Az adatbázis-tervezés alapjai

Az adatbázis-tervezés alapfogalmai

Adat: nyers tények, amelyek önmagukban nem hordoznak jelentést.

Információ: az adatok feldolgozott értelmezett formája, ami döntéstámogatásra alkalmas.

Adatbázis: strukturált adathalmaz, amelyet hatékonyan lehet tárolni és lekérdezni.

Adatbázis-kezelő rendszer (DBMS - Database Management System): szoftver az adatok kezelésére (pl. MySQL, PostgreSQL, Oracle).

Tábla (reláció): relációs adatmodell esetén az adatbázis táblá(ka)t tartalmaz, amelyek sorokból és oszlopokból állnak.

Attribútum (mező): a tábla oszlopai, amelyek az adatok jellemzőit írják le.

Rekord: a tábla sorai, amelyek egy adott objektum adatait tartalmazzák.

Elsődleges kulcs: egyedi azonosító, olyan mező, amelynek az értékei minden rekord esetében különbözőek.

Idegen kulcs: kapcsolatot teremt két tábla között.

Összetett kulcs: több attribútumból álló kulcs.

Kapcsolatok (relációk) típusai:
- 1:1 (egy az egyhez)
- 1:N (egy a többhöz)
- N:M (több a többhöz)

Normalizálás: olyan folyamat, amelynek célja a redundancia csökkentése és az adatintegritás biztosítása.

Redundancia: az adatok felesleges, ismétlődő tárolását jelenti az adatbázisban. Ez gyakori probléma rosszul tervezett adatbázisokban. Olyan problémákat okozhat, mint például a tárhely pazarlás (feleslegesen sok adatot

tárolunk, ami nagy adatbázisoknál komoly erőforrás-pazarlás) és az inkonzisztencia (ha egy adatot frissítenünk kell, de nem minden helyen tesszük meg, eltérések keletkezhetnek). Ha például egy ügyfél adatait minden rendeléshez újra tároljuk (név, cím, telefonszám, stb.), ez redundáns, mert elég lenne egyszer tárolni az ügyféltáblában. Normalizálással csökkenthető a redundancia. A karbantartás bonyolultabbá válik, mivel több helyen kell módosítani ugyanazt az adatot. A lekérdezések lassabbá válhatnak, mivel nagyobb adattömeget kell kezelni.

Példa: egy Rendelések táblában minden rendeléshez tároljuk az ügyfél nevét, címét, telefonszámát. Itt feleslegesen ismétlődik az ügyfél neve, címe, telefonszáma.

RendelésID	ÜgyfélNév	Cím	Telefonszám	Termék
1	Kovács Péter	Budapest	123456789	Laptop
2	Kovács Péter	Budapest	123456789	Monitor

A redundancia csökkentésére a normalizálást használjuk. Ez azt jelenti, hogy az adatokat logikai egységekre bontjuk, és kapcsolatokat hozunk létre közöttük.

Ügyfelek tábla:

ÜgyfélID	ÜgyfélNév	Cím	Telefonszám
1	Kovács Péter	Budapest	123456789

Rendelések tábla:

RendelésID	ÜgyfélID	Termék
1	1	Laptop
2	1	Monitor

Itt az ügyfél adatai csak egyszer szerepelnek az Ügyfelek táblában, és az ÜgyfélID kapcsolja össze a rendelésekkel.

De a termékeket is érdemes külön táblában tárolni, és a Rendelések táblában csak az azonosítójukat szerepeltetni.

Összefoglalva tehát a redundancia felesleges adatismétlődést jelent, ami adatkezelési problémákhoz vezethet. A megoldás kulcsa a helyes adatbázis-tervezés és a normalizálás, ami biztosítja az adatok hatékony és következetes tárolását.

A számított adatok tárolása is redundanciának számít, ha ugyanaz az információ más formában már elérhető az adatbázisban.

Tegyük fel, van egy Személyek tábla:

Név	SzületésiÉv	Életkor
Kovács Péter	1990	35

Itt az Életkor felesleges, mert kiszámítható a születési év és az aktuális év alapján. Az SQL tartalmaz dátumfüggvényeket (lekérdezhető az aktuális év) és matematikai műveleteket (az aktuális évből ki tudjuk vonni a születési évet).

Itt is felmerül az inkonzisztencia veszélye, ha valaki frissíti a születési évet, de elfelejti módosítani az életkort, ellentmondás keletkezik. Minden évben frissíteni kellene az életkort minden egyes személy esetében, ami plusz munka. Ha születési év helyett születési dátummal dolgozunk, ez még nagyobb munka.

Adatintegritás: azt jelenti, hogy az adatok helyesek, pontosak és megbízhatóak maradnak az adatbázisban. Ez biztosítja, hogy az adatok érvényesek legyenek, és ne legyenek hibás vagy ellentmondásos értékek. Például minden ügyfélnek legyen egyedi azonosítója, egy rendelés csak létező ügyfélhez kapcsolódhat, egy termék ára nem lehet negatív szám, stb. Ez a kulcsok, idegen kulcsok, és ellenőrzési szabályok (constraints) alkalmazásával biztosítható.

Normálformák: 1NF, 2NF, 3NF, …

ER-diagram (Entity-Relationship): egyedek, attribútumok és kapcsolatok vizuális ábrázolása. Magyarul EK-diagramnak (Egyed-Kapcsolat) nevezzük.

Egy vizuális eszköz, amely az adatbázis struktúráját mutatja be egyedek, azok attribútumai és kapcsolataik ábrázolásával. Például egyed az ügyfél, a termék, a rendelés, kapcsolat például az ügyfél és a rendelés közötti kapcsolat.

Az anomáliák az adatbázisokban olyan adatkezelési problémák, amelyek akkor jelentkeznek, ha az adatbázis szerkezete nincs megfelelően normalizálva. Ezek leggyakrabban a redundancia miatt fordulnak elő, és három fő típusuk van.

Beszúrási anomália: ez akkor fordul elő, ha nem tudunk új adatot beszúrni anélkül, hogy más, felesleges adatot is meg kellene adnunk.

Példa: egy Diákok és Kurzusok táblában egyszerre kell tárolni a diák és az általa felvett kurzus adatait.

DiákNév	Kurzus
Kiss Anna	Matematika

Mit tegyünk, ha egy új diákot akarunk felvenni, de még nem vett fel kurzust? Nem tudjuk beszúrni, mert a kurzus mező nem lehet üres.

Módosítási anomália: ez akkor fordul elő, ha egy adat módosításakor több helyen is változtatni kell ugyanazt az információt.

Példa: egy táblában tároljuk az ügyfél nevét és címét minden rendelésnél.

ÜgyfélNév	Cím	Rendelés
Kovács Péter	Budapest	Laptop
Kovács Péter	Budapest	Monitor

Ha Péter elköltözik, minden egyes sorban frissíteni kell a címét. Ha valahol elfelejtjük, adatinkonzisztencia alakul ki.

Törlési anomália: ez akkor jelentkezik, ha egy adat törlése más, fontos adatok elvesztéséhez vezet.

Példa: egy táblában tároljuk az alkalmazottak nevét és az általuk vezetett projektek adatait.

Alkalmazott	Projekt
Nagy László	Weboldal
Nagy László	Mobil app

Ha minden projektet törlünk, amit Nagy László vezetett, elveszik az ő adata is az adatbázisból, pedig még alkalmazott.

A redundancia csökkentésére alkalmazható dekompozíció szerepe:

A dekompozíció azt jelenti, hogy egy nagy, összetett táblát kisebb, logikailag összefüggő táblákra bontunk. Ez segít:

- csökkenteni a redundanciát
- megelőzni az anomáliákat (beszúrási, módosítási, törlési)
- fenntartani az adatintegritást

Példa:

DiákID	DiákNév	Kurzus	Oktató
1	Kovács András	SQL	Kiss Bálint
2	Tóth Béla	Programozás	Horváth Csaba
3	Szabó Sándor	SQL	Kiss Bálint

Probléma:

- az oktató neve többször ismétlődik több kurzusnál → redundancia
- ha egy kurzust törlünk, a diák adata is elveszhet → törlési anomália

Dekompozíció után:

Diák tábla

DiákID	DiákNév
1	Kovács András
2	Tóth Béla

3	Szabó Sándor

Kurzus tábla

KurzusID	Kurzus	Oktató
1	SQL	Kiss Bálint
2	Programozás	Horváth Csaba

Jelentkezés tábla

DiákID	KurzusID
1	1
2	2
3	1

A dekompozíció célja:
- redundancia csökkentése: az oktató neve csak egyszer szerepel
- adatintegritás megőrzése: ha egy kurzust törlünk, a diák adatai megmaradnak
- rugalmas adatkezelés: egyszerűbb frissíteni és karbantartani az adatokat

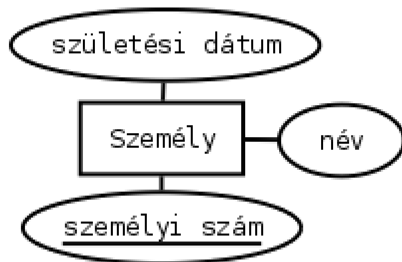
ER-diagram (EK-diagram):

A fő elemei és jelölésrendszere:
- Egyed - téglalap
- Attribútum - ellipszis
    - Elsődleges kulcs: aláhúzva
- Kapcsolat - rombusz

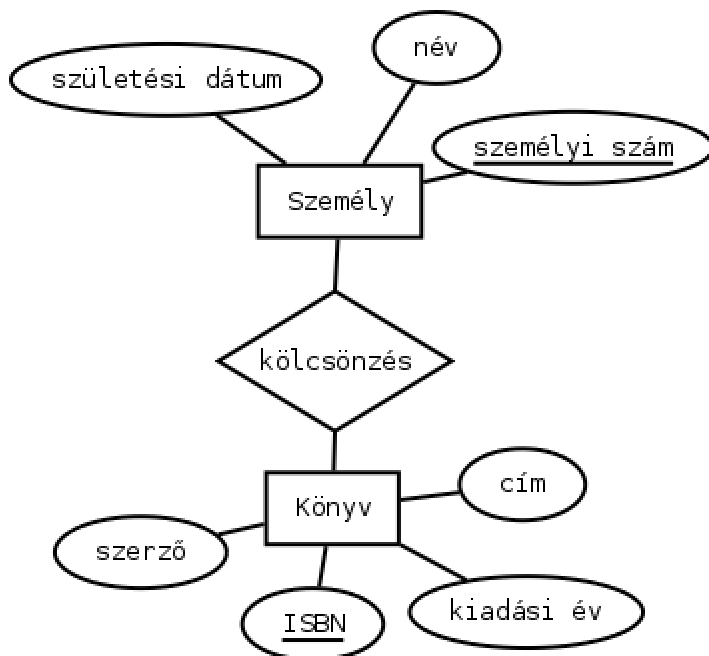
Az adatbázis logikai modelljének elkészítéséhez az alábbi szempontokat kell figyelembe venni:
- miről szeretnénk eltárolni adatokat?
- milyen adatokat szeretnénk tárolni?
- hogyan viszonyulnak egymáshoz a tárolandó adatok?

