PostgreSQL 8.0.1 Eğitmeni

Çeviren:
Nilgün Belma Bugüner
<nilgun (at) belgeler·gen·tr>

Çeviren:
Volkan "knt" Yazıcı
<v0lkany (at) yahoo.com>

Mart 2005

Özet

Bu belge size genel hatlarıyla `PostgreSQL` üzerinde denetim kurabileceğiniz kadar deneyim kazandırmayı amaçlamaktadır. Konuları ve kapsamı itibariyle kesinlikle tam bir rehber görevi görmeyecektir.

Konu Başlıkları

Geçmiş

8.0.1	Mart 2005	NBB
PostgreSQL 8.0.1 ile gelen belgelerden güncelleme yapıldı.		
7.3.2	Temmuz 2003	knt
PostgreSQL 7.3.2 ile gelen belgelerden çevrildi.		

Türkçe çeviri için yasal uyarı

Özgün belge için yasal uyarı

1. Giriş

PostgreSQL Eğitmenine hoşgeldiniz. Sonraki kısımlarda PostgreSQL'e, ilişkisel veritabanı kavramlarına ve SQL dilinin kavramlarına yabancı olanlar için SQL diline basit bir giriş yapılacaktır. Sadece nasıl bilgisayar kullanılacağı üzerine bilginizin olduğu varsayılacaktır. Belli bir Unix veya yazılım geliştirme deneyimi gerekmiyor. Bu belge size genel hatlarıyla PostgreSQL üzerinde denetim kurabileceğiniz kadar deneyim kazandırmayı amaçlamaktadır. Konuları ve kapsamı itibariyle kesinlikle tam bir rehber görevi görmeyecektir.

Bu belgeyi bitirdikten sonra SQL dili üzerine daha ayrıntılı bilgi için PostgreSQL 8.0 belgelerindeki SQL Dili[B1] oylumuna ya da PostgreSQL üzerine geliştireceğiniz uygulamalarda daha geniş bilgi için PostgreSQL 8.0 belgelerindeki İstemci Arayüzleri[B2] oylumuna bakabilirsiniz. Kendi sunucularını kurup üzerinde yönetimi gerçekleştirecek kişiler ayrıca PostgreSQL 8.0 belgelerindeki Sistem Yönetimi[B3] oylumuna başvurabilirler.

2. Başlangıç

Kurulum

PostgreSQL kullanmadan önce kurulumu gerçekleştirmeniz gerekmekte. PostgreSQL, işletim sisteminizin önkurulumu esnasında yüklenmiş olabileceği gibi, sistem yöneticisi tarafından da önceden kurulmuş olabilir. Böyle bir durumda dağıtımınızın belgelerinden ya da sistem yöneticinizden PostgreSQL'e nasıl ulaşabileceğinize dair bilgileri edinebilirsiniz.

PostgreSQL'in kurulu olduğu ya da onu kullanabileceğiniz konusunda emin değilseniz, PostgreSQL'i kendiniz kurabilirsiniz. Kurulum işlemi zor olmamakla birlikte sizin için de iyi bir alıştırma niteliği taşır. PostgreSQL her kullanıcı tarafından kurulabilir; ayrıcalıklı kullanıcı (root) hakları gerekmemektedir.

Eğer PostgreSQL'i kendiniz kuracaksanız bu konuda PostgreSQL 8.0 kitabının Sunucu Yönetimi[B4] oylumunda yer alan kurulum bölümüne bakıp, kurulum tamamlandığında buraya dönebilirsiniz. Ortam değişkenlerinin belirlenmesinin anlatıldığı bölüme bakmayı unutmayın.

Eğer sistem yöneticiniz kurulum işlemlerini öntanımlı yoldan yapmadıysa biraz daha yapacak işiniz var demektir. Örneğin eğer sunucu makineniz uzakta bir sistem ise, PGHOST ortam değişkenine veritabanı sunucusunun ismini atamalısınız. Ayrıca, PGPORT ortam değişkenine de atama yapmanız gerekebilir. Son olarak, eğer çalıştırmayı denediğiniz bir uygulama veritabanına bağlanamadığını belirten bir hata iletisi veriyorsa; sistem yöneticiniz ile bağlantı kurmayı deneyin, eğer yönetici siz iseniz, belgeleri tekrar gözden geçirip çalışma ortamını doğru ayarlayıp ayarlamadığınıza bir kez daha bakın. Eğer bir önceki paragrafı anlamadıysanız, bir sonraki bölümü okuyun.

2.1. Mimarinin Temelleri

Daha ileri bölümleri okumadan önce, PostgreSQL'in temel sistem mimarisini anlamanızda yarar var. PostgreSQL bölümlerinin birbirleri ile nasıl ilişki içinde olduğunu anlamanız yönünde bu kısım bir derece de olsa yararlı olacaktır.

PostgreSQL, veritabanı dilinde sunucu/istemci temeline dayanan bir sistem kullanmaktadır. Bir PostgreSQL oturumu birbirleriyle ilişkili çalışan şu süreçlerden oluşur:

- Veritabanı dosyalarını yöneten bir sunucu süreci istemci uygulamalarından gelen bağlantıları kabul eder ve istenilen işlemleri onlar adına gerçekleştirir. Veritabanı sunucu uygulamasının ismi **postmaster**'dır.
- Kullanıcının istemci uygulaması veritabanında sorgulamanın gerçekleşmesini isteyen uygulamadır. İstemci uygulamaları çok çeşitlidir: salt metin (text) tabanlı bir istemci aracı, grafiksel arayüzlü bir uygulama, veritabanına bağlanıp ilgili HTML sayfaları olarak göstercek bir web sunucusu veya özelleşmiş bir

veritabanı onarım aracı. Bazı istemci uygulamaları PostgreSQL dağıtımları ile sağlanmakta olup çoğu da kullanıcılar tarafından geliştirilmektedir.

Genellikle, sunucu/istemci uygulamalarında, sunucu ve istemci ayrı makinelerde olurlar. Böyle bir durumda birbileri ile TCP/IP ağ bağlantısı üzerinden haberleşirler. Bir istemci makinenin erişebildiği dosyaların veritabanı sunucusu olan makine üzerinde erişilebilir olamayabileceğini (ya da farklı bir dosya ismi ile erişilebilineceğini) aklınızdan çıkarmayın.

PostgreSQL sunucusu çoklu istemci bağlantılarına izin verebilmektedir. Bu amaçla her yeni bağlantı için yeni bir süreç başlatır ("fork"). Bu noktada, istemci ve yeni sunucu süreci özgün **postmaster** süreciyle etkileşime girmeden haberleşebilirler. Bu arada, **postmaster** istemci bağlantılarını bekler, istemcileri ilgili sunucu süreci ile ilişkilendirmeye çalışır. (Tabii ki, bunların hepsi kullanıcıdan habersiz olarak artalanda gerçekleşir. İşlemin nasıl gerçekleştiğini bilin istedik.)

