
dataera

Veri güvenliğiniz için (KVKK) teknik ve idari önlemler alındı mı?

Did you take technical and administrative precautions for GDPR?

Sorgularınız ve raporlarınız olması gerektiği kadar hızlı mı?

Are your SQL statements and reports as fast as it should be?

Yedeklerinizi en son ne zaman test edildi?

When was the last time your backups were tested?

Güvenlik yamaları uygulandı mı?

Were security patches applied?

Farklı bir lokasyonda yedeklerinizin güncel kopyası var mı?

Do you have an up-to-date copy of your backups in a different location?

Kritik sistemleriniz yüksek erişilebilirlik mimarisine sahip mi?

Are your critical systems an architecture with high availability?

Kritik sistemlerinizde bir sorun olması durumunda tekrar çalışır hale getirilmesi için gerekli testler yapıldı mı?

Are your critical systems tested to work in case of a disaster?

Felaket Kurtarma Merkeziniz var mı? Olası bir afet durumunda sisteminizi ayağa kaldırabilmek için, farklı bir ilde sisteminiz yedekli mi?

Do you have a disaster recovery solution? Do you have a backup system in a different city or country to reset your system up in case of a disaster?

dataera

Teknoloji Bilişim Çözümleri

Information Technology Solutions

HAKKIMIZDA / ABOUT US

90'ların ortalarındaki teknoloji devrimi yeni departmanların doğmasını sağlamıştır. Teknolojinin ilerlemesi ve arşivlenmiş veri miktarındaki artış ile bu departman diğer departmanları domine etmeye başlamıştır.

Teknolojinin etkin kullanılması ve arşivlenen verinin kullanılabilir, faydalı bilgi haline getirilmesi, doğru raporlar üretilerek, rekabet ve kârlılık avantajına ciddi katkıda bulunabilmektedir.

Bu faktörleri göz önünde bulunduran Dataera, yerli ve yabancı, çoğunlukla telko ve finans sektörüne, uzun yıllar hizmet vermiş bir kadro tarafından kurulmuştur.

Veritabanı, Linux, Unix ve backup çözümlerinde hem saha tecrübeli hem de resmi sertifikasyona sahip uzman bir ekibe sahiptir.

Dataera, önlem almanın tedaviden daha kolay ve masrafsız olduğu bilinciyle hareket eder. Potansiyel felaket senaryolarına karşı gerekli önlemleri, işin en başından alıp müşterilerinin iş kaybına engel olur.

In the mid 90s the technology revolution ensured the creation of IT departments and this departments has begun to dominate other departments with the progress of the technology and the increasement in the amount of archived data.

Effective use of technology and making the archived data useful information, generating accurate reports can be a significant contribution to the advantage of competition and profitability.

Dataera is considering these factors that established by a team which mostly support both local and foreign telecominication and finance firms.

We have an expert team which has both field experience and official certification in the database, Linux, Unix and backup solutions.

Dataera acts with the awareness that precaution is easier and cheaper than treatment.

HİZMETLERİMİZ / SERVICES

Veritabanı (Oracle, PostgreSQL, MSSQL, MYSQL), İşletim sistemleri (Linux, Redhat Pardus ve Oracle solaris) Devops , CI/CD ve backup alanlarında danışmanlık, bakım, eğitim ve dış kaynak hizmetleri vermekteyiz.

Bu hizmetler hem saha tecrübeli hem de resmi sertifikasyona sahip uzman bir ekiple yürütülmektedir.

Dataera ekibi her zaman amatör ruhu canlı tutar ve hizmetlerini kontrol listelerine ve prosedürlerine göre verir.

Dış kaynak kullanmanın işletmeniz açısından avantajları olabilir. Tam zamanlı bir kişinin tüm günlerini dolduramayacak ya da dönemsel olarak gereksinim duyulan ihtiyaçlar için uzman bir ekipten maliyet avantajları ile faydalanabilirsiniz.

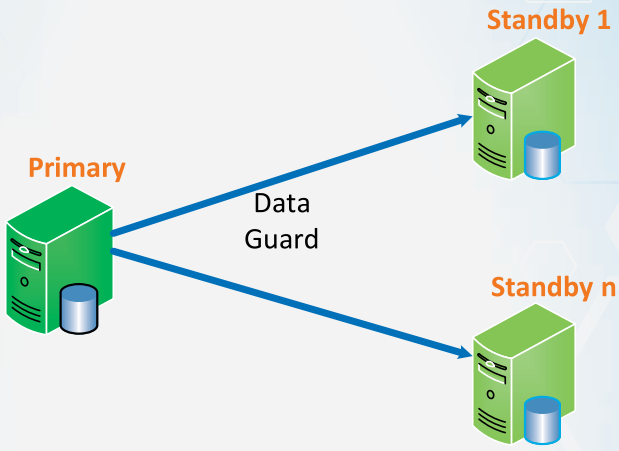
We provide consultancy on databases (Oracle, PostgreSQL, msSQL, mySQL), operating systems (Linux, Redhat Pardus and Oracle solaris) Devops, CI/CD and Backup as welll as maintenance training and outsource services with our experienced certified team.

Dataera team keeps the spirit up and works with check lists and procedures.

Outsourcing is a benefit for your company. It is cost saving for seasonal or temporary needs or for tasks that wont require a fulltime assignment to an employee.

ÇÖZÜMLERİMİZ / SOLUTIONS

Oracle Database Dataguard

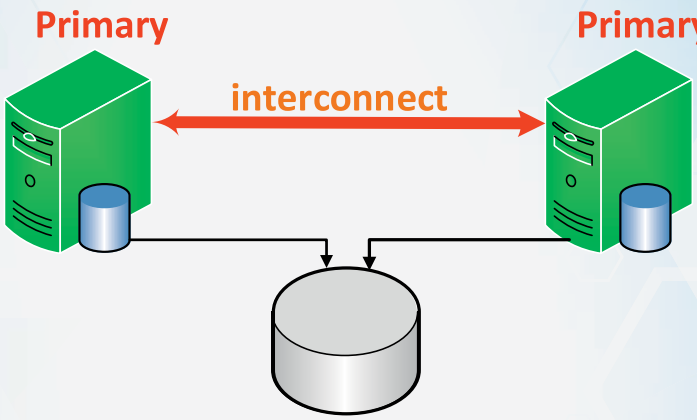


Aynı lokasyonda ve/veya farklı illerde felaket kurtarma merkezi çözümleri (FKM) olarak kullanılabilir. Bu yapıda 1 adet aktif (primary) ve 30 adete kadar pasif (standby) veritabanı sunucunuz olabilir. Eğer aktif dataguard lisansınız varsa, pasif veritabanınızı sadece okumaya açabilirsiniz.
Bu çözümde aktif sunucunuzun başına bir şey gelmesi durumunda pasif veritabanınızı aktif hale getirerek 5-10 dakika içinde kaldığınız yerden çalışmaya devam edebilirsiniz. Böylece iş/prestij kaybı yaşamazsınız.
Dataguard'ı enterprise sürüm lisansınız varsa kullanabilirsiniz. Eğer standart sürüm lisansınız varsa DBVisit gibi 3. parti bir yazılım kullanabiliriz.

Actually Data Guard is a disaster recovery solution that can be located at the same or different locations. It ensures high availability, data protection as well. In this structure, you can have 1 primary and up to 30 standby database servers. If you have active data guard license you can open standby database read only mode.
In case of disaster you can activate your passive database and continue to work within 5-10 minutes without job loss.
You can only use it if you have enterprise license otherwise we need use 3rd party solutions such as DBvisit solutions.

ÇÖZÜMLERİMİZ / SOLUTIONS

Oracle Real Application Cluster

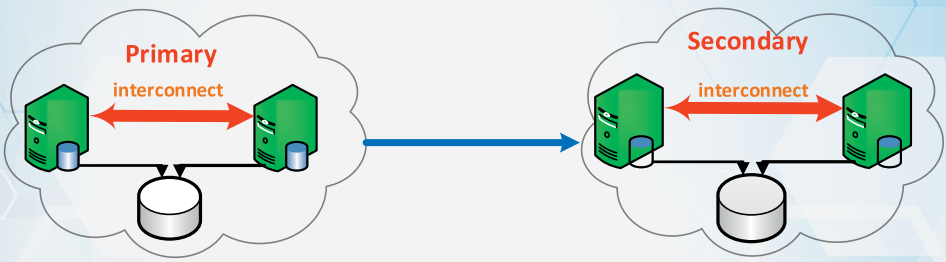


Oracle'ın cluster çözümünde bütün nodlar aktiftir, uygulamalar ya da kullanıcılar herhangi birine bağlanarak çalışabilirler. (Servislerle, uygulamaların/kullanıcıların hangi noda bağlanacağı kurallarla belirlenebilir.)
Bağlı olunan nodda bir sorun olması durumunda diğer nodlardan birine bağlanarak çalışmalarını sürdürebilirler. Herhangi bir hizmet kesintisi yaşanmaz. Kritik yerler için önerdiğimiz bir yapıdır.

All nodes are active in Oracle's cluster solution. Applications or users can connect to any of them. Which servers the users will connect to can be determined by the services. If there is a problem with the connected nod, they can continue to work by connecting to one of the other nods without job loss. We recommend that for critical businesses.

ÇÖZÜMLERİMİZ / SOLUTIONS

Oracle Maximum Availability Architecture



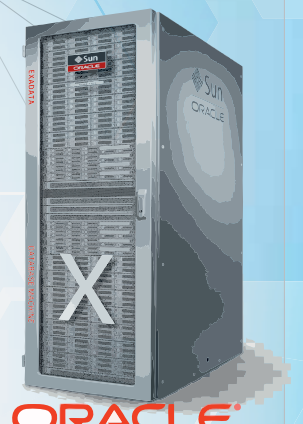
Önceki iki çözümde bahsedilen RAC ve Data Guard yapılarının beraber kullanıldığı çok kritik yerler için kullanılan maximum erişebilirlik mimarisidir.

In this Maximum Availability Architecture, mentioned in the previous two solutions are used together. We recommend for very critical businesses.

ÇÖZÜMLERİMİZ / SOLUTIONS

ExaData / ExaLogic / ODA

ORACLE
DATABASE APPLIANCE

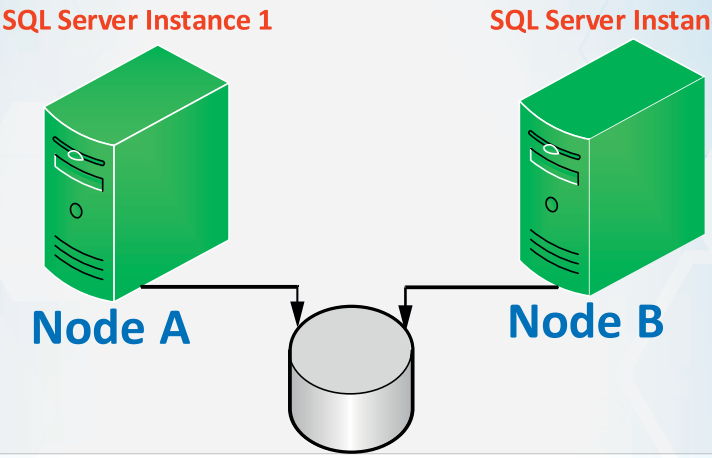


Yaşı ne olursa olsun, Exadata, Exalogic ve Oracle database appliance gibi yazılım&donanım gömülü mühendislik ürünlerine bakım danışmanlık ve destek hizmetlerini verebiliriz.

Regardless of age, we can provide maintenance consultancy and support services to engineering products such as Exadata, Exalogic and Oracle database appliance.

ÇÖZÜMLERİMİZ / SOLUTIONS

MSSQL Server FailOver Cluster

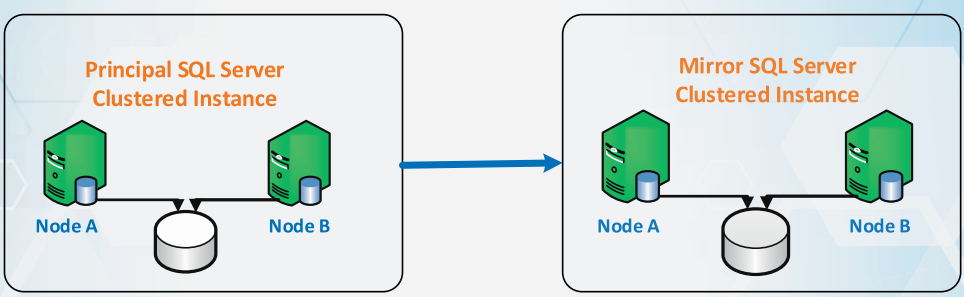


Aynı clusterda instance seviyesinde ayarlanabilen yüksek erişilebilirlik çözümüdür (msSQL High Availability HA). Nodlara aynı storage daki aynı diskler mount edildiği için veritabanı katmanında disk yedekliliği yoktur. Eğer storage larınız da clusterlı ise sorun olmayacaktır. Otomatik failover işlemi ile SQL servisi, sorunlu noddan, sorunsuz noda otomatik olarak taşınabilir. Veritabanı seviyesinde yapılamadığı, sadece instance seviyesinde yapılabildiği için ve pasif nodlardan okuma yazma yapılamadığı için esnek değildir.

High availability solution that can be set at the instance level in the same cluster. There is no disk redundancy at the database tier because the same disks are mounted.
With automatic failover, SQL service can be moved automatically from failed nod to other nodes. It can only be done at instance level and cannot open in read only mode.

ÇÖZÜMLERİMİZ / SOLUTIONS

MSSQL Server Database Mirroring

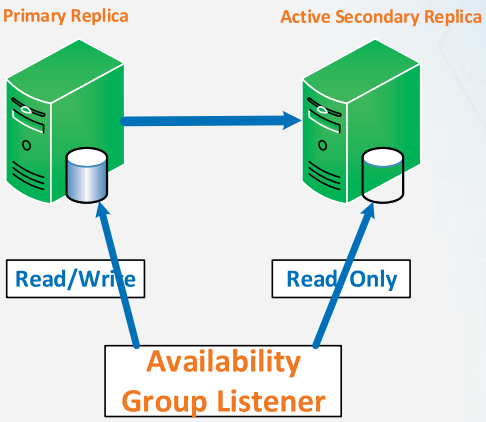


Burda da pasif noddan okuma/yazma yapılamaz. Okuma yapabilmek için snapshot alınması gerekir. Yüksek erişilebilirlik (High Availability – HA) için senkron , Felaket kurtarma merkezi (Disaster Recovery – DR) çözümü için asenkron ayarlanır. Veritabanı bazında mirroring yapılabilir. Eğer fazla sayıda veritabanınız varsa hepsi için ayrı ayrı işlemleri tekrar etmeniz gerekir. Aktif ve pasif nodlar verileri kendi disklerinde tuttukları için disk yedekliliği vardır. Lakin Microsoft, bu çözüme devam etmeme kararı aldı.

Cannot open in read only mode also. Snapshot needs to be taken to read data. It should be set sync for high availability and async for disaster recovery solution. Database level mirroring is possible.
If you have a large number of databases, you should repeat the operations for each one separately. Active and passive servers have disk redundancy because they keep their data on their disks. Microsoft has decided not to continue this resolution.

ÇÖZÜMLERİMİZ / SOLUTIONS

MSSQL AlwaysOn

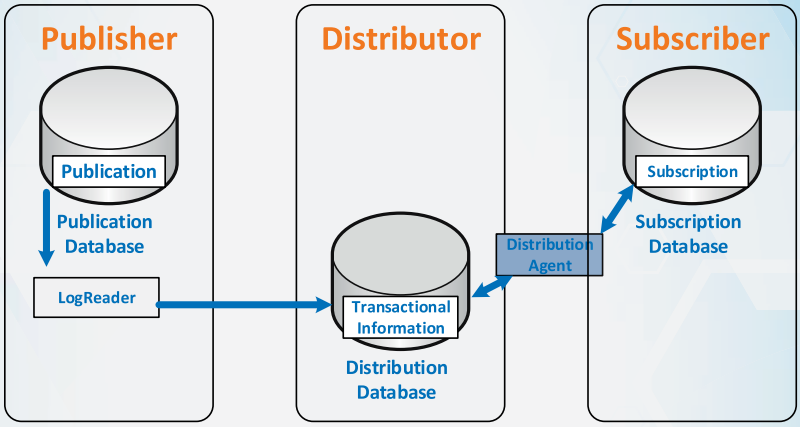


Veritabanı seviyesinde gruplar oluşturarak ayarlanabilir. Pasif noddan okuma yapılabilir. Page repair özelliğine sahip. Disk yedekliliği vardır. Yüksek erişebilirlik ya da felaket kurtarma merkezi çözümü olarak kullanılabilir. AlwayOn msSQL server'ın en esnek ve kullanışlı çözümüdür diyebiliriz.

Can be set by creating groups at database level. Passive node can be open read/only mode. Active and passive servers have disk redundancy because they keep their data on their disks and it has page repair feature. We can say that it is the most flexible and useful solution of Always On SQL server..

ÇÖZÜMLERİMİZ / SOLUTIONS

MSSQL Replication

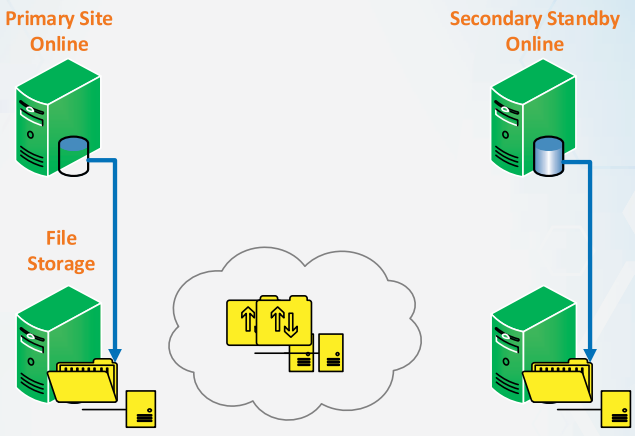


Mantıksal yani DML cümleleri ile replikasyon yapıldığından daha çok raporlama (OLP/DataWareHouse) için kullanılır. Örneğin, tablo bazında replikasyona izin verir hedef sunucuda ihtiyaçlarımıza göre objelerde değişikliklere gidebiliriz. İndex oluşturmak gibi. Replikasyon araçlarına örnek vermek gerekirse Oracle Golden Gate, dbVisit Realtime data replication, Striim vb.

Used for more reporting since it is replicated with SQL syntaxes. Allows table-based replication then you can create new indexes or delete unused ones. To give examples of replication tools dbVisit, Golden gate, Striim etc..

ÇÖZÜMLERİMİZ / SOLUTIONS

MSSQL LogShipping

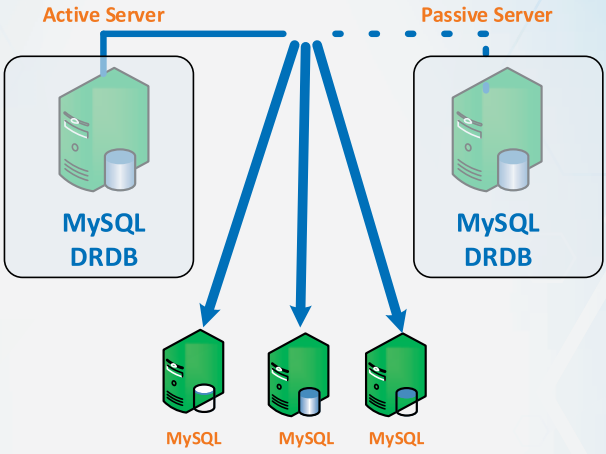


Felaket kurtarma merkezi çözümüdür. Veritabanı bazında yapılabilir , pasif nodda okuma yapılabilir.

Another disaster recovery centre solution. Can be config at the database level. Passive node can be open read/only mode.

ÇÖZÜMLERİMİZ / SOLUTIONS

MYSQL Cluster (Active-Passive)

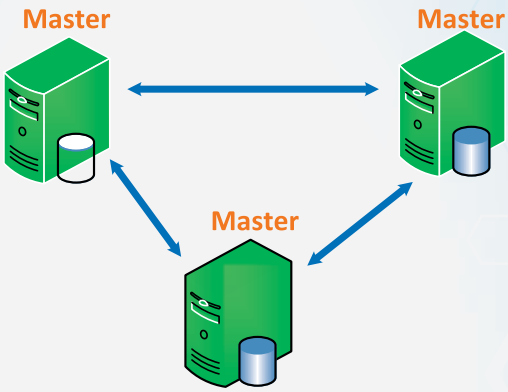


Önce açık kaynak kodlu ücretsiz olan MySQL, sonrasında Oracle'ın devreye girmesi ile hem enterprise (lisanslı/ücretli) sürümü hem de community sürümü (ücretsiz/açık kaynak kodlu) olmak üzere iki farklı seçenekle devam etmektedir. Ücretli ve ücretsiz sürümlerin sahip olduğu özellikler açısından farklılıklar mevcuttur. Kritik üretim ortamları için seçilen sürüme ve özelliklere dikkat edilmesi gerekir.
MySQL'in aktif-pasif çözümünde binary-log dosyaları standby sunucuya gönderilip işlenir. Bir felaket durumunda standby sunucu aktif edilerek hizmet kesintisi minimuma çekilir. MySQL tarafında çeşitli yüksek erişebilirlik ve felakat kurtarma merkezi çözümleri mevcuttur. İş ihtiyaçlarınıza göre en uygun çözüme birlikte karar verebiliriz.

MySQL, which is free of charge with open source code, is followed by two different options, namely enterprise version and community version, with the introduction of Oracle afterwards. When we compare enterprise edition (paid version) and community edition (free) there have different features. For critical production environments, attention should be paid to the selected versions and features.
In MySQL's active-passive solution, binarylog files are sent to the standby server and applied. In case of disaster standby server is activated and service interruption is minimized.

ÇÖZÜMLERİMİZ / SOLUTIONS

MYSQL Cluster (Multi-Master)

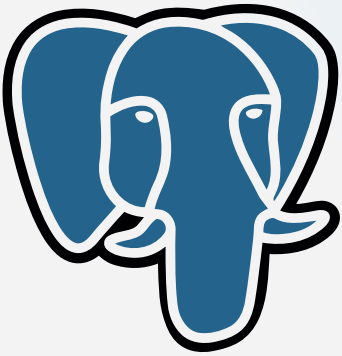


Paylaşımlı disk mimarisi yerine lokal diskler kullanılır her nod kendine gelen transactionları diğer masterlara replike eder. Arada çakışmaları önleyecek mekanizmalar mevcuttur.

Local disks are used instead of shared disk architecture. Each server replicates its own transactions to other masters. Mechanisms exist to prevent conflicts.

ÇÖZÜMLERİMİZ / SOLUTIONS

PostgreSQL



Açık kaynak kodlu ve 40 yılı aşkın geçmişi ile en güçlü veritabanı sistemlerinden biridir.

PostgreSQL, gömülü (built-in) ve ticari sürümlerle gelen bir çok yüksek erişebilirlik ve otomatik failover özelliklerine sahiptir.

Bu çözümler için, paylaşımlı disk mimarisi, transaction logların pasif sunuculara gönderilmesi ve SQL ifadelerin pasiflere gönderilmesi gibi farklı yöntemler kullanılmaktadır. Bu çözümlerin listesini aşağıda bulabilirsiniz. İş ihtiyaçlarınıza ve maliyet avantajlarına göre uygun yöntemi işletmenizde hayata geçirebiliriz.

