ŽILINSKÁ UNIVERZITA V ŽILINE

FAKULTA RIADENIA A INFORMATIKY

SPRACOVANIE A VYHĽADÁVANIE V NEŠTRUKTÚROVANÝCH DATABÁZACH

DIZERTAČNÁ PRÁCA

Evidenčné číslo: 28360020213002

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, 2021

Ing. Roman Čerešňák

Poďakovanie

Chcel by som sa poďakovať svojmu školiteľovi doc. Ing. Michalovi Kvetovi, PhD. za jeho rady, pripomienky a návrhy, ktorými ma usmerňoval pri písaní mojej práce.

Rád by som sa tiež poďakoval svojmu školiteľovi špecialistovi prof. Ing. Karolovi Matiaškovi, PhD. za jeho trpezlivosť a drahocenný čas, ktorý mi počas môjho štúdia venoval.

Nakoniec by som sa rád poďakoval kolegom z Katedry informatiky Fakulty riadenia a informatiky Žilinskej univerzity v Žiline za mnohé zaujímavé myšlienky, podnety a cenné odborné diskusie.

ABSTRAKT V ŠTÁTNOM JAZYKU

ČEREŠŇÁK, Roman: Spracovanie a vyhľadávanie v neštruktúrovaných databázach [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, 2021.

Dizertačná práca sa zaoberá problematikou správy a komplexným návrhom migračného mechanizmu, ktorý zabezpečuje manažment transformačných pravidiel potrebných nielen na presun dát, ale aj presun schémy z relačných databáz na nerelačné. V prípade nutnosti je tento mechanizmus možné aplikovať aj smerom z nerelačnej databázy na relačnú databázu. V procese analýzy existujúcich možností sme dospeli k záveru, že ponechanie voľnosti dát a dátovej štruktúry v nerelačných databázach spôsobuje nadmernú nekonzistentnosť, čo v prípade nutnosti transformačného procesu spôsobuje zdĺhavé konsolidačné čakanie, a tým predlžuje čas celkovej zmeny. Výhodou nami navrhnutých metód je automatická konsolidácia údajov na základe stanovenej granularity, a tým zníženie času potrebného na transformáciu. K ďalšej výhode nami navrhnutých metód patrí zadefinovanie pravidiel, ktoré riešia nesúlad medzi dátovými typmi a ich následnú dátovú zmenu. V práci sme taktiež definovali mechanizmus, ktorý dokáže ovplyvňovať dáta, ktoré boli v systéme nahromadené, dátami ktoré do procesu vstupujú („real-time data") a nahromadené dáta ovplyvňujú. Ďalej sme vytvorili koncept automatického prispôsobovania procesov na základe výkonu, čím sme optimalizovali paralelizmus, a tým dokážeme spustené procesy pridávať a odoberať. Taktiež bola navrhnutá metóda verzionovania, čo nám umožňuje v prípade problému vo veľkej miere redukovať čas, a tým začať proces od bodu zlyhania. V závere práce sa venujeme zrýchleniu vyhľadávacieho procesu v nerelačných databázach MongoDB a DynamoDB, vytváraniu temporálnych indexov v pamäti a zlepšenie implementácie umelej inteligencie. Experimentálne sme odhadli pravidlo aplikovania vyhľadávania umelej inteligencie pri veľkom množstve údajov („Big Data").

Kľúčové slová: Štruktúrovaná databáza, neštruktúrovaná databáza, veľké dáta, migračný mechanizmus, vyhľadávanie v neštruktúrovaných databázach, paralelizmus, migračný mechanizmus, ovplyvňovanie dát v reálnom čase.

ABSTRAKT V CUDZOM JAZYKU

ČEREŠŇÁK, Roman: Processing and searching in unstructured databases [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, 2021.

Main objective of this dissertation thesis is the management and complex design of migration mechanism which handles management of transformation rules used not only for data transfers, but also in the transfer of schema from relational to non-relational databases. This mechanism can also be applied in the other direction - from a non-relational database to a relational one. In the process of analysis of the existing solutions, we came to the conclusion, that maintaining flexibility of data and data structure in non-relational databases causes enormous inconsistencies. In the necessity of transformation, these inconsistencies lead to higher computational times of transformation related to ineffective consolidation of data. The advantages of our original methods are automatic consolidation of a database on the basis of given granularity and time reduction of the transformation process. Another advantage of our original methods lays in the definition of rules, which solve the inconsistency between data types and their following data change. We also define a mechanism, which can influence data, which have been accumulated in the system, with the use of data which enter the process during its' run (real-time data) and influence accumulated data. We created the concept of automatic scaling of the processes based on the performance, which we used for the purposes of optimization of parallelism. With this mechanism, we can add or remove running processes. Also, new versioning method, which can be used to start the process at the point of failure, and which leads to time reduction as well, was designed and implemented. In the end, we deal with the improving of the query performance in non-relational database MongoDB and DynamoDB, creating temporal in-memory indexes, and improving searching performance with the use of machine learning. We experimentally estimated the rule of applying machine learning to Big Data.

Key words: Structured database, unstructured database, big data, migration mechanism, search in unstructured databases, parallelism, migration mechanism, influencing data in real time.

Obsah

1 Úvod do problematiky ... 15
2 Požiadavky na spracovanie a vyhľadávanie v neštruktúrovaných databázach a stanovenie cieľov ... 17
3 Ciele práce ... 19
4 Historický vývoj a súčasný stav transformačných mechanizmov ... 23
4.1 Správa štruktúry dátovej schémy a presun údajov ... 23
5 Distribuované spracovanie dát ... 27
5.1 Aktuálny stav distribuovaného spracovania dát ... 32
5.2 Práce zameriavajúce sa na riešenie problémov s distribuovaným spracovaním dát ... 34
5.2.1 Optimalizácia paralelizmu ... 35
5.2.2 Optimalizácia ceny ... 35
5.2.3 Ochrana údajov ... 36
5.2.4 Dôvod skúmania danej problematiky ... 37
5.3 Automatické škálovanie vlákien na základe dopytu ... 37
5.3.1 Škálovanie smerom nahor na základe prichádzajúcich údajov ... 40
5.3.2 Škálovanie smerom nadol na základe prichádzajúcich údajov ... 42
5.3.3 Výpočet prahovej hodnoty na základe vyťaženia servera ... 42
5.3.4 Medziregionálna replikácia ... 43
5.3.5 Experimentálna činnosť ... 43
5.4 Spoľahlivosť systému ... 48
5.4.1 Aktuálny stav riešenia spoľahlivosti systému ... 49
5.4.2 Dôvod zaoberania sa skúmanou problematikou spoľahlivosti systému ... 52
5.4.3 Databázový prístup pri stanovení replikačného koeficientu dát ... 52
5.4.3.1 Automatické škálovanie smerom nahor ... 57
5.4.3.2 Automatické škálovanie smerom nadol ... 58
5.4.4 Experimentálna činnosť replikačnej metódy ... 58
5.5 Zvýšenie paralelizmu pri migrácii dát ... 63
5.5.1 Štúdie zaoberajúce sa paralelným spracovaním dát ... 64
5.5.2 Pomocná databáza Redis ... 67
5.5.3 Pomocná databáza Memcached ... 68
5.5.4 Dôvod skúmania problematiky zvyšovania paralelizmu ... 68
5.5.5 Naše riešenie zvýšenia paralelného spracovania dát ... 69
5.5.6 Experimenty potvrdzujúce efektivitu našej paralelnej metódy ... 74
5.6 Vytváranie verzií dát počas migračných procesov v cloud-ovom prostredí ... 79
5.6.1 Existujúce riešenia venujúce sa problematike vytvárania verzií ... 79
5.6.2 Dôvod skúmania verzionovania údajov ... 81
5.6.3 Nami vytvorené riešenie problematiky verzionovania dát ... 82
5.6.3.1 Objektové porovnávanie (Object comparator) ... 84
5.6.4 Experimentovanie s dátami pri verzionovaní údajov ... 87
5.6.5 Zhrnutie metódy verzionovania údajov ... 91

5.7 Bezpečnosť spracovania dát v reálnom čase ... 92
5.7.1 Existujúce riešenia zaoberajúce sa bezpečnosťou ... 92
5.7.2 Dôvod skúmania bezpečnosti spracovania dát v reálnom čase ... 96
5.7.2.1 Úprava dát v reálnom čase ... 97
5.7.2.2 Presun dát medzi databázou a dátovým skladom ... 98
5.7.3 Experimenty pre metódu presunu dát v reálnom čase ... 100
5.7.4 Zhrnutie riešenia metódy presúvania údajov v reálnom čase ... 104
5.8 Ovplyvňovanie nahromadených dát reálnymi dátami ... 105
5.8.1 Aktuálny stav skúmanej problematiky ovplyvňovania nahromadených dát reálnymi dátami ... 106
5.8.2 Dôvod skúmania ovplyvňovania nahromadených dát reálnymi dátami ... 107
5.8.3 Prínos do problematiky ovplyvňovania nahromadených dát reálnymi dátami spojenými s prichádzajúcimi dátami v reálnom čase ... 108
5.8.3.1 Vrchná vetva navrhnutej architektúry ... 109
5.8.3.2 Spodná vetva navrhnutej architektúry ... 111
5.8.4 Apache Hive s externou tabuľkou ... 112
5.8.4.1 MapReduce ... 112
5.8.4.2 Spojenie spodnej a vrchnej vetvy nami navrhnutej architektúry ... 114
5.8.5 Experimentálna časť ... 115
5.8.5.1 Porovnanie rýchlosti MapReduce vs. Hive ... 116
5.8.6 Zistenie výhodnosti novo navrhnutého procesu oproti úprave dát po transformácii so zväčšujúcim sa počtom údajov ... 117
6 Vyhľadávanie v nerelačných databázach ... 121
6.1 Vyhľadávanie na disku ... 121
6.2 Vyhľadávanie v pamäti ... 122
6.3 Porovnanie súčasných databáz ... 124
6.3.1 Základné informácie o databázach ... 124
6.3.1.1 Sql vs. NoSql ... 125
6.3.1.2 Oracle vs. Sql Server vs. MySql ... 126
6.3.2 NoSql databázy ... 127
6.3.2.1 Databáza typu kľúč / hodnota (Key-Value Store) ... 128
6.3.2.2 Databáza založená na dokumentoch ... 129
6.3.2.3 Databáza založená na stĺpcoch ... 130
6.3.2.4 Databáza NoSql založená na grafoch ... 131
6.3.3 Dôvod skúmania danej problematiky ... 132
6.3.4 Experimenty súvisiace s rýchlosťou základných operácií pri relačných a nerelačných databázach ... 132
6.3.4.1 Výsledok experimentu graficky ... 134
6.4 Metóda na zefektívnenie vyhľadávania v nerelačných databázach ... 136
6.4.1 Existujúce riešenia zaoberajúce sa zefektívnením vyhľadávania v nerelačných databázach ... 136
6.4.2 Dôvod skúmania danej problematiky vyhľadávania v nerelačných databázach ... 138
6.4.3 Naše riešenie súvisiace s vyhľadávaním v nerelačných databázach ... 138

6.4.3.1 Moduly s medzipamäťou 138
6.4.3.2 Dátový elastický modul 141
6.4.4 Experimenty potvrdzujúce efektívnosť nami navrhnutej metódy 142
6.4.4.1 Experimenty s relačnou databázou Oracle 142
6.4.4.2 Experimenty pre nerelačnú databázu DynamoDB 143
6.4.4.3 Experimenty s databázou v pamäti Redis 144
6.4.4.4 Porovnanie konečných výsledkov 145
6.4.5 Stručné zhrnutie výsledkov 148
6.5 Metóda na zrýchlenie vyhľadávania dát v nerelačnej databáze MongoDB 150
6.5.1 Práce zaoberajúce sa vyhľadávaním dát v nerelačných databázach pomocou relačných databáz 150
6.5.2 Dôvod skúmania problematiky zrýchľovania vyhľadávania dát v nerelačnej databáze MongoDB 153
6.5.3 Nami navrhnuté riešenie skúmanej problematiky 154
6.5.4 Experimentálna časť slúžiaca na overenie nášho prístupu 156
6.5.5 Zhrnutie výsledkov metódy zrýchľovania vyhľadávania dát v nerelačnej databáze MongoDB 159
7 Výsledky práce a diskusia 161
7.1 Výsledky práce 161
7.2 Diskusia 163
Záver 167
Príloha 171
Publikované práce 173
Zoznam obrázkov 177
Zoznam tabuliek 179
Zoznam použitej literatúry 181
Zoznam skratiek 193

1 Úvod do problematiky

Pojem databáza je v dnešnej dobre veľmi rozšírený a známy. Už v minulosti mali ľudia tendenciu ukladať nahromadené dáta do určitých štruktúr. Medzi prvé dáta, ktoré bolo nutné ukladať, patrili údaje o obyvateľstve, zozname kníh alebo bankách. S nárastom obyvateľstva sa veľkosť spomenutých záznamov postupne zväčšovala, čo malo za následok klesanie efektívnosti tradičných typov databáz. S masovým nárastom množstva dát sa stretávame s javom, kedy začínajú byť dáta neštruktúrované. S veľkým objemom údajov, ktoré prichádzajú do systému súvisí aj rozmanitá heterogénnosť údajov. Tento aspekt spôsobil, že pevne daná štruktúra, respektíve schéma, ktorá je udržiavaná v relačnej databáze nedokáže pokryť tieto neustále zmeny. Spomenutý jav má za následok, že tradičný manažment relačných dát pomocou relačných schém a referencií prestáva byť použiteľný. Tieto nedostatky viedli k vzniku odlišných typov databázových systémov.

Potreby ukladania a spracovania údajov v moderných internetových službách, ako sú sociálne siete, online nakupovanie, analýza údajov a vizualizácia, si vyžadovali nový druh úložných systémov, nazývaných databázy NoSql (nielen Sql). Takéto databázy používajú nové techniky, ktoré podporujú paralelné spracovanie a replikáciu údajov vo viacerých uzloch, aby sa zabezpečil zlepšený výkon a dostupnosť údajov [39].

Väčšina údajov, ktoré boli v minulosti vytvorené, boli uložené a v mnohých prípadoch sú stále ukladané do relačných databáz, akými sú Oracle, MySql, Microsoft Sql prípadne PostgreSql. S nárastom popularity NoSql databáz sa viaceré spoločnosti domnievali, že vlastnosti nerelačných databáz oproti vlastnostiam relačných databáz môžu pozitívnym spôsobom ovplyvniť prácu s údajmi. Za týmto účelom vytvorili niekoľko experimentov, ktorých cieľom bol presun dát z relačných databáz na nerelačné databázy. Hlavnými cieľmi, ktoré sme sa pri experimentoch snažili dosiahnuť bolo zachovanie referenčných integrít, dátových typov a jasne definovaných vzťahov medzi tabuľkami respektíve kolekciami. Mnohé zo štúdií, ktoré sa venovali presunu nahromadených dát v relačných databázach, vytvorili metodiky a všeobecné metódy, ktoré majú za cieľ presunúť dáta z jedného typu databáz na iný typ v snahe zachovania dátových typov a iných obmedzení [61] [38].

Počas transformačného procesu je nutné zachovať nielen dostupnosť dát, ale taktiež stav, kedy môžu byť údaje ovplyvnené dátami, ktoré vstupujú do spusteného procesu. Medzi spomenuté operácie patria príkazy *Update*, ktoré menia hodnotu daného záznamu prípadne príkaz *Delete*, ktorý vymazáva záznam, a tým pádom, nie je potrebné danú hodnotu upravovať a následne ukladať do cieľovej databázy. V prípade ignorovania alebo nevytvorenia mechanizmu na úpravu dát v spustenom transformačnom procese by bolo nutné dáta po presunutí zo zdrojovej databázy na cieľovú znovu prehľadať a tento fakt by mal za následok nárast času potrebného na presun údajov [115].

Navyše nerelačné databázy, ktoré majú spĺňať kritérium bezproblémovej obsluhy takzvanej 3V(*Velocity, Velocity, Variety*) musia spoľahlivo a efektívne vyhľadávať údaje aj s narastajúcim množstvom dát v konkrétnej nerelačnej databáze. Čím ďalej tým viac narastajú požiadavky nie len na databázu, ale aj na iné odvetvia informačných technológií na rýchlejšie získavanie dát a zrýchlenie procesov. Práve zrýchlenie časového hľadiska a redukcia náročnosti na databázu sú pre nás kľúčovým faktorom. Ako ukážeme v ďalšej časti práce, súčasné riešenia neposkytujú dostatočne rýchlu odozvu na získanie objektov.

Predkladanú prácu je možné rozdeliť na 3 logické celky. V prvej časti sa zameriavame na historický vývoj nerelačných databáz a porovnávame jednotlivé transformačné prístupy, či sa už jedná o transformáciu relačnej databázy Oracle na nerelačnú databázu DynamoDB alebo ide o spätnú transformáciu. Druhá časť práce sa zameriava na navrhnuté a implementované riešenie - mapovacie pravidlá, verzionovací systém, riešenie paralelizmu v snahe optimalizovať náklady, konsolidačný mechanizmus na úpravu pevnej štruktúry a v neposlednom rade mechanizmus na urýchlenie vyhľadávania v nerelačnej databáze. V tretej časti vyhodnocujeme kvalitu, robustnosť a spoľahlivosť implementovaných riešení na základe vykonaných experimentov.

Tradične na záver chceme uviesť, že v texte používame výraz „záznam“, „dáta“ a „údaje“, ktoré považujeme za synonymum a majú ten istý význam.

2 Požiadavky na spracovanie a vyhľadávanie v neštruktúrovaných databázach a stanovenie cieľov

Požiadavky kladené na systém, ktorého hlavnou úlohou je spracovanie a vyhľadávanie v neštruktúrovaných databázach sú zamerané na jednoduchú manipuláciu s dátami. Podľa nášho názoru mnohé zo systémov nútia používateľa priamo vstupovať do mechanizmu, a tým bránia mechanickej úprave dát. Na základe odhalených nedostatkov môžeme jednotlivé požiadavky rozdeliť do 5 skupín:

1. ***Aspekt pevnej dátovej štruktúry,*** ktorý sa práve zameriava na jednoduchosť použitia konsolidačného mechanizmu. Cieľom mechanizmu je poskytnúť riešenie, ktoré v stanovenom čase alebo podľa naplánovanej granularity vykoná konsolidačné riešenie, ktoré udrží cieľovú a zdrojovú databázu transformačne kompatibilnú za použitia definovaných pravidiel. Tento aspekt je zameraný na riešenie problémov bez používateľského prístupu.
2. ***Aspekt vyhľadávania,*** je založený na zrýchlení prístupu k aktuálnym dátam. Časový aspekt pri vyhľadávaní dát patrí k požiadavkám, na ktoré je kladený veľký dôraz.
3. ***Aspekt bezpečnosti,*** je založený na zaistení kontroly transformovaných dát zo zdrojovej dátovej štruktúry na cieľovú dátovú štruktúru s dôrazom na identifikáciu už spracovaných údajov a ich vynechaní z ďalších krokov transformácie. Našou snahou bolo vytvorenie verzionovacieho mechanizmu, ktorý dokáže manipulovať so stavmi príslušného objektu, a ktorý môžeme v rámci potreby porovnávať na zistenie zhody .
4. ***Aspekt paralelizmu,*** slúži v prípade narastajúceho dopytu na databázu. Účelom je zabezpečiť, aby používateľ nikdy nespozoroval zaťaženie systému, či už ide o splnenie požiadaviek pre jedného používateľa alebo pre milión používateľov.
5. ***Aspekt škálovateľnosti***, je založený na automatickom prispôsobovaní sa štruktúry na základe zvyšujúceho sa dopytu pri využívaní operácií počas transformácie. V prípade transformácie malého množstva údajov (MB) alebo až po veľké množstvo údajov (PB, EX) nám systém musí poskytnúť dostatočné výpočtové jednotky.

Na základe požiadaviek a na základe zistenia určitých nedostatkov, sme stanovili nasledujúce ciele:

- Navrhneme algoritmus, ktorý umožní získanie dát viacerými používateľmi aj v čase, keď nad dátami budú vykonávané niektoré zo základných databázových operácií ako je vloženie (*insert*), aktualizovanie (*update*) a mazanie (*delete*) údajov.
- Navrhneme spôsob ako optimalizovať náklady spojené s komunikáciou medzi klientami a servismi na základe priemerného vyťaženia zavedením horizontálnej škálovateľnosti, pomocou využitia technológie kubernetes pre optimalizovanie nízkej ceny.
- Navrhneme a implementujeme architektúru systému, ktorá bude pracovať spoľahlivo s veľkým množstvom dát počas celého fungovania aplikácie. Navrhnutá aplikácia sa bude skladať z niekoľkých servisov, ktoré v prípade výpadku budú automaticky spustené a proces bude ďalej pokračovať. Tento prístup preferujeme pred monolitom, ktorý v prípade poruchy prestane pracovať. Z programu, zloženého zo servisov v prípade zlyhania určitej časti prestane pracovať iba časť a program sa môže ďalej využívať.
- Navrhneme, implementujeme a overíme metodiku vyhľadávania dát v nerelačných databázach s dôrazom na časový aspekt.
- Navrhneme a implementujeme metodiku zameriavajúcu sa na zvýšenie bezpečnosti transformačného procesu.

3 Ciele práce

V súčasnosti rapídne narastá objem heterogénnych dát z distribuovaných zdrojov, čím vznikajú nové výzvy pri extrahovaní údajov. Takými aplikáciami môže byť modelovanie, simulácia, rozpoznávanie obrazov, vizualizácia a podobne v rôznych oblastiach, ako sú napr. biomedicína, astrofyzika, životné prostredie, aeronautika, automobilový priemysel, energetika prípadne materiálové vedy. Vzhľadom na veľkosť dát, ktoré sú často označované ako veľké, resp. extrémne, je potrebné navrhnúť metodológiu, robustné metódy a nástroje pre extrémne škálovateľnú analytiku v súčinnosti s distribuovanými architektúrami pre zber a manažovanie obrovského množstva dát, akými sú cloud-ové technológie a IoT. Aj keď sa v danej problematike urobil veľký krok vpred, stále existuje niekoľko problémov, ktoré by dokázali zlepšiť distribuované spracovanie dát.

Pri spracovávaní veľkého množstva údajov existuje niekoľko výziev, ktoré sme na základe logických súvislostí rozdelili do 10 kategórií :

1. ***Zber údajov*** - Úplne prvou výzvou pri spracovaní údajov je zber alebo získanie správnych údajov pre vstup.
2. ***Duplicita údajov*** - Keďže sa údaje zhromažďujú z rôznych zdrojov údajov, často sa stáva, že dôjde k duplicite údajov. Rovnaké záznamy a entity sa môžu počas fázy kódovania údajov vyskytnúť niekoľkokrát. Tieto duplicitné údaje sú nadbytočné a môžu viesť k nesprávnym výsledkom.
3. ***Nekonzistencia údajov*** - Z dôvodu zhromažďovania obrovského množstva údajov, neexistuje záruka, že informácie budú úplné alebo že všetky polia, ktoré potrebujeme, sú vyplnené správne. Údaje môžu byť navyše nejednoznačné.
4. ***Rozmanitosť údajov*** - Vstupné údaje, ktoré sa zhromažďujú z rôznych zdrojov, môžu obsahovať rôzne formy. Riadky a stĺpce relačnej databázy neobmedzujú údaje. Údaje sa líšia od aplikácie k aplikácii a od zdroja po zdroj. Väčšina týchto údajov je neštruktúrovaná a nezmestia sa do tabuľky alebo do relačnej databázy.
5. ***Integrácia údajov*** - Dátová integrácia znamená kombinovať údaje z rôznych zdrojov a prezentovať ich v jednotnom zobrazení. S rastúcou rozmanitosťou údajov a rôznymi formátmi údajov sa výzva integrovať údaje stáva čoraz väčšou.

6. ***Objem a ukladanie údajov*** - Pri spracovávaní veľkých dát je objem dát značný. Veľké dáta pozostávajú zo štruktúrovaných aj neštruktúrovaných údajov. Patria sem údaje dostupné na stránkach sociálnych sietí, záznamy spoločností, údaje zo zdrojov sledovania, údaje o výskume a vývoji, a oveľa viac. Prichádza výzva ukladať a spravovať tento obrovský objem dát. Ďalšou výzvou je, aké množstvo dát má byť obsiahnuté v RAM, aby bolo spracovanie rýchlejšie a využitie zdrojov inteligentné.
7. ***Zlý popis a meta dáta*** - Jedným z hlavných zdrojov vstupných údajov sú údaje, ktoré sa časom ukladajú do relačnej databázy. Problémom týchto dát je nesprávne naformátované a neexistuje meta popis úložiska, štruktúry a vzájomného vzťahu dátových entít. Tento scenár sa ešte zhoršuje, ak je objem dát veľký a samotná databáza je prepojená s inými databázami. Bez náležitej dokumentácie databázy je dosť ťažké vyťažiť správne vstupné údaje z databáz.
8. ***Úprava sieťových údajov*** - Dáta sú distribuované a navzájom spolu súvisia v zložitej štruktúre. Výzvou je upraviť štruktúru údajov alebo k nim pridať nejaké údaje. Internet je sieť, ktorá pozostáva z rôznych údajov, množstva aplikácií a webových stránok generujúce údaje, ktoré majú rôzne formy a vlastnosti. Schéma ich všetky prepája.
9. ***Bezpečnosť*** - V dátovom poli zohráva najdôležitejšiu úlohu bezpečnosť. Hackovanie údajov môže mať za následok únik údajov. Pre spoločnosť zaoberajúcu sa spracovaním údajov môže predstavovať zvýšené náklady. Hacker môže pri veľkom úsilí dokonca zmeniť alebo vymazať údaje, ktoré sme získali a spracovali.
10. ***Náklady*** - Náklady sú vecou úvahy. Keď sa množstvo údajov zvýši, potom sa náklady v každej fáze spracovania údajov zvyšujú postupne.

Ako ďalšiu súčasť práce tvorí vyhľadávanie v nerelačných databázach. Dopyt po dátach je v dnešnej dobe enormný. Na rozdiel od relačných databáz, je vyhľadávanie v nerelačných databázach ovplyvnené heterogénnosťou údajov, s čím súvisí viacero komplikácií ako je vyhľadávanie podľa určitého atribútu alebo vytváranie indexov nad jedným prípadne skupinou atribútov.

Na základe spomenutých výziev a požiadaviek na zefektívnenie procesu vyhľadávania v nerelačných databázach, sme stanovili nasledujúce ciele:

- Navrhneme architektúru a algoritmus, ktorý dokáže spracovávať dáta v reálnom čase a ovplyvňovať práve sa transformujúce hodnoty.
- Navrhneme postupy na redukciu rizika straty dát na jednotlivých výpočtových jednotkách.
- Navrhneme a implementujeme metódu na redukciu duplicít záznamov a experimentálne ju overíme.
- Navrhneme metódu na automatické prispôsobovanie spracovania veľkého množstva dát, ktoré prichádzajú do systému.
- Vytvoríme modul na zachytávanie a mapovanie referenčných integrít dát medzi relačnými a nerelačnými databázami pri transformačných procesoch.
- Navrhneme a implementujeme rôzne varianty, ktoré zefektívnia proces vyhľadávania, ktoré nám pomôžu udržiavať štruktúru údajov a umožnia nám definovať indexy nad neúplnými dátami.

Nami navrhnuté kapitoly majú nasledujúcu štruktúru:

- V úvode predstavíme základné informácie o skúmanej problematike.
- Následná časť kapitoly sa zaoberá riešeniami, ktoré predstavili výskumníci, ktorí sa taktiež zaoberali skúmanou problematikou.
- Po predstavení metód od výskumníkov nasleduje kapitola, v ktorej predstavíme nedokonalosti metód, ktoré boli predstavené výskumníkmi a definujeme možnosti, ktoré by ich metódy zdokonalili.
- Nasleduje podkapitola, v ktorej predstavíme naše riešenie, respektíve metódy, ktorých hlavným cieľom je zdokonalenie metód a tým zefektívnenie skúmaného problému.
- Po implementovaní navrhnutej metódy sme vytvorili experimenty, ktorých cieľom je potvrdenie respektíve vyvrátenie efektívnosti nami vytvorenej metódy.
- Posledná časť kapitoly má za úlohu zhrnúť vytvorené riešenie a navrhnúť ďalšiu možnosť skúmania danej problematiky.

4 Historický vývoj a súčasný stav transformačných mechanizmov

Vývoj transformačných mechanizmov spojených so zmenou dát bol ovplyvnený dvomi aspektami. Prvý aspekt súvisel s nárastom populácie a klesaním efektivity dovtedy využívaného dátového úložiska. Druhý aspekt súvisel s nárastom nových typov nerelačných databáz, a s tým súvisiacimi vhodnejšími dátovými štruktúrami.

4.1 Správa štruktúry dátovej schémy a presun údajov

Požiadavka na vytvorenie migračného nástroja, ktorý dokáže na základe definovaných pravidiel a pomocou algoritmu súvisieť s presunutím dáta z jedného typu databáz na iný typ, bola ovplyvnená 2 faktormi:

- prvý faktor súvisel s klesajúcou efektívnosťou konvenčných databáz, medzi ktoré zaraďujeme známe databázy Oracle, MySql, MsSql alebo PostgreSql,
- druhý faktor súvisel s narastajúcim počtom obyvateľstva, ktoré vyprodukuje veľké množstvo údajov, ktoré nemajú jasne definovanú štruktúru.

Vytvorením transformačného algoritmu sa zaoberali mnohí výskumníci už v minulosti, problematika tohto problému je však tak náročná a vývoj databáz napreduje tak rýchlo, že každý deň vznikajú nové a nové otázky, dohady a výzvy, ktoré je potrebné riešiť. Už pri vzniku prvých spôsobov transformačných algoritmov bolo jasné, že pri presune dát z jedného typu databáz na druhý typ bude potrebné riešiť aj spätnú kompatibilitu. Navyše problémy súvisiace so zmenou databázy sú ovplyvnené aj veľkou škálou nerelačných databáz. Ďalší problém, ktorý súvisí s modelovaním výstupných dát je ich výsledná typová štruktúra, ktorá z hľadiska spätnej kompatibility patrí ku kľúčovým problémom.

V súčasnosti je síce riešenie migračného procesu ľahšie realizovateľné a viacerí výskumníci zaoberajúci sa databázami a ich transformáciou už publikovali niekoľko článkov na túto tému, avšak stále viac otázok súvisí s nárastom nových dátových typov a štruktúrou dát prichádzajúcich do procesu. Komplikovanejšie úlohy [60], donútili nás výskumníkov vytvoriť komplexný prístup takým spôsobom, aby všetky už vyvinuté metódy boli schopné prispôsobiť sa vzniknutej situácii bez potreby úpravy zdrojových kódov a nastavení.

Situácia ohľadom voľnosti dát patrí k príkladu, kedy je potrebné vyriešiť prispôsobenie navrhnutého algoritmu na základe 3 typov dát:

- štruktúrovaných dát,
- neštruktúrovaných dát,
- semi-štruktúrovaných dát.

Štruktúrované dáta majú v mnohých ohľadoch pre nás výhodné vlastnosti pri operácii vyhľadávania (*select*), pretože dáta sú uložené v jasne definovaných štruktúrach, čo umožňuje vyhľadávaciemu algoritmu jednoduchšie prehľadávať a usporiadať dáta [96].

Neštruktúrované údaje sú údaje, ktoré nemôžu byť obsiahnuté v databáze formou riadkov a stĺpcov a nemajú priradený dátový model. Nedostatok štruktúry sťažil vyhľadávanie, správu a analýzu neštruktúrovaných údajov, čo je dôvod, prečo spoločnosti vo veľkej miere vyradili neštruktúrované údaje, až kým nedávne šírenie algoritmov umelej inteligencie a strojového učenia uľahčilo ich spracovanie [96] [54].

Okrem štruktúrovaných a neštruktúrovaných údajov existuje tretia kategória, ktorá je v podstate kombináciou oboch. Typ údajov definovaných ako polo-štruktúrované údaje má určité definičné alebo konzistentné charakteristiky, ale nezodpovedá štruktúre tak rigidnej, ako sa očakáva pri relačnej databáze. Z tohto dôvodu existujú určité organizačné vlastnosti, ako napríklad sémantické značky alebo metadáta [60] [96] [57], ktoré uľahčujú usporiadanie.

Navrhnúť optimálny a univerzálny modul, ktorý dokáže pracovať nielen s dátami, ale dokáže aj mapovať dáta na základe stanovených pravidiel nebolo možné na základe logických aspektov skĺbiť do jedného modulu. Z logického hľadiska sa viacerí výskumníci rozhodli tento problém rozdeliť na dve časti:

- modul migrácie údajov,
- modul mapovania údajov.

Vytvorené moduly umožnili čiastočne vytvoriť riešenie transformácie dát zo zdrojovej dátovej štruktúry na cieľovú štruktúru. Vytvorené moduly umožňovali aj spätnú kompatibilitu údajov, ale hlavný problém stále súvisí so situáciou, ak dochádza ku transformácii nerelačnej databázy na relačnú databázu.

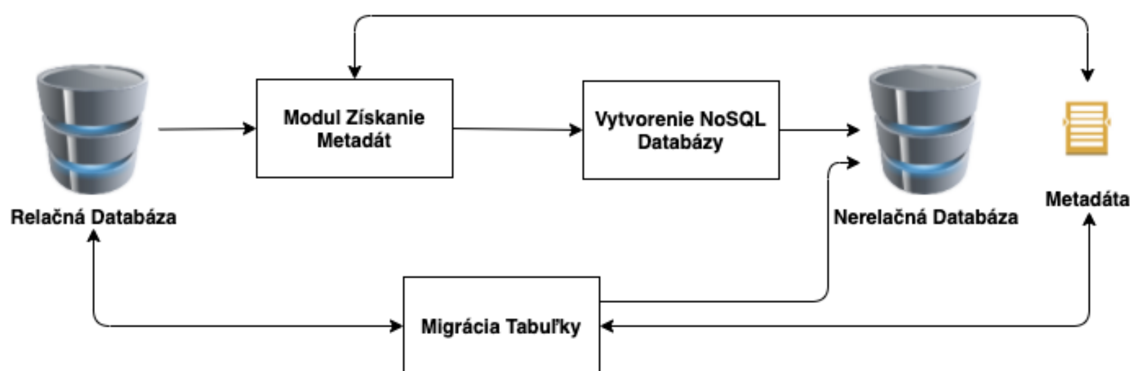
Hlavné problémy, ktoré vznikajú pri spomenutom procese je nedefinovaná štruktúra, referenčná integrita a dátové typy, ktoré nie sú v nerelačnej databáze kompatibilné s vlastnosťami a typmi v ľubovoľnej relačnej databáze.

Modul migrácie údajov sa začína analýzou relačnej databázy, aby sa zistilo, ktoré metadáta sa požadujú pri procese prevodu. Cieľom modulu je identifikovať všetky prvky, ktoré patria do databázy. Diagram fungovania migrácie údajov je znázornený na obr. 1.

Modul získavania metadát je tzv. vstupný modul pri migračnom procese. Hlavným účelom tohto modulu je získanie všetkých dostupných údajov o tabuľkách, veľkosti údajov, typov údajov, referenčných integritách a podobne. Po získaní a spracovaní potrebných údajov tento modul poskytne všetky informácie modulu na vytvorenie NoSql databázy. Všetky získané dáta sú uchovávané na centrálnom úložnom priestore, ku ktorému ma prístup len samotný modul. V prípade spustenia opätovného procesu sú hodnoty kontrolované priamo so súborom, ktorý sa načíta do pamäte a následne dochádza k rýchlejšej kontrole zmien.

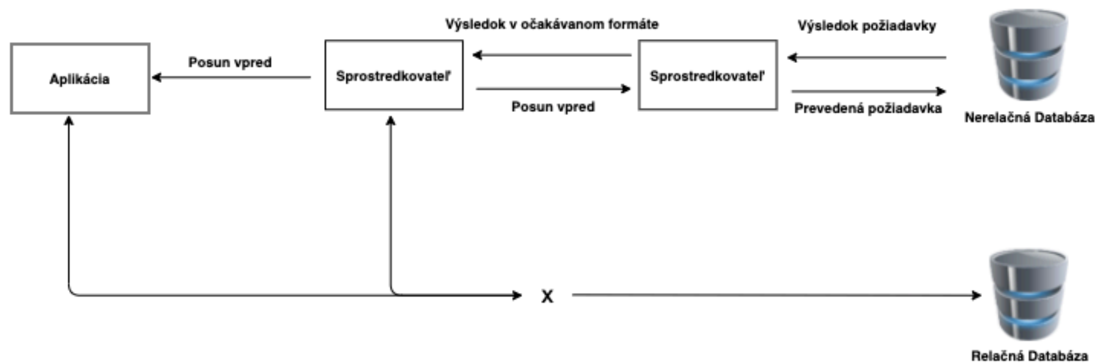
Modul na vytvorenie NoSql databázy je modul, ktorý na základe získaných informácií od predchádzajúceho modulu vytvára databázu, tabuľky a na základe pravidiel aj referenčné integrity. V prípade, ak nedokáže modul splniť príkaz na vytvorenie integrity je príkaz odignorovaný, prípadne sa snaží riadiaca jednotka splniť príkaz na základe iného preddefinovaného pravidla.

Akonáhle dôjde k vytvoreniu tabuliek, je spustený proces ich migrácie. V mnohých prípadoch dochádza k tomuto procesu až po upravení referenčných integrít pre jednotlivé tabuľky, avšak toto kritérium sa často ignoruje a referenčná integrita je kontrolovaná úplne na konci, keď došlo k migrácii všetkých tabuliek.



Obrázok 1. Diagram pre modul migrácie údajov

Modul mapovania údajov poskytuje abstraktnú vrstvu (t. j. vrstvu perzistencie) medzi aplikáciou a DBMS (t. j. MongoDB). Cieľom tohto modulu je umožniť bezproblémovú migráciu databázy, zabránenie akejkoľvek zmene v aplikačnom kóde pred zmenou použitého dátového modelu. Z tohto dôvodu vývojári aplikácií budú naďalej vytvárať dotazy v relačnom modeli, ale údaje budú poskytované z databázy NoSql a budú ťažiť z výhod výkonu a škálovateľnosti ponúkaných týmto systémom riadenia. Obr. 2 zobrazuje jednotlivé kroky tohto modulu.



Obrázok 2. Diagram pre modul mapovania údajov

5 Distribuované spracovanie dát

Veľký počet zariadení, veľké množstvo údajov, používateľov a aplikácií poháňajú digitálny svet dopredu rýchlejšie ako kedykoľvek predtým. Aby boli spoločnosti v dnešnej digitálnej ekonomike konkurencieschopné, musia spracovávať veľké objemy dynamicky sa meniacich údajov v reálnom čase. Existuje mnoho odvetví z oblasti zdravotníctva, elektronického obchodu, poisťovníctva a telekomunikácií s rôznymi prípadmi použitia, ako napríklad sekvenovanie DNA, získavanie poznatkov o zákazníkoch, ponuky v reálnom čase, vysokofrekvenčné obchodovanie a detekcia vniknutia v reálnom čase, ktoré využili na potrebné účely distribuované spracovanie dát. Pri analýze veľkých dát s dôrazom na kritické rozhodnutia sa pridanie distribuovaného spracovania údajov prejavilo zvýšením efektivity a rýchlosti získania informácií pre podporu rozhodovania [81].

Na druhej strane sa internet vecí (IoT) stáva hlavným dôvodom získavania údajov a analýzy veľkých údajov [30]. Vďaka rýchlemu rastu internetu vecí (IoT) a prípadov ich použitia v rôznych oblastiach, ako sú Smart City, Mobile e-Health a Smart Grid, streamingové aplikácie prinášajú novú vlnu dátových revolúcií. Vo väčšine aplikácií internetu vecí poskytuje výsledná analýza systému spätnú väzbu na jeho vylepšenie [124]. V porovnaní s ostatnými doménami veľkých dát existuje medzi reakciami systému cyklus s nízkou latenciou, ktorý vyžaduje spracovanie udalostí v reálnom čase, aby sa dosiahla prijateľná odozva. Vo všetkých týchto oblastiach je jednou z najzákladnejších výziev preskúmať veľké objemy údajov a získať užitočné informácie pre budúce činnosti. Tento prieskum v reálnom čase sa musí vykonať najmä v obrovských mierkach.

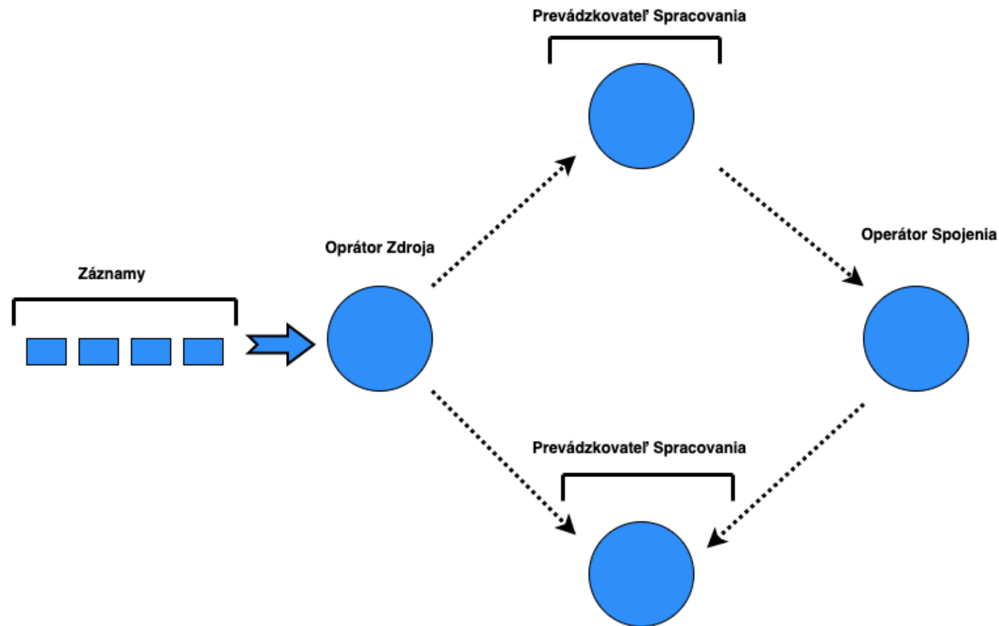
V súčasnosti generované údaje v dobe internetu vecí majú niekoľko charakteristík, ktoré ich zaraďujú do triedy veľkých dát [22]. V Smart City majú generované dáta zvyčajne tieto charakteristiky:

- Veľké objemy údajov generovaných v reálnom čase rôznymi aplikáciami v inteligentnom meste môžu byť rádovo v zetabajtoch.
- Heterogénne zdroje údajov v inteligentných mestách sú rôzne. Napríklad existuje veľa údajov o senzoroch, RFID (Rádiofrekvenčná identifikácia), kamerách, ľudsky generovaných údajoch atď.
- Heterogénne typy údajov zozbierané rôznymi zariadeniami sa líšia formátom, veľkosťou paketu, požadovanou presnosťou a časom príchodu.

Mnoho distribuovaných platforiem veľkých dát je navrhnutých tak, aby poskytovali škálovateľné spracovanie na komoditných zoskupeniach. Apache Hadoop [44] je jedným z najpopulárnejších rámcov pre dávkové spracovanie, ktorý používa programovací model MapReduce [26]. Existuje niekoľko rozsiahlych výpočtových architektúr prispôsobených na dávkové spracovanie [40], nie sú však vhodné na spracovanie prúdu, pretože v paradigme MapReduce musia byť všetky vstupné údaje pred začatím spracovania uložené v distribuovanom súborovom systéme (napríklad HDFS). Na riešenie rozsiahleho problému so spracovaním v reálnom čase sa objavili niektoré distribuované rámce, ako napríklad Apache Storm [1], Spark Streaming [116] a Apache Flink [55]. Spracúvajú nepretržitý tok správ o distribuovaných zdrojoch s nízkou latenciou a vysokou priepustnosťou.

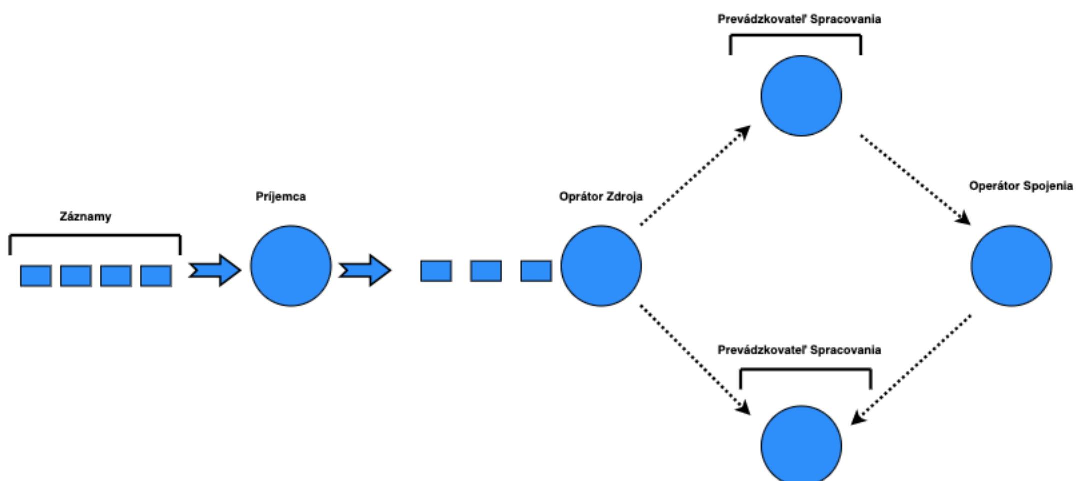
Najdôležitejším aspektom systému spracovania je pravdepodobne programovací model, pretože definuje budúce obmedzenia, náklady a dostupné operácie. Pri spracovaní toku je jednou zo základných častí programovacieho modelu to, ako sa každé nové údaje spracúvajú pri ich príchode. Toto rozlíšenie rozdeľuje spracovanie toku na dve kategórie: natívne a mikro-dávkové (*micro-batch*).

V natívnom streamingovom modeli sa všetky prichádzajúce n-tice spracúvajú hneď po ich prijatí. To znamená, že k spracovaniu údajov dochádza na základe spracovania jednotlivým dielom pred uložením údajov na pamäťové médium, namiesto toho aby sa k dátam pristupovala naraz. Obr. 3 zobrazuje dátový tok natívneho streaming-ového modelu a jednotlivé komponenty, ktoré sa podieľajú na spracovaní údajov.



Obrázok 3. Model dátového toku prichádzajúcich dát s komponentmi

Druhým prístupom k spracovaniu toku je mikro-dávkovanie. V tejto metóde sa zo vstupných zväzkov vytvárajú kolekcie n-tice (tuples), ktoré sa nazývajú krátke šarže, a prechádzajú systémom. Tieto krátke šarže sa vytvárajú podľa naplánovaného časového intervalu (napríklad každých 10 minút) alebo podľa jednej alebo viacerých spustených podmienok (napr. spracúvajú každú šaržu, ak má viac alebo je rovná 100 KB údajov). Dávkové spracovanie môže byť užitočné, ak nie je dôležité mať najaktuálnejšie údaje. Na obr. 4 je znázornený mechanizmus modelu mikro-dávkového (*micro-batch*) spracovania.



Obrázok 4. Model mikro-dávkového spracovania

Obe metódy majú svoje klady a zápory. Veľkou výhodou natívneho streamovania je jeho expresivita. Pretože dáta do systému prichádzajú v časovom poradí (Neumožňuje sa ich výmena alebo presun. Dáta sú zoradené na základe časového aspektu.) nie je potrebné vykonávať žiadnou dodatočnú abstrakciou, pretože dáta sú na sebe časovo závislé. Aj v porovnaní s mikro-dávkovými (*micro-batch*) systémami je dosiahnuteľný čas vykonávania v natívnom streamovaní vždy oveľa lepší, pretože n-tice sa spracúvajú okamžite po príchode. Na druhej strane, systémy natívneho streamovania majú zvyčajne nižšiu priepustnosť. Okrem toho je odolnosť proti poruchám a vyvažovanie záťaže drahšie v natívnom streamovaní, než v mikro-dávkových (*micro-batch*) systémoch [89] [45].

V mikrosystémoch dávkovania rozdelenie dátových tokov na mikro-dávky znižuje náklady systému [118]. Niektoré operácie, ako napríklad správa stavu, sa implementujú oveľa ťažšie, pretože systém sa musí zaoberať celou dávkou [83].

Medzi najznámejšie a najpoužívanejšie rámce spracovania distribuovaných tokov patria Apache *Storm*, Apache *Flink*, Apache *Spark*, Apache *Heron* a Apache *Samza*.

Apache *Storm* je najobľúbenejšia a široko využívaná otvorená distribuovaná počítačová platforma v reálnom čase, ktorú zaviedla spoločnosť Twitter [102]. Podobne ako Hadoop pracuje s dátami vo forme dávkového spracovania, Apache Storm vykonáva operácie spoľahlivo s neohraničenými tokmi údajov. Klaster *Storm* pozostáva z dvoch typov uzlov: hlavného a pracovného uzla. Hlavný uzol prevádzkuje démona s názvom *Nimbus*, ktorý je ústrednou súčasťou Apache *Storm*. *Nimbus* je zodpovedný za distribúciu kódov a prideľovanie úloh pracovným uzlom. Monitoruje tiež zdravie klastra počúvaním vytvorených prezenčných signálov pracovnými uzlami, a v prípade potreby priraďuje neúspešné úlohy [81]. Uzly pracovníka (*pracovného uzla*) vykonávajú skutočné vykonanie streamingovej aplikácie. Každý pracovný uzol prevádzkuje démona s názvom *Supervisor*, ktorý pracuje s *Nimbus*-om a podľa potreby spúšťa a zastavuje pracovné procesy [81]. Konfigurácia pracovných uzlov určuje, koľko slotov môžu poskytnúť pre klaster, takže každý pracovný uzol môže spustiť jeden alebo viac pracovných procesov v závislosti od počtu prevádzkových intervalov [1]. Keďže Apache *Storm* nedokáže riadiť svoj stav klastra, spolieha sa na tento účel na Apache *Zookeeper* [47]. *Zookeeper* zjednodušuje komunikáciu medzi *Nimbusom*-om a supervízormi pomocou potvrdzovania správ, stavu spracovania atď.

Apache *Flink* je open-source streamingová platforma, ktorá poskytuje schopnosť prevádzkovať prostredie na spracovanie údajov v reálnom čase spôsobom odolným voči poruchám na stupnici miliónov n-tíc za sekundu [46]. *Flink* je založený skôr na spracovaní natívnym prúdom než na spracovaní mikro-dávok (*micro-batch*). *Flink* spracováva používateľom definovaný funkčný kód cez systémový zásobník. *Flink* má architektúru master-slave, ktorá pozostáva z manažéra úloh a jedného alebo viacerých správcov úloh [55]. Úlohou manažéra úloh je koordinácia všetkých výpočtov v systéme *Flink*, zatiaľ čo manažéri úloh sa používajú ako pracovníci a vykonávajú časti paralelných programov. *Flink* je známy svojou schopnosťou veľmi efektívne vypočítať bežné operácie, ako je hashovanie.

Apache *Spark* je široko používaný, vysoko flexibilný motor na dávkové spracovanie a spracovanie údajov prúdu, ktorý je dobre vyvinutý pre škálovateľný výkon pri veľkých objemoch operácií [117]. Aby sa maximalizoval výkon aplikácií na analýzu veľkých dát, *Spark* podporuje spracovanie v pamäti, ale môže tiež vykonávať spracovanie na disku, keď sú súbory údajov príliš veľké na to, aby sa zmestili do dostupnej pamäte. Architektúra Apache *Spark* je založená na týchto komponentoch [139]:

- Ovládač spark (*spark driver*).
- Správca klastrov.
- Vykonávateľ (*executor*).

Aplikácia *Spark* preberá údaje zo súboru zdrojov (*HDFS*, NoSql a relačných databáz atď.). Potom na údaje aplikuje sadu transformácií a nakoniec vykoná akciu, ktorá generuje zmysluplné výsledky. Ovládač iskier (*spark driver*), ktorý je hlavným uzlom v klastri *Spark*, prevádza aplikáciu na množinu úloh, ktoré má vykonať skupina vykonávateľov (*executors).* Akonáhle program ovládač spark *(spark driver*) prevedie aplikáciu na množinu úloh, odovzdá ich správcovi klastrov na distribúciu. Účelom správcu klastrov je pochopiť, kde sa požadované údaje nachádzajú, a distribuovať úlohy na najvhodnejší server v klastri. Každý server v klastri má spustiteľný program, ktorý prijíma úlohy od správcu klastrov, vykonáva ich, a potom vracia výsledky späť správcovi klastrov. Následne je na zodpovednosti manažéra klastra skombinovať výsledky od všetkých exekútorov a odpovedať na ovládač spark (*spark driver).*

Twitter s otvorenými zdrojmi *Heron* je znovu predstavený *Storm* s dôrazom na vyššiu škálovateľnosť a lepšie ladiace schopnosti [62]. Ciele vývoja *Heron* sú spracovanie petabajtov údajov, zvýšenie produktivity vývojárov, zjednodušenie ladenia a poskytnutie lepšej účinnosti [62]. *Heron* pozostáva z troch kľúčových komponentov:

- *Majster topológie*: zodpovedá za správu štruktúry od jej návrhu až po jej vytvorenie.
- *Stream Manager*: jeho úlohou je riadiť smerovanie n-tíc medzi komponentmi topológie.
- *Heron Instance*: je to proces, ktorý vykonáva jednu úlohu a umožňuje ľahké ladenie a profilovanie.

Apache *Samza* je tvorený kombináciou Apache *Kafka* [37] a *YARN* [104] na vykonávanie výpočtu cez dátové toky. Medzi hlavné ciele Apache *Samza* patrí lepšia odolnosť voči chybám, izolácia procesora, bezpečnosť a správa prostriedkov [59]. Vstupné toky n-tíc sa rozložia a rozdelia tak, aby sa vytvoril graf toku údajov. Každý graf obsahuje viacero tokov a úloh, ktoré umožňujú užívateľovi rozdeliť toky a paralelizovať vykonávanie operátorov na klastri strojov [53].

5.1 Aktuálny stav distribuovaného spracovania dát

Distribuované spracovanie údajov je metóda, pri ktorej viac počítačov distribuovaných na rôznych miestach spracúva dáta cez komunikačnú sieť tvorenú oddelenými pc. Tento spôsob je diametrálne odlišný od spôsobu centralizovaného servera, ktorý riadi a poskytuje možnosti spracovania všetkým pripojeným systémom z jedného centrálneho servera.

Počítače, ktoré tvoria distribuovanú sieť na spracovanie údajov, sú umiestnené na rôznych miestach, ale sú vzájomne prepojené prostredníctvom bezdrôtových alebo satelitných spojení.

Masívny nárast údajov v posledných rokoch vedie k rastúcemu dopytu po spracovaní veľkých údajov v moderných dátových centrách, ktoré sú zvyčajne distribuované v rôznych geografických regiónoch, napríklad v 13 dátových centrách spoločnosti Google v 8 krajinách na 4 kontinentoch [52]. Analýza veľkých údajov preukázala svoj veľký potenciál pri odhaľovaní cenných poznatkov o údajoch s cieľom zlepšiť rozhodovanie, minimalizovať riziká a vyvíjať nové produkty a služby.

Na druhej strane sa veľké údaje už premietli do vysokých nákladoch kvôli vysokému dopytu po výpočtových a komunikačných zdrojoch [123]. Výskumníci predpokladajú, že náklady na hardvér dátových centier budú neustále rásť, s čím súvisia aj ich vysoké výdavky na zvyšovanie efektivity a dopyt od zákazníka. Zo spomenutého dôvodu je nevyhnutné analyzovať problém minimalizácie nákladov na spracovanie veľkých údajov, ktoré sú distribuované na viacerých inštanciách dátových centier.

