MySQL

BU NOT DEFTERİ HAKKINDA

- Bu not defteri veritabanıyla ilgili aldığım dersler ve ardından aldığım pek çok kurstan sonra sonra mysqltutorial.org adresindeki eğitimlerin büyük bir kısmını uygulayarak aldığım notlar ve internet üzerinden yaptığım araştırmalar sonucu oluşan notlardan müteşekkildir.
- Oluşturduğum örnekleri denedim, çalışması gerekiyor; fakat ufak tefek hatâ görürseniz lütfen bildirin.
- Bu notlara ulaşmak için Apache 2.0 açık kaynak lisansı altındaki Github hesâbımdaki (github.com/369553) defterler deposuna bakabilirsiniz.
- Bu not defterinin kapsamı 'sıfırdan ileri seviye ortalarına kadar' olarak nitelendirilebilir.
- Not defterinde ifâdelerin söz diziminde kullanılan köşeli parantez ('[]') o ifâdenin zorunlu olmadığı, tercihî olduğu anlamını taşıyor; kullanılan 'pipe' işâreti ('|') ise farklı seçenekleri birbirinden ayırmak için kullanılıyor; yine kullanılan süslü parantez ('{}') ise bu seçenekleri gruplamak için kullanılıyor; yanî bunlar söz dizimine dâhil değil. Misal, `INT UNSIGNED` tanımında `UNSIGNED` ifâdesinin zorunlu olmadığını belirtmek için bu ifâde köşeli parantez içerisine alınırken, başka bir seçenek olan `SIGNED` seçeneğini de berâber ifâde etmek için şu metîn yer alır: `INT [UNSIGNED | SIGNED]` Buna göre, `INT`, `INT UNSIGNED` ve `INT SIGNED` ifâdeleri geçerlidir. Eğer bu seçenekler tercihe dayalı olmasaydı, seçenekler arasında '|' işâreti kullanılırdı ve tüm seçenekler süslü parantez içerisinde yer alabilirdi.
- Bilhassa not defterinin temel kısımlarında yalın bir üslûp kullanmaya çalıştım, hayırlı çalışmalar dilerim.
- Yazar, Mehmet Âkif SOLAK.

GENEL BİLGİLER

- MySQL Genel Kamu Lisansı (GNU Public License) altında açık kaynak olarak sunulan, geniş bir topluluk desteği bulunan, güçlü bir ilişkisel veritabanı yazılımıdır.
- Birçok ilişkisel veritabanında olduğu sunucu bazlı çalışmaktadır; yanî yazdığınız kodlar ağ protokolleri üzerinden MySQL'e iletilir ve işlenir.
- MySQL, ilişkisel veritabanı dili olan 'Yapısal Sorgu Dili' ('Structured Query Language', SQL)'ni destekler; ilâveten pek çok özelliği, kolay bir yazım üslûbuyla sunar.

- Birden fazla komut çalıştırılacaksa her komut arasını noktalı virgülle (';') ayırın.
- Bir satırı yorum satırı yapmak için şu iki karakterden birisini kullan : `#` veyâ `--`
- Birden fazla satırı tek bir yorum hâline getirmek için bu satırları `/*...*/` karakterleri arasına alın (üç nokta olan yere yorum satırlarınızı yazın).
- MySQL komutları varsayılan olarak büyük - küçük harf duyarsızdır.
- Genel kullanım şekli komutların büyük harfle yazılması şeklindedir.
- MySQL'de bir komutun bir satırda yer alması şart değildir; komutu okunabilirlik için alt satıra kaydırabilirsiniz ve bu yaygın olarak kullanılır.
- MySQL'de metînsel veriler çift tırnak veyâ tek tırnak içerisinde yazılır.
- MySQL'de sayısal veriler tırnaksız şekilde yazılır.
- Bilgisayarımızda kurulu MySQL veritabanının sürüm numarasını öğrenmek için 'mysql' ortamında veyâ MySQL Workbench içerisinde `SELECT VERSION();` kodunu çalıştırın.
- MySQL'i indirmek istiyorsanız şu adresi ziyâret edin: https://www.mysql.com/downloads/
- MySQL'in asıl öğrenim kılavuzu yeni başlayan için çok detaylı ve dolayısıyla zorlayıcı gelebilir, bu sebeple ilk başta yeni başlayanlar için hâzırlanmış olan kılavuzu (https://dev.mysql.com/doc/refman/8.0/en/tutorial.html) okumak isteyebilirsiniz; fakat ilerledikten sonra bu adresi ziyâret etmeniz münâsip olur: https://dev.mysql.com/doc/
- Sorgu yazmaya başlamadan evvel `USE veritabaniIsmi;` kodunu çalıştırarak bir veritabanı seçin, `SELECT DATABASE();` kodunu çalıştırarak hangi veritabanını seçtiğinizi görün.

İLİŞKİSEL VERİTABANININ GENEL YAPISI

- Bir ilişkisel veritabanı tablo, tetikleyici, derlenmiş komutlar ('stored procedure') gibi pek çok yapıdan oluşur; fakat veritabanının kalbi, verilerin saklandığı yer olan tablolardır.
- Tablolar satır ve sütunlardan oluşur. Satırlar verileri, sütunlar ise verinin özelliklerini belirtir. Misal, müşteri bilgilerinin tutulduğu bir ilişkisel veritabanında her bir satır yeni bir müşterinin bilgisini içerirken, sütunlar ise **veri tipi belirtilmiş** özel bir veriyi saklar; meselâ müşterinin 'isim', 'soyisim', 'numara' verileri ayrı sütunlarda tutulur.
- İlişkisel veritabanları satırlar arasında performanslı şekilde arama yapacak şekilde tasarlanırlar.

- Veritabanından istediğimiz şarta uyan satırların istediğimiz sütun verilerini istediğimiz sıralamayla almamız ilişkisel veritabanlarında yapabildiğimiz temel bir işlemdir.
- Verilerin tekrarlanmaması ve işlemlerin performanslı şekilde icrâ edilmesi gibi amaçlarla tabloların normalizasyonu yapılmalı ve tablolar uygun şekilde sütunsal olarak bölünmelidir. Misal, müşterinin alışveriş verileri ayrı bir tabloda, müşteri bilgileri ayrı bir tabloda tutulmalıdır. Bu durumda tablolar arasında geçiş yapabilmek ve farklı tablolardan verileri çekerek tek bir tablo olarak görüntüleyebilmek için tabloda her satırı ifâde eden bir anahtar olması gerekmektedir. Buna, **birincil anahtar** denir. Bu birincil anahtar başka tablonun dış anahtar bölümüne bağlanarak veri bütünlüğü ve veriler arasındaki ilişki sağlanır. Böyle yapıldığında veriler bir arada tutulmaz; sadece iki tablo arasında eşleştirme yapılarak sorgulama yapılması sağlanır; dolayısıyla performans ve hafıza açısından daha etkin bir sistem kurulmuş olur. Örnekte soldaki tablodaki 'id' alanı birincil anahtar iken, sağdaki tabloda bu alanın bağlandığı yer 'costumerId' alanı dış anahtardır. Bu sayede havlu ve bardak ürünlerini Mehmet isimli kişinin aldığı görülebiliyor:

id	name	surname	##	id	costumerId	product
1	Âkif	SOLAK	##	1	2	"Havlu"
2	Mehmet	SOLAK	##	2	2	"Bardak"

- Bir tablo içerisinde bulunan ve başka tablonun birincil anahtarını işâret eden sütuna **dış anahtar** denir.

MySQL VERİ TİPLERİ

- Temel düzeyde SQL bilmiyorsanız, bu kısmı sonra okumak isteyebilirsiniz; veri tipleri detaylı şekilde anlatılmaktadır.

1) BIT

- 0 ve 1 değerlerinden başka değer barındırmayan veri tipidir.
- `sutunIsmi BIT` biçiminde tanımlanabilir.
- Sütunun kaç adet bit tutacağını belirtebilirsiniz: `BIT(12)`
- Sütunun kaç bitlik veri tutacağı belirtilmezse varsayılan olarak bir bit tutar.
- `BIT` veri tipindeki bir sütuna veri eklerken derleyicinin belirtilen veriyi tamsayıdan ayırabilmesi için `b'deger'` veyâ `0bdeger` şeklinde yazmalıyız:

```
INSERT INTO data(name, gender)
VALUES ('Ahmed', b'1');
```

- **! Yukarıdaki sözdiziminde `b` harfi ASLA BÜYÜK HARFLE YAZILMAMALIdır.**
- MySQL içerisinde `BOOLEAN` veri tipindeki veriler `TINYINT(1)` formundaki `BOOLEAN` veri tipiyle de tutulabiliyor. `BIT` ile `BOOLEAN` arasında fark `BOOLEAN` veri tipi `TINYINT(1)` formatında `BOOLEAN` veri tipiyle tanımlanmış sütuna `0` ve `1` dışında bir veri ekleyebilirsiniz. `0` olmayan tüm veriler `TRUE` gibi değerlendirilir.
- Sütuna veri eklerken doğrudan `TRUE` ve `FALSE` değerini yazabildiğiniz gibi `1` ve `0` da yazabilirsiniz. Sorgulama yaparken de bu özellik kullanılabilir. `BIT` veri tipi genişleyerek birden fazla bit alabilirken, `BOOLEAN` veri tipi sâbit olarak 1 byte uzunluktadır. `BIT` veri tipinde her bit karşılaştırılır; `BIT` veri tipi `BOOLEAN` ifâdeler için daha iyi olabilir.

2) INT

- Tamsayı tutmak için kullanılan veri tipidir.
- En çok kullanılan veri tipi olmasına karşın küçük sayılar için `TINYINT` ve `SMALLINT` orta büyüklükteki sayılar için `MEDIUMINT` ve büyük tam sayılar için `BIGINT` veri tipleri kullanılabilir; doğru veritabanı tasarımı için uygun olanı kullanmak gerekir.
- Tıpkı programlama dillerinde olduğu gibi işâretli veyâ işâretsiz olarak iki farklı çeşitte kullanılabilir. Varsayılan olarak işâretli veri barındırır; eğer bir sütunun negatif değerler alması söz konusu değilse işâretsiz olarak kullanarak bir kat daha fazla kapasite sağlanabilir.
- Tanımı:

```
sutunIsmi INT [SIGNED | UNSIGNED]
# Köşeli parantez içinde yazan kısım tercihîdir;
# SIGNED veyâ UNSIGNED yazılabilir.
```

- Tamsayı veri tipi alan optimizasyonu açısından farklı üst sınırlara sâhip alt veri tiplerine dönüştürülmüştür; ilgili sütun için en az yer kaplayan veri tipi seçilmelidir; aşağıda işâretli tam sayılar için veri tiplerinin değer aralığı verilmiştir:

Veri Tipi	Boyutu	Değer aralığı
TINYINT	1 byte	[-128; 127]
SMALLINT	2 byte	[-32768; 32767]
MEDIUMINT	3 byte	[-8,388,608; 8,388,607]
INT	4 byte	[-2,147,483,648; 2,147,483,647]
BIGINT	8 byte	[-9,223,372,036,854,775,808; 9,223,372,036,843,775,807]

- İşâretsiz tamsayı veri tipleri negatif değer alamadığından değer aralığı işâretli veri tiplerinin üst sınırının 2 katından bir eksiktir. Misal, `TINYINT` için 127*2-1 = 255

3) DECIMAL

- Sayı tutmak için kullanılan esnek bir veri tipidir.
- Tutmak istediğiniz sayının basamak sayısını ve kesir basamak sayısını belirleme imkânınızın olduğu bir veri tipidir.
- Tanımı: `sutunIsmi DECIMAL(ToplamBasamak, kesirBasamak)`
- Toplam basamak sayısı 1-65 arasında değer alır. Kesir kısmının basamak sayısı ise 0 ile 30 arasında değer alır.
- Kesir kısmının basamak sayısının her zamân için toplam basamak sayısından küçük olması gerektiği unutulmamalıdır.
- Eğer eklenmek istenen verinin **kesir kısmının basamak sayısı belirtilenden daha fazla ise, sayı yuvarlanarak istenilen formata getirilir**; daha az ise ilgili kısımlar 0 ile doldurulur.
- Eğer eklenmek istenen verinin **tamsayı basamak sayısı belirtilenden fazla ise veri eklenmez ve sistem hatâ döndürür**.
- Misal:

```
CREATE TABLE veri(sayi DECIMAL(3, 2));
INSERT INTO veri(sayi) VALUES (1.12, 1.129, 1.1, 1);
SELECT * FROM veri;# Çıktısı aşağıdaki gibidir:
# sayi
# 1.00 --> '00' eklenerek kesir kısmı doldurulmuş
# 1.10 --> '0' eklenerek kesir kısmı doldurulmuş
# 1.12 --> Belirtilen formata uygun veri
# 1.13 --> Kesir kısmının basamak sayısı fazla olduğundan yuvarlanmış
```

> ***NOT :*** `DECIMAL` **veri tipiyle çalışırken dikkat etmemiz gereken önemli bir husus ilk yazdığımız basamak sayısının toplam basamak sayısı olduğunu unutmamamız gerektiğidir**. Misal, `DECIMAL(3,2)` tanımında kesir kısmının basamak sayısı 2 olduğundan tamsayı kısmı sadece tek basamaklı bir değer alabilir.

- `DECIMAL` ifâdesi yerine `DEC` , `FIXED` veyâ `NUMERIC` ifâdesi yazılabilir.
- MySQL `DECIMAL` veri tipi için `BINARY` formatı kullanmaktadır. Her 9 basamak sayısı için 4 byte alana ihtiyaç vardır. Misal, `DECIMAL(19,9)` formatındaki verinin 9 basamaklık kesir kısmı için 4 byte alana, 10 basamaklık tam sayı kısmının 9 basamağı için 4 byte alana ihtiyaç vardır. Gerekli bir basamak için ise 1 byte alana ihtiyaç vardır; yanî basamak sayısı 9'un katları şeklinde ilerlemediğinde gerekli alan da 4 byte olarak ilerlememektedir. 9'dan sonraki basamak sayılarının sayısına göre gerekli alan şöyledir:

Artık basamak sayısı	Gerekli alan
1-2	1
3-4	2
5-6	3
7-9	4

> ***NOT :*** `DECIMAL` veri tipi parasal veriler için tercih edilen bir veri tipidir.

- DECIMAL veri tiplerinde kesir kısmı sınır değerini aşarsa yuvarlanır; fakat kullanıcıya uyarı da verilir.

4) FLOAT

- Noktalı sayı tutmak için kullanılır; genellikle noktalı sayının hayâtî öneme sâhip olmadığı ve noktadan sonraki basamak sayısının çok olmadığı durumlarda tercih edilir.
- Sütun tanımı : `sutunIsmi FLOAT(toplamBasamak, kesirBasamak)`
- 4 byte yer kaplar.

- Sütun tanımında belirtilen basamak sayısı verinin kapladığı alanı etkilemez.
- Basamak sayısı belirtilmeyebilir.
- Eğer sayının kesirli kısmı istenenden uzun ise, kesir kısmı bir üste basamağa yuvarlanarak bu sayı düşürülür.

5) DOUBLE

- Hassas kesirli kısım içeren sayılar için kullanılan bir veri tipidir.
- 8 byte yer kaplar.
- Sütun tanımı : sutunIsmi `DOUBLE(toplamBasamak, kesirBasamak)`
- Basamak sayısı belirtilmeyebilir; bu veri tipinin boyutunu etkilemez.
- **Çok geniş bir kesir aralığı sağlamasına rağmen, `DOUBLE` veri tipinin kesir aralığı aşılırsa sayının kesir kısmı otomatik olarak yuvarlanır.**

6) CHAR

- Karakter depolamak için kullanılan metîn veri tipidir.
- **Verileri sâbit uzunlukta tutar**; bu uzunluk sütun oluşturulurken belirtilmelidir.
- Her karakter 1 byte yer kaplar; yanî 'Unicode' karakterler desteklenmez.
- Tanımı : `sutunIsmi CHAR(karakterSayisi)`

> ***NOT :*** Eğer sütun oluşturulurken karakter sayısı belirtilmezse sütun tek karakterlik veri tutar. Yanî, `sutunIsmi CHAR` = `sutunIsmi CHAR(1)`

- Karakter varsayılan olarak veritabanı sisteminin karakter setini kullanıyorlarmış.
- Eğer eklenmek istenen metnin karakter sayısı sütunun karakter sayısından az ise veritabanı metni otomatik olarak boşluk ile doldurur; fakat **sorgu ile bu metni çekmek istediğinizde size boşluklarını kaldırarak getirir.**
- Bu veri tipi sâbit uzunluktaki veriler için (telefon numarası, kimlik numarası gibi) kullanılabilir.
- Veritabanı dinamik uzunluktaki metînsel bir alanda sorgu yaparken evvelâ metnin uzunluğunu hesaplar (belki hesaplamak yerine kaydedip kaydettiği yere bakıyor olabilir); sâbit uzunluktaki metînlerde ise bu işleme gerek olmadığından sorgu daha hızlı işletilir.
- `CHAR` veri tipinin azamî uzunluğu 255'tir (MySQL).

7) VARCHAR

- Dinamik uzunluktaki bir metînsel veri tipidir.
- Tanımı : sutunIsmi `VARCHAR(karakterSayisi)`
- Tanımdaki 'karakterSayisi' sütunun en fazla alabileceği karakter sayısıdır. **Karakter sayısı mutlaka belirtilmelidir; aksi hâlde hatâ verir!**
- Unicode metînleri destekler; `CHAR`'ın aksine `VARCHAR` veri tipinde her satır için bir veyâ 2 byte ziyâde alan harcanır; bu alan verinin kaç byte uzunluğunda olduğunu belirten bir ön veri (prefix)dir. Eğer metîn için gerekli alan 255 byte'dan büyükse önveri 2 byte boyutundadır; diğer durumda ise önveri 1 byte uzunluğundadır.
- Azamî uzunluğu diğer sütunlara göre değişmektedir; bunun sebebi MySQL'de bir satırın en fazla 65535 byte yer kaplayabiliyor olmasıdır.
- Tek bir sütun veri tutan bir tablo için `VARCHAR` veri tipinin uzunluğu 16383 olabilir. Eğer metînsel veri tutan bir sütun daha olursa bu uzunluğu paylaşmaları gerekir.

8) TEXT

- Metînsel veri tutarken sınırların büyük ölçüde kalktığı bir veri tipidir.
- Her karakter 1 byte yer kaplar.
- 4GB uzunluktaki bir metni tutabilir.
- **MySQL bu veri tipiyle tanımlanmış verileri veritabanı sunucu hâfızasında değil, yerel diskte depolar**; bu da bu veri tipiyle tanımlanmış alanlar üzerinde yapılan sorgulamaların **daha yavaş** çalışmasına sebep olmaktadır.
- Veri eklerken veyâ veri çekerken boşluk ekleme ve silme gibi işlemler uygulanmaz.
- Sütun tanımı : `sutunIsmi TEXT`
- MySQL'de TEXT veri tipinin 4 alt çeşidi bulunmaktadır:

Veri Tipi	Azamî hacmi	Ek yük
TINYTEXT	255 byte (255 karakter)	1 byte
TEXT	64 KB (65,535 karakter)	2 byte
MEDIUMTEXT	16 MB (16,277,215 karakter)	3 byte
LONGTEXT	4 GB (4,294,967,295 karakter)	4 byte

- Ek yük, kuvvetle muhtemel metnin uzunluğunu tutmak için kullanılmaktadır.

9) DATETIME

- Hem târih, hem de saat saklanabilen bir veri tipidir.
- Metîn gibi çift tırnak içerisinde yazılır; fakat belli bir formatı vardır: `'YYYY-MM-DD HH:MM:SS'` Buradaki formata göre; yıl 4 basamak olarak, ay 2 basamak olarak, gün 2 basamak olarak ve saat kısmında ise sırasıyla saat, dakîka, saniye 2'şer basamak olarak yazılmalıdır. Misal : `'2024-11-12 12:51:29'`
- Sütun tanımı : `sutunIsmi DATETIME`
- Târih ve saat tipindeki verilerde varsayılan olarak o anki değerin eklenmesi çokça ihtiyaç duyulan bir işlemdir. MySQL'de bunu `CURRENT_TIMESTAMP` ile veyâ `NOW()` fonksiyonuyla yapabilirsiniz:

```
CREATE TABLE records(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50),
    date_ DATETIME DEFAULT NOW(),# O anki târih saat değeri atanır
    date2 DATETIME DEFAULT CURRENT_TIMESTAMP# Aynı işlev...
);
```

- `DATETIME` verisi 5 byte yer kaplar. En küçük değeri `1000-01-01 00:00:00` değeridir. En büyük değeri `9999-12-31 23:59:59` değeridir.
- `DATETIME` alanı varsayılan olarak milisaniye verisi tutmaz; fakat istenilirse tutması sağlanabilir. Bunun için saniyeden sonra kaç basamak kullanılmak istendiği bilgisi sütunu tanımlanırken belirtilmelidir. Misal, `DECIMAL(2)` tanımı saniyeden sonra 2 basamak hassâsiyyette veri tutulacağını ifâde etmektedir.
- Saniyeden sonra tutulacak basamak sayısı için ilâve alana ihtiyaç vardır. Aşağıdaki tablo saniyeden sonraki basamak sayısına göre gereken alanı göstermektedir:

Basamak sayısı	Gerekli alan (byte)
1-2	1
3-4	2
5-6	3

- Eklenmek istenen târih - saat verisinin saniyeden sonraki kısmı belirtilen formata uygun değilse uyarlanır; yanî fazla basamak sayısı belirtildiyse yuvarlanır, az belirtildiyse sıfır ile doldurulur.

- `DATETIME` ile tanımlanmış bir alanın değeri tablodaki ilgili satırdaki herhangi bir sütun tazelendiğinde otomatik olarak tazelenecek şekilde ayarlanabilir:

```
# Anlamak için tablo oluşturalım:
CREATE TABLE SysLog(
name VARCHAR(50), activeClientId INT, changedInfo VARCHAR(50),
time DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP);
# Veri ekle:
INSERT INTO SysLog(name, activeClientId, changedInfo)
VALUES ('user1', 2, 'profileName');
# Veriyi tazelediğimizde 'time' verisi otomatik olarak tazelenir:
UPDATE SysLogSET activeClientId = 1 LIMIT 1;
```

10) TIMESTAMP

- Bir başka târih - saat verisi veri tipidir.

- Sütun tanımı : `sutunIsmi TIMESTAMP` şeklindedir.

- `TIMESTAMP` veri tipi 4 byte'lık bir yer kaplar.

- Değer aralığı `1970-01-01 00:00:01 UTC` değerinden `2038-01-19 03:14:07 UTC` değerine kadar olan kısımdır.

- `TIMESTAMP` veri tipinin en büyük farklılığı tuttuğu târih değerinin küçük bir aralığı simgelemesi ve tuttuğu târih değerinin UTC zamân dilimince tutulmasıdır.

- Tuttuğu târih verisinin UTC zamân dilimine göre tutulmasından ötürü veritabanının zamân dilimi değiştiğinde ilgili târih de buna göre değişir.

- `TIMESTAMP` veri tipi ile tanımlanmış bir sütun da veri tazelendiğinde otomatik tazelenme özelliğine sâhiptir.

11) DATE

- `DATE` alanı sadece târih bilgisi tutar.

- Şu formattadır : `'YYYY-MM-DD'`

- Asgarî '1000-01-01' târih değerini, azamî '9999-12-31' târih değerini tutabilir.

- 3 byte yer kaplar.
- Tanımı : `sutunIsmi DATE`

12) TIME

- Sadece saat verisi tutar.
- Veriyi bu formatta alır ve verir : `'HH:MM:SS'`
- `DATETIME` verisinde olduğu gibi saat verisi saniyeden 6 basamak hassâsiyyette veri tutabilir.
- Saniyeden sonra veri barındırmayan `TIME` veri tipi formu 3 byte yer kaplar. Saniyeden sonra veri barındırılması için ilâve alana ihtiyaç vardır:

Basamak sayısı	Gerekli alan
1-2	1
3-4	2
5-6	3

- Saniyeden sonra kaç basamak hassâsiyyette veri tutulmak isteniyorsa tablo oluşturulurken şu şekilde belirtilmelidir: `sutunIsmi TIME(basamakSayisi)`

13) BINARY

- Sâbit uzunluklu ikili (binary) verileri tutmak için kullanılır.
- Anlaşılacağı üzere bu veri doğrudan bir metîn verisi gibi okunamaz.
- İmza ve özet (hash) gibi sâbit uzunlukların tutulması için, kriptolojik verilerin tutulması için uygundur.
- Sütun tanımı : `sutunIsmi BINARY(boyut)`
- 'boyut' byte cinsinden belirtilmelidir, boyut belirtilmezse sütun boyutu 1 byte olur.
- Veri eklenirken, eklenmek istenen veri sütunun büyüklüğünden az ise, veritabanı sistemi verinin **sağ tarafına 0 ekleyerek boşlukları doldurur; veriyi çekerken bu boşlukları kaldırmaz**.
- Bu sütun üzerinde `WHERE`, `ORDER` veyâ `DISTINCT` gibi sorgu deyimleriyle sorgu yaptığınızda veritabanı sistemi tüm byte'ları karşılaştırır.

- Sıralama yaptığınızda `NULL` byte'lar boşluk karakterinden önce gelmektedir.
- İkili veri girişi birkaç farklı şekilde yapılabilir. Birincisi `0x` ile 16'lık formatta veri girişi; ikincisi ise verinin başına `0b` yazarak ikili formatta veri girişidir:

```
INSERT INTO veri(ikiliVeri) VALUES (0x11);# hex 11 = dec 17
INSERT INTO veri(ikiliVeri) VALUES (0b101);# bin 101 = dec 5
INSERT INTO veri(ikiliVeri) VALUES (b'101');# Yukarıdaki ile aynı
```

- BINARY veriyi sorgularken `HEX()` fonksiyonunu kullanarak 16'lık formatta alın:

```
SELECT HEX(ikiliVeri) FROM veri;
```

- Pek çok kriptoloji anahtarı, imza özet bilgileri hexadecimal formatta tutulmaktadır; fakat bunlar bir fonksiyondan dönen veriler olmaktadır. Bu durumda gelen 16'lık (hexadecimal) formattaki veriyi ikili veriye çevirecek bir fonksiyona ihtiyacımız var:

```
INSERT INTO veri(ikiliVeri)
VALUES (UNHEX(SHA2('anahtar', 256));
# 1->'anahtar' ifâdesinin SHA256 özeti alınır (16'lık formatta)
# 'UNHEX()' ile gelen 16'lık formattaki veri ikili formata çevrilir
```

14) VARBINARY

- Bu veri tipi de ikili formatta veri tutar; `BINARY` veri tipinden farkı sâbit değil, dinamik uzunlukta veri tutmasıdır. Yanî verinizin sütun boyutundan küçükse verinizin sağına 0'lar eklenerek doldurma ('padding') işlemi yapılmaz.
- `VARBINARY` veri tipini tanımlarken boyut belirtmek zorundasınız. Sütun tanımı : `sutunIsmi VARBINARY(boyut)`
- Yukarıdaki tanımdaki 'boyut' bir verinin azamî büyüklüğüdür. Belirtilen boyuttan küçük veri eklenirse veri boşlukla doldurulmaz, kendi miktarı kadar yer kaplar; belirtilen boyuttan büyük veri eklenmek istenirse sistem hatâ verir.

15) BLOB

- Büyük ikili verileri tutmak için kullanılan veri yapısıdır. Yanî resim, video gibi verileri bu veri tipiyle tanımlanmış sütunda tutabilirsiniz.
- Dinamik bellek tahsîsi sistemine göre çalıştığından **dosya boyutu + ek yük kadar yer kaplar**.

- Sütun tanımı : `sutunIsmi BLOB`
- BLOB veri tipi bu çerçevedeki 4 veri tipinden birisidir, aralarındaki fark hacimleridir:

Veri Tipi	Boyut
TINYBLOB	255 byte
BLOB	64 KB (65,535 byte)
MEDIUMBLOB	16 MB (16,777,215 byte)
LONGBLOB	4 GB (4,294,967,295 byte)

- `BLOB` veri tipiyle çalışırken dikkat etmemiz gereken şey dosyanın nereye yüklendiği olmalıdır. MySQL veritabanı sisteminin dosyayı yüklediği yere yükleme yapmazsak, dosyayı okumak istediğimizde NULL sonucunu alabiliriz.
- MySQL'de sistemin dosyaları yüklemek için kullandığı yeri öğrenmek için sistem değişkeni olan `@@secure_file_priv` değişkenini kullanabilirsiniz:

```
SELECT @@secure_file_priv;
```

- `BLOB` veri tipiyle tanımlanmış bir sütuna dosya yüklerken LOAD_FILE() fonksiyonunu kullanınız:

```
INSERT INTO veri VALUES (
LOAD_FILE('C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\resim.png'));
```

- **! Sistem belirtilen adreste dosyayı bulamazsa hatâ vermez; `NULL` değerini ekler; çünkü `LOAD_FILE()` fonksiyonu dosyayı bulamazsa `NULL` değeri döndürür.**

> ***ÖNEMLİ NOT :*** Windows işletim sistemi için belirttiğiniz adresteki ters ayraç işâretleri normal ayraç hâline getirilmelidir. Misal, yukarıdaki için Windows dosya yolu şudur : `C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/resim.png`

- **! Sistem `@@secure_file_priv` değişkeniyle belirtilen konumun dizinin dışındaki bir dizinden dosya yüklemez!** Dosyalarınızı o dizine yüklemelisiniz.
- `BLOB` veri tipinde `TINYBLOB` için 1 byte, BLOB için 2 byte, `MEDIUMBLOB` için 3 byte ve `LONGBLOB` için 4 byte fazladan alana ihtiyaç vardır.

16) ENUM

- Bu veri tipi tıpkı programlama dillerinde vâr olan ENUM veri tipi gibi kullanıcının tanımladığı değerlerin alınabildiği bir metînsel veri tipidir. Hiçbir kayıt bu seçili değerler dışında bir değer alamaz.
- Bu değeler sütun tanımlanırken belirtilir: `sutunIsmi ENUM('veri1', 'veri2')`
- Veri ekleme işlemi normal metîn verisi ekleme gibidir; eğer eklemek istediğiniz değer sütun tanımlanırken belirtilen seçili veriler arasında yoksa, hatâ alırsınız.
- Verinin listedeki sırasını belirterek de veri ekleyebilirsiniz:

```
INSERT INTO veri VALUES(2);
# Sütun oluşturulurken 2. sırada yazılan veri tabloya eklenir.
```

- Verinin listedeki sırasını belirterek veriye ulaşma işlemini sorgulama yaparken de kullanabilirsiniz.
- `ENUM` veriler veritabanı sisteminde tamsayı olarak tutulur.

17) BOOLEAN

- Bitsel veriler için `BIT` veri tipini de kullanabilirsiniz.
- `BOOLEAN` veri tipi aslında `TINYINT(1)` biçiminde bir veri tipidir.
- Sütun tanımı : `sutunIsmi BOOLEAN` veyâ `sutunIsmi BOOL`
- Sütuna veri eklerken doğrudan `TRUE` ve `FALSE` değerini yazabildiğiniz gibi `1` ve `0` da yazabilirsiniz. Sorgulama yaparken de bu özelliği kullanabilirsiniz.
- Aralarındaki fark için '1) BIT' alt başlığına bakınız; kullanımı aynıdır.

18) JSON

- NoSQL (İlişkisel olmayan) veritabanı sistemleri bâzen sorunumuz için doğru bir seçenek olmaktadır. İlişkisel veritabanları da bu sebeple ilişkisel olmayan veritabanlarının son derece uyum sağladığı `JSON` veri tipini desteklemeye başlamıştır. MySQL 5.7.8 sürümünden beri `JSON` veri tipini desteklemektedir.
- `JSON` verisinin bir metîn olarak tutulması o verinin içerisindeki değişkenlere, dizilere doğrudan ulaşamayacağımızı, sorgu yazarken onları kullanamayacağımızı ifâde etmektedir. Bunun aksine `JSON` veri tipi buna imkân tanımaktadır / tanıyabilmektedir.
- Sütun tanımı : `sutunIsmi JSON`

- Sütuna veri eklerken metîn verisi ekler gibi JSON metni eklenir.
- Daha fazla bilgi için **SQL içinde NoSQL** kısmına bakınız.

TEMEL KULLANIM ve BASİTÇE KAYIT SORGULAMA

- Bir veritabanı üzerinde çalışmadan evvel o veritabanında çalışacağımızı belirtmek için şu şekilde kod çalıştırmalıyız:

```
USE publish;# 'publish' isimli veritabanı seçilmiş oluyor
```

- Bir tablodan kayıt sorgulama basitçe şöyledir:

```
SELECT id FROM articles;
# 'articles' tablosundaki tüm satırların 'id' sütununun
# bilgisi getirilir
```

- Bir tablodan birden fazla sütunun bilgisini getirmek için sütun isimlerini aralarına virgül koyarak yaz:

```
SELECT id, name FROM articles;
# 'articles' tablosundaki tüm satırların 'id' ve 'name' sütunundaki
# veriler getirilir
```

- Bir tablodaki tüm sütunların verisini almak için `*` karakteri kullanılır:

```
SELECT * FROM articles;
# 'articles' tablosundaki tüm satırların
# tüm sütunlarındaki veriler getirilir.
```

> ***NOT :*** MySQL'de sütun ismi ve tablo ismi boşluk içeremez; fakat '-' gibi bâzı özel karakterler içerebilir; bu tür durumda yazdığımız sorgu, derleyici tarafından hatâlı şeklinde algılanabilir; bundan kaçınmak için sütun ismi veyâ tablo ismi gibi özel isimlerin başına ve sonuna bir belirteç koyabiliriz. Bu belirteç bir nesne isminin başladığını ve bittiğini belirtmek için kullanılır. MySQL için bu belirteç ters ayraç (`) işâretidir. Misal : `articles` gibi..

- `SELECT` deyimi yalnızca tablodan kayıt sorgulamak için değil, bir değeri seçmek için de kullanılır. Misal, `SELECT` ifâdesinden sonra bir fonksiyon çalıştırılsa, fonksiyonun sonucu seçilmiş ve kullanıcıya çıktı olarak verilmiş olur.