2.2. Bir Veritabanının Oluşturulması

Veritabanına sunucusuna erişiminiz olup olmadığını görmek için yapılacak ilk sınama bir veritabanı oluşturmaya çalışmaktır. Çalışan bir PostgreSQL sunucusu çok sayıda veritabanını yönetebilir. Genellikle, her proje ya da her kullanıcı için ayrı bir veritabanı kullanılır.

Muhtemelen, sistem yöneticiniz sizin için bir veritabanını zaten oluşturmuştur ve size oluşturduğu veritabanı ismini de söylemiştir. Böyle bir durumda bu adımı geçerek bir sonraki bölüme bakabilirsiniz.

Bu örnekte, yeni bir veritabanını mydb ismiyle şöyle oluşturabilirsiniz:

```
$ createdb mydb
```

Çıktısının şöyle olması lazım:

```
CREATE DATABASE
```

Bu çıktıyı alıyorsanız, bu adım tamamlanmış demektir.

Ama, **createdb** komutunun bulunamadığına ilişkin, şöyle bir çıktı alıyorsanız,

```
createdb: command not found
```

PostgreSQL olması gerektiği gibi kurulmamış demektir. Ya hiç kurulum yapılmamıştır ya da dosya arama yolları ($PATH) doğru yapılandırılmamıştır. Komutu dosya yolunu belirterek kullanmayı deneyin:

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

Bu dosya yolu sisteminizde farklı olabilir. Böyle bir durumda sistem yöneticisi ile bağlantı kurmayı deneyin ya da kurulum adımlarını tekrar gözden geçirip sorunu tespit etmeye çalışın.

Çıktı şöyle de olabilirdi:

```
createdb: could not connect to database template1: could not connect to server:
No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

Böyle bir hatanın anlamı ya sunucu başlatılmamıştır ya da **createdb** sunucunun aradığı yerde değildir. Kurulum adımlarını yeniden gözden geçirin ya da sistem yöneticisi ile temasa geçin.
Yanıt şöyle de olabilirdi:

```
createdb: could not connect to database template1: FATAL:  user "nilgun" does not
```

```
exist
```

Burada "nilgun" yerine sizin kullanıcı isminiz görünecektir. Bu, sistem yöneticinizin sizin için bir PostgreSQL kullanıcısı hesabı açmadığı anlamına gelir. (PostgreSQL kullanıcı hesapları, sistem kullanıcı hesaplarından ayrıdır.) Eğer sistem yöneticisi sizseniz, hesapların oluşturulması ile ilgili bilgi edinmek için PostgreSQL 8.0 belgelerindeki Veritabanı Kullanıcıları ve Grupları[B5] kısmına bakın. İlk kullanıcı hesabını oluşturmak için PostgreSQL'i kuran işletim sistemi kullanıcısı (genellikle bu kullanıcı postgres'dir) olmanız gerekecektir. İsterseniz, bu amaçla, bir sistem kullanıcı isminden farklı bir PostgreSQL kullanıcı ismini de kullanabilirsiniz; PostgreSQL kullanıcı isminini belirtmek için ya -U seçeneğini kullanmalı ya da bu ismi PGUSER ortam değişkenine atamalısınız.

Eğer bir kullanıcı hesabınız varsa, ama bir veritabanı oluşturma izniniz yoksa, şöyle bir çıktı alacaksınız:

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

Her kullanıcının yeni bir veritabanı oluşturma yetkisi yoktur. Eğer PostgreSQL sizin veritabanı oluşturma isteğinizi geri çeviriyorsa, sistem yöneticisinin size gerekli izinleri vermesi gerekmektedir. Böyle bir durumda sistem yöneticisi ile temasa geçin. Eğer sistem yöneticisi siz iseniz böyle bir izin işlemi için sunucuyu hangi kullanıcı ile başlatmışsanız onun ile sisteme giriş yapın ve bu eğiticiyi okuyup uygulamak isteyenlere gerekli hakları tanıyın.[1]

İsterseniz başka isimler ile de veritabanları oluşturabilirsiniz. PostgreSQL istediğiniz sayıda veritabanı oluşturma imkanını sunmaktadır. Veritabanı isminin ilk karakteri bir harf olmalı ve isim 63 karakterden daha uzun olmamalıdır. Tercihen kullanıcı adınız ile aynı ismi taşıyan veritabanları oluşturulması tavsiye olunur. Çoğu araç, böyle bir veritabanı ismini öntanımlı olarak kabul eder ve bu sizi az da olsa yazmaktan kurtarır. Yeni bir veritabanı oluşturmak için, basitçe şunu yazın:

```
$ createdb
```

Eğer veritabanınızı artık kullanmak istemiyorsanız onu kaldırabilirsiniz. Örnek olarak, eğer siz mydb adlı veritabanının sahibi iseniz, bunu şöyle silebilirsiniz:

```
$ dropdb mydb
```

(Bu komut için, veritabanı ismi öntanımlı olarak kullanıcı ismi değildir. Daima bu ismi belirtmeniz gerekir.) Bu işlem sonucunda fiziksel olarak veritabanınız ile ilgili bütün dosyalar silinecektir ve veritabanınızı geri alamayacaksınız. Bu yüzden bu komutu uygulamadan önce kararınızı tekrar gözden geçirmeniz tavsiye olunur.

createdb ve **dropdb** hakkında daha fazla bilgi edinmek için PostgreSQL 8.0 belgelerindeki createdb[B6] ve dropdb[B7] komutlarının açıklamalarına bakınız.

2.3. Bir Veritabanına Erişim

Veritabanını oluşturduktan sonra, ona şöyle erişebilirsiniz:

- **psql** adlı etkileşimli PostgreSQL uçbirim uygulaması ile veritabanına giriş yapıp, istediğiniz işlemleri gerçekleştirir ve SQL komutlarınızı çalıştırabilirsiniz.
- **PgAccess** gibi çizgesel arayüzlü bir uygulama ya da ODBC destekli ofis yazılımlarını kullanarak veritabanı oluşturabilir ve üzerinde işlem yapabilirsiniz. Bu tür uygulamalar ve kullanımları bu eğitmenin kapsamı dışındadır.
- Uygun yazılım geliştirme dilleri ile kendi uygulamalarınızı da yazabilirsiniz. Bu konu hakkında ayrıntılı bilgiyi PostgreSQL 8.0 belgelerindeki İstemci Arayüzleri[B8] oylumunda bulabilirsiniz.

Bu belgedeki alıştırmaları denemek için muhtemelen **psql**'i kullanmak isteyeceksiniz. **psql**'i mydb adlı veritabanına erişmek için şu şekilde başlatabilirsiniz:

```
$ psql mydb
```

Eğer bir veritabanı ismi belirtmezseniz öntanımlı olarak kullanıcı adınız ile aynı ismi taşıyan veritabanına erişilmeye çalışılacaktır. Bu konudan önceki bölümde bahsetmiştik.

psql sizi aşağıdaki gibi bir çıktı ile karşılayacaktır:

```
Welcome to psql 8.0.1, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

mydb=>
```

Son satır şu şekilde de olabilir:

```
mydb=#
```

Bunun anlamı, çoğunlukla olduğu gibi eğer PostgreSQL'i kendiniz kurduysanız, veritabanının en yetkili kullanıcısı olduğunuz anlamına gelir. En yetkili kullanıcı olmak, hiçbir izin işlemine tabi tutulmayacağınız anlamına gelir. Fakat bu konu bu eğitmenin kapsamında değildir.

Eğer **psql**'i çalıştırmakta sorun yaşarsanız önceki bölüme dönün. **psql** ile **createdb** arasında çalıştırma sorunlarına tanı konulması bakımından fark yoktur. Eğer **createdb** çalışıyorsa, **psql**'in de çalışması gerekir.

Son satırda yer alan mydb=> komut satırı **psql**'in kendi çalışma alanı içine SQL sorguları yazmanızı beklediği anlamına gelir. Şu komutları deneyiniz:

```
mydb=> SELECT version();
                           version
------------------------------------------------------------------
 PostgreSQL 8.0.1 on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)

mydb=> SELECT current_date;
    date
------------
 2002-08-31
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

Bunun dışında **psql** kendine ait, standart SQL komutu olmayan, bir kaç dahili komuta daha sahiptir. Bu tür komutlar bir tersbölü karakteri (****) ile başlar. Bu komutların birkaçı karşılama ekranında listelenmişti. Örnek olarak, çeşitli PostgreSQL SQL komutları hakkında yardım almak için şunu yazabilirsiniz:

```
mydb=> \h
```

psql'den çıkmak için şunu kullanabilirsiniz:

```
mydb=> \q
```

Böylece **psql**'den çıkıp sistemin komut satırına geri döneceksiniz. (Daha fazla dahili komut için psql satırında **\?** komutunu kullanabilirsiniz.) **psql**'in tam olarak ne yapabildiği hakkında ayrıntılı bilgiyi PostgreSQL 8.0 belgelerindeki psql komut açıklamasında[B9] bulabilirsiniz. Eğer PostgreSQL kurulumunu tam olarak gerçekleştirdiyseniz sistemin komut satırında **man psql** yazarak da bu bilgilere ulaşabilirsiniz. Bu eğitmende bu özelliklerin hepsine değinmeyeceğiz ama kendiniz hepsini deneyebilirsiniz.

3. SQL Dili

Bu kısımda basit işlemleri uygulamak için SQL kullanımına kısaca değineceğiz. Verilecek SQL bilgisi bir başlangıç niteliği taşımaktadır, kesinlikle tam bir SQL eğitmeni değildir. SQL dili üzerine sayısız kitap yazılmıştır, bunlar içinde Yeni SQL Dilinin Anlaşılması[B10] ve SQL Standartları Kılavuzu[B11]'nu örnek gösterebiliriz. Bazı PostgreSQL dil özelliklerinin standarttan fazlasını içerdiğini bilmenizde yarar var.

Bu kısımdaki örneklerde, önceki bölümde **psql**'i başlatırken kullandığınız, mydb isimli veritabanının oluşturmuş olduğunuz varsayılmıştır.

Bu belgedeki örnekleri ayrıca PostgreSQL kaynak paketinde yer alan src/tutorial/ dizininde de bulabilirsiniz. Bu dosyaları kullanmaya başlamadan önce dizine girip **make** komutunu vermelisiniz:

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

Böylece, kullanıcı tanımlı işlevlerle türleri içeren C dosyaları derlenmiş ve betikler oluşturulmuş olur. (Bu işlem için GNU **make** kullanılmalıdır. Sisteminizdeki ismi farklı, belki de **gmake** olabilir.) Eğitmeni şöyle başlatabilirsiniz:

```
$ cd ..../src/tutorial
$ psql -s mydb
  ...

mydb=> \i basics.sql
```

Buradaki **\i** komutu, komutları belirtilen dosyadan okur. -s seçeneği sizi, her komutu sunucuya göndermeden önce bekleyen tek adımlık kipe sokar. Bu bölümde kullanılmış olan komutları basics.sql dosyasında bulabilirsiniz.

3.1. Kavramlar

PostgreSQL bir ilişkisel veritabanı yönetim sistemidir (RDBMS – Relational Database Management System). Bunun anlamı, PostgreSQL'in ilişkilerin içerdiği verileri yöneten bir sistem olduğudur. Burada bahsedilen ***ilişki*** (relation) aslında ***tablo*** karşılığı bir matematik terimidir. Verinin tablolarda saklanması olayı günümüzde gayet olağan gibi görünürse de veritabanlarının organize edilmesi için daha pek çok yol vardır. Unix türevi işletim sistemlerindeki dosya/dizin yapısı hiyerarşik veritabanlarına güzel bir örnektir. Günümüzde daha gelişmiş bir veritabanı türü de nesne yönelimli veritabanlarıdır.

Her tablo satırlardan oluşur. Ve her ***satır*** kendi içinde, belli veri türlerine özel ***sütun***lara ayrılmıştır. Sütunlar her satırda aynı sayıda ve sırada olmasına karşın, SQL aynı şeyi satır sıralaması için garanti etmez (yine de çıktı alırken isteğe bağlı olarak sıralandırılabilirler).

Tablolar veritabanları halinde gruplanır ve tek bir PostgreSQL sunucu süreci tarafından yönetilen veritabanları bir veritabanı ***küme***si oluştururlar.

3.2. Yeni bir Tablonun Oluşturulması

Yeni bir tabloyu, tablo ismini ve içerdiği sütün isimlerini veri türleri ile birlikte belirterek oluşturabilirsiniz:

```
CREATE TABLE weather (
    city            varchar(80),
    tmp_lo          int,           -- en düşük sıcaklık
    tmp_hi          int,           -- en yüksek sıcaklık
    prcp            real,          -- yağış miktarı
    date            date
);
```

Bu komut listesini **psql** komut satırına aynen buradaki gibi girebilirsiniz (alt satıra geçmek için <enter>'a basmanız yeterli olacaktır). **psql** en sonda yer alan noktalı virgülü görene kadar komutun bitmediğini anlayacaktır.