Na optimalizáciu cien sa pri optimálnych výpočtových jednotkách vyvinuli viaceré metódy. Ako hlavný nedostatok sme spozorovali zanedbanie návrhu veľkosti dátového centra na zníženie výpočtových nákladov úpravou počtu aktivovaných serverov prostredníctvom umiestnenia úloh [110]. Na základe zmeny veľkosti dátového centra sa navrhla štúdia zameriavajúca sa na geografické rozmiestnenie dátových centier [121] [68]. Problém s distribuovane spracovanými dátami súvisí aj s efektívne navrhnutou komunikáciou. Zaznamenali sme štúdie, ktoré sa venujú spomenutému problému v snahe zlepšenia údajov umiestnením úloh na server, na ktorých sa umiestnia vstupné údaje, aby sa zabránilo vzdialenému načítaniu záznamov [43].

Aj keď vyššie spomenuté štúdie priniesli určité pozitíva pri distribuovane spracovaných procesoch, diametrálne sa odlišujú od systému s nákladovo efektívnym spracovaním údajov a to z nasledujúcich dôvodov:

- *Plytvanie zdrojmi* - pre dáta, ktoré sa v systéme vyskytujú je nutné zabezpečiť efektívnu správu. Niektoré dáta sa v systéme vyskytujú častejšie, k niektorým údajom sa pristupuje menej často, a preto nie je potrebné ich mať neustále k dispozícii. Tento fakt nám umožňuje efektívne manipulovať s dátami, ktoré nie sú často dopytované, a tým sa budú redukovať náklady, prípadne redukovať množstvo serverov potrebných na spracovanie.
- *Spojenie v sieťach* - rýchlosť dátových centier je ovplyvnená viacerými faktormi, s ktorými súvisí aj ich nákladová vlastnosť [77]. Existujúca stratégia smerovania medzi dátovými centrami však nevyužíva rozmanitosť prepojení sietí dátových centier. Z dôvodu obmedzení kapacity ukladania a výpočtov nie je možné všetky úlohy ukladať na ten istý server, na ktorom sú uložené príslušné údaje. Je nevyhnutné, aby sa určité údaje stiahli zo vzdialeného servera.

- *Neefektivita údajov* - v systéme sa vyskytujú údaje, ktorých replikovateľnosť môže závisieť aj od ich vhodného využívania. Často používané dáta vyžadujú väčšiu replikáciu z dôvodu ich kľúčovej úlohy ako dáta, ku ktorým sa počas existencie údajov často nepristupuje. Tým pádom by dochádzalo k redukcii zaťaženia jednotlivých serverov, a taktiež by sa redukovalo množstvo spotrebovaného úložného priestoru.

Aby sme prekonali nedostatky existujúcich riešení, zaoberáme sa problémom nákladov na spracovanie veľkých dát, a tiež efektívnu replikáciu údajov medzi viacerými servermi. Pri našej optimalizácii nákladov berieme do úvahy obmedzené množstvo výpočtových prostriedkov pri optimálnom množstve poskytnutých serverov. Naším cieľom je optimalizovať ukladanie veľkého množstva dát, eliminovať množstvo potrebných replikácií. Naše hlavné príspevky sú zhrnuté takto:

- Optimalizovať množstvo paralelných procesov, ktoré sú využívané pri spracovaní veľkého množstva údajov, a tým zamedziť nadmernému vyťaženiu serverov.
- Optimalizovanie množstva replikovaných údajov na základe váhy jednotlivých údajov, a tým optimalizovať množstvo dátových jednotiek potrebných na správu údajov.

Preštudované distribuované spracovanie nám prinieslo množstvo poznatkov a určilo smer akým ďalej postupovať. Navrhnuté metódy fungujú efektívne, spoľahlivo a rýchlo. V mnohých smeroch však neodzrkadľujú ďalšie dodatočné fakty akými sú náklady, automatické prispôsobovanie, manažment vlákien a podobne, ktoré sú v súčasnosti požadované. Nami vytvorené metódy už zohľadňujú požadované vlastnosti.

5.2 Práce zameriavajúce sa na riešenie problémov s distribuovaným spracovaním dát

Pri študovaní problému sme sa zamerali na odhaľovanie nedostatkov systému, ktoré sme pri skúmaní problematiky distribuovaného spracovania dát odhalili a zhrnuli sme ich do 3 kľúčových sekcií:

5.2.1 Optimalizácia paralelizmu

Distribuovane spracovávané dáta je v súčasnosti možné využiť mnohými nástrojmi ako napríklad *Hadoop*, *Spark*, *Flink*, *Samza* a mnohými inými. V týchto systémoch sa rozdelenie údajov používa na riadenie paralelizmu a je pre tieto systémy ústredné dosiahnuť škálovateľnosť veľkých výpočtových klastrov. Avšak, techniky rozdelenia používané systémami sú veľmi primitívne, čo vedie k vážnym problémom s výkonom.

V článku venujúcemu sa danej problematike výskumník Scheinder spolu s tímom [94] predstavili kompilátor a runtime systém, ktorý automaticky extrahuje paralelitu údajov pre spracovanie distribuovaných tokov. Podľa dosiahnutých výsledkov ich prístup pri zostavovaní paralelných oblastí zaisťuje kompilátor bezpečnosť tým, že v grafe zohľadňuje selektivitu operátora, stav, rozdelenie a závislosti od operátorov. Distribuovaný runtime systém zaisťuje, že n-tice vždy opúšťajú paralelné oblasti v rovnakom poradí, v akom by boli bez dátového paralelizmu, s použitím najúčinnejšej stratégie identifikovanej kompilátorom. Iný pohľad na danú problematiku má výskumník Dean spolu s Ghemawat [26], ktorí prišli s myšlienkou využívania implementácie MapReduce bežiacom na veľkom zoskupení komoditných strojov. Výpočet presunuli na niekoľko zariadení a pri výskume používajú klaster spoločnosti Google.

5.2.2 Optimalizácia ceny

Pri výpočte veľkého množstva dát sa donedávna na tento účel využívali rozsiahle dátové centrá. Dátové centrá v mnohých ohľadoch poskytujú niekoľko výpočtových účelov, s čím súvisí aj ich nákladovosť. Podľa [89] môže dátové centrum pozostávať z veľkého počtu serverov a spotrebovávať megawatty energie. Cena za náklady na elektrinu predstavuje pre prevádzkovateľov dátových centier veľké negatívum pri spracovaní veľkého množstva údajov. Z tohto dôvodu sa znižovaniu nákladov na elektrinu venovala značná pozornosť akademickej obce, ako aj priemyslu [35] [41]. Medzi mechanizmy, ktoré boli doteraz navrhnuté pre správu energie v dátových centrách, patria techniky, ktoré priťahujú veľa pozornosti. Medzi najznámejšie spôsoby zaraďujeme úlohy umiestňovania a DCR (DDL Command Replication).

DCR a umiestnenie úloh sa zvyčajne považujú za štandardy, aby zodpovedali na otázky k výpočtovým požiadavkám. Liu a kol. [74] ten istý problém opätovne preskúmali zohľadnením oneskorenia v sieti. Fan a kol. [35] skúmajú stratégie poskytovania energie,

koľko výpočtového zariadenia možno bezpečne a efektívne umiestniť v rámci daného rozpočtu na energiu. Rao a kol. skúmajú, ako znížiť náklady na elektrinu smerovaním požiadaviek používateľov do geograficky distribuovaných dátových centier s príslušne aktualizovanými veľkosťami, ktoré zodpovedajú požiadavkám. Veľmi odlišnú myšlienku na optimalizáciu cien poskytli výskumníci ako sú Sharon [114] spolu s kolektívom, ktorí sa zameriavajú a porovnávajú efektivitu prechodu z fyzického servera na cloud-ové služby. Spomenutá štúdia porovnáva centralizované dátové centrum, priestor virtuálnych serverov a bezpečný prenos údajov cez internet. Podobnej problematike sa venoval ďalší výskumník z vedeckej obce Qian spolu s kolektívom v práci [88], ktorí publikovali článok zameriavajúci sa na definovanie pojmu, histórie, výhod a nevýhod cloud computingu, ako aj na definovanie hodnotových reťazcov a úsilie o štandardizáciu.

5.2.3 Ochrana údajov

Ochrana údajov patrí v súčasnosti k veľmi dôležitej časti vývoja akéhokoľvek systému. Pri svojom výskume sme zaznamenali 4 kľúčové práce od výskumníkov Agarwal, Cidona, Shachnai a Jin spolu s ich kolektívmi, ktorí sa venujú danej problematike. Agarwal spolu s jeho výskumným tímom (A. P. Aakash, 2020) navrhli mechanizmus umiestnenia údajov *Volley* pre geograficky distribuované cloud-ové služby. Táto práca berie do úvahy náklady na šírku pásma WAN(rozsiahla sieť(wide area network)), vzájomné závislosti, obmedzené kapacity dátového centra a podobne. Volley analyzuje protokoly na základe takzvaného “interakčného optimalizačného algoritmu”. Algoritmus je založený na prístupových údajoch a umiestneniach klientov. Spomenutý článok taktiež poskytuje odporúčania migrácie späť do cloud-ovej služby. Druhá zaujímavá štúdia od výskumníkov Cidon spolu s kolektívom [25] súvisí s mechanizmom, ktorý pomenovali *MinCopysets*. Algoritmus je založený na efektívnom umiestňovaní replikácií údajov z dôvodu zefektívnenia vlastnosti trvanlivosti údajov v distribuovaných dátových centrách. Tretí zmienený výskumník spolu s kolektívom [120] nadviazali na algoritmus MinCopysets s tým, že sa zaoberali otázkou ako vytvorený mechanizmus pracuje s umiestnením kópií viacerých video súborov na serveroch. Po aplikovaní experimentov zistili, že s narastajúcou kapacitou načítania priradenej kópii je možné minimalizovať náklady spojené s ich správou pri zabezpečení vysoko dostupného koncového bodu pre používateľa. Nedávno publikovaná štúdia od Jin spolu s kolektívom [51] riešila návrh spoločnej optimalizačnej schémy,

ktorá nielenže optimalizuje umiestnenie virtuálneho stroja, ale poskytuje odporúčanie pri smerovaní toku siete z dôvodu úspory energie.

Zmienené práce zaoberajúce sa optimalizáciou nákladov pri distribuovanom spracovaní dát sa zameriavajú hlavne na 1, prípadne 2 aspekty. Aby sme sa mohli efektívne zaoberať distribuovaným spracovaním dát, tvrdíme, že je nevyhnutné systematicky posudzovať výkon aplikácie na základe vyťaženia vybraného nástroja pri dosiahnutí optimálneho výkonu s ohľadom na úsporu nákladov a optimálny počet výpočtových a úložných uzlov.

5.2.4 Dôvod skúmania danej problematiky

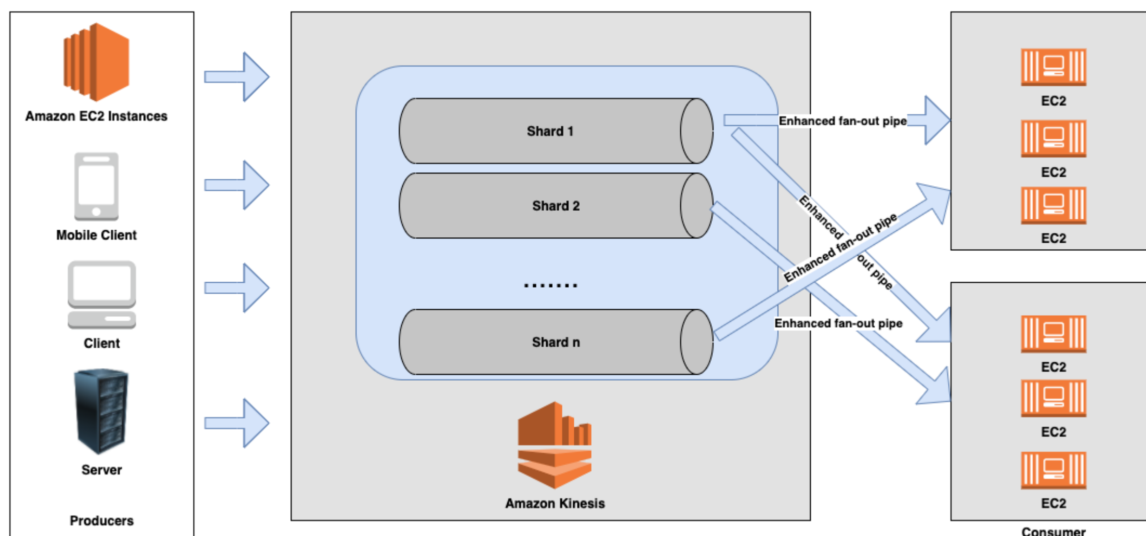
Veľkosť dát, ktoré prichádzajú do systému, nie je v súčasnosti ľahké odhadnúť. Pri veľkom počte prichádzajúcich dát, ktoré spracovávame distribuovane vstupujú do systému dáta vo veľmi malom objeme o veľkosti niekoľkých megabitov až po dáta o veľkosti rádovo gigabitov, terabitov, prípadne exabytov.

Na základe viacerých štúdií, ktoré sa zaoberali myšlienkou distribuovaného spracovania dát a paralelizmu, pri ktorej výskumníci zohľadňujú paralelizmus pri veľkom dôraze nákladov [35] a ochrane údajov [25], sa viacerí výskumníci obracajú na konštruovanie a správu paralelizmu na veľké cloud-ové spoločnosti ako je Google [29].

V mnohých prípadoch je spomenuté riešenie správne, avšak mechanizmus posudzovania paralelných procesov nemusí vyhovovať všetkým aplikáciám. Viaceré aplikácie nie sú závislé iba na počte dát, ale aj na vyťažení procesora, vyťažení vlákien vykonávajúcich výpočtové operácie a podobne.

5.3 Automatické škálovanie vlákien na základe dopytu

Pre účely distribuovaného spracovania dát sme využili Amazon Kinesis Stream v cloud-ovej službe Amazon. Amazon Kinesis Streams je trvalá a škálovateľná služba v reálnom čase. Môže zhromažďovať gigabajty údajov za sekundu zo stoviek tisícov zdrojov, vrátane prúdov databázových udalostí, tokov kliknutí na webe, finančných transakcií, protokolov IT, kanálov sociálnych médií a udalostí sledovania polohy. Zachytené údaje sa poskytujú v milisekundách pre prípady použitia analytických údajov v reálnom čase, vrátane detekcie anomálií v reálnom čase, dashboardov v reálnom čase a dynamického stanovovania cien.



Obrázok 5. Ukážka fungovania služby Amazon Kinesis

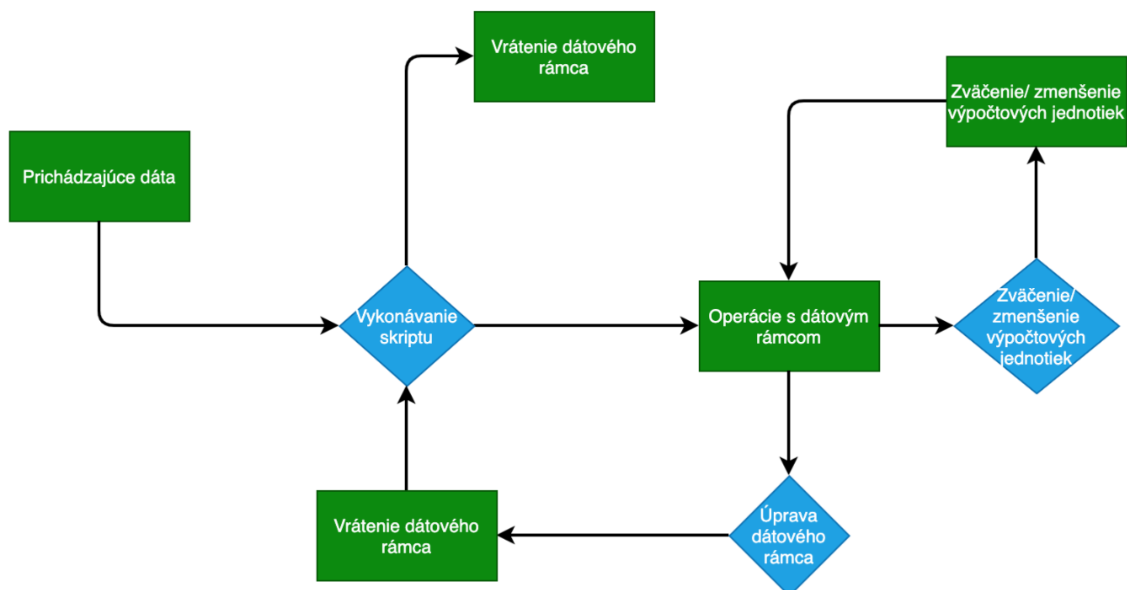
Hodnoty, ktoré do systému prichádzajú sú zobrazené na obr. 5 vľavo. Je vidieť, že záznamy prichádzajú z rôznych výpočtových jednotiek EC2, prípadne z ľubovoľnej mobilnej aplikácie, ktorá má sprístupnené API klienta pomocou služby Cognito. Samozrejmosťou je aj možnosť pripojenia sa k službe Kinesis pomocou rôznych počítačov a servera. Samotné spracovanie prebieha takto:

- Aplikácie, ktoré produkujú dáta, vkladajú záznamy (príjem údajov) do KDS (*Kinesis Data Streams*). AWS poskytuje Kinesis Producer Library (KPL) na zjednodušenie vývoja aplikácií producentov a na dosiahnutie vysokej priepustnosti zápisu do dátového toku Kinesis.
- Dátový prúd Kinesis je sada procesov. Každý proces obsahuje postupnosť dátových záznamov. Údajové záznamy sa skladajú z poradového čísla, kľúča oddielu a údajového blob-u (do 1 MB), čo je nemenná postupnosť bajtov.
- Spotrebitelia získavajú záznamy z dátových tokov Kinesis a spracúvajú ich. Svoje aplikácie sme zostavovali pomocou Kinesis Data Analytics, Kinesis API alebo Kinesis Client Library (KCL).

Rýchlosť spracovania údajov v procese zobrazenom na obr. 5 je závislá na efektívnosti operácií Kinesis. Z toho dôvodu sme poskytli službe Kinesis flexibilitu škálovania na základe vyťaženia.

Na účely spracovania údajov v službe Kinesis sme vytvorili skript, ktorý sme uložili na adrese https://github.com/romanceresnak/kinesis/blob/master/script.py. Spomínaný skript vykonáva nasledujúce kroky:

- Najskôr sa spustil časovač na zachytenie času vykonávania skriptu.
- Vytvorili sme klienta s Kinesis vo francúzskom regióne (eu-west-1, tento región sme vybrali z dôvodu blízkosti od našej aktuálnej polohy, pre rýchlejšiu manipuláciu dát a nízku latenciu).
- Kinesis načíta údaje z formátu CSV premenovaného na „data.csv“ a vráti dátový rámec.
- Následne dochádza k úprave údajového rámca tak, aby opravil pole datetime a následne pripojil nové pole datetime a aktuálny čas. Následne je vrátený údajový rámec.
- Polia každého záznamu sú spojené pomocou znaku „|“ (pajpa), ktorý neskôr využijeme. Dávkujeme všetky údaje a odošleme ich na spracovanie pomocou klienta, ktorého sme vytvorili, s uvedením názvu Stream a počtu zlomkov. Črepy (*shards*) sa prispôsobujú tak, aby sa umožnilo správne rozvetvenie pracovného zaťaženia, tu však používame iba 1. Celý postup je zobrazený na obr. 6.
- Následne sme dáta uložili do 2 regiónov.



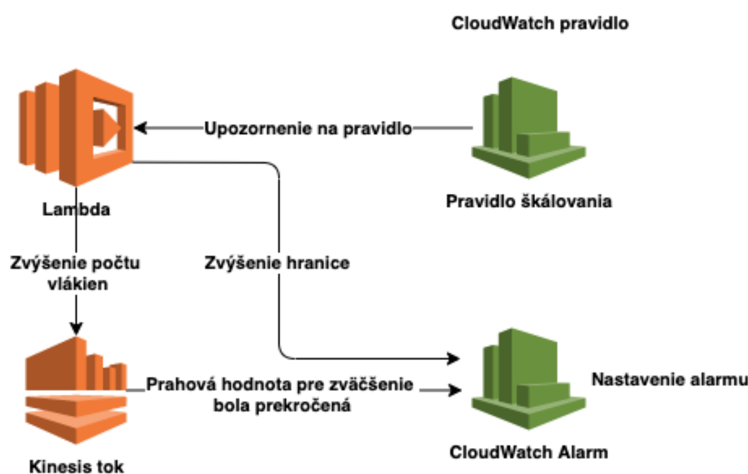
Obrázok 6. Dátový diagram prichádzajúcich dát

Dáta sme z dôvodu vyššej bezpečnosti replikovali do 2 regiónov, čo znamená, že replikačný koeficient je nastavený na hodnotu 2. V prípade výpadku regiónu je vyvolaná automatická replikácia dát, ktorej úlohou je zabezpečiť stály počet replikácií jednotlivých údajov. Princíp fungovania straty regiónu je nasledovný : automaticky algoritmus zistí počet kópií. Ak sa replikácia dát nerovná hodnote 2, dáta sú automaticky replikované do ďalšieho regiónu, v inom prípade sa kontrola replikácií údajov presúva na ďalšie časové obdobie. Pre spomenutý prípad sme nastavili automatickú správu IP adries pomocou služby *Elastic Load Balancer*, ktorá má za účel maskovanie straty regiónu v snahe zabrániť akémukoľvek povšimnutiu straty dát koncovým používateľom.

Všetky dáta, ktoré do procesu vstúpili sú po spracovaní automaticky presunuté do služby Amazon S3, ktorá na základe replikačného koeficientu riadi replikovanie dát a kontroluje stratu dát v regióne.

5.3.1 Škálovanie smerom nahor na základe prichádzajúcich údajov

Automatické škálovanie smerom nahor umožňuje prispôsobovať množstvo potrebných vlákien na základe zvyšovania dát respektíve zvýšenia vyťaženosti servera.



Obrázok 7. Architektúra pre zvýšenie škálovania

Na škálovanie množstva vlákien sme vytvorili architektúru, ktorá je zobrazená na obr. 7. Skladá sa zo 4 služieb, ktorými sú Kinesis Stream, Lambda funkcia, Simple Notified Service (SNS) a Amazon CloudWatch [1].

Alarm služby CloudWatch monitoruje metriky služby Kinesis Data Stream.

[1] Problematiku distribuovaného spracovania dát sme prezentovali a diskutovali na IEEE konferencii FRUCT v Rusku (Moskva) - 27. - 29. január 2021.

Keď sa dosiahne prah alarmu, napríklad z dôvodu nárastu žiadostí, prípadne počtu vlákien, tak niektorá zo spomenutých udalostí spustí alarm. Toto spustenie alarmu odošle upozornenie na automatického prispôsobovania aplikácií, ktorá reaguje na základe uvedených preferencií automatickým škálovaním smerom nahor.

Keď sa aktivuje politika zmeny mierky, automatické škálovanie aplikácií zavolá operáciu API. Volanie odovzdá nový počet úlomkov (*shards*) dátového toku Kinesis pre požadovanú kapacitu. Volanie tiež odovzdá názov zdroja v mierke, ktorý poskytuje služba Amazon API Gateway. Amazon API Gateway vyvoláva funkciu AWS Lambda. Na základe informácií zaslaných automatickým škálovaním, funkcia Lambda zvyšuje alebo znižuje počet zlomkov v kinesis dátovom prúde. Robí to pomocou operácie *UpdateShardCount* API Kinesis Data Stream.

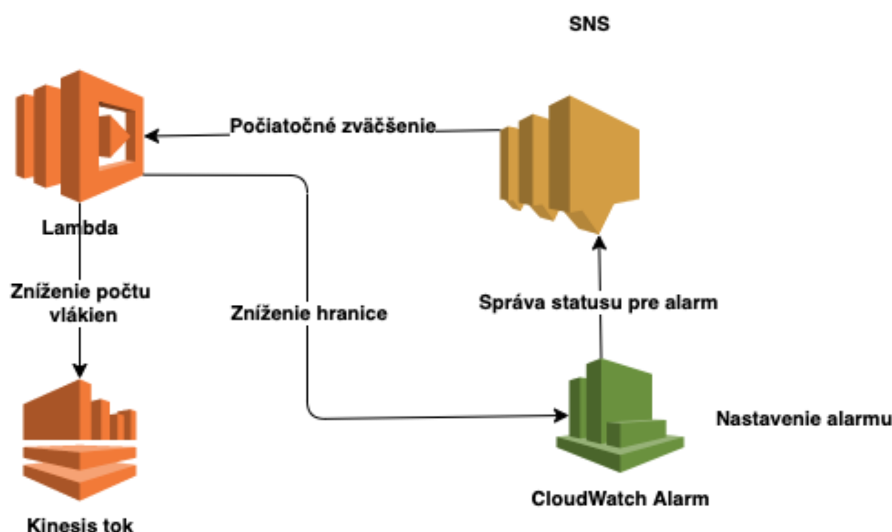
Aby bolo možné sledovať, kedy dôjde k zväčšeniu, bude Lambda hlásiť dve vlastné metriky (*OpenShards* a *ConcurrencyLimit*) do služby CloudWatch, kedykoľvek bude úspešne vyvolaná. Tieto vlastné metriky nám umožnia monitorovať správanie pri zmene mierky.

Ako už bolo spomenuté, Scale Up Lambda použije alarm na sledovanie kinezickej metriky, aby zistil, či neprekračuje vypočítanú hranicu.

Odporúčaným prístupom je meranie súčtu *IncomingRecords* alebo *IncomingBytes* z pridruženého prúdu Kinesis počas 5 minút. Získame tak priamy prehľad o tom, koľko dát prúdi do toku, a môžeme tak prijímať informované rozhodnutia o škálovaní.

5.3.2 Škálovanie smerom nadol na základe prichádzajúcich údajov

Lambda nám poskytuje funkciu, ktorá zmenšuje prúd Kinesis, umožňuje akceptovanie alarmu a dokáže upravovať definované nastavenia.



Obrázok 8. Architektúra pre zníženie škálovania

Raz za deň, v čase mimo špičku, (po spracovaní neúspešných protokolov) pravidlo *CloudWatch* spustí Scale Down Lambda v 10-minútových intervaloch. Tento postup, ktorý je zobrazený na obr. 8 sa vykonáva s cieľom vyrovnať sa s obmedzením, ktoré má *Kinesis* pri zmenšovaní (najnižší platný cieľový počet úlomkov je polovica súčasného počtu otvorených úlomkov).

Následne *Lambda* upraví proces zmenšenia, ak je sledovaná štatistika momentálne nad úrovňou vyťaženia, prípadne ak sa práve zmenšuje alebo ak už bol zmenšený na predvolený počet.

Rovnako ako škálovanie smerom nahor s použitím *Lambda* funkcie, aj táto Lambda bude hlásiť dve vlastné metriky (*OpenShards* a *ConcurrencyLimit*) do služby *CloudWatch*, kedykoľvek bude úspešne vyvolaná.

5.3.3 Výpočet prahovej hodnoty na základe vyťaženia servera

Na základe potreby aplikácie dochádza k automatickému posudzovaniu prahovej hodnoty.

Pre stream Kinesis s ***n*** črepinami (*shards*) bude Lambda škálovať najviac na ***n*** vyvolaní (riadené jeho vyhradenými súbežnými vykonaniami).

Každá Lambda pošle priemerne ***m*** záznamov do toku *Kinesis* za sekundu. Perióda, do ktorej alarm sleduje súčet metriky, je ***s*** sekúnd.

Prahová hodnota na monitorovanie je teda

$$n * m * s \quad (1)$$

Aby sme zaistili, že k zväčšeniu dôjde skôr, ako údaje zaostanú, môžeme namiesto toho monitorovať percento vypočítanej prahovej hodnoty. Pretože 80% považuje AWS za najlepší postup, budeme túto hodnotu namiesto toho monitorovať.

5.3.4 Medziregionálna replikácia

Hodnoty, ktoré sa spracovávajú, je nutné po úspešnej manipulácii efektívne uložiť. Veľké množstvo dát nie je možné ukladať na centrálnom počítači, a preto sa ukladajú distribuovane. Z dôvodu manipulácie s veľkým množstvom údajov, by prípadná strata údajov mohla viesť k zdĺhavému procesu ich obnovenia.

V službe Amazon sú dáta ukladané do Amazon S3, ktoré ma spoľahlivosť definovanú na 99,999999999%. Aj napriek tak vysokému číslu je potrebné zachovať spoľahlivosť a prístupnosť dát, a preto sme sa rozhodli replikovať dáta medzi jednotlivými regiónmi v sieti.

Týmto spôsobom sme definovali replikačný koeficient na 2, čo v prípade poruchy alebo zlyhania automaticky vyvolá operácie, ktoré kontrolujú množstvo replikovaných dát v systéme. Ak sa replikačný koeficient dát po zlyhaní regiónu alebo ľubovoľnej hodnoty nerovná 2 dáta sú automaticky replikované do iného regiónu.

Na základe dát replikovaných v 2 regiónoch sme splnili dôležitú podmienku uchovávania kópií kritických údajov na miestach vzdialených od seba stovky kilometrov. Taktiež naše riešenie spĺňa povinnosti udržiavania prísnych regulačných požiadaviek na uchovávanie citlivých finančných a osobných údajov.

5.3.5 Experimentálna činnosť

Na účely experimentov sme vytvorili 3 súbory s rôznymi veľkosťami súborov a rôznymi objemami súborov. Súbory sú nahraté na týchto adresách:

- https://github.com/romanceresnak/kinesis/blob/master/data2010-1000.csv
- https://github.com/romanceresnak/kinesis/blob/master/data2010-50000.csv
- https://github.com/romanceresnak/kinesis/blob/master/data2010-100000.csv

Vytvorené súbory obsahujú 1 000, 50 000 a 100 000 záznamov. Všetky operácie boli vykonávané na nasledujúcej konfigurácii:

Tabuľka 1. Konfigurácie servera

EC2 Instance	a1.medium vCPU: 1 MeM(GiB): 2
EMR cluster	master: 1x m3.xlarge core: 2x m4.4xlarge

Hodnoty, ktoré sme namerali pri spracovaní záznamov o veľkosti 1 000, 50 000 a 100 000 sú priemerné hodnoty pri uskutočnení 5 000 replikácií a ktoré sú nasledovné:

Pre 1 000:

Celkový počet záznamov odoslaných do Kinesis: 1 000
Celková dĺžka behu programu: 6.78234 [s]

Pre 50 000:

Celkový počet záznamov odoslaných do Kinesis: 50 000
Celková dĺžka behu programu: 8.23432 [s]

Pre 100 000:

Celkový počet záznamov odoslaných do Kinesis: 100 000
Celková dĺžka behu programu: 8.96464 [s]

Ako je vidieť z výsledkov pre počet záznamov 10 000, 50 000 a 100 000 tak hodnoty nám nedegradovali a exponenciálne nerástli so zvyšujúcim počtom záznamov. Tento fakt je práve ovplyvnený automatickým sa prispôsobovaním výpočtových jednotiek na základe zväčšujúceho sa počtu dát, s čím súvisel aj ich narastajúci dopyt po výpočtových jednotkách.

Obrázok 9. Automatický nárast a pokles vlákien

Ako je vidieť z obr. 9, hodnoty ktoré sme namerali pri veľkosti 1 000 záznamov ukazujú, že automatické škálovanie nebolo potrebné pre tieto účely, z dôvodu nízkej kapacity záznamov. Nami nastavená hodnota vyťaženia servera bola stanovená na hodnotu 80 percent, čo nebolo pri vstupnom toku dát prekročené, a preto nenastal žiadny nárast počtu výpočtových jednotiek. Okrem spomenutého faktu, hodnota 1 je vždy hodnota, ktorá predstavuje minimálny počet výpočtových jednotiek, ktorými server pri začiatku a konci procesu figuruje.

Pri porovnaní hodnôt, ktoré je možné pozorovať na obr. 9 nám s nárastom dát na hodnotu 50 000 záznamov narástol počet výpočtových jednotiek. Ako bolo spomínané vyššie v práci, tak vyťaženosť výpočtovej jednotky pri spracovaní 50 000 záznamov prekročila hranicu 80 percent, a tým spôsobila nárast počtu výpočtových jednotiek. Keďže ani počet výpočtových jednotiek rovnajúci sa hodnote 2 nestačil na zníženie vyťaženia servera, tak znovu bola pridaná ďalšia výpočtová jednotka. Pri počte rovnajúcej sa hodnote 3 sa vyťaženosť servera dostala do úrovne, kedy nebolo potrebné ani pridať a ani odobrať jednotky a server s optimálnym vyťažením dokázal spracovávať prichádzajúce dáta. Po spracovaní približne 2/3 záznamov došlo k opačnému efektu a množstvo výpočtových jednotiek potrebných na spracovanie údajov začal klesať až pokiaľ nebolo potrebné na spracovanie dát iba 1 jednotka. Tým pádom sa neoptimalizovala iba vyťaženosť servera, ale aj optimalizácia nákladov spojených s prevádzkovými nákladmi na spracovanie údajov.

Podobný efekt sme zaznamenali aj pri náraste dát na hodnotu 100 000 záznamov. Pri automatickom škálovaní sme sa s výpočtovými jednotkami dostali až na hranicu rovnú 4 po približne spracovaní polovice údajov. Následne sa vyťaženie servera znižovalo, s čím súvisí aj klesajúca hodnota výpočtových jednotiek až smerom na 1.

Ako si môžeme všimnúť, tak automatické prispôsobovanie výkonu funguje efektívne pri počte záznamov 1 000, tak aj 100 000, a tým nám pomáha zabezpečiť vždy dostatočný počet jednotiek na to, aby systém pracoval pri optimálnom vyťažení servera s dôrazom na efektívnu manipuláciu s prevádzkovými nákladmi. Radi by sme poznamenali, že dosiahnuté hodnoty sú primerné hodnotám pri vykonaní 1 000 simulácií.

Ako môžeme vidieť z tabuľky 2 **,** tak sledované hodnoty, ktoré sú pre nás podstatné a znamenajú pre nás pokrok oproti konvenčnej metóde sú zvýraznené zelenou farbou a negatívne aspekty našej metódy sú zvýraznené červenou farbou.

Efektivita nami zavedenej metódy môže byť ovplyvnená viacerými faktormi. Okrem samotného servera, počtu jadier a RAM je to taktiež dĺžka rady pre prichádzajúce záznamy. Je nutné zdôrazniť, že navrhnutá metóda bude mať odlišnú efektivitu, ak bude algoritmus naprogramovaný v jazyku Java, GO, Python alebo c#.

Po aplikovaní nami navrhnutej metódy dochádza k zvýšeniu režijných nákladov, čo sme od úplného začiatku predpokladali. Pri využití programovacieho jazyka Go (Golang) sme získali najrýchlejšie hodnoty, nakoľko jazyk Go dokáže pracovať na viacerých jadrách súčasne. Aj keď spomenutá vlastnosť jazyka Go je pre nás veľkým plusom, pri spracovaní môže dôjsť k situácii ovplyvňovania výpočtových jadier z čoho vyplynie časovo náročnejšia správa výpočtov ako pri konvenčnej metóde.

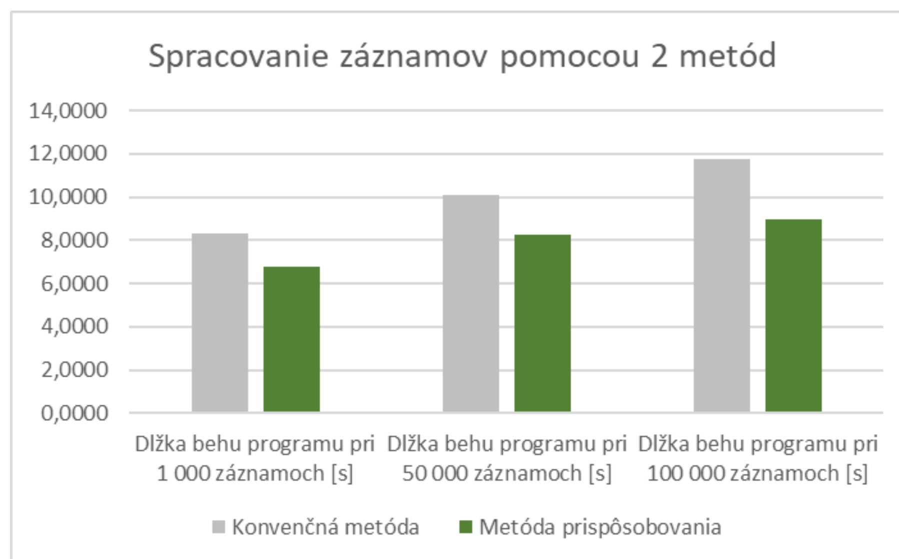
Tabuľka 2. Porovnanie sledovaných hodnôt medzi konvenčnou metódou a metódou správy záznamov

Číslo vlastnosti	Sledovaná vlastnosť	Konvenčná metóda	Metóda správy záznamov
1	Závislosť na RAM [Áno/Nie]	Nie	Áno
2	Správa na počet jadier [Áno/Nie]	Nie	Áno
3	Nutnosť fronty [Áno/Nie]	Nie	Áno
4	Správa zápisu [Áno/Nie]	Áno	Nie
5	Závislosť na jazyku [Áno/Nie]	Nie	Áno
6	Redukcia počtu vlákien [Áno/Nie]	Áno	Áno
7	Riadená redukcia počtu vlákien [Áno/Nie]	Nie	Áno
8	Správa údajov [s]	Nie	Áno
9	Dĺžka behu programu pri 1 000 záznamoch [s]	8,29837	6,78234
10	Dĺžka behu programu pri 50 000 záznamoch [s]	10,10978	8,23432
11	Dĺžka behu programu pri 100 000 záznamoch [s]	11,72834	8,96464

Ako je vidieť z grafu na obr. 10, tak grafické zobrazenie výsledkov ukazuje na od začiatku jasný trend. Aj pri malom množstve bol manažment a správa procesov dostatočne efektívna, aby spracovala záznamy a bola rýchlejšia ako konvenčná metóda.

Ako môžeme vidieť z grafu na obr. 10, kde nám zelená farba predstavuje hodnoty po aplikovaní našej metódy (hodnoty bližšie k nule sú lepšie hodnoty), tak získané výsledky pri implementovaní našej metódy boli vždy lepšie. Pri tomto tvrdení však musíme povedať veľké ale. Ak sme ten istý princíp aplikovali pri jazykoch java a python, tak výsledky boli vždy horšie oproti konvenčnej metóde. Musíme jednoznačne usúdiť, že kvalita viac jadrového jazyka nám umožnila využiť celý výpočtový rozsah nami použitého servera.

Aj keď výsledky hovoria v náš prospech, pri operáciách, ktoré si vyžadujú viacnásobné výpočty prebiehajúce na odlišných jadrách môže dôjsť k istému spomaleniu. Spomalenie súvisí si dynamickým vytváraním dodatočných vlákien, ktoré musí hlavné jadro spravovať.



Obrázok 10. Porovnanie spracovania záznamov medzi dvomi metódami

Na základe výsledkov, ktoré sme získali pri experimentálnej činnosti vidíme jasne preukázateľné zlepšenie po zavedení našej metódy. Pri experimentoch so serverom vidíme zhoršujúci efekt, ktorý sa týka zvyšovania režijných nákladov.

Počas tohto procesu museli vlákna zapisovať výsledky do distribuovaného systému. Aj keď sme pri experimentoch nemerali vyťaženie replikačného zápisu, všimli sme si, že distribuovaný systém obsahoval veľký počet replikácií aj pre dáta, ktoré boli podľa nášho názoru ľahko obnoviteľné a nemali veľkú podstatu v systéme.

Po vzniknutej situácii sme usúdili, že by stálo za zváženie vytvorenie prístupu, ktorý nám určitým spôsobom ešte pred vstupom do distribuovaného systému umožní odhadnúť istú využiteľnosť dát, a tým redukovať využitie diskového priestoru.

5.4 Spoľahlivosť systému

Na základe záveru v experimentoch v predchádzajúcej kapitole, kde sme usúdili, že replikačný koeficient nastavený na rovnakú hodnotu pre všetky údaje nemusí byť z pohľadu úložného miesta a spracovania dostatočne efektívny. V aktuálnej kapitole sme sa rozhodli riešiť spomenutý úsudok.

Replikácia dát sa často používa ako prostriedok na zvýšenie dostupnosti dát úložného systému, ako je napríklad súborový systém Google. Ak na rôznych dátových uzloch existuje viac kópií bloku, kanály najmenej jednej prístupnej kópie sa zvýšia. Ak jeden dátový uzol zlyhá, dáta sú stále k dispozícii z replík. Náklady na správu sa výrazne zvýšia s rastúcim počtom replík. Mnoho replík nemusí výrazne vylepšiť dostupnosť [19], ale namiesto toho prinesie zbytočné výdavky. Kľúčovou otázkou je, aké je ideálne množstvo replík, ktoré zabezpečí optimálne fungovanie systému pri výpadku výpočtových uzlov.

Keď ponecháme všetky repliky aktívne, môžu sa ďalšie kópie použiť nielen na zlepšenie dostupnosti, ale aj na zlepšenie vyváženia záťaže a celkového výkonu, ak sú repliky primerane distribuované. Pri stanovení umiestnenia repliky musíme brať do úvahy aj veľkosť systému a počet výpočtových uzlov, na ktorom sa budú vykonávať výpočtové operácie. Pravé takýmito otázkami sa zaoberalo viacero výskumníkov vo svojich štúdiách.

Cieľom kapitoly, ktorá sa zaoberá problematikou replikačného koeficientu je:

- Preskúmanie replikačného koeficientu nastaveného na hodnotu 3.
- Vytvorenie metodiky na posudzovanie vyťaženia údajov.
- Úprava replikačného koeficientu na základe štatistickej zmeny údajov.

Nakoľko zadefinovanie replikačného koeficientu výrazne ovplyvňuje nielen rýchlosť, ale aj cenu distribuovaného systému viacerí výskumníci sa zaoberali skúmanou problematikou. Nakoľko si myslíme, že existuje aj v tejto oblasti priestor na zlepšenie, v nasledujúcej podkapitole sa budeme zaoberať už existujúcimi riešeniami, aby sme spomenutý proces ešte viac zdokonalili.

5.4.1 Aktuálny stav riešenia spoľahlivosti systému

Súčasné inovácie ako Sybase Replication Server, Prophet Symmetric Replication Technology, Ingres Replicator a výmenné obrazovky ako DEC-ACMS poskytujú základné kapacity fyzickej replikácii informačných častí. Umožňujú odvíjanie koherencie na pevných úrovniach: starý štýl 2PC, primárny sekundárny prístup, idealistické využitie časových pečiatok alebo úplne nekontrolované. Tieto definované rámce neposkytujú nástroje, ktoré dodržiavajú podmienky výslovnej koherencie aplikácie.

Výskum hodnoty replikácie v distribuovanom systéme je založený na HDFS a zvýšenej spoľahlivosti pre spoľahlivé ukladanie veľmi veľkých súborov naprieč distribuovanými komoditnými strojmi vo veľkom klastri. Ukladá každý dokument ako usporiadanie štvorcov. Všetky štvorce v zázname majú podobnú veľkosť, okrem posledného. Druhé štvorce dokumentu sú duplikované kvôli pochopeniu vykonania a adaptácie na interné zlyhanie. HDFS predstavuje základnú, ale mimoriadne úspešnú stratégiu strojnásobenia priraďovania kópií k štvorcu. Predvolená stratégia polohy kopírovania HDFS je umiestniť jednu reprodukciu na jeden rozbočovač v susednom stojane. Hodnota druhej replikácie je umiestnená na vzdialenom stojane a tento princíp pokračuje na ďalšom stojane. Definovaná stratégia umiestňovania záznamu funguje medzi regálovými skladmi, čo celkovo zlepšuje vykonávanie skladania. Dôvodom metodiky usporiadania kópií zameraných na stojan (*rack*) je zlepšenie informačnej spoľahlivosti, dostupnosti a využitia prenosovej kapacity organizácie. Dohľad nad skupinami HDFS môže označiť predvolený faktor replikácie (počet reprodukcií) alebo faktor replikácie pre jednotlivé informácie. Môžu tiež vykonať vlastnú metodiku kopírovania situácie pre HDFS. Aj keď spomenutý spôsob manipulácie s dátami je spoľahlivý, z hľadiska bezpečnosti faktor replikácie a usporiadanie reprodukcie sú hlavnými otázkami replikácie dosky. Problematika dynamickej replikácie systému výkonného riaditeľa pre HDFS vyvolala značné úvahy.

Pri skúmaní spoľahlivosti systému sme zaznamenali prácu CDRM (Cross Domain Resource Manager) [108], ktorá sa zaoberá dynamickou replikáciou, pre rámec širokého rozsahu distribuovaného úložného priestoru. V metóde CDRM je vyvinutý nákladový model na zachytenie spojenia medzi prístupnosťou a faktorom replikácie. Vzhľadom na tento model je možné vyriešiť dolnú hranicu referenčného čísla kópie, aby sa splnila nevyhnutnosť dostupnosti.

CDRM ďalej umiestňuje kópie medzi cirkulačné rozbočovače, aby sa obmedzilo bránenie pravdepodobnosti, zlepšila sa parita zaťaženia a všeobecne povedané vykonávanie.

Iný pohľad na vec sme zaznamenali pri štúdii [2] zaoberajúcej sa vytvorením všestranných komponentov na replikáciu informácií pre HDFS. Vytvorený komponent využíva pravdepodobnostné testovanie a výpočet skutočného dozrievania autonómne na každom náboji, aby rozhodol o množstve imitácií, ktoré sa majú distribuovať každému záznamu a oblasti pre každú reprodukciu. Program DARE (*Adaptive Data replication*) využíva existujúce vzdialené informácie na zotavenie a vyberá podmnožinu informácií, ktoré sa majú vložiť do rámca dokumentu, čím umožňuje napodobňovanie bez vynaloženia ďalších prostriedkov na organizáciu a výpočet.

CDRM a DARE sa pokúšajú nastaviť vhodnú replikačnú hodnotu pre každý záznam a umiestniť ich do citlivých dátových uzlov podľa aktuálneho nevyrovnaného zaťaženia a limitu strediska (*hub*). V žiadnom prípade neberú do úvahy dostupnosť informácií, ktoré majú nízke reprodukčné faktory. DiskReduce predstavuje stratégiu RAID (*redundant array of independent disks*) iba pre občasne využité informácie, ale neprináša podrobné pokyny, aby mohol rozhodnúť o skvelých informáciách. V ERMS (*Elastic Replication Management*) používame jedinečné imitačné prístupy pre rôzne informácie. Pre dôležite informácie rozširujeme číslo replikácie, aby sa zlepšilo vykonávanie. Pre bežné informácie využíva predvolenú trojnásobnú stratégiu. Rovnako zobrazujeme konkrétnu techniku situácie reprodukcie pre ďalšie imitácie horúcej informácie a rovnosť RAID.

V literatúre [56], v ktorej Khan a kol. predstavili vo svojom výskume výpočet, ktorý vyhľadáva ideálny počet obrazov kódových slov potrebných na zotavenie pre akýkoľvek kód odstránenia na základe XOR a poskytuje plány zotavenia, ktoré využívajú základnú mieru informácií. Tento výpočet zlepšuje vykonávanie I / O pre veľké štvorcové veľkosti používané v cloud-ových záznamových rámcoch, napríklad HDFS.

V literatúre [3], v ktorej autor Abawajy formuloval problém s replikáciou dát a navrhol distribuovaný algoritmus replikácie dát so zárukou konzistencie pre dátovú mriežku. Prístup spočíva v systematickom organizovaní lokalít údajovej mriežky do odlišných oblastí, v novej politike umiestňovania replík a v novej politike riadenia replík v kvorumbálnom tvare. Kvórum slúži ako základný nástroj na zabezpečenie jednotného a spoľahlivého spôsobu dosiahnutia konzistencie medzi replikami systému.

Hlavnou výhodou protokolov replikácie založených na kvóre je ich odolnosť voči poruchám uzlov a sietí. Z toho dôvodu každé kvórum s plne funkčnými uzlami môže udeľovať povolenia na čítanie a zápis, čo zlepšuje dostupnosť systému.

Ďalšou metodikou, ktorou sa špecialisti v obore zaoberajú je spoľahlivosťou systému. Zo skúmaní danej problematiky sa zistil problém s replikáciou, ktorý bol predstavený pod pojmom „replikácia dokumentov". K tomuto účelu boli predstavené dva postupy. Prvým postupom je ORCS (*Organizational Reliability Capability Assessment*) a druhým postupom je DMS (*Document Management System*). Obidva postupy boli predstavené v literatúre [113]. Autori práce v nej reprodukujú záznamy na základe stálej hodnoty. Akonáhle sa zmení jeho hodnota, algoritmus navrhne spôsob úpravy tak, aby systém DMS upravil informačnú oblasť podľa požiadaviek klienta a zmenšil odstup medzi centrami „peruser" a napodobeninami informácií. Navyše, ORCS a DMS pri hodnotení replikačného faktora berú do úvahy dostupnú ďalšiu miestnosť každého hostiteľa. Aby zhromaždili požadované údaje, používajú pozorovacie prístroje ako NWS (*Network Weather Service*) a Ganglia, a tieto údaje ukladajú do informačnej základne. Tento nástroj považuje za jeden zo svojich hlavných bodov záujmu vstupné územie pri replikácii dokumentov.

Mnohé spomenuté štúdie priniesli zaujímavé myšlienky a postupy, ako sa pozerať na spôsob získania replikačných koeficientov. V spomenutých metódach sa však neberie do úvahy využiteľnosť dát, ktoré sa v systéme nachádzajú. Neodzrkadľujú efektívne využitie replikačného koeficientu, s čím súvisí plytvanie úložného priestoru, spomaľovanie distribúcie dát a v neposlednom rade plytvanie nákladov na zabezpečenie klastra.

Po preštudovaní spomenutých prác stále vidíme priestor na zlepšenie. Rozhodli sme sa implementovať do procesu metódu, ktorá by dokázala proces posudzovania replikačného koeficientu pre jednotlivé tabuľky zefektívniť vo viacerých smeroch. Redukcia množstva replikačného koeficientu pre jednotlivé tabuľky, redukcia množstva záznamov v jednotlivých dátových uzloch a skrátenie času potrebného na presun dát do distribuovaného systému sú vlastnosti, ktoré nami navrhnutá metóda dokáže zefektívniť oproti už existujúcim riešeniam.

5.4.2 Dôvod zaoberania sa skúmanou problematikou spoľahlivosti systému

Ako hlavný problém, ktorý sme pri skúmaní problematiky vyhodnotili za kľúčovú zložku je replikačný koeficient. Pri presune dát je replikačný koeficient nastavený na základe striktných pravidiel, ktoré sa aplikujú až po presune dát do distribuovaného systému bez ohľadu na to, ako sú, respektíve boli dáta používané v ľubovoľnej databáze.

So zanedbaním posúdenia využiteľnosti jednotlivých dát narastá nutnosť ukladania väčšieho množstva záznamov v diskovom priestore, taktiež narastá čas presunu jednotlivých dát z databázy, a s tým súvisí aj vyššia vyťaženosť servera, ktorý musí vykonávať potrebnú operáciu.

V mnohých prípadoch je replikačný koeficient nastavený na prednastavenú hodnotu 3 [113]. Táto hodnota je často naozaj správne definovaná, avšak pri menšom počte, respektíve využiteľnosti dát je táto hodnota pri správne definovaných procesoch zbytočne veľká, a tým spôsobuje plytvanie výpočtových a diskových nákladov.

5.4.3 Databázový prístup pri stanovení replikačného koeficientu dát

Prístup, ktorým pristupujeme my k stanoveniu replikačného koeficientu sa oproti spomenutým metódam v kapitole 5.4.1 líši tým, že hodnoty sú posudzované na základe vyťaženia jednotlivých záznamov v databáze.

V súčasnosti sa stretávame so situáciou, kedy v mnohých vývojárskych článkoch a blogoch je stanovený replikačný koeficient na hodnotu 3. Podľa nášho úsudku replikačný koeficient by sa mal odvíjať od vyťaženosti dát v dátovom úložisku. Jednotlivé operácie výberu, aktualizovania a mazania dát pre jednotlivé tabuľky by mal určovať, koľkokrát sú dáta v stanovenom systéme replikované.

Na tieto účely sme využili metódu, ktorá nám poskytla informácie o vyťaženosti jednotlivých operácií v databáze Oracle a vyzerá takto:

```
SELECT  vss.owner,
              vss.object_name,
              vss.subobject_name,
              vss.object_type ,
             vss.tablespace_name,
SUM(CASE statistic_name WHEN 'logical reads'  THEN value ELSE 0 END
        + CASE statistic_name WHEN 'physical reads' THEN value ELSE 0 END) AS reads,
SUM(CASE statistic_name WHEN 'logical reads' THEN value ELSE 0 END) AS
        logical_reads ,
SUM(CASE statistic_name WHEN 'physical reads' THEN value ELSE 0 END) AS
        physical_reads ,
SUM(CASE statistic_name WHEN 'segment scans' THEN value ELSE 0 END)
        segment_scans ,
SUM(CASE statistic_name WHEN 'physical writes' THEN value ELSE 0 END) AS
        writes
FROM    v$segment_statistics vss
WHERE   vss.owner NOT IN ('SYS', 'SYSTEM')
AND vss.object_type = 'TABLE'
GROUP BY vss.owner,
                  vss.object_name,
                 vss.object_type,
                 vss.subobject_name ,
                 vss.tablespace_name
ORDER BY reads DESC;
```

Na základe spusteného príkazu sme získali celkové štatistiky o používateľoch a operáciách (výsledok je zobrazený na obr. 11), ktoré boli nad jednotlivými tabuľkami vykonávané.

Na základe získaných hodnôt sme jasne konštatovali, že niektoré záznamy sú pre správne fungovanie aplikácie dôležitejšie, to znamená, že nad dátami v jednotlivých tabuľkách boli vykonávané operácie vyhľadávania, respektíve úpravy a mazania častejšie ako pri iných tabuľkách.

	OWNER	OBJECT_NAME	OBJECT_TYPE	TABLESPACE_NAME	READS	LOGICAL_READS	PHYSICAL_READS	SEGMENT_SCANS	WRITES
1	KRUPA16	HASICSKY_UTOK	TABLE	STUDENTI_PDBS_2020	29347	17984	11363	0	0
2	HELIOS	FLIGHT_AUA_ACT_TAB_1512	TABLE	STUDENTI_PDBS_2020	28720	28720	0	2	0
3	GSMADMIN_INTERNAL	DDLID$	TABLE	SYSAUX	23264	23264	0	0	0
4	TESTER	P_PRISPEVKY	TABLE	UCITELIA	18254	17456	798	0	0
5	SOC_POISTOVNA	P_ODVOD_PLATBA	TABLE	UCITELIA	15878	11280	4598	0	0
6	GLEMBA1	SHOW_TIME	TABLE	APEX_28040623704324914	15280	8640	6640	0	0
7	PRIKLAD_DB2	ZAP_PREDMETY	TABLE	UCITELIA	13623	11856	1767	0	0
8	GLEMBA1	GENRES_OF_MOVIE	TABLE	APEX_28040623704324914	11421	6880	4541	0	0
9	PAVLECH1_APEX	GENRES_OF_MOVIE	TABLE	APEX_28040623704324914	8639	6352	2287	0	0
10	PAVLECH1_APEX	TICKET	TABLE	APEX_28040623704324914	7030	4544	2486	0	0
11	DBSNMP	BSLN_BASELINES	TABLE	SYSAUX	5842	5840	2	0	2
12	URBANIK7_APEX	FEE	TABLE	APEX_83802699956455532	4563	4464	99	0	0
13	XDB	XDB$RESOURCE	TABLE	SYSAUX	3712	1888	1824	0	0
14	TESTER	P_OSOBA	TABLE	UCITELIA	3476	3344	132	0	0
15	PAVLECH1_APEX	STAFF	TABLE	APEX_28040623704324914	1971	1792	179	0	0

Obrázok 11. Výsledok sql príkazu

Replikačný koeficient sa musí odvíjať práve od reálnych hodnôt, ktoré sme získali z databázy. Podľa hodnoty celkového čítania záznamov sme vytvorili pravidlo, ktoré nám určuje, koľkokrát musia byť dáta, s ktorými pracujeme replikovať, aby nedošlo k potenciálnej strate záznamov.