SEÇİLİ DEĞERİ İSİMLENDİRME

- `SELECT` ile seçtiğimiz değer bir sütun olmak zorunda değil; seçtiğimiz değer ne olursa olsun, bunun çıktı olarak verilirken farklı bir isimle isimlendirilmesini isteyebiliriz. Buna 'takma isim takma' ('aliasing') denir ve şu biçimde yazılır:

```
SELECT `id` AS `numara`, `name` AS `isim` FROM `articles`
# 'articles' tablosundaki tüm satırların 'id' ve 'name' sütunundaki
# veriler 'numara' ve 'isim' sütun ismiyle getirilir

# Bu sorgu şu şekilde de yazılabilir, ama birincisi daha anlaşılır:
SELECT `id` `numara`, `name` `isim` FROM `articles`
```

KAYITLARI SIRALI ÇEKME

- Seçtiğimiz kayıtları istediğimiz bir sütuna göre artan veyâ azalan sırada çekebiliriz.
- Bunun için `ORDER BY` komutu kullanılır; bu komut tüm ilişkisel veritabanlarında kullanılabilir. Bu komut, seçme ifâdesi tamâmlandıktan sonra yazılmalıdır:

```
SELECT * FROM articles ORDER BY id;
# 'articles' tablosundaki kayıtlar
# 'id' sütununa göre artan sırada getirilir.
```

- Sıralama varsayılan olarak artan sırada yapılır. Bu sıralama tipinin kodu 'ASC'dir. 'ascending' (artan) ifâdesinden gelmektedir. Azalan sıralamanın kodu ise 'DESC'dir ve 'descending' (azalan) ifâdesinin kısa hâlidir. Sıralama çeşidini sıralama yapmak istediğimiz sütun isminden hemen sonra yazarak belirtebiliriz:

```
SELECT * FROM articles ORDER BY id DESC;
# 'articles' tablosundaki kayıtlar
# 'id' sütununa göre azalan sırada getirilir...
```

- Sıralama işlemi birden fazla sütun için uygulanabilir. Bu durumda sıralama evvelâ ilk yazılan sütuna göre yapılır; ardından ilk yapılan sıralama bozulmayacak şekilde diğer sütuna göre sıralama yapılır. Bu, kayıtların sıralama yaptığımız sütunlarında aynı değerin olduğu durumlarda farklılık gösterir. Misal vermek gerekirse, bir müşteri tablosundan müşteri ismine göre sıralama yaptığımızı ve 'Ahmed' isimli beş müşterimizin olduğunu düşünelim. Sıralamayı isim alanına göre yaptığımızda 'Ahmed' isimli beş müşterinin bilgileri art arda gelir; eğer ikinci bir sıralama olarak müşteri numarası alanını seçersek bu durumda

müşteri numarası en küçük olan 'Ahmed' isimli kayıt diğer 'Ahmed' isimli kayıtlardan önce gelir:

```
SELECT * FROM `customers` ORDER BY `name` ASC, `id` DESC;
# 'customers' tablosundaki kayıtlar önce isme göre artan sırada,
# ardından 'id' alanına azalan sırada sıralanır ve getirilir.
```

> ***NOT :*** MySQL'de NULL değerler non-NULL değerlerden önce gelir. Misal, NULL veri içerebilen ve veri tipi sayısal olan bir sütunda NULL değerler 'ASC' (artan) sıralamada sıfır(0) değerinden önce gelir.

- Misal:

```
SELECT *
FROM articles
WHERE page >= 10
ORDER BY page ASC
LIMIT 2;# Sayfa sayısı >= 10 olan en kısa 2 makâle getirilir.
```

SORGU DİLİNİ KULLANMA

Basitçe Sorgu (WHERE İfâdesi)

- Koşullu sorgular veritabanının olmazsa olmazıdır.
- `WHERE` esnek koşullu sorgular yazmak için kullanılan bir anahtar kelîmedir; kendinden evvelki yazılan ifâde için bir şart belirtir.
- `WHERE` ifâdesi `SELECT` ve `FROM` deyimlerinden sonra gelir. Yazım şekli:

```
SELECT * FROM table WHERE condition;
#'condition' yazan yere şart yazılır.
```

- `WHERE` ifâdesinden sonra ilgili sütun ismi zikredilip, bir şart yazılabilir. Tablodaki satırların ilgili sütunu için belirtilen şartı taşıyanlar sorgu sonucu olarak döndürülür:

```
SELECT * FROM `articles`
WHERE `publishDate` > '2020-10-01';
#'articles' tablosundaki satırlar arasından 'publishDate' alanı
#01.10.2020 değerinden büyük olan kayıtlar getirilir.
```

'Eşittir' ve 'Eşit Değildir' İşleci

- Bir sütunun değerinin belirtilen değere eşit olup, olmadığını belirtmek için kullanılır:

```
SELECT * FROM `articles` WHERE `id` = 1;
#'id' sütununun değeri 1 olan kayıt getirilir.
```

- Bu işleç ilgili sütunun veri tipinden bağımsızdır. Misal, metînsel ifâdeler için de kullanılabilir:

```
SELECT * FROM `articles` WHERE `name` = "Soykırıma karşı hamleler";
#'articles' tablosunun 'name' sütununun
#değeri 'Soykırıma karşı hamleler' olan kayıt(lar) getirilir.

SELECT * FROM `articles` WHERE `writer` != "Mehmet Âkif SOLAK";
# Yazarı ('writer') 'Mehmet Âkif SOLAK' olmayan tüm kayıtlar getirilir
```

Büyüktür, Küçüktür, Büyük Eşittir, Küçük Eşittir İşleçleri

- Bu işleçler de anladığımız manâda ilgili sütunun değeri üzerinde şart belirtmek için kullanılır; metînsel veriler için büyüklük - küçüklük ASCII veyâ UNICODE kodunun büyüklüğüne göredir; yanî kabaca alfabetik sıralamaya göredir; fakat bu sıralama alfabetik sıralamanın aynısı değildir; karakterin karakter kodlama sistemindeki sayısal büyüklüğünün sıralamasına göredir. Misal, 'a' harfi 'A' harfinden büyüktür.

Birden Fazla Şart Belirtme

- Programlama dillerinde olduğu gibi SQL'de de birden fazla şart belirtme işlemi vardır. Bunun için `AND` (ve) ve `OR` (veyâ) işleçleri kullanılır.

- Bilindiği üzere iki şartın birden sağlanması bekleniyorsa ve 've' (`AND`) mantıksal işleci, iki şarttan herhangi birinin sağlanmasının kâfî olduğu durumlarda ise 'veyâ' (`OR`) mantıksal işleci kullanılır:

```
SELECT * FROM `articles`
WHERE page = 11 AND `publishDate` = '2024-06-31';
#'articles' tablosundaki sayfa (page) sayısı 11 olan ve
#yayım târihi ('publishDate') 31.06.2024 olan kayıt(lar) getirilir.

SELECT * FROM `articles` WHERE page = 11 OR page = 12;
#11 veyâ 12 sayfa uzunluğunda olan makâleler getirilir
```

> ***NOT :*** '=', '<', '>' gibi mantıksal işleçler sonuç olarak 1, 0 veyâ NULL döndürür; 1 döndürüyorsa koşul cümlesi doğru demektir, yanî şart sağlanıyor demektir. 0 döndürüyorsa koşul cümlesi yanlış demektir; NULL döndürüyorsa koşulun sınanmasına engel olan bir NULL değeri var demektir:
>
> ```
> SELECT 1 AND 0;# 0 döndürülür
> SELECT 1 AND 1;# 1 döndürülür
> SELECT 1 AND NULL;# NULL döndürülür
> ```

- Bunun yanında aynı şart için birden fazla kabûl edilen eşitlik değeri olması isteniyorsa `IN` ve `NOT IN` işleçleri kullanılabilir:

```
SELECT writer, name, page
FROM articles
WHERE writer IN ("Mehmet Âkif SOLAK", "Darren BYLER");
# Mehmet Âkif SOLAK ve Darren BYLER tarafından yazılmış
# makelelerin yazar, isim ve sayfa bilgisi döndürülür.
#
SELECT name FROM articles WHERE writer NOT IN ("Mehmet Âkif SOLAK");
# Mehmet Âkif SOLAK tarafından yazılmayan
# makâlelerin ismi döndürülür.
```

'Arasında' Şartı

- Bir değerin verilen iki değerin arasında olup, olmadığını kontrol etmek için `BETWEEN` işleci kullanılır:

```
SELECT * FROM `articles` WHERE `page` BETWEEN 7 AND 27;
# Sayfa sayısı ('page') 1 ile 3 arasında olan makâleler getirilir.
# 7 ve 27 dâhildir.
```

Koşulu Sağlayan Referans Değerleri Liste Olarak Belirtme

- Koşul belirtirken hem sorgu metninin uzunluğunu kısaltmak, hem de sorgu performansını arttırmak için koşulu sağlayan referans değerleri liste olarak belirtebiliriz.
- Bunu aynı sütun için birden fazla şart yazdığımız durumlarda kullanabiliriz:

```
SELECT * FROM `articles`WHERE `writer` IN (
"Mehmet Âkif SOLAK",
"Ömer Fâruk UZUNKAYA");
# Yukarıdaki kod, makâle yazarı ('writer') 'Mehmet Âkif SOLAK'
# veyâ 'Ömer Fâruk UZUNKAYA' olan makâleleri getirir.
# Bu kodun uzun hâli şöyledir:
SELECT * FROM `articles` WHERE `writer` = 'Mehmet Âkif SOLAK'
OR `writer` = 'Ömer Fâruk UZUNKAYA';
```

- Bunun tersi olan durumu, yanî müşteri ismi belirtilen 'listede bulunmayan satırları getir' sorgusunu yazmak için `IN` işlecinin önüne `NOT` işleci getirilir.

Koşul Belirtirken Örüntü Kullanmak

- Yazdığımız koşul bâzen örüntü olabilir. Misal, ismi 'Ahmed' olan tüm müşterilerimizin bilgilerini getirmek istiyor olalım; bunu eşitlik işleciyle yazdığımızda 'Ahmed Fâruk' ismindeki müşterinin bilgileri gelmez. Çünkü eşitlik işleci iki değer bire bir aynı ise 1 ('TRUE') değeri üretir.
- Bu tür durumlarda örüntülerden ve dolayısıyla düzenli ifâdeleri kullanmalıyız.
- Öncelikle ilgili şart ifâdesinde sütun isminden sonra, ilgili sütun için düzenli ifâde kullanacağımızı belirten `LIKE` işlecini kullanmalıyız.
- Ardından belirttiğimiz değer içerisinde özel anlamı olan 'joker karakter'ler ('wildcards') kullanmalıyız. Bâzıları şunlardır:
    1. `%` : 0 veyâ daha fazla karakter ile eşleşir.
    2. `_` : Sadece bir karakter ile eşleşir.
- Yukarıdaki misal için yazmamız gereken sorgu şöyledir:

```
SELECT * FROM customers WHERE name LIKE '%Ahmed%';
# Yukarıdaki kod 'name' (isim) alanı içerisinde
# 'Ahmed' ifâdesini barındıran tüm kayıtlar getirilir..
```

- Kod içerisinde 'Ahmed' isminden hem önce, hem de sonra '%' işleci kullandığımızdan ismi içerisinde 'Ahmed' değeri geçen tüm kayıtlar getirilir. Eğer 'Ahmed' isminden yalnızca sonra '%' işâreti koysaydık ilk ismi 'Ahmed' olan kayıtlar getirilirdi.

> ***NOT :*** `LIKE` işleci mâliyyetli bir işleçtir. Performansı düşürmemek için bu işleci sadece gerektiğinde kullanmak önemlidir.

Değer Olmadığı Durumu (NULL) ve Tersini (NOT NULL) Kontrol Etme

- Bir tablo içerisindeki belirtilen sütunu NULL olan satırları almak istediğimizde eşittir (=) işlecini kullamayız. Bu durumda `IS` işleci kullanılır; `NULL` ise olduğu gibi yazılır:

```
SELECT * FROM `articles` WHERE `writer` IS NULL;
# Yazar ('writer') belirtilmeyen kayıtlar getirilir.
```

- Eğer bir sütun için değeri belirtildiği koşulunu yazmak istiyorsak mantıksal ifâdeyi ters çeviren `NOT` işlecini kullanmalıyız:

```
SELECT * FROM `articles` WHERE `writer` IS NOT NULL;
# Yazarı belirtilen kayıtlar getirilir.
```

> ***NOT :*** NULL değerinin boş metîn veyâ sıfır (0) anlamına gelmediğini unutmayınız. Birinci misaldeki yazar ismi boş metîn ('') olarak verildiğinde NULL kayıtlar arasında getirilmez

> ***NOT :*** Bu mantıksal işleç her yerde kullanılamaz. `NULL` ifâdesinden ve `IN` ifâdesinden önce yaygın olarak kullanılır.

Münferid (Tekil) Değerleri Seçme

- Eğer aksi yönde bir kısıt eklenmediyse bir tabloda bir sütun içerisinde aynı değerden birden fazla olabilir. Bu sütunda kaç farklı değer olduğunu ve bunların ne olduğunu öğrenmek için `SELECT DISTINCT` komutu kullanılır:

```
SELECT DISTINCT `writer` FROM `articles`;
# `articles` tablosu içerisinde en az bir makâlesi olan yazarların
# ismi gelir. Birden fazla makâlesi olan yazarların ismi bir kez gelir
```

- **! DISTINCT komutu verilen sütunların hepsi için münferid (tekil) olma durumunu kontrol eder.** Dolayısıyla 'writer' sütununun yanına 'page' sütununu da eklersek aynı yazarın farklı sayfa sayısında olan makâleleri getilir; yanî bir yazar için birden fazla kayıt getirilebilir.

TABLO BİRLEŞTİRME - KAYNAŞTIRMA (JOIN)

- İlişkisel veritabanının en önemli özelliklerinden birisi tabloları birleştirerek sorgulama yapabilmektir. Bunun için birleştirilecek tabloların nasıl birleştirileceğini belirtmemiz gerekir. Bu birleştirme işlemine 'join' denir.

- Tablo birleştirme kod söz dizimi şu şekildedir:

```
SELECT [seciliSutunlar] FROM tablo_1
JOIN tablo_2 ON tablo_1.ortakSutun = tablo_2.ortakSutun;
```

- **! JOIN sözdizimi WHERE ifâdesinden önce olmalıdır.**

- JOIN komutunu kullanırken hangi tablonun hangi sütununun hangi tablonun hangi sütunuyla eşleşeceğini belirtmek gerekiyor. **Burada, `tablo.sutunIsmi` söz dizimi uygulanır.**

- Misal, "Soykırıma Karşı Hamleler" makâlesinin yazarını ve yazarın telefon numarasını öğrenmek için şu sorguyu yazabiliriz:

```
SELECT writer, phoneNumber
FROM `articles`
JOIN `writers` ON `articles`.`writer` = `writers`.`name`
WHERE `articles`.`name` = "Soykırıma Karşı Hamleler";
```

- Burada ilgili ismini de getirmek istediğimizde sizce seçilen sütun isimleri arasına 'name' değerini eklemek kâfî midir, beklenen sonuç nedir?

- Yukarıdaki sonucun doğru cevâbı : kâfi değildir, beklenen sonuç hatâdır. Oluşan hatâ ise şudur: Error Code: 1052. Column 'name' in field list is ambiguous MySQL veritabanı sistemi ilgili sütun isminin belirsiz olduğu şeklinde bir hatâ fırlattı. Bu hatânın sebebi her iki tabloda da 'name' isminde sütun olmasıdır. Bunun yerine hangi tablonun hangi sütununu seçtiğimizi belirtmeliydik.

- Tablo birleştirme işlemi uygulanan sorgularda isimlendirme kullanmak

    1. sorguyu kısaltır,
    2. anlaşılırlığı arttırır,
    3. sorgu yazım süresini kısaltır,
    4. yukarıdaki gibi hatâların önüne daha kolay geçebilmeyi sağlar.

- Yukarıdaki sorgunun kısa yazımı şu şekildedir:

```
SELECT a.name, a.writer, w.phoneNumber
FROM `articles` AS a
JOIN `writers` AS w ON a.`writer` = w.`name`
WHERE a.`name` = "Soykırıma Karşı Hamleler";
```

> ***NOT :* İsimlendirme kullanıldığında sorguda tablonun gerçek ismi kullanılamaz!**

> ***NOT :*** Tablo birleştirme işlemi yapılırken tabloları hangi sütun üzerinden birleştirmek istediğimiz `tablo1.sutunIsmi ON tablo2.sutunIsmi` biçimindeki kodla yapılıyor. Eğer birleştirilmek istenen tablolardaki sütunların isimleri aynı ise bunun yerine `USING (sutunIsmi)` şeklindeki kod kullanılabilir.

Diğer Birleştirme Çeşitleri

- Yukarıda gösterilen örnekteki birleştirme çeşidinin ismi 'inner join'dir ve varsayılan birleştirme çeşidir budur. Bunların dışında başka birleştirme çeşitleri de vardır. Bunlar, ilgili tabloda ilgili sütunların NULL veri içerdiği durumlarda birleştirmenin nasıl yapılacağını belirtmek için kullanılır.
- Birleştirme çeşitlerine giriş yapmadan evvel şunu anlamalıyız. Sorguda `FROM` ifâdesiyle belirtilen tablo sol tablodur; `JOIN` ifâdesinden sonra yazılan tablo ise sağ tablodur.
- `INNER JOIN` işleminde SAĞ ve SOL tablonun kesişimi alınır; yanî SAĞ tablonun anahtarına karşılık SOL tablodaki dış anahtar sütununda aynı değer yoksa, o satır alınmaz. Misal, "Soykırıma Karşı Hamleler" makâlesinin yazarının bilgileri 'writers' tablosunda olmasaydı sorgudan geriye hiçbir veri dönmezdi.
- `LEFT JOIN` işleminde SOL tablodaki kayıtların tümü getirilir; SAĞ tabloda ilgili kayda denk düşen kayıtlar getirilir. Yanî tabloda yazarı yazarlar ('writers') tablosuna yazılmamış değerler de getirilir.
- `RIGHT JOIN` işleminde ise, yukarıda anlatılan işlemin tam teri uygulanır.
- `FULL JOIN` işleminde ise her iki işlem de uygulanır!

Farklı Bir Birleştirme Çeşidi : ÇAPRAZ BİRLEŞTİRME (CROSS JOIN)

- `CROSS JOIN` tabloların çaprazlama çarpımını almak gibidir. Sol tablodaki her satır, sağ tablodaki her satırla birleştirilerek yeni bir satır olarak sonuç kümesine eklenir ve bu sonuç kümesi tablo olarak döndürülür.
- Yanî çapraz birleştirme ('cross join') işlemiyle gelen verilerin bâzıları doğru olmayabilir. Misal, 'articles' tablosu ile 'writers tablosu' çaprazlamaya sokulduğunda her makâle

bilgisinin yanına her yazar bilgisi yeni bir satır olarak gelir:

```
SELECT * FROM articles CROSS JOIN writers;
# 2 makâle ve 2 yazar varsa, geriye 4 satırlık sonuç döner.
```

Başka Acâyip Bir Tablo Birleştirme Çeşidi : 'Self Join'

- Sözdizimi normal tablo birleştirme gibidir; fakat aynı tabloya ikinci kez referans verildiğinde takma isim kullanmak gerekir.
- Karışmaması açısından ilk seferinde tabloya farklı bir takma isim verip, ikinci sefer farklı bir takma isim vermek kullanışlı olabilir.

Birden Fazla Tablo Birleştirme

- Birden fazla tablo birleştirme işlemi için sözdizimi:

```
SELECT * FROM articles AS a
JOIN writers AS w ON a.writer = w.name
JOIN publisher AS p ON a.publisher = p.name;
```

- Birden fazla tablo birleştirmede ikinci tablodaki bir sütun, üçüncü tablodaki bir sütun ile bağlanarak tablo birleştirilebilir. Misal, müşteriler tablosundan müşteri ismini, adresler tablosundan müşterinin numarasıyla kayıtlı olan adresin il bilgisini, ilçeler tablosundan ise o ilin ilçelerini göstermek istersek, birinci tablo ile ikinci tablo müşteri numarası üzerinden, ikinci tablo ile üçüncü tablo ise il kodu veyâ il ismi gibi ili belirten sütun üzerinden yapılır.

> ***NOT :*** `LEFT JOIN` komutu yerine `LEFT OUTER JOIN`, `RIGHT JOIN` komutu yerine `RIGHT OUTER JOIN` komutu kullanılabilir. 'OUTER' ifâdesinin geçerli olması kuvvetli muhtemel veritabanları arasındaki uyumu sağlamak içindir.

KAYITLARI GRUPLANDIRARAK SORGULAMA

- Ticâret verilerinizin olduğu bir veritabanınızın olduğunu düşünün; satış verilerinizi bir tabloda tutuyorsunuz ve her satır bir satış işleminin bilgisini içeriyor. Satış işleminin yapıldığı saat bilgisi de burada yer alıyor. Siz, hangi saatlerde daha çok satış yapıldığını hesâplamak istiyorsunuz, işte `GROUP BY` komutu bu işe yarıyor.
- Veyâ satışlarınızı illere göre gruplandırarak hangi ilde daha çok satış yaptığınızı görmek istiyorsunuz; işte `GROUP BY` komutu bunun için kullanılıyor.

- Gruplama işlemi `GROUP BY` deyimiyle yapılır ve bu deyimden sonra gruplama işlemi hangi sütun üzerinden yapılmak isteniyorsa bu sütunun ismi yazılır:

```
SELECT COUNT(*) FROM articles GROUP BY writer;
# Her yazar için kaç makâlenin olduğu bilgisi döndürülür.
# COUNT() fonksiyonu bir AGGREGATE fonksiyonudur ve gelen
# sonuçta verilen sütunu dolu olan kaç satır olduğu bilgisini döndürür
# Boş bir değer olan bir sütun olabilir; bunun için '*' yazabiliriz.
```

- **! `GROUP BY` deyimi hem `ORDER BY` deyiminden ÖNCE, `WHERE` deyiminden SONRA gelir.**

- Gruplama işlemleri kullanma amacımız ya satır toplamını bulmak, ya en düşük değeri bulmak, yâ en yüksek değeri bulmak, yâ dâ buna benzer bir işlemin sonucunu öğrenmek içindir. Bunun için `GROUP BY` deyimi 'Aggregate functions' ismiyle yâd edilen fonksiyonlarla berâber kullanılır.

'Aggregate' (Toplam, Tüm Satırlar Üzerine Uygulanan) Fonksiyonlar

- Bu fonksiyonlar tüm satırlar üzerine uygulanarak sonuç elde edilmek istendiği durumlarda kullanılır. Tüm satırlar içinden en yüksek değerli olanı, en düşük değerli olanı, satır sayısı, satırların ilgili sütundaki toplamları gibi değerleri elde etmek için bu fonksiyonlar kullanılır.

- Bu fonksiyonlar genellikle gruplama işlemlerinde kullanılır; fakat bu fonksiyonların kullanılması için gruplama işlemini yapmaya gerek yoktur.

Temel Toplam fonksiyonları:

1. `SUM(sutun1)` : Çekilen sonuç tablosundaki tüm 'sutun1' sütunu değerlerini toplar.
2. `COUNT(sutun1)` : sutun1 sütunundaki dolu satır sayısını döndürür.
3. `MIN(sutun1)` : 'sutun1' sütunundaki en düşük (asğarî) değeri döndürür.
4. `MAX(sutun1)` : 'sutun1' sütunundaki en yüksek (azamî) değeri döndürür.
5. `AVG(sutun1)` : 'sutun1' sütunundakilerin ortalama değerini döndürür.

Örnekler:

```
SELECT COUNT(*) AS makâleSayısı, SUM(page) AS toplamYazdığıSayfaSayısı
FROM articles
WHERE writer = "Mehmet Âkif SOLAK"
GROUP BY writer;
# Yazarı 'Mehmet Âkif SOLAK' olan toplam makâle sayısını
# 'makâleSayısı' sütun ismi altında, yazarın makâlelerin sayfa toplamını
# 'toplamYazdığıSayfaSayısı' sütunu altında döndürür.
```

```
SELECT SUM(page) FROM articles;# Tüm makâlelerin toplam uzunluğunu verir
#
SELECT writer, SUM(page) FROM articles GROUP BY writer;
# Toplam makâle sayfa sayısını yazara göre gruplandırarak verir.
# Başka bir deyişle, her yazarın kaç sayfa makâle yazdığı bilgisini verir
```

> ***NOT :*** Toplam fonksiyonları `ORDER BY`, `GROUP BY` ifâdeleriyle veyâ `SELECT` ifâdesiyle kullanılabilir; fakat `WHERE` şartıyla kullanılamaz!

Kayıtları Gruplandırmanın Çetrefilli Yanı

- Kayıtları gruplandırırken dikkat etmemiz gereken şey seçtiğimiz sütunların ya bir toplam ('aggregate') fonksiyonu olması veyâ `GROUP BY` deyimiyle gruplandırılmasıdır.
- Bunu daha iyi anlamak için bir örnek üzerinden gidelim, inşâAllâh.. Bir yazarın en kısa yazısının kaç sayfa olduğunu yazarın ismiyle berâber döndüren kodumuz çok güzel duruyor. Peki, burada yazarın en kısa makâlesinin ismini de almak isteseydik ne yapmamız gerekirdi?
- Aklınıza gelen ilk çözüm `SELECT` deyimiyle seçtiğimiz sütunlara 'name' sütununu da eklemek olabilir; fakat böyle yaptığınızda MySQL'de yanlış sonuç alırsınız (tevâfuk olarak doğru sonuç rast gelmediyse), MsSQL'de ise kodunuz hiç çalışmaz. Yanî şu kod aradığımız sonuç değildir:

  ```
  SELECT name, writer, SUM(page) FROM articles GROUP BY writer;
  # MySQL'de yanlış sonuç verir; MsSQL'de ise kod hiç çalışmaz!
  ```

- Kodu çalıştırdığınızda size ilgili yazarın tabloya kaydedilen ilk makâlesinin ismi getirilir; en kısa olanın değil! Çünkü yaptığımız gruplama işleminde 'name' sütununu kullanmıyoruz, o sütunu yaptığımız gruplama içerisine almadığımız için yaptığımız gruplamanın sonucunda gelen sütun bilgisinin âit olduğu satırdaki diğer bilgiye erişemiyoruz. Diğer sütunlarda bir sorun çıkmadı, çünkü o sütunlardan birisi toplam fonksiyonu içeriyordu, diğeriyse `GROUP BY` deyimi içerisinde yer alıyordu.

- Eğer yazarın en kısa makâlesinin ismini de istiyorsak şunu çalıştırıp, gelen sonucu incelemeliyiz (yanî bu da tam istediğimizi vermiyor):

```
SELECT name, writer, MIN(page) AS enKısaMakâle
FROM articles
GROUP BY writer, name
ORDER BY MIN(page), writer;
```

Gruplandırmada Şart Belirtme : 'HAVING'

- `WHERE` deyimi yalın sütunlar üzerinde veyâ sütundan bağımsız şartlar belirtmek için kullanılır.
- `GROUP BY` ile tablonun ham hâlinde olmayan bir sütun oluşturduğumuz için gruplandırdığımız gruplar üzerinde `WHERE` deyimiyle şart belirtemeyiz!
- Benzer şekilde `WHERE` deyimi içerisinde gruplandırma işlemi yapan toplam fonksiyonlarını kullanamayız.
- Bu eksiklikler için önerilen çözüm `HAVING` ifâdesidir; `HAVING` ifâdesiyle gruplar üzerinde şart belirtebilir, toplam fonksiyonlarını kullanabiliriz.
- Çünkü HAVING deyimi GROUP BY deyiminden sonra çalıştırılır; WHERE deyimi ise HAVING deyiminden önce çalıştırılır. **Deyimlerin çalıştırılma sıraları, sözdizimindeki yazım sıralarıyla aynı değildir!** Deyimlerin çalıştırılma sıraları şöyledir:

  FROM -> WHERE -> GROUP BY -> HAVING -> SELECT -> DISTINCT -> ORDER BY -> LIMIT

- Örnekler:

```
SELECT writer AS yazar,
SUM(page) AS sayfaSayısı,
COUNT(*) AS makâleSayısı
FROM articles
GROUP BY writer
HAVING SUM(page) >= 50
ORDER BY SUM(page) DESC;
# Toplam yazdığı sayfa sayısı >= 50 olan yazarların ismi,
# kaç sayfa makâle yazdıkları ve kaç adet makâle
# yazdıkları bilgisi en çok makâle yazan önce olacak şekilde
# sıralanmış olarak döndürülür.
```

```
SELECT a.writer,
SUM(a.page) AS toplamSayfaSayısı
FROM articles AS a
INNER JOIN writers AS w ON a.writer = w.name
GROUP BY writer
HAVING SUM(a.page) > 50 AND w.phoneNumber LIKE "123%"
ORDER BY SUM(a.page) DESC;
# Telefon numarası '123' ile başlayan ve
# yazdığı toplam sayfa sayısı 50'den fazla olan yazarın
# seçili bilgilerini sayfa sayısı
# en fazla olan yazar önce olacak şekilde getir
```

> ***NOT :*** **`HAVING` deyimiyle kullanmak istediğimiz sütunları `GROUP BY` deyimiyle seçmiş olmalıyız! ! Eğer bir sütunu yalın olarak seçmeyeceksek, toplam fonksiyonuyla berâber kullanacaksak `GROUP BY` deyiminde seçmemize gerek yok**

- ..

Gruptakileri sayma : 'HAVING' içinde 'COUNT' kullanımı

- Gruptaki satır sayısına göre filtreleme uygulamak istediğimizde kullanılabilir.
- Kullanımı basittir; yeni bir sözdizimi yoktur; `HAVING` deyimi içerisinde `COUNT()` fonksiyonunu kullanıyoruz:

```
SELECT writer, COUNT(*) AS makâleSayısı
FROM articles
GROUP BY writer
HAVING COUNT(*) > 2
ORDER BY makâleSayısı;
# Makâleleri yazara göre gruplandırır;
# 2'den fazla makâlesi olan yazarın ismini
# ve toplam makâle sayısını gösterir
```

Birden Fazla Gruplama (Super Aggregate) : UNION ALL

- `UNION ALL` ifâdesi birden fazla gruplama işleminin birleştirilmesine yönelik bir işlemdir.
- Misal, her yazarın kaç sayfa makâle yazdığı bilgisiyle berâber tüm makâlelerin toplam sayfa uzunluğunu da öğrenmek istiyorsak şu kodu çalıştırabiliriz:

```
SELECT writer, SUM(page)
FROM articles
GROUP BY writer
#ORDER BY SUM(page) --> BU SATIRI AÇARSAK NE OLUR?
UNION ALL# Buradan sonra ikinci sorgu yazılır:
SELECT NULL, SUM(page) FROM articles;
# İkinci sorguyla gelen sütun sayısı
# birinci sorguyla gelen sütun sayısıyla
# aynı olmak zorundadır; bu sebeple 'NULL' yazılmıştır.
# Gelen sonuç şöyledir:
# writer            SUM(page)
# Mehmet Âkif SOLAK    53
# Darren BYLER        310
# NULL                363 --> Bu satıra 'super aggregate' satırı denir.
```

- `UNION ALL` deyimini kullanmanın menfî (olumsuz) bir yönü `ORDER BY` deyimini kullanamıyor olmamızdır. Yukarıdaki sorgu içerisinde yoruma alınmış olan `ORDER BY` deyiminin bulunduğu satırı açarsak kodumuz çalışmaz. Bu sorunun çözümü için alt sorgu ('subquery') tekniği kullanılabilir:

```
SELECT *
FROM
(
    SELECT writer, SUM(page) AS toplam
    FROM articles
    GROUP BY writer
    UNION ALL
    SELECT NULL, SUM(page) FROM articles
) AS A
ORDER BY toplam DESC;
# Alt sorgu tekniği için alt sorgu bölümüne bakınız.
```

- `UNION ALL` deyiminin menfî yönleri

    1. sorgunun uzun olması,
    2. veritabanı motorunun iki sorguyu ayrı ayrı çalıştırması gerektiğinden mâliyetinin (okunan 'page' sayısı, hız) fazla olması
    3. `ORDER BY` deyiminin yalın olarak kullanılamamasıdır.

Gruplama Üzerinde Gruplama : ' WITH ROLLUP'

- `UNION ALL` deyimiyle aynı işleve sâhip `ROLLUP` deyimi, `UNION ALL` deyiminin üç menfî yönünü de kapatmaktadır. Aşağıdaki kod, bir önceki örnekteki kod ile aynı sonucu döndürür:

```
SELECT writer, SUM(page)
FROM articles
GROUP BY writer WITH ROLLUP;
```

- ! Ayrıca `WITH ROLLUP` ifâdesiyle `ORDER BY` deyimi yalın olarak kullanılabilir
- Birden fazla grup seçildiyse `WITH ROLLUP` deyimi **GROUP BY ile belirtilen sırada alt gruplama yapar.**
- Misal, mağaza bazında satış toplamlarımızı ve ay bazında satış toplamlarımızı `WITH ROLLUP` deyimiyle berâber çağırdığımızda her mağaza için ay sütun değerlerinin bittiği yerde mağazanın tüm senedeki satış toplamı görünür; tüm mağazaların sonunda da tüm mağazaların tüm senelik satış toplamı görünür; tabi bu satış toplamının olduğu satırlarda, boş olan sütunlar 'NULL' değeriyle doldurulur; `UNION ALL` ifâdesinde bunu elle yapmamız gerekiyordu, fakat burada gerekmiyor. Aynı sorguda `GROUP BY` deyimindeki mağaza ve ay sütunlarının yerini yanî gruplama sırasını değiştirirsek, her ay için tüm mağazaların ayrı satır olarak satış toplamlarıyla görünür; ay sonunda tüm mağazaların o aydaki satış toplamı görünür; tüm ayların sonunda da tüm ayların tüm mağazalar için olan satış toplamı tek satır olarak görünür.

'Grouping' fonksiyonu

- Grup içinde gruplama yaptığımızda ikinci gruplama değerinin olduğu satır veriler arasında 'NULL' olarak görünüyordu. Buna çözüm olarak `GROUPING` fonksiyonu kullanılır.
- `GROUPING` fonksiyonu ilgili satırın grup içinde gruplama yapıldığı satır olup, olmadığı bilgisini BIT tipinde (1 veyâ 0) bir veri olarak döndürür. Bu yeni bir sütun olarak yazılabilir:

```
SELECT writer, SUM(page), GROUPING(writer)
FROM articles
GROUP BY writer WITH ROLLUP;
#Şu sonucu döndürür:
#writer             SUM(page)   GROUPING(writer)
#Darren BYLER        310            0
#Mehmet Âkif SOLAK    53            0
#NULL                 363            1
```

- Bunu bu şekilde yapmak yerine `IF` fonksiyonuyla (`IF` deyimiyle değil) kullanarak daha iyi bir çözüm elde edebiliriz:

```
SELECT IF(GROUPING(writer), 'Toplam', writer) as yazar,
SUM(page)
FROM articles
GROUP BY writer WITH ROLLUP;
# yazar                 SUM(page)
# Darren BYLER            53
# Mehmet Âkif SOLAK        310
# Toplam                363
```

- GROUPING fonksiyonu `SELECT` deyimi bünyesinde kullanılabilir. MySQL 8.0.12'den itibâren `ORDER BY` deyimi bünyesinde de kullanılabilir.

ALT SORGU (İÇ İÇE SORGU) (SUBQUERY, INNER QUERY)

- Sorgu içerisinde sorgu yazılması işlemidir. Bu işlem bâzı kere istenilen sorgunun çekilebilmesi için tek yöntemdir.
- İçteki sorguya 'iç sorgu' ('inner query') denir; dıştaki sorguya 'dış sorgu' ('outer query') denir.
- İç sorgu kullanılırken normal parantezler kullanılır; ve bu iki parantez arasına ilgili sorgu yazılır:

```
SELECT name as n, phoneNumber,
(SELECT sum(page) FROM articles WHERE writer=n) as toplamSayfa
FROM writers;
# Her yazarın ismi, telefon numarası ve yazdığı makâlelerin
# toplam sayfa sayısı döndürülür.
# İç sorgu içerisinde dış sorgudaki değişkenin kullanılabildiğine
# dikkat ediniz.

# Fakat yazılan sorgunun MsSQL için de
# çalışması isteniyorsa şu şekilde yazılmalı:
SELECT name, phoneNumber,
(SELECT sum(page) FROM articles WHERE writer=w.name) as toplamSayfa
FROM writers as w;
```

- İç sorgu bâzı yerlerde tablo birleştirme ('join') gibibir işlev sağlamaktadır, bu durumlarda 'join' kullanmak daha doğru olur; çünkü iç sorgu sorgunun uzun olması demektir ve her iç sorgu ayrı ayrı işlendiğinden iç sorgunun tablo birleştirme işleminden daha düşüktür. Yukarıdaki işlem de böyledir.