A fenti kérdésekre adjuk meg a választ az egyed-kapcsolat diagram jelölésrendszere segítségével.

Egyed (entitás): valódi vagy fiktív dolog, amely tulajdonságokkal, jellemzőkkel bír és önálló elemként tartjuk számon (pl. autó, személy, könyv, stb.) Az egyedeket téglalappal jelöljük és beleírjuk az egyed nevét.



Kapcsolat: két vagy több egyed között határoz meg relációt (pl. könyvtári adatbázisban ha van egy személy egyed és egy könyv egyed, akkor a köztük lévő kapcsolat a kölcsönzés). A kapcsolatokat rombusszal jelöljük, és beleírjuk a kapcsolat nevét.



Attribútum: egyedek és kapcsolatok tulajdonságát jelöli (pl. egy személy egyed esetében tulajdonság lehet a név, lakcím, születési dátum, adóazonosító, stb.). Minden egyed kell, hogy rendelkezzen legalább egy attribútummal. Kapcsolatok is rendelkezhetnek attribútummal (pl. a könyvtári adatbázisban a kölcsönzés kapcsolat rendelkezhet egy dátum attribútummal, amely a kölcsönzés dátumát jelöli). Az attribútumokat ellipszissel jelöljük és

beleírjuk a nevét. Azt a legszűkebb attribútumhalmazt, amely egyértelműen azonosítja az egyedet, kulcsnak nevezzük. A kulcsban szereplő attribútumok nevét aláhúzással jelöljük.

Összetett attribútum: olyan attribútum, amely maga is rendelkezik részattribútumokkal (pl. a lakcím attribútum lehet összetett attribútum, ha rendelkezik irányítószám, utca és házszám attribútummal). Az összetett attribútumot és a részattribútumokat is ellipszissel jelöljük. A részattribútumokat az összetett attribútumhoz kötjük.

Az EK-diagram relációs modellre való átalakításának szabályai:

Az EK-diagram egy logikai ábrázolás, de az adatokat végül relációs adatbázisban (táblákban) tároljuk. Az átalakítás során az egyedekből táblák, az attribútumokból oszlopok, a kapcsolatokból pedig kulcsok lesznek.

Minden egyes egyed egy tábla lesz.
Az egyedekhez tartozó attribútumok a tábla oszlopai lesznek.
Az egyed azonosítója az elsődleges kulcs lesz.
A kapcsolatokból kulcsok, idegen kulcsok vagy kapcsolótáblák lesznek.

Adatbázisok létrehozása

XAMPP: egy szabad és nyílt forrású platformfüggetlen webszerver-szoftvercsomag, amelynek legfőbb alkotóelemei az Apache webszerver, a MariaDB (korábban a MySQL) adatbázis-kezelő, valamint a PHP és a Perl programozási nyelvek értelmezői. Ez a szoftvercsomag egy integrált rendszert alkot, amely webes alkalmazások készítését, tesztelését és futtatását célozza, és ehhez egy csomagban minden szükséges összetevőt

tartalmaz. A rendszer egyik nagy előnye az összehangolt elemek könnyű telepíthetősége.

A XAMPP egy rövidítés, betűi a következő kifejezésekből származnak:

- X – eredeti olvasatban az angol cross-platform szót helyettesíti, amely a platformfüggetlenséget jelenti
- Apache webszerver
- MariaDB adatbázis-kezelő (korábban MySQL)
- PHP szerveroldali szkriptnyelv
- Perl általános célú szkriptnyelv

A MariaDB egy többfelhasználós SQL-alapú relációsadatbázis-kezelő szerver.

Letöltés: https://www.apachefriends.org/hu/index.html

A XAMPP egyetlen állományba van csomagolva, telepítéséhez mindössze ezt a fájlt kell letölteni és futtatni. A telepítés elvégzi az alapbeállításokat, azokon csak nagyon keveset vagy éppen semmit nem kell változtatni, ezután a rendszer készen áll a webszerver és a mintaalkalmazások futtatására. A XAMPP-csomagot rendszeresen frissítik, így az mindig az Apache, MariaDB, PHP és Perl legújabb változatát tartalmazza, valamint más kiegészítőket is, mint pl. a phpMyAdmin.

Hivatalosan a XAMPP tervezői az eszközt egy fejlesztőrendszernek szánták, amellyel a web-tervezők és programozók internetes kapcsolat nélkül fejleszthetik és tesztelhetik alkalmazásaikat. Ennek érdekében több fontos biztonsági funkció alapértelmezésben ki van kapcsolva a csomagban, ennek ellenére a XAMPP szoftvert valódi webes szolgáltatóként is használják.

A XAMPP többféle adatbázis-kezelő használatát is támogatja, ilyenek pl. a MySQL, az SQLite, stb.

A XAMPP telepítése után a helyi gép (a localhost) hálózati gépként is hozzáférhetővé válik.

A phpMyAdmin egy nyílt forrású eszköz, amit PHP-ban írtak a MySQL menedzselésére (adatbázisok készítése / eldobása, táblák készítése / eldobása / módosítása, mezők hozzáadása / módosítása / törlése, SQL parancsok futtatása, stb.) az interneten keresztül.

Az adatbázis kezeléséhez az XAMPP Control Panelen az Apache webszervert és a MySQL adatbázis szervert kell elindítani. A felület a localhost/phpmyadmin URL segítségével érhető el böngészőben.

A következő témakörben az SQL parancsok közül a DDL parancsokkal foglalkozunk (Data Definition Language - Adatdefiníciós nyelv).

CREATE utasítás adatbázisok és táblák létrehozására

A CREATE utasítás az SQL-ben arra szolgál, hogy új adatbázist, táblát, indexet vagy egyéb adatbázis-objektumot hozzunk létre. Itt most az adatbázis és táblák létrehozására fogunk koncentrálni.

Az adatbázis létrehozásához az alábbi SQL parancsot használhatjuk: CREATE DATABASE adatbazis_neve; pl. CREATE DATABASE dolgozok;

A táblák azok az objektumok, amelyek az adatokat tárolják az adatbázisban. A tábla létrehozásához az alábbi szintaxist használjuk:

```
CREATE TABLE tabla_neve (
oszlop1 adattipus [opcionalis_zaradek],
oszlop2 adattipus [opcionalis_zaradek],
...
oszlopN adattipus [opcionalis_zaradek]
);
```

Például:

```
CREATE TABLE dolgozok (
id INT PRIMARY KEY,
nev VARCHAR(100) NOT NULL,
szuletesi_datum DATE,
email VARCHAR(255) UNIQUE
);
```

A tábla a következő oszlopokat tartalmazza:

- id: az oszlop típusa INT (egész szám), és PRIMARY KEY, ami azt jelenti, hogy az értékének egyedinek kell lennie, és kulcsként használjuk az egyes rekordok azonosítására.
- nev: a dolgozó neve, amely VARCHAR(100) típusú (szöveg és maximum 100 karakter lehet), és NOT NULL, tehát nem lehet üres.

- szuletesi_datum: a dolgozó születési dátuma DATE típusú, és nem kötelező kitölteni.
- email: a dolgozó email címe, amely VARCHAR(255) típusú, és UNIQUE, ami azt jelenti, hogy minden email címnek egyedinek kell lennie a táblán belül, de nem kulcsként használjuk a mezőt.

A legfontosabb adattípusok a MariaDB-ben (mivel XAMPP-ot és MariaDB-t használunk, az adattípusok alapvetően ugyanazok, mint amiket MySQL-ben is használunk, mivel a MariaDB MySQL-kompatibilis adatbázis-kezelő)

Szám típusok:

- INT: egész számok tárolására szolgál. Ha nem adunk meg más paramétert, akkor alapértelmezésben 4 bájton tárolódik, és -2 147 483 648 és 2 147 483 647 közötti értékeket tud tárolni.
- TINYINT: kisebb egész számok tárolására szolgál, 1 byte-on tárolja az adatokat, és -128 és 127 közötti értékeket tud tárolni.
- SMALLINT: kisebb egész számok tárolására szolgál, 2 byte-on tárolódik. Az értéktartomány -32 768 és 32 767 közötti.
- MEDIUMINT: közepes méretű egész számok tárolására szolgál, 3 byte-on tárolódik. Az értéktartomány -8 388 608 és 8 388 607 között van.
- BIGINT: nagy egész számok tárolására szolgál, 8 byte-on tárolódik, és -9 223 372 036 854 775 808 és 9 223 372 036 854 775 807 közötti számokat tud tárolni.
- DECIMAL (vagy NUMERIC): pontos számok tárolására szolgál, amelyeket a pénzügyi alkalmazásokban használnak. Az adattípus egy fix pontosságú számot tárol. Pl. ar DECIMAL(10, 2) – ez azt jelenti, hogy a szám 10 számjegyű, ebből 2 a tizedesvessző után van.
- FLOAT: lebegőpontos számok tárolása, kisebb pontossággal.
- DOUBLE: nagyobb pontosságú lebegőpontos számok.

Szöveg típusok:

- VARCHAR(n): változó hosszúságú karakterláncok, ahol az n az oszlop maximális hosszát határozza meg. Az adatbázis dinamikusan tárolja a karaktereket, tehát csak a ténylegesen tárolt karakterek hosszát használja.
- CHAR(n): fix hosszúságú karakterláncok, ahol n a karakterek pontos száma. Ha a tárolt szöveg rövidebb, mint az n értéke, a rendszer kitölti a különbséget szóközökkel.