Na základe odporúčaných literatúr [113], kde je uvádzaný replikačný koeficient na hodnotu 3 sme usúdili, že hodnota 3 je pre kľúčové hodnoty naozaj opodstatnená, avšak na základe nášho navrhnutého algoritmu, ktorý sleduje vyťaženosť hodnôt dochádza k stanoveniu replikačného koeficientu dát nasledovne:

- Algoritmus rozdelí záznamy podľa počtu čítania a zápisu do tabuľky.
  - Pre zápis sme definovali váhový koeficient rovný 1 z dôvodu, že pri zápise je v mnohých prípadoch v pozadí oproti operácii čítania dát do databázy.
  - Pre čítanie sme stanovili váhový koeficient rovný 2 z dôvodu, že čítanie je v mnohých prípadoch častejšia operácia ako operácia zápisu dát do databázy.

- Následne sa vypočíta medián záznamov na základe všetkých tabuliek pomocou vzorca:

$$x = \frac{\sum_{1}^{n} * Rc * Wrc + Wc * Rwc}{n} \qquad (2)$$

kde ***x*** - znamená koeficienčný priemer

n - znamená počet tabuliek

Rc - znamená počet čítaní z tabuľky

Wrc - znamená váhový čítací koeficient

Wc - znamená počet zápisov do tabuľky

Rwc - znamená váhový zápisný koeficient

- Algoritmus prechádza hodnotami získanými z databázy a vykonáva nasledujúcu operáciu:
  - ak operácia pre prvú tabuľku je menšia ako *x* to znamená, že platí pravidlo

$$Rcl * Wrc + Wc * Rwc < x \qquad (3)$$

  - tak sa hodnota replikačného koeficientu nastaví na hodnotu 2, v inom prípade sa replikačný koeficient nastaví na hodnotu 3
- Následne algoritmus vykoná krok vyššie pre všetky tabuľky a výsledkom bude rôznorodý replikačný koeficient pre nami poskytnutú dátovú schému.

Na základe spusteného algoritmu sme vytvorili štruktúru, ktorá ku každej tabuľke definuje replikačný koeficient. Pomocou dátového modelu, ktorý je zobrazený na obr. 12, sa na Fakulte riadenia a informatiky na Žilinskej univerzite v Žiline vyučuje predmet Databázové systémy. Algoritmus na kontrolu replikácií pre jednotlivé tabuľky je definovaný v súbore a má jednoduchú štruktúru typu názov tabuľky a replikačný koeficient. Ukážku štruktúry súboru sme priložili nižšie.

personal_data 3

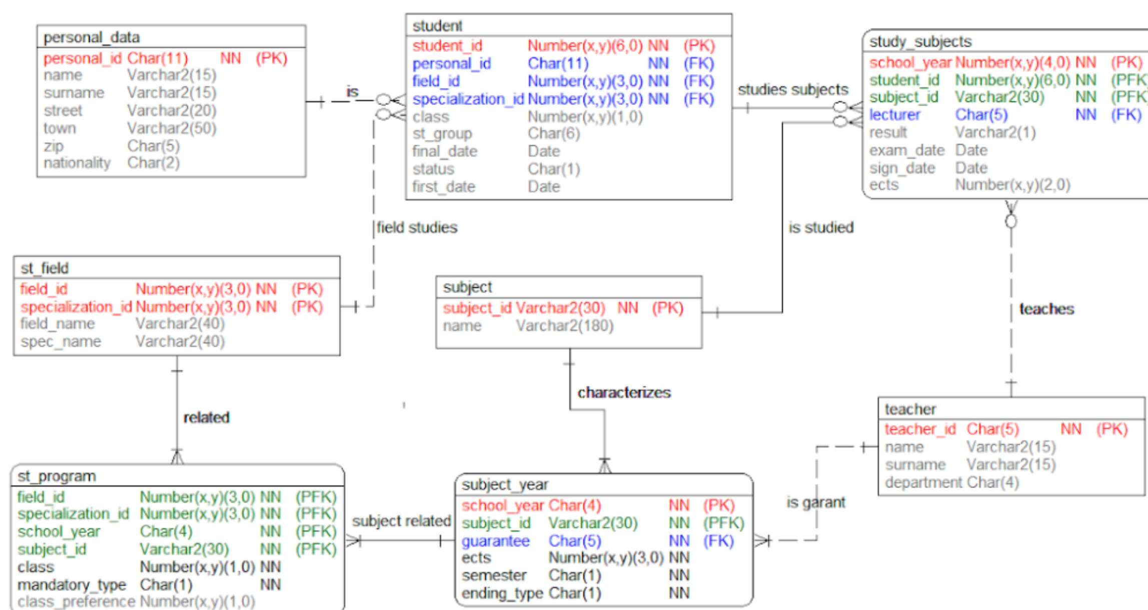
student 3

teacher 2

......

Záznamy a aj tabuľky sa samozrejme môžu počas sledovania replikačného koeficientu meniť, upravovať a tento súbor slúži na kontrolu správnosti replikovania údajov.

Experimentálne sme skúmali hodnotu koeficienčného priemeru s hodnotami záznamov z tabuliek a na základe týchto hodnôt sme upravili súbor s replikačnými koeficientami. Do súboru sme pridali počet záznamov, kedy je vhodné zvýšiť respektíve znížiť hodnotu replikačného koeficientu. Na tieto účely sme vytvorili 2 jednoduché metódy, a to metódu automatického škálovania smerom nahor a smerom nadol.



Obrázok 12. Vzorový dátový model

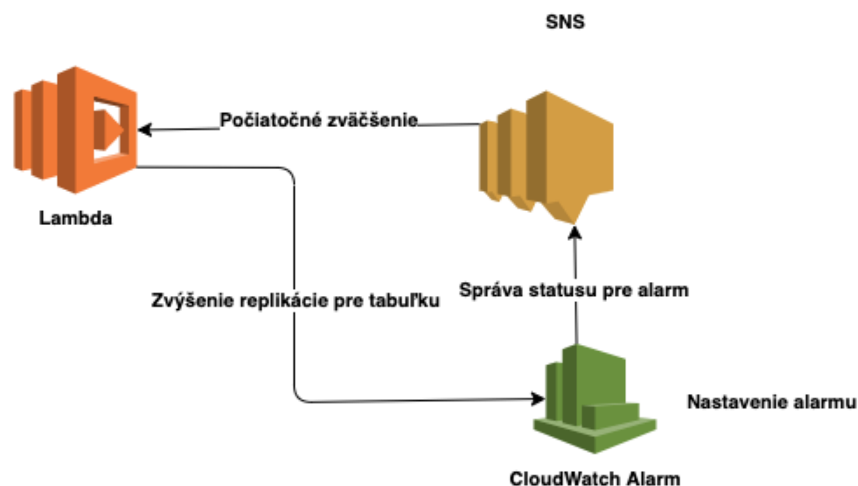
Počas testovania príkazu sme mali možnosť vidieť ako sa jednotlivé záznamy počas spúšťania dotazu nad databázou menia a ovplyvňujú nám predošlé výsledky. Na základe pozorovania sa meniacich výsledkov sme riešili aj situáciu spojenú s meniacimi sa hodnotami čítania a zápisu hodnôt do databázy a z databázy. Za účelom automatického prispôsobovania sa záznamov sme navrhli a implementovali 2 riešenia.

5.4.3.1 Automatické škálovanie smerom nahor

Automatické škálovanie smerom nahor umožňuje prispôsobovať množstvo potrebných replík dát na základe databázovej štatistiky a replikačného koeficientu. Architektúra automatického škálovania je zobrazená na obr. 13.

Vytvorená architektúra pre účely sledovania replikačného koeficientu funguje nasledovne.

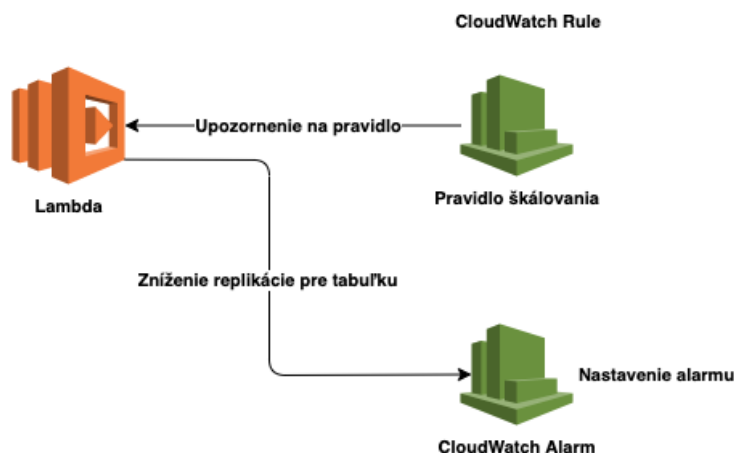
- Pomocou služby CloudWatch sledujeme štatistiky jednotlivých dát.
- Ak dôjde k situácii veľkého čítania záznamov tak zároveň dochádza aj k zmene štatistík.
- Vyvolaná metóda získa novú štatistiku a hodnoty sú následne pomocou vzorca znovu prepočítané :
    - Ak operácie čítania alebo zápisu nespôsobili zmeny v počte replikácií nedochádza k žiadnej zmene.
    - Ak operácie čítania alebo zápisu spôsobili zmeny v počte replikácií dochádza k zmene.



Obrázok 13. Architektúra pre zvyšovanie replikačného koeficientu

5.4.3.2 Automatické škálovanie smerom nadol

Automatické škálovanie smerom nadol umožňuje pripojovať množstvo potrebných replík dát na základe databázovej štatistiky a replikačného koeficientu. Architektúra automatického škálovania je zobrazená na obr. 14.



Obrázok 14. Architektúra pre zníženie replikačného koeficientu

5.4.4 Experimentálna činnosť replikačnej metódy

Na zistenie správneho priebehu našej aplikácie sme využili model zobrazený na obr. 12. Pre tieto účely sme sledovali hodnoty získané počas 1 semestra, študentami absolvujúcimi predmet Databázové systémy. Údaje z tabuliek sme uložili do súboru a nahrali sme ho na nasledujúcom odkaze https://github.com/romanceresnak/system-reliability. Pri využití príkazu (2) sme získali hodnoty, ktoré sú automaticky dosadené do vzorca (3) a získali sme koeficient priemeru.

Tento koeficient nám pomohol pri určovaní, koľkokrát majú byť dáta pre danú tabuľku replikovateľné a výsledok bol nasledovný:

- personal_data replication coefficient = 3
- student replication coefficient = 3
- study_subject replication coefficient = 3
- subject_year replication coefficient = 3
- teacher replication coefficient = 3
- st_field replication coefficient = 2
- st_program replication coefficient = 2
- subject replication coefficient = 2

Hodnoty pre každú tabuľku boli podrobené na začiatku procesu počiatočnej analýze. Na základe dosadenia hodnôt, ktoré sme získali z príkazu (1) boli dosadené do vzorca (2), ktorý nám určil hraničnú hodnotu pre replikačný koeficient. Po získaní hraničnej hodnoty sme dosadili hodnoty pre každú tabuľku do vzorca (3). Výsledná hodnota zo vzorca (3) nám definovala, či replikačný koeficient pre tabuľku je rovný hodnote 2 alebo 3.

Na základe týchto hodnôt sme vytvorili súbor s identickou štruktúrou. Hodnoty replikačných koeficientov vstupujú do systému vždy, keď dochádza k spusteniu operácie *map reduce* ako vstupný parameter hovoriaci, koľkokrát sa dáta budú v systéme vyskytovať. Na základe tohto aspektu sme dynamicky menili hodnoty replikácie na základe dát, ktoré sa v systéme nachádzali.

Systém Hadoop Distributed File System (HDFS) ukladá súbory ako dátové bloky a distribuuje ich v celom klastri. Pretože bol systém HDFS navrhnutý tak, aby bol odolný voči chybám a fungoval na komoditnom hardvéri, bloky sa opakujú niekoľkokrát, aby sa zabezpečila vysoká dostupnosť údajov. Faktor replikácie je vlastnosť, ktorú je možné nastaviť v konfiguračnom súbore HDFS, a ktorá nám umožňuje upraviť globálny faktor replikácie pre celý klaster. Pre každý blok uložený v HDFS je v klastri distribuovaných n - 1 duplikovaných blokov.

Pre účely replík sme v konfiguračnom súbore hdfs-site.xml (ktorému sme nemenili pozíciu a pri predvolenom nastavení sa nachádza v priečinku *conf/*) pridali nasledujúcu vlastnosť pre každú tabuľku:

```
<property>
    <name>dfs.replication<name>
    <value>3<value>
    <description>Block Replication<description>
<property>
```

Vytvorená konfigurácia ma predvolené nastavenie replikačného koeficientu na hodnotu 3. Pri prvých 5 tabuľkách v zozname je tento koeficient správny, ale pre posledné 3 sa musí replikačný koeficient zmeniť na hodnotu 2, a preto je do systému HDFS vyslaný nasledujúci príkaz:

```
hadoop fs –setrep –w 2 /my/replica_file
```

Príkaz zabezpečí zmenu replikačného koeficientu na hodnotu 2.

Pre tieto účely sme využili cloud od spoločnosti Amazon s nasledujúcou konfiguráciou.

Tabuľka 3. Konfigurácia klastra

EC2 Instance	a1.medium vCPU: 1 MeM(GiB): 2
EMR cluster	master: 1x m3.xlarge core: 2x m4.4xlarge

Pre Hadoop sme zvolili nasledujúcu konfiguráciu:

```
Host namenode
  HostName ec2-18-216-40-160.us-east 2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

```
Host datanode1
  HostName ec2-18-220-65-115.us-east-2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

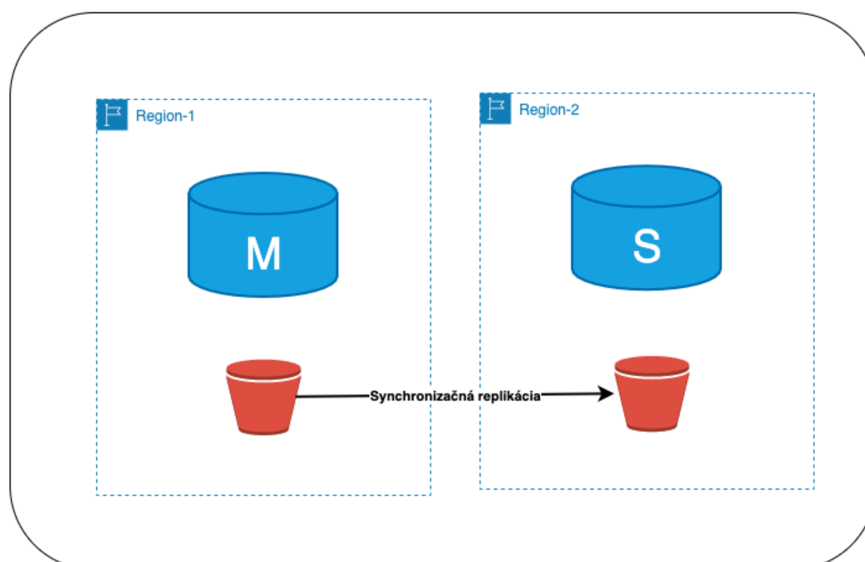
```
Host datanode2
  HostName ec2-52-15-229-142.us-east-2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

```
Host datanode3
  HostName ec2-18-220-72-56.us-east-2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

Pri prvotnom spustení experimentov s odporúčanou hodnotou replikačného koeficientu rovnou 3 a hodnotami, ktoré sme získali pri využití nami vytvorenej metódy replikačného koeficientu, bol čas potrebný na úpravu nasledujúci:

- Pri predvolenej hodnote bol čas 2 minúty a 38 s
- Pri upravenej metóde bol čas 2 minúty a 24 s

Pri zvolenej hodnote replikačného koeficientu rovnajúcemu sa 2, by v prípade výpadku 1 uzla znamenalo, že nám ostali dáta iba pre 1 replikáciu. Rozhodli sme sa dáta replikovať namiesto v 1 zóne dostupnosti v 1 regióne, na 1 zónu dostupnosti v 2 regiónoch ako je zobrazené na obr. 15, čo nám redukuje množstvo chýb od výpadku siete až po rôzne katastrofy.



Obrázok 15. Navrhnutá architektúra

Pri hlbšom preskúmaní jednotlivých dátových uzlov a vyťaženia výpočtových jednotiek, sme dokázali redukovať množstvo celkového vyťaženia uzlov v klastri v prepočte o 2 % na 1 uzol[2].

Túto hodnotu sme získali experimentom, pri ktorom sme porovnávali rovnaké množstvo hodnôt pri nasadení našej hodnoty oproti konvenčnej metóde. S touto metódou súvisí aj redukcia množstva úložného priestoru potrebného na replikáciu údajov v dátových úložiskách, a tiež redukcia výpočtových jednotiek potrebných na vykonanie potrebnej operácie.

Ako je vidieť z tabuľky 4**,** tak sledované hodnoty, ktoré sú pre nás podstatné sa oproti konvenčnej metóde pri rovnakej konfigurácií zlepšili takmer v každom spektre. Operácia potrebná na získanie informácií z databázy je pre nás veľmi dôležitá na získanie štatistických ukazovateľov, s ktorými následne v procese pracujeme.

Aj keď spomenutá hodnota operácie pred spustením procesu trvá o 2 sekundy dlhšie v našom prístupe ako operácia po spustení procesu v konvenčnej metóde, (operácia pred spustením procesu v našom prístupe je 8 sekúnd a operácia po spustení procesu v konvenčnej metóde je 6) mohol by sa tento prístup javiť ako časovo zdĺhavejší.

[2] Problematiku replikačného koeficientu sme prezentovali a diskutovali na IEEE konferencii Reliability Engineering and Computational Intelligence na Slovensku (Žilina) - 27.- 29. október 2020.

Nakoľko s vykonaním operácie pred spustením procesu vieme efektívne spracovať počet replík, redukcia celkového množstva ako je ukázané v tabuľke 4 spôsobila, že celkový čas na spracovanie všetkých záznamov a replík je v konečnom dôsledku efektívnejší oproti konvenčnej metóde.

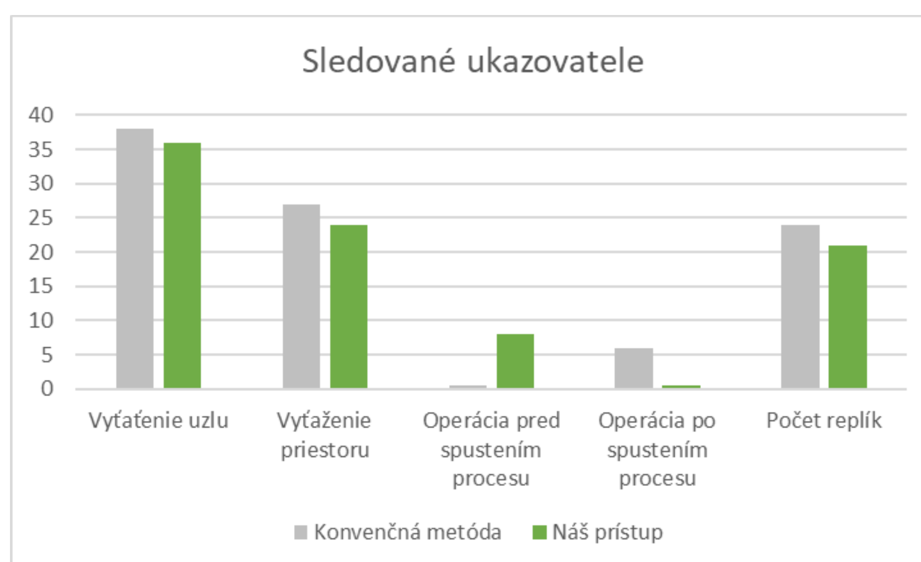
Tabuľka 4. Porovnanie sledovaných hodnôt medzi konvenčnou metódou a naším prístupom

Číslo vlastnosti	Sledovaná vlastnosť	Konvenčná metóda	Náš prístup
1	Vyťaženie uzlu [%]	38	36
2	Vyťaženie priestoru [%]	27	24
3	Operácia pred spustením procesu [s]	0	8
4	Operácia po spustením procesu [s]	6	0
5	Počet replík	24	21
6	Celkový čas	2 minúty a 38 s	2 minúty a 24 s

Ako je vidieť z grafu na obr. 16, tak grafické zobrazenie výsledkov poukazuje na jasný trend. Zelenou farbou sú zobrazené výsledky po implementovaní nášho prístupu a šedou sú zobrazené výsledky s použitím konvenčnej metódy.

Okrem sledovaného ukazovateľa „operácia pred spustením procesu“ sú sledované hodnoty v porovnaní s hodnotami dosiahnutými konvenčnou metódou vždy efektívnejšie, či už sa jedná o jednotky v percentách, sekundách a počte replík.

Po skončení procesu sme získali uspokojivý výsledok, ktorý bol v prepočte o 12 sekúnd kratší ako za použitia konvenčnej metódy.



Obrázok 16. Porovnanie sledovaných ukazovateľov pri využití dvoch prístupov

Pri distribuovanom spracovaní údajov sa nám v mnohých prípadoch vyskytol jav, ktorý pri presune údajov do systému považujeme za veľké negatívum, a taktiež aj ako priestor na zlepšenie procesu. Za zmieneným jav považujeme sekvenčné spracovanie údajov v tabuľkách.

Sekvenčné spracovanie spôsobovalo čakanie na spracovanie jednotlivých údajov v mnohých tabuľkách, čo malo za následok tzv. „uviaznutie“ tabuliek, ktoré sa mohli spracovávať paralelne s inými tabuľkami, a tým redukovať čas potrebný na presun záznamov do distribuovaného systému.

Pri sledovaní spomenutého nedostatku nám relačná databáza stále slúžila ako „živá“ databáza (databáza stále plnila účel primárneho úložiska) a počas procesu presunu dát nám do databázy prichádzali stále nové dáta a nové operácie (operácie aktualizovania (*update*) a mazania záznamov (operácia (*delete*)).

Na základe zisteného nedostatku sa domnievame, že zavedenie paralelných procesov a metóda vzájomného ovplyvňovania záznamov pri presune údajov z tabuliek do distribuovaného systému má potenciál na zrýchlenie procesu a redukciu času.

5.5 Zvýšenie paralelizmu pri migrácii dát

Na základe predpokladu z konca predchádzajúcej kapitoly sme usúdili, že nahradenie sekvenčného spracovania paralelným spracovaním môže viesť k rýchlejšiemu presunu záznamov z relačnej respektíve nerelačnej databázy do distribuovaného systému, a tým pádom ešte viac zrýchliť spomenutý proces.

Nakoľko operácia presunu údajov z databázy do distribuovaného systému trvá určitý čas a databáza stále plní úlohu primárneho úložiska, musíme zobrať do úvahy aj dáta, ktoré sa počas tohto procesu vyskytnú tzv. „dáta v reálnom čase“.

Na základe spomenutých nedostatkov a prichádzajúcich dát sme si stanovili pri skúmaní tejto problematiky nasledujúce ciele:

- Redukovať sekvenčné spracovanie záznamov na minimálnu hodnotu
- Aplikovať paralelný proces pri presune záznamov
- Redukovať množstvo dodatočných úprav v distribuovanom systéme
- V čo najväčšej miere ovplyvňovať práve sa transformujúce údaje, údajmi v reálnom čase

Nakoľko zrýchlenie procesu spracovania údajov je pomerne rozšírená téma a viacerí výskumníci sa zamerali na spomenutú problematiku, tak v nasledujúcej podkapitole preskúmame už existujúce riešenia.

5.5.1 Štúdie zaoberajúce sa paralelným spracovaním dát

Relačné databázy a ďalšie tradičné databázy sú spravované prísne definovanou štruktúrou pre organizáciu údajov generovaných z rôznych aplikácií. Databázy NoSql však poskytujú flexibilitu počas organizácie údajov [34], čo uľahčuje prístup k údajom. Dáta generované zo sociálnych stránok a aplikácií v reálnom čase si vyžadujú flexibilný a škálovateľný systém, ktorý zvyšuje potrebu NoSql [21]. Pre migráciu dát bol navrhnutý multidimenzionálny model. Pre každú migráciu údajov je veľkou výzvou práca s údajmi vstupujúcimi do procesu transformácie. Mnoho výskumníkov sa zaoberalo spomenutým problémom so zámerom zabezpečiť riešenie, ktoré dokáže efektívne manipulovať s dvoma prichádzajúcimi dátovými typmi. Prvým problémom je reprezentácia aplikačnej domény na základe dátového toku, ktorá poskytuje progresívne zobrazenie udalostí v čase ich výskytu. Nezachovala ich však bez časového obmedzenia. Údajové položky sú k dispozícii iba v konkrétnom časovom spektre, a v prípade zlyhania sa plnenie procesu vykonania operácie z procesu vymaže.

Druhým problémom sú situácie zachytávajúce údaje, ktoré sa vyskytujú v prevádzke. V prípade ich iného výskytu alebo ovplyvnenia údajov to pre systém nemusí byť potrebné. Preto neovplyvňujú meniace sa údaje [70] [23].

Vzhľadom na to, že databáza má obmedzený počet prístupov, je potrebné posúdiť hodnoty, ktoré svoju prácu vykonali súčasne a rozhodnúť o ďalšom výskyte v systéme. Predchádzanie hromadeniu zbytočných údajov v tradičných databázach [32], databázach prúdov a databázach, ktoré uchovávajú hodnoty v pamäti, je možné definovať časové intervaly. Celkovú efektivitu a hromadenie zbytočných informácií v databázach možno teda zvýšiť [92].

Niektorí výskumníci sa zaoberali výskytom údajov v reálnom čase. V takýchto systémoch sa používa pre každú operáciu časová pečiatka na plánovanie udalostí. Jednotka mapovača priorít priradí každej udalosti počas vstupu do systému úroveň dôležitosti, ktorá závisí od toho, ako systém sleduje časy a ďalšie priority [72] [111]. Spôsob časovej pečiatky závisí od času vstupu do systému.

Vedci naznačujú, že pre väčšinu štúdií sú udalosti sporadické s nepredvídateľným časom príchodu. Vedci nastavili porovnanie rôznych plánovacích algoritmov. Medzi najčastejšie používané postupy patria tieto štyri:

- ***Prioritné blokovanie (PB) -*** mechanizmus PB je podobný bežným blokovacím protokolom v tom, že transakcia je vždy zablokovaná, keď dôjde ku konfliktu operácií a zámok je možné získať až po vyriešení konfliktu. Fronta požiadaviek na uzamknutie je však zoradená podľa priority transakcie .
- ***Prioritné prerušenie (PA) -*** schéma PA sa pokúša vyriešiť všetky konflikty údajov v prospech transakcií s vysokou prioritou. Konkrétne, v čase konfliktu zámku údajov, ak má kohorta (*cohort*) alebo aktualizátor, ktorý drží zámok, vyššiu prioritu ako priorita kohorty alebo aktualizátora, ktorý žiada o zámok, je žiadateľ zablokovaný. V opačnom prípade je držiak zámku prerušený a zámok je pridelený žiadateľovi. Po prerušení kohorty (*cohort*) aktualizátora (*updater*) sa na jej hlavný server kohorta (*cohort*) odošle správa, aby prerušila a potom opäť spustila celú transakciu (ak do tejto doby ešte neuplynula lehota).
- ***Prioritné dedičstvo (PI) -*** kedykoľvek dôjde ku konfliktu údajov, je žiadateľ vložený do zámku front požiadaviek, ktorý je zoradený podľa priority. Ak priorita žiadateľa je vyššia ako priorita ktorejkoľvek zo súčasných držiakov (*holder*) zámku, potom tie s nízkou prioritou kohorta (*cohort*) (*skupiny*) držiace zámok následne požiadajú o prioritu žiadateľa, to znamená, že „dedia“ túto prioritu. To znamená, že držiaky (*holders*) zámkov vždy vykonávajú buď svoju vlastnú prioritu, alebo prioritu skupiny s najvyššou prioritou čakajúcou na zámok, podľa toho, ktorá hodnota je väčšia.
- ***Prednostné čakanie (PW) -*** V mechanizme PW, ktorý sa používa v spojení s protokolom OCC je transakcia, ktorá dosiahne validáciu a nájde vyššiu prioritu transakcie v jej súbore konfliktov sú „umiestnené na poličke“, čo znamená, že sú definované na počkanie a nemôžu sa okamžite zaviazať. To dáva transakciám s vyššou prioritou šancu stihnúť termíny ako prvé. Keď všetky konfliktné transakcie s vyššou prioritou opustia konflikt nastavený buď kvôli príkazu, alebo kvôli prerušeniu, môže sa čakateľ zaviazať. Upozorňujeme, že čakajúca transakcia sa môže reštartovať z dôvodu vykonania jednej z konfliktných transakcií s vyššou prioritou.

V článku [106] vedci uskutočňujú migráciu databáz pomocou synchronizácie údajov a replikácie transakcií. Vedci používajú webové služby Azure a Amazon ako služby migrácie databáz. Tieto techniky umožňujú ľahkú replikáciu údajov databázy Sql a umožňujú viacerým databázam Sql izolovať kritické pracovné záťaže od analytických dotazov, ktoré bežia relatívne dlhšie. Vyhodnotenie porovnáva tieto dve migračné služby na základe synchronizačného času migrácie plne načítanej databázy.

Výsledok migrácie ukazuje, že migrácia databázy pomocou transakčnej replikácie a synchronizácie údajov je ovplyvnená počtom riadkov a veľkosťou tabuľky, umiestnením servera a rýchlosťou nahrávania a sťahovania v každej službe migrácie databázy.

Vedci v článku [31] popisujú techniku efektívnej migrácie živej (stále používanej) databázy v transakčnom procese. Autor Zephyr využíva fázy dopytu a asynchrónneho posielania dát, vyžaduje minimálnu synchronizáciu, je výsledkom nedostupnosti služby a niekoľkých alebo žiadnych prerušených transakcií, minimalizuje réžiu prenosu dát, poskytuje počas migrácie záruky ACID a zaisťuje správnosť v prípade zlyhania. V článku [31] výskumníci načrtávajú implementáciu prototypu pomocou modulu otvorenej relačnej databázy a predkladajú dôkladné vyhodnotenie pomocou rôznych transakčných úloh. Účinnosť metódy od autora Zephyr je zrejmá z niekoľkých desiatok zlyhaných operácií, zmeny priemernej latencie transakcií o 10 - 20%, minimálneho množstva správ a žiadnych réžií počas normálneho procesu pri migrácii živej databázy.

S prácou, ktorá sa zaoberá časovo orientovanou databázovou architektúrou, sme sa stretli počas nášho výskumu v príspevku [64], ktorý spravuje nedefinované hodnoty a navrhuje komplexnú klasifikáciu systémov na transakciách, prístupoch a indexoch. Pretože do nášho systému môžu vstupovať rôzne dátové typy, či už sú to štruktúrované alebo neštruktúrované dáta, v spomínanej práci sa zaznamenáva modelovanie nedefinovaných hodnôt. Ďalej pokrýva synchronizačné procesy využívajúce skupiny údajov. Kritickou súčasťou uvedeného článku sú riešenia pre efektívny zber dát s dôrazom na nedefinované hodnoty a stavy.

Ďalší pohľad na to, ako zlepšiť migračný proces, je uvedený v článku [50]. Hlavnou myšlienkou je asynchrónne spravovať údaje, ktoré sa následne zlúčia. Štúdia sa začala v roku 2006 a je výsledkom hĺbkovej analýzy. Dosiahnutým výsledkom štúdie je vytvorenie architektonického návrhu distribuovaného informačného systému s asynchrónnymi aktualizačnými údajmi.

Počas vývoja vedcov došlo k záveru, že je potrebné ukladať dáta na serveri vo verzii. Ich odlišný prístup k riešeniu spoľahlivosti v spomínanom príspevku využíva nové techniky ukladania verzionovaných údajov do unitemporálnej relačnej databázy. Ukladanie predstavuje odklon od tradičných bezpečnostných postupov. Vytvorené riešenie môže tiež zachovať výhody RDBMS, ako je referenčná integrita a spracovanie transakcií.

Ako je zrejmé z mnohých publikácií, proces migrácie údajov sa uskutočňuje nielen na pevných počítačoch, ale aj na cloud-ových službách od spoločností Amazon, Microsoft alebo Google. Tieto práce sa však nezameriavajú na vzájomné ovplyvňovanie údajov, ktoré do systému vstupujú v reálnom čase a nechávajú túto úpravu úplne na koniec. Naša štúdia tento problém rieši počas prebiehajúceho procesu, presne vtedy, keď údaje vstúpia do systému.

Paralelný proces, ktorý je potrebný pri migrácii dát musí spĺňať viacero kritérií, ktoré sme v preštudovaných článkoch nenašli, a preto sme aplikovali odlišný spôsob práce. Metóda funguje na princípe vyťažovania jadier, vzájomného ovplyvňovania spustených procesov a inej štruktúre ako doteraz publikované štúdie.

5.5.2 Pomocná databáza Redis

Redis je veľmi ľahko použiteľné a pritom výkonné NoSql pamäťové (*in-memory*) úložisko typu *key-value*.

- NoSql - nejde o relačnú databázu, preto nie sme limitovaný žiadnou predpísanou štruktúrou dát, a tak môžeme ukladať akékoľvek štruktúrované dáta,
- in-memory - dáta sa ukladajú v operačnej pamäti servera, takže je práca s dátami extrémne rýchla,
- key-value - dáta (*value*) sú zapisované a aj čítané na základe kľúča (*key*), ktorý si zvolíme sami.

Redis sa využíva hlavne ako medzikrok, medzi aplikáciou a dátovým úložiskom. Týmto spôsobom dokážeme zrýchliť webovú stránku. Ak aplikácia potrebuje určité dáta, najskôr sa pozrie do databázy Redis, či také dáta má uložené. Ak sú v Redis, tak si ich aplikácia pomocou príkazu vyžiada. Ak sa tieto dáta v Redis nenachádzajú, aplikácia si ich vyžiada z trvalej databázy ako je ľubovoľný typ či už sa jedná o relačnú alebo nerelačnú databázu, pričom ich zároveň zapíše aj do Redis, aby boli pri budúcej požiadavke dostupné rýchlejšie priamo z Redis.

Najčastejšie sa Redis používa ako medzipamäť (*cache*) pre trvalé dáta uložené iným spôsobom. Nie je dôležité, aby boli dáta trvalo uložené na serveri, preto pri reštarte servera dôjde k vymazaniu pamäte a celý cyklus s čítaním a zápisom, spomínaný vyššie, sa zopakuje. Aplikácie obyčajne sami nastavujú čas exspirácie údajov v Redis, pretože majú obyčajne iba obmedzenú platnosť a veľkosť pamäte.

5.5.3 Pomocná databáza Memcached

Memcached je open-source produkt šírený zdarma, ktorého účelom je poskytnúť systém pre *cachovanie* dát v pamäti s dôrazom na vysoký výkon, škálovateľnosť a nasadenie v distribuovaných scenároch. Projekt vznikol pri vývoji služby *LiveJournal* a prvotným účelom bolo zrýchliť aplikáciu implementovaním medzipamäte (*cachovaním*) často používaných dát z databázy do pamäte. Dnes je memcached jedným z najpoužívanejších nástrojov svojho druhu a jedným zo základných pilierov infraštruktúry najväčších služieb na webe ako napríklad Facebook, Wikipedia, Google/YouTube, Digg, Amazon a mnoho ďalších.

Koncept memcached je ten, že memcached poskytuje *key-value* úložisko bežiace ako služba separátne od aplikácie. Nezávislosť na aplikácii pracujúcou s memcached je veľmi dôležitým faktorom, ktorý umožňuje okrem iného napríklad aj jednoduchý nasadenie (*deployment*) ďalších inštancií memcached serverov. Memcached teda beží ako separátna služba/daemon a čaká na prichádzajúce spojenia na TCP porte. Klient, najbežnejšie webový server, iniciuje TCP spojenie na memcached server, pošle príkaz pre získanie alebo odoslanie dát, spracuje odpoveď a odpojí sa. V praxi ale klientske knižnice pre memcached využívajú rôzne implementácie tzv. *socket pools*, aby sa eliminovala potreba otvárať separátne spojenie pre každú požiadavku (*request*) do medzipamäte (*cache*).

5.5.4 Dôvod skúmania problematiky zvyšovania paralelizmu

Zvýšenie rýchlosti presunu dát môžu ovplyvniť viaceré faktory, medzi ktoré patrí počet tabuliek, ktoré sa majú presunúť z jedného dátového úložiska do druhého, referenčná integrita a samozrejme množstvo záznamov v tabuľkách. Z dôvodu riešenia presunu dát za pomoci sekvenčného prístupu často dochádza k pomalému presunu záznamov.

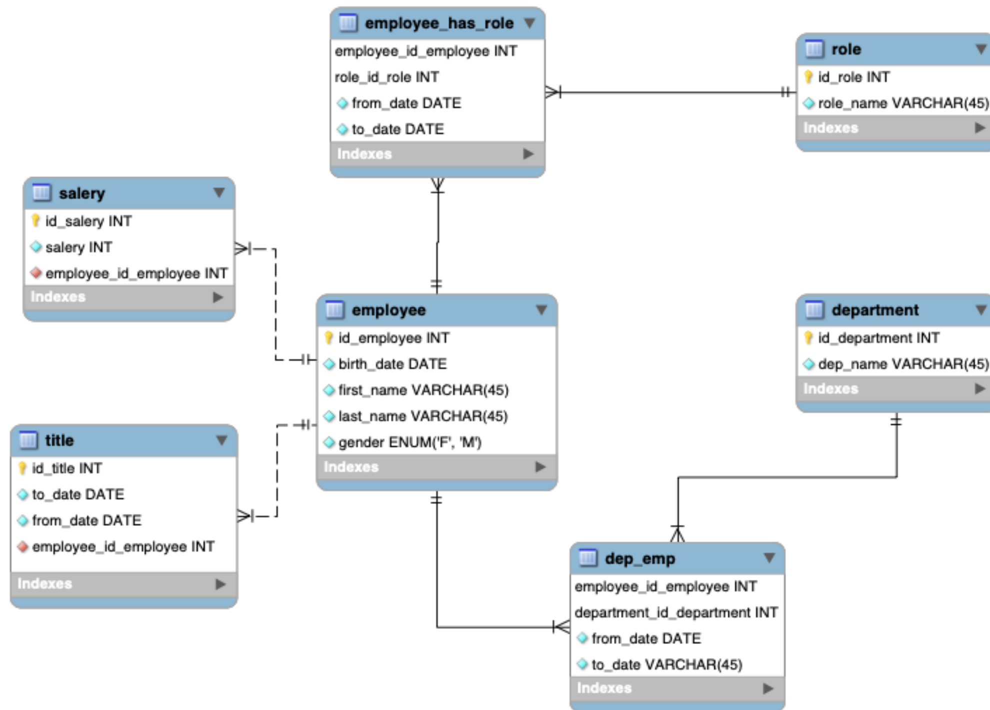
Pri preverovaní tohto problému sme skúmali štúdiu [31], v ktorej sa autor zameral na migráciu živej databázy pomocou fázy dopytu a asynchrónneho posielania údajov, čo v konečnom dôsledku spôsobuje nedostupnosti služby a niekoľkých alebo žiadnych prerušených transakcií .

Na základe záverov výskumníka Zephyra sme konštatovali, že aj keď sekvenčné spracovanie je dostatočne efektívne, má jeden nedostatok, a tým je zanedbanie spustenia paralelného spracovania tabuliek, ktoré nemajú na sebe nejakú logickú závislosť (neovplyvňujú sa alebo na seba priamo neodkazujú). Z tohto dôvodu sme vytvorili princíp výpočtu vzájomných vzťahov, ktorý sme popísali nižšie.

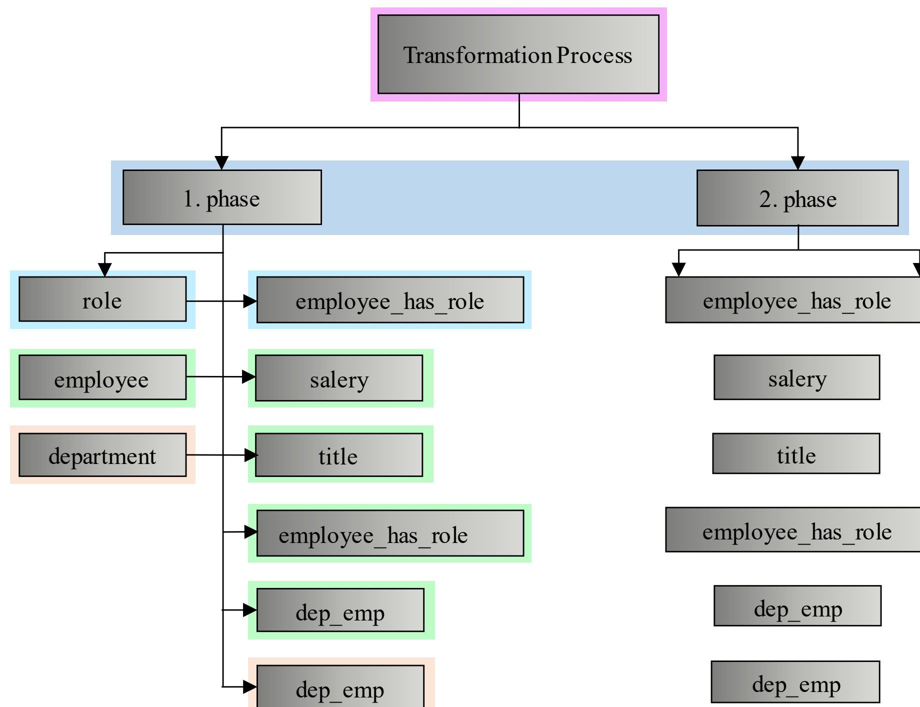
5.5.5 Naše riešenie zvýšenia paralelného spracovania dát

Hlavnou myšlienkou tejto časti je navrhnúť metódu, ktorá dokáže efektívne manipulovať s údajmi počas prebiehajúceho procesu transformácie. Na tieto účely sme využili údaje na nasledujúcej adrese https://github.com/datacharmer/test_db. Zmena údajov z relačnej databázy MySql na nerelačnú databázu MongoDB a späť, z databázy MongoDB do databázy MySql. Dáta vstupujúce do prebiehajúceho cyklu sú zachytávané synchronizačnou bariérou, ktorá pri svojej zmene upravuje transformačný proces[3]. Preto znižuje čas ich modifikácie po ukončení transformačného procesu.

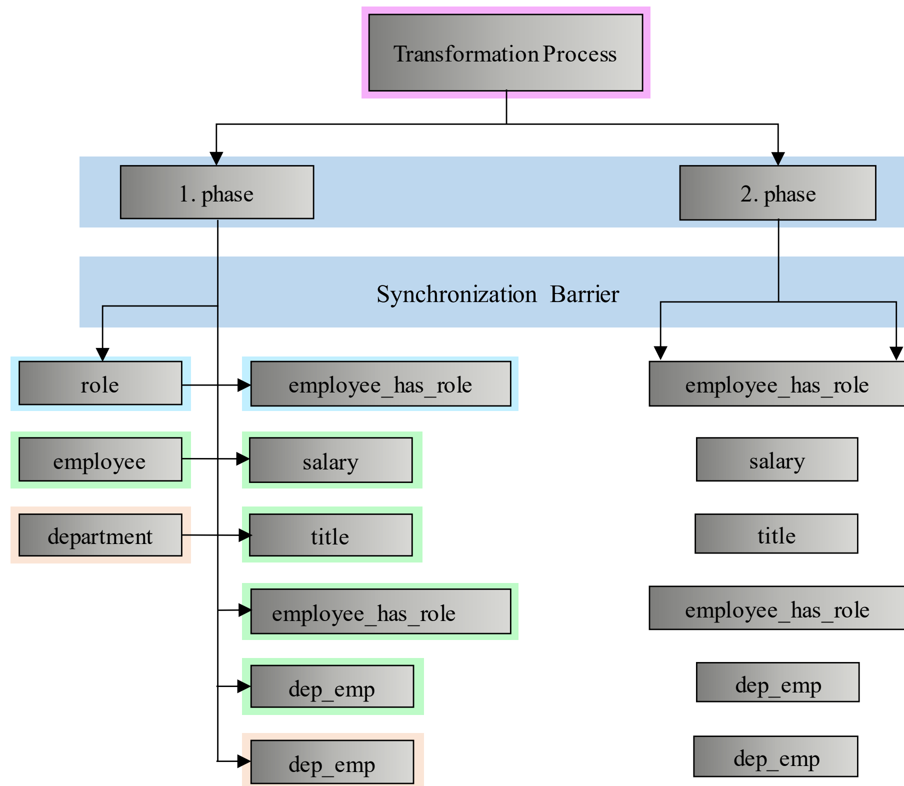
[3] Problematiku synchronizačnej bariéry sme prezentovali a diskutovali na IEEE konferencii ICETA na Slovensku (Vysoké Tatry) - 11. - 12. november 2021.



Obrázok 17. Dátový model pre transformačné účely



Obrázok 18. Transformačná fáza



Obrázok 19. Transformačná fáza s využitím synchronizačnej bariéry

Hlavným cieľom transformačnej bariéry je sledovanie dvoch typov údajov. Prvým typom sú údaje, ktoré sú v systéme už dlho. Údaje sú uložené v databáze MySql. Počas tohto procesu ich môžu ovplyvňovať hodnoty, ktoré vstupujú do systému v čase zmeny alebo prenosu hodnôt z jedného typu databázy do druhého. Počas tohto procesu môžu existovať určité kolízie, ktoré by bolo potrebné vykonať po transformačnom procese, čo by negatívne ovplyvnilo čas potrebný na transformáciu hodnôt. Kolízia je situácia, kedy dáta, ktoré sa práve transformujú sú ovplyvňované hodnotami alebo príkazmi, ktoré práve transformované dáta ovplyvňujú. Môžu aktualizovať práve transformované dáta, môžu vymazávať konkrétnu hodnotu, ba dokonca môžu vymazať celú tabuľku. Kolízia spôsobuje rýchlejšiu transformáciu, pretože môže redukovať vykonanie operácií pri transformačnom procese. Synchronizačná bariéra rieši vplyv údajov akumulovaných v systéme na hodnoty systému. Tu je jednoduchá situácia, ktorá jasne popisuje dôležitosť metódy, ktorú sme vytvorili:

Hodnoty z tabuľky oddelenia (*department*) prichádzajú do systému. Počas presunu údajov z jedného typu databázy do iného typu vstupuje do systému požiadavka, aby odstránila všetky hodnoty z tabuľky oddelenia (*department*).

Tento stav je zachytený synchronizačnou bariérou a následne je uložený v danom stave. Po zachytení tejto operácie v reálnom čase sa vždy odstránia všetky hodnoty, ktoré sa majú presunúť do cieľovej databázy tabuliek oddelení (*department*).

Druhou časťou nami navrhovanej metódy je zvýšiť paralelnosť pri presune údajov z jedného typu databázy do druhého, ktorá funguje nasledovne:

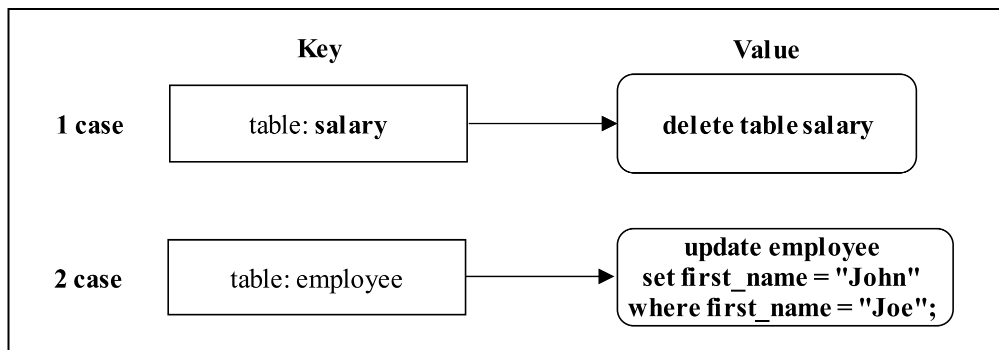
V prvom kroku sa skúma štruktúra. Vytvorený algoritmus prechádza celou schémou, zistí počet cudzích kľúčov a zložených primárnych kľúčov. Na obr. 17 je vidieť tabuľkový zoznam, ktorý sa používa na experimentálnu akciu a pohodlnejšiu predstavu o tom, ako funguje synchronizačná bariéra. Dátový model predstavuje situáciu, ktorá modeluje prípad v jednoduchom systéme. Osoba v spoločnosti je zamestnancom a môže mať rôzne pozície. Dostáva plat za svoju prácu a je pridelená na určité obdobie. V prvom kroku alebo v prvej fáze, ako je znázornené na obr. 18, sa najskôr transformujú tabuľky, respektíve dátové záznamy, ktoré majú najmenšie množstvo cudzích kľúčov alebo zložených primárnych kľúčov.

Tabuľky Rola, Zamestnanec a Oddelenie vidíme na obr. 18, v prvej fáze vľavo, ktorá obsahuje najmenšie množstvo cudzích, respektíve kompozitných primárnych kľúčov. V pravej časti na obr. 19 sú tabuľky, konkrétne tabuľky *zamestnanec_has_role, plat, titul, dep_emp*, ktorých funkčná závislosť je na hodnotách, respektíve tabuľkách vľavo. Uvedené tabuľky a hodnoty v nich na seba nepôsobia, a tak sa počas transformácie vyhodnocujú súčasne rovnakým spôsobom ako ich hodnoty. Tabuľky a hodnoty, ktorých funkčná závislosť bola od prvej fázy, sa transformujú v druhom kroku, respektíve v druhej fáze. Jedná sa konkrétne o štruktúru tabuľky *zamestnanec_has_role, plat, titul a dep_emp.*

Tabuľka *employee_has_role* obsahuje dva zložené primárne kľúče, *id_employee*, ktorý odkazuje na primárny kľúč tabuľky zamestnancov, a *id_role*, čo odkazuje k primárnemu kľúču v tabuľke roľa (*role).* Počas transformácie sa zmenili obe tabuľky - zamestnanci a aj tabuľka rolí (*role*). Ich hodnoty boli v prvom kroku transformované, čo umožnilo transformáciu funkčne a hodnotovo závislých tabuliek na nich.

Hlavný rozdiel medzi obr. 18 a obr. 19 je použitie synchronizačnej bariéry. Uvedená bariéra funguje ako kontrolný mechanizmus, ktorý monitoruje, aká hodnota sa transformuje, a či uvedené záznamy nie sú ovplyvnené hodnotou, ktorá práve vstúpila do systému, a teda aktualizuje alebo odstraňuje konkrétne hodnoty. Synchronizačná bariéra spolupracuje v nerelačnej databáze, ktorá pracuje v pamäti a sleduje hodnoty počas transformácie.

Pretože je pamäť počítača alebo cloud-ovej služby obmedzená, museli sme navrhnúť riešenie, ktoré neznižuje svoju efektivitu so zvyšovaním záznamov v reálnom čase bez toho, aby došlo k pretečeniu pamäte - vytvorila sa štruktúra synchronizačnej bariéry, ktorá dokáže manažovať dáta na základe využiteľnosti. Kľúčom je názov tabuľky a hodnotami je definovaný príkaz. Táto štruktúra je znázornená na obr. 20.



Obrázok 20. Štruktúra synchronizačnej bariéry

So zvyšujúcim sa počtom záznamov môže prehľadávanie operácií degradovať (môže rýchlosť operácií exponenciálne rásť), množstvá vstupujúce do systému v reálnom čase už nemôžu ovplyvniť údaje v transformačnom procese a sú buď vymazané z databázy alebo uložené v štruktúre, ktorá prevádzkuje operácie vstupujúce do systému v reálnom čase, po celej transformácii.

Operácia aktualizovania zobrazená na obr. 20 v spodnej pravej časti spôsobí aktualizáciu tabuľky zamestnanca (*employee*). To znamená, že databáza Redis vyhľadá všetky záznamy v transformačnom procese ovplyvnené touto hodnotou a nahradí ich aktualizovanými údajmi. Predpokladajme, že proces transformácie už vykonal úplnú zmenu údajov z konkrétnej tabuľky. V takom prípade sa táto hodnota odstráni z databázy Redis, aby sa ušetrila pamäť a nedegradovalo sa tým vyhľadávanie.

Vymazanie operácie zobrazené na obr. 20 je v prvom prípade založené na rovnakom princípe. Požiadavka v reálnom čase prišla do systému, keď sa proces transformácie údajov zmenil z jednej dátovej štruktúry na druhú. Je potrebné odstrániť tabuľku plat (*salary*). V takom prípade sa tabuľka plat (*salary*) odstráni a všetky záznamy, ktoré sa mali v procese transformovať a vložiť do cieľovej databázy sú vymazané spolu s ňou.

5.5.6 Experimenty potvrdzujúce efektivitu našej paralelnej metódy

Primárnym účelom synchronizačnej bariéry je skrátenie času potrebného na úplnú zmenu údajov z konkrétneho typu na iný. V našom prípade ide o recipročnú migráciu údajov z relačnej databázy do nerelačnej databázy. S ohľadom na tento účel bola vytvorená dátová štruktúra zobrazená na obr. 17. Hodnoty sa do tabuľky postupne vkladali v tomto poradí: *department, role, employee, salary title, dep_emp, employee_has_role*.

Tieto hodnoty boli vygenerované pomocou webovej stránky *https://www.generatedata.com/.* Táto stránka nám pomohla zadefinovať názov atribútu a jeho dátový typ. Po vytvorení celkovej štruktúry sme definovali počet záznamov, ktoré sa majú generovať. V dôsledku generovania sme pre Sql definovali názov databázy a názvy tabuliek. Výstupom je tabuľka oddelenia (*department*), ktorá je zobrazená nižšie.

```
DROP TABLE department;
```

```
CREATE TABLE department (
            id mediumint(8) unsigned NOT NULL auto_increment,
            id_department mediumint,
            dep_name varchar(255) default NULL,
            PRIMARY KEY (id)
) AUTO_INCREMENT=1;
```

```
INSERT INTO department (id_department, dep_name) VALUES (1,"Accounting");
INSERT INTO department (id_department, dep_name) VALUES (11,"Legal Department");
INSERT INTO department (id_department, dep_name) VALUES (21,"Quality Assurance");
INSERT INTO department (id_department, dep_name) VALUES (31,"Public Relations");
INSERT INTO department (id_department, dep_name) VALUES (41,"Public Relations");
INSERT INTO department (id_department, dep_name) VALUES (51,"Accounting");
INSERT INTO department (id_department, dep_name) VALUES (61,"Public Relations");
INSERT INTO department (id_department, dep_name)  VALUES (71,"Customer Relations");
INSERT INTO department (id_department, dep_name) VALUES (81,"Legal Department");
INSERT INTO department (id_department, dep_name) VALUES (91,"Media Relations");
```

Nasledovným spôsobom sme postupovali pre každú tabuľku.

Po naplnení tabuliek s hodnotami sme zahájili migráciu z relačnej databázy do nerelačnej. Jednalo sa vyslovene o relačnú databázu MySql a nerelačnú databázu MongoDB. V prvom kroku necháme proces transformácie vykonať zmenu údajov bez toho, aby sa údaje dostali do procesu v reálnom čase. V prvom prípade synchronizačná bariéra nerobí žiadnu úlohu. Úlohou bariéry je manipulovať s údajmi vstupujúcimi do systému v reálnom čase a pomocou týchto údajov upravovať aktuálne sa transformujúce údaje. Táto hodnota je zobrazená na obr. 20 úplne vľavo.

V druhom kroku sme náhodne spustili štyri operácie, išlo o dve operácie aktualizácie a dve operácie odstránenia. Tieto operácie sú zobrazené nižšie.

```
update eployee set first_name = "Eric" where last_name = "Hope";
```

```
update department set dep_name = "Accounting" where dep_name = "Public Relations";
```

```
delete table salery;
```

```
delete from title where to_date= "17.05.2001";
```

Aby sme presnosť výsledkov otestovali, vybrali sme si cloud-ovú službu Amazon. Všetky služby, ktoré sa používali na správu modulov, boli takzvané „bezplatná vrstva“ (*free tier*). Pretože sme dáta preniesli distribuovaným spôsobom, vybrali sme si na tento účel Hadoop s konfiguráciou zobrazenou v tabuľke 5.

