

Základy databázových systémov

Rastislav Bencel

Document Information

User: xdrozdj (7e36580a-7926-4593-8c16-6a589fd6cabd)

Title: Základy databázových systémov

Document ID: 7a12671a-4f58-4de7-9da6-93b2cc5be777

Generated at: 2024-12-03T08:24:17.326613+00:00

License Terms

LICENSE AND ACCESS. This license grants a non-transferable, non-exclusive, limited license to access and use specified documents in compliance with the platform's terms.

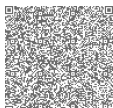
This document was distributed using open-source project EvilFlowersCatalog part of ELVIRA project.

Všetky práva vyhradené. Nijaká časť textu nesmie byť použitá na ďalšie šírenie akoukoľvek formou bez predchádzajúceho súhlasu autorov alebo vydavateľstva.

Ing. Rastislav. Bencel, PhD.

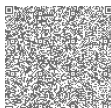
Toto skriptum vzniklo vďaka podpore projektu KEGA 025STU-4/2022.

ISBN 978-80-227-5249-7



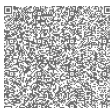
Obsah

Zoznam obrázkov	1
Zoznam tabuliek.....	2
Zoznam skratiek.....	3
Predhovor.....	4
1 Úvod do databázových systémov.....	5
1.1 Architektúra databázových aplikácií	5
2 Dátový model.....	8
2.1 Relačný dátový model	9
3 SQL	12
3.1 História SQL	12
3.2 Data Definition Language	13
3.3 Integritné obmedzenia	17
3.4 Základná štruktúra SQL dopytu	20
3.5 Modifikácia záznamov	21
3.6 Usporiadanie výsledkov	24
3.7 Jednoznačnosť atribútov a aliasy	25
3.8 Agregácie	26
3.9 Práca s viacerými tabuľkami.....	29
3.10 Vnorené dopyty.....	39
3.11 Window Functions.....	41
3.12 View	43
3.13 Funkcie a procedúry	47
3.14 Triggers	50
4 Fyzické úložisko	55
4.1 Načítavanie a ukladanie dát v DBMS.....	56
5 Indexy.....	59
5.1 SQL syntax	60
5.2 Hash index	61
5.3 B+tree index	62
5.4 Bitmap index.....	65
6 Transakcie	67
6.1 Sériovateľnosť	68
6.2 Isolation (izolácia v transakciach).....	73
6.3 Manažment buffer poolu	75
6.4 Atomicity a Durability.....	78
6.5 Príklady na úrovne izolácie.....	80
Zoznam použitej literatúry.....	87



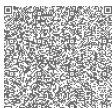
Zoznam obrázkov

Obr. 1. 2-úrovňová a 3-úrovňová architektúra aplikácií využívajúcich databázu.....	6
Obr. 2. Ukážka komunikácie medzi klientom a serverom v 3-úrovňovej architektúre	7
Obr. 3. Prepojenie v relačnom dátovom modeli	11
Obr. 4. Rozdelenie SQL príkazov	13
Obr. 5. Príklad s porušením referenčnej integrity	19
Obr. 6. Typy spojenia tabuliek	31
Obr. 7. Znázornenie vonkajšieho a vnútorného dopytu	40
Obr. 8. Hierarchia pamäti z pohľadu kapacity a rýchlosti.....	55
Obr. 9. Načítanie a ukladanie dát v rámci DBMS	58
Obr. 10. Princíp fungovania statického hash indexu	62
Obr. 11. Štruktúra B+tree	63
Obr. 12. Štruktúra vnútorných uzlov B+tree.....	63
Obr. 13. Štruktúra listov v B+tree	63
Obr. 14. Ukážka indexu pomocou b+tree	64
Obr. 15 Ukážka bitmap indexu.....	66
Obr. 16. Rozvrh S ₁ – sériový Obr. 17. Rozvrh S ₂ - seriovateľný	71
Obr. 18. Rozvrh S ₃ – neseriovateľný– generujúci nesprávny výsledok	72
Obr. 19. Množiny typov rozvrhov a ich prekrývanie.....	72
Obr. 20. Ukážka Snapshot isolation pre dve transakcie	75
Obr. 21. Ilustrovanie problému s rozhodnutím kedy má byť stránka presunutá na fyzické úložisko	76
Obr. 22. Príklad na politiky FORCE a STEAL manažmentu Buffer Pool	76
Obr. 23. Porovnanie manažmentu buffer pool z pohľadu výkonnosti systému a rýchlosti obnovy.....	78
Obr. 24. Ukážka logovania založeného na princípe Write-Ahead Logging.....	79



Zoznam tabuliek

Tabuľka 1. Ukážka relácie/tabuľky v relačnom dátovom modeli	10
Tabuľka 2 Správanie hodnoty NULL pre operácie negácie, logického súčinu a súčtu	14
Tabuľka 3. Ukážka dát tabuľky players	32
Tabuľka 4. Ukážka dát tabuľky teams	32
Tabuľka 5. Ukážka dát tabuľky player_seasons	32
Tabuľka 6. Výsledok dopytu pre INNER JOIN pre nájdenie všetkých hráčov, ktorý odohrali nejakú sezónu	34
Tabuľka 7. Výsledok dopytu pre INNER JOIN nad tromi tabuľkami (players, player_seasons, teams)	35
Tabuľka 8. Výsledok dopytu pre LEFT JOIN pre nájdenie všetkých hráčov spolu s ich sezónami	37
Tabuľka 9. Prehľadová tabuľka popisu vlastností pre procedúry a funkcie	47
Tabuľka 10. Orientačné časy prístupu pre úrovne pamäti	56
Tabuľka 11 Zoznam príkazov pre ukážku izolácie Read committed	81
Tabuľka 12. Zoznam príkazov pre ukážku izolácie Repeatable read	83
Tabuľka 13. Zoznam príkazov pre ukážku izolácie Repeatable Read s porušením sériovateľnosti	85



Zoznam skratiek

API - Application Programming Interface

ARIES - Algorithms for Recovery and Isolation Exploiting Semantics

DBMS – Database-Management System

DCL – Data Control Language

DQL – Data Query Language

DDL – Data Definition Language

DML – Data Manipulation Language

HDD – Hard Disk Drive

NoSQL - Non-SQL

REST - REpresentational State Transfer

Sequel - Structured English Query Language

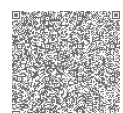
SOAP - Simple Object Access Protocol

SQL - Structured Query Language

SSD – Solid State Drive

WAL – Write-Ahead Logging

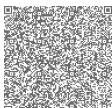
WF – Window Function



Predhovor

Tieto skriptá sú primárne určené ako podporná literatúra pre predmet Databázové systémy a okrajovo je ich možné použiť aj v rámci predmetu Pokročilé databázové technológie, ktoré sa vyučujú na Fakulte informatiky a informačných technológií Slovenskej technickej univerzity v Bratislave. Čitateľ sa oboznámi so základmi databázových systémov so zameraním na relačné databázy. Veľká časť obsahu je venovaná problematike SQL jazyka, okrajovo sú rozobraté aj oblasti, ktoré sú vykonávané na pozadí databázového systému. Tento materiál nepokrýva všetky oblasti, ktoré sú vyučované v rámci predmetu Databázové systémy. Oblasť návrhu relačnej databázy a objektovo relačného mapovania (ORM) nie sú vôbec v texte spomínané.

V úvode je priblížené, čo je databázový systém a architektúra aplikácií využívajúcich databázu. Druhá kapitola sa venuje dátový modelom a stručne opisuje relačný dátový model. V tretej kapitole je popísaný SQL jazyk. V štvrtej kapitole je popísané fyzické úložisko a ako sú načítavané dáta v databázovom systéme. V piatej kapitole sú popísané indexy a záverečná kapitola sa venuje transakciám, kde sú popísané požiadavky na transakčný systém (ACID vlastnosti).



1 Úvod do databázových systémov

Database-managment systém (DBMS) predstavuje kolekciu vzájomne súvisiacich dát a súbor programov, ktorých úlohou je pristupovať k jednotlivým dátam. Samotné dáta sú označované ako **databáza**. Pojem databáza nie je viazaný na konkrétny databázový systém, ale je ho možné používať pre čokoľvek, čo uchováva informácie napr. textový súbor. Úlohou databázového systému je primárne ukladanie a získavanie údajov z databázy efektívne a pohodlne.

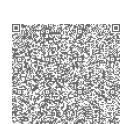
Databázové systémy sú navrhnuté tak, aby umožňovali spravovať veľké množstvo dát a používateľ nemusel riešiť rôzne mechanizmy, napríklad akú štruktúru reprezentácie dát je vhodné použiť, ako obnoviť dáta v prípade výpadku, ako uskutočniť transakcie alebo ako riešiť manažment prístupu. Pod pojmom používateľ databázového systému rozumieme, či už človeka alebo aplikáciu, ktorá využíva databázový systém pre prístup k uloženým dátam.

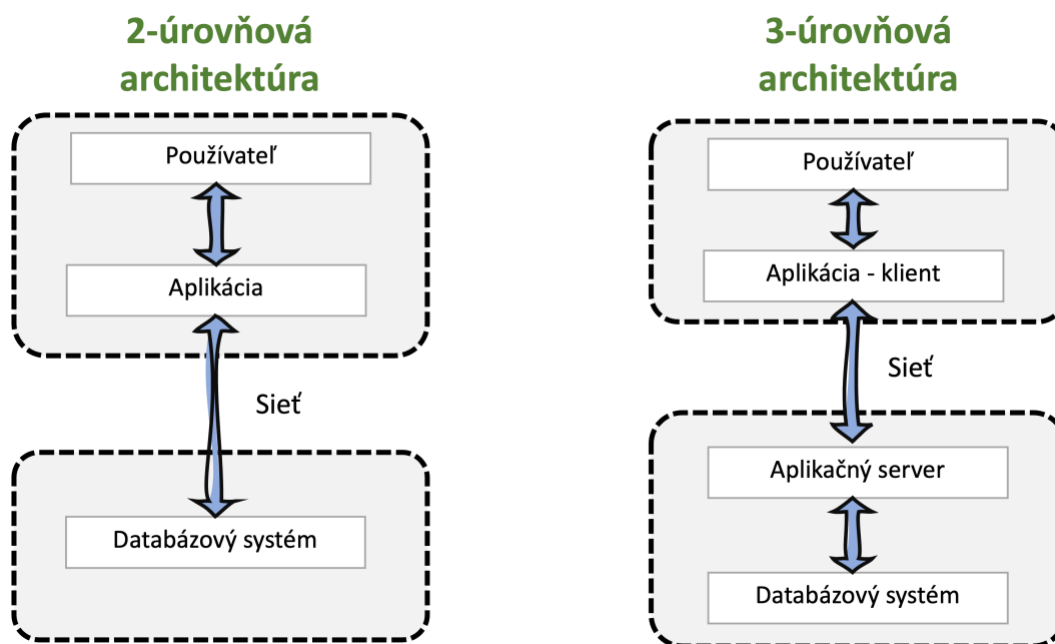
1.1 Architektúra databázových aplikácií

V rámci aplikácií, ktoré využívajú databázu existujú dve architektúry a to **2-úrovňová architektúra** a **3-úrovňová architektúra**. Tieto architektúry sú znázornené na Obr. 1.

2-úrovňová architektúra pozostávala z aplikácie, ku ktorej pristupoval používateľ. Samotná aplikácia bola spustená na používateľovom počítači, odkiaľ posielala cez sieť dopyty na server, na ktorom sa nachádzal databázový systém. Bezpečnosť takéhoto systému je menšia vzhľadom na priamy prístup klientskej aplikácie do databázy. V prípade zmeny na strane databázy je potrebné aktualizovať všetky klientske aplikácie.

Novšia **3-úrovňová architektúra** má aplikáciu rozdelenú na dve časti - klientsku časť a serverovú časť. Klientska časť má na starosti zobrazovanie informácií a interakciu s používateľom a posiela cez sieť informácie, aké dáta chce získať alebo uložiť zo servera. Až samotná serverová časť komunikuje s databázou pomocou dopytovacieho jazyka. Tým, že klientska časť nemá prístup priamo k databáze, dochádza k zvýšeniu bezpečnosti, a tiež nie je nutné modifikovať klientsku aplikáciu pri prípadných zmenách na dátovej vrstve. Výkonnosť takéhoto systému je tiež vyššia.

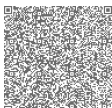


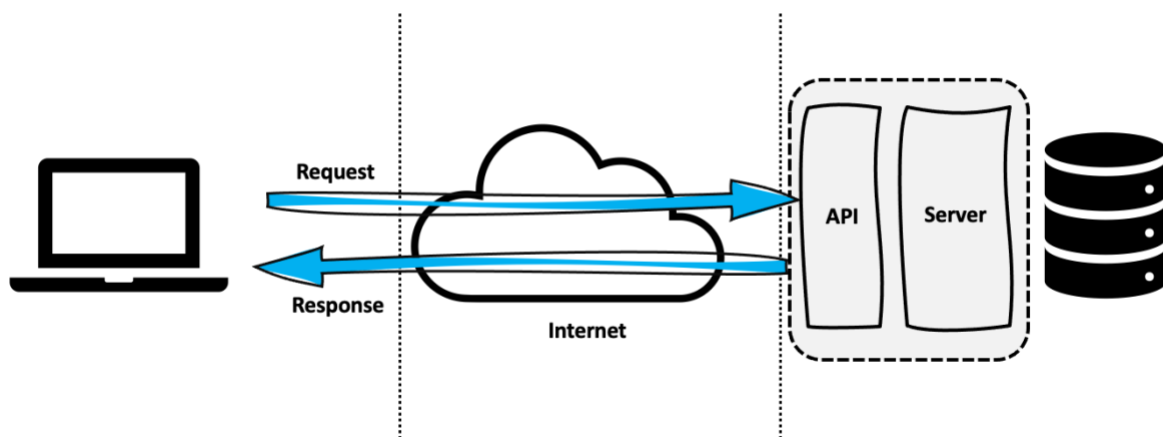


Obr. 1. 2-úrovňová a 3-úrovňová architektúra aplikácií využívajúcich databázu

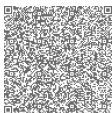
Ukážka komunikácie medzi klientom a serverom v rámci aplikácie využívajúcej 3-úrovňovú architektúru a je znázornená na Obr. 2. Komunikácia prebieha nasledovne:

1. Používateľ odošle pomocou klientskej aplikácie (napr. web stránka) požiadavku (z *angl. request*) na server.
2. Požiadavka je prijatá na serveri v rámci API rozhrania. API rozhranie má definované, aké požiadavky prijíma a aké parametre musí obsahovať daná požiadavka. Príkladom API rozhrania je REST API (REpresentational State Transfer Application Programming Interface) alebo SOAP (Simple Object Access Protocol).
3. Prijaté dáta sú následne spracované serverom, ktorý vykonáva biznis logiku. V rámci definovanej biznis logiky sa server rozhodne, čo potrebuje získať alebo modifikovať na strane databázy.
4. Server po vykonaní biznis logiky odpovedá klientskej aplikácii pomocou API, na ktorom bola prijatá požiadavka. Odpoveď (z *angl. response*) má tiež definované parametre, ktoré má obsahovať
5. Odpoveď je prijatá na klientskej strane a informácie sú poskytnuté používateľovi.





Obr. 2. Ukážka komunikácie medzi klientom a serverom v 3-úrovňovej architektúre



2 Dátový model

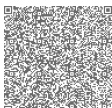
Pri dátových modeloch je potrebné rozlišovať pojmy **dátový model** a **schéma** v rámci konkrétneho dátového modelu. **Dátový model** predstavuje súbor princípov/pravidiel pre popísanie dát, ktoré majú byť uchovávané v databáze. Tieto pravidla definujú, ako sú dáta reprezentované v dátovom modeli, aké dáta je možné uchovávať, ako sú definované vzťahy medzi dátami atď. **Schéma** je viazaná na konkrétny dátový model s tým, že sa viaže na konkrétne dáta a ich popis. Neobsahuje však konkrétne dáta, iba popisuje, že takéto typy a vzťahy má daná databáza. Konkrétne dáta môžu byť označované ako záznam alebo je možné použiť označenie inštancia databázy (z *angl. database instance*), čo predstavuje *snapshot*, konkrétnej databázy v určitom čase.

V rámci databázových systémov existujú rôzne dátové modely a samotné implementácie databázových systémov sa viažu na konkrétny dátový model. Niektoré databázové systémy ponúkajú rozšírenia, ktoré umožní aplikovanie rôznych dodatočných funkcionalít využívaných inými databázovými systémami.

Existujú rôzne dátové modely, ktoré môžu byť nasledovné:

- Relačný
- Key/Value
- Graph
- Document
- Column family
- Array-matrix
- Hierarchical
- Network

Relačný dátový model patrí v súčasnosti medzi najčastejšie používaný. Dátové modely **Key/Value**, **Graph** (Grafové databázy), **Document**, **Column family** patria do rodiny NoSQL



databáz. **Array-matrix** dátový model nachádza uplatnenie v rámci strojového učenia (z *angl. machine learning*). Posledné dva dátové modely **Hierarchical** a **Network** sú reprezentantmi historických dátových modelov, ktoré sa v súčasnosti nevyužívajú a neprinášajú žiaden benefit.

V ďalšej časti bude stručne popísaný relačný dátový model, ktorý súvisí s používaním jazyka SQL, ktorému sa venujeme v osobitnej kapitole.

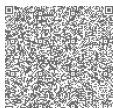
2.1 Relačný dátový model

Relačný dátový model bol predstavený matematikom Edgarom Frankom Coddom v roku 1970 a je popísaný v článku [1]. V rámci relačného modelu uvažujeme o neusporiadanej množine, kde dáta sú uložené v reláciách/tabuľkách (z *angl. relation*). V tejto časti skript budú používané obidva pojmy (relácia, tabuľka) kvôli získaniu prehľadu aj druhého názvoslovia, ktoré je viazané na relačný dátový model. V časti, kde je popísané SQL a fungovanie databázových systémov je používaný už len pojem *tabuľka*.

Medzi základný komponent, z ktorého sa skladá relačný dátový model je relácia/tabuľka. Každá relácia/tabuľka musí v rámci databázy obsahovať unikátne meno. Nie je možné aby dve relácie/tabuľky obsahovali rovnaké meno.

Medzi základné pojmy v rámci relácie sú:

- **n-tica** – je používaný pojem pre jeden záznam v rámci relácie. V kontexte pojmu tabuľka sa používa pojem riadok a tiež je používaný pojem záznam.
- **atribút** – reprezentuje stĺpec tabuľky.
- **hodnota atribútu** – reprezentuje hodnotu daného atribútu pre konkrétny záznam (n-ticu). V rámci terminológie používanej pre tabuľky by bolo možné použiť označenie bunka tabuľky.
- **primárny kľúč** – predstavuje jednoznačný identifikátor záznamu, ktorý pozostáva z jedného alebo viacerých záznamov.
- **doména** – predstavuje množinu všetkých hodnôt, ktoré nadobúda daný atribút.



- **cudzí kľúč** - slúži pre odkazovanie sa na iný záznam relácie.

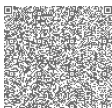
Ukážka príkladu relácie *authors*, je zobrazená v Tabuľka 1, kde môžeme vidieť štyri atribúty (stĺpce) a to *id*, *name*, *country*, *year*. Relácia obsahuje tri n-tice (záznamy).

Tabuľka 1. Ukážka relácie/tabuľky v relačnom dátovom modeli

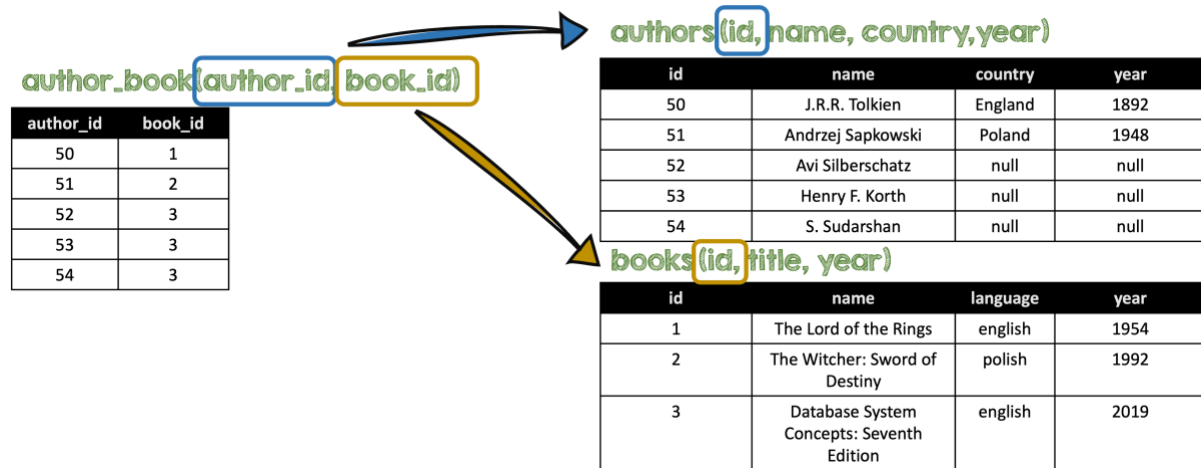
id	name	country	year
123	J.R.R. Tolkien	England	1892
124	František Modrý	Poland	1948
125	František Modrý	Czech Republic	1957

V rámci relačného modelu vzhľadom na matematický základ sa uvažuje o relácií ako o množine **n-tic** (záznamov), čo znamená, že v celej relácii sa nenachádzajú dve rovnaké **n-tice** (záznamy). Vzhľadom na definíciu primárneho kľúča, to ani nie je možné, pretože má jednoznačne identifikovať záznam v relácii. V prípade definovaných operácií nad reláciami v rámci relačnej algebry je možné dosiahnuť duplicitné záznamy, ale pretože pracujeme s množinou a nie s multi-množinou, tak sú tieto záznamy odstránené. V rámci relačnej algebry existuje viacero operácií a sú to napríklad selekcia (výber riadkov), projekcia (výber stĺpcov), množinové operácie (prienik, zjednotenie, rozdiel), spájanie tabuliek, premenovanie. Príkladom relačnej algebry, ktorú využíva aj SQL sa však nebudeme venovať.

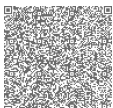
Silnou stránkou relačného dátového modelu je možnosť vytvárania vzťahov medzi jednotlivými tabuľkami. V rámci relácie vieme definovať vzťah pomocou odkazovania sa na iný záznam pomocou cudzieho kľúča, ktorý odkazuje na primárny kľúč. Príklad prepojenia relácií je znázornený na Obr. 3., kde vidíme relácie: *authors*, ktorá obsahuje zoznam autorov; reláciu *books*, ktorá obsahuje zoznam kníh; reláciu *author_book*, ktorá v tomto prípade prepája relácie *authors* a *books* z dôvodu, že jedna kniha môže byť napísaná viacerými autormi. Ak by jedna kniha mohla byť napísaná iba jedným autorom, tak by sme nepotrebovali reláciu *author_book*. Relácia *author_book* obsahuje cudzie kľúče na tabuľky *authors* a *books*. Zo záznamu v *author_book* je možné povedať, že knihu s *id* = 1 napísal autor s *id* = 50.



Prepojenie je docielené tak, že atribút *author_id* odkazuje na primárny kľúč v tabuľke *authors* (atribút *id*). Rovnako je to aj v prípade prepojenia na tabuľku *books*, len sa používa atribút *book_id*.



Obr. 3. Prepojenie v relačnom dátovom modeli



3 SQL

Táto kapitola sa podrobne venuje jazyku SQL (Structured Query Language).

3.1 História SQL

Prvotná verzia jazyka SQL (Structured Query Language) nazývaná ako Sequel (Structured English Query Language) bola vyvinutá firmou IBM v 70. rokoch 20. storočia ako súčasť projektu System R (databázový systém). V súčasnosti je vysoká podpora SQL jazyka v rámci relačných databázových systémov a je štandardizovaný. Prvý štandard bol vydaný v roku 1986 s označením SQL-86 a bol zastrešený organizáciami American National Standards Institute (ANSI) a o rok neskôr 1987 ho vydala aj International Organization for Standardization (ISO).

