## week16d05

## mai témák:

## Constraintek

## Full outer join

## megjelenítjük a leftből és a rightból is a hiányzót

## MariaDB nem ismeri a full outert; helyette: UNION (ALL – az all duplikat nem szűri ki)!

**SELECT** \* **FROM** `pizza` **WHERE** `pizza\_id` < 2000

**UNION**

**SELECT** \* **FROM** `pizza` **WHERE** `pizza\_name` **LIKE** '%m%';

## ugyanolyan típusú és ugyanolyan sorrendű oszlopok kellenek!

**SELECT** \*

**FROM**

`order` **AS** o **LEFT** **JOIN** `pizza` **AS** p

**ON** o.pizza\_id = p.pizza\_id

**UNION**

**SELECT** \*

**FROM**

`order` **AS** o **RIGHT** **JOIN** `pizza` **AS** p

**ON** o.pizza\_id = p.pizza\_id;

## ha a MAX- MINhez akarok még plusz oszlop adatot, arra order by és limit ( vagy subselect? – nem ment a tanárnak se)

## Constraints

## eddigi tábláinkban betartottuk az iratlan szabályokat (id egyik a másikban is ugyanaz legyen, hogy tudjunk joinolni stb.); de ha ezeket elrontanánk, a joinjaink széthullanának

## constraint(kényszer) – ezeket kényszeríti ki

## minden, amit az oszlopokhoz írunk, a típusdefinicíó után, az kényszrer (pl NOT NULL? UNIQUE, AUTO\_INCREMENT)

**CREATE** **DATABASE** `logistics`;

**USE** `logistics`;

**CREATE** **TABLE** `suppliers` (

`supplier id` **INT**,

`name` **VARCHAR**(255) **NOT** **NULL**,

`phone` **VARCHAR**(15) **NOT** **NULL** **UNIQUE**,

`address` **VARCHAR**(255) **NOT** **NULL**

);

**INSERT** **INTO** `suppliers` **VALUES** (1, 'Tamas', 1234, 'valahol');

**INSERT** **INTO** `suppliers` **VALUES** (1, 'Tamas', 1234, 'valahol');

## > duplán nem ment a unique miatt

## *Constraintek:*

## not null

## unique

## primary key

## foreign key

## check

## default

## index – keresést gyorsít! SELECT-ben sok sor összehasonlítás

## auto increment, alter table auto\_increment

## *Entity relationship diagram (er),* de ez már nem túl modern; mai jobb ábrkákon szimbólumokkal mutatják a constrainteket is

## DESCRIBE `tábla` - hasznos parancs, kilistázza a tábla adatait

## felülethez ne szokjunk hozzá, hanem konzol és kódolás!

## konzol mindenhol lesz

## `` - oszlopnév, táblanév, adatbázisnévnél

## ’ ’ – értékeknél

## ”” - soha

## PRIMARY KEY

## egyedileg azonosít minden sort ezen oszlopadat alapján

## …

`supplier id` **INT** **PRIMARY** **KEY**,

## ..

## autoincrement fontosságú túlértékelt, régi cucc

## nagy projekten térben és időben szét van pakolva, kifejezetten nem jó, nem tartalmaz elég infót

## ma már nem egy számot pörgetnek, hanem egy globális azonosítót (pl. abc45465g-34354ffgh665, alfanumerikus és teljesen egyedi)

## > UUID unique identifier, egyedi szekvencia generátorokkal csinálják

## kisvállalkozások esetében oké az auto\_increment

## programozói keretrendszerekben még kevésbé lesz fontos, helyettesíteni fogja a rendszer a saját algoritmusával és szoftverből fog jönni az egyedi azonosító

## (cloud native java)

## törlésnél megszakad a számozás; visszaállítás ALTER TABLE-el

## melyik serverről jön a rekord? egyik auto incrementje pl. 10000től, másik serveren pedig 90000től induljon

## PRIMAREY KEY része lehet több oszlop is! = összetett kulcs

## PRIMARY KEY (`supplier\_id`, `phone`); attól ez még egy kulcs!

## idegen kulcsokra is működik

## FOREIGN KEY

## táblák között kell kialakítani olyan vizsgálható relációt, amit a beszúráskor már lehet vizsgálni, hogy valid-e, beszúrható-e

## összetett idegen kulcsokra nincs nagyon szükség

## CONSTRAINEK fontosak Java szempontból!

## REFERENCES… meg lehet adni különböző utasításokat:

## pl. egy kliens törlésnél már rajta ülnek taskok és noteok a kényszerek miatt

## ON DELETE CASCADE: töri a hozzátartozókat is (rekordhoz tartozó összes másik rekordot az idegenek táblákból a foreign kulcs mentén)

## ON UPTADE CASCADE – végigviszi a változtatást az összes kacsolódó táblában is

## SET DEFAULT – megmarad az adat, ráállítják egy default azonosítóra

## nem törlünk, hanem inkább archiválunk vagy set default egy törlésre előkészített azonosítóra

## ON DELETE SET NUL – ami foreign keyből jön, arra törléskor(pl tábla drop) NULL-t állít be

## mai adatbázisok képesek titkosítani pl. jelszavakat

## Kapcsoló tábla 2.19 senior konzi!!

## Eddig volt mondjuk Products és Suppliers tábla és köztük egy egy-a többhöz kapcsolat (egy s -> több p)

## Noteokhoz Categóriák - > egy notenak lehet mondjuk 2 katja, egy katnak meg sok noteja

## *ORM* – object relationshop mapping

## a több többhöz kapcsolatot kiváltjuk egy kapcsolótáblával

## ncd id / nc id / c id

## 

## minden táblázat az egy lista javaban

## pl. egy rendeléshez több pizza, kapcsoló tábla kell (rendelés plusz pizza felsorolás)

## .txt file ;el elválasztva vagy adatbázis? mindig kérdés; a használt keretrendszer (JDBC) fogja eldönteni; a keretrendszer fogja azt csinálni, amit mi most csináltunk a héten?

## igyekezzünk kivenni amit tudunk Javaból

## java classomat tudom-e még optimalizálni, az lesz inkább a kihívás

## 

## Constrainekkel:

## 

## nem ment, f1 nevet lecserélte fk\_notetable-re és a másikat is fk\_remindertable-re; két constraintnek nem lehet ugyanaz a neve amúgy

## week16d04

## select, beszúrás, törlés és változtatás – ezeket kell tudnunk

## COUNT – számolja meg, hogy mennyi van belőle

**SELECT** **COUNT**(\*) vagy valamelyik egyedi azonosító; aggregátor; ilyen még az AVG meg a SUM

MIN és MAX függvények -> szintén egy értékkel térnek vissza

## ez nem ok:

**WHERE** 50000 < `terulet` < 150000;

## így ok:

**SELECT** **COUNT**(\*) **FROM** `orszagok` **WHERE** `terulet` > 50000

**AND** `terulet` < 150000;

## vagy BETWEEN-el

## Beágyazott lekérdezések

## egy query közepén egy beágyazott query

## általánosságban a WHERE feltételben használt subquery

## a reláció jobb oldalán lévő kifejezést zárójelbe!

## Ami az értéket adja nekem az összehasonlításhoz, azt a lekérdezést ágyazom be

**SELECT** `orszag` **FROM** `orszagok`

**WHERE** `terulet` < (**SELECT** `terulet` **FROM** `orszagok` **WHERE** `orszag` = 'MAGYARORSZÁG');

**SELECT** `orszag`, `terulet` **FROM** `orszagok`

**WHERE** `terulet` = (**SELECT** **MAX**(`terulet`) **FROM** `orszagok`);

**SELECT** `orszag`, `terulet`

**FROM** `orszagok`

**WHERE** `terulet` = (**SELECT** **MIN**(`terulet`) **FROM** `orszagok`);

## A másik lehetőség beágyazásra, hogy a FROM után nem egy táblát írok, hanem egy olyan SELECT-et, ami több soros eredményt ad:

## 

## Egynél több táblából adat leolvasás

## követelményeknek(feladat) megfelelően mely tábláink tartalmazzák az adatokat?

## A -🡪 C 🡨-B

## aid🡨------🡪bid

## össze kell párosítanom a sorokat

## JOIN!

## egyik táblához hozzáválogatja a másik tábla sorait az adott illesztés oszlopadatai mentén

## *fajtái:* INNER JOIN, LEFT JOIN, RIGHT JOIN (és FULL OUTER JOIN)

## JOIN kifejezés felépítése, referenciák használatával

## pizza -🡪 C 🡨-order

## pizza\_id< ------ > pizza\_id

## 1001🡨------🡪 1001

## SELECT \* FROM

## `pizza` AS p LEFT JOIN `order` as o ON p.pizza\_id = o.pizza\_id;

## LEft – összes első táblás sor benne lesz, és hozzá ha van, a második táblás adatok még

## INNER JOIN a RIGHT és a LEFT metszete; nem lesz hiányzó adatos sor

## Importálás CSVből!

## 

## Eszközök – beállítások - nyelv

## táblán az UTF8 beállítása – jobb klikk az adatbázison, Edit és Collation(Illesztés; sorrendek, relációk stb a nem ASCI kódú karakterekre)

## week16d03

## csak első x rekord - ORDER BY.. DESC LIMIT 10!

## adatbázis serveren; le kell tudnunk kérdezni és adatot kigyűjteni

## külön lekérdezési lapok

## selecteket kell tudjunk majd írni! SELECT…FROM…WHERE

## aktuális kijelölés futtatása

## rendezzünk sorba, mielőtt csoportosítunk, akkor egyszerűbb az élete a kezelőnek

## DROP TABLE ÉS DELETE(rekord)

## *meglévő tábla bővítése*

**ALTER** **TABLE** `befiz` **ADD** `ugyfel\_azon` **INT**(10) **UNSIGNED** **NOT** **NULL**;

## *update*

**UPDATE** `befiz` **SET** `ugyfel\_azon` = 9001 **WHERE** `azon` = 1001 **OR** `azon` = 1002;

## kreálás után USE – ezt a szervernek majd valahol meg kell mondani, hogy melyikre csatlakozzon rá; server, port, adatbázis, felhasználó, jelszó

