Masarykova univerzita
Fakulta Informatiky

In-Memory Data Management - porovnání s tradičními relačními databázemi

Diplomová práca

Bc. Pavel Démuth

Brno, 2016

Prehlásenie

Prehlasujem, že táto diplomová práca je mojim pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovávaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Bc. Pavel Démuth

Vedúci práce: RNDr. Pavel Hajn

Poďakovanie

Rád by som na tomto mieste poďakoval RNDr. Pavlovi Hajnovi za vedenie a inšpiráciu pri tvorbe diplomovej práce.

Zhrnutie

Diplomová práca sa zaoberá porovnaním in-memory relačných databáz s tradičnými relačnými databázami. Na začiatku práce je stručne vysvetlený doterajší spôsob použitia databáz v podnikových systémoch a aké zmeny umožnili zavedenie in-memory prístupu. Ďalej je v práci popísaná architektúra in-memory data managementu a základné operácie v tomto type databáz. Ako príklad som použil in-memory databázu SAP HANA. Praktická časť obsahuje stručne popísané testované databázy a aplikáciu použitú na testovanie aj s dátovým modelom. V poslednej kapitole sú predstavené jednotlivé výkonnostné testy a ich vyhodnotenie.

Kľúčové slová

in-memory data management, relačná databáza, SAP HANA, Oracle, výkonnostné testovanie, stĺpcové ukladanie dát, column layout

Obsah

Obsah . . . . . **v**

1 Úvod **1**

2 Základné pojmy **3**

3 Úvod do tématiky **4**
3.1 Podnikové systémy 4
3.2 Bežný prístup 4
3.2.1 OLTP - Online Transaction Processing 5
3.2.2 OLAP - Online Analytical Processing 5
3.2.3 Kombinácia OLTP a OLAP 5
3.3 Zmeny v hardware-i 6
3.3.1 Hierarchia pamätí 7

4 In-memory data management - architektúra **8**
4.1 Ukladanie dát v hlavnej pamäti 8
4.2 Data layout v hlavnej pamäti 9
4.2.1 Row Layout - Riadkové uloženie 9
4.2.2 Column layout - Stĺpcové uloženie 10
4.2.3 Hybrid Layout 11
4.3 Slovníkové kódovanie 12
4.4 Kompresia dát 15
4.4.1 Prefixové kódovanie 15
4.4.2 Run-Length kódovanie 16
4.5 Prehľad arhitektúry 17
4.5.1 Diferenčný buffer a merge 18
4.5.2 Logovanie a obnova 19

5 Základné operácie **21**
5.1 Zmazanie záznamu - Delete 21
5.2 Vloženie záznamu - Insert, Insert Only 21

5.2.1 Insert Only 22
5.3 Vybratie záznamu - Select 22
5.4 Materializované stratégie 24
5.5 Spojenie - Join 25
5.6 Agregačné funkcie 26

6 Testovacie prostredie 28
6.1 Databázové systémy 28
6.1.1 SAP HANA 28
6.1.2 Oracle 29
6.1.3 MaxDB 29
6.1.4 Sybase 30
6.2 Testovacia aplikácia 31
6.2.1 Programovací jazyk ABAP 31
6.2.2 RFC - Vzdialené volanie funkcií 32
6.2.3 Dátový model 33
6.2.4 Implementácia 34

7 Výsledky testovania 36
7.1 Veľkosť databázy 37
7.2 Insert - Vloženie záznamov 38
7.3 Delete - Vymazanie záznamov 40
7.4 Test 1 41
7.5 Test 2 42
7.6 Test 3 43
7.7 Test 4 44
7.8 Test 5 45
7.9 Test 6 47
7.10 Test 7 48
7.11 Test 8 50
7.12 Test 9 51
7.13 Test 10 52
7.14 Vyhodnotenie testov 53

8 Záver **55**

Literatúra **56**

9 Príloha **57**

9.1 Exekučné plány 57

9.1.1 Test 1 57

9.1.2 Test 2 58

9.1.3 Test 3 59

9.1.4 Test 4 61

9.1.5 Test 5 63

9.1.6 Test 6 65

9.1.7 Test 7 66

9.1.8 Test 8 68

9.1.9 Test 9 69

9.1.10 Test 10 71

1 Úvod

Žijeme v dobe, keď žiadna spoločnosť, žiadny prosperujúci biznis nedokáže fungovať bez kvalitného databázového systému. Neuvedomujeme si to, ale nejaký druh databázy je súčasťou skoro každej našej činnosti. Najzákladnejší druh databázy, s ktorým sa stretol každý z nás je napríklad telefónny zoznam v mobile, či už je zoznam fyzicky uložený v mobile alebo ho získavame online z cloudu. Každý náš nákup v obchode je zaznamenaný v databáze daného obchodu. Niektoré spoločnosti, ako je napríklad Google, sledujú každý náš pohyb na internete a vďaka analýze týchto dát sú nám na internete ponúkané cielené reklamy. Určite ste už aj Vy zažili, že ste hľadali nejaký tovar na internete a následne potom na každej internetovej stránke sa Vám zobrazovali reklamy ohľadom toho alebo podobného tovaru. Ak máte internet aj v mobile, tieto spoločnosti analyzujú aj dáta z GPS senzora mobilu. Google to napríklad používa na analýzu premávky na cestách a pomocou týchto informácií dokáže navigačná aplikácia Google Maps vypočítať najlepšiu a najrýchlejšiu cestu k cieľu. V posledných rokoch aj mnoho podnikových spoločností používa pokrokovú analýzu dát napríklad na sledovanie predajnosti tovaru. Ktorý tovar a kde sa najviac predáva a hlavne komu. Aby mohli podľa toho cielene oslovovať svojich zákazníkov a zvýšiť svoj obrat.

Nič z toho by nebolo možné bez moderných databázových systémov a rýchlej analýze dát. Práve na analýzu dát sú kladené stále väčšie nároky. Veľkosť analyzovaných dát sa stále zvyšuje. Ale najväčší dôraz sa kladie práve na rýchlosť analýzy. Terajšie relačné databázové systémy, ktoré používa väčšina veľkých podnikových spoločností, nestačia stále rastúcim nárokom na rýchlosť. Práve preto veľa spoločností zameraných na informačné systémy začalo rozmýšľať nad novými spôsobmi ukladania dát, ktoré by rýchlosť spracovania zvýšili. Zlepšujúce sa vlastnosti hardwarových komponent počítačov, najmä pamätí a procesorov, umožnili zmeniť miesto ukladania dát v databáze. Z ukladania na disk sa prešlo na ukladanie do hlavnej pamäte počítača a tým sa výrazne urýchlil proces spracovania informácií a ich analýzy. Vznikli takzvané in-memory databázy.

Úlohou mojej diplomovej práce je vysvetliť základné princípy In-memory databázových systémov a porovnať in-memory sytém SAP HANA s inými tradičnými

relačnými databázovými systémami. V kapitole č.3 stručne vysvetlím spôsob, aký sa doteraz používal v podnikových systémoch na ukladanie dát a na ich rýchlu analýzu. Ďalej v tejto kapitole popíšem aké zmeny nastali v hardware-i počítača, že bolo možné uvažovať o pamäti ako dátovom úložisku databázy. V kapitole č.4 popíšem architektúru in-memory databáz a technologické zmeny, ktoré museli nastať, aby mohla táto architektúra fungovať. Lebo nestačí len zmeniť miesto ukladania, ale musí sa zmeniť aj celkový spôsob, ako sa tieto dáta do pamäte ukladajú a ako sa s nimi pracuje. V nasledujúcej kapitole č.5 popíšem základné operácie v tomto type databáz a aké zmeny v nich nastali kvôli in-memory prístupu. V praktickej časti diplomovej práce predstavím testovacie prostredie, ktoré som použil na výkonnostné testovanie. Stručne predstavím databázy, nad ktorými som vykonával výkonnostné testy, testovaciu aplikáciu a programovací jazyk, ktorý som použil na jej vytvorenie. Nakoniec vysvetlím dátový model použitý pri testovaní. V poslednej kapitole opíšem jednotlivé výkonnostné testy a zhodnotím ich výsledky pomocou grafov rýchlostí jednotlivých databáz.

2 Základné pojmy

V texte práce sa čitateľ môže stretnúť s pojmami, ktoré sú špecifické pre oblasť, v ktorej bola táto práca vytvorená. Preto na začiatku uvádzam základné pojmy so stručným vysvetlením.
Štrukturované dáta - dáta vo formáte ľahko spracovateľnom počítačom. Väčšinou uložené v tabuľkách, poliach a štruktúrach.
Neštrukturované dáta - dáta, ktoré nie je ľahké analyzovať. Napríklad hudba, video, fotky a podobne.
Multidimensionálna kocka - poskytuje viacrozmerný pohľad na uložené dáta.
Konceptuálny pohľad - idealizovaný pohľad na niečo, ako funguje.
Data layout - spôsob usporiadania dát.
Agregačné funkcie - vstavané príkazy v SQL, ktoré vykonávajú nad stĺpcami matematické operácie.
Kardinalita hodnôt - počet odlišných hodnôt.
Default hodnota - štandardná, východzia hodnota.
Entropia stĺpcov - miera, ktorá vyjadruje koľko informácií obsahuje daný stĺpec.
Metadata - štrukturovaným spôsobom popisujú zložitejšie neštrukturované dáta.
Diferenčný buffer - technológia použitá v in-memory databázach. Slúži ako vyrovnávacia pamäť na rýchlejšie vkladanie záznamov.
Log - záznam informácií o činnosti a behu databázy. Slúži k spätnej analýze ak nastal chybový stav.
Snapshot - stav systému v určitom časovom okamihu.
Merge - operácia zlúčenia dvoch alebo viacerých množín.
Join operácia - slúži na spájanie relácií, tabuliek v databáze.
Sort - zatriedenie množiny, napríklad podľa abecedy.
Predikát - výraz, o ktorom je možné rozhodnúť či je pravdivý alebo nepravdivý.
Cluster - najmenšia logická jednotka, do ktorej sa ukladajú dáta na disku.
Runtime systém - viacero programov určených na podporu realizácie počítačových programov, napísaných v ľubovoľnom programovacom jazyku.
Char - dátový typ určený na ukladanie znakov. Väčšinou doplnený o dĺžku reťazca.
Integer - dátový typ určený na ukladanie celých čísiel.

3 Úvod do tématiky

Problematika databázových systémov a podnikových aplikácií v posledných rokoch zažíva veľké zmeny vo vnímaní dát a ich ukladania. Spoločnosti potrebujú uchovávať stále viac a viac dát a vykonávať nad nimi rýchle analytické operácie. Do databáz sa neukladajú už len štrukturované dáta, ale do popredia sa dostávajú aj neštrukturované dáta. Neštrukturované dáta sa väčšinou získavajú z internetu, sociálnych sietí, podporných systémov a podobne. Podnik musí byť schopný v rýchlosti analyzovať tieto dáta, prichádzajúce z rôznych zdrojov.

V tejto kapitole popíšem ako to bežne funguje v podnikových systémoch a čo in-memory prístup umožňuje zmeniť. Ďalej popíšem vďaka čomu je možné prejsť z ukladania dát na disk na ukladanie dát do pamäti.

3.1 Podnikové systémy

Podnikové systémy slúžia pre spoločnosti ako komplexné riešenie ich podnikových procesov. Podnikové systémy zohrávajú kľúčovú úlohu vo viacerých častiach spoločnosti - účtovníctvo, plánovanie, personalistika, logistika a podobne. Zákazníci si môžu vybrať z jednotlivých častí a upraviť si ich podľa seba.

3.2 Bežný prístup

Doteraz sa v podnikových systémoch na zrýchlenie transakcií a analytických operácií používal prístup dvoch databáz (OLTP a OLAP). Prvá databáza je optimalizovaná na vkladanie, upravovanie záznamov a obsluhovanie viacerých transakcií naraz. Dáta v tejto databáze sú vždy aktuálne. Druhá databáza je vytvorená kopírovaním z prvej. Je optimalizovaná na analytické procesy. To znamená, že analytické dáta sú dopredu predpočítané a uložené do komplexných tabuliek, z ktorých sa potom pomocou aplikácií znovu získavajú. Dáta v tejto databáze nie sú väčšinou aktuálne.

3.2.1 OLTP - Online Transaction Processing

OLTP databáza je optimalizovaná na veľké množstvo malých transakcií. Dáta sú uložené v jednoduchých originálnych tabuľkách. Hlavný dôraz sa kladie na udržanie integrity dát a veľmi rýchle spracovanie dotazov. Je typicky používaná na vkladanie a úpravu záznamov, operácie nad celými riadkami a na jednoduché dotazy s malými výstupmi.[3]

3.2.2 OLAP - Online Analytical Processing

Systémy typu OLAP sú prispôsobené na analytické spracovanie veľkého množstva dát. Dáta do systému prichádzajú ako preddefinovaná podmnožina transakčných dát. Dáta sa kopírujú v pravidelných intervaloch, čo spôsobuje zastaralosť informácií v niektorých komplexných dotazoch. Aby analytické procesy boli čo najrýchlejšie, dáta sú často ukladané redundantne, čo je jednou z veľkých nevýhod OLAP systémov. Najčastejšie použitie je na plánovanie, predpovedanie a reporty. Na OLAP systémy sa dá pozerať aj ako na multidimensionálnu kocku.[2]

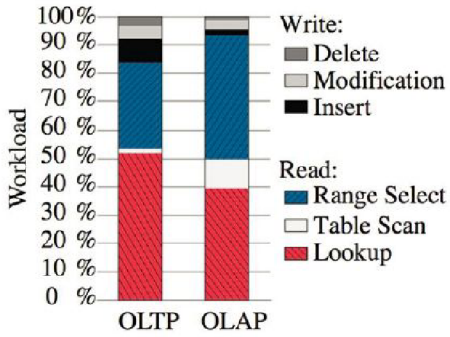
3.2.3 Kombinácia OLTP a OLAP

Dnešné systémy sú optimalizované buď na analytické alebo transakčné zaťaženie. Profesor Hasso Plattner so svojím tímom skúmali záťaž viacero systémov, zákazníckych systémov s množstvom dát. Ich cieľom bolo zistiť aké príkazy sa najčastejšie používajú pri OLTP systémoch a aké pri OLAP systémoch. Na obrázku 1 je možné vidieť výsledok ich výskumu.[1]

Z obrázka je jasné, že využitie oboch systémov je veľmi podobné. Tým pádom ich rozdelenie nedáva veľký zmysel. Jedným z dôvodov rozdelenia do dvoch systémov bolo, aby analytické príkazy boli vykonané čo najrýchlejšie. Kvôli tomu sa používal OLAP systém, kde bola väčšina dát už predpočítaná. Ale v dnešnej dobe rýchlych počítačov a možnosti umiestniť dáta iba do pamäti, je možné si analytické dáta vypočítavať v reálnom čase.

Hlavnou výhodou spojenia OLTP a OLAP systému je bezpochyby možnosť vykonávať transakčné a analytické príkazy nad jednou databázou, jednou množinou dát. Tým pádom odpadá zbytočná réžia kopírovania dát z jedného systému

do druhého. Redundancia dát je takmer nulová. Taktiež analytické procesy budú vykonané nad aktuálnymi dátami.



Obr. 1: Štatistika príkazov na OLTP a OLAP systémoch [1]

3.3 Zmeny v hardware-i

Ako som spomenul vyššie, hlavnou príčinou uvažovania nad pamäťou ako hlavným dátovým úložiskom v databáze boli neustále zmeny v hardware-i: Masívne paralelné systémy s narastajúcim počtom procesorov a ich jadier a zvyšujúce sa veľkosti hlavných pamätí.

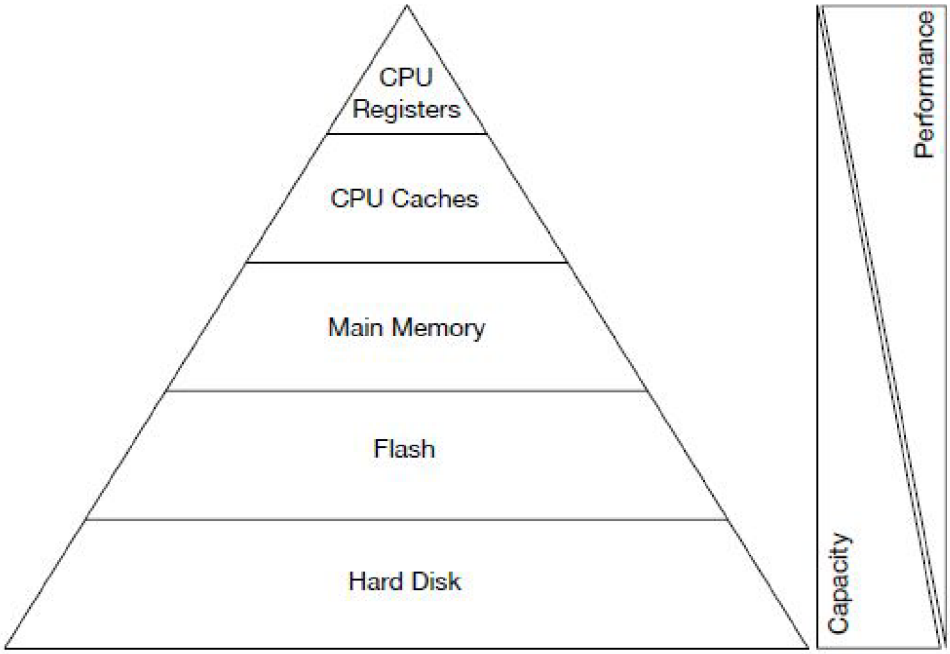
Na začiatku roku 2000 boli zavedené viacjadrové procesorové architektúry, čo sa odzrkadlilo aj na začínajúcom trende k rastúcemu paralelizmu. Dnes typická matičná doska pre podnikové výpočty sa skladá z ôsmich procesorov a z desiatich až šestnástich jadier na CPU. Celkovo 80 až 128 jadier na server. Adresný priestor o veľkosti 64 bitov zabezpečuje, že čoraz väčšie množstvo hlavnej pamäte môže byť adresované, s dnešnými servermi až do 6 TB. Maximálna priepustnosť medzi CPU a DRAM je približne 85 GB/s a je možné prenášať až 6 GB dát za sekundu medzi servermi.

Napriek predstaveniu masívneho paralelizmu ešte donedávna bolo všetko stále zamerané na disky a ich výkonnostnú optimalizáciu. Disky boli príliš pomalé, ale dôležité na ukladanie dát. V porovnaní s rýchlosťou vývoja procesorov bol vývoj diskov veľmi pomalý. To malo za následok celkovú zmenu pohľadu na prácu s databázami a veľkým množstvom dát. Dnes rastúca veľkosť hlavnej pamäti v serveroch a ich stále sa znižujúca cena započali posun z diskovo orientovaných systémov na

pamäťovo orientované databázové systémy. Tieto systémy si primárnu kópiu dát držia v hlavnej pamäti a disk používajú iba na zálohu, obnovu a archiváciu.[1]

3.3.1 Hierarchia pamätí

Na hierarchiu pamätí sa môžeme pozerať ako na pyramídu úložných médií. Čím je médium pomalšie, tým je lacnejšie. Z toho aj vyplýva, že kapacita na pomalšom médiu rýchlejšie rastie, vďaka jeho nízkej cene.



Obr. 2: Konceptuálny pohľad na hierarchiu pamätí [1]

Najlacnejšie médium a zároveň s najväčšou kapacitou je disk. Po ňom nasleduje Flash, ktorý je podstatne rýchlejší ako disk, ale stále používa rovnaké softwarové princípy ako disk. Na ďalšom stupni pyramídy sa nachádza hlavná pamäť, ktorá je priamo prístupná. Táto pamäť prešla za posledné roky najväčším vývojom a teraz dokáže ponúknuť viac dátového priestoru. Na predposlednej vrstve sa nachádzajú CPU cache (vyrovnávacie pamäte) L1, L2 a L3. CPU registre, v ktorých musia byť umiestnené dáta, ak ich chceme použiť v kalkuláciách, sa nachádzajú na vrchu pyramídy. Keďže každá operácia prebieha v CPU a dáta musia byť v CPU

registroch, tak sa na transport dát z disku väčšinou používajú iba štyri vrstvy.