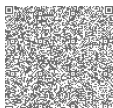
SQL štandard pridáva nové možnosti. Stručný prehľad verzií SQL štandardu je:

- SQL:2019: Multi-dimensional arrays (SQL/MDA)
- SQL:2016 – JSON, Polymorphic table
- SQL:2011 – Dočasné DB, Pipelined DML
- SQL:2008 – Truncation,
- SQL:2003 – XML, Windows, Sekvencie, Auto-Gen ID
- SQL:1999 - Regex, Triggers

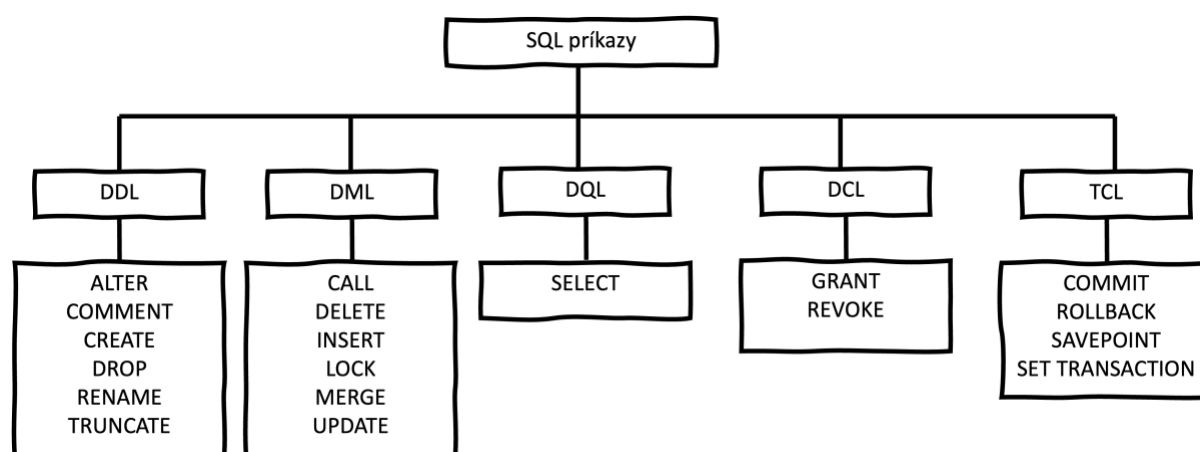
Rôzne databázové systémy obsahujú implementovaný rôzny rozsah SQL štandardu a je vždy potrebné poznať konkrétny databázový systém.

Samotný SQL jazyk obsahuje viacero častí, ktoré sú zamerané na rôzne oblasti a je ich možné rozdeliť na dve hlavné časti:

- **Data Definition Language** (DDL) – poskytuje príkazy, ktoré slúžia pre definovanie relačnej schémy. Obsahujú príkazy pre vytvorenie, vymazanie či modifikáciu relácie/tabuľky.
- **Data Manipulation Language** (DML) – obsahuje príkaz, ktoré slúžia pre prácu so samotnými dátami uložených v reláciách/tabuľkách.



Okrem popísaného rozdelenia je možné sa stretnúť aj s rozdelením, ktoré rozširuje tieto dve skupiny o Data Control Language (DCL), Data Query Language (DQL) a Transaction Control Language (TCL). V prípade rozdelenia na dve skupiny (DDL a DML) sú príkazy DQL, DCL a TCL zahrnuté v rámci DML. Samotný DQL obsahuje príkaz **SELECT**, ktorý slúži pre získavanie dát z relácií/tabuľky. DCL obsahuje príkazy pre správu privilégii nad databázou. V rámci TCL sú zahrnuté príkazy pre transakcie ako je napr. **COMMIT**. Rozdelenie do piatich skupín je znázornené na Obr. 4.



Obr. 4. Rozdelenie SQL príkazov

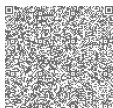
3.2 Data Definition Language

V rámci DDL definujem samotné tabuľky, pre ktoré definujeme nasledujúce informácie:

- **Atribúty** tabuľky spolu s ich dátovým typom
- Integritné obmedzenia
- Indexy

V rámci DBMS existujú rozličné dátové typy, ktoré sa líšia od implementácie napr. pre *PostgreSQL* vo verzii 15 je ich zoznam dostupný v rámci oficiálnej dokumentácie¹. Základné

¹ <https://www.postgresql.org/docs/15/datatype.html>



dátové typy sú rôzne textové alebo číselné. Oproti štandardným programovacím jazykom môže každý dátový typ obsahovať špeciálnu hodnotu **NULL**.

3.2.1 Hodnota NULL

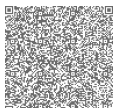
V rámci SQL existuje špeciálna hodnota **NULL**, ktorá predstavuje neznámu hodnotu pre atribúty. Je možné si to predstaviť, že v rámci databázy je evidencia osôb s kontaktnými údajmi, medzi ktoré patrí telefónne číslo. Pri niektorých osobách však toto číslo nie je známe, a preto nie je vyplnené. V takomto prípade obsahuje atribút pre telefónne číslo hodnotu **NULL**. Táto hodnota sa však z hľadiska podmienok správa podľa uvedených **tabuliek X**.

Tabuľka 2 Správanie hodnoty NULL pre operácie negácie, logického súčinu a súčtu

Hodnota	Negácia hodnoty	AND			OR		
		True	False	NULL	True	False	Null
TRUE	FALSE	TRUE	FALSE	NULL	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	NULL
NULL	NULL	NULL	FALSE	NULL	TRUE	NULL	NULL

3.2.2 Definovanie tabuľky

V rámci SQL pre vytvorenie novej tabuľky je potrebné použiť príkaz **CREATE TABLE ()**, v rámci ktorého je potrebné definovať jednotlivé atribúty samotnej tabuľky. Všeobecné definovanie tabuľky je nasledovné:



```
CREATE TABLE meno_tabuľky
(
    názov_atribútu_1 typ [integritné_obmedzenia],
    názov_atribútu_2 typ [integritné_obmedzenia],
    ...
    názov_atribútu_N typ [integritné_obmedzenia],
);
```

Ukážka 1. Všeobecná definícia SQL vytvorenia tabuľky

Uvažujme príklad, kde chceme vytvoriť tabuľku, ktorá bude evidovať hráčov s ich menami, platom a dátumom narodenia. Definovanie takejto tabuľky pre PostgreSQL vyzerá nasledovne:

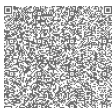
```
CREATE TABLE players (
    id int,
    name varchar(30),
    surname varchar(30),
    salary numeric,
    date_of_birth date );
```

Ukážka 2. Definovanie tabuľky players v PostgreSQL

V rámci uvedeného príkladu sú použité pre atribúty iba dátové typy bez definovaných integritných obmedzení. Uvedená definícia v Ukážka 1 nič negarantuje, že v rámci tabuľky sa neocitnú dva záznamy s rovnakým ID. Pridaním klauzuly **PRIMARY KEY** k atribútu *id* by sme zaručili unikátnosť každého záznamu. Definícia atribútu *id* ako primárny kľúč by vyzerala nasledovne:

```
id int PRIMARY KEY,
```

Ukážka 3. Definovanie primárneho kľúča priamo pri definovaní atribútu



Tiež je možné definovať primárny kľúč aj samostatne a to:

```
....  
date_of_birth date,  
PRIMARY KEY (id) );
```

Ukážka 4. Definovanie primárneho kľúča po definícii atribútov

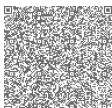
Jednotlivé integritné obmedzenia sú popísané v osobitnej časti 3.3.

3.2.3 Vytvorenie cudzieho kľúča

V tejto časti je popísaná tvorba prepojenia tabuliek v rámci relačného modelu. Aby sme mohli vytvoriť takého prepojenie, je potrebné zdefinovať novú tabuľku a to tabuľku s tímami s názvom *teams* a tabuľku s názvom *player_seasons* pre prepojenie medzi tabuľkou *players* a *teams* za účelom sledovania tímu, za ktorý hráč nastúpil v danej sezóne. Tabuľka *player_seasons* obsahuje referenciu *player_id* na tabuľku *players* a referenciu *team_id* na tabuľku *teams*. Definícia tabuliek *teams* a *player_seasons* je nasledovná:

```
CREATE TABLE teams (  
    id int PRIMARY KEY,  
    name varchar(30),  
    stadium_name varchar(30));  
  
CREATE TABLE player_seasons (  
    id int PRIMARY KEY,  
    player_id int NOT NULL REFERENCES players(id),  
    team_id int NOT NULL REFERENCES teams(id),  
    valid_from date,  
    valid_to date );
```

Ukážka 5. Definícia tabuliek teams a player_seasons spolu s primárnym a cudzím kľúčom

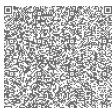


3.3 Integritné obmedzenia

V rámci databázy je dôležité dodržiavanie integritných obmedzení pre zabezpečenie, že v prípade zmien v databáze autorizovaným používateľom nedôjde k strate konzistencie. Dochádza teda k ochrane dát pred náhodným poškodením používateľom (pod používateľom rozumieme aj aplikáciu, pretože do databázy prístupuje cez prihlasovacie údaje). Integritné obmedzenia je možné definovať pri vytváraní tabuľky, ale je ich tiež možné pridať dodatočne pomocou klauzuly **ALTER**. V prípade dodatočného pridania však všetky už vložené záznamy musia spĺňať dané obmedzenie, inak pridanie integritného obmedzenia nebude akceptované.

Medzi integritné obmedzenia, ktoré je možné definovať v rámci tabuľky zaraďujeme nasledovné:

- **NOT NULL** – atribút nemôže obsahovať NULL hodnotu, vždy musí obsahovať nejakú hodnotu. Nastavuje sa pomocou klauzuly **NOT NULL**, ktorá je uvedená pri konkrétnom atribúte pri definovaní tabuľky napr. používateľ musí mať vyplnené prihlasovacie meno.
- **UNIQUE** – atribút v rámci tabuľky môže nadobúdať len jedinečné hodnoty. Hodnota atribútu v jednom zázname musí byť rozdielna voči všetkým ostatným záznamom v danej tabuľke. Platí to pre každý záznam v tabuľke napr. prihlasovacie meno musí byť jedinečné pre každého používateľa. Definuje sa pomocou klauzuly **UNIQUE**, ktorá je uvedená pri atribúte v rámci definovania tabuľky.
- **CHECK** – atribút obsahuje kontrolu, ktorá povoľuje pridanie iba takých hodnôt, ktoré spĺňajú uvedenú podmienku. Definuje hodnoty, ktoré môže nadobúdať atribút napr. či nie je vložený záporný počet strelených gólov v zápase. Definuje sa pomocou klauzuly **CHECK** spolu s podmienkou, ktorá je uvedená pri atribúte v rámci definovania tabuľky.
- **DEFAULT** – v prípade, že počas vkladania záznamu nie je nastavená hodnota, tak sa použije hodnota definovaná v rámci klauzuly **DEFAULT** uvedenej pri atribúte počas definovania tabuľky napr. nastavenie automatického času pre čas vytvorenia záznamu.



- **PRIMARY KEY** – definuje primárny kľúč v rámci tabuľky. Jednoznačne identifikuje záznam v tabuľke a môže sa skladať z jedného alebo viacerých atribútov. Definuje sa pomocou klauzuly **PRIMARY KEY**. Väčšinou sa používa atribút *id* pre identifikovanie záznamu. Pre zvýšenie bezpečnosti je lepšie pre atribút *id* používať typ UUID namiesto inkrementálneho zvyšovania čísla. Ďalšou funkciou primárneho kľúča je, že vstupuje do vzťahu medzi tabuľkami.
- **FOREIGN KEY** – definuje cudzí kľúč, ktorý odkazuje na inú tabuľku a jej primárny kľúč. Slúži na vytvorenie vzťahu medzi tabuľkami. Pre vytvorenie sa používa klauzula **REFERENCES** a viac informácií je popísaných v časti referenčná integrita.

Všetky vyššie integritné obmedzenia je možné definovať aj dodatočne.

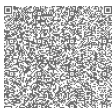
3.3.1 Referenčná integrita

Referenčná integrita predstavuje stav v databáze, kedy odkazujúce záznamy sa vždy odvolávajú na platné záznamy. Na Obr. 5 je znázornené porušenie referenčnej integrity, kde tabuľka *author_book*, ktorá sa odvoláva na tabuľky *authors* a *books* obsahuje záznam, ktorý odkazuje na knihu s *id* = 34. Takýto záznam sa však nenachádza v rámci tabuľky *books*. V prípade tohto záznamu prišlo k porušeniu referenčnej integrity. Databázový systém nedovolí, aby nastala takáto situácia, či už pri vkladaní, modifikácií alebo mazaní záznamov pokiaľ existuje vytvorený vzťah medzi tabuľkami (primárny a cudzí kľúč). Referenčná integrita je kontrolovaná aj v prípade, že nad existujúcimi tabuľkami so záznamami ideme vytvoriť vzťah.

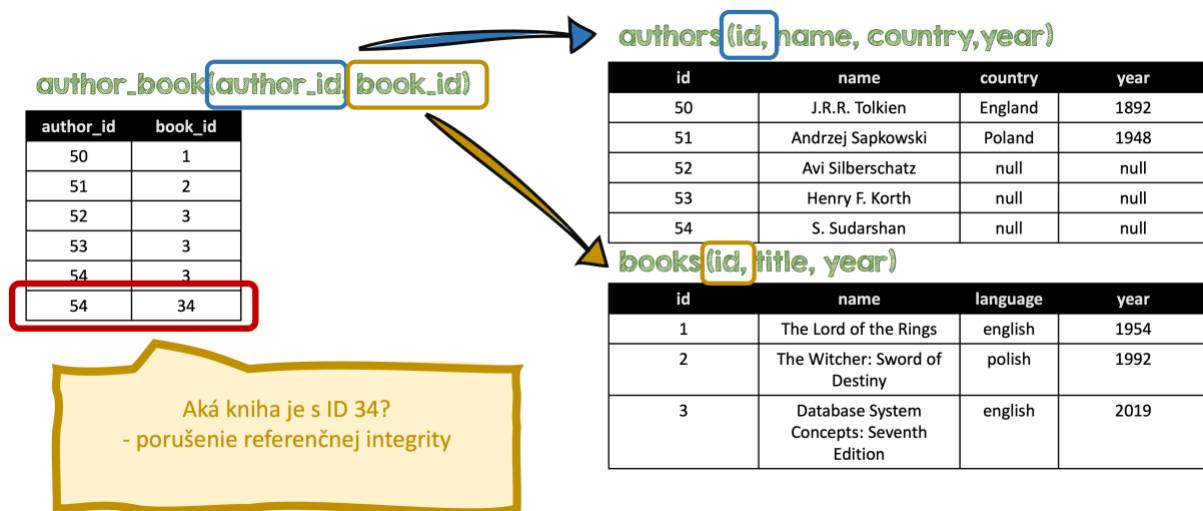
Pri vytvorení vzťahu medzi dvomi tabuľkami sa používa označenie *parent* a *child* tabuľka. *Parent* tabuľka predstavuje tabuľku, na ktorú sa odkazuje iná tabuľka pomocou cudzieho kľúča. Tabuľka s cudzím kľúčom je označovaná ako *child*, pretože sa odkazuje na tabuľku s primárnym kľúčom. V texte budeme používať označenie *parent* a *child* záznam pre reprezentáciu konkrétneho záznamu v daných tabuľkách.

Porušenie referenčnej integrity nastáva v týchto situáciách:

- odstránenie záznamu, na ktorý odkazuje iný záznam - odstránenie *parent* záznamu, ktorý obsahuje primárny kľúč, na ktorý odkazuje iný záznam pomocou cudzieho kľúča



- aktualizácia referencie cudzieho kľúča záznamu na neexistujúci primárny kľúč
- aktualizácia primárneho kľúča záznamu a neaktualizovanie *child* záznamu, ktorý odkazuje na daný *parent* záznam. Ak je však potrebné aktualizovať z nejakého dôvodu primárny kľúč záznamu, tak je potrebné sa zamyslieť nad tým, či je zvolený vhodný dizajn aplikácie.



Obr. 5. Príklad s porušením referenčnej integrity

Pri modifikovaní a mazaní záznamov v rámci tabuliek existujú stratégie ako sa vysporiadať s porušením referenčnej integrity. Všeobecná definícia cudzieho kľúča spolu so stratégiou pri modifikovaní *parent* záznamu je:

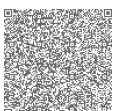
```

...
názov_atribútu_pre_cudzí_kľúč dátový_typ
REFERENCES tabuľka(primárny_kľúč)
ON UPDATE typ_stratégie
ON DELETE typ_stratégie,
...

```

Ukážka 6. Všeobecná definícia cudzieho kľúča spolu s definovaním správania pri modifikácii *parent* záznamu

Pri vytvorení cudzieho kľúča je možné definovať správanie v prípade vymazania alebo modifikácie *parent* záznamu. Klauzula **ON UPDATE** slúži pre definovanie správania pri modifikácii *parent* záznamu. Modifikáciou *parent* záznamu rozumieme zmenu hodnoty



atribútu primárneho kľúča. Druhá klauzula **ON DELETE** slúži pre definovanie správania pri vymazaní *parent* záznamu. Typ stratégie pri týchto dvoch klauzulách definuje správanie *child* záznamu v daných situáciach. V rámci typu stratégie rozlišujeme nasledujúce typy podľa PostgreSQL dokumentácie:

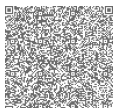
- **CASCADE** – v prípade vymazania *parent* záznamu nastáva zároveň vymazanie záznamov, ktoré naň odkazujú (*child* záznamy). Aktualizovanie *parent* záznamu spôsobí aj aktualizáciu *child* záznamu.
- **RESTRICT** – v prípade vymazania alebo modifikácií *parent* záznamu, nebude umožnené uskutočniť vymazanie alebo modifikáciu, pokiaľ existuje *child* záznam.
- **SET NULL** – v prípade vymazania alebo modifikácií *parent* záznamu bude v rámci *child* záznamu nastavená hodnota cudzieho kľúča na hodnotu *NULL*.
- **SET DEFAULT** - v prípade vymazania alebo modifikácií *parent* záznamu bude v rámci *child* záznamu nastavená hodnota cudzieho kľúča na *DEFAULT* hodnotu, ktorá je definovaná pre cudzí kľúč.
- **NO ACTION** – rovnaké správanie ako v prípade typu **RESTRICT** s tým rozdielom, že v prípade transakcie je možné odložiť kontrolu na neskôr, čo v prípade **RESTRICT** nie je možné.

3.4 Základná štruktúra SQL dopytu

Pre získanie požadovaných informácií z databázy je potrebné napísať dopyt (z *angl. query*), ktorý hovorí, aké dáta nás zaujímajú a nie ako majú byť získané. Jednoduchý dopyt pozostáva z klauzúl **SELECT**, **FROM**, **WHERE**. Z hľadiska syntaxe nezáleží, či sú názvy príkazov napísané veľkými alebo malými písmenami (prípadne ich kombináciou). Príklad použitia je nasledovný:

```
SELECT stĺpce tabuľky  
FROM tabuľky  
WHERE podmienka;
```

Ukážka 7. Všeobecná definícia dopytu pre získanie dát



Príkaz **SELECT** hovorí, ktoré stĺpce z tabuliek definovaných v rámci príkazu **FROM** sú získané. Získavanie informácií z viacerých tabuliek je popísané v časti 3.9. Klauzula **WHERE** filtruje záznamy. Vykonávanie takéhoto dopytu je nasledovné:

1. **FROM** – vyberú sa záznamy zo všetkých uvedených tabuliek, pre ktoré sa uskutoční karteziánsky súčin $\text{tabuľka}_1 \times \text{tabuľka}_2 \times \text{tabuľka}_3 \times \dots$

- Takéto spojenie sa nazýva **CROSS JOIN**

2. **WHERE** – na výsledok z klauzuly **FROM** je aplikovaný uvedený filter

3. **SELECT** – sú vybraté len uvedené stĺpce.

Pozn. Aplikovanie filtrov pre záznamy je z pohľadu výsledkov možné použiť pred samotným spojením tabuliek. Je to efektívnejšie z hľadiska JOIN algoritmov.

3.5 Modifikácia záznamov

V tejto časti sú popísané syntaxe pre pridanie, vymazanie a modifikovanie záznamov v tabuľke.

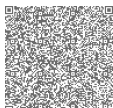
3.5.1 Pridanie záznamov

Pridanie záznamov do tabuľky je možné uskutočniť dvomi spôsobmi. Tieto dva spôsoby sa líšia iba v tom, že v prípade prvého spôsobu nie je potrebné definovať stĺpce, ktoré budú napĺňané v rámci tabuľky. V tomto prípade sa berie poradie stĺpcov tak, ako sú definované pri vytváraní tabuľky. Druhý spôsob definuje, ktoré stĺpce budú napĺňané v rámci tabuľky s tým, že poradie môže byť odlišné oproti definícii tabuľky. Syntax pre prvý spôsob je nasledovná:

```
INSERT INTO tabuľka
VALUES (hodnota1, hodnota2, ..., hodnotaN);
```

Ukážka 8. Všeobecná definícia vloženia záznamu bez špecifikovania stĺpcov

Vloženie do tabuľky začína klauzulou **INSERT INTO**, za ktorou je definovaná tabuľka, do ktorej budú pridávané nové záznamy. Nasleduje klauzula **VALUES**, ktorá definuje hodnoty, ktoré



majú byť pridané ako nový záznam. Poradie je v tomto prípade potrebné dodržať s poradím definície tabuľky.

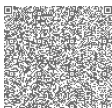
Syntax pre druhý spôsob je nasledovná:

```
INSERT INTO tabuľka (názov_atribútu1, názov_atribútu2, ... ,  
                    názov_atribútuN)  
    VALUES (hodnota_1, hodnota_2, .... , hodnota_N),  
            (hodnota_x1, hodnota_x2, .... , hodnota_xN);
```

Ukážka 9. Všeobecná definícia vloženia záznamu s definovaním stĺpcov

Oproti prvému spôsobu sú za názvom tabuľky definované atribúty, ktoré budú pridávané. Niektoré atribúty v rámci tabuľky môžu byť vyplňané automaticky prípadne neskôr a takýmto spôsobom ich nie je nutné riešiť. Tiež je v rámci tohto spôsobu znázornené pridanie dvoch záznamov v rámci jedného dopytu. Záznamy sú ohraničené pomocou zátvoriek a sú oddelené pomocou čiarky. Pridanie viacerých záznamov je možné aj v rámci prvého spôsobu rovnakým štýlom ako v prípade druhého spôsobu.

Pre potreby príkladov v týchto skriptách sú tabuľky *players*, *player_seasons*, *teams* z časti 3.2 naplnené nasledujúcimi záznamami:



```

INSERT INTO players (id, name, surname, salary, date_of_birth)
VALUES (100, 'Joe', 'Sakic', 100000, '7/7/1969'),
      (200, 'Patrice', 'Bergeron', 90000, '7/24/1985'),
      (300, 'Auston', 'Matthews', 120000, '9/17/1997');

INSERT INTO teams (id,name,stadium_name)
VALUES (5,'Colorado','Ball Arena'),
      (6,'Boston','Boston Garden'),
      (7,'Toronto','Air Canada Centre');

INSERT INTO player_seasons (id,player_id,team_id,valid_from,valid_to)
VALUES (1,100,5,'9/1/2022','6/30/2023'),
      (2,100,5,'9/1/2023','6/30/2024'),
      (3,200,6,'9/1/2022','6/30/2023');

```

Ukážka 10. Naplnenie tabuliek players, teams a player_seasons záznamami

3.5.2 Vymazanie záznamu

Vymazanie záznamu z tabuľky je uskutočnené pomocou klauzuly **DELETE FROM**, ktorá definuje názov tabuľky. Nasleduje klauzula **WHERE**, ktorá definuje podmienku, na základe ktorej budú vymazané tie záznamy, ktoré vyhovuje tejto podmienke. Syntax vyzerá nasledovne:

```

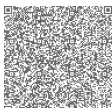
DELETE FROM tabuľka
WHERE podmienka

```

Ukážka 11. Všeobecná definícia vymazania záznamov z tabuľky

3.5.3 Aktualizácia záznamu

Úprava hodnôt v rámci existujúceho záznamu je možná v rámci SQL pomocou klauzuly **UPDATE**, ktorá definuje tabuľku. Nasleduje klauzula **SET**, kde sa nastavujú nové parametre existujúcich atribútov v rámci tabuľky. Je možné nastaviť viacero atribútov naraz. Pre špecifikovanie záznamov, pre ktoré má dôjsť k úprave hodnôt je potrebné použiť klauzulu



WHERE rovnako ako v prípade **SELECT** alebo **DELETE FROM**. Aktualizované sú všetky záznamy, ktoré vyhovujú podmienke, a teda je potrebné dávať pozor, aby neprišlo k neželaným prepisom. V podmienke sa porovnávajú hodnoty ešte pred samotnou úpravou definovanou v rámci klauzuly **SET**. Všeobecná syntax pre aktualizovanie záznamu vyzerá nasledovne:

```
UPDATE tabuľka
    SET atribút_1 = nová_hodnota_1, ..., atribút_N = nová_hodnota_N
    WHERE podmienka;
```

Ukážka 12. Všeobecná definícia aktualizovania záznamu v tabuľke

3.6 Usporiadanie výsledkov

Pre usporiadanie výsledkov sa využíva príkaz **ORDER BY**, v rámci ktorého sú definované stĺpce, podľa ktorých je uskutočnené usporiadanie. Poradie stĺpcov určuje ich prioritu v rámci usporiadania. V prípade, že dva výsledky sú rovnaké podľa prvého definovaného stĺpca, tak sa pokračuje na ďalší stĺpec definovaný v rámci **ORDER BY**. Okrem vymenovania stĺpcov je možné v rámci PostgreSQL použiť aj poradie stĺpcov napr. 1, 2 ktoré vychádza z usporiadania stĺpcov v rámci klauzuly **SELECT**.