## cheatsheet!

## charral szemben a varchar dinamikusan foglal le területet

## week16d02

## relációs adatbázis az öregebb, de kipusztíthatatlan kategória

## relációs adatbázis nem létezik adatbázist kezelő rendszer nélkül, DB és DBMS(ezt telepítjük – valamilyen msi; alapértelmezett sémákat is létrehoz)

## hogy tudok a dbmsel kommunikálni? szükségünk vagy egy kliensre, de nem feltétlen ugyanoda, ahol mi dolgozunk, a fejlesztőgépünkre tegyük; az általunk betáplált sql nyelvű utasításokra képes válaszolni a kliens elküldi ezeket a kéréseket a dbmshez és a server visszaválaszol; telepített server és telepített kliens; mariadb(relációs dbms) és heidisql(kliens)

## SQL – struktúrált adatlekérő nyelv; különböző szabványok

## sqlen kívüli világ, NEM CSAK sqlre képes (no SQL jelentse)

## MariaDB

## majdhogynem mysql

## több képességgel

## az adatbázisok háttérben fileban is tárolva vannak a windows valamilyen mappájában;

## ALTGR+7 ``

## számoknál fontos spórolási tényező, ne legyen pl az id negatív ! UNSIGNED (Ctrl+space)

## not null – legyen kötelező megadni

## INSERT INTO – REKORD BESZÚRÁSA:

**INSERT** **INTO** `befiz` **VALUES** (

1001, '2020-12-12 12:12:12', 455000);

## minden sor törlése:

**DELETE** **FROM** `befiz`**WHERE** 1;

**mindennel spúrnak kell lenni! annyi karakter legyen a számjegy amennyi kell**

## MariaDB - bevezetés az SQL és a relációs adatbázisok világába

## az adat = érték

## nagy adatbázisokból kinyerni infót

## sql nyelv – leíró nyelv, szabványos formája az adatok kinyerésének

## MariaDB

## acid alapelvek

## 

## adatbázis kezelő rendszer = DBMS

## sok adattal dolgoznak és fontos, hogy a rábízott elmentett adatokat ugyanúgy adják vissza

## acid elvek -> azt mondják meg, mit követelünk megy egy dbmstől?

## Atomicitás

## csak akkor tekintjük sikeresnek a tranzakciót, ha minden részművelet is végbement; pl. rendelés során a rendelés adatait eltároljuk, de ha bármelyik adat megsérül és nem kerül mentésre, az nem lesz sikeres

## Konzisztencia

## az adatok a tranzakció előtti érvényes állapotból ismét egy érvényes állapotba kerülnek nem lehet arról szó hogy megsérülnek és valamilyen rossz adat kerül a rendszerbe; nem lehet, olyan hogy egy kimaradás vagy szoftveres hiba miatt félbe marad az adat beszúrása ami miatt az adatbázis olvashatatlanná válik

## Izoláció

## tranzakciók izolációja biztosítja, hogy az egy időben zajló tranzakciók olyan állapothoz vezet ez, mint amit a sorban végrehajtott tranzakciók érnének el; pl összesítem a rendeléske értékét, de közben valaki új rendelést ad le, az új rendelést ne vegyem figyelembe addig, amig az telesen le nem záródott és sikeres lett

## Durability

## Tartósan kell tárolni a tranzakciók és a változások eredményét, nem csak RAMban, hanem a sikeres tranzakciókat l kell tárolni máshol is ahol hosszútávon is elérhetőek lesznek

## MariaDb letöltés és beállítás

## mariadb.org

## Heidsql is együtt települ vele; megnyitottuk, átneveztük az unnnamed kapcsolatot SQLcourse-ra és beállítottuk a root jelszót és megnyitás

## Adatbázisokkal kapcsolatos alapfogalmak

## SQL nyelv részei

## DML - Adat manipuláló

## DDL - Adat definiáló (adatokat létrehozó, adatstruktúrákat meghatározó; != manipulációval)

## DCL – jogokat biztosítja az adatokhoz (Revoke = elvesz)

## TCL – tranzakció kezelést segíti; amikor összefűzünk tranzakciókat egy nagy csomagba és azokat egyszerre hajtjuk végre (pl. lekérünk, módosítunk, frissítünk töröljük egyszerre – összevárjuk ezt a sok utasítást és egy lépésben hajtjuk végre -> ha valamiért nem sikerült, akkor a művelet visszavonható (pl. ha nem sikerült a beszúrás, nem hajtom végre a törlést)

## 

## A különböző kapcsolatok típusai

## Az adatbázisokban az adatokat táblákban vagy táblázatokban tároljuk

## *egy Webshop felépítése:*

## 

## adatbázis -> táblák fejléccel (piros és kék) -> mezők =a táblázat oszlopai; annak fejlécei; oszlopokban cellák, amikben vannak az adatok

## a táblák nem önállóan léteznek, vannak közöttük kapcsolatok

## Production piros fejléces táblák vizsgálata

## categories-product kapcsolat

## egy kategóriához sok termék tartozik - > ahol a háromlábú jel van, az lesz a több

## két párhuzamos vonal jelentése: kötelező, hogy a productnak legyen kategóriája, viszont a kategóriának nem kötelező product -> ezt jelzi a kör a másik oldalon

## ez egy tipikusan egy a többhöz kapcsolat; egy kategóriához több termék tartozik

## product-brands kapcsolat

## a terméknek kell, hogy legyen márkája (két párhuzamos vonal), a márkáknak nem kötelező, hogy legyen terméke (kör); egy márkához sok termék kapcsolódik (háromláb) - > ez is egy a többhöz kapcsolat

## products-stocks

## ez egy az egyhez kapcsolat (egy termékhez egy raktári bejegyzés/polc tartozik)

## Sales kék fejléces táblák vizsgálata

## customers-orders kapcsolat

## egy vásárlóhoz sok megrendelés tartozhat (háromláb), de nem kötelező, hogy legyenek megrendelései (kör); viszont ha van egy megrendelés, az kell, hogy valamilyen vásárlóhoz tartozzon (ha a másikból van, akkor már kötelező -> vonal és kör!); egy a többhöz kapcsolat

## orders-staffs

## egy a többhöz kapcsolat

## staffs

## az adott alkalmazott főnöke is egy alkalmazott; táblán belüli kapcsolat

## 1-1, 1-több és ezek variációi

## Szintaxis, adattípusok és táblák

## az SQL lekérdező nyelv írásmódja = a parancsokat milyen formában kell kiadni a programnak?

## HeidiSQL

## egyenlőre bal oldalt az SQLcourse alatt csak a gyári beállítások, gyári táblák

## Lekérdezés - > ide lehet írni saját SQL parancsokat

## SELECT \* FROM Customers – kiválasztani minden adatot a customer táblából

## parancs nagybetű, tábla kisbetű ;-t tegyük ki mindig!

## új adatbázis létrehozása

## CREATE DATABASE Hambi; majd jobb klikk és futtatás vagy F9

## lenti ablakban írja mit hajtott végre és a válaszokat

## még nem látszott az új adatbázis bal oldalt - > frissíteni kell -> jobb klikk frissítés vagy F5

## új tábla létrehozása

## CREATE TABLE Products (név és típus, felsorolással):

## (

## id int,

## name varchar (255= ennyi char hosszú),

## description varchar(255),

## price int

## );

## lehet egy sorba is, de így áttekinthetőbb

## lefoglalt neveknél dobhat hibát, ilyenkor más név kell (pl desc)

## hambit lenyitottuk, megjelent a Products tábla

## rákatt a táblára és Adat fül mutatja, hogy milyen adatok vannak benne

## adattípusok még:

## float, boolean, blob?(byteok)

## Egyszerű lekérdezések

## SELECT - Adatok kiválasztása az adatbázis tábláiból

## adatok lekérésére

## sintaxis: SELECT \* FROM table\_name;

## (tábla létrehozása grafikus módon a HeidiSQL-ben)

## *tábla létrehozása Lekérdezéssel:*

## CREATE TABLE Customers (

## id int,

## name varchar (100),

## contact varchar (100),

## address varchar(250),

## city varchar(50),

## postal\_code int,

## country varchar (50),

## );

## mezőnév több szóból áll: alulvonással!

## *Töltsük fel a táblát teszt adatokkal:*

## Adat fül – >jobb klikk –> insert row (egyenlőre manuálisan csináljuk)

## adatok beírása után félrekatt fehér területre, erre lent az ablakban lefuttatott egy lekérdezést (INSERT INTO..) amivel beszúrta a táblába az adatokat

## *Duplicate row és pár adat átír*

## CTRL+ - oszlopokat kihúzza

## *Összes kiválasztása:*

## Lekérdezés -> SELECT \* FROM Customers (CTRL+Space felhozza segítségként a táblázat)

## *Csak bizonyos oszlopokat szeretnék az eredménybe látni:*

## SELECT name, city FROM customers

## *Csak egy bizonyos oszlop előforduló értékeinek halmazával térjen vissza:*

## SELECT DISTINCT country FROM customers;

## *+ehhez még nevezzük át a lekérdezett fejléc nevét ’Ország’-ra AS-el:*

## SELECT DISTINCT country AS ’Ország’ FROM customers; szóköz vagy ékezetes betű esetén kell ’’

A legelemibb parancs a SELECT. Segítségével tudsz lekérni adatokat az adatbázisból.  
Ismerjük hát meg.

