2006-02-14

PostgreSQL ile Programlama

Volkan YAZICI

İçindekiler

Table of Contents

I. Giriş....4
II. C Arayüzü....6
1. libpq Kütüphanesinin Kurulumu....6
2. Tanıtım Uygulaması....7
3. Bağlantı Kurulumu....15
3.A. Senkron Bağlantı Kurulumu....15
3.B. Asenkron Bağlantı Kurulumu....17
4. Bağlantı Üzerinde İşlemler....18
4.A. Bağlantı Hakkında Bilgi Edinme....18
4.B. Bağlantı Üzerinden Sunucu Hakkında Bilgi Edinme....20
4.C. Bağlantı Üzerinde İşlemler....22
5. Sorgu İşletimi....24
5.A. Senkron Sorgu İşletimi....24
A.1. Parametre Kullanan Fonksiyonlar Hakkında....26
5.B. Asenkron Sorgu İşletimi....27
6. Sorgu İptali....29
7. Uygulama İçinde COPY Kullanımı....30
8. Sorgu Sonuçları Üzerinde İşlemler....34
8.A. Sonuç Hakkında Durum Bilgisi....34
8.B. Sonuç Hakkında Tablo/Satır/Sütun Bilgisi....35
8.C. Sonuç Üzerinde İşlemler....40
9. Asenkron Uyarılma....41
10. Uyarı Mesajlarının İşlenmesi....42
11. Büyük Boyutlu Nesneler....43
11.A. Neden LO?....44
11.B. LO Nasıl?....45
11.C. Kütüphane LO Fonksiyonları....46
11.D. Sunucu Taraflı LO Fonksiyonları....49
12. Diğer Fonksiyonlar....50
13. SSL Desteği....53
14. Şifre Dosyası Kullanımı....53
15. Çevre Değişkenleri....53
16. Programların Derlenmesi....54
17. Yiv Korunaklı Programlama....55
18. Fonksiyon Tablosu....57
19. Örnekler....58
III. PHP Arayüzü....59

1. PostgreSQL Desteğinin Etkileştirilmesi....59
2. Tanıtım Uygulaması....60
3. Bağlantı Kurulumu....63
4. Bağlantı Üzerinde İşlemler....63
4.A. Bağlantı Hakkında Bilgi Edinme....63
4.B. Sunucu Hakkında Bilgi Edinme....65
4.C. Kurulu Bağlantı Üzerinde İşlemler....65
5. Sorgu İşletimi....66
5.A. Senkron Sorgu İşletimi....66
6. Asenkron Sorgu İşletimi....67
6.A. COPY Komutu Kullanımı....69
6.B. İşletilen Sorgunun İptal Edilmesi....70
6.C. Diğer Sorgu Fonksiyonları....71
7. Sorgu Sonuçları Üzerinde İşlemler....71
7.A. Sonuç Hakkında Durum Bilgisi....71
7.B. Sonuç Hakkında Tablo/Satır/Sütun Bilgisi....72
8. Veri Alımı....73
9. Büyük Boyutlu Nesneler....75
10. Diğer Fonksiyonlar....77
11. Çalışma Anı (INI) Ayarları....79
12. Fonksiyon Tablosu....80
IV. Python Arayüzü....82
1. Neden psycopg?....82
2. psycopg Kurulumu....83
3. Tanıtım Uygulaması....85
4. Modül Arayüzü....89
5. Bağlantı Nesneleri....90
6. İmleç Nesneleri....91
7. Veri Tipi Geçişleri....93
8. Genişletilmiş DB-API Özellikleri....94
9. Standart Olmayan psycopg Özellikleri....95
9.A. Veri Tipleri....95
9.B. İstemci Karakter Kodlaması....95
9.C. Uyarı Mesajları....96
9.D. İzolasyon Seviyesi....96
9.E. İmleç Fonksiyonları....96
9.F. Diğer Özellikler....97
V. Programlama ve Güvenlik....98
1. Güvenli Programlama....98
1.A. C Programlama Dili....98
1.B. PHP Programlama Dili....99
1.C. Python Programlama Dili....99
1.D. Dönen Sonuçların Kontrolü....99
1.E. Güvenilmeyen Kaynaklı Verilerin Kontrolü....100
E.1. SQL Injection Saldırıları....100
1.F. API Güvenliği....102
F.1. C Arayüzü....102
F.2. PHP Arayüzü....102
F.3. Python Arayüzü....103
VI. Sık Karşılaşılan Problemler....104
1. Genel....104
2. C Arayüzü....117

3. PHP Arayüzü....117
4. Python Arayüzü....120

I. Giriş

PostgreSQL, diğer bir çok gelişmiş veritabanından olduğu gibi dışarıdan kendisine bağlanan istemciler ile belirli bir dilde konuşup anlaşabilmek için bir Sunucu/İstemci Protolüne[1] sahiptir. Herhangi bir istemci bu mimarinin getirdiği standartlar doğrultusunda veritabanı üzerinde protokolce desteklenen tüm sorgulama işlemlerini gerçekleştirebilir. Bunun için yapılması gereken tek şey veritabanı sunucusu ile kurulacak bir soket bağlantısı üzerinden basit `read`/`write` sistem çağrılarında bulunmaktan ibaret olacaktır. Fakat hiçbir kullanıcı - özel amaçlar doğrultusunda sıfırdan bir kütüphane geliştirmediği sürece - PostgreSQL'in bu iç işleyişi ile uğraşmak istemez. İşte bu noktada devreye *API* (*Application Programming Interface*/Uygulama Programlama Arayüzü) kavramı girmektedir.

PostgreSQL, dışarıdan kendisine bağlanacak programlama dilleri için bir ara katman oluşturan uygulama arayüzlerine sahiptir. Bu sayede, veritabanına ulaşmak isteyen programcılar, PostgreSQL'in kendi iç işleyişi ile ilgili hiçbir çalışma mekanizmasından haberdar olmak zorunda kalmadan da, kullandıkları dilin kendilerine sundukları *API* kütüphanesinden faydalanarak kolaylıkla veritabanı ile etkileşime geçip, ilgili sorgulamalarını gerçekleştirebilirler.

Yazılım piyasasındaki neredeyse tüm popüler programlama dilleri PostgreSQL veritabanı için bir programlama arayüzünü beraberinde sunmaktadır. Bu programlama dillerinden ve bunların beraberinde sağladıkları *API*'lerin birkaçını şu şekilde listeleyebiliriz:

Programlama Dili	**Sağladığı API**
C	libpq
C++	libpq++, libpqxx
Common Lisp	Pg, pg-dot-lisp
Java	pgjdbc
Lua	LuaSQL
ODBC Kullanan	psqlODBC
.NET	Npgsql
Perl	pgperl, DBD-Pg
PHP	PostgreSQL PHP Modülü
Python	PyGreSql, psycopg
Ruby	PQA
Scheme	SQLI/SQLD
Tcl	pgtclng, pgtcl

Listelenen arayüzlerin kendileri de çoğunlukla C ile geliştirildiklerinden dolayı, çok büyük bir kısmı veritabanı ile aralarında ara katman olarak libpq kütüphanesini kullanmaktadırlar. Bu nedenle yaptıkları, veritabanı ile doğrudan etkileşime geçmekten çok, *API* olarak sunuldukları programlama dilinden gelen istekleri, libpq kütüphanesi

1 PostgreSQL Sunucu/İstemci Protokolü bu kitabın konusu dışına taştığından dolayı burada işlenmeyecektir. Arzu edenler ayrıntılı bilgi için PostgreSQL'in resmi dökümantasyonundan yararlanabilirler.

aracılığı ile veritabanına ulaştırmaktan ibarettir. Bu göz önüne alındığında, sonraki bölümlerde inceleyeceğimiz programlama dilleri arayüzlerinin sundukları fonksiyonlar, libpq kütüphanesindeki bir çok fonksiyonun aslında karşılığı olup, aldıkları parametreden yaptığı işlere kadar bir çok yerde benzerlik gösterecektir.

PostgreSQL veritabanı ile etkileşime geçmenin elbette tek yolu herhangi bir programlama dilini kullanarak *API*'ler aracılığı ile veritabanına bağlantı kurmak değildir. Elbette kimse basit bir sorgulama işlemi için, yüzlerce satır kod yazmak istemez. Bu nedenle `psql`'in (PostgreSQL komut satırı işlemcisi) ve *GUI*'ye (*Graphical User Interface*/Grafiksel Kullanıcı Arayüzü) daha bir çok PostgreSQL istemcisinin[2] varlığından tüm programcıları, burada son bir kez haberdar etmek isteriz.

Bu kitapta yukarıdaki tabloda listelenen programla dillerinin hepsinden bahsedilmek yerine, kendimizce gerekli ve şuan için en azından yeterli gördüğümüz PHP, Python ve - olmazsa olmaz - C dillerinin PostgreSQL bağlantı arayüzleri incelenmeye çalışılacaktır.

Programlama arayüzlerinin incelenmesine geçmeden önce burada küçük fakat önemli bir hatırlatmayı yapmak yerinde olacaktır: Bir programlama dilinin uygulama arayüzünden bahsedilirken, okuyucunun o programlama dili hakkında belirli düzey bir bilgi sahibi olduğu varsayılıp, kullanılacak olan temel sistem ve *nix terimleri hakkında da genel bir birikimi olduğu kabulü ile yola çıkılacaktır.

2 İstemciler hakkında ayrıntılı bilgi için kitabın PostgreSQL istemcileri ile ilgili kısmına bakabilirsiniz.

II. C Arayüzü

libpq kütüphanesi, programcıların PostgreSQL veritabanına bağlanıp, sorgulamalarını gerçekleştirdikten sonra dönen sonuçlar üzerinde işlem yapmalarını sağlayan PostgreSQL'in C programlama dili arayüzüdür.

libpq, PostgreSQL veritabanının resmi C programlama arayüzü olup, PostgreSQL iletişim protokolünün sağladığı tüm özellikleri eksiksiz olarak sunan tek kütüphanedir ve yazılımın geliştirimi de yine PostgreSQL takımı tarafından üstlenilmiştir. Bu nedenle PostgreSQL'e arayüz sağlayan diğer bir çok programlama dili, libpq kütüphanesini kendilerine ikinci bir ara katman olarak kullanmaktadırlar.

1. libpq Kütüphanesinin Kurulumu

PostgreSQL kaynak kodunu derleyerek gerçekleştirilecek bir kurulumda, libpq öntanımlı olarak veritabanı ile birlikte sisteme yüklenmektedir. Bunun ile beraber, çeşitli dağıtımlar tarafından sağlanan kütüphane ikili paketlerini şu şekilde listeleyebiliriz:

Paket Biçimi	Başlık Dosyaları	Kütüphane Dosyaları
DEB (Linux)	postgresql-dev	libpq3
RPM (Linux)	postgresql-devel	postgresql-libs

libpq kütüphanesi kullanan programların derlenmesi ve derlenmiş programların kütüphane dosyaları ile bağlanması için libpq başlık ve kütüphane dosyalarına gerek duyulmaktadır. Bu başlık ve kütüphane dosyalarının sisteminizde kurulu olup olmadığını anlamak için aşağıdaki adımları izleyebilirsiniz.

İlk iş olarak, PostgreSQL'in C başlık dosyalarının yüklendiği dizin içerisinde `libpq-fe.h` dosyasının olup olmadığına bakabilirsiniz. PostgreSQL'in sisteme kurulumu ile birlikte gelen `pg_config`[3] programını bu iş için kullanacak olursak:

```
$ pg_config –includedir
/usr/local/pgsql/include
```

PostgreSQL başlık dosyalarının nereye yükleneceğini, kurulum esnasında `./configure` betiğine `--includedir` parametresi ile belirtebilirsiniz. (Bu dizin PostgreSQL tarafından öntanımlı olarak `/usr/local/pgsql/include` olarak alınacaktır.)

Derlenen programın kütüphaneler ile bağlanması esnasında gerekecek olan libpq kütüphane dosyası için ise, PostgreSQL'in kütüphane dosyalarının bulunduğu dizin altında `libpq.so.x` dosyasının yer alıp almadığına bakabilirsiniz. Yine yukarıdakine benzer şekilde `pg_config` programını kullanarak kütüphane dosyalarının yüklendiği dizini şu şekilde öğrenebilirsiniz:

```
$ pg_config –libdir
/usr/local/pgsql/include
```

Kütüphane dosyalarının nereye yükleneceğini `./configure` betiğine `--libdir` parametresi ile

3 `pg_config` hakkında ayrıntılı bilgi için programa `pg_config –help` komutunu çalıştırmanız yeterli olacaktır.

bildirebilirsiniz. (PostgreSQL tarafından öntanımlı olarak bu dizin `/usr/local/pgsql/lib` olarak alınacaktır.)

2. Tanıtım Uygulaması

Kütüphane tarafından sağlanan fonksiyonların ve kütüphanenin sunduğu diğer özelliklerin ayrıntılı olarak incelenmesine geçmeden önce, okuyucunun konu hakkında genel bir bakış açısı kazanması açısından libpq kütüphanesini kullanan basit bir veritabanı uygulamasını burada vereceğiz.

Örnek olarak elimizde bir firmanın ürün stoklarının tutulduğu veritabanı tablosu olduğunu varsayalım. Yazacağımız program ile bu tabloya ürün girişini gerçekleştirip, isteğe bağlı olarak da alımını sağlayacağız. Genelde veritabanı kullanan yazılımların ihtiyaçları bu kadar basit olmasa da, konunun ana hatlarının açık olarak irdelenmesi için bu şekilde yalın bir tasarım izleyeceğiz.

İlk önce ilgili kayıtları tutacak olan veritabanı tablomuzu oluşturuyoruz:

```
CREATE SEQUENCE urunler_urun_seq;
CREATE TABLE urunler (
  urun    bigint  DEFAULT nextval('urunler_urun_seq') PRIMARY KEY,
  model   text    NOT NULL UNIQUE,
  adet    integer NOT NULL
);
CREATE UNIQUE INDEX urunler_urun_idx ON urunler (urun);
```

Tabloda yer alacak ürünleri `ID` değerleri ile diğer tablolardan çağıracağız. Model isimleri ise, tabloda aynı modelden iki adet olmayacak şekilde yerleştirilecek.

Tablomuzu oluşturduktan sonra, ürün ekleme ve çıkarma işlemlerinde gerek kullanıcıdan gelecek verinin, gerekse sorgu sonucu dönecek cevapların belirli kontrollerden geçirilmesi gerekmektedir. Bu denetlemeleri kendi yazacağımız istemci uyulamalarında teker teker yapmaktansa, bunları bizim yerimize gerçekleştirecek prosedüler kullanacağız.

Prosedürlerin döndürecekleri değerler ve bunu alacak istemcilerin doğru bir şekilde anlaşabilmesi için standart bir yapı izleyeceğiz. Şöyle ki, prosedür çağrısı sonucu dönen değerin ilk 8 biti son durumu, bundan sonraki bitler ise ilgili ek bilgiyi içerecek.

Durum Kodu	**Açıklama**
`1 << 0`	İşlem başarı ile gerçekleşti. (Ek bilgi: Mevcut ürün adeti ya da *`ID`* değeri.)
`1 << 1`	Parametre olarak girilen adet pozitif bir tamsayı olmalı.
`1 << 2`	Eklenmek istenen model kayıtlarda mevcut. (Ek bilgi: Mevcut ürünün *`ID`* değeri.)
`1 << 3`	İstenen model kayıtlarda mevcut değil.
`1 << 4`	İstenen adeti karşılayacak kadar kayıt mevcut değil. (Ek bilgi: Mevcut adet sayısı.)
`1 << 5`	Beklenmeyen bir hata oluştu.
`1 << 6,7`	İleride kullanılmak üzere ayrıldı.

Şimdi bu bilgiler doğrultusunda, prosedürlerimizi oluşturmaya geçebiliriz.

```
CREATE FUNCTION urun_ekle(yeni_model text, yeni_adet integer) RETURNS integer AS $$
DECLARE
  kayit   record;
  urun    integer;
BEGIN
  IF yeni_adet < 1 THEN
    RETURN 1 << 1;
  END IF;

  BEGIN
    INSERT INTO urunler (model, adet) VALUES (yeni_model, yeni_adet);
```

```
    SELECT INTO urun currval('urunler_urun_seq');
  EXCEPTION
    WHEN unique_violation THEN
      SELECT INTO kayit urun FROM urunler WHERE model = yeni_model;
      -- Ek bilgi: Mevcut ürünün ID değeri.
      RETURN (1 << 2) + (kayit.urun << 8);

    WHEN OTHERS THEN
      RETURN (1 << 5);
  END;

  -- Ek bilgi: Ürün ID değeri.
  RETURN (1 << 0) + (urun << 8);
END;
$$ LANGUAGE plpgsql STRICT;
```

Kitabın ilgili bölümlerinde PL/pgSQL dili izah edilmiş olsa da, yukarıdaki ufak prosedürün ne yaptığı hakkında burada okuyucuyu bir kez daha bilgilendirmek yerinde olacaktır. Ürünler tablomuza yapacağımız herhangi bir yeni ürün eklemesi için yukarıdaki `urun_ekle()` prosedürünü çağıracağız. `urun_ekle()` bizim için ürün adedinin pozitif bir tamsayı olup olmadığına bakıp, daha önceden böyle bir ürünün varlığını kontrol edecektir. Ve en son olarak döndüreceği değere ilgili ek bilgiyi yerleştirerek bizi son durumdan haberdar edecektir. Bu yolla, yazdığımız her programda bu kontrolleri kendimiz yapmak yerine işi ilgili prosedüre bırakacağız.

Benzer şekilde, `urun_cikar()`, `adet_ekle()` ve `adet_cikar()` prosedürlerini de geliştirecek olursak:

```
--
-- Belirtilen ID'ye sahip ürünü tablodan sil.
--
CREATE FUNCTION urun_cikar(istenen_urun bigint) RETURNS integer AS $$
DECLARE
  etkilenenler integer;
BEGIN
  DELETE FROM urunler WHERE urun = istenen_urun;
  GET DIAGNOSTICS etkilenenler = ROW_COUNT;

  IF etkilenenler < 1 THEN
    RETURN 1 << 3;
  END IF;

  RETURN 1 << 0;
END;
$$ LANGUAGE plpgsql STRICT;

--
-- Girilen ID'ye sahip ürününün adetini belirtilen miktar kadar arttır.
--
CREATE FUNCTION adet_ekle(istenen_urun bigint, yeni_adet integer) RETURNS integer AS $$
DECLARE
  kayit       record;
  son_adet    integer;
BEGIN
  IF yeni_adet < 1 THEN
    RETURN 1 << 1;
  END IF;

  SELECT INTO kayit adet FROM urunler WHERE urun = istenen_urun;
  IF NOT FOUND THEN
    return 1 << 3;
  END IF;

  son_adet = kayit.adet + yeni_adet;
  UPDATE urunler SET adet = son_adet WHERE urun = istenen_urun;

  -- Ek bilgi: Mevcut ürün adeti.
  RETURN (1 << 0) + (son_adet << 8);
```

```
END;
$$ LANGUAGE plpgsql STRICT;

--
-- Girilen ID'ye sahip ürünün adetini istenilen miktar kadar azalt.
--
CREATE FUNCTION adet_cikar(istenen_urun bigint, istenen_adet integer) RETURNS integer AS
$$
DECLARE
  kayit       record;
  son_adet    integer;
BEGIN
  IF istenen_adet IS NULL OR istenen_adet < 1 THEN
    RETURN 1 << 1;
  END IF;

  SELECT INTO kayit adet FROM urunler WHERE urun = istenen_urun;
  IF NOT FOUND THEN
    RETURN 1 << 3;
  END IF;

  IF kayit.adet < istenen_adet THEN
    -- Ek bilgi: Mevcut ürün adeti.
    RETURN (1 << 4) + (kayit.adet << 8);
  END IF;

  son_adet = kayit.adet - istenen_adet;
  UPDATE urunler
    SET adet = son_adet
    WHERE urun = istenen_urun;

  -- Ek bilgi: Kalan ürün adeti.
  RETURN (1 << 0) + (son_adet << 8);
END;
$$ LANGUAGE plpgsql;
```

Tablomuzu ve denetleme işlemlerini gerçekleştirecek prosedüleri oluşturduğumuza göre, yazacağımız programın ana taslağı konusunda artık konuşabiliriz.

```
/* API tarafından sağlanan bağlantı ve sorgu sonucu yapıları. */
PGconn   *bag;
PGresult *sonuc;

bag = PQconnectdb(...); /* Veritabanına bağlanılıyor. */

while (MenuyuEkranaBas() && KullaniciGirdisiniOku())
{
    switch (KullaniciGirdisi)
    {
        case URUNLERI_LISTELE:
            sprintf(komut, "SELECT urun, model, adet FROM urunler");
            sonuc = PQexec(bag, komut);
            if (PQresultStatus(sonuc) == PGRES_TUPLES_OK)
            {
                satir_sayisi = PQntuples(sonuc);
                sutun_sayisi = PQnfields(sonuc);
                for ( i = 0; i < satir_sayisi; i++ )
                    for ( j = 0; j < sutun_sayisi; j++ )
                        printf("[%d][%d]: %s\n", i, j, PQgetvalue(sonuc, i, j));
                }
            }
            else if (PQresultStatus(sonuc) == PGRES_COMMAND_OK)
                /* Sorgu başarı ile gerçekleşti fakat hiçbir satır dönmedi. */
            else
                /* Beklenmeyen bir durum oluştu. */
            break;

        case URUN_EKLE:
            scanf("%s", model);
```

```
            scanf("%d", &adet);
            sprintf(komut, "SELECT urun_ekle('%s', %d)", model, adet);
            sonuc = PQexec(bag, komut);
            ...
    }
}
```

Yukarıdan da anlaşılabileceği üzere programa girdiğimizde veritabanına bağlandıktan sonra, kullanıcıya bir menü sunup, ona göre veritabanı üzerinde ilgili sorgulamaları gerçekleştiriyoruz ilk olarak.

`main()` fonksiyonunu yazmaya başlamadan önce, işimize yarayacak bir kaç genel değişken tanımlıyoruz:

```
/* Karakter dizileri için maksimum büyüklük. */
#define MAX_TAMPON      1024

/*
 * Prosedürlerden dönecek cevapların tanımları.
 * (İlk 8 bitten sonrası ekbilgi olarak işlenecek.)
 */
#define TAMAM           1 << 0
#define YANLIS_ADET     1 << 1
#define KAYIT_MEVCUT    1 << 2
#define KAYIT_YOK       1 << 3
#define ADET_YETERSIZ   1 << 4
#define BILINMEYEN_HATA 1 << 5

/* Menü işlemlerini numaralandırıyoruz. */
enum
{
    URUN_LISTELE = 1,
    URUN_EKLE,
    URUN_CIKAR,
    ADET_EKLE,
    ADET_CIKAR,
    PROG_CIKIS
};
```

`main()` içini fazla kalabalık tutmaktansa, yapacağımız işleri şu şekilde fonksiyonlara bölelim:

```
/* Beklenmedik bir durumuda bağlantıyı kapatıp hata kodu ile çıkılacak. */
void
Cik(PGconn *bag)
{
    if (PQstatus(bag) != CONNECTION_OK)
        fprintf(stderr, "Hata mesajı:\n%s", PQerrorMessage(bag));

    PQfinish(bag);
    exit(1);
}
```

Artık yavaş yavaş kullandığımız libpq tip ve fonksiyonlarından bahsedebiliriz.

- `PGconn`: Veritabanı ile kurulacak bağlantı esnasında kullanılacak olan kütüphane yapısı.
- `ConnStatusType PQstatus(PGconn *)`: Girilen bağlantı yapısının son durumunu kendisini çağırana döndürecektir.
- `char *PQerrorMessage(PGconn *)`: Parametre olarak aldığı `PGconn` yapısının işaret ettiği bağlantıda herhangi bir hata gerçekleşmiş ise, bunun ile ilgili hata mesajını kendisini çağırana döndürecektir.
- `void PQfinish(PGconn *)`: Veritabanı ile kurulu olan bağlantının kapatılması için kullanılır. Ayrıca, `PGconn` yapısı tarafından tutulan bellek alanını da serbest bırakacaktır.

```
/*
 * Veritabanı ile bağlantı kurulumunu gerçekleştirecek fonksiyon.
 */
PGconn *
BaglantiKur(const char *ozellikler)
{
    PGconn *bag;

    bag = PQconnectdb(ozellikler);
    if (PQstatus(bag) != CONNECTION_OK)
        Cik(bag);

    return bag;
}
```

- `PGconn *PQconnectdb(const char *)`: Parametre olarak aldığı karakter katarının belirttiği özellikler doğrultusunda veritabanı ile bağlantı kurmaya çalışır. Başarılı olduğu durumda, kendisini çağırana bağlantının tutulduğu `PGconn` yapısını döndürecektir.

```
/*
 * Parametre olarak aldığı prosedürlerden dönen sorgu sonucunu durumuna
 * göre değerlendirip, cevabı kendisine çağırana iletecek olan fonksiyon.
 * (En sonda PQclear() ile sonucu bir daha kullanılmamak üzere bellekten
 * bırakacaktır.)
 */
int
ProsedurKontrol(PGresult *sonuc)
{
    int durum;

    if (PQresultStatus(sonuc) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "  Sorgu sonucu beklenmeyen bir hata oluştu!\n");
        fprintf(stderr, "  Hata mesajı:\n%s", PQresultErrorMessage(sonuc));

        durum = 0;
    }
    else
        durum = atoi(PQgetvalue(sonuc, 0, 0));
    PQclear(sonuc);

    return durum;
}
```

- `ExecStatusType PQresultStatus(PGresult *)`: Parametre olarak aldığı sonuç yapısının durumunu döndürür.
- `char *PQgetvalue(const PGresult *, int, int)`: Girilen `PGresult` sonuç yapısının berlitilen satır ve sütününa karşılık gelen alanı, kendisini çağırana karakter katarı olarak döndürür.
- `void PQclear(PGresult *)`: Belirtilen sonuç yapısı tarafından alıkonulan bellek alanını serbest bırakır.

```
/* Kullanıcıdan girdi almak için kullanacağımız fonksiyon. */
int
SatirOku(char *tampon, size_t uzunluk, int isnumeric)
{
    char *c = tampon;

    fgets(tampon, uzunluk, stdin);

    /* Sondaki yeni satır karakteri (\n) siliniyor. */
    while (uzunluk-- && *c != '\0')
        ++c;
    if (*(c-1) == '\n')
        *(c-1) = '\0';
```

```
    return (isnumeric) ? atoi(tampon) : 0;
}

/* Kullanıcı menüsünü ekrana basacak olan fonksiyon. */
int
Menu(void)
{
    int         girdi;
    static char gecici[MAX_TAMPON];

    while (1)
    {
        printf("\nAşağıdaki menüden yapmak istediğiniz işlemi seçiniz:\n");
        printf("\t%d  Ürünleri Listele\n", URUN_LISTELE);
        printf("\t%d  Ürün Ekle\n", URUN_EKLE);
        printf("\t%d  Ürün Çıkar\n", URUN_CIKAR);
        printf("\t%d  Adet Ekle\n", ADET_EKLE);
        printf("\t%d  Adet Çıkar\n", ADET_CIKAR);
        printf("\t%d  Programdan Çıkış\n", PROG_CIKIS);

        printf("Girdi: ");
        girdi = SatirOku(gecici, MAX_TAMPON, 1);
        if (girdi < URUN_LISTELE || girdi > PROG_CIKIS)
        {
            fprintf(stderr, "Yanlış menü numarası!\n");
            continue;
        }
        break;
    }

    return girdi;
}

/*
 * Prosedürlerden dönen kodların açıklamasını ekrana yazıp,
 * duruma göre ilgili ek bilgiyi geri döndürecek olan fonksiyon.
 */
int
DurumAciklama(int durum)
{
    int ekbilgi = -1;

    if (!durum)
        printf("Hata oluştu!\n");
    else if ((durum&TAMAM))
    {
        printf("Sorgu başarı ile gerçekleşti.\n");
        ekbilgi = durum >> 8;
    }
    else if ((durum&YANLIS_ADET))
        printf("Girilen adet pozitif bir tamsayı olmalı!\n");
    else if ((durum&KAYIT_MEVCUT))
    {
        printf("Eklenmek istenen model zaten mevcut.\n");
        ekbilgi = durum >> 8;
    }
    else if ((durum&KAYIT_YOK))
        printf("Böyle bir ürün mevcut değil!\n");
    else if ((durum&ADET_YETERSIZ))
    {
        printf("Mevcut ürün sayısı istenen adetin altında.\n");
        ekbilgi = durum >> 8;
    }

    return ekbilgi;
}
```

Bit bazında değişkenler kullanıldığı zaman herhangi bir değerin kontrol edilmesi esnasında olası `OR` işlemlerine karşı `(A&B) == B` kullanılmalıdır. Fakat yazdığımız bu ufak programda, değişkenlerin herhangi bir OR işlemine tabi tutulması durumu söz konusu olmadığından, `(A&B)` bizim için yeterlidir.

Yardımcı fonksiyonları şimdilik bunlar ile sınırlı tutacağız. Artık `main()` fonksiyonunu yazabiliriz:

```
int
main(void)
{
    PGconn      *bag;
    PGresult    *sonuc;
    int          menu, urun, adet, durum, ekbilgi;
    char         tampon[MAX_TAMPON], komut[MAX_TAMPON];
    int          i, j;

    bag = BaglantiKur("dbname=test");

    while ((menu = Menu()))
    {
        switch (menu)
        {
            case URUN_LISTELE:
                printf("Ürünler listeleniyor:\n");
                sonuc = PQexec(bag, "SELECT urun, model, adet FROM urunler");
                durum = PQresultStatus(sonuc);
                if (durum == PGRES_TUPLES_OK)
                {
                    i = PQntuples(sonuc);
                    if (!i)
                        printf("  Tabloda şuan hiçbir kayıt bulunmuyor.\n");
                    else
                    {
                        printf("   Ürün (ID) | Model           | Adet\n");
                        printf("  -----------+-----------------+------\n");
                        for (j = 0; j < i; j++)
                            printf("   %9s | %15s | %s\n",
                                   PQgetvalue(sonuc, j, 0),
                                   PQgetvalue(sonuc, j, 1),
                                   PQgetvalue(sonuc, j, 2));
                    }
                }
                else
                {
                    fprintf(stderr, "Sorgu sonucunda beklenmeyen bir cevap döndü!\n");
                    fprintf(stderr, "Sorgu durumu: %s\n",
                            PQresStatus(PQresultStatus(sonuc)));
                    fprintf(stderr, "İlgili hata mesajı: %s",
                            PQresultErrorMessage(sonuc));
                }
                PQclear(sonuc);
                break;

            case URUN_EKLE:
                {
                    size_t  max_tampon = MAX_TAMPON / 2;
                    char   *tampon1 = tampon;
                    char   *tampon2 = tampon + max_tampon;

                    printf("Yeni ürün ekleniyor:\n");
                    printf("  Ürün modelini girin: ");
                    SatirOku(tampon1, max_tampon, 0);
                    printf("  Ürün adetini girin : ");
                    adet = SatirOku(tampon2, max_tampon, 1);
                    sprintf(komut, "SELECT urun_ekle('%s', %d)", tampon1, adet);
                    sonuc = PQexec(bag, komut);
                    durum = ProsedurKontrol(sonuc);
                    printf("  Son durum          : ");
```

```
                    ekbilgi = DurumAciklama(durum);
                    if (ekbilgi != -1)
                        printf("  Ürün ID değeri     : %d\n", ekbilgi);
                }
                break;

            /* URUN_CIKAR, ADET_EKLE, ADET_CIKAR buraya gelecek.*/

            case PROG_CIKIS:
                printf("Programdan çıkılıyor...\n");
                PQfinish(bag);
                return 0;
        }
    }

    return 0;
}
```

- `PGresult *PQexec(PGconn *, const char *)`: Parametre olarak aldığı SQL komut katarını, belirtilen bağlantı üzerinde sorgulayıp sonucunu kendisini çağırana döndürür.
- `int PQntuples(const PGresult *)`: Belirtilen sonucun, kaç satırdan oluştuğunu kendisini çağırana döndürecektir.
- `char *PQresStatus(ExecStatusType)`: Girilen sonuç durumunun değişken adını karakter katarı olarak döndürecektir. (`PGRES_COMMAND_OK` ya da `PGRES_TUPLES_OK` gibi.)

Programı elimizden geldiğince yalın tutmaya çalışsak da, okuyucunun kafasında soru işareti kalmaması açısından bazı durumlarda ayrıntıdan kaçınmadık. Fakat yine de, taslak olarak işleyişin yüzeysel de olsa anlaşılabileceğini düşünüyoruz.

> *Unutulmamalıdır ki yukarıdaki örnek, işleyişi esnasında bir çok noktada `Segmentation Fault` hatası vermeye açık olup, SQL Injection saldırılarına karşı tamamen savunmasızdır. Kendi yazacağınız programlarda, tasarım açısından bu taslağa bağlı kalmanızda hiçbir sakınca olmamasına rağmen, karakter katarlarının kullanıcıdan alınıp sorgu içine dahil edilmesinde kesinlikle bu yöntemler kullanılmamalıdır.*

Programın çalışmasına ilişkin örnek bir çıktıyı buraya yerleştiriyoruz:

```
Aşağıdaki menüden yapmak istediğiniz işlemi seçiniz:
        1  Ürünleri Listele
        2  Ürün Ekle
        3  Ürün Çıkar
        4  Adet Ekle
        5  Adet Çıkar
        6  Programdan Çıkış
Girdi: 1
Ürünler listeleniyor:
  Şu an kayıtlı hiçbir ürün bulunmamakta.

Aşağıdaki menüden yapmak istediğiniz işlemi seçiniz:
        1  Ürünleri Listele
        2  Ürün Ekle
        3  Ürün Çıkar
        4  Adet Ekle
        5  Adet Çıkar
        6  Programdan Çıkış
Girdi: 2
Yeni ürün ekleniyor:
  Ürün modelini girin: Murat 124
  Ürün adetini girin : 24
  Son durum          : Sorgu başarı ile gerçekleşti.
  Ürün ID değeri     : 1

Aşağıdaki menüden yapmak istediğiniz işlemi seçiniz:
        1  Ürünleri Listele
        2  Ürün Ekle
        3  Ürün Çıkar
```

```
        4  Adet Ekle
        5  Adet Çıkar
        6  Programdan Çıkış
Girdi: 1
Ürünler listeleniyor:
   Ürün (ID) | Model           | Adet
  -----------+-----------------+------
           1 |       Murat 124 | 24

Aşağıdaki menüden yapmak istediğiniz işlemi seçiniz:
        1  Ürünleri Listele
        2  Ürün Ekle
        3  Ürün Çıkar
        4  Adet Ekle
        5  Adet Çıkar
        6  Programdan Çıkış
Girdi: 5
Tablodan ürün alınıyor:
  Ürün ID değerini girin: 1
  İstenen adeti girin   : 25
  Son durum             : Mevcut ürün sayısı istenen adetin altında.
  Kalan ürün adeti      : 24

Aşağıdaki menüden yapmak istediğiniz işlemi seçiniz:
        1  Ürünleri Listele
        2  Ürün Ekle
        3  Ürün Çıkar
        4  Adet Ekle
        5  Adet Çıkar
        6  Programdan Çıkış
Girdi: 4
Tabloya adet ekleniyor:
  Ürün ID değerini girin: 1
  Ürün adetini girin    : 100
  Son durum             : Sorgu başarı ile gerçekleşti.
  Yeni ürün adeti       : 124

Aşağıdaki menüden yapmak istediğiniz işlemi seçiniz:
        1  Ürünleri Listele
        2  Ürün Ekle
        3  Ürün Çıkar
        4  Adet Ekle
        5  Adet Çıkar
        6  Programdan Çıkış
Girdi: 6
Programdan çıkılıyor...
```

3. Bağlantı Kurulumu

Bu bölümde libpq kütüphanesini kullanarak PostgreSQL veritabanına nasıl bağlanabileceğimiz üzerinde durmaya çalışacağız.

PostgreSQL tarafından sağlanan bağlantı metodları senkron ve asenkron olmak üzere ikiye ayrılmaktadır. Senkron bağlantı kurulum fonksiyonları, sunucu ile istemci arasında kurulmasını istediğimiz bağlantı bir sonuca ulaşana kadar programın beklemesine sebebiyet verecek türde yapılanmalardır. Fakat bir veritabanı bağlantısının kurulacağı gelişmiş uygulamalarda çoğu zaman ilk önce veritabanı bağlantısını sağlayacak olan fonksiyon ayrı bir yiv (*thread*) olarak artalan da çalışmaya bırakılır ve periyodik kontroller ile durumu gözetilirken, programın geri kalan kısmı kendi akışında ilerlemeye devam eder. İşte libpq tarafından sağlanan asenkron bağlantı fonksiyonları da bize, senkron eşleri karşısında bu esnekliği sağlamaktadır. Fakat asenkron mantığın çoğu programda kullanımı bir çok kontrolü ve bunun ile doğru orantılı olarak sırf bağlantı kurulumu için daha karmaşık bir yapıyı gerektirdiğinden, çoğu yazılımda senkron bağlantı kurulumu tercih edilir.

Senkron bağlantı fonksiyonlarını biz kendi senkron bağlantılarımızda kullanacak olsak da, bunların herbiri aslında kendi içinde asenkron bağlantı fonksiyonlarını bizim yerimize kullanan başka fonksiyonlardan ibarettir. Daha net olarak irdeleyecek olursak, senkron bağlantı fonksiyonları, asenkron bağlantı fonksiyonlarını kullanarak bir kaç komut çağrısı sonucunda yapılabilecek bir işi, bizim için tek bir komutla yapılabilecek hale indirgerler.

3.A. Senkron Bağlantı Kurulumu

Bu bölümde veritabanı ile nasıl senkron bir bağlantı kurulumu gerçekleştirebileceğimiz ve bunların nasıl kapatılabileceği üzerinde durmaya çalışacağız.

```
PGconn *
PQconnectdb(const char *baglanti_secenekleri);
```

`PQconnectdb()` fonksiyonu, parametre olarak aldığı bağlantı seçenekleri doğrultusunda veritabanı ile bağlantı kurulmasını sağlar. Bağlantı seçenekleri genelde `değişken = değer` biçimindedir. (`=' (eşittir) işaretinden önce ve sonraki boşlukların bir önemi yoktur.) Bu seçenekler birbirlerinden boşluklar ile ayrılabilir olup, değer kısmında kullanılan seçenek boşluk içeriyorsa da, `anahtar = 'deger'` şeklinde tek tırnak içinde yazılabilir. Ek olarak, bağlantı seçenekleri içinde kullanacağınız değerler tek tırnak ya da ters bölme çizgisi içeriyorsa, bunu o karakterin önüne bir ters bölme çizgisi koyarak (o karakteri ayıklayarak) belirtmelisiniz. Bağlantıda belirtmediğiniz seçenekler için, seçeneklerin öntanımlı değerleri kullanılacaktır.

`PQconnectdb()` fonksiyonu ile birlikte kullanabileceğimiz bağlantı seçeneklerini şu şekilde listeleyebiliriz:

`host`	Bağlanacağınız sunucunun adresini bildirmek için kullanılır. Herhangi bir değer belirtilmediği taktirde, Unix soket yapısına sahip sistemlerde /tmp dizini[4] altındaki sokete, aksi halde *TCP/IP* kullanılınarak yerel sisteme (`localhost`) bağlanılmaya çalışılınır. Gireceğiniz değer bir bölme işareti ile başlıyorsa, protokol olarak *TCP/IP* yerine Unix soket yapısı kullanacağınız şeklinde algılanır.
`hostaddr`[5]	Bağlanılacak olan sunucunun numerik adresini belirtir. Standart olarak IPv4 adres biçimi kullanabileceğiniz gibi, sisteminiz desteklediği sürece IPv6 kullanmanız da mümkündür. Öntanımlı değeri dışında herhangi bir değer atandığında, veritabanı sunucusuna *TCP/IP* ile bağlanılmaya çalışılacaktır.
`port`	Sunucuya bağlanılırken kullanılacak olan port numarası ya da unix soket dosyasının adresini belirtir.
`dbname`	Bağlanılacak olan veritabanı ismi. (Öntanımlı olarak, bağlantıyı kurmaya çalışan kullanıcının işletim sistemindeki kullanıcı adı alınır.)
`user`	Veritabanına bağlanırken kullanılacak olan kullanıcı adı. (Öntanımlı olarak, programı çalıştıran kullanıcının, işletim sistemindeki kullanıcı adı yer alır.)
`password`	Sunucu herhangi bir kimlik kontrolü talep ettiği taktirde kullanılacak olan şifre.
`connect_timeout`	Bağlantı kurulurken kontrol edilecek zaman aşımı sürecini (saniye cinsinden) belirtir. 0 yada negatif bir değer zaman aşımı sınırını etkisiz hale getirir. Değer olarak bir tamsayı girilmelidir.

4 Unix soket dizininin öntanımlı değerini PostgreSQL'in kaynak kodu içerisindeki `src/include/pg_config_manual.h` dosyasında yer alan `DEFAULT_PGSOCKET_DIR` değişkeni ile ayarlayabilirsiniz. Bu değişiklik aynı zamanda `postgres`, `postmaster` gibi bu dosyayı kullanarak derlenecek diğer programları da etkiyecektir. Fakat `postmaster`'a vereceğiniz `-k` parametresi ile bunu çalışma esnasında değiştirmeniz de mümkün.

5 Tek başına `hostaddr` yerine `host` kullanıldığı taktirde, `host` için adres çözümlemesi yapılacağından, zamanın önemli olduğu programlarda, bu çözümleme zaman kaybına neden olabilir. Ayrıca, Kerberos kullanılmadığı ve `hostaddr` seçeneğinin bulunduğu bir durumda, `host` kısmı göz ardı edilecektir. Kerberos kullanıldığı zaman ise, `host` ve `hostaddr`'in ikisinin de bulunduğu durumlarda, `hostaddr` için ters adres çözümlemesi gerçekleştirilip, `host` kısmında girilen değer ile örtüşüp örtüşmediği kontrol edilecektir. Değerlerin örtüşmediği durumda kimlik denetimi başarısız olarak sonuçlanacaktır.

`options`	Sunucuya gönderilmesi istenen komut satırı seçenekleri.
`sslmode`	Sunucu ile aranızdaki bağlantının *SSL* ile gerçekleştirilip gerçekleştirilmeyeceğine karar vermenizi sağlar. Parametre ile birlikte kullanabileceğiniz değerler şu şekildedir: `disable`: Şifrelenmemiş bir *SSL* bağlantısı kur. `allow`: İlk önce normal bağlantı, olmazsa *SSL* denenecek. `prefer`: İlk önce *SSL*, olmazsa normal bağlantı denenecek. `require`: Sadece *SSL* bağlantısı gerçekleştirilecek. PostgreSQL kurulumunuz *SSL* desteği ile yapılandırılmamışsa, `allow` ve `prefer` seçeneklerini kullanmanız herhangi bir sorun çıkarmadan *SSL* seçeneğini gözardı edecek olurken, `require` bir hata mesajı almanıza yol açacaktır.
`service`	`pg_service.conf` dosyasındaki servis tanımlamalarından birinin kullanılmasını sağlar. Bu sayede diğer seçenekler otomatik olarak belirtilen tanımda yer aldığı şekli ile bağlantıya katılır.

Bağlantı seçeneklerini, sistem çevre değişkenleri kullanarak da yapılandırmanız mümkün. Ayrıntılı bilgi için Çevre Değişkenleri bölümüne bakabilirsiniz.

```
PGconn *
PQsetdbLogin(const char *host,
             const char *port,
             const char *options,
             const char *tty,
             const char *dbname,
             const char *user,
             const char *password);
```

`PQsetdbLogin()` fonksiyonu, `PQconnectdb()` ile aynı işleve sahip olup, aralarındaki tek fark `PQsetdbLogin()` fonksiyonunun sabit parametreler içermesidir. Girilecek parametrelerden herhangi birinde öntanımlı değeri kullanmak istiyorsanız, fonksiyonu çağırırken ilgili parametreye `NULL` değerini ya da boş bir karakter katarını girmeniz yeterli.

```
PGconn *
PQsetdb(char *host,
        char *port,
        char *options,
        char *tty,
        char *dbname);
```

`PQsetdb()` fonksiyonu, içerdiği parametreler dışında kalan seçeneklerde öntanımlı değerleri kullanarak, user ve password değerlerini de `NULL` şeklinde atayıp, `PQsetdbLogin()` fonksiyonunu çağıran bir makrodur. Daha çok libpq ile yazılmış eski programların, yeni libpq sürümleri ile uyumluluğun sağlanması için halen kütüphanede yer aldığından kullanılmamasını tavsiye ederiz.

3.B. Asenkron Bağlantı Kurulumu

Buraya kadar incelenen senkron bağlantı kurulum fonksiyonlarından sonra, bu bölümde libpq tarafından sağlanan asenkron bağlantı kurulum fonksiyonları tanıtılmaya çalışılacaktır.

```
PGconn *
PQconnectStart(const char *baglanti_secenekleri);
```

`PQconnectStart()` fonksiyonu, parametre olarak aldığı seçenekler doğrultusunda

veritabanı ile asenkron bir bağlantı kurulumu girişiminin başlatılmasını sağlar. Fonksiyon ile birlikte kullanılan bağlantı seçenekleri, PQconnectdb() fonksiyonundakiler ile aynıdır. Asenkron bir bağlantı sayesinde, bağlantı kurulumu esnasında uygulamanızın bağlantıyı sağlayan yivi dışındaki komutlarınız, bağlantı sonucunu beklemek zorunda kalmaz. Bunun yardımıyla bağlantı kurulumunun denetimini program içinde istediğiniz yerde yapabilecek olurken, uygulamanız diğer işlemlerine devam edebilir.

Burada tüm bu asenkron bağlantı anlayışının bozulmasına sebebiyet verebilecek bir kaç istisnai durum mevcuttur:

- host ve hostaddr bağlantı seçeneklerinin doğru kullanılmaması, normal ve ters adres çözümlemelerinde beklemeye sebebiyet verebilir.
- PQtrace() komutunun yazacağı dosya üzerinde herhangi bir yazma sınırlaması, asenkron işleyişi bloke ederek, istenilen yiv mantığını bozabilir.

Fonksiyon çağrısının sonucunda NULL dönerse, ilgili PGconn yapısı için bellek alanı ayrılamadığı; dönen bağlantının PQstatus() ile kontrolünde CONNECTION_BAD dönerse de bağlantı kurulumunda sorun çıktığı anlamına gelir. Sonuç olarak CONNECTION_OK dönen bir PQconnectStart() cağrısı sonucunda, bağlantı isteminiz başarı ile gerçekleştirilmiş demektir. Fakat şuan veritabanına halen tam bir bağlantı sağlamış sayılmazsınız. Bütün bir veritabanı bağlantısını (ve bu bağlantının ne durumda olduğunu öğrenmek için) ise program akışı esnasında PQconnectPoll() komutu ile bağlantıyı yoklayarak elde edebilirsiniz.

> *PQconnectStart() ile asenkron bir bağlantı girişiminde bulunduğunuz taktirde, bağlantı özellikleri içindeki connect_timeout gözardı edilecektir. Bu yüzden programcı kendi kullanacağı teknikler ile (örneğin PQconnectPoll() ile döngüye girerek) geçen süreyi hesaplayarak programın gidişatı hakkında karar kılmalıdır.*

```
PostgresPollingStatusType
PQconnectPoll(PGconn *baglanti);
```

PQconnectPoll() fonksiyonu, PQconnectStart() ile başarılı bir bağlantı girişiminden sonra bağlantı durumunu yoklayarak bağlantıyı son haline ulaştırmak için kullanılır.

> *PQconnectPoll() fonksiyonunu çağırmadan önce PQsocket() komutu ile soketinizin bağlantı için uygun durumda olup olmadığından emin olmalısınız. Aksi taktirde bu hata yüzünden program bekleme sürecine girebilir, ki bu da asenkron bir bağlantıda beklenen bir nitelik değildir.*

PQconnectPoll() komutu sonucu dönecek olan PostgresPollingStatusType numaralandırılmış dizisi içindeki değerleri şöyle listeleyebiliriz:

```
typedef enum
{
    PGRES_POLLING_FAILED = 0, /* Yoklama işlemi hata ile sonuçlandı. */
    PGRES_POLLING_READING,    /* Yoklama sonucu okunuyor. */
    PGRES_POLLING_WRITING,    /* Yoklama için ilgili veri gönderiliyor. */
    PGRES_POLLING_OK,         /* Yoklama tamamlandı. */
    PGRES_POLLING_ACTIVE      /* Kullanılmıyor. (Eski programlar ile
                                 geri uyumluluk için saklanıyor.) */
} PostgresPollingStatusType;
```

PQconnectStart() ile başlatılan bir bağlantıyı, başarılı bir şekilde kurmak için PQconnectPoll() ile yoklamamız gerektiğinden bahsetmiştik. Bu yoklama işleminin nasıl ve kaç adımda yapılacağını örnekler kısmındaki asenkron_baglanti.c programında bulabilirsiniz.

4. Bağlantı Üzerinde İşlemler

Herhangi bir bağlantı kurma girişiminin sonucunda dönecek olan PGconn yapısı üzerinde bu bölümde incelenecek olan fonksiyonları kullanarak çeşitli işlemler yapabilirsiniz.

4.A. Bağlantı Hakkında Bilgi Edinme

Sunucuya yapılan (başarılı ya da başarısız) bir bağlantı girişimi hakkında bilgi edinmek için aşağıdaki fonksiyonları kullanabilirsiniz.

```
ConnStatusType
PQstatus(const PGconn *baglanti);
```

PQstatus() fonksiyonu, kendine parametre olarak girilen bağlantının o anki durumunu kendisini çağırana döndürür. ConnStatusType numaralanmış dizisi içinde, dönecek olası durumları ve özelliklerini şu şekilde listeleyebiliriz:

```
typedef enum
{
    /* Senkron/Asenkron bağlantı durum bilgileri. */
    CONNECTION_OK               /* Bağlantı sağlandı. */
    CONNECTION_BAD              /* Bağlantı sağlanırken hata oluştu. */

    /*
     * Bundan sonraki değerler sadece asenkron bir
     * bağlantı girişimi durumunda dönebilir.
     */
    CONNECTION_STARTED          /* Bağlantı kurulumu başlatıldı. */
    CONNCETION_MADE             /* Veritabanına bağlanıldı, diğer
                                   bağlantı adımlarına geçiliyor. */
    CONNECTION_AWAITING_RESPONSE /* Sunucudan cevap bekleniyor. */
    CONNECTION_AUTH_OK          /* Kimlik doğrulaması tamamlandı. */
    CONNECTION_SETENV           /* Çevre değişkenleri düzenleniyor. */
    CONNECTION_SSL_STARTUP      /* SSL ile kimlik denetimi başlatıldı. */
    CONNECTION_NEEDED           /* Kütüphane fonksiyonlarının kendi
                                   iç mekanizmaları esnasında kullandıkları
                                   bir değer. */
} ConnStatusType;
```

Programın normal işleyişi esnasında, bağlantıda bir sorun çıkmadığı ve bağlantı PQfinish() ile sonlandırılmadığı sürece PQstatus() fonksiyonu geçerli bağlantı için CONNECTION_OK değerini döndürecektir. Herhangi bir bağlantı kopması durumunda ise, PQstatus() fonksiyonu, CONNECTION_BAD değerini döndürür. Böyle bir durumda sorunun anlaşılmasında PQreset() ve PQerrorMessage() fonksiyon çiftinden yararlanılabilinir.

```
int
PQprotocolVersion(const PGconn *baglanti);
```

PQprotocolVersion() fonksiyonu, istemci ve sunucu arasındaki haberleşme için (parametre olarak aldığı kurulu bağlantıda) kullanılan protokolün sürüm numarasının *major* kısmını kendisini çağırana döndürecektir.

> *PostgreSQL 7.4 ve sonraki sürümlerde 3.0 protokolü kullanılırken, daha düşük sürümlerde 2.0 kullanılır.*

```
PQconninfoOption *
PQconndefaults(void);
```

PQconndefaults() fonksiyonu, veritabanına bağlanırken kullanılacak olan öntanımlı değerlerin bilgilerini PQconninfoOption yapısı içinde kendisini çağırana döndürür.

```
typedef struct {
    char   *keyword;  /* Kullanılan değişkenin adı. */
    char   *envvar;   /* Değişkenin sistem genelindeki adı. */
    char   *compiled; /* Değişkenin libpq'nun derlenme
                         esnasında atanmış değeri. */

    char   *val;      /* Seçeneğin şimdiki değeri. */
    char   *label;    /* Seçeneğin bağlantı dialoğunda
                         kullanılan ismi. */
    char   *dispchar; /* Bu değerin bağlantı dialoğunda
                         gösterilip gösterilmeyeceğinin ayarı.
                         Alabileceği değerler:
                         ""  Girilen değeri olduğu gibi göster.
                         "*" Şifre alanı. (Değeri gizle.)
                         "D" Hata ayıklama seçeneği. (Öntanımlı
                         olarak gösterme.) */
    int     dispsize; /* Bağlantı dialoğunda kullanılan değerin
                         karakter uzunluğu. */
} PQconninfoOption;
```

Bu yapıyı herhangi bir `PQconninfoOption` tipindeki işaretçiye atadıktan sonra, her seferinde işaretçiyi bir arttırmak suretiyle döngüye girilerek, işaretçi `NULL` değeri dönene kadar, değişkenlerin herbirine ulaşılabilir.

> *Kullanımının nasıl olduğu hakkında, örnekler bölümündeki `senkron_baglanti_ve_durum_bilgisi.c` dosyasına bakabilirsiniz.*

```
char *
PQhost(const PGconn *baglanti);
```

`PQhost()` fonksiyonu, parametre olarak aldığı bağlantıda kullanılan sunucu adresini kendisini çağırana döndürecektir. (Bağlantı seçeneklerindeki host seçeneğine denk.)

> *Eğer bağlantıda Unix soket yapısı kullanılmışsa, soket dosyasının adresi döndürülecektir.*
>
> *Karakter katarı döndüren fonksiyonların katar işaretçileri bellekten temizlenmemeli ve eğer üzerlerinde bir işlem yapılacaksa da mümkün olduğu kadar `strdup()` gibi standart kütüphane fonksiyonları kullanılarak çoğaltıldıktan sonra bu kopyaları üzerinde değişiklik yapılmalıdır. Sonuç ile işimiz bitip, ilgili yapıyı `PQclear()` ile zaten temizleyeceğimiz için, fonksiyondan dönen katarlar temizlenmemelidir.*

```
char *
PQuser(const PGconn *baglanti);
```

`PQuser()` fonksiyonu, parametre olarak aldığı bağlantıda kullanılan kullanıcı adını döndürecektir. (Bağlantı seçeneklerindeki `user` seçeneğine denk.)

```
char *
PQpass(const PGconn *baglanti);
```

`PQpass()` fonksiyonu, parametre olarak aldığı bağlantıda kullanılan kullanıcı şifresini döndürcektir. (Bağlantı seçeneklerindeki `password` seçeneğine denk.)

```
char *
PQdb(const PGconn *baglanti);
```

`PQdb()` fonksiyonu, parametre olarak aldığı bağlantıda kullanılan veritabanının adını döndürecektir. (Bağlantı seçeneklerindeki `dbname` seçeneğine denk.)

```
char *
PQport(const PGconn *baglanti);
```

`PQport()` fonksiyonu, parametre olarak aldığı bağlantıda kullanılan port numarasını

döndürecektir. (Bağlantı seçeneklerindeki `port` seçeneğine denk.)

```
char *
PQoptions(const PGconn *baglanti);
```

`PQoptions()` fonksiyonu, parametre olarak aldığı bağlantı kurulumunda sunucuya geçilen komut satırı seçeneklerini döndürecektir. (Bağlantı seçeneklerindeki `options` seçeneğine denk.)

```
SSL *
PQgetssl(const PGconn *baglanti);
```

`PQgetssl()` fonksiyonu, kendisini çağırana bağlantıda kullanılan *SSL* yapısını[6]; bağlantıda *SSL* kullanılmamışsa `NULL` değerini döndürecektir.

> *Fonksiyonun tam olarak çalışabilmesi için `USE_SSL` seçeneğinin (`#define USE_SSL` ile) tanımlanması gerekmektedir. Bu şekilde `ssl.h` dosyası da uygulamaya çalışma esnasında eklenecektir.*

```
char *
PQsocket(const PGconn *baglanti);
```

`PQsocket()` fonksiyonu, parametre olarak aldığı bağlantının kurulduğu sokette kullanılan dosya tanımlayıcısının numarasını kendisini çağırana döndürür. Başarılı bir bağlantıda `PQsocket()` fonksiyonu 0 ve 0'dan büyük bir değer; diğer türlü -1 döndürecektir.

```
int
PQclientEncoding(const PGconn *baglanti);
```

`PQclientEncoding()` fonksiyonu, istemci karakter kodlamasının PostgreSQL tarafınca tanımlı tamsayı değerini kendisini çağırana döndürecektir.

> *`PQclientEncoding()` fonksiyonu tarafından döndürülen değerler `include/postgresql/server/mb/pg_wchar.h`[7] başlık dosyasında tanımlı `pg_encoding_to_char()` fonksiyonu ile karakter katarı hallerine çevrilebilirler. Ayrıca `PQparameterStatus()` fonksiyonunu kullanarak da istemci karakter kodlamasını karakter katarı olarak öğrenebilirsiniz.*

```
int
PQisnonblocking(const PGconn *baglanti);
```

`PQisnonblocking()` fonksiyonu, parametre olarak aldığı bağlantının asenkron olması durumunda 1, senkron olması durumunda ise 0 değerini döndürecektir.

```
char *
PQerrorMessage(const PGconn *baglanti);
```

`PQerrorMessage()` fonksiyonu, kendisine parametre olarak geçilen bağlantı üzerinde gerçekleşmiş en son hatanın, hata mesajı katarını - sonuna bir yeni satır (`\n`) karakteri ekleyerek - kendisini çağırana döndürür. Aslına bakılacak olursa, neredeyse tüm libpq fonksiyonları, herhangi bir hata durumunda, ilgili bir hata mesajı oluştururlar. Fakat şu da unutulmamalıdır ki, bağlantı üzerinde işlem yapılmaya devam edildiği sürece hata mesajının aynı kalması beklenemez.

6 *SSL* yapısının tam bir dökümü için `ssl.h` dosyasındaki `ssl_st` yapısına bakabilirsiniz.

7 pg_wchar.h başlık dosyası, kendi iç tanımlamalarında bool tipini kullandığından dolayı, programınızın bu tipi önceden tanımlamış olması gerekmektedir.

4.B. Bağlantı Üzerinden Sunucu Hakkında Bilgi Edinme

Başarılı bir bağlantı sonucunda, bu bölümde tanıtılacak olan fonksiyonları kullanarak bağlanılan karşı sunucu hakkında bilgi edinebilirsiniz.

```
PGtransactionStatusType
PQtransactionStatus(const PGconn *baglanti);
```

`PQtransactionStatus()` fonksiyonu, sunucunun o anki *transaction* durumu hakkında bilgi verir. Döndürdüğü değerleri şu şekilde sıralayabiliriz:

```
typedef enum
{
    PQTRANS_IDLE     /* Bekleme konumunda. */
    PQTRANS_ACTIVE   /* Şu an bir komut çalıştırmakta. */
    PQTRANS_INTRANS  /* Hatasız devam eden bir transaction
                        bloğu içinde bekleme konumunda. */
    PQTRANS_INERROR  /* Hata oluşan bir transaction bloğu
                        içinde bekleme konumunda. */
    PQTRANS_UNKNOWN  /* Durum belirlenemediğinde dönen bir değer. */
} PGTransactionStatusType;
```

> *`PQtransactionStatus()` fonksiyonunun `autocommit` özelliği kapalı PostgreSQL 7.3 sunucularda hatalı değer döndürmesi olasıdır. (Sunucu taraflı `autocommit` özelliği PostgreSQL 7.3 sonrası sürümlerde bulunmamaktadır.)*

```
int
PQisBusy(PGconn *baglanti);
```

`PQisBusy()` fonksiyonu, sunucu üzerinde herhangi bir komut çalışıyorsa 1, aksi halde 0 değerini döndürecektir.

> *`PQisBusy()` fonksiyonu sunucu durumu için istemcinin bağlantısını gerçekleştiren sunucu yivini göz önüne alacaktır.*

```
const char *
PQparameterStatus(const PGconn *baglanti,
                  const char   *secenek_adi);
```

`PQparameterStatus()` fonksiyonu, çalışmakta olan sunucunun parametre olarak girilen komut satırı seçeneğinin o anki değerini; böyle bir seçenek bilinmiyorsa `NULL` değerini kendisini çağırana döndürecektir.

`PQparameterStatus()` fonksiyonu ile birlikte kullanabileceğiniz seçenek adları:

`server_version`	Sunucu sürümü.
`server_encoding`[8]	Sunucu karakter kodlaması.
`client_encoding`	İstemci karakter kodlaması.
`is_superuser`	İstemci `superuser` haklarına mı sahip?
`session_authorization`	Oturumun açıldığı kullanıcı adı.
`DateStyle`	Sunucu tarih biçimi.
`TimeZone`[8]	Sunucu yerel saati.
`integer_datetimes`[8]	PostgreSQL derlenme esnasında `--enable-integer-datetimes` parametresi alarak mı derlenmiş?

8 Bu özellikler 8.0 öncesi sürümler tarafından desteklenmemektedir.

```
int
PQserverVersion(const PGconn *baglanti);
```

`PQserverVersion()` fonksiyonu, parametre olarak geçilen bağlantının kurulu olduğu sunucunun sürüm numarasını belirten bir tamsayı değerini kendisini çağırana döndürecektir. Döndürdüğü değer, sunucunun sürüm numaralarının `major` ve `minor` kısımlarının iki haneli biçimleri, artı revizyon numarası şeklindedir. Şöyle ki:

```
# Sürüm 8.1 için dönecek olan değer
80100

# Sürüm 7.4.2 için dönecek olan değer
70402
```

Parametre olarak geçilen bağlantının kurulu olmadığı bir durumda ise fonksiyon 0 değeri döndürecektir.

```
int
PQbackendPID(const PGconn *baglanti);
```

`PQbackendPID()` fonksiyonu, sunucunun kurulu bağlantıyı sağlayan işlemin numarasını (`PID` değerini) kensini çağırana döndürür.