SQL komutları içinde boşluklar (boşluk karakteri, sekme ve satırsonu karakteri) özgürce kullanılabilir. Yani, yukarıdaki komut listesini siz istediğiniz gibi yazabilirsiniz; hatta hepsini tek bir satıra dahi girebilirsiniz. Yanyana gelen iki tire ("--") açıklama satırları için kullanılır. Bu işaretten sonra yazılan her şey o satırın sonuna kadar ihmal edilecektir. SQL komutlarının normalde büyük–küçük harf duyarlılığı yoktur. Duruma bağlı olarak değişkenlerin çift tırnak içine alınması onların büyük–küçük harf duyarlı olduğunu gösterir (Yukarıda bu yöntem kullanılmamıştır).

`varchar(80)` 80 karakter uzunluğundaki bir dizgeyi tutabilecek bir veri türü belirtir. `int` normal bir tamsayıyı niteler. `real` tek hassasiyetli (single precision) gerçel sayılar için kullanılır. `date` alanı da adından anlaşılacağı üzere tarih saklamak için kullanılır.

`PostgreSQL` standart `SQL` veri türlerinden `int`, `smallint`, `real`, `double`, `char(`N`)`, `varchar(`N`)`, `date`, `time`, `timestamp` ve `interval` ile birlikte diğer genel araç türleriyle zengin bir geometrik tür ailesini destekler. `PostgreSQL` sınırsız sayıda kullanıcı tanımlı veri türleri ile özelleştirilebilir. Dolayısla, tür isimleri `SQL` standardındaki özel durumların desteklenmesinin gerektiği yerler dışında sözdizimsel anahtar sözcükler değildir.

İkinci örnek, şehir isimlerini ve bulundukları coğrafik bölgeleri saklayacaktır:

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

Buradaki `point`, PostgreSQL'e özel veri türüne bir örnektir.

Son olarak, eğer bir tabloya artık ihtiyacınız kalmadıysa ya da onu baştan oluşturmak istiyorsanız şu komutu kullanabilirsiniz:

```
DROP TABLE tabloismi;
```

3.3. Tablolara Satırların Girilmesi

INSERT cümlesi tablolara veri girişi için kullanılır:

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

Veri giriş işlemlerinde tüm verilerin açıkça belirtilmesi gerekir. Sabitler sadece basit rakamsal değerler değillerdir, örnekte görüldüğü gibi tek tırnak içine alınıp belirtilmeleri şarttır. `date` türü esnek bir çeşit olduğundan neredeyse girilen tüm tarih çeşitlerini kabul eder. Fakat biz bu belgede belirsizlik yaratmaması açısından bu örnekteki biçimi kullanacağız.

point veri türü için bir koordinat çiftine ihtiyacımız olacak:

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

Çok sayıda sütun olduğunda bilginin hangi sırada girileceğini hatırlamanız zorlaşır. Sütun isimlerinin de belirtilebileceği bir sözdizimi bunu kolaylaştırır:

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

İsterseniz, yukarıda bahsedilen yöntemi kullanarak verileri gireceğiniz sütunların yerlerini değiştirebilir ya da hiç yokmuş gibi farzedebilirsiniz, örneğin yağış miktarını yoksayalım:

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

Birçok geliştirici mevcut sütun sırasına göre veri girmektense sıralamayı açıkça belirtmeyi tercih eder.

Lütfen yukarıdaki komutların hepsini girin ki, ileride üzerinde alıştırma yapabileceğimiz bir kaç verimiz olsun.

Çok fazla komutu teker teker girmek yerine bunların hepsini tek bir metin dosyasından **COPY** cümlesi ile okutabilirsiniz. **COPY** cümlesi sırf bu amaç için tasarlandığından **INSERT** cümlesine göre daha hızlı çalışmasına karşın, onun kadar esnek değildir. Bir örnek:

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

Belirtilen dosya sunucunun erişebileceği bir yerde olmalıdır, istemcinin değil. **COPY** cümlesi hakkında daha fazla bilgi için PostgreSQL 8.0 belgelerindeki **COPY** cümlesinin açıklamasına[B12] bakınız.

3.4. Bir Tablonun Sorgulanması

Bir tablodan verileri almak için tablo ***sorgulanır***. Bunun için bir SQL cümlesi olan **SELECT** kullanılır. Cümle, bir seçim listesi (istenen sütunları içeren bir liste), bir tablo listesi (verilerin alınacağı tabloların listesi) ve isteğe bağlı bir niteleme (sınırlamaların belirtildiği kısım) içerir. Örneğin, weather tablosundaki satırların tamamını almak için şunu yazın:

```
SELECT * FROM weather;
```

Burada *, "tüm sütunlar" anlamına gelen bir kısayoldur.[2]

Yani, aynı sonuç böyle de alınacaktır:

```
SELECT city, tmp_lo, tmp_hi, prcp, date FROM weather;
```

Çıktı şöyle olmalıdır:

```
     city      | tmp_lo | tmp_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)
```

Seçim listesinde sadece sütun isimlerini değil, ifadeleri de kullanabilirsiniz. Örnek:

```
SELECT city, (tmp_hi+tmp_lo)/2 AS tmp_avg, date FROM weather;
```

Bunun çıktısı şöyle olacaktır:

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

AS deyiminin çıktılanacak sütunu yeniden isimlendirmekte nasıl kullanıldığına dikkat edin. (AS deyimi isteğe bağlıdır.)

Bir sorgu, istenen satırların yerini belirtmek üzere bir WHERE deyimi eklenerek nitelikli yapılabilir. WHERE deyimi bir mantıksal ifade içerir ve sadece mantıksal ifadeyi doğrulayan satırlar döndürülür. Niteleme amacıyla mantıksal işleçlere (AND, OR ve NOT) izin verilir. Örneğin, San Francisco'nun yağışlı olduğu günleri bulalım:

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

Sonuç:

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

Sorgu sonucunun sıralanmış olmasını da isteyebilirsiniz:

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

Sonuç:

```
     city      | tmp_lo | tmp_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
```

Bu örnekte, sıralamanın nasıl yapılacağı tam olarak belirtilmemiştir, dolayısıyla hangi San Fransisco satırının önce geleceği belli olmaz. Fakat aşağıdaki sorgu daima bu sıralamayla dönecektir.

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

Bir sorgunun sonucundan yinelenmiş satırların kaldırılmasını isteyebilirsiniz:

```
SELECT DISTINCT city
    FROM weather;
```

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

Burada da yine satırların sırası her sorguda farklı olabilir. Sonucun istediğimiz sırada olmasını DISTINCT ve ORDER BY deyimlerini birlikte kullanarak sağlayabilirsiniz[3]:

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

3.5. Tablolar Arası Katılım

Buraya kadar, yaptığımız sorgulamalarda her seferinde sadece bir tabloya erişildi. Oysa sorgulamalar aynı andan birden çok tabloya erişebildiği gibi, aynı tabloya birden fazla kez erişerek satırlara daha çeşitli yaptırımlar uygulayabilir. Aynı anda birden fazla satır ya da birden fazla tabloya erişen sorgulara ***katılımlı sorgu*** denir. Sözgelimi (daha önce oluşturduğumuz tablolardaki) tüm şehirlerin hava durumlarını ve konumlarını aynı anda listelemek istiyoruz. Bunun için `weather` tablosundaki tüm `city` sütunları ile `cities` tablosundaki tüm `name` sütunlarını karşılaştırıp, aynı olan satır çiftlerini seçmek gerekir.