Shared Disk Yerine geçme,
Dosya sistemi (Block Device) replikasyon,
Write-Ahead (Önce loga yazma) Log Shipping,
Mantıksal replikasyon
Tetikleme temelli replikasyon,
İfade temelli replikasyon,
Asenkron çoklu master replikasyon,
Senkron çoklu master replikasyon,
PAF, Postgres Automatic Failover
ve ticari sürümlerle gelen çözümler.

It is one of the most powerful database systems with open source code and over 40 years of history.

PostgreSQL has several high availability, automatic failover and disaster recovery centre solutions that come with built-in and commercial versions.

For these solutions, different methods are used such as shared disk architecture, sending transaction logs to passive servers and sending sql expressions to passive nodes.Below you can find a list of these solutions. Depending on your business needs and cost advantages, we can implement correct method to your environment.

Shared Disk Failover,
File System (Block Device) replication,
Write-Ahead Log Shipping,
Logical Replication,
Trigger-Based Master-Standby replication,
Statement-Based replication middleware,
Asynchronous Multimaster replication,
Synchronous Multimaster replication,
PAF, Postgres Automatic Failover,
and commercial solutions (EnterPrise DB).

ÇÖZÜMLERİMİZ / SOLUTIONS

Red Hat

Açık kaynak kodlu kurumsal işletim sistemi olan Red Hat linux. İşletmelerin kritik sistemlerinde tercih edilmektedir.
Uçtan uca her alanda ürünleri bulunan Red Hat'ın başlıca çözümleri ; işletim sistemi, orta katman, storage, sanallaştırma, alt yapı otomasyonu, bulut teknolojisi ve Sdx software defined everything gibi geleceğin teknolojileri oluşturmaktadır. İş ihtiyaçlarınıza uygun bilişim alt yapınızı uzman kadromuzla hayata geçirebiliriz.

Red Hat is an open source enterprise operating system, mostly preferred in business critical systems. Red Hat provides end to end solutions for your business. We can list Red Hat's solutions as follows ; OS, middleware, storage, virtualization, Cloud, infrastructure automation and Sdx software defined everything. we can implement correct method to your environment.

ÇÖZÜMLERİMİZ / SOLUTIONS

Solaris-Unix

Sun tarafından geliştirilmiş Unix işletim sistemidir. Oracle'ın satın alması ile ismi Oracle solaris olmuştur. Sparc ve x86_64 mimarilerine sahiptir. Özellikle kendine ait sparc mimarisi kritik yerler için tercih edilen işletim sistemidir. Ldom, opscenter, OVM, zone, cluster, zfs storage, exadata ve exalogic alanlarında danışmanlık, bakım ve kurulum hizmetleri vermekteyiz.

Solaris is a Unix OS developed by Sun Microsystems. After the Sun acquisition by Oracle, it was renamed Oracle Solaris. It has Sparc and x86_64 architectures.
Especially, its own sparc architecture is the preferred operating system for most critical environments. We can list Solaris's featured as follows ; Ldom (virtualization), Opscenter, OVM, zone, cluster, ZFS storage, ExaData and Exalogic. We can implement correct method to your environment.

Kitabın Adı: **Postgresql SQL & PLpgSQL**

Hazırlayan: **Meryem Sağol**

Tasarım: **Dataera VTY**

Isbn: **978-605-72878-1-6**

Baskı Tarihi: **2. Baskı / Kasım 2024**

Baskı ve Cilt: **Akademi Matbaa**
Sertifika No: 47610
Maltepe Mah. Davutpaşa Cd.
Güven İş Merkezi D Blok No: 83/114
Topkapı - Zeytinburnu / İstanbul

Yayınevi: **Şamil Yayıncılık Tic. San. Ltd. Şti.**
Sertifika No: 51260
Mehmet Akif Mah. Senem Sok. No: 2/A
Çekmeköy / İstanbul T:02166424250

dataera

dataera.com.tr
info@dataera.com.tr
0216 706 30 80

Postgresql

SQL & PLpgSQL

meryem sağol

İstanbul - 2024

dataera

İçindekiler

SQL

1. BÖLÜM

Veri Sorgulama (Quering Data) **28**
- Veri Sorgulama (Select) 30
- Sorgu Sonucunu Sıralama (Order By) 32
- Yinelenen Satırları Kaldırma (Distinct) 33

2. BÖLÜM

Veri Filtreleme (Filtering Data) **34**
- Where 36
- Limit 38
- In 39
- Between 41
- Like - ilike 42

3. BÖLÜM

Tabloların Birleştirilmesi (Joining Multiple Tables) 44

4. BÖLÜM

Veri Gruplama (Grouping Data) **50**
- Group by 52
- Having 53
- Having vs. Where 53

5. BÖLÜM

Set Operations **56**
- Union 58
- Intersect 61
- Except 61

6. BÖLÜM

Alt Sorgu (Subquery) **62**
- Alt Sorgular 64
- Any Operatörü 68

7. BÖLÜM

Veri, Ekleme Güncelleme ve Silme işlemleri (Modifing Data) **70**
Insert 72
Update 75
Update Join 77
Delete 78

8. BÖLÜM

Transaction **80**
Begin 82
commit 83
Rollback 84

9. BÖLÜM

CSV İmport Export **86**

10. BÖLÜM

Tablo Yönetim (Managing Tables) **94**
Veri Türleri 96
Tablo Oluşturma 97
Constraint 97
Create Tables AS 99
Alter, Rename, Drop, Truncate Table 100
Temporary 109
Copy Table 110

11. BÖLÜM

Kısıtlamalar (Constraints) **112**
Primary Key 114
Foreign Key 115
Check Constraint 115
Unique Constraint 116
Not-Null Constraint 117

12. BÖLÜM

Matematik İşlemleri Tarih & Zaman Fonsksiyonları **118**
(Mathematical Operations, count, sum, avg, max-min, abs, ceil, floor – Date Time Functions)

13. BÖLÜM

psql Komutları **126**

14. BÖLÜM

Örnek Yöntemler (SQL Recipes) **132**

PLpgSQL

1. BÖLÜM

PLpgSQL'e Giriş **140**
String Sabitleri 142
Block Yapısı 143

2. BÖLÜM

Değişkenler ve Sabitler (Variables & Constants) **146**
Değişkenler 148
Veri Türlerini Kopyalama 149
Subblock 150
Pl/Pgsql Satır Türleri 151
Kayıt Türü 152
Sabitler 153

3. BÖLÜM

Mesajlar ve Hatalr (Reporting Messages and Errors) **154**

4. BÖLÜM

Kontrol Yapıları (Control Structures) **158**
If Then 160
Else-If 161
If-Then-Elseif 162
Case 167
Loop 171
While Loop 173
For Loop 174
Sonuç Kümesi Üzerinde (Query Result) Loop Döngüsü 175
Dinamik Bir Sorgunun Sonuç Kümesinde Loop Döngüsü 176
Exit İfadesi 177
Continue İfadesi 179

5. BÖLÜM

Veri Kullanicı Tanımlı Fonksiyonlar (User - Defined Functions) **180**
Fonksiyon Oluşturma 182
Fonksiyonları Çağırma 185
Fonksiyon Parametreleri 186
Fonksiyon Overloading 189
Tablo Döndüren Fonksiyon 191
Fonksiyon Silmek 193

6. BÖLÜM

Stored Procedures **194**

7. BÖLÜM

Triggers 204

8. BÖLÜM

Views 214

9. BÖLÜM

Indexes.....220

B-tree Index..... 223
Hash Index..... 223
Brin Index 224

SQL

1

Veri Sorgulama

(Quering Data)

- Veri Sorgulama (Select)
- Sorgu Sonucunu Sıralama (Order By)
- Yinelenen Satırları Kaldırma (Distinct)

SELECT

Tablolardan ve view lerden select cümleleri ile veri sorgulaması yapılır.

```
SELECT  kolon1, kolon2, kolon3 FROM  tablo_adı;
```

ÖRNEKLER:

```
# Bir tablodaki tüm kolonları sorgular.
SELECT * FROM rental ;

# Belirli kolonları sorgulamak için.
SELECT country_name FROM adress ;
--veya
SELECT first_name, email, username FROM staff ;


# Spesifik bir kolon belirtmeyip '*' ifadesi kullandığınız
takdirde belirttiğiniz tablodaki tüm satır ve sütun sonuçları
gelecektir.
Select * from rental;


# Tabloda birden çok kolondaki değerleri çekmek istediğinizde
',' ile tek tek kolon adı belirtmeniz durumunda belirttiğiniz
sıra ile sorgu sonucu gelecektir.

SELECT address_id FROM address ;
SELECT country_id FROM city ;
SELECT first_name, email, username FROM staff ;
```

Kolon Birleştirme

Sql cümlesi first_name ve last_name satırlarının değerlerini tek bir kolonda getirir, diğer satırda da email kolonu yer alır.

```
SELECT  first_name || ' ' || last_name, email FROM  customer;
```

Alias

Sorgu ile kolona takma ad verilir.

```
SELECT kolon_adı AS alias_name FROM tablo_adı;

# Yukarıdaki sorgu first_name ve last_name satırlarını tek bir
kolonda bir araya getirip, kolona 'name' takma adını vermek
için kullanılır.

SELECT  first_name || ' ' || last_name as name , email FROM
customer;

SELECT 20/5  as sonuc ;
SELECT last_name as sure_name , first_name from customer ;
SELECT  first_name, last_name surname FROM customer;

# Sütunun takma adı boşluk içeriyorsa çift tırnak içerisinde
takma ad yazılır.

SELECT first_name || ' ' || last_name "full name" FROM customer;
```

Sorgu Sonucunu Belirli Bir Sıraya Göre Listeleme

```
SELECT select_listesi FROM tablo_adı ORDER BY sıralama_ifadesi1
[ASC | DESC], ..,sıralama_ifadesiN [ASC | DESC];

SELECT actor_id, last_name FROM actor ORDER BY actor_id DESC;

SELECT title, fulltext FROM film ORDER BY title ASC;

SELECT first_name, last_name FROM customer ORDER BY last_name
DESC, first_name ASC;
```

Bir tablodan veri sorguladığınızda, `SELECT` ifadesi satırları düzensiz bir sırada döndürür. Sonuç kümesinin satırlarını sıralamak için `SELECT` cümlesinde `ORDER BY` ifadesi kullanılır. `ORDER BY` ifadesi, bir `SELECT` cümlesi tarafından döndürülen satırları, bir sıralama ifadesine göre artan veya azalan düzende sıralamanıza olanak sağlar.

Sonuç kümesini birden çok sütuna veya ifadeye göre sıralamak istiyorsanız, bunları ayırmak için iki sütun veya ifade arasına virgül (,) koyulması gerekir. Satırları artan düzende sıralamak için 'ASC' seçeneği ve azalan düzende sıralamak için 'DESC' seçeneği kullanılır. `ASC` veya `DESC` seçeneklerinden biri belirtilmezse, `ORDER BY` varsayılan olarak `ASC`'yi kullanır.

```
SELECT first_name, LENGTH(first_name) len FROM customer ORDER BY
len DESC;
```

Sorgu customer tablosunda `first_name` sütunundaki değerlerin `LENGTH()` fonksiyonu ile (harf olarak) uzunluğunu hesaplar.
Uzunluk sonucunun kolonuna alias olarak `len` yazar ve bu kolona göre azalan değerde sıralama yapıp sonuçları getirir.

Sonuç Kümesinden Yinelenen Satırları Kaldırmak

```
SELECT DISTINCT kolon1 FROM tablo_adı;

SELECT DISTINCT kolon1,kolon2 FROM  tablo_adı;

SELECT  DISTINCT last_update FROM country ;

SELECT  DISTINCT last_update, country FROM country order by
country desc ;
```

`DISTINCT` ifadesi, bir sonuç kümesinden yinelenen satırları kaldırmak için `SELECT` cümlesinde kullanılır. `DISTINCT` ifadesi, yinelenen her grup için bir satır tutar. `SELECT` cümlesinin seçim listesindeki bir veya daha fazla sütuna uygulanabilir.

Birden çok sütun belirtirseniz, DISTINCT ifadesi bu sütunların değerlerinin birleşimine göre kopyayı değerlendirir.

2

Veri Filtreleme

(Filtering Data)

- Where
- Limit
- In
- Between
- Like - ilike

Veri Filtreleme

where

```
SELECT select_listesi FROM tablo_adi WHERE koşul ;

SELECT select_listesi FROM tablo_adi WHERE koşul ORDER BY
sıralama_ifadesi
```

`SELECT` ifadesi, bir tablodaki ya da view deki bir veya daha fazla sütundaki tüm satırları döndürür. Belirli bir koşulu karşılayan satırları seçmek için bir `WHERE` koşul tümcesi kullanılır.

`WHERE` koşulu oluşturmak için aşağıdaki karşılaştırma ve mantıksal operatörler kullanılır.

Operatör	Açıklama
=	Eşittir
>	Büyüktür
<	Küçüktür
>=	Eşittir veya büyüktür
<=	Eşittir veya küçüktür
< > veya !=	Eşit değildir
AND	VE (Mantıksal operatör)
OR	VEYA (Mantıksal operatör)
IN	Bir değer, listedeki herhangi bir değerle eşleşirse true değerini döndürür
BETWEEN	Bir değer, belirtilen değer aralığındaysa true değerini döndürür.
LIKE	Bir değer, belirtilen kalıp ile eşleşirse true değerini döndürür
IS NULL	Bir değer NULL değilse true değerini döndürür.
NOT	Diğer operatörlerin sonucunu olumsuzlaştırır.

```
SELECT last_name, first_name FROM customer WHERE first_name =
'Jamie' AND  last_name = 'Rice';
```

```
SELECT first_name, last_name FROM customer WHERE  first_name IN
('Ann','Anne','Annie');
```

first_name değeri ('Ann','Anne','Annie') olan satırları ve buna karşılık gelen last_name değerlerini getirir.

```
SELECT first_name, last_name FROM customer WHERE  first_name LIKE
'Ann%' ORDER BY last_name DESC;
```

first_name kolonundaki değerlerden Ann ile başlayanları ve karşılığındaki last_name sütununu getirir ve last_name sütununa göre alfabetik sıraya göre azalan şekilde sıralar.

```
SELECT first_name, LENGTH(first_name) name_length FROM customer
WHERE  first_name LIKE '%Al%' AND LENGTH(first_name) BETWEEN 3
AND 5 ORDER BY name_length;
```

Sorgu first_name kolonundaki değerlerin uzunluğunu hesaplar, first_name uzunluğu 3 ile 5 arasında olan değerlerden içerisinde 'Al' ifadesi bulunanları ekrana getirip first_name uzunluğuna göre artan şekilde sıralar. LIKE '%Al%' içerisinde Al ifadesi bulunan değerleri filtreler.

```
SELECT  first_name,  last_name FROM customer WHERE first_name
LIKE 'Bra%' AND last_name <> 'Motley';
```

first_name değeri 'Bra' ile başlayanları ve last_name kolonu 'Motley' ifadesine eşit olmayan sonuçları getirir.

limit

`LIMIT` ifadesi, sorgu tarafından döndürülen satır sayısını sınırlayan `SELECT` ifadesinin opsiyonel bir yan tümcesidir.

```
SELECT select_listesi FROM tablo_adı ORDER BY sıralama_ifadesi
LIMIT satır_sayısı;
```

`satır_sayısı` satırlarını döndürmeden önce birkaç satırı atlamak isterseniz, aşağıdaki ifadedeki gibi `LIMIT` yan tümcesinden sonra gelen `OFFSET` yan tümcesini kullanırsınız:

```
SELECT select_listesi FROM tablo_adı LIMIT satır_sayısı OFFSET
atlanacak_satır ;

SELECT film_id, title, release_year FROM film ORDER BY film_id
LIMIT 5;
```

```
SELECT film_id, title,release_year FROM film ORDER BY film_id
LIMIT 5 OFFSET 4 ;
```

Sorgu 4 satırı atlayıp, 5.satırdan itibaren 5 satır sonuç getirir.

```
SELECT film_id, title,rental_rate FROM film ORDER BY rental_rate
DESC LIMIT 10 ;
```

in

Bir değerin değerler listesindeki herhangi bir değerle eşleşip eşleşmediğini kontrol etmek için `WHERE` yan tümcesinde IN operatörü kulanılır.

`değer IN (değer1, değer2,...)`
Değer listesi, sayılar, string ifadeler gibi değişmez değerlerin bir listesi veya aşağıdaki gibi bir `SELECT` ifadesinin sonucu olabilir:

`değer IN (SELECT kolon_adı FROM tablo_adı);`
(Parantez içerisindeki sorgu bir alt sorgudur ve sonraki bölümlerde detaylı bilgi verilecektir.)

```
SELECT customer_id, rental_id, return_date FROM rental WHERE
customer_id IN (1, 2) ORDER BY return_date DESC;
```

`customer_id` değeri 1 ve 2 olan değerlerin sonuçları döner.

```
SELECT customer_id FROM rental WHERE CAST (return_date AS DATE) =
'2005-05-27' ORDER BY customer_id;
```

`return_date` değeri '2005-05-27' olan sütunlara ait customer_id değerini getirir.

```
SELECT customer_id, rental_id, return_date FROM rental WHERE
customer_id NOT IN (1, 2);
```

`customer_id` değeri 1 ve 2 olanlar hariç diğer sonuçları getirir.

```
SELECT customer_id, rental_id, return_date FROM rental WHERE
customer_id <> 1 AND customer_id <> 2;
```

Bir önceki sorgu ile aynı sonucu getirir.

```
SELECT
      customer_id,
      first_name,
      last_name
FROM customer
WHERE
customer_id IN (
            SELECT customer_id
            FROM rental
            WHERE CAST (return_date AS DATE) = '2005-05-27' )
ORDER BY customer_id;
```

Çalışma mantığı şu şekildedir;
`IN` fonksiyonu içerisindeki sorgu çalışır. Sorgu sonucu dönen `customer_id` değerleri `IN` fonksiyonu içerisinde geçerli olup, o değerlere uygun kolonlar sonuç olarak ekrana gelir.

between

Bir değeri bir dizi değerle eşleştirmek için BETWEEN operatörü kullanılır. Değer, düşük değerden büyük veya ona eşit ve yüksek değerden küçük veya ona eşitse, ifade true, aksi takdirde false döndürür.

`BETWEEN` operatörünü,

'`büyük veya eşittir ( >=)` ' veya '`küçük veya eşittir (<=)`' operatörlerini kullanarak da yeniden yazabilirsiniz.

```
.. BETWEEN alçak_değer AND yüksek_değer;
```

```
SELECT customer_id, payment_id, amount FROM payment WHERE
amount BETWEEN 8 AND 9;

SELECT customer_id, payment_id, amount FROM payment WHERE
amount NOT BETWEEN 8 AND 9;

SELECT customer_id, payment_id, amount, payment_date FROM
payment WHERE payment_date BETWEEN  '2007-02-07' AND '2007-02-
15';
```

like ve ilike

Kalıp eşleştirmelerini kullanarak verileri sorgulamak için `LIKE` ve `ILIKE` operatörleri kullanılır.

```
SELECT first_name, last_name FROM customer WHERE first_name LIKE
'Jen%';
```

`first_name` değeri Jen ile başlayan sonuçları getirir. Sonuçta Jen'den sonra gelecek olan kısım önemli değildir.

```
SELECT first_name, last_name FROM customer WHERE first_name LIKE
'%er%' ORDER BY first_name;
```

İçerinde 'er' harfleri olan first_name değerlerini ve ona karşılık gelen `last_name` değerlerini getiren ve `first_name` değerlerine göre artan şekilde sıralayan sorgudur.

```
SELECT first_name, last_name FROM customer WHERE first_name LIKE
'_her%' ORDER BY  first_name;
```

Bu sorguda `first_name` değerinin ilk harfi bilinmez(_) ancak 2.harften itibaren 'her' harflerini içeren sorgu döner.'her' harflerinden sonra devamı önemli değil.

```
SELECT first_name, last_name FROM customer WHERE first_name NOT
LIKE 'Jen%' ORDER BY first_name ;
SELECT first_name, last_name FROM customer WHERE first_name ILIKE
'BAR%';
```

`ILIKE` fonksiyonunun büyük küçük harf duyarlılığı yoktur.

```
SELECT first_name, last_name FROM customer WHERE first_name ILIKE
'BaR%';
```

Bir önceki sorgu ile aynı sonuç döner.

Veritabanında NULL, boş veya eksik bilgi veya uygulanamaz anlamına gelir. NULL bir değer değildir, bu nedenle onu sayılar veya dizeler gibi diğer değerlerle karşılaştıramazsınız. NULL'un bir değerle karşılaştırılması her zaman NULL ile sonuçlanır, bu da bilinmeyen bir sonuç anlamına gelir.

```
SELECT first_name, last_name , email FROM customer WHERE email
IS NULL;
```

Email kolonu null olan first_name ve last_name değerlerini getirir.

```
SELECT first_name, last_name, email FROM customer WHERE email IS
NOT NULL;
```

Email kolonu null olmayan first_name ve last_name değerleri döner.

3

Tabloların Birleştirilmesi

(Joining Multiple Tables)

JOIN

Customer

Column	Type
customer_id	integer
store_id	smallint
first_name	character
last_name	character
email	character
address_id	smallint
activebool	boolean
create_date	date
last_update	times with
active	integer

Payment

Column	Type
payment_id	integer
customer_id	smallint
staff_id	smallint
rental_id	integer
amount	numeri
payment_date	timestam

Staff

Column	Type
staff_id	integer
first_name	character
last_name	character
address_id	smallint
email	character
store_id	smallint
active	boolean
username	character
password	character
last_update	timestamp
picture	bytea

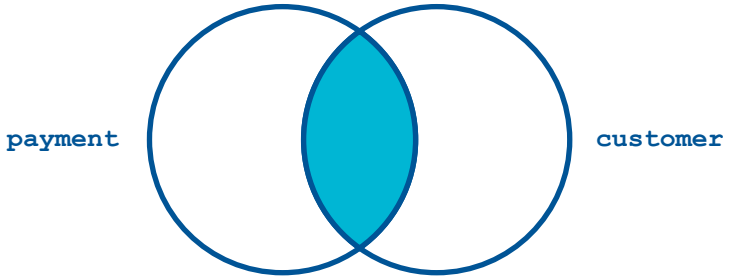
`INNER JOIN` iki veya daha fazla tablodaki ilişkili değerleri seçmek/birleştirmek için kullanılır. Özetle; SQL INNER JOIN ifadesi ortak değere sahip tabloları birleştirmek için kullanılır.

```
SELECT c.customer_id, first_name, amount, payment_date
FROM customer as c INNER JOIN payment p ON p.customer_id=
c.customer_id ORDER BY payment_date DESC;
```

Sorguya iki tane tablo katılmış. Customer tablosundaki `customer_id` değeri ile payment tablosundaki `customer_id` değerlerini karşılaştırıp eşit olanların `customer_id`,`first_name`,`amount` ve `payment_date` değerlerini getiren sorgudur.