- TEXT: nagyobb szövegek tárolására szolgál, mint a VARCHAR, 65 535 karakterig terjedhet.
- LONGTEXT: még nagyobb szövegek tárolására szolgál, akár 4 GB-nyi szöveg is elhelyezhető benne.

Dátum és idő típusok:

- DATE: dátumok tárolására szolgál, csak az évet, a hónapot és a napot tartalmazza. Formátuma: YYYY-MM-DD.
- DATETIME: dátum és idő tárolása szolgál, formátuma YYYY-MM-DD HH:MM:SS.
- TIMESTAMP: az UTC időzónában (az a hivatkozási időzóna, amelyhez a Föld többi időzónáját viszonyítjuk) tárolja a dátumot és időt függetlenül attól, hogy milyen időzónában fut az adatbázis. Lekérdezéskor a kliens időzónája szerint alakítja át az időt. Például egy magyarországi szerverről beszúrunk egy időbélyeget. Ha nyáron (UTC+2) 14:00-kor futtatjuk, a TIMESTAMP típusú oszlopban 12:00 (UTC) lesz tárolva. Ha lekérdezzük a saját időzónánkban: magyar idő szerint 14:00-ként jelenik meg, mert az adatbázis visszaalakítja. Az alapértelmezett formátuma szintén YYYY-MM-DD HH:MM:SS.
- TIME: csak az idő (óra, perc, másodperc) tárolása, formátuma HH:MM:SS.
- YEAR: csak az év tárolása, formátuma YYYY.

Logikai típusok:

- BOOLEAN: igaz / hamis értékek tárolására szolgál. A MariaDB-ben a BOOLEAN alapértelmezés szerint 1 vagy 0 értékkel tárolja az adatokat (1 - igaz, 0 - hamis).

Bináris típusok:

- BLOB (Binary Large Object): bináris adatokat tárol, mint például képek, videók, hangfájlok, stb. A BLOB típusnak különböző méretű változatai vannak: TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB. A beszúrást elvégző SQL utasításban egy megfelelő függvényt kell használni, amelynek paramétere a fájl elérési útvonala és neve. Lekérdezéskor pedig a bináris adatot általában egy programkód dolgozza fel. Hátránya, hogy nagy mennyiségű BLOB adat nagyon megnöveli az adatbázis méretét és ez lassítja a lekérdezéseket is. Gyakran az a jobb megoldás, ha a fájlokat a fájlrendszerben tároljuk és csak az elérési utakat mentjük el az adatbázisban.

Előre definiált értékeket tároló típusok:

- ENUM('érték1', 'érték2', …): olyan mező, amely csak az előre meghatározott értékek egyikét veheti fel (pl. férfi, nő).
- SET('érték1', 'érték2', …): több előre definiált értéket is tartalmazhat egy mezőben (pl. egy felhasználó több hobbija).

Ezek a leggyakrabban használt adattípusok, ezeken kívül vannak még speciális adattípusok is.

<u>A táblák létrehozásánál alkalmazható mezőszintű és táblaszintű záradékok</u>

Az SQL-ben a záradékok (constraints) meghatározzák az adatbázis integritását, vagyis biztosítják, hogy az adatok megfeleljenek bizonyos feltételeknek. A záradékokat mezőszinten (az érintett oszlopon belül) vagy táblaszinten (az egész táblára vonatkozóan) adhatjuk meg.

Mezőszintű záradékok: ezeket egy konkrét oszlop definiálásánál adjuk meg.

NULL / NOT NULL: meghatározza, hogy az adott mező lehet-e NULL értékű (nem kötelező kitölteni) vagy nem lehet NULL értékű (kötelező kitölteni).

```
CREATE TABLE alkalmazott (
id INT NOT NULL,
nev VARCHAR(100) NOT NULL,
email VARCHAR(100) NULL
);
```

Az id és nev mezőket mindig ki kell tölteni, de az e-mail mezőt nem kötelező.

UNIQUE: biztosítja, hogy egy oszlop értékei egyediek legyenek.

```
CREATE TABLE felhasznalok (
id int PRIMARY KEY,
felhasznalonev VARCHAR(100) UNIQUE
);
```

A felhasznalonev oszlopban nem szerepelhet kétszer ugyanaz az érték. Az id oszlopban sem, de azt elsődleges kulcsként használjuk, és annak ez a velejárója. A felhasznalonev oszlopot nem használjuk elsődleges kulcsként, de szeretnénk, ha minden érték egyedi lenne.

PRIMARY KEY: minden érték egyedi, így az elsődleges kulcsként használt mező alkalmas a rekordok azonosítására. Nem lehet NULL, de ezt nem kell külön megadni.

```
CREATE TABLE termekek (
id INT PRIMARY KEY,
nev VARCHAR(100) NOT NULL
);
```

A termék id mindig egyedi és nem lehet NULL.

DEFAULT: alapértelmezett értéket rendel egy mezőhöz, ha nem adunk meg értéket.

```
CREATE TABLE megrendelesek (
id INT PRIMARY KEY,
statusz VARCHAR(100) DEFAULT 'folyamatban'
);
```

Ha nincs megadva a statusz értéke akkor, automatikusan 'folyamatban' lesz.

CHECK: biztosítja, hogy egy oszlopba csak bizonyos feltételnek megfelelő értékek kerülhessenek.

```
CREATE TABLE dolgozok (
id INT PRIMARY KEY,
nev VARCHAR(100),
eletkor INT CHECK (eletkor >= 18)
);
```

18 alatti életkort nem lehet rögzíteni az a táblában.

AUTO_INCREMENT: automatikus sorszámozás.

```
CREATE TABLE dolgozok (
id INT PRIMARY KEY AUTO_INCREMENT,
nev VARCHAR(100) NOT NULL
);
```

Beszúráskor az id-t nem kell megadni, csak a nevet, az első id automatikusan 1 lesz, majd minden egyes beszúrásnál automatikusan növekszik.

Táblaszintű záradékok: ezeket a tábla létrehozásánál, az oszlopok definiálása után adjuk meg.

PRIMARY KEY (több oszlop esetén):

```
CREATE TABLE megrendelesek (
termekID INT,
felhasznaloID INT,
datum DATE,
PRIMARY KEY (termekID, felhasznaloID)
);
```

A termekID és a felhasznaloID együtt alkotják az egyedi azonosítót.

FOREIGN KEY (REFERENCES): az egyik tábla mezőjét egy másik tábla egyik mezőjéhez köti.

```
CREATE TABLE ugyfelek (
ugyfelID INT PRIMARY KEY,
ugyfelNev VARCHAR(100)
);

CREATE TABLE bevetelek (
bevetelID INT PRIMARY KEY,
ugyfelID INT,
bevetel INT,
FOREIGN KEY (ugyfelID) REFERENCES ugyfelek(ugyfelID)
);
```

Az ugyfelID oszlop csak olyan értéket vehet fel, amely az ugyfelek tábla ugyfelID oszlopában szerepel. Az ugyfelID ebben a táblában idegen kulcs, az ugyfelek táblában viszont elsődleges kulcs.

UNIQUE (több oszlop esetén):

```
CREATE TABLE berlesek (
berlesID INT PRIMARY KEY,
autoID INT NOT NULL,
ugyfelID INT NOT NULL,
datum DATE NOT NULL,
UNIQUE (autoID, datum)
);
```

A kombinációnak egyedinek kell lennie. Tehát egy adott autó egy adott napon csak egyszer szerepelhet a bérlések között.

ON DELETE / ON UPDATE: műveletek kapcsolt táblákon. Meghatározza, mi történjen, ha egy rekordot törlünk vagy módosítunk a másik táblában.

```
CREATE TABLE osztalyok (
osztalyID INT PRIMARY KEY,
osztalyNev VARCHAR(100)
);

CREATE TABLE diakok (
diakID INT PRIMARY KEY,
diakNev VARCHAR(50),
osztalyID INT,
FOREIGN KEY (osztalyID) REFERENCES osztalyok(osztalyID)
);
```

Ha nem használjuk ezeket a záradékokat, akkor nem tudunk törölni olyan osztályt, amelyikhez tartozik diák a másik táblában. Valamint nem tudjuk módosítani annak az osztálynak az id-ját, amelyikhez tartozik diák a másik táblában. Az osztály nevét tudjuk módosítani.

ON DELETE: a különböző lehetőségek meghatározzák, mi történjen azokkal a rekordokkal, amelyek kapcsolódnak a törölt rekordhoz.

- CASCADE: ha egy rekordot törlünk az elsődleges táblából (ebben az esetben az osztalyok táblából), akkor az összes olyan rekord, amely kapcsolódik hozzá a másodlagos táblában (itt a diakok), szintén törlődik. Ez biztosítja, hogy ne maradjanak árva rekordok a másodlagos táblában.

```
FOREIGN KEY (osztalyID) REFERENCES osztalyok(osztalyID)
ON DELETE CASCADE
```

ON UPDATE: azt szabályozza, mi történik, ha az elsődleges táblában lévő rekord egy oszlopának értéke módosul.

- CASCADE: ha egy rekord frissül az elsődleges táblában (például az osztalyok táblában módosítjuk az osztalyID értékét), akkor a

másodlagos táblában lévő összes kapcsolódó rekord is frissül, és az új értéket tükrözi. Ez segít abban, hogy a kapcsolatok mindig naprakészen tükrözzék az elsődleges tábla változásait.

```
FOREIGN KEY (osztalyID) REFERENCES osztalyok(osztalyID)
ON UPDATE CASCADE
```

Összegzés: CASCADE esetén az események mindkét oldalon automatikusan végrehajtódnak (törlés vagy frissítés).

További lehetőségek ON DELETE és ON UPDATE esetén:
- SET NULL: az idegen kulcs mezők értékei NULL-ra állítódnak a kapcsolódó táblában.
- NO ACTION / RESTRICT: nem engedi törölni vagy frissíteni a rekordot, ha ahhoz kapcsolódó rekordok vannak az idegen kulcsot tartalmazó táblában.
- SET DEFAULT: ha a rekordot töröljük vagy frissítjük, a kapcsolódó rekordok mezői alapértelmezett értékre lesznek állítva.