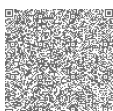
Usporiadanie je možné uskutočniť vzostupne alebo zostupne s možnosťou definovania pre každý stĺpec osobitne v rámci klauzuly **ORDER BY**. Definovanie poradia je nasledovné:

- **ASC** – *ascending* – vzostupné usporiadanie – od najmenšieho po najväčšie
- **DESC** – *descending* – zostupné usporiadanie – od najväčšieho po najmenšie

Všeobecný príklad použitia je nasledovný:

```
SELECT stĺpce tabuľky
    FROM tabuľka
    WHERE podmienka
    ORDER BY stĺpec_1 ASC stĺpec_2 DESC;
```

Ukážka 13. Všeobecná definícia usporiadania v rámci SQL dopytu



3.7 Jednoznačnosť atribútov a aliasy

V rámci SQL existuje možnosť definovania dočasných názvov pre názvy stĺpcov a tabuliek v rámci samotného dopytu. Takéto dočasné názvy sa nazývajú aliasy. Použitie aliasov v rámci SQL dopytu má dva hlavné významy:

- Skrátenie zápisu a zlepšenie čitateľnosti dopytu
- Zabezpečenie jednoznačnosti atribútov

Zabezpečenie jednoznačnosti atribútov je možné pridaním názvu tabuľky pred atribút a zápis vyzerá nasledovne:

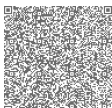
```
názov_tabuľky.atribút
```

Ukážka 14. Všeobecná syntax Jednoznačného určenia atribútu

Význam takejto definície ma v prípade spájania viacerých tabuliek, kde sa môže rovnaký názov atribútu (názov stĺpca) vyskytnúť viackrát, napríklad *id* je štandardne používaný ako identifikátor záznamu v tabuľke. Každá tabuľka teda obsahuje názov stĺpca *id*. V takomto prípade je preto nevyhnutné rozlíšiť tieto stĺpce od seba. V snahe vyhnúť sa používaniu dlhých názvov tabuliek pre zabezpečenie jednoznačnosti atribútov, je možné použiť práve aliasy. Aliasy sú v niektorých prípadoch nevyhnutné pre zabezpečenie jednoznačnosti atribútov kvôli tomu, že rovnaká tabuľka, môže byť spojená sama so sebou a v takom prípade je nevyhnutné použiť alias pre rozlíšenie týchto dvoch tabuliek. Definovanie aliasu prebieha pomocou klauzuly **AS** a môže vyzeráť nasledovne:

```
SELECT pl.id, pl.name, pl.salary AS player_salary  
FROM players AS pl
```

Ukážka 15. Definovanie aliasu pre atribút a názov tabuľky pomocou klauzuly AS



Implementácia PostgreSQL umožňuje vynechanie klauzulu **AS** a predchádzajúci príklad by vyzeral nasledovne:

```
SELECT pl.id, pl.name, pl.salary player_salary
FROM players pl
```

Ukážka 16. Definovanie aliasu pre atribút a názov tabuľky bez klauzuly AS

Uvedený príklad definuje alias *pl* pre tabuľku *players* a ako je vidieť tak v rámci klauzuly **SELECT** má každý atribút definované, z ktorej tabuľky pochádza atribút s tým, že už je použitý definovaný alias pre tabuľku. Okrem toho je definovaný alias *player_salary* pre atribútu *salary*.

3.8 Agregácie

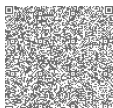
Použitie agregáčnych funkcií v rámci SQL umožňuje výpočet sumarizačnej hodnoty (závisle od použitej funkcie) nad množinou záznamov z tabuľky. Výsledkom je jedná hodnota pre množinu záznamov. Medzi základné agregáčné funkcie zahrňame:

- `avg()` – výpočet priemeru pre definovaný atribút z množiny záznamov.
- `count()` – vráti počet záznamov, či už pre počet záznamov alebo len pre tie záznamy, ktoré neobsahujú hodnotu **null**.
- `max()` – vráti najväčšiu hodnotu pre vybraný atribút z množiny záznamov.
- `min()` – vráti najmenšiu hodnotu pre vybraný atribút z množiny záznamov.
- `sum()` – vráti súčet pre definovaný atribút z množiny záznamov.

Všeobecná ukážka použitia agregáčnej funkcie je nasledovná:

```
SELECT agregáčné funkcie (stĺpec tabuľky)
FROM tabuľka
WHERE podmienka
```

Ukážka 17. Všeobecná definícia použitia agregáčnej funkcie bez klauzuly GROUP BY



V tomto prípade je agregáčna funkcia vypočítaná nad všetkými záznamami, ktoré spĺňajú podmienku definovanú vo **WHERE** klauzule. Ak by sme chceli vypočítať štatistiku pre hráčov jedného tímu a ich priemerný počet gólov za sezónu. *Pozn. Pre jednoduchosť príkladu uvažujeme o jednej tabuľke. V dobre navrhnutom dátovom modeli by sa niektoré informácie nachádzali v iných tabuľkách.* Uvažujme, že tabuľka *players* obsahuje parametre *id; name; surname; team; season; goals; assists*. Dopyt pre sezónu 2022/2023 a tím *Colorado* by vyzeral nasledovne:

```
SELECT avg(goals)
FROM players
WHERE season = '2022/2023' AND team = 'Colorado';
```

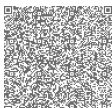
Ukážka 18. Výpočet priemerného počtu gólov tímu (na hráča tímu) pre konkrétny tím a sezónu bez klauzuly GROUP BY

Takýto dopyt by najprv vyfiltroval tím *Colorado* pre sezónu 2022/2023 a následne by vyrátal pre vyfiltrované záznamy gólový priemer pre všetkých nájdených hráčov. Keby sme však chceli vyrátať gólový priemer na hráča pre každý tím v sezóne 2022/2023, tak by dopyt musel byť spustený toľkokrát ako je počet tímov s tým, že by muselo prísť k modifikácii podmienky pre vyfiltrovanie správneho tímu. Aby vytváranie štatistík pre určitú množinu záznamov nebolo komplikované, existuje klauzula **GROUP BY**, ktorá umožní definovať, podľa čoho majú byť jednotlivé záznamy zoskupené. Vyberajú sa stĺpce tabuľky podľa ktorých bude uskutočnené zoskupenie. Následne je možné vyrátať vybranú agregáčnu funkciu. Všeobecná syntax pre použitie klauzuly **GROUP BY** vyzera nasledovne:

```
SELECT stĺpce z GROUP BY, agregáčne funkcie (stĺpec tabuľky)
FROM tabuľka
WHERE podmienka
GROUP BY stĺpce tabuľky
```

Ukážka 19. Všeobecná definícia použitia agregáčnej funkcie pomocou klauzuly GROUP BY

V časti pre **SELECT** sa nachádzajú stĺpce použité v rámci klauzuly **GROUP BY** a tiež agregáčne funkcie, ktoré sú predmetom daného dopytu. Oproti základnému dopytu tu nie je možné použiť všetky stĺpce z tabuľky kvôli tomu, že nie je možné povedať, ktorá hodnota by mala byť



použitá vzhľadom na zoskupovanie množstva záznamov do jedného výsledku. V prípade, kde boli rátaný priemerný počet gólov pre tím a sezónu, tak v prípade, že v klauzule **SELECT** by bol uvedený aj stĺpec *name* (meno hráča), tak databáza by musela náhodne vybrať aké meno hráča vyberie pre výsledok. SQL dopyt pre vyrátanie gólového priemeru na hráča pre každý tím by vyzeral nasledovne:

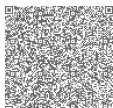
```
SELECT team, avg(goals)
FROM players
WHERE season = '2022/2023'
GROUP BY team;
```

Ukážka 20. Výpočet priemerného počtu gólov pre tímy (na hráča tímu) v sezóne 2022/2023 pomocou klauzuly GROUP BY

SQL dopyt najprv vyfiltruje sezónu 2022/2023 a následne zoskupí záznamy podľa stĺpca **team**. Pre vzniknuté zoskupenia podľa tímov je vyrátaný gólový priemer, ktorý sa vo výsledku uvedie spolu s názvom tímu, ktorému patrí daný priemer. V rámci klauzuly **SELECT** boli použité iba stĺpce z **GROUP BY** a agregáčna funkcia. Niektoré databázové systémy umožňujú použiť aj iné stĺpce z tabuľky ako sú definované v **GROUP BY**, avšak takéto správanie nie je podľa štandardu. Je preto potrebné poznať používaný databázový systém a jeho správanie v takýchto situáciách. Existuje však jedna výnimka, kedy je možné použiť aj stĺpce mimo **GROUP BY** a to v prípade, že klauzula **GROUP BY** obsahuje stĺpec, ktorý je **primárny kľúč**. V tomto prípade, je možné použiť všetky stĺpce z tabuľky, z ktorého pochádza daný primárny kľúč. Je to z toho dôvodu, že primárny kľúč jednoznačne identifikuje záznam, ktorý vždy bude mať atribúty rovnaké v rámci zoskupenia. Ak by sme napríklad rátali celkový počet gólov hráča pre všetky sezóny na základe ID hráča, tak vieme povedať že meno bude mať tento hráč rovnaké pre všetky nájdené záznamy a preto je ho možné vo výpise uviesť.

3.8.1 Filtrovanie záznamov po výpočte agregáčnej funkcie

Aby bolo možné odfiltrovať záznamy z výstupu agregáčnej funkcie existuje klauzula **HAVING**. Táto klauzula predstavuje podmienku, na základe ktorej sú vo výstupe zobrazené iba tie záznamy, ktoré spĺňajú definovanú podmienku. Oproti klauzule **WHERE** je rozdiel v tom, že táto podmienka je aplikovaná až po vypočítaní agregácií a teda je možné odfiltrovať výsledky, ktoré nie sú relevantné, napríklad tímy, kde priemerný počet gólov na hráča je menej ako 1.



Klauzula **WHERE** sa aplikuje ešte predtým, ako sú záznamy zoskupené podľa **GROUP BY**. Všeobecná definícia dopytu je nasledovná:

```
SELECT stĺpce z GROUP BY, agregáčn  funkcie (st pec tabu ky)
FROM tabu ka
WHERE podmienka
GROUP BY st pce tabu ky
HAVING v stupn  st pce z agrega n ch funkci 
```

Uk  ka 21. V eobecn  defin cia odfiltrovanie v sledkov po v po te agrega   pomocou klauzuly **HAVING**

Ak by sme chceli odfiltrova  t my, kde priemern  po et g lov na hr  a je men   ako 1, tak dopyt by vyzeral nasledovne:

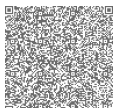
```
SELECT team, avg(goals)
FROM players
WHERE season = '2022/2023'
GROUP BY team
HAVING avg(goals) >= 1;
```

Uk  ka 22. Odfiltrovanie z znamov, kde priemern  po et g lov na hr  a je men   ako 1 pomocou klauzuly **HAVING**

V r mci klauzuly **HAVING** je uveden  `avg(goals) >= 1`,  o reprezentuje v stup, pod a ktor ho m  prebehn   filtrovanie po v po te priemeru. V r mci implementa cie PostgreSQL nie je mo n  vyu  va  aliasy v r mci klauzuly **HAVING**, aj ke  je ich mo n  pou  va  v r mci klauzuly **ORDER BY**. Je preto potrebn  nap sa  samotn  agrega n  funkciu, ktor  bola pou  t  v r mci klauzuly **SELECT** teda `avg(goals)`. Toto spr vanie sa v  ak m  e l     od implementa cie datab zov ho syst mu.

3.9 Pr a s viacer ymi tabu kami

Preto e rela n  model funguje na princ pe tabuliek, medzi ktor mi s  definovan  vz ahy, je  iadu e, aby bolo mo n  uskuto n va  dopyty nad viacer ymi tabu kami. Preto je potrebn  definova , z ktor ch tabuliek bud  z skavan  d ta a na z klade,  oho bud  prepojen . Na



prepojenie tabuliek v rámci SQL sa používa klauzula **JOIN ... ON ...**. Použitie operácie **JOIN** vyzerá nasledovne:

```
SELECT stĺpce_tabuľky
FROM tabuľka_1
JOIN tabuľka_2
ON podmienka_spojenia
```

Ukážka 23. Všeobecná definícia pre spojenie tabuliek pomocou klauzuly JOIN ... ON ...

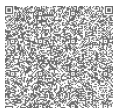
V uvedenom všeobecnom príklade SQL dopyt spája *tabuľka_1* s *tabuľka_2*. Prvá tabuľka pre výber sa nachádza za klauzulou **FROM** a následne je použitá klauzula **JOIN**, ktorá definuje, s akou tabuľkou má byť spojená *tabuľka_1*. V tomto prípade je použitá *tabuľka_2*. Po definovaní tabuliek spojenia nasleduje klauzula **ON**, ktorá obsahuje podmienku spojenia. Tabuľka nachádzajúca sa v rámci klauzuly **FROM** je považovaná za ľavú tabuľku (z angl. *Left table*) a tabuľka v rámci klauzuly **JOIN** je považovaná za pravú tabuľku (z angl. *Right table*). Štandardne spojenie je definované na kombinácii primárneho a cudzieho kľúča napríklad:

```
tabuľka_1.id = tabuľka_2.tabuľka_1_id
```

Ukážka 24. Podmienka pre spojenie dvoch tabuliek - kombinácia primárny a cudzí kľúč

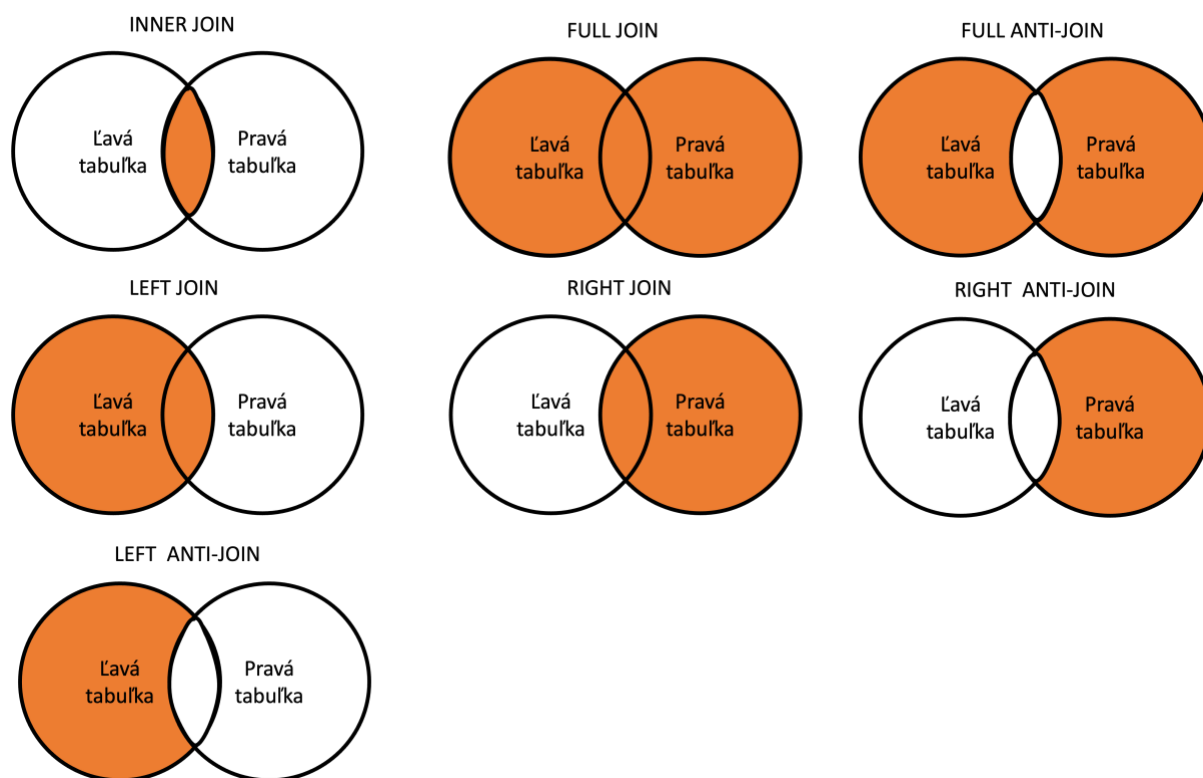
V prípade, že používateľ chce spojiť viacero tabuliek nie len dve, tak za podmienkou spojenia môže nasledovať ďalšia operácia **JOIN**. Pri práci so spájaním viac ako dvoch tabuliek treba pristupovať k spájaniu tak, že najprv sú spojené prvé dve tabuľky do jednej väčšej a následne táto spojená tabuľka je spojená s ďalšou tabuľkou v poradí do novej väčšej tabuľky. Toto spojenie je len dočasné a je dostupné iba počas vykonávania daného dopytu. Z pohľadu SQL nie je limitácia na počet tabuliek, ktoré je možné spojiť v rámci dopytu. Chápanie logiky spájania viacerých tabuliek je dôležité, keďže existuje viacero typov operácií **JOIN** a ich použitie vie ovplyvniť výsledok spojenia rovnako aj definované poradie spájania v rámci SQL dopytu. Poznáme nasledujúce typy **JOIN** operácií:

- **CROSS JOIN**
- **INNER JOIN** alebo **JOIN**



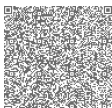
- **LEFT OUTER JOIN** alebo **LEFT JOIN**
- **RIGHT OUTER JOIN** alebo **RIGHT JOIN**
- **FULL OUTER JOIN** alebo **FULL JOIN**

Vizualizácia typov **JOIN** operácie je znázornená na Obr. 6, kde sú znázornené **JOIN**, **LEFT JOIN**, **RIGHT JOIN** a **FULL JOIN**. Existuje ešte typ **ANTI JOIN**, ktorý súvisí s typmi **LEFT JOIN**, **RIGHT JOIN** a **FULL JOIN** a je dosiahnutý pomocou pridania filtra do klauzuly **WHERE** pre zobrazenie iba tých záznamov, ktoré obsahujú NULL hodnotu pre cudzí kľúč, na základe ktorého sa uskutočnilo spojenie medzi dvomi tabuľkami.



Obr. 6. Typy spojenia tabuliek

Jednotlivé typy **JOIN** si ukážeme na tabuľkách, ktoré boli definované v časti 3.2. Tieto tabuľky obsahujú údaje, ktoré sú znázornené Tabuľka 3, Tabuľka 4 a Tabuľka 5. Príkazy pre naplnenie v rámci SQL sa nachádzajú v časti 3.5.1.



Tabuľka 3. Ukážka dát tabuľky players

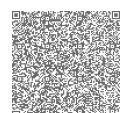
id	name	surname	salary	date_of_birth
100	Joe	Sakic	100 000	7/7/1969
200	Patrice	Bergeron	90 000	7/24/1985
300	Auston	Matthews	120 000	9/17/1997

Tabuľka 4. Ukážka dát tabuľky teams

id	name	stadium_name
5	Colorado	Ball Arena
6	Boston	Boston Garden
7	Toronto	Air Canada Centre

Tabuľka 5. Ukážka dát tabuľky player_seasons

id	player_id	team_id	valid_from	valid_to
1	100	5	2022	2023
2	100	5	2023	2024
3	200	6	2022	2023



3.9.1 CROSS JOIN

Tento typ **JOIN** predstavuje karteziánsky súčin medzi tabuľkami `tabuľka1 × tabuľka2` a neuvažuje sa o použití podmienky pre definovanie, ako má byť uskutočnené spojenie. Pre spojenie tabuliek *players* a *player_season* vyzerá dopyt nasledovne:

```
SELECT *  
FROM players  
CROSS JOIN player_seasons
```

Ukážka 25. Všeobecná definícia **CROSS JOIN** pomocou klauzuly **JOIN**

Je tiež možné použiť zápis:

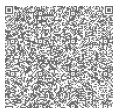
```
SELECT *  
FROM players, player_seasons
```

Ukážka 26. Všeobecná definícia **CROSS JOIN** pomocou klauzuly **FROM**

Obidva dva uvedené zápisy reprezentujú **CROSS JOIN**, ktorý vráti pre uvedené dáta v tabuľkách deväť záznamov – každý záznam z tabuľky *players* je spojený s každým záznamom z tabuľky *player_seasons*. V prípade, že existencie **WHERE** podmienky `player.id = player_seasons.player_id`, tak vykonávanie by sa správalo ako **INNER JOIN**. V rámci implementácie PostgreSQL optimalizátor dopytov vyhodnotí daný dopyt ako **INNER JOIN** a podľa toho použije algoritmus.

3.9.2 INNER JOIN

Predstavuje najviac používaný typ **JOIN**, ktorý vracia prienik spoločných záznamov, ako je znázornené na Obr. 6. Je potrebné, aby v rámci klauzuly **ON** bola definovaná podmienka, na základe ktorej je uskutočnené spojenie. Uvažujme o príklade kedy chceme zistiť všetkých hráčov, ktorý odohrali nejakú sezónu s tým, že vo výpise budú aj informácie o každej jeho sezóne. V rámci nášho zjednodušeného modelu troch tabuliek, je potrebné pre záznamy



z tabuľky *players* vybrať všetkých hráčov, ktorý majú záznam aj v tabuľke *player_seasons*. SQL dopyt na získanie týchto dát by vyzeral nasledovne:

```
SELECT pl.name,pl.surname, ps.team_id, ps.valid_from, ps.valid_to
FROM players AS pl
JOIN player_seasons AS ps
ON pl.id = ps.player_id;
```

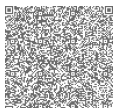
Ukážka 27. Dopyt pre výpis hráčov, ktorý odohrali nejakú sezónu

Výsledok tohto dopytu je ukázaný v tabuľke Tabuľka 6. Ako je vidieť boli vrátené iba tri záznamy oproti deviatim záznamom v prípade typu **CROSS JOIN**. Záznam hráča s *id* = 300 s *name* = Auston Matthews nie je zobrazený, pretože neexistuje žiaden záznam v tabuľke *player_seasons*.

Tabuľka 6. Výsledok dopytu pre INNER JOIN pre nájdenie všetkých hráčov, ktorý odohrali nejakú sezónu

name	surname	team_id	valid_from	valid_to
Joe	Sakic	5	2022-09-01	2023-06-30
Joe	Sakic	5	2023-09-01	2024-06-30
Patrice	Bergeron	6	2022-09-01	2023-06-30

V rámci výsledku SQL dopytu je vidieť *id* tímu, za ktorý odohrali hráči sezónu, ale nie je zrejmé aký je názov tímu. *Id* tímu nám slúži na jednoznačné identifikovanie záznamu v tabuľke *teams* a z pohľadu koncového používateľa nepredstavuje pridanú informáciu. Aby sa vo výsledku zobrazil názov tímu, za ktorý daný hráč nastúpil v sezóne je potrebné prepojiť okrem tabuliek *players*, *player_seasons* aj tabuľky *player_seasons* a *teams*. Pretože v predchádzajúcom dopyte máme už prepojené tabuľky *players* a *player_seasons*, tak nám zostáva výsledok tohto prepojenia spojiť s tabuľkou *teams*. Výsledný SQL dopyt je:



```

SELECT pl.name, pl.surname, ps.team_id, t.name AS team_name,
       ps.valid_from, ps.valid_to
FROM players AS pl
JOIN player_seasons AS ps
  ON pl.id = ps.player_id
JOIN teams AS t
  ON t.id = ps.team_id;

```

Ukážka 28. Dopyt pre výpis hráčov, ktorý odohrali nejakú sezónu spolu s názvom tímu

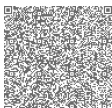
Za prvým spojením tabuliek nasleduje ďalšia klauzula **JOIN**, ktorá obsahuje spojenie s tabuľkou *teams* na základe parametrov `t.id = ps.team_id`. V rámci dopytu boli využité aliasy napr. pre atribút *name* z tabuľky *teams* bol použitý názov *team_name*, aby bolo rozlíšené meno tímu od mena hráča, pretože v oboch tabuľkách je použitý názov *name*. Výsledok dopytu spolu s názvom tímu je zobrazený v Tabuľka 7.