> *Dönen `PID` değeri, `LISTEN` ile dinlenen `NOTIFY` asenkron uyarıları ile birlikte dönen `PID` değerlerinin kıyaslanması için de kullanılabilir.*

4.C. Bağlantı Üzerinde İşlemler

Başarılı bir şekilde gerçekleştirilen sunucu bağlantısı sonucu dönen `PGconn` yapısı üzerinde bu bölümde tanıtılacak olan fonksiyonları kullanarak işlemler yapabilirsiniz.

```
int
PQsetClientEncoding(PGconn *baglanti, const char *kodlama)
```

`PQsetClientEncoding()` fonksiyonu, parametre olarak aldığı bağlantının istemci karakter kodlamasını belirtilen değere ayarlar. Başarılı olduğu durumda 0, aksi halde -1 değeri döndürecektir.

> *`PQsetClientEncoding()` fonksiyonu, aslında sizin yerinize sunucuya `SET client_encoding TO <KARAKTER-SETI>` SQL komutunu gönderip, sonucun başarılı olması durumunda yeni karakter kodlamasının değerini `PGconn` bağlantı yapısı içine yazdıktan sonra 0 değeri döndürür.*

```
PGVerbosity
PQsetErrorVerbosity(PGconn *baglanti, PGVerbosity duyarlilik);
```

`PQsetErrorVerbosity()` fonksiyonu, `PQerrorMessage()` ve `PQresultErrorMessage()` fonksiyonları sonucu dönecek olan hata mesajlarının duyarlılığını ayarlamanızı sağlar. Fonksiyona parametre olarak geçeceğiniz duyarlılık kısmında aşağıdaki değerleri kullanabilirsiniz:

```
typedef enum
{
    PQERRORS_TERSE,   /* Tek satırlık hata mesajları:
                         Hatanın Önemi + Asıl Mesaj + Hata Konumu */
    PQERRORS_DEFAULT, /* Öntanımlı biçim:
                         PQERRORS_TERSE + Ayrıntı + Çözüm Tavsiyesi + Bağlam */
    PQERRORS_VERBOSE  /* Mümkün olan tüm ayrıntılar. */
} PGVerbosity;
```

PQsetErrorVerbosity() fonksiyonu herhangi bir sorgu sonrası çalıştırıldığında, üretilmiş olan PGresult yapısının hata mesajı üzerinde bir etkiye sahip olmayacaktır.

Mesaj duyarlılığı başarılı bir şekilde ayarlandığında, fonksiyon eski duyarlılık derecesini kendisini çağırana döndürecektir.

```
int
PQsetnonblocking(PGconn *baglanti, int baglanti_tipi);
```

PQsetnonblocking() fonksiyonu, parametre olarak aldığı bağlantı tipini (senkron/asenkron) değiştirmek için kullanılır. Aldığı bağlantı tipi değişkeninde 1 (ya da herhangi bir pozitif değer) girilmesi durumunda, bağlantı asenkron, 0 girilmesi durumunda ise senkron hale getirilecektir. Daha açık olmak gerekirse, senkron duruma getirilen bir bağlantıda asenkron sorgu işletim fonksiyonları da sunucuya ilettikleri sorguların cevaplarını (senkron denkleri gibi) bekleyeceklerdir. Fonksiyon, başarılı olması durumunda 0, aksi halde -1 değerini döndürecektir.

```
int
PQconsumeInput(PGconn *baglanti);
```

PQconsumeInput() fonksiyonu, soket üzerinde biriken sunucu tarafında işlenmiş tüm verinin alınıp, istemci tarafındaki bellek üzerinde saklanmasını sağlar. Başarılı olduğu durumda 1, aksi halde 0 değerini döndürecektir.

Başarısız olduğu durumda herhangi bir bağlantı probleminden şüphelenilmesi gerektiği için PQerrorMessage() fonksiyonu ile sorun hakkında bilgi sahibi olunulabilir.

PQconsumeInput() fonksiyonu, bağlantının asenkron olduğu bir durumda ilk önce bellekte gönderilmek üzere bekleyen veri varsa bunları PQflush() ile temizler. (Aksi halde hala cevabı beklenen veri, gönderim kuyruğunda ise PQconsumeInput() fonksiyonunu çağırmanın bir anlamı kalmaz.) Ardından soket üzerinde birikmiş veri varsa bunu bellek üzerine yazıp, soket üzerinde bekleyen verinin boşaltılmasını sağlar. Bu sayede istemci herhangi bir select/read döngüsünde ise - yani sırası ile select() ardından ona göre read() fonksiyonunu çağırılıyorsa - anlamsız yere döngü tekrarından kaçınılmış olur.

```
int
PQflush(PGconn *baglanti);
```

PQflush() fonksiyonu, veritabanına gönderilmek üzere sırada bekleyen tüm verinin yollanmasını sağlar. Başardığı ya da gönderilmek üzere bekleyen komut olmadığı durumda 0, bir nedenden dolayı başarısız olursa -1, verinin hepsini gönderemediği durumunda ise 1 değerini döndürecektir.

1 değeri yalnızca asenkron bir bağlantı tipinde döndürülür. Böyle bir durumda bağlantı soketinin tekrar yazılabilir hale gelmesinden sonra, PQflush() fonksiyonu bir kez daha çağrılır. Bu şekilde, PQflush() 0 değeri döndürene kadar fonksiyon ardarda çağrılarak kuyruk boşaltılır. (Soketin yazılabilir durumda olup olmadığını basit bir select() sorgusu sonucu anlayabilirsiniz.

Örnekler bölümündeki select() sorgusu kullanan asenkron_veri_alimi.c dosyasına bakarak da konu hakkında fikir sahibi olmanız mümkün.

PQflush() fonksiyonu, aslında her asenkron veri gönderiminden sonra verinin karşı tarafa yollanıp yollanmadığından emin olmak için çağrılmalıdır. Fakat sunucundan dönen sonuçların alınmasında kullanılan PQgetResult() fonksiyonu, bunu bizim için fazlasıyla yerine getirdiğinden, elle PQflush() çağrısı çoğu zaman yapılmamaktadır.

Ek olarak, libpq kütüphanesi 8K'dan daha küçük veri içeren sorgulamaları sunucuya

göndermemektedir. Bunun yerine, sorgu alınmak istendiğinde `PQflush()`'ın `PQgetResult()` tarafından çağrıldığından daha önce de bahsetmiştik. Fakat, verinin karşa tarafa hemen sorgu çağrısının ardından yollanmasının önemli olduğu durumlarda, 8K'dan küçük sorguları çalıştırdıktan sonra elle `PQflush()` çağrısında bulunulabilir.

```
void
PQfinish(PGconn *baglanti);
```

`PQfinish()` fonksiyonu, veritabanı ile kurulan bağlantıyı koparıp, bağlantı değişkeni için bellekte tutulan yeri serbest bırakmakta kullanılır. Bu nedenden dolayı, bir bağlantı başarısız olsa dahi, bağlantı değişkeni tarafından tutulan bellek alanını serbest bırakmak için `PQfinish()` fonksiyonu çağrılmalıdır.

> *Herhangi bir `PGconn` tipi bağlantı değişkeni, bir kez `PQfinish()` tarafından serbest bırakıldıktan sonra kullanılamaz hale gelir.*

```
void
PQreset(PGconn *baglanti);
```

`PQreset()` fonksiyonu, kurulu veritabanı bağlantısını koparıp, kullanılan eski bağlantı seçenekleri ile veritabanına tekrar bağlanmak için kullanılır.

> *`PQreset()` fonksiyonu, daha çok herhangi bir bağlantı kopması durumunda, hatayı tespit etmek için kullanılır. Bu yolla bağlantı tekrar kurulmaya çalışılarak, tekrar başarısız olunması durumunda hata mesajı elde edilir.*

```
int
PQresetStart(PGconn *baglanti);
```

`PQresetStart()` fonksiyonu, varolan asenkron bir bağlantının koparılıp yeniden sağlanması (sıfırlanması) için kullanılır. Asenkron bir bağlantının tam olarak sıfırlanması için `PQresetStart()` fonksiyonunun çalıştırılması sonucu 1 değeri dönmesinin ardından `PQresetPoll()` fonksiyonunun (asenkron bir bağlantı kurulurken yapılana benzer şekilde) çağrılması gerekir. `PQresetStart()` fonksiyonunun başarısızlığa uğraması durumunda fonksiyon 0 değeri döndürecektir.

```
PostgresPollingStatusType
PQresetPoll(PGconn *baglanti);
```

`PQresetPoll()` fonksiyonu, asenkron bir bağlantıda, `PQresetStart()` ile başlatılmış bir sıfırlama işleminin tamamlanması için kullanılır. Kullanımı `PQconnectPoll()` fonksiyonuna benzer şekilde olup, döngüye girilerek safha safha çağrılır.

> *`PQresetPoll()` fonksiyonu, `PQconnectPoll()` fonksiyonuna bir arayüzden ibarettir. Fakat sıfırlama (reset) işlemi için gerekli fonksiyonların daha derli toplu halde bulunması açısından `reset` adı altında tekrar yer almaktadır.*

5. Sorgu İşletimi

Buraya kadar veritabanına nasıl bağlanılacağı üzerinde durmaya çalıştıktan sonra, bu bölümde kurulan başarılı bir bağlantı yoluyla, veritabanı üzerinde sorgulamalarımızı nasıl gerçekleştirebileceğimizi anlatmaya çalışacağız.

Bağlantı kurulumunda olduğu gibi, sorgu işletimi de senkron ve asenkron olmak üzere ikiye ayrılmaktadır. Nasıl ki senkron bağlantılar, aslında asenkron bağlantı fonksiyonlarını kullanan başka bağlantı fonksiyonlarından ibaret; aslında senkron sorgu işletim

fonksiyonları da asenkron sorgu fonksiyonlarını çağıran birer ara katmandan ibarettir.

Bizim için asenkron sorgu komutlarını doğru sıra ile işleten bir senkron sorgu işletim fonksiyonu kümemiz varken, peki neden biz tüm bu ara işlemleri teker teker elle yapmak isteyelim? İşte burada asenkron işleyiş ile kastımızın tam üstünde duruyoruz. Herhangi bir senkron sorgu fonksiyonunu çağırdığınız zaman, sorgu işletimi - sunucu ve istemci arayüzü olmak üzere - her iki taraf tarafından sonuçlanana kadar uygulamanız, programın akışı esnasında, o noktadaki fonksiyondan gelecek sonucu bekler vaziyette kalır. Fakat asenkron bir sorgu işletiminde, siz sorgunuzu bir kere karşı tarafa ilettikten sonra, kendi işlerinize devam edebilir, sorgu sonucunu kullanacağınız anda sonucun ulaşıp ulaşmadığını kontrol ettikten sonra onu da alıp yolunuza devam edebilirsiniz.

5.A. Senkron Sorgu İşletimi

Bu bölümde, kurulu bir bağlantı üzerinden veritabanına nasıl senkron bir şekilde sorgu gönderip, sonucunu alabileceğimiz üzerinde duracağız.

> *`PQexec()`, `PQexecParams()` ve `PQprepare()` senkron sorgu işletim fonksiyonları herhangi bir sorgu gönderiminde bulunmadan önce, kendi içlerinde `PQexecStart()` fonksiyonunu çağırırlar. `PQexecStart()` fonksiyonunu incelediğimizde ise, fonksiyonun sunucu üzerinde daha önceden kalan sonuç verisini alıp direk temizlediğini görürüz. Kısaca, ard arda çalıştırılan senkron sorgu gönderimleri, kendilerinden bir önceki sonucun silinmesine sebebiyet vereceklerdir. Bu nedenle, senkron sorgu işletim fonksiyonlarını kullanırken, evvelsinde sunucu üzerinde beklediğiniz bir verinin bulunup bulunmadığına dikkat ediniz.*

```
PGresult *
PQexec(PGconn *baglanti, const char *komut);
```

`PQexec()` fonksiyonu, parametre olarak gönderilen bağlantı üzerinde, belirtilen *SQL* komutunu çalıştırıp, sorgulamanın cevabını bekledikten sonra, sonucu `PGresult` tipi değişken içinde kendisini çağırana döndürecektir.

Fonksiyon `NULL` değeri döndürdüğü zaman, gerekli değişkenler için bellek ayırılamadığını ya da veritabanı üzerinde sorgulamayı gerçekleştiremediğini belirtir. Böyle bir durumda `NULL` değeri, `PGRES_FATAL_RESULT` durumu ile bir kabul edilmelidir. Fakat, `NULL` döndürmemiş bir `PQexec()` çağrısının da başarı ile sorgulamayı gerçekleştirdiği anlamı çıkarılmamalıdır. Dönen tüm `PGresult` tipindeki sonuç yapıları üzerinde `PQstatus()` fonksiyonu kullanılarak sonucun son durumu hakkında kesin bir bilgi edinilmelidir.

> *`PQexec()` fonksiyonuna göndereceğiniz komutları birbirinden noktalı virgül ile ayırarak, aynı anda birden fazla komut gönderiminde bulunabilirsiniz. `PQexec()` bunların hepsini bir transaction bloğu içinde işleyecektir. Fakat unutulmamalıdır ki `PQexec()` en son sorgunun sonucunu döndürecektir. Bu nedenle, birden fazla komut gönderileceği durumda, sorguları sunucuya ayrı ayrı göndermeniz, istediğiniz sonucu elde etmeniz açısından daha sağlıklı olacaktır.*

```
PGresult *
PQexecParams(PGconn            *baglanti,
             const char        *komut,
             int                parametre_sayisi,
             const Oid         *parametre_tipleri,
             const char *const *parametre_degerleri,
             const int         *parametre_uzunluklari,
             const int         *parametre_bicimleri,
             int                sonuc_bicimi);
```

`PQexecParams()` fonksiyonu, parametre kullanılarak veritabanı üzerinde sorgulama gerçekleştirilmesine olanak sağlar. Diğer senkron sorgu işletim fonksiyonlarına ek olarak, `PQexecParams()` fonksiyonu ile aynı zamanda gönderilen ve dönecek olan verinin biçimini

de (ikili ya da metin şeklinde) belirleyebilirsiniz.
Fonksiyonun aldığı parametreleri açıklamaları ile birlikte verecek olursak:

`baglanti`	Sorgulamanın üzerinde işletileceği veritabanı bağlantısının işaretçisi.
`komut`	İçinde `$1`, `$2`, `$3`... şeklinde parametre içerebilen *SQL* komutu. Komutun içinde kullanılan parametreleri tırnak içine almamalısınız. Aksi halde şuna benzer bir hata ile karşılaşırsınız: `ERROR: 08P01: bind message supplies 2 parameters, but prepared statement "" requires 0`
`parametre_sayisi`	*SQL* komutunda yer alan parametre sayısı.
`parametre_tipleri`	Bir sonraki parametrede dizi (`array`) içinde gireceğimiz değerlerin tiplerini içeren dizidir. Bu dizide kullanacağımız değerin tipini `OID` değeri[9] ile belirleyebileceğimiz gibi, 0 ya da `NULL` girerek, bunlara sunucu tarafından karar verilmesini sağlayabiliriz.
`parametre_degerleri`	*SQL* komutunda kullanılan parametrelerin sırasıyla değlerlerinin bulunduğu dizi.
`parametre_uzunluklari`	`parametre_degerleri` kısmında kullanılan ikili sistemdeki (`binary`) değerlerin uzunluklarını bu dizi ile belirtebilirsiniz. `parametre_degerleri` dizisinde kullanılan, ikili dışında kalan katar ya da `NULL` değerlerinde `parametre_uzunluklari` dizisinin ilgili değeri önemsenmeyecek olup, uzunluklar (`strlen()` fonksiyonu kullanılarak) fonksiyon tarafından hesaplanacaktır.
`parametre_bicimleri`	`parametre_degerleri` dizisinde karşılık gelen değerlerin biçimlerini atamak için kullanılır. 0 değeri karakter katarı anlamına gelirken, 1 karşılık gelen verinin ikili biçimde olduğu anlamına gelecektir. Eğer `parametre_bicimleri` dizisi `NULL` ise, tüm değerler karakter katarı olarak alınacaktır.
`sonuc_bicimi`	0 dönecek olan sonucun karakter katarı, 1 ise ikili biçimde olduğunu ifade eder.

A.1. Parametre Kullanan Fonksiyonlar Hakkında

Parametre kullanımına izin veren senkron ve asenkron tüm fonksiyonlar, her seferinde tek bir komut gönderimine müsade etmektedirler. (Aksi halde `ERROR: cannot insert multiple commands into a prepared statement` şeklinde bir hata mesajı ile `PGRES_FATAL_ERROR` durumunda bir sonuç elde edersiniz.) Bu kullanılan protokolün getirdiği bir kısıtlama olsa da, bir yerde *SQL Injection* saldırılarına karşı da bir savunma biçimidir. Ayrıca bu fonksiyonlar 3.0 öncesi protokoller tarafından da desteklenmemektedirler.

Ek olarak, veritabanı sorgulamalarınızda parametre kullanarak gönderdiğiniz verinin tipini `parametre_tipleri` dizisi içinde belirtmeyip, bunun sunucu tarafından tahmin edilmesini istediğiniz durumda ufak bir istisna bulunmaktadır. PostgreSQL parametre kullanılarak gönderilen verinin biçimini ilgili tablo sütunun biçimi ile bir tutacaktır. Örneğin

```
SELECT ... WHERE x = $1 ...
```

sorgusunu ele alalım. Burada `$1` verisinin biçimi açık olarak belirtilmediği taktirde, sunucu bunu `x` değerinin biçimi ile bir tutacaktır. Bu gibi durumlardan kaçınmak için gönderilecek verinin tipinin açık olarak belirtilmesinde yarar vardır. Yani – `$1` değişkenin tipinin `bigint` olduğunu varsayacak olursak – yukarıdaki sorgunun

```
SELECT ... WHERE x = $1::bigint ...
```

şeklinde gönderilmesi daha sağlıklı sonuç verecektir. Özellikle `sonuc_bicimi`'nin ikili olduğu durumda, ikili verinin tipinin tahmin edilmesi – karakter katarı biçimindeki sorgulara oranla – daha zor bir hal alacaktır.

9 Herhangi bir tipin `OID` değerine `pg_type` tablosu üzerinde yapacağınız `SELECT oid FROM pg_type WHERE typname = 'tip_ismi'` sorgusu ile ulaşabileceğiniz gibi, `postgresql/server/catalog/pg_type.h` başlık dosyasındaki tanımlamalardan da yararlanabilirsiniz.

```
PGresult *
PQprepare(PGconn     *baglanti,
          const char *ifade_ismi,
          const char *komut,
          int         parametre_sayisi,
          const Oid  *parametre_tipleri);
```

`PQprepare()` fonksiyonu, sunucuda ileride kullanılmak üzere (daha sonra `PQexecPrepared()` fonksiyonu ile çağırabileceğiniz) sorgular hazırlamanıza olanak sağlar. `PQprepare()` fonksiyonu, sık kullanılacak olan komutların her seferinde çalıştırılmasındansa, kullanılacakları zaman, hazır olarak çağrılabilmeleri nedeniyle tercih edilir.

> *`PQexecPrepared()` ile çalıştırılmak üzere hazır bekleyen sorgulamaları `PREPARE` komutu ile de oluşturabilirsiniz. Fakat `PQprepare()` fonksiyonu, tüm parametre tiplerini kullanıcı tarafından girilmek zorunda bırakmadığından daha esnek bir yapı sunmaktadır. (Ayrıntılı bilgi için `PREPARE` komutunun nasıl kullanıldığına bakabilirsiniz.)*

`PQprepare()` (yada `PREPARE`) ile sunucu tarafında bir komut hazırlandığında, ileride gerçekleşecek olan sorgudaki komut tümcesinin ayrıştırılması, baştan yazılması ve sorgunun planlanması işlemleri otomatik olarak yapılacaktır. Bu nedenle, sorgu daha sonradan çağrılacağı zaman, önceden yapılan tüm bu işlemlerden dolayı zaman kazancı sağlanacak olması şüphesiz ki muhtemeldir. Fakat eğer bir komut sadece tek bir kez çalıştırılacaksa da, bunun için `PREPARE` ifadesi kullanmanın bir anlamı yoktur. Onun yerine sorgunun direk sunucuya gönderilmesi daha az zahmetli olacaktır.

`PQprepare()` kullanımı, `PQexecParams()` fonksiyonuna çok benzerdir. Burada tek fark, fonksiyonu direk çalıştırmak yerine ilk önce `PQprepare()` ile sorgulama işlemi sunucu tarafında sorgulanmak üzere hazır hale getirilir, ardından çalıştırılmak istendiği zaman `PQexecPrepared()` ile ilgili sorgu başlatılır. Fonksiyona gönderilen komut, `parametre_sayisi`, `parametre_tipleri` parametrelerinin kullanımı `PQexecParams()`'daki ile aynıdır. (Dolayısıyla girilecek *SQL* komutu içerisinde `$1`, `$2`, `$3`... şeklinde parametre kullanılabilir.)

> *`PQprepare()` fonksiyonu da, `PQexecParams()` fonksiyonuna benzer şekilde, komut parametresi olarak birden fazla sorguya izin vermemektedir.*

`PQprepare()` ile hazırladığımız sorgulamaları birbirine karıştırmamak için herbirine kendine ait bir ifade ismi (`ifade_ismi`) verilir ve çağrılırken de buna göre çağrılır. Aynı ifade ismine sahip iki sorgu hazırlamaya çalışmak hata döndürecektir. Ayrıca, `ifade_ismi` kısmına boş bir karakter katarı girildiği taktirde, isimsiz bir sorgu hazırlanacaktır. Eğer hali hazırda zaten böyle isimsiz bir sorgu mevcutsa da, artık onun yerine bu yeni girilen geçerli olacaktır.

> *libpq kütüphanesinde hazırlanmış bir sorguyu silmek için herhangi bir komut bulunmamasına rağmen, `DEALLOCATE` komutunu bu iş için kullanabilirsiniz. (Ayrıntılı bilgi için `DEALLOCATE` komutunun kullanımına bakabilirsiniz.) Bunun dışında, her oturum kapanışında, hazırlanan sorgular sunucu tarafından otomatik olarak serbest bırakılacaktır zaten.*

```
PGresult *
PQexecPrepared(PGconn             *baglanti,
               const char         *ifade_ismi,
               int                 parametre_sayisi,
               const char *const  *parametre_degerleri,
               const int          *parametre_uzunlugu,
               const int          *parametre_bicimi,
               int                 sonuc_bicimi);
```

`PQexecPrepared()` fonksiyonu, bir önce kısımda tanıtılan `PQprepare()` fonksiyonu ile sunucu tarafında hazır olarak bekletilmekte olan `ifade_ismi` adlı sorguyu belirtilen parametreler doğrultusunda gerçekleştirmenizi sağlar. `PQexecPrepared()` fonksiyonunun `PQexecParams()` fonksiyonu ile ortak olarak paylaştığı parametrelerin (`parametre_sayisi`, `parametre_degerleri`, `parametre_uzunluklari`, `parametre_bicimleri` ve `sonuc_bicimi`) kullanımı aynıdır. İsimsiz bir hazırlanmış sorgunun çalıştırılmasının istendiği durumda ise, `ifade_ismi` kısmına boş bir karakter katarı girilmesi yeterli olacaktır.

5.B. Asenkron Sorgu İşletimi

Bu bölümde, kurulu bir bağlantı üzerinden libpq kütüphanesi tarafından sağlanan fonksiyonları kullanarak nasıl asenkron bir şekilde sorgu işletebileceğimiz üzerinde durmaya çalışacağız.

```
int
PQsendQuery(PGconn *baglanti, const char *komut);
```

`PQsendQuery()` fonksiyonu, sunucuya, sonucunu beklemeksizin, bir SQL sorgusu iletmenizi sağlar. Fonksiyon, sorgunun başarı ile karşı tarafa ulaştığı durumda 1, aksi halde 0 değeri döndürecektir.

Asenkron sorgu işletim fonksiyonlarının, program işleyişini bloke etmemelerinin dışında senkron denklerine oranla en büyük avantajları, aynı anda birden fazla sorgu gönderiminde bulunulduğu zaman, bunların sonucunun teker teker alınabilecek olmasıdır. Bunun için birden fazla sorgu tek seferde gönderildikten sonra, `PQgetResult()` fonksiyonu ile döngüye girilerek, her bir sorgunun cevabı ayrı ayrı alınabilir.

Asenkron sorgu işletiminde, programcı sunucuya birden fazla komut gönderiminde bulunacağı zaman, sunucunun o an başka bir iş üzerinde uğraşıyor olmamasına[10] ve sunucu tarafında önceden döndürülen tüm verinin alınmasına dikkat etmelidir. Aksi halde, sunucu o an meşgulken, başka bir komut gönderimi durumunda `another command is already in progress` mesajına benzer bir hata ile karşılaşırsınız. Bu nedenle aynı anda birden fazla sorgunun gönderileceği durumlarda, genelde sorgu komutları `';'` (noktalı virgül) ile birleştirilerek yollanılır; yahut sorgu gönderilip, onun sonucu alındıktan sonra bir sonraki sorguya geçilirerek bu şekilde devam edilir.

10 Sunucunun o an meşgul olup olmadığını `PQisBusy()` fonksiyonu ile öğrenebilirsiniz.

```
int
PQsendQueryParams(PGconn             *baglanti,
                  const char         *komut,
                  int                 parametre_sayisi,
                  const Oid          *parametre_tipleri,
                  const char *const  *parametre_degerleri,
                  const int          *parametre_uzunluklari,
                  const int          *parametre_bicimleri,
                  int                 sonuc_bicimi);
```

`PQsendQueryParams()` fonksiyonu, senkron sorgu işletiminde tanıttığımız `PQexecParams()` fonksiyonu ile aynı işleve sahip olup, asenkron olarak çalışmaktadır. Bu yüzden `PQsendQueryParams()` fonksiyonunun aldığı parametrelerin kullanımı `PQexecParams()` fonksiyonununkiler ile aynıdır. Fonksiyon, ilgili sorguyu karşı tarafa ilettikten sonra dönecek sorgu sonucu için beklemeden kendisini çağırana geri döner. `PQsendQuery()` gibi, döndürdüğü sonuç değeri 1 başarılı, 0 ise hatalı bir iletim gerçekleştiği anlamına gelir.

```
int
PQsendPrepare(PGconn     *baglanti,
              const char *ifade_ismi,
              const char *komut,
              int         parametre_sayisi,
              const Oid  *parametre_tipleri);
```

`PQsendPrepare()` fonksiyonu, senkron sorgu işletiminde kullanılan `PQprepare()` fonksiyonu ile aynı işleve sahip olup, işleyiş tarzı olarak asenkron çalışır - yani sorgu hazırlığı karşı tarafa iletildikten sonra, cevabı için beklemenize gerek yoktur. Başarılı olduğu durumda 1, hata oluşumunda ise 0 değeri döndürecektir.

```
int
PQsendQueryPrepared(PGconn             *baglanti,
                    const char         *ifade_ismi,
                    int                 parametre_sayisi,
                    const char *const  *parametre_degerleri,
                    const int          *parametre_uzunluklari,
                    const int          *parametre_bicimleri,
                    int                 sonuc_bicimi);
```

`PQsendQueryPrepared()` fonksiyonu, `PQexecPrepared()` fonksiyonunun asenkron olarak çalışan halidir. Ayrıca, aldığı parametrelerin kullanımı da `PQexecPrepared()` fonksiyonundaki karşılıkları ile aynıdır.

```
PGresult *
PQgetResult(PGconn *baglanti);
```

`PQgetResult()` fonksiyonu, asenkron sorgu işletiminden dolayı (`PQsendQuery()`, `PQsendQueryParams()`, `PQsendPrepare()` ya da `PQsendQueryPrepared()` fonksiyonları ile) sunucuda oluşan sonucun alınmasını sağlar. Fonksiyon `NULL` değeri döndürdüğü zaman, sunucuda alınacak sonuç kalmadığı anlamına gelir. Bu nedenle açıktır ki, `PQgetResult()` genelde sunucuda sonuç kalmayana kadar döngü içinde tekrar tekrar çalıştırılarak kullanılır.

`PQgetResult()` fonksiyonunun döngüsel olarak kullanımı şuna benzer bir yapı izler:

```
...
while ((res = PQgetResult(baglanti)) != NULL)
    ...
```

Bir tek sorgu işletimi sonucunda dönen verinin alımı döngüsel olarak değil de, elle yapmak istendiğinde, `PQgetResult()` fonksiyonunun iki kez çağrılması gerekmektedir. Şöyle ki:

```
/*
 * Sunucuya tek bir asenkron sorgu gönderiminde bulunulur.
 * (Yani ; ile ayrılmış birden fazla sorgu bulunmamalıdır.)
 */
...

/* Sonuç yapısı alınır. */
sonuc = PQgetResult(baglanti);

/* Dönen sonuç kullanılıp iş bittiğinde temizlenir. */
...
PQclear(sonuc);

/*
 * İşimiz bittiğinde dönen tüm veriyi kapatmak için
 * PQgetResult() fonksiyonunu son bir kez çağırıyoruz.
 */
PQgetResult(baglanti); /* NULL döndürecek. */
```

Hatırlarsak senkron sorgu işletim fonksiyonları ile aynı anda birden fazla *SQL* komutu gönderilip, sonucunu almaya çalıştığımızda bize yapılan en son sorgunun sonucu döndürülüyordu. Dikkat edersek `PQsendQuery()` ve `PQgetResult()` fonksiyonlarının kullanımının bizi bu sınırlamadan kurtardığını görebiliriz: `PQsendQuery()` ile birden fazla *SQL* komutu yolladıktan sonra, bunların sonuçlarını teker teker `PQgetResult()` ile alabiliriz.

> *Herbir `PQgetResult()` sonucu ile işimiz bittiğinde, `PQclear()` ile sonuç yapısını temizlemeliyiz.*

`PQgetResult()` ile sunucudan sonuç almaya çalıştığımızda, fonksiyonun beklemesine sebebiyet verecek iki durum vardır. Bunlardan birincisi, sunucunun o an (sonucunu almaya çalıştığımız sorguyu işliyor olması ya da bir önceki sorgudan dönen sonucun alınmaması gibi durumlarda) meşgul olmasıdır, ki bunu `PQisBusy()` fonksiyonu ile anlayabiliriz. İkincisi ise, soket üzerinde okunan veri bir `PGresult` yapısı oluşturabilecek kadar yeterli değildir. İkinci durumda, ilk önce bağlantı soketi üzerinde kuralacak basit bir `select()` döngüsü ile veri gelip gelmediğinden haberdar olduktan sonra, gelen veri `PQconsumeInput()` ile okutulur. Ardından `PQisBusy()` ile halen gelecek veri olup olmadığına karar verilip, gelecek veri kalmamışsa `PQgetResult()` fonksiyonu çağrılır. (Cümleler ile ifade edilmeye çalışıldığında ne kadar karmaşık gibi gözüküyor olsa da, örnekler kısmında yer alan ilgili örneğe göz attıktan sonra işleyişin nasıl olduğuna dair kafanızda daha da net bir görüntü oluşacağını düşünüyoruz.)

6. Sorgu İptali

Bazı durumlarda, uygulama işlenmekte olan belirli bir veritabanı sorgusunun iptalini isteyebilir. Bu bölümde, böyle bir durumda başvurabileceğimiz kütüphane fonksiyonları üzerinde duracağız.

Konuya girmeden önce, sorgu iptalinin ne gibi durumlarda kullanılabileceği üzerinde durmanın yararlı olabileceğini düşündük. Sorgu iptalinin, o an çalışmakta olan sorgu için gerçekleştirileceği düşünüldüğünde, gerçekleştirilen sorgunun asenkron olma gerekliliği aşikardır. Aksi halde, senkron sorgu işletim fonksiyonları ile çalıştırılan bir sorgunun iptali için iptal fonksiyonlarını ayrı bir yiv altında çalıştırmak zorunda kalırız; ki bu oldukça karmaşık bir yapıyı gerektirir. Bunun yerine sorgu asenkron olarak sunucuya bir kez gönderildikten sonra, sonucu beklemeye gerek kalmadan program kendi içinde akışına devam ederken istendiği yerde iptal işlemi gerçekleştirilebilir.

Özetlemek gerekirse, işletimi uzun süren sorgular asenkron olarak karşıya bir kez iletildikten sonra, sonucun beklenmesi esnasında kullanıcıya bir iptal seçeneği sunulabilir. Bu gibi durumlarda da aşağıda bahsi geçen işletilen sorgunun iptal edilmesinde kullanılacak kütüphane fonksiyonlarından yararlanılabilir.

```
PGcancel *
PQgetCancel(PGconn *baglanti);
```

`PQgetCancel()` fonksiyonu, parametre olarak girilen bağlantı üzerinde yapılacak herhangi bir sorgu iptali için gerek duyalacak `PGcancel` yapısını kendisini çağırana döndürer. Fonksiyona parametre olarak verilen bağlantının kurulu olmadığı, ya da `PGcancel` nesnesi için bellekte yeterli alan ayrılamadığı zaman, fonksiyon `NULL` değerini döndürecektir.

```
void
PQfreeCancel(PGcancel *iptal_yapisi);
```

`PQfreeCancel()` fonksiyonu, herhangi bir sorgunun iptalinde kullanılmak üzere `PQgetCancel()` tarafından oluşturulmuş olan `PGcancel` iptal yapısı için ayrılan alanı bellekten bırakmak için kullanılır.

```
int
PQcancel(PGcancel *iptal_yapisi,
         char     *hata_katari,
         int       hata_katarinin_buyuklugu);
```

`PQcancel()` fonksiyonu, o an sunucuda gerçekleşmekte olan bir sorgunun iptalini ister. Fonksiyon, başarılı olması durumunda 1, hata durumunda ise - hata katarına sorunu açıklayıcı bir hata mesajı (belirtilen hata katarı büyüklüğünü aşmayacak şekilde) yerleştirerek - 0 değerini döndürecektir. (Önerilen hata katarı büyüklüğü 256 bayttır.)

Eğer iptal edilmesini istediğimiz sorgu tamamlandıktan sonra, `PQcancel()` etki gösterirse, iptal işleminin sorgu üzerinde hiçbir etkisi olmayacaktır; etki sorgu esnasında olduğunda ise iptal işleminin başarı ile gerçekleştiği durumuda iptalini istediğimiz sorgu sonucu hata döndürecektir.

7. Uygulama İçinde COPY Kullanımı

`COPY` komutu içeren sorgulamaların gerçekleştirilmesinde libpq kütüphanesi programcı tarafından kullanılmak üzere bir kaç fonksiyon sağlamaktadır. Bu bölümde bu fonksiyonlar ve kullanımları üzerinde durmaya çalışacağız.

> *Konu anlatımı kısmında, kullanıcının `COPY` komutu hakkında yüzeysel bir bilgiye sahip olduğu farz edilip ilerlenecektir. (`COPY` komutu hakkında ayrıntılı bilgi için kitabın ilgili bölümüne göz atabilirsiniz.)*

`COPY` komutu kullanımında, sorgu işletim fonksiyonları ile `COPY` komutu içeren bir *SQL* sorgu gönderiminin sonucunda dönecek olan `PGresult` nesnesinin durumuna bakılarak

bundan sonraki adımda ne yapılacağına karar verilir. Başarılı bir COPY komutu gönderiminde dönecek olan PGresult yapısı PGRES_COPY_IN ya da PGRES_COPY_OUT durumunda olacaktır. Gerekli COPY durmuna sahip bir sorgu sonucu aldıktan sonra tanıtılacak olan fonksiyonları kullanarak, geçerli bağlantı üzerinde veri aktarımı ya da alımında bulunulabilir.

Sonuç yapısı belirtilen COPY durumlarından birinde olmadığı halde aşağıdaki COPY fonksiyonlarının kullanımı halinde, `no COPY in progress` şeklinde bir hata mesajı ile karşılaşırsınız.

COPY işlemi içindeki veri alımı ya da aktarımı bittiği zaman, PQgetResult() çağrılarak COPY işleminin son durumu öğrenilir. Dönen durum PGRES_COMMAND_OK ise işlemimiz sorunsuz gerçekleşmiş demektir; aksi halde PGRES_FATAL_ERROR durumu döndürülecektir.

COPY komutu kullanan genel bir program işleyiş iskeleti şuna benzer olacaktır:

```
/* COPY isteğimiz karşı tarafa iletilir. */
sonuc = PQexec(baglanti, "COPY ...");

/* Gelen sonucu kontrol ediyoruz. */
if (PQresultStatus(sonuc) != PGRES_COPY_...) {
    fprintf(stderr, "COPY sorgusu sonucu beklenen durum dönmedi!\n");
    ...
} else
    PQclear(sonuc); /* Bundan sonra işimize yaramayacak sonuç yapısını temizliyoruz. */

/*
 * COPY isteğimize göre, libpq kütüphane fonksiyonlarını
 * kullanarak veri aktarımını gerçekleştiriyoruz.
 */

/* (Veri gönderiminde bulunduysak COPY işlemini sonlandırıyoruz.) */

/* COPY bittikten sonra, son durumu öğreniyoruz. */
sonuc = PQgetResult(conn);
if (PQresultStatus(sonuc) != PGRES_COMMAND_OK) {
    fprintf(stderr, "COPY sonucu beklenmeyen bir hata oluştu!\n");
    ...
} else {
    printf("COPY sorugusu başarı ile tamamlandı.\n");
    PQclear(sonuc);
}
```

İlk önce nasıl veri alabileceğimize bakalım. Herhangi bir COPY tabloadi TO STDOUT tipinde sorgu sonucu dönecek olan veriyi PQgetCopyData() fonksiyonu yardımıyla alabiliriz. Bu fonksiyonu inceleyecek olursak:

```
int
PQgetCopyData(PGconn  *baglanti,
              char   **katar_isaretcisi,
              int      baglanti_tipi);
```

PQgetCopyData() fonksiyonu, PGRES_COPY_OUT durumunda olan bir PGresult sonuç yapısından dönen veriyi, her çağrılışında bir satır okumak suretiyle, bellekte ayırdığı bir karakter alanına atıp, fonksiyona paramtere olarak geçilen katar işaretçisini, ayırdığı bu alanı gösterecek şekilde değiştirdikten sonra, okuduğu satır uzunluğunu (her zaman 0'dan büyük olacak şekilde) kendisini çağırana döndürür.

COPY komutunun sonucu PGRES_COPY_OUT değilse, fonksiyon çalışmayacaktır.

Sonuç olarak, o an için hiçbir veri yoksa (bağlantı tipi asenkron olarak belirtildiğinde gerçekleşir bu durum) 0, verinin sonuna gelindiyse -1 ve bir hata durumunda da -2 değerini döndürecektir.

> *Fonksiyona parametre olarak geçilen katar işaretçisi `NULL` olmamalıdır ve döndürülecek olan katar ile işimiz bittiğinde ise `PQfreemem()` ile bellekten bırakılması gerekir.*

Fonksiyona parametre olarak girilen bağlantı tipi eğer 1 ise bağlantı asenkron, 0 ise senkron olarak algılanacaktır. Senkron bir bağlantıda `PQgetCopyData()` gelecek veri için bekleyecek olurken, asenkron bir bağlantıda okunacak bir veri o an mevcut değilse fonksiyon 0 değerini döneceğinden beklenmek zorunda kalınmayacaktır. Onun yerine soket bir kez daha okunabilir olana kadar uygulama akışı içinde programa devam edilip, soket tekrar okunabilir olduğunda - `PQconsumeInput()` ile gelen veri alındıktan sonra - `PQgetCopyData()` fonksiyonu yeniden çağırılabilir.

Genel bir `PQgetCopyData()` işleyişinin iskeleti şu şekilde olacaktır.

```
PGresult *sonuc;
int      cpsonuc;
char     *satir;

/* COPY isteğimizi gönderiyoruz. */
sonuc = PQexec(baglanti, "COPY ornektablo TO STDOUT");
if (PQresultStatus(sonuc) != PGRES_COPY_OUT)
    /* Beklenmeyen bir hata oluştu. */
else
    PQclear(sonuc);

/*
 * Burada soketin okunabilir olduğu ile ilgili kod kısmını atlamak
 * için PQgetCopyData() fonksiyonunu senkron olarak çağırıyoruz.
 */
while ((cpsonuc = PQgetCopyData(conn, &satir, 0)))
{
    if (cpsonuc == -1)
        /* COPY aktarımının sonuna gelindi. */
    else if (cpsonuc == -2)
        /* Hata oluştu. */

    PQfreemem(satir);
}

/* Son durum PQgetResult() ile alınıp kontrol edilecek. */
```

`COPY` komutu ile veri alımının nasıl olacağı hakkında bahsettiğimize göre, şimdi de `COPY tabloadi FROM STDIN` tipinde bir sorgu ile sunucuya nasıl veri gönderebileceğimize bakalım. Bunun için `COPY` sorgusunu başlattığımızda dönen `PGresult` yapısının durumunun `PGRES_COPY_IN` olduğundan emin olduktan sonra aşağıdaki fonksiyonları kullanabiliriz.

```
int
PQputCopyData(PGconn     *baglanti,
              const char *katar,
              int         katar_uzunlugu);
```

`PQputCopyData()` fonksiyonu, `PGresult` yapısı `PGRES_COPY_IN` dönen bir `COPY` sorgu başlangıcı sonucunda, parametre olarak aldığı katarın, belirtilen uzunluktaki kısmını sunucuya göndermeye çalışır.

> *`COPY` komutunun sonucu `PGRES_COPY_IN` değilse, fonksiyon çalışmayacaktır.*

Başarı olduğu durumda 1, verinin hepsinin gönderilemediği durumda (ki bu ancak

asenkron bir bağlantıda gerçekleşebilir) 0 ya da gönderimin başarısız olduğu durumda -1 değerini kendisini çağırana döndürecektir.

> *PQputCopyData() sonucunun 0 döndüğü bir durumda, PQflush() kullanıldıktan sonra soket tekrar yazılabilir duruma gelene kadar beklenerek gönderim tekrarlanır.*

PQputCopyData() fonksiyonu ile veri uzunluğunu belirtikten sonra istediğiniz uzunlukta katar gönderimini sağlayabileceğiniz gibi, bunu parçalar halinde göndermeyi de deneyebilirsiniz. Sadece gönderdiğiniz katarın COPY komutunca anlaşılabilecek bir yapıyı (ayraçlar, yeni satır (\n) karakterleri, vs.) sağladığından emin olmasınınız.

```
int
PQputCopyEnd(PGconn *baglanti, const char *hata_mesajı);
```

PQputCopyEnd() fonksiyonu, PGRES_COPY_IN durumunda olan bir bağlantıda, PQputCopyData() fonksiyonu ile başladığımız veri aktarımının bittiğini sunucuya bildirir.

> *PQputCopyEnd() kullanılarak veri aktarımının bitiminin belirtildiği bir durumda, gönderilen verinin en son satırında \. katarının kullanımına gerek kalmamaktadır.*

Parametre olarak girilen hata mesajı NULL ise veri aktarmının sona erdiği bildirilirken, NULL'dan farklı bir katar içeriyorsa, veri aktarımı kasıtlı olarak kesilip iptal edildiği anlamına gelir ve hata mesajı olarak da girilen parametre kullanılır.

> *Sunucudan dönecek olan hata mesajı katarında, eğer sunucuda PQputCopyEnd() çağrılmadan daha önce bir hata oluşmuşsa parametre olarak gönderdiğiniz hata mesajı yerine, sunucunun kendi döndürdüğü hata mesajı kullanılacaktır.*

Fonksiyonun başarılı olması durumunda 1, sonlandırma bildiriminin tamamen iletilememesi durumunda 0 (ki bu ancak asenkron bir bağlantıda meydana gelebilir), bir hata durumunda ise -1 değeri döndürülecektir.

> *Veri aktarımının kesilip iptal edilmesi özelliği 3.0 öncesi protokollerde çalışmayacaktır. 3.0 öncesi protokollerde PQputCopyEnd() ile birlikte bir hata mesajının belirtildiği durumda function requires at least protocol version 3.0 şeklinde bir mesaj düşülecek ve COPY işlemi sonlandırılmayacak olup -1 değeri döndürülecektir. Bu yüzden 2.0 ve öncesi protokollerde COPY ile veri iletiminin sonlandırılması için hata mesajı değeri olarak NULL gönderilmedilir.*

PQputCopyData() ve PQputCopyEnd() fonksiyonlarını kullanarak oluşturulan bir COPY ile veri iletiminde, programın işleyiş mantığı veri alırken uyguladığımıza çok benzerdir. Örnek bir program iskeleti oluşturacak olursak:

```
PGresult *sonuc;
int       cpsonuc;
char     *s;

/*
 * Karşı tarafa COPY ... FROM STDIN isteğimiz gönderilir.
 * Dönen sonuç PGRES_COPY_IN olduğu durumda programın
 * normal akışına aynen devam edilir.
 */

/*
 * Soketin yazılabilir olduğuna dair işlemler için gerekli koddan
 * kaçınıp, senkron bir veri transferi kuruyoruz.
 */
if (PQsetnonblocking(baglanti, 0) != 0)
{
    fprintf(stderr, "Bağlantı asenkron hale getirilirken hata oluştu.\n");
    ...
}
```

```
/*
 * Çoğu zaman, döngüye girilerek s içindeki veri değiştirilmek
 * sureti ile, PQputCopyData() ard arda çağrılır.
 */
while (...)
{
    /* s içine gerekli katarın atamasını gerçekleştiriyoruz. */

    /*
     * Gönderdiğimiz verinin karakter katarı olduğunu farzedip,
     * katar uzunluğu için strlen() fonksiyonunu kullanıyoruz.
     */
    cpsonuc = PQputCopyData(baglanti, s, strlen(s));
    if (cpsonuc != 1)
    {
        fprintf(stderr, "PQputCopyData() hata döndürdü! [%d]\n%s",
                cpsonuc, PQerrorMessage(baglanti));
        ...
    }
    ...
}

/*
 * Veriyi karşı tarafa başarı ile ilettikten sonra,
 * gönderimin sonlandığını bildiriyoruz.
 */
cpsonuc = PQputCopyEnd(baglanti, NULL);
if (cpsonuc != 1)
{
    fprintf(stderr, "PQputCopyEnd() hata döndürdü! [%d]\n%s",
            cpsonuc, PQerrorMessage(baglanti));
    ...
}
else
{
    printf("Veri aktarımı başarı ile sonlandırıldı.\n");
    ...
}

/* PQgetResult() ile son duruma bakıp program akışına devam ediyoruz. */
```

8. Sorgu Sonuçları Üzerinde İşlemler

Bu bölüme kadar, veritabanı ile nasıl başarılı bir bağlantı kurup, bunu nasıl denetleyeceğimiz ve ardından bu bağlantı üzerinden veritabanı sunucusu üzerinde nasıl sorgulama işlemleri gerçekleştirebileceğimiz üzerinde durmaya çalıştık. Bu bölümde ise, yapılan sorgular sonucunda dönen yapıları işleyen fonksiyonlar hakkında bahsedeceğiz.

> *Her tablonun satır ve sütun kısmı da bir dizi gibi düşünülmesi gerektiğinden (dizilerin indislerinin 0'dan başladığı göz önüne alınacak olursa) inceleyeğimiz tüm satır ve sütun numarası içeren fonksiyonların, döndürdükleri ve parametre olarak alacakları satır ve sütun numaralarının 0'dan başlayacak olmasına dikkat ediniz.*

8.A. Sonuç Hakkında Durum Bilgisi

Sorgu sonucu, `PGresult` yapısı içinde dönen bir sonuç hakkında durum bilgisi elde etmek için bu bölümde tanıtılacak fonksiyonları kullanabilirsiniz.

```
ExecStatusType
PQresultStatus(const PGresult *sonuc);
```

`PQresultStatus()` fonksiyonu, sunucu üstünde çalıştırılan bir sorgunun sonuç durumunu öğrenmemizi sağlar. Döndürdüğü `ExecStatusType` numaralandırılmış dizisi ve dizi elemanlarının açıklaması şu şekildedir:

```
typedef enum
{
    PGRES_EMPTY_QUERY    /* Sunucuya gönderilen SQL komutu boş ya da
                            herhangi bir sorgu içermiyor. */
    PGRES_COMMAND_OK     /* Sorgu başarı ile gerçekleşti ve sonuç
                            olarak bir veri geri döndürmedi. */
    PGRES_TUPLES_OK      /* Sorgu başarı ile gerçekleşti ve
                            sonuç olarak geri veri döndürdü. */
    PGRES_COPY_IN        /* COPY komutu ile sunucu tarafına
                            veri aktarma konumunda. */
    PGRES_COPY_OUT       /* COPY komutu ile sunucudan istemciye
                            veri alımı konumunda. */
    PGRES_BAD_RESPONSE   /* Sunucu tarafından döndürülen
                            değer anlaşılamadı. */
    PGRES_NONFATAL_ERROR /* Kritik öneme sahip olmayan bir hata
                            (hatırlatma ya da uyarı) oluştu. */
    PGRES_FATAL_ERROR    /* Kritik bir hata oluştu. (Bu tür uyarılar, ileriki
                            bölümlerde göreceğimiz, uyarı işlemcilerine aktarılır.) */
} ExecStatusType;

char *
PQresStatus(ExecStatusType durum);
```

`PQresStatus()` fonksiyonu, `PQresultStatus()` tarafından döndürülen listece numaralandılırılmış (*enumarated*) durum değerlerinin, kelime katarı hallerini (`PGRES_EMPTY_QUERY`, `PGRES_COMMAND_OK`, vs. gibi) kendisini çağırana döndürür.

```
char *
PQresultErrorMessage(const PGresult *sonuc);
```

`PQresultErrorMessage()` fonksiyonu, kendisine parametre olarak geçilen sonucun yapılan en son sorgulamasında herhangi bir hata oluşmuşsa, ilgili hata mesajını; sorgu sorunsuz gerçekleşmişse boş bir karakter katarını döndürür. (Dönen hata mesajı yeni satır karakteri içerecektir.)

> *`PQexec()` yada `PQgetResult()` ile bir sorgu sonucunda oluşan hatanın hemen ardından `PQerrorMessage()` fonksiyonu çağrısı, `PQresultErrorMessage()` ile aynı hata mesajını döndürecektir. Fakat hata mesajının elde edildiği `PGresult` yapısı bellekten bırakılana kadar sahip olduğu hata mesajını koruyacak olurken, `PQerrorMessage()` fonksiyonunun tuttuğu mesaj her sorgu sonucunda değişecektir.*

```
char *
PQresultErrorField(const PGresult *sonuc, int alan_kodu);
```

`PQresultErrorField()` fonksiyonu, herhangi bir sorgulama sonucunda doğan hata sebebiyle oluşan hata metninin istenilen alanını kendisini çağırana döndürecektir. (Dönen hata mesajı yeni satır karakteri içermeyecektir.) Çağrı sonucu dönen bir `NULL` değeri böyle bir hata alanı bulunmadığını, ya da parametre olarak gönderilen sorgulama sonucunda hata veya bir uyarının oluşmadığını belirtir.

Fonksiyona parametre olarak göndermek üzere aşağıdaki alan kodlarını kullanabilirsiniz.

`PG_DIAG_SEVERITY`	Dönen mesajın önemini içerir. Eğer dönen bir hata mesajı ise `ERROR`, `FATAL`, `PANIC` değerlerinden birini; bir uyarı mesajı ise `WARNING`, `NOTICE`, `DEBUG`, `INFO`, `LOG` değerlerinden birini yada bunların yerel değişkenlerce tanımlanmış çevirilerini döndürür.

PG_DIAG_SQLSTATE	Hata yada uyarının SQLSTATE kodunu[11] döndürür.
PG_DIAG_MESSAGE_PRIMARY	Kullanıcı tarafından anlaşılabilir tek satırlık açıklama.
PG_DIAG_MESSAGE_DETAIL	Sorun hakkında daha fazla bilgi içeren isteğe bağlı (genelde bir satırdan daha fazla olan) açıklama.
PG_DIAG_MESSAGE_HINT	Sorunun neden kaynaklı olup nasıl çözülebileceği hakkında ipucu. (Uzunluğu bir satırı aşabilir.)
PG_DIAG_STATEMENT_POSITION	Orjinal *SQL* komutu içinde hata ya da uyarının kaçıncı karakterde gözlendiği.
PG_DIAG_INTERNAL_POSITION[12]	Kullanıcının girdiği *SQL* komutu yerine istemcinin kendi gönderdiği (iç mekanizma sonucu işletilen) komutta hatanın bulunduğu karakterinin indisi. (Bu indis tanımlı olduğu sürece, PG_DIAG_INTERNAL_QUERY değeri de gözükecektir.)
PG_DIAG_INTERNAL_QUERY[12]	İç mekanizma sonucu işletilen komutta meydana gelen hata metni. (Örneğin bir PL/pgSQL fonksiyonu tarafından çağrılan *SQL* sorgusu.)
PG_DIAG_CONTEXT	Hatanın oluştuğu komut bağlamının indislenerek satır satır görüntülenmesi.
PG_DIAG_SOURCE_FILE	Hatanın meydana geldiği kaynak dosyasının ismi.
PG_DIAG_SOURCE_LINE	Kaynak dosyasında hataya yol açan sorgunun bulunduğu satır.
PG_DIAG_SOURCE_FUNCTION	Kaynak dosyasında hatayı raporlayan fonksiyon.

Alan kodları, sunucu ve istemci arasındaki hata iletiminde veri paylaşımını sağlamak için oluşturulmuş olup, kodların tam bir listesine `include/postgres_ext.h` (ya da `include/postgresql/server/postgres_ext.h`) dosyasından da ulaşabilirsiniz.

8.B. Sonuç Hakkında Tablo/Satır/Sütun Bilgisi

Herhangi bir sorgulama sonucu dönen PGresult yapısı üzerinden tablo/satır/sütun bilgilerini bu bölümde tanıtılacak olan fonksiyonları kullanarak öğrenebilirsiniz.

```
char *
PQcmdStatus(PGresult *sonuc);
```

PQcmdStatus() fonksiyonu, parametre olarak aldığı sorgu sonucunu oluşturan *SQL* komutunun PostgreSQL tarafından aldığı son cevabı kendisini çağırana karakter kataı işaretçisi olarak döndürür. (Çıktı genellikle her komut için değişecek olup, psql ile herhangi bir sorgu sonrası en alt satırda yer alan durum bilgisi benzer gelir. Örneğin basit bir INSERT komutu sonucunda PQcmdStatus() fonksiyonunun döndürdüğü karakter kataı INSERT 17266 1 iken, basit bir SELECT sorgusunu sonucunda sadece SELECT döndürecektir.)

Aşağıda tanıtılacak olan `PQcmdTuples()`, `PQoidValue()` ve `PQoidStatus()` fonksiyonları, aslında `PQcmdStatus()` fonksiyonundan dönecek çıktı üzerinde gerekli değişiklikleri yapan fonksiyonlardan ibarettir. Bu nedenle, benzer rutinleri kendi programınızda da yazmanız mümkün.

```
char *
PQcmdTuples(PGresult *sonuc);
```

PQcmdTuples() fonksiyonu, parametre olarak aldığı sorgu sonucunda kaç satırın etkilendiğini kendisini çağırana karakter kataı olarak döndürecektir. Eğer sorgu sonucu, satırlarda herhangi bir etki yaratmamışsa, fonksiyon kendisini çağırana boş bir karakter katar döndürür.

11 Tüm bir kod listesi için kitabın arkasında yer alan PostgreSQL Hata Kodları kısmına bakabilirsiniz.

12 Bu özellik PostgreSQL 8.0 ve daha üstü sürümlerce desteklenmektedir.

Herhangi bir sorgunun tablo satırlarını etkilemesi için bu sorguda INSERT, DELETE, FETCH, MOVE, UPDATE komutlarından en az birinin kullanılmış olması ya da bu komutlardan en az birini içeren hazırlanmış bir sorgunun EXECUTE ile çağrılması gerekmektedir.

```
Oid
PQoidValue(const PGresult *sonuc);
```

PQoidValues() fonksiyonu, parametre olarak gönderilen sonucu greçekleştiren ilgili *SQL* tümcesinde yer alan INSERT komutunun (ya da INSERT içeren hazırlanmış bir sorgunun EXECUTE ile) çalıştırılması sonucu (OID değerleri olan bir tabloya) sadece bir tek satır eklenmesi durumunda, fonksiyon kendisini çağırana bu eklenen satırın OID değerini döndürür. Satırın eklendiği tablo herhangi bir OID değeri içermiyorsa, fonksiyon InvalidOid değeri döndürecektir.

```
char *
PQoidStatus(const PGresult *sonuc);
```

PQoidStatus() fonksiyonu, bir üstte yer alan PQoidValue() fonksiyonu ile aynı işleve (ve kısıtlamalara) sahip olup, döndürdüğü OID değeri karakter katarı içinde gelir. Herhangi bir kısıtlama durumunda PQoidValue() gibi InvalidOid değeri döndürmek yerine, boş bir karakter katarı döndürecektir.

```
Oid
PQftable(const PGresult *sonuc, int sutun_numarasi);
```

PQftable() fonksiyonu, kendisine parametre olarak gönderilen sonuçtaki, ilgili sütunun geldiği tablonun sahip olduğu OID değerini kendisini çağırana döndürür. Hatalı bir sütun numarası girildiğinde, 3.0 öncesi protokoller kullanıldığında ya da ilgili sütunun tek bir tablo tarafından oluşturulmadığı durumlarda fonksiyon InvalidOid değerini döndürecektir.

Dönecek olan OID değerini kullanarak tablonun tam ismine şu yöntemlerden birini kullanarak ulaşabilirsiniz:

```
-- Dönen tablo OID değerinin 18647 ve tablo
-- adının ornektablo olduğunu farzedersek:
=> SELECT 18647::regclass;
 regclass
----------
 ornektablo
(1 row)

-- Namespace değeri ile birlikte alıyoruz.
=> SELECT n.nspname, c.relname
->   FROM pg_catalog.pg_class AS c
->     LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
->   WHERE pg_catalog.pg_table_is_visible(c.oid)
->     AND c.oid = 18647;
 nspname | relname
---------+---------
 public  | ornektablo
(1 row)
```

```
int
PQftablecol(const PGresult *sonuc, int sutun_numarasi);
```

PQftablecol() fonksiyonu, parametre olarak aldığı sonucun belirtilen sütununun, üretildiği tablodaki sütun indisini kendisini çağırana döndürecektir. Hatalı sütun numarası girildiği, 3.0 öncesi protokoller kullanıldığı ya da ilgili sütunun tek bir tablo tarafından oluşturulmadığı gibi durumlarda fonksiyon 0 sunucunu döndürür.

```
Oid
PQftype(const PGresult *sonuc, int sutun_numarasi);
```

`PQftype()` fonksiyonu, parametre olarak aldığı sonucun, yine parametre olarak aldığı sütun numarasının hangi tablonun hangi sütununundan döndüğünü bulduktan sonra, o sütun tipinin `OID` değerini kendisini çağırana döndürecektir.

Sütun tipinin dönecek olan `OID` değerine bakarak, tip ismini şu şekilde bulabilirsiniz:

```
-- Örneğin bytea alanı 17 OID değerine sahiptir:
=> SELECT typname FROM pg_type WHERE oid = 17;
 typname
---------
 bytea
(1 row)
```

```
int
PQfmod(const PGresult *sonuc, int sutun_numarasi);
```

`PQfmod()` fonksiyonu, sonucun üretildiği tablodaki, ilgili sütunun niteleyicisini döndürür. Niteleyiciler veri tiplerine bağlı olup, veri tipleri hakkında anlamlı rakam kesinliği (*precision*) ve boyut sınırı bilgilerini içermelerine rağmen, şu an için çoğu veri tipinin niteleyicisi bulunmamaktadır. Bu durumda, çoğu veri tipi için `PQfmod()` fonksiyonu -1 değerini dönecektir.

`PQfmod()` fonksiyonunu kullanarak bir tablo alanının boyut sınırı bilgilerini öğrenebileceğinizi belirtmiştik. Fakat boyut sınırı bilgilerinin ediniminde bir kaç istisnai durum bulunmaktadır. Şöyle ki:

Alan Tipi:	**`PQfmod()` sonucu dönecek olan uzunluk:**

`bit(n)`
`bit varying(n)`
`character(n)`
`character varying(n)`
	`n + VARHDRSZ`

`numeric(p, s)`	`(p << 16) + s + VARHDRSZ`
`time(p)` `timestamp(p)`	`p`

Yukarıdaki tabloda `VARHDRSZ` ile kast edilen, değişken başlıklarını tutmak için kullanılacak bellek alanının uzunluğudur ve (platformdan bağımsız olarak) daima 4 değeri dönecektir. `include/postgresql/server/c.h` dosyasındaki ilgili satırlardan bazılarını kopyalayacak olursak:

```
#typedef signed int int32;
...
/* ----------------
 *      Variable-length datatypes all share the 'struct varlena' header.
 *
 * NOTE: for TOASTable types, this is an oversimplification, since the value
 * may be compressed or moved out-of-line.  However datatype-specific routines
 * are mostly content to deal with de-TOASTed values only, and of course
 * client-side routines should never see a TOASTed value.  See postgres.h for
 * details of the TOASTed form.
 * ----------------
 */
struct varlena
{
    int32  vl_len;
    char   vl_dat[1];
};
#define VARHDRSZ ((int32) sizeof(int32))
```

```
char *
PQfname(const PGresult *sonuc, int sutun_indisi);
```

`PQfname()` fonksiyonu, parametre olarak aldığı sonucun belirtilen indise sahip sütunun başlığını döndürür. Belirtilen indis toplam sütun sayısından fazla ya da girilen sonuç geçersiz ise fonksiyon `NULL` değeri döndürecektir.

```
int
PQfnumber(const PGresult *sonuc, const char *sutun_basligi);
```

`PQfnumber()` fonksiyonu, başlığı girilen sütunun, sonuç tablosu içindeki indisini kendisini çağırana döndürür. Başlık ile örtüşen sütun değeri bulunmadığı taktirde -1 değeri dönecektir. Parametre olarak girilen sonuç tablosunda aynı başlığa sahip birden fazla alan bulunması durumunda fonksiyon bunlardan ilkinin indisini döndürecektir.

