Bölüm 9

Veri Tabanları

Bu Bölümdekiler

- İlişkisel Teori ve Kavramları
- İndeksler, Görüntüler, Tetikler
- Oracle, PostgreSQL ve MySQL'i Unix üzerinde kurmak

Her kurumsal uygulamanın etrafında döndüğü, bir depoya alıp verdiği, sorgulayıp bulduğu ve biriktirdiği metin ve sayısal bilgilerin toplamına *veri* diyoruz. Bir kurumsal uygulamada veriye erişim, en hayati noktalardan biridir. İşyerleri verilerinin doğruluğuna ve bütünlüğüne çok önem verirler. Hâtta kitabımızın konusu olan kurumsal uygulamalar için denebilir ki, aslında bir uygulamanın yegâne görevi veri tabanındaki veriyi göze güzel bir şekilde sunmaktan ve göze güzel gelen şekilde almaktan ibarettir.

Veri depolama ve geri almanın modellerinden biri olan ilişkisel model, E. F. Codd [9] tarafından ortaya sürülmüş ve günümüzdeki modern tüm ilişkisel veri tabanların işleyişini tanımlayan bir teorik altyapıdır. Bu modelin temeli küme teorisine dayanır, ve tamamıyle tanımlı, iç bütünlüğü kurulmuş sağlam matematiksel bir yapıdır. Yapı öyledir ki, bir ilişkisel model üzerinde işleyen bir sorgunun "doğru olup olmadığını" bile matematiksel olarak ispatlayabilirsiniz[1]. İlişkisel model, ortaya atıldığı 70'li yıllarda, akademik çevrelerde müthiş bir çarpışmaya sebebiyet vermişti. O vakitte yarışta olan diğer veri modeli olan hiyerarşik model, ilişkisel model ile girdiği çarpışmadan yenik ayrılmıştır[2]. Akabinde ilişkisel modeli ticari ürün hâline getiren ve SQL'i ilk destekleyen şirketlerden olan Oracle, bu alana çok hızlı girmesiyle müthiş bir piyasa hakimiyetine ve finansal başarıya kavuşmuştur. Önyüze odaklı olan Microsoft'tan sonra bir servis tarafı teknolojisi satıcısı olan Oracle'ın ikinci büyük yazılım şirketi olması, veri depolaması ve erişiminin önemine işaret etmektedir.

Günümüzde artık SQL dili ile erişilen ilişkisel veri tabanları, ticariden açık yazılıma, küçükten büyüğe giden geniş bir yelpazede bulunabilen hale gelmişlerdir. Günümüzde eğer finans, telekom, sağlık sektöründen biri için bir kurumsal yazılım geliştiriyorsak, arka plandaki veri tabanını olarak ilişkisel bir teknoloji bulmamızın olasılığı artık çok yüksektir.

Verinin depolama ve erişimin bu seviyede önemi sebebiyle, verinin tutulduğu ilişkisel veri tabanlarını, ilişkisel şemaların altyapısını tanımlayan ilişkisel modeli, ve tüm teori ve teknolojinin bir araya geldiği SQL erişim dilini öğreneceğiz.

9.1 İlişkisel Model

İlişkisel modelin temeli, birden fazla kolonu birarada tutarak onlar arasında bir *alakâ* kurulmasını sağlayan *ilişki* kavramıdır [10, sf. 138]. İlişki, modern veri tabanlarında bir *tabloya* tekabül eder. O zaman bir tabloda, yâni bir ilişkide, tipik olarak birden fazla kolon olur. Tablo 9.1 üzerinde örnek bir ilişki görüyoruz.

Eğer bir tabloya veri eklemek istiyorsak, o tablonun ilişki yapısına uyan verileri SQL ile veri tabanına verebiliriz. Bunun için SQL'de `INSERT` komutunu kullanabiliriz. Örnek bir INSERT ibaresi altta gözükmektedir.

[1] Aynı şekilde bir analiz ne yazık ki yazılım mühendisliği gibi alanlarda mümkün olmamaktadır, çünkü bu alanlardaki eşya makina değil, insandır

[2] Nesnesel veri tabanları da hiyerarşik modelin günümüzdeki yansımalarıdırlar, ve bir türlü sektörde kabul görmemelerinin sebebi, 70'lerdeki savaşı kaybetmiş olmalarının bir sonucudur

Tablo 9.1: Örnek Tablo (Car)

LICENSE_PLATE	DESCRIPTION

Tablo 9.2: Örnek Tablo (Veri İle)

LICENSE_PLATE	DESCRIPTION
34 TTD 2202	sarı araba
34 GD 22	kırmızı araba
35 TG 54	benim tarifim
16 RR 344	bu araba ikinci el
33 RE 43	sarı araba

```
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('34 TTD 2202', 'sarı araba');
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('34 GD 22', 'kırmızı araba');
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('35 TG 54', 'benim tarifim');
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('16 RR 344', 'bu araba ikinci el');
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('33 RE 43', 'sarı araba');
..
```

Bu komut, `CAR` adlı tabloya beş satır veri ekleyecektir. Komut işledikten sonra sonuç, Tablo 9.2 üzerindeki gibi gözükecektir. `CAR` tablosundan veri almak için, SQL'in `SELECT` komutu kullanabiliriz.

```
SELECT * from CAR;
```

komutu, `CAR` arabasındaki tüm verileri ve tüm kolonları (`*` işareti ile) geri getirir. Eğer geri getirilen veriler üzerinde (satır bazında) filtreleme yapmak istiyorsak, `SELECT` komutuna `WHERE` komutunu eklemek zorundayız. Meselâ

```
SELECT * from CAR WHERE LICENSE_PLATE='34 TTD 2202'
```

komutu, plakası `34 TTD 2202` olan tüm arabaları (yani tek satır) geri getirecektir.

Bir tablodaki veriyi güncellemek için, `UPDATE` komutunu kullanmamız gerekir. `UPDATE`, güncellenmek istenen verinin yeni hâlini `set` komutundan sonra alır. Güncelleyeceği satırın hangisi olacağını ise `WHERE` komutuna verilen bir filtre değeri ile anlar.

```
UPDATE CAR set DESCRIPTION = 'siyah araba'
WHERE LICENSE_PLATE = '34 TTD 2202'
```

Tablo 9.3: Garage Tablosu

ID	DESCRIPTION
1	Garage 1
2	Garage 2

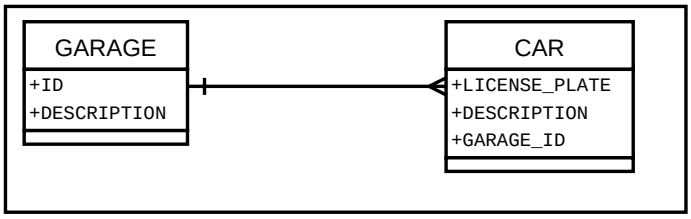
Tablo 9.4: Yeni Car

LICENSE_PLATE	DESCRIPTION	GARAGE_ID
34 TTD 2202	sarı araba	1
34 GD 22	kırmızı araba	1
35 TG 54	benim tarifim	2
16 RR 344	bu araba ikinci el	2
33 RE 43	sarı araba	2

Böylece daha önce 'sarı araba' olan tanım (description) `UPDATE` komutundan sonra 'siyah araba' hâline gelecektir.

9.1.1 Tablo Arası İlişkiler

İlişkisel modelde kolonların arasında ilişki kurulduğu gibi (bir tablo), tablolar arasında da ilişki kurmak mümkündür. Bunu yapmak için ilişkisel model, yabancı anahtar (foreign key) kavramını kullanır. Yabancı anahtar, bir tablodaki asal anahtar, kimlik niteliği taşıyan bir kolonun, diğer bir tabloya bir işaret edici olarak konmasıdır. Meselâ Tablo 9.3 üzerinde gösterilen yeni tabloya işaret etmek için, bu tablodaki bir satırı (garajı) tekil olarak seçebilen bir kolon değerini (`ID`), işaret *eden* tablonun üzerine yabancı anahtar olarak (`GARAGE_ID` kolonu) koyarız (Tablo 9.4). Bu ilişkinin kuşbakışı görüntüsü Şekil 9.1 üzerinde görülmektedir.



Şekil 9.1: Garage ve Car İlişkisi

Bütünlük Kontrolleri

İki tablo arasında yabancı anahtar ilişkisi var ise, işaret *eden* tabloya bir veri eklerken, yabancı anahtarın işaret ettiği *diğer* tablodaki satırın mevcut olmasını kontrol etmek, iyi bir prensip sayılır. Meselâ, sisteme bir `CAR` ekliyorsak ve bu sistemin iş kuralları bağlamında "bir arabanın her zaman bir garaj altında olması gerektiğini" biliyorsak, o zaman `CAR` tablosu üzerindeki `garage_id` kolonu üzerinde bir bütünlük kontrolü (integrity constraint) koyabiliriz.

Bütünlük kontolleri, bizim belirttiğimiz bir kolon ve hedef kolon üzerinde, her veri eklendiğinde veri tabanı tarafından yapılan kontrollere verilen isimdir. Bu kontrollere göre eklenen yabancı anahtarın değeri, işaret edilen tablodaki asal anahtar içinde olup olmadığı kontrol edilir; Eğer bu veri yok ise, veri tabanı `INSERT` komutunuza bir hata mesajı ile cevap verecektir. Bütünlük kontrolleri koymak için MySQL tabanında şu komutu kullanabilirsiniz.

```
alter table car add FOREIGN KEY (garage_id) REFERENCES garage(id);
```

Diğer tabanlarda sözdizim farklı olabilir, fakat işi özünde bilmeniz gereken, kaynak ve hedef tablo+kolon ikilisini bilmektir. Yukarıdaki bütünlük kontrolü komutu işletildikten sonra, artık `car.garage_id` kolonuna koyulacak her verinin `garage.id` kolonunda olması mecbur olacaktır.

Birleştirim

İki tablo arasında ilişkiyi kurduktan sonra eğer bir tablodaki satırdan, o satırın ilişkisini olduğu diğer tablodaki satıra atlamak istersek, ilişkisel modelde birleştirim (join) kavramını kullanmamız gerekir. Birleştirim, küme teorisinden gelen bir kavramdır, ve SQL'de şu şekilde gösterilir.

```
SELECT * FROM GARAGE, CAR;
```

Bu komut, her iki tablodaki *her* satırı diğer tablodaki diğer *her* satır ile teker teker kombinasyona sokarak, bir kartezyen kombinasyonu olarak geri getirir. Birleştirime sokulan her tablo, `FROM` ibaresinden sonra birbirinden virgülle ayrılarak belirtilmelidir. Eğer `CAR` tablosunda 5, `GARAGE` tablosunda 2 satır var ise, kartezyen birleşim 10 tane satır geri getirecektir. Kartezyen birleşim sonucunu Tablo 9.5 üzerinde görüyoruz.

