trường hợp này (đây là một thay đổi từ hành vi phiên bản trước 8.3). Ví dụ,

```
SELECT setval('foo', 42); Next nextval will return 43
SELECT setval('foo', 42, true); Same as above
SELECT setval('foo', 42, false); Next nextval will return 42
```

Kết quả được setval trả về chỉ là giá trị của đối số thứ 2 của nó.

Nếu một đối tượng tuần tự từng được tạo ra với các tham số mặc định, thì các lời gọi nextval tiếp sau sẽ trả về các giá trị tiếp sau bắt đầu bằng 1. Các hành vi khác có thể có được bằng việc sử dụng các tham số đặc biệt trong lệnh CREATE SEQUENCE; xem trang tham chiếu lệnh của nó để có thêm thông tin.

Quan trọng: Để tránh việc khóa các giao dịch đồng thời mà có được các số từ cùng y hệt sự tuần tự, một hoạt động nextval là không bao giờ quay ngược lại được; đó là, một khi một giá trị từng được lấy thì nó được xem là được sử dụng, thậm chí nếu giao dịch mà đã làm cho nextval sau đó hỏng. Điều này có nghĩa là các giao dịch bị hỏng có thể để lại "các lỗ hổng" không được sử dụng trong tuần tự của các giá trị được chỉ định. Các hành động của setval cũng không bao giờ quay ngược trở lại được.

9.16. Biểu thức điều kiện

Phần này mô tả các biểu thức điều kiện tuân thủ SQL có sẵn trong PostgreSQL.

Mẹo: Nếu nhu cầu của bạn vượt ra khỏi các tương thích của các biểu thức điều kiện đó, thì bạn có thể muốn xem xét việc viết một thủ tục được lưu lại trong một ngôn ngữ lập trình có ý nghĩa hơn.

9.16.1. CASE

Biểu thức SQL CASE là một biểu thức điều kiện chung, tương tự như các lệnh if/else trong các ngôn ngữ lập trình khác:

```
CASE WHEN condition THEN result
        [WHEN ...]
        [ELSE result]
END
```

Các mệnh đề CASE có thể được sử dụng bất kỳ ở đâu một biểu thức là hợp lệ. Từng điều kiện condition là một biểu thức trả về một kết quả boolean. Nếu kết quả điều kiện là đúng, thì giá trị của biểu thức CASE là kết quả mà tuân theo điều kiện đó, và phần còn lại của biểu thức CASE không được xử lý nữa. Nếu kết quả điều kiện là không đúng, thì bất kỳ mệnh đề WHEN tiếp sau nào cũng sẽ được xem xét theo cách thức y hệt. Nếu không có điều kiện WHEN nào là đúng, thì giá trị của biểu thức CASE là kết quả của mệnh đề ELSE. Nếu mệnh đề ELSE bị mờ và không điều kiện nào là đúng, thì kết quả là null. Một ví dụ:

```
SELECT * FROM test;
a
---
1
2
3
SELECT a,
        CASE WHEN a=1 THEN 'one'
```

```
            WHEN a=2 THEN 'two'
            ELSE 'other'
        END
    FROM test;

a | case
---+-------
1 | one
2 | two
3 | other
```

Các dạng dữ liệu của tất cả các biểu thức kết quả phải có khả năng biến đổi thành một dạng đầu ra duy nhất. Xem Phần 10.5 để có thêm chi tiết.

Có một dạng "đơn giản" biểu thức CASE mà là một phương án của dạng chung ở trên:

```
CASE expression
        WHEN value THEN result
        [WHEN ...]
        [ELSE result]
END
```

Biểu thức đầu tiên được tính toán, rồi được so sánh với từng trong số các biểu thức giá trị value trong các mệnh đề WHEN cho tới khi một giá trị được tìm thấy mà bằng nó. Nếu không sự trùng khớp nào được tìm thấy, thì kết quả của mệnh đề ELSE (hoặc một giá trị null) được trả về. Điều này là tương tự với lệnh switch trong C.

Ví dụ ở trên có thể được viết bằng việc sử dụng cú pháp CASE đơn giản:

```
SELECT a,
        CASE a WHEN 1 THEN 'one'
            WHEN 2 THEN 'two'
            ELSE 'other'
        END
    FROM test;

a | case
---+-------
1 | one
2 | two
3 | other
```

Một biểu thức CASE không ước lượng được bất kỳ biểu thức con nào mà không cần thiết để xác định kết quả.

Ví dụ, đây là cách thức có khả năng để tránh một lỗi chia cho 0:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

9.16.2. COALESCE

```
COALESCE(value [, ...])
```

Hàm COALESCE trả về trước hết các đối số của nó mà không là null. Null chỉ được trả về nếu tất cả các đối số là null. Nó thường được sử dụng để thay thế một giá trị mặc định đối với các giá trị null khi dữ liệu được truy xuất để hiển thị, ví dụ:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Giống như biểu thức CASE, COALESCE chỉ đánh giá các đối số cần thiết để xác định kết quả; đó là,

các đối số ở bên phải của đối số không null đầu tiên sẽ không được đánh giá. Hàm tiêu chuẩn SQL này đưa ra các khả năng tương tự như NVL và IFNULL, chúng được sử dụng trong một số hệ thống cơ sở dữ liệu khác.

9.16.3. NULLIF

NULLIF(value1, value2)

Hàm NULLIF trả về một giá trị null nếu value1 bằng value2; nếu không thì nó trả về value1. Điều này có thể được sử dụng để thực hiện hành động ngược lại của ví dụ COALESCE được đưa ra ở trên:

SELECT NULLIF(value, '(none)') ...

Trong ví dụ này, nếu giá trị value là (none), thì null sẽ được trả về, nếu không thì giá trị của value sẽ được trả về.

9.16.4. GREATEST và LEAST

GREATEST(value [, ...])
LEAST(value [, ...])

Các hàm GREATEST và LEAST lựa chọn giá trị lớn nhất hoặc nhỏ nhất từ một danh sách bất kỳ số lượng nào các biểu thức. Các biểu thức tất cả phải tương thích với một dạng dữ liệu chung, nó sẽ là dạng của kết quả (xem Phần 10.5 để có các chi tiết). Các giá trị null trong danh sách sẽ bị bỏ qua. Kết quả sẽ là NULL chỉ nếu tất cả các biểu thức đánh giá về NULL.

Lưu ý rằng GREATEST và LEAST không phải là tiêu chuẩn SQL, nhưng là một mở rộng chung. Một số cơ sở dữ liệu khác làm cho chúng trả về NULL nếu bất kỳ đối số nào là NULL, thay vì chỉ khi tất cả là NULL.

9.17. Hàm và toán tử mảng

Bảng 9-40 chỉ ra các toán tử sẵn sàng cho các dạng mảng.

Bảng 9-40. Các toán tử mảng

Toán tử	Mô tả	Ví dụ	Kết quả
=	bằng	ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]	t
<>	không bằng	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	nhỏ hơn	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	lớn hơn	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	nhỏ hơn hoặc bằng	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	lớn hơn hoặc bằng	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
@>	bao gồm	ARRAY[1,4,3] @> ARRAY[3,1]	t
<@	nằm trong	ARRAY[2,7] <@ ARRAY[1,7,4,2,6]	t
&&	chồng lấn (có các phần tử chung)	ARRAY[1,4,3] && ARRAY[2,1]	t
||	ghép nối mảng với mảng	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}

Toán tử	Mô tả	Ví dụ	Kết quả
||	ghép nối mảng với mảng	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
||	ghép nối phần tử với mảng	3 || ARRAY[4,5,6]	{3,4,5,6}
||	ghép nối mảng với phần tử	ARRAY[4,5,6] || 7	{4,5,6,7}

Các so sánh mảng so sánh các nội dung mảng phần tử với phần tử, bằng việc sử dụng hàm so sánh cây B (B-tree) mặc định cho dạng dữ liệu phần tử. Trong các mảng đa chiều, các phần tử được viếng thăm theo trật tự hàng chính (subscript cuối cùng khác nhanh nhất). Nếu các nội dung của 2 mảng bằng nhau nhưng khác nhau về chiều, thì sự khác biệt đầu tiên trong thông tin theo chiều xác định trật tự sắp xếp. (Đây là sự thay đổi từ các phiên bản của PostgreSQL trước 8.2: các phiên bản cũ hơn có thể nói rằng 2 mảng với cùng nội dung là bằng nhau, thậm chí nếu số lượng các chiều hoặc các dải subscript là khác nhau).

Xem Phần 8.14 để có các chi tiết về hành vi của toán tử mảng.

Bảng 9-41 chỉ ra các hàm sẵn sàng để sử dụng với các dạng mảng. Xem Phần 8.14 để có thêm thông tin và các ví dụ sử dụng các hàm đó.

Bảng 9-41. Các hàm mảng

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
array_append(anyarray, anyelement)	anyarray	nối một phần tử vào cuối một mảng	array_append(ARRAY[1,2], 3)	{1,2,3}
array_cat(anyarray, anyarray)	anyarray	ghép 2 mảng	array_cat(ARRAY[1,2,3], ARRAY[4,5])	{1,2,3,4,5}
array_ndims(anyarray)	int	trả về số chiều của mảng	array_ndims(ARRAY[[1,2,3], [4,5,6]])	2
array_dims(anyarray)	text	trả về một đại diện văn bản các chiều của mảng	array_dims(ARRAY[[1,2,3], [4,5,6]])	[1:2][1:3]
array_fill(anyelement, int[], [, int[]])	anyarray	trả về mảng được khởi tạo với giá trị và các chiều được cung cấp, hợp lý với các ràng buộc thấp hơn khác 1	array_fill(7, ARRAY[3], ARRAY[2])	[2:4]={7,7,7}
array_length(anyarray, int)	int	trả về chiều dài chiều mảng được yêu cầu	array_length(array[1,2,3], 1)	3
array_lower(anyarray, int)	int	trả về ràng buộc thấp hơn chiều mảng được yêu cầu	array_lower('[0:2]={1,2,3}'::int[], 1)	0
array_prepend(anyelement, anyarray)	anyarray	nối phần tử vào đầu của mảng	array_prepend(1, ARRAY[2,3])	{1,2,3}
array_to_string(anyarray, text)	text	ghép các phần tử mảng bằng sử dụng phân cách được cung cấp	array_to_string(ARRAY[1, 2, 3], '~^~')	1~^~2~^~3
array_upper(anyarray, int)	int	trả về ràng buộc trên hơn các chiều mảng được yêu cầu	array_upper(ARRAY[1,2,3,4], 1)	4

Hàm	Dạng trả về	Mô tả	Ví dụ	Kết quả
string_to_array(text, text)	text[]	chia chuỗi thành các phần tử mảng bằng việc sử dụng phân cách được cung cấp	string_to_array('xx~^~yy~^~zz', '~^~')	{xx,yy,zz}
unnest(anyarray)	setof	bất kỳ phần tử nào mở rộng một mảng tới một tập hợp các hàng	unnest(ARRAY[1,21]2)	(2 rows)

Xem Phần 9.18 về hàm tổng hợp array_agg để sử dụng với các mảng.

9.18. Hàm tổng hợp

Các *hàm tổng hợp (aggregate function)* tính toán một kết quả duy nhất từ một tập hợp các giá trị đầu vào. Các hàm tổng hợp được xây dựng sẵn được liệt kê trong Bảng 9-42 và Bảng 9-43. Các cân nhắc cú pháp đặc biệt cho các hàm tổng hợp được giải thích trong Phần 4.2.7. Hãy xem thêm Phần 2.7 để có thông tin giới thiệu bổ sung.

Bảng 9-42. Các hàm tổng hợp mục đích chung

Hàm	(Các) dạng đối số	Dạng trả về	Mô tả
array_agg(expression)	bất kỳ	mảng dạng của đối số	các giá trị đầu vào, bao gồm cả null, được ghép trong một mảng
avg(expression)	smallint, int, bigint, real, double precision, numeric, hoặc interval	numeric tho bất kỳ đối số dạng integer nào, double precision cho một đối số dấu thập phần, nếu không thì y hệt như dạng dữ liệu của đối số	trung bình (trung bình số học) của tất cả các giá trị đầu vào
bit_and(expression)	smallint, int, bigint, hoặc bit	hệt như dạng dữ liệu của đối số	AND theo bit đối với tất cả các giá trị đầu vào không null, hoặc null nếu có một giá trị như vậy
bit_or(expression)	smallint, int, bigint, hoặc bit	hệt như dạng dữ liệu của đối số	OR theo bit đối với tất cả các giá trị đầu vào không null, hoặc null nếu có một giá trị như vậy
bool_and(expression)	bool	bool	đúng (true) nếu các giá trị đầu vào true, nếu không là sai (false)
bool_or(expression)	bool	bool	true nếu ít nhất một giá trị đầu vào là true, nếu không là false
count(*)		bigint	số lượng các hàng đầu vào
count(expression)	any	bigint	số lượng các hàng đầu vào theo đó giá trị biểu thức không null
every(expression)	bool	bool	tương đương với bool_and
max(expression)	any array, numeric, string, or date/time type	y hệt như dạng của đối số	giá trị cực đại của biểu thức khắp tất cả các giá trị đầu vào
min(expression)	any array, numeric, string, or date/time type	y hệt như dạng của đối số	giá trị cực tiểu của biểu thức khắp tất cả các giá trị đầu vào
string_agg(expression, delimiter)	text, text	text	các giá trị đầu vào được ghép trong một chuỗi, tách bạch nhau

Hàm	(Các) dạng đối số	Dạng trả về	Mô tả
			bằng các dấu ngắt
sum(expression)	smallint, int, bigint, real, double precision, numeric, hoặc interval	bigint cho các đối số smallint hoặc int, numeric cho các đối số bigint, double precision cho các đối số các dấu phân cách, nếu không thì y hệt như dạng dữ liệu của đối số	tổng biểu thức khắp tất cả các giá trị đầu vào
xmlagg(expression)	xml	xml	ghép nối các giá trị XML (xem thêm Phần 9.14.1.7)

Được lưu ý rằng ngoại trừ đối với count, các hàm đó trả về một giá trị null khi không hàng nào được chọn. Đặc biệt, sum của không hàng nào sẽ trả về null, không phải là 0 như ai đó có thể kỳ vọng, và array_agg trả về null thay vì một mảng rỗng khi không có các hàng đầu vào. Hàm coalesce có thể được sử dụng để thay thế 0 hoặc một mảng rỗng cho null khi cần thiết.

> **Lưu ý**: Boolean tổng hợp bool_and và bool_or tương ứng với các tổng hợp SQL every và any hoặc some. Như đối với any và some, dường như là có một sự mù mờ được xây dựng trong cú pháp tiêu chuẩn:
>
> ```
> SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
> ```
>
> Ở đây ANY có thể được xem hoặc như việc giới thiệu một truy vấn con, hoặc như đang là một hàm tổng hợp, nếu truy vấn con trả về một hàng với một giá trị Boolean. Vì thế tên tiêu chuẩn không thể được đưa ra cho các tổng hợp đó.
>
> **Lưu ý**: Người sử dụng đã quen làm việc với các hệ quản trị cơ sở dữ liệu SQL khác có thể thất vọng vì hiệu năng của tổng hợp count khi mà nó được áp dụng cho toàn bộ bảng. Một truy vấn như:
>
> ```
> SELECT count(*) FROM sometable;
> ```
>
> sẽ được PostgreSQL thực thi bằng việc sử dụng một sự quét tuần tự toàn bộ bảng.

Các hàm tổng hợp array_agg, string_agg và xmlagg, cũng như các hàm tổng hợp tương tự do người sử dụng định nghĩa, tạo ra các giá trị kết quả có ý nghĩa khác nhau, phụ thuộc vào trật tự của các giá trị đầu vào. Trật tự này được chỉ định mặc định, nhưng có thể được kiểm soát bằng việc viết một mệnh đề ORDER BY bên trong lời gọi tổng hợp đó, như được chỉ ra trong Phần 4.2.7. Như một sự lựa chọn, việc cung cấp các giá trị đầu vào từ một truy vấn con có sắp xếp thường sẽ làm việc. Ví dụ:

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

Nhưng cú pháp này là không được phép trong tiêu chuẩn SQL, và không khả chuyển được cho các hệ thống cơ sở dữ liệu khác.

Bảng 9-43 chỉ ra các hàm tổng hợp thường được sử dụng trong phân tích các số thống kê (chúng được tách bạch nhau chỉ để tránh việc hỗn độn danh sách các tổng hợp được sử dụng phổ biến hơn).

Ở những nơi mà mô tả nhắc tới N, nó có nghĩa là số lượng các hàng đầu vào theo đó tất cả các biểu

thức đầu vào là không null. Trong tất cả mọi trường hợp, null được trả về nếu cấu hình là không có ý nghĩa, ví dụ khi N là 0.

Bảng 9-43. Các hàm tổng hợp cho thống kê

Hàm	Dạng đối số	Dạng trả về	Mô tả
corr(Y, X)	double precision	double precision	hệ số tương quan
covar_pop(Y, X)	double precision	double precision	hiệp phương sai (population covariance)
covar_samp(Y, X)	double precision	double precision	phương sai mẫu
regr_avgx(Y, X)	double precision	double precision	trung bình phương sai độc lập (sum(X)/N)
regr_avgy(Y, X)	double precision	double precision	trung bình biến độc lập (sum(Y)/N)
regr_count(Y, X)	double precision	bigint	số hàng đầu vào theo đó cả 2 biểu thức là không null
regr_intercept(Y, X)	double precision	double precision	giao cắt trục y phương trình tuyến tính các bình phương nhỏ nhất vừa khớp được các cặp (X, Y) xác định
regr_r2(Y, X)	double precision	ouble precision	dsquare của hệ số tương quan
regr_slope(Y, X)	double precision	double precision	độ dốc phương trình tuyến tính các bình phương nhỏ nhất vừa khớp được các cặp (X, Y) xác định
regr_sxx(Y, X)	double precision	double precision	sum(X^2) - sum(X)^2/N (“tổng các bình phương” của biến độc lập)
regr_sxy(Y, X)	double precision	double precision	sum(X*Y) – sum(X) * sum(Y)/N (“tổng các sản phẩm” của biến phụ thuộc theo thời gian độc lập)
regr_syy(Y, X)	double precision	double precision	sum(Y^2) - sum(Y)^2/N (“tổng bình phương” của biến phụ thuộc)
stddev(expression)	smallint, int, bigint, real, double precision, hoặc numeric	double precision cho các đối số dấu chấm động, nếu không thì là numeric	alias theo lịch sử cho stddev_samp
stddev_pop(expression)	smallint, int, bigint, real, double precision, hoặc numeric	double precision cho các đối số dấu chấm động, nếu không thì là numeric	hiệp phương sai tiêu chuẩn của các giá trị đầu vào
stddev_samp(expression)	smallint, int, bigint, real, double precision, hoặc numeric	double precision cho các đối số dấu chấm động, nếu không thì là numeric	phương sai mẫu tiêu chuẩn của các giá trị đầu vào
variance(expression)	smallint, int, bigint, real, double precision, hoặc numeric	double precision cho các đối số dấu chấm động, nếu không thì là numeric	alias theo lịch sử cho var_samp
var_pop(expression)	smallint, int, bigint, real, double precision, hoặc numeric	double precision cho các đối số dấu chấm động, nếu không thì là numeric	hiệp phương sai của các giá trị đầu vào (bình phương của hiệp phương sai tiêu chuẩn)
var_samp(expression)	smallint, int, bigint, real, double precision, hoặc numeric	double precision cho các đối số dấu chấm động, nếu không thì là numeric	phương sai mẫu của các giá trị đầu vào (bình phương phương sai mẫu tiêu chuẩn)

9.19. Hàm cửa sổ

Các hàm cửa sổ đưa ra khả năng thực hiện các tính toán khắp các tập hợp các hàng có liên quan tới hàng truy vấn hiện hành. Xem Phần 3.5 để có sự giới thiệu cho chức năng này.

Các hàm cửa sổ được xây dựng sẵn được liệt kê trong Bảng 9-44. Lưu ý rằng các hàm đó phải được triệu gọi bằng việc sử dụng cú pháp các hàm cửa số; đó là một mệnh đề OVER theo yêu cầu.

Bổ sung thêm vào các hàm đó, bất kỳ hàm tổng hợp được xây dựng sẵn hoặc do người sử dụng tự định nghĩa nào cũng có thể được sử dụng như một hàm cửa sổ (xem Phần 9.18 để có một danh sách các tổng hợp được xây dựng sẵn đó). Các hàm tổng hợp hành động như các hàm cửa sổ chỉ khi một mệnh đề OVER đi sau lời gọi; nếu không thì chúng hành động như là các tổng hợp thông thường.

Bảng 9-44. Các hàm cửa sổ mục đích chung

Hàm	Dạng trả về	Mô tả
row_number()	bigint	số hàng hiện hành trong phân vùng của nó, tính từ 1
rank()	bigint	hạng của hàng hiện hành với các khoảng cách; y hệt như row_number điểm ngang hàng đầu tiên của nó
dense_rank()	bigint	hạng của hàng hiện hành không có khoảng cách; hàm này tính các nhóm ngang hàng
percent_rank()	double precision	hạng tương đối của hàng hiện hành: (hạng - 1) / (tổng số hàng - 1)
cume_dist()	double precision	hạng tương đối của hàng hiện hành: (số hàng đi trước hoặc ngang hàng với hàng hiện hành) / (tổng số hàng)
ntile(num_buckets integer)	integer	số nguyên xếp từ 1 tới giá trị đối số, chia cho phần bằng nhau có thể
lag(value any [, offset integer [, default any]])	dạng y hệt như giá trị	trả về giá trị được đánh giá trong hàng mà là các hàng bù trước hàng hiện hành trong phân vùng đó; nếu không có hàng nào như vậy, thì thay vào đó trả về mặc định. Cả phần bù và mặc định được đánh giá với lưu ý cho hàng hiện hành. Nếu bị mờ, phần bù mặc định là 1 và mặc định là 0
lead(value any [, offset integer [, default any]])	dạng y hệt như giá trị	trả về giá trị được đánh giá trong hàng mà là các hàng bù sau hàng hiện hành trong phân vùng đó; nếu không có hàng nào như vậy, thì thay vào đó trả về mặc định. Cả phần bù và mặc định được đánh giá với lưu ý cho hàng hiện hành. Nếu bị mờ, phần bù mặc định là 1 và mặc định là 0
first_value(value any)	dạng y hệt như giá trị	trả về giá trị được đánh giá trong hàng mà là hàng đầu tiên của khung cửa sổ
last_value(value any)	dạng y hệt như giá trị	trả về giá trị được đánh giá trong hàng mà là hàng cuối của khung cửa sổ
nth_value(value any, nth integer)	dạng y hệt như giá trị	trả về giá trị được đánh giá ở hàng mà là hàng thứ n của khung cửa sổ (tính từ 1); null nếu không có hàng nào như vậy

Tất cả các hàm được liệt kê trong Bảng 9-44 phụ thuộc vào trật tự sắp xếp được mệnh đề ORDER BY của định nghĩa cửa sổ có liên quan, chỉ định. Các hàng mà không phân biệt trong trật tự sắp xếp ORDER BY sẽ được nói là các điểm ngang hàng; 4 hàm xếp hạng được xác định sao cho chúng đưa ra

câu trả lời y hệt cho bất kỳ 2 hàng đồng hàng nào.

Lưu ý rằng first_value, last_value và nth_value chỉ xem xét các hàng bên trong "khung cửa sổ", mà mặc định bao gồm các hàng từ đầu của phân vùng qua sự ngang hàng cuối cùng của hàng hiện hành. Điều này có khả năng đưa ra các kết quả không hữu dụng cho last_value và đôi khi cả nth_value. Bạn có thể tái xác định khung đó bằng việc bổ sung thêm một đặc tả khung phù hợp (RANGE hoặc ROWS) tới mệnh đề OVER. Xem Phần 4.2.8 để có thêm thông tin về các đặc tả khung.

Khi một hàm tổng hợp được sử dụng như một hàm cửa sổ, nó tổng hợp qua các hàng bên trong khung cửa sổ của hàng hiện hành. Một sự tổng hợp được sử dụng với ORDER BY và định nghĩa khung cửa sổ mặc định tạo ra một dạng hành vi "chạy tổng", nó có thể có hoặc có thể không là những gì được mong muốn. Để có được sự tổng hợp đối với toàn bộ phần đó, hãy làm mờ ORDER BY hoặc sử dụng ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. Các đặc tả khác của khung có thể được sử dụng để giành được các hiệu ứng khác.

> **Lưu ý**: Tiêu chuẩn SQL xác định một lựa chọn RESPECT NULLS hoặc IGNORE NULLS cho lead, lag, first_value, last_value và nth_value. Điều này không được triển khai trong PostgreSQL: hành vi đó luôn là y hệt như mặc định tiêu chuẩn, ấy là RESPECT NULLS. Cũng vậy, lựa chọn tiêu chuẩn FROM FIRST hoặc FROM LAST đối với nth_value không được triển khai: chỉ hành vi mặc định FROM FIRST được hỗ trợ. (Bạn có thể đạt được kết quả đó của FROM LAST bằng việc lật ngược lại trật tự sắp xếp của ORDER BY).

9.20. Biểu thức truy vấn con

Phần này mô tả các biểu thức truy vấn con tuân thủ SQL có sẵn trong PostgreSQL. Tất cả các mẫu biểu thức được ghi chép trong phần này trả về các kết quả Boolean (đúng/sai – true/false).

9.20.1. EXISTS

EXISTS (subquery)

Đối số của EXISTS là một lệnh tùy ý SELECT, hoặc một truy vấn con. Truy vấn con được đánh giá để xác định liệu nó có trả về bất kỳ hàng nào hay không. Nếu nó trả về ít nhất 1 hàng, thì kết quả của EXISTS là đúng "true"; Nếu truy vấn phụ không trả về hàng nào, thì kết quả của EXISTS là sai "false".

Truy vấn con có thể tham chiếu tới các biến từ truy vấn bao quanh, nó sẽ hành động như các hằng số trong bất kỳ một đánh giá nào của truy vấn con đó.

Truy vấn con thường sẽ chỉ được thực thi đủ lâu để xác định liệu có ít nhất một hàng được trả về hay không, chứ không phải trên đường hoàn tất. Là không khôn ngoan để viết một truy vấn con mà có các hiệu ứng phụ (như gọi các hàm tuần tự); liệu các hiệu ứng phụ xảy ra có là không thể đoán trước được hay không.

Vì kết quả chỉ phụ thuộc vào liệu có bất kỳ hàng nào được trả về hay không, và không dựa vào các nội dung của các hàng đó, danh sách đầu ra của truy vấn con đó thường là quan trọng. Một qui ước lập trình chung là phải viết tất cả các kiểm thử EXISTS ở dạng EXISTS(SELECT 1 WHERE ...). Tuy nhiên,

có các ngoại lệ cho qui tắc này, như các truy vấn con mà sử dụng INTERSECT.

Ví dụ đơn giản này giống như một kết nối bên trong ở col2, nhưng nó tạo ra nhiều nhất một hàng đầu ra cho từng hàng của tab1, thậm chí nếu có vài hàng của tab2 khớp:

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.20.2. IN

expression IN (subquery)

Phía bên tay phải là một truy vấn con trong các dấu ngoặc đơn, nó phải trả về chính xác một cột. Biểu thức bên tay trái được đánh giá và được so sánh với từng hàng kết quả của truy vấn con đó. Kết quả của IN là đúng "true" nếu bất kỳ hàng của truy vấn con nào bằng được tìm thấy. Kết quả là sai "false" nếu không hàng nào bằng được thấy (bao gồm cả trường hợp nơi mà truy vấn con đó không trả về hàng nào cả).

Lưu ý rằng nếu biểu thức bên tay trái cho ra null, hoặc nếu không có các giá trị bằng nào bên tay phải và ít nhất một hàng bên tay phải có null, thì kết quả của cấu trúc IN sẽ là null, chứ không phải là false. Điều này là tuân theo với các qui tắc thông thường của SQL cho các kết hợp Boolean của các giá trị null.

Như với EXISTS, không khôn ngoan để giả thiết rằng truy vấn con đó sẽ được đánh giá hoàn chỉnh.

row_constructor IN (subquery)

Phía bên tay trái của mẫu này của IN là một cấu trúc hàng, như được mô tả trong Phần 4.2.12. Phần bên tay phải là một truy vấn con trong các dấu ngoặc đơn, nó phải trả về chính xác càng nhiều cột như có các biểu thức ở hàng bên tay trái. Các biểu thức bên tay trái được đánh giá và được so sánh từng hàng một đối với kết quả của truy vấn con. Kết quả của IN là "true" nếu bất kỳ hàng của truy vấn con bằng nhau nào được thấy. Kết quả là "false" nếu không hàng bằng nhau nào được thấy (bao gồm cả trường hợp nơi mà truy vấn con không trả về hàng nào).

Như thường lệ, các giá trị null trong các hàng được kết hợp theo các qui tắc thông thường của các biểu thức Boolean của SQL. Hai hàng được xem là bằng nhau nếu tất cả các thành viên tương ứng của chúng là không null và bằng nhau; các hàng không bằng nhau nếu bất kỳ các thành viên tương ứng nào là không null và không bằng nhau; nếu không thì kết quả của sự so sánh hàng đó là không rõ (null). Nếu tất cả các kết quả theo hàng hoặc không bằng nhau hoặc là null, với ít nhất một kết quả null, thì kết quả của IN là null.

9.20.3. NOT IN

expression NOT IN (subquery)

Bên tay phải là một truy vấn con trong các dấu ngoặc đơn, nó phải trả về chính xác 1 cột. Biểu thức bên tay trái được đánh giá và được so sánh với từng hàng kết quả của truy vấn con đó. Kết quả của NOT IN là "true" nếu chỉ các hàng không bằng nhau của truy vấn con được thấy (bao gồm cả trường

hợp nơi mà truy vấn con đó không trả về hàng nào). Kết quả là "false" nếu bất kỳ hàng bằng nhau nào được tìm thấy.

Lưu ý rằng nếu biểu thức bên tay trái cho null, hoặc nếu không có các giá trị bằng nhau bên tay phải và ít nhất một hàng bên tay phải cho null, thì kết quả của cấu trúc NOT IN sẽ là null, chứ không phải là true. Điều này là tuân theo với các qui tắc thông thường của SQL cho các kết hợp Boolean của các giá trị null.

Như với EXISTS, là không khôn ngoan để giả thiết rằng truy vấn con sẽ được đánh giá hoàn chỉnh.

row_constructor NOT IN (subquery)

Bên tay trái của mẫu này của NOT IN là một cấu trúc hàng, như được mô tả trong Phần 4.2.12. Bên tay phải là một truy vấn con trong các dấu ngoặc đơn, nó phải trả về chính xác càng nhiều cột như có các biểu thức ở hàng bên tay trái. Các biểu thức bên tay trái được đánh giá và được so sánh với từng hàng của kết quả truy vấn con. Kết quả của NOT IN là "true" nếu chỉ các hàng truy vấn con không bằng nhau được tìm thấy (bao gồm trường hợp nơi mà truy vấn con đó không trả về hàng nào). Kết quả là "false" nếu bất kỳ hàng bằng nhau nào được tìm thấy.

Như thường lệ, các giá trị null trong các hàng được kết hợp theo các qui tắc thông thường của các biểu thức Boolean SQL. Hai hàng được xem là bằng nhau nếu tất cả các thành viên tương ứng của chúng là không null và bằng nhau; các hàng là không bằng nhau nếu bất kỳ các thành viên tương ứng nào là không null và không bằng nhau; nếu không thì kết quả của so sánh hàng đó là không rõ (null). Nếu tất cả các kết quả theo hàng hoặc không bằng nhau hoặc là null, với ít nhất một kết quả null, thì sau đó kết quả của NOT IN là null.

9.20.4. ANY/SOME

expression operator ANY (*subquery*)
expression operator SOME (*subquery*)

Bên tay phải là một truy vấn con trong các dấu ngoặc đơn, nó phải trả về chính xác một cột. Biểu thức bên tay trái được đánh giá và được so sánh hàng khôn ngoan theo từng hàng kết quả của truy vấn con bằng việc sử dụng operator được đưa ra, nó phải cho một kết quả Boolean. Kết quả của ANY là "true" nếu bất kỳ kết quả true nào có được. Kết quả là "false" nếu không kết quả true nào được thấy (bao gồm trường hợp nơi mà truy vấn con không trả về hàng nào).

SOME là một đồng nghĩa với ANY. IN là tương đương với = ANY.

Lưu ý là nếu không có thành công và ít nhất một hàng bên tay phải có null cho kết quả của toán tử đó, thì kết quả của cấu trúc ANY sẽ là null, không phải là false. Điều này là tuân theo với các qui tắc thông thường của SQL đối với các kết hợp Boolean của các giá trị null.

Như với EXISTS, là không khôn ngoan để giả thiết rằng truy vấn con sẽ được đánh giá hoàn chỉnh.

row_constructor operator ANY (*subquery*)
row_constructor operator SOME (*subquery*)

Bên tay trái của mẫu này của ANY là một cấu trúc hàng, như được mô tả trong Phần 4.2.12. Bên tay

phải là một truy vấn con trong các dấu ngoặc đơn, nó phải trả về chính xác càng nhiều cột như có các biểu thức trong hàng bên tay trái càng tốt. Các biểu thức bên tay trái được đánh giá và được so sánh hàng khôn ngoan theo từng hàng kết quả của truy vấn con đó, bằng việc sử dụng operator được đưa ra. Kết quả của ANY là "true" nếu sự so sánh đó trả về true cho bất kỳ hàng truy vấn con nào. Kết quả là "false" nếu so sánh đó trả về false cho từng hàng của truy vấn con đó (bao gồm cả trường hợp nơi mà truy vấn con đó không trả về hàng nào). Kết quả là NULL nếu so sánh đó không trả về true cho bất kỳ hàng nào, và nó trả về NULL cho ít nhất một hàng.

Xem Phần 9.21.5 về các chi tiết về ý nghĩa của một so sánh hàng khôn ngoan.

9.20.5. ALL

expression operator ALL (*subquery*)

Bên tay phải là một truy vấn con trong các dấu ngoặc đơn, nó phải trả về chính xác 1 cột. Biểu thức bên tay trái được đánh giá và được so sánh tới từng hàng kết quả của truy vấn con bằng việc sử dụng operator được đưa ra, nó phải cho một kết quả Boolean. Kết quả của ALL là "true" nếu tất cả các hàng cho true (bao gồm cả trường hợp nơi mà truy vấn con đó không trả về hàng nào). Các kết quả là "false" nếu bất kỳ kết quả false nào được tìm thấy. Kết quả là NULL nếu so sánh đó không trả về false cho bất kỳ hàng nào, và nó trả về NULL cho ít nhất một hàng.

NOT IN là tương đương với <> ALL.

Như với EXISTS, là không khôn ngoan để giả thiết rằng truy vấn con sẽ được đánh giá hoàn chỉnh.

row_constructor operator ALL (*subquery*)

Bên tay trái của mẫu này của ALL là một cấu trúc hàng, như được mô tả trong Phần 4.2.12. Bên tay phải là một truy vấn con trong các dấu ngoặc đơn, nó phải trả về chính xác càng nhiều cột như có các biểu thức ở hàng bên tay trái càng tốt. Các biểu thức bên tay trái được đánh giá và được so sánh hàng khôn ngoan cho từng hàng kết quả của truy vấn con, bằng việc sử dụng operator được đưa ra. Kết quả của ALL là "true" nếu so sánh đó trả về true cho tất cả các hàng của truy vấn con đó (bao gồm cả trường hợp nơi mà truy vấn con đó không trả về hàng nào). Kết quả là "false" nếu so sánh đó trả về false cho bất kỳ hàng nào của truy vấn con. Kết quả là NULL nếu so sánh đó không trả về false cho bất ký hàng truy vấn con nào, và nó trả về NULL cho ít nhất một hàng.

Xem Phần 9.21.5 để có các chi tiết về ý nghĩa của một so sánh hàng khôn ngoan.

9.20.6. So sánh hàng khôn ngoan

row_constructor operator (subquery)

Bên tay trái là một cấu trúc hàng, như được mô tả trong Phần 4.2.12. Bên tay phải là một truy vấn con trong các dấu ngoặc đơn, nó phải trả về chính xác càng nhiều cột như có các biểu thức ở hàng bên tay trái càng tốt. Hơn nữa, truy vấn con không thể trả về nhiều hơn 1 hàng. (Nếu nó không trả về hàng nào, thì kết quả được lấy là null). Bên tay trái được đánh giá và được so sánh hàng khôn ngoan cho từng hàng kết quả của truy vấn con duy nhất.

Xem Phần 9.21.5 để có các chi tiết về ý nghĩa của một so sánh hàng khôn ngoan.

9.21. So sánh hàng và mảng

Phần này mô tả vài cấu trúc đặc biệt cho việc tiến hành nhiều so sánh giữa các nhóm giá trị. Các mẫu có liên quan về cú pháp đối với các mẫu truy vấn con của phần trước, nhưng không có liên quan tới các truy vấn con. Các mẫu có liên quan tới các biểu thức con của mảng là các mở rộng của PostgreSQL; phần còn lại là tuân thủ SQL. Tất cả các mẫu biểu thức được ghi thành tài liệu trong phần này trả về các kết quả Boolean (true/false).

9.21.1. IN

expression IN (value [, ...])

Bên tay phải là một danh sách các biểu thức vô hướng trong các dấu ngoặc đơn. Kết quả là "true" nếu kết quả biểu thức bên tay trái là bằng với bất kỳ biểu thức nào bên tay phải. Đây là một ký hiệu tốc ký cho

expression = value1
OR
expression = value2
OR
...

Lưu ý rằng nếu biểu thức bên tay trái có null, hoặc nếu không có các giá trị bằng nhau nào bên tay phải và ít nhất một biểu thức bên tay phải có null, thì kết quả của cấu trúc IN sẽ là null, không là "false". Điều này tuân theo qui tắc thông thường của SQL về các kết hợp Boolean các giá trị null.

9.21.2. NOT IN

expression NOT IN (*value* [, ...])

Bên tay phải là một danh sách các biểu thức vô hướng trong các dấu ngoặc đơn. Kết quả là "true" nếu kết quả biểu thức bên tay trái là bằng với tất cả các biểu thức bên tay phải. Đây là một ký hiệu tốc ký cho

expression <> value1
AND
expression <> value2
AND
...

Lưu ý rằng nếu biểu thức bên tay trái cho null, hoặc nếu không có giá trị bằng nhau nào bên tay phải và ít nhất một biểu thức bên tay phải cho null, thì kết quả của cấu trúc NOT IN sẽ là null, không đúng như bạn có thể ngây thơ kỳ vọng. Điều này là tuân theo với các qui tắc SQL thông thường cho các kết hợp Boolean các giá trị null.

> **Mẹo**: x NOT IN y là tương đương với NOT (x IN y) trong mọi trường hợp. Tuy nhiên, các giá trị null giống hơn nhiều để lên cái mới khi làm việc với NOT IN so với khi làm việc với IN. Là tốt nhất để thể hiện điều kiện của bạn một cách tích cực nếu có thể.

9.21.3. ANY/SOME (mảng)

expression operator ANY (*array expression*)
expression operator SOME (*array expression*)

Bên tay phải là một biểu thức trong dấu ngoặc đơn, nó phải cho một giá trị mảng. Biểu thức bên tay trái được đánh giá và được so sánh với từng phần tử của mảng bằng việc sử dụng operator được đưa ra, nó phải cho một kết quả Boolean. Kết quả của ANY là "true" nếu bất kỳ kết quả true nào có được. Kết quả là "false" nếu không kết quả true nào được tìm thấy (bao gồm cả trường hợp nơi mà mảng đó không có phần tử nào).

Nếu biểu thức mảng đó cho một mảng null, thì kết quả của ANY sẽ là null. Nếu biểu thức bên tay trái cho null, thì kết quả của ANY thường là null (dù một toán tử so sánh không khắt khe có thể có khả năng cho một kết quả khác). Hơn nữa, nếu mảng bên tay phải có bất kỳ các thành phần null nào và kết quả so sánh không đúng có được, thì kết quả của ANY vẫn sẽ là null, không phải là false (một lần nữa, giả thiết một toán tử so sánh khắt khe). Điều này tuân thủ với các qui tắc SQL thông thường cho các kết hợp Boolean của các giá trị null.

SOME là một đồng nghĩa với ANY.

9.21.4. ALL (mảng)

expression operator ALL (array expression)

Bên tay phải là một biểu thức trong các dấu ngoặc đơn, nó phải cho một giá trị mảng. Biểu thức bên tay trái được đánh giá và được so sánh với từng phần tử mảng bằng việc sử dụng operator được đưa ra, nó phải cho một kết quả Boolean. Kết quả của ALL là "true" nếu tất cả các so sánh cho là đúng (bao gồm cả trường hợp nơi mà mảng đó có các phần tử 0). Kết quả là "false" nếu bất kỳ kết quả sai nào được thấy.

Nếu biểu thức mảng cho một mảng null, thì kết quả của ALL sẽ là null. Nếu biểu thức bên tay trái cho null, thì kết quả của ALL thường là null (dù một toán tử so sánh không khắt khe có thể có khả năng cho một kết quả khác). Hơn nữa, nếu mảng bên tay phải có bất kỳ phần tử null nào và kết quả so sánh không đúng có được, thì kết quả của ALL sẽ là null, không phải false (một lần nữa, giả thiết một toán tử so sánh khắt khe). Điều này là tuân thủ với các qui tắc SQL thông thường cho các kết hợp Boolean của các giá trị null.

9.21.5. So sánh hàng khôn ngoan

row_constructor operator row_constructor

Từng bên là một cấu trúc hàng, như được mô tả trong Phần 4.2.12. Hai giá trị hàng đó phải có cùng số các trường. Từng bên được đánh giá và chúng được so sánh hàng khôn ngoan. Các so sánh hàng được phép khi operator là = , <> , < , <= , > or >=, hoặc có ngữ nghĩa tương tự một trong số đó. (Sẽ là đặc biệt, một toán tử có thể là một toán tử so sánh hàng nếu nó là một thành viên của một lớp toán tử B-tree, hoặc là phủ định của thành viên = trong một lớp toán tử B-tree).

Các trường hợp = và <> làm việc khá khác nhau so với các trường hợp khác. Hai hàng được xem là bằng nhau nếu tất cả các thành viên tương ứng của chúng là không null và bằng nhau; các hàng là không bằng nhau nếu bất kỳ thành viên tương ứng nào là không null và không bằng nhau; nếu không thì kết quả của so sánh hàng đó là không rõ (null).

Đối với các trường hợp <, <=, > và >=, các phần tử hàng được so sánh từ trái qua phải, dừng ngay khi một cặp các phần tử không bằng nhau hoặc null được thấy. Nếu hoặc cặp các phần tử này là null, thì kết quả của so sánh hàng là không rõ (null); nếu không thì so sánh cặp các phần tử này xác định kết quả. Ví dụ, `ROW(1,2,NULL) < ROW(1,3,0)` cho true, không phải null, vì cặp các phần tử thứ 3 sẽ không được xem xét.

> **Lưu ý**: Trước phiên bản PostgreSQL 8.2, các trường hợp <, <=, > và >= từng không được đặc tả SQL nắm. Một so sánh như `ROW(a,b) < ROW(c,d)` từng được triển khai như là `a < c AND b < d`, trong khi hành vi đúng là tương đương với `a < c OR (a = c AND b < d)`.

row_constructor IS DISTINCT FROM row_constructor

Cấu trúc này giống với một so sánh hàng <>, nhưng nó không cho null đối với các đầu vào null. Thay vào đó, bất kỳ giá trị null nào được xem là không bằng với (khác với) bất kỳ giá trị không null nào, và bất kỳ 2 null nào được xem là bằng nhau (không khác nhau). Vì thế kết quả hoặc sẽ là true hoặc là false, không bao giờ null.

row_constructor IS NOT DISTINCT FROM row_constructor

Cấu trúc này là tương tự với một so sánh hàng =, nhưng nó không cho null đối với các đầu vào null. Thay vào đó, bất kỳ giá trị null nào cũng được xem là không bằng với (khác với) bất kỳ giá trị không null nào, và bất kỳ 2 null nào cũng được xem là bằng nhau (không khác). Vì thế kết quả sẽ luôn hoặc là true hoặc là false, không bao giờ null.

> **Lưu ý**: Đặc tả SQL đòi hỏi so sánh hàng khôn ngoan để trả về NULL nếu kết quả phụ thuộc vào việc so sánh 2 giá trị NULL hoặc một NULL và một không NULL. PostgreSQL chỉ làm điều này khi so sánh các kết quả của 2 cấu trúc hàng hoặc so sánh một cấu trúc hàng với đầu ra của một truy vấn con (như trong Phần 9.20). Trong các ngữ cảnh khác nơi mà 2 giá trị dạng tổng hợp được so sánh, thì 2 giá trị trường NULL được xem là bằng nhau, và một NULL được xem là lớn hơn so với một không NULL. Điều này là cần thiết để có hành vi của việc sắp xếp và đánh chỉ số nhất quán cho các dạng tổng hợp.

9.22. Hàm thiết lập trả về

Phần này mô tả các hàm có khả năng trả về nhiều hơn một hàng. Hiện hành chỉ các hàm trong lớp này là các hàm sinh loạt, như được chi tiết hóa trong Bảng 9-45 và Bảng 9-46.

Bảng 9-45. Các hàm sinh loạt

Hàm	Dạng đối số	Dạng trả về	Mô tả
generate_series(start, stop)	int or bigint	setof int hoặc setof bigint (hệt như dạng đối số)	Sinh một loạt các giá trị, từ đầu tới khi dừng với bước kích cỡ bằng 1

Hàm	Dạng đối số	Dạng trả về	Mô tả
generate_series(start, stop, step)	int or bigint	setof int hoặc setof bigint (hệt như dạng đối số)	Sinh một loạt các giá trị, từ đầu tới khi dừng với bước kích cỡ bằng bước.
generate_series(start, stop, step interval)	timestamp or timestamp with time zone	setof timestamp hoặc setof timestamp with time zone (hệt như dạng đối số)	Sinh một loạt các giá trị, từ đầu tới khi dừng với bước kích cỡ bằng bước.

Khi bước là dương, thì các hàng 0 sẽ được trả về nếu bắt đầu là lớn hơn so với bước. Ngược lại, khi bước là âm, các hàng 0 sẽ được trả về nếu bắt đầu ít hơn so với điểm dừng. Các hàng 0 cũng sẽ được trả về cho các đầu vào NULL. Là một lỗi cho bước sẽ là 0. Một số ví dụ như sau:

```
SELECT * FROM generate_series(2,4);

generate_series
-----------------
2
3
4
(3 rows)

SELECT * FROM generate_series(5,1,-2);

generate_series
-----------------
5
3
1
(3 rows)

SELECT * FROM generate_series(4,3);

generate_series
-----------------
(0 rows)
```

-- ví dụ này dựa vào toán tử ngày tháng cộng với số nguyên

```
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);

dates
------------
2004-02-05
2004-02-12
2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
'2008-03-04 12:00', '10 hours');

generate_series
---------------------
2008-03-01 00:00:00
2008-03-01 10:00:00
2008-03-01 20:00:00
2008-03-02 06:00:00
2008-03-02 16:00:00
2008-03-03 02:00:00
2008-03-03 12:00:00
2008-03-03 22:00:00
```

```
2008-03-04 08:00:00
(9 rows)
```

Bảng 9-46. Các hàm sinh ra subscript

Hàm	Dạng trả về	Mô tả
generate_subscripts(array anyarray, dim int)	setof int	Sinh ra một dãy từ các subscript của mảng được đưa ra.
generate_subscripts(array anyarray, dim int, reverse boolean)	setof int	Sinh ra một dãy các subscript của mảng được đưa ra. Khi ngược lại là đúng, thì các dãy được trả về theo trật tự ngược lại.

generate_subscripts là một hàm thuận tiện sinh ra tập hợp các subscript hợp lệ được chỉ định cho chiều của mảng được đưa ra. Các hàng 0 được trả về cho các mảng mà không có chiều được yêu cầu, hoặc các mảng NULL (nhưng các subscript hợp lệ được trả về cho các phần tử mảng NULL). Một số ví dụ như sau:

```
-- sử dụng cơ bản
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;

s
---
1
2
3
4
(4 rows)
-- thể hiện một mảng, subscript và được subscript

-- giá trị đòi hỏi một truy vấn phụ
SELECT * FROM arrays;

a
--------------------
{-1,-2}
{100,200,300}
(2 rows)
SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;

array              | subscript | value
-------------------+-----------+----------
         {-1,-2} |         1 |    -1
         {-1,-2} |         2 |    -2
{100,200,300} |         1 | 100
{100,200,300} |         2 | 200
{100,200,300} |         3 | 300
(5 rows)

-- bỏ lồng một mảng 2D
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
    from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
```

```
CREATE FUNCTION
postgres=# SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);

unnest2
---------
1
2
3
4
(4 rows)
```

9.23. Hàm thông tin hệ thống

Bảng 9-47 chỉ ra vài hàm trích thông tin phiên làm việc và hệ thống. Bổ sung thêm vào các hàm được liệt kê trong phần này, có một số hàm có liên quan tới hệ thống thống kê cũng cung cấp thông tin hệ thống. Xem Phần 27.2.2 để có thêm thông tin.

