ŽILINSKÁ UNIVERZITA V ŽILINE

FAKULTA RIADENIA A INFORMATIKY

ANALYTICKÉ SPRACOVANIE DÁT

Dizertačná práca

28360020243005

Študijný program:	Aplikovaná informatika
Študijný odbor:	Informatika
Pracovisko:	Katedra informatiky
	Fakulta riadenia a informatiky, Žilinská univerzita v Žiline
Školiteľ :	doc. Ing. Michal Kvet, PhD.

Žilina, 2024 **Ing. Martina Hrínová Durneková**

Poďakovanie

Chcela by som sa úprimne poďakovať môjmu školiteľovi doc. Ing. Michalovi Kvetovi, PhD. za jeho trpezlivosť, podporu, cenné rady a pripomienky, ktoré boli kľúčové pre úspešné dokončenie mojej práce. Jeho odborné znalosti a vedomosti sú pre mňa neoceniteľný zdroj inšpirácie.

Veľmi si vážim zaujímavé myšlienky mojich kolegov z Katedry informatiky Fakulty riadenia a informatiky Žilinskej univerzity v Žiline, ktoré viedli obohacujúcim diskusiám.

Veľká vďaka patrí mojim rodičom, môjmu manželovi, deťom a celej mojej rodine za ich neustálu podporu, trpezlivosť a pochopenie počas doktorandského štúdia. Ďakujem, že mi vždy stoja po boku a s láskou ma sprevádzajú na každom kroku.

ABSTRAKT

HRÍNOVÁ DURNEKOVÁ, Martina: *Analytické spracovanie dát* [dizertačná práca] – Žilinská univerzita v Žiline. Fakulta riadenia a informatiky; Katedra informatiky. – Školiteľ: doc. Ing. Michal Kvet, PhD. – Stupeň odbornej kvalifikácie: Doktor filozofie v študijnom odbore Informatika. Žilina: FRI ŽU v Žiline, 2024.

V dizertačnej práci sa venujeme analytickému spracovaniu dát, vďaka ktorému dokážeme transformovať dostupné dáta do cenných informácií, ktoré sú kľúčové pre úspech našej spoločnosti, mnohých firiem a organizácií. Tieto informácie slúžia ako podklad pre strategické rozhodnutia, ktoré vedú k optimalizácii procesov, zvýšeniu efektivity a celkovej prosperite. Naším cieľom je orientovať sa na efektívne ukladanie veľkého množstva dát do databáz a dátových skladov, pričom je potrebné klásť dôraz na rýchlosť a efektivitu vyhľadávania položiek v rámci úložiska. Správny návrh a optimalizácia dátových a indexových štruktúr pokladá základy pre dosahovanie vhodných výkonnostných úrovní. Okrem toho je nutné dbať na adekvátny návrh algoritmov pre analytické spracovanie dát, vyhľadávanie a spracovanie nedefinovaných údajov, a v neposlednom rade aj efektívnu prácu s agregačnými a analytickými funkciami v rámci analýzy dát.

Celý výskum je založený na reálnych dátach získaných z dopravných systémov a z leteckej dopravy. Navrhli a vytvorili sme dátové sklady a dátové modely, pričom sme kládli dôraz na efektivitu ukladania dát. Následne sme sa venovali optimalizácii dátovej štruktúry. Časť výskumu v dizertačnej práci sa venuje porovnávaniu rýchlostí metód importu dát do databázy a exportu dát z databázy, okrem toho sme porovnávali efektivitu vyhľadávania údajov v interných a externých tabuľkách. Práca s analytickými funkciami tvorí dôležitý aspekt efektivity riešenia, preto sme navrhli a experimentálne overili vplyv vytvorených indexov pre príkaz Select, ktorý obsahuje analytické funkcie. Taktiež sme vytvorili koncept pre referencie funkcií v príkaze Select a spracovanie odkazov temporálnych funkcií v relačných databázach. V samotnej časti sa venujeme problematike nedefinovaných hodnôt, ktorých nesprávne spracovanie môže viesť ku skresleným výsledkom, pričom sme vytvorili metodiku spracovania záznamov s nedefinovanými hodnotami v tabuľkách rozdelených na partície.

Kľúčové slová: analytické spracovanie dát, dátový sklad, indexy, analytické funkcie, optimalizácia

ABSTRACT

HRÍNOVÁ DURNEKOVÁ, Martina: Analytical data processing [dissertation thesis] – University of Žilina. Faculty of Management Science and Informatics; Department of Informatics. – Supervisor: doc. Ing. Michal Kvet, PhD. – Qualification level: Philosophiae doctor in the study field Informatics. Žilina: FRI ŽU in Žilina, 2024.

The dissertation is focused on analytical data processing, which allows us to transform available data into valuable information that is crucial for the success of companies and organizations. This information serves as the basis for strategic decisions that lead to process optimization, increased efficiency, and overall prosperity. Our aim is to focus on the effective storage of large amounts of data in databases and data warehouses, while emphasizing the speed and efficiency of retrieving records within the storage. Proper design and optimization of data and index structures lay the foundation for achieving appropriate performance level. In addition, it is necessary to pay attention to the adequate design of algorithms for analytical data processing, searching, and processing of undefined data, and efficient work with aggregation and window functions within data analysis.

The entire research is based on real-world data obtained from intelligent transport systems and aviation systems. We designed and created data warehouses and data models, placing emphasis on data storage efficiency. Subsequently, we focused on optimizing the data structure. Part of the research is dedicated to comparing the speed of data import methods into the database and data export methods from the database. In addition, we compared the efficiency of searching for data in internal and external tables. Working with window functions is an important aspect of the solution's efficiency. Therefore, we designed and experimentally verified the impact of the created indexes for the Select statement, which contains window functions. We also created a concept for function references in the Select statement and treating temporal function references in relational database management system. The section itself deals with the problem of undefined values, the incorrect processing of which can lead to distorted results. We created a methodology for processing records with undefined values in partitioned tables.

Key words: analytical data processing, data warehouse, indexes, window functions, optimization

Obsah

Úvod

„Sme obklopení dátami, ale hladní po poznatkoch.“ Tento výrok, ktorý vyslovil Jay Baer, odborník v oblasti marketingu, dáva do pozornosti dôležitosť dát a ich správne využitie. Dáta sú všade okolo nás. Ich množstvo každý deň enormne narastá. No nie vždy je dostatočne využitý ich potenciál. Nie vždy sa dokážu správnym spôsobom pochopiť a interpretovať. Preto sa v súčasnosti kladie čoraz väčší dôraz na dátovú analýzu. Vďaka nej sme schopní dostupné dáta spracovať, vykonať nad nimi rôzne analýzy, štatistiky, reporty, na základe ktorých vieme získať hodnotné informácie, ktoré sú pre spoločnosť, ale i pre mnohé firmy tak dôležité. Firmy a organizácie na základe získaných poznatkov vedia robiť strategické rozhodnutia, ktoré vedú k optimalizácii procesov, zvýšeniu efektivity a celkovej prosperite. Okrem toho vedia pružne reagovať na rýchlo sa rozvíjajúci priemysel a mnohé ďalšie požiadavky [36]. Pre zabezpečenie bezproblémovej používateľskej skúsenosti počas analýzy je rýchle vyhľadávanie údajov a spracovanie príkazov nevyhnutné. To znamená, že je potrebné minimalizovať celkovú dobu čakania na výsledky [37].

Preto, hlavným cieľom dizertačnej práce je zamerať sa na efektívne a rýchle vyhľadávanie údajov, spracovanie štruktúr, ukladanie veľkého množstva dát a ich analýzu. V rámci toho je dôležité orientovať sa na vhodný návrh dátového skladu, v ktorom je potrebné sústrediť sa nie len na rýchlosť vyhľadávania položiek v dátovom sklade na základe vhodného návrhu indexových štruktúr, ale i následnému optimalizovaniu týchto. Ďalším cieľom je tvorba algoritmov pre extrakciu, transformáciu a import dát do dátového skladu. Taktiež je potrebné zamerať sa na návrh a implementáciu algoritmov pre rýchle analytické spracovanie dát, vyhľadanie a spracovanie chybných či nedefinovaných údajov. Práca s analytickými a agregačnými funkciami výrazne ovplyvňuje výkonnosť príkazov, takže tvorí dôležitý aspekt efektivity riešenia. Celý výskum bude založený na reálnych dátach z dopravných systémov a leteckej dopravy. Na základe týchto dát môžeme demonštrovať príklady a experimentálne overiť či navrhované riešenia prinášajú požadované výsledky.

Práca pozostáva z niekoľkých častí rozdelených do samostatných celkov. V úvode sú bližšie špecifikované jednotlivé ciele dizertačnej práce. Prvá kapitola je zameraná na analýzu súčasného stavu, v ktorej je bližšie popísané čo je to analytické spracovanie dát, dátové sklady – ich návrh, charakteristiky a porovnanie s transakčnými databázami. Ďalšiu časť tejto kapitoly tvorí popis existujúcich nástrojov, ktoré slúžia na analýzu údajov. Okrem toho táto kapitola obsahuje aj časť s názvom *Optimalizácia výkonnosti v databázových*

systémoch, kde sú bližšie popísané možnosti a techniky ako zlepšiť efektívnosť vykonávaných dotazov. V rámci analýzy súčasného stavu sa taktiež venujeme popisu existujúcich funkcií v databázových systémoch. Druhá kapitola je zameraná na popis dát, nad ktorými budeme vykonávať definované experimenty, pričom tieto dáta reprezentujú reálne dáta z dopravných systémov a leteckej dopravy. Tretia kapitola obsahuje informácie o doterajšom výskume, popisuje jednotlivé hodnotiace štúdie a experimenty, na základe ktorých dokážeme prezentovať dosiahnuté výsledky. Súčasťou štvrtej kapitoly je prehľad riešenej problematiky v rámci vedeckého portfólia. Posledné dve kapitoly sú zamerané na popis doterajších publikácií a záver obsahuje zhrnutie doterajších výsledkov.

V jednotlivých častiach sa budem odkazovať na moje doterajšie publikácie, ktoré sú označené začiatočnými písmenami môjho priezviska, za ktorými nasleduje poradové číslo publikácie, ktoré sú vypísané v kapitole *Publikácie*. Príklad označenia publikácie je [HD1].

Ciele práce

Táto kapitola detailne popisuje ciele dizertačnej práce, ktorým sa v rámci výskumu venujeme.

Optimalizácia

Keďže sa v dátových skladoch nachádza obrovské množstvo dát, s ktorými sa pracuje, je potrebné k nim efektívne pristupovať. Z toho dôvodu sa v rámci výskumu venujeme ***rýchlosti vyhľadávania potrebných údajov*** v dátových skladoch. Na základe toho vytvárame metodiky na efektívne spracovanie analytických dotazov, s čím úzko súvisí ***vytváranie indexov*** a ich optimalizácia tak, aby náklady na vyhľadávanie boli čo najmenšie. Okrem toho sa zameriavame na ***využiteľnosť analytických funkcií***, kde kladieme dôraz na ich ***výkonnosť a rýchlosť v rámci plánu vykonávania***. Súčasťou toho je nutné analyzovať vhodnosť vytvorených indexov pre dané spracovanie a ***optimalizovať príkazy***, pričom sa bližšie zameriavame na analytické funkcie LEAD(), LAG() a posuvné okná. Vzhľadom na zlepšenie výkonu spracovania príkazov sa zaoberáme ***návrhom referencovania funkcií*** v príkaze Select a ***spracovaním odkazov temporálnych funkcií***. Okrem efektívneho vyhľadávania, tvorby indexov a ich optimalizácii sa venujeme ***návrhu dátových skladov*** a ***optimalizácii dátovej štruktúry***. Vzhľadom na prácu s rozsiahlymi súbormi dát je kľúčovým faktorom ***rýchlosť importu / exportu týchto dát do / z databázy***.

Algoritmy pre analytické spracovanie dát

Dáta prichádzajú z rôznych zdrojov. Nie je možné sa zaručiť za ich správnosť a kompletnosť. Preto jedným z problémov obrovského množstva dát je ***výskyt nedefinovaných hodnôt***. Tie sa snažíme efektívnym spôsobom spracovať tak, aby sme zabezpečili správnosť výstupov analýzy. S problematikou neúplnosti dát sa spája potreba ***vytvoriť metodiku pre správu záznamov nadobúdajúcich nedefinované hodnoty*** v stĺpci / stĺpcoch, ktoré slúžia ako kľúčové stĺpce tvorby partícií. Ďalšou oblasťou výskumu je ***návrh a tvorba vlastných algoritmov*** pre analytické spracovanie dát.

1 Analýza súčasného stavu

V prvej etape práce je vždy nutné získať väčší prehľad o danej problematike a zozbierať dôležité informácie, ktoré s prácou súvisia. Tomu sa venuje táto kapitola pod názvom *Analýza súčasného stavu*. V tejto kapitole sa zameriavame na popis analytického spracovania dát, dátových skladov, optimalizácii, funkcií a existujúcich systémov, ktoré umožňujú dáta analyzovať. Analytickému spracovaniu dát v teoretickej rovine sa venuje moja publikácia [HD3].

1.1 Analytické spracovanie dát

Dátová analýza je proces čistenia, transformácie, modelovania a interpretácie údajov. Jej cieľom je získať užitočné informácie a poznatky, na základe ktorých je možné vykonať kvalitnejšie rozhodnutia [37]. Vďaka dátovej analýze je možné lepšie pochopiť predchádzajúce udalosti, zamerať sa na lepšie rozhodnutia súčasnosti a predpokladať smerovanie do budúcnosti. V Všetky sféry podnikania sa dnes spoliehajú na dáta a ich analýzu pri prijímaní dôležitých rozhodnutí [10], [36].

1.1.1 Fázy analytického spracovania dát

Analytické spracovanie dát sa skladá z niekoľkých fáz, ktoré sú bližšie popísané v nasledujúcej časti.

Požiadavky na dáta

V prvej fáze analytického spracovania dát je potrebné zistiť, aké sú požiadavky na dáta, prečo sa bude analýza vykonávať, aké údaje budú analyzované, aký bude cieľ experimentov a pre koho budú výsledky analýzy určené.

Zber údajov

Zber dát je druhá fáza. Zameriava sa na zhromažďovanie údajov podľa predmetu záujmu a požiadaviek. Dáta sú zberané z rôznych heterogénnych zdrojov, ako napríklad satelity, kamery, senzory, online zdroje, prípadové štúdie, prieskumy a iné.

Spracovanie údajov

Ďalšia fáza sa zaoberá spracovaním zozbieraných údajov z rôznych zdrojov. V tejto fáze dochádza k reorganizácii dát do podoby vhodnej na ukladanie do dátového úložiska. Taktiež je dôležité údaje vyčistiť od duplicít, nedefinovaných hodnôt a chybných dát, ktoré môžu spôsobiť nekonzistenciu.

Analýzy dát

Keď sú dáta spracované, vyčistené a pripravené, dochádza k samotnej analýze údajov pomocou rôznych matematických funkcií, modelov a nástrojov určených na analýzu údajov.

Interpretácia

Poslednou fázou analytického spracovania dát je interpretácia, v ktorej dochádza k prezentácii výstupov prostredníctvom reportov a rôznych dátových vizualizácií v podobe grafov, tabuliek, máp a množstva iných metód. Na základe týchto výsledkov je ďalej možné vykonať vhodné a dôležité rozhodnutia na zlepšenie kvality práce spoločnosti.

1.2 Dátový sklad

Dátový sklad je centrálne úložisko integrovaných dát, ktoré prichádzajú z jedného alebo viacerých zdrojov [37]. Považuje sa za hlavný komponent pre systém Business Intelligence, ktorý je určený na analýzu dát. Uchováva historické údaje na jednom mieste po dlhé časové obdobie [3]. Tieto dáta sa následne využívajú na analýzu a tvorbu reportov, ktoré slúžia na zlepšenie kvality strategických rozhodnutí v organizáciách [11], [36]. Dátové sklady sú navrhnuté hlavne na rýchle vykonávanie dotazov, preto často obsahujú nenormalizovanú schému, ktorá vedie k optimalizácii výkonu a rýchlejšej odozve [38]. Na zabezpečenie vysokej priepustnosti údajov a rýchleho vykonávania dotazov je potrebné dátový sklad vhodne navrhnúť. Dátové sklady využívajú multidimenzionálny prístup k dátam [39]. Taktiež poskytujú OLAP nástroje, ktoré pomáhajú efektívne a interaktívne analyzovať údaje [3], [10].

1.2.1 Návrh dátového skladu

Dátový sklad obsahuje veľké množstvo údajov, ktoré je potrebné vhodne uchovávať. Aby bolo možné údaje efektívne využívať, je potrebné identifikovať aspekty organizácie, ktorým je potrebné porozumieť. Následne sa dáta rozdelia do príslušných tabuliek. Dátový sklad využíva dva typy tabuliek:

- *Tabuľka faktov*
- *Tabuľka dimenzií*

Tabuľka faktov obsahuje merateľné atribúty, zatiaľ čo ***tabuľka dimenzií*** obsahuje atribúty, ktoré popisujú prvky organizácie. Dimenzie reprezentujú kategórie, ktoré sa používajú na organizáciu dát, ukladanie a načítanie hodnôt [6]. Existuje niekoľko typov

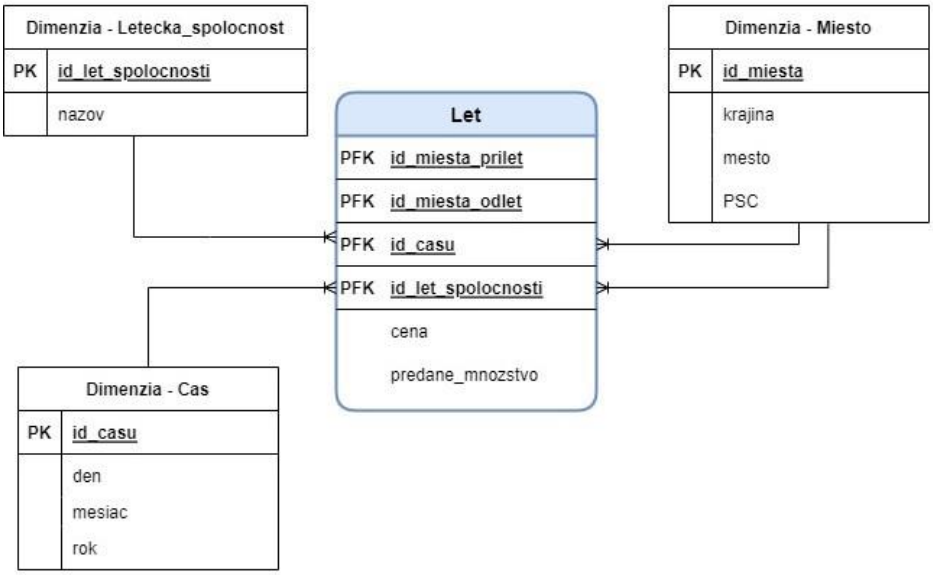
schém dátových skladov, ktoré vytvárajú a logicky popisujú určitú databázu. Medzi tieto schémy patrí [39]:

- *Star schéma*
- *Snowflake schéma*
- *Fact Constellation schéma*

Star schéma

Star schéma je najjednoduchšia schéma, ktorá je zložená z jednej tabuľky faktov, obsahujúcou kľúče jednotlivých dimenzionálnych tabuliek a merateľné atribúty. Ďalej obsahuje niekoľko tabuliek, ktoré predstavujú dimenzie. Dimenzie sú reprezentované jedno-dimenzionálnou tabuľkou, ktorá obsahuje atribúty a môže byť nenormalizovaná [1], [16], [7].

Obrázok 1 znázorňuje Star schému, ktorá reprezentuje zjednodušený model letov. V reálnom svete by bol model komplexnejší a každá letenka by sa evidovala osobitne, avšak v tomto prípade ide o nenormalizovanú schému. Dátový sklad obsahuje tri dimenzionálne tabuľky: *Čas*, *Letecka_spolocnost* a *Miesto*. Tabuľku faktov reprezentuje tabuľka *Let,* ktorá ukladá informácie o predanom množstve leteniek daného letu za určitú cenu. Atribúty v tabuľke faktov predstavujú *príletové* a *odletové stredisko*, *čas letu*, *spoločnosť*, ktorá let vykonáva, *predané množstvo* reprezentuje počet predaných leteniek a *cena* reprezentuje cenu letu. Cena sa môže v čase meniť.

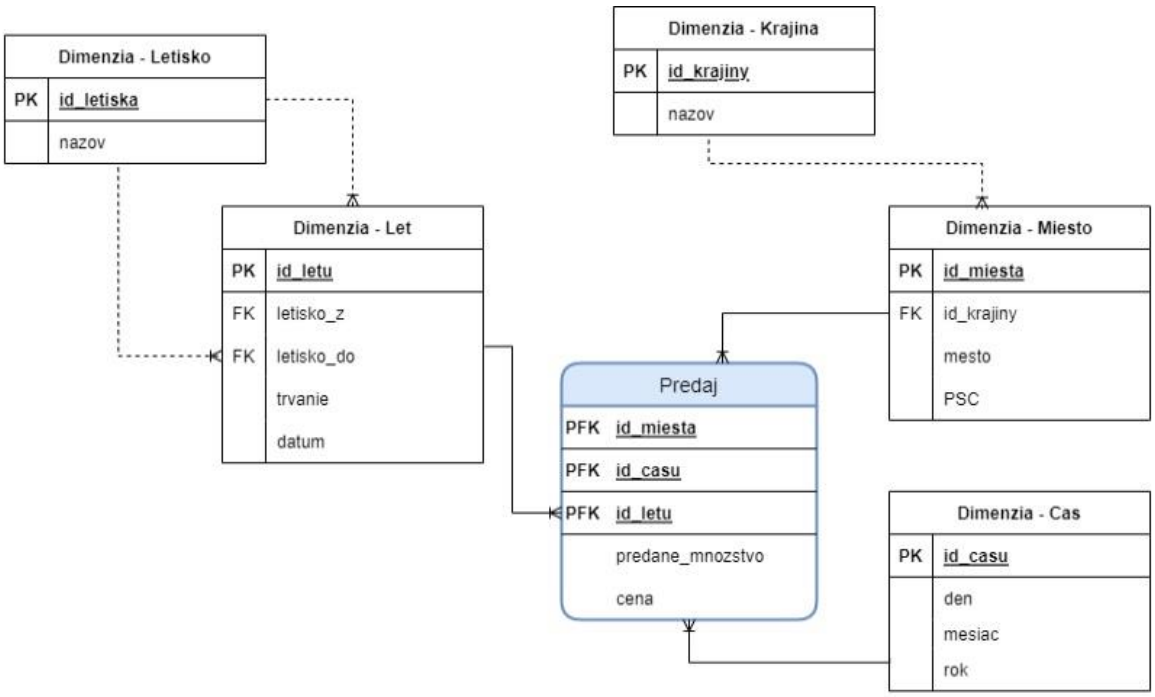


Obrázok 1: Star schéma

Snowflake schéma

Snowflake schéma je schéma, ktorá sa vyznačuje normalizovanými dimenzionálnymi tabuľkami, takže dochádza k redukcii dátovej redundancie. To vedie k zníženiu pamäťového priestoru a k ľahšiemu udržiavaniu dát v úložisku [1], [16], [7].

Obrázok 2 znázorňuje Snowflake schému, ktorá uchováva informácie o predanom množstve leteniek pre daný let. Obsahuje jednu tabuľku faktov *Predaj* a normalizované tabuľky dimenzií *Čas*, *Let*, *Letisko*, *Miesto*, *Krajina*.

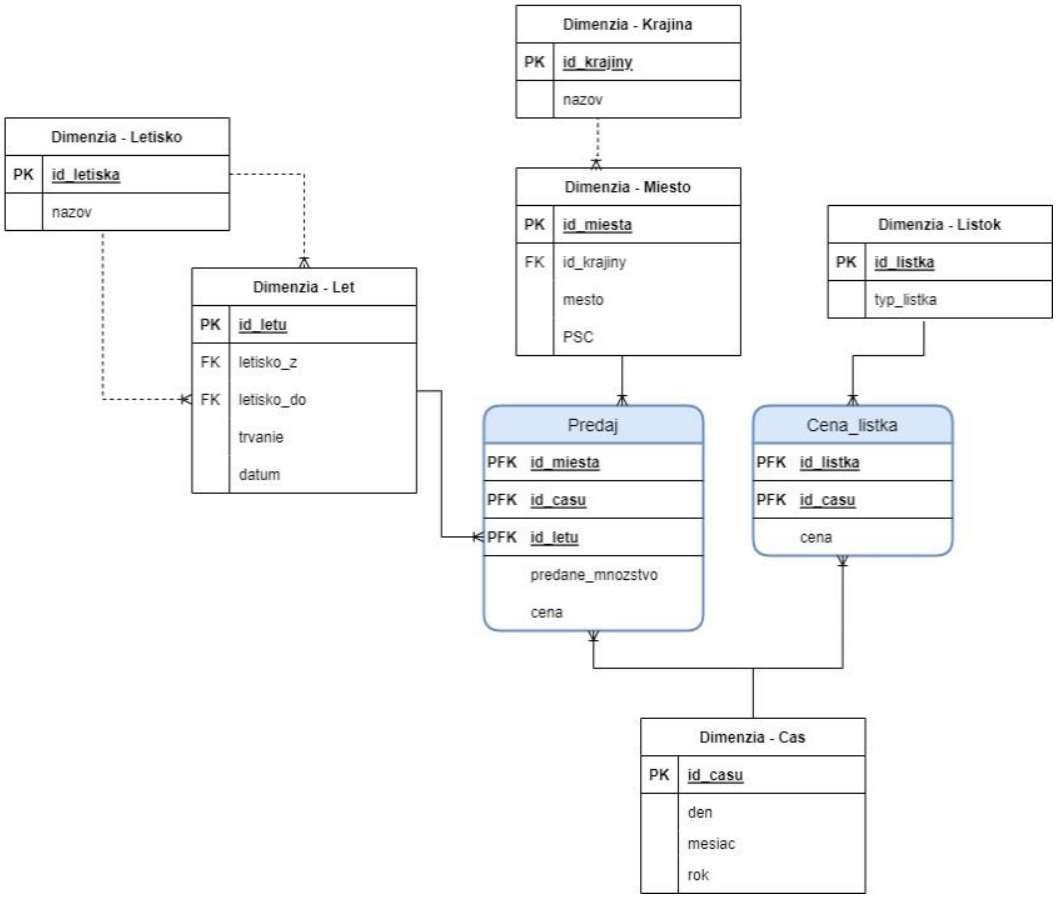


Obrázok 2: Snowflake schéma

Fact constellation schéma

Posledným typom schémy je ***Fact constellation schéma***, taktiež označovaná aj ako ***Galaxy schéma***. Je to typ schémy, ktorá obsahuje normalizované dimenzionálne tabuľky a niekoľko tabuliek faktov.

Obrázok 3 znázorňuje Galaxy schému, ktorá reprezentuje opäť predaj leteniek v čase, pričom je rozšírená o ďalšiu tabuľku faktov *Cena_listka* a tabuľku dimenzií *Listok*.



Obrázok 3: Galaxy schéma

1.2.2 Charakteristiky dátového skladu

Dátový sklad má niekoľko charakteristík, ktoré popísal vedec Bill Inmon, ktorý je považovaný za otca dátových skladov [6], [40]. Medzi tieto charakteristiky patrí:

Orientácia na predmet

Dátové sklady tvoria štruktúru vhodnú na analyzovanie dát, ktoré sú zamerané na konkrétny predmet. Predmet môže reprezentovať predaj, distribúciu a iné.

Integrácia

Dátové sklady obsahujú veľké množstvo dát z rôznych zdrojov, preto je potrebné zaistiť ich konzistenciu a eliminovať akúkoľvek nekonzistenciu medzi dátami. Pre zabezpečenie konzistencie je potrebné dodržať určitý spôsob merania premenných, konvenciu pomenovávania a niekoľko ďalších pravidiel.

Časová zložka

Dáta v dátovom sklade reprezentujú historické dáta zozbierané v rámci dlhodobého časového horizontu, preto je pri analýze potrebné zamerať sa na zmeny v čase.

Stabilita

Dáta v dátovom sklade sú nemenné, takže nie je možné ich akokoľvek meniť, modifikovať alebo mazať ich hodnoty. Dáta sú určené výlučne na čítanie.

1.2.3 Dátový sklad a transakčné databázy

Dátové sklady majú množstvo odlišností od klasických transakčných databáz. Nasledujúca *Tabuľka 1* obsahuje porovnanie dátových skladov a transakčných databáz.

Tabuľka 1: Porovnanie dátových skladov a transakčných databáz

Dátový sklad	Transakčná databáza
Udržiava historické dáta	Udržiava aktuálne dáta
Stabilita, dáta je možné len pridávať	Dáta je možné modifikovať
Orientácia na predmet	Orientácia na procesy
Zameriava sa na výstup informácií	Zameriava sa na dáta vstupujúce do systému
Uchováva veľké množstvo dát	Uchováva menšie množstvo dát
Navrhnuté pre OLAP	Vytvorené pre OLTP
Založené na troch typoch schém	Založené na ERA modeli

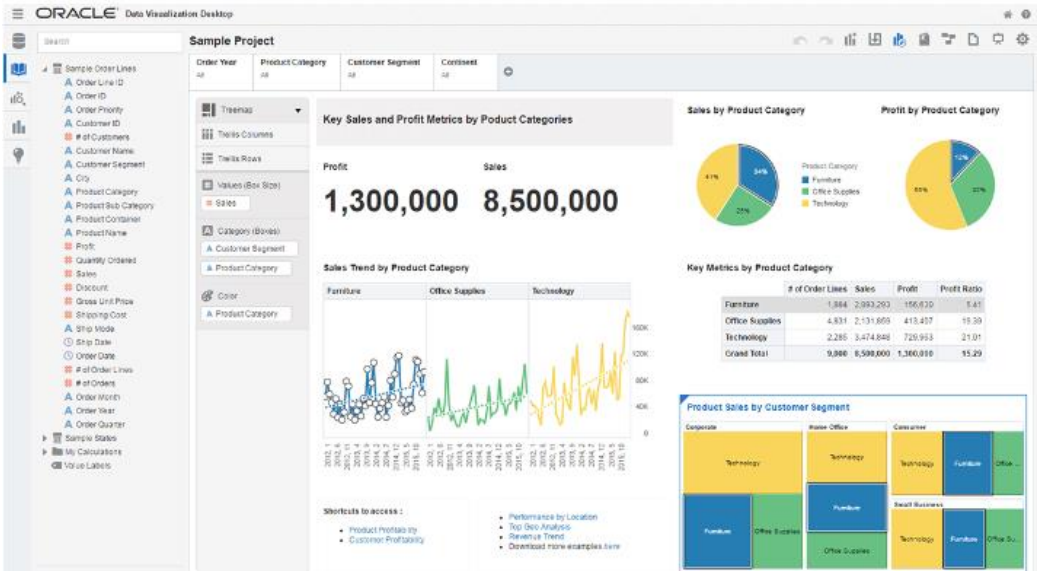
1.3 Existujúce nástroje

Informačné technológie a metódy zamerané na zber, analýzu, normalizáciu a interpretáciu údajov pre organizácie, sa nazývajú *Business Intelligence* (označenie BI). Hlavným cieľom BI systémov je napomáhať organizáciám vo fáze plánovania a rozhodovania v rôznych sférach, ktoré v rámci organizácie prebiehajú [40]. Preto sa tieto systémy niekedy označujú aj ako *Systémy na podporu rozhodovania* [12]. Existuje niekoľko nástrojov určených na prácu s dátami v dátových skladoch. V tejto časti sa venujeme zhrnutiu dostupných nástrojov určených na analýzu dát, ktoré úzko súvisia s témou práce. Avšak vzhľadom na to, že sa náš výskum zameriava na nižšiu úroveň spracovania dát, hlbšej analýze týchto nástrojov sa venovať nebudeme.

1.3.1 Oracle Data Visualization Desktop

Nástroj *Oracle Data Visualization Desktop* je desktopový nástroj vyvinutý pre vizualizáciu a analýzu údajov viazaný na Oracle Cloud technológie. Je to nástroj schopný pracovať s veľkým množstvom údajov zozbieraných z viacerých zdrojov, na základe ktorých dokáže vytvárať rôzne typy analýz a vizualizácií. Tento nástroj je jednoduchý na používanie, rýchly pri vykonávaní rôznych úloh a v neposlednom rade je to inteligentný nástroj, ktorý dokáže prepájať viaceré druhy vizualizácií. Taktiež umožňuje veľmi

jednoducho vykonávať komplexné agregačné funkcie [5]. Medzi jeho ďalšie vlastnosti môžeme zaradiť rýchle a jednoduché pripojenie na rôzne zdroje dát, vysoký výkon a vylepšenia v procese analýzy.



Obrázok 4: *Oracle Data Visualization Desktop*

Zdroj: https://content.dsp.co.uk/apex/oracle-data-visualization

Obrázok 4 znázorňuje ukážku nástroja *Oracle Data Visualization Desktop* s rôznymi typmi analýz, grafmi a panelom s nástrojmi.

Okrem desktopovej verzie existuje takzvaný *Oracle Analytics Cloud*. Je to verejná a škálovateľná cloudová služba, ktorá poskytuje používateľom moderné a intuitívne prostredie s množstvom funkcií na prípravu dát, tvorbu rôznych typov analýz, reportov a ich vizualizáciu. *Oracle Analytics Cloud* podporuje strojové učenie, ktoré umožňuje organizáciám zlepšovať svoje kompetencie na základe analýzy [23].

1.3.2 R-Programming

R je jazyk určený na vykonávanie štatistických výpočtov a analýzu rozsiahlych dát. Medzi jeho kľúčové vlastnosti môžeme zaradiť, že je to prostredie pre ukladanie a manipuláciu veľkého množstva dát, obsahuje mnoho výpočtových funkcií a operátorov, zahŕňa strojové učenie, regresnú analýzu a poskytuje grafické prostriedky na analýzu údajov [10]. Okrem toho umožňuje efektívne spracovanie a ukladanie údajov, je to jednoduchý programovací jazyk, ktorý obsahuje cykly, podmienky a rôzne funkcie. Je to rýchlo rozvíjajúci sa jazyk, ktorý sa stále rozširuje o nové balíčky [19], [41].

1.3.3 MATLAB

MATLAB (MATrix LABoratory) a je to vysokoúrovňové programovacie prostredie, ktoré sa využíva na matematické výpočty, vďaka množstvu vstavaných funkcií, návrhov algoritmov, analýze a ukladaniu údajov, návrhov komunikačných a riadiacich systémov. Okrem toho umožňuje vykreslenie funkcií a prepojenie s kódmi programov písaných v programovacích jazykoch C, C++, FORTRAN a Java. Je to programovacie prostredie vyvinuté spoločnosťou MathWorks [13].

1.3.4 Power BI

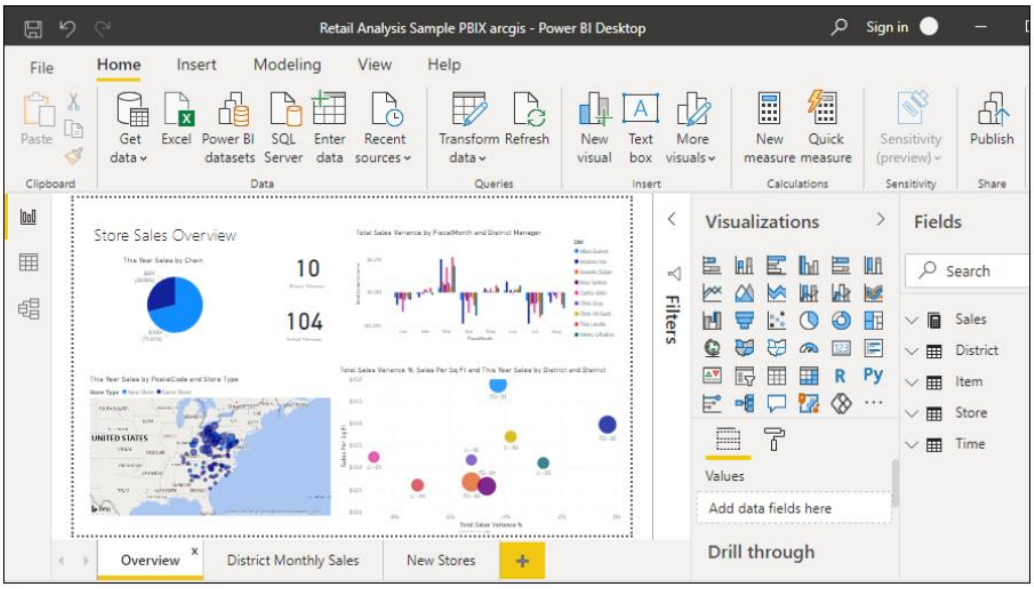
Power BI je zoskupenie aplikácií a služieb, ktoré spolu vytvárajú komplexný nástroj, vďaka ktorému je možné zjednotiť dáta z rôznych zdrojov tak, aby bolo možné s nimi jednoducho pracovať a vytvárať z nich vizualizácie a iné interaktívne prehľady. Tento nástroj sa dokáže jednoducho pripojiť k zdrojovým údajom, a preto môžu byť údaje v rôznych formátoch, ako napríklad excel súbor, súbory typu CSV, XML, JSON, údaje uložené na cloude či iných lokálnych úložiskách. Ide o nástroj poskytovaný spoločnosťou Microsoft [14].

Power BI je zložený z niekoľkých častí, ktoré spoločne kooperujú. Pozostáva z troch základných častí:

- *Power BI Desktop* – je počítačová aplikácia
- *Power BI service* – je online služba známa pod typom „softvér ako služba“
- *Mobilná aplikácia* – je aplikácia vyvinutá pre zariadenia s rôznymi operačnými systémami
- *Power BI Report Server* – je lokálny server reportov, na ktorom je možné vytvorené reporty publikovať
- *Power BI Report Builder* – je nástroj na vytváranie stránkových reportov

Tieto prvky sú navrhnuté tak, aby bolo vytváranie, používanie a zdieľanie vytvorených reportov čo najefektívnejšie [14].

Obrázok 5 zobrazuje ukážku nástroja Power BI, na ktorom je možné vidieť rôzne typy analýz zobrazené na grafoch a panel s nástrojmi.



Obrázok 5: Nástroj Power BI

Zdroj: https://docs.microsoft.com/sk-sk/power-bi/fundamentals/desktop-what-is-desktop

1.4 Optimalizácia výkonnosti v databázových systémoch

Výkonnosť databázy v rámci informačných systémov je v dnešnej dobe kľúčový faktor pre organizácie, ktoré využívajú tieto systémy na dennej báze. Preto cieľom optimalizácie v databázových systémoch je zabezpečiť, aby vykonávanie dotazov bolo čo najefektívnejšie. Výkon databázy ovplyvňuje niekoľko faktorov ako napríklad už samotný návrh databázy, jej štruktúra, množstvo dát a ďalšie [12], [25]. Aby sme vedeli zlepšiť a zrýchliť aplikácie využívajúce obrovské množstvo dát, je potrebné pochopiť, čo sa deje na pozadí databázového systému, v akých krokoch sa vykonávajú dotazy, ako prebieha ich spracovanie, a aké sú možnosti optimalizácie.

Databázové systémy využívajú hlavne dva typy optimalizácie dotazov. Prvou z nich je ***Optimalizácia založená na pravidlách*** (Rule-Based Optimization, RBO) a druhou je ***Cenovo orientovaná optimalizácia*** (Cost-Based Optimization, CBO). Oba spôsoby optimalizácie sú navrhnuté tak, aby určili najefektívnejší spôsob vykonania SQL dotazov. Hlavný rozdiel medzi nimi je spôsob prístupu, ktorý používajú pri vyhodnotení a výbere najvhodnejšieho plánu vykonania [12], [24].

Optimalizácia založená na pravidlách

Optimalizácia založená na pravidlách je založená na množine vopred definovaných pravidiel, ktoré optimalizátor využíva na určenie plánu vykonania daného SQL dotazu. To znamená, že ten istý SQL príkaz sa vždy vykoná rovnakým spôsobom. Tento spôsob

optimalizácie nezohľadňuje množstvo dát, nie je možné zmeniť vybraný plán vykonania a zároveň môžeme povedať, že plán vykonania je predvídateľnejší. Napríklad, ak je nad tabuľkou definovaný index, vytvorí sa pravidlo, ktoré navrhne použitie tohto indexu [24]. Pravidlá sú zotriedené tak, že v prípade existencie dvoch pravidiel, ktoré by sa dali využiť na definovaný príkaz, by sa použilo to pravidlo, ktoré ma najnižšiu hodnotu nákladov [27]. Optimalizácia založená na pravidlách je založená na heuristických optimalizačných pravidlách. Medzi tieto heuristiky môžeme zaradiť:

- Unikátny kľúč a primárny kľúč má najvyššiu prioritu
- Neunikátny kľúč je ďalší v poradí
- Kontrola indexovaných stĺpcov
- Celkové prehľadávanie tabuľky sa vykonáva na poslednom mieste [28]

Cenovo orientovaná optimalizácia

Cenovo orientovaná optimalizácia je optimalizácia založená na určení dotazu s minimálnymi nákladmi na jeho spracovanie. To znamená, že v prvom kroku sa zisťujú plány vykonania dotazu a následne sa vyberá najlacnejší plán vykonania tohto dotazu. *Náklady,* inak povedané *cena,* predstavujú internú hodnotu optimalizátora pre odhadované využitie zdrojov pre daný plán. Čím nižšie náklady na definovaný príkaz, tým bude plán vykonania efektívnejší [12], [24], [27]. Aby bolo možné určiť najlacnejší plán vykonania, musí mať optimalizátor k dispozícii všetky informácie o objektoch (indexoch, tabuľkách, ...), ku ktorým sa pristupuje v SQL príkaze a o systéme, na ktorom je SQL dotaz spustený. Tento spôsob optimalizácie je podporovaný vo väčšine databázových systémov a v databáze Oracle je možné ho využívať od verzie 7.0 [27].

Optimalizátor, plán vykonania, štatistiky

Nástroj, ktorý slúži na určenie najefektívnejšieho spôsobu vykonania dotazu sa nazýva ***optimalizátor***. Je to nástroj, ktorý obsahuje rôzne prístupové scenáre, z ktorých sa pomocou heuristiky vyberá najlepší, prípadne sub-optimálny plán vykonania. Medzi dôležité údaje pre rozhodnutie najvhodnejšieho plánu vykonania patria popisné údaje, štatistiky databázy, porovnanie odhadovaných nákladov na spracovanie blokov a náklady na sekvenčné spracovanie blokov. Takýto plán vykonania sa dočasne uloží do Shared pool pamäte inštancie, takže sa pri opätovnom použití daného príkazu môže využiť tento plán vykonania. ***Plán vykonania*** obsahuje jednotlivé kroky na vykonanie SQL príkazu. Tieto kroky sú vyjadrené prostredníctvom databázových operátorov, ktoré produkujú riadky.

O poradí týchto operátorov a ich implementácii rozhoduje optimalizátor. Plán vykonania využíva stromovú reprezentáciu, zatiaľ čo na displeji sa zobrazuje jeho tabuľková reprezentácia [26].

Nasledujúci príklad (*def. príkazu 1*) znázorňuje jednoduchý príkaz SELECT, ktorý zobrazuje identifikačné číslo nehody, dátum nehody identifikačné číslo poveternostných podmienok počas nehody a popis. Spája dve tabuľky, tabuľku *nehody* a tabuľku *poveternostné podmienky*, pričom vyberáme len tie záznamy, v ktorých bolo identifikačné číslo poveternostných podmienok 2 (hmla).

```
SELECT id_nehody, p2a_datum,id_poveternostnych_podmienok, popis
 FROM nehody
  JOIN p18_poveternostne_podmienky
   USING(id_poveternostnych_podmienok)
 WHERE id_poveternostnych_podmienok = 2;
```

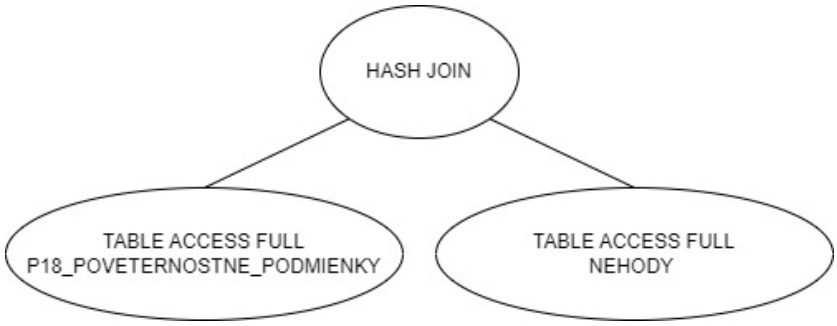
Definícia príkazu 1: Jednoduchý príkaz SELECT spájajúci dve tabuľky

Tabuľková reprezentácia plánu vykonania vyššie uvedeného príkazu SELECT je zobrazená na nasledujúcom *Obrázku 6*.

```
-------------------------------------------------------------------------------------------------
| Id  |  Operation          | Name                        |  Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 |  SELECT STATEMENT   |                             |  87448 |  4099K|  7744   (1)| 00:00:01 |
|*  1 |   HASH JOIN         |                             |  87448 |  4099K|  7744   (1)| 00:00:01 |
|*  2 |    TABLE ACCESS FULL| P18_POVETERNOSTNE_PODMIENKY |      1 |    25 |     3   (0)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| NEHODY                      |    699K|    15M|  7737   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Obrázok 6: Plán vykonania – tabuľková reprezentácia

Stromová reprezentácia vyššie uvedeného príkazu SELECT (*def. príkazu 1*) je znázornená na *Obrázku 7*.



Obrázok 7: Plán vykonania – stromová reprezentácia

Pri čítaní stromu vykonávania je potrebné začať listami stromu a postupovať zľava-doprava a postupne nahor. Vo vyššie uvedenom príklade začíname pohľadom na listy stromu, t.j. tabuľka *p18_poveternostne_podmienky* a *nehody*. Systém pristúpi ku všetkým záznamom tabuliek pomocou prístupovej metódy *Table access full* (je niekoľko

prístupových metód). Nad týmito záznamami sa ďalej vykoná operácia spojenia resp. *Hash join* (je niekoľko ďalších spôsobov spájania). Následne sa výsledná množina záznamov vráti používateľovi.

Výsledný plán vykonávania, ktorý vyberie optimalizátor, je len jeden z viacerých alternatívnych plánov vykonávania pre daný príkaz SELECT. Optimalizátor vyberie taký plán vykonávania, ktorý má najnižšie náklady na vykonanie, pričom náklady predstavujú internú hodnotu optimalizátora pre odhadované využitie zdrojov pre daný plán. Čím sú náklady na príkaz SELECT nižšie, tým bude plán vykonávania efektívnejší. Preto hovoríme, že optimalizátor je nákladovo orientovaný. Celkové náklady a náklady každej operácie sú uvedené v pláne vykonávania [26].

Plán vykonávania je možné ovplyvniť niekoľkými spôsobmi, čím sa ovplyvní aj rýchlosť dotazu. Databázový systém Oracle umožňuje nasledujúce spôsoby:

- Hinty (nápovedy), ktoré umožňujú syntaxou riadenú optimalizáciu
- Osnovy slúžia na uloženie plánu vykonania často sa opakujúcich dotazov
- Manažment aktuálnych štatistík
- Prepis dotazu
- Tvorba indexov [12]

Aby sa optimalizátor vedel správne rozhodovať a vybrať najefektívnejší plán vykonania je potrebné evidovať aktuálne ***štatistiky***, vďaka ktorým je možné odhadnúť náklady plánov vykonania pre definovaný SQL príkaz [12]. Tieto štatistiky predstavujú súbor údajov, ktoré popisujú objekty v databáze a databázu ako takú [27]. Generovanie štatistík, by sa malo vykonávať pre každý objekt, ktorý sa nachádza v SQL dotaze a po každej rozsiahlejšej zmene by sa mali štatistiky taktiež pregenerovať, aby boli vždy aktuálne. Databázový systém rozoznáva 3 druhy štatistík:

- ***Systémové štatistiky*** – zahŕňajú systémové informácie, napr. využitie CPU
- ***Tabuľkové štatistiky*** – zahŕňajú informácie o databázových tabuľkách, napr. počet riadkov
- ***Stĺpcové štatistiky*** – zahŕňajú informácie o stĺpcoch v databázových tabuľkách, napr. počet odlišných hodnôt v stĺpci [12]

1.4.1 Indexy

Na optimalizovanie výkonu vyhľadávania údajov v databázových tabuľkách je možné využiť indexy. Index je databázový objekt, vďaka ktorému je možné zrýchliť vykonávanie dotazov. Výhodou indexov je, že na lokalizovanie príslušných dát prostredníctvom indexov nie je potrebné prehľadávanie databázovej tabuľky riadok po riadku, avšak je potrebné vedieť, že pri operáciách *INSERT*, *DELETE* a *UPDATE* spôsobuje dodatočnú réžiu, pretože okrem upravenia dátových blokov je potrebné upraviť i samotný index [3].

Index môže byť tvorený jedným alebo viacerými stĺpcami tabuľky. Poradie stĺpcov má veľký vplyv na celkové náklady vykonávaného SQL dotazu. Okrem poradia stĺpcov je pri vytváraní indexu dôležité myslieť na podmienky definované v klauzule *WHERE*, podmienky spájania tabuliek a tiež atribúty, ktoré sú získané v klauzule *SELECT*. Index pozostáva z dvoch častí. Obsahuje kľúč a referenciu na dáta v databázovej tabuľke. V rámci indexov je možné zistiť selektivitu, ktorá je definovaná pomerom medzi počtom odlišných hodnôt k celkovému počtu riadkov. Jej rozsah je možné určiť na intervale <0, 1> [3].

Selektivita = počet odlišných hodnôt / celkový počet riadkov

Na vytvorenie indexu sa používa nasledujúca syntax (*def. príkazu 2*):

```
CREATE [ UNIQUE ] INDEX nazov_indexu
ON nazov_tabulky ( nazov_stlpca [ ASC | DESC ], … );
```

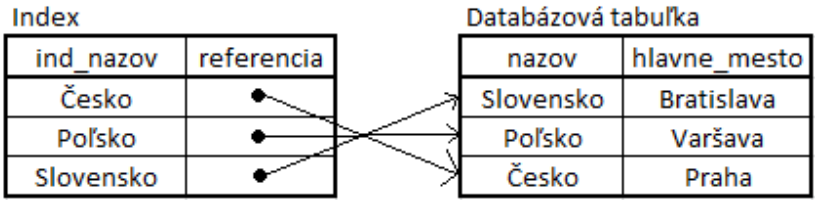
Definícia príkazu 2: Syntax na vytvorenie indexu

Pomocou nasledujúceho príkazu (*def. príkazu 3*) sa vytvorí index typu B+ strom s názvom *ind_nazov* nad stĺpcom *nazov* v tabuľke *krajiny*.

```
CREATE INDEX ind_nazov
 ON krajiny (nazov);
```

Definícia príkazu 3: Vytvorenie indexu typu B+strom

Obrázok 8 zobrazuje vyššie vytvorený index (*def. príkazu 3*).



Obrázok 8: Grafická reprezentácia indexu *ind_nazov*

Typy indexov

Existuje niekoľko typov indexov. Štandardne sa vytvára index typu *B+strom*. Okrem neho je možné vytvoriť aj iný typ indexovej štruktúry, ako napríklad *Hash index, Bitmap index, Funkcionálny index, Cluster index* a ďalšie. V prípade veľkých tabuliek je výhodné využiť rozdelenie tabuliek alebo indexov na partície, ktoré sa vytvárajú na základe zoznamu hodnôt intervalového rozdelenia alebo hodnoty hash funkcie [3]. Dátové sklady najčastejšie využívajú indexovú štruktúru typu *B+strom* a *Bitmap index.*

Bitmap index

Bitmap index je jeden z typov databázových indexov. Je veľmi rozšírený v dátových skladoch, v ktorých dochádza hlavne k vyhľadávaniu údajov a nie k ich aktualizácii. Výhodou Bitmap indexov je, že sú veľmi rýchle, pokiaľ počet odlišných hodnôt v stĺpci je malý, takže selektivita indexu je čo najmenšia. Bitmap index využíva bitmapy, ktoré sú reprezentované dvojrozmerným poľom. To znamená, že v prvom stĺpci sa uchováva odkaz na záznam v dátovom súbore, takzvané *ROWID,* a ďalšie stĺpce uchovávajú sekvenciu bitov pre jednotlivé hodnoty indexovaného stĺpca.

Nasledujúci príkaz (*def. príkazu 4*) slúži na vytvorenie Bitmap indexu s názvom *ind_batozina* nad stĺpcom *batožina* v tabuľke *letenka.*

```
CREATE BITMAP INDEX ind_batozina
 ON letenka (batozina);
```

Definícia príkazu 4: Vytvorenie indexu typu Bitmap

Tabuľka 2 reprezentuje databázovú tabuľku *letenka* spolu so záznamami a *Tabuľka 3* reprezentuje Bitmap index ako sekvenciu bitov nad stĺpcom *batožina*, ktorý nadobúda hodnoty áno / nie.

Tabuľka 2: Databázová tabuľka *letenka* spolu s dátami

ID letenky	ID cestujúceho	Cena	Batožina	Číslo sedadla
1001	1	54,60	Nie	23A
1002	2	70,50	Áno	15B
1003	3	58,20	Nie	20C
1004	4	63,40	Áno	14A

Tabuľka 3: Bitmap index pre *id_letenky*

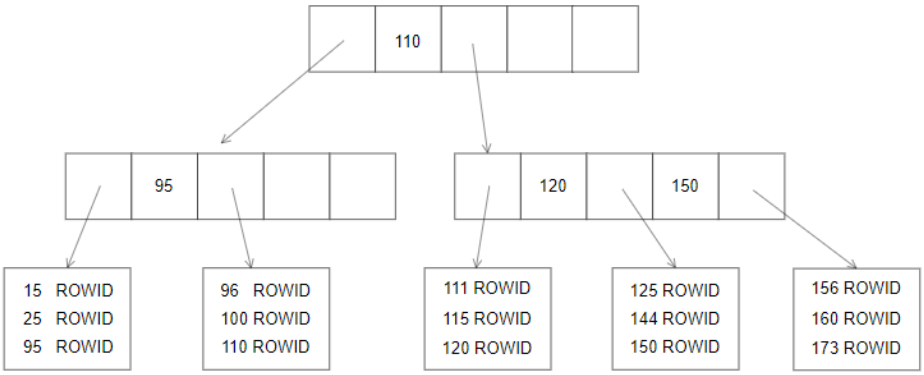
ID letenky	Bitmap index	
	Áno	Nie
1001	0	1
1002	1	0
1003	0	1

ID letenky	Bitmap index	
	Áno	Nie
1004	1	0

B+strom indexová štruktúra

Indexová štruktúra *B+strom* je štruktúra, ktorá zachováva efektivitu aj napriek často vykonávaným operáciám modifikácie tabuľky. B+strom indexová štruktúra je založená na údajovej štruktúre B-strom a je reprezentovaná formou vyváženého stromu, čo znamená, že cesta z koreňa do každého listu stromu má rovnakú dĺžku. B-strom má 3 typy vrcholov: **koreň** a **medziľahlý vrchol** uchovávajú hodnoty kľúča a smerníky, zatiaľ čo **list** odkazuje na dáta v dátovom súbore [3].

Medzi dôležitý parameter B-stromu patrí rád stromu, ktorý obsahuje maximálny počet kľúčov vrcholu, tak isto ovplyvňuje výšku stromu a v neposlednom rade aj veľkosť bloku indexu. Ďalším parametrom B-stromu je výška stromu, ktorá reprezentuje vzdialenosť od koreňa k listu stromu a ovplyvňuje počet blokov, ktoré je potrebné prehľadať v indexovej štruktúre, aby sa získala požadovaná položka [3]. *Obrázok 9* zobrazuje indexovú štruktúru B-stromu.



Obrázok 9: B-stom indexová štruktúra

Špeciálnym prípadom B-stromu je B+strom, ktorý je štandardne v databáze najčastejšie využívaná štruktúra. Rozdiel medzi týmito dvomi štruktúrami je v tom, že v B+ strome naviac dochádza k zreťazeniu vrcholov na listovej úrovni, čo zabezpečuje rýchle triedenie údajov [3].

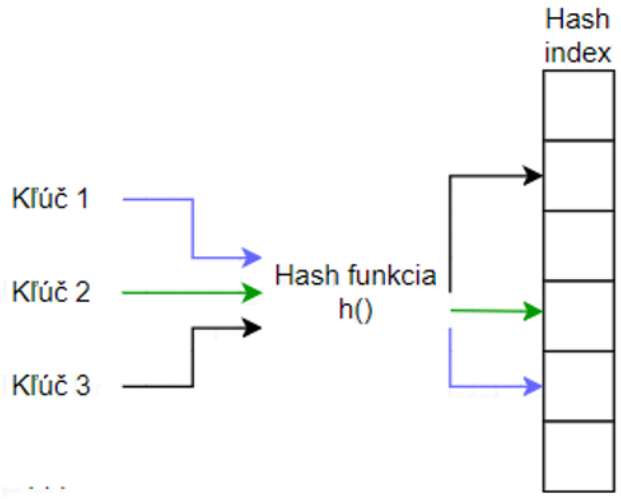
Nevýhodou B+stromu je, že vyžaduje viac úložného miesta na disku ako iné typy indexov, čo môže byť problém pre databázy s obrovským množstvom údajov alebo databázy s obmedzeným úložným priestorom. S rastúcim množstvom údajov výkonnosť tejto indexovej štruktúry klesá, keďže sa index musí taktiež aktualizovať. Na druhej strane

výhody tohto typu indexovej štruktúry prevažujú, a preto je to najčastejšie používaná štruktúra v databázach [3].

Hash index

Hash index je ďalší typ indexov, pomocou ktorého môžeme indexovať riadky relácie. Je zložený z množiny blokov. Veľkosť týchto blokov je zväčša jedna prípadne viac stránok. Tento spôsob indexovania využíva hash funkciu *h()*, pomocou ktorej sa hodnota kľúča mapuje na index bloku. Blok obsahuje dvojice (hodnota, smerník). Hash funkcia nie je jednoznačná a môže mapovať viaceré záznamy do toho istého bloku. V tom prípade je potrebné pridať preplňovací blok. Bloky by mali byť rovnomerne rozdelené a nemali by byť príliš veľké. Preto je potrebné hash funkciu navrhnúť tak, aby zabezpečila rovnomerné rozdelenie záznamov do jednotlivých blokov a mapovanie všetkých prípustných hodnôt stĺpca na index bloku [3]. *Obrázok 10* ilustruje mapovanie hash funkcie.

Vzhľadom na to, že nie je možné nájsť optimálnu hash funkciu a problémom spojeným s hash indexami, niektoré databázové systémy ako napr. Oracle tento druh indexu nepodporuje.



Obrázok 10: Mapovanie Hash funkcie

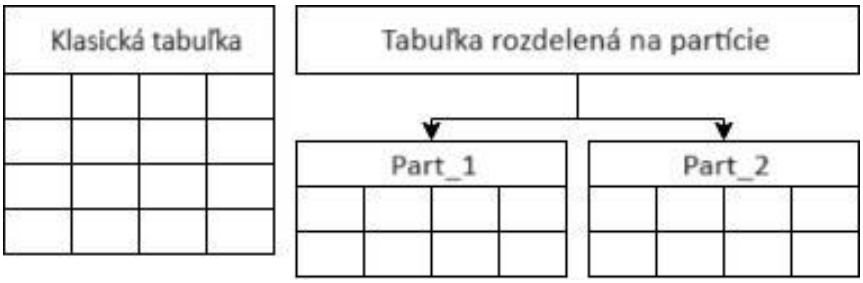
1.4.2 Partitioning

Nárast objemu dát v databázových tabuľkách môže viesť k spomaleniu a zníženiu efektivity operácií spracovania. Jednou z ďalších optimalizačných metód je rozdelenie tabuliek na menšie, ľahšie udržiavateľné časti nazývané ***partície*** [44], [45]. Táto optimalizačná metóda zjednodušuje údržbu databázy a zároveň prináša nárast výkonu a efektivity pri spracovaní dotazov [46], [47]. Partitioning je spôsob, ako rozdeliť databázové tabuľky a indexy, na menšie časti. Každá partícia je definovaná menom

a špecifickými nastaveniami. Partitioning znižuje celkové náklady spracovania dotazov [45]. Práca s tabuľkou rozdelenou na partície na nelíši od práce s klasickými tabuľkami. Stále je to jedna entita, ku ktorej môžeme pristupovať pomocou DML operácií rovnakým spôsobom ako pri nepartíciovaných tabuľkách.

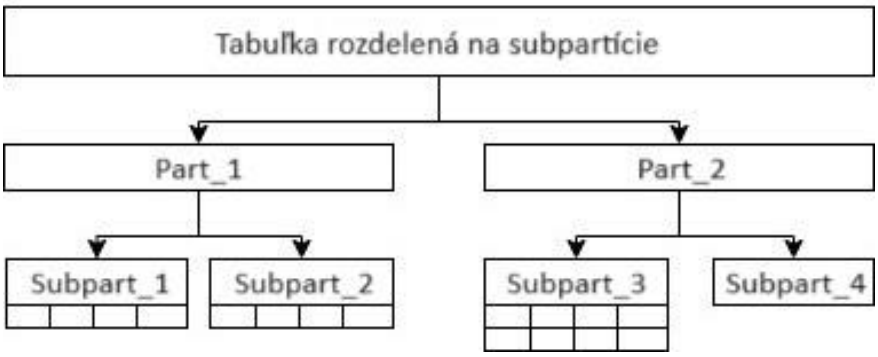
Za vytvorenie partícií je zodpovedný kľúč, ktorý môže pozostávať z jedného alebo viacerých stĺpcov tabuľky. Na základe hodnoty kľúča sa určuje, do ktorej partície sa daný záznam zaradí [48].

Grafická reprezentácia klasickej tabuľky a tabuľky rozdelenej na partície je znázornená na *Obrázku 11*.



Obrázok 11: Grafická reprezentácia klasickej tabuľky a tabuľky rozdelenej na partície

Databázové systémy poskytujú dve stratégie partíciovania. Prvou z nich je jednoduché (jednoúrovňové) partíciovanie, druhou stratégiou je kompozitné (viacúrovňové) partíciovanie. *Obrázok 11* reprezentuje prvý typ stratégie, čo znamená, že tabuľka je rozdelená len na partície. Druhá možnosť je znázornená na *Obrázku 12*, kde je možné vidieť, že každá partícia môže byť rozdelená do subpartícií.



Obrázok 12: Grafická reprezentácia tabuľky rozdelenej na subpartície

Partície a subpartície je možné vytvoriť buď pri definícii tabuľky, alebo dynamicky pri vkladaní nových záznamov do tabuľky. V databázových systémoch sa stretávame s troma základnými metódami distribúcie dát do partícií – Range, List a Hast partitioning.

Range partitioning

Range partitioning distribuuje záznamy do partícií na základe kľúčového stĺpca partícií. Tieto hodnoty musia patriť do definovaného intervalu. Syntax pre vytvorenie partícií s využitím range partitioning je znázornená v *def. príkazu 5*.