> ***NOT :*** İç sorgu içerisinde dış sorgudaki SELECT değişkeni kullanılabilir.

- Yukarıdaki kod iç sorgunun `SELECT` içerisinde kullanımına örnek idi. İç sorgu `WHERE` deyimiyle berâber de kullanılabilir:

```
SELECT writer, name, page
FROM articles AS a
WHERE page = (SELECT MIN(page) FROM articles WHERE writer = a.writer)
ORDER BY page DESC;
# Her yazarın yazdığı en kısa makâlenin bilgileri getirilir.
# Bu sorgu daha evvel yazmak istediğimiz sorgudur.
```

- WHERE deyimiyle belirtilen iç sorgunun sonucunda birden fazla değer gel(ebil)iyorsa, `IN` ve `NOT IN` işleçleri kullanılabilir:

```
SELECT writer, name, page
FROM articles AS a
WHERE writer NOT IN (
    SELECT name FROM writers WHERE phoneNumber LIKE '123%'
)
ORDER BY page DESC;
# Telefon numarası '123' ile başlamayan yazarların
# makâlelerinin yazar, isim ve sayfa bilgileri döndürülür.
```

- İç sorgu dış sorgudan bağımsız olduğunda iç sorgu ayrıca çalıştırılır; yukarıdaki kodda iç sorgu dış sorgudan bağımsızdır.

- Eğer iç sorgu içerisinde dış sorguda seçtiğimiz bir sütunu, değeri kullanırsak, bu değer her satır için değişeceğinden iç sorgu sorgunun her satırında o satırdaki değere özel olarak çalıştırılır; bu şekilde ki iç sorgulara 'ilişkili alt sorgu' ('correlated subquery') denir:

```
SELECT name, writer, page
FROM articles a
WHERE page > (
    SELECT AVG(page) FROM articles
    WHERE writer = a.writer GROUP BY writer
);
# Yazarın kendi makâle sayfa ortalamasının
# üzerinde olan makâlelerini döndürür.
```

- Alt sorgu ile bir sorgunun var olup, olmadığı sorgulanıp; buna göre ilgili satırdaki bilgilerin getirilmesi sağlanabilir:

```
SELECT name, writer FROM articles
WHERE EXISTS(
    SELECT writer FROM writers
    WHERE phoneNumber LIKE '123%'
);
# Telefon numarası '123' ile başlayan
# yazarın ismi ve makâlelerinin ismi döndürülür.
# Bunun yerine tablo birleştirme ('join') kullanılmalıydı;
# bu sorgu sadece örnek olması için gösteriliyor.
```

- `EXISTS` fonksiyonu dönen değer olduğunda TRUE değerini üretir; eğer dönen değer olmadığında TRUE değerinin üretilmesi isteniyorsa fonksiyonun önüne `NOT` eklenir : `NOT EXISTS(...)`
- Alt sorgu `FROM` deyimiyle de berâber kullanılabilir; bu durumda alt sorgudan gelen sonuç bir geçici tabloda saklanır.

GEÇİCİ TABLO ('TEMPORARY TABLE') ve TÜRETİLMİŞ TABLO ('DERIVED TABLE')

- Geçici tablolar oturum boyunca geçerli olan, **oturum sonlandığında veyâ bağlantı koptuğunda otomatik olarak silinen** ve yalnızca oluşturulduğu oturum üzerinden erişilebilen bir tablo çeşidîdir.
- Geçici tablolar gelen sorgu sonuçlarını tutmak, henüz tamâmen bitmemiş işlemlerin bitmiş kısımlarını tutmak, verileri önce çekip, sonra gerçek tabloya aktarmak, oturum boyunca sıkça erişilen verileri bir yerde tutarak, performans ve zamân kazanmak gibi amaçlar için kullanılırlar.
- Geçici tablolar gerçek tablo olmadığı için her oturumda aynı veritabanı üzerinde aynı isimde bir geçici tablo oluşturulmasına bir mânî yoktur.
- Geçici tablo oluşturmak için `CREATE TEMPORARY TABLE geciciTabloIsmi` biçimindeki kod kullanılır. Bu, MySQL'de böyledir; misal MsSQL'de geçici tablo oluştururken yalnızca tablo isminin başına `#` işâreti eklenir:

```
CREATE TEMPORARY TABLE geciciTablo;
```

- Bİr veritabanı üzerinde çalışırken veritabanında var olan bir tablo ile aynı isimde bir geçici tablo oluşturulduğunda artık o isimle geçici tabloya erişilir; geçici tablo silinmeden o oturumda gerçek tabloya ismiyle erişilemez.

- **!** Geçici tablolarla çalışırken dikkat edilmesi gereken husus geçici tabloların veritabanındaki gerçek tablolarla aynı isimde olması durumunda, bağlantı kopukluğu gibi durumlardan sonra geçici tablo üzerinde yapılan işlemlerin gerçek tablo üzerinde yapıldığının gözden kaçabilen bir durum olmasıdır.

- Geçici tabloların gerçek tablolarla aynı isimde olması yukarıdaki gibi sıkıntılı durumlar oluşturabilir. Benzer bir durum geçici tablo silinirken yanlışlıkla gerçek tablonun silinmesi işlemidir. Bu sorundan kaçınmak için geçici tabloyu silerken `DROP TABLE` ifâdesi yerine `DROP TEMPORARY TABLE` ifâdesini kullanılmalıdır. Bu durumda ilgili isimde bir geçici tablo yoksa, gerçek tablo silinmez ve hatâ alınır.

> ***NOT :*** Veritabanıyla iletişim kuran bir uygulama geliştirirken uygulamadaki bir istemcinin bağlantısının sonlanmasıyla geçici tablo silinmemiş olabilir; uygulamalar, daha sonraki istekler için bâzı istekleri bir havuzda açık olarak bıraktıklarında böyle sorunlar ortaya çıkabilir. Bu durumda diğer oturumdaki verilerle bu istemcinin verileri karışabilir, güvenlik sıkıntıları ve daha pek çok sıkıntılar ortaya çıkabilir; bu **sorundan kaçınmak için bağlantıyı sonlandırmadan evvel geçici tabloyu komut göndererek siiliniz**.

- MySQL içerisinde geçici tablo olup, olmadığını sorgulamak için doğrudan bir fonksiyon sağlanmamaktadır; fakat bunun yerine bir 'saklı işlem yığını', 'saklı yordam' yazılabilir:

```
DELIMITER $$
CREATE PROCEDURE temporary_table_exists(tableName VARCHAR(100))
BEGIN
    #...
END$$
DELIMITER ;
```

Türetilmiş Tablo ('Derived Table')

- Türetilmiş tablolar bir çeşit geçici tablolardır; fakat bu tablolar kullanıcılar tarafından doğrudan oluşturulmaz.

- Bir sorgu içerisinde `FROM` deyimi içerisinde iç sorgu ('subquery') yazıldığında bu sorgudan dönen değerler için bir türetilmiş tablo oluşturulur:

```
SELECT a.altSorguSutun1, a.altSorguSutun2
FROM (
    SELECT altSorguSutun1, altSorguSutun2 FROM tabloAdi
) AS a
WHERE a.altSorguSutun1 IS NOT NULL;
```

- İstenildiği takdirde alt sorguda seçilen sütunlara takma isim verilebilir ve bu takma isim ana sorguda kullanılabilir.
- **İç sorgu `FROM` ifâdesiyle kullanıldığında oluşan türetilmiş tabloya bir isim verilmesi zorunludur!**

TÜRETİLMİŞ SÜTUN ('GENERATED COLUMN')

- Bâzı durumlarda bir kaydın bâzı bilgileri alındığında bâzı bilgileri otomatik olarak oluşturulabilir. Misal, bir ürünün KDV dâhil satış fiyatı bilgisini ayrı bir sütunda tutmak istiyorsak, ürünün ham fiyat bilgisi girildiğinde bunu hesaplayıp, yeni bir sütunda depolayan bir kod yazabiliriz.. Aslında ürünün KDV dâhil satış fiyatını ayrı sütunda tutmak veritabanı mâliyetimizi arttırır; fakat hem ham fiyat sütununu, hem de KDV'li satış fiyatı sütununu sıkça kullanıyorsak performans kazancı için bunu yapmak isteyebiliriz
- Türetilmiş sütunun basit tanımı şöyledir:

```
CREATE TABLE users(
    #...
    columnName GENERATED_ALWAYS AS (...)
);
```

- Misal, ürün ham fiyatı girildiğinde KDV'li fiyatı hesâplamak istiyorsak:

```
CREATE TABLE ETradeProducts(
    id INT AUTO_INCREMENT PRIMARY KEY,#ürün numarası
    name VARCHAR(50),# ürün ismi
    categoryId INT,# ürün sınıf numarası
    naivePrice DOUBLE,# ürün ham satış fiyatı
    sellPrice DOUBLE GENERATED ALWAYS AS (
        naivePrice + ((naivePrice / 100) * 18)
    )# ürün KDV'li satış fiyatı
);
```

- Yukarıdaki gibi bir tablo oluşturduğumuzda, otomatik üretilen sütuna değer veremeyiz; ekleme işlemi şöyle olmalıdır:

```
INSERT INTO ETradeProducts(
    name, categoryId, naivePrice)
VALUES(
    Telefon', 1, 100.0
)
```

> ***NOT :*** Değeri otomatik oluşturulan sütunlarla ilgili bilmemiz gereken önemli şeylerden birisi, `AS` ifâdesinden sonra parantez içerisinde belirttiğimiz kodun 'değişmez' (deterministik) olmasıdır. Bir işlem aynı girdilerle hep aynı sonucu veriyorsa bu işleme değişmez (deterministik) denir. Misal, 2 + 2 her zamân böyle bir işlemdir; fakat zamân ve târih belirten fonksiyonlar böyle olmadığından (farklı zamânlarda farklı sonuçlar verdiğinden) bu fonksiyonları bu şekilde kullanamıyoruz. Misal, kullanıcının doğum târihini bildiğimizde yaşını hesâplayıp, bir sütun olarak otomatik kaydetme işlemini bu yapıyla ('generated column') yapamayız. Yâni şu kod hatâ verir:
>
> ```
> CREATE TABLE users(
>     id INT AUTO_INCREMENT PRIMARY KEY,
>     name VARCHAR(50),
>     surname VARCHAR(50),
>     birthdate DATE,
>     age INT GENERATED ALWAYS AS (YEAR(CURDATE()) - YEAR(birthdate))
> );# KOD ÇALIŞMAZ : Error Code: 3763. Expression of generated
> # column 'age' contains a disallowed function: curdate.
> ```

- MySQL'de iki tip türetilmiş sütun vardır : sanal ('virtual') ve saklı ('stored') Sanal türetilmiş sütun aslında var olmayan; fakat veri okunurken hesaplanarak kullanıcıya döndürülen sütundur.
- Saklı ('stored') sütun ise veritabanına gerçekten ayrı bir sütun olarak kaydedilen sütundur.
- Türetilmiş sütunun tam tanımı şöyledir, köşeli parantez içerisindekiler ihtiyârî (tercihe bağlı)dir:
- ```
  CREATE TABLE users(
      #...
      columnName [GENERATED_ALWAYS] AS (...)
      [VIRTUAL | STORED]
  );
  ```

KISITLAR ('CONSTRAINTS')
  ```

- Kısıtlar veritabanının tablo üzerinde işlem yaparken dikkate aldığı kurallardır.
- Veritabanında pek çok kısıt bulunmaktadır; veritabanının farklı isimle anılan bâzı özellikleri de aslında birer kısıttır. Misal, anahtarlar da birer kısıttır.
- Anahtar kısıtları bir sonraki bölümde ayrıca işlendiğinden burada detayını anlatılmıyor.
- Bir tablonun herhangi bir sütunu için bir kural belirleyerek, veritabanının ekleme, tazeleme işleminde bu kısıtı kontrol etmesini sağlayabilirsiniz.
- Kısıt tanımlanmasının genel sözdizimi şöyledir:

```
CREATE TABLE tablo(
    sutun1 veriTipi,
    sutun2 veriTipi,
    # ... : diğer sütunlar
    CONSTRAINT kısıtIsmi
    kısıtTipi kısıtlaIlgiliKod
);
```

- MySQL için pek çok farklı kısıt tipi bulunmaktadır.

> ***NOT ! :*** **MySQL'de kısıt isimleri tablo bazında değil, veritabanı bazındadır; dolayısıyla bir tabloda kullanılan bir kısıt ismi başka tabloda da olsa yeniden kullanılamaz!**

1) Münferid Değer (Tekillik) Kısıtı ('UNIQUE')

- Veritabanındaki en önemli şeylerden birisi her satır veri kaydının birbirinden ayırt edilebilmesi, aranan kayda hızlı şekilde erişilebilmesidir.
- Her kaydı birbirinden ayırt etmek için bir sütunu münferid yaparız. Bu, bir kaydın bu sütundaki değerinin başka bir kayıtta olmadığının garanti edilmesidir.
- Böyle bir sütunu eklemenin en önemli iki faydası vardır:
    1. Veriler birbirinden ayırt edilebilir
    2. Verilerin bu sütuna göre sıralaması kaydedileceğinden ötürü **bu sütun üzerinde yapılan** bir arama sorgusunda aranan kayda hızlıca ulaşılır.
- Bu kısıt iki farklı şekilde eklenebilir; köşeli parantez içini yazmak tercihe bağlıdır:

```
# Birinci usûl:
CREATE TABLE users(
    id INT PRIMARY KEY,
    name VARCHAR(50),
    surname VARCHAR(50),
    phoneNumber CHAR(11) UNIQUE# İlgili sütundan sonra 'UNIQUE' yaz
);

# İkinci usûl:
CREATE TABLE users(
    id INT PRIMARY KEY,
    name VARCHAR(50),
    surname VARCHAR(50),
    phoneNumber CHAR(11),
    # ... : diğer sütun tanımları..
    CONSTRAINT kısıtIsmi# Bu satırı yazmak tercihe bağlıdır.
    UNIQUE(phoneNumber)# Sütunların sonuna 'UNIQUE(sutun)' biçiminde..
);
```

- Eğer ilgili kısıta bir isim verilirse o isimde bir sıralama (indeks) eklenir.
- Bir münferidlik kısıtı birden fazla sütunu kapsayabilir. Aşağıdaki örnekte kullanıcının ismi sütunu ('name') aynı değeri taşıyabilir; kullanıcının soyismi sütunu ('surname') aynı değeri taşıyabilir; fakat kullanıcının ismi + soyismi aynı olamaz:

```
CREATE TABLE users(
    id INT PRIMARY KEY,
    name VARCHAR(50),
    surname VARCHAR(50),
    phoneNumber CHAR(11),
    UNIQUE(phoneNumber),
    UNIQUE(name, surname)
);
```

> **NOT! :** MySQL'de münferidlik kısıtı olan bir alana `NULL` değeri verilebilir; zîrâ NULL değeri bir değer olmama durumunu işâret eden bir simgedir; fakat MsSQL'de `UNIQUE` kısıtı eklenmiş bir sütuna en fazla bir tâne NULL değeri atanabilir.

- Eğer ilgili `UNIQUE` kısıtına isim vermezsek, oluşan indeksin ismi varsayılan olarak sütun ismidir; eğer sütun isminde bir indeks varsa...

Münferid değer kısıtını kilme

- 'Münferid değer' kısıtı aslında bir sıralama ('indeks') oluşturur; bu indeks silinirse ilgili `UNIQUE` kısıtı düşer. Dolayısıyla indeksi kaldırmak bu kısıtı silmek için kâfî:

```
DROP INDEX kısıtIsmi ON tabloIsmi;
```

- Muadil bir başka yöntem ise `ALTER TABLE` ise tabloyu değiştirmektir:

```
ALTER TABLE tabloIsmi
    DROP INDEX kısıtIsmi;
```

- ..

2) Boş Olmama ('NOT NULL') Kısıtı

- Bir satırda bir sütunun boş olmaması isteniyorsa bu sütun için **NOT NULL** kısıtı eklenir.
- Sözdizimi:

```
#İlgili sütundan sonra 'NOT NULL' eklenir:
CREATE TABLE users(
    id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    surname VARCHAR(50) NOT NULL,
    phoneNumber VARCHAR(11) NOT NULL UNIQUE
);
```

- Tabloya bir veri eklerken veyâ tablodan bir veri silerken `NOT NULL` kısıtı eklenen sütunun değeri boş bırakılamaz. Bunun bir istisnası vardır; eğer ilgili sütun için varsayılan bir değer belirtildiyse (`DEFAULT` kısıtına bakınız), sorun olmaz!.

Vâr olan bir tabloya boş olmama kısıtının eklenmesi

- Bir tabloya boş olmama kısıtını sonradan eklemek istediğimizde dikkat etmemiz gereken şey tablodaki hiçbir satırın ilgili sütundaki değerinin boş olmadığından emîn olmaktır.
- Ardından, şu biçimdeki kodu kullanarak ilgili sütunu yeniden tanımlayabiliriz:

```
ALTER TABLE tabloIsmi
    CHANGE
        sutunIsmi
        sutunYeniIsmi veriTipi NOT NULL;
# 'NOT NULL' ifâdesinden önce veyâ sonra
# -eğer varsa- diğer kısıtlar eklenmelidir.
```

- **Bir tabloda kayıtlardan en az birisinde A sütunu için `NULL` verisi varsa, A sütunu için `NOT NULL` kısıtı eklenemez!**

Boş olmama kısıtının kaldırılması

- Bunun için şu sözdizimindeki kod çalıştırılır:

```
ALTER TABLE tabloIsmi
    MODIFY sutunIsmi veriTipi;
# 'veriTipi'nden sonra varsa diğer kısıtlar eklenmelidir.
```

- Aslında bir evvelki kodda olduğu gibi `CHANGE` deyimiyle de sütun yeniden tanımlanabilir ve yeni tanıma `NOT NULL` ifâdesi eklenmediğinde ilgili kısıt kaldırılmış olur.

Varsayılan Değer ('DEFAULT') Kısıtı

- Varsayılan değer bir tabloya veri eklenirken verilerin bâzı sütunlarının otomatik olarak doldurulması için kullanılan bir deyimdir. Eğer ilgili sütun için veri belirtilmezse bu sütuna varsayılan değer yazılır.

- Sözdizimi:

```
CREATE TABLE users(
    id INT PRIMARY KEY,
    name VARCHAR(50) DEFAULT 'isimsiz' NOT NULL,
    surname VARCHAR(50) NOT NULL,
    phoneNumber VARCHAR(11) NOT NULL UNIQUE
);
```

- 'users' tablosuna veri eklerken 'name' sütunu için bir değer belirtmediğimizde bu yeni eklenen kaydın 'name' sütununa otomatik olarak 'isimsiz' değeri yazılır.

> ***NOT :*** MySQL'de `DEFAULT` kısıtı değer belirtilmeyen sütunlar için çalışır. Bir sütunun değerinin belirtilmemesi o sütuna `NULL` değerinin atanması ile aynı şey değildir. Dolayısıyla DEFAULT kısıtı eklenen bir sütun için `NULL` değeri atanırsa, eğer ilgili sütun `NULL` değerini kabûl ediyorsa, o sütun için `NULL` değeri atanır; kabûl etmiyorsa, veritabanı sistemi hatâ verir.

'Varsayılan değer' kısıtının sonradan eklenmesi

- Diğer kısıtlarda olduğu gibi `CHANGE` ve `MODIFY` deyimleriyle yapılabilir:

```
# Yöntem - 1:
ALTER TABLE users
    CHANGE
        surname
        surname VARCHAR(50) NOT NULL DEFAULT 'soyisimsiz';
# Yöntem - 2:
ALTER TABLE users
    MODIFY surname VARCHAR(50) NOT NULL DEFAULT 'soyisimsiz';
```

> ***NOT :*** `ALTER TABLE tabloIsmi ALTER COLUMN sutunIsmi SET DEFAULT deger` biçimindeki kod MySQL'in 8.0.24 sürümünde çalışmıyor.

'Varsayılan değer' kısıtının kaldırılması

- Varsayılan değer kısıtının kaldırılması için `CHANGE` ve `MODIFY` deyimleri kullanılabilir; fakat bu deyimleri kullandığımızda sütun üzerindeki diğer kısıtları yeniden tanımlamamız gerekiyor; bu sebeple şu kodu tercih etmeliyiz:

```
ALTER TABLE tabloIsmi
    ALTER COLUMN sutunIsmi DROP DEFAULT;
```

- `DEFAULT` kısıtının en çok kullanıldığı alanlardan birisi de târih ve zamân barındıran alanlardır. Bir sütuna varsayılan târih değeri olarak sorgunun çalıştırıldığı târihin atanması isteniyorsa bunun için `DEFAULT` tanımından sonra `NOW()` yazılabilir; yâ dâ `CURRENT_TIMESTAMP` yazılabilir.

- Bir tabloda bir veri değiştirme / tazeleme işleminde veri değişikliğinin yapıldığı târih otomatik olarak kaydedilmek isteniyorsa, bunun birkaç yolu vardır. Veritabanı sunucusuna gelen trafiği izlemek, bir tetikleyici yazmak gibi çözümler uygulanabilir; fakat bu değişikliğin târihi, veri açısından önemliyse ve tabloda tutulmak isteniyorsa bunun için özel bir yapı vardır:

```
CREATE TABLE users(
  id INT
    PRIMARY KEY
    AUTO_INCREMENT,
  name VARCHAR(50)
    NOT NULL,
  updatedAt DATETIME
      NOT NULL
      DEFAULT NOW()
      ON UPDATE CURRENT_TIMESTAMP
  );# CURRENT_TIMESTAMP yerine NOW() da yazılabilir.
```

Kontrol ('CHECK') Kısıtı

- Bu kısıt bir sütuna veri eklenirken veri üzerinde yapılması gereken kontrol işlemlerini tanımlamak için kullanılır. Misal, bu kısıtı kullanarak yaşı 13'den küçük olan bir kullanıcı kaydının oluşmasına engel olunabilir.
- **Bu özellik MySQL'in 8.0.16 sürümüyle gelmiştir; dolayısıyla önceki sürümlerde böyle bir işlev için tetikleyici gibi yapılar kullanılmalıdır.**
- Şu sözdizimi kullanılır; köşeli parantez içindekiler ihtiyârî (tercihî)dir:

```
CREATE TABLE tabloIsmi(
    sutunIsmi veriTipi,
    # ... : diğer sütun tanımlamaları
    CONSTRAINT kısıtIsmi
    CHECK(...)# '...' yerine true | false değeri üreten deyim yazılır
    [ENFORCED | NOT ENFORCED]
);
```

- Misal:

```
CREATE TABLE users(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    surname VARCHAR(50) NOT NULL,
    age INT NOT NULL DEFAULT 0,
    phoneNumber VARCHAR(11) NOT NULL UNIQUE,
    CONSTRAINT isimKontrol
    UNIQUE(name, surname),
    CONSTRAINT yasKontrol
    CHECK(age >= 13)
);
```

- Kontrol kısıtı oluşturmanın bir başka yolu ise sütun tanımının hemen peşine şu sözdizimiyle ilgili kısıtı eklemektir: `CHECK(ifade) [ENFORCED | NOT ENFORCED]` Fakat bu söz dizimini kullanmanın bir zayıf yönü vardır; o da bu kısıt içerisinde sadece belirtilen kısıta referans verebiliyor olmanızdır. Misal, şu ifâde hatâ fırlatır:

```
CREATE TABLE users(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL,
    surname VARCHAR(50) NOT NULL,
    age INT NOT NULL DEFAULT 0,
    phoneNumber VARCHAR(11) UNIQUE
        CHECK(age >= 13 OR phoneNumber IS NOT NULL),
    CONSTRAINT isimKontrol
    UNIQUE(name, surname)
);
```

> ***NOT :*** Bir `CHECK` kısıtına isim verilmezse bu kısıt varsayılan olarak şu şekilde isimlendirilir : tabloIsmi_chk_n Buradaki 'n' kısıt rakamıdır; 1'den başlar; ikinci kısıt için 2 değeri alır..

ANAHTARLAR ('KEY')

- İlişkisel veritabanında ilişkiyi sağlayan şey anahtar yapısıdır. Anahtarlar kayıtları diğerlerinden ayırt etmeyi sağlayan yapılardır.
- Anahtar, bir tabloda bir veriyi diğerlerinden ayıran sütun(lar) veyâ başka tablodaki bir kaydı işâret eden sütun olarak tanımlanabilir.
- Anahtarlar 'birincil anahtar' ('primary key') ve dış anahtar ('foreign key') olmak üzere ikiye ayrılır.

Birincil Anahtar ('Primary Key')

- Birincil anahtar bir tabloda verileri diğerlerinden ayırt etmeyi sağlayan bir yapıdır.
- Veriler arasında münferidliği ('tekilliği') sağlar.
- Dolayısıyla birincil anahtar alanı olan bir tabloda hiçbir kayıt diğer kaydın birebir aynısı olamaz.
- **Bir tabloda sadece bir tâne birincil anahtar olabilir.**
- Veriler, birincil anahtara göre indeksli olarak sıralanır; dolayısıyla birincil anahtar sorgulama yapılırken kullanılırsa, ilgili veriye çok daha hızlı şekilde ulaşılabilir.
- Bir birincil anahtar tablodaki bir sütun olabilir; fakat birden fazla sütunun birleşimi de olabilir. Misal, araba marka - model kayıtlarının tutulduğu bir tablo düşünelim. Aynı markanın onlarca model arabası olabildiğinden marka sütunu benzersiz değildir. Farklı markaların aynı isimde arabaları olabileceğinden ötürü model sütunu da tek başına benzersiz değildir; fakat marka+model sütunu benzersizdir; yanî bir arabanın hem markası, hem de model ismi aynıysa o araba aynı arabadır. Aşağıdaki tabloyu inceleyin:

Marka	Model	Marka + Model
A	T1	A T1
B	T1	B T1
A	T2	A T2

  Yukarıdaki tabloda 'Marka' sütunu benzersiz değildir; 'A' değerini içeren iki satır vardır. 'Model' sütunu da benzersiz değildir, 'T1' modelinden iki araba vardır; fakat Marka + Model benzersiz bir veridir. Tablomuzda böyle bir anahtar oluşturmak istediğimizde bu anahtarı yeni bir sütunda tutmak zorunda değiliz; yanî gerçekte 'Marka + Model' diye bir sütun olmadan da 'Marka + Model' birincil anahtarımız olabilir.
- Bir birincil anahtar şunları içerir:
  1. Boş olmama kısıtı : `NOT NULL`
  2. Benzersizlik, Münferidlik kısıtı : `UNIQUE`
  3. Fiziksel, benzersiz Sıralama (indeks) : `CLUSTERED UNIQUE INDEX`
- Bir birincil anahtar eklemenin 2 farklı sözdizimi vardır:

```
CREATE TABLE products(
    id INT PRIMARY KEY,# 1 : Sütunun peşine PRIMARY KEY yazarak
    name VARCHAR(100),
    #... : diğer sütunlar
);
###
CREATE TABLE products(
    id INT,
    name VARCHAR(100),
    #... : diğer sütunlar
    PRIMARY KEY(id)# 2 : Sütunların sonuna bir kısıt gibi ekleyerek
);
```

- Eğer birden fazla sütunu kapsayan bir birincil anahtar oluşturulmak isteniyorsa;

```
CREATE TABLE cars(
    brand VARCHAR(50),
    model VARCHAR(50)
    #... : diğer sütunlar
    PRIMARY KEY(brand, model)
);
```

- Mevcut bir birincil anahtar şu şekilde eklenir:

```
ALTER TABLE cars
ADD PRIMARY KEY(brand, model);
```

- Mevcut tablodan bir birincil anahtar şu şekilde silinir:

```
ALTER TABLE cars
DROP PRIMARY KEY;
```

Dış Anahtar ('Foreign Key')

- Dış anahtar bir tablodaki bir satırın başka tablodaki başka bir satırı işâret eden sütununa verilen isimdir. İlişkisel veritabanında ilişkileri kurmak açısından en önemli yapı dış anahtar yapısıdır. Misal, kullanıcıların yaşadıkları şehir bilgisini tutmak istiyorsunuz; bu durumda her kullanıcı için şehir ismini tutmak hem mâliyyetlidir, hem de güvenlik riski barındırmaktadır, zîrâ kullanıcılar gerçekten vâr olmayan bir şehir ismi girerek sizi yanıltabilir. Bunun yerine tüm şehirleri bir tabloda tutup, ilgili tablodaki 'id' alanını kullanıcı tablosunda tutmak daha doğru olur. Şehirler tablosu:

id	name
01	Adana
02	Adıyaman

Kullanıcılar tablosu:

id	name	city
01	Ahmed	02

- Yukarıda, Adıyaman'da yaşayan Ahmed'in bilgileri gösterilmektedir.
- Bu şekildeki bir kodlama bize bir ilişki olduğunu anlatır; fakat iki tablo arasında herhangi bir ilişki kurulduğunu ispat etmez. Her iki tabloyu birbirinden bağımsız olarak tuttuğumuzda veritabanı şehirler tablosunda kaydı olmayan bir şehrin kullanıcılar tablosundaki 'city' sütununa eklenmesini önlemeyecektir; yanî veri tutarlılığı sağlanmamış oluyor. Benzer şekilde eğer tabloları birbirinden bağımsız tutarsak Adıyaman'da oturan bir müşterinin kaydının tutulduğu veritabanının şehirler tablosundaki Adıyaman satırını kaldırdığımızda veritabanı bizi engellemeyecektir; yanî veri tutarlılığı sağlanmayacaktır. İşte bu sebeple 'dış anahtar' kullanılır.
- Bir tabloda birden fazla sütun bir dış anahtarı tanımlayabilir.
- Bir dış anahtarın oluşturulmasının sözdizimi şu şekildedir:

```
CREATE TABLE cities(
    id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE users(
    id int PRIMARY KEY,
    name VARCHAR(50),
    city INT,
    CONSTRAINT sehirDisAnahtari
    FOREIGN KEY(city) REFERENCES cities(id)
);
```

- Bir dış anahtar aslında bir kısıt olduğundan bu kısıta istediğimiz ismi verebiliriz; eğer bu kısıta bir isim vermezsek veritabanı sistemi otomatik isim atar.

- **Mantık olarak bir dış anahtar mutlaka benzersiz bir satırı işâret etmelidir.** Bu sebeple bir dış anahtarın işâret ettiği sütun(lar) benzersiz olması gerekir; fakat MySQL'de dış anahtarın benzersiz bir satırı işâret etmesi zorunlu değildir (misal, MsSQL'de zorunlu); fakat doğru bir veritabanında çoğu kez bir **dış anahtar bir sütunu ifâde ediyorsa o sütun benzersiz değer içerir** (ya `PRIMARY KEY` olmalı, ya da münferidlik kısıtı (`UNIQUE`) barındırır).

> ***NOT :*** Bir dış anahtar hedef tabloda birden fazla satırı işâret ediyorsa, bu durumda doğru tasarım için hedef tabloda ilgili satırların birleşimi için bir münferidlik kısıtı bulunmalıdır. Hedef tablodaki sütunların münferidlik kısıtı bulunsa bile kâfî değildir; ilgili sütunların birleşimi için bir münferidlik kısıtı olmalıdır. Aşağıdaki kodu inceleyiniz:

```
CREATE TABLE cars(
    brand VARCHAR(50) PRIMARY KEY,
    model VARCHAR(50) UNIQUE
);# Bu tablo tanımıyla aşağıdaki 'users' tablosu oluşmaz:
CREATE TABLE users(
    id INT,
    fKeyPart1 VARCHAR(50),
    fKeyPart2 VARCHAR(50),
    CONSTRAINT kullaniciArabasi
        FOREIGN KEY (fKeyPart1, fKeyPart2)
        REFERENCES cars(brand, model)
);
# 'users' tablosunun oluşabilmesi için doğru kod şudur:
CREATE TABLE cars(
    brand VARCHAR(50),
    model VARCHAR(50),
    PRIMARY KEY(brand, model)
);
```

- Veri bütünlüğünü ve veri tutarlılığını sağlamak bir dış anahtarın farklı yapılandırma ayarları vardır.
- Bir dış anahtar oluşturduğunuzda ilgili tablodaki dış anahtar sütunu (veyâ sütunları) için bir sıralama (indeks) eklenir.
- Dış anahtar oluşturmak bize pek çok kısıtın otomatik eklenmesini sağlar.
- **Bir tablonun bir sütunu başka bir tabloda dış anahtar olarak bulunuyorsa, bu tablo silinemez.**
- Dış anahtar oluşturmanın tam tanımı şöyledir, köşeli parantez içerisinde yazılanlar ihtiyârîdir (tercihe bağlıdır):

```
CREATE TABLE tabloIsmi(
    sutun1 veriTipi,
    sutun2 veriTipi,
    #... : diğer sütunlar
    [CONSTRAINT kisitIsmi]
    FOREIGN KEY [disAnahtarIsmi] (sutun1, ...)
    REFERENCES hedefTablo(hedefTabloSutun1, ...)
    [ON DELETE optionReference]
    [ON UPDATE optionReference]
);
```

- Yukarıdaki kodda tercihî olarak belirtilen son iki satırı inceleyelim inşâAllâh:

  Buradaki ayarlar hedef tablodaki satırların ilgili sütunları silindiğinde veyâ değeri değiştirildiğinde bu tabloda o satıra referans veren satır(lar) için ne işlem yapılacağını bildiriyor. Burada beş seçenek mevcut, 'optionReference' kısmına şunlardan birisi yazılabilir:

  1. `CASCADE` : Hedef tabloda işâret edilen satır(lar)ın ilgili sütunları değiştiğinde veyâ tazelendiğinde bu tablodaki ilgili satırlar silinir.
  2. `SET NULL` : Hedef tablonun ilgili sütununda / sütunlarında değişiklik yapıldığında bu tablodaki ilgili sütuna NULL değeri atanır.
  3. `RESTRICT` : **Varsayılan ayardır**. Hedef tablonun ilgili sütununda değişiklik yapılmasına izin verilmez.
  4. `NO ACTION` : `RESTRICT` ile aynı işlevdedir. MsSQL'de `RESTRICT` geçerli değildir; fakat `NO ACTION` geçerlidir.
  5. `SET DEFAULT` : İlgili ayar MySQL ayrıştırıcısı('parser')na bırakılır. Bu ayar InnoDB motoru ve NDB tabloları için geçerli değilmiş.

- MySQL `RESTRICT`, `CASCADE` ve `SET NULL` ayarını tamâmen destekliyormuş.

> ***NOT :*** Bâzı durumlarda dış anahtar kısıtını devredışı bırakmak isteyebiliriz. Misal, yüksek miktarda ve dış anahtar bakımından tutarlı bir veriyi içeri aktarmak istiyorsak, bu durmda dış anahtarı devredışı bırakmak uygun olabilir. Çünkü veritabanı hedef tablodaki her kayıt tazelemede, her silme işleminde ve dış tablodaki her yeni veri ekleme işleminde ilgili ilişkiyi kontrol eder; bu da nisbeten mâliyyetli bir işlemdir. Aşağıdaki kod bunu açıp, kapatmayı göstermektedir; fakat **dikkat edilmesi gereken husus yapılan bu ayarın sadece ilgili bağlantı için geçerli olduğudur. Eğer bağlantı kapanır ve yeniden bağlanırsak, varsayılan olarak dış anahtar denetleme bayrağı 'true' olacaktır**:

```
SET foreign_key_checks = 0;# Dış anahtar denetlemeyi kapat
SET foreign_key_checks = 1;# Dış anahtar denetlemeyi aç
```

Genel Anahtar ('KEY')

- Yukarıdakilerin yanı sıra MySQL'de sadece `KEY` yazılarak da anahtar belirtilebilir:

```
CREATE TABLE users(
    id int AUTO_INCREMENT,
    name VARCHAR(100),
    surname VARCHAR(100),
    email VARCHAR(200),
    PRIMARY KEY(id),# Birincil anahtar..
    KEY(email)# Bu sütun için sıralama (indeks) eklenir.
);
```

- Yukarıdaki aslında sadece 'email' sütunu için bir sıralama (indeks) ekler.

> ***NOT :*** Sadece `KEY` anahtar kelîmesi MsSQL veritabanı sisteminde çalışmaz.

VERİ EKLEME ('INSERT')

- Bir tabloya veri eklemek için INSERT komutu şu şekildedir kullanılır:

```
INSERT INTO tabloIsmi (sutun1, sutun2, ..)
VALUES (deger1, deger2, ...);
```

- Buradaki sütun isimlerinin yeri değiştirilebilir; fakat bu durumda ilgili değerlerin sütunlara karşılık gelen sırada yazılması gerekmektedir.

- Birden fazla satırı kolayca eklemek için şu sözdizimi kullanılabilir:

```
INSERT INTO tabloIsmi (sutun1, sutun2, ..)
VALUES
    (deger1, deger2, ...),
    (deger1, deger2, ...),
    (deger1, deger2, ...);
```

- Veri ekleme işleminde eğer bir sütun için varsayılan değerin atanmasını istiyorsak, bu durumda iki yöntem uygulayabiliriz:

    1. Sütun ismini ve sütun değerini yazmayarak:

    ```
    INSERT INTO users (name, age, phoneNumber)# 'surname' yazılmadı
    VALUES ('Mehmed', 26, '12345678901');# soyisim de yazılmamalı
    ```

    2. İlgili sütunun değer kısmındaki karşılığına `DEFAULT` yazmak:

    ```
    INSERT INTO users (name, surname, age, phoneNumber)
    VALUES ('Mehmed', DEFAULT, 26, '12345678901');
    ```

- Veri eklerken dikkat edilmesi gereken husus otomatik artan olarak belirtilen alan için herhangi bir değerin belirtilmemesi gerektiğidir. Misal, yeni oluşturulan bir tablonun 'id' sütununun otomatik artan olarak ayarlandığını düşünelim. Bu durumda aşağıdaki kod çalışır; fakat 'id' sütunu kaldığı yerden devâm ettiğinden 'id' değeri elle belirtilmeyen ikinci kaydın münferid numarası ('id'si) 7 olur.