> *`PQfnumber()` fonksiyonu, girilen sütun başlığını bir SQL ifadesi içindeymişcesine işleyip, ona göre çift tırnak içine alınan değerleri ayrı değerlendirecektir. Fakat çift tırnak içine alınmayan karakterlerin küçük harfe çevrilmesi esnasında sunucu ve istemci tarafındaki farklı yerel (`locale`) ayarları sorun yaratabilir.*

Fonksiyon her ne kadar açık olsa da, girilen sütun başlığı değerine göre öne çıkan bir kaç istisnai durum mevcuttur. Bunları bir örnek üzerinde gösterelim:

```
sonuc = PQexec(bag, "SELECT 1 as BASLIK1, 2 as \"basLIK2\"");

/* İlk önce indislerine göre cağırıyoruz. */
printf("%s\n", PQfname(bag, 0));                /* baslik1 */
printf("%s\n", PQfname(bag, 1));                /* BASLIK2 */

/* Şimdi ise başlığa göre çağırıp, indisini öğreniyoruz. */
printf("%d\n", PQfnumber(sonuc, "BASLIK1"));     /*  0 */
printf("%d\n", PQfnumber(sonuc, "baslik1"));     /*  0 */
printf("%d\n", PQfnumber(sonuc, "BASLIK2"));     /* -1 */
printf("%d\n", PQfnumber(sonuc, "\"basLIK2\"")); /*  1 */
printf("%d\n", PQfnumber(sonuc, "bas\"LIK2\"")); /*  1 */
printf("%d\n", PQfnumber(sonuc, "BAS\"LIK2\"")); /*  1 */
```

```
int
PQfsize(const PGresult *sonuc, int sutun_numarasi);
```

`PQfsize()` fonksiyonu, parametre olarak girilen sonucun belirtilen sütununun sahip olduğu veri tipinin, veritabanının kendi iç gösterimi esnasında ne kadar yer kapladığını gösterir. Geri dönen negatif bir değer, veri tipinin değişken uzunlukta[13] olduğu anlamına gelecektir.

```
int
PQnfields(const PGresult *sonuc);
```

`PQnfields()` fonksiyonu, parametre olarak aldığı sonuç tablosundaki toplam sütun sayısını kendisini çağırana döndürecektir.

```
int
PQntuples(const PGresult *sonuc);
```

`PQntuples()` fonksiyonu, sorgu sonucunda dönen toplam satır sayısını kendisini çağırana döndürecektir.

13 Burada değişken uzunluktaki tipler ile kast edilen, `varlena` tipleridir. Değişken uzunluktaki tiplerin tam bir listesini `pg_type` tablosunda `typlen`'in -1 değerini aldığı satırları döndüren bir sorgu ile öğrenebilirsiniz.

```
int
PQgetlength(const PGresult *sonuc,
            int             satir_numarasi,
            int             sutun_numarasi);
```

`PQgetlength()` fonksiyonu, parametre olarak aldığı sonuç tablosu üzerindeki belirtilen satır ve sütunun boyutunu kendisini çağırana döndürür. Katar tipindeki alanlar için `strlen()` fonksiyonu gibi işlev gösterirken, ikili biçimde veri içeren alanlar için ise alanın bayt cinsinden boyutunu gösterecektir.

```
int
PQbinaryTuples(const PGresult *sonuc);
```

`PQbinaryTuples()` fonksiyonu, parametre olarak aldığı sonuç tablosunda dönen tüm değerler ikili biçimde ise 1 değerini döndürür. Aksi tüm durumlarda (sonucun karakter katarından oluşuyor olması ya da bazı sütunlarının karakter katarı olması durumunda) fonksiyon 0 değeri döndürecektir.

> *Bir sonuçta hem karakter katarı, hem de ikili biçimde veri bulunabileceği düşünüldüğünde, `PQbinaryTuples()` fonksiyonunun yetersiz kaldığı noktalar olmaktadır. Fakat bu PostgreSQL tarafından kullanılan protokolün bir kısıtlaması olmayıp, libpq tarafından arayüzü karmaşık hale getirmemek için geliştirilmemiştir.*

```
int
PQgetisnull(const PGresult *sonuc,
            int             satir_numarasi,
            int             sutun_numarasi);
```

`PQgetisnull()` fonksiyonu, belirtilen sonucun, ilgili satır ve sütunundaki değerin `NULL` olması durumunda 1, aksi halde 0 döndürecektir.

```
char *
PQgetvalue(const PGresult *sonuc,
           int             satir_numarasi,
           int             sutun_numarasi);
```

`PQgetvalue()` fonksiyonu, ilgili sonucun belirtilen satır ve sütunundaki değerini kendisini çağırana karakter katarı olarak döndürür.

> *Eğer belirtilen satır ve sütündaki değer `NULL` ise, fonksiyon boş bir karakter dizisi dödüreceğinden, bu tür bir örtüşmenin olmaması için ilgili sütunda `NOT NULL` niteleyicisi bulundurulmalı ya da `PQgetisnull()` fonksiyonu kullanılmalıdır.*

Karakter katarı biçiminde veri içeren alanlar için, `EOF` (`\0`) ile sonlandırılmış bir dize işaretçisi döndürülecektir. Belirtilen satırın ilgili sütunu içindeki veri ikili biçimde ise, geri dönecek değer, ilgili veri tipinin `typsend`[14] fonksiyonu ile belirlenip yine ikili biçimde geri döndürülecektir.

PostgreSQL birçok veri tipini kendi içinde destekliyor olsa da, herhangi bir sorgu sonucu dönen verinin *API* kullanarak alımında bu tiplerden sadece çok azı kullanılan programlama dilleri tarafından standart olarak desteklenmektedir. Örnek vermek gerekirse, şöyle bir tablomuz olduğunu varsayalım:

```
-- PostgreSQL tarafından desteklenen, fakat verinin API kullanarak
-- alımında kısıtlamalar yaratacak tipler içeren örnek tablo.
```

14 `typsend` ve `typreceive` fonksiyonları sunucunun istemci ile arasındaki veri iletişimde alınıp gönderilen verinin uygun biçime çevrilmesinde kullanılır. Hangi tipin hangi `typsend` ve `typreceive` fonksiyonlarına sahip olduğunu ise `SELECT typsend, typreceive FROM pg_type WHERE typname = 'tip_ismi'` sorgusu ile öğrenebilirsiniz.

```
=> CREATE TABLE ornek_tipler (
->   daire      circle,
->   mac_adresi macaddr,
->   ip_adresi  cidr,
->   dizi       int[3]
-> );

-- Tabloya örnek veri giriyoruz.
=> INSERT INTO ornek_tipler (daire, mac_adresi, ip_adresi, dizi)
-> VALUES (
->   '<(1,2),3>',                          -- circle
->   '00:C0:49:D5:9D:8B',                  -- macaddr
->   '2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128', -- cidr
->   '{1,2,3}'                             -- int[3]
-> );

-- Tablonun son durumuna bakıyoruz.
=> \x
Expanded display is on.
=> SELECT * FROM ornek_tipler;
-[ RECORD 1 ]-----------------------------------
daire      | <(1,2),3>
mac_adresi | 00:c0:49:d5:9d:8b
ip_adresi  | 2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128
dizi       | {1,2,3}
```

Tablonun içinde ne kadar farklı tipte veri olursa olsun, biz bunları `PQgetvalue()` ile sadece karakter katarı cinsinde elde edebiliyoruz. Aslında bu libpq'nun bir kısıtlamasından çok, kullanılan tiplerin standart C kütüphanelerince desteklenmemesinden kaynaklanıyor. Bu nedenle karakter katarı dışında alınacak veri tipleri `PQgetvalue()` ile bir katar işaretçisine yazıldıktan sonra, onun üzerinde uygulamamızın ihtiyacına göre işlem yapmak zorunda kalıyoruz. (Örneğin `atoi()`, `atof()` fonksiyonlarını kullanmak gibi.)

Diğer birçok *API*'nin de arka tarafta libpq kullandığı düşünülecek olursa, aynı kısıtlama onlar için de geçerli hale geliyor. Fakat diğer programlama dillerinin sağladıkları arayüzler, alınan veriyi isteğe göre çeşitli tiplere uyarlamasını başarabilmektedir. (Örneğin, bir sonraki bölümüde üzerinde duracağımız PHP programlama dili, tam, reel ve ondalık sayıları otomatik olarak algılayabilme özelliğine sahiptir.)

Peki madem karakter tipileri (`varchar`, `char`, `text`, vs.) dışında kalan tipler, bu kadar çok kısıtlama içeriyorsa, neden veritabanında farklı tipler kullanalım ki? Sonuç itibari ile veriyi veritabanında saklamamızın nedeni, sonradan başka bir yerde onu alıp kullanacak olmamız; alım esnasında böyle bir kısıtlama olduktan sonra ne diye bu kadar çok farklı tip kullanarak vakit kaybedelim? Aslında tam aksine, uygun veri için uygun tipin kullanımı size çok büyük bir zaman kazancı sağlayacaktır. Farklı tipleri *API* ile çağırırken tek bir biçimde dönüyor olması, bunların veritabanı içinde kullanımlarında da tekbir biçim baz alındığı anlamına gelmez. Örneğin sayıları büyüklük, küçüklük ve daha bir çok mantıksal operatör ile kıyaslayabilirken, aynı şeyi karakter katarları içine kayıtlı sayılar ile yapmanız epey zor olacaktır. Bu verilebilecek en basit ve yüzeysel örnek. Bunların çok daha karışık kombinasyonları da oluşturulabilir. Yine basit bir emsal vermek gerekirse:

```
-- <(3,2),1> ve <(7,8),2> çemberleri arasındaki uzaklık
=> SELECT circle '<(3,2),1>' <-> circle '<(7,8),2>';
     ?column?
------------------
 4.21110255092798
(1 row)
```

Durumu özetlemek gerekirse, veritabanı tasarımında bir alanın hangi tipte veri barındıracağı, hangi verinin hangi tablo altında saklanacağından sonra doğru atılması gereken en önemli adımdır. Aynı biçimde veri içerecek alanların uygun tipler altında saklanması durumunda, veritabanı tarafından sağlanan yardımcı araçları kullanılarak, çok daha komplike sorgu tümceleri oluşturulabeceği gibi, programcının boş yere bunları kendi başına işlemesi esnasında oluşacak zaman kaybından da kurtulunmuş olur.

8.C. Sonuç Üzerinde İşlemler

Başarılı bir sorgu sonucu dönen yapı üzerinde işlem yapmak için bu bölümde tanıtılacak olan fonksiyonları kullanabilirsiniz.

```
void
PQprint(FILE             *dosya,
        const PGresult   *sonuc,
        const PQprintOpt *secenek);
```

PQprint() fonksiyonunu, herhangi bir sorgu sonucu dönen sonuçları belirli bir biçimde dosyaya yazdırmak için kullanabilirsiniz.

Fonksiyona `PQprintOpt` tipinde parametre olarak göndereceğiniz seçenekler ile, alacağınız çıktının biçiminde çeşitli oynamalar yapabilirsiniz. Bunların açıklamak için `PQprintOpt` yapısına bakacak olursak:

```
typedef char pqbool;
...
typedef struct {
    pqbool    header;      /* Sütun başlıkları ile toplam satır sayısını da yazdır. */
    pqbool    align;       /* Sütunları hizala. */
    pqbool    standard;    /* Standart çıktı al. */
    pqbool    html3;       /* HTML tabloları kullan. */
    pqbool    expanded;    /* Tabloları veriye göre genişlet. */
    pqbool    pager;       /* Gerek duyulduğunda çıktıyı sayfalara ayır. */
    char     *fieldSep;    /* Kullanılacak alan ayıracı. */
    char     *tableOpt;    /* HTML tablosunda kullanılacak <table> için özellikler. */
    char     *caption;     /* HTML tablo başlığı. */
    char    **fieldName;   /* Sütun adlarının yazılı olduğu dizi. */
} PQprintOpt;
```

> *`PQprint()` fonksiyonu tüm verinizin karakter katarı olduğunu farzedip, parametre olarak adlığı dosyaya o şekilde yazacaktır.*

```
void
PQclear(PGresult *sonuc);
```

`PQclear()` fonksiyonu, işi bitmiş bir sorgu sonucu işaretçisini bellekten boşaltmak için kullanılır. Bağlantıyı kapatsanız dahi, sonuç nesnesi (`PGresult`) olduğu gibi kalacaktır. Bu yüzden herhangi bir sorgu sonucunu işiniz bittiği zaman bellekten temizlememeniz yazdığınız yazılımda hafıza açıklarına neden olabilir.

9. Asenkron Uyarılma

`LISTEN` ve `NOTIFY` komutlarını kullanarak veritabanına bağlı istemcilerin herhangi bir işlem sonrası (dinledikleri veritabanı nesnelerine göre) asenkron olarak uyarılmalarını sağlayabilirsiniz.

Bu mekanizma daha çok, veritabanında yapılan bir değişiklik sonucu, değişikliğin yapıldığı nesneyi dinleyen istemcilerin uyarılması için kullanılır. (Örneğin belirli bir tabloya yeni bir satır eklenmesi sonucu, bu tabloyu dinleyen istemcilerin uyarılması gibi.) İstemcilere sadece bir uyarı mesajı yollanıp herhangi bir ek bilgi gönderilmeyecektir.

> *`LISTEN` ve `NOTIFY` komutları hakkında ayrıntılı bilgi için kitabın ilgili bölümüne göz atınız.*

Asenkron uyarılma ile ilgili olarak libpq kütüphanesi bize `PQnotifies()` fonksiyonunu sunmaktadır. `PQnotifies()` kullanılarak, dinlenen yapıdan herhangi bir uyarı alınması

durumunda, gelen uyarının istemci tarafından nasıl öğrenilebileceği hakkında bahsedelim.

```
PGnotify *
PQnotifies(PGconn *baglanti);
```

`PQnotifies()` fonksiyonu, kurulu bağlantı üzerinden istemciye herhangi bir uyarı ulaşmışsa, kendisini çağırana `PGnotify` yapısını, aksi halde `NULL` değerini döndürecektir.

> *Dönecek olan `PGnotify` yapısı kullanıldıktan sonra `PQfreemem()` ile bellekten bırakılmalıdır.*

`PGnotify` yapısını inceleyecek olursak:

```
typedef struct pgNotify
{
    char *relname;            /* Uyarı ismi. */
    int   be_pid;             /* Sunucu tarafından yürütülen işlemin PID değeri. */
    char *extra;              /* Uyarı seçenekleri. (Bu özellik şu an için
                                 sadece boş bir katar döndürecektir.) */

    struct pgNotify *next;    /* libpq kütüphanesinin kendi iç mekanizmasınca
                                 kullanılıp, daima NULL değeri dönecektir. */
} PGnotify;
```

Asenkron uyarılmada, sunucu tarafından gönderilen mesaj `PQconsumeInput()` ile soket üzerinden bellekteki kütüphane yapılarına aktarılır. Ardından `PQnotify()` döngüsel olarak çağrılarak, fonksiyon `NULL` değeri dönene kadar ulaşmış uyarı mesajları elde edilir. Örnek olması açısından, asenkron uyarı mesajı alan basit bir program iskeleti:

```
/* Soket üzerinde veri olup olmadığını kontrol ediyoruz. */
while ((yoklamaSonucu = poll(...)))
{
    /* Soket yoklanırken hata oluştu! */
    if (yoklamaSonucu == -1) { ... }

    /* Soketin yoklanması esnasında hiçbir veri dönmeyip zaman aşımına uğrandı. */
    else if (yoklamaSonucu == 0) { ... }

    /* Gelen veri, soket üzerinden bellekteki kütüphane yapılarına atılıyor. */
    PQconsumeInput(baglanti);

    /* Gelen asenkron uyarılar sıra ile alınıyor. */
    while ((asenk_uyari = PQnotifies(conn)) != NULL)
    {
        ...
    }
}
```

> *Asenkron uyarılma konusunun daha iyi anlaşılması için, bölümün en sonunda yer alan `asenkron_uyarilma.c` örneğine bakabilirsiniz.*

Buna ek olarak, herhangi bir sorgu sonucunda bizi ilgilendiren bir uyarının olup olmadığını, sorgunun tamamlanmasından hemen sonra çalıştıracağımız `PQnotifies()` ile öğrenebiliriz. (Sorgu sonucunun alımı esnasında, `PQconsumeInput()` fonksiyonunun bizim yerimize çalıştırılacağını hatırlayınız.)

> *`LISTEN` ile dinlenen bir uyarının, dinlenmesini iptal etmek için `UNLISTEN` komutunu kullanabiliriz.*

10. Uyarı Mesajlarının İşlenmesi

Veritabanından dönen `PGRES_NONFATAL_ERROR` durumundaki sonuçlar ile ulaşan uyarı mesajlarının (sonuç yapısı içine gömülmek yerine) uyarı mesajı işlemcilerine aktarılacağından daha önceden bahsetmiştik. Bu sebeple, herhangi bir uyarı mesajına `PQresultError()` ya da `PQresultErrorField()` fonksiyonlarını kullanarak erişmek yerine, uyarıyı alıp işleyecek olan fonksiyonlar üzerinde değişiklik yapılarak ulaşılabilir. Bunu yaparaken de kullanılacak olan uyarı işlemci fonksiyonlarının öntanımlıları yerine kendi yazacağınız fonksiyonları devreye sokarak, uyarı mesajlarına erişebilirsiniz.

Uyarı mesajlarının işlenmesinde, libpq kütüphanesinin eski sürümleri ile uyumluluğun sağlanması açısından, alıcı ve işleyici olmak üzere iki fonksiyon kullanılmaktadır. Alıcı fonksiyon uyarı mesajını oluşturun sonuç yapısını aldıktan sonra, işleyici fonksiyonu çağırarak uyarı mesajının işlenmesini sağlar.

> *Uyarı mesajlarının işlenmesi her zaman bu sırada gerçekleşmek zorunda değildir. İlgili fonksiyonlar tanıtıldıktan sonra, bu durum üzerinde daha detaylı olarak durulacaktır.*

libpq kütüphanesince öntanımlı olarak sadece uyarı işleyici fonksiyon tanımlanmış olup, istemciye herhangi bir uyarı mesajı ulaşması durumunda da bunu sadece `stderr`'e yazacak şekilde programlanmıştır. Şimdi, uyarı mesajlarını işleyecek kendi yazdığımız fonksiyonları, bu öntanımlı fonksiyonun yerine nasıl ayarlayabileceğimize bakalım:

```
/*
 * İşaret edilen PQnoticeReceiver tipindeki fonksiyonun aslında void tipinde
 * (void tipinde bir argüman ile bağlantı sonucunu parametre olarak alan)
 * bir fonksiyon olacağı tanımlanıyor.
 */
typedef void (*PQnoticeReceiver) (void *argumanlar, const PGresult *sonuc);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn           *baglanti,
                    PQnoticeReceiver  aliciFonksiyon,
                    void             *argumanlar);
```

`PQsetNoticeReceiver()` fonksiyonu ile uyarıyı alacak olan fonksiyonu ve bu uyarıyı alacak olan fonksiyonun çağırılırken alacağı argümanları ayarlayabilirsiniz. (Bu ayarlanacak olan argümanlar, genelde programınızın kendi içindeki haberleşmesi ve birbirleri ile veri gönderiminde bulunmaları için kullanılır.)

Benzer şekilde, uyarıyı işleyecek olan fonksiyon için kullanılacak olan `PQsetNoticeProcessor()` fonksiyonu şu şekilde:

```
typedef void (*PQnoticeProcessor) (void *argumanlar, const char *uyari_mesaji);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn            *baglanti,
                     PQnoticeProcessor  isleyiciFonksiyon,
                     void              *argumanlar);
```

> *Uyarıyı işleyecek fonksiyona parametre olarak geçilen uyarı mesajı yeni satır karakteri içerecektir.*

`PQsetNoticeReceiver()` ve `PQsetNoticeProcessor()` fonksiyonları, parametre olarak belirtilen bağlantının geçersiz olması durumunda `NULL`, başarı durumunda ise bir önceki tanımlı `PQnoticeReceiver` ya da `PQnoticeProcessor` fonksiyonunu kendisini çağırana geri döndürecektir.

Uyarı fonksiyonların işlenmesinde iki adımın bulunduğunu ve isteğe bağlı olarak bu adımların sıralarının değişebileceğinden söz etmiştik. Şimdi bunun nasıl yapılabileceğini 2 tip üzerinde incelemeye çalışalım:

1. `PQsetNoticeProcessor()` ile uyarıyı işleyecek olan fonksiyon ayarlanır, ve

herhangi bir uyarı durumunda sadece bu fonksiyon çağrılır.

2. `PQsetNoticeReceiver()` ile uyarıyı alacak olan fonksiyon ayarlanır. Herhangi bir uyarı esnasında çalıştırılan fonksiyon parametre olarak aldığı `PGresult` sonuç yapısını kullanarak ilgili uyarı mesajını çıkarır (ve isterse bunu ayrıca tekrar uyarıyı işleyecek olan fonksiyona gönderebilir).

Burada dikkat etmeniz gereken şöyle bir nokta var: Aynı anda alıcı ve işleyici fonksiyonların her ikisinin de libpq tarafından - sırası önemli olmamak üzere - otomatik olarak çağrılmasını sağlayamazsınız. `PQsetNoticeProcessor()` ve `PQsetNoticeReceiver()` fonksiyonlarının alt alta kullanımı sonucunda, herhangi bir uyarı durumunda işleyici fonksiyon değil, sadece alıcı çalıştırılacaktır. Her ikisinin de çalıştırılmasını istediğiniz bir durumda, alıcı fonksiyon içinden işleyici fonksiyonu çağırmalısınız.

Uyarı mesajlarının nasıl işleneceği hakkında daha ayrıntılı bilgi için `uyari_mesajlarinin_islenmesi.c` örneğine bakabilirsiniz.

Sunucu veya istemci tarafından üretilen hata mesajlarını yakalamak dışında, kendi sorgularınızda da uyarı mesajları yaratarak sorgu akışının kontrolü için bu özellikten faydalanabilirsiniz. Örneğin, herhangi bir fonksiyon, yapılacak kontrol sonucunda ileride çalışacak bir *SQL* sorgusunun çalıştırılmasına gerek kalmadığını fark ederek, bunu bir uyarı mesajı ile istemciye bildirebilir. Bu sayede istemci o adıma gelindiğinde ne yapılacağı hakkında daha sağlıklı bir karar verebilir. (Kim bilir, belki sistem kaynakları açısından bize çok pahalıya mal olabilecek bir sorgudan kurtulmuş oluruz.)

11. Büyük Boyutlu Nesneler

Veritabanında saklanacak veriler her zaman bir kaç yüz karakteri geçmeyen düz metin alanlarından ibaret olmayabilir. Kimi zaman ayrıntılı bir bölge haritası ya da aralarındaki farkların tarandığı bir kaç saat uzunluğundaki ses kayıtları gibi büyük boyutlu dosyaları veritabanında saklamak gerekebilir. Bu gibi durumlarda PostgreSQL tarafından sunulan *Large Objects* tipindeki nesneleri kullanabiliriz. Bu noktada, *LO* tipindeki alanlar 2 Gb boyutlarına kadar ulaşabilen verilerin veritabanında uygun algoritmalar ile saklanmasına olanak sağlamaktadırlar.

PostgreSQL ile birlikte LO desteğini kullanabilmesi için, (`--enable-largefiles` parametresi ile) veritabanının büyük dosyalar desteği etkinleştirilmiş halde derlenmiş olması gerekmektedir.

LO tipindeki alanlar ile normal bir tablo satırı arasındaki artı ve eksileri incelemeden önce, normal tablo satırlarının veri saklama kapasiteleri ile ilgili şöyle ufak bir altyapı vermeyi uygun bulduk:

> PostgreSQL 7.1 sürümünden önceki sürümlerde öntanımlı olarak 8 KB gibi sabit boyutta sayfalar kullanıldığından, verinin birden çok sayfaya taşması durumunda çeşitli kısıtlamalar ortaya çıkıyordu. 7.1 sürümü ile geliştirilen `TOAST` mekanizması sayesinde, bu sınırlar ortadan kaldırıldı. Böylece, büyük boyuttaki satırlar sıkıştırılarak ya da parçalara ayrılarak fiziksel olarak birden fazla satırda - kendisini kullanıcıya tek bir satırmışcasına gösterecek şekilde - saklanabilir hale geldi. `TOAST` mekanizması sayesinde normal bir tablo satırında 1 GB (2^{32}-1 bayt) boyutlarına varabilen büyüklüklerde veri saklanmasına imkan tanınmış oldu.

11.A. Neden LO?

Tek bir satırda 1 Gb saklayabilecekken, *LO* tipindeki veri alanlarında 2 Gb boyutlarına

kadar veri saklayabiliyoruz. Bu durumuda bir kaç soru ile karşılaşıyoruz: *LO* kayıtları, bize veri boyutu sınırlarını 2 Gb'ye taşımanın dışında başka hangi özellikleri sunmaktadır?

- Herbir *LO* kaydı kendisine ait bir `OID` değeri ile veritabanında tutulduğundan, nesneye erişilmek istendiğinde rahatlıkla `OID` değeri referans gösterilerek çağırılabilir. Böylece büyük boyutlu nesneler tek bir noktada toplanmış olur.
- *LO* kayıtları üzerinde (*LO* fonksiyonlarını kullanarak) parça parça işlem yapmak, normal bir tablo satırının sütununda saklanan veri üzerinde işlem yapmaya kıyasla çok daha fazla kolaylık sağlayacaktır.
- *LO* alanları yerine, normal tablo sütunlarında saklanan verininin sunumu (ve özellikle geri ayıklanması (`unescape`) işlemi) sistem kaynakları açısından oldukça ciddi derecede yoğun işlemci ve bellek gücü gerektirmektedir.

> *Verinin ayıklanması problemi, 3.0 protokolü ile tanıtılan parametre kullanımı ile aşılabilmektedir.*

- *LO* kaydı şeklinde tutulan veri, B ağaçları ile indekslenerek (*B-Tree Indexing*) satırlar üzerinde parçalar halinde saklanır. Bu sayede verinin taranıp yazılması işlemlerinde hız artışları sağlanacaktır.
- *TOAST* mekanizması LZ (Lempel-Ziv) ailesinden sıkıştırma algoritmaları ile desteklendiğinden, söz konusu büyük boyutta veriler olduğu için, bunların veritabanına yazılıp, veritabanından okunması gibi durumlarda yoğun işlemci kullanımından dolayı bir performans kaybı oluşacak olması muhtemeldir. Fakat aynı şey *LO* kayıtları için geçerli değildir.

Neden bu kadar büyük boyutta bir dosyayı veritabanında tutmak isteyelim ki? Verinin sunucuda yer alan dosya sistemi üzerinde tutulduktan sonra, sadece dosya adlarının veritabanında saklanıp, istendiği zaman dosya adlarına göre dosyaların çağrılması bize daha büyük bir kolaylık sağlayamacak mıdır?

- Dosyaların veritabanında saklanıyor olması, tüm bilginin tek bir noktada toplanmasını sağlayacaktır. Böylece uzaktan veritabanına erişen istemci uygulamaları, herhangi bir dosya sistemi (ya da NFS) derdine düşmeden de, istediği bilgiye rahatlıkla ulaşabilecektir.
- Dosyalar, veritabanlarının en büyük özelliği olan bütünlük ilkesi ile daha kararlı bir yapı altında tutulacaktır. Verinin herhangi bir hata görmesi durumunda, ilgili veritabanı kayıtları sayesinde veri kurtarımı kolaylıkla gerçekleştirilebilecektir.
- *LO* kullamak yerine bazen sistem yöneticileri sadece dosya yollarını veritabanında saklayıp, dosyaların kendisini veritabanında saklama yoluna gidebiliyorlar. Böyle bir yöntem kullanıldığında ise, sistemdeki dosyalardan birinin silinmesi ya da isim değişikliği durumunda aynı dosyanın veritabanında geçtiği tüm satırların da güncellenmesi gerekebiliyor. Yani dosyaların sistem ile veritabanı arasındaki senkronizasyonu için de ayrı bir çaba gerekiyor. Bu şekilde düşünüldüğünde, dosyaların veritabanında saklanması sistem yöneticileri ve dosyaların bütünlüğü açısından büyük bir kolaylık sağlar hale geliyor.

LO kayıtlarının bahsi geçen artıları yanından, bazı eksileri de bulunmaktadır. Bunların birkaçını şu şekilde sıralamamız mümkün:

- KaydıA tutulan veriye ulaşılmak istendiğinde, doğrudan dosya sisteminden erişmek yerine, veritabanı aracılığı ile ulaşılacak olması belli bir hız kaybına sebebiyet verecektir.

> *Böyle bir durumunda LO kayıtları ile şöyle bir yol izlenebilir: Arşiv için dosyalar LO alanlarında saklanıp, üretim için gerektiğinde ayrıca dosya sistemine açılabilirler. Bu sayede arşiv veritabanı üzerinde daima bütün yapısını korurken, üretim için gerekli çalışma, ilgili verilerin dosya sistemi üzerindeki kopyaları ile sağlanabilir.*

- *LO* kayıtları veritabanında apayrı bir tablo üzerinde tutulduğundan ve saklanırken de

OID değerleri hariç hiçbir özellikleri saklandıkları tablo üzerinde tutulamadığı için, çoğu zaman veritabanı yöneticilerinin gözünden kaçıp, hiç kullanılmadıkları halde veritabanında kayıtlı tutuldukları oluyor.

pg_largeobject tablosundaki kayıtlardan veritabanındaki diğer hiçbir tablo tarafından kullanılmıyor olanlarını silmek içik PostgreSQL kaynak kodu ile gelen contrib/vacuumlo programını kullanabilirsiniz.

- Veritabanının yedeği alınırken *LO* kayıtları için ayrı işlem yapılması gerektiği eklenince, dikkatli olunmadığı taktirde *LO* kayıtlarının veritabanı üzerindeki kontrolleri zorlaşabiliyor.

LO kayıtlarının nasıl yedek alınacağı hakkında ayrıntılı bilgi için Sık Karşılaşılan Problemler kısmındaki ilgili başlığa bakabilirsiniz.

- *LO* kayıtları üzerinde yapılan değişiklikler - *LO* kayıtlarının birden fazla satıra dağıtılarak veritabanı üzerinde kaydedildiği düşünüldüğünde - *MVCC* (*Multiversion Concurrency Control*) mimarisi gereği pg_largeobjects tablosu üzerinde VACUUM işlemi gerçekleştirilmesini gerekli kılmaktadır. Eğer *LO* kayıtlarınız üzerinde sık sık değişiklik yapılıyorsa, böyle bir durumda VACUUM işlemi süresince sunucu üzerindeki yoğunluğun normal VACUUM işlemine oranla daha fazla olacağı aşikardır.

Listelenen karşılaştırma maddeleri dışında, veritabanında dosya saklanması ile ilgili olarak ayrıntılı bilgi için PHP, PostgreSQL Arayüzü için Sık Karşılaşılan Problemler kısmındaki *Veritabanında nasıl resim, müzik dosyası gibi ikili biçimde veri saklayabilirim?* başlığına bakabilirsiniz.

11.B. LO Nasıl?

Peki PostgreSQL nasıl bir mekanizma izleyerek *LO* alanlarına veritabanında bu boyutlarda yer açmayı başarabiliyor? PostgreSQL bunu yaparken veriyi LOBLKSIZE boyutlarında parçalara ayırdıktan sonra, oluşan parçaları aynı loid değeri altında pg_largeobject tablosunda saklıyor.

Öntanımlı olarak, LOBLKSIZE boyutu kaynak kodu içindeki include/storage/large_object.h dosyasında BLCKSZ/4 bayt (genelde 2 KB), BLCKSZ ise include/pg_config_manual.h dosyasında 8192 bayt olarak tanımlanmıştır. Bu değerlerin her ikisi de (çeşitli kısıtlamalar gözönünde bulundurularak) isteğe bağlı olarak değiştirilebilmektedir. (Kısıtlamalar için ilgili başlık dosyalarında yer alan açıklama alanlarına göz atılabilir.)

Basit bir örnek ile olayın daha da netleşeceğini düşünüyoruz:

```
# Dosyanın boyutunu bayt cinsinden öğreniyoruz.
$ find /tmp/ornek01.file -printf "%s\n"
5351549

# Dosyayı 2 KB'lik parçalara böldüğümüzde karşımıza çıkacak parça sayısı.
$ echo "scale=3; 5351549/2048" | bc
2613.061
# Görüldüğü üzere parça sayısı 2613.061 ≈ 2614 olacak. Bakalım gerçekten de öyle mi?

$ psql
...
=# SELECT lo_import('/tmp/ornek01.file');
 lo_import
-----------
     18450
(1 row)

=# SELECT count(pageno) FROM pg_largeobject WHERE loid = 18450;
 count
```

```
-------
  2614
(1 row)
```

Gerçekten de görüldüğü üzere parça sayısı 2614 olarak karşımıza çıktı. Benzer şekilde *LO* kaydı şeklindeki diğer veri dosyalarınız da parçalar halinde `pg_largeobject` tablosunda tutulacaktır. Tablo alanları üzerinde tanımlanan B ağaçları ve yardımcı *LO* fonksiyonları ile de bu dosyalar üzerinde işlem yapmak oldukça kolay hale gelmektedir.

LO kayıtlarının, veritabanında (kendilerine ait `loid` değerleri ile) `pg_largeobject` tablosunda tutulduklarından bahsetmiştik.

```
=> \d pg_largeobject;
Table "pg_catalog.pg_largeobject"
 Column |  Type   | Modifiers
--------+---------+-----------
 loid   | oid     | not null
 pageno | integer | not null
 data   | bytea   |
Indexes:
    "pg_largeobject_loid_pn_index" UNIQUE, btree (loid, pageno)
```

Yukarıdaki çıktıdan da anlaşabileceği üzere, *LO* kayıtları genelde diğer tablolardan `loid` değerleri sayesinde referans verilerek çağrılırlar. Ek olarak, `pg_largeobject` tablosunda asıl verinin saklandığı `data` sütununun da `bytea` tipinde olduğunu görüyoruz.

11.C. Kütüphane LO Fonksiyonları

LO kayıtları üzerinde işlem yapmak için libpq kütüphanesi tarafından çeşitli fonksiyonlar sağlanmaktadır. Bu bölümde bu fonksiyonlar üzerinde durmaya çalışacağız.

Kütüphanece sağlanan *LO* fonksiyonlarının kullanılabilmesi için `libpq-fe.h` yanında, `libpq/libpq-fs.h` başlık dosyasının da C kodunca dahil edilmesi gerekmektedir. Aksi taktirde *LO* fonksiyonlarını kullanmaya çalıştığınız zaman ilgili fonksiyonların olmadığına dair hata mesajı alırsınız.

> ***Kütüphane LO fonksiyonları transaction blokları içinde kullanılmalıdır.*** *Aksi halde sunucu hata mesajı düşecektir.*

```
Oid
lo_creat(PGconn *baglanti, int secenekler);
```

`lo_creat()` fonksiyonu, parametre olarak geçilen bağlantının kurulu olduğu veritabanı üzerinde yeni bir *LO* kaydı yaratır. Başarılı olması durumunda yaratılan nesnenin `OID`, aksi halde `InvalidOid` (0) değerini kendisini çağırana döndürecektir. Fonksiyona parametre olarak geçilebilecek seçenekler `INV_READ` ve `INV_WRITE` (ya da bunların `OR` mantıksal operatörü ile birleştirilmiş halleri) olmasına karşın, bu değerlerin fonksiyona herhangi bir etkisi olmayacaktır. Sadece eski programlar ile uyumluluğun sağlanması için tutulmaktadır. Fakat, herhangi bir hata mesajı ile karşılaşmamak için, yine de her bağlantıda bunların kullanılması gerekir.

```
Oid
lo_create(PGconn *baglanti, Oid loid);
```

`lo_create()` fonksiyonu, lo_creat() fonksiyonuna benzer şekilde veritabanında yeni bir *LO* kaydı oluşturulmasını sağlar; ek olarak bunu yaparken oluşturulacak *LO* nesnesinin alacağı `OID` değerinin belirtilmesine olanak sağlamaktadır. Belirtilen `loid` değerine sahip bir kayıt veritabanında hali hazırda mevcut ise ya da kaydın oluşturulması esnasında herhangi bir hata oluşması durumunda fonksiyon `InvalidOid` değerini döndürecektir.

Eğer loid parametresi olarak InvalidOid değeri girilecek olursa, fonksiyon, veritabanında bulduğu ilk boş loid değerini kullanacaktır.

lo_create() fonksiyonu 8.1 sürümü ve sonraki sürümlerce desteklenmektedir.

```
Oid
lo_import(PGconn *baglanti, const char *dosya_adi);
```

lo_import() fonksiyonu, sistemdeki bir dosyanın ilgili bağlantı üzerinden veritabanına *LO* kaydı şeklinde aktarılması için kullanılmaktadır. Fonksiyon başarılı olması durumunda eklenen nesnenin OID, aksi halde InvalidOid değerini döndürecektir.

İstemci taraflı lo_import() fonksiyonu ile veritabanına eklenecek dosyanın istemcinin bulunduğu sistem üzerinde yer alması gerekmektedir.

Aslında lo_import() fonksiyonu, bizim bir kaç fonksiyon çağrısında yapacağımız bir işlemi tek adıma indirger. Bunu yaparken, lo_creat() ile bir *LO* kaydı oluşturduktan sonra, lo_write() ile LO_BUFSIZE (8192 bayt) boyutunda tampon bellekler içinde veriyi *LO* alanına yazar.

```
int
lo_unlink(PGconn *baglanti, Oid lo_nesnesi);
```

lo_unlink() fonksiyonu, parametre olarak aldığı bağlantının kurulu olduğu veritabanı üzerindeki belirtilen OID değerine sahip *LO* kaydının silinmesini sağlar. Başarılı olması durumunda 1, aksi halde -1 değeri döndürecektir.

```
int
lo_export(PGconn     *baglanti,
          Oid         lo_nesnesi,
          const char *dosya_adi);
```

lo_export() fonksiyonu, parametre olarak girilen bağlantı üzerinden belirtilen OID değerine sahip *LO* kaydının belirtilen dosya adında istemci tarafındaki sisteme aktarılmasını sağlar. Başarılı olması durumunda 1, aksi halde -1 değeri döndürecektir.

```
int
lo_open(PGconn *baglanti, Oid lo_nesnesi, int secenekler);
```

lo_open() fonksiyonu, ilgili bağlantının gerçekleştiği veritabanındaki belirtilen OID değerine sahip *LO* kaydının (kayıt üzerinde ileride işlem yapılmak üzere) açılmasını sağlar. (Standart kütüphanedeki open() fonksiyonuna eşdeğerdir.) Fonksiyona secenekler kısmında parametre olarak geçebileceğiniz değerler ve açıklamaları şu şekilde:

INV_READ	*LO* kaydı üzerinde sadece okuma işlemi gerçekleştirebilirsiniz. Ayrıca, dönecek olan *LO* kaydı dosya işaretçisi, *LO* nesnesinin ilgili lo_open() komutunun çalıştırıldığı andaki kopyasını (*snapshot*'ını) yansıtacak olup diğer *transaction*'ların nesne üzerindeki değişikliklerini görüntülemeyecektir. (*Transaction*'ların SERIALIZABLE izolasyon seviyesinde olduğu gibi.)
INV_WRITE INV_WRITE|INV_READ	*LO* kaydı üzerinde okuma ve yazma işlemleri gerçekleştirebilecek olup, *LO* kaydı üzerinde diğer *transaction*'lar tarafından COMMIT edilecek değişiklikler yansıtılacaktır. (*Transaction*'ların READ COMMITTED izolasyon seviyesinde olduğu gibi.)

Fonksiyon, başarılı olması durumunda (pozitif) bir dosya tanımlaycısı (*file descriptor*) döndürecek olurken, hata durumunda -1 değeri döndürecektir.

lo_open() ile açılan bir LO kaydını kapatılmadan önce, bağlantının bitirilmemesi gerekir. Aksi halde, LO

işlemlerinin transaction blokları içinde yapıldığı düşünülürse, yapılan değişiklikler COMMIT edilmeyen bir transaction bloğu ile bir tutulup ROLLBACK ile geri alınacaktır.

```
int
lo_write(PGconn     *baglanti,
         int         dosya_tanimlayicisi,
         const char *katar,
         size_t      uzunluk);
```

`lo_write()` fonksiyonu, ilgili bağlantı üzerinden açılan *LO* kaydının üzerine parametre olarak aldığı katarın, belirtilen uzunluktaki kısmını yazar. (Standart kütüphanedeki `write()` fonksiyonu ile eşdeğerdir.) Fonksiyon, başarı ile gerçekleşen bir yazma işleminin sonunda toplam yazılan katar uzunluğunu, hata durumunda ise -1 değerini kendisini çağırana döndürecektir.

```
int
lo_read(PGconn *baglanti,
        int     dosya_tanimlayicisi,
        char   *katar,
        size_t  uzunluk);
```

`lo_read()` fonksiyonu, aldığı bağlantı işaretçisi ve dosya tanımlayıcısından girilen uzunlukta karakter katarını okuyup, belirtilen katarın içerisine okunan veriyi yazacaktır. (Standart kütüphanedeki `read()` fonksiyonu gibi.) Fonksiyon, başarılı olması durumunda toplam okunan bayt sayısını, hata durumunda ise -1 değerini döndürecektir.

Parametre olarak geçilen katarın, belirtilen uzunluktaki okunan veriyi içinde tutabilecek kadar bellek alanına sahip olması, programcının sorumluluğu altında olup, `lo_read()` bunu kontrol etmeyecektir.

```
int
lo_tell(PGconn *baglanti,
        int     dosya_tanimlayicisi);
```

`lo_tell()` fonksiyonu, parametre olarak aldığı bağlantının gerçekleştiği veritabanında açılan *LO* kaydının dosya tanımlayıcısının (dosya başından itibaren olan) konumunu kendisini çağırana döndürür. Bağlantının geçersiz olduğu ya da olası diğer bir hata durumunda fonksiyon -1 değeri döndürecektir.

```
int
lo_lseek(PGconn *baglanti,
         int     dosya_tanimlayicisi,
         int     ilerlenecek_uzunluk,
         int     tarama_konumu);
```

`lo_lseek()` fonksiyonu, parametre olarak aldığı dosya tanımlayıcısının o anki konumunu belirtilen uzunlukta ilerletir. (Standart kütüphanedeki `lseek()` fonksiyonu gibi.) Fonksiyon, herhangi bir hata durumunda -1 değeri döndürecektir.

Tarama konumunda kullanabileceğiniz değişkenler şu şekilde:

```
SEEK_SET  /* İşaretçiyi, dosyanın en başından ilerlet.          */
SEEK_CUR  /*            o anki konumundan ilerlet. (Öntanımlı.) */
SEEK_END  /*            dosyanın en sonundan ilerlet.          */
```

```
int
lo_close(PGconn *baglanti, int dosya_tanimlayicisi);
```

`lo_close()` fonksiyonu, belirtilen bağlantı üzerinden `lo_open()` ile açılmış *LO* kaydının

kapatılmasını sağlar. Başarılı olması durumunda 0, aksi halde -1 değerini döndürecektir.

> *Oturum esnasında açılıp daha sonra kapatılmayan LO kayıtları, transaction sonlandığında otomatik olarak - yapılan değişiklikler ROLLBACK ile geri alınarak - kapatılacaktır.*

11.D. Sunucu Taraflı LO Fonksiyonları

Buraya kadar *LO* kayıtlarının kullanımı için tanıtılan fonksiyonlar hep istemci taraflıydı. Bu bölümde ise benzer fonksiyonların sunucu tarafından *SQL* komutları şeklinde kullanılabilecek olanları üzerinde duracağız.

> *Sunucu taraflı LO fonksiyonları sadece veritabanındaki üst seviyeli kullanıcılar (superuser) tarafından kullanılabilecek olurken, istemci taraflı LO fonksiyonları libpq kütüphanesine erişimi olan herkes tarafından kullanılabilir.*

Veritabanına *LO* tipinde veri olarak aktarmak istediğiniz bir dosya her zaman istemci tarafında olmak zorunda değildir. Bazı durumlarda sunucunun bulunduğu sistemdeki dosyaların da veritabanına *LO* fonksiyonları kullanarak aktarılması gerektiği durumla olabilir. Yahut veritabanındaki herhangi bir sorgu sonucu dönen veri ile yeni bir *LO* kaydı oluşturulması ya da varolan bir *LO* kaydından alınan verinin sorgu içinde kullanılmasının gerektiği anlar olabilir. İşte böyle bir durumda sunucu taraflı *LO* fonksiyonlarını rahatlıkla kullanabilirsiniz.

Bahsi geçen bu fonksiyonlar, alacakları parametreler ve dönecek olan değerleri ile şu şekilde listelenebilir:

```
 Fonksiyon | Sonuç Tipi |     Parametre Tipleri
-----------+------------+---------------------------
 lo_close  | integer    | integer
 lo_creat  | oid        | integer
 lo_create | oid        | oid
 lo_export | integer    | oid, text
 lo_import | oid        | text
 lo_lseek  | integer    | integer, integer, integer
 lo_open   | integer    | oid, integer
 lo_tell   | integer    | integer
 lo_unlink | integer    | oid
 loread    | bytea      | integer, integer
 lowrite   | integer    | integer, bytea
```

Sunucu taraflı *LO* fonksiyonlarının kullanımı hakkında bir fikir edinebilmeniz için basit birkaç örnek vermeyi uygun bulduk:

```
-- lo tablomuza göz atıyoruz.
=# SELECT DISTINCT loid FROM pg_largeobject;
 loid
-------
 18083
 18084
(2 rows)

-- Sunucu taraflı fonksiyonları kullanabilmek için
-- veritabanına superuser olarak bağlandığımızdan emin
-- olduktan sonra aşağıdaki komutları çalıştırmalıyız.

-- İlk LO kaydını silelim
=# SELECT lo_unlink(18083);
 lo_unlink
-----------
         1
(1 row)
-- Şimdi sadece 18084 OID değerli LO kaydı kaldı.
```

```
-- Yeni bir LO oluşturuyoruz.
=# SELECT lo_creat(-1);
 lo_creat
----------
    18099
(1 row)
-- lo_creat() fonksiyonu bize yeni eklenen LO kaydının OID değerini döndürdü.
-- Yazılacak SQL prosedürlerinde bu değerler değişkenler içinde tutularak
-- lo_open(), loread(), lowrite() fonksiyonlarına parametre olarak geçilebilir.

-- LO listesinin son durumu:
=# SELECT loid FROM pg_largeobject;
 loid
-------
 18084
 18099
(2 rows)

-- Veritabanında ilgili LO kaydını sisteme aktarıyoruz.
=# SELECT lo_export(kayit.loid, '/backup/lo/' || kayit.loid)
>     FROM pg_largeobject AS kayit
>     WHERE kayit.loid = 18084;
 lo_export
-----------
         1
(1 row)
-- Bu işlemi pg_largeobject tablosu üzerinde yapmak zorunda değildik.
-- Benzer olarak loid alanlarına referans veren başka tablolar için
-- de aynı işlem tekrarlanabilirdi.

-- Sistem kataloglarına (dolayısıyla pg_largeobject kataloğuna da) referans
-- verilemeyeceği göz önünde tutulacak olursa, girilen tüm LO kayıtlarının
-- ayrı tek bir tablo üzerinde tutulup, diğer tabloların bu tablodaki
-- satırlara referans vermesi bu kısıtlamaya güzel bir çözüm olabilir.

-- LO kaydı saklayacak bir tablomuz olsun.
=# \d lo_bulunur
        Table "public.lo_bulunur"
  Column  |       Type        | Modifiers
----------+-------------------+-----------
 aciklama | character varying |
 loref    | oid               |

-- Yeni bir LO kaydı çekiyoruz.
=# INSERT INTO lo_bulunur VALUES (
>     'Evrak Yedekleri',
>     lo_import('/tmp/18084.lo')
> );
INSERT 18111 1

-- LO kayıtlarının son durumu:
=# SELECT loid FROM pg_largeobject;
 loid
-------
 18084
 18099
 18110
(3 rows)
```

12. Diğer Fonksiyonlar

Bu bölümde katagorize edemediğimiz libpq kütüphane fonksiyonlarına yer vermeye çalıştık.

```
void
PQtrace(PGconn *baglanti, FILE *dosya_isaretcisi);
```

`PQtrace()` fonksiyonu, sunucu ile istemci arasındaki iletişim çıktısını parametre olarak aldığı dosya işaretçisine yazmaya çalışacaktır.

> *Dosyanın açılıp açılmadığı ya da kullanıcının dosya üzerinde yazma haklarına sahip olup olmadığından programcı sorumludur. Kütüphane fonksiyonları direk `fprintf()` fonksiyonu ile dosya üzerine yazmaya çalışacaktır. Bu yüzden dosya işaretçisi kontrolünün doğru yapılmadığı bir durumda program beklemeye sebebiyet verebilir.*

```
void
PQuntrace(PGconn *baglanti);
```

`PQuntrace()` fonksiyonu, `PQtrace()` ile başlatılan bir bağlantı izlenimini sonlandırmak için kullanılır.

```
PGresult *
PQmakeEmptyPGresult(PGconn         *baglanti,
                    ExecStatusType  durum);
```

`PQmakeEmptyPGresult()` fonksiyonu, boş bir `PGresult` nesnesi oluşturmak için kullanılmaktadır. Uygulamalar bu fonksiyonu daha çok istedikleri durumlarda (`ExecStatusType`) sorgu sonuçları (`PGresult`) oluşturmak için kullanırlar. Bazen bağlantının sorunsuz gözüktüğü, fakat sorgu durumunun hatalı döndüğü anlar olur. Böyle durumlarda boş bir `PGresult` nesnesi oluşturursanız, içine geçerli bağlantı ile ilgili hata mesajı yerleştirilir.

`PQmakeEmptyPGresult()` fonksiyonu, parametre olarak aldığı bağlantının `NULL` olmaması ve belirtilen durumun `PGRES_BAD_RESPONSE`, `PGRES_NONFATAL_ERROR` ya da `PGRES_FATAL_ERROR` olması durumunda mevcut olan hata mesajını döndüreceği `PGresult` nesnesi içine yazmaya çalışacaktır.

> *`PQmakeEmptyPGresult()` sonucu dönen `PGresult` yapısı da diğer sorgu sonuçlarından farklı olmadığı için, işi bittiği zaman `PQclear()` ile temizlenmelidir.*

```
size_t
PQescapeString(char       *yeni_katar,
               const char *taranacak_SQL_komutu,
               size_t      uzunluk);
```

`PQescapeString()` fonksiyonu, parametre olarak aldığı taranacak *SQL* komut katarının belirtilen uzunluktaki kısmı içinde `'` karakterini `''` ile, `\` karakterini `\\` ile değiştirdikten sonra bunu belirtilen yeni katar içine atarak, yeni katarın uzunluğunu kendisini çağırana döndürecektir.

> *Herhangi bir PostgreSQL sorgusunda `'` işaretini, karakterin başına bir ters bölme işareti koyarak da (`\'` şeklinde) ayıklayabilecek olup, bu sadece PostgreSQL'e özgü (SQL standartlarında yer almayan) bir durum olduğu için tercih edilmemektedir. Bu yüzden `\'` yerine `''` kullanılmaktadır. Programcıların yazacakları ifadelerde, bu durumu göz önünde bulundurmaları tavsiye olunur.*
>
> *`PQescapeString()` ve `PQescapeBytea()` fonksiyonlarının tam olarak görevi, SQL sorgu komutunda tırnak içinde yerleştirilecek verinin SQL Injection saldırılarına neden olmayacak şekilde ayıklanmasını sağlamaktır.*

Bunların dışında dikkat edilmesi gereken noktaları şu şekilde listeleyebiliriz:

- Taranacak *SQL* komutu içinde, belirtilen uzunluğa ulaşmadan daha önce bir `\0` (`EOF`) karakterine rastlanırsa, tarama işlemi o karakterde sonlanacaktır.

- Kontrolden geçirilmiş *SQL* komutunun kaydedileceği yeni katar, belirtilen komut uzunluğunun iki katından bir fazla uzunlukta olmalıdır ve bunun için gerekli bellek alanının ayrılması programcının sorumluluğundadır; `PQescapeString()` bunu kendi başına kontrol etmeyecektir.
- Kontrolden geçirilmiş *SQL* sorgusunu koyduğumuz yeni katarın sonuna bir `\0` karakteri eklenecektir. Fakat fonksiyonun döndürdüğü katar uzunluğunda sondaki `\0` karakteri sayılmayacaktır.

```
unsigned char *
PQescapeBytea(const unsigned char *taranacak_ikili_katar,
              size_t               girdinin_uzunlugu,
              size_t              *ciktinin_uzunlugu);
```

`PQescapeBytea()` fonksiyonu, belirtilen (`bytea` veri tipinde) ikili katar üzerindeki ayıklanması gereken ikili karakterleri herhangi bir *SQL Injection* saldırısına yol açmayacak şekilde eşleri ile değiştirerek, çıktı uzunluğunu parametre olarak aldığı tamsayı işaretçisine atadıktan sonra, düzeltilmiş ikili katarın bellekte tuttuğu yerin işaretçisini kendisini çağırana döndürür. Fonksiyon, döndürülecek katar için yeterli bellek alanı ayrılamadığı zaman `NULL` değeri döndürecektir.

`PQescapeBytea()` fonksiyonu kullanımında dikkat edilmesi gereken noktaları şu şekilde listeleyebiliriz:

- İkili bir veri içinde bir çok `NULL` (`\0`) karakteri olabileceği için, belirtilen katar taranırken herhangi bir `\0` karakteri ayıklama işlemini kesmek için dikkate alınmayacaktır. Fonksiyon için tarama sınırı parametre olarak girilen girdi uzunluğundan oluşur.
- Kontrolden geçirilmiş ikili katar, fonksiyon tarafından ayrılan bellekte tutulacağından, katar ile işimiz bittiği zaman `PQfreemem()` ile bellekten bırakılmalıdır.
- Kontrolden geçirilmiş ikili katarın sonunda bir `\0` karakteri bulunacaktır ve çıktı uzunluğu hesaplanırken bu `\0` karakteri de toplama dahil edilecektir.

Yapılacak ayıklama işleminde çevrimler şu şekilde olacaktır.

Girdi	**Çıktı**
`\0`	`\\000`
`\'`	`\'`
`\\`	`\\\\`
0x20 - 0x7E aralığındaki karakterler	`\\vyz` şeklinde sekizlik eşleri

```
unsigned char *
PQunescapeBytea(const unsigned char *geri_ayiklanacak_katar,
                size_t              *geri_ayiklanmis_uzunluk);
```

`PQunescapeBytea()` fonksiyonu, `PQescapeBytea()` fonksiyonu ile ayıklanmış ya da `PQgetvalue()` ile bir tablo satırının `bytea` tipindeki bir sütunundan dönmüş (daha önceden ayıklanmış) ikili katarı eski ayıklanmamış haline geri çevirir. `PQunescapeBytea()` fonksiyonu, geri ayıkladığı katarı kendi bellekte alıkoyduğu alana taşıyıp, oluşan bu yeni katarın uzunluğunu parametre olarak aldığı uzunluk işaretçisi gösterecek şekilde ayarladıktan sonra, kendisini çağırana geri ayıklanmış katarın işaretçisini döndürecektir.

> *`PQunescapeBytea()` sonucu dönen katar ile işimiz bittiğinde, bellekte alıkoyulan alan `PQfreemem()` ile serbest bırakılmalıdır.*

`PQunescapeBytea()` fonksiyonu tarafından yapılacak karakter yer değiştirmelerini şu şekilde listeleyebiliriz:

Girdi	**Çıktı**

\'	'
\\	\
\VYZ	VYZ sekizlik değerindeki karakter.
\x	x (x yukarıdaki kriterlere uymayan herhangi bir karakter.)

```
void
PQfreemem(void *isaretci);
```

PQfreemem() fonksiyonu, PQescapeBytea(), PQunescapeBytea() ve PQnotifies() fonksiyonları sonucunda dönen katarların bellekte alıkoyduğu alanı serbest bırakmak için kullanılır.

13. SSL Desteği

Bağlantı kurulumunda *SSL* kullanan bir uygulamada sunucu tarafı, ilgili sertifikaları talep ettiği zaman, libpq kütüphanesi tarafından sağlanan *API* bunları kullanıcının ev dizini altında .postgresql klasörü içinde aşağıdaki dosya adları ile arayacaktır:

~/.postgresql/postgresql.crt	Sunucuya gönderilecek olan sertifika.
~/.postgresql/postgresql.key	Kişisel anahtarların bulunduğu dosya. (Sahibi dışındakiler için erişilemez olmasına dikkat ediniz.)

Eğer ~/.postgresql/ altında root.crt dosyası da mevcutsa, libpq sertifikaları bu listeden arayacaktır.

> *Sunucu tarafında herhangi bir sertifika mevcut değilse (Nasıl kurulacağını kitabın ilgili bölümünde bulabilirsiniz.) kimlik kontrolü başarısız olarak sonuçlanacaktır.*
>
> *Microsoft Windows platformunda ~/.postgresql/ dizini yerine genellikle geçerli kullanıcının ev dizini altındaki postgresql klasörü kullanılır.*

14. Şifre Dosyası Kullanımı

Veritabanına bağlanacak olan kullanıcının ev dizini altında yer alan .pgpass dosyası tarafından tutulan kimlik bilgileri sayesinde, sunucu herhangi bir kimlik denetimi talep ettiği zaman, kullandığınız bağlantı fonksiyonunda (PQconnectdb(), PQconnectStart(), vs.) belirtmediğiniz parametrelerin (şifre, ya da şifre+kullanıcı adı, vs.) bu dosyadan alınmasını sağlayabilirsiniz.

> *POSIX uyumlu sistemlerde bu dosya ~/.pgpass şeklinde yer alırken, Microsoft Windows sistemlerde kullanıcının ev dizini altındaki postgresql\pgpass.conf dosyası şeklinde bulunmaktadır. Ayrıca, Çevre Değişkenleri bölümünde tanıtılan PGPASSFILE değişkeni ile .pgpass dosyasının sistemdeki yerini belirleyebilirsiniz.*

Şifre dosyasının herbir satırı aşağıdaki biçimde bulunmalıdır:

```
host:port:dbname:user:password
```

host, port, dbname ve user alanında kullanmak istediğiniz değerlerin dışında, öntanımlı değerlerin kullanılmasını sağlamak için * girebilirsiniz. Bu alanların herhangi birinde üst üste iki nokta (:) ya da ters bölme işareti (\) kullanmak isterseniz, o karakterin önüne bir ters bölme işareti koyarak bunu belirtmelisiniz. Şöyle ki:

```
# Örnek .pgpass girdisi:
# (Kullanılan şifrenin düz metin hali: `xİ:Ğ1\c9J')
```

```
$ cat .pgpass
/tmp:*:ornekdb:knt:xİ\:Ğ1\\c9J
```

Herhangi bir bağlantıda sunucu tarafından kimlik denetimi talep edildiği zaman, ilk önce girdiğiniz bağlantı seçenekleri kullanıldıktan sonra, (gerek duyulduğu yerde) `.pgpass` dosyası ilk satırından itibaren taranarak, bağlantı seçenekleri ile örtüşen ilk satır kimlik denetimi için kullanılacaktır. (Bu nedenle `.pgpass` dosyasında kullanacağınız alanlarda ne kadar az `*` değeri kullanırsanız istenen satırın belirlenmesi açısından o kadar iyi olur.)

> *`.pgpass` dosyası şifrelerinizi tutan düz bir metin dosyasından oluştuğu için dosyanın okuma hakları dikkat edilerek ayarlanmalıdır. `.pgpass` dosyası 0600'ün üstünde bir dosya hakkında sahip olduğunda, bağlantılarda hiçbir şekilde dikkate alınmayacaktır. (Microsoft Windows sistemlerde bu kontrol mekanizması henüz çalışmamaktadır.)*

15. Çevre Değişkenleri

Bazı çevre değişkenleri sayesinde bağlantı fonksiyonlarında parametre olarak kullanılmayan seçeneklerin öntanımlı değerlerini rahatlıkla belirleyebiliriz. Aşağıdaki listede bu çevre değişkenlerini ve açıklamalarını bulabilirsiniz.

`PGHOST`	
`PGHOSTADDR`	
`PGPORT`	
`PGDATABASE`	
`PGUSER`	
`PGOPTIONS`	
`PGSSLMODE`	
`PGCONNECT_TIMEOUT`	
`PGPASSFILE`	libpq tarafından kullanılacak `.pgpass` şifre dosyasının sistem üzerindeki yerini belirtir.
`PGSERVICE`	`pg_service.conf` dosyasında aranacak servis adını belirtir.
`PGREALM`	Kerberos bölgesi tanımlamak için kullanılır. Öntanımlı olarak yerel bölge kullanılacaktır. `PGREALM` değişkeni kullanıldığı taktirde, libpq bağlantı fonksiyonları, yerel sistem bölgesi yerine belirtilen Kerberos bölgesini kullanacaklardır. Sunucu kimlik kontrolü için Kerberos kullandığı zaman bu özellik devreye sokulur. (Örnek: `CMF.NRL.NAVY.MIL`, `ATHENA.MIT.EDU` gibi.)
`PGKRBSRVNAME`	Kerberos 5 kullanılarak yapılacak kimlik denetiminde kullanılacak olan Kerberos servisinin adını belirtir.
`PGDATESTYLE`[15]	Öntanımlı tarih/zaman gösterim biçimi. (`SET datestyle TO 'ISO, MDY'`)
`PGTZ`[15]	Öntanımlı yerel saat (timezone) değeri. (`SET timezone TO 'Europe/Istanbul'`)
`PGCLIENTENCODING`[15]	Öntanımlı istemci karakter kodlaması. (`SET client_encoding TO 'UNICODE'`)
`PGGEQO`[15]	Öntanımlı genetik optimizasyon durumu. (`SET geqo TO 'on'`)
`PGSYSCONFDIR`	`pg_service.conf` dosyasının bulunduğu dizini belirtir.
`PGLOCALEDIR`	Üretilen mesajların yerelleştirilmesinde kullanılacak `locale` dosyalarının bulunduğu dizini belirtir.