Tabuľka 7. Výsledok dopytu pre **INNER JOIN** nad tromi tabuľkami (*players*, *player_seasons*, *teams*)

name	surname	team_id	team_name	valid_from	valid_to
Joe	Sakic	5	Colorado	2022-09-01	2023-06-30
Joe	Sakic	5	Colorado	2023-09-01	2024-06-30
Patrice	Bergeron	6	Boston	2022-09-01	2023-06-30

3.9.3 LEFT OUTER JOIN a RIGHT OUTER JOIN

Predstavuje ďalší typ **JOIN** operácie, ktorý však nevyberá prekryv záznamov dvoch tabuliek, ale vyberá kompletne jednu tabuľku a pripája k nej záznamy z druhej tabuľky. To, ktorá tabuľka je obsiahnutá celá rozhoduje, či je použitý **LEFT JOIN** alebo **RIGHT JOIN**. Všeobecné spojenie tabuliek pre **LEFT JOIN** je nasledovné:



```
... FROM tabuľka_1 AS t1
LEFT JOIN tabuľka_2 AS t2
      ON t1.id = t2.tabuľka_1_id
```

Ukážka 29. Všeobecná definícia spojenia tabuliek pre typ **LEFT JOIN**

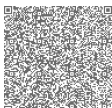
Predchádzajúci dopyt pre **LEFT JOIN** dá rovnaký výsledok ako dopyt:

```
... FROM tabuľka2 AS t2
RIGHT JOIN tabuľka_1 AS t1
      ON t1.id = t2.tabuľka_1_id
```

Ukážka 30. Všeobecná definícia prepísania spojenia tabuliek z **LEFT JOIN** na **RIGHT JOIN**

To, či je tabuľka označovaná ako **LEFT** alebo **RIGHT**, určuje poradie, ako sú zapísané. Z pohľadu syntaxe sa prvá tabuľka uvádza v rámci klauzuly **FROM** a táto tabuľka je označovaná ako **LEFT**. Tabuľka v rámci klauzuly **ON** je označovaná ako **RIGHT**. Keby boli tieto príkazy zapísané v rámci jedného riadku, tak klauzula **FROM** by sa nachádzalo naľavo a klauzula **ON** napravo. V rámci prvého príkladu je použitý **LEFT JOIN**, kde tabuľka *tabuľka_1* sa nachádza na ľavej strane a predstavuje teda **LEFT** tabuľku. Tabuľka *tabuľka_2* predstavuje **RIGHT** tabuľku. Pretože je použitý **LEFT JOIN**, tak sa zoberú všetky záznamy z *tabuľka_1*, pre ktoré sa hľadajú záznamy z *tabuľka_2*. V prípade, že nie je nájdený záznam, ktorý by vyhovoval podmienke spojenia, sú pre tento záznam doplnené hodnoty **NULL** v závislosti od počtu stĺpcov, ktoré majú byť zobrazené z *tabuľka_2*. V prípade druhého príkladu, je poradie tabuliek vymenené a teda *tabuľka_2* sa stáva **LEFT** tabuľkou a *tabuľka_1* je **RIGHT** tabuľkou. Keďže prišlo k zmene z **LEFT JOIN** na **RIGHT JOIN**, výsledok je rovnaký ako v prvom príklade v dôsledku toho, že teraz sú zobrazené všetky záznamy z **RIGHT** tabuľky, pre ktoré sú pridávané informácie z **LEFT** tabuľky.

Uvažujme o príklade, kde chceme vypísať všetkých hráčov bez ohľadu na to, či odohrali nejakú sezónu. V prípade, že odohrali nejakú sezónu, tak vo výpise budú všetky hráčove sezóny. SQL dopyt pre takúto úlohu môže vyzeráť nasledovne:



```

SELECT pl.name,pl.surname, ps.team_id, ps.valid_from, ps.valid_to
FROM players AS pl
LEFT JOIN player_seasons AS ps
ON pl.id = ps.player_id;

```

Ukážka 31. Príklad pre výpis všetkých hráčov so sezónami bez ohľadu na počet záznamov v tabuľke *player_seasons*

Výstup tohto dopytu je znázornený v Tabuľka 8. Oproti príkladu pre **INNER JOIN** pribudol záznam s hráčom *Auston Matthews*, ktorý má pre atribúty z tabuľky *player_seasons* nastavené hodnoty *NULL*, pretože táto tabuľka neobsahuje žiaden záznam pre tohto hráča.

Tabuľka 8. Výsledok dopytu pre *LEFT JOIN* pre nájdenie všetkých hráčov spolu s ich sezónami

name	surname	team_id	valid_from	valid_to
Joe	Sakic	5	2022-09-01	2023-06-30
Joe	Sakic	5	2023-09-01	2024-06-30
Patrice	Bergeron	6	2022-09-01	2023-06-30
Auston	Matthews	NULL	NULL	NULL

V prípade, žeby nás zaujímali hráči, ktorí nedohrali žiadnu sezónu, je možné použiť rovnaký dopyt ako v predchádzajúcej úlohe s tým, že je rozšírený o nasledujúcu podmienku:

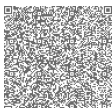
```

... WHERE ps.player_id IS NULL;

```

Ukážka 32. Transformovanie spojenia typu *LEFT JOIN* na *ANTI-JOIN*

Do podmienky je pridaný atribút z tabuľky *player_seasons*, podľa ktorého je možné odfiltrovať záznamy. Teoretický je možné použiť hociktorý atribút, ale v prípade, že sa použije atribút, ktorý môže obsahovať hodnotu *NULL*, tak dopyt môže vrátiť aj záznamy, ktoré by nemali byť vo výstupe. Je preto dôležité používať atribút, ktorým je modelovaný vzťah medzi tabuľkami. V tomto prípade je to atribút *player_id*, čo je cudzí kľúč na tabuľku *players*. Tento dopyt spadá do typu **ANTI-JOIN**, pretože vo výsledku majú byť záznamy, ktoré nemajú žiaden prienik so záznamami z tabuľky *player_seasons*. Samotný databázový systém na pozadí použije algoritmus pre **ANTI-JOIN** pre optimalizovanie vykonávania.



3.9.4 FULL OUTER JOIN

Posledným typom spájania tabuliek je typ **FULL JOIN**, ktorý pokrýva všetky záznamy z obidvoch tabuliek vrátane prekryvaniu sa týchto tabuliek. Všeobecný dopyt pre **FULL JOIN** vyzerá nasledovne:

```
... FROM tabuľka_1 AS t1
FULL JOIN tabuľka_2 AS t2
      ON t1.id = t2.tabuľka_1_id
```

Ukážka 33. Všeobecná definícia spojenia tabuliek pre typ FULL JOIN

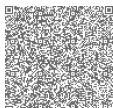
Uvažujme o príklade, kde je potrebné vypísať všetky tímy a všetkých hráčov s odohratými sezónami spolu s tímom, za ktorý odohrali danú sezónu. V prípade, že hráč neodohral sezónu, tak vo výsledku bude uvedený rovnako aj tím, ak neexistuje hráč, ktorí by nastúpil za daný tím. Dopyt pre takýto príklad by vyzeral nasledovne:

```
SELECT pl.name, pl.surname, ps.team_id, t.name AS team_name,
       ps.valid_from, ps.valid_to
FROM   players AS pl
FULL JOIN player_seasons AS ps
      ON pl.id = ps.player_id
FULL JOIN teams AS t
      ON t.id = ps.team_id;
```

Ukážka 34. Dopyt pre získanie všetkých tímov a hráčov spolu s odohratými sezónami

Pre tento dopyt je vrátených päť záznamov s tým, že okrem nájdeného hráča bez sezóny *Auston Matthews* je nájdený aj tím *Toronto*, za ktorý nehral žiaden hráč. Všetky hodnoty okrem názvu tímu pre tento záznam sú nastavené na hodnotu NULL. Pretože v rámci tohto dopytu pracujeme s tromi tabuľkami, je možné rovnaký výsledok dosiahnuť aj pomocou kombinácie **LEFT JOIN** a **RIGHT JOIN**.

Uvažujme, že existujú prepojené dve tabuľky pomocou jedného cudzieho kľúča, ktorý môže obsahovať NULL hodnotu. Pre tieto tabuľky chceme získať všetky záznamy, ktoré sú, ale aj nie



sú prepojené medzi týmito dvomi tabuľkami. Pre tento príklad je potrebné použiť **FULL JOIN** oproti príkladu s prepojením troch tabuliek, kde je možné uplatniť kombináciu **LEFT JOIN** a **RIGHT JOIN**.

Pre získanie všetkých záznamov ktoré nemajú prekryv (typ **ANTI-JOIN**) je potrebné odfiltrovať záznamy, ktoré sú prepojené. To je docielené pomocou hľadania záznamov, kde sú hodnoty pre atribúty prvej tabuľky nastavené na NULL (neexistuje prepojenie a nie je teda možné nastaviť konkrétne hodnoty) a atribúty druhej tabuľky obsahujú hodnoty. Rovnako to musí platiť aj naopak, kde druhá tabuľka ma nastavené NULL hodnoty pre atribúty (neexistuje prepojenie a nie je teda možné nastaviť konkrétne hodnoty) a prvá tabuľka obsahuje hodnoty. Všeobecné definovanie **ANTI-JOIN** vyzerá nasledovne:

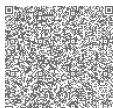
```
... FROM tabuľka_1 AS t1
FULL JOIN tabuľka_2 AS t2
      ON t1.id = t2.tabuľka_1_id
WHERE t1.id IS NULL OR t2.id IS NULL;
```

Ukážka 35. Transformovanie spojenia typu FULL JOIN na ANTI-JOIN

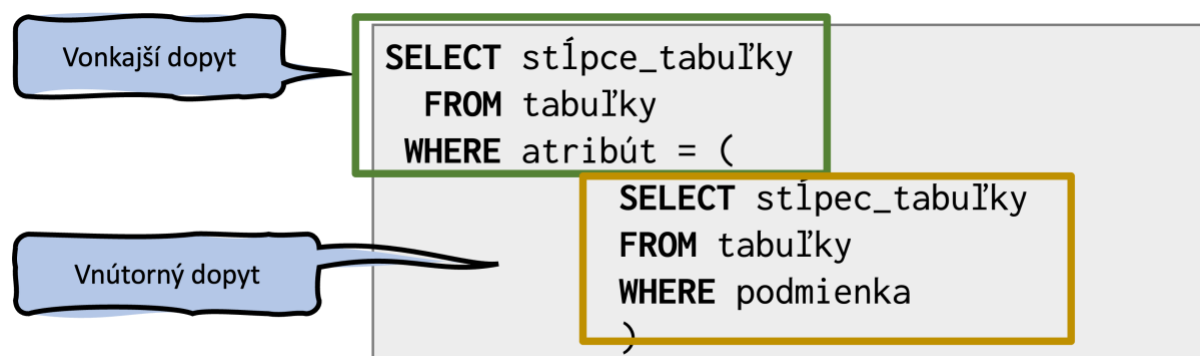
3.10 Vnorené dopyty

Samotný SQL dopyt môže obsahovať v rámci seba ďalšie dopyty, ktoré nazývame vnorené dopyty. Počet vnorených dopytov nie je nijako obmedzený a ich umiestnenie môže byť v rámci klauzul **SELECT**, **FROM**, **WHERE**. Ak dopyt obsahuje vnorený dopyt, tak sa používajú pojmy **vonkajší dopyt** a **vnútorný dopyt** pre rozlíšenie jednotlivých dopytov. Tieto dva typy sú znázornené na Obr. 7. Vonkajší dopyt je označovaný dopyt, ktorý obsahuje v rámci seba iný dopyt. Vnútorný dopyt je označovaný dopyt, ktorý je vložený do iného dopyt, za účelom získania nejakej informácie, ktorá má dopomôcť k riešeniu úlohy.

Je však dôležité aby výsledok vnútorného dopytu korešpondoval s očakávaným výstupom vonkajšieho dopytu. Všeobecný príklad znázornený na Obr. 7, obsahuje vnútorný dopyt v rámci klauzuly **WHERE**, kde atribút z vonkajšieho dopytu porovnáva hodnotu z výstupu vnútorného dopytu. Pretože vieme porovnávať iba dve hodnoty, tak je potrebné, aby vnútorný dopyt vrátil iba jednu hodnotu. Ak vráti viac ako jednu hodnotu, tak dopyt hodí



chybu v dôsledku nemožnosti porovnania hodnoty vonkajšieho dopytu s viacerými hodnotami vrátených vnútorným dopytom.

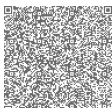


Obr. 7. Znáozornenie vonkajšieho a vnútorného dopytu

Aby sme vedeli pracovať aj s viacerými hodnotami, ktoré vráti vnútorný dopyt, tak je možné použiť k štandardným možnostiam porovnávania nasledujúce rozšírenia:

- **ALL** – vráti hodnotu TRUE, ak všetky hodnoty v rámci vnútorného dopytu (vnoreného dopytu) splnia podmienku.
- **ANY** – vráti hodnotu TRUE, ak niektorá hodnota v rámci vnútorného dopytu (vnoreného dopytu) splnila podmienku.
- **IN** – podobná funkcionálna ako **ANY** s tým rozdielom, že **IN** sa viaže na operáciu zhody = a operácia **ANY** potrebuje definovanie operanda. **IN** vie byť použitý pre skrátenie zápisu v klauzule **WHERE** pre porovnávanie pomocou **OR**.
- **EXISTS** – v prípade, že existuje nejaký záznam v rámci vnútorného dopytu (vnoreného dopytu) vráti hodnotu TRUE.

Vnútorný dopyt môže obsahovať v rámci seba aj atribút z vonkajšieho dopytu a takýto typ vnoreného dopytu sa nazýva korelovaný vnorený dopyt (z *angl. Correlated subqueries*). Celková časová zložitosť v prípade *correlated vnoreného dopytu* je väčšia v dôsledku nutnosti vykonávania vnútorného dopytu pre každý jeden záznam z vonkajšieho záznamu. V prípade, že vnútorný dopyt neobsahuje atribút z vonkajšieho dopytu, tak je postačujúce vykonať iba raz vnútorný dopyt pre získanie hľadanej hodnoty.



Vnorené dopyty sú častokrát používané aj v prípade, keď je možné použitie **JOIN**, ktoré je častokrát efektívnejšie ako použitie vnorených dopytov. Databázový systém vie lepšie optimalizovať **JOIN** ako vnorené dopyty. Pretože nie je limitácia na stupeň vnorenia v rámci používania vnorených dopytov, je vždy potrebné zvážiť ich využívanie, hlavne ak dopyt obsahuje stupeň vnorenia väčší ako štyri.

3.11 Window Functions

Štandardné využitie agregáčnych funkcií vyrába výsledok pre určitú skupinu (množina záznamov) dát definovaných v rámci klauzuly **GROUP BY**. Výsledok je v podobe jedného záznamu pre danú množinu záznamov, čím dochádza k strate informácií o jednotlivých záznamoch. Takéto správanie je postačujúce pre veľké množstvo scenárov, no v niektorých prípadoch nie je postačujúce alebo je málo efektívne. Aby sme neprichádzali o informácie o jednotlivých záznamoch a vedeli presnejšie definovať množinu záznamov, nad ktorou majú byť aplikované výpočty, tak vznikli **window functions** (WF). *Window functions* umožňujú aplikovanie agregáčnych funkcií alebo špeciálnych funkcií viazaných výlučne na WF, tak že výsledok výpočtu je uvedený pre každý záznam. Všeobecná syntax vyzerá nasledovne:

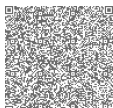
```
SELECT ...,
        názov_funkcie(...) OVER (...)
FROM tabuľka
```

Ukážka 36. Všeobecná syntax pre window functions

Window functions sú definované v rámci klauzuly **SELECT** spolu s ostatnými atribútmi, ktoré majú byť zobrazené vo výstupe. V rámci definovania WF je najprv definovaná agregáčna alebo špeciálna funkcia, ktorá ma byť aplikovaná nad množinou záznamov definovaných v rámci klauzuly **OVER**. Samotný **OVER** definuje, ako sú dáta zoskupené a zoradené. Táto klauzula je tiež niekedy označovaná ako *window*, pretože definuje okno, s ktorým pracuje funkcia.

Pre *window functions* poznáme nasledujúce špeciálne funkcie:

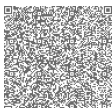
- **RANK()** – vytvorí poradie pre jednotlivé záznamy podľa definovania poradia v rámci klauzuly **OVER**. V prípade, že dva záznamy obsahujú rovnakú hodnotu, tak dostávajú rovnaké poradie a ďalší záznam je posunutý o počet výskytov danej hodnoty. *Napr.*



dva záznamy dostanú priradené poradie jedna, tak ďalší záznam v poradí dostane poradie tri.

- **DENSE_RANK()** – vytvára rovnako poradie ako RANK() s tým rozdielom, že v prípade rovnakého poradia ďalšie poradie pokračuje kontinuálne bez vynechávania poradových čísiel. *Napr. dva záznamy dostanú priradené poradie jedna, tak ďalší záznam v poradí dostane poradie dva.*
- **ROW_NUMBER()** – na základe poradia záznamov pridá každému záznamu unikátnu hodnotu, ktorá korešponduje s číslom riadku v rámci usporiadania záznamov definovaného v **OVER**.
- **PERCENT_RANK()** – vráti relatívnu hodnotu poradia pre záznam, ktorá je vypočítaná ako $(r-1)/(n-1)$, kde **r** je poradie záznamu a **n** je počet záznamov. V prípade že **n = 1**, tak výsledok je **NULL**. Rozsah hodnoty je v rozmedzí 0 až 1.
- **LAG(atribút,n)** – vráti hodnotu atribútu, ktorý sa nachádza **n** pred aktuálnym záznamom podľa definovaného usporiadania v **OVER**.
- **LEAD(atribút,n)** - vráti hodnotu atribútu, ktorý sa nachádza **n** po aktuálnom zázname podľa definovaného usporiadania v **OVER**.
- **NTILE(n)** – funkcia vytvorí **n** sektorov, do ktorého rozdelí záznamy podľa spôsobu zoradenia v klauzule **OVER** a priradí im hodnotu daného sektora. Možnosť využitia v histogramoch alebo v rozdelení na kvartáli.
- **CUME_DIST()** – predstavuje kumulatívnu distribúciu (z angl. *cumulative distribution*) pre záznam, kde výpočet je p/n . kde **p** je počet záznamov, ktoré predchádzajú alebo sú rovné s poradím daného záznamu a **n** je počet záznamov.

V rámci definovania okna v **OVER** sa využívajú tri klauzuly a to **PARTITION BY**, **ORDER BY** a **ROWS**. Prvá klauzula **PARTITION BY** určuje do akých skupín budú dáta rozdelené. Plní rovnakú funkcionality ako **GROUP BY** v rámci štandardných agregácií. Môže obsahovať 1 alebo viacero atribútov, ktoré môžu vstupovať do zoskupenia dát. Druhá klauzula **ORDER BY** predstavuje usporiadanie, ktorého správanie je rovnaké ako v prípade usporiadania výsledku dopytu.

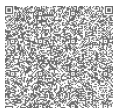


Usporiadanie môže byť zostupne, vzostupne a tiež môže doňho vstupovať viacero atribútov. Tretia klauzula **ROWS** umožňuje definovať aj pohyblivé okno nad zoskupenými a usporiadanými dátami. Správanie **GROUP BY** pracuje s fixným oknom, ktoré je pevne dané na základe zoskupenia dát. V rámci pohyblivého okna vieme pracovať s tým, že budeme pre aktuálny záznam pozeráť **X** záznamov dozadu a **Y** záznamov dopredu. V rámci **ROWS** je možné použiť nasledujúce klauzuly:

- **PRECEDING** – hovorí o počte predchádzajúcich záznamov, ktoré vstupujú do funkcie. Do funkcie môže prísť aj menší počet predchádzajúcich záznamov v prípade, že neexistuje dostatočný počet predchádzajúcich záznamov. *Napr. ROWS 2 PRECEDING berie do úvahy dva predchádzajúce záznamy.*
- **FOLLOWING** – hovorí o počte nasledujúcich záznamov, ktoré vstupujú do funkcie. Do funkcie môže prísť aj menší počet nasledujúcich záznamov v prípade, že neexistuje dostatočný počet nasledujúcich záznamov. *Napr. ROWS 2 FOLLOWING berie do úvahy dva nasledujúce záznamy.*
- **BETWEEN** – pre prepojenie klauzúl **PRECEDING** a **FOLLOWING** za účelom definovania počtu predchádzajúcich a nasledujúcich záznamov. *Napr. ROWS BETWEEN 3 PRECEDING AND 2 FOLLOWING berie do úvahy tri predchádzajúce záznamy a dva nasledujúce záznamy.*
- **CURRENT ROW** – predstavuje aktuálny záznam, ktorý je spracovávaný. Je možné použiť pre ohraničenie okna. *Napr. ROWS BETWEEN 3 PRECEDING AND CURRENT ROW berie do úvahy tri predchádzajúce záznamy po aktuálny záznam.*
- **UNBOUNDED** – pre možnosť definovania neobmedzeného počtu predchádzajúcich a nasledujúcich záznamov. Klauzula je používaná v spojení s klauzulami **PRECEDING** a **FOLLOWING**. *Napr. ROWS UNBOUNDED PRECEDING berie do úvahy neobmedzený počet predchádzajúcich záznamov.*

3.12 View

V rámci databázy nie je vždy žiadúce, aby všetci používatelia mali prístup ku všetkým dátam, ktoré sú uložené. Niektoré informácie sú zaujímavé pre účtovníka, ale bežný zamestnanec



nemusí disponovať prístupom k nim. V rámci databázového systému je možné nastaviť pre rôznych používateľov rôzne prístupy pre tabuľky. Nie je to však možné robiť na úrovni atribútov. Riešením tohto problému je použitie **VIEW**, ktorý umožňuje výsledok dopytu uložiť do „virtuálnej“ tabuľky. V prípade štandardného **VIEW** je uložený iba dopyt, ktorý sa má vykonať a nie je uložený samotný výsledok. Vykonávanie dopytu uloženého pre **VIEW** je uskutočnené zakaždým, keď databázový systém vykonáva dopyt, ktorý obsahuje **VIEW**. Uvažujme príklad, kde by sme chceli vypísať mená a priezviska všetkých hráčov, ktorých máme uložených v databáze. Dopyt vyzerá nasledovne:

```
SELECT name, surname
FROM players;
```

Ukážka 37. Výpis mena a priezviska pre všetkých hráčov

Z bezpečnostných dôvodov však nechceme aby používateľ, ktorý má prístup do databázy, mal prístup aj k atribútu *date_of_birth*. Je možné vytvoriť **VIEW**, ktorý bude poskytovať iba údaje z dopytu znázorneného vyššie. Všeobecná syntax pre vytvorenie **VIEW** je nasledujúca:

```
CREATE VIEW názov_view
AS dopyt_pre_získanie_dát
```

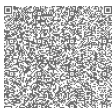
Ukážka 38. Všeobecná syntax pre vytvorenie VIEW

Vytvorenie *view*, ktorý vráti mená a priezviska všetkých hráčov, vyzerá nasledovne:

```
CREATE VIEW players_name AS
SELECT name, surname
FROM players;
```

Ukážka 39. Vytvorenie VIEW pre mená a priezviska všetkých hráčov

Použitie **VIEW** je rovnaké ako používanie tabuliek v dopyte. Ak by sme chceli teraz použiť *players_name*, stačí nám ho vložiť do klauzuly **FROM** a vieme získať mená hráčov.



Problém, ktorý nastáva s použitím **VIEW** sú spojené s modifikáciou dát. Ak by sme chceli teraz vložiť záznam do *players_name*, tak jediné parametre, ktoré vieme vložiť sú *name* a *surname*. Avšak záznam nebude vkladáný priamo do *players_name*, ale do tabuľky *players*, z ktorej vznikol daný *view*. Tabuľka *players* však obsahuje aj atribúty *id*, *salary*, *date_of_birth*, ktoré však pri vkladaní dát do *view players_name* nemajú byť ako nastavené. V prípade, že niektoré z týchto atribútov majú obmedzenie *NOT NULL*, tak vloženie záznamu nie je možné.

