**SQL KOMUTLARI**

|  |  |
| --- | --- |
| **SELECT**  **FROM** | **SELECT** first\_name **FROM** employees; |
| **DISTINCT** | SELECT DISTINCT column\_name FROM table\_name  SELECT **DISTINCT** first\_name, gender FROM employees;  ***{Distinct=farklı. Unique=benzersiz. Belirtilen column(lar)daki distinct/unique* olan datayı çek getir. Birden fazla column belirtmişsek onların kesişimlerini alıyor}** |
| **WHERE** | SELECT column\_name(s) FROM table\_name WHERE condition(s);  SELECT \* FROM student\_table **WHERE** grade > 70 ***{if conditions gibi düşün}*** |
| **LIMIT** | SELECT column\_name(s) FROM table\_name LIMIT number\_rows;  SELECT \* FROM student\_table WHERE grade > 70 **LIMIT** 2;  ***{LIMIT’ten sonraki sayı kadar satır bilgisini seç getir}*** |
| **ORDER BY**  **ASC, DESC** | SELECT column\_name(s) FROM table\_name ORDER BY column\_name(s) ASC|DESC;  SELECT \* FROM employees **ORDER BY** first\_name **ASC**;  SELECT first\_name, last\_name, salary FROM employees **ORDER BY** salary **DESC**;  SELECT column\_name(s) FROM table\_name ORDER BY column1 ASC|DESC, column2 ASC|DESC, columnN ASC|DESC;  SELECT \* FROM employees **ORDER BY** ***gender* DESC, *first\_name*** **ASC**;  ***{Önce gender’i sıralıyor. Sonra gender’ler içinde first\_name’e göre sıralama yapıyor.}*** |
| **WHERE +**  **ORDER BY** | SELECT column\_name(s) FROM table\_name WHERE condition ORDER BY column\_name(s)s ASC|DESC;  SELECT \*  FROM employees  **WHERE** salary > 80000 **{Önce WHERE geliyor sonra ORDER BY!!}**  **ORDER BY** first\_name DESC; |

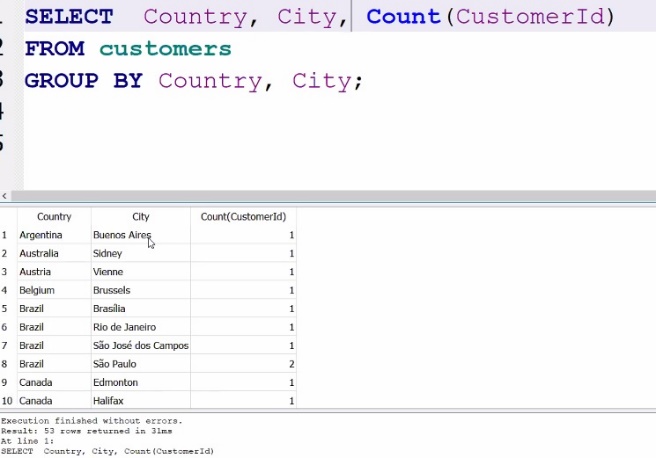
|  |  |
| --- | --- |
| **WHERE**  **AND** | WHERE left\_conditon AND right\_condition  SELECT \*  FROM employees  **WHERE** job\_title = 'Data Scientist' **AND** gender = 'Male'; |
| **WHERE**  **OR** | WHERE first\_condition OR second\_condition  SELECT \*  FROM employees  **WHERE** job\_title **=** 'Data Scientist' **OR** gender **=** 'Male'; |
| **WHERE NOT** | WHERE NOT first\_condition  **WHERE NOT** gender = 'Female'; |
| **WHERE**  **BETWEEN**  **AND** | WHERE test\_expression BETWEEN low\_expression AND high\_expression  Also we could write this query like this:  WHERE test\_expression >= low\_expression AND test\_expression <= high\_expression  SELECT \*  FROM employees  **WHERE** salary **BETWEEN** 80000 **AND** 90000;  Also we could write this query like this:  SELECT \*  FROM employees  **WHERE** salary **>=** 80000 **AND** salary **<=** 90000;  {Between’de sol ve sağdaki **iki değer de DAHİL** olarak kabul edilecek. **İstisnası**, eğer tarih verisi **date-time** şeklinde ise **sondaki tarih dahil olmuyor**. Buna bir gün eklememiz gerekiyor. Çünkü 23:59’dan sonra bir sonraki güne geçiyor ve date-time’da time olarak 00:00 yazıyor} |
| **WHERE**  **NOT BETWEEN**  **AND** | WHERE test\_expression NOT BETWEEN low\_expression AND high\_expression  SELECT \*  FROM employees  **WHERE** salary **NOT BETWEEN** 80000 **AND** 90000;  *Also we could write this query like this:*  SELECT \*  FROM employees  **WHERE** salary **<** 80000 **OR** salary **>** 90000; |
| **BETWEEN AND ORDER BY** | SELECT \* FROM employees  **WHERE** hire\_date **BETWEEN** '2018-06-01' **AND** '2019-03-31'  **ORDER BY** hire\_date; |