Fakat bu birleşim, hiçbir pratik uygulama için faydalı olmayacaktır. Geri gelen sonuç listesini bir şekilde daraltmamız gerekmektedir. Çoğunlukla ihtiyacımız olan, başlangıç tablolarındaki bir tablodan (ve onun bir satırından) bir diğerine (onun bir satırına) atlayabilmektir. O zaman, kartezyen sonuçta *yabancı anahtarı* ve asal anahtarı birbirine uyan satırları bu büyük birleşimden çekip çıkartmayı deneyebiliriz. Çünkü, gördüğümüz gibi, artık yabancı anahtar ve tekil anahtarlar, aynı büyük tablo içinde yanyana gelmişlerdir, ve basit bir `WHERE` şartı, birbirine uyan satırları çekip çıkarmak için yeterli olacaktır.

```
SELECT *
```

Tablo 9.5: Kartezyen Birleşimi

ID	DESCRIPTION	LICENSE_PLATE	DESCRIPTION	GARAGE_ID
1	Garage 1	34 TTD 2202	sarı araba	1
2	Garage 2	34 TTD 2202	sarı araba	1
1	Garage 1	34 GD 22	kırmızı araba	1
2	Garage 2	34 GD 22	kırmızı araba	1
1	Garage 1	35 TG 54	benim tarifim	2
2	Garage 2	35 TG 54	benim tarifim	2
1	Garage 1	16 RR 344	bu araba ikinci el	2
2	Garage 2	16 RR 344	bu araba ikinci el	2
1	Garage 1	33 RE 43	sarı araba	2
2	Garage 2	33 RE 43	sarı araba	2

Tablo 9.6: Filtrelenmiş Kartezyen Birleşimi

ID	DESCRIPTION	LICENSE_PLATE	DESCRIPTION	GARAGE_ID
1	Garage 1	34 TTD 2202	sarı araba	1
1	Garage 1	34 GD 22	kırmızı araba	1
2	Garage 2	35 TG 54	benim tarifim	2
2	Garage 2	16 RR 344	bu araba ikinci el	2
2	Garage 2	33 RE 43	sarı araba	2

```
FROM GARAGE, CAR
WHERE ID = GARAGE_ID;
```

Sonucu Tablo 9.6 üzerinde görüyoruz. Filtrelenmiş kartezyen kombinasyonundan gelen bu şekilde sonuçlar, daha pratik uygulaması olabilecek türden sonuçlardır. Garajlar ve onların altında olan arabalar, artık aynı satırda yanyana hâlde gösterilmiştir. Eğer daha da detaylı bir sonuç görmek istersek, meselâ

Tablo 9.7: Tek Garajın Altındaki Arabalar

ID	DESCRIPTION	LICENSE_PLATE	DESCRIPTION	GARAGE_ID
1	Garage 1	34 TTD 2202	sarı araba	1
1	Garage 1	34 GD 22	kırmızı araba	1

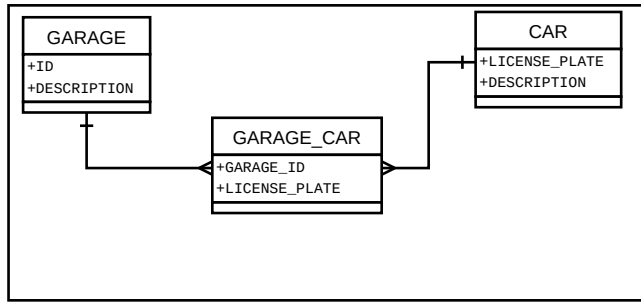
sadece 1 no'lu garaj altındaki tüm arabaları görmek istersek, üstteki sorguyu biraz daha genişleterek şöyle bir SQL kullanabiliriz.

```
SELECT *
FROM GARAGE, CAR
WHERE ID = GARAGE_ID
AND GARAGE_ID = 1
```

Bu sorgunun sonucu Tablo 9.7 üzerinde sergilenmiştir. Artık sadece 2 no'lu ve altındaki arabaları listemizde görmekteyiz.

Eğer geri getirilen kolonları (satırları değil) filtrelemek istiyorsak, `SELECT` komutundan sonra belirtilen kolon listesi ile bu isteğimizi belirtebiliriz. Örneğimizde, `*` işaretini kullanarak tüm kolonları almak istediğimizi belirttik, fakat `SELECT LICENSE_PLATE` ile sadece plaka no'larını almamız mümkün olabilirdi.

Çoka çok türden ilişkiler kurmak için iki tablo arasında bir ara tablo kullanmayı seçebilirsiniz. Bu durumda ilişki, Şekil 9.2 üzerindeki gibi gözükecektir.



Şekil 9.2: Ara Tablo ile İlişki

Ara tabloyu birleştirime dahil etmek için, `FROM`'dan sonra gelen listeye, ara tabloyu da eklemeniz gerekecektir. Bunun yapmanın genel mantığı, iki tablolu kullanım ile aynıdır.

```
SELECT *
FROM GARAGE, CAR, GARAGE_CAR
WHERE ID = GARAGE_ID AND
```

```
CAR.LICENSE_PLATE = GARAGE_CAR.LICENSE_PLATE AND
GARAGE_ID = 1;
```

Bir Tabloda Olan, Ötekinde Olmayan Satırlar

İki tabloda birbirine uyan satırları birleştirim (join) ile getirmek mümkündür. Eğer bir tabloda olan ama bir diğerinden *olmayan* satırları geri getirmek istersek, SQL'in `WHERE NOT EXISTS` komutunu kullanmamız gerekecektir. `NOT EXISTS`, her zaman bir alt sorgu (subsquery) içinde kullanılır. Alt sorgular, SQL sorgularınızı daha detaylandırmak için kullanılan tekniklerden biridir; Bu teknik ile ana sorgudan sonra parantezler içinde üst sorgudaki küme içinden daha detaylı seçimler yapmak mümkündür.

Alt sorguları, çoğu durumda, bir birleştirim formuna çevirmek mümkündür. Fakat sektörümüzde genel kullanım birleştirim ile çözülebilecek problemlerin direk birleştirim kullanılarak çözülmesi yönünde olduğu için, alt sorgular birleştirim sözdizimi kadar rağbet görmemiştir; `NOT EXISTS` kullanımı bunun haricindedir, çünkü standart SQL bağlamında "bir satırda olan ama ötekinde olmayan sonuçları bulma" sorgusunu daha temiz bir şekilde gerçekleştirmenin yolu alt sorgular üzerinden `NOT EXISTS` komutunu kullanmaktan geçer.

`NOT EXISTS` kavramını anlatmak için bir örnek geliştirelim: Bu örnekte iki tablo yaratalım, `passengers` (yolcular) `cars` (arabalar) [3]. Bu örnekte yolcular bir treni oluşturan kompartmanlarda oturmaktadırlar. Her yolcu, `passengers` tablosunda ismiyle kimliklendirilecektir, ve kompartmanlar `compartment` kolonu ile `cars` tablosu üzerinde tutulacaktır.

```
CREATE TABLE passengers (
   name VARCHAR(15),
   compartment INT);

INSERT INTO passengers VALUES ('smith',20);
INSERT INTO passengers VALUES ('jones',25);

CREATE TABLE cars (
   compartment INT,
   class VARCHAR(10));

INSERT INTO cars VALUES (20,'compartmant 1');
```

Bu şemaya ve örnek veriye göre, `smith` adlı yolcu `20` no'lu ve `cars` *tablosunda mevcut* bir kompartmanda oturmaktadır. Bizim sorguyla bulmak istediğimiz `jones` adlı yolcudur, çünkü bu yolcu `25` no'lu ve `cars` üzerinde *olmayan* bir kompartmanda oturmaktadır.

[3] http://dev.mysql.com/tech-resources/articles/4.1/subqueries.html

> **Not:** Sadece örnek olarak verilmiş bu şemanın gerçek dünya şartlarına pek uyduğu söylenemez, çünkü bu tür bir uygulamalarda genellikle `cars`'ın asal anahtarı `compartmant` ve `passengers`'daki yabancı anahtar `compartmant` arasında bir bütünlük kontrolü (9.1.1) konacak, ve `passengers`'a yapılan her `INSERT` kontrol edilerek `INSERT INTO passengers VALUES ('jones',25)` komutu bu kontrolden geçmeyecekti. Sadece örnek amaçlı olarak bu şemayı ve veriyi kullanıyoruz, `NOT EXISTS` genellikle arasında yabancı/asal anahtar ilişkisi olmayan tablolar için kullanılır.

`NOT EXISTS` içeren alt sorgu kullanan örneğimizdeki SQL komutu, şöyle olacaktır.

```
SELECT * FROM passengers
WHERE NOT EXISTS
   (SELECT * FROM cars
    WHERE cars.compartment = passengers.compartment);
```

Bu sorguda üst sorgu, `passengers` tablosundaki verilerden sorgulamaya başlamıştır. Üst sorgudan sonra parantez içinde bir sorgu daha görüyoruz, demek ki alt sorgu mevcuttur. Alt sorguların işleyişini hayal etmek için şöyle düşünmek faydalıdır; Sanki üst sorgudaki her satırın teker teker alt sorguya verildiğini düşünün. Eğer üst sorguda `NOT EXISTS` kullanımı mevcut ise, ve alt alt sorgudan, üst sorgudan verilen artı kendi yaptığı bazı filtrelemeler sonucunda "hiç bir sonuç" geri gelmez ise, üst sorgudaki `NOT EXISTS` şartına uyulmuş olacağı için, üstten alta verilmiş olan satır, başarılı bir satır olarak geri döndürülecektir.

Alt sorguda `cars.compartment` kullanımına dikkat çekmek isterim; Yâni üst sorgudaki tabloya alt sorgudan erişmek mümkündür (ama tersi mümkün değildir). Ayrıca bu durum önceki paragrafta bahsettiğimiz üstten alta teker teker gönderilen satırlar betimlemesine de uygundur.

Bu örnek sorgudan cevap olarak tek bir `passengers` satırı geri gelecektir; O da `25` no'lu kompartmanda oturan `jones` adlı kişidir.

9.1.2 Veri Modelini Normâlleştirmek

Normâlleştirme (normalization), ilişkisel bir veri tabanı modelinin daha arı hale getirilmesidir. Normâlleştirme, aynı zamanda "ilk taslak" veri tabanı tasarımının üzerinde revizyonlar yapmanının yolu, taslağı son haline yaklaştırmanın yöntemlerden birisidir. Normâlleştirmenin altyapısı matematikseldir, aynen ilişkisel modelin kendisinin matematiksel olması gibi. Temel alınan kavram, işlevsel bağlılık (functional dependency) denen bir kavramdır. Normalleştirmenin amaçları şunlardır.