16. Programların Derlenmesi

libpq arayüzünü kullanan programların derlenmesi ve kütüphanelerin bağlanması

15 Değişkenler herbir oturum için ayrıca ayarlanabilir.

esnasında, kütüphane paylaşımlı nesnelerinin ve başlık dosyalarının bulunduğu dizinin kullanacağınız derleyiciye gösterilmesi ve ilgili başlık dosyalarının da programınıza eklenmesi gerekmektedir. Bu gerekli adımların yapılmamaları sonucunda doğacak olası hata çıktıları hakkında bahsetmeye çalışacağız bu bölümde.

libpq arayüzünü kullanan bir uygulamada, çağrılacak olan fonksiyonların ve kullanılan *API* tip ve değişkenlerinin tanımlandığı başlık dosyası olan `libpq-fe.h` dosyasını programınızın en başında yer alan başlık dosyalarının bulunduğu bölüme `include` ifadesi ile eklemelisiniz.

libpq kütüphane fonksiyonlarını kullanmanıza rağmen, başlık dosyasını unutup programı derlemeye çalıştığınızda, karşınıza şuna benzer bir hata mesajı çıkacaktır:

```
example.c: In function `main':
...
example.c:53: `PGconn' undeclared (first use in this function)
example.c:54: `PGresult' undeclared (first use in this function)
...
example.c:71: `PGRES_NONFATAL_ERROR' undeclared (first use in this function)
...
```

libpq başlık dosyaları genelde sisteminizde çoğu derleyicinin öntanımlı olarak baktığı dizinler altına yerleştirildikleri halde, bazı özel kurulumlarda bu başlık dosyalarının derleyiciler tarafından bulunamadığı da olur. Böyle durumlarda libpq başlık dosyalarının bulunduğu dizini kullandığınız derleyiciye göstermeniz gerekmektedir.

```
# Başlık dosyalarının bulunduğu dizini derleyiciye parametre olarak geçiyoruz.
$ cc -I/usr/local/pgsql/include -c ornek-uygulama.c

# Eğer Makefile dosyaları kullanıyorsak, başlık dosyalarının bulunduğu dizini
# C önişlemcisinin sistem çevre değişkeni olan CPPFLAGS değişkenine eklemeliyiz.
$ CPPFLAGS += -I/usr/local/pgsql/include
```

Ek olarak, `libpq-fe.h` başlık dosyasının derleyici tarafından bulunamadığı bir durumda alacağınız olası hata mesajı ise aşağıdakine benzer olacaktır:

```
ornek-uygulama.c:41:22: libpq-fe.h: No such file or directory
```

Bahsi geçen başlık dosyalarının PostgreSQL kurulumunda nereye kopyalandığını `pg_config` komutuna `--includedir` parametresi vererek öğrenebilirsiniz.

```
# pg_config komutundan libpq başlık dosyalarının nerede olduklarını öğreniyoruz.
$ pg_config --includedir
/usr/local/pgsql/include
```

Son olarak, derlenen programın paylaşımlı kütüphane dosyaları ile bağlanması gerekmektedir. Bunun için derleyicinize `-lpq` parametresi ile libpq kütüphanesinin de bağlanacağını bildirmeniz gerekmektedir.

```
# Son olarak libpq kütüphanesini oluşturduğumuz ornek-uygulama.o dosyasına bağlıyoruz.
$ cc -lpq -o ornek_uygulama ornek-uygulama.o
```

libpq kütüphanesinin programa bağlanacağı bildiriminin (`-lpq` parametresinin) unutulması durumunda derleyici şuna benzer bir hata mesajı verecektir:

```
...
testlibpq.o(.text+0x4a): undefined reference to `PQconnectStart'
testlibpq.o(.text+0x52): undefined reference to `PQconnectPoll'
testlibpq.o(.text+0xf3): undefined reference to `PQgetResult'
...
```

Paylaşımlı libpq kütüphanelerin bulunamaması gibi bir durumda aşağıdaki çıktıya benzer bir hata mesajı ile karşılaşılacaktır:

```
/usr/bin/ld: cannot find -lpq
```

Böyle bir durumda derleyiciye libpq paylaşımlı kütüphane dosyalarının yerinin elle

girilmesi gerekmektedir.

```
# pg_config komutundan kütüphane dosyalarının nerede olduklarını öğreniyoruz.
$ pg_config --libdir
/usr/local/pgsql/lib

$ cc -L/usr/local/pgsql/lib -lpq -o ornek-uygulama ornek-uygulama.o
```

17. Yiv Korunaklı Programlama

Aynı anda birden fazla yivin çalıştığı programlarda, kaynakların yivler arası paylaşımı bir takım sorunları da beraberinde getirmektedir. Ortak kullanılan bellek alanlarının yivler arası paylaşımı ana program tarafından doğru bir şekilde düzenlenmediği sürece yarış koşulları (*race condition*), yiv kilitlenmeleri (*deadlock*), öncelik terslikleri (*priority inversion*) gibi bir çok sorunun ortaya çıkması oldukça olasıdır. Bu bölümde, libpq kütüphanesini kullanarak geliştirdiğiniz yazılımlarda bu gibi potansiyel sorunlara karşı ne gibi önlemlerin alınabileceği üzerinde durmaya çalışacağız.

Her şeyden önce, yivler arası libpq çağrıları (ve bunlar sonucu oluşan yapıların) kullanımında kütüphane dosyalarının yiv korunaklı seçeneği aktif hale getirilerek derlenmiş olması gerekmektedir. Bunun için PostgreSQL derlenirken `configure` betiğine `--enaable-thread-safety` parametresinin verilmiş olması gerekmektedir. Bu sayede, libpq yivler arası paylaşımda sorun yaratacak çağrıları kilitler ve diğer uygun yöntemler ile yiv korunaklı bir şekilde sunacaktır.

libpq tarafından yiv korunaklı hale getirilen önemli çağrılardan bazıları, kullanılan sistem tarafından sağlanan `strerror()`, `getpwuid()` ve `gethostbyname()` fonksiyonlarıdır. libpq `configure` esnasında yaptığı kontroller ile eğer mevcutsa bunların sistem tarafından sağlanan yiv korunaklı alternatiflerini (`strerror_r()` gibi) kullanacaktır. Aksi halde kendi oluşturduğu kilitler ile bunların yiv korunaklı türevlerini oluşturacaktır.

> *libpq tarafından kullanılacak sistem çağrılarınızın yiv korunaklı olup olmadığını kontrol etmek için PostgreSQL ile birlikte gelen `src/tools/thread` programını kullanabilirsiniz.*

Bunun dışında libpq tarafından sağlanan bazı nesnelerin yiv korunaklı olarak nasıl kullanılabileceklerini şu şekilde listeleyebiliriz:

`PGconn`	Farklı yivler tarafından aynı bağlantıyı gösteren `PGconn` nesnelerinin kullanımı istemci tarafından tutarsızlık yaratacağından, programcının bunu kendi belirleyeceği metodlar ile (örneğin `mutex` ile kilitleyerek) yiv korunaklı halde yivler arası paylaştırması gerektirmektedir. Aynı anda iki yiv tarafından sorgu yapılmasına ihtiyaç duyulan durumlarda, veritabanına birden fazla bağlantı kurularak - ya da bağlantı havuzu oluşturularak – sorgular farklı bağlantılar üzerinden gönderilmedilir.
`PGresult`	Sorgu sonucu dönecek olan `PGresult` yapıları istemci belleği üzerinde sadece okumaya müsade edecek şekilde (*read-only*) saklandığından, bu yapıların yivler arası kullanımı herhangi bir sorun teşkil etmemektedir.

Yukarıdakilere ek olarak, Kerberos kimlik denetimi kullanılarak gerçekleştirilmeye çalışılan veritabanı bağlantılarında Kerberos çağrıları yiv korunaklı olmadığından, bu çağrılar programcı tarafından ayrıca kilitlenerek gerçekleştirilmelidir. Kerberos çağrıları esnasında programınızda kullandığınız kilitleme mekanizmalarını libpq'nun Kerberos kimlik kontrolü esnasında kullandığı kilitlere de uygulamak için `PQregisterThreadLock()` fonksiyonunu kullanabilirsiniz. Fonksiyonun nasıl kullanıldığına dair fikir edinmeniz açısında ilgili dosyalardan şu satırları verebiliriz:

```
/* pgthreadlock_t tipinde bir fonksiyon işaretçisi oluşturuluyor. */
typedef void (*pgthreadlock_t) (int acquire);

/*
```

```
 * Kerberos kimlik denetimi ensasında kullanılacak olan
 * pg_g_threadlock fonksiyonu işaretçisi tanımlanıyor.
 */
extern pgthreadlock_t pg_g_threadlock;

/*
 * PQregisterThreadLock() fonksiyonuna işaretçi olarak geçeceğiniz fonksiyon aldığı
 * boolean parametre değerine göre yivler genelinde oluşturulan kiliti açıp kapayacak.
 */
#define pglock_thread()     pg_g_threadlock(true)
#define pgunlock_thread()   pg_g_threadlock(false)

/*
 * Bahsi geçen PQregisterThreadLock() fonksiyonu, parametre olarak aldığı
 * kilit fonksiyonunu kütüphane tarafından kullanılacak olan pg_g_threadlock
 * işaretçisine atayacak.
 */
pgthreadlock_t
PQregisterThreadLock(pgthreadlock_t newhandler)
{
    pgthreadlock_t prev = pg_g_threadlock;

    if (newhandler)
        pg_g_threadlock = newhandler;
    else
        pg_g_threadlock = default_threadlock;

    return prev;
}
```

Yukarıdaki gibi, `PQregisterThreadLock()` fonksiyonunu kullanarak programınızda kullandığınız yiv kilitleme mekanizmasının libpq kütüphanesinin Kerberos kimlik denetimi esnasında da kullanmasını sağlayabilirsiniz.

> *`PQregisterThreadLock()` ile libpq kütüphanesine kullanım için bir kilit fonksiyonu belirtmediğiniz taktirde dahi kütüphane bunu kendi içindeki öntanımlı `default_threadlock()` fonksiyonu ile gerçekleştirecektir.*

18. Fonksiyon Tablosu

PostgreSQL C programlama arayüzü içinde incelenmiş fonksiyonların kısa birer açıklama satırı ile birlikte tam bir listesini aşağıdaki tabloda bulabilirsiniz.

Fonksiyon	**Açıklama**

`PQconnectdb`
`PQsetdbLogin`
`PQsetdb`
	Veritabanına senkron bağlantı fonksiyonları.

`PQconnectStart` `PQconnectPoll`	Asenkron bağlantı fonksiyonları.
`PQstatus`	Bağlantı durumu.
`PQprotocolVersion`	Bağlantıda kullanılan protokol versiyonu.
`PQconndefaults`	Bağlantı özellikleri.

`PQhost`
`PQuser`
`PQpass`
`PQdb`
`PQport`
`PQoptions`
	Bağlantıda kullanılan seçenekler.

`PQgetssl`	Bağlantıda (*SSL* kullanılmışsa) kullanılan *SSL* yapısı.
`PQsocket`	Bağlantı soketi.

PQclientEncoding	Bağlantıda kullanılan istemci karakter kodlaması.
PQtransactionStatus	Transaction durumu.
PQisBusy	Sunucunun o an bir sorgu üzerinde çalışıp çalışmadığının durumu.
PQisnonblocking	Kullanılan bağlantı tipinin programı bloke edip etmeyeceği.
PQerrorMessage	Bağlantıda gerçekleşen en son hata mesajı.
PQparameterStatus	Sunucu seçenekleri.
PQserverVersion	Bağlantının kurulu olduğu sunucunun sürüm numarası.
PQbackendPID	Bağlantıyı sağlayan sunucu işleminin *PID* değeri.
PQsetClientEncoding	İstemci karakter kodlamasının ayarlanması.
PQsetErrorVerbosity	Dönen hata mesajı duyarlılığının ayarlanması.
PQsetnonblocking	Kullanılan bağlantı tipinin programı bloke edip etmeyeceğinin ayarlanması.
PQconsumeInput	Sunucudan dönen verinin istemci soketi üzerinden alınması.
PQflush	İstemci soketindeki verinin sunucuya gönderilmek üzere temizlenmesi.
PQfinish	Bağlantının kapatılması.
PQregisterThreadLock	Bağlantı fonksiyonları tarafından kullanılmak üzere yivler arası kilit sağlayacak fonksiyonun bildirimi.
PQreset	Senkron bir bağlantının sıfırlanıp tekrar kurulması.
PQresetStart PQresetPoll	Asenkron bir bağlantının sıfırlanıp tekrar kurulması.
PQexec PQexecParams	Senkron sorgu işletimi.
PQprepare PQexecPrepared	Senkron olarak sorgunun hazırlanıp, daha önceden hazırlanan bir sorgunun yine senkron olarak işletilmesi.
PQsendQuery PQsendQueryParams	Asenkron sorgu işletimi.
PQsendPrepare PQsendQueryPrepared	Asenkron sorgu hazırlanması ve işletimi.
PQgetResult	Asenkron sorgu sonuçlarının toplanması.
PQgetCopyData PQputCopyData PQendCopyData	COPY sorgusu için yardımcı fonksiyonlar.
PQgetCancel PQfreeCancel PQcancel PQrequestCancel	Sorgu iptali için yardımcı fonksiyonlar.
PQresultStatus PQresStatus	Sorgu durumu.
PQresultErrorMessage PQresultErrorField	Başarısız sorgu sonucu dönecek hata mesajlarının alımı.
PQcmdStatus	Sorgu sonucu istemciye dönen cevap.
PQcmdTuples	Sorgu sonucu etkilenen satır sayısı.
PQoidValue PQoidStatus	Sorgu sonucu hakkında ilgili *OID* değerleri.
PQftable PQftablecol PQftype PQfmod PQfname PQfnumber PQfsize	Sorgu sonucu dönen bir sütun hakkında ilgili bilgi.

PQgetlength PQisnull	
PQnfields PQntuples	Sorgu sonucunun kaç satır ve sütundan oluştuğu.
PQbinaryTuples	Sorgu sonucunun ikili biçimde veri içerip içermediği.
PQgetvalue	Sorgu sonucunun belirtilen satır ve sütundaki değerinin alınması.
PQprint	Sorgu sonucunun belirtilen dosya işaretçisine yazılması.
PQclear	Sorgu sonucunun istemci belleğinden temizlenmesi.
PQnotifies	Asenkron uyarılma için yardımcı fonksiyon.
PQsetNoticeProcessor PQsetNoticeReceiver	Uyarı mesajlarının alınıp işlenmesini sağlayacak fonksiyonların atanması.
lo_creat	*LO* kaydının oluşturulması.
lo_unlink	*LO* kaydının silinmesi.
lo_export	*LO* kaydının aktarılması.
lo_open lo_write lo_read lo_tell lo_lseek lo_close	*LO* kaydına ulaşılması.
PQtrace PQuntrace	Bağlantının dinlenip, kayıtların belirtilen dosya işaretçisine yazılması.
PQmakeEmptyPGresult	Boş bir sonuç yapısının oluşturulması.
PQescapeString PQescapeBytea PQunescapeString	Karakter katarlarının *SQL-Injection*'a karşı ayıklanması.
PQfreemem	PGnotify yapısı ve ayıklanmış karakter katarlarının istemci belleğinden temizlenmesi.

19. Örnekler

Buraya kadar yer vermeye çalıştığımız tüm libpq kütüphane fonksiyonlarına aşağıdaki örneklerde yer vermeye çalıştık.

```
/*
 * Dosya   : tanitim_uygulamasi.c
 * Açıklama: PostgreSQL C API kütüphanesine (libpq) giriş amaçlı bir örnek.
 *           Yaratılan bir ürün tablosu ve çeşitli Pl/PgSQL prosedürleri
 *           yardımıyla tabloya girdi eklenip üzerinde düzenleme
 *           yapılmasını sağlıyor.
 *
 * Kullanılan kütüphane fonksiyonları:
 * - Örnek ile ilgili : PQconnectdb PQerrorMessage PQexec PQfinish PQgetvalue
 *                      PQntuples PQresultErrorMessage PQresultStatus PQstatus
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include “libpq-fe.h”

/* Karakter dizileri için maksimum büyüklük. */
#define MAX_TAMPON      1024

/*
 * Pl/PgSQL prosedürlerinden dönecek cevapların anlamı.
```

```
 * (İlk 8 bitten sonrası ekbilgi olarak işlenecek.)
 */
#define TAMAM           1 << 0
#define YANLIS_ADET     1 << 1
#define KAYIT_MEVCUT    1 << 2
#define KAYIT_YOK       1 << 3
#define ADET_YETERSIZ   1 << 4
#define BILINMEYEN_HATA 1 << 5

/* Menü işlemlerini numaralandırıyoruz. */
enum
{
    URUN_LISTELE = 1,
    URUN_EKLE,
    URUN_CIKAR,
    ADET_EKLE,
    ADET_CIKAR,
    PROG_CIKIS
};

void     Cik(PGconn *);
PGconn  *BaglantiKur(const char *);
int      SorguKontrol(PGresult *);
int      SatirOku(char *, size_t, int);
int      DurumAciklama(int);
int      Menu(void);


/* Beklenmedik bir durumuda bağlantıyı kapatıp hata kodu ile çıkılacak. */
void
Cik(PGconn *bag)
{
    PQfinish(bag);
    exit(1);
}

/*
 * Veritabanı ile bağlantı kurmaya çalışıp ilgili
 * kontrolleri gerçekleştirecek fonksiyon.
 */
PGconn *
BaglantiKur(const char *ozellikler)
{
    PGconn *bag;

    bag = PQconnectdb(ozellikler);
    if (PQstatus(bag) != CONNECTION_OK)
    {
        fprintf(stderr, "Bağlantı kurulurken bir hata oluştu!\n");
        fprintf(stderr, "Hata mesajı:\n%s", PQerrorMessage(bag));
        Cik(bag);
    }

    return bag;
}

/*
 * Parametre olarak aldığı prosedürlerden dönen sorgu sonucunu durumuna
 * göre değerlendirip, cevabı kendisine çağırana iletecek olan fonksiyon.
 * (En sonda PQclear() ile sonucun bir daha kullanılmamak üzere bellekten
 * bırakılmasını sağlıyor.)
 */
int
SorguKontrol(PGresult *sonuc)
{
    int durum;

    if (PQresultStatus(sonuc) != PGRES_TUPLES_OK)
    {
```

```
        fprintf(stderr, "  Sorgu sonucu beklenmeyen bir hata oluştu!\n");
        fprintf(stderr, "  Hata mesajı:\n%s", PQresultErrorMessage(sonuc));

        durum = 0;
    }
    else
        durum = atoi(PQgetvalue(sonuc, 0, 0));

    PQclear(sonuc);
    return durum;
}

/* Kullanıcıdan girdi almak için kullanacağımız fonksiyon. */
int
SatirOku(char *tampon, size_t uzunluk, int isnumeric)
{
    fgets(tampon, uzunluk, stdin);
    return (isnumeric) ? atoi(tampon) : 0;
}

/* Kullanıcı menüsünü ekrana basacak olan fonksiyon. */
int
Menu(void)
{
    int         girdi;
    static char tampon[MAX_TAMPON];

    while (1)
    {
        printf("\nAşağıdaki menüden yapmak istediğiniz işlemi seçiniz:\n");
        printf("\t%d  Ürünleri Listele\n", URUN_LISTELE);
        printf("\t%d  Ürün Ekle\n", URUN_EKLE);
        printf("\t%d  Ürün Çıkar\n", URUN_CIKAR);
        printf("\t%d  Adet Ekle\n", ADET_EKLE);
        printf("\t%d  Adet Çıkar\n", ADET_CIKAR);
        printf("\t%d  Programdan Çıkış\n", PROG_CIKIS);

        printf("Girdi: "); girdi = SatirOku(tampon, MAX_TAMPON, 1);
        if ( girdi < URUN_LISTELE || girdi > PROG_CIKIS )
        {
            fprintf(stderr, "Yanlış menü numarası!\n");
            continue;
        }
        break;
    }

    return girdi;
}

/*
 * Prosedürlerden dönen kodların açıklamasını ekrana yazıp,
 * duruma göre ilgili ek bilgiyi geri döndürecek olan fonksiyon.
 */
int
DurumAciklama(int durum)
{
    int ekbilgi = -1;

    if (!durum)
        printf("Hata oluştu!\n");
    else if ((durum&TAMAM))
    {
        printf("Sorgu başarı ile gerçekleşti.\n");
        ekbilgi = durum >> 8;
    }
    else if ((durum&YANLIS_ADET))
        printf("Girilen adet pozitif bir tamsayı olmalı!\n");
    else if ((durum&KAYIT_MEVCUT))
    {
```

```
        printf("Eklenmek istenen model zaten mevcut.\n");
        ekbilgi = durum >> 8;
    }
    else if ((durum&KAYIT_YOK))
        printf("Böyle bir ürün mevcut değil!\n");
    else if ((durum&ADET_YETERSIZ))
    {
        printf("Mevcut ürün sayısı istenen adetin altında.\n");
        ekbilgi = durum >> 8;
    }

    return ekbilgi;
}

int
main(void)
{
    PGconn      *bag;
    PGresult    *sonuc;
    int          menu, urun, adet, durum, ekbilgi;
    char         tampon[MAX_TAMPON], komut[MAX_TAMPON];
    int          i, j;

    bag = BaglantiKur("dbname=test");

    while ((menu = Menu()))
    {
        switch (menu)
        {
            case URUN_LISTELE:
                printf("Ürünler listeleniyor:\n");
                sonuc = PQexec(bag, "SELECT urun, model, adet FROM urunler");
                durum = PQresultStatus(sonuc);
                if (durum == PGRES_TUPLES_OK)
                {
                    i = PQntuples(sonuc);
                    if (!i)
                        printf("  Şu an kayıtlı hiçbir ürün bulunmamakta.\n");
                    else
                    {
                        printf("   Ürün (ID) | Model           | Adet\n");
                        printf("  -----------+-----------------+------\n");
                        for (j = 0; j < i; j++)
                            printf("   %9s | %15s | %s\n",
                                   PQgetvalue(sonuc, j, 0),
                                   PQgetvalue(sonuc, j, 1),
                                   PQgetvalue(sonuc, j, 2));
                    }
                }
                else
                {
                    fprintf(stderr, "Sorgu sonucunda beklenmeyen bir cevap döndü!\n");
                    fprintf(stderr, "Sorgu durumu: %s\n", PQresStatus(PQresultStatus(sonuc)));
                    fprintf(stderr, "İlgili hata mesajı: %s", PQresultErrorMessage(sonuc));
                }
                PQclear(sonuc);
                break;

            case URUN_EKLE:
                printf("Yeni ürün ekleniyor:\n");
                printf("  Ürün modelini girin: ");
                SatirOku(tampon, (MAX_TAMPON/2), 0);
                printf("  Ürün adetini girin : ");
                adet = SatirOku(tampon+(MAX_TAMPON/2), (MAX_TAMPON/2), 1);
                sprintf(komut, "SELECT urun_ekle('%s', %d)", tampon, adet);
                sonuc = PQexec(bag, komut);
                durum = SorguKontrol(sonuc);
                printf("  Son durum          : ");
                ekbilgi = DurumAciklama(durum);
```

```
                if (ekbilgi != -1)
                    printf("  Ürün ID değeri     : %d\n", ekbilgi);
                break;

            case URUN_CIKAR:
                printf("Ürün çıkarılıyor:\n");
                printf("  Ürün ID değerini girin: ");
                urun = SatirOku(tampon, MAX_TAMPON, 1);
                sprintf(komut, "SELECT urun_cikar(%d)", urun);
                sonuc = PQexec(bag, komut);
                durum = SorguKontrol(sonuc);
                printf("  Son durum          : ");
                ekbilgi = DurumAciklama(durum);
                break;

            case ADET_EKLE:
                printf("Tabloya adet ekleniyor:\n");
                printf("  Ürün ID değerini girin: ");
                urun = SatirOku(tampon, MAX_TAMPON, 1);
                printf("  Ürün adetini girin    : ");
                adet = SatirOku(tampon, MAX_TAMPON, 1);
                sprintf(komut, "SELECT adet_ekle(%d, %d)", urun, adet);
                sonuc = PQexec(bag, komut);
                durum = SorguKontrol(sonuc);
                printf("  Son durum             : ");
                ekbilgi = DurumAciklama(durum);
                if (ekbilgi != -1)
                    printf("  Yeni ürün adeti       : %d\n", ekbilgi);
                break;

            case ADET_CIKAR:
                printf("Tablodan ürün alınıyor:\n");
                printf("  Ürün ID değerini girin: ");
                urun = SatirOku(tampon, MAX_TAMPON, 1);
                printf("  İstenen adeti girin   : ");
                adet = SatirOku(tampon, MAX_TAMPON, 1);
                sprintf(komut, "SELECT adet_cikar(%d, %d)", urun, adet);
                sonuc = PQexec(bag, komut);
                durum = SorguKontrol(sonuc);
                printf("  Son durum             : ");
                ekbilgi = DurumAciklama(durum);
                if (ekbilgi != -1)
                    printf("  Kalan ürün adeti      : %d\n", ekbilgi);
                break;

            case PROG_CIKIS:
                printf("Programdan çıkılıyor...\n");
                PQfinish(bag);
                return 0;
        }
    }

    PQfinish(bag);
    return 0;
}


/*
 * Dosya   : senkron_baglanti_ve_durum_bilgisi.c
 * Açıklama: Tanıtılan tüm senkron bağlantı ve durum bilgisi
 *           fonksiyonlarının örnek kullanımları.
 *
 * Kullanılan kütüphane fonksiyonları:
 * - Örnek ile ilgili : PQbackendPID PQconndefaults PQdb PQgetssl PQhost
 *                      PQoptions PQparameterStatus PQpass PQport
 *                      PQprotocolVersion PQserverVersion
 *                      PQtransactionStatus PQtty PQuser
 * - Sık kullanılanlar: PQconnectdb PQerrorMessage PQfinish PQstatus
```

```
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include “libpq-fe.h”

int
main(void)
{
    /* Bağlantı katarı. */
    const char *bagKatari = "dbname=template1";

    /* Öntanımlı bağlantı değişkenleri. */
    PQconninfoOption *ontanimliBagDeg;

    /* Bağlantı değişkeni. */
    PGconn *bag;

    /* Bağlantı kurmaya çalışıyoruz... */
    printf("Bağlantı kuruluyor... ");
    bag = PQconnectdb(bagKatari);
    if (PQstatus(bag) != CONNECTION_OK)
    {
        printf("hata!\n");
        fprintf(stderr, "Bağlantı kurulmaya çalışırken hata oluştu. "
                "İlgili hata mesajı: %s", PQerrorMessage(bag));
        /*
         * Her seferinde (hata oluşsa dahi) bağlantıyı kapatıyoruz ki
         * bağlantı için bellekte kullanılan bir çok tampon alan
         * serbest bırakılsın. (Buna bağlantı değişkeni ve hata mesajları
         * için tutulan alan da dahil olmak üzre.)
         */
        PQfinish(bag);

        exit(1);
    }
    else
        printf("tamam.\n");

    /* Bakalım bağlanırken ne gibi bağlantı seçenekleri kullanmışız. */
    printf("Bağlantıda kullanılan `dbname'   : %s\n", PQdb(bag));
    printf("                   ... `user'     : %s\n", PQuser(bag));
    printf("                   ... `password' : %s\n", PQpass(bag));
    printf("                   ... `host'     : %s\n", PQhost(bag));
    printf("                   ... `port'     : %s\n", PQport(bag));
    printf("                   ... `tty'      : %s\n", PQtty(bag));
    printf("                   ... `options'  : %s\n", PQoptions(bag));

    /* Öntanımlı bağlantı değişkenleri. */
    ontanimliBagDeg = PQconndefaults();
    printf("\nÖntanımlı bağlantı değişkenleri:\n\n");
    while (ontanimliBagDeg->keyword != NULL)
    {
        printf("\tkeyword  : %s\n", ontanimliBagDeg->keyword);
        printf("\tenvvar   : %s\n", ontanimliBagDeg->envvar);
        printf("\tcompiled : %s\n", ontanimliBagDeg->compiled);
        printf("\tval      : %s\n", ontanimliBagDeg->val);
        printf("\tlabel    : %s\n", ontanimliBagDeg->label);
        printf("\tdispchar : %s\n", ontanimliBagDeg->dispchar);
        printf("\tdispsize : %d\n", ontanimliBagDeg->dispsize);
        putchar('\n');
        ++ontanimliBagDeg;
    }

    /* Bağlantının kurulu olduğu sunucu hakkında bilgi dökümü. */
    printf("Kullanılan bağlantı protokolünün "
           "versiyonu: %d\n", PQprotocolVersion(bag));
```

```
    printf("Veritabanı sunucusunun versiyonu: %d\n", PQserverVersion(bag));
    printf("Bağlantıyı oluşturan işlemin PID'i: %d\n", PQbackendPID(bag));

    printf("Veritabanı sunucusunun çalıştırılması "
           "esnasında kullanılan parametreler:\n");
    printf("  server_version        : %s\n",
           PQparameterStatus(bag, "server_version"));
    printf("  server_encoding       : %s\n",
           PQparameterStatus(bag, "server_encoding"));
    printf("  client_encoding       : %s\n",
           PQparameterStatus(bag, "client_encoding"));
    printf("  is_superuser          : %s\n",
           PQparameterStatus(bag, "is_superuser"));
    printf("  session_authorization : %s\n",
           PQparameterStatus(bag, "session_authorization"));
    printf("  DateStyle             : %s\n",
           PQparameterStatus(bag, "DateStyle"));
    printf("  TimeZone              : %s\n",
           PQparameterStatus(bag, "TimeZone"));
    printf("  integer_datetimes     : %s\n",
           PQparameterStatus(bag, "integer_datetimes"));

    /* Bağlantının SSL desteğine sahip olup olmadığına bakılıyor. */
    printf("Bağlantı SSL desteğine sahip mi? ");
    if (PQgetssl(bag) != NULL)
        printf("Evet.\n");
    else
        printf("Hayır!\n");

    /* Veritabanımız ne durumda? */
    printf("Veritabanı transaction durumu: ");
    switch (PQtransactionStatus(bag)) {
        case PQTRANS_IDLE: printf("IDLE\n"); break;
        case PQTRANS_ACTIVE: printf("ACTIVE\n"); break;
        case PQTRANS_INTRANS: printf("INTRANS\n"); break;
        case PQTRANS_INERROR: printf("INERROR\n"); break;
        case PQTRANS_UNKNOWN: printf("PQTRANS_UNKNOWN\n"); break;
        default: printf("Bilinmiyor!\n");
    }

    /* Bağlantıyı kapatıyoruz. */
    PQfinish(bag);

    return 0;
}


/*
 * Dosya   : senkron_sorgu_isletimi.c
 * Açıklama: Sırasıyla şu adımları gerçekleştirmeye çalışacağız:
 *
 *               i. Veritabanı ile senkron bir bağlantı kur.
 *              ii. Bağlantı üzerinden kullanacağımız geçici tabloyu yarat.
 *             iii. Yaratılan tablo üzerine rastgele veriler gir.
 *              iv. Girilen verileri tablodan okuyup ekrana yaz.
 *             vi. Bağlantıyı kopar.
 *
 * Hazırlık: Program içinde bağlantı hareketleri, programın çalıştırılacağı
 *             dizin altındaki KAYIT_DOSYASI içine yazılacaktır. (Programı
 *             çalıştırmadan önce bu dosyayı oluşturmalısınız.) Programı
 *             çalıştırdıktan sonra kayıt dosyasına, bağlantıda geçen iletişimi
 *             gözlemlemek için bakabilirsiniz.
 *
 * Kullanılan kütüphane fonksiyonları:
 * - Örnek ile ilgili : PQexec PQfname PQgetvalue PQnfields PQntuples
 *                      PQtrace PQuntrace
 * - Sık kullanılanlar: PQclear PQconnectdb PQerrorMessage PQfinish
 *                      PQresultErrorMessage PQresultStatus PQstatus
```

```
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include “libpq-fe.h”

/*
 * SQL komutlarının saklanmasında (programı uzatmamak için) sınırlı
 * bir katar kullanacağız. Kataın sınırlarını burada belirliyoruz.
 */
#define MAX_SORGU_UZ 128

/* Kaç adet satırı veri olarak gireceğimiz. */
#define MAX_SATIR_SAY 10

/*
 * Aşağıda kullanacağımız türkçe karakter(ler) için sunucu ile
 * aramızdaki karakter kodlamasını ayarlıyoruz. (Ben bu C dosyasını
 * yazarken LATIN5 karakter setini kullandığımdan bağlantı için de
 * karakter kodu olarak LATIN5 kullanacağım. Siz isteğinize bağlı
 * olarak UTF-8 de kullanabilirsiniz.)
 */
#define KAR_KODLAMASI "LATIN5"

/* Bağlantı hareketlerinin kaydının tutulacağı dosya. */
#define KAYIT_DOSYASI "baglanti_kaydi.txt"

void         sisCikis(PGconn *, int);
void         hataliSonucCikis(PGconn *, PGresult *, int);
PGresult    *sorgula(PGconn *, const char *, const char *, ExecStatusType);

/* Sistemden çıkış için kullanacağımız fonksiyon. */
void
sisCikis(PGconn *bag, int hatanu)
{
    if (PQstatus(bag) != CONNECTION_OK)
        fprintf(stderr, "Bağlantı hata mesajı: %s", PQerrorMessage(bag));
    PQfinish(bag);

    if (hatanu)
    {
        printf("Sistem hata kodu açıklaması: %s\n", strerror(hatanu));
        exit(hatanu);
    }
    else
        exit(1);
}

/* Beklenmeyen bir sonuç nedeniyle çıkış. */
void
hataliSonucCikis(PGconn *bag, PGresult *sonuc, int hatanu)
{
    fprintf(stderr, "Beklenmeyen bir hata oluştu.\n");
    fprintf(stderr, "PGresult yapısının son durumu: `%s'.\n",
            PQresStatus(PQresultStatus(sonuc)));
    fprintf(stderr, "Dönen sonuç için hata mesajı:\n%s",
            PQresultErrorMessage(sonuc));

    fprintf(stderr, "Transaction iptal ediliyor... ");
    sonuc = PQexec(bag, "ROLLBACK");
    if (PQresultStatus(sonuc) == PGRES_COMMAND_OK)
        printf("tamam.\n");
    else
        printf("hata!\n");

    PQclear(sonuc);
```

```
    sisCikis(bag, hatanu);
}

/*
 * Yapacağımız bir çok sorgulama işlemi için her seferinde
 * tekrar edeceğimiz kontrol mekanizmalarını tek bir
 * fonksiyonda topluyoruz.
 */
PGresult *
sorgula(PGconn *bag, const char *aciklama,
        const char *sorguKat, ExecStatusType beklenenDurum)
{
    int        hatanu = 0;
    PGresult  *sonuc;

    printf("%s... ", aciklama);
    sonuc = PQexec(bag, sorguKat);
    if (PQresultStatus(sonuc) != beklenenDurum)
    {
        hatanu = errno;
        printf("hata!\n");
        hataliSonucCikis(bag, sonuc, hatanu);
    }
    else
        printf("tamam.\n");

    return sonuc;
}

int
main(void)
{
    /* Hata numarasını saklayacağımız değişken. */
    int             hatanu = 0;

    /* İçinde SQL komutlarını tutacağımız katar. */
    char            sorguKat[MAX_SORGU_UZ];

    PGconn         *bag;
    PGresult       *sonuc;

    /* Bağlantı hareketlerinin kaydedileceği dosya işaretçisi. */
    FILE           *kayitDosyasi;

    register int    i, j;
    int             sutunSayisi, satirSayisi;

    printf("Veritabanı ile senkron bağlantı kuruluyor... ");
    bag = PQconnectdb("dbname=test");
    if (PQstatus(bag) != CONNECTION_OK)
    {
        hatanu = errno;
        printf("hata!\n");
        sisCikis(bag, hatanu);
    }
    else
        printf("tamam.\n");

    printf("Kayıt dosyası açılıyor... ");
    kayitDosyasi = fopen(KAYIT_DOSYASI, "r+");
    if (kayitDosyasi == NULL)
    {
        hatanu = errno;
        printf("hata!\n");
        sisCikis(bag, hatanu);
    }
    else
    {
        printf("tamam.\n");
```

```
    PQtrace(bag, kayitDosyasi);
}

/* Hata mesajı duyarlılığı ayarlanıyor. */
PQsetErrorVerbosity(bag, PQERRORS_VERBOSE);

snprintf(sorguKat, MAX_SORGU_UZ,
        "SET client_encoding TO '%s'", KAR_KODLAMASI);
PQclear(sorgula(bag, "Karakter kodlaması ayarlanıyor",
        sorguKat, PGRES_COMMAND_OK));

PQclear(sorgula(bag, "Transaction başlatılıyor",
        "BEGIN", PGRES_COMMAND_OK));

PQclear(sorgula(bag, "Tablo oluşturuluyor",
        "CREATE TEMPORARY TABLE ornek_tablo ("
        "  nu SMALLINT, "
        "  veri VARCHAR(128)"
        ")",
        PGRES_COMMAND_OK));

printf("Tabloya veri giriliyor... [");
for (i = 0; i < MAX_SATIR_SAY; i++)
{
    printf("%d,", i);
    snprintf(sorguKat, MAX_SORGU_UZ,
             "INSERT INTO ornek_tablo VALUES "
             "(%d, '%d tas has hoşaf.')", i, i);
    sonuc = PQexec(bag, sorguKat);
    if (PQresultStatus(sonuc) == PGRES_COMMAND_OK)
        PQclear(sonuc);
    else
    {
        hatanu = errno;
        printf("\b] hata!\n");
        hataliSonucCikis(bag, sonuc, hatanu);
    }
}
printf("\b] tamam.\n");

sonuc = sorgula(bag, "Tablodan veri okunuyor",
                "SELECT * FROM ornek_tablo", PGRES_TUPLES_OK);

satirSayisi = PQntuples(sonuc);
sutunSayisi = PQnfields(sonuc);
printf("\nToplam satır sayısı: %d\n", satirSayisi);
printf("       sütun sayısı: %d\n\n", sutunSayisi);

printf("Sütun başlıkları:\n");
for (j = 0; j < sutunSayisi; j++)
    printf("  %d: %s\n",
           PQfnumber(sonuc, PQfname(sonuc, j)),
           PQfname(sonuc, j));
putchar('\n');

printf("Okunan veri:\n");
for (i = 0; i < satirSayisi; i++)
{
    for (j = 0; j < sutunSayisi; j++)
        printf("`%s' ", PQgetvalue(sonuc, i, j));
    putchar('\n');
}
putchar('\n');

PQclear(sonuc);

PQclear(sorgula(bag, "Transaction sonlandırılıyor",
        "COMMIT", PGRES_COMMAND_OK));
```

```
    /* Bağlantı hareket kaydını kapatıyoruz. */
    PQuntrace(bag);

    PQfinish(bag);

    return 0;
}



/*
 * [asenkron_baglanti.c]
 * Veritabanı ile asenkron bir bağlantı kurmak için,
 * sunucunun nasıl yoklanacağına dair ufak bir örnek.
 *
 * Kullanılan kütüphane fonksiyonları:
 * - Örnek ile ilgili : PQconnectPoll PQconnectStart PQsocket
 * - Sık kullanılanlar: PQerrorMessage PQfinish PQstatus
 */

#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <sys/poll.h>

#include “libpq-fe.h”

void    Cikis(PGconn *);
void    Yaz_BaglantiDurumu(PGconn *);
void    Yaz_YoklamaDurumu(PGconn *, int);
int     SoketDinle(int, int, int, time_t);

void
Cikis(PGconn *bag)
{
    if (PQstatus(bag) != CONNECTION_OK)
        fprintf(stderr, "Bağlantı hata mesajı: %s", PQerrorMessage(bag));

    PQfinish(bag);
    exit(1);
}

/* Bağlantı durumunu ekrana yazdır. */
void
Yaz_BaglantiDurumu(PGconn *bag)
{
    printf("Bağlantı durumu: ");
    switch (PQstatus(bag))
    {
        case CONNECTION_OK: printf("OK\n"); break;
        case CONNECTION_BAD: printf("BAD\n"); Cikis(bag); break;
        case CONNECTION_STARTED: printf("STARTED\n"); break;
        case CONNECTION_MADE: printf("MADE\n"); break;
        case CONNECTION_AWAITING_RESPONSE: printf("AWAITING_RESPONSE\n"); break;
        case CONNECTION_AUTH_OK: printf("AUTH_OK\n"); break;
        case CONNECTION_SETENV: printf("SETENV\n"); break;
        case CONNECTION_SSL_STARTUP: printf("SSL_STARTUP\n"); break;
        case CONNECTION_NEEDED: printf("NEEDED\n"); break;
        default: printf("UNKNOWN\n"); Cikis(bag);
    }
}

/* Yoklama durumunu ekrana yazdır. */
void
Yaz_YoklamaDurumu(PGconn *bag, int yoklama_sonucu)
{
    printf("Yoklama durumu: ");
    switch (yoklama_sonucu)
```

```
    {
        case PGRES_POLLING_OK: printf("OK\n"); break;
        case PGRES_POLLING_READING: printf("READING\n"); break;
        case PGRES_POLLING_WRITING: printf("WRITING\n"); break;
        case PGRES_POLLING_ACTIVE: printf("ACTIVE\n"); break;
        case PGRES_POLLING_FAILED: printf("FAILED\n"); Cik(bag);
        default: printf("UNKNOWN\n"); Cikis(bag);
    }
}

/*
 * İlgili soketin belirtilen kipte (okuma ve/veya yazma)
 * dinlenmesi için kullanılacak olan fonksiyon.
 */
int
SoketDinle(int soket, int okuma, int yazma, time_t zaman_asimi)
{
    struct pollfd girdi_dt;

    girdi_dt.fd = soket;
    girdi_dt.events = POLLERR;
    if (okuma) girdi_dt.events |= POLLIN;
    if (yazma) girdi_dt.events |= POLLOUT;
    girdi_dt.revents = 0;

    /*
     * Soketi dinlerken standart kütüphane tarafından sağlanan
     * poll() fonksiyonunu kullandık. (İsteğe bağlı olarak
     * select() de kullanılabilir.)
     */
    return poll(&girdi_dt, 1, zaman_asimi);
}

int
main(void)
{
    PGconn     *bag;
    const char  bag_secenekleri[] = "dbname=test";
    int         yoklama_durumu;
    int         soket;

    time_t      toplam_bekleme_suresi = 3000000; /* 3.0 saniye. */
    time_t      bekleme_suresi = 300000;         /* 0.3 saniye. */
    time_t      gecen_sure = 0;

    bag = PQconnectStart(bagSecenekleri);
    if (!bag)
    {
        fprintf(stderr, "Bağlantı yapısı için yeterli bellek ayrılamadı!\n");
        Cik(bag);
    }
    else
        printf("tamam.\n");

    /*
     * Asenkron işleyişi bloke etmek ile ilgili olan uyarıyı
     * hatırlayıp, kullanacağımız soketi öğrenip ekrana yazdırıyoruz.
     */
    soket = PQsocket(bag);
    if (soket < 0)
    {
        fprintf(stderr, "Bağlantı soketinde hata oluştu!\n");
        Cik(bag);
    }
    else
        printf("Bağlantı soketi: %d\n", soket);

    /*
     * İlk yoklama sonucu olarak değer atayıp,
```

```
     * bu değer ile döngüye gireceğiz.
     */
    yoklamaDurumu = PGRES_POLLING_WRITING;
    for (;;)
    {
        /* Zaman aşımı kontrolü. */
        if (gecenSure == toplamBeklemeSuresi)
        {
            fprintf(stderr, "Toplam bekleme süresi (%f saniye) aşıldı!\n",
                    ((float) toplamBeklemeSuresi / 1000000));
            Cik(bag);
        }
        printf("Toplam geçen süre : %.3f saniye.\n",
               ((float) gecenSure / 1000000));

        /* Son durumu ekrana yazdır. */
        yoklamaDurumKatari(bag, yoklamaDurumu);
        baglantiDurumKatari(bag);
        putchar('\n');

        switch (yoklamaDurumu)
        {
            case PGRES_POLLING_OK:
                printf("Bağlantı başarı ile gerçekleştirildi.\n");
                goto dongudenCik;

            case PGRES_POLLING_READING:
                if (soketDinle(soket, 1, 0, beklemeSuresi) < 0)
                    Cik(bag);
                break;

            case PGRES_POLLING_WRITING:
                if (soketDinle(soket, 0, 1, beklemeSuresi) < 0)
                    Cik(bag);
                break;

            default:
                fprintf(stderr, "Beklenmeyen bir hata oluştu.");
                Cik(bag);
        }

        /* Geçen süreyi hesaplayıp, sunucuyu terar yokluyoruz. */
        gecenSure += beklemeSuresi;
        yoklamaDurumu = PQconnectPoll(bag);
    }

dongudenCik:

    PQfinish(bag);
    return 0;
}



/*
 * Dosya   : asenkron_veri_alimi.c
 * Açıklama: Asenkron olarak, sunucu tarafından verinin
 *           beklenip, uygun olunduğunda alınması.
 *
 * Hazırlık: Aşağıdaki örnekte kullanacağımız ornektablo'nun
 *              yapısı şu şekilde olacak:
 *
 *              => CREATE TABLE ornektablo (
 *              -> numara int2
 *              ->   NOT NULL
 *              ->   DEFAULT 0,
 *              -> veri varchar(32)
 *              ->   NOT NULL
 *              ->   DEFAULT ''::character varying
```

```
 *              -> );
 *
 *              Ve tablomuzun içine kullanılmak üzere bir
 *              kaç satır ekliyoruz:
 *
 *              => SELECT numara,veri FROM ornektablo;
 *               numara | veri
 *              --------+-------
 *                    1 | veri1
 *                    2 | veri2
 *                    3 | veri3
 *                    4 | veri4
 *                    5 | veri5
 *                    6 | veri6
 *           (6 rows)
 *
 * Kullanılan kütüphane fonksiyonları:
 * - Örnek ile ilgili : PQconsumeInput PQgetResult PQisBusy PQsendQuery
 *                      PQsocket
 * - Sık kullanılanlar: PQconnectdb PQclear PQerrorMessage PQfinish
 *                      PQresultErrorMessage PQresultStatus PQstatus
 */

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#include “libpq-fe.h”

void    sisCikis(PGconn *);

/*
 * Herhangi bir hata durumunda bağlantıyı kapatıp çıkmak
 * için kullanacağımız fonksiyon.
 */
void
sisCikis(PGconn *baglanti)
{
    if (PQstatus(baglanti) != CONNECTION_OK)
        fprintf(stderr, "Bağlantı hata mesajı: %s", PQerrorMessage(baglanti));

    PQfinish(baglanti);
    exit(1);
}

int
main(void)
{
    const char     bagSecenekleri[] = "dbname=template1";
    PGconn        *bag;
    PGresult      *sonuc;

    /* Soketin dinlenmesinde kullanılacak olan değişkenler. */
    int            soket;
    fd_set         girdi;

    printf("Bağlantı kuruluyor... ");
    bag = PQconnectdb(bagSecenekleri);
    if (PQstatus(bag) != CONNECTION_OK)
    {
        printf("hata!\n");
        fprintf(stderr, "Bağlantı kurulurken beklenmedik bir hata oluştu!\n");
        sisCikis(bag);
    }
```

```
else
    printf("tamam.\n");

/* Sorgu karşı tarafa gönderiliyor. */
if (PQsendQuery(bag, "SELECT numara,veri FROM ornektablo") != 1)
{
    fprintf(stderr, "Sorgu gönderiminde hata oluştu!\n");
    sisCikis(bag);
}

/* Bağlantı soketi alınıyor. */
soket = PQsocket(bag);
if (soket < 0 )
{
    fprintf(stderr, "Bağlantı soketinde hata oluştu!\n");
    sisCikis(bag);
}
else
    printf("Bağlantı soketi: %d\n", soket);

/* select() için kullanacağımız yapılar hazırlanıyor. */
FD_ZERO(&girdi);
FD_SET(soket, &girdi);

for (;;)
{
    /* Girdi soketinde veri olup olmadığını dinliyoruz. */
    printf("Veri bekleniyor... ");
    if (select(soket+1, &girdi, NULL, NULL, NULL) < 0) {
        printf("hata!\n");
        fprintf(stderr, "select() esnasında hata oluştu!\n"
                "İlgili hata mesajı: [%d] %s\n",
                errno, strerror(errno));
        sisCikis(bag);
    }
    else
        printf("alındı.\n");

    /* Soket üzerinden veriyi çekiyoruz. */
    PQconsumeInput(bag);

    /* Eğer sunucu hala işlem yapıyorsa, daha veri gelecek demektir. */
    if (!PQisBusy(bag))
        break;
}

/*
 * Gelecek olan verinin dinlenip, uygun olunduğunda alınması genelde
 * bu şekilde yapılmamakla birlikte, bu örnekte sadece soketin nasıl
 * dinlenip, verinin nasıl alınabileceğini anlatmak amacıyla ana
 * hatların altı çizilmeye çalışılmıştır. Genelde gerçek bir uygulamada
 * ana bir döngü soket üzerinde dinleme işlemini gerçekleştirip, aldığı
 * veriye göre yivler aracılığı ile program akışının kontrolünü ve
 * alınan verinin işlenmesini sağlar.
 */

/* PGresult yapısı oluşturacak kadar veri ulaştı elimize. */
sonuc = PQgetResult(bag);

/* Alınan sonucun durumuna bakılıyor. */
if (PQresultStatus(sonuc) == PGRES_TUPLES_OK)
{
    printf("Sorgu başarı ile gerçekleşti.\n");

    /* Şu an için sonuç ile hiçbir işimiz yok. */
    PQclear(sonuc);
}
else
{
```

```
        fprintf(stderr, "Sorgu sonucunda hata oluştu!\n");
        fprintf(stderr, "İlgili hata mesajı: %s", PQresultErrorMessage(sonuc));
        PQclear(sonuc);
        sisCikis(bag);
    }

    PQfinish(bag);
    return 0;
}



/*
 * Dosya   : lo_fonksiyonlari.c
 * Açıklama: `Large Object' fonksiyonlarının kullanımı
 *           hakkında basit bir örnek.
 *
 * Kullanılan kütüphane fonksiyonları:
 * - Örnek ile ilgili : lo_creat, lo_import
 * - Sık kullanılanlar: PQclear PQconnectdb PQerrorMessage PQexec
 *                      PQfinish PQresultErrorMessage PQresultStatus
 *                      PQstatus
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include “libpq-fe.h”
#include “postgres_ext.h”        /* Oid tipi için */
#include “libpq/libpq-fs.h”      /* lo fonksiyonları için */

void    sisCikis(PGconn *, int);
void    dosyaCek(PGconn *);
void    dosyayiKendinOlustur(PGconn *);
void    sorgula(PGconn *, char *, char *, int);

/* Veritabanına eklenecek olan dosya. */
#define DOSYA_ADI "ornek_metin_dosyasi.txt"

/* Verinin okunup yazılmasından kullanılacak olan tampon bellek boyutu. */
#define TAMPON    8192

/* Sistemden çıkmak için kullanacağımız fonksiyon. */
void
sisCikis(PGconn *bag, int hatanu)
{
    if (PQstatus(bag) != CONNECTION_OK)
        fprintf(stderr, "Bağlantı hata mesajı: %s", PQerrorMessage(bag));
    PQfinish(bag);

    if (hatanu)
    {
        fprintf(stderr, "Sistem hata kodu açıklaması: [%d] %s\n",
                hatanu, strerror(hatanu));
        exit(hatanu);
    }
    else
        exit(1);
}

/* lo_import() komutu ile dosyayı veritabanına aktaracağız. */
void
dosyaCek(PGconn *bag)
```

```
{
    int hatanu;
    Oid yeni_lo;

    printf("Dosya lo_import() ile veritabanına aktarılıyor... ");
    yeni_lo = lo_import(bag, DOSYA_ADI);
    if (yeni_lo == InvalidOid)
    {
        hatanu = errno;
        printf("hata!\n");
        sisCikis(bag, hatanu);
    }
    else
    {
        printf("tamam.\n");
        printf("Yeni nesnenin OID değeri: %d.\n\n", yeni_lo);
    }
}

/*
 * Bu sefer lo_import() fonksiyonunu kullanmaktansa, burada kendi
 * yazacağımız kod ile sistem üzerindeki bir dosyayı okuduktan sonra
 * veritabanında oluşturduğumuz lo alanı içine atacağız.
 *
 * Takip edilecek işlem sırası tersine çevrildiğinde ise, lo_export()
 * fonksiyonunun işlevi ile karşılaşırız. Böyle bir durumda ise veritabanı
 * üzerindeki belirli bir lo kaydını lo_open() ile açıp lo_read()
 * ile okuduktan sonra standart kütüphanedeki open(), write()
 * fonksiyonlarını kullanarak ilgili dosyaya aktarabiliriz.
 */
void
dosyayiKendinOlustur(PGconn *bag)
{
    Oid     lo_oid;
    char    satir[TAMPON];
    int     hatanu;
    int     dt, lo_dt;
    int     okunan, yazilan;

    printf("Aktarımı gerçekleşecek dosya açılıyor... ");
    dt = open(DOSYA_ADI, O_RDONLY, S_IRUSR|S_IRGRP|S_IROTH);
    if (dt < 0)
    {
        hatanu = errno;
        printf("hata!\n");
        sisCikis(bag, hatanu);
    }
    else
        printf("tamam.\n");

    printf("Yeni lo oluşturuluyor... ");
    lo_oid = lo_creat(bag, INV_READ|INV_WRITE);
    if (lo_oid == InvalidOid) {
        hatanu = errno;
        printf("hata!\n");
        sisCikis(bag, hatanu);
    }
    else
        printf("tamam. [OID: %d]\n", lo_oid);

    printf("Oluşturulan lo açılıyor... ");
    lo_dt = lo_open(bag, lo_oid, INV_WRITE);
    if (lo_dt < 0)
    {
        hatanu = errno;
        printf("hata!\n");

        lo_close(bag, lo_dt);
        close(dt);
```

```
        sisCikis(bag, hatanu);
    }
    else
        printf("tamam.\n");

    printf("Dosya lo nesnesi içine yazdırılıyor... ");
    while ((okunan = read(dt, satir, TAMPON)) > 0)
        if ((yazilan = lo_write(bag, lo_dt, satir, okunan)) < okunan)
        {
            hatanu = errno;
            printf("hata!\n");
            fprintf(stderr, "Yazılan: %d/%d\n", yazilan, okunan);

            lo_close(bag, lo_dt);
            close(dt);
            sisCikis(bag, hatanu);
        }
    printf("tamam.\n");

    close(dt);
    lo_close(bag, lo_dt);
}

/*
 * Aynı satırları birkaç kez tekrarlamaktansa, hepsini bizim
 * yerimize bir tek fonksiyonda toplayacak bir sorgu fonksiyonu.
 */
void
sorgula(PGconn *bag, char *mesaj, char *komut, int durum)
{
    PGresult    *sonuc;
    int          hatanu;

    printf("%s... ", mesaj);
    sonuc = PQexec(bag, komut);
    if (PQresultStatus(sonuc) != durum)
    {
        hatanu = errno;
        printf("hata!\n");
        fprintf(stderr, "Sorgu hata mesajı: %s", PQresultErrorMessage(sonuc));
        PQclear(sonuc);

        if (PQstatus(bag) == CONNECTION_OK)
            PQclear(PQexec(bag, "ROLLBACK"));

        sisCikis(bag, hatanu);
    }
    else
    {
        printf("tamam.\n");
        PQclear(sonuc);
    }
}

int
main(void)
{
    PGconn    *bag;

    printf("Veritabanı ile bağlantı kuruluyor... ");
    bag = PQconnectdb("dbname=template1");
    if (PQstatus(bag) != CONNECTION_OK) {
        printf("hata!\n");
        sisCikis(bag, 0);
    }
    else
        printf("tamam.\n");

    /*
```

```
     * lo fonksiyonlarının trasnaction blokları içinde kullanıldığını
     * hatırlayarak, bir transaction başlatıyoruz.
     */
    sorgula(bag, "Transaction bloğu açılıyor",
            "BEGIN", PGRES_COMMAND_OK);

    /* Dosyayı lo_import() ile alacağız. */
    dosyaCek(bag);

    /*
     * Dosyayı kendimiz okuyup, yine kendimizin
     * oluşturduğu lo içine atacağız.
     */
    dosyayiKendinOlustur(bag);

    sorgula(bag, "Transaction bloğu kapatılıyor",
            "COMMIT", PGRES_COMMAND_OK);

    PQfinish(bag);
    return 0;
}



/*
 * Dosya   : sql_injection.c
 * Açıklama: SQL-Injection saldırılarının nasıl gerçekleşebileceği
 *           ve bu saldırılara karşı sorgu komutlarının nasıl kontrol
 *           edileceği hakkında kısa bir program.
 *
 * Hazırlık: Programı çalıştırmadan önce aşağıdaki işlemleri
 *             bağlanacağınız veritabanında gerçekleştirdiğinizden
 *             emin olun:
 *
 *             => CREATE TABLE kullanici_kayitlari (
 *             ->     kullanici VARCHAR(64) PRIMARY KEY,
 *             ->     parola VARCHAR(32) NOT NULL,
 *             ->     gizli_bilgi TEXT NOT NULL
 *             -> );
 *
 *             => INSERT INTO kullanici_kayitlari VALUES
 *             ->     ('volkan', 'parolam', 'Banka hesap şifrem...');
 *
 * Kullanılan kütüphane fonksiyonları:
 * - Örnek ile ilgili : PQescapeString PQexecParams
 * - Sık kullanılanlar: PQclear PQconnectdb PQerrorMessage PQexec
 *                      PQfinish PQntuples PQresultErrorMessage
 *                      PQresultStatus PQsetErrorVerbosity PQstatus
 */

#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

#include “libpq-fe.h”

/* Bir sorgu komutunda girilebilecek en fazla karakter sayısı. */
#define MAX_KATAR_UZ 512

void    sisCikis(PGconn *, int);
void    hataliCikis(PGconn *, PGresult *, int);
void    sonucuKontrolEt(const char *, PGconn *, PGresult *, int);

/* Sistemden çıkış. */
void
sisCikis(PGconn *bag, int hatanu)
{
    if (PQstatus(bag) != CONNECTION_OK)
```

```
        fprintf(stderr, "-> Bağlantı hata mesajı: %s", PQerrorMessage(bag));
    PQfinish(bag);

    if (hatanu)
    {
        printf("Sistem hata kodu açıklaması: %s\n", strerror(hatanu));
        exit(hatanu);
    }
    else
        exit(1);
}

/*
 * Sorgu sonucu başarılı değilse, programdan
 * çıkmak için kullanacağımız fonksiyon.
 */
void
hataliCikis(PGconn *bag, PGresult *sonuc, int hatanu)
{
    if (sonuc)
    {
        fprintf(stderr, "-> PGresult yapısının son durumu: `%s'.\n",
                PQresStatus(PQresultStatus(sonuc)));
        fprintf(stderr, "-> Dönen hata mesajı: %s",
                PQresultErrorMessage(sonuc));
    }
    PQclear(sonuc);

    sisCikis(bag, hatanu);
}

/*
 * PGresult yapılarından dönen sonuçları kontrol ediyoruz. Bakalım
 * SQL-Injection saldırısı önlenmiş mi yoksa başarılı mı olmuş?
 */
void
sonucuKontrolEt(const char *mesaj, PGconn *bag,
                      PGresult *sonuc, int hatanu)
{
    ExecStatusType durum = PQresultStatus(sonuc);

    printf("%s\n", mesaj);

    if (durum == PGRES_TUPLES_OK)
    {
        if (PQntuples(sonuc) == 0)
            printf("=> Saldırı başarısız oldu.\n");
        else
            printf("=> Saldırı başarılı oldu!\n");
    }
    else
    {
        fprintf(stderr, "-> Beklenmeyen bir hata oluştu!\n");
        hataliCikis(bag, sonuc, hatanu);
    }

    PQclear(sonuc);
}

int
main(void)
{
    PGconn     *bag;
    PGresult   *sonuc;

    /* Sistem hata kodu. */
    int         hatanu = 0;

    /* Kullanıcı bilgileri. */
```

```
const char  kullanici[] = "volkan";
const char  parola[] = "tehlikeliParola' OR '' = '";

/* PQescapeString() fonksiyonu için gerekli katarlar. */
char       *kontrolEdilmis_Kullanici;
char       *kontrolEdilmis_Parola;

/* PQexecParams() fonksiyonunda değerleri taşıyacak olan dizi. */
const char *degerler[2];

/* SQL sorgu komutlarımızı içine yazacağımız katar. */
char        komut[MAX_KATAR_UZ];

printf("-> Veritabanı ile bağlantı kuruluyor... ");
bag = PQconnectdb("dbname=test");
if (PQstatus(bag) != CONNECTION_OK)
{
    hatanu = errno;
    printf("hata!\n");
    sisCikis(bag, hatanu);
}
else
    printf("tamam.\n\n");

/* Hata mesajı duyarlılığı ayarlanıyor. */
PQsetErrorVerbosity(bag, PQERRORS_VERBOSE);

/*
 * Hiçbir kontrolden geçirmeden, girdileri komutta
 * yerine koyup aynen sorguyu çalıştırıyoruz.
 */
snprintf(komut, MAX_KATAR_UZ,
         "SELECT gizli_bilgi FROM kullanici_kayitlari WHERE "
         "kullanici = '%s' AND parola = '%s'",
         kullanici, parola);
printf("-> Çalıştırılacak olan komut: %s\n", komut);
sonuc = PQexec(bag, komut);
sonucuKontrolEt("-> Kullanıcı girdileri snprintf() ile sorgu komutu "
                "içine alındı.", bag, sonuc, errno);
putchar('\n');

/*
 * Girdileri PQescapeString() ile ayıkladıktan
 * sonra komutta yerine koyup sorgulamaya geçiyoruz.
 */

/* Değişkenler için gerekecek bellek alanı tahsis ediliyor. */
kontrolEdilmis_Kullanici = malloc(2 * strlen(kullanici) + 1);
kontrolEdilmis_Parola = malloc(2 * strlen(parola) + 1);
if (!kontrolEdilmis_Kullanici || !kontrolEdilmis_Parola)
{
    hatanu = errno;
    fprintf(stderr, "Değişkenler için gerekli bellek alanı ayrılamadı.");
    SisCikis(bag, hatanu);
}

/* Kullanıcı adı ve parola ayıklanıyor. */
PQescapeString(kontrolEdilmis_Kullanici, kullanici, strlen(kullanici));
PQescapeString(kontrolEdilmis_Parola, parola, strlen(parola));

snprintf(komut, MAX_KATAR_UZ,
         "SELECT gizli_bilgi FROM kullanici_kayitlari WHERE "
         "kullanici = '%s' AND parola = '%s'",
         kontrolEdilmis_Kullanici, kontrolEdilmis_Parola);
printf("-> Çalıştırılacak olan komut: %s\n", komut);
sonuc = PQexec(bag, komut);
sonucuKontrolEt("-> Kullanıcı girdileri PQescapeString() ile "
                "temizlendikten sonra sorgu komutu içine alındı.",
                bag, sonuc, errno);
```

```
    putchar('\n');

    free(kontrolEdilmis_Kullanici);
    free(kontrolEdilmis_Parola);

    /*
     * Sorgulamayı (girdileri ayıklamadan)
     * PQexecParams() kullanarak gerçekleştiriyoruz.
     */

    degerler[0] = kullanici;
    degerler[1] = parola;

    /*
     * PQexecParams() ile girilen komutta parametrelerin
     * etrafına ' işareti konmadığına dikkat edin.
     */
    sonuc = PQexecParams(bag,
                         "SELECT gizli_bilgi FROM kullanici_kayitlari "
                         "WHERE kullanici = $1 AND parola = $2",
                         2, NULL, degerler, NULL, NULL, 0);
    sonucuKontrolEt("-> Kullanıcı girdileri PQexecParams() ile "
                    "sorgulandı.", bag, sonuc, errno);

    PQfinish(bag);
    return 0;
}



/*
 * Dosya   : uyari_mesajlarinin_islenmesi.c
 * Açıklama: Sunucudan gelen uyarı mesajlarının (iki adımda ve tek adımda
 *           olmak üzere) iki yolla da nasıl işleneceğine dair basit
 *           bir örnek.
 *
 * Hazırlık: Uyarı mesajlarının oluşturulmasında örnek içinde kullanılacak
 *           aşağıdaki fonksiyonun, bağlanılacak veritabanı üzerinde
 *           oluşturulması gerekmektedir.
 *
 *           CREATE OR REPLACE FUNCTION uyari_uret () RETURNS integer AS $$
 *           BEGIN
 *               RAISE NOTICE 'NOTICE derecesinde örnek mesaj.';
 *               RAISE WARNING 'WARNING derecesinde örnek mesaj.';
 *
 *               RETURN 0;
 *           END
 *           $$ LANGUAGE 'plpgsql';
 *
 * Kullanılan kütüphane fonksiyonları:
 * - Örnek ile ilgili : PQsetNoticeProcessor PQsetNoticeReceiver
 * - Sık kullanılanlar: PQclear PQconnectdb PQexec PQfinish
 *                      PQresultErrorField PQresultErrorMessage
 *                      PQresultStatus PQstatus
 */

#include <stdio.h>
#include <stdlib.h>

#include “libpq-fe.h”

void    sisCikis(PGconn *);
void    uyariyiIsle(const char *, const char *);
void    uyariAldiktanSonraIsle(void *, const PGresult *);
void    banaGuzelBirUyariUret(PGconn *);

/*
 * Bağlantı nesnesini bellekten bırakıp,
 * programdan hata kodu ile çıkacak fonksiyon.
```

```
 */
void
sisCikis(PGconn *bag)
{
    if (PQstatus(bag) != CONNECTION_OK)
        fprintf(stderr, "!> Bağlantı hata mesajı: %s", PQerrorMessage(bag));
    PQfinish(bag);

    exit(1);
}

/* (1. tip) Uyarıyı işleyecek olan fonksiyon. */
void
uyariyiIsle(const char *arguman, const char *uyariMesaji)
{
    printf("#> uyariyiTekBasinaIsle:\n");
    printf("-> Gelen argüman: %s\n", arguman);
    printf("-> Gelen uyarı mesajı: %s", uyariMesaji);
}

/*
 * (2. tip) Uyarıyı aldıktan sonra, uyarıyı işleyecek
 * olan fonksiyonu çağıracak olan fonksiyon.
 */
void
uyariAldiktanSonraIsle(void *argumanlar, const PGresult *sonuc)
{
    printf("#> uyariAldiktanSonraIsle:\n");
    printf("-> PGresult yapısından elde edilen "
           "hata mesajı: %s", PQresultErrorMessage(sonuc));
    printf("-> Şimdi uyarıyı işleyecek olan fonksiyon çağrılıyor.\n");

    /*
     * Burada daha ayrıntılı sonuç almak için
     * PQresultErrorField() fonksiyonunu da kullanabilirsiniz.
     */
    uyariyiIsle("uyariAldiktanSonraIsle() fonksiyonundan geldi.",
                PQresultErrorMessage(sonuc));
}

/*
 * Uyarı işleyecek fonksiyonlarımız için birbirinden güzel
 * uyarılar üretecek olan sorguları çalıştıracak fonksiyonumuz.
 */
void
banaGuzelBirUyariUret(PGconn *bag)
{
    PGresult *sonuc;

    printf("&> banaGuzelBirUyariUret:\n");
    sonuc = PQexec(bag, "SELECT uyari_uret()");
    if (PQresultStatus(sonuc) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "!> Komut sorgulanırken hata oluştu! (%s)\n",
                PQresStatus(PQresultStatus(sonuc)));
        fprintf(stderr, "!> İlgili hata mesajı: %s", PQresultErrorMessage(sonuc));

        PQclear(sonuc);
        sisCikis(bag);
    }

    PQclear(sonuc);
    putchar('\n');
}

int
main(void)
{
    PGconn *bag;
```

```
    bag = PQconnectdb("dbname=test");
    if (PQstatus(bag) != CONNECTION_OK)
    {
        fprintf(stderr, "!> Bağlantı kurulumunda hata oluştu!\n");
        sisCikis(bag);
    }

    /* (1. tip) Sadece uyarıyı işleyecek olan fonksiyonu çağır. */
    PQsetNoticeProcessor(bag,
                         (PQnoticeProcessor) uyariyiIsle,
                         (void *) "PQsetNoticeProcessor() tarafıyla geldi.");
    banaGuzelBirUyariUret(bag);


    /*
     * (2. tip) Uyarıyı alacak olan fonksiyonu çağırıyoruz.
     * Fonksiyon kendi içinde daha sonra uyarıyı işleyecek
     * olan fonksiyonu çağırıyor.
     */
    PQsetNoticeReceiver(bag,
                        (PQnoticeReceiver) uyariAldiktanSonraIsle,
                        NULL);
    banaGuzelBirUyariUret(bag);

    PQfinish(bag);
    return 0;
}



/*
 * Dosya   : asenkron_uyarilma.c
 * Açıklama: Asenkron uyarı alımı için örnek program.
 *
 * Hazırlık: Programı denemek için derledikten sonra, bir terminalden
 *           programı çalıştırırken, diğer bir terminalden şu komutları
 *           veriniz:
 *
 *           -- Normal bir uyarı
 *           => NOTIFY uyari1;
 *
 *           -- Uyarı tekrarı
 *           => NOTIFY uyari1;
 *
 *           -- Çıkış sinyali
 *           => NOTIFY uyari2;
 *
 * Kullanılan kütüphane fonksiyonları:
 * - Örnek ile ilgili : PQconsumeInput PQfreemem PQnotifies PQsocket
 * - Sık kullanılanlar: PQclear PQconnectdb PQerrorMessage PQexec
 *                      PQfinish PQresultErrorMessage PQresultStatus
 *                      PQstatus
 */

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#include “libpq-fe.h”

/* Sorgu tümcelerini yerleştireceğimiz tampon bellek boyutu. */
#define MAX_SORGU_UZ    512
```

```
void    sisCikis(PGconn *, int);
void    sorguHata(PGconn *, PGresult *, int);
void    dinlemeyeAl(PGconn *, const char *);

/*
 * Herhangi bir bağlantı hatası durumunda bağlantı hata mesajını
 * ekrana yazdırıp, bağlantıyı kapattıktan sonra sistemden çıkmak
 * için kullanacağımız fonksiyon.
 */
void
sisCikis(PGconn *bag, int hatanu)
{
    if (PQstatus(bag) != CONNECTION_OK)
        fprintf(stderr, "-> Bağlantı hata mesajı: %s", PQerrorMessage(bag));
    PQfinish(bag);

    if (hatanu)
    {
        fprintf(stderr, "-> Sistem hata kodu açıklaması: %s\n",
                strerror(hatanu));
        exit(hatanu);
    } else
        exit(1);
}

/*
 * Herhangi bir sorgu hatası sonucunda ilgili hata mesajını ekrana
 * yazdırıp, sisCikis() cagrisi için kullancağımız fonksiyon.
 */
void
sorguHata(PGconn *bag, PGresult *sonuc, int hatanu)
{
    fprintf(stderr, "-> Sorgu hata mesajı: %s", PQresultErrorMessage(sonuc));
    PQclear(sonuc);

    sisCikis(bag, hatanu);
}

/* Herhangi bir uyarıyı aşağıdaki fonksiyon ile dinlemeye alacağız. */
void
dinlemeyeAl(PGconn *bag, const char *uyariAdi)
{
    int              hatanu = 0;
    static PGresult *sonuc;
    static char      sorguKomutu[MAX_SORGU_UZ];
    size_t           sorguUzunlugu;

    snprintf(sorguKomutu, MAX_SORGU_UZ, "LISTEN %s", uyariAdi);

    printf("-> `%s' dinlemeye alınıyor... ", uyariAdi);
    sonuc = PQexec(bag, sorguKomutu);
    if (PQresultStatus(sonuc) != PGRES_COMMAND_OK)
    {
        hatanu = errno;
        printf("hata!\n");
        sorguHata(bag, sonuc, hatanu);
    }
    else
    {
        printf("tamam.\n");
        PQclear(sonuc);
    }
}

int
main(void)
{
    /* Sistem hata numarasının tutulacağı değişken. */
    int            hatanu = 0;
```

```
PGconn        *bag;
PGnotify      *uyari;

/* Soket yapısında kullanacağımız değişkenler. */
int            soket;
fd_set         girdi;

printf("-> Veritabanı ile bağlantı kuruluyor... ");
bag = PQconnectdb("dbname=template1");
if (PQstatus(bag) != CONNECTION_OK)
{
    hatanu = errno;
    printf("hata!\n");
    sisCikis(bag, hatanu);
}
else
    printf("tamam.\n\n");

/*
 * Bağlantı soketi belirlenip, soket
 * değişkenleri ayarlanıyor.
 */
soket = PQsocket(bag);
if (soket < 0)
{
    hatanu = errno;
    fprintf(stderr, "-> Bağlantı soketini almaya çalışırken "
            "beklenmedik bir hata oluştu!\n");
    sisCikis(bag, hatanu);
}
else
    printf("Soket dosyası tanımlayıcısı: %d\n", soket);

/*
 * girdi değişkeni sıfırlandıktan sonra, dinlenecek
 * soketi tutacak şekilde ayarlanıyor.
 */
FD_ZERO(&girdi);
FD_SET(soket, &girdi);

dinlemeyeAl(bag, "uyari1");
dinlemeyeAl(bag, "uyari2");
putchar('\n');

for (;;)
{
    /* Bağlantı soketi dinleniyor. */
    if (select(soket + 1, &girdi, NULL, NULL, NULL) < 0)
    {
        hatanu = errno;
        fprintf(stderr, "select() işlemi başarısız sonuçlandı!\n");
        sisCikis(bag, hatanu);
    }

    /* Soket üzerindeki mevcut veriyi okuyoruz. */
    PQconsumeInput(bag);

    /*
     * Gelen veri üzerinde bizi ilgilendiren bir uyarı olup
     * olmadığını kontrol ediyoruz.
     */
    while ((uyari = PQnotifies(bag)) != NULL) {
        if (strcmp(uyari->relname, "uyari1") == 0)
        {
            printf("[%d] Uyarı1\n", uyari->be_pid);
            PQfreemem(uyari);
        }
        else
```

```
            {
                printf("[%d] Uyarı2 (Çıkış İsteği)\n", uyari->be_pid);
                PQfreemem(uyari);
                goto dongudenCik;
            }
        }
    }

dongudenCik:

    PQfinish(bag);
    return 0;
}
```