Bảng 9-47. Các hàm thông tin phiên

Hàm	Dạng trả về	Mô tả
current_catalog	name	tên của cơ sở dữ liệu hiện hành (gọi là “catalog” theo tiêu chuẩn SQL)
current_database()	name	tên của cơ sở dữ liệu hiện hành
current_schema[()]	name	tên của cơ sở dữ liệu hiện hành
current_schemas(boolean)	name[]	tên các sơ đồ trong đường tìm kiếm, tùy ý đưa vào các sơ đồ ngầm định
current_user	name	tên của người sử dụng theo ngữ cảnh thực thi hiện hành
current_query()	text	văn bản truy vấn thực thi hiện hành, như được máy trạm đệ trình (có thể chứa nhiều hơn một lệnh)
pg_backend_pid()	int	ID qui trình của tiến trình máy chủ được gắn cho phiên hiện hành
pg_listening_channels()	setof text	tên các kênh mà phiên đang nghe hiện hành
inet_client_addr()	inet	địa chỉ của kết nối từ xa
inet_client_port()	int	cổng của kết nối từ xa
inet_server_addr()	inet	địa chỉ của kết nối cục bộ
inet_server_port()	int	cổng của kết nối cục bộ
pg_my_temp_schema()	oid	OID của sơ đồ tạm thời của phiên, hoặc 0 nếu không có
pg_is_other_temp_schema(oid)	boolean	là sơ đồ của sơ đồ tạm thời của phiên khác?
pg_postmaster_start_time()	timestamp with time zone	thời gian khởi động máy chủ
pg_conf_load_time()	timestamp with time zone	thời gian tải cấu hình
session_user	name	tên người sử dụng phiên
user	name	tương đương với current_user
version()	text	thông tin phiên bản của PostgreSQL

Lưu ý: current_catalog, current_schema, current_user, session_user và user có tình trạng cú pháp đặc

biệt trong SQL: chúng phải được gọi mà không có các dấu ngoặc đơn đi sau. (Trong PostgreSQL, các dấu ngoặc đơn có thể tùy ý được sử dụng với current_schema, nhưng không với các thứ khác).

session_user là thông thường mà người sử dụng đã khởi tạo kết nối cơ sở dữ liệu hiện hành; nhưng các siêu người sử dụng (superusers) có thể thay đổi thiết lập này với SET SESSION AUTHORIZATION. current_user là mã định danh người sử dụng mà có thể áp dụng được cho việc kiểm tra quyền. Thường thì nó là ngang bằng với người sử dụng phiên, nhưng nó có thể bị thay đổi với SET ROLE. Nó cũng thay đổi trong quá trình thực thi các hàm với thuộc tính SECURITY DEFINER. Trong cách nói của Unix, người sử dụng phiên là "người sử dụng thực sự" và người sử dụng hiện hành là "người sử dụng có hiệu quả".

current_schema trả về tên của sơ đồ mà là đầu tiên trong đường tìm kiếm (hoặc một giá trị null nếu đường tìm kiếm là rỗng). Đây là sơ đồ mà sẽ được sử dụng cho bất kỳ bảng nào hoặc các đối tượng có tên khác được tạo ra mà không có việc chỉ định một sơ đồ đích. current_schemas(boolean) trả về một mảng các tên của tất cả các sơ đồ hiện có trong đường tìm kiếm. Lựa chọn boolean xác định liệu có hay không các sơ đồ hệ thống được ngầm định đưa vào như pg_catalog được đưa vào trong đường tìm kiếm được trả về.

Lưu ý: Đường tìm kiếm có thể được tùy chỉnh theo thời gian chạy. Lệnh đó là:

```
SET search_path TO schema [, schema, ...]
```

pg_listening_channels trả về một tập hợp các tên các kênh mà phiên hiện hành đang nghe. Xem LISTEN để có thêm thông tin.

inet_client_addr trả về địa chỉ IP của máy trạm hiện hành, và inet_client_port trả về số cổng. inet_server_addr trả về địa chỉ IP trong đó máy chủ đó đã chấp nhận kết nối hiện hành, và inet_server_port trả về số cổng. Tất cả các hàm đó trả về NULL nếu kết nối hiện hành là thông qua một socket miền Unix.

pg_my_temp_schema trả về OID của sơ đồ tạm thời phiên hiện hành, hoặc 0 nếu nó không có (vì nó đã không tạo ra bất kỳ bảng tạm nào). pg_is_other_temp_schema trả về true nếu OID được đưa ra là OID của một sơ đồ tạm của phiên khác. (Điều này có thể được sử dụng, ví dụ, để loại trừ các bảng tạm của phiên khác khỏi hiển thị trong một catalog).

pg_postmaster_start_time trả về timestamp with time zone khi máy chủ được khởi động.

pg_conf_load_time trả về timestamp with time zone khi các tệp cấu hình máy chủ đã được tải lên mới nhất. (Nếu phiên hiện hành là sống khi đó, thì đây sẽ là thời gian khi bản thân phiên đó đọc lại các tệp cấu hình, sao cho việc đọc sẽ khác nhau một chút trong các phiên khác nhau. Nếu không thì đây là thời gian khi qui trình chủ (postmaster) đọc lại các tệp cấu hình).

version trả về một chuỗi mô tả phiên bản máy chủ PostgreSQL.

Bảng 9-48 liệt kê các hàm cho phép người sử dụng truy vấn các quyền ưu tiên truy cập đối tượng lập trình. Xem Phần 5.6 để có thêm thông tin về các quyền ưu tiên.

Bảng 9-48. Các hàm yêu cầu quyền ưu tiên truy cập

Hàm	Dạng trả về	Mô tả
has_any_column_privilege(user, table, privilege)	boolean	người sử dụng có quyền ưu tiên cho bất kỳ cột nào của bảng
has_any_column_privilege(table, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho bất kỳ cột nào của bảng
has_column_privilege(user, table, column, privilege)	boolean	người sử dụng có quyền ưu tiên cho cột
has_column_privilege(table, column, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho cột
has_database_privilege(user, database, privilege)	boolean	người sử dụng có quyền ưu tiên cho cơ sở dữ liệu
has_database_privilege(database, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho cơ sở dữ liệu
has_foreign_data_wrapper_privilege(user, fdw, privilege)	boolean	người sử dụng có quyền ưu tiên cho trình gói lại dữ liệu bên ngoài
has_foreign_data_wrapper_privilege(fdw, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho trình gói lại dữ liệu bên ngoài
has_function_privilege(user, function, privilege)	boolean	người sử dụng có quyền ưu tiên cho hàm
has_function_privilege(function, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho hàm
has_language_privilege(user, language, privilege)	boolean	người sử dụng có quyền ưu tiên cho ngôn ngữ
has_language_privilege(language, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho ngôn ngữ
has_schema_privilege(user, schema, privilege)	boolean	người sử dụng có quyền ưu tiên cho sơ đồ
has_schema_privilege(schema, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho sơ đồ
has_server_privilege(user, server, privilege)	boolean	người sử dụng có quyền ưu tiên cho máy chủ ngoài
has_server_privilege(server, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho máy chủ ngoài
has_sequence_privilege(user, sequence, privilege)	boolean	người sử dụng có quyền ưu tiên cho sự tuần tự
has_sequence_privilege(sequence, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho sự tuần tự
has_table_privilege(user, table, privilege)	boolean	người sử dụng có quyền ưu tiên cho bảng
has_table_privilege(table, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho bảng
has_tablespace_privilege(user, tablespace, privilege)	boolean	người sử dụng có quyền ưu tiên cho không gian bảng
has_tablespace_privilege(tablespace, privilege)	boolean	người sử dụng hiện hành có quyền ưu tiên cho không gian bảng
pg_has_role(user, role, privilege)	boolean	người sử dụng có quyền ưu tiên cho vai trò

has_table_privilege kiểm tra liệu một người sử dụng có thể truy cập một bảng theo một cách thức đặc biệt hay không. Người sử dụng có thể được chỉ định bằng tên hoặc bằng OID (pg_authid.oid), hoặc nếu đối số bị làm mờ current_user được giả thiết. Bảng đó có thể được chỉ định bằng tên hoặc bằng OID. (Vì thế, thực sự có 6 phương án của has_table_privilege, nó có thể nhờ số lượng dạng các đối số của chúng phân biệt). Khi chỉ định bằng tên, thì tên đó có thể được định tính theo sơ đồ nếu cần

thiết. Dạng quyền ưu tiên truy cập mong muốn được chỉ định bằng một chuỗi văn bản, nó phải đánh giá theo một trong các giá trị SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES hoặc TRIGGER. Như một sự lựa chọn, WITH GRANT OPTION có thể được bổ sung vào một dạng quyền ưu tiên để kiểm thử liệu quyền ưu tiên đó có được giữ với lựa chọn được trao hay không. Hơn nữa, các dạng nhiều quyền ưu tiên có thể được liệt kê tách bạch nhau bằng dấu phẩy, trong trường hợp đó kết quả sẽ là true nếu bất kỳ quyền ưu tiên được liệt kê nào được giữ. (Trường hợp chuỗi quyền ưu tiên là không đáng kể, và các dấu trắng thừa ra được phép ở giữa nhưng không bên trong các tên quyền ưu tiên). Một số ví dụ:

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');
```

has_sequence_privilege kiểm tra liệu một người sử dụng có thể truy cập một sự tuần tự theo một cách thức đặc biệt hay không. Các khả năng cho các đối số của nó là tương tự như has_table_privilege. Dạng quyền ưu tiên truy cập mong muốn phải đánh giá với một trong số USAGE, SELECT hoặc UPDATE.

has_any_column_privilege kiểm tra liệu một người sử dụng có thể truy cập bất kỳ cột nào của một bảng theo một cách thức đặc biệt hay không. Các khả năng đối số của nó là tương tự với has_table_privilege, ngoại trừ là dạng quyền ưu tiên truy cập mong muốn phải đánh giá một số kết hợp của SELECT, INSERT, UPDATE hoặc REFERENCES. Lưu ý rằng việc có bất kỳ các quyền ưu tiên đó ở mức bảng ngầm trao nó cho từng cột của bảng đó, nên has_any_column_privilege sẽ luôn trả về true nếu has_table_privilege làm vì các đối số y hệt. Nhưng has_any_column_privilege cũng thành công nếu có một sự trao mức cột quyền ưu tiên cho ít nhất một cột.

has_column_privilege kiểm tra liệu một người sử dụng có thể truy cập một cột theo một cách thức đặc biệt hay không. Các khả năng đối số của nó là tương tự với has_table_privilege, với sự bổ sung rằng cột đó có thể được chỉ định hoặc bằng tên hoặc số thuộc tính. Dạng quyền ưu tiên truy cập mong muốn phải đánh giá với một số kết hợp của SELECT, INSERT, UPDATE hoặc REFERENCES. Lưu ý rằng việc có bất kỳ quyền ưu tiên đó ở mức bảng ngầm định trao nó cho từng cột của bảng đó.

has_database_privilege kiểm tra liệu một người sử dụng có thể truy cập được một cơ sở dữ liệu theo một cách thức đặc biệt hay không. Các khả năng đối số của nó là tương tự với has_table_privilege. Dạng các quyền ưu tiên truy cập mong muốn phải đánh giá với một số kết hợp của CREATE, CONNECT, TEMPORARY hoặc TEMP (nó là tương đương với TEMPORARY).

has_function_privilege kiểm tra liệu một người sử dụng có thể truy cập một hàm theo một cách thức đặc biệt hay không. Các khả năng đối số của nó là tương tự với has_table_privilege. Khi chỉ định một hàm bằng một chuỗi văn bản thay vì bằng OID, đầu vào được phép là tên như đối với dạng dữ liệu regprocedure (xem Phần 8.16). Dạng quyền ưu tiên truy cập mong muốn phải đánh giá với EXECUTE. Một ví dụ là:

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

has_foreign_data_wrapper_privilege kiểm tra liệu một người sử dụng có thể truy cập một bộ đóng gói dữ liệu ngoài theo một cách đặc biệt hay không. Các khả năng đối số của nó là tương tự với

has_table_privilege. Dạng quyền ưu tiên truy cập mong muốn phải đánh giá với USAGE.

has_language_privilege kiểm tra liệu một người sử dụng có thể truy cập một ngôn ngữ thủ tục theo một cách thức đặc biệt hay không. Các khả năng đối số của nó là tương tự với has_table_privilege. Dạng quyền ưu tiên truy cập mong muốn phải đánh giá với USAGE.

has_schema_privilege kiểm tra liệu một người sử dụng có thể truy cập một sơ đồ theo một cách đặc biệt hay không. Các khả năng đối số của nó là tương tự với has_table_privilege. Dạng quyền ưu tiên truy cập mong muốn phải đánh giá với một số kết hợp của CREATE hoặc USAGE.

has_server_privilege kiểm tra liệu một người sử dụng có thể truy cập được một máy chủ ngoài theo một cách đặc biệt hay không. Các khả năng đối số của nó là tương tự với has_table_privilege. Dạng quyền ưu tiên truy cập mong muốn phải đánh giá theo USAGE.

has_tablespace_privilege kiểm tra liệu một người sử dụng có thể truy cập một không gian bảng theo một cách thức đặc biệt hay không. Các khả năng đối số của nó là tương tự với has_table_privilege. Dạng quyền ưu tiên truy cập mong muốn phải đánh giá theo CREATE.

pg_has_role kiểm tra liệu một người sử dụng có thể truy cập một vai trò theo một cách đặc biệt hay không. Các khả năng đối số của nó là tương tự với has_table_privilege. Dạng quyền ưu tiên truy cập mong muốn phải đánh giá theo một số kết hợp của MEMBER hoặc USAGE. MEMBER biểu thị quan hệ thành viên trực tiếp hoặc gián tiếp trong vai trò đó (đó là, quyền để thực hiện SET ROLE), trong khi USAGE biểu thị việc liệu các quyền ưu tiên của vai trò đó có sẵn sàng ngay lập tức mà không phải thực hiện SET ROLE hay không.

Bảng 9-49 chỉ ra các hàm xác định liệu một đối tượng nhất định có là trực quan hay không trong đường tìm kiếm sơ đồ hiện hành. Ví dụ, một bảng được nói sẽ là trực quan nếu sơ đồ bao gồm của nó là trong đường tìm kiếm và không có bảng cùng y hệt tên nào xuất hiện trước đó trong đường tìm kiếm đó. Điều này tương đương với tuyên bố rằng bảng đó có thể được tham chiếu tới bằng tên không có khả năng sơ đồ rõ ràng. Để liệt kê các tên của tất cả các bảng nhìn thấy được:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Bảng 9-49. Các hàm đòi hỏi sự trực quan sơ đồ

Hàm	Dạng trả về	Mô tả
pg_conversion_is_visible(conversion_oid)	boolean	là biến đổi nhìn thấy trong đường tìm kiếm
pg_function_is_visible(function_oid)	boolean	là hàm nhìn thấy trong đường tìm kiếm
pg_operator_is_visible(operator_oid)	boolean	là toán tử nhìn thấy trong đường tìm kiếm
pg_opclass_is_visible(opclass_oid)	boolean	là lớp toán tử nhìn thấy trong đường tìm kiếm
pg_table_is_visible(table_oid)	boolean	là bảng nhìn thấy trong đường tìm kiếm
pg_ts_config_is_visible(config_oid)	boolean	là cấu hình tìm kiếm văn bản nhìn thấy trong đường tìm kiếm
pg_ts_dict_is_visible(dict_oid)	boolean	là thư mục tìm kiếm văn bản nhìn thấy trong đường tìm kiếm
pg_ts_parser_is_visible(parser_oie)	boolean	là trình phân tích tìm kiếm văn bản nhìn thấy trong

Hàm	Dạng trả về	Mô tả
		đường tìm kiếm
pg_ts_template_is_visible(templateoid)	boolean	là mẫu template tìm kiếm văn bản nhìn thấy trong đường tìm kiếm
pg_type_is_visible(type_oid)	boolean	là dạng (hoặc miền) nhìn thấy trong đường tìm kiếm

Từng hàm thực thi kiểm tra tính trực quan cho một dạng đối tượng cơ sở dữ liệu. Lưu ý rằng pg_table_is_visible cũng có thể được sử dụng với các kiểu nhìn, các chỉ số và các tuần tự; pg_type_is_visible cũng có thể được sử dụng với các miền. Đối với các hàm và toán tử, một đối tượng trong đường tìm kiếm là nhìn thấy nếu không có đối tượng nào cùng tên y hệt và (các) dạng dữ liệu của đối số trước đó trong đường tìm kiếm. Đối với các lớp toán tử, cả tên và phương pháp truy cập chỉ số có liên quan được xem xét.

Tất cả các hàm đó đòi hỏi các OID đối tượng để nhận diện đối tượng sẽ được kiểm tra. Nếu bạn muốn kiểm tra một đối tượng theo tên, là thuận tiện để sử dụng các dạng tên hiệu (alias) OID (regclass, regtype, regprocedure, regoperator, regconfig hoặc regdictionary), ví dụ:

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Lưu ý là có thể không có nhiều ý nghĩa để kiểm thử một tên dạng không được định tính theo sơ đồ theo cách này - nếu tên đó có thể được thừa nhận, thì nó phải là nhìn thấy được.

Bảng 9-50 liệt kê các hàm trích thông tin từ các catalog hệ thống.

Bảng 9-50. Các hàm thông tin catalog hệ thống.

Tên	Dạng trả về	Mô tả
format_type(type_oid, typemod)	text	có tên SQL của một dạng dữ liệu
pg_get_keywords()	setof record	có danh sách các từ khóa SQL và các chủng loại của chúng
pg_get_constraintdef(constrainoid)	text	có định nghĩa của một ràng buộc
pg_get_constraintdef(constraint_oid, pretty_bool)	text	có định nghĩa của một ràng buộc
pg_get_expr(expr_text, relation_oid)	text	dịch ngược mẫu nội bộ của biểu thức, giả thiết là bất kỳ Vars nào trong đó cũng tham chiếu tới quan hệ được tham số thứ 2 chỉ ra
pg_get_expr(expr_text, relation_oid, pretty_bool)	text	dịch ngược mẫu nội bộ của biểu thức, giả thiết là bất kỳ Vars nào trong đó cũng tham chiếu tới quan hệ được tham số thứ 2 chỉ ra
pg_get_functiondef(func_oid)	text	có định nghĩa của một hàm
pg_get_function_arguments(func_oid)	text	có danh sách đối số định nghĩa hàm (với các giá trị mặc định)
pg_get_function_identity_arguments(func_oid)	text	có danh sách đối số để nhận diện hàm (với các giá trị mặc định)
pg_get_function_result(func_oid)	text	có mệnh đề RETURNS cho hàm
pg_get_indexdef(index_oid)	text	có lệnh CREATE INDEX cho chỉ số
pg_get_indexdef(index_oid, column_no, pretty_bool)	text	có lệnh CREATE INDEX cho chỉ số, hoặc định nghĩa của chỉ một cột chỉ số khi column_no không là 0

Tên	Dạng trả về	Mô tả
pg_get_ruledef(rule_oid)	text	có lệnh CREATE RULE cho qui tắc
pg_get_ruledef(rule_oid, pretty_bool)	text	có lệnh CREATE RULE cho qui tắc
pg_get_serial_sequence(table_name, column_name)	text	có tên sự tuần tự mà một cột serial hoặc bigserial sử dụng
pg_get_triggerdef(trigger_oid)	text	có lệnh CREATE [CONSTRAINT] TRIGGER cho trigger
pg_get_triggerdef(trigger_oid, pretty_bool)	text	có lệnh CREATE [CONSTRAINT] TRIGGER cho trigger
pg_get_userbyid(role_oid)	name	có tên vai trò với OID được đưa ra
pg_get_viewdef(view_name)	text	có lệnh SELECT nằm bên dưới để xem (được yêu cầu)
pg_get_viewdef(view_name, pretty_bool)	text	có lệnh SELECT nằm bên dưới để xem (được yêu cầu)
pg_get_viewdef(view_oid)	text	có lệnh SELECT nằm bên dưới để xem
pg_get_viewdef(view_oid, pretty_bool)	text	có lệnh SELECT nằm bên dưới để xem
pg_tablespace_databases(tablespace_oid)	setof oid	có tập hợp các OID cơ sở dữ liệu mà có các đối tượng trong không gian bảng
pg_typeof(any)	regtype	có dạng dữ liệu của bất kỳ giá trị nào

format_type trả về tên dạng dữ liệu theo SQL được dạng OID và có khả năng một trình sửa đổi dạng của nó nhận diện. Hãy truyền NULL qua cho trình sửa đổi dạng nếu không có trình sửa đổi đặc biệt nào được biết.

pg_get_keywords trả về một tập hợp các bản ghi mô tả các từ khóa SQL được máy chủ thừa nhận. Cột word chứa các từ khóa. Cột catcode chứa một mã chủng loại: U cho không được duy trì, C cho tên cột, T cho dạng hoặc tên hàm, hoặc R cho được duy trì. Cột catdesc có một chuỗi có khả năng bản địa hóa mô tả chủng loại đó.

pg_get_constraintdef, pg_get_indexdef, pg_get_ruledef và pg_get_triggerdef một cách tương ứng, tái tạo lại lệnh tạo một ràng buộc, chỉ số, qui tắc hoặc trigger. (Lưu ý rằng đây là một sự tái tạo lại được biên dịch ngược, không phải văn bản lệnh gốc ban đầu). pg_get_expr dịch ngược dạng nội bộ của một biểu thức riêng rẽ, như giá trị mặc định cho một cột. Có thể là hữu dụng khi xem xét các nội dung của các catalog hệ thống. Nếu biểu thức đó có thể chứa các Vars, hãy chỉ định OID của quan hệ mà chúng tham chiếu tới như là tham số thứ 2; nếu không Vars nào được kỳ vọng, thì 0 là đủ. pg_get_viewdef tái tạo truy vấn SELECT mà xác định một kiểu nhìn. Hầu hết các hàm đi với 2 phương án, một trong số đó có thể tùy chọn kết quả "in khá". Định dạng in được khá đọc được nhiều hơn, nhưng định dạng mặc định là có khả năng hơn để được diễn giải theo cách y hệt trong các phiên bản trong tương lai của PostgreSQL; tránh sử dụng đầu ra in được khá cho các mục đích bỏ (dump). Việc truyền false qua cho tham số in được khá đưa ra kết quả y hệt như phương án mà không có tham số hoàn toàn.

pg_get_functiondef trả về một lệnh hoàn chỉnh CREATE OR REPLACE FUNCTION cho một hàm. pg_get_function_arguments trả về danh sách đối số của một hàm, ở dạng nó có thể cần xuất hiện trong CREATE FUNCTION. pg_get_function_result tương tự trả về mệnh đề RETURNS phù hợp cho hàm đó. pg_get_function_identity_arguments trả về danh sách đối số cần thiết để nhận diện một hàm, ở dạng nó có thể cần phải xuất hiện trong ALTER FUNCTION, ví dụ thế. Dạng này làm mờ các giá trị mặc định.

pg_get_serial_sequence trả về tên của tuần tự có liên quan tới một cột, hoặc NULL nếu không có sự tuần tự nào có liên quan tới cột đó. Tham số đầu vào đầu tiên là một tên bảng với sơ đồ tùy chọn, và tham số thứ hai là một tên cột. Vì tham số thứ nhất tiềm tàng là một sơ đồ và bảng, nó sẽ không được đối xử như một mã định danh trong dấu ngoặc kép, nghĩa là nó ở dạng các chữ thường một cách mặc định, trong khi tham số thứ 2, chỉ là tên một cột, sẽ được đối xử như trong các dấu ngoặc kép và giữ lại được dạng chữ của nó. Hàm đó trả về một giá trị được định dạng phù hợp cho việc truyền tới các hàm tuần tự (xem Phần 9.15). Sự liên quan này có thể được sửa đổi hoặc loại bỏ bằng ALTER SEQUENCE OWNED BY. (Hàm đó có khả năng sẽ được gọi là pg_get_owned_sequence; tên hiện hành của nó phản ánh thực tế rằng nó thường được sử dụng với các cột serial hoặc bigserial).

pg_get_userbyid trích một tên vai trò đưa ra OID của nó.

pg_tablespace_databases cho phép một không gian bảng sẽ được kiểm tra. Nó trả về tập hợp các OID các cơ sở dữ liệu mà có các đối tượng được lưu trữ trong không gian bảng. Nếu hàm này trả về bất kỳ hàng nào, thì không gian bảng không rỗng và không thể bị loại bỏ. Để hiển thị các đối tượng đặc biệt đưa ra không gian bảng, bạn sẽ cần phải kết nối tới các cơ sở dữ liệu được nhận diện bằng pg_tablespace_databases và truy vấn các catalog pg_class của chúng.

pg_typeof trả về OID dạng dữ liệu của giá trị được truyền qua nó. Điều này có thể là hữu dụng cho việc xử lý sự cố hoặc xây dựng một cách động các truy vấn SQL. Hàm đó được khai báo như việc trả về regtype, nó là một dạng tên hiệu OID (xem Phần 8.16); điều này có nghĩa là nó là y hệt như một OID cho các mục đích so sánh nhưng hiển thị như một tên dạng. Ví dụ:

```
SELECT pg_typeof(33);

pg_typeof
-----------
integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);

typlen
--------
4
(1 row)
```

Các hàm chỉ ra trong Bảng 9-51 trích các chú giải trước đó được lưu giữ với lệnh COMMENT.

Một giá trị null được trả về nếu không chú giải nào có thể được tìm thấy cho các tham số được chỉ định.

Bảng 9-51. Các hàm thông tin chú giải

Tên	Dạng trả về	Mô tả
col_description(table_oid, column_number)	text	có chú giải cho một cột của bảng
obj_description(object_oid, catalog_name)	text	có chú giải cho một đối tượng cơ sở dữ liệu
obj_description(object_oid)	text	có chú giải cho một đối tượng cơ sở dữ liệu (được yêu cầu)
shobj_description(object_oid, catalog_name)	text	có chú giải cho một đối tượng cơ sở dữ liệu được chia sẻ

col_description trả về chú giải cho một cột của bảng, nó được OID bảng của nó và số cột của nó chỉ định. obj_description không thể được sử dụng cho các cột của bảng vì các cột không có các OID của riêng chúng.

Mẫu 2 tham số của obj_description trả về chú giải cho một đối tượng cơ sở dữ liệu được OID và tên của catalog hệ thống chứa nó chỉ định. Ví dụ, obj_description(123456,'pg_class') có thể truy xuất chú giải cho bảng với OID 123456. Mẫu một tham số của obj_description chỉ đòi hỏi OID đối tượng. Nó được yêu cầu vì không có đảm bảo rằng OID là độc nhất xuyên khắp catalog các hệ thống khác nhau; vì thế, chú giải sai có thể được trả về.

shobj_description được sử dụng chỉ giống như obj_description ngoại trừ nó được sử dụng cho việc truy xuất các chú giải về các đối tượng được chia sẻ. Một số catalog là tổng thể cho tất cả các cơ sở dữ liệu bên trong từng bó và các mô tả của chúng được lưu giữ cũng bằng cách tổng thể.

Các hàm được chỉ ra trong Bảng 9-52 đưa ra thông tin giao dịch máy chủ ở một mẫu có thể xuất được. Sự sử dụng chính các hàm đó là để xác định các giao dịch nào đã được thực hiện giữa 2 hình chụp màn hình.

Bảng 9-52. Các ID giao dịch và các hình chụp màn hình

Tên	Dạng trả về	Mô tả
txid_current()	bigint	có ID giao dịch hiện hành
txid_current_snapshot()	txid_snapshot	có hình chụp màn hình hiện hành
txid_snapshot_xmin(txid_snapshot)	bigint	có xmin của hình chụp màn hình
txid_snapshot_xmax(txid_snapshot)	bigint	có xmax của hình chụp màn hình
txid_snapshot_xip(txid_snapshot)	setof bigint	có các ID giao dịch đang tiến triển trong hình chụp màn hình
txid_visible_in_snapshot(bigint, txid_snapshot)	boolean	là ID giao dịch nhìn thấy được trong hình chụp màn hình? (không sử dụng với các ids các giao dịch con)

Dạng ID giao dịch nội bộ (xid) là 32 bit rộng và bao bọc khoảng mỗi 4 tỷ giao dịch. Tuy nhiên, các giao dịch đó xuất khẩu một định dạng 64 bit mà được mở rộng với một bộ đếm "thế kỷ" sao cho nó sẽ không bao quanh trong khi đang cài đặt. Dạng dữ liệu được các hàm đó sử dụng, txid_snapshot, lưu trữ thông tin về khả năng nhìn thấy ID giao dịch ở một thời điểm đặc biệt theo thời gian.

Các thành phần của nó được mô tả trong Bảng 9-53.

Bảng 9-53. Các chú giải hình chụp màn hình.

Tên	Mô tả
xmin	ID giao dịch sớm nhất (txid) mà vẫn còn hoạt động. Tất cả các giao dịch sớm hơn sẽ hoặc được đệ trình và nhìn thấy, hoặc được quay trở ngược lại và chết.
xmax	txid còn chưa được ký đầu tiên. Tất cả các txids lớn hơn hoặc bằng thứ này còn chưa được khởi động vào thời gian của hình chụp màn hình, và vì thế không nhìn thấy.
xip_list	txids tích cực vào thời điểm của hình chụp màn hình. Danh sách bao gồm chỉ những txids tích cực giữa xmin và xmax; có thể có các txids tích cực cao hơn so với xmax. Một txid mà là xmin <= txid < xmax và không nằm trong danh sách này từng phức tạp rồi vào thời điểm của hình chụp màn hình, và vì thế hoặc nhìn thấy hoặc chết tùy theo tình trạng đệ trình của nó. Danh sách đó không bao gồm txids của các giao dịch.

Sự diễn tả văn bản của txid_snapshot là xmin:xmax:xip_list. Ví dụ 10:20:10,14,15 nghĩa là xmin=10, xmax=20, xip_list=10, 14, 15.

9.24. Hàm quản trị hệ thống

Bảng 9-54 chỉ ra các hàm sẵn sàng để truy vấn và tùy biến các tham số cấu hình theo thời gian chạy. (run-time).

Bảng 9-54. Các hàm thiết lập cấu hình

Tên	Dạng trả về	Mô tả
current_setting(setting_name)	text	có giá trị của thiết lập hiện hành
set_config(setting_name, new_value, is_local)	text	thiết lập tham số và trả về giá trị mới

Hàm current_setting có giá trị hiện hành của thiết lập setting_name. Nó tương ứng với lệnh SQL SHOW. Một ví dụ:

```
SELECT current_setting('datestyle');

current_setting
-----------------
ISO, MDY
(1 row)
```

set_config thiết lập tham số setting_name cho new_value. Nếu is_local là true, thì giá trị mới đó sẽ chỉ áp dụng cho giao dịch hiện hành. Nếu bạn muốn giá trị mới đó áp dụng cho phiên hiện hành, hãy sử dụng false thay vào đó. Hàm đó tương ứng với lệnh SQL SET. Một ví dụ:

```
SELECT set_config('log_statement_stats', 'off', false);

set_config
------------
off
(1 row)
```

Hàm đó được chỉ ra trong Bảng 9-55 gửi đi các tín hiệu kiểm soát cho các tiến trình khác của máy chủ. Sử dụng các hàm đó bị hạn chế chỉ cho các siêu người sử dụng.

Bảng 9-55. Các hàm đánh tín hiệu cho máy chủ

Tên	Dạng trả về	Mô tả
pg_cancel_backend(pid int)	boolean	Hoãn một truy vấn hiện hành ở phần phụ trợ (backend)
pg_terminate_backend(pid int)	boolean	Kết thúc một phần phụ trợ
pg_reload_conf()	boolean	Làm cho các tiến trình máy chủ tải lại các tệp cấu hình của chúng.
pg_rotate_logfile()	boolean	Xoay tệp lưu ký của máy chủ

Từng trong số các hàm đó trả về true nếu thành công và false nếu là khác.

pg_cancel_backend và pg_terminate_backend gửi các tín hiệu (SIGINT hoặc SIGTERM một cách tương ứng) tới các tiến trình phần phụ trợ (backend) được ID tiến trình xác định. ID tiến trình của một phần phụ trợ tích cực có thể thấy từ cột procpid của kiểu nhìn pg_stat_activity, hoặc bằng việc liệt kê các tiến trình postgres trên máy chủ (bằng việc sử dụng ps trên Unix hoặc Task Manager - Trình quản lý tác vụ trên Windows).

pg_reload_conf gửi một tín hiệu SIGHUP tới máy chủ, làm cho các tệp cấu hình sẽ được tất cả các tiến trình của máy chủ tải lại.

pg_rotate_logfile đánh tín hiệu cho trình quản lý tệp lưu ký để chuyển tới một tệp đầu ra mới ngay lập tức. Điều này chỉ làm việc khi bộ thu thập lưu ký được xây dựng sẵn rồi đang chạy, vì nếu khác thì sẽ không có các tiến trình con quản lý tệp lưu ký.

Các hàm được chỉ ra trong Bảng 9-56 hỗ trợ trong việc tiến hành các sao lưu trực tuyến. Các hàm đó không thể được thực thi trong quá trình phục hồi. Sử dụng 3 hàm đầu tiên bị hạn chế chỉ cho các siêu người sử dụng (superusers).

Bảng 9-56. Các hàm kiểm soát sao lưu

Tên	Dạng trả về	Mô tả
pg_start_backup(label text [, fast boolean])	text	Chuẩn bị cho việc thực thi sao lưu trực tuyến
pg_stop_backup()	text	Kết thúc việc thực thi sao lưu trực tuyến
pg_switch_xlog()	text	Ép chuyển sang một tệp lưu ký giao dịch mới
pg_current_xlog_location()	text	Có vị trí viết lưu ký giao dịch hiện hành
pg_current_xlog_insert_location()	text	Có vị trí chèn lưu ký giao dịch hiện hành
pg_xlogfile_name_offset(location text)	text, integer	Biến đổi chuỗi vị trí lưu ký giao dịch thành tên tệp và phần bù thập phân byte trong tệp
pg_xlogfile_name(location text)	text	Biến đổi chuỗi vị trí lưu ký thành tên tệp

pg_start_backup chấp nhận một nhãn tùy ý do người sử dụng định nghĩa cho sao lưu. (Thường thì điều này có thể là tên theo đó tệp dump sao lưu sẽ được lưu trữ). Hàm đó viết một tệp nhãn sao lưu (backup_label) vào thư mục dữ liệu của bó cơ sở dữ liệu đó, thực thi một điểm kiểm tra, và sau đó trả về cho việc khởi đầu sao lưu vị trí lưu ký giao dịch như là văn bản. Người sử dụng có thể bỏ qua

giá trị kết quả này, nhưng nó được cung cấp trong trường hợp điều đó là hữu dụng.

```
postgres=# select pg_start_backup('label_goes_here');

pg_start_backup
-----------------
0/D4445B8
(1 row)
```

Có một tham số lựa chọn thứ 2 dạng boolean. Nếu true, nó chỉ định việc thực thi pg_start_backup càng nhanh càng tốt. Điều này ép một điểm kiểm tra ngay lập tức mà sẽ gây ra một sự tăng đột biến trong các hoạt động I/O, làm chậm lại bất kỳ truy vấn đang đồng thời thực thi nào.

pg_stop_backup loại bỏ tệp nhãn được pg_start_backup tạo ra, và tạo ra một tệp lịch sử sao lưu trong khu vực lưu trữ lưu ký giao dịch. Tệp lịch sử đó bao gồm nhãn được đưa ra cho pg_start_backup, các vị trí bắt đầu và kết thúc lưu ký giao dịch để sao lưu, và thời điểm bắt đầu và kết thúc sự sao lưu đó. Giá trị trả về là vị trí lưu ký giao dịch kết thúc của sự sao lưu đó (nó một lần nữa có thể bị bỏ qua). Sau việc ghi lại vị trí kết thúc, điểm chèn lưu ký giao dịch hiện hành được tự động tiến đến tệp lưu ký giao dịch tiếp sau, sao cho tệp lưu ký giao dịch kết thúc có thể được lưu trữ ngay lập tức để hoàn tất sao lưu.

pg_switch_xlog chuyển sang tệp lưu ký giao dịch tiếp sau, cho phép tệp hiện hành sẽ được lưu trữ (giả thiết bạn đang sử dụng việc lưu trữ liên tục). Giá trị trả về là vị trí lưu ký giao dịch kết thúc +1 bên trong tệp lưu ký giao dịch vừa được hoàn tất. Nếu chưa từng có hoạt động lưu ký giao dịch nào kể từ sự chuyển lưu ký giao dịch cuối cùng, thì pg_switch_xlog không làm gì cả và trả về vị trí ban đầu tệp lưu ký giao dịch hiện đang được sử dụng.

pg_current_xlog_location hiển thị vị trí ghi lưu ký giao dịch hiện hành trong cùng một định dạng được các hàm ở trên sử dụng. Tương tự, pg_current_xlog_insert_location hiển thị điểm chèn lưu ký giao dịch hiện hành. Điểm chèn đó là kết thúc "logic" của lưu ký giao dịch trong bất kỳ sự kiện nào, trong khi vị trí ghi là kết thúc của những gì thực sự từng được ghi ra từ bộ nhớ đệm (buffer) nội bộ của máy chủ, và thường là những gì bạn muốn nếu bạn có quan tâm trong việc lưu trữ các tệp lưu ký giao dịch được hoàn tất một phần. Điểm chèn được làm cho sẵn sàng trước hết cho các mục đích gỡ lỗi của máy chủ. Chúng là cả các hoạt động chỉ đọc và không đòi hỏi các quyền của siêu người sử dụng.

Bạn có thể sử dụng pg_xlogfile_name_offset để trích tên và phần bù theo byte các tệp lưu ký giao dịch tương ứng từ các kết quả của bất kỳ hàm nào ở trên. Ví dụ:

```
postgres=# SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());

file_name | file_offset
--------------------------+-------------
00000001000000000000000D | 4039624
(1 row)
```

Tương tự, pg_xlogfile_name trích chỉ tên tệp lưu ký giao dịch. Khi vị trí lưu ký giao dịch được đưa ra chính xác trong một giới hạn tệp lưu ký giao dịch, thì cả các hàm đó đều trả về tên của tệp lưu ký giao dịch trước đó. Điều này thường là hành vi được mong muốn cho việc quản lý hành vi lưu trữ lưu ký giao dịch, vì tệp trước đó là tệp cuối cùng mà hiện hành cần phải được lưu trữ.

Để có các chi tiết về sử dụng đúng các hàm đó, xem Phần 24.3.

Các hàm được chỉ ra trong Bảng 9-57 đưa ra thông tin về hiện trạng của chế độ chờ. Các hàm đó có thể được thực thi trong quá trình phục hồi và trong khi chạy bình thường.

Bảng 9-57. Các hàm thông tin phục hồi

Tên	Dạng trả về	Mô tả
pg_is_in_recovery()	bool	True nếu phục hồi vẫn còn đang diễn ra.
pg_last_xlog_receive_location()	text	Có vị trí lưu ký giao dịch mới nhất nhận được và được đồng bộ vào đĩa bằng nhân bản luồng. Trong khi sự nhân bản luồng đang diễn ra thì điều này sẽ làm gia tăng một cách đơn điệu. Nhưng khi sự nhân bản luồng được khởi động lại thì điều này sẽ quay trở lại với vị trí bắt đầu nhân bản, thường là bắt đầu tệp WAL có chứa vị trí chơi lại hiện hành. Nếu sự phục hồi đã hoàn tất thì điều này sẽ vẫn là tĩnh trong giá trị của bản ghi WAL mới nhất được nhận và được đồng bộ vào đĩa trong quá trình phục hồi. Nếu nhân bản luồng bị vô hiệu hóa, hoặc nếu nó còn chưa bắt đầu, thì hàm đó trả về NULL.
pg_last_xlog_replay_location()	text	Có vị trí lưu ký giao dịch được chơi lại trong quá trình phục hồi. Nếu sự phục hồi vẫn còn diễn ra thì điều này sẽ làm gia tăng một cách đơn điệu. Nếu sự phục hồi đã hoàn tất thì giá trị này vẫn giữ là tĩnh trong giá trị của bản ghi WAL mới nhất được áp dụng trong quá trình phục hồi đó. Khi máy chủ đã được khởi tạo bình thường mà không có phục hồi thì hàm đó trả về NULL.

Các hàm được chỉ ra trong Bảng 9-58 tính sử dụng không gian đĩa của các đối tượng cơ sở dữ liệu.

Bảng 9-58. Các hàm kích thước đối tượng cơ sở dữ liệu

Tên	Dạng trả về	Mô tả
pg_column_size(any)	int	Số byte được sử dụng để lưu trữ một giá trị đặc biệt (có thể được nén)
pg_total_relation_size(regclass)	bigint	Tổng không gian đĩa được bảng sử dụng với OID hoặc tên được chỉ định, bao gồm tất cả các chỉ số và dữ liệu TOAST
pg_table_size(regclass)	bigint	Không gian đĩa được bảng sử dụng với OID hoặc tên được chỉ định, ngoại trừ các chỉ số (nhưng bao gồm TOAST, bản đồ không gian tự do, và bản đồ nhìn thấy được)
pg_indexes_size(regclass)	bigint	Tổng không gian đĩa được các chỉ số sử dụng được gắn vào bảng với OID hoặc tên được chỉ định
pg_database_size(oid)	bigint	Không gian đĩa được cơ sở dữ liệu sử dụng với OID được chỉ định
pg_database_size(name)	bigint	Không gian đĩa được cơ sở dữ liệu sử dụng với OID được chỉ định
pg_tablespace_size(oid)	bigint	Không gian đĩa được cơ sở dữ liệu sử dụng với OID được chỉ định
pg_tablespace_size(name)	bigint	Không gian đĩa được cơ sở dữ liệu sử dụng với OID được chỉ định
pg_relation_size(relation regclass, fork text)	bigint	Không gian đĩa được bảng rẽ nhánh chỉ định (’main’, ’fsm’ or ’vm’) hoặc chỉ số với OID hoặc tên được chỉ định
pg_relation_size(relation regclass)	bigint	Viết tắt cho pg_relation_size(..., ’main’)
pg_size_pretty(bigint)	text	Biến đổi một kích thước dạng byte thành một định dạng người đọc được với các đơn vị của kích thước

pg_column_size chỉ ra không gian được sử dụng để lưu trữ bất kỳ giá trị dữ liệu riêng rẽ nào.

pg_total_relation_size chấp nhận OID hoặc tên của một bảng hoặc bảng toast, và trả về tổng không gian trên đĩa được sử dụng cho bảng đó, bao gồm tất cả các chỉ số có liên quan. Hàm này là tương đương với pg_table_size + pg_indexes_size.

pg_table_size chấp nhận OID hoặc tên của một bảng và trả về không gian đĩa cần thiết cho bảng đó, ngoại trừ các chỉ số. (Không gian TOAST, bản đồ không gian tự do, và bản đồ trực quan bao gồm).

pg_indexes_size chấp nhận OID hoặc tên của một bảng và trả về tổng không gian đĩa được tất cả các chỉ số sử dụng được gắn tới bảng đó.

pg_database_size và pg_tablespace_size chấp nhận OID hoặc tên của một bảng hoặc không gian bảng, và trả về tổng không gian đĩa được sử dụng ở trong đó.

pg_relation_size chấp nhận OID hoặc tên của một bảng, chỉ số hoặc bảng toast, và trả về kích cỡ trên đĩa theo các byte. Việc chỉ định 'main' hoặc để ra đối số thứ 2 trả về kích cỡ của rẽ nhánh dữ liệu chính của quan hệ đó. Việc chỉ định 'fsm' trả về kích cỡ của Bản đồ Không gian Tự do (xem Phần 54.3) có liên quan tới quan hệ này. Việc chỉ định 'vm' trả về kích cỡ của Bản đồ Trực quan (xem Phần 54.4) có liên quan tới quan hệ này. Lưu ý rằng hàm này chỉ ra kích cỡ của chỉ một rẽ nhánh; vì hầu hết các mục đích là thuận tiện hơn để sử dụng các hàm mức cao hơn pg_total_relation_size hoặc pg_table_size.

pg_size_pretty có thể được sử dụng để định dạng kết quả của một trong các hàm khác theo cách con người có thể đọc được, bằng việc sử dụng kB, MB, GB hoặc TB một cách phù hợp.

Các hàm được chỉ ra trong Bảng 9-59 hỗ trợ trong việc nhận diện các tệp đĩa đặc thù có liên quan tới các đối tượng cơ sở dữ liệu.

Bảng 9-59. Các hàm vị trí đối tượng cơ sở dữ liệu

Tên	Dạng trả về	Mô tả
pg_relation_filenode(relation regclass)	oid	Số nút tệp của quan hệ với OID hoặc tên được chỉ định
pg_relation_filepath(relation regclass)	text	Tên đường dẫn tệp của quan hệ với OID hoặc tên được chỉ định

pg_relation_filenode chấp nhận OID hoặc tên của một bảng, chỉ số, sự tuần tự hoặc bảng toast, và trả về số "nút tệp" (filenode) hiện được chỉ định cho nó. Nút tệp là thành phần cơ bản của (các) tên tệp được sử dụng cho quan hệ đó (xem Phần 54.l để có thêm thông tin). Đối với hầu hết các bảng thì kết quả là y hệt như pg_class .relfilenode, nhưng đối với các catalog hệ thống nhất định thì relfilenode là 0 và hàm này phải được sử dụng để có giá trị đúng. Hàm đó trả về NULL nếu đã truyền một quan hệ mà không có lưu trữ, như một kiểu nhìn chẳng hạn.

pg_relation_filepath là tương tự như pg_relation_filenode, nhưng nó trả về toàn bộ tên đường dẫn tệp (liên quan tới thư mục dữ liệu bó cơ sở dữ liệu PGDATA) của quan hệ đó.

Các hàm được chỉ ra trong Bảng 9-60 đưa ra sự truy cập bẩm sinh tới các tệp trên máy đặt chỗ

(hosting) cho máy chủ. Chỉ các tệp trong thư mục bó cơ sở dữ liệu và log_directory có thể được truy cập. Hãy sử dụng một đường dẫn tương đối cho các tệp trong thư mục bó đó, và một đường dẫn khớp với thiết lập cấu hình log_directory cho các tệp lưu ký. Sự sử dụng các hàm đó giới hạn cho các siêu người sử dụng.

Bảng 9-60. Các hàm truy cập tệp chung

Tên	Dạng trả về	Mô tả
pg_ls_dir(dirname text)	setof text	Liệt kê các nội dung của một thư mục
pg_read_file(filename text, offset bigint, length bigint)	text	Trả về các nội dung của một tệp văn bản
pg_stat_file(filename text)	record	Trả về thông tin về một tệp

pg_ls_dir trả về tất cả các tên trong thư mục được chỉ định, ngoại trừ các đầu vào đặc biệt “.” và “..”.

pg_read_file trả về một phần của một tệp văn bản, bắt đầu ở offset được đưa ra, trả về nhiều nhất length các byte (ít hơn nếu kết thúc tệp đạt tới được đầu tiên). Nếu offset là âm, thì nó là ở cuối tệp.

pg_stat_file trả về một bản ghi chứa kích thước tệp, dấu thời gian được truy cập lần cuối, dấu thời gian được sửa đổi lần cuối, dấu thời gian thay đổi tình trạng tệp lần cuối (chỉ cho các nền tảng Unix), dấu thời gian tạo tệp (chỉ cho Windows), và một boolean chỉ ra nếu đó là một thư mục. Các sử dụng điển hình bao gồm:

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

Các hàm được chỉ ra trong Bảng 9-61 quản lý các khóa cố vấn. Để có các chi tiết về sử dụng đúng các hàm đó, hãy xem Phần 13.3.4.

Bảng 9-61. Các hàm khóa cố vấn

Tên	Dạng trả về	Mô tả
pg_advisory_lock(key bigint)	void	Có được khóa cố vấn độc quyền
pg_advisory_lock(key1 int, key2 int)	void	Có được khóa cố vấn độc quyền
pg_advisory_lock_shared(key bigint)	void	Có được khóa cố vấn chia sẻ
pg_advisory_lock_shared(key1 int, key2 int)	void	Có được khóa cố vấn chia sẻ
pg_try_advisory_lock(key bigint)	boolean	Có được khóa cố vấn độc quyền nếu sẵn sàng
pg_try_advisory_lock(key1 int, key2 int)	boolean	Có được khóa cố vấn độc quyền nếu sẵn sàng
pg_try_advisory_lock_shared(key bigint)	boolean	Có được khóa cố vấn chia sẻ nếu sẵn sàng
pg_try_advisory_lock_shared(key1 int, key2 int)	boolean	Có được khóa cố vấn chia sẻ nếu sẵn sàng
pg_advisory_unlock(key bigint)	boolean	Đưa ra một khóa cố vấn độc quyền
pg_advisory_unlock(key1 int, key2 int)	boolean	Đưa ra một khóa cố vấn độc quyền
pg_advisory_unlock_shared(key bigint)	boolean	Đưa ra một khóa cố vấn chia sẻ
pg_advisory_unlock_shared(key1 int, key2 int)	boolean	Đưa ra một khóa cố vấn chia sẻ
pg_advisory_unlock_all()	void	Đưa ra tất cả các khóa cố vấn được phiên làm việc hiện hành nắm giữ

pg_advisory_lock khóa một tài nguyên do ứng dụng định nghĩa, nó có thể được xác định hoặc bằng một giá trị khóa 64 bit duy nhất hoặc 2 giá trị khóa 32 bit (lưu ý là 2 không gian khóa đó không chồng lấn nhau). Dạng khóa được chỉ định trong pg_locks.objid. Nếu phiên khác giữ một khóa trong cùng tài nguyên, thì hàm đó sẽ chờ cho tới khi tài nguyên đó sẵn sàng. Khóa đó là độc quyền. Nhiều khóa yêu cầu ngăn xếp (stack), sao cho nếu cùng y hệt tài nguyên bị khóa 3 lần thì nó cũng phải được mở khóa 3 lần để được đưa ra cho các phiên khác sử dụng.

pg_advisory_lock_shared làm việc y hệt như pg_advisory_lock, ngoại trừ khóa đó có thể được chia sẻ với các phiên khác đòi hỏi các khóa chia sẻ. Chỉ những khóa độc quyền có thể sẽ bị khóa.

pg_try_advisory_lock là tương tự như pg_advisory_lock, ngoại trừ hàm đó sẽ không chờ khóa đó trở nên sẵn sàng. Nó sẽ hoặc là giành lấy khóa đó ngay lập tức và trả về true, hoặc trả về false nếu khóa đó không thể có được ngay lập tức.

pg_try_advisory_lock_shared làm việc hệt như pg_try_advisory_lock, ngoại trừ là nó cố gắng có được một khóa chia sẻ hơn là khóa độc quyền.

pg_advisory_unlock sẽ đưa ra một khóa cố vấn độc quyền được yêu cầu trước đó. Nó trả về true nếu khóa đó được đưa ra thành công. Nếu khóa đó không nắm giữ được, thì nó sẽ trả về false, và hơn nữa, một cảnh báo SQL sẽ được máy chủ đưa ra.

pg_advisory_unlock_shared làm việc hệt như pg_advisory_unlock, ngoại trừ nó đưa ra một khóa cố vấn chia sẻ.

pg_advisory_unlock_all sẽ đưa ra tất cả các khóa cố vấn được phiên hiện hành nắm giữ. (Hàm này được triệu gọi ngầm ở cuối phiên, thậm chí nếu máy trạm bỏ kết nối một cách đột ngột).

9.25. Hàm Trigger

PostgreSQL hiện hành đưa ra một xây dựng trong hàm trigger, suppress_redundant_updates_trigger, nó sẽ ngăn cản sự cập nhật mà không thực sự thay đổi các dữ liệu trong hàng khỏi việc chiếm chỗ, ngược lại với hành vi thông thường mà luôn thực hiện cập nhật bất kể liệu có hay không các dữ liệu đã thay đổi. (Hành vi bình thường này làm cho các cập nhật chạy nhanh hơn, vì không có việc kiểm tra nào được yêu cầu, và cũng là hữu dụng hơn trong các trường hợp nhất định).

Lý tưởng, bạn thường nên tránh chạy các cập nhật không thực sự thay đổi các dữ liệu trong bản ghi. Các cập nhật dư thừa có thể lấy đi thời gian đáng kể không cần thiết, đặc biệt nếu có nhiều chỉ số phải tùy biến, và không gian trong các hàng chết cuối cùng sẽ phải được loại bỏ. Tuy nhiên, việc dò tìm ra được các tình huống như vậy trong mã của khách hàng không phải lúc nào cũng dễ dàng, hoặc thậm chí có thể, và việc viết các biểu thức để dò tìm ra chúng có thể bị lỗi. Một lựa chọn thay thế là hãy sử dụng suppress_redundant_updates_trigger, nó sẽ bỏ qua các cập nhật không thay đổi dữ liệu. Tuy nhiên, bạn nên sử dụng cái này cẩn thận. Trigger tốn ít thời gian mà không tầm thường cho từng bản ghi, vì thế nếu hầu hết các bản ghi bị ảnh hưởng vì một cập nhật thực sự được thay đổi, thì sử dụng trigger này sẽ thực sự làm cho cập nhật chạy chậm hơn.

Hàm suppress_redundant_updates_trigger có thể được bổ sung vào một bảng như thế này:

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE PROCEDURE suppress_redundant_updates_trigger();
```