Veri Bütünlüğü: Sadece bu öğe bile normalleştirme ile uğraşmak için yeterli bir sebeptir. Veri, bütünlüğü bozulmamış bir şekilde kalır, çünkü her çeşit bilgi, sadece bir kere tek bir yerde saklanır. Yani, bir veri çeşidinin kopyasının

veri tabanının değişik çizelgeleri üzerinde saklanması gerekmez. Eğer veri gereksiz şekilde kopyalanmış ise, bu değişik kopyalar, kopyalardan habersiz olan uygulama kodları yüzünden bir süre sonra birbirinden farklı değerleri taşımaya başlayabilirler. Bu, doğruluk ve tutarlılık açısından çok kötü bir sonuçtur. Bu gibi durumlarda ilişkisel veri taban paketinizin otomatik bütünlük kontrol (automatic integrity check) mekanizmaları bile işe yaramaz. Tedavinin, uygulama seviyesinde yapılması gerekir. Fakat bu da uygulama programlarını daha kompleks hâle getirecek, dolayısıyla bakımını zorlaştıracaktır.

Uygulamadan Bağımsız Veri Modeli: Normalleştirme, genelde bilinen ve takip edilen "ilişkisel model, verinin içeriğine göre kurulmalı, uygulamaya göre değil" kavramını bir adım daha öne alır. Bu sayede veri modeli, üzerinde onu kullanan uygulama değişse bile daha tutarlı, sabit ve değişmez olarak kalacaktır. Uygulama programının gereksinimlerinin veri tabanı mantıksal (logical) model üzerindeki etkisi sıfır olmalıdır. Daha ileride göreceğimiz gibi, uygulama, mantıksal model üzerinde değil, fiziksel (physical) model üzerinde etki yapar.

Depolama Yeri Azaltımı: Yabancı/göstergeç anahtarların haricinde, tamamiyle normâlleştirilmiş bir veri tabanı gereksiz (kopyalanmış) veri miktarını en aza indirecektir. Kopyalanma miktarı azaldığı için, depo yerine olan ihtiyaçta azalır. Ve gene bu sayede, veri tabanı motorunun arama yapması da daha rahatlaşacaktır.

Birinci Normal Formu

Bu form, tekrar eden hiç bir gurup taşımaz. Bir başka deyişle, bir hücre üzerinde taşınan değer tek ve basit olmalıdır. Aşağıdaki örnek veride, son sene nüfusu %5'den fazla artmış olan şehirlerden bazılarını görüyoruz. Şehir bilgilerinin bazılarının aynı hücre içerisinde guruplandığını görüyoruz. Bu yüzden bu tablo, normal değildir (hâtta 1NF bile değildir).

Bu tablodaki bilgiye bakarak, şehir bilgilerinden mesela `BU_YILIN_NFUSU` kolonundaki nüfuslardan hangisinin hangi şehre ait olduğunu nereden bilebiliriz? Bir kolon içinde bile birçok nüfus ve birçok şehir var. Sorumluluğu veri modelinden uygulama programına atarak, sıraya göre bir eşleme düşünülebilir, fakat bu da en temel ilişkisel veri tabanı kuralı olan 'kolon içinde sıra olmaz' kuralının ihlalidir.

Bu veri yapısını 1NF (birinci normal formu) hâline getirmek için, tekrar eden gurupları (verileri) tek kolondan çıkarıp, değişik satırlara yaymak gerekir. Alttaki tabloda 9.9 üzerinde, tablo 9.8 verisinin 1NF'e getirilmiş halini görebilirsiniz.

Bu yeni veri modelinde, eyalet kolonu asal anahtar (primary key) olarak addedildi. Fakat bu son tabloda da bazı problemler var. Güncellemek ve silmek için, hala birçok veriye teker teker uğrayıp, verinin bütünlüğünü kod yazarak birarada tutmamız gerekiyor. Mesela eğer yeni bir şehir satırı ekleyecek olsak, beraberinde eyalet verisi de eklememiz lazım. Ya da bir eyaletin son şehrini veri tabanından silsek, bu eyaletin verisini veri tabanından tamamen kaybetmiş oluyoruz. Demek ki 1NF'ten daha optimal bir yapıya geçmemiz gerekiyor.

Tablo 9.8: Normal Olmayan Tablo

EYALET	KISALTMA	EYALET NÜFUSU	ŞEHİR	GEÇEN YILIN NÜFUSU	BU YIL NÜFUS	YÜZDE ARTIŞ
North Carolina	NC	5M	Burlington, Raligh	40k, 200k	44k, 222k	10%, 11%
Vermont	VT	4M	Burlington	60k	67.2k	12%
New York	NY	17M	Albany, New York City, White Plains	500k, 14M,		
100k	570k, 14.7M, 106k	8%, 5%, 6%				

Tablo 9.9: 1NF

EYALET	KISALTMA	EYALET NÜFUSU	ŞEHİR	GEÇEN YIL NÜFUS	BU YIL NÜFUS	YÜZDE ARTIŞ
North Coralina	NC	5M	Burlington	40k	44k	10%
North Coralina	NC	5M	Raleigh	200k	222k	11%
Vermont	VT	4M	Burlington	60k	67.2k	12%
New York	NY	17M	Albany	500k	540k	8%
New York	NY	17M	New York City	14M	14.7M	5%
New York	NY	17M	White Plains	100k	106k	6%

İkinci Normal Formu

2NF yapısında 1NF örneğinde gördüğümüz şekilde yarım bağımlılık bulunmaz. Anahtar olmayan her kolon, asal anahtara tamamen bağlı olmalıdır. Anahtar kolon, birden fazla kolonu kapsıyorsa, anahtar olmayan kolonlar anahtar kolonların hepsine tamamen bağlı olmalıdır. Tablo 9.9 bu kritere hâla uymuyor. Şehir bilgileri eyalet bilgisine bağlı değil. Daha detaylı olarak, bütün şehir kolonları (`SEHIR`, `GECEN_YIL_NUFUS`, `BU_YIL_NUFUS`, `YUZDE_ARTIS`) asal anahtar eyalet kolonuna `EYALET` tamamen bağlı durumda değildir.

Bu yüzden, birbirine tam bağlı olmayan bilgileri ayrı tablolara parçalamamız gerekiyor. Tablo 9.10 ve 9.11 üzerinde 2NF şemanın son hâlini görüyoruz.

Tablo 9.10: 2NF Eyalet Tablosu

EYALET	KISALTMA	EYALET_NÜFUSU
North Carolina	NC	5M
Vermont	VT	4M
New York	NY	17M

Tablo 9.11: 2NF Şehir Tablosu

EYALET	KISALTMA	ŞEHİR	GEÇ YIL NÜFUS	BU YIL NÜFUS	YÜZDE ARTIŞ
North Coralina	NC	Burlington	40k	44k	10%
North Coralina	NC	Raleigh	200k	222k	11%
Vermont	VT	Burlington	60k	67.2k	12%
New York	NY	Albany	500k	540k	8%
New York	NY	New York City	14M	14.7M	5%
New York	NY	White Plains	100k	106k	6%

Üçüncü Normal Formu

3NF veri modelinde, anahtar olmayan hiçbir kolon, başka anahtar olmayan kolona bağlı olamaz. 2NF'de bütün kolonların asal anahtara bağlı olduğunu

söylemiştik. 3NF'e göre, bu bağlantı **dolaylı bile olamaz**. Mesela, `YUZDE_ARTIS` kolonuna bakalım. Bu kolon, `GECEN_YILIN_NUFUSU` ve `BU_YILIN_NUFUSU` kolonlarına bağlıdır, çünkü bu iki kolondaki değerlerden **hesaplanır** (Hesaplanan kolonlara literatürde türetilen (derived) kolon ismi de verilmektedir). Üretilen kolonlar asal anahtara bağlılardır ama, bu bağlılık dolaylıdır, yani bağlı oldukları esas iki kolon asal anahtara bağlı olduğu için onlar da asal anahtara bağlılardır.

Üçüncü normal formunda böyle kolonlara izin verilmez. `YUZDE_ARTIS` kolonu 3NF'de şemadan atılması gerekmektedir. Türetilen değerler, uygulama programı tarafından anlık hesaplanacaktır. Eğer hesaplanan değer çok sık erişiliyor ise, o zaman Oracle görüntü kavramı kullanılarak bu hesabın hayali bir tablo çerçevesinde servis edilmesi performans açısından yararlı olabilir.

Veri tabanının 3NF (en son) halini Tablo 9.12, 9.13 ve 9.14 üzerinde görüyoruz.

Tablo 9.12: 3NF

EYALET	EYALET NÜFUS
North Carolina	5M
Vermont	4M
New York	17M

Tablo 9.13: 3NF

EYALET	KISALTMA
North Carolina	NC
Vermont	VT
New York	NY

9.2 Yardımcı Kavramlar

Tablo, SQL gibi ana kavramların üstüne, çoğu veri taban ürünü bazı ek servisleri kullanıcılarına sunmaktadır. Bu servisler, bir tablo üzerinde yapılan silme, güncelleme, ekleme gibi bir işlem olduğu *anda* bir ek komutun işletilmesini sağlayan tetik (trigger) kavramı olabilir. Tetik, normâl proglamlama dillerinden bize tanıdık gelecek bir kavramdır, bir çengel (hook) olarak görülebilir, ve çengel, takıldığı yer aktif olunca kendisi de aktif hâle gelecektir. Diğer kavramlardan görüntü (view), dizi (sequence) ve eşanlam (synonym) sayılabilir.

Tablo 9.14: 3NF

ŞEHİR	KISALTMA	GEÇEN YIL NÜFUS	BU YIL NÜFUS	YÜZDE ARTIŞ
Burlington	NC	40k	44k	10%
Raleigh	NC	200k	222k	11%
Burlington	VT	60k	67.2k	12%
Albany	NY	500k	540k	8%
New York City	NY	14M	14.7M	5%
White Plains	NY	100k	106k	6%

Bu kavramları şimdiye kadar Oracle üzerinde kullandığımız için, bu bölümde de Oracle üzerinden işleyeceğiz. Fakat kendi ürününüz üzerinde bu kavramların karşılıklarını bulabilirsiniz (özellikle PostgreSQL üzerinde, çünkü bu taban özellik olarak Oracle'a en yakın olan açık yazılım ürünüdür).

SID

Veri tabanı kelimesini kullandığımızda genelde birçok şeyden aynı anda bahsediyoruz. Oracle paket programının tamamına, içinde bilgi depolayan kayıtları, kullanıcı isimlerinin toplamina aynı anda veri tabanı deniyor. Fakat, Oracle dunyasinda, bu kavramlari daha da berraklaştırmamız gerekecek. SID= Veri tabanı diyeceğiz, ve, veri tabanı Oracle paket programı değildir gibi bir tanım yapacağız. Peki o zaman, veri tabanı nedir?