Bilgi

Bu sadece kavramsal bir modeldir. Katılımlı sorgular, aslında, her olası satır çiftini karşılaştırmaktan biraz daha verimli bir anlamda uygulanır ama bu işlemi kullanıcı görmez.

Yukarıda bahsedilen işlemi şu sorgu ile elde edebiliriz:

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

```
     city      | tmp_lo | tmp_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)
```

Çıktıda dikkat edilmesi gereken iki nokta bulunmakta:

- Haywrad şehri için hiçbir çıktı alınmadı dikkat edildiyse. Bunun nedeni ise `cities` tablosunda Hayward adlı bir şehir olmaması ve dolayısıyla `JOIN` bu şehri eledi. İleride bunun nasıl düzeltilebileceği üzerinde durulacak.
- Bir diğer dikkat çeken nokta ise, şehirlerin adını yazan iki tane sütun olması. Bunun sebebi `weather` ve `cities` tablosunun birleştirilmesidir. Pratikte bu istenmeyen bir sonuçtur. Böyle bir durumda buna neden olan `*` ifadesi yerine açıkça listelenmesini istediğimiz sütunları yazarak bu işi halledebiliriz:

  ```
  SELECT city, tmp_lo, tmp_hi, prcp, date, location
      FROM weather, cities
      WHERE city = name;
  ```

Alıştırma:

`WHERE` deyimi kalktığında ortaya çıkan sonucun nedenini bulmaya çalışın.

Tablolardaki tüm sütun isimleri farklı olduğundan çözümleyici hangi ismin hangi tabloya ait olduğunu bulur. Ama bunu daha da açıkça belirtmek isimler aynı olduğunda dahi sorun çıkmasını önler ve tavsiye edilen de budur:

```
SELECT weather.city, weather.tmp_lo, weather.tmp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

Şimdiye kadar gördüğümüz katılım sorguları ayrıca şu şekilde de yazılabilir:

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

Bu sözdizimi yukarıdaki örneklerden biri için çok kullanılan bir sözdizimi değildir, ama bundan sonraki konuları anlayabilmek için yardımcı olacağından burada gösterdik.

Şimdi Hayward kayıtlarına nasıl kavuşacağımızı işleyeceğiz. İstediğimiz şey `weather` tablosu üzerinde tarama yapıp, `cities` tablosunda bunlarla eşleşen satırları bulmak. Eğer `cities` tablosunda herhangi bir eşleşme bulamazsak, o sütun `cities` tablosu alanında boş gözükecek. Bu tür sorgulama işlemleri ***haricen katılım*** (outer join) olarak bilinir. (Şimdiye kadar gördüğümüz katılım sorgularında ise hep ***dahilen katılım*** (inner join) kullanmıştık.) Komut şöyle görünür:

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

     city      | tmp_lo | tmp_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)
```

Bu sorguya ***sola haricen katılımlı*** (left outer join) sorgu denir. Böyle adlandırılmasının sebebi soldaki tablonun tüm satırları en az bir kere listelenirken, sağda yer alan tablonun sadece soldaki tablonun satırlarıyla eşleşen satırlarının listelenmesidir. Bir sol–tablo satırı çıktılanırken sağ–tabloda bu satırla eşleşen bir satır yoksa, sağ–tablonun sütunları boş kalır.

Alıştırma:

Ayrıca, ***sağa haricen katılımlı*** (right outer join) ve ***iki yönlü haricen katılımlı*** (full outer join) sorgu türleri de var. Bunların ne yaptığını da siz bulmayı deneyin.

Ayrıca, bir tabloyu kendine katılımlı olarak da sorgulayabiliriz ve buna ***kendine katılımlı*** sorgu denir. Bir örnek olarak, diğer hava durumu kayıtlarının sıcaklık aralığı içinde kalan hava durumu kayıtlarını bulmak isteyelim. Yani, her `weather` satırının `tmp_lo` ve `tmp_hi` sütununu diğer `weather` satırlarının `tmp_lo` ve `tmp_hi` sütunu ile karşılaştıracağız. Bunu şu sorgu ile yapabiliriz:

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

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

Burada katılımın sol ve sağ taraflarını ayırabilmek için `weather` tablosunu `W1` ve `W2` olarak yeniden isimlendirdik. Ayrıca, bu çeşit isimlendirmeleri aynı şeyleri uzun uzadıya yazmaktan kaçınmak için diğer sorgularda da kullanabilirsiniz. Örnek:

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

Bu tarz kısaltmalarla sıkça karşılaşacaksınız.

3.6. Ortak Değer İşlevleri

Çoğu ilişkisel veritabanı ürünü gibi PostgreSQL'de ortak değer işlevlerini destekler. Bir ortak değer işlevi çok sayıda satırı girdi olarak alır ve bunlardan tek bir sonuç elde eder. Belli bir satır grubu üzerinde işlem yaparak, bunların sayısını bulan `count`, değerlerinin toplamını bulan `sum`, değerlerinin ortalamasını hesaplayan `avg`, en büyük ve en küçük değerleri bulan `max` ve `min` işlevleri bunlara birer örnektir.

Örnek olarak, düşük sıcaklık değerlerinin en yükseğini bulalım:

```
SELECT max(tmp_lo) FROM weather;
 max
-----
  46
(1 row)
```

Eğer bu sıcaklığın hangi şehir (veya şehirlerde) ortaya çıktığını bulmak istersek,

```
SELECT city FROM weather WHERE tmp_lo = max(tmp_lo);     YANLIŞ
```

bu çalışmaz, çünkü `max` işlevi `WHERE` deyiminde kullanılamaz. (Böyle bir sınırlamanın olmasının sebebi, `WHERE` deyiminin ortak değeri bulunacak satırların belirlenmesinde kullanılmak zorunda olmasıdır; yani, deyim, işlevden önce değerlendirilmiş olmalıdır.) Bu durumda böyle bir sorunu gidermek için sorgunun yeniden durumlanabilmesini sağlayan aşağıdaki gibi bir ***altsorgu*** (subquery) kullanılır:

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

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

Şimdi her şey yolunda. Çünkü ortak değer bulma bir altsorgu ile yapıldıktan sonra sonuç dış sorguda karşılaştırma değeri olarak kullanıldı.