III. PHP Arayüzü

Bu bölümde, PHP programlama dilini kullanarak PostgreSQL veritabanına nasıl bağlanıp, üzerinde sorgulamalarımız gerçekleştirdikten sonra, sonuçlarını nasıl alabileceğimiz üzerinde durmaya çalışacağız.

PHP ile PostgreSQL veritabanı üzerinde gerçekleştirilecek işlemler için kullanılan PHP kurulumunda PostgreSQL desteği etkinleştirilmiş olmalı. Aksi halde, bu bölümde anlatılacak olan fonksiyonlar çağrılmaya çalışıldığında, PHP, öyle bir fonksiyon bulunamadığına dair hata verecektir.

> *PHP, PostgreSQL Programlama Arayüzü, artalanda libpq kütüphanesini kullandığından dolayı tanıtılacak fonksiyonlar hakkında ayrıntılı bilgi için ilgili fonksiyonların libpq kütüphanesindeki eşdeğerlerinin incelendiği kısımlara göz atabilirsiniz. (Hangi PHP fonksiyonunun, libpq kütüphanesindeki hangi fonksiyon ile eşdeğer olduğunu bölümün en sonunda yer alan Fonksiyon Tablosu başlığından öğrenebilirsiniz.)*

1. PostgreSQL Desteğinin Etkileştirilmesi

PHP kurulumunda `./configure` işlemine vereceğiniz bir `--with-pgsql` parametresi ile PHP'nin PostgreSQL desteğini etkinleştirebilirsiniz.

> *`--with-pgsql` parametresine herhangi bir değer girmediğiniz taktirde öntanımlı olarak, `/usr/local/pgsql` dizini PostgreSQL'in kurulu olduğu dizin olarak kullanılacaktır.*

Eğer PHP kurulumunuz hazır derlenmiş paketler kullanan bir işletim sisteminin paket yöneticisi tarafından gerçekleştirilmişse, PostgreSQL desteği modül halinde eklenmiş olmalıdır. Bu nedenle, eğer PostgreSQL desteği, kurduğunuz PHP paketleri sonucunda hala mevcut değilse, PHP PostgreSQL modüllerinin bulunduğu paketi de kurmanız gerekmektedir.

> *`RPM` ve `DEB` paketlerinin kullanan Linux dağıtımlarında PHP PostgreSQL modüllerinin bulunduğu paket, genellikle `php-pgsql` adı ile geçmektedir.*

Bunun yanında, `php.ini` dosyasındaki ilgili satırın başında bulunan `;' (noktalı vürgül) işaretini kaldırıp, `pgsql` modülü etkin hale getirilmelidir.

Sisteminizde o an kurulu PHP sürümündeki PostgreSQL desteğinin etkin olup olmadığını öğrenmek için aşağıdaki yöntemlerden birini kullanabilirsiniz:

- Kurulum esnasında `--with-pgsql` parametresinin kullanılıp kullanılmadığını öğrenmek için `phpinfo()` fonksiyonu sonucu çıkacak olan tablo listesinde *Configure Command* başlıklı satıra (ya da mevcutsa pgsql başlıklı bölüme) bakabilirsiniz.
- `get_loaded_extensions()` fonksiyonunu kullanarak o an etkin halde olan eklentileri listeleyebilirsiniz. (Komut satırında `php -m` komutu `get_loaded_extensions()` ile aynı işleve sahiptir.) Veyahut sadece tek bir modülün etkinleştirilip etkinleştirilmediğini öğrenmek için `extension_loaded()` fonksiyonunu da kullanabilirsiniz.

> *`php.ini` dosyasında PostgreSQL ile ilgili ayarlamalar için ekler bölümünde yer alacak olan `php.ini` dosyasının yapılandırılması başlıklı alt bölümü okuyabilirsiniz.*

Unutulmaması gereken bir diğer nokta ise, PHP tarafından sağlanan PostgreSQL desteğinin yaptığı işin aslında sisteminizde kurulu olan libpq kütüphane fonksiyonları ile programcı arasında bir ara katman oluşturmaktan ibaret olduğudur. Hal böyle olunca, PHP tarafından arayüz sağlanan fonksiyonlar, kurulu libpq kütüphaneniz tarafından desteklenmiyorsa, desteklenmeyen bu fonksiyonların PHP tarafından da kullanımı olanaksızlaşır. Bu nedenle, ileriki başlıklarda tanıtılacak olan fonksiyonların hepsi kullandığınız sistemde çalışmayabilir.

> *Bu alt bölümden sonraki adımlarda sisteminizde kullandığınız PHP kurulumunun PostgreSQL desteğine sahip olduğu varsayılıp, buna göre hareket edilecektir.*

2. Tanıtım Uygulaması

PHP tarafından sağlanan PostgreSQL fonksiyonlarının tanıtımına geçmeden önce, okuyucuya genel bir fikir vermesi açısından, PHP ile PostgreSQL veritabanı üzerinde nasıl işlem yapılabileceğini kabataslak da olsa daha iyi ifade etmesi açısından basit bir tanıtım uygulaması ile giriş yapacağız.

Yazacağımız uygulamada bir markette yer alan ürünlerin satış kayıtları ile birlikte market raflarındaki yerleşimleri veritabanındaki tablolar üzerinde tutulacak. Ardından yazacağımız PHP betiği ile belirli bir zamanda satılan bir ürünün bulunduğu rafın yanındaki raflarda yer alan diğer ürünlerin de bu satışlardan etkilenip etkilenmediği kontrol edilecek.

> *Bunun ile aynı katagoride ortalama bir veri madenciliği uygulamasında bile çok daha karmaşık tablo ilişkileri, sorgular ve bu sorgulamalar esnasında başarım kazancı sağlayacak değişik algoritmalar kullanılmaktadır. Burada konunun izahı esnasında yardımcı olabileceğini düşündüğümüz çok basit bir örnekleme ele alınmıştır.*

İlk olarak, analizin gerçekleşeceği tabloları oluşturalım.

```
CREATE TABLE urunler
(
  id      serial PRIMARY KEY,
  marka   varchar NOT NULL,
  urun    varchar NOT NULL
);
CREATE UNIQUE INDEX urunler_id_idx ON urunler (id);
CREATE UNIQUE INDEX urunler_marka_urun_idx ON urunler (marka, urun);
```

Ürünler tablosunda tutulacak herbir ürünün kendisine ait bir `id` değeri ve markası olacak. Marka ve ürün alanlarında oluşturduğumuz `UNIQUE INDEX` sayesinde, aynı marka ve ürüne sahip bir malın, tabloya iki kez eklenme olasılığından sakınmış olacağız. `id` alanlarını oluştururken ise basit bir `SEQUENCE` tipinden yararlanıyoruz.

```
CREATE TABLE satislar
(
  tarih   timestamp without time zone DEFAULT CURRENT_TIMESTAMP NOT NULL ,
  id      bigint REFERENCES urunler (id)
);
```

Satışlar tablosunda, kasalarda gerçekleşen ürün satışlarının tarihlenmiş kayıtları tutulacak. (Böyle bir tablonun ne kadar hızla büyüyebileceği, bu konunun adının neden veri madenciliği olduğu hakkında belirgin bir cevap sunmakta.)

```
CREATE TABLE yerlesim
(
```

```
  tarih   timestamp without time zone DEFAULT CURRENT_TIMESTAMP NOT NULL,
  id      bigint REFERENCES urunler (id),
  dolap   integer NOT NULL
  raf     integer NOT NULL
);
CREATE UNIQUE INDEX yerlesim_raf_bolme_idx ON yerlesim (dolap, raf);
```

Yer alan ürünlerin marketteki raflar üzerinde yerleşimleri burada iki ayrı alt katagoriye indirgenmiştir: dolap ve raf. Her dolap, belirli sayıda raftan oluşacak şekilde düşünülmüştür.

Şimdi bu tablolar eldesiyle, yapacağımız işlemin adımlarını şu şekilde özetleyebiliriz:

1. Son bir ay içinde raflarda en çok satılan ürünlerin listesi çıkarılacak.
2. En çok satılan ürünlerin bulunduğu dolapta ve bu dolabın yanındaki diğer dolaplardaki ürünlerin listesi çıkarılacak.
3. Listesi elde edilen bu ürünlerin son bir ay içindeki satışları ile bir önceki ay içindeki satışlarının karşılaştırılması yapılacak.

Yardımcı bileşenleri çıkardığımıza göre, artık programımızı yazabiliriz:

```
$bag = pg_connect("dbname=test")
       or die("Veritabanına bağlanırken bir hata oluştu!");

$zaman_araligi = "'1 month'::interval"; /* Çok satanlara bakacağımız zaman aralığı. */
$cok_satan_sayisi = 10; /* En çok kaç tane çok satan ürün listeleyeceğiz? */

/*
 * Kitap satışlarını, en çok satandan en aza doğru, marka, ürün,
 * dolap, raf satış sayısı bilgileri ile birlikte listeliyoruz.
 */
$komut = <<<EOF
SELECT id, u.marka, u.urun, y.dolap, y.raf, count(s.id) AS satis_sayisi
  FROM urunler u
  LEFT JOIN satislar s USING (id)
  LEFT JOIN yerlesim y USING (id)
  WHERE s.tarih > now() - $zaman_araligi
  GROUP BY id, u.marka, u.urun, y.dolap, y.raf
  ORDER BY satis_sayisi DESC
  LIMIT $cok_satan_sayisi
EOF;
$sorgu1 = pg_query($bag, $komut)
          or die("Sorgulama esnasında bir hata oluştu: ".pg_last_error());

print <<<EOF
<table border="1">
<tr style="background: #EDEDED">
    <td>ID</td>
    <td>Marka</td>
    <td>Ürün</td>
    <td>Dolap</td>
    <td>Raf</td>
    <td>Önceki Satış Sayısı</td>
    <td>Satış Sayısı</td>
</tr>
EOF;

while ($urun = pg_fetch_array($sorgu1))
{
    print <<< EOF
<tr style="background: #EFEFEF">
    <td>{$urun["id"]}</td>
    <td>{$urun["marka"]}</td>
    <td>{$urun["urun"]}</td>
    <td>{$urun["dolap"]}</td>
    <td>{$urun["raf"]}</td>
    <td>-</td>
    <td>{$urun["satis_sayisi"]}</td>
```

```
</tr>
EOF;

    /*
     * Şu an (foreach döngüsü ile) bakmakta olduğumuz ürünün yer aldığı dolabın
     * yanındaki (yani dolap numarası kendisininkinden bir fazla ve bir eksik
     * numaralı) dolaplarda yer alan ürünlerin, o dolaba yerleştirilme tarihinden
     * önceki ve sonraki satış sayılarını buluyoruz.
     * Ürünün önceki satış sayısını bulurken şöyle bir filtreleme uyguluyoruz:
     * Satış tarihi dolaba yerleştirilme tarihinden küçük; ürün, dolaba
     * yerleştirilme tarihi ile şu ana kadar olan zaman farkı kadar eski olacak.
     * Bir örnek vermek gerekirse, eğer ürün o dolapta 1 ayden beri yer alıyorsa,
     * önceki satış sayısını hesaplarken YerlestirilmeTarihi - 1ay zaman aralığını
     * kullanıyoruz.
     */
    $onceki  = $urun["dolap"] - 1;
    $sonraki = $urun["dolap"] + 1;
    $komut   = <<<EOF
SELECT u.id, u.marka, u.urun, y.dolap, y.raf
  (SELECT count(id) FROM satislar WHERE id = y.id
    AND tarih < y.tarih
    AND tarih > (y.tarih - (now() - y.tarih))) AS onceki_satis_sayisi,
  (SELECT count(id) FROM satislar WHERE id = y.id
    AND tarih >= y.tarih) AS satis_sayisi
  FROM yerlesim y
  LEFT JOIN urunler u USING (id)
  WHERE y.dolap = $onceki OR y.dolap = $sonraki
  ORDER BY satis_sayisi DESC
EOF;
    $sorgu2 = pg_query($bag, $komut)
              or die("Sorgulama esnasında bir hata oluştu: ".pg_last_error());

    while ($yan_raftaki = pg_fetch_array($sorgu2))
        print <<<EOF
<tr>
    <td>{$yan_raftaki["id"]}</td>
    <td>{$yan_raftaki["marka"]}</td>
    <td>{$yan_raftaki["urun"]}</td>
    <td>{$yan_raftaki["dolap"]}</td>
    <td>{$yan_raftaki["raf"]}</td>
    <td>{$yan_raftaki["onceki_satis_sayisi"]}</td>
    <td>{$yan_raftaki["satis_sayisi"]}</td>
</tr>
EOF;
}

print "\n\n</table>\n";
```

PHP'nin kendi veritabanı arayüzlerini oluştururken izlediği standart biçim duyarlılığından ötürü, sağladığı veritabanı fonksiyonlarının çok büyük bir kısmı birbirine benzerdir ve bir veritabanı için sağlanan bir fonksiyonun, çok benzer bir isimle başka bir veritabanı için de desteklenmesi olasılığı oldukça fazladır. Bu sebepten ötürü, PHP ile daha önceden herhangi bir veritabanı ile sorgulama yapmış biri için yukarıdaki kod parçası oldukça tanıdık gelecektir. Yine de yukarıda kullandığımız veritabanı fonksiyonlarını kısa bir şekilde açıklamamız gerekirse:

- `pg_connect()` fonksiyonu, belirtilen PostgreSQL veritabanı ile bağlantı kurulmasını sağlar.
- `pg_query()` fonksiyonu, parametre olarak aldığı komutun sorgusunu geçerli ya da belirtilen veritabanı bağlantısı üzerinden gerçekleştirmek için kullanılır.
- `pg_last_error()` fonksiyonu, parametre olarak aldığı ya da geçerli veritabanı bağlantısı üzerinde gerçekleşmiş en son hataya ait mesajı kendisini çağırana döndürür.
- `pg_fetch_array()` fonksiyonu, parametre olarak aldığı sonucun herhangi bir veri içermesi durumunda, ilgili verinin her çağrılışında bir satırını kendisini çağırana dizi (`array`) olarak döndürecektir.

Genelde veritabanı ile etkileşime geçen PHP betiklerinde yukarıdakine benzer bir yapı izlenmektedir: Veritabanı bağlantısı gerçekleştirildikten sonra gereken sorgu tümceleri oluşturulup ilgili sorgulamalar ile istenilen sonuçlar karşılaştırılıp bir sonraki adıma karar verilir. Fonksiyon ve bu fonksiyonların aldıkları parametrelerin sözdizim yapıları da sabit olduğundan, yazılan program ister istemez genel bir biçim içinde ortaya konur. Fakat bu ve benzeri detaylara kitabın ilerki bölümlerinde değineceğiz.

Yazdığımız PHP betiğini çalıştırdığımızda karşımıza (bizim bu örneği uygularken oluşturduğumuz tablo verilerine göre) şuna benzer bir tablo çıkacak:

ID	Marka	Ürün	Dolap	Raf	Önceki Satış Sayısı	Satış Sayısı
4	Kabalcı Yayınevi	Gödel, Escher, Bach	2	3	-	16
2	Çikita	Muz	1	4	5	8
3	Casio	Algebra GX 2.0	3	2	5	0
5	Ülker	İçim Süt	1	3	0	0
2	Çikita	Muz	1	4	-	13
4	Kabalcı Yayınevi	Gödel, Escher, Bach	2	3	16	0
3	Casio	Algebra GX 2.0	3	2	-	5
1	Ülker	Sütlü Çikolata	4	1	5	0
4	Kabalcı Yayınevi	Gödel, Escher, Bach	2	3	16	0
1	Ülker	Sütlü Çikolata	4	1	-	5
3	Casio	Algebra GX 2.0	3	2	5	0

Bu tabloya bakarak, sanal marketimizdeki satış istatistiğinden çok, yazarın o an ki psiklojisini uygun veri madenciliği teknikleri ile açıklığa kavuşturmayı okuyucunun kişisel beceri ve ilgi alanı doğrultusunda ayrı bir ödev olarak bırakıyoruz.

3. Bağlantı Kurulumu

Bu bölümde veritabanı ile nasıl bağlantı kurulacağı hakkında bahsedip, bunun için PHP tarafından sağlanan fonksiyonlar üzerinde durmaya çalışacağız.

```
resource pg_connect (   string baglanti_secenekleri
                      [,int    baglanti_tipi] )
```

`pg_connect()` fonksiyonu, parametre olarak aldığı bağlantı seçeneklerini kullanarak veritabanına ile bağlantı kurulmasını sağlar. Başarılı bir bağlantı sonucunda, kendisini çağırana (diğer PostgreSQL fonksiyonları tarafından kullanılacak olan) veritabanı kaynağını döndürecektir. Herhangi bir hata durumunda `FALSE` değeri döndürür.

> *Fonksiyon tarafından kullanılan bağlantı seçenekleri, libpq kütüphanesi tarafından sağlanan bağlantı fonksiyonlarınca (`PQconnectdb()`, ...) kullanılan bağlantı seçenekleri ile aynı yapıya (ve kısıtlamalara) sahiptir.*

Bağlantı tipi `PQSQL_CONNECT_FORCE_NEW` olarak belirtilmediği sürece, aynı bağlantı seçenekleri ile ikinci bir `pg_connect()` çağrısı yapıldığında, fonksiyon varolan bağlantının kaynağını döndürecektir. Bunun dışında, `PQSQL_CONNECT_FORCE_NEW` değeri bağlantı tipi olarak fonksiyona parametre ile belirtildiğinde, ikinci bir `pg_connect()` çağrısında yeni bir veritabanı bağlantısı döndürülecektir.

> *Yapılabilecek en fazla bağlantı sayısını `php.ini` dosyasındaki `pgsql.max_links` ile ayarlayabilirsiniz. (-1 değeri herhangi bir sınır olmadığı anlamına gelir.)*

```
resource pg_pconnect (   string baglanti_secenkeleri
                       [,int    baglanti_tipi] )
```

pg_pconnect() fonksiyonu, pg_connect() fonksiyonu ile aynı özelliklere (ve kısıtlamalara) sahip olup, veritabanı ile kalıcı[16] (*persistent*) bir bağlantı kurulmasını sağlar.

> *Veritabanı ile kalıcı bir bağlantı kurabilmek için php.ini dosyasındaki pgsql.allow_persistent değişkeni açık durumda bulunmalıdır. pgsql.max_persistent INI değişkeni ile de veritabanı ile kurulabilecek en fazla kalıcı bağlantı sayısını ayarlayabilirsiniz.*

4. Bağlantı Üzerinde İşlemler

Veritabanı ile kurulan başarılı bir bağlantı sonucunda, dönen bağlantı kaynağını kullanarak bağlantı ve sunucu hakkından bilgi edinip, bağlantı üzerinde değişiklikler yapmak için bu bölümde tanıtılacak olan fonksiyonları kullanabilirsiniz.

4.A. Bağlantı Hakkında Bilgi Edinme

Varolan bir bağlantı hakkında bilgi edinmek için bu bölümde tanıtılacak olan fonksiyonları kullanabilirsiniz.

```
int pg_connection_status ( resource baglanti )
```

pg_connection_status() fonksiyonu, parametre olarak aldığı bağlantı kaynağının durumunu kendisini çağırana döndürür. libpq kütüphanesi tarafından sağlanan PQstatus() fonksiyonu için bir ara katman oluşturuyor olsa da, döneceği değerler yalnızca (başarılı bir bağlantı durumunda) PGSQL_CONNECTION_OK (0 değerine sahiptir) ve (bağlantının sorunlu olduğu bir durumda da) PGSQL_CONNECTION_BAD değerlerinden ibaret olacaktır.

> *pg_connection_status() fonksiyonunun, PQstatus() fonksiyonundan dönecek olan cevabı kendisini çağırana döndüreceği düşünüldüğünde, neden sadece 2 tane durum değeri döndürdüğü, bağlantının senkron olması ile cevaplanabilir. Bu yüzden ara safhalardaki diğer PQstatus() değerleri gözlemlenemeyecektir.*

```
string pg_dbname ( [resource baglanti] )
```

pg_dbname() fonksiyonu, geçerli bağlantının kurulu olduğu veritabanı adını kendisini çağırana döndürür. Geçerli bir bağlantının olmadığı durumda fonksiyon FALSE değeri döndürecektir.

```
string pg_host ( [resource baglanti] )
```

pg_host() fonksiyonu, parametre olarak aldığı bağlantının bağlı olduğu sunucu adını, bağlantının geçerli olmadığı bir durumda ise FALSE değerini kendisini çağırana döndürür.

```
string pg_options ( [resource baglanti] )
```

pg_options() fonksiyonu, belirtilen bağlantıda kullanılan bağlantı seçenekleri arasındaki options değişkeninin değerini kendisini çağırana döndürür. Geçerli bir bağlantının olmadığı durumda fonksiyon FALSE değeri döndürecektir.

16 Kalıcı bağlantıların ne olup, nasıl bir durumunda kullanılması gerektiği hakkında daha ayrıntılı bilgi için sık karşılaşılan sorunlar kısmındaki ilgili başlığa göz atabilirsiniz.

```
int pg_port ( [resource baglanti] )
```

pg_port() fonksiyonu, parametre olarak aldığı bağlantıda kullanılan port değişkenin değerini kendisini çağırana döndürür. Fonksiyon, bağlantının geçerli olmadığı bir durumda FALSE değeri döndürecektir.

```
string pg_client_encoding ( [resource baglanti] )
```

pg_client_encoding() fonksiyonu, istemci tarafından kullanılan karakter kodlamasının kendisini çağırana karakter katarı halinde döndürür. Geçerli bağlantının bulunamaması durumunda FALSE değeri dönecektir.

```
int pg_transaction_status ( resource baglanti )
```

pg_transaction_status() fonksiyonu, parametre olarak aldığı veritabanı bağlantısında o anda gerçekleşmekte olan *transaction* durumunu kendisini çağırana döndürecektir. Döndüreceği değerleri şu şekilde sıralayabiliriz:

```
PGSQL_TRANSACTION_IDLE      PGSQL_TRANSACTION_INTRANS   PGSQL_TRANSACTION_UNKNOWN
PGSQL_TRANSACTION_ACTIVE    PGSQL_TRANSACTION_INERROR
```

```
string pg_last_error ( [resource baglanti] )
```

pg_last_error() fonksiyonu, geçerli ya da belirtilen bağlantı üzerinde gerçekleşen en son hata mesajını kendisini çağırana döndürecektir.

> *pg_last_error() fonksiyonunun, libpq kütüphanesindeki PQerrorMessage() ile aynı özelliklere (ve kısıtlamalara) sahip olduğundan, bağlantı üzerinde aynı anda birden fazla hata gerçekleşmesi durumunda döndüreceği hata mesajının sabit kalmayacağını hatırlatırız.*

```
string pg_last_notice ( resource baglanti )
```

pg_last_notice() fonksiyonu, parametre olarak aldığı veritabanı bağlantısı üzerinde oluşan uyarı mesajlarını kendisini çağırana döndürecektir.

> *4.0.3 ve daha düşük PHP sürümlerinde pg_last_notice() fonksiyonu ile ilgili önemli bir açık bulunduğundan, pg_last_notice() fonksiyonunu kullanacak geliştiricilerin PHP 4.1.0 ya da daha üst bir sürümü tercih etmeleri tavsiye olunur.*

PHP 4.3.0 sürümünden itibaren php.ini dosyasındaki pgsql.ignore_notice seçeneğini etkinleştirerek, uyarı mesajlarını kapatabilirsiniz. Benzer şekilde, 4.3.0 sürümünden itibaren pgsq.log_notice seçeneği ile uyarı mesajlarının kaydının tutulmasını sağlayabilirsiniz. Fakat aşikardır ki, pgsql.ignore_notice seçeneği etkin olduğu sürece pgsql.log_notice işlev göstermeyecektir.

4.B. Sunucu Hakkında Bilgi Edinme

Varolan bir bağlantı kullanılarak, diğer uçtaki PostgreSQL veritabanı sunucusu hakkında bilgi edinmek için bu bölümde tanıtılacak olan fonksiyonları kullanabilirsiniz.

```
bool pg_connection_busy ( [resource baglanti] )
```

pg_connection_busy() fonksiyonu, geçerli bağlantının kurulu olduğu sunucu eğer o an ilgili istemci tarafından gönderilen herhangi bir işlem üzerinde meşgul ise TRUE, aksi halde FALSE değerini döndürür.

```
string pg_parameter_status ( [resource baglanti,]string parametre )
```

pg_parameter_status() fonksiyonu, bağlantının gerçekleştiği sunucunun belirtilen parametresinin değerini, bağlantının geçerli olmadığı ya da belirtilen parametrenin bulunmadığı bir durumda FALSE değerini kendisini çağırana döndürecektir.

Belirtebileceğiniz sunucu parametreleri: server_version, server_encoding[17], client_encoding, is_superuser, session_authorization, DateStyle, TimeZone* ve integer_datetimes* şeklindedir.

```
array pg_version ( [resource baglanti] )
```

pg_version() fonksiyonu, geçerli bağlantının diğer ucundaki sunucu versiyon bilgilerini kendisini çağırana dizi halinde, herhangi bir hata durumunda ise FALSE değerini döndürecektir.

Dönen dizi *client* (istemci sürüm numarası), *protocol* (kullanılan protokol sürümü) ve *server* (sunucu sürüm numarası) anahtar değişkenlerini içerecektir.

```
int pg_get_pid ( resource baglanti )
```

pg_get_pid() fonksiyonu, bağlantının sağlandığı sunucunun o anki kurulu bağlantıyı sağlayan işleminin PID değerini kendisini çağırana döndürecektir. Fonksiyon, bağlantının geçerli olmadığı bir durumda FALSE değeri döndürür.

4.C. Kurulu Bağlantı Üzerinde İşlemler

Herhangi bir veritabanı bağlantısını kullanarak, geçerli bağlantılar üzerinde çeşitli değişiklikler yapmak için bu bölümde incelenecek olan fonksiyonları kullanabilirsiniz.

```
int pg_set_client_encoding ( [resource baglanti,] string kodlama )
```

pg_set_client_encoding() fonksiyonu, geçerli bağlantının karakter kodlamasını belirtilen değer olarak değiştirir. Başarılı olması durumunda 0, aksi halde -1 değeri döndürecektir.

```
int pg_set_error_verbosity ( [resource baglanti,] int duyarlilik )
```

pg_set_error_verbosity() fonksiyonu, geçerli (ya da parametre olarak girilen) bağlantının hata mesajı duyarlılığını belirtilen duyarlılığa ayarlar. Fonksiyon ile birlikte kullanabilecek duyarlılık dereceleri şu şekilde:

PGSQL_ERRORS_TERSE PGSQL_ERRORS_DEFAULT PGSQL_ERRORS_VERBOSE

```
bool pg_close ( [resource baglanti] )
```

pg_close() fonksiyonu, geçerli kalıcı olmayan bağlantının kapatılmasını sağlar. Başarılı olduğu durumda TRUE, aksi halde FALSE değeri döndürecektir.

> *Program işleyişinin sonuna gelindiğinde, kalıcı olmayan bağlantılar PHP tarafından otomatik olarak kapatılacaktır.*

```
bool pg_connection_reset ( [resource baglanti] )
```

pg_connection_reset() fonksiyonu, geçerli bağlantının koparılıp, eski bağlantı seçenekleri ile tekrardan kurulmasını sağlar.

17 PostgreSQL 8.0 ve sonrası sürümlerce desteklenmektedir.

pg_connection_reset() fonksiyonu, daha çok herhangi bir bağlantı kopması durumunda, hatayı tespit etmek için kullanılır. Bu yolla bağlantı tekrar kurulmaya çalışılarak, tekrar başarısız olunması durumunda gerekli hata mesajı elde edilir.

5. Sorgu İşletimi

Bu bölümde incelenecek olan fonksiyonları kullanarak, varolan bir veritabanı bağlantısı üzerinde ilgili sorgulamalarımızı gerçekleştirebiliriz.

5.A. Senkron Sorgu İşletimi

Geçerli bir bağlantıyı kullanarak, veritabanı üzerinde sorgulamalarınızı senkron olarak gerçekleştirmek için kullanabileceğiniz fonksiyonlar hakkında bu bölümde bahsedilecektir.

```
resource pg_query ( [resource baglanti,] string komut )
```

pg_query() komutu ile geçerli veritabanı bağlantısı üzerinden, fonksiyona parametre olarak gireceğiniz *SQL* sorgulamalarınızı gerçekleştirebilirsiniz.

Herhangi bir hata oluşması durumunda fonksiyon FALSE değeri döndürecektir. Dönecek olan sonucun durumunu öğrenmek için pg_result_status() fonksiyonunu kullanabilirsiniz.

```
$sorgu = pg_query($baglanti, $sql_komutu)
        or die ("Sorgu gerçekleştiriminde bir hata oluştu: ".pg_last_error($baglanti));
```

```
resource pg_query_params ( [resource baglanti,]
                           string   komut,
                           array    parametreler )
```

pg_query_params() fonksiyonu, belirtilen parametre değerlerine sahip sorgunun geçerli bağlantı üzerinde çalıştırılmasını sağlar. Fonksiyon, herhangi bir hata durumunda FALSE değeri dönecektir.

```
/* Parametrelerin etrafında tırnak kullanılmadığına dikkat ediniz. */
$sql_komutu = "SELECT adet FROM stok WHERE urun_adi = $1 AND urun_tipi = $2";

$parametreler = array("Çikolata", "Bitter");
$sorgu = pg_query_params($baglanti, $sql_komutu, $parametreler);
```

```
resource pg_prepare ( [resource baglanti,]
                      string   sorgu_adi,
                      string   komutu )
```

pg_prepare() fonksiyonu, geçerli bağlantının kurulu olduğu sunucu üzerinde, belirtilen sorgu adı ve komut doğrultusunda, daha sonradan çağrılmak üzere bekleyen hazır bir sorgu oluşturulmasını sağlar. Fonksiyon, hata durumunda FALSE değeri döndürecektir.

```
resource pg_execute ( [resource baglanti,]
                      string   sorgu_adi,
                      array    parametreler )
```

pg_execute() fonksiyonu, geçerli bağlantının kurulu olduğu sunucu üzerinde belirtilen ada sahip hazır sorgunun gösterilen parametre değerleri ile çalıştırılmasını sağlar. Fonksiyon herhangi bir hata durumunda kendisini çağırana FALSE değeri döndürecektir.

```
/* Daha sonradan kullanılacak olan sorgu hazırlanıyor. */
$sorgu_adi = “urun”;
pg_prepare($baglanti, $sorgu_adi, “SELECT urun FROM urunler WHERE id = $1”);

/* Belirtilen bir array içinde yer alan ID değerlerine sahip ürünler listeleniyor. */
foreach ($id_listesi as $id)
{
    $sorgu = pg_execute($baglanti, $sorgu_adi, array($id));
    ...
}
```

6. Asenkron Sorgu İşletimi

Veritabanı üzerinde asenkron olarak sorgu işletimi için bu bölümde tanıtılacak olan fonksiyonları kullanabilirsiniz.

Bölüm altında tanıtılacak olan fonksiyonlara geçmeden önce, okuyucunun öncellikle (asenkron sorgu işletiminin ne gibi özelliklere sahip olduğu hakkında daha ayrıntılı bilgi için) libpq kütüphanesinin tanıtımında yer alan ilgili başlık altındaki bölümü okumasını önemle tavsiye ederiz. Aksi halde çalışma mantığı yanlış anlaşılmış bir asenkron sorgu işlemi, programda istenmeyen sonuçlara yol açabileceği gibi kodun gereksiz yere uzamasına da sebebiyet verecektir.

```
bool pg_send_query ( resource baglanti, string komut )
```

`pg_send_query()` fonksiyonu, geçerli bağlantı üzerinden parametre olarak girilen komutun (sonucu beklemeksizin) sunucuya gönderimini sağlar. Bu sayede program akışına, cevabın gelmesini beklemeye gerek kalmadan devam edilebilinir. Fonksiyon herhangi bir hata durumunda `FALSE` değerini döndürecektir.

```
bool pg_send_query_params ( resource baglanti,
                            string   komut,
                            array    parametreler )
```

`pg_send_query_params()` fonksiyonu, `pg_query_params()` fonksiyonunun asenkron işleyişe sahip olan türevidir. Dolayısıyla, fonksiyonun aldığı parametreler `pg_query_params()` fonksiyonundaki eşleri ile aynıdır.

```
bool pg_send_prepare ( resource baglanti,
                       string   sorgu_adi,
                       string   komut )
```

`pg_send_prepare()` fonksiyonu, `pg_prepare()` fonksiyonunun asenkron işleyişe sahip türevidir.

```
bool pg_send_execute ( resource baglanti,
                       string   ifade_ismi,
                       array    parametreler )
```

`pg_send_execute()` fonksiyonu, `pg_send_prepare()` fonksiyonu ya da diğer bir `PREPARE` sorgusu ile sunucu tarafında oluşturulmuş olan bir hazır sorgunun (cevabını beklemeden) çağırılmasını sağlar.

```
resource pg_get_result ( resource baglanti )
```

`pg_get_result()` fonksiyonu, `pg_send_query()`, `pg_send_params()` ya da `pg_send_execute()` ile sunucu tarafında asenkron olarak işletilmekte olan sorgunun sonucunu almak için kullanılır. Birden fazla sorgunun işletilmesi sonucu oluşacak sonuçların cevabını alırken ardarda (sunucu üzerinde sonuç kalmayana kadar)

`pg_get_result()` fonksiyonunu çağırmanız gerekir. Şu da unutulmamalıdır ki, *FIFO* (*First In, First Out*/İlk Giren, İlk Çıkar) mantığıyla, herbir sorgunun sonucunu teker teker sırayla almanız gerekmektedir. Sunucu üzerinde herhangi bir sonuç kalmadığında, fonksiyon `FALSE` değeri döndürecektir.

Asenkron sorgu işletiminde, sorgunun işlenmesi karşı tarafta bittikten sonra sonuç `pg_get_result()` ile alınabilir duruma gelecektir. Genelde bunun için, sokete veri ulaşıp ulaşmadığına dair haber almada kullanılan `select()` ailesi fonksiyonlarından biri ile bağlantı soketi dinlenir. Fakat PHP bize bağlantı soketine herhangi bir ulaşım hakkı tanımadığından, böyle bir özellik henüz sağlanmamaktadır. Bunun için `pg_connection_busy()` fonksiyonu ile - `usleep()` ya da `sleep()` çağrıları ile arada yeterince ufak zaman aralıkları bırakılara - döngüye girilir. Bu pek de hoş olmayan bir yöntem olmasına karşın, şu an için önümüzdeki tek çözüm yoludur.

```
if (pg_connection_busy($baglanti))
{
    /*
     * Yeni bir sorgu gönderimi, bir önceki sorgunun sonucunu bellekten
     * temizleyeceğinden, ilk öne o sorgunun bitmesini bekliyoruz.
     */
}

/* Asenkron sorgu gönderiminde bulunuyoruz. */
$gonderim = pg_send_query($baglanti, "SELECT ...");
if (!$gonderim)
    /*
     * Sorgu gönderimi esnasında hata oluştu.
     * (pg_last_error() ile hata hakkında bilgi sahibi olabiliriz.
     */

/*
 * Sorgunun sonucunu beklemeye gerek kalmadan, o süre zarfı
 * içinde istediğimiz diğer işlemlerimizi gerçekleştiriyoruz.
 */

/*
 * Sunucunun işini bitirip bitirmediğine bakıyoruz. Eğer
 * sorgu hala devam ediyorsa, onun sonlanmasını bekliyoruz.
 */
while (!pg_connection_busy($baglanti))
    /*
     * İsteğe bağlı olarak programcının kendi seçeceği bir
     * yöntem ile belirli bir süre beklenecek burada.
     */

$sonuc = pg_get_result($baglanti);
if (!$sonuc)
    /* Sorgu sonucunu alınırken bir hata oluştu. */

/* Sorgu sonucu başarı ile alındı. */
```

6.A. COPY Komutu Kullanımı

Veritabanına yapılacak yüklü veri transferlerinde `COPY`[18] komutu büyük hız artışlarına olanak sağlamaktadır. Komutun kullanımında oluşturacağınız kendi `COPY` içeren sorgularınıza ek olarak, `COPY` komutuna yardımcı olması için kütüphane tarafından bir kaç fonksiyon sunulmaktadır. Bu bölümde bu fonksiyonlar üzerinde duracağız.

18 `COPY` komutu hakkında ayrıntılı bilgi için kitabın ilgili bölümüne bakabilirsiniz.

```
array pg_copy_to (   resource baglanti,
                     string   tablo_adi
                   [, string   ayrac
                    [,string   null_alan]] )
```

pg_copy_to() fonksiyonu, geçerli bağlantı üzerinden belirtilen tablodaki herbir satırı dizi içine atıp, kendisini çağırana bu diziyi döndürür. Herhangi bir hata durumunda fonksiyon FALSE değeri döndürecektir.

Döndürülen dizide her bir satır ilgili tablo satırına karşılık gelecek olurken, tablo alanları belirtilen ayraç ile ayırılacaktır. (Öntanımlı ayraç değeri \t karakteridir.) NULL alanlarının belirtilmesi için herhangi bir parametre girilmediği taktirde NULL değeri olan \N kullanılacaktır. Eğer NULL alanların da boş karakter katarı ile bir tutulmasını istiyorsanız, ayraç olarak boş bir karakter katarı da (“” gibi) kullanabilirsiniz.

```
bool pg_copy_from (   resource baglanti,
                      string   tablo_adi,
                      array    deger_dizisi
                    [, string   ayrac
                     [,string   null_alan]] )
```

pg_copy_from() fonksiyonu, parametre olarak aldığı dizi içindeki satırları, veritabanındaki ilgili tabloya eklemek için kullanılır.

Parametre olarak geçilecek olan ayraç ve NULL alanı değerlerinin kullanımı pg_copy_to() fonksiyonundaki eşleri ile aynıdır.

```
bool pg_put_line ( [resource baglanti,] string veri )
```

pg_put_line() fonksiyonu, geçerli bağlantı üzerinden parametre olarak aldığı verinin COPY_IN durumundaki veritabanına aktarılmasını sağlar.

```
COPY ornektablo FROM stdin;
```

şeklinde standart girdiden veri okunacak şekilde aktarıma başlandığı zaman, her bir satırdan sonra yeni satır karakteri yer almalı. Fakat yer alacak yeni satır karakterinin her gönderimde tek bir kez kullanılma gibi bir zorunluluğu yoktur. İsteğe bağlı olarak, bir tek PQputline() çağrısında aynı anda birden fazla satır gönderiminde bulunulabilinir.

pg_put_line() fonksiyonu ile başlanan veri gönderiminde aktarılacak olan veri bittiğinde, en son olarak sunucuya aktarımın bittiğini bildirmek üzere bir \.[19] karakteri gönderildikten sonra COPY işlemi sonlandırılmak için pg_end_copy() fonksiyonu çağırılır.

```
bool pg_end_copy ( [resource baglanti] )
```

pg_end_copy() fonksiyonunu, COPY komutu ile başlatılan herhangi bir veri aktarımının sonunun geldiğini bildirmek için kullanabilirsiniz.

```
$tablo_adi = “ornek_tablo”;

/*
 * pg_copy_from() kullanılarak, belirtilen tabloya
 * dizi içindeki değerler aktarılıyor.
 */
pg_copy_from($baglanti, $tablo_adi,
             array(“11\t12\t13\n”,       /* Burada \n kullanımı isteğe bağlı. */
                   “21\t22\t23”,
                   “31\t32\t33”,
                   “41\t\\N\t43”         /* Örnek NULL değer giriyoruz. */
```

19 3.0 protokolünün kullanılmaya başlanmasıyla birlikte veri aktarımı bittiğinde pg_end_copy() fonksiyonundan önce \. karakteri gönderimi isteğe bağlı bir hale gelmiş olup kullanım zorunluluğu ortadan kalkmıştır.

```
            ));

/* pg_put_line() kullanılarak gönderim gerçekleştiriliyor. */
pg_query($baglanti, "COPY $tablo_adi FROM stdin");
pg_put_line($baglanti, "51\t52\t53\n");  /* Bir seferde tek satır. */
pg_put_line($baglanti, "61\t62\t63\n".   /* Tek seferde iki satır. */
                       "71\t72\t73\n");
pg_put_line($baglanti, "81\t\\N\t83\n"); /* Örnek NULL değeri. */
pg_put_line($baglanti, "\.\n");          /* Sonlandırıcı karakter. */
pg_end_copy($baglanti);                  /* Veri transferini sonlandırıyoruz. */

/* Tüm gönderdiğimiz değerleri pg_copy_to() ile alıyoruz. */
$dizi = pg_copy_to($baglanti, $tablo_adi);

/*
 * print_r($dizi) çıktısı:
 *
 * Array
 * (
 *     [0] => 11   12   13
 *     [1] => 21   22   23
 *     [2] => 31   32   33
 *     [3] => 41   \N   43
 *     [4] => 51   52   53
 *     [5] => 61   62   63
 *     [6] => 71   72   73
 *     [7] => 81   \N   83
 * )
 */
```

6.B. İşletilen Sorgunun İptal Edilmesi

Asenkron olarak (`pg_send_query()` ve `pg_send_execute()` fonksiyonları) ile gönderilen uzun sorgulamalarda, program belirli bir anda sorgunun iptalini isteyebilir. Bunun için `pg_cancel_query()` fonksiyonunu kullanabilirsiniz.

```
bool pg_cancel_query ( resource baglanti )
```

`pg_cancel_query()` fonksiyonu, belirtilen bağlantı üzerinde o an gerçekleşmekte olan asenkron sorgulamanın işlem ortasında iptal edilmesini sağlar.

```
/*
 * İlk önce bağlantıyı sağlayan sunucu işleminin
 * bekleme konumunda olduğundan emin oluyoruz.
 */
if (!pg_connection_busy($baglanti))
    /*
     * Bağlantının sorgu gönderimine uygun olduğunu öğrendikten sonra,
     * ilgili sorgunun gönderimini asenkron olarak gerçekleştiriyoruz.
     */
    pg_send_query($baglanti, "SELECT ...");

/*
 * Belirli bir duruma bağlı olarak, programın herhangi
 * bir noktasında sorgunun iptali isteniyor.
 */
pg_cancel_query($baglanti);
```

`pg_cancel_query()` fonksiyonunun, parametre olarak aldığı bağlantıyı sağlayan sunucu işlemi üzerinde gerçekleşmekte olan sorgunun iptalini istediğine dikkat edilmelidir. İptal istemi sunucuya ulaşmadan önce sorgu işletiminin sonuçlanmış olması durumunda, iptal istemi başarısızlık ile sonuçlanacaktır. Başarılı bir sorgu iptalinde ise, iptali gerçekleşmiş olan sorgu sonucu işlemin tamamlanamadığına dair hata döndürecektir.

6.C. Diğer Sorgu Fonksiyonları

Sorgu fonksiyonları başlığı altında katagorize edemediğimiz diğer ilgili fonksiyonları bu bölümde tanıtmaya çalışacağız.

```
array pg_meta_data ( resource baglanti, string tablo_adi )
```

pg_meta_data() fonksiyonu, parametre olarak aldığı bağlantı üzerinden belirtilen tablo hakkında çeşitli bilgileri kendisine çağırana dizi halinde döndürür. Fonksiyon herhangi bir hata durumunda FALSE değeri döndürecektir.

pg_meta_data() fonksiyonunun yaptığı iş, kullanıcı için aşağıdaki *SQL* komutunun sorgusunu gerçekleştirip, topladığı çeşitli değerleri dizi içinde döndürmekten ibarettir.

```
SELECT a.attname, a.attnum, t.typname, a.attlen, a.attnotNULL, a.atthasdef, a.attndims
  FROM pg_class AS c, pg_attribute AS a, pg_type AS t
  WHERE
    a.attnum > 0 AND a.attrelid = c.oid AND c.relname = '<tablo_adi>'
    AND a.atttypid = t.oid
  ORDER BY a.attnum;
```

7. Sorgu Sonuçları Üzerinde İşlemler

Bu bölüme kadar veritabanı ile program arasında nasıl başarılı bir bağlantı kurup, kurulu bağlantı durumu hakkında bilgi aldıktan sonra ilgili sorgulamalarımızı nasıl gerçekleştirebileceğimiz üzerinde durduk. Bu bölümde ise dönen sorgu sonuçları üzerinde işlem yapmak için kullanabileceğimiz fonksiyonlar üzerinde durulmaya çalışacağız.

7.A. Sonuç Hakkında Durum Bilgisi

Veritabanı üzerinde gerçekleştirilen herhangi bir sorgunun sonucunda dönecek olan değerler hakkında durum bilgisi edinmek için bu bölümde tanıtılacak olan fonksiyonları kullanabilirsiniz.

```
mixed pg_result_status ( resource sonuc [,int donecek_deger_tipi] )
```

pg_result_status() fonksiyonu, parametre olarak aldığı sorgunun sonucunu kendisini çağırana döndürecektir.

Fonksiyonun kendisini çağırana döndürecek olduğu değeri, parametre olarak aşağıdaki değişkenleri kullanarak ayarlayabilirsiniz:

```
PGSQL_STATUS_LONG    /* Durum kodunu döndür. (Öntanımlı seçenek) */
PGSQL_STATUS_STRING  /* Durum kodunun değişken adını döndürür. */
```

pg_result_status() fonksiyonu sonucu dönecek olan durum kodlarının listesine aşağıdaki tablodan ulaşabilirsiniz.

PGSQL_EMPTY_QUERY	PGSQL_COPY_TO	PGSQL_NONFATAL_ERROR
PGSQL_COMMAND_OK	PGSQL_COPY_FROM	PGSQL_FATAL_ERROR
PGSQL_TUPLES_OK	PGSQL_BAD_RESPONSE	

```
string pg_result_error ( resource sonuc )
```

pg_result_error() fonksiyonu, hatalı bir sorgu sonucunda oluşan hata mesajını kendisini çağırana döndürür.

Asenkron sorgu işletiminde yapılan sorgulamalar yiv mantığı ile çalıştığından, sunucu üzerindeki hata mesajı sürekli değişmektedir. Dolayısıyla, herhangi bir sorgu sonrasında

oluşan hata mesajını öğrenmek için kullandığınız pg_last_error() fonksiyonu beklediğinizden başka (daha sonra oluşmuş olan) bir hata mesajını döndürebilir. Bu nedenle belirli bir sorgu sonucunun hata mesajını almak için pg_result_error() ya da pg_result_error_field() fonksiyonları kullanılır.

Senkron sorgu işletim fonksiyonları, hata durumunda FALSE değeri döndürdüğü için elimize herhangi bir sonuç yapısı ulaşmamaktadır. Dolayısıyla bu tür fonksiyonların sonuçları üzerinde pg_result_error() ve pg_result_error_field() fonksiyonlarını kullanamayız. Bu nedenle, bir sonuç yapısına ihtiyaç duyan fonksiyonları, asenkron bir sorgu işletiminden sonra pg_get_result() fonksiyonundan elde edilecek sonuç yapısı ile birlikte kullanabiliriz ancak.

```
string pg_result_error_field ( resource sonuc,
                               int      hata_alan_kodu )
```

pg_result_error_field() fonksiyonu, parametre olarak girilen sonuç yapısı içindeki belirtilen hata mesajı alanını kendisini çağırana döndürür. Fonksiyon ile birlikte kullanabileceğiniz hata alan kodlarını aşağıdaki tabloda bulabilirsiniz.

PGSQL_DIAG_SEVERITY	PGSQL_DIAG_SEVERITY
PGSQL_DIAG_SQLSTATE	PGSQL_DIAG_SQLSTATE
PGSQL_DIAG_MESSAGE_PRIMARY	PGSQL_DIAG_MESSAGE_PRIMARY
PGSQL_DIAG_MESSAGE_DETAIL	PGSQL_DIAG_MESSAGE_DETAIL
PGSQL_DIAG_MESSAGE_HINT	PGSQL_DIAG_MESSAGE_HINT
PGSQL_DIAG_STATEMENT_POSITION	

```
/* Veritabanı üzerinde asenkron bir sorgu gerçekleştiriyoruz. */
pg_send_query($baglanti, “SELECT ...”);

/* Dönen sorgu sonucunu alıyoruz. */
$sonuc = pg_get_result($baglanti);

/* Sorgu sonucunun son durumuna bakıyoruz. */
if (pg_result_status($sonuc) != PGSQL_TUPLES_OK)
{
    /* Sorgu sonucu beklediğimiz satırlar dönmemiş. */

    /* Sonuç hata mesajını yazdırıyoruz. */
    print “Sorgu sonucu hata döndü!\n”;
    print pg_result_error($sonuc);
}
```

7.B. Sonuç Hakkında Tablo/Satır/Sütun Bilgisi

Başarılı bir sorgu sonucu üzerinde bu bölümde tanıtılacak olan fonksiyonları kullanarak, çeşitli tablo, satır ve sütun bilgilerine ulaşabilirsiniz.

```
int pg_affected_rows ( resource sonuc )
```

pg_affected_rows() fonksiyonu, parametre olarak aldığı sonuçtan etkilenen sütun sayısını kendisini çağırana döndürecektir.

```
int pg_num_fields ( resource sonuc )
```

pg_num_fields() fonksiyonu, parametre olarak aldığı sonucun kaç sütundan oluştuğunu kendisini çağırana döndürür. Herhangi bir hata durumunda fonksiyon -1 değeri döndürecektir.

```
int pg_num_rows ( resource sonuc )
```

pg_num_rows() fonksiyonu, parametre olarak aldığı sonucun toplam kaç satırdan

oluştuğunu kendisini çağırana döndürür. Herhangi bir hata durumunda fonksiyon -1 değeri döndürecektir.

```
int pg_field_is_null ( resource sonuc, int satir, mixed sutun )
```

pg_field_is_null() fonksiyonu, parametre olarak aldığı sonucun, belirtilen satır ve sütunundaki alanın değerinin NULL olması durumunda 1, aksi halde 0 değeri döndürür.

Sütun kısmına, ilgili sütünun indisini girebileceğiniz gibi, başlığını da girebilirsiniz.

```
string pg_field_name ( resource sonuc, int sutun )
```

pg_field_name() fonksiyonu, belirtilen sonuç tablosundaki, parametre olarak girilen sütununun başlığını kendisini çağırana döndürür.

```
int pg_field_num ( resource sonuc, string alan_adi )
```

pg_field_num() fonksiyonu, parametre olarak aldığı sonucun belirtilen alan adının sütun indisini kendisini çağırana döndürür. Herhangi bir hata durumunda fonksiyon -1 döndürecektir.

```
int pg_field_prtlen ( resource sonuc, int satir, mixed sutun )
```

pg_field_prtlen() fonksiyonu, parametre olarak aldığı sonuçtaki belirtilen satır ve sütunda bulunan verinin uzunluğunu kendisini çağırana döndürür.

```
int pg_field_size ( resource sonuc, int sutun )
```

pg_field_size() fonksiyonu, parametre olarak aldığı sonucun belirtilen sütununun tipinin boyutunu kendisini çağırana döndürür.

```
int pg_field_type_oid ( resource sonuc, int sutun )
```

pg_field_type_oid() fonksiyonu, parametre olarak aldığı bağlantının belirtilen sütunun tipininin OID değerini kendisini çağırana döndürür.

```
string pg_field_type ( resource sonuc, int sutun )
```

pg_field_type() fonksiyonu, parametre olarak aldığı sonucun ilgili sütununun tipini kendisini çağırana karakter katarı olarak döndürür.

pg_field_type() fonksiyonu, sizin yerinize pg_field_type_oid() fonksiyonundaki rutinleri kullanarak elde ettiği *OID* değeri ile aşağıdaki *SQL* sorgusunu gerçekleştirir.

```
SELECT typname FROM pg_type WHERE oid = <OID>;
```

8. Veri Alımı

Bu bölümde başarı ile gerçekleşen bir sorgu sonucunda dönecek olan sonuç kaynağından verilerin alınmasında kullanılacak olan fonksiyonları incelemeye çalışacağız.

