VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Reliacinės ir kolonėlinės duomenų bazių palyginimas

Comparison of relational and columnar database systems

Bakalauro baigiamasis darbas

Atliko:	Justinas Matulevičius	(parašas)
Darbo vadovas:	Andrius Adamonis	(parašas)
Recenzentas:	Elita Pakalnickienė	(parašas)

Vilnius – 2016

Santrauka

Šiame darbe nagrinėjamos dviejų skirtingų tipų (reliacinės ir kolonėlinės) duomenų bazės, analizuojami skirtumai tarp įvairių duomenų struktūrų realizacijų nagrinėjamose duomenų bazėse, bei ieškoma geriausių duomenų struktūrų saugoti konkretaus tipo duomenimis. Nagrinėjama problema – turint didelį kiekį, reikalingų apdoroti, duomenų didelę įtaką daro ir struktūra, kuria tie duomenys išsaugoti. Kokia yra tinkamiausia struktūra, norint kuo optimaliau išvesti vidurkius, atlikti palyginimus ir bandoma išsiaiškinti. Darbo eigoje buvo atliktas bandymas su dideliu duomenų kiekiu, kurio analizavimui reikalingos įvairios aritmetinės operacijos filtravimas pagal jų rezultatus. Buvo išsiaiškinta, kad analizuojant duomenis, tai yra – skaitant iš duomenų bazės, kolonelinės duomenų bazės didėjant duomenų kiekiui darbą atlieka greičiau, tačiau, darbą atliekant su mažais duomenų kiekiais, reliacinės duomenų bazės yra lengviau paruošiamos darbui bei veikia netgi greičiau. Praktinis eksperimentas buvo atliktas naudojant Ruby programavimo kalbą (karkasas nebuvo naudotas, siekiant išvengti papildomo sluoksnio kodo, kuris, galbūt, iškreiptų rezultatus) bei tris skirtingas duomenų bazes: “Cassandra” naudota kaip kolinėlinės duomenų bazės pavyzdys, kuri siekia būti kuo panašesnia į reliacinę duomenų bazę ir netgi stengiasi suteikti galimybes duomenis apdoroti eilutėmis, “Postgresql” naudota kaip reliacinės duomenų bazės pavyzdys, o “Druid” naudota kaip kolonėlinės duomenų bazės, skirtos būtent duomenų analizavimui, pavyzdys . Praktinio darbo metu buvo atliktus užduotys su trimis skirtingo dydžio įvesties failais, siekiant įvertinti ne tik kaip duomenų kiekis įtakoja duomenų bazės veikimo greitį, bet ir tai, kokį duomenų kiekį turint tiriamoje situacijoje reikėtų naudoti vieną duomenų bazę, o kuomet jau kitą.

Raktiniai žodžiai: NoSQL, Duomenų Bazės, Kolonėlinės duomenų bazės, Reliacinės duomenų bazės, Dideli duomenų kiekiai, Našumas

Summary

In this document two different types (relational and columnar) of databases will be compared by analysing differences between various data storage realisations in each type of selected databases, looking for the best data storage choice when saving particular type of data. The objective of this work is to overcome the performance issues when working with a big set of analytical data by selecting the best possible data storage format and database which implements selected structure the best. During the process the test was carried out. A big set of data was prepared for various agregation operations (such as calculation of averages). Results have showed that analysing(reading data from database) is done better by columnar type of database, especially when working with bigger sets of data. However, relational databases are easier to set up and perform almost the same or even better when working with little sets of data. The whole test was performed using Ruby programming language (no framework was used due to possibility of unwanted overhead which might make the results less accurate) and three databases: "Cassandra" as row-oriented columnar database, Druid as completely columnar database which is configured to work best when working with analytical data and "Postgresql" as relational database. Three tests were made in total, each of them had different data set size with the purpose of finding the point, when switching from one database to another is the best option.

Keywords: NoSQL, Databases, Columnar databases, Relational databases, Big data, Performance

TURINYS

Įvadas

Įvairioms sistemoms persikeliant į kompiuterinę erdvę, o dažniausiai – būtent į internetinę, vis dažniau susiduriama su problema, jog dirbti su dideliais duomenų kiekiais, sprendimai, lygi šiol buvę universalūs ir tinkami viskam, nebetinka. Pastaruoju metu ”Big Data” raktažodį darbo skelbimuose aptikti galima vis dažniau, todėl nuspręsta išsiaiškinti kokie yra galimi būdai dideliems duomenų kiekiams saugoti/agreguoti bei analizuoti ir kodėl tie būdai yra geresni nei universalūs. Svarbu paminėti, kad norint pasiekti kuo geresnį našumą reikia tobulinti/optimizuoti kiek įmanoma žemesnio lygio sistemos komponentus. Pavyzdžiui, duomenų bazes pirmiau nei programavimo kalbas ar karkasus. Taip yra todėl, kad atmetus tikimybę jog vienos duomenų bazės įgyvendintos geriau nei kitos, abi kalbos turėtų dirbti su ta pačia duomenų baze, kuri bus arba tinkama situacijai arba ne. Sakykime, kad darbo patirtis su reliacinėmis duomenų bazėmis kol kas, atrodo, suteikia viską ko reikia ir nėra matomas joks tikslas keisti esamos programos infrastruktūrą. Lygiai taip kažkada ir karieta su arkliu buvo pakankamai greitas būdas nukeliauti iš Vilniaus į Trakus, tačiau, atėjo momentas, kai prireikė greičio ir patogumo. Ta pati situacija aktuali ir informacinėse technologijose.

Šiuo metu vienas iš kriterijų lemiančių programavimo kalbos pasirinkimą yra jos greitis, kadangi jis įtakoją galimybes augti/plėstis. Problema - padidėjus produkto naudojimui atsiranda nenumatytų bėdų – pritrūksta resursų. Vienas geresnių to pavyzdžių – didelių reliacinių lentelių sąjungos (join’ai). Kuo didesnes lenteles ketinama sujungti užklausos metu tuo daugiau laisvos virtualios atminties serveris privalo turėti, tuo lėčiau užklausa bus vykdoma ir tuo ilgiau tinklalapis bus nepasiekiamas (su sąlyga, kad visi serverio resursai dalinasi vieną giją, kuomet taip nėra – reikia galingesnio procesoriaus). Dar viena problema – dirbant su dideliais duomenų kiekiais svarbu išsaugoti duomenų vientisumą. Dabartiniame (reliaciniame) modelyje duomenų vientisumas užtikrinamas visas susijusias operacijas atliekant tranzakcijose (viskas arba nieko). Dėja, tranzakcijos taip pat yra ganėtinai lėta operacija, o be viso to jos dar ir užrakina resursus, su kuriais tuo metu dirbama, nuo prieigos. Taip esant didesnei sistemos apkrovai susidaro vartotojų, laukiančių savo teisės skaityti arba rašyti duomenis, eilės, resursai tampa nebepasiekiami, o tinklalapis atsako vis lėčiau.