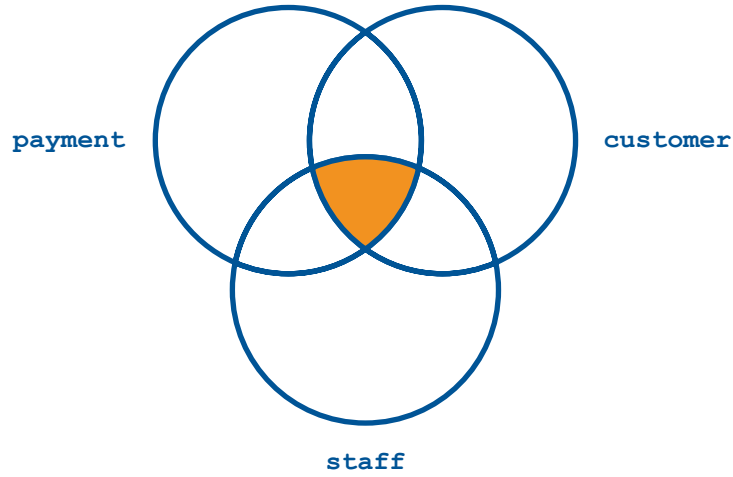
"`customer as c`" ile customer tablosuna '`c`' takma adı verilmiştir.
`c.customer_id = customer.customer_id` alias verilmesinin sebebi sorguyu kısaltmaktır.
"`payment p`" ile payment tablosuna '`p`' takma adı verilmiştir.

Her iki tablo da aynı `customer_id` sütununa sahip olduğundan, `USING` ifadesini kullanılabilir.




```
SELECT customer_id,first_name,last_name,amount,payment_date
FROM customer INNER JOIN payment USING(customer_id) ORDER BY
payment_date;

SELECT c.customer_id, c.first_name customer_first_name, c.last_
name customer_last_name, s.first_name staff_first_name, s.last_
name staff_last_name, amount, payment_date FROM customer c INNER
JOIN payment p ON p.customer_id = c.customer_id INNER JOIN staff
s ON p.staff_id = s.staff_id ORDER BY payment_date;
```




```
SELECT inventory.film_id FROM rental INNER JOIN inventory ON
inventory.inventory_id = rental.inventory_id WHERE return_date
BETWEEN '2005-05-29' AND '2005-05-30';
```

left join

sol tablodaki (tablo1) tüm satırları ve koşul ile belirtilen sağ tablodaki (tablo2) satırları seçmek/birleştirmek için kullanılır. Sol tablodaki değer ile sağ tablodaki eşleşmeyen değer olması durumunda sağ tablodaki değerler `'NULL'` değerini alır.

```
SELECT alanadlari FROM tablo1 LEFT JOIN tablo2 ON tablo1.alan_
adi = tablo2.alan_adi WHERE sartlar;
```

```
SELECT film.film_id, title, inventory_id FROM film LEFT JOIN
inventory ON inventory.film_id = film.film_id ORDER BY title;
```

`Film` tablosundaki her satır, `inventory` tablosunda sıfır veya birçok satıra sahip olabilir. `inventory` tablosundaki her satır, `film` tablosunda yalnızca bir satıra sahiptir. `film_id` sütunu, `film` ve `inventory` tabloları arasındaki bağlantıyı kurar.

```
SELECT film.film_id, film.title, inventory_id FROM film LEFT JOIN
inventory  ON inventory.film_id = film.film_id WHERE inventory.
film_id IS NULL ORDER BY title ;
```

Her iki tablo da ON yan tümcesinde kullanılan sütun adına sahipse, USING ifadesini şu şekilde kullanabilirsiniz:

```
SELECT f.film_id, title, inventory_id FROM film f LEFT JOIN
inventory i USING (film_id) WHERE i.film_id IS NULL ORDER
BY title;
```

self join

self join işlemi, bir tablonun kendisi ile join işlemine girmesi işlemidir. Tablo içinde bulunan satırlar arasında referans ilişkisi vardır.

ÖRNEK: Sorgu aynı uzunluktaki film çiftlerini ve uzunluklarını ekrana getirir.

```
SELECT f1.title, f2.title, f1.length FROM film f1
 INNER JOIN film f2 ON f1.film_id <> f2.film_id
 AND f1.length = f2.length;
```

4

Veri Gruplama

(Grouping Data)

- Group by
- Having
- Having vs. Where

group by

GROUP BY, SELECT ile getirilen veriyi gruplar.

```
SELECT customer_id, SUM (amount) FROM payment GROUP BY
customer_id;
```

Sorgu payment tablosundaki `customer_id` değerlerini bulur, aynı `customer__id` değerine sahip olanların 'amount' değerlerini toplar ve `customer_id`' ye göre gruplar.
Yani spesifik bir `customer_id` değerine sahip olan satırların amount değerleri toplamını verir.

```
SELECT customer_id, SUM (amount) FROM payment GROUP BY
customer_id ORDER BY SUM (amount) DESC;

SELECT first_name || ' ' || last_name full_name, SUM (amount)
as amount FROM payment INNER JOIN customer USING (customer_id)
GROUP BY full_name ORDER BY amount DESC;
```

Sorgu her customer kolonu için toplam tutarı (SUM(amount)) almak için INNER JOIN yan tümcesi ile GROUP BY deyimini kullanır. Bu sorgu payment tablosunu customer tablosuyla birleştirir ve full_name kolonuna göre gruplandırır.

```
SELECT staff_id , COUNT (payment_id)  FROM payment GROUP BY
staff_id;
SELECT  customer_id,  staff_id,  SUM(amount) FROM  payment GROUP
BY  staff_id,  customer_id ORDER BY customer_id;
SELECT  DATE(payment_date) paid_date,  SUM(amount) sum FROM
payment GROUP BY DATE(payment_date);
```

`Payment_date` bir timestamp sütunudur. Ödemeleri(payments) tarihlere göre gruplandırmak için, önce timestamp'ler tarihlere dönüştürülmeli. Ödemeleri (payments) sonuç tarihine göre gruplandırmak için DATE() işlevini kullanılır.Sorgu sonucu paid_date alias'ına sahip sütundaki tarihlerde alınacak ödemelerin toplamı sum alias'ına sahip sütunda belirtilir.

having

`Having` yan tümcesi, bir grup veya küme için bir arama koşulu belirtir. HAVING, belirli bir koşula göre grupları veya kümeleri filtrelemek için genellikle `GROUP BY` ifadesiyle birlikte kullanılır.

NOT: HAVING yan tümcesi SELECT yan tümcesinden önce değerlendirildiğinden, HAVING ifadesinde sütun takma adlarını kullanamazsınız. Çünkü HAVING ifadesi değerlendirilirken, SELECT yan tümcesinde belirtilen sütun takma adları kullanılamaz.

having vs where

`WHERE` yan tümcesi, belirtilen bir koşula göre satırları filtreler. Ancak HAVING yan tümcesi, belirtilen bir koşula göre satır gruplarını filtreler.
Başka bir deyişle, WHERE yan tümcesi satırlara uygulanırken HAVING yan tümcesi satır gruplarına uygulanmaktadır.

```
SELECT customer_id, SUM (amount) FROM payment GROUP BY
customer_id HAVING SUM (amount) > 200;
SELECT store_id, COUNT (customer_id) FROM customer GROUP BY
store_id;
```

Sorgu store_id'lere göre gruplandırma yapar ve bu store_id değerlerine sahip kaç tane `customer_id` satırı olduğunu veya kaç tane satır olduğunu hesaplar.(COUNT(customer_id)) Aşağıdaki sorgu ile aynı sonucu verir.

```
SELECT store_id, COUNT (*) FROM customer GROUP BY store_id;

SELECT store_id, COUNT (customer_id) FROM customer GROUP BY
store_id HAVING COUNT (customer_id) > 300;
```

Bu sorguda ise gruplandırma yapıldıktan ve satır sayıları hesaplandıktan sonra toplam customer_id satır sayısı 300'den büyük olan store_id değerini ve bu store_id değerine sahip customer_id satır sayısını ekrana getirir.

```
SELECT rental_id , min(amount) min_amount FROM  payment GROUP
BY rental_id HAVING min(amount)>10 ORDER BY min(amount) ;
```

Sorgu rental_id değerine göre gruplandırma yaparken bu rental_id değerlerine karşılık gelen minimum amount değerini hesaplar. Minimum amount değeri 10'dan büyük rental_id değerlerin ekrana getirir.

5

Set Operations

- Union
- Intersect
- Except

union

İki veya daha fazla `SELECT` ifadesinin sonuç kümelerini tek bir sonuç kümesinde birleştiren operatördür. `UNION` operatörünü kullanarak iki sorgunun sonuç kümelerini birleştirmek için sorguların aşağıdaki kurallara uyması gerekir.
1)Her iki sorgunun select listesindeki sütunların sayısı ve sırası aynı olmalıdır.
2)Veri türleri uyumlu olmalıdır.
3)`UNION` operatörü, birleştirilmiş veri kümesinden tüm yinelenen satırları kaldırır. Yinelenen satırları korumak için `UNION ALL` ifadesi kullanılır.

```
CREATE TABLE customer_1 ( name VARCHAR NOT NULL, category_id
SMALLINT );
CREATE TABLE customer_2 ( name VARCHAR NOT NULL, category_id
SMALLINT );

INSERT INTO customer_1(name, category_id)
VALUES ('Derya',36),
       ('Ali',19),
       ('Gamze',72),
       ('Meltem',57);

INSERT INTO customer_2(name, category_id)
VALUES ('Derya',36),
       ('Arda',20),
       ('Tarık',97),
       ('Gül',02);
```

Öncelikle customer_1 ve customer_2 adında müşteri bilgileri içeren tablolar oluşturulup, tablolara veriler girilmiştir.

union

```
SELECT * FROM customer_1
 UNION
SELECT * FROM customer_2

 name   | category_id
--------+-------------
 Gül    |           2
 Ali    |          19
 Arda   |          20
 Meltem |          57
 Derya  |          36
 Gamze  |          72
 Tarık  |          97
(7 rows)
```

Select sorgusu sonucu union operatörü sayesinde iki tablonun verileri alt alta ekrana gelmiştir. İki tabloda da aynı sonuçlara sahip satırlar sadece bir kere yazdırılmıştır. (Resim 5-1)

union all

```
SELECT * FROM customer_1
 UNION ALL
SELECT * FROM customer_2

name    | category_id
--------+-------------
 Derya  |          36
 Ali    |          19
 Gamze  |          72
 Meltem |          57
 Derya  |          36
 Arda   |          20
 Tarık  |          97
 Gül    |           2
(8 rows)
```

Union all operatörü sayesinde sorgu sonucunda iki tablodaki tüm veriler, tekrar eden kayıtlar dikkate alınmaksızın sıralanmıştır. (Resim 5-2)

intersect

Union operatörü gibi birden fazla select sorgusunun sonuç kümelerini birleştirmeye yarayan bir diğer operatördür. Intersect ile tabloların sadece aynı olan satırları 1 kere yazdırılır.

```
SELECT * FROM customer_1
 INTERSECT
SELECT * FROM customer_2

name  | category_id
-------+-------------
 Derya |          36
(1 row)
```

except

Bu operatör ile ilk select sorgusunda yazılan tabloda olan ve diğer select ifadesinde olmayan sonuçlar yazdırılır.

```
SELECT * FROM customer_1
 EXCEPT
SELECT * FROM customer_2
name  | category_id
--------+-------------
 Gamze  |          72
 Meltem |          57
 Ali    |          19
(3 rows)
```

6

Alt Sorgu

(Subquery)

- Alt Sorgular
- Any Operatörü

Örnek: Kiralama oranı (`rental_rate`) ortamalama kiralama oranından daha yüksek olan filmlere iki adımda ulaşabiliriz.
1-SELECT ifadesini ve AVG (ortalama) fonksiyonunu kullanarak ortalama kiralama oranı (AVG(rental_rate)) bulunur.

```
SELECT AVG (rental_rate) FROM film;
(Sonuç 2.98 döner.)
```

2-Bulduğumuz bu AVG(`rental_rate`)değerini ikinci sorguda WHERE koşulu ile kullanarak ortalamanın üstündeki filmleri listeleyebiliriz.

```
SELECT film_id, title, rental_rate FROM film WHERE rental_rate >
2.98;
```

Bu şekilde iki adımda ulaşmak yerine alt sorgu ile tek seferde istenen sonuca ulaşılabilir.

```
SELECT film_id, title, rental_rate FROM film
      WHERE
      rental_rate > (SELECT AVG (rental_rate)  FROM film)
      ORDER  BY rental_rate;
```

Parantez içindeki sorguya alt sorgu veya iç sorgu denir.
Alt sorguyu içeren sorgu, dış sorgu olarak bilinir.

PostgreSQL, bir alt sorgu içeren sorguyu aşağıdaki sırayla çalıştırır ;

1-İlk olarak, alt sorguyu çalıştırır.

2-Sonucu alır ve dış sorguya iletir.

3-Dış sorguyu yürütür.

Örnek: `return_date` (iade tarihi) '2005-05-30' ile '2005-05-29' arasında olan filmlerin listelenmesi.

`return_date`, `film_id` ve `title`(film ismi) değerlerine ihtiyaç var.
`film_id` ve `return_date` kolonlarını baz alırsak her iki kolonun da bulunduğu bir tablo olmadığı için alt sorgu (subquery) ve birleştirme (inner join) fonksiyonlarını kullanabiliriz.

- `film_id` değeri `film` tablosunda ve inventory tablosunda bulunmakta.
- `return_date` değeri de `rental` tablosunda bulunmakta.
- `Rental` tablosu ile inventory tablosunun ortak kolonu olan `inventory_id` eşleştirme için kullanılabilir.

Öncelikle `rental` ve `inventory` tablosunda eşit olan `inventory_id` değerleri bulunur. Bu eşit olan değerlerden `where` koşuluna uygun olanlar tutulur.
Inventory tablosunda bu tutulan `inventory_id` değerlerine karşılık gelen `film_id` değerleri seçilir.
Bulduğumuz bu `film_id` değerlerini `IN` fonksiyonu ile kullanıp istediğimiz tarihlerde iade edilecek olan filmleri ve başlıkları `film` tablosundan çekip sıralarız.

Sonuç :

```
SELECT film_id, title FROM film
WHERE film_id IN
      ( SELECT inventory.film_id FROM rental INNER JOIN
      inventory ON inventory.inventory_id = rental.inventory_id
      WHERE return_date BETWEEN '2005-05-29' AND '2005-05-30' )
```

Bu sorgu ile aynı sonucu döndürecek sorgulara örnek verecek olursak;

```
SELECT film.film_id, film.title FROM film
      INNER JOIN inventory ON inventory.film_id = film.film_id
      INNER JOIN rental ON rental.inventory_id = inventory.
      inventory_id WHERE rental.return_date BETWEEN '2005-05-29'
      AND '2005-05-30'  ;
```

Örnek: Ödeme miktarı (amount) 0.99 olan müşterilerin `rental_date`, `rental_id` ve ödeme miktarlarını ekrana getiren sorgu şu şekilde yazılır.

```
SELECT rental.rental_date, rental.rental_id, payment.amount
FROM rental
      INNER JOIN payment USING(rental_id)
      WHERE rental_id
      IN(SELECT rental_id from payment where amount = 0.99)
      ORDER BY rental_date;
```

Örnek: `first_name` değeri 'Jared' olan müşteriyle aynı `customer_id` değerine sahip olan müşterilerin ödeme miktarlarına (amount) %50 zam yapan sorgu şu şekilde yazılır.

```
UPDATE payment SET amount = amount+amount *0.5
      WHERE customer_id=
             (SELECT customer_id FROM
             customer WHERE first_name = 'Jared');
```

Ödeme miktarı yani amount değeri payment tablosunda first_name değeri ise customer tablosunda bulunmakta. Bu iki tablonun ortak olan customer_id kolonundan yola çıkıldı. Payment tablosunda değişiklik yapılacak satırlar customer_id değeriyle koşullandırıldı. Alt sorgu ile 'Jared' isimli müşterinin customer_id değeri elde edildi.

Örnek: ‘ Chamber Italian’ adlı filmi kiralayan müşterilerin yapacağı ödemeye %20 zam yapan sorgu adım adım şu şekildedir. (İç içe kullanılan sorgular tek tek adım adım açıklanmıştır.) Sorgu için ilişkili tablolar bulunmalı ve kullanılmalıdır.

1-Film’in isminden yola çıkarak film_id değeri ve buna karşılık gelen inventory_id değerleri öğrenilir.

```
SELECT inventory_id, film_id FROM inventory WHERE film_id=(SELECT
film_id FROM film WHERE title = ‘Chamber Italian’) ;
```

2- Bulunan inventory_id değerleri kullanılarak filmin rental_id değeri öğrenilir. Inventory_id değerlerinin kullanılabilmesi için bir önceki sorgu alt sorgu olur.

```
SELECT rental_id ,inventory_id FROM rental WHERE inventory_id
IN
(SELECT inventory_id FROM
      inventory WHERE film_id=
      (SELECT film_id from film WHERE title =’Chamber Italian’))
```

3-Elde edilen rental_id değerleri payment tablosunda kullanılır. Payment tablosunda bu rental_id değerlerine karşılık gelen amount değerlerine %20 zam uygulanır. Rental_id değerlerinin kullanılabilmesi için bir önceki sorgu alt sorgu olur.

```
UPDATE payment SET amount=amount+amount*0.2
      WHERE rental_id IN(SELECT rental_id FROM rental
            WHERE inventory_id IN
            (SELECT inventory_id from inventory
            where film_id=(SELECT film_id from film
            WHERE title =’Chamber Italian’))) returning *;
```

**returning ifadesi ile değişiklik yapılan satırları görmek mümkün.

** Başka ilişkili tablolar kullanılarak aynı sonucu verecek başka sorgular yazılabilir.

any

Bir değeri bir alt sorgu tarafından döndürülen bir dizi değer ile karşılaştırır.
Örnek: Aşağıdaki sorgu filmlerin uzunluklarını kategorilerine göre listeler.

```
SELECT MAX(length), category_id FROM film
            INNER JOIN film_category
            USING(film_id)
            GROUP BY category_id
            ORDER BY category_id;
```

Bu sorguyu, uzunlukları herhangi bir film kategorisinin maksimum uzunluğundan büyük veya ona eşit olan filmleri bulan aşağıdaki ifadede bir alt sorgu olarak kullanabilirsiniz.

```
SELECT title, length len FROM film
      WHERE length >= ANY
      ( SELECT MAX( length ) FROM film
            INNER JOIN film_category
            USING(film_id)
                  GROUP BY  category_id )
                        ORDER BY len;
```

Her kategori için alt sorgu maksimum uzunluğu bulur. Dış sorgu tüm bu değerlere bakar ve film tablosundaki hangi film uzunluklarının herhangi bir film kategorisinin maksimum uzunluğundan daha büyük veya ona eşit olduğunu belirler.

Not : =`ANY` operatörü `IN` operatörü ile eşdeğerdir.
`<> ANY` operatörü ile `NOT IN` operatörü birbirlerinden farklıdır.

```
SELECT title, length len FROM film
      WHERE length =
      ANY( SELECT MAX( length ) FROM film
            INNER JOIN film_category
            USING(film_id) GROUP BY  category_id )
                  ORDER BY len ;
```

Sorgusu ile aşağıdaki sorgu aynı sonucu verir.

```
SELECT title , length len FROM film
      WHERE length IN
      ( SELECT MAX( length ) FROM film
            INNER JOIN film_category
                  USING(film_id) GROUP BY  category_id )
                        ORDER BY len ;
```

7

Veri, Ekleme Güncelleme ve Silme işlemleri

(Modifing Data)

- Insert
- Update
- Update Join
- Delete

insert (Tabloda İstenilen Kolona Yeni Bir Satır Ekleme)

```
INSERT INTO tablo_adi (kolon1, kolon2, …) VALUES (değer1,
değer2, …);
```

Kolon1, kolon2 .. ifadeleri ekleme yapılacak sütunu ,

Değer1, değer2 .. ifadeleri bu sütunlara eklenecek değerleri gösterir.

Sütun ve değer listelerindeki sütunlar ve değerler aynı sırada olmalıdır.

Örnek: Aşağıdaki örneklerde film tablosunun kolonlarına değerler eklemek için sorgular çalıştırılmıştır.

```
insert into film (film_id, length, release_year, language_id)
values (10004, 115, 2015, 6) ;

ERROR:  null value in column "title" of relation "film" violates
not-null constraint
DETAIL:  Failing row contains (10004, null, null, 2015, 6, 3,
4.99, 115, 19.99, G, 2022-11-12 15:36:13.187935, null, ).
```

Sorgu sonucu ekranda dönen mesajda 'title' kolonunun not null constraint'e sahip olduğu ve boş geçilemeyeceği belirtilmiştir.

```
insert into film (film_id , title ,length , release_year ,
language_id) values (100 , 'film',115 , 2015 , 6) ;

ERROR:duplicate key value violates unique constraint "film_pkey"
DETAIL:  Key (film_id)=(100) already exists.
```

Sorgu sonucu ekranda dönen mesajda film_id kolonunun unique kısıtına sahip olduğu ve film_id = 100 değerinin daha önce kullanılmasından dolayı kullanılmayacağı belirtilmiştir.

```
insert into film (film_id , title ,length , release_year ,
language_id) values (10001 , 'film',115 , 2015 , 18) ;
```

Sorgu sonucu ekranda dönen mesajda '`language_id`' kolonunun '`language`' tablosunda referanslı bir foreign key kısıtına sahip olduğu belirtilmiştir. Bu sebeple bu kolona değer eklerken language tablosunda `language_id` değerleri kontrol edilmeli ve bu değerlerden biri seçilmelidir.

	language_id [PK] integer	name character (20)	last_update timestamp without time zone
1	1	English	2006-02-15 10:02:19
2	2	Italian	2006-02-15 10:02:19
3	3	Japanese	2006-02-15 10:02:19
4	4	Mandarin	2006-02-15 10:02:19
5	5	French	2006-02-15 10:02:19
6	6	German	2006-02-15 10:02:19

Tüm bu sorguları düzenleyip hatasız çalışmasını sağlarsak yeni sorgu şu şekilde olur.

```
insert into film (film_id , title ,length , release_year ,
language_id) values (10001 , 'film', 115 , 2015 , 2);
```

Tablodaki tarih türündeki (`date type`) bir kolona tarih değeri eklemek için tarih 'YYYY-AA-GG' ('YYYY-MM-DD') biçiminde kullanılır.

```
insert into rental (rental_date , inventory_id , customer_id ,
staff_id) values ('2005-02-15', 80, 25, 2) ;
```

Eklenen satırın tüm kolonlarını görüntülemek için `INSERT` komutunun sonuna `RETURNING` ifadesi eklenir.

```
insert into rental (rental_date , inventory_id , customer_id ,
staff_id) values('2005-02-15', 81, 25, 2) returning * ;
```

Eklenen satırdaki belli bir kolonu görüntülemek için ;

```
insert into rental (rental_date , inventory_id , customer_id ,
staff_id) values('2005-02-15',82,25,2) returning rental_id as id
;
```

update (Tablodaki Verileri Değiştirme)

```
UPDATE tablo_adi SET kolon1 = değer1, kolon2 = değer2,  ...
WHERE koşul;
```

`where` ifadesi opsiyoneldir. `where` ile koşul belirtilmezse değişiklik tablodaki tüm satırlarda yapılır.

```
UPDATE film SET title = 'Zoo Lander Fiction' WHERE film_id = 999;
```

`film_id = 999` olan satırda title kolonundaki değeri değiştirdi. Eğer `where` ile koşul belirtmeseydik tablodaki title kolonundaki tüm değerler 'Zoo Lander Fiction' değeri ile değişecekti.

```
UPDATE film SET release_year = 'İki bin On' WHERE film_id = 99 ;
```

Sorgu sonucu ekranda dönen hata mesajında kolonun belirtilmiş türü ile yeni verilen değerin türünün aynı olmadığı bildirilir.
**Veriler update edilirken kolonların türü dikkate alınmalıdır.