Bir çok veri tipi PostgreSQL tarafından destekleniyor olsa da (Bkz. `SELECT typname FROM pg_type`) sadece temel bir kaç tip PHP tarafından aynı tipce desteklenmektedir. Bunlar arasında, tüm tamsayı değerler `integer` tipi, ondalık ve reel sayılar `float` tipi şeklinde çağrıldıkları değişkenlere dönüş yapacaklardır. `boolean` tipi ise doğru için `t`, yanlış için `f` döndürecektir. Bunun dışında kalan diğer tüm tipler, karakter katarı halinde dönüş yapacaktır. (Bu konu hakkında ayrıntılı bilgi için libpq kütüphanesinde incelenen PQgetvalue() fonksiyonuna bakınız.)

```
mixed pg_fetch_result ( resource sonuc, [int satir,] mixed sutun )
```

pg_fetch_result() fonksiyonu, parametre olarak aldığı sonucun, belirtilen satır ve sütundaki değeri kendisini çağırana döndürür. Sütun parametresi olarak isteğe bağlı olarak ilgili sütunun indisini ya da başlığını kullanabilirsiniz.

```
array pg_fetch_all ( resource sonuc )
```

pg_fetch_all() fonksiyonu, parametre olarak aldığı sonucu dizi halinde içinde toparlayıp kendisini çağırana döndürür. Tablodaki NULL alan değerleri için PHP'nin kendi içinde tanımlanmış NULL değeri kullanılacaktır. Fonksiyon, parametre olarak aldığı sonuç satır içermediği zaman FALSE değeri döndürücektir.

> *Tablonun çok büyük olduğu durumlarda dizi oluşturulurken php.ini dosyasında tanımlanmış kaynak sınırlarına (memory_limit gibi) dikkat etmelisiniz. Aksi halde tablonun tamamının diziye aktarımı gerçekleşmeyebilir.*

```
array pg_fetch_array (   resource sonuc
                       [, int      satir
                        [,int      dizi_tipi]] )
```

pg_fetch_array() fonksiyonu, parametre olarak aldığı sorgu sonucunun belirtilen satırını gösterilen dizi tipinde kendisini çağırana döndürür. Tablodaki NULL alan değerleri için PHP'nin kendi içinde tanımlanmış NULL değeri kullanılacaktır.

Herhangi bir satır belirtilmediği taktirde, pg_fetch_array() sonuç üzerinden bir sonraki satırı kendisini çağırana döndürecektir. Bu şekilde döngü yapıları (for, while, vs. gibi) içinde ardarda çağrılmak suretiyle fonksiyon rahatlıkla kullanılabilir.

Sonuç tipinde kullanılabilecek olan değerlere aşağıdaki listeden ulaşabilirsiniz.

```
PGSQL_ASSOC  /* Dizinin anahtar değerlerini sütun başlıkları oluştursun. */
PGSQL_NUM    /*                            sütun indisleri oluştursun. */
PGSQL_BOTH   /* Anahtar değerler hem sütun başlıkları, hem de sütun indisileri
                ile ayrı ayrı oluşturulsun. (Öntanımlı seçenek.) */
```

Dizi tipinin belirtilmek istendiği, fakat satır sayısının kullanılmak istenmediği durumlarda, satir parametresi yerine NULL girdikten sonra dizi_tipi değerini yazabilirsiniz.

pg_fetch_array() fonksiyonu, pg_fetch_row() fonksiyonuna oranla çok daha kullanışlı olmasına karşı, kullanıcılar genelde sütun başlıklarının alınırken ayrı birer sorgu yapıldığını zannedip bunun performansı düşürebileceğinden dolayı pg_fetch_array() yerine pg_fetch_row() fonksiyonunu tercih etmektedirler. Bunun aksine, sütun başlıkları zaten sonuç yapısı ile beraber döndüğünden dolayı bunların ayrıca alımı, yok denecek kadar az derecede bir performans kaybına yol açmaktadır.

```
array pg_fetch_assoc ( resource sonuc [,int satir] )
```

pg_fetch_assoc() fonksiyonu, pg_fetch_array(sonuc, [satir,] PGSQL_ASSOC) şeklinde, pg_fetch_array() fonksiyonunun parametreleri kısaltılmış bir türevidir.

```
array pg_fetch_row ( resource sonuc [,int satir] )
```

pg_fetch_assoc() fonksiyonu, pg_fetch_array(sonuc[, satir], PGSQL_NUM) şeklinde, pg_fetch_array() fonksiyonunun parametreleri kısaltılmış bir türevidir.

```
object pg_fetch_object (   resource sonuc
                         [, int      satir
                          [,int      sonuc_tipi]] )
```

pg_fetch_object() fonksiyonu, pg_fetch_array() fonksiyonuna benzer olarak işlev

gösterip, ondan farklı olarak kendisini çağırana sonuç değelerini bir dizi içinde değil de, nesne içinde döndürecektir.

Sonuç tipi öntanımlı olarak PGSQL_ASSOC şeklinde olup, sonuç tipinde kullanıcının herhangi bir değer girmesine izin verilmemektedir. Bu özellik, PHP'nin eski sürümleri ile yazılmış programlar ile uyumluluğu sağlamak için barındırılmaktadır.

```
bool pg_result_seek ( resource sonuc, int baslangic_satiri )
```

pg_result_seek() fonksiyonu, sonuç alımında kullanılacak olan başlangıç konumunu belirtilen satıra öteler.

```
/* Örnek tablo, v0, v1, v2... şeklinde değerler barındıran c sütununa sahip. */

$komut = <<<EOF
SELECT
  '1' || c AS “1. Sütun”,
  '2' || c AS “2. Sütun”,
  '3' || c AS “3. Sütun”
FROM
  ornek_tablo
EOF;
$sonuc = pg_query($baglanti, $komut);
if (!$sonuc)
    /* Sorgu başarısız oldu. */

$satir_sayisi = pg_num_rows($sonuc);
$sutun_sayisi = pg_num_fields($sonuc);

/* Başlıkları yazdırıyoruz. */
for ($j = 0; $j < $sutun_sayisi; $j++)
    print “$j: ”.pg_field_name($sonuc, $j).”\n”;

/* İlk 3 satırı yazdırıyoruz. */
for ($i = 0; $i < 3; $i++)
{
    $satir = pg_fetch_row($sonuc, $i, PGSQL_NUM);
    for ($j = 0; $j < $sutun_sayisi; $j++)
        print "[$i][$j]: $satir[$j]\n";
}

/* 2 satır ilerletiyoruz konumumuzu. */
pg_result_seek($sonuc, 2);

/* Varılan konumdaki satırı yazdırıyoruz. */
$satir = pg_fetch_array($sonuc, NULL, PGSQL_ASSOC);
print "C1: ".$satir["1. Sütun"]."\n".
      "C2: ".$satir["2. Sütun"]."\n".
      "C3: ".$satir["3. Sütun"]."\n";
```

Bu ufak programın çıktısı aşağıdaki gibi olacaktır.

```
0: 1. Sütun
1: 2. Sütun
2: 3. Sütun
[0][0]: 1v0
[0][1]: 2v0
[0][2]: 3v0
[1][0]: 1v1
[1][1]: 2v1
[1][2]: 3v1
[2][0]: 1v2
[2][1]: 2v2
[2][2]: 3v2
C1: 1v5
C2: 2v5
C3: 3v5
```

9. Büyük Boyutlu Nesneler

Bu bölümde, PostgreSQL *LO* (*Large Objects*) kayıtları üzerinde işlem yapmak için PHP tarafından sağlanan fonksiyonları tanıtmaya çalışacağız.

Fonksiyonların tanıtımına geçmeden önce, *LO* hakkında daha ayrıntılı bilgi için öncelikli olarak libpq kütüphanesinin tanıtımında yer alan ilgili bölüme göz atılmasını tavsiye ederiz. Aksi halde bahsi geçecek olan fonksiyonlar, bunları ilk defa görecek biri için pek bir anlam ifade etmeyecektir.

> *LO fonksiyonlarını kullanabilmek için, komutları `transaction` blokları içinde göndermeniz gerekmektedir.*

Bu bölümde tanıtılacak olan fonksiyonların istemci taraflı olduğuna dikkat ediniz. Yani, gerçekleştirilecek işlemlerde kullanılacak dosya sistemi, istemcininki olacaktır. Örneğin, herhangi bir dosya gönderileceği (ya da kaydedileceği) zaman, istemci dosya sisteminden alınacaktır (ya da istemci dosya sistemine kaydedilecektir).

```
int pg_lo_create ( [resource baglanti] )
```

`pg_lo_create()` fonksiyonu, geçerli bağlantının sağlandığı veritabanı üzerinde yeni bir *LO* kaydı oluşturulmasını sağlar. Başarılı olması durumunda oluşturulan yeni *LO* kaydının `loid` değerini, aksi halde `FALSE` değeri döndürecektir.

```
int pg_lo_import ( resource baglanti, string dosya_yolu )
```

`pg_lo_import()` fonksiyonu, istemci üzerinde yolu belirtilen dosyanın geçerli bağlantının kurulu olduğu sunucuda oluşturulan *LO* içine kaydedilmesini sağlar. Fonksiyon başarılı olması durumunda oluşturulan *LO* kaydının `loid`, aksi halde `FALSE` değerini döndürecektir.

> *LO içinde kaydedilecek dosyanın istemci tarafından alınıp, bağlantının diğer ucundaki veritabanına gönderileceğine dikkat ediniz.*
>
> *`php.ini` dosyasında tanımlanan `safe_modu` seçeneğinin etkin olması durumunda, `pg_lo_import()` fonksiyonu ile aktarılacak dosyanın ve fonksiyonu çalıştıran kullanıcının aynı `UID` değerine sahip olması şartı koşulacaktır.*

```
bool pg_lo_unlink ( resource baglanti, int loid )
```

`pg_lo_unlink()` fonksiyonu, geçerli bağlantının kurulu olduğu veritabanı sunucusu üzerinde belirtilen `loid` değerine sahip *LO* kaydını silecektir.

```
bool pg_lo_export ( resource baglanti, int loid, string dosya_yolu )
```

`pg_lo_export()` fonksiyonu, bağlantının gerçekleştiği veritabanı üzerindeki berlitilen `loid` değerine sahip *LO* kaydının, istemci dosya sistemindeki girilen dosya yoluna aktarılmasını sağlar.

```
resource pg_lo_open ( resource baglanti, int loid, string bicim )
```

`pl_lo_open()` fonksiyonu, geçerli bağlantının sağlandığı veritabanı üzerindeki ilgili `loid` değerine sahip *LO* kaydının belirtilen biçimde açılmasını sağlar. Başarılı olması durumunda açılan *LO* kaydının kullanımı için gerekli dosya yapısı işaretçisini, aksi halde `FALSE` değerini döndürecektir.

> *`lo_open()` ile açılan bir LO kaydını kapatmadan önce, bağlantıyı bitirmemelisiniz. Aksi halde, yaptığınız tüm değişiklikler (transaction içinde olduğunuzdan dolayı) `ROLLBACK` ile geri alınacaktır.*

```
string pg_lo_read ( resource lo_kaydi [,int uzunluk] )
```

pg_lo_read() fonksiyonu, belirtilen (daha önceden açılmış) *LO* kaydı üzerinden parametre olarak girilen (ya da öntanımlı olarak 8192 bayt) uzunluktaki veriyi okuyarak, okunan veriyi kendisini çağırana karakter katarı olarak döndürür. Herhangi bir hata durumunda fonksiyon FALSE değeri döndürecektir.

```
int pg_lo_read_all ( resource lo_kaydi )
```

pg_lo_read_all() fonksiyonu, daha önce pg_lo_open() ile açılmış olan *LO* kaydını veritabanından (lo_read() ile 8192 baytlık bloklar halinde) okuyarak, ekrana yazdırır. Sonuç olarak toplam okunan karakter sayısını, herhangi bir hata durumunda ise FALSE değerini kendisini çağırana döndürecektir.

```
bool pg_lo_seek (  resource lo_kaydi,
                    int      ilerlenecek_uzunluk
                  [,int      tarama_konumu] )
```

pg_lo_seek() fonksiyonu, girilen (daha önceden pg_lo_open() ile açılmış) *LO* kaydı üzerindeki geçerli işaretçiyi belirtilen uzunlukta ilerletmek için kullanılır. Tarama konumu olarak girebileceğiniz değerler şu şekilde:

```
PGSQL_SEEK_SET  /* İşaretçiyi, dosyanın en başından ilerlet.           */
PGSQL_SEEK_CUR  /*            o anki konumundan ilerlet. (Öntanımlı.) */
PGSQL_SEEK_END  /*            dosyanın en sonundan ilerlet.           */
```

```
int pg_lo_tell ( resource lo_kaydi )
```

pg_lo_tell() fonksiyonu, parametre olarak aldığı (daha önceden pg_lo_open() ile açılmış) *LO* kaydı üzerindeki dosya işaretçisinin (dosyanın başından itibaren olan) konumunu kendisini çağırana döndürür.

```
int pg_lo_write ( resource lo_kaydi, string katar [,int uzunluk] )
```

pg_lo_write() fonksiyonu, parametre olarak aldığı (daha önceden pg_lo_open() ile açılmış) *LO* kaydı üzerinde bulunan dosya işaretçisinin konumundan itibaren, belirtilen karakter katarını *LO* kaydı üzerine yazacaktır. Sonuç olarak toplam yazılan karakter uzunluğunu, herhangi bir hata gerçekleşmesi durumunda ise FALSE değerini kendisini çağırana döndürecektir.

```
bool pg_lo_close ( resource lo_kaydi )
```

pg_lo_close() fonksiyonu, parametre olarak girilen pg_lo_open() ile açılmış *LO* kaydının kapatılmasını sağlar.

10. Diğer Fonksiyonlar

Önceki bölümlerde katagorize edemediğimiz fonksiyonları bu bölüm altında toparlamaya çalıştık.

```
array pg_get_notify ( resource baglanti [,int sonuc_tipi] )
```

pg_get_notify() fonksiyonu, geçerli bağlantı üzerinde (LISTEN komutu ile) dinlenmekte olan uyaranlardan NOTIFY komutu ile herhangi bir uyarının istemciye ulaşması durumunda, uyarının bulunduğu diziyi kendisini çağırana döndürür. Herhangi bir uyarının bulunmadığı durumda fonksiyon FALSE değeri döndürecektir.

Birden fazla uyaranın dinlendiği durumda, fonksiyon `FALSE` değeri dönene kadar döngü içinde birden çok kez çağırılarak ulaşan tüm uyarılar teker teker alınabilir.

Sonuç tipinde kullanılabilecek olan değerleri şu şekilde listeleyebiliriz.

```
PGSQL_ASSOC /* Dizinin anahtar değerlerini sütun başlıkları oluştursun. (Öntanımlı) */
PGSQL_NUM   /*                                  indisleri oluştursun. */
PGSQL_BOTH  /* Dizinin anahtar değerler olarak hem sütun başlıkları, hem
               de indisileri kullanılsın. */
```

Dönecek olan diziye örnek vermek gerekirse:

```
/*
 * `test' isimli uyarının pg_get_notify($baglanti, PGSQL_BOTH)
 * ile alınıp var_dump() ile ekrana dökümü.
 */
array(4) {
  [0]=>                 /* PGSQL_NUM kısmı. */
  string(1) "test"
  [1]=>
  int(2317)
  ["message"]=>         /* PGSQL_ASSOC kısmı. */
  string(1) "test"
  ["pid"]=>
  int(2317)
}
```

`string` **`pg_last_notice`** `( resource baglanti )`

`pg_last_notice()` fonksiyonu, geçerli bağlantı üzerinde meydana gelen en son uyarı mesajını, herhangi bir hata durumunda ise `FALSE` değerini kendisini çağırana döndürecektir.

`pg_last_notice()` fonksiyonu, libpq kütüphanesindeki uyarı işlemcilerine bir arayüz sağlamasına rağmen, tıpatıp aynı işlev göstermez. Şöyle ki, uyarı işlemcileri herhangi bir uyarı durumunda direk çağrıldıkları halde, `pg_last_notice()` sadece program akışı içinde isteğe bağlı olarak çağrılır ve sunucuda o ana kadar gerçekleşmiş olan en son uyarı mesajını döndürür. Yani o an bir uyarı mesajı gelmiş olmasa dahi, sunucudan istemciye daha önceden bir uyarı mesajı ulaşmışsa `pg_last_notice()` onu döndürecektir. Ya da benzer olarak aynı anda birden fazla uyarı mesajı gelmesi durumunda, `pg_last_notice()` sadece en sonuncusunun mesajını kendisini çağırana döndürecektir.

Uyarı mesajlarının gözardı edilip edilmeyeceğini `php.ini` dosyasındaki `pgsql.ignore_notice` parametresi ile ayarlayabilirsiniz. Ek olarak, `pgsql.ignore_notice` parametresi etkin olduğu durumda, `pgsql.log_notice` parametresinin ayarlanması ile isteğe bağlı olarak uyarı mesajları kayıt dosyalarında tutulabilir.

`bool` **`pg_free_result`** `( resource sonuc )`

`pg_free_result()` fonksiyonu, parametre olarak aldığı sorgu sonucunun bellekten bırakılmasını sağlar. Şu da unutulmamalıdır ki, bellekten serbest bırakılan bir sonuç tekrar kullanılamaz hale gelecektir.

PHP, programınız sonladığı zaman betiğin çalışması esnasında kullanılan yapılarca harcanan tüm ayrılmış bellek alanını zaten serbest bırakacaktır. Bu nedenle `pg_free_result()` sadece program akışı içinde çok fazla bellek kullanımını gerektiren sorgulamaların yer aldığı durumlarda çalıştırılmalıdır.

```
bool pg_trace (   string   dosya_yolu
                [, string   dosya_acma_bicimi
                 [,resource baglanti]] )
```

pg_trace() fonksiyonu, geçerli bağlantı esnasında sunucu ile istemci arasındaki iletişim kaydının belirtilen dosyaya tutulmasını sağlar.

pg_trace() fonksiyonu libpq kütüphanesindeki PQtrace() fonksiyonuna bir arayüz sağladığından, PQtrace() için gerekli dosya işaretçisi PHP tarafından oluşturulur. Bu sayede, isteğe bağlı olarak pg_trace() fonksiyonuna PHP kütüphanesinde yer alan fopen() dosya açma biçimi parametrelerinden (r, r+, w, w+, a, a+, x ve x+) biri verilebilir. (Öntanımlı dosya açma biçimi olarak w kullanılmaktadır.)

```
bool pg_untrace ( [resource baglanti] )
```

pg_untrace() fonksiyonu, sunucu ile istemci arasında tutulan iletişim raporlamasının kapatılmasını sağlar.

```
bool pg_ping ( [resource baglanti] )
```

pg_ping() fonksiyonu, geçerli veritabanı bağlantısının kontrol edilip, bağlantının kopuk bulunması durumunda tekrar bağlanılması için kullanılır.

> *pg_ping() fonksiyonunu geçerli veritabanı bağlantısı üzerinde bir SELECT 1 sorgusu çalıştırdıktan sonra, bağlantı durumunun CONNECTION_OK olması durumunda TRUE değeri döndürecektir. Aksi, halde PQreset() ile bağlantıyı tekrar kurmaya çalışıp sonucun başarı olması durumunda TRUE, başarısız olması durumunda ise FALSE değeri döndürecektir.*

```
array pg_convert (   resource baglanti,
                     string   tablo_adi,
                     array    dizi
                   [,int      secenekler] )
```

pg_convert() fonksiyonu, parametre olarak aldığı dizi elemanlarını, belirtilen tablodaki dizi anahtar değerleri ile örtüşen alanların tipi için *SQL* cümlesinde kullanılabilecek hale sokarak, yeni bir dizi halinde kendisini çağırana döndürecektir. (Örneğin PostgreSQL'in bool tipi için, gönderdiğiniz dizideki PHP'nin TRUE ya da FALSE değeri yerine, PostgreSQL tarafından anlaşılabilecek t ya da f karakterlerini kullanacaktır.)

Önkoşul olarak, belirtilen tablonun parametre olarak girilen bağlantının diğer ucundaki veritabanında yer alıyor olmasının yanısıra, girilen dizinin anahtar değerlerinin tablodaki alan adları ile örtüşüyor ve dizi değerlerinin ilgili alan tipleri ile uyuşuyor olması gerekmektedir.

Fonksiyon ile kullanılabilecek seçenekler şu şekilde:

```
PGSQL_CONV_IGNORE_DEFAULT   /* Alanın öntanımlı değeri mevcutsa onu kullan. */
PGSQL_CONV_FORCE_NULL       /* Boş katarları NULL tipine çevir. */
PGSQL_CONV_IGNORE_NOT_NULL  /* NOT NULL bağımlılığını gözardı et. */
```

```
string pg_escape_string ( string katar )
```

pg_escape_string() fonksiyonu, güvensiz bir kaynaktan alınan karakter katarının, veritabanına gönderileceği sorgu içine yerleştirilmeden önce olası *SQL Injection* saldırılarına karşı ayıklanması için kullanılır.

```
string pg_escape_bytea ( string veri )
```

pg_escape_bytea() fonksiyonu, veritabanında saklanacak ikili verinin ayıklanması için kullanılır.

```
string pg_unescape_bytea ( string veri )
```

pg_unescape_bytea() fonksiyonu, daha önceden ayıklanmış bytea tipindeki verinin tekrar geri ayıklanmasını için kullanılır. Fonksiyon, ayıklanarak veri tabanı üzerine kaydedilmiş bir verinin, tekrar okunduktan sonra kullanılacağı zaman işletilir.

11. Çalışma Anı (INI) Ayarları

Programın çalışma esnasında, bu bölümde tanıtılacak olan PostgreSQL ile ilgili PHP'nin *INI* değerlerini isteğe bağlı olarak değiştirebilirsiniz.

> *Herhangi bir INI ayarı için gerekli değişikliği sunucu dosya sistemi üzerindeki php.ini dosyasından yapabileceğiniz gibi; herhangi bir INI ayarının geçerli değerini ini_get() fonksiyonu ile alıp, ini_set() fonksiyonu ile de ayarlayabilirsiniz. (Bunun dışında ini_restore() ve get_cfg_var() fonksiyonlarına da göz atabilirsiniz.)*

Ayar Değişkeni	Değişken Tipi	Öntanımlı Değeri	Geçerliliği	Açıklama
pgsql.allow_persistent	boolean	1	PHP_INI_SYSTEM	Kalıcı bağlantı izni.
pgsql.max_persistent	integer	-1	PHP_INI_SYSTEM	Her işlem başına yapılabilecek en çok kalıcı bağlantı sayısı.
pgsql.max_links	integer	-1	PHP_INI_SYSTEM	Her işlem başına yapılabilecek (kalıcı bağlantılar da dahil olmak üzere) en çok bağlantı sayısı.
pgsql.auto_reset_persistent	integer	0	PHP_INI_SYSTEM	Kopan kalıcı bağlantıların otomatik olarak onarılması.
pgsql.ignore_notice	integer	0	PHP_INI_ALL	Uyarı mesajlarının gözardı edilmesi.
pgsql.log_notice	integer	0	PHP_INI_ALL	Uyarı mesajlarının (pgsql.ignore_notice kapalı olduğu zaman) kaydının tutulması.

Geçerliliğin bulunduğu sütunda yer alan değerlerin açıklaması ise şu şekilde:

Geçerlilik	Açıklama
PHP_INI_SYSTEM	İlgili ayar php.ini ve httpd.conf dosyalarından yapılandırılabilir.
PHP_INI_ALL	İlgili ayar php.ini, httpd.conf, .htaccess, kullanıcı betikleri (ini_set() fonksiyonu ile) ve Windows kayıtlarından (*registry*) yapılandırılabilir.

12. Fonksiyon Tablosu

Buraya kadar tanıtılan PHP tarafından sağlanan PostgreSQL fonksiyonlarının kısa birer açıklama ile birlikte tam bir listesini aşağıdaki tabloda bulabilirsiniz.

Fonksiyon	Açıklama	libpq Kütüphanesindeki Eşdeğeri
pg_connect pg_pconnect	Bağlantı kurulumu.	PQconnectdb
pg_connection_status	Bağlantı durumu.	PQstatus
pg_connection_busy	Sunucunun o anki geçerli bağlantı üzerinde bir işlem ile uğraşıp uğraşmadığı.	PQisBusy

pg_dbname pg_host pg_options pg_port	İlgili bağlantı seçenekleri.	PQdbname PQhost PQoptions PQport
pg_client_encoding	Bağlantıda kullanılan istemci karakter kodlaması.	PQclientEncoding
pg_transaction_status	*Transaction* durumu.	PQtransactionStatus
pg_last_error	Sunucudan dönen en son hata ve uyarı mesajı.	PQerrorMessage
pg_parameter_status	Sunucu seçenekleri.	PQparameterStatus
pg_version	Sunucu sürüm numarası.	PQparameterStatus PQprotocolVersion
pg_get_pid	Sunucunun istemci bağlantısını sağlayan işleminin *PID* değeri.	PQbackendPID
pg_set_client_encoding	İstemci karakter kodlamasının ayarlanması.	PQsetClientEncoding
pg_set_error_verbosity	İstemciye dönecek hata mesajı duyarlılığının ayarlanması.	PQsetErrorVerbosity
pg_close	Bağlantının kapatılması.	PQfinish
pg_connection_reset	Bağlantının sıfırlanıp, baştan kurulması.	PQreset
pg_query pg_query_params	Senkron sorgu işletimi.	PQexec PQexecParams
pg_prepare pg_execute	Senkron sorgu hazırlanıp işletimi.	PQprepare PQexecPrepared
pg_send_query pg_send_query_params	Asenkron sorgu işletimi.	PQsendQuery PQsendQueryParams
pg_send_prepare pg_send_execute	Asenkron sorgu hazırlanıp, hazırlanan sorgunun işletilmesi.	PQsendPrepare PQsendQueryPrepared
pg_get_result	Asenkron sorgu sonuçlarının alınması.	PQgetResult
pg_copy_to pg_copy_from pg_put_line pg_end_copy	COPY komutuna yardımcı fonksiyonlar.	PQgetCopyData PQputCopyData PQputCopyEnd
pg_cancel_query	Sorgu iptali.	PQrequestCancel
pg_meta_data	Belirtilen tablo hakkında bilgi.	
pg_result_status	Sunucudan dönen bir sonucun durumu.	PQresultStatus
pg_result_error pg_result_error_field	Başarısız bir sorgu sonucu dönen hata mesajı.	PQresultErrorMessage PQresultErrorField
pg_affected_rows	Sorgu sonucu etkilenen satır sayısı.	PQcmdTuples
pg_num_fields pg_num_rows	Sorgu sonucunun kaç satır ve sütundan oluştuğu.	PQnfields PQntuples
pg_field_is_null pg_field_name pg_field_num pg_field_prtlen pg_field_size pg_field_type	Belirtilen sorgu sonucu alanı hakkında ilgili bilgi.	PQisnull PQfname PQfnumber PQgetlength PQfsize PQftype
pg_result_seek	Sorgu sonucunun belirli bir satırına geçmek için.	
pg_fetch_all pg_fetch_all pg_fetch_array pg_fetch_assoc pg_fetch_row pg_fetch_object	Sorgu sonucunun belirtilen kısmını almak için.	PQgetvalue
pg_lo_create	*LO* oluşturulması.	lo_creat

		`lo_create`
`pg_lo_import`	*LO* kaydının belirli bir noktaya aktarılması.	`lo_import`
`pg_lo_unlink`	*LO* kaydının silinmesi.	`lo_unlink`
`pg_lo_export`	Belirli bir veri kaynağının *LO* olarak aktarılması.	`lo_export`

`pg_lo_open`
`pg_lo_read`
`pg_lo_read_all`
`pg_lo_seek`
`pg_lo_tell`
`pg_lo_write`
`pg_lo_close`
	LO kaydının açılıp, okuma/yazma işlemleri gerçekleştirilmesi için yardımcı fonksiyonlar.	`lo_open`
		`lo_read`
		`lo_lseek`
		`lo_tell`
		`lo_write`
		`lo_close`

`pg_get_notify`	Asenkron uyarı mesajlarının alınması.	`PQnotifies`
`pg_last_notice`	Sunucudan dönen en son uyarı mesajı.	`PQnoticeReceiver` `PQnoticeProcessor`
`pg_free_result`	Sorgu sonucunun istemci belleğinden temizlenmesi.	`PQclear`
`pg_trace` `pg_untrace`	Bağlantının trafiğinin dinlenip belirli bir yere kaydedilmesi için yardımcı fonksiyonlar.	`PQtrace` `PQuntrace`
`pg_ping`	Bağlantının durumunu kontrol etmek için `PING` işlemi.	
`pg_convert`	Gönderilecek verinin ilgili tablo alanları tipine uyarlanması.	

`pg_escape_string`
`pg_escape_bytea`
`pg_unescape_bytea`
	SQL Injection saldırılarına karşı gönderilecek verinin ayıklanması.	`PQescapeString`
		`PQescapeBytea`
		`PQunescapeBytea`

IV. Python Arayüzü

Python programlama dili ile PostgreSQL veritabanına bağlanmak istediğimizde, uygulama arayüzü (*API*) olarak karşımıza bir kaç seçenek çıkmaktadır. Bu arayüzler arasından, zaman içinde gelişimin sürdürmüş ve yaygın olarak kullanılmakta olanlarının dışında, birçoğu miladını doldurduğu için gelişimi durmuştur. Biz bu kitapta, psycopg uygulama arayüzü hakkında bahsedip, psycopg kullanarak PostgreSQL veritabanı ile nasıl etkileşime geçebileceğimiz üzerinde durmaya çalışacağız.

1. Neden psycopg?

psycopg, gelişimi öncesi belirlenmiş işlevsel, başarımı yüksek ve kararlı tasarımı sayesinde, bugün bir çok alternatifine oranla çok daha olgun bir yapıya sahiptir. psycopg, barındırdığı Python *DB-API* v2.0 uyumluluğunun yanısıra, yoğun çoklu yiv desteği, sık imleç kullanımı için önbellek oluşturması, PostgreSQL veri tiplerini Python dilindeki doğal eşlerine dönüştürmesi gibi bir çok özelliği sağlamaktadır. Bunun yanında, çoğu Zope[20] kullanıcısı PostgreSQL veritabanı arayüzü olarak psycopg'yi tercih etmektedir.

Python DB-API 2.0, Python Geliştirme Önergeleri[21] (*Python Enhancement Proposals*, kısaca *PEP*) içerisinde yer alan Veritabanı *API*'si Teknik Özellikleri (*Database API Specification*) önergesinin 2.0 sürümüdür. *PEP*'ler, Python için geliştirilecek programların bir standarta bağlanmasını sağlamış olmanın yanısıra, bir yazılımın farklı ortamlarda çalışması için çok kritik bir önem teşkil etmektedir. Python *DB-API* uyumlu bir veritabanı arayüzü kullanarak geliştirdiğiniz herhangi bir uygulama, SQL sorguları standarda uygun olduğu sürece, *DB-API* uyumlu diğer veritabanı arayüzlerinde de çalışacaktır. Örnek vererek anlatmak gerekirse: Python *DB-API* uyumlu bir PostgreSQL *API*'sinde yazdığınız yazılımı, uygun olarak kodlandığı sürece, yine Python *DB-API* uyumlu bir Firebird *API*'sine taşıyabilirsiniz. Bu sayede uygulamanız, isteğe bağlı olarak farklı veritabanları ile birlikte sorunsuzca çalışabilir duruma gelecektir, hem de bunun için ayrıca oturup tüm veritabanı sorgularının baştan yazılmasına gerek duyulmadan.

> *Buna ek olarak psycopg, PEP 246 ile tanımlanmış nesne adaptasyonunu (Object Adaptation) da (kısmi olarak) desteklemektedir. Bunun ile ilgili bir örnek Sık Karşılaşılan Problemler kısmında incelenecektir.*

Çoklu yiv desteğine sahip programlarda, her bir yiv için ayrı bir bağlantı oluşturmak oldukça kaynak tüketen bir tasarım olmanın yanında, neredeyse çok nadir durumlar dışında asla tercih edilmez. Bunun yerine varolan belirli sayıdaki bağlantı yapısı, yivler arası paylaştırılarak kaynak tasarrufu sağlanır. (Bunun, başarım artışını da beraberinde getireceği aşikardır.) Fakat bu işlem, kullanılacak arayüzün yiv-korunaklı (*thread-safe*) olmasını gerektirir. Bunun ile kasıt şudur: Uygulama arayüzü tarafından sağlanan herhangi bir yapıya, aynı anda birden çok işlemin farklı yivler aracılığı ile erişmeye çalışması durumunda, program akışında hiçbir sorun olmadan her şey yolunda gitmelidir.

20 Zope, içerik yönetim sistemleri, iç ağlar, portal siteleri ve genel uygulamalar için açık kaynak kodlu çok gelişmiş bir uygulama sunucusudur. Ayrıntılı bilgi için http://www.zope.org/ adresine bakabilirsiniz.

21 Tüm bir PEP içeriğine http://www.python.org/peps/ adresinden ulaşabilirsiniz.

Konumuzla ilgili olarak, psycopg, bağlantı yapılarının yivler arası kullanımında herhangi bir sorun çıkarmayacak şekilde tasarlanmıştır. Bu nedenle psycopg, (çok yoğun erişim altında dahi) kararlı bir yiv-korunaklı yapı sunmaktadır.

Bir veritabanı uygulamasında imleçlerin (`CURSOR`) açılıp kapatılması, sık tekrarlanan bir rutindir. Bu nedenle, bir arayüzün imleç tasarımı ne derece iyiyse, bu yazılım performansına o kadar artış şeklinde yansıyacaktır. Bu noktada psycopg, en önemli özelliklerinden biri olan imleç havuzları ile bir adım öne çıkmaktadır. Yivler arası paylaştırılan imleçler yiv sonlanarak kapatıldığında, yivin veritbanı ile olan fiziksel bağlantısı koparılmaz. Başka bir yiv, aynı bağlantı üzerinde tekrar bir imleç isteği bildirdiğinde ise, havuzda bulunan mevcut imleçler arasından kendisini çağırana bir tane gönderilir. Bu sayede, yivler arası imleç kullanımında çok yüksek bir performans sağlanır.

libpq kütüphanesinde, `PQgetvalue()` fonksiyonunu incelerken kütüphane tarafından dönecek olan tüm verilerin karakter katarı ya da ikili biçimde olacağından bahsetmiştik. PostgreSQL'in birbirinden farklı onlarca tipi desteklediği ve bir de buna kullanılcıların kendi tiplerini oluşturabileceği eklendiğinde, bir verinin sadece tek bir biçimde dönecek olması, programcı için bazen çok büyük bir kısıtlama haline gelebilir. Bu esnada psycopg, Python tarafından desteklenen tiplerin gücünü de arkasına alarak, bu sıkıntıyı en aza indirgemeyi başarıyor. Bunu yaparken, sunucudan dönen verinin geldiği sütunun tipine bakarak, bunu Python tarafından doğal olarak desteklenen tipler ile eşleştirir[22]. Bu sayede programcıya Python dili tarafından sunulan tiplerden istediği gibi yararlanma fırsatı tanınmış olur.

Tüm bu sayılan özelliklerin yanısıra, internette bulabileceğiniz bir çok karşılaştırmada, psycopg'nin alternatiflerine oranla ne derece yüksek performans sergilediğini gözlemleyebilirsiniz. Bu, kendi tasarımı ile birlikte gelen üstün yiv ve imleç havuzu özelliklerinin beraberinde sağladığı aşikar bir getiridir.

Kuşkusuz ki, kullanılacak bir yazılım için, o yazılımın barındırdığı özelliklerden sonra olması beklenen en önemli özellik, yazılımın teknik desteğidir. Bu konuda psycopg'nin hiç de haksız olmadığı bir üne sahip olduğunu söylememiz pek de abartı olmaz. psycopg ile ile yaşadığınız herhangi bir problem hakkında posta listelerini başvurabilirsiniz. Özellikle arayüzün iç işleyişi ile ilgili olan sorunlarda en geç 1-2 saat içinde yanıt almanız oldukça olasıdır.

2. psycopg Kurulumu

Bu bölümde, psycopg'nin kaynak kodundan nasıl yükleneceğini inceleyeceğiz.

> *psycopg de, bir çok PostgreSQL arayüzü gibi libpq kütüphanesine ihtiyaç duymaktadır. Bu nedenle, psycopg'yi yüklemeden önce sisteminizde libpq kütüphanesinin kurulu olup olmadığını kontrol ediniz.*

Python için yazılan bir çok veritabanı arayüzü, tarih ve zaman tiplemeleri için eGenix mxDateTime[23] paketini kullanmaktadır. psycopg de 1.x sürümlerinde bu bağımlılığı gösteren arayüzlerden biri olduğu için, kurulum esnasında mxDateTime paketine ihtiyaç duyulmaktadır. Fakat bu bağımlılık 2.x sürümleri ile birlikte pyDateTime seçeneği sayesinde isteğe bağlı olarak ortadan kalkmıştır.

> *psycopg 2.0 bu kitabın geliştirilmekte olduğu tarih itibari ile halen beta sürümlerinde olsa da, gerek 1.x sürümlerine getirdiği yenilikler açısından, gerekse en kısa zamanda kararlı sürüme ulaşacak olmasından*

22 PostgreSQL veri tiplerinin, kullanılan programlama dili tarafından destekleniyor olması programcı açısından göz ardı edilemeyecek bir kolaylık sağlar. psycopg'de bu kolaylığı programcısına vermek için bir kaç seçenek sunmaktadır. Bunun dışında kendiniz de biraz uğraşarak yazacağınız psycopg yamaları ile PostgreSQL veri tiplerini Python içinde rahatlıkla kullanabilirsiniz. (Bunun ile ilgili bir örneği psycopg'nin internet sayfasında bulabilirsiniz.)

23 eGenix mxDateTime paketine http://www.egenix.com/files/python/mxDateTime.html adresinden ulaşabilirsiniz.

biz de bu kitapta bu sürümü kullanacağız.

psycopg 1.x, şuan için kararlı olarak tabir edilen sürüm olduğundan, çoğu Linux ve BSD dağıtımı, paket sisteminde bu versiyona yer vermektedir. psycopg 1.x sürümünün ikili paketlerine şu adreslerden ulaşabilirsiniz.

Paket Biçimi	**Adres**
RPM	http://download.fedora.redhat.com/pub/fedora/linux/extras/4/i386/
DEB	python2.3-psycopg paketini Debian GNU/Linux yansılarının herhangi birinden edinebilirsiniz.
pkgsrc	NetBSD pkgsrc CVS ağacında databases/py-psycopg altında bulabilirsiniz.
TAR	psycopg'yi kaynak kodlarından derlemek için http://initd.org/pub/software/psycopg/ adresindeki sıkıştırılmış TAR paketlerinden birini kullanabilirsiniz.

Microsoft Windows kullanıcıları psycopg'yi kaynak kodundan derleyebilecekleri gibi, Jason Erickson tarafından http://stickpeople.com/projects/python/win-psycopg/ adresinde sağlanan paketleri de kullanabilirler.

psycopg bir Python modülü olarak işlev gösterdiğinden, kurulum gerçekleştikten sonra modül dosyalarını Python'un `site-packages` dizini altında bulabilirsiniz:

```
# Örnek bir site-packages dosya listesi
~$ ls /usr/lib/python2.3/site-packages/
debconf.py   gtk-2.0/  Numeric.pth       pygtk.py@  pygtk.py.python2.3-gtk2
debconf.pyc  mx/       psycopgmodule.so  pygtk.pyc  README
debconf.pyo  Numeric/  pygtk.pth         pygtk.pyo

~$ ls /usr/lib/python2.3/site-packages/psycopg2
extensions.py   extras.pyc    pool.py      psycopg1.pyc  tz.pyc
extensions.pyc  __init__.py   pool.pyc     _psycopg.so*
extras.py       __init__.pyc  psycopg1.py  tz.py
```

psycopg2'nin kaynak kodundan kurulumu için gerekli adımları şu şekilde kısaca özetleyebiliriz:

```
~$ cd /tmp
/tmp$ wget http://initd.org/pub/software/psycopg/psycopg2-2.0b6.tar.gz
/tmp$ tar -zxf psycopg2-2.0b6.tar.gz
/tmp$ cd  psycopg2-2.0b6

# İsteğe bağlı olarak kurulum dosyalarında bir kaç ufak değişiklik yapıyoruz.
/tmp/psycopg2-2.0b6$ vim setup.cfg
/tmp/psycopg2-2.0b6$ vim setup.py

# Paketi derleyip, oluşan ikili paketleri sistemde doğru yerlerine taşıyoruz.
/tmp/psycopg2-2.0b6$ python setup.py build
/tmp/psycopg2-2.0b6$ su -c “python setup.py install”
```

psycopg 1.x sürümlerinin kaynak kodundan kurulumunu anlatmamamızın sebebi, ilk olarak bu sürümün bir çok dağıtım tarafından ikili paketinin birlikte gelmesi ve de artık psycopg kullanıcılarının 2.0 sürümlerine geçişini kolaylaştırmayı düşündüğümüz içindir.

Ek olarak Microsoft Windows kullanıcılarının psycopg paketini nasıl derleyebilecekleri, kaynak kodu ile birlikte gelen INSTALL dosyasında ayrıntılı olarak incelenmiştir.

psycopg paketinin kurulu olup olmadığını öğrenmek için, Python komut satırından psycopg modülünü kullanmaya çalışarak basit bir test yapmamız mümkün:

```
# python yorumlayıcısının bulunduğumuz dizindeki modülü algılamasını
# istemediğimizden, sistem genelinde yüklenen modülü algılaması için
# kurulumu gerçekleştirdiğimiz dizinin dışında işleme devam ediyoruz.
/tmp/psycopg-2.0b6$ cd /tmp
```

```
/tmp$ python -v
...
>>> import psycopg2
import psycopg2 # directory /usr/lib/python2.3/site-packages/psycopg2
# /usr/lib/python2.3/site-packages/psycopg2/__init__.pyc matches /usr/lib/python2.3/site-
packages/psycopg2/__init__.py
import psycopg2 # precompiled from /usr/lib/python2.3/site-packages/psycopg2/__init__.pyc
dlopen("/usr/lib/python2.3/lib-dynload/datetime.so", 2);
import datetime # dynamically loaded from /usr/lib/python2.3/lib-dynload/datetime.so
# /usr/lib/python2.3/site-packages/psycopg2/tz.pyc matches /usr/lib/python2.3/site-
packages/psycopg2/tz.py
import psycopg2.tz # precompiled from /usr/lib/python2.3/site-packages/psycopg2/tz.pyc
dlopen("/usr/lib/python2.3/lib-dynload/time.so", 2);
import time # dynamically loaded from /usr/lib/python2.3/lib-dynload/time.so
dlopen("/usr/lib/python2.3/site-packages/psycopg2/_psycopg.so", 2);
import psycopg2._psycopg # dynamically loaded from /usr/lib/python2.3/site-
packages/psycopg2/_psycopg.so
>>>
```

Yukarıdakine benzer bir çıktı, psycopg paketinizin sorunsuz kurulduğu anlamına gelir.

> *Bundan sonraki bölümlerde psycopg paketini sorunsuz bir şekilde kurduğunuzu farzedip, anlatıma bu elde ile devam edilecektir.*

İleriki bölümlerde, psycopg fonksiyonlarının tanıtımı esnasında, elimizden geldiğince Python *DB-API* v2.0 önergesine sadık kalmaya çalıştık. Bunun en büyük sebebi, yazılacak yazılımlarının taşınılabilirliğinin önemli olduğunu düşünüp, bu konuda okuyucuyu da biliçlendirmeye çalışmamızda yatmaktadır. Bu nedenle, herhangi bir fonksiyon hakkında aktarılan bilgilin yeterli olmadığını düşündüğünüz bir anda, kurduğunuz psycopg paketinin kaynak kodundan ilgili fonksiyon ya da özelliğin işleyişi hakkında daha fazla bilgi edinebilirsiniz.

3. Tanıtım Uygulaması

Python ile PostgreSQL veritabanına ulaşmak için psycopg tarafından sağlanan *API* fonksiyonlarının tanıtımına geçmeden önce, okuyucuya konu hakkında yüzeysel bir bakış açısı vermesi açısında giriş seviyesinde bir tanıtım uygulaması hazırlayacağız.

Bu örnek uygulamada, bir *WAN* (*Wide Area Network*) ağında, sistemler arası bant yoğunluğu hakkında bize (kısmen) bilgi veren `ping`[24] istatistiklerini bir veritabanı üstünde tutuyor olacağız. Ve yazacağımız uygulama ile bu veritabanına bağlanıp, `ping` süreleri verilen bir aralığın dışına taşan hatları ekrana dökmeye çalışacağız.

İlk olarak bahsi geçen bu bilgileri tutacak olan veritabanı tablolarımızı oluşturarak işe başlayalım.

```
CREATE TABLE sunucular
(
    id      serial PRIMARY KEY,
    addr    inet NOT NULL,
    bilgi   varchar
);
CREATE UNIQUE INDEX sunucular_addr_idx ON sunucular (addr);

CREATE TABLE ping_istatistikleri
(
    tarih       timestamp without time zone NOT NULL,
    kaynak      bigint REFERENCES sunucular (id),
```

24 ICMP ping istatistikleri bir ağın trafiği hakkında bize gerekli seviyede bilgi ulaştırmamakla birlikte çoğu sistem ICMP protokolüne doğrudan kapalıdır. Fakat bu örnekte konunun anlaşılması açısından böyle ilkel bir yöntem seçilmiştir.

```
    hedef       bigint REFERENCES sunucular (id),
    ttl         int2 DEFAULT 255,
    pingz       real[],
    paketist    real[3],
    harcananz   int4,
    rtt         real[4]
);
CREATE UNIQUE INDEX ping_istatistikleri_tarih_idx ON ping_istatistikleri (tarih);
```

Tabloların ne tür veri içerdiğine dair örnek bir çıktıyı şu şekilde verebiliriz:

```
-- Sunucu tablosundan kullanılan kesitin çıktısı.
=> SELECT * FROM sunucular;
 id |      addr      |           bilgi
----+----------------+--------------------------
...
 13 | 195.142.106.12 | Sistem Odası 1
 14 | 195.142.104.17 | Sistem Odası 2
...
(17 rows)

=> \x
Expanded display is on.
=> SELECT * FROM ping_istatistikleri WHERE kaynak = 13 AND hedef = 14;
...
-[ RECORD 184 ]-------------------------
tarih     | 2005-10-30 15:46:19.14566      -- ping işleminin başlangıç tarihi
kaynak    | 13                             -- Kaynak sistem
hedef     | 14                             -- Hedef sistem
ttl       | 255                            -- TTL (Time To Leave) değeri
pingz     | {1.92,1.51,1.42,1.48,1.52}     -- ping zamanları
paketist  | {5,5,0}                        -- Gönderilen/Alınan/Kaybedilen paket sayısı
harcananz | 7230                           -- Program tarafından harcanan zaman
rtt       | {1.42,1.57,1.52,0.181}         -- RTT (Round To Trip) zamanları:
                                           -- Minimum/Ortalama/Maksimum/Sapma
...
```

Şimdi elimizde bulunan bu tablo için şu sonuçları ekrana basacak sorgulama işlemlerini yazacağız:

- Geçen hafta içinde ortalama `RTT` değeri 2 saniyenin üstünde olan sunucular:

```
SELECT
  tarih,
  (SELECT addr FROM sunucular WHERE id = kaynak) as kaynak,
  (SELECT addr FROM sunucular WHERE id = hedef) as hedef,
  ttl, pingz, paketist, harcananz, rtt
  FROM ping_istatistikleri
  WHERE tarih > (now() - '1 week'::interval)
  AND rtt[1] > 2;
```

- Son 4 gün içinde kaybedilen paket sayısı oranının %50'den fazla olduğu sunucular:

```
SELECT
  tarih,
  (SELECT addr FROM sunucular WHERE id = kaynak) as kaynak,
  (SELECT addr FROM sunucular WHERE id = hedef) as hedef,
  ttl, pingz, paketist, harcananz, rtt
  FROM ping_istatistikleri
  WHERE tarih > (now() - '4 days'::interval)
  AND paketist[2] > 50;
```

Elimizdeki verileri ve bu verilerden çıkarılması gereken sonuçları oluşturduğumuza göre, artık ilgili Python betiğimizi yazabiliriz.

```
#!/usr/bin/python
# -*- encoding: iso-8859-9 -*-

# gerekli modülleri yüklüyoruz
from psycopg2 import connect
```

```
from sys import exit
from string import join

# DSN (Data Source Name) atanıyor
DSN = “dbname=test”
print "Veritabanı ile bağlantı kuruluyor... (DSN: %s)" % (DSN, )
try:
    bag = connect(DSN)
except StandardError, hata:
    print "Veritabanı ile bağlantı kurulurken beklenmeyen bir hata oluştu!"
    print hata
    exit(1)

# Veritabanı ile sorgulamalarımızı gerçekleştirecek
# olan imleç (cursor) oluşturuluyor.
imlec = bag.cursor()

def Sorgula(imlec, komut):
    """ Belirtilen imleci kullanarak veritabanı üzerinde ilgili
    sorgulamayı gerçekleştirecek olan fonksiyon. """

    try:
        imlec.execute(komut)
    except StandardError, hata:
        print "İlgili sorgulamayı gerçekleştirirken hata oluştu!"
        print "Sorgu tümcesi:", komut
        print "Dönen hata mesajı:", hata
        exit(1)

def TabloyuDok(imlec):
    """ Üzerinde sorgulama gerçekleştirilmiş imleçten sonuçları alarak,
    ekrana bir tablo şeklinde bu sonuçları dökecek olan fonksiyon. """

    # Dönen sonuç tablosunun başlıkları imleç üzerinden okunup, satırlar
    # bu imleçleri baz alan sözlükler (dictionary) şeklinde oluşturuluyor.
    basliklar = [b[0] for b in imlec.description]
    satirlar = [dict(zip(basliklar, satir)) for satir in imlec.fetchall()]

    if len(satirlar) == 0:
        print "[Sonuç yok.]"
        return

    sayac = 1
    for satir in satirlar:
        print "[%d]" % (sayac, )
        print "  Tarih                              :", satir["tarih"]
        print "  Kaynak Sistem                      :", satir["kaynak"]
        print "  Hedef Sistem                       :", satir["hedef"]
        print "  TTL                                :", satir["ttl"]
        pingz = join(["%.3f" % (i, ) for i in satir["pingz"]], ' ')
        print "  ping Zamanları (ms)                :", pingz
        paketist = join(["%.3f" % (i, ) for i in satir["paketist"]], ' ')
        print "  Gönderilen Paket Sayısı            :", paketist[0]
        print "  Alınan Paket Sayısı                :", paketist[1]
        print "  Kaybedilen Paket Yüzdesi           :", paketist[2]
        print "  Harcanan Zaman (ms)                :", satir["harcananz"]
        rtt = join(["%.3f" % (i, ) for i in satir["rtt"]], ' ')
        print "  RTT Minimum/Ortalama/Maksimum/Sapma:", rtt
        print ""
        sayac += 1

Sorgula(imlec,
        "SELECT"
        "  tarih,"
        "  (SELECT addr FROM sunucular WHERE id = kaynak) as kaynak,"
        "  (SELECT addr FROM sunucular WHERE id = hedef) as hedef,"
        "  ttl, pingz, paketist, harcananz, rtt"
        "  FROM ping_istatistikleri"
        "  WHERE tarih > (now() - '1 week'::interval)"
```

```
        "  AND rtt[1] > 2")
print ""
print "Geçen hafta içinde ortalama RTT değeri 2 saniyenin üstünde olan sunucular"
print "-------------------------------------------------------------------------"
TabloyuDok(imlec)

Sorgula(imlec,
        "SELECT"
        "  tarih,"
        "  (SELECT addr FROM sunucular WHERE id = kaynak) as kaynak,"
        "  (SELECT addr FROM sunucular WHERE id = hedef) as hedef,"
        "  ttl, pingz, paketist, harcananz, rtt"
        "  FROM ping_istatistikleri"
        "  WHERE tarih > (now() - '4 days'::interval)"
        "  AND paketist[2] > 50")
print ""
print "Geçen dört gün içinde paket kaybı %50 değerini aşan sunucular"
print "-------------------------------------------------------------"
TabloyuDok(imlec)
```

Yukarıdaki satırlardan da anlaşılabileceği üzere, `psycopg2` yazan satırı herhangi bir başka DB-API 2.0 uyumlu modül ile değiştirdiğiniz taktirde (sorgulamalarınız da standart bir yapıya sahipse) programınız hiçbir sorun çıkarmadan ilgili veritabanı üzerinde de çalışacaktır. Sanırız yazılan betiğin bu derece işlevsel olmasını, Python'un oldukça sağlam sınırlar ile çevrilmiş sözdizim yapısı ve modüller üzerine getirdiği standatlara borçluyuz.

Yukarıdaki betikte izlediğimiz adımları şu şekilde kısaca açıklamamız mümkün:

1. İlk önce `psycopg2` modülünden `connect` nesnesini çektik.
2. Ardından bunu kullanarak veritabanı ile girdiğimiz `DSN` doğrultusunda bir bağlantı gerçekleştirmeye çalıştık.
3. Veritabanı üzerinde ilgili işlemlerimizi gerçekleştirmek için, bağlantı nesnesini kullanarak bir imleç (*cursor*[25]) oluşturduk.
4. Yapılacak sorgulamaların hata denetimini daha kolay hale getirmek açısından `Sorgula()` şeklinde bir fonksiyon oluşturup, bunun içine girilen *SQL* tümcesini belirtilen imleç üzerinde çalıştıracak olan satırları yerleştirdik.
5. `ping_istatistikleri` tablosundan dönecek sonuçları düzenli bir şekilde ekrana yansıtacak `TobloyuDok()` fonksiyonunu oluşturduk. Bu fonksiyonun içinde, `fetchall()` ile aldığımız sonuçları `description` içinde yer alan tablo başlıkları ile birleştirip bir sözlük (*dictionary*) yapısı oluşturduk ve iterasyonu bu sözlükler dizisi üstünde gerçekleştirerek, tablo sütunlarına sözlüklerin anahtar değerleri vasıtası ile ulaştık.

Betiğin çalışması esnasında ekrana gelecek örnek bir çıktıyı şu şekilde verebiliriz:

```
Veritabanı ile bağlantı kuruluyor... (DSN: dbname=test)

Geçen hafta içinde ortalama RTT değeri 2 saniyenin üstünde olan sunucular
-------------------------------------------------------------------------
...
[13]
  Tarih                           : 2005-09-12 15:46:19.145660
  Kaynak Sistem                   : 195.142.109.49
  Hedef Sistem                    : 195.142.109.47
  TTL                             : 255
  ping Zamanları (ms)             : 2.41 2.959 2.958 0.938 0.99
  Gönderilen Paket Sayısı         : 5
  Alınan Paket Sayısı             : 5
  Kaybedilen Paket Yüzdesi        : 0
  Harcanan Zaman (ms)             : 9023
  RTT Minimum/Ortalama/Maksimum/Sapma: 0.938 2.051 2.959 0.181
...
```

25 Python *DB-API* tanımında yer alan `cursor` ifadesinin, veritabanında kullanılan `CURSOR` ile bir ilişkisi yoktur. (Bu konu hakkında ileriki bölümde ayrıntılı olarak bahsedilecektir.)

```
Geçen dört gün içinde paket kaybı %50 değerini aşan sunucular
-------------------------------------------------------------
...
[3]
  Tarih                             : 2005-10-30 15:46:19.145660
...
  Kaybedilen Paket Yüzdesi          : 100
...
```

4. Modül Arayüzü

DB-API v2.0 ile belirlenen modül arayüzü özellikleri, Python için yazılan herhangi bir veritabanı modülü için çeşitli fonksiyon çağrılarını ve sabit değişkenlerini gerektirmektedir. Bu bölümde bu özellikleri incelemeye çalışacağız.

İlk önce bağlantı nesnesini ele alarak başlıyoruz:

connect(dsn)

`connect()`, *DB-API* v2.0 uyumlu modüller arasında en çok uyumsuzluk gösteren çağrı olmasına rağmen aldığı ilk parametre olan `dsn` (*Data Source Name*) genellikle tüm modüller için sabittir.

> *`connect()` fonksiyonu, libpq kütüphanesindeki `PQconnectdb()` fonksiyonuna bir arayüz sağladığından, `dsn` olarak girebileceğiniz değerler, `PQconnectdb()` fonksiyonuna geçeceğiniz değerler ile aynıdır.*

`connect()` fonksiyonu için örnek bir kullanım şu şekilde verilebilir:

```
>>> bag = psycopg2.connect("dbname=test")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
psycopg.OperationalError: could not connect to server: )V
        Is the server running locally and accepting
        connections on Unix domain socket "/var/run/postgresql/.s.PGSQL.5432"?

>>> bag = psycopg2.connect("host=/tmp dbname=test")
>>>
```

Bir *DB-API* v2.0 uyumlu Python veritabanı modülü bünyesinde çeşitli sabitleri barındırmak zorundadır. Bu sabitler ve anlamlarını şu şekilde listeleyebiliriz:

apilevel

İlgili modülün *DB-API PEP*'inin hangi sürümünü desteklendiğini göstermektedir. Olası çıktı sonuçları (şuan için) 1.0 ve 2.0'dan ibarettir.

psycopg, çok eski sürümler kullanılmadığı sürece `apilevel` olarak 2.0 seviyesindedir.

threadsafety

Modülün yiv korunaklılığının (*thread safety*) seviyesini belirtir. Burada yer alabilecek değerler ve açıklamaları şu şekilde:

0	Yivler kendi aralarında modülü paylaşamazlar. Yani herhangi bir kilitleme mekanizması (örneğin `mutex`[26] yapısı) kullanılmadığı taktirde, aynı anda iki farklı yiv aynı modüle erişemezler.
1	Yivler - varolan bağlantı dışında - modülü paylaşabilirler.

26 `mutex`'ler, *Mutual Exclusion* (Müşterek Soyutlama) teriminin kısaltması olup yivli programlamada kullanılan bir kavramdır.

2	Yivler – varolan bağlantılar da dahil olmak üzere – modülü paylaşabilirler.
3	Yivler – varolan bağlantılar ile imleçler de dahil olmak üzere – modülü paylaşabilirler.

psycopg, 2 dereceden yiv korunaklı (*thread safe*) bir yapı sunar.

Bağlantı ve imleçler arası `transaction`'larda kullanılacak yiv korunaklılığı ile ilgili olarak Ek 1.4'te incelenen İzolasyon Seviyesi başlığına bakabilirsiniz.

`paramstyle`

Sorgu gönderiminde kullanılabilecek parametre stilini belirtir. Olası değerler şu şekilde:

`qmark`	Soru işareti şeklinde: `... WHERE name=?`
`numeric`	Numaralar ile indislenmiş şekilde: `... WHERE name=:1`
`named`	Değişken adı şeklinde: `... WHERE name=:name`
`format`	ANSI C `printf()` fonksiyonu biçiminde: `... WHERE name=%s`
`pyformat`	Python değişken isimleri ile birlikte: `... WHERE name=%(name)s`

psycopg, parametre stili olarak `pyformat` biçimini kullanmaktadır.

Modül tarafından üretilebilecek istisnasi durumların (*exception*) hiyerarşik bir tablosunu şu şekilde verebiliriz:

- `StandardError`
  - `Warning`
  - `Error`
    - `InterfaceError`
    - `DatabaseError`
      - `DataError`
      - `OperationalError`
      - `IntegrityError`
      - `InternalError`
      - `ProgrammingError`

5. Bağlantı Nesneleri

Bu bölümde oluşturulan bir bağlantı nesnesi tarafından sağlanan modül çağrıları hakkında bahsetmeye çalışacağız.

`close()`

Varolan bir bağlantının kapatılması için close() çağrısını kullanabilirsiniz. Bir bağlantı kapatıldıktan sonra, onu ya da ondan türemiş imleçleri kullanmaya çalışacak herhangi bir işleme Error durumu döndürülecektir.

Transaction destekleyen veritabanlarının genel özelliği olarak, herhangi bir transaction bloğu `commit` edilmeden önce kesilirse, tüm işlemler `rollback` edilip geri alınacaktır. Aynı durum, `commit` edilmemiş bir transaction bulunduğu bağlantı nesnesinin `close()` cağrısı ile kapatılmasında da meydana gelecektir.

`commit()`

Geçerli bir bağlantı üzerindeki bekleme konumundaki tüm transcation bloklarının `commit` edilmesi için `commit()` çağrısını kullanabilirsiniz.

rollback()

`rollback()` çağrısı, geçerli veritabanı bağlantısı üserinde bekleme konumundaki tüm transaction işlemlerinin `rollback` edilip geri alınmasını sağlar.

cursor()

Bağlantı üzerinde yeni bir imleç nesnesi oluşturmak için `cursor()` çağrısını kullanabilirsiniz.

psycopg bağlantı nesnesi tarafından sağlanan imleçlerin, SQL sorguları ile tanımlanan `CURSOR` ile bir ilgisi yoktur. Kavram olarak ikisi de aynı işleyiş yapısına sahip olsalar da, psycopg tarafından oluşturulan imleçler istemci tarafında organize edilirken, SQL sorguları ile oluşturacağımız imleçler sunucu tarafında tutulacaktır.

6. İmleç Nesneleri

Geçerli bir bağlantı üzerinde gerçekleştirilen `cursor()` çağrısı ile oluşturulan bir imleç nesnesi çeşitli fonksiyonlar ve değişkenler ile birlikte bazı diğer özellikleri de bünyesinde barındırmaktadır. Bu bölümde bahsi geçen bu özelliklerin nitelikleri hakkında bahsedeceğiz.

description

`description` sabiti, imleç üzerinde gerçekleştirilen en son sorgu hakkında 7 elemandan oluşan bir `tuple` (salt okunabilir dizi) listesi tutar. Bu 7 eleman ise indisleri ile birlikte sırasıyla şu bilgilere sahip olacaktır:

İndis	İçerik	libpq Çağrısı
0	Sütun/Alan adı.	`PQfname()`
1	Alan tipinin iç gösterimdeki kod numarası.	`PQftype()`
2	Gösterim boyutu.	
3	İç boyutu.	`PQfsize()`
4	Basamak duyarlılığı.	`PQfmod()`
5	Boyut sınırı.	`PQfmod()`
6	Alan `NULL` destekliyor mu?[27]	

`rowcount`

`rowcount` sabiti, ilgili imleç üzerinde gerçekleştirilen en son sorgu sonucuda dönen toplam satır sayısını içerir. `UPDATE`, `DELETE` veya `INSERT` türü sorgulamalarda toplam etkilenen satır sayısını (libpq kütüphanesindeki `PQcmdTuples()` fonksiyonunu kullanarak) döndürecektir.

callproc(prosedür[,parametreler])

Veritabanı üzerinde yer alan bir prosedürü belirtilen parametreler ile çağırmak için `callproc()` fonksiyonunu kullanabilirsiniz.

psycopg bu işlemi gerçekleştirirken verilen prosedürü girilen parametreler ile birleştirerek bağlantı üzerinde basit bir `SELECT prosedür(parametreler)` sorgusu gerçekleştirir.