Najväčším spomalením pri prístupe k dátam je aj podľa obrázku disk. Pre porovnanie je uvedená doba prístupu k trom dôležitým vrstvám pamäťovej hierarchie:

Prístup k L1 cache 0.5 nanosekundy
Prístup k hlavnej pamäti 100 nanosekúnd
Prístup na disk 10 000 000 nanosekúnd

Na pamäťovom prístupe v databázových systémoch nie je nič nezvyčajné. Všetky výpočty boli vždy v pamäti, pretože skutočné výpočty môžu prebiehať iba v CPU. Výkon aplikácie využívajúcej veľké množstvo dát je väčšinou určovaný rýchlosťou, akou sa tieto dáta prenesú cez pamäťovú hierarchiu až do CPU registrov. Pri predpoklade, že primárna kópia dát je čo najbližšie k CPU, sa odozva znižuje a tým sa zvyšuje výkon celého systému. Najjednoduchšia operácia, ktorú vie CPU vykonať je porovnanie. Používa sa napríklad pri filtrácii atribútu v databázovej tabuľke. Pri predpokladanej rýchlosti skenovania 4 MB/ms pre túto operáciu na jednom jadre procesoru, je teoretická rýchlosť na 15 jadrovom procesore 60 GB/s. Na typickom 8 procesorovom uzle je táto rýchlosť 480 GB/s. Vzhľadom na dostupnosť veľkých multi-uzlových systémov, ktoré majú 10 uzlov a 120 jadier s dátami rovnomerne rozdelenými medzi jednotlivé uzly je veľmi obtiažne nájsť príklad v podnikových systémoch, kde by výpočet trval dlhšie ako sekundu.[1]

4 In-memory data management - architektúra

4.1 Ukladanie dát v hlavnej pamäti

Vo väčšine dnešných databázových systémoch sa na ukladanie dát primárne používa pevný disk. Ako som spomenul v podkapitole 3.3.1, rýchlosť prístupu k diskom sa veľmi nemení a tak sa začalo rozmýšľať nad iným miestom na ukladanie dát. Pri in-memory databázach sa ako permanentné úložisko používa hlavná pamäť. Ale na logovanie a obnovu stále potrebujeme energeticky nezávislú pamäť - pevný disk.

Použitie hlavnej pamäte ako primárne úložisko vedie k rozdielnemu usporiadaniu dát.

4.2 Data layout v hlavnej pamäti

Obyčajný človek vidí dáta v databáze ako dvoj-dimenzionálne pole, tabuľku, ktorá má niekoľko stĺpcov (atribútov) a riadkov (záznamov). Lenže pamäť má v dnešných počítačoch stále iba lineárny adresný priestor. Tým pádom je potrebné dvoj-dimenzionálne dáta dostať do lineárnej podoby. V tejto podkapitole predstavím tri spôsoby, akými je možné reprezentovať tabuľky v pamäti.

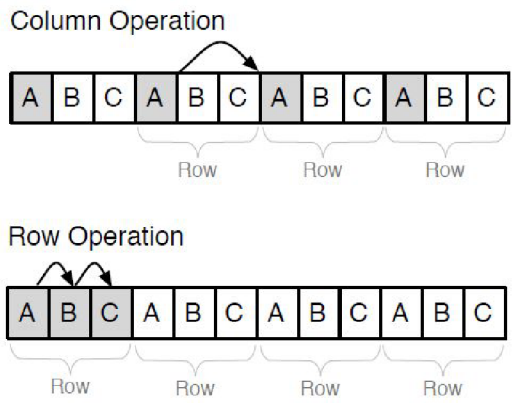
Pre jednoduchšie vysvetlenie jednotlivých spôsobov si predstavme nasledujúcu tabuľku troch osôb a pokúsime sa ju previesť do lineárneho zápisu.

ID	**Meno**	**Krajina**	**Mesto**
1	Pavel	Austrália	Sydney
2	Jozef	USA	Washington
3	Daniela	Nemecko	Berlín

Tabuľka 1: Jednoduchý príklad tabuľky osôb

4.2.1 Row Layout - Riadkové uloženie

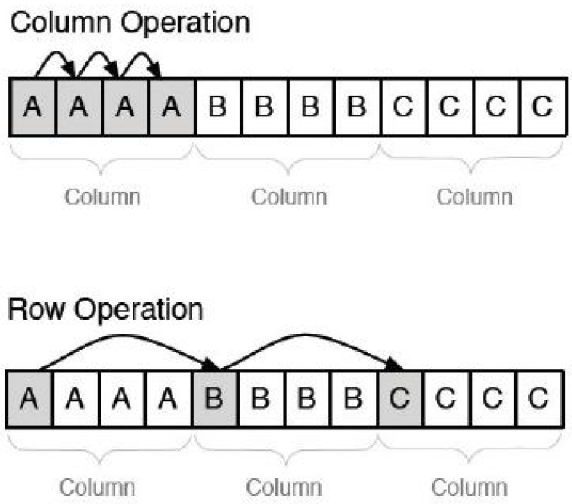
Klasický a očividný spôsob transformovania dvoj-dimenzionálnej tabuľky do lineárnej podoby je riadkový alebo záznamový prístup. V tomto prípade sú všetky atribúty záznamu uložené v pamäti sekvenčne. Inými slovami, dáta sú uložené podľa záznamov. V prípade nášho príkladu by to vyzeralo nasledovne: **"1, Pavel, Austrália, Sydney; 2, Jozef, USA, Washington; 3, Daniela, Nemecko, Berlín"**. Pri tomto prístupe je rekonštrukcia záznamov veľmi rýchla, ale pomalšie je sekvenčné skenovanie jedného atribútu. Takisto vkladanie nových záznamov je pri riadkovom usporiadaní rýchlejšie a jednoduchšie. Na Obrázku 3 môžete vidieť ako je k dátam pristupované pri riadkových a pri stĺpcových operáciach. [4]



Obr. 3: Prístup k dátam pri riadkovom usporiadaní [1]

4.2.2 Column layout - Stĺpcové uloženie

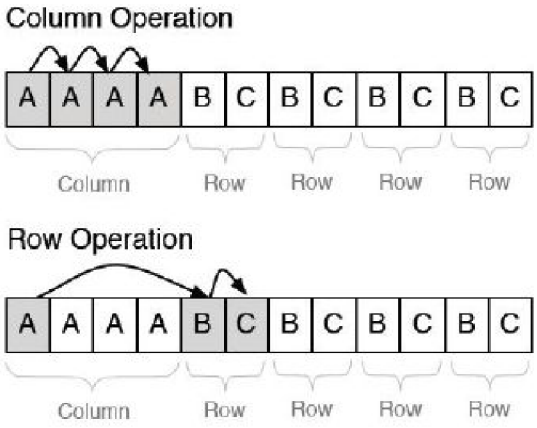
Pri použití stĺpcového prístupu sú hodnoty jedného stĺpca uložené spolu, stĺpec po stĺpci. V našom príklade by dáta boli uložené nasledovne: **"1, 2, 3; Pavel, Jozef, Daniela; Austrália, USA, Nemecko; Sydney, Washington, Berlín"**. Stĺpcový prístup je efektívny najmä pri operáciách, ktoré pracujú na viacerých riadkoch, ale len na značne menšom počte stĺpcov. Je to vďaka tomu, že hodnoty jednotlivých stĺpcov sa čítajú sekvenčne. Využíva sa to napríklad pri vykonávaní agregačných výpočtov. Avšak stĺpcový prístup je nevýhodný pri vykonávaní operácií na jednotlivých záznamoch alebo pri vkladaní nových riadkov. Existuje niekoľko výhod v prospech stĺpcového prístupu pri použití v podnikových systémoch. Napríklad nám dovoľuje použiť rôzne kompresné techniky, ktoré v prípade riadkového prístupu nie sú dostatočne výhodné a jednoduché. Ďalej umožňuje veľmi rýchle skenovanie stĺpcov, čo otvára možnosti pre počítanie agregátov v reálnom čase, tým pádom si ich nemusíme predpočítavať dopredu a zaberať nadbytočné miesto v pamäti. Na Obrázku 4 môžete vidieť ako je k dátam pristupované pri riadkových a pri stĺpcových operáciach. [4]



Obr. 4: Prístup k dátam pri stĺpcovom usporiadaní [1]

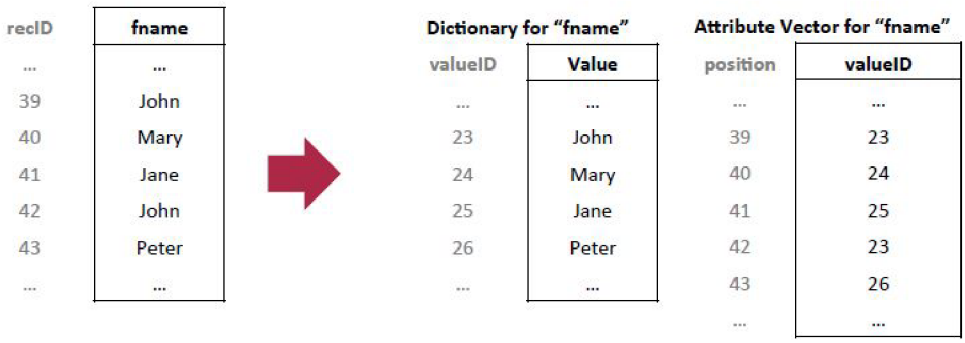
4.2.3 Hybrid Layout

Každý systém má špecifické požiadavky na dáta. Vzhľadom na tieto požiadavky jednému systému môže vyhovovať riadkový prístup a inému zase stĺpcový prístup. Hybridný prístup kombinuje výhody obidvoch predchádzajúcich. Umožňuje ukladať niektoré atribúty stĺpcovým prístupom, zatiaľ čo iné spoločne ukladá riadkovým prístupom. Optimálna kombinácia záleží hlavne na požiadavkách jednotlivých systémov a ich záťažiach. Pre jej výpočet sa používajú rôzne algoritmy. Predstavme si atribúty, ktoré v komerčných aplikáciách neodmysliteľne patria k sebe, hodnota a jej jednotka, suma a príslušná mena a podobne. Hlavnou myšlienkou hybridného prístupu je, aby skupina atribútov, ktorá je procesovaná spoločne bola z hľadiska rýchlosti aj fyzicky uložená spoločne. Predstavme si pomocou príkladu tabuľky osôb, keď atribúty ”ID” a ”Meno” sú často procesované spoločne. Tým pádom nám vychádza nasledujúci návrch: **"1, Pavel; 2, Jozef; 3, Daniela; Austrália, USA, Nemecko; Sydney, Washington, Berlín"**. V tomto prípade sú ”ID” a ”Meno” uložené riadkovým spôsobom a ”Krajina”, ”Mesto” stĺpcovým spôsobom. Na Obrázku 5 môžete vidieť ako je k dátam pristupované pri riadkových a pri stĺpcových operáciach.[1]



Obr. 5: Prístup k dátam pri hybridnom usporiadaní [1]

4.3 Slovníkové kódovanie



Obr. 6: Príklad slovníkového kódovania [1]

Vzhľadom na to, že dáta ukladáme do hlavnej pamäte, musíme sa zamyslieť nad ich veľkosťou a ako ich chceme procesovať. Dôležitým faktorom je aj čo najmenší počet prístupov do pamäte. Toto môžeme dosiahnuť buď prístupom k menšiemu počtu stĺpcov dotazovaním len potrebnými atribútmi. Na druhej strane zmenšením počtu bitov na reprezentáciu dát v pamäti, môžeme zredukovať spotrebu pamäte a zároveň aj dobu prenosu.

Analýzou podnikových dát bolo zistené, že väčšina atribútov tabuľky sa nepoužíva a pritom tabuľky majú príliš veľa atribútov. 55% stĺpcov sa priemerne

nepoužíva a existujú tabuľky, ktoré majú až 100 stĺpcov. Väčšina stĺpcov má malú kardinalitu hodnôt. Navyše v niektorých stĺpcoch prevažuje hodnota NULL alebo iné default hodnoty, takže entropia týchto stĺpcov je veľmi malá. Vďaka týmto zisteniam je možné na tieto typy tabuliek použiť **slovníkové kódovanie**. Hlavným efektom slovníkového kódovania je, že dlhé hodnoty, ako sú napríklad texty, sú reprezentované ako malé číselné hodnoty. Princíp slovníkového kódovania spočíva v rozdelení každého stĺpca na **slovník** a **atribút vektor**. V slovníku sú uložené všetky **rozdielne hodnoty** a ich **"valueID"**. Atribút vektor obsahuje iba **"valueID"**, ktoré odpovedá tomu v slovníku. Týmto spôsobom sme schopní ukladať všetky informácie vo forme Integeru namiesto iných väčších dátových typov. Na obrázku 6 môžete vidieť príklad slovníkového kódovania na stĺpci krstných mien. Pre ešte lepší výkon je potrebné dáta v slovníku zoradiť. Vďaka tomu môžeme pre vyhľadanie hodnôt použiť rýchle binárne vyhľadávanie, čo nás ale donúti zmeniť spôsob vkladania nových záznamov, čo vysvetlím v neskoršej kapitole.[1]

Kvôli lepšiemu pochopeniu tejto techniky, na ktorej stojí princíp in-memory databáz, použijem príklad tabuľky svetovej populácie s 8 miliardami záznamami. Bližšie vysvetlím kompresiu pre jeden stĺpec z obrázka 7

Zoberme si napríklad stĺpec reprezentujúci názvy krajín. Pre jednoduchosť rátam s tým, že vo svete existuje 200 krajín. Na uloženie jednej krajiny potrebujeme 47 Bajtov.

Bez použitia slovníkového kódovania by sme na uloženie stĺpca, určeného pre krajiny, obsahujúceho 8 miliard záznamov potrebovali 376 GB.

$$8miliárd \times 47B \approx 376GB$$

Na zakódovanie 200 krajín vo forme Integeru nám postačí 8 bitov.

$$log_2(200) = 8bitov$$

Na vytvorenie atribút vektoru z 8 miliard záznamov nám pri slovníkovom kódovaní postačí 8GB.

$$8miliárd \times 8bit \approx 8GB$$

A na vytvorenie slovníka, do ktorého vložíme našich 200 krajín, pričom každá krajina má maximálnu veľkosť 47 Bajtov, potrebujeme 9,4 kB

$$47B \times 200 \approx 9,4kB$$

To znamená, že po použití slovníkového kódovania sa veľkosť stĺpca v pamäti zredukuje z 376GB na niečo vyše 8GB. Kompresný faktor bude nasledujúci

$$\frac{veľkosť\ bez\ kompresie}{kompresovaná\ veľkosť} = \frac{376GB}{8GB + 9,4kB} \approx 47$$

Column	Cardi-nality	Bits Needed	Item Size	Plain Size	Size with Dictionary (Dictionary + Column)	Compression Factor
First names	5 millions	23 bit	50 Byte	400GB	250MB + 23GB	≈ 17
Last names	8 millions	23 bit	50 Byte	400GB	400MB + 23GB	≈ 17
Gender	2	1 bit	1 Byte	8GB	2b + 1GB	≈ 8
City	1 million	20 bit	50 Byte	400GB	50MB + 20GB	≈ 20
Country	200	8 bit	47 Byte	376GB	9.4kB + 8GB	≈ 47
Birthday	40000	16 bit	2 Byte	16GB	80kB + 16GB	≈ 1
Totals			**200 Byte**	**≈ 1.6TB**	**≈ 92GB**	**≈ 17**

Obr. 7: Ukážka kompresie slovníkového kódovania na tabuľke svetovej populácie [1]

Na základe obrázku 7 môžeme vidieť, že tabuľka mala na začiatku veľkosť **1,6TB** a po aplikácií slovníkového kódovania sa jej veľkosť zredukovala na **92GB**. Vďaka takejto razantnej redukcii je oveľa jednoduchšie uložiť dáta do hlavnej pamäte. Kompresný faktor závisí na veľkosti iniciálneho dátového typu a entropie stĺpca. Tá sa vypočíta pomocou kardinality stĺpca a kardinality celej tabuľky.

4.4 Kompresia dát

Napriek použitiu slovníkového kódovania, tabuľky v podnikových systémoch stále môžu narásť do veľkých rozmerov. Kvôli tomu sa na tabuľky so slovníkovým kódovaním používajú ešte ďalšie kompresné techniky, aby sa znížili požiadavky na hlavnú pamäť. Stĺpcové ukladanie dát je výhodné pre použitie kompresných techník, pretože dáta rovnakého typu sú uložené za sebou a ich spracovanie je efektívne.

Ďalšou výhodou je zmenšenie veľkosti dát, ktoré je potrebné preniesť medzi CPU a hlavnou pamäťou, čo má za následok zrýchlenie vykonávania dotazov.

V tejto podkapitole predstavím dve najznámejšie kompresné techniky, ktoré ponúkajú prijateľný kompromis medzi kompresným faktorom a zložitosťou kódovania a dekódovania. [1].

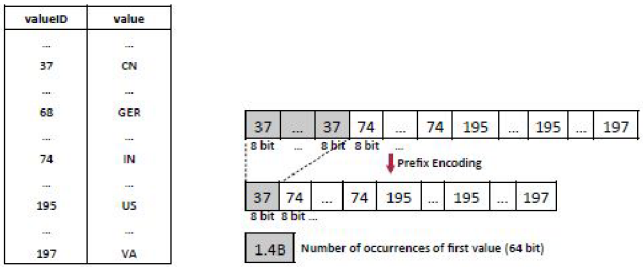
4.4.1 Prefixové kódovanie

V databázach často nastane situácia, že v stĺpci existuje veľa rovnakých hodnôt a menej rozdielnych. To by malo za následok, že jednu rovnakú hodnotu by sme mali v databáze uloženú niekoľkokrát v nekomprimovanej podobe. Tento problém jednoduchým spôsobom rieši prefixové kódovanie. Aby sme mohli aplikovať toto kódovanie, dáta musia byť zoradené podľa prevládajúcej hodnoty a atribút vektor musí začínať touto hodnotou. Pre úspešnú kompresiu stĺpca sa neuloží každý výskyt prevládajúcej hodnoty, ale iba jej počet a jedna jej inštancia. To znamená, že prefixovo zakódovaný atribút vektor bude obsahovať:

- počet výskytov prevládajúcej hodnoty
- valueID prevládajúcej hodnoty
- valueID zvyšných hodnôt

Príklad

Predstavme si znovu tabuľku svetovej populácie, ktorú zoradíme podľa krajiny. To znamená, že keďže na svete je najviac obyvateľov Číny (približne 1.4 miliardy), tak na začiatku stĺpca krajín sa bude nachádzať 1.4 miliard-krát Čína. Názorne to môžeme vidieť na obrázku 8



Obr. 8: Príklad prefixového kódovania [1]

Pred aplikáciou prefixového kódovania bolo potrebných na uloženie atribút vektoru:

$$8miliárd \times 8b = 7.45GB$$

Ak je stĺpec krajín prefixovo zakódovaný, tak namiesto uloženia 1.4miliard-krát 8bitov, nám stačí uložiť iba raz 8 bitov. Navyše potrebujeme 64 bitov dlhý Integer na uloženie počtu výskytu všetkých hodnôt. Celkovo budeme potrebovať:

$$(8miliárd - 1.4miliárd) \times 8b + 64b + 8b = 6,15GB$$

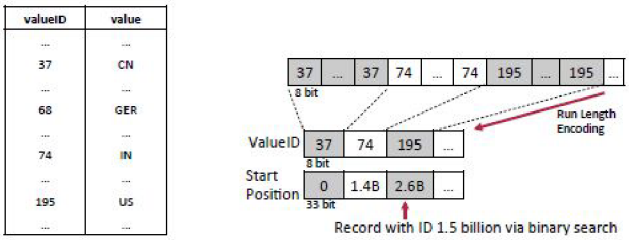
V našom príklade sme ušetrili 1.3GB po použití prefixového kódovania.[1]

4.4.2 Run-Length kódovanie

V predchádzajúcej kapitole som predstavil prefixové kódovanie. Aj keď týmto kódovaním sa evidentne zmenšilo potrebné miesto na uloženie dát, v tabuľke sa stále nachádzajú redundantné informácie pre zvyšné krajiny. Pre vyriešenie tohto problému bolo predstavené run-length kódovanie.

Pre maximálnu kompresiu je najlepšie ak je stĺpec zatriedený, aby výskyty rovnakých hodnôt boli pri sebe. Následne sa sekvencie rovnakých hodnôt vymenia za jednu inštanciu tejto hodnoty a to:

- buď počtom výskytov
- alebo offsetom štartovacej pozície.



Obr. 9: Príklad run-length kódovania [1]

Aplikovaním na náš príklad stĺpca krajín, namiesto uloženia dát o veľkosti 7.45GB, uložíme dva vektory:

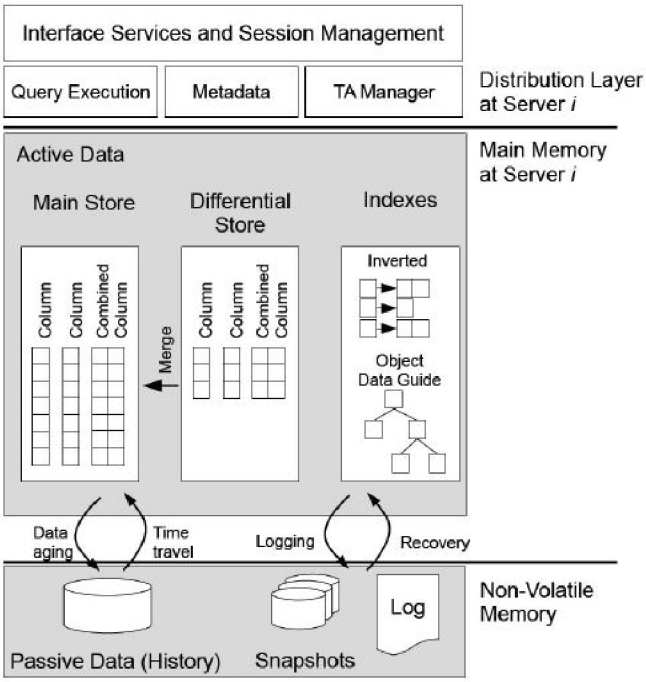
- jeden obsahujúci všetky rozdielne hodnoty: $200 \times 8bit$
- druhý s offsetmi: $200 \times 33bit$, 33bitov je potrebných na uloženie jednotlivých offsetov $(log_2(8miliárd) = 33b$.