Okrem bezpečnostných dôvodov má význam použitia *view* aj z pohľadu lepšej čitateľnosti dopytov a ich zjednodušenie. Je si však potrebné uvedomiť, že použitie *view* nijako nezrýchľuje vykonávanie samotných dopytov.

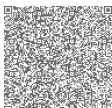
3.12.1 Modifikácia záznamov vo View

Ako už bolo načrtnuté v predchádzajúcej časti, tak modifikácia záznamov v rámci *view* je problémová v dôsledku straty informácií a možného porušenia definovaných integritných obmedzení. Okrem týchto spomenutých problémov existujú aj prípady, kedy nie je možné modifikovať dáta v dôsledku aplikovania operácie pre odstránenie duplícít alebo použitia agregácií nad záznamami.

Ďalším problémom pre modifikáciu dát je, ak *view* je vytvorený z prepojenia viacerých tabuliek. Uvažujme o príklade, kde by sme chceli vypísať mená hráčov spolu s tímom a časom odkedy do kedy hral daný hráč za daný tím. Definícia *view* by vyzerala nasledovne:

```
CREATE VIEW players_teams AS
SELECT pl.name, pl.surname, t.name AS team_name,
       ps.valid_from, ps.valid_to
FROM players AS pl
JOIN player_seasons AS ps
  ON pl.id = ps.player_id
JOIN teams AS t
  ON t.id = ps.team_id;
```

Ukážka 40. Definovanie *view* pre zoznam hráčov spolu s tímom, za ktorý hrali a časom nastupovania za daný tím



Pri vkladani záznamu do vyššie uvedeného *view players_teams* môže nastať prvý problém v prípade, že primárne kľúče nie sú generované automaticky, pretože vo *view* nie sú uvedené, čo by vyvolalo porušenie obmedzenia *NOT NULL* pre atribút *id*. Ďalší problém vloženia záznamu je ten, že ak by sa nám aj podarilo vložiť záznam do tabuliek, tak v rámci tabuľky *player_seasons* by boli vložené iba atribúty *valid_from* a *valid_to*. Atribúty *player_id* a *team_id* by neboli vložené, pričom tieto atribúty prepájajú záznamy medzi tabuľkami *players* a *teams*. Z logického hľadiska by takéto vloženie nedávalo význam, pretože by neboli vytvorené vzťahy medzi tabuľkami a záznam vloženého hráča by nikdy nemal vzťah s tímom, z ktorým mal mať vytvorený vzťah v rámci relačného modelu.

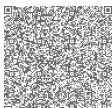
Aby bolo zamedzené nesprávnej modifikácii dát tak existujú všeobecné podmienky kedy je možné modifikovať *view* a sú to:

- *view* obsahuje v rámci klauzuly **FROM** iba jednu tabuľku
- klauzula **SELECT** obsahuje iba mena atribútov a žiadne expresné výrazy, agregácie alebo odstránenie duplicit (**DISTINCT**)
- atribúty, ktoré nie sú uvedené v **SELECT** klauzule môžu byť nastavené na hodnotu NULL a teda nevyvolajú podmienku porušenia integritných obmedzení, a rovnako nie sú súčasťou primárneho kľúča
- dopyt neobsahuje klauzuly **GROUP BY** a **HAVING**

V rámci modifikácie dát vo *view* je možné vložiť alebo aktualizovať dáta vo *view*, ale vo *view* by sa nikdy nezobrazili. Ak by *view* obsahoval podmienku pre zobrazenie záznamov, ktoré sú väčšie ako hodnota **X**, tak v prípade vloženia záznamu, ktorého hodnota je menšia ako **X**, tak takýto záznam bude vložený do tabuľky, ale pri použití *view* nebude zobrazený. Pre zamedzeniu vkladania záznamov do *view*, ktoré nebudú zobrazené v rámci *view* je možné použiť klauzulu **WITH CHECK OPTION** pri definovaní *view*. V tomto prípade nebude umožnené vložiť záznamy, ktoré sa nezobrazia aj v samotnom *view*.

3.12.2 Materialized View

Okrem *view*, ktoré je uložené len ako objekt s definovaným dopytom, je možné využiť aj **materialized view**, ktoré je uložené na disku so všetkými výsledkami. Tento typ *view* má



výhodu v rýchlejšom získavaní výsledku, pretože už nie je nutné uskutočňovať pôvodný dopyt pre získanie výstupu pre *view*. Zmysel použitia *materialized view* je v prípade vykonávania agregáčnej funkcie nad veľkým množstvom záznamov alebo pri veľmi častom využívaní *view* aplikáciou.

Pretože *materialized view* je uložený na disku aj s výsledkami, tak je potrebné zabezpečiť, že v prípade modifikácie dát v pôvodných tabuľkách sa tieto zmeny prenesú aj do *materialized view*. Samotná aktualizácia dát vo *view* môže prebiehať nasledujúcimi spôsobmi:

- okamžite po modifikácii dát
- po požiadavke na prístup daného *view*
- periodicky – v tomto prípade hrozí neaktuálnosť dát vo *view*, a preto nie je vhodné použiť tento typ, ak aplikácia vyžaduje aktuálnosť dát

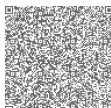
Niektoré DBMS umožňujú vybrať si spôsob ako bude *materialized view* udržiavaný. Pri využívaní tohto typu *view* je potrebné brať do úvahy dodatočné požiadavky na úložisko a tiež na réžiu, ktorá plynie z potreby udržiavať záznamy aktuálne v rámci *materialized view*.

3.13 Funkcie a procedúry

V rámci SQL je možné okrem štandardných funkcií (napr. výpočet priemeru) definovať vlastné funkcie. Okrem toho je možné definovať aj vlastné procedúry. Prehľad základných vlastností pre procedúry a funkcie je obsiahnutá v rámci Tabuľka 9.

Tabuľka 9. Prehľadová tabuľka popisu vlastností pre procedúry a funkcie

Procedúra	Funkcia
Môže vrátiť nula, jednu alebo viacero hodnôt	Musí vrátiť hodnotu (skalár, tabuľku)
Možnosť použitia transakcie v rámci procedúry	Nie je možné použiť transakciu v rámci funkcie



Môže mať vstupné a výstupné parametre	Obsahuje iba vstupné parametre
Možnosť volania funkcií z procedúry	Nie je možné volať procedúru z funkcie
Nie je možné použiť v rámci klauzúl SELECT, WHERE, HAVING	Možnosť použiť v rámci klauzúl SELECT, WHERE, HAVING
Možnosť modifikovania objektov v databáze	Nie je možné modifikovať objekty v databáze

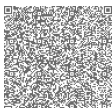
Výhoda použitia funkcií a procedúr je možnosť presunutia biznis logiky bližšie k databáze a nie je ju teda potrebné vykonávať v rámci aplikácie. Hlavne v prípade, ak viacero aplikácií používa rovnakú funkcionality, tak je postačujúce ju modifikovať iba na strane databázy a nie v každej aplikácii zvlášť. V prípade funkcií a procedúr je však potrebné vždy zvážiť dopad na výkonnosť systému hlavne v prípade, ak sa vykonáva komplexná funkcionality nad veľkým množstvom záznamov.

Samotný štandard SQL definuje syntax pre vytváranie funkcií a procedúr no samotné databázové systémy využívajú vlastné verzie syntaxe (*Oracle - PL/SQL, Microsoft SQL server TransactSQL, PostgreSQL - PL/pgSQL*). Princíp fungovania však zostáva rovnaký.

Uvažujme o funkcii, ktorá má slúžiť na spočítavanie bodov hráča z počtu gólov a asistencií. Definovanie takejto funkcie v rámci implementácie PostgreSQL vyzerá nasledovne:

```
CREATE FUNCTION points_calculation (goals integer, assists integer)
  RETURNS integer
  AS $$
  BEGIN
    RETURN goals + assists;
  END; $$
LANGUAGE PLPGSQL;
```

Ukážka 41. Funkcia pre výpočet bodov hráča (góly + asistencie) v PostgreSQL



Funkcia je vytvorená pomocou klauzuly **CREATE FUNCTION**, za ktorou nasleduje názov funkcie. V tomto prípade je to *points_calculation*. Následne sú definované vstupné parametre pre funkciu a sú to dva parametre a to *goals* a *assists*, ktoré sú typu *integer*. Okrem vstupných parametrov je potrebné definovať aj to, aký výstup vracia táto funkcia. Pre tento príklad je to typ *integer*. Nasleduje telo funkcie, ktoré začína ohraničením **AS \$\$ BEGIN** a končí **END; \$\$**. V rámci tela funkcie sa nachádza vrátenie súčtu vstupných parametrov. Na záver definície funkcie je uvedený jazyk, v ktorom bola napísaná funkcia. V tomto prípade to bolo PL/pgSQL.

Uvažujme, že máme tabuľku účtov *accounts*, ktorá obsahuje zostatky na účtoch. Chceme vytvoriť procedúru, ktorá automaticky uskutoční prevod peňazí medzi dvomi účtami. Definovanie procedúry v rámci implementácie PostgreSQL vyzerá nasledovne:

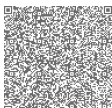
```
CREATE PROCEDURE transfer (sender integer, recipient integer,
amount integer)
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE accounts
        SET balance = balance - amount
        WHERE account_id = sender;

    UPDATE accounts
        SET balance = balance + amount
        WHERE account_id = recipient;
END;
$$

CALL transfer (3, 5, 1000);
```

Ukážka 42. Definovanie procedúry pre uskutočnenie prevodu peňazí medzi dvomi účtami

Procedúra je vytvorená pomocou klauzuly **CREATE PROCEDURE**, ktorá obsahuje definovanie názvu procedúry a parametrov pre danú procedúru. Ostatné časti sú rovnaké ako v prípade definovania funkcie a to definovanie jazyka, v ktorom je písaná procedúra a samotné telo



procedúry, kde je uskutočnené vykonávanie. V tomto príklade telo procedúry obsahuje dve modifikácie záznamov v tabuľke *accounts* a to pre odosielateľa peňazí odpočítanie sumy a v prípade príjemcu pripočítanie sumy. Pre zabezpečenie správneho fungovania by bolo potrebné, aby procedúra *transfer* bola vykonávaná ako transakcia a teda mala garantované ACID vlastnosti. Transakcie sú popísané v osobitnej časti týchto skrípt. Zavolanie procedúry sa uskutoční pomocou klauzuly **CALL**, ktorá obsahuje názov procedúry a atribúty.

3.14 Triggers

Trigger alebo *spúšťač* predstavujú príkazy v rámci databázy, ktoré databázový systém vykoná ako vedľajší efekt modifikácie databázy. Samotný *trigger* musí definovať nasledujúce časti:

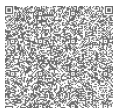
- Kedy ma byť *trigger* vykonaný – táto časť je rozdelená na dve **udalosť** a **podmienku**.
- Akciu – reprezentuje akcie, ktoré sa vykonajú v prípade, že *trigger* sa ma vykonať.

Udalosť určuje pri akom type modifikácie dát dochádza k spusteniu *trigger*. V rámci typu modifikácie dát rozlišujeme operácie:

- vloženie dát – **INSERT**
- modifikácia dát – **UPDATE**
- vymazanie dát – **DELETE**

Okrem samotnej typu modifikácie dát je dôležité tiež definovať, či sa má *trigger* začať posudzovať pred tým ako sa samotná modifikácia uskutoční alebo až po. Ak je *trigger* nastavený, aby sa uskutočnil pred samotnou modifikáciou, je možné opraviť prípadné porušenia referenčnej integrity prípadne iných obmedzení. V prípade, že je *trigger* nastavený pre vykonanie, až po modifikácii dát a dôjde k porušeniu obmedzení, tak pre *trigger* sa nezačne ani overovať podmienka, pretože databázový systém zamietne danú modifikáciu v dôsledku integritných obmedzení.

V rámci samotnej udalosti je tiež možné definovať, či sa má *trigger* uskutočňovať pre každý modifikovaný záznam osobitne alebo sa vykonanie *trigger* uskutoční nad celým príkazom (z *angl. statement*), ktorý uskutočňuje modifikáciu.



Po definovaní udalosti nasleduje definovanie **podmienky** a v prípade jej splnenia dochádza k vykonaniu samotnej **akcie** nad databázou. V rámci podmienky je možné využívať dopyty nad databázou pre získanie požadovaných informácií. Okrem získavania dát nad databázou vieme pristupovať k novej verzii záznamu alebo k starej verzii záznamu. To či je možné pristupovať k novému alebo starému záznamu závisí od typu modifikácie dát a rozdelenie je nasledovné:

- **INSERT** – umožňuje referovať iba **nový záznam**
- **UPDATE** – umožňuje referovať **nový** a aj **starý záznam**
- **DELETE** – umožňuje referovať iba **starý záznam**

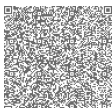
V rámci používania *trigger* je dôležité si dať pozor na vznik cyklov pri spúšťaní samotných *trigger* napr. prvý *trigger* vyvolá spustenie druhého *trigger* a ten vyvolá spustenie tretieho *trigger*, ktorý následne opäťovne spustí prvý *trigger*. Rovnako je dôležité nepoužívať *trigger* na funkcionality, ktorú má v sebe integrovaný samotný databázový systém napr. referenčná integrita.

3.14.1 Syntax

Samotný SQL štandard definuje syntax pre *trigger*, ale väčšina implementácií používa vlastnú syntax. V tejto časti je uvedená všeobecná syntax, ktorej princípy sa využívajú aj v ostatných implementáciách databázových systémoch. Všeobecná syntax pre *trigger*, ktorý sa má vykonať po vložení záznamu do tabuľky vyzerá nasledovne:

```
CREATE TRIGGER názov_trigger AFTER INSERT ON názov_tabuľky
REFERENCING NEW ROW AS alias_pre_nový_záznam
FOR EACH ROW
    WHEN podmienka
BEGIN
    akcia
END;
```

Ukážka 43. Všeobecná definícia trigger pre udalosť po vložení do tabuľky

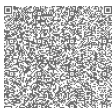


Klauzula **CREATE TRIGGER** definuje vytvorenie *trigger* s definovaným názvom, za ktorým nasleduje definovanie udalosti. V tomto prípade je to udalosť typu **AFTER INSERT ON**, čo znamená že po vložení záznamu do tabuľky sa spustí posudzovanie tohto *trigger*. Nasleduje **REFERENCING NEW ROW AS**, čo je definovanie označenia ako bude referencovaný nový záznam, ktorý bol už vložený do tabuľky. Klauzula **FOR EACH ROW** definuje, že posudzovanie podmienky a prípadná akcia je vykonávaná pre každý záznam, ktorý bolo vložený do tabuľky. V rámci klauzuly **WHEN** je definovaná podmienka, a ak je splnená, tak *trigger* postupuje do vykonania akcie, ktorá je definovaná v rámci bloku **BEGIN** a **END**. V prípade, že podmienka nie je splnená tak akcia nie je vykonaná. Pre iný príklad, kde by sme chceli spustiť *trigger* pred vložením záznam by bolo v rámci typu udalosti namiesto klauzuly **AFTER** použitá klauzula **BEFORE**.

3.14.2 Syntax v PostgreSQL

Definovanie *trigger* v rámci PostgreSQL je mierne odlišné oproti všeobecnej syntaxi a to z toho dôvodu, že je potrebné definovať samostatnú *trigger* funkciu. V tejto funkcii je referovanie na nový alebo starý záznam pomocou výrazov NEW a OLD a obsahuje podmienku spolu s akciou. Funkcia je následne použitá v rámci definovania *trigger*, ktorý obsahuje informáciu, pri akom type udalosti sa vykoná definovaná *trigger* funkcia. Je možné definovať aj viacero udalosti, pri ktorom sa má uskutočniť *trigger*.

Uvažujme príklad, kedy pri vkladaní záznamu do tabuľky alebo jeho aktualizovaní je potrebné overiť, či v danom čase tím má menej ako 21 hráčov. Pre jednoduchosť príkladu budeme porovnávať dátum iba s hodnotou 2022-09-01. Vytvorenie funkcie a *trigger* v PostgreSQL vyzerá nasledovne:



```

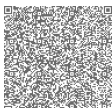
CREATE FUNCTION players_in_team_count_check_trigger() RETURNS trigger
LANGUAGE plpgsql AS $$
BEGIN
    IF (
        SELECT count(*)
        FROM player_seasons
        WHERE team_id = NEW.team_id AND
              valid_from = '2022-09-01') > 20
    THEN
        RAISE EXCEPTION 'Tím môže mať naraz maximálne 20 hráčov';
    END IF;
    RETURN NEW;
END;
$$;

CREATE TRIGGER players_in_team_count_check_trigger
AFTER INSERT OR UPDATE ON player_seasons
FOR EACH ROW EXECUTE PROCEDURE players_in_team_count_check_trigger ();

```

Ukážka 44. Definovanie trigger v PostgreSQL pre overenie počtu hráčov v danom čase

Na začiatku príkladu je potrebné definovať funkciu s názvom, ktorá vracia **trigger** (**RETURNS trigger**) a tiež je potrebné definovať, aký jazyk bude použitý pre definovanie podmienky a akcie v prípade splnenia podmienky. V tomto prípade bol použitý PL/pgSQL a podmienka a akcia je umiestnená medzi **\$\$**. Blok vykonávania obsahuje podmienku **IF**, ktorá obsahuje dopyt na získanie počtu záznamov pre tím pomocou filtra `team_id = NEW.team_id`. Hodnota `NEW.team_id` reprezentuje hodnotu *team_id* pre vložený záznam. Druhá časť podmienky hovorí, že chceme tento počet iba pre záznamy, ktorých atribútu *valid_from* je nastavený na hodnotu `2022-09-01`. Ak je vrátená hodnota dopytu väčšia ako 20, tak je vyvolaná výnimka s popisom. V prípade, že podmienka nie je splnená tak vykonávanie pokračuje ďalej a záznam je pridaný do tabuľky tím, že je vrátený záznam **RETURN NEW**. Po vytvorení funkcie je potrebné ešte definovať samotný **trigger**, ktorý obsahuje názov a typ operácie, pri ktorej sa má vykonať. V tomto prípade je nastavený typ udalosti až po vložení alebo aktualizovaní záznamu **AFTER INSERT OR UPDATE ON**. Aj napriek tomu, že **trigger** je vykonaný až po, tak



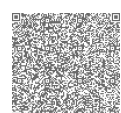
v prípade vyvolania výnimky nie je záznam uložený a databáza je vrátená do pôvodného stavu. Bolo by možné použiť aj **BEFORE**. Záverečný príkaz definuje, že definovaná funkcia je vykonaná pre každý záznam samostatne.

V prípade, že je potrebné vymazať *trigger*, tak je najprv potrebné vymazať samotný *trigger* a až následne je možné vymazať *trigger* funkciu. Vymazanie pre tento príklad je vykonané nasledovne:

```
DROP TRIGGER players_in_team_count_check_trigger ON player_seasons;  
DROP FUNCTION players_in_team_count_check_trigger;
```

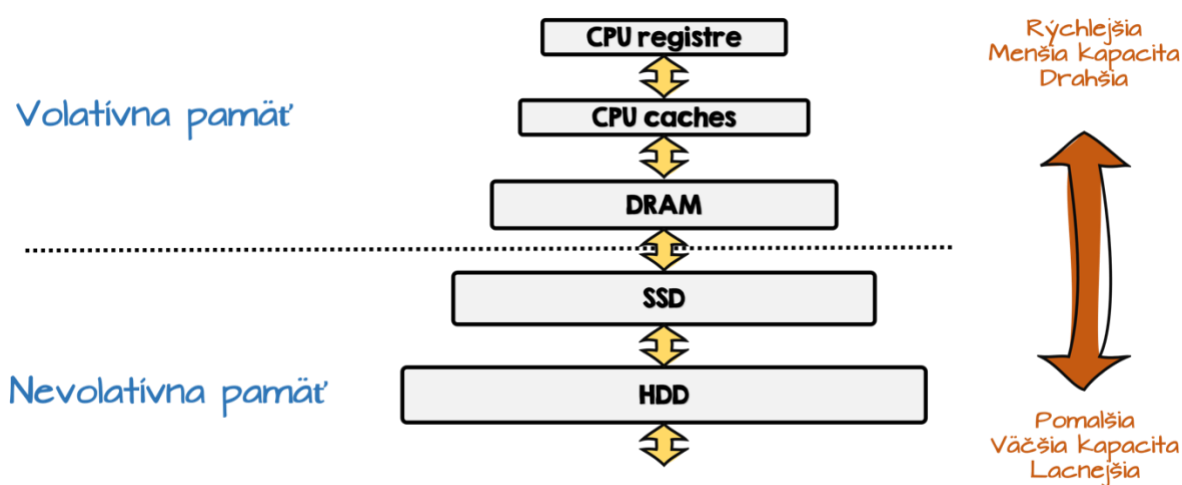
Ukážka 45. Vymazanie trigger

Pre vymazanie *trigger* je potrebné definovať aj nad ktorou tabuľkou je *trigger* definovaný, pretože rovnaký názov môže byť použitý aj pre iné tabuľky.



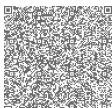
4 Fyzické úložisko

V architektúre počítačov sa využíva veľké množstvo typov pamäti, niektoré z nich stratia informácie bez napájania a iné nie. Pamäte dosahujúce vysokú rýchlosť a nízku odozvu poskytujú malú kapacitu a sú v prepočte na ponúkanú kapacitu drahé. Niektoré z týchto pamäti sú umiestnené priamo v procesore (registre a vyrovnávacia pamäť). Na druhej strane pamäte, ktoré ponúkajú veľké úložisko sú lacnejšie, ale ich rýchlosť je menšia a odozva prístupu je väčšia. Jednotlivé typy pamäti z pohľadu kapacity, rýchlosti a odozvy sú znázornené na Obr. 8.



Obr. 8. Hierarchia pamäti z pohľadu kapacity a rýchlosti

Tabuľka 10 ukazuje približné časy prístupu k jednotlivým typom pamäti (Tieto časy sa môžu líšiť od výrobcu). V rámci tabuľky sú znázornené časy prenesené do vnímania času človeka pre lepšiu predstavu. Reálne časy prístupov k jednotlivým pamätiam sú vynásobené 10^9 , kde hodnota po vynásobení pre L1 pamäť je 0,5s a pre štandardné HDD je približne 16,5 týždňa. Z pohľadu prístupu k dátam a ich spracovaniu je pre databázový systém rozdiel, či potrebuje získať informáciu z hlavnej pamäte alebo potrebuje načítať dáta z disku. Príchodom SSD sa však rýchlosť prístupu zrýchlila oproti magnetickým diskom (HDD).



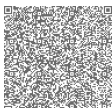
Tabuľka 10. Orientačné časy prístupu pre úrovne pamäti

typ pamäte	čas	Vynasobené 10 ⁹
L1 cache	0,5 ns	0,5s
L2 cache	7 ns	7s
DRAM	100 ns	100 s
SSD	150 000 ns	1,7 dňa
HDD	10 000 000 ns	16,5 týždňov
Network storage	30 000 000 ns	11,4 miesacov
Magnetické pásky	1 000 000 000 ns	31,7 rokov

Z pohľadu výkonnosti databázových systémov nás budú zaujímať dva typy pamäte - a to volatílna a nevolatílna. Nevolatílna pamäť nie je závislá na zdroji napájania oproti volatílnaj pamäti a dáta sú zachované aj v prípade výpadku zdroja. Táto vlastnosť je žiadúca aby dáta boli zachované v prípade výpadku databázových systémov (môžu nastať rôzne typy zlyhaní). Samotné dáta sú uložené v blokoch rovnako aj ich adresácia – blokovo adresovateľné. Je preto potrebné vždy načítanie celého bloku do hlavnej pamäte. Reprezentantom tohto typu pamäte je SSD (Solid-State Disk) alebo HDD (Hard Disk Drive). V prípade volatílnaj pamäte sú dáta stratené v prípade výpadku a prístup k samotným dátam je možné po Bytoch – Bytovo adresovateľné. Tento typ pamäte sa využíva ako hlavná pamäť v rámci počítača.