```
INSERT INTO users (id, name, surname, age, phoneNumber)
VALUES (6, 'Mehmed', DEFAULT, 26, '12345678901');
```

- MySQL'de tüm sütunlar için veri girişi yapacaksanız sütun isimlerini tek tek belirtmenize gerek yok; fakat bu durumda otomatik artan alan için elle değer belirtmeniz gerekeceğinden otomatik artan bir sütun varsa (ki hemen her tabloda olmalı) bu seçeneği tercih etmemelisiniz. Öte taraftan yazdığınız SQL kodunun başka sistemler için de geçerli olmasını istiyorsanız, bunu tercih etmek istemeyebilirsiniz (MsSQL de sütun isimleri yazılmadan veri eklemeyi destekliyor; ama desteklemeyenler vardı sanırım).

> ***NOT :*** MySQL'de birden fazla veriyi tek bir deyimle içe aktarmaya çalıştığınızda, verilerden birisi eklenemediğinde sistem geriye sarılır ve hiçbir veriyi ekleyememiş olursunuz!

- MySQL'de aynı anda kaç veri ekleyebileceğiniz konusunda net bir şey söylemek zordur; teorik olarak buna engel yok; fakat gönderdiğiniz verilerin boyutu veritabanı sistemine

aktarılan paketin boyutunu aşarsa sistem hatâ verir ve bağlantıyı sonlandırır. Azamî paket boyutunun ne olduğunu öğrenmek için şu kodu çalıştırabilirsiniz:

```
SHOW VARIABLES LIKE 'max_allowed_packet';
# Sonuç olarak dönen değer byte cinsindendir.
```

- Bu değeri değiştirmek için Değişkenler -> Sistem Değişkenleri kısmına bakınız.

Sorgu sonucunu bir tabloya veri olarak ekleme

- Bir başka veri ekleme yöntemi ise bir sorgudan dönen değerleri başka bir tabloya eklemektir. Böyle yaptığınızda `max_allowed_packet` değişkeni ekleyeceğiniz verinin boyutunu değiştirmez; çünkü tüm işlem veritabanında gerçekleştirilmektedir.

- Bu usûlün sözdizimi şöyledir:

```
INSERT INTO tabloIsmi(sutun1, sutun2, ...)
    SELECT secim_listesi FROM digerTablonunIsmi WHERE sart
```

- Misal:

```
INSERT INTO usersYedek(id, name, surname, age, phoneNumber)
SELECT id, name, surname, age, phoneNumber
FROM users;
##
INSERT INTO usersYedek SELECT * FROM users LIMIT 1;
# Tüm sütunlar seçildiğinde sütun isimlerine gerek yok.
```

- Bu şekilde veri ekleme işlemi yürütülürken bir hatânın oluşması durumunda işlem geriye sarılır ve tüm satırlar eklenmemiş olur.

Sorgu sonucunu eklerken kayıt çakışması durumu

- Müşteri verilerinin tutulduğu bir tablonuzun olduğunu düşünün. Vâr olan bir yedeği tazelemek istiyorsunuz. Yedek tabloda müşterilerinizin verileri var; fakat bu tablo biraz eski; dolayısıyla yeni eklenen müşterilerin yeni müşteri olarak eklenmesini; fakat vâr olan müşterilerin ise verilerinin tazelenmesini istiyorsunuz. Bu durumda `INSERT ON DUPLICATE KEY UPDATE` deyimini kullanabilirsiniz, vâr olan müşterilerinizin bilgileri tazelenir, olmayan müşteriler yeni müşteri olarak eklenir:

```
INSERT INTO usersYedek
    SELECT * FROM users U
ON DUPLICATE KEY UPDATE# Anahtar deyim.
# Çakışma olursa tazelenecek satırlar alta şu şekilde yazılır:
    phoneNumber = U.phoneNumber,
    email = U.email;
# Çalıştırıldığında yeni veriler eklenir, çakışan verilerin
# telefon numarası ve e-posta adresi tazelenir.
```

- Bu deyimle çok kullanışlı olan bir özellik yeni satırın ve sütunların isimlendirilmesidir. Diğer -> Yeni Eklenen Veride İsimlendirme kısmına bakınız.

> ***NOT :*** `ON DUPLICATE KEY UPDATE` ifâdesi çalıştırıldığında geriye dönen değer mesajda farklılık oluyor. Geriye dönen mesajda 'Duplicates: ' biçimindeki kısımda şu üç değerden biri olabilir: 1 : En az bir yeni satır eklenmiştir.
>
> 2 : Mevcut satır tazelenmiştir.
>
> 0 : Yapılan işlem hedef tabloda hiçbir değişikliğe sebebiyyet vermemiştir

Birden fazla veri eklemede hatâ olması durumunu ele alma

- Varsayılan olarak, bir ifâdeyi çalıştırdığımızda bu ifâde bizim için bir bütüncül işlem ('transaction') olarak ele alınır ve herhangi bir hatâ olması durumunda sistem geriye sarılarak eski hâline getirilir.
- Birden fazla veri eklerken de bu durum böyle oluyor; bunun önüne geçmek için veri `INSERT IGNORE` deyimi kullanılır. Bu deyim kullanıldığında, herhangi bir satır eklenemediğinde sistem bunu görmezden gelir ve başarılı eklenen satırlar kaydedilir, hatâ yerine uyarı fırlatılır:

```
INSERT IGNORE INTO tabloIsmi(sutun1, sutun2)
VALUES
    (kayit1Sutun1, kayit1Sutun2),
    (kayit2Sutun1, kayit2Sutun2);
```

DATETIME , DATE ve TIME tipleri için veri ekleme

- Bu tipler için veri eklenirken veri metînsel ifâde gibi tek veyâ çift tırnak içerisinde yazılır. Burada verilerin yazım formatı önemlidir.
- `DATETIME` `YYYY-AA-GG SS-DD-SA` formatında, `DATE` `YYYY-AA-GG` formatında ve `TIME` `SS-DD-SA` formatında yazılır:

```
INSERT INTO veri#DATETIME, DATE ve TIME sütunları için
VALUES('2024-05-23 16:34:41', '2024-06-24', '15:33:11');
```

- Şu anki târih değerini eklemek için `NOW()` fonksiyonu kullanılır. Bu fonksiyon üç veri tipi (`DATETIME` , `DATE` ve `TIME`) için de kullanılabilir; veri eklerken uyarı görüntülenir; zîrâ `NOW()` fonksiyonu şu anki târih ve saati döndürdüğünden `DATE` ve `TIME` tipindeki sütunlara bu değer eklenirken veri kaybı olur, bu uyarı bundan dolayıdır; fakat istenen sonuç elde edilir..
- MySQL'i Workbench yazılım arayüzüyle kullanırken fark etmeyebiliriz, biz verimizi metîn gibi çift tırnak veyâ tek tırnak içerisinde yazsak da verimiz yazılım tarafından otomatik olarak hedef veri tipine dönüştürülür. MySQL'i bir API aracılığıyla kullandığımızda veyâ metîn tipindeki bir sütundaki veriyi çekip, târih saat tipindeki bir sütuna otomatik olarak eklememiz gerektiğinde metni ilgili târih saat formatına (üçünden birisi) dönüştürmemiz lazımdır. Bunun için de `STR_TO_DATE` fonksiyonu kullanılır. Bu fonksiyonun kullanımı için ilgili kısma bakınız.

Veri tazeleme (UPDATE)

- Veritabanında kayıtlı olan verilerin tazelenmesi için kullanılır.
- `UPDATE` komutunun kullanımının en önemli kısmı tazelenecek (verisi değiştirilecek) satırların sınırlandırıldığı `WHERE` ifâdesi kısmıdır; **eğer bu kısmı unutursanız tüm tablonun verisini kaybedebilirsiniz.**
- Sözdizimi:

```
UPDATE [LOW_PRIORITY] [IGNORE] tabloIsmi
SET
    sutun1 = yeniDeger1,
    sutun2 = yeniDeger2,
    sutun3 = yeniDeger3;
[WHERE sart];
```

- Yukarıdaki sözdiziminde `IGNORE` ifâdesi çeşitli sebeplerle (kısıtlar sebebiyle, tetikleyiciler sebebiyle vs) tazelenemeyen satırların olması durumunda kalan satırların tazelenmesini sağlamak için yazılabilen bir komuttur. İhtiyacınıza göre bunu kullanabilirsiniz.
- `SET` ile belirtilen kısımda ise hangi sütun değeri için hangi yeni değerin atanacağı bilgisi belirtilir.

- `WHERE` kısmında ise bu işlemin hangi satırlar için uygulanacağı belirtilir.
- `LOW_PRIORITY` ise tazeleme işleminin yapılma biçimiyle ilgili tercihî bir komuttur. Bu komut yazıldığında tazeleme işlemi tabloya bağlı hiçbir bağlantı kalmadığında gerçekleştirilir. "MyISAM", "MERGE" ve "MEMORY" depolama motorları tablo seviyesinde kilitlemeyi desteklediğinden bu komut bu depolama birimleriyle çalışan sistemler için kullanışlı olabilir.
- `WHERE` kısmının bir tazeleme komutunun en önemli kısmı olduğunu belirtmiştik. Silinecek verinin belirli olması önemli olduğundan MySQL'i Workbench üzerinde kullandığınızda varsayılan olarak `WHERE` deyimi içerisinde `UNIQUE` olmayan bir sütun için değer şartı koyamıyorsun. Misal, tablonuzda isim alanı varsa ve bu alan münferîd (`UNIQUE`) değilse, `WHERE name='Ahmed'` şeklinde bir şart yazdığınız `UPDATE` komutu çalışmıyor. Şöyle bir hatâ ve ipucu döndürülüyor:

  You are using safe update mode and you tried to update a table without a WHERE that uses a KEY column. To disable safe mode, toggle the option in Preferences -> SQL Editor and reconnect
- İşlem yapacağınız satır sayısını kısıtlamak için alt sorgu `LIMIT` deyimi gibi çeşitli yapıları da kullanabilirsiniz.

Tablo birleştirerek tazeleme (UPDATE JOIN)

- Tablo birleştirme işlemi (`JOIN`) `UPDATE` deyimiyle berâber kullanılabilir.
- Bu, tablodaki bir verinin başka tablodaki verinin değişmesi üzerinde doğrudan etkisi olduğu durumlarda kullanışlı olabilir.
- Söz dizimi:

```
UPDATE tablo1 [INNER JOIN | LEFT JOIN] tablo2
ON tablo1.name = tablo2.name
SET
    tablo2.name = CONCAT("kullanici", tablo1.id);
```

- Yukarıdaki kod tablo2'deki 'name' alanında kaydı bulunan kayıtların isim alanını kullanıcı + tablo1'deki 'id' verisi şeklinde ayarlar..
- `SET` deyimi içerisinde hem sağ, hem de sol tablo için atama işlemi yazılabiliyor.
- Yukarıdaki deyim yerine şu deyim de kullanılabilir:

```
UPDATE tablo1, tablo2
SET
    tablo2.name = CONCAT("kullanici", tablo1.id)
WHERE tablo1.name = tablo2.name;
```

Vâr Olan Satırları Yenisiyle Tazeleme (REPLACE deyimi)

- `REPLACE` deyimi SQL standardının bir özelliğidir.
- Bu deyim, `PRIMARY KEY` ve `UNIQUE` kısıtlarını kullanarak vâr olan satırların yenisiyle değiştirilmesini sağlar.
- Sistem şöyle çalışır:
    1. Eklenmek istenen satır tabloya eklenir; eğer 'yinelenen anahtar' hatâsı oluşmazsa..
    2. Eğer bu hatâ oluşursa, bu hatâya sebebiyyet veren satır tablodan silinir, yeni satır tabloya eklenir.
- Misal, ürünler tablosunda zâten kayıtlı olan bir ürün toplu işlem sebebiyle yeniden eklenmek istendiğinde sistem hatâ verir; fakat bu yapı kullanılırsa, eski ürün bilgisi silinir ve yeni ürün bilgisi eklenir. Bunun için ürün ismi gibi münferid bir sütunun olması gerektiğine dikkat ediniz.
- Söz dizimi:

```
REPLACE [INTO] tabloIsmi(sutun1, sutun2) VALUES(deger1, deger2);
```

- Bu sözdiziminin bir muadili şu şekildedir:

```
REPLACE INTO tabloIsmi
SET
    sutun1 = deger1,
    sutun2 = deger2;
```

- Bu komutun ilkinden farkı ise, belirtilmeyen sütunlar için sütun varsayılan değerinin atanmasıdır; eğer sütun için varsayılan değer belirtilmemişse hatâ oluşur.
- `REPLACE` deyimi için bir misal:

```
# Tablo oluştur:
CREATE TABLE products(
    id int PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) NOT NULL UNIQUE,
    info VARCHAR(100));

# Veri ekle:
INSERT INTO productsOther (name, info) VALUES
('Turbox fare', '3200 dpi destekler'),
('Samsung A50', '2 yıl garantili..'),
('Monster Notebook', 'Mühendislik uygulamaları için hârika');

# Verileri sorgula:
SELECT * FROM productsOther;
# Çıktı:
# id        name               info
# 1         Turbox fare        3200 dpi destekler
# 2         Samsung A50        2 yıl garantili..
# 3         Monster Notebook   Mühendislik uygulamaları için hârika!

# REPLACE deyimi:
REPLACE INTO productsOther (name, info) VALUES
('JBL kulaklık', 'Samsung kalitesi...'),
('Monster Notebook', '32 GB RAM ve
yüksek önbelleğiyle mühendislik uygulamaları için hârika!');

# Veri sorgula:
SELECT * FROM productsOther;
# Çıktı:
# id        name               info
# 1         Turbox fare        3200 dpi destekler
# 2         Samsung A50        2 yıl garantili..
# 3         Monster Notebook   32 GB RAM ve yüksek önbelleğiyle
#                              mühendislik uygulamaları için hârika!
```

VERİ SİLME (DELETE)

- Veri silme işlemi veri tazeleme (`UPDATE`) işlemine benziyor.
- Burada da en önemli kısım, silinecek verinin belirli olmasını sağlayan `WHERE` ve / veyâ `LIMIT` ifâdesidir.
- Basit söz dizimi:

```
DELETE FROM tablo1 [WHERE sart];
```

Tabloyu boşaltma (TRUNCATE TABLE)

- `DELETE FROM` deyiminin en bâriz yan etkilerinden birisi tablodaki otomatik artan sütunların kaldığı yerden devâm etmesidir. Eğer tablonun daha evvel hiç veri eklenmemiş gibi sıfırlanmasını istiyorsak `TRUNCATE TABLE` deyimini kullanmalıyız:

```
TRUNCATE [TABLE] tabloIsmi;
```

- Bunu yaptıktan sonra tabloya yeni veri eklediğinizde tablodaki otomatik artan sütunlar baştan başlayacaktır.
- `TRUNCATE TABLE` deyimi boş tabloda da çalıştırılabilir, bu durumda otomatik artan sütunlar başlangıçtaki hâline getirilir.
- `TRUNCATE TABLE` ifâdesindeki `TABLE` ifâdesi MySQL'de ihtiyârîdir (tercihidir), MsSQL'de ise böyle değildir, mutlaka yazılmalıdır.
- Tablodaki otomatik artan sütunlar için bir değer belirtmek istiyorsak bunu tablo yapılandırmasını değiştirerek de yapabiliyoruz.

Birbiriyle ilişkili tablolarda veri silme işlemi

- İlişkisel veritabanlarının en önemli özelliklerinden birisi tabloların birbirleriyle bağlanmasıdır. Birbirine bağlı satırların bulunduğu tablolarda veri silme işleminde ilgili verilerin silinmesine izin verilmesi veyâ ilgili verilerin dış tablodaki ilişkili verilerin de silinmesi gibi birkaç seçenek ortaya çıkıyor, bu konuda detaylı malûmat "*Anahtarlar -> Dış Anahtar ('Foreign Key')*" bölümünde vardır. Basitçe, `ON DELETE` ayarı için verilebilecek şu değerlerin karşılığında şu etkiler oluşmaktadır:
    1. `CASCADE` : İlgili veri silindiğinde, bu veriye bağlı olan dış veri de silinir.
    2. `SET NULL` : İlgili veri silindiğinde, bu veriye bağlı olan dış verinin ilgili sütunu `NULL` yapılır.
    3. `RESTRICT` : A tablosunda bir satır B tablosundaki bir satır tarafından bağlanmışsa, A tablosundaki satırın silinmesine izin verilmez.
- `ON DELETE CASCADE` ayarı için bir misal:

```
# Tabloları oluştur:
CREATE TABLE tbl1(name varchar(20), INDEX(name));
CREATE TABLE tbl2(
    id int PRIMARY KEY AUTO_INCREMENT,
    name varchar(20),
    FOREIGN KEY(name)
        REFERENCES tbl1(name) ON DELETE CASCADE);

# Veri ekle:
INSERT INTO tbl1 VALUES('Mehmed'), ('Fâruk'), ('Ömer'), ('Fâzıl');
INSERT INTO tbl2(name) VALUES('Mehmed'), ('Fâruk'), ('Ömer');
DELETE FROM tbl1 WHERE name = 'Mehmed';
# Yukarıdaki satır 'tbl2' tablosundaki 'Mehmed' verisini de siler.
```

Tablo birleştirerek veri silme (DELETE JOIN)

- Temel söz dizimi:

```
DELETE tablo1, tablo2
FROM tablo1
INNER JOIN tablo2 ON tablo1.ilgiliSutun = tablo2.ilgiliSutun;
```

- Bu şekilde birbiriyle ilişkili kayıtlar kolayca silinebilir..
- Bu deyim, tabloların dış anahtarla ilişkilendirilmesi durumunda çok kullanışlıdır; zîrâ böyle bir durumda kayıtların başka tablodaki bir satırla ilişkilendirilmesi sebebiyle silinmesini engelleniyor olabilir (bu ayar tablo bazındadır ve kullanıcı tarafından belirlenebilir). Yanî `DELETE JOIN` ifâdesi birbiriyle ilişkili satırların silinmesi için güvenli bir yoldur.
- `DELETE JOIN` ifâdesi için bir misal:

```
# Tabloları oluştur:
CREATE TABLE tablo1(name VARCHAR(10));
CREATE TABLE tablo2(name VARCHAR(10));

# Verileri ekle:
INSERT INTO tablo1 (name) VALUES('Mehmed'), ('Ahmed'), ('Fâruk');
INSERT INTO tablo2 (name) VALUES('Mehmed'), ('Hasân'), ('Kâmil');

# Veri sorgula:
SELECT * FROM tablo1;
SELECT * FROM tablo2;
# Çıktı1:             Çıktı2:
# name                name
# Mehmed              Mehmed
# Fâruk               Hasân
# Ahmed               Kâmil

# DELETE JOIN ifâdesini çalıştır:
DELETE tablo1, tablo2 FROM tablo1
    INNER JOIN tablo2 ON tablo1.name = tablo2.name;

# Veri sorgula:
SELECT * FROM tablo1;
SELECT * FROM tablo2;
# Çıktı1:             Çıktı2:
# name                name
# Fâruk               Hasân
# Ahmed               Kâmil
```

SQL içinde NoSQL

- SQL (Structured Query Language) ilişkisel veritabanlarının ortak dilidir. İlişkisel veritabanları pek çok durumda ihtiyaçlara cevâp verebilen dayanıklı sistemlerdir; fakat verinin çok büyük ve parçalanabilir olmasının gerektiği durumlarda, tekil değil, çoğul verinin söz konusu olduğu durumlarda, "nesnelerin interneti" (IoT) cihazları gibi çok sık kayıt eklenen sistemlerin olduğu durumlarda ilişkisel veritabanları ihtiyaca cevâp verememektedir. Ayrıca bu tür durumlarda okuma kadar, belki okumadan daha fazla yazma hızı da çok önemlidir. İlişkisel veritabanları ise okumanın öne çıktığı sistemlerdir. Hattâ veritabanı indekslerinin içerdiği boşluktan fazla kayıt eklenmesi durumunda indeksler bozulur ve artık okuma performansı da düşer. İlişkisel olmayan veritabanlarında ise bu tür sorunlara çözüm gelmiştir; fakat oradaki yapı bütünlüğünün sağlandığı, yapı şemasının tek olduğu ilişkisel veritabanları gibi olmadığından pek çok menfî yönü vardır.

- Pek çok ihtiyaç için uygun çözüm olan ilişkisel veritabanlarının içerisinde liste, dizi gibi çoklu verileri tek sütunda tutma kabiliyyeti, ilişkisel olmayan yapıları da içerebilme kâbiliyyeti getirilmesi de artık ihtiyaç olmuştur. Buna çözüm olarak ilişkisel olmayan veritabanlarının temel yapı birimi olan `JSON` yapısında veri saklanması desteği MySQL ve PostgreSQL gibi veritabanları için getirilmiştir.
- `JSON` veri tipinde saklanan veriler `JSON` tipine uygun olarak sorgulanabilir, üzerinde silme ve tazeleme işlemleri yapılabilir.
- `JSON` veri tipi 5.7.8 sürümünden itibâren MySQL tarafından desteklenmektedir.
- `JSON` verileri hızlı aramayı da desteklemek için ikili (`BINARY`) formatta tutulur.
- Yaklaşık olarak `LONGTEXT` veri tipi kadar yer kaplar.
- `JSON` tipli bir sütun `NULL` değer kabûl etmektedir.
- `JSON` tipindeki sütunlar için varsayılan değer verme özelliği MySQL 8.0.13 sürümüyle sisteme eklenmiştir.
- `JSON` sütununa doğrudan indeks (`INDEX`) eklemek mümkün değildir; fakat bu sütundan çıkartılan veriler için fonksiyonel indeks veyâ indeks eklenebilir.
- `JSON` sütunu üzerinde sorgulama işlemi yapıldığında MySQL performansı arttırmak için fonksiyonel indeks ('functional index') yapısını kullanır.
- `JSON` tipindeki bir sütun için veri eklerken veriyi metîn biçiminde yazmamız gerekiyor; bu durumda ana metni tek tırnak içerisinde yazmak daha iyi olur; zîrâ çift tırnak kullandığımızda metîn içerisindeki her çift tırnaktan evvel kaçış karakteri kullanmamız gerekir.
- `JSON` tipindeki sütuna `JSON` ile uyuşmayan formatta bir veri girilemez.
- `JSON` veri tipiyle çalışmak demek, bol bol fonksiyon kullanmak demektir.

JSON Tipinde Bir Sütun Tanımlama

- Veri tipi yerine `JSON` yazmak kâfîdir:

```
CREATE TABLE veri(info JSON);
```

Veri Sorgulama

- `JSON` tipindeki veriyi doğrudan `JSON` metni olarak çekebiliriz:

```
SELECT * FROM veri;
```

- Gelen `JSON` metnini güzelleştirmek için `JSON_PRETTY()` fonksiyonu kullanılır. Bu fonksiyon içerisinde `JSON` metnini alır.

JSON verisinin anahtarlarını çekme

- Bilindiği üzere nesne biçimindeki `JSON` verisi anahtar - değer şeklinde özellikler barındırmaktadır. Bir `JSON` nesnesi içerisindeki anahtarların ismini çekmek için `JSON_KEYS()` fonksiyonu kullanılabilir.
- Eğer sütundaki JSON verisi dizi biçimindeyse `JSON_KEYS()` fonksiyonu o sütun için `NULL` değeri döndürür.

JSON verisi gezinme

- JSON verileriyle çalışırken bir ilişkisel olmayan (No-SQL) veritabanı deneyiminin sağlanması çok önemlidir. MySQL'de bunun için JSON fonksiyonları ve 'JSON path' isimli yapı kullanılır.
- Bu amaç için kullanılan en temel fonksiyonlardan bir tânesi `JSON_EXTRACT()`'tır.
- 'JSON path' ise bu fonksiyona parametre olarak verilen ve JSON tipindeki bir alanın alt alanlarını veyâ alt elemanlarını çekebilmek için kullanılan bir yapıdır.
- Fonksiyonunun temel yapısı: `JSON_EXTRACT(jsonVerisi, json_path)`
- JSON path çok basit bir yazım şeklidir. `$` işâreti mevcut JSON belgesini işâret eder; dizide alt elemana erişmek için belge işâretinden sonra köşeli parantez içerisinde indis (sıra numarası) belirtilir; indis sıfırdan başlar. Belge bir JSON nesnesi ise alt özelliklere nokta(`.`) işleciyle erişilir; alt özellik ismi yazılmalıdır. Birkaç misal:

  `$` : Mevcut belgeyi ifâde eder.

  `$[0]` : Belge bir JSON dizisi ise dizinin ilk elemanını ifâde eder, belge bir JSON nesnesi ise JSON nesnesinin tamâmını ifâde eder `$[*]` : Belge bir JSON dizisi ise dizinin tamâmı JSON metni biçiminde döndürülür.

  `$.name` : JSON nesnesinin 'name' özelliğini ifâde eder

  `$.notes[0]` : JSON nesnesinin 'notes' isimli dizisinin ilk elemanını ifâde eder. `$.*` : Nesne için tüm alt özellik değerlerini döndürür, dizi için NULL döndürür.

> ***NOT :*** Çift yıldız işâreti (`**`) JSON içerisindeki tüm alt JSON belgelerinin aranmasını ifâde eden özel bir simgedir.

JSON dizisi oluşturma

- Bir JSON dizisi oluşturmak için `JSON_ARRAY()` fonksiyonu kullanılabilir. Bu fonksiyon girdi olarak dizi elemanlarını alır ve geriye JSON dizisini temsîl eden metni döndürür:

```
SELECT JSON_ARRAY('eleman1', 2, 'eleman3') AS diziMetni;
# Çıktı : diziMetni : ["eleman1", 2, "eleman3"]
```

JSON nesnesi oluşturma

- JSON nesnesi oluşturmak için `JSON_OBJECT()` fonksiyonu kullanılabilir.
- Fonksiyon girdi olarak anahtar ve değerleri virgülle ayrılmış biçimde alır:

```
SELECT JSON_OBJECT(
    'isim', 'Ahmed',
    'soyisim', 'Arslanoğlu',
    'numara', 324) AS metin;
# metin : {"isim": "Ahmed", "numara": 324, "soyisim": "Arslanoğlu"}
```

JSON belgesi içerisinde aranan yolun olup, olmadığını kontrol etme

- Bunun için `JSON_CONTAINS_PATH()` fonksiyonu kullanılır.
- Fonksiyonun parametreleri:
    1. İlk parametre içersinde bir yol ('path') aranan JSON metnidir.
    2. İkinci parametre verilen yolun / yolların JSON belgesi içerisinde en az bir tânesinin mi arandığı, yoksa tamâmının mı arandığı bilgisidir. Bu parametre iki değer alabilir : `'one'` veyâ `'all'`. Bu parametreler metîn olarak yazılan parametrelerdir, yanî tek veyâ çift tırnak içerisinde yazmalısınız. `'one'` değeri yazılırsa, verilen yollardan en az birisi belge içerisinde varsa geriye `1` döndürülür, diğer durumda `0` döndürülür. `'all'` değeri yazılırsa verilen yolların tümünün JSON belgesi içerisinde olduğu durumda `1`, diğer durumda `0` döndürülür.
    3. Belge içerisinde aranmak istenen JSON yolu verilir.
    4. Belge içerisinde aranmak istenen 2. JSON yolu verilir, daha fazla varsa, 5. parametre olarak, 6. parametre olarak bu parametre sayısı arttırılabilir.

- Kullanımı hakkında misal:

```
SELECT JSON_CONTAINS_PATH(
    '{"osName" : "Kali", "ram":16, "caches" : [2, 16, 32]}',
    'one',
    '$.caches[2]'
) AS isL3Supported;
# Çıktı: isL3Supported : 1
```

JSON belgesinde aranan belgenin olup, olmadığını kontrol etme

- `JSON_CONTAINS(json, arananJson [, path])` yöntemi kullanılır.
- İlk parametre JSON metnidir, ikinci parametre aranan JSON metnidir.
- Üçüncü parametre JSON yolunu belirtmek için kullanılır, zorunlu değildir.

```
SELECT JSON_CONTAINS(
'{"brand":"Samsung", "ram": 12,
"details": {"frontCamera":64, "tripleBackCamera":true}}',
'{"brand":"Samsung"}'
);# 1 döndürür

SELECT JSON_CONTAINS(
    '{"brand":"Samsung", "ram": 12,
    "details": {"frontCamera":64, "tripleBackCamera":true}}',
    '{"tripleBackCamera":true}', '$.details'
    );
```

- Fonksiyon, aranan JSON varsa `1`, aksi hâlde `0` döndürür.

Diğer JSON fonksiyonları

- `JSON_OVERLAPS(json1, json2)` : Aldığı json1 ve json2 belgelerini karşılaştırır. Belgeler JSON nesnesi ise en az bir ortak özellik - değer olması durumunda; belgeler dizi ise en az bir ortak eleman olması durumunda `1` döndürür, diğer durumda `0` döndürür.
- `JSON_SEARCH(json, 'one' | 'all', aranan_deger [, escape_character])` : Verilen `json` metni içerisinde `'one'` veyâ `'all'` parametresine bağlı olarak bir veyâ tüm `aranan_deger` ifâdelerinin json yollarını döndürür. `escape_character` parametresi isteğe bağlı olarak belirtilen kaçış karakteridir. `aranan_deger` ifâdesi metîn olmalıdır. Bu metîn JSON nesnesinin değerlerinde ve dizi elemanlarında aranır. Aranan ifâdeyi kısmî belirtebilmek için 'joker karakter' ('wildcard') denen karakterler kullanılabilir. Bunlar `%`, `_`

karakterleridir. `%` karakteri birden fazla karakteri simgeleyen bir ön karakterdir; `_` karakteri ise tek bir karakteri simgeliyor. Birkaç misal: `Ahmed%` : 'Ahmed' ismiyle başlayan tüm metînlerle eşleşir. `Ahme_` : 'Ahme' ifâdesinden sonra gelen tek bir karakterle eşleşir. Kaçış karakteri belirtilmezse varsayılan olarak ters ayraçtır (`\`).

- `JSON_INSERT(json, path, value [, path, value, ...])` : 'path' parametresiyle verilen JSON yoluna verilen değer eklenir; eğer eklenmek istenen değerler sonraki parametre itibâriyle başlanarak 'path, value' şeklinde verilebilir.'json' parametresi verinin eklenmesi istenen JSON metnidir.

- `JSON_REPLACE(json, path, value [, path, value, ...])` : Verilen JSON yolundaki eleman verilen değerle değiştirilir.

- `JSON_SET(json, path, value [, path, value, ...])` : Verilen JSON yolunda bir eleman varsa değiştirilir, yoksa eklenir(?).

- `JSON_REMOVE(json, path [, path, ...])` : Verilen JSON yolundaki eleman / özellik kaldırılır, diğerlerinde olduğu gibi birden fazla JSON yolu verilebilir.

- `JSON_MERGE_PATCH(jsonBelgesi1, jsonBelgesi2 [, jsonBelgesi3, ...])` : Verilen JSON belgelerini birleştirir. Metînlerde aynı anahtar için farklı JSON nesneleri varsa, bunlar birleştirilir; fakat tekrârlayan anahtarlar JSON nesneleri değil de sayı, metîn gibi değerler ise sonra gelen belgedeki JSON anahtarı belirleyici olur. Ayrıca, bir belgede bir JSON anahtarının karşısına `NULL` yazıldıysa, bu anahtar silinir. Verilen metînler JSON formatına uygun değilse, hatâ alınır. İlk JSON metni nesne değilse ve ikincisi dizi ise, geriye ikinci eleman döndürülür; ilk JSON metni dizi ve ikincisi nesne ise, geriye ikinci eleman döndürülür.

```
SELECT JSON_MERGE_PATCH(
'{"name":"Akif"}',
'{"surname":"SOLAK"}'
);# Sonuç : '{"name":"Akif", "surname":"SOLAK"}'
SELECT JSON_MERGE_PATCH(
'{"name":"Akif", "surname":"SOLAK"}',
'{"surname":null}'
);# Sonuç : '{"name":"Akif"}'
```

- `JSON_MERGE_PRESERVE(json1, json2 [, json3, ...])` : `JSON_MERGE_PATCH` ile benzer kullanımdadır; ondan farkı dizi ile nesneyi birleştirebiliyor olmasıdır. Misal, birinci JSON belgesi nesne ve ikinci JSON belgesi dizi ise, ikinci eleman birinci parametreyle verilen dizinin içerisine eklenir.

- `JSON_ARRAY_APPEND(json, path, value [, path2, value2, ...])` : Verilen JSON belgesine verilen adrese göre verilen değeri ekler. Birden fazla değer verilecekse, her bir değerden evvel adres belirtmeyi unutmayın.