Tabuľka 5. Konfigurácia klastra

EC2 Instance	a1.medium vCPU: 1 MeM(GiB): 2
EMR cluster	master: 1x m3.xlarge core: 2x m4.4xlarge

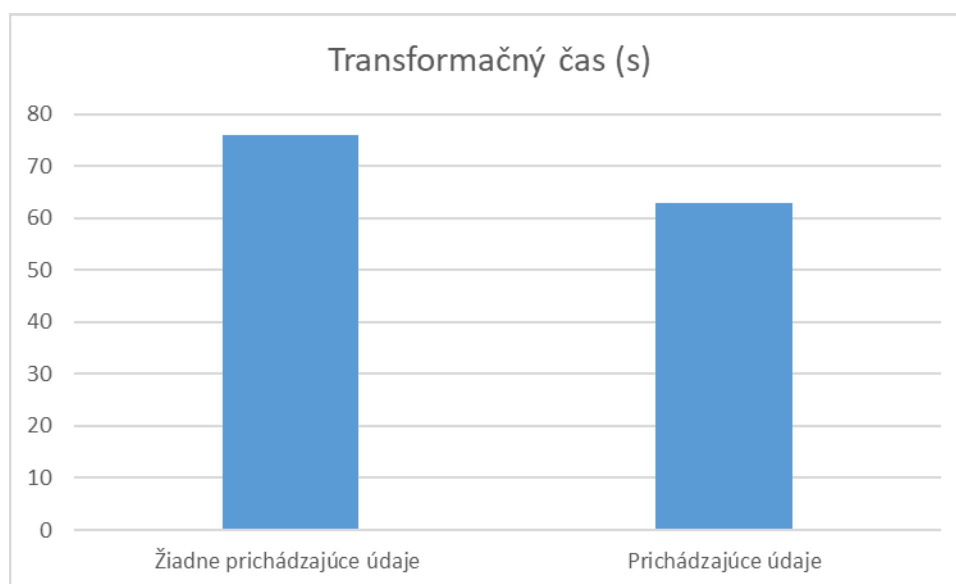
```
Host namenode
  HostName ec2-18-216-40-160.us-east 2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

```
Host datanode1
  HostName ec2-18-220-65-115.us-east-2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

```
Host datanode2
  HostName ec2-52-15-229-142.us-east-2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

```
Host datanode3
  HostName ec2-18-220-72-56.us-east-2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

Hodnoty namerané v situácii, keď došlo k migrácii údajov a údaje prichádzajúce do systému v reálnom čase, sú zaznamenané na obr. 21 vpravo.



Obrázok 21. Dĺžka trvania transformačného procesu

Pri experimentoch sa skúmalo akým spôsobom sú práve transformujúce sa dáta ovplyvnené dátami v reálnom čase. Pri experimentoch bol v prvom kroku zistený čas, ktorý je potrebný na celkový transformačný proces. Následne sme stanovili časový interval posielania príkazov do procesu, ktoré ovplyvňovali práve sa transformujúce dáta. Medzi tieto operácie patrili príkazy aktualizovania (*update*) a mazania (*delete*) záznamov. Cieľom bolo zistiť efektívnosť navrhnutej metódy za účasti dát v reálnom čase na nahromadené údaje.

Počas procesu transformácie bolo do systému odoslaných niekoľko operácií, ktoré ovplyvnili akumulované hodnoty systému a boli presunuté do cieľovej databázy. Porovnaním nášho procesu sme získali nasledujúce výsledky:

- S naším prístupom: žiadne ďalšie úpravy databázy
- Bez nášho prístupu: bolo treba vykonať 20 operácií (12 aktualizácií, sedem odstránení a jedno vloženie)

Čas potrebný na zmenu hodnôt, ktoré boli ovplyvnené prichádzajúcimi údajmi:

- Náš prístup: 4 s
- Bez nášho prístupu: 15 s

Celkový čas úpravy:

- Náš prístup: 2 minúty a 5 sekúnd
- Bez nášho prístupu: 2 minúty a 24 sekúnd (plus potreba ďalších úprav)

Na základe výsledkov, ktoré sme namerali počas našich experimentov, musíme vyvodiť jednu nevýhodu. Náš poskytnutý mechanizmus funguje efektívne v situáciách, keď databázové tabuľky nie sú navzájom veľmi závislé, to znamená, že migrácia dát z viacerých tabuliek môže prebiehať súčasne. Pri vzájomnej závislosti databázových tabuliek sa stáva nepriaznivý paralelizmus, ktorý sme implementovali do paralelných procesov. Sekvenčný prístup, ktorý je v prípade pevných vzájomných vzťahov skutočnosťou v nami navrhovanej metóde, je plytvaním času a proces migrácie v určitých momentoch sa dokonca spomalí. Pri porovnaní nami navrhovanej metódy algoritmus funguje rovnako rýchlo a efektívne.

Ako je vidieť z tabuľky 6, tak sledované hodnoty, ktoré sú pre nás podstatné sa oproti konvenčnej metóde pri rovnakej konfigurácii zlepšili takmer v každom spektre. Nami navrhnutá metóda umožnila pridať do procesu vlastnosti, akými sú umožnenie paralelného spracovania, ovplyvňovanie dát a samozrejme redukciu dodatočnej úpravy.

Aj keď spomenutá metóda pracuje efektívne a v mnohých smeroch prevyšuje konvenčnú metódu, nami navrhnutá a implementovaná metóda spôsobila vyššie režijné náklady, čo môže pri obmedzených výpočtových nákladoch zohrávať dôležitú úlohu.

Nakoľko čas potrebný na spracovanie rovnakého množstva a typu údajov sa po aplikovaní našej metódy prejavil 19 sekundovou redukciou, považujeme tento princíp za dostatočne efektívny aj za cenu zvýšených režijných nákladov.

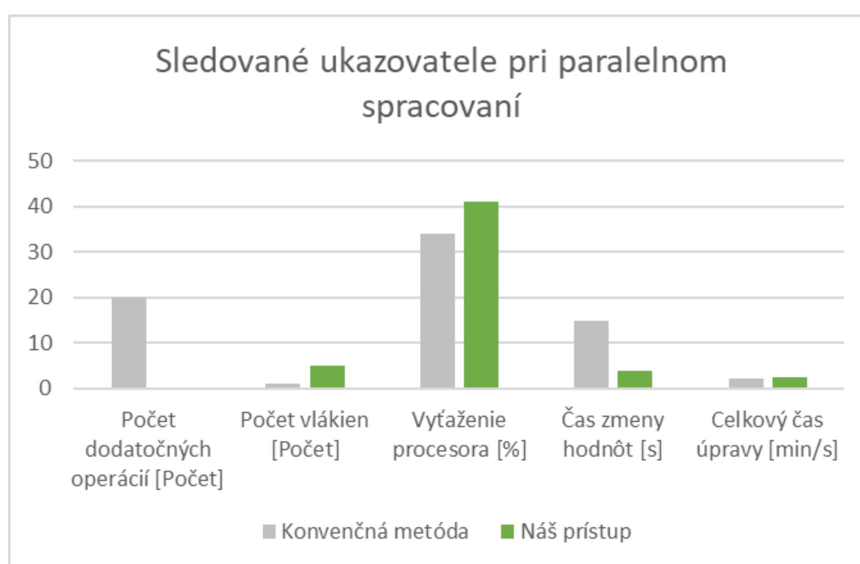
Tabuľka 6. Porovanie sledovaných hodnôt medzi konvenčnou metódou a naším prístupom

Číslo vlastnosti	Sledovaná vlastnosť	Konvenčná metóda	Náš prístup
1	Sekvenčné spracovanie [Áno/Nie]	Áno	Áno
2	Paralelné spracovanie [Áno/Nie]	Nie	Áno
3	Ovplyvňovanie [Áno/Nie]	Nie	Áno
4	Dodatočná úprava [Áno/Nie]	Áno	Nie
5	Počet dodatočných operácií [Počet]	20	0
6	Počet vlákien [Počet]	1	2 - 5
7	Vyťaženie procesora [%]	34	41
8	Čas zmeny hodnôt [s]	15	4
9	Celkový čas úpravy [min/s]	2 minúty a 24 s	2 minúty a 5 s

Ako je vidieť z grafu na obr. 22, tak grafické zobrazenie výsledkov ukazuje jasný trend. Zelenou farbou sú zobrazené výsledky po implementovaní nášho prístupu a šedou sú zobrazené výsledky s použitím konvenčnej metódy.

Okrem sledovaného ukazovateľa „vyťaženie procesora“ sú sledované hodnoty v porovnaní s hodnotami dosiahnutými konvenčnou metódou vždy efektívnejšie či už sa jedná o jednotky v percentách, sekundách a ďalšej úpravy. So zvýšeným vyťažením procesora sme od začiatku návrhu počítali, nakoľko procesor musí spravovať väčší počet vlákien ako pri konvenčnej metóde.

Po skončení procesu sme získali uspokojivý výsledok, ktorý bol v prepočte o 19 sekúnd kratší ako za použitia konvenčnej metódy a nebola potrebná žiadna dodatočná úprava.



Obrázok 22. Porovnanie sledovaných hodnôt medzi konvenčnou metódou a naším prístupom

Pri spustení veľkého množstva procesov sa nám v malom množstve, približne pri 12 pokusoch z 1 000 stalo, že nám proces prestal pracovať a migračný proces jednoducho skončil, nebol dokončený.

Pri malom množstve záznamov v opätovnom spustení procesu nevidíme veľký problém, pretože proces, ktorý trvá pár sekúnd, prípadne niekoľko minút môžeme ľahko spustiť znovu. Problém nastáva vtedy, ak proces trvá rádovo hodiny. V tom prípade je nutné vymazať veľké množstvo záznamov a spustiť proces odznovu.

Na základe výskytu tohto problému sa domnievame, že spustený proces je nutné podrobiť istému dohľadu nad dátami, ktorý zabezpečí vrátenie sa do bodu zlyhania. Tým pádom nebude nutné opätovné spustenie procesu a proces bude fungovať od bodu zlyhania.

5.6 Vytváranie verzií dát počas migračných procesov v cloud-ovom prostredí

Na základe nedostatku, ktorý vyplynul z predchádzajúcej kapitoly sme usúdili, že spomenutý problém je nevyhnutné riešiť pri migrácií veľkého množstva dát.

Nakoľko distribuované spracovanie slúži v mnohých prípadoch na správu veľkých dát (*Big Data*) je pre nás nevyhnutné, aby sa proces v prípade poruchy alebo výpadku systému vrátil do bodu zlyhania.

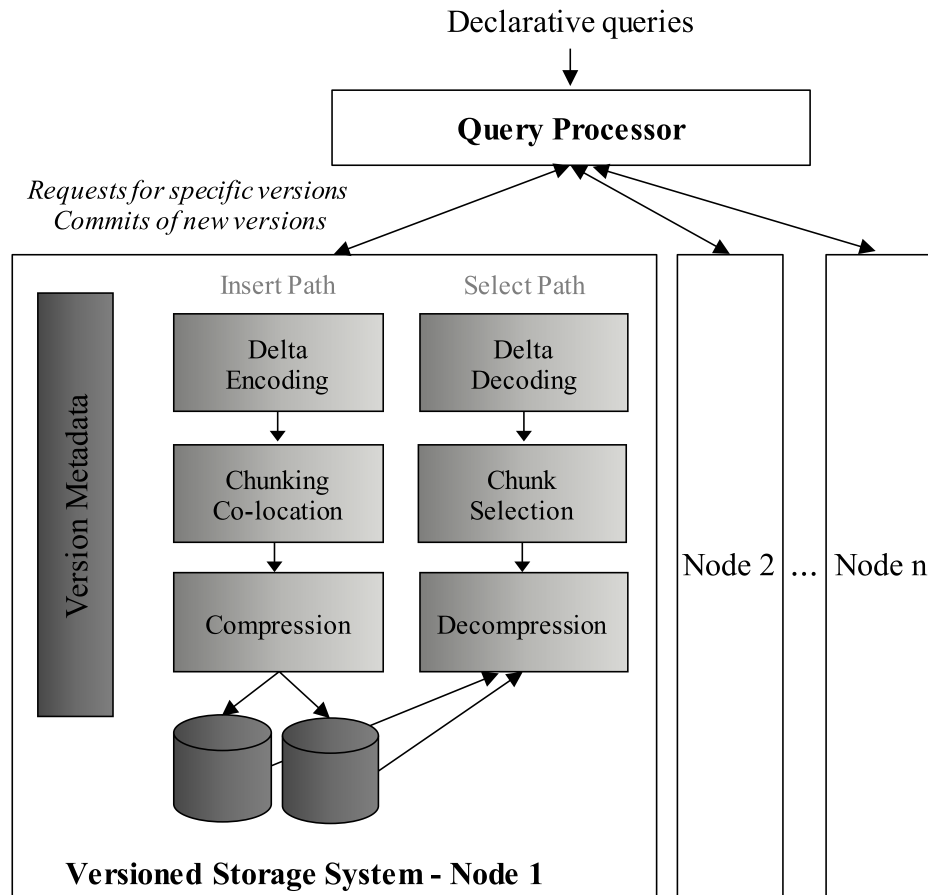
Na základe spomenutých nedostatkov a možnosti výpadku systému sme si stanovili pri skúmaní tejto problematiky nasledujúce ciele:

- Redukovať opätovné spustenie toho istého procesu v prípade zlyhania
- Aplikovať vytváranie verzií dát

Nakoľko nie sme prví výskumníci, ktorí sa danou problematikou výskytu problému zaoberali, tak v nasledujúcej podkapitole preskúmame už existujúce riešenia.

5.6.1 Existujúce riešenia venujúce sa problematike vytvárania verzií

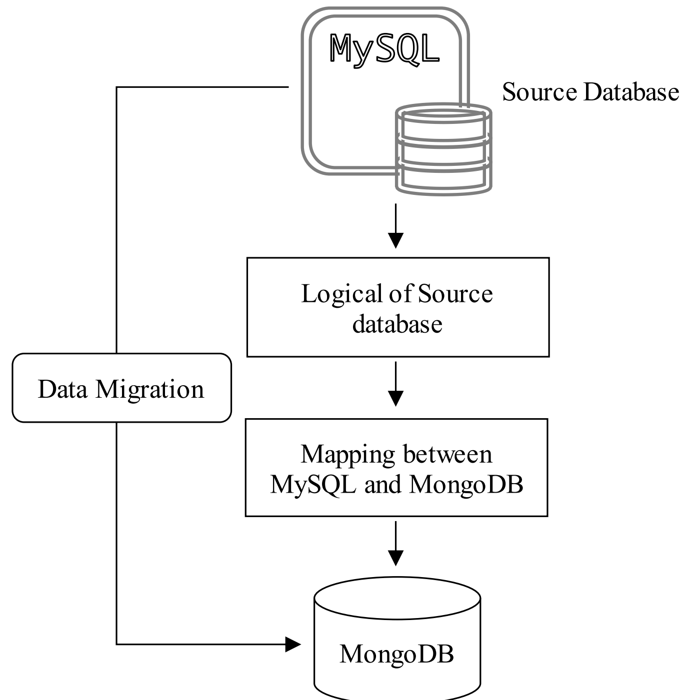
Množstvo vedcov sa zaoberalo možnosťou výskytu chyby počas transformačného procesu. Medzi prvou takto navrhnutou architektúrou je najskôr transformácia schémy, ktorá pevne drží štruktúru a následne manipuluje s dátami. Potom sa výskumníci pokúsili „verzionovať“ nie priamo dáta, ale celý transformačný proces. [81] Tento proces je zobrazený na obr. 23.



Obrázok 23. Architektúra verzionovania, ktorá zobrazuje kroky na vloženie novej verzie a výber verzie

V našom prípade je tento proces neefektívny, pokiaľ ide o aspekt a niekoľko architektúr so značným množstvom údajov.

V prípade verzionovania dát sme sa stretli aj s riešením, ktoré fungovalo nasledovne. Všetky dáta spracovávajú transformáciu dát z relačnej databázy MySql na nerelačnú databázu MongoDB jedným procesom. Proces fungoval na princípe transakcie, teda v prípade konkrétnej chyby počas transformácie došlo k vráteniu stavu pred spustením transformácie a neuložením spracovaných údajov. Potom sa všetky údaje automaticky odstránili a bolo potrebné začať odznova. Tento princíp funguje na základe rozdelenia na logické prvky a špecifických pravidiel mapovania, ktorých výstup je uložený v nerelačnej databáze MongoDB. Táto architektúra je zobrazená na obr. 24.



Obrázok 24. Transformačná architektúra

Tieto postupy efektívne neriešia dva prípady.

- V prvom prípade, keď je potrebné vrátiť sa ku konkrétnym verziám údajov, ktoré je možné časom zmeniť, pridať alebo odstrániť.
- Druhý prípad súvisí s rastúcim objemom údajov, respektíve s veľkosťou údajov potrebných na efektívny pokrok počas tejto transformácie.

Nakoľko sa nám javilo verzionovanie celého systému za plytvanie diskového priestoru, rozhodli sme sa implementovať odlišný spôsob, ktorý bude rovnako prípadne viac efektívny, ale hlavne bude redukovať množstvo priestoru potrebného na vrátenie sa k bodu zlyhania spusteného procesu.

5.6.2 Dôvod skúmania verzionovania údajov

Bezpečnosť pri transformovaní dát je pri každom výskume braná do úvahy. Pri transformačnom procese, v ktorom dochádza k masívnemu presunu, respektíve k úprave dát zo zdrojovej dátovej štruktúry na cieľovú je potrebné zabezpečiť poruchu, resp. stav, do ktorého sa musí aplikácia vrátiť.

Pri zanedbaní tohto procesu sa musia údaje, ktoré už boli zmenené vymazať alebo upraviť tak, aby spustený proces fungoval bez opätovného transformovania rovnakých dát. Týmto spôsobom by dochádzalo k zvyšujúcej degradácií procesu, predlžoval by sa čas potrebný na zmenu údajov a narastalo by vyťaženie servera. Spomenutým problémom sa zaoberali viacerí výskumníci, ktorý napríklad v článku [81] navrhli namiesto verzionovania dát verzionovať celý proces presunu informácií.

Podľa nášho názoru je tento proces v mnohých ohľadoch neefektívny, a to z dvoch dôvodov. Prvým dôvodom je ukladanie všetkých procesov. V prípade poruchy resp. zlyhania procesu budú uložené procesy niekoľkokrát. Druhým dôvodom je kontrola zmeny dát, čo znamená, že v prípade zmeny určitých dát tento proces neodzrkadlí zmeny. Preto sme navrhli a aplikovali spôsob, ktorý je popísaný nižšie.

5.6.3 Nami vytvorené riešenie problematiky verzionovania dát

Hlavnou myšlienkou je vytvorenie verzionovacieho systému schopného efektívneho návratu do bodu možnej chyby alebo práca s rôznymi dátovými verziami uloženými v nerelačnej databáze MongoDB.

Proces transformácie, pri ktorom zmena údajov z relačnej databázy MySql na MongoDB s väčším počtom tabuliek a údajov, môže trvať niekoľko minút až niekoľko hodín. V prípade chyby, ktorá sa vyskytne počas procesu transformácie, mohlo byť potrebné vymazať všetky hodnoty z nerelačnej databázy a podprogramov výhod a následne spustiť tento proces. Vyvinuli sme riešenie schopné pracovať podobným spôsobom ako verzionovacie systémami Github, BitBucket alebo Gitlab. Spomenuté systémy pracujú na porovnaní konkrétnych verzií zdrojového kódu s lokálnym úložiskom umiestneným v počítači a úložiskom na serveri obsahujúcom zdrojový kód.

Pokiaľ ide o túto logiku, vytvorili sme bránu „verzie“. Brána funguje na nasledujúcom princípe. Algoritmus zisťuje, či sú v zdrojovej nerelačnej databáze určité špecifické hodnoty. V prípade prázdnej databázy sa hodnoty ukladajú automaticky a nie je potrebné vykonať nijakú opravu údajov.

V opačnom prípade je potrebné sledovať hodnoty objektov už umiestnených v databáze. Je užitočné sledovať hodnoty objektov v kolekcii, ale hodnoty objektov je možné priebežne meniť, mazať jednotlivé objekty alebo pridávať ďalšie atribúty.

Na základe týchto problémov sme použili *The Document Versioning Pattern* priamo v nerelačnej databáze MongoDB[4].

Model sa zameriava na problém vyžadujúci neustálu revíziu niektorých dokumentov MongoDB namiesto zavedenia druhého riadiaceho systému. Aby sme to dosiahli, do každého dokumentu pridáme políčko, ktoré nám umožňuje sledovať verziu dokumentu. Databáza bude mať dve zbierky: jednu obsahujúcu najnovšie (a najčastejšie kladené otázky) a druhú obsahujúcu všetky kontroly údajov.

Vzor spravovania verzií dokumentu robí určité úkony a pristupuje k údajom na základe špecifikácií prístupu.

Každý dokument neobsahuje veľké množstvo verzií a je založený na princípe:

- Väčšina požiadaviek, ktoré boli vykonané, sa vykonáva v najnovšej verzii dokumentu.
- Tento účel je veľmi vhodný, pretože najnovšie údaje sa vždy vyberú z databázy a s historickými údajmi sa bude manipulovať iba vo verzii systému.

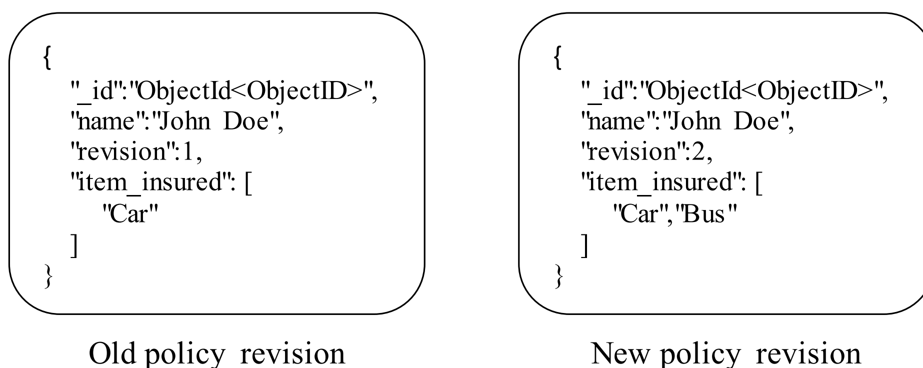
Problém súvisiaci s vytváraním verzií je kombinovaný s meniacim sa objektom, upravenými atribútmi a vymazanými hodnotami. V relačnej tabuľke počas transformácie je objekt, ktorý bol zmenený na nerelačných databázových objektoch, porovnaný. Počas chyby procesu transformácie, keď bolo možné vykonať opakovanie začiatku procesu, nemohlo dôjsť k duplikovanému vloženiu. Vloženie tohto procesu bolo možné preskočiť. Je potrebné si uvedomiť, že tento objekt sa časom mení, a tak, ak dôjde k spusteniu rovnakého procesu po určitom čase, tieto objekty nebudú rovnaké a objekt bude vložený znova. Pokiaľ ide o tento problém, rozhodli sme sa použiť vzor riadenia verzií dokumentu. Uvidíme to na príklade.

Majme objekt, ktorý je zobrazený nižšie:

```
{
"_id":"ObjectId<ObjectId>",
"name":"John Doe",
"revision":1,
"item_insured":[
"Car"
]
}
```

[4] Problematiku verzionovania dát sme prezentovali a diskutovali na IEEE konferencii ICETA na Slovensku (Vysoké Tatry) - 11. - 12. novembra 2021.

V prípade zmeny atribútu dôjde k revízii objektu, do atribútu *item_insured* sa pridá nová hodnota. Tento princíp činnosti je zobrazený na obr. 25.



Obrázok 25. Ukážka revízie

Časovým aspektom, ktorým sme sa pri výskume zaoberali, bolo vytvorenie riešenia, ktoré je časovo efektívne a v prípade zmien v raste dát bude kombinované s tým, že k významnej segregácii spomínaného riešenia nedôjde. S ohľadom na tento nedostatok sme vytvorili metódu objektového porovnávania.

5.6.3.1 Objektové porovnávanie (Object comparator)

Objektové porovnávanie nám pomáha pri skenovaní záznamov medzi pôvodnou relačnou databázou a vytvorenými objektmi v nerelačnej databáze. Druhou úlohou tejto metódy je rozhodnúť, či daný objekt musí alebo nemusí byť vytvorený, a teda či sa objavia duplicity v nerelačných databázach.

Pokiaľ ide o toto riešenie, nastáva situácia súvisiaca s hodnotou *_id* v nerelačnej databáze. Je nutné, aby hodnota *_id* bola v každom okamihu jedinečná. Mohlo by to spôsobiť dva alebo tri voliteľné objekty, ktorých hodnota by mohla mať rovnakú hodnotu ako primárny kľúč.

Nutnosť držania hodnoty je samozrejmosťou a nevyhnutnosťou pri rýchlom a efektívnom zisťovaní danej hodnoty. Z tohto dôvodu je nevyhnutné, aby sme vždy vedeli, ktorú hodnotu sme vytvorili v nerelačnej databáze MongoDB. Vytvorili sme spúšťač, ktorý nám na tieto účely vráti hodnotu *_id*.

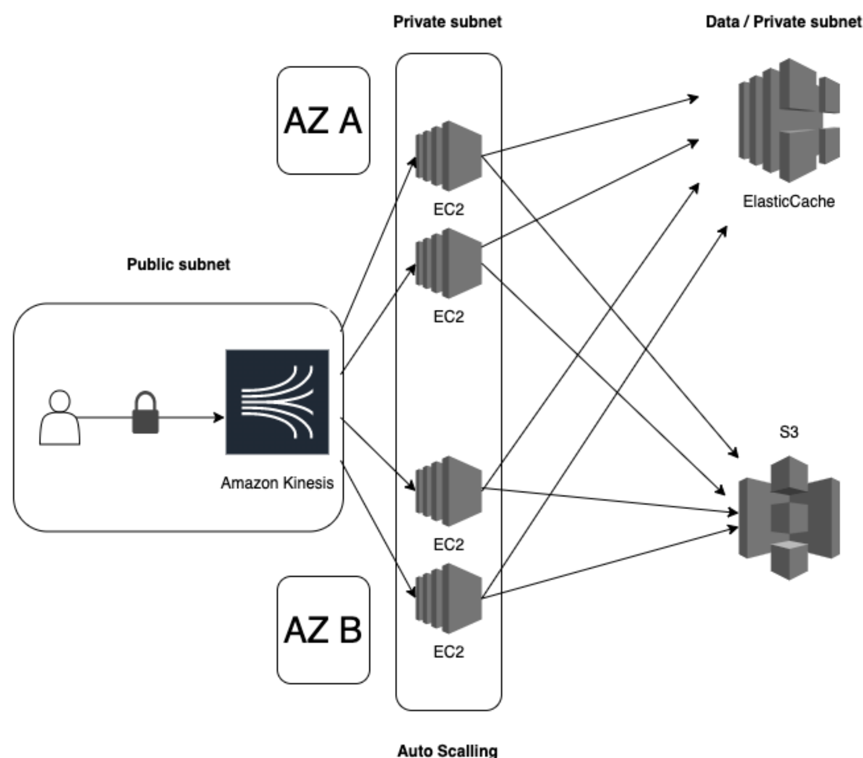
Musíme vložiť záznam typu produktu (*product*) do databázy s atribútmi názov (*name*), množstvo (*quantity*), kategória (*category*) a recenzie (*review*).

```
const newProduct = {
  "name": "Plastic Bricks",
  "quantity": 10,
  "category": "toys",
  "reviews": [{ "username": "legolover", "comment": "These are awesome!" }]
};
```

Počas vkladania sme vytvorili spôsob, ktorý nám vždy vráti primárny kľúč. Tento spôsob vyzerá nasledovne:

```
itemsCollection.insertOne(newProduct)
  .then(result => console.log(`Successfully inserted item with _id: ${result.insertedId}`))
  .catch(err => console.error(`Failed to insert item: ${err}`))
```

V jeho fáze už vieme, akú hodnotu mal záznam vložený do nerelačnej databázy. Musíme zistiť hodnotu primárneho procesu kritického zadávania záznamu, ktorý bol zmenený na objekt *newProduct.* Táto hodnota sa získa automaticky po začiatku transformačného procesu. V našej architektúre je uložený v S3 (*Simple Storage Services*) a hodnota primárneho kľúča je uložená v nerelačnej databáze *Redis*. Proces spustenia a uloženia hodnôt do databázy Redis a S3 je zobrazený na obr. 26.



Obrázok 26. Proces plnenia údajov

Proces začína spustením transformačného procesu používateľom. Následne dáta prechádzajú cez Amazon Kinesis do rôznych zón dostupnosti a tie pomocou rôznych výpočtových jednotiek EC2 presúvajú informácie do *S3* a *ElasticCache*.

Po spustení procesu transformácie a uložení údajov do nerelačnej databázy je štruktúra nerelačnej databázy v *ElasticCache* nasledovná.

Tabuľka 7. Primárny kľúč v Sql a primárny kľúč v NoSql

SQL	**NoSQL**
234d3nc3y2un34fn7	Xj234nx83nxf837fnx
X3f23fcf3324v34v3	323fx234f245v2tc2x2
2x3fx23f23v432vtw	23vr23rx234x2r4x2r4

Hodnota stĺpca vľavo v tabuľke 7 predstavuje hodnotu primárneho kľúča pre relačnú databázu pre tabuľku produkt (*product*). Tým sa k nej pridá hodnota primárneho kľúča v nerelačnej databáze MongoDB, ktorá je uvedená v tabuľke 7 vpravo.

V prípade rovnakej transformácie dôjde k porovnaniu objektu spúšťajúceho dáta na základe primárneho kľúča. Hodnotu primárneho kľúča získame z procesu zadávania objektu. Pokiaľ ide o tento atribút zisťujeme, či táto hodnota už existuje v databáze Redis. Rýchlosť vyhľadávania v databáze je rýchla, pretože všetko sa deje v pamäti.

Hodnota, ktorú nebolo možné nájsť v databáze Redis, označuje záznam s týmto primárnym kľúčom v procese transformácie, sa zatiaľ nestala a je potrebné ju poslať ďalej do tohto procesu.

V prípade výskytu hodnoty v databáze Redis bude nasledovať jej vyhľadanie. Najskôr nájdeme záznam hodnoty v ľavom stĺpci zodpovedajúci hodnote nájdenej v pravom stĺpci. Nasleduje vyhľadávanie revízia (*verzia objektu*) daného objektu. Počas testu nastala v mnohých prípadoch situácia, keď bola prvá revízia objektu rovnaká so záznamom relačnej databázy. Ak dôjde k zhode objektov, tento záznam už do transformačného procesu nevstupuje.

Objekt, ktorý nemal ďalšie kontroly a nezhodoval sa s daným objektom, bol tak pustený do ďalšej fázy transformačného procesu.

5.6.4 Experimentovanie s dátami pri verzionovaní údajov

Ako sme už spomínali, objekty môžu mať rôzne verzie, a preto bolo potrebné prehľadať všetky. Pokiaľ sa hodnoty žiadnej revízie nezhodovali s daným objektom, objekt pokračoval v transformačnom procese a po dokončení prehľadal hodnotu primárneho kľúča a hodnota kľúča priamo vytvoreného objektu bola zapísaná a zmapovaná rovnakým spôsobom ako sa zobrazuje v tabuľke 7.

Tabuľka 8. Konfigurácia klastra

EC2 Instance	a1.medium vCPU: 1 MeM(GiB): 2
EMR cluster	master: 1x m3.xlarge core: 2x m4.4xlarge

```
Host namenode
  HostName ec2-18-216-40-160.us-east 2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

```
Host datanode1
  HostName ec2-18-220-65-115.us-east-2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

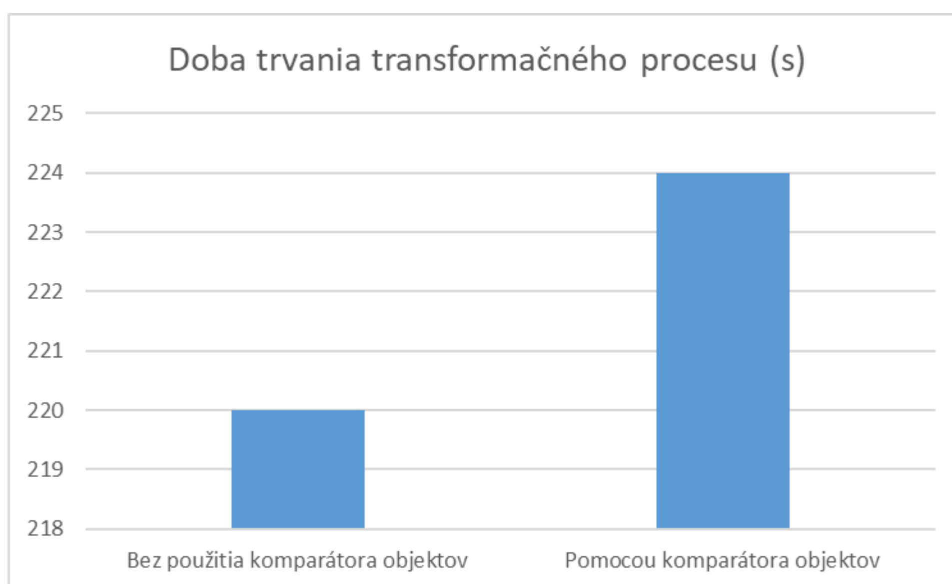
```
Host datanode2
  HostName ec2-52-15-229-142.us-east-2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

```
Host datanode3
  HostName ec2-18-220-72-56.us-east-2.compute.amazonaws.com
  User ubuntu
  IdentityFile ~/.ssh/MyLab_Machine.pem
```

Vytvorené metódy a postupy boli testované na základnej infraštruktúre, ktorú služba Amazon poskytuje. Najskôr sme vytvorili relačnú databázu a vložili sme do nej nové záznamy. Použité hodnoty sú dostupné zadarmo a sú k dispozícii na tomto linku: *https://github.com/awesomedata/awesome-public-datasets#socialnetworks.*

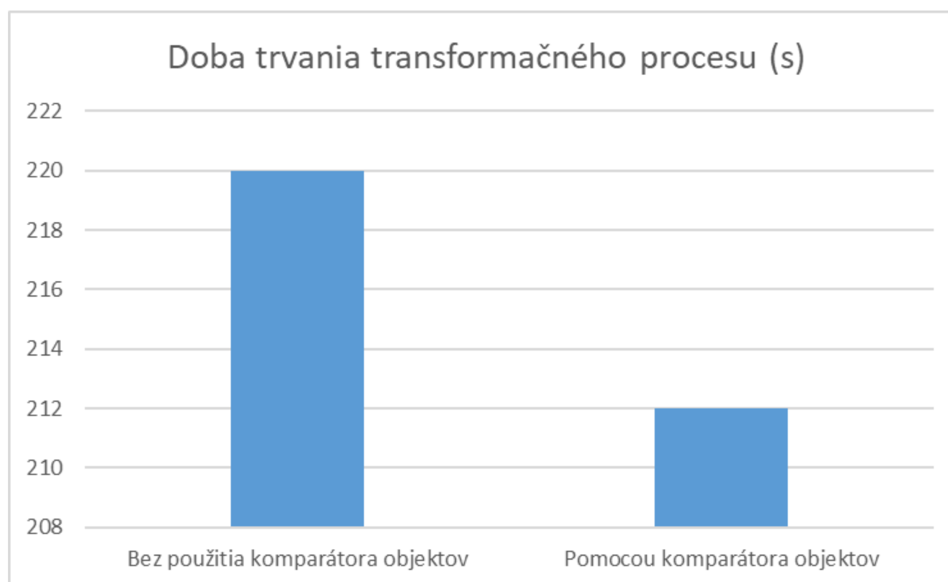
Hodnoty zaznamenávajú štruktúru sietí „priateľstva“ na Facebooku na sto amerických vysokých školách a univerzitách v jednom okamihu a štúdia skúmala úlohy atribútov používateľov - pohlavie, ročník, hlavné štúdium, stredná škola a bydlisko - v týchto inštitúciách.

Následne bol vytvorený experiment, ktorý sledoval účinnosť našich nových metód. Sledovali sme čas potrebný na vloženie záznamov do nerelačnej databázy. MongoDB bez toho, že záznamy už boli v databáze výsledkov s časom potrebným na vykonanie procesu transformácie, keď v konečnej nerelačnej databáze už existovalo 10 000 záznamov. Výsledky času boli zaznamenané a zobrazené na obr. 27.



Obrázok 27. Odlišný čas medzi použitou metódou bez údajov v databáze

Ako vidíme na obr. 27, hodnoty potrebné počas transformácie údajov, keď v cieľovej databáze MongoDB neexistovali žiadne hodnoty, negatívne ovplyvnili čas potrebný na úplnú zmenu údajov počas transformácie údajov z relačnej databázy MySql na MongoDB. Tento problém súvisel s neustálou kontrolou prichádzajúcich záznamov s existujúcimi záznamami v databáze. Aj keď je algoritmus efektívne navrhnutý, kontrola objektov spôsobila oneskorenie oproti situácii, keď sa dáta nekontrolovali. Použitie nášho riešenia v situácii, keď hodnoty už v databáze existovali, malo opačný efekt. Išlo konkrétne o 10 000 záznamov. Rozdiel, ktorý sme spomenuli, je zobrazený na obr. 28.



Obrázok 28. Odlišný čas medzi použitou metódou a údajmi v databáze

Pri porovnaní výsledkov na obr. 27 a obr. 28 môžeme vidieť, kedy sú údaje v databáze a kedy nie sú v databáze. Časový rozdiel pri databáze obsahujúcej údaje sa transformačný proces pri výskute chyby za pomocou našich postupov sa znížil o 8 sekúnd, čo bol výrazný pohyb s malým objemom údajov. Naopak, ak bolo potrebné neustále kontrolovať údaje v databáze Redis, čas sa znížil o 4 sekundy.

Hodnoty, ktoré sme dosiahli v experimentálnej práci, demonštrujú efektívnosť našej metódy. Metóda kontroluje údaje, ktoré vstupujú do systému, a porovnáva ich s hodnotami, ktoré sa už v systéme nachádzajú. Výsledkom tejto metódy je povolenie alebo nepovolenie vstupu údajov do systému. Uvedená kontrola s minimálnym objemom údajov nie je praktická pri vyhľadávaní, pri porovnávaní prichádzajúcich údajov a pri systémoch so zvyšujúcim sa objemom údajov, ktoré prichádzajú do systému. V situácií, kde sa rovnaké údaje už v systéme nachádzajú dochádza k zvyšeniu efektivity nami navhrnutej metódy, čo sa preukázalo aj pri experimentálnej činnosti.

Ako je vidieť z tabuľky 9, tak sledované hodnoty, ktoré sú pre nás podstatné sa oproti konvenčnej metóde pri rovnakej konfigurácii, rovnakom počte procesov a verzií zlepšili takmer v každom spektre. Nami navrhnutá metóda umožnila radikálne znížiť počet úložného miesta potrebného na spracovanie jednotlivých verzií.

Aj keď spomenutá metóda pracuje efektívne a v mnohých smeroch prevyšuje konvenčnú metódu, nami navrhnutá a implementovaná metóda spôsobila vyššie režijné náklady, čo môže pri obmedzených výpočtových nákladoch zohrávať dôležitú úlohu.

Nakoľko čas potrebný na spracovanie rovnakého množstva a typov údajov sa po aplikovaní našej metódy prejavil zlepšením o 8 sekúnd, považujeme tento princíp za dostatočne efektívny aj za cenu zvýšených režijných nákladov.

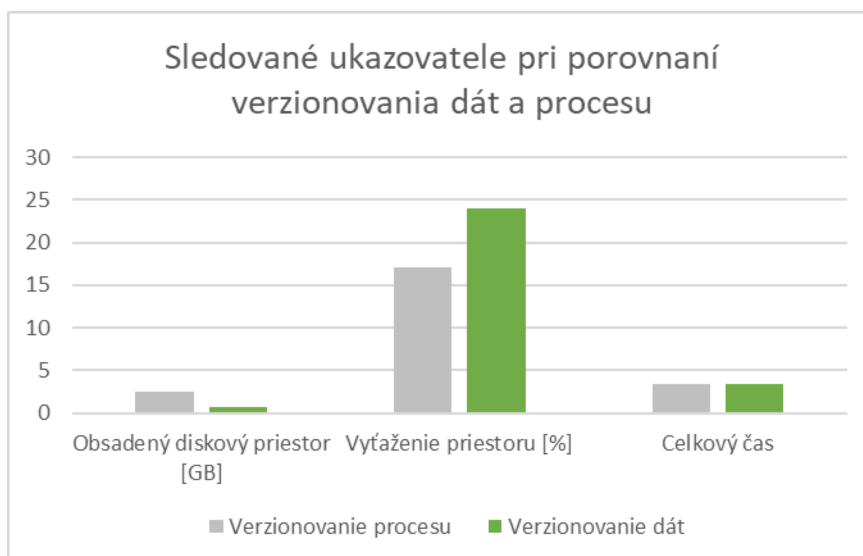
Tabuľka 9. Porovanie sledovaných hodnôt medzi verzionovaním procesu a verzionovaním dát

Číslo vlastnosti	Sledovaná vlastnosť	Verzionovanie procesu	Verzionovanie dát
1	Obsadený diskový priestor [GB]	2,5	0,7
2	Vyťaženie priestoru [%]	17	24
3	Vrátenie sa k bodu zlyhania [Áno/Nie]	Nie	Áno
4	Kontrola objektov [Áno/Nie]	Nie	Áno
5	Kontrola procesu [Áno/Nie]	Áno	Nie
6	Počet verzií	10	10
7	Celkový čas	3 minúty a 40 s	3 minúty a 32 s

Ako je vidieť z grafu na obr. 29, tak grafické zobrazenie výsledkov ukazuje jasný trend. Zelenou farbou sú zobrazené výsledky po implementovaní nášho prístupu a šedou sú zobrazené výsledky s použitím konvenčnej metódy.

Okrem sledovaného ukazovateľa „vyťaženie procesora" sú vždy sledované hodnoty v porovnaní s hodnotami dosiahnutými metódou verzionovania procesu vždy efektívnejšie či už sa jedná o jednotky v percentách, sekundách a ďalšej úpravy. So zvýšeným vyťažením procesora sme od začiatku návrhu rátali, nakoľko proces musí byť zabezpečený pri každej kontrole minimálne 2 spustenými vláknami, pri ktorom sa kontrolujú vezie objektu a následne ďalšie vlákno proces vykonáva.

Po skončení procesu sme získali uspokojivý výsledok, ktorý bol v prepočte o 8 sekúnd kratší ako za použitia metódy verzionovania procesu.



Obrázok 29. Grafické porovnanie výsledkov dosiahnutých pri verzionovaní procesu a dát

5.6.5 Zhrnutie metódy verzionovania údajov

Prístup, ktorý sme definovali pri návrhu riešenia, jasne ukazuje našu výhodu v tomto smere a naznačuje trend, v ktorom budeme v ďalšej časti výskumu pokračovať. Možnosť verziovania údajov spôsobil zníženie času potrebného na zmenu skutočných údajov v procese transformácie. Došlo k zmene údajov z relačnej databázy MySql na nerelačnú databázu MongoDB.

Na ukladanie zmien objektov v kolekciách sme použili *The Document Versioning Pattern*, ktorý zachováva autentickú verziu objektu a dokončuje zmeny objektov počas celého procesu existencie konkrétneho objektu. Táto metóda pomáha objektu porovnávať a zisťovať, či konkrétny objekt alebo jeho predchádzajúca verzia už boli rovnaké ako objekty, ktoré sa majú znova vložiť do databázy.

Hlavná myšlienka verzovacieho systému bola definovaná v komparátore objektov metódy. Na základe jedinečných hodnôt, ktoré vstupujú do procesu, dochádza k porovnaniu hodnôt s už existujúcimi hodnotami v systéme. V prípade situácie, ak je v systéme už objekt s rovnakou unikátnou hodnotou dôjde k porovnaniu ich verzií. Ak neexistuje rovnaká verzia objektu, aký je už v systéme v porovnaní s prichádzajúcim objektom, tak dochádza k pridaniu verzie objektu v systéme. V prípade, že sa objekt s rovnakou verziou nachádza v systéme, tak je objekt následne vynechaný z procesu a nasleduje porovannie ďaľších objektov.

Experimenty, ktoré sme navrhli, signalizujú, že keď sú údaje v databáze, komparátor objektov metódy je výhodný, pokiaľ ide o časovú redukciu transformačného procesu a zabráneniu duplikácií v nerelačnej databáze.

Pri rôznych pokusoch sme zistili jeden závažný nedostatok, ktorý je spojený s veľkým počtom verzií jednotlivých objektov. V prípade, ak sme spustili proces s napr. 10 000 objektmi a každý jeden objekt obsahuje 10 verzií, čo celkovo predstavuje maximálne 100 000 kontrol objektov, tak efektivita nášho riešenia degraduje. Pri hlbšom testovaní sa nám osvedčil počet verzií nastavených na hodnotu 5, kedy je efektivita nášho riešenia stále výhodnejšia oproti konvečným metódam.

Pri nasadení aplikácie do produkcie nám aplikácia fungovala správne a efektívne bez výpadkov a pri zámernom narušení systému aplikácia fungovala podľa predpokladov. Počas testovania sme si však všimli, že sa k dátam a aplikácii snažili získať prístup aj aplikácie tretích strán.

Na základe 3 mesačných pozorovaní sme zistili, že došlo k získaniu a narušeniu dát, a tým aj k ich získaniu. Tento fakt nás donútil zamyslieť sa nad bezpečnostným aspektami nami navrhnutých aplikácií.

Na základe výskytu tohto problému sa domnievame, že je do procesu zabezpečenia dát potrebné aplikovať ďalšie bezpečnostné prvky. Tým pádom sa zvýši bezpečnosť a redukuje sa množstvo hrozieb.

5.7 Bezpečnosť spracovania dát v reálnom čase

Na základe zistenia z predchádzajúcej kapitoly sme si uvedomili bezpečnostné riziká, ktoré vyplývajú pri práci s veľkými údajmi a ich spracovaním.

Bezpečnostné aspekty, ktoré sú v súčasnosti veľmi rozšírenou problematikou nám pomôžu v mnohých aspektoch redukovať riziko, a tým pádom ešte viac zefektívniť proces spracovania údajov.

Na základe spomenutých nedostatkov a možnosti straty údajov, prípadne možnosti zlyhania celého systému sme si stanovili pri skúmaní tejto problematiky nasledujúce ciele:

- viacstupňová kontrola prístupu
- redukcia vkladania pomocou presúvania údajov na základe časového aspektu,
- presúvanie informácií pre zvýšenie kontroly.

Nakoľko nie sme prví a ani poslední výskumníci, ktorí sa problematikou bezpečnosti údajov zaoberali, rozhodli sme sa, že v nasledujúcej podkapitole preskúmame už existujúce riešenia.

5.7.1 Existujúce riešenia zaoberajúce sa bezpečnosťou

Databázové typy či už hovoríme o relačných alebo nerelačných databázach, majú svoje výhody a nevýhody. Pri výbere tej správnej databázy zohrávajú vlastnosti jedného alebo druhého typu kľúčové body. V súčasnosti veľká časť firiem prechádza na cloud-ové riešenia. Tento nedávny pokrok v oblasti cloud computingu a distribuovaných webových aplikácií vyvolal potrebu ukladania veľkého množstva údajov do distribuovaných databáz, ktoré poskytujú vysokú dostupnosť a škálovateľnosť. V posledných rokoch si rastúci počet spoločností osvojil rôzne typy nerelačných databáz, ktoré sa bežne označujú ako NoSql databázy, a keďže sa objavujú aplikácie, ktoré ich používajú čoraz častejšie, získavajú na trhu rozsiahly záujem.

Nové databázové systémy nie sú z definície relačné, a preto nepodporujú úplnú funkčnosť Sql. Navyše na rozdiel od relačných databáz pracujú s konzistenciou a bezpečnosťou dát pri vysokom výkone a škálovateľnosti. Pretože sa v databázach NoSql ukladajú čoraz citlivejšie údaje, bezpečnostné problémy sa stávajú čoraz viac znepokojujúcimi. V príspevku [82] výskumníci skúmajú dve najobľúbenejšie NoSql databázy (Cassandra a MongoDB) a popisujú ich hlavné bezpečnostné prvky a problémy.

Rast a penetrácia aplikácií NoSql, poháňaná gigantmi zo Silicon Valley, ako sú Facebook, Twitter, Yahoo, Google a LinkedIn, vytvoril bezprecedentnú databázovú revolúciu, aby inšpiroval menšie spoločnosti k tomu, aby sa pripojili k rozbehnutému trendu NoSql. Aj keď rozširovanie a rast týchto databáz dodáva rýchlosť a spokojnosť mnohým podnikovým oddeleniam IT, je veľmi dôležité preskúmať bezpečnostné aspekty týchto databáz novej doby. Dôvernosť, integrita a dostupnosť (CIA) sú základom ochrany údajov a súkromia. V článku [99] sa výskumníci venujú prieskumu, analýze a vyhodnoteniu vyspelosti databáz NoSql na základe vlastností CIA. Zatiaľ čo koncept CIA má pôvod v relačných databázach, je veľmi dôležité pochopiť, preskúmať a vymedziť bezpečnostné možnosti tejto databázy novej generácie z hľadiska plnenia CIA.

Bezpečnostné modely vyvinuté pre databázy sa líšia v mnohých aspektoch tým, že sa zameriavajú na rôzne vlastnosti bezpečnostného problému databázy, alebo preto, že vytvárajú odlišné predpoklady o tom, čo predstavuje bezpečnú databázu. Spomenutý predpoklad vedie k nesúrodému a neúplnému pochopeniu bezpečnostnej stratégie organizácie a sťažuje zosúladenie rôznych bezpečnostných požiadaviek. V článku [48] výskumníci skúmajú rôzne bezpečnostné problémy v databázach. Článok [48] je užitočný na plánovanie explicitných a direktívne založených bezpečnostných požiadaviek na databázu.

Pri skúmaní danej problematiky sme preskúmali štúdiu, ktorá má odlišný pohľad na vec oproti publikovaným článkom. Štúdia poskytuje viacúrovňový zabezpečený databázový systém. Modelovanie sémantiky údajov aplikačnej domény bol predmetom výskumu mnohých rokov v databázovej komunite, ale tieto snahy iba riešia vlastnosti integrity údajov. Príspevok výskumníkov v článku [98] demonštruje použitie rozšíreného dátového modelu, ktorý predstavuje integritu aj aspekty utajenia údajov.

Túto techniku modelovania je možné použiť ako nástroj na návrh databázy a čo je dôležitejšie, ako prostriedok pre doménových expertov, návrhárov databáz a pracovníkov bezpečnostnej služby, aby presne definovali bezpečnostné požiadavky pre aplikačnú doménu. Druhým prínosom v príspevku [98] je komplexná taxonómia bezpečnostnej sémantiky údajov, ktorá musí byť zachytená a pripravená implementovať viacúrovňový bezpečný automatizovaný informačný systém.

V súčasnosti populárne distribuované spracovanie dát nebolo nepovšimnuté ani pri bezpečnosti. Zatiaľ čo výskum pôvodu je v distribuovaných systémoch bežný, mnoho navrhovaných riešení sa nezaoberá bezpečnosťou systémov a zodpovednosťou za údaje uložené v týchto systémoch. V príspevku [100] výskumníci skúmajú pôvodné (prevenance) riešenia, ktoré boli navrhnuté na riešenie problémov so zabezpečením systému a zodpovednosťou za údaje v distribuovaných systémoch. Z ich prieskumu bol odvodený súbor minimálnych požiadaviek, ktoré sú potrebné na to, aby bol pôvodný systém (prevenance) účinný pri riešení týchto dvoch problémov. Na záver výskumníci identifikujú niekoľko nedostatkov v skúmaných riešeniach a predstavili ich ako výzvy, ktorými by sa mali budúci výskumníci zaoberať. Autori tvrdia, že tieto nedostatky je potrebné vyriešiť skôr, ako bude možné v budúcnosti dospieť k úplnému a spoľahlivému pôvodnému riešeniu (prevenance).

Počas rôznych útokov, či už sa jedná a webovú aplikáciu alebo priamo útok na databázu môže dôjsť k viacerých poškodeniam údajov. Hlavným aspektom väčšiny firiem je spoľahnutie sa na správnosť dát v systéme. Keď užívateľ vloží záznamy do databázy očakáva, že ich kedykoľvek v nej aj nájde. Pri útoku môže dôjsť k úprave, prípadne aj k zmazaniu záznamov. Touto problematikou sa zaoberali výskumníci v článku [64], kde skúmajú riešenia, pri ktorých sa údaje aktualizovali vždy synchrónne v presne stanovených časových bodoch, frekvencii a granularite. Výskumníci tvrdia, že je nevyhnutné vytvoriť riešenie vhodné pre konkrétny systém s cieľom dosiahnuť najlepší výkon. Odkazujú na atribútovo orientovaný dočasný model s reflexiou na technológie zoskupovania údajov. Mnohokrát existuje prípad, počas ktorého je objekt definovaný iba čiastočne alebo n-tica údajov nie je prítomná vôbec. Z týchto dôvodov musia byť nedefinované hodnoty uložené v databáze vo forme samotného času alebo atribútu vyjadrujúceho stav objektu.

Publikovaný príspevok sa zaoberá časovo orientovanými databázovými architektúrami, spravuje nedefinované hodnoty a navrhuje komplexnú systémovú klasifikáciu založenú na transakciách, prístupoch a indexoch. Zaoberajú sa technikami modelovania nedefinovaných hodnôt a pokrývajú synchronizačné procesy pomocou dátových skupín. Výskumníci navrhli riešenia pre efektívne získavanie údajov s dôrazom na nedefinované hodnoty a stavy.

V nerelačných databázach je zabezpečenie údajov zdieľaných na rôznych serveroch náročným problémom z dôvodu údajov, ktoré sa distribuujú a prenášajú cez nezabezpečenú sieť. Uskutočnil sa rozsiahly výskum mechanizmov delenia NoSql, ale nebolo zadefinované žiadne konkrétne kritérium na analýzu bezpečnosti rozdelenej architektúry. V príspevku [119] výskumníci navrhujú hodnotiace kritérium zahŕňajúce rôzne bezpečnostné prvky pre analýzu horizontálnych NoSql databáz. Poskytujú podrobný pohľad na bezpečnostné prvky ponúkané databázami NoSql a analyzujú ich s ohľadom na navrhované hodnotiace kritériá. Predložená analýza pomáha rôznym organizáciám pri výbere vhodnej a spoľahlivej databázy v súlade s ich preferenciami a bezpečnostnými požiadavkami.

Pri skúmaní bezpečnosti vedci odporučili riešiť problematiku pomocou trendovej technológie blockchain. V dokumente [90] výskumníci navrhujú využiť distribuovanú schému založenú na blockchaine - známej tiež ako verejná účtovná kniha - na vytvorenie VWN (*virtual wireless networks*), kde vlastníci primárnych bezdrôtových zdrojov (PWRO (*Primary Wireless Resource-Owner*)) prenajímajú svoje bezdrôtové zdroje (napr. časť vysokofrekvenčného spektra, infraštruktúru) mobilným virtuálnym serverom prevádzkovatelia sietí (MVNO (*Mobile Virtual Network Operators*)) využívajúci komunikáciu medzi strojmi na základe dohôd o úrovni služieb (SLA (*Service Level Agreements*)) medzi PWRO a MVNO. Navrhnutá distribuovaná schéma založená na blockchaine poskytuje bezpečnosť zúčastneným PWRO a MVNO, ako aj zabráni PWRO v nadmernom využívaní ich zdrojov (čo zastaví dvojité výdavky) a pomáha MVNO splniť požiadavky QoS ich používateľov. Na tomto rámci sa podieľa Federálna komunikačná komisia USA (FCC (*Federal Communications Commission*)) alebo podobné regulačné orgány v iných krajinách poskytovaním pokynov a predpisov o maximálnych úrovniach napájania, licenciách a geografickom pokrytí atď. To v podstate pomáha používateľom splniť požadované požiadavky QoS(*Quality-of-Service*) pri dodržaní zásad nariadenia vlády. Výkon sa hodnotí pomocou číselných výsledkov.

Ako je možné vidieť tak viaceré štúdie sa zaoberali problematikou bezpečnosti. Pri návrhu našej novej metódy sme sa inšpirovali prácou [65],v ktorej sa autor zaoberá riadením granularity dočasného systému a navrhuje model zdieľania údajov založený na spoľahlivosti, citlivosti a presnosti poskytovateľov údajov. Je predstavená nová koncepcia časovej výhody, ktorá je následne vyhodnotená v experimentálnej časti. Optimalizácia dátového toku pomocou historickej agregácie dát a obmedzenie množstva dát je kľúčovou súčasťou rozhodovania systému, zatiaľ čo čas na prenos dát je prísne obmedzený.

Pre naše účely migrácie dát sme museli vytvoriť spôsob na presun údajov z databázy do databázového skladu. K tomuto účelu sme skúmali metódu, ktorá je v súčasnosti najrozšírenejšia. Výskumníci vo svojom článku [95] popísali metódu ETL (*Extract, Transform and Loading*) ako proces ukladania údajov, ktorý migruje údaje zo zdrojovej databázy vykonaním určitých pravidiel transformácie nad extrahovanými údajmi. Tieto transformované údaje sa načítajú späť do cieľovej databázy. So vznikom veľkých dát rôzne organizácie smerujú k veľkým dátovým technológiám ako Hadoop, Hadoop Ecosystem Projects ako Hive a HBase na ukladanie svojich dát. Organizácia používa proces migrácie údajov na migráciu údajov z RDBMS na Hadoop a tieto údaje sa ďalej používajú na rôzne analytické účely. Úloha migrácie údajov však niekedy spôsobuje nezrovnalosti v dátach z rôznych dôvodov, ktoré môžu viesť k nepresnej analýze údajov. Tento príspevok hovorí o zovšeobecňujúcom rámci pre overovanie údajov medzi RDBMS a Hadoop.

Bezpečnostné hrozby a ich riešenia boli predstavené v nasledujúcej kapitole. Nakoľko zavedenie bezpečnostných opatrení redukuje riziko straty údajov rozhodli sme sa aplikovať odlišný typ bezpečnostných pravidiel ako bolo doteraz publikované inými výskumníkmi. Pridaná ďalšia kontrola vstupov, presúvanie záznamov a ďalšie kroky našej metódy majú za úlohu redukovať stratu dát, ukradnutie údajov a redukovať množstvo útokov na našu aplikáciu.

5.7.2 Dôvod skúmania bezpečnosti spracovania dát v reálnom čase

Bezpečnosť uložených, respektíve poskytnutých údajov je v súčasnosti veľmi rozšírený a dôležitý aspekt pri manipulácii s dátami. V súčasnosti sa každý deň uskutočňuje množstvo útokov v snahe získať, respektíve odcudziť dáta, a tak negatívnym spôsobom ovplyvniť správne fungovanie aplikácie alebo databázy.

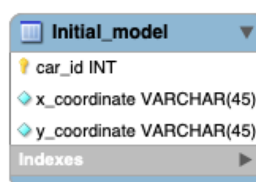
So zanedbaním bezpečnosti, respektíve s poľavením bezpečnosti pri správe údajov bolo už publikovaných mnoho článkov od viacerých výskumníkov. Pri tomto procese si viacerí výskumníci uvedomili, že so zvýšením bezpečnosti narastá aj dĺžka manipulácie s údajmi. Za zaujímavý aspekt sme považovali spravovať dáta, a následne tým redukovať stratu údajov pri rôznych útokoch alebo poškodení dát.

Na základe štúdie [65], ktorej budúci výskum chceli autori venovať návrhu presunu dát sme sa rozhodli aj my vytvoriť metódu presúvania dát z databázy do dátového skladu, a v prípade potreby z dátového skladu do databázy na základe pravidiel. Tento presun sme podmienili dodatočným bezpečnostným prvkom.

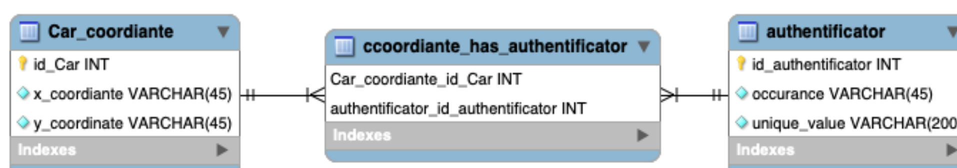
5.7.2.1 Úprava dát v reálnom čase

Automobilová doprava patrí v súčasnosti k veľmi rozšíreným a populárnym odvetviam. Viaceré spoločnosti z dôvodu bezpečnosti a ochrany tovaru monitorujú svojich zamestnancov a spomenuté monitorovanie vytvára veľký objem údajov.

Dáta, ktoré prichádzajú od každého auta do systému sú vždy upravené. Každé auto, ktoré pomocou sledovača odošle informácie o pozícii má len 3 hodnoty. Prvou hodnotou je unikátna hodnota auta, druhou hodnotou je x-ová súradnica na mape a treťou je y-ová súradnica na mape. Veľký problém pri útoku vidíme v zmenách jednotlivých dát a ich následným vložením do databázy. Za týmto účelom sme pridali do tabuľky zobrazenej na obr. 30 ďalšie 2 parametre, a tými sú čas pridania, respektíve úpravy dát, a taktiež unikátna hodnota záznamu. Výsledná tabuľka je zobrazená na obr. 31.



Obrázok 30. Základný dátový model



Obrázok 31. Rozšírený dátový model

Hodnoty, ktoré sme zaznamenali, sme vkladali do relačnej tabuľky MySql. Aj keď sme skúšali aj nerelačnú databázu DynamoDB na základe pevnej štruktúry sme sa rozhodli použiť práve relačnú databázu MySql.

Hodnoty, ktoré nám prichádzajú do tabuliek sú povolené iba pre konkrétnu aplikáciu, ktorá vykonáva monitorovanie cesty jednotlivých dopravných systémoch, v našom prípade sa jedná o automobily. Za týmto účelom sme vytvorili rolu, konkrétne IAM Role, ktorá povoľuje vkladať záznamy iba aplikácii.

Hlavnou myšlienkou rozšírenia tabuľky bola kontrola stavu operácií jednotlivých záznamov. Máme na mysli operáciu aktualizovania (*update*) a zmazania (*delete*) záznamov.

Za týmto účelom sme vytvorili nad databázou notifikáciu. Vždy chceme, aby každá operácia *update* a *delete* bola kontrolovaná pred vykonaním jednotlivých operácií. Ak aplikácia vykonáva spomenuté operácie *update* a *delete*, tak sa najskôr skontroluje ich úloha/rola, ktorá je priradená do aplikácie. Následne je skontrolovaná unikátna hodnota z tabuľky autentifikátor (*authentificator)*. Každá úloha (*rola*) ma vytvorených 5 náhodných token-ov, ktorých hlavička sa posiela automaticky pri zavolaní operácie *update* a *delete*. Týchto 5 náhodných vytvorených token-ov je závislých aj na čase vytvorenia záznamu a dochádza k ich automatickej úprave každých 12 hodín. Úprava token-u funguje dovtedy, pokiaľ nedôjde k neoprávnenému pokusu o úpravu dát. Ak sa takáto situácia vyskytne, tak sú tokeny pregenerované automaticky a je vyvolaná zmena na vytvorenie novej role pre aplikáciu. Upravenie notifikácie je druhou samozrejmou vecou, ktorá je vykonaná automaticky pri neoprávnenej snahe o získanie dát.