Veľkosť výsledného atribút vektora môže byť zredukovaná až na 1kB.[1]

$$200 \times (33b + 8b) + 33b \approx 1kB$$

4.5 Prehľad arhitektúry

Prototyp in-memory databázy, na ktorom je postavená SAP HANA, používa architektúru zobrazenú na obrázku 10. Distribučná vrstva poskytuje komunikáciu s aplikáciami, metadata a vypočítava exekučné plány dotazov. Hlavné úložisko, Diferenčný buffer a Indexy sa nachádzajú v hlavnej pamäti. Posledná vrstva obsahuje Logy, Snapshoty a pasívne dáta a je uložená na energeticky nezávislej pamäti.



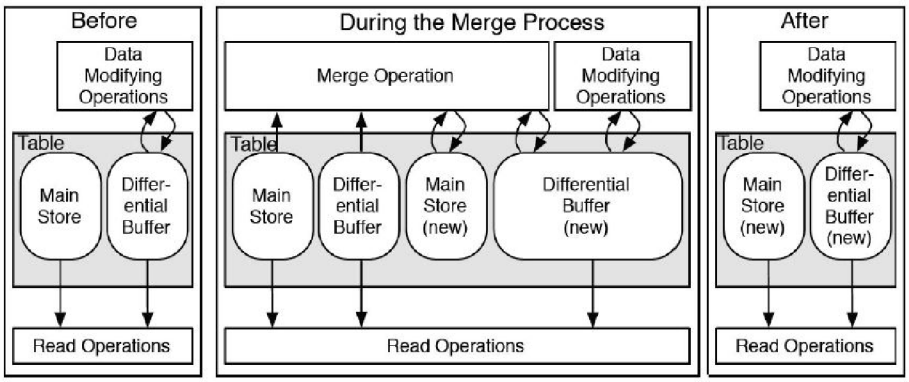
Obr. 10: Architektúra prototypu in-memory databázy [1]

4.5.1 Diferenčný buffer a merge

Keď vkladáme nový záznam, nejde do hlavného úložiska, ale do Diferenčného buffera. V tomto buffery sú dáta taktiež uložené v stĺpcovom formáte, ale jeho slovník je niekoľkokrát menší ako slovník hlavného úložiska. Navyše nie je potrebné ho držať v utriedenom stave. Vďaka tomu je vkladanie do diferenčného buffera vždy rýchlejšie ako priamo do hlavného úložiska. Všetky operácie typu Insert, Delete alebo Update sú vykonávané výhradne na diferenčnom bufferi. Pri vykonávaní vyhľadávania je príkaz logicky rozdelený na proces bežiaci v hlavnom úložisku a v diferenčnom bufferi. Po získaní odpovedí z oboch úložísk sa ich kombináciou vytvorí výsledný výsledok, ktorý reprezentuje súčasný stav databázy.

Pre garanciu optimálneho výkonu musí byť diferenčný buffer pravidelne zlučovaný s hlavným úložiskom. Tento proces obsahuje tri fázy: prípravu, zlúčenie atribútov a commit. Na obrázku 11 môžete vidieť detaily tejto operácie. Pred štartom zlučovacej operácie máme diferenčný buffer a hlavné úložisko. Po štarte

musíme vytvoriť nový buffer a nové hlavné úložisko, pretože systém musí neustále bežať, je stále pripravený na spracovanie príkazov, vďaka čomu docielime asynchrónné zlúčenie. Počas zlučovania všetky príkazy na modifikáciu dát prebiehajú nad novým diferenčným bufferom a príkazy čítania z databázy prebiehajú nad starým hlavným úložiskom, starým a novým bufferom. Do nového úložiska zatiaľ zlučujeme staré úložisko a starý buffer. Na konci nám zostane iba nové úložisko a nový buffer. [1]



Obr. 11: Merge operácia [1]

4.5.2 Logovanie a obnova

Ak chceme databázu použiť v produkčnom podnikovom systéme, je potrebné zabezpečiť záruku trvalosti (ako súčasť ACID vlastností). Aby sme mohli poskynúť tieto záruky, musí byť zabezpečená odolnosť voči chybám a vysoká dostupnosť databázy. Štandardný postup umožňujúci obnovu databázy je logovanie. Dáta sa zapisujú do log súborov a ukladajú sa do trvalej pamäte. Logovanie v in-memory databázach má tri časti: snapshot hlavného úložiska, logy hodnôt a slovníka. Snapshot sa väčšinou ukladá po každej operácii zlúčenia a zmeny diferenčného buffera sa zapisujú do log súborov. Hodnoty a databázový slovník sa ukladajú do rozdielnych súborov, aby sa obnova mohla spracovávať paralelne.

Ak server zlyhá, je pravdepodobné, že ho bude potrebné reštartovať a obnoviť. Zlyhanie môže byť spôsobené výpadkom alebo chybou servera. Keď začne obnova systému, najprv sa načítajú metadata o veľkosti tabuliek, posledné snapshoty a

podobne. Keď získame všetky informácie, môžeme paralelne vykonať nasledujúce tri operácie:

- obnova hlavného úložiska zo snapshotu
- obnova slovníka diferenčného buffera pomocou slovníkových logov
- obnova valueID vektorov diferenčného buffera

Po importe log súborov sa vykoná ďalší beh obnovy cez importované záznamy. Je to spôsobené slovníkovo-kódovacím logovaním, ktoré loguje len zmenené stĺpce. Na konci musia byť importovaným záznamom skontrolované a doplnené prázdne atribúty. Robí sa to iteráciou cez všetky verzie záznamu, ktoré sú zaznamenané pomocou validačného príznaku. [1]

5 Základné operácie

Ukladanie dát stĺpcovým spôsobom prináša niekoľko zmien v dátabázovom managemente oproti riadkovému prístupu. V tejto kapitole bližšie priblížím ako fungujú základné operácie nad databázou, ak použijeme stĺpcový prístup.

5.1 Zmazanie záznamu - Delete

Operácia Delete v podstate zruší validitu vybraného záznamu v databáze. Táto operácia môže byť buď fyzická alebo logická. Fyzické vymazanie vymaže záznam z databázy a už nemôže byť vrátený. Na rozdiel od toho, logické vymazanie iba zruší validitu záznamu v tabuľke, ale záznam je stále k dispozícií napríklad kvôli histórií.

Predstavme si, že chceme fyzicky vymazať záznam z tabuľky svetovej populácie, na ktorú je aplikované slovníkové kódovanie pomocou atribútov **meno** a **priezvisko**. Najprv si musíme v príslušných slovníkoch vyhľadať **valueID** pre naše **meno** a **priezvisko**. Potom pomocou **valueID** nájdeme v **atribút vektoroch** príslušný záznam, ktorý vymažeme. Atribút vektory musíme následne upraviť aby nevznikali zbytočné medzery. Táto operácia je príliš časovo náročná. V podkapitole 5.2 popíšem prístup **Insert Only**, vďaka ktorej je možné vykonať logické vymazanie záznamu. [1]

5.2 Vloženie záznamu - Insert, Insert Only

V porovnaní s riadkovým prístupom je vloženie záznamu v stĺpcovom prístupe značne zložitejsie. V riadkovo orientovanej databáze sa nový záznam jednoducho vloží na koniec tabuľky. Napriek tomu pri stĺpcovo orientovanej databáze je potrebné skontrolovať slovník, či sa tam už hodnota nachádza. Ak nie, je potrebné ju vložiť. Nasledovne je konkrétne **valueID** vložené do **atribút vektora**. Ak počítame s tým, že náš slovník je zatriedený, vloženie nového záznamu má tri scenáre:

- **Vloženie bez novej hodnoty v slovníku:** Najjednoduchšia možnosť nastane, keď náš slovník už obsahuje vkladanú hodnotu. V tomto prípade iba vložíme jej valueID na koniec atribút vektora.

- **Vloženie s novou hodnotou v slovníku, ale bez potreby zatriedenia:** v tomto prípade je nová hodnota pridaná do slovníka a zvyšok je rovnaký ako v predchádzajúcom prípade
- **Vloženie s novou hodnotou v slovníku a potrebou triedenia:** pre tento prípad musíme vykonať nasledujúce kroky
  1. Vloženie novej hodnoty na koniec slovníka
  2. Zatriediť slovník
  3. Zmeniť všetky valueID v atribút vektore
  4. Pridať novú valueID do atribút vektora

5.2.1 Insert Only

Dáta v databázových tabuľkách sa časom menia a malo by byť možné tieto zmeny sledovať, pretože podniky potrebujú pristupovať k historickým dátam. Vzhľadom k potrebe uchovávania historických dát vznikol Insert Only prístup. Pomocou neho aplikácie nevykonávajú **Update** a **Delete** operácie nad záznamami fyzicky uloženými v databáze, ale vytvárajú nové záznamy a nastavujú validitu.[1]

Inými slovami, zastarané dáta nie sú upravované ani vymazávané, ale zneplatnené. Zneplatnenie môže byť nastavené pomocou pridaného atribútu k záznamu. Existujú dva prístupy na zneplatnenie záznamu:

1. Bodová reprezentácia - k záznamu sa pridá jeden atribút reprezentujúci od kedy je daný záznam valídny - platný.

2. Intervalová reprezentácia - k záznamu sa pridajú dva atribúty, reprezentujúce od kedy do kedy je záznam platný.

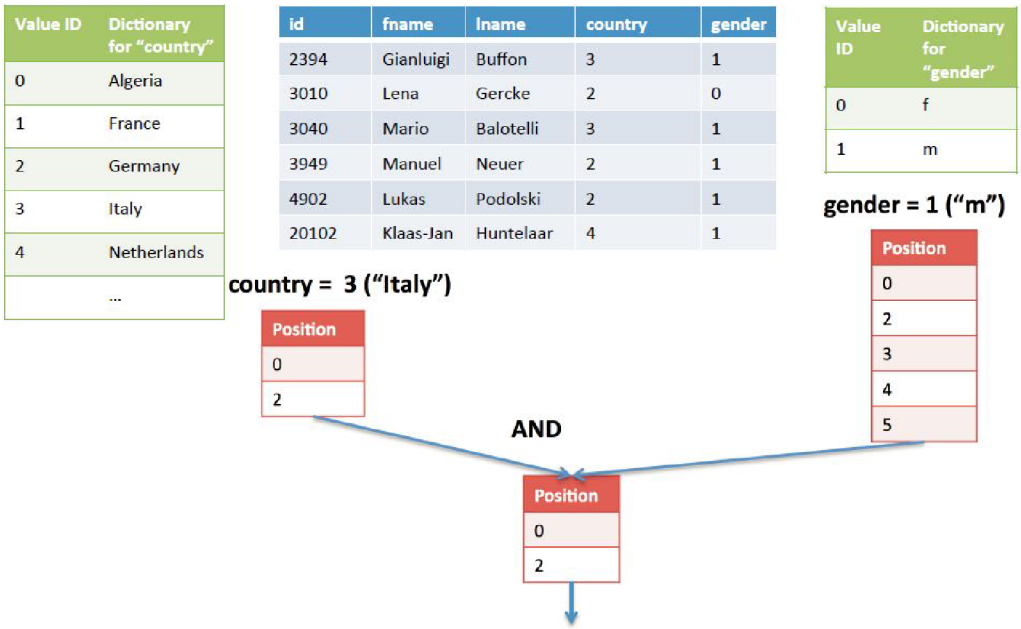
5.3 Vybratie záznamu - Select

SELECT je vo väčšine aplikácií najpoužívanejší príkaz. Slúži na získanie dát z databázy. Reprezentuje deklaratívny popis výsledku dotazovaného z databázy. Aby boli dáta získané, musia sa vykonať presne určené kroky - exekučný plán. Pre

každý dotaz existuje niekoľko exekučných plánov, ale s rôznou výslednou rýchlosťou. Na to aby bol vybraný ten najlepší plán, slúži optimalizátor dotazov. Hlavným cieľom je čo najskôr zredukovať veľkosť výsledku. Je to možné dosiahnuť aplikovaním selekcií čo najskôr, zoradením selekcií (tie najviac obmedzujúce ako prvé) alebo zoradením JOIN operácií (ako prvé sa vykonajú JOIN operácie na menších tabuľkách).[1]

Predstavme si tabuľku svetovej populácie a nad ňou vykonaný nasledujúci príkaz: `SELECT fname, lname FROM population WHERE country = 'Italy' AND gender = 'm'` Na obrázku 12. môžeme vidieť exekučný plán tohto príkazu:

1. Paralelné skenovanie hodnoty ValueID v slovníkoch pre Country a Gender
2. Paralelné skenovanie pre získanie pozície z atribút vektorov
3. Logické AND na pozíciach atribútov Country a Gender
4. Finálny zoznam pozícií, ktoré chceme vybrať



Obr. 12: Ukážka exekučného plánu príkazu SELECT [1]

5.4 Materializované stratégie

Pre potrebu poskytnutia rovnakého dátového rozhrania, ako používa riadkové usporiadanie, je súčasťou príkazu SELECT aj materializácia. Vrátené výsledky musia byť transformované do záznamov v riadkovom formáte. Pre túto operáciu poznáme dva typy materializácie - **skorá** a **neskorá materializácia** (z angličtiny early, late materialization). V závislosti na tom, či sa používajú komprimované alebo nekomprimované dáta, či sa používa alebo nepoužíva slovníkové kódovanie sa vyberá, ktorý typ materializácie sa použije.[1]

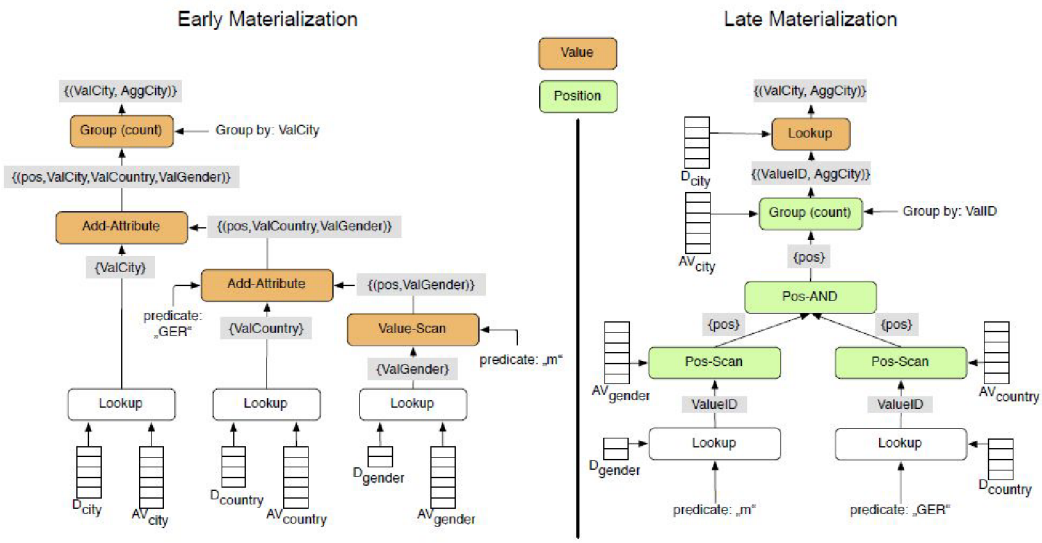
V skratke, skorá materializácia dekomprimuje a dekóduje dáta skorej a potom pracuje napríklad s reťazcami. Zatiaľ čo neskorá materializácia pracuje s číslami a stĺpcami najdlhšie ako sa dá a až potom dekomprimuje a dekóduje. Praktickú ukážku oboch typov je možné vidieť na obrázku 13. Na tabuľku svetovej populácie, ktorú som spomínal v predchádzajúcich kapitolách, je aplikovaný príkaz `SELECT city, COUNT(*) FROM population WHERE gender = 'm' AND country 'GER' GROUP BY city`. Pre slovníkovo zakódované stĺpcové dátové štruktúry sa väčšinou používa neskorá materializácia. Pri skorej materializácií sú všetky požadované stĺpce najprv materializované. Podľa obrázka 13 musíme pre skorú materializáciu vykonať tieto kroky:

1. Skenovanie kvôli obmedzujúcej podmienke na atribúte **GENDER**.
2. Skenovanie kvôli podmienke na **COUNTRY** a pridanie do výsledku.
3. Vyhľadanie **hodnôt CITY** v **slovníku**.
4. Pridanie **atribút vektora CITY** do výsledku.
5. Vykonanie operácie **GROUP** a **COUNT**.

V neskorej materializácii je vyhľadávanie v slovníku vykonané v poslednom kroku pred vrátením výsledku. Na pravej strane obrázka 13 sú vykonané nasledujúce kroky:

1. Vyhľadanie hodnôt obmedzujúcich podmienok v **slovníku**.
2. Skenovanie obmedzujúcich podmienok v **atribút vektore**.

3. Logické **AND**.
4. Vykonanie operácie **GROUP** a **COUNT**.
5. Vyhľadanie **hodnôt CITY** v **slovníku**.



Obr. 13: Ukážka porovnania skorej a neskorej materializácie [1]

5.5 Spojenie - Join

Pre maximálne využitie vlastností in-memory databáz sa používajú dva spojovacie algoritmy - **Hash Join** a **Sort-merge Join**.

Hash Join používa hašovaciu funkciu na vytvorenie tabuľky s konštantným prístupom (hašovacia mapa - Hash map) pre spojovacie predikáty prvej tabuľky. Táto hašovacia mapa sa potom používa v druhej fáze algoritmu na prehľadanie každého spojovacieho predikátu druhej tabuľky pre prípadné zhody s hodnotami prvej tabuľky.

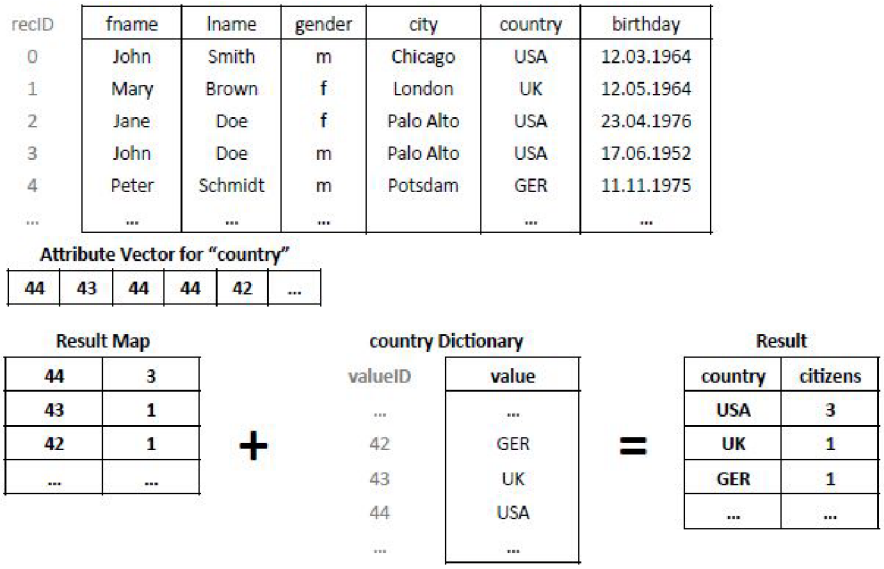
Sorted-merge Join funguje iným spôsobom. Obidve tabuľky sú najprv zatriedené podľa spojovacieho predikátu. Spojovacia fáza je vďaka zatriedeným tabuľkám efektívna, pretože k spájaným záznamom v oboch tabuľkách pristupuje takmer naraz.[1]

5.6 Agregačné funkcie

Vykonávanie agregačných funkcií v in-memory databázach najlepšie vysvetlím na príklade. Na ukážku použijem agregačnú funkciu **COUNT** a už známu tabuľku svetovej populácie.

Na obrázku 14 je výrez z tabuľky svetovej populácie a znázornené jednotlivé kroky **COUNT** funkcie. Úlohou príkladu je získať počet obyvateľov každej krajiny použitím nasledujúceho príkazu:

```
SELECT country, COUNT(*) AS citizens FROM world_population
   GROUP BY country
```



Obr. 14: Ukážka agregačnej funkcie COUNT [1]

Najprv systém prejde celým atribút vektorom pre atribút COUNTRY a vytvára pomocnú tabuľku **Result Map**. Pre každú novú nájdenú hodnotu **ValueID** je do pomocnej tabuľky pridaný nový záznam s počiatočnou hodnotou "1". Ak nájdená hodnota **ValueID** už existuje v **pomocnej tabuľke**, tak hodnota v **Result Map** je zvýšená o jedna. Po tomto procese obsahuje pomocná tabuľka ValueID každej krajiny a počet jej výskytov v **atribút vektore**.

Potom prichádza materializácia výsledku. Zo **slovníka** sa získajú názvy krajín pomocou **ValueID** v pomocnej tabuľke a je vytvorený finálny výsledok funkcie **COUNT**. Výsledkom je tabuľka, ktorá obsahuje dva atribúty - **názov krajiny** a **počet jej výskytu** v tabuľke populácie, ktorý korešponduje s počtom obyvateľov.