```
SELECT JSON_ARRAY_APPEND(
'[11, 14]', '$', 3, '$[1]', 5
);# Sonuç: [11, [14, 5], 3]
```

- `JSON_VALUE(json, path [RETURNING veriTipi] [{NULL | ERROR | DEFAULT deger} ON EMPTY] [{NULL | ERROR | DEFAULT deger} ON ERROR])` : Verilen JSON metni içinden verilen yoldaki nesneyi getirir; çoklu nesne seçimi için uygun değildir; fakat bu fonksiyonun kullanışlı olan yanı gelen verinin dönüştürülebileceği veri tipinin belirtilebilmesidir. Eğer dönüş tipi belirtilmezse veri `VARCHAR(512)` tipinde getirilir. Fakat dikkat edilmesi gereken bir şey vardır, eğer dönüş tipinin tamsayı olması isteniyorsa `INT` ifâdesi yazılmamalı, sadece `SIGNED` veyâ `UNSIGNED` yazılmalıdır. Veri tipinden sonra, fonksiyonun verilen JSON yolunun bir elemanı işâret etmediği durumunda nasıl davranacağı ve hatâ durumunda nasıl davranacağını belirtilebilir. Fonksiyon MySQL 8.0.21 sürümünden itibâren kullanıma sunulmuştur.

```
SELECT JSON_VALUE('[2, 16.3, 32]',
'$[1]' RETURNING DOUBLE DEFAULT -1 ON EMPTY NULL ON ERROR
) AS jsonIcindenCekilenVeri;
# Dizinin ikinci elemanını DOUBLE olarak çeker;
# Eğer ilgili adreste veri yoksa, -1 döndürür
# JSON adresi uygun formatta değilse NULL döndürür
```

Diğer bilgiler

- Bir değerin bir JSON dizisi içerisinde olup, olmadığını öğrenmek için `MEMBER OF` işlecini kullanabilirsiniz; ilgili değer dizi içerisinde varsa `1`, yoksa `0` döndürülür:

```
SELECT 'Ahmed' MEMBER OF('["Ahmed", "Mehmed", "Fâruk"]');
```

GENEL YAPIYI ANLAMAK İÇİN BÂZI NOTLAR

- Bilhassa `GROUP BY` deyiminin çalışması ve `GROUP BY` deyimi kullanıldığında `SELECT` deyiminin de kullanılmasının gerektiği bilgisi biraz kafa karıştırıcı olabilir.
- MySQL'in hangi deyimi ne zamân ele aldığını bilmek bu konuda yardımcı olabilir, ufkunuzu açabilir:

FROM -> WHERE -> GROUP BY -> HAVING -> SELECT -> DISTINCT -> ORDER BY -> LIMIT

İŞLEMİN UYGULANDIĞI SATIR SAYISINI SINIRLANDIRMA

- Bir kayıt çekme sorgusu sonucu çok sayıda kayıt gelebilir. En üst sıradaki kaydı almak, belirtilen şarta uygun ilk n sayıda kaydı almak, sadece tek bir kaydı çekerek tablonun neye benzediğini gözlemlemek gibi amaçlarla sorgumuza bir limit ekleyebiliriz. Bu limit yapılan işlemin kaç kayıt için geçerli olduğunu belirtmek için kullanılır ve bu özellik güncelleme, veri ekleme ve veri silme için de uygulanabilir:

```
SELECT * FROM `articles` ORDER BY 'id' LIMIT 5;
#'articles' tablosundaki 'id' değeri en büyük 5 satır getirilir.
```

- **! `LIMIT` komutu komut cümlesinin en sonuna yazılmalıdır.**

TABLO KİLİTLEME ('TABLE LOCKING')

- Eşzamânlı ve uzun süren işlemlerde veri bütünlüğünün sağlanması için tablo, oturum ('session') tarafından kilitlenebilir.
- Tablo kilitleme, tablonun başka oturumlar için belirlenen modda erişime kapalı olmasıdır.
- Kilitlenen bir tablo diğer oturumlar tarafından veri değişikliğine ve varsayılan olarak okumaya kapalıdır.
- MySQL'de 2 tablo kilitleme modu vardır : `READ` ve `WRITE`
- `READ` kilidi tabloyu başkalarının yazma erişimine kapatır, `READ` kilidi okuma erişimini kısıtlamaz, hem kilidi oluşturan oturumda, hem de başka oturumlarda tablo verisi okunabilir. Yazma erişimi ise, tüm oturumlar için kapatılır; kilidi yapan oturum tabloya veri yazmak istediğinde hatâ alır, diğer tablolar ise işlem zamân aşımına uğramadığı sürece kuyrukta bekler. Tablo kilidi ortadan kalkınca bu bekleyen işlemler işlenir.
- `WRITE` kilidi ise, tabloyu diğer oturumlar içintamâmen kilitler, diğer oturumlar tablodaki veriyi ne okuyabilir, ne de değiştirebilir; fakat kilidi oluşturan oturum tabloyu sorgulayabilir ve tabloya yeni veri yazabilir.
- Tablo kilitleme söz dizimi:

```
LOCK TABLES tabloIsmi READ | WRITE
# | işâreti seçenekleri göstermek için kullanılır : READ veyâ WRITE
```

- Kilit oluşturulduktan sonra bir şekilde bağlantı koparsa, MySQL kilidi kaldırır.
- `READ` kilidi diğer oturumların tabloyu `READ` ile kilitlemesini engellemez, `WRITE` ile kilitlemesini engeller.
- `WRITE` kilidi ise, diğer oturumların tablo üzerindeki tüm yetkisini iptâl ettiğinden tabloyu kilitlemelerini engeller.
- Oturum içerisinde kilitlenmiş tüm tabloların kilidini açmak için `UNLOCK TABLES;` deyimi kullanılır; müşahhas bir tablonun kilidini açmak söz konusu değildir.
- Birden fazla tabloyu kilitlemek için şu söz dizimince kod yazılır:

```
LOCK TABLES tablo READ | WRITE,
            tablo2 READ | WRITE
```

> ***NOT :*** MySQL'de `LOCK TABLES` deyimi çalıştırıldığında oturum tarafından daha önce kilitlenen tüm tabloların kilidi açılır ve yeni kilitler uygulanır. Yanî 2 tabloyu kilitlemek istiyorsanız, bunları ayrı ayrı kilitleyemezsiniz; tek bir SQL komutuyla yukarıdaki gibi birden fazla tablo belirterek kilitlemelisiniz.

VERİTABANI NESNELERİ ÜZERİNDE DEĞİŞİKLİK

- Veritabanında nesneler (tablolar, veritabanı, saklı yordamlar, vs) `CREATE` komutu ile oluşturulur, `ALTER` komutuyla değiştirilir ve `DROP` komutuyla silinir. Bu üç komuta 'veri tanımlama dili' ('data definition language' (DDL)) denir.
- Veritabanı nesneleri üzerinde nasıl değişiklik yapıldığına yakından bakalım inşâAllâh..

Tablolar Üzerinde Değişiklik Yapma

- Tablo yapısı üzerinde değişiklik yapmak için önce `ALTER TABLE tabloIsmi` yazılır.

1) Sütun ekleme

- Yeni sütun eklemek için söz dizimi:

```
ALTER TABLE tabloIsmi
ADD
    yeniSutun yeniSutunTanimi
    [{FIRST} | {AFTER komsuSutunIsmi}]
```

- Yukarıdaki ifâdede yeni sütun tanımı en basit olarak sütun veri tipini gerektirir; arkasından `NOT NULL`, `UNIQUE` gibi kısıtlar yazılmak isteniyorsa, yazılabilir.
- Sütunun ekleneceği yer tercîhîdir (koddaki en alt satır bunu ifâde ediyor). `FIRST` yeni sütunun tablonun ilk sütunu olması için kullanılırken, `AFTER komsuSutunIsmi` ise yeni sütunun tablonun belirtilen sütundan sonra olması için kullanılır. Eğer mevkî belirtilmezse, sütun tablonun en sonuna eklenir.
- Tek deyimle, birden fazla sütun eklenebilir:

```
ALTER TABLE tabloIsmi
    ADD COLUMN yeni yeniSutunTanimi [{FIRST} | {AFTER komsuSutunIsmi}],
    ADD COLUMN yeni2 yeni2Tanimi [{FIRST} | {AFTER komsuSutunIsmi}]
```

- Eğer bir otomatik artan bir birincil anahtar sütunu eklerseniz, MySQL önceki verileri sizin için numaralandırır.

2) Tabloyu yeniden isimlendirme

- Bunun için şu sözdizimi tâkip edilir:

```
ALTER TABLE tabloIsmi
    RENAME TO tablonunYeniIsmi;
```

- Daha kısa olarak şu şekilde yapılabilir:

```
RENAME TABLE tabloIsmi TO tablonunYeniIsmi
```

- `RENAME TABLE` deyimi geçici tablo ('temporary table')ları isimlendirmede çalışmaz; Bunun için `ALTER TABLE .. RENAME TO` kullanılmalıdır.
- `RENAME TABLE` ifâdesiyle birden fazla tablonun ismi tek komutla değiştirilebilir:

```
RENAME TABLE
    tabloIsmi TO tablonunYeniIsmi,
    tablo2Ismi TO ikinciTablonunYeniIsmi,
    ...;
```

3) Sütun tanımını değiştirme

- Bir sütunun tanımının değiştirilmesi isteniyorsa şu sözdizimi uygulanır:

```
ALTER TABLE tabloIsmi
    MODIFY sutunIsmi sutununYeniTanimi
    [{FIRST} | {AFTER komsuSutunIsmi}]
```

- Sütun tanımını değiştirmek için yazılan kod, mevcut verilerin tabloda kalmasına engel teşkil ediyorsa, kodunuz çalışmaz. Misal, `NOT NULL` kısıtı eklemeye çalıştığınız sütunda `NULL` veri varsa hatâ alırsınız.
- `MODIFY` deyimiyle sütunun;
    1. Veri tipi
    2. Sütunun tablodaki mevkîsi
    3. Varsayılan değer kısıtı, NOT NULL kısıtı gibi kısıtlar değiştirilebilir.
- `MODIFY` deyimiyle sütunun;
    1. İsmi değiştirilemez.
    2. `CHECK` kısıtı eklenemiyor, kaldırılamıyor (Kısıtlar için farklı bir deyim var).
- Bir sütunun veri tipi değiştilmek istendğinde, mevcut veriler buna uygun değilse, hatâ oluşur. Misal, 33000 değerini içeren bir satırın olduğu `INTEGER` tipindeki bir sütunun veri tipi `SMALLINT` olarak değiştirilmek istendiğinde, işlem başarısızlıkla sonuçlandırılır; çünkü 33000, `SMALLINT` tipinin kapsamı dışındadır.
- Yanî sütun üzerinde değişiklik yapılabilmesi için tablodaki mevcut verinin buna uygun olması gerekir; öte taraftan sütun üzerinde değişiklik yapılmasını engelleyen farklı bir durum daha vardır, biraz sonraki 'Önemli bir mesele' başlığına bakınız.

4) Sütun ismini değiştirme

- Bunun için şu sözdizimi tâkip edilir:

```
ALTER TABLE tabloIsmi
    CHANGE [COLUMN] sutunIsmi sutununYeniIsmi sutunTanimi
    [{FIRST} | {AFTER komsuSutunIsmi}]
```

- Sütun ismini değiştirmeyle ilgili bâzı meseleler var. Bir genel kısıt tarafından kullanılan sütun silinemez ve değiştirilemez.

Önemli bir mesele:

- Yukarıda, `ALTER TABLE .. MODIFY` deyimiyle `CHECK` kısıtının eklenemediğini, fakat diğer bâzı kısıtların eklenebildiğini görmüştük. Bunun sebebi, `CHECK` kısıtı bir genel kısıttır. Tablo bazında `CHECK` kısıtı zâten birden fazla tabloyu kapsar; fakat sütun bazındaki `CHECK` kısıtı da genel bir kısıttır; bu sebeple bir `CHECK` kısıtı tarafından kullanılan bir sütun silinemez, düşürülemez; fakat `NOT NULL` , `UNIQUE` , `DEFAULT` gibi kapsamı sadece o sütun olan kısıtlar eklenebilir, silinebilir. Bunun için ikisi de kullanılabilir: `ALTER TABLE .. MODIFY` veyâ `ALTER TABLE .. CHANGE`

4) Sütun silme

- Bir sütunu silebilmek için o sütun üzerinde genel bir kısıtın olmaması gerekmektedir.
- Öte yandan bir sütunun silinmesi beklenmedik yan etkiler doğurabilir, buna birazdan değineceğiz inşâAllâh.
- Sütun silindiğinde o sütundaki tüm verilerin kaybolacağını gözden kaçırmayınız.
- Söz dizimi:

```
ALTER TABLE DROP [COLUMN] sutunIsmi;
```

Sütun ve tablo üzerindeki değişikliklerin olası yan etkileri

- Sütunun silinmesi beklenmedik etkilere yol açabilir.

1) İndekslerin yapısının değişmesi

- Misal, iki sütunu kullanan bir `UNIQUE INDEX` olduğunu düşünelim. Bu durum sütunlardan birisinin silinmesi için engel değildir; fakat bir sütunu sildiğinizde, sütun indeksten de silinir ve indeks sadece bir sütun üzerinde münferid veri olmasını sağlayan bir indeks hâline gelir. Eğer tablodaki veriniz buna uygun değilse, sütunu silerken hatâ alırsınız, tablodaki veriniz buna uygunsa, veritabanı tasarımınız değişmiş olur ve fark edemeyebilirsiniz.

- Yukarıdaki durumu örneklendirmek istersek; Bir yazarın birden fazla kitâbının olduğu ve farklı yazarların aynı isimde kitâplarının olabileceği bir veritabanınız olsun. Bunun için şu kodu çalıştırabilirsiniz:

```
CREATE TABLE library(
    writer VARCHAR(100),
    bookName VARCHAR(50),
    UNIQUE INDEX munferid(writer, bookName));
```

- Şimdi tabloya rastgele isimlerde veriler ekleyelim:

```
INSERT INTO library VALUES
    ('Ahmed Yılmaz', 'Bir sabah doğuyor'),
    ('Ahmed Yılmaz', 'Umut fidanları büyüyor'),
    ('Mehmed Soylu', 'Usûl');
```

- Şimdi tablodaki yazar ('writer') alanını kaldırırsak;

```
ALTER TABLE library DROP writer;# İşlem başarılı
```

- Tabloya son bir veri daha eklemek isteyelim:

```
INSERT INTO library (bookName) VALUES ('Usûl');
# Hatâ aldık : Error Code: 1062.
# Duplicate entry 'Usûl' for key 'library.munferid'
```

- Bunun sebebi, sütunu sildiğimizde sütunun münferid indeks (`UNIQUE INDEX`)ten de silinmesi sonucu yeni indeks yapımız şu oldu : `UNIQUE INDEX(bookName)` Bu indeks de verilerin eklenmesini engelliyor.
- Eğer en son eklemek istediğimiz veriyi tabloya daha önce ekleseydik, sütunu silemezdik.

2) Tablo ismi değiştirildiğinde ilgili saklı yordamın, 'view'ın ve dış anahtarların elle değiştirilmesi gerekir

- Eğer, başka bir tablo tarafından dış anahtarla bağlanmış bir tablonun ismini değiştirirseniz, bu dış anahtar çalışmayacaktır.
- Benzer şekilde view ve saklı yordamların da değiştirilmesi gerekmektedir.
- Ezcümle, bir sütunun yapısını değiştirmek veyâ sütunu silmek farklı sebepleri doğurabilecek eylemlerdir. Bunu yaparken tablonuzun mevcut verisini, tasarımını ve şu

yapıları dikkate alın : `INDEX` , `FOREIGN KEY` , `VİEW` , `STORED PROCEDURE`

Nesneleri Silme

- Veritabanı nesneleri `DROP` komutuyla silinir.

1) Tablo Silme

- Tablo silmenin söz dizimi şöyledir:

```
DROP [TEMPORARY] TABLE [IF EXISTS]
tabloIsmi [, tablo2Ismi, tablo3 Ismi, ...]
[RESTRICT | CASCADE]
```

- `TEMPORARY` ifâdesi tablo geçici tablo olduğu durumda eklenmesi tavsiye edilen ifâdedir, yazılması zorunlu değildir; fakat geçici tablo yerine yanlışlıkla gerçek tablo silmenizi önlemek için geçici tablo silerken kullanılması tavsiye edilir.
- `IF EXISTS` ifâdesi tablonun varlığından emîn olunmadığında kullanılabilir. Bu ifâdeyle tablo silmek istediğinizde tablo varsa silinir, eğer tablo yoksa hatâ yerine uyarı alınır; bu, MySQL'i API ile kullandığınız durumlarda uygulamanın hatâ vermeden çalışması gibi ihtiyaçlarınız için kullanışlı olabilir.
- Aynı anda birden fazla tablo silinmek isteniyorsa, tablo isimleri aralarına virgül konarak yazılabilir. Eğer tablolardan birisi bile silinemiyorsa (tablo yoksa veyâ bunu yapmaya iznimiz yoksa), hatâ alınır, işlem sonlandırılır ve hiçbir tablo silinmemiş olur. Birden fazla tabloyu aynı anda silerken `IF EXISTS` ifâdesini kullanarak bu sorunu çözebilirsiniz.
- `RESTRICT` ve `CASCADE` ifâdeleri ise, MySQL'in sonraki sürümleri için ayrılmış ifâdelermiş.
- Bir tablo başka bir tablo tarafından dış anahtarla bağlıysa silinemez.

Saklı yordam, görünüm, tetikleyici, indeks vd. veritabanı nesnelerinin silinmesi ilgili konu altında anlatıldığından burada tekrar değinmiyorum.

GÖRÜNÜM ('VIEW')

- Görünümler bir sorgunun tablo gibi sorgulanabilmesini sağlayan yapılardır.
- Bu görünüm sorguları veritabanında kayıtlıdır ve sadece bir isimle çağrılabilir; yanî bir sorgunun isimlendirilerek, sadece bir isimle çalışmasını sağlar
- View (görünüm) yerine, 'virtual table' (sanal tablo) da denir.

- Görünümler çok kullanışlı olabilir. Bâzı kullanım durumları şöyledir:

    1. Birden fazla tablodan seçili sütunların getirilmesi gibi sorguların tek bir deyimle çalıştırılabilmesini sağlar
    2. Bir sorgudan dönen verinin farklı dil, format ve yapılarla getirilmesini sağlar
    3. Kapsamlı bir sorgunun özelleştirilmesini sağlar
    4. Sorgu metînlerinin istemci tarafında tutulması sebebiyle güvenlik zafiyetinin ve 'SQL injection' saldırılarının önüne geçilmesi için destek sağlar
    5. Sorgunun gerektirdiği izinlerin istemcilere verilmemesi sayesinde yüksek bir güvenlik desteği sağlar
    6. Uzun sorgu metînlerinin ağ üzerinden gitmesine engel olunduğu için iletişimde hız, veri kullanımı tasarrufu ve bant genişliği sağlar
    7. Karmaşık sorguların yanlış yazılması gibi hatâların önüne geçilmesini sağlar.
    8. Sorguların değiştirilmesi gerektiği durumlarda, sorgunun her istemci için (mobil uygulama, web uygulaması vs.) değiştirilmesinin önüne geçmeyi sağlar
    9. Veritabanı kullanımı hakkındaki derin bilgi olmayan çalışanın istediği sorguyu yapabilmesi noktasında esneklik sağlar
    10. Tabloda bir değişiklik yapıldığında, misal ismi değiştiğinde, istemcinin hatâ vermemesi için kullanılabilir
    11. Gerekli şartlar sağlandığında, görünüm üzerinden veri silme, veri tazeleme işlemleri yapılabildiğinden, karmaşık analizler sonucu silinmesi istenen verinin kolayca silinmesi gibi faydalar elde edilebilir; bu, hem güvenlik demek, hem de bu işlemler için yüksek SQL bilgisine bağımlı olmamak demektir.

Görünüm Oluşturma veyâ Vâr Olan Görünümü Değiştirme

- Görünüm oluşturma için söz dizimi:

```
CREATE [OR REPLACE] VIEW gorunumIsmi[(secilenSutunIsimleri)]
AS
    sorguCumlesi;
```

- Bir görünüm bir tabloyla aynı isimde olamaz.

- Aynı isimde birden fazla görünüm olamaz.
- `OR REPLACE` deyimi görünüm varsa, vâr olan görünümün değiştirilmesini, yoksa yeni görünüm oluşturulmasını sağlayan ihtiyârî (tercihi) bir deyimdir.
- Kısa yazım özellikleri ve karakter seti hatâsı oluşma ihtimalinin olmaması gibi sebeplerden ötürü veritabanı nesneleri genellikle İngilizce yazılır. Sorgu cümlesi içerisinde isimlendirme yapılabildiği gibi, sorgu cümlesiyle gelen sütunlar görünüm isminden hemen sonra yazılabilen 'secilenSutunIsimleri' ifâdesinin olduğu yerde de isimlendirilebilir. İkincisi kullanılırken dikkat edilmesi gereken şey, verilen sütun isimlerinin sorguyla gelen sütun sayısıyla ve sırasıyla uyumlu olması gerektiğidir.
- `SHOW CREATE VIEW gorunumIsmi` deyimi kullanılarak bir görünümün oluşturulması için gerekli kodlar elde edilebilir.
- `ALTER VIEW gorunumIsmi AS ...` deyimiyle de görünüm ('view') değiştirilebilir.

Görünümü Silme

- Oldukça basittir:

```
DROP VIEW [IF EXISTS] gorunumIsmi;
```

Görünüm ('view') Üzerinden Veri Değişikliği Yapma

- Her şeyden önce, bir görünüm üzerinden veri silme, veri güncelleme gibi işlemlerin yapılabilmesi için bu görünümün sağlaması gereken ağır şartların olduğunu bilelim: Sorgu içerisinde;
    1. toplam (aggregate) fonksiyonları kullanılmamalı
    2. `DISTINCT` deyimi kullanılmamalı
    3. `GROUP BY` deyimi kullanılmamalı
    4. Dolayısıyla `HAVING` deyimi de kullanılamaz
    5. Sol birleştirme (`LEFT JOIN`) ve dışlayarak birleştirme (`OUTER JOIN`) olmamalı
    6. `UNION` ve `UNION ALL` deyimleri olmamalı
    7. `SELECT` deyimi içerisinde iç sorgu (`SUBQUERY`) olmamalı

8. `WHERE` deyimi içerisinde sorgunun çekildiği esas tabloya (`FROM` ile belirtilen) atıfta bulunan bir iç sorgu (`SUBQUERY`) olmamalı
9. Sâbit değerler yer almamalı
10. Temel tablonun bir sütununa birden fazla atıf olmamalı
11. Bu şartları sağlamayan bir görünüme atıf olmamalı

- Yukarıdaki şartlardan anlaşılabileceği üzere sorgumuz oldukça yalın bir şekilde sütun isimlerinin belirtildiği, `AS` ile takma isim kullanılabildiği, `ORDER BY` ile sıralama içerebilen, ana tabloya atıfta bulunan bir iç sorgu içermeyen `WHERE` ile filtreleme yapılabildiği bir sorgu olmalıdır.
- Bu şartları sağlayan görünüme tazelenebilir ('is_updatable') görünüm denir.
- Görünüm üzerinden veri silme, değiştirme işlemi tablo üzerinden yapıldığı gibidir.
- Bir görünümün tazelenebilir (is_updatable) olup, olmadığı şu şekilde sorgulanabilir:

```
SELECT table_name, is_updatable FROM information_schema.views
WHERE table_schema = veritabaniIsmi;
```

Görünüm Üzerinden İşlem Yapmayı Güvenli Hâle Getirme

- Bir görünüm aracılığıyla veri ekleme ve tazeleme işlemi yaparken, o görünüm ile gelmeyen satırların verisinin değiştirilmesini engellemek için bir yapı sunulmaktadır: `WITH CHECK OPTION`
- Bu, bir görünümü sunduğumuz bir birimin tablo üzerindeki etkisini sınırlandırmak için harika bir çözümdür. Çünkü, tabloya doğrudan ekleme - yazma izni verilmiyor, fakat bununla beraber istemcinin görünüm üzerinden bulabildiği geniş yazma imkanı da kısıtlanıyor.
- Bir görünüm içerisinden başka bir görünüm çağrılabildiğinden dolayı, bu ifâdenin veri tutarlılığı açısından kullanılması tavsiye edilir.
- Misal, şûbe ve işçi bilgilerinin tutulduğu iki tablodan oluşan bir veritabanında, şûbeler için görünüm hâzırladığımızı ve şûbelerin çalışan kaydı oluşturabildiğini düşünelim:

```sql
CREATE DATABASE resources;# Veritabanını oluşturalım
USE resources;# Veritabanını seçelim
# Şubeleri ve işçileri tutmak için iki tablo oluşturalım
CREATE TABLE substations(
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(50) UNIQUE NOT NULL);
CREATE TABLE workers(
    id int PRIMARY KEY AUTO_INCREMENT,
    firstName VARCHAR(50),
    lastName VARCHAR(50),
    substationId int,
    FOREIGN KEY substationKey(substationId)
        REFERENCES substations(id)  ON DELETE SET NULL);

# Birkaç veri ekleyelim:
INSERT INTO substations(name) VALUES
    ('İstanbul'), ('Bursa');

INSERT INTO workers(firstName, lastName, substationId) VALUES
    ('Ahmed', 'Âlimoğlu',
        (SELECT id FROM substations WHERE name = 'Bursa')),
    ('Ömer Fâruk', 'Çalışkan',
        (SELECT id FROM substations WHERE name = 'İstanbul')),
    ('Ahmed Kürşat', 'Kahramanoğlu',
        (SELECT id FROM substations WHERE name = 'İstanbul'));

# Verilere bakalım:
SELECT * FROM workers;
#     id    firstName        lastName        substationId
#      1    Ahmed            Âlimoğlu        2
#      2    Ömer Fâruk       Çalışkan        1
#      3    Ahme Kürşat      Kahramanoğlu    1
SELECT * FROM substations;
#     id    name
#      2    Bursa
#      1    İstanbul
# Görünümü oluşturalım:
CREATE OR REPLACE VIEW myWorkersBursa
AS
    SELECT w.firstName, w.lastName, w.substationId, subSt.name AS substationName
    FROM workers AS w
    INNER JOIN substations AS subSt ON w.substationId = subSt.id
    WHERE subSt.name = 'Bursa';

# Bu görünümün tazelenebilir mi ('IS_UPDATABLE')?
SELECT TABLE_NAME, IS_UPDATABLE
    FROM INFORMATION_SCHEMA.VIEWS
```

```
    WHERE TABLE_SCHEMA = 'resources'
    AND TABLE_NAME = 'myWorkersBursa';
#    TABLE_NAME        IS_UPDATABLE
#    myworkersbursa       YES
# Evet, görünümümüz veri değişikilğini destekliyor.
SELECT * FROM myWorkersBursa;
# firstName        lastName    substationId    substationName
# Ahmed               Âlimoğlu       2               Bursa

# 'Bursa' şûbesi tarafından 'Bursa' şûbesine yeni birisi alınsın:
INSERT INTO myWorkersBursa(firstName, lastName, substationId)
VALUES ('Hasân', 'Akkoyunlu',
    (SELECT id FROM substations WHERE name = 'Bursa'));
# Yeni veriler şu şekilde:
SELECT * FROM myWorkersBursa;
# firstName        lastName    substationId    substationName
# Ahmed            Âlimoğlu    2               Bursa
# Hasân            Akkoyunlu   2               Bursa
# 'Bursa' şûbesi 'İstanbul' şûbesine yeni birisi almak isterse:
INSERT INTO myWorkersBursa(firstName, lastName, substationId)
VALUES ('Mert', 'Yılmaz',
    (SELECT id FROM substations WHERE name = 'İstanbul'));

# İşlem başarılı, İstanbul şûbesine yeni bir kişi eklendi.
SELECT * FROM workers
    WHERE substationId = (SELECT id FROM substations
                          WHERE name = 'İstanbul');
# id    firstName        lastName        substationId
#  2    Ömer Fâruk       Çalışkan         1
#  3    Ahmed Kürşat     Kahramanoğlu     1
#  5    Mert             Yılmaz           1
# Biz, her şûbenin sadece kendi çalışanlarını belirlemesine
# yönelik bir kısıt koymak istiyoruz.
# O zamân görünümümüze 'WITH CHECK OPTION' ifâdesini ekleyelim:
CREATE OR REPLACE VIEW myWorkersBursa
AS
    SELECT w.firstName,
        w.lastName, w.substationId, subSt.name AS substationName
    FROM workers AS w
    INNER JOIN substations AS subSt
        ON w.substationId = subSt.id
    WHERE subSt.name = 'Bursa'
    WITH CHECK OPTION;

# Bursa şûbesi için hâzırlanan myWorkersBursa görünümü
# üzerinden İstanbul şûbesine işçi eklenmek istenirse;
INSERT INTO myWorkersBursa(firstName, lastName, substationId)
VALUES ('Mert', 'Yılmaz', (SELECT id FROM substations
```

```
                            WHERE name = 'İstanbul'));
# Hatâ alınır : Error Code: 1369.
# CHECK OPTION failed 'resources.myworkersbursa'
DROP DATABASE resources;
```

WITH CHECK OPTION ifâdesini özelleştirme

- ..

Veritabanındaki Tüm Görünümleri Elde Etme

- Veritabanı sunucusundaki tüm görünümlerin ismini öğrenmek veyâ belli bir desene bağlı olarak görünümleri çekmek için şu deyim kullanılabilir:

```
SHOW FULL TABLES FROM sys WHERE Table_Type = 'VIEW';
```

- Belli bir veritabanındakileri öğrenmek için şu söz dizimi doldurulmalıdır:

```
SHOW FULL TABLES {FROM | IN} veritabanıIsmi
WHERE Table_Type = 'VIEW';
```

- `FROM` yerine `IN` deyimi kullanılabilir.
- Bu veri `INFORMATION_SCHEMA.VIEWS` içerisinden de çekilebilir.

Görünüm İsmini Değiştirme

- Tablo ismini değiştirmeyle aynıdır:

```
RENAME TABLE gorunumunIsmi TO gorunumunYeniIsmi;
```

İNDEKSLER (INDEXES)

- İndeksler, verilerin sıralamasını gösteren bir veri yapısıdır.
- Misal, bir sözlükteki kelîmelerin alfabetik sıraya göre dizilmiş olması bir indekstir.
- Tıpkı, sözlükteki gibi veriler de bir sütuna veyâ birden fazla sütüna göre sıralanabilirler.
- Elbette gerçek sıralama bir tânedir ve buna `CLUSTERED INDEX` denir.

- MySQL'de verilerin gerçek sıralamasını belirten indeksin (`CLUSTERED INDEX`) ismi `PRIMARY`'dir.
- İndeksler, anahtarların (`KEY`s) arkaplanındaki yapıdır.
- Bir verinin indekslenmiş bir yapı içerisinde aranması, indekslenmemiş bir yapı içerisinde aranmasından çok daha hızlıdır.
- Ne kadar hızlı? Alfabetik sıralamalı bir sözlükte kelîme aramanın rastgele kelîmelerden oluşan bir defterde bir kelîme aramaya olan üstünlüğü kadar.
- Bir indeks sadece bir sütuna göre sıralanmayabilir, şûbelerinizi şehir koduna göre sıraladıktan sonra aynı şehir içinde olan şûbelerinizin isimlerine göre alfabetik sırada sıralanmasını isteyebilirsiniz. Bu durumda indeksiniz, iki alana göre sıralama yapan bir `COMPOSITE INDEX` olmuş olur(?).

İndekslerin Çalışma Mantığı

- Bir indeks içerisinde arama işlemi ikiye bölme algoritmasıyla ('B-TREE') yapılır.
- Bu algoritma, veriyi ikiye böler ve ortadaki veri aranan veri ise, işlemi sonlandırır; Değilse, verinin ortadaki veriden büyük mü, küçük mü olduğuna bakar ve buna göre veriyi aramaya hangi yarım üzerinden gideceğine karar verir. Sonra o yarımı da ikiye böler ve aynı kontrolleri yine yapar..
- Misal, bu algoritmayla aşağıdaki liste içerisinde 7 sayısını aramak istersek; 1, 4, 6, 7, 19, 21, 25, 27, 36 Öncelikle ortadaki veri alınır : 19 (**İlk veri kontrolü**) 19 = 7 mi : HAYIR 19 > 7 mi : EVET O hâlde aradığımız veri sol blokta (1, 4, 6, 7) (Verinin yarısından kurtulduk) Ortadaki (ortadaki yok, yarı sıradaki) veriyi alalım : 4 (**2. veri kontrolü**) 4 = 7 mi : HAYIR 4 > 7 mi : HAYIR O hâlde aradığımız veri sağ blokta (6, 7) 6 = 7 mi : HAYIR 7 = 7 mi : EVET Yarıya bölme algoritması yaklaşık olarak böyle çalışır.
- Her adımda verinin yaklaşık yarısı elendiğinden büyük veride üzerinde arama işleminde müthiş performans sağlar; misal, 250 bin satırlık bir veride indeks olmadan arama yaparsak, 250 bin kez kontrol yapmamız gerekebilir; fakat indeks ile arama yapmış olsaydık, yaklaşık 30-40 adımda sonuca ulaşabiliriz. Kontrol etmek için sürekli diskten okuma yazma yapmamız ve işlemcide karşılaştırma yapmamız gerektiğinden bu, çok sayıda kontrol sistem için ağır bir yük olmaktadır.

İndeks Oluşturma

- Tablo tanımından sonra virgül koyup, ardından indeks belirtebilirsiniz:

```
CREATE TABLE tabloIsmi(
    sutunIsmi sutunTanimi,
    sutun2Ismi sutun2Tanimi,
    ...
    INDEX [indeksIsmi] (sutunIsmi [, sutunIsmi2, sutunIsmi3, ...])
);
```

- İndeks ismini belirtmek zorunda değilsiniz, eğer belirtmezseniz, MySQL otomatik olarak o sütunun ismini indeks ismi olarak atar.
- Bir indeksin birden fazla sütuna göre yapılandırılabileceğini söylemiştik, ikinci, üçüncü sütunlar ihtiyârîdir (tercihidir).
- İndeksleri tablo oluşturulurken belirtmeniz tavsiye edilir; zîrâ gelen birincil indekse göre yerleştirilir, diğer indekslerin defterine de yerleştirilir; böylece verinizin sıralanması için bir sistem yükü oluşmaz.
- Eğer tabloya sonradan bir indeks eklerseniz ve hele ki bu indeks birincil indeks ise, sisteme yük biner; fakat tabloda indeksiniz yoksa her sorgulamada yaklaşık olarak bir indeks oluşturacak kadar yük oluşması yerine hemen bir indeks oluşturun:

```
CREATE INDEX indeksIsmi ON tabloIsmi (sutun [,sutun2, sutun3, ...]);
# Başka bir usûl:
ALTER TABLE tabloIsmi ADD INDEX [indeksIsmi](sutun1 [,sutun2, ...]);
```

> ***DETAY İSTEYENLER İÇİN NOT :*** MySQL'de indeks ismini belirtmezseniz o sütun isminde bir indeks oluşturulur. Eğer o sütunun isminde bir indeks varsa 'sutunIsmi_2' biçiminde bir isim verir; fakat, sütun isimlerini indeks ismi olarak kullanıyorsanız, isimlendirilmiş indekslerinizi önce yazmanız uygun olur; zîrâ önce isimsiz indeksi yazıp, sonra o isimsiz indekse atanacak isimle aynı isimde bir indeks yazarsanız tablo oluşturulmaz, kod hatâ verir.

- İndeks oluşturmayla ilgili başka detaylar da var, hepsi bu değil! Okumaya devâm..

Mevcut İndeksleri Görüntüleme

- İndeksler, birkaç farklı şekilde görüntülenebilir:

```
SHOW INDEXES FROM tabloIsmi IN veritabaniIsmi;
SHOW INDEXES FROM [veritabaniIsmi.]tabloIsmi;
SHOW KEYS FROM [veritabaniIsmi.]tabloIsmi;
SHOW KEYS FROM tabloIsmi IN veritabaniIsmi;
```

- Veritabanı belirtilmezse bağlantı kurulan veritabanındaki indeksler görüntülenir.
- Bu sorgu sonucunda indekslerle ilgili pek çok bilgi görüntülenir, kafanızın karışmasından endişe ediyorsanız, bunlara indeks konusunu okumayı bitirdikten sonra bakabilirsiniz:

1. **non_unique** : İndeks içerisindeki verilerin münferid (tekil) olup, olmadığını bildirir.
2. **key_name** : İndeks ismi. Birincil anahtar dâimâ `PRIMARY` isimli indekstir.
3. **seq_in_index** : İndeksin tablodaki sütun numarası. Tabloda sütun numaraları 1'den başlar.
4. **column_name** : Sütun ismi
5. **collation** : İndeksin sıralanma hizâsını gösterir. 'A', (ASCENDING) artan sırada sıralandığını, 'B', (DESCENDING) azalan sırada sıralandığını gösterir; 'NULL' ise sıralanmadığını gösterir.
6. **cardinality** : Sıralanan verinin ne kadar münferid olduğunu gösteren bir parametredir. Misal, 100 tâne Ahmed verisi varsa, 'cardinality' değeri düşüktür; fakat her isimden sadece bir kez varsa bu değer yüksektir. Bu değerin yüksek olması, sorgu performansının daha iyi olacağına işâret eder. Sorgu eniyileyicisi sorguyu çalıştırmak için hangi indeksi seçeceğine karar verirken bu değere bakar. Bu değer de 'statistics' (istatistik) denilen bir yapıya dayanır. İstatistikler, hangi veriden kaç tâne olduğu bilgisini tutan yapılardır. 'cardinality' değeri basitçe sütundaki münferid değer bilgisini verir; fakat istatistikler taze değilse, bu değer yanlış olabilir. Bu değer zamânla bozulabildiğinden, uygun zamânlarda `ANALYZE TABLE tabloIsmi;` deyimi çalıştırılmalıdır.
7. **sub_part** : Sütundaki tüm veriler indekslenmişse NULL'dır. Sütundaki tüm veriler indekslenmemişse, parçalı indeksin karakter sayısını gösterir (Evet, bir sütuna indeks eklediğimizde zamânla eklenen veriler indekse göre yerleşemeyebilir, biraz sonra değineceğim inşâAllâh)
8. **null** : Sütunun NULL veri barındırıp, barındıramayacağını belirtir.
9. **index_type** : İndeksin oluşturulduğu algoritmayı gösterir : BTREE, HASH, RTREE veyâ FULLTEXT...

10. **index_comment** : İndeks oluşturulurken eklenen yorumu gösterir
11. **visible** : İndeksin, sorgu eniyileyicisi ('query optimizer') tarafından görünebilir (kullanılabilir) olup, olmadığını belirtir.

İndeksleri Yakından Tanıyalım (Detaylara bakalım)

- **İndeksler zamânla bozulabilir**; bunu daha iyi anlamak için elinizde bir telefon defteri olduğunu düşünün. Telefon defterini alfabetik sıralamaya göre tutuyorsunuz ve gelen numarayı yerleştirirken yer değiştirme yapmanız gerekiyor. Misal Ahmed Kasım isminden sonra Ahmed Kerîm ismi yazılmış. Şimdi yeni bir isim olan Ahmed Kâdir ismini yazmanız gerekiyor. Bu durumda Ahmed Kerîm ismini silmeniz ve Ahmed Kâdir ismini yazdıktan sonra Ahmed Kerîm ismini yeniden yazmanız gerekiyor. Böyle olduğunda veriniz hâlen alfabetik sıralamaya uygun ve çok fazla bir iş yapmadınız. Sayfada yeterince boşluk varsa ne âlâ... Peki sayfa dolduysa? Sayfa dolduysa sonraki sayfadaki tüm verileri kaydırmanız gerekecek, o sayfa da doluysa ondan sonraki sayfadaki verileri de kaydırmanız gerekecek... İşte bu, çok büyük bir iş yükü oluşturduğundan, yeni gelen ismi en sona yazarsınız ve o isimde indekslenmemiş olur, yanî alfabetik sıralamanın dışarısında kalır. Veritabanı sistemi de böyle çalışır ve yer yoksa veriyi sona ekler. Bir arama yaptığınızda önce indekslenmiş veri aranır, aranan veri bulunamadıysa indekslenmemiş verilerin her birine teker teker bakılır. İşte bu indekslenmemiş verinin indekslenmiş veriye oranı indeksin bozulma oranıdır. İndeksin bozulma oranı %30'dan fazlaysa indeksi düzeltmeniz tavsiye edilir. Çünkü, indeks artık işe yaramamaya başlamış demektir.
- **İndekslerin çeşitleri vardır**; gerçek sıralama birincil indeks (`CLUSTERED INDEX`)'tir. Diğer indekslere ise `NON-CLUSTERED INDEX` denir. Bunun dışında indeksler, münferid veri barındırıp, barındırmadığına göre iki çeşide ayrılır : `UNIQUE INDEX` ve `NON-UNIQUE INDEX` . `UNIQUE INDEX` 'te veri tekil olduğundan veri bulunduğunda arama sonlandırılır, `NON-UNIQUE INDEX` 'te ise eğer sorgu sonucuna bir satır sayısı sınırı (`LIMIT`) konulmamışsa verinin başından, sonuna kadar aranan veri bulununcaya kadar arama devâm eder. Misal, yalnızca isim sütununa göre indeksimiz var ve ismi 'Ahmed' soyismi 'Piyâde olan birisini arıyoruz. `Ahmed` verisi bulunduktan sonra bulunan tüm Ahmed'lerin soyismi kontrol edilir ve soyismi 'Piyâde' olan Ahmed bulunduktan sonra veriler alınmaya başlanır, soyismi Piyâde'den farklı olan ilk satıra gelinceye kadar bu işlem devâm eder; çünkü birden fazla Ahmed Piyâde bulunabilir. Bu sebeple, verimiz gerçekten bir tâne ise, misal T.C. kimlik numarası, okul numarası, diploma numarası, ürün kodu... gibiyse, indeksimizin `UNIQUE INDEX` olması bizim performansı arttır. Tablonun birincil anahtarı (`PRIMARY KEY`) hem `UNIQUE` , hem de `NOT NULL` 'dır, unutmayalım.
- **Bir anahtarın pek çok özelliği, aynı zamânda bir indeksin de özelliğidir**; çünkü indeksler, anahtarların işlevlerini sağlayan yapılardır. `NOT NULL` kısıtı bir anahtar

olduğundan, aynı zamânda o indeksin de özelliğidir.

- **İndekslerin dayanıklılığını arttırmak mümkündür**; bu başlık altında, ilk maddede indekslerin bozulabileceğinden söz etmiştik. Bu, verinin olması gereken sırada yeterli boş alanın bulunmadığı durumda oluşan bir durumdur. Tablonun yoğun bir şekilde ekleme işlemi alıp, almamasına; tablodaki verilerin yeniden indekslenmesinin çok mâliyyetli olup, olmamasına bağlı olarak, verilerin tutulduğu 'page' yapılarında boşluklar bırakarak indekslerin bozulmasını geciktirebiliriz. Bu işlem, telefon defterinde her sayfanın altında boş alan bırakmaya benzer. Bıraktığımız boş alanın disk açısından olan mâliyyeti, sistemin indeksinin tahmînî bozulma süresinin yakınlığı ve bozulması durumunda oluşan mâliyyetine göre az ise, verilerimizin arasında boşluk ekleyerek indeksimizin bozulmasını önemli ölçüde geciktirebiliriz. Bilhassa büyük veriler için ve sürekli veri eklenen tablolar için ısrarla tavsiye edilir. Bu boşluk miktarına 'fill factor' ('doluluk oranı') denir.
- ..

İndeks Çeşitlerine Detaylı Bakış

- İndeksler farklı açılardan birden fazla çeşide ayrılırlar.
- Burada çeşit sayısı çok fazla olmadığından, burada bakış açılarına göre ayrılmamıştır; yanî bir indeks bu listedeki birden fazla indeks tipinde olabilir.

1) Birincil (CLUSTERED) ve ardıl (NON-CLUSTERED) indeks

- Bir indeks, verinin gerçek anlamda sıralanmış hâli ise, buna `PRIMARY` veyâ `CLUSTERED INDEX` denir. Veri, diskte bu indekse göre sıralandığından, indeks için neredeyse hiç hâfıza harcanmaz.
- Tablodaki birincil (`PRIMARY`) indeks, birincil anahtardır. Eğer birincil anahtar yoksa, tablo oluşturulurken belirtilen `UNIQUE` bir alan birincil indeks olur.
- Sonraki indeksler birer `NON-CLUSTERED INDEX`'tir.
- `NON-CLUSTERED INDEX` verinin gerçek sıralaması şeklinde olmadığından, verinin ilgili kısmının kopyalanmasını gerektirir. Misal, müşteri verisi isme göre sıralandıysa, ve telefon numarasına göre yeni bir indeks eklediysek, tablodaki telefon numaraları + isim kadar bir depolama alanı bu indeks için ayrılır ve bu sıralama telefon numarasına göre yapılır.
- Her ardıl indeks kendi indeksindeki sütun verisi + birincil indeks verisi kadar yer kaplar.
- Sıkça sorgulanmayan veyâ sorgulanması mâliyyetli olmayan alanlar için ardıl indeks oluşturmak hâfızadad lüzumsuz yer tutar; bunlar ihtiyaca göre kullanılmalı.

- **İPUCU** : Her ardıl indeks, birincil indeks + kendi indeksi kadar yer kapladığından ötürü birincil indeksi kısa bir alan tutmak çok önemlidir. Bu sebeple her tabloda id isimli bir birincil anahtar oluşturmak neredeyse her durum için iyi bir çözümdür.

2) PREFIX indeks

- Metînsel alanlar çok uzun alanlar olabilir ve biz bir metînsel alan için indeks eklemeye kalktığımızda verinin alfabetik sıraya göre dizilmesi tüm harfler için uygulanacaktır; yeni bir veri eklendiğinde verinin tüm harfleri kontrol edilmek sûretiyle ilgili alana yerleştirilir. Misal, 'Uzungöl' kelîmesi yerleştirilirken, U harfi bulunur, sonra U ile başlayan veri arasında Z ile başlayan aranır, yoksa 'Uzungöl' en sona eklenir, varsa, bulunan o verinin 3. harfinin 'u'dan önce mi, sonra mı olduğuna bakılır, bu böyle devâm eder..
- İşte bu, ekleme işlemini oldukça yavaşlatan bir durumdur. Eğer bizim için metnin belli bir karakter sayısına göre sıralanması yeterliyse, bunu belirterek sistemi ekleme işlemlerinde fazla yormamış oluruz. Misal, bunu 2 seçtiysek, sistem 'Uzungöl' kelîmesindeki z harfi bulunduktan sonraki ilk boşluğa verimiz eklenir.
- Bir PREFIX indeks oluşturmak için sütun isminden sonra parantez içerisinde kaç karaktere göre indeksleme yapılacağını belirtin: `CREATE INDEX indeksIsmi ON tabloIsmi(sutunIsmi(karakterSayisi));`
- Tablo oluştururken şöyle belirtebilirsiniz:

```
CREATE TABLE tabloIsmi(
    #sutunlar ve tanımları
    INDEX [indeksIsmi] sutunIsmi(karakterSayisi)
);
```

3) Münferid, tekil ('UNIQUE') indeks

- Sadece tekrar etmeyen verilerden oluşan bir sütun için `UNIQUE` indeks oluşturabilirsiniz:

```
CREATE UNIQUE {INDEX | KEY} indeksIsmi
    ON tabloIsmi(sutunIsmi [,sutun2Ismi, sutun3Ismi, ...])
```

- Görüldüğü üzere, `INDEX` yerine `KEY` anahtar kelîmesi yazılabilir.
- Bir münferid indeks birden fazla sütunu kapsayacak şekilde oluşturulabilir, bu durumda tabloya iki sütun değeri de aynı olan bir satır eklenemez. Misal, isim ve soyisim sütunlarını kapsayacak şekilde bir indeks eklendiyse tabloda ismi ve soyismi aynı olan kişiler olamaz; fakat ikisinden birisi farklı olmak kaydıyla aynı isimde ve aynı soyisimde kişiler olabilir.

- **ÖNEMLİ NOT** : MySQL'de `UNIQUE` indeksi olan bir sütunda birden fazla `NULL` veri olması bu indekse aykırı değildir. Diğer bâzı veritabanlarında (misal, MsSQL'de) `UNIQUE` indeksi olan bir sütuna sadece bir `NULL` veri eklenebilir.

4) Birleştirilmiş (COMPOSITE) indeks

- Birleştirilmiş indeks, oluşturulurken birden fazla sütun belirtilen, yanî birden fazla sütuna göre sıralanan indekstir:

```
CREATE INDEX indeksIsmi
    ON tabloIsmi(sutun1, sutun2[, sutun3, ...]);
```

- Bir önceki indeks çeşidinde de değinildiği üzere, bir birleştirilmiş indeks eğer münferid (tekil) indeks ise verilen sütundaki verilerin aynı anda aynı olduğu ikinci bir satır olamaz, fakat bir sütunda aynı veriler olabilir.

- Misal, isim - soyisim alanlarını kapsayan bir birleştirilmiş münferid indeks içeren tabloda 'Câfer Kaptan', 'Câfer Fırıncıoğlu', 'Kâsım Fırıncıoğlu', 'Melik Kaptan' verileri olabilir, fakat 'Câfer Kaptan' olamaz.

5) Görünmez ('INVISIBLE') indeks

- Bir indeksin görünmez olmasıı o indeksin sorgu eniyileyicisi tarafından kullanılmayacağı anlamına gelir. Nâdir olarak, ihtiyaç duyulan bir şeydir.

- İndeksin etkin veyâ kapalı olması olarak da değerlendirilebilir.

- İndeks oluşturulurken belirtilebilir:

```
CREATE INDEX indeksIsmi
    ON tabloIsmi(sutunIsmi [, sutun2Ismi,..]) [VISIBLE | INVISIBLE];
```

- Varsayılan olarak tüm indeksler görünürdür.

- İndeksin görünmezliğini şu şekilde değiştirebiliriz:

```
ALTER INDEX indeksIsmi [VISIBLE | INVISIBLE];
```

- Birincil anahtar görünmez indeks olamaz.

- MySQL sorgu eniyileyicisi varsayılan olarak görünmez indeksleri kullanmaz, fakat bu ayar değiştirilebilir.

6) Azalan Sıralamalı (DESCENDING) indeks

- Pek çok sorguda sıralama büyükten küçüğe doğrudur. İndeks, küçükten büyüğe doğru yapıldığında bu sorguların çalışma performansı düşer.
- Bu sebeple MySQL 8.0.0 sürümünden itibâren azalan sıralamalı indeksi sundu.
- Basitçe indeksten sonra indeksin sıralama yönünü belirtebilirsiniz:

```
CREATE INDEX indeksIsmi
ON tabloIsmi(sutunIsmi [ASC | DESC]
    [, sutun2Ismi [ASC | DESC], ..]);
# ASC : ASCENDING (artan), DESC : DESCENDING (azalan)
```

- Her sütun için azalan sırada mı, artan sırada mı sıralanacağı belirtilebilir.
- Varsayılan sıralama artan sıralamadır (`ASC`).

7) Fonksiyonel indeks

- Fonksiyonel indeks, verilerin bir sütuna göre değil, bir fonksiyona göre sıralamasının çıkarıldığı indeks çeşididir:

```
CREATE INDEX indeksIsmi ON tabloIsmi ((fonksiyonIsmi(sutun)));
# Çift parantez kullanıldığına dikkat ediniz!
# fonksiyonIsmine girdi olan sutun ismi / isimleri verilmeli
```

- Bu indeks, verilerinize bir fonksiyon uygulandığı durumda kullanışlı olabilir.
- Misal, satışlarınızı tutan bir tablonuzda satış târihi ('saleDate')nin yıla göre indeksini oluşturmak istiyorsanız, şöyle bir kod yazabilirsiniz:

```
CREATE INDEX iOnSaleDateAsYear ON sales(( YEAR(saleDate) ));
```

İndeks Silme

- Diğer veritabanı nesneleri gibi `DROP` komutuyla silinir:

```
DROP INDEX indeksIsmi ON tabloIsmi [algoritmaTipi | kilitTipi];
```

- Algoritma veyâ kilit seçeneği zorunlu alan değildir, belirtilmediğinde varsayılan usûl uygulanır.

- İndeks kaldırılırken kullanılacak algoritma belirtilebilir.
- Algoritma için söz dizimi şöyledir: `ALGORITHM [=] {DEFAULT | INPLACE | COPY}`
- `COPY` seçeneği kullanılırsa, MySQL her bir satırı yeni bir tabloya kopyalar ve ardından yeni tablo üzerinde indeks kaldırma işlemi yürütülür. Bu kopyalama esnasında veri değişikliğine izin verilmez, yanî kopyalama bitene kadar tablo üzerinde `INSERT` ve `UPDATE` işlemi çalıştırılamaz.
- `INPLACE` seçeneği kullanılırsa, tablo olduğu yerde yeniden inşâ edilir, veriler yeni bir tabloya kopyalanmaz. İşlem hâzırlanırken ve yürütülürken tabloda özel bir önveri ('metadata') kilidi çalışmaktadır, bu kilit indeks kaldırma işlemi yürütülürken veri değişikliğinin (`INSERT` , `UPDATE`) gerçekleşmesine müsaade eder.
- `DEFAULT` seçeneği algoritma belirtmemekle aynı şeydir, varsayılanı seçmektir.
- Kilit için söz dizimi : `LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}`
- `DEFAULT` seçeneği varsayılaı ifâde eder. Kilit varsayılan olarak şu yaklaşımla belirlenir. Silme işlemi devâm ederken tabloya yazma ve okuma mümkünse bu seçilir, bu mümkün değilse, ama eşzamânlı okuma mümkünse bu seçilir, o da mümkün değilse tabloya yazma ve okuma kısıtlaması koyan `EXCLUSIVE` seçilir.
- `NONE` seçeneği, tabloya eşzamânlı okuma ve yazmanın mümkün olduğu seçenektir; fakat bu seçenek sistem tarafından desteklenmiyor olabilir.
- `SHARED` seçeneği, eşzamânlı okumayı destekler, yanî indeks kaldırma kaldırılırken tablodan okuma yapabilirsiniz; fakat eşzamânlı yazmayı desteklemez.
- `EXCLUSIVE` seçeneği, tabloyu tamâmen kilitler ve tablodaki veriye erişimi kapatır.

Sorgu Eniyileyicisi Sorgu Planını Öğrenme

- Sorgu eniyileyici sorguyu daha verimli çalıştıracak indeksleri seçmek için istatistikleri kullanır.
- Sorgu eniyileyicisinin sorguyu nasıl planladığını görmek, hem istatistik, hem de indeks için güzel bir ipucu sağlayabilir; hangi indeksi kullanmanın verimli olduğunu görmek gibi isteklerle sorgu planını öğrenmek isteyebiliriz:

```
EXPLAIN SELECT * FROM tabloIsmi;
```

İNDEKS DAYANIKLILIĞINI ARTTIRMA ve DOLULUK ORANINI DEĞİŞTİRME

- İndekslerin kullanışlılığı iki şeye dayanmaktadır: istatistiklerin tazeliği ve indekslerin dayanıklılığı.
- İndeks dayanıklılığı, tabloya gelen veri değişikliğinin veyâ veri ekleme işleminin indeksi kolayca bozamamasını ifâde etmektedir.
- Bu da bir gerçekliğe dayanır, tablo indekslerinin yerleştiği ve veritabanının en küçük birimi olan 'page'lerin içerisindeki boşluk oranı.. Bu boşluk oranına 'fill factor' denir.
- Bu boşluk oranı, yeni gelen verinin indeks içerisinde yerleştirilmesi için gereken kaydırma miktarını bir page gibi (varsayılan olarak 16 KB) küçük bir boyuta indirgediğinden yeni gelen verinin indekse yerleşmesini sağlar. 'page'de boş yer yoksa, yeni gelen verinin indeksi indekslenemeyeceğinden, indeksin sonunda düzensiz olarak saklanır. Bu düzensiz verilerin artması, sorgunun indekssiz aramaya dönmesi anlamına geldiğinden, performansı ciddi ölçüde düşürür.
- Bu boşluk oranı, büyük hacimli ve sürekli aktif olan sistemlerde her tabloya göre özel belirlenmelidir.
- MySQL'de 'InnoDB' veritabanı motorunda 'fill factor' özelliği vardır; fakat bu özellik indeks oluşturulurken verilemiyor; bu yapılandırma ayarını veritabanı sunucusu düzeyinde veyâ tablo düzeyinde değiştirebilirsiniz.
- Bir indeks eklenirken veritabanındaki `@@innodb_fill_factor` özelliğine göre 'page'lerde boşluk bırakılır. Bu değer varsayılan olarak 100'dür. Bu, sayfanın ('page'in) 1/16'sının gelecekteki büyüme için ayrılacağını ifâde ediyor (diğer veritabanlarında hiç ayrılmayacağını ifâde ediyor olabilir).
- `SELECT @@innodb_fill_factor;` komutunu çalıştırarak sunucu düzeyindeki doluluk oranı ('fill factor')nı öğrenebilirsiniz.
- `SET GLOBAL innodb_fill_factor = 80;` kodunu kullanarak bu değeri değiştirebilirsiniz. Bu oranı değiştirmek, tüm veritabanlarını etkiler.
- Maâlesef, doluluk oranını veritabanı düzeyinde ayarlayamıyorsunuz. Bu sebeple bir tabloyu veyâ indeksi oluşturmadan evvel, küresel `innodb_fill_factor` değerini istediğiniz seviyeye getirmeli ve sonra geri sunucu ayarına döndürmelisiniz (tabi isterseniz öyle bırakabilirsiniz).
- Tablo ve indekslerin istatistiğini tazelemek için `ANALYZE TABLE tabloIsmi` deyimini kullanabilirsiniz. Bu, tablo istatistiklerini tazeler; verinizi veyâ indeks doluluk oranını değiştirmez.

- Tablo ve indekslerin bakımı için ise `OPTIMIZE TABLE tabloIsmi;` komutunu çalıştırabilirsiniz; fakat çalıştırmadan dikkat etmeniz gereken bir şey var. Bu sorgu, gereksiz verileri ve boşlukları kaldırır, tablo ve bağlı indekslerin aralarındaki boşlukları veritabanı `innodb_fill_factor` değerine uygun şekle getirir. Eğer tablonuzun doluluk oranının değeri veritabanının doluluk oranının değerinden ayrı olacaksa, bu sorguyu çalıştırmadan evvel az önce anlatıldığı gibi küresel `innodb_fill_factor` değerini değiştirip, işlemden sonra eski hâline getirmelisiniz. Bunun için bir saklı yordam yazabilirsiniz.

- İndekslerin yenilenmesi için birkaç yöntem bulunuyor; fakat InnoDB için özel bir deyim olan `REINDEX TABLE tabloIsmi;` deyimi çalışmıyor.

- MyISAM, CSV ve ARCHIVE veritabanı motorları için `REPAIR TABLE tabloIsmi;` komutu kullanılabilir.

- InnoDB veritabanı motoru için, indeksi silip, yeniden oluşturmak, indeksin yeniden inşâ edilmesini ve indekslenmeyen verilerin indekse dâhil olmasını sağlıyor. Bunun için şu komutları sırasıyla çalıştırın:

```
ALTER TABLE tabloIsmi
    DROP INDEX indeksIsmi,
    ADD INDEX indeksIsmi(sutun [, sutun2, ...]);
```

- Ayrıca tablonun oluşturulduğu veritabanı motorunu değiştirip, eski hâline getirmek tablonun ve indekslerin yeniden inşâ edilmesi anlamına gelir:

```
ALTER TABLE tabloIsmi ENGINE = MyISAM;
ALTER TABLE tabloIsmi ENGINE = InnoDB;
```

- İndekslerin içerisinde çok boşluk olmasının, sorgularda fazla sayıda 'page' okunabileceğına, bilhassa veri ekleme - silme - tazeleme işlemlerinin uzun sürmesine sebebiyyet vereceğine dikkat ediniz.

Sorgu Eniyileyicisi İçin İndeks Önerme ve Belirtme

- MySQL, sorgunun çekilirken hangi indeksin kullanılacağını belirtmenize / önermenize izin verir. Bunun için `USE INDEX` ifâdesi kullanılır.

```
SELECT * FROM tabloIsmi USE INDEX(index1 [, index2, index2])
WHERE sart;
```

- Sorgu enyileyicisi sorgu planına dayanarak önerdiğiniz indekslerin birini veyâ birkaçını veyâ hepsini kullanmayabilir.
- Nâdir de olsa, bâzen bir indeksin kullanımını zorunlu kılmak isteyebiliriz. Bu durumda sorgu eniyileyicisini belirttiğimiz indeksleri kullanmaya zorlayabiliriz:

```
SELECT * FROM tabloIsmi FORCE INDEX(index1 [, index2, index2])
WHERE sart;
```

- Sorgu eniyileyicisini İndeks kullanmaya zorlamak isteyebileceğimiz bâzı durumlar şunlardır:
    1. İstatistiklerimiz yanlıştır, henüz yenilenmemiştir ve biz o sorgunun hangi indeksle daha verimli çalışabileceğini biliyoruzdur.
    2. İstatistikler çok geniş bir değer aralığına sâhipse, sorgu eniyileyicisinin istatistiği kontrol ederek indeks seçmesi performansı biraz düşürebilir. Eğer biz doğru indeksi biliyorsak, sorgu eniyileyicisinin istatistikleri kontrol etmeden indeks kullanmasını sağlayabiliriz.
- Sorgu eniyileyicisi `USE INDEX` ile belirtilen indeks(ler)i kullanmayı değerlendirir; fakat `FORCE INDEX` ile belirtilen indeks(ler)i kullanır.
- **ÖNEMLİ NOT** : Bâzen sorgumuzun için uygun bir indeks olduğu hâlde, sorgu eniyileyicisi indeks kullanmayabilir. Büyük sorgularda sorgunun hangi indeksle çalıştığını öğrenip, gerekirse `FORCE INDEX` deyimini kullanmanız uygun olabilir.

İndeks Onarımı

- Daha evvel değinildiği gibi indeksler bozulabilir.
- İndekslerin çalışma performansını arttırmak için farklı çeşitte işlemler vardır.

İstatistikleri tazeleme

- En basiti, sorgu eniyileyicisinin indeks seçerken göz önüne aldığı istatistikleri tazelemektir. Bu, genellikle sisteme fazla yük getirmeyen bir işlemdir; çünkü bu işlem verimizin yapısını ve indeksi etkilemez; çünkü sadece indeks içerisinde hangi değerden kaç adet olduğu bilgisi tazelenir, bunun için de tablonun ilgili sütunu baştan sona okunur; bu işlem bilhasssa münferid (`UNIQUE`) olmayan indeksler için uygulanmalıdır. İşlemin ne sıklıkta uygulanacağı veritabanınıza gelen veri değişikliklerinin miktarıyla alâkalıdır. İndekslerimizin tutulduğu B-Tree ağacı sağlamsa (misal, tablo oluşturulduktan sonra çok miktarda yeni veri eklenmemişse veyâ 'page'lerdeki boşluk oranı ('fill factor') dolup, çok taşmamışsa, ezcümle indekssiz veri sayısı çok fazla değilse) uygulanabilir.

- Bu işlem için `ANALYZE TABLE tabloIsmi;` kodu kullanılabilir.

İndekslerle İlgili Diğer Notlar

- MySQL'de aynı işlevde birden fazla indeks eklenebildiğinden, isimleri farklı, fakat işlevleri aynı olan indeksler olabilir; yâ dâ birbirini kapsayan indeksler olabilir. Bu sebeple indekslerinizi dikkatlice oluşturun ki, sisteme yük getiren, ama fayda sağlamayan çöp indeksler olmasın.

TETİKLEYİCİLER (TRIGGERS)

- Tetikleyiciler, bir olay gerçekleştikten önce veyâ sonra çalıştırılmak istenen fonksiyonların, kod parçalarının olduğu durumlarda kullanılan vt nesneleridir.
- İki çeşit tetikleyici vardır : Satır düzeyinde tetikleyici, durum düzeyinde tetikleyici
- Satır düzeyinde tetikleyici tablo üzerinde gerçekleştirilmek istenen her `INSERT`, `UPDATE` ve `DELETE` işlemlerinden önce veyâ sonra çalışan yapılardır. Toplu bir işlem yapılsa dahî tetikleyici her satır için tek tek çalışır; **MySQL bunu destekler**.
- Durum düzeyinde tetikleyici ise bir bütünsel işlem ('transaction') için bir kez çalışan ve kaç satırın eklendiğini / silindiğini / tazelendiğini öğrenmeye yardımcı olan tetikleyicilerdir; **MySQL bunu desteklemez**.
- Tetikleyici kullanmanın pek çok olası faydaları vardır:
    1. Veriniz üzerindeki tüm değişikliklerin yapılmasını önleyici ('proaktif') biçimde izleyebilirsiniz; yazdığınız tetikleyici gövde kodu ile ilgili değişikliği kabûl edebilir veyâ iptal edebilirsiniz.
    2. Bir olay `INSERT`, `UPDATE` veyâ `DELETE`) işlemi gerçekleştiğinde başka işlemlerin çalışmasını sağlayabilirsiniz.
    3. Tablo üzerindeki kontrolünüzü arttırdığından size veri bütünlüğü ve güvenlik desteği sağlar.
    4. İşlem gerçekleştiğinde olan hatâları daha veritabanındayken ele almayı ve çözmeyi sağladığından daha kararlı bir sistem oluşturmanızı sağlar.
    5. Zamânlanmış görevleri çalıştırmak için muadil bir seçenektir.
- Bunun yanında tetikleyicilerin sisteme yük bindirmesi, hatâ durumlarında hatâ ayıklamayı zorlaştırabilmesi gibi menfî yönleri de vardır.

Tetikleyici Oluşturma

- Söz dizimi:

```
CREATE TRIGGER tetikleyiciIsmi
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
FOR EACH ROW
BEGIN
    # triggerGövdesi
END;
```

- Yukarıdaki kodda, `BEFORE` veyâ `AFTER` deyimi tetikleyicinin çalışma zamânını belirtiyor, `INSERT` , `UPDATE` veyâ `DELETE` ise işlem tipini belirtiyor.

- Tetikleyicinin sonuna noktalı virgül koyduğumuza dikkat ediniz. Bu, tetikleyici gövdesinin tek bir komuttan oluştuğu ve komutun sonunda noktalı virgül içermediği durumda kullanılabilir; fakat tetikleyiciniz birden fazla komuttan oluşuyorsa aralarına satır sonu ifâdesi olan noktalı virgülü koymanız gerektiğinden, tetikleyicinin başında satır sonu ifâdesini değiştirmelisiniz. Daha fazla bilgi için Saklı Yordamlar konusuna bakınız. Tetikleyici içerisinde birden fazla komut yazabilmemiz için tetikleyici şu yapıda olmalıdır:

```
DELIMITER $$
CREATE TRIGGER tetikleyiciIsmi
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
FOR EACH ROW
BEGIN
    # triggerGövdesi
    # Burada noktalı virgülle ayrılmış birden fazla kod yazabilirsiniz
END$$
DELIMITER ;
```

- Tetikleyici içerisinde döngüler, şart ifâdeleri gibi yapılar kullanılabilir.

- Tetikleyici konusunda bilmemiz gereken en temel bilgilerden birisi, yapılan işlem sonucu değişen verilere ve eski hâllerine `OLD` ve `NEW` tabloları üzerinden ulaşabildiğimizdir. `OLD` tablosu verinin işlemden önceki hâlini tutuyorken, `NEW` tablosu verinin işlemden sonraki hâlini tutmaktadır. Yaptığımız işleme göre bu tabloların ikisine veyâ birisine erişebiliriz. Misal, bir ekleme işlemi öncesinde `OLD` tablosunda bir veri yoktur (veyâ `OLD` tablosu yoktur); çünkü eklenen satır yeni eklendi ve bu işlemden önce öyle bir satır yoktu.

- Satırların verilerine erişirken nesne yönelimli programlamada olduğu gibi `OLD.sutunIsmi` veyâ `NEW.sutunIsmi` şeklinde erişilir.

- Telefon, fiyat ve bildirim tablolarından oluşan bir veritabanında fiyat değişimini bildirimler tablosuna kaydeden bir bir tetikleyici yazalım inşâAllâh:

```
DROP TABLE IF EXISTS prices, phones;# Bu isimde tablolar varsa silelim
# Tabloları oluşturalım:
CREATE TABLE phones(
    id int PRIMARY KEY AUTO_INCREMENT,
    brand VARCHAR(50),
    model VARCHAR(50),
    info VARCHAR(50));
CREATE TABLE prices(
    phoneId INT,
    price DOUBLE,
    FOREIGN KEY(phoneId)
        REFERENCES phones(id) ON DELETE CASCADE);
CREATE TABLE notifications(
    id int PRIMARY KEY AUTO_INCREMENT,
    phoneId int,# Telefonun stok kodu
    percentOfChange DOUBLE,# Fiyat değişim oranı
    price DOUBLE,# Yeni fiyat
    FOREIGN KEY phoneKey(phoneId)
        REFERENCES prices(phoneId) ON DELETE CASCADE);

# Veri ekleyelim:
INSERT INTO phones(brand, model, info) VALUES
    ('Samsung', 'S23', 'Özellikleri çok fazla..'),
    ('Samsung', 'A50', 'Daha uygun fiyatlı..'),
    ('Samsung', 'S25', 'Yepyeni...');
INSERT INTO prices VALUES
    ((SELECT id FROM phones WHERE
            brand = 'Samsung' AND model = 'S23'),
    32450),
    ((SELECT id FROM phones
        WHERE brand = 'Samsung' AND model = 'S25'),
    45000),
    ((SELECT id FROM phones
        WHERE brand = 'Samsung' AND model = 'A50'),
    14000);

# Şimdi tetikleyici yazma vakti:
DELIMITER $$
CREATE TRIGGER trgForChanging
AFTER UPDATE ON prices
FOR EACH ROW
BEGIN
    INSERT INTO notifications(
        phoneId, percentOfChange, price)
    VALUES
        (OLD.phoneId,
        ROUND(((NEW.price - OLD.price) / OLD.price * 100), 2),
```