|  |  |
| --- | --- |
| **WHERE**  **NOT IN** | WHERE column\_name IN (value\_list)  SELECT \*  FROM employees  **WHERE** job\_title **IN** *('Data Scientist', 'Business Analyst', 'Project Manager', 'Web Developer')*;  SELECT \*  FROM employees  **WHERE** job\_title  **NOT IN** ('Operations Director', 'HR Manager', 'Sales Manager'); |
| **WHERE**  **LİKE**  **%**  **\_** | SELECT column\_name(s)  FROM table\_name  WHERE column\_1 LIKE pattern;  SELECT \*  FROM student\_info  **WHERE** county **LIKE** 'Wo**%**'; -- Wo ile başlayan ülkeler  SELECT \*  FROM student\_info  **WHERE** field **LIKE** '**%**Developer'; --sonu Developer ile bitenler  SELECT first\_name  FROM employees  WHERE first\_name LIKE 'El**\_**is';  ***{Linda’daki i ve n harleri için 2 alt tire kullandık}*** |
| **PRAGMA** | -- if you want to make **LIKE** operator **case-sensitive**, you need to use **PRAGMA** statement  PRAGMA case\_sensitive\_like = true;  **PRAGMA** *case\_sensitive\_like = true*;  SELECT \*  FROM student\_info  WHERE field LIKE '%developer';, |

|  |  |
| --- | --- |
| **COUNT( )** | SELECT COUNT(column\_name)  FROM table\_name;  SELECT **COUNT(first\_name)**  FROM student\_info;  ***{COUNT (\*) şeklinde kullanırsan tekrarları ve içeriği NULL olan satırları da getirir. Sütun ismiyle kullanırsan tekrarları getirir ama NULL ları getirmez. NULL ların sayılmasını istemiyorsak \* kullanmayıp parantez içine mutlaka sütun ismi girmeliyiz.}*** |
| **AS** | SELECT COUNT(first\_name) **AS** count\_of\_students  FROM student\_info;  ***{AS komutu ile listelenen bilgiye takma ad veriyoruz böylece ekrana getirilen sonucun ne olduğunu daha iyi anlayabiliyoruz.}*** |
| **COUNT( DISTINCT)** | SELECT **COUNT(DISTINCT field)** AS count\_of\_field  FROM student\_info;  **{Birbirinden farklı (unique) kaç bilgi varsa onun sayısını verir.}** |
| **COUNT**  **WHERE** | SELECT COUNT(\*) AS count\_of\_students  FROM student\_info  WHERE state = 'Virginia'; |
| **MIN** | SELECT MIN(column\_name)  FROM table\_name; ***{ignores the NULL values}***  SELECT **MIN(salary)** AS lowest\_salary  FROM employees;  SELECT **MIN(salary) AS** lowest\_salary  FROM employees  **WHERE** gender = 'Female';  SELECT **MIN(hire\_date)** AS earliest\_date  FROM employees; |
| **MAX** | SELECT MAX(column\_name)  FROM table\_name; ***{ignores the NULL values}***  SELECT **MAX(salary)** AS highest\_salary  FROM employees;  SELECT **MAX(salary)** AS highest\_salary  FROM employees  **WHERE** gender = 'Male';  SELECT **MAX(hire\_date)** AS newest\_date  FROM employees; |