```
aws rds create-event-subscription \
    --subscription-name myeventsubscription \
    --sns-topic-arn arn:aws:sns:us-east-1:802#########:myawsuser-RDS \
    --enabled
```

```
aws rds delete-event-subscription --subscription-name myrdssubscription
```

```
aws rds remove-source-identifier-from-subscription \
    --subscription-name myrdseventsubscription \
    --source-identifier mysqldb
```

5.7.2.2 Presun dát medzi databázou a dátovým skladom

Podľa nášho úsudku sú dáta najviac citlivé na útok na úrovni databázy, a preto sme sa rozhodli chrániť tieto údaje pomocou ich presunu do dátového skladu. Tento presun sa deje automaticky na základe posledných zmien pri dátach.

Ak hodnota dát, ktoré sa nachádzajú v databáze presiahne hodnotu 1 mesiaca, tak sú dáta automaticky presunuté, a to nasledovným spôsobom :

- Je vyvolaný skript nad tabuľkou car_coordinate, ktorý vyzerá nasledovne:

```
mysql> SELECT * FROM  car_coordinate
         JOIN ccoordinate_has_authenticator USING id_car
        JOIN authenticator USING (id_authenitcator)
        WHERE DATE_SUB((DATE_SUB(curdate(), INTERVAL 1 MONTH)). LIMIT 0,1000000
        INTO OUTFILE '/tmr,/car_coordinatel.csv FIELDS TERMINATED BY ' ENCLOSED BY '"'
        LINES TERMINATED BY '
```

- Následne je zavolaná procedúra

```
mysql> call export_csv_split(‘tbl_product’,1000000);
```

- Postupne dochádza k presunu vygenerovaných súborov CSV do adresára s názvom s3-redshift:

```
mkdir ~/s3-redshift
mv /tmp/*.csv ~/s3-redshift/
```

- Keďže spustená inštancia musí mať právo spúšťať a manipulovať s S3, tak sme museli na tieto účely nainštalovať *s3cmd* príkazový riadok:

```
rpm -Uhv http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
```

- Následne sme vytvorili nové vedro (*bucket*) pre Redshift

```
s3cmd mb s3://s3-rshift
```

- Po vytvorení vedra (*bucket*) sme začali synchronizovať dátový adresár CSV do vedra S3:

```
s3cmd sync s3-redshift s3://s3rshift
```

- V ďalších krokoch sme si pripravili Redshift:
- Redshift beží ako verzia na PostgreSql 8.X. Na prístup do klastra Redshift s poskytnutým koncovým bodom a povereniami môžeme použiť štandardného klienta PostgreSql.
- Na inštaláciu príkazu PostgreSql sme využili nasledujúci príkaz:

```
yum install -y postgresql #Redhat/CentOS
sudo apt-get install -y postgresql #Debian
```

- Na prístup do klastra Redshift sme použili program psql:

```
psql --host=mysql-redshift.cd23imcbfhbd.ap-southeeast-1.redshift.amazon.com --port=5
```

- Následne sme vytvorili tabuľku podobnú tabuľke v MySql ktorej skript vyzerá nasledovne:

```
CREATE TABLE car_coordinate (car_id INT PRIMARY KEY NOT NULL, x_coordinate VARCAHR2(45) NOT NULL, y_coordinate VARCAHR2(45) NOT NULL)
```

- Ďalej bol spustený príkaz COPY, aby sme získali prístup k súborom CSV v našom segmente S3 a paralelne ich načítali do tabuľky:

```
copy car_coordinate FROM ‚s3://s3-rshift/s3-redshift/ car_coordinate ‘credentials ‚aws_access_key_redshift‘
```

- Po počiatočnom načítaní údajov z segmentu S3 musíme spustiť príkaz *VACUUM* na usporiadanie naších údajov a príkazy „analyzovať“ na aktualizáciu štatistík tabuľky:

```
vacuum car_coordinate
analyze table_product
```

- Po vykonaní všetkých krokov došlo k overeniu štruktúry tabuliek:

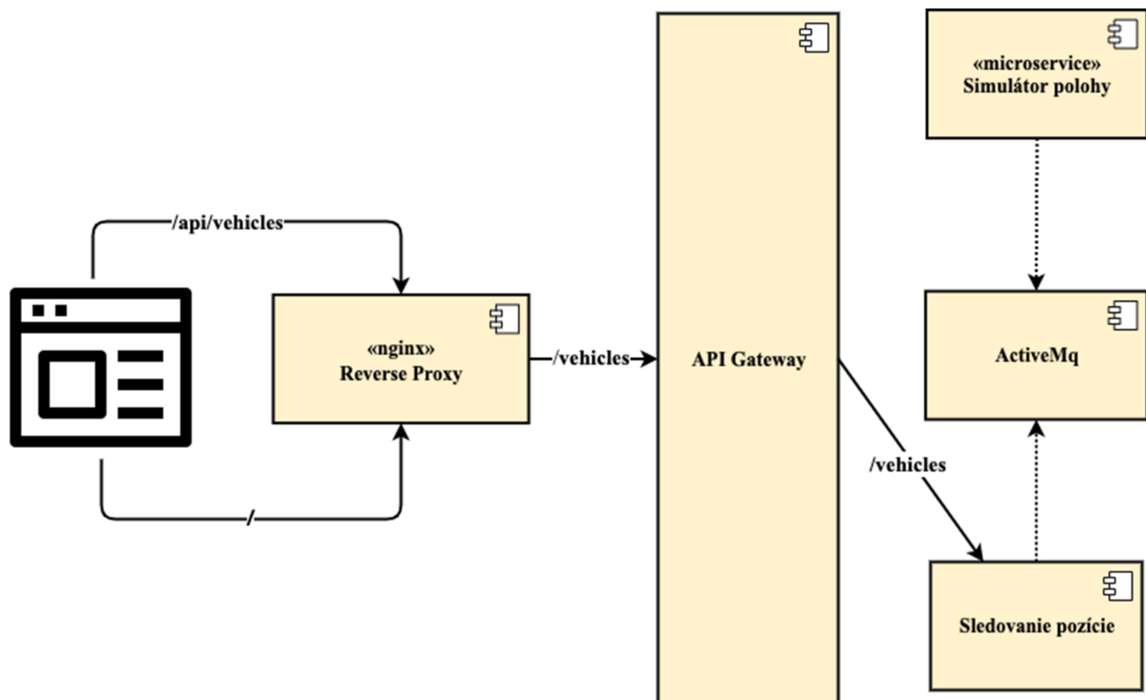
```
analytical=# \d car_coordiante;
TABLE “public.car_coordinate”
```

Po aplikovaní všetkých krokov, ktoré nám pomáhajú presunúť dáta z relačnej databázy MySql do dátového skladu Amazon Redshift došlo k zmazaniu presunutých záznamov z tabuliek: *car_coordiante, authenitificator* a následne aj *ccar_coordinate_has_authetificator,* aby došlo k redukovaniu vyťaženia databázy, a taktiež k redukcii možného útoku na veľké množstvo údajov, ktoré sa nachádzali v databáze, ktorá monitoruje pozíciu automobilov na mape[5].

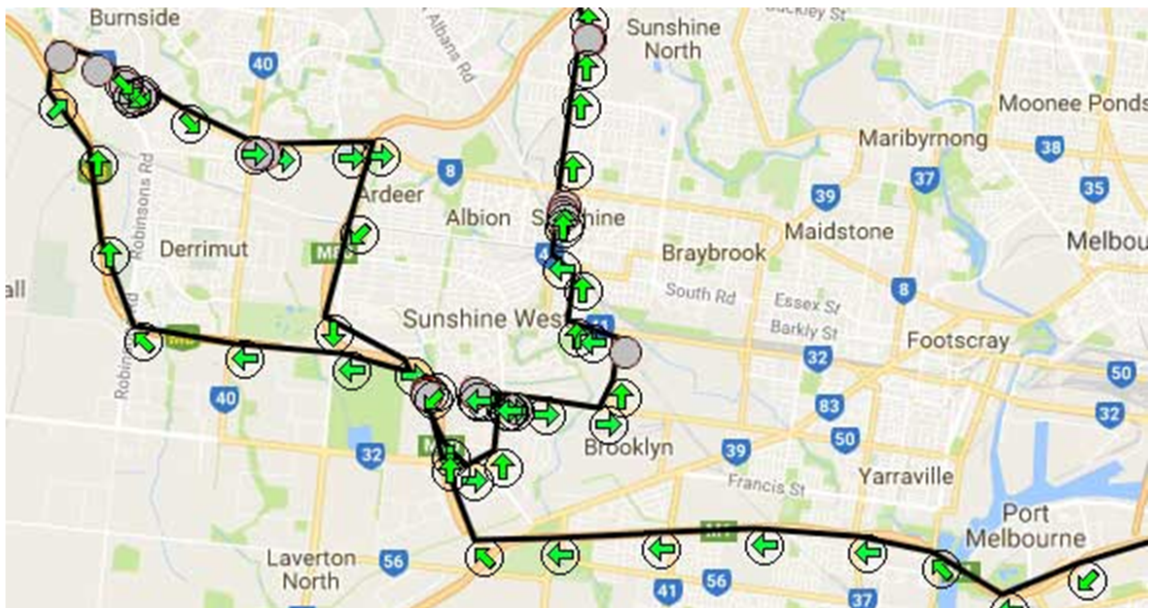
5.7.3 Experimenty pre metódu presunu dát v reálnom čase

K experimentálnej činnosti sme vytvorili aplikáciu, ktorá je zobrazená na obr. 32 a 33. Vytvorená aplikácia monitoruje automobily, ktoré rozvážajú tovar zákazníkom po celom Anglicku.

[5] Problematiku bezpečnosti dát sme publikovali v časopise Scientific Letters of the University of Žilina / Communications na Slovensku (Žilina).



Obrázok 32. Architektúra aplikácie



Obrázok 33. Monitorovanie áut pomocou GPS

Počas experimentálnej činnosti bolo pomocou áut, ktoré sa nachádzali v systéme, vložených za 10 minút približne 10 000 záznamov. Aj keď sa niektoré pozície už v systéme nachádzali, tak databáza fungovala efektívne a dokázala prichádzajúce dáta spracovávať korektne.

Systém, na ktorý sme nasadili aplikáciu a databázu je cloud od spoločnosti Amazon. Aby nedochádzalo nikdy k žiadnemu nedostatku výkonnosti, keďže výpočtové jednotky EC2 majú svoje výkonnostné obmedzenie, boli sme nútení zabezpečiť flexibilnosť výkonu pomocou služby *Amazon Auto Scaling Group*.

Pri pokuse o úpravu hodnôt sme skúsili poslať rovnakú požiadavku ako posielala aplikácia pomocou rôznych aplikácií a služieb so snahou získať, respektíve upraviť dáta. V snahe overiť naše riešenie sme skopírovali prihlasovací token, ktorý by nám už mal v bežnej prevádzke zabezpečiť prístup k dátam. Tento pokus mal reálnejšiu kontúru na úpravu chýb, ale navrhnutá metóda skontrolovala pravosť aplikácie a vyhodnotila náš pokus ako neplatný a požiadavku automaticky vymazala.

Pri vypnutí našej kontrolnej metódy sme vyslali rovnakú podmienku, a v tom prípade boli dáta zmenené. Naša druhá metóda, ktorá presúva automaticky dáta na základe časového rozpätia z relačnej databázy MySql do dátového skladu Redshift spôsobila, že došlo k úprave iba malého množstva záznamov a ostatné hodnoty, ktoré boli presunuté, zostali ochránené.

Ako je vidieť z tabuľky 10, tak sledované hodnoty, ktoré sú pre nás podstatné sa oproti konvenčnej metóde pri rovnakej konfigurácii a rovnakom počte útokov zlepšili takmer v každom spektre, okrem spektra času. Nami navrhnutá metóda umožnila zvýšiť bezpečnosť práce s údajmi v mnohých aspektoch.

Pri testovaní metódy sme využili testovacie metódy tretích strán, nakoľko sa nevenujeme obchádzaniu bezpečnostných pravidiel. Preto sme na spomenuté účely využili stránky *sonarcube* a *hcltechsw*, ktoré nám poskytli potrebné výsledky.

Po aplikovaní nami navrhnutej metódy sa čas potrebný na spracovanie základných operácií ako sú operácie zmazania, úpravy a vloženia údajov zvýšil, pretože dodatočná kontrola si vyžiadala dodatočné kroky, ktoré proces spomaľujú. Aj keď je pre nás čas veľmi dôležitým faktorom, tak zvýšenie bezpečnosti o niekoľko percent nám zvýšenie času vykompenzovalo.

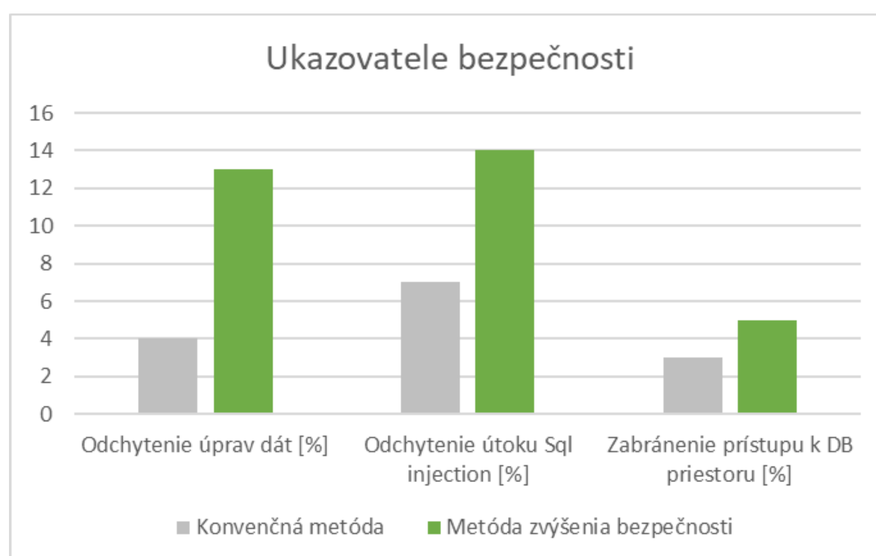
Tabuľka 10. Porovanie sledovaných hodnôt medzi konvenčnou metódou a metódou zvýšenia bezpečnosti

Číslo vlastnosti	Sledovaná vlastnosť	Konvenčná metóda	Metóda zvýšenia bezpečnosti
1	Presúvanie záznamov [Áno/Nie]	Nie	Áno
2	Časová pečiatka [Áno/Nie]	Nie	Áno
3	Odchytenie úprav dát [%]	4	13
4	Odchytenie útoku Sql injection [%]	7	14
5	Zabránenie prístupu k DB priestoru [%]	3	5
6	Počet útokov	10 000	10 000
7	Čas zmazania údajov	0.004	0.005
8	Čas úpravy údajov	0.009	0.0011
9	Čas vloženia údajov	0.005	0.006

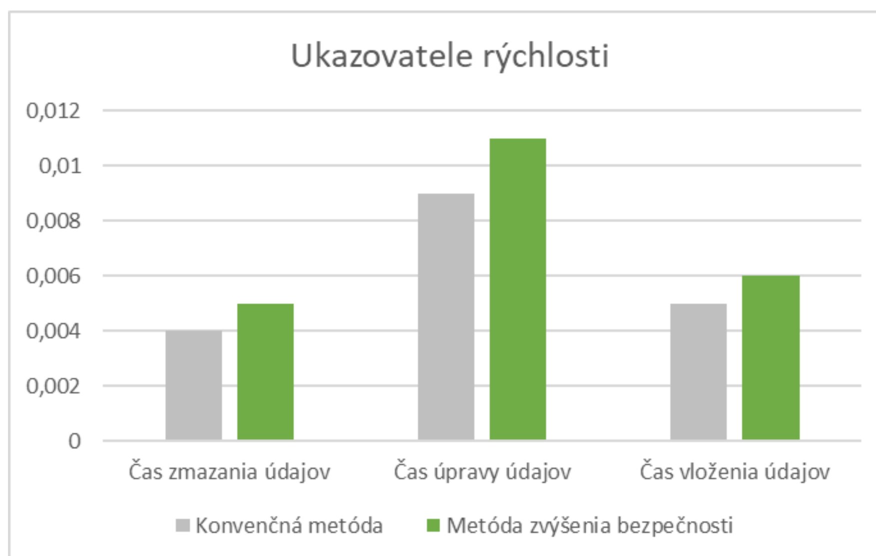
Ako je vidieť z grafu na obr. 34 a obr. 35, tak grafické zobrazenie výsledkov ukazuje jasný trend. Zelenou farbou sú zobrazené výsledky po implementovaní nášho prístupu a šedou sú zobrazené výsledky s použitím konvenčnej metódy.

Ako je vidieť na obr. 34, kde nám zelená farba predstavuje hodnoty po aplikovaní našej metódy (väčší stĺpec znamená lepšie výsledky), tak sme v niektorých oblastiach dosiahli aj 3-násobné zlepšenie ochrany (viď stĺpec *Odchytenie úprav dát*). Bohužiaľ, tento trend nepokračoval a ako je vidieť na obr. 35, tak sa nám sledované hodnoty aspektov zhoršili (čím vyšší stĺpec, tým horší výsledok).

Po skončení procesu sme získali výsledky, ktoré síce boli pri zvýšení odolnosti uspokojivé, ale pri rýchlosti operácií zmazania, úpravy a vloženia údajov neuspokojivé, z dôvodu malého zväčšenia časového aspektu.



Obrázok 34. Grafické zobrazenie redukcie bezpečnosti po aplikovaní našej metódy



Obrázok 35. Grafické zobrazenie klesania rýchlosti po aplikácií našej metódy do procesu

5.7.4 Zhrnutie riešenia metódy presúvania údajov v reálnom čase

Súčasná výrazne rastúca doprava si vyžaduje sofistikované riešenia na minimalizáciu straty údajov, nežiadúcu úpravu hodnôt tretími stranami, a taktiež redukciu útokov na aplikáciu alebo databázu s citlivými údajmi. Množstvo útokov na citlivé údaje núti viaceré podniky zaoberať sa vo väčšej miere bezpečnosťou systémov, a taktiež kladie väčší dôraz na efektívny spôsob manipulácie s dátami v aplikácii. Metódy, ktoré boli v minulosti považované za dostačujúce riešenie na zabezpečenie citlivých údajov sú dnes buď už zastarané alebo nepoužívané z dôsledku nedostatočného zabezpečenia pred rôznymi typmi útokov. Ako dokazuje nami navrhnutá metóda, efektívny spôsob ako zamedziť k strate údajov alebo redukovať neoprávnený prístup je kontrola dát na základe časového aspektu.

V našom navrhnutom riešení používame riešenie pridávania hodnôt pri každej prichádzajúcej hodnote z každého auta. Pridaním časovej pečiatky následne kontrolujeme pravosť každého záznamu. Táto hodnota nie je pre každé auto unikátna, preto sme vytvorili štruktúru, ku ktorej má prístup iba samotný systém a slúži na overenie pravosti záznamu. Keďže dátové úložisko je pre nás alfou a omegou vytvorili sme aj metódu presunu informácií z databázy do dátového skladu na základe časového spektra, ktorý pomáha nielen k zníženiu bezpečnostného rizika, ale taktiež redukuje vyťaženie primárneho dátového úložiska. Na základe vytvorenej metódy na presun dát sme redukovali vyťaženie primárneho dátového úložiska približne o 34 percent každý mesiac. Nami navrhnutá metóda nám umožnila zachytiť 13 percent úmyselných zmien dát, ktoré nemali poverenie na danú zmenu.

Pri spustení aplikácie do procesu sme mali možnosť otestovať množstvo funkcionalít s rôznou veľkosťou dát. Bohužiaľ, náš systém nebol dostatočne pripravený na masívnejší počet záznamov, ktoré prichádzali v reálnom čase do procesu, a tým ovplyvnili práve sa transformujúce sa dáta.

Už pri malej veľkosti údajov(testovali sme súbor o veľkosti 400 MB) a vstupu reálnych dát, ktoré prichádzali do systému každých 5 sekúnd systém spracovával záznamy až po finálnom procese, čo si myslíme, že je pomerne neskoro s ohľadom na možnosť ich vzájomného ovplyvňovania.

Na základe výskytu tohto nedostatku sa domnievame, že dáta, ktoré sa práve transformujú, by mali byť ovplyvňované dátami v reálnom čase, v čase vstupu do systému a nie až po jeho skončení. Tým pádom nebude nutná zdĺhavá kontrola a úprava po skončení celého procesu.

5.8 Ovplyvňovanie nahromadených dát reálnymi dátami

Na základe záverov z predchádzajúcej kapitoly a odhalených nedostatkov sme museli riešiť problém spojený s narastajúcim objemom reálnych dát a ich vplyvom na dáta, ktoré sa v systéme práve transformujú.

Podľa nášho názoru a testov nie je možné nechať úpravu dát na koniec procesu a následne upravovať transformované dáta, hodnotami, ktoré prišli do procesu počas ich transformácie. Proces vyhľadania dát vo veľkom množstve je zdĺhavý a neefektívny.

Na základe spomenutých nedostatkov a nutnosti dodatočnej úpravy sme si stanovili ciele a využili prostriedky, ktorých cieľom je:

- redukovať počet úprav po transformácii,
- ovplyvňovať nahromadené údaje, ktoré sa transformujú údajmi, ktoré počas procesu vstupujú do systému,
- maximalizovať počet úprav počas transformačného procesu.

Nakoľko nie sme prví a ani poslední výskumníci, ktorí sa danou problematikou ovplyvňovania nahromadených dát s dátami, ktoré prichádzajú počas procesu do systému zaoberajú, tak v nasledujúcej podkapitole preskúmame už existujúce riešenia.

5.8.1 Aktuálny stav skúmanej problematiky ovplyvňovania nahromadených dát reálnymi dátami

Porovnanie relačných dátových modelov s nerelačnými dátovými modelmi NoSql už bolo uvedené v mnohých článkoch [112][27][11][107]. V spomínaných článkoch autori dokázali, ktoré typy operácií sa z hľadiska efektívnosti lepšie hodia k databáze a oproti iným databázam ponúkajú rýchlejšiu odozvu. V článku [20] autori analyzovali výkonnosť medzi databázami Sql a databázami NoSql s typom kľúč - hodnota (*key-value*). Tieto experimenty v článkoch [14] a [80] porovnávali tradičnú databázu Oracle a databázu MongoDB.

Na základe týchto porovnaní a experimentov poskytuje článok [80] podrobné výsledky a analýzy týkajúce sa efektívnosti údajov ako článok [14]. Autori identifikovali problémy a výzvy pri spracovaní veľkých dát pomocou MapReduce. Štyri hlavné kategórie sú (1) ukladanie údajov, (2) analýza veľkých údajov, (3) online spracovanie a (4) bezpečnosť a ochrana osobných údajov. Vedci [93] poskytli prístup k migrácii medzi rôznymi databázami NoSql orientovanými na stĺpce. Bolo navrhnuté, aby údaje predstavovali v štandardnom formáte a zodpovedali za preklad zdrojovej databázy do cieľovej. Výsledky metód v článku [93] sú z hľadiska efektívnosti asi o 10% účinnejšie ako výsledky publikované v článku [42].

Pri porovnaní dvoch článkov zameraných na problém transformácie údajov (Pinto, 2009) a [109] sme mali možnosť vidieť situácie, keď vedci pri migrácii údajov z relačnej databázy do nerelačnej použili tri časti. Článok [109] je efektívnejší a ľahšie rozšíriteľný na základe experimentov a možnej migrácie dát a zaktiež ponúka všeobecnejší modul pre budúcu prácu ako práca v článku (Pinto, 2009).

V mnohých prípadoch vedci používajú *Apache Sqoop* [105] a *DataX* [69] pri prenose údajov z relačnej databázy do nerelačnej databázy. *Apache Sqoop* je založený na prenose dát z relačnej databázy do Apache Hadoop. *Apache DataX* je vytvorený na princípe výmeny údajov medzi heterogénnymi zdrojmi údajov.

Pri riešení problému je možné identifikovať dva kmeňové príspevky. Prvý príspevok [24] je rámec pre prenos z Sql na NoSql pomocou MapReduce. Princíp tejto metódy je založený na ukladaní všetkých tabuliek relačnej databázy do jednej tabuľky v HBase. Druhý príspevok [70] predstavuje tri pokyny pre transformáciu relačnej schémy na schému HBase na základe údajov modelu HBase.

Vyjadruje vzťahy medzi ich schémami ako súbor vnorených mapovacích schém pri automatickej transformácii relačných údajov na reprezentáciu HBase. Tieto tri pokyny zahŕňajú zoskupenie korelovaných údajov s rodinou stĺpcov, pridanie odkazov na cudzie kľúče, ak jedna strana potrebuje prístup k údajom na druhej strane a zlúčenie prepojených tabuliek s údajmi, aby sa znížil počet cudzích kľúčov.

S prácou, ktorá sa zaoberá časovo orientovanou databázovou architektúrou, sme sa stretli počas nášho výskumu v príspevku [93], ktorý spravuje nedefinované hodnoty a navrhuje komplexnú klasifikáciu systémov na transakciách, prístupoch a indexoch. Pretože do nášho systému môžu vstupovať rôzne dátové typy, či už sú to štruktúrované alebo neštruktúrované dáta, v spomínanej práci sa zaznamenáva modelovanie nedefinovaných hodnôt. Ďalej článok pokrýva synchronizačné procesy využívajúce skupiny údajov. Kritickou súčasťou uvedeného článku sú riešenia pre efektívny zber dát s dôrazom na nedefinované hodnoty a stavy.

Stručne zhrnuté, predchádzajúce práce sú zamerané predovšetkým na zlepšenie výkonu čítania alebo na migráciu údajov z databázy NoSql, aby sa získala vysoká dostupnosť a škálovateľnosť. Neposkytujú však dostatočne efektívne riešenie pre prichádzajúce údaje, ktoré môžu vstúpiť počas prebiehajúceho procesu migrácie.

Po preštudovaní spomenutých prác stále vidíme priestor na zlepšenie. Rozhodli sme sa implementovať do procesu metódu, ktorá bola už výskumníkmi pridaná do procesu, ale podľa nášho úsudku príliš neskoro, čo vedie k oneskorenému spracovaniu. Doplnením metódy do procesu redukujeme počet operácií po skončení procesu a efektívnejšie dokážeme zachytiť prichádzajúce dáta. Z procesu sme existujúcu metódu neodstránili. V našom procese existujúca metóda stále vykonáva svoju činnosť, ale proces je doplnený o našu metódu, čo v konečnom dôsledku redukuje čas potrebný na spracovanie existujúcej metódy a taktiež naša metóda redukuje počet operácií existujúcej metódy.

5.8.2 Dôvod skúmania ovplyvňovania nahromadených dát reálnymi dátami

Transformačný proces, v ktorom dochádza k zmene štruktúry zo zdrojovej databázy na cieľovú databázu v mnohých prípadoch trvá od niekoľkých minút až po desiatky hodín. Počas tohto procesu môžu prichádzajúce dáta výrazne ovplyvniť práve sa transformujúce hodnoty, a tým výrazne redukovať čas potrebný na dodatočnú úpravu po skončení transformačného procesu.

Tento fakt bol povšimnutý výskumníkmi [109] [24], ktorí navrhli efektívne metódy ako vyriešiť problém s reálnymi dátami. Podľa nášho názoru je implementované riešenie prvým krokom ako správne zohľadniť prichádzajúce dáta, a tým ovplyvniť konečný výsledok v cieľovej databáze, ale myslíme si, že k spomenutému ovplyvneniu dochádza až príliš neskoro.

Na základe nášho úsudku, sme sa rozhodli posunúť ovplyvňovanie dát namiesto úplného konca procesu, to znamená pred uložením do databázy, do času, keď sa nahromadené dáta aktuálne transformujú a môžu byť ovplyvnené prichádzajúcimi dátami. Náš navrhnutý princíp, ktorý je popísaný nižšie úplne neodstránil metódu, ktorá bola spomenutá, ale výrazne ju redukoval.

5.8.3 Prínos do problematiky ovplyvňovania nahromadených dát reálnymi dátami spojenými s prichádzajúcimi dátami v reálnom čase

Náš prínos je zameraný na situáciu, ktorá vychádza zo zanedbania zachytávania dát v reálnom čase pri migrácií dát z relačnej databázy na nerelačnú. V tomto procese zohrávajú dáta, ktoré do procesu prichádzajú počas prebiehajúcej transformácie dát významnú úlohu, a to nie len z dôvodu možnosti ovplyvňovania dát, ktoré sú nahromadené a už dochádza k ich transformácii, ale najmä z dôvodu nutnej úpravy dát v cieľovej databáze. Ako bolo publikované [20] operácia vyhľadávania nie je v nerelačných databázach rovnako efektívna ako pri databázach relačných.

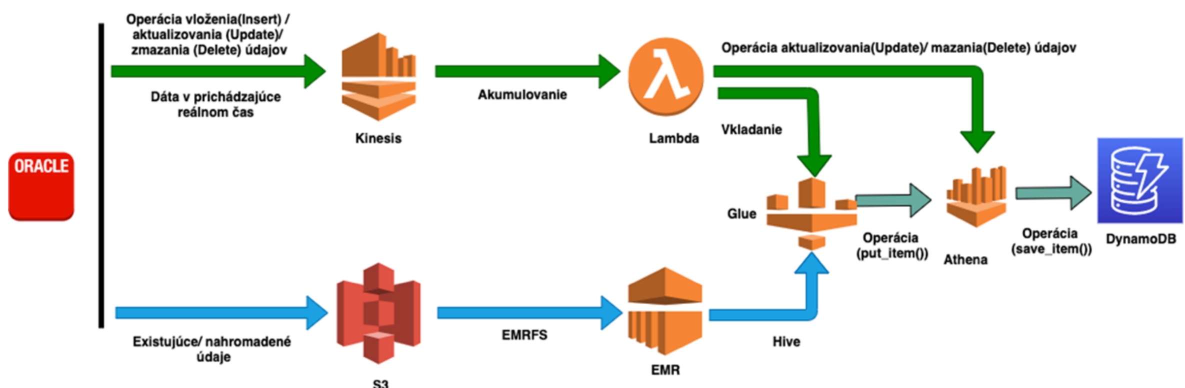
Zo spomenutého dôvodu sme vytvorili architektúru, ktorá umožňuje pri transformačnom procese zachytávať dáta v reálnom čase, a tým zamedziť dodatočnej úprave po presune dát z relačnej databázy Oracle na nerelačnú databázu DynamoDB[6]. Táto architektúra sa skladá z 2 častí:

- *Vrchná vetva* (Dáta v reálnom čase)
- *Spodná vetva* (Nahromadené dáta)

Dáta v reálnom čase sú dáta, ktoré do systému vstúpili počas spusteného transformačného procesu, to znamená, že sa do databázy neukladajú, ale sú automaticky presúvané do transformačného procesu. Dáta, ktoré pôsobili v databáze Oracle dlhší čas, napríklad niekoľko dní, mesiacov alebo rokov, nazývame nahromadené dáta.

[6] Problematiku ovplyvňovania dát sme prezentovali a diskutovali na IEEE konferencii FRUCT v Rusku (Moskva) - 27. - 29. január 2021.

Tieto dáta je nutné počas zmeny podrobiť určitému typu zmeny dát. Dáta mali štruktúru, respektíve boli uložené v relačnej databáze a musia byť z dôvodu efektívnosti zmenené, aby došlo k ich efektívnej manipulácii a spomenutý formát vyhovoval databázovému typu, pre ktorý je transformačný proces určený, keďže existujú väčšie množstvá nerelačných databáz. Na proces zmeny dát z jedného typu na iný typ sa používajú rôzne nástroje ako je *Apache Flink*, *Spark*, *Samza*, *Hadoop* alebo *Hive*, ktoré spĺňajú požiadavky efektívnej zmeny dátových štruktúr na základe definovaných pravidiel.



Obrázok 36. Návrh zachytávajúci dáta v reálnom čase a nahromadené dáta

5.8.3.1 Vrchná vetva navrhnutej architektúry

Počas vykonávania transformačného procesu je nevyhnutné zabezpečiť stav, kedy do procesu prichádzajú nové príkazy ako je vloženie, aktualizovanie a zmazanie dát.

Hodnoty, ktoré zaznamenávame sú pre tabuľku zákazník, ktorá je zobrazená na obr. 37. Tabuľka obsahuje 4 atribúty, ktoré sú typu INT a primárny kľuč je definovaný ako kompozitný primárny kľúč zložený z atribútov *customer_id, order_id* a *product_id* .

Obrázok 37. Databázová tabuľka zákazník (customer)

Horná vetva, ktorá je zobrazená na obr. 37 slúži na spomenutý účel. Dáta, ktoré prichádzajú do systému sú zachytené pomocou nástroja Amazon Kinesis. Amazon Kinesis uľahčuje zber, spracovanie a analýzu streamovaných údajov v reálnom čase, takže nám pomáha získať aktuálne informácie a rýchlo na ne reagovať. V konfigurácii Oracle sme nastavili binlog_format na ROW na zachytenie transakcií pomocou modulu *BinLogStreamReader*.

Na povolenie *binlog*, sme nastavili tiež hodnotu parametra *log_bin*. Skript na nastavenie vyzerá nasledujúco :

```
[oracleld]
secure-file-priv = ""
log_bin=/data/binlog/binlog
binlog_format=ROW
server-id = 1
tmpdir=/data/tmp
```

Z dôvodu zachytenia dát z relačnej databázy Oracle, sme vytvorili skript, ktorý zachytáva transakcie a odosiela ich priamo do služby Amazon Kinesis Streams. Skript potrebný na vykonanie tohto procesu je poskytnutý na aktuálnej adrese: *https://github.com/romanceresnak/real-time-data-capture/blob/master/kinesis.py*

```
INSERT INTO customer
(customer_id, order_id, product_id, quantity)
VALUES (5000, 2345,234,356);                                    (1)
```

```
UPDATE customer
SET quantity = 55
WHERE customer_id = 55;                                         (2)
```

```
DELETE customer WHERE order_id = 50;                            (3)
```

Poskytnutý skript predstavuje názorne údaje JSON generovane python skriptom. Atribút *type* definuje transakcie záznamom typu JSON:

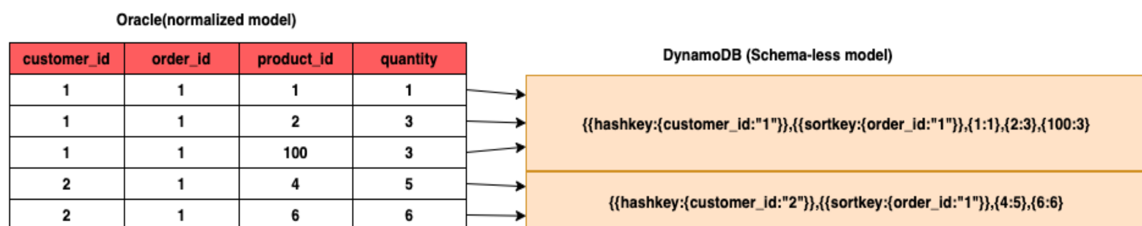
- *WriteRowsEvent(INSERT)*
- *UpdateRowsEvent(UPDATE)*
- *DeleteRowsEvent(DELETE)*

Tu je ukážka údajov v JSON formáte:

*{"table": "customer", "row": {"values": {"order_id": "1", "quantity": 100, "customer_id": "74187", "product_id": "1"}}, "type": "**WriteRowsEvent**", "schema": "test"}*

*{"table": "customer", "row": {"before_values": {"order_id": "1", "quantity": 1, "customer_id": "74187", "product_id": "1"}, "after_values": {"order_id": "1", "quantity": 99, "customer_id": "74187", "product_id": "1"}}, "type": "**UpdateRowsEvent**", "schema": "test"}*

*{"table": "customer", "row": {"values": {"order_id": "100", "quantity": 1, "customer_id": "74187", "product_id": "1"}}, "type": "**DeleteRowsEvent**", "schema": "test"}*



Obrázok 38. Mapovacie pravidlo

5.8.3.2 *Spodná vetva navrhnutej architektúry*

Dáta, ktoré do systému vstupujú je možné do systému nahrať 2 spôsobmi. Prvým spôsobom je vytvorenie export databázy, ktorá sa už nachádza v prostredí Amazon. Export každej databázy v prostredí Amazonu vytvára export dát do S3 vedra (*bucket)* vo formáte CSV. Druhým spôsobom je vytvorenie exportu relačnej databázy na ľubovoľnom zariadení a následne nahrať súbor vo formáte CSV do Amazon S3 bucket.

```
SELECT * FROM customer WHERE <condition_1>
INTO OUTFILE '/data/export/customer/1.csv' FIELDS TERMINATED BY ',' ESCAPED BY '\\' LINES TERMINATED BY '\n';
```

```
SELECT * FROM customer WHERE <condition_2>
INTO OUTFILE '/data/export/customer/2.csv' FIELDS TERMINATED BY ',' ESCAPED BY '\\' LINES TERMINATED BY '\n';
...
SELECT * FROM customer WHERE <condition_n>
INTO OUTFILE '/data/export/customer/n.csv' FIELDS TERMINATED BY ',' ESCAPED BY '\\' LINES TERMINATED BY '\n'
```

Pre tento účel použijeme príkaz *aws s3 sync*. Tento príkaz pracuje interne s funkciou viacdielneho nahrávania S3. Zhoda vzorov môže vylúčiť alebo zahrnúť konkrétne súbory. Okrem toho, ak proces synchronizácie zlyhá v priebehu spracovania, nemusíme znova odovzdávať rovnaké súbory.

Príkaz *sync* porovnáva veľkosť a upravený čas súborov medzi lokálnymi verziami a verziami S3 a synchronizuje iba miestne súbory, ktorých veľkosť a upravený čas sa líšia od súborov v S3. Príkazy vyzerajú nasledovne:

```
aws s3 sync /data/export/purchase/ s3://<your bucket name>/purchase/
aws s3 sync /data/export/<other path_1>/ s3://<your bucket name>/<other path_1>/
...
aws s3 sync /data/export/<other path_n>/ s3://<your bucket name>/<other path_n>/
```

Súčasťou práce je experiment súvisiaci s porovnaním metód, ktoré patria k veľmi populárnym metódam mapovania objektov pri veľkých dátach. Pri transformačnom procese porovnáme ako nám zmena frameworku ovplyvní rýchlosť zmeny dát. Na tento účel sme vybrali 2 nástroje a postupy:

- Apache Hive s externou tabuľkou
- Využitie metódy MapReduce

5.8.4 Apache Hive s externou tabuľkou

Pomocou vlastnosti *org.apache.hadoop.hive.dynamodb.DynamoDBStorageHandler* sme vytvorili externú tabuľku *Hive* pro údaje na S3 a vložili ju do inej externej tabuľky oproti tabuľke DynamoDB. Nasledujúci vzorový kód predpokladá, že tabuľka Hive pre DynamoDB sa vytvorí so stĺpcom *customer*, ktorý je typu *ARRAY <STRING>*. Stĺpce *product_id* a *kvantity_id* sú agregované, zoskupené podľa *customer_id* a *order_id* a vložené do stĺpca *customer* so stĺpcami *CollectUDAF* od *Brickhouse*. Kód poskytujúci úpravu je možné vidieť na tomto odkaze: *https://github.com/romanceresnak/real-time-data-capture/blob/master/HiveMapper.txt* .

Bohužiaľ, dátové typy *MAP, LIST, BOOLEAN a NULL* nie sú podporované triedou *DynamoDBStorageHandler*, takže bol vybraný dátový typ *ARRAY <STRING>*. Stĺpec produktov typu *ARRAY <STRING>* v Hive je porovnávaný s atribútom typu *StringSet* v DynamoDB. Vzorový kód väčšinou ukazuje, ako *Brickhouse* funguje, a iba pre tých, ktorí chcú agregovať viac záznamov do jedného atribútu typu *StringSet* v DynamoDB.

5.8.4.1 MapReduce

Úloha mapovača je čítať každý záznam zo vstupných údajov na S3 a mapovať vstupné páry typu kľúč – hodnota (*key-value*) na stredné páry kľúč - hodnota. Rozdeľuje zdrojové údaje zo S3 na dve časti (kľúčová a hodnotová časť) oddelené znakom TAB („\ t“).

Údaje mapovača sú usporiadané podľa ich sprostredkujúceho kľúča (*customerid* a *orderid*) a odoslané do reduktora. Záznamy sa vkladajú do DynamoDB v redukčnom kroku.

Prikladáme skript napísaný v jazyku python s názvom mapper.py:

```
#!/usr/bin/env python
import sys
# get all lines from stdin
for line in sys.stdin:
    line = line.strip()
    cols = line.split(',')
# divide source data into Key and attribute part.
# example output : "1,1           1,10"
            print '%s,%s\t%s,%s' % (cols[0],cols[1],cols[2],cols[3] )
```

Úloha redukcie všeobecne je prijímať výstup vytvorený po spracovaní mapy (čo sú páry kľúč / zoznam hodnôt), a potom vykonať operáciu na zozname hodnôt proti každému kľúču.

V tomto prípade je reduktor napísaný v jazyku Python a je založený na streamovaní STDIN / STDOUT / hadoop. Reduktor prijíma dáta zoradené a usporiadané prostredným kľúčom nastaveným v mapovači, identifikátore zákazníka a objednávke (stĺpce [0], stĺpce [1]) a ukladá všetky atribúty pre konkrétny kľúč do slovníka *item_data*. Atribúty v slovníku *item_data* sú vložené alebo vyprázdnené do DynamoDB zakaždým, keď zo servera *sys.stdin* pochádza nový sprostredkujúci kľúč. Skript, ktorý sme využili je možné nájsť na nasledujúcom odkaze:
https://github.com/romanceresnak/real-time-data-capture/blob/master/MapReduce.py, ktorý bude v programe označovaný ako reducer.py.

Po napísaní skriptov sme spustili úlohu MapReduce, pripojili sa k hlavnému uzlu EMR a spustili úlohu streamovania Hadoop. Umiestnenie alebo názov súboru *hadoop-streaming.jar* sa môže líšiť v závislosti od verzie EMR. Výnimočné správy, ktoré sa vyskytli počas spustenia reduktorov, sa ukladajú do adresára priradeného ako voľba – output. Hodnoty kľúča hash a kľúčov rozsahu sa tiež zaznamenávali, aby sa zistilo, ktoré údaje spôsobujú výnimky alebo chyby.