Oracle'a göre veri tabanı, hakkında konuşabileceğimiz en büyük kayıt birimidir. İçinde tablolar, onların yazıldığı dosyalar, bu tablolara erişecek kullanıcı isimleri, ve paraloların toplamına veri tabanı diyoruz. Bir proje icinde şu kelimeleri duyabilirsiniz.

- Hangi veri tabanına bağlandın, tablo x'i bu tabanda bulamıyorum...
- (Admin'e) Benim kullanıcı ismimi bu veri tabanında da yaratır mısınız ? Kullanıcı şifrem kabul edilmiyor
- Fazla veri tabanı yaratmaya gerek yok, bir tane üzerinde çalışsak olmaz mı?
- Hayır olmaz, çünkü veri tabanı idarecileri iki veri tabanı olursa idaresi kolaylaşır diyorlar.

SID, veri tabanına erişmemizi saylayacak bir isimden ibarettir. Örnek olarak SATIS, BORDRO, MUSTERI gibi veri tabanı isimleri olabilir.

9.2.1 Tablo Alanı

Tablo alanı (tablespace), tabloların üzerinde depolanacağı dosya ismidir, yâni `/usr/dizin1/dizin2/cizelge_alan_1.dbf` gibi bir gerçek Unix dosyasından bahsediyoruz. Oracle veri tabanının, gerçek dünya ile (işletim sistemi) buluştuğu yere tablo alanıdır.Tablo ile tablo alanlarının bağlantısı, alan yaratılırken sâdece bir kere yapılır. Ondan sonra ne zaman bu tabloya erişseniz, önceden tanımlanmış olan dosyadan oku/yaz otomatik olarak yapılacaktır.

Tablo alanı yaratmak için, şöyle bir komut işletebiliriz.

```
CREATE TABLESPACE cizelge_alan_1 DATAFILE
'/usr/local/dizin1/oracle/cizelge_alan_1.dbf' SIZE 200M;
```

9.2.2 Şema

Şema=kullanıcı gibi bir tanım yapabiliriz, fakat Oracle dünyasında şema biraz daha güçlü bir kavramdır. Eğer daha basit veri tabanlarına alışıksak, herhalde her kullanıcının her tabloyu görmesine alışığızdır. Fakat Oracle için tabloların erişilip erişilmeyeceği, tabloların nerede tutulacağı (yâni sahiplenme mekanizması) şema bazında idare edilmektedir. Bir veri tabanına (`SID=ORNEK1`) bağlandığınız zaman, bu tabanda bazı tabloları göremeyebilirsiniz. Görmek için, belli bir kullanıcı ve şema kullanarak bağlanmanız gerekecektir. Aynı isimdeki iki tabloyu değişik şemalarda yaratabiliriz; Oracle bundan yakınmaz. Yâni SID'den sonra Oracle'ı paylaştırmanın/bölmenin ikinci bir yolu şemadır.

Oracle idarecileri az miktarda SID ve paylaştırmak için çok miktarda şema yaratmayı tercih ederler. Şema yaratmak için kullanıcı yaratmamız yeterlidir.

```
CREATE USER hasan IDENTIFIED BY hasanslx DEFAULT TABLESPACE alan1;
```

Bu konu hakkında bazı notlar şunlardır:

- Kullanıcı MEHMET olarak SID'e bağlandıysanız, ve CREATE TABLE komutunu işletip bir tablo yarattıysanız, bu tablo MEHMET şemasına ait olacaktır. Bu tabloyu başkaları göremez. Görmesi için özel izin 'vermeniz' gerekir.

- Eğer MEHMET kullanıcısı olarak bir tablo yarattıysanız ve detay belirtmediyseniz, bu tablo DEFAULT TABLESPACE diye yukarıda belirttiğimiz yokluk değeri (default) `alan1` altında yaratılır.

9.2.3 Görüntü

Görüntüler, önceden depolanmış ve sorgulanabilen SQL kodlarıdır. Bir görüntü, güvenlik ötürü bazı çizgeleri göstermeden, bu çizgelerin verisinin bir kısmını göstermek için kullanılabilir. Mesela halkla ilişkiler bölümü için, isim, soyad ve adres bir görüntü içinden gösterilebilir, öteki bilgiler saklanabilir. Ya da, dağınık

veri tabanlarından toparlanacak bilgiler, bir görüntü ile önceden kodlanır ise, artık birden fazla veri tabanına bağlanmak yerine, tek bir görüntü üzerinden veri alınabilir. Özet olarak, normalde işleyen her SQL kodu, görüntü yaramak için kullanılabilir.

```
create or replace view goruntu_1 as
select isim, soyad
from MUSTERI;
```

9.2.4 Dizi (Sequence)

Tekil sayı yaratmak için kullanılan Oracle nesnesine dizi adı verilir. Dizi yaratmak için başlangıç değeri, artış miktarı ve bitiş sayısı vermek yeterlidir. Yani, 1..n arası her seferinde 1 kadar artacak bir dizi yaratmak mümkün. Bu sayılar genelde kimlik no gibi özgün olması gereken değerler için kullanılır. Her seferinde kod içinde, 'önceki değer + 1' demek yerine, SQL kullanarak her diziden yeni sayı isteyişimizde, güncel dizi artış değeri kadar otomatik olarak arttırılır ve alınan sayı SQL sonucu olarak verilir.

Bir dizi yarattığınız zaman, `NEXTVAL` ve `CURRVAL` sanal kolonlarını kullanırız. Mesela `UYE_DIZISI` adlı bir dizi yarattı isek, `SELECT UYE_DIZI.NEXTVAL FROM DUAL` diyerek o anki değeri alabiliriz. Bu komut sonucunda ayrıca dizi değeri 1 arttırılır. `SELECT UYE_DIZI.CURRVAL FROM DUAL` kullanımı o anki değeri verecektir. Fakat diziyi arttırmaz.

```
CREATE SEQUENCE DIZI_ISMI
INCREMENT BY 1 -- artış
START WITH 100; -- başlangıç
```

9.2.5 Tetik

Tetikler (trigger), depolanmış PL/SQL kodlarıdır. Önceden öngörülen belli bazı tablolara, gene önceden öngörülen şekilde bir erişim olduğunda tetikler ateşlenirler, ve kodlanan işlemi yerine getirirler. Tetikler, satır ekleme, silme, güncelleme ya da bütün bu temel işlemlerin değişik guruplamaları için kodlanabilirler. Tetiklerin en çok kullanıldığı alan, veri bütünlüğü için kısıtlamalar koymaktır. Bu tür kısıtlamalar Oracle terimleri ile, veri bütünlük kontrolleri (integrity constraints) olarak bilinir, ve satırlar arasında bazı kontroller getirebilir. Fakat bu tür satır bazlı kontrollerin yetmediği hallerde, tetik kodları işe yarayabilir. Çünkü tetik kodları içine PL/SQL ile kodlayabildiğiniz her türlü kod koyabilirsiniz.

```
create to replace trigger musteri_tetik_1 INSTEAD OF
insert on musteri for each row
begin
insert into maas values
('1', '2'); -- buraya daha değişken kod koyabilirsiniz
```

```
end;
/
```

9.2.6 Dizi ve Tetik

Bazı veri şemalarında ID kolonu atanan bir değer değil, üretilmesi gereken bir sayıdır. Bu sayı, en basit hâliyle 1'den başlayarak her yeni veri satırı için birer farkla artması gereken bir sayıdır.

O zaman her satırı yazarken bu ID'nin üretilmesi gerekmektedir. Bu üretimi, ya uygulama içinde yapacağız, ya da veri tabanın otomatik olarak yapmasını sağlayacağız.

Otomatik attırım bazı veri tabanlarında bir kolon tipi olarak bile karşınıza çıkabilir. Oracle'da bu işi yapmak için bir sequence ve bir trigger kullanmamız gerekiyor. Oracle, bu tekil sayının üretimini ve tabloya atanması işini birbirinden ayırmıştır. Bu da sanıyorum isabet olmuştur, auto-increment kolon tipi hakikaten çok basitleyici bir çözümdür.

Oracle'da otomatik ID üretimi için iki şey gerekir; Bir sequence, bir de trigger. Meselâ, şöyle bir tablomuz olduğunu düşünelim.

```
create table test (
id number,
veri varchar2(20),
...
);
```

Bu tablo için bir sequence ve bir trigger yaratalım.

```
create sequence test_seq
start with 1
increment by 1
nomaxvalue
;

create trigger test_trigger
before insert on test
for each row
begin
select test_seq.nextval into :new.id from dual;
end;


INSERT into test (veri) values ('blablabla');
```

gibi bir `INSERT` kullandığımızda, hiç ID'ye dokunmamıza gerek kalmadan, bir sonraki ID sayısı hesaplanacak ve kolona koyulacaktır.

9.2.7 Veri Taban Köprüsü

Köprüler (database link) nesne (tablo, kolon, vs) bazında değil, tüm veri tabanı bazında bağlantı kurar. Yani, uzakta olan bir veri tabanına, sürekli olarak ve uzak adresini kullanarak erişmek istemiyorsanız, bir köprü, yani kestirme yaratarak o veri tabanına sanki yerel bir tabanmış gibi erişebilirsiniz.

```
CREATE PUBLIC DATABASE LINK baglanti_1
CONNECT TO kullanici
IDENTIFIED BY sifre
USING 'UZAK_VERI_TABAN_SID_DEGERI';
```

9.2.8 Eşanlam (Synonym)

Eşanlamlar, bir veri tabanındaki tablolarden diğer tablolara kestirme işaret olarak görülebilir. Bir eşanlam yaratmak için eşanlam ismi, ve o eşanlamın yerini tuttuğu Oracle nesnesinin ismi gereklidir. SQLPlus kullanarak bu eşanlama eriştiğimizde, Oracle arka planda eşanlamın yerini tuttuğu nesneyi bulur, ve işlemi o öteki nesne üzerinde yapar.

İki türlü eşanlam vardır. Umumi (public) ve özel (private). Özel eşanlamlar içinde yaratıldıkları şemaya özel olurlar, o şemaya ait kalırlar ve başka kullanıcılar tarafından erişilemez, hatta görülmezler bile. Eğer eşanlamlar umumi ise, bütün şematikler tarafından kullanılabilirler.

Oracle'ın bir nesneye erişmek için hangi tür bir algoritma izlediğine gelelim. Eğer şöyle bir kod işletildi ise,

```
SELECT * FROM FROM MAAS
```

Oracle MAAS nesnesini bulmak için şunları yapar.