Pirma į galvą atkeliaujanti mintis – pridėti resursų. Kompiuterio resursai šiais laikais pigūs, todėl, atrodo, galima nusipirkti ir dar kelis papildomus diskus, galingesnį procesorių ar daugiau virtualios atminties. Vis dėl to, jeigu tinklalapis ir toliau sėkmingai augs ir vartotojų daugės, problemos tikrai sugrįš. Ką darome tada? Kadangi serverio resursai jau maksimalūs, ieškome kitų sprendimų kaip padidinti resursų kiekį. Į pagalbą ateina duomenų bazių replikos. Tiesa, tai atneša ligi tol nebuvusių rūpesčių – užtikrinimas, kad duomenys vientisi bei duomenų valdymas, kuomet operacijos nepavyksta yra sudėtingi uždaviniai, kuomet dirbama su keliomis duomenų bazių replikomis. Taip pat galimai išjungiamas .log tipo failų pildymas (tam, kad sumažintume disko įvesties bei išvesties (I/O) apkrovą), kas iš tiesų nėra itin geras pasirinkimas. Dar vėliau pradedama kreipti dėmesį į pačius saugomus duomenis. Pradedami dėlioti papildomi indeksai, optimizuojamos užklausos, bet esant dideliam apkrovimui ir tokiems aplikacijos mastams, tai neatneš itin didelės naudos, kadangi šie darbai bus bent dalinai padaryti jau anksčiau. Taigi, ieškojimas vietų, kurias

galima optimizuoti, tampa daug laiko atimančiu procesu, kuris nebūtinai duos tiek daug naudos. Pamatę, kad pastovus duomenų traukimas iš duomenų bazės yra brangi operaciją ir nebegalime jos optimizuoti greičiausiai bandysime duomenis kešuoti (užklausų rezultatus atnaujinti tik kas tam tikrą laiko intervalą ir tik pasikeitus). Tačiau, tai ir vėl atneša papildomų rūpesčių – tenka užtikrinti, kad visi reikalingi duomenys yra validūs, nepasenę ir visa informacija atnaujinama laiku. Maždaug šitoje vietoje esame išbandę pačius populiariausius sprendimus. Ką darome toliau? Pamatę, kad jau turime bendresnį programos vaizdą galime kai kuriuos duomenis iš atskirų lentelių sujungti į bendras (taip, dublikuoti) pagal tai, kurių lentelių jungimai užklausose sunaudoja didžiausią laiko dalį. Tai procesas, vadinamas denormalizacija. Kitaip sakant, vienintelis būdas toliau augti ir leisti sau priimti didesnį kiekį naudotojų – keisti duomenų bazės struktūrą.[Tra13]
Todėl šio darbo tikslas - išanalizuoti, kokie yra principiniai skirtumai tarp reliacinių ir kolonėlinių duomenų bazių uždaviniams, kuriuose reikalinga saugoti ir agreguoti didelius duomenų kiekius. Keliama hipotezė – kolonėlinio tipo duomenų bazės yra geresnis pasirinkimas duomenų agregavimui ir skaitymui iš duomenų bazės, kuomet duomenų kiekis yra didelis ir norint atlikti operaciją reikia tik dalies visų duomenų.

Tikslą įgyvendinsiu trimis etapais

- **Išsiaiškinti abiejų tipų duomenų bazių veikimo principus. Bandymo metu bus analizuojamos dviejų tipų duomenų bazės - reliacinės ir kolonelinės.**
  Reliacinėms duomenų bazėms atstovaus ”Postgresql” duomenų bazė (nors ją galima naudoti ir kaip kolonėlinę, tačiau jos pirminė paskirtis visi gi tokia nėra), kadangi ji vienaip ar kitaip laikoma viena geriausių (greičiausių, stabiliausių, patikimiausių) iš visų atviro kodo reliacinių duomenų bazių. Viena to priežasčių – 15 metų aktyvaus programos palaikymo ir kūrimo. Šiuo metu “Postgresql” versija yra net 9.5. Ji veikia įvairiausiose operacinėse sistemose, pradedant “linux” ar “OS X” ir baigiant “Windows”. Be standartinių SQL programavimo kalbos galimybių ji taip pat palaiko MVCC, GiST indeksavimą, procedūrų bei trigerių kūrimą ir saugojimą. “Postgresql” yra suderinama su C, C++, JAVA, PHP, RUBY ir kitomis programavimo kalbomis.
  Tuo tarpu kolonėlinės duomenų bazės atitikmenimis pasirinktos dvi duomenų bazės – “Cassandra” ir “Druid”. ”Cassandra” - visai nauja, atviro kodo duomenų bazė pasirodžiusi 2008 metų pabaigoje. Ji vertinama už itin patikimas operacijas su duomenimis ir tai lemia didelį duomenų vientisumą. “Cassandra” taip pat pasižymi tuo, kad yra bene geriausius našumo rodiklius turinti duomenų bazė, kuri įgyvendinta kolonėlinės duomenų struktūros principu, tačiau labai primena eilutines duomenų bazes. Tiesa, siekiant pačių didžiusių laimėjimų našume, “Cassandra” duomenų bazėje nėra galimos lentelių jungimo operacijos, reliacinėms duomenų bazėms būdingų tranzakcijų bei automatinių sąryšių tarp lentelių užtikrinimo (išorinių ir vidinių raktų sąryšių).
  Tuo tarpu “Druid” duomenų bazė yra visiškai atitinkanti kolonėlinio tipo duomenų bazės aprašymą. Ji taip pat nepalaiko tranzakcijų, sąryšių ir sąjungų tarp lentelių, tačiau ji yra optmizuota būtent duomenų analizavimui, ko mūsų uždavinys ir reikalauja.

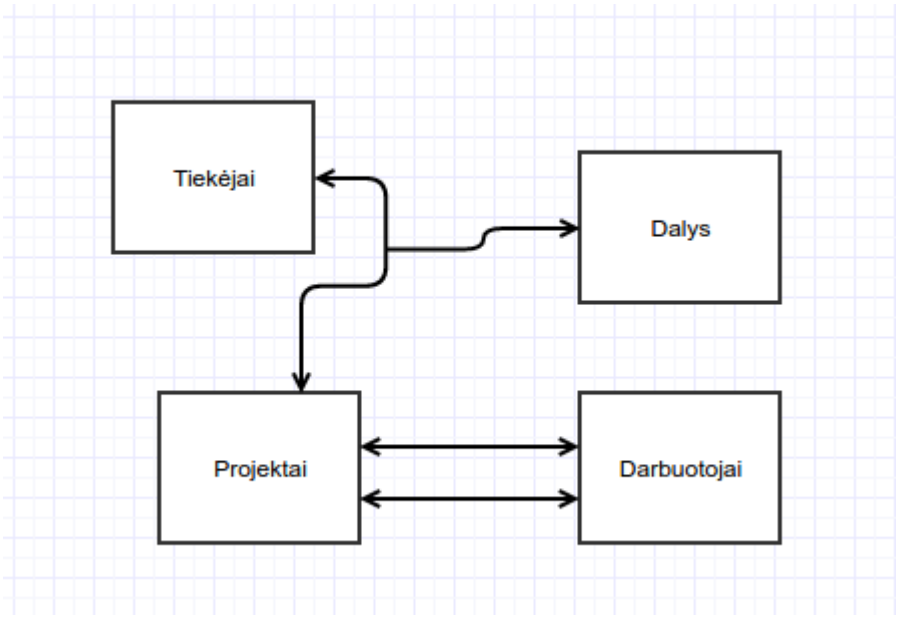
Apžvelgsiu abiejų duomenų bazių esmines funkcionalumo galimybes taip išsiaiškindamas kiekvienos duomenų bazės privalumus ir trūkumus. Kitaip sakant – išanalizuosiu pačias duomenų bazes ir jų duomenų saugojimo principus.