### SELECT és FROM

A SELECT kiválasztást jelent, azaz kiválasztunk bizonyos adatsorokat vagy azok  
részeit az adatbázis táblából. Ebből már lehet következtetni arra, hogy milyen  
adatokat kell megadnunk ha SELECT parancsot akarunk futtatni. Egy példa:

SELECT \* FROM table\_name;

Tehát a SELECT -ből tudja az adatbázismotor hogy keresni akarunk adatokat. A FROM  
pedig megadja, hogy hol, egészen pontosan honnan szeretnénk ezeket az adatokat  
lekérni. A FROM után jön az adatbázis tábla neve.  
Ez a lekérdezés **az adatbázis tábla teljes tartalmát visszaadja**. Azaz az összes sorát és azon belül az összes oszlopát.

### Column SELECT

Gyakran előfordul hogy nincs szükségünk az összes adatra egy soron belül. Az  
egyes adatok az adatbázis oszlopainak felelnek meg, ennek neve az angol Column.  
Ezt úgy hívjuk hogy szűkítjük a lekérdezést bizonyos oszlopokra:

SELECT CustomerName, City FROM Customers;

**Elemezzük a lekérdezést:**  
A SELECT után megadtunk két oszlopnevet, CustomerName és City. Most minden sort  
visszakapunk ugyanúgy mint a SELECT \* esetében, azzal a különbséggel, hogy soronként  
csak két oszlopunk lesz az eredményben, vagyis két cella ha Excel analógiával  
gondolkodunk. Egy két oszlopból álló táblázatunk lesz.

### SELECT DISTINCT

A DISTINCT azt jelenti hogy különböző - eltérő. Magyarul nem fogja visszaadni  
az összes értéket, csak a különbözőeket. Ha például tízszer szerepel a "Hungary"  
a Country oszopban, mi csak egyszer fogjuk látni az eredményben.

SELECT DISTINCT Country FROM Customers;

Ez főleg akkor jön jól, ha például szeretnél egy listát arról, hogy milyen  
országokból vannak vásárlóid.

## WHERE - Adatok kiválasztása feltételek alapján

## =lekérdezés feltételhez kötve; csak azok kellenek, akik..

## *Csak a magyarokat kérdeztük le:*

## SELECT \* FROM customers

## WHERE country = ’Hungary’;

## idézőjelbe kell, mert nem egy oszlopnév, hanem egy cella érték

## sor = rekord (bejegyzés)

## *Nem szöveg, hanem azonosító alapján kiválasztani a felhasználót*

## SELECT \* FROM customers

## WHERE id =1; -> lehetne a számot is idézőjelbe, de nem szükséges

## = -> operátor

## < > 1, nem egyenlő

## < és > külön, vagy <= és >= (relációs jel elöl!)

## BETWEEN 2 AND 4; (szélsőértékeket is beleveszi!)

## IN (2, 4, 5); - felsorolás operátor

Már szépen ki tudsz választani adatsorokat, sőt akár szűkíteni is tudod őket  
oszlopok szerint. Most kössük feltételhez a sorok kiválasztását is.

### WHERE

Ha magyarul is értene az SQL, akkor azt mondanánk neki:

- Kérem az összes falhasználót aki Mexikóban lalik!

SELECT \* FROM Customers

WHERE Country = 'Mexico';

**Kérem az összes felhasználó összes oszlopát a Customers táblából, ahol a Country oszlop értéke 'Mexico'**

A felszólítás hatására az adatbáziskezelő motor megkeresi a Customers táblát,  
végigmegy a sorokon és minden olyan sort vissza fog adni ahol a Country értéke  
Mexico. Ennyi az egész.  
Tehát a **WHERE** arra való hogy feltételhez kösd a sorok megjelenítését.

### TEXT vs NUMERIC

A fenti példában szöveges mezőben kerestünk, ezért szépen aposztrófok közé tettem  
a lekérdezésben azt hogy 'Mexico'. Mi van ha szám alapján akarok keresni?

SELECT \* FROM Customers

WHERE CustomerID = 1;

Ahogy látod, szám esetén nem kell aposztróf. Itt az 1 -es id -jű vásárló rekordját  
fogja visszaadni az adatbázis motor. Jegyezd meg, SQL esetén a sorokat rekordoknak  
nevezzük. Mert ezek gyakorlatilag bejegyzések az adatbázis táblában.

### Operators

A WHERE feltételben úgynevezett összehasonlító operátorokat használunk. Például  
kisebb, nagyobb, egyenlő stb. Itt egy lista róluk, pár példával fűszerezve.

1. = Egyenlő (WHERE CustomerID = 1)
2. <> Nem egyenlő (WHERE CustomerID <> 1)
3. > Nagyobb mint (WHERE CustomerID > 1)
4. < Kisebb mint (WHERE CustomerID < 10)
5. >= Nagyobb vagy egyenlő (WHERE CustomerID >= 1)
6. <= Kisebb vagy egyenlő (WHERE CustomerID <= 10< em>)
7. BETWEEN A megadott tartományon belül (WHERE CustomerID BETWEEN 10 AND 30)
8. LIKE Hasonló (WHERE Country LIKE 'Hun%') részletesen később
9. IN A felsorolásból valamelyik (WHERE CustomerID IN (1, 4, 7, 99))

### Dokumentáció:

[SQL WHERE](https://www.w3schools.com/sql/sql_where.asp)

### Önálló feladatok

**Töltsd fel az adatbázist!**

A második témánál létrehozott adatbázisba vedd fel a következő adatokat:

1. A Teacher tábla tartalmazza a Mentorok adatait akik most tanítanak téged.
2. A Course tábla tartalmazzon négy kurzust: basic, dotnet, java, frontend
3. A Student táblába pedig vedd fel az összes csoporttársadat.

**Írj lekérdezéseket és mentsd el őket.**

Most szűrjük az adatokat különböző lekérdezésekkel.

1. Először a Teacher táblából válaszd ki az összes debreceni mentort.
2. A Studnet táblából pedig válaszd ki az összes olyan budapesti tanulót akinek  
   gmail -es az email címe.

## AND, OR, NOT - Feltételek összekapcsolása a lekérdezésekben

## kibővítés és összefűzés

## *AND - Magyarországi budapestiek:*

## SELECT \* FROM customers

## WHERE

## country = ’Hungary’

## AND

## city = ’Budapest’;

## *OR – Magyarországi vagy budapesti:*

## *SELECT \* FROM customers*

## WHERE

## country = ’Hungary’

## OR

## city = ’Budapest’;

## *Tagadás – NOT :*

## *SELECT \* FROM customers*

## WHERE

## NOT country = ’Germany’;

## *Kombinálás:*

## *SELECT \* FROM customers*

## WHERE

## country = ’Hungary’

## AND

## city = Berlin;

## OR

## city = Budapest’; // a belrines rekordot nem veszi bele

## Ugyanez zárójellel – először kiértékeli a zárójelest, abból lesz egy igaz vagy hamis érték, és hozzánézi az első feltételt

## *SELECT \* FROM customers*

## WHERE

## country = ’Hungary’

## AND

## (

## city = Berlin;

## OR

## city = Budapest’

## ) // a berlines rekordot is beleveszi

A WHERE parancs után jönnek a feltételek. Ezek között is lehetnek logikai  
kapcsolatok és vannak is. Az egyes feltételeket fogjuk most összefűzni.

### AND

A feltételek között logikai kapcsolat áll fenn, ezen belül az AND kapcsolat  
a legegyszerűbb. Az ismert programozási szabályok vonatkoznak rá, azaz akkor lesz  
igaz, ha az mindkettő feltétel igaz azok közül amelyek az AND két oldalán állnak.

SELECT \* FROM Customers

WHERE Country="Germany" AND City="Berlin";

Ez egy tipikus AND kapcsolat. Akkor fog a lekérdezés eredményébe bekerülni egy  
rekord, ha a vásárló országa Germany és a városa Berlin. Tehát ha mondjuk  
Németországban lakik az illető de München -ben és nem Berlinben, akkor nem lesz  
látható az eredmények között. Jegyezd meg, az AND feltétel esetén mindkét oldalnak  
igaznak kell lennie.

### OR

Nem nehéz kitalálni, hogy ez ugyanaz lesz mint mondjuk JAVASCRIPT estén a két  
függőleges vonallal jelölt "||" or feltétel. Tehát ha a két feltétel közül csak  
az egyik igaz, akkor is igaz lesz az egész kifejezés.

SELECT \* FROM Customers

WHERE City='Berlin' OR City='München';

Ez a lekérdezés mindenkit vissza fog adni aki Berlinben vagy Münchenben lakik.

### NOT

A NOT szaknyelven a negáció. Azaz ha megfordítom valaminek a logikai értékét,  
azaz tagadom. Ez JAVASCRIPT esetén a felkiáltójel "!".

SELECT \* FROM Customers

WHERE NOT Country='Germany';

Így mindenki meg fogunk kapni a Customers táblából aki nem németországban lakik.

### Kombináljunk

A fenti kifejezéseket tetszőleges módon kombinálhatod is.

SELECT \* FROM Customers

WHERE Country='Germany' AND (City='Berlin' OR City='München');

Nézzük sorban.

1. Először bontsuk fel a zárójelet: mindenkit visszaad, akinek Berlin vagy München  
   a városa.
2. Az AND miatt pedig csak azokat fogja visszaadni akikre igaz az első feltétel  
   és még ennek tetejében Germany az országuk is.

Összefoglalva, mindenki aki Németországban lakik és azon belül a városa  
Berlin vagy München.

### Dokumentáció:

[AND OR NOT](https://www.w3schools.com/sql/sql_and_or.asp)

### Önálló feladatok

**A YellowRoad adatbázisban dolgozz!**

1. Készíts lekérdezést, amely kiválasztja a Teacher táblából az összes tanárt  
   akinek 1-nél nagyobb az id-je és Budapesten vagy Debrecenben lakik.
2. Készítsd el ugyanezt a Student táblára is.