|  |  |
| --- | --- |
| **SUM** | SELECT SUM(column\_name)  FROM table\_name; {***returns the sum of a numeric column}***  ***Sadece numeric datalara uygulanıyor!!***  SELECT **SUM(salary)** AS total\_salary  FROM employees; |
| **AVG** | SELECT AVG(column\_name)  FROM table\_name; ***{calculates the average of numeric colum}***  ***Sadece numeric datalara uygulanıyor!!***  SELECT **AVG(salary)** AS average\_salary  FROM employees;  SELECT name, AVG(Milliseconds)  FROM tracks  WHERE Milliseconds > 393599.212103911;  SELECT name  FROM tracks  WHERE Milliseconds > (SELECT AVG(Milliseconds)  FROM tracks); |
| **GROUP BY** | SELECT column\_1, **aggregate\_function**(column\_2)  FROM table\_name {**Her zaman WHERE’den sonra gelir**  GROUP BY column\_1; **Her zaman ORDER BY’dan önce gelir.}**    **{veri grubu başına yalınızca bir sonuç döndürür. Bunun için verileri unique olarak alır yani kendi içinde DISTINCT yapar. Mesela aşağıdaki örneğe göre Male ve Female’ler için iki ayrı sonuç döndürür.}**    !!! ***SELECT’teki nonaggregate ifade (column\_1), GROUP BY cümlesinde olmalı!!! Yani neyi GROUP BY’ın yanına yazıp grupluyor isek aynen onu SELECT’in yanına yapıştır.***    SELECT *gender*, **COUNT**(gender)  FROM employees  **GROUP BY** *gender*; ***{gender’ların sayılarını gender lara göre (yani gender sütunundaki verileri unique/DISTINCT yaparak) grupla. Yani burada gender Male ve Female’lerden oluştuğu için, Male ve Female başlıkları altında iki grup olarak sayıları ver.}*** |

|  |  |
| --- | --- |
| **GROUP BY**  **ÖRNEKLERİ** | SELECT gender, **COUNT**(job\_title)  FROM employees  **WHERE** job\_title = 'Data Scientist'  **GROUP BY** gender; **{job\_title’lar arasında Data Scientist’leri çek, onları gender lara göre gruplandır.}**  SELECT job\_title, count(job\_title)  FROM employees  GROUP BY gender, job\_title; {gender’lara göre hangi iş pozisyonlarından kaçar tane var}  SELECT job\_title, gender, count(gender)  FROM employees  GROUP BY job\_title, gender ; { iş pozisyonlarına göre gender’lardan kaçar tane var}  SELECT job\_title, gender, count(gender)  FROM employees  WHERE job\_title='Data Scientist' {Data Scientist olarak gender’lardan kaçar tane GROUP BY job\_title, gender ; var}  SELECT Composer, COUNT(TrackId)  FROM tracks  GROUP BY Composer; {Her bestecinin kaçar şarkısı var?  Burada null olanları da alır.  SELECT Composer, COUNT(TrackId)  FROM tracks  WHERE Composer IS NOT NULL {Her bestecinin kaçar şarkısı var?  GROUP BY Composer; burada null olanları çıkarttık  SELECT Country, COUNT(CustomerId)  FROM customers  GROUP BY Country; {her ülkeden kaçar tane müşteri var?}  SELECT AlbumId, MIN(Milliseconds)  FROM tracks  GROUP BY AlbumId; {her albümdeki süresi en kısa olan şarkıyı listele} |