Birden çok satırda değişiklik yapmak için;

```
UPDATE film SET language_id=3  WHERE film_id BETWEEN 17 and 20 ;
```

Sorgu film_id değeri 17 ve 20 arasında olan satırlarda değişiklik yapar.

```
UPDATE film SET language_id=1 WHERE (film_id=17 or film_id=20) ;
```

Sorgu film_id değeri 17 veya 20 olan satırlarda değişiklik yapar. (Yani 2 satırda değişiklik yapılır.)

```
UPDATE film SET language_id=2 WHERE (film_id=17 and film_id=20) ;
```

Sorgu mantık olarak doğrudur çalışır ancak hiçbir satırda değişiklik yapılmaz bu yüzden UPDATE 0 değeri ekranda döner. Çünkü film_id değeri hem 17 hem de 20 olan bir satır yoktur.

Örnek:

```
UPDATE film SET language_id=2 WHERE title LIKE '%lone%'
returning *;
```

```
UPDATE film SET language_id=2 WHERE title LIKE 'alone%' ;
```

Sorgu çalışır ancak herhangi bir satırda değişiklik olmaz sorgunun doğru çalışması için aşağıdaki iki komut kullanılmalır.

```
UPDATE film SET language_id=2 WHERE title LIKE 'Alone%' ;
UPDATE film SET language_id=2 WHERE title ILIKE 'alone%' ;
UPDATE film SET language_id=2 WHERE title ILIKE 'Alone%' ;
```

update join (Bir Tablodaki Verileri Başka Tablodaki Verilere Göre Güncelleme)

`UPDATE` ifadesi `t1` tablosunun her satırı için `t2` tablosunun her satırını inceler.
Tablo `t1'in c2` sütunundaki değer,
tablo `t2'nin c2` sütunundaki değere eşitse,
`UPDATE` ifadesi tablo `t1'in c1` sütunundaki değeri yeni değer (yeni_değer) olarak günceller.

```
UPDATE t1 SET t1.c1 = yeni_değer FROM t2 WHERE t1.c2 = t2.c2;
```

Öncelikle `rental` tablosuna `odeme_miktarı` isimli kolon eklenmiştir. Daha sonra bu kolonda `payment` tablosundaki `amount` kolonu baz alınıp, her iki tabloda `rental_id` değerleri aynı olan satırlar update edilmiş, verileri girilmiştir.

```
alter table rental add  column odeme_miktari numeric ;

UPDATE
    rental r
SET
    odeme_miktari = amount + (amount * 0.1)
FROM
    payment p
WHERE
   p.rental_id=r.rental_id ;
```

delete (Bir Tablodan Satır Silme)

```
DELETE FROM tablo_adi;
```

Tablodaki bütün veriyi siler. Büyük tablolarda epey WAL üretir. Yani IO yapar.

`WHERE` koşulu kullanmak önemlidir.

```
DELETE FROM film_rating WHERE rating = 'G' returning * ;
```

Sorgu `film_rating` tablosundan rating kolonundaki değeri 'G' olan satırları siler ve sorgu sonucu silinen satırların çıktısı ekranda görünür.

Birden fazla satır silmek için aşağıdaki ifade kullanılır.

```
DELETE FROM tablo_adi WHERE koşul IN (değer1 , değer2) ;
DELETE FROM film_rating WHERE rating IN ('G' , 'R') returning *;
```

8

Transaction

- Begin
- commit
- Rollback

transaction

Bir veya daha fazla işlemden oluşan tek bir iş birimidir. Bir PostgreSQL işlemi atomik(atomic), tutarlı(consistent), izole edilmiş(isolated) ve kalıcıdır(durable).

-Atomik olması, işlemin ya hep ya hiç şeklinde tamamlanmasıdır (Bir sorun durumunda commit edilde ise yazılır commit edilemedi ise rollback edilir).

-Tutarlı olması, veritabanına yazılan verilerde yapılan değişikliğin geçerli olmasını ve önceden tanımlanmış kurallara uymasını garanti eder.

-İzole edilmesi, işlem bütünlüğünün diğer işlemlere nasıl görüneceğini belirler.
-Dayanıklı olması taahhüt edilen işlemlerin veri tabanında kalıcı olarak saklanmasını garanti eder.

ÖRNEK: Hesaplar adında bir tablo oluşturup veri girelim.

```
DROP TABLE IF EXISTS hesaplar;
CREATE TABLE hesaplar (
    id INT GENERATED BY DEFAULT AS IDENTITY,
    ad VARCHAR(100) NOT NULL,
    bakiye DEC(15,2) NOT NULL,
    PRIMARY KEY(id));
```

BEGIN TRANSACTION (İşlemi Başlatma)

```
INSERT INTO hesaplar(ad,bakiye) VALUES('ALİ',10000);
```

`id` için değer girmedik. `GENERATED BY DEFAULT AS IDENTITY` özelliğinden dolayı `id` değeri otomatik girildi. İstenirse girilebilir.
PostgreSQL, hesaplar tablosuna hemen yeni bir satır ekledi. Bu durumda, işlemin ne zaman başladığı bilinemez ve geri alma gibi değişiklik engellenemez.
Bir işlemi başlatmak için şu ifadeler kullanılır:

```
BEGIN TRANSACTION;
BEGIN WORK ;
BEGIN;
```

Örneğin; hesaplar tablosuna yeni bir satır ekleyelim ve yeni bir işlem başlatalım.

```
INSERT INTO hesaplar(id,ad,bakiye) VALUES( 2,'MERT',20000);
```

Şu an kullanmakta olduğumuz sessionda `SELECT * FROM hesaplar;` sorgusunu çalıştırdığımızda yeni eklediğimiz satırı görüyoruz. Ancak yeni bir session(oturum) başlatıp aynı sorguyu çalıştırırsak eklediğimiz 2.satırı göremeyiz. (kullanılan uygulamada autocommit açık ise otomaik olarak commit eder.)

commit

Değişikliğin diğer oturumlar (veya kullanıcılar) tarafından görülebilmesi için, COMMIT ifadesini kullanarak işlemi gerçekleştirmek gerekir:

```
COMMIT;
COMMIT WORK;
COMMIT TRANSACTION ;
```

`COMMIT` deyimini yürüttükten sonra PostgreSQL, bir çökme olursa değişikliğin kalıcı olacağını da garanti eder.

Tüm bu deyimleri bir arada kullanırsak:

```
-- İşlemi başlatmak (start transaction)
BEGIN;
-- tabloya yeni satır insert etmek
INSERT INTO hesaplar(ad,bakiye) VALUES('AHMET',3550);
-- değişikliği işlemek (commit etmek)
COMMIT;
```

Örnek

```
BEGIN;
UPDATE hesaplar SET bakiye = bakiye + 1200 WHERE id = 1;
UPDATE hesaplar SET bakiye = bakiye - 250 WHERE id = 2;
SELECT id, ad, bakiye FROM hesaplar;
```

**Mevcut sessionda değişiklik görülür başka session ile kontrol ettiğimizde göremeyiz.

commit ; komutun çalıştırılması ile tüm sessionlarda aynı sonuç görülür.

rollback

Geçerli işlemin değişikliğini geri almak için kullanılır.

```
ROLLBACK ;
ROLLBACK TRANSACTION ;
ROLLBACK WORK ;
```

Örnek: Hesaplara transfer gerçekleştirirken yanlış hesaba transfer yapmış olalım ve değişikliği iptal edelim**.**

```
BEGIN;
UPDATE hesaplar  SET bakiye = bakiye - 1500 WHERE id = 3;
UPDATE hesaplar SET bakiye = bakiye + 1500 WHERE id = 2;
ROLLBACK;
```

9

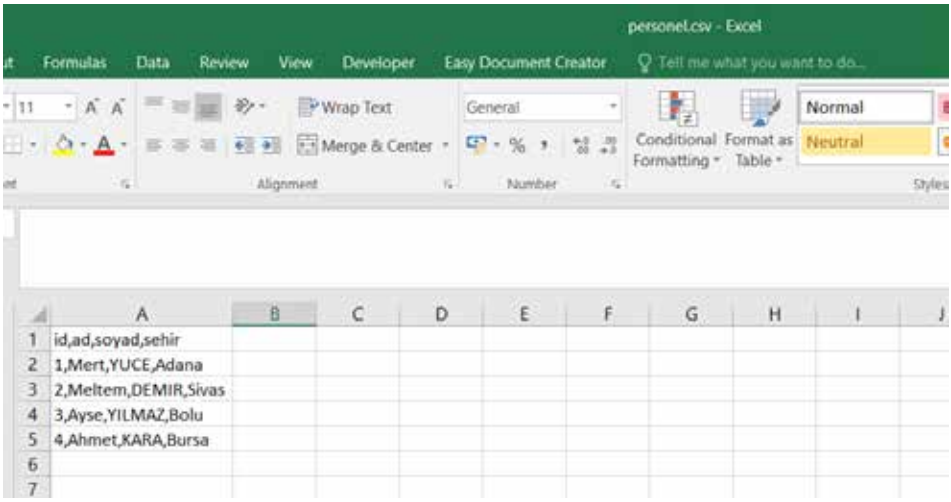
CSV İmport Export

CSV Formatındaki Dosyayı Postgresql'e Import

Comma Separated Values (CSV): (Virgül ile ayrılmış değerler dosyası). Bir veri listesi içeren düz metin dosyasıdır. Genellikle farklı uygulamalar arasında veri alışverişinde kullanılırlar. Veritabanları çoğunlukla CSV dosyalarını desteklemektedir.

Öncelikle veritabanımızda personel isimli tabloyu create ediyoruz.

```
CREATE TABLE personel (
  id INT,
  ad VARCHAR(50),
  soyad VARCHAR(50),
  sehir VARCHAR(255),
 PRIMARY KEY (id) ) ;
```

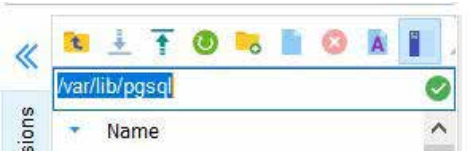


İkinci olarak belirtilen formatta bir CSV dosyası hazırlıyoruz.

Dosyayı windowsta Excel'de hazırladık. Linux'ta istediğimiz dizine almak için alternatifler şu şekildedir.

1. Alternatif

Postgresql dizininde pwd komutunu çalıştırıp bulunduğumuz full path'i kopyalıyoruz.



Mobaxterm programında kopyaladığımız dizini işaretlediğimiz yere yapıştırıp o dizinin altına gidiyoruz.

İşaretlediğimiz yere tıklayıp csv formatındaki dosyayı seçiyoruz ve böylelikle istediğimiz dizinin altına dosyayı kopyalamış oluyoruz.

Aşağıdaki komutları çalıştırıp gerekli izin ve sahiplikleri düzenliyoruz. (Dizinler düzenlenebilir.)

```
chown postgres:postgres /var/lib/pgsql/personel.csv
```

2. Alternatif

WinSCP programını kullanarak Windows tarafındaki dosyayı Linux tarafında istenilen dizine kopyalanabilir. Örneğin /var/lib/pgsql/ .

import

Dosyanın bulunduğu dizini kullanarak aşağıdaki komutu çalıştırıyoruz. Böylelikle csv dosyasında virgül (,) ile ayrılan değerleri tabloya import etmiş oluyoruz.

```
COPY personel
FROM '/var/lib/pgsql/personel.csv'
DELIMITER ','
CSV HEADER;

# Kontrol
SELECT * FROM personel ;
```

export

İlk olarak belirttiğimiz dizinde .csv uzantısı ile dosya oluşturuyoruz. İzin ve sahipliklerini tanımlıyoruz.

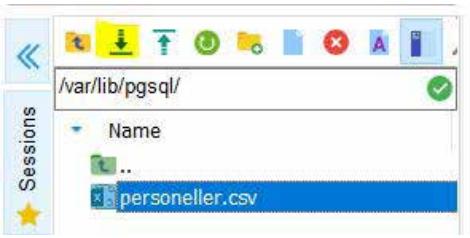
```
touch personeller.csv
chown postgres:postgres personeller.csv
```

İkinci olarak tablodaki verilerin ‘,’ formatı ile dizindeki dosyaya kopyalanması için aşağıdaki komutu çalıştırıyoruz.

```
COPY personel TO '/var/lib/pgsql/personeller.csv' DELIMITER ','
CSV HEADER;
```

**cat komutu ile dosyanın içini okuduğumuzda verileri görebiliriz.

Dosyayı windows tarafına almak için yine `WinSCP` programından ya da Mobaxterm programından faydalanabiliriz.



Mobaxtermde; windows tarafına almak istediğimiz dosyayı seçtikten sonra işaretlediğimiz butona tıklıyoruz.

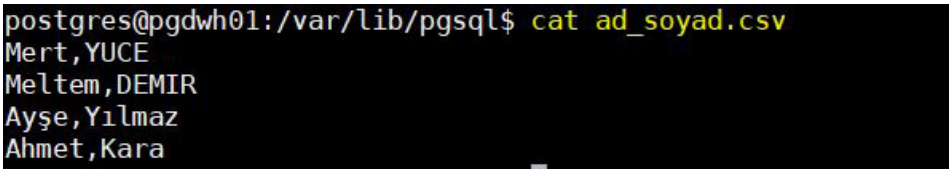
Tablonun Bazi Kolonlarini Export Etmek

```
COPY personel(id,ad,soyad) TO '/var/lib/pgsql/personeller.csv'
DELIMITER ',' CSV HEADER;
```

Tablonun Sütun Adlarini İçeren Başliği Export Etmemek

```
COPY personel(ad,soyad) TO '/var/lib/pgsql/ad_soyad.csv'
DELIMITER ',' CSV;
```

(`ad_soyad.csv` adlı dosyayı Linux'ta oluşturduk. Kopyalama işleminden sonra dosyayı windows tarafına alabilirsiniz.)



10

Tablo Yönetimi

(Managing Tables)

- Veri Türleri
- Tablo Oluşturma
- Constraint
- Create Tables AS
- Alter, Rename, Drop, Truncate Table
- Temporary
- Copy Table

Data Type (Veri Türleri)

Boolean : TRUE, FALSE veya bilinmeyen durumlarda kullanılır. NULL değerlerini alabilir.

Karakter : PostgreSQL üç karakter veri türü sağlar: CHAR(n), VARCHAR(n) ve TEXT. CHAR(n), boşluk dolgulu **sabit uzunluklu** karakterdir. Sütunun uzunluğundan daha kısa bir string ifade eklerseniz, PostgreSQL boşlukları doldurur. Sütunun uzunluğundan daha uzun bir string ifade eklerseniz, PostgreSQL bir hata verir.

VARCHAR(n) değişken uzunluklu karakter dizisidir. VARCHAR(n) ile n karaktere kadar saklayabilirsiniz. PostgreSQL, saklanan string sütunun uzunluğundan daha kısa olduğunda boşlukları doldurmaz.

TEXT, değişken uzunluktaki karakter dizisidir. Teorik olarak, metin verileri sınırsız uzunlukta bir karakter dizisidir.

Sayısal : PostgreSQL iki farklı sayısal (numeric) tür sağlar: Integer ve Floating-point number.

Integer : PostgreSQL'de üç tür integer vardır.

Small integer (**SMALLINT**); 2 baytlık işaretli tamsayıdır.

Integer (INT) : 4 baytlık işaretli bir tamsayıdır.

PostgreSQL'in değerleri otomatik olarak üretip `SERIAL` sütununa yerleştirmesi dışında

serial : integer ile aynıdır.

Floating-Point Number : PostgreSQL'de üç türü vardır.

float(n) : hassasiyeti en az n olan, en fazla 8 bayta kadar olan sayıdır.

real or float8, 4 baytlık sayıdır.

numeric veya numeric(p,s), ondalık noktadan sonra s ile p basamaklı gerçek bir sayıdır.

Geçici Veri Türleri (Temporal Data Type) Geçici veri türleri, tarih ve/veya saat verilerini saklamaya izin verir. PostgreSQL'in beş ana temporal data türü vardır.

`Date`, yalnızca tarihleri saklar.

`Time`, günün saati değerlerini saklar.

`Timestamp` hem tarih hem de saat değerlerini saklar.

`Timestampz`, saat dilimine duyarlı bir zaman damgası veri türüdür.

`Interval` zaman dilimlerini saklar.

`Timestampz`, PostgreSQL'in SQL standardının geçici veri türlerine uzantısıdır.

create table (Tablo oluşturma)

Yeni bir tablo oluşturmak için kullanılan syntax aşağıdaki gibidir.

```
CREATE TABLE [IF NOT EXISTS] tablo_adı (
    kolon1 datatype(length) kolon_kısıtı,
    kolon2 datatype(length) kolon_kısıtı,
    kolon3 datatype(length) kolon_kısıtı,
    tablo_kısıtı );
```

Kolon_kısıtı (column_contraint); kolonda depolanan verilerin uyması gereken kuralları belirtir.

constraint (Kısıtlamalar)

NOT NULL, bir sütundaki değerlerin NULL olmasına izin vermez yani boş bırakılamaz.
UNIQUE, aynı tablodaki satırlar arasında benzersiz bir sütundaki değerleri sağlar. Mükerrer kayda izin vermez.
PRIMARY KEY, bir tablodaki satırları benzersiz şekilde tanımlayan birincil anahtar (primary key) sütunu. Primary key kısıtlaması, bir tablonun birincil anahtarını tanımlamanıza olanak tanır.
CHECK, verilerin bir boolean ifadesini karşılamasını sağlar.
FOREIGN KEY, bir sütundaki veya bir tablodaki sütun grubundaki değerlerin başka bir tablodaki bir sütunda veya sütun grubunda var olmasını sağlar. Bir tablonun birçok foreign key'i olabilir.
Tablo kısıtlamaları, birden fazla sütuna uygulanmaları dışında sütun kısıtlamalarına benzer.

Örnekler : Birbiri ile ilişkili 3 tane tablo create edilmiştir.

```
CREATE TABLE hesaplar (
      user_id serial PRIMARY KEY,
      username VARCHAR ( 75 ) UNIQUE NOT NULL,
      password VARCHAR ( 75 ) NOT NULL,
      email VARCHAR ( 275 ) UNIQUE NOT NULL,
      created_on TIMESTAMP NOT NULL) ;

CREATE TABLE role(
   role_id serial PRIMARY KEY,
   role_name VARCHAR (275) UNIQUE NOT NULL );


CREATE TABLE hesap_role (
  user_id INT NOT NULL,
  role_id INT NOT NULL,
  PRIMARY KEY (user_id, role_id),
  FOREIGN KEY (role_id) REFERENCES role (role_id),
  FOREIGN KEY (user_id) REFERENCES hesaplar (user_id) ) ;
```

`hesap_role` tablosunun primary key'i iki sütundan oluşur: `user_id` ve `role_id`, bu nedenle primary key kısıtlamasını bir tablo kısıtlaması olarak tanımlamamız gerekti.

`user_id` sütunu, hesaplar tablosundaki `user_id` sütununa başvurduğundan, `user_id` sütunu için bir foreign key kısıtlaması tanımlamamız gerekti.

`role_id` sütunu, role tablosundaki role_id sütununa başvurduğundan, ayrıca `role_id` sütunu için bir foreign key kısıtlaması tanımlamamız gerekti.

create table as (CTAS)

`CREATE TABLE AS` deyimi yeni bir tablo oluşturur ve sorgulanan verilerle doldurur.

```
CREATE TABLE yeni_tablo_ismi AS query;
```

```
CREATE TABLE yeni_filmler AS SELECT film_id, title, release_year,
length, rating FROM film INNER JOIN film_category USING (film_id)
WHERE category_id = 1;
```

```
SELECT * FROM yeni_filmler ORDER BY title;
```

Kolonları görüntülemek için üstteki sorgu çalıştırılır.

```
CREATE TABLE IF NOT EXISTS film_reytingleri (rating, film_count)
AS SELECT rating, COUNT (film_id) FROM film GROUP BY rating;
```

Sorgu sonucu `film_reytingleri` adında 2 tane kolonu olan (rating , film_count) tablo oluşturulmuştur. Tablo verileri film tablosundan alınmıştır.

alter table

Mevcut bir tablonun yapısını değiştirmek için kullanılır.

```
ALTER TABLE tablo_adi action;
```

- Sütun ekleme
- Sütun silme
- Bir sütunun veri türünü (data type) değiştirme
- Bir sütunu yeniden adlandırma
- Sütun için varsayılan bir değer (default value) belirleme
- Bir sütuna bir kısıtlama (constrait) ekleme
- Bir tabloyu yeniden adlandırma

Sütun eklemek için `ALTER TABLE ADD COLUMN` ifadesi kullanılır.

```
ALTER TABLE tablo_adı  ADD COLUMN kolon_adı datatype kolon_
kısıtı;
```

Sütun silmek için `ALTER TABLE DROP COLUMN` ifadesi kullanılır.

```
ALTER TABLE tablo_adı  DROP COLUMN kolon_adı;
```

Bir sütunu yeniden adlandırmak için `ALTER TABLE RENAME COLUMN TO` ifadesi kullanılır.

```
ALTER TABLE tablo_adı RENAME COLUMN kolon_adı TO yeni_kolon_
adı;
```

Sütunun varsayılan değerini (default value) değiştirmek için `ALTER TABLE ALTER COLUMN SET DEFAULT` veya `DROP DEFAULT` öğesi kullanılır.

```
ALTER TABLE tablo_adı ALTER COLUMN kolon_adı  [SET DEFAULT
value | DROP DEFAULT];
```

`NOT NULL` kısıtlamasını değiştirmek için `ALTER TABLE ALTER COLUMN` ifadesi kullanılır.

```
ALTER TABLE tablo_adı ALTER COLUMN kolon_adı[SET NOT NULL| DROP
NOT NULL];
```

`CHECK` kısıtlaması eklemek için `ALTER TABLE ADD CHECK` deyimi kullanılır.

```
ALTER TABLE tablo_adı ADD CHECK expression;
```

Genel olarak, bir tabloya kısıtlama eklemek için `ALTER TABLE ADD CONSTRAINT` ifadesi kullanılır.

```
ALTER TABLE tablo_adı  ADD CONSTRAINT kısıt_adı kısıt_tanımı;
```

Bir tabloyu yeniden adlandırmak için `ALTER TABLE RENAME TO` ifadesi kullanılır.

```
ALTER TABLE tablo_adı  RENAME TO yeni_tablo_adı ;
```

Örnekler :

```
CREATE TABLE links (
   link_id serial PRIMARY KEY,
   title VARCHAR (518) NOT NULL,
   url VARCHAR (1034) NOT NULL ) ;
```

Aktif adında yeni bir sütun eklemek için aşağıdaki ifade kullanılır.

```
ALTER TABLE links ADD COLUMN aktif boolean;
```

Aşağıdaki ifade, aktif sütununu link tablosundan kaldırır.

```
ALTER TABLE links  DROP COLUMN aktif;
```