Rovnakým spôsobom sa vykonávajú aj ďalšie agregačné funkcie **SUM**, **AVG**, **MAX** a **MIN**.[1]

6 Testovacie prostredie

Dostávame sa do praktickej časti mojej diplomovej práce. V nasledujúcich kapitolách stručne predstavím databázové systémy, na ktorých som vykonával výkonové testy. Ďalej popíšem svoju aplikáciu, ktorú som použil na zjednodušenie testovania, predstavím programovací jazyk, v ktorom som danú aplikáciu naprogramoval. Nakoniec popíšem dátový model použitý pri testovaní a vysvetlím priebeh jednotlivých testov.

6.1 Databázové systémy

V nasledujúcej podkapitole v stručnosti popíšem jednotlivé databázové systémy, ktoré som použil na otestovanie rýchlosti tradičných databázových systémov s in-memory systémom. Vzhľadom na to, že pracujem v spoločnosti SAP, ktorá sa zaoberá tvorbou informačných systémov a patrí medzi priekopníkov v in-memory databázových systémoch, som mal príležitosť vykonať svoje testy priamo na interných systémoch. Celkovo som testy vykonal na štyroch systémoch - **SAP HANA**, **Oracle**, **MaxDB** a **Sybase**. Kedže všetky tieto databázy bežia na interných systémoch, hardwarová konfigurácia serverov bola veľmi podobná až rovnaká. Všetky systémy sú na serveroch s operačným systémom Linux x64.

6.1.1 SAP HANA

SAP HANA je aplikačná a databázová platforma firmy **SAP**, ktorá umožňuje rýchle spracovanie veľkých objemov dát v reálnom čase a zároveň aj ich analýzu. SAP HANA pracuje na princípe in-memory data managementu popísaného v kapitole 3. Pracuje s dátami výhradne len v pamäti a disk používa len na zálohu, obnovu a logovanie. Dáta v SAP HANA-e sa ukladajú stĺpcovým spôsobom. Na moje testovanie som systém SAP HANA použil dvomi spôsobmi. Chcel som porovnať, či stačí mať iba in-memory databázu alebo dôležitým faktorom pre rýchlosť tohto typu databázy je aj stĺpcový prístup k dátam. Tým pádom som SAP HANA-u použil v testoch s tabuľkami s riadkovým prístupom a s tabuľkami so stĺpcovým prístupom.[5]

6.1.2 Oracle

Oracle je moderný multiplatformový databázový systém vyvíjaný spoločnosťou Oracle Corporation. Je charakteristický pokročilými možnosťami spravovania dát, vysokým výkonom a jednoduchou škálovateľnosťou. Oracle systém ukladá dáta logicky vo forme tabuľkového priestoru a fyzicky vo forme dátových súborov. Tabuľkový priestor môže obsahovať rôzne typy pamäťových segmentov (dátové segmenty, indexové segmenty ...). Segmenty následne tvoria jeden alebo viac úsekov. Úsek zahŕňa jeden alebo viac dátových blokov. Dátové bloky tvoria základnú jednotku úložného priestoru dát. Na fyzickej úrovni dátové súbory obsahujú jeden alebo viac dátových blokov.

Oracle databáza monitoruje veľkosť v dátovom úložisku pomocou informácií uložených v Systémovej tabuľke. Táto tabuľka obsahuje dátový slovník, indexy a clustery. Dátový slovník obsahuje niekoľko tabuliek a pohľadov, ktoré popisujú štruktúru databázy. Ukladá sa do neho informácia a logickej a fyzickej štruktúre databázy. Dátový slovník obsahuje napríklad informácie o užívateľoch a ich právach, názvoch a typoch všetkých stĺpcov použitých v databázových tabuľkách a informácie o alokovanom a použitom mieste databázovými objektami.[6]

Na testovanie som použil Oracle databázu na internom systéme spoločnosti SAP s riadkovým prístupom.

6.1.3 MaxDB

MaxDB je ANSI SQL-92 kompatibilný systém riadenia relačných databáz patriaci pod spoločnosť SAP AG. Tvorba tohto systému začala v roku 1977 na Technickej Univerzite v Berlíne. Od roku 1977 patril pod viacero spoločností a mal rôzne názvy. V roku 1997 ho získal SAP AG a vyvíjal ho pod menom SAP DB. V roku 2003 SAP AG a MySQL AB ho spoločne premenovali na MaxDB. Medzi rokmu 2003 a 2007 bol tento systém dodávaný aj spoločnosťou MySQL AB. Od roku 2007 patrí iba pod SAP AG.[8]

MaxDB je hlavne zameraný na splnenie požiadaviek SAP zákazníkov a SAP aplikácií, slúži ako lacnejšia alternatíva oproti systémom od iných dodávateľov. Dnes, väčšina SAP zákazníkov prechádza na SAP HANA databázový systém, ale ešte

stále množstvo zákazníkov a interných systémov používa MaxDB. To mi umožnilo aby som mohol tento systém použiť v rámci môjho testovania ako ďalšiu tradičnú relačnú databázu s riadkovým prístupom.

MaxDB používa technológiu SAP liveCache, ktorá umožnuje systému vysokú spoľahlivosť, škálovateľnosť a dostupnosť pre konfigurácie všetkých veľkostí. SAP liveCache technológia je objektovo orientované rozšírenie MaxDB databázového systému. Vytvára pocit primitívnej in-memory databázy. Vďaka tejto technológií si systém drží v pamäti najviac používané a pre systém dôležité dáta. Poskytuje dátové štruktúry a metódy pre rýchlejšie spracovávanie.[9]

6.1.4 Sybase

Sybase SQL server je databázový systém vytvorený spoločnosťou Sybase Inc. Databázové jadro skryté za Microsoft SQL Server a Sybase SQL server je to isté. V roku 1992 chcela firma Microsoft preniknúť na trh databázových systémov a tak sa obrátila na spoločnosť Sybase. Kód Sybase SQL serveru bol základným kameňom na vytvorenie prvého Microsoft SQL serveru. Aj keď svoju spoluprácu spoločnosti po niekoľkých rokoch ukončili a pokračovali v tvorbe systémov osobitne, základné jadro ich databázových systémov je stále veľmi podobné a vychádza z rovnakých začiatkov.[7]

Na testovanie som použil databázu Sybase, kedže väčšina interných systémov bežiacich na MSSQL bola v priebehu roka preportovaná na Sybase databázový systém. V rámci svojho testovania som použil jeden takýto systém s riadkovým prístupom.

6.2 Testovacia aplikácia

Na zjednodušenie a sprehľadnenie testovania som vytvoril jednoduchú aplikáciu. Vďaka tomu, že pracujem ako vývojár vo firme SAP, vybavil som si prístup k štyrom systémom, na ktorých som vykonal testy. Aplikáciu som vytvoril na systéme s databázovým systémom SAP HANA, v zvyšných troch systémoch som iba vytvoril funkcie, ktoré som vzdialene volal zo svojej aplikácie. V tejto podkapitole zjednodušene predstavím svoju aplikáciu, programovací jazyk **ABAP**, v ktorom je aplikácia napísaná a technológiu, vďaka ktorej možem vzdialene volať funkcie z iného systému. Nakoniec vysvetlím dátový model, na ktorom prebehlo testovanie.

6.2.1 Programovací jazyk ABAP

ABAP je high-level programovací jazyk vytvorený spoločnosťou SAP. V roku 1999 bolo na trh predstavené objektovo orientované rozšírenie. ABAP bol jedným z prvých jazykov, ktorý zahŕňa koncept logickej databázy, ktorý poskytuje vysokú mieru abstrakcie medzi podnikovými aplikáciami, operačným systémom a databázou. Vďaka tomu aplikácie nezávisia na špecifickom serveri alebo databáze a môžu byť jednoducho portované z jednej platformy na druhú.

Všetky programy napísané v jazyku ABAP sú uložené v SAP databáze. Nie sú uložene v externých súboroch ako to používa C++ alebo Java. V databáze sú programy uložené v dvoch formách: zdrojový kód, ktorý môže byť prehliadaný a editovaný pomocou ABAP Workbenchu. A generovaný kód, binárna reprezentácia, ktorú je možné prirovnať k Java bytekódu. Vykonávanie ABAP programov je riadené pomocou runtime systému, ktorý je súčasťou SAP kernelu. Runtime systém je zodpovedný za vykonávanie ABAP príkazov, kontroluje logiku vykonávania obrazoviek a reagovanie na eventy (napríklad kliknutie myšou). Kľúčovou komponentou Runtime systému je databázové rozhranie, ktoré prekladá databázovo nezávislé ABAP príkazy (Open SQL) do príkazov databázy, nad ktorou beží aplikácia.[10]

Programy v ABAP-e sa rozdeľujú na dva typy:

- Spustiteľné programy
  - **Reporty** - sú jednoduché programy, v ktorých užívateľ zadá vstupné

parametre a program na základe týchto parametrov vyprodukuje report vo forme interaktívneho listu.
  - **Module pools** - definuje zložitejší spôsob interakcie s užívateľom pomocou viacerých obrazoviek. Obrazovka znamená fyzický obraz, ktorý užívateľ vidí a pracuje s ním (zadáva parametre, prepína medzi obrazovkami, atď). Každá obrazovka má svoju vlastnú logiku spracovania. Moja aplikácia je Module pool, pretože používa jednu obrazovku a niekoľko podobrazoviek.

- Nespustiteľné programy
  - **INCLUDE moduly** - slúžia na rozdelenie veľkých programov na menšie časti (napríklad deklarácia premenných, inicializačné funkcie a podobne)
  - **Subroutine pools** - v inom slova zmysle sa dajú pomenovať ako funkcie vo vnútri jednotlivých programov. Sú uzavreté pomocou FORM/ENDFORM a volajú sa príkazom PERFORM.
  - **Funkčné grupy** - ak chceme aby sa kód mohol volať z viacerých programov, vytvoríme Funkčné moduly. Funkčné moduly sa následne zoskupujú do takzvaných knižníc - funkčných grup
  - **Objektové triedy** - slúžia podobne ako v iných programovacích jazykoch, ako napríklad Java. Obsahujú metódy a atribúty
  - **Rozhrania** - obsahujú definície metód
  - **Type pools** - v jednotlivých programoch si pomocou Type poolov môžeme definovať dátové typy a konštanty.

6.2.2 RFC - Vzdialené volanie funkcií

Aby som nemusel vytvárať svoju aplikáciu v každom systéme, zvolil som si jeden systém ako hlavný a z neho som potom volal funkcie vo zvyšných systémoch. Za hlavný systém som zvolil systém s databázou SAP HANA. V ostatných som len vytvoril funkciu pre každý test. Z hlavného systému som zavolal túto funkciu a naspäť som vracal čas testu. V ABAP-e sa na vzdialenú komunikáciu používa technológia RFC.

RFC (Remote Function Call) je rozhranie, ktoré umožnuje komunikáciu medzi SAP systémami alebo inými externými systémami. Používa na to TCP/IP alebo CPI-C spojenie. Na hlavnom systéme je iba potrebné definovať RFC spojenie s odkazom na cieľový systém. Potom v kóde aplikácie sa zavolá funkcia z cieľového systému s dodatočným parametrom **DESTINATION**. Pre predstavu, v cieľovom systéme mám vytvorenú funkciu `TEST1`. Mám nadefinované RFC spojenie s názvom `RFC_TO_MAX`. V kóde aplikácie v hlavnom systéme zavolám funkciu týmto spôsobom: `CALL FUNCTION 'TEST1' DESTINATION 'RFC_TO_MAX'`.[12]

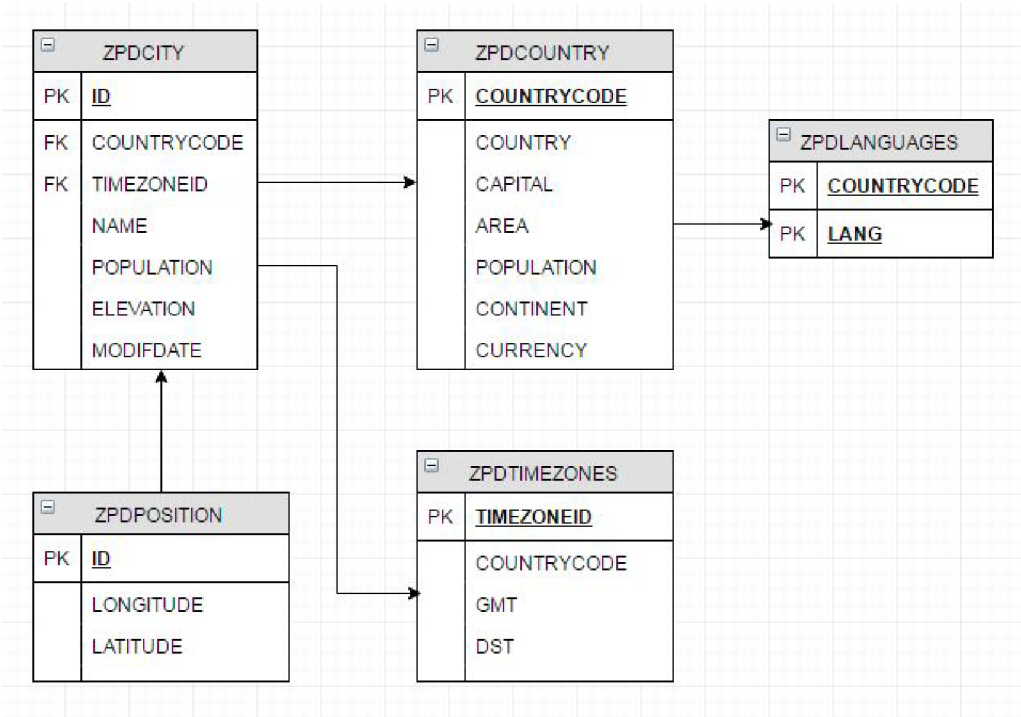
6.2.3 Dátový model

Jednou z mojich najdôležitejších úloh bolo nájsť vhodné dáta na testovanie. Aby moje výsledky mali nejaký zmysel, nestačilo len použiť jednu tabuľku s množstvom záznamov.

Ako zdroj dát pre moje testovanie som použil voľne dostupnú geografickú databázu GeoNames. Informácie o tejto databáze sú umiestnené na internetovej stránke www.geonames.org[11], z ktorej je aj možné stiahnuť textové súbory s obsahom databázy. Táto databáza obsahuje základné informácie o všetkých mestách a krajinách vo svete. Spolu z ich geografickými informáciami. Pre svoje testovanie som z tejto stránky použil tri textové súbory: informácie o mestách, krajinách a časových zónach. Kedže textový súbor s informáciami o mestách obsahoval najviac záznamov (okolo 10 miliónov), rozhodol som sa, že z neho vytvorím svoju hlavnú tabuľku. Vznikli mi dve tabuľky ZPDCITY a ZPDPOSITION. V tabuľke ZPDCITY sú uložené základné informácie o meste, kód krajiny do ktorej patrí, názov, počet obyvateľov a podobne. V tabuľke ZPDPOSITION je uložená pozícia každého mesta. Na obrázku 15 môžete vidieť celý dátový model použitý v mojej aplikácii. Textový súbor s informáciami o krajinách som použil na vytvorenie ďalších dvoch databázových tabuliek ZPDCOUNTRY a ZPDLANGUAGES. V tabuľke ZPDCOUNTRY je uložená základná informácia o krajine, jej kód, názov, hlavné mesto, kontinent a podobne. ZPDLANGUAGES je pomocná tabuľka, v ktorej sú uložené jazyky, ktoré sa v danej krajine používajú. Z posledného textového súboru o časových zónach som vytvoril databázovú tabuľku ZPDTIMEZONES.

Vzhľadom na to, že som chcel moje testovanie vykonať na rôznom počte dát,

musel som tabuľku ZPDCITY naplniť postupne rôznym počtom záznamov - **10 000, 50 000, 100 000, 500 000, 1 milión a 5 miliónov** záznamov. Vzhľadom na to, že tabuľka ZPDCITY je prepojená pomocou kódu krajiny s tabuľkou ZPDCOUNTRY, musel som zabezpečiť, že keď bude v tabuľke iba 10 000 záznamov, nebudú všetky z jednej krajiny. Inak by neboli testy na menšom počte dát relevantné. Kvôli tomu som si vytvoril vedľajšiu aplikáciu, pomocou ktorej som zmenšoval počet miest takým spôsobom, aby pomer miest z jednotlivých krajín bol vždy rovnaký.



Obr. 15: Dátový model

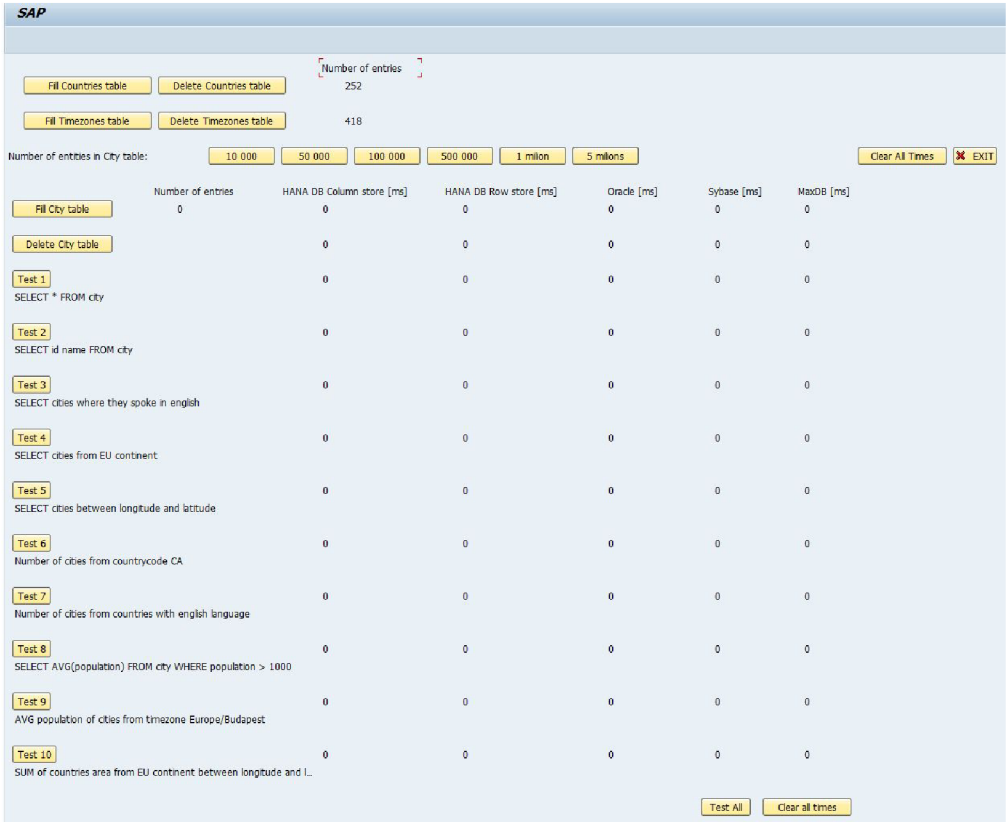
6.2.4 Implementácia

Moju aplikáciu som vytvoril pomocou programovacieho jazyka ABAP a editoru ABAP Workbench, ktorý sa používa v interných systémoch spoločnosti SAP.

Aplikácia sa skladá z jednej hlavnej obrazovky a zo siedmich podobrazoviek, každá pre jednu množinu dát. Na obrázku 16 môžete vidieť vzhľad aplikácie. Na vrchnej časti sa nachádzajú tlačítka na vloženie a vymazanie tabuliek krajín (zpd-

country, zpdlanguages) a časových zón (zpdtimezones). Vedľa nich sa nachádza informácia o počte záznamov v týchto tabuľkách. Pod nimi sa nachádzajú tlačítka na prepínanie medzi jednotlivými podobrazovkami. Na kraji je umiestnené tlačítko na vymazanie všetkých nameraných časov a vypnutie aplikácie. Zvyšok obrazovky je vyplnený jednotlivými podobrazovkami, podľa toho, ktorú množinu dát chceme testovať.

Každá podobrazovka obsahuje tlačítka na naplnenie a vymazanie tabuľky miest. Následne tlačitka na vykonanie postupne každého z desiatich testov. Väčšinu obrazovky vypĺňa tabuľka, ktorej záhlavie tvoria jednotlivé testované databázy a riadky sú testy. Na spodnej časti obrazovky sa nachádza tlačítko, ktoré vykoná všetky testy naraz a ďalšie tlačítko na vymazanie všetkých nameraných časov, len pre zvolenú množinu dát.