```
CREATE TABLE [schéma.]<nazov_tabulky>(
 <definicia_tabulky> )
PARTITION BY RANGE (<stĺpec> [, <stĺpec>, ...])
 [INTERVAL (<konštanta> | <výraz>)]...
( PARTITION [<nazov_particie>]
   VALUES LESS THAN (<hodnota>[, <hodnota>]...),
   [TABLESPACE <nazov_tablespace>]
 ...)
/
```

Definícia príkazu 5: Range partitioning

Range partitioning sa vyznačuje spojitým rozdelením, pri ktorom sa nevytvárajú medzery [48]. Predstavme si, že chceme dáta rozdeliť do partícií podľa rokov. Pri definovaní partícií vždy určíme hornú hranicu danej partície. Dolná hranica sa automaticky nastaví na hodnotu hornej hranice predchádzajúcej partície. Hranice sú usporiadané vzostupne, to znamená, že prvá partícia pokrýva všetky záznamy s nižšou hodnotou ako je definovaná hodná hranica. Poslednú partíciu je možné definovať s maximálnou hranicou [44]. Túto hranicu je potom potrebné označiť ako MAXVALUE. Každá hranica predstavuje otvorený interval [45], [47], [49].

List partitioning

List partitioning je technika, pri ktorej sa záznamy nerozdeľujú na základe rozsahu hodnôt ako v predchádzajúcom prípade, ale na základe presne definovaných (diskrétnych) hodnôt v stĺpci kľúča partícií [49]. V prípade, že vkladané záznamy nenadobúdajú žiadnu z definovaných hodnôt, a teda nepatria do žiadnej partície, je možné definovať poslednú partíciu ako DEFAULT, ktorá uchováva všetky záznamy s odlišnými hodnotami ako je definované. Syntax pre vytvorenie partícií s využitím list partitioning je znázornená v *def. príkazu 6*.

```
CREATE TABLE [schéma.]<nazov_tabulky>(
 <definicia_tabulky> )
PARTITION BY LIST (<stĺpec>) [AUTOMATIC](
 PARTITION [<nazov_particie>]
  VALUES (<hodnota>[, <hodnota>]...),
  [TABLESPACE <nazov_tablespace>]
 ...)
/
```

Definícia príkazu 6: List partitioning

Hash partitioning

Na rozdiel od predchádzajúcich techník tvorby partícií, hash partitioning využíva vlastný, interný hashovací algoritmus [47], [49]. Na základe tohto algoritmu sa záznamy delia do jednotlivých partícií. Syntax pre vytvorenie partícií s využitím hash partitioning je znázornená v *def. príkazu 7*.

```
CREATE TABLE [schéma.]<nazov_tabulky>(
 <definicia_tabulky> )
PARTITION BY HASH (<stĺpec> [, <stĺpec>, ...])
 [PARTITIONS <číslo>] [STORE IN <nazov_tablespace>
                    [, < nazov_tablespace >, ...]](
   PARTITION [<nazov_particie>]
    [TABLESPACE <nazov_tablespace >]
  ...)
/
```

Definícia príkazu 7: Hash partitioning

Hash partitioning sa využíva hlavne v prípade, keď nie je zrejmý rozsah alebo zoznam hodnôt. Zabezpečuje približne rovnakú veľkosť partícií. Je však ťažké očakávať súvislosť medzi partíciami a záznamami. Pri využití tejto techniky je možné definovať počet partícií, ktoré majú byť vytvorené. Toto číslo však musí byť mocninou čísla 2, takže táto vlastnosť výrazne ovplyvňuje vyváženosť veľkosti partícií [44], [48].

Touto problematikou sa bližšie zaoberá moja publikácia [HD9].

1.5 Funkcie

V databázových systémoch sa stretávame s troma základnými typmi funkcií. Prvým z nich sú štandardné funkcie, nasledujú agregačné funkcie a Ďalším typom sú analytické funkcie. Podrobnejší popis funkcií sa nachádza v nasledujúcich podkapitolách.

1.5.1 Štandardné funkcie

Kategória štandardných funkcií zahŕňa všetky vstavané funkcie databázového systému s výnimkou agregačných a analytických funkcií, ktorým sa venujeme v nasledujúcich kapitolách. Táto časť slúži ako prehľad najbežnejšie používaných štandardných funkcií. Vo všeobecnosti, je ich omnoho viac [42], [43]. Aj keď sa im v tejto práci nebudeme podrobne venovať, je dôležité mať o nich základné informácie.

Funkcie pre prácu s reťazcami

- *SUBSTR(reťazec, m [,n])* – funkcia vracia časť reťazca
- *UPPER(reťazec)* – funkcia na konverziu reťazca na veľké písmo
- *LOWER(reťazec)* – funkcia na konverziu reťazca na malé písmo
- *LENGTH(reťazec)* – funkcia vracia dĺžku reťazca
- ...

Funkcie pre prácu s číslami

- *ROUND(n [, m])* – funkcia na zaokrúhlenie hodnôt
- *TRUNC(n [, m])* – funkcia na orezanie hodnôt
- ...

Funkcie pre prácu s dátumom

- *ADD_MONTHS(d, n)* – funkcia vracia dátum d navýšený o hodnotu n mesiacov
- *LAST_DAY(d)* – funkcia vracia posledný deň v mesiaci
- *TO_CHAR(d [, formát])* – funkcia na konverziu dátumu na reťazec
- *TO_DATE(reťazec [, formát])* – funkcia na konverziu reťazca na dátum
- ....

Ďalšie funkcie

- *COALESCE(výraz1, ... , výrazN)* – funkcia vracia prvú NOT NULL hodnotu
- *NVL(výraz1, výraz2)* – funkcia na preformulovanie NULL hodnôt výrazu
- ...

1.5.2 Agregačné funkcie

Agregačné funkcie sú funkcie, ktorých výsledkom je jedna hodnota pre viac riadkov príkazu *SELECT*. Agregačné funkcie sa môžu nachádzať v klauzule *SELECT, HAVING* a *ORDER BY*. Bežne sa používajú spolu s klauzulou *GROUP BY*, v ktorej dochádza k rozdeľovaniu riadkov tabuľky do skupín. Pre každú skupinu je aplikovaná daná agregačná funkcia, ktorá pre každú skupinu vráti výsledok v podobe jedného riadku. Na poradí atribútov v časti *GROUP BY* nezáleží, pretože je spracovaná celá množina. V prípade, že klauzula *GROUP BY* nie je použitá, výsledkom je jeden riadok pre celú množinu záznamov. Použitie agregačnej funkcie v časti *HAVING* eliminuje skupiny riadkov, ktoré danú podmienku nespĺňajú [42].

Typy agregačných funkcií

V databázach sa stretávame s rôznymi typmi agregačných funkcií. Medzi päť základných funkcií patrí *COUNT(), SUM(), MIN(), MAX() AVG()* [42].

Medzi ďalšie agregačné funkcie patria:

- *VARIANCE()* – funkcia pre výpočet rozptylu
- *STDEV()* – funkcia pre výpočet odchýlky
- *LISTAGG()* – funkcia pre zreťazenie hodnôt oddelené daným oddeľovačom
- ...

Funkcia COUNT()

Funkcia *COUNT()* je funkcia, ktorá vracia počet riadkov vybranej skupiny záznamov. Táto funkcia môže mať niekoľko parametrov. Nasledujúca syntax (*def. príkazu 8*) vracia počet riadkov v danej skupine, v ktorých je definovaná hodnota. Nedefinovaná hodnota *NULL* nie je započítaná [42].

```
SELECT COUNT( nazov_stlpca )
 FROM nazov_tabulky;
```

Definícia príkazu 8: Syntax agregačnej funkcie COUNT()

Nasledujúca syntax (*def. príkazu 9*) vracia počet všetkých riadkov danej tabuľky:

```
SELECT COUNT( * )
 FROM nazov_tabulky;
```

Definícia príkazu 9: Agregačná funkcia COUNT() – počet všetkých riadkov

Nasledujúca syntax (*def. príkazu 10*) vracia počet unikátnych hodnôt v danom stĺpci tabuľky:

```
SELECT COUNT( DISTINCT nazov_stlpca )
 FROM nazov_tabulky;
```

Definícia príkazu 10: Agregačná funkcia COUNT() - počet unikátnych riadkov

Funkcia SUM()

Funkcia *SUM()* vracia celkový súčet hodnôt za skupinu zo stĺpca, ktorý obsahuje len číselné hodnoty [42]. Syntax tejto funkcie je nasledovná (*def. príkazu 11*):

```
SELECT SUM( nazov_stlpca )
 FROM nazov_tabulky;
```

Definícia príkazu 11: Syntax agregačnej funkcie SUM()

Funkcia AVG()

Funkcia *AVG()* vracia priemernú hodnotu zo skupiny hodnôt v stĺpci, ktorý obsahuje len číselné hodnoty [42]. Syntax tejto funkcie je nasledovná (*def. príkazu 12*):

```
SELECT AVG( nazov_stlpca )
 FROM nazov_tabulky;
```

Definícia príkazu 12: Syntax agregačnej funkcie AVG()

Funkcia MIN()

Funkcia *MIN()* vracia najmenšiu hodnotu zo skupiny hodnôt z daného stĺpca [42]. Syntax tejto funkcie je nasledovná (*def. príkazu 13*):

```
SELECT MIN( nazov_stlpca )
 FROM nazov_tabulky;
```

Definícia príkazu 13: Syntax agregačnej funkcie MIN()

Funkcia MAX()

Funkcia *MAX()* vracia najväčšiu hodnotu zo skupiny hodnôt z daného stĺpca [42]. Syntax tejto funkcie je nasledovná (*def. príkazu 14):*

```
SELECT MAX( nazov_stlpca )
 FROM nazov_tabulky;
```

Definícia príkazu 14: Syntax agregačnej funkcie MAX()

Príklad využitia agregačných funkcií

Nasledujúci príkaz *SELECT* demonštruje príklad na vyššie uvedené agregačné funkcie a ich využitie. Pre každú leteckú spoločnosť je potrebné zistiť počet letov, ktoré vykonala z letiska v Ríme (XVY) do letiska v Paríži (POX). Okrem toho je potrebné zistiť priemernú cenu letu, minimálnu a maximálnu cenu letu, a celkovú cenu letov. Dotaz tohto zadania môže vyzerať nasledovne (*def. príkazu 15*):

```
SELECT nazov_spolocnosti,
     COUNT( * ) AS pocet_letov,
     AVG( cena_letu ) AS priemerna_cena_letu,
     MIN( cena_letu ) AS minimalna_cena_letu,
     MAX( cena_letu ) AS maximalna_cena_letu,
     SUM( cena_letu ) AS celkova_cena_letov
 FROM let JOIN let_spolocnost
     USING (id_spolocnosti)
 WHERE letisko_z = 'XVY'
     AND letisko_do = 'POX'
 GROUP BY id_spolocnosti, nazov_spolocnosti;
```

Definícia príkazu 15: Využitie agregačných funkcií

Výsledok tohto dotazu je vyobrazený v *Tabuľke 4.*

Tabuľka 4: Výsledok použitia agregačných funkcií

Letecká spoločnosť	Počet letov COUNT()	Priemerná cena letu AVG()	Minimálna cena letu MIN()	Maximálna cena letu MAX()	Celková cena letov SUM()
Austrian	177	245,57	15,26	498,84	43 465,04
Lufthansa	181	230,37	1,21	497,06	41 696,98
EasyJet	361	258,03	2,66	499,29	93 147,87
Finnair	509	249,63	1,78	497,28	127 064,17
Darwin Airline	180	251,38	1,05	499,97	45 248,42
Jet2	179	265,33	5,60	499,56	47 494,75

NULL hodnoty a agregačné funkcie

Agregačné funkcie NULL hodnoty ignorujú, čo môže viesť k skresleným výsledkom. V prípade, že potrebujeme pracovať aj s NULL hodnotami, je potrebné využiť niektorú z funkcií, ktorá NULL hodnotu transformuje na určitú hodnotou. Nasledujúca syntax (*def. príkazu 16*) nahradí nedefinovanú hodnotu v danom stĺpci hodnotou 1 a vypočíta priemernú hodnotu v danom stĺpci.

```
SELECT AVG( NVL( nazov_stlpca, 1 ) )
 FROM nazov_tabulky;
```

Definícia príkazu 16: Syntax agregačnej funkcie s transformáciou NULL hodnoty

1.5.3 Analytické funkcie

Predchádzajúca kapitola bola zameraná na agregačné funkcie, ktorých výsledkom je jedna hodnota pre definovanú množinu riadkov. Tieto funkcie sú veľmi dôležité a užitočné v tom prípade, keď potrebujeme zosumarizovať hodnoty skupiny riadkov do jedného výsledku. Na druhej strane je v niektorých prípadoch potrebné zachovať jednotlivé riadky a získať súhrnnú hodnotu. Na to sa využívajú analytické funkcie, ktoré sa tiež nazývajú aj *funkcie okna (window functions).* Analytické funkcie sú prospešné na prípravu dát, výpočet medzivýsledkov, výpočet nových typov štatistík, ako napríklad kĺzavé mediány, poradie [43]. Inak povedané, analytické funkcie môžu pracovať s viacerými riadkami údajov, spracujú ich, no stále uchovajú všetky informácie v daných riadkoch [10]. Syntax analytických funkcií je nasledovná (*def. príkazu 17*):

```
SELECT {columns},
       {window_function} OVER ( PARTITION BY {partition_key}
                                ORDER BY {order_key} )
 FROM table_name;
```

Definícia príkazu 17: Všeobecná syntax analytických funkcií

pričom:

- *{columns}* – sú stĺpce dopytu
- *{window_function}* – je analytická funkcia
- *{partition_key}* – je stĺpec/stĺpce partícií
- *{order_key}* – je stĺpec/stĺpce zotriedenia
- *table_name* – je názov tabuľky, z ktorej sú čerpané údaje
- *OVER* – je kľúčové slovo, ktoré indikuje začiatok analytickej funkcie [10]

Typy analytických funkcií

V databázových systémoch sa stretávame s množstvom analytických funkcií. Okrem toho všetky agregačné funkcie (*AVG(), SUM(), COUNT(), atď.*) môžu byť použité ako analytické funkcie. V *Tabuľke 5* sú bližšie popísané ďalšie analytické funkcie:

Tabuľka 5: Prehľad analytických funkcií [10]

Funkcia	Popis
RANK()	udáva číslo riadku v rámci danej partície na základe ORDER BY, pričom v prípade zhody sú číslované rovnako, čím vznikajú medzery
DENSE_RANK()	udáva číslo riadku v rámci danej partície na základe ORDER BY, pričom v prípade zhody sú číslované rovnako, avšak medzery nevznikajú
ROW_NUMBER()	udáva číslo riadku v rámci danej partície
FIRST_VALUE()	vráti prvú hodnotu v usporiadanej množine hodnôt
LAST_VALUE()	vráti poslednú hodnotu v usporiadanej množine hodnôt
LAG(column, offset)	vráti hodnotu v stĺpci *column*, ktorá je získaná ako celočíselný posun pred aktuálnym riadkom na základe ORDER BY
LEAD(column, offset)	vráti hodnotu v stĺpci *column*, ktorá je získaná ako celočíselný posun po aktuálnom riadku na základe ORDER BY
NTH_VALUE()	vráti N-tú hodnotu z množiny hodnôt

Funkcia RANK()

Funkcia *RANK()* je analytická funkcia, ktorá udáva poradie v rámci danej partície na základe *ORDER BY*. V prípade, že sú hodnoty porovnávaných stĺpcov zhodné, tak ich poradie je ohodnotené rovnakým číslom. Ďalšie poradie sa vypočíta pričítaním zhodných riadkov k zhodnému poradiu, takže poradia nemusia byť po sebe idúce čísla, čím vznikajú medzery. Syntax analytickej funkcie *RANK()* je nasledovná (*def. príkazu 18*):

```
RANK() OVER ( [ query_partition_clause ] order_by_clause)
```

Definícia príkazu 18: Syntax analytickej funkcie RANK()

pričom:

- *order_by_clause* – je povinná časť a reprezentuje poradie riadkov v každej partícii, na ktorú je aplikovaná funkcia *RANK()*
- *query_partition_clause* – nie je povinná klauzula, rozdeľuje riadky na partície. V prípade, že sa táto klauzula vynechá, tak sa celá množina považuje za jednu partíciu

Funkcia DENSE_RANK()

Funkcia *DENSE_RANK()* je analytická funkcia, ktorá udáva poradie v rámci danej partície na základe *ORDER BY*. V prípade, že sú hodnoty porovnávaných stĺpcov zhodné, tak ich poradie je ohodnotené rovnakým číslom, a ďalšie poradie je nasledujúce číslo, čím nevznikajú medzery. Syntax analytickej funkcie *DENSE_RANK()* je nasledovná (*def. príkazu 19*):

```
DENSE_RANK() OVER ( [ query_partition_clause ] order_by_clause)
```

Definícia príkazu 19: Syntax analytickej funkcie DENSE_RANK()

pričom:

- *order_by_clause* – je povinná časť a reprezentuje poradie riadkov v každej partícií, na ktorú je aplikovaná funkcia *DENSE_RANK()*
- *query_partition_clause* – nie je povinná klauzula, rozdeľuje riadky na partície. V prípade, že sa táto klauzula vynechá, tak sa celá množina považuje za jednu partíciu

Funkcia ROW_NUMBER()

Funkcia *ROW_NUMBER()* je analytická funkcia, ktorá udáva číslo riadku v danej partícií na základe *ORDER BY*. Číslo riadku je sekvenčné celé číslo, ktoré je pridelené každému riadku, na ktorý je aplikovaná daná funkcia. Táto funkcia je užitočná pri stránkovaní. Syntax analytickej funkcie *ROW_NUMBER()* je nasledovná (*def. príkazu 20*):

```
ROW_NUMBER() OVER ( [ query_partition_clause ] order_by_clause)
```

Definícia príkazu 20: Syntax analytickej funkcie ROW_NUMBER()

pričom:

- *order_by_clause* – je povinná časť a reprezentuje poradie riadkov v každej partícií, na ktorú je aplikovaná funkcia *ROW_NUMBER()*
- *query_partition_clause* – nie je povinná klauzula, rozdeľuje riadky na partície. V prípade, že sa táto klauzula vynechá tak sa celá množina považuje za jednu partíciu

Porovnanie funkcií RANK(), DENSE_RANK() a ROW_NUMBER()

Nasledujúci príklad porovnáva tri analytické funkcie, ktoré slúžia na získanie poradia. Spoločnosť Czech_Airlines by chcela odmeniť svojich zamestnancov, pričom by

rada zohľadnila dĺžku ich trvania v spoločnosti. Preto by chcela získať menný zoznam zamestnancov utriedených podľa dátumu prijatia do spoločnosti. Na prvom mieste by mal byť zamestnanec / zamestnanci, ktorí nastúpili ako prví do zamestnania a na poslednom mieste tí, ktorí nastúpili najneskôr. Dotaz tohto zadania môže vyzerať nasledovne (*def. príkazu 21*):

```
SELECT ( meno || ' ' || priezvisko ) AS meno, datum_prijatia,
     ROW_NUMBER() OVER ( ORDER BY datum_prijatia )
          AS row_number_f,
     RANK() OVER (ORDER BY datum_prijatia) AS rank_f,
     DENSE_RANK() OVER (ORDER BY datum_prijatia) AS dense_rank_f
 FROM l_osoba JOIN l_zamestnanec
     USING(cislo_dokladu, typ_dokladu)
 WHERE id_spolocnosti = 9
     AND datum_zrusenia IS NULL
 ORDER BY datum_prijatia, priezvisko;
```

Definícia príkazu 21: Využitie analytických funkcií RANK(), DENSE_RANK(), ROW_NUMBER()

Výsledok tohto dotazu je znázornený v *Tabuľke 6*. Je to tabuľka aktuálnych zamestnancov spoločnosti Czech_Airlines. Okrem mien zamestnancov obsahuje aj dátum prijatia do zamestnania a ďalšie tri stĺpce, ktoré vznikli ako výsledok funkcií *ROW_NUMBER()*, *RANK()* a *DENSE_RANK()*. Pri použití funkcie *ROW_NUMBER()* jednoducho dostaneme poradie za sebou idúcich čísel bez ohľadu na zhodnosť hodnôt v danom stĺpci. Pri použití funkcie *RANK()* dostávajú riadky s rovnakými hodnotami zhodné poradie. V prípade, ak ku zhode nedôjde riadok nadobudne hodnotu, ako keby sme použili funkciu *ROW_NUMBER()*. Posledná funkcia *DENSE_RANK()* udáva rovnaké poradie v prípade zhody. V opačnom prípade riadok ohodnotí navýšením o hodnotu 1.

Tabuľka 6: Výsledok využitia analytických funkcií RANK(), DENSE_RANK(), ROW_NUMBER()

Meno	Dátum prijatia	ROW_NUMBER_F	RANK_F	DENSE_RANK_F
Marta Drotarova	17.02.1994	1	1	1
Juraj Gancarcik	23.08.1995	2	2	2
Elena Simakova	23.08.1995	3	2	2
Antonin Kalma	07.05.1996	4	4	3
Tomas Baculak	12.07.1996	5	5	4
Michal Kral	12.07.1996	6	5	4
Lucia Papezova	12.07.1996	7	5	4

Meno	Dátum prijatia	ROW_NUMBER_F	RANK_F	DENSE_RANK_F
Richard Vojtasak	16.07.1997	8	8	5
Rastislav Zizlavsky	16.07.1997	9	8	5
Milan Dubec	13.11.1997	10	10	6

Funkcia LAG()

Analytická funkcia *LAG(column, offset)* je funkcia, ktorá slúži na získanie hodnôt z predchádzajúcich N riadkov. Funkcia má dva parametre. Parameter *column* určuje stĺpec, z ktorého sa má hodnota získať a parameter *offset* udáva počet riadkov na získanie predchádzajúceho záznamu. Syntax analytickej funkcie *LAG()* je nasledovná (*def. príkazu 22*):

```
LAG( expression [, offset [, default]] )
 OVER ( [ query_partition_clause ] order_by_clause)
```

Definícia príkazu 22: Syntax analytickej funkcie LAG()

pričom:

- *expression* – je výraz / stĺpec, z ktorého sa majú čerpať hodnoty
- *offset* – je nepovinná časť, ktorá definuje posun. Ak nie je definovaný automaticky sa berie preddefinovaná hodnota 1
- *default* – je nepovinná časť, vyjadruje hodnotu, ktorá sa vráti v prípade prekročenia veľkosti tabuľky. V prípade ak nie je definovaná, tak bude vrátená predvolená hodnota *NULL*.
- *order_by_clause* – je povinná časť a reprezentuje poradie riadkov v každej partícií, na ktorú je aplikovaná funkcia *ROW_NUMBER()*
- *query_partition_clause* - nie je povinná klauzula, rozdeľuje riadky na partície. V prípade, že sa táto klauzula vynechá, tak sa celá množina považuje za jednu partíciu.

Využitie tejto funkcie môžeme demonštrovať na nasledujúcom príklade. Je potrebné vypočítať vzdialenosť, ktorú prešlo lietadlo z odletového letiska do príletového letiska, pričom máme k dispozícii súradnice bodov letu lietadla, kde došlo k zmene trajektórie. Celkovú vzdialenosť vypočítame ako súčet jednotlivých vzdialeností medzi týmito bodmi. Na to, aby sme vypočítali vzdialenosť medzi dvomi bodmi potrebujeme získať súradnice predchádzajúceho bodu lietadla. Let je označený identifikačným číslom 218860672. Dotaz, ktorým získame tieto dáta môže vyzerať nasledovne (*def. príkazu 23*):

```
SELECT id_letu, por_cislo_bodu, latitude, longitude,
     LAG( latitude, 1 ) OVER ( PARTITION BY id_letu
          ORDER BY id_letu, por_cislo_bodu)
               AS predchodca_latitude,
     LAG(longitude, 1) OVER ( PARTITION BY id_letu
          ORDER BY id_letu, por_cislo_bodu)
               AS predchodca_longitude
 FROM let_body_act_tab
 WHERE id_letu = 218860672;
```

Definícia príkazu 23: Využitie analytickej funkcie LAG()

Výsledok tohto dotazu je zobrazený v *Tabuľke 7*, pričom stĺpec *Vzdialenosť* je vypočítaná prostredníctvom nami definovanej funkcie.

Tabuľka 7: Výsledok dotazu s využitím analytickej funkcie LAG()

ID letu	Poradové číslo bodu	Latitude	Longitude	Predchodca latitude	Predchodca longitude	Vzdialenosť (NM)
218860672	0	36.98222	35.28028			
218860672	1	36.98222	35.28028	36.98222	35.28028	0
218860672	2	36.94056	35.21028	36.98222	35.28028	4.18740
218860672	3	36.91472	34.41139	36.94056	35.21028	38.3748
218860672	4	36.86278	33.29167	36.91472	34.41139	53.8596
218860672	5	36.88778	32.47833	36.86278	33.29167	39.0926
218860672	6	36.90528	31.74639	36.88778	32.47833	35.1602
218860672	7	36.89472	31.41611	36.90528	31.74639	15.8705
218860672	8	36.87639	31.22750	36.89472	31.41611	9.12410
218860672	9	36.90306	30.93556	36.87639	31.22750	14.1101
218860672	10	36.67083	30.90333	36.90306	30.93556	14.0291
218860672	11	36.90028	30.79278	36.67083	30.90333	14.7663

Funkcia LEAD()

Analytická funkcia *LEAD(column, offset)* je funkcia, ktorá slúži na získanie hodnôt z nasledujúcich N riadkov. Funkcia má dva parametre. Parameter *column* určuje stĺpec, z ktorého sa má hodnota získať a parameter *offset* udáva počet riadkov na získanie nasledujúceho záznamu. Môžeme povedať, že je to opačná funkcia k funkcii *LAG()*. Syntax analytickej funkcie *LEAD()* je nasledovná (def. príkazu 21):

```
LEAD( expression [, offset [, default]] )
 OVER ( [ query_partition_clause ] order_by_clause)
```

Definícia príkazu 24: Syntax analytickej funkcie LEAD()

pričom:

- *expression* – je výraz / stĺpec, z ktorého sa majú čerpať hodnoty
- *offset* – je nepovinná časť, ktorá definuje posun. Ak nie je definovaný automaticky sa berie preddefinovaná hodnota 1
- *default* – je nepovinná časť, vyjadruje hodnotu, ktorá sa vráti v prípade prekročenia veľkosti tabuľky. V prípade ak nie je definovaná, tak bude vrátená predvolená hodnota *NULL*.
- *order_by_clause* – je povinná časť a reprezentuje poradie riadkov v každej partícií, na ktorú je aplikovaná funkcia *ROW_NUMBER()*
- *query_partition_clause* - nie je povinná klauzula, rozdeľuje riadky na partície. V prípade, že sa táto klauzula vynechá tak sa celá množina považuje za jednu partíciu.

1.5.4 Rozšírenie klauzuly *GROUP BY*

Špeciálnymi funkciami, ktoré sa radia medzi rozšírenia klauzuly *GROUP BY* sú funkcie ROLLUP() a CUBE().

Vďaka nim je možné priamo v SQL ľahko vytvoriť reportované agregácie bez toho, aby sa použili príkazy reportu v SQL*Plus. Funkcie sú špecifické tým, že samotná funkcia je v časti *GROUP BY* [43].

Rozšírenie ROLLUP()

Funkcia *ROLLUP()* umožňuje vypočítať medzivýsledky agregačných funkcií v rámci dimenzií, ktoré reprezentujú viaceré úrovne. Okrem toho umožňuje vypočítať aj celkovú hodnotu agregačnej funkcie. Funkcia *ROLLUP()* rozširuje klauzulu *GROUP BY*, pričom je potrebné spomenúť, že poradie stĺpcov je dôležité, pretože ovplyvňuje výsledok. Syntax funkcie *ROLLUP()* je nasledovná (*def. príkazu 25*):

```
SELECT zoznam_stlpcov
 FROM nazov_tabulky
  ....
 GROUP BY ROLLUP(zoznam_stlpcov);
```

Definícia príkazu 25: Syntax ROLLUP()

Nasledujúci príklad (*def. príkazu 26*) reprezentuje praktickú ukážku využitia funkcie *ROLLUP()* v príkaze *SELECT*.

```
SELECT id_spolocnosti, nazov_typu, COUNT( * )
 FROM lietadlo JOIN typ_lietadla
     USING( id_typu )
 GROUP BY ROLLUP( id_spolocnosti, nazov_typu );
```

Definícia príkazu 26: Využitie ROLLUP()

Počet jednotlivých typov lietadiel pre každú spoločnosť je jeden z výsledkov, ktoré získame. Navyše pre každú leteckú spoločnosť sa získa celkový počet lietadiel, a taktiež aj celkový počet lietadiel všetkých spoločností. *Tabuľka 8* znázorňuje výslednú tabuľku spolu s dátami.

Tabuľka 8: Výsledná množina údajov získaných pomocou ROLLUP()

Spoločnosť	Typ lietadla	Počet lietadiel
Ryanair	Boeing 737-800s	10
	Boeing 737 MAX 200s	15
		25
Lufthansa	Boeing 747-400	13
	Airbus A380	14
		27
Wizz Air	Airbus A320	12
	Airbus A321	15
		27
		79

Hodnoty sú vypočítané pre:

- *GROUP BY* id_spolocnosti, nazov_typu
- *GROUP BY* id_spolocnosti
- Pre celú tabuľku

Rozšírenie CUBE()

Funkcia *CUBE()* umožňuje vypočítať medzivýsledky agregačných funkcií v rámci dimenzií pre všetky možné kombinácie jej stĺpcov. Okrem toho umožňuje vypočítať aj celkovú hodnotu agregačnej funkcie. Funkcia *CUBE()*, podobne ako funkcia *ROLLUP()*, rozširuje klauzulu *GROUP BY*, pričom je potrebné spomenúť, že poradie stĺpcov je dôležité, pretože ovplyvňuje výsledok [43].

Nasledujúci príklad (*def. príkazu 27*) reprezentuje praktickú ukážku využitia funkcie *CUBE()* v príkaze *SELECT*.

```
SELECT id_spolocnosti, nazov_typu, COUNT( * )
 FROM lietadlo JOIN typ_lietadla
     USING( id_typu )
 GROUP BY CUBE( id_spolocnosti, nazov_typu );
```

Definícia príkazu 27: Využitie CUBE()

Počet jednotlivých typov lietadiel pre každú spoločnosť je jeden z výsledkov, ktoré získame. Navyše pre každú leteckú spoločnosť sa získa celkový počet lietadiel, ďalej sa získa počet lietadiel daných typov, a taktiež aj celkový počet lietadiel všetkých spoločností. *Tabuľka 9* znázorňuje výslednú tabuľku spolu s dátami.

Tabuľka 9: Výsledná množina údajov získaných pomocou CUBE()

Spoločnosť	Typ lietadla	Počet lietadiel
Ryanair	Boeing 737-800s	10
	Boeing 737 MAX 200s	15
		25
Lufthansa	Boeing 747-400	13
	Airbus A380	14
		27
Wizz Air	Airbus A320	12
	Airbus A321	15
		27
	Boeing 737-800s	10
	Boeing 737 MAX 200s	15
	Boeing 747-400	13
	Airbus A380	14
	Airbus A320	12
	Airbus A321	15
		79

Hodnoty sú vypočítané pre:

- *GROUP BY* id_spolocnosti, nazov_typu
- *GROUP BY* id_spolocnosti
- *GROUP BY* nazov_typu
- Pre celú tabuľku

2 Dáta

Dizertačná práca je zameraná na analytické spracovanie dát, ktorej cieľom je získať užitočné informácie z údajov, na základe optimalizácie dotazov a vytvorených reportov. Optimalizácia spočíva v navrhnutí vhodných indexových štruktúr využívaných v dátových skladoch, ktoré pomôžu zvýšiť rýchlosť dotazov. Analytické spracovanie bude vykonávané nad reálnymi dátami, z ktorých sme vybrali dáta z leteckej dopravy a dáta z dopravných systémov, nad ktorými budú vytvárané rôzne štatistiky.

2.1 Dáta z leteckej dopravy

V rámci výskumu pre spoločnosť Helios sme pracovali s reálnymi dátami z leteckej dopravy, ktoré nám boli poskytnuté organizáciou EUROCONTROL. EUROCONTROL je celoeurópska, civilno-vojenská organizácia, ktorá sa snaží o to, aby európske letectvo bolo efektívnejšie, bezpečnejšie a s minimálnym dopadom na životné prostredie [4]. Cieľom projektu bola analýza dát, na základe ktorej bolo možné určiť vplyv letov na zmenu podnebia či vynaložené náklady na jednotlivé lety.

Dáta, s ktorými sme pracovali reprezentovali jednotlivé lety nad Európou počas 4 rokov. Všetky záznamy boli uložené v súboroch v CSV formáte. Dáta boli zozbierané z rokov 2015 – 2018, pričom pre každý rok sme mali k dispozícii údaje za každý kvartál. Dáta obsahovali letové informačné regióny (FIR), reguláciu a riadenie leteckých informácií (AIRAC), plánované a aktuálne preletené lety spolu so všetkými informáciami o nich. V *Tabuľke 10* je zobrazená štruktúra bodov aktuálnych letov z roku 2015 z prvého kvartálu.

Tabuľka 10: Štruktúra dát z leteckej dopravy

ID letu	Poradové číslo bodu	Čas	Letová výška	Latitude	Longitude
184408024	0	01-03-2015 1:00:00	0	41.98000	-87.90500
184408024	1	01-03-2015 1:47:31	330	42.06361	-84.32950
184408024	2	01-03-2015 2:05:33	330	42.03611	-80.75360
...	...	...	...	...	...
184408024	33	01-03-2015 8:19:07	0	52.30806	4.76417
184408028	0	01-03-2015 0:03:00	0	49.00972	2.54778
184408028	1	01-03-2015 0:21:00	96	49.01222	2.36139
184408028	2	01-03-2015 0:22:07	143	49.03833	2.14667
...	...	...	...	...	...
184408028	13	01-03-2015 1:10:51	0	45.63000	8.72306
...	...	...	...	...	...

V prvom rade bolo potrebné spracovať obrovské množstvo dát a transformovať ho do vhodnej podoby na import do databázy. Bolo potrebné identifikovať problémové a chybné dáta a opraviť ich. Medzi problémy, ktoré sa vyskytli počas importu údajov do databázy môžeme zaradiť rôzne formáty dátumových reťazcov, ktoré bolo potrebné zjednotiť, nedefinované hodnoty a neúplnosť údajov.

Po odstránení týchto problémov sme mohli pristúpiť k importu všetkých dát do databázy prostredníctvom importu pomocou dátovej pumpy. Je to najrýchlejší spôsob importu spomedzi zvyšných typov importov. Následne sme začali pracovať na výskume, ktorý spočíval vo výpočte prejdených vzdialeností jednotlivých letov, kde sme využili príslušné analytické funkcie.

V rámci toho bolo potrebné vytvoriť funkciu, ktorá umožní vypočítať vzdialenosť medzi dvoma bodmi pričom bolo potrebné zohľadniť zakrivenie Zeme. Funkcia je znázornená na nasledujúcom kóde (*def. príkazu 28*).

Do funkcie vstupujú 4 parametre, ktorými sú súradnice dvoch bodov zložené zo zemepisnej šírky a dĺžky každého bodu. Funkcia obsahuje konštantu PI a polomer zeme. Súradnice sa následne premenia na radiány a vypočíta sa druhá mocnina polovice dĺžky tetivy medzi danými bodmi. Následne sa vypočíta uhlová vzdialenosť a konečná vzdialenosť je dosiahnutá vynásobením týchto dvoch medzivýpočtov.

Táto funkcia bola použitá pri výpočte celkovej vzdialenosti letu, ktorú sme získali súčtom jednotlivých vzdialeností. K dispozícii sme mali body letu, ktoré sa zapisovali vždy pri zmene trajektórie lietadla. Prostredníctvom analytickej funkcie *LAG()* sme získali súradnice predchádzajúcich bodov, takže sme mali vždy k dispozícii súradnice oboch bodov letu. Táto vzdialenosť bola počítaná pre vybrané lety.

```
CREATE OR REPLACE FUNCTION distance (
    lat1 IN NUMBER,
    lat2 IN NUMBER,
    lon1 IN NUMBER,
    lon2 IN NUMBER)
RETURN NUMBER IS
 p_pi         CONSTANT NUMBER    := 3.14159265358979323846264643;
 earth_radius CONSTANT NUMBER    := 6371000;

 half_chord_length    NUMBER:= 0.0;
 angular_distance     NUMBER:= 0.0;
 p_distance           NUMBER:= 0.0;
BEGIN
 half_chord_length :=
          POWER(SIN( ((lat2-lat1) * p_pi/180)/2 ), 2 ) +
                (COS( lat1 * p_pi/180 ) *
                 COS( lat2 * p_pi/180 ) *
                 POWER(SIN( ((lon2-lon1) * p_pi/180)/2 ),2 ) );
 angular_distance := 2 * ATAN2( SQRT( half_chord_length ),
                                SQRT( 1-half_chord_length ) );
 p_distance := earth_radius * angular_distance;
 RETURN p_distance;
END distance;
```

Definícia príkazu 28: Funkcia na výpočet vzdialenosti medzi dvoma bodmi

S nedefinovanými záznamami sme sa stretli aj pri výpočte vzdialeností, kde sme zistili, že niektoré súradnice príletových a odletových stredísk nie sú vyplnené pri definícii počiatočných a koncových súradníc lietadiel. Preto bolo potrebné hodnoty týchto súradníc doplniť tak, že sme vytvorili procedúru, ktorá zaistí vyplnenie všetkých nedefinovaných počiatočných a koncových súradníc lietadiel na základe súradníc daných letísk.

Okrem výpočtu vzdialeností jednotlivých letov, sme ďalej pracovali na určovaní letov do príslušného letového regiónu na základe súradníc v jednotlivých bodoch letu. Na riešenie tohto problému sme vytvorili príkaz UPDATE, ktorý nastavil hodnotu atribútu fir_id, ktorú sme získali na základe času vstupu a výstupu lietadla z danej oblasti. Úryvok tohto kódu je zobrazený na *def. príkazu 29*.

```
UPDATE flight_points_act_tab
 SET fir_id =
     (SELECT flight_fir_act_tab.fir_id
       FROM flight_fir_act_tab
        WHERE flight_fir_act_tab.ectrl_id =
              flight_points_act_tab.ectrl_id
         AND flight_fir_act_tab.entry_time <=
             flight_points_act_tab.time_over
         AND flight_fir_act_tab.exit_time >
             flight_points_act_tab.time_over)
 WHERE ectrl_id IN (SELECT ectrl_id
                      FROM weather_flights_tab);
```

Definícia príkazu 29: Zaradenie letu do správneho FIR

2.2 Dáta z dopravných systémov

Ďalšie údaje, ktoré tvoria základ pre vyhodnocovanie, experimentovanie a overovanie vytvorených metód, indexov a funkcií, sú reálne dáta z dopravných systémov Českej republiky. Tieto datasety sú verejne dostupné na webovej stránke polície Českej republiky https://www.policie.cz/default.aspx. K dispozícii sú údaje z dopravných systémov pre každý mesiac od roku 2015 do 2023. Datasety obsahujú niekoľko súborov, ktoré zaznamenávajú informácie o tom, kde vznikli nehody, na akom type cestnej komunikácie došlo k nehode, aký druh vozidla spôsobilo nehodu a mnoho ďalších parametrov. Súčasťou množiny dát je aj súbor s informáciami o chodcoch. Poskytnuté údaje sú v podstate modelom dátového skladu, kde je centrálna tabuľka faktov s odkazom na jednotlivé dimenzie (číselníky) v osobitných súboroch. V *Tabuľke 11* je znázornená štruktúra súboru o chodcoch. Súbor obsahuje údaje o kategórii chodca, stave chodca, jeho chovaní a situácii na mieste nehody.

Tabuľka 11: Štruktúra údajov z dopravných nehôd (chodci)

Identifikačné číslo	Kategória chodca	Stav chodca	Chovanie chodca	Situácia v mieste nehody
2100210073	1	1	1	8
2100210101	2	1	1	4
2100210105	2	2	3	0
2100210131	2	1	1	1
...	...	...	...	...

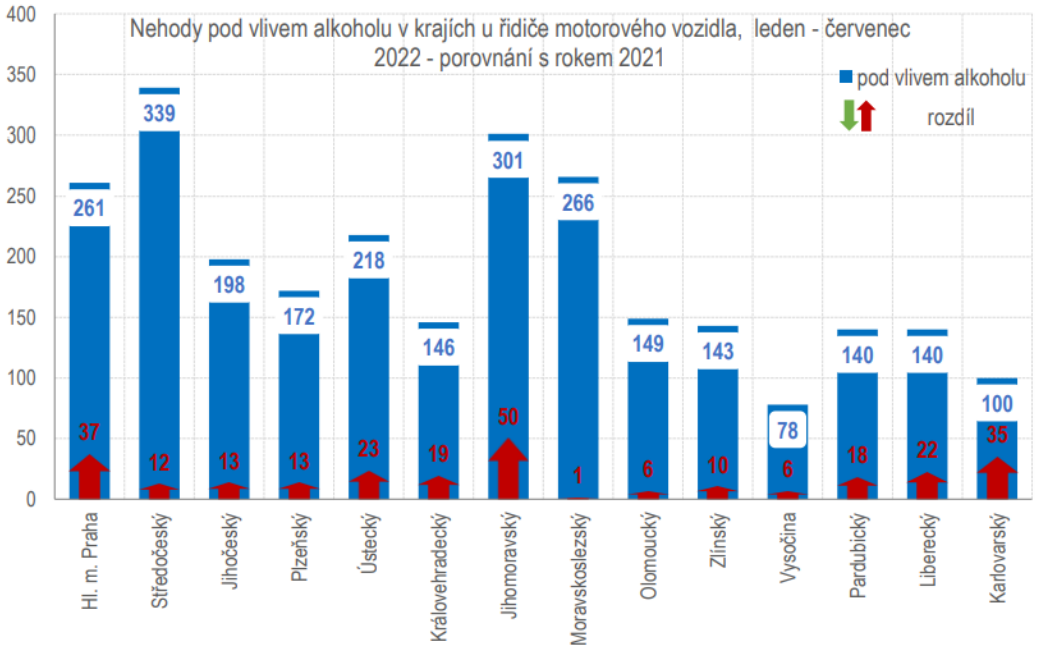
Identifikačné číslo	Kategória chodca	Stav chodca	Chovanie chodca	Situácia v mieste nehody
2100210725	5	0	0	0
2100210746	1	1	1	0
2100210754	1	4	2	10
2100210767	2	1	1	4

Na danej webovej stránke sa nachádza aj popis položiek jednotlivých súborov. Každé číslo v stĺpci má svoj význam. Pre každý stĺpec je vytvorený číselník, na základe ktorého je možné určiť dané kategórie. *Tabuľka 12* znázorňuje číselník pre stĺpec *Kategória chodca*.

Tabuľka 12: Číselník pre stĺpec *Kategória chodca*

Kategória chodca	Popis
1	Muž
2	Žena
3	Dieťa
4	Skupina detí
5	Iná skupina

Okrem samotných dát je možné prezrieť si informácie o nehodovosti za jednotlivé mesiace vo vybranom roku. Je to PDF dokument, ktorý obsahuje štatistiky, grafy pre rôzne merania. *Obrázok 13* reprezentuje graf znázorňujúci porovnanie počtu nehôd pod vplyvom alkoholu v jednotlivých krajoch Českej republiky u šoférov motorových vozidiel za obdobie januára až júla rokov 2021 a 2022.



Obrázok 13: Nehody spôsobené pod vplyvom alkoholu v jednotlivých krajoch (január - júl)

Zdroj: https://www.policie.cz/

3 Výskum a vlastný prínos

Nasledujúca kapitola sa venuje detailnému rozpracovaniu nášho vlastného prínosu. Každý typ výskumu predstavuje samostatnú časť, v ktorej je zhrnutá podstata a výstupy experimentálnych štúdií. Výsledky jednotlivých tém a experimentov boli transformované do článkov publikovaných na rôznych medzinárodných konferenciách.

Experimenty boli vykonávané na databázovom prostredí Oracle, pretože Oracle je výkonný a všestranný databázový systém, ktorý ponúka širokú škálu funkcií, ktoré sú relevantné pre analytické prostredie, poskytuje technológie zamerané na analytické spracovanie, ako napríklad Oracle In-Memory Analytics, Oracle Data Science Cloud, Oracle Data Warehouse. Okrem toho má množstvo výhod. Je jedným najrozšírenejších databázových systémov. Poskytuje riešenia s vysokým výkonom a škálovateľnosťou, zaručuje vysokú dostupnosť a spoľahlivosť a zároveň ponúka aj rozsiahly súbor nástrojov pre zálohovanie a obnovenie dát. Z pohľadu analýzy dát, dokáže spracovávať veľké objemy dát s vysokou rýchlosťou a efektivitou čím je ideálny pre Big Data analýzu a aplikácie.

Parametre počítača, ktorý bol využívaný na výskum, boli nasledovné:

- ***Operačný systém*** – Microsoft Windows Server 2020 Standard
- ***RAM*** – 48 GB
- ***SSD*** – 1 TB
- ***Procesor*** – Intel Xeon; CPU E5620; 2,40 GHz (4 jadrá); 8 logických procesorov

Parametre siete:

- ***Rýchlosť načítania*** – 5 Mbps
- ***Rýchlosť sťahovania*** – 50 Mbps
- ***Oneskorenie*** – 11,5 ms

3.1 Metódy importu a exportu dát

Dizertačná práca je zameraná na analytické spracovanie dát, v ktorom sa spracováva obrovské množstvo údajov. Preto je potrebné správne si zvoliť metódy importu a exportu dát. V rámci ďalšieho výskumu sme sa zamerali na porovnanie metód importu dát do

databázy a exportu dát z databázy pre transakčné spracovanie. Celou touto problematikou sa venuje publikácia [HD4]. Cieľom výskumu bolo:

- porovnanie rýchlostí jednotlivých metód importu a exportu dát
- určenie najefektívnejšej možnosti importu a exportu dát
- porovnanie rýchlostí vykonania príkazu *SELECT* v interných a externých tabuľkách

Všetky typy experimentov boli testované na troch rôznych dátových setoch, ktoré sú znázornené v *Tabuľke 13*. Najmenší z nich obsahoval približne 20 000 záznamov, stredný obsahoval 6,5 milióna záznamov a najväčší dátový set mal takmer 32,5 milióna záznamov.

Tabuľka 13: Dátové sety

Dátový set	Počet záznamov
Malý dátový set	19 532
Stredný dátový set	6 500 000
Veľký dátový set	32 400 000

3.1.1 Metódy importu dát

Metódy importu dát sú metódy, ktoré umožňujú obnoviť, respektíve vložiť dáta zo zdrojového súboru do databázy. V rámci výskumu boli testované a porovnávané tri typy importných metód:

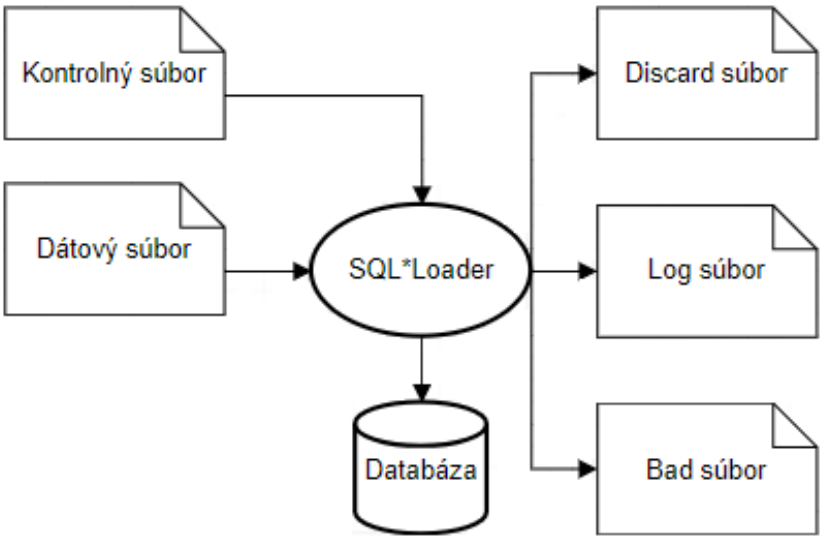
- Metóda SQL*Loader
- Metóda IMP
- Metóda IMPDP

Metóda SQL*Loader

SQL*Loader je nástroj databázového systému Oracle, vďaka ktorému je možný import údajov zo zdrojového súboru do databázovej tabuľky. Súbor môže byť v rôznych formátoch, ktoré boli vytvorené ako výstup z iných systémov. SQL*Loader dokáže čítať súbory, v ktorých sú dáta oddelené definovaným oddeľovačom. Preto sú dáta väčšinou v CSV formáte [18], [50]. SQL*Loader rozlišuje dva typy metód:

- ***CONVENTIONAL*** – je predvolená metóda systému. Zo spracovaných údajov sa generujú príkazy *INSERT*, pričom sa vykonávajú v rámci transakcií a generujú *REDO/UNDO*

- ***DIRECT*** – je metóda, ktorá zapisuje dáta priamo do dátových súborov. Používa tzv. *High Water Mark,* čo je ukazovateľ na posledný blok tabuľky



Obrázok 14 Koncept metódy SQL*Loader

Obrázok 14 reprezentuje koncept metódy SQL*Loader v databázovom systéme. Ako je možné vidieť, vstupom do tejto metódy sú dva typy súborov kontrolný súbor (.ctl) a dátový súbor. SQL*Loader načíta dáta z dátového súboru do databázy a vygeneruje tri typy súborov logovací súbor (.log), bad súbor (.bad) a discard súbor (.dsc). Logovací súbor sa vytvára vždy a obsahuje detailné informácie o priebehu importu dát do databázy, taktiež zahŕňa popis chýb, ktoré sa vyskytli počas importu. Záznamy, ktoré neboli načítané z dôvodu nevyhovujúcich kritérií definovaných v kontrolnom súbore sú uložené do discard súboru. Chybné súbory sa nachádzajú v bad súbore.

Import pomocou nástroja SQL*Loader prebieha v určitých krokoch. V prvom rade je potrebné mať údaje v súbore v správnom formáte. Ďalej je potrebné vytvoriť databázovú tabuľku s príslušnými stĺpcami, dátovými typmi a ďalšími obmedzeniami. Následne sa vytvorí kontrolný súbor, ktorý popisuje ako budú dáta v súbore interpretované a tiež obsahuje ďalšie možnosti načítania údajov. Na záver sa spustí nástroj SQL*Loader s definovaným súborom [50]. *Def. príkazu 30* zobrazuje príkaz pre spustenie SQL*Loader.

```
$ SQLLDR login/password@connect_string
  CONTROL=control_file_name.ctl
```

Definícia príkazu 30: Spustenie metódy SQL*Loader

Nasledujúci kód zobrazuje štruktúru kontrolného súboru (*def. príkazu 31*).

```
OPTIONS(
     DIRECT    =    {TRUE | FALSE}
     PARALLEL  =    {TRUE | FALSE}
     SKIP      =    n
      . . .)
LOAD DATA
CHARACTERSET UTF8
INFILE 'CESTA_K_SUBORU' "str'\n'"
BADFILE 'CESTA_K_SUBORU'
DISCARDFILE 'CESTA_K_SUBORU'
APPEND
INTO TABLE NAZOV_TABULKY
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"' AND '"'
TRAILING NULLCOLS(
          stlpec1 ,
          stlpec2 ,
          stlpec3 ,
          . . . )
```

Definícia príkazu 31: Štruktúra kontrolného súboru

Metóda IMP

Metóda importu IMP sa využíva na obnovenie databázy alebo na získanie prenesených údajov z jednej databázy do inej. Funkcionalita IMP je nástroj, ktorý využíva klient-server architektúru. To znamená, že dump súbor je načítaný z lokálneho adresára na klientskom zariadení, kde sa nachádza aj nástroj [17] [20]. Metóda IMP načíta dátové bloky zo súboru vygenerovaného metódou EXP. Vytvoria sa príkazy *INSERT* a ďalšie DDL príkazy. Dáta sa následne vložia do databázy. Tieto DDL príkazy sú vykonávané na strane servera. Ak dôjde k chybe komunikácie medzi klientom a serverom import končí chybou. Ďalším obmedzením tohto typu metódy je rýchlosť importu, ktorá závisí od rýchlosti sieťovej komunikácie. Z toho dôvodu tento typ metódy nie je efektívny.

Nasledujúci príkaz (*def. príkazu 32*) slúži na spustenie metódy IMP a vloženie údajov do databázy.

```
$ IMP login/heslo@connect_string
  FILE=nazov_export_suboru.exp
```

Definícia príkazu 32: Spustenie metódy IMP

Názov exportného súboru je názov dump súboru, ktorý bol vytvorený metódou EXP. Tento súbor je v binárnom formáte a obsahuje objekty v definovanom poradí. Typ definície je umiestnený v súbore na prvom mieste. Potom nasledujú definície tabuliek, dáta, indexy, pohľady, procedúry a posledná časť súboru obsahuje bitmap indexy a ďalšie typy indexov. Príkaz IMP obsahuje množstvo ďalších parametrov, ako napríklad veľkosť súboru, log súbor, štatistiky, tabuľky a ďalšie.

Metóda IMPDP

Tento typ importu je známy ako import dátovej pumpy. Je to taktiež nástroj, ktorý sa používa na načítanie dump súboru, ktorý je vygenerovaný nástrojom na export, do cieľovej databázy. Súbor je v binárnom tvare [17]. Rozdiel medzi touto metódou a klasickou metódou IMP je, že IMPDP metóda sa vykonáva len na strane servera. Preto odpadá komunikácia medzi klientom a serverom [20]. Pri použití metódy IMPDP, je potrebné vytvoriť mapovací objekt na prístup do dátového úložiska na strane servera. Používateľ zvyčajne nemá prístup do dátového úložiska, čo je dôležité hlavne kvôli bezpečnosti.

Nasledujúce kroky popisujú použitie importnej metódy IMPDP. V prvom kroku je potrebné vytvoriť adresár na strane servera. Následne sa vytvorí Oracle adresár. Tento Oracle adresár sa mapuje na vytvorený adresár na serveri. Oracle adresár je globálny databázový objekt, ktorého vlastník je SYS. Mapovanie sa vykoná nasledujúcim príkazom (*def. príkazu 33*):

```
CREATE DIRECTORY nazov_adresara AS cesta_k_adresaru;
```

Definícia príkazu 33: Mapovanie Oracle adresára na server

Keď sú vytvorené adresáre, je potrebné nastaviť práva na zápis a vykonanie v rámci týchto adresárov. Potom je možné spustiť import nasledujúcim príkazom (*def. príkazu 34*):

```
$ IMPDP login/heslo@connect_string
  DIRECTORY=Oracle_adresar
  DUMPFILE=nazov_export_suboru.dmp
```

Definícia príkazu 34: Spustenie metódy IMPDP

Metóda IMPDP obsahuje veľa ďalších parametrov, pomocou ktorých je možné spresniť samotný import. Medzi niektoré z nich patria parametre na definíciu tabuliek, schémy, generovanie log súboru a súboru s detailným popisom vzniknutých chýb. Tieto

parametre sú taktiež užitočné, pretože napomáhajú vyhnúť sa niektorým chybám pri zadávaní dlhých príkazov dátovej pumpy na príkazovom riadku [8].

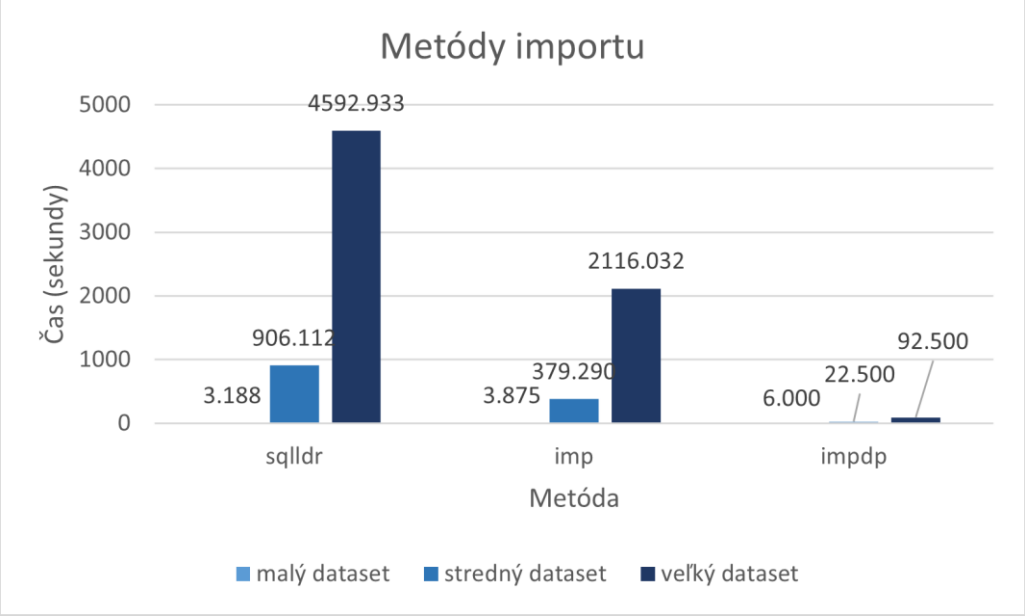
Experimenty

Nasledujúca kapitola popisuje výsledky výskumu, ktorý bol zameraný na porovnanie jednotlivých typov importu údajov do databázy. *Tabuľka 14* zobrazuje trvanie importu pre každú množinu dát. Čas importu v tabuľke je uvedený v sekundách, zaokrúhlený na 3 desatinné miesta.

Tabuľka 14: Trvanie metód importu (v sekundách)

	sqlldr	imp	impdp
Malý dátový set	3.188	3.875	6.000
Stredný dátový set	906.112	379.290	22.500
Veľký dátový set	4592.933	2116.032	92.500

Grafickú reprezentáciu hodnôt času trvania importu dát do databázy v sekundách zobrazuje *Obrázok 15*.



Obrázok 15: Čas trvania importu dát do databázy

Ako je znázornené na grafe, s malou množinou dát sú výsledky jednotlivých metód porovnateľné. Malé odchýlky môžu byť spôsobené aktuálnym zaťažením servera. Najväčší rozdiel v čase vykonávania importu sa prejavil na veľkej množine dát. Import trval pri metóde SQLLDR viac ako hodinu a 16 minút, pri metóde IMP trval okolo 35 minút a najrýchlejšie sa vykonal pri metóde IMPDP, kde trval minútu a 30 sekúnd.

Záver tohto experimentu priniesol výsledok, že najrýchlejšia metóda pre import dát do databázy je IMPDP, pretože sa vykonáva len na strane servera, na rozdiel od zvyšných dvoch metód, ktoré využívajú stranu klienta aj servera, a preto sú pri nich dodatočné náklady na prepojenie. Pri malom súbore dát čas vykonávania nie je až taký rozdielny, ale pri veľkých súboroch sú metódy IMP a SQLLDR oveľa pomalšie.

3.1.2 Metódy exportu dát

Metódy exportu dát z databázy sú metódy, ktoré umožňujú zálohovať, respektíve uložiť dáta z databázy do cieľového súboru. V rámci výskumu boli testované a porovnávané dva typy exportných metód:

- Metóda EXP
- Metóda EXPDP

Metóda EXP

Metóda EXP je metóda, ktorá umožňuje presun dát medzi databázovými systémami. Počas exportu sa generuje príkaz *SELECT*, výsledný súbor je v binárnej forme a je uložený do klientskeho zariadenia. Takto vytvorený exportný súbor môže byť prostredníctvom metódy IMP načítaný do databázy.

Funkcionalita EXP využíva klient-server architektúru. To znamená, že dump súbor je zapísaný do lokálneho adresára na klientskom zariadení, kde je spustený nástroj [17]. Ak dôjde k chybe v komunikácii medzi klientom a serverom, export skončí chybou. Ďalšou nevýhodou tejto metódy je rýchlosť exportu, ktorá je závislá od rýchlosti siete [20]. Nasledujúci príkaz slúži na spustenie metódy EXP (*def. príkazu 35*):

```
$ EXP login/heslo@connect_string
  FILE=nazov_export_suboru.exp
  TABLES=zoznam_tabuliek_na_export
```

Definícia príkazu 35: Spustenie metódy EXP

Názov exportného súboru reprezentuje súbor, ktorý sa vytvorí po úspešnom vykonaní metódy EXP. Parameter *TABLES* obsahuje zoznam tabuliek, ktoré sú exportované. V príkaze je možné použiť mnoho ďalších parametrov ako napríklad parameter *DIRECT*, ktorý slúži na extrakciu údajov priamym čítaním údajov. Nespracováva SQL príkazy[15] [9].

Metóda EXPDP

Tento typ exportu je známy ako export dátovej pumpy. Je to nástroj, ktorý sa považuje za rýchlejší typ exportu ako predchádzajúci typ exportu, pretože sa vykonáva len na strane servera. Takže nie sú potrebné dodatočné náklady na pripájanie medzi klientom a serverom [20]. Rovnako ako pri metóde IMPDP je potrebné vytvoriť mapovací objekt a Oracle adresár. Nasledujúci príkaz slúži na spustenie metódy EXPDP (*def. príkazu 36*):

```
$ EXPDP login/heslo@connect_string
  TABLES=zoznam_tabuliek
  DIRECTORY=Oracle_adresar
  DUMPFILE=nazov_export_suboru.dmp
```

Definícia príkazu 36: Spustenie metódy EXPDP

Názov exportného súboru reprezentuje dump súbor, ktorý sa vytvorí po úspešnom vykonaní metódy EXPDP. Parameter *TABLES* definuje zoznam tabuliek, ktoré majú byť exportované. Mnoho ďalších parametrov je dostupných v rámci metódy EXPDP. Medzi tieto parametre môžeme zaradiť, napríklad *LOGFILE*, ktorý slúži na vytvorenie logovacieho súboru. Tento typ súboru obsahuje informácie o vykonanom exporte. Iné typy parametrov môžu byť, napríklad *SCHEMAS, CONTENT, QUERY* a ďalšie.

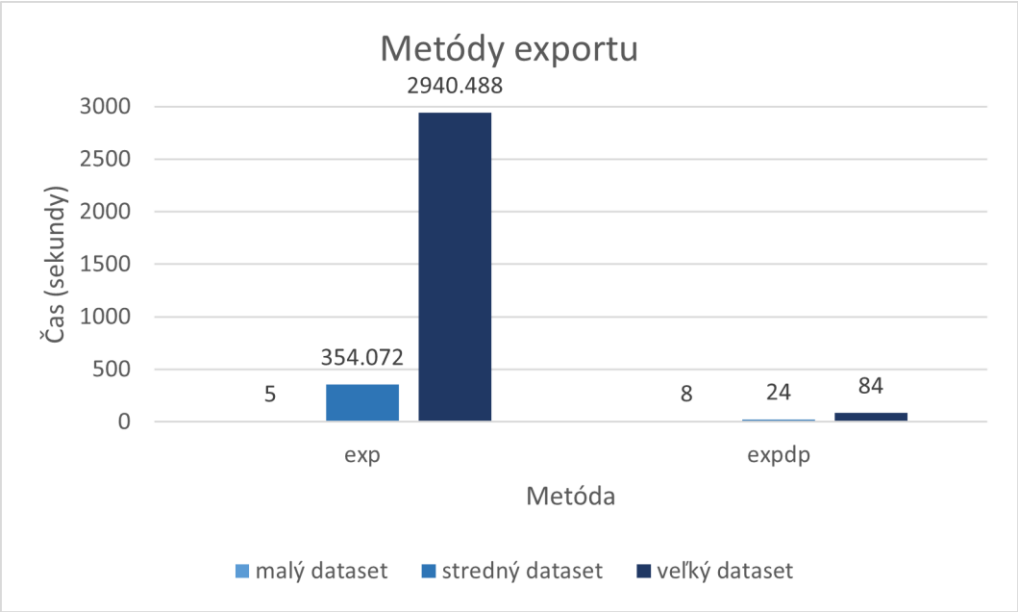
Experimenty

Nasledujúca kapitola popisuje výsledky výskumu, ktorý bol zameraný na porovnanie jednotlivých typov exportu údajov z databázy. *Tabuľka 15* zobrazuje trvanie exportu pre každý dátový set. Čas trvania exportu v tabuľke je uvedený v sekundách, zaokrúhlený na 3 desatinné miesta.

Tabuľka 15: Trvanie metód exportu (v sekundách)

	exp	expdp
Malý dátový set	5.266	7.889
Stredný dátový set	354.072	24.000
Veľký dátový set	2940.488	84.000

Grafickú reprezentáciu zobrazuje *Obrázok 16*. Hodnoty v grafe reprezentujú čas trvania exportu v sekundách.



Obrázok 16: Čas trvania exportu dát z databázy

Ako je znázornené v grafe, s malou množinou dát sú výsledky jednotlivých metód exportu porovnateľné, rovnako ako pri metódach importu. Rozdiely v trvaní exportu začínajú byť viditeľné až pri väčších množinách dát. Pri strednom dátovom sete metóda EXP trvá približne 6 minút, zatiaľ čo metóda EXPDP trvá 24 sekúnd. Najväčší rozdiel v čase vykonávania exportu sa prejavil na veľkej množine dát. Čas vykonávania metódy EXP trval približne 50 minút. Na druhej strane export prostredníctvom metódy EXPDP trval len 84 sekúnd.

Záver tohto experimentu priniesol výsledok, že najrýchlejšia metóda pre export dát do databázy je metóda EXPDP, pretože rovnako, ako pri metódach importu, aj táto metóda pracuje len na strane servera, čím sú eliminované dodatočné náklady na prepájanie klienta a servera.

3.2 Externé a interné tabuľky

Nasledujúci typ experimentu porovnáva rýchlosť vykonávania príkazu *SELECT* v interných a externých tabuľkách. Rozdiel medzi internými a externými tabuľkami je v tom, že externé tabuľky pristupujú k dátam, ktoré sú uložené v externom zdroji, to znamená, že dáta nie sú priamo uložené v databáze.

3.2.1 Externé tabuľky

Externé tabuľky sú dodatočným komponentom funkcie SQL*Loader. Umožňujú nám pristupovať k dátam v externých zdrojoch, ako keby boli údaje uložené v databázových

tabuľkách. V minulosti bolo možné z externých tabuľkách len čítať údaje, no v novších verziách databázového systému Oracle je možné do externých tabuliek aj zapisovať údaje [2].