## ORDER BY - Találatok sorba rendezése

## lekérdezés kombinálása sorba rendezéssel valami logika alapján; lássuk pl. a 10 legtöbbet költő vásárlót

## =ORDER BY parancs

## *Sorbarendezés ország szerint (abc):*

## SELECT \* FROM customers

## ORDER BY country;

## *Sorbarendezés ország szerint, azon belül pedig név alapján:*

## SELECT \* FROM customers

## ORDER BY country, name;

## a név itt alárendelt rendezési szempont; csak az adott országon belül rendez így

## *Rendezés iránya legyen csökkenő – DESC* (az alapértelmezett az ASC=Ascending)

## SELECT \* FROM customers

## ORDER BY country DESC, name;

## SELECT \* FROM customers

## ORDER BY country ASC, name DESC;

Alapvető igény hogy sorba lehessen rendezni az adatokat amelyek az adatbázisból  
jönnek. Ezt megtehetjük kliens oldalon is, de sokkal gyorsabb ha már eleve sorban  
érkeznek.

### Legyen egy táblánk

Az egyszerűség kedvéért a példákat az alábbi táblázaton fogom értelmezni:



### ORDER BY alapok

Az ORDER BY utasításnak mindig a WHERE után kell állnia, mivel amíg nem szűrtük  
le az adatokat világos hogy nincs mit sorbarendezni sem. Ha nincs WHERE utasítás,  
azaz minden rekordot lekérünk egy táblából, akkor jöhet közvetlenül is a  
FROM táblanév után.

SELECT \* FROM Customers

ORDER BY Country;

Így az összes felhasználót megkapod, az országuk alapján emelkedő sorrendben.  
Azaz a Germany után fog jönni a Hungary például.

### ORDER BY több mezőre

Oké, de mi van akkor ha nem csak egy feltétel alapján akarsz rendezni. Mondjuk  
szeretnéd ha elsősorban ország alapján lennének sorban, de azon belül a nevük  
szerint a vásárlók.

SELECT \* FROM Customers

ORDER BY Country, CustomerName;

### ASC, DESC

Eredetileg szép latin szavak ezek, csak a Brittek lenyúlták őket.

1. **ascending**: emelkedő
2. **descending**: csökkenő

Ezek a rendezés irányára vonatkoznak. Alapértelmezett az emelkedő, de ezt  
bármikor felül tudod bírálni.

SELECT \* FROM Customers

ORDER BY Country ASC, CustomerName DESC;

Ahogy látod, az irányt oszoponként is módosíthatod. A példában az országok  
A -> Z irányban, a nevek Z -> A irányban lesznek rendezve.

### Dokumentáció:

[ORDER BY](https://www.w3schools.com/sql/sql_orderby.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd meg és futtasd!**

1. Készíts lekérdezést, amely kiválasztja a Teacher táblából az összes tanárt  
   akinek 1-nél nagyobb az id-je és Budapesten vagy Debrecenben lakik, id alapján  
   csökkenő de a tanár neve alapján növekvő sorrendban.
2. Készítsd el ugyanezt a Student táblára is, csak itt az email cím alapján legyenek  
   csökkenő sorrendben az adatok.

## MIN, MAX - A minimum és maximum értékek keresése

aggregálás = csoportosítás vagy összesítés

*Szélsőérték vizsgálat, min és max, alias névvel:*

SELECT **MIN**(price) AS SmallestPrice

FROM products;

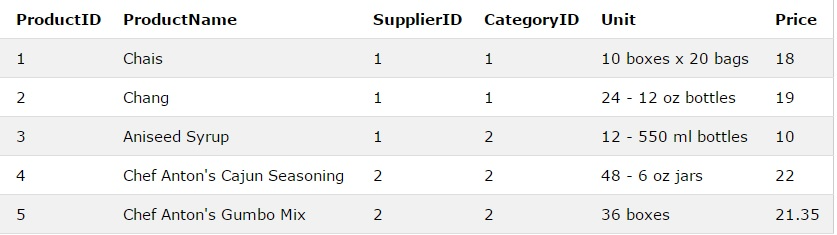
SELECT **MAX**(price) AS LargestPrice

FROM products;

Aggregate. Csoportosításnak vagy összesítésnek is lehet fodítani, a lényeg hogy  
nem egy konkrét mező adatait szeretném lekérni hanem összesített adatokat.  
Ha jól megnézzük ezek igazából függvények, zárójelekkel és paraméterekkel ahogy  
más nyelvekben is.

### Dolgozzunk ebben

A példakódokhoz az alábbi táblázatot fogom használni:



### MIN()

Tehát ez a MIN() függvény. Ahogy a neve is mutatja a megadott értékek közül a  
legkisebbet fogja kiválasztani.

SELECT MIN(Price) AS SmallestPrice

FROM Products;

Ez a legalacsonyabb árat fogja visszaadni SmallestPrice néven. Fontos hogy nem  
rekordokat ad vissza, hanem egy konkrét értéket.

### MAX()

Az előbbi ellentéte, ez a legnagyobb árat fogja visszaadni, LargestPrice néven.

SELECT MAX(Price) AS LargestPrice

FROM Products;

### Dokumentáció:

[MIN & MAX](https://www.w3schools.com/sql/sql_min_max.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Vegyél fel egy új oszlopot a Student táblába a student\_email oszlop után.  
   Neve: "credit", típusa: "Int". Töltsd fel ízlés szerint számokkal az összes rekord  
   esetén.
2. Írj lekérdezést a Student táblára, ami MaxCredit néven visszaadja a legmagasabb  
   kreditet a Budapesti tanulók közül.

## COUNT, AVG, SUM - Megszámlálás, átlagolás és összegzés

## COUNT *– rekordok számát adja vissza:*

SELECT **COUNT**(id)

FROM products

WHERE category\_Id = 2;

## *Egyik product árát nullra rakta:*

SELECT COUNT(price)

FROM products

WHERE category\_Id = 2;

* *>* itt már csak egyet adott vissza, mert a price alapján számolta össze a rekordokat és a *nullt nem vette bele*

## AVG – átlag

SELECT AVG(price)

FROM products;

## SUM – összeg

SELECT SUM(price)

FROM products;

SELECT SUM(price)

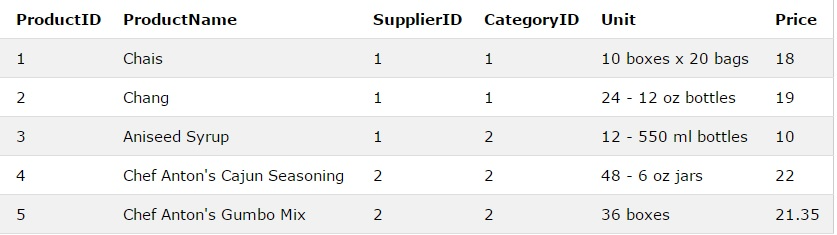
FROM products;

## WHERE category\_id=1;

Újabb aggregáló függvények következnek. Most számlálni, átlagot számítani és  
összesíteni fogunk.

### Minta táblázatunk

A példakódokhoz az alábbi táblázatot fogom használni:



### COUNT()

A megadott feltételeknek megfelelő rekordok számát adja vissza.

SELECT COUNT(ProductID)

FROM Products WHERE SupplierID = 2;

Itt a fenti mintatábla alapján 2 lesz az eredmény. Azért mert a ProductID mezők  
számát kérem le, azaz a rekordok számát. De leszűkítettem a rekordok körét, mert  
csak a 2 -es SupplierID -vel rendelkezőkre vagyok kíváncsi, abból pedig csupán  
kettő van.

### AVG()

Most számoljunk átlagot. Az átlagszámítás szabálya szerint az összesített értéket  
osztja a rekordok számával.

SELECT AVG(Price)

FROM Products WHERE SupplierID = 1;

Itt sem az összes rekordot vizsgálom, figyeld meg hogy csak az 1-es SupplierID  
érdekel. Vajon mennyi lesz az eredmény?

(18 + 19 + 10) / 3 = 15.66

### SUM()

Ez pedig a jó öreg Summa. Azaz összesítjük, summázzuk az adatokat. (Ezek az  
angolok szinte a teljes latin szótárt átvették.)

SELECT SUM(Price)

FROM Products;

Így az eredmény nem meglepően: 90.35

### Dokumentáció:

[COUNT(), AVG(), SUM()](https://www.w3schools.com/sql/sql_count_avg_sum.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Írj lekérdezést a Student táblára, ami SumCredit néven visszaadja a kreditek  
   összegét a nem Budapesti tanulók esetén.
2. Írj lekérdezést a Teacher táblára, ami NumTeachers néven visszaadja a  
   Budapesti tanárok számát.
3. Írj lekérdezést a Course táblára, ami Actual néven visszaadja a  
   jelenleg futó kurzusok számát.
4. Írj lekérdezést a Student táblára, ami AvgCredit néven visszaadja a kreditek  
   átlagát a Budapesti tanulók esetén.

## GROUP BY - Találatok csoportosítása

## Adatokat nem csak sorba rendezni, hanem csoportosítani is lehet

## =GROUP BY

## séma: csak a lekérdezés logikáját mutatja be, nem konkrét nevekkel:

## 

## Tábla: products – ebben a nézetben tudok oszlopot beszúrni; beszúrt a products-ba egy category id-t, majd INSERT INTO-val beszúrt még pár sort

## végrehajtási sorrend: adatbázis motor nem a beírás sorrendjében, hanem logikai sorrendben futtat (1.From, 2.Where, 3.Group by, 4.Select, 5.Ordered By – nem biztos, hogy minden adatbázis motor megeszi a saját sorrendet..)