Ortak değer işlevleri `GROUP BY` deyimi ile kullanıldığında oldukça yararlıdır. Örneğin her şehrin en yüksek düşük sıcaklığını bulmak için şunu yazabiliriz:

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

Bu bize her şehir için bir değer verecektir:

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

Burada, satırlar şehirlere göre gruplanmakta, her gruptaki satırlar üzerinde max işlevi hesaplanmakta ve sonuçlar listelenmektedir. Hesaplamaya dahil olacak satırları HAVING deyimini kullanarak gruplayabiliriz:

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

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

Sadece tmp_lo değeri 40'ın altında olan şehirleri listelelemsi dışında bu cümle de aynı sonucu verir. Eğer bir de bu işi abartıp sadece "S" ile başlayan şehir isimlerini istersek:

```
SELECT city, max(tmp_lo)
    FROM weather
    WHERE city LIKE 'S%'  ①
    GROUP BY city
    HAVING max(tmp_lo) < 40;
```

① LIKE işleci kalıp eşleştirmesi yapar ve PostgreSQL 8.0 belgelerindeki Kalıp Eşleme[B13] bölümünde açıklanmıştır.

SQL dilinde WHERE ve HAVING deyimlerinin ortak değer işlevleri ile nasıl etkileşime girdiğinin anlaşılması önemlidir. WHERE ve HAVING deyimleri arasındaki temel fark şudur: WHERE satırları, gruplar ve ortak değerler hesaplanmadan önce seçer (ortak değer hesaplamasında kullanılacak satırları seçer), HAVING deyimi ise ortak değerler hesaplandıktan ve gruplamalar yapıldıktan sonra işleme sokulur. Sonuç olarak, WHERE ifadelerinde (altsorgu dışında) ortak değer bulma işlemleri kullanılmazken, HAVING deyimlerinde kaçınılmazdır. (Aslında HAVING deyimleri içinde ortak değer işlevleri dışında ifadeler de kullanmanıza izin verilmiştir ama, bu biraz savurganlık olur; böyle bir koşulu WHERE deyiminde kullanmak daha verimlidir.)

Önceki örnekte, WHERE deyiminde bir ortak değer bulma işlemine ihtiyaç duyulmadığından, şehir isimlerine kısıtlama uygulamıştık. Bu, kısıtlamanın HAVING ile sağlanmasından daha uygundur; çünkü gruplamanın ve ortak değer hesaplamasının WHERE sınamasından geçemeyen satırlar için yapılması gereksizdir.

3.7. Verilerin Güncellenmesi

Mevcut satırları UPDATE cümlesini kullanarak güncelleyebilirsiniz. Farzedelim ki, ayın 28'inden sonraki tüm sıcaklıkların 2 derece daha az olması gerektiğini fark ettiniz. Bu güncelleme işlemini şöyle yapabilirsiniz:

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

Verinin yeni durumuna bakalım:

```
SELECT * FROM weather;

     city      | tmp_lo | tmp_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)
```

3.8. Veri Silme

Bir tablodan satırları silmek için `DELETE` cümlesini kullanabilirsiniz. Hayward'ın hava durumuyla artık ilgilenmediğinizi varsayalım. Tablodan bu satırları silmek için şunu yazabilirsiniz:

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

Böylece, `weather` tablosundan Hayward ile ilgili kayıtlar silinir.

```
SELECT * FROM weather;
```

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

Özellikle sakınılması gereken şöyle bir sorgulama da var:

```
DELETE FROM tabloismi;
```

Bir niteleme olmaksızın, `DELETE` cümlesi belirtilen tablodaki *bütün* satırları silecek, tabloyu boşaltacaktır. Üstelik, sistem bunu yapmadan önce sizden bir doğrulama da istemeyecektir!

4. Gelişmiş Özellikler

Bundan önceki bölümlerde bir `PostgreSQL` veritabanına nasıl veri girilip, onlara nasıl erişileceğini işledik. Bu bölümde ise PostgreSQL'in daha gelişmiş özellikleri olan verileri nasıl daha kolay idare edilebileceği ve veri kaybına ya da bozulma riskine karşı alınacak önlemlerden bahsedeceğiz. En sonunda, PostgreSQL'in bir kaç ek özelliğine göz atma şansımız olacak.

Bu kısımda *SQL Dili* (sayfa: 8) kısmında gördüğünüz örnekleri çoğaltma ve geliştirme şansınız olacak, ki bu yüzden bu kısmın da dikkatle okunması sizin lehinize olur. Bu bölümden sonraki alıştırmaları ise `tutorial/advanced.sql` dosyasında bulabilirsiniz. Bu dosya ek olarak burada tekrar belirtmeyeceğimiz ama yüklenmesi gereken örnek veriler içermektedir. (*SQL Dili* (sayfa: 8) sayfasında dosya kullanımı açıklanmıştır.)

4.1. Sanal Tablolar

Bu bölümü okumadan önce *Tablolar Arası Katılım* (sayfa: 12) bölümünü okumanızı öneririz. Diyelim ki, uygulamanızda hava durumu kayıtları ile şehirlerin yerlerinin birarada listelenmesi ile çok ilgileniyorsunuz. Bunun için bir sorgu oluşturup bu sorguya isim verebilir ve bu sorguya herhangi bir tabloya erişir gibi erişebilirsiniz.

```
CREATE VIEW myview AS
    SELECT city, tmp_lo, tmp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

Sanal tablo kullanımı iyi SQL veritabanı tasarımında önemli bir rol oynar. Sanal tablolar, tablolarınızdaki yapının ayrıntılarını toparlamanızı mümkün kılarak, arkasında kararlı bir arayüz olarak uygulamanızın gelişimini değiştirebilir.

Sanal tablolar, gerçek bir tablonun kullanılabildiği hemen her yerde kullanılabilir. Fakat, sanal tabloları başka sanal tabloları oluşturmak için kullanmak pek iyi bir yöntem değildir.

4.2. Anahtarlar

SQL Dili (sayfa: 8) kısmındaki `weather` ve `cities` tablolarını tekrar ele alalım ve `weather` tablosuna girilecek kayıtlardan `cities` tablosundaki kayıtlarla eşleşmeyecek olanlarının tabloya girilmeyeceğinden emin olmak istediğinizi varsayalım. Buna ***verilerin göreli bütünlüğünün sağlanaması*** diyoruz. Basitleştirmeli veritabanı sistemlerinde bu şöyle gerçeklenir: Önce `cities` tablosunda eşleşen bir kaydın olup olmadığına bakılır ve yeni `weather` kaydının tabloya girilip girilmeyeceğine karar verilir. Bu yaklaşım çok sakıncalı sorunlar içerir, ancak `PostgreSQL` bunu sizin için yapabilir.

Tabloların yeni bildirimleri şöyle olurdu:

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

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

Şimdi geçersiz bir kaydı girmeye çalışalım:

```
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_fkey"
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

Anahtarların davranışları uygulamanıza en iyi şekilde uyarlanabilir. Bu eğitmende bu basit örnekten daha ileri gitmeyeceğiz, fakat daha fazla bilgi edinmek isterseniz, `PostgreSQL` 8.0 belgelerindeki Veri Tanımlama[B18] kısmına bakabilirsiniz. Anahtarları doğru şekilde kullanarak veritabanı uygulamalarınızın kalitesini oldukça arttırabilirsiniz, dolayısıyla anahtar kullanımını iyi öğrenmenizi öneririz.