```
CREATE TABLE NAZOV_TABULKY(
    stlpec1   datovy_typ,
    ....)
ORGANIZATION EXTERNAL(
    TYPE{ ORACLE_LOADER | ORACLE_DATAPUMP }
    DEFAULT DIRECTORY NAZOV_ADRESARA
    ACCESS PARAMETERS
      (RECORDS DELIMITED BY '\n'
       SKIP 1
       BADFILE 'NAZOV_BAD_SUBORU.bad'
       DISCARDFILE 'NAZOV_DISCARD_SUBORU.dsc'
       LOGFILE 'NAZOV_LOG_SUBORU.log'
       FIELDS TERMINATED BY ','
       OPTIONALLY ENCLOSED BY '"'
       ....
       MISSING FIELD VALUES ARE NULL(
        stlpec1           ,
        ....               ))
 LOCATION ('NAZOV_SUBORU')) REJECT LIMIT UNLIMITED;
```

Definícia príkazu 37: Syntax pre vytvorenie externej tabuľky

Def. príkazu 37 zobrazuje syntax pre vytvorenie externej tabuľky. Ako je možné vidieť, externé tabuľky sú vytvárané pomocou nasledujúceho SQL príkazu (*def. príkazu 38*)

```
CREATE TABLE
     ...
ORGANIZATION EXTERNAL.
```

Definícia príkazu 38: SQL príkaz vytvorenia externej tabuľky

Následne je dôležité definovať ďalšie atribúty. Jedným z atribútov je ORACLE_LOADER a ORACLE_DATAPUMP. Rozdiel medzi nimi je v tom, že ORACLE_LOADER je preddefinovaný a načítava údaje, ktoré prichádzajú z textových dátových súborov. Na druhej strane ORACLE_DATAPUMP načítava údaje, ktoré prichádzajú z binárnych dump súborov [12].

Pri vytvorení a pri práci s externými tabuľkami je najskôr potrebné vytvoriť objekt reprezentujúci fyzický adresár operačného systému, v ktorom je uložený dátový súbor. Tento súbor obsahuje údaje, ktoré sa budú načítavať. Ďalší krok je vytvorenie externej tabuľky. Následne je možné pracovať s obsahom súboru [2] [12].

Externé tabuľky majú určité obmedzenia, do ktorých môžeme zaradiť:

- nie je možné načítavať údaje do stĺpcov definovaných dátovým typom Long
- nie je možné popisovať údaje, ktoré sú uložené v databáze
- nie je možné meniť údaje pomocou SQL príkazu *UPDATE*

3.2.2 Experimenty

Nasledujúca kapitola popisuje výsledky výskumu, ktorý bol zameraný na porovnanie rýchlostí vykonania príkazu *SELECT* v interných a externých tabuľkách. Pre tento účel boli vytvorené tri externé tabuľky a tri interné tabuľky, pričom interné tabuľky obsahovali tie isté dáta ako externé tabuľky. V rámci experimentu sa porovnávali rýchlosti jednoduchého príkazu *SELECT*, ktorý vráti počet všetkých riadkov tabuľky. Príkaz *SELECT* vyzeral nasledovne (*def. príkazu 39*):

```
SELECT COUNT(*)
 FROM nazov_tabulky;
```

Definícia príkazu 39: Príkaz Select - vráti počet všetkých riadkov tabuľky

Tento príkaz bol nad každou tabuľkou vykonaný 10-krát. *Tabuľka 16* zobrazuje priemerný čas vykonania tohto príkazu nad každou tabuľkou. Čas v tabuľke je uvedený v sekundách.

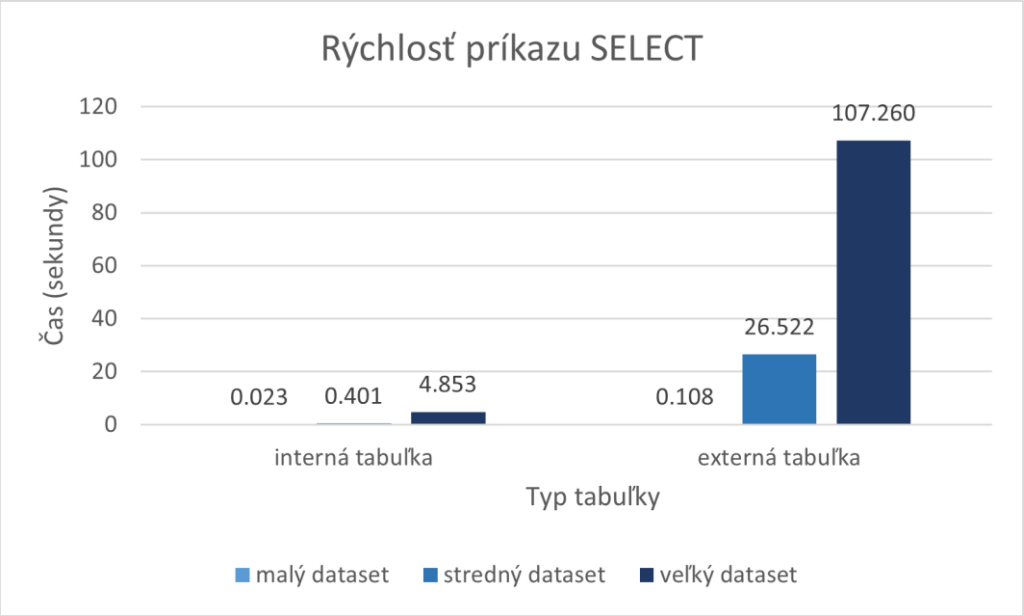
Tabuľka 16: Rýchlosti príkazu *SELECT* pre interné a externé tabuľky (v sekundách)

	Interná tabuľka	Externá tabuľka
Malý dátový set	0.023	0.108
Stredný dátový set	0.401	26.522
Veľký dátový set	4.853	107.260

Grafickú reprezentáciu zobrazuje *Obrázok 17*. Hodnoty v grafe reprezentujú čas trvania príkazu *SELECT* v sekundách.

Ako je znázornené na grafe, s malou množinou dát je trvanie príkazu *SELECT* v internej tabuľke v priemere 0.023 sekundy, zatiaľ čo v externej tabuľke je to 0.108 sekundy. Väčšie rozdiely sa vyskytnú pri strednej množine dát. Interná tabuľka je schopná vykonať príkaz *SELECT* v priemere za 0.401 sekundy, pričom v externej tabuľke to trvá

v priemere 26.522 sekúnd. Najväčší rozdiel nastal pri veľkej množine dát. Priemerný čas trvania príkazu *SELECT* v internej tabuľke je 4.853 sekúnd, zatiaľ čo v externej tabuľke to v priemere trvalo 1 minútu a 47 sekúnd.



Obrázok 17: Rýchlosť vykonania príkazu *SELECT*

Záver tohto experimentu priniesol výsledok, že vykonanie príkazu *SELECT* je v internej tabuľke rýchlejšie ako v externej tabuľke. Je to z toho dôvodu, že sú dáta uložené priamo v databázových tabuľkách. Mapovanie externých tabuliek je veľmi rýchle, ale musia byť prepojené k externému zdroju, ktorý je uložený na strane servera, a preto je čas vykonania príkazu *SELECT* dlhší.

3.3 Optimalizácia príkazu SELECT obsahujúceho analytické funkcie

S nárastom množstva dát uložených v databáze rastú aj náklady na vykonanie dotazov, obzvlášť ak využívajú funkcie na výpočet dodatočných informácií. Rýchlosť spracovania dotazu, ktorý obsahuje či už analytické alebo agregačné funkcie, je rozhodujúca, ak pracujeme s veľkým množstvom údajov v databáze.

Oracle používa vstavaný databázový softvér, nazývaný optimalizátor dotazov, ktorý slúži na určenie najefektívnejšieho plánu vykonania pre daný SQL príkaz. Optimalizátor zabezpečuje, že spomedzi všetkých plánov vykonávania príkazu vyberie ten, ktorý má najnižšie náklady na vykonanie daného príkazu [21], [22]. Plán vykonávania obsahuje jednotlivé kroky, ktoré databáza vykoná na získanie požadovaných údajov príkazu SELECT. Okrem jednotlivých krokov zobrazuje aj odporúčaný spôsob prístupovej metódy a celkové

náklady na vykonanie dotazu. ***Celkové náklady*** predstavujú internú hodnotu, ktorá slúži na porovnanie jednotlivých plánov [21], [22].

Existuje niekoľko spôsobov ako optimalizovať SQL príkazy. V rámci optimalizácie príkazu SELECT sme sa zamerali na porovnanie piatich rôznych metód vytvorenia dotazu spolu s vytvorením vhodných indexov, ktoré môžu zlepšiť plán vykonania daného dotazu. Prvou metódou je klasický príkaz *SELECT* obsahujúci priamo analytickú funkciu LAG(), ktorá vracia hodnotu definovaného stĺpca s definovaným posunom pred aktuálnym riadkom. Ďalej nasledujú tri rôzne používateľom definované funkcie a poslednou porovnávanou metódou je použitie materializovaného pohľadu. Všetky porovnávané metódy vrátia rovnaký výsledok, len iným spôsobom. Týmto výskumom sa bližšie zaoberá publikácia [HD7].

Na demonštrovanie experimentov sme využili dáta z leteckej dopravy, resp. letové údaje, v ktorých sme získali hodnoty súradníc z predchádzajúceho záznamu, ktoré sme následne použili na výpočet vzdialenosti medzi dvoma bodmi. Všetky experimenty boli testované na datasete, ktorý obsahoval 3 360 932 záznamov.

3.3.1 Príkaz SELECT obsahujúci analytickú funkciu LAG()

Prvý experiment, ktorý sme vykonali, pozostával z vytvorenia klasického príkazu *SELECT*, ktorý obsahoval priamo analytickú funkciu LAG() vo svojej definícii. Pomocou tejto funkcie sme získali zemepisnú šírku a dĺžku predchádzajúceho záznamu. Príkaz *SELECT* vyzeral nasledovne (*def. príkazu 40*):

```
SELECT ectrl_id, sequence_number, latitude, longitude,
       LAG(latitude, 1) OVER
          (PARTITION BY ectrl_id
           ORDER BY ectrl_id, sequence_number) AS pr_lat,
       LAG(longitude, 1) OVER
          (PARTITION BY ectrl_id
           ORDER BY ectrl_id, sequence_number) AS pr_lon
 FROM flight_tab_1512
 WHERE ectrl_id BETWEEN 192254169 AND 192340127;
```

Definícia príkazu 40: Príkaz *SELECT* obsahujúci priamo analytickú funkciu

Celkové náklady bez vytvorenia indexu

Najskôr sme vykonali príkaz bez vytvorenia akýchkoľvek indexov. Poradie krokov, v akom bol vykonaný príkaz *SELECT*, je znázornené na *Obrázku 18*.

```
-------------------------------------------------------------------------------------------
| Id | Operation              | Name                | Rows    | Cost   (%CPU)| Time        |
-------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT       |                     | 1964817 | 18921     (1)| 00:00:01    |
|  1 |  WINDOW (SORT)         |                     | 1964817 | 18921     (1)| 00:00:01    |
|* 2 |   TABLE ACCESS FULL    |   FLIGHT_TAB_1512   | 1964817 |  5838     (1)| 00:00:01    |
-------------------------------------------------------------------------------------------
```

Obrázok 18: Plán vykonania príkazu *SELECT* obsahujúceho priamo analytickú funkciu LAG() bez použitia indexu

V prvom rade, sa ako prvá vykonala podmienka *WHERE*. Vďaka nej sa vybrali požadované záznamy z definovaného rozsahu. Na vybratie týchto záznamov bolo potrebné prehľadať celú tabuľku *FLIGHT_TAB_1512*, preto celkové náklady na vykonanie tohto príkazu boli 18 921.

Celkové náklady s využitím indexu

Ďalší krok, ktorý sme spravili bolo vytvorenie indexu nad týmto príkazom *SELECT*. Vytvorili sme index, ktorý obsahoval všetky atribúty okrem analytickej funkcie, pretože tieto funkcie nie je možné priamo indexovať. *Def. príkazu 41* znázorňuje kód vytvoreného indexu.

```
CREATE INDEX ind_1
 ON flight_tab_1512
  ( ectrl_id, sequence_number, latitude, longitude );
```

Definícia príkazu 41: Index pre príkaz *SELECT* obsahujúci priamo analytickú funkciu

Po vykonaní príkazu sme získali plán vykonania zobrazený na *Obrázku 19*.

```
-------------------------------------------------------------------------------------------
| Id | Operation              | Name                | Rows    | Cost   (%CPU)| Time        |
-------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT       |                     | 1964817 | 8978      (1)| 00:00:01    |
|  1 |  WINDOW (BUFFER)       |                     | 1964817 | 8978      (1)| 00:00:01    |
|* 2 |   INDEX (RANGE SCAN)   |   IND_1             | 1964817 | 8978      (1)| 00:00:01    |
-------------------------------------------------------------------------------------------
```

Obrázok 19: Plán vykonania pre príkaz SELECT obsahujúci priamo funkciu LAG() spolu s vytvoreným indexom

Najskôr sa vykonala podmienka, podobne ako v prvom prípade, no rozdiel nastal v prístupovej metóde, ktorá bola použitá. Namiesto prehľadávania celej tabuľky optimalizátor použil prístupovú metódu *INDEX RANGE SCAN*, vďaka ktorej sa využil nami vytvorený index (IND_1). Preto sa celkové náklady na vykonanie tohto príkazu *SELECT* znížili na hodnotu 8 978.

3.3.2 Príkaz SELECT obsahujúci používateľom definované funkcie GET_PREDECESSOR_LAT() a GET_PREDECESSOR_LON()

Ďalší experiment pozostával z vytvorenia dvoch používateľom definovaných funkcií, ktoré nahradili analytickú funkciu, ktorá bola priamo v príkaze *SELECT*.

Prvá funkcia sa nazýva GET_PREDECESSOR_LAT() a slúži na vrátenie súradnice zemepisnej šírky z predchádzajúceho záznamu. Vstupnými parametrami tejto funkcie sú identifikačné číslo letu (ectrl_id) a poradie zaznamenaného záznamu (sequence_number). Kód tejto funkcie je možné vidieť v *def. príkazu 42*.

```
CREATE OR REPLACE FUNCTION get_predecessor_lat(
     p_ectrl_id           NUMBER,
     p_sequence_number    NUMBER)
RETURN NUMBER
DETERMINISTIC
IS
  res NUMBER;
BEGIN
  SELECT lat INTO res
  FROM (
       SELECT sequence_number AS sequence_n,
           LAG(latitude, 1) OVER
              (ORDER BY ectrl_id, sequence_number) AS lat
       FROM flight_tab_1512
       WHERE ectrl_id = p_ectrl_id)
  WHERE sequence_n = p_sequence_number;
 RETURN res;
 EXCEPTION WHEN OTHERS THEN RETURN -1;
END;
/
```

Definícia príkazu 42: Funkcia GET_PREDECESSOR_LAT()

Druhá funkcia sa nazýva GET_PREDECESSOR_LON(). Funkcia je založená na rovnakom princípe ako vyššie uvedená funkcia, avšak výsledkom tejto funkcie je súradnica zemepisnej dĺžky z predchádzajúceho záznamu. Vytvorená funkcia je znázornená v *def. príkazu 43*.

```
CREATE OR REPLECE FUNCTION get_predecessor_lon(
     p_ectrl_id            NUMBER,
     p_sequence_number     NUMBER)
RETURN NUMBER
DETERMINISTIC
IS
  res NUMBER;
BEGIN
  SELECT lon INTO res
  FROM (
       SELECT sequence_number AS sequence_n,
              LAG(longitude, 1) OVER
                  (ORDER BY ectrl_id, sequence_number) AS lon
       FROM flight_tab_1512
       WHERE ectrl_id = p_ectrl_id)
  WHERE sequence_n = p_sequence_number;
 RETURN res;
 EXCEPTION WHEN OTHERS THEN RETURN -1;
END;
/
```

Definícia príkazu 43: Funkcia GET_PREDECESSOR_LON()

Akonáhle sme vytvorili funkcie, upravili sme príkaz *SELECT* tak, že sme nahradili analytickú funkciu, nami definovanými funkciami. Príkaz *SELECT* vyzeral nasledovne (*def. príkazu 44*):

```
SELECT ectrl_id, sequence_number, latitude, longitude,
     get_predecessor_lat(ectrl_id, sequence_number) AS pr_lat,
     get_predecessor_lon(ectrl_id, sequence_number) AS pr_lon
 FROM flight_tab_1512
 WHERE ectrl_id BETWEEN 192254169 AND 192340127;
```

Definícia príkazu 44: Príkaz *SELECT* s funkciami GET_PREDECESSOR_LAT() a GET_PREDECESSOR_LON()

Porovnanie tohto príkazu *SELECT* bez vytvoreného indexu nie je zaujímavé z hľadiska výkonnosti, pretože náklady na jeho vykonanie sú príliš vysoké, takže sme rovno vytvorili index, ktorý okrem identifikačného čísla letu (ectrl_id), poradového čísla letu (sequence_number), zemepisnej šírky a dĺžky obsahoval aj používateľom definované

funkcie GET_PREDECESSOR_LAT() a GET_PREDECESSOR_LON(). Vytvorený index je možné vidieť v *def. príkazu 45*.

```
CREATE INDEX ind_4
 ON flight_tab_1512
 ( ectrl_id, sequence_number, latitude, longitude,
   get_predecessor_lat(ectrl_id, sequence_number),
   get_predecessor_lon(ectrl_id, sequence_number));
```

Definícia príkazu 45: Index s funkciami GET_PREDECESSOR_LAT() a GET_PREDECESSOR_LON()

Po vykonaní príkazu *SELECT* a získaní plánu vykonania sme zistili, že celkové náklady na vykonanie tohto príkazu sú 12 092. Postupnosť krokov vykonania príkazu *SELECT* je zobrazená na *Obrázku 20*, pričom je vidieť, že optimalizátor použil prístupovú metódu *INDEX RANGE SCAN* a využil nami vytvorený index (IND_4).

```
-------------------------------------------------------------------------------------------
| Id | Operation              | Name               | Rows    | Cost   (%CPU)| Time        |
-------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT       |                    | 1964817 | 12092     (1)| 00:00:01    |
|  1 |   INDEX (RANGE SCAN)   |   IND_4            | 1964817 | 12092     (1)| 00:00:01    |
-------------------------------------------------------------------------------------------
```

Obrázok 20: Plán vykonania príkazu *SELECT* s indexom obsahujúci funkcie GET_PREDECESSOR_LAT() a GET_PREDECESSOR_LON()

3.3.3 Príkaz SELECT s používateľom definovanou funkciou GET_PREDECESSOR_LAT_LON()

Cieľom nasledujúceho experimentu bolo vytvoriť funkciu, ktorá vráti reťazec znakov, oddelený bodkočiarkou, obsahujúci údaje o zemepisnej šírke a dĺžke z predchádzajúceho záznamu. Vytvorená funkcia je znázornená pomocou *def. príkazu 46*. Funkcia má dva vstupné parametre a to identifikačné číslo letu (ectrl_id) a poradové číslo letu (sequence_number). Návratová hodnota je maximálne 30 znakový reťazec. Formát návratovej hodnoty je:

Návratová hodnota = latitude;longitude

```
CREATE OR REPLACE FUNCTION get_predecessor_lat_lon(
     p_ectrl_id            NUMBER,
     p_sequence_number     NUMBER)
RETURN VARCHAR2
DETERMINISTIC
IS
  res VARCHAR2(30);
BEGIN
  SELECT lat || ';' || lon INTO res
  FROM (
       SELECT sequence_number AS sequence_n,
              LAG(latitude, 1) OVER
                  (ORDER BY ectrl_id, sequence_number) AS lat,
              LAG(longitude, 1) OVER
                  (ORDER BY ectrl_id, sequence_number) AS lon
       FROM flight_tab_1512
       WHERE ectrl_id = p_ectrl_id)
  WHERE sequence_n = p_sequence_number;
 RETURN res;
 EXCEPTION WHEN OTHERS THEN RETURN -1;
END;
/
```

Definícia príkazu 46: Funkcia GET_PREDECESSOR_LAT_LON()

Po vytvorení funkcie sme modifikovali príkaz *SELECT* tak, že výsledok z funkcie sme rozparsovali podľa definovaného oddeľovača (bodkočiarky) a získali sme tak súradnice zemepisnej šírky a dĺžky. Na parsovanie súradníc sme využili funkciu SUBSTR(). Okrem toho sme použili aj funkciu INSTR(), ktorá slúži na vyhľadanie určitého podreťazca alebo znaku v reťazci a jej návratová hodnota je pozícia tohto znaku (celočíselná hodnota). *Def. príkazu 47* znázorňuje príkaz *SELECT* s využitím funkcie GET_PREDECESSOR_LAT_LON()

```
SELECT ectrl_id, sequence_number, latitude, longitude,
  SUBSTR(
   get_predecessor_lat_lon(ectrl_id, sequence_number), 1,
   (INSTR(get_predecessor_lat_lon(ectrl_id, sequence_number), ';')
     -1)) AS lat,
  SUBSTR(
   get_predecessor_lat_lon(ectrl_id, sequence_number),
   (INSTR(get_predecessor_lat_lon(ectrl_id, sequence_number), ';')
     +1),
     LENGTH(get_predecessor_lat_lon(ectrl_id, sequence_number)))
          AS lon
 FROM flight_tab_1512
 WHERE ectrl_id BETWEEN 192254169 AND 192340127;
```

Definícia príkazu 47: Príkaz *SELECT* s funkciou GET_PREDECESSOR_LAT_LON()

Následne sme vytvorili index, ktorého definícia je nasledovná (*def. príkazu 48*).

```
CREATE INDEX ind_6
 ON flight_tab_1512
  ( ectrl_id, sequence_number, latitude, longitude,
    get_predecessor_lat_lon(ectrl_id, sequence_number));
```

Definícia príkazu 48: Index s funkciou GET_PREDECESSOR_LAT_LON()

Po spustení príkazu sme získali plán vykonania, ktorý je zobrazený na *Obrázku 21*. Ako je možné vidieť, celkové náklady na vykonanie tohto príkazu *SELECT* s definovaným indexom (IND_6) sú 13 477, pričom sa využila prístupová metóda *INDEX RANGE SCAN*.

```
-------------------------------------------------------------------------------------------
| Id | Operation              | Name                | Rows    | Cost   (%CPU)| Time        |
-------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT       |                     | 1964817 | 13477     (1)| 00:00:01    |
|  1 |   INDEX (RANGE SCAN)   |   IND_6             | 1964817 | 13477     (1)| 00:00:01    |
-------------------------------------------------------------------------------------------
```

Obrázok 21: Plán vykonania príkazu *SELECT* s indexom obsahujúci funkciu GET_PREDECESSOR_LAT_LON()

3.3.4 Príkaz SELECT s používateľom definovanou funkciou GET_PREDECESSOR_LAT_LON_REC()

Ďalší spôsob, ktorým sme sa snažili zlepšiť náklady na vykonanie príkazu *SELECT* obsahujúci analytické funkcie, bolo vytvorenie funkcie, ktorá vráti objektový typ *RECORD,* na základe vstupných parametrov. Najskôr sme museli vytvoriť nový typ, ktorý obsahoval dva atribúty (zemepisnú šírku a dĺžku). Vytvorený objektový typ zobrazuje *def. príkazu 49*.

```
CREATE OR REPLACE TYPE coordinates_obj_type IS OBJECT(
  lat_coord NUMBER(9,6),
  lon_coord NUMBER(9,6))
/
```

Definícia príkazu 49: Objektový typ COORDINATES_OBJ_TYPE

Následne sme vytvorili funkciu GET_PREDECESSOR_LAT_LON_REC(), ktorej návratová hodnota je jeden záznam tohto typu. Funkcia má obdobne ako predchádzajúce funkcie dva vstupné parametre, identifikačné číslo letu (ectrl_id) a poradové číslo letu (sequence_number). Na základe týchto parametrov funkcia vyberie súradnice zemepisnej šírky a dĺžky. Vytvorenú funkciu zobrazuje *def. príkazu 50*.

```
CREATE OR REPLACE FUNCTION get_predecessor_lat_lon_rec(
     p_ectrl_id           NUMBER,
     p_sequence_number    NUMBER)
RETURN COORDINATES_OBJ_TYPE
DETERMINISTIC
IS
   res COORDINATES_OBJ_TYPE := COORDINATES_OBJ_TYPE(NULL, NULL);
 BEGIN
   SELECT coord.lat, coord.lon INTO res.lat_coord, res.lon_coord
   FROM(
        SELECT sequence_number AS sequence_n,
             LAG(latitude, 1) OVER
                 (ORDER BY ectrl_id, sequence_number) AS lat,
             LAG(longitude, 1) OVER
                 (ORDER BY ectrl_id, sequence_number) AS lon
         FROM flight_tab_1512
         WHERE ectrl_id = p_ectrl_id) coord
   WHERE sequence_n = p_sequence_number;
 RETURN res;
 EXCEPTION WHEN OTHERS THEN RETURN NULL;
END;
/
```

Definícia príkazu 50: Funkcia GET_PREDECESSOR_LAT_LON_REC()

Po vytvorení tejto funkcie sme upravili príkaz *SELECT* nasledovne (*def. príkazu 51*):

```
SELECT ectrl_id, sequence_number, latitude, longitude,
    get_predecessor_lat_lon_rec(ectrl_id, sequence_number).lat_coord
          AS pr_lat,
    get_predecessor_lat_lon_rec(ectrl_id, sequence_number).lon_coord
          AS pr_lon
 FROM flight_tab_1512
 WHERE ectrl_id BETWEEN 192254169 AND 192340127;
```

Definícia príkazu 51: Príkaz *SELECT* s funkciou GET_PREDECESSOR_LAT_LON_REC()

Následne sme vytvorili funkcionálny index, ktorý okrem identifikačného čísla letu (ectrl_id), poradového čísla letu (sequence_number) a súradníc latitude a longitude obsahoval aj túto funkciu. *Def. príkazu 52* zobrazuje vytvorený index.

```
CREATE INDEX ind_7
 ON flight_tab_1512
 ( ectrl_id, sequence_number, latitude, longitude,
   get_predecessor_lat_lon_rec(ectrl_id, sequence_number));
```

Definícia príkazu 52: Index s funkciou GET_PREDECESSOR_LAT_LON_REC()

Po vytvorení indexu systém vyvolal výnimku, ktorá informovala o tom, že nemôže byť vytvorený index nad objektovým typom *RECORD*. Takže sme vykonali príkaz *SELECT* bez vytvorenia tohto indexu. Keď sme si zobrazili plán vykonania, bolo zrejmé, že optimalizátor využil prístupovú metódu *FAST FULL SCAN*, pričom použil index (IND_1). To znamená, že každopádne sa museli prechádzať všetky záznamy. Celkové náklady na vykonanie tohto príkazu *SELECT* boli preto oveľa vyššie ako v predchádzajúcich prípadoch. Výsledný plán vykonania je zobrazený na *Obrázku 22*.

Plán vykonania je doplnený o náklady na vykonanie vytvorenej funkcie. Náklady na získanie jedného záznamu z funkcie sú 3. Následne sa táto hodnota vynásobila počtom záznamov, ktoré sa spracovali, čím sme získali náklady na vykonanie funkcie pre jednu súradnicu. Rovnaký postup sa zopakoval aj pre druhú súradnicu a pridali sa náklady na vykonanie príkazu *SELECT*. Preto celkové náklady stúpli na hodnotu 11 793 090.

```
-------------------------------------------------------------------------------------------
|Id | Operation                              | Name    | Rows    | Cost    (%CPU)| Time      |
-------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT                       |         | 1964817 | 11793090  (1)| 00:00:01  |
| 1 |  INDEX (RANGE SCAN)                    |  IND_1  | 1964817 |     4188  (1)| 00:00:01  |
| 2 |  GET_PREDECESSOR_LAT_LON_REC.LAT_COORD |         | 1964817 |  5894451  (1)| 00:00:01  |
| 3 |  GET_PREDECESSOR_LAT_LON_REC.LAT_COORD |         | 1964817 |  5894451  (1)| 00:00:01  |
-------------------------------------------------------------------------------------------
```

Obrázok 22: Plán vykonania príkazu *SELECT* s funkciou GET_PREDECESSOR_LAT_LON_REC() a indexom IND_1

3.3.5 Materializovaný pohľad

Posledný experiment, ktorý sme vykonali bol zameraný na materializované pohľady. Je to ďalší spôsob, ktorým môžeme efektívne pracovať s analytickými funkciami. Cieľom tohto experimentu bolo zistiť, či sa s vytvorením materializovaného pohľadu dokážu znížiť náklady na vykonanie príkazu *SELECT* obsahujúceho analytické funkcie.

V prvom kroku sme vytvorili materializovaný pohľad COORDINATES_VIEW_ALL, ktorý obsahoval všetky atribúty z pôvodnej tabuľky *FLIGHT_TAB_1512*. Pohľad sme doplnili o ďalšie dva atribúty, ktoré predstavujú súradnice zemepisnej šírky a dĺžky z predchádzajúceho záznamu, pomocou funkcie LAG(). *Def. príkazu 53* znázorňuje vytvorený materializovaný pohľad.

```
CREATE MATERIALIZED VIEW coordinates_view_all
 AS
 SELECT ectrl_id, sequence_number, time_over, flight_level,
        latitude, longitude, fir_id, distance_nm,
         LAG(latitude, 1) OVER (ORDER BY ectrl_id, sequence_number)
           AS lat,
         LAG(longitude, 1) OVER (ORDER BY ectrl_id, sequence_number)
           AS lon
 FROM flight_tab_1512;
```

Definícia príkazu 53: Materializovaný pohľad COORDINATES_VIEW_ALL

Po vytvorení pohľadu sme vytvorili príkaz *SELECT*, v ktorom sme vyberali tie isté údaje ako v predchádzajúcich experimentoch, avšak ako zdrojovú tabuľku sme použili vytvorený materializovaný pohľad. Príkaz *SELECT* je znázornený v *def. príkazu 54*.

```
SELECT ectrl_id, sequence_number, latitude, longitude, lat, lon
 FROM coordinates_view_all
 WHERE ectrl_id BETWEEN 192254169 AND 192340127;
```

Definícia príkazu 54: Príkaz *SELECT* s materializovaným pohľadom

Celkové náklady bez vytvorenia indexu

Po spustení tohto príkazu sme získali plán vykonania, ktorý je znázornený na *Obrázku 23*. Ako je možné vidieť, celkové náklady na vykonanie tohto príkazu sa znížili na hodnotu 7 479.

```
-------------------------------------------------------------------------------------------------
|Id | Operation               | Name                    | Rows    | Cost   (%CPU)| Time      |
-------------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT        |                         | 1964817 | 7479      (1)| 00:00:01  |
| 1 |  MAT_VIEW Access (Full) |  COORDINATES_VIEW_ALL   | 1964817 | 7479      (1)| 00:00:01  |
-------------------------------------------------------------------------------------------------
```

Obrázok 23: Plán vykonania materializovaného pohľadu bez indexu

Celkové náklady s vytvoreným indexom

Následne sme vytvorili index nad týmto pohľadom, ktorý by mohol znížiť náklady na vykonanie príkazu *SELECT*. Vytvorený index je znázornený v *def. príkazu 55*.

```
CREATE INDEX ind_8
 ON coordinates_view_all
 ( ectrl_id, sequence_number, latitude, longitude, lat, lon );
```

Definícia príkazu 55: Index pre materializovaný pohľad

Keď sme opäť vykonali príkaz *SELECT* a zobrazili plán vykonania, celkové náklady na jeho vykonanie sa znížili na hodnotu 5 682, keďže optimalizátor využil nami vytvorený index (IND_8). Plán vykonania je znázornený na *Obrázku 24*.

```
-------------------------------------------------------------------------------------------------
|Id | Operation               | Name                    | Rows    | Cost   (%CPU)| Time      |
-------------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT        |                         | 1964817 | 5682      (1)| 00:00:01  |
| 1 |  INDEX (FAST FULL SCAN) |  IND_8                  | 1964817 | 5682      (1)| 00:00:01  |
-------------------------------------------------------------------------------------------------
```

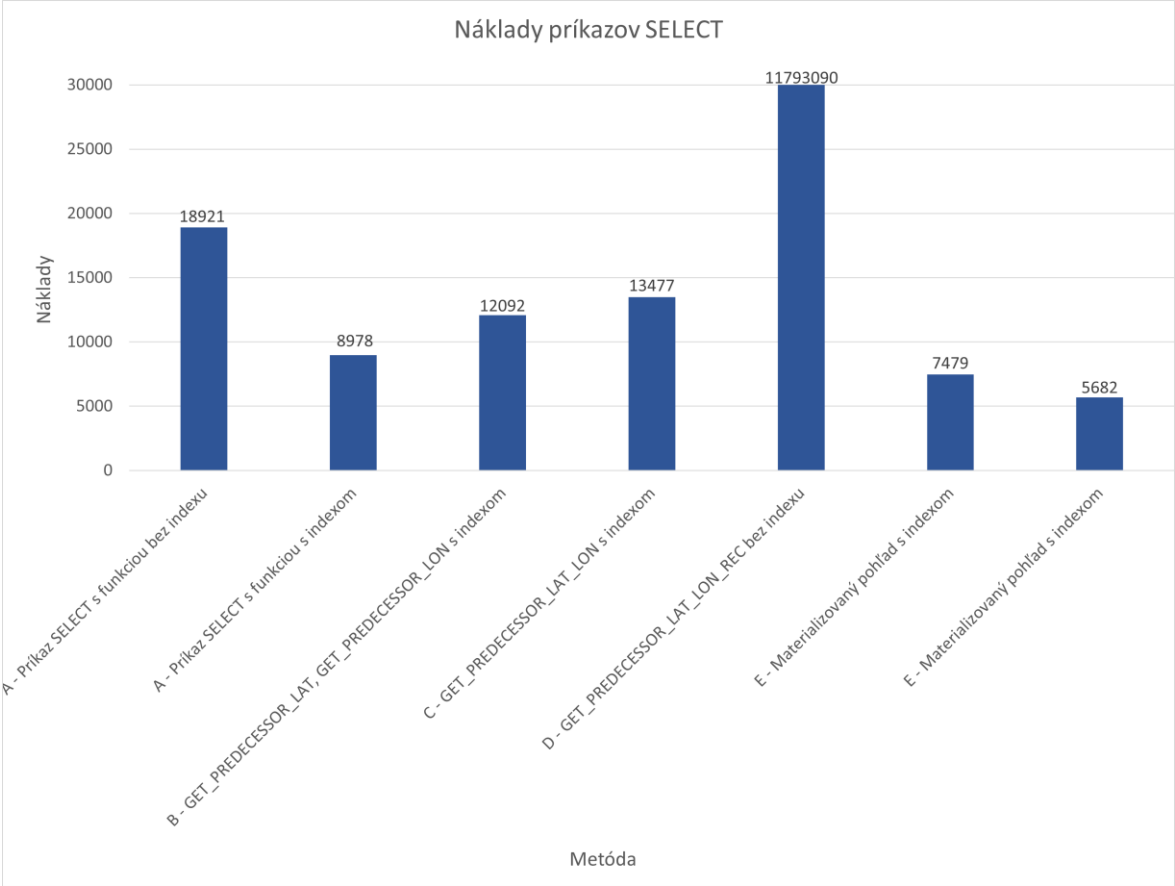
Obrázok 24: Plán vykonania pre materializovaný pohľad s indexom

3.3.6 Zhrnutie experimentov

Cieľom týchto experimentov bolo porovnať a optimalizovať náklady pre rôzne metódy získania tých istých údajov pre príkaz *SELECT* s využitím analytických funkcií. *Obrázok 25* zobrazuje stĺpcový graf, ktorý obsahuje celkové náklady porovnávaných metód.

Ako je možné vidieť na stĺpcovom grafe, najvyššie náklady na vykonanie príkazu *SELECT* dosiahla používateľom definovaná funkcia GET_PREDECESSOR_LAT_LON_REC(), pretože nebolo možné nad ňou vytvoriť vhodný index, keďže návratová hodnota tejto funkcie je typu *RECORD*. V poradí druhou metódou s najvyššími celkovými nákladmi na vykonanie bol klasický príkaz *SELECT* obsahujúci priamo analytickú funkciu LAG() bez využitia indexu. Pri využití používateľom definovaných funkcií GET_PREDECESSOR_LAT(), GET_PREDECESSOR_LON()

a GET_PREDECESSOR_LAT_LON(), sú celkové náklady na vykonanie porovnateľné. Najlepšie výsledky dosiahol materializovaný pohľad s vytvoreným indexom, ktorý optimalizoval daný príkaz a znížil celkové náklady na jeho vykonanie na hodnotu 5 682.



Obrázok 25: Stĺpcový graf s celkovými nákladmi porovnávaných metód príkazu SELECT

3.4 Návrh dátového skladu

V rámci analytického spracovania dát je potrebné získané údaje určitým spôsobom uchovávať v databáze. Dátový sklad je vhodnou metódou ako tieto údaje ukladať. My sme sa v rámci nášho výskumu zamerali na vytvorenie dátového skladu pre policajné dáta, ktoré následne využívame na ďalší výskum zameraný na optimalizáciu dátovej štruktúry a optimalizáciu vykonávania dotazov v dátových skladoch.

Policajné dáta z dopravných nehôd, ktoré sa zaznamenávajú, majú jednotnú štruktúru. V rámci daného roku sú tieto dáta zbierané mesačne pre jednotlivé kraje Českej Republiky. Okrem samotných nehôd sa zaznamenávajú aj informácie o chodcoch, ktorí boli súčasťou dopravnej nehody. Datasety o nehodách, predstavujú tabuľku faktov a jednotlivé dimenzie reprezentujú číselníky, ktoré sú taktiež k dispozícii.

3.4.1 Dátový sklad nehôd

Dátový sklad nehôd pozostáva z jednej tabuľky faktov a niekoľkých dimenzii. V nasledujúcom texte si bližšie popíšeme význam jednotlivých tabuliek. Grafická reprezentácia dátového skladu sa nachádza v *Prílohe A* na konci tejto dizertačnej práce.

Dimenzie dátového skladu nehôd

- Lokalita nehody – určuje, či sa nehoda stala v alebo mimo obce
  - označenie *p5a_lokalita_nehody*
- Druh nehody – určuje, či išlo o zrážku s nejakým objektom, haváriu, prípadne iný druh nehody
  - označenie *p6_druh_nehody*
- Druh zrážky idúcich vozidiel – určuje o aký druh zrážky išlo (čelný náraz, bočný náraz, ...)
  - označenie *p7_druh_zrazky_vozidiel*
- Druh pevnej prekážky – popisuje prekážky na cestách a mimo nich (strom, stĺp, zvodidlo, ...)
  - označenie *p8_druh_pevnej_prekazky*
- Charakter nehody – popisuje či išlo o nehodu s hmotnou škodou
  - označenie *p9_charakter_nehody*
- Zavinenie nehody – určuje, kto spôsobil nehodu (chodec, vodič, zver, ...)
  - označenie *p10_zavinenie_nehody*
- Alkohol u vinníka nehody – určuje, či bol vinník pod vplyvom alkoholu
  - označenie *p11_alkohol*
- Hlavná príčina nehody – popisuje, čo bolo hlavnou príčinou nehody (neprimeraná rýchlosť jazdy, predbiehanie, ...)
  - označenie *p12_hlavna_pricina_nehody*
- Druh povrchu vozovky – popisuje typ vozovky (dlažba, betón, asfalt, ...)
  - označenie *p15_druh_povrchu_vozovky*
- Stav povrchu vozovky počas nehody – určuje, v akom stave bol povrch vozovky (suchý, mokrý, zasnežený, ...)
  - označenie *p16_stav_povrchu_vozovky*
- Stav cestnej komunikácie – popisuje stav cestnej komunikácie (bez poškodenia, výtlky, zúženie vozovky, ...)

    - označenie *p17_stav_cestnej_komunikacie*
- Poveternostné podmienky v čase nehody – popisujú či bola v čase nehody hmla, dážď, sneh, ...
    - označenie *p18_poveternostne_podmienky*
- Viditeľnosť – popisuje, aká viditeľnosť bola v čase nehody (zhoršená, s/bez verejného osvetlenia, ...)
    - označenie *p19_viditelnost*
- Výhľad počas nehody – popisuje, či pred vozidlom išlo iné vozidlo, či sa pri komunikácii nachádzali budovy, ...
    - označenie *p20_rozhledove_pomery*
- Delenie cestnej komunikácie – určuje, či ide o dvoj-, trojpruhovú cestnú komunikáciu, ...
    - označenie *p21_delenie_cestnej_komunikacie*
- Umiestnenie nehody na cestnej komunikácii – určuje, kde sa stala nehoda (na krajnici, na odbočovacom pruhu, ...)
    - označenie *p22_situovanie_nehody_na_komunikacii*
- Riadenie premávky v čase nehody – popisuje, či bola premávka v čase nehody riadená policajtom, svetelným označením, ...
    - označenie *p23_riadenie_premavky*
- Dočasná úprava prednosti v jazde – určuje, či bola dočasne zmenená prednosť v jazde
    - označenie *p24_miestna_uprava_prednosti_jazdy*
- Špecifické miesta a objekty v mieste nehody – popisuje, či boli v okolí miesta nehody nejaké objekty (prechod pre chodcov, most, nadjazd, ...)
    - označenie *p27_miesta_objekty*
- Pomer smeru cestnej komunikácie – určuje, či sa nehoda udiala na križovatke, kruhovom objazde, zákrute, ...
    - označenie *p28_smerove_pomery*
- Miesto dopravnej nehody – popisuje miesto dopravnej nehody (mimo križovatku, na križovatke, ...)
    - označenie *p35_miesto_dopravnej_nehody*
- Druh pozemnej komunikácie – popisuje, či sa nehoda stala na diaľnici, ceste 1. triedy, ...

    - označenie *p36_druh_pozemnej_komunikacie*
- Druh križujúci cestnú komunikáciu – popisuje či komunikáciu križovala iná cestná komunikácia (1. triedy, 2. triedy, ...)
    - označenie *p39_druh_krizujuci_komunikaciu*
- Druh vozidla – popisuje aké vozidlo bolo súčasťou dopravnej nehody (osobný automobil, trolejbus, vlak, ...)
    - označenie *p44_druh_vozidla*
- Výrobná značka motorového vozidla – popisuje značku vozidla (Audi, BMW, Jaguar, ...)
    - označenie *p45a_vyrobna_znacka_vozidla*
- Charakteristika vozidla – popisuje či je vozidlo súkromné, či je súčasťou hromadnej dopravy, štátne, ...
    - označenie *p48a_charakteristika_vozidla*
- Šmyk – určuje, či dopravnú nehodu spôsobil šmyk
    - označenie *p49_smyk*
- Vozidlo po nehode – popisuje či počas nehody došlo k požiaru alebo nie, či vodič utiekol, ...
    - označenie *p50a_vozidlo_po_nehode*
- Únik prevážaných hmôt – popisuje, či počas nehody došlo k úniku prevážaných hmôt
    - označenie *p50b_unik_prepravovanych_hmot*
- Spôsob vyslobodenia osôb z vozidla – popisuje či bolo potrebné vyslobodiť osoby z vozidla
    - označenie *p51_sposob_vyslobodenia_osob_z_vozidla*
- Smer jazdy alebo postavenie vozidla – popisuje smer jazdy prípadne postavenie vozidla počas nehody (idúce vozidlo, odstavené vozidlo, ...)
    - označenie *p52_smer_jazdy_postavenie_vozidla*
- Kategória vodiča – popisuje, aké vodičské oprávnenie má vodič dopravnej nehody
    - označenie *p55a_kategoria_vodica*
- Stav vodiča – popisuje, v akom stave bol vodič počas dopravnej nehody (unavený, pod vplyvom alkoholu, chorý, ...)
    - označenie *p57_stav_vodica*

- Vonkajšie ovplyvnenie vodiča – popisuje, či bol vodič ovplyvnený vplyvom z vonkajšieho prostredia
    - označenie *p58_vonkajsie_ovplyvnenie_vodica*
- Kraj – popisuje kraj Českej republiky
    - označenie *kraj*

Tabuľka faktov dátového skladu nehôd

Vytvorený dátový sklad obsahuje jednu tabuľku faktov, ktorá sa nazýva *nehody*. Tabuľka má ako primárny kľúč identifikačné číslo nehody, ďalej obsahuje všetky kľúče jednotlivých dimenzií, okrem toho má ďalšie atribúty, ktoré určujú číslo pozemnej komunikácie, dátum nehody, deň v týždni, čas dopravnej nehody, následky zranených osôb, celkovú hmotnú škodu, počet zúčastnených vozidiel na dopravnej nehode, rok výroby vozidla, škoda na vozidle, súradnice dopravnej nehody.

3.4.2 Dátový sklad chodcov

Dátový sklad, vytvorený pre kategóriu chodcov, uchováva informácie o chodcoch, ktorí boli zúčastnení na dopravnej nehode. Grafická reprezentácia dátového skladu sa nachádza v *Prílohe B*.

Dimenzie dátového skladu chodcov

- Kategória chodca – popisuje, či bol chodec muž, žena, dieťa, ...
    - označenie *p29_kategoria_chodca*
- Stav chodca – popisuje, či bol chodec pod vplyvom liekov, alkoholu, nepozorný, ...
    - označenie *p30_stav_chodca*
- Správanie chodca – popisuje správanie chodca v čase dopravnej nehody (primerané, nerozhodné, náhle vstúpenie na vozovku, ...)
    - označenie *p31_chovanie_chodca*
- Situácia v mieste nehody – popisuje, či chodec vstúpil na vozovku, prechádzal okolo idúceho vozidla, ...
    - označenie *p32_situacia_miesta_nehody*

Tabuľka faktov dátového skladu chodcov

Dátový sklad obsahuje jednu tabuľku faktov s názvom *chodci* s primárnym kľúčom reprezentujúcim identifikačné číslo a kľúčmi dimenzionálnych tabuliek.

3.5 Optimalizácia dátovej štruktúry

Po vytvorení dátového skladu pre dáta, ktoré boli dostupné z dopravných nehôd sme pristúpili k experimentom, ktorých cieľom bolo určiť najefektívnejší spôsob vkladania a vyhľadávania údajov do dátového skladu, resp. tabuľky faktov.

V princípe sme vytvorili niekoľko tabuliek faktov pre nehody, ktoré sa líšili v spôsobe vytvorenia. Prvou z nich bola klasická tabuľka faktov s názvom *nehody*, ktorá obsahovala kľúče z jednotlivých dimenzionálnych tabuliek a ďalšie dodatočné informácie. *Def. príkazu 56* znázorňuje definíciu tabuľky faktov *nehody*.

```
CREATE TABLE nehody(
  id_nehody VARCHAR2(30) NOT NULL,
  id_druhu_pozemnej_komunikacie VARCHAR2(3),
  p2a_datum DATE,
  p2a_den_tyzdna VARCHAR2(3),
   ...,
  id_kraja VARCHAR2(3)
)
```

Definícia príkazu 56: Tabuľka faktov *nehody*

Druhú tabuľku faktov, ktorú sme vytvorili sa nazývala *nehody_particie*. Táto tabuľka obsahuje rovnaké atribúty ako vyššie uvedená tabuľka faktov *nehody*, avšak líši sa v tom, že má definované partície podľa dátumu nehody. Partície sme vytvorili pomocou kľúčového slova PARTITION BY RANGE, čo znamená, že sa vytvárajú partície definovaného rozsahu. Všetky partície, do ktorých budú dáta rozdelené sú vymenované v definícii tabuľky, pričom v tomto prípade počiatočná partícia *nehody_part_2015* bude obsahovať všetky údaje s dátumom nehody menším ako 1.1.2016 a posledná partícia *nehody_part_2022* bude obsahovať všetky údaje s dátumom väčším ako 31.12.2021. *Def. príkazu 57* znázorňuje definíciu tabuľky faktov *nehody_particie*.

```
CREATE TABLE nehody_particie(
  id_nehody VARCHAR2(30) NOT NULL,
  id_druhu_pozemnej_komunikacie VARCHAR2(3),
  p2a_datum DATE,
  p2a_den_tyzdna VARCHAR2(3),
  ...,
  id_kraja VARCHAR2(3))
PARTITION BY RANGE (p2a_datum)(
     PARTITION nehody_part_2015 VALUES LESS THAN
          (TO_DATE('01.01.2016', 'DD.MM.YYYY')),
     PARTITION nehody_part_2016 VALUES LESS THAN
          (TO_DATE('01.01.2017', 'DD.MM.YYYY')),
     PARTITION nehody_part_2017 VALUES LESS THAN
          (TO_DATE('01.01.2018', 'DD.MM.YYYY')),
     PARTITION nehody_part_2018 VALUES LESS THAN
          (TO_DATE('01.01.2019', 'DD.MM.YYYY')),
     PARTITION nehody_part_2019 VALUES LESS THAN
          (TO_DATE('01.01.2020', 'DD.MM.YYYY')),
     PARTITION nehody_part_2020 VALUES LESS THAN
          (TO_DATE('01.01.2021', 'DD.MM.YYYY')),
     PARTITION nehody_part_2021 VALUES LESS THAN
          (TO_DATE('01.01.2022', 'DD.MM.YYYY')),
     PARTITION nehody_part_2022
          VALUES LESS THAN (MAXVALUE)
)
```

Definícia príkazu 57: Tabuľka faktov *nehody_particie*

Tretí typ tabuľky je tabuľka faktov s názvom *nehody_particie_dyn*. Tabuľka je rozdelená do partícií podľa dátumu nehody, avšak partície sa vytvárajú dynamicky, to znamená, že sa vytvorí toľko partícií, koľko rôznych rokov budú obsahovať údaje. Funkcia *NUMTOYMINTERVAL(n, výraz)* konvertuje hodnotu *n* na interval pre rok alebo mesiac. Hodnota *n* je definovaná číslom, pričom *výraz* je jednotka konverzie, môže nadobúdať len 2 hodnoty: *YEAR* alebo *MONTH*. Definícia tejto tabuľky je znázornená v *def. príkazu 58*.

```
CREATE TABLE nehody_particie_dyn(
  id_nehody VARCHAR2(30) NOT NULL,
  id_druhu_pozemnej_komunikacie VARCHAR2(3),
  p2a_datum DATE,
  p2a_den_tyzdna VARCHAR2(3),
  ...,
  id_kraja VARCHAR2(3))
PARTITION BY RANGE (p2a_datum)
  INTERVAL( NUMTOYMINTERVAL(1, 'YEAR'))(
     PARTITION nehody_part_dyn1 VALUES LESS THAN
          (TO_DATE('01.01.2016', 'DD.MM.YYYY'))
)
```

Definícia príkazu 58: Tabuľka faktov *nehody_particie_dyn*

Ďalší typ tabuľky je tabuľka s názvom nehody_subparticie. Táto tabuľka je rozdelená na partície podľa rokov a subpartície podľa mesiacov. Partície podľa rokov sa vytvárajú dynamicky s využitím funkcie *NUMTOYMINTERVAL(n, výraz)* a vytvárajú sa pomocou klauzuly *PARTITION BY RANGE* (intervalové rozdelenie). Subpartície sú vytvorené pomocou kľúčového slova *SUBPARTITION BY LIST*, čo znamená, že sa vytvárajú subpartície s presne definovanou množinou hodnôt, v tomto prípade sú to hodnoty od 1 do 12, čo reprezentuje daný mesiac. Aby sme zistili mesiac z dátumu nehody a podľa neho vytvárali subpartície, bolo potrebné vytvoriť virtuálny stĺpec s názvom mesiac, ktorého hodnota sa automaticky doplní pri pridávaní údajov do tabuľky. Definícia tabuľky je znázornená v *def. príkazu 59*.

```
CREATE TABLE nehody_subparticie(
  id_nehody VARCHAR2(30) NOT NULL,
  id_druhu_pozemnej_komunikacie VARCHAR2(3),
  p2a_datum DATE,
  p2a_den_tyzdna VARCHAR2(3),
  ...,
  id_kraja VARCHAR2(3),
  mesiac INTEGER GENERATED ALWAYS AS
          (EXTRACT(MONTH FROM p2a_datum)) VIRTUAL)
PARTITION BY RANGE (p2a_datum)
     INTERVAL( NUMTOYMINTERVAL(1, 'YEAR'))
SUBPARTITION BY LIST (mesiac)
SUBPARTITION TEMPLATE(
   SUBPARTITION mesiac_1 VALUES (1),
   SUBPARTITION mesiac_2 VALUES (2),
   SUBPARTITION mesiac_3 VALUES (3),
   SUBPARTITION mesiac_4 VALUES (4),
   SUBPARTITION mesiac_5 VALUES (5),
   SUBPARTITION mesiac_6 VALUES (6),
   SUBPARTITION mesiac_7 VALUES (7),
   SUBPARTITION mesiac_8 VALUES (8),
   SUBPARTITION mesiac_9 VALUES (9),
   SUBPARTITION mesiac_10 VALUES (10),
   SUBPARTITION mesiac_11 VALUES (11),
   SUBPARTITION mesiac_12 VALUES (12))(
     PARTITION nehody_rok VALUES LESS THAN
          (TO_DATE('01.01.2016', 'DD.MM.YYYY'))
)
```

Definícia príkazu 59: Tabuľka faktov nehody_subparticie

Po vytvorení týchto tabuliek faktov, sme vytvorili jednotlivé *.ctl* súbory, ktoré slúžia na import dát do databázy. *Def. príkazu 60* znázorňuje *.ctl* súbor pre tabuľku *nehody*.

```
OPTIONS(SKIP=1)
LOAD DATA
CHARACTERSET UTF8
INFILE
'C:\Users\durne\Desktop\Dopravne_nehody\data\nehody.csv' "str
'\n'"
APPEND
INTO TABLE nehody
FIELDS TERMINATED BY ';'
OPTIONALLY ENCLOSED BY '"' AND '"'
TRAILING NULLCOLS
     ( ID_NEHODY,
       ID_DRUHU_POZEMNEJ_KOMUNIKACIE,
       P2A_DATUM DATE "YYYY-MM-DD",
       P2A_DEN_TYZDNA,
       ...,
       id_kraja
      )
```

Definícia príkazu 60: *.ctl* súbor pre tabuľku *nehody*

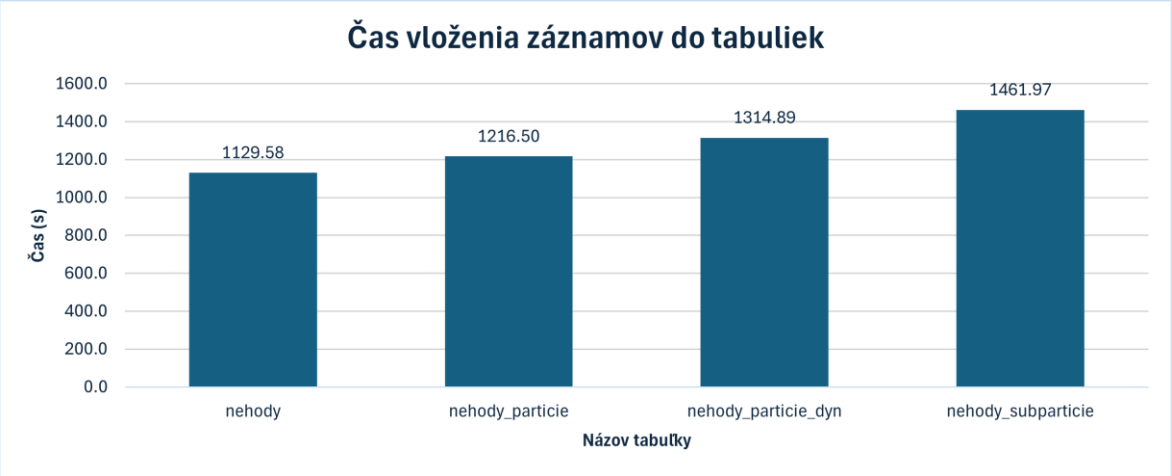
V súbore sme definovali kódovanie znakov pomocou UTF-8. V časti *INFILE* sme definovali úplnú cestu k *.csv* súboru, ktorý obsahuje všetky údaje pre načítanie. Následne sme definovali tabuľku, do ktorej sa údaje budú vkladať. Pre každú tabuľku je vytvorený vlastný *.ctl* súbor. Tieto súbory sú rovnaké, líšia sa len názvom tabuľky, do ktorej sa budú dáta vkladať. Jednotlivé hodnoty sú oddelené ; (bodkočiarkou), čo sme definovali v časti *FIELDS TERMINATED BY*. Niektoré hodnoty môžu byť uzatvorené v zátvorkách, to sa definuje v časti *OPTIONALLY ENCLOSED BY*. V súbore bolo potrebné použiť aj klauzulu *TRAILING NULLCOLS*, čo znamená, že všetky stĺpce, ktoré nie sú vyplnené budú nadobúdať NULL hodnotu. Potom sme vypísali stĺpce v rovnakom poradí ako sú v *.csv* súbore.

Dataset, ktorý sme vkladali do tabuliek faktov obsahoval približne 700 000 záznamov. Na vloženie záznamov do databázy sme využili metódu SQL*Loader. *Tabuľka 17* obsahuje čas vloženia záznamov v sekundách do jednotlivých tabuliek faktov.

Tabuľka 17: Čas vkladania údajov do tabuliek faktov (v sekundách)

Tabuľka	Čas (s)
nehody	1 129,58
nehody_particie	1 216,50
nehody_particie_dyn	1 314,89
nehody_subparticie	1 461,97

Ako je možné vidieť v tabuľke, čas vkladania záznamov do klasickej tabuľky faktov bez partícií je najnižší, a to z toho dôvodu, že nepotrebuje dodatočnú réžiu na vytváranie jednotlivých partícií. Na druhej strane, tabuľka faktov, v ktorej sa vytvárali subpartície mesiacov pre jednotlivé partície rokov, mala čas vkladania záznamov do databázy najvyšší. Na *Obrázku 26* je grafické zobrazenie času vkladania záznamov do tabuliek.



Obrázok 26: Stĺpcový graf - čas vloženia záznamov do tabuliek

XML dokumenty

XML (*eXtensible Markup Language*) je jazyk pre popis dát. Ukladá štruktúrovaný a čiastočne štruktúrovaný text, ktorý sa využíva v rozličných aplikáciách. XML bol navrhnutý pre ukladanie a transport dát. XML dokumenty obsahujú špecifické znaky nazývané tagy, entity a elementy. Konečný XML dokument je samo popisujúci, to znamená, že okrem definície dát definuje aj ich význam. XML nemá preddefinované značky (tagy), preto je nutné vytvoriť si vlastné značky. XML dokument musí obsahovať jeden takzvaný *koreňový element*, ktorý pokrýva všetky ostatné elementy [12] [52].

Každý XML dokument musí byť správne štruktúrovaný. Preto tvorba XML dokumentov má určité pravidlá medzi, ktoré patria napríklad:

- XML elementy sú párové značky, musia mať uzatvárajúci tag
- XML tagy sú case-sensitive (rozlišujú veľké a malé písmená)

- Elementy môžu byť ľubovoľne vnorené, avšak musia byť správne uzatvorené
- Atribúty musia byť v úvodzovkách
- ...

Základná syntax XML dokumentu je nasledovná (*def. príkazu 61*):

```
<root>
   <element>
      . . .
   </element>
</root>
```

Definícia príkazu 61: Syntax XML dokumentu

Nasledujúci príklad (*def. príkazu 62*) je ukážkou XML dokumentu, ktorý sme si vytvorili z údajov z dopravných nehôd. Ako je možné vidieť na príklade, XML dokument obsahuje prvý riadok, takzvaný *XML Prológ* <? xml ..., ktorý je voliteľný a zahŕňa verziu a kódovanie znakov. V našom prípade sú znaky kódované pomocou UTF-8.

Koreňový element predstavuje element *nehoda*, a má 2 atribúty, *id_nehody* a *p2a_datum*. Následne obsahuje ďalšie vnorené elementy.