`title` adlı sütunun adını `link_title` olarak değiştirmek için aşağıdaki ifade kullanılır.

```
ALTER TABLE links  RENAME COLUMN title TO link_title;
```

Aşağıdaki ifade, links tablosuna target adında yeni bir sütun ekler.

```
ALTER TABLE links  ADD COLUMN target VARCHAR(10);
```

`links` tablosunda target sütunu için default değer olarak `_blank` ayarlamak için aşağıdaki ifadeyi kullanılır.

```
ALTER TABLE links  ALTER COLUMN target SET DEFAULT '_blank';
```

Yeni satırı, target sütunu için bir değer belirtmeden links tablosuna eklerseniz, target sütunu, _blank değerini varsayılan değer olarak alır.

```
INSERT INTO links (link_title, url)
VALUES('PostgreSQL Egitimi','https://www.postgresqlegitimi.
com/');
```

Data Output | Explain | Messages | Notifications

	link_id [PK] integer	link_title character varying (512)	url character varying (1024)	target character varying (10)
1	1	PostgreSQL Egitimi	https://www.postgresqlegitimi.com/	_blank

target sütunun kabul edeceği değerleri blank, parent, ve invalid ile kısıtlar.

```
ALTER TABLE links  ADD CHECK (target IN ('blank', 'parent',
'invalid'));
```

target sütunu için ayarlanan CHECK kısıtlamasını ihlal eden satır eklemeye çalışıldığında, aşağıdaki gibi bir hata verir.

```
INSERT INTO links (link_title, url, target) VALUES('PostgreSQL
Egitimi','https://www.postgresqlegitimleri.com/', 'db');
```

```
ERROR:  new row for relation "links" violates check constraint "links_target_check"
DETAIL:  Failing row contains (3, PostgreSQL Egitimi, https://www.postgresqlegitimi.com/, db).
SQL state: 23514
```

links tablosunun url sütununa `UNIQUE` kısıtlaması ekler.

```
ALTER TABLE links  ADD CONSTRAINT unique_url UNIQUE ( url );
```

Aşağıdaki ifade, zaten var olan url'yi eklemeye çalıştığı için hata döner.

```
INSERT INTO links(link_title,url)
VALUES('PostgreSQL','https://www.postgresqlegitimleri.com/');
```

```
ERROR:  duplicate key value violates unique constraint "unique_url"
DETAIL:  Key (url)=(https://www.postgresqlegitimleri.com/) already exists.
SQL state: 23505
```

rename table (tablo adı değiştirme)

Mevcut bir tabloyu yeniden adlandırmak için ALTER TABLE ifadesi aşağıdaki gibi kullanılır.

```
ALTER TABLE tablo_adi RENAME TO yeni_tablo_adi;
ALTER TABLE IF EXISTS tablo_adi RENAME TO yeni_tablo_adi;
```

Örnekler :

```
CREATE TABLE sellers (
    id serial PRIMARY KEY,
    name VARCHAR NOT NULL ) ;
```

`sellers` tablosunu suppliers olarak yeniden adlandırmak için aşağıdaki `ALTER TABLE RENAME TO` ifadesini kullanılır.

```
ALTER TABLE sellers RENAME TO suppliers;
```

`Suppliers` tablosuyla ilişkili tablo oluşturalım.

```
CREATE TABLE supplier_groups (id serial PRIMARY KEY, name
VARCHAR NOT NULL);
```

`suppliers` tablosuna `group_id` adlı yeni bir sütun ekleyelim. Bu sütun, `suppliers_group` tablosunun id sütununa bağlanan foreign key sütunudur.

```
ALTER TABLE suppliers ADD COLUMN group_id INT NOT NULL;
ALTER TABLE suppliers ADD FOREIGN KEY (group_id) REFERENCES
supplier_groups(id);
```

Suppliers tablosunu aşağıdaki gibi tanımlayarak suppliers tablosundaki foreign key kısıtlamasını doğrulayabiliriz.

```
ALTER TABLE supplier_groups RENAME TO groups;
```

```
dvdrental=# \d suppliers
                                  Table "public.suppliers"
  Column  |       Type        | Collation | Nullable |                Default
----------+-------------------+-----------+----------+---------------------------------------
 id       | integer           |           | not null | nextval('vendors_id_seq'::regclass)
 name     | character varying |           | not null |
 group_id | integer           |           | not null |
Indexes:
    "vendors_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "suppliers_group_id_fkey" FOREIGN KEY (group_id) REFERENCES supplier_groups(id)

dvdrental=# \d suppliers
                                  Table "public.suppliers"
  Column  |       Type        | Collation | Nullable |                Default
----------+-------------------+-----------+----------+---------------------------------------
 id       | integer           |           | not null | nextval('vendors_id_seq'::regclass)
 name     | character varying |           | not null |
 group_id | integer           |           | not null |
Indexes:
    "vendors_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "suppliers_group_id_fkey" FOREIGN KEY (group_id) REFERENCES groups(id)

dvdrental=#
```

`drop table` (tablo silme)

```
DROP TABLE [IF EXISTS] tablo_adı  [CASCADE | RESTRICT];
```

Drop etmek istenilen tablo, views, triggers, functions, ve stored procedures gibi diğer nesnelerde kullanılıyorsa, DROP TABLE ifadesi tabloyu silemez, hata alır. Bu durumda iki seçenek vardır:

1-`CASCADE` seçeneği, tabloyu ve ona bağlı nesneleri kaldırmanıza olanak tanır.

2-`RESTRICT` seçeneği, tabloya bağlı herhangi bir nesne varsa tabloyu reddeder. DROP TABLE deyiminde açıkça belirtmezseniz, RESTRICT seçeneği varsayılandır.

Birden çok tabloyu aynı anda silmek için;

```
DROP TABLE [IF EXISTS]  tablo_adı1,  tablo_adı2,  ... [CASCADE
| RESTRICT];
```

NOT: Tabloları drop etmek için superuser, schema, owner veya table owner rollerine sahip olmak gerekmektedir.

Örnekler :

```
CREATE TABLE book_author (
      authors_id INT PRIMARY KEY,
      name VARCHAR (50));
```

```
CREATE TABLE book_pages (
      pages_id serial PRIMARY KEY,
      title VARCHAR (255) NOT NULL,
      authors_id INT NOT NULL,
      FOREIGN KEY (authors_id)REFERENCES book_author (authors_
id) );
```

`book_author` tablosunu silmek için ;

```
DROP TABLE IF EXISTS book_author;
```

```
ERROR:  cannot drop table book_author because other objects depend on it
DETAIL:  constraint book_pages_author_id_fkey on table book_pages depends on table book_author
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
SQL state: 2BP01
```

`DROP TABLE` ifadesi, drop edilen tablonun bağımlı nesnelerini kaldırırsa, aşağıdaki gibi bir mesaj yayınlar.

```
DROP TABLE book_author CASCADE;
```

```
NOTICE:  drop cascades to constraint book_pages_author_id_fkey on table book_pages
DROP TABLE

Query returned successfully in 111 msec.
```

`truncate table` (tablodaki tüm verileri siler)

Tablo içerisindeki tüm veriyi çok hızlı siler. Transacation log yani WAL üretmez. Geri dönüşü yoktur. `where` filtresi kullanılamaz.

```
TRUNCATE TABLE tablo_adi ;
```

Büyük bir tablo silinecekse (drop), önce içerisindeki veri truncate edilip sonra tablo drop edilirse hem sistemi yormaz hem de daha hızlı olur.

Birden fazla tablodan tüm verileri silmek için ;

```
TRUNCATE TABLE  tablo_adi1,  tablo_adi2,  ...;
```

Bir tablodan ve tabloya foreign key referansı olan diğer tablolardan veri silmek için `TRUNCATE TABLE` ifadesinde `CASCADE` seçeneği kullanılır.

```
TRUNCATE TABLE tablo_adi CASCADE;
```

`create temporary table` (geçici tablo oluşturma)

Geçici bir tablo bir veritabanı oturumu süresince var olan kısa ömürlü bir tablodur. Bir oturumun veya işlemin sonunda geçici tablolar otomatik olarak drop edilir.

```
CREATE TEMPORARY TABLE temp_table_name(kolon_listesi );
```

**Temporary bir tablo yalnızca onu oluşturan oturum tarafından görülebilir. Başka bir deyişle, diğer oturumlara görünmez.
PostgreSQL özel bir şemada temporary tablolar oluşturur, bu nedenle CREATE TEMP TABLE ifadesinde şemayı belirtemezsiniz.
**Aynı isimle kalıcı ve geçici (temporary) tablo oluşturulursa oturum boyunca temporary tablo listelenir.

```
CREATE TABLE musteri (id SERIAL PRIMARY KEY, name VARCHAR NOT
NULL);
CREATE TEMP TABLE musteri (customer_id INT );

Select * from musteri;
```

Şekilde görüldüğü gibi temp table kolonu sonuç olarak geldi.

Temporary tabloyu silmek için;

```
DROP TABLE geçici_tablo_ismi
```

`copy table` (tablo kopyalama)

Bir tabloyu hem tablo yapısı hem de verileri dahil olmak üzere tamamen kopyalamak için aşağıdaki ifade kullanılır.

```
CREATE TABLE yeni_tablo_adi AS  TABLE kopyalanacak_tablo_adi;
```

Veri içermeyen tablo yapısını kopyalamak için ,

```
CREATE TABLE yeni_tablo_adi AS  TABLE kopyalanacak_tablo_adi
WITH NO DATA;
```

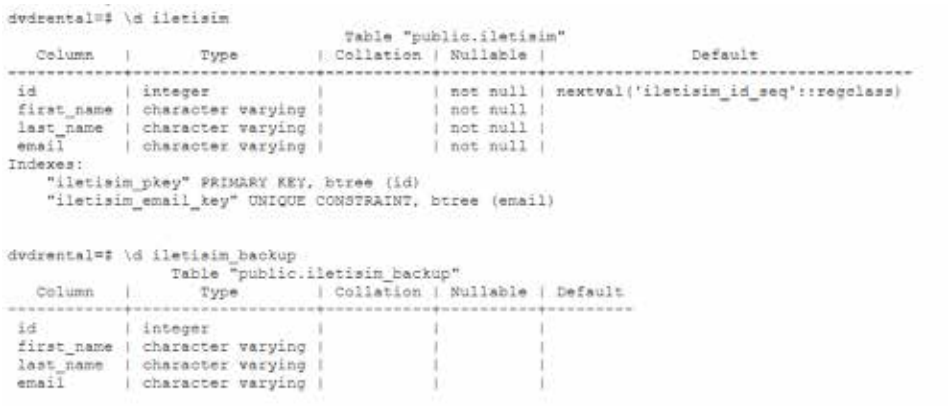
Var olan bir tablodan verilerin bir kısmını kopyalayıp yeni tablo oluşturmak için;

```
CREATE TABLE yeni_tablo_adi AS  SELECT * FROM kopyalanacak_
tablo_adi WHERE koşul;
```

Örnekler :

```
CREATE TABLE iletisim(
    id SERIAL PRIMARY KEY,
    first_name VARCHAR NOT NULL,
    last_name VARCHAR NOT NULL,
    email VARCHAR NOT NULL UNIQUE );

INSERT INTO iletisim(first_name, last_name, email)
VALUES('Ali','Yılmaz','ayılmaz@postgres.com'),
('Davut','Yıldırım','davudyıl @postgres.com');
CREATE TABLE iletisim_backup AS TABLE iletisim;
```



```
dvdrental=$ \d iletisim
                                  Table "public.iletisim"
   Column   |       Type        | Collation | Nullable |                Default
------------+-------------------+-----------+----------+---------------------------------------
 id         | integer           |           | not null | nextval('iletisim_id_seq'::regclass)
 first_name | character varying |           | not null |
 last_name  | character varying |           | not null |
 email      | character varying |           | not null |
Indexes:
    "iletisim_pkey" PRIMARY KEY, btree (id)
    "iletisim_email_key" UNIQUE CONSTRAINT, btree (email)


dvdrental=$ \d iletisim_backup
                Table "public.iletisim_backup"
   Column   |       Type        | Collation | Nullable | Default
------------+-------------------+-----------+----------+---------
 id         | integer           |           |          |
 first_name | character varying |           |          |
 last_name  | character varying |           |          |
 email      | character varying |           |          |
```

Resimde görüldüğü üzere `iletisim_backup` tablosunun yapısı indexler hariç iletisim tablosu ile aynıdır.

`iletişim_backup` tablosuna primary key ve `UNIQUE` kısıtlamaları eklemek için aşağıdaki `ALTER TABLE` ifadelerini çalıştırılır.

```
ALTER TABLE iletisim_backup ADD PRIMARY KEY(id);
ALTER TABLE iletisim_backup ADD UNIQUE(email);
```

11

Kısıtlamalar

(Constraints)

- Primary Key
- Foreign Key
- Check Constraint
- Unique Constraint
- Not-Null Constraint

Primary Key

Primary key, bir tablodaki bir satırı benzersiz bir şekilde tanımlamak için kullanılan bir sütun veya bir sütun grubudur. Teknik olarak, NULL kısıtlaması ile UNIQUE kısıtlamasının birleşimidir.

```
CREATE TABLE tablo_adı ( kolon1 data_type PRIMARY KEY, kolon2
data_type, …) ;
```

Primary key'in iki veya daha fazla sütundan oluşması durumunda, primary key kısıtlaması aşağıdaki gibi tanımlanır.

```
CREATE TABLE tablo_adı ( kolon1 data_type, kolon2 data_type, …
PRIMARY KEY (kolon1,kolon2) );

ALTER TABLE tablo_adı ADD PRIMARY KEY (kolon1 , kolon2);
```

İsim belirtilmezse sistem default bir isim atar.

```
CONSTRAINT kısıt_adı PRIMARY KEY(kolon1, kolon2,...);
```

constraint in silinmesi

```
ALTER TABLE tablo_adı DROP CONSTRAINT primary_key_constraint;
```

Foreign Key

Foreign key, bir tablodaki başka bir tablonun primary key kısıtına başvuran bir sütun veya sütun grubudur. Foreign key kısıtlaması, alt ve üst tablolar arasındaki verilerin başvurusal bütünlüğünü korur. Foreign key kısıtlaması, alt tablodaki bir sütundaki veya bir sütun grubundaki değerlerin, ana tablonun bir sütunundaki veya bir sütun grubundaki değerlere eşit olduğunu göstermektedir.

Bir tablonun diğer tablolarla olan ilişkilerine bağlı olarak birden fazla foreign key kısıtı olabilir.

```
CREATE TABLE musteriler(musteri_id INT GENERATED ALWAYS
AS IDENTITY,  musteri_ad VARCHAR(255) NOT NULL,  PRIMARY
KEY(musteri_id) );

CREATE TABLE iletisim_bilgileri( irtibat_id INT GENERATED
ALWAYS AS IDENTITY, musteri_id INT, irtibat_ad VARCHAR(255) NOT
NULL,  tel VARCHAR(15), mail VARCHAR(100), PRIMARY KEY(irtibat_
id), CONSTRAINT fk_musteri FOREIGN KEY(musteri_id) REFERENCES
musteriler(musteri_id) );
```

Check Constraint

`CHECK` kısıtlaması, bir sütundaki değerlerin belirli bir gereksinimi karşılaması gerekip gerekmediğini belirtmeye olanak tanıyan bir tür kısıtlamadır. Değerler `CHECK` kısıtından geçerse, PostgreSQL bu değerleri sütuna ekler veya günceller. Aksi takdirde, PostgreSQL değişiklikleri reddedecek ve bir kısıtlama ihlali hatası verecektir.

```
CREATE TABLE isciler (id SERIAL PRIMARY KEY, ad VARCHAR (50),
soyad VARCHAR (50), dogum_tarihi DATE CHECK (dogum_tarihi >
'1960-01-01'), baslama_tarihi DATE CHECK (baslama_tarihi >
dogum_tarihi), maas INT CHECK (maas > 500))


INSERT INTO isciler(ad, soyad, dogum_tarihi, baslama_tarihi,
maas) VALUES ('Ayse', 'Çiçek', '1972-01-01', '2015-07-01',
250);
```

`maas > 500 koşuluna` uymadığı için son sorgu hata verir.

Unique Constraint

UNIQUE kısıtlaması uygulandığında, her yeni satır eklediğinizde, değerin zaten tabloda olup olmadığını kontrol eder. Değer kolonda mevcutsa değişikliği reddeder ve bir hata verir. Aynı işlem mevcut verilerin güncellenmesi için de gerçekleştirilir.

```
CREATE TABLE tablo_adı (
 c1 data_type,
 c2 data_type,
 c3 data_type,
 UNIQUE (c2, c3) );
```

Örnek

```
CREATE TABLE ekipmanlar (id SERIAL PRIMARY KEY, name VARCHAR
(50) NOT NULL, ekipman_id VARCHAR (16) NOT NUL) );

CREATE UNIQUE INDEX CONCURRENTLY ekipman_ekip_id  ON ekipmanlar
(ekipman_id);
```

```
ALTER TABLE ekipmanlar  ADD CONSTRAINT unique_ekipman_id UNIQUE
USING INDEX ekipman_ekip_id;
```

Komut ekipmanlar tablosuna `ekipman_ekip_id` indexini kullanarak unique kısıtı ekler.

Not-Null Constraint

Veritabanı teorisinde `NULL`, bilinmeyeni veya eksik bilgiyi temsil eder. `NULL`, boş bir dize veya sıfır sayısı ile aynı değildir. Bu kısıt sayesinde kısıtın eklendiği sütundaki herhangi bir satır boş geçilemez. Mutlaka değer girilmelidir.

```
CREATE TABLE tablo_adi( kolon_adi data_type NOT NULL , ....);
```

Var olan kolona `NOT NULL` kısıtı eklemek için;

```
ALTER TABLE tablo_adi ALTER COLUMN kolon_adi SET NOT NULL;
```

12

Matematik İşlemleri Tarih Zaman Fonsksiyonları

(Mathematical Operations – Date Time Functions)

count

Parantez içinde verilen değere sahip kaç tane satır olduğunu sonuç olarak döndüren fonksiyondur. Veritabanındaki kayıtları sayar.

Bir tabloda kaç tane satır olduğunu döndüren sorgu:

```
select count(*) from film;
```

`WHERE` koşulu ile kullanımı;

```
SELECT COUNT(*) from city where country_id=2 ;

SELECT COUNT(*) as toplam from payment
      where amount > 8.99 ;

SELECT COUNT (*) sonuc FROM rental
      WHERE  staff_id = 1
      AND
      customer_id = 3 ;

SELECT COUNT (*) as sonuc FROM rental
      WHERE  staff_id = 1
      OR
      customer_id = 3 ;
```

```
SELECT staff_id , COUNT (payment_id)  FROM payment
      GROUP BY staff_id;
```

Sorgu payment tablosundaki `staff_id` değerlerini bulur ve gruplar.
Bu değerlere sahip olan satır sayısını hesaplar.
Aşağıdaki sorgu ile aynı sonucu verir çünkü `payment_id` sütununde mükerrer kayıt yoktur (`unique`).

```
SELECT staff_id , COUNT (*)  FROM payment GROUP BY staff_id;
```

İade tarihi (return_date) 2005-05-26 ile 2005-05-30 arasında kaç tane film olduğu sonucunu veren sorgu:

```
SELECT COUNT (*) sonuc FROM rental  WHERE  return_date BETWEEN
'2005-05-26' and '2005-05-30'  ;
```

sum

Belirtilen alandaki değerleri toplayan fonksiyondur.

Payment tablosundaki ödemelerin toplamını veren sorgu :

```
SELECT SUM(amount) FROM  payment ;
```

`WHERE` koşulu ile kullanımı:

```
SELECT SUM(amount) FROM  payment
      where customer_id = 341
      and
      staff_id = 2;

SELECT SUM(amount) FROM  payment
      where customer_id = 341
      OR
      staff_id = 2;

SELECT SUM(amount) FROM  payment
      WHERE customer_id NOT IN (341, 342, 343, 344) ;

SELECT SUM(amount) as toplam  FROM  payment
      WHERE customer_id NOT IN
            ( SELECT customer_id from payment
                  where customer_id BETWEEN 341 and 402) ;
```

Alt sorguda değeri 341 ile 402 arasında olan `customer_id` değerleri bulunur.Bu değerlerin dışındaki (`not in`) `customer_id` değerlerine ait amount değerleri toplanır.

```
SELECT SUM (amount) as Toplam FROM payment WHERE staff_id!=1 ;
SELECT SUM (amount) as Toplam FROM payment WHERE staff_id!=2 ;
```

1.sorgu staff_id değeri 1 olmayan

2. Sorgu ise staff_id değeri 2 olmayan satırların amount değerlerini toplar.

avg

Belirtilen alandaki değerlerin ortalamasını alan fonksiyondur.

```
SELECT AVG(amount) ortalama FROM payment ;

SELECT AVG(amount) ortalama FROM payment
     WHERE payment_id BETWEEN 22430 and 22436 ;

SELECT staff_id, AVG(amount) FROM payment GROUP BY staff_id ;
```

max-min

Belirtilen alandaki en yüksek (MAX) veya en düşük (MIN) değeri bulmamıza yarayan fonksiyondur.

```
SELECT MAX(amount) - MIN(amount) as fark from payment ;

SELECT rental_id, MIN(amount) FROM payment
      GROUP BY rental_id
      ORDER BY rental_id DESC;
```

abs

Sayısal değeri negatiflikten kurtaran fonksiyondur.

```
SELECT abs (-5) ;
```

ceil

Ondalıklı değeri bir üst sayıya yuvarlayan fonksiyondur.

```
SELECT ceil(amount), payment_id FROM payment WHERE amount > 5 ;
```

floor

Ondalıklı değeri bir alt sayıya yuvarlayan fonksiyondur.

```
SELECT floor(amount), payment_id, customer_id FROM payment
      WHERE amount > 5 ;
```

tarih – zaman fonksiyonları

Mevcut tarihi öğrenmek için ;

```
SELECT Current_Date;
```

Mevcut saati öğrenmek için ;

```
SELECT Current_Time;
```

Mevcut zamanı öğrenmek için ;

```
SELECT now();
```

Bugünün tarihi ile girdiğimiz tarih arasında kaç yıl ay gün fark olduğunu bulmak için ;

```
SELECT age(Timestamp '2019-12-01') ;
```

Örnek Tablodan tarih kolonundaki değerler ile bugünün tarihi arasındaki farkı bulmak için ;

```
SELECT rental_date, customer_id, age(now(), rental_date) from
rental ;
```

13

psql Komutları

Terminal ekranından veritabanına bağlanmak için;

```
psql -d veritabanı_adı -U kullanıcı_adı -W
Psql -d dvdrental -U postgres -W
```

Başka bir makinede bulunan veritabanına bağlanmak için;

```
psql -h host -d veritabanı_adı -U kuallnıcı_adı -W
```

Bir veritabanına bağlandığınızda, bağlantıyı, -kullanıcı tarafından belirtilen- bir kullanıcı altında yeni bir veritabanına değiştirebilirsiniz. Önceki bağlantı kapatılır. Kullanıcı parametresini atlarsanız, geçerli kullanıcı olduğu varsayılır.

```
\c vt_adı kullanıcı_adı
\c dvdrental
```

Veritabanı isimlerini listelemek için;

```
\l
```

Tablo isimlerini listelemek için;

```
\dt
```

Bağlı olunan veritabanının tüm şemalarını listelemek için;

```
\dn
```

Mevcut veritabanının viewlarını listelemek için;

```
\dv
```

Tüm kullanıcıları ve atama rollerini listelemek için;

```
\du
```

PostgreSQL'in versiyonunu öğrenmek için;

```
SELECT version();
```

Bir önceki komutu tekrar çalıştırmak için;

```
\g
```

Komut geçmişini görmek için;

```
\s
```

Komut geçmişini bir dosyaya kaydetmek için;

```
\s dosya_adi
```

Bir dosyadan psql komutlarını çalıştırmak için;

```
\i dosya_adı
```

Mevcut tüm psql komutlarını görmek için;

```
\?
```

Belirli bir PostgreSQL deyimi hakkında yardım almak için \h komutunu kullanırsınız. Örneğin ALTER TABLE deyimi hakkında detaylı bilgi almak istiyorsanız aşağıdaki komutu kullanırsınız.

```
\h ALTER TABLE
```

Sorgu execution süresini açmak için;

```
\timing
```

Aynı komutu tekrar çalıştırdığınızda süre kapanır.