Trong hầu hết các trường hợp, bạn có thể muốn sử dụng trigger này cuối cùng cho từng hàng. Nhớ trong đầu rằng các trigger sử dụng theo trật tự tên, bạn có thể sau đó chọn một tên trigger mà đi sau tên của bất kỳ trigger nào khác mà bạn có thể có trong bảng đó.

Để có thêm thông tin về việc tạo các trigger, hãy xem CREATE TRIGGER.

Chương 10. Biến đổi dạng

Các lệnh SQL có thể, cố ý hay không, đòi hỏi việc pha trộn các dạng dữ liệu khác nhau trong cùng y hệt một biểu thức. PostgreSQL có các cơ sở tăng cường cho việc đánh giá các biểu thức dạng được pha trộn.

Trong nhiều trường hợp người sử dụng không cần phải hiểu các chi tiết của cơ chế biến đổi dạng. Tuy nhiên, các biến đổi ngầm được PostgreSQL thực hiện có thể ảnh hưởng tới kết quả của truy vấn. Khi cần thiết, các kết quả đó có thể được tùy biến bằng việc sử dụng biến đổi dạng rõ ràng.

Chương này giới thiệu các cơ chế và các qui ước biến đổi dạng của PostgreSQL. Hãy tham chiếu tới các phần tương ứng trong Chương 8 và Chương 9 để có thêm thông tin về các dạng dữ liệu đặc thù và các hàm và toán tử được phép.

10.1. Tổng quan

SQL là một ngôn ngữ có khuôn dạng mạnh. Đó là, mỗi khoản dữ liệu có một dạng dữ liệu có liên quan mà xác định hành vi và sử dụng được phép của nó. PostgreSQL có một hệ thống dạng được tăng cường mà là chung và mềm dẻo hơn so với các triển khai SQL khác. Vì thế, hầu hết các hành vi biến đổi dạng trong PostgreSQL được các qui tắc chung điều chỉnh hơn là bằng các công nghệ đặc biệt. Điều này cho phép sử dụng các biểu thức dạng pha trộn thậm chí với các dạng do người sử dụng định nghĩa.

Các trình quét / trình phân tích của PostgreSQL chia các phần tử từ ngữ thành 5 chủng loại cơ bản: các số nguyên, các số không nguyên, các chuỗi, các mã định danh và các từ khóa. Các hằng của hầu hết các dạng không phải số trước hết được phân loại như là các chuỗi. Định nghĩa ngôn ngữ SQL cho phép chỉ định các tên dạng với các chuỗi, và cơ chế này có thể được sử dụng trong PostgreSQL để khởi tạo trình phân tích đi theo đường đúng. Ví dụ, truy vấn:

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

Có 2 hằng ký tự, dạng `text` và `point`. Nếu một dạng không được chỉ định cho một hằng chuỗi, thì dạng đặt chỗ sẵn `unknown` được chỉ định từ đầu, sẽ được giải quyết trong các giai đoạn sau như được mô tả bên dưới.

Có 4 cấu trúc SQL cơ bản đòi hỏi các qui tắc biến đổi dạng phân biệt nhau trong trình phân tích của PostgreSQL.

Các lời gọi hàm

> Nhiều hệ thống dạng PostgreSQL được xây dựng xung quanh một tập hợp giàu có các hàm. Các hàm có thể có một hoặc nhiều đối số. Vì PostgreSQL cho phép hàm quá tải, nên tên của hàm đó một mình không duy nhất nhận diện hàm sẽ được gọi; trình phân tích phải chọn hàm

đúng dựa vào các dạng dữ liệu của các đối số được cung cấp.

Các toán tử

PostgreSQL cho phép các biểu thức với các toán tử unary tiền tố và hậu tố (một đối số), cũng như các toán tử nhị phân (2 đối số). Giống như các hàm, các toán tử có thể sẽ quá tải, nên vấn đề y hệt của việc chọn đúng toán tử cũng tồn tại.

Lưu trữ giá trị

Các lệnh SQL INSERT và UPDATE đặt kết quả của các biểu thức vào trong một bảng. Các biểu thức trong lệnh đó phải được khớp với, và có lẽ phải được biến đổi thành, các dạng cột đích.

UNION, CASE và các cấu trúc có liên quan

Vì tất cả các kết quả truy vấn từ một lệnh SELECT phải xuất hiện trong một tập hợp duy nhất các cột, nên các dạng kết quả của từng mệnh để SELECT phải được khớp và được biến đổi thành một tập hợp đồng nhất. Tương tự, các biểu thức kết quả của cấu trúc CASE phải được biến đổi thành một dạng chung sao cho biểu thức CASE như một tổng thể có một dạng đầu ra được biết. Y hệt như vậy cho các cấu trúc ARRAY, và cho các hàm GREATEST và LEAST.

Các catalog hệ thống lưu trữ thông tin về các biến đổi hoặc cast nào, tồn tại giữa các dạng dữ liệu nào, và làm thế nào để thực hiện các biến đổi đó. Các cast bổ sung có thể được người sử dụng với lệnh CREATE CAST đưa thêm vào. (Điều này thường được thực hiện trong sự kết hợp với việc định nghĩa các dạng dữ liệu mới. Tập hợp các cast giữa các dạng được xây dựng sẵn từng được làm cẩn thận và tốt nhất là không tùy chỉnh).

Một kinh nghiệm giải quyết vấn đề bổ sung được trình phân tích cung cấp cho phép định nghĩa được cải thiện đối với hành vi đưa ra phù hợp trong các nhóm các dạng có các cast ngầm định. Các dạng dữ liệu được chia thành vài chủng loại dạng cơ bản, bao gồm numeric, string, bitstring, datetime, timespan, geometric, network và do người sử dụng định nghĩa. (danh sách có trong Bảng 45-45; nhưng lưu ý là cũng có khả năng tạo các chủng loại dạng tùy biến). Trong từng chủng loại có thể có 1 hoặc nhiều hơn các dạng được ưu tiên, chúng được ưu tiên khi có một sự lựa chọn các dạng có khả năng. Với sự lựa chọn cẩn thận các dạng được ưu tiên và các cast ngầm định có sẵn, có khả năng để đảm bảo rằng các biểu thức mù mờ (các biểu thức với nhiều giải pháp phân tích tùy chọn) có thể được giải quyết theo một cách thức hữu dụng.

Tất cả các qui tắc biến đổi dạng được thiết kế với vài nguyên tắc trong đầu:

- Các biến đổi ngầm định nên không bao giờ có các đầu ra gây ngạc nhiên hoặc không thể đoán trước được.
- Nên không có chi phí thêm trong trình phân tích hoặc trình thực thi nếu một truy vấn không cần biến đổi dạng ngầm định. Đó là, nếu một truy vấn được hình thành tốt và các dạng khớp được rồi, thì truy vấn đó nên thực thi mà không mất thêm thời gian trong trình phân tích và không đưa ra các lời gọi biến đổi ngầm định không cần thiết trong truy vấn đó.

Hơn nữa, nếu một truy vấn thường yêu cầu một biến đổi ngầm định cho một hàm, và nếu sau đó người sử dụng định nghĩa một hàm mới với các dạng đối số đúng, thì trình phân tích nên sử dụng hàm mới và không còn thực hiện biến đổi ngầm định để sử dụng hàm cũ nữa.

10.2. Toán tử

Toán tử đặc biệt được một biểu thức toán tử tham chiếu tới được xác định bằng việc sử dụng thủ tục sau. Lưu ý rằng thủ tục này gián tiếp được sự ưu tiên trước của các toán tử có liên quan tác động, vì điều đó sẽ xác định các biểu thức con nào được lấy làm các đầu vào của các toán tử nào. Xem Phần 4.1.6 để có thêm thông tin.

Quyết định dạng toán tử

1. Chọn các toán tử sẽ được xem xét từ catalog hệ thống pg_operator. Nếu một tên toán tử không đủ điều kiện về sơ đồ từng được sử dụng (trường hợp thông thường), thì các toán tử được xem xét là những toán tử với tên trùng khớp và tính toán đối số mà nhìn thấy được trong đường tìm kiếm hiện hành (xem Phần 5.7.3). Nếu một tên toán tử đủ điều kiện đã được đưa ra, thì chỉ các toán tử nào trong sơ đồ được chỉ định sẽ được xem xét.

   a) Nếu đường tìm kiếm thấy nhiều toán tử với các dạng đối số hệt nhau, thì chỉ toán tử nào xuất hiện sớm nhất trong đường đó được xem xét. Các toán tử với các dạng đối số khác nhau sẽ được xem xét ngang bằng nhau bất kể vị trí của đường tìm kiếm.

2. Kiểm tra một toán tử chấp nhận chính xác các dạng đối số đầu vào. Nếu có một toán tử tồn tại (có thể chỉ có một toán tử khớp chính xác trong tập hợp các toán tử được xem xét), hãy sử dụng nó.

   a) Nếu một đối số của một lời gọi toán tử nhị phân là dạng còn chưa được biết unknown, thì giả thiết nó là dạng y hệt như đối số khác cho kiểm tra này. Các lời gọi có liên quan tới 2 đầu vào unknown, hoặc một toán tử duy nhất với một đầu vào unknown, sẽ không bao giờ thấy một sự trùng khớp ở bước này.

3. Tìm kiếm sự trùng khớp tốt nhất

   a) Bỏ các toán tử ứng viên mà đối với chúng các dạng đầu vào không khớp và không thể biến đổi được (bằng việc sử dụng một biến đổi ngầm định) để khớp. Các hằng unknown sẽ được giả thiết là có khả năng biến đổi được sang bất kỳ điều gì vì mục đích này. Nếu chỉ có một ứng viên, thì hãy sử dụng nó; nếu khác thì hãy tiếp tục sang bước tiếp sau.

   b) Chạy qua tất cả các ứng viên và giữ lại những ứng viên nào với sự trùng khớp chính xác nhất ở các dạng đầu vào. (Các miền được xem xét y hệt như dạng cơ bản của chúng về mục đích này). Hãy giữ lại tất cả các ứng viên nếu không có các trùng khớp chính xác. Nếu chỉ một ứng viên còn lại, hãy sử dụng nó; nếu khác, hãy tiếp tục các bước tiếp sau.

   c) Chạy qua tất cả các ứng viên và giữ lại các ứng viên mà chấp nhận các dạng được ưu tiên (của chủng loại dạng các dạng dữ liệu đầu vào) trong hầu hết các vị trí nơi mà biến

đổi dạng sẽ được yêu cầu. Hãy giữ lại tất cả các ứng viên nếu không ứng viên nào chấp nhận các dạng được ưu tiên. Nếu chỉ còn lại một ứng viên, hãy sử dụng nó; nếu khác thì hãy tiếp tục bước tiếp sau.

d) Nếu bất kỳ đối số đầu vào nào là unknown, hãy kiểm tra các chủng loại dạng được chấp nhận ở các vị trí đối số đó với các ứng viên còn lại. Tại từng vị trí, hãy chọn chủng loại string nếu bất kỳ ứng viên nào chấp nhận chủng loại đó. (Khuynh hướng hướng tới chuỗi này là phù hợp vì một hằng dạng không biết trông giống như một chuỗi). Nếu không, nếu tất cả các ứng viên còn lại chấp nhận chủng loại dạng y hệt, thì hãy chọn chủng loại đó; nếu không sẽ hỏng vì sự lựa chọn đúng không thể được suy luận mà không có nhiều hơn các manh mối. Bây giờ hãy loại bỏ các ứng viên mà không chấp nhận chủng loại dạng được lựa chọn. Hơn nữa, nếu bất kỳ ứng viên nào chấp nhận dạng được ưu tiên theo chủng loại đó, thì hãy bỏ các ứng viên mà chấp nhận các dạng không được ưu tiên đối với đối số đó.

e) Nếu chỉ một ứng viên còn lại, hãy sử dụng nó. Nếu không ứng viên nào hoặc hơn 1 ứng viên còn lại, thì hỏng.

Một số ví dụ sau.

Ví dụ 10-1. Quyết định dạng toán tử yếu tố

Chỉ có một toán tử yếu tố (postfix !) được xác định trong catalog tiêu chuẩn, và nó lấy một đối số dạng bigint. Trình quét chỉ định một dạng ban đầu integer cho đối số đó trong biểu thức truy vấn này:

```
SELECT 40 ! AS "40 factorial";

40 factorial
----------------------------------------------
815915283247897734345611269596115894272000000000
(1 row)
```

Vì thế trình phân tích làm một biến đổi dạng trong toán hạng và truy vấn là tương đương với:

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

Ví dụ 10-2. Quyết định dạng toán tử ghép nối chuỗi

Cú pháp giống chuỗi được sử dụng để làm việc với các dạng chuỗi và làm việc với các dạng mở rộng phức tạp. Các chuỗi với dạng không được chỉ định sẽ khớp với các ứng viên toán tử có khả năng. Một ví dụ với một đối số không được chỉ định:

```
SELECT text 'abc' || 'def' AS "text and unknown";

text and unknown
------------------
abcdef
(1 row)
```

Trong trường hợp này trình phân tích tìm xem liệu có một toán tử lấy text cho cả 2 đối số hay không. Vì là có, nên nó giả thiết rằng đối số thứ 2 sẽ được diễn giải như dạng text.

Đây là một sự ghép nối trong các dạng không được chỉ định:

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

Trong trường hợp này không có mẹo ban đầu nào cho dạng nào để sử dụng, vì không dạng nào được chỉ định trong truy vấn. Vì thế, trình phân tích tìm tất cả các toán tử ứng viên và thấy rằng có các ứng viên chấp nhận cả các đầu vào chủng loại chuỗi và chủng loại chuỗi bit. Vì chủng loại chuỗi được ưu tiên khi sẵn sàng, chủng loại đó được chọn, và sau đó dạng được ưu tiên cho các chuỗi, text, được sử dụng như dạng đặc biệt để giải quyết các hằng không được rõ.

Ví dụ 10-3. Quyết định dạng toán tử phủ định và giá trị tuyệt đối

Catalog các toán tử của PostgreSQL có vài khoản cho toán tử tiền tố @, tất cả thứ đó triển khai các hoạt động giá trị tuyệt đối cho các dạng dữ liệu số khác nhau. Một trong những khoản đó là cho float8, nó là dạng được ưu tiên trong chủng loại số. Vì thế, PostgreSQL sẽ sử dụng khoản đó khi đối mặt với một đầu vào không rõ:

```
SELECT @ '-4.5' AS "abs";

 abs
-----
 4.5
(1 row)
```

Ở đây hệ thống đã giải quyết một cách ngầm định hằng dạng không rõ như dạng float8 trước khi áp dụng toán tử được chọn. Chúng ta có thể kiểm tra hợp lệ rằng float8 và một số dạng không phải khác từng được sử dụng:

```
SELECT @ '-4.5e500' AS "abs";
ERROR: "-4.5e500" is out of range for type double precision
```

Mặt khác, toán tử tiền tố ~ (phủ định dạng bitwise) chỉ được xác định cho các dạng dữ liệu số nguyên, không cho float8. Vì thế, nếu chúng ta thử một trường hợp tương tự với ~, thì ta có:

```
SELECT ~ '20' AS "negation";
ERROR: operator is not unique: ~ "unknown"
```

Mẹo: Không thể chọn một toán tử ứng viên tốt nhất. Bạn có thể cần thêm các cast dạng ngầm định.

Điều này xảy ra vì hệ thống không thể quyết định khả năng nào trong vài khả năng các toán tử ~ sẽ được ưu tiên. Chúng ta có thể giúp nó đưa ra với một cast ngầm định:

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
 -21
(1 row)
```

10.3. Hàm

Hàm đặc biệt được một lời gọi hàm tham chiếu tới được xác định bằng việc sử dụng thủ tục sau.

Quyết định dạng hàm

1. Chọn các hàm sẽ được xem xét từ catalog hệ thống pg_proc. Nếu một tên hàm đủ điều kiện không theo sơ đồ từng được sử dụng, thì các hàm được xem xét đó là các hàm với tên trùng khớp và tính toán đối số là trực quan trong đường tìm kiếm hiện hành (xem Phần 5.7.3). Nếu một tên hàm đủ điều kiện từng được đưa ra, thì chỉ các hàm trong sơ đồ được chỉ định sẽ được xem xét.

   a) Nếu đường tìm kiếm thấy nhiều hàm các dạng đối số y hệt nhau, thì chỉ hàm nào xuất hiện trước nhất trong đường đó sẽ được xem xét. Các hàm các dạng đối số khác nhau sẽ được xem xét trong cơ sở ngang bằng nhau bất kể vị trí đường tìm kiếm.

   b) Nếu một hàm được khai báo với một tham số mảng VARIADIC, và lời gọi không sử dụng từ khóa VARIADIC, thì hàm đó được đối xử như thể tham số mảng đó đã bị thay thế bằng một hoặc nhiều trường hợp dạng phần tử của nó, như cần thiết để khớp với lời gọi. Sau sự mở rộng như vậy thì hàm đó có thể có các dạng đối số hiệu quả ngang bằng với một số hàm không nhiều đối số (nonvariadic). Trong trường hợp đó hàm xuất hiện sớm hơn trong đường tìm kiếm sẽ được sử dụng, hoặc nếu 2 hàm đó là trong cùng y hệt sơ đồ, thì hàm không nhiều đối số sẽ được ưu tiên.

   c) Các hàm mà có các giá trị mặc định đối với các tham số sẽ được xem xét để khớp với bất kỳ lời gọi nào mà làm mờ số 0 hoặc nhiều vị trí tham số có khả năng mặc định hơn. Nếu hơn một hàm như vậy khớp với một lời gọi, thì một hàm xuất hiện sớm nhất trong đường tìm kiếm sẽ được sử dụng. Nếu có 2 hoặc nhiều hơn các hàm như vậy trong sơ đồ y hệt với các dạng tham số giống hệt nhau trong các vị trí không phải là mặc định (có khả năng nếu chúng có các tập hợp khác nhau các tham số không có khả năng mặc định), thì hệ thống sẽ không có khả năng xác định hàm nào sẽ ưu tiên, và vì thế một lỗi "lời gọi hàm mù mờ" sẽ xảy ra nếu không có sự trùng khớp nào tốt hơn cho lời gọi đó có thể được tìm thấy.

2. Hãy kiểm tra một hàm chấp nhận chính xác các dạng đối số đầu vào đó. Nếu một hàm đang tồn tại (có thể chỉ là một sự trùng khớp chính xác trong tập hợp các hàm được xem xét), thì hãy sử dụng nó. (Các trường hợp có liên quan tới unknown sẽ không bao giờ thấy một sự trùng khớp ở bước này).

3. Nếu không sự trùng khớp chính xác nào được tìm thấy, hãy xem liệu lời gọi hàm đó có xuất hiện để là một yêu cầu biến đổi dạng đặc biệt hay không. Điều này xảy ra nếu lời gọi hàm đó chỉ có một đối số và tên hàm đó là y hệt như tên (nội bộ) của một số dạng dữ liệu. Hơn nữa, đối số hàm phải hoặc là một hằng dạng không được biết, hoặc một dạng mà có thể ép buộc nhị phân tới dạng dữ liệu được đặt tên, hoặc một dạng mà có thể được biến đổi thành dạng dữ liệu được đặt tên bằng việc áp dụng các hàm I/O dạng đó (đó là, biến đổi hoặc là tới hoặc từ một trong những dạng chuỗi tiêu chuẩn). Khi các điều kiện đó được thỏa mãn, thì

lời gọi hàm đó được đối xử như một dạng đặc tả CAST[1].

4. Tìm kiếm sự trùng khớp tốt nhất.

   a) Hãy loại bỏ các hàm ứng viên theo đó các dạng đầu vào không trùng khớp và không thể được biến đổi (bằng việc sử dụng một biến đổi ngầm) để trùng khớp. Các hằng unknown sẽ được giả thiết có khả năng biến đổi được thành bất kỳ điều gì cho mục đích này. Nếu chỉ một ứng viên còn lại, hãy sử dụng nó; nếu không hãy tiếp tục bước tiếp sau.

   b) Hãy chạy qua tất cả các ứng viên và giữ các ứng viên nào có các trùng khớp chính xác nhất trong các dạng đầu vào. (Các miền sẽ được xem xét y hệt như dạng cơ bản của chúng cho mục đích này). Hãy giữ tất cả các ứng viên nếu không ứng viên nào có các trùng khớp chính xác. Nếu chỉ một ứng viên còn lại, hãy sử dụng nó; nếu không hãy tiếp tục bước tiếp sau.

   c) Hãy chạy qua tất cả các ứng viên và giữ lại các ứng viên nào mà chấp nhận các dạng được ưu tiên (đối với chủng loại dạng các dạng dữ liệu đầu vào) ở hầu hết các vị trí nơi mà sự biến đổi dạng sẽ được yêu cầu. Hãy giữ tất cả các ứng viên nếu không ứng viên nào chấp nhận các dạng được ưu tiên. Nếu chỉ một ứng viên còn lại, hãy sử dụng nó; nếu không, hãy tiếp tục bước tiếp sau.

   d) Nếu bất kỳ đối số đầu vào nào là unknown, hãy kiểm tra các chủng loại dạng được chấp nhận ở các vị trí đối số đó bằng việc giữ lại các ứng viên. Tại từng vị trí, hãy chọn chủng loại chuỗi string nếu bất kỳ ứng viên nào chấp nhận chủng loại đó. (Khuynh hướng hướng tới chuỗi này là phù hợp vì một hằng dạng không biết trông giống như một chuỗi). Nếu không, nếu tất cả các ứng viên còn lại chấp nhận chủng loại dạng y hệt, thì hãy chọn chủng loại đó; nếu không sẽ hỏng vì sự lựa chọn đúng không thể được suy ra mà không có manh mối nào hơn. Bây giờ hãy loại bỏ các ứng viên mà không chấp nhận chủng loại dạng được chọn. Hơn nữa, nếu bất kỳ ứng viên nào chấp nhận một dạng được ưu tiên trong chủng loại đó, hãy loại bỏ các ứng viên mà chấp nhận các dạng không được ưu tiên đối với đối số đó.

   e) Nếu chỉ một ứng viên còn lại, hãy sử dụng nó. Nếu không ứng viên nào hoặc hơn một ứng viên còn lại, thì hỏng.

   Lưu ý rằng các qui tắc "trùng khớp tốt nhất" là y như nhau đối với quyết định dạng hàm và toán tử. Một số ví dụ bên dưới.

Ví dụ 10-4. Quyết định dạng đối số hàm số làm tròn

Chỉ có một hàm round có 2 đối số; nó lấy đối số đầu dạng số numeric và đối số thứ 2 dạng số nguyên integer. Vì thế truy vấn sau sẽ tự động biến đổi đối số đầu từ dạng integer thành numeric:

```
SELECT round(4, 4);
```

1 Lý do cho bước này là để hỗ trợ các đặc tả cast dạng hàm trong các trường hợp nơi mà không có một hàm cast thực sự. Nếu có một hàm cast, thì nó được đặt tên theo qui định sau dạng đầu ra của nó, và vì thế sẽ không có nhu cầu phải có một trường hợp đặc biệt. Xem CREATE CAST để có chú giải bổ sung.

```
round
--------
4.0000
(1 row)
```

Truy vấn đó thực sự được trình phân tích biến thành:

```
SELECT round(CAST (4 AS numeric), 4);
```

Vì các hằng số dạng số với các dấu thập phân ban đầu được chỉ định cho dạng numeric, nên truy vấn sau sẽ không đòi hỏi sự biến đổi dạng và vì thế có thể hiệu lực hơn một chút:

```
SELECT round(4.0, 4);
```

Ví dụ 10-5. Quyết định dạng hàm chuỗi con

Có vài hàm substr, một trong số chúng lấy các dạng text và integer. Nếu được gọi bằng một hằng chuỗi của dạng không được chỉ định, thì hệ thống sẽ chọn hàm ứng viên mà chấp nhận một đối số của chủng loại string được ưu tiên (ấy là dạng text).

```
SELECT substr('1234', 3);

substr
--------
34
(1 row)
```

Nếu chuỗi đó được khai báo sẽ là ở dạng varchar, như có thể là trong trường hợp nếu nó tới từ một bảng, thì trình phân tích sẽ cố gắng biến đổi nó để trở thành text:

```
SELECT substr(varchar '1234', 3);

substr
--------
34
(1 row)
```

Điều này được trình phân tích biến đổi để trở nên có hiệu lực:

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **Lưu ý**: Trình phân tích học từ catalog pg_cast catalog rằng text và varchar là tương thích nhị phân, nghĩa là một thứ có thể được truyền tới một hàm mà chấp nhận thứ khác mà không làm bất kỳ biến đổi vật lý nào. Vì thế, không lời gọi biến đổi dạng nào thực sự được chèn vào trong trường hợp này.

Và, nếu hàm đó được gọi với đối số dạng integer, thì trình phân tích sẽ cố biến đổi nó thành text:

```
SELECT substr(1234, 3);
ERROR: hàm substr(integer, integer) không tồn tại
Gợi ý: Không hàm nào khớp với các dạng đối số và tên được đưa ra. Bạn có lẽ cần bổ sung thêm các cast dạng rõ ràng.
```

Điều này không làm việc vì integer không có một cast ngầm cho text. Một cast rõ ràng sẽ làm việc, tuy nhiên:

```
SELECT substr(CAST (1234 AS text), 3);
substr
--------
34
(1 row)
```

10.4. Lưu giá trị

Các giá trị được chèn vào trong một bảng sẽ được biến đổi sang dạng dữ liệu của các cột đích theo các bước sau.

Biến đổi dạng lưu giá trị

1. Kiểm tra một sự trùng khớp chính xác với đích.
2. Nếu không, cố gắng biến đổi biểu thức đó thành dạng đích. Điều này sẽ thành công nếu có một cast được đăng ký giữa 2 dạng đó. Nếu biểu thức là một hằng dạng không được biết, thì các nội dung của chuỗi hằng đó sẽ được đưa vào thủ tục biến đổi đầu vào đối với dạng đích.
3. Kiểm tra xem nếu có một cast kích cỡ cho dạng đích. Một cast kích cỡ là một cast từ dạng đó tới bản thân nó. Nếu một cast kích cỡ được thấy trong catalog pg_cast, hãy áp dụng nó cho biểu thức đó trước khi lưu vào cột đích. Hàm triển khai cho một cast như vậy luôn lấy một tham số dư thừa dạng integer, nó nhận giá trị atttypmod của cột đích (thường là độ dài được khai báo của nó, dù sự diễn giải atttypmod là khác nhau đối với các dạng dữ liệu khác nhau), và nó có thể lấy một tham số boolean mà nói liệu cast đó có là rõ ràng hay ẩn. Hàm cast đó có trách nhiệm về việc áp dụng bất kỳ ngữ nghĩa phụ thuộc độ dài nào như việc kiểm tra kích cỡ hoặc cắt ngắn bớt.

Ví dụ 10-6. Biến đổi dạng lưu character

Đối với một cột đích được khai báo như là character(20) thì lệnh sau đây chỉ ra rằng giá trị được lưu có kích cỡ đúng:

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

v | octet_length
----------------------+--------------
abcdef | 20
(1 row)
```

Những gì thực sự xảy ra ở đây là 2 hằng không được rõ được giải quyết thành text một cách mặc định, cho phép toán tử || sẽ được giải quyết như là sự ghép nối text. Sau đó kết quả text của toán tử đó được biến đổi thành bpchar ("ký tự trống được nối vào", tên nội bộ của dạng dữ liệu character) sẽ khớp với dạng cột đích. (Vì sự biến đổi từ text sang bpchar là ép nhị phân, nên biến đổi này không chèn bất kỳ lời gọi hàm thực sự nào). Cuối cùng, hàm kích cỡ bpchar(bpchar, integer, boolean) được thấy trong catalog hệ thống và được áp dụng cho kết quả của toán tử đó và độ dài cột được lưu trữ. Hàm có dạng đặc thù này thực hiện kiểm tra độ dài được yêu cầu và sự bổ sung các khoảng trống nối thêm vào.

10.5. UNION, CASE và các cấu trúc có liên quan

Các cấu trúc SQL UNION phải khớp có lẽ với các dạng không tương tự để trở thành một tập kết quả duy nhất. Quyết định thuật toán được áp dụng một cách tách biệt đối với từng cột đầu ra của một

truy vấn thống nhất. Các cấu trúc INTERSECT và EXCEPT giải quyết các dạng không tương tự theo cách y hệt như UNION. Các cấu trúc CASE, ARRAY, VALUES, GREATEST và LEAST sử dụng thuật toán hệt như nhau để khớp các biểu thức thành phần của chúng và chọn một dạng dữ liệu kết quả.

Quyết định dạng cho UNION, CASE và các cấu trúc có liên quan

1. Nếu tất cả các đầu vào là cùng một dạng như nhau, và nó không phải là unknown, hãy giải quyết như dạng đó. Nếu không, hãy thay thế bất kỳ các dạng miền nào trong danh sách với các dạng cơ bản bên dưới của chúng.
2. Nếu tất cả các đầu vào là dạng unknown, hãy giải quyết như dạng text (dạng được ưu tiên của chủng loại chuỗi). Nếu không các đầu vào unknown sẽ bị bỏ qua.
3. Nếu các đầu vào không phải tất cả cùng chủng loại dạng y hệt nhau, thì hỏng.
4. Hãy chọn dạng đầu vào không phải là không rõ mà là một dạng được ưu tiên trong chủng loại đó, nếu có một dạng như vậy.
5. Nếu không, hãy chọn dạng đầu vào cuối cùng không phải là không rõ mà cho phép tất cả các đầu vào không phải không rõ đứng đằng trước sẽ được biến đổi một cách ngầm sang nó. (Luôn có một dạng như vậy, vì ít nhất dạng đầu tiên trong danh sách phải thỏa mãn điều kiện này).
6. Hãy biến đổi tất cả các đầu vào thành dạng được lựa chọn. Hỏng nếu không có một biến đổi nào từ một đầu vào được đưa ra sang dạng được chọn.

Một số ví dụ sau.

Ví dụ 10-7. Quyết định dạng với các dạng theo qui định trong một liên đoàn (Union)

```
SELECT text ’a’ AS "text" UNION SELECT ’b’;

text
------
a
b
(2 rows)
```

Ở đây, hằng dạng không rõ 'b' sẽ được giải quyết thành dạng text.

Ví dụ 10-8. Quyết định dạng trong một liên đoàn đơn giản

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

numeric
---------
1
1.2
(2 rows)
```

Hằng 1.2 là dạng numeric, và giá trị 1 integer có thể là cast ẩn đối với numeric, nên dạng đó được sử dụng.

Ví dụ 10-9. Quyết định dạng trong một liên đoàn được chuyển tải

```
SELECT 1 AS "real" UNION SELECT CAST(’2.2’ AS REAL);
```

```
real
------
1
2.2
(2 rows)
```

Ở đây, vì dạng real không thể là một cast ẩn cho integer, nhưng integer có thể là cast ẩn cho real, nên dạng kết quả liên đoàn được giải quyết như là real.

Chương 11. Các chỉ số

Các chỉ số là một cách phổ biến để cải thiện hiệu năng của cơ sở dữ liệu. Một chỉ số cho phép máy chủ cơ sở dữ liệu tìm và truy xuất các hàng đặc biệt nhanh hơn nhiều so với nó có thể làm mà không có một chỉ số. Nhưng các chỉ số cũng bổ sung thêm chi phí tổng vào hệ thống cơ sở dữ liệu như một tổng thể, vì thế chúng nên được sử dụng một cách hợp lý.

11.1. Giới thiệu

Giả thiết chúng ta có một bảng tương tự với điều này:

```
CREATE TABLE test1 (
        id integer,
        content varchar
);
```

và ứng dụng đưa ra nhiều truy vấn có dạng:

```
SELECT content FROM test1 WHERE id = constant;
```

Không có chuẩn bị trước, hệ thống có thể phải quét toàn bộ bảng test1, từng hàng một, để tìm tất cả các khoản trùng khớp. Nếu có nhiều hàng trong bảng test1 và chỉ một ít hàng (có thể là 0 hoặc 1) mà có thể được trả về bằng một truy vấn như vậy, thì điều này rõ ràng là một phương pháp không hiệu quả. Nhưng nếu hệ thống từng được ra lệnh phải duy trì một chỉ số trong cột id, thì nó có thể sử dụng một phương pháp hiệu quả hơn cho việc định vị các hàng trùng khớp. Ví dụ, nó có thể chỉ phải đi qua một ít mức sâu trong một cây tìm kiếm.

Một tiếp cận tương tự được sử dụng trong hầu hết các sách không viễn tưởng: các khái niệm và các khoản mà thường xuyên được các độc giả tra cứu sẽ được thu thập trong một chỉ số theo vần abc ở cuối của cuốn sách. Độc giả có quan tâm có thể quét chỉ số đó khá nhanh và lật sang (các) trang tương ứng, thay vì phải đọc toàn bộ cuốn sách để tìm ra tư liệu quan tâm. Đó chính là tác vụ của tác giả để cho biết trước các khoản mà các độc giả có khả năng sẽ tra cứu, đó chính là tác vụ của người lập trình cơ sở dữ liệu để thấy trước được các chỉ số nào sẽ là hữu dụng.

Lệnh sau đây có thể được sử dụng để tạo ra một chỉ số trong cột id, như được thảo luận:

```
CREATE INDEX test1_id_index ON test1 (id);
```

Tên test1_id_index có thể được chọn tự do, nhưng bạn nên chọn thứ gì đó mà cho phép bạn nhớ sau này chỉ số đó là về cái gì.

Để loại bỏ một chỉ số, hãy sử dụng lệnh DROP INDEX. Các chỉ số có thể được thêm vào và bị loại bỏ từ các bảng bất kỳ lúc nào.

Một khi một chỉ số được tạo ra, không sự can thiệp tiếp sau nào được yêu cầu: hệ thống sẽ cập nhật chỉ số đó khi bảng được sửa đổi, và nó sẽ sử dụng chỉ số đó trong các truy vấn khi nó nghĩ làm như vậy có thể có hiệu quả hơn so với một sự quét bảng tuần tự. Nhưng bạn có thể phải chạy lệnh ANALYZE thường xuyên để cập nhật các thống kê để cho phép trình hoạch định truy vấn đó tạo ra các

quyết định được cân nhắc kỹ. Xem Chương 14 để có thông tin về cách tìm ra liệu một chỉ số có được sử dụng và khi nào và vì sao trình hoạch định có thể chọn không sử dụng một chỉ số.

Các chỉ số cũng có thể có lợi cho các lệnh UPDATE và DELETE với các điều kiện tìm kiếm. Các chỉ số có thể được sử dụng nhiều hơn trong các tìm kiếm chung. Vì thế, một chỉ số được định nghĩa trong một cột mà là một phần của một điều kiện chung cũng có thể làm nhanh đáng kể các truy vấn với các việc chung đó.

Việc tạo ra một chỉ số trong một bảng lớn có thể mất thời gian. Mặc định, PostgreSQL cho phép đọc (các lệnh SELECT) sẽ xảy ra trong bảng song song với việc tạo chỉ số, nhưng ghi (INSERT, UPDATE, DELETE) bị khóa cho tới khi sự xây chỉ số đó hoàn tất. Trong các môi trường sản xuất thì điều này thường không chấp nhận được. Có khả năng để cho phép ghi xảy ra song song với sự tạo chỉ số, nhưng có vài cạm bẫy phải nhận thức được - để có thêm thông tin, hãy xem *Xây dựng các chỉ số cùng đồng thời*.

Sau khi một chỉ số được tạo ra, hệ thống phải giữ nó được đồng bộ với bảng đó. Điều này bổ sung thêm chi phí tổng cho các hoạt động điều khiển dữ liệu. Vì thế các chỉ số mà ít khi hoặc không bao giờ được sử dụng trong các truy vấn nên bị loại bỏ.

11.2. Các dạng chỉ số

PostgreSQL đưa ra vài dạng chỉ số: B-tree, Hash, GiST và GIN. Từng dạng chỉ số sử dụng một thuật toán khác nhau phù hợp nhất cho các dạng truy vấn khác nhau. Mặc định, lệnh CREATE INDEX sẽ tạo ra các chỉ số B-tree, nó phù hợp với hầu hết các tình huống phổ biến.

B-tree có thể điều khiển các truy vấn ngang bằng nhau và theo dãy trong các dữ liệu có thể được lưu giữ theo một số trật tự. Đặc biệt, trình hoạch định truy vấn của PostgreSQL sẽ xem xét sử dụng một chỉ số B-tree bất kỳ khi nào một cột chỉ số có liên quan trong một so sánh bằng việc sử dụng một trong các toán tử:

```
<
<=
=
>=
>
```

Các cấu trúc tương đương với các kết hợp của các toán tử đó, như BETWEEN và IN, có thể cũng được triển khai với tìm kiếm chỉ số B-tree. Hơn nữa, một điều kiện IS NULL hoặc IS NOT NULL trong một cột chỉ số có thể được sử dụng với một chỉ số B-tree.

Trình tối ưu hóa cũng có thể sử dụng một chỉ số B-tree cho các truy vấn có liên quan tới mẫu khớp với các toán tử LIKE và ~ nếu mẫu đó là một hằng số và nằm ở đầu của chuỗi đó - ví dụ, col LIKE 'foo %' hoặc col ~ '^foo', nhưng không phải là col LIKE '%bar'. Tuy nhiên, nếu cơ sở dữ liệu của bạn không sử dụng miền địa phương C thì bạn sẽ cần phải tạo chỉ số bằng một lớp toán tử đặc biệt để hỗ trợ việc đánh chỉ số của các truy vấn khớp mẫu; xem Phần 11.9 bên dưới. Cũng có khả năng sử dụng các chỉ số B-tree cho ILIKE và ~* nhưng chỉ nếu mẫu đó bắt đầu với các ký tự không phải abc, nghĩa là các ký tự không bị ảnh hưởng vì biến đổi chữ hoa/chữ thường.

Các chỉ số Hash cũng có thể điều khiển các so sánh ngang bằng nhau đơn giản. Trình hoạch định truy vấn sẽ xem xét sử dụng một chỉ số hash bất kỳ khi nào một cột được đánh chỉ số có liên quan trong một so sánh bằng việc sử dụng toán tử =. Lệnh sau đây được sử dụng để tạo một chỉ số hash:

```
CREATE INDEX name ON table USING hash (column);
```

Chú ý

Các hoạt động của chỉ số hash hiện không được ghi lưu ký WAL, nên các chỉ số hash có thể cần phải được xây dựng lại với REINDEX sau khi một cơ sở dữ liệu hỏng nếu từng có những thay đổi không được ghi. Hơn nữa, các thay đổi đối với các chỉ số hash sẽ không được nhân bản qua sự nhân bản dựa vào tệp hoặc nắn dòng sau sao lưu cơ bản ban đầu, vì thế chúng đưa ra các câu trả lời sai cho các truy vấn mà sau đó sử dụng chúng. Vì các lý do đó, sử dụng chỉ số hash hiện không được khuyến khích.

Các chỉ số GiST không phải là một dạng chỉ số duy nhất, mà là một hạ tầng trong đó nhiều chiến lược đánh chỉ số có thể được triển khai. Một cách tương xứng, các toán tử đặc biệt với chúng một chỉ số GiST có thể được sử dụng là khác nhau, phụ thuộc vào chiến lược đánh chỉ số (lớp toán tử). Như một ví dụ, phân phối tiêu chuẩn của PostgreSQL bao gồm các lớp toán tử GiST cho vài dạng dữ liệu địa lý 2 chiều, chúng hỗ trợ các truy vấn được đánh chỉ số bằng việc sử dụng các toán tử đó:

```
<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&
```

(Xem Phần 9.11 về ý nghĩa của các toán tử đó). Nhiều lớp toán tử khác của GiST là sẵn sàng trong bộ sưu tập contrib hoặc như các dự án riêng rẽ. Để có thêm thông tin, xem Chương 52.

Các chỉ số GIN là các chỉ số nghịch đảo, chúng có thể điều khiển các giá trị chứa hơn một khóa, các mảng, ví dụ thế. Như một ví dụ, phân phối PostgreSQL tiêu chuẩn bao gồm các lớp toán tử GIN cho các mảng 1 chiều, chúng hỗ trợ các truy vấn được đánh chỉ số bằng việc sử dụng các toán tử đó:

```
<@
@>
=
&&
```

(Xem Phần 9.17 về ý nghĩa của các toán tử đó). Nhiều lớp toán tử khác của GIN là sẵn sàng trong bộ sưu tập contrib hoặc như các dự án riêng biệt. Để có thêm thông tin, xem Chương 53.

11.3. Các chỉ số nhiều cột

Một chỉ số có thể được định nghĩa trong hơn 1 cột của 1 bảng. Ví dụ, nếu bạn có 1 bảng dạng này:

```
CREATE TABLE test2 (
        major int,
        minor int,
        name varchar
);
```

(nói, bạn giữ thư mục /dev của bạn trong một cơ sở dữ liệu...) và bạn thường xuyên đưa ra các truy vấn như:

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

sau đó có thể là phù hợp để định nghĩa một chỉ số trong các cột major và minor cùng nhau, nghĩa là:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

Hiện hành, chỉ các dạng chỉ số B-tree, GiST và GIN hỗ trợ các chỉ số nhiều cột. Tới 32 cột có thể được chỉ định. (Giới hạn này có thể được tùy chỉnh khi xây dựng PostgreSQL; xem tệp pg_config_manual.h).

Một chỉ số B-tree nhiều cột có thể được sử dụng với các điều kiện truy vấn mà liên quan tới bất kỳ tập con nào của các cột chỉ số, nhưng chỉ số đó là hiệu quả nhất khi có các ràng buộc trong các cột dẫn đầu (bên trái nhất). Qui tắc chính xác là các ràng buộc ngang nhau trong các cột dẫn dắt, cộng với bất kỳ ràng buộc không bằng nhau nào trong cột đầu mà không có một ràng buộc bằng nhau, sẽ được sử dụng để giới hạn phần của chỉ số được quét. Các ràng buộc trong các cột ở bên phải của các cột đó được kiểm tra trong chỉ số đó, sao cho chúng bảo vệ các cuộc viếng thăm tới bảng đó một cách đúng đắn, nhưng chúng không làm giảm phần của chỉ số mà phải được quét. Ví dụ, đưa ra một chỉ số trong (a, b, c) và một điều kiện truy vấn WHERE a = 5 AND b >= 42 AND c < 77, thì chỉ số đó có thể phải được quét từ khoản đầu với a = 5 và b = 42 cho qua tới khoản cuối với a = 5. Các khoản chỉ số với c > = 77 có thể bị bỏ qua, nhưng chúng có thể vẫn sẽ phải được quét qua. Chỉ số này có thể, về nguyên tắc, được sử dụng cho các truy vấn có các ràng buộc trong b và/hoặc c với không có ràng buộc trong a - nhưng toàn bộ chỉ số đó có thể phải được quét, nên trong hầu hết các trường hợp thì trình hoạch định có thể ưu tiên một sự quét bảng tuần tự hơn là bằng việc sử dụng chỉ số đó.

Một chỉ số GiST nhiều cột có thể được sử dụng với các điều kiện truy vấn mà có liên quan tới bất kỳ tập con nào của các cột chỉ số. Các điều kiện trong các cột bổ sung giới hạn các khoản đầu vào được chỉ số đó trả về, nhưng điều kiện trong cột đầu là quan trọng nhất cho việc xác định chỉ số đó cần phải được quét bao nhiêu. Một chỉ số GiST sẽ là khá không hiệu quả nếu cột đầu của nó chỉ có một ít các giá trị phân biệt, thậm chí nếu có nhiều giá trị phân biệt trong các cột bổ sung thêm.

Một chỉ số GIN nhiều cột có thể được sử dụng với các điều kiện truy vấn có liên quan tới bất kỳ tập con nào của các cột chỉ số. Không giống như B-tree hoặc GiST, tính hiệu quả tìm kiếm của chỉ số là y hệt bất chấp (các) cột chỉ số nào các điều kiện truy vấn sử dụng.

Tất nhiên, từng cột phải được sử dụng với các toán tử phù hợp cho dạng chỉ số đó; các mệnh đề có liên quan tới các toán tử khác sẽ không được xem xét.

Các chỉ số nhiều cột nên được sử dụng dè sẻn. Trong hầu hết các tình huống, một chỉ số trong một cột duy nhất là đủ và tiết kiệm không gian và thời gian. Các chỉ số với nhiều hơn 3 cột có lẽ không là hữu dụng trừ phi sự sử dụng bảng đó là cực kỳ cách điệu. Xem thêm Phần 11.5 về một số thảo luận các giá trị của các cấu hình chỉ số khác nhau.

11.4. Chỉ số và ORDER BY

Bổ sung thêm vào việc tìm kiếm đơn giản các hàng sẽ được trả về bằng một truy vấn, một chỉ số có thể có khả năng đưa chúng ra theo một trật tự được sắp xếp. Điều này cho phép một đặc tả ORDER BY của truy vấn đó sẽ được vinh danh mà không có một bước sắp xếp riêng rẽ. Đối với các dạng chỉ số hiện đang được PostgreSQL hỗ trợ, chỉ B-tree có thể cho kết quả đầu ra được sắp xếp - các dạng chỉ số khác trả về các hàng trùng khớp theo một trật tự không xác định, phụ thuộc vào sự triển khai.

Trình hoạch định sẽ xem xét làm thỏa mãn đặc tả ORDER BY hoặc bằng việc quét một chỉ số có sẵn mà khớp được đặc tả đó, hoặc bằng việc quét bảng theo trật tự vật lý và tiến hành sắp xếp rõ ràng. Đối với một truy vấn mà đòi hỏi việc quét một phần lớn của bảng, thì một sự sắp xếp rõ ràng có khả năng sẽ là nhanh hơn so với việc sử dụng một chỉ số vì nó đòi hỏi ít I/O của đĩa hơn vì tuân theo một mẫu truy cập tuần tự. Các chỉ số là hữu dụng hơn chỉ khi một ít hàng cần phải được lấy. Một trường hợp đặc biệt quan trọng là ORDER BY trong sự kết hợp với LIMIT n: một sự sắp xếp rõ ràng sẽ phải xử lý tất cả các dữ liệu để nhận diện n hàng đầu tiên, nhưng nếu có một chỉ số trùng khớp với ORDER BY, thì n hàng đầu tiên đó có thể được truy xuất trực tiếp, hoàn toàn không có việc quét phần còn lại.

Mặc định, các chỉ số B-tree lưu trữ các khoản đầu vào của chúng theo trật tự tăng dần với các null cuối cùng. Điều này có nghĩa là một sự quét tiến của một chỉ số trong cột x tạo ra đầu ra thỏa mãn ORDER BY x (hoặc một cách chi tiết, ORDER BY x ASC NULLS LAST). Chỉ số đó cũng có thể được quét ngược, tạo ra các kết quả đầu ra thỏa mãn ORDER BY x DESC (hoặc một cách chi tiết hơn, ORDER BY x DESC NULLS FIRST, vì NULLS FIRST là mặc định cho ORDER BY DESC).

Bạn có thể tinh chỉnh trật tự của một chỉ số B-tree bằng việc đưa vào các lựa chọn ASC, DESC, NULLS FIRST, và/hoặc NULLS LAST khi tạo chỉ số đó; ví dụ:

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

Một chỉ số được lưu giữ theo trật tự tăng dần với các null trước hết có thể làm thỏa mãn hoặc ORDER BY x ASC NULLS FIRST hoặc ORDER BY x DESC NULLS LAST phụ thuộc vào theo chiều nào nó được quét.

Bạn có thể hồ nghi vì sao chúng đưa ra tất cả 4 lựa chọn, khi mà 2 lựa chọn cùng nhau với khả năng quét ngược có thể bao trùm tất cả các phương án của ORDER BY. Trong các chỉ số 1 cột duy nhất thì các lựa chọn quả thực là dư thừa, nhưng trong các chỉ số nhiều cột thì chúng có thể là hữu dụng. Hãy xem xét một chỉ số 2 cột trong (x, y): điều này có thể làm thỏa mãn ORDER BY x, y nếu chúng ta quét tiến, hoặc ORDER BY x DESC, y DESC nếu chúng ta quét lùi. Nhưng nó có thể là ứng dụng thường xuyên cần phải sử dụng ORDER BY x ASC, y DESC. Không có cách nào để có được trật tự đó từ một

chỉ số thô, mà có khả năng nếu chỉ số đó được định nghĩa như (x ASC, y DESC) hoặc (x DESC, y ASC).

Rõ ràng, các chỉ số với các trật tự sắp xếp không mặc định là một chức năng khá chuyên dụng, nhưng đôi khi chúng có thể tạo ra sự tăng tốc nhanh to lớn cho các truy vấn nhất định. Liệu có đáng giá cho việc duy trì một chỉ số như vậy hay không, phụ thuộc vào việc bạn thường xuyên sử dụng thế nào các truy vấn đòi hỏi một trật tự sắp xếp đặc biệt.

11.5. Kết hợp nhiều chỉ số

Một sự quét chỉ số duy nhất chỉ có thể sử dụng các mệnh đề truy vấn mà sử dụng các cột chỉ số với các toán tử của lớp toán tử của nó được kết nối bằng AND. Ví dụ, đưa ra một chỉ số trên (a, b) thì một chỉ số như WHERE a = 5 AND b = 6 có thể sử dụng chỉ số đó, nhưng một truy vấn như WHERE a = 5 OR b = 6 có thể không trực tiếp sử dụng chỉ số đó.

May thay, PostgreSQL có khả năng kết hợp nhiều chỉ số (bao gồm nhiều sử dụng cùng y hệt chỉ số đó) để điều khiển các trường hợp mà không thể được triển khai bằng các sự quét chỉ số duy nhất. Hệ thống đó có thể tạo thành các điều kiện AND và OR khắp vài sự quét chỉ số. Ví dụ, một truy vấn giống như WHERE x = 42 OR x = 47 OR x = 53 OR x = 99 có thể là được chia thành 4 sự quét riêng biệt của một chỉ số trên x, mỗi sự quét sử dụng một trong các mệnh đề truy vấn đó. Các kết quả của các sự quét đó sau đó sẽ được hoặc (OR) cùng nhau để tạo ra kết quả đó. Ví dụ khác là nếu chúng ta có các chỉ số riêng rẽ trong x và y, thì một triển khai có khả năng của một truy vấn giống như WHERE x = 5 AND y = 6 là sử dụng từng chỉ số với mệnh đề truy vấn đúng phù hợp và sau đó và (AND) cùng nhau các kết quả chỉ số đó để nhận diện các hàng kết quả.

Để kết hợp nhiều chỉ số, hệ thống đó quét từng chỉ số cần thiết và chuẩn bị một bitmap trong bộ nhớ trao các vị trí các hàng của bảng mà sẽ được nêu như là khớp với các điều kiện của chỉ số đó. Các bitmap sau đó được và (AND) và được hoặc (OR) cùng với nhau như cần thiết bằng truy vấn đó. Cuối cùng, các hàng của bảng thực sự được viếng thăm và được trả về. Các hàng của bảng được viếng thăm theo trật tự vật lý, vì đó là cách mà bitmap đó được đưa ra; điều này có nghĩa là bất kỳ trật tự nào của các chỉ số gốc ban đầu là mất, và vì thế một bước sắp xếp riêng rẽ sẽ là cần thiết nếu truy vấn đó có một mệnh đề ORDER BY. Vì lý do này, và vì từng sự quét chỉ số bổ sung thêm thời gian dư thừa, mà trình hoạch định đôi khi sẽ chọn sử dụng một sự quét chỉ số đơn giản thậm chí dù các chỉ số bổ sung thêm là có sẵn mà có thể cũng từng được sử dụng.

Trong tất cả ngoại trừ các ứng dụng đơn giản nhất, có hàng loạt các sự kết hợp các chỉ số mà có thể là hữu dụng, và lập trình viên cơ sở dữ liệu phải lựa chọn để quyết định các chỉ số nào sẽ cung cấp. Đôi khi các chỉ số nhiều cột là tốt nhất, nhưng đôi khi sẽ là tốt hơn để tạo ra các chỉ số riêng rẽ và dựa vào chức năng kết hợp chỉ số. Ví dụ, nếu tải công việc của bạn bao gồm một sự pha trộn của các truy vấn mà đôi khi chỉ liên quan tới cột x, đôi khi chỉ liên quan tới cột y, và đôi khi cả 2 cột, thì bạn có thể chọn tạo ra 2 chỉ số riêng rẽ cho x và y, dựa vào sự kết hợp các chỉ số để xử lý các câu hỏi mà sử dụng cả 2 cột đó. Bạn cũng có thể tạo ra một chỉ số nhiều cột (x, y). Chỉ số này thường có thể sẽ hiệu quả hơn so với sự kết hợp chỉ số cho các truy vấn có liên quan tới cả 2 cột, mà như được thảo luận trong Phần 11.3, nó có thể hầu như là vô dụng đối với các truy vấn chỉ liên quan tới y, sao

cho nó sẽ không là chỉ số duy nhất. Một sự kết hợp chỉ số nhiều cột và một chỉ số riêng rẽ trên y cũng có thể phục vụ một cách hợp lý. Đối với các truy vấn chỉ liên quan tới x, chỉ số nhiều cột có thể được sử dụng, dù nó có thể lớn hơn và vì thế chậm hơn so với một chỉ số trên chỉ x. Lựa chọn thay thế sau là để tạo ra tất cả 3 chỉ số, mà điều này có khả năng chỉ hợp lý nếu bảng đó được tìm kiếm thường nhiều hơn so với nó được cập nhật và tất cả 3 dạng truy vấn đó là phổ biến. Nếu một trong các dạng truy vấn là phổ biến ít hơn so với dạng khác, thì bạn có lẽ muốn thiết lập cho việc tạo ra chỉ 2 chỉ số mà trùng khớp tốt nhất với các dạng phổ biến đó.

11.6. Chỉ số duy nhất

Các chỉ số cũng được sử dụng để ép tuân thủ tính duy nhất giá trị của một cột, hoặc tính duy nhất các giá trị được kết hợp của nhiều hơn một cột.

```
CREATE UNIQUE INDEX name ON table (column [, ...])
```

Hiện hành, chỉ các chỉ số B-tree có thể được khai báo duy nhất.

Khi một chỉ số được khai báo duy nhất, các hàng của nhiều bảng với các giá trị được đánh chỉ số như nhau sẽ không được phép. Các giá trị null không được xem xét bằng nhau. Một chỉ số duy nhất của nhiều cột sẽ chỉ từ chối các trường hợp nơi mà tất cả các cột được đánh chỉ số là bằng nhau trong nhiều hàng.

PostgreSQL tự động tạo một chỉ số duy nhất khi một ràng buộc duy nhất hoặc khóa chính được xác định cho một bảng. Chỉ số đó bao trùm các cột tạo thành khóa chính hoặc ràng buộc duy nhất (một chỉ số nhiều cột, nếu phù hợp), và là cơ chế ép tuân thủ ràng buộc đó.

> **Lưu ý**: Cách được ưu tiên để bổ sung một ràng buộc duy nhất tới một bảng là ALTER TABLE ... ADD CONSTRAINT. Sử dụng các chỉ số để ép tuân thủ các ràng buộc duy nhất có thể được xem là chi tiết triển khai sẽ không được đánh giá trực tiếp. Tuy nhiên, người ta sẽ nhận thức được rằng không có nhu cầu để bằng tay tạo ra các chỉ số trong các cột duy nhất; làm như vậy có thể chỉ đúp bản chỉ số được tạo ra đó một cách tự động.

11.7. Chỉ số trong các biểu thức

Một cột chỉ số không chỉ cần là một cột của bảng nằm bên dưới, mà có thể là một hàm hoặc biểu thức vô hướng được tính toán từ một hoặc nhiều cột của bảng. Chức năng này là hữu dụng để có được sự truy cập nhanh tới các bảng dựa vào các kết quả của các tính toán.

Ví dụ, một cách phổ biến để tiến hành các so sánh chữ hoa chữ thường là sử dụng hàm lower:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

Truy vấn này có thể sử dụng một chỉ số nếu một chỉ số từng được xác định trong kết quả của hàm lower(col1):

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Nếu chúng ta từng khai báo chỉ số UNIQUE này, thì nó có thể ngăn cản sự tạo ra các hàng mà các giá

trị col1 của chúng chỉ khác theo chữ hoa chữ thường, cũng như các hàng mà các giá trị col1 của chúng thực sự y hệt nhau. Vì thế, các chỉ số trong các biểu thức có thể được sử dụng để ép tuân thủ các ràng buộc mà không xác định được như là các ràng buộc duy nhất đơn giản.

Như một ví dụ khác, nếu một người thường tiến hành các truy vấn như:

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

thì có thể đáng tạo ra một chỉ số giống thế này:

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

Cú pháp của lệnh CREATE INDEX thường đòi hỏi viết các dấu ngoặc đơn xung quanh các biểu thức chỉ số, như được thấy trong ví dụ thứ 2. Các dấu ngoặc đơn đó có thể được bỏ qua nếu biểu thức chỉ là một lời gọi hàm, như trong ví dụ đầu.

Các biểu thức chỉ số khá là đắt giá để duy trì, vì (các) biểu thức dẫn xuất phải được tính toán sẽ không được tính toán lại trong quá trình một tìm kiếm được đánh chỉ số, vì chúng được lưu trữ rồi trong chỉ số đó. Trong cả 2 ví dụ ở trên, hệ thống coi truy vấn đó chỉ như WHERE indexedcolumn = 'constant' và vì thế tốc độ của tìm kiếm là tương đương với bất kỳ truy vấn chỉ số đơn giản nào khác. Vì thế, các chỉ số trong các biểu thức là hữu dụng khi tốc độc truy vấn là quan trọng hơn so với tốc độ chèn và cập nhật.

11.8. Chỉ số một phần

Một *chỉ số một phần* (*Partial Index*) là một chỉ số được xây dựng qua một tập con của một bảng; tập con đó được một biểu thức điều kiện xác định (được gọi là thuộc tính của chỉ số một phần đó). Chỉ số đó bao gồm các khoản đầu vào chỉ cho các hàng của các bảng thỏa mãn thuộc tính đó. Các chỉ số một phần là một chức năng chuyên biệt, nhưng có vài tình huống theo đó chúng là hữu dụng.

Một lý do chính cho việc sử dụng một chỉ số một phần là để tránh việc đánh chỉ số các giá trị phổ biến. Vì việc tìm kiếm truy vấn cho một giá trị phổ biến (một giá trị mà tính tới hơn một ít % của tất cả các hàng của bảng) sẽ không sử dụng chỉ số đó, nên không quan trọng trong việc giữ các hàng đó trong chỉ số đó. Điều này làm giảm kích cỡ của chỉ số, nó sẽ làm tăng tốc độ các truy vấn mà sử dụng chỉ số đó. Nó cũng sẽ tăng tốc nhiều hoạt động cập nhật bảng vì chỉ số đó không cần phải được cập nhật trong tất cả các trường hợp. Ví dụ 11-1 chỉ ra một ứng dụng có thể của ý tưởng này.

Ví dụ 11-1. Thiết lập một chỉ số một phần để loại trừ các giá trị phổ biến

Giả sử bạn đang lưu trữ các lưu ký truy cập máy chủ web trong một cơ sở dữ liệu. Hầu hết các truy cập xuất phát từ dãy địa chỉ IP của tổ chức của bạn nhưng một số là từ đâu đó khác (như, các nhân viên trong các kết nối quay số điện thoại).

Nếu các tìm kiếm IP của bạn ban đầu là cho các truy cập bên ngoài, thì bạn có thể không cần đánh chỉ số dãy IP tương ứng với đoạn mạng của tổ chức của bạn.

Giả thiết một bảng giống thế này:

```
CREATE TABLE access_log (
        url varchar,
        client_ip inet,
        ...
);
```

Để tạo một chỉ số một phần phù hợp với ví dụ của chúng tôi, hãy sử dụng một lệnh như thế này:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
        client_ip < inet '192.168.100.255');
```

Một truy vấn điển hình có thể sử dụng chỉ số này có thể là:

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

Một truy vấn không thể sử dụng chỉ số này là:

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

Quan sát thấy rằng dạng chỉ số một phần này đòi hỏi các giá trị phổ biến được xác định trước, sao cho các chỉ số một phần như vậy được sử dụng tốt nhất cho các phân phối dữ liệu không thay đổi. Các chỉ số đó có thể thỉnh thoảng được tái tạo lại để tinh chỉnh cho các phân phối dữ liệu mới, mà điều này bổ sung thêm cho nỗ lực duy trì.

Sử dụng có thể khác đối với một chỉ số một phần là để loại trừ các giá trị khỏi chỉ số mà tải công việc của truy vấn điển hình không được quan tâm; điều này được chỉ ra trong ví dụ 11-2. Điều này dẫn tới một số ưu điểm như được liệt kê ở trên, nó ngăn chặn các giá trị "không quan tâm" khỏi việc được truy cập thông qua chỉ số đó, thậm chí nếu một sự quét chỉ số có thể có lợi trong trường hợp đó. Rõ ràng, việc thiết lập các chỉ số một phần cho dạng kịch bản này sẽ đòi hỏi nhiều thận trọng và kinh nghiệm hơn.

Ví dụ 11-2. Thiết lập một chỉ số một phần để loại trừ các giá trị không quan tâm

Nếu bạn có một bảng chứa các đơn hàng có hóa đơn và không có hóa đơn, nơi mà các đơn hàng không có hóa đơn chiếm một phần nhỏ của bảng tổng và chúng là các hàng được truy cập nhiều nhất, thì bạn có thể cải thiện hiệu năng bằng việc tạo một chỉ số chỉ trong các hàng không có hóa đơn đó. Lệnh để tạo chỉ số đó có thể trông giống thế này:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
        WHERE billed is not true;
```

Một truy vấn có thể sẽ sử dụng chỉ số này có thể là:

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

Tuy nhiên, chỉ số đó cũng có thể được sử dụng trong các truy vấn không liên quan tới order_nr, như:

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

Điều này không thật hiệu quả như một chỉ số một phần trong cột amount có thể, vì hệ thống phải quét toàn bộ chỉ số. Hơn nữa, nếu có khá ít đơn hàng không có hóa đơn, thì bằng việc sử dụng chỉ số một phần này chỉ tìm thấy các đơn hàng không có hóa đơn có thể là một thành công.

Lưu ý rằng truy vấn đó không thể sử dụng chỉ số này:

```
SELECT * FROM orders WHERE order_nr = 3501;
```

Đơn hàng 3501 có thể nằm trong số các đơn hàng có hóa đơn hoặc không có hóa đơn.

Ví dụ 11-2 cũng minh họa rằng cột được đánh chỉ số và cột được sử dụng trong thuộc tính không cần trùng nhau. PostgreSQL hỗ trợ các chỉ số một phần với các thuộc tính tùy chọn, miễn là chỉ các cột của bảng đang được đánh chỉ số có liên quan. Tuy nhiên hãy nhớ trong đầu rằng thuộc tính phải trùng khớp với các điều kiện được sử dụng trong các truy vấn được hỗ trợ để có lợi từ chỉ số đó. Để chính xác, một chỉ số một phần có thể được sử dụng trong một truy vấn chỉ nếu hệ thống đó có thể thừa nhận rằng điều kiện WHERE của truy vấn tự động ngụ ý thuộc tính của chỉ số đó. PostgreSQL không có một trình chứng minh định lý có thể nhận thức được các biểu thức tương đương nhau về mặt toán học được viết ở các dạng khác nhau. (Một trình chứng minh định lý chung như vậy không chỉ là khó để tạo ra, mà nó còn có thể là quá chậm đối với bất kỳ sự sử dụng thực tế nào). Hệ thống có thể nhận biết được các tác động không bằng nhau đơn giản, ví dụ "`x < 1`" ngụ ý "`x < 2`"; nếu không thì điều kiện thuộc tính phải chính xác khớp với phần của điều kiện WHERE của truy vấn hoặc chỉ số đó sẽ không được thừa nhận là có khả năng sử dụng. Việc trùng khớp diễn ra trong thời gian lập kế hoạch truy vấn, không trong thời gian chạy. Kết quả là, các mệnh đề truy vấn có tham số không làm việc với một chỉ số một phần. Ví dụ, truy vấn được chuẩn bị với một tham số có thể chỉ định "`x < ?`" mà sẽ không bao giờ ngụ ý "`x < 2`" đối với tất cả các giá trị có thể của tham số đó.

Sử dụng có khả năng thứ 3 cho các chỉ số một phần không đòi hỏi chỉ số đó sẽ được sử dụng trong các truy vấn hoàn toàn. Ý tưởng ở đây là để tạo ra một chỉ số duy nhất đối với một tập con của một bảng, như trong ví dụ 11-3. Điều này ép tuân thủ tính độc nhất trong các hàng thỏa mãn thuộc tính chỉ số đó, không có việc ràng buộc các hàng không thỏa mãn.

Ví dụ 11-3. Thiết lập một chỉ số một phần duy nhất

Giả thiết là chúng ta có một bảng mô tả các đầu ra của kiểm thử. Chúng ta muốn đảm bảo rằng chỉ có một khoản đầu vào "thành công" cho một đối tượng được đưa ra và sự kết hợp đích, nhưng có thể có bất kỳ số các khoản đầu vào "không thành công" nào. Đây là một cách để làm điều này:

```
CREATE TABLE tests (
        subject text,
        target text,
        success boolean,
        ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
        WHERE success;
```

Đây là một tiếp cận đặc biệt hiệu quả khi có ít các kiểm thử thành công và nhiều kiểm thử không thành công.

Cuối cùng, một chỉ số một phần cũng được sử dụng để ghi đè các lựa chọn kế hoạch truy vấn của hệ thống. Hơn nữa, các tập hợp dữ liệu với các phân phối đặc biệt có thể làm cho hệ thống sử dụng một chỉ số khi nó thực sự không nên. Trong trường hợp đó chỉ số có thể được thiết lập sao cho nó là

không sẵn sàng đối với truy vấn vi phạm. Thông thường, PostgreSQL thực hiện các lựa chọn hợp lý về sử dụng chỉ số (như, nó tránh chúng khi truy xuất các giá trị phổ biến, nên ví dụ trước đó thực sự chỉ tiết kiệm kích cỡ của chỉ số, nó không được yêu cầu để tránh sử dụng chỉ số), và các lựa chọn kế hoạch không đúng một cách hiển nhiên là lý do cho một báo cáo lỗi.

Nhớ trong đầu rằng việc thiết lập một chỉ số một phần chỉ ra rằng bạn biết ít nhất như nhiều trình hoạch định truy vấn cũng biết, đặc biệt bạn biết khi một chỉ số có thể có lợi. Việc hình thành tri thức này đòi hỏi kinh nghiệm và sự hiểu biết cách mà các chỉ số trong PostgreSQL làm việc. Trong hầu hết các trường hợp, ưu thế của một chỉ số một phần đối với một chỉ số thông thường sẽ là tối thiểu.

Nhiều thông tin hơn về các chỉ số một phần có thể thấy trong *Trường hợp cho các chỉ số một phần, Việc đánh chỉ số một phần trong PostgreSQL: dự án nghiên cứu, và các chỉ số một phần được khái quát hóa (phiên bản được nắm bắt)*.

11.9. Lớp toán tử và họ toán tử

Một định nghĩa chỉ số có thể chỉ định một lớp toán tử cho từng cột của một chỉ số.

```
CREATE INDEX name ON table (column opclass [sort options] [, ...]);
```

Lớp toán tử nhận diện các toán tử sẽ được chỉ số cho cột đó sử dụng. Ví dụ, một chỉ số B-tree ở dạng int4 có thể sử dụng lớp int4_ops; lớp toán tử này bao gồm các hàm so sánh cho các giá trị dạng int4. Trong thực tế lớp toán tử mặc định cho dạng dữ liệu cột thường là đủ. Lý do chính cho việc có các lớp toán tử là đối với một số dạng dữ liệu, có thể có nhiều hơn một hành vi chỉ số có ý nghĩa. Ví dụ, chúng ta có thể muốn sắp xếp một dạng dữ liệu cột phức tạp hoặc bằng giá trị tuyệt đối hoặc bằng phần thực tế. Chúng ta có thể làm điều này bằng việc xác định 2 lớp toán tử cho dạng dữ liệu đó và sau đó lựa chọn lớp phù hợp khi thực hiện một chỉ số. Lớp toán tử đó xác định trật tự sắp xếp cơ bản (mà có thể sau đó được thay đổi bằng việc bổ sung thêm các lựa chọn sắp xếp ASC /DESC và/hoặc NULLS FIRST /NULLS LAST).

Cũng có một số lớp toán tử được xây dựng sẵn ngoài các lớp mặc định:

- Các lớp toán tử text_pattern_ops , varchar_pattern_ops và bpchar_pattern_ops hỗ trợ các chỉ số B-tree ở các dạng text, varchar và char một cách tương ứng. Sự khác biệt với các lớp toán tử mặc định là các giá trị sẽ được so sánh khắt khe từng ký tự một thay vì theo các qui tắc đối chiếu đặc thù bản địa. Điều này làm cho các lớp toán tử đó phù hợp để các truy vấn có liên quan tới các biểu thức khớp mẫu sử dụng (các biểu thức thông thường LIKE hoặc POSIX) khi cơ sở dữ liệu không sử dụng bản địa tiêu chuẩn "C". Như một ví dụ, bạn có thể đánh chỉ số một cột varchar như thế này:

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  Lưu ý là bạn cũng nên tạo một chỉ số với lớp toán tử mặc định nếu bạn muốn các chỉ số có liên quan tới các toán tử so sánh thông thường <, <=, >, hoặc >= sử dụng một chỉ số. Các truy vấn như vậy không thể sử dụng các lớp toán tử xxx_pattern_ops. (Tuy nhiên, thường thì các so sánh bằng nhau có thể sử dụng các lớp toán tử đó). Nếu có khả năng để tạo nhiều chỉ

số trong cùng cột y hệt với các lớp khác nhau. Nếu bạn sử dụng bản địa C, thì bạn không cần các lớp toán tử xxx_pattern_ops, vì một chỉ số với lớp toán tử mặc định là sử dụng được cho các truy vấn khớp mẫu trong bản địa C.

Truy vấn sau chỉ ra tất cả các lớp toán tử được xác định:

```
SELECT am.amname AS index_method,
        opc.opcname AS opclass_name
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

Một lớp toán tử thực sự chỉ là một tập con của một cấu trúc lớn hơn gọi là một họ toán tử. Trong các trường hợp nơi mà vài dạng dữ liệu có các hành vi tương tự nhau thì thường hữu dụng để định nghĩa các toán tử dạng liên dữ liệu và cho phép chúng làm việc với các chỉ số. Để làm điều này, các lớp toán tử cho từng trong số các dạng đó phải được nhóm vào trong cùng một họ toán tử. Các toán tử liên dạng là các thành viên của họ đó, nhưng không có liên quan với bất kỳ lớp duy nhất nào trong họ đó.

Truy vấn này chỉ ra tất cả các họ toán tử được định nghĩa và tất cả các toán tử được đưa vào trong từng họ:

```
SELECT am.amname AS index_method,
        opf.opfname AS opfamily_name,
        amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
        amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

11.10. Kiểm tra sử dụng chỉ số

Dù các chỉ số trong PostgreSQL không cần duy trì hay tinh chỉnh, thì vẫn quan trọng phải kiểm tra các chỉ số nào thực sự được tải công việc truy vấn trong cuộc sống thực sử dụng. Việc kiểm tra sử dụng chỉ số cho một truy vấn riêng rẽ được thực hiện với lệnh EXPLAIN; ứng dụng của nó cho mục đích này được minh họa trong Phần 14.1. Cũng có khả năng tập hợp toàn bộ các số liệu thống kê về sử dụng các chỉ số trong một máy chủ đang chạy, như được mô tả trong Phần 27.2.

Là khó để tạo một thủ tục chung cho việc xác định các chỉ số nào để tạo ra. Có một số trường hợp điển hình đã được chỉ ra trong các ví dụ khắp các phần trước. Một sự việc thí điểm tốt thường là cần thiết. Phần còn lại của phần này đưa ra một số mẹo cho điều đó:

- Luôn chạy lệnh ANALYZE trước tiên. Lệnh này thu thập các số liệu thống kê về phân phối các giá trị trong bảng. Thông tin này được yêu cầu để đánh giá số lượng các hàng được một truy vấn trả về, nó là cần thiết cho trình hoạch định để chỉ định các chi phí thực tế cho từng kế hoạch truy vấn có khả năng. Thiếu bất kỳ số liệu thống kê thực tế nào, thì một số giá trị mặc định sẽ được giả thiết, chúng hầu như chắc chắn sẽ là không chính xác. Việc kiểm tra sự sử dụng chỉ số của một ứng dụng mà không chạy ANALYZE vì thế là một lý do bị thiếu. Xem Phần 23.1.3 và Phần 23.1.5 để có thêm thông tin.

- Sử dụng các dữ liệu thực tế cho thí điểm. Việc sử dụng các dữ liệu kiểm thử cho việc thiết lập các chỉ số sẽ nói cho bạn các chỉ số nào bạn cần cho các dữ liệu kiểm thử đó, mà điều đó là tất cả.

  Đặc biệt sống còn để sử dụng các tập hợp dữ liệu kiểm thử rất nhỏ. Trong khi việc lựa chọn 1.000 trong số 100.000 hàng có thể là một ứng viên cho một chỉ số, thì việc lựa chọn 1 trong số 100 hàng sẽ khó là như vậy, vì 100 hàng có thể phù hợp trong một trang đĩa duy nhất, và không có kế hoạch có thể tuần tự chiếm được 1 trang đĩa.

  Cũng thận trọng khi lấy các dữ liệu kiểm thử, nó thường có khả năng tránh được khi ứng dụng còn chưa nằm trong sản xuất. Các giá trị là rất nhỏ, hoàn toàn ngẫu nhiên, hoặc được chèn vào theo trật tự được sắp xếp sẽ bóp méo các số liệu thống kê khỏi sự phân phối mà dữ liệu thực có thể có.

- Khi các chỉ số không được sử dụng, có thể là hữu dụng cho việc kiểm thử để ép tuân thủ sử dụng chúng. Có các tham số thời gian chạy (run time) có thể tắt các dạng kế hoạch khác nhau (xem Phần 18.6.1). Ví dụ, việc tắt các sự quét tuần tự (enable_seqscan) và các liên kết lặp lồng nhau (enable_nestloop), chúng là các kế hoạch cơ bản nhất, sẽ ép hệ thống phải sử dụng một kế hoạch khác. Nếu hệ thống vẫn còn chọn một sự quét tuần tự hoặc liên kết lồng nhau thì có thể sẽ có một lý do cơ bản hơn vì sao chỉ số đó không được sử dụng; ví dụ, điều kiện truy vấn không khớp với chỉ số đó. (Dạng truy vấn nào có thể sử dụng dạng chỉ số nào được giải thích trong các phần trước).

- Nếu việc ép sử dụng chỉ số không sử dụng chỉ số, thì sau đó có 2 khả năng: Hoặc hệ thống là đúng và việc sử dụng chỉ số quả thực là không phù hợp, hoặc các ước lượng chi phí của các kế hoạch truy vấn đang không phản ánh được thực tế. Vì thế bạn nên định thời gian cho truy vấn của bạn với và không với các chỉ số. Lệnh EXPLAIN ANALYZE có thể là hữu dụng ở đây.

- Nếu hóa ra là các ước lượng chi phí là sai, một lần nữa, có thể có 2 khả năng. Tổng chi phí được tính toán từ các chi phí cho từng hàng của từng nút kế hoạch định thời gian cho ước lượng chọn lọc của nút kế hoạch. Các chi phí được ước tính cho các nút kế hoạch đó có thể được tinh chỉnh do các số liệu thống kê không đủ. Có thể là có khả năng để cải thiện điều này bằng việc tinh chỉnh các số liệu thống kê - việc tập hợp các tham số (xem ALTER TABLE).

  Nếu bạn không thành công trong việc tinh chỉnh các chi phí cho phù hợp hơn, thì bạn có thể phải sắp xếp lại để ép sử dụng chỉ số một cách rõ ràng. Bạn cũng có thể muốn liên hệ với các lập trình viên PostgreSQL để xem xét vấn đề.

Chương 12. Tìm kiếm toàn văn

12.1. Giới thiệu

Tìm kiếm toàn văn (hoặc chỉ là tìm kiếm văn bản) đưa ra khả năng nhận diện các tài liệu ngôn ngữ tự nhiên mà làm thỏa mãn một truy vấn, và thường để sắp xếp chúng thích đáng cho sự truy vấn. Dạng tìm kiếm phổ biến nhất là tìm tất cả các tài liệu có chứa các khoản truy vấn được đưa ra và trả chúng về theo trật tự tương tự của chúng đối với truy vấn đó. Các khái niệm query và similarity là rất mềm dẻo và phụ thuộc vào ứng dụng cụ thể. Tìm kiếm đơn giản nhất coi query như một tập hợp các từ và similarity như tần suất của các từ truy vấn trong tài liệu đó.

Các toán tử tìm kiếm văn bản đã tồn tại trong các cơ sở dữ liệu nhiều năm. PostgreSQL có các toán tử ~, ~*, LIKE, và ILIKE cho các dạng dữ liệu văn bản, nhưng chúng thiếu nhiều thuộc tính cơ bản được các hệ thống thông tin hiện đại yêu cầu:

- Không có hỗ trợ ngôn ngữ, thậm chí cho tiếng Anh. Các biểu thức thông thường là không đủ vì chúng không thể dễ dàng điều khiển các từ dẫn xuất, như, satisfies và satisfy. Bạn có thể bỏ qua các tài liệu mà có chứa satisfies, dù bạn có thể muốn tìm chúng khi tìm kiếm satisfy. Có khả năng sử dụng OR để tìm nhiều dạng được dẫn xuất, nhưng điều này là nặng nề và dễ bị lỗi (một số từ có thể có vài ngàn dẫn xuất).
- Chúng không đưa ra trật tự (xếp hạng) các kết quả tìm kiếm, mà làm cho chúng không hiệu quả khi hàng ngàn tài liệu trùng khớp được tìm thấy.
- Chúng có xu hướng chậm vì không có hỗ trợ chỉ số, nên chúng phải xử lý tất cả các tài liệu cho từng tìm kiếm.

Đánh chỉ số toàn văn cho phép các tài liệu sẽ được tiền xử lý và một chỉ số được lưu cho việc tìm kiếm nhanh sau này. Việc xử lý bao gồm:

> *Việc phân tích các tài liệu trong các thẻ token*. Là hữu dụng để nhận diện các lớp thẻ token khác nhau, như, các số, từ, từ phức tạp, địa chỉ thư điện tử, sao cho chúng có thể được xử lý khác nhau. Về nguyên tắc các lớp thẻ token phụ thuộc vào ứng dụng đặc thù, nhưng đối với hầu hết các mục đích thì là phù hợp để sử dụng một tập hợp các lớp được xác định sẵn trước. PostgreSQL sử dụng một trình phân tích để thực hiện bước này. Một trình phân tích tiêu chuẩn được cung cấp, và các trình phân tích tùy biến có thể được tạo ra cho nhu cầu đặc thù.
>
> *Việc biến đổi các thẻ token thành các từ vị*. Một từ vị là một chuỗi, hệt như một thẻ token, nhưng nó được bình thường hóa sao cho các dạng khác nhau của cùng một từ được làm cho giống nhau. Ví dụ, sự bình thường hóa hầu hết luôn bao gồm việc biến các chữ hoa thành chữ thường, và thường có liên quan tới việc loại bỏ các hậu tố (như ký tự s hoặc es trong tiếng Anh). Điều này cho phép các tìm kiếm để tìm các dạng phương án của cùng một từ, không nặng nhọc đưa vào tất cả các phương án có thể. Hơn nữa, bước này thường loại bỏ các từ chết (stop word), chúng là các từ quá phổ biến mà chúng là vô dụng cho việc tìm

kiếm. (Ngắn gọn, sau đó, các thẻ token là các phân đoạn thô của văn bản tài liệu, trong khi các từ vị là các từ được tin tưởng là hữu dụng cho việc đánh chỉ số và tìm kiếm). PostgreSQL sử dụng các từ điển để thực hiện bước này. Các từ điển tiêu chuẩn khác nhau được cung cấp, và các từ điển tùy biến có thể được tạo ra cho các nhu cầu đặc thù.

Việc lưu trữ các tài liệu được tiền xử lý được tối ưu hóa cho việc tìm kiếm. Ví dụ, từng tài liệu có thể được trình bày như một mảng được sắp xếp các từ vị được bình thường hóa. Cùng với các từ vị thường có mong muốn lưu trữ các thông tin vị trí để sử dụng cho xếp hạng gần đúng, sao cho một tài liệu có chứa một vùng "đậm đặc" hơn các từ truy vấn được chỉ định hạng cao hơn so với một tài liệu với các từ truy vấn rời rạc.

Các từ điển cho phép kiểm soát mịn hơn đối với cách mà các thẻ token được bình thường hóa. Với các từ điển thích hợp, bạn có thể:

- Định nghĩa các từ chết sẽ không được đánh chỉ số.
- Ánh xạ các từ đồng nghĩa tới một từ duy nhất bằng việc sử dụng Ispell.
- Ánh xạ các cụm từ tới một từ duy nhất bằng việc sử dụng một từ điển đồng nghĩa.
- Ánh xạ các phương án khác nhau của một từ tới một dạng kinh điển bằng việc sử dụng một từ điển Ispell.
- Ánh xạ các phương án khác nhau của một từ tới một dạng kinh điển bằng việc sử dụng các qui tắc cọng bông tuyết (Snowball stemmer).

Dạng dữ liệu tsvector được cung cấp cho việc lưu trữ các tài liệu được tiền xử lý, cùng với một dạng tsquery cho việc thể hiện các truy vấn được xử lý (Phần 8.11). Có nhiều hàm và toán tử có sẵn cho các dạng dữ liệu đó (Phần 9.13), quan trọng nhất trong số đó là toán tử trùng khớp @@, mà chúng tôi giới thiệu trong Phần 12.1.2. Các tìm kiếm toàn văn có thể được tăng tốc bằng việc sử dụng các chỉ số (Phần 12.9).

12.1.1. Tài liệu là gì?

Một *tài liệu* là đơn vị tìm kiếm trong một hệ thống tìm kiếm toàn văn; ví dụ, bài báo của một tạp chí hoặc thông điệp thư điện tử. Máy tìm kiếm văn bản phải có khả năng phân tích các tài liệu và lưu trữ các điều liên quan của các từ vựng (các từ khóa) với tài liệu cha của chúng. Sau này, các điều liên quan được sử dụng để tìm kiếm các tài liệu có chứa các từ truy vấn.

Đối với các tìm kiếm trong PostgreSQL, một tài liệu thường là một trường văn bản bên trong một hàng của một bảng cơ sở dữ liệu, hoặc có thể là một sự kết hợp (sự ghép) các trường như vậy, có thể được lưu trữ trong vài bảng hoặc có được một cách năng động. Nói cách khác, một tài liệu có thể được xây dựng từ các phần khác nhau cho việc đánh chỉ số và nó có thể không được lưu trữ ở bất kỳ đâu như một tổng thể. Ví dụ:

```
SELECT title || ' ' || author || ' ' || abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;
```

```
SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE mid = did AND mid = 12;
```

> **Lưu ý**: Thực sự, trong các truy vấn ví dụ đó, coalesce sẽ được sử dụng để ngăn ngừa một thuộc tính NULL duy nhất khỏi việc gây ra một kết quả NULL cho tài liệu tổng thể.

Khả năng khác là lưu trữ các tài liệu như các tệp văn bản đơn giản trong hệ thống tệp. Trong trường hợp này, cơ sở dữ liệu có thể được sử dụng để lưu trữ chỉ số toàn văn và để thực thi các tìm kiếm, và một số mã định danh độc nhất có thể được sử dụng để truy xuất tài liệu từ hệ thống tệp. Tuy nhiên, việc truy xuất các tài liệu từ bên ngoài cơ sở dữ liệu đòi hỏi các quyền của siêu người sử dụng (superuser) hoặc sự hỗ trợ của các hàm đặc biệt, nên điều này thường ít thuận tiện hơn là việc giữ cho tất cả các dữ liệu nằm bên trong PostgreSQL. Hơn nữa, việc giữ mọi điều bên trong cơ sở dữ liệu cho phép dễ dàng truy cập tới các siêu dữ liệu của tài liệu để hỗ trợ trong việc đánh chỉ số và hiển thị.

Vì các mục đích tìm kiếm văn bản, từng tài liệu phải được giảm thiểu về định dạng tsvector được tiền xử lý. Việc tìm kiếm và xếp hạng được thực hiện toàn bộ trong sự trình bày một tài liệu của tsvector - văn bản gốc chỉ cần được truy xuất khi tài liệu đó được chọn để hiển thị cho một người sử dụng. Chúng tôi vì thế thường nói về tsvector như là tài liệu, nhưng tất nhiên nó chỉ là một sự trình bày cô đọng của tài liệu đầy đủ đó.

12.1.2. Trùng khớp văn bản cơ bản

Việc tìm kiếm toàn văn trong PostgreSQL dựa vào toán tử trùng khớp @@, nó trả về đúng nếu một tsvector (tài liệu) trùng với tsquery (truy vấn). Không thành vấn đề dạng dữ liệu nào được ghi trước:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;

?column?
----------
t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;

?column?
----------
f
```

Như ví dụ ở trên gợi ý, một tsquery không chỉ là một văn bản thô, mà là bất kỳ văn bản nào nhiều hơn một tsvector. Một tsquery chứa các khoản tìm kiếm, chúng phải là các từ vị được bình thường hóa rồi, và có thể kết hợp nhiều khoản bằng việc sử dụng các toán tử AND, OR, và NOT. (Để có các chi tiết, xem Phần 8.11). Có các hàm to_tsquery và plainto_tsquery là hữu ích trong việc biến đổi văn bản do người sử dụng viết thành một tsquery phù hợp, ví dụ bằng việc bình thường hóa các từ xuất hiện trong văn bản đó. Tương tự, to_tsvector được sử dụng để phân tích và bình thường hóa một chuỗi của tài liệu. Vì thế trong thực tế một sự trùng khớp tìm kiếm văn bản có thể trông giống hơn thế này:

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');

?column?
----------
t
```

Quan sát thấy rằng sự trùng khớp này có thể không thành công nếu được viết như là:

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');

?column?
----------
f
```

vì ở đây không sự bình thường hóa nào của từ rats sẽ xảy ra cả. Các phần tử của một tsvector là các từ vị, chúng được giả thiết được bình thường hóa rồi, nên rats không khớp với rat.

Toán tử @@ cũng hỗ trợ đầu vào text, cho phép sự biến đổi rõ ràng của một chuỗi văn bản sang tsvector hoặc tsquery sẽ được bỏ qua trong các trường hợp đơn giản. Các biến thể sẵn sàng là:

```
tsvector @@ tsquery
tsquery @@ tsvector
text @@ tsquery
text @@ text
```

2 dòng đầu của các dòng trên chúng ta đã thấy rồi. Văn bản dạng text @@ tsquery là tương đương với to_tsvector(x) @@ y. Dạng text @@ text là tương đương với to_tsvector(x) @@ plainto_tsquery(y).

12.1.3. Cấu hình

Bên trên là tất cả các ví dụ tìm kiếm văn bản đơn giản. Như được nêu trước đó, chức năng tìm kiếm toàn văn bao gồm khả năng thực hiện nhiều điều hơn: bỏ qua việc đánh chỉ số các từ nhất định (các từ chết), xử lý các từ đồng nghĩa và sử dụng việc phân tích phức tạp, như, phân tích dựa vào nhiều hơn là chỉ khoảng trắng. Chức năng này được các *cấu hình tìm kiếm văn bản* kiểm soát. PostgreSQL đi với các cấu hình được định nghĩa trước cho nhiều ngôn ngữ, và bạn có thể dễ dàng tạo các cấu hình của riêng bạn. (lệnh \dF của psql chỉ ra tất cả các cấu hình có sẵn).

Trong quá trình cài đặt một cấu hình phù hợp được lựa chọn và default_text_search_config được thiết lập phù hợp trong postgresql.conf. Nếu bạn đang sử dụng cấu hình tìm kiếm văn bản y hệt cho toàn bộ cụm đó thì bạn có thể sử dụng giá trị trong postgresql.conf. Để sử dụng các cấu hình khác thông qua cụm đó nhưng cấu hình y hệt trong bất kỳ một cơ sở dữ liệu nào, hãy sử dụng ALTER DATABASE ... SET. Nếu không, bạn có thể thiết lập default_text_search_config trong từng phiên.