```
$ hadoop fs -rm -r s3://<transformation>/<nosql/result>
$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar \
-input s3://< transformation >/<input path> -output s3://<bucket name>/<output path>\
-file /<transformation>/mapper.py -mapper /< transformation >/mapper.py \
-file /< transformation >/reducer.py -reducer /< transformation >/reducer.py
```

5.8.4.2 Spojenie spodnej a vrchnej vetvy nami navrhnutej architektúry

K spojeniu vetiev dochádza, vždy po úspešnej zmene dát v hornej a dolnej vetve. Transformácia dát môže trvať od niekoľkých minút až po množstvo hodín a počas tejto zmeny nám do systému môže prísť ľubovoľné množstvo údajov, ktoré sa môžu upraviť, vymazať a samozrejme dochádza aj k vloženiu dát.

Všetky nové dáta spojené s operáciu *insert* sú automaticky ukladané do S3 s názvom transformácie do priečinku *new.txt*. Po skončení transformácie nahromadených dát sme využili skript napísaný v jazyku python, ktorý nám umožní spojiť súbory, ktoré sú výsledkom transformácií spodnej vetvy, čiže vetvy s nahromadenými dátami a hornej vetvy s dátami, ktoré do procesu prichádzali počas transformačného procesu. Skript, ktorý sme k tomuto účelu vytvorili je na nasledujúcej adrese:

https://github.com/romanceresnak/real-time-data-capture/blob/master/merge.py

Amazon Glue spojí všetky súbory do 1 veľkého súboru s názvom *merge.txt* a následne pomocou Amazon Atheny dochádza k úprave údajov, ktoré sú odchytené službou Lambda. Amazon Athena hodnoty, ktoré sa nachádzajú vo fronte Lambdy funkcie porovnáva s hodnotami v súbore na rovnakom princípe ako relačná databáza. V prípade, ak sa vo fronte nachádza operácia *update*, ktorá sa nachádza v súbore, tak je táto zmena vykonaná, správa z frontu je odstránená, a takto sa pokračuje až pokiaľ sa rad nevyprázdni. Po vyprázdnení frontu nasleduje automatické ukladanie hodnôt do databázy DynamoDB.

Pre konfiguráciu servera sme využili nasledujúcu konfiguráciu:

Tabuľka 11. Konfigurácia servera

Oracle Instance	m4.2xlarge
EMR cluster	master : 1x m3.xlarge core: 2x m4.4xlarge
DynamoDB	2000 write capacity unit

Pre dáta sme využili nasledujúcu konfiguráciu:

Tabuľka 12. Tabuľka dát

Počet záznamov	1,000,000 500,000,000 1,000,000,000
Database file size (.lbc)	4,9 GB 52,6 GB 108,4 GB
DynamoDB	2000 write capacity unit

Pre získanie časov na zmenu dát sme získali nasledujúce hodnoty:

Tabuľka 13. Tabuľka výkonnosti

Export to CSV	32 sec 4 min and 30 sec 6 min and 50 sec
Upload to S3 (sync)	22 sec 1 min and 48 sec 3 min and 30 sec
Import to DynamoDB	-

Počas ukladania už nie je možné dáta ďalej upravovať a hodnoty, ktoré počas tohto ukladania do procesu prídu prechádzajú samotnou vetvou do DynamoDB. Na vykonanie spomenutej operácie sme vytvorili ďalší skript v pythone, ktorý nám pomáha zachytiť spomenuté dáta. Skript je možné vidieť na nasledujúcom odkaze: *https://github.com/romanceresnak/real-time-data-capture/blob/master/lambda.py* .

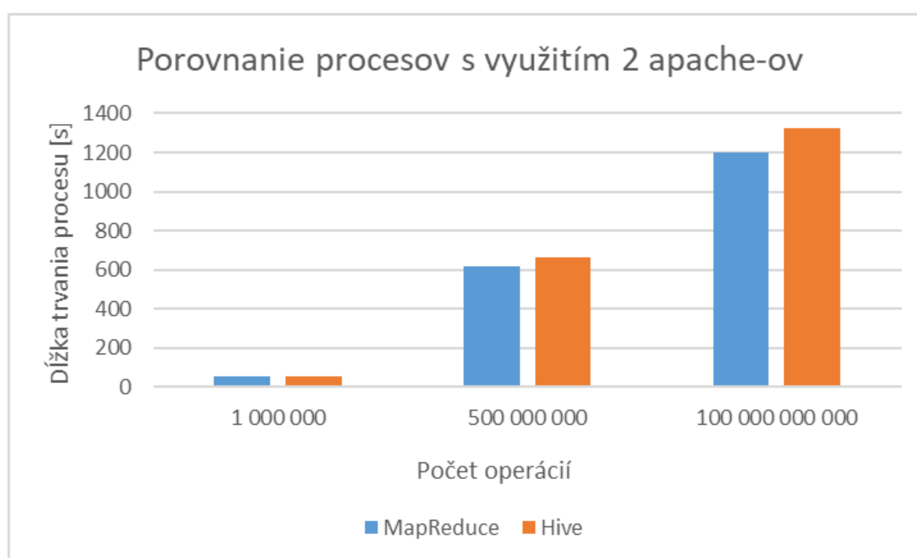
5.8.5 Experimentálna časť

Po vytvorení novej architektúry je dôležité otestovať správnosť fungovania rýchlosti a spoľahlivosti našej novo vytvorenej architektúry. Experimenty, ktoré sme naplánovali sú nasledovné :

- zistenie ako zmena apache-u ovplyvňuje rýchlosť procesu so zväčšujúcim sa počtom údajov,
- zistenie výhodnosti novo navrhnutého procesu oproti úprave dát po transformácii so zväčšujúcim sa počtom údajov.

Na experimentálnu činnosť sme využili dáta na nasledujúcej adrese https://massive.ucsd.edu/ProteoSAFe/dataset_files.jsp?task=7108e98682834ce48564dd7349b18235#%7B%22table_sort_history%22%3A%22main.collection_asc%22%7D

5.8.5.1 Porovnanie rýchlosti MapReduce vs. Hive



Obrázok 39. Efektívnosť metód MapReduce vs. Hive

Hodnoty, ktoré sme namerali majú z pohľadu budúceho výskumu pre nás rozhodujúce smerovania. Ako je vidieť z obr. 35 pri prvotnom množstve údajov (1 000 000), ktoré predstavuje stĺpec vľavo sme namerali pri spustenej transformácii hodnotu pre MapReduce 54 sekúnd a pre Apache Hive bolo nameraných 57 sekúnd. Táto hodnota súvisí s nutnosťou pretypovania, ktorá bola potrebná pri zmene štruktúry dát z relačnej databázy Oracle na nerelačnú databázu DynamoDB. Už pri takom malom množstve údajov ako je 1 000 000 záznamov bolo vidieť malú zmenu. Pri spustenom transformačnom procese, kedy bolo v relačnej databáze Oraclu už 500 000 000 záznamov, sa rozdiel prejavil výraznejšie. Ako je vidieť na obr. 39, tak v prostrednom stĺpci pre počet záznamov 5 000 000 (stĺpec vpravo) zobrazuje hodnotu, potrebnú na vykonanie transformačného procesu Apache-om Hive. Rozdiel, ktorý sme získali s využitím Apache-u Hive bol o 40 sekúnd časovo neefektívnejší ako s použitím technológie MapReduce. Na základe predchádzajúcich výsledkov, ktoré sme dosiahli pri transformačnom procese s 1 000 000 a 500 000 000 záznamami sme nepredpokladali, že s narastajúcim počtom záznamov sa pre nás stane Apache Hive niekedy efektívnejší.

Našu premisu sme podrobili aj tak experimentu, kedy transformačný proces mal za úlohu spracovať 100 000 000 000 záznamov. Pri dosiahnutí výsledkov sme dostali predpokladaný záver, ktorý ukázal, že aj pri počte záznamov o veľkosti 100 000 000 000 sa pre nás stále metóda MapReduce, respektíve Apache Hadoop javí stále časovo efektívnejšia ako Apache Hive. Výsledok 3 pokusu, v ktorom bola porovnávaná metóda Hive a metóda MapReduce ukázala, že MapReduce je efektívnejší o približne 124 sekúnd, čo je približne 2 minúty a 4 sekundy a jasne ukazuje prostriedok, ktorý bude pre nás v nasledujúcom skúmaní a ďalšom pokuse voľbou číslo jedna. Aj keď sa metóda MapReduce respektíve Apache Hadoop javí ako efektívnejšia, nemusí to pri všetkých typoch dát platiť. V situáciách, ktoré využívajú podporované dátové typy, ktoré podporuje Apache Hive by výsledky mohli byt diametrálne odlišné.

5.8.6 Zistenie výhodnosti novo navrhnutého procesu oproti úprave dát po transformácii so zväčšujúcim sa počtom údajov

Tabuľka 14. Nameraný čas potrebný na transformáciu údajov pomocou našej architektúry a bez našej architektúry pomocou technológie Hadoop

Experiment	Počet záznamov	Navrhnutá architektúra	Bez našej architektúry	Počet operácií
(1)	1,000,000	1 minúta	50 sekúnd	3
(2)	10,000,000	6 minút	6 minút and 10 sekúnd	18
(3)	500,000,000	12 minút	13 minút and 48 sekúnd	36
(4)	1000,000,000	20 minút	22 minút and 24 sekúnd	60

K testovacím potrebám sme použili nasledujúce operácie:

```
UPDATE customer  SET quantity = 55
WHERE customer_id = 55;
```

```
DELETE customer WHERE order_id = 50;
```

```
INSERT INTO customer
(customer_id, order_id, product_id, quantity)
VALUES (5000, 2345,234,356);
```

Hodnoty atribútov *quantity, customer_id* a *order_id* boli náhodné v závislosti od veľkosti dát, ktoré do transformačného procesu prichádzali. Tieto operácie boli do databázy posielané každých 20 sekúnd. V tabuľke 9 stĺpec úplne vpravo zobrazuje počet operácií, ktoré do procesu prišli počas transformácie. Pri porovnaní hodnôt, ktoré sme namerali, máme možnosť konštatovať, že pri prvom experimente (1) naša navrhnutá architektúra, nebola z pohľadu rýchlosti efektívnejšia ako pri procese, keď sa úpravy

vykonávajú až v databáze. V tomto prípade sa počet operácií, ktoré do systému vstúpia rovná počtu 3, čo nie je pre nás výhodné z pohľadu spájania súborov s nahromadenými údajmi a novými údajmi a následnou úpravou. S narastajúcim počtom údajov, ktoré do systému prišli (experiment (2), (3) a (4)) sa efektivita nášho riešenia prejavila výraznejšie, a s týmito výsledkami môžeme konštatovať, že našu metódu je výhodné použiť z dôvodu zvýšenia efektivity. S navrhnutou architektúrou je možné odbremeniť používateľa od nutnosti vykonávať dodatočné úpravy po skončení transformačného procesu.

Ako je vidieť z tabuľky 15, tak sledované hodnoty, ktoré sú pre nás podstatné a znamenajú pre nás pokrok oproti konvenčnej metóde sú zvýraznené zelenou farbou a negatívne aspekty našej metódy sú zvýraznené červenou farbou.

Zavedenie našej metódy do procesu môže v prípadoch s malým počtom ovplyvnení spôsobovať pravý opak účelu, na ktorý bola metóda vytvorená a práve tento proces spomaliť. Ako je vidieť z riadku 10, z tabuľky 15, tak pri definovanom počte záznamov sa efektívnosť metódy neprejavila.

Po aplikovaní nami navrhnutej metódy a zväčšenia počtu záznamov a samozrejme aj počtu ovplyvnení sa efektívnosť metódy prejavila až pri počte záznamov 10 000 000. Nakoľko spracovanie a vyhľadávanie vo veľkom množstve záznamov nie je časovo efektívne, tak prvotná kontrola dokázala redukovať počet operácií o 47 percent, čo je takmer 50 percentná úspešnosť. Celková úspešnosť prvotnej kontroly závisí na momente vstupu dát, ktoré prichádzajú do systému. Nemôžeme už transformované dáta, ktoré sa v procese transformácie už nenachádzajú ovplyvniť operáciami, ktoré do systému práve vstúpili.

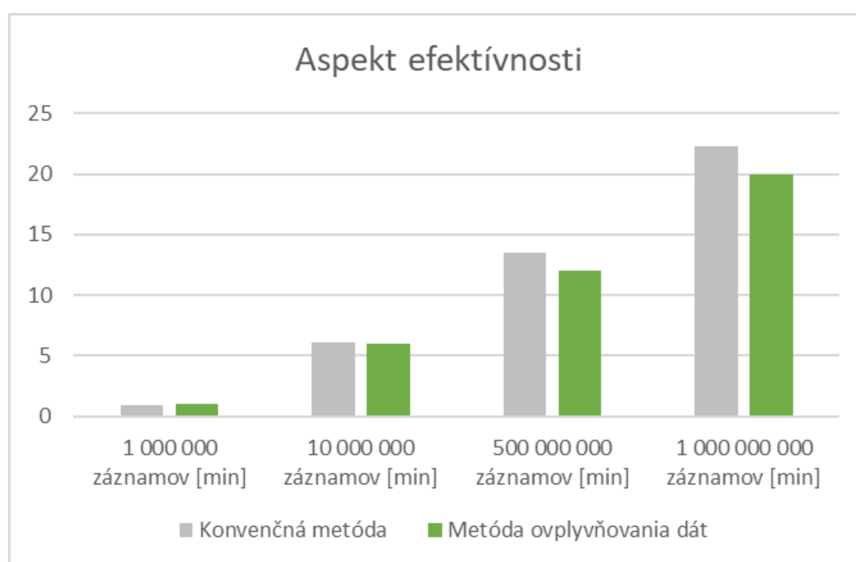
Tabuľka 15. Porovanie sledovaných hodnôt medzi konvenčnou metódou a metódou ovplyvňovania dát

Číslo vlastnosti	Sledovaná vlastnosť	Konvenčná metóda	Metóda ovplyvňovania dát
1	Kontrola pri výskyte [Áno/Nie]	Nie	Áno
2	Kontrola po skončení procesu [Áno/Nie]	Áno	Áno
3	Riešenie kolízií [%]	Nie	Áno
4	Viacnásobná kontrola	Nie	Áno
5	Podpora príkazu výberu údajov	Nie	Áno
6	Podpora príkazu aktualizovania údajov	Nie	Áno
7	Podpora príkazu vymazania údajov	Nie	Áno
8	Podpora príkazu vloženia údajov	Nie	Áno
9	Možné oneskorenie [Áno/Nie]	Nie	Áno
10	1 000 000 záznamov [min]	0,9	1
11	10 000 000 záznamov [min]	6,1	6
12	500 000 000 záznamov [min]	13,48	12
13	1 000 000 000 záznamov [min]	22,24	20

Ako je vidieť z grafu na obr. 40**,** tak grafické zobrazenie výsledkov ukazuje na začiatku procesu výhodu konvenčnej metódy. Zelenou farbou sú zobrazené výsledky po implementovaní nášho prístupu a šedou sú zobrazené výsledky s použitím konvenčnej metódy.

Ako je vidieť na obr. 40, kde nám zelená farba predstavuje hodnoty po aplikovaní našej metódy (vyšší stĺpec znamená horší výsledok), tak sme v prvej skupine prvkov, konkrétne 1 000 000 údajov dosiahli horší výsledok. Nakoľko tento proces trval krátky čas a pri spomenutom počte nedošlo k výraznému počtu ovplyvňovaní, tak nám dvojnásobná kontrola spôsobila zdržanie celého procesu.

Pri spracovaní väčšieho počtu údajov, konkrétne 10 000 000 sa krivka „zlomila“ v prospech nami navrhnutej metódy, a tento proces ďalej pokračuje aj so zvyšovaním počtu údajov, ktoré prichádzajú do systému počas transformačného procesu.



Obrázok 40. Grafické zobrazenie aspektu efektívnosti po aplikovaní metódy ovplyvňovania dát

Na základe výsledkov, ktoré sme získali pri experimentálnej činnosti vidíme rozdiely pri základných operáciách v relačných a nerelačných databázach. Pri experimentoch sme si všimli operáciu vloženia záznamov a následného výberu týchto dát.

Pri prvotnom povšimnutí tohto problému sme si uvedomili neefektívnosť takéhoto procesu. Predstavme si situáciu, pri ktorej sa používateľ zaregistruje do aplikácie a následne sa chce hneď do aplikácie aj prihlásiť. V takomto procese by bolo nutné vykonať 2 operácie.

Na základe tejto situácie sme usúdili, že je vhodné určité dáta ukladať do medzipamäte, a tým pádom spomenutej situácii zamedziť, čo nám umožní zrýchliť čas potrebný na spomenutú operáciu (registrovanie používateľa (insert) a následné prihlásenie používateľa (select).

6 Vyhľadávanie v nerelačných databázach

Rýchlosť vyhľadávania dát v nerelačných databázach je ovplyvnená 2 faktormi. Prvým faktorom je typ nerelačnej databázy, ktorá na základe typu štruktúry dokáže ovplyvniť rýchlosť získaných dát. S prvým faktorom je spojená aj možnosť vytvárania sekundárnych indexov. Druhý faktor súvisí s možnosťou ukladania potrebných dát do pamäte.

6.1 Vyhľadávanie na disku

Výkon databázy závisí od jej fyzického návrhu. Podstatnou časťou fyzického návrhu je výber správnej sady indexov týkajúcich sa konkrétnej pracovnej záťaže. Vytvorením indexu ako štruktúry môžeme zrýchliť dopytovanie sa k dátam, ale vytvorením veľkého množstva indexov je možné proces výberu dát aj spomaliť.

Na riešenie spomenutého problému existuje niekoľko optimalizačných metód. Niektoré nedávne prístupy [15] zanedbávajú aspekt spojený s obmedzením ukladania a namiesto toho vypočítajú dolnú hranicu nákladov na pracovné zaťaženie na základe individuálneho optimálneho indexu každého dotazu.

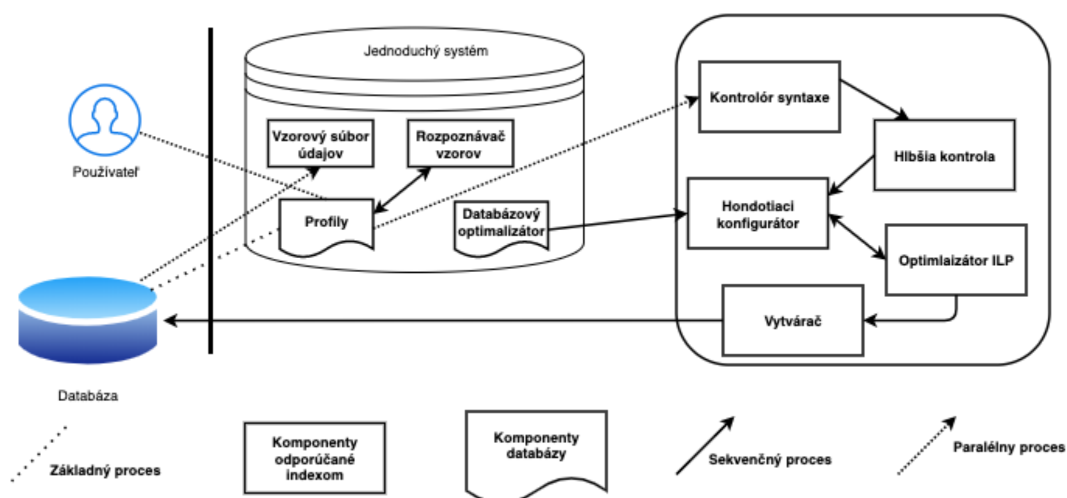
Viacerí výskumníci sa zaoberali danou problematikou a publikovali niekoľko článkov venujúcich sa optimalizačným metódam na nájdenie optimálneho riešenia tohto problému, napríklad s použitím algoritmu. Použitie problému s batohom (Knapsack problem usage) [103], generický algoritmus (generic algorihtm) (Kratica J., 2003) alebo dokonca techniky optimalizácie lineárneho programovania [17] viazané na vetvy [23]. Vo väčšine prípadov je tento problém riešený pomocou chamtivého algoritmu (*greedy algorithm*) [4], [7] [5].

V moderných databázach existujú nielen tradičné vzostupné a zostupné typy indexov, ale aj veľa nových typov indexov, napr. priestorové, textové indexy a podobne. Sú to typy indexov v databáze s ***n*** atribútmi v ***n*** kolekciách. Nasledujúca rovnica udáva počet možných indexov s jedným a viacerými atribútmi v tejto kolekcii:

$$\sum_{n=1}^{n} * \left(\frac{(s^k)n!}{(n-k)!} \right) \quad (4)$$

kde ***k*** je počet polí a $k \leq n$ [5]. V kolekcii sa nachádza iba päť atribútov a možnosť vytvoriť štyri typy indexov. Počet možných indexov presahuje 150 000.

Mnohé štúdie sa rozhodli využívať namiesto chamtivého algoritmu (*greede algorithm*) metódu ILP (*Integer Linear Program*), pretože to nielen umožňuje, aby sme preskúmali viac prípadov, ako napríklad chamtivý algoritmus, ale tiež umožňuje vyhodnotiť kvalitu optimálneho riešenia. Aplikácia relaxácie lineárneho programovania nám umožňuje získať užitočné informácie o riešeniach, ktoré mali optimálny výkon, ale z dôvodu nedostatku úložného priestoru neboli brané do úvahy. Na porovnanie efektívnosti chamtivého algoritmu sme využili metódu IRS(*Index Recommendation System*), ktorá na základe otestovaných indexov poskytuje usmernenia, čo sa týka efektívnosti jednotlivých metód.



Obrázok 41. Architektúra systému Index odporúčaný systémom

6.2 Vyhľadávanie v pamäti

Tendencia ukladania dát sa časom menila, a z dôvodu nedostatočne rýchlych jednotlivých operácií je dnešným trendom zmena jedného typu databázy na iný typ. Presun respektíve výmena databázy má jeden hlavný účel, a tým je zefektívnenie operácií výber (*select*), vkladanie (*insert*), aktualizovanie dát (*update*) alebo zmazanie záznamov (*delete*).

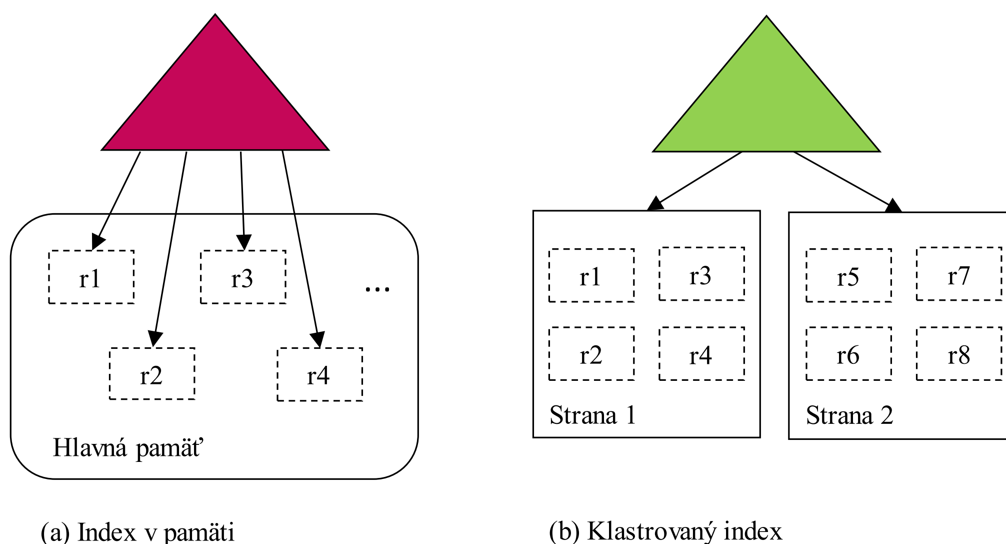
Primárnym úložiskom pre záznamy v databázach s hlavnou pamäťou je RAM. Tieto systémy nie sú obmedzené potrebou stránkovania údajov na disku na požiadanie počas spracovania dotazu. Z tohto dôvodu sa moderné databázy hlavnej pamäte vyhýbajú presmerovaniu na stránke prostredníctvom oblasti vyrovnávacej pamäte.

Tieto systémy nepoužívajú identifikátory logických záznamov formulára (id stránky, ofset) ako sa to robí v tradičných databázových systémoch. Namiesto toho je bežnou praxou, že hlavná pamäť používa ukazovatele v pamäti na priamy prístup k záznamom.

Moderné systémy s hlavnou pamäťou vyvolali nový záujem o vysoko výkonné metódy indexovania [76]. Indexy nie len v databáze v pamäti, ale aj relačných a nerelačných databázach majú za úlohu zrýchliť získavanie údajov. Indexová štruktúra, ktorá sa používa v relačnej, nerelačnej a databáze v pamäti je založená na B+ strome.

V databázach s hlavnou pamäťou je organizácia indexov trochu odlišná od organizácie používanej systémom založeným na diskoch. Po prvé, neexistuje presmerovanie založené na stránke, preto sa indexové uzly nemusia mapovať na databázovú stránku. Napríklad index radixov ART používaný v *HyPer* používa pre svoje indexové uzly štyri samostatné triedy veľkostí. Podobne index Bw-strom používaný v Hekatone prideľuje špecifickú pamäť pre každý z uzlov, ktorá je potrebná na uloženie jeho kľúčov a užitočného zaťaženia, nezanecháva nevyužitý priestor v uzle [7] [76]. Ďalej strom Bw využíva flexibilnú politiku rozdelenia, výber rozdelenia uzlov stromu B+ , keď je to vhodné, nie keď veľkosť uzlov dosiahne pevný prah ako v indexoch založených na disku.

Bežnou záležitosťou je, že indexy hlavnej pamäte ukladajú priame ukazovatele na záznamy, nie na identifikátory logických záznamov alebo primárne kľúče (v prípade sekundárnych indexov), ako sa to robí v diskových systémoch [7], [33]. Obr. 42 poskytuje vizuálny príklad tohto rozdielu. Obr. 42 (b) zobrazuje skupinový index založený na disku, kde index ukazuje na dátové (listové) stránky, ktoré ukladajú záznamy v usporiadanom poradí. Na druhej strane, obr. 42 (a) zobrazuje prístup k hlavnej pamäti, ktorý indexuje záznamy uložené v pamäti bez akéhokoľvek konkrétneho zoskupovania alebo organizácie.



Obrázok 42. Porovnanie štruktúr indexov v pamäti a na disku

6.3 Porovnanie súčasných databáz

V súčasnosti sa mnohé spoločnosti rozhodli k presunu z relačnej databázy na nerelačnú s odôvodnením, že nerelačné databázy sú pri každej operácií rýchlejšie a lepšie v porovnaní s relačnou databázou. Preto sme sa na začiatku tohto výskumu rozhodli otestovať správnosť výroku a pozreli sme sa na rozdiely medzi relačnými a nerelačnými databázami.

6.3.1 Základné informácie o databázach

Databáza (zriedkavo dátová banka) je sada štruktúrovaných údajov alebo informácií uložených v počítačovom systéme. Ľubovoľný počítačový program alebo osoba môže na vyhľadanie informácií použiť vyhľadávací jazyk. Takto získané informácie môžu byť použité v rozhodovacom procese. Počítačový program používaný na správu údajov a na dotazy sa označuje ako *SRBD* (Database Management System). Vedci v oblasti počítačov môžu klasifikovať systém správy databáz podľa podporovaných databázových modelov. Prvé zmienené databázy (relačné) sa stali dominantnými v 80. rokoch. Podporujú použitie riadkov a stĺpcov v sérii tabuliek. Najvýznamnejšia väčšina z nich tiež používa Sql na zápis a dopytovanie údajov. Nerelačné databázy sa stali populárnymi v 20. rokoch 20. storočia a nazývajú sa NoSql, pretože používajú iný dotazovací jazyk.

6.3.1.1 *Sql vs. NoSql*

Najvýznamnejším rozdielom medzi týmito konceptmi je, že databáza Sql je relačná a obsahuje cudzie kľúče. Naopak, databáza NoSql je nerelačná, a tak nedefinuje vzťahy. Tabuľka 16 ukazuje rôzne vlastnosti databáz Sql a NoSql [75].

Tabuľka 16. Rozdiel medzi relačnými a nerelačnými databázami

Vlastnosť	**Sql**	**NoSql**
Spôsob ukladania údajov	Tabuľky	Dokumenty, kľúčová hodnota
Organizácia údajov	Preddefinovaná schéma	Dynamická schéma
Škálovateľnosť (zvýšenie výkonu)	Vertikálne (silnejší procesor)	Horizontálne (viac serverov, inštancií)
Dotazovací jazyk	Štandardizovaný Sql	Vlastný dotazovací jazyk
Dátový styk	Cudzie kľúče	Vnorené dokumenty
Bezpečnosť	Transakcie, dôslednosť, izolácia	Neexistuje

Na prvý pohľad sa môže zdať, že Sql prevažuje nad výhodami NoSql, ale nemusí to platiť automaticky. NoSql nám neponúka žiadne bezpečnostné prvky, ale viac číta a zapisuje údaje. Preto sa používa pre aplikácie veľkých dát (BigData), kde sa očakávajú terabajtové údaje. NoSql je tiež ideálnou voľbou, ak potrebujeme implementovať komplexný fulltextový vyhľadávací modul, ktorý zohľadňuje podobné slová, mimiku alebo iné gramatické pravidlá. NoSql tiež rieši problém, keď najskôr nepoznáme databázové schémy. Naopak, musíme sa zaoberať konzistenciou údajov na strane aplikácie, a to je nevýhoda. Ďalšou nevýhodou je, že databáza NoSql nepodporuje transakčné spracovanie [78].

Existuje oveľa viac typov databáz NoSql ako len tie, ktoré sa zaoberajú fulltextovým vyhľadávaním. Líšia sa štýlom ukladania údajov, ako aj používaním. V súčasnosti poznáme databázové typy akými sú *dokument*, *kľúč - hodnota* alebo *graf* databázy NoSql. Pokiaľ ide o fulltextové vyhľadávanie, hovoríme o databázach dokumentov. Používame databázu typu kľúč - hodnota(*key-value*), pretože keď pracujeme s vyrovnávacou pamäťou, musíme niekde údaje uložiť. Medzi najrozšírenejšími typmi databáz pracujúce v pamäti zaraďujeme Redis alebo Memcached. Sociálne siete sa najskôr zobrazia v databáze grafov [84].

Rozdiely medzi relačnými a nerelačnými databázami sú jasné, avšak aj medzi relačnými databázami existujú rozdiely, ktoré ich od seba odlišujú a sú znázornené v tabuľke 17.

Tabuľka 17. Rozdielne vlastnosti medzi 3 najpoužívanejšími relačnými databázami

Vlastnosti	Microsoft Sql	Oracle RDBMS	MySql
Typické aplikácie	SharePoint, SCOM, SCCM, WSUS	OBI, SAP	Joomla, WordPress, MyBB, phpBB, Drupal, veľa open-source
Operačný systém	Windows Server, klient Windows	Windows, Unix, Linux	Windows, Unix, Linux, Mac a mnoho ďalších
Vodiči (Drivers)	ODBC, JDBC, ADO.NET, OLEDB, Microsoft Visual Studio	-	ODBC, JDBC, ADO.NET, Microsoft Visual Studio
Licencovanie	Uzavretý zdroj, vlastnícke	Uzavretý zdroj, vlastnícke	Open-source GNU-GPL
Štandardizované	ANSI-SQL		ANSI-SQL
Transakcie	Áno	Áno	Používa sa úložný modul InnoDB
Čiastočný index	Áno	Áno	Nie
Schéma	Áno	Áno	Nie
Zlyhanie			Používanie MyISAM: Vyžaduje sa UPS, predpokladá sa neprerušovaná prevádzka
Nástroje na grafickú správu	Áno: Management Studio a BI Studio	Enterprise Manager	MySQL Workbench. Ropucha.
Vypočítané stĺpce	Áno	Áno	Nie
Aktívne / aktívne klastrovanie	Iba na čítanie v druhom uzle	Áno (RAC)	Nie
Sprievodca plánom údržby	Áno		Nie
Plánovanie práce	Áno (agent)	Áno (Oracle Scheduler)	v5.1 (Plánovač udalostí)
História	Prvé vydanie v roku 1989, založené na Ingress (1974) / Sybase (1987)	1979	1995

6.3.1.2 Oracle vs. Sql Server vs. MySql

Okrem funkcií, na ktoré upozorňuje vyššie uvedená tabuľka 17, existuje niekoľko ďalších bodov, na základe ktorých môžeme tieto tri databázy porovnať. Uviedli sme ich nižšie:

- Funkcia zápisu je k dispozícii vo všetkých troch databázach a podporuje XML a sekundárne indexy.
- Bežné API pre každú z databáz sú ADO.NET, JDBC a ODBC. Zatiaľ čo Sql Server podporuje aj OLE DB a TDS, Oracle podporuje aj ODP.NET a OCI.
- V prípade Oracle existuje dlhý zoznam podporovaných programovacích jazykov akými sú Delphi, Lisp, Java, C ++, C #, Ruby, PHP, Visual Basic a JavaScript, ktoré tiež podporujú MySql a Sql Server. Oracle podporuje mnoho ďalších programovacích jazykov, ktoré zvyšné dve databázy nepodporujú. Sú to jazyky ako Scala, Fortran a ďalšie jazyky.
- Sql Server používa na skriptovanie na strane servera transakcie Sql a .NET, zatiaľ čo Oracle používa jazyky PL / Sql.
- Spúšťače sú dostupné vo všetkých troch databázach.
- Všetky databázy podporujú koncepciu súbežnosti, trvanlivosti, správu pamäte a koncepcie cudzích kľúčov.

- Podporujú vlastnosti ACID, čo sa týka koncepcií transakcií.
- V systémoch Oracle a MySql môže mať replikáciu Master-Master a Master-Slave pre stratégiu replikácie a závisí to od verzie na serveri Sql Server.

Existujú tri ďalšie faktory, na základe ktorých môžeme tieto databázy porovnávať. Jazyk, ktorý používajú ako hlavnú vlastnosť ľubovoľného RDBMS, je jazyk používaný na vykonávanie dotazov má vplyv na výkon databázy. Aj keď všetky tri databázy používajú jazyk Sql alebo štruktúrovaný dotazovací jazyk, Sql Server používa aj T-Sql vyvinutý spoločnosťou Sybase (rozšírenie Sql). Spoločnosť Oracle používa PL / Sql alebo procedurálny programovací jazyk.

Oba jazyky majú rozdielnu syntax a možnosti. Hlavný rozdiel medzi týmito jazykmi spočíva v spôsobe zaobchádzania s uloženými procedúrami, premennými a vstavanými funkciami. V prostredí Oracle PL / Sql možno postupy tiež zoskupiť do balíkov, ktoré sa nedajú vykonať na serveri Sql Server. Preto môže byť postupy trochu zložitejšie a oveľa výkonnejšie. Implementácia T-Sql je jednoduchšia, ale na druhej strane MySql používa odľahčenú verziu T-Sql a kombináciu procedurálnych jazykov.

S každým novým pripojením databázy sa v systéme Oracle zaobchádza ako s novou transakciou. Pre všetky dotazy a vykonávanie príkazov sa zmeny vykonajú iba v pamäti zostávajúcej v pamäti *cache*. Nič nie je vykonané bez výslovného príkazu *COMMIT*. Zmeny vykonané v databáze zostanú v pamäti *cache* a spočiatku sa vykonajú v pamäti. Keď sa vykoná *ZÁZNAM*, nová inštrukcia spustí novú transakciu. Týmto spôsobom sa vývojárom ponúka významnejšia transakčná funkcia a kontrolu chýb je tiež možné vykonať rýchlo.

V prípade MySql transakcie vybavuje InnoDB. InnoDB je ukladací modul a je predvolene k dispozícii v MySql. Poskytuje tiež funkcie na *ACID*, ako je podpora cudzieho kľúča a spracovanie transakcií.

6.3.2 NoSql databázy

Vývojári potrebujú riešenia, ktoré budú v súlade s realitou moderných údajov a postupmi vývoja iteračného softvéru. V posledných rokoch sa databázy NoSql objavili ako odpoveď na obmedzenia tradičných relačných databáz, a tiež ako zabezpečenie výkonu, škálovateľnosti a flexibility, ktoré sa vyžadujú v moderných aplikáciách [78].

Väčšina aspektov týchto technológií NoSql sa veľmi líši a má málo spoločného, až na to, že nepoužívajú relačný dátový model. Existujú štyri typy systémov na správu databáz NoSql:

- Typ s kľúčmi a hodnotami (Key-value Store) - má veľkú hashovaciu tabuľku kľúčov a hodnôt {Napríklad - Riak, Amazon S3 (Dynamo)}.
- Dokumentový typ (Document-based Store) - ukladá dokumenty zložené z označených prvkov. {Napríklad - CouchDB}.
- Typ založený na stĺpcoch (Column-based Store) - každý blok úložiska obsahuje údaje iba z jedného stĺpca, {Napríklad - HBase, Cassandra}.
- Graph-based- sieťová databáza, ktorá používa hrany a uzly na reprezentáciu a ukladanie údajov {Napríklad - Neo4J} [13].

V ďalšej časti práce sme popísali najzákladnejšie typy nerelačných databáz, ku ktorým patria:

6.3.2.1 Databáza typu kľúč / hodnota (Key-Value Store)

Formát bez schémy databázového typu kľúč - hodnota, ako je Riak je práve to, čo je nevyhnutné pre potreby úložiska. Kľúč môže byť opakovaný alebo automaticky generovaný, zatiaľ čo hodnotou môže byť typu *String*, *JSON*, *BLOB* (binárny veľký objekt), atď.

Typ kľúč - hodnota v zásade používa hashovaciu tabuľku, v ktorej je jedinečný kľúč a ukazovateľ na konkrétnu položku údajov. Vedro (*bucket*) je logická skupina kľúčov - ale fyzicky nezoskupujú údaje. V rôznych vedrách (*bucket-och*) môžu byť rovnaké kľúče.

Výkon je do značnej miery zvýšený kvôli mechanizmom vyrovnávacej pamäte, ktoré sprevádzajú mapovania. Je potrebné poznať kľúč aj segment, aby sme mohli načítať hodnotu, pretože skutočný kľúč je hash (Bucket + Key).

Databázový model *Key-Value Store* nie je nijako zložitý, pretože ho možno ľahko implementovať. Nie je to ideálna metóda, ak sa vyhľadáva iba aktualizácia časti hodnoty alebo dotaz v databáze.

Je zrejmé, že pri manipulácii s kľúčovými hodnotami dochádza k efektívnej manipulácii s objektami s dôrazom na aspekty ako sú dostupnosť a rozdelenia, ale dochádza ku strate konzistencie. Kľúč môže byť opakovaný alebo automaticky generovaný, zatiaľ čo hodnotou môže byť *String*, *JSON*, *BLOB* (binárny veľký objekt).

Táto databáza typu kľúč - hodnota umožňuje klientom čítať a zapisovať hodnoty pomocou kľúča nasledovne:

- ***Získať*** (kľúč) - vráti hodnotu spojenú s poskytnutým kľúčom.
- ***Put*** (kľúč, hodnota) - spája hodnotu s kľúčom.
- ***Multi-get*** (key1, key2, .., keyN) - vráti zoznam hodnôt spojených so zoznamom kľúčov.
- ***Vymazať*** (kľúč) - odstráni položku pre kľúč z úložiska dát.

Aj keď sa databáza typu kľúč - hodnota zdá byť v niektorých prípadoch užitočná, má aj niektoré slabé stránky. Model spočiatku nebude poskytovať žiadne tradičné databázové možnosti (ako napríklad atomicita transakcií alebo konzistencia, keď sa vykonáva viac transakcií súčasne). Samotná aplikácia musí také schopnosti poskytovať [28].

Po druhé, so zvyšujúcim sa objemom údajov môže byť udržiavanie jedinečných hodnôt, ako sú kľúče, zložitejšie. Riešenie tohto problému si vyžaduje zavedenie určitej zložitosti do generujúcich znakových reťazcov, ktoré zostanú medzi obrovskou sadou kľúčov jedinečné.

6.3.2.2 Databáza založená na dokumentoch

Dáta, ktoré sú kolekciou párov *kľúč - hodnota*, sú komprimované ako úložisko dokumentov dosť podobné úložisku kľúč - hodnota, avšak jediný rozdiel je v tom, že uložené hodnoty (označované ako „dokumenty“) poskytujú určitú štruktúru a kódovanie spravovaných údajov. *XML*, *JSON* (JavaScript Object Notation), *BSON* (čo je binárne kódovanie objektov JSON) sú niektoré bežné štandardné kódovania.

Nasledujúci príklad ukazuje hodnoty údajov zhromaždené ako „dokument“ predstavujúci názvy konkrétnych maloobchodných predajní. Upozorňujeme, že zatiaľ čo všetky tri príklady predstavujú umiestnenia, reprezentatívne modely sa líšia.

```
    {officeName:”Europalace”, {Street: “Vysokoskolakov, City:”Zilina”, Pincode:”95802”} }
    {officeName:”Mirage”, {Street:”Mestska”, Block:”B, Ist Floor”, City: “Topolcany”, Pincode: 59682”}
}
    {officeName:”Eurovea”, {Latitude:”40.748328”, Longitude:”-73.985560”} }
```

Príklady databázy dokumentov sú:

- MongoDB
- Couchbase

6.3.2.3 Databáza založená na stĺpcoch

V stĺpcovo orientovanej databáze NoSql sú údaje uložené skôr v bunkách zoskupených v stĺpcoch s údajmi, ako v riadkoch s údajmi. Stĺpce sú logicky zoskupené do skupín stĺpcov. Skupiny stĺpcov môžu obsahovať prakticky neobmedzený počet stĺpcov, ktoré je možné vytvoriť za behu programu alebo definíciu schémy. Čítanie a zápis sa vykonáva pomocou stĺpcov a nie riadkov, ako je to napríklad u relačných databáz.

Výhodou ukladania údajov do stĺpcov je rýchle vyhľadávanie / prístup a agregácia údajov. Relačné databázy ukladajú jeden riadok ako súvislý disk. Rôzne riadky sú uložené na rôznych miestach na disku, zatiaľ čo stĺpcové databázy ukladajú všetky bunky zodpovedajúce stĺpcu ako súvislý záznam na disk, a tak rýchlejšie vyhľadávajú / získavajú prístup.

Napríklad: Dopytovanie nadpisov z množstva miliónov článkov bude náročnou úlohou pri používaní relačných databáz, pretože bude prechádzať cez každé miesto, odchádzať od názvov položiek. Na druhej strane, názov všetkých položiek je možné získať iba s prístupom na jeden disk.

Dátový model:

- ColumnFamily: ColumnFamily je jedna štruktúra, ktorá umožňuje ľahké zoskupovanie stĺpcov a supercolónov.
- Kľúč: trvalý názov záznamu. Kľúče majú rôzny počet stĺpcov, aby bolo možné databázu nepravidelne zmenšiť.
- Najznámejšie príklady sú BigTable spoločnosti Google a HBase & Cassandra inšpirované BigTable.
- Napríklad BigTable je komprimovaný vysoko výkonný a patentovaný systém na ukladanie dát, ktorý vlastní spoločnosť Google.

Má nasledujúce atribúty:

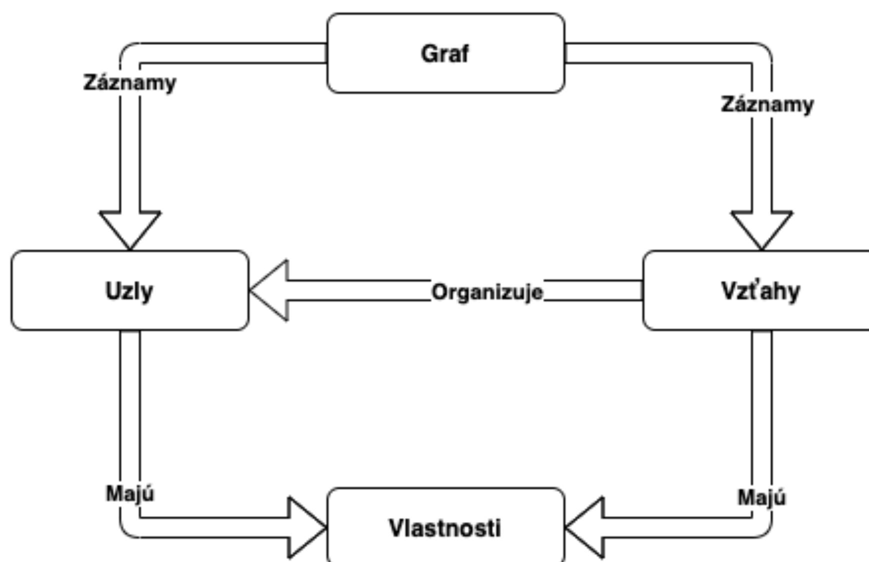
- *Riedke* - niektoré bunky môžu byť prázdne,
- *Distribuované* - dáta sú rozdelené na mnohých hostiteľov,
- *Multidimenzionálne* - viac ako jedna dimenzia,
- *Zoradené* - mapy sa spravidla netriedia, ale táto je.

6.3.2.4 *Databáza NoSql založená na grafoch*

V databáze založenej na grafoch nie je možné nájsť rigidný formát Sql alebo reprezentáciu tabuliek a stĺpcov. Namiesto toho sa použije flexibilné grafické znázornenie, ktoré je ideálne na riešenie problémov so škálovateľnosťou. Štruktúry grafov sa používajú s hranami, uzlami a vlastnosťami a poskytujú bezindexovú susednosť. Dáta možno ľahko transformovať z jedného modelu do druhého pomocou databázy Graph Base NoSql [71].

Tieto databázy používajú okraje (*edges*) a uzly (*nodes*) na reprezentáciu a ukladanie údajov ako je zobrazené na obr. 38.

- Niektoré vzájomné vzťahy organizujú tieto uzly, čo predstavuje hrana medzi uzlami.
- Uzly aj vzťahy majú niektoré definované vlastnosti



Obrázok 43. Organizácia v NoSql databáze založenej na grafoch

Niektoré funkcie grafovej databázy sú vysvetlené na príklade nižšie:

Označený, usmernený a pripisovaný multi-graf: Graf obsahuje uzly označené správne niektorými vlastnosťami. Tieto uzly majú určitý vzájomný vzťah, ktorý je znázornený smerovými hranami.

6.3.3 Dôvod skúmania danej problematiky

Pri prvotnom skúmaní sme potrebovali získať komplexnejší pohľad k operáciám z relačných/nerelačných databáz výberu dát (*select/find()*), vloženia záznamov (*insert/insert()*), aktualizovaniu hodnôt (*update/update()*) a vymazaniu záznamu (*delete/remove()*).

S nutnosťou preskúmania viacerých relačných a nerelačných databáz sme potrebovali získať hodnoty, ktoré budú otestované a namerané na rovnakých typoch databáz, v rovnakom počítači, respektíve serveri a pri rovnakých konfiguračných nastaveniach, aby nedošlo k žiadnemu skresleniu hodnôt a my sme mohli vyvodiť potrebné závery.

K aktuálnej problematike sa vyjadrilo viacero odborníkov a publikovali mnohé štúdie [71] [84], avšak mnohé z publikovaných štúdií sa venovali porovnaniu iba relačných, nerelačných, prípadne porovnávali jednu relačnú a nerelačnú databázu. Z dôvodu komplexnosti riešenia sme získali dáta o dopravných spojeniach, ktoré sme následne vložili do viacerých relačných a nerelačných databáz a otestovali sme spomenuté operácie vyššie.

6.3.4 Experimenty súvisiace s rýchlosťou základných operácií pri relačných a nerelačných databázach

V snahe získať komplexnejšie výsledky sme museli zistiť trvanie pre jednotlivé operácie akými sú operácie vloženia (*insert*), aktualizovania (*update*), zmazania (*delete*) a výberu (*select*) údajov. Aj keď sme pri skúmaní našli viaceré výsledky spomenutých operácií nikdy sme nenašli výsledky, ktoré by porovnávali nami vyžadované databázové typy alebo boli porovnávania vykonané pri rôznej konfigurácií servera.

Z toho dôvodu sme sa rozhodli nakonfigurovať všetky databázové typy, ktoré sme uviedli v hlavičke tabuľky 18. Všetky tieto databázy boli nakonfigurované na rovnakú a default-nú hodnotu, ktorá je odporúčaná na oficiálnych stránkach každej zmienej databázy. Aby sme zamedzili istým výkyvom, resp. nežiadúcim skresleniam, vykonávali sme príkazy iba pri spustení 1 programu. Všetky experimenty sme uskutočnili na počítači MacBook pro rok 2015 s 8 GB RAM a procesorom I5.

Tabuľka 18. Výkon rôznych typov relačných a nerelačných databáz pre 10 000 záznamov nameraných v milisekundách

Typ DB/ Príkaz	Oracle	MySql	MsSql	Mongo	Redis	GraphQL	Cassandra
Vloženie	0.076	0,093	0.093	0.005	0.009	0.008	0.011
Aktualizovanie	0.018	0.017	0.018	0.021	0.025	0.0023	0.029
Zmazanie	0.059	0.025	0.093	0.01	0.021	0.017	0.018
Výber	0.015	0.016	0.017	0.023	0.024	0.022	0.025

Všetky operácie, ktoré boli sledované a zaznamenané v tabuľke 18, budú sledované aj pri vytváraní indexu nad požadovanými atribútmi. Experimenty odrážali vyhľadávanie autobusových alebo vlakových spojení ako je na stránke *http://www.slovakrail.sk/.* Na tejto webovej stránke osoba definuje miesto kam sa chce dostať, kde chce vystúpiť a dátum, kedy bude cestovať. Stránka poskytuje výsledok s mnohými spojeniami. Ak pripojenie neexistuje, na stránke sa zobrazí pripojenie s väčším počtom prestupov. Nad týmito údajmi sa vytvorí index pre rýchlejšie vyhľadávanie.

Všetky predchádzajúce testy sa uskutočňovali s rozsahom 10 000 záznamov. Rýchlosť a efektívnosť databázy NoSql sa prejavila pri vyššom počte záznamov, napríklad 10 000 a viac. Pre tieto potreby testovania efektivity sme vytvorili falošné záznamy a následne sme ich otestovali.

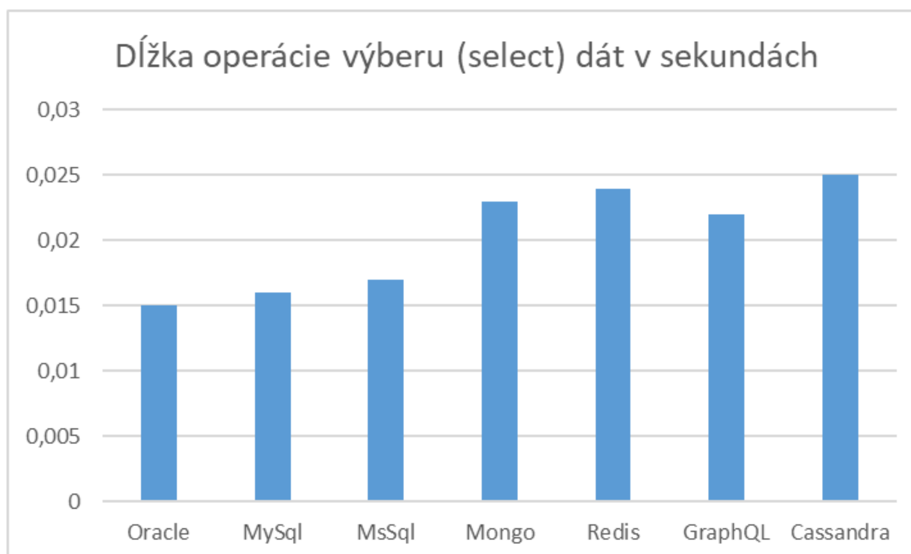
Vo všetkých tabuľkách máme 90 000 záznamov pri každom novom testovaní. Rozhodli sme sa použiť skenovanie rozsahu v tabuľke[7], ktoré je veľmi potrebné na urýchlenie dotazov. Všetky výsledky sme zdokumentovali v tabuľke 19.

Tabuľka 19. Výkon rôznych typov relačných a nerelačných databáz pre 100 000 záznamov nameraných v milisekundách

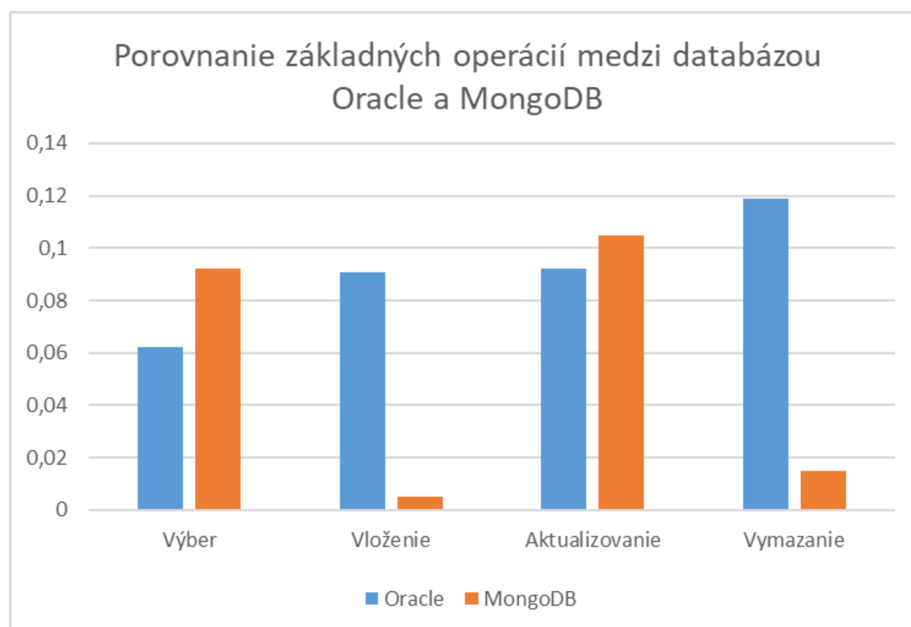
Typ DB/ Príkaz	Oracle	MySql	MsSql	Mongo	Redis	GraphQL	Cassandra
Vloženie	0.091	0.038	0.093	0.005	0.010	0.008	0.011
Aktualizovanie	0.092	0.068	0.075	0.105	0.102	0.120	0.104
Zmazanie	0.119	0.047	0.171	0.015	0.021	0.018	0.019
Výber	0.062	0.067	0.060	0.092	0.089	0.094	0.094

[7] Porovnanie výkonu jednotlivých databáz sme prezentovali a diskutovali na IEEE konferencii International Scientific conference on sustainable, modern and safe transport na Slovensku (Vysoké Tatry) - 29. - 31. máj 2019.

6.3.4.1 Výsledok experimentu graficky



Obrázok 44. Výkon dopytu v milisekundách



Obrázok 45. Výkon jednotlivých príkazov pre relačnú databázu Oracle a nerelačnú databázu MongoDB

Na obr. 45 je zobrazený čas výsledku pre operácie výberu (*select/find()*) v relačných databázach (Oracle, MySql a MsSql) a nerelačných databázach (Mongo, Redis, Cassandra a GraphQL). Tento výsledok sme očakávali. V NoSql, ako sú Redis, Mongo, Cassandra alebo GraphQL, sú dáta ukladané vo forme kolekcií, kde sú tieto dáta duplikované na rôznych miestach. Znamená to, že operácie čítania alebo zápisu na jednu entitu sa stali prístupnejšie a rýchlejšie [19].

Na druhej strane, vo vzťahu k databázam, je potrebné údaje rozdeliť do niekoľkých malých logických tabuliek, aby sa zabránilo duplicite a redundancii. Z tohto dôvodu je potrebné pomôcť normalizácii. Normalizácia umožňuje správne a efektívne spravovať údaje. Rozdelením údajov na niekoľko súvisiacich tabuliek, ktoré súvisia s normalizáciou, brzdí výkonnosť spracovania údajov v relačných databázach pomocou protokolu Sql, a preto je čas potrebný na získanie hodnoty z relačných databáz oveľa vyšší ako v prípade nesúvisiacich vzťahov, ako vidíme na obr. 45.

Pravý stĺpec (zobrazený modrou farbou na obr. 45) nám ukazuje, aké časy boli namerané pri relačnej databáze Oracle. Ľavý stĺpec predstavuje nerelačnú databázu Mongo. Na porovnanie, pomer medzi databázami pre príjem údajov je takmer 1:3 pre výkon vybranej aplikácie, 1:6 pre operáciu vymazania, 1:9 pre operáciu aktualizácie a 1:15 pre operáciu vloženia.

Pri porovnaní dvoch skupín nie je rozdiel medzi relačnými a nerelačnými databázami v čase dotazu diametrálne odlišný, ako je zrejmé z tabuľky 19. Rozdiel bol upravený po implementácii skenovania rozsahu na kľúčovom atribúte. V prípade veľkosti vlakovej stanice na Slovensku sa dá stále použiť relačná databáza. V prípade krajín s vyšším počtom železničných staníc a zastávok, ako sú Nemecko, Holandsko a ďalšie, by bolo efektívnejšie použitie nerelačných databáz.

Na základe výsledkov, ktoré sme získali pri experimentálnej činnosti vidíme jasne preukázateľné zlepšenie po zavedení našej metódy. Pri experimentoch so serverom vidíme rôznu náročnosť pri odlišných typoch databáz.

Pri prvotnom skúmaní sme chceli nad dátami v nerelačnej databáze zaviesť sekundárny index, prípadne využívať medzipamäť, čo by nám pomohlo pri zrýchlení vyhľadávania v nerelačných databázach. Výhodou nerelačných databáz, ale v tomto prípade je to pre nás nevýhoda, je voľnosť štruktúry.

Na základe situácie sme usúdili, že by stálo za zváženie vytvorenie prístupu, ktorý nám určitým spôsobom spravuje záznamy medzi primárnym dátovým úložiskom, vymieňa a spolupracuje s databázou v pamäti, a tým pádom zrýchľuje proces vyhľadávania.

6.4 Metóda na zefektívnenie vyhľadávania v nerelačných databázach

Zo záverov z predchádzajúcej kapitoly a odhalených nedostatkov sme navrhli riešenie spojené so správou jednotlivých údajov a vzájomnej správy medzi primárnou databázou a databázou v pamäti.

Podľa nášho názoru a testov je efektívnejšie spravovanie dát medzi nerelačnou databázou a databázou v pamäti, čím sa redukuje počet prístupov v primárnej databáze a na základe korektnej správy umožní výrazne eliminovať čas potrebný nielen na získanie údajov, ale aj na ich uloženie resp. zmazanie.

Spomenuté nedostatky a možnosti zrýchlenia procesu vyhľadávania nás viedli k stanoveniu cieľov a využitiu prostriedkov, ktorých cieľom je:

- spravovanie záznamov medzi primárnou databázou a databázou v pamäti,
- vytvorenie správy záznamov medzi primárnym dátovým úložiskom a databázou v pamäti,
- redukovanie počet prístupov do primárneho dátového úložiska.

V nasledujúcej kapitole preskúmame už existujúce riešenia, pochádzajúce z dielne mnohých autorov, venujúcich sa danej problematike.

6.4.1 Existujúce riešenia zaoberajúce sa zefektívnením vyhľadávania v nerelačných databázach

V rôznych prácach už bolo uvedené porovnanie relačných dátových modelov s nerelačnými dátovými modelmi. Napríklad medzi týmito dvoma typmi databáz sa zobrazujú a zaznamenávajú časy potrebné na vykonanie základných operácií, ako je výber údajov, vloženie údajov, aktualizácia údajov a vymazanie údajov. Niekoľko štatistík poukazuje na skutočnosť, že najbežnejšou operáciou, ktorá sa vyžaduje v relačnej a nerelačnej databáze, je operácia výberu údajov (*select*). Mnoho autorov vo svojich prácach ukázalo, že čas potrebný na získanie údajov v nerelačnej databáze je podstatne horší ako čas potrebný na získanie údajov z relačnej databázy. Z dôvodu zrýchlenia resp. zlepšenia času potrebného na získanie údajov z nerelačnej databázy je možné dáta uložiť do medzipamäte, a tým znížiť opakované hľadanie z nerelačnej databázy. Touto metódou sa skracuje nielen čas, ale aj niekoľko prístupov do databázy. Autori vykonali rôzne porovnania medzi databázami v pamäti, ako sú Redis, Memcached, a nerelačné databázy MongoDB, Casandra

a H2. Jedným z hlavných zistení týchto prác je overenie aktualizácie a mazania údajov so zvyšujúcim sa počtom údajov.

Počas nášho výskumu sme preskúmali prácu, ktorá sa zameriava na riešenie problémov s rastúcim objemom dát. V práci [49] autori vytvorili modul pomocou knižnice *Lontar*, ktorá odošle údaje do relačnej databázy Hibernate ako framework a relačný mapovač, v prípade požiadavky používateľa. Následne režim dlhodobého spánku pristupuje k MySql a mapuje relačné údaje na objektovo orientované a potom ich odosiela do nerelačnej databázy. Vyhľadávanie potom funguje pomocou mapovača, takže *Lontar* mohol čítať údaje v relačnom vzťahu. Podľa autorov dosiahli niektoré dátové súbory lepšie výsledky v nerelačnej databáze MongoDB s operačným vyhľadávaním ako v relačnej databáze MySql. Avšak v určitých situáciách, ako autori ďalej popisujú, mala relačná databáza lepšie výsledky ako nerelačná databáza.

Autori predstavili rámec schopný manipulovať s dátami s cieľom prekonať problémy spojené so znižujúcou sa efektívnosťou vyhľadávania v nerelačných databázach oproti vyhľadávaniu v relačných databázach, ešte pred vykonaním základných operácií výberu dát, vkladania dát, aktualizácie dát a odstránenie údajov. Hlavnou úlohou mapovača je zmena údajov na základe pravidiel do takej podoby, ktorá viac vyhovuje zásadám nerelačnej databázy MongoDB, čo sa týka operácie vyhľadávania. Týmto modulom sa vo väčšine prípadov stáva situácia, keď sú dáta rýchlejšie vyhľadávané pomocou nerelačnej databázy MongoDB a potom relačnej databázy MySql. Ďalším konceptom použitým v tomto rámci je modul *Cataloging*, ktorý využíva JSP (Java Server Pages) (JAVA) ako technológiu webového programovania a MySql ako DMBS [91]. Existujú dva rámce, ktoré ho podporujú, *Struts* a *Hibernate* [91]. Štruktúry sa používajú na nastavenie užívateľského rozhrania a režim dlhodobého spánku sa používa na mapovanie relačných údajov na objektovo orientované údaje, ktoré použije JSP. V našej práci navrhujeme rámec, ktorý taktiež využíva dva typy databáz. Prvou databázou je nerelačná databáza DynamoDB slúžiaca ako primárne úložisko údajov. V prípade, že užívateľ požaduje vyžiadané údaje, hodnoty mu nebudú dané priamo z nerelačnej databázy, ale hodnoty sa prenesú do databázy v pamäti. Hlavnou výzvou na dosiahnutie tohto cieľa je prenos údajov z nerelačného modelu do modelu v pamäti.

Preskúmaná problematika zrýchľovania dát z nerelačnej databázy nám umožnila získať nielen poznatky o skúmanej problematike, ale určila nám aj vstupný bod, od ktorého môžeme tento proces zrýchliť. Nakoľko proces správy čítania dát bol už aplikovaný nemuseli sme proces opätovne zavádzať a stačilo nám metódu ďalej dopĺňať o správu čítania a následne správu čítania-zápisu. Nami doplnený princíp nám redukoval počet čítaní z primárnej databázy, počet zápisov údajov a zrýchlil celý proces.

6.4.2 Dôvod skúmania danej problematiky vyhľadávania v nerelačných databázach

Pri skúmaní danej problematiky sme sa zoznámili s možnosťou správy často dopytovaných dát a ich presunu do dát v pamäti akými sú databázy Redis alebo Memcached. Tieto databázy poskytujú účinný spôsob ako zefektívniť proces vyhľadávania v pamäti. Za nedostatok však môže byť považované ich pamäťové ohraničenie.

Ako nedostatok, ktorý sme počas výskumu zaznamenali je neefektívna manipulácia s dátami, ktoré boli spracované pomocou rôznych metód a ktoré boli publikované v článku [49] a následne presunuté z primárnej databázy do pamäte.

Na základe spomenutého nedostatku manažovania dát, kedy sú dáta presúvané z databázy do vyrovnávacej pamäte (*cache*), sme vytvorili spôsob, akým manažujeme nie len hodnoty čítania dát z databázy, ale manipulujeme resp. presúvame hodnoty do medzipamäte už pri zápise hodnôt, a tým redukujeme čas potrebný na opätovné načítanie záznamu hneď po jeho zápise.

6.4.3 Naše riešenie súvisiace s vyhľadávaním v nerelačných databázach

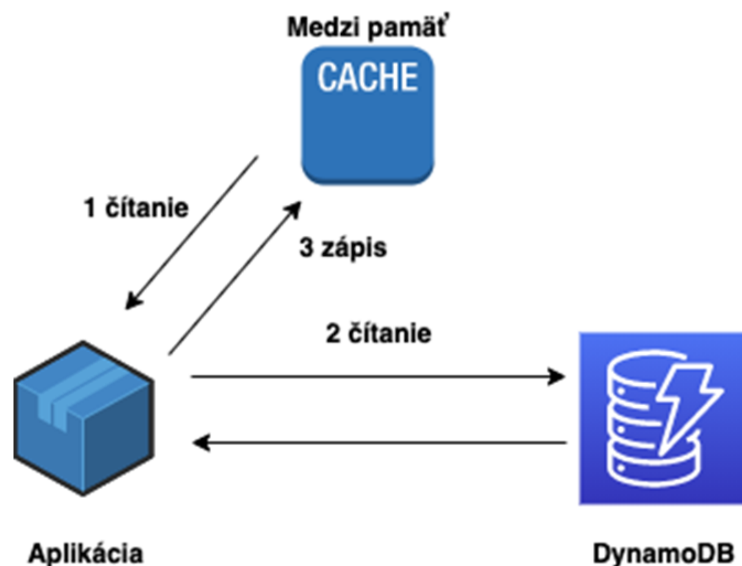
V tejto časti predstavíme dva moduly, ktoré nám pomáhajú zefektívniť vyhľadávanie v nerelačných databázach. Modul s medzi pamäťou s údajmi a modul s elastickými údajmi. DCM (*Data Cached Module*) slúži ako sklad dátových úložísk, ktorého úlohou je prenos dát do medzipamäte. DEM (*Data Elastic Module*) slúži na automatické prispôsobenie veľkosti dát.

6.4.3.1 Moduly s medzipamäťou

Vytvorený modul slúži na spracovanie údajov v pamäti. Údaje, ktoré ukladáme do nerelačnej databázy DynamoDB, sme pomocou rozhrania API prepojili s vysoko dostupnou vyrovnávacou pamäťou *Amazon DynamoDB Accelerator*, veľmi skrátene DAX.

Táto metóda nám pomáha pri 3 prístupoch: bočná vyrovnávacia pamäť (obr. 46), medzipamäť na čítanie (obr. 47) a medzipamäť na zápis (obr. 48).

a) *Bočná vyrovnávacia pamäť* (*Side-cache)* - tento princíp nám pomáha pri vysokom preťažení počas čítania informácií z pamäte. Tento princíp funguje nasledovne:
   1. Aplikácia sa najskôr pokúsi načítať údaje z medzipamäte pre daný pár *kľúč - hodnota*. Ak bola vyrovnávacia pamäť naplnená údajmi (prístup do vyrovnávacej pamäte), hodnota sa vráti. Ak nie, nasleduje krok 2.
   2. Aplikácia načíta údaje z úložiska základných údajov(*primárneho úložiska*), pretože sa nenašiel požadovaný pár *kľúč - hodnota*.
   3. Do vyrovnávacej pamäte sa zapíše pár *kľúč - hodnota* z kroku 2, aby sa zabezpečilo, že sú prítomné údaje, keď je potrebné aplikáciu znova načítať.



Obrázok 46. Algoritmus bočnej pamäte (Side-cache algorithm)

b) *Medzipamäť na čítanie* – DAX (*Amazon DynamoDB Accelerator*) je vyrovnávacia pamäť na čítanie, pretože je kompatibilná s API na čítanie API DynamoDB a ukladá výsledky *GetItem*, *BatchGetlem*, *Scan* a *Query* do vyrovnávacej pamäte, ak nie sú práve v DAX. Vyrovnávacia pamäť pre čítanie je účinná pri náročných pracovných zaťaženiach. Tento princíp funguje nasledovne:
   1. Pokiaľ ide o pár *kľúč - hodnota* z aplikácie, najskôr sa pokúsi načítať údaje pomocou DAX. Ak bola vyrovnávacia pamäť naplnená údajmi (prístup do vyrovnávacej pamäte), hodnota sa vráti. Ak nie, nasleduje krok 2.

2. Transparentne pre aplikáciu, ak sa stane polovičná pamäť, DAX načíta pár párov *kľúč – hodnota* z DynamoDB.
3. Aby boli údaje k dispozícii pre každé nasledujúce čítanie, potom sa pár *kľúč - hodnota* zaplní v semi-pamäti DAX.
4. Pár *kľúč – hodnota* potom vráti hodnotu aplikácii.

Obrázok 47. Algoritmus medzipamäte načítania

c) *Medzipamäť na zápis* - Podobne ako semi-pamäť na čítanie, aj semi-pamäť na zápis dát pracuje v súlade s databázou a aktualizuje semi-pamäť, keď sa dáta zapisujú do pamäte základných údajov. Jazyk DAX tiež ukladal do vyrovnávacej pamäte pre zápis, pretože do medzipamäte (alebo do nej aktualizuje) ukladá položky pomocou API *PutItem*, *UpdateItem*, *DeleteItem* a *BatchWriteItem*, pretože údaje sa zapisujú alebo aktualizujú v DynamoDB. Najskôr sa aktualizuje jazyk DAX (pre aplikáciu je všetko transparentné). Nasledujúce kroky označujú postup pre zápis typu vyrovnávacej pamäte.

1. Aplikácia sa sama zapíše do koncového bodu DAX pre daný pár párov kľúč - hodnota.
2. DAX zachytí zápis a potom zapíše pár kľúč - hodnota do DynamoDB.
3. Po úspešnom zápise DAX aktualizuje vyrovnávaciu pamäť DAX s novou hodnotou, takže podľa toho, či po prečítaní toho istého páru *kľúč - hodnota* bude mať za následok nájdenie hodnoty vo vyrovnávacej pamäti. Ak zápis nebude úspešný, vráti sa do aplikácie výnimka.
4. Potvrdenie o úspešnom zapísaní sa potom vráti do aplikácie.

Obrázok 48. Medzipamäť na zápis

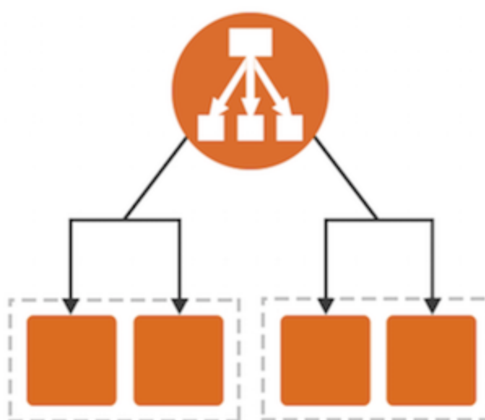
6.4.3.2 *Dátový elastický modul*

Vytvorený modul slúži na zväčšovanie údajov v medzipamäti. Nerelačná databáza DynamoDB plní úlohu širokého úložiska dát a v prípade prenosu dát do databázy v pamäti sa dáta môžu pohybovať od niekoľkých megabajtov do gigabajtov. Tento spomínaný problém momentálne rieši vytvorený modul.

Monitorovanie metrík sme nakonfigurovali v cloud-ovej službe Amazon pomocou služby Amazon *CloudWatch*. Uvedená služba umožňuje editovať, pridávať a odstraňovať nové výpočtové jednotky v prípade horizontálneho alebo vertikálneho upravovania. Výhodnou charakteristikou tejto metódy je horizontálne škálovanie, ktoré v prípade veľkého počtu dátových prenosov vyvolá varovanie pred veľkým preťažením a skript na načítanie informácií z iných replík v službe *CloudWatch*. Horizontálna replika je časť skriptu vykonávaná automaticky počas konfigurácie databázy v pamäti DAX nasledujúcim skriptom.