- MAAS adlı bir tablo ya da görüntü var mı?
- Bu ikisi yoksa, Oracle MAAS adında özel bir eşanlam arar.
- Var ise, özel eşanlamın gösterdiği nesne kullanılır.
- Yok ise, MAAS adlı umumi eşanlam aranır.
- Hiçbiri yok ise, Oracle ORA-00942 hata mesajını verecektir.

```
create synonym MUSTERI for BIZIM_UZUN_MUSTERI_ISMI;
```

9.2.9 İndeksler

Bir tablodan veri almak için, kullandığımız `SELECT` komutunun `WHERE` kısmı içinde filtre şartları belirtmemiz gerekir. Bu şartlar, veri tabanı tablosu taranırken gereken satırları diğerlerinden ayırıp onları çekip çıkarabilmemiz için kullanılır. Arama şartları tanımlandıktan sonra, veri tabanı kapalı perdeler arkasında tarama işlemini gerçekleştirmek için şunları yapar: Tabloyu tararken, ilk satırdan başlayıp teker teker sonuncu satıra doğru tüm satırlara bakar, ve filtre şartlarının uyup uymadığını kontrol eder.

Fakat tüm satırlara bakılan türden bir arama, özellikle büyük tablolar için çok uzun zaman alacaktır. Eğer tüm satırlara bakılan türden bir tarama yapılmasını istemiyorsak, veri tabanına arama yaparken bir şekilde *yardım etmenin* yolunu bulmalıyız.

İndeksler bu türden yardım yöntemidirler. İndeksler, kendileri veri içermezler, sadece gerçek veriyi içeren tablonun belli satırlarına *işaretler* taşırlar. Çok hızlı aranabilen türden veri yapılarıdırlar. Bu işaretler, belli anahtar kolonlarına göre (indeksle) yapılmıştır. İndeksler bu açıdan bir fihriste benzerler.

Bir indeksin kullanıma konması şöyle olur: Eğer `SELECT` filtremiz içinde kullanılan kolonlar üzerinde bir indeks mevcutsa, veri tabanı filtreyi önce indeksden tarar, buradan gelen satır işaretini kullanarak ta gerçek tabloya giderek aranan gerçek satırı döndürür.

İndeksler gerçek tabloyu indeksledikleri için, bu tabloda olan her değişim indeksleri etkiler. Meselâ tabloya yeni bir satır eklenirse, indeks bu ekleme işlemini yansıtacak şekilde güncellenir. Bu sebeple gereğinden fazla indeks eklenmesi bir tabloya yapılan `INSERT` işlemlerini yavaşlatacaktır. Tam kararında kurulan indeksler, hem `SELECT` sorgularımızı hızlandırır, hem de `INSERT`'lerimiz üzerinde fazla bir etkide bulunmazlar.

Şu anda piyasada olan her profosyonel veri taban ürünü, indeks teknolojisini desteklemektedir. Bu bölümün geri kalanında PostgreSQL, Oracle ve MySQL tabanlarında indeks eklemenin yollarını, ve en önemlisi, herhangi bir sorgunun üzerinde analiz yaparak indeks kullanıp kullanmadığını görebilmek için gereken komutları öğreneceğiz.

Elimizde `car` ve `garage` adlı iki tablo olduğunu düşünün. Bu tablolar arasında bire çok türünden bir ilişki olsun ve bir `car` birçok garajın altında olabilsin. Bu ilişkiyi fiziksel anlamda, `car` tablosu üzerinde bir `garage_id` koyarak gerçekleştiriyoruz.

PostgreSQL

Bir sorgunun indeks kullanıp kullanmadığını analiz etmek için, o sorguyu `psql` ile PostgreSQL komut satırına girip, `EXPLAIN` komutu eşliğinde işletmemiz gerekmektedir.

```
explain
select * from car where license_plate = '34 THY 334';
```

Eğer hiçbir indeks tanımlamamışsak, şöyle bir cevap geriye gelecektir.

```
QUERY PLAN
-----------------------------------------------------
Seq Scan on car (cost=0.00..330.00 rows=44 width=170)
Filter: ((license_plate)::text = '34 THY 334'::text)
```

Bu çıktıda ilk dikkatimizi çeken kelimeler, `Seq Scan` kelimeleri olmalıdır. Bu kelimeler Sequential Scan (sırayla tarama, teker teker bakma) kelimesinin bir kısaltmasıdır, ve `SELECT` komutunun teker teker her satıra baktığı anlamına gelir. Bu performans açısından istediğimiz bir durum değildir. Biz indekslerin kullanılmasını istiyoruz. Ve gördük ki, hiçbir indeks kullanılmamıstır. Çünkü daha indeks tanımlamadık! Aynı şekilde,

```
explain
select c.*
from
car c, garage g
where
c.garage_id = g.id and
g.id = 2 and
c.available = 't';
```

sorgusunun analizi de şu cevabı getirecektir.

```
QUERY PLAN
------------------------------------------------------------------------
Nested Loop (cost=18.50..372.26 rows=88 width=170)
-> Seq Scan on car c (cost=0.00..352.00 rows=22 width=170)
Filter: ((available = true) AND (2::numeric = garage_id))
-> Materialize (cost=18.50..18.54 rows=4 width=23)
-> Seq Scan on garage g (cost=0.00..18.50 rows=4 width=23)
Filter: (id = 2::numeric)
```

Çok kötü. Bu sorguda da, hem iki tablo birleştirimi (join) hem de `available` kolonu üzerinde yaptığımız filtreleme üzerinde `seq scan` taraması yapıldığını görüyoruz. Satır sayısı oldukça fazla olan `car` ve `garage` tabloları kullanıyor olsaydık, bu son sorgunun da performanı çok kötü olacaktır.

O zaman indeksleri ekleyelim. PostgreSQL'da indeks şöyle eklenir:

```
CREATE UNIQUE INDEX car_license_plate_idx on car (license_plate);
CREATE UNIQUE INDEX garage_id_idx on garage (id);
CREATE INDEX car_available_idx on car (available);
CREATE INDEX car_garage_idx on car (garage_id);
```

`car` tablosunda `license_plate` kolonu üzerinden işleyen bir indeks ekledik. Aynı şekilde `garage.id` kolonuna, `car.available` kolonuna ve `car.garage_id` kolonuna indeksler ekledik. İndeksler eklendikten sonra, sırasıyla her iki sorguda yapılan `EXPLAIN` komutu, şu sonucu döndürecektir.

```
QUERY PLAN
------------------------------------------------------------------------
```

```
Index Scan using car_license_plate_idx on car (cost=0.00..6.00 rows=1
width=170)
Index Cond: ((license_plate)::text = '701790-171'::text)

QUERY PLAN
------------------------------------------------------------------------------
Nested Loop (cost=0.00..15.03 rows=2 width=170)
->  Seq Scan on garage g (cost=0.00..1.05 rows=1 width=23)
Filter: (id = 2::numeric)
->  Index Scan using car_garage_idx on car c (cost=0.00..13.96 rows=2
width=170)
Index Cond: (2::numeric = garage_id)
Filter: (available = true)
```

Sonuca bakarak görüyoruz ki, birinci sorguda `Seq Scan` ibaresinde kurtulduk. Artık bu sorguya erişim, `Index Scan` ile olmaktadır, yâni verdiğimiz sorgu `car` tablosuna yarattığımız indeks üzerinden erişilmektedir. Bir hız ilerlemesi kaydettik. İkinci sorguda, `available` kolonu üzerindeki sorgudaki filtre ibaresi de indeks kullanmaya başlamıştır.

Fakat PostgreSQL sorgu hızlandırıcısı (optimizer), `garage.id` üzerindeki indeksi devreye sokmamıştır, çünkü `garage` tablosunda çok az veri vardır (örnek verimiz böyle idi) ve bu şekilde az veri taşıyan tablolarda indeks kullanıp kullanmamak bir hız farkı getirmeyecektir.

Oracle

Oracle üzerinde indeks yaratmak için `CREATE INDEX`, sorguların indeks kullanımını analiz etmek için `EXPLAIN PLAN` komutları kullanılır. `EXPLAIN PLAN`, komutu işlettiğini şemada işleyebilmek için `PLAN_TABLE` adlı bir sistem tablosunu kullanır. Oracle 10g üzerinde bu tablo her veri tabanı içinde kurulmuş olur, ama eğer kullandığınız versiyonda mevcut değilse, tabloyu şu komut ile yaratabilirsiniz (`ORACLE_HOME`, Oracle paketinin kurulmuş olduğu dizini temsil eder).

```
\$ sqlplus user/pass@SID
@ORACLE_HOME/product/<versiyon>/<Db>/rdbms/admin/utlxplan.sql
```

`utlxplan.sql` dosyasında `plan_table` ve ilgili tüm tabloların yaratılması için gereken işlemler yapılmaktadır.

Analiz için şu komutu kullanabilirsiniz.

```
explain plan SET STATEMENT_ID='SORGU_1' FOR
select * from car where license_plate = '34 THY 334';

SELECT LPAD(' ',2*DEPTH) | | OPERATION | | ' ' | | OPTIONS | | ' '
| | OBJECT_NAME | | ' ' | | DECODE(ID, 0, 'COST= '| | POSITION)
"QUERY PLAN"
FROM PLAN_TABLE START WITH ID=0 AND STATEMENT_ID='SORGU_1'
```

```
CONNECT BY PRIOR ID=PARENT_ID;
```

Sonuç şöyle gelir:

```
QUERY PLAN
---------------------------------

SELECT STATEMENT COST= 3
TABLE ACCESS FULL CAR
```

Bu işletim planına bakarsak, `TABLE ACCESS FULL` kelimelerini hemen gözümüze çarpar. Bu kelimeler bize işletilen sorgunun hiçbir indeks kullanmadığını göstermektedir, ve Oracle tablodaki satırlara teker teker ve sırayla erişmeye çalışmaktadır. Aynı şekilde diğer sorgumuza bakarsak;

```
explain plan SET STATEMENT_ID='SORGU_1' FOR
select c.*
from
car c, garage g
where
c.garage_id = g.id and
g.id = 2 and
c.available = 't';

SELECT LPAD(' ',2*DEPTH) | | OPERATION | | ' ' | | OPTIONS | | ' '
| | OBJECT_NAME | | ' ' | | DECODE(ID, 0, 'COST= '| | POSITION)
"QUERY PLAN"
FROM PLAN_TABLE START WITH ID=0 AND STATEMENT_ID='SORGU_1'
CONNECT BY PRIOR ID=PARENT_ID;
```

şu sonucu görürüz

```
QUERY PLAN
---------------------------------

SELECT STATEMENT COST= 6
MERGE JOIN CARTESIAN
TABLE ACCESS FULL CAR
BUFFER SORT
TABLE ACCESS FULL GARAGE
```