|  |  |  |  |
| --- | --- | --- | --- |
|  | **GROUP BY, Aggregate function’ı çağırmadan önce sonuçları gruplandırır. Bu, tüm sorgu yerine gruplara aggregate func. uygulamanıza olanak tanır.**  **WHERE clause operates on the data before the aggregation.**  **WHERE clause happens before the GROUP BY clause.**  **Only the rows that meet the conditions in the WHERE clause are grouped.**  **ORDER BY, follows GROUP BY.**  ***Yani ilk önce WHERE şartı uygulanıp filtrelenir sonra üzerine GROUP BY, sonra aggregate function uygulanır son olarak da sonuç ORDER BY a göre sıralanır.***  ***(çalışma sütunlarını + koşula göre seç + fonksiyona göre gruplandır + sıralamasını belirle)*** | | |
| **MIN**  **GROUP BY** | SELECT gender, **MIN**(salary) AS min\_salary  FROM employees  **GROUP BY** gender; | | |
| **MAX**  **GROUP BY** | | SELECT gender, **MAX**(salary) AS max\_salary  FROM employees  **GROUP BY** gender; |
| **GROUP BY**  **ORDER BY** | | SELECT gender,  MAX(salary) AS max\_salary  FROM employees  **GROUP BY** gender  **ORDER BY** max\_salary DESC ;  {salary içerisindeki maksimum ücreti genderlarla grupla yani F’lerden en yüksek alan ile M’lerden en yüksek alanları bir araya getir. bunları yüksekten alçağa doğru sırala}  ***{ yukarda MAX(salary)’e max\_salary olarak takma ad vermiştik. Ve bu adı ORDER BY max\_salary ifadesinde kullandık. Bunun yerine ORDER BY MAX(salary) olarak da kullanabiliyoruz.}*** |
| **SUM / AVG**  **GROUP BY** | | SELECT gender, **SUM(salary)** **AS** total\_salary  FROM employees  **GROUP BY** gender;  SELECT gender, **AVG(salary)** **AS** average\_salary  FROM employees  GROUP BY gender; |

|  |  |  |
| --- | --- | --- |
| **JOIN** | ***combine records (rows) from two or more tables in a SQL database based on a related column between the two****.*   * **INNER JOIN:** Returns the common records in both tables. * **LEFT JOIN:** Returns all records from the left table and matching records from the right table. * **RIGHT JOIN:** Returns all records from the right table and matching records from the left table. * **FULL OUTER JOIN:** Returns all records of both left and right tables. * **CROSS JOIN:** Returns the Cartesian product of records in joined tables. * **SELF JOIN:** A join of a table to itself | |
| **INNER JOIN** | SELECT columns ***{her iki tablodaki istenen sütunlar}***  FROM table\_A  INNER JOIN table\_B ON join\_conditions  INNER JOIN table C ON join\_conditions **{üçüncü ve başka tablolar da birleştirilebilir}**  SELECT columns  FROM table\_A  **INNER JOIN** table\_B  **ON** join\_conditions1 **AND** join\_conditions2  **INNER JOIN** table\_C  **ON** join\_conditions3 **OR** join\_conditions4  ...  ***{INNER JOIN is the most common type}*** | |
| **INNER JOIN ÖRNEĞİ** | SELECT  employees.emp\_id,  employees.first\_name,  employees.last\_name,  departments.dept\_name,  departments.dept\_id  FROM employees  **INNER JOIN** departments  **ON** employees.emp\_id = departments.emp\_id;  ***{employees tablosu ile department tablosundaki SELECT te belirtilen 5 sütunu birleştiriyoruz. Bunları ortak sütun olan emp\_id ile birbirine ilişkilendiriyoruz. (bu sütunun SELECT’in içinde gösterilmesi şart değil ama ilişkilendirmede kullanıyoruz.) İlişkilendirdiğimiz sütun isimlerini INNER JOIN ON cümlesi içinde birbirine eşitliyoruz. Burdaki condition eşitlik olmaya da bilirdi.***  ***Sütun isimlerini belirlerken kullandığımız syntax a dikkat!}***  ***{çıktı olarak; emp\_id’lerin kesişim kümesi alınıyor. eşitlenen iki sütunda (emp\_id’ler) da aynı anda bulunmayan id’ler birleştirilmiyor. Employees.emp\_id 12 satır ise çıktıda 9 satır bulunabilir.}*** |

|  |  |
| --- | --- |
| **INNER JOIN ÖRNEĞİ** | {write a query to return the departments of female employees. Show just first name, last name, salary and department of the employees}  **SELECT** employees.first\_name,  employees.last\_name,  employees.salary,  departments.dept\_name  **FROM** employees  **INNER JOIN** departments  **ON** employees.emp\_id=departments.emp\_id  **WHERE** employees.gender='Female'; |