Mỗi hàm tìm kiếm văn bản phụ thuộc vào một cấu hình có một đối số tùy chọn regconfig, sao cho cấu hình sẽ sử dụng có thể được chỉ định một cách rõ ràng. default_text_search_config được sử dụng chỉ khi đối số này bị bỏ qua.

Để làm cho nó dễ dàng hơn để xây dựng các cấu hình tìm kiếm văn bản tùy biến, một cấu hình được xây dựng từ các đối tượng cơ sở dữ liệu đơn giản hơn. Cơ sở tìm kiếm văn bản của PostgreSQL đưa ra 4 dạng đối tượng cơ sở dữ liệu có liên quan tới cấu hình:

- Các *trình phân tích tìm kiếm văn bản* chia các tài liệu thành các thẻ token và phân loại từng token (ví dụ, như các từ hoặc các số).

- Các *từ điển tìm kiếm văn bản* biến đổi các thẻ token thành dạng được bình thường hóa và từ chối các từ chết.
- Các *mẫu template tìm kiếm văn bản* đưa ra các hàm cho các từ điển nằm bên dưới. (Một từ điển đơn giản chỉ định một mẫu template và một tập hợp các tham số cho mẫu temlate đó).
- Các *cấu hình tìm kiếm văn bản* lựa chọn một trình phân tích và một tập hợp các từ điển để sử dụng để bình thường hóa các thẻ token được trình phân tích đó tạo ra.

Các trình phân tích tìm kiếm văn bản và các mẫu template được xây dựng từ các hàm C mức thấp; vì thế nó đòi hỏi khả năng lập trình C để phát triển các hàm mới, và các quyền ưu tiên của siêu người sử dụng - superuser để cài đặt một hàm vào một cơ sở dữ liệu. (Có các ví dụ về các trình phân tích và các mẫu template bổ sung trong vùng contrib/ của phân phối PostgreSQL). Vì các từ điển và các cấu hình chỉ tham số hóa và kết nối cùng với một số trình phân tích và mẫu template nằm bên dưới, nên không quyền ưu tiên đặc biệt nào là cần thiết để tạo ra một từ điển hoặc cấu hình mới. Các ví dụ về việc tạo các từ điển và cấu hình tùy biến xuất hiện sau trong chương này.

12.2. Bảng và chỉ số

Các ví dụ trong phần trước đã minh họa việc trùng khớp toàn văn bằng việc sử dụng các chuỗi hằng đơn giản. Phần này chỉ ra cách để tìm các dữ liệu của bảng, sử dụng tùy chọn các chỉ số.

12.2.1. Tìm kiếm bảng

Có khả năng tiến hành một tìm kiếm toàn văn mà không có một chỉ số. Một truy vấn đơn giản in ra tiêu đề của từng hàng có chứa từ friend trong trường thân của nó là:

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

Điều này cũng sẽ tìm các từ có liên quan như friends và friendly, vì tất cả chúng được giảm về cùng y hệt từ vị được bình thường hóa.

Truy vấn ở trên chỉ định rằng cấu hình english sẽ được sử dụng để phân tích và bình thường hóa các chuỗi đó. Như một lựa chọn chúng ta có thể bỏ qua các tham số cấu hình:

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

Truy vấn này sẽ sử dụng cấu hình được thiết lập bằng default_text_search_config.

Một ví dụ phức tạp hơn là để lựa chọn 10 tài liệu gần đây nhất mà có chứa create và table trong tiêu đề hoặc thân:

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

Để làm rõ chúng tôi đã bỏ qua các lời gọi hàm coalesce mà nó có thể sẽ cần thiết để tìm các hàng có

chứa NULL trong 1 trong 2 trường đó.

Dù các truy vấn đó sẽ làm việc không có một chỉ số, thì hầu hết các ứng dụng sẽ thấy tiếp cận này quá chậm, ngoại trừ có lẽ trong trường hợp các tìm kiếm đặc biệt. Sử dụng thực tiễn tìm kiếm văn bản thường đòi hỏi việc tạo một chỉ số.

12.2.2. Tạo chỉ số

Chúng ta có thể tạo một chỉ số GIN (Phần 12.9) để tăng tốc các tìm kiếm văn bản:

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', body));
```

Lưu ý rằng phiên bản 2 đối số của to_tsvector được sử dụng. Chỉ các hàm tìm kiếm văn bản chỉ định một tên cấu hình mới có thể được sử dụng trong các chỉ số của biểu thức (Phần 11.7). Điều này là vì các nội dung chỉ số phải không bị ảnh hưởng vì default_text_search_config. Nếu chúng đã bị ảnh hưởng, thì các nội dung chỉ số có thể là không nhất quán vì các khoản đầu vào khác nhau có thể bao gồm các tsvector từng được tạo ra với các cấu hình tìm kiếm văn bản khác nhau, và có thể không có cách nào để gợi ý là cấu hình nào. Có thể không có khả năng để bỏ và phục hồi chỉ số như vậy một cách đúng đắn.

Vì phiên bản 2 đối số của to_tsvector từng được sử dụng trong chỉ số ở trên, chỉ một tham chiếu truy vấn mà sử dụng phiên bản 2 đối số của to_tsvector với cùng y hệt tên cấu hình sẽ sử dụng chỉ số đó. Đó là, WHERE to_tsvector('english', body) @@ 'a & b' có thể sử dụng chỉ số đó, nhưng WHERE to_tsvector(body) @@ 'a & b' thì không thể. Điều này đảm bảo rằng một chỉ số sẽ chỉ được sử dụng với cùng y hệt cấu hình được sử dụng để tạo các khoản đầu vào của chỉ số đó.

Có khả năng để thiết lập các chỉ số biểu thức phức tạp hơn trong khi tên cấu hình được cột khác chỉ định, như:

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector(config_name, body));
```

trong đó config_name là một cột trong bảng pgweb. Điều này cho phép các cấu hình được trộn trong cùng chỉ số trong khi ghi lại cấu hình nào từng được sử dụng cho từng khoản đầu vào của chỉ số. Điều này có thể là hữu dụng, ví dụ, nếu sự thu thập tài liệu đã có chứa các tài liệu trong các ngôn ngữ khác. Một lần nữa, các truy vấn có nghĩa để sử dụng chỉ số đó phải được cấu tạo thành các cụm từ trùng khớp nhau, như, WHERE to_tsvector(config_name, body) @@ 'a & b'.

Các chỉ số có thể thậm chí ghép nối các cột:

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', title || ' ' || body));
```

Một tiếp cận khác là để tạo ra một cột tsvector riêng rẽ để giữ đầu ra của to_tsvector. Ví dụ này là một sự ghép nối title và body, bằng việc sử dụng coalesce để đảm bảo rằng một trường vẫn sẽ được đánh chỉ số khi trường khác là NULL:

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
        to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));
```

Sau đó chúng ta tạo một chỉ số GIN để tăng tốc tìm kiếm:

```
CREATE INDEX textsearch_idx ON pgweb USING gin(textsearchable_index_col);
```

Bây giờ chúng ta sẵn sàng để thực hiện một tìm kiếm toàn văn nhanh:

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

Khi sử dụng một cột riêng rẽ để lưu trữ đại diện tsvector, điều cần thiết phải tạo ra một trigger để giữ cho cột tsvector là hiện hành bất kỳ khi nào title hoặc body thay đổi. Phần 12.4.3 giải thích làm điều đó như thế nào.

Một ưu điểm của tiếp cận cột riêng rẽ đối với một chỉ số biểu thức là nó không cần phải chỉ định rõ ràng cấu hình tìm kiếm văn bản trong các truy vấn để sử dụng chỉ số đó. Như được chỉ ra trong ví dụ ở trên, truy vấn đó có thể phụ thuộc vào default_text_search_config. Một ưu điểm khác là các tìm kiếm sẽ là nhanh hơn, vì nó sẽ không cần phải làm lại các lời gọi của to_tsvector để kiểm tra các trùng khớp chỉ số. (Điều này quan trọng hơn khi sử dụng một chỉ số GiST hơn là một chỉ số GIN; xem Phần 12.9). Tuy nhiên, tiếp cận chỉ số biểu thức là đơn giản hơn để thiết lập, và nó đòi hỏi ít không gian đĩa hơn vì đại diện tsvector không được lưu trữ rõ ràng.

12.3. Kiểm soát tìm kiếm toàn văn

Để triển khai tìm kiếm toàn văn sẽ phải có một hàm để tạo một tsvector từ một tài liệu và một tsquery từ một truy vấn người sử dụng. Hơn nữa, chúng ta cần phải trả về các két quả theo một trật tự hữu dụng, nên chúng ta cần một hàm so sánh các tài liệu với lưu ý về tính phù hợp của chúng đối với truy vấn đó. Cũng quan trọng để có khả năng hiển thị các kết quả một cách sáng sủa. PostgreSQL đưa ra sự hỗ trợ cho tất cả các hàm đó.

12.3.1. Phân tích tài liệu

PostgreSQL đưa ra hàm to_tsvector cho việc biến đổi một tài liệu sang dạng dữ liệu tsvector.

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

to_tsvector phân tích một tài liệu văn bản thành các thẻ token, giảm các thẻ token thành các từ vị, và trả về một tsvector mà liệt kê các từ vị cùng với các vị trí của chúng trong tài liệu đó. Tài liệu được xử lý theo cấu hình tìm kiếm văn bản mặc định hoặc được chỉ định. Ở đây là một ví dụ đơn giản:

```
SELECT to_tsvector('english', 'a fat cat sat on a mat - it ate a fat rats');

to_tsvector
-----------------------------------------------------
'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

Trong ví dụ ở trên chúng ta thấy rằng kết quả tsvector không chứa các từ a, on, hoặc it, từ rats trở thành rat, và dấu gạch ngang - đã bị bỏ qua.

Hàm tsvector ban đầu gọi một trình phân tích chia văn bản tài liệu thành các thẻ token và chỉ định một dạng cho từng thẻ đó. Đối với từng thẻ token, một danh sách các từ điển (Phần 12.6) được tư vấn, nơi mà danh sách đó có thể khác nhau phụ thuộc vào dạng thẻ token. Từ điển đầu tiên nhận

thức được thẻ token đó phát ra một hoặc nhiều từ vị được bình thường hóa hơn để đại diện cho thẻ token đó. Ví dụ, rats trở thành rat vì một trong các từ điển đã nhận thức được rằng từ rats là một dạng số nhiều của rat. Một số từ được nhận thức như là các từ chết (Phần 12.6.1), chúng vì thế bị bỏ qua vì chúng xảy ra quá thường xuyên không hữu dụng khi tìm kiếm. Trong ví dụ của chúng ta đó là a, on, và it. Nếu không từ điển nào trong danh sách đó nhận thức được thẻ token thì nó cũng bị bỏ qua. Trong ví dụ này điều đó đã xảy ra đối với dấu gạch ngang - vì trong thực tế không từ điển nào được chỉ định cho dạng thẻ token của nó (các ký tự trắng), nghĩa là các thẻ token trắng sẽ không bao giờ được đánh chỉ số. Các lựa chọn của trình phân tích, các từ điển và các dạng nào của các thẻ token đánh chỉ số sẽ được cấu hình tìm kiếm văn bản được lựa chọn xác định (Phần 12.7). Có khả năng có nhiều cấu hình khác nhau trong cùng một cơ sở dữ liệu, và các cấu hình được xác định sẵn trước là sẵn sàng cho nhiều ngôn ngữ. Trong ví dụ của chúng ta, chúng ta đã sử dụng cấu hình mặc định english cho ngôn ngữ tiếng Anh.

Hàm setweight có thể được sử dụng để gắn nhãn cho các khoản đầu vào của một tsvector với một trọng số được đưa ra, nơi mà một trọng số là một trong các ký tự A, B, C, hoặc D. Điều này điển hình được sử dụng để đánh dấu các khoản đầu vào tới từ các phần khác nhau của một tài liệu, như tiêu đề so với thân. Sau đó, thông tin này có thể được sử dụng cho việc xếp hạng các kết quả tìm kiếm.

Vì to_tsvector(NULL) sẽ trả về NULL, được khuyến cáo sử dụng coalesce bất kỳ khi nào một trường có thể là null. Ở đây phương pháp được khuyến cáo cho việc tạo tsvector từ một tài liệu có cấu trúc:

```
UPDATE tt SET ti =
        setweight(to_tsvector(coalesce(title,'')), 'A') ||
        setweight(to_tsvector(coalesce(keyword,'')), 'B') ||
        setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
        setweight(to_tsvector(coalesce(body,'')), 'D');
```

Chúng ta ở đây đã sử dụng setweight để gắn nhãn cho nguồn của từng từ vị trong tsvector được kết thúc, và sau đó đã trộn các giá trị của tsvector được gắn nhãn bằng việc sử dụng toán tử ghép tsvector là ||. (Phần 12.4.1 đưa ra các chi tiết về các hoạt động đó).

12.3.2. Phân tích truy vấn

PostgreSQL đưa ra các hàm to_tsquery và plainto_tsquery cho việc biến đổi một truy vấn thành dạng dữ liệu tsquery. to_tsquery đưa ra sự truy cập tới nhiều chức năng hơn plainto_tsquery, nhưng ít tha thứ hơn về đầu vào của nó.

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

to_tsquery tạo ra một giá trị tsquery từ querytext, nó phải bao gồm các thẻ token duy nhất được các toán từ & (AND), | (OR) và ! (NOT) tách bạch ra. Các toán tử đó có thể được nhóm lại bằng việc sử dụng các dấu ngoặc đơn. Nói cách khác, đầu vào đối với to_tsquery phải tuân theo rồi các qui tắc chung đối với đầu vào tsquery, như được mô tả trong Phần 8.11. Sự khác biệt là trong khi đầu vào cơ bản tsquery lấy thẻ token ở giá pháp định (face value), thì to_tsquery bình thường hóa từng thẻ token thành một từ vị bằng việc sử dụng cấu hình được chỉ định hoặc mặc định, và hủy bỏ bất kỳ thẻ token nào mà là các từ chết theo cấu hình đó. Ví dụ:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
```

```
to_tsquery
---------------
'fat' & 'rat'
```

Như trong đầu vào cơ bản tsquery, (các) trọng số có thể được gắn vào từng từ vị để hạn chế nó trùng khớp chỉ tsvector các từ vị của (các) trọng số đó. Ví dụ:

```
SELECT to_tsquery('english', 'Fat | Rats:AB');

to_tsquery
------------------
'fat' | 'rat':AB
```

Hơn nữa, * cũng được gắn vào một từ vị để chỉ định việc khớp tiền tố:

```
SELECT to_tsquery('supern:*A & star:A*B');

to_tsquery
--------------------------
'supern':*A & 'star':*AB
```

Một từ vị như vậy sẽ khớp với bất kỳ từ nào trong một tsvector mà bắt đầu với chuỗi được đưa ra. to_tsquery cũng có thể chấp nhận các mệnh đề trong các dấu ngoặc đơn. Trước hết điều này là hữu dụng khi cấu hình đó bao gồm một từ điển các từ đồng nghĩa mà có thể làm bật ra các mệnh đề như vậy. Trong ví dụ bên dưới, một từ đồng nghĩa có chứa qui tắc supernovae stars : sn:

```
SELECT to_tsquery('''supernovae stars'' & !crab');

to_tsquery
---------------
'sn' & !'crab'
```

Không có các dấu ngoặc, to_tsquery sẽ tạo ra lỗi cú pháp đối với các thẻ token mà không được một toán tử AND hoặc OR tách biệt nhau.

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

plainto_tsquery biến đổi văn bản không được định dạng querytext sang tsquery. Văn bản đó được phân tích và bình thường hóa nhiều như đối với to_tsvector, sau đó toán tử & (AND) Boolean sẽ được chèn vào giữa các từ đang sống sót.

Ví dụ:

```
SELECT plainto_tsquery('english', 'The Fat Rats');

plainto_tsquery
-----------------
'fat' & 'rat'
```

Lưu ý rằng plainto_tsquery không thể nhận biết được các toán tử Boolean, các nhãn trọng số hoặc các nhãn khớp tiền tố ở đầu vào của nó:

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');

plainto_tsquery
---------------------
'fat' & 'rat' & 'c'
```

Ở đây, tất cả các dấu ngắt ở đầu vào từng bị bỏ qua như đang là các ký hiệu trắng.

12.3.3. Xếp hạng các kết quả tìm kiếm

Các cố gắng đo đếm việc xếp hạng các tài liệu phù hợp thế nào là đối với một truy vấn cụ thể, sao cho khi có nhiều sự trùng khớp thì các trùng khớp phù hợp nhất có thể được trình bày đầu tiên. PostgreSQL đưa ra 2 hàm xếp hạng được định nghĩa trước, chúng tính tới thông tin về từ vị, tính gần đúng và cấu trúc; đó là, chúng xem xét các khoản của truy vấn đó thường xuyên xuất hiện thế nào trong tài liệu, các khoản trong tài liệu đó gần nhau như thế nào, và quan trọng thế nào phần của tài liệu nơi mà chúng xảy ra. Tuy nhiên, khái niệm về tính phù hợp là mơ hồ và rất đặc thù ứng dụng. Các ứng dụng khác nhau có thể đòi hỏi thông tin bổ sung cho việc xếp hạng, như, thời gian sửa đổi tài liệu. Các hàm xếp hạng được xây dựng sẵn chỉ là những ví dụ. Bạn có thể tự mình viết các hàm xếp hạng và/hoặc các kết quả của chúng với các yếu tố bổ sung thêm cho khớp với các nhu cầu đặc thù.

2 hàm xếp hạng hiện sẵn có là:

ts_rank([weights float4[],] vector tsvector,
query tsquery [, normalization integer]) returns float4

Hàm xếp hạng tiêu chuẩn

ts_rank_cd([weights float4[],] vector tsvector,
query tsquery [, normalization integer]) returns float4

Hàm này tính toán xếp hạng mật độ bao trùm đối với vector và truy vấn tài liệu được đưa ra, như được mô tả trong "Xếp hạng phù hợp cho 1 tới 3 khoản truy vấn" của các tác giả Clarke, Cormack, và Tudhope trong tạp chí "Xử lý và Quản lý Thông tin", 1999.

Hàm này đòi hỏi thông tin vị trí ở đầu vào của nó. Vì thế nó sẽ không làm việc trong các giá trị "bị tước bỏ" của tsvector - nó sẽ luôn trả về 0.

Đối với các hàm đó, đối số tùy chọn weights đưa ra khả năng đánh trọng số các lần xuất hiện của từ nhiều hơn hoặc ít hơn, phụ thuộc nhiều vào cách mà chúng được gắn nhãn. Các mảng trọng số chỉ định cách đánh trọng số nặng thế nào cho từng chủng loại từ, theo trật tự:

{D-weight, C-weight, B-weight, A-weight}

Nếu không trọng số nào được đưa ra, thì các mặc định đó sẽ được sử dụng:

{0.1, 0.2, 0.4, 1.0}

Các trọng số điển hình được sử dụng để đánh dấu các từ từ các vùng đặc biệt của tài liệu, như tiêu đề hoặc một trích đoạn ban đầu, sao cho chúng có thể được đối xử với nhiều hoặc ít tầm quan trọng hơn các từ trong thân của tài liệu đó.

Vì một tài liệu dài hơn có một cơ hội lớn hơn trong việc có một khoản truy vấn là hợp lý để tính tới kích cỡ của tài liệu, như, một tài liệu hàng trăm từ với 5 lần xuất hiện của một từ tìm kiếm có thể là phù hợp hơn so với một tài liệu hàng ngàn từ với 5 lần xuất hiện. Các hàm xếp hạng lấy một lựa chọn normalization số nguyên chỉ định liệu và như thế nào độ dài của một tài liệu sẽ tác động tới sự xếp hạng của nó. Tùy chọn số nguyên sẽ kiểm soát vài hành vi, vì thế đó là một mặt nạ bit: bạn có thể chỉ định một hoặc nhiều hành vi bằng việc sử dụng | (ví dụ, 2|4).

- 0 (mặc định) bỏ qua độ dài tài liệu
- 1 chia xếp hạng cho 1 + logarit của chiều dài tài liệu
- 2 chia xếp hạng cho độ dài tài liệu
- 4 chia xếp hạng cho trung bình khoảng cách hài hòa giữa các mở rộng (điều này chỉ được triển khai với ts_rank_cd)
- 8 chia xếp hạng cho số các từ duy nhất trong tài liệu
- 16 chia xếp hạng cho 1 + logarit của số các từ duy nhất trong tài liệu
- 32 chia xếp hạng cho bản thân nó + 1

Nếu hơn một cờ bit được chỉ định, thì các biến đổi được áp dụng trong trật tự được liệt kê.

Là không quan trọng để lưu ý rằng các hàm xếp hạng không sử dụng bất kỳ thông tin tổng thể nào, nên không có khả năng để tạo ra một sự bình thường hóa công bằng tới 1% hoặc 100% như đôi khi được mong muốn. Lựa chọn bình thường hóa 32 (rank/(rank+1)) có thể được áp dụng để mở rộng phạm vi cho tất cả các xếp hạng trong dải từ 0 tới 1, nhưng tất nhiên điều này chỉ là một thay đổi nhỏ; nó sẽ không ảnh hưởng tới việc xếp thứ tự các kết quả tìm kiếm.

Ở đây là một ví dụ mà chỉ lựa chọn 10 sự trùng khớp được xếp hạng cao nhất:

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
```

```
title                                                                    | rank
-------------------------------------------------------------------------+----------
Neutrinos in the Sun                                                     | 3.1
The Sudbury Neutrino Detector                                            | 2.4
A MACHO View of Galactic Dark Matter                                     | 2.01317
Hot Gas and Dark Matter                                                  | 1.91171
The Virgo Cluster: Hot Plasma and Dark Matter                            | 1.90953
Rafting for Solar Neutrinos                                              | 1.9
NGC 4650A: Strange Galaxy and Dark Matter                                | 1.85774
Hot Gas and Dark Matter                                                  | 1.6123
Ice Fishing for Cosmic Neutrinos                                         | 1.6
Weak Lensing Distorts the Universe                                       | 0.818218
```

Đây là ví dụ y hệt bằng việc sử dụng xếp hạng được bình thường hóa:

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
```

```
title                                                                    | rank
-------------------------------------------------------------------------+-----------------
Neutrinos in the Sun                                                     | 0.756097569485493
The Sudbury Neutrino Detector                                            | 0.705882361190954
```

```
A MACHO View of Galactic Dark Matter                | 0.668123210574724
Hot Gas and Dark Matter                             | 0.65655958650282
The Virgo Cluster: Hot Plasma and Dark Matter       | 0.656301290640973
Rafting for Solar Neutrinos                         | 0.655172410958162
NGC 4650A: Strange Galaxy and Dark Matter           | 0.650072921219637
Hot Gas and Dark Matter                             | 0.617195790024749
Ice Fishing for Cosmic Neutrinos                    | 0.615384618911517
Weak Lensing Distorts the Universe                  | 0.450010798361481
```

Việc xếp hạng có thể là đắt giá vì nó đòi hỏi việc tư vấn tsvector đối với từng tài liệu trùng khớp, nó có thể là ràng buộc I/O và vì thế chậm. Không may, hầu như không có khả năng để tránh vì các truy vấn thực tiễn thường dẫn tới các số lượng trùng khớp lớn.

12.3.4. Nhấn mạnh các kết quả

Để thể hiện các kết quả, là lý tưởng để chỉ ra một phần của từng tài liệu và cách mà nó có liên quan tới truy vấn đó. Thường thì các máy tìm kiếm chỉ ra các đoạn tài liệu với các khoản tìm kiếm được đánh dấu. PostgreSQL đưa ra một hàm ts_headline, nó triển khai chức năng này.

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text ]) returns text
```

ts_headline chấp nhận một tài liệu đi với một truy vấn, và trả về một trích đoạn từ tài liệu trong đó các khoản từ truy vấn được nhấn mạnh. Cấu hình sẽ được sử dụng để phân tích tài liệu có thể được chỉ định bằng config; Nếu config bị làm mờ đi, thì cấu hình default_text_search_config được sử dụng.

Nếu một chuỗi options được chỉ định thì nó phải bao gồm một danh sách tách bạch nhau bằng dấu phẩy của một hoặc nhiều cặp option=value. Các lựa chọn sẵn sàng là:

- StartSel , StopSel: các chuỗi với chúng để bỏ hạn chế các từ truy vấn xuất hiện trong tài liệu, để phân biệt chúng với các từ được trích đoạn khác. Bạn phải đưa các chuỗi đó vào các dấu ngoặc kép nếu chúng có chứa các khoảng trống hoặc các dấu phẩy.
- MaxWords, MinWords: các số đó xác định các đầu đề dài nhất và ngắn nhất cho đầu ra.
- ShortWord: các từ có độ dài này hoặc ít hơn sẽ bị bỏ đi ở đầu và cuối của một đầu đề. Giá trị mặc định là 3 loại bỏ các bài tiếng Anh phổ biến.
- HighlightAll: cờ Boolean; nếu là true thì toàn bộ tài liệu sẽ được sử dụng như là đầu đề, bỏ qua 3 tham số đi đầu.
- MaxFragments: số lượng tối đa các trích đoạn hoặc đoạn văn bản được hiển thị. Giá trị mặc định là 0 sẽ lựa chọn một phương pháp tạo đầu đề hướng tới không có đoạn nào. Một giá trị lớn hơn 0 chọn sự tạo ra đầu đề dựa vào sự phân đoạn. Phương pháp này thấy các phân đoạn văn bản với càng nhiều từ truy vấn càng tốt và trải các đoạn đó xung quanh các từ truy vấn. Kết quả là các từ truy vấn nằm gần giữa của từng đoạn và có các từ nằm về các bên. Mỗi đoạn sẽ có hầu hết MaxWords và các từ độ dài ShortWord hoặc nhỏ hơn sẽ bị bỏ đi ở đầu và cuối của từng đoạn. Nếu không phải tất cả các từ truy vấn được thấy trong tài liệu, thì một đoạn duy nhất của MinWords đầu tiên trong tài liệu sẽ được hiển thị.

- FragmentDelimiter: Khi nhiều hơn 1 đoạn được hiển thị, thì các đoạn sẽ được cách nhau bằng chuỗi này.

Bất kỳ lựa chọn không được chỉ định nào cũng nhận các mặc định đó:

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

Ví dụ:

```
SELECT ts_headline('english',
        'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
to_tsquery('query & similarity'));

ts_headline
------------------------------------------------------------
containing given <b>query</b> terms
and return them in order of their <b>similarity</b> to the
<b>query</b>.

SELECT ts_headline('english',
'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
to_tsquery('query & similarity'),
'StartSel = <, StopSel = >');

ts_headline
-------------------------------------------------------
containing given <query> terms
and return them in order of their <similarity> to the
<query>.
```

ts_headline sử dụng tài liệu gốc ban đầu, chứ không phải một tóm tắt tsvector, nên nó có thể là chậm và nên được sử dụng thận trọng. Một sai sót điển hình là gọi ts_headline cho từng tài liệu trùng khớp khi chỉ 10 tài liệu sẽ được hiển thị. Các truy vấn phụ SQL có thể giúp: đây là một ví dụ:

```
SELECT id, ts_headline(body, q), rank
FROM (SELECT id, body, q, ts_rank_cd(ti, q) AS rank
        FROM apod, to_tsquery('stars') q
        WHERE ti @@ q
        ORDER BY rank DESC
        LIMIT 10) AS foo;
```

12.4. Tính năng bổ sung

Phần này mô tả các hàm và toán tử bổ sung mà là hữu dụng trong sự kết nối với tìm kiếm văn bản.

12.4.1. Điều khiển tài liệu

Phần 12.3.1 đã chỉ ra cách mà các tài liệu văn bản thô có thể được biến đổi thành các giá trị tsvector.

PostgreSQL cũng đưa ra các hàm và toán tử có thể được sử dụng để điều khiển các tài liệu mà ở trong dạng tsvector rồi.

tsvector || tsvector

Toán tử ghép nối tsvector trả về một vector kết nối các từ vị và thông tin vị trí của 2 vector được đưa ra như là các đối số. Các vị trí và các nhãn trọng số được giữ lại trong quá trình ghép nối. Các vị trí xuất hiện trong vector bên tay phải được bù trừ bằng vị trí rộng lớn nhất được nhắc tới trong vector bên tay trái, sao cho kết quả là gần tương đương với kết quả của việc thực hiện to_tsvector trong sự ghép nối của 2 chuỗi trong tài liệu ban đầu. (Sự ngang bằng đó là không chính xác, vì bất kỳ các từ chết nào bị loại bỏ khỏi cuối của đối số bên tay trái cũng sẽ không ảnh hưởng tới kết quả, trong khi chúng có thể đã ảnh hưởng tới các vị trí của các từ vị trong đối số bên tay phải nếu sự ghép nối văn bản đã được sử dụng).

Một ưu điểm của việc sử dụng ghép nối ở dạng vector, thay vì việc ghép nối văn bản trước khi áp dụng to_tsvector, là bạn có thể sử dụng các cấu hình khác nhau để phân tích các phần khác nhau của tài liệu. Hơn nữa, vì hàm setweight đánh dấu tất cả các từ vị của vector được đưa ra theo cùng cách y hệt, là cần thiết để phân tích văn bản và thực hiện setweight trước việc ghép nối nếu bạn muốn gắn nhãn cho các phần khác nhau của tài liệu bằng các trọng số khác nhau.

setweight(vector tsvector, weight "char") returns tsvector

setweight trả về một bảo sao của vector đầu vào trong đó mỗi vị trí đã được gắn nhãn với weight được đưa ra, hoặc A, B, C, hoặc D. (D là mặc định cho các vector mới và như vậy là không được hiển thị ở đầu ra). Các nhãn đó được giữ lại khi các vector được ghép nối, cho phép các từ từ các phần khác nhau của một tài liệu sẽ được đánh trọng số khác nhau bằng việc xếp hạng các hàm.

Lưu ý rằng các nhãn trọng số áp dụng cho các vị trí, không cho các từ vị. Nếu vector đầu vào từng bị tước mất các vị trí thì setweight không làm gì cả.

length(vector tsvector) returns integer

Trả về số từ vị được lưu trữ trong vector đó.

strip(vector tsvector) returns tsvector

Trả về một vector liệt kê các từ vị y hệt như vector được đưa ra, nhưng thiếu bất kỳ vị trí hoặc thông tin trọng số nào. Trong khi vector được trả về là ít hữu dụng hơn nhiều so với một vector không bị tước bỏ về xếp hạng tương ứng, thì nó thường sẽ là nhỏ hơn nhiều.

12.4.2. Điều khiển truy vấn

Phần 12.3.2 đã chỉ ra cách mà các truy vấn thô có thể được biến đổi thành các giá trị tsquery. PostgreSQL cũng đưa ra các hàm và toán tử có thể được sử dụng để điều khiển các truy vấn nằm ở dạng tsquery rồi.

tsquery && tsquery

Trả về sự kết hợp AND của 2 truy vấn được đưa ra.

tsquery || tsquery

Trả về sự kết hợp OR của 2 truy vấn được đưa ra.

!! tsquery

Trả về phủ định (NOT) của truy vấn được đưa ra.

numnode(query tsquery) returns integer

Trả về số nút (các từ vị cộng với các toán tử) trong một tsquery. Hàm này là hữu dụng để xác định liệu query đó có ý nghĩa (trả về > 0), hay chỉ bao gồm các từ chết (trả về 0).

Các ví dụ:

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE: query contains only stopword(s) or doesn't contain lexeme(s), ignored

numnode
---------
0


SELECT numnode('foo & bar'::tsquery);

numnode
---------
3
```

querytree(query tsquery) returns text

Trả về vị trí của một tsquery mà có thể được sử dụng cho việc tìm kiếm một chỉ số. Hàm này là hữu dụng cho việc dò tìm các truy vấn không được đánh chỉ số, ví dụ các truy vấn chỉ chứa các từ chết hoặc chỉ các khoản phủ định. Ví dụ:

```
SELECT querytree(to_tsquery('!defined'));

querytree
-----------
```

12.4.2.1. Viết truy vấn

Họ các hàm ts_rewrite tìm kiếm một tsquery được đưa ra cho các lần xuất hiện của một truy vấn con đích, và thay thế từng lần xuất hiện bằng một truy vấn con thay thế. Về cơ bản hoạt động này là một phiên bản đặc biệt của tsquery đối với sự thay thế các chuỗi con. Một sự kết hợp đích và thay thế có thể được nghĩ như một quy tắc viết lại truy vấn. Một tập hợp các qui tắc viết lại như vậy có thể là một sự trợ giúp tìm kiếm mạnh. Ví dụ, bạn có thể mở rộng sự tìm kiếm bằng việc sử dụng các từ đồng nghĩa (như, new york, big apple, nyc, gotham) hoặc làm hẹp lại tìm kiếm đó để hướng người sử dụng vào một số chủ đề nóng. Có một số sự chồng lấn trng chức năng giữa tính năng này và các từ điển từ đồng nghĩa (Phần 12.6.4). Tuy nhiên, bạn có thể sửa một tập hợp các qui định viết lại khi làm việc mà không phải đánh chỉ số lại, trong khi việc cập nhật một từ điển từ đồng nghĩa đòi hỏi việc đánh chỉ số lại phải được thực hiện.

ts_rewrite (query tsquery, target tsquery, substitute tsquery) returns tsquery

Dạng này của ts_rewrite đơn giản áp dụng một qui tắc viết lại duy nhất: target được thay thế bằng substitute bất kể khi nào nó xuất hiện trong query. Ví dụ:

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);

ts_rewrite
------------
'b' & 'c'
```

ts_rewrite (query tsquery, select text) returns tsquery

Dạng ts_rewrite này chấp nhận một sự khởi đầu query và một lệnh SQL select, nó được đưa ra như một chuỗi văn bản. select phải có 2 cột dạng tsquery. Đối với từng hàng của kết quả select, các lần xuất hiện của giá trị cột đầu tiên (đích) được thay thế bằng giá trị cột thứ 2 (thay thế) trong giá trị query hiện hành. Ví dụ:

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');

ts_rewrite
------------
'b' & 'c'
```

Lưu ý rằng khi nhiều qui tắc viết lại được áp dụng theo cách này, thì trật tự ứng dụng có thể là quan trọng; nên trong thực tế bạn sẽ muốn truy vấn nguồn đối với ORDER BY một vài khóa xếp thứ tự.

Hãy xem xét một ví dụ khám phá vũ trụ cuộc sống thực. Chúng tôi sẽ mở rộng truy vấn supernovae bằng việc sử dụng các qui tắc viết lại do bảng dẫn dắt:

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));
SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');

ts_rewrite
---------------------------------
'crab' & ( 'supernova' | 'sn' )
```

Chúng ta có thể thay đổi các qui định viết lại chỉ bằng việc cập nhật bảng đó:

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');

ts_rewrite
---------------------------------------------
'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

Việc viết lại có thể là chậm khi có nhiều qui tắc viết lại, vì nó kiểm tra từng qui tắc cho sự trùng khớp có khả năng. Để lọc ra các qui tắc không phải là ứng viên rõ ràng thì chúng ta có thể sử dụng

các toán tử ghép nối cho dạng tsquery. Trong ví dụ bên dưới, chúng ta chỉ chọn các qui tắc nào có thể trùng khớp với truy vấn ban đầu:

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE "a & b"::tsquery @> t');
ts_rewrite
------------
'b' & 'c'
```

12.4.3. Trigger cho cập nhật tự động

Khi sử dụng một cột riêng rẽ để lưu trữ đại diện tsvector các tài liệu của bạn, cần thiết phải tạo ra một trigger để cập nhật cột tsvector khi thay đổi các cột nội dung tài liệu. 2 hàm trigger được xây dựng sẵn là có sẵn cho điều này, hoặc bạn có thể viết cho riêng bạn.

```
tsvector_update_trigger(tsvector_column_name, config_name, text_column_name [, ... ])
tsvector_update_trigger_column(tsvector_column_name, config_column_name, text_column_name [, ...
```

Các hàm trigger đó tự động tính toán một cột tsvector từ một hoặc nhiều cột văn bản, dưới sự kiểm soát các tham số được chỉ định trong lệnh CREATE TRIGGER. Một ví dụ sử dụng chúng là:

```
CREATE TABLE messages (
        title text,
        body text,
        tsv tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);
INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;

title               | body                      | tsv
--------------------+---------------------------+-------------------------------
title here          | the body text is here  | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');

title               | body
--------------------+---------------------------
title here          | the body text is here
```

Khi đã tạo ra được trigger này thì bất kỳ sự thay đổi nào trong title hoặc body sẽ tự động được phản ánh trong tsv, mà ứng dụng không phải lo về nó nữa.

Đối số đầu tiên của trigger phải là tên của cột tsvector sẽ được cập nhật. Đối số thứ 2 chỉ định cấu hình tìm kiếm văn bản sẽ được sử dụng để thực hiện sự biến đổi. Đối với tsvector_update_trigger, tên cấu hình đơn giản được đưa ra như là đối số thứ 2 của trigger. Nó phải đủ điều kiện theo sơ đồ như được nêu ở trên, sao cho hành vi của trigger đó sẽ không thay đổi với các thay đổi trong search_path. Đối với tsvector_update_trigger_column, đối số thứ 2 của trigger là tên của cột khác của bảng, nó phải ở dạng regconfig. Điều này cho phép một lựa chọn cấu hình theo từng hàng được thực hiện. (Các) đối số còn lại là các tên của các cột văn bản (dạng text, varchar hoặc char). Chúng sẽ

được đưa vào trong tài liệu theo trật tự được đưa ra. Các giá trị NULL sẽ được bỏ qua (nhưng các cột khác vẫn sẽ được đánh chỉ số).

Một hạn chế của các trigger được xây dựng sẵn đó là chúng đối xử với tất cả các cột đầu vào như nhau. Để xử lý các cột một cách khác nhau - ví dụ, để đánh trọng số tiêu đề khác nhau đối với thân - cần thiết phải viết một trigger tùy biến. Đây là một ví dụ bằng việc sử dụng PL/pgSQL như là ngôn ngữ của trigger đó:

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
    new.tsv :=
        setweight(to_tsvector('pg_catalog.english', coalesce(new.title,'')), 'A') ||
        setweight(to_tsvector('pg_catalog.english', coalesce(new.body,'')), 'D');
    return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
        ON messages FOR EACH ROW EXECUTE PROCEDURE messages_trigger();
```

Giữ trong đầu rằng điều quan trọng để chỉ định tên cấu hình rõ ràng khi tạo các giá trị tsvector bên trong các trigger, sao cho các nội dung cột sẽ không bị ảnh hưởng vì những thay đổi đối với default_text_search_config. Không làm được điều này có khả năng dẫn tới các vấn đề như các kết quả tìm kiếm thay đổi sau một sự hỏng và tải lại.

12.4.4. Thu thập thống kê tài liệu

Hàm ts_stat là hữu dụng cho việc kiểm tra cấu hình của bạn và cho việc tìm kiếm các ứng viên là các từ chết:

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

sqlquery là giá trị văn bản bao gồm một truy vấn SQL mà nó phải trả về một cột tsvector duy nhất. ts_stat thực thi truy vấn và trả về các thống kê về từng từ vị (từ) khác nhau có trong các dữ liệu tsvector. Các cột được trả về là:

- word text - giá trị của một từ vị
- ndoc integer - số các tài liệu (tsvectors) mà từ đó xuất hiện trong
- nentry integer - tổng số các lần xuất hiện của từ đó

Nếu weights được cung cấp, thì chỉ các lần xuất hiện có một trong các trọng số đó được tính.

Ví dụ, để tìm kiếm 10 từ thường xuyên nhất trong một bộ các tài liệu:

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

Y hệt, nhưng chỉ tính các trường hợp từ với trọng số A hoặc B:

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

12.5. Trình phân tích

Các trình phân tích tìm kiếm văn bản có trách nhiệm cho việc tách các văn bản tài liệu thô thành các *thẻ token* và việc nhận diện từng dạng thẻ token nơi mà tập hợp các dạng có thể được bản thân trình phân tích đó xác định. Lưu ý là một trình phân tích hoàn toàn không sửa văn bản - nó đơn giản nhận diện các biên giới từ thật đúng. Vì phạm vi có giới hạn này, có ít nhu cầu cho các trình phân tích tùy biến đặc thù ứng dụng hơn là có các từ điển tùy biến. Hiện tại PostgreSQL chỉ đưa ra một trình phân tích được xây dựng sẵn, nó từng được thấy là hữu dụng cho một dải rộng lớn các ứng dụng.

Trình phân tích được xây dựng sẵn được đặt tên là `pg_catalog.default`. Nó nhận thức được 23 dạng thẻ token:

Bảng 12-1. Các dạng thẻ token của các trình phân tích mặc định

Tên hiệu (Alias)	Mô tả	Ví dụ
asciiword	Từ, tất cả các ký tự ASCII	elephant
word	Từ, tất cả các ký tự	mañana
numword	Từ, các ký tự và chữ số	beta1
asciihword	Từ nối, tất cả ASCII	up-to-date
hword	Từ nối, tất cả các ký tự	lógico-matemática
numhword	Từ nối, các ký tự và chữ số	postgresql-beta1
hword_asciipart	Phần của từ nối, tất cả ASCII	postgresql in the context postgresql-beta1
hword_part	Phần của từ nối, tất cả các ký tự	lógico or matemática in the context lógico-matemática
hword_numpart	Phần của từ nối, các ký tự và chữ số	beta1 in the context postgresql-beta1
email	Địa chỉ thư điện tử	foo@example.com
protocol	Đầu đề của giao thức	http://
url	URL	example.com/stuff/index.html
host	Host	example.com
url_path	Đường dẫn URL	/stuff/index.html, in the context of a URL
file	Tệp hoặc tên đường dẫn	/usr/local/foo.txt, if not within a URL
sfloat	Ký hiệu khoa học	-1.234e56
float	Ký hiệu thập phân	-1.234
int	Số nguyên được ký	-1234
uint	Số nguyên không được ký	1234
version	Số phiên bản	8.3.0
tag	Thẻ XML	<a href="dictionaries.html">
entity	Thực thể XML	&

Lưu ý: Ký hiệu của trình phân tích của một "ký tự" được thiết lập bản địa của cơ sở dữ liệu xác định, đặc biệt lc_ctype. Các từ chỉ chứa các ký tự ASCII cơ bản được nêu như một dạng thẻ token riêng rẽ, vì đôi khi là hữu dụng để phân biệt chúng. Trong hầu hết các ngôn ngữ châu Âu, các dạng thẻ token word và asciiword sẽ được đối xử như nhau. email không hỗ trợ tất cả các ký tự thư điện tử hợp lệ như được RFC 5322 định nghĩa. Đặc biệt, chỉ các ký tự không phải abc được hỗ trợ cho các tên người sử dụng thư điện tử là dấu chấm, dấu gạch ngang và dấu gạch chân.

Có khả năng đối với trình phân tích để tạo ra các thẻ token chồng lấn nhau từ cùng các mẩu văn bản. Như một ví dụ, một từ nối sẽ được nêu cả như toàn bộ từ và như từng thành phần:

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
alias          | description                                 | token
---------------+---------------------------------------------+--------------------
numhword       | Hyphenated word, letters and digits         | foo-bar-beta1
hword_asciipart | Hyphenated word part, all ASCII            | foo
blank          | Space symbols                               | -
hword_asciipart | Hyphenated word part, all ASCII            | bar
blank          | Space symbols                               | -
hword_numpart  | Hyphenated word part, letters and digits    | beta1
```

Hành vi này là mong muốn vì nó cho phép các tìm kiếm làm việc cho cả toàn bộ từ tổ hợp và cho các thành phần. Đây là ví dụ trực quan khác:

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
alias          | description     | token
---------------+-----------------+------------------------------------
protocol       | Protocol head   | http://
url            | URL             | example.com/stuff/index.html
host           | Host            | example.com
url_path       | URL path        | /stuff/index.html
```

12.6. Từ điển

Các từ điển được sử dụng để loại bỏ các từ sẽ không được xem xét trong một tìm kiếm (*các từ chết*), và để *bình thường hóa* các từ sao cho các dạng dẫn xuất khác nhau của từ y hệt sẽ khớp. Một từ được bình thường hóa thành công được gọi là một từ vị. Ngoài việc cải thiện chất lượng tìm kiếm, sự bình thường hóa và loại bỏ các từ chết làm giảm kích cỡ của đại diện tsvector của một tài liệu, vì thế cải thiện được hiệu năng. Bình thường hóa không luôn có ý nghĩa về ngôn ngữ học và thường phụ thuộc vào ngữ nghĩa của ứng dụng.

Một số ví dụ của sự bình thường hóa:

- Theo ngôn ngữ học - các từ điển Ispell cố làm giảm các từ đầu vào tới một dạng được bình thường hóa; các từ điển cọng (stemmer) loại bỏ các đuôi từ
- Các vị trí URL có thể được kinh điển hóa để làm cho các URL tương đương khớp:
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html

- Các tên màu có thể được các giá trị hệ 16 (hexadecimal) thay thế, như, red, green, blue, magenta → FF0000, 00FF00, 0000FF, FF00FF
- Nếu đánh chỉ số các số, thì chúng ta có thể loại bỏ vài chữ số thập phân để giảm dải các số có khả năng, ví dụ 3.14159265359, 3.1415926, 3.14 sẽ là hệt nhau sau sự bình thường hóa nếu chỉ 2 chữ số được giữ lại sau dấu thập phân.

Một từ điển là một chương trình chấp nhận một thẻ token như là đầu vào và trả về:

- một mảng các từ vị nếu thẻ token đầu vào được biết đối với từ điển đó (lưu ý rằng một thẻ token có thể tạo ra nhiều hơn 1 từ vị)
- một từ vị duy nhất với cờ TSL_FILTER được thiết lập, để thay thế thẻ token gốc bằng một thẻ token sẽ được thông qua tới các từ điển sau đó (một từ điển làm điều này được gọi là một từ điển lọc)
- một mảng rỗng nếu từ điển biết thẻ token đó, nhưng đó là một từ chết
- NULL nếu từ điển không nhận biết được thẻ token đầu vào

PostgreSQL đưa ra các từ điển được xác định trước cho nhiều ngôn ngữ. Cũng có vài mẫu template được xác định trước có thể được sử dụng để tạo ra các từ điển mới với các tham số tùy biến. Mỗi mẫu template từ điển được xác định trước được mô tả bên dưới. Nếu không mẫu template đang tồn tại nào là phù hợp, thì có khả năng phải tạo ra các mẫu mới; xem vùng contrib/ của phát tán PostgreSQL để có các ví dụ.

Một cấu hình tìm kiếm văn bản ràng buộc một trình phân tích cùng với một tập hợp các từ điển để xử lý các thẻ token đầu ra của trình phân tích. Đối với từng dạng thẻ token mà trình phân tích đó có thể trả về, một danh sách riêng rẽ các từ điển được cấu hình đó chỉ định. Khi một thẻ token của dạng đó được trình phân tích đó tìm thấy, thì từng từ điển trong danh sách đó được tư vấn lần lượt, cho tới khi một số từ điển nhận biết được nó như một từ biết rồi. Nếu nó được nhận diện như là một từ chết, hoặc nếu không từ điển nào nhận biết được thẻ token đó, thì nó sẽ bị loại bỏ và không được đánh chỉ số hoặc không được tìm kiếm.

Thông thường, từ điển đầu tiên trả về một đầu ra không NULL sẽ xác định kết quả, và bất kỳ từ điển nào còn lại cũng sẽ không được tư vấn; nhưng một việc lọc từ điển có thể thay thế từ được đưa ra bằng một từ được sửa đổi, nó sau đó được thông qua tới các từ điển tiếp sau.