```
<?xml version="1.0" encoding="UTF-8"?>
     <nehoda id_nehody="002100170121" p2a_datum="2017-01-03">
          <p2a_den_tyzdna>2</p2a_den_tyzdna>
          <p2b_cas>1130</p2b_cas>
          <miesto_nehody>
               <id_lokality>1</id_lokality>
               <id_kraja>0</id_kraja>
               <id_miesta_nehody>00</id_miesta_nehody>
               <id_situovania_nehody>1</id_situovania_nehody>
          </miesto_nehody>
          <pozemna_komunikacia>
               <id_druhu_pozemnej_komunikacie>6
                    </id_druhu_pozemnej_komunikacie>
               <id_stavu_komunikacie>01</id_stavu_komunikacie>
               <id_delenia_komunikacie>1</id_delenia_komunikacie>
          </pozemna_komunikacia>
          ...
          <nasledky_nehody>
               <p13a_nasledky_usmrtene_osoby>0
                    </p13a_nasledky_usmrtene_osoby>
               <p13b_nasledky_tazko_zranene_osoby>0
                    </p13b_nasledky_tazko_zranene_osoby>
               ...
          </nasledky_nehody>
          ...
     </nehoda>
```

Definícia príkazu 62: Štruktúra XML dokumentu pre nehody

Na pohyb v rámci stromu XML dokumentu slúži jazyk XPath. Ak chceme získať hodnotu daného elementu alebo atribútu, musíme zadať k nemu správnu cestu.

- Koreňový element získame nasledujúcim spôsobom (*def. príkazu 63*):

```
/nehoda
```

Definícia príkazu 63: Koreňový element

- Vnútorný element získame vypísaním absolútnej cesty (*def. príkazu 64*):

```
/nehoda/p2a_den_tyzdna
```

Definícia príkazu 64: Vnútorný element

- Hodnotu atribútu získame pomocou znaku @ (*def. príkazu 65*):

```
/nehoda/@p2a_datum
```

Definícia príkazu 65: Atribút XML dokumentu

Je mnoho ďalších spôsobov ako pristúpime k elementu, prostredníctvom relatívnej cesty, prípadne získame hodnoty elementov podľa definovanej podmienky, avšak v našom prípade budeme potrebovať pristúpiť k elementom a atribútom.

SQLX funkcie sú funkcie, ktoré sa využívajú na generovanie XML dokumentov v SQL. Medzi niektoré z týchto funkcií patria [12] [55]:

- *XMLRoot()* – zabezpečuje vytvorenie prvého tagu *<?xml…* Okrem toho slúži aj na naformátovanie výsledného XML dokumentu
- *XMLElement()* – zabezpečuje vytvorenie XML elementu
- *XMLAttributes()* – zabezpečuje vytvorenie XML atribútu pre element, hodnoty atribútov sú v úvodzovkách a atribúty sú oddelené čiarkami
- *XMLForest()* – zabezpečuje vytvorenie XML elementov, ktoré sa nachádzajú na rovnakej úrovni
- *XMLAgg()* – vytvorí jeden XML dokument z niekoľkých riadkov získaných príkazom *SELECT*, je to agregačná funkcia.
- *XMLConcat()* – zabezpečuje spojenie viacerých XML dokumentov do jedného
- *XMLComment()* – zabezpečuje vytvorenie komentáru

Ďalšie funkcie, ktoré sa využívajú v databázových systémoch, sú funkcie pre výber prvkov z XML dokumentov. Medzi tieto funkcie patrí:

- *Extract(xml_type, xml_path)* – sprístupňuje podstrom XML dokumentu
- *ExtractValue(xml_type, xml_path)* – sprístupňuje skalárnu hodnotu elementu XML dokumentu
- *ExistsNode(xml_type, xml_path)* – vracia hodnotu 1, ak sa v XML dokumente nachádza príslušný uzol

JSON dokumenty

JSON (*JavaScript Object Notation*) je taktiež textový formát na ukladanie a prenos údajov, podobne ako XML aj JSON je samopopisný, jazykovo nezávislý, obsahuje hierarchickú štruktúru. Syntax JSON-u je odvodená od JavaScript syntaxe a má určité pravidlá [51] [52]:

- Údaje sú uvedené v tvare "názov": "hodnota"
- Údaje sú oddelené čiarkami
- Objekty sa nachádzajú v { } zátvorkách
- Polia sa nachádzajú v [ ] zátvorkách

JSON nepoužíva tagy, je rýchlejší na zápis a čítanie, ale najväčší rozdiel medzi ním a XML je v parsovaní. XML na parsovanie využíva XML parser, zatiaľ čo JSON môže byť parsovaný štandardne funkciou JavaScript-u.

JSON má ohraničené možnosti využívania dátových typov. Hodnoty musia nadobúdať buď typ reťazec, číslo, objekt, pole, boolean alebo null. JSON nedokáže spracovať hodnoty dátumových typov a nedefinované hodnoty.

Syntax JSON dokumentu je znázornená na nasledujúcom kóde (*def. príkazu 66*):

```
{
   "kľúč" : "hodnota",
   "kľúč" :
   {
      "kľúč" : "hodnota",
      "kľúč" :
         [
            {"kľúč" : "hodnota"},
            {"kľúč" : "hodnota"}
         ]
   }
}
```

Definícia príkazu 66: Syntax JSON dokumentu

Rovnako ako pri XML dokumente, aj tu sme si vytvorili štruktúru výsledného JSON dokumentu z údajov z dopravných nehôd, ktorý budeme vkladať do databázovej tabuľky. JSON dokument má nasledovnú štruktúru (*def. príkazu 67*):

```
{
   "id_nehody" : "002100170121",
   "p2a_datum" : "2017-01-03T00:00:00",
   "p2a_den_tyzdna" : "2",
   "p2b_cas" : "1130",
   "miesto_nehody" :
     {
        "id_lokality" : "1",
        "id_kraja" : "0",
        "id_miesta_nehody" : "00",
        "id_situovania_nehody" : "1"
     },
   "pozemna_komunikacia" :
     {
        "id_druhu_pozemnej_komunikacie" : "6",
        "p37_cislo_pozemnej_komunikacie" : null,
        "id_stavu_komunikacie" : "01",
        "id_druhu_krizujucej_komunikaciu" : null
     }
   . . .
   "nasledky_nehody" :
     {
        "p13a_nasledky_usmrtene_osoby" : "0",
        "p13b_nasledky_tazko_zranene_osoby" : "0"
        . . .
     }
   . . .
}
```

Definícia príkazu 67: Štruktúra JSON dokumentu pre nehody

V databázovom systéme Oracle je prístup k elementom JSON dokumentu zabezpečený funkciami [53]:

- *JSON_VALUE(json, path)* – zabezpečuje výber požadovaných údajov z JSON dokumentu
- *JSON_QUERY(json, path)* – extrahuje pole alebo reťazec z dokumentu
- *JSON_TABLE(json, path)* – mapuje JSON dáta do stĺpcov tabuľky

Podmienky ktoré sa viažu k JSON elementom sú [53]:

- *JSON_EXISTS(json, path)* – vráti boolean hodnotu *TRUE,* ak dokument obsahuje zadané údaje, v opačnom prípade vráti hodnotu *FALSE*
- *ISJSON(json)* **–** kontroluje, či je JSON správny
- *JSON_TEXTCONTAINS(json, path)* – kontroluje, či sa definovaný reťazec nachádza v JSON dokumente

Vďaka vyššie uvedeným funkciám je možné pracovať s obsahom JSON dokumentov. Na to, aby sme mohli taký JSON dokument vygenerovať, existujú funkcie, ktoré to zabezpečujú. Medzi tieto funkcie patrí [54]:

- *JSON_Object()* – zabezpečuje vytvorenie JSON objektov
- *JSON_Array()* – zabezpečuje vytvorenie polí
- *JSON_ObjectAgg()* – agregačná funkcia, ktorá zabezpečuje vytvorenie objektu množiny kľúč-hodnota
- *JSON_ArrayAgg()* – agregačná funkcia, ktorá zabezpečuje združenie hodnôt do poľa

Experimenty

Cieľom nasledujúcich experimentov je určiť najefektívnejší spôsob vyhľadávania údajov v rôznych tabuľkách, ktoré uchovávajú dáta rozličnými spôsobmi:

- klasická tabuľka faktov
- tabuľky rozdelené na partície (vymenovaním, dynamicky)
- tabuľka rozdelená na subpartície
- tabuľky uchovávajúce objekty typu XML a JSON

Efektívnosť vyhľadávania sa budeme snažiť zlepšiť vytvorením vhodných indexov, ktoré zlepšia rýchlosť vyhľadávania v jednotlivých tabuľkách. Príkaz *SELECT*, ktorý budeme testovať bude vyberať údaje na základe dátumu nehody, pričom budeme pozorovať ako sa výkonnosť príkazu mení v prípade, že vyberáme údaje podľa:

- definovaného roku
- definovaného mesiaca v danom roku
- definovaného mesiaca vo všetkých rokoch

Aby sme mohli pracovať s objektami typu XML a JSON bolo potrebné vytvoriť ďalšie tabuľky a vložiť do nich všetky záznamy.

Tabuľka s názvom *nehody_xml_tab_1* reprezentuje tabuľku ako tabuľku objektov XML dokumentov, v ktorej je každý XML dokument nový riadok tabuľky. Táto tabuľka obsahuje XML dokumenty, v ktorej bude vyhľadávaný stĺpec (dátum) element XML dokumentu. Tabuľku sme vytvorili nasledovným spôsobom (*def. príkazu 68*):

```
CREATE TABLE nehody_xml_tab_1 OF XMLTYPE;
```

Definícia príkazu 68: Vytvorenie tabuľky XML dokumentov

Tabuľka *nehody_xml_tab_2* reprezentuje taktiež tabuľku XML dokumentov, avšak obsahuje XML dokumenty, v ktorých je vyhľadávaný stĺpec ako atribút elementu.

Tabuľka *nehody_xml_atr_1* reprezentuje tabuľku, v ktorej je jeden zo stĺpcov samotný XML dokument. Tento stĺpec je definovaný typom *XMLTYPE*. Okrem toho, tabuľka obsahuje ďalší stĺpec *id_xml*, ktorý reprezentuje identifikačné číslo XML dokumentu. Toto číslo sa bude navyšovať o hodnotu 1, takže predstavuje sekvenciu, ktorú sme vytvorili pod názvom *seq_nehody_xml_atr_1*. Nasledujúci SQL kód predstavuje vytvorenie tabuľky (*def príkazu 69*) a sekvencie (*def. príkazu 70*).

```
CREATE TABLE nehody_xml_atr_1(
  id_xml    INTEGER NOT NULL PRIMARY KEY,
  xml_data  XMLTYPE NOT NULL
);
```

Definícia príkazu 69: Tabuľka s XML dokumentom ako atribútom

Táto tabuľka obsahuje XML dokumenty, v ktorej vyhľadávaný stĺpec (dátum) je vnútorný element dokumentu.

```
CREATE SEQUENCE seq_nehody_xml_atr_1
 START WITH 1
 INCREMENT BY 1;
```

Definícia príkazu 70: Sekvencia pre XML dokumenty

Tabuľka *nehody_xml_atr_2* reprezentuje taktiež tabuľku, ktorá obsahuje stĺpec typu XML dokument, avšak XML dokumenty majú vyhľadávaný stĺpec reprezentovaný ako atribút elementu. Rovnako ako vo vyššie uvedenom príklade aj tu sme vytvorili sekvenciu *seq_nehody_xml_atr_2*, ktorá sa pri vkladaní XML dokumentov automaticky navyšuje o hodnotu 1.

Pre JSON dokumenty sme vytvorili tabuľku s názvom *nehody_json_1*, ktorá obsahuje dva atribúty, *id_json* reprezentuje identifikačné číslo JSON dokumentu, pre ktorý bola vytvorená sekvencia *seq_nehody_json_1* (*def. príkazu 72*) na automatické navyšovanie hodnoty o 1. Druhý atribút je *json_data*, ktorý je typu *CLOB,* avšak môže nadobúdať len JSON hodnoty, čo je definované v časti *CONSTRAINT*. Definíciu tabuľky zobrazuje *def. príkazu 71*.

```
CREATE TABLE nehody_json_1(
  id_json    INTEGER NOT NULL PRIMARY KEY,
  json_data  CLOB NOT NULL,
   CONSTRAINT valid_json_cons CHECK (json_data IS JSON)
);
```

Definícia príkazu 71: Tabuľka pre JSON dokumenty

```
CREATE SEQUENCE seq_nehody_json_1
 START WITH 1
 INCREMENT BY 1;
```

Definícia príkazu 72: Sekvencia pre JSON dokumenty

Po vytvorení všetkých objektov sme si prostredníctvom metód na generovanie XML a JSON dokumentov naplnili tabuľky všetkými záznamami z tabuľky *nehody*. SQL kód pre naplnenie tabuľky *nehody_xml_atr_2* dátami zobrazuje *def. príkazu 73*, na ktorom je definovaný SQL príkaz *INSERT*, ktorý prostredníctvom príkazu *SELECT* vloží do tabuľky *nehody_xml_atr_2* identifikačné číslo dokumentu, vďaka vytvorenej sekvencii a jej metóde *NEXTVAL* a ako druhý atribút je vygenerovaný XML dokument z tabuľky *nehody*. Podobným spôsobom sme vložili záznamy aj do tabuľky nehody_xml_atr_1, ale vygenerovaný XML dokument obsahoval hodnotu stĺpca p2a_datum ako samostatný element, ktorý sa nachádza na úrovni elementu pre označenia dňa v týždni a času. Pre tabuľku objektov *nehody_xml_tab_1* a *nehody_xml_tab_2* sme použili rovnaký formát generovania XML dokumentov, avšak príkaz *INSERT* obsahoval len *SELECT* s generovaním XML dokumentu, bez sekvencie, keďže tieto tabuľky sú tabuľky objektov, ktoré majú len jeden stĺpec s názvom *OBJECT_VALUE*.

```
INSERT INTO nehody_xml_atr_2(
  SELECT seq_nehody_xml_atr_2.NEXTVAL,
    XMLRoot(
      XMLElement("nehoda",
         XMLAttributes(id_nehody AS "id_nehody",
                       p2a_datum AS "p2a_datum"),
        XMLForest( p2a_den_tyzdna AS "p2a_den_tyzdna",
                   p2b_cas AS "p2b_cas"),
      XMLElement("miesto_nehody",
        XMLForest(id_lokality AS "id_lokality",
                  id_kraja AS "id_kraja",
                  id_miesta_nehody AS "id_miesta_nehody",
                  id_situovania_nehody AS "id_situovania_nehody")),
      XMLElement("pozemna_komunikacia",
        XMLForest(id_druhu_pozemnej_komunikacie AS
                           "id_druhu_pozemnej_komunikacie",
                  p37_cislo_pozemnej_komunikacie AS
                           "p37_cislo_pozemnej_komunikacie",
                  id_stavu_komunikacie AS "id_stavu_komunikacie",
                  id_delenia_komunikacie AS
                           "id_delenia_komunikacie",
                  id_druhu_krizujucej_komunikaciu AS
                           "id_druhu_krizujucej_komunikaciu")),
          ...
       XMLElement("nasledky_nehody",
         XMLForest(p13a_nasledky_usmrtene_osoby AS
                           "p13a_nasledky_usmrtene_osoby",
                   p13b_nasledky_tazko_zranene_osoby AS
                           "p13b_nasledky_tazko_zranene_osoby",
                   ...)),
            ... ), VERSION '1.0" encoding="UTF-8' ) AS XML
  FROM nehody );
```

Definícia príkazu 73: Vloženie záznamov do tabuľky s XML atribútom

SQL kód pre naplnenie tabuľky *nehody_json_1* zobrazuje *def. príkazu 74*. Je tu definovaný príkaz *INSERT*, ktorý pomocou príkazu *SELECT* vloží do tabuľky identifikačné číslo JSON dokumentu vďaka vytvorenej sekvencii a jej metóde *NEXTVAL*, a ako druhý atribút príkazu je vygenerovaný JSON dokument z tabuľky *nehody*.

```
INSERT INTO nehody_json_1(
  SELECT seq_nehody_json_1.NEXTVAL,
         JSON_OBJECT(
          'id_nehody' VALUE id_nehody,
          'p2a_datum' VALUE p2a_datum,
          'p2a_den_tyzdna' VALUE p2a_den_tyzdna,
          'p2b_cas' VALUE p2b_cas,
          'miesto_nehody' VALUE
             JSON_OBJECT(
              'id_lokality' VALUE id_lokality,
              'id_kraja' VALUE id_kraja,
              'id_miesta_nehody' VALUE id_miesta_nehody,
              'id_situovania_nehody' VALUE id_situovania_nehody),
          'pozemna_komunikacia' VALUE
             JSON_OBJECT(
              'id_druhu_pozemnej_komunikacie' VALUE
                     id_druhu_pozemnej_komunikacie,
              'p37_cislo_pozemnej_komunikacie' VALUE
                     p37_cislo_pozemnej_komunikacie,
              'id_stavu_komunikacie' VALUE id_stavu_komunikacie,
              . . . ),
          . . .
          'nasledky_nehody' VALUE
             JSON_OBJECT(
              'p13a_nasledky_usmrtene_osoby' VALUE
                     p13a_nasledky_usmrtene_osoby,
              'p13b_nasledky_tazko_zranene_osoby' VALUE
                     p13b_nasledky_tazko_zranene_osoby,
              . . . ),
           . . .
      )
  FROM nehody );
```

Definícia príkazu 74: Vloženie záznamov do tabuľky s JSON dokumentami

Tabuľka faktov *nehody*

Experimenty sme vykonali nad klasickou tabuľkou faktov *nehody*. Vytvorili sme tri typy príkazu *SELECT*, ktorý sa líšil v podmienke. V prvom kroku sme získali náklady na

vykonanie príkazov *SELECT* bez vytvorenia indexov. *Def. príkazu 75* zobrazuje príkaz, ktorý vyberie všetky záznamy z tabuľky *nehody*, ktoré sa stali v roku 2018. Na výber roku zo stĺpca *p2a_datum* sme využili funkciu TO_CHAR().

```
SELECT *
 FROM nehody
 WHERE TO_CHAR(p2a_datum, 'YYYY') = '2018';
```

Definícia príkazu 75: *Select* - rok nehody pre tabuľku *nehody*

Plán vykonania tohto príkazu je znázornený na *Obrázku 27*. Ako je možné vidieť, optimalizátor využil prístupovú metódu *TABLE ACCESS FULL* a prehľadával celú tabuľku. Celkové náklady na vykonanie tohto príkazu sú 7 745.

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |        |  6996 |  2042K|  7745   (1)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| NEHODY |  6996 |  2042K|  7745   (1)| 00:00:01 |
-------------------------------------------------------------------------------
```

Obrázok 27: Plán vykonania - *Select* rok nehody pre tabuľku *nehody*

Druhý príkaz, ktorý sme vykonali, zobrazuje *def. príkazu 76*. Tento príkaz sa líši v podmienke, v ktorej sme vybrali všetky záznamy nehôd z mája 2018.

```
SELECT *
 FROM nehody
 WHERE TO_CHAR(p2a_datum, 'MM.YYYY') = '05.2018';
```

Definícia príkazu 76: *Select* - mesiac a rok nehody pre tabuľku *nehody*

Plán vykonania tohto príkazu sa nezmenil oproti vyššie uvedenému, keďže optimalizátor využil prístupovú metódu *TABLE ACCESS FULL* a prehľadával opäť celú tabuľku. Celkové náklady na vykonanie tohto príkazu zostali nezmenené 7 745. Plán vykonania je znázornený na *Obrázku 27*.

Tretí príkaz, ktorý sme sledovali, vyberá z tabuľky záznamy nehôd, ktoré sa udiali v máji, pričom rok vôbec nezohľadňujeme. *Def. príkazu 77* zobrazuje tento príkaz.

```
SELECT *
 FROM nehody
 WHERE TO_CHAR(p2a_datum, 'MM') = '05';
```

Definícia príkazu 77: *Select* - mesiac nehody pre tabuľku *nehody*

Plán vykonania tohto príkazu je znázornený na *Obrázku 27*, pretože ako v predchádzajúcom príklade, ani v tomto sa náklady na vykonanie príkazu nezmenili, keďže

optimalizátor využil prístupovú metódu *TABLE ACCESS FULL* a prehľadával celú tabuľku. Celkové náklady na vykonanie tohto príkazu zostali 7 745.

Následne sme si vytvorili indexy a sledovali sme, či sa náklady na jednotlivé príkazy zmenia alebo nie. Ako prvý index sme vytvorili funkcionálny index *ind_nehody_datum_rok*, znázornený v *def. príkazu 78*. Index obsahoval funkciu TO_CHAR(), vďaka ktorej získame rok z dátumu.

```
CREATE INDEX ind_nehody_datum_rok
 ON nehody
 (TO_CHAR(p2a_datum, 'YYYY'));
```

Definícia príkazu 78: Index - pre rok nehody v tabuľke *nehody*

Po opätovnom spustení prvého príkazu a zobrazení plánu vykonania vidíme, že náklady na vykonanie tohto príkazu sa znížili na hodnotu 338, a to preto, lebo sa využil vytvorený index. Plán vykonania príkazu po vytvorení indexu je znázornený na *Obrázku 28*.

```
-------------------------------------------------------------------------------------------
| Id  | Operation              | Name                 | Rows  |Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                      |  6996 | 2042K|   338   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY       |                      |       |      |            |          |
|   2 |    INDEX ROWID BATCHED | NEHODY               |  6996 | 2042K|   338   (0)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN     | IND_NEHODY_DATUM_ROK |  2798 |      |   224   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 28: Plán vykonania - *Select* rok nehody s indexom pre tabuľku nehody

Def. príkazu 79 zobrazuje kód ďalšieho funkcionálneho indexu, ktorý sme vytvorili pre príkaz *SELECT* s podmienkou pre mesiac a rok nehody.

```
CREATE INDEX ind_nehody_datum_mesiac_rok
 ON nehody
 (TO_CHAR(p2a_datum, 'MM.YYYY'));
```

Definícia príkazu 79: Index - pre mesiac a rok nehody v tabuľke *nehody*

Po opätovnom spustení príkazu dostávame plán vykonania, ktorý je znázornený na *Obrázku 29*. Ako je možné vidieť, náklady na vykonanie príkazu s podmienkou pre mesiac a rok sa znížili na hodnotu 201, pričom optimalizátor použil prístupovú metódu s využitím indexu.

```
-------------------------------------------------------------------------------------------
| Id | Operation              | Name                       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT       |                            |  6996 |  2042K|   201   (0)| 00:00:01 |
|  1 |  TABLE ACCESS BY       |                            |       |       |            |          |
|  2 |    INDEX ROWID BATCHED | NEHODY                     |  6996 |  2042K|   201   (0)| 00:00:01 |
|* 3 |   INDEX RANGE SCAN     | IND_NEHODY_DATUM_MESIAC_ROK |  2798 |       |    24   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 29: Plán vykonania - *Select* mesiac a rok nehody s indexom pre tabuľku *nehody*

Posledný index, ktorý sme vytvorili je určený pre tretí príkaz a je ho možné vidieť v *def. príkazu 80*, pričom je to funkcionálny index s využitím funkcie TO_CHAR(), pomocou ktorej vyberáme mesiac z daného dátumu.

```
CREATE INDEX ind_nehody_datum_mesiac
 ON nehody
 (TO_CHAR(p2a_datum, 'MM'));
```

Definícia príkazu 80: Index - pre mesiac nehody v tabuľke *nehody*

Po spustení tretieho príkazu sme získali plán vykonania, ako je zobrazené na *Obrázku 30*. Po vytvorení indexu je možné vidieť zníženie nákladov na hodnotu 292, pričom optimalizátor využil vytvorený index pre mesiac.

```
-----------------------------------------------------------------------------------------------
| Id | Operation               | Name                   | Rows | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT        |                        | 6996 |  2042K|   292   (0)| 00:00:01 |
|  1 |  TABLE ACCESS BY        |                        |      |       |            |          |
|  2 |    INDEX ROWID BATCHED | NEHODY                 | 6996 |  2042K|   292   (0)| 00:00:01 |
|* 3 |   INDEX RANGE SCAN      | IND_NEHODY_DATUM_MESIAC | 2798 |       |   115   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

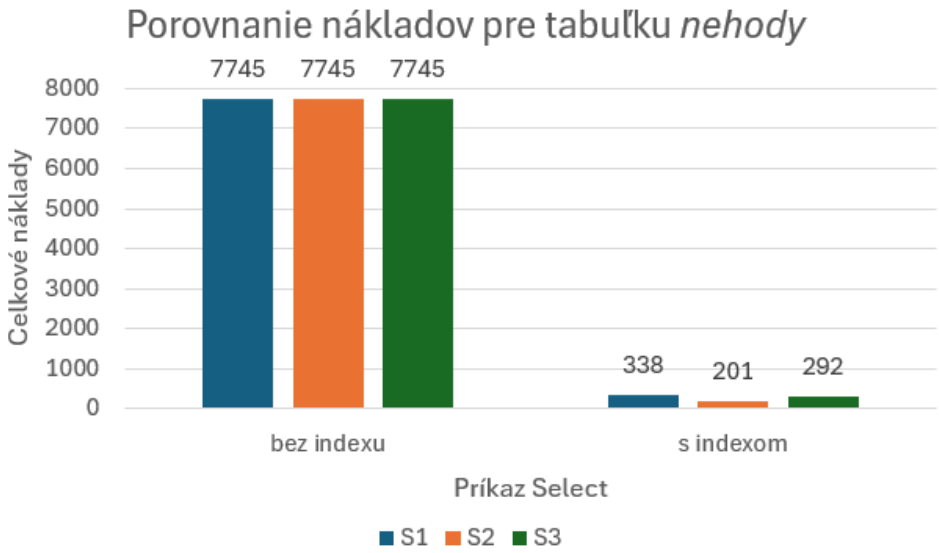
Obrázok 30: Plán vykonania - *Select* mesiac nehody s indexom pre tabuľku *nehody*

Zhrnutie jednotlivých nákladov na vykonanie vyššie uvedených príkazov sa nachádza v *Tabuľke 18*.

Tabuľka 18: Zhrnutie nákladov príkazu *SELECT* pre tabuľku *nehody*

Podmienka	Náklady (bez indexu)	Náklady (s indexom)
Príkaz pre rok	7 745	338
Príkaz pre mesiac a rok	7 745	201
Príkaz pre mesiac	7 745	292

Grafická reprezentácia porovnania celkových nákladov na vykonanie príkazov sa nachádza na *Obrázku 31*. Celkové náklady predstavujú internú hodnotu, na základe ktorej sa optimalizátor rozhoduje.



Obrázok 31: Stĺpcový graf porovnania nákladov pre tabuľku *nehody*

Tabuľka faktov *nehody_particie*

Ďalšie experimenty sme vykonali nad tabuľkou faktov *nehody_particie*, ktorá je rozdelená na partície podľa roku nehody. Sledovali sme zmeny, ktoré nastali v pláne vykonania a celkových nákladoch na vykonanie príkazov *SELECT*. Najskôr sme tieto príkazy spustili bez vytvorenia indexov. V príkaze, ktorý vyberá všetky záznamy nehôd, ktoré sa stali v roku 2018, sme využili klauzulu *PARTITION,* v ktorej sme napísali partíciu, z ktorej chceme záznamy vyberať. Tento príkaz je znázornený na nasledujúcom príklade (*def. príkazu 81*). Partície tabuliek je možné zistiť vďaka systémovej tabuľke *USER_TAB_PARTITIONS*.

```
SELECT *
 FROM nehody_particie PARTITION (NEHODY_PART_2018);
```

Definícia príkazu 81: *Select* - rok nehody pre tabuľku *nehody_particie*

Plán vykonania pre tento príkaz je znázornený na *Obrázku 32*. Ako je možné vidieť, celkové náklady na vykonanie tohto príkazu sú 1 375.

```
-------------------------------------------------------------------------------------------
| Id  | Operation              | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                 |   104K|    29M|  1375   (1)| 00:00:01 |
|   1 |  PARTITION RANGE SINGLE|                 |   104K|    29M|  1375   (1)| 00:00:01 |
|   2 |   TABLE ACCESS FULL    | NEHODY_PARTICIE |   104K|    29M|  1375   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 32: Plán vykonania - *Select* rok nehody pre tabuľku *nehody_partície*

Nasledujúci príkaz (*def. príkazu 82*) vyberie z tabuľky *nehody_particie* všetky záznamy nehôd, ktoré sa stali v máji 2018. V definícií príkazu sme použili klauzulu

PARTITION, v ktorej sme definovali partíciu pre rok 2018 a v podmienke sme definovali mesiac.

```
SELECT *
 FROM nehody_particie PARTITION(NEHODY_PART_2018)
 WHERE TO_CHAR(p2a_datum, 'MM') = '05';
```

Definícia príkazu 82: *Select* - mesiac a rok nehody pre tabuľku *nehody_particie*

Plán vykonania tohto príkazu je znázornený na *Obrázku 33.* Ako je možné vidieť, náklady na vykonanie tohto príkazu nadobudli hodnotu 1 372.

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                 |  1048 |   306K|  1372   (1)| 00:00:01 |
|   1 |  PARTITION RANGE SINGLE|                 |  1048 |   306K|  1372   (1)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL   | NEHODY_PARTICIE |  1048 |   306K|  1372   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 33: Plán vykonania - *Select* mesiac a rok nehody pre tabuľku *nehody_partície*

Nasledujúci príkaz (*def. príkazu 83*) vyberie z tabuľky *nehody_particie* všetky záznamy nehôd, ktoré sa stali v máji, nezávisle od roku.

```
SELECT *
 FROM nehody_particie
 WHERE TO_CHAR(p2a_datum, 'MM') = '05';
```

Definícia príkazu 83: *Select* - mesiac nehody pre tabuľku *nehody_particie*

Plán vykonania tohto príkazu je zobrazený na *Obrázku 34.* Keďže v predchádzajúcich dvoch príkladoch sme vyhľadávali záznamy z konkrétnej partície, náklady na vykonanie príkazov boli nižšie. V tomto prípade však vyhľadávame záznamy z celej tabuľky a vyberáme len tie, ktoré sa stali v máji. Preto celkové náklady na vykonanie príkazu majú hodnotu 8 873.

```
-------------------------------------------------------------------------------------------
| Id  | Operation           | Name            | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                 |  6996 |  2022K|  8873   (1)| 00:00:01 |
|   1 |  PARTITION RANGE ALL|                 |  6996 |  2022K|  8873   (1)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL | NEHODY_PARTICIE |  6996 |  2022K|  8873   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 34: Plán vykonania - *Select* mesiac nehody pre tabuľku *nehody_partície*

V tabuľkách rozdelených na partície sa využívajú dva typy indexov – ***globálny index*** a ***lokálny index***. Experimenty sme vykonali nad oboma typmi indexov a sledovali sme zmeny, ktoré nastali v pláne vykonania. V prvom rade sme si vytvorili niekoľko typov globálnych indexov, v ktorých sme indexovali časti dátumu. Nasledujúci príkaz (*def. príkazu 84*) vytvorí globálny funkcionálny index pre rok nad tabuľkou *nehody_particie*.

```
CREATE INDEX ind_nehody_particie_datum_rok
 ON nehody_particie
 (TO_CHAR(p2a_datum, 'YYYY'));
```

Definícia príkazu 84: Globálny index - pre rok nehody v tabuľke *nehody_particie*

Ďalší príkaz (*def. príkazu 85*) vytvorí index, ktorý indexuje mesiace a rok.

```
CREATE INDEX ind_nehody_particie_datum_mesiac_rok
 ON nehody_particie
 (TO_CHAR(p2a_datum, 'MM.YYYY'));
```

Definícia príkazu 85: Globálny index - pre mesiac a rok nehody v tabuľke *nehody_particie*

Posledný index (*def. príkazu 86*), ktorý sme vytvorili, indexuje mesiace z dátumu.

```
CREATE INDEX ind_nehody_particie_datum_mesiac
 ON nehody_particie
 (TO_CHAR(p2a_datum, 'MM'));
```

Definícia príkazu 86: Globálny index - pre mesiac nehody v tabuľke *nehody_particie*

Po opätovnom spustení jednotlivých príkazov sme sledovali zmeny, ktoré optimalizátor vykonal na základe vytvorených globálnych indexov.

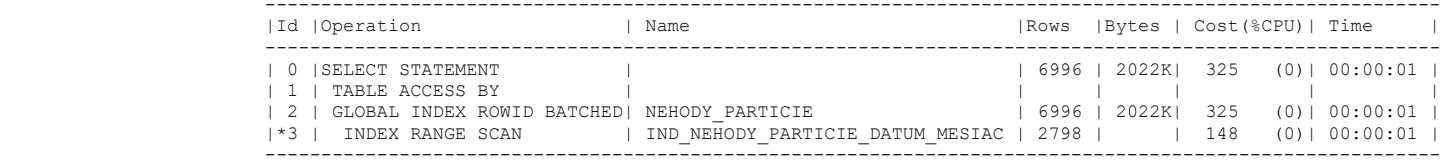
Po spustení príkazu definovaného v *def. príkazu 81* sa plán vykonania nezmenil, optimalizátor nevyužil žiaden nami vytvorený index, pretože neobsahuje podmienku *WHERE,* a preto náklady na vykonanie tohto príkazu ostali nezmenené s hodnotou 1 375.

Po spustení druhého príkazu definovaného v *def. príkazu 82* a zobrazení plánu vykonania zisťujeme, že optimalizátor využil index definovaný nad mesiacom. Plán vykonania tohto príkazu je zobrazený na *Obrázku 35* a náklady na vykonanie tohto príkazu sú 325.

```
-------------------------------------------------------------------------------------------------------
|Id |Operation               |Name                            | Rows |Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT        |                                | 1048 |  306K|   325   (0)| 00:00:01 |
| 1 | TABLE ACCESS BY GLOBAL |                                |      |      |            |          |
| 2 | INDEX ROWID BATCHED    |NEHODY_PARTICIE                 | 1048 |  306K|   325   (0)| 00:00:01 |
|*3 |  INDEX RANGE SCAN      |IND_NEHODY_PARTICIE_DATUM_MESIAC | 2798 |      |   148   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
```

Obrázok 35: Plán vykonania - *Select* mesiac a rok nehody s globálnym indexom pre tabuľku *nehody_particie*

Po spustení príkazu definovaného v *def. príkazu 83* a zobrazení plánu vykonania zisťujeme, že optimalizátor využil opäť index definovaný nad mesiacom. Plán vykonania tohto príkazu je zobrazený na *Obrázku 36* a náklady na vykonanie tohto príkazu sú 325.



```
-----------------------------------------------------------------------------------------------------------
|Id |Operation                  | Name                              |Rows  |Bytes | Cost(%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT           |                                   | 6996 | 2022K|  325   (0)| 00:00:01 |
| 1 | TABLE ACCESS BY           |                                   |      |      |           |          |
| 2 | GLOBAL INDEX ROWID BATCHED| NEHODY_PARTICIE                   | 6996 | 2022K|  325   (0)| 00:00:01 |
|*3 |  INDEX RANGE SCAN         | IND_NEHODY_PARTICIE_DATUM_MESIAC | 2798 |      |  148   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------
```

Obrázok 36: Plán vykonania - *Select* mesiac nehody s globálnym indexom pre tabuľku *nehody_particie*

Výsledky vyššie uvedených plánov vykonania sme sa rozhodli porovnať ešte s výsledkami plánov vykonania pre lokálne indexy. Opäť sme si vytvorili 3 lokálne funkcionálne indexy. Prvý indexuje rok a jeho definícia je nasledovná (*def. príkazu 87*):

```
CREATE INDEX ind_nehody_particie_datum_rok_l
 ON nehody_particie
 (TO_CHAR(p2a_datum, 'YYYY')) LOCAL;
```

Definícia príkazu 87: Lokálny index - pre rok nehody v tabuľke *nehody_particie*

Druhý lokálny index indexuje mesiac a rok. Jeho definícia je na nasledujúcom príkaze (*def. príkazu 88*):

```
CREATE INDEX ind_nehody_particie_datum_mesiac_rok_l
 ON nehody_particie
 (TO_CHAR(p2a_datum, 'MM.YYYY')) LOCAL;
```

Definícia príkazu 88: Lokálny index pre mesiac a rok nehody v tabuľke *nehody_particie*

Tretí lokálny index indexuje mesiac a jeho definícia je zobrazená v *def. príkazu 89*.

```
CREATE INDEX ind_nehody_particie_datum_mesiac_l
 ON nehody_particie
 (TO_CHAR(p2a_datum, 'MM')) LOCAL;
```

Definícia príkazu 89: Lokálny index - pre mesiac nehody v tabuľke *nehody_particie*

Po spustení prvého príkazu *SELECT*, ktorý je definovaný v *def. príkazu 81* a zobrazení plánu vykonania sme zistili, že optimalizátor nevyužil žiadny index, ktorý by dokázal náklady na vykonanie tohto príkazu zlepšiť. Preto náklady ostali nezmenené a plán vykonania pre tento príkaz zobrazuje *Obrázok 32*.

Po spustení druhého príkazu *SELECT* definovaného v *def. príkazu 82* a zobrazení plánu vykonania sme získali zlepšenie nákladov na vykonanie dotazu. Celkové náklady, s využitím lokálneho indexu pre mesiace (*def. príkazu 89*), sa znížili na hodnotu 44. Plán vykonania tohto príkazu je zobrazený na *Obrázku 37*.

```
-----------------------------------------------------------------------------------------------------------
|Id |Operation               | Name                              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT        |                                   |  1048 |   306K|     44   (0)| 00:00:01 |
| 1 | PARTITION RANGE SINGLE |                                   |  1048 |   306K|     44   (0)| 00:00:01 |
| 2 |  TABLE ACCESS BY LOCAL |                                   |       |       |             |          |
| 3 |   INDEX ROWID BATCHED  | NEHODY_PARTICIE                   |  1048 |   306K|     44   (0)| 00:00:01 |
|*4 |    INDEX RANGE SCAN    | IND_NEHODY_PARTICIE_DATUM_MESIAC_L |   419 |       |     17   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------
```

Obrázok 37: Plán vykonania - *Select* mesiac a rok nehody s lokálnym indexom pre tabuľku *nehody_particie*

Po spustení tretieho príkazu definovaného v *def. príkazu 83* a zobrazení plánu vykonania sme získali zlepšenie vykonania dotazu. Celkové náklady, s využitím lokálneho indexu pre mesiace (*def. príkazu 89*), sa znížili na hodnotu 200. Plán vykonania tohto príkazu je zobrazený na *Obrázku 38*.

```
-----------------------------------------------------------------------------------------------------------
|Id |Operation                    | Name                              |Rows|Bytes | Cost(%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT             |                                   |6996| 2022K|  200   (0)| 00:00:01 |
| 1 | PARTITION RANGE ALL         |                                   |6996| 2022K|  200   (0)| 00:00:01 |
| 2 |  TABLE ACCESS BY            |                                   |    |      |           |          |
| 3 |   LOCAL INDEX ROWID BATCHED| NEHODY_PARTICIE                   |6996| 2022K|  200   (0)| 00:00:01 |
|*4 |   INDEX RANGE SCAN          | IND_NEHODY_PARTICIE_DATUM_MESIAC_L |2798|      |   23   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------
```

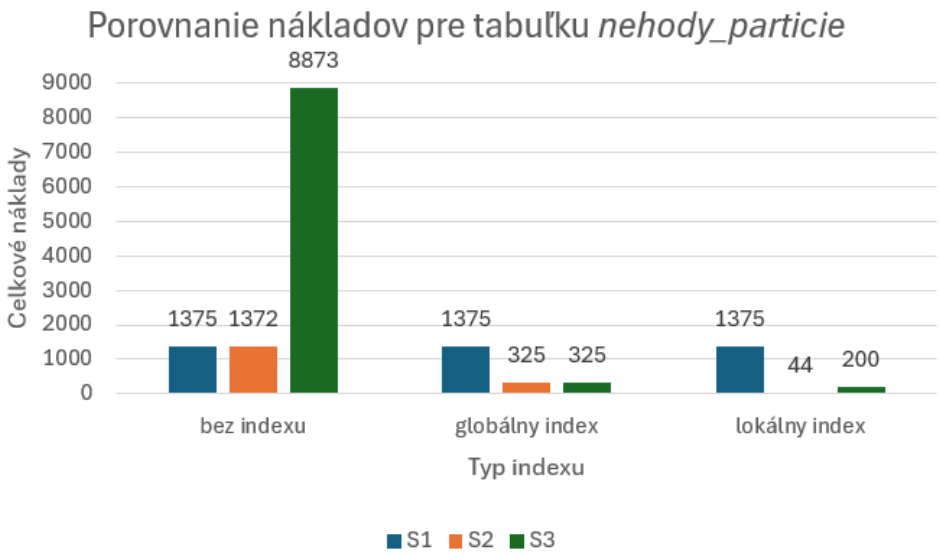
Obrázok 38: Plán vykonania - *Select* mesiac nehody s lokálnym indexom pre tabuľku *nehody_particie*

Zhrnutie jednotlivých nákladov na vykonanie príkazov *SELECT* pre tabuľku *nehody_particie* sa nachádza v *Tabuľke 19*.

Tabuľka 19: Zhrnutie nákladov príkazu *SELECT* pre tabuľku *nehody_particie*

Podmienka	Náklady (bez indexu)	Náklady (globálny index)	Náklady (lokálny index)
Príkaz pre rok	1 375	1 375	1 375
Príkaz pre mesiac a rok	1 372	325	44
Príkaz pre mesiac	8 873	325	200

Grafická reprezentácia porovnania celkových nákladov na vykonanie príkazov sa nachádza na *Obrázku 39*. Celkové náklady predstavujú internú hodnotu, na základe ktorej sa optimalizátor rozhoduje.



Obrázok 39: Stĺpcový graf porovnania nákladov pre tabuľku *nehody_particie*

Tabuľka faktov *nehody_particie_dyn*

Ďalšou porovnávanou tabuľkou je tabuľka faktov s názvom *nehody_particie_dyn*. Táto tabuľka je taktiež rozdelená na partície podľa roku nehody, rozdiel s vyššie uvedenou tabuľkou je v spôsobe vytvárania partícií. Tabuľka *nehody_particie* obsahuje presne definované partície, zatiaľ čo pre tabuľku *nehody_particie_dyn* sa partície vytvárajú dynamicky. Keďže táto tabuľka vytvára partície dynamicky, tak názvy jednotlivých partícií generuje systém. Na zistenie názvov partícií danej tabuľky môžeme použiť systémovú tabuľku *USER_TAB_PARTITIONS* a jej stĺpec s názvom *PARTITION_NAME*. Nasledujúci príkaz (*def. príkazu 90*) zobrazí názvy partícií, hodnoty intervalov partícií a počet záznamov v danej partícii pre tabuľku *nehody_particie_dyn*.

```
SELECT PARTITION_NAME, HIGH_VALUE, NUM_ROWS
 FROM USER_TAB_PARTITIONS
 WHERE TABLE_NAME = 'NEHODY_PARTICIE_DYN'
 ORDER BY PARTITION_NAME;
```

Definícia príkazu 90: Príkaz *Select* na získanie názvov partícií

Def. príkazu 91 zobrazuje kód príkazu *SELECT*, ktorý vyberie záznamy nehôd z tabuľky *nehody_particie_dyn*, ktoré sa udiali v roku 2018. Záznamy sa vyberajú prostredníctvom partície s názvom *SYS_P33987*.

```
SELECT *
 FROM nehody_particie_dyn PARTITION(SYS_P33987);
```

Definícia príkazu 91: *Select* - rok nehody pre tabuľku *nehody_particie_dyn*

Plán vykonania tohto príkazu je znázornený na *Obrázku 40*. Ako je možné vidieť, náklady na vykonanie príkazu boli 1 273.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name                | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                     |  104K|    30M|  1273   (1)| 00:00:01 |
|   1 |  PARTITION RANGE SINGLE|                     |  104K|    30M|  1273   (1)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL    | NEHODY_PARTICIE_DYN |  104K|    30M|  1273   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Obrázok 40: Plán vykonania - *Select* rok nehody pre tabuľku *nehody_particie_dyn*

Nasledujúci príkaz (*def. príkazu 92*) vyberie z tabuľky *nehody_particie_dyn* všetky záznamy nehôd, ktoré sa stali v máji 2018. V definícií príkazu sme použili klauzulu *PARTITION*, v ktorej sme definovali partíciu pre rok 2018 a v podmienke sme definovali mesiac.

```
SELECT *
 FROM nehody_particie_dyn PARTITION(SYS_P33987)
 WHERE TO_CHAR(p2a_datum, 'MM') = '05';
```

Definícia príkazu 92: *Select* – mesiac a rok nehody pre tabuľku *nehody_particie_dyn*

Náklady tohto príkazu sa znížili na hodnotu 1 270, pričom plán vykonania sa nachádza na *Obrázku 41*.

```
-------------------------------------------------------------------------------------------------
| Id | Operation              | Name                | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT       |                     | 1048 |   309K|  1270   (1)| 00:00:01 |
|  1 |  PARTITION RANGE SINGLE|                     | 1048 |   309K|  1270   (1)| 00:00:01 |
|* 2 |   TABLE ACCESS FULL    | NEHODY_PARTICIE_DYN | 1048 |   309K|  1270   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

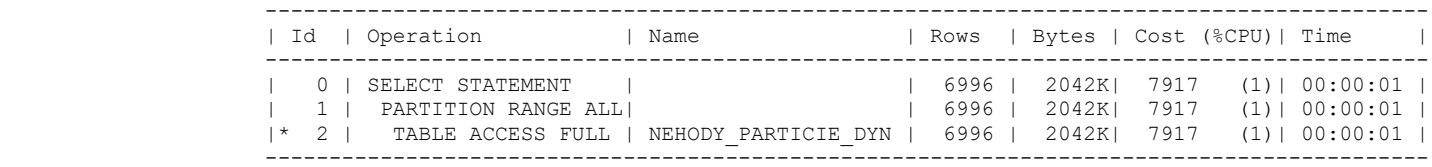
Obrázok 41: Plán vykonania - *Select* mesiac a rok nehody pre tabuľku *nehody_particie_dyn*

Nasledujúci príkaz (*def. príkazu 93*) vyberie z tabuľky *nehody_particie_dyn* všetky záznamy nehôd, ktoré sa stali v máji bez ohľadu na rok. V tomto prípade názov partície nie je možné použiť, pretože potrebujeme vyhľadať záznamy zo všetkých partícií, pre ktoré platí, že mesiac nehody je máj.

```
SELECT *
 FROM nehody_particie_dyn
 WHERE TO_CHAR(p2a_datum, 'MM') = '05';
```

Definícia príkazu 93: *Select* – mesiac nehody pre tabuľku *nehody_particie_dyn*

Plán vykonania tohto príkazu je zobrazený na *Obrázku 42*. Keďže je potrebné prehľadávať celú tabuľku, všetky záznamy v jednotlivých partíciách, tak celkové náklady na tento príkaz sú 7 917. Plán vykonania pre príkaz definovaný v *def. príkazu 93* je zobrazený na *Obrázku 42*.



```
-------------------------------------------------------------------------------------------
| Id  | Operation           | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                     |  6996 |  2042K|  7917   (1)| 00:00:01 |
|   1 |  PARTITION RANGE ALL|                     |  6996 |  2042K|  7917   (1)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL | NEHODY_PARTICIE_DYN |  6996 |  2042K|  7917   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 42: Plán vykonania – *Select* mesiac nehody pre tabuľku *nehody_particie_dyn*

Ako ďalšiu časť experimentov, sme vytvorili globálne indexy pre jednotlivé príkazy. Nasledujúci príkaz (*def. príkazu 94*) vytvorí globálny funkcionálny index pre rok nad tabuľkou *nehody_particie_dyn.*

```
CREATE INDEX ind_nehody_particie_dyn_datum_rok
 ON nehody_particie_dyn
 (TO_CHAR(p2a_datum, 'YYYY'));
```

Definícia príkazu 94: Globálny index - pre rok nehody v tabuľke *nehody_particie_dyn*

Ďalší príkaz (*def. príkazu 95*) vytvorí globálny index, ktorý indexuje mesiace a roky.

```
CREATE INDEX ind_nehody_particie_dyn_datum_mesiac_rok
 ON nehody_particie_dyn
 (TO_CHAR(p2a_datum, 'MM.YYYY'));
```

Definícia príkazu 95: Globálny index - pre mesiac a rok nehody v tabuľke *nehody_particie_dyn*

Posledný index (*def. príkazu 96*), ktorý sme vytvorili, indexuje mesiace z dátumu.

```
CREATE INDEX ind_nehody_particie_dyn_datum_mesiac
 ON nehody_particie_dyn
 (TO_CHAR(p2a_datum, 'MM'));
```

Definícia príkazu 96: Globálny index - pre mesiac nehody v tabuľke *nehody_particie_dyn*

Po opätovnom spustení jednotlivých príkazov sme sledovali zmeny, ktoré optimalizátor vykonal na základe vytvorených globálnych indexov.

Po spustení príkazu definovaného v *def. príkazu 91* sa plán vykonania nezmenil, optimalizátor nevyužil žiaden nami vytvorený index, pretože neobsahuje podmienku *WHERE,* a preto náklady na vykonanie tohto príkazu ostali nezmenené s hodnotou 1 273, ako je zobrazené na *Obrázku 40*.

Po spustení druhého príkazu definovaného v *def. príkazu 92* a zobrazení plánu vykonania zisťujeme, že optimalizátor využil index definovaný nad mesiacom. Plán vykonania príkazu je zobrazený na *Obrázku 43* a náklady na jeho vykonanie sú 325.

```
-------------------------------------------------------------------------------------------------------
|Id |Operation               | Name                                 |Rows  |Bytes |Cost(%CPU)|Time    |
-------------------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT        |                                      | 1048 |  309K|   325  (0)|00:00:01|
| 1 | TABLE ACCESS BY GLOBAL |                                      |      |      |           |        |
| 2 | INDEX ROWID BATCHED    | NEHODY_PARTICIE_DYN                  | 1048 |  309K|   325  (0)|00:00:01|
|*3 |  INDEX RANGE SCAN      | IND_NEHODY_PARTICIE_DYN_DATUM_MESIAC | 2798 |      |   148  (0)|00:00:01|
-------------------------------------------------------------------------------------------------------
```

Obrázok 43: Plán vykonania - *Select* mesiac a rok nehody s globálnym indexom pre tabuľku *nehody_particie_dyn*

Po spustení príkazu definovaného v *def. príkazu 93* a zobrazení plánu vykonania zisťujeme, že optimalizátor využil opäť index definovaný nad mesiacom. Plán vykonania príkazu je zobrazený na *Obrázku 44* a náklady na jeho vykonanie príkazu sú 325.

```
-------------------------------------------------------------------------------------------------------
|Id |Operation               | Name                                 |Rows |Bytes | Cost(%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT        |                                      |6996 | 2042K|   325  (0)| 00:00:01 |
| 1 | TABLE ACCESS BY GLOBAL |                                      |     |      |           |          |
| 2 | INDEX ROWID BATCHED    | NEHODY_PARTICIE_DYN                  |6996 | 2042K|   325  (0)| 00:00:01 |
|*3 |  INDEX RANGE SCAN      | IND_NEHODY_PARTICIE_DYN_DATUM_MESIAC |2798 |      |   148  (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
```

Obrázok 44: Plán vykonania - *Select* mesiac nehody s globálnym indexom pre tabuľku *nehody_particie_dyn*

Rovnako ako globálne indexy, sme vytvorili aj lokálne indexy pre tieto tri možnosti podmienok. Indexy mali rovnakú štruktúru aj podmienku, líšili sa len v tom, že pri lokálnych indexoch sme naviac použili lúčové slovo *LOCAL*.

Po spustení prvého príkazu SELECT (*def. príkazu 91*) sme získali plán vykonania, ktorý však nevyužil žiaden index. Celkové náklady na jeho vykonanie boli 1 273. Plán vykonania je zobrazený na *Obrázku 40.*

Po spustení druhého príkazu (*def. príkazu 92*) a zobrazení plánu vykonania sme zistili, že náklady na vykonanie dorazu dosiahli hodnotu 50. Plán vykonania príkazu s podmienku pre mesiac a rok je zobrazený na *Obrázku 45*.

```
-------------------------------------------------------------------------------------------------------
|Id |Operation               |Name                                  |Rows|Bytes| Cost(%CPU)|Time    |
-------------------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT        |                                      |1048| 309K|   50   (0)|00:00:01|
| 1 | PARTITION RANGE SINGLE |                                      |1048| 309K|   50   (0)|00:00:01|
| 2 |  TABLE ACCESS BY LOCAL |                                      |    |     |           |        |
| 3 |  INDEX ROWID BATCHED   |NEHODY_PARTICIE_DYN                   |1048| 309K|   50   (0)|00:00:01|
|*4 |   INDEX RANGE SCAN     |IND_NEHODY_PARTICIE_DYN_DATUM_MESIAC_L | 419|     |   23   (0)|00:00:01|
-------------------------------------------------------------------------------------------------------
```

Obrázok 45: Plán vykonania - *Select* mesiac a rok nehody s lokálnym indexom pre tabuľku *nehody_particie_dyn*

Po spustení tretieho príkazu (*def. príkazu 93*) a zobrazení plánu vykonania sme zistili, že náklady na vykonanie dotazu dosiahli hodnotu 200. Plán vykonania príkazu s podmienku pre mesiac a rok je zobrazený na *Obrázku 46.*

```
-------------------------------------------------------------------------------------------------------------
|Id |Operation               | Name                                |Rows |Bytes | Cost(%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT        |                                     |6996 | 2042K|   200  (0)| 00:00:01|
| 1 | PARTITION RANGE ALL    |                                     |6996 | 2042K|   200  (0)| 00:00:01|
| 2 |  TABLE ACCESS BY LOCAL |                                     |     |      |           |         |
| 2 |  INDEX ROWID BATCHED   | NEHODY_PARTICIE_DYN                 |6996 | 2042K|   200  (0)| 00:00:01|
|*3 |   INDEX RANGE SCAN     | IND_NEHODY_PARTICIE_DYN_DATUM_MESIAC_L |2798 |      |    23  (0)| 00:00:01|
-------------------------------------------------------------------------------------------------------------
```

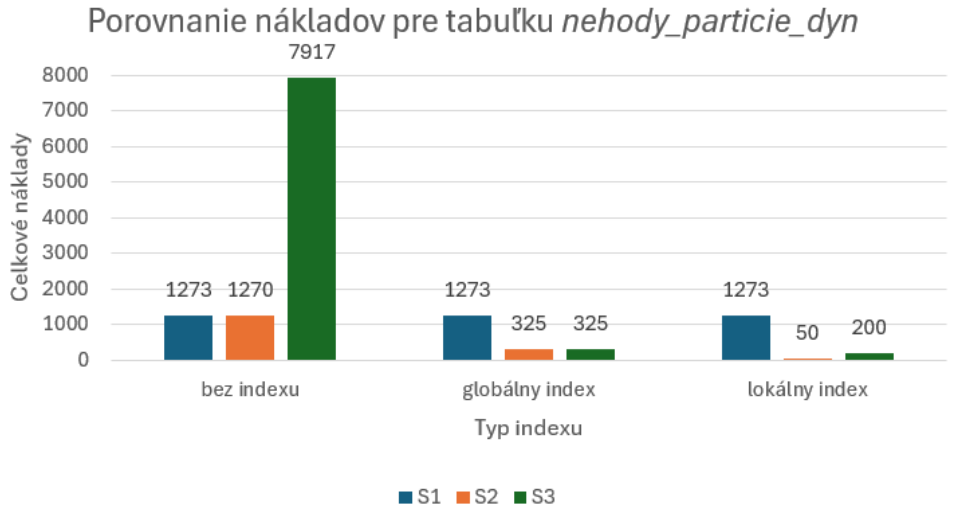
Obrázok 46: Plán vykonania - *Select* mesiac nehody s lokálnym indexom pre tabuľku *nehody_particie_dyn*

Zhrnutie jednotlivých nákladov na vykonanie príkazov *SELECT* pre tabuľku *nehody_particie_dyn* sa nachádza v *Tabuľke 20*.

Tabuľka 20: Zhrnutie nákladov príkazu *SELECT* pre tabuľku *nehody_particie_dyn*

Podmienka	Náklady (bez indexu)	Náklady (globálny index)	Náklady (lokálny index)
Príkaz pre rok	1 273	1 273	1 273
Príkaz pre mesiac a rok	1 270	325	50
Príkaz pre mesiac	7 917	325	200

Grafická reprezentácia porovnania celkových nákladov na vykonanie príkazov sa nachádza na *Obrázku 47*. Celkové náklady predstavujú internú hodnotu, na základe ktorej sa optimalizátor rozhoduje.



Obrázok 47: Stĺpcový graf porovnania nákladov pre tabuľku *nehody_particie_dyn*

Tabuľka faktov *nehody_subparticie*

Ďalšou tabuľkou, nad ktorou sledujeme zmeny v pláne vykonania pre vyhľadávanie záznamov podľa definovaných podmienok, je tabuľka, ktorá je okrem partícií podľa rokov rozdelená aj na subpartície podľa mesiacov. Opäť je možné získať názvy partícií a subpartícií zo systémovej tabuľky.

Def. príkazu 97 obsahuje kód príkazu *SELECT*, ktorý vyberie záznamy nehôd z tabuľky *nehody_subparticie*, ktoré sa udiali v roku 2018. Názov partície, z ktorej čerpáme záznamy je *SYS_P33920*.

```
SELECT *
 FROM nehody_subparticie PARTITION(SYS_P33920);
```

Definícia príkazu 97: *Select* - rok nehody pre tabuľku *nehody_subparticie*

Plán vykonania tohto príkazu je znázornený na *Obrázku 48*. Ako je možné vidieť, v rámci partície bolo potrebné prehľadávať aj všetky subpartície po mesiacoch. Preto náklady na vykonanie tohto príkazu boli 3 282.

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost(%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |  104K |  30M  | 3282   (1)| 00:00:01 |
|   1 |  PARTITION RANGE SINGLE|                   |  104K |  30M  | 3282   (1)| 00:00:01 |
|   2 |   PARTITION LIST ALL  |                    |  104K |  30M  | 3282   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL  | NEHODY_SUBPARTICIE |  104K |  30M  | 3282   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 48: Plán vykonania - *Select* rok nehody pre tabuľku *nehody_subparticie*

Nasledujúci príkaz (*def. príkazu 98*) vyberie z tabuľky *nehody_subparticie* všetky záznamy nehôd, ktoré sa stali v máji 2018. Tieto záznamy sú uložené v subpartícii s názvom *SYS_SUBP33912*.

```
SELECT *
 FROM nehody_subparticie SUBPARTITION(SYS_SUBP33912);
```

Definícia príkazu 98: *Select* – mesiac a rok nehody pre tabuľku *nehody_subparticie*

Plán vykonania tohto príkazu je zobrazený na *Obrázku 49*. Ako je možné vidieť, celkové náklady na vykonanie príkazu sa znížili na hodnotu 275, pretože sme pristúpili priamo k záznamom, ktoré sme chceli vyhľadávať.

```
-------------------------------------------------------------------------------------------
|Id |Operation                  |Name              | Rows  | Bytes | Cost(%CPU)| Time     |
-------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT           |                  |  9444 |  2822K|  275   (1)| 00:00:01 |
| 1 | PARTITION COMBINED ITERATOR|                  |  9444 |  2822K|  275   (1)| 00:00:01 |
|*2 |  TABLE ACCESS FULL        |NEHODY_SUBPARTICIE|  9444 |  2822K|  275   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 49: Plán vykonania - *Select* mesiac a rok nehody pre tabuľku *nehody_subparticie*

Nasledujúci príkaz (*def. príkazu 99*) vyberie z tabuľky *nehody_subparticie* všetky záznamy nehôd, ktoré sa stali v máji bez ohľadu na rok.

```
SELECT *
 FROM nehody_subparticie
 WHERE TO_CHAR(p2a_datum, 'MM') = '05';
```

Definícia príkazu 99: *Select* – mesiac nehody pre tabuľku *nehody_subparticie*

Plán vykonania tohto príkazu je zobrazený na *Obrázku 50*. Ako je možné vidieť, keďže vyhľadávame len mesiac, je potrebné prechádzať všetkými partíciami a subpartíciami. Preto náklady na vykonanie tohto príkazu sú 17 648.

```
-------------------------------------------------------------------------------------------
| Id  | Operation           | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                    |  6996 |  2063K| 17648   (1)| 00:00:01 |
|   1 |  PARTITION RANGE ALL|                    |  6996 |  2063K| 17648   (1)| 00:00:01 |
|   2 |   PARTITION LIST ALL|                    |  6996 |  2063K| 17648   (1)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL| NEHODY_SUBPARTICIE |  6996 |  2063K| 17648   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 50: Plán vykonania - *Select* mesiac nehody pre tabuľku *nehody_subparticie*

Ako ďalšiu časť experimentov, sme vytvorili globálne indexy pre jednotlivé príkazy. Nasledujúci príkaz (*def. príkazu 100*) vytvorí globálny funkcionálny index pre rok nad tabuľkou *nehody_particie_dyn*.

```
CREATE INDEX ind_nehody_subparticie_rok
 ON nehody_subparticie
 (TO_CHAR(p2a_datum, 'YYYY'));
```

Definícia príkazu 100: Globálny index - pre rok nehody v tabuľke *nehody_subparticie*

Ďalší príkaz (*def. príkazu 101*) vytvorí index, ktorý indexuje mesiace a roky.

```
CREATE INDEX ind_nehody_subparticie_mesiac_rok
 ON nehody_subparticie
 (TO_CHAR(p2a_datum, 'MM.YYYY'));
```

Definícia príkazu 101: Globálny index - pre mesiac a rok nehody v tabuľke *nehody_subparticie*

Posledný index (*def. príkazu 102*), ktorý sme vytvorili, indexuje mesiace z dátumu.