|  |  |
| --- | --- |
| **LEFT JOIN** | ***{soldaki tablonun tamamı ile sağdaki tabloyla ortak kesişimlerini katıyor}***  SELECT columns  FROM table\_A  LEFT JOIN table\_B ON join\_conditions  ***{If no match is found for a particular row, NULL is returned.}***  SELECT  employees.emp\_id,  employees.first\_name,  employees.last\_name,  departments.dept\_name,  departments.dept\_id  FROM employees  **LEFT JOIN** departments  **ON** employees.emp\_id = departments.emp\_id;  ***{Her iki listeden de emp\_id’leri uysa da uymasa da birleştirdi. Uyuşmayan id’lere değer olarak NULL verdi.}*** |

|  |  |
| --- | --- |
| **LEFT JOIN ÖRNEK** | SELECT tracks.Name, genres.name  FROM tracks  **INNER JOIN** genres **ON** genres.GenreId = tracks.GenreId  **{Her müşterinin faturalarını getir}**  SELECT i.InvoiceId, c.FirstName, c.LastName FROM customers c JOIN invoices i ON c.CustomerId = i.CustomerId  **{Her artistin albüm isimlerini getir}**  SELECT ar.ArtistId, ar.name, al.AlbumId, al.Title  FROM artists ar  LEFT JOIN albums al  ON ar.ArtistId = al.ArtistId  **{Sanatçıların albüm bilgilerini sorarsa LEFT tablo yani kapsayıcı tablo sanatçılar olacak.. albümlerin artist isimlerini sorsaydı LEFT tablo albümler olacaktı!!!}** |
| **LEFT JOIN- GROUP BY**  **ÖRNEK** | lists the number of orders sent by each shipper:    **SELECT** Shippers.ShipperName,**COUNT**(Orders.OrderID) **AS** NumberOfOrders  **FROM** Orders  **LEFT JOIN** Shippers **ON** Orders.ShipperID = Shippers.ShipperID  **GROUP BY** ShipperName;  **{Shippers ile Orders tablolarını eşit olan ShipperID sütunları üzerinden birleştirdik. ShipperName sütununun elemanlarına göre gruplayarak karşılarına denk gelen Orders tablosundaki OrderID sayılarını yazdırdık.}** |

|  |  |
| --- | --- |
| **SUBQUERY** | **SELECT, WHERE ve FROM clause larının içerisinde kullanılabiliyor.**  **Farklı tablolardan değer çekiyorsak mutlaka tabloların ortak sütunu olmak zorunda.. onları eşitleyip onlar üzerinden WHERE clause’a veri çekiyoruz.**  SELECT column\_name {this is  FROM table\_1, table\_2 outer  WHERE column\_name OPERATOR ( query}  SELECT column\_name {this is  FROM table\_1, table\_2); inner query}  {**The inner query is executed first before the outer query. The results of the inner query are passed to the outer query.}**   * **A subquery must be enclosed in parentheses.** * **An ORDER BY clause is not allowed to use in a subquery.** * **The BETWEEN operator can't be used with a subquery. But you can use BETWEEN within the subquery** |
| **SUBQUERY**  **SINGLE ROW** | **Single-row subqueries return one row with only one column and are used with single-row operators such as =, >, >=, <=, <>, != .**  **{find the employees who get paid more than Rodney Weaver. Our query should return first name, last name, and salary info of the employees.}**  **SELECT** first\_name, last\_name, salary  **FROM** employees  **WHERE** salary >  (**SELECT** salary **{iç query: employees tablosundan ismi Rodney olan çalışanın** **FROM** employees **ücretini seç.. dış querydeki koşula uygula. (WHERE Salary>….)**  **WHERE** first\_name = "Rodney"); **ve buradan yalnızca 1 değer geldiğine dikkat et}**  SELECT TrackId, name, AlbumId  FROM tracks  WHERE AlbumId= **(**  **SELECT AlbumId**  **FROM albums**  **WHERE Title='Faceless'**  **)**  --LEFT JOIN ile de benzer işlemi yapabilirsin.  --farkı şu; subquery'de bir tek sonuç çıkartıyor bu yüzden  --outer query de aynı sütun ismi varsa ancak o sütuna data sağlıyor.  --bu örnekteki AlbumId’lerin aynı olduğu gibi..  --JOIN de ise başka bir sütunu ilave edebiliyoruz |