Komutu editörde düzenlemek için;

```
\e
```

Komutu verdikten sonra psql, EDITOR ortam değişkeniniz tarafından tanımlanan metin düzenleyiciyi açar ve psql'e girdiğiniz en son komutu düzenleyiciye yerleştirir. Komutu editöre yazıp kaydedip editörü kapattıktan sonra, psql komutu yürütecek ve sonucu döndürecektir.

psql, bazı çıktı biçimi türlerini destekler ve çıktının anında nasıl biçimlendirileceğini özelleştirmenize olanak tanır.

```
\a komut
```

Hizalanmış sütun çıktısından hizalanmamış sütun çıktısına geçiş yapar.

Çıktıyı HTML biçiminde biçimlendirmek için;

```
\H
```

psql'den çıkmak için;

```
\q
```

Sorgu sonucunu HTML çıktısı olarak almak için alternatif olarak aşağıdaki komutlar kullanılabilir.

-Linux'te .html uzantılı dosya oluşturulur.

```
touch test.html
```

psql'de sırayla aşağıdaki komutlar çalıştırılır.

```
\H
```

Sonucun kaydedileceği dosyanın pathi yazılır.)

```
\o /data/postgresql/test.html
```

Sonucunun kaydedilmesini istediğiniz sorgu çalıştırılır.

```
SELECT rental_id, MIN(amount) FROM payment GROUP BY rental_id
ORDER BY rental_id DESC;
```

html uzantılı dosyayı Bölüm 9'da anlatıldığı gibi Windows ortamına kopyalanabilir.

14

Örnek Yöntemler

(SQL Recipes)

İki Tabloyu Karşılaştırmak

Aralarındaki farkları bulmak için iki tablonun içeriğini karşılaştırmanın birkaç yolu vardır.

Örnek: Karşılaştırılacak 2 tablo oluşturulup örnek veri gerişi yapılır.

```
CREATE TABLE tab1 (
      ID INT PRIMARY KEY,
      AD VARCHAR (50) );

INSERT INTO tab1 (ID, AD)
VALUES
      (1, 'd'),
      (2, 'b');
```

```
CREATE TABLE tab2 (
      ID INT PRIMARY KEY,
      AD VARCHAR (50) );

INSERT INTO tab2 (ID, AD)
VALUES
       (1, 'a'),
       (2, 'c');
```

```
SELECT
      ID,
      AD,
      'tab2de yok' AS note
FROM tab1
EXCEPT
SELECT
      ID,
      AD,
      'tab2de yok.' AS note
FROM tab2 ;
```

Yinelenen Satırları Silme

Örnek:

```
CREATE TABLE dersler(
    id SERIAL PRIMARY KEY,
    ders VARCHAR(50) NOT NULL);

INSERT INTO dersler(ders) values('ÜRETİM PLANLAMA');
INSERT INTO dersler(ders) values('BENZETİM');
INSERT INTO dersler(ders) values('MATEMATİK');
INSERT INTO dersler(ders) values('MATEMATİK');
INSERT INTO dersler(ders) values('BENZETİM');
INSERT INTO dersler(ders) values('YÖNEYLEM');
```

Aynı insert komutlarını birkaç kez çalıştırıp yinelenen satır oluşturduk.

Yinelenen satırları görüntelemek için;

```
SELECT
    ders,
    COUNT( ders )
FROM dersler
GROUP BY ders
HAVING
    COUNT( ders )> 1
ORDER BY ders;
```

Yinelenen satırları silmek için;

```
DELETE FROM dersler d USING dersler b
WHERE
    d.id < b.id AND d.ders = b.ders;
```

Sorguda `where` koşulu ile ders değerleri aynı fakat id değerleri farklı olan satırlar seçilir.

Yeni bir tablo oluşturarak yinelenen satırları silmek için;

```
-- Adım 1
CREATE TABLE dersler_temp (LIKE dersler);

-- Adım 2
INSERT INTO dersler_temp(ders, id)
SELECT
    DISTINCT ON (ders) ders,
    id
FROM dersler;
(select * from dersler_temp  )

-- Adım 3
DROP TABLE dersler;

-- Adım 4
ALTER TABLE dersler_temp
RENAME TO dersler ;
```

PLpgSQL

1

PLpgSQL'e Giriş

- String Sabitleri
- Block Yapısı

PL pgSQL'e Giriş

PostgreSQL veritabanı için prosedürel programlama dilidir. Veritabanı sunucusunun işlevselliğini artırır.
Bazı Özellikleri :
- Kullanıcı tanımlı fonksiyon (user-defined functions), prosedür ve trigger oluşturulabilir.
- `If`, `case` ve `loop` deyimleri gibi kontrol yapıları ekleyerek standart SQL'i genişletmek
PLpgSQL, SQL veritabanından kolayca verileri sorgulamanıza olanak sağlayar.
PostgreSQL, SQL deyimlerini tek tek çalıştırabilir. Bu duruma çözüm olarak PostgreSQL PLpgSQL kullanır. PLpgSQL birden çok ifadeyi tek bir nesne olarak PostgreSQL veritabanı sunucusunda depolar.

Dolar ($$) Alıntılı String Sabitleri

PostgreSQL string sabiti için `SELECT` deyiminden sonra tek tırnak kullanarak ekrana istediğiniz mesajınızı dönebilirsiniz.

```
SELECT 'HELLO WORLD'
SELECT 'TÜRKİYE''NİN KUZEYİ'
SELECT 'DÜNYA''NIN UYDUSU'
```

`$$` işaretleri ile de ekrana mesaj dönebilirsiniz.

```
SELECT $$TÜRKİYE'NİN ŞEHİRLERİ $$ ;
SELECT $message$TÜRKİYE'NİN GÜNEYİ $message$ as mesaj ;
```

PLpgSQL Blok Yapısı

PL/pgSQL, blok yapılı bir dildir. Bu nedenle PL/pgSQL fonksiyonları veya prosedürleri bloklar halinde düzenlenir. Her bloğun iki bölümü vardır: bildirim(declaration) ve gövde(body). Gövde bölümü gereklidir, bildirim bölümü ise isteğe bağlıdır.
Bir blok, END sözcüğünden sonra noktalı virgül (;) ile sonlandırılır.
Bir bloğun başında ve sonunda bulunan isteğe bağlı bir etiket olabilir. Blok etiketini, blok gövdesinin EXIT deyiminde belirtmek istediğinizde veya blokta bildirilen değişkenlerin adlarını nitelemek istediğinizde kullanabilirsiniz.
Bildirim(declaration) bölümü, gövde bölümünde kullanılan tüm değişkenleri bildirdiğiniz yerdir. Bildirim bölümündeki her ifade noktalı virgül (;) ile sonlandırılır.
Gövde bölümü, kodu yerleştirdiğiniz yerdir. Gövde bölümündeki her ifade de noktalı virgül (;) ile sonlandırılır.
ÖRNEK: Basit bir blok yapısı örneğidir. Anonim blok olarak adlandırılır. Sonuç olarak kayıtlı toplam film sayısı belirttiğimiz mesaj ile döner.

```
do $$
<<ilk_blok>>
declare
  film_count integer := 0;
begin
   -- film sayısını almak
   select count(*) into film_count from film;
   -- Ekrana yazılacak mesaj
   raise notice 'Toplam Film Sayısı : %', film_count;
end ilk_blok $$;
```

`do` ifadesi bloğa ait değildir. Anonim blok çalıştırmak (execute) için kullanılır. Örneğin daha okunabilir hale getirmek için `dolar($$)` ile syntax kullanılmıştır.

Blokların incelenmesi

```
--film_count değişkeni tanımlandı ve değerine sıfır atandı.

film_count integer := 0;

select count(*) into film_count from film;

Gövde bölümünde film tablosundan film sayısını alacak ve sonucu
film_count değişkenine atayacak şekilde count(*) işleviyle
SELECT INTO ifadesini kullanıldı.


raise notice 'Toplam Film Sayısı : %', film_count;

Ekrana dönmesi istenilen mesajı raise notice yapısı ile
kullanıldı. Mesaj sonuna film_count değeri eklenildi.
%, film_count değişkeninin içeriğiyle değişen bir yer tutucudur.
```

`ilk_blok` etiketi yalnızca tanıtım amaçlıdır.

Örnek

```
do $$
declare
   aktör_sayısı integer;
begin
   -- Actor tablosundan toplam aktör sayısını almak
   select count(*)into aktör_sayısı from actor;
   -- Aktör sayısını ekranda göstermek
   raise notice 'Toplam aktör sayısı: %', aktör_sayısı;
end; $$
```

2

Değişkenler ve Sabitler

(Variables & Constants)

- Değişkenler
- Veri Türlerini Kopyalama
- Subblock
- Pl/Pgsql Satır Türleri
- Kayıt Türü
- Sabitler

Değişkenler

Değişken blok aracılığıyla değiştirilebilen bir değeri tutar. Her zaman belirli bir veri türüyle ilişkilendirilir.
Bir değişken kullanılmadan önce blokta `declaration` (bildirim) bölümünde tanımlanmalıdır.

```
değişken_adı veri_türü [:= expression];
```

İsteğe bağlı olarak değişkene varsayılan bir değer atanır. Değer atanmazsa değişkenin başlangıç değeri NULL 'dur.

Örnek:

```
do $$
declare
   sayac integer := 1;
   ad varchar(50) := 'Can';
   soyad varchar(50) := 'Doğu';
   ödeme numeric(11,2) := 27.8;
begin
   raise notice '% - % % yaptığı ödeme miktarı % TL',
          sayac,
         ad, soyad,
         ödeme;
end $$;
```

sayac değişkeni, 1 olarak başlatılan integer(tam sayı) türünde değişkendir.
ad ve soyad varchar(50) veri türüne sahip ve 'Can' ve 'Doğu' string sabitleriyle başlatılan değişkenlerdir.
Ödeme veri türü sayısal olan ve değeri 27.8 olarak başlatılan değişkendir.

copying types

%type ifadesi ile yapılır. Bir tablo sütunundan veya başka bir değişkenin veri türünü kopyalar. Referansta bulunduğunuz sütunun türünü bilmenize gerek kalmaz. Başvurulan sütun adının veya değişkenin veri türü değişirse, fonksiyonun tanımını değiştirmeniz gerekmez.

```
-Değişken_adı tablo_adı.kolon_adı%type
-değişken_adı değişken%type
```

Örnek: id değeri 200 olan film adını mesaj olarak döndürmek için;

```
do $$
declare
   film_title film.title%type;
   title_type film_title%type;
begin
   -- id değeri 200 olan film_title almak için
   select title from film
   into film_title
   where film_id = 200;
   -- Film ismini ekranda göstermek için
   raise notice 'Film adı -id 200-: %s', film_title;
        end $$;
```

Subblock (Alt Blok)

Bir alt blokta, dış bloktaki başka bir değişkenle aynı ada sahip bir değişken tanımladığınızda, dış bloktaki değişken alt blokta gizlenir. Dış bloktaki değişkene erişmek isterseniz de örnekteki gibi dış blok etiketini belirtmeniz gerekir.

```
do $$
<<dıs_blok>>
declare
  counter integer := 0;
begin
   counter := counter + 1;
   raise notice 'Mevcut sayaç değeri %', counter;
   declare
       counter integer := 0;
   begin
       counter := counter + 10;
       raise notice 'Alt bloktaki sayac değeri %', counter;
       raise notice 'Dış bloktaki sayaç değeri %', dıs_blok.
counter;
   end;
   raise notice 'Dış bloktaki sayaç değeri %', counter;
  end dıs_blok $$;
```

row type

Bir sonuç kümesinin tam satırını tutan satır değişkenlerini declare etmek için `row type`(satır türleri) kullanılır.

```
satır_değişkeni tablo.adı%ROWTYPE ;
```

Örnek id değeri 10 olan aktör adını ekrana mesaj olarak döndürmek için;

```
do $$
declare
   selected_actor actor%rowtype;
begin
   -- id değeri 10 olan actor'ü seçmek için
   select * from actor into selected_actor where actor_id = 10;
   raise notice 'Aktör Adı -id 10- : % %',
      selected_actor.first_name,
      selected_actor.last_name;
end; $$
```

İlk olarak veri tipi actor tablosundaki satırla aynı olan selected_actor değişkeni tanımlandı.
İkinci olarak actor_id değeri 10 olan satır select into ifadesi ile selected_actor değişkenine atandı.
Seçilen aktörün ad ve soyadını göstermek için nokta gösterimi ile first_name ve last_name alanlarına erişildi.

record type

`Row type`'a benzer. Record adı verilen bir tür sağlar. Bir sonuç kümesinin yalnızca bir satırını tutabilir. Row type'dan farklı olarak önceden tanımlanmış bir yapıya sahip değildir. Record değişkeninin yapısı, select veya for deyimi kendisine gerçek bir satır atadığında belirlenir. Kayıttaki alana ulaşmak için (.) syntax'ı kullanılır.

```
-değişken adı record ;
```

```
do
$$
declare
      rec record;
begin
      select film_id, title, length into rec from film where film_
id = 278;
      raise notice '% % süresi % dk', rec.film_id, rec.title,
rec.length;
end;
$$
language plpgsql;
```

Constants

Değişkenin aksine sabitin değeri başlatıldıktan sonra değiştirilemez. Kodları daha okunabilir ve sürdürülebilir kılar.

```
-sabit_adi constant data türü :=expression ;
```

Örnek: 0,17 kar elde etmek istediğimiz ürünün satış fiyatını ekrana yazdırmak için aşağıdaki örneği kullanabiliriz.

```
do $$
declare
   vt constant numeric := 0.17;
   net_fiyat numeric := 29.86;
begin
   raise notice 'Satış fiyatı : %', net_fiyat * ( 1 + vt );
end $$;
```

3

Mesajlar ve Hatalr

(Reporting Messages and Errors)

Örnek:

```
do $$
begin
  raise info 'bilgi mesajı %', now() ;
  raise log 'log mesajı %', now();
  raise debug 'debug mesajı %', now();
  raise warning 'uyarı mesajı %', now();
  raise notice 'bildirim mesajı %', now();
end $$;
```

Örnek: Ekranda error mesajı döndürmek için;

```
do $$
declare
  email varchar(255) := 'info@dataera.com.tr';
begin
       raise exception 'Daha önce kayıtlı e-mail adresi
kullanılamaz. %', email
            using hint = 'Tekrar kontrol edin.';
end $$;
```

Fonksiyonda kullanılabilecek seçenekler şu şekildedir:

`Message`: Hata mesajını ayarlama

`Hint`: Hatanın kök nedeninin kolayca bulunması için ipucu mesajı

`Detail`: Hata hakkında detaylı bilgi

`Errcode`: Koşul adına veya doğrudan beş karakterli SQLSTATE koduna göre olabilen hata kodunu tanımlama

Assert İfadesi

Hata ayıklama denetimlerini PLpgSQL koduna eklemek için kullanışlı bir kısayoldur.

Örnek:

```
do $$
declare
   film_count integer;
begin select count(*) into film_count from film;
    assert film_count > 1998, 'Film bulunamadı,film tablosunu
kontrol edin.';
end$$;
```

4

Kontrol Yapıları

(Control Structures)

- If Then
- Else-If
- If-Then-Elseif
- Case
- Loop
- While Loop
- For Loop
- Sonuç Kümesi Üzerinde (Query Result) Loop Döngüsü
- Dinamik Bir Sorgunun Sonuç Kümesinde Loop Döngüsü
- Exit İfadesi
- Continue İfadesi

if – then

Belli bir koşula dayalı olarak komut yürütmek için `IF` deyimi kullanılır.
`if` ifadesi, bir boole ifadesinin sonucuna göre hangi ifadelerin yürütüleceğini belirler.

```
     if şart then .......;
end if;
```

Eğer bir koşul doğruysa `if` ifadesi then ifadesinden sonrasını çalıştırır. Koşul false olarak değerlendirilirse, kontrol `END IF` kısmından sonraki, bir sonraki deyime iletilir. Koşul, doğru veya yanlış olarak değerlendirilen bir `boole` ifadesidir.
Koşul doğruysa yürütülecek bir veya daha fazla ifade olabilir. Herhangi bir geçerli ifade, hatta başka bir if ifadesi olabilir. Bir if ifadesi başka bir if ifadesinin içine yerleştirildiğinde, buna iç içe if ifadesi denir.

Örnek:

```
do $$
declare
  selected_film film%rowtype;
  input_film_id film.film_id%type := 157802;
begin
 select * from film into selected_film where film_id = input_film_
id;
  if not found then
     raise notice 'id : % olan film bulunamadı.',  input_film_id;
  end if;
end $$
```

else-if

```
if koşul then .... ;
      else alternatif_ifade;
END if;
```

```
do $$
declare
  secilen_film film%rowtype;
  input_film_id film.film_id%type := 116;
begin
 select * from film into secilen_film where film_id = input_film_id;
  if not found then
     raise notice 'Film % bulunamadı',
          input_film_id;
  else
     raise notice 'Film adı : %', secilen_film.title;
  end if;
end $$
```

if - then – elseif

Birden çok koşulu değerlendirir. Bir koşul doğruysa o daldaki karşılık gelen ifade yürütülür. Örneğin `koşul_1` doğruysa `ifade_1` çalıştırılır. Diğer koşullar değerlendirilmez. Tüm koşullar yanlış olarak değerlendirilirse, else dalındaki ifadeler çalıştırılır.

```
if koşul_1 then ifade_1;
elsif koşul_2 then ifade_2
...
elsif koşul_n then  ifade_n;
else
  else-ifadesi;
end if;
```

Örnek:

```
do $$
declare
   x_film film%rowtype;
   uzunluk_tanımı varchar(100);
begin
  select * from film  into x_film where film_id = 847;
  if not found then
     raise notice 'id bilgisi verilen film bulunamadı.';
  else
      if x_film.length >0 and x_film.length <= 50 then
             uzunluk_tanımı := 'Kısa';
        elsif x_film.length > 50 and x_film.length < 120 then
             uzunluk_tanımı := 'Orta';
        elsif x_film.length > 120 then
             uzunluk_tanımı:= 'Uzun';
        else
            uzunluk_tanımı := 'Uzunluk kategorisi bulunamadı.';
        end if;

        raise notice '% filmi süresi % filmler kategorisinde.',
           x_film.title,
           uzunluk_tanımı;
  end if;
end $$
```

Yukarıdaki örneği satırlarıyla birlikte açıklayacak olursak
x_film değişkenini film tablosundaki kolonlara göre tanımladık. Yani film tablosundan hangi kolondan sorgu çekersek değişken o kolonun türünü alacaktır. Ayrıca uzunluk_ tanımı adında türü `varchar` olan başka bir değişken daha tanımladık.

```
declare
   x_film film%rowtype;
   uzunluk_tanımı varchar(100);
```

Aşağıdaki `SELECT` sorgusunda `film_id` değeri 847 olan filmin satırlarını `SELECT INTO` deyimi ile x_film değişkenine atadık.
(`WHERE film_id` ifadesinde id değerini değiştirerek istediğimiz film için sorguyu çalıştırabiliriz.)

```
begin
  select * from film into x_film where film_id = 847;
```

İlk `if` ifademizde id değerini girdiğimiz film, tabloda yoksa verdiğimiz mesajı ekrana döndürmesini istedik. Eğer girdiğimiz `id` değerine sahip bir film varsa sorgu bu ilk `if` dalını atlar ve `else` dalına geçer.

```
if not found then
     raise notice 'Film bulunamadı;
```

else dalında başka bir if ifadesi bulunmakta. Eğer girdiğimiz id değeri tabloda varsa sorgu ilk if dalını atlayıp else dalına geliyordu.
Else dalına geldiğinde sorgu bir alttaki if ifadesine atlar. x_film değişkeni film tablosunda tanımlıydı.

Değişken bu tablodaki girilen id değerinin 'length' kolonunu (x_film.length) kontrol eder.

Eğer filmin uzunluğu 0-50 arasındaysa uzunluk_tanımı değişkenine 'Kısa' olarak atanır. Film uzunluğu bu aralıkta değilse bir alttaki elsif dalına atlanır. Buradaki koşul kontrol edilir. elsif dallarında belirtilen koşullar true olana kadar değerlendirme devam eder.

Doğru olan koşul sonucu uzunluk_tanımı değişkenine uygun ifade atanır. Ve else dalından çıkılarak raise notice kısmına gelinir. Burada değişkenlerin değerleri % sembolünün tuttuğu yerlere yazılarak uygun olan mesaj ekrana yazılır. Ve ilk if ifadesine ait olan endif ifadesi ile if yapısından tamamen çıkılır. Bir alttaki end ifadesiyle de sorgu sonlandırılır.
Örnekte id değeri 847 olan film için sorgu çektik. Bu id değerine sahip film tabloda olduğu için sorgu tarafından ilk if ifadesi atlandı ve else dalına gelindi. Burada bir alttaki if ifadesi değerlendirildi.
Belirtilen uzunluk koşulu filmin uzunluk aralığına uygun olmadığı için bir alttaki elsif dalına atlandı. Buradaki uzunluk aralığı da uygun değildi. Bir alttaki elsif dalına atlandı. Buradaki uzunluk koşulu filmin uzunluk aralığına uygun olduğu için burası değerlendirildi ve değişkene 'uzun' ifadesi atandı. Değerlendirme sonucu true olduğu için bu elsif'ten sonraki hiçbir koşul değerlendirilmedi ve end if ifadesiyle else 'nin altında başlayan if yapısından çıkılarak raise notice ile mesaj ekrana yazıldı. Daha sonra en alttaki endif ifadesi ile ilk if yapısı tamamen terkedildi.

```
else
      if v_film.length >0 and v_film.length <= 50 then
            uzunluk_tanımı := 'Kısa';
        elsif v_film.length > 50 and v_film.length < 120 then
            uzunluk_tanımı := 'Orta';
        elsif v_film.length > 120 then
            uzunluk_tanımı := 'Uzun';
        else
            uzunluk_tanımı := 'N/A';
        end if;

        raise notice ' % filmi süresi % filmler kategorisinde.',
           v_film.title,
           uzunluk_tanımı;
```

case

Diğer programlama dillerindeki `IF/ELSE` ifadesi ile aynı amaçla kullanılır.
Güçlü bir sorgu oluşturmak için sorguya `IF/ELSE` mantığının eklenmesini sağlayan ifadedir.