4.1 Načítavanie a ukladanie dát v DBMS

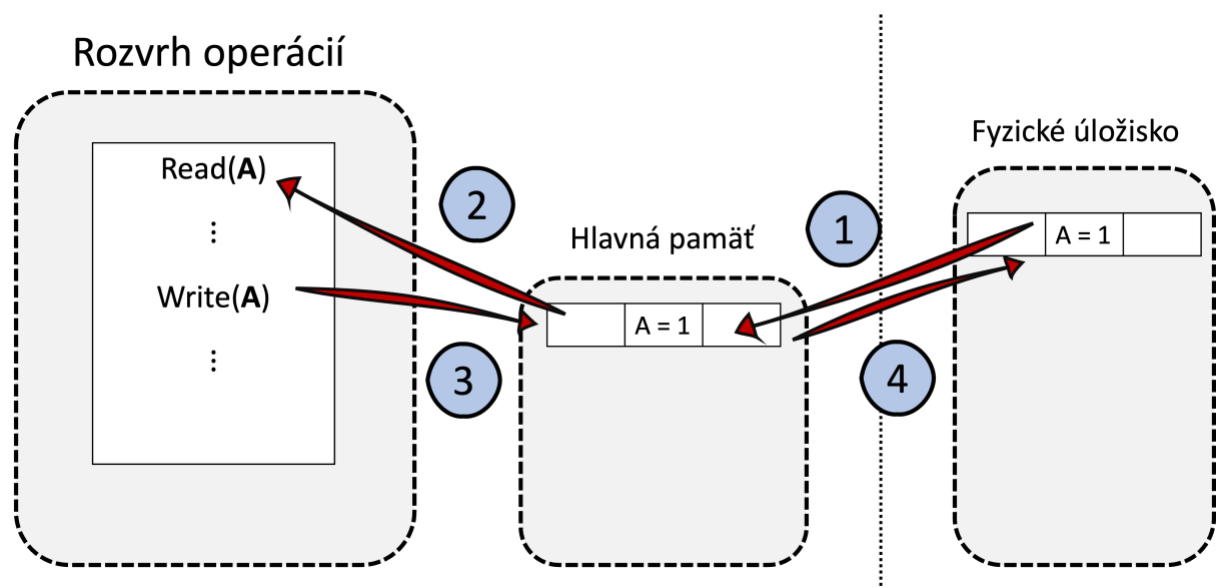
Aby databázový systém mohol pristupovať k samotným dátam, tak tieto dáta musia byť načítané z disku (nevolatílna pamäť – HDD, SSD) do hlavnej pamäte, kde k nim DBMS vie pristupovať a uskutočňovať operácie. Tento proces je znázornený na Obr. 9, kde sú nasledujúce kroky:



1. **Krok** – načítanie dát z disku do hlavnej pamäte. V rámci rozvrhu operácií, sa nachádza požiadavka pre načítanie *objektu A* (*Read(A)*). Databázový systém musí najprv načítať blok dát z fyzického úložiska. Tento blok dát je označovaný ako **stránka**. *Objekt A* sa však nachádza v rámci stránky na disku, ktorý obsahuje aj iné záznamy databázy. Do hlavnej pamäte sú teda načítané aj dáta z iných záznamov, hoci nie sú v danom momente potrebné.
2. **Krok** – keď sa *objekt A* nachádza v hlavnej pamäti, tak DBMS môže pristúpiť priamo k *objektu A* a následne uskutočniť prípadnú modifikáciu, ako je znázornené na obrázku. V tomto prípade databázový systém chce uskutočniť zápis *objektu A* v dôsledku modifikácie.
3. **Krok** – nová hodnota napr. 3 je zapísaná do hlavnej pamäte namiesto pôvodnej hodnoty (obrázok obsahuje pôvodnú hodnotu $A=1$).
4. **Krok** – následne je celá stránka z hlavnej pamäte zapísaná na disk. V tomto momente aj v prípade výpadku dochádza k zachovaniu dát.

V okamihu, keď DBMS už má načítané požadované dáta v hlavnej pamäti, sú samotné operácie vykonávané nad nimi rýchle, pravdaže pokiaľ sa neuskutočňujú náročné operácie nad týmito dátami. Z pohľadu času vykonávania dopytu je najviac ovplyvňujúci faktor práve presun jednotlivých stránok medzi diskom a hlavnou pamäťou. Použitie SSD disku vie výrazne zrýchliť vykonávanie dopytu vzhľadom na rýchlosť prístupu k samotným blokom (viď tabuľka Tabuľka 10).





Obr. 9. Načítanie a ukladanie dát v rámci DBMS

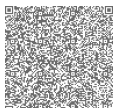
5 Indexy

Cieľom indexov je zrýchlenie nájdenia požadovaných informácií v databáze. Toto zrýchlenie je docielené pridaním dodatočných informácií do databázy za účelom zvýšenia výkonu. Dodatočné informácie umožnia databáze neprehľadávať všetky záznamy v tabuľke, ale vybrať iba tie stránky z pamäte, v ktorých sa nachádza hľadaná informácia. Index v databáze je možné prirovnať ku indexu v knihe, kde index zaberá tiež dodatočné miesto, ale namiesto toho aby bolo nutné čítať celú knihu od začiatku pre nájdenie informácie, tak nájdením informácie v indexe vieme povedať, kde sa nachádza daná informácia v knihe. Oproti knihe sa však databáza neustále modifikuje, a preto je potrebné, aby aj index bol neustále udržiavaný aktuálny.

Vytvorený index v databáze predstavuje perzistentný objekt, ktorý je vytvorený nad jedným alebo viacerými atribútmi (stĺpcami) tabuľky. Tiež je ho potrebné udržiavať aktuálny, pokiaľ dochádza k modifikáciám tabuľky (pridanie, aktualizovanie alebo vymazanie záznamov). Rozhodnutie o použití indexu počas vykonávania dopytu neuskutočňuje používateľ, ale samotný databázový systém (optimalizátor dopytov). Administrátor databázy môže rozhodnúť, aké indexy budú vytvorené. Je preto vhodné poznať správanie používateľov, aké dopyty sú uskutočňované nad databázou a podľa toho nastaviť jednotlivé indexy.

Samotný index neobsahuje uložený celý záznam tabuľky, ale obsahuje iba kľúče, podľa ktorých sa uskutočňuje vyhľadanie a následne odkaz na záznam tabuľky (*record ID*). Tento záznam je potrebné ešte načítať pre získanie ostatných atribútov. Výnimkou je *clustered* index, ktorý obsahuje celý záznam, pretože samotný index je používaný ako celá tabuľka. Takýto index vie byť len jeden v rámci tabuľky a ostatné indexy nad touto tabuľkou obsahujú už *record id*.

Plánovač vykonávania dopytov sa rozhodne pre index v prípade, že jeho použitie docielí rýchlejšie nájdenie výsledku. Použitie indexov je závislé od množstva záznamov, ktoré majú byť nájdené. Ak dopyt má vrátiť veľké množstvo záznamov, tak z pohľadu výkonu je lepšie uskutočniť sekvenčný sken (prehľadanie všetkých záznamov) ako použitie indexu, ktorý spôsobí náhodný prístup na disk, ktorý bude vo výsledku trvať dlhšie ako sekvenčné čítanie disku pre načítanie celej tabuľky.



V rámci databázových systémov existuje veľké množstvo indexov, ktoré je možné použiť. V ďalších častiach sa budeme venovať nasledujúcim indexom:

- **Hash**
- **B+tree**
- **Bitmap**

Existujú aj ďalšie indexy ako sú **inverted**, **trie**, **radix** atď. Samotný PostgreSQL (verzia 15) podporuje indexy: **B+tree**, **Hash**, **GiST**, **SP-GiST**, **GIN** a **BRIN**.

Okrem zrýchlenia vyhľadávania majú indexy aj negatívne aspekty a sú to:

- vyžadujú dodatočné miesto na disku – indexy musia byť uložené na fyzickom úložisku v rámci databázového systému. Dochádza teda k zvýšeniu požiadaviek na samotné úložisko a môžu zaberať veľké množstvo dát (viac ako samotné tabuľky).
- spomalenie výkonnú databázového systému počas modifikácie dát - v prípade nepoužitia indexov je vykonaná iba modifikácia dát a nie je nutné vykonať žiadne dodatočné operácie. Ak sú vytvorené indexy nad tabuľkou, tak databázový systém musí tiež aktualizovať samotný index (údržba indexu), aby obsahoval aktuálne hodnoty.

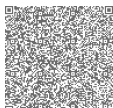
5.1 SQL syntax

Definovania indexov v rámci SQL syntaxu je nasledovné:

```
CREATE INDEX názov_indexu ON tabuľka(atribúty);
```

Ukážka 46. Všeobecná syntax pre definovanie indexu

V rámci implementácie PostgreSQL je možné definovať typ indexu pomocou klauzuly **USING**. V rámci definovania atribútu, ktorý má byť predmetom indexu, je možné tiež pre atribút definovať, ako má byť uskutočnené porovnávanie. V prípade vyhľadávania textu v rámci B+tree indexu je potrebné definovať *pattern* porovnávania pre možnosť vyhľadávania iba



začiatku textu napríklad hľadanie priezviska v atribúte *surname*, ktoré začína na „Ka“. V rámci PostgreSQL definovanie typu indexu vyzerá nasledovne:

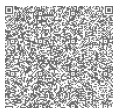
```
CREATE INDEX názov_indexu ON tabuľka(atribút pattern_operator) USING  
typ_indexu;
```

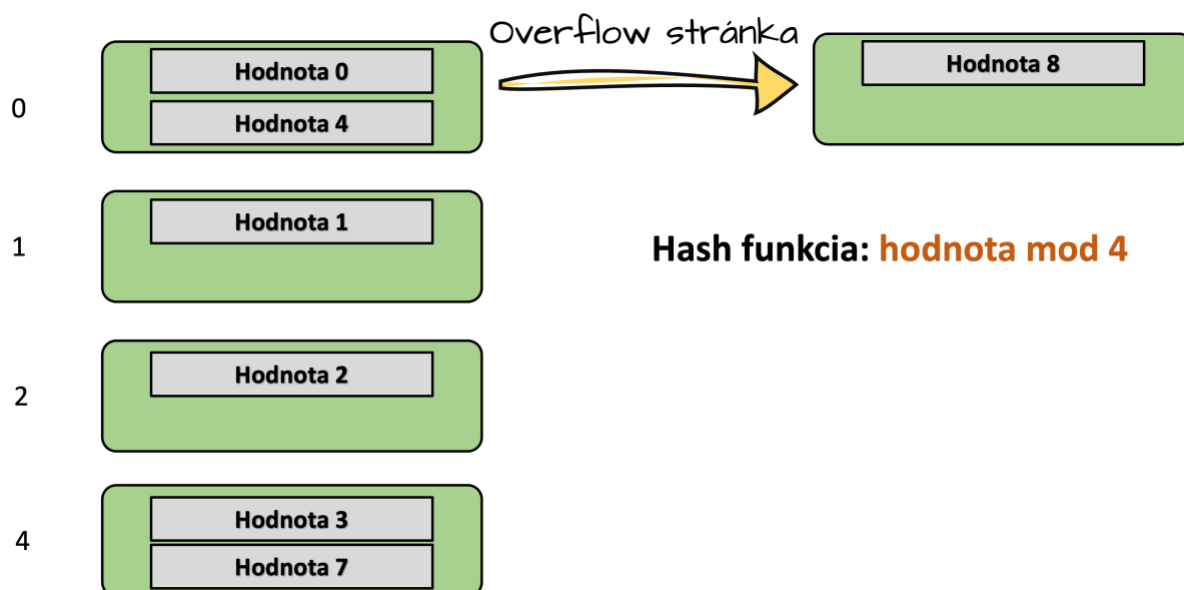
Ukážka 47. Všeobecné vytvorenie indexu v PostgreSQL s definovaním typu indexu

5.2 Hash index

Pre tento typ indexu je potrebné zadať pojem **oblasť** (z *angl. bucket*), do ktorej budú ukladané kľúče a *record id*. Princíp fungovania je založený na hash funkcii, do ktorej vstupuje hľadaný kľúč. Výsledkom je číslo oblasti, v ktorej sa má nachádzať daný záznam (rozumieme *record_id*) alebo v prípade modifikácie záznamu je to, kam má byť daný záznam uložený. Z pohľadu databázového systému je dôležitý výber hash funkcie, pretože nie je potrebné mať kryptografickú hash funkciu (napr. sha-256), ktorá síce nedáva kolízie, ale pre potreby rýchleho vyhľadávania je pomalá. Je dôležité mať rýchlu hash funkciu s primeraným počtom kolízií pre indexované dáta.

Princíp fungovania je znázornený na Obr. 11. V rámci príkladu je použitá hash funkcia *mod 4*. Jednotlivé hodnoty sú uložené do **oblastí**. Každá oblasť môže obsahovať rôzne hodnoty, a preto pri vyhľadávaní záznamu je potrebné ešte vyhľadať kľúč aj v samotných stránkach danej oblasti. V prípade, že prvá stránka oblasti je už plná, tak je potrebné zapisovať ďalšie záznamy na druhú stránku. Prvá stránka bude následne ukazovať na druhú stránku. Pri prehľadávaní danej oblasti je potrebné prejsť všetky zreťazené stránky. Takto zreťazene stránky sa nazývajú *overflow stránky* (z *angl. overflow pages*).



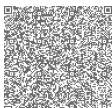


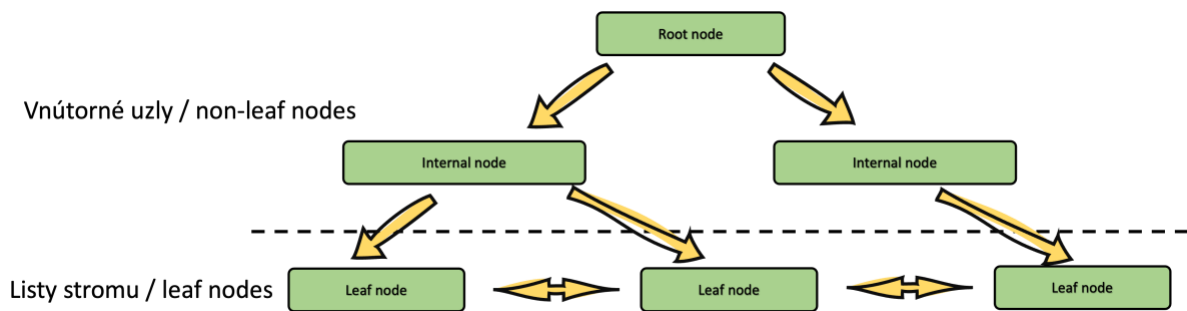
Obr. 10. Princíp fungovania statického hash indexu

Časová zložitosť tohto typu indexu je $O(1)$ + prehľadávanie prípadných *overflow* stránok. Hash index je možné využiť iba pri porovnávaní rovnosti (*hľadaná hodnota = atribút*) vzhľadom na to, že hodnoty sú hash funkciou neusporiadane roz distribuované do oblasti. Nie je možné uskutočniť *range* prehľadávanie (*range search*), ani čiastočné vyhľadávanie napr. poznáme prvú polku slova.

5.3 B+tree index

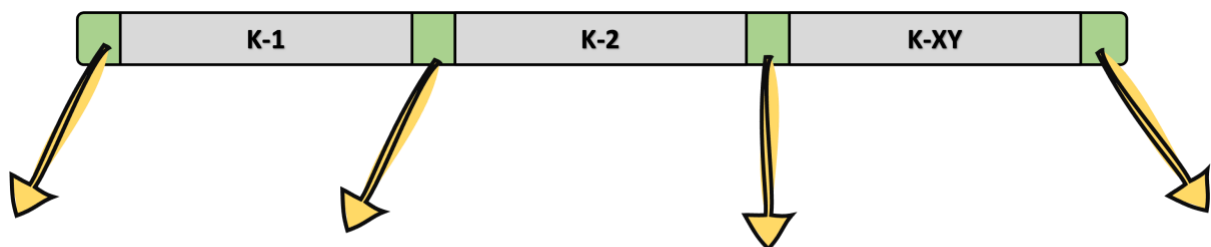
Ďalší typom indexu je B+tree index. V rámci indexu je využívaný B+tree, ktorý predstavuje vyvážený strom a je rozšírením B-tree. Oproti B-tree má uložené dáta iba v listoch, čím je dosiahnuté rýchlejšie vyhľadávanie, jednoduchšie mazanie záznamov a možnosť prepojenia samotných listov. Nevýhodou je prítomnosť redundancie kľúčov. Štruktúra B+tree je znázornená na Obr. 11. V rámci B+tree rozoznávame dva typy uzlov a to vnútorné uzly (z *angl. non-leaf nodes*) a listy stromu (z *angl. leaf nodes*). V rámci vnútorných uzlov je potrebné ešte rozlišovať koreňový uzol (z *angl. root node*), ktorý je len jeden a zvyšku vnútorných uzlov označovaných v angličtine *internal nodes*. V rámci databázových systémov predstavuje uzol stránku v pamäti.





Obr. 11. Štruktúra B+tree

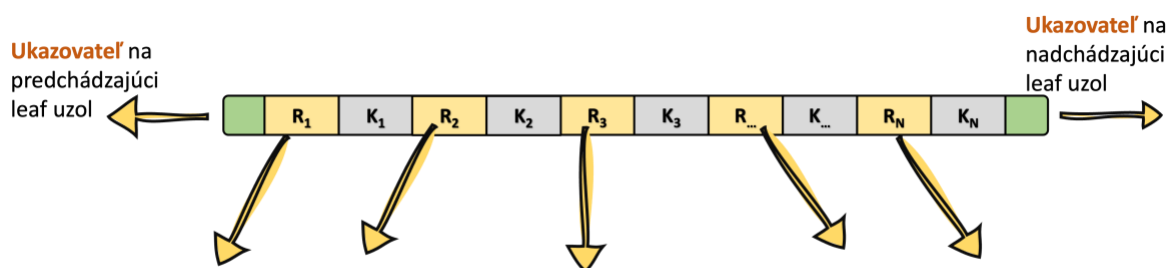
Štruktúra všetkých vnútorných uzlov vrátane *root* uzla je znázornená na Obr. 12. Každý uzol obsahuje ukazovatele na ďalšie uzly (stránky B+tree) a samotné kľúče.



Ukazovateľe (Pointers) na ďalšie stránky B+tree

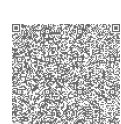
Obr. 12. Štruktúra vnútorných uzlov B+tree

Štruktúra listov je znázornená na Obr. 13, kde môžeme pozorovať rozdielnú štruktúru oproti vnútorným uzlom. Listy obsahujú páry kľúč s *record id*, ktorý odkazuje na konkrétny záznam v tabuľke. Okrem toho obsahuje ešte ukazovatele na predchádzajúci a ďalší list, čo má za následok, že je možné použiť index na uskutočňovať *range* vyhľadávania napr. uskutočnené objednávky medzi dvoma dátumami.



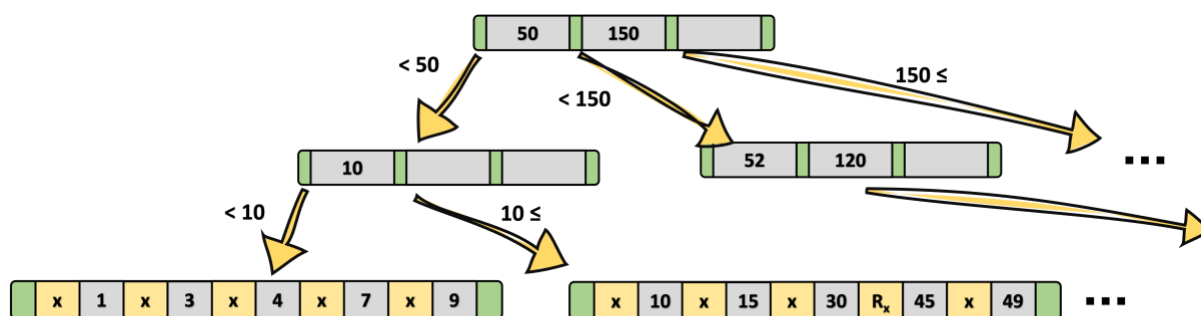
Ukazovateľe (Pointers) na konkrétny záznam tabuľky (record)

Obr. 13. Štruktúra listov v B+tree



Ukážka príkladu pre b+tree index je znázornená na Obr. 14, kde sú použité hodnoty typu *integer* ako kľúč. Uvažujme, že hľadáme záznam, ktorého hodnota je väčšia alebo rovná ako tri a menšia ako 25. Postup v rámci vyhľadávania je nasledovný:

- *Root* uzol – prechádzajú sa jednotlivé hodnoty kľúčov, ktoré obsahuje *root* uzol. V tomto prípade sa porovnáva hodnota v *root* uzle s najmenšou hľadanou hodnotou tri. Porovnanie $3 < 50$ sa vyhodnotí ako pravda a použije sa ukazovateľ pre tento kľúč.
- Vnútorňý uzol – prechádza sa ďalší uzol v hierarchii b+tree. V tomto prípade sa uskutoční porovnanie $3 < 10$, ktoré je opäť vyhodnotené ako pravda. Použije sa ukazovateľ priradený ku kľúču s hodnotou 10. Ak by sme hľadali napr. hodnotu 10 alebo 11, tak sa použije ukazovateľ pre hodnoty väčšie alebo rovne ako 10.
- List stromu - v rámci vyhľadávania sa nachádzame už na úrovni listov. Prechádza sa list a porovnávajú sa kľúče. Keď je objavená hodnota 3, ktorá sa nachádza vo vyhľadávaní, tak ku kľúču je priradený *record id*, ktorý nám odkazuje na konkrétny záznam. Keďže vo vyhľadaní nás zaujímajú záznamy až po menšie ako 25, tak sa pokračuje v prehľadávaní listu. Všetky záznamy vyhovujú podmienke v rámci daného listu. Pretože list odkazuje na ďalší list tak vieme pokračovať vo vyhľadávaní v ďalšom nasledujúcom liste, kde sú hodnotu 10, 15, 30 Hodnoty 10, 15 budú vyhodnotené, že sú menšie ako 25, ale hodnota 30 už nevyhovuje. Na hodnote 30 skončí vyhľadávanie v rámci b+tree indexu.



x – odkaz na Record ID alebo samotné dáta

Obr. 14. Ukážka indexu pomocou b+tree

V rámci vyhľadávania v B+tree vieme okrem vyhľadávania rovnosti vyhľadávať aj rozsahy tak, ako bolo uvedené vo vyššie uvedenom príklade. V rámci tohto typu indexu je možné uskutočniť aj čiastočné vyhľadanie textu, ak poznáme začiatok hľadaného slova, pretože nie je počítaná žiadna hash funkcia a je uskutočňované iba porovnávanie.

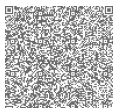
V rámci implementácie PostgreSQL medzi verziami 12 a 13 prišlo k optimalizácií veľkosti B+tree indexu pri duplicitách. Vo verzií 12 boli duplicity riešené kľúč a *record id* a tým dochádzalo k zbytočnej duplicite hľadaného kľúča. V rámci verzie 13 bol tento prístup nahradený spôsobom, že pre jeden kľúč sú uvedené všetky *record id* a nedochádza tak k zbytočnej duplicite kľúčov v rámci listov.

Pre vizualizáciu vkladania, mazania a vyhľadávania v B+tree je možné použiť nasledujúci odkaz².

5.4 Bitmap index

Bitmap index je špeciálny typ indexu, kedy je vytváraná bit mapa pre jeden hľadaný kľúč. Táto bit mapa hovorí, v ktorých záznamoch sa nachádza daný kľúč. Pretože každý kľúč musí obsahovať vlastnú bit mapu, tak tento typ indexu je vhodný pre atribúty, kde počet rôznych hodnôt je malý. Vhodný by bol napríklad pre určenie pohlavia žena alebo muž. Príklad bitmap indexu je znázornený na Obr. 15. V rámci príkladu sú vytvorené bitmap indexy pre atribúty *gender* a *income_level*. Pre atribút *gender* sú vytvorené dva indexy, kde prvý index je pre pohlavie muža a druhý index je vytvorený pre pohlavie ženy. Jednotlivé hodnoty 0 a 1 podľa pozície určujú, či záznam obsahuje daný kľúč alebo nie. Hodnota 1 reprezentuje, že kľúč sa nachádza v danom zázname a hodnota 0 znamená, že kľúč sa nenachádza v zázname. Na príklade indexu pre kľúč muža (hodnota m), sa hodnota 1 nachádza na prvej a štvrtej pozícii, čo korešponduje s tabuľkou, kde táto hodnota je v rámci prvého (*record_number* = 0) a štvrtého (*record_number* = 3) záznamu. V prípade indexu pre atribút *income_level* vidíme až päť indexov, každý pre jednu platovú triedu.