A választott lehetőség attól függ, hogy milyen adatbázis-integritási szabályokat szeretnénk alkalmazni, és hogyan akarjuk kezelni a kapcsolódó rekordok törlését vagy frissítését.

Az indexek szerepe és létrehozása

Az indexek kulcsszerepet játszanak az adatbázisok teljesítményében, különösen akkor, ha nagy adatállománnyal dolgozunk, és gyors kereséseket vagy lekérdezéseket kell végezni. Segítenek abban, hogy az adatbázis gyorsabban találjon meg egyes rekordokat a táblákban, anélkül hogy végig kellene néznie az összes sort.

Az indexek szerepe:
- teljesítmény javítása: az indexek gyorsítják a kereséseket és a lekérdezéseket, mivel az adatbázis nem keres végig minden rekordot, hanem az indexek alapján gyorsan elérheti a keresett adatokat
- korlátozott helyigény: az indexek nem tárolják az adatokat magukat, hanem csak mutatókat, amelyek a táblák soraiban található rekordok helyére mutatnak
- az indexek rendezett adatokat biztosítanak, így gyorsabbá válik a rendezési műveletek végrehajtása is

Az indexek létrehozása

Az indexek létrehozásához a CREATE INDEX utasítást használhatjuk.

Szintaxis:

```
CREATE INDEX index_neve ON tabla_neve (oszlop1, oszlop2, …);
```

Ez létrehoz egy indexet a megadott oszlopokon, amely gyorsítja a keresést ezekben az oszlopokban. Ha több oszlopot adunk meg, akkor az index az oszlopok kombinációján alapul, így többfeltételes kereséseknél is segít.

Például:

```
CREATE TABLE ugyfelek (
id INT PRIMARY KEY,
vezetekNev VARCHAR(50),
keresztNev VARCHAR(50),
email VARCHAR(100)
);
```

```
CREATE INDEX ugyfel_nev ON ugyfelek (vezetekNev, keresztNev);
```

Ez az index az ugyfelek táblában lévő vezetekNev és keresztNev oszlopokon alapul, így gyorsabban találhatók meg azok az ügyfelek, akikre a neveik alapján keresünk.

```
CREATE UNIQUE INDEX email_unique ON ugyfelek (email);
```

Ez az index az ugyfelek táblában lévő e-mail oszlopon alapul és biztosítja, hogy minden érték egyedi legyen.

Az indexek típusai:
- elsődleges index (Primary Index): az elsődleges kulcson alapuló index. Egy táblában csak egy elsődleges index lehet, és az automatikusan egyediséget biztosít a rekordok számára.
- egyedi index (Unique Index): az index biztosítja, hogy az adott oszlopban szereplő értékek egyediek legyenek.
- normál index: az alapértelmezett index, amely nem biztosít egyediséget. Csak a keresések gyorsítására szolgál.

Az indexek gyorsítják a keresést, de lassíthatják az adatokat módosító műveleteket (INSERT, UPDATE, DELETE), mivel az indexek is frissítésre kerülnek ezek során. Ezért fontos mérlegelni, hogy hol és mikor használjunk indexeket, különösen nagy adatbázisok esetén.

Az indexek hatékonyan működnek olyan adattípusokkal, mint az egész számok, a dátumok, és a szövegek, de nem mindig ideálisak például nagy szövegmezők vagy bináris adatok indexelésére, mivel az indexek sok helyet igényelhetnek.

Hogyan választjuk meg, hogy mikor érdemes indexet létrehozni? Ha gyakran hajtunk végre kereséseket vagy lekérdezéseket egy adott oszlopon.
Az indexek és a záradékok (például PRIMARY KEY vagy UNIQUE) szorosan együttműködnek. Ha például egy oszlopot PRIMARY KEY-ként definiálunk, akkor az automatikusan egy indexet hoz létre az oszlophoz.

ALTER utasítás az adatbázisok, a táblák és az indexek módosítására

Az ALTER utasítás az adatbázisok, a táblák és az indexek módosítására szolgál. Az ALTER utasítás rendkívül hasznos, mivel lehetővé teszi a meglévő adatbázisok és táblák szerkezetének változtatását anélkül, hogy teljesen új adatbázist vagy táblát kellene létrehozni.

Adatbázis módosítása: ALTER DATABASE

Az ALTER DATABASE utasítással adatbázis szintű változtatásokat végezhetünk. Például az adatbázis karakterkészletét vagy kollációját módosíthatjuk. Ezzel szabályozhatjuk, hogyan történik az adatok tárolása és rendezése, valamint az összehasonlítási szabályok alkalmazása.

Szintaxis: ALTER DATABASE adatbazis_neve CHARACTER SET karakterkeszlet COLLATE kollacio;

Például: ALTER DATABASE dolgozok CHARACTER SET utf8 COLLATE utf8_hungarian_ci;

Az adatbázis karakterkészlete és kollációja hatással van az egész adatbázisra, így az új karakterkészlet az új táblákra, oszlopokra vonatkozik, de már meglévő adatokra nem feltétlenül.

Tábla módosítása: ALTER TABLE

Az ALTER TABLE utasítással lehetőség van a tábla szerkezetének módosítására. Ezen keresztül oszlopokat adhatunk hozzá, távolíthatunk el, vagy módosíthatjuk azok típusát, nevét, illetve indexeket, kulcsokat is létrehozhatunk vagy törölhetünk.

Oszlop hozzáadása szintaxis: `ALTER TABLE tabla_neve ADD oszlop_neve oszlop_tipus;`

Például:

```
CREATE TABLE ugyfelek (
id INT PRIMARY KEY,
vezetekNev VARCHAR(50),
keresztNev VARCHAR(50),
email VARCHAR(100)
);

ALTER TABLE ugyfelek ADD cim VARCHAR(100);
```

Oszlop eltávolítása szintaxis: `ALTER TABLE tabla_neve DROP COLUMN oszlop_neve;`

Például: `ALTER TABLE ugyfelek DROP COLUMN cim;`

Oszlop típusának módosítása szintaxis: `ALTER TABLE tabla_neve MODIFY COLUMN oszlop_neve uj_tipus;`

Például: `ALTER TABLE ugyfelek MODIFY COLUMN email VARCHAR(50);`

Oszlop átnevezése szintaxis: `ALTER TABLE tabla_neve CHANGE oszlop_regi_neve oszlop_uj_neve uj_tipus;`

Például: `ALTER TABLE ugyfelek CHANGE email email_cim VARCHAR(100);`

Tábla átnevezése szintaxis: `ALTER TABLE tabla_regi_neve RENAME TO tabla_uj_neve;`

Például: `ALTER TABLE ugyfelek RENAME TO partnerek;`

DROP utasítás az adatbázisok, a táblák és az indexek törlésére

A DROP utasítást akkor használjuk, amikor egy adatbázist, táblát vagy indexet véglegesen el akarunk távolítani az adatbázisból. Fontos, hogy a DROP művelet visszavonhatatlan, tehát a törölt objektumok (adatbázisok, táblák, indexek) és azok adatai véglegesen elvesznek. Természetesen, ha van biztonsági mentés, akkor vissza tudjuk állítani az adatokat.

Index törlése: DROP INDEX

A DROP INDEX utasítást használjuk indexek törlésére. Az index törlésével az adott oszlopok keresési teljesítménye csökkenhet, mivel az indexek segítenek a gyorsabb lekérdezések végrehajtásában.

Szintaxis: `DROP INDEX index_neve ON tabla_neve`

Példa: `DROP INDEX ugyfel_nev ON ugyfelek;`

Az indexek törlése nem törli a táblát vagy az adatokat, csupán azt a segédstruktúrát távolítja el, amely a gyors keresést segítette.

Tábla törlése: DROP TABLE

A DROP TABLE parancs segítségével egy adott táblát törölhetünk az adatbázisból, beleértve annak összes rekordját is.

Szintaxis: `DROP TABLE tabla_neve;`

Például: `DROP TABLE ugyfelek;`

Adatbázis törlése: DROP DATABASE

Ha egy adatbázist szeretnénk törölni, használjuk a DROP DATABASE parancsot. A parancs törli az adatbázist és minden olyan objektumot, ami abban található, például táblák, indexek, stb.

Szintaxis: `DROP DATABASE adatbazis_neve;`

Például: `DROP DATABASE dolgozok;`

Minden olyan esetben, amikor egy törlés adatvesztéssel jár, a rendszer megerősítést kér a törlés előtt.

Adatok kezelése

A témakör az adatbázis-kezelésnél használt legfontosabb DML-parancsokkal (Data Manipulation Language - Adatmanipulációs nyelv) foglalkozik, ezek teszik lehetővé adatok beszúrását, módosítását és törlését egy relációs adatbázisban.

<u>Adatok beszúrása (INSERT)</u>

Az INSERT utasítással új rekordokat adhatunk az adatbázis egy táblájához.

Szintaxis: `INSERT INTO tabla_neve (oszlop1, oszlop2, …) VALUES (ertek1, ertek2, …);`

Példa:

```
CREATE TABLE dolgozok(
id INT PRIMARY KEY AUTO_INCREMENT,
nev VARCHAR(100),
pozicio VARCHAR(100),
fizetes INT DEFAULT 400000
);
```

```
INSERT INTO dolgozok (id, nev, pozicio, fizetes) VALUES (1, 'Kis Péter',
'rendszergazda', 600000);
```

Automatikus sorszámozás esetén az adott mező kihagyható.

```
INSERT INTO dolgozok (nev, pozicio, fizetes) VALUES ('Nagy Zoltán', 'fejlesztő',
700000);
```

Ha minden oszlopba szúrunk be értéket, a mezőnevek kihagyhatók.

```
INSERT INTO dolgozok VALUES (3, 'Kovács Anna', 'könyvelő', 500000);
```

Ha valamelyik oszlop DEFAULT értékkel rendelkezik és nem adjuk meg az adott értéket, akkor automatikusan az alapértelmezett értéket kapja.

INSERT INTO dolgozok (id, nev, pozicio) VALUES (4, 'Tóth József', 'karbantartó');

Adatok módosítása (UPDATE)

Az UPDATE utasítással a meglévő rekordokat módosíthatjuk. A módosítás több rekordot is érinthet.