|  |  |  |
| --- | --- | --- |
| **SUBQUERY**  **SINGLE ROW**  **with**  **AGGREGATE** | | **{ find out the employees who get paid more than the average salary. Our query should return first name, last name, and salary info of the employees. }**  SELECT first\_name, last\_name, salary  FROM employees  WHERE salary >  (SELECT **AVG**(salary) **{Buradan yalnızca 1 satır değeri geldiğine dikkat et}**  FROM employees); |
| **SUBQUERY**  **MULTIPLE ROW** | **Multiple-row subqueries return sets of rows and are used with multiple-row operators such as IN, NOT IN, ANY, ALL**  **{find the employees (first name, last name from employees table) who work under the Operations department (departments table). Our query should return first name and last name info.}**  **SELECT** first\_name, last\_name  **FROM** employees  **WHERE** emp\_id **IN**  (**SELECT** emp\_id **{ismi Operations olan departmanların emp\_id’lerini seç}**  **FROM** departments **{burada satır kümesi bilgisi geldiğine dikkat et! }**  **WHERE** dept\_name = **'Operations'**); | |
| **DDL COMMANDS** | * DDL - Data Definition Language   **CREATE, ALTER, DROP**   * DML - Data Manipulation Language   **INSERT, UPDATE, DELETE, SELECT**   * D DCL - Data Control Language   **REVOKE, GRANT {iptal etme, verme(hibe etme)}** | |
| **typeof function** | {typeof in SQLite to display the data types of the values stored in the columns. }  SELECT **typeof**(emp\_id),  **typeof**(first\_name),  **typeof**(hire\_date)  FROM employees  LIMIT 1  *OUTPUT*  typeof(emp\_id) typeof(first\_name) typeof(hire\_date)  -------------- ------------------ -----------------  integer text text | |

|  |  |
| --- | --- |
| **DATA TYPES** | **In SQLite, there are five primitive data types.**   * **NULL:**The value is a NULL value. NULL values mean missing information or unknown. * **INTEGER:**The value is a whole number (positive or negative), stored in 1, 2, 3, 4, 6 or 8 bytes depending on the magnitude of the value. * **REAL:** The value is a floating-point value, stored as an 8-byte float. They are real numbers with decimal values. * **TEXT:** The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE). * **BLOB:** BLOB stands for a binary large object that can store any kind of data. The value is a blob of data, stored exactly as it was input |
| **CREATE TABLE** | CREATE TABLE database\_name.table\_name (  column1 data\_type,  column2 data\_type,  ...  );  **CREATE TABLE** Ogrenci(      OgrenciId       **int**,      OgrenciNumarasi **int**,      OgrenciAdi      **varchar**(255),      OgrenciSoyadi   **varchar**(255),      OgrenciMatNotu  **int** ,  **PRIMARY KEY**(OgrenciId)  );  **{Yukarıda bir Ogrenci tablosu oluşturmuş olduk. Toplam 5 alan ekledik. OgrenciId alanını integer olarak belirledik ve OgrenciId Primary key yaptık yani OgrenciId benzersiz olmuş oldu. OgrenciId si aynı olan birden fazla kayıt olmaması için bunu yaptık.}** |
| **.table** | **{Komut satırında .table yazarsan o database’e ait tablolar listelenir.}** |

