Prírodovedecká fakulta UPJŠ Košice

Nám z DBS manuál nestačí II.

(neoficiálne skriptá pre letný semester Databázových systémov)

prednáša: RNDr. Stanislav Krajči, PhD.

Verzia 29. 5. 2002 (δ, X)
Zostavila Petra Murtinová
Portions: Róbert Novotný
`http://skmi.science.upjs.sk/~novotnyr`

Obsah

4 Jazyk SQL **3**
4.1 Úvod . . . 3
4.1.1 História . . . 3
4.1.2 Rysy modelovania dát pomocou SQL . . . 3
4.1.3 Použitie SQL . . . 3
4.2 Definícia dát – metadáta . . . 3
4.2.1 Numerické typy . . . 4
4.2.2 Znakové reťazce . . . 4
4.2.3 Veľké údaje – `LOBS` . . . 4
4.2.4 Časové údaje . . . 4
4.2.5 Linky na dáta uložené mimo databázy . . . 4
4.3 Práca s tabuľkou . . . 5
4.3.1 Vytvorenie tabuľky – príkaz `CREATE TABLE` . . . 5
4.3.2 Úprava štruktúry existujúcej tabuľky – príkaz `ALTER TABLE` . . . 6
4.3.3 Zrušenie tabuľky – príkaz `DROP TABLE` . . . 8
4.3.4 Indexy . . . 8
4.3.5 Pohľady (views) . . . 9
4.3.6 Systémový katalóg . . . 9
4.4 Príkaz `SELECT` . . . 10
4.4.1 `SELECT-FROM-WHERE`-blok (skrátene `SELECT`) . . . 10
4.4.2 Niektoré užitočné funkcie . . . 14
4.4.3 Agregačné funkcie . . . 16
4.4.4 Spojenia tabuliek . . . 17
4.5 Zložitejšie nadstavby nad príkazom SELECT . . . 19
4.5.1 Množinové operácie (`INTERSECT`, `UNION`, `EXCEPT`) . . . 19
4.5.2 Pomocné tabuľky – klauzula `WITH` . . . 20
4.5.3 Tranzitívny uzáver . . . 21
4.6 Aktualizácia dát . . . 23
4.6.1 `INSERT` – vkladanie nových záznamov . . . 23
4.6.2 `DELETE` – mazanie existujúcich záznamov . . . 24
4.6.3 `UPDATE` – zmena hodnôt existujúcich záznamov . . . 24
4.7 Triggery . . . 25
4.8 Ochrana dát . . . 26
4.9 Integrácia SQL do programovacieho jazyka . . . 27

4 Jazyk SQL

4.1 Úvod

4.1.1 História

- v roku 1974 vznikol dopytovací jazyk Sequel (**S**tructured **E**nglish **Que**ry **L**anguage) bol súčasťou Systému R vyvinutého v laboratóriu IBM v San José (zamestnávateľ autora relačného modelu dát E. F. Codda); po úprave koncepcie a zjednodušení mena na SQL (**S**tructured **Q**uery **L**anguage) sa stal základom prvých významnejších databázových produktov DB2, SQL/DS a QMF od IBM
- s nástupom PC v 80. rokoch sa mnoho svetových výrobcov software-u začalo orientovať na implementáciu SQL, od jednopoužívateľských sa postupne prechádzalo na databázové servery založené na SQL
- v roku 1986 nastal zlom – štandardizácia SQL organizáciou ANSI (pod názvom SQL86), v roku 1987 bola táto štandardizácia prijatá organizáciou ISO; štandard bol dialekt SQL firmy IBM a bol charakterizovaný ako „prienik existujúcich implementácií“
- v roku 1989 sa jazyk rozšíril o možnosť definovať integritné obmedzenia, táto verzia sa volá SQL89
- ďalší vývoj vyústil v roku 1992 do štandardu SQL92, ktorý ešte doteraz nie je úplne implementovaný
- ďalšie štandardy
  - X/OPEN SQL súvisí s použitím UNIX-u ako implementačného prostredia
  - SAA-SQL (Systems Application Architecture Database Interface) – vlastný štandard firmy IBM

4.1.2 Rysy modelovania dát pomocou SQL

- SQL je neprocedurálny jazyk (popisuje, čo požadujeme od databázy, nie to, ako to treba urobiť)
- dáta sú vždy prezentované používateľovi ako tabuľky, bez ohľadu na ich vnútornú štruktúru použitú v databáze
- poloha tabuliek v databáze nie je dôležitá, sú identifikované menom
- poradie stĺpcov v tabuľke nie je dôležité, sú identifikované menom
- poradie riadkov v tabuľke nie je dôležité, sú identifikované hodnotami v stĺpcoch

4.1.3 Použitie SQL

- dopytovací a manipulačný jazyk pre relačné databázy
- zložka hostiteľského jazyka na programovanie databázových aplikácií (*embedded SQL*)
- jazyk komunikácie medzi rôznymi zdrojmi dát

V ďalšom sa budeme zaoberať SQL vo verzii DB2

4.2 Definícia dát – metadáta

Najpoužívanejšie dátové typy si rozdelíme do niekoľkých logických skupín.

4.2.1 Numerické typy

- presné numerické typy
  - celé čísla
    - `INTEGER` (alebo `INT`) – 4 byte
    - `SMALLINT` – 2 byte
    - `BIGINT` – 8 byte
  - desatinné čísla
    - `DECIMAL` (alebo `DEC`) alebo
      `NUMERIC` (alebo `NUM`) `(`*počet_cifier* `[,` *počet_desatinných_miest*`])`
- aproximatívne numerické typy
  - reálne čísla
    - `FLOAT` alebo `FLOAT(`*presnosť*`)`
    - `REAL` – pevná presnosť (daná implementáciou)
    - `DOUBLE` (alebo `DOUBLE PRECISION`) – pevná presnosť (dvojnásobná než pri `REAL`)

4.2.2 Znakové reťazce

- `CHARACTER` (alebo `CHAR`) `(`*počet_znakov*`)`
  - kratší reťazec doplnený prázdnymi znakmi – počet znakov do 254
- `CHARACTER VARYING` (alebo `VARCHAR`) `(`*maximálny_počet_znakov*`)`
  - kratší reťazec nie je doplnený prázdnymi znakmi
  - počet znakov do 32 672

4.2.3 Veľké údaje – `LOBS`

Nazývajú sa aj *loby* (z angl. **L**arge **O**bjects)

- `BLOB` (veľkosť (v B)) alebo (veľkosť K alebo M alebo G) – binárne
- `CLOB` (veľkosť (v B)) alebo (veľkosť K alebo M alebo G) – reťazcové

4.2.4 Časové údaje

- `DATE` – dátum
  - spravidla v tvare `YYYY-MM-DD`, ale povolené je aj `D.M.YYYY`
  - `CURRENT DATE` – aktuálny dátum
- `TIME` – čas
  - v tvare `HH:MM:SS`
  - `CURRENT TIME` – aktuálny čas
- `TIMESTAMP` – časová pečiatka (dátum a čas)
  - v tvare `YYYY-MM-DD HH:MM:SS` alebo `YYYY-MM-DD HH:MM:SS.mmmmmm`
  - `CURRENT TIMESTAMP`

4.2.5 Linky na dáta uložené mimo databázy

- `DATALINK`

4.3 Práca s tabuľkou

4.3.1 Vytvorenie tabuľky – príkaz CREATE TABLE

- **Syntax** (časť):

  `CREATE TABLE` *meno_tabuľky* `(`*zoznam_prvkov_tabuľky*`)`

  zoznam_prvkov_tabuľky := *prvok_tabuľky* `[,` *prvok_tabuľky*`]`
  prvok_tabuľky := *definícia_stĺpca* | *definícia_IO_tabuľky*
  definícia_stĺpca := *meno_stĺpca* *dátový_typ* `[`*IO_stĺpca*`]`
  IO_stĺpca := `NOT NULL|DEFAULT|UNIQUE|PRIMARY KEY|FOREIGN KEY|CHECK`

  - `NOT NULL` – stĺpec nesmie mať prázdnu hodnotu (`NULL`)
  - `DEFAULT` – implicitná hodnota stĺpca
  - `UNIQUE` – všetky hodnoty v stĺpci musia byť jedinečné (`NULL` hodnota neprekáža)
  - `PRIMARY KEY` – stĺpec je súčasťou primárneho kľúča tabuľky
  - `FOREIGN KEY` – stĺpec je cudzí kľúč definujúci referenčnú integritu s inou tabuľkou
  - `CHECK` – zadaný logický výraz definuje integritné obmedzenie