## élő példa:

## SELECT category\_id

## FROM products

## GROUP BY category\_id;

Nem csak arra van lehetőséged hogy sorba rendezd az adatokat, de csoportosíthatod  
is őket.

### Használjuk ezt a táblázatot



### A séma

Itt már bonyolódnak kicsit a dolgok, ezért először mutatok egy elméleti sémát.

SELECT column\_name(s)

FROM table\_name

WHERE condition

GROUP BY column\_name(s)

ORDER BY column\_name(s);

**Ami fontos. A GROUP BY mindig az ORDER BY előtt van.** Azért mert a csoportosításnál  
megváltoznak a megjelenített adatok, így azokat ismerni kell ahhoz hogy sorba  
tudjuk rendezni őket.

### GROUP BY

Mivel az ország alapján csoportosítjuk a rekordokat, így nem annyi rekord fog  
megjelenni az eredményben ahányszor szerepel az adott ország a táblában. Például  
a következő lekérdezésben a mexikóiak így lesznek benne: 2, Mexico

SELECT COUNT(CustomerID), Country

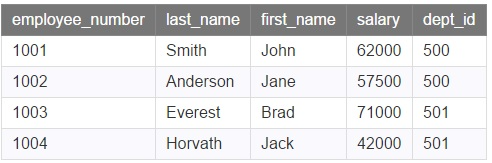
FROM Customers

GROUP BY Country;

### Másik példa

Ebben a példában munkavállókkal dolgozunk. Azt látjuk hogy a részleg id-je nem  
mindig különbözik, ketten - ketten ugyanazon a részlegen dolgoznak. Nem vagyok most  
kíváncsi az egyes dolgozók adataira, csupán azt szeretném megtudni hogy melyik  
részlegen mennyi az összesített fizetés.

**Tábla:**



**Lekérdezés:**

SELECT dept\_id, SUM(salary) AS total\_salaries

FROM employees

GROUP BY dept\_id;

**Eredmény:**



### **Végrehajtási sorrend**

Az adatbázis kezelő nem abban a sorrendben hajtja végre a lekérdezéseket, ahogyan azokat megírjuk. Az értelmezési sorrend a következő:

1. **FROM**: kiválasztja a táblát vagy join -olt táblákat amelyekben keres
2. **WHERE**: kiválasztja a rekordokat a megadott feltételek alapján
3. **GROUP BY**: csoportosítja a rekordokat ha szükséges
4. **SELECT**: kiválasztja a megjelenítendő cellákat
5. **ORDER BY**: sorbarendezi a találatokat

A végrehajtási sorrend azért fontos, mert például ha a FROM résznél álnevet adsz a tábláknak, akkor már azzal kell hivatkoznod a lekérdezés többi részében is rájuk. Vagy nem tudod a WHERE részben használni a SELECT -nél megadott álneveket, mivel a SELECT később fut le.

### Dokumentáció:

[GROUP BY](https://www.w3schools.com/sql/sql_groupby.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Írj lekérdezést a Student táblára, ami a tanulók számát város szerint csoportosítva adja vissza.

## LIKE - Keresés hasonlóság alapján

## =összehasonlító operátor

## séma:

## 

## patternek:

## ’a%’ – bármi, ami a betűvel kezdődik

## ’\_r%’ – egy bármilyen karakter

## ’a\_%\_%’ – legalább három karakter hosszú és a-val kezdődő

## élő példa:

## SELECT \*

## FROM customers

## WHERE name LIKE ’a%’

## ..

## WHERE name LIKE ’\_e%’

Nem mindig tudunk exakt adatot megadni amikor keresünk valamire. Például egy  
keresőmezőben a felhasználó csak egy szórészletet ad meg, vagy nem tudja a teljes  
nevet, csupán a vezeték vagy keresztnevét valakinek. Ilyenkor jön jól a LIKE.

### LIKE elmélet

Az alábbi séma mutatja a LIKE használatát.

SELECT column1, column2, ...

FROM table\_name

WHERE columnN LIKE pattern;

Azaz nem =, \< vagy > hanem az összehasonlító operátor a LIKE. Hasonló, vagy  
részleteiben megegyezik.

A LIKE után az alábbi szöveg helyettesítő karaktereket használhatod:

* "\_": bármilyen karakter, de csak egy darab
* "%": bármilyen karakter bárhányszor

**Példák:**

* WHERE CustomerName LIKE 'a%': minden ami a betűvel kezdődik
* WHERE CustomerName LIKE '%a': a -ra végződik
* WHERE CustomerName LIKE '%or%': bárhol szerepel benne hogy 'or'
* WHERE CustomerName LIKE '\_r%': az r a második karakter
* WHERE CustomerName LIKE 'a\_%\_%': a -val kezdődik és legalább három karakter hosszú
* WHERE ContactName LIKE 'a%o': a- val kezdődik és o-ra végződik

### Demo tábla



### LIKE

Akiknek a neve a-val kezdődik (4 rekord):

SELECT \* FROM Customers

WHERE CustomerName LIKE 'a%';

Akiknek a nevében benne van az 'or' (2 rekord):

SELECT \* FROM Customers

WHERE CustomerName LIKE '%or%';

Akiknek a nevében a második betű 'r' (1 rekord):

SELECT \* FROM Customers

WHERE CustomerName LIKE '\_r%';

### Dokumentáció:

[SQL LIKE](https://www.w3schools.com/sql/sql_like.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Írj lekérdezést a Student táblára, ami összesítve visszaadja az összes olyan  
   tanuló kreditjeit, akinek gmail -es az email címe.

**ÖSSZETETT LEKÉRDEZÉSEK ÉS ADATMANIPULÁCIÓK**

**INSERT – Új rekordok beszúrása az adattáblákba**

séma:

**INSERT** **INTO** `table\_name` (**COLUMNS**) **VALUES** (**values**)

oszlopfelsorolásnál ugyanaz a sorrend kell, mint az értékfelsorolásnál!

úgyis meg tudok adni érétkeket, ha kihagyom az oszlopnevet viszont akkor egyetlen oszlopot sem hagyhatok ki! oszlop megadással meg tetszőleges számú oszlop értékét inicializálhatom, vagy ha nem adtma neki értéket, akkor null less (ha engedi)

egyszerre több rekord beszúrása – felsorolással, ctrlc ctrlv, sorok végére , legvégére ;

sql dumpban 50 rekordonként van egy lekérdezés – külön lementett sql fileok futtatása

Kevés lenne ha csak lekérni tudnánk adatokat, fontos hogy tudjunk beszúrni új  
sorokat is az adatbázisunk tábláiba.

### INSERT INTO

Tehát ezt úgy hívják hogy INSERT. Az INTO kulcsszó után kell megadni a tábla  
nevét, ahová a beszúrás meg fog történni.

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)

VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

FONTOS! Az adatok sorrendje kötött, azaz ahogy a tábla neve utáni zárójelben  
megadod az oszlopneveket, ugyanabban a sorrendben kell megadnod az adatokat is a  
VALUES kulcsszó után. Nem lehet sem több, sem kevesebb adat, ugyanannyi adatnak  
kell lennie a felsorolásban ahány oszlopnevet megadtál. Ha ezektől eltérsz, hibára  
fut a lekérdezésed és nem kerülnek be az adatok az adatbázisba mert hibás lesz a  
lekérdezésed.

### Oszlopnevek nélkül

Arra is van lehetőséged, hogy nem adod meg az oszlopok nevét. Ekkor viszont  
minden adatot abban a sorrendben kell felsorolnod ahogy az adattábla oszlopai  
egymás után jönnek.

INSERT INTO Customers

      VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

Így a VALUES kulcsszót nem előzi meg az oszlopnevek zárójelbe tett listája.

### Egyszerre több rekord

Egy lekérdezéssel több rekordot is beszúrhatsz. Ekkor a VALUES kulcsszó után  
egy felsorolást kell megadnod.

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)

VALUES

('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway'),

('King', 'Koen Jensen', 'Oslo 21', 'Oslo', '7000', 'Norway'),

('Cool', 'Viking Kirksen', 'Berlin 44', 'Bergen', '2001', 'Norway');

### Dokumentáció:

[INSERT INTO](https://www.w3schools.com/sql/sql_insert.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Szúrj be három új felhasználót lekérdezéssel a Students táblába. A neveik sorban  
   legyenek: "Robert Downey Jr., "Jon Favreau", "Gwyneth Paltrow". A többi adat  
   lehet szabadon választott.

## UPDATE - Az egyes rekordok frissítése

## séma:

**UPDATE** `table\_name` **SET** **column** = 'value', city = 'Frankfurt' **WHERE** id = 1;

## mindig legyen WHERE feltétel! ha nincs, mindenre lefut

## ha mégis több rekordra szeretnék, akkor is egy általánosabb WHERE feltétellel

Mostmár le tudod kérni az adatokat, sőt fel is tudsz vinni újakat. Viszont  
frissíteni még nem tudod őket. Tanuljuk meg ezt is.

### Legyen egy táblánk

Az egyszerűség kedvéért dolgozzunk ugyanazzal a táblával:



### UPDATE table

Itt az UPDATE kulcsszóval kell dolgoznod. Ezután meg kall adnond a tábla nevét  
amit frissíteni akarsz, majd a mezőket és az új értékeket egy felsorolásban.

UPDATE Customers

SET ContactName = 'Alfred Schmidt', City = 'Frankfurt'

WHERE CustomerID = 1;

Ahogy látod, ez eltér a SELECT vagy INSERT logikájától, mert nem külön adod meg  
a mezőneveket és az értékeket hanem rögtön egymás után.