```
        NEW.price);
END$$
DELIMITER ;

# bildirimler ('notifications') tablomuz şu an boş
# Bir ürünün fiyatını değiştirelim:
UPDATE prices SET price = 30000
WHERE phoneId = (SELECT id FROM phones
    WHERE brand = 'Samsung' AND model = 'S23');
# Şimdi bildirimler tablomuza bakalım:
SELECT * FROM notifications;
# id    phoneId    percentOfChange    price
#  1    1            -7.55              30000
```

Bir Tetikleyiciyi Silme

- Söz dizimi:

```
DROP TRIGGER [IF EXISTS] [veritabanıIsmi.]tetikleyiciIsmi;
```

- Veritabanı ismini yazmazsak, oturumda kullanılan seçili veritabanında işlem yapılır.
- `IF EXISTS` deyimi tetikleyici varsa silme işlemi yapar, yoksa hatâ yerine uyarı verir.

Tetikleyiciler Hakkında Diğer Bilgiler

- Bir tablo silindiğinde, tablo üzerindeki tüm tetikleyiciler de silinir.
- Bir tablo yeniden isimlendirildiğinde, ilgili isim tetikleyiciler için de tazelenir.
- Bir tetikleyici içerisinde yalnızca `IN` tipinde parametre alan bir saklı yordamı çalıştırabilirsiniz.
- Bir tetikleyici içerisindeki kod başarısız olduğunda yapılan işlem geriye sarılır. Misal, kullanıcı bir tabloya bir veri eklediğinde, e - posta atan bir tetikleyici çalışsa ve e - posta atılamasa, veri ekleme işlemi başarısız olur. Bunun böyle olmaması için tetikleyici içerisinde yaptığımız işlemlerde hatâ kontrolü yapmamız gerekebilir.
- Dikkatinizi çekmek istiyorum, bİr tetikleyici derlenmiş bir yapı değildir; bir kod parçasıdır. Dolayısıyla bir tetikleyici içerisinde vâr olmayan bir saklı yordamı çağırabilir, vâr olmayan bir tabloya veri ekleme işlemi yapabilirsiniz; hatâyı tetikleyici oluşturulurken değil, çalıştırılırken alırsınız.
- Veritabanı sunucusundaki tetikleyicileri öğrenmek için `SHOW TRIGGERS;` çalıştırlır.

- `SHOW TRIGGERS {FROM | IN} veritabaniIsmi;` deyimi ise müşahhas bir veritabanı üzerindeki tetikleyicileri öğrenmek için kullanılır.
- MySQL 5.7.2 sürümünden evvel bir tablo için en fazla 6 adet tetikleyici tanımlayabiliyordunuz. Yanî bir tabloya ekleme işleminden sonra çalıştırmak istediğiniz bir işlem varsa, bunun için `AFTER INSERT` tetikleyicinizi düzenlemeniz ve yeni kodları eklemeniz gerekiyordu. 5.7.2 sürümünden sonra ise, aynı işlem ve aynı çalıştırılma zamânı için birden fazla tetikleyici tanımlayabilirsiniz.
- Tetikleyiciler tablo üzerinde `INSERT`, `DELETE`, `UPDATE` deyimleri kullanılmadan yapılan değişikliklerde çalışmaz. Yanî `TRUNCATE TABLE` komutuyla tabloyu boşaltsanız `DELETE` tetikleyicisi çalışmaz; fakat arkaplanda bu üç deyimi kullanan ifâdelerle yapılan değişikliklerde tetikleyici çalışır. Misal, `REPLACE` ifâdesi arkaplanda `INSERT` deyimini kullandığından `REPLACE` ile veri değişikliği yapıldığında `INSERT` tetikleyicisi çalışır.
- Tetikleyiciler, 3 işlem (`INSERT`, `UPDATE` ve `DELETE`) ve iki zamân (`AFTER` ve `BEFORE`) kombinasyonlarından oluştuğu için karıştırılması muhtemeldir. Bunun için yaygın olarak kullanılan şu isimlendirme kalıbını kullanabilirsiniz: `TRG_tabloIsmi_{AFTER | BEFORE}_{INSERT | UPDATE | DELETE}` Misal, TRG_prices_AFTER_UPDATE gibi..
- Tetikleyiciler, veri dizininde depolanırlar ve iki dosyadan oluşurlar. Birincisi, 'tabloIsmi.trg' biçimindeki dosyadır; bu dosya tabloyla tetikleyicileri ilişkilendirir. İkincisi, 'tetikleyiciIsmi.trn' biçimindeki dosyadır ve bu dosya tetikleyici tanımını barındırır.
- Tetikleyici dosyaları kopyalanarak, tetikleyiciler yedeklenebilir.
- Tetikleyiciler şu ifâdeleri gövdesinde kullanamazlar: `SHOW`, `LOAD DATA`, `FLUSH`, `RETURN`, `BACKUP DATABASE`, `RESTORE`
- Tetikleyiciler hâzılanmış sorgu ('prepared statement') yapısını kullanamazlar (`PREPARE` ve `EXECUTE` deyimleri)
- Tetikleyiciler dinamik SQL cümlelerini kullanmayı desteklemezler (?).

TETİKLEYİCİ ÜZERİNE TETİKLEYİCİ...

- Bir tetikleyici tetiklendikten sonra, başka bir tetikleyicinin tetiklenmesi özelliği MySQL 5.7.2 ile sunulmuştur.
- Söz dizimi şöyledir:

```
CREATE TRIGGER yeniTetikleyici
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
FOR EACH ROW
{FOLLOWS | PRECEDES} digerTetikleyici
BEGIN
    # triggerGövdesi
END;
```

- Burada, `FOLLOWS` ifâdesi seçilirse tanımlanan bu 'yeniTetikleyici', 'digerTetikleyici'den sonra çalışır; `PRECEDES` seçilirse tam tersi..
- Tetikleyicilerin çalışma sırasını öğrenmek için şu kod kullanılabilir:

```
SELECT TRIGGER_NAME, ACTION_ORDER
FROM INFORMATION_SCHEMA.TRIGGERS
WHERE TRIGGER_SCHEMA = 'veritabanıIsmi';
```

DERLİ SAKLI KOMUT KÜMESİ, SAKLI YORDAM (STORED PROCEDURE)

- Veritabanı işlemlerinde pek çok benzer sorguyu çalıştırmamız gerekiyor. Bu, uzun sorgu cümlelerinin her seferinde veritabanı sunucusuna iletilmesi, derlenmesi anlamına gelmektedir. Bunun yerine bir sorguyu bir fonksiyon gibi isimlendirerek veritabanında saklayabilir ve belirlediğimiz isimle çalışmasını sağlayabiliriz.
- İlişkisel veritabanı sistemlerinde bunu sağlayan yapıya 'stored procedure' denir.
- Bunu Türkçe'ye 'derli saklı komut kümesi', 'saklı yordam' gibi ifâdelerle çevirebiliriz.
- MySQL'de saklı yordamlar bir oturumda ilk kez çalıştırıldıklarında diğer sorgular gibi derlenir ve çalıştırılır; fakat ikinci kez çalıştırılmak istendiklerinde önbellekteki derlenmiş hâli çalıştırılır; böylece ilgili kodlar yeniden derlenmemiş olur; performans kazancı sağlanır.
- Saklı yordam kullanmanın faydaları şunlardır:
    1. Uzun sorgu metînleri veritabanı sunucusuna iletilmediğinden ağ trafiğini azaltır
    2. Hızlı çalışır, bu komut kümeleri bir kez derlendikten sonra tekrar derlenmeye ihtiyaç duymazlar; fakat bu işlem sadece o oturum için geçerlidir; yanî her oturum için bir kez derleme işlemi icrâ edilir; bu işlem de oturum açıldığında yapılmaz, saklı yordam ilk çağrıldığında yapılır.

3. Sorguların veritabanı sunucusunun bulunduğu merkezden kontrol edilmesini sağlar. Sorgu metninde bir değişiklik yapılmak istendiğinde tüm istemcilerdeki sorgu metninin değiştirilmesine gerek kalmaz.

4. Güvenlik : Saklı yordam kullanmak, sorgu metninin istemcilerde bulunması, istemcilerin istenen tablolara erişim hakkının bulunarak olası güvenlik ihlâlinde veritabanının saldırıya açık olması, sorgu metînlerine 'SQL injection' denilen yöntemle zararlı kodların zerk edilerek veritabanında çalıştırılmasına karşı veritabanının daha savunmasız olması gibi güvenlik - gizlilik sorunlarının üstesinden kolayca gelmemizi sağlar.

- Bir saklı yordam şu şekilde tanımlanır:

```
DELIMITER $$
CREATE PROCEDURE getWriterInfoOfArticle()
BEGIN
    SELECT a.name, a.writer, w.phoneNumber
    FROM `articles` AS a
    JOIN `writers` AS w ON a.`writer` = w.`name`
    WHERE a.`name` = "Soykırıma Karşı Hamleler";
END$$
DELIMITER ;
```

- Yukarıdaki kod incelersek;

1. İlk satırda bir komut satırı sonunun belirten ifâde `$$` yapılıyor; son satırda da bu, eski ayarına döndürülüyor; bunun sebebine birazdan bakalım, inşâAllâh..

2. İkinci satırda bir saklı yordam tanımlanıyor, bu saklı yordamın ismi 'getWriterInfoOfArticle'

3. Üçüncü satırda saklı yordamın başlayacağı kod bloğu `BEGIN` ile başlatılıyor.

4. Kod bloğu içerisine yazmak istediğimiz sorgu yazılıyor.

5. `END$$` satırında da kod bloğunun bittiği belirtiliyor.

- Bu saklı yordam şu şekilde çağrılır:

```
CALL getWriterInfoOfArticle();
```

- Bir saklı yordam oluşturulurken dikkate almamız gereken şey, aynı isimde başka bir saklı yordam olması durumunda hatâ döndürülmesidir. Bu durumda bu hatâyı ele almak zorunda

kalabiliriz. Çözüm olarak `CREATE PROCEDURE` deyimini `CREATE PROCEDURE IF NOT EXISTS` deyimiyle yer değiştirebiliriz; fakat **bu ikinci deyim MySQL 8.0.29 versiyonundan önceki MySQL sürümlerinde çalışmaz!**

Satır Sonu İfâdesini Değiştirme ('DELIMITER' komutu)

- Birden fazla sorgu cümlesini peş peşe çalıştırmak istediğimizde, her sorgu cümlesinin sonuna noktalı virgül konulması gerektiğini biliyoruz.
- Bir saklı yordam içerisinde birden fazla sorgu cümlesi olduğunda derleyici saklı yordamın tanımlanması işlemini tek bir sorgu şeklinde algılayamayabileceğinden ötürü satır sonu ifâdesini değiştirmemiz ve sonra eski hâline geri getirmemiz gerekir.
- Bunu yaparak saklı yordam içerisinde yazılan kodların ayrı ayrı işleme alınması sorununun önüne geçilmiş olunuyor..
- Satır sonu ifâdesini değiştirmek için `DELIMITER` komutu kullanılır; bu komuttan sonra bir boşluk karakteri konduktan sonra yazılan ifâde satır sonu ifâdesi olur:

```
DELIMITER &&
CALL getWriterInfoOfArticle()&&
# İlgili saklı yordam çalıştırılır.
```

> ***NOT :*** Satır sonu ifâdesini değiştirirken ilgili ifâdenin yazdığınız saklı yordam içerisinde bulunmamış olmasına dikkat ediniz.

Değişkenler

- Bir fonksiyonun olmazsa olmazı değişkenlerdir. MySQL, bir saklı yordam içerisinde değişken tanımlamanıza izin verir. Basitçe şu şekilde tanımlanır:

```
DECLARE degiskenIsmi veriTipi [DEFAULT varsayilanDegeri];
```

- Tablo oluştururken kullandığımız yapıya benzemektedir.
- Eğer değişkene varsayılan değer atanmazsa, varsayılan değeri `NULL`'dır.
- Bir tanım cümlesi içerisinde aynı veri tipinde birden fazla değişken tanımlanabilir:

```
DECLARE degisken1, degisken2 veriTipi [DEFAULT varsayilanDeger];
```

- Değişken(ler)e değer atamak için `SET` deyimi kullanılır:

```
SET degiskenIsmi = deger [, degisken2 = deger2, ...];
```

- Bir değişkene dinamik olarak değer atamak da mümkündür:

```
SET degisken2 = degisken1;# Başka değişkenin değeri kopyalanır
DECLARE maxScore DEFAULT 0;
SELECT MAX(score) INTO maxScore FROM scores;
# Bir tablodaki en yüksek 'score' değeri 'maxScore'a atanır
```

- **ÖNEMLİ NOT** : Saklı yordam içerisinde `SELECT` veyâ `SET` deyiminden sonra yeni bir değişken tanımlanamaz!
- **ÖNEMLİ NOT** : Değişkenler değer tiplidirler; referans tipli değildirler.
- **ÖNEMLİ NOT** : Saklı yordam içerisinde değişken tanımlanırken, sadece isim kullanmak kâfîdir; fakat saklı yordam dışında değişken tanımlarken değişkenin sadece o oturum için geçerli olduğunu belirtmek ve derleyici tarafından sistem değişkeni olarak algılanmasını önlemek için `@` işâreti konulmalıdır. Kısaca, saklı yordam dışındaki değişkenlerinizin başına `@` işâreti koymalısınız.

Değişkenlerin kapsamı

- Programlama dillerindekine benzer şekilde saklı yordam içerisinde tanımladığınız değişkenler saklı yordamdan çıkarken bellekten silinir.
- Bir saklı yordam içerisinde birden fazla `BEGIN ... END` bloğu olabilir; böyle olduğunda `BEGIN ... END` içerisinde tanımlanan değişkenlere bu blok dışarısında erişilemez.
- Döngü içerisinde tanımlanan değişkenlere döngü dışında erişilemez.
- Farklı kapsamlarda aynı isimde değişkenler olabilir.

Kontrol Yapıları

- Kontrol yapıları içerisinde iki değerin birbirine eşit olup, olmadığını kontrol etmek için eşittir ('=') işleci kullanılır: `IF sayi = 3` (sayi 3'e eşit mi, demektir)
- **Bir değişkenin `NULL` olup, olmadığını kontrol etmek için eşitlik işleci kullanılamaz**. Bunun yerine `IS` ve `IS NOT` işleçleri kullanılır: `IF sayi IS NULL` (sayi NULL mı, demektir)

1) IF deyimi

- Tüm programlama dillerindeki en temel kontrol yapısı `IF` yapısı MySQL içerisinde aynı işlevdedir; fakat söz dizimi biraz farklıdır.
- Söz dizimi:

```
IF sart [{AND | OR} sart2 {AND | OR} sart3]
THEN
    # sart sağlandığında çalışası istenen kodlar;
[ELSEIF baskaSart [{AND | OR} baskaSart2 ...]
THEN
    # 'sart' şart sağlanmayıp, 'baskaKart' sağlandığında
    # çalışması istenen kodlar;
]
[ELSE
    # Hiçbir şart sağlanmadığında çalışması istenen kodlar;
]
END IF;# IF şartının sonunu belirtir
```

- `ELSE` deyimi ve `ELSEIF` deyimi isteğe bağlıdır, birden fazla şart yazmak da öyle...
- MySQL'de `IF` deyimi ile `IF()` fonksiyonu benzerdir, fakat birisi fonksiyondur.

2) CASE deyimi

- Diğer programlama dillerinde olduğu gibi, sadece bir değer üzerinde eşitlik kontrolü durumunda `CASE` deyimi kullanılır.
- Kodun daha okunabilir ve performanslı olması için `IF` deyimi yerine mümkünse `CASE` deyimi kullanılmalıdır.
- İki çeşit `CASE` deyimi vardır : Basit `CASE` deyimi ve aranan `CASE` deyimi
- Basit `CASE` deyiminin söz dizimi şöyledir:

```
CASE kontrolEdilenDeger
    WHEN deger1 THEN # kodlar;
    WHEN deger 2 THEN # kodlar;
    [ELSE # hiçbir değerin sağlanmadığı durumda kodlar;]
END CASE;
```

- Her ne kadar `ELSE` deyimi ihtiyârî (tercihî) olsa da, yazılması çok önemlidir; çünkü verilen değer hiçbir seçenekle eşleşmezse ve `ELSE` deyimi yoksa kod hatâ verir.
- `ELSE` bloğu içerisinde `BEGIN ... END` kod bloğu oluşturulabilir.

- `WHEN` deyimiyle `NULL` kontrolü yapılamaz; çünkü `NULL = NULL` ifâdesi geriye `FALSE` döndürmektedir.
- `CASE()` fonksiyonu ile `CASE` deyimi benzerdir, fakat birisi fonksiyondur.
- Aranan `CASE` deyimi, eşitlik dışındaki kontroller veyâ birden fazla kontrol kullanılması gereken durumlarda kullanılır. Misal, bir sayının belli bir aralıkta olup, olmadığını basit `CASE` deyimiyle tek `WHEN` deyimi kullanarak yapamazsınız. Aranan `CASE` deyimi `IF` deyimi gibidir; fakat daha okunabilirdir:

```
CASE
    WHEN sart [, sart2, ...] THEN kodlar;
    WHEN baskaSart [, baskaSart2, ...] THEN kodlar;
    [ELSE kodlar;]
END CASE;
```

- Hiçbir şart sağlanmazsa ve `ELSE` deyimi yoksa sistem hatâ verir.

> ***NOT :*** MySQL'de `CASE` deyimi içerisindeki hiçbir `WHEN` deyimi sağlanmazsa ve `ELSE` deyimi yoksa hatâ oluşur. Bunun oluşmaması için boş bir `ELSE` deyimi yazmak isteyebiliriniz; fakat `CASE` deyimi içerisinde boş bir `ELSE` deyimi yazamazsınız. Bunu yapmak için boş bir `BEGIN ... END` bloğu oluşturun:
>
> ```
> CASE degisken
>     WHEN deger THEN kodlar;
>     WHEN deger2 THEN kodlar2;
>     ELSE BEGIN END;
> END CASE;
> ```

Döngüler

- Bir işlemin bir miktara veyâ bir şarta göre tekrar etmesini sağlayan yapılardır.
- MySQL'deki döngülerin düşük seviyeli programlamaya âit bir özelliği vardır : zıplama. Döngülerin başına ve sonuna tercîhen bir anahtar kelîme koyabilir ve `LEAVE anahtarKelime;` komutuyla döngüden çıkabilirsiniz.

1) WHILE döngüsü

- Bir kodun belirtilen şart sağlandığı sürece tekrar etmesini sağlayan döngüdür.
- Söz dizimi:

```
[etiket] WHILE sart DO
    kodlar;
END WHILE [etiket];
```

2) REPEAT döngüsü

- Programlama dillerindeki 'do-while' yapısına karşılık gelir. Söz dizimi:

```
[etiket:] REPEAT
    kodlar;
UNTIL sart
END REPEAT [etiket];
```

3) LOOP döngüsü

- Sonlanma şartı belirtilmeden çalışan bir döngüdür; döngünün sonlanması döngü içerisinde yazmanız gereken kod iledir:

```
[etiket:] LOOP
    # kodlar;
END LOOP;
```

- Döngü içerisinde mevcut yinelemeyi atlayıp, yeni yinelemeye başlamak için `ITERATE` deyimi kullanılır:

```
DELIMITER $$
CREATE PROCEDURE procE6()
BEGIN
    DECLARE sayi INT DEFAULT 0;
    dongu: LOOP
        SET sayi = sayi + 1;# sayı 1 arttırılır
        IF sayi = 5 THEN ITERATE dongu; END IF;# sayi = 5 ise başa dön
        IF sayi > 4 THEN LEAVE dongu; END IF;# sayi > 4 ise çık
    END LOOP;
    SELECT sayi;
END$$
DELIMITER ;
# Çıktı:
#     sayi
#     6
```

Döngüyü terk etme özelliği

- Bir döngünün içeriden sonlandırılması için programlama dillerinin çoğunda `break` deyimi vardır. MySQL'de ise, bunun yerine düşük seviye programlama dillerinde olduğu gibi zıplama özelliği vardır.
- Döngünün başında bir etiketle döngüden çıkılabilir:

```
etiket: WHILE DO
    IF sart THEN LEAVE etiket; END IF;
END WHILE[etiket];
```

- Döngünün sonuna yazmak ihtiyârî (tercihî) dir.
- Etiketi saklı yordamın başına yazarsanız, `LEAVE` ile çağrıldığında saklı yordam sonlandırılır:

```
DELIMITER $$
CREATE PROCUDURE procName():
proc: BEGIN
    # kodlar..
    LEAVE proc;# Saklı yordamdan çıkılır
    # diğer kodlar : Buradaki kodlar çalışmaz
END$$
DELIMITER ;
```

Saklı Yordamı Parametre İle Çağırmak

- Saklı yordamların birer fonksiyon gibi kullanılabilmesi için parametre kabûl etmesi önem arz eder. Bu, saklı yordamı daha esnek ve kullanışlı yapan bir yapıdır. Bir fonksiyonun olmazsa olmazı parametreler MySQL saklı yordam yapısında üç farklı kapsamda tanımlanabilir. Parametre çeşidini parametre isminden önce yazmalıyız:

    1. `IN` : Varsayılan parametredir. Bu parametre saklı yordamın her çağrılmasında verilen değer olarak yeniden üretilir ve saklı yordamın çalışması sona erdiğinde bellekten silinir. Eğer saklı yordam çağrılırken IN ile tanımlanmış bir parametreye bir değişken verilirse ve saklı yordam içerisinde verilen parametre değiştirilirse, saklı yordamın çağrıldığı yerdeki değişkeninin değeri değişmez:

```
DELIMITER $$
CREATE PROCEDURE proc1(name VARCHAR(50))
BEGIN
    SET name = "Âkif";
END$$
DELIMITER ;

SET @str = "Mehmed";
CALL proc1(@str);
SELECT @str;#'Mehmed' sonucunu döndürür.
```

2. `OUT` : Bu parametre tipi kullanıldığında, saklı yordama parametre olarak verilen değişken değiştirildiğinde değişiklikler ilgili değişken üzerinde görülebilir; fakat OUT parametresiyle gönderilen değişkenin değeri saklı yordam içerisinde okunamaz:

```
DELIMITER $$
CREATE PROCEDURE proc1(name VARCHAR(50))
BEGIN
    SET name = "Âkif";
END$$
DELIMITER ;

SET @str = "Mehmed";
CALL proc1(@str);
SELECT @str;#'Mehmed' sonucunu döndürür.
```

3. `INOUT` : İki parametre çeşidinin bir kombinasyonudur. Gönderilen parametre saklı yordam içerisinde okunabilir (`IN` özelliği), yapılan değişiklikler saklı yordama gönderilen değişken üzerinde görülebilir (`OUT` özelliği):

```
DELIMITER $$
CREATE PROCEDURE proc(INOUT name VARCHAR(50))
BEGIN
    SET name = CONCAT(name, " ", "Âkif");# Metînleri birleştirir
END$$
DELIMITER ;

SET @str = "Mehmed";
CALL proc(@str);
SELECT @str;# 'Mehmed Âkif' sonucunu döndürür.
```

> ***SORU :*** Sizce saklı yordamlar için neden iki yerine üç çeşit parametre var?

Saklı Yordamın Silinmesi

- Bir saklı yordamın silinmesi kodunun sözdizimi şu şekildedir:

```
DROP PROCEDURE proc3;# 'proc3' isimli saklı yordam silinir.
```

- Yukarıdaki kodun çalıştırıldığı veritabanında 'proc3' isimli bir saklı yordam yoksa hatâ alınır : Error Code: 1305. PROCEDURE classicmodels.ae does not exist

- Bu hatânın alınmaması için aynı işlemi şu kod ile yapabiliriz; bu kod ilgili saklı yordam olmadığında hatâ değil, uyarı verir; dolayısıyla bir uygulama içerisinde bu kod çalıştırıldığında hatâ fırlatılmaz; uygulama 'catch' bloğuna düşmez!:

```
DROP PROCEDURE IF EXISTS proc3;# 'proc3' saklı yordamı varsa silinir.
```

> ***NOT :*** **DROP PROCEDURE IF EXISTS** deyiminin çalışması için MySQL'in 8.0.29 ve üstü sürümünün olmasına gerek yoktur.

- **Bir saklı yordamı değiştirmenin en uygun yolu, onu silip yeniden oluşturmaktır.**

Saklı Yordamları Listeleme

- Söz dizimi:

```
SHOW PROCEDURE STATUS [LIKE 'oruntu' | WHERE sart];
```

- Bu kod veritabanı sunucusundaki tüm saklı yordamları listeler.

- Belli bir veritabanındakileri listelemek istiyorsak, şart cümlesinde belirtmeliyiz:

```
SHOW PROCEDURE STATUS WHERE Db = 'veritabanıIsmi';
```

- Saklı yordamlar hakkındaki tüm bilgiler `INFORMATION_SCHEMA` veritabanı altındaki `ROUTINES` tablosundadır; `WHERE` şartında `ROUTINE_TYPE = 'PROCEDURE'` yazmayı unutmayın.

Görünüm (VIEW) ile Saklı Yordam Arasındaki Farklar

1. Saklı yordam parametre alır; görünüm ise alamaz.

2. Saklı yordam içerisinde döngü gibi programlama elemanları kullanılabilir, görünüm içerisinde şart kontrol yapılarıyla aynı işlevde olan `IF()` gibi fonksiyonlar yer alabilir; programlama yapıları yer alamaz.
3. **Görünüm bir tablo döndürür, saklı yordam ise birden fazla sonuç kümesi (tablo) döndürebilir.**
4. Saklı yordamlar geriye dönen sonuç kümelerini kendi içinde belirleyebilir.

MYSQL TEMEL FONKSİYONLAR

Toplam Fonksiyonları

- Bu fonksiyonların bâzısına evvelce değinilmişti. Bunlar tablo üzerinde genellikle tüm satırların işleme tutulduğu ve sonucunda özel bir bilginin elde edildiği türden fonksiyonlardır. Fonksiyonun girdi verisi birden fazladır, çıktı verisi tek bir veridir.
- Toplam fonksiyonlarının sık kullanıldığı bir durum vardır; o da verilerin tümünün fonksiyona girmesi yerine münferid (tekil) olanların fonksiyona girmesini sağlayan `DISTINCT` deyimi:

```
fonksiyon([DISTINCT | ALL] ifade);
# Varsayılan olarak ALL geçerlidir
```

1) AVG

- Avarage (ortalama) kelîmesinin kısa hâlidir; sütun ortalamasını verir.

2) SUM

- Summary (toplam, özet) kelîmesinin kısa hâlidir; verilen sütunun toplamını verir.

3) MIN

- Minimum (asgarî) kelîmesinin kısa hâlidir; sütundaki en küçük değeri verir.

4) MAX

- Maximum (azamî) kelîmesinin kısa hâlidir; sütundaki en büyük değeri verir

5) COUNT

- 'Saymak' demektir; sütundaki satır sayısını sayar. Sadece münferit ('tekil') verilerin sayılması isteniyorsa, `COUNT(DISTINCT ...)` biçiminde kullanılmalıdır.

6) GROUP_CONCAT

- Gelen birden fazla sonucu, belirtilen ayraçla birleştirip, tek bir değer hâline getirir:

```
GROUP_CONCAT([DISTINCT] ifade [ORDER by siralamaIfadesi]
            [SEPERATOR 'ayirici']);
# Yukarıdaki ifâdelerin aralarında virgül olmadığına dikkat ediniz
# Misal:
CREATE TABLE names(name VARCHAR(20));
INSERT INTO names VALUES('Ali'), ('Hasân'), ('Cemil'), ('Ferhat');
SELECT GROUP_CONCAT(
    DISTINCT name
    ORDER BY name
    SEPARATOR '|') AS hepsi
FROM names;
# hepsi
# Ali|Cemil|Ferhat|Hasân
```

- `GROUP_CONCAT()` `NULL` değerleri ihmâl eder.
- Dönen sonuç girdinin tipine göre ikili veyâ ikili olmayan veri tipinde olabilir.
- Dönen sonucun azamî uzunluğu 1024'tür; fakat bu değer değiştirilebilir:

```
SELECT @@session.group_concat_max_len;# Oturum düzeyindeki ayarlanan
SELECT @@global.group_concat_max_len;# Sunucu düzeyindeki varsayılan
# Bu uzunluğu oturum düzeyinde değiştirmek için:
SET @@session.group_concat_max_len = 2048;
```

7) BIT_AND(), BIT_OR() ve BIT_XOR()

- Eşit uzunluktaki iki ikili veriyi veyâ ikili sayıyı (`INT` olarak tanımlanması sıkıntı değil) bit düzeyinde VE(`AND`) işlemine tabi tutar; hepsi de `1` sonucunu üretiyorsa `1` döndürür, aksi hâlde `0` döndürür. Satırların her biri eşit uzunlukta olmalıdır; yoksa farklı bir sonuç ortaya çıkar.
- BIT_OR() fonksiyonunun tek farkı verilerin VEYÂ (`OR`) kapısına sokulmasıdır.
- BIT_XOR() fonksiyonu ise verileri ÖZEL VEYÂ (`XOR`) işlemine tâbî tutar.

8) COUNT(IF())

- Sayılacak satır sayısının belli bir koşula göre yapılmasını sağlamak için kullanılan yapıdır.

- `IF()` fonksiyonu yapısı : `IF(ifade, trueDegeri, falseDegeri)` Eğer ifâde `TRUE` sonucunu üretiyorsa, trueDegeri, `FALSE` sonucunu üretiyorsa `FALSE` değeri döndürülür.
- `COUNT(IF())` yapısı pek çok durumda iç sorgu ('subquery') yerine kullanılabilir.
- Misal:

```
# IF örneği:
SELECT IF(sayi = 3, 1, NULL);# sayi 3 ise 1, değilse NULL döndürür

# COUNT(IF()) örneği:
CREATE TABLE sales(saleDate DATE, amount int);
INSERT INTO sales VALUES
    ('20250312', 7),('20250314', 8), ('20250315', 1), ('20250312', 9);
SELECT COUNT(
    IF(saleDate = '20250312', 1, NULL)
) AS soldOn12ndDay FROM SALES;
# 12 Mart 2025'te yapılan satış sayısı (satır sayısı) döndürülür
# soldOn12ndDay
# 2
```

9) SUM(IF())

- Toplam fonksiyonu içerisine tüm satırların dâhil edilmesini istemediğimiz, belli bir filtre uygulamak istediğimiz durumlarda kullanılabilir.
- Misal:

```
CREATE TABLE nums(sayi INT);
INSERT INTO nums VALUES (12), (11), (14), (5), (3), (6), (2);
SELECT SUM(IF(sayi > 10, sayi, NULL)) FROM nums;# 37 döndürür
SELECT SUM(IF(sayi > 10 AND sayi %2 = 1, sayi, NULL)) FROM nums;# 11
```

- `IF` fonk.'u içerisinde `AND` ve `OR` ifâdeleriyle başka şartlar da belirtilebiliyor.

10) STD, STDDEV, STDDEV_POP

- Standart sapma verilerin ortalamadan ne kadar saptığını gösteren bir matematiksel ölçü birimidir ve varyansın kareköküdür. Kümedeki tüm elemanların standart sapmaya katılması 'popülasyon standart sapma' olarak isimlendirilirken, küme elemanlarının bir örneklemesinin standart sapma işlemine tâbî tutulması 'örneklemeli standart sapma' olarak isimlendirilir.
- `STD()`, `STDDEV()` ve `STDDEV_POP()` ayni işlevi yapar; standart sapmayı hesaplar.

- `STDDEV_SAMP()` ise örneklemeli standart sapmayı hesaplamak için kullanılır.
- Söz dizimi : `STD(sutunIsmi, [virgulSonrasıBasamakHassasiyeti])`
- Örneklemeli standart sapma için sütun ismi yerine ifâde yazılır; misal, `WHERE` filtresi uygulamak için bir iç sorgu yazılabilir.

11) VARIANCE, VAR_POP

- Aaynı işlevi yaparlar; `VAR_POP()` fonksiyon isminin olması SQL standardı gereğidir.
- Varyans, verilerin ortalamadan sapmalarının karelerinin toplamı olarak bir dağılım ölçüsüdür.
- Söz dizimi : `VARIANCE(sutunIsmi)`
- Virgülden sonraki basamak hassasiyeti için `CAST()` fonksiyonu kullanılabilir.

12) VAR_SAMP

- Örneklemeli varyans hesaplamak için kullanılır. Söz dizimi : `VAR_SAMP(sutunIsmi)`

13) JSON_OBJECTAGG ve JSON_ARRAYAGG

- `JSON_OBJECTAGG(key, value)` fonksiyonu verilen sütun isimlerini bir anahtar değer ikilisine dönüştürür ve JSON metni biçiminde döndürür.
- Misal, kullanıcıların farklı oturumları var ve siz oturum ismine göre gruplandırmayı biliyorsunuz; fakat bu veriyi bir API'ya sağlarken JSON'a çevirmek isteyebilirsiniz; bu durumda `JSON_OBJECTAGG(sessionIP, connectedDevice)` gibi bir fonksiyonla id'e göre her biri gruplanmış oturum IP adresi ve bağlı cihaz bilgilerini JSON formatında alabilirsiniz. Her bir satır, farklı bir id'e karşılık gelir; tabi bunun için tablonuzda id gibi birincil anahtar sütununuzun olması gerekir; id, sütununu da SELECT içerisinde seçmelisiniz.
- Güzel bir misal, yukarıdaki senaryoyu yapmaya çalışalım:

```
# Kullanıcı tablosu oluşturalım:
CREATE TABLE users(
    id int PRIMARY KEY AUTO_INCREMENT,
    userName VARCHAR(100) NOT NULL UNIQUE);
# Bağlantı tablosu oluşturalım:
CREATE TABLE connections(
    id INT PRIMARY KEY AUTO_INCREMENT,
    userId INT NOT NULL,
    deviceName VARCHAR(50) NOT NULL UNIQUE,
    sessionIp INT UNSIGNED NOT NULL UNIQUE,
    FOREIGN KEY(userId) REFERENCES users(id) ON DELETE RESTRICT);
# Kullanıcı ve bağlantı ekleyelim:
INSERT INTO users(userName) VALUES ('Akif'), ('Mehmed');
INSERT INTO connections (userId, deviceName, sessionIp) VALUES
    ((SELECT id FROM users WHERE userName = 'Akif'),
        'Samsung A20', INET_ATON('127.0.0.1')
    ),
    ((SELECT id FROM users WHERE userName = 'Akif'),
        'General Mobile Tab5', INET_ATON('127.0.0.2')
    ),
    ((SELECT id FROM users WHERE userName = 'Mehmed'),
        'Toshiba old', INET_ATON('127.0.0.3')
    );
# Verilere bir bakalım:
SELECT
    (SELECT userName FROM users WHERE id = userId) as kullaniciIsmi,
    deviceName as CihazIsmi,
    INET_NTOA(sessionIp) AS IP
FROM connections;
# kullaniciIsmi    CihazIsmi             IP
# Akif             Samsung A20           127.0.0.1
# Akif             General Mobile Tab5   127.0.0.2
# Mehmed           Toshiba old           127.0.0.3