² <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>



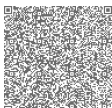
				Bitmap index pre atribút Gender		Bitmap index pre atribút Income_level	
record number	id	gender	income_level				
0	11111	m	L1	m	10010	L1	10100
1	22222	f	L2	w	01101	L2	01000
2	12345	f	L1			L3	00001
3	89879	m	L4			L4	00010
4	12345	f	L3			L5	00000

Obr. 15 Ukážka bitmap indexu

Bitmap index vzhľadom na veľkú réžiu pri vkladaní a mazaní záznamov sa nepoužíva ako samostatný index, ktorý by bol permanentne uložený a udržiavaný. Používa sa vo forme in-memory indexy, ktorý je dočasne vytvorený pre konkrétny dopyt. Význam má pri prehľadávaní indexu, kedy sa očakáva nájdenie väčšieho množstva záznamov, ktoré už nie je efektívne získavať pomocou samotného indexu (veľké množstvo náhodných prístupov), ale nie je dostatočné veľké pre použitie sekvenčného skenu. Počas prehľadávania sa vytvorí bit mapa pre záznamy, ktoré vyhovujú podmienke a následne sú do hlavnej pamäte vytiahnuté iba tie stránky, na ktorých sa nachádzajú vyhovujúce záznamy.

Pokiaľ sú dve bit mapy vytvorené nad rovnakými záznamami, je možné medzi nimi aplikovať logické operácie pre získanie výsledku zložitej podmienky. Napríklad ak by bola *podmienka_1* AND *podmienka_2* a je možné vytvoriť bit mapy pre obidve podmienky, tak pre výsledné bit mapy je možné aplikovať logickú operáciu AND. Ak pre rovnaký záznam je v prvej aj druhej bit mape hodnota 1, výsledok bude vyhodnotený ako pravdivý a záznam je potrebné získať.

PostgreSQL používa popísaný mechanizmus na zrýchlenie vyhľadávania pomocou indexov v prípade nájdenia väčšieho množstva záznamov. Pretože vytvorený bitmap index sa nie vždy musí zmestiť do pamäte, tak databázový systém prejde na reprezentáciu, že jeden bit nereprezentuje záznam, ale stránku. V tomto prípade je potrebné po vytiahnutí stránok skontrolovať ešte záznamy v samotných stránkach, ktoré záznamy spĺňajú podmienku.



6 Transakcie

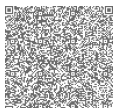
Transakcia predstavuje skupinu operácií, ktorej cieľom je poskytnutie komplexnejšej funkcionality napr. prevod prostriedkov z jedného účtu na druhý. Transakcia v rámci SQL je ohraničená syntaxou pre začiatok a koniec danej transakcie. Samotná syntax je závislá od implementácie napr. PostgreSQL využíva pre začiatok BEGIN a pre ukončenie COMMIT/ABORT. V MySQL je možné použiť pre začiatok START TRANSACTION. SQL syntax pre transakciu v rámci PostgreSQL je nasledovná:

```
BEGIN;  
SQL príkazy (SELECT, INSERT, ...)  
COMMIT;
```

Ukážka 48. Všeobecné definovanie transakcie

Požiadavky na samotnú transakciu alebo transakčný systém pre relačný databázový systém sú Atomicita (z angl. *Atomicity*); Konzistencia (z angl. *Consistency*); Izolácia (z angl. *Isolation*); odolnosť (z angl. *Durability*). Tieto štyri uvedené vlastností vystupujú pod skratkou **ACID**, ktorá reprezentuje tieto vlastnosti. V prípade NoSQL databáz sa hovorí o vlastnostiach CAP (Consistency Availability, Partition tolerance) alebo jeho rozšírenie v podobe PACELC, čo znamená „if Partitioned, then Availability and Consistency; else, Latency and Consistency.“ My sa zameriame na vlastnosti ACID, pretože sa venujeme relačným databázam.

Vlastnosť **Atomicity** nám zabezpečuje, že transakcia je buď vykonaná celá alebo nie je vykonaná vôbec. To znamená, že v prípade akéhokoľvek zlyhania, či už na strane samotného databázového systému, operačného systému alebo hardvéru (pokiaľ sa nejedná o nenávratné poškodenie dát), je možné dosiahnuť túto vlastnosť. Časť vykonávaných dát môže byť počas zlyhania už uložených v rámci nevolatívnej pamäte, a preto je dôležité aby DBMS vedel odstrániť efekt týchto dát z databázy napr. žiadny z používateľov využívajúci služby banky by nebol spokojný, ak by v prípade výpadku prišiel o peniaze v dôsledku toho, že databázový systém nestihol aktualizovať záznam, kde mali byť pripočítané peniaze z prevodu peňazí medzi účtami.



Druhá vlastnosť **Consistency** znamená, že pri vykonávaní transakcie dochádza prechodu z konzistentného stavu databázy opäť do konzistentného stavu. Komplexita samotných transakcií môže presahovať integritné obmedzenia (primárny kľúč, referenčná integrita, check podmienky atď.), ktoré je možné dosiahnuť v rámci DBMS. Preto na zabezpečení konzistencie sa podieľa aj samotný implementátor transakcie/aplikácie, ktorý je zodpovedný za to, že databáza dosiahne opäť konzistentný stav.

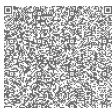
Predposledná vlastnosť je **Isolation**, ktorá zabezpečuje, že systém môže vykonávať viacero transakcií súbežne a výsledný efekt v databáze musí byť rovnaký ako keby boli dané transakcie vykonávané jedna po druhej. V prípade, že nie je dodržaná táto vlastnosť, tak môže transakcia získať nesprávny výsledok výpočtu.

Posledná vlastnosť ACID je **Durability** a zabezpečuje, že v prípade úspešného skončenia transakcie (prebehne COMMIT) sú uskutočnené zmeny zachované v rámci databázy aj v prípade výpadku.

6.1 Sériovateľnosť

V rámci súbežného vykonávania transakcií je potrebné dosiahnuť vlastnosť **Isolation**, ktorá zabezpečuje, že výsledky transakcií uložených v databáze sú zhodné so sériovým vykonávaním transakcií. Sériové vykonávanie garantuje správny výsledok, ale z pohľadu výkonnosti systému sa nejedná o najlepšiu možnosť. Aby sme vedeli garantovať správny výsledok potrebujeme dosiahnuť poradie vykonávania transakcií, ktoré bude mať rovnaký výsledok ako sériové vykonávanie. Takéto vykonávanie nazývame **sériovateľné**.

Sériovateľné vykonávanie má na stav databázy rovnaký efekt ako sériové vykonávanie bez ohľadu na stav databázy a na typ a množstvo operácií v transakciách. Samotná postupnosť vykonávania transakcií a ich operácií sa nazýva **rozvrh** (z *angl. Shedule*). Definícia **rozvrhu** je, že rozvrh vznikne premiešaním transakcií a ich operácií s tým, že poradie operácií je zachované pre jednotlivé transakcie tj. ak transakcia číta objekt X a potom zapisuje objekt Y, tak toto poradie nie je možné zmeniť. Do rozvrhu môžu vstupovať aj nekompletné transakcie (neobsahujú začiatok a koniec transakcie). V ďalších častiach budeme pracovať s pojmami sériový a sériovateľný rozvrh.



Okrem sériových a sériovateľných rozvrhov existujú aj rozvrhy, ktoré neposkytujú správne výsledky vo všetkých prípadoch. V niektorých prípadoch môžu dať správny výsledok vzhľadom na hodnotu objektov a operácie, ktoré sú nad nimi uskutočňované.

V ďalších častiach budeme používať notáciu:

- $R(x)$ – čítanie objektu X
- $W(x)$ – zápis objektu X
- $W_1(X)$ – zápis objektu X pre transakciu X

Príklad:

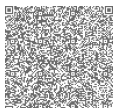
Máme dve transakcie T_1 a T_2 . Prvá transakcia T_1 uskutočňuje prevod medzi dvomi účtami a druhá transakcia T_2 pozostáva z priradenia 10% úroku pre jednotlivé účty.

Transakcia T_1 – $R(A)$; $A = A - 100$; $W(A)$; $R(B)$; $B = B + 100$; $W(B)$

*Transakcia T_2 – $R(A)$; $A = A * 1,1$; $W(A)$; $R(B)$; $B = B * 1,1$; $W(B)$*

Rozvrhy pre dané dve transakcie sú znázornené na obrázkoch Obr. 16 , Obr. 17 a Obr. 18. V rámci prvého rozvrhu S_1 sú jednotlivé transakcie vykonané postupne za sebou. Najprv je vykonaná transakcia T_1 , ktorá je dokončená a až následne je začaté vykonávanie T_2 . V tomto prípade hovoríme o sériovom vykonávaní a výsledok je vždy správny.

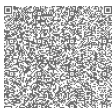
V druhom rozvrhu S_2 môžeme vidieť, že transakcie T_1 a T_2 sú premiešané a ich vykonávanie sa strieda. V rámci transakcie T_1 je najprv vykonané čítanie a následné zapísanie objektu A . Následne začína transakcia T_2 , ktorá vykonáva operácie nad objektom A . Po zápise objektu A transakciou T_2 sa presúva vykonávanie opäť na transakciu T_1 , ktorá pracuje s objektom B – čítanie a zápis. Na záver rozvrhu S_2 vykoná transakcia T_2 čítanie a zápis nad objektom B . Napriek tomu, že transakcie pracovali s rovnakými objektami a vykonávanie prebiehalo súbežne bude výsledok vždy správny. To je zabezpečené tým, že poradie vykonávania nad objektom A a B prechádzalo vždy z T_1 do T_2 . Objekt A aj B bol vždy najprv prečítaný a zapísaný T_1 a až následne T_2 . Pre tento seriovateľný rozvrh je zabezpečené, že výsledok je vždy správny



a poradie vykonávania je rovnaké ako v prípade sériového rozvrhu a to v poradí vykonávania T_1, T_2 .

V treťom rozvrhu S_3 sa nachádza opäť súbežné spracovanie ako v prípade rozvrhu S_2 . Je tu však rozdiel a to ten, že po vykonaní čítaní a zápisu nad objektom A transakciou T_1 dochádza k čítaniu a zápisu nad objektami A, B transakciou T_2 . Po dokončení transakcie T_2 je následne dokončená transakcia T_1 v podobe čítania a zápisu nad objektom B. Toto poradie má za následok, že pripočítanie úroku je uskutočnené uprostred prevodu prostriedkov medzi účtami tj. účtu sú odpočítané prostriedky, následne je vyrátaný úrok a nakoniec sú prirátané prostriedky účtu. V prípade sériového rozvrhu v poradí T_1 , pre hodnoty $A = 1000$ a $B = 500$ by bol výsledok $A = 990$ a $B = 660$. V prípade sériového rozvrhu v poradí T_2 , T_1 je výsledok $A = 1000$ a $B = 650$. V prípade rozvrhu S_3 je však výsledok $A = 990$ a $B = 650$. Môžeme vidieť, že rozvrh S_3 nedal správny výsledok pre dané transakcie. Takýto rozvrh porušuje vlastnosť *isolation* v rámci ACID vlastností.

Pre dosiahnutie sériovateľnosti rozlišujeme **konflikt-sériovateľnosť** a **view-sériovateľnosť**, ktoré hovoria o tom ako je určované, či daný rozvrh je sériovateľný. Konflikt-sériovateľný rozvrh je prísnejší pri posudzovaní sériovateľnosti a je ho možné overiť napríklad pomocou grafu plánovania udalosti (z *angl. Precedence graph*), ktorý obsahuje konfliktné hrany medzi transakciami a v prípade vzniku cyklu je tento rozvrh považovaný ako konflikt-nesériovateľný. Konfliktné hrany vznikajú medzi akýmikoľvek operáciami, kde sa vyskytuje operácia zápisu (write) a sú to kombinácie: R-W; W-R; W-W. Smer hrany v grafe je určený poradím operácií $W_1(A); R_2(X)$ vytvorí hranu $T_1 \rightarrow T_2$. V prípade **view-sériovateľného** rozvrhu je posudzovanie založené na menej prísnom prístupe a pozerá sa na poradie čítania a zapisovania objektov medzi transakciami. Vzhľadom na menej prísne posudzovanie je počet view-sériovateľných rozvrhov väčší ako konflikt-sériovateľných a zároveň platí, že každý konflikt-sériovateľný rozvrh je zároveň aj view-sériovateľný. Znázornenie množín rozvrhov je znázornené na Obr. 19. Nevýhodou view-sériovateľných rozvrhov je náročnosť ich testovania, ktoré nie je možné uskutočniť v reálnom čase, a preto z hľadiska databázového systému je cieľom dosiahnuť nejakú časť konflikt-sériovateľných rozvrhov, ktoré je možné dosiahnuť napríklad dvojfázovým zamykaním - 2PL (z *angl. two-phase locking*)

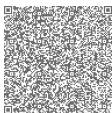


T1	T2
Read(A) A := A - 100 Write(A) Read(B) B := B + 100 Write(B) COMMIT	 Read(A) A := A * 1.1 Write(A) Read(B) B := B * 1.1 Write(B) COMMIT

Obr. 16. Rozvrh S1 – sériový

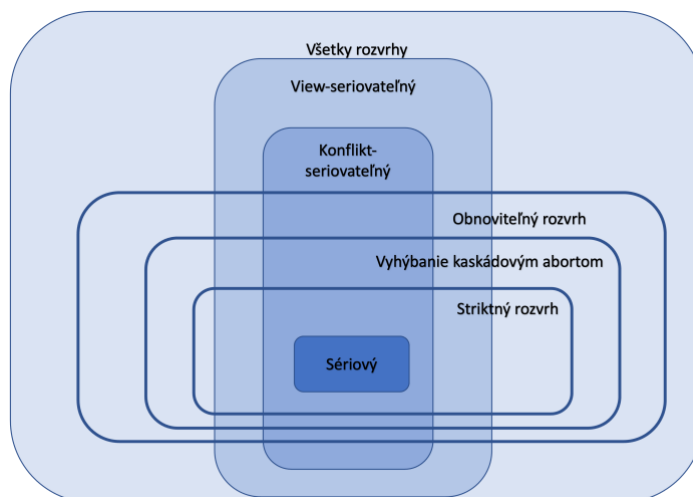
T1	T2
Read(A) A := A - 100 Write(A) Read(B) B := B + 100 Write(B) COMMIT	 Read(A) A := A * 1.1 Write(A) Read(B) B := B * 1.1 Write(B) COMMIT

Obr. 17. Rozvrh S2 – seriovateľný



T1	T2
Read(A) A := A - 100 Write(A) Read(B) B := B + 100 Write(B) COMMIT	 Read(A) A := A * 1.1 Write(A) Read(B) B := B * 1.1 Write(B) COMMIT

Obr. 18. Rozvrh S3 – neseriovateľný – generujúci nesprávny výsledok

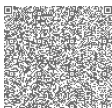


Obr. 19. Množiny typov rozvrhov a ich prekrývanie

6.1.1 Úrovne izolácie v transakciách

Štandard SQL definuje štyri úrovne izolácií:

- *Read uncommitted* – predstavuje najmenej prísnu úroveň, ktorá umožňuje čítanie dát, pre ktoré neprebehol *commit*.
- *Read committed* – umožňuje čítanie iba tých dát, pre ktoré bol uskutočnený *commit*. V prípade, že *commit* prebehol, môže transakcia pri opätovnom čítaní objektu získať rozdielnu hodnotu, ktorá bola zapísaná inou *committed* transakciou.



- **Repeatable read** – oproti úrovni *read committed* neumožňuje čítanie rozdielnej hodnoty v dôsledku toho, že iná transakcia aktualizovala hodnotu a uskutočnila aj *commit*. Aj napriek tomu, že transakcia má prístup k rovnakej hodnote objektu, tak nie je zabezpečená sériovateľnosť vykonávania.
- **Serializable** – zabezpečuje sériovateľnosť vykonávania transakcií. V prípade niektorých DBMS je uvádzané, že poskytujú tento typ izolácie, ale v určitých prípadoch umožňujú aj nesériovateľné vykonávanie.

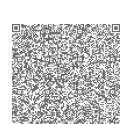
6.2 Isolation (izolácia v transakciach)

V rámci DBMS, kde sú vykonávané transakcie súbežne je potrebné zabezpečiť aby ich výsledok bol správny vzhľadom na nastavenú úroveň izolácie. Pre zabezpečenie správneho výsledku sa používajú protokoly pre riadenie súbežnosti (z *angl. concurrency control protocol*). Je tiež používaný pojem schéma namiesto protokolu. Tieto protokoly môžeme rozdeliť do dvoch kategórií z pohľadu vzniku konfliktov medzi transakciami a sú to:

- **Optimistické** – predpokladajú, že ku konfliktom medzi transakciami dochádza zriedka a je preto lepšie ich riešiť až v prípade ich vzniku.
- **Pesimistické** – predpokladajú, že konflikty vznikajú často, a preto sa tieto protokoly snažia zamedziť ich vzniku.

Samotné protokoly pre riadenie súbežnosti môžeme rozdeliť do týchto kategórií:

- **Lock-Based protokoly** - sú založené na zamykaní dátových objektov pred čítaním a zápisom. Transakcia získa všetky zámky, ktoré potrebuje pre vykonávanie aby ostatné transakcie nemohli ľubovoľne pristupovať k týmto objektom. Zástupcom týchto protokolov je napríklad 2-fázove zamykanie (z *angl. 2-phase locking*). Patrí do skupiny pesimistických protokolov, pretože predchádza vzniku konfliktov.
- **Timestamp-Based protokoly** – sú založené na časových pečiatkach. Na základe časových pečiatok sa systém rozhoduje pri operáciách čítania a zápisu, či transakcia bude pokračovať vo vykonávaní alebo nie.



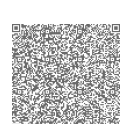
- **Optimistic concurrency-control** - označovaný tiež ako validačný algoritmus. Je zástupcom optimistického prístupu, ktorý vykonáva operácie tak, ako idú. Keď transakcia uskutočňuje COMMIT, tak algoritmus kontroluje, či transakcia môže uskutočniť COMMIT.
- **Multiversion schémy** – v súčasnosti populárny prístup, ktorý využíva veľké množstvo databázových systém aj PostgreSQL. Každá transakcia si zapisuje svoje zmeny do svojej vlastnej lokálnej kópie. Objekty sa nezapisujú do hlavnej databázy. Systém sa až pri žiadosti o COMMIT rozhodne, či sa dané zmeny môžu zapísať do databázy alebo nie.

6.2.1 Riadenie súbežnosti pomocou multiversion schémy

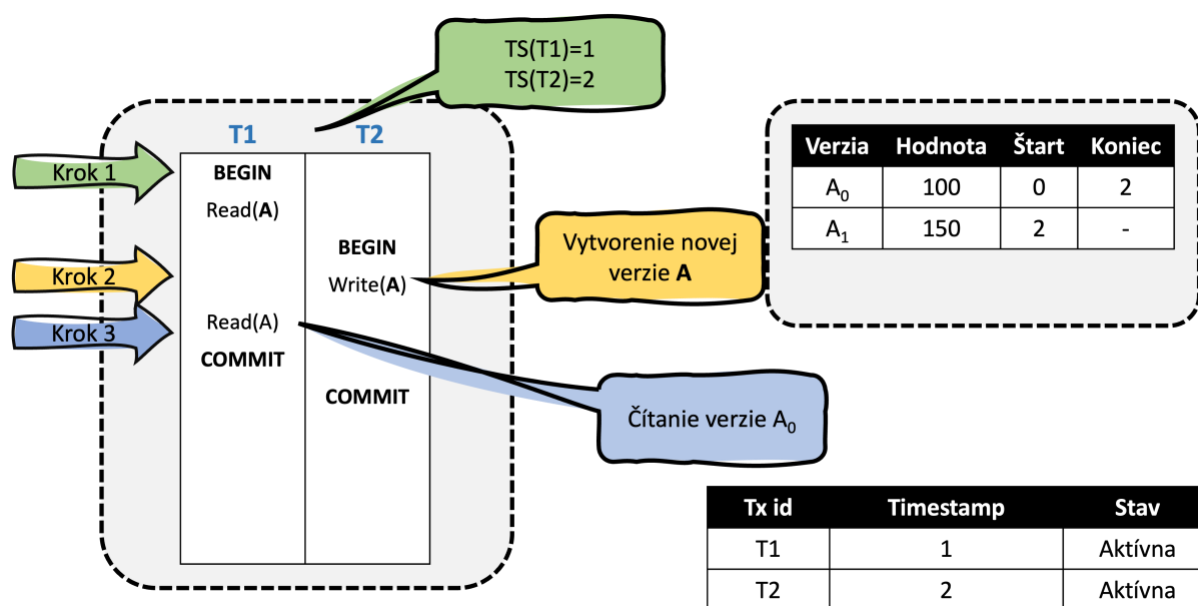
MVCC (Multiversion concurrency control) je založený na princípe, že existuje viacero kópií toho istého objektu. Tieto kópie objektu vznikajú pri operácií **write** z pohľadu rozvrhu transakcie. Pri čítaní nedochádza k vytváraniu novej kópie, ale databázový systém musí vybrať vhodnú verziu objektu tak, aby bola zabezpečená sériovateľnosť. Výhodou multiversion schémy je, že zapisovanie neblokuje čítanie a rovnako čítanie neblokuje zapisovanie. Transakcie, ktoré iba čítajú, dostávajú konzistentný *snapshot* objektov. V prípade, že by si databázový systém uchovával kompletnú históriu objektov, tak je možné uskutočniť „cestovanie v čase“ a získať výsledok dopytu napr. rok dozadu.

V rámci MVCC musí databázový systém uskutočňovať ukladanie verzií objektu, mazanie starých objektov a riešenie manažmentu indexov, pretože aj rôzne verziu objektov musia byť zohľadnené v rámci indexov. Existujú rôzne verzie MVCC, ktoré využívajú napríklad *Timestamp ordering*; *Two-phase locking* alebo *Snapshot isolation*.

V ďalšej časti si stručne priblížime verziu *snapshot isolation* na príklade, v ktorom vystupujú dve transakcie. Prvá transakcia T_1 uskutočňuje iba dvakrát čítanie nad objektom A a transakcia T_2 zapisuje do objektu A. Príklad je znázornený na Obr. 20. V rámci prvého kroku sú priradené časové pečiatky transakciám. PostgreSQL využíva inkrementálny identifikátor pre transakcie, čo zároveň určuje aj časové poradie daných transakcií. Hodnota objektu A vo verzií A_0 je platná ešte pred príchodom transakcie T_1 , ktorá ma časovú pečiatku s hodnotou 1. Transakcia T_1 číta hodnotu A_0 . Vykonávanie pokračuje a transakcia T_2 zapisuje do objektu A. Tento zápis je znázornený na obrázku ako krok 2. Pretože T_2 uskutočnila zápis,



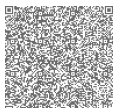
tak je potrebné vytvoriť novú verziu objektu A s označením A₁. Pre objekt A₀ je nastavený koniec platnosti objektu na hodnotu 2, čo znamená že transakcia s časovou pečiatkou 2 zapísala novú hodnotu pre tento objekt. Pre nový objekt A₁ je vytvorený záznam, že platnosť tohto záznamu začína od hodnoty časovej pečiatky 2. Vykonávanie pokračuje a prechádza do kroku tri, kde transakcia T₁ opäť číta objekt A. Pretože transakcie majú mať konzistentný *snapshot*, tak transakcia opäť číta hodnotu A₀ a nie hodnotu uloženú v A₁, ktorá bola modifikovaná transakciou, ktorá prišla až po transakcií T₁.