Obr. 16: Obrázok aplikácie

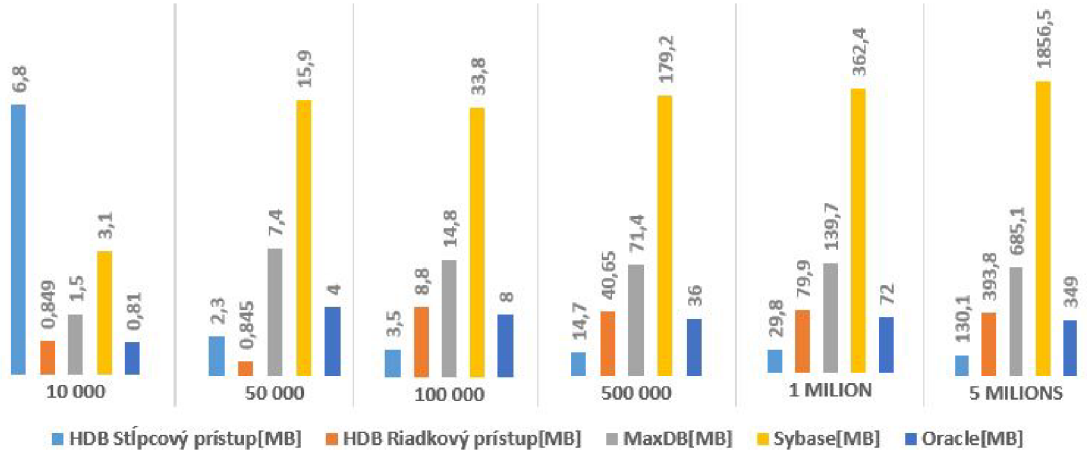
7 Výsledky testovania

V tejto podkapitole popíšem jednotlivé testy, ktoré som použil na otestovanie rýchlosti databáz. Ako som spomínal v kapitole o dátovom modeli, tabuľka Miest je najdôležitejšia tabuľka. Každý test je založený na získaní dát z tejto tabuľky plus prípadne spojením so zvyšnými tabuľkami. Každý test som vykonal nad piatimi možnými veľkosťami tabuľky Miest. Presnejšie, tabuľka miest postupne obsahovala **10 000, 50 000, 100 000, 500 000, 1 milión a 5 miliónov** záznamov. Najprv som testoval koľko miesta zaberá tabuľka Miest v jednotlivých databázach pri rôznom počte záznamov. Ďalej som otestoval vkladanie a vymazávanie z tabuľky. Hlavným ukazovateľom výkonu databáz bolo testovanie príkazu SELECT na získanie záznamu z tabuľky alebo výpočtu agregačných funkcií. Celkovo som na príkaz **SELECT** vykonal 10 testov. Z toho päť testov bolo na jednoduchú rekonštrukciu záznamu z tabuľky. Zvyšných päť testov bolo na použitie agregačných funkcií ako napríklad **AVG**, **SUM** a **COUNT**. Každý typ testu je otestovaný najprv nad jednou tabuľkou a potom nad tou istou tabuľkou ale spojenou príkazom **JOIN** s inými tabuľkami z dátového modelu. Každý test som vykonal 10-krát a do grafu som použil priemer z týchto desiatich pokusov.

V prílohe 9 na konci diplomovej práce sú obrázky exekučných plánov z každej databázy pre všetky vykonané testy. Bohužiaľ interný systém s databázou Sybase používa na zobrazovanie SQL príkazov nové webové prostredie a prehliadanie exekučných plánov nie je funkčné. Vzhľadom k tomu exekučné plány k databáze Sybase chýbajú.

Mojím očakávaním pred testovaním bolo, že SAP HANA databáza by mala byť vo väčšine prípadov jednoznačne rýchlejšia ako zvyšné databáze.

7.1 Veľkosť databázy



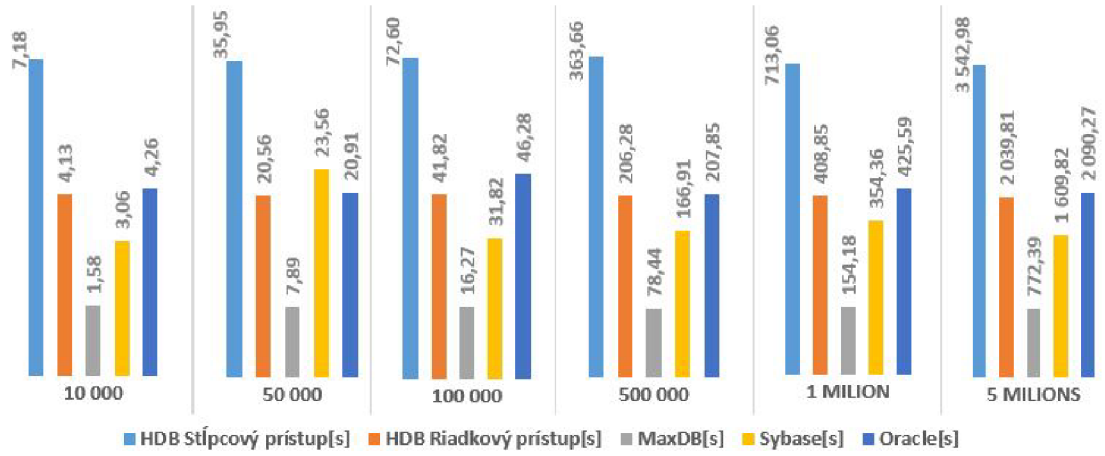
Obr. 17: Graf porovnania veľkostí jednotlivých databáz

Obrázok 17 zobrazuje veľkosti tabuľky ZPDCITY v jednotlivých databázach pri rôznom počte záznamov v tejto tabuľke. Celkovo, čo sa týka veľkosti, najhoršie dopadli databázy **MaxDB** a **Sybase**. Obidve databázy alokujú diskovú pamäť pre všetky atribúty záznamu. Aj keď je hodnota nejakého atribútu NULL (je prázdny), tak databáza alokuje miesto pre tento atribút v závislosti od dátového typu. Kvôli tomu tieto databázy zaberajú najviac miesta.

Zaujímavé je, že **SAP HANA so stĺpcovým prístupom** dopadla pri 10 000 záznamoch najhoršie a pri 5 miliónoch dopadla najlepšie. Je na to jednoduché vysvetlenie. Pri 10 000 záznamoch sa nedá naplno využiť slovníkové kódovanie a kompresné techniky, pretože je v tabuľke ešte veľmi málo záznamov a tieto kompresné techniky sú skôr na škodu ako na úžitok. Pri piatich miliónoch je už v tabuľke dostatok záznamov a databáza môže naplno využiť kompresné techniky.

Ďalšou otázkou je, prečo **SAP HANA s riadkovým prístupom** má väčšiu veľkosť a pri 10 000 záznamoch nie je taký očividný skok. Väčšina kompresných techník in-memory databáz je výhodných len vďaka stĺpcovému prístupu. Vzhľadom k tomu sa pri riadkovom prístupe nepoužije toľko kompresií.

7.2 Insert - Vloženie záznamov

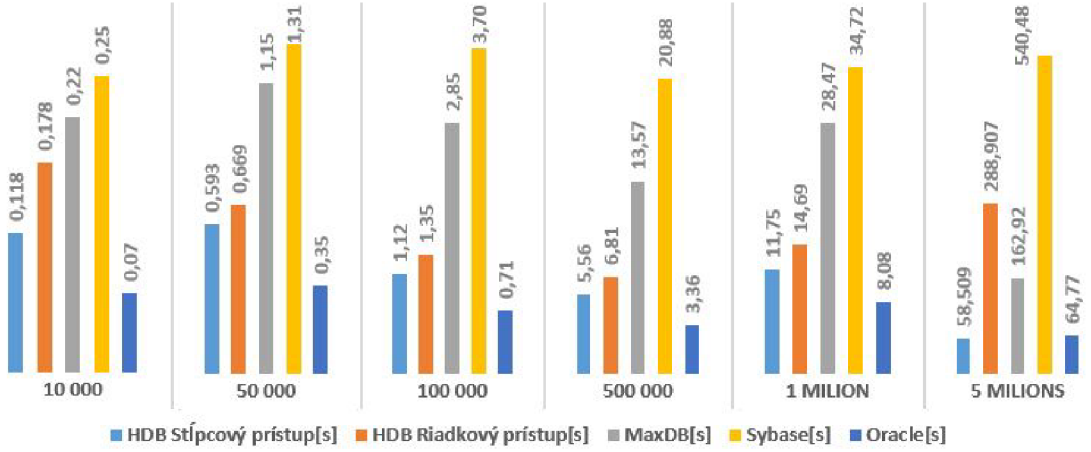


Obr. 18: Insert

Vkladanie záznamov do databázy som testoval dvomi spôsobmi.

Prvý spôsob spočíval v tom, že som jednotlivé množiny záznamov vkladal do tabuľky ZPDCITY záznam po zázname. To znamená, že som napríklad pre množinu 10 000 záznamov, použil 10 000-krát príkaz INSERT. Graf výsledkov prvého spôsobu je možné vidieť na obrázku 18.

Podľa očakávania, rýchlosť INSERTU je najpomalšia pri in-memory databáze **SAP HANA so stĺpcovým prístupom**. Ako som vysvetlil v kapitole 5.2, vkladanie stĺpcovým spôsobom je náročné na čas plus sa ešte prepočítavajú hodnoty v slovníku a atribút vektore. Najrýchlejšia bola databáza **MaxDB**, pretože táto databáza si pri ukladaní nepočíta žiadne pomocné indexy. Približne na rovnakej rýchlostnej úrovni je **SAP HANA s riadkovým prístupom** a **Oracle** databáza. Tu vidíme, že pri jednotlivých INSERToch je jedno, či sú dáta v pamäti alebo na disku, pri riadkovom prístupe je rýchlosť podobná. Druhou najrýchlejšou je **Sybase** databáza, ktorá podobne ako **MaxDB** si pri vkladaní nepočíta žiadne nadbytočné indexy. V tomto momente sú **MaxDB** a **Sybase** najrýchlejšie riešenia, ale čo obidve databázy získali časovo pri INSERTe, strácajú pri SELECT príkazoch.



Obr. 19: Insert

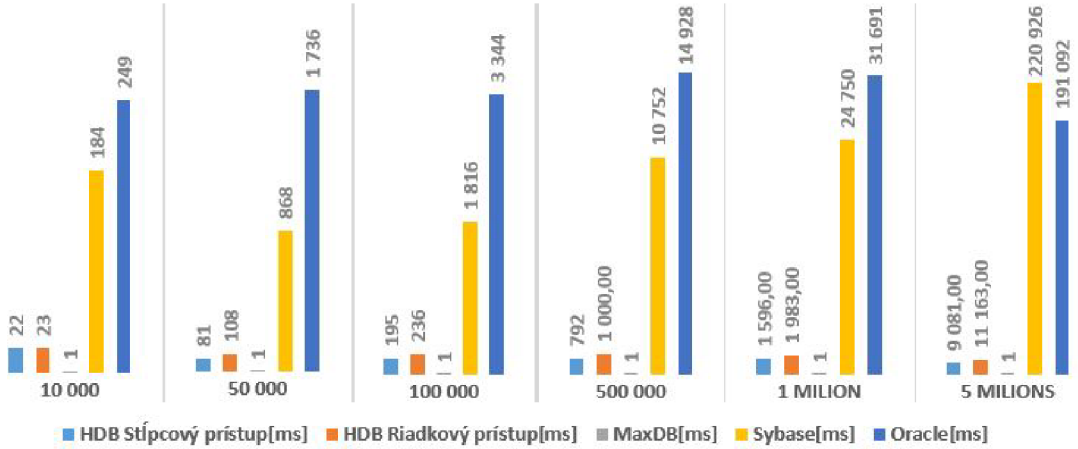
Pri druhom spôsobe som si dáta najprv vložil do internej tabuľky a tú som potom naraz jedným príkazom vložil do databázovej tabuľky. Výsledky pri tomto teste boli opačné oproti prvému spôsobu. Výsledky môžeme vidieť na obrázku 19.

Najrýchlejšou databázou bola **Oracle** a druhou najrýchlejšou **SAP HANA so stĺpcovým prístupom**. Ale všimnime si, že s narastajúcim počtom záznamov sa rozdiel medzi týmito dvomi databázami zmenšoval. Pri vložení 5 miliónov záznamov bola **HDB so stĺpcovým prístupom** dokonca o niečo rýchlejšia. Dôležité je podotknúť, že 5 miliónov záznamov nebolo možné vložiť jedným súborom. Prvý súbor som vložil príkazom INSERT a zvyšné príkazom MODIFY. To spôsobilo aj spomalenie SAP HANA databázy s riadkovým prístupom.

Pri vkladaní viacerých záznamov naraz, databázy **MaxDB** a **Sybase** výrazne zaostávajú, nedokážu rychlejšie spracovať viacero záznamov naraz. Ale môžeme si všimnúť, že pri piatich miliónoch príkaz MODIFY výrazne zrýchlil **MaxDB**.

SAP HANA so stĺpcovým prístupom je rýchlejšia ako **SAP HANA s riadkovým prístupom**, pretože už interná tabuľka je uložená v pamäti stĺpcovým spôsobom a trvá dlhšie kým sa prekonvertuje do riadkového prístupu.

7.3 Delete - Vymazanie záznamov



Obr. 20: Delete

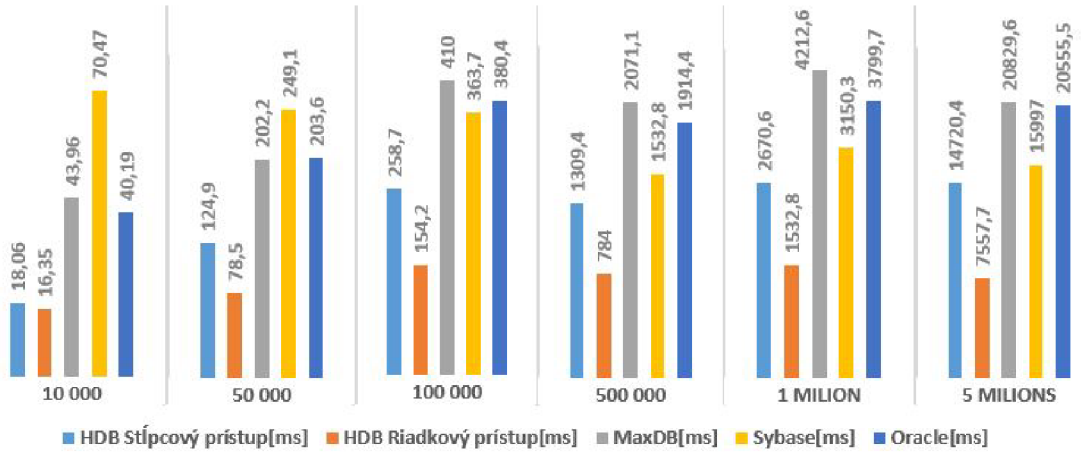
Vymazávanie z tabuľky som testoval iba jedným spôsobom. Použil som na to príkaz `DELETE FROM názov_tabuľky`. Ako možeme vidieť na obrázku 20, najrýchlejšia je **MaxDB** databáza. Vymazanie všetkých záznamov, ľubovoľného počtu, jej vždy trvalo konštantný čas. Je to vďaka technológii SAP liveCache, vďaka ktorej sa **MaxDB** správa ako jednoduchá in-memory databáza, tým pádom vymazanie je extrémne rýchle. Plus, **MaxDB** si nedrží žiadne indexy na optimalizáciu dotazov, takže jej stačí iba vymazať všetky záznamy bez zbytočnej réžie.

Druhá najrýchlejšia je **SAP HANA databáza s obidvomi prístupmi**. Rozdiely medzi nimi sú zanedbateľné. Vďaka tomu, že si všetky dáta drží v pamäti, je jej vymazanie veľmi rýchle.

Najhoršie v tomto teste skončili databázy **Sybase** a **Oracle**. Vzhľadom na to, že sú obe databázy čisto diskovo orientované, bol tento výsledok očakávaný. Zaujímavé je, že iba pri množine s piatimi miliónmi záznamov bol **Oracle** rýchlejší ako **Sybase**. Podľa vývoja grafu, sa dá očakávať, že s pribúdajúcim počtom záznamov, by bol tento rozdiel väčší.

7.4 Test 1

```
SELECT * FROM zpdcity.
```



Obr. 21: Test1

Tento príkaz vracia všetky záznamy z tabuľky ZPDCITY.

Prvý test sa zameriava na získanie všetkých záznamov z tabuľky, postupne vyberá všetky atribúty záznam po zázname.

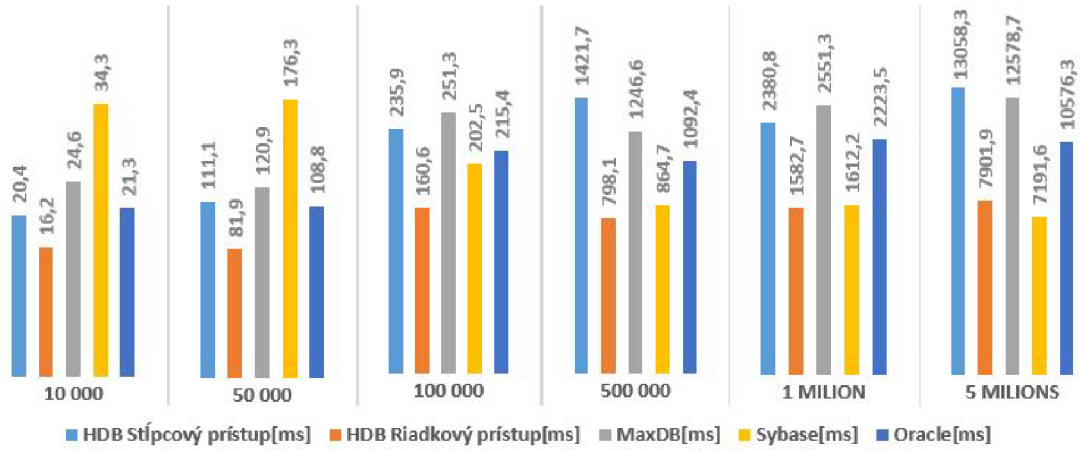
Z obrázka 21 je jasné, že najrýchlejšia je **SAP HANA**. Samozrejme pre získavanie celého záznamu je lepší riadkový prístup a tak **SAP HANA s riadkovým prístupom** je výrazne lepšia ako so stĺpcovým prístupom.

Pomocou exekučných plánov pre prvý test 9.1.1 je očividné, že spomalenie stĺpcového prístupu spôsobila aj technika neskorej materializácie, ktorá sa vykoná z dôvodu dekódovania dát.

Je potrebné si aj všimnúť, že s pribúdajúcim počtom záznamov sa výkon stĺpcového prístupu výrazne zhoršuje, až sa postupne vyrovnáva s výkonom **Sybase** databázy. Keď si detailnejšie prezrieme graf, môžeme si všimnúť, že **Sybase** si počas všetkých množín záznamov, udržiava lineárne spomaľovanie. Napriek tomu **SAP HANA so stĺpcovým prístupom**, **MaxDB** a **Oracle** sa s pribúdajúcimi záznamami výrazne spomaľujú. Pri stĺpcovom prístupe je to logické, pretože databáza robí stále nad väčším počtom záznamov neskorú materializáciu.

7.5 Test 2

```
SELECT id name FROM zpdcity.
```



Obr. 22: Test 2

Pri druhom teste som chcel taktiež otestovať výber všetkých záznamov, ale len dvoch atribútov, pre všetky záznamy - ID a NAME.

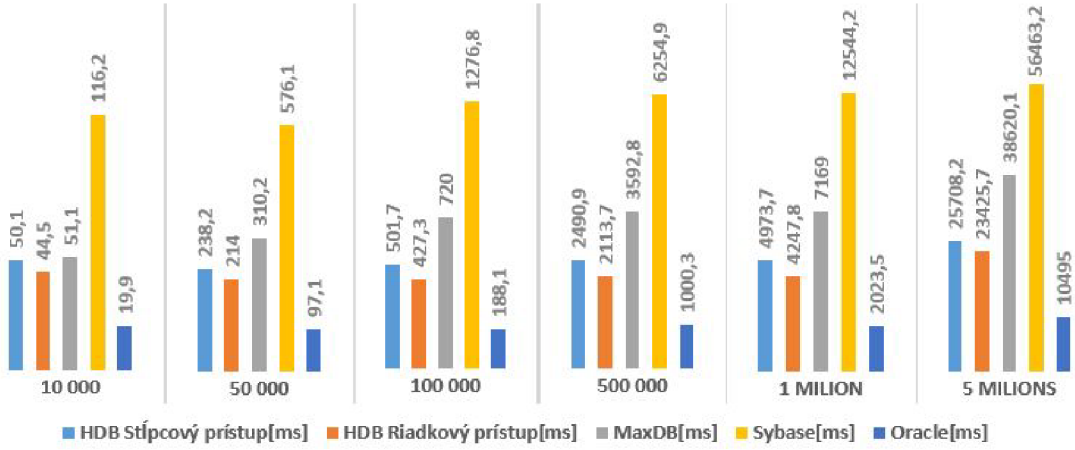
Pri tomto teste som očakával, že najrýchlejšia bude **SAP HANA so stĺpcovým prístupom**, no bol som nemilo prekvapený. Dôvodom zlého výsledku stĺpcového prístupu bola opäť neskorá materializácia. Ale najdôležitejším dôvodom bolo, že atribút NAME je typu CHAR. Takže dekódovanie tohto atribútu výrazne spomaľovalo celkový výkon databázy.

Podobne ako v prvom teste aj v tomto si pri veľkom počte záznamov najlepšie počínala databáza **Sybase**. Cez všetky množiny záznamov si opäť držala lineárne spomaľovanie.