Qui tắc chung cho việc thiết lập cấu hình một danh sách các từ điển là đặt lên trước từ điển hẹp nhất, đặc thù nhất, sau đó là các từ điển phổ biến hơn, kết thúc bằng một từ điển rất chung, như một cọng bông tuyết (Snowball stemmer) hoặc simple, nó nhận biết được mọi điều. Ví dụ, đối với một tìm kiếm đặc thù thiên văn học (cấu hình astro_en) thì một từ điển có thể ràng buộc dạng thẻ token asciiword (từ ASCII) cho một từ điển đồng nghĩa đối với các khái niệm về thiên văn học, một từ điển chung tiếng Anh và một từ điển dạng cọng bông tuyết tiếng Anh:

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

Một từ điển lọc có thể được thay thế ở bất kỳ đâu trong danh sách, ngoại trừ ở cuối nơi mà nó có thể là vô dụng. Việc lọc các từ điển là hữu dụng để bình thường hóa một phần các từ để đơn giản hóa tác vụ của các từ điển sau đó. Ví dụ, một từ điển lọc có thể được sử dụng để loại bỏ các dấu khỏi các ký tự có dấu, như được thực hiện bằng module mở rộng contrib/unaccent.

12.6.1. Từ chết

Các từ chết là các từ rất phổ biến, xuất hiện trong hầu hết từng tài liệu, và không có giá trị phân biệt. Vì thế, chúng có thể bị bỏ qua trong ngữ cảnh của tìm kiếm toàn văn. Ví dụ, mọi văn bản tiếng Anh đều có các từ như a và the, nên là vô dụng để lưu trữ chúng trong một chỉ số. Tuy nhiên, các từ chết ảnh hưởng tới các vị trí trong tsvector, tới lượt nó ảnh hưởng tới việc xếp hạng:

```
SELECT to_tsvector('english','in the list of stop words');

to_tsvector
----------------------------
'list':3 'stop':5 'word':6
```

Các vị trí bị mất 1, 2, 4 là vì các từ chết. Các xếp hạng được tính toán cho các tài liệu với và không với các từ chết là hoàn toàn khác nhau:

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'), to_tsquery('list & ts_rank_cd
------------
0.05

SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list & stop'));
ts_rank_cd
------------
0.1
```

Tùy vào từ điển đặc thù ứng xử thế nào với các từ chết. Ví dụ, các từ điển ispell trước hết bình thường hóa các từ và sau đó xem xét danh sách các từ chết, trong khi các từ điển cọng bông tuyết trước hết kiểm tra danh sách các từ chết. Lý do hành xử khác nhau này là ý định làm giảm nhiễu.

12.6.2. Từ điển đơn giản

Mẫu template từ điển simple vận hành bằng việc biến đổi thẻ đầu vào sang chữ thường và kiểm tra nó đối với một tệp các từ chết. Nếu nó được thấy trong tệp đó thì sau đó một mảng rỗng được trả về, làm cho thẻ token đó sẽ bị hủy bỏ. Nếu không, dạng chữ thường của từ đó được trả về như là từ vị được bình thường hóa. Như một lựa chọn, từ điển đó có thể được thiết lập cấu hình để nêu các từ không chết như không được nhận biết, cho phép chúng được truyền qua tới từ điển tiếp sau trong danh sách.

Đây là một ví dụ định nghĩa một từ điển bằng việc sử dụng mẫu template simple:

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
        TEMPLATE = pg_catalog.simple,
        STOPWORDS = english
);
```

Ở đây, english là tên cơ bản của một tệp các từ chết. Tên đầy đủ của tệp sẽ là $SHAREDIR/tsearch_data/english.stop, trong đó $SHAREDIR có nghĩa là thư mục dữ liệu được chia sẻ

của cài đặt PostgreSQL đó, thường là /usr/local/share/postgresql (sử dụng pg_config--sharedir để xác định nó nếu bạn không chắc chắn). Định dạng tệp đó đơn giản là một danh sách các từ, từng dòng một. Các dòng trống và các không gian trống theo sau sẽ bị bỏ qua, và chữ hoa được biến đổi thành chữ thường, nhưng không xử lý nào khác được thực hiện trong các nội dung tệp đó.

Bây giờ chúng ta có thể kiểm thử thư mục của chúng ta:

```
SELECT ts_lexize('public.simple_dict','YeS');

ts_lexize
-----------
{yes}
SELECT ts_lexize('public.simple_dict','The');

ts_lexize
-----------
{}
```

Chúng ta cũng có thể chọn trả về NULL, thay vì một từ với chữ thường, nếu nó được thấy trong tệp các từ chết. Hành vi này được chọn bằng việc thiết lập tham số Accept của từ điển đó về false.

Tiếp tục ví dụ:

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );
SELECT ts_lexize('public.simple_dict','YeS');
ts_lexize
-----------
SELECT ts_lexize('public.simple_dict','The');
ts_lexize
-----------
{}
```

Với thiết lập mặc định của Accept = true, chỉ hữu dụng để đặt một từ điển simple ở cuối của một danh sách các từ điển, vì nó sẽ không bao giờ truyền qua bất kỳ thẻ token nào tới một từ điển tiếp sau được. Ngược lại, Accept = false chỉ hữu dụng khi có ít nhất một từ điển đứng đằng sau.

Chú ý

Hầu hết các dạng từ điển đều dựa vào các tệp cấu hình, như các tệp các từ chết. Các tệp đó phải được lưu trữ ở mã UTF-8. Chúng sẽ được dịch sang mã cơ sở dữ liệu thực tế, nếu điều đó là khác nhau, khi chúng được đọc trong máy chủ.

Chú ý

Thông thường, một phiên cơ sở dữ liệu sẽ đọc tệp cấu hình của một thư mục chỉ một lần, khi nó lần đầu tiên được sử dụng trong phiên đó. Nếu bạn sửa đổi một tệp cấu hình và muốn ép các phiên đang tồn tại chọn các nội dung mới đó, hãy đưa ra một lệnh ALTER TEXT SEARCH DICTIONARY trong từ điển đó. Điều này có thể là một cập nhật "giả tạo" thực sự không làm thay đổi bất kỳ giá trị tham số nào.

12.6.3. Từ điển từ đồng nghĩa

Mẫu template từ điển này được sử dụng để tạo ra các từ điển thay thế cho một từ bằng một từ đồng nghĩa. Các cụm từ không được hỗ trợ (sử dụng mẫu template từ điển đồng nghĩa (Phần 12.6.4) cho điều đó). Một từ điển các từ đồng nghĩa có thể được sử dụng để khắc phục các vấn đề về ngôn ngữ, ví dụ, để ngăn ngừa một từ điển cọng tiếng Anh khỏi việc làm giảm từ 'Paris' thành 'pari'. Đủ để có một dòng Paris paris trong từ điển đồng nghĩa đó và đặt nó trước từ điển english_stem. Ví dụ:

```
SELECT * FROM ts_debug('english', 'Paris');
alias              | description     | token | dictionaries       | dictionary      | lexemes
-------------------+-----------------+-------+--------------------+-----------------+-------------
asciiword          | Word, all ASCII | Paris | {english_stem}     | english_stem    | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
        TEMPLATE = synonym,
        SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
        ALTER MAPPING FOR asciiword
        WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');

alias              | description     | token | dictionaries                        | dictionary      | lexemes
-------------------+-----------------+-------+-------------------------------------+-----------------+------------
asciiword          | Word, all ASCII | Paris | {my_synonym,english_stem}           | my_synonym      | {paris}
```

Tham số duy nhất được mẫu template từ đồng nghĩa yêu cầu là SYNONYMS, nó là tên cơ bản của tệp cấu hình của nó - my_synonyms trong ví dụ ở trên. Tên đầy đủ của tệp đó sẽ là $SHAREDIR/tsearch_data/my_synonyms.syn (trong đó $SHAREDIR có nghĩa là thư mục dữ liệu chia sẻ của cài đặt PostgreSQL). Định dạng tệp đó chỉ là một từ mỗi dòng sẽ được thay thế, bằng từ đồng nghĩa của nó theo sau, được tách bạch nhau bằng một dấu trắng. Các dòng trống và các dấu trắng đi theo sau sẽ bị bỏ qua.

Mẫu template synonym cũng có một tham số tùy ý CaseSensitive, nó mặc định là sai false. Khi CaseSensitive là false, các từ trong tệp đồng nghĩa được trả về chữ thường, như các thẻ token đầu vào. Khi nó là đúng true, thì các từ và các thẻ token không trả về chữ thường, mà sẽ được so sánh như chúng có.

Một dấu sao (*) có thể được thay thế ở cuối của một từ đồng nghĩa trong tệp cấu hình. Điều này chỉ rằng từ đồng nghĩa là một tiền tố. Dấu * được bỏ qua khi khoản đầu vào được sử dụng trong to_tsvector(), như khi nó được sử dụng trong to_tsquery(), thì kết quả sẽ là một khoản của một truy vấn với con đánh dấu tiền tố trùng khớp (xem Phần 12.3.2). Ví dụ, giả thiết chúng ta có các khoản đầu vào đó trong $SHAREDIR/tsearch_data/synonym_sample.syn:

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle googl
indices index*
```

Rồi chúng ta sẽ có các kết quả này:

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn','indices');

ts_lexize
-----------
{index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst','indices');
to_tsvector
-------------
'index':1
(1 row)

mydb=# SELECT to_tsquery('tst','indices');

to_tsquery
------------
'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;

tsvector
---------------------------------
'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst','indices');

?column?
----------
t
(1 row)
```

12.6.4. Từ điển từ đồng nghĩa

Một từ điển từ đồng nghĩa (đôi khi viết tắt như là TZ) là một tập hợp các từ mà bao gồm thông tin về các mối quan hệ của các từ hoặc các cụm từ, như, các khoản rộng lớn hơn – BT (broader terms), các khoản thu hẹp hơn – NT (narrower terms), các khoản được ưu tiên, các khoản không được ưu tiên, các khoản có liên quan, …

Về cơ bản một từ điển từ đồng nghĩa thay thế cho tất cả các khoản không được ưu tiên bằng một khoản được ưu tiên và, như một tùy chọn, cũng giữ lại các khoản gốc cho việc đánh chỉ số. Triển khai hiện hành của PostgreSQL từ điển từ đồng nghĩa là một sự mở rộng của từ điển đồng nghĩa với sự hỗ trợ các cụm từ được bổ sung thêm vào. Một từ điển từ đồng nghĩa đòi hỏi một tệp cấu hình có định dạng sau:

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

trong đó dấu hai chấm (:) hành động như một dấu ngắt giữa một cụm từ và sự thay thế của nó.

Một từ điển từ đồng nghĩa sử dụng một từ điển con (nó được chỉ định trong cấu hình của từ điển đó) để bình thường hóa văn bản đầu vào trước khi kiểm tra các cụm từ trùng khớp. Chỉ có khả năng để lựa chọn một từ điển con. Một lỗi được nêu nếu từ điển con không nhận biết được một từ. Trong trường hợp đó, bạn nên loại bỏ sử dụng từ đó hoặc dạy cho từ điển con đó về từ đó. Bạn có thể thay thế một dấu sao (*) ở đầu của một từ được đánh chỉ số để bỏ qua việc áp dụng từ điển con đó với nó, nhưng tất cả các từ mẫu phải được biết đối với từ điển con đó.

Từ điển từ đồng nghĩa chọn sự trùng khớp dài nhất nếu có nhiều cụm từ khớp với đầu vào, và các mối liên hệ sẽ bị vỡ bằng việc sử dụng định nghĩa cuối cùng.

Các từ chết đặc biệt được từ điển con thừa nhận không thể được chỉ định; thay vào đó hãy sử dụng dấu hỏi (?) để đánh dấu vị trí nơi mà bất kỳ từ chết nào có thể xuất hiện. Ví dụ, giả thiết rằng a và the là các từ chết theo từ điển con đó.

```
? one ? two : swsw
```

khớp với a one the two và the one a two; cả 2 có thể được thay thế bằng swsw.

Vì một từ điển các từ đồng nghĩa có khả năng nhận biết được các cụm từ mà nó phải nhớ tình trạng của nó và tương tác với trình phân tích. Một từ điển từ đồng nghĩa sử dụng các chỉ định đó để kiểm tra liệu nó có nên điều khiển từ tiếp sau hay dừng sự tích lũy. Từ điển từ đồng nghĩa phải được thiết lập cấu hình một cách thận trọng. Ví dụ, nếu từ điển từ đồng nghĩa được chỉ định để điều khiển chỉ thẻ token asciiword, thì một định nghĩa từ điển từ đồng nghĩa giống như one 7 sẽ không làm việc vì dạng thẻ token uint không được chỉ định cho từ điển từ đồng nghĩa đó.

> **Chú ý**
>
> Các từ điển đồng nghĩa được sử dụng trong quá trình đánh chỉ số nên bất kỳ sự thay đổi nào trong các tham số từ điển từ đồng nghĩa đó đòi hỏi việc đánh chỉ số lại. Đối với hầu hết các dạng từ điển khác, những thay đổi nhỏ như việc thêm hoặc bớt các từ chết không buộc phải đánh chỉ số lại.

12.6.4.1. Cấu hình từ điển từ đồng nghĩa

Để định nghĩa một từ điển từ đồng nghĩa, hãy sử dụng mẫu template thesaurus. Ví dụ:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
        TEMPLATE = thesaurus,
        DictFile = mythesaurus,
        Dictionary = pg_catalog.english_stem
);
```

Ở đây:

- thesaurus_simple là tên từ điển mới

- mythesaurus là tên cơ bản của tệp cấu hình từ điển đồng nghĩa. (Tên đầy đủ của nó sẽ là $SHAREDIR/tsearch_data/mythesaurus.ths, trong đó $SHAREDIR có nghĩa là thư mục dữ liệu được chia sẻ của cài đặt đó).
- pg_catalog.english_stem là từ điển con (ở đây, một cọng bông tuyết tiếng Anh) sẽ sử dụng cho sự bình thường hóa của từ điển đồng nghĩa. Lưu ý rằng từ điển con đó sẽ có cấu hình riêng của nó (ví dụ, các từ chết), nó không được nêu ở đây.

Bây giờ có khả năng để ràng buộc từ điển từ đồng nghĩa thesaurus_simple với các dạng thẻ token mong muốn trong một cấu hình, ví dụ:

```
ALTER TEXT SEARCH CONFIGURATION russian
        ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
        WITH thesaurus_simple;
```

12.6.4.2. Ví dụ về từ điển từ đồng nghĩa

Xem xét một từ điển đồng nghĩa về thiên văn học đơn giản thesaurus_astro, nó bao gồm một số kết hợp từ thiên văn học:

```
supernovae stars : sn
crab nebulae : crab
```

Bên dưới chúng ta tạo một từ điển và ràng buộc một số dạng thẻ token vào một từ điển đồng nghĩa thiên văn học và cọng tiếng Anh:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
        TEMPLATE = thesaurus,
        DictFile = thesaurus_astro,
        Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
        ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
        WITH thesaurus_astro, english_stem;
```

Bây giờ chúng ta có thể thấy cách mà nó làm việc. ts_lexize là không thật hữu dụng cho việc kiểm thử một từ điển đồng nghĩa, vì nó đối xử với đầu vào của nó như một thẻ token duy nhất. Thay vào đó chúng ta có thể sử dụng plainto_tsquery và to_tsvector mà chúng sẽ chia các chuỗi đầu vào của chúng thành nhiều thẻ token:

```
SELECT plainto_tsquery('supernova star');

plainto_tsquery
-----------------
'sn'

SELECT to_tsvector('supernova star');

to_tsvector
-------------
'sn':1
```

Theo nguyên tắc, người ta có thể sử dụng to_tsquery nếu bạn đưa vào ngoặc kép đối số:

```
SELECT to_tsquery('''supernova star''');
```

```
to_tsquery
------------

'sn'
```

Lưu ý rằng supernova star trùng khớp với supernovae stars trong thesaurus_astro vì chúng ta đã chỉ định từ điển cọng english_stem trong định nghĩa của từ điển từ đồng nghĩa. Cọng đó đã bỏ đi e và s.

Để đánh chỉ số các cụm từ gốc cũng như các thay thế, chỉ cần đưa nó vào phần bên tay phải của định nghĩa đó:

```
supernovae stars : sn supernovae stars
SELECT plainto_tsquery('supernova star');
plainto_tsquery
-----------------------------

'sn' & 'supernova' & 'star'
```

12.6.5. Từ điển Ispell

Mẫu template của từ điển Ispell hỗ trợ các từ điển hình thái học, nó có thể bình thường hóa nhiều dạng ngôn ngữ khác nhau của một từ thành từ vị y hệt. Ví dụ, một từ điển Ispell tiếng Anh có thể khớp với tất cả các biến cách và các kết hợp của khoản tìm kiếm bank, như, banking, banked, banks, banks', và bank's.

Phân phối PostgreSQL tiêu chuẩn không bao gồm bất kỳ tệp cấu hình Ispell nào. Các từ điển cho một số lượng lớn các ngôn ngữ là sẵn sàng từ Ispell[1]. Hơn nữa, một số định dạng tệp thư mục hiện đại hơn được hỗ trợ - MySpell[2] (OO < 2.0.1) và Hunspell[3] (OO >= 2.0.2). Một danh sách các từ điển là sẵn sàng trên OpenOffice Wiki[4].

Để tạo một từ điển Ispell, hãy sử dụng mẫu được xây dựng sẵn ispell và chỉ định vài tham số:

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
        TEMPLATE = ispell,
        DictFile = english,
        AffFile = english,
        StopWords = english
);
```

Ở đây, DictFile, AffFile, và StopWords chỉ định các tên cơ bản của từ điển, các phụ tố và các tệp từ chết. Tệp các từ chết có định dạng y hệt được giải thích ở trên cho dạng từ điển simple. Định dạng của các tệp khác không được chỉ định ở đây nhưng là sẵn sàng từ các website được nêu ở trên.

Các từ điển Ispell thường nhận ra một tập hợp hạn chế các từ, nên chúng nên có từ điển rộng hơn khác đi sau; ví dụ, một từ điển bông tuyết, nó nhận biết được mọi điều.

Các từ điển Ispell hỗ trợ việc chia các từ tổ hợp; một chức năng hữu dụng. Lưu ý rằng tệp các phụ tố nên chỉ định một cờ đặc biệt bằng việc sử dụng lệnh compoundwords controlled mà đánh dấu các từ của từ điển có thể tham gia trong sự hình thành tổ hợp đó.

1 http://ficus-www.cs.ucla.edu/geoff/ispell.html
2 http://en.wikipedia.org/wiki/MySpell
3 http://sourceforge.net/projects/hunspell/
4 http://wiki.services.openoffice.org/wiki/Dictionaries

```
compoundwords controlled z
```

Đây là một số ví dụ cho ngôn ngữ Nauy:

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
        {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
        {sjokoladefabrikk,sjokolade,fabrikk}
```

> **Lưu ý**: MySpell không hỗ trợ các từ tổ hợp. Hunspell có sự hỗ trợ phức tạp cho các từ tổ hợp. Hiện tại, PostgreSQL chỉ triển khai các hoạt động từ tổ hợp của Hunspell.

12.6.6. Từ điển bông tuyết (Snowball)

Mẫu của từ điển bông tuyết (Snowball) dựa vào một dự án của Martin Porter, người sáng chế ra thuật toán tước cọng nổi tiếng Porter cho tiếng Anh. Bông tuyết bây giờ đưa ra các thuật toán tước cọng cho nhiều ngôn ngữ (xem site Snowball[5] để có thêm thông tin). Từng thuật toán hiểu cách làm giảm các dạng biến thể phổ biến của các từ về một cơ sở, hoặc cọng, sự đánh vần trong ngôn ngữ đó. Một từ điển bông tuyết đòi hỏi một tham số ngôn ngữ language để nhận diện cọng nào sẽ sử dụng, và một cách tùy chọn có thể chỉ định tên một tệp từ chết stopword mà trao một danh sách các từ để loại bỏ. (Các danh sách từ tiêu chuẩn của PostgreSQL cũng được dự án Snowball đưa ra). Ví dụ, có một định nghĩa được xây dựng sẵn tương đương với:

```
CREATE TEXT SEARCH DICTIONARY english_stem (
        TEMPLATE = snowball,
        Language = english,
        StopWords = english
);
```

Định dạng tệp các từ chết là y hệt như được giải thích rồi.

Một từ điển bông tuyết nhận biết được mọi điều, dù có hay không có khả năng để đơn giản hóa từ đó, nên nó sẽ được đặt ở cuối của danh sách từ điển. Là vô dụng để đặt nó đứng trước bất kỳ từ điển nào khác vì một thẻ token sẽ không bao giờ truyền qua nó tới được từ điển tiếp sau.

12.7. Ví dụ cấu hình

Một cấu hình tìm kiếm văn bản chỉ định tất cả các lựa chọn cần thiết để biến một tài liệu thành một tsvector: trình phân tích sẽ sử dụng để chia văn bản thành các thẻ token, và các từ điển sẽ sử dụng để biến từng thẻ token thành một từ vị. Từng lời gọi của to_tsvector hoặc to_tsquery cần 1 cấu hình tìm kiếm văn bản để thực hiện việc xử lý của nó. Tham số cấu hình default_text_search_config chỉ định tên của cấu hình mặc định, nó là cấu hình được các hàm tìm kiếm văn bản sử dụng nếu một tham số cấu hình rõ ràng bị bỏ qua. Nó có thể được thiết lập trong postgresql.conf, hoặc được thiết lập cho một phiên riêng rẽ bằng việc sử dụng lệnh SET.

Vài cấu hình tìm kiếm văn bản được định nghĩa sẵn trước là sẵn sàng, và bạn có thể tạo ra các cấu hình tùy biến một cách dễ dàng. Để tạo thuận lợi cho quản lý các đối tượng tìm kiếm văn bản, một tập hợp các lệnh SQL là sẵn sàng, và có vài lệnh psql hiển thị thông tin về các đối tượng tìm kiếm

5 http://snowball.tartarus.org

văn bản đó (Phần 12.10).

Như một ví dụ, chúng ta sẽ tạo ra một cấu hình pg, bắt đầu bằng việc đúp bản cấu hình được xây dựng sẵn english:

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

Chúng ta sẽ sử dụng một danh sách từ đồng nghĩa đặc thù PostgreSQL và lưu trữ nó trong $SHAREDIR/tsearch_data/pg_dict.syn. Các nội dung của tệp đó trông giống như:

```
postgres pg
pgsql pg
postgresql pg
```

Chúng ta định nghĩa từ điển từ đồng nghĩa giống thế này:

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
        TEMPLATE = synonym,
        SYNONYMS = pg_dict
);
```

Tiếp theo chúng ta đăng ký từ điển Ispell english_ispell, nó có các tệp cấu hình của riêng nó:

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
        TEMPLATE = ispell,
        DictFile = english,
        AffFile = english,
        StopWords = english
);
```

Bây giờ chúng ta có thể thiết lập các ánh xạ cho các từ trong cấu hình pg:

```
ALTER TEXT SEARCH CONFIGURATION pg
        ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                          word, hword, hword_part
        WITH pg_dict, english_ispell, english_stem;
```

Chúng ta chọn không đánh chỉ số hoặc tìm kiếm các dạng thẻ token mà cấu hình được xây dựng sẵn đang điều khiển:

```
ALTER TEXT SEARCH CONFIGURATION pg
        DROP MAPPING FOR email, url, url_path, sfloat, float;
```

Bây giờ chúng ta có thể kiểm thử cấu hình của chúng ta:

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

Bước tiếp sau là để thiết lập phiên để sử dụng cấu hình mới, nó từng được tạo ra trong sơ đồ public:

```
=> \dF
```

Liệt kê các cấu hình tìm kiếm văn bản

```
Schema | Name | Description
-----------+---------+-------------
public    | pg      |

SET default_text_search_config = 'public.pg';
```

```
SET

SHOW default_text_search_config;

default_text_search_config
----------------------------
public.pg
```

12.8. Kiểm thử và gỡ lỗi tìm kiếm văn bản

Hành vi của một cấu hình tìm kiếm văn bản tùy biến có thể dễ dàng trở nên lúng túng. Các hàm được mô tả trong phần này là hữu dụng cho việc kiểm thử các đối tượng tìm kiếm văn bản. Bạn có thể kiểm thử một cấu hình hoàn chỉnh, hoặc tách bạch kiểm thử các trình phân tích và các từ điển.

12.8.1. Kiểm thử cấu hình

Hàm ts_debug cho phép kiểm thử dễ dàng một cấu hình tìm kiếm văn bản.

```
ts_debug([ config regconfig, ] document text,
        OUT alias text,
        OUT description text,
        OUT token text,
        OUT dictionaries regdictionary[],
        OUT dictionary regdictionary,
        OUT lexemes text[])
        returns setof record
```

ts_debug hiển thị thông tin về từng thẻ token của document như được trình phân tích tạo ra và được các từ điển được cấu hình xử lý. Nó sử dụng cấu hình được config hoặc default_text_search_config chỉ định nếu đối số đó bị bỏ qua.

ts_debug trả về một hàng cho từng thẻ token được trình phân tích nhận diện trong văn bản. Các cột được trả về là

- alias text - tên ngắn gọn của dạng thẻ token
- description text - mô tả dạng thẻ token
- token text - văn bản của thẻ token
- dictionaries regdictionary[] - các thư mục được cấu hình chọn cho dạng thẻ token đó
- dictionary regdictionary - từ điển nhận biết được thẻ token đó, hoặc NULL nếu không nhận ra
- lexemes text[] - (các) từ vị được từ điển đó tạo ra mà nhận biết được thẻ token đó, hoặc NULL nếu không nhận ra; một mảng rỗng ({}) có nghĩa là nó từng được nhận biết như là một từ chết

Đây là một ví dụ đơn giản:

```
SELECT * FROM ts_debug('english','a fat cat sat on a mat - it ate a fat rats');

alias      | description | token | dictionaries | dictionary | lexemes
-----------+-------------+-------+--------------+------------+---------
asciiword  | Word, all ASCII | a | {english_stem} | english_stem | {}
```

```
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | fat  | {english_stem} | english_stem | {fat}
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | cat  | {english_stem} | english_stem | {cat}
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | sat  | {english_stem} | english_stem | {sat}
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | on   | {english_stem} | english_stem | {}
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | a    | {english_stem} | english_stem | {}
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | mat  | {english_stem} | english_stem | {mat}
blank          | Space symbols      |      | {}             |                |
blank          | Space symbols      | -    | {}             |                |
asciiword      | Word, all ASCII    | it   | {english_stem} | english_stem | {}
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | ate  | {english_stem} | english_stem | {ate}
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | a    | {english_stem} | english_stem | {}
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | fat  | {english_stem} | english_stem | {fat}
blank          | Space symbols      |      | {}             |                |
asciiword      | Word, all ASCII    | rats | {english_stem} | english_stem | {rat}
```

Để có sự trình diễn rộng mở hơn, chúng ta trước hết tạo một cấu hình public.english và từ điển Ispell cho ngôn ngữ tiếng Anh:

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );
CREATE TEXT SEARCH DICTIONARY english_ispell (
        TEMPLATE = ispell,
        DictFile = english,
        AffFile = english,
        StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
        ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;

SELECT * FROM ts_debug('public.english','The Brightest supernovaes');
```

```
alias          | description        | token       | dictionaries                      | dictionary
---------------+--------------------+-------------+-----------------------------------+---------------
asciiword      | Word, all ASCII    | The         | {english_ispell,english_stem}     | english_ispell
blank          | Space symbols      |             | {}                                |
asciiword      | Word, all ASCII    | Brightest   | {english_ispell,english_stem}     | english_ispell
blank          | Space symbols      |             | {}                                |
asciiword      | Word, all ASCII    | supernovaes | {english_ispell,english_stem}     | english_stem
```

Trong ví dụ này, từ Brightest đã được trình phân tích nhận ra như là một từ ASCII (tên hiệu asciiword). Đối với dạng thẻ token này thì danh sách các từ điển là english_ispell và english_stem. Từ đó đã được english_ispell nhận ra, nó đã làm giảm từ đó xuống thành danh từ bright. Từ supernovaes không được từ điển english_ispell nhận ra nên nó được truyền tới từ điển tiếp sau, và, may thay, đã được nhận ra (trong thực tế, english_stem là một từ điển bông tuyết mà nhận ra được mọi điều; điều đó giải thích vì sao nó đã được đặt ở cuối của danh sách các từ điển).

Từ The đã được từ điển english_ispell nhận ra như là một từ chết (Phần 12.6.1) và sẽ không được

đánh chỉ số. Các khoảng trống cũng được bỏ qua, vì cấu hình hoàn toàn không đưa ra các từ điển cho chúng.

Bạn có thể giảm độ rộng của đầu ra bằng việc chỉ định rõ ràng các cột nào bạn muốn thấy:

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english','The Brightest supernovaes');

alias          | token       | dictionary      | lexemes
---------------+-------------+-----------------+-----------
asciiword      | The         | english_ispell  | {}
blank          |             |                 |
asciiword      | Brightest   | english_ispell  | {bright}
blank          |             |                 |
asciiword      | supernovaes | english_stem    | {supernova}
```

12.8.2. Kiểm thử trình phân tích

Các hàm sau cho phép kiểm thử trực tiếp một trình phân tích tìm kiếm toàn văn.

ts_parse(parser_name text, document text,
OUT *tokid integer*, OUT *token text) returns setof record*
ts_parse(parser_oid oid, document text,
OUT *tokid integer*, OUT *token text) returns setof record*

ts_parse phân tích document được đưa ra và trả về một loạt các bản ghi, mỗi bản ghi cho từng thẻ token được phân tích đó tạo ra. Từng bản ghi bao gồm một mã tokid chỉ ra dạng thẻ token được chỉ định và một thẻ token là văn bản của thẻ token đó. Ví dụ:

```
SELECT * FROM ts_parse('default', '123 - a number');

tokid | token
-------+--------
   22 | 123
   12 |
   12 | -
    1 | a
   12 |
    1 | number
```

ts_token_type(parser_name text, OUT *tokid integer*,
OUT *alias text*, OUT *description text) returns setof record*
ts_token_type(parser_oid oid, OUT *tokid integer,*
OUT *alias text*, OUT *description text) returns setof record*

ts_token_type trả về một bảng mô tả từng dạng thẻ token mà trình phân tích được chỉ định có thể nhận ra. Đối với từng dạng thẻ token, bảng đó đưa ra tokid số nguyên mà trình phân tích đó sử dụng để gắn nhãn cho một thẻ token của dạng đó, alias mà đặt tên cho dạng thẻ token trong các lệnh cấu hình, và một description ngắn gọn. Ví dụ:

```
SELECT * FROM ts_token_type('default');
tokid    | alias                      | description
----------+----------------------------+------------------------------------------------------------
       1 | asciiword                  | Word, all ASCII
       2 | word                       | Word, all letters
       3 | numword                    | Word, letters and digits
       4 | email                      | Email address
       5 | url                        | URL
       6 | host                       | Host
       7 | sfloat                     | Scientific notation
```

```
 8 | version          | Version number
 9 | hword_numpart    | Hyphenated word part, letters and digits
10 | hword_part       | Hyphenated word part, all letters
11 | hword_asciipart  | Hyphenated word part, all ASCII
12 | blank            | Space symbols
13 | tag              | XML tag
14 | protocol         | Protocol head
15 | numhword         | Hyphenated word, letters and digits
16 | asciihword       | Hyphenated word, all ASCII
17 | hword            | Hyphenated word, all letters
18 | url_path         | URL path
19 | file             | File or path name
20 | float            | Decimal notation
21 | int              | Signed integer
22 | uint             | Unsigned integer
23 | entity           | XML entity
```

12.8.3. Kiểm thử từ điển

Hàm ts_lexize tạo thuận lợi cho kiểm thử từ điển.

ts_lexize(dict regdictionary, token text) returns text[]

ts_lexize trả về một mảng các từ vị nếu đầu vào token được biết đối với từ điển đó, hoặc một mảng rỗng nếu thẻ token đó được biết đối với từ điển đó nhưng nó là một từ chết, hoặc NULL nếu nó là một từ không được biết. Ví dụ:

```
SELECT ts_lexize('english_stem', 'stars');

ts_lexize
-----------
{star}

SELECT ts_lexize('english_stem', 'a');

ts_lexize
-----------
{}
```

Lưu ý: Hàm ts_lexize kỳ vọng một thẻ token duy nhất, không phải văn bản text. Đây là một trường hợp nơi mà điều này có thể làm bối rối:

```
SELECT ts_lexize('thesaurus_astro','supernovae stars') is null;

?column?
----------
t
```

Từ điển các từ đồng nghĩa thesaurus_astro nhận ra cụm từ supernovae stars, nhưng ts_lexize thì không vì nó không phân tích văn bản đầu vào mà ứng xử với nó như một thẻ token duy nhất. Hãy sử dụng plainto_tsquery hoặc to_tsvector để kiểm thử các từ điển từ đồng nghĩa, ví dụ:

```
SELECT plainto_tsquery('supernovae stars');

plainto_tsquery
-----------------
'sn'
```

12.9. Dạng chỉ số GiST và GIN

Có 2 dạng chỉ số có thể được sử dụng để tăng tốc độ các tìm kiếm toàn văn. Lưu ý rằng các chỉ số đó không bắt buộc đối với việc tìm kiếm văn bản, nhưng trong các trường hợp một cột được tìm kiếm thường xuyên, thì một chỉ số thường là mong muốn.

CREATE INDEX name ON table USING gist(column);

Tạo một chỉ số dựa vào cây đảo ngược tổng quát - GiST (Generalized Search Tree). Cột đó có thể là dạng tsvector hoặc tsquery.

CREATE INDEX name ON table USING gin(column);

Tạo một chỉ số dựa vào đảo ngược tổng quát - GIN (Generalized Inverted Index). Cột column phải là dạng tsvector.

Có những khác biệt hiệu năng đáng kể giữa 2 dạng chỉ số đó, nên điều quan trọng phải hiểu các đặc tính của chúng.

Chỉ số GiST là *mất mát (lossy)*, nghĩa là chỉ số đó có thể tạo ra các trùng khớp sai, và cần thiết phải kiểm tra hàng thực sự của bảng để loại bỏ các trùng khớp sai như vậy. (PostgreSQL làm điều này tự động khi cần). Các chỉ số GiST là mất mát vì từng tài liệu được thể hiện trong chỉ số đó bằng một chữ ký có độ dài cố định. Chữ ký đó được tạo ra bằng việc băm từng từ thành một bit duy nhất trong một chuỗi n bit, với tất cả các bit đó được hoặc (OR) cùng nhau để tạo ra một chữ ký tài liệu n bit. Khi 2 từ băm cùng vị trí bit như y hệt thì sẽ có một sự khớp sai. Nếu tất cả các từ trong truy vấn có các sự trùng khớp (thực hoặc sai) thì hàng của bảng phải được truy xuất để xem liệu sự trùng khớp đó có là đúng hay không.

Sự mất mát đó làm giảm hiệu năng vì những lục lọi không cần thiết các bản ghi của bảng mà hóa ra là những trùng khớp sai. Vì sự truy cập ngẫu nhiên tới các bản ghi của bảng là chậm, điều này hạn chế tính hữu dụng của các chỉ số GiST. Khả năng các trùng khớp sai phụ thuộc vào vài yếu tố, đặc biệt số các từ duy nhất, nên việc sử dụng các từ điển để làm giảm số lượng này là được khuyến cáo.

Các chỉ số GIN là không mất mát đối với các truy vấn tiêu chuẩn, nhưng hiệu năng của nó phụ thuộc một cách logarit vào số các từ duy nhất. (Tuy nhiên, các chỉ số GIN chỉ lưu trữ các từ (các từ vị) của các giá trị tsvector, và không đối với các nhãn trọng số của chúng. Vì thế kiểm tra lại hàng của một bảng là cần thiết khi sử dụng một truy vấn mà có liên quan tới các trọng số).

Khi chọn dạng chỉ số nào để sử dụng, GiST hay GIN, hãy cân nhắc các khác biệt hiệu năng đó:

- Các tra cứu chỉ số GIN là 3 lần nhanh hơn so với GiST
- Các chỉ số GIN mất 3 lần thời gian lâu hơn để xây dựng so với GiST
- Các chỉ số GIN là chậm hơn một cách vừa phải để cập nhận so với các chỉ số GiST, nhưng khoảng 10 lần chậm hơn nếu sự hỗ trợ cập nhật nhanh bị vô hiệu hóa (xem Phần 53.3.1 để có thêm chi tiết).
- Các chỉ số GIN là từ 2-3 lần lớn hơn so với các chỉ số GiST

Như một qui tắc ngón tay cái, các chỉ số GIN là tốt nhất cho các dữ liệu tĩnh vì các tra cứu là nhanh hơn. Đối với các dữ liệu động, các chỉ số GiST là nhanh hơn để cập nhật. Đặc biệt, các chỉ số GiST rất tốt cho các dữ liệu động và nhanh nếu số lượng các từ duy nhất (các từ vị) ít hơn 100.000, trong khi các chỉ số GIN sẽ điều khiển 100.000 + các từ vị tốt hơn nhưng cập nhật sẽ chậm hơn.

Lưu ý rằng thời gian xây dựng chỉ số GNI thường có thể được cải thiện bằng việc gia tăng maintenance_work_mem, trong khi thời gian xây dựng chỉ số GiST là không phụ thuộc vào tham số đó.

Việc phân vùng của các bộ sưu tập lớn và sử dụng đúng phù hợp các chỉ số GiST và GIN cho phép sự triển khai các tìm kiếm rất nhanh với cập nhật trực tuyến. Việc phân vùng có thể được thực hiện ở mức cơ sở dữ liệu bằng việc sử dụng di sản của bảng, hoặc bằng việc phân phối các tài liệu qua các máy chủ và việc thu thập các kết quả tìm kiếm bằng việc sử dụng module mở rộng contrib/dblink. Cái sau là có khả năng vì các hàm xếp hạng chỉ sử dụng thông tin cục bộ.

12.10. Hỗ trợ psql

Thông tin về các đối tượng cấu hình tìm kiếm văn bản có thể có được trong psql bằng việc sử dụng một tập hợp các lệnh:

```
\dF{d,p,t}[+] [PATTERN]
```

Một lựa chọn + các thủ tục chi tiết hơn.

Tham số tùy chọn PATTERN có thể là tên của một đối tượng tìm kiếm văn bản, đủ điều kiện theo sơ đồ một cách tùy chọn. Nếu PATTERN bị bỏ qua thì thông tin về tất cả các đối tượng có thể thấy được sẽ được hiển thị. PATTERN có thể là một biểu thức thông thường và có thể đưa ra các mẫu riêng rẽ cho các tên sơ đồ và đối tượng đó. Các ví dụ sau minh họa cho điều này:

```
=> \dF *fulltext*
```

Liệt kê các cấu hình tìm kiếm văn bản:

```
Schema | Name          | Description
-----------+------------------+-------------
public   | fulltext_cfg   |
```

```
=> \dF *.fulltext*
```

Liệt kê các cấu hình tìm kiếm văn bản:

```
Schema | Name          | Description
-----------+------------------+-----------------
fulltext | fulltext_cfg   |
public   | fulltext_cfg   |
```

Các lệnh sẵn sàng là:

```
\dF[+] [PATTERN]
```

Liệt kê các cấu hình tìm kiếm văn bản (bổ sung + nhiều chi tiết hơn).

```
=> \dF russian
```

Liệt kê các cấu hình tìm kiếm văn bản

```
Schema     | Name    | Description
---------------+---------+-----------------------------------------------
pg_catalog | russian | configuration for russian language
```

```
=> \dF+ russian
```

Cấu hình tìm kiếm "pg_catalog.russian"

Trình phân tích: "pg_catalog.default"

```
Token          | Dictionaries
---------------------+--------------
asciihword     | english_stem
asciiword      | english_stem
email          | simple
file           | simple
float          | simple
host           | simple
hword          | russian_stem
hword_asciipart | english_stem
hword_numpart | simple
hword_part     | russian_stem
int            | simple
numhword       | simple
numword        | simple
sfloat         | simple
uint           | simple
url            | simple
url_path       | simple
version        | simple
word           | russian_stem
```

```
\dFd[+] [PATTERN]
```

Liệt kê các thư mục tìm kiếm văn bản (bổ sung + nhiều chi tiết hơn)

```
=> \dFd
```

Liệt kê các thư mục tìm kiếm văn bản

```
Schema     | Name              | Description
---------------+---------------------------+-----------------------------------------------------------------------------------------
pg_catalog | danish_stem       | snowball stemmer for danish language
pg_catalog | dutch_stem        | snowball stemmer for dutch language
pg_catalog | english_stem      | snowball stemmer for english language
pg_catalog | finnish_stem      | snowball stemmer for finnish language
pg_catalog | french_stem       | snowball stemmer for french language
pg_catalog | german_stem       | snowball stemmer for german language
pg_catalog | hungarian_stem    | snowball stemmer for hungarian language
pg_catalog | italian_stem      | snowball stemmer for italian language
pg_catalog | norwegian_stem    | snowball stemmer for norwegian language
pg_catalog | portuguese_stem   | snowball stemmer for portuguese language
pg_catalog | romanian_stem     | snowball stemmer for romanian language
pg_catalog | russian_stem      | snowball stemmer for russian language
pg_catalog | simple            | simple dictionary: just lower case and check for stopword
pg_catalog | spanish_stem      | snowball stemmer for spanish language
pg_catalog | swedish_stem      | snowball stemmer for swedish language
pg_catalog | turkish_stem      | snowball stemmer for turkish language
```

\dFp[+] [PATTERN]

Liệt kê các trình phân tích tìm kiếm văn bản (bổ sung + nhiều chi tiết hơn)

=> \dFp

Liệt kê các trình phân tích tìm kiếm văn bản

```
Schema     | Name    | Description
---------------------+---------+-----------------------------
pg_catalog | default | default word parser
```

=> \dFp+

Trình phân tích tìm kiếm văn bản "pg_catalog.default"

```
Method                | Function                | Description
----------------------------------+-----------------------------+-------------
Start parse           | prsd_start              |
Get next token        | prsd_nexttoken          |
End parse             | prsd_end                |
Get headline          | prsd_headline           |
Get token types       | prsd_lextype            |
```

Token types for parser "pg_catalog.default"

```
Token name            | Description
----------------------------------+-----------------------------------------
asciihword            | Hyphenated word, all ASCII
asciiword             | Word, all ASCII
blank                 | Space symbols
email                 | Email address
entity                | XML entity
file                  | File or path name
float                 | Decimal notation
host                  | Host
hword                 | Hyphenated word, all letters
hword_asciipart       | Hyphenated word part, all ASCII
hword_numpart         | Hyphenated word part, letters and digits
hword_part            | Hyphenated word part, all letters
int                   | Signed integer
numhword              | Hyphenated word, letters and digits
numword               | Word, letters and digits
protocol              | Protocol head
sfloat                | Scientific notation
tag                   | XML tag
uint                  | Unsigned integer
url                   | URL
url_path              | URL path
version               | Version number
word                  | Word, all letters
(23 rows)
```

\dFt[+] [PATTERN]

Liệt kê các mẫu template tìm kiếm văn bản (bổ sung + nhiều chi tiết hơn)

=> \dFt

Liệt kê các mẫu template tìm kiếm văn bản