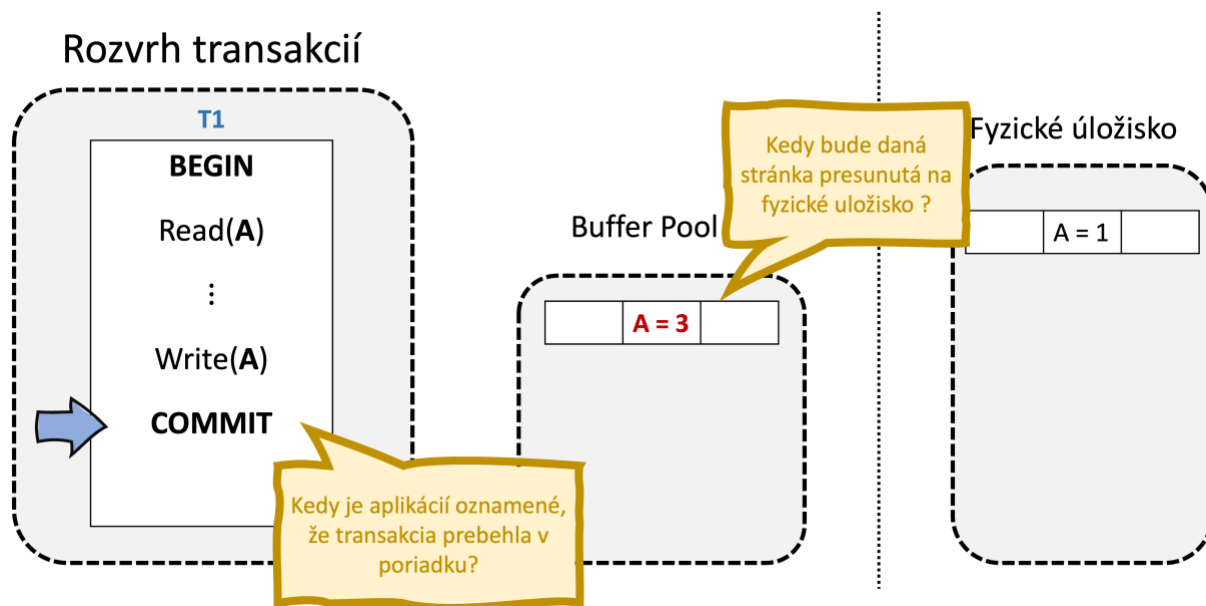


Obr. 20. Ukážka Snapshot isolation pre dve transakcie

6.3 Manažment buffer poolu

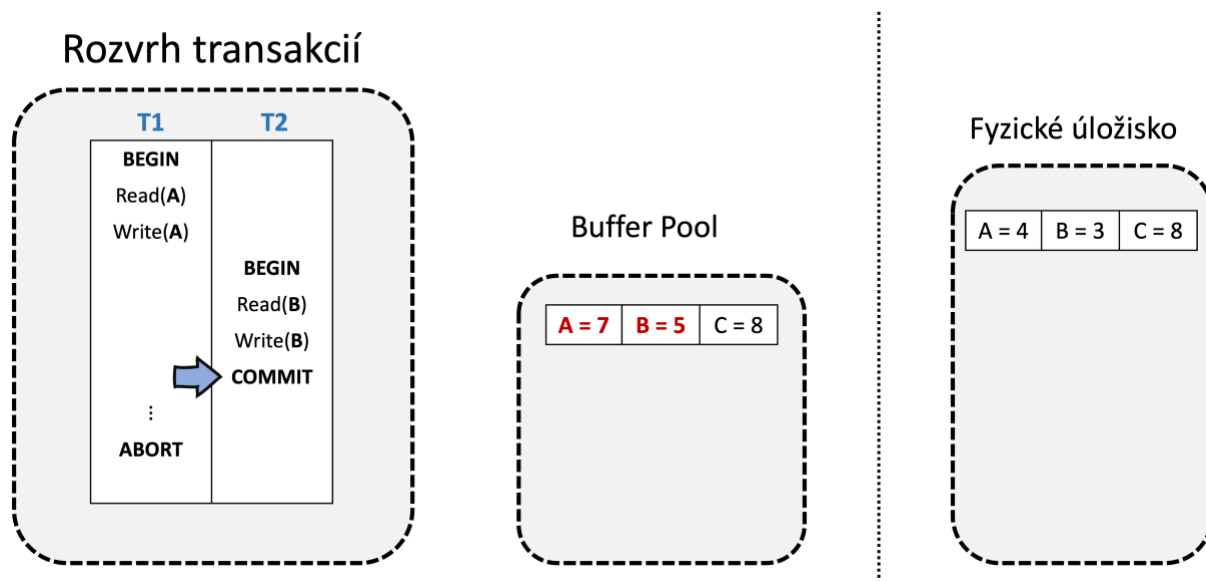
Pojem *Buffer Pool* predstavuje časť v hlavnej pamäti, kde sú načítané stránky z fyzického úložiska ako bolo vysvetlené v kapitole 4. Uvažujme príklad, ktorý je znázornený na Obr. 21, kde sa transakcia nachádza v stave, že prebieha COMMIT. Modifikovaná stránka sa nachádza stále v hlavnej pamäti a v prípade zlyhania dochádza k strate informácií. Pretože transakcia sa nachádza v stave COMMIT prichádzajú dve dôležité rozhodnutia a to, kedy je aplikácií oznámené, že transakcia prebehla úspešne a kedy je stránka presunutá na fyzické úložisko. Ak dôjde k oznámeniu aplikácií, že COMMIT prebehol úspešne a stránka je stále iba v hlavnej pamäti a počas tohto stavu dôjde k výpadku, tak aplikácia si bude myslieť, že modifikácia databázy prebehla úspešne, ale z dôvodu zlyhania to nie je pravda.



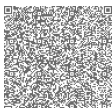


Obr. 21. Ilustrovanie problému s rozhodnutím kedy má byť stránka presunutá na fyzické úložisko

Uvažujme o druhom príklade, kde máme dve transakcie, ktoré uskutočnia zápis nových hodnôt pre rozdielne záznamy v rámci tej istej stránky. Tento príklad je znázornený na Obr. 22. V rámci príkladu sa nachádzame v stave, kedy transakcia T_2 uskutočňuje COMMIT. Je v tomto prípade presunutá stránka s objektom B presunutá na disk? Ak bude presunutá, tak sa presunie aj modifikovaná hodnota objektu A, pre ktorú však nebol ešte uskutočnený COMMIT. V prípade následného výpadku zostane v databáze hodnota, pre ktorú nebol uskutočnený COMMIT a dôjde k porušeniu vlastnosti atomicity.



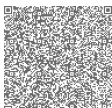
Obr. 22. Príklad na politiky FORCE a STEAL manažmentu Buffer Pool

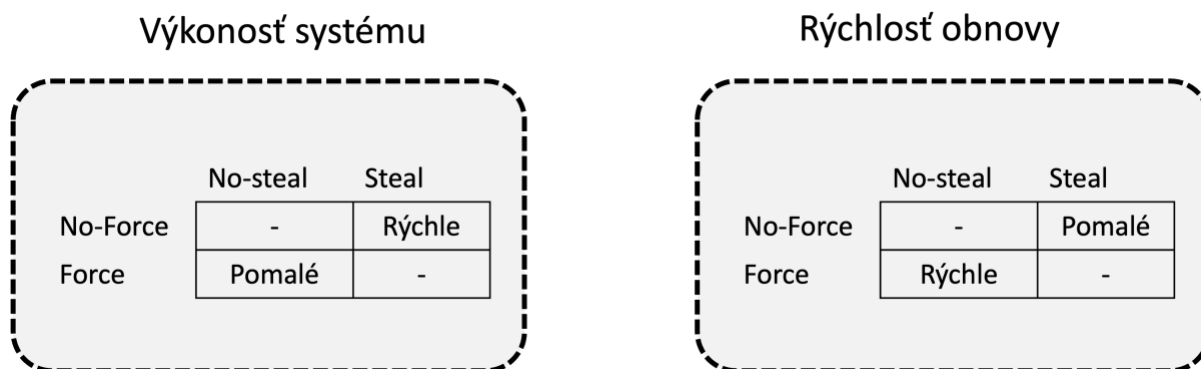


V rámci rozhodovania, kedy má byť stránka presunutá z hlavnej pamäte na fyzické úložisko existujú dva typy politík, ktoré rozhodujú, ako sa má Buffer pool správať pri uskutočňovaní COMMIT. Sú to politiky:

- **Steal politika** - **Steal** nastáva vtedy, keď dochádza k prepísaniu hodnoty v nevolatívnej pamäti (fyzické úložisko) hodnotou transakcie, pre ktorú ešte neprebehol COMMIT. Ak je v rámci manažmentu nastavená politika na **Steal**, tak je možné prepisovať hodnoty na fyzickom úložisku a v prípade **No-Steal** nie je povolené prepisovať hodnoty na fyzickom úložisku.
- **Force politika** – **Force** znamená, že všetky zmenené hodnoty transakciou sú zapísané do nevolatívnej pamäte (fyzické úložisko) pred tým než je samotný COMMIT oznámený aplikácií. Znamená to, že aktualizované hodnoty transakciou sú zapísané do nevolatívnej pamäte a až následne je oznámený commit. Pre nastavenie politiky **No-Force** nie je potrebné pred samotným uskutočnením COMMIT zapísať hodnoty do nevolatívnej pamäte.

Z hľadiska správania sú dve možné kombinácie politík a to **No-Steal + Force** a **Steal + No-Force**. Ich porovnanie je znázornené na Obr. 23, kde môžeme vidieť ich vplyv na výkonnosť systému a rýchlosť obnovy. V súčasnosti sa v rámci databázových systémov využíva iba kombinácia **Steal + No-Force**, ktorá má síce dlhšiu obnovu dát v prípade výpadku, ale výkonnosť systému je výrazne vyššia, pretože nie je potrebné neustále ukladať dáta na fyzické úložisko. V prípade výpadku je potrebné uskutočniť **UNDO** a **REDO** operácie pre vrátenie databázy do stavu, kedy COMMIT transakcie sú uložené a vplyv zrušených transakcií je odstránený z databázy. V prípade politiky **No-Steal + Force** nie je potrebné uskutočňovať **UNDO** a **REDO** operácie, pretože na fyzickom úložisku sa nachádzajú iba COMMIT transakcie.





Obr. 23. Porovnanie manažmentu buffer pool z pohľadu výkonnosti systému a rýchlosti obnovy

Aby bolo možné uskutočniť **UNDO** a **REDO** operácie, tak je potrebné pridať do databázy dodatočnú réžiu. Táto réžia súvisí s dosiahnutím vlastnosti *atomicity* a *durability* pre transakčný databázový systém. Tieto vlastnosti sú popísané v ďalšej časti.

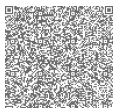
6.4 Atomicity a Durability

Vlastnosti **atomicity** a **durability** vieme dosiahnuť pomocou dvoch prístupov a to **logovania** alebo **shadow-paging**. V rámci ďalšej časti sa zameriame na logovanie, ktoré je využívané v dnešných relačných databázových systémoch. Využíva ho aj implementácia PostgreSQL. Navyše samotné logovanie je dôležité z pohľadu obnovy dát, pretože poskytuje dodatočné informácie databázovému systému, na základe ktorých je možné uskutočniť samotnú obnovu dát.

6.4.1 Logovanie

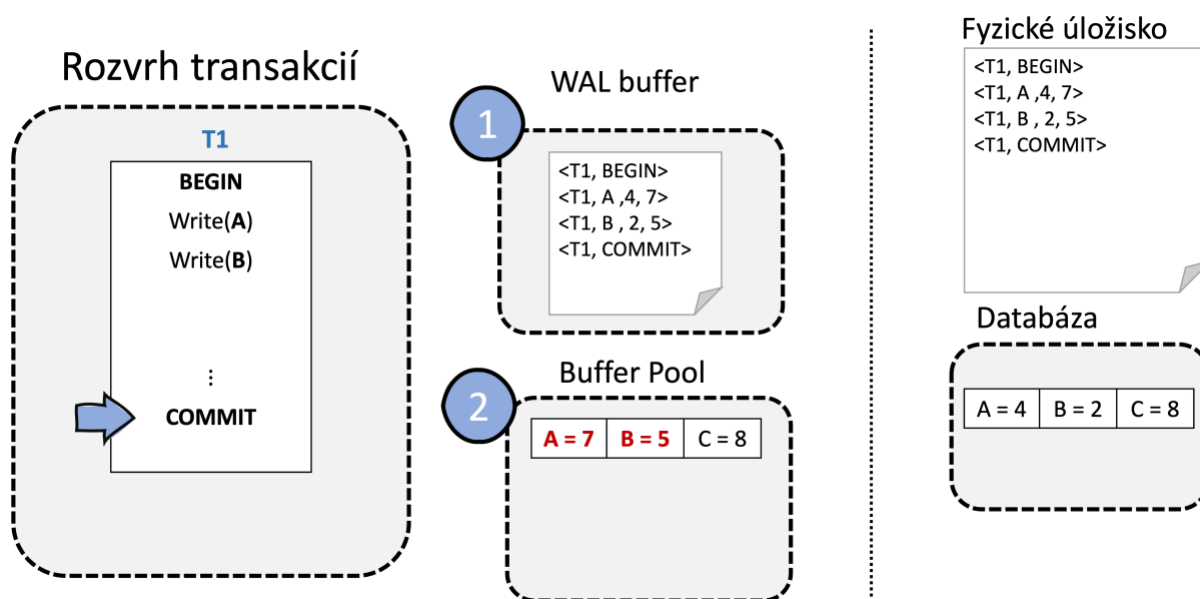
Logovanie v rámci databázového systému ukladá dodatočné informácie, za účelom možnosti obnovenia dát, tak aby bolo garantované, že v prípade zlyhania sú zmeny uskutočnené COMMIT transakciou uložené v databáze a vplyv nedokončených transakcií je odstránený z databázy.

Databázový systém používa logovací súbor, ktorý predstavuje sekvenčný súbor, kde sú zaznamenané zmeny uskutočnené jednotlivými transakciami. Samotný súbor musí byť uložený v nevolatívnej pamäti (fyzické uložisko), aby v prípade zlyhania zostali uskutočnené zmeny zaznamenané. V rámci súboru je potrebné uchovávať všetky potrebné informácie, ktoré budú potrebné pre obnovenie dát.

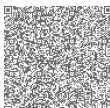


V súčasnosti databázové systémy využívajú buffer pool politiku **Steal + No-Force**, kde je potrebné logovanie. Pre logovanie sa využíva filozofia **Write-Ahead**, ktorá znamená, že predtým ako databázový systém zapíše zmeny v databáze na disk, tak musia byť tieto zmeny zapísané do logovacieho súboru na fyzickom úložisku. **Write-Ahead** logovanie používa skratku WAL (Write-Ahead logging). Ukážka logovania pre transakciu je znázornená na Obr. 24. Proces logovania je nasledovný:

- Transakcia uskutočňuje modifikáciu objektu A. Ako prvá je táto informácia zapísaná do WAL (Write-Ahead logging) buffer v hlavnej pamäti. Následne je zmena uskutočnená aj v rámci stránky nachádzajúcej sa v Buffer Pool.
- Transakcia uskutočňuje modifikáciu objektu B. Ako prvá je táto informácia opäť zapísaná do WAL buffer a následne je modifikovaná stránka v hlavnej pamäti.
- Transakcia uskutočňuje COMMIT. Predtým ako je aplikácií oznámené, že transakcia uskutočnila COMMIT, tak musí byť zapísané na fyzické úložisko obsah WAL Buffer, vrátane operácie COMMIT. Po zápise log záznamov je oznámené aplikácií, že transakcia uskutočnila COMMIT. Stránka s modifikovanými hodnotami nemusí byť zapísaná v tomto momente na fyzické úložisko pretože využívame politiku **Steal + No-Force** a v prípade, že nastane výpadok, tak vieme zrekonštruovať operácie transakcie na databázu.



Obr. 24. Ukážka logovania založeného na princípe Write-Ahead Logging



6.4.2 Obnova dát

Pre samotnú obnovu dát sú potrebné dodatočné informácie, ktoré sú vkladané do log súboru. V súčasnosti sa využívajú princípu algoritmu obnovy ARIES (Algorithms for Recovery and Isolation Exploiting Semantics), ktorý bol vynájdený spoločnosťou IBM v roku 1990 pre ich databázový systém DB2. Princípy tohto algoritmu sa využívajú dodnes s určitým úpravami. Samotný protokol definuje, aké identifikátory musia byť obsiahnuté v rámci log záznamu a pri obnove dát sa uskutočňujú tieto tri fázy:

- **Analýza** – čítanie WAL záznamy uložených na fyzickom úložisku od posledného záznamu za účelom identifikovania stránok, ktoré obsahujú modifikované záznamy, pre ktoré neprebehol ešte COMMIT. Tiež v rámci tejto fázy sa zisťuje zoznam aktívnych transakcií v čase výpadku.
- **REDO** – opakovanie akcií pre vrátenie databázy do stavu v čase výpadku.
- **UNDO** - vrátenie všetkých hodnôt transakcií, pre ktoré neprebehol COMMIT.

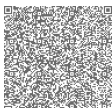
6.5 Príklady na úrovne izolácie

Samotný PostgreSQL neumožňuje úroveň izolácie *read uncommitted*. Preto v nasledujúcich príkladoch budú uvedené ukážky pre úrovne izolácie *read committed*, *repeatable read* a *serializable*. Izolácia *read committed* je v rámci implementácie PostgreSQL nastavená ako východisková a v prípade, že používateľ chce použiť vyššiu úroveň izolácie, je potrebné ju definovať pri začatí transakcie.

Definícia tabuľky pre príklad je:

```
CREATE TABLE users (  
    id      int,  
    name    text,  
    salary  numeric  
);
```

Ukážka 49. Definícia tabuľky users



Naplnenie tabuľky *users* záznamami:

```
INSERT INTO users (id, name, salary)
VALUES (1, 'John', 500),
       (2, 'Peter', 200);
```

Ukážka 50. Vloženie záznamov do tabuľky users.

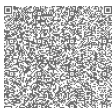
V príkladoch pre jednotlivé úrovne izolácie budú použité SQL príkazy pre získanie hodnoty (SELECT) a aktualizovanie hodnotu (UPDATE).

6.5.1 Read committed

V tejto časti si ukážeme vplyv úrovne izolácie read committed na správanie transakcií.

Tabuľka 11 Zoznam príkazov pre ukážku izolácie Read committed

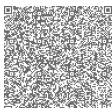
Krok	Transakcia T ₁	Transakcia T ₂	Popis
1.	BEGIN;		Začatie transakcie T ₁
2.		BEGIN;	Začatie transakcie T ₂
3.	SELECT salary FROM users WHERE id = 1;		Vráti hodnotu 500
4.		SELECT salary FROM users WHERE id = 2;	Vráti hodnotu 200
5.	UPDATE users		Aktualizuje hodnotu pre id = 2 na hodnotu 700



	SET salary = salary + 500 WHERE id = 2;		
6.	COMMIT;		Commit transakcie T ₁
7.		SELECT salary FROM users WHERE id = 2;	Vráti hodnotu 700
8.		COMMIT;	Commit transakcie T ₂

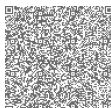
6.5.2 Repeatable read

V rámci tejto úrovne izolácie si ukážeme rovnaký príklad, ako v prípade predchádzajúcej úrovne izolácie *read committed*. Rozdiel oproti predchádzajúcemu príkladu je v kroku 2. a 7. V rámci 2. kroku je pre transakciu T₂ použitá úroveň izolácie *repeatable read*, pričom pri transakcii T₁ je ponechaná na úrovni *read committed*. Transakcia T₁ uskutoční *commit*, čo znamená, že hodnota pre používateľa s id = 2 je aktualizovaná na hodnotu 700. V kroku 7. transakcia 7. uskutoční opäť rovnaký dopyt ako v kroku 4. a tentoraz získava rovnakú hodnotu ako v kroku 4. V tomto prípade, má transakcia T₂ rovnakú verziu dát bez ohľadu na to, či niektorá transakcia uskutočnila *commit* alebo nie.



Tabuľka 12. Zoznam príkazov pre ukážku izolácie Repeatable read

Krok	Transakcia T ₁	Transakcia T ₂	Popis
1.	BEGIN;		Začatie transakcie T ₁
2.		BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;	Začatie transakcie T ₂ s úrovňou izolácie repeatable read
3.	SELECT salary FROM users WHERE id = 1;		Vráti hodnotu 500
4.		SELECT salary FROM users WHERE id = 2;	Vráti hodnotu 200
5.	UPDATE users SET salary = salary + 500 WHERE id = 2;		Aktualizuje hodnotu pre id = 2 na hodnotu 700
6.	COMMIT;		Commit transakcie T ₁
7.		SELECT salary FROM users WHERE id = 2;	Vráti hodnotu 200
8.		COMMIT;	Commit transakcie T ₂



V ďalšom príklade uvedenom v Tabuľka 13 si ukážeme, v akom prípade nám nastavená úroveň izolácie *repeatable read* nepomôže a dôjde k porušeniu sériovateľnosti. V tomto príklade majú obidve transakcie nastavenú úroveň izolácie na *repeatable read*. Obidve transakcie v oboch prípadoch získavajú hodnotu pre konkrétneho používateľa a následne túto hodnotu použijú pre aktualizovanie iného používateľa. Transakcia T_1 získava hodnotu používateľa s $id = 1$ a následne aktualizuje hodnotu používateľa s $id = 2$, kde použije získanú hodnotu. V prípade transakcie T_2 je scenár rovnaký, akurát transakcia získa hodnotu pre používateľa s $id = 2$ a následne túto hodnotu použije pre aktualizovanie hodnoty pre používateľa s $id = 1$. Výsledok pre oboch používateľov je 700. V prípade sériového spracovania transakcií v poradí T_1, T_2 je výsledok po vykonaní transakcie T_1 nasledovný:

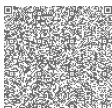
- $id = 1; salary = 500$
- $id = 2; salary = 700$

Po vykonaní T_2 sú hodnoty nasledovné:

- $id = 1; salary = 1200$
- $id = 2; salary = 700$

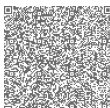
Pri sériovom vykonávaní je odlišný výsledok oproti vykonávaniu v rámci úrovne izolácie *repeatable read* pre ukázaný scenár. V rámci ukázaného scenára boli výsledné hodnoty:

- $id = 1; salary = 700$
- $id = 2; salary = 700$



Tabuľka 13. Zoznam príkazov pre ukážku izolácie Repeatable Read s porušením sériovateľnosti

Krok	Transakcia T ₁	Transakcia T ₂	Popis
1.	BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;		Začatie transakcie T ₁ s úrovňou izolácie repeatable read
2.		BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;	Začatie transakcie T ₂ s úrovňou izolácie repeatable read
3.	SELECT salary FROM users WHERE id = 1;		Vráti hodnotu 500
4.		SELECT salary FROM users WHERE id = 2;	Vráti hodnotu 200
5.	UPDATE users SET salary = salary + 500 WHERE id = 2;		Aktualizuje hodnotu pre id = 2 na hodnotu 700
6.		UPDATE users SET salary = salary + 200 WHERE id = 1;	Aktualizuje hodnotu pre id = 1 na hodnotu 700



7.	COMMIT;		Commit transakcie T ₁
8.		COMMIT;	Commit transakcie T ₂

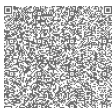
6.5.3 Serializable

Pre príklad, ktorý je uvedený v Tabuľka 13 nastavíme úroveň izolácií pre transakcie T₁, T₂ nasledovne:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

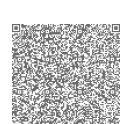
Ukážka 51. Nastavenie najprísnejšej úrovne izolácie - Serializable

Správanie je rozdielne oproti *repeatable read* izolácií v tom, že transakcií T₂ nie je dovolené uskutočniť *commit* v dôsledku porušenia sériovateľnosti ako bolo uvedené v predchádzajúcej časti 6.5.2.



Zoznam použitej literatúry

- [1] Codd, E. F. (1970). *A relational model of data for large shared data banks*. Communications of the ACM, 13(6), 377-387. DOI: 10.1145/362384.362685
- [2] *Modern SQL: A lot has changed since SQL-92*. [online]. [cit. 20.10.2022]. Dostupné z: <https://modern-sql.com/>
- [3] M. Winand, *SQL Performance Explained*. Vienna, Austria: Winand, Markus, 2012, ISBN 978-3950307825
- [4] ELMASRI, Ramez, and Shamkant NAVATHE. *Fundamentals of Database Systems 7th Edition*. 7th ed. Pearson, 2015. ISBN 978-0133970777.
- [5] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. *OLTP through the looking glass, and what we found there*. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08). Association for Computing Machinery, New York, NY, USA, 981–992. DOI: 10.1145/1376616.1376713
- [6] Avi Silberschatz, Henry F. Korth, and S. Sudarshan, *Database System Concepts, Seventh Edition*. McGraw-Hill 2019., ISBN 978-0078022159,
- [7] C. J. Date and Hugh Darwen, *A Guide to the SQL Standard: A user's guide to the standard database language SQL, Fourth Edition*, Addison-Wesley 1997, ISBN 978-0201964264,.
- [8] PostgreSQL 15 Documentation [online]. [cit. 20.10.2022]. Dostupné z: <https://www.postgresql.org/docs/15/index.html>



Ing. Rastislav Bencel, PhD.

Vydala Slovenská technická univerzita v Bratislave vo Vydavateľstve SPEKTRUM STU,
Bratislava, Vazovova 5, v roku 2022

ISBN 978-80-227-5249-7