- Všetky integritné obmedzenia sa môžu definovať aj na úrovni tabuľky; nutné je to vtedy, keď sú integritné obmedzenia kladené na viacero stĺpcov

- **Príklady:**

  - ```
    CREATE TABLE student1 (
      meno       VARCHAR(10),
      priezvisko VARCHAR(20)
    );
    ```

  - ```
    CREATE TABLE student2 (
      id         INTEGER     NOT NULL,
      meno       VARCHAR(10) NOT NULL,
      priezvisko VARCHAR(20) NOT NULL,
      datum_nar  DATE        NOT NULL,
      PRIMARY KEY (id)
    );
    ```

    Poznámka:

    - V tejto tabuľke je *id* primárnym kľúčom

  - ```
    CREATE TABLE student3 (
      id          INTEGER     NOT NULL,
      meno        VARCHAR(10) NOT NULL,
      priezvisko  VARCHAR(20) NOT NULL,
      datum_nar   DATE        NOT NULL,
      pohlavie    CHAR(4)     NOT NULL CHECK (pohlavie IN ('muz', 'zena')),
      rodne_cislo CHAR(10)    NOT NULL UNIQUE,
      id_mesto    INTEGER,
      PRIMARY KEY (id),
      CONSTRAINT FK_mesto FOREIGN KEY (id_mesto) REFERENCES mesto,
      CONSTRAINT nie_kratke CHECK (
        (LENGTH(meno) > 1) AND (LENGTH(priezvisko) > 1)
      )
    );
    ```

    Poznámky:
    ```

    ◦ pohlavie môže nadobúdať iba hodnoty `'muz'` a `'zena'`
    ◦ aj keď rodné číslo nie je primárnym kľúčom, nemôže nadobúdať dve rovnaké hodnoty
    ◦ obmedzenie `FK_mesto` znamená, že stĺpec *id_mesto* môže nadobúdať iba hodnoty existujúce v tabuľke `mesto` (a tá musí tiež, samozrejme, existovať) alebo prázdnu hodnotu
    ◦ obmedzenie `nie_kratke` znamená, že sa do databázy nedajú vložiť záznamy, ktorých by hodnoty stĺpcov *meno* alebo *priezvisko* nemali aspoň dva znaky

  –
```
CREATE TABLE osoba1 (
  id         INTEGER,
  priezvisko VARCHAR(10) DEFAULT
);
```
    Poznámka:
    ◦ po vložení záznamu iba s *id* sa do stĺpca *meno* uloží hodnota `''` (prázdny reťazec)

  –
```
CREATE TABLE osoba2 (
  id         INTEGER,
  priezvisko VARCHAR(10) DEFAULT 'Kovac'
);
```
    Poznámka:
    ◦ po vložení záznamu iba s *id* sa do stĺpca meno uloží hodnota `'Kovac'`

- tabuľku možno vytvoriť aj pomocou príkazu `SELECT` (viď sekcia 4.4)

```
CREATE TABLE student3_kopia AS (
  SELECT *
  FROM student3
)
DEFINITION ONLY;
```

4.3.2 Úprava štruktúry existujúcej tabuľky – príkaz `ALTER TABLE`

- **Syntax** (časť):

  `ALTER TABLE` *meno_tabuľky zmena*

  Povolená *zmena* je:

  – pridanie stĺpca (syntax)

    `ADD` *definícia_stĺpca*

    Po pridaní stĺpca sa všetkým záznamom v tomto stĺpci priradí hodnota `NULL` (ak nie je určené inak)

  – pridanie IO tabuľky (syntax)

    `ADD CONSTRAINT` *definícia_IO_tabuľky*

  – zrušenie IO tabuľky (syntax)

    `DROP CONSTRAINT` *meno_IO_tabuľky*

- **Príklad:**