```
CREATE INDEX ind_nehody_subparticie_mesiac
 ON nehody_subparticie
 (TO_CHAR(p2a_datum, 'MM'));
```

Definícia príkazu 102: Globálny index - pre mesiac nehody v tabuľke *nehody_subparticie*

Po opätovnom spustení jednotlivých príkazov sme sledovali zmeny, ktoré optimalizátor vykonal na základe vytvorených globálnych indexov.

Po spustení príkazu definovaného v *def. príkazu 97* sa plán vykonania nezmenil, optimalizátor nevyužil žiaden nami vytvorený index, pretože neobsahuje podmienku

WHERE, a preto náklady na vykonanie tohto príkazu ostali nezmenené s hodnotou 3 282, ako je zobrazené na *Obrázku 48*.

Po spustení druhého príkazu *SELECT* definovaného v *def. príkazu 98* a zobrazení plánu vykonania zisťujeme, že optimalizátor nevyužil žiaden nami vytvorený index, opäť preto, lebo neobsahuje žiadnu podmienku *WHERE*. Preto sa náklady na vykonanie tohto príkazu nezmenili a nadobúdajú hodnotu 275 ako je na *Obrázku 49*.

Po spustení príkazu definovaného v *def. príkazu 99* a zobrazení plánu vykonania zisťujeme, že optimalizátor využil index definovaný nad mesiacom. Plán vykonania tohto príkazu je zobrazený na *Obrázku 51* a náklady na jeho vykonanie sú 262.

```
-------------------------------------------------------------------------------------------------------
| Id |Operation               | Name                          |Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|  0 |SELECT STATEMENT        |                               |6996 |  2063K|  262    (1)| 00:00:01 |
|  1 | TABLE ACCESS BY GLOBAL |                               |     |       |            |          |
|  2 | INDEX ROWID BATCHED    | NEHODY_SUBPARTICIE            |6996 |  2063K|  262    (1)| 00:00:01 |
|* 3 |  INDEX RANGE SCAN      | IND_NEHODY_SUBPARTICIE_MESIAC |2798 |       |  148    (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
```

Obrázok 51: Plán vykonania - *Select* mesiac nehody s globálnym indexom pre tabuľku *nehody_subparticie*

Rovnako ako globálne indexy, sme vytvorili aj lokálne indexy pre tieto tri možnosti podmienok. Indexy mali rovnakú štruktúru aj podmienku, líšili sa len v tom, že pri lokálnych indexoch sme naviac použili lúčové slovo *LOCAL*.

Po spustení prvého príkazu (*def. príkazu 97*) sme získali plán vykonania, ktorý však nevyužil žiaden index, celkové náklady na jeho vykonanie boli 3 282. Plán vykonania je zobrazený na *Obrázku 48.*

Po spustení druhého príkazu (*def. príkazu 98*) a zobrazení plánu vykonania sme zistili, že náklady na vykonanie dorazu dosiahli hodnotu 275, a teda nedošlo k zlepšeniu. Plán vykonania tohto príkazu je zobrazený na *Obrázku 49.*

Po spustení tretieho príkazu (*def. príkazu 99*) a zobrazení plánu vykonania sme zistili, že náklady na vykonanie dorazu dosiahli hodnotu 226. Plán vykonania príkazu s podmienku pre mesiac je zobrazený na *Obrázku 52.*

```
-------------------------------------------------------------------------------------------------------
|Id |Operation                | Name                           | Rows |Bytes | Cost(%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
| 0 |SELECT STATEMENT         |                                |6996 |  2063K|  226   (1)| 00:00:01|
| 1 | PARTITION RANGE ALL     |                                |6996 |  2063K|  226   (1)| 00:00:01|
| 2 |  PARTITION LIST ALL     |                                |6996 |  2063K|  226   (1)| 00:00:01|
| 3 |   TABLE ACCESS BY LOCAL |                                |     |       |           |         |
| 4 |   INDEX ROWID BATCHED   | NEHODY_SUBPARTICIE             |6996 |  2063K|  226   (1)| 00:00:01|
|*5 |    INDEX RANGE SCAN     | IND_NEHODY_SUBPARTICIE_MESIAC_L |2798 |       |   112   (1)| 00:00:01|
-------------------------------------------------------------------------------------------------------
```

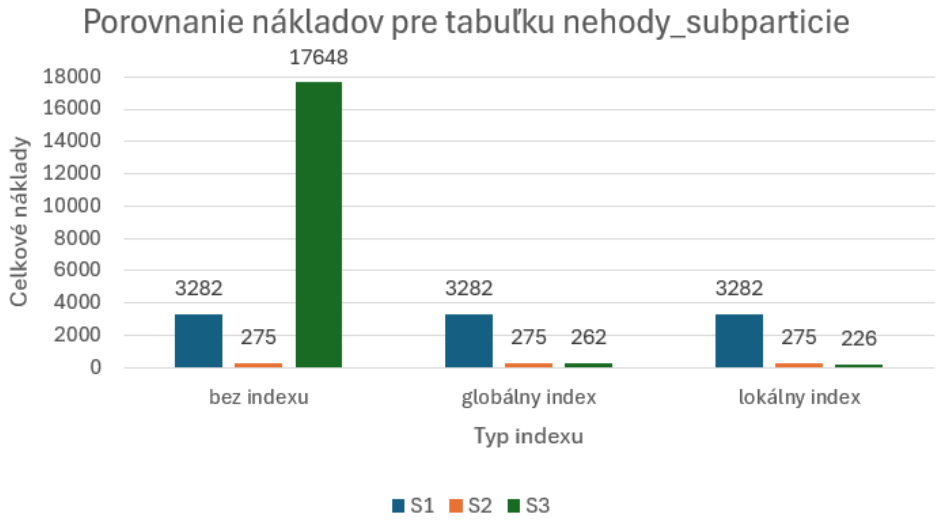
Obrázok 52: Plán vykonania - *Select* mesiac nehody s lokálnym indexom pre tabuľku *nehody_subparticie*

Zhrnutie jednotlivých nákladov na vykonanie vyššie uvedených príkazov sa nachádza v *Tabuľke 21*.

Tabuľka 21: Zhrnutie nákladov príkazu *SELECT* pre tabuľku *nehody_subparticie*

Podmienka	Náklady (bez indexu)	Náklady (globálny index)	Náklady (lokálny index)
Príkaz pre rok	3 282	3 282	3 282
Príkaz pre mesiac a rok	275	275	275
Príkaz pre mesiac	17 648	262	226

Grafická reprezentácia porovnania celkových nákladov na vykonanie príkazov sa nachádza na *Obrázku 53*. Celkové náklady predstavujú internú hodnotu, na základe ktorej sa optimalizátor rozhoduje.



Obrázok 53: Stĺpcový graf porovnania nákladov pre tabuľku *nehody_subparticie*

Tabuľka objektov typu XMLTYPE *nehody_xml_tab_1*

Ďalšou tabuľkou, nad ktorou sledujeme zmeny v pláne vykonania pre vyhľadávanie záznamov podľa definovaných podmienok, je tabuľka objektového typu XMLTYPE s názvom *nehody_xml_tab_1*. Táto tabuľka obsahuje jeden stĺpec, tzv. OBJECT_TYPE, ktorý uchováva XML dokumenty. XML dokumenty v tejto tabuľke majú vybraný stĺpec *p2a_datum* ako vnútorný element XML dokumentu.

Ako aj v predchádzajúcich príkladoch, aj tu sme sledovali plán vykonania príkazov *SELECT* najskôr bez indexov, a potom s vytvorenými indexami. Príkaz *SELECT* sa mierne líšil v spôsobe, akým vyberáme mesiace a roky z elementu *dátum*. To sme dosiahli s použitím funkcie *extractValue(),* vďaka ktorej sme získali hodnotu dátumu na základe definovanej cesty k tomuto elementu. Pomocou funkcie TO_DATE() sme vytvorili dátum z tohto reťazca a následne sme vybrali požadovanú hodnotu (mesiac, rok).

Def. príkazu 103 obsahuje kód príkazu *SELECT*, ktorý vyberie záznamy nehôd z tabuľky objektov *nehody_xml_tab_1*, ktoré sa udiali v roku 2018.

```
SELECT *
 FROM nehody_xml_tab_1 nehoda
 WHERE TO_CHAR(TO_DATE(
           ExtractValue(VALUE(nehoda), '/nehoda/p2a_datum'),
           'YYYY-MM-DD'), 'YYYY') = '2018';
```

Definícia príkazu 103: *Select* - rok nehody pre tabuľku *nehody_xml_tab_1*

Plán vykonania tohto príkazu bez použitia indexov je znázornený na *Obrázku 54*. Ako je možné vidieť, bolo potrebné prehľadávať celú tabuľku, a preto náklady na vykonanie tohto príkazu boli 16 940.

```
-------------------------------------------------------------------------------------------
|  Id  |  Operation          | Name             |  Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|    0 |  SELECT STATEMENT   |                  |   6996 |  4481K| 16940  (11)| 00:00:01 |
|*   1 |   TABLE ACCESS FULL | NEHODY_XML_TAB_1 |   6996 |  4481K| 16940  (11)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 54: Plán vykonania - *Select* rok nehody pre tabuľku *nehody_xml_tab_1*

Nasledujúci príkaz (*def. príkazu 104*) vyberie z tabuľky *nehody_xml_tab_1* všetky záznamy nehôd, ktoré sa stali v máji 2018.

```
SELECT *
 FROM nehody_xml_tab_1 nehoda
 WHERE TO_CHAR(TO_DATE(
           ExtractValue(VALUE(nehoda), '/nehoda/p2a_datum'),
           'YYYY-MM-DD'), 'MM.YYYY') = '05.2018';
```

Definícia príkazu 104: : *Select* – mesiac a rok nehody pre tabuľku *nehody_xml_tab_1*

Plán vykonania tohto príkazu je zobrazený na *Obrázku 54*. Ako je možné vidieť, celkové náklady na vykonanie tohto príkazu ostali nezmenené, pretože sa opäť musela prehľadávať celá tabuľka.

Nasledujúci príkaz (*def. príkazu 105*) vyberie z tabuľky *nehody_xml_tab_1* všetky záznamy nehôd, ktoré sa stali v máji bez ohľadu na rok.

```
SELECT *
 FROM nehody_xml_tab_1 nehoda
 WHERE TO_CHAR(TO_DATE(
          ExtractValue(VALUE(nehoda), '/nehoda/p2a_datum'),
          'YYYY-MM-DD'), 'MM') = '05';
```

Definícia príkazu 105: *Select* – mesiac nehody pre tabuľku *nehody_xml_tab_1*

Plán vykonania tohto príkazu je zobrazený na *Obrázku 54*, pretože rovnako ako v predošlom príklade, ani tu sa náklady na vykonanie príkazu nezmenili, keďže bolo potrebné prehľadávať celú tabuľku.

Výkonnosť týchto príkazov *SELECT* sme sa snažili zlepšiť vytvorením vhodných indexov, ktoré by to zabezpečili. Najskôr sme vytvorili index (*def. príkazu 106*), ktorý indexoval rok z elementu dátumu. Je to funkcionálny index, ktorý pomocou funkcie *ExtractValue()* vyberie dátum z premennej *OBJECT_VALUE* vo forme reťazca, a ďalšími funkciami vyberieme rok.

```
CREATE INDEX ind_nehody_xml_tab_1_rok
 ON nehody_xml_tab_1
 (TO_CHAR(TO_DATE(
     ExtractValue(OBJECT_VALUE,'/nehoda/p2a_datum'),
     'YYYY-MM-DD'),'YYYY'));
```

Definícia príkazu 106: Index - pre rok nehody v tabuľke *nehody_xml_tab_1*

Po spustení príkazu *SELECT* s podmienkou pre rok a definovaným indexom sme zobrazili plán vykonania, ktorého náklady sa znížili na hodnotu 8 559, keďže optimalizátor využil vytvorený index. Plán vykonania je zobrazený na *Obrázku 55*.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation           | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                         |  6996 |  4481K|  8559   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY    |                         |       |       |            |          |
|   2 |  INDEX ROWID BATCHED| NEHODY_XML_TAB_1        |  6996 |  4481K|  8559   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN  | IND_NEHODY_XML_TAB_1_ROK |   104K|       |   237   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Obrázok 55: Plán vykonania - *Select* rok nehody s indexom pre tabuľku *nehody_xml_tab_1*

Druhý index, ktorý sme vytvorili indexoval mesiac a rok nehody. Kód pre tento index je zobrazený v *def. príkazu 107*.

```
CREATE INDEX ind_nehody_xml_tab_1_mesiac_rok
 ON nehody_xml_tab_1
 (TO_CHAR(TO_DATE(
      ExtractValue(OBJECT_VALUE,'/nehoda/p2a_datum'),
      'YYYY-MM-DD'),'MM.YYYY'));
```

Definícia príkazu 107: Index - pre mesiac a rok nehody v tabuľke *nehody_xml_tab_1*

Po spustení príkazu *SELECT* s podmienkou pre mesiac a rok a definovaným indexom, sa náklady na vykonanie tohto príkazu znížili na hodnotu 1 308, ako je možné vidieť na *Obrázku 56*.

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                               |  6996 |  4481K|  1308   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                               |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_TAB_1              |  6996 |  4481K|  1308   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_TAB_1_MESIAC_ROK |  9444 |       |    27   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

Obrázok 56: Plán vykonania - *Select* mesiac a rok nehody s indexom pre tabuľku *nehody_xml_tab_1*

Posledný index, ktorý sme vytvorili indexoval mesiace z dátumu. *Def. príkazu 108* zobrazuje kód tohto indexu.

```
CREATE INDEX ind_nehody_xml_tab_1_mesiac
 ON nehody_xml_tab_1
 (TO_CHAR(TO_DATE(
      ExtractValue(OBJECT_VALUE,'/nehoda/p2a_datum'),
      'YYYY-MM-DD'),'MM'));
```

Definícia príkazu 108: Index - pre mesiac nehody v tabuľke *nehody_xml_tab_1*

Po spustení tretieho príkazu *SELECT* s definovaným indexom a po zobrazení plánu vykonania sme zistili, že náklady klesli na hodnotu 8 663. Plán vykonania je zobrazený na *Obrázku 57*.

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation           | Name                         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |                              |  6996 |  4481K|  8663   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY    |                              |       |       |            |          |
|   2 |  INDEX ROWID BATCHED| NEHODY_XML_TAB_1             |  6996 |  4481K|  8663   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN  | IND_NEHODY_XML_TAB_1_MESIAC | 62977 |       |   126   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

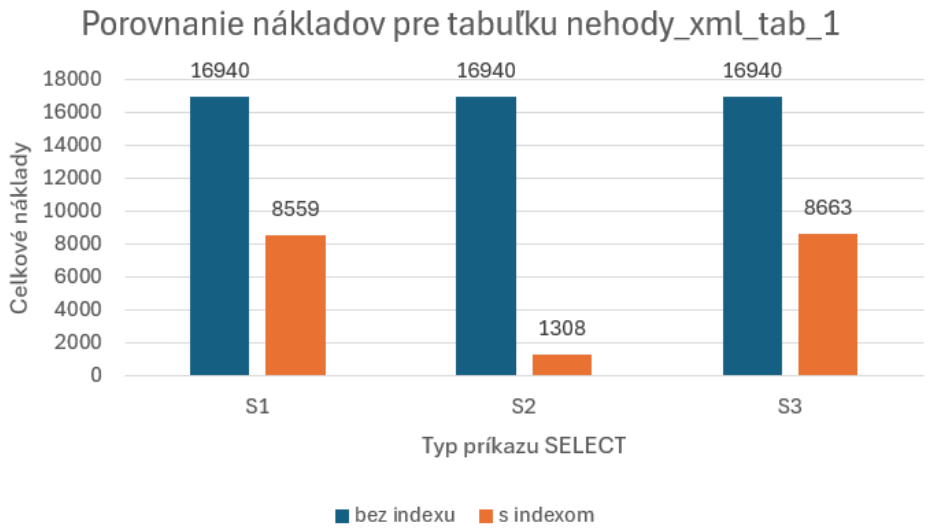
Obrázok 57: Plán vykonania - *Select* mesiac nehody s indexom pre tabuľku *nehody_xml_tab_1*

Zhrnutie jednotlivých nákladov na vykonanie vyššie uvedených príkazov pre tabuľku objektov *nehody_xml_tab_1*, kde je dátum nehody ako vnútorný element XML dokumentu sa nachádza v *Tabuľke 22*.

Tabuľka 22: Zhrnutie nákladov príkazu *SELECT* pre tabuľku *nehody_xml_tab_1*

Podmienka	Náklady (bez indexu)	Náklady (s indexom)
TO_CHAR('YYYY')	16 940	8 559
TO_CHAR('MM.YYYY')	16 940	1 308
TO_CHAR('MM')	16 940	8 663

Grafická reprezentácia porovnania celkových nákladov na vykonanie príkazov sa nachádza na *Obrázku 58*. Celkové náklady predstavujú internú hodnotu, na základe ktorej sa optimalizátor rozhoduje.



Obrázok 58: Stĺpcový graf porovnania nákladov pre tabuľku nehody_xml_tab_1

Tabuľka objektov typu XMLTYPE *nehody_xml_tab_2*

Rovnaké experimenty sme vykonali aj nad druhou tabuľkou objektov *nehody_xml_tab_2*, v ktorej sa ale dátum nehody nachádza ako atribút elementu. Týmto experimentom sme chceli zistiť aký vplyv na celkové náklady vykonania príkazov má umiestnenie vyhľadávanej hodnoty v XML dokumente.

Príkazy *SELECT* vyzerali úplne rovnako líšili sa len v ceste, ktorú zadávame vo funkcii *EctractValue()*, pretože pristupujeme k atribútu. Príkaz *SELECT* s podmienkou výberu záznamov z roku 2018 zobrazuje *def. príkazu 109*.

```
SELECT *
 FROM nehody_xml_tab_2 nehoda
 WHERE TO_CHAR(TO_DATE(
      ExtractValue(VALUE(nehoda), '/nehoda/@p2a_datum'),
      'YYYY-MM-DD'), 'YYYY') = '2018';
```

Definícia príkazu 109: *Select* – rok nehody pre tabuľku *nehody_xml_tab_2*

Plán vykonania pre vyššie uvedený príkaz bez indexov je zobrazený na *Obrázku 59*. Ako je možné vidieť, celkové náklady dosiahli hodnotu 16 940.

```
-------------------------------------------------------------------------------------
| Id  | Operation         | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                  |  6996 |  4263K| 16940  (11)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| NEHODY_XML_TAB_2 |  6996 |  4263K| 16940  (11)| 00:00:01 |
-------------------------------------------------------------------------------------
```

Obrázok 59: Plán vykonania - *Select* rok nehody pre tabuľku *nehody_xml_tab_2*

Následne sme spustili druhý príkaz, ktorý obsahuje podmienku pre mesiac a rok nehody. K hodnote z dátumu sme sa dostali pomocou nasledujúcej cesty (*def. príkazu 110*):

```
/nehoda/@p2a_datum
```

Definícia príkazu 110: Definovanie cesty k dátumu ako atribútu v XML dokumente

Plán vykonania tohto príkazu opäť nadobúdal hodnotu ako predošlý príkaz, pretože bolo potrebné prehľadávať celú tabuľku.

Posledný príkaz *SELECT* s definovanou podmienkou pre mesiac nehody bez ohľadu na rok a bez vytvorených indexov sme taktiež spustili. Plán vykonania pre daný príkaz sa vôbec nezmenil a celkové náklady na jeho vykonanie nadobudli hodnotu 16 940.

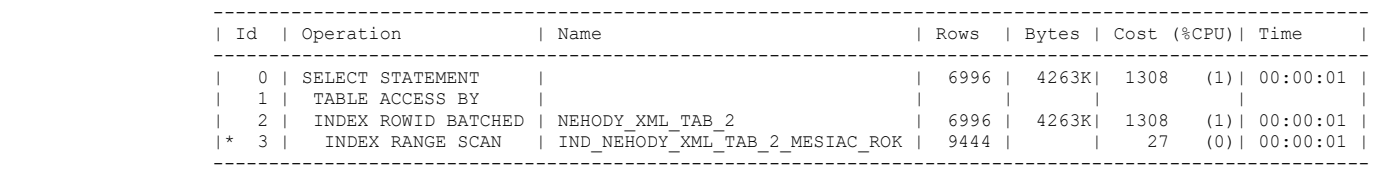
Následne sme vytvorili indexy. Prvý index indexoval roky dátumu. Jeho definícia je rovnaká ako *def. príkazu 106*, avšak pristupujeme k atribútu, takže sme museli využiť cestu definovanú v *def. príkazu 110*. Po spustení príkazu sme dostali plán vykonania, ktorý je možné vidieť na *Obrázku 60*.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation            | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                         |  6996 |  4263K|  8559   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                         |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_TAB_2        |  6996 |  4263K|  8559   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_TAB_2_ROK |   104K|       |   237   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

Obrázok 60: Plán vykonania - *Select* rok nehody s indexom pre tabuľku *nehody_xml_tab_2*

Druhý index indexoval mesiace a roky nehôd. Jeho definícia je podobná ako na *def. príkazu 107*, líši sa len v ceste k atribútu. Po spustení príkazu a zobrazení plánu vykonania

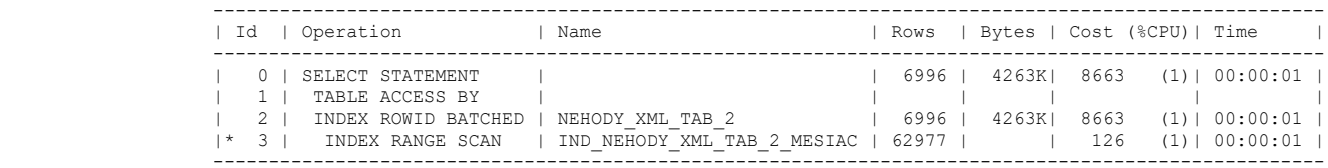
sa ukázalo, že náklady na vykonanie tohto príkazu nadobudli hodnotu 1 308. Plán vykonania je zobrazený na *Obrázku 61*.



```
-----------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                               |  6996 |  4263K|  1308   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                               |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_TAB_2              |  6996 |  4263K|  1308   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_TAB_2_MESIAC_ROK |  9444 |       |    27   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

Obrázok 61: Plán vykonania - *Select* mesiac a rok nehody s indexom pre tabuľku *nehody_xml_tab_2*

Tretí index indexoval mesiace, pričom jeho definícia sa oproti *def. príkazu 108* líšila len v ceste k hodnote dátumu. Po spustení príkazu a zobrazení plánu vykonania sa ukázalo, že náklady na vykonanie tohto príkazu nadobudli hodnotu 8 663. Plán vykonania je zobrazený na *Obrázku 62*.



```
-----------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                               |  6996 |  4263K|  8663   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                               |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_TAB_2              |  6996 |  4263K|  8663   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_TAB_2_MESIAC   | 62977 |       |   126   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

Obrázok 62: Plán vykonania - *Select* mesiac nehody s indexom pre tabuľku *nehody_xml_tab_2*

Zhrnutie jednotlivých nákladov na vykonanie vyššie uvedených príkazov pre tabuľku objektov *nehody_xml_tab_2*, kde je dátum nehody ako atribút elementu v XML dokumente sa nachádza v *Tabuľke 23*.

Tabuľka 23: Zhrnutie nákladov príkazu *SELECT* pre tabuľku *nehody_xml_tab_2*

Podmienka	Náklady (bez indexu)	Náklady (s indexom)
TO_CHAR('YYYY')	16 940	8 559
TO_CHAR('MM.YYYY')	16 940	1 308
TO_CHAR('MM')	16 940	8 663

Ako je možné vidieť na experimentoch pre tabuľky *nehody_xml_tab_1* a *nehody_xml_tab_2*, či je dátum ako vnútorný element alebo ako atribút elementu XML dokumentu, náklady na vykonanie príkazov sa nelíšili.

Tabuľka *nehody_xml_atr_1*

Tabuľka *nehody_xml_atr_1* je databázová tabuľka, ktorá uchováva XML dokumenty, ktoré sú atribútom tabuľky. Okrem tohto atribútu má ešte jeden atribút a to identifikačné číslo XML dokumentu. V XML dokumente sa nachádza dátum ako vnútorný element dokumentu.

Jednotlivé príkazy *SELECT* sa medzi sebou líšia len podmienkou. V prvom rade sme spustili príkazy bez indexov. Ako prvý sme spustili ten, ktorý vyberá všetky záznamy nehôd z tabuľky *nehody_xml_atr_1*, ktorých rok je 2018. *Def. príkazu 111* zobrazuje daný príkaz.

```
SELECT *
 FROM nehody_xml_atr_1
 WHERE TO_CHAR(TO_DATE(
      ExtractValue(xml_data, '/nehoda/p2a_datum'),
      'YYYY-MM-DD'), 'YYYY') = '2018';
```

Definícia príkazu 111: *Select* - rok nehody pre tabuľku *nehody_xml_atr_1*

Plán vykonania vyššie uvedeného príkazu je zobrazený na *Obrázku 63*. Ako je možné vidieť, celkové náklady na vykonanie tohto príkazu sú 16 662.

```
-------------------------------------------------------------------------------------
| Id  | Operation         | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                  |  6996 |  4297K| 16662  (11)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| NEHODY_XML_ATR_1 |  6996 |  4297K| 16662  (11)| 00:00:01 |
-------------------------------------------------------------------------------------
```

Obrázok 63: Plán vykonania - *Select* rok nehody pre tabuľku *nehody_xml_atr_1*

Nasledujúci príkaz (*def. príkazu 112*) vyberie z tabuľky *nehody_xml_atr_1* všetky záznamy nehôd, ktoré sa stali v máji 2018.

```
SELECT *
 FROM nehody_xml_atr_1
 WHERE TO_CHAR(TO_DATE(
      ExtractValue(xml_data, '/nehoda/p2a_datum'),
      'YYYY-MM-DD'), 'MM.YYYY') = '05.2018';
```

Definícia príkazu 112: *Select* – mesiac a rok nehody pre tabuľku *nehody_xml_atr_1*

Plán vykonania druhého príkazu je rovnaký ako na *Obrázoku 63*. Keďže sa musela prehľadávať celá tabuľka, náklady na vykonanie príkazu ostali nezmenené.

Nasledujúci príkaz (*def. príkazu 113*) vyberie z tabuľky *nehody_xml_atr_1* všetky záznamy nehôd, ktoré sa stali v máji bez ohľadu na rok.

```
SELECT *
 FROM nehody_xml_atr_1
 WHERE TO_CHAR(TO_DATE(
      ExtractValue(xml_data, '/nehoda/p2a_datum'),
      'YYYY-MM-DD'), 'MM') = '05';
```

Definícia príkazu 113: *Select* – mesiac nehody pre tabuľku *nehody_xml_atr_1*

Plán vykonania vyššie uvedeného príkazu opäť zostal nezmenený a nadobúda hodnotu 16 662, rovnako ako na *Obrázku 63*.

Následne sme pre jednotlivé príkazy vytvorili vhodné indexy, ktoré by zefektívnili plány vykonania. Ako prvý sme vytvorili index, ktorý indexuje roky. *Def. príkazu 114* zobrazuje kód tohto indexu.

```
CREATE INDEX ind_nehody_xml_atr_1_rok
 ON nehody_xml_atr_1
 (TO_CHAR(TO_DATE(
     ExtractValue(xml_data,'/nehoda/p2a_datum'),
     'YYYY-MM-DD'),'YYYY'));
```

Definícia príkazu 114: Index - pre rok nehody v tabuľke *nehody_xml_atr_1*

Po spustení príkazu s podmienkou pre rok a definovaným indexom sme zobrazili plán vykonania, ktorého náklady sa znížili na hodnotu 8 408, keďže optimalizátor využil vytvorený index. Plán vykonania je zobrazený na *Obrázku 64*.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                         |  6996 |  4297K|  8408   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                         |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_ATR_1        |  6996 |  4297K|  8408   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_ATR_1_ROK |   104K|       |   237   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

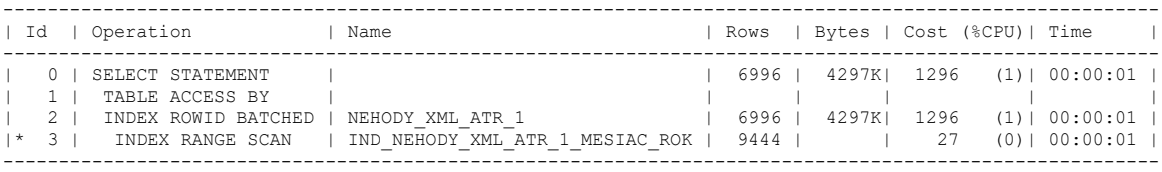
Obrázok 64: Plán vykonania - *Select* rok nehody s indexom pre tabuľku *nehody_xml_atr_1*

Druhý index, ktorý sme vytvorili indexoval mesiac a rok nehody. Kód pre tento index je zobrazený v *def. príkazu 115*.

```
CREATE INDEX ind_nehody_xml_atr_1_mesiac_rok
 ON nehody_xml_atr_1
 (TO_CHAR(TO_DATE(
     ExtractValue(xml_data,'/nehoda/p2a_datum'),
     'YYYY-MM-DD'),'MM.YYYY'));
```

Definícia príkazu 115: Index - pre mesiac a rok nehody v tabuľke *nehody_xml_atr_1*

Po spustení druhého príkazu s podmienkou pre mesiac a rok a definovaným indexom sa náklady na vykonanie tohto príkazu znížili na hodnotu 1 296, ako je možné vidieť na *Obrázku 65*.



```
-----------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                               |  6996 |  4297K|  1296   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                               |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_ATR_1              |  6996 |  4297K|  1296   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_ATR_1_MESIAC_ROK |  9444 |       |    27   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

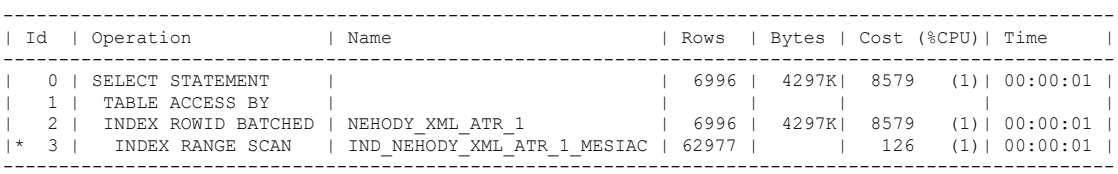
Obrázok 65: Plán vykonania - *Select* mesiac a rok nehody s indexom pre tabuľku *nehody_xml_atr_1*

Posledný index, ktorý sme vytvorili indexoval mesiace z dátumu. *Def. príkazu 116* zobrazuje kód tohto indexu.

```
CREATE INDEX ind_nehody_xml_atr_1_mesiac
 ON nehody_xml_atr_1
 (TO_CHAR(TO_DATE(
      ExtractValue(xml_data,'/nehoda/p2a_datum'),
      'YYYY-MM-DD'),'MM'));
```

Definícia príkazu 116: Index - pre mesiac nehody v tabuľke *nehody_xml_atr_1*

Po spustení tretieho príkazu s definovaným indexom a zobrazení plánu vykonania sme zistili, že náklady klesli na hodnotu 8 579. Plán vykonania je zobrazený na *Obrázku 66*.



```
-----------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                               |  6996 |  4297K|  8579   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                               |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_ATR_1              |  6996 |  4297K|  8579   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_ATR_1_MESIAC   | 62977 |       |   126   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

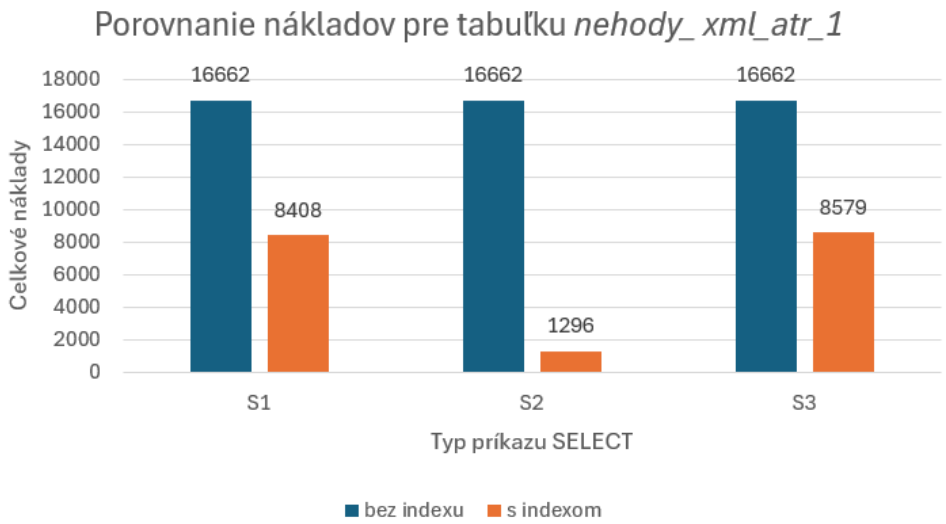
Obrázok 66: Plán vykonania - *Select* mesiac nehody s indexom pre tabuľku *nehody_xml_atr_1*

Zhrnutie jednotlivých nákladov na vykonanie vyššie uvedených príkazov pre tabuľku *nehody_xml_atr_1*, kde je dátum nehody ako vnútorný element v XML dokumente sa nachádza v *Tabuľke 24*.

Tabuľka 24: Zhrnutie nákladov príkazu *SELECT* pre tabuľku *nehody_xml_atr_1*

Podmienka	Náklady (bez indexu)	Náklady (s indexom)
TO_CHAR('YYYY')	16 662	8 408
TO_CHAR('MM.YYYY')	16 662	1 296
TO_CHAR('MM')	16 662	8 579

Grafická reprezentácia porovnania celkových nákladov na vykonanie príkazov sa nachádza na *Obrázku 67*. Celkové náklady predstavujú internú hodnotu, na základe ktorej sa optimalizátor rozhoduje.



Obrázok 67: Stĺpcový graf porovnania nákladov pre tabuľku *nehody_xml_atr_1*

Tabuľka *nehody_xml_atr_2*

Tabuľka *nehody_xml_atr_2* je tabuľka s atribútom typu *XMLTYPE*. Okrem tohto atribútu obsahuje ako ďalší stĺpec aj identifikačné číslo XML dokumentu. Rozdiel oproti tabuľke *nehody_xml_atr_1* je v tom, že v XML dokumente je *p2a_datum* ako atribút elementu. Definície príkazov boli rovnaké ako vyššie uvedené, líšili sa len v ceste k atribútu. *Def. príkazu 117* zobrazuje cestu k atribútu *dátum*, ktorý sme využívali v jednotlivých príkazoch.

```
/nehoda/@p2a_datum
```

Definícia príkazu 117: Cesta k atribútu XML dokumentu

Prvý príkaz, ktorý sme vykonali bol príkaz, ktorý vyberal záznamy z roku 2018. Najskôr sme tento príkaz spustili bez akýchkoľvek indexov. Plán vykonania a celkové náklady tohto príkazu sú zobrazené na *Obrázku 68*.

```
-------------------------------------------------------------------------------------
| Id  | Operation         | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                  |  6996 |  4297K| 16662   (11)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| NEHODY_XML_ATR_2 |  6996 |  4297K| 16662   (11)| 00:00:01 |
-------------------------------------------------------------------------------------
```

Obrázok 68: Plán vykonania - *Select* rok nehody pre tabuľku *nehody_xml_atr_2*

Potom sme spustili druhý aj tretí príkaz *SELECT*, v ktorom vyberáme záznamy z mája 2018 a záznamy len z mája. Po zobrazení plánu vykonania pre oba príkazy sme dostali rovnaké plány vykonania ako na *Obrázku 68* s celkovými nákladmi 16 662.

Ďalej sme chceli zistiť, či indexy ktoré vytvoríme ovplyvnia plány vykonania týchto príkazov. Prvý index bol funkcionálny index, ktorý indexoval roky. Po spustení príkazu

SELECT s týmto indexom sa náklady na jeho vykonanie znížili na hodnotu 8 408. Plán vykonania je možné vidieť na *Obrázku 69*.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                          |  6996 |  4297K|  8408   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                          |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_ATR_2         |  6996 |  4297K|  8408   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_ATR_2_ROK |   104K|       |   237   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Obrázok 69: Plán vykonania - *Select* rok nehody s indexom pre tabuľku *nehody_xml_atr_2*

Druhý index indexoval mesiace a roky nehôd, pričom mal rovnakú definíciu ako *def. príkazu 115*, líšil sa len v ceste, ktorá viedla k atribútu elementu. Po spustení príkazu a zobrazení plánu vykonania sa náklady na jeho vykonanie znížili na hodnotu 1 296. Plán vykonania je znázornený na *Obrázku 70*.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                               |  6996 |  4297K|  1296   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                               |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_ATR_2              |  6996 |  4297K|  1296   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_ATR_2_MESIAC_ROK |  9444 |       |    27   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Obrázok 70: Plán vykonania - *Select* mesiac a rok nehody s indexom pre tabuľku *nehody_xml_atr_2*

Posledný index, ktorý sme vytvorili pre túto tabuľku, indexoval hodnoty mesiacov. Definícia jeho kódu bola podobná ako je ukázané na *def. príkazu 116*, avšak líšil sa v ceste k atribútu. Po spustení a zobrazení plánu vykonania sme zistili, že celkové náklady dosiahli hodnotu 8 579. Plán vykonania je zobrazený na *Obrázku 71*.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                          |  6996 |  4297K|  8579   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                          |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_XML_ATR_2         |  6996 |  4297K|  8579   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_XML_ATR_2_MESIAC | 62977 |       |   126   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Obrázok 71: Plán vykonania - *Select* mesiac nehody s indexom pre tabuľku *nehody_xml_atr_2*

Zhrnutie jednotlivých nákladov na vykonanie vyššie uvedených príkazov pre tabuľku *nehody_xml_atr_2*, kde je dátum nehody ako atribút elementu v XML dokumente sa nachádza v *Tabuľke 25*.

Tabuľka 25: Zhrnutie nákladov príkazu *SELECT* pre tabuľku *nehody_xml_atr_2*

Podmienka	Náklady (bez indexu)	Náklady (s indexom)
TO_CHAR('YYYY')	16 662	8 408
TO_CHAR('MM.YYYY')	16 662	1 296
TO_CHAR('MM')	16 662	8 579

Ako je možné vidieť na experimentoch pre tabuľky *nehody_xml_atr_1* a *nehody_xml_atr_2*, či je dátum ako vnútorný element alebo ako atribút elementu XML dokumentu, náklady na vykonanie príkazov sa nelíšili.

Tabuľka nehody_json_1

Poslednou porovnávanou tabuľkou je tabuľka, ktorá uchováva záznamy o nehodách v tvare JSON dokumentu. Aj pre túto tabuľku sme si vytvorili tri príkazy *SELECT*, ktorých náklady na vykonanie sme si zaznamenali a porovnali s ostatnými typmi tabuliek.

Prvý príkaz vyberal všetky záznamy nehôd, ktoré sa stali v roku 2018. *Def. príkazu 118* zobrazuje kód príkazu.

```
SELECT *
 FROM nehody_json_1
 WHERE TO_CHAR(TO_DATE(
           JSON_VALUE(json_data, '$.p2a_datum'),
           'YYYY-MM-DD"T"HH24:MI:SS'), 'YYYY') = '2018';
```

Definícia príkazu 118: *Select* - rok nehody pre tabuľku *nehody_json_1*

Po spustení príkazu a zobrazenia nákladov na jeho vykonanie vidíme, že náklady dosiahli hodnotu 167 000. *Obrázok 72* zobrazuje plán vykonania tohto príkazu *SELECT*.

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |  6996 |    24M|   167K  (1)| 00:00:07 |
|*  1 |  TABLE ACCESS FULL| NEHODY_JSON_1 |  6996 |    24M|   167K  (1)| 00:00:07 |
-------------------------------------------------------------------------------
```

Obrázok 72: Plán vykonania - *Select* rok nehody pre tabuľku *nehody_json_1*

Druhý príkaz vyberal všetky záznamy nehôd, ktoré sa stali v máji v roku 2018. *Def. príkazu 119* zobrazuje kód príkazu *SELECT*.

```
SELECT *
 FROM nehody_json_1
 WHERE TO_CHAR(TO_DATE(
           JSON_VALUE(json_data, '$.p2a_datum'),
           'YYYY-MM-DD"T"HH24:MI:SS'), 'MM.YYYY') = '05.2018';
```

Definícia príkazu 119: *Select* – mesiac a rok nehody pre tabuľku *nehody_json_1*

Po jeho spustení a zobrazení plánu vykonania sme zistili, že náklady na vykonanie sú rovnaké ako pri prvom príkaze. *Obrázok 65* zobrazuje jeho plán vykonania.

Tretí príkaz *SELECT* zobrazoval všetky záznamy nehôd, ktoré sa stali v máji. *Def. príkazu 120* zobrazuje kód tohto príkazu.

```
SELECT *
 FROM nehody_json_1
 WHERE TO_CHAR(TO_DATE(
           JSON_VALUE(json_data, '$.p2a_datum'),
           'YYYY-MM-DD"T"HH24:MI:SS'), 'MM') = '05';
```

Definícia príkazu 120: *Select* – mesiac nehody pre tabuľku *nehody_json_1*

Po spustení príkazu a zobrazení plánu vykonania sme opäť získali rovnaký výsledok ako je zobrazené na *Obrázku 72*. Celkové náklady na tento príkaz dosiahli hodnotu 167 000.

Pre zlepšenie plánu vykonania vyššie uvedených príkazov sme si vytvorili indexy pre jednotlivé príkazy. Ako prvý sme vytvorili index, ktorý indexoval hodnoty rokov. *Def. príkazu 121* zobrazuje kód indexu.

```
CREATE INDEX ind_nehody_json_rok
 ON nehody_json_1
 (TO_CHAR(TO_DATE(
     JSON_VALUE(json_data, '$.p2a_datum'),
     'YYYY-MM-DD"T"HH24:MI:SS'), 'YYYY'));
```

Definícia príkazu 121: Index - pre rok nehody v tabuľke *nehody_json_1*

Po spustení príkazu s definovaným indexom a zobrazení plánu vykonania si môžeme všimnúť, že celkové náklady sa znížili na hodnotu 92 565. *Obrázok 73* zobrazuje plán vykonania tohto príkazu.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                     |   104K|   374M| 92565   (1)| 00:00:04 |
|   1 |  TABLE ACCESS BY     |                     |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_JSON_1       |   104K|   374M| 92565   (1)| 00:00:04 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_JSON_ROK |   104K|       |   237   (1)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Obrázok 73: Plán vykonania - *Select* rok nehody s indexom pre tabuľku *nehody_json_1*

Druhý index, ktorý sme vytvorili, indexoval hodnoty mesiacov a rokov. *Def. príkazu 122* zobrazuje definovaný index.

```
CREATE INDEX ind_nehody_json_mesiac_rok
 ON nehody_json_1
 (TO_CHAR(TO_DATE(
     JSON_VALUE(json_data, '$.p2a_datum'),
     'YYYY-MM-DD"T"HH24:MI:SS'), 'MM.YYYY'));
```

Definícia príkazu 122: Index - pre mesiac a rok nehody v tabuľke *nehody_json_1*

Po spustení príkazu s definovaným indexom a zobrazení plánu vykonania si môžeme všimnúť, že celkové náklady sa znížili na hodnotu 8 416. *Obrázok 74* zobrazuje plán vykonania príkazu SELECT s definovaným indexom pre mesiac a rok.

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation            | Name                      | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                           |  9444 |    33M|  8416   (1)| 00:00:01 |
|   1 |  TABLE ACCESS BY     |                           |       |       |            |          |
|   2 |  INDEX ROWID BATCHED | NEHODY_JSON_1             |  9444 |    33M|  8416   (1)| 00:00:01 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_JSON_MESIAC_ROK |  9444 |       |    27   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

Obrázok 74: Plán vykonania - *Select* mesiac a rok nehody s indexom pre tabuľku *nehody_json_1*

Posledný index indexoval hodnoty mesiacov nehôd, jeho definícia je zobrazená v *def. príkazu 123*.