### WARNING!!! WHERE

Nagyon figyelj oda, hogy mindig legyen WHERE feltétel az UPDATE utasítás végén.  
Miért? Hát azért mert ha nem adsz meg feltételt, akkor az összes rekord frissülni  
fog, tehát olyan adatokat is könnyen felülírhatsz, amelyeket nem szerettél volna.  
És gyakran előfordul hogy ezek az adatok milliókat érnek. Ha nem akarsz fizetni,  
ne feledkezz meg a WHERE feltételről.

### UPDATE multiple

Az első példában csak egy rekord frissült, mivel a WHERE feltételben megadtam  
hogy csak az 1 -es id -jű Customer frissüljön. De előfordulhat olyan is, hogy  
több rekordot is szeretnék frissíteni egyszerre. Ekkor csak általánosabbá kell  
tennem a feltételt.

UPDATE Customers

SET ContactName='Juan'

WHERE Country='Mexico';

Itt az összes mexikóinak "Juan" lesz a kapcsolattartói neve.

### Dokumentáció:

[UPDATE](https://www.w3schools.com/sql/sql_update.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Adj hozzá a Student táblához egy új oszlopot a végén. Legyen a neve "active",  
   típusa Boolean.
2. Nézd meg a Student táblában hogy hányas student\_id -nél kezdődnek a színészek.  
   Minden rekordot frissíts le úgy hogy az active legyen 1, ahol a student\_id kisebb  
   mint az első színész student\_id -ja. Magyarul az igazi tanulók legyenek aktívak.

## DELETE - Rekordok törlése

CRUD – Create, Read, Update, Delete

itt is legyen WHERE!

cellákat nem tudok törölni, cska komplett rekodrdokat

ha mégis cellaadatot akarok törölni, null értéket kell neki megadni

Az alapvető műveletek közül nincs más hátra mint a törlés.

### DELETE FROM

Itt is van FROM, ugyani meg kell adnod hogy melyik táblából szeretnél törölni.  
A törlés csak teljes rekordra értelmezett, azaz nem tudsz egy cellából adatot  
törölni, csak egy egész adatsort. Ha mégis csak egy cellából akarsz törölni, akkor  
például egy UPDATE paranccsal be tudod állítani az értékét NULL -ra.

DELETE FROM Customers

WHERE CustomerName='Alfreds Futterkiste';

Láthatod, hogy a törlést ahhoz a feltételhez kötöttem hogy a vásárló neve  
'Alfreds Futterkiste' legyen. Ennek hatására az 'Alfreds Futterkiste' nevű  
vásárló teljes rekordja, azaz adatsora törlődni fog a megadott táblából.

### WARNING!!! WHERE

Emlékszel, az UPDATE -nél azt mondtam hogy nagyon fontos a WHERE feltétel. Ez  
itt hatványozottan igaz, mert míg az UPDATE esetén előfordulhat hogy olyan adatokat  
is felülírsz amit nem szerettél volna, itt WHERE nélkül, akár

**a tábla teljes tartalmát törölheted!**

Persze a tábla megmarad, így a struktúrája, indexei beállításai nem vesznek el.

### Dokumentáció:

[SQL DELETE](https://www.w3schools.com/sql/sql_delete.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Nézd meg a Student táblában hogy hányas student\_id -nél kezdődnek a színészek.  
   Töröld ki egy lekérdezéssel a színészeket, a feltételben azt kösd ki hogy egy  
   bizonyos id -nél nagyobb legyen a student\_id.

## JOIN - Táblák összekötése

egyszerre több táblából is szeretnénk lekérni adatot, egy lekérdezéssel

timestamp típus időbélyeg?

**CREATE** **TABLE** `orders` (

id **INT**,

customer\_id **INT**,

product id **INT**,

order\_date **TIMESTAMP**

);

új táblába felvettünk pár rekordot, majd összekötjük a customers táblával:

tábla.oszlop! Az ‘ON’ után megadott alapján veszi összetartozónak a két táblából az egyes rekordokat

**SELECT** `orders`.id, `customers`.**name**, `orders`.order\_date

**FROM** `orders`

**INNER** **JOIN** `customers` **ON** `orders`.customer\_id = `customers`.id;

**ALTGR+7!**

**Milyen fajta joinok vannak?**

inner – csak azokat a rekordokat fogja kihozni, ahol van kapcsolat (ha lenne olyan rendelés, ami nem kapcsolódik egy emberhez sem, azt nem)

left join (bal oldali az, amihez csatolom, ami a FROM után van) – balból az

előzőn felül kihozza azokat is, amiknek nincs kapcsolódása a másihoz, de a job oldaliból csak a kapcsoltat

right – fordítva

full outer join – mindent kihoz

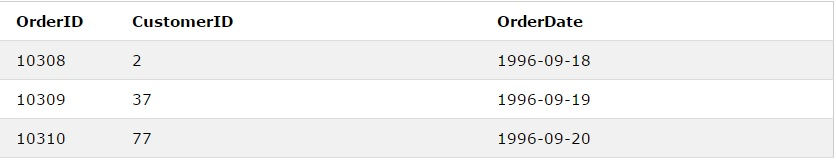
Gyakran előfordul hogy egyszerre több táblából is szeretnénk adatokat kinyerni.  
Ekkor jön jól a JOIN, ami csatlakoztatja a táblákat és úgy futtatja a lekérdezést.

### JOIN

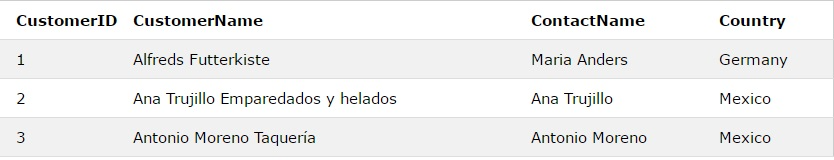
Mit is csinál a JOIN? A különböző táblák rekordjait összefűzi úgy, mintha egy  
táblát alkotnának, így azon lehet futtatni a lekérdezéseket.

### Két tábla az elmélethez

**Orders:**



**Customers:**



A két tábla kapcsolatban van egymással, mivel a rendelésnél nyilván kell tartani  
hogy melyik vásárló rendelt. Ezért az Orders táblában van egy CustomerID mező,  
ami a Customer tábla egy bizonyos sorára hivatkozik.

### Lekérdezés JOIN -al

Mi történik ha a következő lekérdezést lefuttatom?

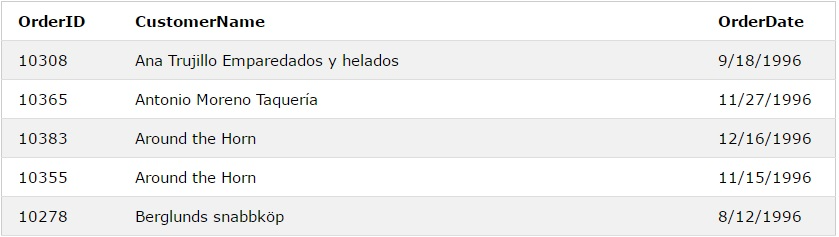
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate

FROM Orders

INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

1. Megadom a mezőket. Figyeld meg, hogy nem csak a mező nevét, hanem a tábla nevét  
   is megadtam. Ez azért kell mert a két táblában vannak azonos nevű mezők, így az SQL  
   nem tudná melyikkel akarok dolgozni.
2. Megadom hogy milyen feltétel alapján válogassa össze a rekordokat. Azt vegye egy  
   rekordnak ahol az Orders.CustomerID megyegyezik a Customers.CustomerID -val.

Ez lesz az eredmény:

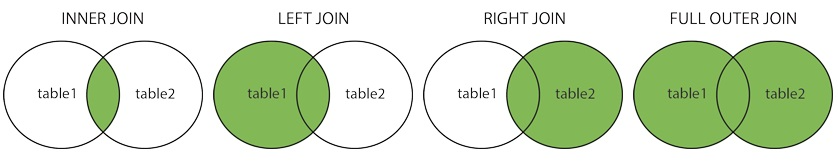


### A JOIN fajtái

Ez egy kis halmaz elmélet lesz. Fogjuk fel úgy a táblázatokat mint halmazokat.  
Tudjuk hogy ezeknek többféle metszete is lehet. A JOIN esetén az alábbi négy fajta  
esetet lehet megkülönböztetni:

1. (INNER) JOIN: csak a közös rész. Ha egyik táblában nem szerepel kapcsolódó  
   rekord, akkor mindkét rekord kimarad. Ez az alapértelmezett.
2. LEFT (OUTER) JOIN: minden a bal táblából és kapcsolat esetén a jobb is
3. RIGHT (OUTER) JOIN: minden a jobb táblából és kapcsolat esetén a bal is
4. FULL (OUTER) JOIN: minden a jobb és a bal táblából is

Itt egy kép, ez jól mutatja hogyan működnek:



### Dokumentáció:

[SQL JOIN](https://www.w3schools.com/sql/sql_join.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Vegyél fel egy új oszlopot a Student táblában a 'student\_email' után. A neve  
   'teacher\_id' legyen, a típusa Int. A Teacher táblában létező 'teacher\_id' -k közül  
   belátásod szerint vegyél fel egyet minden tanulóhoz. Ezzel rendeld hozzá a tanulókat  
   a tanárokhoz.
2. Írj lekérdezést a Student táblára, ami a hozzá 'teacher\_id' alapján kapcsolódó  
   Teacher táblából visszaadja a tanár nevét és a hozzá kapcsolt diákok számát.