Szintaxis: UPDATE tabla_neve SET oszlop1 = ertek1, oszlop2 = ertek2, … WHERE feltetel;

Példa: UPDATE dolgozok SET fizetes = 650000 WHERE nev = 'Kis Péter';

Ha nincs WHERE feltétel, az összes rekord módosul.

UPDATE dolgozok SET fizetes = 650000;

Ha egyik rekord sem felel meg a feltételnek, nem történik módosítás.

UPDATE dolgozok SET fizetes = 650000 WHERE nev = 'Szabó Tamás';

Adatok törlése (DELETE)

A DELETE utasítással eltávolíthatunk rekordokat a táblából. A törlés több rekordot is érinthet.

Szintaxis: DELETE FROM tabla_neve WHERE feltétel;

Példa: DELETE FROM dolgozok WHERE nev = 'Kovács Anna';

A törlés előtt megerősítést kér a rendszer.

Ha nincs WHERE feltétel, az összes rekord törlődik.

DELETE FROM dolgozok;

Ha egyik rekord sem felel meg a feltételnek, nem történik törlés.

DELETE FROM dolgozok WHERE nev = 'Szabó Tamás';

Ha az összes rekordot töröljük a DELETE FROM utasítással, az AUTO_INCREMENT érték nem áll vissza egyre. A következő beszúrásnál az előző legnagyobb azonosító után folytatja a számozást.

Lekérdezések

A témakör az adatok lekérdezésére szolgáló SELECT parancs használatával foglalkozik. A SELECT parancs a DQL (Data Query Language - Adatlekérdező nyelv) kategóriába tartozik, mivel az adatokat kérdezi le az adatbázisból, de nem módosítja azokat.

A SELECT utasítás az SQL egyik legfontosabb parancsa, amely adatok lekérdezésére szolgál az adatbázisból. Segítségével meghatározhatjuk, hogy mely mezőket szeretnénk lekérdezni és milyen feltételek alapján.

Szintaxis:

```
SELECT oszlop1, oszlop2, …
FROM tabla_neve;
```

Ha minden oszlopot le szeretnénk kérdezni:

```
SELECT *
FROM tabla_neve;
```

Példa:

```
CREATE TABLE konyvek(
azonosito INT PRIMARY KEY AUTO_INCREMENT,
cim VARCHAR(100),
szerzo VARCHAR(100),
kiado VARCHAR(100),
kiadasi_ev YEAR,
mufaj VARCHAR(50),
oldalszam INT
);

INSERT INTO konyvek (cim, szerzo, kiado, kiadasi_ev, mufaj, oldalszam) VALUES
('A Gyűrűk Ura', 'J.R.R. Tolkien', 'Európa', 1954, 'Fantasy', 1216),
('Harry Potter és a Bölcsek Köve', 'J.K. Rowling', 'Animus', 1997, 'Fantasy', 320),
('1984', 'George Orwell', 'Európa', 1949, 'Disztópia', 328),
('Pride and Prejudice', 'Jane Austen', 'Penguin', NULL, 'Romantikus', 279),
```

```
('A Mester és Margarita', 'Mihail Bulgakov', 'Európa', 1967, 'Szépirodalom', 432),
('Fahrenheit 451', 'Ray Bradbury', 'Gabo', 1953, 'Sci-fi', 256),
('A Pál utcai fiúk', 'Molnár Ferenc', 'Móra', 1906, 'Ifjúsági', 180),
('A kis herceg', 'Antoine de Saint-Exupéry', 'Kossuth', 1943, 'Meseregény', 96),
('A Da Vinci-kód', 'Dan Brown', 'Gabo', 2003, 'Thriller', 590),
('A szolgálólány meséje', 'Margaret Atwood', 'Jelenkor', 1985, 'Disztópia', 400);
```

```
SELECT szerzo, cim
FROM konyvek;
```

Ez a lekérdezés a konyvek táblából csak a szerzo és cim oszlopokat adja vissza.

```
SELECT *
FROM konyvek;
```

Ez a lekérdezés a konyvek táblából az összes oszlopot visszadja.

<u>A WHERE záradék és az operátorok használata</u>

A WHERE záradék segítségével szűrhetjük az adatsorokat, azaz meghatározhatjuk, hogy milyen feltételeknek kell megfelelniük a visszaadott adatoknak.

Szintaxis:

```
SELECT oszlop1, oszlop2, …
FROM tabla_neve
WHERE feltetel;
```

Gyakori operátorok a WHERE záradékban:
- = (egyenlő)
- != vagy <> (nem egyenlő)
- > (nagyobb)
- < (kisebb)
- >= (nagyobb vagy egyenlő)
- <= (kisebb vagy egyenlő)
- BETWEEN ertek1 AND ertek2 (két érték közötti tartomány, beleértve a megadott értékeket is)
- IN (ertek1, ertek2, …) (több lehetséges érték megadása)
- LIKE minta (mintázat keresés helyettesítő karakterekkel)

- IS NULL (NULL érték keresése)
- IS NOT NULL (nem NULL érték keresése)
- AND (több feltétel egyidejű érvényesítése)
- OR (legalább egy feltétel teljesülése)
- NOT (tagadás)

Példák: ellenőrzésképpen a megjelenített mezők között legyen(ek) ott a feltételnél használt mező(k).

Azoknak a könyveknek a szerzője és címe, amiket 2003-ban adtak ki.

```
SELECT szerzo, cim
FROM konyvek
WHERE kiadasi_ev = 2003;
```

Azoknak a könyveknek a szerzője és címe, amiket nem 2003-ban adtak ki.

```
SELECT szerzo, cim
FROM konyvek
WHERE kiadasi_ev != 2003;
```

vagy

```
SELECT szerzo, cim
FROM konyvek
WHERE kiadasi_ev <> 2003;
```

Azoknak a könyveknek a szerzője, címe és oldalszáma, amelyek több, mint 500 oldalasak.

```
SELECT szerzo, cim, oldalszam
FROM konyvek
WHERE oldalszam > 500;
```

Azoknak a könyveknek a szerzője, címe és oldalszáma, amelyek kevesebb, mint 500 oldalasak.

```
SELECT szerzo, cim, oldalszam
FROM konyvek
WHERE oldalszam < 500;
```

Azoknak a könyveknek a szerzője, címe és oldalszáma, amelyek legalább 400 oldalasak.

```
SELECT szerzo, cim, oldalszam
FROM konyvek
WHERE oldalszam >= 400;
```

Azoknak a könyveknek a szerzője, címe és oldalszáma, amelyek legfeljebb 400 oldalasak.

```
SELECT szerzo, cim, oldalszam
FROM konyvek
WHERE oldalszam <= 400;
```

Azoknak a könyveknek a szerzője, címe és kiadási éve, amelyeket 1950 és 2000 között adtak ki.

```
SELECT szerzo, cim, kiadasi_ev
FROM konyvek
WHERE kiadasi_ev BETWEEN 1950 AND 2000;
```

A BETWEEN helyett meg lehet adni két feltételt is AND kapcsolattal.

A BETWEEN kulcsszó dátumok esetén is alkalmazható.

Azoknak a könyveknek a szerzője, címe és kiadója, amelyeket az Európa Kiadó és a Gabo Kiadó adott ki.

```
SELECT szerzo, cim, kiado
FROM konyvek
WHERE kiado IN ('Európa', 'Gabo');
```

Az IN kulcsszó helyett meg lehet adni több feltételt is OR kapcsolattal.

Azoknak a könyveknek a szerzője és címe, amelyeknek a műfaja fantasy.

```
SELECT szerzo, cim
FROM konyvek
WHERE mufaj LIKE 'Fantasy';
```

Azoknak a könyveknek a szerzője és címe, amelyeknek a címe tartalmazza az és kötőszót.

```
SELECT szerzo, cim
FROM konyvek
WHERE cim LIKE '% és %';
```

Ez minden olyan címet visszaad, amelyben az és kötőszó szerepel. Mivel előtte és utána szerepel szóköz karakter, így nem veszi figyelembe azokat a címeket, ahol az és valamelyik szóban szerepel.

A % karakter több, tetszőleges számú karaktert helyettesít.

Az aláhúzásjel egyetlen karaktert helyettesít.

Azoknak a könyveknek a szerzője és címe, amelyeknél nincs megadva a kiadási év.

```
SELECT szerzo, cim
FROM konyvek
WHERE kiadasi_ev IS NULL;
```

Azoknak a könyveknek a szerzője, címe és kiadási éve, amelyeknél meg van adva a kiadási év.

```
SELECT szerzo, cim, kiadasi_ev
FROM konyvek
WHERE kiadasi_ev IS NOT NULL;
```

Azoknak a könyveknek a szerzője, címe, kiadója és műfaja, amelyeket az Európa Kiadó adott ki és a műfaja fantasy.

```
SELECT szerzo, cim, kiado, mufaj
FROM konyvek
WHERE kiado LIKE 'Európa' AND mufaj LIKE 'Fantasy';
```

Azoknak a könyveknek a szerzője, címe és kiadási éve, amelyeket 1950 és 2000 között adtak ki (BETWEEN helyett két feltétel AND kapcsolattal).

```
SELECT szerzo, cim, kiadasi_ev
FROM konyvek
```

```
WHERE kiadasi_ev >= 1950 AND kiadasi_ev <= 2000;
```

Azoknak a könyveknek a szerzője, címe és kiadója, amelyeket az Európa Kiadó és a Gabo Kiadó adott ki (IN helyett két feltétel OR kapcsolattal).

```
SELECT szerzo, cim, kiado
FROM konyvek
WHERE kiado LIKE 'Európa' OR kiado LIKE 'Gabo';
```

Azoknak a könyveknek a szerzője, címe és kiadója, amelyeket nem az Európa Kiadó adott ki.

```
SELECT szerzo, cim, kiado
FROM konyvek
WHERE kiado NOT LIKE 'Európa';
```

Az adatsorok rendezése az ORDER BY záradékkal

Az ORDER BY záradék segítségével rendezhetjük az adatokat növekvő vagy csökkenő sorrendben. Alapértelmezés szerint az ORDER BY növekvő sorrendet használ, de a DESC kulcsszóval csökkenő sorrendet is beállíthatunk.