- **Duomanų bazių palyginimas**
  Lyginsiu abiejų tipų duomenų bazes pagal tai, kaip jos įgyvendina mano iškeltai hipotezei aktualias operacijas. Ieškosiu kriterijų, kurie padeda pasirinkti kokio tipo duomenų bazę reikėtų naudoti. Kitaip sakant, bus nagrinėjama, kuri duomenų bazė yra labiau tinkama užduočiai atlikti iš teorinės pusės (pagal pirmo punkto rezultatus bei viešai prieinamus atliktus tyrimus).

- **Praktinė darbo dalis**
  Atliksiu konkrečią praktinę užduotį su skirtingo dydžio duomenų kiekiais tam, kad išsiaiškinčiau kuri duomenų bazė yra tinkamesnė mano nagrinėjamai užduočiai. Bandysiu įrodyti, kad kolonėlinės duomenų bazės yra geresnis pasirinkimas norint analizuoti didelius duomenų kiekius (tai reiškia – skaitant duomenis).

1. Duomenų bazės

Visų pirma, reikia išsiaiškinti, kas yra duomenų bazė. Apibrėžimas galėtų skambėti maždaug taip:
Duomenų bazė tai vieta, kurioje saugomos nevienalytės, tačiau tarpusavy prasmiškai susijosios informacijos dalys.



1 pav. Pavyzdinė duomenų bazės schema

Pavyzdinė schema viršuje aprašo kompaniją, kurioje darbuotojai dirba projektuose, o projektai naudoja įrangą/dalis kurias gauna iš tiekėjų. Papildoma rodyklė iš darbuotojų į projektus rodo, kad egzistuoja ir tokių darbuotojų, kurie yra projektų vadovai. Taigi, visa ši informacija yra saugoma duomenų bazėje tam, kad esant reikalui būtų galima patogiai suskaičiuoti, kad ir pavyzdinės sąlygos, nurodytos paveiksliuke, rezultatą.
Viena esminių duomenų bazių savybių yra vykdyti užklausas ir taip išfiltruoti visus turimus duomenis, ko rezultate - grąžinti tik tuos, kurie yra svarbūs (atitinka filtrus).

Duomenims, saugojamiems duomenų bazėse, galioja šios savybės:

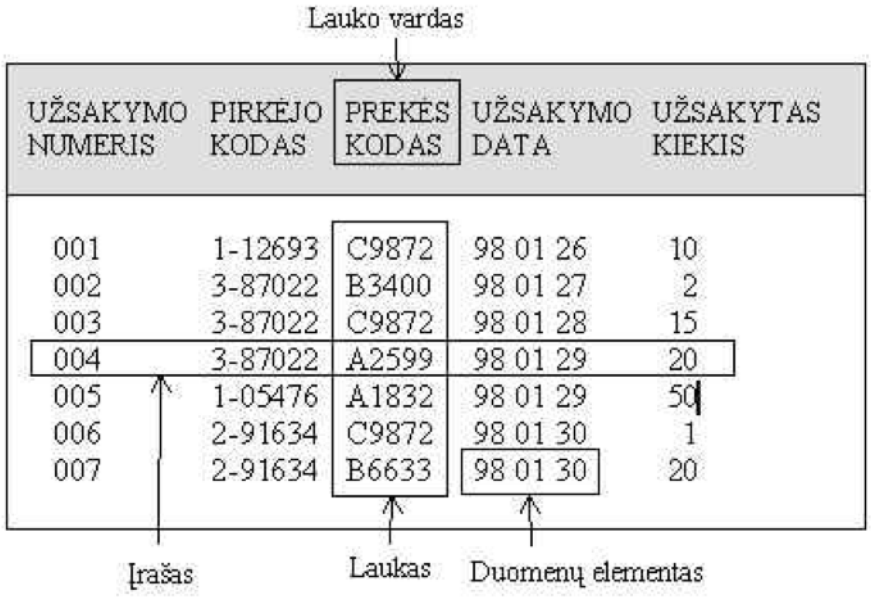
- **Integruotumas**
  Savybė, kai visi duomenys kaupiami ir saugomi kartu nustačius jų tarpusavio ryšius. Taip saugomus duomenis dažniausiai naudoja ne vienas, o keli vartotojai.

- **Neperteklišlumas**
  Duomenys saugomi vengiant dubliavimo. To nedarant, tuos pačius duomenis tenka atnaujinti keliose vietose, be reikalo naudojami kiti resursai (pavyzdžiui, atmintis).

- **Nepriklausomumas**
  savybė nurodanti, kad duomenys yra tokie, kuriuos pakeitus kitais, juos apdorojančios taikomosios programos ir toliau veiks korektiškai.

1.1. Reliacinės duomenų bazės

Reliacinė duomenų bazė – tai tokia duomenų visuma kurioje visa informacija saugoma dvimatėse lentelėse.
Kiekviena lentelė susideda iš eilučių (dar vadinamų įrašais arba kortežais (record)) ir stulpelių (dar vadinamų laukais arba atributais (field))
Eilutėse yra duomenys apie lentelėje pateiktus kortežus (dokumentus, žmones, vienu žodžiu – vieno tipo objektus). Stulpelių ir eilučių susikirtimuose yra konkrečios saugomų lentelėje duomenų reikšmės (laukų reikšmės).



2 pav. Reliacinės duomenų bazės pavyzdys

Reliacinėse duomenų bazėse kiekviena lentelė pasižymi tokiomis savybėmis:

- Visi įrašai yra vienodai grupuoti pagal prasmę, turi tą pačią struktūrą. Visuose iki vieno įrašuose yra tiek pat laukų, kurie yra vienodo tipo. Skirtinguose laukuose gali būti skirtingų tipų duomenys
- Lentelėje negali būti visiškai tuščių arba visiškai identiškų įrašų. Atskiri duomenų elementai gali būti ir tušti ir pasikartojantys
- Įrašų bei laukų tvarka yra nesvarbi. Operacijos gali būti atliekamos bet kokia tvarka.
- Kiekviena lentelė turi suteiktą vardą, kuris savo prasme turėtų atitikti realių duomenų objekto pavadinimą, o lentelės laukai – to objekto atributų pavadinimus.