```
aws dax decrease-replication-factor \
    --cluster-name MyNewCluster  \
    --new-replication-factor 3
```

Monitorovanie metrík rovnakým spôsobom s našou metódou umožňuje aj vertikálne škálovanie (zobrazené na obr. 49), čo je škálovanie pridaním alebo odstránením výpočtových jednotiek.



Obrázok 49. Application Load Balancer pre databázu v pamäti

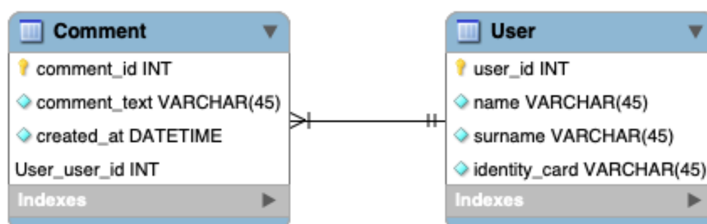
V prípade nastátia situácie, keď sa hodnoty získané z nerelačnej databázy nezmestia do databázy v pamäti, tak sa udalosť vyvolá znova pomocou služby CloudWatch, ktorá spôsobí pridanie novej výpočtovej jednotky.

Ak výpočtová jednotka už nie je potrebná, automaticky sa uvoľní inštancia, čo má za následok úsporu medzipamäte a optimalizáciu ceny.

6.4.4 Experimenty potvrdzujúce efektívnosť nami navrhnutej metódy

Hneď v prvom kroku sme vytvorili jednoduchý databázový model na obr. 50. Tento databázový model je vytvorený z dvoch tabuliek, používateľ (*user*) a komentár (*comment*). Tieto dve tabuľky sú vzájomne prepojené identifikačným vzťahom typu 1:n, čo znamená, že 1 užívateľ môže vytvárať rôzne komentáre a rôzne komentáre v tabuľke patria priamo 1 užívateľovi. Následne sme porovnali rôzne príkazy, ktorých cieľom je získať informácie o stúpajúcom trende vyhľadávania s rastúcim počtom údajov v relačnej databáze Oracle v sekcii A, a potom aj v nerelačnej databáze DynamoDB v sekcii B. Porovnávali sme tiež časy rôznych operácií počas výberu údajov v časti C, pretože dôležitým aspektom tohto experimentu je použitie databázy v pamäti.

6.4.4.1 Experimenty s relačnou databázou Oracle



Obrázok 50. Dátový model

Na základe tejto definovanej štruktúry sme do tabuľky používateľov vložili 1 000 záznamov a 1 000 záznamov aj do komentárov k tabuľke. V prípade tohto experimetnu nás zaujímajú iba informácie o čase potrebnom na vyhľadanie záznamu. Pre tieto účely sme vytvorili 3 príkazy výberu údajov, ktoré vyzerajú nasledovne:

```
(1) SELECT name, surname FROM user
    JOIN comment USING (user_id);
```

```
(2) SELECT * FROM user
    JOIN comment USING (user_id)
    WHERE comment_text LIKE “%today%”;
```

```
(3) SELECT  name, surname,  to_char(created_at, 'YYYY-MM-DD HH24:MI:SS') ca FROM user
    JOIN comment USING (user_id)
    WHERE ca >= TRUNC(current_date)
    and ca < TRUNC (current_date) + 1;
```

Prvých 1 000 záznamov nám slúžilo ako úplne prvé záznamy, od ktorých sa môže náš výskum odraziť.

Hlavným účelom nerelačných databáz je efektívne ukladanie veľkého množstva údajov, a preto sa budú vytvárať záznamy na iné účely s veľkosťou 100 000 záznamov pre používateľov tabuľky a 100 000 záznamov pre komentár k tabuľke. Následne sú všetky záznamy vymazané a záznamy o veľkosti 10 000 000 budú vložené užívateľovi tabuľky a komentáru. Ako poslednú veľkosť záznamov sme vybrali hodnotu 100 000 000 záznamov.

Na vytvorenie záznamu bol použitý generátor s veľkosťou 1 000, 100 000, 10 000 000 a 100 000 000, ktorý je možné nájsť na tejto adrese: *https://www.generatedata.com/*. Generátor poskytuje možnosť definovať mená a typy atribútov a vygenerovať ľubovoľný počet hodnôt. Po naplnení tabuliek vygenerovanými hodnotami zaznamenávame časy potrebné na vykonanie operácií (1), (2) a (3), ktoré sú zobrazené v tabuľke 20.

Tabuľka 20. Nameraný čas pre operácie (1), (2) a (3)

Počet operácií/operácia	1 000	100 000	10 000 000	100 000 000
(1)	0,002	0,004	0,028	0,44
(2)	0,0021	0,0042	0,031	0,45
(3)	0,0021	0,0044	0,03	0,45

Hodnoty potrebné na získanie údajov z relačnej databázy Oracle sú uvedené v tabuľke 20. Všetky dosiahnuté hodnoty pre príkazy (1), (2) a (3) sú odsledované v sekundách.

6.4.4.2 Experimenty pre nerelačnú databázu DynamoDB

Pomocou príkazov (1), (2) a (3) sme zistili rýchlosť dopytu v nerelačnej databáze DynamoDB. Hodnoty vložené do databázy boli ponechané rovnaké ako pri experimentoch v relačnej databáze. Štruktúra je úplne rovnaká, ako je zaznamenaná v tabuľke 21.

Tabuľka 21. Nameraný čas pre operácie (1) (2) (3) v databáze DynamoDB

Počet operácií/operácia	1000	100000	10000000	100000000
(1)	0,0035	0,0064	0,047	0,82
(2)	0,0035	0,0067	0,048	0,83
(3)	0,0037	0,0068	0,046	0,82

Hodnoty potrebné na získanie údajov z nerelačnej databázy DynamoDB sú zaznamenané v tabuľke 21. Všetky dosiahnuté hodnoty pre objednávky (1), (2) a (3) sú merané v sekundách.

Počas porovnania hodnoty tých istých operácií sú hodnoty medzi výsledkami relačných a nepríbuzných databáz výrazne odlišné. Nerelačná databáza v operácii výberu údajov je časovo menej efektívna ako relačná databáza Oracle.

6.4.4.3 Experimenty s databázou v pamäti Redis

Uloženie do databázy v pamäti je diametrálne odlišné ako v relačných alebo nerelačných databázach. Veľkým problémom, už na spomenutú skutočnosť, je aj obmedzenie množstva dát pamäťou počítača. Na účely testovania bola použitá pamäť o veľkosti 8 GB.

Tabuľka 22. Štruktúra dát

ID	Meno	Priezvisko	Identity_card
1	John	Harper	12341324
2	Joe	Bush	12341234
3	George	Obama	23524675
.....	.....	.....	.....
.....	.....	....	.....
1000	Alan	Felps	45674866

Ako je zrejmé z tabuľky 22, vytvorili sme 100, 300, 500 a 10 000 záznamov so štruktúrou ID, menom, priezviskom a identifikačnou kartou. Stránka na tejto adrese https://www.generatedata.com/ bola použitá na tento účel vytvorenia záznamov. Boli vytvorené 4 príkazy na účely testovania databázy z hľadiska efektívnosti pamäte, kde sme sledovali rýchlosť získavania údajov. Sú to nasledujúce prípady:

(4) MGET Name

(5) MGET Name, Surname

(6) MGET Name, Surname, Identity_card

(7) MGET Name, Surname, Identity_card, Age

Pri vypĺňaní databázy sme uplatňovali rovnaký princíp ako v predchádzajúcich krokoch. Vygenerované hodnoty sme vložili do databázy, otestovali operácie (4), (5), (6) a (7) a zaznamenali hodnoty.

Následne sme vymazali všetky záznamy a vložili údaje o veľkosti 300 záznamov do databázy a takto sme pokračovali až do veľkosti 1 000 záznamov v databáze. Hodnoty pre operácie (4),(5),(6) a (7) sú zaznamenané v tabuľke 23.

Tabuľka 23. Nameraný čas pre databázu Redis

Počet operácií/operácia	100	300	500	1 000
(4)	0,0002	0,0002	0,00021	0,00028
(5)	0,0002	0,0002	0,00025	0,00029
(6)	0,0002	0,0002	0,00025	0,00028
(7)	0,0002	0,0002	0,00028	0,00032

Všetky dosiahnuté výsledky, ktoré sme namerali, sú zaznamenané v tabuľke 23 a sú uvedené v sekundách. Siedma operácia je ovplyvnená skutočnosťou, že hodnota „vek" neexistuje. Ako je zrejmé, namerané hodnoty sa diametrálne nelíšia od hodnôt pri zväčšovaní počtu údajov. Je potrebné poukázať na to, že pri definovanom raste záznamov je to logická skutočnosť odzrkadľujúca efektivitu vyhľadávania v pamäti.

6.4.4.4 Porovnanie konečných výsledkov

Výsledné hodnoty z našej experimentálnej činnosti uvedené v tab 23, slúžia práve na porovnanie s nami navrhnutou metódou. Hodnoty potrebné na získanie údajov s operáciami (1) boli namerané a zaznamenané do tabuľky 24.

Tabuľka 24. Porovnanie výkonu dotazu

Počet operácií/prístup	1 000	1 000 000
Oracle	0,002	0,44
DynamoDB	0,0035	0,82
Náš prístup	0,0033	0,42

Ako je zrejmé z tabuľky 24, namerané a získané hodnoty pri použití operácie (1) neukazujú nijaké veľké zlepšenie vyhľadávania v nerelačnej tabuľke pre nás, s nízkym počtom záznamov v tabuľke. Tento jav je ovplyvnený prenosom dát do pamäte[8]. Faktor prenosu naznačuje nutnosť prenosu dát z nerelačnej databázy DynamoDB do medzipamäte DynamoDAX, čo trvá určitý čas a až vtedy užívateľ získa dopytované záznamy.

[8] Vylepšenú metódu výberu údajov v nerelačnej databáze sme prezentovali a diskutovali na IEEE konferencii Information and Digital Technologies na Slovensku (Žilina) - 22. - 24. jún 2021.

To znamená, že pri výbere údajov sa údaje fyzicky nezískajú z nerelačnej databázy DynamoDB, ale z databázy v pamäti.

Na základe prenosu dát bolo možné porovnať aj hodnoty medzi experimentmi s databázou v pamäti a dátami prenesenými do DynamoDAX s veľkosťou 100 a 500 záznamov počas operácie (5).

Tabuľka 25. Výkon dotazu v pamäti

Počet operácií/databáza	**100**	**500**
Redis	0,0002	0,00021
DynamoDAX	0,00015	0,00017

Hodnoty zaznamenané v tabuľke 25 nám ukazujú jasný spôsob a efektívnosť dátového prenosu. Je vidieť, že hodnoty vo vyrovnávacej pamäti Dynamo DAX sú z časového hľadiska efektívnejšie oproti databáze Redis v pamäti.

Celkovo dosiahnuté výsledky spojené s výberom údajov v nerelačnej databáze DynamoDB neboli pred aplikáciou našej metódy včasne rovnaké ako po aplikácii našej metódy. Vďaka použitiu strojového učenia a prenosu údajov do databázy v pamäti sa efektívnosť výberu prevádzkových údajov v nerelačnej databáze stala efektívnejšou po dosiahnutí 1 000 000 záznamov ako pri vyhľadávaní údajov v relačnej databáze Oracle. Obrovská výhoda, ktorá má za následok využitie cloud-ového úložiska Amazon, súvisí aj s možnosťou automatického škálovania, respektíve zvyšovania výkonu a zväčšovania úložného priestoru nielen v nerelačnej databáze DynamoDB, ale hlavne v alternatívnej databáze v pamäti. Ak tak urobíme nie je potrebné toľko výpočtových jednotiek, takže dôjde k zmenšeniu veľkosti dátového úložiska a k zníženiu nákladov spojených s prevádzkou nami navrhnutej metódy.

Ako je vidieť z tabuľky 26, tak sledované hodnoty, ktoré sú pre nás podstatné a znamenajú pre nás pokrok oproti konvenčnej metóde sú zvýraznené zelenou farbou a negatívne aspekty sú zvýraznené červenou farbou. Riadky, ktoré sú zobrazené bielou farbou nespôsobujú veľké ovplyvnenie celkového výsledku.

Zavedenie našej metódy do procesu spôsobuje väčšie režijne náklady, ale tieto namerané hodnoty zobrazené v tabuľke 26, konkrétne v riadkoch 9, 10 a 11 sa so zvyšujúcim množstvom nemenia.

Po aplikovaní nami navrhnutej metódy a zväčšenia počtu záznamov sa efektívnosť metódy prejavila až pri počte záznamov 875 234, čo sme zaokrúhlili na hodnotu 1 000 000 záznamov. Nakoľko pri aplikovaní metódy dochádza k presunu záznamov medzi jednotlivými databázami bolo potrebné vykonávať rozsiahlejšie riadenie toku dát, ktoré je zabezpečené nami navrhnutou metódou, ktoré spôsobilo zväčšenie režijných nákladov.

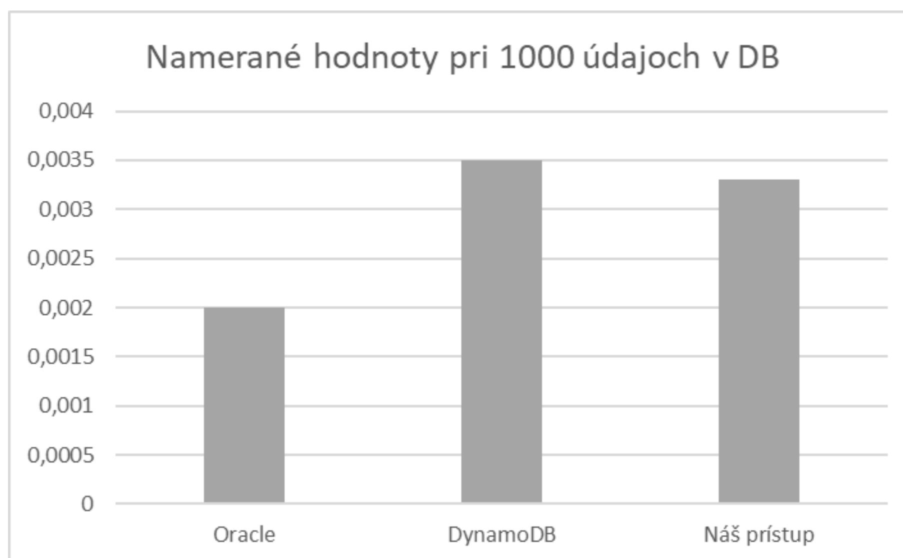
Tabuľka 26. Porovnanie sledovaných hodnôt medzi konvenčnou metódou a metódou správy záznamov

Číslo vlastnosti	Sledovaná vlastnosť	Konvenčná metóda	Metóda správy záznamov
1	Väčší počet databáz [Áno/Nie]	Nie	Áno
2	Správa dát v pamäti [Áno/Nie]	Nie	Áno
3	Správa čítanie [Áno/Nie]	Nie	Áno
4	Správa zápisu [Áno/Nie]	Nie	Áno
5	Riadenie toku [Áno/Nie]	Nie	Áno
6	Redukcia počtu čítania [Áno/Nie]	Nie	Áno
7	Redukcia počtu zápisu [Áno/Nie]	Nie	Áno
8	Správa údajov [s]	0	0,002
9	Režijne náklady [%]	23	26
10	Vyťaženie servera [%]	16	21
11	Počet vlákien [Počet]	1	3

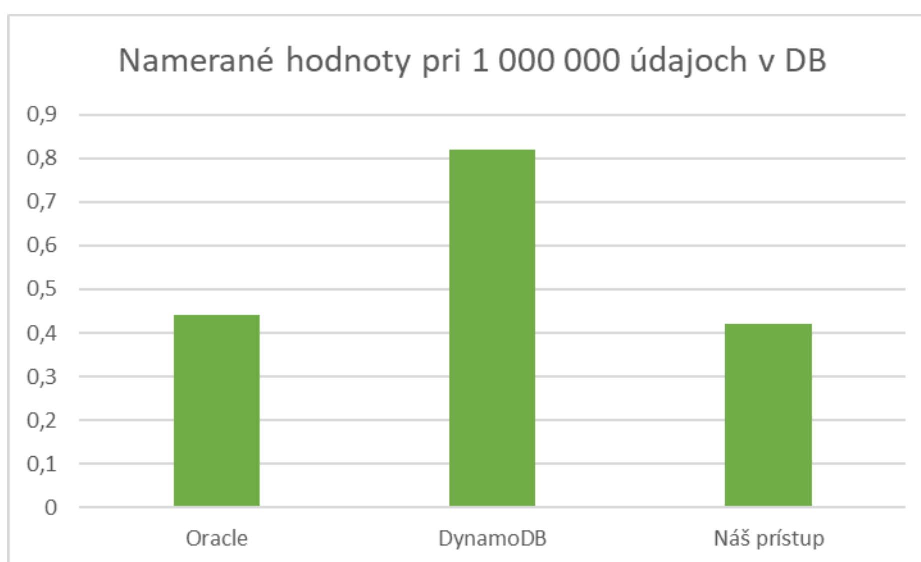
Ako je vidieť z grafu na obr. 51 a 52**,** tak sa nám výsledky rozdelili do 2 skupín. Prvou skupinou je situácia, pri ktorej sa efektivita našej metódy úplne neprejavila, ale získali sme údaje rýchlejšie ako pri využití nerelačnej databázy DynamoDB. Tieto hodnoty je možné vidieť na obr. 51 Na obrázku je vidieť, že relačná databáza je pri počte 1 000 záznamov najviac efektívna.

Pri porovnaní výsledkov s hodnotami na obr. 52 sa už prejavila efektivita našej metódy. V tejto situácii bolo už v databáze 1 000 000 záznamov. Na začiatku je potrebný istý čas pre metódu, ktorá presúva záznamy medzi primárnou databázou a databázou v pamäti a taktiež naučiť sa, ktoré záznamy používateľ používa a ktoré nie.

Pri spracovaní väčšieho počtu údajov, konkrétne išlo o hodnotu 875 234 sa efektívnosť vyhľadávania záznamov medzi relačnou databázou Oracle a nerelačnou databázou DynamoDB dostala na rovnakú časovú úroveň. Od spomenutého počtu záznamov sa efektívnosť našej vytvorenej metódy postupne zlepšovala.



Obrázok 51. Grafické porovanie výberu hodnôt pri 1 000 údajoch z vybratých DB



Obrázok 52. Grafické porovnanie výberu hodnôt pri 1 000 000 údajoch v DB

6.4.5 Stručné zhrnutie výsledkov

Databázy NoSql zohrávajú významnú úlohu pri ukladaní a spracovávaní obrovských dát a používajú sa v rôznych širších sociálnych aplikáciách, ako sú napríklad Twitter, Facebook, Google a Yahoo, ale pomáhajú aj pri podpore rozhodovania alebo pri tvorbe pokročilých analýz. Stali sa pánom vysokej efektívnosti a dostupnosti obrovských dát, ale so stratou efektívneho vyhľadávania oproti tradičným databázam.

Aktuálna časť práce bola venovaná otázke vyhľadávania v databáze NoSql, konkrétne DynamoDB v cloudovom prostredí Amazonu, aby sa minimalizoval dopad tohto problému.

V tomto príspevku sme vyvinuli vyhľadávací algoritmus, ktorý umožňuje zefektívniť rýchlosť vyhľadávania v nerelačnej databáze DynamoDB. Navrhnutý algoritmus sa skladá z dvoch častí, prvá časť je založená na princípe ukladania údajov z nerelačnej databázy DynamoDB do medzipamäte DynamoDAX. Druhá časť je založená na efektívnej správe dát, ktorá je schopná v prípade veľkého množstva operácií automaticky vytvárať a zväčšovať výpočtové jednotky medzipamäte, a tým prispôsobiť veľkosť databázy rastúcim potrebám veľkosti prichádzajúcich údajov. Táto skutočnosť nás zbavila obmedzenia veľkosti databázy smerom k údajom.

Pokusy priniesli niekoľko užitočných informácií o výkonnosti a efektívnosti vytvorenej metódy. Je potrebné poznamenať, že systém na spracovanie umelej inteligencie vyžadoval vyššie režijné náklady spolu s automatickým vytváraním databázy v pamäti, ale tento systém dokázal zefektívniť proces vyhľadávania v nerelačnej databáze. Na základe experimentov je jasne vidieť, že vytvorená metóda je čoraz efektívnejšia s rastúcim objemom dát, ktorý sa deje prenosom dát do pamäte.

Na základe výsledkov, ktoré sme získali pri experimentálnej činnosti vidíme jasne preukázateľné zlepšenie po zavedení našej metódy. Pri experimentoch so serverom vidíme zhoršujúci efekt, ktorý sa týka zvyšovania režijných nákladov.

Pri prvotnom skúmaní sme chceli nad dátami v nerelačnej databáze zaviesť sekundárny index, čo by nám pomohlo pri zrýchlení vyhľadávania v nerelačných databázach, a tým aj rýchlejšie presunúť záznamy do pamäte. Výhodou nerelačných databáz, ale v tomto prípade je to pre nás nevýhoda, je voľnosť štruktúry. Práve tento fakt nám neumožňuje vytvorenie sekundárneho indexu na základe dátových typov, ktoré sú pre nás potrebné.

Na základe situácie sme usúdili, že by stálo za zváženie vytvorenie prístupu, ktorý nám určitým spôsobom udrží pevnú dátovú štruktúru a ktorý nám podporí vytvorenie sekundárneho indexu nad často žiadanými dátami.

6.5 Metóda na zrýchlenie vyhľadávania dát v nerelačnej databáze MongoDB

Na základe záverov z predchádzajúcej kapitoly a odhalených nedostatkov sme navrhli riešenie spojené s udržiavaním pevnej štruktúry dát, a taktiež možnosť vytvorenia sekundárneho indexu nad dátami.

Podľa nášho názoru a testov je efektívnejšie vytvorenie pevnej štruktúry v relačnej databáze ako neustála kontrola vložených dát pomocou triggra v nerelačnej databáze a následná kontrola údajov pri každej operácii vloženia.

Na základe spomenutých nedostatkov a nutnosti udržiavania stálej štruktúry sme si stanovili ciele a využili prostriedky, ktorých cieľom je:

- vytvorenie pevnej štruktúry a definovanie typov údajov,
- vytvorenie procesu vzájomnej komunikácie medzi relačnou a nerelačnou databázou,
- možnosť vytvorenia sekundárneho indexu.

V nasledujúcej podkapitole preskúmame už existujúce riešenia výskumníkov, ktorí sa vo svojich výskumoch venovali spomínanej problematike.

6.5.1 Práce zaoberajúce sa vyhľadávaním dát v nerelačných databázach pomocou relačných databáz

Relačné databázy poskytujú dobrú podporu pre správu štruktúrovaných údajov. Nedávny vývoj v oblasti IT však prináša veľké objemy dát, ktoré sa vyznačujú extrémne veľkým objemom a rozmanitosťou dátových typov a štruktúr. Pre relačné databázy je ťažké spracovať tak veľké objemy dát z dôvodu prísnych obmedzení týkajúcich sa dátovej štruktúry a vzťahov s údajmi atď. Na druhej strane, NoSql databázy, vrátane HBase, MongoDB, Cassandra atď., si získavajú popularitu pre svoju schopnosť zaobchádzať s veľkým množstvom komplexných údajov v rôznych štruktúrach. V súčasnosti neexistuje spôsob, ako poznať uskutočniteľnosť migrácie tradičných relačných databáz do databáz NoSql. Migrácia vyžaduje vyhodnotenie uskutočniteľnosti migrácie a potenciálneho výkonu nových systémov. Príspevok [122] skúma tieto problémy modelovaním jednej z databáz NoSql, MongoDB, pomocou relačnej algebry.

Modelovanie umožní porovnanie sémantických expresných schopností medzi relačnými databázami a MongoDB, analyzuje uskutočniteľnosť migrácie relačnej databázy na MongoDB a jej vyhodnotenie. Očakáva sa, že model uľahčí optimalizáciu novej databázy NoSql.

Výskumy, zaoberajúce sa zrýchlením získavania dát v nerelačnej databáze prišli na niekoľko zaujímavých myšlienok. Bolo aplikovaných množstvo algoritmov, ktorých hlavným účelom je poskytnúť používateľom dopytujúcim sa na dáta rýchlejšie odozvu pri získaní dát z dátového úložiska. K mnohých zaujímavým štúdiám zaraďujeme článok od výskumníka Kveta [66], v ktorej autor predstavuje rozšírenie konceptu štandardných databáz, ktoré spracovávajú iba aktuálne údaje. Príspevok sa zaoberá časovou štruktúrou na úrovni objektu v porovnaní s časovými údajmi na úrovni stĺpcov. Opisuje zásady, požadované metódy, postupy, funkcie a spúšťače zabezpečujúce funkčnosť tohto systému. Definuje tiež možné implementácie a ponúka riešenie na získanie snímky databázy alebo objektu kedykoľvek počas existencie. Dôvod vývoja riešenia na úrovni stĺpcov je založený na heterogenite atribútu čas. Niektoré atribúty však nemenia svoje hodnoty v priebehu času alebo sa aktualizujú veľmi zriedka, a preto nie je potrebné pre tieto atribúty zaznamenávať nové hodnoty.

Jedným zo spôsobov akým je možné zrýchliť získavanie dát je obmedziť dáta pomocou časovej hodnoty. Táto hodnota jednoznačne oddelí dáta, ktoré sú pre nás dôležité od tých, s ktorými sa v systéme menej pracuje. Tento typ bol publikovaný v článku [64], v ktorom autori navrhli novú komplexnú tabuľkovú klasifikáciu v prostredí časovo obmedzenej platnosti s dôrazom na efektivitu liečby, zotavenia a vyhľadávania informácií. Tieto údaje významne ovplyvňujú rozhodovací proces. Zameriavajú sa na rôzne typy definícií tabuliek, ich charakteristiky a vhodnosť na použitie. Následne predstavený prístup, ktorého jadro tvorí časovú zrnitosť na úrovni stĺpca, ktorá bola definovaná v nedávnej minulosti. Hlavnou časťou príspevku [64] je definovať hybridný systém pre konečné porovnanie pre rôzne frekvencie a rýchlosti aktualizácie dát podporované porovnaním výkonu.

V publikovanom článku [6] bola predstavená štúdia, v ktorej bol navrhnutý nový štvorkrokový hybridný prístup k vyhľadávaniu a zostavovaniu video spravodajských relácií na základe informácií obsiahnutých v rôznych množinách metadát.

V prvom kroku autori používajú konvenčné techniky vyhľadávania na izoláciu segmentov videa z dátového vesmíru pomocou metadát segmentu. V druhom kroku sú načítané segmenty zoskupené do potenciálnych noviniek pomocou dynamickej techniky citlivej na informácie obsiahnuté v segmentoch. V treťom kroku navrhnutá metóda používa techniku tranzitívneho vyhľadávania, aby sa zvýšilo vyvolanie vyhľadávacieho systému. V poslednom kroku sa zvyšuje výkonnosť zapamätania identifikáciou segmentov, ktoré majú vzťahy medzi tvorbou a časom. Kvantitatívna analýza výkonnosti procesu pri zostavovaní spravodajského vysielania ukazuje zvýšenie zapamätania o 59 percent v porovnaní s konvenčnou technikou vyhľadávania pomocou kľúčových slov použitou v prvom kroku.

V niektorých štúdiách, kedy dochádza k mapovaniu objektov z nerelačnej databázy na relačnú, je pri vyhľadávaní potrebné riešiť problémy s neúplnými údajmi. Mnohokrát existuje prípad, počas ktorého je objekt definovaný iba čiastočne alebo n-tica údajov nie je definovaná vôbec. Z týchto dôvodov musia byť nedefinované hodnoty uložené v databáze vo forme samotného času alebo atribútu vyjadrujúceho stav objektu. Autori sa vo svojom príspevku [64] zaoberajú časovo orientovanými databázovými architektúrami, spravujú nedefinované hodnoty a navrhujú komplexnú systémovú klasifikáciu založenú na transakciách, prístupoch a indexoch. Zaoberajú sa technikami modelovania nedefinovaných hodnôt a pokrývajú synchronizačné procesy pomocou dátových skupín. Autori v príspevku navrhli riešenia pre efektívne získavanie údajov s dôrazom na nedefinované hodnoty a stavy.

V mnohých smeroch sú v nerelačných databázach ukladané dlhé texty, v ktorých je často vyhľadávanie neefektívne. Autori predstavili štúdiu [29], v ktorej skúmajú kombináciu textových a vizuálnych informácií v databáze lekárskych záznamov s cieľom zlepšiť výkon multimodálneho systému na získavanie informácií. Navrhovaný model sa skladá z dvoch subsystémov: subsystému vyhľadávania informácií založeného na obsahu, ktorý vykonáva vyhľadávanie obrázkov, a subsystému vyhľadávania textových informácií, ktorý vykonáva textové vyhľadávanie. Obrázky a text sa načítajú nezávisle a potom sa spájajú výsledné zoznamy. Bola uskutočnená a analyzovaná štúdia rôznych váhových schém. Získané výsledky ukazujú, že správna integrácia textových informácií zlepšuje konvenčné multimodálne systémy.

Súčasnosť predstavuje veľkú mieru škálovatelnosti a paralelizmu. Tento smer aplikovali vo svojej práci výskumníci [73], ktorí predstavili princípy a implementačné mechanizmy automatického delenia (*auto-sharding)* v databáze MongoDB v ktorom taktiež navrhli vylepšený algoritmus založený na frekvencii dátových operácií, aby sa vyriešil problém nerovnomerného rozdelenia dát v automatickom delení (*auto-sharding)*. Vylepšená stratégia vyvažovania môže efektívne vyvážiť údaje medzi zlomkami a zlepšiť súbežný výkon čítania a zápisu klastra.

Viaceré články, ktoré sme pri našom výskume zaznamenali predstavili zaujímavé riešenia problémov súvisiacich so zefektívnením vyhľadávania v nerelačných databázach. Mnohé riešenia zobrali ako východiskový bod nerelačnú databázu MongoDB, ktorá je považovaná za najuniverzálnejšiu nerelačnú databázu. Sami autori, vo svojich článkoch akceptovali možnosť a ponúkli viaceré návrhy ako zefektívniť spôsob, ktorý dokáže ešte viac zrýchliť výber dát z nerelačnej databázy.

Spomenuté štúdie riešili problematiku vyhľadávania veľkým množstvom spôsobov. Nakoľko si myslíme, že udržanie istého typu štruktúry je riešenie, ktoré by nám prinieslo výhodu pri vyhľadávaní údajov a nezaznamenali sme štúdiu, ktorá by riešila proces zefektívňovania vyhľadávania údajov týmto spôsobom, myslíme si, že by vytvorená metóda mohla priniesť užitočné výsledky. Zavedenie nám prinesie typovosť, rýchlejšie vyhľadávanie a taktiež dokáže objekty rozoznať na základe štruktúry.

6.5.2 Dôvod skúmania problematiky zrýchľovania vyhľadávania dát v nerelačnej databáze MongoDB

Ako hlavný problém, ktorý sme pri skúmaní problematiky vyhodnotili za kľúčovú zložku je dátová štruktúra. Pri relačných databázach je tento problém vyriešený s kontrolou dátovej schémy pri vkladaní záznamov, ale tento aspekt nie je v nerelačných databázach implementovaný pre ich kľúčovú vlastnosť (flexibilnosť).

S neudržiavaním dátovej štruktúry súvisia viaceré problémy, akými sú dopyt dát (keďže nevieme, aké dáta a či nejaké dáta môžeme očakávať), kontrola dátových typov (získame pri dopyte reťazec alebo číslo), prípadne vytvorenie sekundárnych indexov (nachádzajú sa vo všetkých objektoch rovnaké dáta, sú prázdne alebo nie).

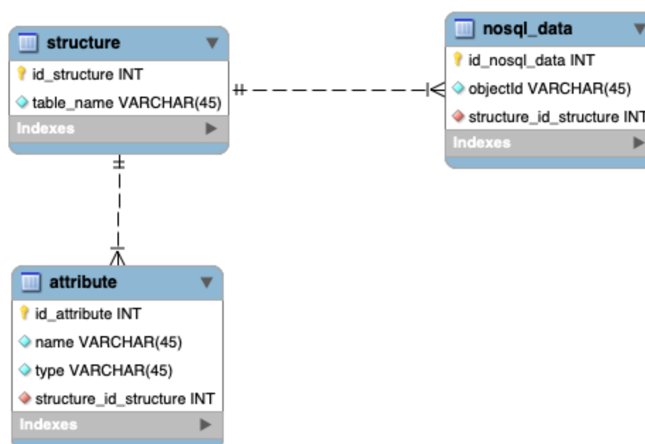
Viacero výskumníkov prinieslo efektívne spôsoby [73] ako spoľahlivo pracovať s flexibilnými a heterogénnymi dátami v nerelačných databázach ako je MongoDB alebo DynamoDB.

Vo viacerých prípadoch naozaj nie je nutné kontrolovať tieto hodnoty, čo redukuje rýchlosť vyhľadávania. Z dôvodu zrýchlenia získavania dát sme vytvorili spôsob, akým tento proces dokážeme zrýchliť, a taktiež udržať pevnú dátovú štruktúru.

6.5.3 Nami navrhnuté riešenie skúmanej problematiky

Hlavným dôvodom viacerých výskumníkov zaoberajúcich sa nerelačnými databázami je degradácia vyhľadávania v nerelačných databázach. My sme sa rozhodli tento problém riešiť odlišným spôsobom, ktorý je založený na 2 databázach. Prvým typom databázy je nerelačná databáza MongoDB, ktorá bude slúžiť ako primárne dátové úložisko a druhým typom je relačná databáza Oracle, ktorá pre nás slúži ako nástroj na držanie istej štruktúry pre nerelačnú databázu. Z dôvodu voľnej štruktúry v nerelačnej databáze sme zaviedli v relačnej databáze dátový model (ktorý je zobrazený na obr. 53), ktorý nám poskytuje udržiavanie všetkých dátových typov na 1 mieste.

Problém, ktorý sa vyskytuje pri vyhľadávaní je spojený s nekompatibilitou príkazov medzi relačnými a nerelačnými databázami[9]. Keď dôjde k situácii vkladania záznamov do nerelačnej databázy, tak pomocou skriptu sú názvy atribútov a ich dátové typu uložené do tabuľky *attribute* a názov štruktúry do tabuľky *structure*. Po úspešnom uložení hodnôt do nerelačnej databázy je vrátený záznam v konkrétnom bloku.



Obrázok 53. Dátový model, ktorý udržiava štruktúru dát nerelačnej databázy

[9] Problematiku vyhľadávania v nerelačných databázach sme prezentovali a diskutovali na IEEE konferencii Multi-Conference on Systems, Signals & Devices v Afrike (Tunisko) - 22. - 25. marec 2021

V prípade získavania hodnôt z nerelačnej databázy je príkaz na výber dát presunutý priamo do programu, kde dochádza k nasledujúcemu prístupu:

- príkaz je automaticky získaný a následne prekonvertovaný z nerelačného dotazu na dotaz relačný,
- príkaz získa objekt, prípadne objekty, ktoré vyhovujú kritériu a následne sú nám vrátené všetky atribúty *objectId*, ktoré spĺňajú kritérium,
- množstvo záznamov, ktoré získame z JSON formátu sa v nerelačných databázach často líši typom a aj množstvom atribútov, preto sme tento aspekt museli vyriešiť prehľadaním celého JSON-u ako je ukázané v 5 príkaze,
- tieto záznamy boli automaticky vložené do relačnej databázy za účelom vytvorenia stálej štruktúry. Pre účely jednoduchšej inkrementácie primárneho kľúča sme vytvorili 3 sekvencie,
- následne sú všetky prvky v nerelačnej databáze prehľadne vložené ako výstup.

```
var mongoToSqlConverter = require("mongo-to-sql-converter")
const MongoDBQuery = "db.user.find({age: {$gte: 21}, name: 'julio', contribs: { $in: [ 'ALGOL', 'Lisp' ]}}, {name: 1, _id: 1});"
```

```
const SQLQuery = mongoToSqlConverter.convertToSQL(MongoDBQuery, true)
```

```
console.log (SQLQuery)                                                    (1)
```

```
select objectId, table_name from nosql_data
join structure using (structure_id_structure)
join attribute using (structure_id_structure)
where name = ""
and type = "";                                                            (2)
```

```
for(var exKey in exjson) {

 console.log("key:"+exKey+", value:"+exjson[exKey]);                      (3)
}
```

```
CREATE SEQUENCE attribute_seq
START WITH 1
INCREMENT BY 1;                                                           (4)
```

```
CREATE SEQUENCE structure_seq
START WITH 1
INCREMENT BY 1;                                                           (5)
```

```
CREATE SEQUENCE nosql_data_seq
START WITH 1
INCREMENT BY 1;                                                           (6)
```

```
INSERT INTO structure (id_structure, table_name)
VALUES (structure_seq.nextval, name);                                     (7)
```

```
INSERT INTO attribute (id_attribute,name, type, structure_id_structure)
SELECT attribute_se.nextval ,name, type, structure_id_structure FROM structure
WHERE (SELECT id_structure WHERE table_name = "name");                    (8)
```

```
INSERT INTO nosql_data (id_nosql_data, objectId, structure_id_structure)
SELECT nosql_data_seq.nextval, objectId, type FROM structure
WHERE (SELECT id_structure WHERE table_name = "name");                    (9)
```

```
db.table.find(ObjectId('4ecc05e55dd98a436ddcc47c'))                      (10)
```

Na základe uvedených príkazov a príkazu výberu informácií je možné konštatovať pri vyberaní jednoznačný fakt. Tým faktom je, že jednotlivé dáta sa budú vyberať na základe dátového typu a názvu atribútu. Pri nerelačnej databáze sa jedná o výber dát pomocou atribútu *objectId*. Z dôvodov jasnej štruktúry a dotazu na rovnaký typ dát nám pri zrýchlení času pomôže vytvorenie sekundárneho indexu aj v relačnej a aj nerelačnej databáze.

Príkaz pre vytvorenie relačnej databázy je označený (11) a príkaz pre vytvorenie sekundárneho indexu pre nerelačnú databázu je označený číslom (12).