27 Bu özellik şu an için psycopg tarafından desteklenmemektedir.

close()

Belirtilen imlecin kapatılması için imleç nesnesi üzerinde `close()` cağrısı gerçekleştirebilirsiniz.

execute(sorgu[,parametreler])

Veritabanı üzerinde herhangi bir sorgu işletiminde bulunmak için imleç nesinesi üzerinde `execute()` çağrısını gerçekleştirebilirsiniz.

> *İmleç nesnesi üzerinde yapılacak bir `execute()` çağrısının asenkron gerçekleşmesi için, en sona parametre olarak `async=1` eklemek yeterli olacaktır. (Bu özellik DB-API tanımlarının dışında kalan bir psycopg özelliğidir.)*

executemany(sorgu,parametre_dizisi)

Veritabanı üzerinde aynı sorguyu birden fazla kez farklı parametreler ile gerçekleştirmek için imleç nesnesi üzerinde `executemany()` fonksiyonunu kullanabilirsiniz. Bunun için sorgu tümcesini ilk parametre olarak girdikten sonra, hangi parametreler ile çalıştırılacağını her bir çalıştırmaya bir dizi karşılık gelecek şekilde bir diziler listesi girerek gerçekleştirebilirsiniz.

fetchone()

İmleç üzerinde gerçekleştirilen en son sorgu sorgu işleminden sonra dönen sonuç setinden bir satırı kendisini çağırana döndürür. Ve bir sonraki çağrılışında, döndürdüğü satırdan bir sonraki satırı döndürecek şekilde konumlanır. Okunacak herhangi bir satır kalmadığı durumda fonksiyon `None` değeri döndürecektir.

> *Bundan sonra aynı sonuç seti üzerinden sonuç çekmeye çalışacak olan çağrılar da çekilen satırları göz ardı edip, en son kalınan yerden sonuçları döndürmeye başlayacaklardır.*

fetchmany([size=cursor.arraysize])

`fetchmany()` fonksiyonu, imleç üzerinde gerçekleştirilen en son sorgu sonucunda dönen tablodan istenen satır kadar veri okunmasını sağlar. Herhangi bir parametre girilmediği taktirde öntanımlı olarak imlecin `arraysize` sabiti kadar satır okuyacaktır. Fonksiyon herbir satır bir dizi belirtecek şekilde bir liste döndürecektir.

> *cursor.arraysize öntanımlı olarak 1 değerine sahip olacaktır.*

Parametre olarak girilen satır sayısı, tabloda okunmadan kalan satır sayısından fazla olduğu durumlarda, fonksiyon kalan satır sayısı kadar satır döndürecektir. Okunacak satır kalmadığı durumda fonksiyon boş bir liste döndürecektir.

fetchall()

Bir imleç üzerindeki bütün (okunmadan kalan) satırları almak için fetchall() fonksiyonunu kullanabilirsiniz. `fetchmany()` fonksiyonu, `fetchall()` fonksiyonuna benzer olarak satırları içeren bir liste döndürecektir – her satır bir dizi belirtecek şekilde.

Şu ana kadar bahsi geçen imleç özelliklerini kısa bir örnek altında toplamaya çalışalım:

```
# Veritabanı ile bağlantı kurup, kullanmak üzere bir imleç nesnesi oluşturuyoruz.
bag = psycopg2.connect('dbname=test')
imlec = bag.cursor()

# Ekrana dökeceğimiz tabloyu getirecek sorgulama gerçekleştiriliyor.
try:
```

```
    imlec.execute("""SELECT objoid, description FROM pg_description
                     WHERE objoid < 100 AND objoid > 70""")
except psycopg2.Error, HataMsj:
    print "İmleç üzerinde ilgili sorgulamayı gerçekleştirirken hata oluştu!"
    print "Hata mesajı:", HataMsj
    exit(1)

# İstediğimiz kadar sonuç dönmüş mü acaba?
sayac = imlec.rowcount
if sayac == 0:
    print "Sorgu sonucu herhangi bir satır dönmedi!"
    exit(0)
else:
    print "Sorgu sonucu dönen satır sayısı:", sayac

print "İlk satır:"
print “ ”, `imlec.fetchone()`

print "İlk satırdan sonraki 4 satır:"
liste = imlec.fetchmany(4)
for satir in liste:
    print " ", `satir`

print "Kalan satırları yazdırıyoruz:"
liste = imlec.fetchall()
for satir in liste:
    print " ", `satir`


# Son olarak basit bir executemany() kullanımı.
imlec.executemany(“SELECT %(v1)s = %(v2)s”, [{'v1':1, 'v2':2}, {'v1':3, 'v2':3}])
# executemany() bizim yerimize şu sorguları sırasıyla gerçekleştirdi:
# SELECT 1 = 2;
# SELECT 3 = 3;
```

Programın örnek bir çıktısı ise şu şekilde:

```
Sorgu sonucu dönen satır sayısı: 9
İlk satır:
  (99, 'Reserved schema for TOAST tables')
İlk satırdan sonraki 4 satır:
  (72, 'less-than-or-equal')
  (73, 'greater-than')
  (74, 'greater-than-or-equal')
  (77, 'convert char to int4')
Kalan satırları yazdırıyoruz:
  (78, 'convert int4 to char')
  (79, 'matches regex., case-sensitive')
  (84, 'not equal')
  (89, 'PostgreSQL version string')
```

7. Veri Tipi Geçişleri

Farklı veritabanlarına çeşitli veri tiplerinin girişinde farklı sözcük dizimlerine ihtiyaç duyulmaktadır. Veyahut tam tersi olarak veritabanından okunan bir alanın, programlama dilindeki eşdeğer tipe dönüştürülmesi hep bir problem oluşturmuştur. Bu bölümde bunu Python programlama dili ile PostgreSQL veritabanına bağlanırken kullandığımız psycopg veritabanı bağdaştırıcısı ile nasıl gerçekleştirebileceğimiz üzerinde duracağız.

Farklı veritabanlarının veri girişlerinde farklı söz dizimine ihtiyaç duyduklarından bahsetmiştik. Bu konunun tam olarak netleşmesi için basit bir örnek verelim. PostgreSQL veritabanındaki `date` tipine sahip bir alana hangi sözdizimleri ile bir tarih girebileceğiniz PostgreSQL dökümantasyonundaki Veri Tipleri (*Data Types*) başlığı altındaki Tarih/Zaman Tipleri (*Date/Time Types*) bölümünde belirtilmiştir. Buradakilerin dışında bir sözdizimi ile

`date` tipindeki bir alana veri girmeye çalıştığınızda sorgu hata döndürecektir. Diğer bir taraftan, başka bir veritabanı sunucusunun bu sözdizim yapılarının dışında bir kurgulama ile tarih girişine olanak sağladığını düşünelim. Peki biz her iki veritabanına da tek standart bir sözdiziminde veri girmek istersek? Bunun için Python veritabanı bağdaştırıcıları tarafından sağlanan veri tipi dönüştürücülerini kullanabiliriz. Bu dönüştürücüler sayesinde girdiğimiz tek bir sözdizim yapısı, bir çok veritabanı tarafından sorunsuzca anlaşılacaktır.

DB-API v2.0 ile belirlenmiş bahsi geçen bu dönüştürücüleri şu şekilde listeleyebiliriz:

`Date(yıl,ay,gün)`	Belirtilen yıl, ay ve gün değerlerine sahip tarih girdisini oluşturur.
`Time(saat,dakika,saniye)`	Belirtilen saat, dakika ve saniye değerlerine sahip zaman girdisini oluşturur.
`Timestamp(yıl,ay,gün,saat,dakika,saniye)`	Belirtilen yıl, ay, gün, saat, dakika ve saniye değerlerine sahip zaman bilgisini oluşturur.
`DateFromTicks(süreç)`	Belirtilen epoch sürecine sahip tarihi oluşturur.
`TimestampFromTicks(süreç)`	Belirtilen epoch sürecine sahip tarih bilgisini oluşturur.
`Binary(katar)`	Girilen veriyi veritabanın ikili veri tipinin algılayabileceği biçime dönüştürür.

Ek olarak DB-API v2.0 uyumlu veritabanı bağdaştırıcılarında şu veri tiplerinin bulunması gerekmektedir:

`STRING`	Karakter katarı bazlı veri tipleri için. (`CHAR` gibi.)
`BINARY`	İkili veri tipleri için. (`BYTEA`, `BLOB`, `RAW` gibi.)
`NUMBER`	Sayısal tipler için.
`DATETIME`	Tarih/Zaman tipleri için.
`ROWID`	Satır `ID` değerlerinin (numaralarının) gösterimi için.

Yukarıdaki tip sınıflarına ek olarak, `NULL` alanı Python diline özgü `None` tipinde gösterilecektir.

```
# Gireceğimiz tarih için PostgreSQL tarafından algılanamayacak,
# fakat günlük hayatta çok sık kullanılan bir biçim kullanıyoruz.
>>> imlec.execute("SELECT '20.10.2005'::date")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
psycopg2.ProgrammingError: date/time field value out of range: "20.10.2005"
HINT:  Perhaps you need a different "datestyle" setting.

# Girilecek tarihi psycopg'yi kullanarak PostgreSQL tarafından algılanabilecek
# bir hale soktuktan sonra sorgu tümcesine yerleştiriyoruz.
>>> imlec.execute("SELECT %s::date" % (psycopg2.Date(2005,10,20), ))
>>> sonuc = imlec.fetchone()

# Bakalım psycopg aldığı tarihi Python'a nasıl aktarmış?
>>> print sonuc
(datetime.date(2005, 10, 20),)
>>> print sonuc[0]
2005-10-20

# Farklı tipler içerecek türden bir sonuç
# döndürecek bir sorgulama gerçekleştiriyoruz.
>>> imlec.execute("SELECT 'abc'::text, 12::integer, '2005-11-20'::timestamp")

# Sorgulama sonucunda dönen alan bilgilerini yazdırıyoruz.
>>> for name, type, width, ds, p, scale, null_ok in imlec.description:
...     print "Name:", name
...     if type == psycopg2.STRING: type = "STRING"
...     elif type == psycopg2.NUMBER: type = "NUMBER"
...     elif type == psycopg2.DATETIME: type = "DATETIME"
...     print "Type:", type
...
```

```
Name: text
Type: STRING
Name: int4
Type: NUMBER
Name: timestamp
Type: DATETIME
```

Konu ile ilgili olarak buraya kadar bahsedilenlerin dışında, veritabanı tarafından desteklenen tiplerin programalama dili içerisindeki eşdeğerlerine dönüştürülmesi (ya da eşdeğerlerinin oluşturulması) bu bölümde incelenmeyecektir. Bu dönüşüm işlemleri hakkında ayrıntılı bilgi için Sık Karşılaşılan Problemler bölümdeki Python başlığı altında incelenen ilgili örneğe bakabilirsiniz.

8. Genişletilmiş *DB-API* Özellikleri

DB-API v2.0 ile tanımlanan genel özelliklere ek olarak, aynı standartta bir kaç farklı genişletilmiş özellik daha yer almakta. Bu bölümde bu özelliklerden psycopg tarafından desteklenenler hakkında bahsetmeye çalışacağız.

`cursor.rownumber`

Geçerli bir imlecin şuan hangi satırda bulunduğu öğrenmek için imleç nesnesinin `rownumber` sabitine bakabilirsiniz.

`cursor.connection`

Herhangi bir imlecin hangi bağlantıdan türediğini öğrenmek için ilgili imlecin `connection` sabitine bakabilirsiniz.

`cursor.scroll``(adım[,mode='relative'])`

Herhangi bir imlecin bulunduğu sonuç üzerindeki satır konumunu belirli bir noktaya kaydırmak için imleç üzerinde `scroll()` çağrısını gerçekleştirebilirsiniz. Fonksiyon, ilk parametre olarak girilen adım sayısı kadar satır ilerleyecektir. `mode` değeri `relative` girildiği taktirde başlangıç konumu olarak en son yer aldığı satırı seçecektir, `absolute` girildiği taktirde ise sonuç tablosunun ilk satırını başlangıç konumu olarak alacaktır.

`cursor.next``()`

İmleç nesnesi üzerinde yapacağınız `next()` çağrısı `fetchone()` ile denk işleve sahiptir. Tek fark olarak, sonuç setinin sonuna ulaşıldığında `StopIteration` uyarısı (*exception*) verilecektir.

9. Standart Olmayan psycopg Özellikleri

psycopg, DB-API v2.0 PEP'ine ek olarak beraberinde bazı standart olmayan özellikler de getirmektedir. Bu bölümde bunları gerektiği kadar incelemeye çalışacağız.

> *Yüzeysel olarak geçilecek olarak tanımların ötesinde, daha ayrıntılı bilgi için psycopg'nin kaynak koduna bakmanızı tavsiye ederiz. Yazarın kod aralarına düştüğü açıklayıcı satırlar doğru takip edildiği zaman konu hakkında oldukça doyurucu yanıtlar sunuyor.*

9.A. Veri Tipleri

Veri Tipi Geçişleri bölümünde tanıttığımız veri tipleri dışında, psycopg kendi içinde PostgreSQL tarafından tanımlanmış birçok veri tipini ifade edebilmektedir. Bunları şu şekilde listelememiz mümkün:

```
BINARYARRAY          FLOAT                ROWIDARRAY
BOOLEAN              FLOATARRAY           STRINGARRAY
BOOLEANARRAY         INTEGER              TIME
DATE                 INTEGERARRAY         TIMEARRAY
DATEARRAY            INTERVAL             UNICODE
DATETIMEARRAY        INTERVALARRAY        UNICODEARRAY
DECIMAL              LONGINTEGER
DECIMALARRAY         LONGINTEGERARRAY
```

Kullandığınız psycopg sürümüne göre bu liste farklılık gösterebileceğinden, tam bir çıktı için Python komut satırından `dir(psycopg2._psycopg)` çıktısına bakabilirsiniz.

9.B. İstemci Karakter Kodlaması

PostgreSQL sunucusuna bağlanırken istemci karakter kodlamasını görmek ya da ayarlamak için

```
SHOW client_encoding;
SET client_encoding TO <ENCODING>;
```

SQL sorgu tümcelerini kullanabileceğiniz gibi geçerli bağlantı nesnesi üzerinde şu çağrıları gerçekleştirebilirsiniz:

```
# bag, geçerli bir psycopg2.connect nesnesini belirtmek üzere:
>>> print bag.encoding
LATIN5
>>> bag.set_client_encoding('UNICODE')
>>> print bag.encoding
UNICODE
```

9.C. Uyarı Mesajları

Veritabanından dönecek olan uyarı mesajlarının (*notices*) bir listesine bağlantı nesnesi altındaki `notices` dizisinden ulaşabilirsiniz.

`NOTIFY` *SQL* tümcesi ile yapılan dinlemelerde oluşan asenkron uyarılma mesajlarına ise yine geçerli bir bağlantı nesnesinin `notifies` listesinden ulaşabilirsiniz. Ulaşacak olan `NOTIFY` mesajlarının ne zaman vardığı anlamak için ilgili `cursor` nesnesini `select()` ile dinlemeniz yeterli.

Bunun ile ilgili bir örneği psycopg'nin kaynak kodu altındaki `examples/notify.py` dosyasında bulabilirsiniz.

9.D. İzolasyon Seviyesi

Bağlantı nesnesi üzerinde tanımlı izolasyon seviyesini[28] öğrenmek ve bu değeri değiştirmek için psycopg tarafından sağlanan şu özellikleri kullanabilirsiniz:

```
# İzolasyon seviyesi 0 ile 3 arasındaki değerleri alabilir.
```

28 İzolasyon seviyesi hakkında ayrıntılı bilgi için kitabın `transaction` ile ilgili bölümüne bakınız.

```
>>> print bag.isolation_level
1

# İzolasyon seviyesini en yüksek meretebeye yükseltiyoruz.
>>> bag.set_isolation_level(3)
```

9.E. İmleç Fonksiyonları

Geçerli bir `cursor` nesnesi üzerinde *DB-API* v2.0 *PEP*'ine ek olarak psycopg bir kaç artı özellik daha tanımlamaktadır. Bunların ufak bir listesini bu bölümde vermeye çalışacağız.

`copy_from``(dosya,tablo,ayrac='\t',null='\N')`

Herhangi bir dosyadaki – belirtilen ayraç ve yeni satır karakteri ile ayrılmış sütun ve satırları – belirli bir tabloya aktarmak için imleç nesnesi üzerindeki `copy_from()` fonksiyonunu kullanabilirsiniz.

Parametre olarak geçilecek dosya nesnesi read() metoduna sahip olmalı.

`copy_to``(dosya,tablo,ayrac='\t',null='\N')`

Herhangi bir tablodaki satırları belirtilen satır ve sütun ayracı ile bir dosyaya yazmak için imleç nesnesi üzerindeki `copy_to()` çağrısını kullanabilirsiniz.

Parametre olarak geçilecek dosya nesnesi write() metoduna sahip olmalı.

`fileno``()`

İmleç üzerinde gerçekleştireceğiniz `fileno()` çağrısı, imlecin bağlı olduğu bağlantı soketini kendisini çağırana döndürecektir. Bu değer ile bağlantı üzerinde istenilen soket çağrısı (`select()`, `poll()`, `read()`, `write()`, vs.) gerçekleştirilebilir.

psycopg soket numarasını libpq kütüphanesindeki PQsocket() fonksiyonundan alır.

`isready``()`

Asenkron olarak yapılan bir sorgulama sonucu dönecek değerlerin hazır olması durumunda `isready()` çağrısı `True`, aksi halde `False` değeri dönecektir.

`isready()` fonksiyonu ilk önce `PQisBusy()` değerine bakıp, bağlantının meşgul olmadığı durumda bağlantı soketine gelen verileri ilgili yapılara aktarır. Son olarak `PQisBusy()` fonksiyonundan aldığı durumu döndürür.

`mogrify``(sorgu, parametreler=None)`

Çalıştırılacak bir sorgunun ilgili parametreler psycopg tarafından yerine konduktan sonraki son halini görmek için `mogrify()` fonksiyonunu kullanabilirsiniz.

`statusmessage`

İmleç üzerinde gerçekleştirilen en son SQL sorgusunun veritabanından aldığı cevabı, ilgili imlecin `statusmessage` sabitinden öğrenebilirsiniz.

psycopg, statusmessage değerini oluşturulurken libpq kütüphanesindeki PQcmdStatus() fonksiyonundan yararlanmaktadır.

9.F. Diğer Özellikler

psycopg'nin yukarıda listelemeye çalıştıklarımızın dışında daha bir çok ek özelliği bulunmaktadır. Bunların bir kısmı için psycopg'nin kaynak kodu ile gelen `examples` dizini altındaki örneklere bakabilirsiniz.

psycopg2 kurulumu ile birlikte `psycopg2.extensions` modülünü de Python modülleri arasına yerleştirmektedir. Bu modül ve sağladığı özellikler hakkında burada bahsedilmeyecektir.[29]

29 Sadece psycopg2 ile birlikte gelen çok kullanışlı bir özellik olan Python Nesne Adaptasyonu ile ilgili bir örneği Sık Karşılaşılan Problemler kısmında vereceğiz.

V. Programlama ve Güvenlik

Bu bölümde, buraya kadar bahsi geçen PostgreSQL uygulama arayüzleri ile geliştirilecek olan yazılımlarda, yazılımın güvenliği açısından dikkat edilmesini gerekli gördüğümüz bir kaç temel konu hakkında bahsetmeye çalışacağız. Kapsamlı bir yazılım güvenliği incelemesi bu kitabın konusu dışına taştığından, sadece önceki bölümlerde incelenen arayüzler üzerinde durulmaya çalışılıp, PostgreSQL kullanarak geliştirilen yazılımların güvenlik kavramı dış hatları ile irdelenecektir.

1. Güvenli Programlama

Kullanılan kütüphanelerin ve diğer yardımcı araçların ötesinde, bir yazılımın güvenliği ilk önce kendi içindeki kararlılığına bağlıdır. Kendi içinde çelişik rekürans gönderimler ve eksik olasılık kontrolleri ile dolu bir program, her ne derleyici, ek kütüphane kullanılırsa kullanılsın içerisinde daima potansiyel güvenlik açıkları barındıracaktır. Benzer bakış açısı ile yaklaşıldığında, aynı şey, C, PHP ve Python kullanarak PostgreSQL veritabanına bağlanan uygulamalar için de geçerlidir. Bu programlama dilleri ve PostgreSQL veritabanında o an itibari ile hiçbir kapatılmamış açık olmamasına rağmen, yanlış yazılabilecek bir kod çok ciddi güvenlik problemlerine sebebiyet verebilir. Bu ve benzeri nedenlerden ötürü, bir uygulamanın güvenliğinde ilk olarak programlamada izlenecek yolun önemli olduğunu düşünerek, öncelikli olarak bunun üzerinde durmanın daha doğru olacağını düşündük.

1.A. C Programlama Dili

PostgreSQL veritabanına C programlama dili ile libpq kütüphanesi kullanılarak bağlanıldığında, doğabilecek olası programlama hatalarının önüne geçebilmek için programcının bir kaç temel noktaya dikkat etmesi gerekmektedir.

Günümüzde C/C++ kullanarak geliştirilen programlarda en sık rastlanan güvenlik açıklarından biri olan bellek taşmaları, bu dili kullanan programcılar tarafından özen gösterilmesi gereken en önemli noktalardan biridir. Özellikle SQL sorgu tümcelerinin birbirleri arasında (`sprintf()` ve türevi fonksiyonlar kullanarak) birleştirilmesinde gerekli bellek alanı doğru ayarlanmadığı taktirde, bellek taşmalarının doğacak olması kaçınılmazdır. Örneğin, bu tür problemlerin en sık ortaya çıktığı yerlerden biri, kullanıcı tarafından alınan herhangi bir girdinin sorgu komutu içine kontrolsüz yerleştirilmesi ile oluşur. Şöyle ki, yeterli boyutta bellek alanına sahip olmayan bir katar bloğuna, alabileceğinden daha fazla uzunlukta bir sorgu gönderimi bellek taşmasına sebep olacaktır. Bu gibi durumlardan kaçınmak için, herhangi bir sorgu katarının hazırlanmasında `sprintf()` yerine `snprintf()` gibi, kopyalanacak bayt uzunluğunun

belirtilebildiği fonksiyonları kullanabiliriz. Fakat kullanıcı taraflı potansiyel tehlikeler bellek taşmasından ibaret değildir. Diğer bir yandan, uygulama mantığının kullanıcıdan alınan girdide yer alacak olası *SQL Injection* saldırılarına karşı da savunma geliştirmiş olması gerekmektedir. (Ve bu savunmanın da beraberinde daha fazla bellek kontrolüne ihtiyaç duyması oldukça olasıdır.) Olayı daha da karmaşık bir hale sokmamak için, kullanıcı girdisi bir kez bellek alanında alıkoyulduktan sonra, sorgu katarı içinde kullanılacağı zaman, parametreler kullanan `PQexecParams()` tarzı fonksiyonlar tercih edilmelidir. Bu sayede bellek alanlarının teker teker elle sorgu katarı haline gelecek şekilde birleştirilmesi işleminden kurtulmuş olduğumuz gibi, *SQL Injection* saldırılarından yana da hiçbir açık nokta bırakmamış oluruz. (Parametre kullanımı ve *SQL Injection* saldırıları hakkında daha sonra bahsedilecektir.)

Bundan başka, fonksiyonlara geçirilen işaretçilerin gerek duyulan bellek alanına sahip olup olmadığı da özellikle kontrol edilmedilir. (libpq kütüphanesinde yer alan bazı fonksiyonların parametre olarak aldıkları bellek işaretçilerinin, en az belli büyüklükte bir yere gereksinim duyduklarını hatırlayınız.) Bunun için hangi fonksiyonun en az ne kadar yere ihtiyaç duyulduğu hesaplandıktan sonra, ona göre ihtiyaç duyulan bellek alanı alıkoyunup işleme devam edilmelidir.

1.B. PHP Programlama Dili

PHP, C/C++ dillerine oranla kullanıcı tarafına daha yakın bir dil olduğundan, olası yapılabilecek hataları bir nebze de olsa kendi kendine giderebilmektedir. Örneğin PHP ile yazdığınız bir yazılım da (çok ender durumlar dışında) hafıza taşmaları meydana gelmeyecektir. Çünkü bellek ayrımı gereken yerlerde PHP bunu kullanıcıya bırakmaktansa kendi yapmayı tercih edecektir. Şu da unutulmamalıdır ki, hiç bir programlama dili yoktur ki programcı tarafından meydana getirilebilecek her güvenlik açığını onarabilecek nitelikte olsun. Dolayısıyla bu noktada sorumluluğun büyük kısmı programcıya düşmektedir. PHP kullanarak geliştiridiğiniz uygulamalarda güvenli programlama hakkında daha ayrıntılı bilgi için, kitaplar ve internette bulabileceğiniz makaleler dışında, PHP paketi ile birlikte gelen dökümantasyonun güvenlik ile ilgili bölümüne (*IV. Security*) [1] göz atabilirsiniz. Ek olarak, PHP Güvenlik Konsorsiyumu (*PHPSEC, PHP Security Consortium*) [2] tarafından sağlanan PHP Güvenlik Kılavuzu'nun (*PHP Security Guide*) [3] da güvenli programlama konusunda her PHP programcısı tarafından bir kez de olsa okunması gerektiğini düşünüyoruz.

[1] http://www.php.net/manual/en/security.php
[2] http://phpsec.org/
[3] http://phpsec.org/projects/guide/

1.C. Python Programlama Dili

Python da PHP gibi sonradan yorumlanan dillerden olup üst seviye bir dil olduğu için, yukarıda PHP için yazdığımız bir çok uyarının Python için de geçerli olacağı aşikardır. Yukarıda yazılanlara ek olarak, geliştiriceğiniz Python uygulamalarında Python Geliştirme Önergelerine [1] (PEP/Python Enhancement Proposals) elden geldiğince bağlı kalınmasının hatadan kaçınmanın en temel yollarından biri olduğu görüşündeyiz.

[1] http://www.python.org/peps/

1.D. Dönen Sonuçların Kontrolü

Program akışı esnasında veritabanından dönen verilerin kontrol edilmesi de güvenlik açısından ayrı bir önem teşkil etmektedir. Yapılan her sorgulamanın sonucunda, dönen

sonucun ve bağlantının durumu ayrı ayrı kontrol edilip, olası beklenmeyen bir durum karşısında program akışına ne yönde devam edileceğine karar verilmedilir. Kimse hata vermiş bir *transaction* bloğuna ya da bağlantısı kopmuş bir veritabanına daha fazla sorgu göndermek istemez. Olası böyle bir durumda, gönderildiğini zannettiğimiz veri karşı tarafa ulaşmamışsa, verinin ulaştığı aksiyomu ile işe başlayacağımız her yolun bir noktasında mutlaka çıkmaza girilecektir. Bu gibi durumlardan kaçınmak için, yapılan her sorgulamanın sonucunda dönen sonuç yapısı beklenen sonuç ile karşılaştırılıp bağlantı durumu gözden geçirilmelidir. Bunu yaparken, dönecek olan sonucu her seferinde `if/else/switch` deyimleri ile teker teker kontrol etmek uğruna aynı satırları onlarca kez tekrarlamaktansa, bu kullanım için özelleşmiş bir fonksiyon oluşturularak bu işi otomatiğe bağlayabilirsiniz. Çok yüzeysel de olsa bir örnek vermek gerekirse:

```
PROCEDURE sorguCalistir(sorgu, beklenenSonucDurum, eylem1, beklenenBagDurum, eylem2)
    sonuc := sorgula(sorgu)
    IF durum(sonuc) != beklenenSonucDurum THEN
        EXECUTE eylem1
    ENDIF

    IF durum(baglanti) != beklenenBagDurum THEN
        EXECUTE eylem2
    ENDIF
END
```

1.E. Güvenilmeyen Kaynaklı Verilerin Kontrolü

Güvenilmeyen kaynaklardan (örneğin bir kullanıcının doldurduğu internet formundan) gelen verinin program içindeki ifadelerce direk kullanılmadan önce bazı kontrollerden geçirilmesi gerekmektedir. Aksi halde kasıtılı olarak gönderilen bir girdinin istenmeyen sonuçlara yol açacak olması oldukça muhtemeldir.

E.1. *SQL Injection* Saldırıları

Yeteri kadar kontrolden geçirilmemiş, güvenilmeyen kaynaklardan gelen bir verinin açacağı problemlerin arasında en sık karşılaşılanı *SQL Injection* saldırılarıdır. Bu tür bir saldırıda, alınan verinin yeterli derecede kontrol edilmeden doğrudan SQL sorgusu içinde veritabanı sunucusuna gönderilmesi, tasarlanan sistemce hiç beklenmedik bir sonuca neden olabilir. Bu sebeple istemcilerin kullanıcıdan girdi almak durumunda olduğu yerlerde, yapılan girdinin kendi SQL sorgumuz içinde sunucuya gönderilmeden önce bir takım kontrollerden geçmesi gerekir ki, herhangi bir *SQL Injection* saldırısına maruz kalmayalım. Konuyu bir örnek ile izah etmeye çalışalım.

Üzerinde çalışacağımız tablo ve kullanacağımız SQL sorgu katarımız şu şekilde olsun:

```
--
-- Tablomuzun nasıl bir şey olduğuna bakıyoruz.
--
=> \d kullanici_kayitlari
        Table "public.kullanici_kayitlari"
   Column    |          Type          | Modifiers
-------------+------------------------+-----------
 kullanici   | character varying(128) | not null
 sifre       | character varying(32)  | not null
 gizli_bilgi | text                   | not null
Indexes:
    "kullanici_kayitlari_pkey" PRIMARY KEY, btree (kullanici)

--
-- Tablo içeriğine göz atıyoruz.
--
=> SELECT * FROM kullanici_kayitlari;
```

```
 kullanici | sifre  |       gizli_bilgi
-----------+--------+-------------------------
 volkan    | sifrem | Banka karti bilgileri...
...

=> SELECT gizli_bilgi FROM kullanici_kayitlari
->   WHERE kullanici = '<kullanici>' AND sifre = '<sifre>';
```

Yukarıdaki `<kullanici>` ve `<sifre>` parametreleri kullanıcıdan gelecek olan “kontrolsüz” girdileri temsil etmektedir. Şimdi değişkenlerimiz ile biraz oynayalım.

```
--
-- <kullanici> = volkan
-- <sifre>     = yanlis_sifre
--
=> SELECT gizli_bilgi FROM kullanici_kayitlari
-> WHERE kullanici = 'volkan' AND sifre = 'yanlis_sifre';
 gizli_bilgi
-------------
(0 rows)
```

Peki ya kullanıcımız gereğinden fazla kurnazsa:

```
--
-- <kullanici> = volkan
-- <sifre>     = yanlis_sifre' OR kullanici = 'volkan
--
=> SELECT gizli_bilgi FROM kullanici_kayitlari
-> WHERE kullanici = 'volkan'
-> AND sifre = 'yanlis_sifre' OR kullanici = 'volkan';
       gizli_bilgi
--------------------------
 Banka karti bilgileri...
(1 row)
```

Bu tür bir saldırıdan kaçınmak için kullanıcıdan alınacak veri içerisinde, kullanımının belli kısıtlamalar gerektirdiği karakterler (`'` ve `\` gibi) programcı tarafından ayıkladıktan sonra, temizlenmiş girdi SQL sorgusu içinde sunucuya göndermelidir. Yukarıdaki örnekte `'` karakteri programcı tarafından ayıklansaydı saldırı etkisiz hale getirilebilecekti. Şöyle ki:

```
--
-- <kullanici> = volkan
-- <sifre>     = yanlis_sifre' OR kullanici = 'volkan
--
-- <sifre> değişkeni düzeltildikten sonra:
-- <sifre> = yanlis_sifre'' OR kullanici = ''volkan
-- (' yerine '' kullanıldığına dikkat ediniz.)
--
=> SELECT gizli_bilgi FROM kullanici_kayitlari
-> WHERE kullanici = 'volkan'
-> AND sifre = 'yanlis_sifre'' OR kullanici = ''volkan';
       gizli_bilgi
--------------------------
(0 row)
```

libpq kütüphanesi, bu tür *SQL Injection* saldırılarından korunmak için girdi katarının SQL tümcesi içinde zararsız bir şekilde yer almasını sağlayacak (kullanılacak veri tipine göre: ikili ya da karakter katarı) `PQescapeString()`, `PQescapeBytea()` ve `PQunescapeBytea()` ayıklama (*escape*) ve geri ayıklama (*unescape*) fonksiyonlarını sunmaktadır. (PHP ve Python *API*'lerinde de aynı fonksiyonların eşdeğerleri yer almaktadır.)

Tüm bunlara ek olarak, gönderilecek olan (ikili ya da düz metin) katar üzerinde değişiklik yapmadan da bu tür saldırılara karşı korunmanız mümkün. `PQexecParams()`, `PQexecPrepared()`, `PQsendQueryParams()` yada `PQsendQueryPrepared()` fonksiyonlarını kullanarak verinizi parametreler aracılığıyla gönderdiğinizde tüm bu karakter ayıklama zahmetinden kurtulabilirsiniz.

libpq kütüphanesi tarafından sağlanan fonksiyonların PHP dilinde karşılıkları için sonunda `params` sözcüğü geçen fonksiyonlara bakabilirsiniz. (`pg_query_params()` gibi.) Aynı fonksiyonların psycopg içindeki karşılığı ne yazık ki şuan için bulunmamaktadır. Fakat, bunun yerine psycopg'de sorgu tümcelerini oluştururken kullandığınız parametreler (`imlec.execute("SELECT * FROM t1 WHERE v = %s", ("' OR ''='", ))` gibi) psycopg tarafından otomatik olarak ayıklanmaktadır.

Parametre kullanan fonksiyonların bu tür *SQL Injection* saldırılarına karşı hiçbir korumaya gerek duymamalarının sebebi, parametre değerlerini direk komutun içine koyup ayıkladıktan sonra öylece gönderlemerindense, sunucuya parametreler halinde göndermelerinden kaynaklanmaktadır. Bunun dışında parametre kullanımının, ayıklama işlemine kıyasla sağladığı bir diğer artı ise, ayıklama esnasında gerek duyulacak işlemci ve bellek kullanımından feragat etmemizdir. (Konu hakkında basit bir örnek için örnekler bölümünde yer alan konu ile ilgili `sql_injection.c` dosyasına göz atabilirsiniz.)

PostgreSQL C API kütüphanesinin bahsi geçen parametre kullanım özelliği, 3.0 protokolü ile birlikte gelmiştir. Bu sebeple, 3.0 ve altı protokol kütüphanesi kullanan istemcilerde çalışmayacaktır.

Elbette, güvenilmeyen kaynaklardan gelen verinin yeterli kontrolden geçirilmeden çeşitli prosedürlerce işletilmesi sonucu doğacak tek saldırı tipi *SQL Injection* değildir. Benzer şekilde, aynı girdi daha bir çok yerde (dosya sistemindeki bir dosyanın yazılacağı esnada, dosya adının kullanıcıdan girilmesi istendiğinde; bir komut çalıştırılacağı zaman, çalıştırılacak komutun kullanıcıdan girilmesi beklendiğinde; vs.) karşımıza bir potansiyel güvenlik tehditi olarak çıkabilir. Fakat konunun bundan sonraki kısmı amaçlarımız dışına taşmaktadır.

1.F. *API* Güvenliği

Bir yazılımın güvenliliği için dikkat edilmesi gereken ilk şeyin güvenli programlama olduğunu vurgulamıştık. Bunun hemen ardından ise kullanılan bileşenlerin kendi içindeki güvenlikleri gelir. Bu bölümde ise PostgreSQL veritabanına bağlanırken kullandığımız arayüzlerin güvenliği üzerinde durmaya çalışacağız.

Yazılım piyasasında belli bir konuma sahip her uygulama, aktif gelişim sürecinde çeşitli potansiyel güvenlik açıkları bulundurabilir ve zamanla bunlar bulunup geliştirici ekip tarafından kapatılmaya çalışılır. Her yeni sürüm ile birlikte yeni özelliklerin geliyor olması ve bunun da yeterince test edilmemiş kod anlamına gelmesi, yazılımın hata bulundurma olasılığını yükseltse de bulunduracağını garantilemez; benzer şekilde yeni bir hata bulunduğu durumda ise, bu da yazılımın güvensiz olduğu fikrinin doğurmaz. Kısacası, gelişmekte olan bir yazılımda yeni güvenlik açıklarının tespiti ve bunların onarımı oldukça olağan bir süreçtir. Bu noktada yazılımın kararlılığını ise, birim zaman içinde bulunan hata sayısı, bunların ne derece öneme sahip oldukları ve bulunan bir hatanın uygulamanın gelişiminin ilerki safhalarında ne gibi bir rol oynayacağı belirler. Bu sebeplerden ötürü, PostgreSQL veritabanına bağlanırken kullandığımız arayüzlerde de zaman içinde çeşitli hatalarının bulunması muhtemeldir. Burada sistem yöneticisi ve programcı için anahtar kelime `güncellik'tir. Kullanılan programların en çok teste tabi tutulmuş ve son kararlı sürümleri kullanılmaya çalışılmalıdır.

F.1. C Arayüzü

libpq kütüphanesi PostgreSQL ile birlikte geliştirildiğinden dolayı, libpq ile ilgili bir güvenlik duyurusu olduğu zaman bunu PostgreSQL'in kendi anasayfasından [1] öğrenebilirsiniz. Ek olarak pgsql-bugs [3], bugtraq [4] listelerini ve SecurityFocus'da yer alan güvenlik açıklarını [5] takip ebilirsiniz. Bunun dışında daha ayrıntılı bir libpq değişim

göstergesi olarak CVS ağacını [6] kullanabilirsiniz.

[1] http://www.postgresql.org/
[2] http://archives.postgresql.org/pgsql-php/
[3] http://archives.postgresql.org/pgsql-bugs/
[4] http://seclists.org/lists/bugtraq/
[5] http://www.securityfocus.com/bid/
[6] http://developer.postgresql.org/cvsweb.cgi/pgsql/src/interfaces/libpq/

F.2. PHP Arayüzü

PHP tarafından sağlanan PostgreSQL uygulama arayüzünün, libpq kütüphanesini kullandığını daha önceden belirtmiştik. Bu yüzden libpq için bulunacak olan her açığın aynı zamanda PHP PostgreSQL *API*'sini de etkileyeceği aşikardır. Bunun dışında, PHP'nin libpq kütüphanesine arayüz sağlarken kullandığı kodda bir açık olduğu durumuda bunu PHP'nin CVS ağacındaki [1] değişikliklerden öğrenebilirsiniz. Ek olarak bugtraq listelerini ve SecurityFocus'da yer alan güvenlik açıklarını takip edebilirsiniz. PostgreSQL için bildirilmiş hata raporlarına PHP'nın hata rapoları [2] sayfasından ulaşabilirsiniz. Duyurulan yeni sürümlerde yapılan değişiklik kayıtları (*changelog*) ise yine PHP'nin kendi sitesinde [3] yer almaktadır.

[1] http://cvs.php.net/php-src/ext/pgsql/
[2] http://bugs.php.net/search.php?limit=All&direction=DESC&cmd=display&
bug_type[]=PostgreSQL+related
[3] http://www.php.net/releases.php

F.3. Python Arayüzü

psycopg de diğer bir çok PostgreSQL veritabanı arayüzü gibi, veritabanına yaptığı çağrılarda libpq kütüphanesini kullanmaktadır. Bu nedenle, libpq için bulunan herbir güvenlik açığı psycopg için de muhtemel bir sorun teşkil edecektir. Buna ek olarak, psycopg'den doğabilecek herhangi bir açık hakkında bilgi almak için psycopg'nin genel [1] ve duyuru [2] e-posta listelerini takip edebilirsiniz. Bunun dışında en güncel kaynak koduna SVN arşivlerinden [3] ulaşmanız mümkün.

[1] http://lists.initd.org/mailman/listinfo/psycopg
[2] http://lists.initd.org/mailman/listinfo/psycopg-announce
[3] http://initd.org/svn/psycopg/

VI. Sık Karşılaşılan Problemler

Bu bölümde, bahsi geçen programlama arayüzleri ile PostgreSQL veritabanını kullanırken sık karşılaşabileceğiniz problemler üzerinde durmaya çalışacağız.

1. Genel

Bu başlık altındaki problemler, genel veritabanı tasarımı ve programlaması ile ilgili olacağından tüm programlama arayüzlerini kapsamaktadır.

Nasıl türkçe karakter içeren sorgular çalıştırabilirim?

libpq kullanan uygulamalarda istemci ve sunucu arasındaki karakter kodlamasını doğru ayarladığınız sürece türkçe karakter içeren sorgulamalarda hiçbir sıkıntı çekmezsiniz. Türkçe karakterlerin bulunduğu PostgreSQL karakter setleri LATIN5 ve UNICODE şeklinde iki sınıftır. Eğer komutların bulunduğu dosyayı (kullandığınız metin editöründen ayarlayabileceğiniz) UNICODE karakter seti ile oluşturduysanız, bağlandıktan sonra karakter kodlamasını UNICODE'a ayarlayarak işinize devam edebilirsiniz. (Ya da dosya LATIN5 ile yazıldıysa, karakter kodlamasını LATIN5'e ayarlayarak.)

> *Python ile geliştirdiğiniz dosyalarda yorumlayıcının yolunu belirttikten (`#!/usr/bin/python` gibi) hemen sonraki satırda Python yorumlayıcısına yazılan betikte kullanılan karakter kodlamasını `-*- encoding: iso8869-9 -*-` satırını ekleyerek göstermelisiniz.*

Örneğin UNICODE karakter seti ile yazılmış sorgular içeren bir C dosyamız mevcut olsun. Bağlantı kurulduktan hemen sonra aşağıdaki komutu çalıştırdığımızda türkçe karakterler ile ilgili önünüzde hiçbir sorun kalmayacaktır.

```
SET client_encoding TO UNICODE;
```

> *Yukarıdaki işlemi `PGCLIENTENCODING` çevre değişkenini ayarlayarak ya da ilgili PostgreSQL kullanıcısının öntanımlı istemci karakter kodlamasını değiştirerek başarabilirsiniz. (Ayrıntılı bilgi ve örnekler için bağlantıda kullanılan çevre değişkenleri bölümüne bakınız.)*

Herhangi bir karakter uyuşmazlığı durumunda aşağıdakine benzer bir hata mesajı alırsınız:

```
invalid byte sequence for encoding "UNICODE": 0xfe6166
```

> *`LATIN5` karakter seti kullanılarak yazılmış bir dosya, istemci karakter kodlaması UNICODE olarak ayarlanmış bir veritabanında sorgulanmaya çalışıldığında sunucu tarafından dönen hata mesajının çıktısı yukarıda yer almaktadır.*

Bahsi geçen tüm bu adımların yanında, veritabanınızın türkçe karakter desteğini sağlayacak şekilde oluşturulmuş olmasına dikkat etmeniz gerektiğini hatırlayınız. Aksi halde - türkçe karakterlerin gönderiminde hiçbir sorun yaşanmamış olsa da - sıralama (ORDER BY) işlemleriniz gibi dile özgü bilgilerin kullanıldığı sorgulamalar hatalı sonuç döndürecektir.

Fazla sayıda satır döndüren işlemlerim neden bu kadar uzun sürüyor?

Veritabanı üzerinde gerçekleştirdiğiniz herhangi bir sorgu sonucunun istemci tarafındaki API aracılığı ile alımında, ulaşan tüm sorgu sonuçları ancak bellekteki ilgili yapılar üzerinde muhafaza edildikten sonra, programcıya bu yapılara erişim imkanı doğar. Bu nedenle, çok fazla sayıda geri dönen sonuç olduğu bir durumuda, bunların bellek üzerine kaydedilmeleri esnasında bir zaman kaybı doğacak olması muhtemeldir. Peki böyle bir durumuda nasıl bir yol izlenerek bu zaman kaybından kurtulunabilir?

İlk yapılması gereken işlem, dönen sonuç sayısını mümkün olan en az sayıya indirgemektir. Bunu yaparken sorgu tümcesi çeşitli ifadeler ile kısıtlanarak (DISTINCT, WHERE, HAVING, LIMIT, OFFSET gibi) sonuçlar gruplar halinde alınmaya çalışılınır. Bu sayede dönecek olan sonuç sayısı azaltılmış olup, daha spesifik bir alana yoğunlaşılmış olur.

Çok büyük miktarda sonuç döndüren sorguların, uzun vadede teorik sınırlar ile karşılacak olmaları da oldukça muhtemeldir. Bunu çok yüzeysel bir şekilde ifade etmeye çalışacak olursak, doğabilecek kayıplar şu şekilde listelenebilir:

- Veritabanı sunucusundan gelen veri istemci tarafındaki bellek üzerinde saklanmaya çalışıldığında, herhangi bir bellek yetmezliği durumunda swap alanı kullanılmaya başlanacaktır; ki bunun çok büyük bir performans kaybına neden olacak olması aşikardır.
- Oluşan satır sayısının artması ile birlikte, sistemin teorik sınırlarına yaklaşılacaktır. Bunun en basit örneği olarak, derleyici tarafından kullanılabilecek en büyük tamsayı değerinin tutulduğu INT_MAX[30] değişkeni gösterilebilir.

Bazı durumlarda, sorgu tümcesi ne kadar kısıtlansa da, dönecek olan sonuç sayısının hala zaman kaybına neden olacak nitelikte fazla olduğu anlar olur. Bu durumda yardımımıza CURSOR[31] ifadesi yetişir. CURSOR mantığı kullanarak DECLARE deyimi ile oluşturduğumuz bir sorgu sonucunu, FETCH ile parçalar halinde alabiliriz.

> *DECLARE kullanımında SCROLL seçeneğinden doğabilecek performans kaybına dikkat ediniz.*

Bunu basit bir örnek ile izah etmeye çalışalım. Elimizde 100,000 sonuç döndürecek bir sorgumuz olduğunu farzedelim. Sorguyu basit bir SELECT çağrısı ile gerçekleştirmeye çalıştığımızda:

```
-- Sorgu, sonucu tek parça halinde alınacak şekilde sunucuya gönderilir.
SELECT kullanici_adi, sifre, ...
  FROM kayitlar
  WHERE ...;
```

Dönecek olan 100,000 sonuç satırının herbiri API tarafından bellek üzerine kaydedildikten sonra, onu çağıran programın kullanımına sunulacaktır. Aynı sorguyu CURSOR kullanarak gerçekleştirmeye çalışırsak:

```
-- CURSOR kullanımı için transaction gerektiğini hatırlayınız.
BEGIN;

-- CURSOR oluşturuluyor.
```

30 Sisteminizdeki limits.h dosyasına bakarak, derleyiciniz tarafından desteklenen veri tiplerinin limitleri hakkında daha fazla bilgiye ulaşabilirsiniz.

31 CURSOR ifadelerinin nasıl kullanılacağı hakkında kitabın ilgili bölümüne bakabilirsiniz. (Beraberinde DECLARE, FETCH ve MOVE komutları da incelenecektir.)

```
DECLARE kullanici_kayitlari NO SCROLL CURSOR FOR
    SELECT kullanici_adi, sifre, ... FROM kayitlar WHERE ...;

-- Ardından veri CURSOR tarafından uygun miktarlarda parçalar
-- halinde alınır. Örneğin her seferinde 500 kayıt alalım:
FETCH FORWARD 500 FROM kullanici_kayitlari;
FETCH FORWARD 500 FROM kullanici_kayitlari;
...

COMMIT;
```

Yukarıdaki örnekte, tek seferde 100,000 kaydın tümünü almak yerine, `FETCH` kullanarak ulaşacak veriyi parçalara böldük. Bu sayede ilkine oranla daha fazla bir başarım sağlamış olmakla kalmayıp, sistem kaynaklarını da daha az tüketmiş olduk.

Veritabanında saklanan verinin doğal biçimi ikili olduğu için, verinin karşı tarafa karakter katarı olarak aktarılacağı bir durumda, ikili biçimdeki verinin karakter katarına dönüştürülmesi gerekir. Karşı tarafa karakter katarı halde ulaşan verinin, istemci üzerinde işlenmek için tekrar ikili hale dönüştürülmesinin gerektiği olabilir. Bu nedenle, veri aktarımı başarımının önemli olduğu yerlerde verinin ikili biçimde gönderilmesi, karakter biçimine oranla daha kârlı olacaktır. Bu sayede hem daha az yer kaplanacak olurken (çünkü karakter katarları genellikle ikili veriye nazaran daha çok yer işgal ederler), hem de olası çevrimlerden dolayı doğacak işlemci kullanımından kazanç sağlanacaktır.

> *Burada olası çevrim ile kast edilen, gelen verinin veritabanının saklama biçimine uygun şekilde uyarlanması esnasında yapılacak fonksiyon çağrılarıdır. Bu çevrimlerde, ikili verinin işlenmesi, karakter katarına oranla her ne kadar daha ucuza mal olacak olsa da, aşırı `NULL` (\0) içeren bir ikili veride, bunun tam tersinin yaşanacak olması da muhtemeldir.*

Veritabanında nasıl resim, müzik dosyası gibi ikili biçimde veri saklayabilirim?

Veritabanında herhangi bir dosyanın saklanmasının gerektiği bir durumuda izlenebilecek genel olarak iki yol vardır:

1. Sunucunun bulunduğu dosya sistemi üzerinde bulanan dosyanın sadece yolunun veritabanında bir tablo alanı içinde tutulması.
2. Dosyanın bir veritabanı alanı içinde (`bytea` ya da *LO* (*large object*) olarak) tutularak, kullanımına ihtiyaç duyulan tablolarda referans verilerek çağrılması.

> *Bazı durumlarda, bu iki yöntemin harmanlanarak kullanıldığı da olur. Dosyalar veritabanında saklanır ve ihtiyaç duyulduğunda dosya sistemine açılır. Dosya sistemine açılan bu dosyaların adresleri veritabanında ayrı bir tablo üzerinde tutularak, okuma işlemleri için dosya sistemindeki kayıtlar kullanılır. Herhangi bir değişikliğin istendiği durumda ise, değişiklik veritabanındaki kayıt üzerinde gerçekleştirildikten sonra, kaydın dosya sistemindeki eşi yeni değişikliğe göre güncellenir.*

Bu iki ayrım başlangıçta önemsiz gibi gözükse de, kendi aralarında çeşitli avantaj ve dezavatajlara sahiptir. Bu nedenle, dosyanın veritabanı mı, yoksa dosya sistemi mi üzerinde saklanacağı sorusu her şeyden önce önümüzdeki aşılmasa gereken ilk engeldir. Şimdi bu iki metodu da kendi aralarında artı ve eksileri ile değerlendirmeye çalışalım.

Devam etmeden önce, sunucu üzerinde veri saklanırken performans açısından düşünüldüğünde şu önemli noktaya değinmenin yerinde olacağını düşündük: Dosyalara yapılan erişimin optimizasyonunda izlenebilecek yollardan biri, verinin saklandığı disk bölümünün dosya sistemi muhasebelerinin etkisiz hale getirilmesidir. (Burada dosya sistemi muhasebesi ile kast edilen, ilgili disk bölümünün ana dosya sistemine bağlanırken, veri düğümlerine her erişimde dosyaya erişim zamanının güncellenmesini sağlayan `atime` ve benzeri özellikleridir.) Yüksek erişim altında kullanılan dosya sistemlerinde, bu yolun kullanımı çok ciddi performans artışlarına sebep olacaktır. Özellikle RAID diskler üzerindeki dosya sistemlerinde, erişim zamanlarının güncellenmesi

beraberinde ilgili verinin sağlama toplamının (`checksum` değerinin) tekrar hesaplanıp yazılmasını gerektirdiğinden, bu özelliğin kapatılması performansın önemli olduğu yerlerde çok daha büyük önem taşıyacaktır.

İlk önce bir dosyanın veritabanı üzerinde saklanacak olmasının bize, aynı dosyanın dosya sistemi üzerinde saklanmasına kıyasla sunduğu artıları masa üzerine yatıralım:

- Bir dosyanın, dosya sistemi yerine veritabanında saklanıyor olması bize *merkeziyetçi bir yapı* sağlar. Bu sayede, dosyalar tek bir noktada (veritabanı sunucusunun bulunduğu sistemde) toplanmış olup, bundan sonra istemciler için önde kalan tek engel bu sunucuya bağlanılması olacaktır. Dosya sistemi uyuşmazlığı (ya da NFS sıkıntısı) gibi problemler ile karşılaşılmayacaktır.
- *Veri bütünlüğü* açısından, veritabanları, dosya sistemlerine göre daha kararlı bir form sunacaktır. Sahip olduğu PITR ve benzeri veri yedekleme yöntemleri ve *transaction* destekleri ile sunucu üzerindeki verinin güvenliği ve sunucu/istemci arasındaki verinin bütünlüğü daha kararlı bir hale bürünecektir. (Benzeri bir yaklaşım, uğraşılırsa dosya sistemleri üzerinde de gerçekleştirilebilir. Fakat bu çok titiz bir çalışmayı gerektirecek olup, astarının yüzünden pahalı hale gelme olasılığı oldukça muhtemeldir.)

> *Dosya sistemi üzerinde bu tür atomik veri transferleri (yani verinin tam olarak istenilen yere - tek parça halinde - ulaştığından emin olunmadan tamam sinyali gönderilmemesi) hakkında daha ayrıntılı bilgi için versiyon kontrol sistemlerinden biri olan Subversion'ı (http://subversion.tigris.org/) inceleyebilirsiniz. Subversion, sunduğu atomik dosya onayı (`commit` işlemi) ile kendi alanında oldukça başarılı bir yazılımdır.*

- *Verinin şifrelenerek yollanması*nı düşünecek olursak, bu bir veritabanında ufak bir parametre kadar yakınken (Bkz. Bağlantıda sslmode parametresinin kullanımı.) dosya sistemleri için birbirinden farklı alternatifler, yine kendi içlerinde ayrıldıkları gruplara göre eksi ve artıları ile bu özelliği sağlarken, bir veritabanının sunduğu kolaylığı sunamayabilirler.

Tüm bu saydığım özellikler yanında, veritabanlarının da dosya sistemlerine kıyasla bu konuda hakkında bazı eksileri bulunmaktadır. Bu eksileri toparlamaya çalışacak olursak:

- RDBMS'lerde, veri iletişimi bir çok kontrol mekanizmasına tabi tutularak gerçekleştiğinden, normal bir dosya sisteminde saklanan dosyaya göre, veritabanında bir dosyanın sunumu *sistem kaynakları* için pek de alçak gönüllü davranmayacaktır. Bu nedenle, dosyanın veritabanından sağlanacak olmasının beraberinde bir yavaşlığa sebebiyet vermesi oldukça muhtemeldir. Özellikle veritabanında `bytea` tipi alan içinde saklanan verinin geri ayıklanması (`unescape`) ve sunumu işlemi oldukça fazla işlemci ve bellek gücüne ihtiyaç duymaktadır. (Bu dezavantaj verilerin parametre kullanılarak ya da *large objects* halinde iletildiği durumlarda geçerli değildir.)
- Olaya uzun vadede yaklaşıldığında, kullanılan RDBMS'in duyurulan yeni sürümünün *verinin saklanmasında kullandığı yeni yöntem*ler eskisi ile uyumlu olmayabilir; ki böyle bir durumda tüm veritabanının geçici bir yere boşaltılıp (*dump* edilip), yeni veritabanına yüklenmesi gerekecektir. Bir sistem yöneticisinin veritabanını dosya saklamak için kullandığı düşünelecek olursa, veri boyutu yüklü olacağından bu geri yükleme işinin epeyce meşakkatli olacağı ortadadır.
- Bazı RDBMS'lerde *dosya boyut sınırları*, dosya sistemi için geçerli olan sınırların çok altındadır. Örnek vermek gerekirse, PostgreSQL'de saklanabilecek en büyük dosya boyutu 2 Gb dolaylarındadır. Diğer taraftan, dosya sistemleri göz önüne alınacak olursa, kullanılan çekirdeğe göre bu sınır değişecek olup, genelde de bir dosya sistemi bir veritabanına göre çok daha büyük boyutlarda dosya tutubilir niteliktedir. Mesela, LFS (*Large File Support*) yamasına sahip bir Linux çekirdeği, üzerinde çalıştığı dosya sistemine göre (dosya başına) 16 GB ile 8 EB arasında değişebilen büyüklüklerde dosya boyutunu destekleyebilmektedir. (1024^0 EB = 1024^1 PB = 1024^2 TB = 1024^3 GB)
- Dosya sisteminde bir dosyanın çeşitli betikler ile kullanıcıya ulaştırılması, beraberinde bazı *potansiyel güvenlik açıkları*nı da getirmektedir. Kullanıcı formundan gelen, yeterli derecede kontrol edilmemiş veri, dosya sisteminde yanlış dosyanın açılmasına sebep

olabilir. (Günümüzde bir çok program sırf bu yüzden çok kritik güvenlik açıkları ile yüzleşmektedir.) Veritabanında saklanan dosyaların sunulmasında ise böyle bir olasılık (çok ender durumlar dışında) yok denecek kadar azdır.

Toparlamaya çalıştığımız bu eksi ve artılar dışında, izlenecek olan yöntemin seçiminde dikkat edilmesi gereken daha bir çok etmen soru işareti halinde önümüzde durmaktadır:

- Kullanılacak veritabanlarının boyutu (Kimse toplamı MB'leri aşmayacak boyutlarda bir veri için, sistemde koskoca bir RDBMS çalıştırmak istemez. Tabi ortada hali hazırda çalışan bir RDBMS varsa, sonuç getiri ve götürülerin farkına bakacaktır.),
- Kullanım amacı (Sık veri giriş ve güncellemeleri mi olacak, yoksa salt okuma işlemi için mi hizmet sunulacak?),
- Sistemin üzerinde koşacağı donanım (Çok basit olarak, 2 MB tampon belleğe sahip bir sabit disk ile, 8 MB tampon belleğesahip bir disk arasında fark doğacak olması aşikardır.)
- Sunucunun üzerinde çalışmakta olduğu işletim sistemi (Kullanılacak olan veritabanı, çekirdeğin barındırdığı özelliğe göre farklı işletim sistemlerinde farklı performans sergileyebilir.),
- Veritabanı sunucusunun ve üzerinde çalıştığı sistemin optimizasyonu için gerekli ayalarmaların yapılı olup olmadığı

gibi daha bir çok etmen bu listeye eklenebilir. Biz aşağıda her iki yaklaşım için de gerekli çözümü sunacak olsak da, amaç doğrultusunda gerekli unsurlar önem sırasına göre masa üstüne yatırıldıktan sonra, yapılacak seçimin bedeli olan götürülerin getirilere oranı göz önünde bulundurularak izlenecek yolun tayini tamamen programcı ve sistem yöneticisine kalmıştır.

> *Aşağıdaki örnekler konunun anlaşılması için oldukça yüzeysel ve basit tutulmaya çalışıldı. Kullanılan senaryoların benzerlerinin pratikte uygulaması buradakine oranla çok daha komplike ve derindir. Bu sebeple betiklerin güvenliği de gözardı edilmiş olup, sadece programın ana hatları yansıtılmaya çalışılmıştır.*

Dosya yolunun Veritabanında Saklanması

Bunun için ilk önce şuna benzer bir tablo hazırlanır:

```
=> \d dosya_listesi
                              Table "public.dosya_listesi"
   Column   |       Type        |                      Modifiers
------------+-------------------+-----------------------------------------------------
 dosya_nu   | integer           | not null default
                                  nextval('public.dosya_listesi_dosya_nu_seq'::text)
 dosya_yolu | character varying | not null
Indexes:
    "dosya_listesi_pkey" PRIMARY KEY, btree (dosya_nu)
    "dosya_listesi_dosya_yolu_key" UNIQUE, btree (dosya_yolu)
```

Herbir dosyaya tek bir `dosya_nu` numarası vererek hepsinin bir kimlik sahibi olmasını sağladık. Bu sayede istenilen dosya, `dosya_listesi` tablosunun ilgili `dosya_nu` alanına referans verilerek çağrılabilecek. Bu kullanımın şöyle bir artısı var: Dosya adı herhangi bir sebeple değiştiği zaman, dosyayı referans veren tablo üzerinde herhangi bir değişiklik yapılmasına gerek kalmayacak olup; sadece `dosya_listesi` tablosunda, değiştirilecek olan dosya yolunun güncellenmesi yeterli olacaktır. Çünkü isteyen zaten ilgili dosyaya ulaşırken, `dosya_listesi` tablosundan kullandığı `dosya_nu` değerindeki `dosya_yolu`'nu ayrıca kendisi alacak. `dosya_yolu` alanının `UNIQUE` olması sayesinde ise, aynı dosyanın tabloda iki yerde birden olması ihtimalinden kurtulunmuş olunur.

Şimdi `dosya_listesi` tablosundaki dosyaları kullanan ayrı bir tablo oluşturalım:

```
=> \d indir_bolumu
 Table "public.indir_bolumu"
```

```
 Column |  Type   | Modifiers
--------+---------+-----------
 dosya  | integer |
...
Foreign-key constraints:
    "indir_bolumu_dosya_fkey" FOREIGN KEY (dosya) REFERENCES dosya_listesi(dosya_nu) ON
UPDATE CASCADE ON DELETE CASCADE
```

Yukarıdaki gibi indir bölümü adı altında yeni bir tablo oluşturduk. (Bir internet sayfamız olduğunu ve burada kullanıcının indirebileceği programların sunulduğunu tasarlıyoruz. Bu yüzden pratikte `indir_bolumu` tablosu elbette sadece dosya sütununa sahip olmayacaktır. Aski halde `dosya_listesi` tablosundan farkı kalmaz.) indir_bolumu tablosunda indirilecek dosyanın saklanacağı dosya sütunu, `dosya_listesi` tablosundaki `dosya_nu` alanına referans verilerek oluşturuluyor. Ve tercihen eklediğimiz `ON UPDATE CASCADE ON DELETE CASCADE` ifadesi ile referans verilen alanın `dosya_listesi` tablosundan silindiği zaman, onu referans gösteren `indir_bolumu` satırlarının da silinmesini sağlıyoruz. Bu sayede `dosya_listesi` tablosundan silinen bir dosya, otomatik olarak onu referans gösteren diğer tablo satırlarının da silinmesini sağlayacaktır.