1.1.1. Reliacinio duomenų bazės modelio aprašymas

Reliacinėje duomenų bazėje visų pirma yra pati duomenų bazė, kuri turi lenteles. Lentelės turi pavadinimus ir turi vieną ar daugiau stulpelių, kurie taip pat turi pavadinimus. Kuomet pridedame duomenis į lentelę mes privalome nurodyti reikšmę kiekvienam laukeliui, kitu atveju bus naudojama arba standartinė nustatyta reikšmė, arba laukelis įgaus reikšmė NULL. Sukurtas įrašas pridės vieną eilutę į lentelę, kurią vėliau galėsime perskaityti, jei žinosime lentelės unikalų identifikatorių (pirminį raktą), arba, sudarysime SQL užklausą, kurą mūsų turima eilutė atitiktų. Norint atnaujinti duomenis lentelėje, galima atnaujinti arba visus arba dalį jų priklausomai nuo to, ar nurodysime "where" sąlygą SQL užklausoje.

1.1.2. Suteikiami funkcionalumai

Tranzakcijos. Duomenų bazės tranzakcija, pasak Jim Gray, yra būsenos transofmacija, kuri turi 4 savybes, šifruojamas raidėmis ACID.

- **A(atomic)**
  Atominės operacijos yra tokios operacijos, kurios turi tik dvi galimas reikšmes. Arba įvykdytos visos (1) arba nė viena (0). Dalinis operacijų įvykdymas nėra įmanomas, kadangi įvykus klaidai, visos iki tol atliktos operacijos yra gražinamos į buvusias reikšmes.
- **C(consistent)**
  Operacijos pastovumas yra tokia savybė, kuri užtikrina, kad kuomet keičiama duomenų būsena ji yra pakeičiama pilnai. Tai reiškia, kad neįmanoma pasiekti duomenų egzistuojančių vidury tranzakcijos (pavyzdžiui, ištrinta tik pusė įrašų).
- **I(isolated)**
  Izoliuotos operacijos yra tos, kurios savo veikimu negali padaryti įtakos kitoms operacijoms. Pavyzdžiui, jei kelios operacijos vienu metu bando pasiekti tą patį įrašą, vienas iš laukelių privalės laukti.
- **D(durable)**
  Operacijos pakeitimai yra garantuoti ir ilgalaikiai. Tai reiškia, kad tranzakcijai pavykus jie įsigalioja ir visos kitos operacijos (įtraukiant ir laukiančias naujas tranzakcijas) naudos naujuosius duomenis.

Kadangi tranzakcijos užrakina reikalingus duomenis, tai reiškia, kad bendras duomenų bazės greitis šiek tiek sulėtėja. Tačiau, įrašant/atnaujinant duomenis tai yra itin svarbus funkcionalumas.

Schema. Tai dažnai išryškinama reliacinių duomenų bazių funkcija. Ji skirta reliacinio modelio objektų atvaizdavimui. Pavyzdžiui, jei turėtume aplikacijoje, kurioje egzistuoja produktų sąrašas ir kategorijų sąrašas, tai duomenų bazės schemoje tai aiškiai matytųsi. Tiesa, kadangi vienas produktas gali priklausyti kelioms kategorijoms, o kategorija gali turėti daugiau nei

vieną produktą, reikalinga dar viena lentelė, kuri leistų tą duomenų sujungimą atlikti. Ir tos lentelės kūrimas ir palaikymas gali būti lėtas, pastangų reikalaujantis procesas.

1.2. PostgreSQL

"PostgreSQL" duomenų bazė buvo pradėta kurti 1986 metais. Tai yra reliacinė duomenų bazė. Susidomėjimas "PostgreSQL" bei naudojimasis ja išaugo prieš porą metų, kadangi tai buvo bene vienintelė reliacinė duomenų bazė, kuri laikui bėgant ir pildėsi naujomis funkcijomis ir greitėjo.
Visi duomenys yra saugomi lentelėse (kadangi tai tradicinė reliacinė duomenų bazė). Kiekviena lentelė gali turėti atskiras prieigos teises. Duomenų bazė turi visas funkcijas, kurios galimos aprašyti SQL kalba. Svarbiausios jų:

lentelių sąjungos (join'ai). Dviejų skirtingų lentelių duomenų sujungimas. Juos verta daryti dėl kelių priežasčių, bet didžiausios būtų šios dvi:

- Galima filtruoti duomenis tiesiai iš dviejų lentelių
- Duomenų bazės yra itin žemo lygio programos. Tai reiškia, kad jos yra itin greitos, todėl dažniausiai apsimoka visas įmanomas operacijas atlikti būtent duomenų bazės lygyje.

Laikinos lentelės (views). Lentelė skirta duomenų skaitymui, kuri yra kažkokios kitos užklausos rezultatas. Iš esmės, tai yra ilgalaikė lentelių sąjungą, kurios nereikia kas kartą perkurti iš naujo (tačiau, galima užtikrinti, kad atsinaujinus duomenims atsinaujins ir laikinoji lentelė)

1.2.1. Našumą didinantys komponentai

1.2.1.1. Indeksai

Indeksas yra speciali struktūra, padedanti turėti nuorodas į konkrečias duomenų eilutes. "Postgresql" duomenų bazėje tai yra kopija originalaus indeksuojamo laukelio kartu su nuoroda į visa eilutę. Kuomet bus bandoma pasiekti duomenis, "Postgresql" pirmiausia bandys juos rasti indeksų lentelėje, o ten neradus, atliks pilną paiešką, kurios metu bus tikrinamas kiekviena eilutė ir tik patikrinus visus bus grąžintas rezultatas.
Privalumai: Indeksai pagreitina paiešką tarp duomenų. Postgresql leidžia indeksuoti ir daugiau nei vieną laukelį į vieną indeksą.
Trūkumai: Indeksai sulėtina įrašymą į duomenų bazę, kadangi be duomenų reikia tvarkyti ir indeksų lentelę (reikia ir įrašyti ir perrikiuoti indeksus, tačiau tai daroma automatiškai). Taip pat, indeksai užima papildomą vietą diske.

1.2.1.2. Valdymo planas

"Postgresql" turi komandą "**explain**", kuri leidžia pažiūrėti kas iš tiesų vyksta vykdant tam tikrą užklausą. Tikslingas šio įrankio naudojimas leidžia nuspręsti kokie galimi duomenų

bazės optimizavimo būdai (pavyzdžiui, kur sudėti indeksus), kadangi yra matoma, kuriose dalyse užklausos užtrunka ilgiausiai.

Komanda "**explain**" gali būti veiksminga analizuojant ne tik SELECT bet ir kitas (INSERT, UPDATE, DELETE, EXECUTE, DECLARE) komandas. Ji turi kelis režimus:

- "**generic**" – rodo tik tai, kokios užklausos bus vykdomos
- "**analyze**" – užklausos iš tiesų vykdomos, todėl gaunami rezultatai, kas ir kiek truko.
- "**verbose**" – itin detalus ir žemo lygio paaiškinimas, kuris yra sunkiai suprantamas vartotojui, todėl praktiškai nenaudojamas.

1.2.1.3. Sąlyginiai duomenų apribojimai

Kartais apribojimai reikalingi tik egzistuojant tam tikroms sąlygoms. Pavyzdžiui, kuomet triname vartotojus, galbūt galime juos tiesiog užšaldyti (tai yra – pakeisti kokios nors vėliavėlės reikšmę). Tuomet, kai vartotojas po kurio laiko nuspręs grįžti, jį bus galime atšaldyti, ir jo perkurti iš naujo nereikės. Taip pat galima naudoti dalinius indeksus tam, kad suindeksuoti būtų tik aktyvūs(neužšaldyti) vartotojai.