Szintaxis:

```
SELECT oszlop1, oszlop2, ...
FROM tabla_neve
[WHERE feltétel]
ORDER BY oszlop1 [ASC|DESC], oszlop2 [ASC|DESC], ...;
```

A rendezés lehet többkulcsos is, több oszlopot is megadhatunk az ORDER BY záradékban.

ASC: növekvő sorrend (alapértelmezett).
DESC: csökkenő sorrend.

Példák:

Az összes könyv lekérdezése (és minden mező) cím szerint abc sorrendben (A-Z).

```
SELECT *
FROM konyvek
ORDER BY cim;
```

vagy

```
SELECT *
FROM konyvek
ORDER BY cim ASC;
```

Az összes könyv lekérdezése (és minden mező) cím szerint fordított abc sorrendben (Z-A).

```
SELECT *
FROM konyvek
ORDER BY cim DESC;
```

Az összes könyv szerzője, címe és oldalszáma oldalszám szerint növekvő sorrendben.

```
SELECT szerzo, cim, oldalszam
FROM konyvek
ORDER BY oldalszam;
```

Az összes könyv szerzője, címe és kiadási éve kiadási év szerint csökkenő sorrendben.

```
SELECT szerzo, cim, kiadasi_ev
FROM konyvek
ORDER BY kiadasi_ev DESC;
```

A NULL értékek mindig a legvégére kerülnek, ha csökkenő sorrendben rendezzük őket.

A többkulcsos rendezésnél több oszlopot is megadhatunk az ORDER BY záradékban és mindegyiknél megadhatjuk a rendezés irányát. Az oszlopok lehetnek különböző típusúak is.

Az összes könyv szerzője, címe, kiadója és kiadási éve kiadó szerint abc sorrendben és azon belül kiadási év szerint csökkenő sorrendben.

```
SELECT szerzo, cim, kiado, kiadasi_ev
FROM konyvek
ORDER BY kiado, kiadasi_ev DESC;
```

A rendezett eredményeket korlátozhatjuk a LIMIT kulcsszóval, hogy csak egy részét jelenítsük meg.

```
SELECT *
FROM konyvek
ORDER BY kiadasi_ev DESC
LIMIT 5;
```

Ez az utasítás az öt legújabb könyvet adja vissza.

Az álnevek szerepe és használata a lekérdezésekben

Az álnevek (aliasok) segítségével az oszlopok és táblák neveit rövidebbé, érthetőbbé vagy jobban olvashatóvá tehetjük a lekérdezésekben. Az álnevek nem módosítják az eredeti adatokat, csak a lekérdezés eredményében jelennek meg.

Az oszlopok álneve (AS):

Az oszlopok álnevezéséhez az AS kulcsszót használjuk, de az AS el is hagyható.

Szintaxis:

```
SELECT oszlop_neve AS alias_nev
FROM tabla_neve
```

Például:

```
SELECT cim AS 'Könyv címe'
FROM konyvek
```

vagy

```
SELECT cim 'Könyv címe'
FROM konyvek
```

Az eredményben a cim oszlop neve Könyv címe lesz.

Ha az álnév szóközt vagy speciális karaktert tartalmaz, idézőjelek, aposztrófok vagy backtick-ek közé kell tenni.

A táblák álneve (AS):

A táblák neveit is rövidíthetjük vagy egyértelműbbé tehetjük álnévvel.

Szintaxis:

```
SELECT alias.oszlop1, alias.oszlop2, …
FROM tabla_neve AS alias;
```

Például:

```
SELECT k.szerzo, k.cim
FROM konyvek AS k;
```

vagy

```
SELECT k.szerzo, k.cim
FROM konyvek k;
```

A konyvek tábla rövid neve k lett, így az oszlopokat a k.cim és a k.szerzo formában hivatkozhatjuk. Ennek többtáblás lekérdezéseknél van inkább jelentősége, ahol több táblában azonos nevű oszlopok vannak.

Az oszlopok álneve megjelenik az eredményben, a táblák álneve a többtáblás lekérdezésekben segít az egyértelmű azonosításban.

A helyettesítő (wildcard) karakterek és alkalmazásuk

A helyettesítő karakterek (wildcards) speciális karakterek, amelyeket a LIKE operátorral használunk szöveges keresések során az SQL-ben. Ezek lehetővé teszik, hogy rugalmasan keressünk részleges egyezéseket az adatokban.

% - tetszőleges számú karakter helyettesítése (0 vagy több)

Például:

```
SELECT cim
FROM konyvek
WHERE cim LIKE 'A %';
```

Az eredmény minden cím, amely 'A' névelővel kezdődik.

_ (aláhúzás) - egy karakter helyettesítése

Az adott pozíción bármi szerepelhet, de a többi résznek meg kell egyeznie a keresési feltétellel.

Például a zuglói irányítószámok mind 114-el kezdődnek, de az utolsó számjegy tetszőleges lehet.

```
SELECT nev
FROM olvasok
WHERE iranyitoszam LIKE '114_';
```

Az ismétlődő sorok elnyomása, a DISTINCT záradék alkalmazása

A DISTINCT záradék segítségével eltávolíthatjuk az ismétlődő sorokat egy lekérdezés eredményéből. Ha egy oszlop vagy több oszlop alapján keresünk adatokat, és nem akarjuk, hogy ugyanazok az értékek többször is szerepeljenek, akkor a DISTINCT kulcsszót használjuk.

Szintaxis:

```
SELECT DISTINCT oszlop1, oszlop2, ...
FROM tabla_neve
[WHERE feltétel];
```

Például:

```
SELECT DISTINCT kiado
FROM konyvek
```

A konyvek táblában több könyvnek is ugyanaz a kiadója. Ezért ha lekérdezzük a kiadókat, az eredményben ismétlődések lesznek. A DISTINCT kulcsszó használata esetén minden kiadó egyszer szerepel az eredményben.

A DISTINCT záradék az összes oszlopra vonatkozik, amelyeket a lekérdezésben felsorolunk. Ha több oszlopot kérdezünk le, akkor a kombinált értékeket tekinti egyedinek.

A táblák összekapcsolása során alkalmazott záradékok (INNER, LEFT, RIGHT JOIN)

A táblák összekapcsolása (más néven JOIN műveletek) lehetővé teszi, hogy több táblát egyszerre lekérdezzünk, és összekapcsoljuk azokat közös mezők alapján. A leggyakrabban használt összekapcsolási típusok az INNER JOIN, a LEFT JOIN, és a RIGHT JOIN.

INNER JOIN

Az INNER JOIN összekapcsolja a táblákat és csak azokat a rekordokat adja vissza, ahol mindkét táblában van megfelelő adat. Ha egy rekord egyik táblában szerepel, de a másikban nem, akkor az nem fog szerepelni az eredményben.

Szintaxis:

```
SELECT oszlopok
FROM tabla1
INNER JOIN tabla2
ON tabla1.oszlop = tabla2.oszlop;
```

Tegyük fel, hogy van egy diakok és egy eredmenyek tábla egy vizsgak nevű adatbázisban. A diakok tábla a diákok adatait, az eredmenyek tábla pedig a diákok vizsga eredményeit tartalmazza.

```
CREATE TABLE diakok (
id INT PRIMARY KEY AUTO_INCREMENT,
nev VARCHAR(100),
email VARCHAR(100) UNIQUE
);

CREATE TABLE eredmenyek (
id INT PRIMARY KEY AUTO_INCREMENT,
diak_id INT,
tantargy VARCHAR(100),
eredmeny INT,
```

```
FOREIGN KEY (diak_id) REFERENCES diakok(id)
);

INSERT INTO diakok (nev, email) VALUES
('Kiss Péter', 'peter.kiss@example.com'),
('Nagy Anna', 'anna.nagy@example.com'),
('Horváth János', 'janos.horvath@example.com'),
('Szabó László', 'laszlo.szabo@example.com'),
('Tóth Marianna', 'marianna.toth@example.com'),
('Bognár Zoltán', 'zoltan.bognar@example.com'),
('Varga Eszter', 'eszter.varga@example.com'),
('Kovács Dávid', 'david.kovacs@example.com'),
('Balogh Katalin', 'katalin.balogh@example.com'),
('Farkas Gábor', 'gabor.farkas@example.com');

INSERT INTO eredmenyek (diak_id, tantargy, eredmeny) VALUES
(1, 'Matematika', 85),
(1, 'Fizika', 90),
(2, 'Matematika', 78),
(2, 'Fizika', 92),
(3, 'Matematika', 88),
(4, 'Matematika', 95),
(5, 'Fizika', 80),
(6, 'Matematika', 70),
(7, 'Fizika', 65),
(8, 'Matematika', 90);
```

Nem minden diákhoz tartozik eredmény, tehát a 9. és 10. diák nem szerepel az eredmenyek táblában.

Az alábbi lekérdezés a diákok nevét és a hozzájuk tartozó vizsgaeredményeket (tantárgy és eredmény) adja vissza:

```
SELECT diakok.nev, eredmenyek.tantargy, eredmenyek.eredmeny
FROM diakok
INNER JOIN eredmenyek
ON diakok.id = eredmenyek.diak_id;
```

Ez a lekérdezés csak azokat a diákokat és eredményeket fogja visszaadni, akiknek van vizsgaeredményük (tehát mindkét táblában található hozzájuk adat).

LEFT JOIN

A LEFT JOIN (más néven LEFT OUTER JOIN) minden rekordot visszaad az első (bal) táblából, és a hozzá tartozó adatokat a második (jobb) táblából. Ha a jobb oldali táblában nincs megfelelő rekord, akkor NULL értéket kapunk.

Szintaxis:

```
SELECT oszlopok
FROM tabla1
LEFT JOIN tabla2
ON tabla1.oszlop = tabla2.oszlop;
```

Ha szeretnénk minden diák nevét és az ő vizsga eredményeiket lekérdezni, még akkor is, ha van olyan diák, akinek nincs eredménye (vagyis nincs megfelelő adat a eredmenyek táblában):

```
SELECT diakok.nev, eredmenyek.tantargy, eredmenyek.eredmeny
FROM diakok
LEFT JOIN eredmenyek
ON diakok.id = eredmenyek.diak_id;
```

Ez a lekérdezés minden diákot visszaad, és ha van vizsga eredményük, akkor az is megjelenik. Ha egy diáknak nincs eredménye, akkor NULL értéket kap a tantárgy és az eredmény helyén.