```
Schema     | Name       | Description
---------------+---------------+-----------------------------------------------------------------------------------------
pg_catalog | ispell     | ispell dictionary
pg_catalog | simple     | simple dictionary: just lower case and check for stopword
pg_catalog | snowball   | snowball stemmer
pg_catalog | synonym    | synonym dictionary: replace word by its synonym
pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution
```

12.11. Hạn chế

Các hạn chế hiện hành của các chức năng tìm kiếm văn bản của PostgreSQL là:

- Độ dài của từng từ vị phải ít hơn 2K byte
- Độ dài của một tsvector (các từ vị + các vị trí) phải ít hơn 1 MB
- Số lượng các từ vị phải nhỏ hơn 2^{64}
- Các giá trị vị trí trong tsvector phải lớn hơn 0 và không lớn hơn 16.383
- Không lớn hơn 256 vị trí cho từng từ vị
- Số lượng các nút (các từ vị + các toán tử) trong một tsquery phải ít hơn 32.768

Để so sánh, tài liệu của PostgreSQL 8.1 chứa 10.441 từ duy nhất, tổng số 335.420 từ, và từ thường xuyên nhất "postgresql" đã được nhắc tới 6.127 lần trong 655 tài liệu.

Một ví dụ khác - tài liệu của PostgreSQL 8.1 có chứa 910.989 từ duy nhất với 57.491.343 từ vị trong 461.020 thông điệp.

12.12. Chuyển tìm kiếm văn bản khỏi phiên bản trước 8.3

- Một số hàm từng được đổi tên hoặc đã có các tinh chỉnh nhỏ trong các danh sách đối số của chúng, và tất cả chúng bây giờ là trong sơ đồ pg_catalog, trong khi một cài đặt trước đó chúng có thể nằm trong sơ đồ public hoặc sơ đồ không hệ thống. Có một phiên bản mới của contrib/tsearch2 (xem Phần F.38), nó đưa ra một lớp tương thích để giải quyết hầu hết các vấn đề trong lĩnh vực này.
- Các hàm cũ contrib/tsearch2 và các đối tượng khác phải được ép khi tải đầu ra pg_dump từ một cơ sở dữ liệu trước phiên bản 8.3. Trong khi nhiều trong số chúng sẽ không tải được bất kỳ cách gì, thì một ít sẽ và sau đó gây ra các vấn đề. Một cách đơn giản để làm việc với điều này là tải module mới contrib/tsearch2 trước khi phục hồi sự hỏng hóc; sau đó nó sẽ khóa các đối tượng cũ khỏi được tải lên. Các ứng dụng đã sử dụng module bổ sung contrib/tsearch2 cho việc tìm kiếm văn bản sẽ cần một số tinh chỉnh để làm việc với các chức năng được xây dựng sẵn:
- Thiết lập cấu hình tìm kiếm văn bản hoàn toàn là khác bây giờ. Thay vì việc chèn bằng tay các hàng vào các bảng cấu hình, thì sự tìm kiếm được cấu hình thông qua các lệnh SQL được chuyên môn hóa được chỉ ra trước đó trong chương này. Không có sự hỗ trợ tự động

cho việc biến đổi một cấu hình tùy biến đang tồn tại cho phiên bản 8.3; tự bạn có của riêng bạn ở đây.

- Hầu hết các dạng từ điển dựa vào một số tệp cấu hình bên ngoài cơ sở dữ liệu. Chúng phần lớn là tương thích với sử dụng trước phiên bản 8.3, nhưng lưu ý những khác biệt sau đây:
  - Các tệp cấu hình phải được đặt trong một thư mục duy nhất được chỉ định (`$SHAREDIR/tsearch_data`), và phải có một mở rộng đặc thù phụ thuộc vào dạng tệp đó, như được nêu trước đó trong các mô tả của các dạng từ điển khác nhau. Hạn chế này từng được thêm vào cho các vấn đề an ninh làm trước.
  - Các tệp cấu hình phải được mã hóa theo UTF-8, bất kể việc mã hóa nào của cơ sở dữ liệu được sử dụng.
  - Trong các tệp cấu hình của từ điển từ đồng nghĩa, các từ chết phải được đánh dấu với ?.

Chương 13. Kiểm soát đồng thời

Chương này mô tả hành vi của hệ cơ sở dữ liệu PostgreSQL khi 2 hoặc nhiều hơn phiên làm việc cố gắng truy cập cùng các dữ liệu cùng một lúc. Các mục tiêu trong tình huống này là để cho phép sự truy cập có hiệu quả cho tất cả các phiên trong khi duy trì được tính toàn vẹn của dữ liệu một cách nghiêm ngặt. Mỗi lập trình viên các ứng dụng cơ sở dữ liệu nên làm quen với các chủ đề được đề cập tới trong chương này.

13.1. Giới thiệu

PostgreSQL đưa ra một tập hợp giàu có các công cụ cho các lập trình viên để quản lý sự truy cập đồng thời tới các dữ liệu. Một cách nội bộ, tính nhất quán của dữ liệu được duy trì bằng việc sử dụng một mô hình nhiều phiên bản (Kiểm soát Đồng thời Nhiều phiên bản – MVCC [MultiVersion Concurrency Control]). Điều này có nghĩa là trong khi truy vấn một cơ sở dữ liệu thì từng giao dịch thấy một hình chụp các dữ liệu (một phiên bản cơ sở dữ liệu) khi nó từng xảy ra lúc nào đó trong quá khứ, bất kể tình trạng hiện hành của dữ liệu nằm bên dưới. Điều này bảo vệ giao dịch khỏi việc thấy các dữ liệu không nhất quán mà có thể xảy ra vì các cập nhật giao dịch đồng thời (khác) trong cùng các hàng dữ liệu y hệt đó, đưa ra sự *cách li giao dịch* cho từng phiên cơ sở dữ liệu. MVCC, bằng việc từ chối các phương pháp khóa độc quyền của các hệ thống cơ sở dữ liệu truyền thống, làm giảm sự tranh khóa để cho phép hiệu năng hợp lý trong các môi trường nhiều người sử dụng.

Ưu điểm chính của việc sử dụng mô hình MVCC đối với kiểm soát đồng thời hơn là việc khóa là ở chỗ các khóa MVCC được yêu cầu truy vấn (đọc) các dữ liệu không xung đột với các khóa được yêu cầu ghi các dữ liệu, và vì thế việc đọc không bao giờ khóa việc ghi và việc ghi không bao giờ khóa việc đọc.

Các cơ sở khóa mức bảng và hàng cũng sẵn sàng trong PostgreSQL cho các ứng dụng mà không thể áp dụng dễ dàng đối với hành vi của MVCC. Tuy nhiên, sử dụng đúng phù hợp của MVCC sẽ thường cung cấp hiệu năng tốt hơn so với các khóa. Hơn nữa, các khóa cố vấn do các ứng dụng định nghĩa đưa ra cơ chế cho việc có được các khóa không bị ràng buộc vào một giao dịch duy nhất.

13.2. Sự cách li giao dịch

Tiêu chuẩn SQL định nghĩa 4 mức cách li giao dịch đối với 3 hiện tượng phải được ngăn chặn giữa các giao dịch đồng thời. Các hiện tượng không mong muốn đó là:

đọc bẩn

Một giao dịch đọc các dữ liệu được một giao dịch đồng thời không thực hiện được ghi lại

đọc không lặp lại

Một giao dịch đọc lại các dữ liệu đã đọc rồi trước đó và thấy các dữ liệu đó từng bị một giao dịch khác (mà đã thực hiện kể từ lần đọc đầu tiên) sửa đổi.

đọc ma

> Một giao dịch thực hiện lại một truy vấn trả về một tập hợp các hàng làm thỏa mãn một điều kiện tìm kiếm và thấy rằng tập hợp các hàng đó làm thỏa mãn điều kiện đã thay đổi vì giao dịch được thực hiện gần đây.

4 mức cách li giao dịch và các hành vi tương ứng được mô tả trong Bảng 13-1.

Bảng 13-1. Các mức cách li giao dịch SQL

Mức cách li	Đọc bẩn	Đọc không lặp lại	Đọc ma
Read uncommitted - Đọc không thực hiện được	Possible - Có thể	Possible - Có thể	Possible - Có thể
Read committed - Đọc thực hiện được	Not possible - Không thể	Possible - Có thể	Possible - Có thể
Repeatable read - Đọc lặp lại	Not possible - Không thể	Not possible - Không thể	Possible - Có thể
Serializable - Có khả năng tuần tự	Not possible - Không thể	Not possible - Không thể	Not possible - Không thể

Trong PostgreSQL, bạn có thể yêu cầu bất kỳ mức nào trong 4 mức cách li giao dịch tiêu chuẩn. Nhưng trong nội bộ, chỉ có 2 mức cách li phân biệt, tương ứng với các mức Đọc thực hiện được (Read Committed) và Có khả năng tuần tự (Serializable). Khi bạn chọn mức Đọc không thực hiện được (Read Uncommitted) thì bạn thực tế sẽ đọc thực hiện được (Read Committed), và khi bạn chọn Đọc lặp lại được thì thực tế bạn Có khả năng tuần tự (Serializable), nên mức cách li thực sự có thể là khắt khe hơn so với bạn chọn. Điều này được tiêu chuẩn SQL cho phép: 4 mức cách li chỉ định nghĩa hiện tượng nào phải không xảy ra, chúng không định nghĩa hiện tượng nào phải xảy ra. Lý do là PostgreSQL chỉ đưa ra 2 mức cách li là điều này chỉ là cách nhạy cảm để ánh xạ các mức cách li tiêu chuẩn với kiến trúc kiểm soát đồng thời nhiều phiên bản. Hành vi của các mức cách li có sẵn được chi tiết hóa trong các phần sau.

Để thiết lập mức cách li của một giao dịch, hãy sử dụng lệnh SET TRANSACTION.

13.2.1. Mức cách li đọc thực hiện được

Đọc thực hiện được là mức cách li mặc định trong PostgreSQL. Khi một giao dịch sử dụng mức cách li này, thì một truy vấn SELECT (không có mệnh đề FOR UPDATE/SHARE) chỉ xem các dữ liệu được thực hiện trước khi truy vấn đó đã bắt đầu; nó không bao giờ xem hoặc các dữ liệu không được thực hiện, hoặc các thay đổi được thực hiện trong quá trình thực hiện truy vấn của các giao dịch đồng thời. Để có hiệu lực, một truy vấn SELECT xem một ảnh chụp cơ sở dữ liệu ngay khi truy vấn đó bắt đầu chạy. Tuy nhiên, SELECT sẽ xem các tác động của các cập nhật trước đó được thực hiện trong giao dịch của riêng nó, thậm chí dù chúng còn chưa được thực hiện xong. Hơn nữa lưu ý rằng 2 lệnh SELECT kế tiếp có thể xem các dữ liệu khác nhau, thậm chí dù chúng là trong một giao dịch duy nhất, nếu các giao dịch khác thực hiện các thay đổi trong quá trình thực thi lệnh SELECT đầu tiên.

Các lệnh UPDATE, DELETE, SELECT FOR UPDATE, và SELECT FOR SHARE hành xử y hệt như lệnh SELECT

trong việc tìm kiếm các hàng đích: chúng sẽ chỉ tìm các hàng đích mà đã được thực hiện tại thời điểm bắt đầu của lệnh đó. Tuy nhiên, một hàng đích như vậy có thể đã được cập nhật rồi (hoặc được xóa hoặc bị khóa) vì một giao dịch đồng thời khác vào thời điểm nó được tìm thấy. Trong trường hợp này, việc cập nhật có thể sẽ chờ cho giao dịch cập nhật đầu tiên thực hiện xong hoặc quay lại (nếu nó vẫn còn diễn ra). Nếu việc cập nhật đầu tiên quay lại, thì các tác động của nó bị phủ định và việc cập nhật thứ 2 có thể tiến hành với việc cập nhật cho hàng ban đầu được tìm thấy. Nếu việc cập nhật đầu tiên thực hiện, thì việc cập nhật thứ 2 sẽ bỏ qua hàng đó nếu việc cập nhật đầu tiên đã xóa nó, nếu không thì nó sẽ cố áp dụng hành động của nó cho phiên bản được cập nhật của hàng đó. Điều kiện tìm kiếm của lệnh (mệnh đề WHERE) được đánh giá lại để xem liệu phiên bản được cập nhật của hàng còn trùng khớp với điều kiện tìm kiếm hay không. Nếu có, thì việc cập nhật thứ 2 tiến hành hoạt động của nó bằng việc sử dụng phiên bản được cập nhật của hàng đó. Trong trường hợp các lệnh SELECT FOR UPDATE và SELECT FOR SHARE, điều này có nghĩa là phiên bản được cập nhật của hàng đó bị khóa và được trả về cho máy trạm.

Vì qui tắc ở trên, có khả năng cho một lệnh cập nhật xem một hình chụp không nhất quán: nó có thể xem các tác động của các lệnh cập nhật đồng thời lên các hàng y hệt mà nó đang cố gắng cập nhật, nhưng nó không xem các tác động của các lệnh đó lên các hàng khác trong cơ sở dữ liệu. Hành vi này làm cho chế độ Đọc thực hiện được không phù hợp với các lệnh có liên quan với các điều kiện tìm kiếm phức tạp; tuy nhiên, điều đó là đúng cho các trường hợp đơn giản hơn. Ví dụ, hãy cân nhắc việc cập nhật các bảng cân đối ngân hàng với các giao dịch như:

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

Nếu 2 giao dịch như vậy đồng thời cố gắng thay đổi bảng cân đối tài khoản 12345, thì chúng ta rõ ràng muốn giao dịch thứ 2 bắt đầu với phiên bản được cập nhật hàng của tài khoản đó. Vì từng lệnh đang có tác động chỉ cho một hàng được xác định trước đó, hãy để nó xem phiên bản được cập nhật của hàng không tạo ra bất kỳ sự không nhất quán đáng lo ngại nào.

Sử dụng phức tạp hơn có thể tạo ra các kết quả không mong muốn trong chế độ Đọc thực hiện được. Ví dụ, hãy cân nhắc việc hoạt động của một lệnh DELETE đối với các dữ liệu đang vừa được bổ sung thêm và vừa được loại bỏ khỏi các tiêu chí hạn chế của nó bằng một lệnh khác, như, giả thiết website là một bảng 2 hàng với website.hits bằng 9 và 10:

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- chạy từ phiên khác: DELETE FROM website WHERE hits = 10;
COMMIT;
```

DELETE sẽ không có tác động thậm chí dù có một hàng website.hits = 10 trước và sau UPDATE. Điều này xảy ra vì giá trị 9 của hàng trước cập nhật, và khi UPDATE hoàn thành và DELETE có một sự khóa, thì giá trị hàng mới không còn là 10 mà là 11, nó không còn trùng khớp với các tiêu chí đó nữa.

Vì chế độ Đọc thực hiện được bắt đầu từng lệnh với một hình chụp mới mà bao gồm tất cả các giao dịch được thực hiện tùy thuộc vào sự tức thì đó, các lệnh tiếp sau trong cùng giao dịch sẽ thấy các

tác động của giao dịch đồng thời được thực hiện trong bất kỳ trường hợp nào. Điểm mấu chốt trong vấn đề ở trên là liệu có hay không một lệnh duy nhất thấy một kiểu nhìn nhất quán tuyệt đối cơ sở dữ liệu đó.

Sự cách li giao dịch một phần được chế độ Đọc thực hiện được cung cấp là phù hợp cho nhiều ứng dụng, và chế độ này là nhanh và đơn giản để sử dụng; tuy nhiên, nó là không đủ cho tất cả các trường hợp. Các ứng dụng mà thực hiện các truy vấn và các cập nhật phức tạp có thể đòi hỏi một kiểu nhìn nhất quán chặt chẽ của cơ sở dữ liệu hơn là chế độ Đọc thực hiện được đưa ra.

13.2.2. Mức cách li có khả năng tuần tự

Mức cách li có khả năng tuần tự đưa ra sự cách li giao dịch chặt chẽ nhất. Mức này giả lập thực thi giao dịch tuần tự, dường như các giao dịch từng được thực thi cái này sau cái kia, một cách tuần tự, thay vì đồng thời. Tuy nhiên, các ứng dụng sử dụng mức này phải được chuẩn bị để thử lại các giao dịch khi sự tuần tự bị hỏng.

Khi một giao dịch đang sử dụng mức tuần tự, một truy vấn SELECT chỉ thấy các dữ liệu được thực hiện trước khi giao dịch đó bắt đầu; nó không bao giờ thấy hoặc các dữ liệu không được thực hiện hoặc những thay đổi được thực hiện trong quá trình thực thi giao dịch đó của các giao dịch đồng thời. (Tuy nhiên, truy vấn đó thấy các hiệu ứng của các cập nhật trước đó được thực thi bên trong giao dịch của riêng nó, thậm chí dù chúng còn chưa được thực hiện xong). Điều này khác với Đọc thực hiện được theo đó một truy vấn trong một giao dịch tuần tự thấy một hình chụp như lúc bắt đầu giao dịch, không như lúc bắt đầu của truy vấn hiện hành trong giao dịch đó. Vì thế, các lệnh SELECT tiếp sau trong một giao dịch duy nhất thấy các dữ liệu y hệt, nghĩa là, chúng không thấy các thay đổi được các giao dịch khác làm mà đã thực hiện xong sau khi giao dịch của riêng chúng đã bắt đầu. (Hành vi này có thể là lý tưởng cho việc báo cáo các ứng dụng).

Các lệnh UPDATE, DELETE, SELECT FOR UPDATE, và SELECT FOR SHARE hành xử y hệt như SELECT về việc tìm kiếm các hàng đích: chúng sẽ chỉ tìm các hàng đích mà đã được thực hiện xong như của thời điểm bắt đầu giao dịch. Tuy nhiên, một hàng đích như vậy có thể đã được cập nhật rồi (hoặc bị xóa hoặc khóa rồi) vì giao dịch đồng thời khác vào thời điểm nó được tìm thấy. Trong trường hợp này, giao dịch tuần tự sẽ chờ cho giao dịch cập nhật đầu tiên thực hiện xong hoặc quay lại (nếu nó vẫn còn diễn ra). Nếu cập nhật đầu tiên quay lại, thì các tác động của nó sẽ bị phủ định và giao dịch tuần tự có thể xử lý bằng việc cập nhật hàng được thấy ban đầu. Nhưng nếu cập nhật đầu tiên thực hiện (và thực tế đã cập nhật hoặc xóa hàng đó rồi, thì sẽ không khóa nó) rồi sau đó giao dịch tuần tự sẽ bị quay lại với thông điệp lỗi

ERROR: could not serialize access due to concurrent update (Lỗi: không thể truy cập tuần tự vì cập nhật đồng thời)

vì một giao dịch tuần tự không thể sửa hoặc khóa các hàng bị các giao dịch khác làm thay đổi sau khi giao dịch tuần tự đó đã bắt đầu.

Khi một ứng dụng nhận được thông điệp lỗi này, nó sẽ hủy bỏ giao dịch hiện hành và thử lại toàn bộ giao dịch đó từ đầu. Lần thứ 2 trôi qua, giao dịch đó sẽ thấy sự thay đổi được thực hiện rồi trước

đó như một phần của kiểu nhìn ban đầu của nó đối với cơ sở dữ liệu, nên sẽ không có xung đột logic trong việc sử dụng phiên bản mới của hàng đó như điểm bắt đầu đối với sự cập nhật của giao dịch mới.

Lưu ý rằng chỉ việc cập nhật các giao dịch có thể cần phải được thử lại; các giao dịch chỉ đọc sẽ không bao giờ có các xung đột tuần tự.

Chế độ tuần tự đưa ra một đảm bảo chặt chẽ rằng từng giao dịch thấy toàn bộ một kiểu nhìn nhất quán của cơ sở dữ liệu. Tuy nhiên, ứng dụng phải được chuẩn bị để thử lại các giao dịch khi các cập nhật đồng thời làm cho nó không có khả năng duy trì bền vững ảo tưởng đối với sự thực thi tuần tự. Vì chi phí của việc hoãn thực hiện các giao dịch phức tạp có thể là đáng kể, nên chế độ tuần tự chỉ được khuyến cáo khi việc cập nhật các giao dịch có sự phức tạp logic đáng kể mà chúng có thể đưa ra các câu trả lời sai trong chế độ Đọc thực hiện được. Phổ biến nhất, chế độ tuần tự là cần thiết khi một giao dịch thực thi vài lệnh tiếp sau mà phải thấy các kiểu nhìn y hệt nhau của cơ sở dữ liệu.

13.2.2.1. Cách li tuần tự so với sự tuần tự đúng

Ý nghĩa trực quan (và định nghĩa toán học) của sự thực thi "tuần tự" là bất kỳ 2 giao dịch đồng thời được thực hiện thành công nào cũng sẽ dường như sẽ phải thực thi tuần tự một cách chặt chẽ, cái này sau cái kia - dù cái nào xảy ra trước đều có thể không đoán trước được trước đó. Điều này quan trọng để nhận thức được rằng việc cấm các hành vi không mong muốn được liệt kê trong Bảng 13-1 là không đủ để đảm bảo tính tuần tự đúng, và trong thực tế chế độ tuần tự được của PostgreSQL *không đảm bảo sự thực thi có thể tuần tự được theo nghĩa này*. Như một ví dụ, hãy cân nhắc một bảng mytab, ban đầu bao gồm:

```
class | value
-------+-------
    1 |  10
    1 |  20
    2 | 100
    2 | 200
```

Giả thiết giao dịch tuần tự A tính:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

và sau đó chèn kết quả (30) là value vào 1 hàng với class = 2. Đồng thời, giao dịch tuần tự B tính:

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

và có kết quả 300, mà nó chèn vào một hàng với class = 1. Sau đó cả 2 giao dịch thực hiện xong. Không hành vi không mong muốn nào được liệt kê đã xảy ra, vâng chúng ta có một kết quả có thể đã không xảy ra trong cả trật tự một cách tuần tự. Nếu A đã thực thi trước B, thì B có thể đã tính tổng 330, chứ không phải là 300, và tương tự trật tự khác có thể đã dẫn tới một tổng số khác được A tính toán.

Để đảm bảo tính tuần tự toán học đúng, cần thiết đối với một hệ thống cơ sở dữ liệu ép tuân thủ việc *khóa đuôi*, nó có nghĩa là một giao dịch không thể chèn hoặc sửa đổi một hàng mà có thể đã trùng khớp điều kiện WHERE của một truy vấn trong giao dịch đồng thời khác. Ví dụ, một khi giao

dịch A đã thực thi truy vấn SELECT ... WHERE class = 1, thì một hệ thống khóa đuôi có thể cấm giao dịch B chèn bất kỳ hàng mới nào với class = 1 cho tới khi A đã thực hiện xong[1]. Một hệ thống khóa như vậy là phức tạp để triển khai và cực kỳ đắt trong thực thi, vì từng phiên phải nhận thức được các chi tiết của từng truy vấn được từng giao dịch đồng thời thực thi. Và chi phí lớn này hầu hết là phí phạm, vì trong thực tế hầu hết các ứng dụng không tiến hành sắp xếp những thứ mà có thể gây ra các vấn đề. (Chắc chắn ví dụ ở trên là được đặt ra và có lẽ không đại diện cho phần mềm thực tế). Vì các lý do đó, PostgreSQL không triển khai khóa đuôi.

Trong các trường hợp nơi mà khả năng thực thi không tuần tự là một mối nguy có thực, thì các vấn đề có thể được ngăn chặn bằng việc sử dụng khóa độc quyền phù hợp. Thảo luận tiếp trong các phần tiếp sau.

13.3. Khóa độc quyền

PostgreSQL đưa ra các chế độ khóa khác nhau để kiểm soát sự truy cập đồng thời tới dữ liệu trong các bảng. Các chế độ đó có thể được sử dụng cho việc khóa được ứng dụng kiểm soát trong các tình huống nơi mà MVCC không đưa ra hành vi mong muốn. Hơn nữa, hầu hết các lệnh PostgreSQL tự động có được các khóa của các chế độ phù hợp để đảm bảo rằng các bảng tham chiếu không bị bỏ đi hoặc bị sửa đổi theo các cách thức không tương thích trong khi lệnh đó thực thi. (Ví dụ, ALTER TABLE không thể được chạy an toàn đồng thời với các hoạt động khác trong cùng y hệt một bảng, nên nó có được một khóa độc quyền trong bảng đó để ép tuân thủ điều đó).

Để kiểm tra một danh sách các khóa phổ biến hiện nay trong một máy chủ cơ sở dữ liệu, hãy sử dụng kiểu nhìn hệ thống pg_locks. Để có thêm thông tin về việc giám sát tình trạng hệ thống con của người quản lý khóa, hãy tham chiếu tới Chương 27.

13.3.1. Khóa mức bảng

Danh sách bên dưới chỉ ra các chế độ khóa có sẵn và các ngữ cảnh trong đó chúng được PostgreSQL sử dụng tự động. Bạn cũng có thể có được bất kỳ khóa rõ ràng nào với lệnh LOCK. Hãy nhớ rằng tất cả các chế độ khóa đó là các khóa mức bảng, thậm chí nếu tên có từ "hàng"; tên các chế độ khóa là theo lịch sử. Ở một số mức độ nào đó thì các tên phản ánh sự sử dụng điển hình của từng chế độ khóa - nhưng ngữ nghĩa tất cả là như nhau. Sự khác biệt thực tế duy nhất giữa chế độ khóa này với chế độ khóa kia là tập hợp các chế độ khóa với nó từng chế độ xung đột (xem Bảng 13-2). 2 giao dịch không thể có các khóa của các chế độ xung đột trong cùng một bảng cùng một thời điểm. (Tuy nhiên, một giao dịch không bao giờ xung đột với bản thân nó. Ví dụ, nó có thể có khóa ACCESS EXCLUSIVE và sau đó có khóa ACCESS SHARE trong cùng y hệt bảng đó). Các chế độ khóa không xung đột có thể được nhiều giao dịch nắm đồng thời. Lưu ý đặc biệt rằng một số chế độ khóa là tự xung đột (ví dụ, một khóa ACCESS EXCLUSIVE không thể được nhiều hơn một giao dịch tại một thời điểm nắm giữ), trong khi các khóa khác là không tự xung đột (ví dụ, một khóa ACCESS

1 Về cơ bản, một hệ thống khóa đuôi ngăn chặn các đọc ma bằng việc hạn chế những gì được viết, trong khi MVCC ngăn chặn chúng bằng việc hạn chế những gì được đọc.

SHARE có thể được nhiều giao dịch nắm giữ).

Các chế độ khóa mức bảng

ACCESS SHARE

Chỉ xung đột với chế độ khóa ACCESS EXCLUSIVE.

Lệnh SELECT có một khóa của chế độ này trong các bảng được tham chiếu. Nói chung, bất kỳ truy vấn nào mà chỉ đọc một bảng và không sửa đổi thì nó sẽ có chế độ khóa này.

ROW SHARE

Xung đột với các chế độ khóa EXCLUSIVE và ACCESS EXCLUSIVE.

Các lệnh SELECT FOR UPDATE và SELECT FOR SHARE có một khóa chế độ này trong (các) bảng đích (bổ sung thêm vào các khóa ACCESS SHARE trong bất kỳ bảng nào khác mà được tham chiếu nhưng không được lựa chọn FOR UPDATE/FOR SHARE).

ROW EXCLUSIVE

Xung đột với các chế độ khóa SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, và ACCESS EXCLUSIVE.

Các lệnh UPDATE, DELETE, và INSERT có chế độ khóa này trong bảng đích (bổ sung thêm vào các khóa ACCESS SHARE trong bất kỳ bảng được tham chiếu nào khác). Nói chung, chế độ khóa này sẽ có được từ bất kỳ lệnh nào mà sửa đổi dữ liệu trong một bảng.

SHARE UPDATE EXCLUSIVE

Xung đột với các chế độ khóa SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, và ACCESS EXCLUSIVE. Chế độ này bảo vệ một bảng chống lại những thay đổi sơ đồ đồng thời và chạy VACUUM.

Có được từ các lệnh VACUUM (không có FULL), ANALYZE, và CREATE INDEX CONCURRENTLY.

SHARE

Xung đột với các chế độ khóa ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, và ACCESS EXCLUSIVE. Chế độ này bảo vệ bảng đối với các thay đổi dữ liệu đồng thời.

Có được bằng lệnh CREATE INDEX (không có CONCURRENTLY).

SHARE ROW EXCLUSIVE

Xung đột với các chế độ khóa ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, và ACCESS EXCLUSIVE.

Chế độ khóa này không tự động có được bằng lệnh PostgreSQL.

EXCLUSIVE

Xung đột với các chế độ khóa ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, và ACCESS EXCLUSIVE. Chế độ này chỉ cho phép

các khóa đồng thời, nghĩa là, chỉ đọc từ bảng có thể được xử lý song song với một giao dịch nắm giữ chế độ khóa này.

Chế độ khóa này không tự động có được trong các bảng của người sử dụng bằng bất kỳ lệnh PostgreSQL nào. Tuy nhiên nó có trong các catalog hệ thống nhất định ở một số hoạt động.

ACCESS EXCLUSIVE

Xung đột với các khóa của tất cả các chế độ ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, và ACCESS EXCLUSIVE. Chế độ này đảm bảo rằng người nắm giữ là giao dịch duy nhất truy cập được bảng theo bất kỳ cách gì.

Có được bằng các lệnh ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, và VACUUM FULL. Đây cũng là chế độ khóa mặc định cho các lệnh LOCK TABLE mà không chỉ định một chế độ rõ ràng.

Mẹo: Chỉ khối ACCESS EXCLUSIVE khóa một lệnh SELECT (không có FOR UPDATE/SHARE).

Một khi có được, một khóa thường được giữ cho tới khi kết thúc giao dịch. Nhưng nếu một khóa có được sau việc thiết lập một điểm an toàn, thì khóa đó được nhả ra ngay lập tức nếu điểm an toàn đó được quay lại. Điều này là nhất quán với nguyên tắc rằng ROLLBACK hoãn tất cả các tác động của các lệnh kể từ điểm an toàn. Điều y hệt giữ cho các khóa có được bên trong một khối ngoại lệ PL/pgSQL: một lỗi thoát khỏi khối đó nhả ra các khóa có được bên trong nó.

Bảng 13-2. Các chế độ khóa xung đột

Chế độ khóa được yêu cầu	**Chế độ khóa hiện hành**							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

13.3.2. Các khóa mức hàng

Bổ sung thêm vào các khóa mức bảng, có các khóa mức hàng, chúng có thể là các khóa độc quyền

hoặc được chia sẻ. Một khóa độc quyền mức hàng trong một hàng cụ thể sẽ tự động có được khi hàng đó được cập nhật hoặc bị xóa. Khóa đó được giữ cho tới khi giao dịch thực hiện xong hoặc quay lại, giống hệt như các khóa mức bảng. Các khóa mức hàng không ảnh hưởng tới việc truy vấn dữ liệu; chúng chỉ khóa việc *ghi vào cùng một hàng*.

Để có được một khóa độc quyền mức hàng trong một hàng mà không thực sự sửa đổi hàng đó, hãy chọn hàng đó bằng lệnh SELECT FOR UPDATE. Lưu ý là một khi khóa mức hàng có được, thì giao dịch có thể cập nhật hàng đó nhiều lần mà không sợ có các xung đột.

Để có được một khóa mức hàng được chia sẻ trong một hàng, hãy chọn hàng đó bằng lệnh SELECT FOR SHARE. Một khóa được chia sẻ không cản trở các giao dịch khác khỏi việc có cùng y hệt khóa được chia sẻ đó. Tuy nhiên, không giao dịch nào được phép cập nhật, xóa, hoặc khóa độc quyền một hàng theo đó bất kỳ giao dịch nào khác nắm giữ một khóa được chia sẻ. Bất kỳ cố gắng nào làm như vậy sẽ khóa cho tới khi (các) khóa chia sẻ được nhả ra.

PostgreSQL không nhớ bất kỳ thông tin nào về các hàng được sửa đổi trong bộ nhớ, nên không có giới hạn về số lượng các hàng bị khóa tại một thời điểm. Tuy nhiên, việc khóa một hàng có thể gây ra một sự ghi đĩa, nghĩa là, SELECT FOR UPDATE sửa đổi các hàng được chọn để đánh dấu chúng bị khóa, và vì thế sẽ gây ra việc ghi đĩa.

Bổ sung thêm vào các khóa hàng và bảng, các khóa chia sẻ/độc quyền mức trang được sử dụng để kiểm soát truy cập đọc/ghi vào các trang của bảng trong kho bộ nhớ đệm (buffer) được chia sẻ. các khóa đó được nhả ra ngay lập tức sau khi một hàng được lấy hoặc được cập nhật. Các lập trình viên ứng dụng thường không cần quan tâm với các khóa mức trang, mà chúng được nhắc ở đây cho đủ.

13.3.3. Khóa chết

Sử dụng việc khóa rõ ràng có thể làm gia tăng khả năng các khóa chết, trong đó 2 (hoặc nhiều hơn) giao dịch, mỗi giao dịch nắm các khóa mà giao dịch kia muốn. Ví dụ, nếu giao dịch 1 đòi hỏi một khóa độc quyền trong bảng A và sau đó cố có được một khóa độc quyền trong bảng B, trong khi giao dịch 2 có rồi bảng B được khóa độc quyền và bây giờ muốn một khóa độc quyền trong bảng A, thì không có giao dịch nào có thể tiến hành được. PostgreSQL tự dò tìm ra các tình huống khóa chết và giải quyết chúng bằng việc hủy bỏ một trong các giao dịch có liên quan, cho phép (các) giao dịch khác hoàn tất. (Chính xác giao dịch nào sẽ bị hủy bỏ là khó đoán trước và không nên dựa vào đó).

Lưu ý rằng các khóa chết cũng có thể xảy ra như là kết quả của các khóa mức hàng (và vì thế, chúng có thể xảy ra thậm chí nếu việc khóa rõ ràng không được sử dụng). Hãy cân nhắc trường hợp trong đó 2 giao dịch đồng thời cùng sửa một bảng. Giao dịch đầu thực thi:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

Điều này làm cho có một khóa mức hàng trong hàng với số tài khoản được chỉ định. Sau đó, giao dịch thứ 2 thực thi:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

Lệnh UPDATE đầu tiên có sự thành công một khóa mức hàng trong hàng được chỉ định, nên nó thành công trong việc cập nhật hàng đó. Tuy nhiên, lệnh UPDATE thứ 2 thấy rằng hàng mà nó định cập nhật đã bị khóa rồi, nên nó chờ cho giao dịch mà đã có khóa hoàn tất. Giao dịch thứ 2 bây giờ đang chờ giao dịch thứ nhất hoàn tất trước khi nó tiếp tục thực thi. Bây giờ, giao dịch 1 thực thi:

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

Giao dịch 1 cố để có một khóa mức hàng trong hàng được chỉ định, nhưng nó không thể: giao dịch 2 giữ rồi một khóa như vậy. Nên nó chờ cho giao dịch 2 hoàn tất. Vì thế, giao dịch 1 bị khóa trong giao dịch 2, và giao dịch 2 bị khóa trong giao dịch 1: một điều kiện khóa chết. PostgreSQL sẽ dò tìm ra tình huống này và hủy bỏ một trong các giao dịch đó.

Sự phòng vệ tốt nhất chống lại các khóa chết thường là tránh chúng bằng việc chắc chắn rằng tất cả các ứng dụng đang sử dụng một cơ sở dữ liệu có các khóa trong nhiều đối tượng theo một trật tự nhất quán. Trong ví dụ ở trên, nếu cả 2 giao dịch đã cập nhật các hàng theo cùng y hệt một trật tự, thì khóa chết có thể đã không xảy ra. Bạn cũng nên đảm bảo rằng khóa đầu có được trong một đối tượng trong một giao dịch là chế độ hạn chế nhất mà sẽ cần thiết cho đối tượng đó. Nếu không khả thi để kiểm tra điều này trước, thì các khóa chết có thể được nắm giữ trong quá trình xử lý bằng việc cố gắng làm lại các giao dịch hủy bỏ vì các khóa chết đó.

Miễn là không tình huống khóa trói nào được dò tìm ra, một giao dịch tìm cách hoặc một khóa mức bảng hoặc mức hàng sẽ chờ một cách xác định cho các khóa xung đột sẽ được thay thế. Điều này có nghĩa đây là một ý tưởng tồi cho các ứng dụng để giữ các giao dịch mở trong các giai đoạn thời gian dài (như, trong khi chờ đầu vào của người sử dụng).

13.3.4. Khóa cố vấn

PostgreSQL đưa ra một công cụ để tạo các khóa mà có các ý nghĩa do các ứng dụng định nghĩa. Chúng được gọi là các khóa cố vấn, vì hệ thống không ép sử dụng chúng - nó là tùy thuộc vào ứng dụng để sử dụng chúng một cách đúng đắn. Các khóa cố vấn có thể là hữu dụng cho các chiến lược khóa mà là một sự phù hợp vụng về đối với mô hình MVCC. Một khi có được, một khóa cố vấn được giữ cho tới khi được nhả ra rõ ràng hoặc kết thúc phiên. Không giống như các khóa tiêu chuẩn, các khóa cố vấn không tôn trọng các ngữ nghĩa của các giao dịch: một khóa có được trong quá trình một giao dịch mà sau đó được quay lại sẽ vẫn được giữ theo sau sự quay lại đó, và hơn nữa một sự mở khóa là có hiệu lực thậm chí nếu việc gọi giao dịch thất bại sau đó. Khóa y hệt có thể có được nhiều lần bằng qui trình của riêng nó: đối với từng yêu cầu khóa sẽ phải có một yêu cầu mở khóa tương ứng trước khi khóa đó thực sự được nhả ra. (Nếu một phiên giữ rồi một khóa được đưa ra, thì các yêu cầu bổ sung sẽ luôn thành công, thậm chí nếu các phiên khác đang chờ khóa đó). Giống như tất cả các khóa trong PostgreSQL, một danh sách hoàn chỉnh các khóa cố vấn hiện được bất kỳ phiên nào nắm giữ có thể được thấy trong kiểu nhìn hệ thống pg_locks.

Các khóa cố vấn được phân bổ ngoài kho bộ nhớ được chia sẻ mà kích cỡ của nó được xác định bằng các biến cấu hình max_locks_per_transaction và max_connections. Điều này đặt ra một giới hạn trên trong số các khóa cố vấn được máy chủ trao, điển hình trong số hàng chục tới hàng trăm

ngàn, phụ thuộc vào cách mà máy chủ đó được thiết lập cấu hình.

Sử dụng phổ biến các khóa cố vấn là để giả lập các chiến lược khóa bi quan thường của cái gọi là các hệ thống quản lý dữ liệu "tệp phẳng" (flat file). Trong khi một cờ được lưu giữ trong một bảng có thể được sử dụng cho cùng y hệt mục đích, thì các khóa cố vấn là nhanh hơn, tránh sự bung nở của MVCC, và tự động được máy chủ làm sạch vào cuối phiên. Trong các trường hợp nhất định việc sử dụng phương pháp khóa này, đặc biệt theo các yêu cầu có liên quan tới trật tự rõ ràng và các mệnh đề LIMIT, sự thận trọng phải được thực hiện để kiểm soát các khóa có được vì trật tự mà theo đó các biểu thức SQL được đánh giá. Ví dụ:

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger!
SELECT pg_advisory_lock(q.id) FROM
(
        SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

Trong các truy vấn ở trên, dạng thứ 2 là nguy hiểm vì LIMIT không được đảm bảo sẽ được áp dụng trước khi việc khóa hàm được thực thi. Điều này có thể dẫn tới một số khóa sẽ có được mà ứng dụng từng không kỳ vọng, và vì thế có thể hỏng để nhả ra (cho tới khi nó kết thúc phiên). Từ quan điểm của ứng dụng, các khóa như vậy có thể là lòng thòng, dù vẫn thấy được trong pg_locks.

Các hàm được đưa ra để điều khiển các khóa cố vấn được mô tả trong Bảng 9-61.

13.4. Kiểm tra sự nhất quán của dữ liệu ở mức ứng dụng

Vì các độc giả trong PostgreSQL không khóa dữ liệu, bất chấp mức cách li giao dịch, các dữ liệu được giao dịch đó đọc được có thể bị giao dịch đồng thời khác ghi đè. Nói cách khác, nếu một hàng được lệnh SELECT trả về thì nó không có nghĩa là hàng đó vẫn hiện hành ở thời điểm mà nó được trả về (nghĩa là, lúc nào đó sau khi truy vấn hiện hành đã bắt đầu). Hàng đó có thể đã được sửa đổi hoặc đã bị xóa từ một giao dịch được thực hiện xong rồi mà đã thực hiện sau khi lệnh SELECT đã bắt đầu. Thậm chí nếu hàng đó vẫn còn hợp lệ "bây giờ", thì nó có thể đã bị thay đổi hoặc bị xóa trước khi giao dịch hiện hành tiến hành một thực hiện xong hoặc quay lại.

Cách khác để nghĩ về nó là từng giao dịch thấy một hình chụp các nội dung cơ sở dữ liệu, và việc đồng thời thực thi các giao dịch có thể nhìn thất rất tốt các hình chụp khác. Vì thế toàn bộ khái niệm "bây giờ" là thứ gì đó được xác định tồi bằng mọi cách. Điều này không thường là một vấn đề lớn nếu các ứng dụng máy trạm được cách li với nhau, nhưng nếu các máy trạm có thể giao tiếp thông qua các kênh bên ngoài cơ sở dữ liệu thì sự lúng túng nghiêm trọng có thể diễn ra sau đó.

Để đảm bảo tính hợp lệ hiện hành của một hàng và bảo vệ nó khỏi các cập nhật đồng thời thì phải sử dụng các lệnh SELECT FOR UPDATE, SELECT FOR SHARE, hoặc một lệnh LOCK TABLE phù hợp. (khóa SELECT FOR UPDATE and SELECT FOR SHARE chỉ khóa các hàng được trả về đối với các cập nhật đồng thời, trong khi LOCK TABLE khóa toàn bộ bảng). Điều này phải được tính tới khi chuyển các ứng dụng sang PostgreSQL từ các môi trường khác.

Các kiểm tra hợp lệ tổng thể đòi hỏi suy nghĩ thêm theo MVCC. Ví dụ, một ứng dụng ngân hàng có

thể muốn kiểm tra rằng tổng tất cả các tín dụng trong một bảng bằng tổng các khoản nợ trong bảng khác, khi cả 2 bảng đều đang được cập nhật tích cực. So sánh các kết quả của 2 lệnh SELECT sum(...) kế tiếp sẽ không làm việc tin cậy trong chế độ Đọc thực hiện được, vì truy vấn thứ 2 có khả năng sẽ bao gồm các kết quả của các giao dịch không được truy vấn đầu tính tới. Việc tính cả 2 tổng trong một giao dịch tuần tự duy nhất sẽ trao một bức tranh chính xác của chỉ các hiệu ứng của các giao dịch được thực hiện xong rồi trước khi giao dịch tuần tự đã bắt đầu - nhưng bạn có thể nghi ngờ hợp lệ liệu câu trả lời có vẫn còn phù hợp vào thời điểm nó được phân phối hay không. Nếu bản thân giao dịch tuần tự đó đã áp dụng một số thay đổi trước khi cố gắng thực hiện sự kiểm tra nhất quán đó, thì tính hữu dụng của kiểm tra đó thậm chí trở nên tranh cãi hơn, vì bây giờ nó bao gồm một số nhưng không phải tất cả các thay đổi sau khi giao dịch đã bắt đầu. Trong các trường hợp như vậy thì một người cẩn thận có thể muốn khóa tất cả các bảng cần thiết để kiểm tra, để có được một bức tranh không gây tranh cãi về thực tế hiện hành. Một khóa chế độ SHARE (hoặc cao hơn) đảm bảo rằng không có những thay đổi không được thực hiện trong bảng bị khóa đó, khác với những thay đổi của giao dịch hiện hành.

Cũng lưu lý rằng nếu bạn đang dựa vào việc khóa rõ ràng để ngăn chặn các thay đổi đồng thời, thì bạn nên hoặc sử dụng chế độ Đọc thực hiện được, hoặc chế độ tuần tự và hãy thận trọng để có được các khóa trước khi tiến hành các truy vấn. Một khóa có được bằng một giao dịch tuần tự đảm bảo rằng không giao dịch nào khác sửa đổi được bảng vẫn còn đang chạy, nhưng nếu hình chụp thấy được từ giao dịch có trước khi giành được khóa đó, thì nó có thể có trước một số thay đổi bây giờ được thực hiện trong bảng đó. Một hình chụp các giao dịch tuần tự thực sự bị đóng băng ở đầu truy vấn đầu tiên hoặc lệnh sửa đổi dữ liệu của nó (SELECT, INSERT, UPDATE, hoặc DELETE), nên có khả năng giành được các khóa rõ ràng trước khi hình chụp đó được đóng băng.

13.5. Khóa và chỉ số

Dù PostgreSQL đưa ra sự truy cập đọc/ghi không khóa cho dữ liệu bảng, thì sự truy cập đọc/ghi không khóa hiện không được chào cho mọi phương pháp truy cập chỉ số được triển khai trong PostgreSQL. Các dạng chỉ số khác nhau được điều khiển như sau:

Các chỉ số B-tree và GiST

> Các khóa mức trang chia sẻ/độc quyền ngắn hạn được sử dụng cho truy cập đọc/ghi. Các khóa sẽ được nhả ra ngay lập tức sau khi từng hàng chỉ số được lấy hoặc được chèn. Các dạng chỉ số đó đưa ra sự đồng thời cao nhất mà không có các điều kiện khóa chết.

Các chỉ số Hash (băm)

> Các khóa mức xô (bucket) băm chia sẻ/độc quyền được sử dụng cho truy cập đọc/ghi. Các khóa sẽ được nhả ra sau khi toàn bộ xô được xử lý. Các khóa mức xô đưa ra sự đồng thời tốt hơn so với các khóa mức chỉ số, nhưng khóa chết là có khả năng vì các khóa được giữ lâu hơn so với một hoạt động chỉ số.

Các chỉ số GIN

Các khóa mức trang chia sẻ/độc quyền ngắn hạn được sử dụng cho truy cập đọc/ghi. Các khóa được nhả ra ngay lập tức sau khi từng hàng chỉ số được lấy hoặc được chèn. Nhưng lưu ý rằng sự chèn một giá trị được đánh chỉ số GIN thường tạo ra vài sự chèn khóa chỉ số cho một hàng, nên GIN có thể tiến hành công việc đáng kể cho một sự chèn giá trị duy nhất.

Hiện hành, các chỉ số B-tree đưa ra hiệu năng tốt nhất cho các ứng dụng đồng thời; vì chúng cũng có các chức năng nhiều hơn so với các chỉ số băm - hash, chúng là dạng chỉ số được khuyến cáo cho các ứng dụng đồng thời mà cần phải đánh chỉ số cho các dữ liệu vô hướng. Khi làm việc với các dữ liệu không vô hướng, các B-tree sẽ không hữu dụng, và các chỉ số GiST và GIN sẽ được sử dụng thay vào đó.

Chương 14. Mẹo cho hiệu năng

Hiệu năng của truy vấn có thể bị ảnh hưởng vì nhiều thứ. Một số chúng có thể được người sử dụng kiểm soát, trong khi những thứ khác là cơ bản đối với thiết kế nằm bên dưới của hệ thống. Chương này đưa ra một số gợi ý về việc hiểu và tinh chỉnh hiệu năng của PostgreSQL.

14.1. Sử dụng EXPLAIN

PostgreSQL sắp đặt một kế hoạch của các truy vấn cho từng truy vấn nó nhận được. Việc chọn kế hoạch đúng để khớp với cấu trúc truy vấn và các thuộc tính dữ liệu là tuyệt đối sống còn cho hiệu năng tốt, vì thế hệ thống bao gồm một *trình hoạch định* phức tạp mà cố chọn các kế hoạch tốt. Bạn có thể sử dụng lệnh EXPLAIN để xem kế hoạch truy vấn gì trình hoạch định đó tạo ra cho bất kỳ truy vấn nào. Việc đọc kế hoạch là một nghệ thuật mà đáng một sách chỉ dẫn tăng cường; mà ở đây có một số thông tin cơ bản.

Cấu trúc kế hoạch của một truy vấn là một cây các *nút kế hoạch*. Các nút ở mức đáy của cây là các nút quét bảng: chúng trả về các hàng thô từ một bảng. Có các dạng khác nhau của các nút quét cho các phương pháp truy cập bảng khác nhau: các quét tuần tự, quét chỉ số và quét chỉ số bitmap. Nếu truy vấn đó đòi hỏi các hoạt động chung, tổng hợp, sắp xếp hoặc khác trong các hàng thô, thì sẽ có các nút bổ sung thêm ở trên mà các nút quét sẽ thực hiện các hoạt động đó. Một lần nữa, thường có nhiều hơn 1 cách thức có thể để thực hiện các hoạt động đó, nên các dạng nút khác nhau có thể cũng xuất hiện ở đây. Đầu ra của EXPLAIN có một dòng cho từng nút trong cây kế hoạch đó, chỉ ra dạng nút cơ bản cộng với các ước tính chi phí mà trình hoạch định thực hiện cho sự thực thi nút kế hoạch đó. Dòng đầu tiên (nút ở đỉnh) có chi phí thực thi tổng được ước lượng cho kế hoạch đó; đó là số lượng mà trình hoạch định tìm cách làm giảm thiểu.

Đây là một ví dụ tầm thường, chỉ để chỉ ra những gì đầu ra trông giống[1]:

```
EXPLAIN SELECT * FROM tenk1;

QUERY PLAN
-------------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)
```

Các số nằm trong ngoặc của EXPLAIN là (từ trái qua phải):

- chi phí khởi đầu ước tính (thời gian bỏ ra trước khi quét đầu ra có thể bắt đầu, nghĩa là, thời gian để thực hiện việc sắp xếp trong một nút sắp xếp).
- chi phí tổng ước tính (nếu tất cả các hàng được truy xuất, dù chúng có thể sẽ không; nghĩa là, một truy vấn với một mệnh đề LIMIT sẽ sớm dừng việc thanh toán chi phí tổng của nút đầu ra của nút kế hoạch Limit.

1 Các ví dụ trong phần này được thiết kế từ cơ sở dữ liệu kiểm thử sự thoái lui sau khi thực hiện một VACUUM ANALYZE, bằng việc sử dụng các nguồn phát triển phiên bản 8.2. Bạn nên có khả năng có các kết quả tương tự nếu bạn tự mình thử các ví dụ đó, nhưng các chi phí ước lượng và các tính toán hàng của bạn có thể hơi khác vì các số liệu thống kê của ANALYZE là các mẫu ngẫu nhiên hơn là chính xác.

- số các hàng đầu ra được ước tính của nút kế hoạch này (một lần nữa, chỉ nếu được thực thi tới hoàn tất).
- độ rộng trung bình ước tính (theo byte) các hàng đầu ra của nút kế hoạch này.

Các chi phí được đo đếm trong các đơn vị tùy ý được các tham số chi phí của trình hoạch định xác định (xem Phần 18.6.2). Thực tiễn truyền thống là để đo đếm các chi phí trong các đơn vị lấy các trang của đĩa; đó là, seq_page_cost là tập hợp theo qui ước tới 1.0 và các tham số chi phí khác được thiết lập tương đối với điều đó. (Các ví dụ trong phần này được chạy với các tham số chi phí mặc định).

Điều quan trọng để lưu ý rằng chi phí của một nút mức cao hơn bao gồm chi phí của tất cả các nút con của nó. Cũng quan trọng để nhận thức được rằng chi phí đó chỉ phản ánh những điều mà trình hoạch định quan tâm. Trong thực tế, chi phí đó không xem xét thời gian bỏ ra cho việc truyền các hàng kết quả tới máy trạm, nó có thể là một yếu tố quan trọng trong thời gian thực đã trôi qua; nhưng trình hoạch định bỏ qua nó vì nó không thể thay đổi nó bằng việc điều chỉnh kế hoạch. (Mỗi kế hoặc đúng sẽ đưa ra kết quả tập hợp y hệt các hàng, chúng ta tin tưởng thế).

Giá trị các hàng là một mẹo vì nó không phải là số hàng được nút kế hoạch đó xử lý hoặc quét. Nó thường ít hơn, phản ánh khả năng lựa chọn được ước lượng của bất kỳ điều kiện mệnh đề WHERE nào mà đang được áp dụng ở nút đó. Lý tưởng là ước tính các hàng mức đỉnh sẽ xấp xỉ số lượng các hàng thực sự được trả về, được cập nhật hoặc bị xóa bởi truy vấn đó.

Chạy với ví dụ của chúng tôi:

```
EXPLAIN SELECT * FROM tenk1;

QUERY PLAN
-------------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)
```

Điều này là đơn giản như nó có. Nếu bạn làm:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

thì bạn sẽ thấy rằng tenk1 có 358 trang đĩa và 10.000 hàng. Chi phí được ước lượng đó được tính toán như là (các trang đĩa được đọc * seq_page_cost) + (các hàng được quét * cpu_tuple_cost). Mặc định, seq_page_cost bằng 1.0 và cpu_tuple_cost bằng 0.01, nên chi phí được ước lượng là:

(358 * 1.0) + (10.000 * 0.01) = 458.