# NOT : INET_ATON ve INET_NTOA IP adreslerinin karşılaştırılabilir
# ve daha verimli tutulması için seçilen INT UNSIGNED tipine
# dönüştüren ve oradan geri metîn görünümüne getiren yrd. fonksiyonlar

# Şimdi asıl anlatmak istediğimize gelelim:
SELECT
    (SELECT userName FROM users WHERE id = userId) as kullaniciIsmi,
    JSON_OBJECTAGG(deviceName, INET_NTOA(sessionIp)) as JSON
FROM connections GROUP BY kullaniciIsmi;
# kullaniciIsmi     JSON
# Akif              {"Samsung A20": "127.0.0.1",
#                   "General Mobile Tab5": "127.0.0.2"}
# Mehmed            {"Toshiba old": "127.0.0.3"}
```

- `JSON_ARRAYAGG(sutun)` ise verdiğiniz alandaki verilerin bir JSON dizisi içerisinde toplanmasını sağlar.
- Aynı misal üzerinden gidecek olursak;

```
# Tüm açık bağlantıların IP adreslerini JSON dizisi olarak döndürür:
SELECT
    JSON_ARRAYAGG(INET_NTOA(sessionIp)) AS acikBaglantilar
FROM connections;
# acikBaglantilar
# ["127.0.0.1", "127.0.0.2", "127.0.0.3"]
# Açık bağlantıların IP adreslerini, kullanıcı ismine göre döndürür:
SELECT
    (SELECT userName FROM uToC WHERE id = userId) as kullaniciIsmi,
    JSON_ARRAYAGG(INET_NTOA(sessionIp)) AS acikBaglantilar
FROM connections GROUP BY kullaniciIsmi;
# kullaniciIsmi        acikBaglantilar
# Akif                 ["127.0.0.1", "127.0.0.2"]
# Mehmed               ["127.0.0.3"]
```

Şart Kontrol Fonksiyonları

- Saklı yordam içerisinde kullanabildiğimiz `IF`, `CASE` gibi yapıların fonksiyon hâlidir.

1) IF

- Yapısı : `IF(ifade, trueDegeri, falseDegeri)` Eğer ifâde `TRUE` sonucunu üretiyorsa, trueDegeri, `FALSE` sonucunu üretiyorsa `FALSE` değeri döndürülür.

2) IFNULL

- Yapısı : `IFNULL(ifade1, ifade2)` ifade1 `NULL` değilse ifade1 döndürülür, aksi hâlde ifade2 döndürülür. `IFNULL(name, 'BOŞ')` : 'name' `NULL` değilse kendisi, `NULL` ise 'BOŞ' döndürülür.

3) NULLIF

- Yapısı : `NULLIF(ifade1, ifade2)` ifade1 = ifade2 ise `NULL` döndürülür, değilse ifade1 döndürülür.

4) CASE

- `CASE` fonksiyonu `CASE` ifâdesine çok benzemektedir; fakat aralarında çok temel bir fark vardır. `CASE` fonksiyonu verilen değerin uyduğu seçenek ne döndüreceğini belirtirken,

`CASE` ifâdesi verilen değerin uyduğu seçenek için hangi kodların çalıştırılacağını belirtir.

- `CASE` ifâdesinin aksine, `CASE` fonksiyonu `SELECT` ifâdesi içerisinde kullanılabilir.
- Birinci kullanım için söz dizimi:

```
SELECT CASE deger
    WHEN secenek1 THEN secenek1IcinDeger
    [WHEN secenek2 THEN secenek2IcinDeger
    ...]
    [ELSE varsayilanDeger]
END
FROM tabloIsmi;
```

- Yukarıda, 'deger' değeri 'secenek1'e eşitse, 'secenek1IcinDeger' seçilir, 'secenek2IcinDeger' seçilir, eğer `ELSE` kısmı yazılmışsa ve 'deger' `WHEN` ile belirtilen seçeneklere uymamışsa, 'sartlarSaglanmadiysaDeger' seçilir.
- İkinci kullanım söz dizimi:

```
SELECT CASE
    WHEN sart1 THEN deger1
    [WHEN sart2 THEN deger2
    ...]
    [ELSE varsayilanDeger]
END
FROM tabloIsmi;
```

- Buradaki kullanımda ise, bir değerin hangi şartlara uyup, uymadığı kontrol edilmiyor; `WHEN` belirtilen şart(lar) sınanıyor ve sağlanan şartın karşılığı olan değer seçiliyor; eğer hiçbir şart sağlanmıyorsa ve `ELSE` kısmı yazıldıysa 'varsayilanDeger' seçilir.

Matematik Fonksiyonları

- `ABS(sayi)` : Verilen sayı değerinin mutlak değerini geri döndürür
- `CEIL(sayi)` : Verilen sayıda sıfırdan farklı kesri varsa sayı yukarı yuvarlanır. `CEIL` yerine `CEILING` de yazılabilir. Verilen değer `NULL` ise, geriye `NULL` döndürür.
- `FLOOR(sayi)` : Verilen sayının sıfırdan farklı kesri varsa, sayı aşağı yuvarlanır. Verilen değer `NULL` ise, geriye `NULL` döndürür.
- `MOD(sayi, bolen)` : Verilen sayının verilen bolen'den kalanını döndürür.

- `ROUND(sayi, [kesirBasamakSayisi])` : Verilen sayı, verilen kesir basamak sayısı kadar basamak sayısı olacak şekilde yuvarlanır; eğer kesir basamak sayısı verilmezse, sayı tam sayıya yuvarlanır.
- `RAND([tohum])` : 0 ile 1 arasında rastgele sayı döndürür; sıfır dâhildir, 1 dâhil değildir; eğer bir tohum ('seed') verilirse, hep aynı sayı üretilir. Bulunan bir rastgele değerin, başka yerde yeniden kullanılması için tohum kullanılması çok uygundur.
- `TRUNCATE(sayi, silinecekKesirSayisi)` : Verilen sayının verilen sayıda basamak değerini siler (yuvarlamaz).
- `PI()` : Pi sayısını döndürür (3.141593 döndürür).
- `POWER(sayi, derece)` : Verilen sayının verilen dereceden üssünü alır, `POWER` yerine `POW` da yazılabilir.
- `SIGN(sayi)` : Verilen sayının işâretini döndürür; sayı negatifse -1, sayı pozitifse +1, sayı 0 ise 0 döndürür.
- `SQRT(sayi)` : Verilen sayının karekökünü döndürür.
- `SIN(derece)` : Verilen derece için sinüs değerini yüksek hassâsiyyetle döndürür.
- `COS(derece)` : Verilen derece için kosinüs değerini yüksek hassâsiyyetle döndürür.
- `TAN(derece)` : Verilen derece için tanjant değerini yüksek hassâsiyyetle döndürür.
- `COT(derece)` : Verilen derece için kotanjant değerini yüksek hassâsiyyetle verir.
- `LOG2(sayi)` : Sayının iki tabanında logaritma değerini verir.
- `LOG10(sayi)` : Sayının on tabanında logaritma değerini verir.
- `LOG(n)` : Sayının n tabanında logaritma değerini verir.
- `EXP(derece)` : e sayısının verilen dereceden üssünü verir.
- `DEGREES(radyan)` : Verilen radyanı dereceye çevirir
- `RADIANS(derece)` : Verilen dereceyi radyana çevirir
- `ATAN(deger)` : Verilen değerin ark tanjant değerini yüksek hassâsiyyetle verir.
- `ASIN(deger)` : Verilen değerin ark sinüs değerini yüksek hassâsiyyetle verir.
- `ACOS(deger)` : Verilen değerin ark kosinüs değerini yüksek hassâsiyyetle verir.

- `LN(deger)` : Verilen değerin doğal logaritmasını verir
- `FORMAT(sayi, kesirBasamakSayisi, bolge)` : Sayıyı belirtilen biçimde ayırmak için kullanılır. Sayı, belirtilen kesir basamak sayısı olacak şekilde yuvarlama / sıfır koyma işlemi yapar; verilen bölgeye uygun olarak binler, milyonlar basamağını ayırır; bölge verilmeyebilir. `SELECT FORMAT(1245483.16, 3);` : 1,245,483.160 `SELECT FORMAT(1245483, 2);` : 1,245,483.00
- `CRC32()` : ...

Metîn Fonksiyonları

- `CONCAT(metin1, [metin2, metin3, ...])` : Verilen metînleri birleştirir; eğer metînlerden birisi `NULL` ise, geriye `NULL` döndürür.
- `CONCAT_WS(separator, metin1, [metin2, ...])` : Verilen metînleri birleştirir; onları birleştirirken aralarına ayırıcı ('separator') koyar. Fonksiyon ismi 'concat_with_separator' ifâdesinin kısaltımıdır. `NULL` değeri ihmâl eder, yoksayar.
- `INSTR(metin, arananAltMetin)` : Metin içerisinde alt metni arar, bulduğunda başlangıç yerini döndürür; fakat programlama dillerindekinin aksine indeks 1'den başlar. Eğer metîn içerisinde verilen alt metîn yoksa 0 döndürür.
- `LENGTH(metin)` : Bir metnin uzunluğunu BYTE cinsinden verir.
- `CHAR_LENGTH(metin)` : Metnin uzunluğunu karakter cinsinden verir.
- `LEFT(metin, miktar)` : Verilen metnin solundan miktar kadar harf alır, döndürür.
- `RIGHT(metin, miktar)` : Metnin sağından miktar kadar harf alır ve döndürür.
- `LOWER(metin)` : Metnin tüm harflerini küçük harfe çevirir.
- `UPPER(metin)` : Metnin tüm harflerini büyük harfe çevirir.
- `LOCATE(altMetin, metin, baslangicMevkisi)` : Verilen metîn içerisinde baslangicMevkisi ile belirtilen indeksten (dikkat, metînlerde indeks 1'den başlar) itibâren alt metni arar; bulursa başlangıç indeksini verir, aksi hâlde 0 döndürür.
- `LTRIM(metin)` : Metnin solundaki (başındaki) boşlukları kaldırır.
- `RTRIM(metin)` : Metnin sağındaki (sonundaki) boşlukları kaldırır.
- `TRIM(metin)` : Metnin başındaki ve sonundaki boşlukları kaldırır

- `REPLACE(metin, altMetin, yeni)` : Verilen metindeki verilen alt metni yeni değerle değiştirir; eğer altMetin değerinden birden fazla varsa, hepsini değiştirir.
- `REPEAT(metin, tekrarAdedi)` : Metni tekrar adedince tekrar edip, döndürür
- `REVERSE(metin)` : Verilen metni ters çevirir.
- `SUBSTRING({metin, baslangic} | {metin FROM baslangic})` : Verilen metnin verilen başlangıç indeksinden itibâren alt metnini oluşturur. İndeks negatif sayı olabilir. MySQL'de metînlerde indeks olarak kullanılan numara 1'den başlar.
- `SUBSTRING_INDEX(metin, ayrac, ayracIndeksi)` : Metîn içerisinde, ayracı arar; 'ayracIndeksi' ile belirtilen sıradaki 'ayrac'a kadar olan metîn alınır. Misal: `SELECT SUBSTRING_INDEX('akşamm', 'm', 1);` : 'akşa' `SELECT SUBSTRING_INDEX('akşamm', 'm', 2);` : 'akşam'
- `FIND_IN_SET(arananKarakterDizisi, metinler)` : Virgülle ayrılmış metînlerin içerisinde aranan karakter dizisini arar, bulduğu elemanın eleman numarasını döndürür (eleman numarası 1'den başlar). Fonksiyon, elemanlar arasında tam karakter eşleşmesi arar; dolayısıyla virgülden sonra boşluk karakteri (' ') dahî olsa engeldir. Ayrıca aranan karakter dizisi, virgül içermemelidir. Eğer aranan karakter dizisi elemanlar içinde bulunamadıysa geriye `0` döndürür.
- `ASCII(harf)` : Verilen harfin ASCII karşılığını döndürür.
- `LPAD(metin, toplamUzunluk, doldurmaKarakterDizisi)` : Metîn belirtilen toplam uzunluğa gelene dek metnin başına 'doldurmaKarakterDizisi' eklenir; eğer toplam uzunluk metnin uzunluğundan kısa ise metîn kırpılmış olarak döndürülür. UTF-8 ve UTF-16 ile düzgün çalışır.
- `RPAD(metin, toplamUzunluk, doldurmaKarakterDizisi)` : Metîn belirtilen toplam uzunluğa gelene kadar metnin sonuna doldurma karakter dizisi eklenir. Toplam uzunluk metnin uzunluğundan kısa ise, metîn kırpılmış olarak döndürülür. UTF-8 ve UTF-16 ile düzgün çalışır.
- `STRCMP()` : İki metni karşılaştırırken kullanılır.

Târih - Saat Fonksiyonları

- MySQL târih ve saat fonksiyonlarının önemli bir kısmının bağlı olduğu saat bölgesi ('time zone')ni değiştirmek için `SET time_zone = bolge;` komutu kullanılabilir.
- Târih fonksiyonları kullanılırken târih değeri metîn olarak verilmelidir. Bu metnin formatı ya `'YYYY-MM-DD HH:MM:SS.MSMSMS'` biçiminde olmalıdır; yâ dâ `'YYYYMMDD`

`HH:MM:SS.MSMSMS'` biçiminde olmalıdır. Târih için örnek : '2025-04-14' veyâ '20250414' Saat için örnek : '12:43:17.443453' (saniyeden sonrası yazılmayabilir) Târih-saat için : '2025-04-14 12:43:17' veyâ '20250414 12:43:17' (mili sn yazılabilir)

- Târih saat fonksiyonlarının bir kısmı belirli bir format değerinde târih ifâdesine izin verir. Bu format örüntü metnini oluştururken zamân birimlerini ne ile ifâde edeceğimizi bilmemiz gerekiyor:

%Y : sene('year')	%m : ay ('month')	%d : gün ('day')
%h : saat ('hour')	%i : dakîka ('second')	%s : saniye ('second')

- Yukarıdaki ifâdelerle dilediğiniz düzende format metni oluşturabilirsiniz: `'%Y-%m-%d %h:%i%s'` : '2025-03-31 06:34:11' gibi târihlerle eşleşir, onları ifâde eder `%Y,%m,%d` : '2025,01,23' gibi târihlerle eşleşir / onları ifâde eder. `%Y%m%d` : '20250514', `%Y--%m--%d %h.%i.%s` : '2024--12--21 11.54.34' ...

- Burada, zamân formatlamayla ilgili, farklı zamân dilimleriyle çalışmayla ilgili temel bilgiler bulunmaktadır; daha fazlası için hâzırlayacağım veritabanı idâreci notları defterinde değineceğim, inşâAllâh.

- `CURDATE()` : Şu anki târih bilgisini metîn olarak getirir. `CURRENT_DATE` , `CURRENT_DATE()` isimleriyle de çağrılabilir. Gelen verinin sayısal olarak gelmesi için `CURDATE() + 0` biçimi kullanılabilir.

- `CURTIME([hassasiyet])` : Şu anki saat bilgisini metîn olarak getirir. `CURRENT_TIME()` ve `CURRENT_TIME` isimleriyle de çağrılabilir. Saniyeden sonra kaç basamak hassâsiyyet isteniyorsa girdi olarak verilebilir.

- `NOW()` : Şu anki târih ve saat bilgisini metîn olarak getirir. Gelen veriyi sayısal olarak almak için `CURDATE() + 0` kullanım biçimi tercih edilebilir. `CURRENT_TIMESTAMP()` ve `CURRENT_TIMESTAMP` isimleriyle de çağrılabilir. Bu fonksiyona belli miktarda târih ekleyip, oluşan yeni târih - saat değerini elde etmek için `INTERVAL` ifâdesini kullanmalısınız. `INTERVAL` ifâdesi 'aralık' anlamına gelir ve hangi târih / saat birimine göre ekleme / çıkarma yapmak istediğinizi belirtmenizi sağlar. Misal, `SELECT NOW() + [INTERVAL] 1 DAY;` kodu tam bir gün sonrasının târih-saat değerini getirir. Bunun gibi diğer aralık isimlerini kullanarak şu anki vakitten farklı uzaklıklardaki vakti elde edebilirsiniz, Allâh'ın izniyle. Eğer `INTERVAL` anahtar kelîmesini kullanmazsanız, sonuç sayısal olarak döndürülür: `YEAR` : Sene , `MONTH` : Ay, `WEEK` : Hafta, `DAY` : Gün, `HOUR` : Saat, `MINUTE` : Dakika, `SECOND` : Saniye , `MICROSECOND` : Mikrosaniye `NOW()` fonksiyonu târih ve saat tipindeki sütunların varsayılan değeri için kullanılabilir: `updatedUser DATETIME DEFAULT NOW() ON UPDATE NOW();`

- `SYSDATE([hassasiyet])` : Şu anki târih ve saat bilgisini metîn olarak verir. Eğer sayısal olarak istiyorsanız `SYSDATE() + 0` biçiminde kullanmalısınız. Saniyeden sonra kaç basamak hassâssiyyetinde veri istediğinizi girdi olarak belirtebilirsiniz. Bu fonksiyonu kullandığınızda, kullandığınız kısımda indeks çalışmaz; bu fonksiyon `SLEEP()` ile sistemi bekletmeniz durumunda bekleme süresinden sonraki vakti göstermez.

- `UTC_TIMESTAMP()` : `CURRENT_TIMESTAMP()` fonksiyonuyla aynı özelliklere sâhiptir; tek farkı mevcut târih - saati sunucu zamân dilimine göre değil, 'UTC +0' zamân dilimine göre getirmesidir.

- `UTC_DATE()` : `CURDATE()` ile aynı özelliklere sâhiptir; tek farkı mevcut târihi sunucu zamân dilimine göre değil, 'UTC+0' zamân dilimine göre getirmesidir.

- `UTC_TIME()` : `CURTIME()` ile aynı özelliklere sâhiptir; tek farkı mevcut saati sunucu zamân dilimine göre değil, 'UTC+0' zamân dilimine göre getirmesidir.

- `ADDTIME(tarihsaatMetni, zaman)` : Metîn olarak verilen tarih saat değerine verilen zamânı ekler. Zamânın formatı saat formatında 'HH:MM:SS' şeklinde olmalıdır; fakat eğer milisaniye eklenmek istenirse, saniye değerinden sonra nokta koyup, 6 hâneye kadar milisaniye değeri eklenebilir. Parametreler metîn olmalıdır. Eğer, verilen değerden belli miktar zamân değerini çıkarmak istiyorsak, ikinci parametrenin başına (metîn içinde olacak şekilde) eksi işâreti koymalıyız.

- `DATE_ADD(tarih, INTERVAL eklenecekTarih birim)` : Verlen târihe, verilen birinde 'eklenecekTarih'i ekler. Birimler târih birimi olmalıdır, YEAR, MONTH, WEEK, DAY olabilir.

- `DATE_SUB(tarih, INTERVAL cikarilacakTarih birim)` : DATE_ADD() ile benzer şekilde çalışır, fakat bir târihten başka târih değerini çıkarmak için kullanılır.

- `DATEDIFF(tarih1, tarih2)` : Verilen târih değerinden, ikinci parametreyle verilen târih değerini çıkarır, yanî arasındaki farkı gün olarak döndürür. Parametreler `DATETIME` tipinde olabilir.

- `TIMEDIFF(zaman1, zaman2)` : İki zamân arasındaki saat cinsinden döndürür, parametreler `DATETIME` tipinde olabilir; birinci zamân değerinden ikincisi çıkarılır. Eğer aralarında gün farkı varsa ihmâl edilmez, gün farkı saate çevrilir.

- `TIMESTAMPADD(birim, deger, tarih)` : Verilen târih değerine verilen birimdeki değer eklenir. Misal, `TIMESTAMPADD(DAY, 1, '2025-04-14')` : Târihe bir gün ekler Buradaki târih değeri `DATETIME` formatındaki bir metîn verisi de olabilir. Değer negatif bir sayı verilirse verilen târih değerinden verilen değer çıkarılır.

- `TIMESTAMPADD(birim, deger, tarih)` : `TIMESTAMPADD()` fonksiyonuyla benzer özellikte çalışır; verilen değer ilgili târihten çıkarılır.
- `TIME_TO_SEC(zaman)` : Verilen zamân verisi saniye verisine çevrilir.
- `TO_DAYS(tarih)` : Verilen târih verisini güne çevirir; târih verisinin başlangıcı miladi takvime göredir; yanî miladi 0'dan verilen târihe kadar geçen gün sayısını verir.
- `FROM_DAYS(günSayisi)` : Verilen gün sayısını mîlâdî 0 senesinden itibâren sayarak hedef târihi bulur ve döndürür.
- `STR_TO_DATE(târih, format)` : Girdiler metîn tipindedir, çıktı ise verilen târih metninin `DATE`, `TIME` veyâ `DATETIME` tipine çevrilmiş hâlidir. Girdi hatâlı ise `NULL` döndürülür. Buradaki târih metni çok biçimli olabilir; dikkat edilmesi gereken şey, verilen format metninin târih metniyle uyumlu olmasıdır. Format metni, hangi zamân biriminin önce eşleştirileceğini belirten bir özel düzenli ifâdedir. Format metnini oluşturmayı öğrenmek için bu bölümün başına bakın. Misal, `SELECT STR_TO_DATE('2025-04-14 12:37:43', '%Y-%m-%d %h:%i:%s');` `SELECT STR_TO_DATE('20250414 12:37:43', '%Y%m%d %h:%i:%s');` `SELECT STR_TO_DATE('2025,04,14 12:37:43', '%Y,%m,%d %h:%i:%s');`
- `FROM_UNIXTIME(saniyeDegeri, [format])` : Linux târih değeri UTC zamân diliminde '1970-01-01 00:00:00' târihinden bu zamâna kadar geçen saniye değerini ifâde etmektedir. Bu saniye değerini verip, târihi almak için bu fonksiyon kullanılır. Format değeri zorunlu değildir; format ifâdeleri ve seçilen ayraçlar kullanılarak oluşturulabilir.
- `UNIX_TIMESTAMP([tarih])` : Verilen târih değerini Unix Timestamp değeri olarak ifâde edilen, 1970'den bu yana olan saniye değeri cinsine çevirir. Eğer bir târih değeri verilmezse, şu anki zamânın değerini getirir. 'Unix Timestamp' yapısı gereiği UTC+0 dilimine göre saniye değeri belirtse de, MySQL bu fonksiyonları çalıştırırken sunucu zamân diliminize dönüştürerek getirir.
- `DATE_FORMAT(tarih, format)` : Verilen târih değerini verdiğiniz formata çevirir ve döndürür. Girdi olarak hem `DATE`, hem de `DATETIME` ifâde eden metînleri verebilirsiniz.
- `TIME_FORMAT(zaman, format)` : Verilen zamân değerini, verdiğiniz formata çevirir ve döndürür. Girdi olarak `TIME` ifâde eden metînleri verebilirsiniz.
- `DATE(tarih)` : Verilen târih değerinin sadece târihle ilgili kısmını geri döndürür.
- `YEAR(tarih)` : Verilen târihin sadece sene değerini `INTEGER` olarak döndürür.
- `HOUR(tarih)` : Verilen târihin sadece saat değerini `INTEGER` olarak döndürür.
- `MINUTE()` : Verilen târihin sadece dakîka değerini `INTEGER` olarak döndürür.

- `SECOND()` : Verilen târihin sadece saniye değerini `INTEGER` olarak döndürür.
- Yukarıdaki son birkaç fonksiyonun, haftanın günü, senenin gün numarası gibi farklı versiyonları da vardır.

1) ROUND

Diğer Fonksiyonlar

1) COALESCE

- Verilen n tâne değer (bir sütun da verilebilir)den `NULL` olanları farklı bir biçimde getirmek istiyorsak kullanabiliriz:

```
CREATE TABLE data(id INT);
INSERT INTO data(NULL), (12), (NULL), (44);
SELECT COALESCE(id, 'SAYI YOK') FROM data AS veri;
# Geriye şu sonuç döner : SAYI YOK, 12, SAYI YOK, 44
```

2) GREATEST - LEAST

- `GREATEST()` aldığı parametreler içerisinden en büyük olanı döndürür; `LEAST()` aldığı parametreler içerisinden en küçük olanı döndürür.
- Bu fonksiyonlar karışık veri tipinde girdiler alabilir; ona göre de farklı değerlendirmeleri vardır, burada detaylı bilgi vermiyorum; fakat farklı veri tiplerinde sayılar içerisinden en büyüğünü almak için kullanabileceğinizi söyleyebilirim.

3) ISNULL

- Çok kullanılan bir fonksiyondur. İçerisine aldığı veri `NULL` ise `1` döndürür, aksi hâlde `0` döndürür.
- MsSQL'deki `ISNULL()` fonksiyonundan farklıdır.

HATÂ AYIKLAMA

- MySQL'de en son çalıştırılan sorgunun oluşturduğu hatâ ve uyarıları görmek için `SHOW WARNINGS;` komutu çalıştırılır.
- Hatâ sayısı şu şekilde öğrenilebilir:

```
SHOW COUNT(*) WARNINGS;
SELECT @@warning_count;
SELECT @@session.warning_count;
```

- Bellki miktarda hatâyı getirmek için `LIMIT` kullanılır : `SHOW WARNINGS LIMIT 3;`
- MySQL sistemi hatâların bir kısmını saklar. Sistemin en fazla kaç hatâ sakladığını öğrenmek için şu kodu çalıştırın:

```
SELECT @@max_error_count;
# Alttaki kod aynı sonucu döndürür:
SHOW VARIABLES WHERE variable_name = 'max_error_count';
```

Hatâyı Ele Almak

- Bir hatâyı, istisnayı ele almak için `DECLARE ... HANDLER` yapısı kullanılır:

```
DELIMITER $$
CREATE PROCEDURE proc()
BEGIN
    DECLARE EXIT HANDLER FOR SQL STATE 'hataKodu'
    BEGIN
       # Hatâ oluştuğunda çalışması istenen kodlar buraya yazılır
    END;
    # 'hataKodu' oluşturabilecek kodlar buraya yazılır
END$$
DELIMITER ;
```

- Bu yapıyı saklı yordamda işlemlerden evvel yazmalıyız.

SIGNAL Yapısı

- Bİr istisna fırlatmak için kullanılan yapıdır.
- Bu yapı saklı yordam, saklı fonksiyon, tetikleyici ve olay (event) içerisinde kullanılabilir.
- Misal:

```
# Bir saklı yordam içerisinde gelen verinin boş metîn olması
# durumunda kullanıcıya hatâ döndürmek isteyelim:
DELIMITER $$
CREATE PROCEDURE bilgiKontrol(IN info VARCHAR(800))
BEGIN
    IF info = ''
    THEN
        SIGNAL SQL STATE '50000'
            SET MESSAGE_TEXT = 'Lütfen bilgi kısmını doldurun';
    END IF;
    # işlemler..
    SELECT 'İşlem başarılı.' as mesaj;
END$$

# Düzgün çalışma:
bilgiKontrol('bilgiler...');
# mesaj
# İşlem başarılı

# Yanlış girdi:
bilgiKontrol('');# İşlem başarısız oldu
SHOW WARNINGS;# Hatâlara bak:
# Level       Code    Message
# Error       1644    Lütfen bilgi kısmını doldurunuz
```

OLAYLAR (EVENTS)

- Veritabanında olay (event), genellikle zamânlanmış görev (scheduled task) yerine kullanılır.
- MySQL, olayları idâre etmek için olay zamânlayıcısı yapısı barındırmaktadır.
- Bir olay, oluşturulma, etkinleştirilme, değiştirilme, pasifleştirilme ve silinme gibi farklı durumlarda bulunabilir.
- Olay zamânlayıcısı pasifse, zamânlanan olaylar çalışmaz. Olay zamânlayıcısının aktif olup, olmadığını öğrenmek için `SHOW PROCESSLIST;` kodunu çalıştırın. Genel sonuç içerisinde 'User' başlığı altında 'event_scheduler' değeri varsa olay zamânlayıcısı şu an çalışıyor demektir.
- Olay zamânlayıcısı şu şekilde açılıp, kapatılabilir:

```
SET GLOBAL event_scheduler = ON;# OFF yazılırsa kapatılır.
```

- `SHOW EVENTS;` kodu veritabanı sunucusundaki tüm zamânlanmış olayları gösterir.
- `SHOW EVENTS FROM veritabaniIsmi;` kodu ilgili veritabanındaki görevleri gösterir.

Yeni Bir Görev Oluşturma

- Şu söz dizimi tâkip edilir:

```
CREATE EVENT [IF NOT EXIST] gorevIsmi
ON SCHEDULE zamanlama
DO
    calistirilacakKod;
```

- Eğer olay gövdesinde birden fazla kod çalıştırılacaksa, `BEGIN ... END` bloğu eklemelisiniz.
- Birden fazla ifâdenin çalıştırılması kodlanmak istendiğinde, her ifâde sonuna koymanız gereken satır sonu karakterinin olay oluşturulurkenki satır sonu karakterinden farklı olması gerektiğini ve bunun için de olay oluşturulmadan evvel satır sonu karakterinin değiştirilmesi gerektiğini unutmayınız. Daha fazla bilgi için 'SAKLI YORDAM' bölümünün 'Satır Sonu İfâdesini Değiştirme ('DELIMITER' komutu)' başlığına bakınız:

```
DELIMITER $$
CREATE EVENT [IF NOT EXIST] gorevIsmi
ON SCHEDULE zamanlama
[ON COMPLETION [NOT PRESERVE]]
[ENABLE | DISABLE]
[COMMENT 'yorum']
DO
    BEGIN
        calistirilmasiIstenenKodlar;
    END$$
DELIMITER ;
```

- Söz diziminde 'zamanlama' yazan yere zamân belirten bir ifâde yazmamız gerekiyor.
- Bu ifâde tekrar eden olaylar ile tek seferlik olaylar için farklıdır.
- Eğer olayın bir kez çalıştırılmasını istiyorsak `AT TIMESTAMP [+ INTERVAL miktar zamanDilimi]` ifâdesi eklenmelidir.
- Olay tekrar eden bir olay ise 'zamanlama' yerine şu biçimde kod yazılmalıdır:

```
EVERY zamanDilimi
    STARTS zaman [+ INTERVAL miktar zamanDilimi]
    END zaman [+ INTERVAL miktar zamanDilimi]
```

- Tek seferlik olay, oluşturulduktan sonra belirtilen zamânda bir kez çalıştırılır ve varsayılan olarak sonra silinir.
- Eğer 'zamanlama' kısmına `AT CURRENT_TIMESTAMP` yazılırsa, olay oluşur oluşmaz çalıştırılır.
- 'zamanlama' kısmı için birkaç misal:

```
CURRENT_TIMESTAMP + INTERVAL 1 HOUR# Bir saat sonra
CURRENT_TIMESTAMP + INTERVAL + 1 WEEK# Bir hafta sonra
CURRENT_TIMESTAMP + INTERVAL + 1 DAY# Bir gün sonra
```

- Yukarıdakiki söz dizimlerinin daha iyi anlaşılması için bakınız:

```
# Bir veritabanı ve tablo oluşturalım:
CREATE DATABASE evDB;
USE evDB;# Komutların çalışacağı veritabanını seçiyoruz
# Tablo oluşturalım:
CREATE TABLE logs(
  id int
      AUTO_INCREMENT
      PRIMARY KEY,
  added_at DATETIME
      NOT NULL
      DEFAULT CURRENT_TIMESTAMP
);
# Ardından olay oluşturalım;
# Olay, tabloya yeni kayıt eklesin
# Olay her 30 saniyede bir çalışsın
# Şimdi başlasın ve 2 dakîka sonra bitsin:
CREATE EVENT ev_addLogEvery2Minute
ON SCHEDULE EVERY 30 SECOND# Her 30 saniyede bir
  STARTS CURRENT_TIMESTAMP# Şimdi başlasın
  ENDS CURRENT_TIMESTAMP + INTERVAL 2 MINUTE# 2 dk sonra bitsin
DO
  BEGIN
      INSERT INTO logs() VALUES();
  END$$
DELIMITER ;
# Yaklaşık 2 dk sonra görev silinir
# Tek seferlik görev eklemek istersek;
CREATE EVENT ev_addLogOnce
ON SCHEDULE
  AT CURRENT_TIMESTAMP + INTERVAL 15 SECOND
DO
  INSERT INTO users() VALUES();
# Tek satırlık görevler için, BEGIN...END bloğuna
# ve satır sonu ifâdesinin değiştirilmesine gerek yok
```

- Söz diziminde, `ON COMPLETION NOT PRESERVE` yazarsak olay tamâmlandıktan sonra silinmez. `NOT` kelîmesini kaldırırsak, olay tamâmlandıktan sonra silinir, varsayılan ayar da budur. Olayın silinmesi belirtilen son sürenin gelmesiyle veyâ tek seferlik bir olay için çalıştırılma zamânının geçmesiyle gerçekleşir; olay çalıştırıldıktan sonra silinecek demek değildir.
- Söz diziminde, `COMMENT 'yorum'` kısmını yazarak yorum belirtebiliriz.
- Söz diziminde, `DISABLE` kısmını yazarak olay oluşturulduğunda etkinleştirilmesini engelleyebiliriz.

Bir Görevin Değiştirilmesi

- Bir görev oluşturulduktan sonra değiştirilebilir.
- Söz dizimi:

```
ALTER EVENT [IF EXISTS] gorevIsmi
ON SCHEDULE zamanlama
[ON COMPLETION [NOT] PRESERVE]
[ENABLE | DISABLE]
[COMMENT 'buraya yorum yazılabilir']
DO
  gorevGovdesi
```

- Görüldüğü üzere, sadece `CREATE` ifâdesi yerine `ALTER` yazılarak görev değiştiriliyor; fakat görevin tüm parçalarını yeniden yazmak zorunda değilsiniz.
- Görevin belli başlı özelliklerini değiştirebilirsiniz:

```
# Tekrar periyodunu değiştirme:
ALTER EVENT ev_addLog
ON SCHEDULE EVERY 3 MINUTE;
# ev_addLog görevinin periyodunu 3 dakika yapar

# Görevi aktif / pasif yapma
ALTER EVENT ev_addLog ENABLE;
# Pasif yapmak için DISABLE yazılmalı

ALTER EVENT ev_addLog
    RENAME TO ev_addLogEvery3Min;
# Görev ismini ev_addLogEvery3Min yapar.
```

Bir Görevin Silinmesi

- Söz dizimi:

```
DROP EVENT IF EXISTS
  [veritabanıIsmi].gorevIsmi;
```

- Veritabanı isminin belirtilmesi zorunlu değildir.
- **ÖNEMLİ NOT: Görev silerken veyâ bir görevin bir şeyini değiştirirken veritabanı ismi belirtmek zorunda değilsiniz; veritabanı ismi belirtmezseniz diğer veritabanlarındaki**

aynı isimli görevler değişir diye endişe etmeyin, değişmiyor.

DİĞER

Yeni Eklenen Veride İsimlendirme

- Yeni eklenen satır isimlendirilebilir:

```
INSERT INTO users(name, surname, age, phoneNumber)
VALUES ('Ahmed', 'Fârukoğlu', 34, '23456789012')
AS newValue;# Satır takma ismi : newValue
```

- Yeni eklenen sütun da isimlendirilebilir:

```
INSERT INTO users(name, surname, age, phoneNumber)
VALUES ('Ahmed', 'Avar', 24, '23678945012')
AS new(isimYeni, soyisimYeni, yasYeni, telYeni);# Sütun takma isimleri
```