```
CREATE INDEX ind_nehody_json_mesiac
 ON nehody_json_1
 (TO_CHAR(TO_DATE(
      JSON_VALUE(json_data, '$.p2a_datum'),
      'YYYY-MM-DD"T"HH24:MI:SS'), 'MM'));
```

Definícia príkazu 123: Index - pre mesiac nehody v tabuľke *nehody_json_1*

Po spustení príkazu s definovaným indexom a zobrazení plánu vykonania si môžeme všimnúť, že celkové náklady sa znížili na hodnotu 56 063. *Obrázok 75* zobrazuje plán vykonania príkazu SELECT s definovaným indexom pre mesiac.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation            | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                     | 62977 |   225M| 56063   (1)| 00:00:03 |
|   1 |  TABLE ACCESS BY     |                     |       |       |            |          |
|   2 |  INDEX ROWID BATCHED| NEHODY_JSON_1        | 62977 |   225M| 56063   (1)| 00:00:03 |
|*  3 |   INDEX RANGE SCAN   | IND_NEHODY_JSON_MESIAC | 62977 |       |   126   (1)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

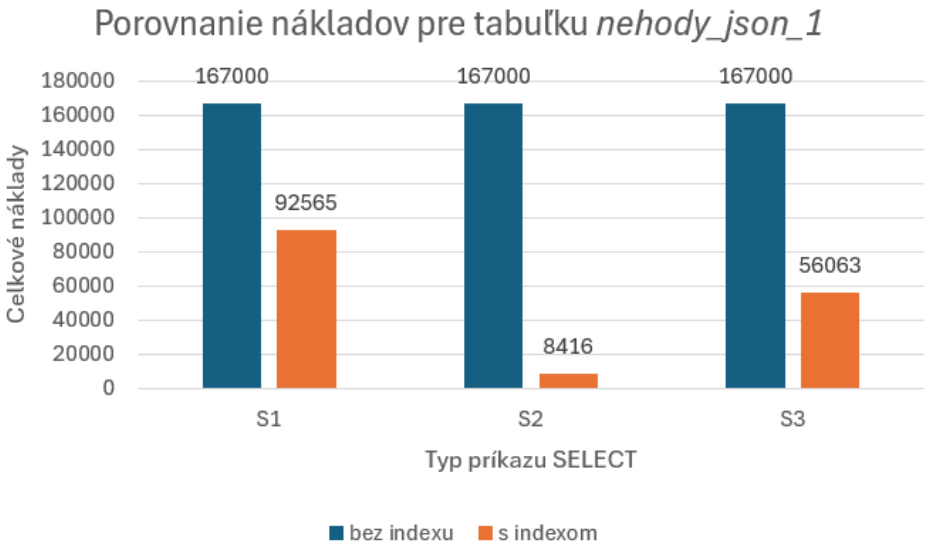
Obrázok 75: Plán vykonania - *Select* mesiac nehody s indexom pre tabuľku *nehody_json_1*

Zhrnutie jednotlivých nákladov na vykonanie vyššie uvedených príkazov pre tabuľku *nehody_json_1* sa nachádza v *Tabuľke 26*.

Tabuľka 26: Zhrnutie nákladov príkazu SELECT pre tabuľku *nehody_json_1*

Podmienka	Náklady (bez indexu)	Náklady (s indexom)
TO_CHAR('YYYY')	167 000	92 565
TO_CHAR('MM.YYYY')	167 000	8 416
TO_CHAR('MM')	167 000	56 063

Grafická reprezentácia porovnania celkových nákladov na vykonanie príkazov sa nachádza na *Obrázku 76*. Celkové náklady predstavujú internú hodnotu, na základe ktorej sa optimalizátor rozhoduje.



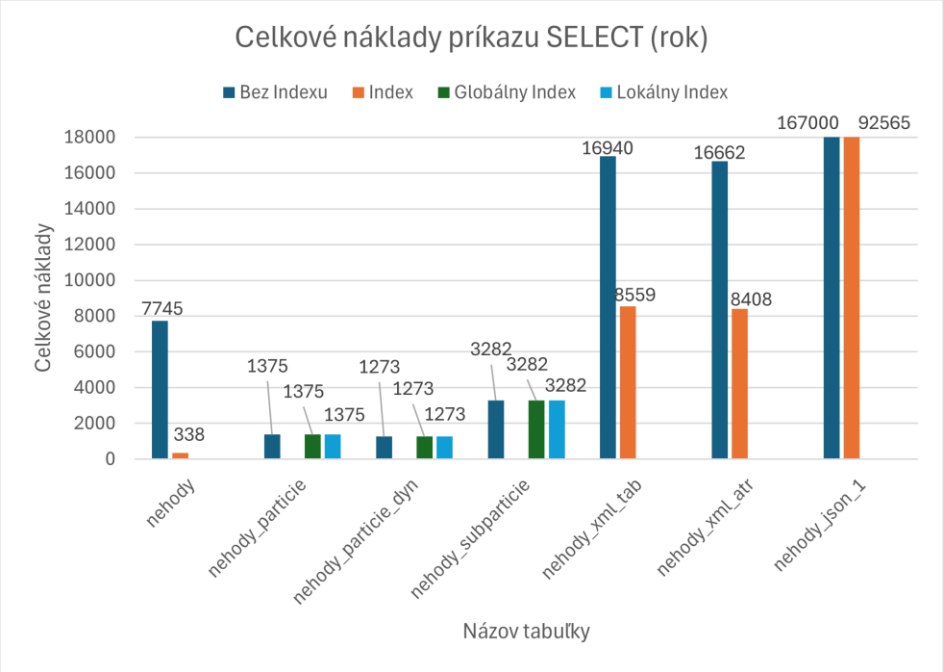
Obrázok 76: Stĺpcový graf porovnania nákladov pre tabuľku nehody_json_1

Zhrnutie experimentov a odporúčania

Experimenty si zhrnieme podľa typov príkazov *SELECT* pre jednotlivé tabuľky. Prvým pozorovaným experimentom bolo vytvorenie príkazu *SELECT*, ktorý vyberal záznamy na základe **roku** z dátumu nehody. Tento experiment sme vykonali najskôr bez možného zlepšenia, a potom sme vytvorili príslušné indexy, ktoré by dokázali zefektívniť príkaz. K dispozícii sme mali 9 tabuliek, klasickú tabuľku faktov, tabuľky rozdelené na partície a subpartície, XML tabuľky a tabuľky obsahujúce JSON dokumenty. Pre zjednodušenie sme zlúčili tabuľky, ktoré obsahovali XML dokumenty ako atribút a tabuľky objektov typu *XMLTYPE*, pretože náklady na vykonanie týchto príkazov boli totožné.

Na *Obrázku 77* je znázornený stĺpcový graf pre prvý typ príkazu *SELECT*, ktorý vyberá záznamy len pre rok 2018. Tmavomodrou farbou sú zobrazené náklady na vykonanie príkazu bez využitia indexov. Vidíme, že s vytvorením indexov sa náklady na vykonanie príkazov značne znížili. Stĺpce označené oranžovou farbou predstavujú náklady na vykonanie príkazov s využitím klasických indexov. Tabuľky rozdelené na partície a subpartície majú 2 typy indexov, a preto sú stĺpce označené zelenou a slabomodrou farbou. Tabuľka, ktorá obsahovala JSON dokumenty sa ukázala ako najmenej efektívna, pretože náklady sú príliš vysoké aj po vytvorení indexu. Medzi ďalšie menej efektívne možnosti ukladania záznamov sa zaradili tabuľky, ktoré obsahujú XML dokumenty. Tabuľka so subpartíciami dosahuje relatívne dobré výsledky, avšak rovnako ako aj pri

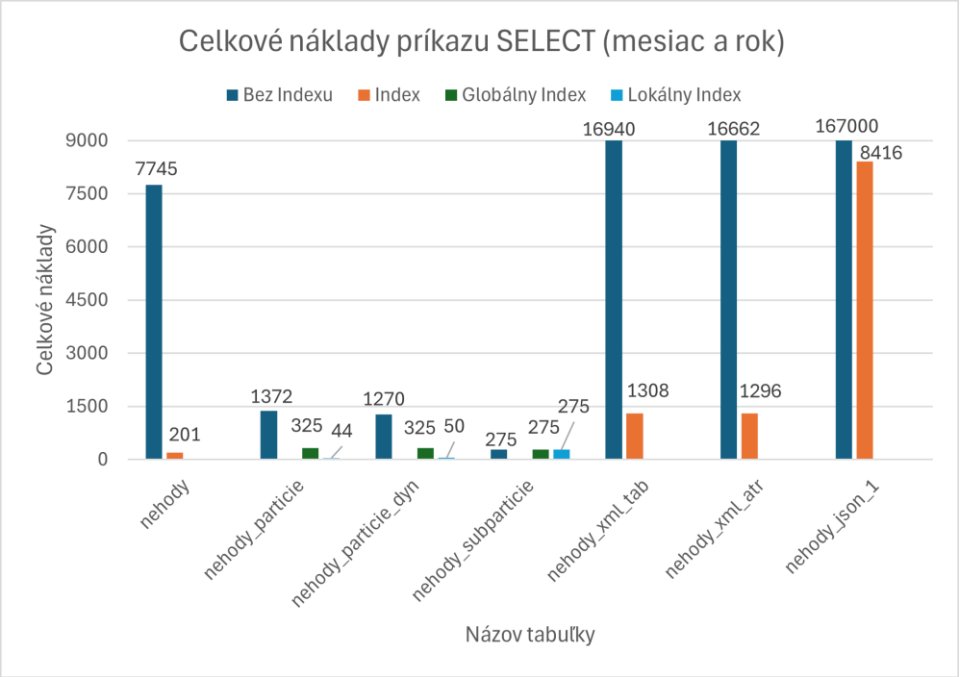
tabuľkách s partíciami náklady na vykonanie príkazov sú rovnaké, keďže vytvorené indexy neboli použité pre tieto príkazy a to z toho dôvodu, že vďaka tomu, že sme pristupovali priamo k partícii daného roku nebola potrebná podmienka *WHERE*. Klasická tabuľka faktov bez využitia indexov nie je tak efektívna, ako tabuľky rozdelené na partície, na druhej strane hodnota celkových nákladov na vykonanie príkazu s vytvoreným indexom oproti ostatným možnostiam sa javí ako najefektívnejšia možnosť keďže ako jedinej sa podarilo dosiahnuť náklady 338.



Obrázok 77: Stĺpcový graf celkových nákladov príkazu *SELECT* (rok)

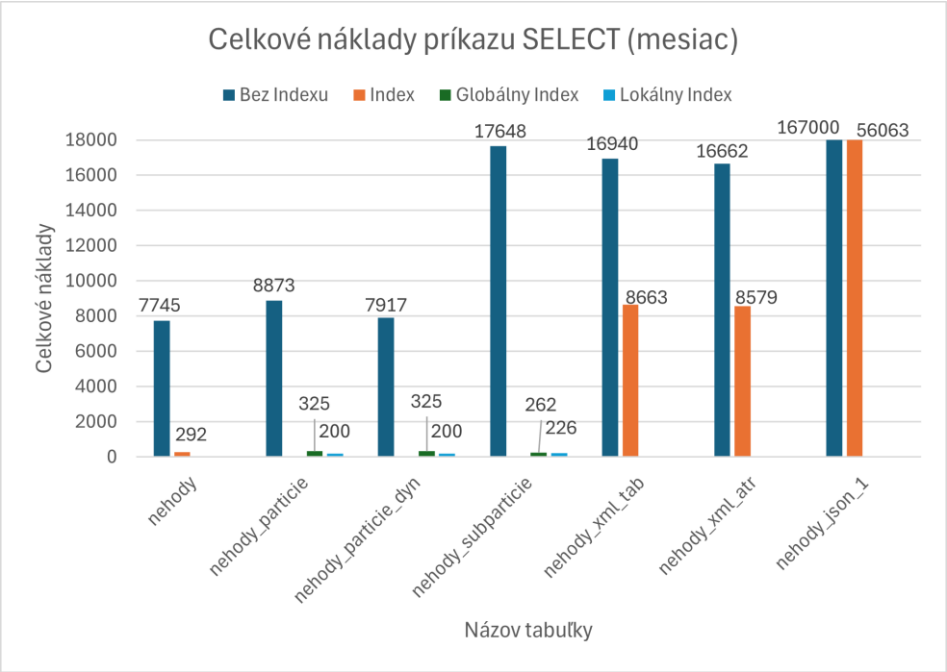
Na *Obrázku 78* je znázornený stĺpcový graf pre druhý typ príkazu *SELECT*, ktorý obsahoval podmienku vyberajúcu záznamy z mája roku 2018. Na obrázku vidíme, že náklady na vykonanie príkazov pre jednotlivé tabuľky bez využitia indexov sú omnoho vyššie ako náklady na vykonanie príkazov s vytvorenými indexami. Opäť je najmenej efektívna možnosť vyhľadávania záznamov v tabuľke uchovávajúcej JSON dokumenty, či bez vytvorených indexov alebo s indexami. XML tabuľky sú druhou najmenej efektívnou možnosťou. Náklady na vykonanie sú však v tabuľke, ktorá uchováva XML dokumenty ako jeden zo stĺpcov, menšie ako tabuľka XML dokumentov. Tabuľka subpartícií má stabilné hodnoty celkových nákladov, keďže počas experimentov s indexami optimalizátor nevyužil ani jeden, avšak v rámci príkazu sa pristupovalo priamo k záznamom z 5 mesiaca roku 2018. Tabuľky rozdelené na partície majú hodnoty nákladov porovnateľné, rozdiel by mohol nastať v prípade, ak by systém dynamicky vytvoril viac partícií, ako keď by sme v definícií tabuľky definovali pevne stanovené partície. V tomto prípade však dosahujú najlepšie

výsledky vyhľadávania záznamov spomedzi všetkých tabuliek s vytvoreným lokálnym indexom. Klasická tabuľka faktov bez vytvorených indexov nedosahuje až tak dobré výsledky ako s vytvoreným indexom, vďaka ktorému sa hodnota celkových nákladov dokázala znížiť na 201.



Obrázok 78: Stĺpcový graf celkových nákladov príkazu *SELECT* (mesiac, rok)

Na *Obrázku 79* je znázornený stĺpcový graf pre posledný typ príkazu *SELECT*, ktorý vyberal záznamy len podľa mesiaca. Na obrázku vidíme, že náklady na vykonanie príkazu bez vytvorených indexov sú podstatne vyššie ako s indexami. Tabuľka, ktorá uchováva JSON dokumenty je aj v tomto prípade najmenej efektívnou metódou pre vyhľadávanie záznamov aj napriek vytvoreným indexom. Tabuľky obsahujúce XML dokumenty majú približne rovnaké náklady na vykonanie príkazu bez vytvorených indexov, ale aj s indexami. Hodnota nákladov v tabuľke rozdelenej na subpartície bez indexov bola porovnateľná ako hodnoty nákladov pre XML dokumenty. S vytvorenými indexami sa náklady na vykonanie príkazu v tabuľke rozdelenej na subpartície značne znížili a vďaka lokálnemu indexu dosahovala takmer najlepšie hodnoty celkových nákladov na vykonanie definovaného príkazu. Klasická tabuľka faktov a tabuľky rozdelené na partície majú bez indexov približne rovnaké náklady na vykonanie dotazu. Rozdiel nastáva pri vytvorení indexov, avšak celkové náklady v týchto tabuľkách sú približne rovnaké. Každopádne tabuľky rozdelené na partície spolu s vytvoreným indexom dosahujú najlepšie výsledky.



Obrázok 79: Stĺpcový graf celkových nákladov príkazu *SELECT* (mesiac)

Taktiež je možné vidieť rozdiel medzi globálnymi a lokálnymi indexami v tabuľkách rozdelených na partície a subpartície. Lokálne indexy dosahujú lepšie výsledky ako globálne indexy.

3.6 Referencie funkcií v príkaze SELECT

Mnoho informačných systémov je zameraných na dátovú analytiku, v rámci ktorej sa produkujú výsledky v akejkoľvek grafickej podobe, na to aby umožnili lepšie pochopiť procesy a spôsob rozhodovania sa na základe vytvorených prognóz [29]. Za získané výstupy môžeme vďačiť dátovej vrstve a jej metódam na prístup, manipuláciu a spracovanie dát. V rámci výskumu sme sa zamerali na spracovanie údajov a optimalizáciu výkonu pričom sa orientujeme na spracovanie a referencie funkcií. Existujúce riešenia správy funkcií zahŕňajú ukladanie výsledkov do vyrovnávacej pamäte, funkcionálne indexy, materializované pohľady, virtuálne stĺpce, alebo optimalizáciu funkcií tak, aby sa dali priamo aplikovať v SQL alebo PL/SQL.

Oracle Database 23c predstavila niekoľko možností zlepšenia výkonu a nový prístup k aliasom výrazov a stĺpcov. Nami navrhované riešenie sa zameriava na identifikáciu a extrakciu aliasov stĺpcov, výrazov a funkcií, ukladanie referencií v pamäti a databázovej vrstve, optimalizáciu prenosu medzi nimi pomocou výmeny (presunu), ako aj na migráciu kontrolných bodov a volania funkcií. Poskytuje komplexnú architektúru, ktorá zahŕňa

riadenie transakcií. Okrem toho poskytuje aj robustnú vrstvu, pričom každá vrstva je detailne analyzovaná z hľadiska výkonu jej výhod a obmedzení. Naša navrhovaná architektúra prináša výkonnostné benefity pre komplexné analytické dotazy, a taktiež, môže byť aplikovaná pri online spracovaní transakcií. Cieľom je extrahovať aliasy a zabezpečiť ich použiteľnosť v rámci celého procesu vykonávania dotazov, pričom sa snažíme znížiť časové nároky a celkové náklady na spracovanie dotazu.

Ako aj v predchádzajúcom výskume, aj tu sme využili databázu Oracle z viacerých dôvodov. Autonómny dátový sklad od spoločnosti Oracle je prvou a jedinou autonómnou databázou, ktorá je optimalizovaná pre analytické úlohy, počnúc dátovými skladmi zahŕňajúcich dátové trhy (data marts) a dátové jazerá (data lakes). Je to ideálne riešenie pre akéhokoľvek používateľa, beží na architektúre Exadata, ktorá znižuje náklady v priemere o 63% [56]. Ďalší dôvod výberu tohto typu databázy sa týka zložitosti a vylepšenia databázy z hľadiska architektúry, v ktorej je možné využiť referencie pre porovnanie výkonu. Nami navrhované riešenie je prevažne analyticky orientované, avšak dá sa nasadiť na akúkoľvek databázu.

Jazyk SQL je štandardom pre vytváranie, prevádzku a manipuláciu s databázou. V dnešnej dobe je to najčastejšie používaný prístup k správe dát. Jazyk SQL sa postupne vyvíjal a každá nová verzia prinášala mnohé vylepšenia. Jedným z týchto vylepšení je využitie aliasov stĺpca, výrazu a funkcií, ktoré sú uvedené v klauzule *SELECT*. Avšak nie je možné odkazovať sa na aliasy v rovnakom príkaze v klauzulách *WHERE, GROUP BY* a *HAVING*, pretože sú v týchto klauzulách pre databázový procesor ešte neznáme. Oracle 23c, vydaný v apríli 2023, je však využitý nový prístup, ktorý umožňuje odkazovať sa na aliasy kdekoľvek v príkaze [57] [58]. Tento prístup používame ako referenciu pre naše navrhované riešenie.

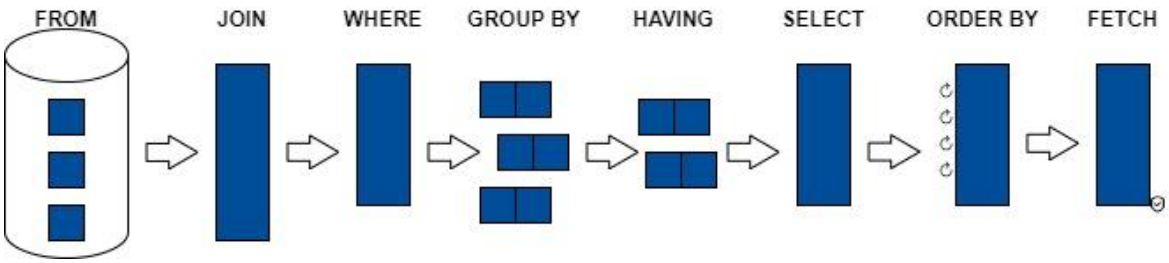
Celý popis štúdie a navrhovaného riešenia spolu s ďalšími podrobnosťami výskumu je možné nájsť v publikácii [34].

3.6.1 Definícia príkazu *SELECT* a jeho spracovanie

Pre analýzu údajov je potrebné získať dáta z databázy, vyhodnotiť ich a zostaviť výstupy. V SQL je to možné prostredníctvom príkazu *SELECT*, ktorý sa skladá z nasledujúcich klauzúl [59] [60]:

- *SELECT* – obsahuje zoznam stĺpcov, výrazov, funkcií, ktoré predstavujú výslednú množinu údajov. Názvy stĺpcov sa štandardne skopírujú z definície, avšak môžeme ich premenovať prostredníctvom aliasov, ktoré sú špecifikované hneď za definíciou daného stĺpca. Alias v sebe ukrýva definíciu a spôsob výpočtu spracovania výrazu, funkcie, čím je možné ľahšie sa odkazovať pri ďalšom spracovaní. Taktiež je potrebný kvôli asociácii s pohľadom, tabuľkou alebo materializovaným pohľadom.
- *FROM* – špecifikuje tabuľky alebo pohľady s voliteľne definovanými operáciami spojenia a podmienkami prepojenia.
- *WHERE* – špecifikuje podmienky na filtrovanie záznamov, vyberá len tie riadky, ktoré spĺňajú všetky podmienky definované v tejto klauzule.
- *GROUP BY* – je spojená s agregačnými funkciami, ktoré sa počítajú pre každú definovanú skupinu.
- *HAVING* – je to filter, ktorý sa používa pre podmienky založené na agregačných funkciách.
- *ORDER BY* – je klauzula, ktorá triedi výslednú množinu záznamov na základe definovaných kritérií.
- *LIMIT* – je klauzula, ktorá znižuje veľkosť výslednej množiny tým, že vyberie len časť údajov.

Príkaz po jeho definovaní je potrebné vykonať. Klauzuly *SELECT* a *FROM* sú povinné. Zvyšné časti príkazu sú voliteľné. Vykonávanie jednotlivých klauzúl má svoje poradie. Na *Obrázku 80* je zobrazené poradie vykonávania jednotlivých klauzúl.



Obrázok 80: Poradie vykonania príkazu *SELECT*

V prvom rade sa začína so zdrojovými tabuľkami. Tabuľky je možné prepojiť pomocou spojenia definovaného v klauzule *FROM*, pričom ako ďalšia v poradí sa spracuje klauzula *WHERE*. Cieľom je čo najviac zredukovať množinu údajov pre ďalšie spracovanie. Nasleduje spracovanie klauzuly *GROUP BY*, pre ktorú sa počítajú agregačné funkcie. Po

vytvorení skupín a výpočte agregačných funkcií dochádza k spracovaniu klauzuly *HAVING,* ktorá umožňuje opäť zredukovať výslednú množinu. Po predchádzajúcich fázach, dochádza k spracovaniu klauzuly *SELECT,* v ktorej sa určí štruktúra výslednej množiny. V tejto časti sa môžu definovať aliasy stĺpcov, ktoré slúžia na naformátovanie názvov jednotlivých stĺpcov, napríklad pri použití funkcií zaokrúhlenia, výberu reťazcov a podobne. Takéto dáta sa môžu použiť aj v iných klauzulách. Vyžaduje to však skopírovanie definície funkcií, pretože alias definovaný v klauzule *SELECT*, nemôže byť použitý v klauzulách *WHERE, GROUP BY, HAVING*, pretože tieto klauzuly sa vykonávajú pred spracovaním klauzuly *SELECT*. Toto obmedzenie môže viesť k mnohým chybám a môže priniesť dodatočné nároky v rámci programovania, ale i zvýšiť nároky na čas a celkové náklady na spracovanie príkazu, hlavne preto, lebo ak je volanie funkcie použité v príkaze viackrát, funkcia bude vypočítaná toľkokrát, koľkokrát sa v príkaze nachádza. Verzia Oracle 23c priniesla možnosť odkazovať sa na aliasy kdekoľvek v príkaze *SELECT*.

Jazyk SQL je neprocedurálny jazyk, v ktorom databázový optimalizátor musí vybrať najvhodnejšiu možnosť pre lokalizovanie, prístup a zlúčenie množiny údajov, aby poskytol výstupnú množinu dát v správnom formáte a za vhodný čas. Preto boli navrhnuté a vyvinuté indexy, ktoré umožňujú zjednodušiť proces vyhľadávania údajov tak, že nie je potrebné prehľadávať celú tabuľku riadok po riadku [61] [62]. Jednotlivé metódy spojenia tabuliek, prípadne prístupové cesty k dátam sú vykonávané v počiatočných fázach. V tejto fáze sa taktiež môžu volať jednotlivé funkcie slúžiace na filtrovanie. Preto je ešte predtým nutné identifikovať všetky aliasy stĺpcov, aby ich bolo možné využiť v rámci celého príkazu *SELECT*. To však môže ovplyvniť aj plán vykonania.

3.6.2 Navrhované riešenie

Naše navrhované riešenie vychádza z existujúcich prístupov, ktoré využívame ako základ. Jeho cieľom je optimalizovať výkon správy aliasov, pričom toto riešenie nesmie negatívne ovplyvniť ani zmeniť štruktúru príkazu. Každopádne je potrebné vytvoriť vlastnú architektúru a dátové štruktúry na to, aby bolo možné identifikovať aliasy v príkaze, a zabezpečiť ich použiteľnosť v jednotlivých klauzulách tohto príkazu alebo v jednotlivých vetvách vnorených dotazov. Preto sme sa pri implementácii tohto riešenia zamerali na zachovanie neprocedurálnej povahy definície príkazov ako základu jazyka SQL.

Identifikácia a extrakcia aliasov

Aliasy stĺpcov, funkcií a výrazov sa zvyčajne extrahovali a spracovávali počas vykonávania klauzuly *SELECT*. Tieto aliasy sa využívali hlavne na úpravu formátu názvov

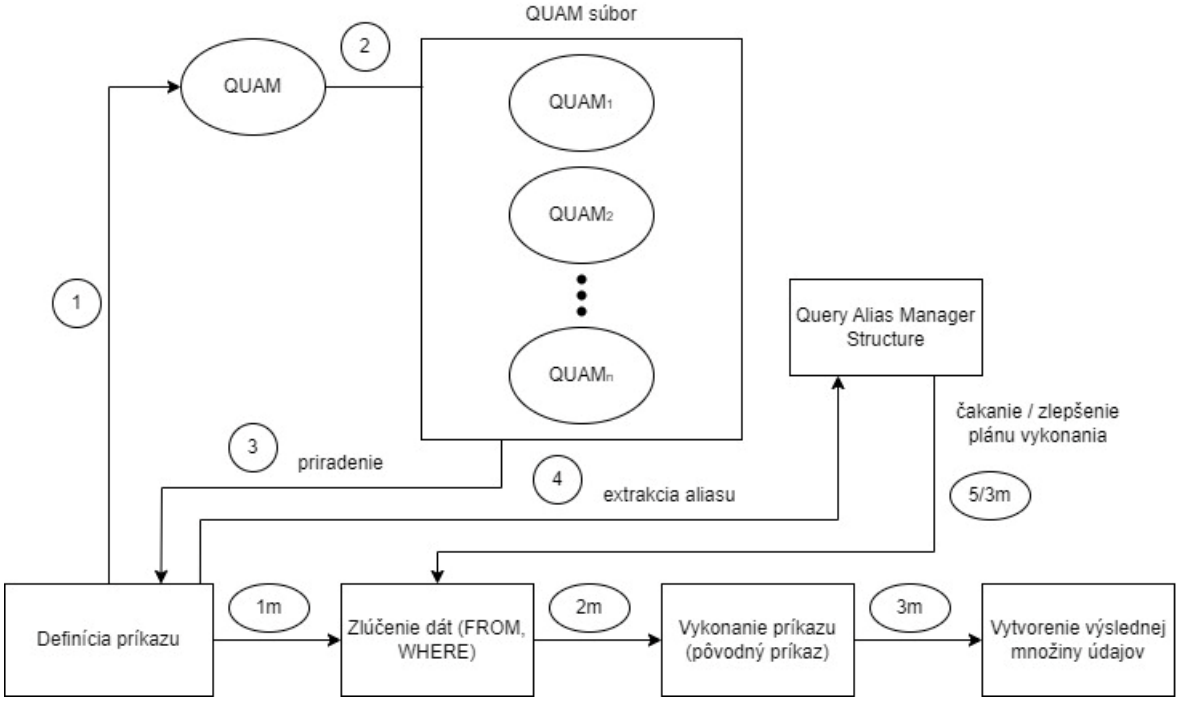
stĺpcov výslednej množiny. S ďalšími zlepšeniami sa príkazy stávali čoraz komplexnejšie s čím súviselo aj vnáranie jednotlivých príkazov. Oracle 23c umožňuje extrakciu aliasov v prvej fáze, takže zvyšok vykonávania príkazu čaká na dokončenie extrakcie, čo môže viesť k rôznym obmedzeniam, napríklad v prípade, keď sa aliasy vôbec nepoužijú.

Navrhované riešenie zavádza na pozadí nový proces takzvaný Query Alias Extractor Master (QUAM), ktorý predstavuje master (nadradený) proces celej aktivity a vyvažuje aktivitu záťaže v rámci množiny takzvaných worker (pracovných) procesov – Query Alias Extractor Worker (QUAWn). Riadiaci proces je len jeden, zatiaľ čo worker procesov môže byť ľubovoľný počet, pričom ich je možné dynamicky vytvárať a uvoľňovať.

Hlavným cieľom master procesu je spustiť proces extrakcie okamžite, pridelením worker procesu k dotazu. Na spustenie procesu extrakcie, musí byť prítomný aspoň jeden worker proces, pričom ich môže byť aj viac, v prípade existencie vnorených príkazov. To znamená, že každej klauzule *SELECT* môže byť pridelený jeden worker. *Obrázok 81* znázorňuje proces extrakcie. V prípade, že sa má dotaz vykonať, proces QUAM bude notifikovaný o hĺbke príkazu, to je počet vnorení a zložitosti dotazu. Následne dôjde k vytvoreniu štruktúry, ktorá bude obsahovať aliasy, ich viditeľnosť a použiteľnosť v rámci dotazu. Potom sa každej klauzule *SELECT* priradia dostupné worker procesy. Ak nie je k dispozícii dostatok voľných worker procesov, tak jeden worker proces môže byť priradený viacerým klauzulám. V takom prípade sa vnorené dotazy spracujú postupne na základe definície dotazu a odkazov. V prípade úplného prehľadania určitej klauzuly bude vytvorená štruktúra notifikovaná. Nakoniec, ak worker proces dokončí svoju úlohu, zmení sa jeho stav na *dostupný* a presmeruje sa do súboru voľných worker procesov. V tomto prípade už QUAM nie je informovaný, čo obmedzuje jeho zaťaženie. Je evidentné, že extrakcia a správa aliasov sa vykonávajú samostatne, a teda vykonanie pôvodného príkazu *SELECT* nie je ovplyvnené. Ak sú aliasy prítomné a odkazuje sa na ne v procese spájania zdrojových tabuliek, ich definícia a mapovanie sú vyhľadané vo vytvorenej štruktúre spravovanej hlavným procesom. Ak sa v nej už definícia aliasu nachádza, spracovanie môže priamo pokračovať. Naopak, ak daný alias ešte nebol definovaný, databázový optimalizátor rozhodne a vyberie ďalší krok spracovania, buď zmenou plánu vykonania alebo počkaním na spracovanie aliasu.

Obrázok 81 znázorňuje architektúru, databázové objekty, procesy a tok dát. Označenia 1m, 2m a 3m odkazujú na pôvodný tok spracovania dotazu. Označenie 1 – 5

predstavuje rozšírenie správy aliasov. Štruktúra na uchovávanie zoznamu identifikovaných aliasov je označená ako Query Alias Manager Structure.



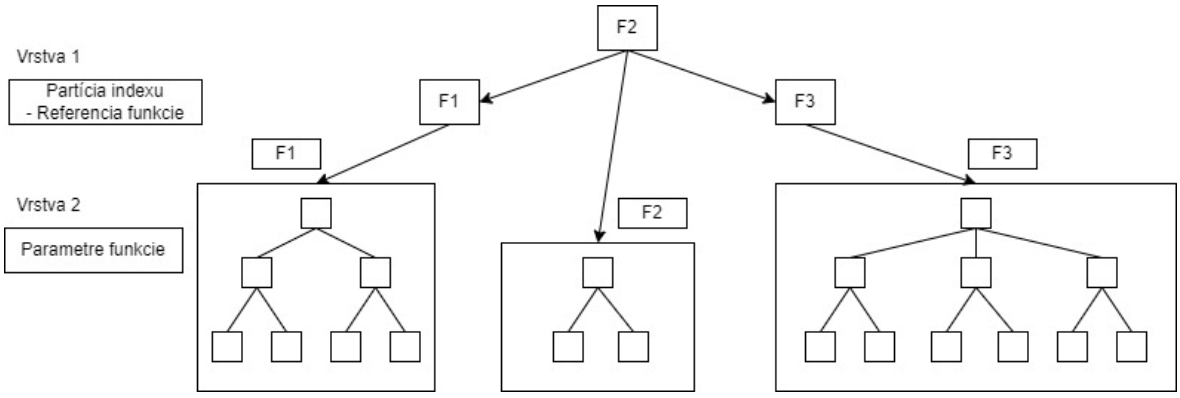
Obrázok 81: Proces extrakcie príkazu

V porovnaní s existujúcim prístupom je dôležité spomenúť, že pôvodný príkaz sa nepredefinuje použitím aliasov, namiesto toho sa spracovávajú dynamicky počas vykonávania jednotlivých klauzúl, nie celého dotazu.

Výpis aliasov

Query Alias Manager Structure bola predstavená v predchádzajúcej časti. Tá je priamo spojená s dotazom, a preto je súčasťou pamäte prepojenou so session tzv. Private Global Area (PGA). Tento typ pamäte sa nijakým spôsobom nezdieľa a je vyhradený výlučne len pre definíciu príkazu. Je organizovaná pomocou dátovej štruktúry B-strom, pričom kľúčom je celá definícia funkcie alebo môže byť použitý aj jej hash, ktorý pozostáva z odkazov na parametre. Štruktúra B+strom sa nepoužíva preto, lebo nie je potrebné triedenie dát na listovej úrovni.

Jedným z navrhovaných zlepšení je využitie multi-indexu. Prvá vrstva je kľúč partície definovaný funkciou a druhá vrstva je samotný index, priradený každému uzlu predchádzajúcej vrstvy, pričom kľúčom indexu je zoznam parametrov. Architektúra odkazu multi-indexu na funkciu je zobrazená na *Obrázku 82*.



Obrázok 82: Multi-index

Obmedzenie multi-indexu spočíva práve v druhej vrstve. Mnohé varianty volania funkcií môžu byť rozšírené o voliteľné parametre, preddefinované hodnoty a preťaženie v balíkoch. Aby sa však zabezpečila ich použiteľnosť musia byť splnené nasledovné pravidlá:

- Každá funkcia musí využívať svoju úplnú definíciu, pričom chýbajúce hodnoty budú nahradené preddefinovanými hodnotami.
- Každá preťažovaná funkcia sa považuje za samostatnú definíciu

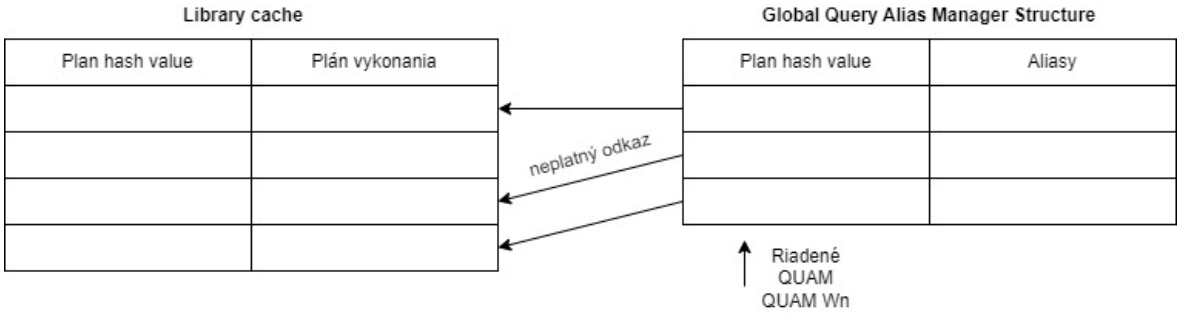
Tieto pravidlá sa aplikujú na parametre tak, aby mali spoločný formát, ktorý je využitý ako kľúč indexu.

Lokálne úložisko príkazov

Query Alias Manager Structure, ktorú sme navrhli, je umiestnená v pamäti inštancie v privátnej oblasti spojenej so session a dotazom. Po dokončení príkazu je možné túto referenciu uvoľniť. V tomto riešení sa snažíme odložiť operáciu uvoľnenia na koniec transakcie alebo session (ak sa nepresiahne pridelená kapacita PGA). Predpokladá sa, že sa daný príkaz môže neskôr využiť v jeho pôvodnej definícii. Takto je možné využiť už extrahované aliasy a odkazy. Rovnaká identifikácia príkazu sa spolieha na hash hodnotu vypočítanú pre každý príkaz *SELECT*, ktorá sa tiež používa aj na identifikáciu plánu vykonania a hash hodnotu referenčného plánu.

Query Alias Manager Structure je privátna štruktúra a nie je zdieľaná medzi viacerými používateľmi alebo reláciami (session). Rovnaký príkaz však môžu spustiť rôzne aplikácie alebo prihlásení používatelia. Preto jedným z navrhovaných vylepšení je presunúť Query Alias Manager Structure do globálneho úložiska s názvom Global Query Alias Manager Structure. Táto štruktúra sa presunie do zdieľanej pamäte inštancie. Využíva

dvojicu kľúčov – hash hodnotu plánu a množinu aliasov s ich definíciami. Na obmedzenie kapacity tejto štruktúry sa využili rôzne techniky (podľa počtu odkazov, komplexnosti celkových nákladov spracovania, atď). Na základe ďalších štúdií sa najsľubnejšie riešenie javí frekvencia volania danej funkcie. Ak existuje niekoľko takýchto funkcií, potom sa monitoruje komplexnosť dotazu a počet extrahovaných aliasov.



Obrázok 83: Správa extrakcie aliasov

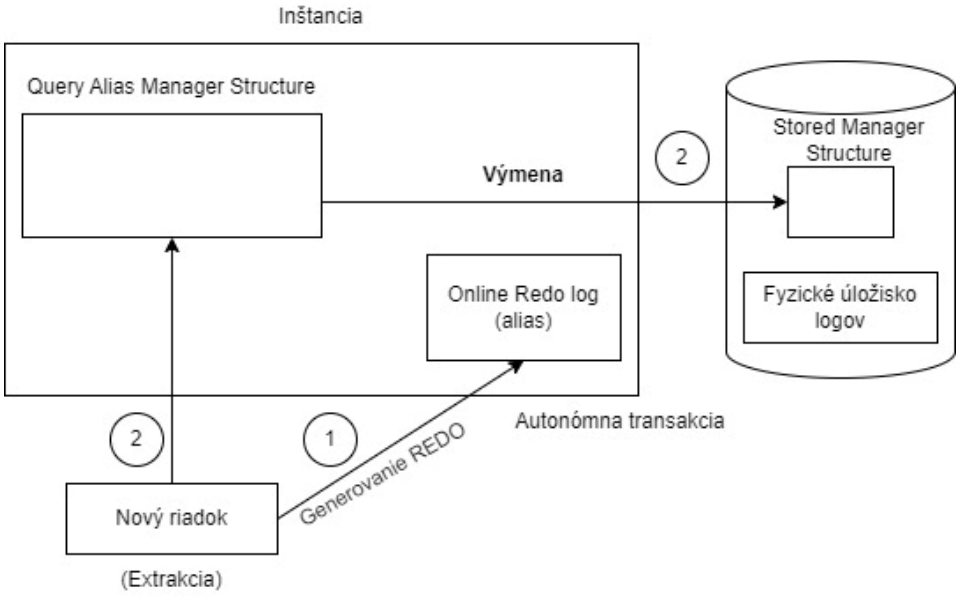
Zoznam extrahovaných aliasov je prepojený s plánom vykonania na základe hodnôt Plan hash values, čo je zobrazené na *Obrázku 83*. Smer odkazovania prechádza z Global Query Alias Manager Structure do úložiska plánu vykonania. Vytvára to obmedzenie pre neplatné odkazy v prípade uvoľnenia pamäte pre referencie aliasov. Navyše adaptívne plány vykonania môžu použiť rovnakú množinu aliasov pre rovnakú definíciu príkazu.

Výmena štruktúry

Global Query Alias Manager Structure sa nachádza v pamäti inštancie a je zdieľaná medzi používateľmi, ktorí môžu využívať daný dotaz. Jej veľkosť je dynamická a existujúci proces na pozadí tzv. Memory Manager, ktorý je v databázovom systéme vždy aktívny, nastavuje správnu hodnotu na základe zaťaženia a kapacity. Nie je však možné neobmedzene rozširovať veľkosť. Skôr či neskôr by bolo potrebné uvoľniť niektoré referencie a extrahované aliasy. Ako však už bolo spomenuté, je možné použiť niekoľko spôsobov, ako vybrať správny záznam na uvoľnenie zo štruktúry. Namiesto fyzického odstránenia zo štruktúry, je možné použiť len logické operácie. To znamená, že pred odstránením sa daný záznam uloží do archívu, ktorý sa nachádza v databáze. Aj tam sa môžu aplikovať rovnaké pravidlá pre obmedzenie kapacity úložiska, ale vo všeobecnosti je diskové úložisko menej nákladné.

Okrem toho, ak je databáza prevádzkovaná v cloude, tak na vyváženie výkonu a nákladov sa môžu využiť rôzne typy úložísk a parametrov. Preto sa zavádza na pozadí proces Alias Archiver, ktorý zabezpečuje uloženie údajov v úložisku databázy pred ich

odstránením z globálnej pamäte inštancie. Architektúra a správa záznamov je znázornená na *Obrázku 84*.



Obrázok 84: Štruktúra výmeny

Spracovanie funkcií - identifikácia funkcií, nielen aliasov, načítanie definície

Celý výskum a navrhované riešenie bolo inšpirované databázovým systémom Oracle 23c, ktorý umožňuje využívať referencie aliasov v akejkoľvek časti príkazu *SELECT*. Takže sme sa zamerali hlavne na toto rozšírenie. Pôvodný dotaz bol v praxi prepísaný nahradením aliasov s priamymi definíciami. Ak sa odkazoval na volanie funkcií, namiesto ich opakovaného volania sa vypočítané výsledky uložili do cache pamäte lokálne pre dotaz alebo globálne do štruktúry pamäte inštancie Result cache. Iná situácia nastáva v prípade, keď alias nezapuzdrí volanie funkcie. Vtedy sa neextrahuje definícia funkcie. Ak Result cache nie je explicitne nastavená pre danú session alebo príkaz, jedna funkcia sa môže volať viackrát. To platí aj v prípade, ak je príkaz komplexný a obsahuje vnorené príkazy.

Naše navrhované riešenie vylepšuje správu aliasov pomocou volania funkcií. Vylepšenie spočíva v tom, že sa pred spracovaním funkcie automaticky vytvorí alias, ktorý sa spracuje rovnakým spôsobom, ako v predchádzajúcich prípadoch. Preto, ak alias nezapuzdruje volanie funkcie, automaticky sa vytvorí pred spustením príkazu. To má za následok, že každá funkcia je identifikovaná definíciou aliasu, vypočítanou z hash-u definície funkcie, každá funkcia je volaná iba raz a príkaz sa prepíše v počiatočnej fáze.

Spracovanie príkazov z pohľadu volania funkcií je výkonnostne efektívne, avšak má jednu nevýhodu, ktorá súvisí s prepisom príkazu v počiatočnej fáze. Hodnota Hash plan

value je vypočítaná z rozšíreného príkazu, čo vedie k spracovaniu rôznych vstupov. Pre jeden príkaz sa môže vypočítať viacero hodnôt Hash plan values, pretože môže byť prepísaný do viacerých formátov, a tým aj rôznych obmien vstupu. Preto je potrebné zaviesť nový parameter v session alebo systéme (*def. príkazu 124*):

```
ALTER {SESSION | SYSTEM}
 SET generate_function_alias = {RESTRICTED | IGNORE | MAP | FORCE};
```

Definícia príkazu 124: Parametre session/system

Existujú štyri možnosti, ktoré môže nadobúdať *generate_function_alias*:

- *RESTRICTED* – generovanie ďalších aliasov pre volania funkcií je obmedzené, preto pôvodný príkaz zostáva nezmenený a považuje sa za vstup pre výpočet Hash plan value
- *IGNORE* – spracovanie viacerých hodnôt Hash plan value sa ignoruje a vygenerovaný alias zapuzdruje každé volanie funkcie
- *MAP* – aj keď aliasy funkcie môžu byť definované v pôvodnom dotaze, vždy sa nahradia všeobecnou definíciou získanou pomocou hash-u. Preto sa pôvodný príkaz vždy mapuje do rovnakého formátu. Bez ohľadu na to, či boli definované aliasy, hodnota Hash plan value bude vždy rovnaká
- *FORCE* – definícia aliasu musí explicitne ohraničiť každú funkciu

Princíp tohto riešenia je uvedený v *def. príkazu 125*.

```
-- pôvodný príkaz
SELECT COUNT(*), TO_CHAR(datum_od, 'MMYYYY') datum
 FROM data
  GROUP BY 2;
-- restricted
SELECT COUNT(*), TO_CHAR(datum_od, 'MMYYYY') datum
 FROM data
  GROUP BY 2;
-- ignore
SELECT COUNT(*) AS cis, TO_CHAR(datum_od, 'MMYYYY') datum
 FROM data
  GROUP BY datum;
-- map
SELECT COUNT(*) AS A1, TO_CHAR(datum_od, 'MMYYYY') AS A2
  FROM data
    GROUP BY A2;
-- force
Výnimka – Očakávaný pôvodný format:
SELECT COUNT(*) AS cis, TO_CHAR(datum_od, 'MMYYYY') datum
  FROM data
    GROUP BY 2;
```

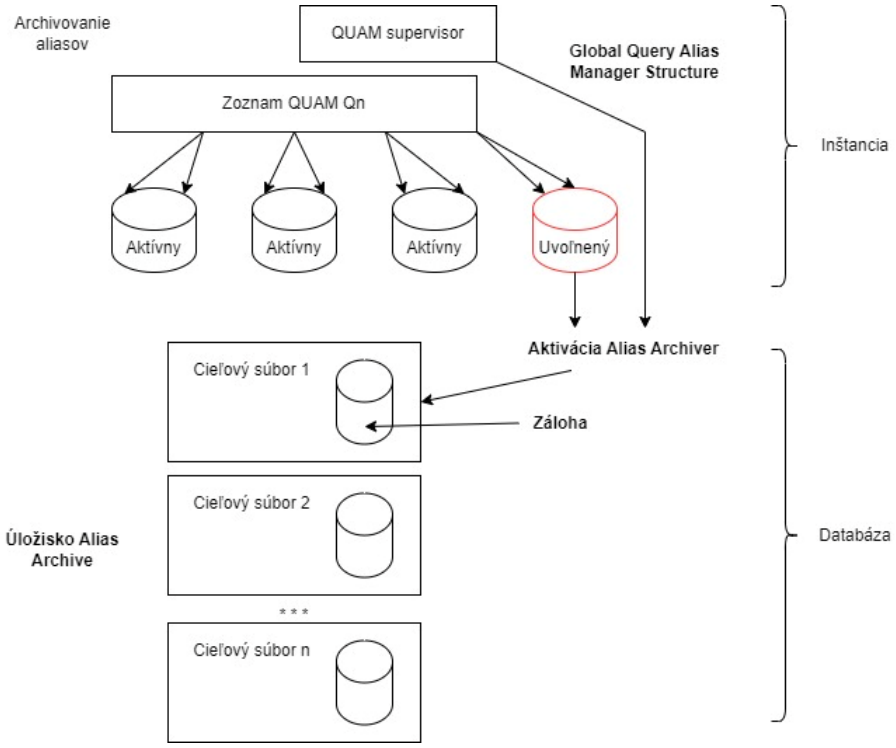
Definícia príkazu 125: Princíp aliasov volania funkcií

Migrácia volaní funkcií

Štruktúry obsahujúce údaje, extrahované aliasy a odkazy na funkcie môžu pozostávať zo stoviek príkazov, ich aliasov, rôznych funkcií, atď. Vytvorenie, údržba a napĺňanie týchto štruktúr, či už umiestnených v pamäti alebo čiastočne v databáze prostredníctvom procesu archivovania aliasov môže predstavovať enormné úsilie. Každopádne, zabezpečenie týchto dát a ich sprístupnenie po obnovení inštancie alebo akomkoľvek zlyhaní je vždy užitočné. Hoci možná strata alebo poškodenie tejto štruktúry nespôsobí stratu dát ani neporuší integritu, náklady na jej opätovné vytvorenie a obnovenie môžu byť príliš vysoké. Preto, Query Alias Manager Structure môže byť zapuzdrená a aktívne spravovaná v rámci transakcií. Je dôležité však poznamenať, že takéto transakcie sú autonómne a nemajú vplyv na pôvodné dáta. Transakcie automaticky zaznamenajú každý nový riadok, ktorý sa má načítať do úložiska. To by však malo dopad na výkonnosť. Preto sa generuje iba REDO a presúva sa do UNDO tabuľkového priestoru. Keďže sa príkazy nevyvíjajú, pre správu aliasov neprebieha žiadna operácia aktualizácie, pretože by to mohlo

viesť ku generovaniu iného plánu vykonania s hodnotou Plan hash value. Preto, nie je potrebné obnovovať pôvodný stav a operácie UNDO môžu byť obmedzené. Vďaka tomu je po každom zlyhaní možné obnoviť štruktúru do stavu, v akom bola pred zlyhaním, pričom dodatočná správa transakcií nemá vplyv na celkový výkon.

Transakčné logy sa zoskupujú a voliteľne sa spracovávajú procesom archivovania logov, rovnako ako štandardné databázové transakcie. Vylepšená správa transakcií pre Query Alias Manager Structure je znázornená na *Obrázku 85*. Obrázok znázorňuje zredukovanú správu transakcií, ktorá dosahuje len štruktúru REDO. Generovanie REDO prebieha v prvej fáze, potom sa uloží nový riadok do Query Alias Manager Structure. Paralelne sa vygenerovaný log uloží do databázy.



Obrázok 85: Vylepšená správa transakcií

Ďalším navrhovaným prístupom je úplne sa vzdať správy transakcií pre štruktúru správy aliasov, čo prináša ušetrenie nákladov a zdrojov. Transakcie nepokrývajú zmeny, takže pre správcu transakcií nevznikajú žiadne ďalšie náklady. Taktiež nie je potrebné generovať a ukladať štruktúru REDO a nie je potrebné aplikovať proces archivovania. Namiesto toho sa všetky zmeny v štruktúre zapisujú rovno do databázy. V takom prípade ide o tzv. zrkadlenie na úrovni disku. Takýmto spôsobom je možné obnoviť dáta zo štruktúry dokonca aj pri kolapse disku. Navyše treba poznamenať, že Query Alias Manager Structure neukladá žiadne špeciálne údaje a celý obsah sa dá opätovne extrahovať z pôvodných dát,

hoci to môže mať vplyv na zvýšenie nákladov. Na zabezpečenie konzistencie a odolnosti voči zmenám a poruchám, je pre štruktúru správy aliasov na úrovni databázy povolená verzia súborov.

Nevýhodou odmietnutia transakcií a fyzickej správy dát v databáze sú duplicity. Nový záznam je vytvorený a uložený do pamäte, ale kópia sa uloží aj do databázy. Z hľadiska efektivity ukladania sa najrelevantnejšie údaje ukladajú a spracovávajú dvakrát.

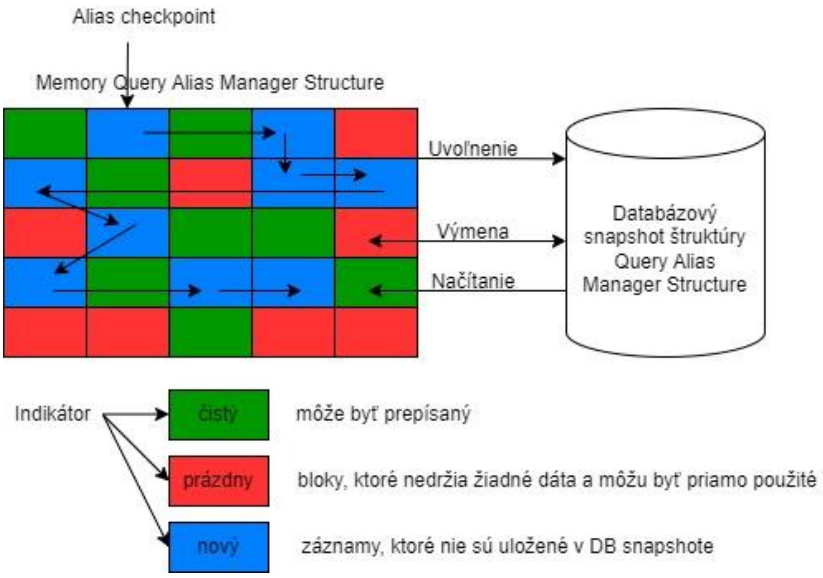
Pravidlo uvoľnenia v prostredí jadra transakcie

Query Alias Manager Structure môže byť v pamäti inštancie, buď v privátnej oblasti alebo zdieľaná medzi používateľmi. V takom prípade musia byť prítomné operácie transakcie REDO, aby bola zabezpečená obnoviteľnosť. V opačnom prípade by sa stratila celá štruktúra po zlyhaní inštancie alebo po reštarte servera. Na druhej strane, ak transakcie nezabezpečujú správu, celá štruktúra sa uloží do databázy a je potrebné I/O načítanie. Okrem toho existujú aj ďalšie požiadavky, zatiaľ čo sa časť štruktúry kopíruje z databázy do pamäte inštancie. Nakoniec sa každý nový riadok primárne uloží do databázy, aby sa zabezpečila jeho použiteľnosť a obnoviteľnosť v prípade poruchy inštancie.

Navrhované možné vylepšenie medzi uchovávaním Query Alias Manager Structure iba v pamäti a fyzickou vrstvou dát ohraničenou výmenou súvisí s pravidlom uvoľnenia v prostredí jadra transakcie. Údaje v pamäti nie sú pokryté transakciami, ale vymenené záznamy sú fyzicky uložené. Väčšina dát je v prípade zlyhania systému chránená pretože sú uložené vo fyzickom úložisku. Avšak, v pamäti sú uložené najnovšie a najčastejšie používané záznamy, ktoré však chránené nie sú. Preto ich nemožno opomenúť. Takýto predpoklad dosiahnutia lepšieho výkonu obmedzením generovania REDO neprináša v prípade zlyhania systému dostatočnú silu.

Checkpointing

Počas optimalizácie štruktúry Query Alias Manager Structure a výmeny medzi pamäťou a diskovým úložiskom sme skúmali optimalizačné techniky na zabezpečenie dostupnosti a obnoviteľnosti, avšak bez špeciálnej podpory transakcií. Navrhované riešenie v tejto časti využíva checkpointing pamäte, pričom navrhovaná architektúra je znázornená na *Obrázku 86*.



Obrázok 86: Checkpointing

Časť pamäte Query Alias Manager Structure je blokovo orientovaná, avšak ide skôr o logický koncept ako fyzický. Každý blok obsahuje extrahovanie len jedného príkazu. Bloky obsahujú indikátory – vymazaný, prázdny a nový. Prázdne bloky sú voľné a môžu byť priamo použité. Sú najvhodnejšie pre ukladanie nových záznamov alebo načítanie z úložiska. Hoci vymazané bloky obsahujú dáta, môžu byť prepísané, pretože obsahujú iba dátovú časť snapshotu databázy. Bloky označené ako nové/načítané obsahujú dáta, ktoré je potrebné uvoľniť do DB snapshotu. V prípade poruchy inštancie by sa takéto údaje mohli stratiť. Takéto záznamy boli v princípe vytvorené nedávno, definovaním nových príkazov na spracovanie a extrahovanie aliasov a funkcií. Tieto bloky sú prepojené a tvoria lineárny zoznam. Smerník Alias Checkpointer ukazuje na prvý výskyt nového bloku. Niektoré nové bloky sa skopírujú do DB snapshotu v definovanej frekvencii a označia sa ako čisté. Alias Checkpointer sa presunie na ďalšiu množinu blokov. Toto zabezpečuje proces s názvom Alias Swapper, ktorý má parametre:

- *check_freq* – ako často sa spustí Alias Checkpoint. Definuje sa časovým rámcom, čo predstavuje čas v sekundách alebo dynamicky, čo predstavuje pomer medzi čistými, prázdnymi a novými blokmi s cieľom mať vždy voľné bloky k dispozícii
- *check_ratio* – definuje počet blokov, ktoré sa majú vymieňať

Parametre sa dajú nastaviť na systémovej úrovni pomocou príkazu, ktorý je definovaný v *def. príkazu 126*.

```
ALTER SYSTEM
 SET check_freq = {hodnota v sekundach | dynamic};
ALTER SYSTEM
 SET check_ratio = {pocet blokov | percentualny pomer};
```

Definícia príkazu 126: Parametre Checkpointingu

Percentuálny pomer definuje pomer medzi všetkými novými blokmi a tými, ktoré budú zálohované. Checkpoint môže byť spustený aj manuálne pomocou príkazu (*def. príkazu 127*).

```
ALTER SYSTEM checkpoint aliases;
```

Definícia príkazu 127: Manuálne spustenie checkpointu

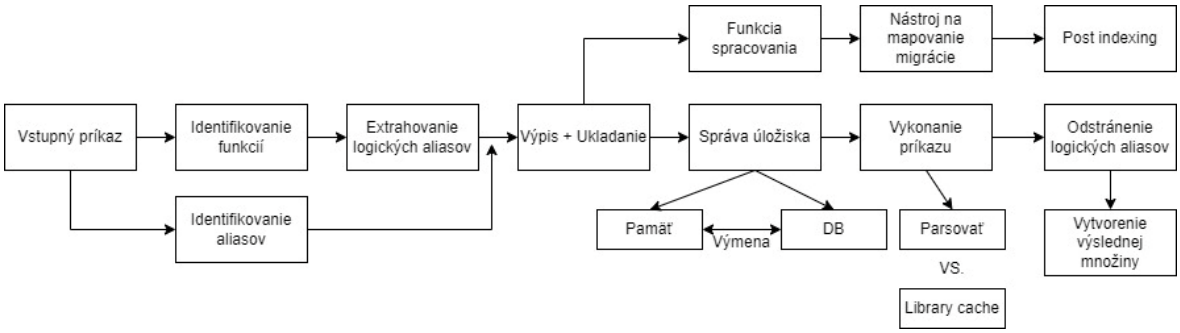
Nesprávne nastavenie parametrov môže spôsobiť konflikty medzi blokmi. V takom prípade, ak nie je k dispozícii žiaden voľný blok, môžu sa prepísať aj nové bloky, aby sa uvoľnilo miesto pre nový príkaz. To však vedie k zhoršeniu výkonu, pretože pre ten istý príkaz sa stratia extrahované aliasy a odkazy volaní funkcií a pri budúcom spracovaní by bolo potrebné ich znova extrahovať.

Zhrnutie navrhovaného riešenia

V rámci výskumu a riešenia tohto problému sme sa zamerali na niektoré techniky a vylepšenia na vybudovanie robustného riešenia pre extrakciu aliasov stĺpcov, výrazov a volaní funkcií. Cieľom je obmedziť volania rovnakých funkcií v rôznych častiach príkazu, do ktorého zaraďujeme aj vnorené príkazy. Zavádzame vlastné identifikátory a logické rámce aliasov pre volania funkcií, vďaka ktorým je možné sa na tieto funkcie priamo odkazovať a spracovávať ich rovnako ako keby mali aliasy. Ide len o logickú definíciu, ktorá sa neodrazí vo formáte výslednej množiny.

Navrhované riešenie začína s identifikáciou a extrakciou aliasov umiestnených v klauzule *SELECT*, po ktorej nasleduje výpis aliasov. Pre tento účel sa zavádza nový proces na pozadí QUAM, ktorý predstavuje hlavný proces dohliadajúci na činnosti worker procesov QUAWn. Extrahované aliasy a volania funkcií sa umiestnia do optimalizovanej dátovej štruktúry - Query Alias Manager Structure, ktorá sa nachádza v pamäti inštancie. Na základe evaluačnej štúdie sme dospeli k záveru, že ukladanie výhradne v pamäti je nepraktické kvôli nárokom na zdroje a kapacitu pamäte. Preto sa štruktúra presunula z privátnej oblasti do zdieľaného úložiska, čo umožňuje zdieľanie rovnakých plánov vykonania a extrahovaných prvkov. Preto sme zaviedli lokálne úložisko dotazov, definované výmenou medzi pamäťou inštancie a databázou. Na zistenie konzistentnosti, spoľahlivosti, odolnosti voči chybám

a obnoviteľnosti sa používali transakcie. To však prinieslo dodatočné nároky na spracovanie, obmedzenie extrahovania aliasov a správy volania funkcií v samostatných autonómnych transakciách. Okrem toho sme zaviedli pravidlá transakcií a checkpointing správy aliasov na zabezpečenie výkonu a robustnosti. Schéma celého riešenia je znázornená na *Obrázku 87*.

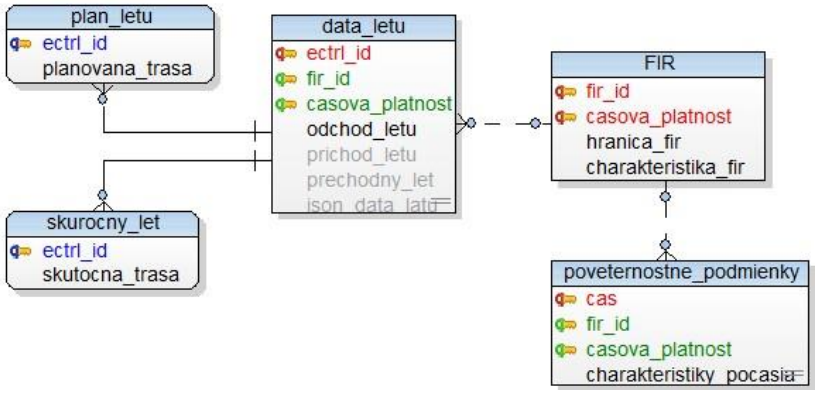


Obrázok 87: Schéma navrhovaného riešenia

3.6.3 Experimentálne hodnotenie výkonu

Viaceré výskumy a štúdie hodnotenia výkonnosti sa zameriavajú na spúšťanie príkazov, optimalizáciu plánu vykonania, po ktorých nasleduje ich uloženie do Library cache v štruktúre pamäte Shared pool. Ukladanie výsledkov volania funkcií môže priniesť výhody, ak sa určitá funkcia s tými istými parametrami vykonáva niekoľkokrát. Ukladanie výsledkov však neprináša až také výhody, ako v prípade extrakcie komplexných funkcií na základe vyhodnotenia doby spracovania a nákladov na vykonanie. V práci [30] sú výsledky funkcií ohraničené virtuálnymi stĺpcami, nie sú fyzicky uložené. Mapovanie funkcií zjednodušuje celý proces vykonávania. Na základe zložitosti sa môžu znížiť nároky na dobu spracovania až o 30%. Ďalšie štúdie [31] a [32] popisujú redukciu prepínania kontextu. Použitie PRAGMA_UDF na optimalizáciu vykonávania funkcií v prostredí SQL prináša len malé zlepšenia, pretože parsovaná funkcia sa na účely vyhodnotenia nachádza v pamäti inštancie [33]. Najperspektívnejšie existujúce riešenie súvisí s funkcionálnymi indexami, ktoré sú založené na B+strome. Kľúčom indexu je samotné volanie funkcie rozšírené o rôzne stĺpce. Listová úroveň obsahuje odkazy na výsledky. Štruktúra je zostavená na základe odkazu na funkciu, ktorá je deterministická. Hlavnou výhodou je priame mapovanie volania funkcie s definovanými parametrami.

Na nasledujúce experimenty boli použité údaje monitorovania letov nad Európou. Množina údajov obsahovala 500 000 letov z rokov 2015 – 2018. Obsahuje údaje o jednotlivých letoch, pozíciách lietadla počas letu, letovej oblasti, vzdušnom priestore a poveternostných podmienkach. Dátový model je znázornený na *Obrázku 88*.



Obrázok 88: Dátový model leteckej dopravy

Väčšina parametrov je uložená vo formáte JSON. V porovnaní s formátom XML sa ušetrí v priemere 5%. Je to spôsobené kontrolou schémy XML, ktorá pri formáte JSON nie je potrebná. Ak by sa však kontrola formátu definície vykonala, boli by potrebné dodatočné 3%. Štruktúra JSON pre lety pozostáva z 50 parametrov. Parametre servera, ktorý bol použitý pri experimentoch sú:

- *Pamäť*: Kingston, typ DDR4, 2x 32GB, 3200 MHz, CL20
- *Úložisko*: 2TB, typ disku NVMe, PCIe Gen3 x 4, 3500 MB/s pre operácie čítania a zápisu
- *OS*: Windows Server 2022, x64
- *DS*: Oracle Database 21c Enterprise Edition Release 21.3.0.0.0

V *Tabuľke 27* sa nachádzajú celkové náklady na spracovanie existujúcich riešení. Hodnoty sú uvedené v percentách, aby sa zdôraznili rozdiely medzi jednotlivými riešeniami. Základné riešenie bez vylepšenia sa berie ako referenčný bod s hodnotou 100%. Tabuľka ďalej obsahuje výsledky štúdií pre virtuálne stĺpce, odkazy na výsledky funkcií v úložisku v databáze, prepínanie kontextu, funkcionálne indexy, atď. Zobrazené výsledky sú viazané na dáta monitorovania letov.

Tabuľka 27: Celkové náklady na spracovanie príkazu v percentách

Podmienka	Náklady %	Optimalizované náklady %
Pôvodné riešenie	100	
Result cache	80.25	64.52
Virtuálne stĺpce	79.61	
Špecifická funkcia v úložisku v databáze	61.62	
Redukcia prepínania kontextu	97.99	

Podmienka	Náklady %	Optimalizované náklady %
Funkcionálne indexy	55.12	50.47
Implicitné mapovanie funkcie	92.01	

Result cache dočasne ukladá viacero funkcií a zohľadňuje rôzne parametre a mapovanie výsledkov. Avšak, nie všetky kombinácie parametrov použité pre určitú funkciu sú uložené. Na druhej strane, virtuálny stĺpec vezme všetky výskyty, ale len jednej funkcie. Medzi týmito dvoma riešeniami je len malý rozdiel a to je 0.64%. Avšak, kombináciou týchto dvoch riešení by sa celkové náklady znížili o viac ako 12% a to na hodnotu 64,52%. S využitím uloženia množiny odkazov na funkciu v špeciálnom úložisku databázy sa celkové náklady na spracovanie znížia na hodnotu 61.62%. V porovnaní s pôvodným riešením, ide o zlepšenie o 38.38%. Treba však povedať, že sa ukladajú všetky odkazy na funkcie, aj keď sa už neskôr nebudú používať. To vedie k dodatočným nákladom na diskový priestor, databázu a I/O operácie. Na základe tejto štúdie sú celkové požiadavky pri monitorovaní letov, extrakcii dátumu a sledovaní parametrov lietadla viac ako 10-násobné. Ak sa úložisko zredukuje na minimálne tri použitia, náklady sú 62.74%, čo je však stále lepšie ako kombinácia Result cache a virtuálnych stĺpcov. Rozdiel 1.12% sa týka dodatočných požiadaviek na spustenie funkcie s rovnakými parametrami viackrát. Redukcia prepínania kontextu prináša malé zlepšenie, súvisí to hlavne s tým, že v tomto prostredí je malý súbor používaných funkcií, ktoré sa líšia v zložitosti a použitých parametroch. Najlepšie riešenie dosiahol funkcionálny index. Vyhľadávanie v množine údajov je oveľa efektívnejšie ako virtuálny stĺpec alebo Result cache. S vylepšením tohto riešenia o Result cache sa náklady tohto riešenia znížia na hodnotu 50.47%. Ak sa však nachádza v Buffer cache, celkové náklady sa môžu znížiť na hodnotu 46.01%. Limitovaním tohto prístupu je spoľahlivosť spracovania a mapovanie dátových typov. Vyžaduje si to totiž presnú špecifikáciu dátových typov na oboch stranách podmienky, to znamená, že oba dátové typy musia byť rovnaké, aby sa zabezpečila správna výkonnosť. V opačnom prípade sa na pozadí aplikuje implicitná konverzia. To však znamená, že sa pôvodný funkcionálny index nepoužije a bolo by potrebné to rozšíriť o funkciu konverzie. Celkové náklady na spracovanie však stúpnu na 92.01%, čo predstavuje obrovský nárast oproti funkcionálnym indexom. Ak je index povolený, definícia a spracovanie sa môžu týkať len ukladania hodnôt do Result cache. Určité vylepšenie sa dá použiť, ak je na jednej strane podmienky funkcia a na druhej strane sa nachádza napríklad konštanta alebo bežný stĺpec tabuľky. Potom môže funkcia zostať pôvodná a konverzná funkcia sa aplikuje na bežnú hodnotu. V takom prípade

sa môže využiť hint (nápoveda) tzv. non-convert funkcia. Je možné to demonštrovať na príklade (*def. príkazu 128*):

```
SELECT /*+no-convert*/
 FROM ...
   WHERE TO_CHAR(datum_od, 'MM')=12;
```

Definícia príkazu 128: Príklad využitia non-convert funkcie

Ľavá strana podmienky produkuje reťazec znakov, zatiaľ čo pravá strana reprezentuje numerickú hodnotu. Hint non-convert naviguje systém na konverziu pôvodnej hodnoty stĺpca, nie na volanie funkcie. Na pozadí to vyzerá nasledovne (*def. príkazu 129*):

```
TO_NUMBER(TO_CHAR(datum_od, 'MM'))    = 12;           -- odmietne sa
TO_CHAR(datum_od, 'MM')               = TO_CHAR(12);  -- aplikuje sa
```

Definícia príkazu 129: Konverzia na pozadí

Funkcionálny index sa môže ďalej používať pomocou non-convert nápovedy. Navyše, konverzná funkcia sa môže považovať za bežnú funkciu, ktorá odkazuje na Result cache. Dodatočné náklady na spracovanie sa pohybujú v rozmedzí od 2% do 3% v závislosti od pôvodného a cieľového typu.

Uvedený experiment sleduje celý let, pričom sa získavajú jeho parametre. Dotaz obsahuje niekoľko funkcií na získanie dátumu a času letu, transformáciu JSON-u na text, sledovanie vplyvu počasia, atď. Uvedený dotaz je zobrazený v *def. príkazu 130*.