Bây giờ hãy sửa truy vấn gốc để bổ sung thêm một điều kiện WHERE:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

QUERY PLAN
-------------------------------------------------------------
Seq Scan on tenk1 (cost=0.00..483.00 rows=7033 width=244)
Filter: (unique1 < 7000)
```

Lưu ý rằng đầu ra EXPLAIN chỉ mệnh đề WHERE đang được áp dụng như một điều kiện "bộ lọc"; điều này có nghĩa là nút kế hoạch kiểm tra điều kiện cho từng hàng mà nó quét, và các đầu ra chỉ là các

hàng vượt qua được điều kiện đó. Ước lượng các hàng đầu ra đã giảm vì mệnh đề WHERE. Tuy nhiên, sự quét sẽ vẫn phải tới tất cả 10.000 hàng, nên chi phí sẽ không giảm; trong thực tế nó đã vượt qua được một chút (bằng 10.000 * cpu_operator_cost, một cách chính xác) để phản ánh thời gian dôi dư mà CPU bỏ ra kiểm tra điều kiện WHERE.

Số hàng thực sự truy vấn này có thể lựa chọn là 7.000, nhưng ước lượng rows chỉ là gần đúng. Nếu bạn cố đúp bản thí điểm này, thì bạn sẽ có khả năng có một ước lượng hơi khác; hơn nữa, nó sẽ thay đổi sau từng lệnh ANALYZE, vì các số liệu thống kê được ANALYZE tạo ra được lấy từ một mẫu của bảng được tùy biến.

Bây giờ, hãy thực hiện điều kiện chặt chẽ hơn:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

QUERY PLAN
------------------------------------------------------------------------------
Bitmap Heap Scan on tenk1 (cost=2.37..232.35 rows=106 width=244)
        Recheck Cond: (unique1 < 100)
        -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0)
                Index Cond: (unique1 < 100)
```

Trình hoạch định này đã quyết định sử dụng một kế hoạch 2 bước: nút kế hoạch đáy viếng thăm một chỉ số để tìm các vị trí các hàng khớp với điều kiện của chỉ số đó, và sau đó nút kế hoạch cao hơn thực sự lấy các hàng đó từ bản thân bảng đó. Việc lấy các hàng một cách riêng rẽ sẽ đắt giá hơn so với việc đọc chúng tuần tự, nhưng vì không phải tất cả các trang của bảng phải được viếng thăm, nên nút kế hoạch cao hơn sẽ sắp xếp các vị trí hàng được chỉ số nhận diện theo trật tự vật lý trước khi đọc chúng, để làm giảm thiểu chi phí lấy được riêng rẽ. "Bitmap" được nhắc tới trong các tên nút là cơ chế thực hiện việc sắp xếp.

Nếu điều kiện WHERE là lựa chọn đủ, thì trình hoạch định có thể chuyển sang một kế hoạch quét chỉ số "đơn giản":

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3;

QUERY PLAN
------------------------------------------------------------------------------
Index Scan using tenk1_unique1 on tenk1 (cost=0.00..10.00 rows=2 width=244)
Index Cond: (unique1 < 3)
```

Trong trường hợp này các hàng của bảng được lấy theo trật tự chỉ số, nó làm cho chúng thậm chí đắt giá hơn để đọc, nhưng có quá ít hàng mà các chi phí dư thừa để sắp xếp vị trí hàng là không đáng điều này. Bạn hầu như thường thấy dạng kế hoạch này đối với các truy vấn mà chỉ lấy một hàng duy nhất, và cho cả các truy vấn mà có điều kiện ORDER BY khớp với trật tự chỉ số đó.

Bổ sung thêm điều kiện khác vào mệnh đề WHERE:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 3 AND stringu1 = 'xxx';

QUERY PLAN
------------------------------------------------------------------------------
Index Scan using tenk1_unique1 on tenk1 (cost=0.00..10.01 rows=1 width=244)
        Index Cond: (unique1 < 3)
        Filter: (stringu1 = 'xxx'::name)
```

Điều kiện được bổ sung thêm stringu1 = 'xxx' làm giảm ước lượng các hàng đầu ra, nhưng không phải là chi phí vì chúng ta vẫn phải viếng thăm tập hợp các hàng y hệt. Lưu ý rằng mệnh đề stringu1 không thể được áp dụng như một điều kiện chỉ số (vì chỉ số này chỉ trong cột unique1). Thay vào đó nó được áp dụng như một bộ lọc trong các hàng được chỉ số đó truy xuất. Vì thế chi phí thực sự đã hơi lên một chút để phản ánh việc kiểm tra thêm này.

Nếu có các chỉ số trong vài cột được tham chiếu trong WHERE, thì trình hoạch định có thể chọn để sử dụng sự kết hợp của một AND hoặc OR của các chỉ số:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

QUERY PLAN
-------------------------------------------------------------------------------
Bitmap Heap Scan on tenk1 (cost=11.27..49.11 rows=11 width=244)
        Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
        -> BitmapAnd (cost=11.27..11.27 rows=11 width=0)
                -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0)
                        Index Cond: (unique1 < 100)
                -> Bitmap Index Scan on tenk1_unique2 (cost=0.00..8.65 rows=1042 width=0)
                        Index Cond: (unique2 > 9000)
```

Nhưng điều này đòi hỏi việc viếng thăm cả hai chỉ số, vì vậy không nhất thiết phải là một chiến thắng so với việc sử dụng chỉ một chỉ số và đối xử với điều kiện khác như với một bộ lọc. Nếu bạn làm thay đổi các dải có liên quan thì bạn sẽ thấy sự thay đổi kế hoạch một cách tương ứng.

Hãy thử gộp 2 bảng, bằng việc sử dụng các cột mà chúng ta đã và đang thảo luận:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

QUERY PLAN
------------------------------------------------------------------------------
Nested Loop (cost=2.37..553.11 rows=106 width=488)
        -> Bitmap Heap Scan on tenk1 t1 (cost=2.37..232.35 rows=106 width=244)
                Recheck Cond: (unique1 < 100)
                -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0)
                        Index Cond: (unique1 < 100)
        -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.00..3.01 rows=1 width=244)
                Index Cond: (t2.unique2 = t1.unique2)
```

Trong kết hợp có vòng lặp lồng này, sự quét ngoài (cao hơn) là sự quét chỉ số bitmap y hệt mà chúng ta đã thấy trước đó, và vì thế chi phí và tính toán hàng của nó là y hệt vì chúng ta đang áp dụng mệnh đề WHERE cho unique1 < 100 ở nút đó. Mệnh đề t1.unique2 = t2.unique2 là chưa phù hợp, nên nó không ảnh hưởng tới tính hàng của sự quét ngoài. Đối với sự quét trong (thấp hơn), thì giá trị unique2 của hàng quét ngoài được cài cắm vào sự quét chỉ số bên trong để tạo ra một điều kiện chỉ số như t2.unique2 = constant. Vì thế chúng ta có kế hoạch quét bên trong y hệt và các chi phí mà chúng ta đã có từ EXPLAIN SELECT * FROM tenk2 WHERE unique2 = 42. Các chi phí của nút lặp sau đó được thiết lập trên cơ sở chi phí của sự quét ngoài, cộng với sự lặp lại của sự quét trong cho từng hàng ngoài (106 * 3.01 ở đây), cộng với một chút thời gian CPU cho việc xử lý chung.

Trong ví dụ này sự tính hàng đầu ra chung là y hệt như sản phẩm của 2 sự tính toán hàng quét, mà điều đó không đúng trong tất cả các trường hợp vì bạn có thể có các mệnh đề WHERE mà nhắc tới cả

2 bảng và vì thế chỉ có thể được áp dụng ở điểm chung, không cho sự quét đầu vào. Ví dụ, nếu chúng ta đã thêm WHERE ... AND t1.hundred < t2.hundred, thì điều đó có thể làm giảm sự tính hàng đầu ra của nút chung đó, nhưng không thay đổi sự quét đầu vào.

Một cách để xem xét các kế hoạch khác nhau là hãy ép trình hoạch định bỏ qua bất kỳ chiến lược nào mà nó nghĩ từng là rẻ nhất, bằng việc sử dụng các cờ có/không hiệu lực (enable/disable) được mô tả trong Phần 18.6.1. (Đây là một công cụ thô, nhưng hữu dụng. Xem thêm Phần 14.3).

```
SET enable_nestloop = off;
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

QUERY PLAN
------------------------------------------------------------------------------------------
Hash Join (cost=232.61..741.67 rows=106 width=488)
        Hash Cond: (t2.unique2 = t1.unique2)
        -> Seq Scan on tenk2 t2 (cost=0.00..458.00 rows=10000 width=244)
        -> Hash (cost=232.35..232.35 rows=106 width=244)
                -> Bitmap Heap Scan on tenk1 t1 (cost=2.37..232.35 rows=106 width=244)
                        Recheck Cond: (unique1 < 100)
                        -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0)
                                Index Cond: (unique1 < 100)
```

Kế hoạch này xử lý để trích ra 100 hàng có quan tâm của tenk1 bằng việc sử dụng sự quét chỉ số cũ y hệt, cất chúng vào một bảng băm trong bộ nhớ, và sau đó tiến hành một sự quét tuần tự của tenk2, thăm dò trong bảng băm đối với khả năng lấy của t1.unique2 = t2.unique2 đối với từng hàng tenk2. Chi phí để đọc tenk1 và thiết lập bảng băm là một chi phí khởi đầu cho sự kết hợp băm, vì sẽ không có đầu ra cho tới khi chúng ta có thể bắt đầu đọc tenk2. Ước tính thời gian tổng cho sự kết hợp cũng bao gồm một khoản chi phí khổng lồ cho thời gian CPU để thăm dò bảng băm 10.000 lần. Tuy nhiên, hãy lưu ý rằng chúng ta không đang lấy 10.000 lần của 232.35; thiết lập bảng băm chỉ được thực hiện một lần theo dạng kế hoạch này.

Có khả năng để kiểm tra độ chính xác các chi phí ước lượng của trình hoạch định bằng việc sử dụng EXPLAIN ANALYZE. Lệnh này thực sự thực thi truy vấn, và sau đó hiển thị thời gian chạy đúng được tích tụ bên trong từng nút kế hoạch cùng với các chi phí y hệt được ước lượng mà một kế hoạch EXPLAIN chỉ ra. Ví dụ, chúng ta có thể có một kết quả như thế này:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
Nested Loop (cost=2.37..553.11 rows=106 width=488) (actual time=1.392..12.700 rows=100
        -> Bitmap Heap Scan on tenk1 t1 (cost=2.37..232.35 rows=106 width=244) (actual
        time=Recheck Cond: (unique1 < 100)
                -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0) (actual
                Index Cond: (unique1 < 100)
        -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.00..3.01 rows=1 width=244) (actual
        Index Cond: (t2.unique2 = t1.unique2)
Total runtime: 14.452 ms
```

Lưu ý rằng các giá trị "thời gian thực" là theo mili giây của thời gian thực, trong khi các ước tính

chi phí được thể hiện theo các đơn vị tùy ý; vì thế chúng có lẽ không khớp. Điều phải chú ý tới là liệu các tỷ lệ thời gian thực và các chi phí được ước lượng có là nhất quán hay không.

Trong một số kế hoạch truy vấn, là không thể đối với một nút kế hoạch con sẽ được thực thi hơn một lần. Ví dụ, sự quét chỉ số bên trong được thực thi một lần cho từng hàng bên ngoài trong kế hoạch có lặp lồng nhau ở trên. Trong các trường hợp như vậy, giá trị loops nêu lên tổng số các thực thi của nút đó, và thời gian thực sự và các giá trị hàng được trình bày là trung bình cho mỗi sự thực thi. Điều này được thực hiện để làm cho các số so sánh được với cách thức mà các ước tính chi phí được trình bày. Nhân với giá trị loops để có được tổng thời gian thực sự được bỏ ra trong nút đó.

For INSERT , UPDATE , and DELETE commands, the time spent applying the table changes is charged to a top-level Insert, Update, or Delete plan node

Tổng thời gian Total runtime được EXPLAIN ANALYZE chỉ ra bao gồm thời gian khởi động và tắt của trình thực thi, chứ không phải của việc phân tích, viết lại, hay thời gian lên kế hoạch. Đối với các lệnh INSERT, UPDATE, và DELETE, thời gian bỏ ra cho việc áp dụng những thay đổi của bảng được áp cho nút kế hoạch của một mức đỉnh các lệnh Insert, Update hoặc Delete. (Các nút kế hoạch bên dưới nút này trình bày công việc định vị các hàng và/hoặc việc tính các hàng mới). Thời gian bỏ ra cho việc thực thi BEFORE các trigger, nếu có, được lấy chi phí đối với nút Insert, Update hoặc Delete có liên quan, dù thời gian bỏ ra cho việc thực thi AFTER các trigger là không lấy. Thời gian bỏ ra trong từng trigger (hoặc BEFORE hoặc AFTER) cũng được bày ra riêng rẽ và được đưa vào trong thời gian chạy tổng. Tuy nhiên, lưu ý rằng các trigger ràng buộc bị hoãn lại sẽ không được thực thi cho tới kết thúc giao dịch và vì thế không được EXPLAIN ANALYZE bày ra.

Có 2 cách đáng kể trong đó các thời gian chạy được EXPLAIN ANALYZE đo đếm có thể đi chệch khỏi sự thực thi thông thường của truy vấn y hệt. Trước hết, vì không hàng đầu ra nào được phân phối cho máy trạm, nên các chi phí truyền qua mạng và các chi phí định dạng I/O không được đưa vào. Thứ 2, tổng chi phí được EXPLAIN ANALYZE thêm vào có thể là đáng kể, đặc biệt trong các máy với các lời gọi nhân (kernel) gettimeofday() chậm.

Đáng lưu ý rằng các kết quả của EXPLAIN nên không bị ngoại suy tới các tình huống khác với tình huống mà bạn thực sự đang kiểm thử; ví dụ, các kết quả trong một bảng kích cỡ trò chơi không thể được giả thiết để áp dụng cho các bảng lớn. Các ước tính chi phí của trình hoạch định là không tuyến tính và vì thế nó có thể chọn một kế hoạch khác đối với một bảng lớn hơn hoặc nhỏ hơn. Một ví dụ cực kỳ là trong một bảng mà chỉ chiếm một trang đĩa, bạn sẽ gần như luôn có một kế hoạch quét tuần tự bất kể các chỉ số là sẵn sàng hay không. Trình hoạch định nhận thức được rằng nó sẽ lấy một trang đĩa đọc để xử lý bảng đó trong bất kỳ trường hợp nào, nên không có giá trị nào trong trang thêm vào đọc để xem một chỉ số.

14.2. Số liệu thống kê được trình hoạch định sử dụng

Như chúng ta đã thấy trong phần trước, trình hoạch định truy vấn cần ước lượng số các hàng được một truy vấn truy xuất để tiến hành các lựa chọn tốt đối với các kế hoạch truy vấn. Phần này lướt nhanh qua các số liệu thống kê mà hệ thống sử dụng cho các ước tính đó.

Một thành phần của số liệu thống kê là tổng số các khoản đầu vào trong từng bảng và chỉ số, cũng như số lượng các khối đĩa bị từng bảng và chỉ số chiếm. Thông tin này được giữ trong pg_class của bảng, trong các cột reltuples và relpages. Chúng ta có thể xem nó với các truy vấn tương tự thế này:

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

relname                | relkind | reltuples | relpages
-----------------------+---------+-----------+----------
tenk1                  | r       |   10000 | 358
tenk1_hundred          | i       |   10000 |  30
tenk1_thous_tenthous   | i       |   10000 |  30
tenk1_unique1          | i       |   10000 |  30
tenk1_unique2          | i       |   10000 |  30
(5 rows)
```

Ở đây chúng ta có thể thấy rằng tenk1 chứa 10.000 hàng, các chỉ số của nó cũng vậy, nhưng các chỉ số là (không ngạc nhiên) nhỏ hơn nhiều so với bảng.

Vì các lý do hiệu quả, reltuples và relpages không được cập nhật lúc đang chạy, và vì thế chúng thường bao gồm thứ gì đó như các dữ liệu lỗi thời. Chúng được VACUUM, ANALYZE và một ít lệnh DDL như CREATE INDEX cập nhật. Một lệnh ANALYZE đứng một mình, nó là một phần của VACUUM, sinh ra một giá trị gần với reltuples vì nó không đọc mọi hàng trong bảng. Trình hoạch định sẽ mở rộng phạm vi các giá trị mà nó thấy trong pg_class để khớp với kích cỡ bảng vật lý hiện hành, vì thế có được sự gần đúng sát hơn.

Hầu hết các truy vấn chỉ truy xuất một phần các hàng trong một bảng, vì các mệnh đề WHERE hạn chế các hàng sẽ được kiểm tra. Trình hoạch định vì thế cần phải thực hiện một ước lượng khả năng lựa chọn các mệnh đề WHERE, đó là, một phần các hàng khớp với từng điều kiện trong mệnh đề WHERE. Thông tin này được sử dụng cho tác vụ này được lưu giữ trong catalog hệ thống pg_statistic. Các khoản đầu vào pg_statistic được các lệnh ANALYZE và VACUUM ANALYZE cập nhật, và luôn gần đúng thậm chí khi được cập nhất tươi mới.

Thay vì xem trực tiếp pg_statistic, là tốt hơn để xem kiểu nhìn pg_stats của nó khi xem xét các số liệu thống kê một cách thủ công. pg_stats được thiết kế để được đọc dễ dàng hơn. Hơn nữa, ai cũng đọc được pg_stats, trong khi chỉ một mình siêu người sử dụng (superuser) đọc được pg_statistic. (Điều này ngăn cản những người sử dụng không có quyền khỏi việc học thứ gì đó về các nội dung các bảng của những người khác từ các số liệu thống kế đó. Kiểu nhìn pg_stats được đăng ký để chỉ ra chỉ các hàng về các bảng mà người sử dụng hiện hành có thể đọc). Ví dụ, chúng ta có thể làm:

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

attname | inherited | n_distinct          | most_common_vals
--------+-----------+---------------------+-----------------------------------
name    | f         |         -0.363388 | I- 580 Ramp+
        |           |                     | I- 880 Ramp+
        |           |                     | Sp Railroad +
        |           |                     | I- 580 +
```

```
        |          |                 | I- 680 Ramp
name    | t        |      -0.284859  | I- 880 Ramp+
        |          |                 | I- 580 Ramp+
        |          |                 | I- 680 Ramp+
        |          |                 | I- 580 +
        |          |                 | State Hwy 13 Ramp
(2 rows)
```

Lưu ý rằng 2 hàng được hiển thị cho cùng cột y hệt, một hàng tương ứng với hệ thống phân cấp kế thừa hoàn chỉnh bắt đầu ở bảng road (inherited=t), và hàng khác chỉ bao gồm bản thân bảng road (inherited=f).

Lượng thông tin được ANALYZE lưu trữ trong pg_statistic, đặc biệt số lượng tối đa các khoản đầu vào trong các mảng most_common_vals và histogram_bounds cho từng cột, có thể được thiết lập trên cơ sở từng cột một bằng việc sử dụng lệnh ALTER TABLE SET STATISTICS, hoặc toàn thể bằng việc thiết lập biến cấu hình default_statistics_target. Giới hạn mặc định hiện hành là 100 khoản đầu vào. Việc nâng lên giới hạn này có thể cho phép nhiều ước tính chính xác hơn của trình hoạch định được thực hiện, đặc biệt cho các cột với các phân phối dữ liệu bất thường, với giá thành của việc tiêu dùng nhiều không gian hơn trong pg_statistic và hơi nhiều thời gian hơn một chút để tính toán các ước lượng đó. Ngược lại, một giới hạn thấp hơn có thể là đủ cho các cột với các phân phối dữ liệu đơn giản.

Chi tiết xa hơn về sử dụng các số liệu thống kê của trình hoạch định có thể thấy trong Chương 56.

14.3. Kiểm soát trình hoạch định với mệnh đề rõ ràng JOIN

Có khả năng kiểm soát trình hoạch định truy vấn ở một vài mức độ bằng việc sử dụng cú pháp rõ ràng JOIN. Để xem vì sao điều này là cần thiết, trước hết chúng ta cần một số nền tảng cơ bản.

Trong một truy vấn liên kết đơn giản, như:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

trình hoạch định là tự do để kết nối các bảng được đưa ra theo bất kỳ thứ tự nào. Ví dụ, nó có thể tạo một kế hoạch truy vấn mà kết nối A với B, bằng việc sử dụng điều kiện WHERE a.id = b.id, sau đó kết nối C với bảng được kết nối này, bằng việc sử dụng điều kiện WHERE khác. Hoặc nó có thể kết nối B với C và sau đó kết nối A vào kết quả đó. Hoặc nó có thể kết nối A với C và sau đó kết nối chúng với B - nhưng điều đó có thể là không đủ, vì sản phẩm Đề các (Cartesian) của A và C có thể phải được hình thành, đang không có điều kiện áp dụng được trong mệnh đề WHERE để cho phép sự tối ưu hóa liên kết đó. (Tất cả các liên kết trong trình thực thi PostgreSQL xảy ra giữa 2 bảng đầu vào, nên là cần thiết để xây dựng kết quả theo một trong các cách thức đó). Điểm quan trọng là những khả năng liên kết đó đưa ra các kết quả tương đương về ngữ nghĩa nhưng có thể có các chi phí thực thi khác nhau khổng lồ. Vì thế, trình hoạch định sẽ khai thác tất cả chúng để cố tìm ra kế hoạch truy vấn có hiệu quả nhất.

Khi một truy vấn chỉ liên quan tới 2 hoặc 3 bảng, thì sẽ không có nhiều thứ tự liên kết để lo lắng.

Nhưng số lượng các thứ tự có khả năng liên kết tăng theo hàm mũ khi số lượng các bảng gia tăng. Ngoài khoảng 10 bảng đầu vào thì không còn là thực tế nữa để thực hiện một tìm kiếm đầy đủ của tất cả các khả năng, và thậm chí đối với 6 hoặc 7 bảng thì việc lên kế hoạch có thể mất khá nhiều thời gian. Khi có quá nhiều các bảng đầu vào, thì trình hoạch định của PostgreSQL sẽ chuyển từ tìm kiếm đầy đủ sang một tìm kiếm chung xác suất thông qua một số khả năng giới hạn. (Ngưỡng để chuyển qua được tham số thời gian chạy geqo_threshold thiết lập). Tìm kiếm chung mất ít thời gian hơn, nhưng nó sẽ không nhất thiết tìm kế hoạch có khả năng tốt nhất.

Khi truy vấn liên quan tới các liên kết, thì trình hoạch định có ít sự tự do hơn so với nó có cho các liên kết thô (nội bộ). Ví dụ, hãy xem xét:

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Dù các hạn chế của truy vấn này là hơi tương tự với ví dụ trước, thì ngữ nghĩa là khác vì một hàng phải được đưa ra cho từng hàng của A mà không có hàng trùng khớp trong liên kết của B và C. Vì thế trình hoạch định không có sự lựa chọn thứ tự liên kết ở đây: nó phải liên kết B với C và sau đó liên kết A vào kết quả đó. Tương tự, truy vấn này lấy ít thời gian hơn so với truy vấn trước đó. Trong các trường hợp khác, trình hoạch định có khả năng xác định rằng hơn một thứ tự liên kết là an toàn. Ví dụ, đưa ra:

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

là hợp lệ để liên kết A với hoặc B hoặc C trước. Hiện hành, chỉ FULL JOIN hoàn toàn ràng buộc thứ tự liên kết. Hầu hết các trường hợp thực tiễn có liên quan tới LEFT JOIN hoặc RIGHT JOIN có thể được dàn xếp ở một vài mức độ nào đó.

Cú pháp liên kết nội bộ rõ ràng (INNER JOIN, CROSS JOIN, hoặc unadorned JOIN), về mặt ngữ nghĩa là y hệt như việc liệt kê các quan hệ đầu vào trong FROM, vì thế không ràng buộc thứ tự liên kết.

Thậm chí dù hầu hết các dạng JOIN không hoàn toàn ràng buộc thứ tự liên kết, có khả năng ra lệnh cho trình hoạch định truy vấn của PostgreSQL đối xử với tất cả các mệnh đề JOIN như việc ràng buộc thứ tự liên kết vậy. Ví dụ, 3 truy vấn sau là tương đương nhau về logic:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Nhưng nếu chúng ta nói cho trình hoạch định tôn trọng thứ tự JOIN, thì dòng 2 và 3 mất thời gian để lên kế hoạch hơn so với dòng 1. Hiệu ứng này là không đáng lo ngại vì chỉ có 3 bảng, nhưng nó có thể là một sự cứu sinh với nhiều bảng.

Để ép trình hoạch định tuân theo thứ tự liên kết được đặt ra từ các JOIN rõ ràng, hãy thiết lập tham số thời gian chạy join_collapse_limit về 1. (Các giá trị có thể khác được thảo luận bên dưới).

Bạn không cần ràng buộc thứ tự liên kết hoàn toàn để cắt bỏ thời gian tìm kiếm, vì là OK để sử dụng các toán tử JOIN với các khái niệm của một danh sách đầy đủ FROM. Ví dụ, hãy xem xét:

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

Với join_collapse_limit = 1, điều này ép trình hoạch định phải liên kết A với B trước việc liên kết

chúng với các bảng khác, nhưng không ràng buộc các lựa chọn của nó. Trong ví dụ này, số các thứ tự liên kết có thể được giảm vì một yếu tố của 5.

Việc ràng buộc tìm kiếm của trình hoạch định theo cách này là một kỹ thuật hữu dụng cho việc giảm thời gian lên kế hoạch và cho việc dẫn hướng cho trình hoạch định tới một kế hoạch truy vấn tốt. Nếu trình hoạch định chọn một thứ tự liên kết tồi một cách mặc định, thì bạn có thể ép nó chọn một thứ tự tốt hơn thông qua cú pháp JOIN – giả thiết là bạn biết một thứ tự tốt hơn, là thế. Sự kiểm thử được khuyến cáo.

Một vấn đề có liên quan sát sao ảnh hưởng tới thời gian lên kế hoạch là việc sập các truy vấn con bên trong truy vấn cha.

Ví dụ, hãy xem xét:

```
SELECT *
FROM x, y,
        (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

Tình huống này có thể nảy sinh từ sử dụng một kiểu nhìn mà ràng buộc một liên kết; qui tắc SELECT của một kiểu nhìn sẽ được chèn vào chỗ của tham chiếu kiểu nhìn đó, bằng việc lấy một truy vấn rất giống ở trên. Thông thường, trình hoạch định sẽ cố sập truy vấn con trong truy vấn cha, có:

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

Điều này thường dẫn tới một kế hoạch tốt hơn so với việc lên kế hoạch cho truy vấn con một cách riêng rẽ. (Ví dụ, các điều kiện WHERE bên ngoài có thể là các điều kiện như việc liên kết X với A trước sẽ loại bỏ nhiều hàng của A, vì thế tránh được nhu cầu hình thành đầu ra đầy đủ logic của truy vấn con đó). Nhưng cùng lúc, chúng ta đã làm gia tăng thời gian lên kế hoạch; ở đây, chúng ta có vấn đề liên kết 5 cách thức đang thay thế 2 vấn đề liên kết 3 cách thức riêng rẽ.

Vì sự tăng trưởng hàm mũ của số lượng các khả năng, điều này tạo ra sự khác biệt lớn. Trình hoạch định cố tránh bị kẹt trong các vấn đề tìm kiếm liên kết khổng lồ bằng việc không làm sập một truy vấn con nếu nhiều hơn các khái niệm from_collapse_limit FROM có thể dẫn tới truy vấn cha đó. Bạn có thể cân đối giữa thời gian lên kế hoạch và chất lượng kế hoạch đó bằng việc tinh chỉnh tham số thời gian chạy này lên hoặc xuống.

from_collapse_limit và join_collapse_limit được đặt tên tương tự vì chúng làm hầu hết điều y hệt: một cái kiểm soát khi trình hoạch định sẽ "dàn bẹt" các truy vấn con, còn cái kia kiểm soát khi nó dàn bẹt các liên kết rõ ràng. Điển hình là bạn có thể hoặc thiết lập join_collapse_limit ngang bằng với from_collapse_limit (sao cho các liên kết rõ ràng và các truy vấn con hành động tương tự) hoặc thiết lập join_collapse_limit về 1 (nếu bạn muốn kiểm soát thứ tự liên kết bằng các liên kết rõ ràng). Nhưng bạn có thể thiết lập chúng khác nhau nếu bạn đang cố gắng tinh chỉnh sự cân nhắc giữa thời gian lên kế hoạch và thời gian chạy.

14.4. Đưa dữ liệu vào cơ sở dữ liệu

Bạn có thể cần chèn một lượng lớn dữ liệu khi lần đầu đưa dữ liệu vào cơ sở dữ liệu. Phần này bao

gồm một số gợi ý về cách để tiến hành qui trình này có hiệu quả nhất có thể.

14.4.1. Vô hiệu hóa thực hiện tự động (Autocommit)

Khi sử dụng nhiều lệnh chèn INSERT, hãy tắt tự động thực hiện (autocommit) và chỉ tiến hành một thực hiện (commit) lúc kết thúc. (Trong SQL thông thường, điều này có nghĩa là đưa ra BEGIN ở đầu và COMMIT ở cuối. Một số thư viện máy trạm có thể làm điều này sau lưng bạn, trong trường hợp đó bạn cần chắc chắn thư viện đó thực hiện điều đó khi bạn muốn nó được thực hiện). Nếu bạn cho phép từng sự chèn được thực hiện riêng rẽ, thì PostgreSQL đang làm nhiều công việc cho từng hàng mà được bổ sung thêm vào. Một lợi ích bổ sung của việc tiến hành tất cả sự chèn trong một giao dịch là nếu sự chèn một hàng từng hỏng thì sau đó sự chèn của tất cả các hàng được chèn vào tới điểm đó có thể phải quay lại, sao cho bạn sẽ không bị kẹt với các dữ liệu được tải lên một phần.

14.4.2. Sử dụng COPY

Hãy sử dụng COPY để tải tất cả các hàng trong một lệnh, thay vì sử dụng một loạt các lệnh INSERT. Lệnh COPY được tối ưu hóa cho việc tải số lượng lớn các hàng; là ít mềm dẻo hơn so với INSERT, nhưng chịu ít hơn đáng kể tổng chi phí cho các tải dữ liệu lớn. Vì COPY là một lệnh duy nhất, nên không cần vô hiệu hóa autocommit nếu bạn sử dụng phương pháp này để đưa dữ liệu vào một bảng.

Nếu bạn không thể sử dụng COPY, có thể giúp sử dụng PREPARE để tạo một lệnh INSERT được chuẩn bị trước, và sau đó sử dụng EXECUTE bao nhiêu lần tùy theo yêu cầu. Điều này tránh được một số tổng chi phí của việc phân tích lặp đi lặp lại và lên kế hoạch cho lệnh INSERT. Các giao diện khác nhau đưa ra cơ sở này theo các cách thức khác nhau; hãy tìm kiếm “prepared statements” (“các lệnh được chuẩn bị”) trong tài liệu giao diện.

Lưu ý rằng việc tải một số lượng lớn các hàng bằng việc sử dụng COPY hầu như luôn nhanh hơn so với việc sử dụng INSERT, thậm chí nếu PREPARE được sử dụng và nhiều sự chèn được tạo thành bó trong một giao dịch duy nhất.

COPY là nhanh nhất khi được sử dụng trong giao dịch y hệt như một lệnh CREATE TABLE hoặc TRUNCATE trước đó. Trong các trường hợp như vậy không WAL nào cần phải được viết, vì trong trường hợp có một lỗi, thì các tệp chứa các dữ liệu mới được tải sẽ bị loại bỏ bằng mọi cách. Tuy nhiên, sự xem xét chỉ áp dụng khi wal_level là minimal như tất cả các lệnh nếu không phải viết WAL.

14.4.3. Loại bỏ chỉ số

Nếu bạn đang tải một bảng được tạo mới, thì phương pháp nhanh nhất là tạo bảng đó, tải theo bó các dữ liệu bảng bằng việc sử dụng COPY, rồi tạo các chỉ số bất kỳ cần thiết cho bảng đó. Việc tạo một chỉ số trong các dữ liệu tồn tại trước đó là nhanh hơn so với việc cập nhật nó từng chút một khi từng hàng được tải.

Nếu bạn đang thêm các lượng lớn các dữ liệu vào một bảng đang tồn tại, có thể là một thành công để loại bỏ các chỉ số, tải bảng đó, và sau đó tạo lại các chỉ số. Tất nhiên, hiệu năng của cơ sở dữ liệu

đối với những người sử dụng khác có thể phải chịu trong thời gian các chỉ số không có. Bạn cũng nên nghĩ 2 lần trước khi bỏ một chỉ số duy nhất, vì việc kiểm tra lỗi kham được bằng sự ràng buộc duy nhất sẽ bị mất trong khi chỉ số không còn.

14.4.4. Loại bỏ ràng buộc khóa ngoại

Hệt như với các chỉ số, một ràng buộc khóa ngoại có thể được kiểm tra "theo bó" hiệu quả hơn so với theo từng hàng một. Vì thế có thể là hữu dụng để bỏ các ràng buộc khóa ngoại, tải dữ liệu, và tái tạo lại các ràng buộc. Một lần nữa, có một sự bù trừ giữa tốc độ tải dữ liệu và mất kiểm tra lỗi khi không có ràng buộc.

Hơn nữa, khi bạn tải dữ liệu vào một bảng với các ràng buộc khóa ngoại đang tồn tại, thì từng hàng mới đòi hỏi một khoản đầu vào trong danh sách máy chủ của các sự kiện treo trigger (vì nó là sự phát hỏa của một trigger mà kiểm tra ràng buộc khóa ngoại của hàng). Việc tải nhiều triệu hàng có thể làm cho hàng đợi các sự kiện trigger gây quá tải cho bộ nhớ có sẵn, dẫn tới việc hoán đổi không chịu nổi hoặc thậm chí thất bại hoàn toàn của lệnh. Vì thế có thể là cần thiết, không chỉ mong muốn, bỏ và áp dụng lại các khóa ngoại khi tải các lượng lớn dữ liệu. Nếu việc loại bỏ tạm thời ràng buộc là không chấp nhận được, chỉ quá trình khác có thể chia hoạt động tải thành các giao dịch nhỏ hơn.

14.4.5. Gia tăng maintenance_work_mem

Tăng tạm thời biến cấu hình maintenance_work_mem khi tải lượng lớn các dữ liệu có thể dẫn tới hiệu năng được cải thiện. Điều này sẽ giúp tăng tốc độ các lệnh CREATE INDEX và các lệnh ALTER TABLE ADD FOREIGN KEY. Nó sẽ không làm nhiều cho bản thân lệnh COPY, nên tư vấn này chỉ hữu dụng khi bạn đang sử dụng 1 hoặc 2 kỹ thuật ở trên.

14.4.6. Tăng checkpoint_segments

Tăng tạm thời biến cấu hình checkpoint_segments cũng có thể làm cho dữ liệu lớn tải nhanh hơn. Điều này là vì việc tải một lượng dữ liệu lớn vào PostgreSQL sẽ làm cho các điểm kiểm tra xảy ra thường xuyên hơn so với tần suất kiểm tra điểm thông thường (được biến checkpoint_timeout chỉ định). Bất kỳ khi nào một điểm kiểm tra xảy ra, tất cả các trang bẩn phải được phun ra đĩa. Bằng việc tăng tạm thời checkpoint_segments trong quá trình bó dữ liệu tải lên, số lượng các điểm kiểm tra được yêu cầu có thể sẽ bị giảm đi.

14.4.7. Nhân bản dòng và lưu trữ WAL vô hiệu hóa được

Khi tải lượng lớn các dữ liệu vào một cài đặt mà sử dụng nhân bản dòng hoặc lưu trữ WAL, có thể là nhanh hơn để thực hiện một sao lưu cơ bản mới sau khi tải đó đã hoàn tất so với để xử lý một lượng lớn các dữ liệu WAL từng chút một. Để ngăn chặn việc lưu ký WAL từng chút một trong khi tải, vô hiệu hóa nhân bản dòng và lưu trữ, bằng việc thiết lập wal_level về minimal, archive_mode về off, và max_wal_senders về 0. Nhưng hãy lưu ý rằng việc thay đổi các thiết lập đó đòi hỏi một sự khởi động lại máy chủ.

Ngoài việc tránh thời gian cho lưu trữ hoặc gửi WAL để xử lý các dữ liệu WAL, làm thế này cũng sẽ thực sự tiến hành các lệnh nhất định nhanh hơn, vì chúng được thiết kế để không viết WAL hoàn toàn nếu wal_level là minimal. (Chúng có thể đảm bảo sự mất an toàn rẻ hơn bằng việc thực hiện một fsync ở cuối hơn là bằng việc viết WAL). Điều này áp dụng cho các lệnh sau:

- CREATE TABLE AS SELECT
- CREATE INDEX (and variants such as ALTER TABLE ADD PRIMARY KEY)
- ALTER TABLE SET TABLESPACE
- CLUSTER
- COPY FROM, khi bảng đích từng được tạo ra hoặc cắt bớt trước đó trong cùng giao dịch y hệt.

14.4.8. Chạy ANALYZE sau đó

Bất kỳ khi nào bạn tùy biến đáng kể sự phân phối dữ liệu trong một bảng, thì việc chạy ANALYZE được khuyến cáo mạnh mẽ. Điều này bao gồm việc tải theo đống lượng dữ liệu lớn vào bảng đó. Việc chạy ANALYZE (hoặc VACUUM ANALYZE) đảm bảo rằng trình hoạch định có các số liệu cập nhật về bảng đó. Nếu không có các số liệu thống kê hoặc chúng lỗi thời, thì trình hoạch định có thể đưa ra quyết định nghèo nàn trong việc lên kế hoạch truy vấn, dẫn tới hiệu năng nghèo nàn trong bất kỳ bảng nào với các số liệu thống kê không chính xác hoặc không tồn tại. Lưu ý rằng nếu autovacuum daemon được kích hoạt, thì nó có thể chạy ANALYZE một cách tự động; xem Phần 23.1.3 và Phần 23.1.5 để có thêm thông tin.

14.4.9. Vài lưu ý về pg_dump

Các scripts chữa đổ bể được pg_dump sinh ra tự động áp dụng một vài, nếu không nói là tất cả, các chỉ dẫn ở trên. Để tải lại một sự đổ vỡ pg_dump nhanh nhất có thể, bạn cần làm thêm vài điều bằng tay. (Lưu ý là các điểm đó áp dụng khi phục hồi một đổ vỡ, không phải khi tạo ra nó. Các điểm y hệt áp dụng hoặc việc tải một đổ vỡ văn bản với psql hoặc việc sử dụng pg_restore để tải từ một tệp lưu trữ pg_dump).

Mặc định, pg_dump sử dụng COPY, và khi nó đang sinh ra một sự đổ vỡ sơ đồ và dữ liệu, hãy cẩn thận để tải dữ liệu trước khi tạo ra các chỉ số và các khóa ngoại. Vì thế trong trường hợp này vài chỉ dẫn được điều khiển tự động. Những gì còn lại để bạn phải làm là:

- Thiết lập các giá trị phù hợp (như, lớn hơn là bình thường) cho maintenance_work_mem và checkpoint_segments.
- Nếu việc sử dụng nhân bản dòng hoặc lưu trữ WAL, hãy cân nhắc việc vô hiệu hóa chúng trong quá trình phục hồi. Để làm điều đó, hãy thiết lập archive_mode về off, wal_level về minimal, và max_wal_senders về 0 trước khi tải sự đổ vỡ đó. Sau đó, hãy thiết lập chúng ngược về các giá trị đúng và tiến hành sao lưu cơ bản mới lại.
- Cân nhắc liệu toàn bộ sự đổ vỡ có nên được phục hồi lại như một giao dịch duy nhất hay không. Để làm điều đó, hãy truyền lựa chọn dòng lệnh -1 hoặc --single-transaction tới psql hoặc pg_restore. Khi sử dụng chế độ này, thậm chí các lỗi nhỏ nhất sẽ quay lại phục hồi

toàn bộ, có khả năng hủy bỏ nhiều giờ xử lý. Phụ thuộc vào cách dữ liệu có liên quan tới nhau như thế nào, điều đó có thể coi là được ưu tiên để làm sạch bằng tay, hay là không. Các lệnh COPY sẽ chạy nhanh nhất nếu bạn sử dụng một giao dịch duy nhất và có lưu trữ WAL được tắt.

- Nếu nhiều CPU là sẵn sàng trong máy chủ cơ sở dữ liệu, hãy cân nhắc sử dụng lựa chọn --jobs của pg_restore. Điều này cho phép tải các dữ liệu hiện hành và tạo chỉ số.
- Chạy ANALYZE sau đó.

Một sự đổ vỡ chỉ dữ liệu sẽ vẫn sử dụng COPY, nhưng nó không bỏ hoặc tái tạo lại các chỉ số, và nó thường không động chạm tới các khóa ngoại[2]. Vì thế khi tải một đổ vỡ chỉ dữ liệu, tùy bạn bỏ và tái tạo lại các chỉ số và các khóa ngoại nếu bạn muốn sử dụng các kỹ thuật đó. Vẫn là hữu dụng để tăng checkpoint_segments trong khi tải các dữ liệu đó, nhưng đừng có bận tâm tới việc gia tăng maintenance_work_mem; thay vào đó, bạn nên làm thế trong khi tái tạo bằng tay các chỉ số và các khóa ngoại sau đó. Và đừng quên ANALYZE khi bạn thực hiện xong; xem Phần 23.1.3 và Phần 23.1.5 để có thêm thông tin.

14.5. Thiết lập không bền vững

Tính bền vững là một chức năng của cơ sở dữ liệu mà đảm bảo việc ghi các giao dịch được thực hiện xong thậm chí nếu máy chủ hỏng hoặc mất điện. Tuy nhiên, tính bền vững bổ sung thêm tổng chi phí đáng kể cho cơ sở dữ liệu, nên nếu site của bạn không đòi hỏi một sự đảm bảo như vậy, thì PostgreSQL có thể được thiết lập cấu hình để chạy nhanh hơn nhiều. Sau đây là những thay đổi cấu hình mà bạn có thể làm để cải thiện hiệu năng trong các trường hợp như vậy; chúng không vô hiệu hóa thực hiện các đảm bảo có liên quan tới các hỏng hóc cơ sở dữ liệu, chỉ đột ngột dừng hệ điều hành, ngoại trừ như được nhắc ở dưới:

- Đặt thư mục dữ liệu của bó cơ sở dữ liệu trong một hệ thống tệp được bộ nhớ hỗ trợ (như đĩa RAM). Điều này loại bỏ tất cả I/O đĩa cơ sở dữ liệu, nhưng hạn chế lưu trữ dữ liệu về lượng bộ nhớ sẵn sàng (và có thể là hoán đổi swap).
- Tắt fsync; không có nhu cầu để phóng dữ liệu ra đĩa.
- Tắt full_page_writes; không có nhu cầu để canh phòng chống ghi trang một phần.
- Gia tăng checkpoint_segments và checkpoint_timeout; điều này làm giảm tần suất của các điểm kiểm tra, nhưng làm gia tăng các yêu cầu lưu trữ của /pg_xlog.
- Tắt synchronous_commit; có thể không có nhu cầu ghi WAL lên đĩa trong mỗi lần thực hiện xong. Điều này ảnh hưởng tới độ bền giao dịch khi hỏng cơ sở dữ liệu.

2 Bạn có thể có hiệu ứng vô hiệu hóa các khóa ngoại bằng việc sử dụng lựa chọn --disable-triggers - nhưng nhận thấy rằng nó loại trừ, thay vì chỉ trì hoãn, kiểm tra hợp lệ khóa ngoại, và vì thế có khả năng chèn các dữ liệu xấu nếu bạn sử dụng nó.

Tham khảo thư loại

Các tham chiếu được lựa chọn để đọc về SQL và PostgreSQL

Một số sách trắng và báo cáo kỹ thuật từ đội phát triển gốc ban đầu POSTGRES là sẵn sàng tại webiste của Phòng Khoa học Máy tính, Đại học California, Berkeley[11].

Các sách SQL tham khảo

Judith Bowman, Sandra Emerson, and Marcy Darnovsky, The Practical SQL Handbook: Using SQL Variants , Fourth Edition, Addison-Wesley Professional, ISBN 0-201-70309-2, 2001.

C. J. Date and Hugh Darwen, A Guide to the SQL Standard: A user's guide to the standard database language SQL , Fourth Edition, Addison-Wesley, ISBN 0-201-96426-0, 1997.

C. J. Date, An Introduction to Database Systems , Eighth Edition, Addison-Wesley, ISBN 0-321-19784-4, 2003.

Ramez Elmasri and Shamkant Navathe, Fundamentals of Database Systems , Fourth Edition, Addison-Wesley, ISBN 0-321-12226-7, 2003.

Jim Melton and Alan R. Simon, Understanding the New SQL: A complete guide , Morgan Kaufmann, ISBN 1-55860-245-3, 1993.

Jeffrey D. Ullman, Principles of Database and Knowledge: Base Systems , Volume 1, Computer Science Press, 1988.

Tài liệu PostgreSQL đặc thù

Stefan Simkovics, Enhancement of the ANSI SQL Implementation of PostgreSQL , Department of Information Systems, Vienna University of Technology, November 29, 1998.

Discusses SQL history and syntax, and describes the addition of INTERSECT and EXCEPT constructs into PostgreSQL. Prepared as a Master's Thesis with the support of O. Univ. Prof. Dr. Georg Gottlob and Univ. Ass. Mag. Katrin Seyr at Vienna University of Technology.

A. Yu and J. Chen, The POSTGRES Group, The Postgres95 User Manual , University of California, Sept. 5, 1995.

Zelaine Fong, The design and implementation of the POSTGRES query optimizer[12], University of California, Berkeley, Computer Science Department.

11 http://db.cs.berkeley.edu/papers/
12 http://db.cs.berkeley.edu/papers/UCB-MS-zfong.pdf

Các kỷ yếu và bài báo

Nels Olson, Partial indexing in POSTGRES: research project , University of California, UCB Engin T7.49.1993 O676, 1993.

L. Ong and J. Goh, “A Unified Framework for Version Modeling Using Production Rules in a Database System”, ERL Technical Memorandum M90/33 , University of California, April, 1990.

L. Rowe and M. Stonebraker, “ The POSTGRES data model 3 ”, Proc. VLDB Conference, Sept. 1987.

P. Seshadri and A. Swami, “Generalized Partial Indexes (cached version) 4 ”, Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, 420-7.

M. Stonebraker and L. Rowe, “ The design of POSTGRES 5 ”, Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

M. Stonebraker, E. Hanson, and C. H. Hong, “The design of the POSTGRES rules system”, Proc. IEEE Conference on Data Engineering, Feb. 1987.

M. Stonebraker, “ The design of the POSTGRES storage system 6 ”, Proc. VLDB Conference, Sept. 1987.

M. Stonebraker, M. Hearst, and S. Potamianos, “ A commentary on the POSTGRES rules system 7 ”, SIGMOD Record 18(3) , Sept. 1989.

M. Stonebraker, “ The case for partial indexes 8 ”, SIGMOD Record 18(4) , Dec. 1989, 4-11.

M. Stonebraker, L. A. Rowe, and M. Hirohama, “ The implementation of POSTGRES 9 ”, Transactions on Knowledge and Data Engineering 2(1) , IEEE, March 1990.

M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, “ On Rules, Procedures, Caching and Views in Database Systems 10 ”, Proc. ACM-SIGMOD Conference on Management of Data, June 1990.