Bu analiz sonucunda da bol bol `TABLE ACCESS FULL` görmekteyiz. Demek ki gerekli yerlere indeksler eklememiz gerekiyor.

```
CREATE UNIQUE INDEX car_license_plate_idx on car (license_plate);
CREATE UNIQUE INDEX garage_id_idx on garage (id);
CREATE INDEX car_available_idx on car (available);
CREATE INDEX car_garage_idx on car (garage_id);
```

Şimdi `EXPLAIN PLAN`'i tekrar işletirsek, şu sonuçları göreceğiz.

```
QUERY PLAN
----------------------------------------------

SELECT STATEMENT COST= 1
TABLE ACCESS BY INDEX ROWID CAR
INDEX UNIQUE SCAN CAR_LICENSE_PLATE_IDX

QUERY PLAN
-----------------------------------------------

SELECT STATEMENT COST= 0
NESTED LOOPS
INDEX UNIQUE SCAN GARAGE_ID_IDX
TABLE ACCESS BY INDEX ROWID CAR
BITMAP CONVERSION TO ROWIDS
BITMAP AND
BITMAP CONVERSION FROM ROWIDS
INDEX RANGE SCAN CAR_AVAILABLE_IDX
BITMAP CONVERSION FROM ROWIDS
INDEX RANGE SCAN CAR_GARAGE_IDX
```

Bu sonuçlar çok daha iyidir. Birinci sorgunun analizi `TABLE INDEX BY ROWID CAR` kelimesini taşıyor, bu demektir ki tabloya erişim artık bir indeks üzerinden gerçekleşmektedir. Kullanılan indeks `CAR_LICENSE_PLATE_IDX` isimli indekstir, bu da hemen altındaki satırda belirtilmiştir.

İkinci sorgunun sonucu biraz daha çetrefildir, fakat burada da görüldüğü gibi artık `TABLE ACCESS FULL` sözü kaybolmuştur, onun yerine indekslerin kullanılmaya başlandığı `INDEX UNIQUE SCAN`, `INDEX RANGE SCAN` ve `TABLE ACCESS BY ROWID` kelimelerinden belli olmaktadır. `INDEX UNIQUE` ile `INDEX RANGE` arasındaki fark, ilkinin tekil bir satıra işaret edebilmesi, ikincisinin tekrar eden anahtar değerlere işaret eden türden bir indeks olmasıdır.

Sonuç olarak Oracle dünyasında sorgu optimizasyonu ile amacımız, `EXPLAIN PLAN`'den gelen sonuçlarda görülebilecek `TABLE ACCESS FULL` ibaresinden kurtulmaktır.

MySQL

MySQL veri tabanında analiz için `explain` komutu kullanılır. Aynen PostgreSQL örneğinde olduğu gibi, analiz etmek istediğiniz sorgunun önüne `explain` ibaresini eklerseniz, MySQL analiz sonucunu gösterecektir. Örnek olarak `car` ve `garage` üzerindeki yapılan iki sorguyu analiz edelim:

```
explain select * from car where license_plate = '32 TF 22';
```

```
+-------+------+---------------+------+---------+------+------+-------------+
| table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+-------+------+---------------+------+---------+------+------+-------------+
| car   | ALL  | NULL          | NULL |    NULL | NULL |    5 | Using where |
+-------+------+---------------+------+---------+------+------+-------------+
```

```
explain
select c.*
from
car c, garage g
where
c.garage_id = g.id and
g.id = 2 and
c.available = 't';
```

```
+-------+------+---------------+------+---------+------+------+-------------+
| table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+-------+------+---------------+------+---------+------+------+-------------+
| c     | ALL  | NULL          | NULL |    NULL | NULL |   47 | Using where |
| g     | ALL  | NULL          | NULL |    NULL | NULL |    4 | Using where |
+-------+------+---------------+------+---------+------+------+-------------+
```

`explain` komutundan gelen üstteki sonuçlar, her tablo için bir satır olmak üzere listelenecektir. Her tabloya yapılan erişimin yöntemini, `type` altında bulabilirsiniz. MySQL dünyasında, sorgu analizinde bir tablo için `type` için `ALL` görmek *iyi değildir*. `type` için gelmesi muhtemel değerleri en iyiden en kötüye olacak şekilde bir sıralamasını aşağıda görüyoruz.

1. `system`
2. `const`
3. `eq_ref`
4. `ref`
5. `ref_or_null`
6. `index_merge`
7. `ununique_subquery`
8. `range`
9. `index`
10. `ALL`

`ALL` değeri en alttadır, yâni yukarıdaki sorgularımızın eriştiği tablolara indeks eklemezsek, yukarıdaki sorgular en kötü performans ile işletiliyor olacaktırlar. O zaman bu durumu düzeltmek için indekslerimizi ekleyelim:

```
CREATE UNIQUE INDEX garage_id_idx on garage (id);
CREATE INDEX car_available_idx on car (available);
CREATE INDEX car_garage_idx on car (garage_id);
CREATE UNIQUE INDEX car_license_plate_idx on car (license_plate);
alter table garage add PRIMARY KEY(id);
alter table car add PRIMARY KEY(license_plate);
alter table car add FOREIGN KEY (garage_id) REFERENCES garage(id);
```

Tekrar `explain` eşliğinde iki sorguyu işletelim ve sonuçları görelim.

```
+-------+------+----------------------+---------+---------+-------+------+-------+
| table | type | possible_keys        | key     | key_len | ref   | rows | Extra |
+-------+------+----------------------+---------+---------+-------+------+-------+
| car   | const | PRIMARY,license_plate | PRIMARY |      30 | const |    1 |       |
+-------+------+----------------------+---------+---------+-------+------+-------+

+-------+------+--------------+-----------+---------+-------+------+-------------+
| table | type | possible_keys | key       | key_len | ref   | rows | Extra       |
+-------+------+--------------+-----------+---------+-------+------+-------------+
| g     | const | PRIMARY,id    | PRIMARY   |      31 | const |    1 |             |
| c     | ref   | garage_id     | garage_id |      32 | const |   10 | Using where |
+-------+------+--------------+-----------+---------+-------+------+-------------+
```

Sonuçlar daha iyileşti. Birinci sorguda `car` tablosuna olan erişim, sorgu iyilik derecesinde 2. seviyede olan `const` seviyesine yükseldi. İkinci sorguda ise `garage` (`g`) ve `car` (`c`) tablolarının erişimi ise `garage` için `const` ve `car` için iyilik derecesinde 4. seviyede olan `ref` erişimine yükseldi.

9.2.10 Oracle SQL*Loader

Metin bazlı bilgileri Oracle veri tabanına yüklemek istiyorsanız, bunun en rahat yolu SQL*Loader adlı programı kullanmaktır. SQL*Loader, kontrol dosyası denilen bir ayartanım dosyası eşliğinde, virgül ayırımlı, boşluk ayırımlı, tab ayırımlı, ya da sabit uzunluktaki kolonlar içeren metinlerin hepsini veri tabanına yükleyebilir. Bunu bir örnek üzerinde görelim: Önce üzerinde yükleme yapacağımız tabloyu veri tabanında yaratalım.

```
create table musteri (

  cust_nbr       number(7)       not null,
  cust_name      varchar2(100)   not null,
  cust_addr1     varchar2(50),
  cust_addr2     varchar2(50),
  cust_city      varchar2(30),
  cust_state     varchar2(2),
  cust_zip       varchar2(10),
  cust_phone     varchar2(20),
  cust_birthday  date)
/

create table hesap (

  cust_nbr       number(7)       not null,
  acct_nbr       number(10)      not null,
  acct_name      varchar2(40)    not null)
/
```

Üstteki komutları kullanarak tabloyu yarattıktan sonra, örnek metin dosyalarını `SqlLodader` projesinden alıp kullanabilirsiniz.

Sabit Uzunluklu Kayıt Yüklemek

Şimdi, SQL*loader'ın çalışması için bir kontrol dosyası lazım. Müşteri verisi için yazılmıs aşağıdaki kontrol dosyası örneğini, load1.ctl adında diskinize yazın. Bu örnek, sabit uzunluklu bir veri dosyasını yüklemek için verilmiştir.

```
LOAD DATA
INFILE 'cust.dat'
INTO TABLE musteri
   (cust_nbr     POSITION(01:07)  INTEGER EXTERNAL,
    cust_name    POSITION(08:27)  CHAR,
    cust_addr1   POSITION(28:47)  CHAR,
    cust_city    POSITION(48:67)  CHAR,
    cust_state   POSITION(68:69)  CHAR,
    cust_zip     POSITION(70:79)  CHAR,
    cust_phone   POSITION(80:91)  CHAR,
    cust_birthday POSITION(100:108) DATE "DD-MON-YY" NULLIF
                                    cust_birthday=BLANKS)
```

Artık müşteriyi veri tabanına yüklemeye hazırız. Aşağıdaki komut ile bunu yapabiliriz.

```
\$ sqlldr kullanici/sifre control=load1.ctl log=load1.log bad=load1.bad
   discard=load1.dis
```

Bilgisayardan gelen yanıt:

```
SQL*Loader: Release 8.0.3.0.0 - Production on Wed Mar 10 8:10:23 1999
(c) Copyright 1997 Oracle Corporation. All rights reserved.
Commit point reached - logical record count 23
```

Çıktıya (`sqlloader_musteri_cikti.txt`) bakarak SQL*Loader'ın başarıya ulaşıp ulaşmadığını anlayabiliriz.

Bu sonuca göre, olanlar şunlardır: 2'inci kayıt veri tabanına kabul edilmedi, çünkü tarih verisi Oracle tarafından geçersiz bulundu. Bu geçersiz kayıt, kötü (bad) dosyasına yazıldı. Kötü dosyasının ismini komut satırından `bad=load1.bad` ibaresini kullanarak verilmişti.

Değişken Uzunluklu Kayıt Yüklemek

Şimdi hesap verisini virgül ayrıklı bir veri dosyasından veri tabanına yükleyelim. Bu şekilde bir yükleme için lazım olan kontrol dosyası (load2.ctl) aşağıdadır.

```
LOAD DATA

INFILE 'acct.dat'

INTO TABLE hesap

FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(cust_nbr, acct_nbr, acct_name)
```

Bu kontrol dosyasını da komut satırından işletelim.

```
\$ sqlldr kullanici/sifre control=load2.ctl log=load2.log
  bad=load2.bad discard=load2.dis
```

Bilgisayardan gelen yanıt:

```
SQL*Loader: Release 8.0.3.0.0 - Production on Wed Mar 10 8:10:23 1999
(c) Copyright 1997 Oracle Corporation. All rights reserved.
Commit point reached - logical record count 12
```

Çıktıya (`sqlloader_hesap_cikti.txt`) bakarak SQL*Loader'ın başarıya ulaşıp ulaşmadığını anlayabiliriz.