- ```
  CREATE TABLE student (
    id         INTEGER NOT NULL,
    meno       VARCHAR(10),
    priezvisko VARCHAR(20),
    datum_nar  DATE,
    id_mesto   INTEGER,
    PRIMARY KEY (id)
  );
  ```
- ```
  CREATE TABLE mesto (
    id    INTEGER NOT NULL,
    nazov VARCHAR(20),
    PRIMARY KEY (id)
  );
  ```

- vytvorenie cudzieho kľúča – väzby medzi dvoma existujúcimi tabuľkami – 3 možnosti vzhľadom na *delete*
  - *s obmedzeným delete*-om (záznam, ktorý je aspoň raz odkazovaný, potom nemožno vymazať)
    - **Syntax**:
      ```
      ALTER TABLE student
      ADD FOREIGN KEY sm (id_mesto) REFERENCES mesto;
      ```
    - alebo alternatívne
      ```
      ALTER TABLE student
      ADD CONSTRAINT sm FOREIGN KEY (id_mesto) REFERENCES mesto;
      ```
      Poznámka:
      Toto je defaultná (implicitná) možnosť z uvedených troch možností.
  - *s kaskádnym delete*-om (záznam, ktorý je aspoň raz odkazovaný, možno vymazať, ale iba vtedy, ak s ním možno vymazať aj všetky záznamy, ktoré naň odkazujú)
    - ```
      ALTER TABLE student
      ADD FOREIGN KEY sm (id_mesto) REFERENCES mesto ON DELETE CASCADE;
      ```
      Poznámka:
      V prípade úspechu môže nastať celá kaskáda *delete*-ov, treba byť preto pri používaní tejto možnosti opatrný.
  - so `SET-NULL` *delete*-om (záznam, ktorý je aspoň raz odkazovaný, možno vymazať, ale iba vtedy, ak všetky hodnoty, ktoré naň odkazujú, možno nastaviť na `NULL` (t.j. nie sú `NOT NULL`))
    - ```
      ALTER TABLE student
      ADD FOREIGN KEY sm (id_mesto) REFERENCES mesto ON DELETE SET NULL;
      ```
  - **Príklad:**
    - ak tabuľka STUDENT obsahuje záznamy `(1,..., 1)`, `(2,..., 2)`, `(3,..., 1)` a tabuľka MESTO o.i. záznam `(1, 'Košice')`, pri kaskádnom delete-e sa pri vymazaní Košíc vymažú aj záznamy `(1, ..., 1)` a `(3, ..., 1)`; v prípade obmedzeného delete-u sú Košice nevymazateľné a v prípade `SET NULL` sa hodnoty stĺpca *id_mesto* v záznamoch `(1,..., 1)` a `(3,..., 1)` zmenia na `NULL`
  - cudzí kľúč môže odkazovať aj na stĺpec, ktorý nie je primárnym kľúčom, ten však musí byť jednoznačný (`UNIQUE`)

- pridanie obmedzenia
  - `ALTER TABLE student ADD CONSTRAINT dk CHECK (YEAR(datum_nar)>=2000);`

- zrušenie obmedzenia
  - `ALTER TABLE student DROP CONSTRAINT dk;`
- pridanie nového stĺpca
  - `ALTER TABLE student ADD COLUMN rocnik INTEGER;`
  - `ALTER TABLE student ADD COLUMN rocnik INTEGER NOT NULL;`
    - nesprávne definovanie – nie je ho čím naplniť, prázdnu hodnotu sme práve zakázali
  - `ALTER TABLE student ADD COLUMN rocnik INTEGER NOT NULL DEFAULT 1;`
    - všetky doterajšie záznamy budú mať v tomto stĺpci hodnotu 1

4.3.3 Zrušenie tabuľky – príkaz `DROP TABLE`

- **Syntax**:

  `DROP TABLE` *meno_tabuľky*

- s vymazaním tabuľky sa vymažú aj všetky jej integritné obmedzenia i tie cudzie kľúče iných tabuliek, ktoré na ňu odkazujú
- **Príklad:**
  - pri vymazaní tabuľky `mesto` sa automaticky vymaže jej primárny kľúč, ale aj cudzí kľúč tabuľky `student`, ktorý na ňu odkazoval

4.3.4 Indexy

- index slúži na efektívnejšie vyhľadávanie záznamov, nemôže však byť súčasťou dopytu – optimalizátor vyhodnocovania dopytu sám rozhodne, či a ktorý index použije
- index možno definovať pre jeden alebo kombináciu stĺpcov (pre také, ktoré sa často vyskytujú v podmienkach dopytov)
- pre jednu tabuľku môže byť definovaný ľubovoľný počet indexov, mená indexov musia byť rôzne
- indexy sú spravidla implementované pomocou binárnych stromov
- vytvorenie indexu
  - **Syntax** (časť):

    `CREATE [UNIQUE] INDEX` *meno_indexu*
    `ON` *meno_tabuľky* `(`*meno_stĺpca* `[ASC|DESC] [,` *meno_stĺpca* `[ASC|DESC]])`
    `[CLUSTER]`

    - `ASC` resp. `DESC` – vzostupné resp. zostupné usporiadanie
    - `UNIQUE` – všetky hodnoty v stĺpci musia byť jedinečné (`NULL` hodnota neprekáža)
    - `CLUSTER` – záznamy sú v implementácii zoradené podľa neho, ležia na disku fyzicky za sebou
      - ▷ iba jeden z indexov môže byť `CLUSTER`
      - ▷ s pribúdajúcimi riadkami sa však správanie takejto štruktúry zhoršuje
    - **Príklad:**

      `CREATE UNIQUE INDEX pm ON student (priezvisko, meno);`
- zrušenie indexu

– **Syntax**:

  DROP INDEX *meno_indexu*

  Poznámka:
  Zrušením indexu sa nezrušia dáta.

4.3.5 Pohľady (views)

- pohľad možno považovať za virtuálnu tabuľku, ktorá nemá vlastné dáta, slúži iba ako filter na dáta v iných tabuľkách; so zmenou dát v pôvodných tabuľkách sa tak automaticky mení aj výsledok pohľadu
- pôvodná tabuľka, možno obsahujúca citlivé dáta, môže byť bežnému používateľovi ako celok neprístupná, pohľadom mu však možno sprístupniť aspoň jej zverejniteľnú časť
- vytvorenie pohľadu
  - **Syntax** (časť):

    CREATE VIEW *meno_pohľadu* [(*meno_stĺpca* [, *meno_stĺpca*]])]
      AS *príkaz_SELECT*

    - príkaz `SELECT` bude popísaný v sekcii 4.4
    - zoznam mien stĺpcov a ich dátových typov je určený príkazom `SELECT`; ak sa mená stĺpcov z definujúceho príkazu `SELECT` nedajú odvodiť, zoznam mien stĺpcov pohľadu je povinný
  - **Príklady:** (využívame tabuľky Student a Mesto definované v predošlom odseku)
    - `CREATE VIEW meno_studenta AS SELECT meno, priezvisko FROM student;`
    - ```
      CREATE VIEW student_mesto (meno, priezvisko, mesto) AS
        SELECT s.meno, s.priezvisko, m.nazov
        FROM student s, mesto m
        WHERE s.id_mesto = m.id;
      ```
- zrušenie pohľadu
  - **Syntax**:

    DROP VIEW *meno_pohľadu*

    Poznámka:
    So zrušením pohľadu sa nerušia tabuľky, na ktorých bol vybudovaný.
  - **Príklad:**
    - `DROP VIEW meno_studenta;`
- pohľady môžu byť v istých prípadoch aj aktualizovateľné

4.3.6 Systémový katalóg

- podáva informácie o metadátach
  - tabuľky: meno, tvorca, dátum vytvorenia, počet stĺpcov, . . . , komentár
  - stĺpce: z ktorej tabuľky, meno, dátový typ, . . . , komentár
  - ďalšie informácie o systéme, každý typ je v osobitnej tabuľke – pohľady, indexy, integritné obmedzenia, . . .
- je zložený opäť z databázových tabuliek (presnejšie z pohľadov) a je prístupný pomocou SQL dopytov
      ```

- katalóg by sa nemal dať aktualizovať bežnými aktualizačnými prostriedkami SQL, robí sa to automaticky
- aktualizovať však možno komentáre k tabuľkám a stĺpcom
  - **Syntax**:
    - pre tabuľky:
      `COMMENT ON TABLE` *meno_tabuľky* `IS` *komentár_v_apostrofoch*
    - pre stĺpce:
      `COMMENT ON COLUMN` *meno_tabuľky.meno_stĺpca* `IS` *komentár_v_apostrofoch*
  - **Príklady:**
    - `COMMENT ON TABLE student IS 'Info o (po)riadnych studentoch';`
    - `COMMENT ON COLUMN student.id IS 'Identifikator';`

4.4 Príkaz SELECT

4.4.1 SELECT-FROM-WHERE-blok (skrátene SELECT)

- základný konštrukt SQL na dopytovanie (t.j. výber dát)
- `SELECT` – klauzula obsahujúca zoznam stĺpcov výslednej tabuľky (v prípade jednoduchších mená, v prípade zložitejších ich konštrukciu), implicitne sú zahrnuté aj ich dátové typy
- `FROM` – klauzula obsahujúca zoznam tabuliek, nad ktorými je dopyt definovaný
- `WHERE` – klauzula obsahujúca podmienku, ktorú musia vybraté dáta spĺňať
  - v prípade, že táto klauzula chýba, považuje sa podmienka za tautológiu
- vzťah k relačnej algebre
  - dopyt tvaru:
    `SELECT` $A_1, \ldots, A_j$
    `FROM` $R_1, \ldots, R_k$
    `WHERE` φ
    zodpovedá v relačnej algebre výrazu

$$(R_1 \times \ldots \times R_k)(\varphi)[A_1, \ldots, A_j]$$

- **Príklady:**
  - odpoveď na dopyt `SELECT * FROM tabuľka` je kópia tabuľky
  - dopyt

```
SELECT priezvisko, nazov
FROM student, mesto
WHERE student.id_mesto = mesto.id
```

    zodpovedá výrazu

$$\textsc{Student} \times \textsc{Mesto}(\textsc{Student}.\mathit{id_mesto} = \textsc{Mesto}.\mathit{id})[\mathit{priezvisko}, \mathit{nazov}]$$

  - v skutočnosti dopyt nemusí byť vyhodnocovaný takýmto často neefektívnym spôsobom
  - na rozdiel od relačnej algebry `SELECT` neeliminuje (automaticky) z výsledku duplicitné riadky
- **Syntax** (zjednodušená):

```
SELECT [DISTINCT]
  *|konštrukcia_stĺpca [AS alias_stĺpca] [, konštrukcia_stĺpca [AS alias_stĺpca]]
FROM
  meno_tabuľky [AS alias_tabuľky] [, meno_tabuľky [AS alias_tabuľky]]
[WHERE
  podmienka]
[ORDER BY
  špecifikácia_usporiadania_podľa_stĺpca [, špecifikácia_usporiadania_podľa_stĺpca]]
```

- **Príklad:**

```
SELECT student.priezvisko, nazov AS mesto, mesto.*
FROM student, mesto
WHERE student.id_mesto = mesto.id
ORDER BY 1 ASC, 2 DESC
```

- konštrukcia stĺpca výslednej tabuľky je v najjednoduchšom a najčastejšom prípade meno niektorého stĺpca niektorej tabuľky z `FROM` klauzuly, a to v tvare

  meno_stĺpca alebo MENO_TABUĽKY.*meno_stĺpca* resp. ALIAS_TABUĽKY.*meno_stĺpca*

- `DISTINCT` – eliminuje z výsledku duplicitné riadky
- Znak `*` znamená, že vyberáme všetky stĺpce zo všetkých tabuliek, MENO_TABUĽKY.`*` znamená, že vyberáme všetky stĺpce z tabuľky MENO_TABUĽKY
- *alias_stĺpca* a *alias_tabuľky* sú lokálne premenovania (v rámci `SELECT`u)
- mená tabuliek sa môžu opakovať, musia však potom byť odlíšené pomocou aliasov
- klauzula `ORDER BY` presahuje rámec relačných štruktúr (tam na poradí riadkov nezáleží). Potom