`SELECT, WHERE, GROUP BY, HAVING` yan tümceleri için kullanılabilir. `CASE` ifadesinin iki genel biçimi vardır: genel ve basit biçim

```
CASE
   WHEN condition_1  THEN result_1
   WHEN condition_2  THEN result_2
   [WHEN ...]
   [ELSE else_result]
END
```

Bu sözdiziminde, her koşul (condition_1, condition_2…), `true` veya `false` döndüren bir `boolean` ifadesidir. Bir koşul yanlış olarak değerlendirildiğinde, `CASE` ifadesi, doğru olarak değerlendirilen bir koşul bulana kadar bir sonraki koşulu yukarıdan aşağıya doğru değerlendirir. Bir koşul doğru olarak değerlendirilirse, `CASE` ifadesi koşula karşılık gelen sonucu döndürür. Bir sonraki ifadeyi değerlendirmeyi durdurur.
Tüm koşullar yanlış olarak değerlendirilirse, `CASE` ifadesi, `ELSE` sözcüğünü izleyen sonucu (`else_result`) döndürür. `ELSE` yan tümcesinin atlanması durumunda, `CASE` ifadesi `NULL` değerini döndürür.

Örnek: Sorguda film süreleri baz alınarak, filmler kategorilere ayrılmıştır. Süresi 0-60 arası olan filmler kısa, 60-130 arası olan filmler orta, 130'dan fazla olan filmler ise uzun olarak kategorize edilmiştir. Ve filmler uzunluklarına göre sıralanmıştır.

```
SELECT title, length, CASE
           WHEN length> 0 AND length <= 60 THEN 'Kısa'
           WHEN length > 60 AND length <= 130 THEN 'Orta'
           WHEN length> 130 THEN 'Uzun'
       END Kategori FROM film ORDER BY length;
```

Örnek: Sorguda filmler kiralama oranları baz alınarak kategorize edilmiştir. `SUM` fonksiyonu ile de 3 kategoriye ait kaç tane film olduğu sonucu döndürülmektedir.

```
SELECT
      SUM (CASE WHEN rental_rate = 0.99 THEN 1 ELSE 0
            END) AS "Ekonomi",
      SUM ( CASE WHEN rental_rate = 2.99 THEN 1 ELSE 0
            END) AS "Kitlesel",
      SUM (CASE WHEN rental_rate = 4.99 THEN 1 ELSE 0
            END) AS "Premium"
FROM film;
```

Sorgu sonucu aşağıdaki gibidir;

Data Output | Explain | Messages | Notifications

	Ekonomi bigint	Kitlesel bigint	Premium bigint
1	341	323	340

Örnek: Sorguda müşterilerin filmi kiraladıkları tarihe göre kiralama durumları belirtilmiştir. Sorgu `rental` tablosunda çalışmaktadır. Müşterilerin ad ve soyad bilgisinin alınması için customer tablosu sorguya katılmıştır. (`JOIN`)

```
SELECT customer_id ,rental_id , customer.first_name || ' ' ||
customer.last_name as Name ,
      rental_date ,
      CASE
            WHEN rental_date > '2005-02-15'
            AND rental_date < '2005-05-15' THEN 'KİRALAMA AKTİF
DEĞİL'
            WHEN rental_date > '2005-05-15'
            AND rental_date < '2005-08-15' THEN 'GECİKMELİ
TESLİM - %0,2 CEZAİ İŞLEM'
            WHEN rental_date > '2005-08-15'
            AND rental_date < '2005-11-15' THEN 'GECİKMELİ
TESLİM - %0,1 CEZAİ İŞLEM'
            WHEN rental_date > '2005-11-15' THEN 'KİRALAMA
SÜRECİ DEVAM EDİYOR.'
      END as "Kiralama Durumu"
FROM rental INNER JOIN customer USING (customer_id)
ORDER BY rental_date DESC ;
```

Yukarıdaki sonucun satır sayılarına göre toplamını veren sorgu şu şekildedir:

```
SELECT
      SUM (CASE
WHEN rental_date >='2005-02-15' AND rental_date < '2005-05-15'
THEN 1 ELSE 0 END ) as "KİRALAMA AKTİF OLMAYAN" ,
      SUM (CASE
WHEN rental_date > '2005-05-15' AND rental_date < '2005-08-15'
THEN 1 ELSE 0 END) as "%0,2 CEZAİ İŞLEM UYGULANACAK" ,
      SUM (CASE
WHEN rental_date > '2005-08-15' AND rental_date < '2005-11-15'
THEN 1 ELSE 0 END) as "%0,1 CEZAİ İŞLEM UYGULANACAK" ,
      SUM (CASE
WHEN rental_date > '2005-11-15' THEN 1 ELSE 0 END) as "KİRALAMA
SÜRECİ DEVAM EDEN"
FROM rental ;
```

Verdiğimiz örneklerde `WHERE` koşulunu kullanıp müşteri adına göre arama yapmak istersek sorgu şu şekilde olur;

```
SELECT customer_id ,rental_id , customer.first_name || ' ' ||
customer.last_name as Name ,
      rental_date ,
      CASE
            WHEN rental_date >= '2005-02-15'
            AND rental_date < '2005-05-15' THEN 'KİRALAMA AKTİF
DEĞİL'
            WHEN rental_date > '2005-05-15'
            AND rental_date < '2005-08-15' THEN 'GECİKMELİ
TESLİM - %0,2 CEZAİ İŞLEM'
            WHEN rental_date > '2005-08-15'
            AND rental_date < '2005-11-15' THEN 'GECİKMELİ
TESLİM - %0,1 CEZAİ İŞLEM'
            WHEN rental_date > '2005-11-15' THEN 'KİRALAMA
SÜRECİ DEVAM EDİYOR.'
      END as "Kiralama Durumu"
FROM rental  INNER JOIN customer USING (customer_id) WHERE
customer.first_name || ' ' || customer.last_name = 'Mary Smith'
ORDER BY rental_date ASC ;
```

loop

Döngü bir '`exit`' veya '`return`' ifadesi ile sonlandırılana kadar bir kod bloğunu tekrar tekrar yürüten koşulsuz döngü tanımlar.

```
<<etiket>>
loop
   ifadeler;
end loop;
```

Bir koşula dayalı olarak `loop` döngüsünü sonlandırmak için;

```
<<etiket>>
loop
   ifadeler;
   if koşul then
      exit;
   end if;
end loop;
```

Bir döngü ifadesini başka bir döngü ifadesi içine de yerleştirmek mümkündür. (İç içe döngü)

```
<<en_dış>>
loop
   ifadeler;
   <<iç>>
   loop
     /* ... */
     exit <<iç>>
   end loop;
end loop;
```

Bu tür döngülerde ifadelerin hangi döngüye atıfta bulunduğunu belirtmek için ve çıkış ve devam ifadelerinin karışmaması için döngü etiketi kullanılmalıdır.

Örnek:

```
do $$
declare
   n integer:= 10;
   f integer := 0;
   sayac integer := 0 ;
   i integer := 0 ;
   j integer := 1 ;
begin
      if (n < 1) then
            f := 0 ;
      end if;
      loop
            exit when sayac = n ;
            sayac := sayac + 1 ;
            select j, i + j into i, j ;
            raise notice 'sayac:% -- i:% -- j:% -- i + j :%' ,
sayac ,i ,j , i + j;
      end loop;
      f := i;
    raise notice 'SONUC : %', f;
end; $$
```

Yukarıdaki sorguyu açıklayacak olursak;
`Declare` kısmında değişkenler tanımlandı. `If` bloğundaki `n<1` koşulu `true` olmadığı için `if` bloğundan çıkıldı.(`n=10`). `Loop` bloğunda döngüye girildi.

`exit when sayac = n ;` ifadesi sayac değişkeninin değerinin 10 (n=10) olması durumunda döngüden çıkılmasını sağlar. Yani sayac değişkeninin değeri 10'a eşit olana kadar döngü çalışır. Döngüde her seferinde sayac değişkeninin değeri 1 artar.(sayac := sayac + 1)
Ve daha sonra `SELECT INTO` ifadesi ile `i` ve `j` değerlerine atamalar yapılır. Sayac değeri artarak 10'a geldiğinde tüm bu işlemler durur. Değişkenler en son aldıkları değerde kalırlar. `end loop`; ifadesi ile döngüden çıkılır. Döngü sonucu `i` değişkeninin aldığı son değer `f` değişkenine atanır ve ekrana mesaj dönülür.
(Ekrana dönen mesajda gelen sayac, `i, j ve i+j` değerleri sayesinde döngü mantığını anlayabilirsiniz.)

while loop

Örnek:

```
do $$
declare
   sayac integer := 0;
begin
   while sayac < 5 loop
      raise notice 'sayac: %', sayac;
        sayac:= sayac + 1;
   end loop;
end$$;
```

Sorguda `while` ifadesindeki koşul `false` olana kadar döngü gerçekleşir. Her döngüde `while` ifadesine tekrar dönmeden önce sayac değişkenine +1 değer eklenir.

for loop

```
[ <<etiket>> ]
for loop_counter in [ reverse ] from.. to [ by adım ] loop
    ifadeler
end loop [ etiket ];
```

İlk olarak `for` döngüsü yalnızca döngü içerisinde erişilebilen bir döngü_sayacı (`loop_counter`) oluşturur. Varsayılan olarak for döngüsü her yinelemeden sonra adımı `loop_counter` öğesine ekler. Ancak reverse seçeneğini kullandığımızda adımı çıkarır.

`From` ve `to` aralığın alt ve üst sınırını belirten ifadelerdir. `For` döngüsü döngüye başlamadan önce bu ifadeleri değerlendirir.

by anahtar sözcüğünü izleyen adım yineleme adımını belirtir. Varsayılan olarak 1'dir. `For` döngüsü bu adım ifadesini yalnızca 1 kere değerlendirir.

Örnek:

```
do $$
begin
   for sayac in 1..5 loop
      raise notice 'sayac: %', sayac;
   end loop;
end; $$
```

Sorguda `for` döngüsünde sayac değişkeninin değeri 1 ile 5 arasında değişmektedir. (1 ve 5 dahil.) Her seferinde default olarak sayac değişkeninin değeri 1 artmaktadır. Değişkenin değeri artıp 6 olduğunda `for` döngüsünden çıkılır end `loop`; ifadesinin altındaki satır değerlendirilir.

Örnek:

```
do $$
begin
   for sayac in reverse 8..1 by 2 loop
      raise notice 'sayac: %', sayac;
   end loop;
end; $$
```

Sonuç Kümesi Üzerinde (Query Result) Loop Döngüsü

```
[ <<etiket>> ]
for target in query loop
    ifadeler
end loop [ etiket ];
```

Örnek: En uzun 10 filmi uzunluk ve title değerlerine göre sıralayıp ekrana yazan sorgu:

```
do
$$
declare
    i record;
begin
    for i in select title, length, film_id from film
             order by length desc, title
                limit 10

    loop
      raise notice 'Film id değeri:% - Film Adı:% - Film
Süresi:% dakika',i.film_id, i.title, i.length;
    end loop;
end;
$$
```

Dinamik Bir Sorgunun Sonuç Kümesinde Loop Döngüsü

```
[ <<label>> ]
for row in execute query_expression [ using query_param [, ...
] ]
loop
    statements
end loop [ label ];
```

Using yan tümcesi sorguya parametreleri iletmek için kullanılır.

Örnek: Aşağıdaki sorgu sort_type değişkenine dayalı oluşturulur ve sonuç kümesinin satırı üzerinde yineleme yapmak için for loop kullanır.

```
do $$
declare
    -- sıralama türü(sort_type) 1: title, 2: release year
    sort_type smallint := 1;
      -- Filmlerin sayısını döndürmek
      f_count int := 10;
      -- Film tablosunda yineleme yapmak için
      f record;
      -- dynamic query(dinamik sorgu)
    sorgu text;
begin
            sorgu := 'select title, release_year from film ';
      if sort_type = 1 then
            sorgu := sorgu || 'order by title';
      elsif sort_type = 2 then
                  sorgu := sorgu || 'order by release_year';
      else
         raise 'invalid sort type %s', sort_type;
      end if;
             sorgu := sorgu || ' limit $1';
      for f in execute sorgu using f_count
        loop
           raise notice '% - %', f.release_year, f.title;
      end loop;
end; $$
```

`sort_type` 1; filmleri title'a göre 2; release year değerine göre sıralar. (Değişkeni 2 yapıp sorguyu tekrar çalıştırabilirsiniz.)

`f_count` film tablosundan sorgulanacak satır sayısıdır.

exit

```
exit [label] [when boolean_expression]
```

boolean_expression true olarak değerlendirilirse, exit ifadesi döngüyü sonlandırır. label(etiket) kullanılmazsa exit ifadesi içinde bulunduğu mevcut döngüyü sonlandırır.

```
<<block_label>>
BEGIN
    -- bazı kodlar
    EXIT [block_label] [WHEN şart];
    -- bazı kodlar
END block_label;
```

Örnek:

```
do $$
declare
   i int = 0;
   j int = 0;
begin
  <<dıs_loop>>
  loop
     i = i + 1;
     exit when i > 3;
       -- ic loop
       j = 0;
     <<ic_loop>>
     loop
            j = j + 1;
            exit when j > 3;
            raise notice '(i,j): (%,%)', i, j;
       end loop ic_loop;
  end loop dıs_loop;
end;
$$
```

Sorguda öncelikle ilk `loop` çalışır ve içteki `loop`'a girer. İçteki loop döngüsünü tamamladıktan sonra(`j>3` olduğunda) dıştaki `loop`'a gelir. Ta ki `i>3` olana kadar devam eder. İçteki `loop`'un her tamamlanıp sorgunun dıştaki `loop`'a dönmesiyle `j=0` olur.

Örnek: Bu örnekte içteki loop döngüsünde j değişkeninin değeri 4 olduğunda exit ifadesiyle dıştaki loop döngüsü terkedilir.

```
do
$$
declare
   i int = 0;
   j int = 0;
begin
  <<dıs_loop>>
  loop
     i = i + 1;
     exit when i > 3;
       -- ic loop
       j = 0;
     <<ic_loop>>
     loop
            j = j + 1;
            exit dıs_loop when j > 3;
            raise notice '(i,j): (%,%)', i, j;
      end loop ic_loop;
  end loop dıs_loop;
end; $$
```

Continue İfadesi

```
continue [loop_label] [when koşul] ;
**loop_label, geçerli yinelemeyi atlamak istediğiniz döngünün
etiketidir.
```

Örnek:

```
do
$$
declare
   sayac int = 0;
begin
  loop
     sayac = sayac + 1;
       -- sayac > 10 ise döngüden çık
       exit when sayac > 10;
       -- sayac 2'ye tam bölünebilen bir sayıysa yinelemeyi
atla değilse raise notice satırını çalıştır.
       continue when mod(sayac,2) = 0;
       -- sayac çıktısını yaz
       raise notice '%', sayac;
  end loop;
end;
$$
```

**Sorguda sayac=0 şeklinde değişken tanımladık. Sayac değişkeni döngünün her başlangıcında 1 artar. 10 olunca döngüden çıkılır ve end loop; ifadesinden sonraki satır çalışır.
continue when mod(sayac,2) = 0; ifadesi sayac değişkeninin aldığı değer eğer 2'ye tam bölünebiliyorsa döngünün başından devam edilmesini sağlar. Eğer koşul true değilse bir alt satıra atlanır.

5

Veri Kullanicı Tanımlı Fonksiyonlar

(User - Defined Functions)

- Fonksiyon Oluşturma
- Fonksiyonları Çağırma
- Fonksiyon Parametreleri
- Fonksiyon Overloading
- Tablo Döndüren Fonksiyon
- Fonksiyon Silmek

Fonksiyonlar işimizi kolaylaştırmak için sürekli olarak kullandığımız sql sorgularına tek bir noktadan erişmemizi sağlar.

Fonksiyon Oluşturma (create function)

```
create [or replace] function fonksiyon_adı(param_list)
   returns return_type
   language plpgsql
  as $$
declare
--
begin
 -- logic
end; $$
```

`Create function` sözcüklerinden sonra fonksiyon adı yazılır. Mevcut işlev değiştirilmek isteniyorsa `or replace` ifadesi kullanılır.
İşlev adından sonra parametre listesi belirtilir. Bir fonksiyon sıfır veya daha çok parametreye eşit olabilir.
`return` ifadesinden sonra döndürülen değerin veri türü belirtilir.
Fonksiyonun prosedürel dilini belirtmek için plpgsql dili seçilir. (PostgreSQL sadece bu dili değil, birçok prosedürel dili destekler.)
Son olarak dolar($$) ile alıntılanan string sabitine bir blok yerleştirilir.

Örnek:

```
create function toplam_film_sayısı(len_from int, len_to int)
returns int
language plpgsql
as
$$
declare
   film_sayısı integer;
begin
   select count(*)  into film_sayısı from film where length
between len_from and len_to;
   return film_sayısı;
end;
$$;
```

Yukarıdaki sorguyu detaylı incelersek;
Sorgu sonucu create edilen toplam_film_sayısı fonksiyonu iki ana bölümden oluşur: başlık(`header`) ve gövde(`body`)
Başlık bölümünde:
İlk olarak `create function` ifadesinden sonra fonksiyon adı belirtildi.
İkinci olarak bu fonksiyon `integer`(tam sayı) veri türünde `len_from` ve `len_to` adlı iki parametreyi kabul etti.
Üçüncü olarak toplam_film_sayısı fonksiyonu return int ifadesi tarafından belirtilen bir tam sayısı(int) döndürür.
Son olarak işlev dili için plpgsql belirtildi.
Gövde bölümünde:
`$$` işaretleri arasına fonksiyonun mantığını ve deklarasyonunu(bildirimini) içeren blok yerleştirildi.
Deklarasyon bölümünde film tablosundan seçilen filmlerin sayısını saklayan film_sayısı değişkeni tanımlandı.

Bloğun gövdesinde uzunlukları `len_from` ve `len_to` arasında olan filmlerin sayısını seçmek için `SELECT INTO` deyimi kullanıldı ve sonucu film_sayısı değişkenine atandı.
Blok sonunda film_sayısı değerini döndürmek için `return` ifadesi kullanıldı.

pgAdmin 'de fonksiyon listesini görüntülemek için resimde görünen functions sekmesine tıklanır.

Terminal ortamında fonksiyon listesini görüntülemek için `\df` komutu kullanılır.

Fonksiyonları Çağırmak (call function)

Fonksiyonları kullanmak için aşağıdaki yöntemler izlenir.

```
select toplam_film_sayısı(50,85);
```

Uzunluğu 50 ile 80 arasında bir değere sahip olan toplam film sayısı sonuç olarak döner.

Fonksiyonun birkaç parametresi olduğunda konum gösterimi kullanılarak fonksiyon çağırılır. Fonksiyon çağrısını daha belirgin hale getirir.

```
select toplam_film_sayısı( len_from => 50,   len_to => 85);
select toplam_film_sayısı(len_from := 50, len_to := 85 );
select toplam_film_sayısı(50, len_to => 85);
select toplam_film_sayısı(len_from => 50, 85);
```

Son sorgu hata verir.

Fonksiyon Parametre Modları (In, Out, Inout) In Modu

Default moddur. Fonksiyona bir değer iletir. Parametlerde sabitler gibi davranır. Bir değer atanamaz.

Örnek: Aşağıdaki sorgu film_id değerini kullanarak film adını bulan bir fonksiyon create etmiştir.

```
create or replace function id_ile_film_bulma(p_film_id int)
returns varchar
language plpgsql
as $$
declare
   film_title film.title%type;
begin
  -- id kullanarak film adını bulma
  select title  into film_title from film where film_id = p_film_
id;
  if not found then
     raise 'Film id % not found', p_film_id;
  end if;
  return film_title;
  end;$$
```

```
select id_ile_film_bulma(788)
```

OUT MODU

Out parametleri birden çok değer döndürmesi gereken işlevlerde çok kullanışlıdır.

Örnek:

```
create or replace function film_uzunluk(
    out min_uzunluk int,
    out max_uzunluk int,
    out avg_uzunluk numeric)
language plpgsql
as $$
begin
  select min(length),  max(length), avg(length)::numeric(5,1)
  into min_uzunluk, max_uzunluk, avg_uzunluk
  from film;
end;$$
```

Fonksiyonu çağırmak için;

```
SELECT * FROM film_uzunluk() ;
```

INOUT MODU

`IN` ve `OUT` modlarının birleşimidir. Yani arayan bir fonksiyona bir argüman iletebilir. Fonksiyon argümanı değiştirir ve güncellenmiş değeri döndürür.

Örnek:

```
create or replace function inout_test(
      inout x int,
      inout y int
)
language plpgsql
as $$
begin
   select x,y into y,x;
end; $$;

select * from inout_test(5,3)
```

Fonksiyon Overloading

PostgreSQL farklı argümanlara sahip oldukları sürece birden fazla fonksiyonun aynı adı paylaşmasına izin verir. İki veya daha fazla fonksiyon aynı adı paylaşıyorsa function overloading denir. Bu tür fonksiyonları çağırdığınız zaman PostgreSQL fonksiyon argüman listesine göre yürütülecek en iyi aday fonksiyonu seçer.

Örnek: `id` değerlerine göre müşterilerin kira sürelerini döndüren fonksiyon örneğidir.

```
create or replace function kiralama_süresi( p_musteri_id
integer)
returns integer
language plpgsql
as $$
declare
      kiralama_süresi integer;
begin
      select
            sum( extract(day from return_date - rental_date))
      into kiralama_süresi
    from rental
      where customer_id = p_musteri_id;
      return kiralama_süresi;
end; $$

select * from kiralama_süresi(205) ;
```

(extract(day from return_date - rental_date) ifadesi belirtilen tarihin gün değerinin döndürülmesini sağlar.

Belirli bir tarihten bugüne kadar bir müşterinin kiralama süresini öğrenmek istediğimizi varsayalım. Bunun için yukarıdaki fonksiyona `p_from_date` parametresi ekleyebiliriz. Ve aynı ada sahip yeni bir fonksiyon geliştirebiliriz.

```
create or replace function kiralama_süresi(
      p_musteri_id integer,
      p_from_date date
)
returns integer
language plpgsql
as $$
declare
      kiralama_süresi integer;
begin
      select sum( extract( day from return_date + '12:00:00' -
rental_date))
      into kiralama_süresi
      from rental
      where customer_id = p_musteri_id and
               rental_date >= p_from_date;

      return kiralama_süresi;
end; $$
```

```
select kiralama_süresi(189 , '2005-08-01') ;
```

Tablo Döndüren Fonksiyon

Bu fonksiyonlar tek bir değer döndürmek yerine belirtilen sütun listesine sahip bir tablo döndürür.

ÖRNEK: Aşağıdaki fonksiyon, `ILIKE` operatörünü kullanarak title değerleri belirli bir kalıpla eşleşen tüm sonuçları döndürür.

```
create or replace function get_film (p_kalıp varchar)
returns table (film_title varchar, film_release_year int)
      language plpgsql
as $$
begin
      return query
            select title, release_year::integer from film where
title ilike p_kalıp;
end;$$
```

```
select * from  get_film('Al%') ;
```

Sonuç kümesi sütunları, return table yan tümcesinden sonra tanımlanan sütunlar ile aynıdır.