MaxDB a **Oracle** sa postupne pri väčších počtoch dát zhoršovali, ale najvýraznejšie sa spomalila práve **SAP HANA so stĺpcovým prístupom**. Pri exekučných plánoch 9.1.2 si môžeme všimnúť. že pre **Oracle** a **MaxDB** sa tieto plány vôbec nezmenili oproti prvému testu.

7.6 Test 3

```
SELECT c~id c~name FROM zpdcountry AS co
   JOIN zpdcity AS c ON c~countrycode = co~countrycode
   JOIN zpdlanguages AS l ON co~countrycode = l~countrycode
 WHERE lang = 'en'.
```



Obr. 23: Test 3

Tento príkaz vyberá všetky ID a názvy miest, ktoré sa nachádzajú v krajinách kde sa hovorí anglickým jazykom.

Týmto testom som chcel otestovať taktiež výber dvoch atribútov z informácií o mestách, ale s príkazom JOIN určeným na spojenie s ďalšími dvomi tabuľkami. Presnejšie s tabuľkami ZPDCOUNTRY a ZPDLANGUAGES.

Z obrázka 23 môžeme vyčítať, že tento príkaz najrýchlejšie zvládla databáza **Oracle**. Druhá bola databáza **SAP HANA pre obidva prístupy**, rozdiely medzi nimi boli minimálne. Najpomalšia databáza bola **Sybase**.

Pomocou exekučných plánov 9.1.3 pochopíme, prečo to tak dopadlo. Pri exekučnom pláne **Oracle** databáze vidíme, že najprv sa vykoná JOIN pomocou vnorených cyklov na menších tabuľkách ZPDCOUNTRY A ZPDLANGUAGES. Pomocou týchto vnorených cyklov sa vyfiltrujú len tie krajiny, v ktorých sa hovorí po anglicky. JOIN je vykonaný pomocou indexov, pretože spájame cez primárne

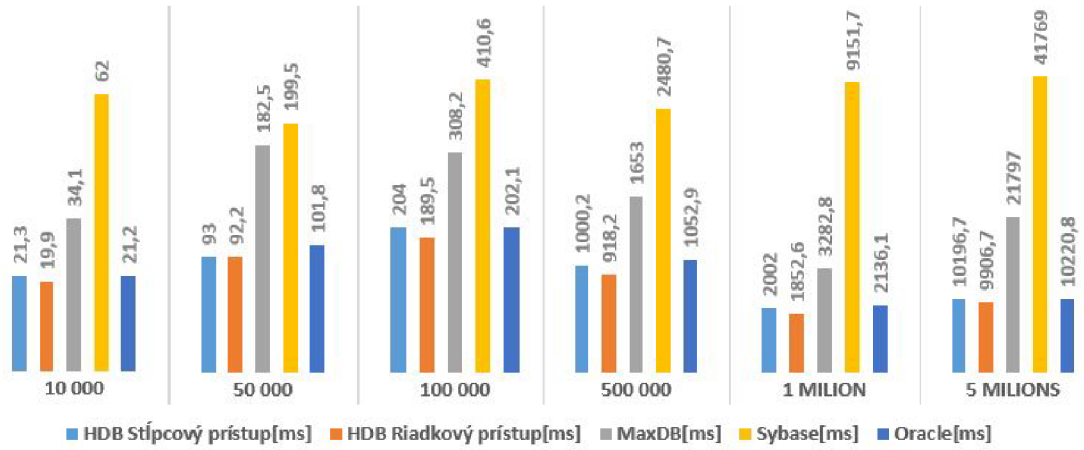
kľúče oboch tabuliek. Potom prebehne FULL TABLE SCAN na tabuľke ZPDCITY a vykoná sa Hašovaný JOIN cez countrycode atribút. Je jasné, že skorším obmedzením množiny kódov krajín, sme urýchlili spojenie s tabuľkou ZPDCITY.

Pri **SAP HANA so stĺpcovým prístupom** sa najprv oskenuje tabuľka ZPDCOUNTRY, potom sa oskenuje tabuľka ZPDPOSITION a aplikuje sa obmedzujúca podmienka. Následne sa oskenuje tabuľka ZPDCITY. Nakoniec sa vykoná JOIN cez všetky tri tabuľky a vráti sa výsledok.

Pri **riadkovom prístupe** sa na začiatku prečíta tabuľka ZPDCOUNTRY. Potom sa pomocou indexového stromu vykoná spojenie JOIN na tabuľkách ZPDCOUNTRY a ZPDLANGUAGES a aplikuje sa podmienka. Následne sa prečíta tabuľka ZPDCITY. Ako posledné sa vykoná Hashovaný JOIN na tabuľke ZPDCITY a na zvyšných spojených tabuľkach. Na záver sa už iba vráti výsledok.

7.7 Test 4

```
SELECT c~id c~name FROM zpdcountry AS co
   JOIN zpdcity AS c ON c~countrycode = co~countrycode
   WHERE continent = 'EU'.
```



Obr. 24: Test 4

Týmto príkazom získavam ID a názov len tých miest, ktoré sa nachádzajú na Európskom kontinente.

Štvrtý test je veľmi podobný tomu predchádzajúcemu. Rozdiel je len v tom, že spájam len dve tabuľky ZPDCITY a ZPDCOUNTRY.

Výsledky z obrázka 24 sú taktiež veľmi podobné tým z tretieho testu. Lenže vďaka tomu, že spájam iba dve tabuľky, rýchlosť **SAP HANA** a **Oracle** sa vyrovnala. **Sybase** je stále najpomalšia a **MaxDB** je len o trochu horšia od prvých.

Exekučné plány 9.1.4 pre prvé tri databázy sa tiež veľmi nelíšia. Zaujímavé je, že **stĺpcový prístup v SAP HANA** databázy aplikuje najprv podmienku na tabuľku ZPDCOUNTRY a potom vykoná obyčajný JOIN. Na rozdiel od toho, **riadkový prístup** v tejto databáze podobne najprv použije obmedzujúcu podmienku na tabuľku ZPDCOUNTRY, ale potom vykoná Hašovaný JOIN. Vďaka tomuto, je riadkový prístup vždy o trochu rýchlejší.

Oracle databáza dvakrát pristupuje k tabuľke ZPDCOUNTRY a kvôli tomu sa výkonnostne trochu zhoršila oproti testu číslo 3.

7.8 Test 5

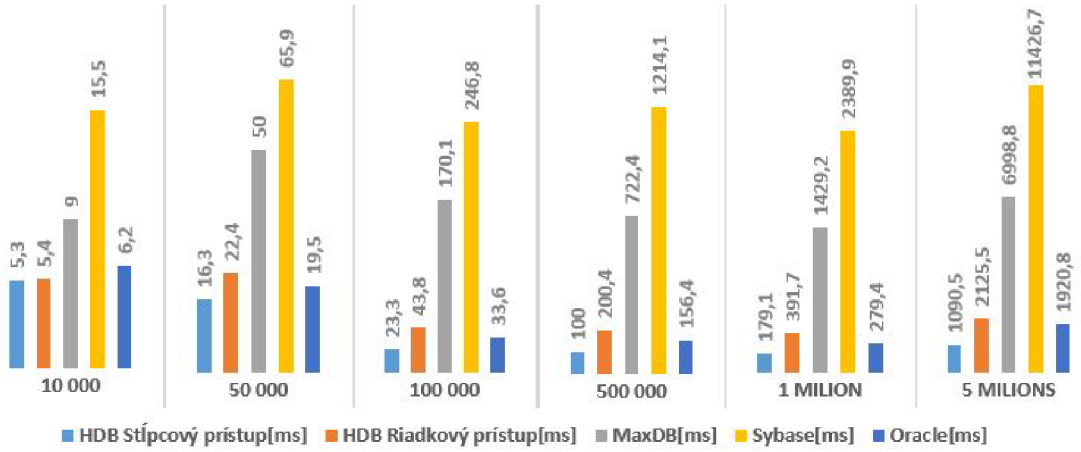
```
SELECT c~id c~name p~latitude p~longitude
   FROM zpdcity AS c JOIN zpdposition AS p ON c~id = p~id
     WHERE latitude BETWEEN 30 AND 570 AND
           longitude BETWEEN 20 AND 40.
```

Vyberám id, názov mesta, zemepisnú šírku a dĺžku len z tých miest, ktoré sa nachádzajú medzi 30 a 570 poludníkom a medzi 20 a 40 rovnobežkou.

V tomto teste opäť spájam dve tabuľky ZPDCITY a ZPDPOSITION, ale teraz v časti WHERE nefiltrujem pomocou hodnoty CHAR alebo pomocou hodnoty INTEGER. Vzhľadom na to, že in-memory databáza pracuje lepšie s číslami, očakával som zlepšenie výkonu u **SAP HANA so stĺpcovým prístupom**, obr. 25.

Pri prechádzaní obrázkom 25 s výsledkami, môžem povedať, že moje očakávanie bolo splnené. **SAP HANA so stĺpcovým prístupom** tento typ príkazu spracovala najrýchlejšie. Druhé najrýchlejšie sú **SAP HANA s riadkovým prí-**

stupom a **Oracle** s veľmi malými rozdielmi. Najpomalšia je **Sybase** databáza.



Obr. 25: Test 5

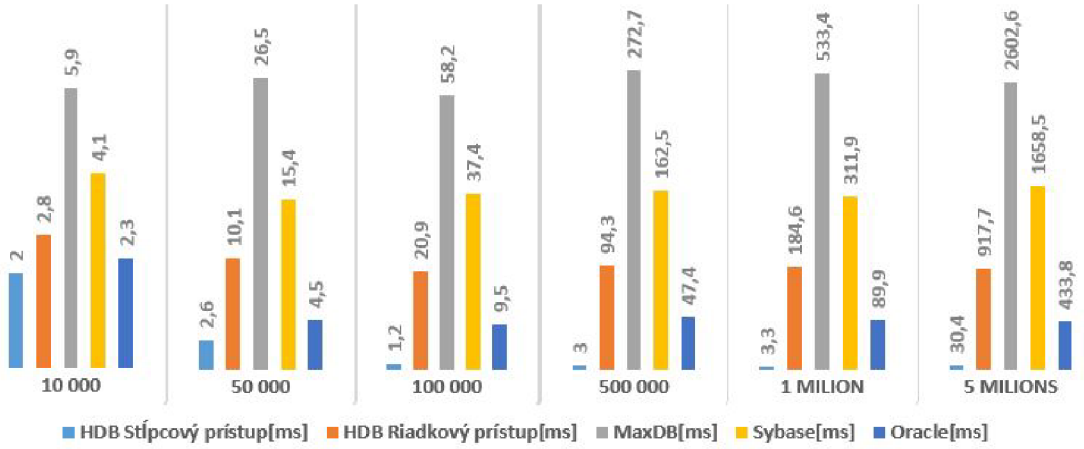
V exekučných plánoch 9.1.5 je viditeľný malý rozdiel. Pri **SAP HANA so stĺpcovým prístupom** sa najprv prečíta tabuľka ZPDCITY, potom sa prečíta spolu s aplikáciou podmienky tabuľka ZPDPOSITION. Následne sa vykoná operácia JOIN na tabuľky ZPDCITY a ZPDPOSITION. Toto spojenie prebieha na primárnych kľúčoch, tak je dostatočne rýchle. Na záver sa už len vráti požadovaný výsledok.

Pri **riadkovom prístupe v SAP HANA** databáze sa podobne ako pri stĺpcovom prístupe najprv prečíta tabuľka ZPDCITY. Potom prebehne spojenie. Prebieha pomocou stromového indexu. Podmienka sa aplikuje zároveň so spojením. Lenže, čo platí aj pre **Oracle**, **MaxDB** a **Sybase** databázu v časti WHERE, tentokrát obmedzujeme výsledok pomocou dvoch atribútov. A pri prístupe až k dvom atribútom bude vždy rýchlejší stĺpcový prístup, lebo môže priamo pristúpiť k daným položkám. Pričom pri riadkovom prístupe vždy prečítame nejaké dáta navyše.

Podľa exekučného plánu **Oracle** databázy si môžeme všimnúť, že pristupuje k tabuľke ZPDCITY znovu dvakrát. Kvôli čomu je výkon zrovnateľný so **SAP HANA s riadkovým prístupom**.

7.9 Test 6

```
SELECT COUNT(DISTINCT name) FROM zpdcity WHERE countrycode = 'CA'.
```



Obr. 26: Test 6

Týmto testom začínam sériu testov na agregačné funkcie. Moje očakávanie je, že pri použití agregačných funkcií by mala byť **SAP HANA so stĺpcovým prístupom** vždy najrýchlejšia.

Použitím agregačnej funkcie COUNT v tomto prípade vraciam počet miest v krajine Kanada. Funkcia je použitá na názov mesta.

Z obrázka 26 vidíme, že podľa očakávania je narýchlejšou **SAP HANA so stĺpcovým prístupom** . Pri 10 000 záznamoch je rozdiel oproti **Oracle** minimálny, ale pri 5 miliónoch je už rozdiel 400 ms. Tretia najrýchlejšia je **SAP HANA s riadkovým prístupom** a najpomalšia je databáza **MaxDB**. Podľa exekučných plánov 9.1.6 **MaxDB** skenuje celú tabuľku a preto jej to trvá najdlhšie.

Vďaka tomu, že funkcia COUNT je použitá len na jeden atribút, je možné plne využiť výhody stĺpcového prístupu. Pri **stĺpcovom prístupe** sa najprv prečíta tabuľka ZPDCITY a vyfiltrujú sa pomocou podmienky len tie mestá, ktoré sa nachádzajú v Kanade. Potom sa aplikuje agregačná funkcia COUNT na atribút NAME. Na koniec sa už len vráti počet miest.

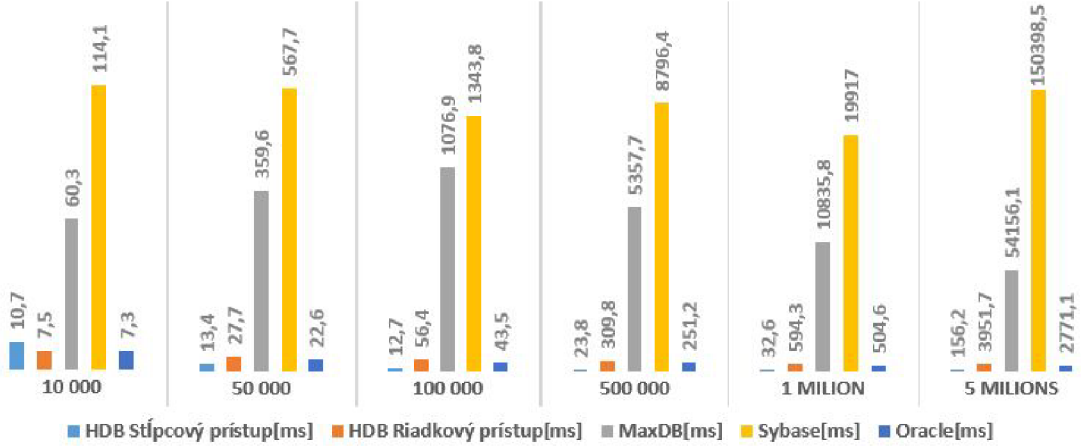
V **riadkovom prístupe SAP HANA** databáze sa vypočítava Agregačná tabuľka, ktorá sa vytvára pri skenovaní tabuľky ZPDCITY. Pri vytváraní agregač-

nej tabuľky sa zároveň aplikuje obmedzujúca podmienka a vykonáva sa funkcia COUNT. Kvôli tomu, že sa vytvára nadbytočná tabuľka v pamäti, tak je tento spôsob pomalší.

Oracle databáza ako prvé oskenuje tabuľku ZPDCITY a aplikuje podmienku. Potom vykoná hašovaný GROUP BY na atribút NAME. Nakoniec zobrazí výsledok.

7.10 Test 7

```
SELECT COUNT(DISTINCT name) FROM zpdcountry AS co
   JOIN zpdcity AS c ON c~countrycode = co~countrycode
   JOIN zpdlanguages AS l ON co~countrycode = l~countrycode
  WHERE lang = 'en'.
```



Obr. 27: Test 7

Tento SELECT príkaz, vráti počet miest z krajín, v ktorých sa hovorí anglickým jazykom.

Oproti predchádzajúcemu testu, ktorý bol vykonaný iba nad jednou tabuľkou, je tento aplikovaný na tri tabuľky spojené príkazom JOIN.

Z obrázka 27 môžeme vidieť, že vykonanie tohto príkazu najrýchlejšie zvládli databázy **SAP HANA** a **Oracle**. S veľkým rozdielom za nimi zaostávajú **MaxDB** a **Sybase**. Pri 10 000 záznamoch je **Oracle** o pár milisekún rýchlejší ako

SAP HANA. Ale pri 5 miliónoch záznamoch je už **SAP HANA so stĺpcovým prístupom** o približne 2600 ms rýchlejšia ako **Oracle**.

Pomocou exekučných plánov 9.1.7 bližšie vysvetlím ako sa vyhodnocuje tento príkaz v SAP HANA a Oracle.

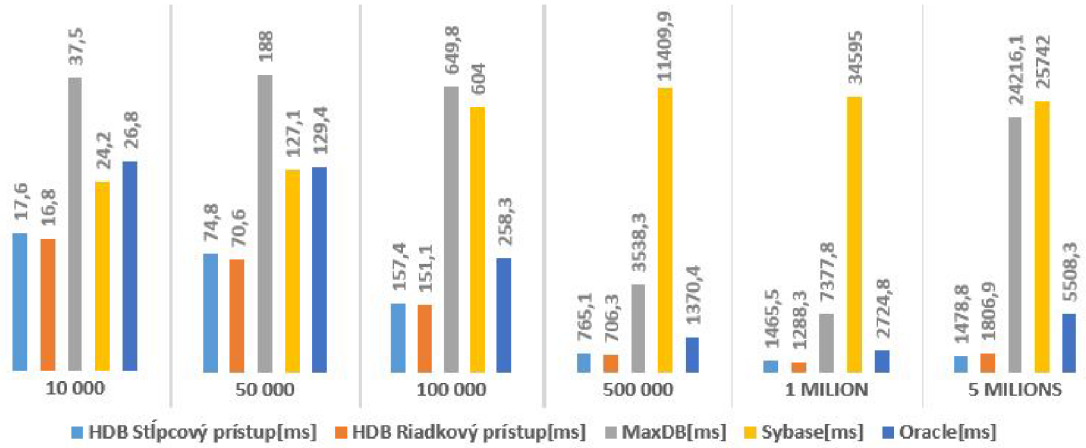
V **SAP HANA so stĺpcovým prístupom** sa najprv aplikuje podmienka na tabuľku ZPDLANGUAGES, potom sa prečíta tabuľka ZPDCOUNTRY a ZPDCITY. Následne sa vykoná JOIN na všetkých troch tabuľkách. Vďaka tomu, že tabuľka ZPDLANGUAGES už bola zmenšená pomocou podmienky, je tento JOIN rýchlejší. Nakoniec sa použije agregačná funkcia COUNT a vráti sa výsledok.

Pri **riadkovom prístupe v SAP HANA** databáze sa najprv prečíta tabuľka ZPDCOUNTRY. Potom sa pomocou stromového indexu spojí s tabuľkou ZPDLANGUAGES a až potom sa aplikuje podmienka. V ďalšom kroku sa prečíta ZPDCITY a spojí sa s výsledkom predchádzajúceho spojenia. Nakoniec sa aplikuje funkcia COUNT a vráti sa výsledok. Vzhľadom na to, že prvé spojenie sa vykonáva pred aplikovaním podmienky, je tento spôsob o niekoľko milisekúnd pomalší.

Oracle databáza najprv pomocou vnorených cyklov spojí tabuľky ZPDCOUNTRY A ZPDLANGUAGES. Ešte predtým ale použije obmedzujúcu podmienku. Vďaka tomuto sa rýchlostne dostane pred **SAP HANA s riadkovým prístupom**. V ďalšom kroku sa prečíta tabuľka ZPDCITY a pomocou HASH JOIN sa spojí s predchádzajúcimi tabuľkami. Následne sa vykoná hašovaný GROUP BY a vráti sa výsledok. Môžeme si všimnúť, že pri menšom počte záznamov sú vnorené cykly a HASH JOIN rýchlejšie. Ale s pribúdajúcimi záznamami sa postupne tento spôsob spomaľuje a opačne **SAP HANA so stĺpcovým prístupom** sa zrýchľuje.

7.11 Test 8

```
SELECT AVG(population) FROM zpdcity WHERE population > 1000
   GROUP BY population.
```



Obr. 28: Test 8

Tento príkaz vráti priemernú populáciu miest, ktoré majú viac ako tisíc obyvateľov. Týmto testom otestujem agregačnú funkciu AVG, aplikovanú na jednu tabuľku. Očakával som, že v tomto teste bude výraznejšie rýchlejšia **SAP HANA** databáza.