  ```
  špecifikácia_usporiadania_podľa_stĺpca :=
    [alias_stĺpca|konštrukcia_stĺpca|poradie_stĺpca] [ASC|DESC]
  ```

  – `ASC` resp. `DESC` – vzostupné resp. zostupné usporiadanie, implicitná hodnota je `ASC`

- v podmienke vo `WHERE` možno zadať ľubovoľný booleovský výraz používajúci:
  – logické operátory `AND` (a zároveň), `OR` (alebo), `NOT` (nie)
    ◦ pre priority operátorov platí:

      priorita(`NOT`) > priorita(`AND`) > priorita(`OR`)

  – porovnávania (`=`, `<>`, `!=`, `<`, `>`, `<=`, `>=`)
  – porovnanie `LIKE`
    ◦ umožňuje pracovať s podreťazcami hodnôt stĺpcov typu `CHAR` alebo `VARCHAR`
    ◦ obvykle vo forme
      meno_stĺpca `LIKE` *maska*
      pričom maska môže obsahovať znaky '`%`' (náhrada za reťazec ľubovoľnej – aj nulovej – dĺžky a '`_`' (náhrada za práve jeden znak)
    ◦ v maske možno použiť aj znak '`%`' alebo '`_`' – treba však použiť formu
      meno_stĺpca `LIKE` *maska* `ESCAPE` *iný_znak*
      pričom pred znakom '`%`' alebo '`_`' je v maske napísaný *iný_znak*.

- ▷ napr. vyhodnoteniu reťazca
  `LIKE '%10@%' ESCAPE '@'`
  vyhovuje reťazec `'10%'` alebo `'2310%'`

– `BETWEEN`
  - ◦ výraz (a `BETWEEN` b `AND` c) znamená ((a `>=` b) `AND` (a `<=` c))
  - ◦ a, b, c musia mať kompatibilné typy

– `IN`
  - ◦ výraz (a `IN` (b_1, ..., b_n)) znamená ((a `=` b_1) `OR ... OR` (a `=` b_n))
  - ◦ $a, b_1, \ldots, b_n$ musia mať kompatibilné typy

– `NULL` (prázdna hodnota)
  - ◦ od dvojhodnotovej logiky (TRUE, FALSE) potom musíme prejsť k trojhodnotovej (naviac hodnota UNKNOWN); pre ľubovoľné porovnanie @ platí:
    - ▷ $x@y$ má hodnotu UNKNOWN práve vtedy, keď aspoň jedno z x, y je `NULL`
    - ▷ špeciálne aj porovnanie NULL = NULL má hodnotu UNKNOWN
  - ◦ pravidlá:
    - ▷ pre NOT:

NOT	
FALSE	TRUE
TRUE	FALSE
UNKNOWN	UNKNOWN

    - ▷ pre AND:

AND	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE
UNKNOWN	FALSE	UNKNOWN	UNKNOWN
TRUE	FALSE	UNKNOWN	TRUE

    - ▷ pre OR:

OR	FALSE	UNKNOWN	TRUE
FALSE	FALSE	UNKNOWN	TRUE
UNKNOWN	UNKNOWN	UNKNOWN	TRUE
TRUE	TRUE	TRUE	TRUE

  - ◦ na porovnávanie s prázdnou hodnotou teda nemožno použiť x @ `NULL`; poslúžia na to porovnania x `IS NULL`, ktoré nadobúda hodnotu TRUE práve vtedy, keď x je prázdna hodnota, resp. x `IS NOT NULL`, s ktorým je to presne naopak
    Príklad:
    ```
    SELECT * FROM student WHERE cislo_izby IS NULL
    ```
    - ▷ študenti, ktorí nebývajú na internáte (číslo ich internátnej izby má teda prázdnu hodnotu)

– vnorený `SELECT`
  - ◦ dosadí sa zaň množina, ktorá vznikne ako jeho výsledok
  - ◦ predikáty pre prácu s množinami
    - ▷ `IN`
      Príklad:

```
SELECT * FROM student
WHERE priezvisko IN (SELECT priezvisko FROM ucitel)
```

◦ nájde všetkých študentov, ktorí majú menovcov medzi učiteľmi

▷ `BETWEEN` – ak je použitý vnorený `SELECT`, jeho výsledok tu musí byť jednoznačný
Príklad:

```
SELECT * FROM student
WHERE datum_nar BETWEEN
(SELECT MIN(datum_nar) FROM ucitel)
AND
(SELECT MAX(datum_nar) FROM ucitel)
```

◦ nájde všetkých študentov, ktorí „sú vo veku učiteľov“

▷ porovnávania (`=`, `<>`, `!=`, `<`, `>`, `<=`, `>=`) – ak je použitý vnorený `SELECT`, jeho výsledok tu musí byť jednoznačný
Príklad:

```
SELECT * FROM student
WHERE datum_nar = (SELECT MAX(datum_nar) FROM student)
```

◦ nájde všetkých najstarších študentov

Príklad:

```
SELECT * FROM student
WHERE priezvisko = (SELECT priezvisko FROM ucitel)
```

◦ nájde všetkých študentov, ktorí majú menovcov medzi učiteľmi
◦ zafunguje iba vtedy, ak je učiteľ jediný, inak vyhodí chybu

▷ `ALL` – vykonáva test na zhodu *všetkých* hodnôt
Príklad:

```
SELECT * FROM student
WHERE priezvisko <> ALL(SELECT priezvisko FROM ucitel)
```

◦ nájde všetkých študentov, ktorí nemajú menovcov medzi učiteľmi
◦ porovnanie s `ALL` aplikovaným na prázdnu množinu dáva TRUE

▷ `SOME` (alebo aj `ANY`) – vykonáva test na existenciu *aspoň jednej* hodnoty
Príklad:

```
SELECT * FROM student
WHERE priezvisko = SOME(SELECT priezvisko FROM ucitel)
```

◦ nájde všetkých študentov, ktorí majú menovcov medzi učiteľmi
◦ porovnanie so `SOME` aplikovaným na prázdnu množinu dáva FALSE

▷ `EXISTS`, `NOT EXISTS` – vykonáva test na existenciu hodnoty v tabuľke (vracia pravdivostnú hodnotu)
Príklad:

```
SELECT * FROM mesto
WHERE EXISTS resp. NOT EXISTS (
  SELECT *
  FROM student
  WHERE student.id_mesto = mesto.id
)
```

◦ mestá, v ktorých býva aspoň jeden resp. nebýva žiaden študent
◦ výraz `EXISTS (SELECT ...)` sa vyhodnotí ako pravdivý, ak je výsledok príkazu `SELECT`-u v zátvorkách neprázdna množina, inak je nepravdivá

▷ `SELECT`y môžu byť vnorené aj do viacerých úrovní
Príklad:

```
SELECT MAX(datum_nar)
FROM student
WHERE datum_nar < (
  SELECT MAX(datum_nar)
  FROM student
  WHERE datum_nar <   (
    SELECT MAX(datum_nar)
    FROM student
  )
)
```

◦ dátum narodenia tretieho najstaršieho študenta

- pri konštrukcii stĺpca výslednej tabuľky, ale i v termoch podmienok možno používať:
  - aritmetické operácie $+, -, *, /$ – ak $@ \in \{+, -, *, /\}$, tak x@y má hodnotu `NULL` práve vtedy, keď aspoň jedno z x a y je `NULL`
  - hodnotové výrazy
    - hodnotový výraz vznikne kombináciou aritmetických výrazov, mien stĺpcov, agregácií, hodnôt skalárnych poddopytov (t.j. poddopytu, ktorého výsledkom je tabuľka s jedným riadkom a jedným stĺpcom) a ďalších špeciálnych funkcií, najčastejšie je to však iba meno stĺpca
    - **Príklady:**

▷
```
SELECT
  (SELECT COUNT(*) FROM mesto m WHERE m.nazov <= mesto.nazov),
  nazov
FROM mesto
ORDER BY 1
```

◦ usporiadanie a očíslovanie miest podľa abecedy

▷
```
SELECT MAX(datum_nar)
FROM student
WHERE datum_nar < (
  SELECT MAX(datum_nar)
  FROM student
)
```

◦ dátum narodenia druhého najmladšieho študenta

  - niektoré špeciálne funkcie
    - funkcie na reťazce (typu `CHAR` alebo `VARCHAR`)

▷ `LENGTH` – dĺžka reťazca
  - `LENGTH('Slovensko') = 9`
  - ak je prvý reťazec typu `CHAR(`n`)`, funkcia `LENGTH` vráti n bez ohľadu na jeho (ozajstnú) dĺžku, v prípade `VARCHAR` sú medzery ignorované

▷ `LTRIM`, `RTRIM` – osekáva medzery zľava, sprava
  - `LTRIM(' Slovensko ') = 'Slovensko '`
  - `RTRIM(' Slovensko ') = ' Slovensko'`
  - `LTRIM(RTRIM(' Slovensko ')) = 'Slovensko'`

▷ `SUBSTR` – vyberá z reťazca podreťazec (substring)

Syntax:

`SUBSTR(`*reťazec*`,` *začiatočný_index_podreťazca*`,` *dĺžka_podreťazca*`)`

  - začiatočný index reťazca je číslo znaku reťazca, to sa počíta od 1
  - súčet začiatočného indexu a dĺžky podreťazca zmenšený o 1 nesmie presiahnuť dĺžku vymedzeného miesta pre reťazec

- ◦ `SUBSTR('Slovensko', 2, 4) = 'love'`
- ▷ `CONCAT` alebo `||` – konkatenuje (lepí) dva reťazce na seba
  Syntax:
  `CONCAT(`*reťazec1*`,` *reťazec2*`)`
  alebo
  reťazec1 `||` *reťazec2*
  - ◦ `CONCAT('myš', 'lienka') = 'myšlienka'`
  ak je prvý reťazec typu `CHAR(`n`)`, ale jeho (ozajstná) dĺžka je $m < n$, pri lepení sa medzi reťazce vloží $n - m$ medzier; v prípade `VARCHAR` sa žiadne medzery nevkladajú
- ▷ `UCASE`, `LCASE` – všetky písmená zmení na veľké resp. malé
  - ◦ `UCASE('Slovensko') = 'SLOVENSKO'`
  - ◦ `LCASE('Slovensko') = 'slovensko'`

◦ funkcie na dátumy a/alebo časovú pečiatku
- ▷ `YEAR` – číslo roka z daného dátumu
- ▷ `MONTH` – číslo mesiaca z daného dátumu
- ▷ `DAY` – číslo dňa z daného dátumu
- ▷ `HOUR` – počet hodín z danej časovej pečiatky
- ▷ `MINUTE` – počet minút z danej časovej pečiatky
- ▷ `SECOND` – počet sekúnd z danej časovej pečiatky
- ▷ `MICROSECOND` – počet mikrosekúnd z danej časovej pečiatky
- ▷ `DAYOFWEEK` – poradie dňa v týždni (nedeľa je 1., pondelok 2.,. . .)
- ▷ `DAYOFYEAR` – poradie dňa v roku (napr. 29. 2. je 60. dňom v týždni)
- ▷ `DAYS` – počet dní od 1. 1. 1

◦ ďalšie užitočné funkcie
- ▷ `CAST` – konverzia medzi typmi
  Syntax:
  `CAST (`*výraz* `AS` *dátový_typ*`)`
  - ◦ `CAST (2002 AS CHAR(4)) = '2002'`
  - ◦ `CAST ('2002' AS INTEGER) = 2002`
  - ◦ `CAST (NULL AS` *dátový_typ*`)` – „všeobecný“ `NULL` bude mať navrhnutý dátový typ
- ▷ `CASE` – umožňuje zadať hodnotu v závislosti na hodnote nejakého stĺpca

  Syntax (možnosť 1):
  ```
  CASE
    WHEN podmienka THEN výraz
    ...
    WHEN podmienka THEN výraz
    [ELSE výraz]
  END
  ```
  Syntax (možnosť 2):
  ```
  CASE výraz
    WHEN hodnota THEN výraz
    ...
    WHEN hodnota THEN výraz
    [ELSE výraz]
  END
  ```
  - ◦ v prípade, že `ELSE` chýba, pod príslušným výrazom sa implicitne rozumie `NULL`
  - ◦ podmienky za `WHEN` sa nemusia vylučovať – ak ich platí viac, vezme sa prvá v poradí
  - ◦ ak je splnená podmienka za `WHEN`, vráti hodnotu výrazu za `THEN`; ak nie je splnená žiadna, vráti hodnotu výrazu po `ELSE`
  - ◦ `SELECT`
    `priezvisko,`