```
create index rande_index on attribute (name, type);                      (11)
```

```
db.tableName.createIndex({"objectId" : 1})                               (12)
```

6.5.4 Experimentálna časť slúžiaca na overenie nášho prístupu

Pre overenie správnosti riešenia boli vytvorené 3 dokumenty, ktoré je možné nájsť na nasledovnej adrese *https://github.com/romanceresnak/transcom-searching*. Ako je vidieť z tabuľky 27 a následne z tabuľky 28 experimentálnu činnosť sme rozdelili do 2 typov experimentov. Prvým je experiment, ktorý sleduje ako sa vyhľadávanie v nerelačnej databáze MongoDB (2) menilo pri zväčšovaní množstva záznamov pri defaultnej databáze MongoDB a porovnáva výsledky s naším uvedeným princípom *"Our approach (1)".* Ako je vidieť z tabuľky 27, tak výsledky, ktoré sme dosiahli pri vyhľadávaní záznamu sa pri malom počte údajov v databáze, konkrétne 10, nepreukázali efektivitu nášho princípu. S týmto faktorom sme od začiatku počítali, nakoľko transformácia vyhľadávacieho príkazu napísaná v nerelačnej databáze musí byť najskôr zmenená na *sql* dotaz, potom pokračuje vyhľadávanie v relačnej štruktúre, prípadne dochádza k vytvoreniu nového záznamu a následne dochádza k vyhľadávaniu, čo zväčšuje celkový čas na získanie dát. Následne sme 10 záznamov vymazali a pokračovali experimentom s počtom údajov, ktorý sme stanovili na hranicu 10 000. Pri tomto údaji záznamov v nerelačnej databáze sa rýchlosť vyhľadávania dostala na takmer rovnakú hodnotu.

So zvyšujúcou hodnotou efektivita nášho riešenia stúpala a vyhľadávanie v databáze MongoDB pomaly degradovalo. Pred vložením 50 000 záznamov sme databázu znovu vymazali a testovali rýchlosť vyhľadávania práve pre nových 50 000 záznamov. Výsledok, ktorý sme získali bol očakávaným výsledkom a jasne preukázal efektívnosť a správnosť nášho riešenia.

Tabuľka 27. Dosiahnutý výsledok bez sekundárneho indexu

Počet záznamov v databáze	**Náš prístup (1)**	**MongoDB (2)**
10	0,006 s	0,002 s
10 000	0,010 s	0,009 s
50 000	0,011 s	0,013 s

Okrem zistenia, že naša navrhnutá metóda poskytuje rýchlejšie získanie dát nám poskytuje veľmi užitočnú vlastnosť. Pevnosť dátovej štruktúry je veľmi užitočná pri vytvorení sekundárneho indexu. Práve vytvorenie sekundárneho indexu nám umožňuje ešte viac zrýchliť vyhľadávanie. Výsledky, ktoré sme zaznamenali v tabuľke 27 boli namerané pri vytvorení sekundárneho indexu aj v relačnej databáze Oracle a aj nerelačnej databáze MongoDB. Pri porovnaní výsledkov z tabuľky 27 s tabuľkou 28 sú výsledky vo všetkých krokoch lepšie pri zavedení sekundárneho indexu. Podobne ako pri zaznamenávaní výsledkov do tabuľky 28 dochádzalo k rovnakému princípu. Najskôr boli vložené záznamy, ktoré boli poskytnuté na adrese https://github.com/romanceresnak/transcom-searching pre 10 záznamov. Po zapísaní hodnôt do tabuľky 28 bolo všetkých 10 záznamov vymazaných a následne vložené záznamy o veľkosti 10 000. Obdobne to fungovalo aj pre 50 000 záznamov.

Tabuľka 28. Dosiahnutý výsledok pomocou sekundárneho indexu

Počet záznamov v databáze	**Náš prístup (3)**	**MongoDB (4)**
10	0,0052 s	0,0021 s
10 000	0,0095 s	0,0085 s
50 000	0,0101 s	0,0119 s

Pre účely nerelačnej databázy sme využili server na adrese mlab.com, čo nám v konečnom dôsledku môže zvyšovať výsledky dosiahnuté pri získavaní dotazu a následným zaznamenaním. Server pre relačnú databázu Oracle mal konfiguráciu Enterprise Edition (3.94 GB) a Character Set mal nastavený na Default (*WEBMSWIN1252*).

Ostatné hodnoty, ktoré boli počas konfigurácie nastavené sú odporúčané nastavenie a je možné ich nájsť na nasledujúcej adrese:

https://docs.oracle.com/cd/E11882_01/server.112/e10897/install.htm#ADMQS023 .

Ako je vidieť z tabuľky 29**,** tak sledované hodnoty, ktoré sú pre nás podstatné a znamenajú pre nás pokrok oproti konvenčnej metóde sú zvýraznené zelenou farbou a negatívne aspekty našej metódy sú zvýraznené červenou farbou.

Zavedenie našej metódy do procesu môže pri prípadoch s malým počtom údajov spôsobiť pravý opak účelu, na ktorý bola metóda vytváraná a práve proces spomaliť. Ako je vidieť z tabuľky 29 a riadkov 7, 8 a 9, tak nám metóda spôsobovala určité oneskorenie pri správe údajov. Nakoľko zmena príkazu z jedného typu databázy na druhý typ si vyžaduje istý čas, je logickým faktom oneskorenie pri našej metóde.

Po aplikovaní nami navrhnutej metódy a zväčšenia počtu záznamov dochádzalo k negatívnemu účinku našej metódy. Ako je vidieť z tabuľky 29 a riadkov 10 a 11, tak nami navrhnutá metóda bola pri definovanom počte záznamov neefektívna. Počas tohto procesu sa v relačnej databáze vytvorila štruktúra, ktorá so zväčšujúcim počtom údajov dokázala zachytiť dátovú štruktúru, a od tohto bodu nebolo potrebné vykonávať žiadne dodatočné pridávanie hodnôt do relačnej databázy.

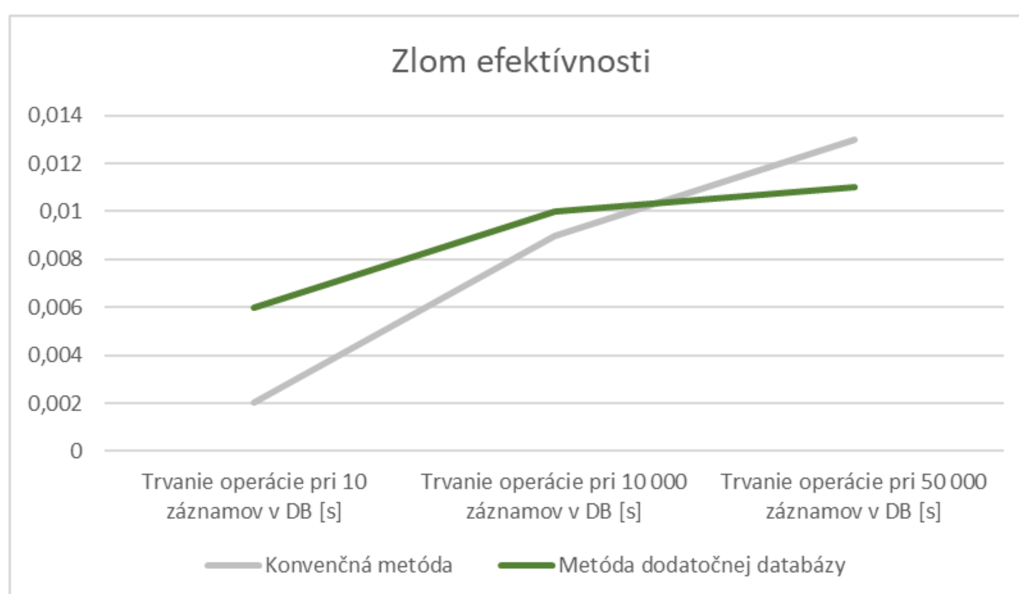
Tabuľka 29. Porovnaie sledovaných vlastností medzi konvenčnou metódou a metódou dodatočnej databázy

Číslo vlastnosti	Sledovaná vlastnosť	Konvenčná metóda	Metóda dodatočnej databázy
1	Pevná štruktúra [Áno/Nie]	Nie	Áno
2	Sledovanie na základe dátových typov [Áno/Nie]	Nie	Áno
3	Viaceré databázy [%]	Áno	Nie
4	Sekundárny index [Áno/Nie]	Nie	Áno
5	Možné oneskorenie [Áno/Nie]	Nie	Áno
6	Vytváranie dodatočnej štruktúry [Áno/Nie]	Nie	Áno
7	Primerný čas oneskorenia pri 10 záznamoch [s]	0	0,0001
8	Primerný čas oneskorenia pri 10 000 záznamoch [s]	0	0,002
9	Primerný čas oneskorenia pri 50 000 záznamoch [s]	0	0,005
10	Trvanie operácie pri 10 záznamov v DB [s]	0,002	0,006
11	Trvanie operácie pri 10 000 záznamov v DB [s]	0,009	0,01
12	Trvanie operácie pri 50 000 záznamov v DB [s]	0,013	0,011

Ako je vidieť z grafu na obr. 54 tak grafické zobrazenie výsledkov ukazuje na začiatku procesu výhodu konvenčnej metódy. Zelenou farbou sú zobrazené výsledky po implementovaní nášho prístupu a šedou sú zobrazené výsledky s použitím konvenčnej metódy.

Ako je vidieť z obr. 54, kde nám zelená farba predstavuje hodnoty po aplikovaní našej metódy (hodnoty bližšie k nule sú lepšie hodnoty), tak sme v prvej skupine prvkov, konkrétne 10 údajov, dosiahli horší výsledok. Nakoľko tento proces obsahoval malý počet údajov a štruktúra v relačnej databáze bola prázdna, muselo dôjsť k naplneniu štruktúry a typov údajov do relačnej databázy. Tento trend plnenia údajov pokračoval a pretrval aj pri vložení 10 000 údajov.

Po dosiahnutí hodnoty približne 12 000 sa krivka „zlomila" a efektívnosť metódy sa naplno prejavila. Účinok bol spôsobený vytvorením štruktúry v relačnej databáze, ktorá obsahovala takmer všetku štruktúru a dátové typy, ktoré sa mali do databázy vložiť.



Obrázok 54. Zlom krivky efektívnosti medzi konvenčnou metódou a metódou dodatočnej databázy

6.5.5 Zhrnutie výsledkov metódy zrýchľovania vyhľadávania dát v nerelačnej databáze MongoDB

Nerelačné databázy sú v súčasnosti veľmi populárne s čím súvisia aj ich pozitívne štatistiky, ktoré sa týkajú využitia, či už sa jedná o databázy dokumentové, *key-value* alebo *grafové*. Na základe narastajúcej popularity sa viacerí výskumníci zaoberali riešeniami, ktoré pomáhajú pri zefektívnení získavania dát a tým pádom poskytujú informácie, na ktoré sa používateľ priamo dopytuje.

Nami vytvorená metóda zaoberajúca sa navrhnutím modulu pri ktorom dochádza k vytvoreniu štruktúry v relačnej databáze Oracle na základe dát, ktoré boli poskytnuté ako výstup z nerelačnej databázy, je vytvorená 2 spôsobmi.

Prvým spôsobom bola implementácia vyhľadávania záznamov bez vytvorenia sekundárneho indexu nad dátovými typmi, ktoré boli uložené ako JSON v nerelačnej databáze MongoDB. Pri vhodnej implementácii a otestovaní riešenia sme nad dátami vytvorili sekundárny index, ktorý nám dokázal ešte viac zefektívniť riešenie a to aj pri masívnejšom náraste dát v nerelačnej databáze a tým pádom aj v relačnej databáze Oracle.

Výsledky, ktoré boli v našej práci vykonané v sekcii experimenty poukazujú na efektívnosť našej vytvorenej metódy. Oproti riešeniu, kedy dochádzalo k uloženiu záznamov v nerelačnej databáze MongoDB a následnému vyhľadávaniu v nerelačnej databáze spolu s vytvoreným sekundárnym indexom, sú dosiahnuté výsledky oproti predvolenému riešeniu efektívnejšie o 12 percent, čo sa týka časového hľadiska.

7 Výsledky práce a diskusia

V tejto kapitole popíšeme metódy, ktoré sme počas doktorandského štúdia dosiahli, a taktiež ich ohraničenie.

7.1 Výsledky práce

1. ***Metóda synchronizačnej bariéry*** - Bariéra slúži na úpravu prichádzajúcich dát (real-time dáta) do systému počas toho, keď sa už začal proces s nahromadenými dátami, ktoré boli v systéme už dlhšiu dobu. Hlavnou úlohou synchronizačnej bariéry je znížiť počet úprav, ktoré by museli byť vykonané po skončení procesu. V každej situácii, kedy operácia príde do procesu, je vyvolaná bariéra, ktorá skontroluje, či operácia s dátami ovplyvní práve sa transformujúce dáta. Spomenutá metóda má výrazný vplyv na dĺžku transformačného procesu a významne redukuje počet dodatočných úprav dát v cieľovej databáze.
2. ***Verzionovacia metóda*** - Verzionovanie slúži na účely zvýšenia spoľahlivosti systému. V ľubovoľnom okamihu môže byť transformačný proces, ktorý spracováva veľké množstvo údajov prerušený, čo by viedlo k opätovnému spracovaniu údajov. Na základe verzionovania sme vytvorili proces, ktorý pri spustení zistí, či sa daný objekt alebo hodnota objektu už v procese vyskytla a v prípade, že už v procese naozaj bola, transformačný proces pomocou tejto metódy preskočí danú hodnotu objektu, čo ma za následok zrýchlenie procesu a zníženie vyťaženia servera.
3. ***Metóda paralelizmu výpočtu*** - Vytvorená metóda má za účel sa automaticky prispôsobovať zvýšenému respektíve zníženému dopytu na výpočtové jednotky. Na základe veľkosti dát, ktoré do systému prichádzajú a rýchlosti spracovania, sú vyvolané respektíve redukované paralelné procesy. Vytvorené paralelné procesy sú riadené hlavným vláknom, ktoré deleguje jednotlivé spracovanie. V prípade, ak sa jednotlivé procesy ovplyvňujú, tak sa hlavné vlákno postará o vytvorenie vlastného potomka, ktorý vstúpi do procesu ako mediátor procesu a vyrieši daný problém s kolíziou.

4. ***Metóda replikačného koeficientu*** - Vytvorená metóda slúži na posudzovanie replikačného koeficientu pre dáta, ktoré sú distribuovane spracovávané. Na základe prístupových hodnôt, ako je počet čítaní, počet zapisovaní do každej tabuľky, ktoré sme získali z databázy pomocou príkazu, vieme na základe pravidiel definovať potrebnosť resp. užitočnosť dát. Z dôvodu neustálych zmien, čo sa týka čítania a zapisovania, sme vytvorili mechanizmus automatických úprav pre jednotlivé tabuľky na základe meniacich sa hodnôt či už sa jedná o masívne čítanie, prípadne zmazanie tabuľky.
5. ***Metóda redukcie duplicít*** - Vytvorená metóda slúži ako z názvu vyplýva z redukcie množstva prijatých hodnôt do systému. Z dôvodu akceptovania hodnôt z viacerých zdrojov sme boli nútení sledovať všetky prichádzajúce hodnoty do systému, a v prípade výskytu rovnakých dát vytvárať referencie na objekty respektíve hodnoty pre viaceré zdroje.
6. ***Metóda paralelizmu procesov*** - Presun dát je vždy spojený s istým čakaním. Z dôvodu závislosti na jednotlivých tabuľkách a vykonávania procesu sekvenčne sme vytvorili algoritmus, ktorý na začiatku zistí funkčnú závislosť na jednotlivých tabuľkách pri zistení závislosti primárnych, cudzích a kompozitných kľúčov a zistí, ktoré tabuľky môžu byť spracované paralelne. V tom prípade dochádza k spusteniu všetkých tabuliek paralelne a nie sekvenčne.
7. ***Metóda bezpečnosti práce s údajmi*** - Bezpečnosť dát patrí k dôležitej súčasti našej práce. Namiesto ukladania dát do viacerých zón dostupnosti sme sa rozhodli ukladať dáta do viacerých regiónov, čo nám redukuje možnosť straty údajov pri distribuovane spracovaných záznamoch. Taktiež sme implementovali spôsob, akým používateľ má možnosť pracovať s dátami a tým redukovali nežiaduci vstup aplikácie alebo používateľa k našim dátam.
8. ***Metóda manažmentu dát*** - Navrhnutá metóda má za účel efektívne spravovať aktívne a pasívne hodnoty, a tým redukovať množstvo dát uložených v primárnej databáze. Za týmto účelom sme vytvorili metódu, ktorá na základe časových hodnôt a dopytu k jednotlivým dátam presúva dáta obidvomi smermi, a to z databázy do dátového skladu alebo z dátového skladu do databázy. Tento manažment priniesol viaceré výhody akými sú menšie vyťaženie servera, rýchlejšie dopytovanie a menšia strata údajov, v prípade výpadku.

9. ***Metóda vytvorenia štruktúry dát*** - Hlavným účelom navrhnutej metódy je uchovávať pomocou štruktúry v relačnej databáze štruktúru objektov v nerelačných databázach. Na základe dopytovaných hodnôt, ktoré sa môžu v nerelačných metódach často odlišovať vieme redukovať množstvo hodnôt, ktoré spĺňajú kritérium na základe napísaného príkazu. Pre zrýchlenie tohto procesu sme vytvorili sekundárny index, ktorý nám v tomto ohľade pomôže ešte viacej zrýchliť proces vyhľadávania v relačnej databáze.
10. ***Metóda strojového učenia*** - Vytvorená metóda má za účel zrýchliť proces vyhľadávania v databázach. Prvotnou myšlienkou bolo zrýchliť vyhľadávanie v nerelačných databázach ako je *MongoDB* a *DynamoDB*, ale tento model sme implementovali aj pre relačné databázy *Oracle* a *MySql*. Algoritmus funguje na 2 princípoch. Prvým princípom je sledovanie príkazu, ktorý používateľ píše a predčasne spustí tento proces. Druhým princípom je predpokladať na základe strojového učenia aké príkazy boli spúšťané a výsledky presunúť do pamäte, a tým pádom príkaz nespustiť priamo nad spustenou databázou, ale nad databázou v pamäti.

7.2 Diskusia

Metódy, ktoré sme navrhli počas práce fungujú v mnohých prípadoch efektívne a zrýchľujú, respektíve zefektívňujú požadované procesy. Z dôvodu masívnejších testov však boli pri algoritmoch a navrhnutých metódach zistené situácie, kedy implementované procesy nespĺňali svoje účel.

1. ***Metóda synchronizačnej bariéry*** - Synchronizačná bariéra je efektívne pracujúca metóda, ktorá dokáže ovplyvňovať nahromadené dáta dátami, ktoré práve vstupujú do procesu. Pri experimentálnom testovaní sme však zistili, že pri veľkom počte údajov nemusí bezpodmienečne dôjsť k úplne všetkým úpravám počas transformácie a je nutná dodatočná úprava už transformovaných dát.

2. ***Verzionovacia metóda -*** Nami zvolený spôsob verzionovania dát priniesol efektívny spôsob, akým dokážeme efektívne vynechať objekty, ktoré pri prerušenom alebo neúspešnom transformačnom procese boli už uložené. Aj keď správa verzií jednotlivých objektov funguje správne a efektívne, pri narastajúcom počte verzií jednotlivých objektov nám začala efektivita riešenia degradovať.
3. ***Metóda paralelizmu výpočtu -*** Paralelizmus procesu funguje efektívne z pohľadu posudzovania potrebného výkonu pre spracovanie prichádzajúcich dát. Pri experimentoch sa nám 5-krát stalo, že počas výpočtu a klesania výpočtových jednotiek do systému prišli dáta, ktoré potrebovali o 1 výpočtovú jednotku viac ako bol aktuálny stav. Tento problém nebol zohľadnený pri výpočte, čo spôsobilo pokles o 1 jednotku a následne bolo potrebné aktualizovať výpočtovú jednotku o 2.
4. ***Metóda replikačného koeficientu -*** Stanovenie replikácií na základe databázových požiadaviek reálne odráža ich využiteľnosť. Nedostatkom tejto metódy je otváranie spojenia medzi programom a databázou, s čím môžu súvisieť aj bezpečnostné problémy. Hlavný nedostatok však vidíme v ukladaní replikačného koeficientu pre konkrétne tabuľky v jednoduchom súbore v S3, ku ktorému majú prístup všetci používatelia.
5. ***Metóda redukcie duplicít -*** Pri metóde redukcie duplicít sme efektívnym spôsobom potlačili prichádzajúce duplicity z viacerých zdrojov do nášho systému. Na základe navrhnutej správy dát je vytvorená referencia poskytujúca jednotkovú zložitosť, avšak tu vznikol problém s udržiavaním spoľahlivosti dát. Ak by sa dáta stratili alebo by boli poškodené, došlo by k strate údajov pre viaceré zdroje. Preto sme museli vytvoriť viaceré replikácie, s čím súvisí aj veľká nákladnosť procesu a dátového úložiska.
6. ***Metóda paralelizmu procesov -*** Paralelizmus procesov patrí k dôležitej metóde, ktorá urýchľuje proces migrácie dát z jedného typu databázy do druhého. Nedostatkom tohto návrhu je, že v určitých procesoch môže vytvorená metóda degradovať z paralelného procesu na sekvenčný proces.

7. ***Metóda bezpečnosti práce s údajmi -*** Práca s údajmi tvorí kľúčovú zložku pri práci s dátami. Pridaním dodatočných kontrol, čo sa týka prístupu k dátam a spracovaním vo viacerých dostupných regiónoch sme síce redukovali počet vstupov pre jednotlivých používateľov a zvýšili bezpečnosť, ale dodatočná kontrola dát sa prejavila 3 % vyťažením servera, a taktiež predĺžila čas potrebný na získanie hodnôt.
8. ***Metóda manažmentu dát -*** Algoritmus manažmentu posudzuje na základe určitých časových hodnôt a manipulácie s dátami ich presun z databázy do dátového skladu, respektíve z dátového skladu do databázy. Aj keď tento proces funguje efektívne, v mnohých prípadoch, existujú neočakávané situácie, kedy používateľ chce získať dáta, ktoré sa v databáze nenachádzajú a je potrebné ich získať z dátového skladu. V konečnom dôsledku táto požiadavka veľkým spôsobom nespomaľuje získanie dát, ale pri tejto operácii sú potrebné dodatočné kroky.
9. ***Metóda vytvorenia štruktúry dát -*** Vytvorená metóda pracuje efektívne s veľkou škálou dátových objektov. Existujú však prípady, kedy musí metóda na základe nerozoznaného dátového typu oproti požadovanej hodnote v nerelačnej databáze vykonávať 2 konverzie. Prvá konverzia je pri ukladaní dát a druhá je pri dopyte dát. Obe konverzie spôsobujú nárast času potrebného na dopyt údajov.
10. ***Metóda strojového učenia -*** Vytvorená metóda funguje efektívne, avšak počas experimentov sme zistili 2 situácie, ktoré súvisia so strojovým učením. Na začiatku sme potrebovali, aby metóda mala dostatočný počet otestovaných pokusov nad dátami, čo v niektorých momentoch znamenalo potrebu presunutia veľkého množstva údajov. Druhý nedostatok bol opäť pri malom počte údajov, ale súvisel s podobným názvom tabuliek.

Záver

V predkladanej práci sme riešili problémy, ktoré súvisia so stále narastajúcou potrebou rýchlejšej a efektívnejšej manipulácie s dátami, ktoré sú ovplyvnené dátami v reálnom čase. Pri tomto procese sme ukladali dáta do nerelačných databáz akými boli MongoDB a DynamoDB, s čím súvisí aj zvyšujúci nárok na rýchlosť získavania údajov.

Prvá časť práce je viac-menej teoretickou časťou. V práci sa venujeme štandardným (konvenčným) riešeniam, ktoré súvisia s viacerými výzvami, ktoré sme museli zohľadniť pri riešení distribuovane spracovaných dát a taktiež výzvy, ktoré sa týkali procesu vyhľadávania v nerelačných databázach. Zistili sme, že v mnohých prípadoch sú dáta, ktoré prichádzajú do systému počas transformačného procesu zanedbané a následne až po ukončení procesu je s dátami manipulované. Tým pádom dochádza k vyhľadávaniu dát a následnej úprave po skončení transformačného procesu, čo v konečnom dôsledku zvyšuje čas potrebný na celkovú úpravu. Ako ďalší skúmaný problém sme videli v neštandardizovanom spôsobe manipulácie s heterogénnymi dátami, ktoré pri vyhľadávaní spôsobujú značné problémy.

Analýzou existujúcich riešení, algoritmov a systémov sme zistili, že neexistujú komplexné riešenia, ktoré by dokázali na základe univerzálnych modulov riešiť viaceré výzvy pri distribuovanom spracovaní dát. Pri skúmaní riešení, ktoré sa venujú vyhľadávaniu dát, keďže práve nerelačné databázy slúžia aj na ukladanie rozsiahlych dát sme taktiež navrhli viaceré spôsoby, ktoré v konečnom dôsledku dokážu zrýchliť proces vyhľadávania v nerelačných databázach, ale dokonca aj v relačných databázach.

Kľúčovú zložku v našom výskume tvorí zvyšovanie paralelizmu spracovania údajov. Spomenutý aspekt bol súčasťou mnohých diskusných príspevkov a fór, kde sa výskumníci snažili odhaliť ideálnu hodnotu či už vyťaženia servera, počtu vlákien alebo množstva procesov, ktoré sú ideálne pre spracovanie veľkého množstva údajov. Na základe diskusií sme vytvorili princíp automatického prispôsobovania výpočtových jednotiek pri zachovaní optimálnych podmienok pri správnom zaťažení servera a pri efektívnej správe nákladov.

Významný spôsob ovplyvňovania dát vidíme v návrhu synchronizačnej bariéry, ktorá dokáže efektívne vstupovať do procesu práve sa transformujúcich dát (transformujú sa dáta, ktoré boli v procese dlhšiu dobu) a tým pádom na základe operácie aktualizovania, prípadne zmazania údajov, dokáže urýchliť tento proces v časovom rozmedzí od niekoľko

sekúnd až po desiatky minút. Pomocou tohto spôsobu sa na základe dát, ktoré vstúpili do systému môžu zmazať hodnoty od 1 záznamu až po hodnoty celej transformujúcej sa tabuľky až po hodnotu celej kolekcie.

Bezpečnosť pri transformačných procesoch je v súčasnosti veľmi žiadaná, a preto aj zachovanie bezpečnosti či už transformačného procesu (vyriešili sme vytvorením algoritmu na verzionovanie dát), bezpečnosť už uložených dát (vyriešili sme pomocou automatického presúvania údajov medzi databázou a dátovým skladom) alebo potlačením duplicít pri transformácii dát (vyriešili sme sledovaním vstupných hodnôt a vytváraním referencií) zohrali pri našom výskume dôležitú zložku.

Vyhľadávanie pomocou príkazu *Select* v relačných databázach, respektíve príkaz *Find* v nerelačných databázach považujeme taktiež za dôležitú súčasť nášho výskumu. Na základe väčšej flexibility ukladania záznamov v nerelačných databázach sme vytvorili viaceré metódy, ktorých účelom je získať dáta z dátového úložiska rýchlejšie, vytvoriť istý spôsob udržiavania pevnej štruktúry dát tak ako to poznáme z relačných databáz, a taktiež efektívnu manipuláciu s vyrovnávacou pamäťou. Ako veľký posun v tomto smere sme vytvorili modul pomocou strojového učenia, ktorý dokáže sledovať rozpísaný príkaz v relačnej databáze Oracle a nerelačných databázach MongoDB a DynamoDB a ešte pred potvrdením príkazu predčasne presunúť dáta do vyrovnávacej pamäti, čo nám v mnohých prípadoch urýchlilo proces získavania záznamov.

Počas transformačného procesu, kedy dochádzalo k presunu údajov z relačného typu databáz (pomocou stanovených pravidiel) do nerelačného úložiska bol proces oproti opačnému spôsobu relatívne jednoduchší ako pri presune dát opačným smerom. Tento problém súvisel s voľnosťou štruktúry v nerelačných databázach. Pri tomto presune sme museli efektívnym spôsobom riešiť aj problém súvisiaci s nekompatibilnosťou dátových typov a hlavne neúplnosťou vyžadovaných údajov a taktiež referenčnou integritou.

Sumarizácia dosiahnutých cieľov:

- Navrhli sme architektúru, ktorá dokáže v reálnom čase zachytávať prichádzajúce dáta, a tým ovplyvniť práve sa transformujúce údaje.
- Navrhli sme postup ako efektívne manipulovať s výkonnosťou systému, a tým efektívne sa prispôsobovať vyťaženiu servera.
- Navrhli sme metódu na automatické prispôsobovanie počtu replikácií v distribuovanom systéme na základe databázových procesov.

- Vytvorili sme metódu na redukciu duplicít v reálnom čase, a tým sme redukovali aj množstvo úložného priestoru.
- Vytvorili sme architektúru, ktorá dokáže efektívne pracovať s meniacou sa dátovou štruktúrou v nerelačných databázach.
- Vytvorili sme princíp a spôsob manažovania hodnôt z nerelačných databáz MongoDB a DynamoDB pomocou vyrovnávacej pamäte.
- Navrhli a implementovali sme metódu kontroly a správy manažmentu referenčných integrít pri transformačnom procese.
- Navrhli sme mechanizmus strojového učenia, ktorý na základe predchádzajúcich dopytov efektívne manažuje presun dát z relačných databáz Oracle a MySql a nerelačných databáz MongoDB a DynamoDB.
- Experimentálne sme overili výkonnosti vytvorených riešení a postupov na malých (1GB dát), stredných (100 GB dát), veľkých (1 TB dát) a veľmi veľkých (100 TB dát) vzorkách dát.

Vytvorené riešenia boli testované a použité pre prostredie v rozsiahlych dopravných systémoch, avšak poskytnuté riešenia nie sú pre tento proces od nich závislé, keďže sa jedná o všeobecné algoritmy nezávislé na vstupných dátach alebo objemu dát.

Príloha

Prílohu tvorí DVD.

Priložené DVD obsahuje dizertačnú prácu v elektronickej podobe (formát PDF).

Publikované práce

1. AFC

   Influencing migration processes by real-time data / Roman Čerešňák, Karol Matiaško, Adam Dudáš.
   In: Proceedings of the 28th conference of Open innovations association FRUCT [electronic]. - ISSN 2305-7254. - 1. vyd. - [S.l.]: FRUCT Oy, 2021. - ISBN 978-952-69244-4-1 (online). - s. 48-54.
   [Čerešňák Roman - Matiaško Karol - Dudáš Adam]

2. AFD

   Comparison of distributed data transformation and comparing query performance in relational and non-relational database / Roman Čerešňák, Michal Kvet.
   In: ICETA 2019 [electronic, print]: 17th IEEE International conference on emerging elearning technologies and applications: Information and communication technologies in learning: proceedings. - 1. vyd. - Denver: Institute of Electrical and Electronics Engineers, 2019. - ISBN 978-1-7281-4967-7. - s. 108-114 [online].
   [Čerešňák Roman - Kvet Michal]
   Comparison of query performance irelational a non-relation databases

3. AFD

   Comparison of apache technology in distributed environment / Roman Čerešňák, Michal Kvet.
   In: Information and digital technologies 2019 [electronic]: proceedings of the international conference. - 1. vyd. - Danvers: Institute of Electrical and Electronics Engineers, 2019. - ISBN 978-1-7281-1401-9. - s. 80-85.
   [Čerešňák Roman - Kvet Michal]

4. AFD

   Comparison of query performance in relational a non-relation databases / Roman Čerešňák, Michal Kvet.
   In: TRANSCOM 2019 [electronic]: conference proceedings. - ISSN 2352-1465. - 1. vyd. - Amsterdam: Elsevier Science, 2019. - s. 170-177
   [Čerešňák Roman - Kvet Michal]

5. ADF

Comparison of query performance between MySQL and MongoDB database [electronic] [Porovnanie výkonnosti dotazov medzi databázami MySQL a MongoDB] / Roman Čerešňák, Oľga Chovancová.
In: Central European Researchers Journal [print]. - ISSN 2453-7314. - Roč. 5, č. 1 (2019), s. 45-51
[Čerešňák Roman - Chovancová Oľga]

6. AFC

Using replication method to increase reliability in distributed information systems / Roman Čerešňák, Karol Matiaško - Conference Reliability Engineering and Computational Intelligence
[Čerešňák Roman - Matiaško Karol]

7. AFC

Synchronization Barrier in Database Migration Processes / Roman Čerešňák, Karol Matiaško - Conference ICETA 2021
[Čerešňák Roman - Matiaško Karol]

8. AFC

Versioning Data During Migration Processes in Cloud Environment / Roman Čerešňák, Karol Matiaško - Conference ICETA 2021
[Čerešňák Roman - Matiaško Karol]

9. AFC

Various proposed approaches eliminating duplicate data in a system / Roman Čerešňák, Karol Matiaško - Scientific journal Communications
[Čerešňák Roman - Matiaško Karol]

10. AFC

Improvement of Searching Data in MongoDB with the Usage of Oracle Database / Roman Čerešňák, Adam Dudáš - Conference SSD
[Čerešňák Roman - Dudáš Adam]

Články v procese schvaľovania:

1. AFC

   Implementing Machine Learning Methods in Searching Processes - Conference FRUCT29

2. AFC

   Improved the method of selecting data in a nonrelational database - Conference IDT 2021

3. AFC

   Mapping rules of transformation processes - Conference IDT 2021

Neindexované články

Transformation of Indexes between Relational and No-relational Databases for Distributed Data - Conference AGRIS
Improvement of Parallelism Process in Distributed Data Processing - Conferecne FRUCT
Examination Information System - Conference UNINFOS

Počet citácií : 4

Zoznam obrázkov

Obrázok 1. Diagram pre modul migrácie údajov 25
Obrázok 2. Diagram pre modul mapovania údajov 26
Obrázok 3. Model dátového toku prichádzajúcich dát s komponentmi 29
Obrázok 4. Model mikro-dávkového spracovania 29
Obrázok 5. Ukážka fungovania služby Amazon Kinesis 38
Obrázok 6. Dátový diagram prichádzajúcich dát 39
Obrázok 7. Architektúra pre zvýšenie škálovania 40
Obrázok 8. Architektúra pre zníženie škálovania 42
Obrázok 9. Automatický nárast a pokles vlákien 45
Obrázok 10. Porovnanie spracovania záznamov medzi dvomi metódami 47
Obrázok 11. Výsledok sql príkazu 54
Obrázok 12. Vzorový dátový model 56
Obrázok 13. Architektúra pre zvyšovanie replikačného koeficientu 57
Obrázok 14. Architektúra pre zníženie replikačného koeficientu 58
Obrázok 15. Navrhnutá architektúra 61
Obrázok 16. Porovnanie sledovaných ukazovateľov pri využití dvoch prístupov 62
Obrázok 17. Dátový model pre transformačné účely 70
Obrázok 18. Transformačná fáza 70
Obrázok 19. Transformačná fáza s využitím synchronizačnej bariéry 71
Obrázok 20. Štruktúra synchronizačnej bariéry 73
Obrázok 21. Dĺžka trvania transformačného procesu 76
Obrázok 22. Porovnanie sledovaných hodnôt medzi konvenčnou metódou a naším prístupom 78
Obrázok 23. Architektúra verzionovania, ktorá zobrazuje kroky na vloženie novej verzie a výber verzie 80
Obrázok 24. Transformačná architektúra 81
Obrázok 25. Ukážka revízie 84
Obrázok 26. Proces plnenia údajov 85
Obrázok 27. Odlišný čas medzi použitou metódou bez údajov v databáze 88
Obrázok 28. Odlišný čas medzi použitou metódou a údajmi v databáze 89
Obrázok 29. Grafické porovnanie výsledkov dosiahnutých pri verzionovaní procesu a dát 90
Obrázok 30. Základný dátový model 97
Obrázok 31. Rozšírený dátový model 97
Obrázok 32. Architektúra aplikácie 101
Obrázok 33. Monitorovanie áut pomocou GPS 101
Obrázok 34. Grafické zobrazenie redukcie bezpečnosti po aplikovaní našej metódy 103
Obrázok 35. Grafické zobrazenie klesania rýchlosti po aplikácií našej metódy do procesu 104
Obrázok 36. Návrh zachytávajúci dáta v reálnom čase a nahromadené dáta 109
Obrázok 37. Databázová tabuľka zákazník (customer) 109

Obrázok 38. Mapovacie pravidlo ... 111
Obrázok 39. Efektívnosť metód MapReduce vs. Hive ... 116
Obrázok 40. Grafické zobrazenie aspektu efektívnosti po aplikovaní metódy ovplyvňovania dát ... 119
Obrázok 41. Architektúra systému Index odporúčaný systémom ... 122
Obrázok 42. Porovnanie štruktúr indexov v pamäti a na disku ... 124
Obrázok 43. Organizácia v NoSql databáze založenej na grafoch ... 131
Obrázok 44. Výkon dopytu v milisekundách ... 134
Obrázok 45. Výkon jednotlivých príkazov pre relačnú databázu Oracle a nerelačnú databázu MongoDB ... 134
Obrázok 46. Algoritmus bočnej pamäte (Side-cache algorithm) ... 139
Obrázok 47. Algoritmus medzipamäte načítania ... 140
Obrázok 48. Medzipamäť na zápis ... 140
Obrázok 49. Application Load Balancer pre databázu v pamäti ... 141
Obrázok 50. Dátový model ... 142
Obrázok 51. Grafické porovanie výberu hodnôt pri 1 000 údajoch z vybratých DB ... 148
Obrázok 52. Grafické porovnanie výberu hodnôt pri 1 000 000 údajoch v DB ... 148
Obrázok 53. Dátový model, ktorý udržiava štruktúru dát nerelačnej databázy ... 154
Obrázok 54. Zlom krivky efektívnosti medzi konvenčnou metódou a metódou dodatočnej databázy ... 159

Zoznam tabuliek

Tabuľka 1. Konfigurácie servera 44
Tabuľka 2. Porovnanie sledovaných hodnôt medzi konvenčnou metódou a metódou správy záznamov 46
Tabuľka 3. Konfigurácia klastra 60
Tabuľka 4. Porovnanie sledovaných hodnôt medzi konvenčnou metódou a naším prístupom 62
Tabuľka 5. Konfigurácia klastra 75
Tabuľka 6. Porovanie sledovaných hodnôt medzi konvenčnou metódou a naším prístupom 77
Tabuľka 7. Primárny kľúč v Sql a primárny kľúč v NoSql 86
Tabuľka 8. Konfigurácia klastra 87
Tabuľka 9. Porovanie sledovaných hodnôt medzi verzionovaním procesu a verzionovaním dát 90
Tabuľka 10. Porovanie sledovaných hodnôt medzi konvenčnou metódou a metódou zvýšenia bezpečnosti 103
Tabuľka 11. Konfigurácia servera 114
Tabuľka 12. Tabuľka dát 115
Tabuľka 13. Tabuľka výkonnosti 115
Tabuľka 14. Nameraný čas potrebný na transformáciu údajov pomocou našej architektúry a bez našej architektúry pomocou technológie Hadoop 117
Tabuľka 15. Porovanie sledovaných hodnôt medzi konvenčnou metódou a metódou ovplyvňovania dát 118
Tabuľka 16. Rozdiel medzi relačnými a nerelačnými databázami 125
Tabuľka 17. Rozdielne vlastnosti medzi 3 najpoužívanejšími relačnými databázami 126
Tabuľka 18. Výkon rôznych typov relačných a nerelačných databáz pre 10 000 záznamov nameraných v milisekundách 133
Tabuľka 19. Výkon rôznych typov relačných a nerelačných databáz pre 100 000 záznamov nameraných v milisekundách 133
Tabuľka 20. Nameraný čas pre operácie (1), (2) a (3) 143
Tabuľka 21. Nameraný čas pre operácie (1) (2) (3) v databáze DynamoDB 143
Tabuľka 22. Štruktúra dát 144
Tabuľka 23. Nameraný čas pre databázu Redis 145
Tabuľka 24. Porovnanie výkonu dotazu 145
Tabuľka 25. Výkon dotazu v pamäti 146
Tabuľka 26. Porovnanie sledovaných hodnôt medzi konvenčnou metódou a metódou správy záznamov 147
Tabuľka 27. Dosiahnutý výsledok bez sekundárneho indexu 157
Tabuľka 28. Dosiahnutý výsledok pomocou sekundárneho indexu 157
Tabuľka 29. Porovnaie sledovaných vlastností medzi konvenčnou metódou a metódou dodatočnej databázy 158

Zoznam použitej literatúry

[1]. 2014. Proceedings of the ACM SIGMOD International Conference on Management of Data.

[2]. Abad, C., Lu, Y. and Campbell, R., 2011. DARE: Adaptive Data Replication for Efficient Cluster Scheduling. 2011 IEEE International Conference on Cluster Computing,.

[3]. Abawajy, J. and Deris, M., 2014. Data Replication Approach with Consistency Guarantee for Data Grid. IEEE Transactions on Computers, 63(12), pp.2975-2987.

[4]. Agrawal, S., Chaudhuri, S. and Narasayya, V., 2000. Automated selection of materialized views and indexes for SQL databases.

[5]. Agrawal, S., Chaudhuri, S. and Narasayya, V., 2001. Materialized view and index selection tool for Microsoft SQL server 2000. ACM SIGMOD Record, 30(2), p.608.

[6]. Ahanger, G. and Little, T., 2001. Data semantics for improving retrieval performance of digital news video systems. IEEE Transactions on Knowledge and Data Engineering, 13(3), pp.352-360.

[7]. Ameri, P., 2016. Challenges of index recommendation for databases [With specific evaluation on a NoSQL database].

[8]. Ananda, A. and Poo, G., 1995. Distributed systems: Concepts and design. Computer Communications, 18(7), pp.521-522.

[9]. Aouiche, K. and Darmont, J., 2009. Data mining-based materialized view and index selection in data warehouses. Journal of Intelligent Information Systems, 33(1), pp.65-93.

[10]. Barbierato, E., Gribaudo, M. and Iacono, M., 2014. Performance evaluation of NoSQL big-data applications using multi-formalism models. Future Generation Computer Systems, 37, pp.345-353.

[11]. Binani, S., Gutti, A. and Upadhyay, S., 2016. SQL vs. NoSQL vs. NewSQL-A Comparative Study. Communications on Applied Electronics, 6(1), pp.43-46.

[12]. Bjeladinovic, S., 2018. A fresh approach for hybrid SQL/NoSQL database design based on data structuredness. Enterprise Information Systems, 12(8-9), pp.1202-1220.

[13]. Board, R., 1993. Distributed Database Systems. IASSIST Quarterly, 16(3), p.4.

[14]. Boicea, A., Radulescu, F. and Agapin, L., 2012. MongoDB vs Oracle -- Database Comparison. 2012 Third International Conference on Emerging Intelligent Data and Web Technologies,.

[15]. Bruno, N. and Chaudhuri, S., 2005. Automatic physical database tuning. Proceedings of the 2005 ACM SIGMOD international conference on Management of data - SIGMOD '05,.

[16]. Callan, J., Lu, Z. and Croft, W., 2017. Searching Distributed Collections With Inference Networks. ACM SIGIR Forum, 51(2), pp.160-167.

[17]. Caprara, A., Fischetti, M. and Maio, D., 1995. Exact and approximate algorithms for the index selection problem in physical database design. IEEE Transactions on Knowledge and Data Engineering, 7(6), pp.955-967.

[18]. Casado, R. and Younas, M., 2014. Emerging trends and technologies in big data processing. Concurrency and Computation: Practice and Experience, 27(8), pp.2078-2091.

[19]. Celesti, A., Fazio, M., Romano, A., Bramanti, A., Bramanti, P. and Villari, M., 2018. An OAIS-Based Hospital Information System on the Cloud: Analysis of a NoSQL Column-Oriented Approach. IEEE Journal of Biomedical and Health Informatics, 22(3), pp.912-918.

[20]. Čerešňák, R. and Kvet, M., 2019. Comparison of query performance in relational a non-relation databases. Transportation Research Procedia, 40, pp.170-177.

[21]. Cervin, A., Eker, J., Bernhardsson, B. and Årzén, K., 2002. Feedback-feedforward scheduling of control tasks. Real-Time Systems, 23(1/2), pp.25-53.

[22]. Chen, F., Deng, P., Wan, J., Zhang, D., Vasilakos, A. and Rong, X., 2015. Data Mining for the Internet of Things: Literature Review and Challenges. International Journal of Distributed Sensor Networks, 11(8), p.431047.

[23]. Choice Reviews Online, 1997. The Computer science and engineering handbook. 35(04), pp.35-2165-35-2165.

[24]. Chung, W., Lin, H., Chen, S., Jiang, M. and Chung, Y., 2013. JackHare: a framework for SQL to NoSQL translation using MapReduce. Automated Software Engineering, 21(4), pp.489-508.

[25]. Cidon, A., Stutsman, R., Rumble, S., Katti, S., Ousterhout, J. and Rosenblum, M., 2013. MinCopysets : Derandomizing Replication In Cloud Storage.

[26]. Dayalan, M., 2018. MapReduce: Simplified Data Processing on Large Cluster. International Journal of Research and Engineering, 5(5), pp.399-403.

[27]. DB, S., 2021. SQL Vs Nosql Database Differences Explained With Few Example DB. [online] Thegeekstuff.com. Available at: <https://www.thegeekstuff.com/2014/01/sql-vs-nosql-db/> [Accessed 16 January 2021].

[28]. Deari, R., Zenuni, X., Ajdari, J., Ismaili, F. and Raufi, B., 2018. Analysis And Comparision of Document-Based Databases with Relational Databases: MongoDB vs MySQL. 2018 International Conference on Information Technologies (InfoTech),.

[29]. Díaz-Galiano, M., Martín-Valdivia, M., Montejo-Ráez, A. and Ureña-López, L., 2007. Improving Performance of Medical Images Retrieval by Combining Textual and Visual Information. 2007 Sixth Mexican International Conference on Artificial Intelligence, Special Session (MICAI),.

[30]. Díaz, M., Martín, C. and Rubio, B., 2016. State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing. Journal of Network and Computer Applications, 67, pp.99-117.

[31]. Elmore, A., Das, S., Agrawal, D. and El Abbadi, A., 2021. Zephyr : Live Migration in Shared Nothing Databases for Elastic Cloud Platforms Categories and Subject Descriptors.

[32]. Erevelles, S., Fukawa, N. and Swayne, L., 2016. Big Data consumer analytics and the transformation of marketing. Journal of Business Research, 69(2), pp.897-904.

[33]. Faerber, F., Kemper, A., Larson, P., Levandoski, J., Neumann, T. and Pavlo, A., 2017. Main Memory Database Systems. Foundations and Trends® in Databases, 8(1-2), pp.1-130.

[34]. Fan, W. and Bifet, A., 2013. Mining big data. ACM SIGKDD Explorations Newsletter, 14(2), pp.1-5.

[35]. Fan, X., Weber, W. and Barroso, L., 2007. Power provisioning for a warehouse-sized computer. Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07,.

[36]. Fox, P. and Friston, K., 2012. Distributed processing; distributed functions?. NeuroImage, 61(2), pp.407-426.

[37]. Freiknecht, J. and Papp, S., 2018. Apache Kafka. Big Data in der Praxis, pp.449-456.

[38]. Ghotiya, S., Mandal, J. and Kandasamy, S., 2017. Migration from relational to NoSQL database. IOP Conference Series: Materials Science and Engineering, 263, p.042055.

[39]. González-Aparicio, M., Ogunyadeka, A., Younas, M., Tuya, J. and Casado, R., 2017. Transaction processing in consistency-aware user's applications deployed on NoSQL databases. Human-centric Computing and Information Sciences, 7(1).

[40]. Goudarzi, M., 2019. Heterogeneous Architectures for Big Data Batch Processing in MapReduce Paradigm. IEEE Transactions on Big Data, 5(1), pp.18-33.

[41]. Govindan, S., Sivasubramaniam, A. and Urgaonkar, B., 2011. Benefits and limitations of tapping into stored energy for datacenters. Proceeding of the 38th annual international symposium on Computer architecture - ISCA '11,.

[42]. Grolinger, K., Hayes, M., Higashino, W., L'Heureux, A., Allison, D. and Capretz, M., 2014. Challenges for MapReduce in Big Data. 2014 IEEE World Congress on Services,.

[43]. Gu, L., Zeng, D., Guo, S. and Ye, B., 2015. Joint optimization of VM placement and request distribution for electricity cost cut in geo-distributed data centers. 2015 International Conference on Computing, Networking and Communications (ICNC),.

[44]. Guo, Y., Rao, J., Jiang, C. and Zhou, X., 2017. Moving Hadoop into the Cloud with Flexible Slot Management and Speculative Execution. IEEE Transactions on Parallel and Distributed Systems, 28(3), pp.798-812.

[45]. Hirzel, M., Soulé, R., Schneider, S., Gedik, B. and Grimm, R., 2014. A catalog of stream processing optimizations. ACM Computing Surveys, 46(4), pp.1-34.

[46]. Hueske, F. and Walther, T., 2019. Apache Flink. Encyclopedia of Big Data Technologies, pp.51-58.

[47]. IEEE Security & Privacy Magazine, 2011. Usenix 2011 Trade Advertisement. 9(1), pp.c2-c2.

[48]. Imran, S. and Hyder, I., 2009. Security Issues in Databases. 2009 Second International Conference on Future Information Technology and Management Engineering,.

[49]. Jacobs, I. and Bean, C., 1963. Fine Particles, Thin Films and Exchange Anisotropy (Effects of Finite Dimensions and Interfaces on the Basic Properties of Ferromagnets). Spin Arrangements and Crystal Structure, Domains, and Micromagnetics, pp.271-350.

[50]. Janech, J., Tavac, M. and Kvet, M., 2019. Versioned database storage using unitemporal relational database. 2019 IEEE 15th International Scientific Conference on Informatics,.

[51]. Jin, H., Cheocherngngarn, T., Levy, D., Smith, A., Pan, D., Liu, J. and Pissinou, N., 2013. Joint Host-Network Optimization for Energy-Efficient Data Center Networking. 2013 IEEE 27th International Symposium on Parallel and Distributed Processing,.

[52]. Kamal, J., Murshed, M. and Buyya, R., 2016. Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable OLTP applications. Future Generation Computer Systems, 56, pp.421-435.

[53]. Kamburugamuve, S., Fox, G., Leake, D. and Qiu, J., 2013. Survey of distributed stream processing for large stream sources.

[54]. Karnitis, G. and Arnicans, G., 2015. Migration of Relational Database to Document-Oriented Database: Structure Denormalization and Data Transformation. 2015 7th International Conference on Computational Intelligence, Communication Systems and Networks,.

[55]. Katsifodimos, A. and Schelter, S., 2016. Apache Flink: Stream Analytics at Scale. 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW),.

[56]. Khan, O., Burns, R., Plank, J., Pierce, W. and Huang, C., 2019. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads.

[57]. Khare, A., 2016. A Review of NoSQL Databases , Types and Comparison with Relational Database.

[58]. Kleinrock, L., 1985. Distributed systems. Communications of the ACM, 28(11), pp.1200-1213.

[59]. Kleppmann, M., 2018. Samza. Encyclopedia of Big Data Technologies, pp.1-8.

[60]. Klettke, M., Störl, U. and Scherzinger, S., 2015. Schema extraction and structural outlier detection for JSON-based NoSQL Data Stores.

[61]. Kuderu, N. and Kumari, V., 2016. Relational Database to NoSQL Conversion by Schema Migration and Mapping. International Journal of Computer Engineering in Research Trends, 3(9), p.506.

[62]. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J., Ramasamy, K. and Taneja, S., 2015. Twitter Heron. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data,.

[63]. Kuznetsov, S. and Poskonin, A., 2014. NoSQL data management systems. Programming and Computer Software, 40(6), pp.323-332.

[64]. Kvet, M. and Matiasko, K., 2017. Time as the Important Factor of the DataRetrieval – Table Type Classification. Advances in Intelligent Systems and Computing, pp.492-502.

[65]. Kvet, M., 2019. Data Distribution in Ad-hoc Transport Network. 2019 International Conference on Information and Digital Technologies (IDT),.

[66]. Kvet, M., Matiaško, K. and Kvet, M., 2014. Complex time management in databases. Open Computer Science, 4(4).

[67]. Kvet, M., Toth, S. and Krsak, E., 2019. Concept of temporal data retrieval: Undefined value management. Concurrency and Computation: Practice and Experience, 32(13).

[68]. Le Merrer, E. and Le Scouarnec, N., 2016. Efficient User Opt-Out from Block Stores. 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW),.

[69]. Lei, H., Blount, M. and Tait, C., 1999. DataX: an approach to ubiquitous database access. Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications,.

[70]. Li, C., 2010. Transforming relational database into HBase: A case study. 2010 IEEE International Conference on Software Engineering and Service Sciences,.

[71]. Li, S., Jiang, H. and Shi, M., 2017. Redis-based web server cluster session maintaining technology. 2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD),.

[72]. Liu, X., Iftikhar, N. and Xie, X., 2014. Survey of real-time processing systems for big data. Proceedings of the 18th International Database Engineering & Applications Symposium on - IDEAS '14,.

[73]. Liu, Y., Wang, Y. and Jin, Y., 2012. Research on the improvement of MongoDB Auto-Sharding in cloud environment. 2012 7th International Conference on Computer Science & Education (ICCSE),.

[74]. Liu, Z., Lin, M., Wierman, A., Low, S. and Andrew, L., 2015. Greening Geographical Load Balancing. IEEE/ACM Transactions on Networking, 23(2), pp.657-671.

[75]. Lourenço, J., Cabral, B., Carreiro, P., Vieira, M. and Bernardino, J., 2015. Choosing the right NoSQL database for the job: a quality attribute evaluation. Journal of Big Data, 2(1).

[76]. Manegold, S., 2002. Understanding, modeling, and improving main-memory database performance.

[77]. Marshall, I. and Roadknight, C., 1998. Linking cache performance to user behaviour. Computer Networks and ISDN Systems, 30(22-23), pp.2123-2130.

[78]. Mehmood, N., Culmone, R. and Mostarda, L., 2017. Modeling temporal aspects of sensor data for MongoDB NoSQL database. Journal of Big Data, 4(1).

[79]. Mo, X. and Wang, H., 2012. Asynchronous Index Strategy for high performance real-time big data stream storage. 2012 3rd IEEE International Conference on Network Infrastructure and Digital Content,.

[80]. Naheman, W. and Jianxin Wei, 2013. Review of NoSQL databases and performance testing on HBase. Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC),.

[81]. Nasiri, H., Nasehi, S. and Goudarzi, M., 2019. Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities. Journal of Big Data, 6(1).

[82]. Okman, L., Gal-Oz, N., Gonen, Y., Gudes, E. and Abramov, J., 2011. Security Issues in NoSQL Databases. 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications,.

[83]. Oliver, A., 2014. Storm or Spark: Choose your real-time weapon.

[84]. Pala, I., 2014. BMC Medical Informatics and Decision Making. BMC Medical Informatics and Decision Making, 14(1).

[85]. Patel, J., 2015. From Data to Insights @ Bare Metal Speed. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data,.

[86]. Peng, D. and Dabek, F., 2019. Large-scale incremental processing using distributed transactions and notifications.

[87]. Qader, M., Cheng, S. and Hristidis, V., 2018. A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases. Proceedings of the 2018 International Conference on Management of Data,.

[88]. Qian, L., Luo, Z., Du, Y. and Guo, L., 2009. Cloud Computing: An Overview BT - Cloud Computing. pp.626-631.

[89]. Qureshi, A., Weber, R., Balakrishnan, H., Guttag, J. and Maggs, B., 2009. Cutting the electric bill for internet-scale systems. ACM SIGCOMM Computer Communication Review, 39(4), pp.123-134.

[90]. Rawat, D. and Alshaikhi, A., 2018. Leveraging Distributed Blockchain-based Scheme for Wireless Network Virtualization with Security and QoS Constraints. 2018 International Conference on Computing, Networking and Communications (ICNC),.

[91]. Reniers, V., Rafique, A., Van Landuyt, D. and Joosen, W., 2017. Object-NoSQL Database Mappers: a benchmark study on the performance overhead. Journal of Internet Services and Applications, 8(1).

[92]. Sagiroglu, S. and Sinanc, D., 2013. Big data: A review. 2013 International Conference on Collaboration Technologies and Systems (CTS),.

[93]. Scavuzzo, M., Di Nitto, E. and Ceri, S., 2014. Interoperable Data Migration between NoSQL Columnar Databases. 2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations,.

[94]. Schneider, S., Hirzel, M., Gedik, B. and Wu, K., 2012. Auto-parallelizing stateful distributed streaming applications. Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12,.

[95]. Sharma, K. and Attar, V., 2016. Generalized Big Data Test Framework for ETL migration. 2016 International Conference on Computing, Analytics and Security Trends (CAST),.

[96]. Shidong Huang, Lizhi Cai, Zhenyu Liu and Yun Hu, 2012. Non-structure Data Storage Technology: A Discussion. 2012 IEEE/ACIS 11th International Conference on Computer and Information Science,.

[97]. Silva, B., Diyan, M. and Han, K., 2018. Big Data Analytics. Deep Learning: Convergence to Big Data Analytics, pp.13-30.

[98]. Smith, G., 1991. Modeling security-relevant data semantics. IEEE Transactions on Software Engineering, 17(11), pp.1195-1203.

[99]. Srinivas, S. and Nair, A., 2015. Security maturity in NoSQL databases - are they secure enough to haul the modern IT applications?. 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI),.

[100]. Tan, Y., Ko, R. and Holmes, G., 2013. Security and Data Accountability in Distributed Systems: A Provenance Survey. 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing,.

[101]. Tidke, S., 2017. MonogDB: Data management in NoSQL. Privacy and Security Policies in Big Data, pp.64-91.

[102]. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S. and Ryaboy, D., 2014. Apache Storm. Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data,.

[103]. Valentin, G., Zuliani, M., Zilio, D., Lohman, G. and Skelley, A., n.d. DB2 advisor: an optimizer smart enough to recommend its own indexes. Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073),.

[104]. Vavilapalli, V., Murthy, A., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B. and Baldeschwieler, E., 2013. Apache Hadoop YARN: Yet another resource negotiator. Proceedings of the 4th annual Symposium on Cloud Computing,.

[105]. Vohra, D., 2016. Apache Sqoop. Practical Hadoop Ecosystem, pp.261-286.

[106]. VOLUME-8 ISSUE-10, AUGUST 2019, REGULAR ISSUE, 2019. Database Migration using Data Synchronization and Transactional Replication. 8(10), pp.2730-2734.

[107]. W, K., 2010. SQL vs. NoSQL.

[108]. Wei, Q., Veeravalli, B., Gong, B., Zeng, L. and Feng, D., 2010. CDRM: A Cost-Effective Dynamic Replication Management Scheme for Cloud Storage Cluster. 2010 IEEE International Conference on Cluster Computing,.

[109]. Wei, Z., Dejun, J., Pierre, G., Chi, C. and van Steen, M., 2008. Service-oriented data denormalization for scalable web applications. Proceeding of the 17th international conference on World Wide Web - WWW '08,.

[110]. Xia, Q., Liang, W. and Xu, Z., 2014. Data Locality-Aware Query Evaluation for Big Data Analytics in Distributed Clouds. 2014 Second International Conference on Advanced Cloud and Big Data,.

[111]. Xiong, M., Ramamritham, K., Haritsa, J. and Stankovic, J., n.d. MIRROR: a state-conscious concurrency control protocol for replicated real-time databases. Proceedings of International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems. (Cat. No.PR00334),.

[112]. Xplenty, 2017. The SQL vs NoSQL Difference: MySQL vs MongoDB.

[113]. Yang, C., Fu, C. and Hsu, C., 2009. File replication, maintenance, and consistency management services in data grids. The Journal of Supercomputing, 53(3), pp.411-439.

[114]. Yang, S., 2012. Move into the Cloud, shall we?. Library Hi Tech News, 29(1), pp.4-7.

[115]. Yang, W., Liu, X., Zhang, L. and Yang, L., 2013. Big Data Real-Time Processing Based on Storm. 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications,.

[116]. Zaharia, M., Das, T., Li, H., Shenker, S. and Stoica, I., 2020. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters.

[117]. Zaharia, M., Xin, R., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M., Ghodsi, A., Gonzalez, J., Shenker, S. and Stoica, I., 2016. Apache Spark. Communications of the ACM, 59(11), pp.56-65.

[118]. Zaharia, M., Xin, R., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M., Ghodsi, A., Gonzalez, J., Shenker, S. and Stoica, I., 2016. Apache Spark : A unified engine for big data processing. Communications of the ACM, 59(11), pp.56-65.

[119]. Zahid, A., Masood, R. and Shibli, M., 2014. Security of sharded NoSQL databases: A comparative analysis. 2014 Conference on Information Assurance and Cyber Security (CIACS),.

[120]. Zeng, X., Garg, S., Barika, M., Zomaya, A., Wang, L., Villari, M., Chen, D. and Ranjan, R., 2020. SLA Management for Big Data Analytical Applications in Clouds. ACM Computing Surveys, 53(3), pp.1-40.

[121]. Zhang, L., Han, T. and Ansari, N., 2016. Revenue Driven Virtual Machine Management in Green Datacenter Networks Towards Big Data. 2016 IEEE Global Communications Conference (GLOBECOM),.

[122]. Zhao, G., Huang, W., Liang, S. and Tang, Y., 2013. Modeling MongoDB with Relational Model. 2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies,.

[123]. Zhao, Y., Calheiros, R., Bailey, J. and Sinnott, R., 2016. SLA-based profit optimization for resource management of big data analytics-as-a-service platforms in cloud computing environments. 2016 IEEE International Conference on Big Data (Big Data),.

[124]. Zhu, C., Zhou, H., Leung, V., Wang, K., Zhang, Y. and Yang, L., 2017. Toward Big Data in Green City. IEEE Communications Magazine, 55(11), pp.14-18.

Zoznam skratiek

ANSA	Automated Network Simulation and Analysis
API	Application Programming Interface
AWS	Amazon Web Services
BFD	Automated Network Simulation and Analysis
BLOB	Binary Large Object
CDRM	Cross Domain Resource Manager
Cisco IOS	Cisco Internetwork Operating System
CSV	Comma-separated values
DAX	DynamoDB Accelerator
DARE	Adaptive Data replication
DBMS	Database management system
DC	Direct Connect
DCM	Data Cached Module
DCR	DDL Command Replication
DEM	Data Elastic Module
DDL	Data Definition Language
DDoS	Denial Of Service
DEC	Digital Equipment Corporation
DEC-ACMS	Application Control Management System
DMS	Document Management System
EBS	Elastic Block Store
EC2	Elastic Cloud Compute
ERMS	Elastic Replication Management
FCC	Federal Communications Commision
HDFS	Hadoop Distributed File System
ILP	Integer Linear Program
IOT	Internet of things
IOUG	Independent Oracle Users Group
IP	Internet Protocol
IRS	Index Recommendation System
IT	Information Technology
JSON	Javascript Object Notation
JSP	Java Server Pages
KCL	Kinesis Client Library
KDS	Kinesis Data Streams
MB	Megabyte
MVNO	Mobile Virtual Network Operators
NWS	Network Weather Service
ORCS	Organizational Reliability Capability Assessment
PWRO	Primary Wireless Resource-Owners
QoS	Quality-of-Service
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RFID	Radio Frequency Identification
S3	Simple Storage Service

SLA	Service Level Agreement
SNS	Simple Notified Service
SRBD	System riadenia bázy dát (Database management system)
VPC	Virtual Private Cloud
VWN	Virtual Wireless Network
YARN	Yet Another Resource Negotiator