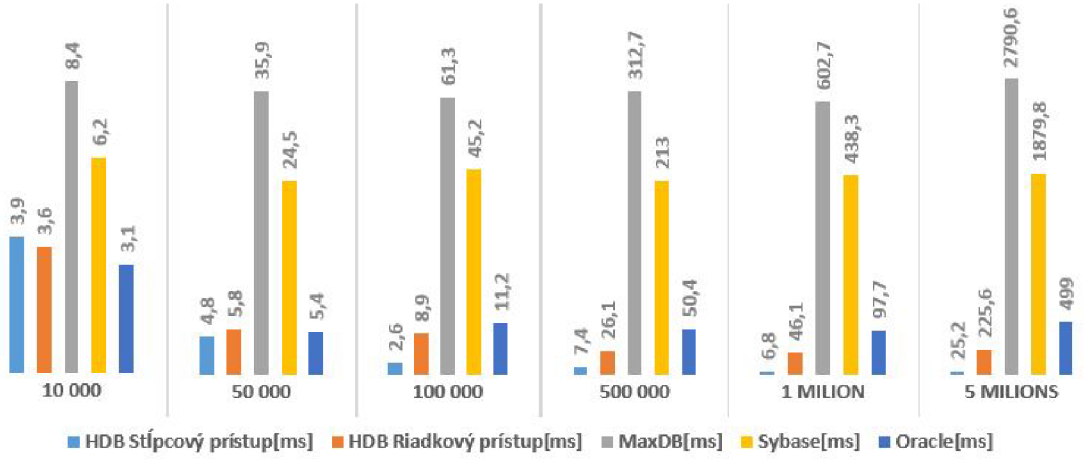
Obrázok 28 potvrdzuje moju hypotézu. **SAP HANA v obidvoch prístupoch** je výrazne rýchlejšia ako druhá, **Oracle** databáza. Najpomalšie sú opäť **Sybase** a **MaxDB**.

V exekučných testoch 9.1.8 **SAP HANA** databázy so **stĺpcovým prístupom** a s **riadkovým prístupom** nie je veľký rozdiel. Rozdiel je iba v spôsobe vyhľadávania populácie a aplikácie podmienky. Stĺpcový prístup vďaka svojim vlastnostiam pristupuje priamo k atribútu POPULATION a filtruje. Oproti tomu, riadkový prístup používa TABLE SCAN a tým pádom načítava aj dáta navyše. Ale aj napriek tomuto malému rozdielu, výkon obidvoch spôsobov je pri tomto teste porovnateľný.

Oracle databáza taktiež používa riadkové usporiadanie a z toho dôvodu najprv prečíta z tabuľky viac informácií ako potrebuje. Najväčší rozdiel pri **Oracle** databáze je použitie hašovaného GROUP BY príkazu, ktorý výkon výrazne spomalí.

7.12 Test 9

```
SELECT AVG( population ) FROM zpdcity AS c
   JOIN zpdtimezones AS t ON c~timezoneid = t~timezoneid
  WHERE population > 1000 AND t~timezoneid = 'Europe/Budapest'
  GROUP BY population.
```



Obr. 29: Test 9

V tomto teste podobne ako v tom predošlom zisťujem priemernú populáciu miest. Ale tentokrát aj v spojení s druhou tabuľkou ZPDTIMEZONES.

Pri tomto teste sa rýchlosť **SAP HANA** databázy najviac odvíjala od počtu záznamov v tabuľke. Ako môžeme vidieť na obrázku 29, pri 10 000 záznamoch bola rýchlejšia databáza **Oracle**, ale pri 5 miliónoch záznamoch bola najrýchlejšia **SAP HANA so stĺpcovým prístupom**. Pre porovnanie **SAP HANA so stĺpcovým prístupom** išla z 3,9 ms na 25,2ms. **Oracle** sa zhoršil niekoľkonásobne z 3,1 ms na 499 ms. Zaujímavé je, že pri tomto teste, kde sa pristupuje k dvom tabuľkám, sú všetky databázy rýchlejšie oproti testu č.8, kde sa pristupovalo iba k jednej tabuľke. Môže to byť spôsobené faktom, že pomocou druhej pridanej podmienky, zmenšujeme tabuľku, na ktorú sa aplikuje agregačná funkcia. Čo má za následok lepší výkon.

V exekučnom pláne 9.1.9 pre **SAP HANA** databázu so **stĺpcovým prístupom** nenastala výraznejšia zmena. Najprv sa aplikuje podmienka na jednu

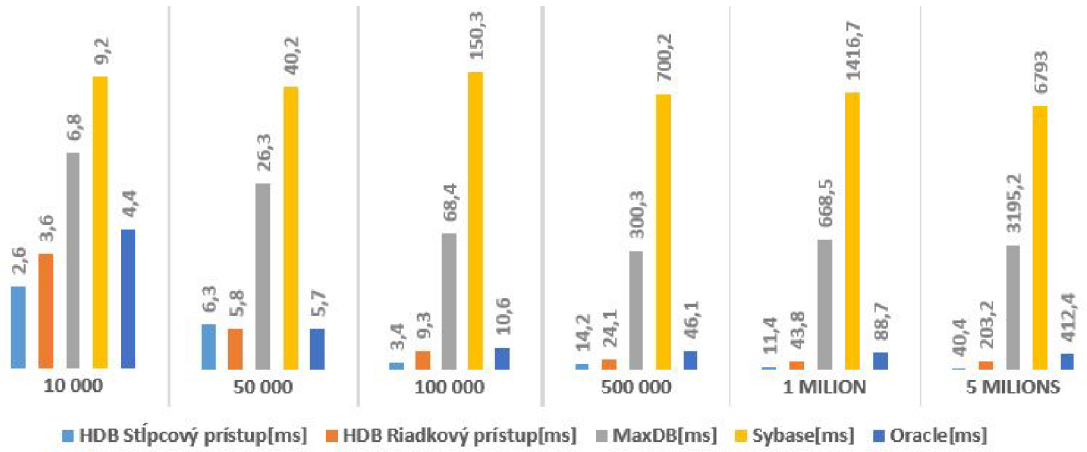
tabuľku, potom sa aplikujú obe podmienky na tabuľku ZDPCITY. Následne sa vykoná operácia JOIN a aplikuje sa funkcia AVG.

Pri **riadkovom prístupe v SAP HANA** nastala zmena. Najprv sa aplikuje stromové indexové vyhľadávanie s aplikáciou podmienky na tabuľku ZPDTIMEZONES. Potom sa prečíta tabuľka ZPDCITY s aplikovaním oboch podmienok. Spojenie týchto dvoch tabuliek prebehne pomocou vnorených cyklov, čo vidíme prvýkrat pri tejto databáze. Nakoniec sa aplikuje agregačná funkcia. Práve vnorené cykly a stromové vyhľadávanie zapríčinili spomalenie pri väčšom množstve dát.

Oracle databáza vykonala tento príkaz podobným spôsobom ako SAP HANA s riadkovým prístupom. S jedným rozdielom, že nepoužila stromové vyhľadávanie, čo malo za následok horší výkon ako spomenutá databáza.

7.13 Test 10

```
SELECT SUM( DISTINCT co~area ) FROM zpdcity AS c
   JOIN zpdcountry AS co ON co~countrycode = c~countrycode
   JOIN zpdposition AS p ON c~id = p~id
  WHERE (latitude BETWEEN 30 AND 40) AND (longitude BETWEEN 20 AND 40)
        AND continent = 'EU'.
```



Obr. 30: Test 10

V poslednom teste som sa rozhodol otestovať funkciu SUM. Aplikujem ju na spojenie troch tabuliek ZPDCITY, ZPDCOUNTRY a ZPDPOSITION.

Aj v poslednom teste agregačných funkcií vidno, že použitie in-memory databázy so stĺpcovým prístupom je najlepšie riešenie. Na obrázku 30 si môžeme všimnúť, že pri všetkých množinách záznamov (okrem jednej) je **SAP HANA so stĺpcovým prístupom** najrýchlejšia. Pomer spomalenia s pribúdajúcim počtom záznamov bol pri zvyšných databázach niekoľko-krát väčší ako pri **SAP HANA** databáze.

V exekučných plánoch 9.1.10 môžeme vidieť dôvod, prečo je **SAP HANA so stĺpcovým prístupom** rýchlejšia. Najväčší rozdiel je v počte JOIN operácií. Napríklad **SAP HANA so stĺpcovým prístupom** vykoná iba jednu JOIN operáciu. **HANA s riadkovým prístupom** vykoná dve JOIN operácie a **Oracle** až trikrát použije na spojenie vnorené cykly.

7.14 Vyhodnotenie testov

Po vykonaní všetkých testov môžem potvrdiť svoju hypotézu z kapitoly 7. Ako som predpokladal SAP HANA bola v priemere vo väčšine testov najrýchlejšia. Jediné v čom som sa zmýlil je, že som očakával jednoznačný rozdiel medzi SAP HANA a zvyšnými databázami. Jedným z dôvodov prečo SAP HANA nebola výraznejšie rýchlejšia je použitie interných systémov. Tieto systémy nemajú takú výpočetnú silu a hlavne veľkosť a rýchlosť vnútornej pamäte, ako sa používa priamo na zákazníckych systémoch. Tým pádom výhody uloženia dát priamo vo vnútornej pamäti nemohli byť naplno využité.

Čo sa týka veľkosti databáz, ukázal som, ako dokáže slovníkové kódovanie a kompresné techniky pri stĺpcovom prístupe zmenšiť celkovú veľkosť databázovej tabuľky. Bolo zaujímavé pozorovať ako veľkosť SAP HANA databázy so stĺpcovým prístupom závisí od počtu a typu záznamov v tabuľke. Keď bolo záznamov ešte príliš málo na využitie plného potencionálu kompresných techník a slovníkového kódovania, tak veľkosť databázy bola najväčšia. No pri 5 miliónoch záznamoch, keď už bolo v tabuľke dostatok dát na využitie výhod kompresných techník, bola jej veľkosť už výrazne najmenšia.

Pri testovaní agregačných funkcií bola vždy rýchlejšia SAP HANA so stĺpcovým prístupom. Hlavným dôvodom bol fakt, že bola schopná pristupovať priamo k potrebným stĺpcom, na rozdiel od databáz s riadkovým prístupom, ktoré si vždy museli načítať aj nadbytočné dáta. Pri tomto type testov bolo taktiež zaujímavé sledovať ako počet záznamov ovplyvňuje rýchlosť jednotlivých databáz. Pri malom počte záznamov nebola SAP HANA so stĺpcovým prístupom vždy najrýchlejšia. Ale stúpajúci počet záznamov mal slabší efekt na spomalenia tejto databázy ako zvyšných databáz, a tak pri veľkej množine záznamov, bola vždy najrýchlejšia.

Pri testovaní rekonštrukcie záznamov z tabuliek, bol vždy rýchlejší riadkový prístup. Či už pri SAP HANA alebo Oracle databáze. Jedine pri teste č.5 (7.8) bola rýchlejšia SAP HANA so stĺpcovým prístupom. Ako som popísal v danom teste, bolo to spôsobené použitím dvoch atribútov vo WHERE časti príkazu a navyše boli typu Integer.

8 Záver

SAP HANA a celkovo In-Memory Data Management je bezpochyby budúcnosťou podnikových systémov. Aj keď vzhľadom k výsledkom môjho testovania, SAP HANA nebola až tak výrazne rýchlejšia. Ale pri použití agregačných funkcií, ktoré sa najviac používajú na analýzu dát, bola jednoznačne najrýchlejšia.

V mojich testoch som použil len jednoduché SELECT príkazy na obmedzenom počte dát, ale pri praktickom použití v reálnych aplikáciach a nad oveľa väčšou množinou dát, sa výhody in-memory prístupu a stĺpcového ukladania dát dostávajú ešte viac do popredia. Dôležité je, uvedomiť si, že nestačí len zmeniť tradičnú relačnú databázu za in-memory databázu, ale je k tomu potrebné prispôsobiť aj kód aplikácie.

Pre príklad uvediem, že zákazníci spoločnosti SAP po prechode na SAP HANA databázový systém pocítili razantné zrýchlenie aplikácií (presnejšie celkový čas bol zredukovaný o 20%), ale vzhľadom k predpokladom neboli spokojní. Bolo potrebné optimalizovať kód aplikácie, aby maximálne využil výhody uloženia dát v pamäti a rýchleho prístupu k nim. Po optimalizácii kódu, sa o niečo zvýšil celkový čas prístupu k databáze, ale čas spracovania dát v aplikácii sa skrátil. Vďaka tomu sa celkový čas zredukoval skoro až o 80%.

Literatúra

[1] PLATTNER, Hasso. *A Course in In-Memory Data Management.* Springer, 2013. 297 s. ISBN 978-3-642-36523-2.

[2] LACKO, Ľuboslav. *Databáze: datové sklady, OLAP a dolování dat.* Computer Press, 2003. 486 s. ISBN 80-7226-969-0.

[3] POKORNÝ, Jaroslav. *Databázové systémy 2.* Vyd. 1. V Praze: České vysoké učení technické, 2007. 190 s. ISBN 9788001037973.

[4] *Column Vs Row Data Storage* [online], 2014, [cit. 18. 05. 2016], SapHanaTutorial. Dostupné z: <`http://saphanatutorial.com/column-data-storage-and-row-data-storage-sap-hana/`>.

[5] BERG, Bjarne and PENNY, Silvia. *SAP HANA: An Introduction.* SAP PRESS, 2012. 426 s. ISBN 978-1-59229-434-3.

[6] LONEY, Kevin. *Oracle Database.* Computer Press, 2010. 1368 s. ISBN 9788025124895.

[7] *Microsoft SQL Server* [online]. aktualizácia 3. 2. 2016, [cit. 18. 5. 2016] Dostupné z: < `https://cs.wikipedia.org/wiki/Microsoft_SQL_Server` >.

[8] *MaxDB* [online]. aktualizácia 18. 4. 2016, [cit. 18. 5. 2016] Dostupné z: < `https://en.wikipedia.org/wiki/MaxDB` >.

[9] BÖGELSACK Andre, GRADL Stephan, MAYER Manuel and KRCMAR Helmut. *SAP MaxDB Administration.* Bonn: Galileo Press, 2009. 326 s. ISBN 978-1-59229-299-8.

[10] FÄRBER, Günther and KIRCHNER Julia. *ABAP Basics 2nd Edition.* Boston: Galileo Press, 2010. 508 s. ISBN 978-1592293698.

[11] *GeoNames* [online]. [cit. 18. 5. 2016] Dostupné z: < `http://www.geonames.org/` >.

[12] WEGELIN, Michael and ENGLBRECHT, Michael. *SAP Interface Programming.* Boston: Galileo Press, 2010. 405 s. ISBN 978-1-59229-661-3.

9 Príloha

9.1 Exekučné plány

9.1.1 Test 1

```
SELECT * FROM zpdcity.
```

Operation	Options	Object_name
COLUMN SEARCH	ZPDCITYC.ID, ZPDCITYC.NAME, ZPDCITYC.COUNTRYCODE, ZPDCITYC.POPULATION, ZPDCITYC.ELEVATION, ZPDCITYC.MODIFDATE, ZPDCITYC.TIMEZONEID (LATE MATERIALIZATION, OLTP SEARCH, ENUM_BY: CS_TABLE)	
COLUMN TABLE		ZPDCITYC

Obr. 31: Exekučný plán - Hana DB stĺpcový prístup

Operation	Options	Object_name
ROW SEARCH	ZPDCITY.ID, ZPDCITY.NAME, ZPDCITY.COUNTRYCODE, ZPDCITY.POPULATION, ZPDCITY.ELEVATION, ZPDCITY.MODIFDATE, ZPDCITY.TIMEZONEID	
TABLE SCAN		ZPDCITY

Obr. 32: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 20 , Estimated #Rows = 0 )
  └──1 TABLE ACCESS FULL ZPDCITY
       ( Estim. Costs = 19 , Estim. #Rows = 10.669 )
       Estim. CPU-Costs = 3.557.167 Estim. IO-Costs = 19
```

Obr. 33: Exekučný plán - Oracle

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
SAPZKU	ZPDCITY		TABLE SCAN	167
			RESULT IS NOT COPIED , COSTVALUE IS	167
	SHOW		QUERYREWRITE : APPLIED RULES:	
	SHOW		DistinctPullUp	1

Obr. 34: Exekučný plán - MaxDB

9.1.2 Test 2

```
SELECT id name FROM zpdcity.
```

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	ZPDCITYC.ID, ZPDCITYC.NAME (LATE MATERIALIZATION, OLTP SEARCH, ENUM_BY: CS_TABLE)	
COLUMN TABLE		ZPDCITYC

Obr. 35: Exekučný plán - Hana DB stĺpcový prístup

Execution Plan

Operation	Options	Object_name
ROW SEARCH	ZPDCITY.ID, ZPDCITY.NAME	
TABLE SCAN		ZPDCITY

Obr. 36: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 20 , Estimated #Rows = 0 )
  └──1 TABLE ACCESS FULL ZPDCITY
       ( Estim. Costs = 19 , Estim. #Rows = 10.669 )
       Estim. CPU-Costs = 2.490.267 Estim. IO-Costs = 19
```

Obr. 37: Exekučný plán - Oracle

Execution Plan

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
SAPZKU	ZPDCITY		TABLE SCAN	167
			RESULT IS NOT COPIED , COSTVALUE IS	167
	SHOW		QUERYREWRITE : APPLIED RULES:	
	SHOW		DistinctPullUp	1

Obr. 38: Exekučný plán - MaxDB

9.1.3 Test 3

```
SELECT c~id c~name FROM zpdcountry AS co
   JOIN zpdcity AS c ON c~countrycode = co~countrycode
   JOIN zpdlanguages AS l ON co~countrycode = l~countrycode
```

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	C.ID, C.NAME (LATE MATERIALIZATION, OLTP SEARCH, ENUM_BY: CS_JOIN)	
JOIN	JOIN CONDITION: (INNER) C.COUNTRYCODE = L.COUNTRYCODE, (INNER) L.COUNTRYCODE = CO.COUNTRYCODE	
COLUMN TABLE		ZPDCITYC
COLUMN TABLE	FILTER CONDITION: L.LANG = n'en'	ZPDLANGUAGESC
COLUMN TABLE		ZPDCOUNTRYC

Obr. 39: Exekučný plán - Hana DB stĺpcový prístup

Execution Plan

Operation	Options	Object_name
ROW SEARCH	C.ID, C.NAME	
HASH JOIN	MANY-TO-ONE HASH BUILD: RIGHT, JOIN CONDITION: C.COUNTRYCODE = CO.COUNTRYCODE	
TABLE SCAN		ZPDCITY
CPBTREE INDEX JOIN	INDEX NAME: _SYS_TREE_RS_#5566906_#0_#P0, INDEX CONDITION: CO.COUNTRYCODE = L.COUNTRYCODE AND 'en' = L.LANG (ENUM_BY: INDEX_JOIN)	ZPDLANGUAGES
TABLE SCAN		ZPDCOUNTRY

Obr. 40: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 20 , Estimated #Rows = 0 )
└─ 5 HASH JOIN
     ( Estim. Costs = 20 , Estim. #Rows = 5.292 )
     Estim. CPU-Costs = 4.443.681 Estim. IO-Costs = 19
     Memory Used KB: 1.613.824
     Access Predicates
   ├─ 3 NESTED LOOPS
        Estim. CPU-Costs = 54.384 Estim. IO-Costs = 0
        ├─ 1 INDEX FULL SCAN ZPDCOUNTRY~0
             Estim. CPU-Costs = 1.464 Estim. IO-Costs = 0
        └─ 2 INDEX UNIQUE SCAN ZPDLANGUAGES~0
             Search Columns: 2
             Estim. CPU-Costs = 210 Estim. IO-Costs = 0
             Access Predicates
   └─ 4 TABLE ACCESS FULL ZPDCITY
        ( Estim. Costs = 19 , Estim. #Rows = 10.669 )
        Estim. CPU-Costs = 2.703.647 Estim. IO-Costs = 19
```

Obr. 41: Exekučný plán - Oracle

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
	C		TABLE SCAN	167
	L		JOIN VIA MULTIPLE KEY COLUMNS	2
			TABLE HASHED	
		COUNTRYCODE	(USED KEY COLUMN)	
		LANG	(USED KEY COLUMN)	
	CO	COUNTRYCODE	JOIN VIA KEY COLUMN	3
			TABLE HASHED	
			NO TEMPORARY RESULTS CREATED	
			RESULT IS COPIED , COSTVALUE IS	1503
	SHOW		QUERYREWRITE : APPLIED RULES:	
	SHOW		PushDownPredicates	1

Obr. 42: Exekučný plán - MaxDB

9.1.4 Test 4