## Join megvalósítása összerendelő tábla segítségével

## Két táblában található adatok összekötése – hogyan?

## Felveszem az egyik táblába a másik egy id-ját, de ez nem annyira rugalmas, mint:

## az összerendelő tábla

## mi van, ha pl egy termék több kategóriához is tartozik?

az összerendelő tábla egyértelműen megadja a hozzárendelést, egy idhoz több record is

**grafikusan hoztunk létre 2 új táblát!** obb klikk a hambira és új létrehozása

*egyszerűsítések: Aliast és azt Innert sem írta már ki!*

**SELECT** \*

**FROM** products p

**JOIN** product\_to\_category ptc **ON** ptc.product\_id = p.Productid

**JOIN** categories c **ON** c.id = ptc.category\_id

**ORDER** **by** c.id **DESC**;

## Egy tábla csatolása saját magához

egyezéseket keresünk, vagy párokba szeretnénk rendezni a felhasználókat -> saját magával joinoljuk a táblát (self join)

azonos városok, vagy országok

**SELECT** A.**name** **AS** CustomerName1, B.**name** **AS** CustomerName2, A.city

**FROM** customers A, customers B

**WHERE** A.id <> B.id

**AND** A.city = B.city;

**Megszorítások**

**Adatok integritásának, épségének megőrzése különböző megszorításokkal**

* 1. bizonyos hosszúságú legyen vagy formátumú
  2. ne ismétlődjenek adatok

1.

**ALTER** **TABLE** products

**MODIFY** ProductName **VARCHAR**(100) **NOT** **NULL** **CHECK** (**CHAR\_LENGTH**(ProductName) > 5);

**INSERT** **INTO** products (ProductId, `ProductName`) **VALUES** (10, 'kapa');

2.egyediség – **UNIQUE kulcsszó!**

**MODIFY** ProductName **VARCHAR**(100) **NOT** **NULL** **UNIQUE** **CHECK** (**CHAR\_LENGTH**(ProductName) > 5);

* valami nem volt ok! nnem egedte

## Sorrend, névvel ellátott megszorítások

Összetett megszorítások, több mezőre, és a megszorítást névvel lássuk el

pl két mező kombinációja ne ismétlődhessen; név és contact

CONSTRAINT! = megszorítás

**ALTER** **TABLE** customers

**ADD** **CONSTRAINT** unique\_name\_pairs **UNIQUE** (`name`, `contact`);

**INSERT** **INTO** customers (`name`, `contact`) **VALUES** ('Béci', 'Lali');

**INSERT** **INTO** customers (`name`, `contact`) **VALUES** ('Béci', 'Lali');

* duplicate entryre hivatkozva nem egedte

ALTER TABLE customers

**DROP** **CONSTRAINT** unique\_name\_pairs;

## Értékek automatikus növelése

**auto increment –** cella értékét az előzőhöz képest megnöveli eggyel, ha beszúrok egy rekordot

**ALTER** **TABLE** `customers` **MODIFY** **COLUMN** `id` **int** **NOT** **NULL** **AUTO\_INCREMENT**;

* nem ment!

beszúrás után kitöröltük és újra beszúrás – 5ös után 7es jött, nem megint 6os; hogyan szabályozzuk ezt?

**ALTER** **TABLE** `customers` **AUTO\_INCREMENT** = 6;

**Index**

speciális memória terület az SQLen belül – külön kigyűjti az adatokat és sokkal gyorsabban tud bennük keresni

ha pl név és contact alapján sokszor keresek, létrehozok egy indexet, ami csak a nevet és kontaktot tartalmazza, **gyorsabb** lesz, de plusz memóriaterületet foglal, mert külön kigyűjti a nem azonos értékpárokat (hátrány)

**CREATE** **INDEX** name\_contact

**ON** customers (**NAME**, contact);

felhasználás – keresés név és kontakt alapján

táblában az indexeknél mutatta is

Van hogy nem mi akarjuk megadni egy mező értékét, hanem az adatbáziskezelőre  
bízzuk hogy új rekord beszúrása esetén egyel növelje azt.

### AUTO\_INCREMENT

Most megoldjuk, hogy a Person tábla ID oszlopa automatikusan növekedjen.

CREATE TABLE Persons (

    ID int NOT NULL AUTO\_INCREMENT,

    LastName varchar(255) NOT NULL,

    FirstName varchar(255),

    Age int,

    PRIMARY KEY (ID)

 );

Ha nem szeretnéd hogy a kezdőérték ahonnan a növelést kezdi 1 legyen, akkor egy  
ALTER TABLE paranccsal át tudod állítani, persze miután a táblát létrehoztad.

ALTER TABLE Persons AUTO\_INCREMENT=100;

Ezután beszúrunk kettő új rekordot:

INSERT INTO Persons (FirstName,LastName)

VALUES ('Lars','Monsen'), ('Kis', 'Kirsten');

Az elsőnek 101, a másodiknak 102 lesz az ID -je.

### INDEX

Végül jöjjenek az indexek. Ezeket külön memória területen tárolja az adatbázis  
kezelő, olyan mintha egy rövidített kivonata lenne a táblából. Ha egy lekérdezést  
futtatunk amiben a WHERE feltételben az index mezői vannak, akkor nem fogja végignézni  
a teljes táblát, hanem az indexek között megkeresi hogy hol van a kért adat és az  
alapján fogja megtalálni. Mivel ezek az indexelt adatok sokszor ismétlődhetnek is,  
így sokkal kevesebb rekordot kell végignéznie mintha az eredeti táblában keresne.

CREATE INDEX idx\_pname

ON Persons (LastName, FirstName);

Ez az index akkor hasznos, ha gyakran keresünk teljes névre.

### Dokumentáció:

[AUTO\_INCREMENT](https://www.w3schools.com/sql/sql_autoincrement.asp)  
[INDEX](https://www.w3schools.com/sql/sql_create_index.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Állítsd be a Student táblában autoincrement -re a 'student\_id' mezőt.
2. Szúrj be három új felhasználót lekérdezéssel a Students táblába. A neveik sorban  
   legyenek: "Robert Downey Jr., "Jon Favreau", "Gwyneth Paltrow". A többi adat  
   lehet szabadon választott.

## UNION - Táblák egyesítése

kettő vagy több selectet kombinál össze

megkötések:

selecteknek azonos számú oszlopot és azonos típusú adatot kell tartalmazniuk, és a sorrendnek is egyeznie kell

séma:

**SELECT** `column\_name`(s) **FROM** `table1`

**UNION**

**SELECT** `column\_name`(s) **FROM** `table1`

az ismétlődő rekordokat hogy kihozza, az **UNION ALL parancsot** kelll használni!

**SELECT** `city` **FROM** `customers`

**UNION** **ALL**

**SELECT** `city` **FROM** `suppliers`

ugyanez WHERE feltétellel:

**SELECT** `city` **FROM** `customers` **WHERE** `country` = 'Hungary'

**UNION** **ALL**

**SELECT** `city` **FROM** `suppliers`;

Az UNION utasítás kettő vagy több SELECT utasítás eredménykészletét kombinálja.

### UNION utasítás jellemzői

* Minden UNION-ban lévő SELECT utasításnak azonos számú oszlopot kell tartalmaznia. Ha nem azonos az oszlopok száma akkor hibát kapunk.
* Az egyes SELECT utasítások által visszaadott oszlopoknak azonos adattípusúnak kell lenniük.
* Az oszlopoknak ugyanabban a sorrendben kell lenniük.

### UNION szintaxisa

Ahogy a többi utasítás esetén, úgy az UNION esetén is kötött a szintaxis (írásmód).

SELECT column\_name(s) FROM table1

UNION

SELECT column\_name(s) FROM table2;

### UNION ALL

Az UNION operátor alapértelmezés szerint csak különböző értékeket ad vissza. Az ismétlődő értékek engedélyezéséhez az UNION ALL parancsot használjuk:

SELECT column\_name(s) FROM table1

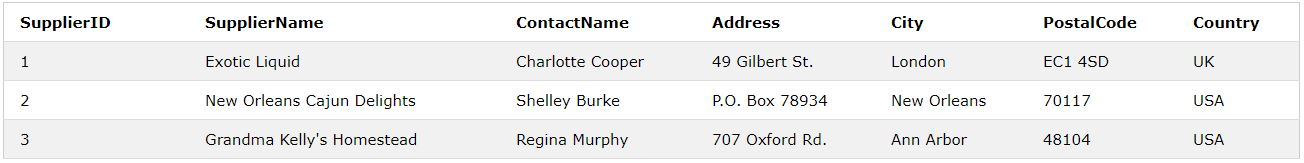
UNION ALL

SELECT column\_name(s) FROM table2;

### Példa

Az alábbi két adattáblában dolgozunk majd:





#### Különböző városok a Customers és Suppliers táblából

SELECT City FROM Customers

UNION

SELECT City FROM Suppliers

ORDER BY City;

Ha bizonyos városok mind a két táblában előfordulnak, esetleg többször is, a sima UNION miatt csak egyszer fognak szerepelni a lekérdezés eredményében.

#### Összes város

SELECT City FROM Customers

UNION ALL

SELECT City FROM Suppliers

ORDER BY City;

Itt annyiszor fog szerepelni egy város, ahány rekordban megtalálható.

#### UNION és WHERE - felételes eredmények uniója

Az alábbi lekérdezésben WHERE feltételt is alkalmazunk, egyébként megegyezik az előbbiekkel. Így megkapjuk az összes Német várost, de nem fog kétszer előfordulni ugyanaz a város. Ha az ismétlődő értékeket is látni szeretnénk, akkor az UNION ALL parancsot kell használnunk:

SELECT City, Country FROM Customers

WHERE Country='Germany'

UNION

SELECT City, Country FROM Suppliers

WHERE Country='Germany'

ORDER BY City;

#### Az összes Customer és Supplier

SELECT 'Customer' As Type, ContactName, City, Country

FROM Customers

UNION

SELECT 'Supplier', ContactName, City, Country

FROM Suppliers;

Érdekesség, hogy az első oszlopot az első lekérdezésben nevezzük el.