Bu fonksiyon film_title ile eşleştirmek istediğiniz kalıp olan bir p_kalıp parametresini kabul eder. Film tablosundaki release_year sütununun veri türü `int` olmadığı için bunu `cast operatörü ::` kullanarak integer'a çevirdik.

```
create or replace function get_film (p_kalıp varchar, p_year int
)
returns table (film_title varchar, film_release_year int )
language plpgsql
as $$
declare
    var_r record;
begin
      for var_r in(
            select title, release_year from film where title
ilike p_kalıp and
                  release_year = p_year
        ) loop  film_title := upper(var_r.title) ;
            film_release_year := var_r.release_year;
           return next;
      end loop;
end; $$
```

Gövde kısmında sorguyu satır satır işlemek için bir `for loop` kullanıldı.
return next ifadesi, fonksiyonun döndürülen tablosuna satır ekler.

Fonksiyon Silmek (Drop Function)

```
drop function [if exists] fonksiyon_adı(argument_list)
[cascade | restrict]

DROP FUNCTION get_film ;
```

6

Stored Procedures

Stored procedures, belli bir işlevi yerine getirmek için kullanılırlar. Her seferinde aynı işlemleri yapma gereksinimini ortadan kaldırır, zamandan tasarruf edilmesini sağlar.
Giriş (input) parametleri alınır ve çağrıldığında çıktı (output) değeri döndürür.
`Begin ... End, If-Else, While, Loop, Case` yapıları ile kullanılabilir.
Kullanıcı tanımlı fonksiyonlar ile benzerler.
`CALL` ifadesi kullanılarak çağrılırlar.
Bir defa derlendikten sonra tekrar derlenmesine gerek kalmaz. Veritabanında `'execution plan'` olarak saklanırlar. Yani kod ilk çalışmada derlenir. Sonrasında hafızada execution plan olarak saklanır. Tekrar çalıştırılmak istendiğinde execution plan hali kullanılır, tekrar derlenmesine gerek olmaz. Bu sayede daha hızlı execution gerçekleştirilir.

Prosedur Oluşturma (create procedure)

```
create [or replace] procedure procedure_adı(parametre_list)
language plpgsql
as $$
declare
--
begin
-- stored procedure body
end; $$
```

Stored proceduredeki parametreler IN ve INOUT modlarına sahip olabilir, OUT moduna sahip olamaz.

Stored procedure aşağıdaki gibi bir değer döndürmez.
return ifade.. ;
Ancak stored prosedürü hemen durdurmak için aşağıdaki gibi return deyimini ifade kısmı olmadan kullanabilirsiniz.

Return ;
**Bir stored procedure'den bir değer döndürmek isterseniz parametreleri INOUT moduyla kullanabilirsiniz.

Örnek:

```
CREATE PROCEDURE deneme()
language plpgsql
as $$
begin
raise notice 'PostgreSQL Dokümanları' ;
end ; $$
call deneme()
```

Oluşturduğumuz procedure içinde değişiklik yapmak için;

```
CREATE or replace PROCEDURE deneme()
language plpgsql
as $$
begin
raise notice 'PostgreSQL Dokümanları' ;
raise notice 'SQL Dokümanları' ;
end ; $$

call deneme()
```

Örnek: Hesaplar isimli tablo oluşturup, tabloya değerler girelim. Ve bu tabloyu kullanarak procedure oluşturalım. (Aşağıdaki create ve insert into deyimlerinin her birini tek tek çalıştıralım.)

```
create table hesaplar (
    id int generated by default as identity,
    ad varchar(100) not null,
      soyad varchar(100) not null,
    bakiye dec(15,2) not null,
    primary key(id)
);

insert into hesaplar(ad,soyad,bakiye)
values('Barış','KAYA' , 10000);

insert into hesaplar(ad,soyad,bakiye)
values('Aysel','DENİZ', 12000);

insert into hesaplar(ad,soyad,bakiye)
values('Gamze','ÇINAR', 14500);

insert into hesaplar(ad,soyad,bakiye)
values('Mert','ATEŞ', 18000);

select * from hesaplar ;
```

Tabloya kayıtlı hesaplar arasında bakiyeleri kullanarak transfer gerçekleştirebileceğimiz bir procedure oluşturalım. Procedurde belirtilen parametreler kullanıcılardan alınır. Kullanıcı göndericinin ve alıcının `id` değerlerini ve gönderilecek miktarı girer.

```
create or replace procedure transfer(gönderici int,alıcı int,
miktar dec)
language plpgsql
as $$
begin
    -- Göndericinin hesabından gönderilen miktar kadar azaltmak
için
    update hesaplar set bakiye = bakiye-miktar where id =
gönderici;
    -- Alıcının hesabına gelen miktar kadar eklemek için
    update hesaplar set bakiye = bakiye + miktar where id =
alıcı;
    commit;
end;$$
```

	id [PK] integer	ad character varying (100)	soyad character varying (100)	bakiye numeric (15,2)
1	1	Barış	KAYA	10000.00
2	2	Aysel	DENİZ	12000.00
3	3	Gamze	ÇINAR	14500.00
4	4	Mert	ATEŞ	18000.00

Procedure çağırmadan önce;

```
call transfer(2,4,1000);
```

Data Output | Explain | Messages | Notifications

	id [PK] integer	ad character varying (100)	soyad character varying (100)	bakiye numeric (15,2)
1	1	Barış	KAYA	10000.00
2	2	Aysel	DENİZ	11000.00
3	3	Gamze	ÇINAR	14500.00
4	4	Mert	ATEŞ	19000.00

Prosedur Silme (drop procedure)

```
drop procedure [if exists] procedure_adi (argument_list)
[cascade | restrict]
```

Stored procedure adı veritabanında unique (benzersiz) değilse, silerken stored procedure argüman listesini belirtmek gerekir. Çünkü farklı bağımsız değişken listesine sahip aynı isimli stored procedureler olabilir.

Stored procedure'e bağımlı nesneler varsa o nesneleri ve prosedürü silmek için `cascade` ifadesi kullanılır. Default seçenek, procedure herhangi bir nesneye sahip olması durumunda kaldırılmasını reddeder. (`restrict`)

Birden fazla procedure silmek için;

```
drop procedure [if exists] ad1, ad2, ...;
```

Örnek: İlk olarak kullanıcı tarafından tam_ad şeklinde girilen aktör isminin ad ve soyad değerlerine ayrılmasını (split_part ile) daha sonra bu değerlerin actor tablosunda first_name ve last_name kolonlarına yazılmasını sağlayan bir procedure oluşturalım.

```
create or replace procedure actor_ekle (tam_ad varchar)
language plpgsql
as $$
declare
      ad varchar;
      soyad varchar;
begin
      -- Tam adı ad ve soyad değerlerine bölmek için
      select
            split_part(tam_ad,' ', 1),
            split_part(tam_ad,' ', 2)
      into ad, soyad;
      -- Tabloya ad ve soyad değerleri eklemek için ;
      insert into actor(first_name, last_name)
      values(ad,soyad);
end;  $$;
```

```
call actor_ekle('Asım Kaya')
select * from actor where first_name = 'Asım'
```

Oluşturduğumuz bu procedürü silmek için;

```
drop procedure actor_ekle;
-Procedure adı unique olmasaydı eğer;
drop procedure actor_ekle(varchar) ;
```

7

Triggers

`Insert, update, delete` gibi olay gerçekleştiğinde çağrılan özel bir kullanıcı tanımlı fonksiyondur. Yeni bir trigger create etmek için önce bir trigger fonksiyonu tanımlanır ve ardından bu fonksiyon bir tabloya bağlanır. Bir trigger ile kullanıcı tanımlı fonksiyon arasındaki fark; bir tetikleyici olay meydana geldiğinde tetikleyicinin otomatik olarak çağrılmasıdır.

PostgreSQL'de 2 ana tür trigger vardır:
satır ve ifade düzeyinde (row and statement level). İkisi arasındaki fark, tetikleyicinin kaç kez ne zaman çağrıldığıdır.
Örneğin; 17 satırı etkileyen bir update ifadesi kullanıllırsa, row level trigger 17 kez çağrılır. Statement level düzeyinde ise trigger bir kez çağırılır.
Trigger'ın bir olaydan önce mi yoksa sonra mı çağrılacağını belirtebilirsiniz. Önce çağrılırsa, geçerli satır için işlemi atlayabilir hatta güncellenen veya eklenen satırı değiştirebilir. Olaydan sonra çağırlırsa eğer, tüm değişiklikler trigger için kullanılabilir.
Triggerları karmaşık veri bütünlüğü kurallarını korumak için de kullanabilirsiniz. Örneğin müşteri tablosuna yeni bir satır eklendiğinde, banka ve kredi tablolarında başka satırlar da oluşturulmalıdır.
En büyük dezavantajı; veri değiştiğinde etkilerini anlayabilmek için tetikleyicinin var olduğunu bilmek ve mantığını anlamak gerekir.
PostgreSQL trigger içn SQL standartını kullansa da, bazı belirli özellikleri vardır:
PostgreSQL `TRUNCATE` olayı için tetikleyiciyi tetikler.
PostgreSQL viewlarda statement level bir trigger tanımlamaya olanak tanır.
PostgreSQL, trigger eylemi olarak kullanıcı tanımlı fonksiyon tanımlanmasına izin verirken, SQL standartı herhangi bir SQL komutunu kullanmanıza izin verir.

Trigger Oluşturmak (Create Trigger)

İlk olarak `CREATE FUNCTION` ifadesi ile bir trigger fonksiyonu oluşturulur. Daha sonra `CREATE TRIGGER` ifadesi ile trigger fonksiyonu bir tabloya bağlanır.
Bir trigger fonksiyonu herhangi bir bağımsız değişken almaz ve tür tetikleyici (type trigger) ile bir dönüş değerine sahiptir.

```
CREATE FUNCTION trigger_fonksiyonu()
   RETURNS TRIGGER
   LANGUAGE PLPGSQL
AS $$
BEGIN
   -- trigger logic
END; $$
```

Bir trigger fonksiyonu bir dizi local değişken içeren TriggerData adlı özel bir yapı aracılığla çağrı ortamı hakkında verileri alır.
Bir trigger fonksiyonu tanımlandığında onu insert, update, delete gibi bir veya daha fazla tetikleyici olaya bağlayabilirsiniz.

```
CREATE TRIGGER trigger_adı
   {BEFORE | AFTER} { olay}
   ON tablo_adı
   [FOR [EACH] { ROW | STATEMENT }]
       EXECUTE PROCEDURE trigger_fonksiyonu
```

```
{BEFORE | AFTER} Tetikleme zamanı belirtilir.
{ Olay} Tetikleyiciyi çağıran olay belirtilir.
ON tablo_adı Tetikleyiciyle ilişkili tablo adı belirtilir.
 [FOR [EACH] { ROW | STATEMENT }] Tetikleyici türü belirtilir.
```

`EXECUTE PROCEDURE` trigger_fonksiyonu Tetikleyici fonksiyon adı belirtilir.

Örnek: Bir çalışanın adı değiştiğinde, değişiklikleri calısan_denetimleri adlı ayrı bir tabloda kaydetmek istediğimizi varsayalım.

```
DROP TABLE IF EXISTS calısanlar;
CREATE TABLE calısanlar(
   id INT GENERATED ALWAYS AS IDENTITY,
   ad VARCHAR(40) NOT NULL,
   soyad VARCHAR(40) NOT NULL,
   PRIMARY KEY(id));

CREATE TABLE calısan_denetimleri (
   id INT GENERATED ALWAYS AS IDENTITY,
   calısan_id INT NOT NULL,
   soyad VARCHAR(40) NOT NULL,
   degısıklık_tarıhı TIMESTAMP(6) NOT NULL );
```

Fonksiyon oluşturmak için;

```
CREATE OR REPLACE FUNCTION soyad_degısıklık_kaydı()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
$$
BEGIN
      IF NEW.soyad <> OLD.soyad THEN
              INSERT INTO calısan_denetimleri(calısan_
id,soyad,degısıklık_tarıhı)
              VALUES(OLD.id,OLD.soyad,now());
      END IF;
      RETURN NEW;
END;
$$
```

`OLD` `insert` işleminden önceki satırı `NEW` ise sonraki satırı temsil eder. Eğer bu iki satır değişkeni birbirine eşit değilse calısan_denetimleri tablosuna yeni değerler girilir.

Trigger oluşturmak için;

```
CREATE TRIGGER soyad_degisikligi
  BEFORE UPDATE
  ON calısanlar
  FOR EACH ROW
  EXECUTE PROCEDURE soyad_degısıklık_kaydı();
```

Tabloya bazı değerler ekleyelim ve bu degerlerde güncelleme yapalım.

```
INSERT INTO calısanlar (ad, soyad)
VALUES ('Kübra', 'Dogan');

INSERT INTO calısanlar (ad, soyad)
VALUES ('Lale', 'Burak');

INSERT INTO calısanlar (ad, soyad)
VALUES ('Beyza', 'Türk');

UPDATE calısanlar
SET soyad = 'Dogan Yılmaz'
WHERE ID = 1;
```

Tabloları kontrol edelim.

```
select * from calısanlar ;
select*from calısan_denetimleri ;
```

Trigger Silmek (Drop Trigger)

```
DROP TRIGGER [IF EXISTS] trigger_adı
ON tablo_adı [ CASCADE | RESTRICT ];
```

Örnek: staff isimli tabloya yeni veri girişi yapıldığında username değerinin uzunluğunu kontrol eden ve uzunluk <12 ise ekranda uyarı döndüren trigger oluşturalım.

```
CREATE FUNCTION check_kullanıcı()
    RETURNS TRIGGER
AS $$
BEGIN
    IF length(NEW.username) < 12 OR NEW.username IS NULL THEN
        RAISE EXCEPTION 'Username değeri 12 karakterden kısa
olamaz.';
    END IF;
    IF NEW.NAME IS NULL THEN
        RAISE EXCEPTION 'Username değeri boş geçilemez.';
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;
------------------------------------------------------------
--
CREATE TRIGGER kullanıcıadı_check
    BEFORE INSERT OR UPDATE
ON staff
FOR EACH ROW
    EXECUTE PROCEDURE check_kullanıcı();
```

Trigger'ı kontrol edelim.

```
insert into staff(staff_id,username)
values(3,'Nilay') ;
```

**Sorgu sonucu ekranda fonksiyon içerisinde belirttiğimiz hata döner.

Trigger silmek için:

```
DROP TRIGGER kullanıcıadı_check
ON staff;
```

Trigger'da Değişiklik Yapmak (Alter Trigger)

```
ALTER TRIGGER trigger_adı
ON tablo_adı
RENAME TO yeni_trigger_adı;
```

Bu söz dizimi trigger ismini değiştirir.

Örnek:

```
ALTER TRIGGER soyad_degısıklıgı
ON calısanlar
RENAME TO degısıklık_yapılan_satır_kaydı ;
```

PostgreSQL trigger `OR REPLACE` ifadesini desteklemez. Triggerda değişiklik yapmak için `CREATE` ya da `DROP` ifadesi kullanılır.

Örnek: Daha önce oluşturduğumuz kullanıcıadı_check adlı trigger'ın fonksiyonunu değiştirmek isteyelim. (Yeni fonksiyon create edilmiş varsayıldı.)

```
CREATE TRIGGER kullanıcıadı_check
    BEFORE INSERT OR DELETE
ON staff
FOR EACH ROW
    EXECUTE PROCEDURE check_kullanıcı();
```

Trigger Devre Dışı Bırakmak (Disable Trigger)

```
--Tetikleyicinin ilişkili olduğu tablo adı

ALTER TABLE tablo_adi
DISABLE TRIGGER trigger_adi| ALL
```

Tabloyla ilgili tüm triggerları devre dışı bırakmak için ALL sözcüğü kullanılır.
Trigger devre dışı bırakıldığında hala veritabanında mevcuttur. Ancak bu trigger ile ilgili bir olay gerçekleştiğinde tetiklenmez.

```
ALTER TABLE calısanlar
DISABLE TRIGGER degısıklık_yapılan_satır_kaydı ;
```

Trigger Etkinleştirmek (Enable Trigger)

```
ALTER TABLE tablo_adi
ENABLE TRIGGER trigger_adi |  ALL;
```

```
ALTER TABLE calısanlar
ENABLE TRIGGER degısıklık_yapılan_satır_kaydı ;
```

8
Views

Bir tür sanal tablodur (görüntü). Gerçek tablodaki gibi satır ve sütunlara sahiptir. Veritabanında bulunan bir veya daha fazla tablodan alanlar seçilerek bir view oluşturulabilir. Bir tablonun tüm satırlarına veya belirtilen koşula uygun belirli satırlarına sahip olabilir. Sorguları basitleştirir ve sürelerini kısaltır. İçerisinde veri yoktur, sadece tabloların görünümleridir. Bir görünüm oluşturulduğunda temel olarak bir sorgu oluşturur ve sorguya bir ad atanır.

View Oluşturma (Create View)

Örnek:

```
CREATE VIEW kiralama_durumu_sorgulama AS
SELECT customer_id as Musteri_id , rental_id, customer.first_
name || ' ' || customer.last_name as Musteri_Adı ,
staff.first_name || ' '|| staff.last_name as Personel,
      rental_date,
      CASE
            WHEN rental_date > '2005-02-15'
            AND rental_date < '2005-05-15' THEN 'KİRALAMA AKTİF
DEĞİL'
            WHEN rental_date > '2005-05-15'
            AND rental_date < '2005-08-15' THEN 'GECİKMELİ
TESLİM - %0,2 CEZAİ İŞLEM'
            WHEN rental_date > '2005-08-15'
            AND rental_date < '2005-11-15' THEN 'GECİKMELİ
TESLİM - %0,1 CEZAİ İŞLEM'
            WHEN rental_date > '2005-11-15' THEN 'KİRALAMA
SÜRECİ DEVAM EDİYOR.'
      END as "Kiralama Durumu"
FROM rental INNER JOIN customer USING (customer_id)
INNER JOIN staff USING(staff_id)
ORDER BY rental_date DESC ;
```

Örnekler :

```
CREATE VIEW film_bilgisi AS
SELECT film_id, title, release_year, category.name FROM film
INNER JOIN film_category USING (film_id)
INNER JOIN category USING(category_id);

SELECT * FROM film_bilgisi ;
```

Sorguda filmlerin kategorisi, adı, yayınlandığı yıl gibi bilgilerini getiren `view` oluşturuldu.

```
CREATE VIEW horror_film AS
SELECT film_id, title, release_year, length
FROM film_info WHERE name = 'Horror';

SELECT * FROM horror_film ;
```

Sorgu ile korku filmi kategorisine ait filmleri listeleyen `view` oluşturuldu.

```
CREATE VIEW film_category_info AS
SELECT name, COUNT(category_id) FROM category
INNER JOIN film_category USING (category_id)
INNER JOIN film USING (film_id) GROUP BY name;
```

Sorgu ile film kategorilerine ait kaç film olduğunu listeleyen sorgu oluşturuldu. `COUNT(category_id)` ifadesi yerine `COUNT(film_id)` veya `COUNT(name)` ifadeleri yazılabilir.

```
CREATE VIEW film_length_info AS
SELECT name, SUM(length) film_length FROM category
INNER JOIN film_category USING (category_id)
INNER JOIN film USING (film_id)
GROUP BY name;

SELECT * FROM film_length_info ;
```

Sorgu kategorilerin toplam film uzunluğunu listeleyen view oluşturur.

View Silme

```
DROP VIEW film_length_info ;
DROP VIEW film_category_info  ;
DROP VIEW film_info  ;
```

Sorgu sonucu hata ile karşılaşırız. Silmek istediğimiz `view'a` bağlı başka `view` olduğu için hata mesajı döndü. Bu tür view'ları silmek için `CASCADE` ifadesi kullanılır.

```
DROP VIEW film_info  CASCADE ;
```

Sorgu sonucu dönen mesajda bu view'a bağlı olan horror_films view'ınında da silindiği belirtilir.

9
Indexes

- B-tree Index
- Hash Index
- Brin Index

PostgreSQL indeksleri, veritabanı performansını artırmak için etkili araçlardır. Indeksler, veritabanı sunucusunun belirli satırları indeksler olmadan yapabileceğinden çok daha hızlı bulmasına yardımcı olur.
Ancak indeksler, veritabanı sistemine yazma ve depolama ek yükleri ekler. Bu nedenle, bunları uygun şekilde kullanmak çok önemlidir.
Aşağıdaki yapı ile oluşturulur.

```
CREATE INDEX index_adi ON tablo_adi [USING method]
(
    kolon_adi [ASC | DESC] [NULLS {FIRST | LAST }],
    ...
);
```

PostgreSQL'de indexler, verilere erişimi hızlandırmak için tasarlanmış veritabanı nesneleridir. Her index `DROP index_adı` ifadesiyle silinebilir.
Bir index oluşturulurken o tablo kilitlenmektedir. Bu sebeple özellikle canlı (productiong) ortamlarda index oluşturulurken dikkat edilmesi gerekmektedir. Bu sorunun etkilerini en aza indirmek için `create index` ifadesine `concurrently` ifadesi eklenebilir. Ancak bu ifadeyle index create işlemi daha uzun sürecektir.
Index ifadeleri ile `unique` indexler de oluşturulabilir. Index oluşturulan kolona unique değerlerin girilmesini sağlar. Aksi halde kullanıcı hata alacaktır.
Indexlere "`where`" koşulu da eklenebilmektedir. `Partial index` olarak adlandırılan bu index sayesinde `where` koşulu ile belirtilen veriler indexlenir. Daha az alan tarandığı için de doğru kullanımı performansta artış sağlayabilmektedir.

B-Tree Index

Index oluşturulurken türü belirtilmezse, default olarak `btree` index türünde oluşturulur. Bu index türü özellikle `büyüktür, küçüktür, büyük eşittir, küçük eşittir, between, is null, is not null` gibi sorgular için kullanılmaktadır. B-tree'nin index satırları sayfalar halinde paketlenir. Çoklu kolon indexlemesi yapılabilir. Çoğu sorgu türü için de performans olarak en iyi seçenektir.

Örnek:

```
CREATE INDEX film_title_indx ON film(title) ;
CREATE INDEX title_id_indx ON film(title,film_id) ;
```

Hash Index

Flat yapıda tutulan index türüdür. Bu sebeple B-tree index'e göre daha az yer tutmaktadır. Ancak bu türde index create edilirken harcanan süre daha uzundur. Daha çok eşitlik alanında kullanılan sorgular için uygun olan bir index türüdür.
Unique constraintler ile kullanılamaz ve çoklu kolonlar için oluşturulamazlar. Bu index türü bir çok kısıtlamaya tabi olduğu için kullanırken dikkat edilmesi gerekir.

```
CREATE INDEX year_idx ON film USING HASH(release_year) ;
```

Brin Index

Bu indexleme türünde, verilerin tutulduğu bloklar içerisinde en küçük ve en büyük değerler dikkate alınır. Yani blok içerisinde sıralanmış olan tüm değerler değil, sadece minimum ve maximum değerler tutulmaktadır.
Brin, değerlerin tablodaki fiziksel konumlarıyla ilişkili olduğu sütunlar için performanslı çalışmaktadır. Belirli veriler tutulduğu için B-tree indexe göre daha az alan kaplamaktadır.
Bir veri aralığında üzerinde işlem yapılacaksa, doğru kullanıldığı takdirde çok performanslı çalışabilir. Veri aralığı içeren sorgulara uygun olduğundan big data ve veri analizi alanlarında uygun olmaktadır.

```
CREATE INDEX amount_brn_idx ON payment USING brin(amount) ;
```