4.3. Hareketler

Hareketler tüm veritabanı sistemlerinin en temel konularından biridir. Bir hareketin başlıca özelliği ya hep ya hiç şeklinde uygulanmak üzere çok sayıda adımın tek bir adım haline getirilmesidir. Hareketi oluşturan adımlar arasındaki işlemler onunla işbirliği yapan diğer hareketlere görünür değildir ve hareketin tamamlanmasını engelleyen bazı olumsuzluklar olduğunda hareketi oluşturan adımların hiçbiri veritabanını etkilemez.

Örneğin, bir bankanın şube hesapları (branches) olsun ve bu hesaplarda çeşitli müşteri hesapları (accounts) ve bu hesaplarda da bir miktar nakit (balance) bulunsun. Alice'in hesabından 100.00 doları Bob'ın hesabına geçirmek istediğimizi kabul edelim. Son derece basitleştirerek, SQL komutları şöyle olurdu:

```
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');
```

Bu komutların ayrıntılarının burada bir önemi yoktur; önemli olan bunun basit işlemler olarak değil, ayrı ayrı güncellemelerin hepsinin birden yapılmasıdır. Bankamızın memurları bu işlemlerin ya hepsinin yapılmasını ya da hiçbirinin yapılmamasını ister. Eğer Bob, Alice'in hesabından yeterli miktarı alamazsa ya da Alice'in hesabından gerekli miktar alındığı halde Bob'un hesabına geçmezse sistemin hata vermesinden başka her iki müşteri de memnun olmayacaktır. Yani, eğer işlemlerden biri gerçekleşmezse bu adımların hiçbirinin veritabanını etkilemeyeceğini garantilemeliyiz. Yapılacak işlemlerin bir ***hareket*** içinde gruplanması bize bu garantiyi sağlar. Bir hareket ***atomik*** olmalıdır, denir: diğer hareketler açısından ya tüm adımların hepsi gerçekleşmeli ya da hiçbiri gerçekleşmemelidir.

Kesinlikle emin olmamız gereken bir nokta ise bir hareket başarı ile işlemi yürütmüş olsa bile, bilginin tamam olarak veritabanına geçip geçmediğidir, son anda bir sistem kaynaklı hata olsa bile. Örneğin, Bob'un hesabından para çekmeye çalıştığımızda, o daha bankanın kapısında çıkmadan, paranın bir hata sonucu onun hesabında çekilmiş olarak gözükmemesi gibi bir şansı göze alamayız. Tam bu noktada bir veritabanı, bir hareketle ilgili tüm işlemler yapılıp kayıtlar sabit disk gibi bir saklama alanına aktarılmadan 'tamam' sonucunu göndermez.

Bir diğer önemli nokta ise çok sayıda hareket aynı anda çalışırken birbirlerinin tamamlanmamış sonuçlarını görmemesi gerektiğidir. Örneğin bir hareket tüm şubelerdeki (branches) hesap miktarlarını toplarken başka bir hareket Alice ya da Bob'un hesabı üzerinde işlem yapamayacaktır. Kısaca bir hareket tamam benim işim bitti demeden, diğer bir hareket bir işlem başlatamayacaktır.

PostgreSQL'de bir hareket, **BEGIN** ve **COMMIT** SQL komutları ile sarmalanmış adımlardan oluşur. Bu durumda, banka işlemlerimiz aslında şöyle görünecektir:

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

Hareketin de belli bir noktasında işlemin iptal edilmesi gerekebilir (Mesela Alice'in hesabı aktarılacak miktar için yetmeyip negatife düşerse), bunun için **COMMIT** yerine **ROLLBACK** kullanabiliriz ve böyle bir durumda tüm güncellemeler iptal edilir.

PostgreSQL aslında her SQL cümlesini sanki bir hareket gerçekleştiriyormuş gibi ele alır. Bir **BEGIN** komutu kullanmazsanız, her cümlenin başına örtük bir **BEGIN** ve cümle başarılı ise sonuna da örtük bir **COMMIT** getirilir. Bu nedenle, **BEGIN** ve **COMMIT** komutları ile sarmalanmış cümlelere bazan ***hareket kümesi*** de dendiği olur.

Bilgi

Bazı istemci kütüphaneleri **BEGIN** ve **COMMIT** komutlarını kendiliğinden koyar, böylece istemeden hareket kümelerinin etkileriyle karşılaşırsınız. Bu bakımdan kullandığınız arayüzün belgelerine bakmayı unutmayın.

Bir hareketi içinde kayıt noktaları belirterek cümle cümle denetlemek de mümkündür. Kayıt noktaları bir hareketin belli parçalarını seçerek iptal etmeyi mümkün kılar. Bir kayıt noktasını **SAVEPOINT** ile tanımladıktan sonra

ihtiyaç duyarsanız, **ROLLBACK TO** ile bu kayıt noktasına kadar olan kısmı geri sarabilirsiniz. Bir hareketin bu iki komut arasında kalan veritabanı değişiklikleri iptal edilir, fakat, bu bölümden önce yapılanlar veritabanında kalır.

Bir kayıt noktasına kadar geri sarıldıktan sonra, işlem bu noktadan devam eder, öyle ki, bu defalarca yapılabilir. Tersine, belli bir kayıt noktasına geri sarmaya artık ihtiyaç duymayacağınızdan emin olduğunuzda, onu serbest bırakabilirsiniz, böylece sistem bazı özkaynakları serbest bırakabilir. Serbest bırakmanın da, bir kayıt noktasına geridönmenin de tanımlanmasının ardından tüm kayıt noktalarının özdevimli olarak serbest bırakılacağını unutmayın.

Bunların hepsi hareket kümesinin içinde gerçekleşir, dolayısıyla, bu işlemlerin hiçbiri diğer veritabanı oturumlarına görünür değildir. Bir hareket kümesini işleme sokulduğunda, geriye sarma işlemleri diğer oturumlara asla görünür olmazken, işleme sokulan diğer eylemler bir birim olarak diğer oturumlara görünür hale gelir.