```
SELECT /*+no-convert*/
        TO_CHAR(odchod_letu, 'DD.MM.YYYY HH24:MI:SS'),
        TO_CHAR(prichod_letu, 'DD.MM.YYYY HH24:MI:SS'),
        EXTRACT(hour from prechodny_let),
        EXTRACT(minute from prechodny_let),
        EXTRACT(second from prechodny_let),
        monitor_flight(json_data_letu, ECTRL_ID),
        get_FIR_assignment(ECTRL_ID, prechodny_let)
 FROM data_letu
   JOIN FIR ON(fir_borders(lon, lat)=hranica_fir)
   JOIN poveternostne_podmienky(cas)
    WHERE TO_CHAR(odchod_letu, 'YYYY')=2018;
```

Definícia príkazu 130: Príkaz sledovania letu

Druhá výskumná časť sa zaoberá existujúcou správou aliasov a prináša vylepšenia v Oracle 23c. Používa päť riešení súvisiace s odkazovaním sa na volanie funkcií a referencie

aliasov. Riešenie REF nezohľadňuje aliasy a na funkcie sa odkazuje viackrát, je vylepšené o využitie Result cache, takže funkcia sa pre danú sadu atribútov zavolá len raz. Riešenie R1 využíva v prvom kroku vnorenie príkazov výpočtom výstupov funkcií, potom nasleduje ich priame použitie vo vonkajšom dotaze. Takto, jednotlivé klauzuly vonkajšieho dotazu odkazujú na priame hodnoty, ako na atribúty získané z databázy. Riešenie R2 je založené na predspracovaní pomocou dynamického pohľadu (s rozšírením klauzúl zavedenými v Oracle 12c Release, ktoré umožňujú vylepšiť klauzuly o faktorovanie poddotazov, využitie kľúčového slova *DETERMINISTIC* alebo *PRAGMA_UDF*). Riešenie R3 využíva materializovaný pohľad pre každú tabuľku. Riešenie R4 používa vylepšenie zavedené v Oracle Database 23c, a to je používanie aliasov. Vylepšenie spočíva v tom, že je možné odkazovať sa na definované aliasy v ľubovoľnej klauzule toho istého príkazu. *Tabuľka 28* zobrazuje výsledky času spracovania a vplyv implicitných konverzií na reťazce a číselné hodnoty. Vplyv implicitných konverzií sa pohybuje v rozmedzí od 6.28% do 13.47%.

Tabuľka 28: Výsledky - vplyv implicitnej konverzie

		REF	R1	R2	R3	R4
	Náklady	3660 (24% CPU)	3660 (24% CPU)	3660 (24% CPU)	2562 (16% CPU)	3660 (24% CPU)
	Čas spracovania [ss.ff]	23.25	27.33	21.78	14.46	18.24
to_number implicitná konverzia	Čas spracovania [ss.ff]	24.71	29.36	23.76	15.90	19.74
to_char implicitná konverzia	Čas spracovania [ss.ff]	25.87	31.01	23.96	16.21	20.06

Existujúce riešenia poskytujú relevantnú výkonnostnú vrstvu pre správu dát a odkazy na funkcie. Väčšina opísaných riešení zlepšuje výkonnosť poklesom doby spracovania, s výnimkou riešenia R1 založeného na vnorení príkazov. V prvom rade si to vyžaduje parsovanie, spracovanie, manipuláciu a vyhodnotenie ďalších príkazov. Nevýhodou tohto prístupu je extrakcia podmienky *WHERE* do samostatnej fázy spracovania. Preto sa faktor redukcie dát nedá okamžite použiť, a preto je potrebné spracovať veľké množstvo údajov. Riešenia R2 a R3 využívajú pohľady, pričom najlepšie riešenie súvisí s materializovaným pohľadom, v ktorom sú výsledky funkcie vždy vypočítané. Platí to, však, len pre proces získavania dát, takže príkaz *SELECT*. Pri akejkoľvek operácii zmeny je potrebné rýchle obnovenie samotného materializovaného pohľadu. Riešenie R4 rozširuje princípy jazyka SQL tak, že umožňuje odkazovanie na aliasy v ľubovoľnej klauzule príkazu. Nie je potrebné vnáranie dotazov ani prepočty. Namiesto toho sa definície aliasov extrahujú pred spracovaním príkazu.

Ďalší výskum sa zaoberá výkonnosťou navrhovaných riešení, pričom zdôrazňuje dva aspekty. Prvý aspekt je monitorovanie celého letu od vzletu po pristátie, rolovanie a parkovanie (ES1), druhý aspekt sa zameriava na získanie pozície lietadla v definovanom čase (ES2). Zároveň sa snaží získať parametre letu, stav, priradenie do FIR a poveternostnú situáciu. Množina údajov a parametre prostredia sa nemenia.

Tabuľka 29 zobrazuje výsledky fázy ES1 s dôrazom na jednotlivé časti navrhovaného riešenia. Každé hodnotenie sa vykonalo desaťkrát pre 100 letov. Výsledky predstavujú priemerné hodnoty. Referenčným riešením je rozšírená správa aliasov zavedená v databáze Oracle 23c, ktorá je vylepšená o Result cache pre 20% najčastejšie volaných funkcií. Na základe predchádzajúcich výsledkov sa zdá, že pridanie vyššieho percenta na ukladanie výsledkov neprináša žiadne významné benefity kvôli nárokom na veľkosť. Príliš vysoká hodnota Result cache čoskoro dosiahne kapacitu celej pamäte. Avšak Result cache štruktúry pamäte Shared pool je len malou časťou požadovanej pamäte inštancie. Jednotlivé časti výskumu navrhovaných riešení na seba nadväzujú. Hodnotenie prebieha v štyroch fázach:

- fáza 1 – identifikácia, extrakcia a výpis aliasov
- fáza 2 – lokálne úložisko dotazov
- fáza 3 – výmena štruktúry extrahovaných aliasov a funkcií medzi pamäťou inštancie a databázou prevádzkovaná QUAM a jeho workermi. Celá štruktúra je zdieľaná medzi session v celej inštancii
- fáza 4 – zlepšenie systému pomocou virtuálnych logických aliasov spracovávajúcich funkcií

Tabuľka 29: ES1 - výsledky výkonnosti - fázy

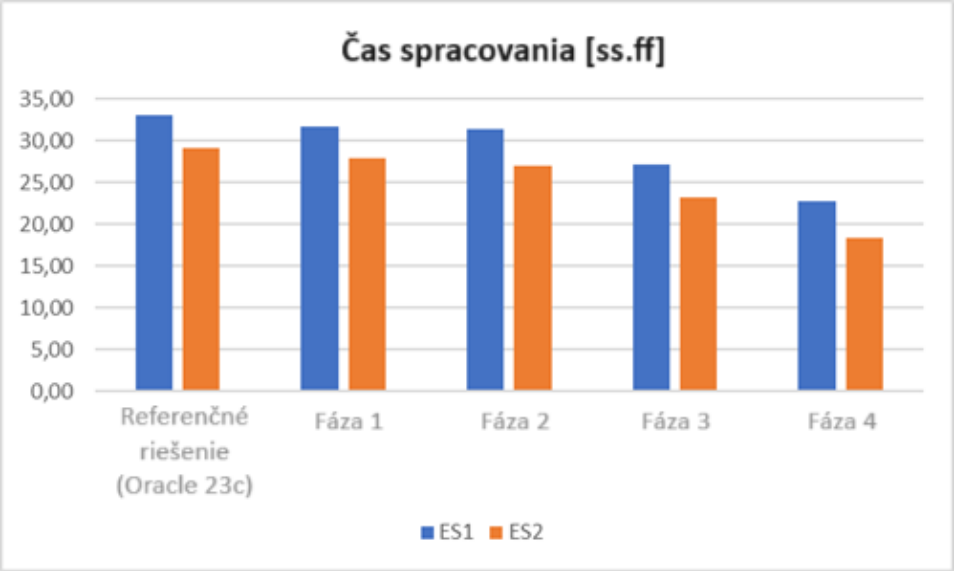
ES1	Referenčné riešenie (Oracle 23c)	Fáza 1	Fáza 2	Fáza 3	Fáza 4
Čas spracovania [ss.ff]	33.12	31.77	31.45	27.01	22.07
Čas spracovania [%]	100	95.92	94.96	81.55	66.64

ES2 pracovalo s rovnakými podmienkami a spracovávalo rovnaké množstvo údajov ako v prípade ES1, avšak v inom formáte a štruktúre. Ako je možné vidieť v *Tabuľke 30*, nároky na spracovanie času sú takmer rovnaké a rozdiely medzi jednotlivými technikami a fázami sú analogické.

Tabuľka 30: ES2 - výsledky výkonnosti - fázy

ES2	Referenčné riešenie (Oracle 23c)	Fáza 1	Fáza 2	Fáza 3	Fáza 4
Čas spracovania [ss.ff]	29.12	27.09	26.93	23.26	18.42
Čas spracovania [%]	100	93.03	92.48	79.87	63.26

Fáza 1 je počiatočná fáza pre úložisko obsahujúce extrahované aliasy a ich údržbu. V porovnaní s ES1 prináša zlepšenie o 4% a v porovnaní s ES2 je zlepšenie o 7%. Lokálne ukladanie údajov neprináša výrazné benefity, pretože tie isté príkazy, plány vykonania a funkcie môžu byť zdieľané viacerými používateľmi. Výrazný pokles nákladov na spracovanie sa prejavilo vo fáze 3, ktorá zabezpečuje výmenu. Pre ES1 pokles predstavuje 13.41% v porovnaní s fázou 2. Celkovo to predstavuje pokles nákladov na spracovanie o 18.45%. Analogicky pre ES2 ide o pokles nákladov na spracovanie o 12.61% v porovnaní s fázou 2. V porovnaní s referenčným riešením zavedeným v Oracle 23c sa dosiahne pokles o 20.13%. Poslednú fázu pokladáme za najsľubnejšiu, v prípade, ak definície aliasov nezapuzdrujú funkcie. Na základe toho môžu virtuálne logické aliasy zlepšiť spracovanie pre ES1 až o 33.36% a o 36.74% pre ES2. *Obrázok 89* znázorňuje stĺpcový graf, ktorý obsahuje čas spracovania príkazu v sekundách pre jednotlivé riešenia a fázy.



Obrázok 89: Porovnanie existujúceho riešenia s navrhovaným vylepšením (v sekundách)

Limitovaním vyššie uvedených riešení je však spoľahlivosť a konzistencia. Ak je napríklad funkcia zdieľaná v celej inštancií, musí sa vždy skontrolovať referencia a vlastník funkcie, aby sa zabezpečilo správne vykonanie volania. Pre tento účel sa v navrhovanej štruktúre ukladá referencia vlastníka. Bolo by však potrebné ukladať viac informácií, pretože

hlavička funkcie môže byť rozšírená o definíciu schémy autorizácie, ktorá nie je priamo prítomná pri parsovaní dotazu. Tento aspekt by bolo potrebné zvážiť pri ďalšom výskume. Okrem toho existuje veľa budúcich stratégií na preskúmanie, ako napríklad nastavenie pravidiel na minimalizáciu možných odkazov na funkcie po reštarte inštancie, zabezpečenie správneho výkonu, presun dát medzi pamäťou inštancie a fyzickým úložiskom.

3.7 Spracovanie odkazov temporálnych funkcií v relačných databázach

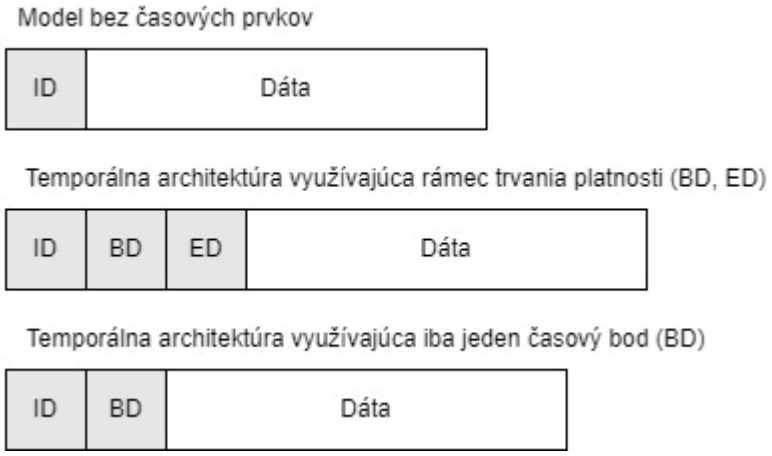
Počas vývoja databázových technológií a paradigmy SQL bol zavedený koncept temporálnych databáz. Odvtedy je možné ukladať jednotlivé stavy objektov v čase a sledovať ich zmeny. Časové rámce teda zapuzdria každý riadok dát, pričom najčastejšie vyjadrujú platný rozsah. Následne možno jeden stav objektu vyjadriť viacerými riadkami. Platný môže byť však len jeden stav. Temporálne (časové) databázy sú databázy, ktoré odkazujú na samotný objekt a jeho viaceré verzie počas definovaného časového rozsahu. Samotný identifikátor slúžiaci na unikátnosť záznamu v nich preto nestačí. Definícia stavu sa skladá z troch prvkov – identifikácia objektu, časového rozsahu platnosti a samotných dát. V tom sa líšia od klasických databáz [63].

S vývojom objektov a jednotlivých reprezentácií stavov je potrebné riešiť aj odkazy na funkcie a ich vývoj v čase. Preto sme sa bližšie zamerali na riešenia správy funkcií, ktorých kód sa môže v priebehu času meniť, a to nie len z hľadiska optimalizácie kódu, ale aj z hľadiska produkcie výstupov. Výpočty a spracovanie dát sa môžu vyvíjať a pre rovnaké vstupy môžu produkovať rôzne výstupy na základe časových verzií funkcie. Naším cieľom je poskytnúť robustné riešenie transformácie a mapovania, ktoré kladie dôraz na jednotlivé verzie funkcií a volá relevantné funkcie na základe odkazovaných časových rámcov. Vďaka poskytnutej spoľahlivej transformačnej vrstve môže existovať viacero verzií funkcií, pričom sa rešpektuje platná verzia funkcie, ktorá existovala v čase platnosti príslušnej n-tice [64] [65]. Navrhované riešenie je založené na správe multiverzií prostredníctvom nových klauzúl, príkazov a procesov na pozadí, ktoré autonómne spravujú celý systém. Navyše jednotlivé verzie môžu byť neskôr vylepšené o optimalizácie výkonu, ako sú sofistikovanejšie dátové štruktúry, vďaka čomu môže mať každá verzia viacero implementácií.

Celý popis štúdie a navrhovaného riešenia spolu s ďalšími podrobnosťami výskumu je možné nájsť v publikácii [HD8].

V prvej fáze výskumu sme sa zamerali na analýzu existujúcich prístupov k verziovaniu funkcií na úrovni databázy, pričom sa snažíme zistiť, aké sú obmedzenia súčasných riešení. Cieľom je vytvoriť systém verziovania.

Architektúra temporálnych systémov na úrovni objektu je znázornená na *Obrázku 90*. Identifikátor objektu v temporálnych databázach je rozšírený o aspoň jednu časovú dimenziu, ktorá charakterizuje platnosť. Môže existovať aj viacero časových dimenzií, ako je napríklad platnosť transakcie, čo umožňuje správne uchovávať už uložené a overené transakcie [66]. Jedna dimenzia môže byť definovaná dvoma časovými bodmi – začiatočným (BD) a koncovým bodom (ED). Ak sa použije len jeden bod (BD), ktorý vyjadruje začiatok platnosti alebo použiteľnosti, potom každý nasledujúci stav hraničí s priamym predchodcom.



Obrázok 90: Architektúra temporálnych systémov na úrovni objektu [67]

Hlavnou nevýhodou architektúry temporálnych databáz na úrovni objektu je efektivita ukladania. Môže sa uchovávať veľa duplicitných hodnôt, najmä pri operácii *UPDATE* môže dôjsť k ponechaniu pôvodných hodnôt atribútov, pretože tieto atribúty sa musia skopírovať do nového stavu. Takže aj požiadavky na úložisko môžu byť značne vysoké, zatiaľ čo informačná hodnota dát zostáva rovnaká [68] [69].

Existuje ďalší prístup temporálnych databáz a ten sa nazýva systém orientovaný na atribúty [67]. Tento systém spočíva v priradení časových dimenzií ku každému atribútu samostatne. Výhoda tohto riešenia spočíva v schopnosti spracovať každý atribút samostatne pomocou operácie Update. Takto nevznikajú žiadne duplicitné n-tice a dochádza k optimalizácií databázovej vrstvy. Na druhej strane vznikajú dodatočné nároky na dáta kvôli časovým rámcom, na ktoré sa odvoláva každý atribút. Navyše získanie ľubovoľného stavu si vyžaduje kombinovanie jednotlivých hodnôt atribútov.

Obe uvedené riešenia majú svoje výhody, ale aj niekoľko nevýhod, ktoré obmedzujú výkonnosť. Je zrejmé, že zvolené riešenie by malo úzko súvisieť s predpokladaným pracovným zaťažením, čo môže byť problematické, ak sa štruktúra a vlastnosti dát v priebehu času menia.

V temporálnom prostredí je potrebné sa venovať aj odkazom na funkcie. Definícia, implementácia a výsledok funkcie sa môžu v priebehu času meniť na základe nejakých predpisov. Napríklad metóda výpočtu mzdy a dane sa môže vy určitých časových obdobiach meniť, pričom funkcia zostáva rovnaká (parametre, hlavička, ...), ale jej vnútorné pravidlá sa menia.

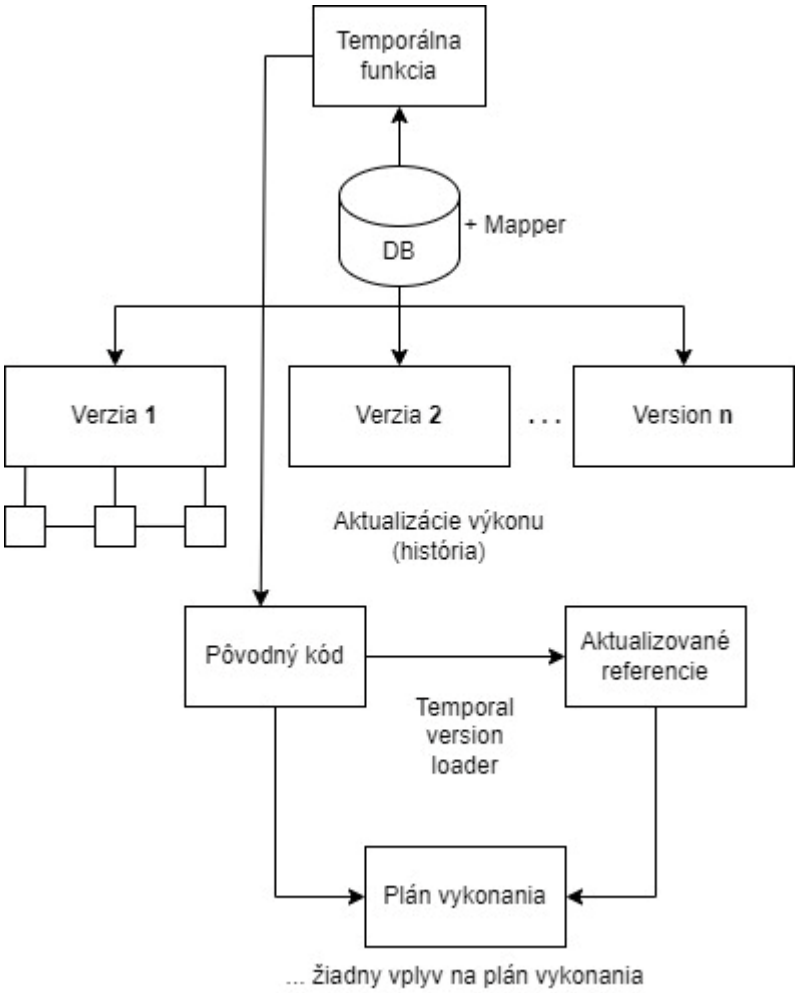
3.7.1 Navrhované riešenie

V našom navrhovanom riešení sa primárne snažíme zabezpečiť ***vysokú mieru paralelizmu***, ktorý sa snažíme dosiahnuť obmedzením nutnosti parsovania, načítavania a transformácie medzi verziami, ***dynamickému mapovaniu verzií***, pri ktorom zostáva hlavička jednej temporálnej funkcie rovnaká pre všetky verzie, ***identifikácii a registrácii verzii*** na obmedzenie testovacích prípadov, ***aktualizácii a implementácii verzii*** na zlepšenie výkonu. Cieľom tohto riešenia je zabezpečiť aby sa vždy vykonala najefektívnejšia implementácia danej verzie a ***registrácii novej verzie funkcie***, ktorá sa stane platnou v definovanom časovom bode.

Na *Obrázku 91* sa nachádza architektúra navrhovaného riešenia. Jednotlivé verzie sú kontrolované procesom Temporal_version_leader na pozadí. Tento proces zabezpečuje načítanie verzie, ako aj správne mapovanie implementácie jednotlivých verzií, pričom sa kladie dôraz na ich platnosť. Navyše sa tu nachádza proces Temporal_registrator, ktorý je zodpovedný za zaznamenávanie novej verzie v temporálnej vrstve riadiacich funkcií. Okrem toho kontroluje konzistenciu časových rámcov, aby sa obmedzilo prekrývanie verzií v čase. Správca temporálnych funkcií poskytuje:

- Registráciu temporálnej funkcie
- Registráciu novej verzie (platná okamžite, platná v budúcnosti)
- Registráciu zmien implementácie verzie
- Kompiláciu po registrácii novej verzie / zmeny implementácie verzie
- Zakázať VŠETKO – to znamená, že temporálna funkcia už nebude prijímať žiadne nové verzie ani zmeny v implementácii danej verzie

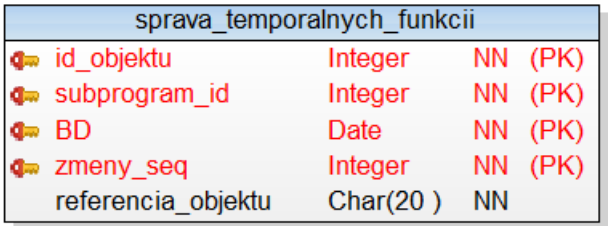
- Zakázať verziovanie – to znamená, že temporálna funkcia už neprijíma žiadne nové vezie, ale implementácia existujúcej verzie sa môže zmeniť, ak to má vplyv na zlepšenie výkonu
- Zlúčenie verzií – táto funkcionalita sa použije v prípade, ak sa kód niekoľkokrát kompiluje, čím sa vytvárajú nové verzie, ak je kód rovnaký, tak sa tieto verzie môžu zlúčiť dohromady



Obrázok 91: Navrhovaná architektúra verziovania temporálnych funkcií

Na O*brázku 92* je znázornená databázová tabuľka registrácie temporálnych funkcií. Primárny kľúč tejto tabuľky pozostáva z *id_objektu* a *subprogram_id,* potom nasleduje časová referencia, ktorá definuje začiatok platnosti. Následne každý nový stav automaticky obmedzuje platnosť predchádzajúcej verzie. Môže nastať situácia, kedy by mala byť verzia použitá počas definovaného časového rámca, ale nasledujúca verzia nie je ešte k dispozícii. Pre zabezpečenie správneho časového ohraničenia (ED) sa zaregistruje verzia funkcie, ktorá vyjadruje, že kód nie je k dispozícii, ale existujúca verzia sa už nemôže použiť. Ďalší atribút

predstavuje sekvenciu pre kontrolu výkonu zmien implementácie. Ako posledný atribút sa v tabuľke nachádza odkaz na objekt parsovanej verzie zdrojového kódu. Tým sa syntaktické a sémantické oprávnenia kontrolujú iba raz, počas počiatočného parsovania a načítavania. Medzi nimi je uvedený aj vlastník objektu a metóda authid.



Obrázok 92: Databázová tabuľka *sprava_temporalnych_funkcii*

Atribúty *id_objektu, subprogram_id* reprezentujú identifikátor metódy, BD predstavuje prvú temporálnu úroveň a reprezentuje začiatok platnosti, *zmeny_seq* predstavuje druhú temporálnu úroveň a spustiteľná verzia funkcie je dostupná prostredníctvom atribútu *referencia_objektu*.

V tomto prípade ide o tzv. bi-temporálny model, zaoberajúci sa platnosťou verzií (prvá temporálna vrstva) a odráža zmeny výkonu v implementácii (druhá temporálna vrstva). Môže sa však stať, že existujúca verzia je neskôr identifikovaná ako nesprávna a musí byť nahradená. Systém preto musí zaznamenať informáciu o tom, že v minulosti bola uvoľnená nesprávna verzia, pretože by sa mohla použiť na analýzu, hodnotenie a spracovanie. Preto je možné toto navrhované riešenie rozšíriť na multi-temporálny systém identifikovaním troch temporálnych dimenzií, ktorými sú platnosť verzie, zmeny výkonu implementácie a opravy verzií.

Vplyv na štruktúru databázy

Hlavnou výhodou navrhovaného riešenia je využívanie logických identifikátorov verzií, ktoré sú súčasťou adresy referencie objektu. Vďaka tomu zostáva názov objektu rovnaký. Používateľský kód sa vôbec nemení. Takto zostávajú existujúce plány vykonania zaregistrované pre definované príkazy a nemusia sa vyhodnocovať. Navyše samotný kód nie je závislý od verzie, pretože správna verzia sa aplikuje dynamicky na základe dát. Preto, ak sa uvoľní nová verzia funkcie, nemá to žiaden vplyv na pôvodný zdrojový kód. Ďalší aspekt sa týka optimalizácie štruktúry na zlepšenie výkonu. Vo väčšine prípadov sa najnovšia metóda považuje za najefektívnejšiu, preto sa na vykonanie vyberie verzia s maximálnym *id_sekvencie*.

Databázová vrstva a optimalizácia majú priamy prístup ku všetkým verziám funkcií, takže nie je potrebné parsovanie, lokalizovanie a načítanie. Všetky požadované údaje sú už prítomné v tabuľkách databázového systému a priamo dostupné pomocou uloženého smerníka verzie objektu. Preto je jedinou činnosťou načítanie obsahu verzie funkcie z databázy do pamäte inštancie, ak sa tam ešte nenachádza. Rôzne vlastnosti sa môžu získať z databázových tabuliek *DBA_PROCEDURES* alebo vo všeobecnosti z *ALL_OBJECTS*, na základe typu podmienky. *Def. príkazu 131* znázorňuje získanie všetkých informácií z týchto tabuliek. Prvý príkaz *SELECT* využíva tabuľku *DBA_PROCEDURES* a druhý príkaz čerpá informácie z tabuľky *ALL_OBJECTS*.

```
SELECT *
  FROM dba_procedures;

SELECT  *
  FROM all_objects
   WHERE object_type
    IN ('PROCEDURE', 'FUNCTION', 'PACKAGE');
```

Definícia príkazu 131: Získanie vlastností z tabuliek dba_procedures a all_objects

Špecifikácie parametrov sa môžu získať pomocou pohľadu *DBA_ARGUMENTS*. *OBJECT_NAME* reprezentuje názov root metódy a *PACKAGE_NAME* reprezentuje názov balíka. *Def. príkazu 132* zobrazuje príkaz *SELECT*, pomocou ktorého získavame všetky atribúty spolu s typom objektu.

```
SELECT a.*, object_type
 FROM dba_arguments a
  JOIN dba_objects o ON(a.object_id=o.object_id)
 WHERE object_type
  IN ('PROCEDURE', 'FUNCTION', 'PACKAGE');
```

Definícia príkazu 132: Získanie špecifikácie parametrov

OBJECT_ID je jedinečný identifikátor metódy. *ARGUMENT_NAME* odkazuje na parametre, ktoré sú oddelené atribútom *DATA_TYPE* a *POSITION*. Kladná hodnota atribútu *POSITION* definuje poradie parametra na vstupe. Hodnota 0 atribútu *POSITION* odkazuje na typ, ktorý sa má vrátiť. V takom prípade má *ARGUMENT_NAME* hodnotu NULL. Kompletný zoznam jednotlivých stĺpcov, dátových typov a popisov je možné nájsť v [35].

Príkazy na registráciu a implementáciu

Pri aplikácii riešenia sa môžu použiť tri úrovne – systémová úroveň, úroveň session a úroveň objektu. Je potrebné však nastaviť nasledujúce dva parametre:

- *temporal_registraion* – ako predvolenú hodnotu využíva FALSE, vyjadruje správanie novej definície funkcie, ak je tento atribút nastavený na FALSE, tak nová funkcia nevytvára jednotlivé verzie a je považovaná za konvenčnú (ak sa má skompilovať nová verzia, nahradí pôvodnú verziu)
- *temporal_versioning* – je použiteľný pre temporálnu funkciu, ako predvolenú hodnotu využíva TRUE, čo znamená, že každé nové prekompilovanie metódy automaticky vytvorí novú verziu, ak je nastavená na FALSE, nová verzia sa musí explicitne zadať a registrovať

Presnosť na úrovni servera je vždy nastavená, prípadne sa použijú prednastavené hodnoty. Ak nie sú nastavené hodnoty na úrovni session, zdedia sa z hodnôt servera. Podobný prístup sa využíva aj na úrovni objektu, pričom objekt je charakterizovaný samotnou metódou. Nastavenie na úrovni objektu môže byť definované počas kompilácie pomocou kľúčového slova *PRAGMA*, alebo sa môže použiť príkaz ***ALTER OBJECT ... COMPILE***, ktorý je možné vylepšiť nastavením parametrov objektu.

Registrácia funkcie

Použitie kľúčového slova *PRAGMA_TEMPORAL_REG* zabezpečuje, že novovytvorená funkcia sa zaregistruje a bude vytvárať jednotlivé verzie. Ak už funkcia v systéme existuje a má minimálne jednu verziu, daná PRAGMA sa ignoruje, pretože už bolo nastavené registrovanie temporálnych verzií. Pomocou tejto definície kľúčového slova PRAGMA je možné transformovať konvenčné (neverziované) funkcie na temporálne. Po technickej stránke táto PRAGMA vyvolá na pozadí proces, ktorý kontroluje rámec časovej platnosti a registruje funkciu. Vytvorenie funkcie s využitím kľúčového slova PRAGMA je možné vidieť v *def. príkazu 133*.

```
CREATE OR REPLACE FUNCTION nazov_funkcie
 RETURN datovy_typ
 IS PRAGMA_TEMPORAL_REG;
   [definícia lokálnych premenných]
 BEGIN
  -- telo funkcie
 END;
/
```

Definícia príkazu 133: Registrácia funkcie

Registrácia verzie

Nie vždy je pri vývoji potrebné vytvárať nové verzie. Dokonca niektoré verzie môžu byť automaticky nahradené bez toho, aby sa to prejavilo z hľadiska času. Aj to je jeden zo spôsobov optimalizácie existujúceho kódu bez ohľadu na zmenu mapovania vstupnej a výstupnej hodnoty. Novú verziu je možné zaregistrovať pomocou kľúčového slova *PRAGMA_VERSION_REG*. Na nasledujúcom kóde (*def. príkazu 134*) je možné vidieť registráciu verzie.

```
CREATE OR REPLACE FUNCTION nazov_funkcie
 RETURN datovy_typ
 IS PRAGMA_VERSION_REG;
   [definícia lokálnych premenných]
BEGIN
 -- telo funkcie
END;
/
```

Definícia príkazu 134: Registrácia verzie

Zlúčenie verzií

Môže sa stať, že sa rovnaký kód skompiluje viackrát, čo vedie k vytvoreniu viacerých verzií. Vďaka príkazu (*def. príkazu 135*) je možné jednotlivé verzie zlúčiť, pričom sa predpokladá že všetky vymenované verzie sú rovnaké. Zachová sa prvá verzia a ostatné sa odstránia. Jednotlivé verzie sa dajú identifikovať pomocou názvu alebo sa môže uviesť časový rámec platnosti.

```
ALTER FUNCTION nazov_funkcie MERGE VERSIONS
 [ {(zoznam verzií) | (BD, ED)} ];
```

Definícia príkazu 135: Zlúčenie verzií

Je potrebné si všimnúť, že sa spájajú len tie verzie, ktoré sú úplne pokryté časovým rámcom platnosti, ako zobrazuje *def. príkazu 136*.

```
SELECT verzie
 FROM sprava_temporalnych_funkcii
  WHERE process_temporal_frame BETWEEN BD AND ED;
```

Definícia príkazu 136: úplné pokrytie časovým rámcom platnosti

Funkcia *process_temporal_frame* vytvára rozsah platnosti pre konkrétnu verziu funkcie. Používa timestamp. Koniec platnosti verzie sa vypočíta na základe identifikácie nasledujúcej verzie, ktorá obmedzuje platnosť predchádzajúcej verzie.

Správa testovacích verzií

Testovacie verzie sú špecifickou kategóriou manažmentu temporálnych funkcií. Ak je kód označený ako TEST, potom sa s ním nepracuje ako s temporálnym, neregistruje sa žiadna nová verzia ani aktualizácia výkonu. Primárne sa využíva vo vývojovom prostredí na kontrolu správnosti nového kódu. Takúto funkciu je možné skompilovať, ale ostáva v testovacej verzii. Nasledujúci kód (*def. príkazu 137*) zobrazuje vytvorenie testovacej verzie.

```
CREATE OR REPLACE FUNCTION nazov_funkcie
 RETURN datovy_typ
 IS PRAGMA_TEST;
   [definícia lokálnych premenných]
 BEGIN
  -- telo funkcie
 END;
/
```

Definícia príkazu 137: Testovacia verzia

Rekompilácia

Už vytvorený a skompilovaný kód sa dá vylepšiť temporálnou registráciou pomocou temporálnych alebo verziovacích možností. Kľúčové slovo *TEMPORAL_REG* požiada systém, aby registroval funkciu ako temporálnu a vytvoril verziu, zatiaľ čo pomocou kľúčového slova *VERSION_REG* sa vytvorí len nová verzia pre funkciu, ktorá je už

označená ako temporálna. Nasledujúci kód (*def. príkazu 138*) znázorňuje spôsob rekompilácie funkcie.

```
ALTER FUNCTION nazov_funkcie COMPILE
  {TEMPORAL_REG | VERSION_REG};
```

Definícia príkazu 138: Rekompilácia funkcie

Plánovanie

Vyššie uvedené princípy predpokladajú, že novo zaregistrovaná verzia funkcie nadobudne platnosť hneď po kompilácii. Parameter PRAGMA umožňuje presne nastaviť počiatočný bod platnosti, buď pomocou intervalu, alebo špecifikáciou časovej pečiatky. Hodnota sa musí vzťahovať na budúcnosť. V opačnom prípade sa vyvolá výnimka a nová verzia nebude ani skompilovaná, ani zaregistrovaná. Nasledujúci kód (*def. príkazu 139*) znázorňuje plánovanie.

```
ALTER FUNCTION nazov_funkcie COMPILE
 {TEMPORAL_REG | VERSION_REG}
   VALIDITY {TO_DATE(input_val, 'DD.MM.YYYY HH24:MI:SS) |
             INTERVAL 'value' {DAY TO SECOND | YEAR TO MONTH}};
```

Definícia príkazu 139: Plánovanie

Registrácia aktualizácie výkonu

Táto klauzula umožňuje optimalizovať výkon existujúcej verzie buď aplikáciou nových funkcií, vytváraním nových metód, optimalizáciou toku dát, atď. Predvolene je optimalizovaná aktuálna verzia, ale definíciu je možné rozšíriť uvedením identifikátora verzie. Nasledujúci kód (*def. príkazu 140*) znázorňuje vytvorenie funkcie s rozšírením o registráciu aktualizácie výkonu.

```
CREATE OR REPLACE FUNCTION nazov_funkcie
 RETURN datovy_typ
   IS PRAGMA_PERF_REG [identifikator_verzie];
     [definícia lokálnych premenných]
 BEGIN
  -- telo funkcie
 END;
/
```

Definícia príkazu 140: Registrácia aktualizácie výkonu

Ak by daný identifikátor neexistoval, spracovanie by sa skončilo chybou.

3.7.2 Experimentálna časť

Štúdia hodnotenia výkonu sa uskutočnila pomocou množiny údajov o monitorovaní letovej prevádzky. Táto množina zahŕňa lety prevádzkované v európskom priestore v rokoch 2018 až 2022. Každý let je možné jednoznačne identifikovať pomocou atribútu ECTRL_ID. Súvisiace merania a letové parametre sú ohraničené časovou pečiatkou výskytu. Hodnoty sú navyše zoradené v čase prostredníctvom atribútu SEQ_ID. Súbor dát pozostáva zo 100 letových parametrov a plánovaných a skutočných trás. Zahŕňa monitorovanie letu a to vzlet, samotný let, pristátie, rolovanie a parkovanie. Okrem toho sa na základe pozičných údajov dajú identifikovať a priradiť letové informačné regióny (FIR) danému letu. V letectve je FIR určité územie vzdušného priestoru, v ktorom sa poskytuje letecká informačná služba a služba upozorňovania (ALRS). Každý FIR je súčasťou jedného leteckého navigačného regiónu: Africký a Indooceánsky región (AFI); Ázijský región (ASIA); Karibský región (CAR); Európsky región (EUR); Región Stredného východu (MID); Severoamerický región (NAM); Severoatlantický región (NAT); Tichomorský región (PAC); a Juhoamerický región (SAM). FIR sa líši veľkosťou. Menšie krajiny môžu mať nad sebou iba jeden FIR a väčšie krajiny ich môžu mať viacero. Vzdušný priestor nad oceánom je zvyčajne rozdelený na dva alebo viac FIR a je delegovaný riadiacim orgánom krajiny, nad ktorou je. Významný aspekt FIR-u spočíva v jeho vývoji v čase. Konkrétne hranice a pozície nie sú striktne stanovené a sú pravidelne prehodnocované. Teda aj ich definície sa v priebehu času vyvíjajú. V tomto prostredí sa jednotlivé lety monitorujú na výpočet efektivity letu, spotreby paliva, nákladov, ale najmä na vplyv na životné prostredie. V súčasnosti je problém ešte vážnejší, je nevyhnutné minimalizovať vplyv na životné prostredie. Táto množina údajov bola vybraná aj na základe podporných projektov tohto výskumu – analýzy environmentálnych dát pomocou projektu EverGreen, ako aj výskumu optimalizácie inteligentných dopravných systémov (VEGA 1/0192/24). Súbor dát pozostáva z 500 000 letov s rôznou dĺžkou a frekvenciou získavania údajov. Celkovo množina obsahuje pre každý let jednu plánovanú trasu a jednu skutočnú trasu, pokrytú 100 parametrami (80 je reálnych, 10 je celočíselných, 5 je textových a 4 sú odkazy na časové výrazy a 1 je JSON). V priemere každý let obsahoval 1 000 riadkov údajov (súbor dát bol modifikovaný znížením vzorkovacej frekvencie údajov). Príklad údajov je uvedený na *Obrázku 93*.

```
"ECTRL ID","Sequence Number","AUA ID","Entry Time","Exit Time"
"184408024","1","EGGXOCA","01-03-2015 05:54:29","01-03-2015 06:51:47"
"184408024","2","EISNCTA","01-03-2015 06:51:47","01-03-2015 07:17:44"
"184408024","3","EGTTCTA","01-03-2015 07:17:44","01-03-2015 08:00:03"
"184408024","4","ATC_UNK","01-03-2015 08:00:03","01-03-2015 08:00:54"
"184408024","5","EHAACTA","01-03-2015 08:00:54","01-03-2015 08:08:45"
```

Obrázok 93: Štruktúra dát z leteckej dopravy

Pre účely tejto hodnotiacej štúdie sa využil databázový systém Oracle 23c, distribučný balík Oracle 23c Free, Developer Release Version 23.2.0.0.0. Štúdia sa zameriava na nasledujúce aspekty, spracované temporálnymi funkciami:

- F1: Konverzia typu timestamp do formátu UTC, pričom sa zohľadňuje letný a zimný čas. Pre každé ročné obdobie sa výpočet mení.
- F2: Získanie FIR na základe pozičných údajov.
- F3: Vyhodnotenie vplyvu na znečistenie ovzdušia.
- F4: Vyhodnotenie efektívnosti letu porovnaním optimálnej a skutočnej trasy zohľadnením aktuálnych poveternostných podmienok a ďalších obmedzení.
- F5: Komplexné monitorovanie letu generujúce JSON, ktoré charakterizuje trasu (plánovanú a skutočnú), efektivitu letu v jednotlivých FIR a parametre letu vo všetkých regiónoch.

Výpočtová štúdia poskytuje päť riešení:

- REF – odkazuje na existujúce riešenie, v ktorom temporálne verziovanie nemá špecifickú podporu, jednotlivé verzie sú umiestnené v súboroch. Na získanie správnej funkcie je potrebné identifikovať konkrétnu verziu na základe kódu hlavičky funkcie, nasledovaného parsovaním, kontrolou, načítaním a nakoniec spustením.
- R1 – prináša dodatočnú bezpečnostnú vrstvu tým, že sa kód verzií ukladá v databáze. Stále je však potrebné parsovať, kontrolovať a predspracovať kód funkcie pred spustením.
- R2 – odstraňuje potrebu parsovania a kontroly, pretože verzia sa ukladá v parsovanej podobe, čo sa vykoná raz počas počiatočného spracovania.

- R3 – poskytuje prístup ku všetkým verziám, ktoré sa líšia v názvoch. Všetky verzie sú priamo dostupné na spustenie, avšak je potrebné štrukturalizovať definície kódu nahradením pôvodného názvu funkcie príslušnou verziou. Zatiaľ čo sa definícia kódu zmení, vytvorené plány vykonania už nebudú priradené a neskôr sa na ne nemožno odkazovať. Preto, každá verzia funkcie vytvára inú hodnotu Hash Plan ID.
- R4 – popisuje navrhované riešenie zavedením transformačného mapovacieho modulu špecifikovaného temporálnou vrstvou. Každá verzia je spojená s časovým rámcom platnosti, ako aj s možnosťami aktualizácie výkonu, ktoré tvoria bi-temporálnu vrstvu sekvencií.

Pre štúdiu hodnotenia sa uskutočnili tri experimenty, ktoré zachytávali:

- Vplyv parsovania a načítavania
- Vplyv identifikácie správnej verzie
- Načítanie kódu z databázy (alebo súboru) do pamäte inštancie databázy

Prvý experiment sa zaoberá vplyvom parsovania a načítavania verzií. Riešenie REF neobsahuje žiadny modul na prácu s verziovaním. Verzie sú uložené externe, mimo databázy, v súborovom systéme. Každá verzia je uložená v samostatnom súbore. Hoci je celý proces jednoduchý, veľa prechodov medzi databázou a súborovým systémom robí toto riešenie príliš náročným. Vylepšené riešenie (REFopt) vytvára partície dátových verzií pre spustenie. Takto sa každá verzia načíta iba raz a potom sa spustí pre každý výskyt. Hoci to prináša výhody v rámci optimalizácie, pretrváva významný nedostatok v podobe možnosti spracovať funkcie a jednotlivé verzie v paralelnom prostredí. Zdrojové súbory obsahujúce kódy verzií funkcií sú prístupné v directory object v databáze, vylepšené operačným systémom a udelenými právami v databáze.

Ukladanie jednotlivých zdrojových súborov verzií v databázovej vrstve môže priniesť ďalšie benefity. Kód je uložený v databázových tabuľkách pomocou definícií atribútov veľkých objektov (Large Object Attribute). Každá verzia kódu sa umiestni priamo do databázy, dokonca aj v neparsovanej (originálnej) forme. S úložiskom súborov nie je potrebné manipulovať. Na základe toho má inštancia databázového systému priamy prístup k databázovému repozitáru pomocou procesov na pozadí, čím sa optimalizujú I/O operácie. Nevýhoda tohto riešenia je blokovo orientovaná štruktúra. Čo znamená, že bez ohľadu na

množstvo relevantných dát v bloku sa musí vždy načítať a spracovať celý blok. Navyše, používateľ potrebuje vhodnú metódu na transformáciu pôvodného zdrojového kódu na štruktúru veľkého objektu.

R2 vylepšuje existujúce princípy riešenia R1 tým, že ukladá predparsované a skontrolované verzie ako objekt. Nutnosť parsovania a kontroly je obmedzená, pretože sa vykonáva len raz počas počiatočného spracovania verzie. Načítanie je jednoduchšie, hoci musí byť k dispozícii dodatočný modul na konverziu predparsovanej verzie na spustiteľný zdrojový kód. Nevýhodou tohto riešenia je potreba ukladať aj pôvodný zdrojový kód, takže nároky na úložisko sú takmer dvojnásobné. Ak sa zmenia referenčné objekty, daná verzia funkcie sa označí ako neplatná a pred ďalším použitím sa musí skompilovať. To však vyžaduje pôvodný zdrojový kód na kontrolu dostupnosti zdrojov, oprávnení, odkazov, atď.

Z pohľadu parsovania a načítavania ponúka riešenie R3 dostatočný výkon. Každá verzia je priamo prístupná prostredníctvom jedinečného názvu vytvoreného z pôvodného identifikátora funkcie a prípony verzie. Z prostredia volania sú jednotlivé verzie priamo dostupné prostredníctvom štruktúry pamäte inštancie a popisov systémových tabuliek. Avšak, v druhom kroku experimentov, identifikácii správnej verzie, vznikajú dodatočné nároky kvôli potrebe reštrukturalizovať používateľom definovaný kód a aplikovať hlavičky jednotlivých verzií (nahradiť pôvodné názvy funkcií príponami verzií). Následne sa definícia kódu mení za chodu a ďalšia optimalizácia výkonu je náročná, pretože odkazy sa v priebehu času vyvíjajú a musia sa dynamicky aplikovať.

Navrhované riešenie R4 sa odvoláva na temporálnu paradigmu. Všetky verzie sú priamo parsované a dostupné. Pôvodný zdrojový kód sa nemusí ukladať. Priame mapovanie sa vykonáva dynamicky pomocou smerníkov, takže pôvodné názvy a definícia používateľského kódu zostávajú nezmenené. Aplikácia verzie sa vykoná okamžite. Hodnoty Hash Plan ID však zostávajú rovnaké, takže vytvorené plány vykonania zostávajú platné.

Tabuľka 31 zobrazuje výsledky vyjadrujúce náklady. Náklady na spustenie v databáze tvoria metriky pokrývajúce zdroje databázového systému, úložisko, potrebu načítavania, spotrebu pamäte, pripojené procesy na pozadí a čas spracovania. Pre deklaratívne účely sú jednotlivé hodnoty vyjadrené v percentách. Výsledky sú získané na základe volania funkcií F1-F5 10 000-krát, pričom verzie sú vybrané náhodne. Hodnotenie sa vykonalo 10-krát a *Tabuľka 31* vyjadruje priemerné hodnoty.

Tabuľka 31: Vplyv parsovania a kontroly

Vplyv parsovania a kontroly	REF	REFopt	R1	R1opt	R2	R3	R4
Náklady [%]	100.00%	43.24%	72.77%	39.91%	41.76%	33.12%	27.04%

Zatiaľ čo volané verzie sú v prostredí náhodne distribuované, najhoršie riešenie ponúka REF, pretože sa nezaoberá efektivitou parsovania, kontroly a načítavania. REFopt vytvára partície v rámci verzií a načítanie konkrétnej verzie sa vykoná len raz pre celý kód. Použitím tejto optimalizačnej techniky sa môžu celkové náklady na spracovanie znížiť až o 56.76%. Riešenie R1 je analogické riešeniu REF, ale poskytuje dodatočnú bezpečnostnú vrstvu presunutím zdrojového kódu do databázového repozitára. Celkové náklady na spracovanie sú 72.77%, čo predstavuje pokles oproti riešeniu REF o 27.23%. Je to spôsobené obmedzením potreby kontrolovať úložiská súborov mimo databázy. Ak sa však vykonáva optimalizácia s využitím partícií, celkové náklady na spracovanie sa znížia na 39.91%. Pre ostatné architektúry sa partíciovanie a kategorizácia verzií na zaistenie jedného nečítania verzie nepožadujú, pretože verzie sú buď predparsované alebo priamo prístupné databázovým systémom. Riešenie R2 vyžaduje 41.76%. Je dôležité však spomenúť, že porovnanie s optimalizačnými technikami partíciovania verzií, ako je kategorizácia, je zbytočné. Z pohľadu uvedených štatistík sa nevyžaduje vytváranie kategórií a zoskupovanie verzií. Na základe dodatočných experimentov si správa partíciovania verzií vyžaduje ďalších 5-10% nákladov v závislosti od zložitosti príkazov alebo kódu. S rastúcim počtom verzií môžu dodatočné požiadavky výrazne narásť. Riešenie R3 je z pohľadu načítavania výhodné. Jednotlivé verzie sú priamo dostupné, ale obmedzené rôznymi názvami v rámci verzií. Na základe definovaného prostredia bolo potrebných ďalších 22.43% na transformáciu existujúceho zdrojového kódu s vylepšením názvov verzií funkcií. Výraznejšie to bolo pre jazyk SQL, kde sa zmenil formát kódu príkazov, z ktorého sa vypočíta plán vykonania. Aj keď existujúce plány môžu byť v pamäti inštancie v Library Cache prítomné, spracovaná hodnota Hash Plan ID sa líši, takže mapovanie sa neuskutoční a musí sa vytvoriť nový plán vykonania. Navyše, ak sa niektorá verzia zmení, musí sa vytvoriť nový plán vykonania. V SQL pre definované funkcie F1-F5 sú dodatočné požiadavky 34.12%. Pre PL/SQL to nie je také prísne. Aj keď sa zmení plán procesu, prepínače kontextu funkcie môžu zmenu jednoduchšie aplikovať. Presnejšie povedané, dodatočné požiadavky pre PL/SQL sú 10.74%. Najlepšie riešenie ponúka navrhované riešenie R4, pretože celá správa sa vykonáva na úrovni databázy. Každá verzia funkcie sa skompiluje iba raz, a potom je priamo prístupná.

Mapovanie je riadené temporálnym modulom, ktorý dynamicky presmeruje funkciu do konkrétnej verzie. Navyše, pre každú verziu existuje smerník na najnovšiu verziu, takže sa vždy použije najoptimalizovanejšie riešenie. Celkové náklady sú 27.04%. V porovnaní s pôvodným riešením (REF) je to zlepšenie o 72.96%. V porovnaní s riešením R2, ktoré ukladá parsované verzie do databázy, je zlepšenie o 35.25%. V porovnaní s riešením R3 sú náklady na spracovanie 81.64% (zlepšenie o 18.36%).

Treba poznamenať, že okrem navrhovaného riešenia R4, sa pre každú verziu funkcie ukladá a následne načíta iba aktuálna aktualizácia výkonu. Takto bola každá verzia poskytnutá iba jedným zdrojovým kódom – tým najnovším (na základe najvyššieho poradového čísla).

Druhá časť štúdie výkonnosti sa zameriava na proces identifikácie správnej verzie funkcie na základe údajov a referencie platnosti. V tomto prípade nie sú relevantné riešenia REFopt a R1opt, pretože neposkytujú žiadnu ďalšiu vrstvu z hľadiska identifikácie verzie. Nezáleží na tom, či sa tento proces uskutoční počas identifikácie verzie alebo kategorizácie prostredníctvom partíciovania verzií. Na základe hodnotenia je rozdiel medzi pôvodnými a optimalizovanými riešeniami menej ako 1%.

Riešenie REF neukladá a nešpecifikuje intervaly platnosti, takže na identifikáciu vhodnej verzie sa vždy musí načítať hlavička kódu. Ak sa nájde, skenovanie sa môže ukončiť, ale vo všeobecnosti platí, že sa musia prehodnotiť všetky verzie. Navyše, ak neidentifikuje aktualizáciu výkonu, musia sa v vždy prehľadať všetky verzie, pretože môžu byť náhodne distribuované. Ukladanie verzií do databázy (riešenia R1 a R2) v pôvodnom alebo parsovanom formáte má zmysel. Databázová tabuľka ukladajúca obsah je temporálna, takže identifikácia sa vykonáva na úrovni databázy. Bi-temporálna architektúra umožňuje spravovať nielen samotné verzie, ale aj aktualizácie výkonu pre každú verziu. Identifikácia verzie pomocou pomenovania (riešenie R3) sa neosvedčila. S ohľadom na pomenovanie a platnosť vyžaduje reflexia prístup k temporálnej vrstve obsahujúcej verzie, ale aj koreláciu s diskovým úložiskom na lokalizáciu a prenos daných verzií na základe ich názvu. Výsledky experimentov sú uvedené v *Tabuľke 32*.

Tabuľka 32: Identifikácia správnej verzie

Identifikácia správnej verzie	REF	R1	R2	R3	R4
Náklady [%]	100.00%	67.12%	66.59%	93.03%	30.58%

Extrahovaním iba aktuálnych aktualizácií výkonu verzií do samostatnej temporálnej architektúry sa môžu náklady znížiť na 25.32%, čo predstavuje zlepšenie o 17.20%.

Posledná časť štúdie sa týka načítania kódu z databázy alebo úložiska do pamäte inštancie databázy. Ak sú verzie funkcie už dostupné a uložené v databázovom úložisku, stačí ich načítať do pamäte inštancie, spustiteľnú verziu. Aby bolo hodnotenie relevantné a porovnateľné, pred každým experimentom sa pamäť inštancie úplne vyprázdni. Dôvodom je, že verzia tam už mohla byť uložená a jej načítanie by nebolo potrebné, čo by ovplyvnilo výsledky. Čistenie pamäte inštancie prebieha na úrovni jednotiek kódu, takže celý používateľom definovaný kód sa považuje za jednu jednotku. Výsledky experimentov sú uvedené v *Tabuľke 33*.

Riešenie REF je najnáročnejšie, pretože sa musia vykonať dva kroky – načítanie zo súboru do temporálneho databázového úložiska, načítanie do pamäte inštancie a následná správa. Riešenia R1 a R2 poskytujú takmer rovnaké výsledky. Dáta sú prítomné v databáze. Rozdiel medzi nimi je vo veľkosti - parsovaná verzia je zvyčajne optimalizovaná, a preto vyžaduje menšiu veľkosť, takže sa načítava menej blokov a je to rýchlejšie. Rozdiel medzi riešeniami R1 a R2 10.32%. Riešenie R3 vychádza z toho, že verzie sú prítomné v systéme a dá sa na ne odkazovať systémovými tabuľkami. Z pohľadu spustenia jednotlivé verzie fungujú ako bežné konvenčné funkcie. Navrhované riešenie R4 kombinuje výhody všetkých vyššie uvedených architektúr. Jediné, čo je potrebné urobiť, je načítať obsah spustiteľnej verzie funkcie z databázy do pamäte inštancie, rovnako ako bežné funkcie. Z pohľadu správy nie je priestor na ďalšiu optimalizáciu.

Tabuľka 33: Načítanie kódu z databázy do pamäte

Načítanie kódu z databázy do pamäte	REF	R1	R2	R3	R4
Náklady [%]	100.00%	54.91%	49.24%	98.40%	46.06%

Zhrnutie výsledkov

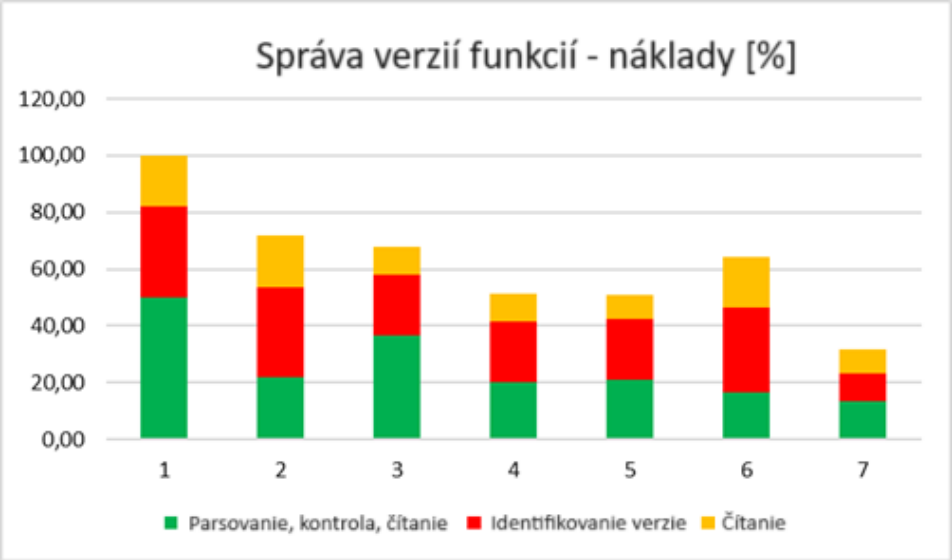
V rámci výskumu sme sa zamerali na tri aspekty správy a manipulácie s verziovaním temporálnych funkcií. Zaoberali sme sa procesom parsovania, kontroly a všeobecne kompiláciou, pretože verzie môžu byť uložené vo formáte zdrojového kódu nachádzajúceho sa mimo databázy v bežných súboroch. Toto je najkritickejšia aktivita a zaberá 50% celého procesu správy funkcií. Druhý aspekt sa týka procesu identifikácie správnej verzie. Táto činnosť zaberie 32%. Zostávajúcich 18% sa týka načítania predparsovanej verzie, čím sa

stáva priamo spustiteľná. Táto aktivita presúva údaje z úložiska do pamäte inštancie. *Tabuľka 34* zobrazuje náklady celého procesu s prihliadnutím na váhy jednotlivých aktivít.

Tabuľka 34: Zhrnutie výsledkov

Správa verzií funkcií	REF	REFopt	R1	R1opt	R2	R3	R4
Náklady [%]	100.00%	71.71%	67.75%	51.32%	51.05%	64.04%	31.60%

Najhoršie výsledky dosiahlo riešenie REF, ktoré neposkytuje žiadne štruktúry na obsluhu a lokalizáciu verzií. Podobne, riešenie R3 nie je pre ďalšiu analýzu a praktické použitie relevantné, práve pre potrebu zmeny plánu vykonania a ovplyvnenia celkového výkonu systému. Pri porovnaní riešení REF a R3 je rozdiel medzi nimi 10.70%. Z globálneho hľadiska sú riešenia R1opt a R2 veľmi podobné a metóda zachovania štruktúry kódu nemá výrazný vplyv. Toto je však spôsobené partíciovaním. Ak by tam nebolo, tak celková náročnosť riešenia R1 by bola 67.75%. Najlepšie riešenie jednoznačne poskytuje R4. Dosahuje najlepšie výsledky na všetkých úrovniach. Celkové náklady na spracovanie sú 31.60%. Grafické znázornenie nákladov je zobrazené na *Obrázku 94*.



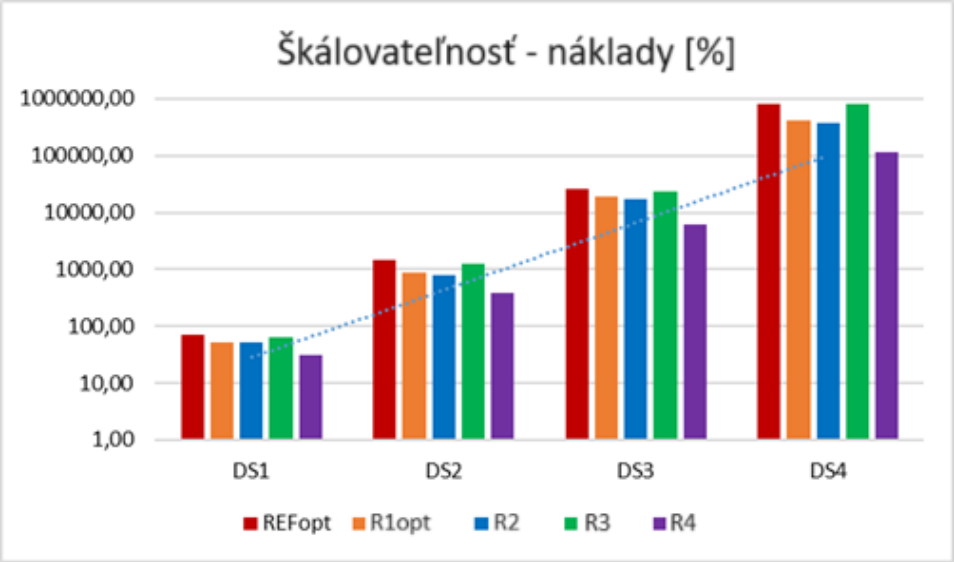
Obrázok 94: Náklady správy verzií funkcií

Škálovateľnosť

Na základe dosiahnutých výsledkov je zrejmé, že navrhované riešenie prináša výrazné zlepšenie výkonnosti a je relevantné v porovnaní s existujúcimi technikami a prístupmi. Temporálne databázy sa však vyznačujú tým, že množstvo dát v priebehu času výrazne narastá a počet údajov, ktoré majú byť spracované a ukladané sa teda neustále zvyšuje. Aby sa zdôraznila flexibilita navrhovaného riešenia, je dôležité ukázať

škálovateľnosť jednotlivých riešení. Pre porovnateľnosť a všeobecné aplikovanie riešení uvedieme len optimalizované riešenia pre REF a R1 kvôli výkonnosti.

Prvá množina údajov zodpovedá pôvodnému množstvu údajov, druhá množina je 10-krát väčšia, tretia množina je 100-krát väčšia a posledná, štvrtá množina je 1000-krát väčšia. Jednotlivé množiny sú označené symbolmi DS1 až DS4. Výsledky sú zobrazené na *Obrázku 95*.



Obrázok 95: Škálovateľnosť

3.8 Problematika nedefinovaných hodnôt, chybné dáta

Problematika nedefinovaných hodnôt a chybných dát pri ukladaní a spracovaní v databázových systémoch je dôležitou témou, keďže v rámci dátových skladov s obrovským množstvom dát je potrebná ich správna interpretácia. Neúplné dáta, prípadne nevedomosť správneho použitia funkcií môže viesť ku skresleným výpočtom a chybnej prezentácii výsledkov, čo môže mať nežiadúce následky pre spoločnosť, pre ktorú sa vykonávala daná analýza.

Dátové sklady obsahujú milióny dát, ktoré nemusia nadobúdať vždy správne hodnoty. Môže sa stať, že niektoré hodnoty stĺpcov nie sú vyplnené, niektoré nadobúdajú hodnoty, ktoré nemusia byť validné, intervaly dátumov nemusia na seba nadväzovať, využité analytické či agregačné funkcie nemusia byť správne použité, môžu skresľovať výsledky, nemusí byť dostatočne využitý ich potenciál a podobne. S týmito a ďalšími problémami sa môžeme stretnúť pri práci s dátami a tvorbe rôznych analýz. Táto kapitola je zameraná na bližší popis a riešenia týchto problémov.

3.8.1 NULL hodnoty

Veľmi často sa stretávame s tým, že niektoré hodnoty stĺpcov nie sú vyplnené, sú prázdne a tým nadobúdajú hodnotu NULL. Ako s takýmito stĺpcami pracovať aby sme si boli istí, že výsledky budú správne? NULL hodnoty sú diskutovanou témou vo všeobecnosti. Nižšie si ukážeme ako spracovávajú rôzne funkcie tieto hodnoty.

V prípade, že chceme nejakým spôsobom ošetriť NULL hodnoty a pracovať s nejakou nami definovanou hodnotu namiesto NULL, existujú funkcie, ktoré dokážu transformovať NULL hodnoty na nami definovanú hodnotu. Medzi tieto funkcie patrí: NVL(), COALESCE(), DECODE(). Pri NULL hodnotách je taktiež dôležité vedieť, že ich nie je možné porovnávať cez znamienko =, ale cez podmienku IS NULL prípadne IS NOT NULL.

NULL hodnoty a agregačné funkcie

Dôležitou informácia, ktorá platí pre agregačné funkcie je tá, že tieto funkcie ignorujú NULL hodnoty. Nasledujúci príkaz *SELECT* (*def. príkazu 141*) znázorňuje príkaz, ktorý vyberie všetky záznamy z tabuľky *nehody*. Výsledok tohto príkazu je 699 588 záznamov.

```
SELECT COUNT(*)
 FROM nehody;
```

Definícia príkazu 141: Príkaz *SELECT*, ktorý vyberie všetky záznamy z tabuľky *nehody*

Znamienko * je možné nahradiť ľubovoľným atribútom. V prípade, ak namiesto znamienka * vložíme akýkoľvek NOT NULL stĺpec, výsledok bude rovnaký. Nasledujúci príkaz (*def. príkazu 142*) vyberie všetky záznamy, ktoré majú definované *ID_NEHODY*. Keďže ide o primárny kľúč tabuľky, nikdy nebude nadobúdať hodnotu NULL.

```
SELECT COUNT(id_nehody)
 FROM nehody;
```

Definícia príkazu 142: Príkaz *SELECT*, ktorý vyberie záznamy z tabuľky *nehody*

Výsledok vyššie uvedeného príkazu je totožný, ako výsledok *def. príkazu 141,* a to je 699 588. Ak nahradíme znamienko * alebo primárny kľúč iným stĺpcom, ktorý môže nadobúdať hodnotu NULL, systém tieto záznamy bude ignorovať a spočíta len tie riadky, ktorých stĺpec je vyplnený. *Def. príkazu 143* zobrazuje príkaz *SELECT*, ktorý spočíta všetky záznamy nehôd s vyplneným *ID_SMYKU*. Keďže tento atribút môže nadobúdať hodnotu NULL, tak systém spočítal len tie riadky, ktoré majú daný stĺpec vyplnený. Preto výsledok tohto príkazu je 577 029.

```
SELECT COUNT(id_smyku)
 FROM nehody;
```

Definícia príkazu 143: Príkaz *SELECT*, ktorý vyberie záznamy z tabuľky *nehody*

Iný spôsob, ako spočítať záznamy, ktoré majú stĺpec *ID_SMYKU* vyplnený, je zobrazený v *def. príkazu 144*. Tento príkaz spočíta všetky záznamy, rozdiel je v tom, že je príkaz doplnený o podmienku *WHERE*, v ktorej sa kontroluje stĺpec *ID_SMYKU*. Výsledok tohto príkazu je rovnaký ako na *def. príkazu 143*.

```
SELECT COUNT(*)
 FROM nehody
 WHERE id_smyku IS NOT NULL;
```

Definícia príkazu 144: Príkaz *SELECT*, ktorý vyberie záznamy z tabuľky *nehody*

Oba spôsoby sú rovnako efektívne čo sa týka celkových nákladov plánu vykonania. Po zobrazení plánu vykonania sme zistili, že celkové náklady pre oba spôsoby nadobúdajú hodnotu 386. Plán vykonania je zobrazený na *Obrázku 96*.

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                   |     1 |     2 |   386   (2)| 00:00:01 |
|   1 |  SORT AGGREGATE       |                   |     1 |     2 |            |          |
|*  2 |   INDEX FAST FULL SCAN| IX_RELATIONSHIP26 |   577K|  1127K|   386   (2)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Obrázok 96: Plán vykonania - agregačná funkcia COUNT()

V prípade agregačnej funkcie SUM() sa NULL hodnoty taktiež ignorujú pred spracovaním záznamov, takže, keď vykonáme príkaz *SELECT* ako zobrazuje *def. príkazu 145*, tak výsledok spracovania bude reprezentovať súčty tých hodnôt, ktoré nenadobúdajú hodnotu NULL.

```
SELECT (TO_CHAR(p2a_datum, 'YYYY')), SUM(p53_skoda_na_vozidle)
 FROM nehody
 GROUP BY (TO_CHAR(p2a_datum, 'YYYY'));
```

Definícia príkazu 145: Príkaz *Select*, ktorý sčíta škody vozidiel za jednotlivé roky

Výsledok spracovania sa nachádza v *Tabuľke 35*. Stĺpec p53_skoda_na_vozidle môže nadobúdať NULL hodnoty, tie sa však pri použití agregačnej funkcie SUM() budú ignorovať.

Tabuľka 35: Výsledok spracovania príkazu s agregačnou funkciou SUM()

Rok	Škoda (česká koruna)
2016	28 876 619
2017	31 466 236
2018	32 477 426

2019	33 881 091
2020	30 561 399
2021	33 298 983
2022	33 828 964

Ak by sme chceli spočítať hodnoty v rámci riadku, pričom by sme využili matematický operátor + a niektorá z hodnôt v stĺpci by nadobúdala hodnotu NULL, výsledok by bol iný. Nasledujúci príkaz spočíta pre každý riadok škodu na vozidle a celkovú hmotnú škodu. *Def. príkazu 146* zobrazuje tento príkaz.

```
SELECT id_nehody, p53_skoda_na_vozidle AS skoda_na_vozidle,
       p14_celkova_hmotna_skoda AS hmotna_skoda,
       p53_skoda_na_vozidle + p14_celkova_hmotna_skoda AS
           celkova_skoda
 FROM nehody;
```

Definícia príkazu 146: Príkaz *SELECT* so sčítaním NULL hodnôt

Ako bolo uvedené vyššie matematické operátory nevedia pracovať s NULL hodnotami, preto výsledok sčítania stĺpcov s NULL hodnotou nadobúda hodnotu NULL, ako je možné vidieť v *Tabuľke 36*.

Tabuľka 36: Výsledok sčítania stĺpcov s NULL hodnotami

ID_NEHODY	SKODA_NA_VOZIDLE	HMOTNA_SKODA	CELKOVA_SKODA
002100170148	0	100	100
002100170149	NULL	200	NULL
002100170150	0	200	200
002100170151	1000	2500	3500
002100170001	0	400	400
002100170002	NULL	200	NULL
002100170003	800	1200	2000

V prípade, že by sme ďalej chceli spracovať tieto výsledky, mohol by nastať problém. Preto je dôležité vhodným spôsobom ošetriť tieto hodnoty a v prípade, že nadobúdajú hodnotu NULL, nahradiť ju inou hodnotou, v našom prípade hodnotou 0. Úpravu vyššie uvedeného príkazu *SELECT* zobrazuje *def. príkazu 147*, ktorá využíva funkciu NVL().

```
SELECT id_nehody, p53_skoda_na_vozidle AS skoda_na_vozidle,
       p14_celkova_hmotna_skoda AS hmotna_skoda,
       NVL(p53_skoda_na_vozidle, 0) + p14_celkova_hmotna_skoda
           AS celkova_suma
FROM nehody;
```

Definícia príkazu 147: Príkaz *SELECT* s ošetrením a sčítaním NULL hodnôt

Výsledok vyššie uvedeného príkazu je zobrazený v *Tabuľke 37*. Ako je možné vidieť, pomocou funkcie NVL() sme ošetrili hodnotu v stĺpci škoda na vozidle a po sčítaní sme dostali správne výsledky.

Tabuľka 37: Výsledok sčítania stĺpcov s ošetrenými NULL hodnotami

ID_NEHODY	SKODA_NA_VOZIDLE	HMOTNA_SKODA	CELKOVA_SKODA
002100170148	0	100	100
002100170149	NULL	200	200
002100170150	0	200	200
002100170151	1000	2500	3500
002100170001	0	400	400
002100170002	NULL	200	200
002100170003	800	1200	2000

NULL hodnoty a indexy

Každý databázový systém pristupuje rôzne k NULL hodnotám, dokonca nie každý ich dokáže indexovať. Preto okrem ošetrovania NULL hodnôt v dotazoch je potrebné myslieť na ich ošetrenie aj pri vytváraní indexov, pretože môže nastať situácia, že aj napriek tomu, že máme vytvorený index nad stĺpcom, ktorý môže nadobúdať NULL hodnoty, tento index sa v skutočnosti nepoužije. Ak by sme chceli takýto index predsa využívať, je potrebné využiť ***funkcionálny index***.

NULL hodnoty a partície

Partície v databázových systémoch slúžia na rozdelenie tabuľky na menšie časti, s cieľom zvýšiť výkon dotazov a zlepšiť udržiavanie tabuliek. Pri definícií partícií je potrebné uviesť stĺpec tabuľky, podľa ktorého sa budú partície vytvárať. Pokiaľ je stĺpec definovaný ako NOT NULL stĺpec, nie je to žiadny problém. Ale čo v prípade ak stĺpec podľa ktorého vytvárame partície môže nadobúdať aj NULL hodnoty? Do akej partície takýto stĺpec zaradiť? Vloží sa taký záznam do tabuľky?

NULL hodnoty v partíciách je možné riešiť niekoľkými spôsobmi. Dôležité je správne definovať partíciu, kam by tieto NULL hodnoty spadali. Hlavným cieľom nasledujúceho výskumu je vytvoriť metodiku spracovania záznamov s NULL hodnotami v tabuľkách rozdelených na partície a ukázať ako je potrebné definovať tabuľky a ich partície tak, aby bolo možné vkladať záznamy s NULL hodnotami kľúčových stĺpcov partícií do tabuliek. Celý popis štúdie a navrhovaného riešenia spolu s ďalšími podrobnosťami výskumu je možné nájsť v publikácii [HD9].

Pre experimenty sme vybrali databázový systém Oracle, ktorý poskytuje širokú škálu možností riešenia tejto problematiky. Navyše podporuje niekoľko druhov rozdeľovania

tabuliek na partície. V našom výskume sme sa zamerali na 3 typy partícií: Range, List a Hash partitioning. Pre každý typ rozdelenia tabuliek na partície sme vytvorili osobitné databázové tabuľky, do ktorých sme vložili záznamy s rôznymi hodnotami, pričom niektoré z týchto záznamov obsahovali aj NULL hodnoty kľúčových stĺpcov, podľa ktorých sa tabuľka delí na partície.

Range partitioning

Najskôr sme sa zamerali na Range partitioning. Range partitioning umožňuje vytvárať partície priamo alebo dynamicky. Každá z týchto možností bude skúmaná hlbšie. Pri definícií partícií priamo sa definuje toľko partícií, koľko potrebujeme. Na druhej strane pri definovaní partícií dynamicky, definujeme prvú, počiatočnú, partíciu a ďalšie partície sa vytvárajú podľa potreby.

Def. príkazu 148 znázorňuje kód vytvorenia tabuľky *tab_particie_null* s 3 stĺpcami (id, dátum vytvorenia záznamu, text) a definíciou partícií podľa dátumu vytvorenia. Tabuľka je rozdelená na štyri partície, pričom prvá partícia pokrýva všetky záznamy, ktoré boli vytvorené do roku 2021, druhá partícia pokrýva záznamy vytvorené v roku 2021, tretia partícia pokrýva záznamy vytvorené v roku 2022 a posledná partícia uchováva záznamy s hodnotou roku vytvorenia vyššou ako 2022.