Bu çıktıya bakarak, şu sonuca varabiliriz: Bütün kayıtlar hatasız bir şekilde yüklendi. Bu örnek veri dosyası değişken uzunluklu veri içerdiği için, metin dosyasının içinde bir ayraç gerektiğini hatırlatmamız lazım. Bu örnek içinde ayraç olarak virgül işareti kullanıldık.

Kayıt Eklemek

INTO TABLE hesap şeklindeki sözdizimi, boş olan bir tablo farzediyordu. Eğer içinde zaten veri mevcut olan bir tabloya veri eklemek istiyorsanız, INTO TABLE ifadesinin başına APPEND eklemeniz gerekiyor. Yani,

```
LOAD DATA

INFILE 'acct.dat'

APPEND INTO TABLE hesap

FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    (cust_nbr, acct_nbr, acct_name)
```

Özet

SQL*Loader için gereken ayarları burada özetleyelim.

- Yüklenecek veri dosyasının ismi, kontrol dosyasında INFILE ibaresinden sonra verilir.
- Hedef tablosunun ismi, kontrol dosyasında INTO TABLE ibaresinden sonra gelir.
- Değişken uzunluklu veri dosyaları için ayraç karakteri, kontrol dosyasında FIELDS TERMINATED BY ibaresinden sonra verilir.
- Sabit uzunluklu veri dosyaları için başlangıç ve bitiş kolonları, kontrol dosyasında POSITION(BAŞLANGIÇ:BİTİŞ) olarak tanımlanır.

9.3 Transaction

Veri tabanını üzerinde işletilen her SQL komutu bir transaction altında yapılır. Transaction, bir ve ya daha fazla SQL işlemini kapsayabilir, zâten çoğunlukla kullanılma amacı budur: Hep beraber etki etmesi gereken değişiklikler için birleştirici bir ünite sağlamak. 2.2.5 bölümündeki örneği tekrarlamak gerekirse, bir banka uygulamasında bir müşterinin iki banka hesabı olsa, ve bu müşteri bir hesaptan ötekine para transfer etmek istese, transaction sayesinde bir hesaptan eksiltmek için yapılan `UPDATE` SQL işlemi, para eklemek için yapılan ekleme `UPDATE` SQL işlemi ile aynı anda etki edebilecektir. Bu iki işlemi aynı transaction altına koyabiliriz.

Transaction kavramı, bir ilişkisel tabanda veri doğruluğunu, veri taban bağlantısını, tablolar üzerine konan kilitleri ve verinin en son görüntüsünü biraraya getiren ve hepsi ile yakın alâkası olan merkezi bir kavramdır. Bir tabanda gerçekleştirdiğimiz her SQL sorgusu, bu sebeple, bir transaction altında yapılacaktır; Eğer bundan haberimiz yoksa, yâni bilerek bir transaction başlatıp durdurmuyorsak kapalı kapılar altında bizim için bir tane muhakkak başlatıp bitiriliyordur.

SQL kullanırken bir transaction'ın başlaması, SQL sorgusunun tabana gelmesiyle ya da transaction başlatma emrinin verilmesiyle olur. Eğer o anda işleyen başka bir transaction yok ise, taban bir tane otomatik olarak başlatır. Transaction'ın bitmesi `commit` ya da `rollback` komutlarından bir tanesi ile olacaktır. `Commit`, o transaction altında yapılan değişikliklerin *kalıcı* olmasına karar verildiği anlamına gelir.

Bir `commit` gelmeden önce diğer transaction'lar (yâni diğer veri taban bağlantılarında SQL işletmekte olanlar) bizim son değişikliklerimizden tamamen habersiz olacaklardır. Bu diğer bağlantılar, sadece verinin bizim transaction *başladığı andaki son* hâlini görürler. Ama bizim bağlantımız içinde içinde biz kendi güncelleme, silme, ekleme işlemlerimizin sonucunu görebiliriz.

Eğer bir transaction altındayken (ve bir takım işlemlerden sonra) bir `rollback` komutu gelirse, bu, yapılan veri değiştirme işlemlerinden *vazgeçildiği* anlamına gelecektir. O zaman veri tabanı, yapılan hiçbir değişikliği kalıcı yapmaz, ve öteki bağlantılar ve transaction'lar eski veriyi görmeye devam ederler.

> **Not**: Günümüzdeki popüler kullanıma göre iki değişik transaction, her zaman iki değişik bağlantı (db connection) anlamına gelir. Aynı bağlantı içinden içiçe geçmiş (nested) transaction başlatmak gereksiz bir karmaşıklık yaratmaktadır, ve bu sebeple konumuz dışındadır.

Eğer bir transaction'ı başlatmış süreç (process) çökerse, o veri tabanı bağlantısı, ve o bağlantıda başlatılmış olan transaction `rollback` edilecektir. Bu, kurumsal bir uygulamanın veri bütünlüğü için çok önemlidir; `Commit` sinyali gelen kadar hiçbir değişikliğin kalıcı olmaması hayati önem taşır. Çöken bir sürecin o ana kadar veri tabanını ne durumda bıraktığını bilemeyiz, belki yapılacak SQL

işlemlerinin sadece yarısı gerçekleştirilebilmiştir. Bu durumdaki bir transaction'ı commit etmek, intihar demek olurdu! Meselâ müşteri örneğine dönersek, belki müşterinin hesabında para eksiltilmiş, ama öteki hesabına para eklenemeden süreç çökmüştür. Bu durumdaki bir transaction commit edilecek olsa, müşteri para kaybetmiş olurdu! Bu tolere edilecek bir davranış değildir. Bu sebeple veri taban ürünleri, çöken bir sürecin bağlantısını ve onun içindeki olduğu transaction'ı otomatik olarak `rollback` ederler.

Ayrıca tüm modern veri taban ürünlerinde transaction ve satır seviye kilit (row level lock) yakınen (birebir) bağlantılıdır. Şöyle ki; Eğer bir satır üzerinde `UPDATE` komutu ile güncelleme yapılmışsa ve o transaction `commit` edilmeden ikinci bir `UPDATE` aynı satırı güncellemeye kalkarsa, ikinci `UPDATE` birincinin bitmesini *bekleyecektir*. Bitmek, daha önce belirttiğimiz gibi, ya `commit` ya da `rollback` ile olması mümkün bir işlemdir.

İkinci `UPDATE` birinci bittiği anda işleme konur ve o da bittiğinde (`commit` ya da `rollback` ile) geri gelir.

9.4 Beklemeden Kitlemek

Eğer bir satır üzerinde kilit olmasını bekliyor, ve o kilit üzerinde takılmak istemiyorsak, Oracle'a o satırı *beklemeden* kitlemeyi denettirebiliriz. Bunun için;

```
SELECT ... WHERE .. FOR UPDATE NOWAIT;
```

komutu kullanılır. Normâlde `SELECT FOR UPDATE` komutunun kitleme davranışı, `UPDATE` komutununki ile aynıdır. Ayrıca `NOWAIT` eki, Oracle'a eğer bir kilit mevcutsa beklemeden dönmesini söyleren ek bir davranış sağlar.

Kilit beklemeden dönme yöntemi toptan (batch) işlem yapan programımız için faydalı olabilir. Meselâ, her yeni `GARAGE`'ın altındaki her `CAR` nesnelerini bir şekilde işlemden geçirmemiz istense ve uygulamamızın çok süreçli (multi process) bir hâlde çalışması istense, birden fazla sürecin aynı tablo üzerine akın etmesi bir problem doğurabilirdi. Paylaşım mekanizması olarak `GARAGE` tablosu üzerinde `SELECT FOR UPDATE NOWAIT` ile alınan bir kilidi kullanırsak, ikinci, üçüncü, vs. gelen süreçler, beklemeden dönecekler, ve başka bir `GARAGE` satırını kilitlemeye (kapmaya) uğraşacaklardır.

Böylece bir `GARAGE` satırının işlenmekte olduğunu, üzerinde kilit olmasından anlayabiliriz. Alternatif olarak `GARAGE`'a `STATUS` adında bir kolon eklesek (üzerinde 'işleniyor','işlenmiyor' gibi bir durum kodu kullanılarak), bu işimize yaramazdı, çünkü bir `GARAGE`'ı işlemekte olan sürecin çökme durumunda, `STATUS` kolonu eski hâline getirilemeyeceği için, sürekli işleniyor modunda gibi gözükecekti; Halbuki kilit kullanıldığı şartlarda, sürecin çöküşü satır kilidinin otomatik olarak bırakılması anlamına gelecek, böylece yeni bir sürecin aynı satırı kitleyerek işleme kalınan yerden devam etmesine izin verilecebilecektir. Altta bahsedilen toptan işlemin taklit kodunu (pseudocode) görüyoruz.

```
list = run_query(‘‘SELECT * from GARAGE’’);
```

```
for garage in list
begin
   result = ‘‘SELECT * FROM GARAGE ’’ +
            ‘‘where ID = garage.id FOR UPDATE NOWAIT’’;
   if (result.size == 0)
   begin
      continue;
   else
      process_cars(garage);
   end;
end
```

9.5 Kurmak

Bu bölümde MySQL, PostgreSQL ve Oracle'ı SuSe Linux üzerinde nasıl kuracağımızı göreceğiz.

9.5.1 Linux Üzerinde Oracle

İlk önce, tüm Oracle işletim seviyesi işlemlerini yapmak için, `oracle` adında bir Unix kullanıcısı yaratmanız gerekiyor. Aşağıdaki komutları `root` olarak işletin.

```
groupadd dba

groupadd oinstall

groupadd oper

mkdir /home/oracle

chown -R oracle /home/oracle

useradd -g oinstall -G dba,oper oracle

passwd oracle
```

Oracle, bir Unix kuruluşundan belli ayarlar bekler, sistemin paketten çıktığı haliyle çalısmayacaktır. SuSE Linux için kernel seviyesinde yapılması gereken ayarlar aşağıda verilmiştir.