Bankanın veritabanını hatırlarsanız, Alice'in hesabından Bob'un hesabına 100 dolar aktarmıştık ama daha sonra baktığımızda, paranın Wally'nin hesabına geçmesi gerektiğini keşfetmiş olalım. Bunun için kayıt noktalarını şöyle kullanabiliriz:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- dur bakalım ... Wally'nin hesabını kullanacağız
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

Bu örnek, şüphesiz fazla basit, fakat bir hareket bloğu üzerinde kayıt noktalarınını kullanımı ile ilgili yeterince denetim var. Dahası, sistem tarafından bir hatadan dolayı çıkış istendiğinde, **ROLLBACK TO** bir hareket kümesinin denetimini yeniden kazanmanın tek yoludur, tamamen gerisarma yapılıp tekrar başlanabilir.

4.4. Kalıtım

Kalıtım (miras alma), nesne yönelimli veritabanlarından kaynaklanan bir kavramdır. Bu sayede veritabanı tasarımında ilginç ve yeni olasılıkların yolu açılmıştır.

İki tablo oluşturalım: Bir `cities` tablosu ile bir `capitals` tablosu. Doğal olarak, başkentler aynı zamanda şehirdirler, dolayısıyla şehirleri listelerken dolaylı olarak başkentleri de bir şekilde göstermek isteriz. Biraz akıllıca hareket ederek, şöyle bir şema kullanabilirsiniz:

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

CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int      -- (feet cinsinden)
);
```

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

Çok fazla sorgulamadan bu doğru çalışır, fakat, bazı satırları güncellemek istediğinizde tuhaf şeyler olur.

Bu daha iyi bir çözümdür:

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int     -- (feet cinsinden)
);

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

Bu durumda `capitals` tablosu, `cities` tablosundaki bütün sütunları (`name`, `population` ve `altitude`) miras alacaktır. `name` sütununun türü `text` olup, değişken uzunlukta dizge kabul eden bir PostgreSQL türüdür. Eyalet başkentleri ek bir sütuna, eyalet sütununa sahiptir ve hangi eyaletin başkenti olduğu bu sütunda belirtilir. PostgreSQL'de bir tablo, sıfır ya da daha fazla tablo miras alabilir.

Örneğin, ağağıdaki sorgu, tüm şehirlerin isimleri arasından başkentler de dahil 500 feet'ten daha yüksekteki şehirleri bulmaktadır:

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

Sorgu şöyle dönecektir:

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

Aşağıdaki sorgu ise, eyalet başkenti olmayan şehirlerden 500 feet ve daha yüksekte olan şehirleri bulmaktadır:

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

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

Buradaki `ONLY` deyimi sorgunun sadece `cities` tablosunda yapılacağını `cities` tablosunu miras alan tablolarda yapılmayacağını belirtir. Daha önce bahsettiğimiz **`SELECT`**, **`UPDATE`** ve **`DELETE`** cümleleri de dahil olmak üzere pek çok SQL cümlesi `ONLY` deyimini destekler.

Bilgi

Kalıtım yeterince kullanışlı olduğundan, kullanışlılığını sınırlayan tekil kısıtlar ve anahtarlar ile bütünleştirilmemiştir. Daha fazla bilgi için PostgreSQL 8.0 belgelerindeki Kalıtım[B19] bölümüne bakınız.

4.5. Sonuç

PostgreSQL'in bu eğitmende yeni SQL kullanıcılarına yönelik olanlar dışında, bahsedilmeyen daha bir çok özelliği mevcuttur. Bu özellikler hakkında daha ayrıntılı bilgiyi bu eğitmeni de içeren PostgreSQL 8.0 belgelerinde[B20] bulabilirsiniz.

Daha fazla bilgiye ihtiyaç duyarsanız, PostgreSQL web site[B21]sinde yeterince kaynak bulabilirsiniz.

Notlar

Belge içinde dipnotlar ve dış bağlantılar varsa, bunlarla ilgili bilgiler bulundukları sayfanın sonunda dipnot olarak verilmeyip, hepsi toplu olarak burada listelenmiş olacaktır.

[B1] http://www.postgresql.org/docs/8.0/static/sql.html

[B2] http://www.postgresql.org/docs/8.0/static/client-«interfaces.html

[B3] http://www.postgresql.org/docs/8.0/static/admin.html

[B4] http://www.postgresql.org/docs/8.0/static/installation.html

[B5] http://www.postgresql.org/docs/8.0/static/user-«manag.html

[1] Bunun neden böyle çalıştığına dair açıklama: PostgreSQL kullanıcı isimleri sistem kullanıcı isimlerinden bağımsızdır. Eğer bir veritabanına bağlanıyorsanız, hangi PostgreSQL kullanıcı adı ile bağlanacağınızı belirtebilirsiniz; bunu yapmazsanız, sistem kullanıcısı isminiz öntanımlı PostgreSQL kullanıcı isminiz olarak kullanacaktır. Böyle bir durumda veritabanı sunucusunu başlatan kullanıcı ile aynı isme sahip bir PostgreSQL kullanıcısı olacaktır ve böyle bir durumda bu kullanıcı her zaman veritabanı oluşturma izinlerine sahip olacaktır. Her seferinde o kullanıcı ile sisteme girmektense -U seçeneği ile PostgreSQL'e bağlanmak istediğiniz kullanıcı adını belirtebilirsiniz.

[B6] http://www.postgresql.org/docs/8.0/static/app-«createdb.html

[B7] http://www.postgresql.org/docs/8.0/static/app-«dropdb.html

[B8] http://www.postgresql.org/docs/8.0/static/client-«interfaces.html

[B9] http://www.postgresql.org/docs/8.0/static/app-«psql.html

[B10] http://www.postgresql.org/docs/8.0/static/biblio.html#MELT93

[B11] http://www.postgresql.org/docs/8.0/static/biblio.html#DATE97

[B12] http://www.postgresql.org/docs/8.0/static/sql-«copy.html

[2] SELECT * kolay bir sorgulama olarak kullanışlı gibi görünse de, tabloya bir sütun eklemek sonuçları değiştireceğinden uygulamada çoğunlukla kötü bir tarz olarak kabul edilir.

(3) Bazı veritabanı sistemlerinde ve PostgreSQL'in eski sürümlerinde `DISTINCT` gerçeklenimi sıralamayı özdevimli yaptığından `ORDER BY` gereksizdir. Fakat bunun böyle olması SQL standardının bir zorlaması değildir ve şimdiki `PostgreSQL DISTINCT` deyiminin satırları sıralayacağını garanti etmemektedir.

(B13) http://www.postgresql.org/docs/8.0/static/functions-«matching.html

(B18) http://www.postgresql.org/docs/8.0/static/ddl.html

(B19) http://www.postgresql.org/docs/8.0/static/ddl-«inherit.html

(B20) http://www.postgresql.org/docs/8.0/static/index.html

(B21) http://www.postgresql.org

Bu dosya (pgsql-tutorial.pdf), belgenin XML biçiminin TEXLive ve belgeler-xsl paketlerindeki araçlar kullanılarak PDF biçimine dönüştürülmesiyle elde edilmiştir.

20 Ocak 2007