### Dokumentáció:

[SQL UNION](https://www.w3schools.com/sql/sql_union.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Kérd le a Student és a Teacher táblából az összes személy adatait UNION parancs használatával. Ügyelj arra, hogy az oszlopszám és az oszloponkénti adattípus megegyezzen.

## HAVING - Csoportfeltételek létrehozása

ha már csoportosítottam az oszlopokat group by-al, utána már alapba nem tudok szűrőt megadni; ezt lehet megoldani HAVING-el

**CREATE** **TABLE** `employees` (

`tsz` **VARCHAR**(50),

`nev` **VARCHAR**(100),

`Sz\_Helye` **VARCHAR**(50),

`Sz\_Ideje` **DATE**,

`fiz` **INT**

);

**INSERT** **INTO** employees **VALUES**

('T234578', 'Kiss István', 'Eger', '1968.12.11', 120000),

('T134578', 'Kovács István', 'Eger', '1972.01.11', 105000),

('T834578', 'Kiss Timót', 'Budapest', '1972.01.30', 150000),

('T934578', 'Vári Attila', 'Budapest', '1958.12.11', 210000),

('T034578', 'Kovács János', 'Szeged', '1967.12.11', 120000);

**SELECT** Sz\_Helye **AS** 'Születési Hely', **COUNT**(tsz) **AS** 'A dolg. száma'

**FROM** employees

**GROUP** **BY** Sz\_Helye;

most megvalósítjuk a plusz aggregálást:

**SELECT** Sz\_Helye **AS** 'Születési Hely',

**SUM**(fiz) **AS** 'Össz fizetés',

**AVG**(fiz) **AS** 'Átlag fizetés'

**FROM** employees

**GROUP** **BY** Sz\_Helye;

de innen még csak azokat akarom megkapni városokat, ahol az átlag fizetésre is van egy feltétel, pl meghalad egy értéket; GROP BY után jön a HAVING

**SELECT** Sz\_Helye **AS** 'Születési Hely',

**SUM**(fiz) **AS** 'Össz fizetés',

**AVG**(fiz) **AS** 'Átlag fizetés'

**FROM** employees

**GROUP** **BY** Sz\_Helye

**HAVING** **AVG**(fiz) >= 120000;

konklúzió: bizonyos bonyolultság felett már nehéz aggregáltan szűrni, ilyenkor hasznos a HAVING

**A HAVING utasítást akkor használjuk, ha a GROUP BY -ban megadott csoportokra szeretnénk feltételt megadni.**

### GROUP BY probléma

A lekérdezendő oszlopok adataira vonatkozóan mindenképpen alkalmaznunk kell valamilyen aggregáló függvényt (AVG, SUM, MIN, MAX), vagy ha ezt nem tesszük, akkor az oszlopnak szerepelnie kell a csoportosításban részt vevő oszlopok között, azaz a GROUP BY után. Erre azért van szükség, mert enélkül az egyes oszlopokban nem ugyanannyi cella lenne és ebben az esetben hibát fogunk kapni.

### HAVING - a megoldás

Az SQL nyelv lehetőséget biztosít arra is, hogy a csoportosított adatokra vonatkozóan feltételeket adhassunk meg. Ebben az esetben a feltételre nem a **WHERE** kulcsszót használjuk, hanem a **HAVING** kulcsszót.  
Szintaxis: HAVING <csoportfeltétel>

A <csoportfeltétel> paraméterben a hagyományos módon adhatunk meg feltételeket, azzal a különbséggel, hogy a feltételben szereplő oszlopneveknek tartalmazniuk kell valamilyen aggregáló függvényt, és ennek ugyancsak szerepelnie kell a SELECT után.

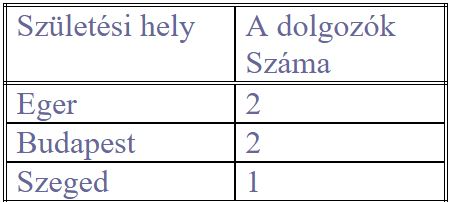
### Példa - lépésről lépésre

**A Dolgozó tábla**: 

**Az egy városban született dolgozók száma:**

SELECT Sz\_Helye AS 'Születési hely',

     COUNT(A\_dolgozó\_törzsszáma) AS 'A dolgozók száma' FROM Dolgozó GROUP BY Sz\_Helye;



**Az egyes városokban született dolgozók összes és átlagfizetése:**

SELECT Sz\_Helye AS 'Születési hely',

     SUM(Fizetés) AS 'A dolgozók összes fizetése',

     AVG(Fizetés) AS 'A dolgozók átlagfizetése' FROM Dolgozó

GROUP BY Sz\_Helye;

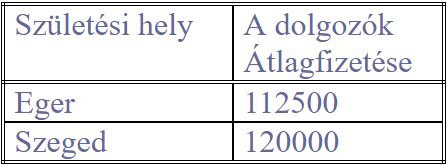


**Az városok, ahol az ott született dolgozók átlagfizetése maximum 120000 Ft:**

SELECT Sz\_Helye AS 'Születési hely',

     AVG (Fizetés) AS 'A dolgozók átlagfizetése'

FROM Dolgozó GROUP BY Sz\_Helye HAVING AVG(Fizetés) <= 120000;



### Konklúzió

Azt figyelhetjük meg a fenti példában, hogy bizonyos bonyolultság felett nehéz az aggregált értékek alapján szűrni. Ilyenkor használhatjuk fel a HAVING parancsot.

### Dokumentáció:

[SQL HAVING](https://www.w3schools.com/sql/sql_having.asp)

### Önálló feladatok

**Yellowroad adatbázis. Írd és futtasd!**

1. Készítsd el a fenti példában található táblát és futtasd le rajta a megadott parancsokat. A tábla és oszlopnevek legyenek angol nyelvűek.

## CREATE - Tábla létrehozása, elsődleges és idegen kulcsok megadása

hogyan kell a kulcsokkal dolgozni

1. id mező növekedjen magától és legyen egy elsődleges kulcs
2. egy táblában cask egy elsődleges kulcsoszlop lehet!

**ALTER** **TABLE** products **ADD** **PRIMARY** **KEY** (id);

**ALTER** **TABLE** products

**MODIFY** id **INT** **NOT** **NULL** **AUTO\_INCREMENT**; -> nem ment

kreálok egy új táblát és megadom, hogy az id ebből a táblából jöjjön; a product\_id nem önálló érték, hanem egy másik tábla id-jét tartalmazza; ha olyan értéket akarok felvenni, ami product\_id nincs a productsban, akkor nem fogja engedni -> óvom az adatbázisomat az ellentmondásokstól

**CREATE** **TABLE** `log` (

id **INT** **NOT** **NULL** **AUTO\_INCREMENT**,

message **VARCHAR**(500) **NOT** **NULL**,

product\_id **INT** **NOT** **NULL**,

**PRIMARY** **KEY** (id),

**FOREIGN** **KEY** (product\_id) **REFERENCES** products (id)

);

auto increment cask arra lehet, ami primary key?

**INSERT** **INTO** `log` (message, product\_id)

**VALUES** ('m', 4);

->engedte, mert volt a product táblában 4-es id

A kulcsok hatalma. A különböző kulcsok arra valók hogy egyedivé tegyük a mezőket  
vagy gyorsítsuk a keresést egy adatbázis táblában.

### **UNIQUE**

Ez az egyedi kulcs. Ha egy mezőre megadod hogy UNIQUE legyen, akkor kétszer  
ugyanaz az érték nem szerepelhet benne. Például a felhasználóknál ilyen szokott  
lenni az email cím:

CREATE TABLE Persons (

    ID int NOT NULL UNIQUE,

    ID email NOT NULL UNIQUE,

    LastName varchar(255) NOT NULL,

    FirstName varchar(255),

    Age int

);

Mi történik ha kétszer ugyanazzal az email címmel akarnak regisztrálni? A MySQL  
meg fogja tagadni a rekord mentését, és ha ügyes programozó vagy akkor elkapod a  
hibát és szépen értesíted a felhasználót hogy az email cím már foglalt, inkább  
jelentkezzen be ha teheti.

### **PRIMARY KEY**

**Az elsődleges kulcs egyedileg azonosít minden rekordot a táblában.** Ilyen módon  
hasonlóan működik mint az UNIQUE, csak kicsit még fontosabb annál. **Használatával  
lehet a leggyorsabban megkeresni egy adott rekordot,** mert az adatbázis kezelő  
egy külön helyen tartja nyilván az ilyen kulcsokat.

CREATE TABLE Persons (

    ID int NOT NULL,

    LastName varchar(255) NOT NULL,

    FirstName varchar(255),

    Age int,

    PRIMARY KEY (ID)

);

### **FOREIGN KEY**

**Ez az idegen kulcs. Tehát egy olyan mező aminek az értéke egy másik táblára mutat.**  
Emlékszel, ilyennel kötöttük össze a rendeléseket és az ügyfeleket vagy a tanárokat  
és a tanulókat.

CREATE TABLE Orders (

    OrderID int NOT NULL,

    OrderNumber int NOT NULL,

    PersonID int,

    PRIMARY KEY (OrderID),

    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)

);

A **REFERENCES** után meg kell adni hogy melyik tábla melyik mezőjére mutat.

### Dokumentáció:

[UNIQUE](https://www.w3schools.com/sql/sql_unique.asp)  
[PRIMARY KEY](https://www.w3schools.com/sql/sql_primarykey.asp)  
[FOREIGN KEY](https://www.w3schools.com/sql/sql_foreignkey.asp)

### Önálló feladatok

1. Módosítsd a Student táblát és állítsd be hogy a 'student\_email' mező UNIQUE  
   legyen.
2. Teszteld le a működést és próbáld többször ugyanazt az email címet felvenni  
   a Student táblába.