|  |  |
| --- | --- |
| **CONSTRAINS** | Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.  CREATE TABLE table\_name (     column1 datatype *constraint*,     column2 datatype *constraint*,     column3 datatype *constraint*,     .... );   * [NOT NULL](https://www.w3schools.com/sql/sql_notnull.asp) - Ensures that a column cannot have a NULL value * [UNIQUE](https://www.w3schools.com/sql/sql_unique.asp) - Ensures that all values in a column are different * [PRIMARY KEY](https://www.w3schools.com/sql/sql_primarykey.asp) - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table * [FOREIGN KEY](https://www.w3schools.com/sql/sql_foreignkey.asp) - Prevents actions that would destroy links between tables * [CHECK](https://www.w3schools.com/sql/sql_check.asp) - Ensures that the values in a column satisfies a specific condition * [DEFAULT](https://www.w3schools.com/sql/sql_default.asp) - Sets a default value for a column if no value is specified * [CREATE INDEX](https://www.w3schools.com/sql/sql_create_index.asp) - Used to create and retrieve data from the database very quickly   CREATE TABLE table\_name (  column1 DATATYPE CONSTRAINT,  column1 DATATYPE CONSTRAINT,  ...  ); |
|  | The DEFAULT constraint is used to set a default value for a column.  The default value will be added to all new records, if no other value is specified.  CREATE TABLE Persons (     ID int NOT NULL,     LastName varchar(255) NOT NULL,     FirstName varchar(255),     Age int,     City varchar(255) DEFAULT 'Sandnes' );  To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:  ALTER TABLE Persons ALTER City SET DEFAULT 'Sandnes';  To drop a DEFAULT constraint, use the following SQL:  ALTER TABLE Persons ALTER City DROP DEFAULT; |

|  |  |
| --- | --- |
| CONSTRAINS  **PRIMARY KEY** | **CREATE TABLE** table\_name(  column\_1 INT **PRIMARY KEY**,  column\_2 TEXT,  ...  );  CREATE TABLE employees(  emp\_id INT NOT NULL **PRIMARY KEY**,  first\_name TEXT,  last\_name TEXT,  salary INT,  job\_title TEXT,  gender TEXT,  hire\_date TEXT);  PRIMARY KEY (emp\_id) **{bu şekilde de kullanılabilir}** |
| CONSTRAINS  **FOREIN KEY** | 1. **TABLO**   CREATE TABLE customers (customer\_id INT PRIMARY KEY,  first\_name TEXT,  second\_name TEXT);   1. **TABLO**   CREATE TABLE orders (  order\_id INT PRIMARY KEY,  order\_number INT,  customer\_id INT,  **FOREIGN KEY** (customer\_id)  **REFERENCES** customers (customer\_id)  );  **In the orders table, we defined the FOREIGN KEY for the customer\_id column and pointed it using REFERENCES to the customer\_id column in the *customers* table.** |
| **INSERT INTO** | **We use INSERT INTO statement to add data.**  INSERT INTO table\_name (column1, column2)  VALUES (value1, value2);  **INSERT INTO** customers (customer\_id, first\_name, last\_name)  **VALUES** (1, 'Robert', 'Cursor'),  (2, 'Julia', 'Loyds'),  (3, 'Jack', 'Morgten');  **INSERT INTO** orders (order\_id, order\_number, customer\_id)  VALUES (1, 101, 3),  (2, 102, 3),  (3, 103, 2),  (4, 104, 1);  **INSERT INTO** orders (order\_id, order\_number, customer\_id)  VALUES (5, 105, 4); |
| CONSTRAINS  **NOT NULL** | By default, a column can hold NULL values.  The NOT NULL constraint enforces a column to NOT accept NULL values.  This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.  CREATE TABLE Persons (     ID int NOT NULL,     LastName varchar(255) NOT NULL,     FirstName varchar(255) NOT NULL,     Age int );  To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:  ALTER TABLE Persons MODIFY Age int NOT NULL;  **CREATE TABLE** orders (  order\_id INT PRIMARY KEY,  order\_number INT **NOT NULL**,  customer\_id INT,  FOREIGN KEY (customer\_id)  REFERENCES customers (customer\_id)  );  **Üstte NOT NULL ile oluşturduğumuz order\_numer’a NULL değeri girmeye çalışırsak;**  INSERT INTO orders (order\_number)  VALUES (NULL);  **If you run the query, you will get a "NOT NULL constraint failed" error.** |
| **ALTER TABLE**  **+**  **RENAME TO**  **ADD COLUMN** | **ALTER TABLE** existing\_table  **RENAME TO** new\_table;  **ALTER TABLE** table  **ADD COLUMN** |
| **DROP TABLE** | We use DROP TABLE statement to remove a table in a database  **DROP TABLE** table\_name;  **DROP TABLE** orders  .table; **{database’e ait tabloları görmek için}** |
| **DELETE** | **DELETE** **FROM** [tablo adı] **WHERE** [şartlar]  Burada önemli bir noktaya temas etmek istiyorum. DELETE deyimiyle yapılan işlemler sonucunda kayıt veya kayıtlar tamamen silinir. Örneğin; yukarıdaki tablodan ID’si 61 olan kaydı silmek istersek kayıt tamamen silinir, yani tabloda artık 3 kayıt kalır. Bir kayıttaki belli alanları silmek (temizlemek) istersek UPDATE deyimini kullanırız.  Bu formatta ilk dikkatimizi çeken şey DELETE deyiminden sonra hiçbir alan adı belirtmiyoruz. Basit bir örnekle başlayalım:   |  |  | | --- | --- | | 1 | DELETE FROM personel |   Bu SQL komutu herhangi bir şart aramaksızın (WHERE deyimini kullanmadık), personel tablosunda bulunan kayıtların hepsini siler. Fakat burada dikkat edilmesi gereken husus, bu işlemden sonra personel tablosu yapısını aynen korur. Yani içindeki kayıtlar silinir ama tablo yapısı, alan isimleri, veri türleri, birincil anahtar, vs. hepsi korunur. Eğer tabloyu içindeki kayıtlarla beraber tamamen silmek istersek DELETE yerine DROP TABLE komutunu kullanırız.  Başka bir örnekle devam edelim. Bu örnekte bölümü Muhasebe olan kayıtları silmek istiyoruz. Dolayısıyla komutumuz şu şekilde olacak:   |  |  | | --- | --- | | 1 | DELETE FROM personel WHERE bolum='Muhasebe' | |
| **ALTER**  **ADD** | ALTER TABLE Table\_Name ADD Column\_Name data\_type Column\_Constraint; |
|  |  |