RIGHT JOIN

A RIGHT JOIN (más néven RIGHT OUTER JOIN) a LEFT JOIN ellentéte. Az összes rekordot visszaadja a második (jobb) táblából, és az első (bal) táblából a hozzájuk tartozó adatokat. Ha a bal oldali táblában nincs megfelelő rekord, akkor NULL értéket kapunk.

Szintaxis:

```
SELECT oszlopok
FROM tabla1
RIGHT JOIN tabla2
ON tabla1.oszlop = tabla2.oszlop;
```

Tegyük fel, hogy van egy alkalmazottak és egy osztalyok tábla egy alkalmazottak nevű adatbázisban.

```
CREATE TABLE osztalyok(
id INT PRIMARY KEY AUTO_INCREMENT,
nev VARCHAR(100)
);

CREATE TABLE alkalmazottak(
id INT PRIMARY KEY AUTO_INCREMENT,
nev VARCHAR(100),
osztaly_id INT,
FOREIGN KEY (osztaly_id) REFERENCES osztalyok(id)
);

INSERT INTO osztalyok (nev) VALUES
('HR'),
('Informatika'),
('Értékesítés'),
('Marketing'),
('Pénzügy');

INSERT INTO alkalmazottak (nev, osztaly_id) VALUES
('Kovács János', 1),
('Szabó Anna', 1),
('Nagy Péter', 2),
('Horváth László', 2),
('Kovács Katalin', 3),
('Varga Gábor', 3),
('Tóth Zsuzsa', 4),
('Farkas Miklós', 4);
```

Nem minden osztályhoz tartozik alkalmazott, tehát az 5. osztály nem szerepel az alkalmazottak táblában.

A RIGHT JOIN minden rekordot visszaad a jobb oldali táblából (osztalyok), és az alkalmazottakhoz tartozó adatokat, ha azok léteznek.

```
SELECT alkalmazottak.nev, osztalyok.nev
FROM alkalmazottak
RIGHT JOIN osztalyok ON alkalmazottak.osztaly_id = osztalyok.id;
```

A Pénzügy osztály esetében meg jeleníti a NULL értéket, mivel nem rendeltünk alkalmazottat hozzá.

Az adatok csoportosítására szolgáló GROUP BY záradék használata

A GROUP BY záradékot akkor használjuk, ha szeretnénk az adatokat valamilyen mező alapján csoportosítani, és az egyes csoportokra vonatkozóan összesítő (aggregált) értékeket számítani (például darabszám, átlag, összeg, stb.).

Szintaxis:

```
SELECT oszlop1, AGGREGALT_FUGGVENY(oszlop2)
FROM tabla_neve
GROUP BY oszlop1;
```

Gyakori aggregált függvények:

- COUNT() – sorok száma
- SUM() – összegzés
- AVG() – átlag
- MIN() – minimum
- MAX() – maximum

Példák:

```
CREATE TABLE alkalmazottak (
id INT PRIMARY KEY,
nev VARCHAR(100),
osztaly VARCHAR(50),
varos VARCHAR(50),
fizetes INT
);

INSERT INTO alkalmazottak (id, nev, osztaly, varos, fizetes) VALUES
(1, 'Kovács Anna', 'HR', 'Budapest', 450000),
(2, 'Szabó Péter', 'IT', 'Budapest', 600000),
(3, 'Nagy Zoltán', 'HR', 'Debrecen', 470000),
(4, 'Tóth Márta', 'IT', 'Debrecen', 650000),
(5, 'Fekete László', 'Pénzügy', 'Budapest', 500000),
(6, 'Kiss Júlia', 'Pénzügy', 'Győr', 520000),
(7, 'Varga Tamás', 'IT', 'Győr', 610000),
(8, 'Balogh Réka', 'HR', 'Győr', 430000),
```

```
(9, 'Lukács András', 'Logisztika', 'Pécs', 480000),
(10, 'Molnár Kitti', 'IT', 'Szeged', 620000),
(11, 'Papp Gábor', 'HR', 'Pécs', 460000),
(12, 'Németh Eszter', 'Pénzügy', 'Debrecen', 510000),
(13, 'Török Róbert', 'Logisztika', 'Győr', 495000),
(14, 'Fehér Dóra', 'IT', 'Budapest', 640000),
(15, 'Jakab Lili', 'HR', 'Budapest', 455000),
(16, 'Sipos Bence', 'Pénzügy', 'Szeged', 530000),
(17, 'Bognár Emese', 'IT', 'Debrecen', 625000),
(18, 'Simon Richárd', 'Logisztika', 'Szeged', 470000),
(19, 'Oláh Petra', 'Pénzügy', 'Győr', 515000),
(20, 'Horváth Levente', 'HR', 'Debrecen', 440000);
```

Hány alkalmazott dolgozik az egyes osztályokon?

```
SELECT osztaly AS Osztály, COUNT(*) AS Létszám
FROM alkalmazottak
GROUP BY osztaly;
```

A lekérdezés eredményében megjelenik az osztályok neve, a csoportosítás miatt csak egyszer. A COUNT függvény megszámolja, hány olyan rekord van, ahol az adott osztály szerepel. Bármelyik olyan mezőnevet használhatjuk, amelyikben nincs NULL érték. A COUNT(*) viszont minden sort megszámol, függetlenül attól, hogy a sorban valamelyik mező NULL értékű-e.

Mekkora az átlagfizetés osztályonként?

```
SELECT osztaly AS Osztály, AVG(fizetes) AS 'Átlag fizetés'
FROM alkalmazottak
GROUP BY osztaly;
```

A lekérdezés eredményében megjelenik az osztályok neve, a csoportosítás miatt csak egyszer. Az AVG függvény osztályonként átlagot számol a fizetésekből.

Ha GROUP BY-t használunk, akkor a SELECT-ben azokat a mezőket szerepeltessük, amelyek benne vannak a GROUP BY-ban, vagy olyan mezőket, amelyekre aggregált függvényeket alkalmazunk. Az aggregált függvényeket csak a csoportosított sorokon lehet értelmezni.

Példa hibás lekérdezésre:

```
SELECT osztaly, nev, AVG(fizetes)
FROM alkalmazottak
GROUP BY osztaly;
```

Az osztály benne van a GROUP BY-ban, a fizetés pedig egy aggregált függvényben (AVG) van, ezek rendben vannak. A név viszont nincs se a GROUP BY-ban és nem is aggregált. Adatbázis-kezelőtől függően vagy hibát kapunk vagy a lekérdezés nem helyes eredményt ad. Például valamelyik tetszőleges értéket jeleníti meg az adott csoportból, vagy az első értéket jeleníti meg az adott csoportból.

Mikor használjunk DISTINCT-et és mikor GROUP BY-t?

Ha a cél az ismétlődő sorok kiszűrése, akkor DISTINCT, ha az adatokat csoportosítani szeretnénk és összesíteni, akkor GROUP BY.

Ha használni szeretnénk WHERE záradékot, akkor azt még a csoportosítás előtt alkalmazzuk a sorok szűrésére. Ha a már csoportosított adatokra szeretnénk szűrést, akkor a HAVING záradékot használjuk (következő témakör).

Példa: hány budapesti alkalmazott dolgozik az egyes osztályokon?

```
SELECT osztaly AS Osztály, COUNT(*) AS Létszám
FROM alkalmazottak
WHERE varos LIKE 'Budapest'
GROUP BY osztaly;
```

<u>A csoportosított adatok szűrése során használt HAVING záradék</u>

A HAVING záradékot akkor használjuk, amikor a csoportosított eredményekre akarunk szűrni, azaz miután az adatok már csoportosítva lettek a GROUP BY záradék segítségével.

A WHERE szűri a sorokat a csoportosítás előtt, a HAVING pedig szűri a csoportosított eredményeket.

Szintaxis:

```
SELECT oszlop1, AGGREGALT_FUGGVENY(oszlop2)
FROM tabla_neve
```

```
GROUP BY oszlop1;
HAVING feltetel;
```

Példa: hány alkalmazott dolgozik az egyes osztályokon? Csak azok az osztályok jelenjenek meg, ahol legalább 5 alkalmazott dolgozik.

```
SELECT osztaly AS Osztály, COUNT(*) AS Létszám
FROM alkalmazottak
GROUP BY osztaly
HAVING COUNT(*) >= 5;
```

vagy

```
SELECT osztaly AS Osztály, COUNT(*) AS Létszám
FROM alkalmazottak
GROUP BY osztaly
HAVING Létszám >= 5;
```

Az aggregált függvény újbóli használata helyett az alias nevet is megadhatjuk a feltételben.

Az osztályokat csoportosítjuk. Az alkalmazottak számát osztályonként a COUNT aggregált függvénnyel megszámoljuk. Csak azokat az osztályokat jelenítjük meg, ahol több, mint 5 alkalmazott dolgozik.

Melyek azok az osztályok, ahol a legnagyobb fizetés meghaladja az 500 000 Ft-ot?

```
SELECT osztaly AS Osztály, MAX(fizetes) AS 'Legnagyobb fizetés'
FROM alkalmazottak
GROUP BY osztaly
HAVING MAX(fizetes) > 500000;
```

A HAVING szűrés csak az aggregált adatokra vonatkozik. A HAVING a GROUP BY után működik, tehát csak akkor alkalmazható, ha a lekérdezés tartalmaz GROUP BY záradékot.

<u>A megjelenő adatsorok limitálása során használt záradék (LIMIT)</u>

A LIMIT záradék segítségével meghatározhatjuk, hogy hány rekordot jelenítsen meg a lekérdezés.

Szintaxis:

```
SELECT oszlop1, oszlop2, …
FROM tabla_neve
LIMIT rekordok_szama;
```

Szűrés a rendezés alapján: gyakran együtt használható a ORDER BY záradékkal.

Példa: a három legnagyobb fizetéssel rendelkező alkalmazott neve és fizetése

```
SELECT nev, fizetes
FROM alkalmazottak
ORDER BY fizetes DESC
LIMIT 3;
```

Az alkalmazottakat fizetés szerint csökkenő sorrendbe rendezzük, majd az első 3 rekordot jelenítjük meg.