1.2.1.4. Kešavimas

Standartinė taisyklė daugumai aplikacijų – tik dalis visų duomenų turėtų būti prieinama tiesiogiai. Yra taisyklė sakanti, virš 80 procentų visų užklausų yra vykdomos tik su 20 procentų duomenų. Tai reiškia, kad dažniausiai vykdomų užklausų rezultatai gali būti saugomi ir užklausos vykdomos tik pasikeitus saugomiems duomenims (smarkiai pasikeitus duomenys turi būti iš naujo kešuojami, maži pakeitimai leidžia kešuotus duomenis tiesiog atnaujinti dalinai). "Postgresql" duomenų bazė pati nustato kurie duomenys yra dažniausiai užklausiami ir juos kešuoja. Rodiklis rodantis, kaip dažnai duomenys traukiami iš kešuotų duomenų vietoj tikrosios užklausos vykdymo yra vadinamas "cache hit rate". Siūloma jį turėti ne mažesnį nei 99 procentų.

1.2.2. "Postgresql" ypatybės

1.2.2.1. HStore duomenų tipas

Tai rakto-reikšmės tipo duomenų struktūra. Ji atitinka hash'us arba žodynus (dictionary) daugumoje programavimo kalbų.
Palaikomas pilnas operacijų rinkinys (stulpelių kūrimas, duomenų įterpimas ir gavimas)

1.2.2.2. Masyvai

Postgresql leidžia stulpeliams būti apibrėžtiems kaip masyvams. Masyvo elementų tipas gali būtų betkoks Postgresql validus tipas, vartotojo apsibrėžtas tipas arba enumerize tipas. Palaikoma paieška masyvuose, įterpimas į masyvus, duomenų išrinkimas iš masyvo tipo stulpelių, duomenų

masyvo viduje keitimas.
Enumerize tipas - tai tipas, kuomet yra išvardijamos visos laukeliui galimos priskirti reikšmės.

1.2.2.3. Bendra lentelių išraiška

Ilgos ir sudėtingos SQL užklausos dažniausiai sunkiai suprantamos. Bendros lentelių išraiškos(CTE's) yra tarsi laikinos lentelės (views), tačiau tik tam tikros užklausos veikimo metu. Galime kurti kelias lentelių išraiškas naudojančias kitas, anksčiau sukurtas, lentelių išraiškas. Iš esmės, bendroji lentelių išraiška reikalinga tuomet, kai vienai užklausai atlikti reikalingas kitos užklausos rezultatas.

1.2.3. Apibendrinimas

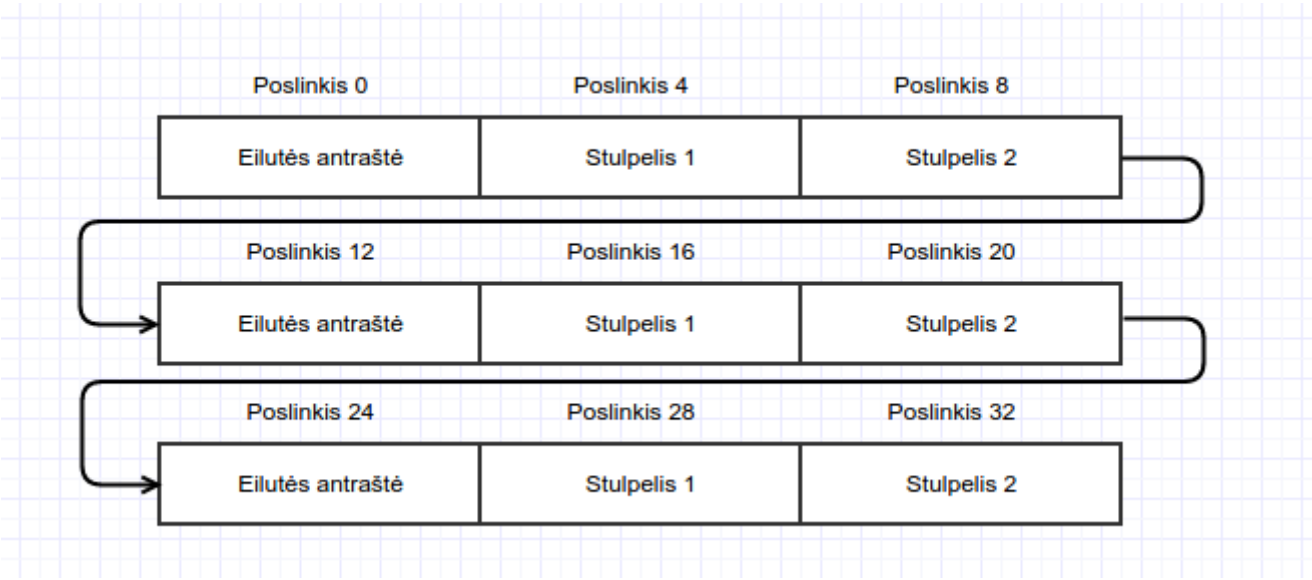
Reliacinės duomenų bazės yra labai geros sprendžiant pagrindinius duomenų saugojimo uždavinius. "Postgresql" yra stipriai ištobulinta duomenų bazė, pasižyminti svarbiausiomis savybėmis – ji greita (lyginant su kitomis reliacinėmis duomenų bazėmis), dažnai atnaujinama (todėl suderinama su įvairiomis duomenų struktūromis), turi keletą būdų kuriais galima paspartinti jos veikimą esant mažam-vidutiniam duomenų kiekiui, tačiau dėl to, kad fokusuojamasi į reliacinį modelį, kuriame duomenų įrašymas ir skaitymas vienodai svarbus, kuriame galima jungti dideles lenteles ir taip atlikti įvairias agregavimo funkcijas, susiduriama su iš to kylančiomis problemomis, kuomet lankomumas išauga. Tuomet tenka atsikratyti kai kurių esminių funkcijų, pavyzdžiui, duomenų lentelių jungimo (join'ų) arba laikinų lentelių (views'ų). Tai daroma denormalizuojant duomenis, o tai reiškia, kad reikės palaikyti tų pačių duomenų atnaujinimą keliose skirtingose vietose, kas iš tiesų, jau prieštarauja duomenų bazių principams. Be viso to, didelių lentelių indeksavimas, tranzakcijos, visa tai, kuomet susiduriama su dideliu duomenų kiekiu, tampa duomenų bazę lėtinančiais aspektais ir jų sudėtingas (o kartais ir neįmanomas) sprendimas verčia ieškoti alternatyvų.