```
    CASE
      WHEN SUBSTR(rodne_cislo,3,1) IN ('0','1') THEN 'muz'
      WHEN SUBSTR(rodne_cislo,3,1) IN ('5','6') THEN 'zena'
      ELSE '???'
    END
  FROM student3;
```

- ◦ SELECT

```
    priezvisko,
    CASE SUBSTR(rodne_cislo,3,1)
      WHEN '0' THEN 'muz'
      WHEN '1' THEN 'muz'
      WHEN '5' THEN 'zena'
      WHEN '6' THEN 'zena'
      ELSE '???'
    END
  FROM student3;
```

▷ `COALESCE` alebo `VALUE` – umožňuje nahradiť `NULL` neprázdnym výrazom

Syntax:

```
COALESCE(V_1, ..., V_n)
```

alebo

```
VALUE(V_1,..., V_n)
```

- ◦ vráti prvú hodnotu V_i zo zoznamu, ktorá nie je `NULL`
- ◦ `COALESCE(priezvisko, meno, 'no name')` dáva pre záznam ((*meno*, *priezvisko*)=) `('Albert', 'Einstein')` hodnotu `'Einstein'`, pre `('', 'Madonna')` hodnotu `'Madonna'` a pre `('','')` hodnotu `'no name'`

▷ `NULLIF` – umožňuje vrátiť prázdnu hodnotu

- ◦ zápis `NULLIF(V_1, V_2)` je ekvivalentný zápisu

```
CASE WHEN V_1 = V_2 THEN NULL ELSE V_1
```

- ◦ `NULLIF(tento, iny)` dáva pre záznam (`(tento, iny)` =) `(1,1)` hodnotu `NULL`, pre záznam `(1,2)` hodnotu `1`

- príkazy s agregačnými funkciami `COUNT`, `SUM`, `MAX`, `MIN`, `AVG`
  - agregačné funkcie sa aplikujú na časti tabuľky, ktoré vzniknú skupinovaním podľa hodnôt stĺpcov resp. z nich vytvoreného výrazu uvedeného v klauzule `GROUP BY` – každá skupina tak prispeje do výslednej tabuľky jedným riadkom (ak tam klauzula `GROUP BY` nie je, agregačná funkcia sa aplikuje na jedinú skupinu – celú tabuľku – a výsledok má jediný riadok)
  - významy
    - ◦ `COUNT` – počet hodnôt
    - ◦ `SUM` – suma hodnôt
    - ◦ `MAX` – maximum hodnôt
    - ◦ `MIN` – minimum hodnôt
    - ◦ `AVG` – (aritmetický) priemer hodnôt
  - **Syntax**:

    agregačná_funkcia`([DISTINCT]` *hodnotový_výraz* `[,` *hodnotový_výraz*`]);`

  - `COUNT(*)` počíta všetky riadky včítane duplicít a riadkov obsahujúcich len hodnoty `NULL`; v iných prípadoch sú hodnoty `NULL` ignorované

- `COUNT(∅)` = 0, ale `SUM(∅)` = `NULL`, `AVG(∅)` = `NULL`, `MIN(∅)` = `NULL`, `MAX(∅)` = `NULL`
- `SUM` a `AVG` možno použiť len na číselné typy
- `DISTINCT` nezapočítava do výsledku duplicitné riadky
- **Príklady:**
  - `SELECT COUNT(*) FROM student` – počet študentov
  - `SELECT COUNT(*) FROM student WHERE (priezvisko LIKE 'P%')` – počet študentov s priezviskom začínajúcim na `'P'`
  - `SELECT COUNT(DISTINCT priezvisko) FROM student` – počet rôznych priezvisk študentov
  - `SELECT MAX(datum_nar) FROM student` – dátum narodenia najmladšieho študenta
- klauzuly `GROUP BY` a `HAVING`
  - ak sa v klauzule `SELECT` vyskytuje aj výraz bez agregácií, musia byť uvedený v klauzule `GROUP BY` všetky mená stĺpcov, ktoré sa vyskytujú pri jeho konštrukcii, alebo on sám
  - pri použití klauzuly `HAVING`, ktorá je nasledovaná logickou podmienkou (ako pri `WHERE`), sa z tabuľky vzniknutej aplikáciou na jednotlivé skupiny vyberú len tie riadky, ktoré podmienke vyhovujú
  - **Príklad:**
    - ▷ výpis miest, z ktorých sú aspoň dvaja študenti:

```
SELECT nazov, COUNT(*)
FROM student, mesto
WHERE mesto.id = student.id_mesto
GROUP BY nazov
HAVING COUNT(*) > 1
```

  - podľa definície sú `SUM(∅)`, `AVG(∅)`, `MAX(∅)` a `MIN(∅)` rovné `NULL`, nie 0; v takom prípade možno namiesto výrazu *agregačná_funkcia*`(`*stĺpec*`)` použiť výraz

    `COALESCE(`*agregačná_funkcia*`(`*stĺpec*`), 0)`
  - pri počítaní `AVG` je výsledok rovnakého typu ako hodnoty, z ktorých sa priemer počíta, čo môže byť v prípade celočíselných typov zavádzajúce; v takom prípade treba celočíselné hodnoty konvertovať na desatinné čísla
  - **Príklad:**
    - ▷ v tabuľke so schémou

      HODNOTENIE(*meno*: `INTEGER`, *znamka*: `INTEGER`)

      nech sú hodnoty

      `('Jano',1), ('Mišo',2), ('Fero',3), ('Ďuro',4);`

      Potom dopyt

      `SELECT AVG(znamka) FROM hodnotenie`

      dáva celočíselnú hodnotu 2, ale správna hodnota je 2.5; dosiahneme ju dopytom

      `SELECT AVG(CAST (znamka AS DECIMAL(5,2))) FROM hodnotenie`

4.4.4 Spojenia tabuliek

- v klasickom `SELECT` príkaze
  `SELECT` $A_1, \ldots, A_j$
  `FROM` $R_1, \ldots, R_k$
  `WHERE` φ
  je φ spravidla konjunkciou podmienok dvoch typov – jednak sú tu väzobné podmienky medzi zúčastnenými tabuľkami $R_1, \ldots, R_k$ a jednak porovnania s konštantami. SQL umožňuje tieto dva typy podmienok oddeliť tým, že väzobné podmienky sa definujú už v klauzule `FROM`, a

to (napríklad) v takejto forme ekvivalentnej s predošlou:
SELECT $A_1, \ldots, A_j$
FROM R_1
JOIN R_2 ON *väzobné_podmienky_medzi_*R_1*_a_*R_2
⋮
JOIN R_k ON *väzobné_podmienky_medzi_*R_i*_a_*R_k
WHERE *ostatné_podmienky*

- **Príklad:**
  - dopyt na všetky informácie o všetkých študentoch s menom Jozef

    ```
    SELECT *
    FROM student s, mesto m, skupina g, izba i, typ_izby t
    WHERE s.id_mesto = m.id
      AND s.id_skupina = g.id
      AND s.id_izba = i.id
      AND i.id_typ = t.id
      AND s.meno = 'Jozef'
    ```

    možno ekvivalentne nahradiť dopytom

    ```
    SELECT *
    FROM student s
      JOIN mesto m ON s.id_mesto = m.id
      JOIN skupina g ON s.id_skupina = g.id
      JOIN izba i ON s.id_izba = i.id
      JOIN typ_izby t ON i.id_typ = t.id
    WHERE s.meno = 'Jozef'
    ```

- možné sú aj kombinácie týchto prístupov – časť väzobných podmienok je v klauzule FROM pod JOIN-mi, zvyšok je vo WHERE klauzule
- **Príklad:**
  - ďalším ekvivalentným zápisom k predošlým dvom je napr. dopyt

    ```
    SELECT *
    FROM student s
      JOIN mesto m ON s.id_mesto = m.id
      JOIN skupina g ON s.id_skupina = g.id
      JOIN izba i ON s.id_izba = i.id, typ_izby t
    WHERE i.id_typ = t.id
      AND s.meno = 'Jozef'
    ```

- pri použití obyčajného spojenia sa (podľa definície) eliminujú riadky, v ktorých je hodnota niektorého prepájacieho stĺpca prázdna. Niekedy je však vhodné mať vo výsledku aj riadky s prázdnymi hodnotami ako zdôraznený signál toho, že na nich nemožno nič naviazať; v takomto prípade namiesto obyčajného spojenia použijeme tzv. *vonkajšie spojenie*
  - typy vonkajšieho spojenia
    - ľavé vonkajšie spojenie – R_1 LEFT OUTER JOIN R_2
      - ▷ do výsledku sa okrem záznamov, za ktoré „môže" obyčajný JOIN, pridajú aj tie záznamy pochádzajúce z R_1, ktoré nemajú partnerov v R_2, pričom miesta stĺpcov z R_2 sú doplnené prázdnymi hodnotami
    - pravé vonkajšie spojenie – R_1 RIGHT OUTER JOIN R_2
      - ▷ úlohy R_1 a R_2 sa vymenia

  ◦ úplné vonkajšie spojenie – R_1 `FULL OUTER JOIN` R_2
    ▷ výsledkom je zjednotenie ľavého a pravého vonkajšieho spojenia

- na zdôraznenie odlišnosti od vonkajších spojení sa obyčajnému spojeniu hovorí aj *vnútorné spojenie* (a namiesto `JOIN` možno použiť `INNER JOIN`)

- **Príklad:**

  – predpokladajme, že tabuľka

  STUDENT(*id*:`INTEGER`, *meno*:`VARCHAR(20)`, *id_mesto*:`INTEGER`)

  obsahuje záznamy

  `(1, 'Ján Hraško', 1), (2, 'Ali Baba', NULL)`

  a tabuľka

  MESTO(*id*:`INTEGER`, *nazov*:`VARCHAR(20)`)

  záznamy

  `(1, 'Hrašovík'), (2, 'Šalgovík')`

Dopyt	Výsledok
`SELECT * FROM student s JOIN mesto m ON s.id_mesto = m.id`	1 | Ján Hraško | 1 | Hrašovík
`SELECT * FROM student s LEFT OUTER JOIN mesto m ON s.id_mesto = m.id`	1 | Ján Hraško | 1 | Hrašovík 2 | Ali Baba | NULL | NULL
`SELECT * FROM student s RIGHT OUTER JOIN mesto m ON s.id_mesto = m.id`	1 | Ján Hraško | 1 | Hrašovík NULL | NULL | 2 | Šalgovík
`SELECT * FROM student s FULL OUTER JOIN mesto m ON s.id_mesto = m.id`	1 | Ján Hraško | 1 | Hrašovík 2 | Ali Baba | NULL | NULL NULL | NULL | 2 | Šalgovík

4.5 Zložitejšie nadstavby nad príkazom SELECT

4.5.1 Množinové operácie (`INTERSECT`, `UNION`, `EXCEPT`)

- aby bolo možné množinové uperácie urobiť, musia byť obe tabuľky kompatibilné – musia mať rovnaký počet riadkov a rovnaké datové typy

- význam
  – `INTERSECT` – prienik
  – `UNION` – zjednotenie
  – `EXCEPT`– rozdiel

- **Syntax** (zjednodušená):

  SELECT_príkaz_1
  množinová_operácia `[ALL]`
  SELECT_príkaz_2
  `[ORDER BY` *špecifikácia_triedenia*`]`

- pri použití `ALL` sa neodstránia duplicity (zato je však operácia `UNION ALL` rýchlejšia než `UNION`)
- v prípade, že nesedia názvy stĺpcov v oboch `SELECT`-och, v `ORDER BY` možno použiť len poradie stĺpcov
- **Príklady:**
  - ```
    SELECT meno, priezvisko FROM student
    UNION ALL
    SELECT meno, priezvisko FROM ucitel
    ORDER BY priezvisko
    ```
    - zoznam všetkých študentov i učiteľov
  - ```
    SELECT meno FROM student
    INTERSECT
    SELECT priezvisko FROM student
    ORDER BY 1
    ```
    - zoznam prievisk študentov, ktoré sa zhodujú s krstným menom niektorého študenta, ale každé iba raz
    - v `ORDER BY` klauzule môže byť iba číslo

4.5.2 Pomocné tabuľky – klauzula `WITH`

- ak na získanie výslednej tabuľky nestačia doterajšie prostriedky, možno pomocou klauzuly `WITH` vytvoriť pomocné tabuľky a tie potom použiť pri konštrukcii výsledku
- **Syntax**:

  `WITH` *definícia_pomocnej_tabuľky* `[,` *definícia_pomocnej_tabuľky*`]` *SELECT_príkaz*

  definícia_pomocnej_tabuľky `:=`
    názov_pomocnej_tabuľky`(`
      názov_stĺpca_pomocnej_tabuľky `[,` *názov_stĺpca_pomocnej_tabuľky*`]`
    `)`
    `AS (`*SELECT_príkaz*`)`

- **Príklad:**
  - ```
    WITH
      mozne (id, kredit) AS (
        SELECT s.id, SUM(kredit)
        FROM student s, zapisane z, predmet p
        WHERE s.id = z.id_student AND p.id = z.id_predmet
        GROUP BY s.id
      ),
      ziskane (id, kredit) AS (
        SELECT s.id, SUM(kredit)
        FROM student s, zapisane z, predmet p
        WHERE s.id = z.id_student
          AND p.id = z.id_predmet
          AND znamka IS NOT NULL
        GROUP BY s.id
      )
    SELECT s.id, s.meno, s.priezvisko, m.kredit, z.kredit
    FROM sk.student s, mozne m, ziskane z
    WHERE m.id = s.id AND z.id = s.id
    ORDER BY 3, 2;
    ```
    ```

  - najprv sme si vo `WITH` klauzule vytvorili pomocnú tabuľku `mozne`, kde sú id študentov so sumami kreditov, ktoré mohli získať, a pomocnú tabuľku `ziskane`, kde sú id študentov so sumami nimi naozaj získaných kreditov; tieto pomocné tabuľky sme potom zaangažovali do výsledného `SELECT`-u

- slabšou alternatívou je definovanie tabuľky priamo vo `FROM` klauzule
  - **Syntax** (časť):

```
SELECT ...
FROM (SELECT_príkaz)
AS meno_pomocnej_tabuľky(
  meno_stĺpca_pomocnej_tabuľky, [meno_stĺpca_pomocnej_tabuľky]
)
```