A LIMIT használata nem csak a rekordok számának meghatározására szolgál, hanem lehetővé teszi a kezdő pozíció megadását is.

Példa: a közepes fizetéssel rendelkező alkalmazottak neve és fizetése (a 6. rekordtól a 15. rekordig)

```
SELECT nev, fizetes
FROM alkalmazottak
ORDER BY fizetes
LIMIT 10 OFFSET 5;
```

Az alkalmazottakat fizetés szerint növekvő sorrendbe rendezzük, majd első 5 rekordot kihagyjuk (OFFSET) és a 6. rekordtól kezdve megjelenítjük a következő 10 rekordot.

Mennyi a legnagyobb fizetés? (függvénnyel)

```
SELECT MAX(fizetes) AS 'Legnagyobb fizetés'
FROM alkalmazottak;
```

Mennyi a legnagyobb fizetés? ORDER BY + LIMIT

```
SELECT fizetes AS 'Legnagyobb fizetés'
FROM alkalmazottak
ORDER BY fizetes DESC
LIMIT 1;
```

Ha csak az értékre van szükségünk, érdemes a MAX függvényt használni. Ha a teljes rekord kell, akkor az ORDER BY + LIMIT kombinációt használjuk.

Számított mezők készítése

A számított mezők lehetővé teszik, hogy új, dinamikusan kiszámolt oszlopokat jelenítsünk meg.

A számított mező egy olyan oszlop, amely nem szerepel az adatbázisban valódi mezőként, hanem a lekérdezés futása közben valamilyen művelet eredményeként jön létre. Például összeadással, szorzással, kivonással, osztással, szöveg összefűzéssel vagy akár függvényekkel.

Szintaxis:

```
SELECT kifejezes AS alnev
FROM tabla_neve;
```

A kifejezés az a művelet vagy számítás, amit egy valódi mezővel elvégzünk. Az álnév pedig az új oszlop neve.

Példa: minden alkalmazott kap egy 50 000 Ft-os bónuszt a fizetésével együtt, jelenítsük meg az alkalmazottak nevét, fizetését, és bónusszal növelt összeget

```
SELECT nev AS Név, fizetes AS Fizetés, fizetes + 50000 AS 'Fizetés bónusszal'
FROM alkalmazottak;
```

Minden alkalmazott kap 10 % fizetésemelést, jelenítsük meg az alkalmazottak nevét, jelenlegi fizetését és a megemelt fizetését.

```
SELECT nev AS Név, fizetes AS Fizetés, fizetes * 1.1 AS 'Megemelt fizetés'
FROM alkalmazottak;
```

A számított mezőre akár ORDER BY vagy HAVING záradékban is hivatkozhatunk.

Azok az alkalmazottak, akik 500 000 Ft-nál kevesebbet keresnek, kapnak 50 000 Ft fizetésemelést, jelenítsük meg az alkalmazottak nevét, fizetését és a megemelt fizetését, és rendezzük megemelt fizetés szerint csökkenő sorrendbe.

```
SELECT nev AS Név, fizetes AS Fizetés,
CASE
WHEN fizetes < 500000 THEN fizetes + 50000
ELSE fizetes
END AS 'Megemelt fizetés'
FROM alkalmazottak
ORDER BY 3 DESC;
```

Elágazás segítségével valósítjuk meg, hogy a számított mezőbe 500 000 Ft alatti fizetések esetén 50 000 Ft-al növelt összegek kerüljenek, egyébként az összeg ne változzon. Ha az ORDER BY-ban a számított mező álnevére hivatkozunk, nem működik, ezért használjuk az oszlop sorszámát.

Melyek azok az osztályok, ahol az átlagfizetés 500 000 Ft felett van? Jelenítsük meg az osztályok nevét és az ott dolgozók átlagfizetését.

```
SELECT osztaly AS Osztály, AVG(fizetes) AS Átlagfizetés
FROM alkalmazottak
GROUP BY osztaly
HAVING Átlagfizetés > 500000;
```

<u>Az aggregált függvények (COUNT(), MIN(), MAX(), SUM(), AVG()) használata</u>

Ezek a függvények több sorból számolnak ki egyetlen eredményt, és leggyakrabban összegzésekhez, statisztikai elemzésekhez használjuk őket.

- COUNT() – sorok megszámlálása
- MIN() – legkisebb érték
- MAX() – legnagyobb érték
- SUM() – összeadás
- AVG() – átlag számítása

Hány alkalmazott van összesen? (COUNT)

```
SELECT COUNT(*) AS 'Alkalmazottak száma'
FROM alkalmazottak;
```

Mennyi a teljes bérköltség? (SUM)

```
SELECT SUM(fizetes) AS 'Teljes bérköltség'
FROM alkalmazottak;
```

Mennyi a legkisebb és a legnagyobb fizetés? (MIN és MAX)

```
SELECT MIN(fizetes) AS 'Legkisebb fizetés', MAX(fizetes) AS 'Legnagyobb fizetés'
FROM alkalmazottak;
```

Mennyi az alkalmazottak átlagfizetése? (AVG)

```
SELECT AVG(fizetes) AS 'Átlagfizetés'
FROM alkalmazottak;
```

<u>A lekérdezésekben használt egyéb függvények</u>

Szöveg függvények

CONCAT (karakterláncok összefűzése):

```
CREATE TABLE alkalmazottak (
id INT PRIMARY KEY AUTO_INCREMENT,
vezeteknev VARCHAR(50),
keresztnev VARCHAR(50),
osztaly VARCHAR(50),
cim VARCHAR(100),
fizetes INT,
email VARCHAR(100)
);

INSERT INTO alkalmazottak (vezeteknev, keresztnev, osztaly, cim, fizetes, email)
VALUES
('Kovács', 'Anna', 'HR', 'Budapest, Fő tér 12.', 450000,
'anna.kovacs@example.com'),
('Szabó', 'Péter', 'IT', 'Budapest, Széchenyi körút 8.', 600000,
'peter.szabo@example.com'),
('Nagy', 'Zoltán', 'HR', 'Debrecen, Petőfi u. 5.', 470000, 'zoltan.nagy@example.com'),
('Tóth', 'Márta', 'IT', 'Debrecen, Bem tér 10.', 650000, 'marta.toth@example.com'),
('Fekete', 'László', 'Pénzügy', 'Budapest, Munkácsy Mihály utca 4.', 500000,
'laszlo.fekete@example.com'),
```

```
('Kiss', 'Júlia', 'Pénzügy', 'Győr, Árpád sétány 6.', 520000, 'julia.kiss@example.com'),
('Varga', 'Tamás', 'IT', 'Győr, Ady Endre út 7.', 610000, 'tamas.varga@example.com'),
('Balogh', 'Réka', 'HR', 'Győr, Rózsa köz 11.', 430000, 'reka.balogh@example.com'),
('Lukács', 'András', 'Logisztika', 'Pécs, Malom utca 2.', 480000,
'andras.lukacs@example.com'),
('Molnár', 'Kitti', 'IT', 'Szeged, Tisza Lajos körút 14.', 620000,
'kitti.molnar@example.com'),
('Papp', 'Gábor', 'HR', 'Pécs, Rákóczi út 3.', 460000, 'gabor.papp@example.com'),
('Németh', 'Eszter', 'Pénzügy', 'Debrecen, Kossuth tér 5.', 510000,
'eszter.nemeth@example.com'),
('Török', 'Róbert', 'Logisztika', 'Győr, Vasút sor 9.', 495000,
'robert.torok@example.com'),
('Fehér', 'Dóra', 'IT', 'Budapest, Bartók Béla út 1.', 640000,
'dora.feher@example.com'),
('Jakab', 'Lili', 'HR', 'Budapest, Dózsa György út 13.', 455000,
'lili.jakab@example.com'),
('Sipos', 'Bence', 'Pénzügy', 'Szeged, Attila utca 6.', 530000,
'bence.sipos@example.com'),
('Bognár', 'Emese', 'IT', 'Debrecen, Hunyadi János utca 12.', 625000,
'emese.bognar@example.com'),
('Simon', 'Richárd', 'Logisztika', 'Szeged, Arany János tér 4.', 470000,
'richard.simon@example.com'),
('Oláh', 'Petra', 'Pénzügy', 'Győr, Báthory utca 8.', 515000,
'petra.olah@example.com'),
('Horváth', 'Levente', 'HR', 'Debrecen, Csapó utca 10.', 440000,
'levente.horvath@example.com');
```

Az alkalmazottak teljes neve a vezetéknév és a keresztnév összefűzésével.

```
SELECT CONCAT(vezeteknev, ' ', keresztnev) AS Név
FROM alkalmazottak;
```

LENGTH (szöveg hossza karakterben):

Jelenítsük meg az e-mail címeket és az e-mail címek hosszát.

```
SELECT email AS 'E-mail cím', LENGTH(email) AS Hossz
FROM alkalmazottak;
```

SUBSTR (szöveg egy részének kivágása):

Jelenítsük meg külön a felhasználók azonosítóit és domain neveit az e-mail címekből.

```
SELECT email AS 'E-mail cím', SUBSTR(email, 1, INSTR(email, '@') - 1) AS
Azonosító, SUBSTR(email, INSTR(email, '@') + 1) AS 'Domain név'
FROM alkalmazottak;
```

Az INSTR függvény megtalálja a @ karakter pozícióját az e-mail címben. Az azonosító az első karakternél kezdődik és a @ karakter előtt végződik. A domain név a @ karakter után kezdődik és az utolsó karakterig tart (ebben az esetben a SUBSTR függvénynek elég két paramétert megadni).

REPLACE (szöveg egy részének cseréje):

Listázzuk ki az osztályok neveit. Az IT osztályt Informatika néven szeretnénk megjeleníteni.

```
SELECT DISTINCT REPLACE(osztaly, 'IT', 'Informatika') AS 'Osztály'
FROM alkalmazottak;
```

Jelenítsük meg az alkalmazottak vezetéknevét, keresztnevét és címét. Ahol az utca úgy szerepel, hogy u., ott is utca jelenjen meg.

```
SELECT vezeteknev AS Vezetéknév, keresztnev AS Keresztnév, REPLACE(cim, 'u.',
'utca') AS Cím
FROM alkalmazottak;
```