Liste 9.1: /etc/sysctl.conf

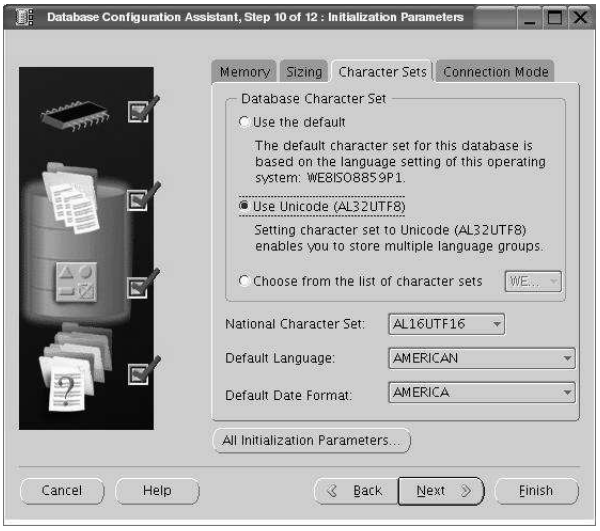
```
kernel.shmall = 2097152
kernel.shmmax = 2147483648
kernel.shmmni = 4096
kernel.sem = 250 32000 100 128
fs.file-max = 65536
net.ipv4.ip_local_port_range = 1024 65000
```

Bu ayarların işleme konması için, SuSE Linux üzerinde `/sbin/chkconfig boot.sysctl on` komutu işletilmelidir.

Artık kuruluş işlemini başlatabiliriz. Bunun için `oracle` kullanıcısı altından `./runInstaller` komutunun işletilmesi gerekiyor. Bu program, X Windows (10.8) üzerinden çalışan görsel bir programdır. Olağan değerleri kullanmak için "Next" düğmesine basarak geçebilirsiniz. Çoğu Oracle kuruluşu için olağan değerler uygundur. Daha ileri türden kuruluşlar için, DBA'inize danışın.

Oracle işler kodlarının kuruluşu bittikten sonra, sıra bir veri tabanı yaratmaya geliyor. Oracle kuruluş sırasında kurulan veri tabanı yerine bir başkasını yaratmak isterseniz, `dbca` görsel programını başlatarak istediğiniz tabanı yaratabilirsiniz. Aynı şekilde, taban için bir isim vererek, geri kalan olağan değerler ile taban yaratmak mümkündür.

Bu aşamada en önemli ve değişik yapmanız gereken işlem, taban seviyesinde Türkçe karakter desteği için yapmanız gereken değişikliktir. Şekil 9.3, istenilen karakter desteği için kullanılması gereken ekranı ve seçilmesi gereken değerleri gösteriyor.



Şekil 9.3: Türkçe Karakter Set Desteği

Taban yaratıldıktan sonra, `/etc/oratab` dosyasına girin, ve burada, yeni yarattığınız taban için olan satırın en sonundaki değeri "N" değerinden "Y" değerine getirin. Bu dosya, `dbstart` komutu kullanıldığında hangi veri tabanlarının başlatılması gerektiğini kontrol eder. Bundan sonra `oracle` kullanıcısından `dbstart` kullanarak tabanınızı başlatabilirsiniz.

```
oracle@linux:~> dbstart
```

```
SQL*Plus: Release 10.1.0.3.0 - Production on Mon Jun 6 15:07:36 2005

Copyright (c) 1982, 2004, Oracle. All rights reserved.

SQL> Connected to an idle instance.
SQL> ORACLE instance started.

Total System Global Area 285212672 bytes
Fixed Size                778776 bytes
Variable Size           95428072 bytes
Database Buffers       188743680 bytes
Redo Buffers              262144 bytes
Database mounted.

Database opened.
SQL> Disconnected from Oracle Database 10g Enterprise Edition Release
     10.1.0.3.0
- Production
With the Partitioning, OLAP and Data Mining options

Database "orcl" warm started.
```

Bu taban üzerinde eğer admin seviyesi işlemler yapmak isterseniz, `dbca`'den admin için verdiğiniz şifreyi kullanarak Oracle'a bağlanabilirsiniz. Meselâ, yeni bir kullanıcı yaratmamız gerekse:

```
sqlplus system/<şifre>@orcl

create user kitapdemo identified by kitapdemo default tablespace users;

grant dba to kitapdemo;
```

Şifre olarak `kitapdemo` metnini kullanacak, ve olağan tablo depolama yeri (tablespace) `users` olacak `kitapdemo` adında bir kullanıcı yarattık, ve bu kullanıcıya DBA hakları verdik.

Diğer bir makinadan yeni tabanınıza bağlanmak için, o makinada Oracle Client kuruluşunu yapmış olmanız gerekir. Bu kuruluşdan sonra Oracle dizinini bulun (meselâ `ORACLE_HOME`), ve bu dizin altındaki `tnsnames.ora` dosyasına servis makinanız vere yeni Oracle tabanınız hakkında bilgileri ekleyin:

Liste 9.2: ORACLE_HOME/ora92/network/admin/tnsnames.ora

```
SAMPLE =
(DESCRIPTION = (ADDRESS_LIST =
(ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.0.1)(PORT = 1521))
)
(CONNECT_DATA =
(SID=SAMPLE)
)
```

```
)
```

Burada belirtilen `HOST`, veri tabanını barındıran makinanın IP adresidir. `SID`, `dbca` ile yarattığınız veri tabanının ismidir. Artık makinanızdan `sqlplus` komutunu kullanarak uzaktaki tabana bağlanabilir ve tablolar yaratıp veri ekleyebilirsiniz. Daha önce yarattığımız `kitapdemo` kullanıcısını kullanalım.

```
sqlplus kitapdemo/kitapdemo@orcl

create table vsvs (kolon1 varchar2(20) ..);
...
```

9.5.2 Linux Üzerinde PostgreSQL

SuSE Linux üzerinde PostgreSQL kurmak için, öncelikle `http://www.postgresql.org/ftp/binary/v8.0.2/linux/suse/sles8-i386/` adresinden

- `postgresql-libs-8.0.2-1.i586.rpm`
- `postgresql-8.0.2-1.i586.rpm`
- `postgresql-server-8.0.2-1.i586.rpm`

dosyalarını indirin ve bu dosyaları `root` kullanıcısı tarafından `rpm -i` ile, teker teker ve gösterildiği sırada, işletin.

Bir PostgreSQL servisini kullanmak için ilk yapmanız gereken, veri tabanı alanını sıfırlamaktır. Fakat bunun için kullanılacak `initdb` komutunu `root` olarak işletmenize izin verilmez. PostgreSQL servisini işletmek için yeni bir kullanıcı yaratmanız gerekiyor. Unix kullanıcısı `root` olarak şunları işletin.

```
mkdir /home/postgres

useradd -d /home/postgres -s /bin/bash -p postgres postgres

chown -R postgres /home/postgres
```

Bu yeni kullanıcı olarak sisteme girin.

```
su - postgres

initdb -E UNICODE

/usr/bin/pg_ctl -D /var/lib/pgsql/data -l logfile start

createdb test
```

Yukarıdaki komutlarla, veri taban alanını sıfırladık ve Türkçe karakter kullanacak `test` adlı bir veri tabanı yarattık. En son olarak veri taban servisini başlattık.

Şimdi demo adında yeni bir kullanıcı ekleyelim. Sorulan y/n sorularına "Enter" tuşuna basarak geçebilirsiniz.

```
postgres@linux:~> createuser -P
Enter name of user to add: demo
Enter password for new user: <biz burada demo değerini kullandık>
Enter it again: <..>
Shall the new user be allowed to create databases? (y/n)
Shall the new user be allowed to create more new users? (y/n)
CREATE USER
```

PostgreSQL'ın çalıştığı sistemi ve servisini JDBC üzerinden gelecek dış bağlantılara açmak için, aşağıdaki dosyaları değiştirmeniz gerekiyor.

Liste 9.3: /var/lib/pgsql/data/postgresql.conf

```
listen_addresses = '*'
```

Liste 9.4: /var/lib/pgsql/data/pg_hba.conf

```
host all      all        192.168.0.2     255.255.255.255 trust
```

satırını ekleyin. Bu satırda belirtilen 192.168.0.2 IP değeri, *bağlanan* bilgisayarın IP adresi olmalıdır. Bu değerlerin etkiye geçmesi için, komut satırında postgres kullanıcısı olarak

```
/usr/bin/pg_ctl restart
```

komutunu işleterek yeni değerlere işleme sokabilirsiniz. Ayrıca, PostgreSQL kullanıcısının (örneğimizde demo) bir tabloya erişebilmesi için

```
grant all on <tablo ismi> to public;
```

komutunu vermeyi unutmayın.

9.5.3 Linux Üzerinde Mysql

MySQL'i Linux ve Solaris üzerinde kurmanın en sağlam yolu, RPM değil, tar¬ .gz sonekli bir dosyayı kullanıp bu dosyayı uygun yere açmanızdır. Bunun için, http://dev.mysql.com/downloads/ adresinden istediğiniz genel sürüm numarasına giderek, buradan Linux (non RPM package) downloads altındaki Standard satırından Pick A Mirror'a tıklayarak mysql-standard-xxx-pc¬ -linux-i686.tar.gz gibi bir dosyayı indirmeye başlayın. İndirdikten sonra aşağıdaki komutları root olarak sırasıyla uygulayın.

```
tar xvzf mysql-standard-4.0.23-pc-linux-i686.tar.gz

ln mysql-standard-4.0.23-pc-linux-i686 -s /usr/local/mysql

cd /usr/local/mysql
```

```
groupadd mysql

useradd -g mysql mysql

./scripts/mysql_install_db --user=mysql

chown -R root .

chown -R mysql data

chgrp -R mysql .

bin/mysqld_safe --user=mysql &

bin/mysql
```

TAR dosyasını açtık, veri tabanını hazırladık ve `mysqld_safe` ile veri taban servisini başlattık. En son komut ile, mysql komut satırına girmemizi sağladı. Bu komut satırında ilk iş olarak MySQL `root` kullanıcısı için bir şifre tanımlamanız iyi olur. Altta bu şifreyi `admin` olarak seçtik.

```
mysql> SET PASSWORD FOR root@'localhost' = PASSWORD('admin');
```

Artık, MySQL komut satırına salt `bin/mysql` ile değil, `bin/mysql -p` ile girmemiz gerekiyor. Şifre sorulunca `admin` girebiliriz.

En son yapılması gereken, `root` kullanıcısını dış makina bağlantılarına açmaktır. Bunu yapmazsak, MySQL'e sadece `localhost`'tan bağlanabiliriz, ki bu da dağıtık bir uygulama için pek faydalı olmazdı.

```
mysql> GRANT ALL ON *.* TO root@'%'
-> IDENTIFIED BY 'admin'
-> WITH GRANT OPTION;
```

Eğer `mysqld_safe` başlamaz ise, `mysqld` programını deneyebilirsiniz. Başlattıktan sonra `mysql` komutu problem çıkartırsa, meselâ `mysql.sock` dosyasının bulunamaması gibi, o zaman bu dosyayı istenen/aranan yere bir Unix sembol bağlantı (symbolic link) ile bağlayarak tekrar deneyebılırsınızç

```
ln -s /var/lib/mysql/mysql.sock /tmp/mysql.sock
```