  - **Príklad:**

```
SELECT AVG(suma)
FROM
(
  SELECT t.kod, SUM(t.poplatok)
  FROM
    student s
   JOIN izba i ON s.id_izba = i.id
   JOIN typ_izby t ON i.id_typ = t.id
  GROUP BY t.kod
)
AS zaplatene(kod, suma)
```

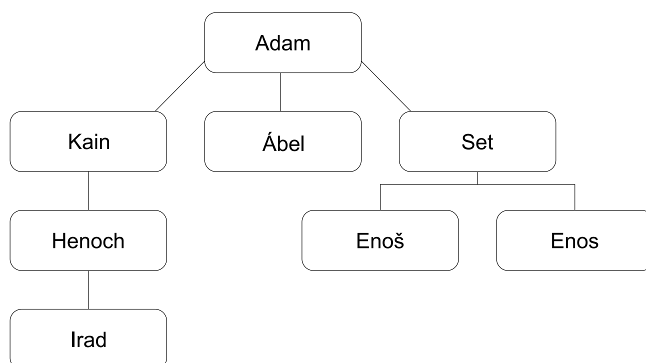
    - v tabuľke `zaplatene` sa vypočíta, koľko bývajúci študenti zaplatili za ten-ktorý typ izby, vo vonkajšom príkaze sa zistí priemerná hodnota týchto čísel

4.5.3 Tranzitívny uzáver

- špeciálnou možnosťou SQL vo verzii DB2 je vyjadrenie tranzitívneho uzáveru pomocou rekurzie
- **Príklad:**
  - Predpokladajme, že máme rodokmeň reprezentovaný tabuľkou

    ČLEN(*id*:`INTEGER`, *id_otec*: `INTEGER`, *meno*: `VARCHAR(10)`)

    (s primárnym kľúčom *id*), v ktorej *id_otec* je cudzím kľúčom do tej istej tabuľky a znamená id otca – priameho predchodcu cez práve jednu generáciu; „praotec“ v našej databáze bude mať v stĺpci *id_otec* prázdnu hodnotu



id	id_otec	meno
1	NULL	Adam
2	1	Kain
3	1	Abel
4	1	Set[1]
5	2	Henoch
6	5	Irad
7	4	Enoš
8	4	Enos

[1] Matematickí informatici s radosťou žartujú, že Set je patrónom teórie množín.

Definícia tohto cudzieho kľúča bude nasledovná:

```
ALTER TABLE clen ADD FOREIGN KEY os(id_otec) REFERENCES clen
```

– **Príklady:**
  - zoznam otcov a synov
    ```
    SELECT o.meno, s.meno FROM clen o, clen s WHERE s.id_otec = o.id
    ```
  - zoznam dedov a vnukov
    ```
    SELECT o.meno, v.meno
    FROM clen d, clen x, clen v
    WHERE v.id_otec = x.id AND x.id = d.id
    ```

– pre ľubovoľný (pevný) rozdiel generácií vieme napísať `SELECT`, problém však nastáva pre prípad, že chceme vypísať všetkých potomkov.

– na to, aby sme získali tabuľku s dvojicami predkov a potomkov cez ľubovoľný počet generácií, vyrobíme pomocnú tabuľku potomkovia, ktorá bude zložená z dvoch častí spojených `UNION`-om. V prvej časti (prvý krok rekurzie) vlastne len skopírujeme tabuľku Člen (až na riadok praotca), ktorá bude znamenať rozdiel jednej generácie, v druhej časti, kde je skrytá samotná rekurzia, na tabuľku potomkovia opakovane naväzujeme tabuľku Člen – každý ďalší medzivýsledok bude mať o jednu generáciu viac (napodobňujeme tak proces spomínaný v časti 3.2)

– riešenie v SQL potom bude nasledovné:

```
WITH
  potomkovia (id, id_potomok) AS (
    SELECT id_otec, id                          }
    FROM clen                                   } 1. krok rekurzie
    WHERE id_otec IS NOT NULL                   }
    UNION ALL
    SELECT otec.id, dieta.id                    }
    FROM potomkovia otec, clen dieta            } 2. krok rekurzie
    WHERE otec.id_potomok = dieta.id_otec       }
  )
SELECT predok.id, predok.meno, potomok.id, potomok.meno
FROM potomkovia p, clen predok, clen potomok
WHERE predok.id = p.id AND potomok.id = p.id_potomok
ORDER BY 1, 2
```

– jednotlivé iterácie potom budú vyzerať nasledovne:
  - `potomkovia`$_0$ (po 1. kroku indukcie)

id	id_potomok
1	2
1	3
1	4
2	5
5	6
4	7
4	8

  - `potomkovia`$_1$ (po 1. vykonaní 2. kroku indukcie)

id	id_potomok
riadky tabuľky `potomkovia`$_0$	
1	5
1	7
1	8
2	6

  - `potomkovia`$_2$ (po 2. vykonaní 2. kroku indukcie)

id	id_potomok
riadky tabuľky `potomkovia`$_0$	
riadky tabuľky `potomkovia`$_1$	
1	6

  ◦ `potomkovia`$_3$ = `potomkovia` (nepribudlo nič nové)

id	id_potomok
1	2
1	3
1	4
2	5
5	6
4	7
4	8
...	

pokr. tabuľky	
1	5
1	7
1	8
2	6
1	6
iterovanie skončí	

- **Príklad:**
  Nasledujúce príkazy vypíšu človeka a jeho generáciu:

```
WITH generacia(id, cislo) AS (
  SELECT id, 1
  FROM clen
  WHERE id_otec IS NULL
  UNION ALL
  SELECT c.id, g.cislo+1
  FROM generacia g, clen c
  WHERE c.id_otec = g.id
)
SELECT clen meno, generacia cislo
FROM generacia, clen
WHERE generacia.id = clen.id
```

4.6 Aktualizácia dát

Medzi aktualizačné príkazy v SQL radíme príkazy `INSERT`, `DELETE` a `UPDATE`.

4.6.1 `INSERT` – vkladanie nových záznamov

- **Syntax**:
  - `INSERT INTO` *meno_tabuľky* `[(`*meno_stĺpca* `[,` *meno_stĺpca*`])]`
    `VALUES` *hodnota_záznamu* `[,` *hodnota_záznamu*`]`

    hodnota_záznamu `:= (`*hodnota_stĺpca* `[,` *hodnota_stĺpca*`])`
  - `INSERT INTO` *meno_tabuľky* `[(`*meno_stĺpca* `[,` *meno_stĺpca*`])]` *SELECT_príkaz*
- počet, poradie a dátové typy stĺpcov musia korešpondovať s príslušnými hodnotami stĺpcov resp. so stĺpcami príkazu `SELECT`
- v prípade, že mená stĺpcov nie sú uvedené, predpokladá sa, že sú všetky a v takom poradí, v akom boli definované pri vytváraní tabuľky
- stĺpce, ktoré nie sú uvedené, sú vyplnené automaticky, a to podľa definície tabuľky (spravidla prázdnou hodnotou, tu sa však vyžaduje, aby príslušný stĺpec prázdne hodnoty pripúšťal (t.j. aby nebol `NOT NULL`))
- **Príklady:**
  - majme ďalšiu tabuľku OSOBA(_*id*_: `INTEGER`, *meno*: `VARCHAR(30)`)
    ◦ príkaz

```
INSERT INTO osoba
VALUES (1, 'Ján Hraško')
```

vloží do tabuľky `osoba` záznam `(1, 'Ján Hraško')` (ak tam ešte záznam s takou hodnotou primárneho kľúča nie je)

- príkaz

```
INSERT INTO osoba
VALUES
  (1, 'Ján Hraško'),
  (2, 'Ján Polienko')
```

vloží do tabuľky `osoba` záznamy `(1, 'Ján Hraško')` a `(2, 'Ján Polienko')` (ak tam ešte záznamy s takými hodnotami primárneho kľúča nie sú)

- príkaz

```
INSERT INTO osoba (id)
VALUES (1)
```

vloží do tabuľky `osoba` záznam `(1, NULL)` (ak tam ešte záznam s takou hodnotou primárneho kľúča nie je)

- príkaz

```
INSERT INTO osoba
  SELECT id, meno || ' ' || priezvisko
  FROM student
```

vloží do tabuľky `osoba` záznamy odvodené zo záznamov z tabuľky ŠTUDENT (ak tam ešte záznamy s takými hodnotami primárneho kľúča nie sú)