1.3. Kolonėlinės duomenų bazės

Reikėtų pradėti nuo to, kodėl prireikė išrasti kolonėlines duomenų bazes. Visų pirma, pagrindinė priežastis, kodėl reliacinės duomenų bazės yra lėtos – lėti diskai, kuriuose saugoma informacija. Laikui bėgant yra tikimasi, kad diskai sparčiai greitės, tačiau diskų greičiai kyla nepakankamai greitai. Netgi serveris su 10 paralelėje veikiančių kietųjų diskų negali veikti taip sparčiai, kad procesoriui netektų laukti. Geriausias dalykas, ką tokioje situacijoje galima padaryti yra skaitymo iš disko greičio didinimas. Kadangi kiekviena nuskaitymo operacija iš disko yra ganėtinai brangi, reikėjo rasti būdą, kaip vienu nuskaitymu išgauti kuo didesnę dalį naudingų(informaciją perteikiančių) bitų. Taip gimė idėja neskaityti nieko daugiau, nei užtenka įvykdyti užklausai. Taip duomenų pralaidumas padidės nors tuo pat metu, jokie pakeitimai serverio infrastruktūroj (pavyzdžiui, kietuosiuose diskuose) nebus atlikti. Būtent tai ir daro kolonėlinės duomenų bazės.

1.3.1. Kodėl tai veikia greičiau

Reliacinėje duomenų bazėje duomenys saugomi eilutėse, kur kiekviena eilutė turi visų laukelių reikšmes. Diske duomenys saugomi tokia struktūra, kurioje eilutės antraštė pateikia informaciją apie eilutėje esančias reikšmes (kurios jų tuščios, pavyzdžiui)

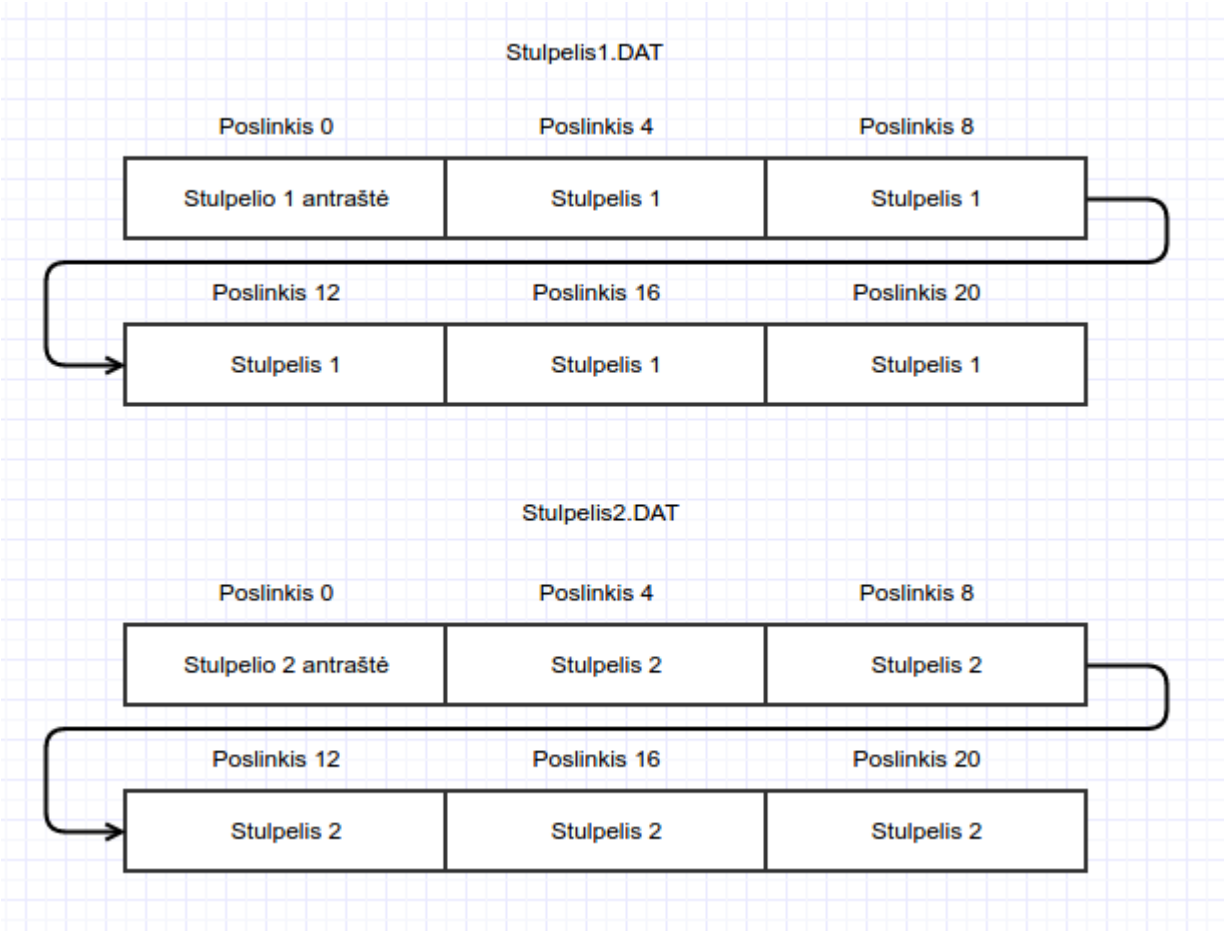


3 pav. Reliacinės duomenų bazės saugojimo diske schema [Tra13]

Toks sprendimas buvo priimtas todėl, kad tiek HDD tiek SSD diskai dirba sparčiausiai, kuomet informacija skaitoma paeiliui. Svarbu žinoti, kad visos eilutės skaitymas nėra brangesnė operacija nei 4 baitų (vienos reikšmės), kadangi dažniausiai yra skaitomas visas disko blokas (4096 baitai).

Kalbant apie uždavinį, mes nagrinėjame duomenų bazių panaudojamumą, kuomet reikia analizuoti duomenis, o tai reiškia – skaityti visą lentelę ir atlikti veiksmus su perskaitytais duomenimis. Tokios operacijos reliacinėje duomenų bazėje nesinaudoja jos teikiamais privalumais (pavyzdžiui, indeksavimu ar tranzakcijos), tačiau yra greitos savaime. Kitaip sakant, reliacinės duomenų bazės bus naudojamos efektyviai, jei vienos užklausos metu mes norėsime nuskaityti daugumos stulpelių

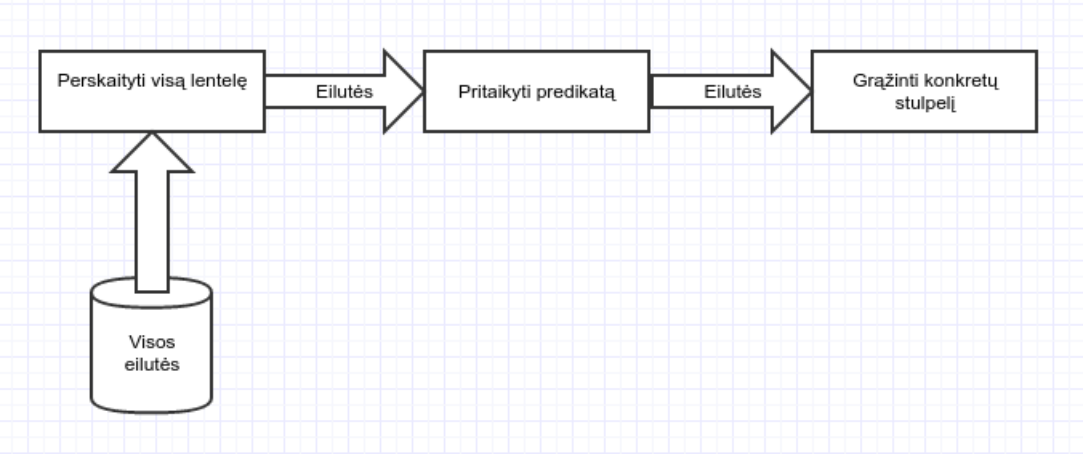
reikšmes iš daugumos eilučių.
Problema tame, kad tokios “SELECT *” tipo užklausos nėra tipinės analizuojant duomenis, kadangi dažniausiai mums reikia ne visų, o tik keleto stulpelių reikšmių. Todėl analitiniam duomenų nagrinėjimui yra naudinga gebėti skaityti reikšmes iš stulpelių, kadangi taip mes gausime didžiausią procentą reikalingų duomenų iš visų nuskaitytų duomenų. kolonėlinės lentelės duomenų bazėje atrodo taip:



4 pav. Kolonėlinės duomenų bazės saugojimo diske schema [Tra13]

Taigi, reliacinė eilutė dabar tampa išskirstyta į keletą atskirų stulpelių blokų (tai gali būti atskiri failai diske). Visos eilutės nuskaitymas reikalauja vienos skaitymo operacijos (lygiai kaip ir reliacinėje duomenų bazėje), bet grąžinamų baitų kiekis yra žymiai mažesnis, kadangi pateikiama tik aktuali informacija.

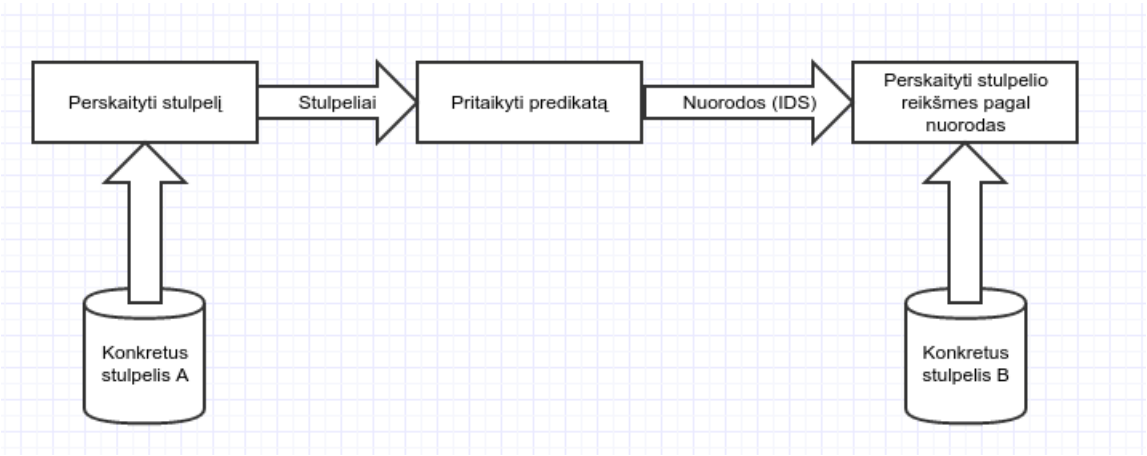
Palyginimui, užklausa norint išfiltruoti rezultatus pagal vieno stulpelio reikšmė reliacinėje duomenų bazėje atrodytų taip:



5 pav. Reliacinės duomenų bazės SELECT WHERE schema [Tra13]

Iš pradžių yra nuskaitoma visa lentelė, tuomet pritaikomas predikatas, kurio pagalba išfiltruojamos reikiamos eilutės ir gražinamas tik užklaustas stulpelis iš kiekvienos eilutės.

Ta pati užklausa kolonėlinėje duomenų bazėje atrodytų taip:



6 pav. Kolonėlinės duomenų bazės SELECT WHERE schema [Tra13]

Ji veikia priešinga, nes ji nuskaito tik predikato sąlygai įgyvendinti reikalingą stulpelį, ir išrenka sąlygą atitinkančių įrašų identifikatorius. Identifikatorius nurodo poziciją, kurioje yra reikšmė, o tai reiškia, kad kitame stulpelyje, tai pačiai eilutei priklausanti reikšmė bus toje pačioje pozicijoje. Nors šioje schemoje ir matoma, kad atliekamos dvi nuskaitymo užklausos iš skirtingų stulpelių, tai vis vien veikia greičiau, nei nuskaitymas visų stulpelių ir atsirinkimas tik reikalingų duomenų. Tai galime įrodyti atlikdami skaičiavimą:
Sakykim, kad eilutė reliacinėje duomenų bazėje yra sudaryta iš 10 stulpelių (kurių kiekvienas užima 4 baitus), ir tokių eilučių duomenų bazėje yra 10 milijonų. Tuomet mes nuskaitysime 32Mb papildomų duomenų, kurie nėra reikalingi, ir tai užtruks apie 3.20 sekundžių, kuomet kolonėlinėje duomenų bazėje tą patį rezultatą išgautume per maždaug 800ms. Tai yra 300 procentų papildomas

darbas, kurio galima išvengti ir šis skaičius tik didėja, kuomet diskai yra lėtesni.
Taigi, pagrindinė kolonėlinių duomenų bazių idėja yra ta, kad suprantant, jog analizuojant duomenis retai prireikia visų duomenų nuskaitymo, o dažniausiai, reikia tik vieno ar kelių stulpelių iš visų duomenų, duomenis ir yra patogu grupuoti stulpeliais. Taip išgauname pigesnes (mažiau resursų reikalaujančias) užklausas stulpelių informacijai išgauti nei skaitant visas eilutes bei traukiant duomenis iš jų.

1.3.2. Kolonėlinių duomenų bazių trūkumai

Žinoma, greitis skaitant informacija iš stulpelių turi ir savų minusų. Sakykime, prireikus iš atskirų stulpelių gauti visa eilutę mes pastebėtume žymiai lėtesnį rezultatą, nei tai darant reliacinėje duomenų bazėje. Taip yra todėl, kad atskirų stulpelių informacija gali būti išdėstyta skirtingose disko vietose, o galbūt ir skirtinguose diskuose. Be to, užklausų kiekis būtų lygus stulpelių kiekiui.

1.4. “Cassandra” duomenų bazė

Pasak (knygos autorius), jis “Cassandra” duomenų bazę galėtų apibūdinti trumpa pastraipa: ”Apache Cassandra yra atviro kodo, dalinama ir decentralizuota, lanksčiai besiplečianti ir itin aukšto pasiekiamumo duombazė, kuri puikiai toleruoja nesėkmes ir duomenis saugo stulpeliuose.. Ji naudojama tarp kai kurių populiariausių tinklalapių internete“

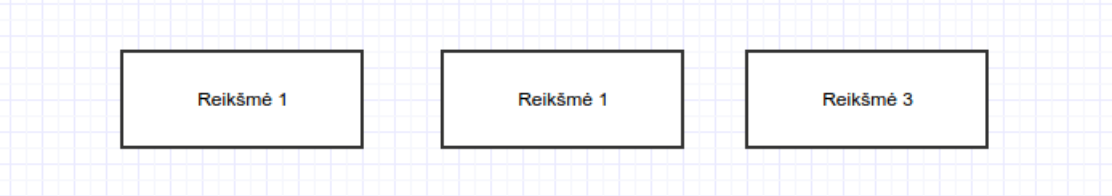
Dalinama duomenų bazė - tai reiškia, kad ji gali veikti keliuose serveriuose vienu metu, kai tuo tarpu vartotojui nebus jokio vizualaus skirtumo nuo duomenų bazės veikiančios viename serveryje. Ir nors reliacinės duomenų bazės irgi yra dalinamos, “Cassandra” tai daro kitokiu principu, apie kurį - kitame skyriuje.