İlk önce `indir_bolumu` tablosundan `dosya_nu` değerini alalım ve bunu kullanıcıya sunacak olan `dosyaGetir.php` dosyasına bu numarayı yollayalım.

```
/*
 * indir_bolumu tablosu üzerinde hangi dosyanın indirileceğine
 * karar veren bir WHERE ifadesi kullanmalıyız. (Dönecek olan sonucun
 * tek bir değer döndürmesi gerektiğine dikkat ediniz.)
 */
$dosyaNuS = pg_query($baglanti, "SELECT dosya_nu FROM indir_bolumu WHERE ...");
$dosyaNu = pg_fetch_result($dosyaNuS, 0, 0);

/* Alınan dosya numarasındaki dosya getiriliyor. */
header("Location:dosyaGetir.php?dosyaNu=".$dosyaNu);
```

Şimdi ise kendisine belirtilen `dosya_nu` değerine sahip dosyayı kullanıcıya gönderecek olan (yukarıda `dosyaGetir.php` şeklinde belirtilen) PHP betiğimizi oluşturalım:

```
/* Gelen dosya_nu değerimizi kontrol edelim. */
$dosyaNu = pg_escape_string($_GET["dosyaNu"]);

/* Ulaşan dosya_nu değerine göre dosya yolunu alıyoruz. */
$sorguTumcesi = "SELECT dosya_yolu FROM dosya_listesi WHERE dosya_nu = ".$dosyaNu;
$dosyaYoluS = pg_query($baglanti, $sorguTumcesi);

if (pg_num_rows($dosyaYoluS) == 0)
    die($dosyaNu." numarasına sahip bir dosya bulunamadı!\n");
else
    $dosyaYolu = pg_fetch_result($dosyaYoluS, 0, 0);

/*
 * Biz burada dosya içerik tipi olarak "application/octet-stream"
 * kullanacağız. İsteğe bağlı olarak dosya_listesi tablosunda
 * bu değeri ayrı bir alanda tutarak daha işlevsel bir yapı oluşturulabilir.
 */
header("Content-Type: application/octet-stream");

/*
 * Kullanabileceğiniz diğer bazı HTTP başlıkları ise şöyle:
 *
 * header("Cache-Control: must-revalidate, post-check=0, pre-check=0");
 * header("Content-Length: ".$dosyaBoyutu);
 * header("Content-Disposition: attachment; filename=".basename($dosyaYolu));
 * header("Content-Type: application/force-download");
 *
 * Bu başlık değerleri tercihe bağlı olup bu örnekte kullanılmayacaktır.
 * Kullanım alanlarına örnek vermek gerekirse, Adobe AcrobatReader'ın bazı
 * sürümleri, "Cache-Control" başlığı olmadığı zaman PDF dosyalarını tarayıcı
```

```
 * penceresi içinde açmayabiliyor.
 *
 * HTTP/1.1 protokolü hakkında ayrıntılı bilgiye RFC 2616'dan ulaşabilirsiniz.
 * Kullanabilecek başlıklar için daha detaylı bilgi ise aynı RFC'nin
 * "Header Field Definitions" başlıklı 14. bölümü altında yer almakta.
 * Ek olarak IANA (Internet Assigned Numbers Authority) tarafından kayıtlı
 * MIME (Multipurpose Internet Mail Extensions) medya tipleri için
 * http://www.iana.org/assignments/media-types/ adresine bakabilirsiniz.
 */

/* Dosyayı aynen ekrana yazdırıyoruz. */
readfile($dosyaYolu);
```

Örnekte sadece konunun anlaşılması açısında üzerinde pek durmamış olsak da, bu tür programlar sıkı güvenlik kontrollerinden geçirildikten sonra kullanılmaya başlanmalı. En başta PHP'nin dosya erişimi kısıtlanıp (Kimse ev dizini altındaki `.pgpasswd` dosyasının açılmasını istemez.), açmasını istediğimiz dosyalara da erişim izni olmalı. Tüm bu olasılıklar programcı ve sistem yöneticisinin sorumluluğunda olup yoğun uğraş gerektirmektedir.

Dosyanın Veritabanında Saklanması

Bir önceki örnekte, veritabanında dosya yolunu saklayıp, dosyaya PHP kütüphanesince sağlanan `readfile()` fonksiyonu ile ulaşıyorduk. Burada ise, dosyamızı veritabanında saklayıp, bu sefer de PHP'ni PostgreSQL API'si tarafından sağlanan fonksiyonları kullanarak veriye ulaşmaya çalışacağız.

İlk önce dosyalarımızın saklanacağı tablo oluşturuluyor:

```
=> \d dosyalar
                                  Table "public.dosyalar"
    Column     |       Type        |                      Modifiers
---------------+-------------------+-----------------------------------------------------
 dosya_nu      | integer           | not null default
                                     nextval ('public.dosyalar_dosya_nu_seq'::text)
 dosya_adi     | character varying | not null
 dosya_icerigi | bytea             | not null
Indexes:
    "dosyalar_pkey" PRIMARY KEY, btree (dosya_nu)
```

Verinin saklanmasında large object kullanabileceğimiz halde, bu örnekte bytea tipinde veri alanı kullanmayı tercih ettik. (Hangisinin seçileceği hakkında libpq dökümantasyonunda incelenen büyük boyutlu nesneler başlığına göz atabilirsiniz.)

Bir önceki örnekte kullanılacak dosyalar, dosya sistemi üzerinde yer aldıklarından onları direk bulundukları yerden çağırabiliyorduk. Fakat bu sefer, kullanılacak olan dosyalar ancak veritabanına atıldıktan sonra sunulmaya hazır bir hale gelecek. Bu nedenle ilk önce dosyaları basit `INSERT` ifadeleri ile dosyalar tablosuna atmalıyız. Biz örnek olması açısından bir kaç tane müzik dosyasını tablo içine atıyoruz:

```
-- Tabloda yer alan dosyaları biz basit bir PHP betiği kullanarak ekledik.
-- Siz kendi ihtiyacınıza göre bunu kolay bir şekilde yapabilirsiniz.
-- Yapmanız gereken tek şey dosyayı açıp pg_escape_bytea() fonksiyonu ile
-- ayıkladıktan sonra INSERT sorgusu içine yerleştirmek. (Ayıklama işleminden
-- ve olası güvenlik açıklarından kaçınmak için dosyayı parametre kullanarak
-- göndermenizi öneririz.)
=> SELECT dosya_nu, dosya_adi FROM dosyalar;
 dosya_nu | dosya_adi
----------+-----------
        1 | ornek01.mp3
        2 | ornek02.mp3
        3 | ornek03.mp3
(3 rows)
```

Ardından, yine bir önceki örnekte izlediğimiz yönteme benzer olarak, dosyalar

tablosundaki dosya_nu kısmına referans gösteren ya da dosyalar tablosunu INHERITANCE ile miras olarak alan tablolar kullanarak dosya_icerigi alanına rahatlıkla ulaşabiliriz. Bu alana ulaştıktan sonra, yapmamız gereken ise, yine bir önceki örnekte izlediğimiz yöntem ile oldukça benzerdir:

```
/* Dosya içeriğini alıyoruz. */
$sTumcesi = “SELECT dosya_adi, dosya_icerigi FROM dosyalar WHERE dosya_nu = 1”;
$dosyaS = pg_query($baglanti, $sTumcesi);
$dosyaAdi = pg_fetch_result($dosyaS, 0);

/*
 * Dosyayı parametre kullanarak eklediyseniz (yani pg_escape_byte()
 * kullanılmadıysa) pg_unescape_bytea() fonksiyonuna gerek kalmayacaktır.
 */
$dosyaIcerigi = pg_unescape_bytea(pg_fetch_result($dosyaS, 1));

/* Dosya içeriği yazdırılıyor. */
header("Content-Type: audio/mpeg");
header("Content-Disposition: attachment; filename=".basename($dosyaAdi));
print $dosyaIcerigi;
```

Neden yazdığım SQL komutunun her anahtar kelimesi için *$1*, *$2*... şeklinde parametre kullanamıyorum?

İlk önce soruyu basit bir örnek ile daha açık bir hale indirgemeye çalışalım:

```
/* Sorunsuz çalışan bir PQexecParams() çağrısı. */
PQexecParams(bag, “SELECT sifre FROM kayitlar WHERE kullanici = $1 ...”, ...);

/*
 * Hata veren bir PQexecParams() çağrısı. Alınacak benzer hata mesajı:
 * ERROR:  syntax error at or near "$1" at character 19
 */
PQexecParams(bag, “SELECT sifre FROM $1 WHERE kullanici = $2...”, ...);
```

Bunlar parametrelerin sadece libpq kütüphanesinin sağladığı fonksiyonlar ile kullanımlarıydı. Peki aynı sorun PREPARE ifadesi ile de gözleniyor mu ona da bir bakalım:

```
-- Sorunsuz çalışan bir PREPARE ifadesi
=> PREPARE fonksiyon (int) AS
-> SELECT sifre FROM kayitlar WHERE kullanici = $1;
PREPARE

=> PREPARE hazir_sorgu (int) AS
-> SELECT sifre FROM $1 WHERE kullanici = $2;
ERROR:  syntax error at or near "$1" at character 46
LINE 2: SELECT sifre FROM $1 WHERE kullanici = 1;
```

Parametre kullanımı, beraberinde “Parametre nedir?” sorusunu da getiriyor. Sorguyu gerçekleştirecek olan SQL komutumuzda yer alan hangi anahtar kelimeler yerine biz $1, $2 şeklinde parametrik değerler kullanabiliriz? Yukarıdaki örneklerden de açıkça anlaşıldığını umduğumuz üzere, sorguda parametre olarak kullanılabilecek alanlar, dönen tablo sütunları ile eşleşecek şekilde olmak zorundadırlar. (Eğer bu bir INSERT sorgusu ise, girilecek değerler için parametre kullanılır; karşılaştırma ifadesi ise, dönecek alanların değerleri ile karşılaştırma yapmak için kullanılır.) Bu yüzden, farklı bir amaçla kullanılmaları durumunda hata oluşacak olması aşikardır.

lo kayıtlarım üzerinde işlem yaparken bağlantı kopması durumunda ne gibi bir sorunla karşılaşırım?

İlk önce şunu hatırlamakta fayda var: Yapılan her lo kaydı *transaction* blokları içinde yer alır. Probleme bu elde ile yaklaştığımızda olay tamamen “Bir *transaction* bloğu içindeyken herhangi bir bağlantı kopması durumunda ne gibi bir sorunla karşılaşırım?” sorusuna

kanalize olur. *Transaction*'lar bağlantı kopması durumunda sunucu tarafından (öntanımlı olarak) ROLLBACK işlemine tabi tutulacaklarından, yapılan değişiklikler geçerli olmayıp geri alınacaktır.

> *Transaction'lar hakkında ayrıntılı bilgi için kitabın ilgili bölümüne göz atabilirsiniz.*

Benzer olarak, COMMIT edil(e)meyen bir lo işlemi de geçerliliğini yitirecektir. Veritabanı bağlantısının tekrar sağlandığında yeni bir *transaction* bloğu içinde önceden yapılıp COMMIT edilmemiş tüm adımların tekrarlanması gerekir.

lo kayıtlarımı *pg_dumpall* ile yedekleyemiyorum. Bunu nasıl başarabilirim?

pg_dumpall komutu, veritabanınızın - çeşitli koşullar altında - birebir kopyasını oluşturmak üzere tüm PostgreSQL veritabanlarını pg_dump aracılığı ile bir düz metin dosyasına yazmanız için arayüz sağlar. Fakat lo tipinde veriler düz metin dosyalarına kayıt edilemeyeceğinden bunun için ayrıca pg_dump komutu kullanılmalıdır. Bunu yapmak için pg_dump ile oluşturulacak verinin düz metin biçiminde olmamasına ve lo kayıtlarını yedekleyecek şekilde olmasına dikkat edilmelidir. Daha açık olması için bir kaç örnek vermeyi uygun bulduk:

```
# Veritabanından pg_largeobject tablosunu yedekliyoruz.
$ pg_dump -Fc -Z0 -t pg_largeobject test > /var/backup/pgLargeObject-Fc.out

# Oluşan pgLargeObject-Fc.out çıktısını tekrar veritabanına eklemeye çalışıyoruz.
# (pg_largeobject tablosuna veri eklerken veritabanında üst seviyeli
# kullanıcı hakkımız olmasına dikkat ediyoruz.)
$ pg_restore -Fc -t pg_largeobject -a -d test /var/backup/pgLargeObject-Fc.out
```

> *Konu hakkında daha ayrıntılı bilgi için pg_dump ve pg_dumpall komutlarının kullanımının bahsedildiği ilgili bölüme bakabilirsiniz.*

En son kaydın *OID* değerini ya da satır niteleyicisini nasıl öğrenebilirim?

Eğer kaydın eklendiği tablo OID değerlerini tutuyorsa (yani tablo oluşturulurken WITHOUT OIDS ifadesi belirtilmediyse ya da WITH OIDS kullanıldıysa) PQoidValue() fonksiyonu ile en son eklenen kaydın OID değerine ulaşabilirsiniz. Şöyle ki:

```
/* İlgili INSERT sorgusunu sunucuya gönderiyoruz. */
sonuc = PQexec(baglanti, "INSERT INTO ... (...) VALUES (...)");
printf("%d\n", PQoidValue(sonuc));
```

Kayıt satırları OID değerleri içermeyen bir tabloda üzerinde çalışıyor iseniz, satır için niteleyiciler kullanarak benzer işleve sahip bir yapı oluşturabilirsiniz. (Burada niteleyiciler ile kastımız SERIAL tipinde tablo alanları.) Basit bir örnek vererek izah etmeye çalışalım:

```
-- OID değeri bulundurmayan tablomuzu oluşturuyoruz.
=> CREATE TABLE oid_degeri_olamayan_tbl (
->   numara serial,
->   veri   varchar
-> ) WITHOUT OIDS;

-- Yukarıdaki tabloyu (serial tipinde alanı kendimiz oluşturarak)
-- ayrıca şu şekilde de yaratabilirdik:
--
-- CREATE SEQUENCE seq_numara START 1;
-- CREATE TABLE oid_degeri_olamayan_tbl (
--   numara integer NOT NULL DEFAULT nextval('public.seq_numara'::text),
--   veri   varchar
-- ) WITHOUT OIDS;
--
```

```
-- serial tipi bizim için yukarıdaki işlemde olduğu gibi bir SEQUENCE
-- yaratıp, onun her seferinde bir sonraki değerini numara sütununun
-- öntanımlı değeri olarak atayacaktır.

=> SELECT currval('public.seq_numara'::text);
=> INSERT INTO oid_degeri_olamayan_tbl (veri) VALUES ('bir');
INSERT 0 1
=> SELECT currval('public.seq_numara'::text);
 currval
---------
       1
(1 row)

=> INSERT INTO oid_degeri_olamayan_tbl (veri) VALUES ('iki');
INSERT 0 1
test=> SELECT currval('public.seq_numara'::text);
 currval
---------
       2
(1 row)

=> SELECT nextval('public.seq_numara'::text);
 nextval
---------
       3
(1 row)

=> SELECT currval('public.seq_numara'::text);
 currval
---------
       3
(1 row)

=> INSERT INTO oid_degeri_olamayan_tbl (veri) VALUES ('uc');
INSERT 0 1
=> SELECT currval('public.seq_numara'::text);
 currval
---------
       4
(1 row)

=> SELECT * FROM oid_degeri_olamayan_tbl;
 numara | veri
--------+------
      1 | bir
      2 | iki
      4 | uc
(3 rows)
```

Gerçekleştirdiğim sorgulamaların sonuçlarını nasıl ikili biçimde alabilirim?

Öncelikli yol olarak, libpq kütüphanesinin 3.0 protokolü ile birlikte sağlanan `PQexecParams()`, `PQexecPrepared()`, `PQsendQueryParams()` ve `PQsendQueryPrepared()` fonksiyonlarını[32] kullanabilirsiniz. Bunlar sayesinde sorgu sonucu dönecek verinin hangi biçimde (karakter katarı ya da ikili) aktarılacağını ayarlanabilirsiniz.

Kullanılan API'nin bu fonksiyonları sağlamadığı bir durumda, `DECLARE` ve `COPY` komutları sonucu ikili biçimde döndürecek şekilde ayarlanabilir. Şöyle ki:

```
-- DECLARE ifadesinin BINARY seçeneği ile birlikte
-- kullanıldığına dikkat ediniz.
DECLARE kullanici_kayitlari BINARY NO SCROLL CURSOR FOR
```

32 PHP ve Python'un PostgreSQL API'lerinin libpq kütüphanesini kullandığından daha önce bahsetmiştik. Bu nedenle, bu programlama dillerinde kullanılan eşdeğer fonksiyonların isimleri de libpq kütüphanesindekilere benzer olacaktır.

```
    SELECT kullanici_adi, sifre FROM kayitlar ...;

-- COPY ifadesinin son satırında BINARY
-- seçeneğinin kullanıldığına dikkat ediniz.
COPY kayitlar (kullanici_adi, sifre)
    TO STDOUT
    WITH BINARY;
```

Neden parametre kullanarak gönderdiğim sorguların işletilmesi esnasında biçim uyuşmazlığı ortaya çıkıyor?

Veritabanı sorgulamalarınızda parametre kullanarak gönderdiğiniz verinin tipini `parametre_bicimleri` dizisi içinde belirtmeyip, bunun sunucu tarafından tahmin edilmesini istediğiniz durumda ufak bir istisna bulunmaktadır. PostgreSQL parametre kullanılarak gönderilen verinin biçimini ilgili tablo sütunun biçimi ile bir tutacaktır. Örnek olarak

```
SELECT ... WHERE x = $1 ...
```

sorgusunu ele alalım. Burada `$1` verisinin biçimi açık olarak belirtilmediği taktirde, sunucu bunu `x` değerinin biçimi ile bir tutacaktır. Bu gibi durumlardan kaçınmak için gönderilecek verinin tipinin açık olarak belirtilmesinde yarar vardır. Yani – `$1` değişkenin tipinin `bigint` olduğunu varsayacak olursak – yukarıdaki sorgunun

```
SELECT ... WHERE x = $1::bigint ...
```

şeklinde gönderilmesi daha sağlıklı sonuç verecektir.

> *Özellikle `sonuc_bicimi`'nin ikili olduğu durumda, ikili verinin tipinin tahmin edilmesi – karakter kataı biçimindeki sorgulara oranla – daha zor bir hal alacaktır.*

Parametre kullanımı veri transferi esnasında bana herhangi bir başarım farkı yaratır mı?

Bu soruya verilecek cevap verinizin büyüklüğüne göre değişecektir. Bunu basit bir örnek üzerinde ifade etmeye çalışalım. Aşağıdaki sorguyu ele alalım:

```
SELECT kullanici_adi FROM kayitlar WHERE kullanici_id = $1;
```

Bu sorgunun parametre kullanılmadan gönderimi, parametre kullanılarak gönderimine oranla muhtemelen daha hızlı olacaktır. Bunun en büyük sebebi, PostgreSQL *planner*'ının `$1` değişkeninin değerini bilmediği için gerekli optimizasyonu yapamamasından kaynaklanmaktadır. Ve bu yanlış *index* kullanımına kadar gidebilir. Bu sebeple bu tarz nispeten ufak verinin gerektiği sorgulamalarınızda parametre kullanılmamasını tavsiye etmekteyiz. Peki hangi durumlarda parametre kullanımı bana bir hız artışı sağlar? Şu sorguyu göz önüne alalım:

```
INSERT INTO muzik_arsivi (parca_id, muzik_dosyasi) VALUES ($1, $2);
```

Burada `muzik_dosyasi` alanında göndereceğiniz veri çok büyük ihtimalle bir kaç 100K (hatta bir kaç MB) boyutunu aşacağından, veriyi parametre kullanmadan iletmeye çalıştığınızda, veri üzerinde ayıklama işlemine gerek duyulacaktır. Ve bu işlem çok yoğun bir işlemci kullanımına gereksinim duymaktadır. Fakat aynı veriyi parametre kullanarak gönderdiğinizde ayıklama işleminin giderinde tamamen kurtulmuş olacaksınız. Bu çok ufak bir fark gibi görünse de, çoğu zaman bir kaç saniye gibi bir veritabanı için oldukça önemli bir zaman farkına yol açmaktadır.

Parametre kullanımının bereberinde sağladığı bir diğer başarım artısı ise ikili tipteki verinin alımına olanak sağlamasıdır. Bir önceki örnekte geçen tabloyu ele alacak olursak:

```
SELECT muzik_dosyasi FROM muzik_arsivi WHERE parca_id = ...;
```

Buradaki gibi alınacak verinin de ikili olduğu durumlarda, bahsi geçen özellik yine fazladan bir başarım artışı sağlayacaktır.

Neden yazdığım programlar değişik mimarilerde çalıştırıldığında, ikili veri döndüren sorgulamalarımda tamsayı değerler hatalı ulaşıyor?

Sorgulama sonucunun ikili olduğu durumlarda, PostgreSQL tamsayıları (sunucunun mimarisinden bağımsız olarak) *Big Endian* ağ bayt sırasında (*network byte order*) iletir. Bu nedenle taşınabilir nitelikte olmayan bir kod, sorgu sonucu ikili verinin döndüğü durumlarda farklı mimarilerde farklı sonuçlar doğuracaktır.

> *Örneğin Intel 80x86 ve türevi işlemciler "Little Endian" bayt sırası kullanırken; Sun'ın SPARC, Motorola'nın 68K işlemcileri "Big Endian" kullanırlar. Ek olarak bazı işlemciler (PowerPC, MIPS, DEC Alpha) "Big Endian" ya da "Little Endian" olacak şekilde ayarlanabilirler.*

Basit bir örnek ile durumu özetlemeye çalışalım:

```
/* Verinin saklanacağı değişken. */
char *numara;

/*
 * PQexecParams() fonksiyonunun sonuncu parametresine 1 değeri
 * vererek, dönecek olan verinin ikili biçimde olmasını
 * istediğimizi belirtiyoruz.
 */
sonuc = PQexecParams(baglanti, "SELECT 123::int4",
                     0, NULL, NULL, NULL, NULL, 1);

/* Numarayı ekrana yazdırıyoruz. */
numara = PQgetvalue(sonuc, 0, 0);

/* int4 tipi 32 bitlik olduğu için int32_t kullanıyoruz. */
printf("%d\n", *((int32_t *) numara));
```

Eğer betiğin çalıştırıldığı sistem *Big Endian* mimarisinde ise, çıktı beklediğimiz gibi 123 olacaktır. Fakat *Little Endian* kullanıcıları 123 yerine 2063597568 çıktısı ile karşılaşacaklardır. Bunun sebebini şu şekilde açıklayabiliriz: onluk sistemdeki 32 bitlik 123 sayısının bellek üzerindeki onaltılık gösterimi 7D 00 00 00 şeklinde olacağından, *Big Endian* mimariler bunu 0x0000007D = 123 şeklinde hesaplayacak olurken, *Little Endian* mimariler 0x7D000000 = 2063597568 şeklinde hesaplayacaklardır.

Bu gibi bir sorundan kaçınmak istiyorsak, yazacağımız kodun mimariler arası taşınabilir olmasına dikkat etmeliyiz. Bunun için ise gelen veriyi çalışmakta olduğumuz mimarinin bayt sırasına çevirmemiz gerekmektedir. Bu amaçla standart kütüphane tarafından sağlanan `htonl`/`htons`/`ntohl`/`ntohs` ailesini kullanabiliriz fonksiyonunu kullanabiliriz. Örnek olması amacıyla, yukarıdaki kodu mimariden bağımsız bir hale getirecek olursak:

```
char *numara;
int inumara;

/*
 * PQexecParams(..., 1) ile (sonuç ikili biçimde
 * dönecek şekilde) sorgumuzu iletiyoruz.
 */

numara = PQgetvalue(...);

/* Gelen verinin bayt sırasını işlemcininkine çeviriyoruz. */
inumara = ntohl(*((int32_t *) numara)); /* int4 tipinin 32 bit olduğunu hatırlayınız. */

/* Numarayı doğru bir şekilde ekrana yazdırıyoruz. */
printf("%d\n", inumara);
```

Uzunluğu 1 bayt olan tiplerde bu tür bir sorun yaşanmamaktadır. Çünkü dönen veri zaten

1 bayt uzunluğunda olduğundan, herhangi bir *Little Endian* ya da *Big Endian* dönüşümüne tabi tutulmaya gerek kalmadan işlenir. Şöyle ki:

```
/*
 * => SELECT typname FROM pg_type WHERE typlen = 1;
 *  typname
 * ---------
 *  bool
 *  char
 * (2 rows)
 *
 * Aşağıdaki “SELECT 'A'::char” yerine herhangi bir karakter
 * girebileceğiniz gibi “SELECT 1::bool” seçeneğini de deneyebilirsiniz.
 */
sonuc = PQexecParams(baglanti, “SELECT 'A'::char”,
                     0, NULL, NULL, NULL, NULL, 1);

gelenVeri = PQgetvalue(sonuc, 0, 0);

/* char tipi 8 bitlik olduğu için int8_t kullanıyoruz. */
printf(“%c\n”, *((int8_t *) gelenVeri));
```

Yukarıdaki programda, kullanılan tipler gereği, istemci mimarisinin *Little Endian* ya da *Big Endian* olmasından bağımsız olarak sonuç ekrana doğru yazılacaktır.

COPY ile veri gönderirken neden bellek yetmezliği oluşabiliyor?

`COPY` komutu ile sunucuya herhangi bir veri gönderiminde tablo üzerinde bulunan `AFTER INSERT` tetikleyicileri (*trigger*'ları) bellekte tutularak veri alımı sunucu tarafından başarıyla gerçekleştirildikten sonra işletilirler. Bu sebeple, tablo üzerinde tanımlı `AFTER INSERT` tetikleyicilerinin ve verinizin yoğunluğuna bağlı olarak bellek yetmezliği yaşanabilir.

Bu durumdan kaçınmak için `COPY` verisinin parçalar halinde gönderimine gidilebilir.

Kimlik denetimini SSL ile gerçekleştirebilir miyim?

PostgreSQL ile *SSL* (*Secure Sockets Layer*) kullanılarak kimlik denetimi gerçekleştirilememektedir. Bu PostgreSQL bir eksiğinden çok, kimlik denetimine yaklaştığı tasarım ile ilgilidir. Örnek olarak, sunucu sertifikası ile imzalanmış bir istemci sertifikasının, başka istemcilere kopyalanması suretiyle o istemcilerin de sunucuya *SSL* ile bağlanması sağlanılabilir. PostgreSQL bu noktada herhangi bir kimlik denetimi gerçekleştirmeyecektir. *SSL*, sadece güvenli veri transferi için bir ara katman olarak kullanılmaktadır. Kimlik denetimi için `pg_hba.conf` dosyasında (ve kullandığınız yönteme göre diğer ilgili dosyalarda) gerekli değişiklikleri gerçekleştirmeniz gerekmektedir.

Bahsi geçen uygulama arayüzleri de PostgreSQL kadar özgür mü?

Şüphesiz ki PostgreSQL'in bu kadar yaygınlaşmasının en büyük sebeplerinden biri sahip olduğu BSD lisansıdır. Bu sayede birçok firma ürettiği cihaz ve programlarda PostgreSQL veritabını kendi kullandıkları lisansı etkilemeyecek şekilde sundukları ürünlerine entegre edebilmektedirler. Bunun geri dönüşü olmayan bir tüketim şekli olduğu düşünülebilir. Fakat, PostgreSQL'in son sürümlerinde sunduğu bir çok endüstriyel özellik, bunun bu şekilde geri dönüşümü olmayan bir mekanizma şeklinde olmadığının en büyük kanıtıdır. Konuya bu açıdan yaklaşıldığında, PostgreSQL ile birlikte kullanılacak yazılımların lisanslarının da bize ne gibi sınırlamalar getireceği önem kazanmaktadır.

> *PostgreSQL tarafından kullanılan BSD lisansı hakkında ayrıntılı bilgi için http://www.postgresql.org/about/licence adresine göz atabilirsiniz.*

PHP programlama dili, 3.0 sürümüne kadar çift lisanslı olarak (GPL ve kendi lisansı ile

birlikte) dağıtılmasına rağmen, 3.0 sürümünden sonra yeni (ve tek) bir lisansa geçmiştir. PHP lisansının 3.0 sürümüne çok kabaca değinecek olursak, lisans topluma açık ya da ticari bir kullanım için, Zend motoru üstünde herhangi bir değişiklik yapılmadığı ve PHP kullanan yazılım ile birlikte uygulamanın PHP kullanılarak geliştirildiğine dair bir ibarenin yer alması dışında istenildiği gibi dağıtılabilir.

> *PHP lisansının 3.0 sürümüne http://www.php.net/license/3_0.txt adresinden erişebileceğiniz gibi, lisans hakkında sık sorulan sorular için http://www.php.net/license/ adresine bakabilirsiniz.*

psycopg, Genel Kamu Lisansı'nın (GPL) 2. sürümü ile dağıtıldığından dolayı, PHP'nin sağladığı rahatlığı sunmamaktadır. Özellikle GPL lisansına sahip bir yazılımın kullanılması durumunda, ilgili yazılımı baz alan ürünlerin (çeşitli istisnalar dışında) dağıtımı ve satışında, kaynak kodunun da ürün ile birlikte dağıtılması zorunluluğu getirilmektedir. Bu nedenle, GPL lisansı BSD ve PHP lisansına oranla ticari yazılımlar için oldukça sert ve keskin haklara sahiptir.

> *Genel Kamu Lisansı hakkında ayrıntılı bilgi için http://www.gnu.org/licenses/licenses.html#TOCGPL adresine başvurabilirsiniz. GPL'in /resmi olmayan) Türkçe bir kopyası için ise http://www.belgeler.org/howto/gpl.html adresine bakabilirsiniz.*

Hangi veritabanı arayüzleri PostgreSQL'i destekliyor?

Herhangi bir programlama dilinde kullanacağınız bir veritabanı arayüzü, yazılan bir programın sabit kütüphane çağrıları ile birçok veritabanı üzerinde çalışmasına olanak sağlar. (Veritabanı arayüzünün sağladığı kütüphaneyi kullanarak geliştirdiğiniz bir yazılımın üzerinde hiçbir değişikliğe gerek kalmadan hem PostgreSQL hem de Oracle veritabanı sunucusu ile çalışabileceğini düşünün.) Bu sayede yazdığınız yazılımın taşınabilirliği arttığı gibi veritabanına olan bağımlılığı da ortadan kalkacaktır. Fakat bunun yanında, bu yöntem beraberinde bazı eksikleri de getirmektedir. Arayüz tarafından sağlanan bir fonksiyonun desteklenen veritabanlarının hepsi tarafından sağlanması gerektiği için bu, yazılım üzerine çok büyük bir kısıtlama getirecektir. Basit bir örnek olarak şunu verebiliriz: PostgreSQL C *API*'si tarafından sağlanan asenkron bağlantı kurulum fonksiyonları X veritabanı sunucusu tarafından desteklenmiyorsa ve kullandığınız veritabanı arayüzüz yazılımı hem X'i hem de PostgreSQL'i desteklemek istediği sürece PostgreSQL'in asenkron bağlantı kurulum fonksiyonlarından yararlanamaycaktır. Aksi halde programınızın taşınabilirliği zarar görecektir. Bu ufak uyarıdan sonra aşağıda çeşitli programlama dillerinde kullanabileceğiniz bir kaç veritabanı arayüzünün listesini bulabilirsiniz:

Prog. Dili	Yazılım	Desteklediği Veritabanı Sunucuları
C	libdbi	Firebird, FreeTDS (MS SQL, Sybase), MySQL, PostgreSQL, SQLite.
	libgda	MySQL, ODBC, PostgreSQL, Sybase.
PHP	ADOdb	ADO, Access, DB2, Firebird, Foxpro, FrontBase, Informix, Interbase, LDAP, MS SQL, MySQL, Netezza, ODBC, ODBTP, Oracle, PostgreSQL, SAP DB, SQLite, Sybase.
	DB (PEAR)	FireBird, Informix, InterBase, mSQL, MS SQL, MySQL, MySQLi, ODBC, Oracle, PostgreSQL, SQLite, Sybase.

> *Yukarıdaki listeye Python programlama dilini almamamızın sebebi, psycopg veritabanı bağdaştırıcısını tanıtırken üzerinde durduğumuz DB-API PEP'inden kaynaklanmaktadır. (Bunun dışından ADOdb'nin bir de Python programlama dili türevinin olduğunu da belirtmekte fayda var.)*

2. C Arayüzü

Bu bölümde, libpq kütüphanesini kullanarak geliştirdiğiniz C/C++ programlarında sık

karşılabileceğiniz sorunlar üzerinde durmaya çalışacağız.

Neden PostgreSQL 8.0.2 sürümüne yükseltildiğinde libpq kütüphanesine bağımlı yazılımlarda sorun çıkıyor?

libpq'nun ana sürüm numarasının PostgreSQL 8.0 ağacı ile birlikte yükseltilmesi planlanmıştı, fakat yaşanan bir aksaklık sonucu bu değişiklik CVS ağacına onaylanamadan 8.0 sürümü duyuruldu. Bu sebepten dolayı (bu değişiklik er yada geç yapılmak zorunda olduğundan) 8.0.2 sürümü ile birlikte bu güncelleme gerçekleşti. Fakat son değişiklik ile birlikte libpq kütüphanesinin bulunduğu `libpq.so.3` dosyasının yerini `libpq.so.4` aldı. Bu sefer ise ortaya libpq bağımlı yazılımlar tarafından sorun çıkmaya başladı: `libpq.so.3` dosyasını kullanarak derlenen programlar yeni `libpq.so.3` yerine gelen `libpq.so.4` dosyasını bulamıyorlardı.

`libpq.so.4` dosyasını `libpq.so.3` dosyasına sembolik olarak bağlamak suretiyle bu sorun kısa vadede çözümlenebilir. Fakat bu ileri aşamalarda sorunlara yol açacaktır. (Sunucu ile istemci arasında sürüm uyuşmazlığından dolayı problemlerin doğacak olması oldukça muhtemeldir.) Sonuç olarak, uzun vadede tek çözüm libpq bağımlılığı gösteren tüm yazılımların baştan derlenmesi olacaktır.

Konu ile ilgili bir kaç bağlantı vermeyi uygun bulduk:

- libpq kütüphanesinin major numarası 1.126.4.2 numaralı revizyon ile CVS ağacına eklendi. Makefile dosyası üzerindeki değişikliklere http://developer.postgresql.org/cvsweb.cgi/pgsql/src/interfaces/libpq/Makefile.diff?r1=1.126.4.1;r2=1.126.4.2;f=h adresinden ulaşabilirsiniz.
- PostgreSQL için RPM paket geliştiricileri listesinde libpq4'ten dolayı doğan bağımlılık probleminin tartışıldığı ilgili posta yivi: http://pgfoundry.org/pipermail/pgsqlrpms-hackers/2005-April/000197.html

3. PHP Arayüzü

Bu bölümde, PHP tarafından sağlanan PostgreSQL programlama arayüzünün kullanımında sık karşılaşılabilecek problemler üzerinde duracağız.

PostgreSQL sunucusunun çalıştığından emin olduğum halde, bağlantı kurulurken sorun yaşıyorum.

Veritabanı sunucusunun bulunduğu sisteme bağlanırken karşılaşacağınız iki çeşit problem vardır. Bunlardan birincisi PHP'nin kendi kısıtlamalarından doğacak olan limitlere takılmanız olup, PostgreSQL veritabanı sunucusu ile bir ilgisi yoktur. İkincisi ise veritabanı ile kimlik kontrolünde çıkan uyuşmazlıktan dolayı doğacak olan hatalardır.

İlk önce birinci tip probleme örnek verelim. *INI* ayarlarında yer alan `pgsql.max_links` değerinin aşıldığı bir durumda doğacak olası hata mesajı şu şekilde olacaktır:

```
Cannot create new link. Too many open links (n).
```

(Burada n varolan toplam bağlantı sayısını göstermektedir.) Böyle bir hata, PHP'nin INI ayarlarınca yapılandırılmış maksimum bağlantı sayısının aşıldığını ve daha fazla bağlantı kuralamayacağı anlamına gelir. Problemin çözümü için `pgsql.max_links` ve eğer kalıcı bağlantı kullanıyorsak `pgsql.max_persistent` *INI* ayarları gözden geçirilip baştan yapılandırılmalıdır.

En çok karşılaşılan problemlerden bir diğeri ise, sunucunun sadece yerel (local) bağlantılara açık olduğu bir durumda, `pg_connect()` fonksiyonunda parametre olarak `host=127.0.0.1` ya da `host=localhost` kullanıldığı halde bağlantı kurulumunda şöyle bir hata mesajı ile karşılaşılması:

```
Warning: pg_connect(): Unable to connect to PostgreSQL server: could not connect to server: Connection refused Is the server running on host localhost and accepting TCP/IP connections on port 5432?
```

Hatırlarsak, sunucumuzun sadece yerel bağlantılara izin verdiğini söylemiştik. Fakat hata mesajını baktığımızda sunucunun TCP/IP bağlantılarını kabul ettiğinden emin olup olmadığımızı soruyor. Fakat biz yerel bağlantıda unix soketleri kullanılsın istememiş miydik? TCP/IP'nin orada işi olmamalıydı! Tüm bu karmaşanın sebebi ise, bağlantı kurulumunda bağlantı fonksiyonuna verdiğimiz `host` (ya da `hostaddr`) parametresi bağlantının TCP/IP olacağı anlamına gelir. (Ayrıntılı bilgi için libpq kütüphanesinde incelenen `PQconnectdb()` fonksiyonuna göz atınız.) Yerel bağlantılar için bağlantı seçeneklerinde sunucu kısmı yerine hiçbir şey belirtilmemelidir.

Şu da unutulmamalıdır ki, aynı hata mesajını veritabanı sunucusunun bulunduğu sistemdeki `postgresql.conf` dosyasında yer alan `listen_addresses` parametresinin içinde, istemcinin adresi bulunmadığı zaman da alabilirsiniz. Bu durumda, veritabanına TCP/IP ile geçerli bir bağlantı sağlamış olmanıza rağmen, `listen_addresses` değerleri arasında olmadığınızdan, veritabanı sunucusu sizi reddecektir.

Tüm bunlara dikkat edildiği halde hala şuna benzer bir hata mesajı alabilirsiniz:

```
pg_connect(): Unable to connect to PostgreSQL server: FATAL:  no pg_hba.conf entry for host "[local]", user "postgres", database "template1", SSL off in /var/www/baglanti.php on line 31
```

Bu durumda kullanıcı kimlik denetimi başarısızlıkla sonuçlanmış demektir. Yani veritabanı ile temas sağlanmış, fakat bağlantının tamamlanması için gerekli kimlik denetimi başarısız olmuştur. Çözüm için ise `pg_hba.conf` dosyanıza tekrar göz atıp, ilgili istemci için doğru yapılandırmayı gerçekleştirdiğinizden emin olmalısınız.

PHP, veritabanı bağlantılarını nasıl sağlar?

Bu soru altında yer alacak açıklama Resmi PHP dökümantasyonunun 40. bölümünün bir çevirisi olup, belgenin asıl ve son haline http://php.net/manual/en/features.connection-handling.php adresinden ulaşabilirsiniz.

PHP iç mekanizması tarafından bir bağlantının sürdürülmesi esnasında meydana gelebilecek olası 3 durum vardır:

Değişken	Değer
CONNECTION_NORMAL	0
CONNECTION_ABORTED	1
CONNECTION_TIMEOUT	2

Bir betiğinin normal olarak çalışması esnasında `NORMAL` durumu etkindir. Diğer uçtaki istemcinin bağlantıyı kesmesi durumunda ise `ABORTED` durumu devreye girecektir. İstemcinin bu tür bir bağlantı kesimini gerçekleştirmesi ise genellikle tarayıcının Dur (*Stop*) düğmesine tıklaması ile gerçekleşir. Diğer bir ihtimal de, PHP tarafından tanımlanmış çalışma süresinin (Bkz. `set_time_limit()`) aşılıp, zaman aşımına uğrandığından dolayı `TIMEOUT` durumunun etkin hale geçmesidir.

İstemci bağlantısı kesildiği anda, betiğinizin çalışmaya devam edip etmeyeceğine karar verebilirsiniz. Bazı durumlarda, istemci üretilecek hiçbir çıktıyı almayacak olsa da, betiğinizin sona kadar çalışmasının gerekli olduğu anlar olabilir. Fakat, öntanımlı olarak istemci bağlantıyı kestiği anda betiğiniz sonlandırılacaktır. Bu tutum, PHP *INI* ayarlarındaki `ignore_user_abort` değişkeni ile yapılandırılabileceği gibi Apache'nin ilgili ayar dosyasına `php_value ignore_user_abort` satırının eklenmesi ya da `ignore_user_abort()` fonksiyon çağrısı ile de değiştirilebilir. İstemci kopmasının göz ardı edileceği PHP'ye bildirmediği sürece, istemci bağlantıyı kestiği o an betiğiniz sonlandırılacaktır. Bu davranışa tek istisna, `register_shutdown_function()` fonksiyonu ile tanımlanmış bir kapanma fonksiyonudur. Kapanma fonksiyonu ile birlikte, istemci dur

emrini verdikten sonra betiğiniz bir sonraki çalıştırılmasında, PHP betiğin bir önceki çalışması esnasında istemci tarafından sonlandırıldığını ve kapanma fonksiyonunun çalıştığını algılayacaktır. Kapanma fonksiyonu, betiğiniz normal bir şekilde sonlandığı zaman da çalıştırılacaktır; bu yüzden beklenmeyen bir kapanma durumunda ayrıca bir fonksiyon çalıştırmak için `connection_aborted()` fonksiyonuna göz atabilirsiniz. Eğer bağlantı beklenmeyen bir şekilde kapatıldıysa bu fonksiyon `TRUE` değeri dönecektir.

Ayrıca, betiğiniz gömülü betik zamanlayıcısı ile de kapatılabilir. Öntanımlı zaman aşımı süresi 30 saniyedir. Bu değer, *INI* yapılandırma dosyasındaki `max_execution_time` değişkeni ile ayarlanabileceği gibi Apache'nin ilgili ayar dosyasına `php_value max_execution_time` satırı eklenerek ya da `set_time_limit()` fonksiyonu çağrılarak da değiştirilebilir. Bu şekilde ayarlanan bir çalışma süresi aşıldığı zaman, betik sonlandırılacaktır ve yukarıdaki istemcinin bağlantıyı kesmesi durumunda olduğu gibi, mevcut kapanma fonksiyonu çalıştırılacaktır. Kapanma fonksiyonu içinde `connection_status()` çağrısında bulunarak, betiğin sonlandırılmasına zaman aşımının mı yoksa başka bir şeyin mi sebep olduğunu öğrenebilirsiniz. Betik zaman aşımından dolayı sonlandırılmışsa fonksiyon `CONNECTION_TIMEOUT (2)` değeri dönecektir.

Şu da unutulmamalıdır ki, `ABORTED` ve `TIMEOUT` durumları aynı anda etkin olabilir. Bu ise, istemcinin bağlantıyı sonlandırmasının göz ardı edildiği bir durumda, istemci bağlantıyı sonlandırdıktan sonra betiğin çalışamaya devam etmesi ile mümkündür. Eğer bu esnada betik zaman aşımına uğrarsa sonlandırılacaktır ve mevcut kapanma fonksiyonu çağrılacaktır. Bu noktada, `connection_status()` çağrısı 3 değeri dönecektir.

Kalıcı (persistent) veritabanı bağlantısı nedir?

Bu soru altında yer alacak açıklama Resmi PHP dökümantasyonunun 41. bölümünün bir çevirisi olup, belgenin asıl ve son haline http://php.net/manual/en/features.persistent-connections.php adresinden ulaşabilirsiniz.

Kalıcı bağlantılar, çalışmakta olan betik sonlandığında dahi sunucu ile istemci arasındaki iletişimin kopmadığı bağlantı türleridir. Herhangi bir kalıcı bağlantı isteğinde, PHP ilk önce belirtilen sunucuya `aynı özellikte' (daha önceden açılmış) varolan bir kalıcı bağlantının olup olmadığına bakıp, böyle bir bağlantının mevcut olması durumunda, varolan mevcut bağlantıyı, aksi halde yeni bir kalıcı bağlantı gerçekleştirip kendisini çağırana onu döndürecektir. Burada `aynı özellikte' bir bağlantı ile kast edilen şey, aynı sunucuya, aynı kullanıcı adı (ve istendiği taktirde aynı parola) ile açılmış bir bağlantıdır.

Herhangi bir ağ sunucusunun çalışma ve yük dağıtımı mantığı konusunda yeterli derecede bilgi sahibi olmayan kullanıcılar, kalıcı bağlantıların tam olarak ne oldukları konusunda yanılgıya düşebilirler. Özel olarak, kalıcı bağlantılar size, aynı bağlantı üzerinde bir `kullanıcı oturumu' açmanızı ya da transaction'larınızı daha esnek bir şekilde gerçekleştirmenizi ve daha bir çok şeyi sağlamazlar. Hatta, konunun tam olarak netleşmesi açısından, kalıcı bağlantılar size, normal bir bağlantı ile yapamayacağınız hiçbir özellik sunmazlar.

Bunun nedeni ağ sunucularının çalışma mantığı ile ilişkilidir. Ağ sunucunuzun sayfaların oluşturulmasında PHP'nin kullanımında izlediği üç yol vardır.

İlk metod, PHP'yi bir CGI sargısı olarak kullanmaktır. Bu yöntemde, ağ sunucunuza gelen bir PHP sayfası istemi için, PHP yorumlayıcısının bir emsali her seferinde yaratılıp, işi bittikten sonra kapatılacaktır. Emsalin işi bittiğinde, kendisi kapatılacağı için doğal olarak açtığı bağlantılar da onunla birlikte kapanacaktır. Böyle bir durumda, kalıcı bağlantı kullanmanın hiçbir anlamı yoktur, ki görüldüğü gibi bu durumda bağlantının bir kalıcılığı da kalmamıştır.

İkinci, ve en çok tercih edilen metod ise PHP'yi çoklu işlemci özelliğine sahip bir ağ sunucusu üzerinde modüler olarak çalıştırmaktır, ki bu şuan sadece Apache tarafından desteklenmekte. Çoklu işlemci özelliğine sahip bir ağ sunucusunda tüm sayfaların

sunulmasını sağlayan alt işlemleri (çoçuk işlemler) kontrol eden bir işlem (baba işlem) bulunur. Sunucuya bir istek ulaştığında, o an sunum yapmayan çoçuk işlemlerden birinin isteğe cevap vermesi sağlanır. Bunun anlamı, bir istemciden aynı sunucuya ikinci kez bir istek ulaştığında, ikinci isteği sunan çoçuk işlem farklı olabilir. Kalıcı bir bağlantıda ise, ardarda gelen herbir SQL servis isteği SQL sunucusuna kurulu olan aynı bağlantıyı tekrar kullanabilirler.

Son yöntem ise PHP'yi çoklu yiv desteğine sahip bir ağ istemcisinde eklenti olarak çalıştırmaktır. Şuan PHP 4, ISAPI, WSAPI ve NSAPI (Windows'da) gibi Netscape FastTrack (iPlanet), Microsoft Internet Information Server (IIS) ve O'Reilly WebSite Pro benzeri çoklu yiv desteğine sahip sunucularda eklenti olarak çalışmasına olanak sağlayan arayüzleri desteklemektedir. Çalışma mantıkları yukarıda bahsi geçen çoklu işlem desteğine sahip sunuculardaki ile aynıdır. (SAPI desteğinin PHP 3'de yer almadığını hatırlayınız.)

Eğer kalıcı bağlantılar herhangi bir ek işlevsellik sunmuyorsa, tam olarak görevleri nedir?

Cevap oldukça basittir: Verimlilik. Eğer veritabanı ile bağlantı kurulması yüklü bir iş ise kalıcı bağlantılar bunun için iyibir çözüm oluşturabilirler. Bu yükün ağır olup olmadığı ise bir çok etkene bağlıdır. Örneğin, karşı tarafın ne tür bir veritabanı olduğu, sunucu ile aynı sistem üzerinde yer alıp almadığı, veritabanı sunucusunun bulunduğu sistem yükünün derecesi ve daha bir çoğu. Özetlemek gerekirse, bağlantı kurulumunun güç gerçekleştiği durumlarda kalıcı bağlantılar size epeyce bir yardımda bulunabilir. İstemci işlemi, her yeni sayfanın işlenmesinde sunucuya tekrar tekrar bağlanmaktansa, tüm işlem ömrü boyunca bağlı kalır. Yani sunucuya bağlantı kuran herbir çocuk işlemin kendisine ait bir kalıcı bağlantısı olacaktır. Örneğin, SQL sunucusuna kalıcı bağlantı kurmuş 20 farklı işleminiz varsa, hepsi için ayrı ayrı 20 adet bağlantınız var demektir.

Fakat bu durum, kalıcı bağlantıların bağlantı sınırını aştığı bir veritabanında çeşitli dezavantajlara yol açabilir. Eğer veritabanınız aynı anda 16 adet simultane bağlantı sınırına sahipse, 17 istemcinin aynı anda bağlanmaya çalıştığı bir durumda, istemcilerden biri açıkta kalacaktır. Hatta eğer programınızda bir hatadan dolayı kalıcı bağlantılar hiç kapanmıyorsa, belki de sunucunuz hep o 16 bağlantı ile çıkmaza düşecektir. (Konu hakkında veritabanınızın terk edilmiş ve beklemede olan bağlantılar hakkındaki dökümantasyonuna göz atabilirsiniz.)

Uyarı: Kalıcı bağlantıların kullanımında akılda tutulması gereken bir kaç ufak nokta daha var. Eğer veritabanına kurulu kalıcı bir bağlantı üzerinde tablo kilitleme mekanizması ile çalışan bir betiğiniz varsa ve bir nedenden dolayı program akışında bu kilit tablo üzerinden kaldırılmazsa, aynı bağlantı üzerindeki diğer betikler ağ ya da veritabanı sunucusu baştan başlatılana kadar bloke olacaktır. Bir diğer husus ise, kalıcı bağlantılar üzerinde transaction kullanırken, transaction sonlamadan önce betiğin çalışması yarıda kesilirse, transaction bloğu kendinden sonraki betiğin ertelenmesine sebebiyet verecektir. Bahsi geçen her iki durumdan da kurtulmak için register_shutdown_function() fonksiyonunu kullanarak, her program çıkışında tablolar üzerindeki kilitleri temizleyip transaction'ları geri alacak bir mekanizma hazırlayabilirsiniz. Hatta en iyisi mi, tablo kilitleri ve transaction kullanan betiklerinizde kalıcı bağlantılar kullanmayın.

Toparlayacak olursak, kalıcı bağlantılar normal bağlantılar ile bire bir eşleşecek şekilde tasarlanmışlardır. Yani kalıcı bağlantı kullanan bir betiği, istediğiniz zaman programın işleyiş mantığında hiçbir aksamaya neden olmayacak şekilde normal bağlantı bir eşi ile değiştirebilirsiniz. Bu programın verimliliğini değiştirebilir (ki muhtemelen değiştirecektir), fakat işleyişine etki etmeyecektir.

4. Python Arayüzü

Bu bölümde Python ile PostgreSQL veritabanına psycopg bağdaştırıcısını kullanarak bağlantı kurmaya çalıştığınızda sık karşılaşılabilecek sorunlar hakkında bahsetmeye çalışacağız.

PostgreSQL tarafından desteklelen fakat Python dilinde karşılığı olmayan tip geçişlerini nasıl sağlarım?

Herhangi bir veri tipinin, veritabanı tarafından desteklenip de kullanılan dil tarafından desteklenmemesi (ya da tam tersi) çoğu zaman veritabanı ile etkileşimli programlar yazanlar için bir problem teşkil etmiştir. 246. Python Geliştirme Önergelersinde (*PEP*) bu konu Nesne Adaptasyonu (*Object Adaptation*) belgesinde incelenmiştir.

Bu başlık altında, psycopg2 ile *PEP* 246 ile tanımlanan nesne adaptasyonunu PostgreSQL veritabanı üzerinde nasıl gerçekleştireceğimiz hakkında basit bir örnek vermeye çalışacağız.

> *psycopg2, PEP 246 standartını tam olarak sağlamasa da bir çok işlevini yerine getirmektedir. Fakat bunun gelişimi ile ilgili iyileştirmeler 2.0 beta sürümlerinde devam etmektedir.*

İlk önce bu örnekte ne yapacağımızı dış hatları ile ortaya koymaya çalışalım. Elimizde içinde PostgreSQL `circle` tipinde bir alan bulunan tablo olsun. Yazacağımız bağdaştırıcı ile Python ve PostgreSQL arasındaki `circle` tipindeki verilerin dönüşümünü sağlamaya çalışacağız. Bunu yaparken `psycopg2.extensions` modülü tarafından sağlanan `new_type()` ve `register_type()` fonksiyonlarını kullanacağız.

> *Devam etmeden önce çok önemli bir noktayı burada açıklığa kavuşturmamız gerekmekte. psycopg, tip dönüşümleri iki sınıf altında toplanıyor. Bunlardan ilki `adapter` (bağdaştırıcılar) ile olanı – bu özelliğin yardımcı fonksiyonlarının içinde `adapter` ifadesi geçer (`register_adapter()` gibi). Burada PostgreSQL veri tiplerini Python dilindeki varolan karşılıklarına dönüştürürüz. (Veritabanındaki `money` tipini, Python'daki `float` tipine dönüştürmek gibi.) İkincisi ise, Python'da hiçbir dengi olmayan tiplerin dönüşümüdür. (Bu özelliğin fonksiyonlar `register_type()` gibi adından type sözcüğünü içerir.) Biz bu örnekte ikinci özellik hakkında bahsedeceğiz. Konu hakkında ayrıntılı bilgi için psycopg'nin kaynak kodu ile gelen `examples` dizini altındaki örneklere yada internet sitesinde yer alan wiki'ye göz atabilirsiniz.*

Peki psycopg bizim istediğimiz veri tipi ile karşılaştığında hangi bağdaştırıcıyı çalıştıracağını nereden anlıyor? Bunu yaparken sorgu sütunlarının `OID` değerlerini bularak, bunu algılayabilecek uygun bağdaştırıcıya sahip olup olmadığına bakıyor. Ve eğer o tipi kavrayan bir adaptör varsa, onu çağırıp ilgili veri üzerinde o bağdaştırıcı çağrısını gerçekleştirdikten sonra veriyi PostgreSQL *SQL* tümcesine (ya da tam tersi olarak PostgreSQL verisinden alıp Python dilindeki - bizim yarattığımız - karşılık gelen nesneye) aktarıyor.

psycopg2 tarafından ötananımlı olarak gelen bağdaştırıcılara bakacak olursak:

```
>>> import psycopg2.extensions
>>> for k, v in psycopg2.extensions.adapters.items():
...     print "%s\n-> %s" % (k, v)
...
(<type 'bool'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <type 'psycopg2._psycopg.Boolean'>
(<type 'buffer'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <type 'psycopg2._psycopg.Binary'>
(<type 'float'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <type 'psycopg2._psycopg.AsIs'>
(<type 'list'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <type 'psycopg2._psycopg.List'>
(<type 'datetime.time'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <built-in function TimeFromPy>
(<type 'str'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <type 'psycopg2._psycopg.QuotedString'>
(<type 'int'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <type 'psycopg2._psycopg.AsIs'>
(<type 'long'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <type 'psycopg2._psycopg.AsIs'>
(<type 'datetime.timedelta'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <built-in function IntervalFromPy>
(<type 'datetime.date'>, <type 'psycopg2._psycopg.ISQLQuote'>)
```

```
-> <built-in function DateFromPy>
(<type 'unicode'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <type 'psycopg2._psycopg.QuotedString'>
(<type 'datetime.datetime'>, <type 'psycopg2._psycopg.ISQLQuote'>)
-> <built-in function TimestampFromPy>
```

Görüldüğü üzere herbiri `psycopg2.extensions.adapters` listesinde tanımlı birer tip dizisinden ibaret. Yani istendiğinde bu tiplerden herhangi birinin üzerine kendi bağdaştırıcılarımızı yazabiliriz.

> *Varolan bir bağdaştırıcının üzerine nasıl kendi fonksiyonlarımızı yazacağımız ilerideki örnekten sonra daha net anlaşılacak.*

Fakat biz bu örnekte varolan bir bağdaştırıcının üzerine bir şeyler yazmaktansa, psycopg tarafından desteklenmeyen bir tip için sıfırdan bir bağdaştırıcının nasıl oluşturulacağı üzerinde duracağız.

Problem olarak elimizde şuna benzer bir tablomuz olduğunu varsayalım:

```
=>  \d baloncuklar
  Table "public.baloncuklar"
  Column  |  Type  | Modifiers
----------+--------+-----------
 baloncuk | circle | not null

=> SELECT baloncuk FROM baloncuklar;
  baloncuk
------------
 <(0,0),1>
 <(3,4),20>
(2 rows)
```

`psycopg2.extensions` ile gelen özelliklerin kullanımına geçmeden önce ihtiyacımız olan bir kaç ufak işlev var. Bunlar `<(x,y),r>` şeklindeki karakter katarını algılayacak bir sınıf (class) ve circle tipinin OID değerinden ibaret.

```
class Circle:
    """ <(x,y),r> katarından x, y ve r parçalarını ayırıp, bunu
    anlamlı bir bütün olarak saklayacak olan sınıf.
    """
    def __init__(self, girdi=None, imlec=None):
        if girdi:
            girdi_l = girdi.split(',')
            self.nokta = (float(girdi_l[0][2:]), float(girdi_l[1][:-1]))
            self.ycap = float(girdi_l[2][:-1])
        else:
            self.nokta, self.ycap = (0.0, 0.0), 0.0

    def __conform__(self, proto):
        """ Sınıf üzerindeki __conform__ işlevi, SQL tümcesine yerleştirilecek
        verilerin istenilen forma sokulmasında kullanılır. (Buradaki kullanım
        şekli pek doğru olmasa da işimizi görmektedir. Tam bir açıklama için
        PEP 246'ya bakılabilir.)
        """
        if proto == psyco_isqlquote:
            return self

    def getquoted(self):
        return "'<(%f,%f),%f>'" % (self.nokta[0], self.nokta[1], self.ycap)

    """ Bundan sonraki işlevler kişisel damak tadına tabidir. """

    def __str__(self):
        return self.getquoted()

    def __repr__(self):
        return "Circle(%s)" % self.__str__()
```

Oluşturduğumuz `circle` sınıfında yapılan açıklama satırlarının dışında `ISQLQuote` sınıfından bahsetmemiz yerinde olacaktır. psycopg bağdaştırıcı protokolü `psycopg2.extensions` altında tanımlanmış `ISQLQuote` sınıfından oluşmaktadır. Bu sınıfın dış hatlarını ile şu şekilde verebiliriz:

```
class ISQLQuote:
    def getquote(self):
        """ Girdinin ayıklanmış çıktısını döndürecek olan fonksiyon. """

    def getbinary(self):
        """ İkili girdinin ayıklanmış biçimini döndürecek fonksiyon. """

    def getbuffer(self):
        """ İşlenmiş girdinin kendisini döndürecek olan fonksiyon. """
```

Görüldüğü üzere bizim bu örnek için ihtiyacımız olacak tek işlev `getquote()` olduğu için biz de yukarıdaki `Circle` sınıfımızda sadece bu fonksiyonu geliştirdik ve yapılan çağrıların `ISQLQuote` için geldiği zaman `__init__()` fonksiyonunun başlatılması için `if proto = psyco_isqlquote` ifadesini ekledik.

Bundan sonra tüm bunları birleştirecek bir örneği şu şekilde verebiliriz:

```
from psycopg2 import connect as psyco_connect
from psycopg2.extensions import ISQLQuote as psyco_isqlquote
from psycopg2.extensions import new_type as psyco_type_new
from psycopg2.extensions import register_type as psyco_type_reg

class Circle:
    ...

bag = psyco_connect('dbname=test')
imlec = bag.cursor()

# circle tipinin bağdaştırıcısını oluşturmak
# için tipin OID değerine ihtiyacımız var.
imlec.execute("SELECT oid FROM pg_type WHERE typname = 'circle'")
circle_oid = imlec.fetchone()[0]

# Yeni bağdaştırıcı kaydediliyor
TYP_CIRCLE = psyco_type_new((circle_oid, ), "circle", Circle)
psyco_type_reg(TYP_CIRCLE)

imlec.execute("INSERT INTO baloncuklar VALUES (%s)", (Circle('<(5,6),7>'), ))
imlec.execute("SELECT baloncuk FROM baloncuklar")
print imlec.fetchall()
```

Yukarıdaki örnekte buraya kadar bahsetmediğimiz işlevler `psycopg2.extensions` altında tanımlı `new_type()` (`psyco_type_new()`) ve `register_type()` (`psyco_type_reg()`) fonksiyonları. Bunları şu şekilde özetlememiz mümkün:

`extensions.new_type``(oid_dizisi,tip_adı,tip_sınıfı)`

Girilen `OID` dizisine sahip, ilgili tip adı altındaki veri tiplerini fonksiyona parametre olarak gireceğiniz tip sınıfı ile Python-PostgreSQL arası bağdaştırmak için bu işlevden yararlanabilirsiniz.

`extensions.register_type``(veri_tipi)`

`extensions.new_type()` ile oluşturulan bir veri tipi kapsülünün psycopg tarafından kullanılması için yine extensions modülü altındaki `register_type()` işlevinden yararlanabilirsiniz.

Programa örnek bir çıktıyı şu şekilde verebiliriz:

```
# Veritabanına Circle() çağrısı ile girilen verilerin, veritabanından alırken
# de direk olarak Circle() sınıfı tarafından algılandığına dikkat ediniz.
~$ python circle_tipi.py
[(Circle('<(0.000000,0.000000),1.000000>'),),
(Circle('<(3.000000,4.000000),20.000000>'),), (Circle('<(5.000000,6.000000),7.000000>'),)]
```

Yukarıda da bahsettiğimiz çalıştığımız gibi, bu örnekte Python tarafında karşlığı olmayan bir veri tipinin nasıl oluşturulup hem sunucu tarafında hem de programlama dili tarafında sorunsuzca kullanılabileceği üzerinde durmaya çalıştık. Bunun dışında şöyle bir soru sormamız mümkün: Peki sunucu tarafındaki bir veri tipini programlama dili tarafından sağlanan bir tip ile nasıl örtüştürebiliriz? Bunun için yapılması gerekenler de oldukça basittir. Fakat biz bu konuyu burada irdelemeyeceğiz. Her iki hal de birbiri ile neredeyse tamamen benzerlik gösterdiğinden konu ile ilgili olarak psycopg'nin kaynak kodundaki `examples` dizini altında gelen `myfirstrecipe.py` örneğine bakılacak olursa aradaki bağlantı kolaylıkla kurulacaktır.