Aggregate fonksiyonları yalnızca bir veri üretir. SUM ve AVG, yalnızca INTEGER değerlere uygulanır.

NULL bilinmeyen değer demek. Sıfır demek değil. NULL lar birbirine eşit değildir. Her NULL un farklı bir değeri vardır.

COUNT \*, NULL dahil tüm satırları sayar. COUNT (colum\_name) şeklinde sütun ismi girilirse NULL’lar hariç satıları sayar.

COUNT \* hariç diğer hepsi (min, max vs.) Null ları göz ardı eder. Zaten MIN, MAX vs. de \* ile kullanma şekli yok.

Birden fazla satıra işlem yapan AVG, SUM, COUNT vs. de önüne bir sütun ismi yazmıyoruz çünkü yazarsak o sütunun bir satırına karşılık değer geleceği için tek satırı alır. Örnek, COUNT tek satır sonuç döndürdüğü için önüne yazacağın sütun için tek satırı alır. Aggragate fonsiyon kullanırken buna dikkat et. Ama MIN, MAX’ın önüne sütun ismi girebilirsin.

WHERE’i Aggregate fonksiyonundan önce çalıştırır ki önce sınırlayıp, sınırlanmış veriye fonksiyon işlemi uygulansın. Böylece sistemi yormamış oluyor.

GROUP BY da WHERE’den sonra çalıştırılıyor. Aynı nedenden dolayı.

FOREIGN key hiçbir zaman PRIMARY KEY’den başka bir değer alamaz. Tam bir uyum sözkonusudur. Tabloları birebir uydurur.

Sanatçıların albüm bilgilerini sorarsa LEFT tablo sanatçılar olacak (kapsayıcı olan o)

Albümlerin artist isimlerini sorsaydı LEFT tablo albüms olacaktı.

Aggragate fonsiyonlar için GROUP BY kullanılıyorsa filtrelemeyi HAVING ile yapıyoruz. (WHERE ile yapmıyoruz) HAVING, Aggregate fonksiyonların filtrelenmesinde kullanılır.

Yani GROUP BY ile kullandığımız herhangi bir AGGREGATE fonksiyonunun filtrelenmesini HAVING ile yapıyoruz. GROUP BY olmadan HAVING olmaz. Mutlaka GROUP BY olacak.

Tablo içindeki sütunlarda filtreleme yaparken WHERE kullanıyoruz, Aggregate fonksiyonlar için HAVING…