4.6.2 `DELETE` – mazanie existujúcich záznamov

- **Syntax**:
  `DELETE FROM` *meno_tabuľky* `[WHERE` *podmienka*`]`
  - vymažú sa všetky záznamy, ktoré spĺňajú uvedenú podmienku (ktorá môže byť aj zložená)
  - ak klauzula `WHERE` nie je uvedená, chápe sa ako pravdivá, t.j. vymažú sa všetky záznamy – celý obsah tabuľky
- **Príklady:**
  - `DELETE FROM osoba WHERE id=1`
    - vymaže sa jeden záznam
  - `DELETE FROM osoba WHERE meno LIKE 'Ján %'`
    - vymažú sa všetky záznamy, ktorých hodnota atribútu meno začína na `'Ján'`
  - `DELETE FROM osoba WHERE id IN (SELECT id FROM student)`
    - vymažú sa všetci študenti

4.6.3 `UPDATE` – zmena hodnôt existujúcich záznamov

- **Syntax**:

  `UPDATE` *meno_tabuľky*
  `SET` *nastavenie_hodnoty_stĺpca* `[,` *nastavenie_hodnoty_stĺpca*`]` `[WHERE` *podmienka*`]`

  nastavenie_hodnoty_stĺpca `:= meno_stĺpca =` *hodnota*

- prepíšu sa všetky záznamy, ktoré spĺňajú uvedenú podmienku (ktorá môže byť aj zložená)
- ak klauzula `WHERE` nie je uvedená, chápe sa ako pravdivá, t.j. vymažú sa všetky záznamy – celý obsah tabuľky

- hodnota nemusí byť len konštanta, môžu v nej figurovať aj iné stĺpce tabuľky, ba aj ten istý
- **Príklady:**
  - `UPDATE student SET meno='Jozef' WHERE id=1`
    - upraví sa meno študenta s id 1
  - `UPDATE student SET meno='Jozef', priezvisko='Mrkvička' WHERE id=1`
    - upraví sa meno a priezvisko študenta s id 1
  - `UPDATE student SET meno='Jozef' WHERE meno='Jožo'`
    - upraví sa meno všetkých Jožov na Jozef
  - ```
    UPDATE osoba
     SET meno='študent '|| meno WHERE id IN (SELECT id FROM student)
    ```
    - upravia sa mená všetkých osôb, ktoré sú študentmi, tak, že sa pred nich predpíše slovo `'študent'`
- pri vhodnej definícii možno upravovať aj hodnoty v „pohľade“ (VIEW-e), upraví sa tým pôvodná tabuľka

4.7 Triggery

- pri zmene nejakých hodnôt v niektorej tabuľke môže byť vhodné či nutné meniť aj stav ďalších hodnôt v inej alebo aj v tej istej tabuľke; je preto vhodné vytvoriť *trigger* – prostriedok, ktorý túto súslednosť zabezpečí
- **Syntax** (časť):

```
CREATE TRIGGER meno_triggeru
{AFTER|ON CASCADE BEFORE}
{INSERT|DELETE|UPDATE[(zoznam_stĺpcov)} ON meno_tabuľky
[REFERENCING
  [OLD AS zvolené_meno_starého_záznamu]
  [NEW AS zvolené_meno_nového_záznamu]
]
FOR EACH {ROW|STATEMENT} MODE DB2SQL
[WHEN (podmienka)]
akcia
```

- trigger sa môže spustiť pred (`ON CASCADE BEFORE`) alebo po (`AFTER`) zmene a vyvolá ho iba príslušná operácia
- ak zoznam stĺpcov pri `UPDATE` prázdny, trigger sa týka zmien všetkých stĺpcov tabuľky
- pri nastavovaní hodnôt možno použiť aj nové alebo staré (prepisované) hodnoty, čo sa uvedie v sekcii `REFERENCING OLD` resp. `NEW`
- pod akciou sa najčastejšie rozumie nastavenie nejakej hodnoty
- akcia sa vykoná iba po splnení podmienky; ak podmienka nie je uvedená, akcia sa vykoná vždy
- **Príklady:**
```

–
```
CREATE TRIGGER student_plus/student_minus
AFTER INSERT/DELETE ON student
FOR EACH ROW MODE DB2SQL
UPDATE statistika
  SET hodnota = hodnota +/- 1
  WHERE parameter='pocet studentov'
```

  ◦ po pridaní nového resp. zmazaní ľubovoľného existujúceho záznamu sa v tabuľke Štatistika(*parameter*, *hodnota*) automaticky o 1 zväčší resp. zmenší hodnota x v zázname `('pocet studentov', x)`

–
```
CREATE TRIGGER student_zmena
AFTER UPDATE ON student
REFERENCING OLD AS o
FOR EACH ROW MODE DB2SQL
INSERT INTO student_historia (id, meno, priezvisko, datum_do) VALUES
  (o.id, o.meno, o.priezvisko, CURRENT TIME)
```

  ◦ po zmene údajov o študentovi sa do tabuľky histórie zmien vloží nový záznam so starými hodnotami študenta a aktuálnou pečiatkou

4.8 Ochrana dát

- ochrana dát – príkaz `GRANT`

  – keďže s databázou spravidla pracuje viacero osôb rôznej kategórie, správca databázy by mal určiť, kto z nich má mať aké práva na čítanie a lebo modifikáciu tabuliek či pohľadov

  – **Syntax** (časť):

    `GRANT` {`ALL`|*zoznam_privilégií*}
    `ON` *objekt*
    `TO` {`PUBLIC`|*zoznam_používateľov*}
    `WITH GRANT OPTION`

    ◦ *zoznam_privilégií* := *privilégium* `[,` *privilégium*`]`
      privilégium := `SELECT|DELETE|UPDATE[`*zoznam_atribútov*`]|...`
    ◦ *zoznam_používateľov* := *používateľ* `[,` *používateľ*`]`
      používateľ := `GROUP` *meno_skupiny*|`USER` *meno_používateľa*
    ◦ *objekt* je meno tabuľky alebo pohľadu
    ◦ klauzula `WITH GRANT OPTION` umožňuje používateľom zo zoznamu udeľovať právo ďalej; prvým udeľovateľom je používateľ, ktorý objekt vyrobil

  – **Príklady:**

    ◦ `GRANT SELECT ON pohlad_osoba TO PUBLIC`
      ▷ čítať pohľad Pohlad_Osoba môže každý
    ◦ `GRANT UPDATE (izba) ON student TO USER riaditel_internatu`
      ▷ riaditeľ internátu má právo meniť číslo izby každého študenta
    ◦ `GRANT DELETE ON osoba TO GROUP mafia`
      ▷ členovia skupiny *mafia* môžu z tabuľky odstraňovať jednotlivé osoby
    ◦ `GRANT ALL ON osoba TO USER Fero WITH GRANT OPTION`
      ▷ Fero má plnú moc nad tabuľkou `osoba`, môže ju podať aj ďalej

4.9 Integrácia SQL do programovacieho jazyka

- SQL príkazy môžu byť súčasťou iného programovacieho jazyka (často C), hovorí sa o tzv. *embedded SQL*
- do zdrojového kódu programu sa vložia príkazy SQL s predtextom `EXEC SQL` a vo fáze predkompilácie sa automaticky nahradia sériou príkazov v príslušnom jazyku; samotná kompilácia už potom používa nahradené riadky
- pre `SELECT` možno deklarovať tzv. kurzor, po jeho otvorení potom možno v cykle pracovať s jednotlivými záznamami – vkladať ich do premenných pomocou príkazu `FETCH`, ktorý umiestni kurzor na ďalšiu odpoveď záznamu; v novších verziách SQL možno kurzor presunúť aj na iné miesta
- pred premenné použité v `EXEC SQL` riadkoch potom treba písať dvojbodku
- **Príklad:**
  - ```
    EXEC SQL DECLARE CURSOR studentiNaPismeno
    FOR SELECT meno, priezvisko
       FROM student
       WHERE SUBSTR(priezvisko,1,1)=:pismeno
       ORDER BY 2, 1
    EXEC SQL OPEN studentiNaPismeno
    ```
    a v cykle:
    ```
    EXEC SQL
      FETCH studentiNaPismeno INTO (:menoStudenta, :priezviskoStudenta)
    ```
    - po deklarácii a otvorení kurzoru sa postupne prečítajú usporiadané mená a priezviská študentov začínajúcich na dané písmeno a výsledky sa ukladajú do príslušných premenných
- spätná väzba sa po vyvolaní SQL príkazu robí pomocou premennej `SQLCODE`; po úspešnom vykonaní príkazu je v tejto premennej 0, hodnota 100 indikuje úspešnosť operácie s upozornením, že neexistujú dáta, záporné čísla znamenajú výskyt chyby
    ```