```
CREATE TABLE tab_particie_null(
  id_zaznamu          NUMBER NOT NULL PRIMARY KEY,
  datum_vytvorenia    DATE,
  text                VARCHAR2(30))
PARTITION BY RANGE (datum_vytvorenia)(
     PARTITION part_2020
          VALUES LESS THAN (TO_DATE('01.01.2021', 'DD.MM.YYYY')),
     PARTITION part_2021
          VALUES LESS THAN (TO_DATE('01.01.2022', 'DD.MM.YYYY')),
     PARTITION part_2022
          VALUES LESS THAN (TO_DATE('01.01.2023', 'DD.MM.YYYY')),
     PARTITION part_20XX
          VALUES LESS THAN (MAXVALUE))
/
```

Definícia príkazu 148: Tabuľka partícií *tab_particie_null*

Po vytvorení tabuľky sme do nej vložili niekoľko záznamov (*def. príkazu 149*), pričom dva z týchto záznamov obsahovali hodnotu NULL v dátume vytvorenia.

```
INSERT INTO tab_particie_null
 VALUES(1, TO_DATE('21.12.2022', 'DD.MM.YYYY'), 'Text1');

INSERT INTO tab_particie_null
 VALUES(2, TO_DATE ('21.12.2021', 'DD.MM.YYYY'), 'Text2');

INSERT INTO tab_particie_null
 VALUES(3, NULL, 'Text3');

INSERT INTO tab_particie_null
 VALUES(4, TO_DATE ('21.10.2023', 'DD.MM.YYYY'), 'Text4');

INSERT INTO tab_particie_null
 VALUES(5, NULL, 'Text5');
```

Definícia príkazu 149: Záznamy na vloženie do tabuliek partícií s NULL hodnotou

Všetky záznamy sa podarilo úspešne vložiť do tabuľky, dokonca aj záznamy s hodnotou NULL. Po spustení príkazu *SELECT* (*def. príkazu 150*) a zistení záznamov v jednotlivých partíciách sme zistili, že záznamy s NULL hodnotou sa nachádzajú v poslednej partícii (part_20XX). Z toho vyplýva, že ak je definovaná partícia, do ktorej spadajú všetky ostatné hodnoty, tak aj záznam s NULL hodnotami sa vloží do tejto partície.

```
SELECT 'PART_2020', COUNT(*)
 FROM tab_particie_null PARTITION (PART_2020)
 UNION ALL
SELECT 'PART_2021', COUNT(*)
 FROM tab_particie_null PARTITION (PART_2021)
 UNION ALL
SELECT 'PART_2022', COUNT(*)
 FROM tab_particie_null PARTITION (PART_2022)
 UNION ALL
SELECT 'PART_20XX', COUNT(*)
 FROM tab_particie_null PARTITION (PART_20XX);
```

Definícia príkazu 150: Príkaz *SELECT*, ktorý zistí počet záznamov v partíciách

Výsledok predchádzajúceho príkazu (*def. príkazu 150*) je zobrazený v *Tabuľke 38*. Tabuľka obsahuje 2 stĺpce, prvý stĺpec predstavuje názov partície a druhý stĺpec predstavuje počet záznamov v partícii.

Tabuľka 38: Počet záznamov v tabuľke *tab_particie_null*

Názov partície	COUNT(*)
PART_2020	0
PART_2021	1
PART_2022	1
PART_20XX	3

Po spustení príkazu *SELECT* pre poslednú partíciu (part_20XX) je možné vidieť, že záznamy s NULL hodnotami stĺpca *DATUM_VYTVORENIA* sú uložené v tejto partícii, pretože pokrýva všetky záznamy, ktoré nepatria do žiadnej predchádzajúcej partície.

Druhou možnosťou, ako vytvárať partície, je pomocou intervalu. Tieto partície sa vytvárajú automaticky. Na začiatku sa určí prvá, počiatočná partícia. Pri vkladaní záznamu sa vezme hodnota záznamu v stĺpci, podľa ktorého sa delí tabuľka na partície, a vloží sa do vhodnej partície. Ak táto partícia neexistuje vytvorí sa automaticky nová partícia. *Def príkazu 151* znázorňuje kód vytvorenia tabuľky s partíciami, ktoré sa vytvárajú dynamicky.

```
CREATE TABLE tab_interval_particie_null(
  id_zaznamu          NUMBER NOT NULL PRIMARY KEY,
  datum_vytvorenia    DATE,
  text                VARCHAR2(30))
PARTITION BY RANGE (datum_vytvorenia)
     INTERVAL( numtoyminterval(1, 'year'))(
  PARTITION part_1
          VALUES LESS THAN (TO_DATE('01.01.2021', 'DD.MM.YYYY')))/
```

Definícia príkazu 151: Tabuľka partícií *tab_interval_particie_null*

Po vytvorení tabuľky sme do nej vložili záznamy, ktoré zobrazuje *def. príkazu 149*. Záznamy s existujúcou hodnotou dátumu vytvorenia vložilo do tabuľky, pričom boli vytvorené príslušné partície. Avšak, systému sa nepodarilo vložiť do tabuľky záznamy s NULL hodnotou dátumu vytvorenia a zobrazil chybu ORA-14300 (*def. príkazu 152*).

```
SQL Error: ORA-14300: partitioning key maps to a partition outside
maximum permitted number of partitions
```

Definícia príkazu 152: Chyba ORA-14300

Systém nedokázal zaradiť NULL hodnotu kľúčového stĺpca do žiadnej partície, a ani nemohol vytvoriť novú partíciu, pretože NULL hodnoty kľúčov pre intervalové rozdelenie partícií nie je podporované systémom.

Riešenie tohto problému s NULL hodnotami pre kľúčový stĺpec, pomocou ktorého sa tabuľka rozdeľuje na partície, by mohol priniesť virtuálny stĺpec tabuľky, ktorý bude

nadobúdať existujúcu hodnotu dátumu vytvorenia, alebo transformuje NULL hodnotu dátumu vytvorenia na nami preddefinovanú hodnotu. Tabuľku *tab_interval_particie_null* rozšírime o virtuálny stĺpec *DATUM_DEF*, ktorý sa bude automaticky počítať na základe toho, či je alebo nie je dátum vytvorenia vyplnený. Ak bude dátum vytvorenia nadobúdať hodnotu NULL, tak sa nastaví preddefinovaný dátum 01.01.2999. V časti *PARTITION BY RANGE* je taktiež potrebné zmeniť stĺpec *DATUM_VYTVORENIA* na *DATUM_DEF*. Rozšírenú tabuľku popisuje *def. príkazu 153*.

```
CREATE TABLE tab_interval_particie_null(
  id_zaznamu          NUMBER NOT NULL PRIMARY KEY,
  datum_vytvorenia    DATE,
  text                VARCHAR2(30),
  datum_def           DATE GENERATED ALWAYS AS
           (COALESCE(datum_vytvorenia,
                     TO_DATE('01.01.2999', 'DD.MM.YYYY'))) VIRTUAL)
PARTITION BY RANGE (datum_def)
     INTERVAL( numtoyminterval(1, 'year'))(
  PARTITION part_1
           VALUES LESS THAN (TO_DATE('01.01.2021', 'DD.MM.YYYY')))/
```

Definícia príkazu 153: Tabuľka *tab_interval_particie_null* s virtuálnym stĺpcom

Po opätovnom vytvorení tabuľky a pridaní záznamov ako popisuje *def. príkazu 149* sme dostali informáciu, že všetky záznamy boli úspešne vložené do databázovej tabuľky. Názvy partícií, hodnoty intervalov a počet riadkov v danej partícií sú dostupné po spustení príkazu, ako popisuje *def. príkazu 154*.

```
SELECT partition_name, high_value, num_rows
 FROM   user_tab_partitions
  WHERE table_name = 'TAB_INTERVAL_PARTICIE_NULL'
   ORDER BY partition_name;
```

Definícia príkazu 154:Príkaz *SELECT* na zistenie partícií pre tabuľku

Ako je vidieť v *Tabuľke 39*, systém vytvoril nami definovanú počiatočnú partíciu a následne automaticky vytvoril chýbajúce partície pre rôzne roky záznamov, ktoré sme vkladali do tabuľky. Partície vytvorené systémom začínajú predponou SYS. V prípade NULL hodnôt sa vytvorila partícia SYS_P34693, ktorá pokryla záznamy s NULL hodnotami pre dátum vytvorenia.

Tabuľka 39: Informácie o partíciách pre tabuľku *tab_interval_particie_null*

NÁZOV PARTÍCIE	HODNOTA	POČET RIADKOV
PART_1	TO_DATE(' 2021-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS)	0
SYS_P34691	TO_DATE(' 2023-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS')	1
SYS_P34692	TO_DATE(' 2022-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS')	1
SYS_P34693	TO_DATE(' 3000-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS')	2
SYS_P34694	TO_DATE(' 2024-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS')	1

List partitioning

List partitioning umožňuj podobný spôsob rozdelenia tabuľky na partície ako Range partitioning, avšak rozdiel je v tom, že List partitioning definuje diskrétne hodnoty partícií, ktoré môžu záznamy nadobúdať. V tejto časti sa zameriame na spôsob spracovania záznamov s NULL hodnotami kľúčových stĺpcov pri ich ukladaní do tabuľky. Vytvorili sme tabuľku *tab_list_particie_null* rozdelenú na partície pomocou List partitioning. Databázová tabuľka *tab_list_particie_null* je rozšírená o ďalší atribút *KOD_STATU*. Na základe tohto atribútu definujeme vytvorenie štyroch partícií. Prvá partícia je vytvorená pre záznamy, ktoré nadobúdajú hodnoty SK alebo CZ v stĺpci *KOD_STATU*. Druhá partícia je vytvorená pre záznamy, ktoré v tomto stĺpci nadobúdajú hodnotu BE alebo FR, záznamy s hodnotou stĺpca *KOD_STATU* DE alebo AT sa radia do tretej partície a do poslednej vytvorenej partície zaraďujeme záznamy s hodnotou stĺpca *KOD_STATU* HU alebo PL. Definícia tabuľky *tab_list_particie_null* je znázornená v *def. príkazu 155*.

```
CREATE TABLE tab_list_particie_null(
  id_zaznamu          NUMBER NOT NULL PRIMARY KEY,
  datum_vytvorenia    DATE,
  kod_statu           VARCHAR2(2)
  text                VARCHAR2(30))
PARTITION BY LIST (kod_statu)(
     PARTITION part_1 VALUES ('SK', 'CZ'),
     PARTITION part_2 VALUES ('BE', 'FR'),
     PARTITION part_3 VALUES ('DE', 'AT'),
     PARTITION part_4 VALUES ('HU', 'PL'))/
```

Definícia príkazu 155: Definícia tabuľky *tab_list_particie_null*

Po vytvorení tabuľky sme sa do nej pokúsili vložiť záznamy, ktoré znázorňuje *def. príkazu 156*. Záznamy obsahovali aj NULL hodnoty stĺpca *KOD_STATU*, ale aj hodnoty, ktoré neboli vhodné do žiadnej partície.

```
INSERT INTO tab_list_particie_null
VALUES(1, TO_DATE('21.12.2022', 'DD.MM.YYYY'), 'SK','Text1');

INSERT INTO tab_list_particie_null
VALUES(2, TO_DATE('21.12.2021', 'DD.MM.YYYY'), NULL, 'Text2');

INSERT INTO tab_list_particie_null
VALUES(3, NULL, 'HU', 'Text3');

INSERT INTO tab_list_particie_null
VALUES(4, TO_DATE('21.10.2023', 'DD.MM.YYYY'), 'AT', 'Text4');

INSERT INTO tab_list_particie_null
VALUES(5, NULL, NULL, 'Text5');

INSERT INTO tab_list_particie_null
VALUES(6, NULL, 'GB', 'Text5');
```

Definícia príkazu 156: Záznamy na vloženie do tabuľky rozdelenej pomocou List partitioningu

Záznamy s vyplnenou hodnotou stĺpca *KOD_STATU* boli úspešne vložené do tabuľky, na druhej strane záznamy s NULL hodnotou alebo inou hodnotou stĺpca *KOD_STATU* nebolo možné vložiť do databázovej tabuľky. Pri vkladaní týchto záznamov do tabuľky systém vyvolal chybu ORA-14400, ktorej bližší popis zobrazuje *def. príkazu 157*.

```
SQL Error: ORA-14400: inserted partition key does not map to any
partition
```

Definícia príkazu 157: Chyba ORA-14400

Na základe tejto chyby, bolo potrebné rozšíriť definíciu tabuľky *tab_list_particie_null* o ďalšie typy partícií. Systém umožňuje použiť klauzulu *DEFAULT*. Následne sa všetky ostatné záznamy, ktoré nespĺňajú podmienky predchádzajúcich partícií, vložia do tejto časti. Záznamy s NULL hodnotou kľúčového stĺpca partícií budú tiež uložené v *DEFAULT* partícii. Ak by sme však chceli oddeliť záznamy s NULL hodnotami od záznamov, ktorých hodnota je vyplnená, ale nepatrí do žiadnej partície, systém umožňuje vytvoriť partíciu typu *NULL*. Nasledujúci príklad, ktorý zobrazuje *def. príkazu 158*, rozširuje

definíciu tabuľky *tab_list_particie_null* o ďalšie dve partície. Prvá bude pokrývať záznamy s NULL hodnotami kľúčových stĺpcov partícií a druhá partícia bude určená pre ostatné záznamy.

```
CREATE TABLE tab_list_particie_null(
  id_zaznamu          NUMBER NOT NULL PRIMARY KEY,
  datum_vytvorenia    DATE,
  kod_statu           VARCHAR2(2)
  text                VARCHAR2(30))
PARTITION BY LIST (kod_statu)(
     PARTITION part_1 VALUES ('SK', 'CZ'),
     PARTITION part_2 VALUES ('BE', 'FR'),
     PARTITION part_3 VALUES ('DE', 'AT'),
     PARTITION part_4 VALUES ('HU', 'PL'),
     PARTITION part_null VALUES (NULL),
     PARTITION part_default VALUES (DEFAULT))/
```

Definícia príkazu 158: Definícia tabuľky *tab_list_particie_null* rozšírená o ďalšie typy partícií

Následne sme sa opäť pokúsili vložiť vyššie uvedené záznamy do tabuľky. Všetky záznamy boli úspešne vložené. Pomocou príkazu *SELECT* (*def. príkazu 159*) sme skontrolovali, koľko záznamov sa nachádza v jednotlivých partíciách.

```
SELECT 'PART_1', COUNT(*)
 FROM tab_list_particie_null PARTITION (PART_1)
 UNION ALL
SELECT 'PART_2', COUNT(*)
 FROM tab_list_particie_null PARTITION (PART_2)
 UNION ALL
SELECT 'PART_3', COUNT(*)
 FROM tab_list_particie_null PARTITION (PART_3)
 UNION ALL
SELECT 'PART_4', COUNT(*)
 FROM tab_list_particie_null PARTITION (PART_4)
 UNION ALL
SELECT 'PART_NULL', COUNT(*)
 FROM tab_list_particie_null PARTITION (PART_NULL)
 UNION ALL
SELECT 'PART_DEFAULT', COUNT(*)
 FROM tab_list_particie_null PARTITION (PART_DEFAULT);
```

Definícia príkazu 159: Príkaz Select zobrazujúci počet záznamov v partíciách

Výsledok vyššie uvedeného príkazu (*def. príkazu 159*) je uvedený v *Tabuľke 40*.

Tabuľka 40: Počet záznamov v partíciách v tabuľke *tab_list_particie_null*

Názov partície	COUNT(*)
PART_1	1
PART_2	0
PART_3	1
PART_4	1
PART_NULL	2
PART_DEFAULT	1

List partitioning ponúka dve možnosti riešenia problému NULL hodnôt. Prvou možnosťou je použitie partície typu *DEFAULT*, no pri tejto voľbe je dôležité uvedomiť si, že okrem záznamov s NULL hodnotami sa do nej budú ukladať aj záznamy s hodnotami, ktoré nepatria do žiadnej inej partície. Druhou možnosťou je použiť partíciu typu *NULL*, takže sa do tejto partície budú ukladať len záznamy s NULL hodnotami kľúčového stĺpca partície.

Hash partitioning

Ďalší spôsob ako rozdeliť databázovú tabuľku na partície je Hash partitioning. Nie je potrebné explicitne špecifikovať partície ako pri delení tabuľky s použitím Range alebo List partitioningu. V tomto prípade špecifikujeme len stĺpec, podľa ktorého sa vytvorí

partícia a počet partícií, ktoré sa vytvoria. Ak počet partícií nie je zahrnutý v definícii, systém vytvorí štandardne jednu partíciu.

Def. príkazu 160 reprezentuje vytvorenie tabuľky *tab_hash_particie_null*, ktorá využíva Hash partitioning. Partície sa vytvárajú pomocou stĺpca *DATUM_VYTVORENIA* a implicitne sme definovali vytvorenie štyroch partícií.

```
CREATE TABLE tab_hash_particie_null(
  id_zaznamu          NUMBER NOT NULL PRIMARY KEY,
  datum_vytvorenia    DATE,
  text                VARCHAR2(30))
PARTITION BY HASH (datum_vytvorenia)
  PARTITIONS
/
```

Definícia príkazu 160: Definícia tabuľky *tab_hash_particie_null*

Po vytvorení tabuľky sme do nej postupne vkladali záznamy ako zobrazuje *def. príkazu 149*. Všetky záznamy boli úspešne vložené do tabuľky. Dokonca aj tie záznamy, ktoré nemali definovaný stĺpec *DATUM_VYTVORENIA*. Následne sme pomocou príkazu *SELECT* (*def. príkazu 161*) vyhľadali počet záznamov v jednotlivých partíciách.

```
SELECT 'PART_1', COUNT(*)
 FROM tab_hash_particie_null PARTITION (SYS_P34704)
 UNION ALL
SELECT 'PART_2', COUNT(*)
 FROM tab_hash_particie_null PARTITION (SYS_P34705)
 UNION ALL
SELECT 'PART_3', COUNT(*)
 FROM tab_hash_particie_null PARTITION (SYS_P34706)
 UNION ALL
SELECT 'PART_4', COUNT(*)
 FROM tab_hash_particie_null PARTITION (SYS_P34707);
```

Definícia príkazu 161: Definícia príkazu *SELECT* zobrazujúceho počet záznamov v partíciách tabuľky *tab_hash_particie_null*

Výsledok vyššie uvedeného príkazu je zobrazený v *Tabuľke 41*.

Tabuľka 41: Počet záznamov v partíciách v tabuľke *tab_hash_particie_null*

Názov partície	COUNT(*)
PART_1	4
PART_2	0
PART_3	0

Názov partície	COUNT(*)
PART_4	1

Po získaní záznamov z prvej partície (*PART_1*) sme získali výstup, ktorý je uložený v *Tabuľke 42*.

Tabuľka 42: Záznamy z partície SYS_P34704

id_zaznamu	datum_vytvorenia	text
2	21.12.2021	Text2
3	NULL	Text3
4	21.10.2023	Text4
5	NULL	Text5

Na tomto príklade je možné vidieť, že Hash partitioning je navrhnutý tak, že dokáže bez problémov spracovať záznamy s NULL hodnotami kľúčových stĺpcov partícií. Systém vloží záznam s nedefinovanými hodnotami do partície na základe výpočtu jeho hash funkcie. Vo všeobecnosti v prípade Hash partitioningu, záznamy s NULL hodnotou kľúčového stĺpca partícií môžu byť uložené v ľubovoľnej partícii.

3.9 Zhrnutie experimentálnych štúdií

Výskum našej práce by sme mohli rozdeliť do dvoch častí – optimalizačná časť a algoritmická časť. V úvode kapitoly *Výskum a vlastný prínos* sme sa venovali porovnaniu rôznych metód importu dát do databázy, pričom metóda dosahujúca najlepšie výsledky bola metóda importu pomocou dátovej pumpy. Obdobne to bolo aj pri exporte dát do databázy, pretože export pomocou dátovej pumpy dosahoval najlepšie výsledky. Bolo to z toho dôvodu, že import aj export pomocou dátovej pumpy sa vykonávajú len na strane servera, takže nie je potrebné prepájať stranu klienta so stranou servera. Ďalšou časťou výskumu bolo porovnanie rýchlostí príkazu *SELECT* v interných a externých tabuľkách, pričom interné tabuľky dosiahli omnoho lepšie výsledky.

Výskum zameraný na optimalizáciu príkazu *SELECT*, ktorý obsahoval analytické funkcie porovnával päť rôznych spôsobov. Naším cieľom bolo porovnať celkové náklady na vykonanie príkazov a optimalizovať ich tak, aby náklady boli čo najmenšie. Po vykonaní experimentov sme dospeli k záveru, že príkaz, ktorý využíval materializovaný pohľad, nad ktorým sme vytvorili index dosahoval najmenšie náklady na vykonanie príkazu obsahujúceho analytické funkcie.

Celý výskum využíva reálne dáta, ktoré bolo potrebné určitým spôsobom spracovať a navrhnúť dátové modely / dátové sklady na ich ukladanie. Pri návrhu sme vykonali

experimenty zamerané na rýchlosť vkladania veľkého množstva údajov do rôznych typov tabuliek, ktoré sme následne porovnali. Porovnávali sme štyri rôzne typy tabuliek, z ktorých sa ukázalo, že do klasickej tabuľky faktov sa údaje vložia najrýchlejšie. Zvyšné tabuľky mali čas vkladania dlhší, hlavne z dôvodu dodatočnej réžií pri vytváraní partícií a vkladania záznamov do vhodnej partície.

Na základe toho sme ďalej pokračovali optimalizáciou dátovej štruktúry, pričom sme porovnávali rýchlosti vyhľadávania záznamov v rôznych typoch tabuliek. Porovnávali sme celkové náklady troch príkazov *SELECT*. Prvý príkaz vyberal záznamy z definovaného roku, druhý príkaz z definovaného mesiaca a roku a posledný len z definovaného mesiaca. Aj keď sa pri vkladaní záznamov ukázalo, že do klasickej tabuľky faktov je možné vložiť záznamy najrýchlejšie, vyhľadávanie záznamov v klasickej tabuľke faktov neprinieslo také výsledky. V tomto prípade sa spomedzi vyhľadávania dokázalo, že tabuľky rozdelené na partície dosahujú najlepšie výsledky pri vyhľadávaní záznamov, čo môže byť kľúčové pokiaľ pracujeme s obrovským množstvom dát.

Ďalším komplexným predmetom skúmania bolo referencovanie funkcií v príkaze *SELECT,* v ktorom sme navrhli a experimentálne otestovali nami navrhnuté zlepšenie odkazovania sa na aliasy funkcií v rôznych častiach príkazu *SELECT*. Toto riešenie sme porovnali s existujúcim riešením predstaveným v Oracle 23c.

Okrem toho sme navrhli, implementovali a experimentálne otestovali spracovanie odkazov temporálnych funkcií v relačných databázach. V tomto výskume bližšie rozoberáme možnosti verziovania temporálnych funkcií a naše riešenia porovnávame už s existujúcimi.

Druhú časť výskumu sme venovali problematike nedefinovaných hodnôt a chybných dát, kde sme vytvorili metodiku pre spracovanie NULL hodnôt v stĺpcoch, ktoré predstavujú kľúčové stĺpce pre vytváranie partícií. Okrem toho sme sa snažili ukázať prácu s NULL hodnotami, akým spôsobom ich spracovať aby boli výsledky dosiahnutých riešení správne. S touto problematikou sme sa stretli aj pri práci na projekte, kde sme museli navrhnúť a implementovať príkazy umožňujúce doplnenie nedefinovaných hodnôt, pre ďalší výskum. V rámci výskumu sme pracovali aj na implementácií procedúr a funkcií, slúžiacich na analýzu leteckých dát.

4 Prehľad riešenej problematiky v rámci vedeckého portfólia

Analytickému spracovaniu dát sa venuje mnoho univerzít a vedeckých pracovísk po celom svete. Táto kapitola obsahuje prehľad zaujímavých článkov, výskumov a trendov v rámci analytiky, ktoré môžeme považovať za rámcové ohraničenie toho, čomu by sme sa chceli v rámci dizertačnej práce venovať resp. toho čomu sme sa už doteraz v rámci výskumu venovali.

Technická univerzita v Pekingu

Y.Wang, Y. Li, J. Sui, Y. Gao, „***Data Factory: An Efficient Data Analysis Solution in the Era of Big Data***“, 5TH IEEE INTERNATIONAL CONFERENCE ON BIG DATA ANALYTICS, pp. 28-32, 2020

- Článok sa venuje návrhu riešenia analýzy údajov veľkých dát - dátových tovární a implementácii softvéru na rýchle budovanie dátovej továrne. Vďaka vytvorenej dátovej továrne je možné získať výsledky analýzy a predikčné modely, následne sa realizuje vyhodnocovanie údajov.

Melbournská univerzita v Austrálii

X. Yu, „***Big Data Driven Model for NEW York Taxi Trips Analysis***“, 6TH IEEE INTERNATIONAL CONFERENCE ON BIG DATA ANALYTICS, pp. 1-4, 2021

- Článok skúma a analyzuje faktory, ktoré vplývajú na celkovú sumu cestovného, účtovaného taxikármi pre cestujúcich na výlete v meste New York. Cieľom štúdie je poskytnúť informácie cestujúcim, aby si výlety plánovali efektívnym a ekonomickým spôsobom.

Univerzita Albaha v Saudskej Arábii

F.A. Aijanobi, J.Lee, „Topological Data Analysis for Classification of Heart Disease Data“, 2021 IEEE INTERNATIONAL CONFERENCE ON BIG DATA AND SMART COMPUTING, pp 210 – 213, 2021

- Článok zdôrazňuje význam topologickej analýzy údajov v lekárstve. Výskum sa zameriava na analýzu a predpovedanie srdcových chorôb s čo najvyššou presnosťou.

Univerzita Montclair State v New Jersey

A.Saxena, S. A. Robila, „***Analysis of the New York City's Vehicle Crash Open Data***“, 9TH IEEE INTERNATIONAL CONFERENCE ON BIG DATA, pp. 6017 – 6019, 2021

- Článok popisuje vývoj nástroja vytvoreného na analýzu nehôd vozidiel v New Yorku. Nástroj analyzuje údaje za posledných 9 rokov a umožňuje rôzne typy vizualizácií.

Univerzita J. Selyeho v Komárne

S. Szenasi, „***Analysis of historical road accident data supporting autonomous vehicle control strategies***“, PEERJ COMPUTER SCIENCE, 2021

- Článok sa zameriava na analýzu historických dát z dopravných nehôd, na základe ktorých je možné predpovedať rizikové zóny, nebezpečné situácie a miesta dopravných uzlov pre autonómne vozidlá. Vďaka týmto informáciám je možné zabudovať do autonómnych vozidiel určitý typ stratégie, podľa ktorej by vedeli vyhodnotiť nepredvídateľné situácié a tým znížiť pravdepodobnosť potenciálnych incidentov.

Univerzita Magna Graecia v Catanzare

G. Agapito, C. Zucco, M. Cannataro, „***COVID-WAREHOUSE: A Data Warehouse of Italian COVID-19, Pollution, and Climate Data***“, INTERNATIONAL JOURNAL OF ENVIRONMENTAL RESEARCH AND PUBLIC HRALTH, 2020

- Výskum sa venuje návrhu a vývoji COVID-WAREHOUSE, čo je dátový sklad, ktorý modeluje integruje a uchováva údaje o COVID-19, ktoré denne zverejňujú talianske úrady. Okrem týchto údajov sa uchovávajú aj údaje o znečistení a klíme. Údaje sú uložené pomocou tabuliek faktov a dimenzií. COVID-WAREHOUSE umožňuje extrahovať vybrané údaje, ich analýzu a vizualizáciu.

M. Milano, C. Zucco, M. Cannataro, „***COVID-19 Community Temporal Visualizer: a new methodology for the network-based analysis and visualization of COVID-19 data***“, NETWORK MODELING AND ANALYSIS IN HEALTH INFORMATICS AND BIOINFORMATICS, 2021

- Výskum je zameraný na sieťovú metodiku slúžiacu na analýzu pandemických dát vírusového ochorenia COVID-19, ktoré zverejnili talianske úrady. Okrem pandemických dát sa vo výskume využívajú aj klimatické dáta. Tieto dáta obsahujú priestorové aj časové prvky, ktoré sú pre analýzu dôležité. Oba datasety sú navzájom zintegrované v dôsledku odhaľovania nových klimatických zmien. Cieľom metodiky je analyzovať súbor homogénnych dát pomocou štatistických testov a hľadanie zhodných či odlišných súborov dát, zaznamenať tieto informácie do grafu a použiť detekčný algoritmus na vizualizáciu a analýzu časopriestorového vývoja dát. Výsledkom výskumu je aplikácia navrhovanej metodiky, ktorá poskytuje sieťovú reprezentáciu opatrení COVID-19 v jednotlivých regiónoch.

State Grid Corporation of China

J. Liu, DP. ZOU, YC. Chen, ZJ. Chu, „***Analysis of Electric Vehicle Charging Facilities Operation Data***“, IEEE 5TH INTERNATIONAL CONFERENCE ON INTELLIGENT TRANSPORTATION ENGINEERING, pp 422 – 426, 2020

- Výskum sa zameriava na analýzu dát z nabíjania elektrických vozidiel. Dáta analyzuje z viacerých perspektív. Z pohľadu používateľa dochádza k analýze mobilného terminálu používateľa, schopnosti spotreby a platby. Z pohľadu času sa skúma spôsob nabíjania. Z pohľadu operátorov sa analyzuje distribúcia zaťaženia nabíjacích staníc, mieru využitia staníc, množstvo transakcií. Cieľom výskumu je predložiť návrhu na zlepšenie a zefektívnenie prevádzky nabíjacích zariadení.

Univerzita v Severnej Karolíne

AAR. Nayeem, M. Elshambakey, T. Dobbs, H. Lee, d. Crichton, YM. Zhu, C. Chokwitthaya, WJ. Tolone, I. Cho, „A Visual Analytics Framework for Distributed Data Analysis Systems“, 9TH IEEE INTERNACIONAL CONFERENCE ON BIG DATA, pp. 229 – 240, 2021

- Výskum je zameraný na návrh vizualizačného analytického framework-u, ktorý umožňuje používateľovi spustiť rôzne typy analýz v distribuovaných systémoch. Okrem toho framework umožňuje zlúčiť získané údaje z rôznych zdrojov, spustenie analýz, monitorovanie priebehu analýzy a vizualizáciu výsledkov. Dáta, na ktorých je testovaný koncept framework-u sú z oblasti výskumu vedy o Zemi a ekosystému.

Inštitút Mines-Telecom, Paríž

JL. Liu, N. Boukhelifa, JR. Egan, „***Understanding the Role of Alternatives in Data Analysis Practices***“, IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, pp 66 – 76, 2020

- Výskum je zameraný na analýzu alternatív, ktorú vykonávajú dátoví pracovníci. Sú to ľudia, ktorí skúmajú rôzne zdroje údajov, súbory hypotéz a teórií, algoritmy, metódy a nástroje. Súhrnne sa dá povedať, že sú to alternatívy. V rámci výskumu boli uskutočnené rozhovory s 12 dátovými pracovníkmi s rôznymi typmi odbornosti. Na základe toho sa vykonali 4 typy analýz, aby sa lepšie pochopilo:
  1. Skúmanie alternatív
  2. Rôzne predstavy o alternatívach a ako zapadajú do procesov
  3. Vysoko-úrovňové procesy alternatív
  4. Stratégie vytvárania, skúmania a riadenia alternatív
- Na základe zistení bol vytvorený rámec, ktorý zaznamenáva stupeň pozornosti, úroveň abstrakcie a analytické procesy. Vďaka tomuto rámcu je možné lepšie pochopiť, ako dátoví pracovníci zvažujú alternatívy vo svojich analýzach a ako môžu programátori vytvárať nástroje na ich podporu.

Univerzita v Južnej Kalifornii

D. Duncan, „***COVID-19 data sharing and collaboration***“, COMMUNICATIONS IN INFORMATION AND SYSTEMS, pp. 325 – 340, 2021

- Výskum je zameraný na návrh platformy centralizovaných webových archívov, ktoré uchovávajú multimodálne dáta súvisiacich s pandémiou COVID-19. Platforma umožňuje sprístupniť údaje vedeckej komunite po celom svete s cieľom uchovávania pandemických dát na jednom mieste a urýchleniu výskumu v tejto oblasti. Taktiež poskytuje nástroje na vizualizáciu a analýzu údajov, ale aj webovú stránku s informáciami a nástrojmi o školeniach, stretnutiach.

Univerzita Jimma v Etiópii

B. Umeta, T. Mulugeta, G. Mamo, S. Alemu, N. Berhanu, G. Milkessa, B. Mengistu, T. Melaku, „An analysis of COVID-19 information sources“, JOURNAL OF PHARMACEUTICAL POLICY AND PRACTICE, 2022

- Poskytovanie dôveryhodných informácií v súčasnej dobe o pandémii COVID-19 je pre občanov a komunity veľmi dôležité. Cieľom výskumu bolo analyzovať a zhodnotiť zdroje informácií o pandémii COVID-19 vo vidieckej komunite v juhozápadnej Etiópii. Výskum bol uskutočnený v 634 vidieckych komunitách juhozápadnej Etiópie. Zber údajov prebiehal rozhovormi s miestnymi obyvateľmi. Následne sa posudzovali faktory, ktoré ovplyvňovali informačné potreby prostredníctvom viacrozmerného logistického regresného modelu. Analýzou sa zistilo, že ako zdroj informácií domácnosti najviac využívali rádio a televíziu. V súvislosti s pandémiou COVID-19 domácnosti dôverovali vláde a zdravotníkom. Avšak účastníci neboli spokojní s množstvom informácií, ktoré dostávali. Preto požadovali dodatočné informácie, ktoré zahŕňali príčinu, symptómy ochorenia a príznaky.

Univerzita v Kolumbii

J. Lee, JH. Kim, C. Liu, G. Hripcsak, K. Natarajan, C. Ta, CH. Weng, „***Columbia Open Health Data for COVID-19 Research: Database Analysis***“, JOURNAL OF MEDICAL INTERNET RESEARCH, 2021

- Výskum je zameraný na problém obrovského množstva dát ochorenia COVID-19, ktoré každým dňom narastá, uchovávanie a dostupnosť týchto dát. Výsledkom výskumu je predstavenie verejne dostupnej databázy pod názvom Columbia Open Health Data , ktorá poskytuje klinické údaje pre hospitalizovaných pacientov s ochorením COVID-19, hospitalizovaných pacientov s chrípkou a hospitalizovaných pacientov všeobecne. Databáza poskytne výskumníkom a lekárom kvantitatívne merania klinických znakov súvisiacich s ochorením COVID-19 na lepšie porozumenie pandémie a boj proti nej.

Leibniz-FH School of Business, Nemecko

Dellnitz A., „***Big data efficiency analysis: Improved algorithms for data envelopment analysis involving large datasets***“, COMPUTERS AND OPERATIONS RESEARCH, 2021

- Výskum sa zameriava na problém veľkých údajov. Zaoberá sa analýzou obalov dát, čo je známy nástroj na určovanie efektívnosti rozhodovacích jednotiek, s použitím lineárneho programovania. Problém nastáva v dimenzionalite. Preto je potreba navrhnúť vylepšené algoritmy, ktoré zahŕňajú rôzne kritéria ukončovania programu a viacvláknové spracovanie. Okrem toho sa výskum venuje aj výkonnosti tejto stratégie veľkých dát pomocou numerickej analýzy na potvrdenie užitočnosti algoritmu. Výzva do budúcnosti je skúmanie, či je možné vyvinúť viacvláknový algoritmus kde sa výpočtový čas zvyšuje v priemere len lineárne vzhľadom na vstupné a výstupné rozmery (dimenzie).

Princess Nourah bint Abdulrahman University Riyadh, Saudská Arábia

M. John, H. Shaiba, „***Data Visualization as a Tool for Decision Making: Analysis based on COVID-19 Data***“, INTERNATIONAL CONFERENCE ON DECISION AID SCIENCES AND APPLICATION, 2021

- Výskum bol zameraný na vizuálnu analýzu údajov ochorenia COVID-19, ktorú možno použiť na pomoc úradom a zodpovedným orgánom pri rozhodovaní o zlepšené zdravotníckych zariadení, zmiernení alebo uvoľnení opatrení. V štúdii boli použité pandemické dáta zo Saudskej Arábie za obdobie od marca do júla roku 2020. Na vizualizáciu výsledkov analýz boli použité interaktívne mapy, tabuľky a teplotné mapy.

Organization for Economic Co-operation & Development, Paríž

G. Ciminelli, S. Garcia-Mandico, „***COVID-19 in Italy: An Analysis of Death Registry Data***“, JOURNAL OF PUBLIC HEALTH, pp. 723 – 730, 2020

- Výskum bol zameraný na analýzu úmrtnosti a rýchlosti šírenia medzi komunitami. Údaje, ktoré sa v štúdii analyzovali pochádzali zo 4100 obcí na severe Talianska. Išlo o údaje z registra úmrtí, ktoré sa porovnávajú s údajmi zo sčítania ľudu. Údaje sa využívajú v regresnom modeli. Výsledky výskumu ukázali, že na ochorenie COVID-19 zomrelo viac ako 0.15% miestnej populácie počas prvej vlny pandémie. Zistilo sa však, že oficiálne štatistiky tento počet obetí podceňujú. Oveľa väčší dopad pandémie bol na obyvateľoch domovov dôchodcov a v obciach s vysokým podielom seniorov, kde bola úmrtnosť dvakrát väčšia ako iných obciach. Záver výskumu poukazuje na to, že aktívnejší prístup pri zvládaní pandémie COVID-19 je kľúčový pre zníženie úmrtnosti obyvateľov. Dôležité je zvýšenie testovacej kapacity a zvýšenie schopnosti vyhľadávania kontaktov.

Záver

Dizertačná práca sa zameriava na oblasť analytického spracovania dát, cieľom ktorého je transformovať dostupné dáta na hodnotné informácie, ktoré sú kľúčové pre úspešné fungovanie mnohých firiem a organizácií. Tieto výstupy sa stávajú základom pre strategické rozhodnutia, vedúce k optimalizácii procesov, nárastu efektivity a celkovému napredovaniu.

Naším cieľom bolo zamerať sa na pozadie týchto procesov, efektivitu ukladania veľkého objemu dát do databáz a dátových skladov, pričom kľúčová je rýchlosť a výkonnosť vyhľadávania položiek v rámci úložiska. Dôraz sme kládli na správny návrh a optimalizáciu dátových a indexových štruktúr, ktoré sú základom pre dosiahnutie požadovanej efektivity. Okrem toho sa práca zaoberá návrhom vhodných algoritmov pre analytické spracovanie dát, v rámci čoho je dôležitá práca s agregačnými a analytickými funkciami. Pre zaistenie spoľahlivosti výsledkov je podstatná konzistencia a bezchybnosť spracovávaných dát. Z toho dôvodu bola práca zameraná aj na návrh a implementáciu vlastných algoritmov pre identifikáciu a odstránenie chybných, nedefinovaných údajov.

Dizertačná práca sa delí na niekoľko častí. Prvou z nich je *Analýza súčasného stavu*, v ktorej detailne popisujem: dátové sklady, slúžiace na ukladanie veľkého objemu dát; existujúce systémy analýzy dát; optimalizačné techniky ako sú indexy a delenie tabuliek na partície; štandardné, agregačné a analytické funkcie s demonštráciou na príkladoch. Druhá časť práce popisuje *Dáta* v rámci výskumu, z ktorých sme vybrali dáta z leteckej dopravy a dáta z dopravných systémov, na ktorých overujeme pravosť experimentov. Tretia časť je zameraná na *Výskum a vlastný prínos*, v ktorej sa venujem experimentálnym štúdiám v danej oblasti. Posledná, štvrtá časť pokrýva *Prehľad riešenej problematiky v rámci vedeckého portfólia*.

Vďaka tejto práci môžeme ďalej rozvíjať poznatky v rôznych oblastiach, ako je napríklad návrh metodiky na automatizované vytváranie indexov na základe dopytov, jej implementácia a testovanie v databázovom prostredí. Do ďalšej oblasti výskumu by sme mohli zaradiť predikciu neúplných dát, s čím sa spája identifikácia a vytváranie vzorov, na základe ktorých by sme vedeli získať chýbajúce informácie. Medzi ďalšiu oblasť výskumu by sme mohli zaradiť návrh optimalizačných stratégií využitia analytických funkcií pre časovo orientované databázy. Okrem vyššie uvedených oblastí, existuje mnoho ďalších tém, ktoré by sa dali skúmať v rámci tejto problematiky.

Publikácie

[HD1]

Parking Information System / Veronika Michalková, Martina Durneková, Marek Kvet.

In: Journal of Information, Control and Management Systems = JICMS. – ISSN 1336-1716. Roč. 17, č. 1 (2019), s. 111-121

[HD2]

School Information System / Martina Hrínová Durneková, Michal Kvet.

In: ICETA 2020: 18th IEEE International conference on emerging technologies and applications: Information and communication technologies in learning: proceedings. – 1. vyd. – Piscataway: Institute of Electronical and Electronics Engineers, 2020 – ISBN 978-0-7381-2366-0. – s. 176 – 181

Zaradené v SCOPUS

[HD3]

Principles of data warehouses in data analytics / Martina Durneková.

In: Mathematics in science and technologies: proceedings of the MIST conference 2021. – 1. vyd. – [S.l.]: [s.n.], 2021 – ISBN 9798748088183. – s. 21-25

[HD4]

Data Import and Export Methods / Martina Durneková, Michal Kvet

In: 29th Conference of Open Innovations Association FRUCT [print, electronic]: Proceedings. – ISSN: 2305-7254. – 1. Vyd. – 2021: FRUCT Oy, 2021. – ISBN: 978-952-69244-5-8. – s. 423-428 [online]

[HD5]

Using Multi-Index Database Layer in Ad-Hoc Communication Networks / Michal Kvet, Martina Durneková

In: IWSSIP 2021: 28th International Conference on Systems, Signals and Image Processing. Bratislava, Slovakia. June 2-4, 2021. – 1 vyd. – Bratislava: Spektrum STU, 2021. – ISBN: 978-80-227-5112-4. – s. 285-296.

[HD6]

Information System for Asset Management in Buildings / Martina Durneková, Michal Kvet

In: ICETA 2022: 20th Anniversary of IEEE International Conference on Emerging eLearning Technologies and Applications: proceedings – 1. vyd. – Piscataway: Institute of Electrical and Electronics Engineers, 2022. ISBN 979-8-3503-2032-9 – s. 141-146

Zaradené v SCOPUS

Prístup: https://ieeexplore.ieee.org/document/9974654

[HD7]

Optimization of the SELECT Statement Containing Window Functions / Martina Hrínová Durneková, Michal Kvet

In: Proceedings of The International Conference on Information and Digital Technologies (IDT) 2023, - 1 vyd. – Danvers: Institute of Electrical and Electronics Engineers, 2023 – ISBN 979-8-3503-0586-9 – s. 267-272

Zaradené v SCOPUS

Prístup: https://ieeexplore.ieee.org/document/10194457

[HD8]

Treating Temporal Function References in Relational Database Management System / Michal Kvet, Jozef Papan, Martina Hrínová Durneková

In: IEEE Access, 2024, Doi: 10.1109/ACCESS.2024.3387046, vol. 12, s. 54535-54552

Zaradené v SCOPUS a WoS

Aktuálne v tlači

[HD9]

Referring Null Values in Partitioned Tables / Martina Hrínová Durneková, Michal Kvet

In: 35th Conference of Open Innovations Association FRUCT, 2024

Zaradené v SCOPUS

Aktuálne je prijatý, ale nie je ešte indexovaný

Zoznam použitej literatúry

[1] Breslin M., „Data Warehousing Battle of the Giants: Comparing the Basics of the Kimball and Inmon Models,“ *BUSINESS INTELLIGENCE JOURNAL,* pp. 6 - 20, 2004.

[2] Clarke J, „Exploiting the Operation System,“ rev. *SQL Injection Attacks and Defense*, 2009, pp. 271-315.

[3] Datta A. a Thomas H., „The cube data model: a conceptual model and algebra for on-line analztical processing in data warehouses,“ *Decision Support System,* zv. 27, pp. 289-301, 1999.

[4] EUROCONTROL, „EUROCONTROL Supporting European Aviation,“ [Online]. Available: www.eurocontrol.int.

[5] Fry N., Brownbridge P. a Rhone S., User’s Guide for Oracle Data Visualization Desktop, 2019.

[6] Harvey R., Oracle Cloud Data preparation in Oracle Analytics Cloud, 2017.

[7] Kvet M. a Matiaško K., „Transaction Management in Temporal System,“ *Iberian Conference on Information Systems and Technologies,* 2014.

[8] Kvet M., Matiaško K. a Kvet M., „Complex time management in databases,“ *Central European Journal of Computer Science,* zv. 4, pp. 269-284, 2014.

[9] Laszewski T. a Nauduri P., „Database Schema and Data Migration,“ rev. *Migrating to the cloud,* 2012, pp. 93-130.

[10] Malik U., Goldwasser M. a Johnston B., SQL for Data Analytics, Birmingham, 2019, ISBN: 978-1-78980-735-6

[11] Mannino M., Hong S. N. a Choi I. J., „Efficiency evaluation of data warehouse operations,“ *Decision Support Systems,* zv. 44, pp. 883-898, 2008.

[12] Matiaško K., Vajsová M. a Kvet M., Pokročilé databázové systémy 2. diel - Architektúra, programovanie s objektmi a XML, Žilina: EDIS, 2017.

[13] MATLAB, MATLAB numerical computing, 2014.

[14] Microsoft, „What is Power BI?,“ 2022. [Online]. Available: https://docs.microsoft.com/sk-sk/power-bi/fundamentals/power-bi-overview.

[15] Pérez J. M., Berlanga R., Aramburu M. J. a Pedersen T. B., „Integrating Data Warehouses with Web Data: A Survey,“ *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, zv. 20, pp. 940-955, 2008.

[16] Stenoish J., „SQL*Loader Express Mode Loading for Oracle Database,“ Oracle, 2021.

[17] Swonger R. F., *Full Transportable Export and Import,* Oracle, 2021.

[18] Thomas B., *Do More with Data Pump: Tips and Techniques,* Dallas.

[19] Venables W. N. a Smith D. M., An Introduction to R: A Programming Environment for Data Analysis and Graphics, version 4.2.1, 2022.

[20] Williams D., Beauregard W., DiPirro S., Kalogeropoulos J., Payne R., Phillips R., Sakayeda M., Stenoish J. a Swonger R., *Oracle Database Utilities, 19c*, Oracle, 2022.

[21] Kvet M., Šalgová V., Kvet M. a Matiaško K., "Master Index Access as a Data Tuple and Block Locator", Conference of Open Innovation Association FRUCT, 2019, pp. 176-183.

[22] Oracle, "The Oracle Optimizer Explain the Execution Plan", 2018.

[23] Harvey R., Getting Started with Oracle Analytics Cloud, 2023.

[24] Bharti Joshi, R.D.Morena, "An Efficient Query Optimization for Object Oriented Database", International Conference on Computing, Communication, Control and Automation, 2017.

[25] Khaled Saleh Maabreh, "Optimizing Database Query Performance Using Table Partitioning Techniques", International Arab Conference on Information Technology, 2018, ISBN 978-172810385-3

[26] Oracle, "The Oracle Optimizer Explain the Explain Plan", 2017

[27] Oracle, "Understanding Optimizer Statistics", 2012

[28] Oracle, "Database Performance Tuning Guide and Reference", 2002

[29] Rani G., Sharma T., a Sharma A., "Future Database Technologies for Big Data Analytics", Proceedings of the 2023 International Conference on Intelligent Systems for Communication, IoT and Security, 2023, pp. 349-354

[30] Chong E.I., Perry M., a Das S., "Improving RDF Query Performance Using In-Memory Virtual Columns in Oracle Database", *Proc Int Conf Data Eng*, 2019, vol. 2019-April, pp 1814 - 1819, ISBN 9781538674741

[31] Kersten M.L., Zhang Y., Katsogridakis P., Koutsourakis p., a Van Ruth J., "Database Resource Allocation Based on Resilient Intermediates", *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, 2018, vol. 2018-December, pp. 314 - 319, ISBN 9781538678992

[32] Kvet M., "Impact of Disc Types on Database Performance", *2022 IEEE 16th International Scientific Conference on Informatics, Informatics 2022 - Proceedings*, 2022, pp. 188-195, ISBN 9798350310344

[33] Li W., Li N., Yan J., Zhang Z., Yu P., a Long G., "Secure and High-Quality Watermarking Algorithms for Relational Database Based on Semantic", *IEEE Trans Knowl Data Eng*, 2022, ISSN 15582191

[34] Kvet M., a Papan J., "Enhancing Analytical Select Statements Using Reference Aliases", IEEE Access, 2024, vol. 12, pp 27311-27330

[35] Myalapalli V.K., a Savarapu P.R., "High performance SQL", *11th IEEE India Conference: Emerging Trends and Innovation in Technology, INDICON 2014*, 2015, Doi: 10.1109/INDICON.2014.7030467

[36] Chiheb F., Boumahdi F., Bouarfa H., "A New Model for Integrating Big Data into Phases of Decision-Making Process", Procedia Computer Science - *The 2nd International Conference on Emerging Data and Industry 4.0*, 2019, vol 151, pp. 636-642, ISSN 18770509, Doi: 10.1016/j.procs.2019.04.085

[37] Moktadir A., Istiak Chowdhury N.M., "Subject Oriented Data Partitioning - A Proposed Data Warehousing Schema", *1st International Conference on Advances in Science, Engineering and Robotics Technology*, 2019, ISBN: 978-172813445-1

[38] Surianszah B., Ilham A.A., Paundu A.W., "Optimization of Data Warehouse Architecture to Improve Information System Performance", *International Conference on Computer Science, Information Technology and Engineering*, 2023, pp 240-245, ISBN: 979-835032095

[39] Golfarelli M., Rizzi S., "From Star Schemas to Big Data: 20+ Years of Data Warehouse Research", A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years , 2018, vol 31, pp 93-107

[40] Khan R.A., Quadri S.M.K., "Business Intelligence: An Integrated Approach", Business Intelligence Journal, 2012, vol. 5, pp. 64-70

[41] Evans R., R-Programming, 2014

[42] Matiaško K., Vajsová M., Zábovský M., Chochlík M., Databázové systémy - Základy databázových systémov, Žilina, EDIS, 2008, ISBN: 978-80-8070-820-7

[43] Matiaško K., Vajsová M., Kvet M., Pokročilé databázové systémy 1. diel - Umenie programovania a administrácie, Žilina, EDIS, 2017, ISBN: 978-80-554-1311-2

[44] Oracle, "Oracle Partitioning", 2019

[45] Šalgová V., Matiaško K., "The Effect of Partitioning and Indexing on Data Access Time", *Conference of Open Innovation Association, FRUCT*, 2021, pp. 301-306

[46] Oracle, VLDB and Partitioning GUIDE, Web: https://docs.oracle.com/en/database/oracle/oracle-database/23/vldbg

[47] Wisal K., Cheng Z., Bin L., Teerath K., Ejaz A, "Robust Partitioning Scheme for Accelerating SQL Database", *International Conference on Emergency Science and Information Technology, ICESIT*, 2021, pp 369-376

[48] Lim L., "Elastic Data Partitioning for Cloud based SQL Processing Systems", *International Conference on Big Data*, 2013, pp 8-16

[49] Šalgová V., Matiaško K., "Reducing Data Access Time using Table Partitioning Techniques", *International Conference on Emerging eLearning Technologies and Applications*, 2020, pp. 564-569

[50] Oracle, Express Mode Loading with SQL Loader in Oracle Database 12c, 2013

[51] Lv T., Yan P., "A Lightweight JSON Storage Method Based on Adjacency Linked Lists", *4th International Conference on Computer Engineering and Intelligent Control, ICCEIC,* 2023, pp 428-432

[52] Lv T., Yan p., "A Probabilistic Model for JSON Data", *2nd International Conference on Computational Modeling, Simulation and Data Analysis, CMSDA,* 2022, pp. 276-280

[53] Fadlallah H., "SQl Server JSON functions: a bridge between NoSQL and relational worlds", 2020, [Online]. Available: https://www.sqlshack.com/sql-server-json-functions-a-bridge-between-nosql-and-relational-worlds/

[54] Oracle, "Generation of JSON Data Using SQL", 2023, [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/23/adjsn/generation.html

[55] Oracle, "SQL/XML (SQLX): Generating XML using SQL", 2019, [Online]. Available: https://oracle-base.com/articles/misc/sqlxml-sqlx-generating-xml-content-using-sql

[56] Liu H., Chen Q., Pan N., Sun Y., An Y., Pan D., "UAV Stocktaking Task-Planning for Industrial Warehouses Based on the Improved Hybrid Differential Evolution Algorithm", *IEEE Trans Industr Inform,* 2022, vol. 18, pp. 582-591, ISSN: 19410050

[57] Oracle, "Oracle-Base - Oracle 23c Articles", 2023, [Online]. Available: https://oracle-base.com/articles/23c/articles-23c

[58] Oracle, "Oracle Database 23c: The Next Long Term Support Release", 2023, [Online]. Available: https://blogs.oracle.com/database/post/oracle-database-23c-the-next-long-term-support-release

[59] Kuhn D., Kyte T., "Export Oracle Database Architecture", Apres 2021, ISBN: 1484274989

[60] Cornejo R., "Dynamic Oracle Performance Analytics Using Normalized Metrics to Improve Database Speed", ISBN: 978-1484241363

[61] Rovnyagin M.M., Dmitriev S.O., Hrapov A.S., Maksutov A.A., Turovskiy I.A., "Database Storage Format for High Performance Analytics of Immutable Data", *IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering,* 2021, pp. 618-622, ISBN: 9780738142753

[62] Arora G., Kalra S., Bhatia A., Tiwari K., "Palmprint Hashing Network for Indexing Large Databases to Boost Identification", *IEEE Access,* 2021, vol. 9, pp. 145913-145928, ISSN: 21693536

[63] Arora s., "A Comparative Study on Temporal Database Models: A Survey", *International Symposium on Advanced Computin and Communication,* 2015, pp 161-167

[64] Chavan S., Hopeman A., Lee S., Lui D., Mylavarapu A., Soylemez E., "Accelerating Joins and Aggregations on the Oracle In-Memory Database", *34th International Conference on Data Engineering,* 2018, pp.1453-1464

[65] Nindito H., Sano A.V.D., Condrobimo A.R., "Comparative Study of Storing Unstructured Data Type Between BasicFile and SecureFile in Oracle Database 12c", *International Conference on Information Management and Technology,* 2016, pp. 146-149

[66] Kvet M., "Enhanced Data Locking to Serve ACID Transaction Properties in the Oracle Database", *Conference of Open Innovation Association, FRUCT,* 2023, pp. 73-80

[67] Kvet M., "Developing Robust Date and Time Oriented Applications in Oracle Cloud: A comprehensive guide to efficient date and time management in Oracle Cloud 1", Amazon.com, 2024, [Online].

[68] Nuijten A., Barel P., "Modern Oracle Database Programming : Level up your Skill Set to Oracle's Latest and Most Powerful Features in SQL, PL/SQL and JSON", p. 576

[69] Maté J., Šafařík J., "Transformation of Relational Database to Transaction-Time Temporal Databases", *2nd Eastern European Regional Conference on the Engineering of Computer Based Systems,* 2011, pp. 27-34

Zoznam skratiek

AIRAC	Aeronautical information regulation and control	Regulácia a riadenie leteckých informácií
BI	Business Intelligence	Dátová analytika
CBO	Cost-Based Optimization	Cenovo orientovaná optimalizácia
FIR	Flight Information Region	Vzdušný priestor
MATLAB	Matrix Laboratory	
OLAP	Online Analytical Processing	Technológia na analytické spracovanie dát
OLTP	Online Transaction Processing	Technológia na transakčné spracovanie dát
RBO	Rule-Based Optimization	Optimalizácia založená na pravidlách

Zoznam obrázkov

Zoznam tabuliek

Zoznam definícií príkazov

Prílohy

Zoznam príloh

Príloha A – Obsah SD karty

Príloha B – Dátový model leteckej dopravy

Príloha C – Model dátového skladu chodcov

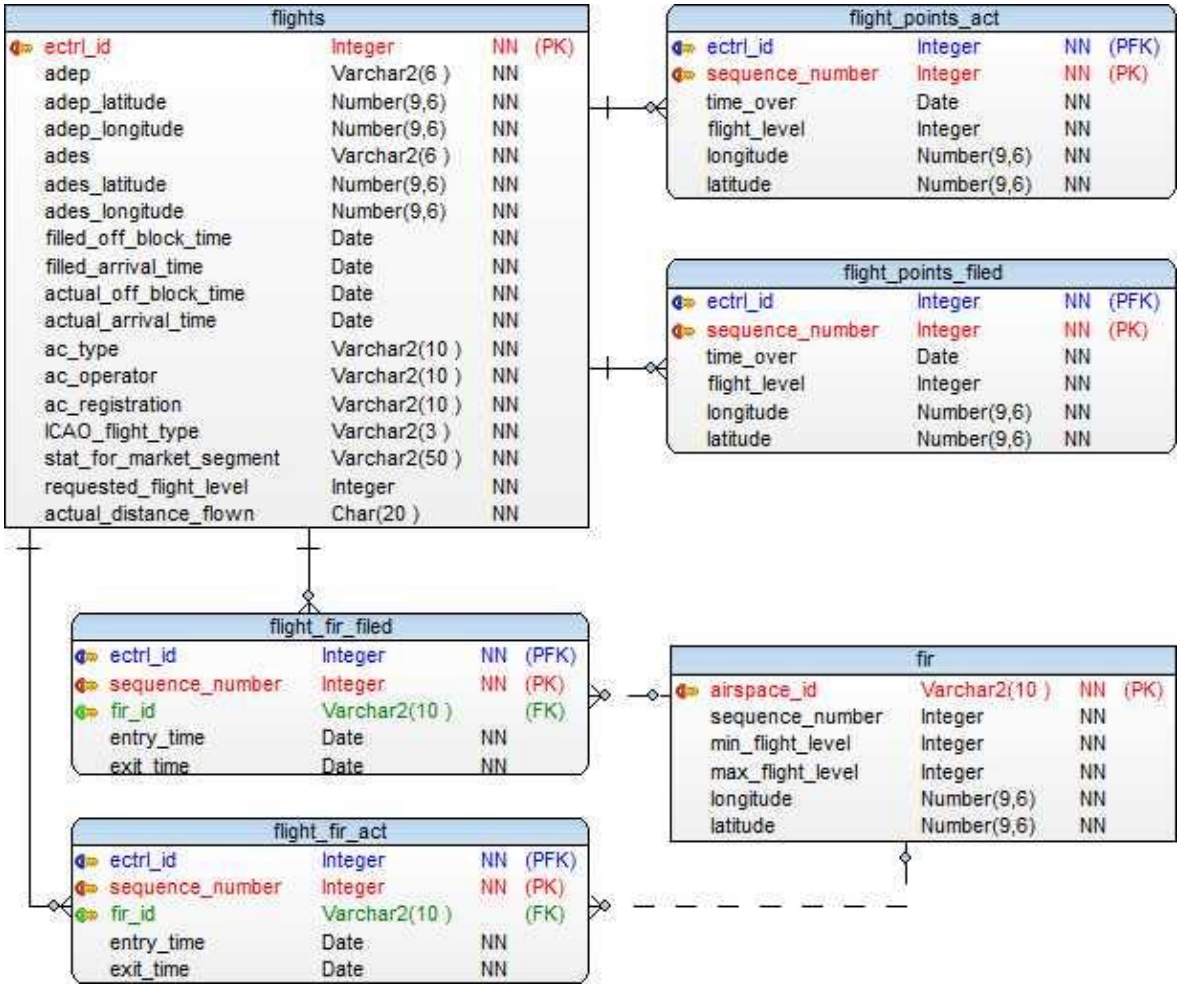
Príloha D – Model dátového skladu dopravných systémov

Príloha A – Obsah SD karty

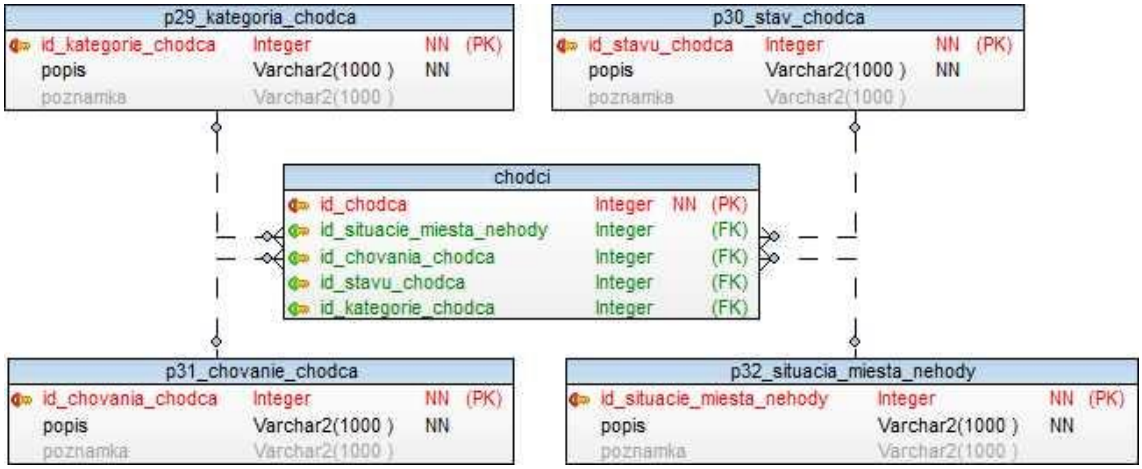
Priložená SD karta obsahuje:

- Dizertačnú prácu v elektronickej podobe (formát PDF)
- Dátový model pre dáta z leteckej dopravy
- Model dátového skladu chodcov
- Model dátového skladu dopravných systémov

Príloha B – Dátový model leteckej dopravy



Príloha C – Model dátového skladu chodcov



Príloha D – Model dátového skladu dopravných systémov