```
SELECT c~id c~name FROM zpdcountry AS co
   JOIN zpdcity AS c ON c~countrycode = co~countrycode
   WHERE continent = 'EU'.
```

Operation	Options	Object_name
COLUMN SEARCH	C.ID, C.NAME (LATE MATERIALIZATION, OLTP SEARCH, ENUM_BY: CS_JOIN)	
JOIN	JOIN CONDITION: (INNER) C.COUNTRYCODE = CO.COUNTRYCODE	
COLUMN TABLE		ZPDCITYC
COLUMN TABLE	FILTER CONDITION: CO.CONTINENT = n'EU'	ZPDCOUNTRYC

Obr. 43: Exekučný plán - Hana DB stĺpcový prístup

Execution Plan

Operation	Options	Object_name
ROW SEARCH	C.ID, C.NAME	
HASH JOIN	HASH BUILD: RIGHT, JOIN CONDITION: C.COUNTRYCODE = CO.COUNTRYCODE	
TABLE SCAN		ZPDCITY
TABLE SCAN	FILTER CONDITION: CO.CONTINENT = 'EU'	ZPDCOUNTRY

Obr. 44: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 20 , Estimated #Rows = 0 )
└─ 5 NESTED LOOPS
     ( Estim. Costs = 20 , Estim. #Rows = 3.841 )
     Estim. CPU-Costs = 5.584.277 Estim. IO-Costs = 19
   ├─ 3 NESTED LOOPS
        ( Estim. Costs = 20 , Estim. #Rows = 10.669 )
        Estim. CPU-Costs = 5.584.277 Estim. IO-Costs = 19
      ├─ 1 TABLE ACCESS FULL ZPDCITY
           ( Estim. Costs = 19 , Estim. #Rows = 10.669 )
           Estim. CPU-Costs = 2.703.647 Estim. IO-Costs = 19
      └─ 2 INDEX UNIQUE SCAN ZPDCOUNTRY~0
           Search Columns: 1
           Estim. CPU-Costs = 210 Estim. IO-Costs = 0
           Access Predicates
   └─ 4 TABLE ACCESS BY INDEX ROWID ZPDCOUNTRY
        Estim. CPU-Costs = 270 Estim. IO-Costs = 0
        Filter Predicates
```

Obr. 45: Exekučný plán - Oracle

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
	C		TABLE SCAN	167
	CO	COUNTRYCODE	JOIN VIA KEY COLUMN	3
			TABLE HASHED	
			NO TEMPORARY RESULTS CREATED	
			RESULT IS COPIED , COSTVALUE IS	941
	SHOW		QUERYREWRITE : APPLIED RULES:	
	SHOW		PushDownPredicates	1

Obr. 46: Exekučný plán - MaxDB

9.1.5 Test 5

```
SELECT c~id c~name p~latitude p~longitude
   FROM zpdcity AS c JOIN zpdposition AS p ON c~id = p~id
     WHERE latitude BETWEEN 30 AND 570 AND
           longitude BETWEEN 20 AND 40.
```

Operation	Options	Object_name
COLUMN SEARCH	C.ID, C.NAME, P.LATITUDE, P.LONGITUDE (LATE MATERIALIZATION, OLTP SEARCH, ENUM_BY: CS_JOIN)	
JOIN	JOIN CONDITION: (INNER) P.ID = C.ID	
COLUMN TABLE	FILTER CONDITION: P.LONGITUDE >= '20' AND P.LATITUDE >= '30' AND P.LONGITUDE <= '40' AND P.LATITUDE <= '570'	ZPDPOSITIONC
COLUMN TABLE		ZPDCITYC

Obr. 47: Exekučný plán - Hana DB stĺpcový prístup

Operation	Options	Object_name
ROW SEARCH	C.ID, C.NAME, P.LATITUDE, P.LONGITUDE	
BTREE INDEX JOIN	INDEX NAME: _SYS_TREE_RS_#5567044_#0_#P0, INDEX CONDITION: C.ID = P.ID, JOIN CONDITION: , INDEX FILTER: '20' <= P.LONGITUDE AND '30' <= P.LATITUDE AND '40' >= P.LONGITUDE AND '570' >= P.LATITUDE (ENUM_BY: INDEX_JOIN)	ZPDPOSITION
TABLE SCAN		ZPDCITY

Obr. 48: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 9 , Estimated #Rows = 0 )
└─ 6 FILTER
     Filter Predicates
   └─ 5 NESTED LOOPS
        ( Estim. Costs = 8 , Estim. #Rows = 1 )
        Estim. CPU-Costs = 2.680.902 Estim. IO-Costs = 8
      ├─ 3 NESTED LOOPS
           ( Estim. Costs = 8 , Estim. #Rows = 1 )
           Estim. CPU-Costs = 2.680.902 Estim. IO-Costs = 8
         ├─ 1 TABLE ACCESS FULL ZPDPOSITION
              ( Estim. Costs = 8 , Estim. #Rows = 1 )
              Estim. CPU-Costs = 2.680.658 Estim. IO-Costs = 8
              Filter Predicates
         └─ 2 INDEX UNIQUE SCAN ZPDCITY~0
              Search Columns: 1
              Estim. CPU-Costs = 210 Estim. IO-Costs = 0
              Access Predicates
      └─ 4 TABLE ACCESS BY INDEX ROWID ZPDCITY
           Estim. CPU-Costs = 244 Estim. IO-Costs = 0
```

Obr. 49: Exekučný plán - Oracle

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
	P		TABLE SCAN	37
	C	ID	JOIN VIA KEY COLUMN	167
			NO TEMPORARY RESULTS CREATED	
			RESULT IS COPIED , COSTVALUE IS	972
	SHOW		QUERYREWRITE : APPLIED RULES:	
	SHOW		PushDownPredicates	1

Obr. 50: Exekučný plán - MaxDB

9.1.6 Test 6

```
SELECT COUNT(DISTINCT name) FROM zpdcity WHERE countrycode = 'CA'.
```

Operation	Options	Object_name
COLUMN SEARCH	COUNT(DISTINCT ZPDCITYC.NAME) (LATE MATERIALIZATION, OLTP SEARCH, ENUM_BY: CS_TABLE)	
AGGREGATION	AGGREGATION: COUNT(DISTINCT ZPDCITYC.NAME)	
COLUMN TABLE	FILTER CONDITION: ZPDCITYC.COUNTRYCODE = 'CA'	ZPDCITYC

Obr. 51: Exekučný plán - Hana DB stĺpcový prístup

Operation	Options	Object_name
ROW SEARCH	COUNT(DISTINCT ZPDCITY.NAME)	
AGGR TABLE	AGGREGATION: COUNT(DISTINCT ZPDCITY.NAME), FILTER CONDITION: ZPDCITY.COUNTRYCODE = 'CA' (ENUM_BY: AGGR_INTO_TABLE)	ZPDCITY

Obr. 52: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 21 , Estimated #Rows = 0 )
  4 SORT AGGREGATE
    3 VIEW VW_DAG_0
        ( Estim. Costs = 21 , Estim. #Rows = 247 )
        Estim. CPU-Costs = 11.819.680 Estim. IO-Costs = 19
      2 HASH GROUP BY
          ( Estim. Costs = 20 , Estim. #Rows = 247 )
          Estim. CPU-Costs = 11.819.680 Estim. IO-Costs = 19
          Memory Used KB: 6.746.112
        1 TABLE ACCESS FULL ZPDCITY
            ( Estim. Costs = 19 , Estim. #Rows = 247 )
            Estim. CPU-Costs = 3.237.079 Estim. IO-Costs = 19
            Filter Predicates
```

Obr. 53: Exekučný plán - Oracle

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
SAPZKU	ZPDCITY		TABLE SCAN RESULT IS COPIED , COSTVALUE IS	167 169

Obr. 54: Exekučný plán - MaxDB

9.1.7 Test 7

```
SELECT COUNT(DISTINCT name) FROM zpdcountry AS co
   JOIN zpdcity AS c ON c~countrycode = co~countrycode
   JOIN zpdlanguages AS l ON co~countrycode = l~countrycode
  WHERE lang = 'en'.
```

Operation	Options	Object_name
COLUMN SEARCH	COUNT(DISTINCT C.NAME) (LATE MATERIALIZATION, ENUM_BY: CS_JOIN)	
AGGREGATION	AGGREGATION: COUNT(DISTINCT C.NAME)	
JOIN	JOIN CONDITION: (INNER) CO.COUNTRYCODE = L.COUNTRYCODE, (INNER) C.COUNTRYCODE = CO.COUNTRYCODE	
COLUMN TABLE	[FACT]	ZPDCITYC
COLUMN TABLE		ZPDCOUNTRYC
COLUMN TABLE	FILTER CONDITION: L.LANG = n'en'	ZPDLANGUAGESC

Obr. 55: Exekučný plán - Hana DB stĺpcový prístup

Operation	Options	Object_name
ROW SEARCH	COUNT(DISTINCT C.NAME)	
AGGREGATION	AGGREGATION: COUNT(DISTINCT C.NAME)	
HASH JOIN	MANY-TO-ONE HASH BUILD: RIGHT, JOIN CONDITION: C.COUNTRYCODE = CO.COUNTRYCODE	
TABLE SCAN		ZPDCITY
CPBTREE INDEX JOIN	INDEX NAME: _SYS_TREE_RS_#5566906_#0_#P0, INDEX CONDITION: CO.COUNTRYCODE = L.COUNTRYCODE AND 'en' = L.LANG (ENUM_BY: INDEX_JOIN)	ZPDLANGUAGES
TABLE SCAN		ZPDCOUNTRY

Obr. 56: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 22 , Estimated #Rows = 0 )
└─ 8 SORT AGGREGATE
   └─ 7 VIEW VW_DAG_0
        ( Estim. Costs = 22 , Estim. #Rows = 4.488 )
        Estim. CPU-Costs = 15.388.099 Estim. IO-Costs = 19
      └─ 6 HASH GROUP BY
           ( Estim. Costs = 21 , Estim. #Rows = 4.488 )
           Estim. CPU-Costs = 15.388.099 Estim. IO-Costs = 19
           Memory Used KB: 89.604.096
         └─ 5 HASH JOIN
              ( Estim. Costs = 20 , Estim. #Rows = 4.488 )
              Estim. CPU-Costs = 4.440.831 Estim. IO-Costs = 19
              Memory Used KB: 1.589.248
              Access Predicates
            ├─ 3 NESTED LOOPS SEMI
                 Estim. CPU-Costs = 54.384 Estim. IO-Costs = 0
               ├─ 1 INDEX FULL SCAN ZPDCOUNTRY~0
                    Estim. CPU-Costs = 1.464 Estim. IO-Costs = 0
               └─ 2 INDEX UNIQUE SCAN ZPDLANGUAGES~0
                    Search Columns: 2
                    Estim. CPU-Costs = 210 Estim. IO-Costs = 0
                    Access Predicates
            └─ 4 TABLE ACCESS FULL ZPDCITY
                 ( Estim. Costs = 19 , Estim. #Rows = 10.669 )
                 Estim. CPU-Costs = 2.703.647 Estim. IO-Costs = 19
```

Obr. 57: Exekučný plán - Oracle

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
	C		TABLE SCAN	167
	L		JOIN VIA MULTIPLE KEY COLUMNS	2
			TABLE HASHED	
		COUNTRYCODE	(USED KEY COLUMN)	
		LANG	(USED KEY COLUMN)	
	CO	COUNTRYCODE	JOIN VIA KEY COLUMN	3
			TABLE HASHED	
			NO TEMPORARY RESULTS CREATED	
			RESULT IS COPIED , COSTVALUE IS	1463
	SHOW		QUERYREWRITE : APPLIED RULES:	
	SHOW		PushDownPredicates	1

Obr. 58: Exekučný plán - MaxDB

9.1.8 Test 8

```
SELECT AVG(population) FROM zpdcity WHERE population > 1000
   GROUP BY population.
```

Operation	Options	Object_name
COLUMN SEARCH	AVG(TO_DECIMAL(ZPDCITYC.POPULATION)) (LATE MATERIALIZATION, OLTP SEARCH, ENUM_BY: CS_TABLE)	
AGGREGATION	GROUPING: ZPDCITYC.POPULATION, AGGREGATION: AVG(TO_DECIMAL(ZPDCITYC.POPULATION))	
COLUMN TABLE	FILTER CONDITION: ZPDCITYC.POPULATION > 1000	ZPDCITYC

Obr. 59: Exekučný plán - Hana DB stĺpcový prístup

Operation	Options	Object_name
ROW SEARCH	AVG(TO_DECIMAL(ZPDCITY.POPULATION))	
AGGREGATION	GROUPING: ZPDCITY.POPULATION, AGGREGATION: AVG(TO_DECIMAL(ZPDCITY.POPULATION))	
TABLE SCAN	FILTER CONDITION: ZPDCITY.POPULATION > 1000	ZPDCITY

Obr. 60: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 22 , Estimated #Rows = 0 )
  2 HASH GROUP BY
    ( Estim. Costs = 21 , Estim. #Rows = 10.625 )
    Estim. CPU-Costs = 18.347.367 Estim. IO-Costs = 19
    Memory Used KB: 71.545.856
      1 TABLE ACCESS FULL ZPDCITY
        ( Estim. Costs = 19 , Estim. #Rows = 10.625 )
        Estim. CPU-Costs = 3.450.459 Estim. IO-Costs = 19
        Filter Predicates
```

Obr. 61: Exekučný plán - Oracle

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
SAPZKU	ZPDCITY		TABLE SCAN	167
			RESULT IS COPIED , COSTVALUE IS	231

Obr. 62: Exekučný plán - MaxDB

9.1.9 Test 9

```
SELECT AVG( population ) FROM zpdcity AS c
   JOIN zpdtimezones AS t ON c~timezoneid = t~timezoneid
  WHERE population > 1000 AND t~timezoneid = 'Europe/Budapest'
  GROUP BY population.
```

Operation	Options	Object_name
COLUMN SEARCH	AVG(TO_DECIMAL(C.POPULATION)) (LATE MATERIALIZATION, OLTP SEARCH, ENUM_BY: CS_JOIN)	
AGGREGATION	GROUPING: C.POPULATION, AGGREGATION: AVG(TO_DECIMAL(C.POPULATION))	
JOIN	JOIN CONDITION: (INNER) ZPDCITYC x ZPDTIMEZONESC	
COLUMN TABLE	FILTER CONDITION: C.TIMEZONEID = n'Europe/Budapest' AND C.POPULATION > 1000	ZPDCITYC
COLUMN TABLE	FILTER CONDITION: T.TIMEZONEID = n'Europe/Budapest'	ZPDTIMEZONESC

Obr. 63: Exekučný plán - Hana DB stĺpcový prístup

Operation	Options	Object_name
ROW SEARCH	AVG(TO_DECIMAL(C.POPULATION))	
AGGREGATION	GROUPING: C.POPULATION, AGGREGATION: AVG(TO_DECIMAL(C.POPULATION))	
NESTED LOOP JOIN	MANY-TO-ONE	
TABLE SCAN	FILTER CONDITION: C.TIMEZONEID = 'Europe/Budapest' AND C.POPULATION > 1000	ZPDCITY
CPBTREE INDEX SEARCH	INDEX NAME: _SYS_TREE_RS_#5566919_#0_#P0, INDEX CONDITION: T.TIMEZONEID = 'Europe/Budapest' (ENUM_BY: INDEX_SEARCH)	ZPDTIMEZONES

Obr. 64: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 21 , Estimated #Rows = 0 )
└─ 4 HASH GROUP BY
     ( Estim. Costs = 20 , Estim. #Rows = 32 )
     Estim. CPU-Costs = 12.597.500 Estim. IO-Costs = 19
     Memory Used KB: 1.471.488
   └─ 3 NESTED LOOPS
        ( Estim. Costs = 19 , Estim. #Rows = 32 )
        Estim. CPU-Costs = 4.096.144 Estim. IO-Costs = 19
        ├─ 1 INDEX UNIQUE SCAN ZPDTIMEZONES~0
        │    Search Columns: 1
        │    Estim. CPU-Costs = 210 Estim. IO-Costs = 0
        │    Access Predicates
        └─ 2 TABLE ACCESS FULL ZPDCITY
             ( Estim. Costs = 19 , Estim. #Rows = 32 )
             Estim. CPU-Costs = 4.095.934 Estim. IO-Costs = 19
             Filter Predicates
```

Obr. 65: Exekučný plán - Oracle

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
	T		EQUAL CONDITION FOR KEY	4
		TIMEZONEID	(USED KEY COLUMN)	
	C		JOIN VIA KEY RANGE	167
			TABLE TEMPORARY SORTED	
		TIMEZONEID	(USED COLUMN)	
			RESULT IS COPIED , COSTVALUE IS	168
	SHOW		QUERYREWRITE : APPLIED RULES:	
	SHOW		PushDownPredicates	1

Obr. 66: Exekučný plán - MaxDB

9.1.10 Test 10

```
SELECT SUM( DISTINCT co~area ) FROM zpdcity AS c
   JOIN zpdcountry AS co ON co~countrycode = c~countrycode
   JOIN zpdposition AS p ON c~id = p~id
  WHERE (latitude BETWEEN 30 AND 40) AND (longitude BETWEEN 20 AND 40)
        AND continent = 'EU'.
```

Operation	Options	Object_name
COLUMN SEARCH	SUM(DISTINCT CO.AREA) (LATE MATERIALIZATION, OLTP SEARCH, ENUM_BY: CS_JOIN)	
AGGREGATION	AGGREGATION: SUM(DISTINCT CO.AREA)	
JOIN	JOIN CONDITION: (INNER) C.COUNTRYCODE = CO.COUNTRYCODE, (INNER) P.ID = C.ID	
COLUMN TABLE	FILTER CONDITION: P.LONGITUDE >= '20' AND P.LATITUDE >= '30' AND P.LONGITUDE <= '40' AND P.LATITUDE <= '40'	ZPDPOSITIONC
COLUMN TABLE		ZPDCITYC
COLUMN TABLE	FILTER CONDITION: CO.CONTINENT = n'EU'	ZPDCOUNTRYC

Obr. 67: Exekučný plán - Hana DB stĺpcový prístup

Operation	Options	Object_name
ROW SEARCH	SUM(DISTINCT CO.AREA)	
AGGREGATION	AGGREGATION: SUM(DISTINCT CO.AREA)	
BTREE INDEX JOIN	INDEX NAME: _SYS_TREE_RS_#5567044_#0_#P0, INDEX CONDITION: C.ID = P.ID, JOIN CONDITION: , INDEX FILTER: '20' <= P.LONGITUDE AND '30' <= P.LATITUDE AND '40' >= P.LONGITUDE AND '40' >= P.LATITUDE (ENUM_BY: INDEX_JOIN)	ZPDPOSITION
HASH JOIN	MANY-TO-ONE HASH BUILD: RIGHT, JOIN CONDITION: C.COUNTRYCODE = CO.COUNTRYCODE	
TABLE SCAN		ZPDCITY
TABLE SCAN	FILTER CONDITION: CO.CONTINENT = 'EU'	ZPDCOUNTRY

Obr. 68: Exekučný plán - Hana DB riadkový prístup

```
SELECT STATEMENT ( Estimated Costs = 10 , Estimated #Rows = 0 )
└─ 12 SORT AGGREGATE
   └─ 11 VIEW VW_DAG_0
         ( Estim. Costs = 10 , Estim. #Rows = 1 )
         Estim. CPU-Costs = 11.175.324 Estim. IO-Costs = 8
      └─ 10 HASH GROUP BY
            ( Estim. Costs = 9 , Estim. #Rows = 1 )
            Estim. CPU-Costs = 11.175.324 Estim. IO-Costs = 8
            Memory Used KB: 1.015.808
         └─ 9 FILTER
               Filter Predicates
            └─ 8 NESTED LOOPS
                  ( Estim. Costs = 8 , Estim. #Rows = 1 )
                  Estim. CPU-Costs = 2.681.176 Estim. IO-Costs = 8
               ├─ 6 NESTED LOOPS
                     ( Estim. Costs = 8 , Estim. #Rows = 1 )
                     Estim. CPU-Costs = 2.681.176 Estim. IO-Costs = 8
                  ├─ 4 NESTED LOOPS
                        ( Estim. Costs = 8 , Estim. #Rows = 1 )
                        Estim. CPU-Costs = 2.680.906 Estim. IO-Costs = 8
                     ├─ 1 TABLE ACCESS FULL ZPDPOSITION
                           ( Estim. Costs = 8 , Estim. #Rows = 1 )
                           Estim. CPU-Costs = 2.680.658 Estim. IO-Costs = 8
                           Filter Predicates
                     └─ 3 TABLE ACCESS BY INDEX ROWID ZPDCITY
                           Estim. CPU-Costs = 248 Estim. IO-Costs = 0
                        └─ 2 INDEX UNIQUE SCAN ZPDCITY~0
                              Search Columns: 1
                              Estim. CPU-Costs = 210 Estim. IO-Costs = 0
                              Access Predicates
                  └─ 5 INDEX UNIQUE SCAN ZPDCOUNTRY~0
                        Search Columns: 1
                        Estim. CPU-Costs = 210 Estim. IO-Costs = 0
                        Access Predicates
               └─ 7 TABLE ACCESS BY INDEX ROWID ZPDCOUNTRY
                     Estim. CPU-Costs = 270 Estim. IO-Costs = 0
                     Filter Predicates
```

Obr. 69: Exekučný plán - Oracle

OWNER	TABLENAME	COLUMN OR INDEX	STRATEGY	PAGECOUNT
	P		TABLE SCAN	37
	C	ID	JOIN VIA KEY COLUMN	167
	CO	COUNTRYCODE	JOIN VIA KEY COLUMN	3
			TABLE HASHED	
			NO TEMPORARY RESULTS CREATED	
			RESULT IS COPIED , COSTVALUE IS	273
	SHOW		QUERYREWRITE : APPLIED RULES:	
	SHOW		PushDownPredicates	1

Obr. 70: Exekučný plán - MaxDB