Lanksčiai besiplečianti - tai reiškia, kad naujų serverių pridėjimas ir esamų išjungimas nesudaro jokių bėdų. “Cassandra” duomenų bazei nereikia jokių sudėtingų konfigūracijų norint pridėti dar vieną serverio instanciją. Užtenka nurodyti kur jis egzistuoja ir “Cassandra” pati jam nusiųs duomenis darbui, o nusprendus išjungti serverį, pasirūpins, kad visi duomenys būti padalinti liekančiuose dirbti serveriuose.

Kada reiktų naudoti Cassandra? Iš esmės, tokio tipo duomenų bazę reikėtų naudoti norint dirbti su dideliu kiekiu duomenų, tačiau, “Cassandra”, skirtingai nei dauguma kitų kolonėlinių duomenų bazių, puikiai optimizuota ir duomenų rašymui.

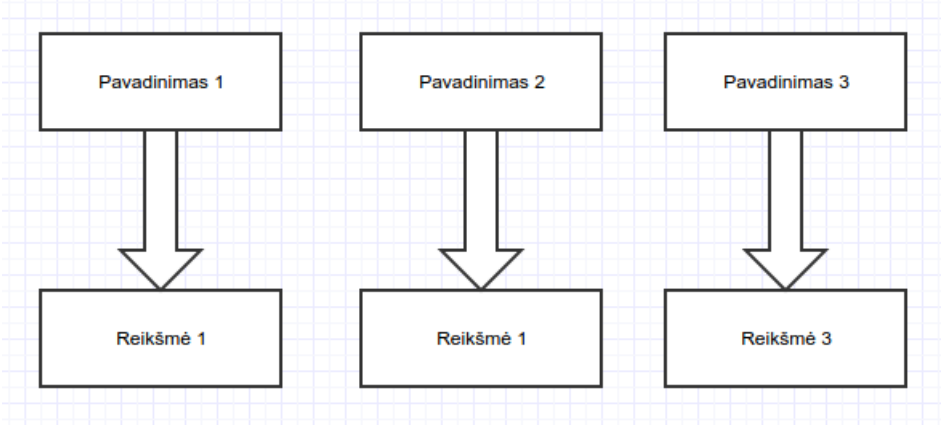
1.4.1. Duomenų modelis

Išanalizuokime “Cassandra” duomenų modelį, aprašytą knygoje. Paprasčiausia įmanoma duomenų struktūra būtų masyvas, kuriame reikšmės būtų išdėstytos tokia tvarka:



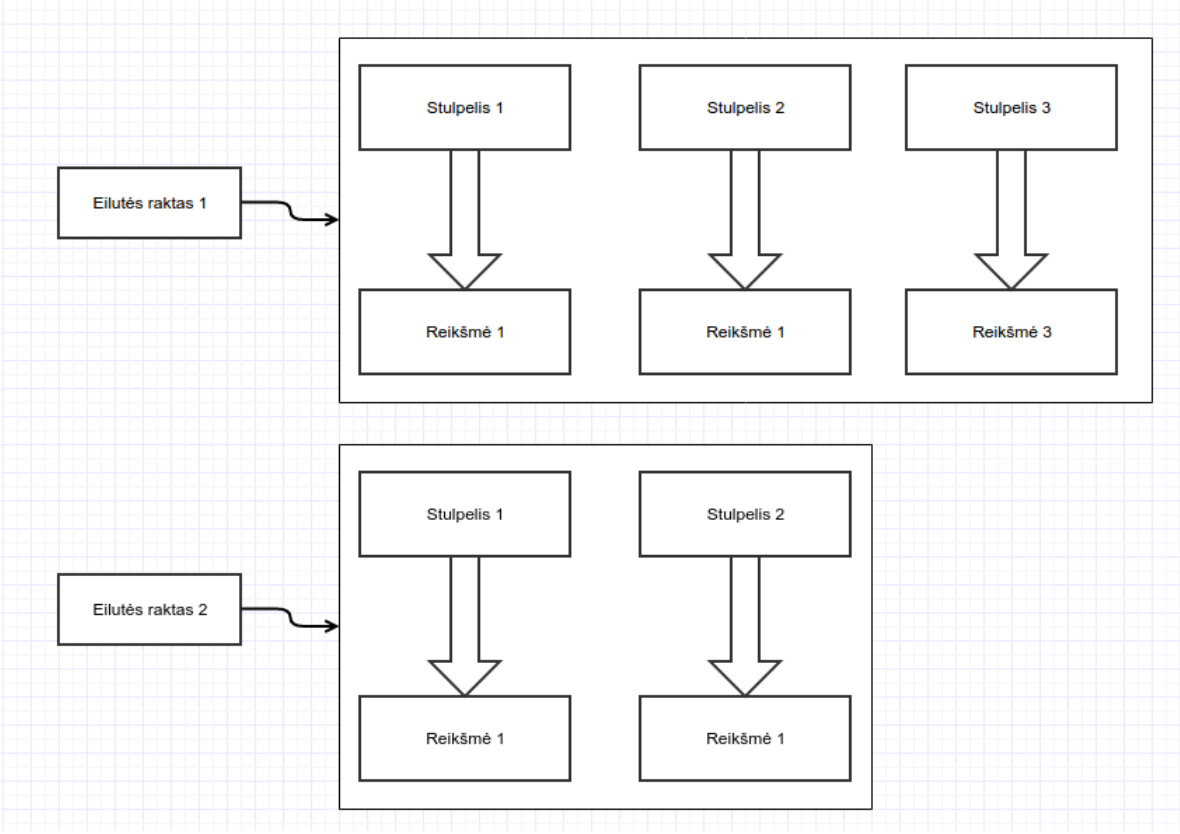
7 pav. Masyvo tipo duomenų struktūra [EJ11]

Tačiau, norint atlikti paiešką tokio tipo duomenų struktūroje tektų eiti per kiekvieną įrašą ir tikrinti jo atitikimą užklausai. Taip pat, iškiltų bėdų, kuomet kažkurios reikšmės nebūtų, kadangi tektų pildyti masyvą tuščiais elementais tam, kad galėtume žinoti, kur kuri reikšmė padėta. Taip pat tektų palaikyti dokumentaciją, kas yra kurioje masyvo vietoje ir po kiekvieno pakeitimo ją pildyti. Taigi, sekantis žingsnis yra pridėti vardus reikšmėms pridedant po dar vieną dimensiją.



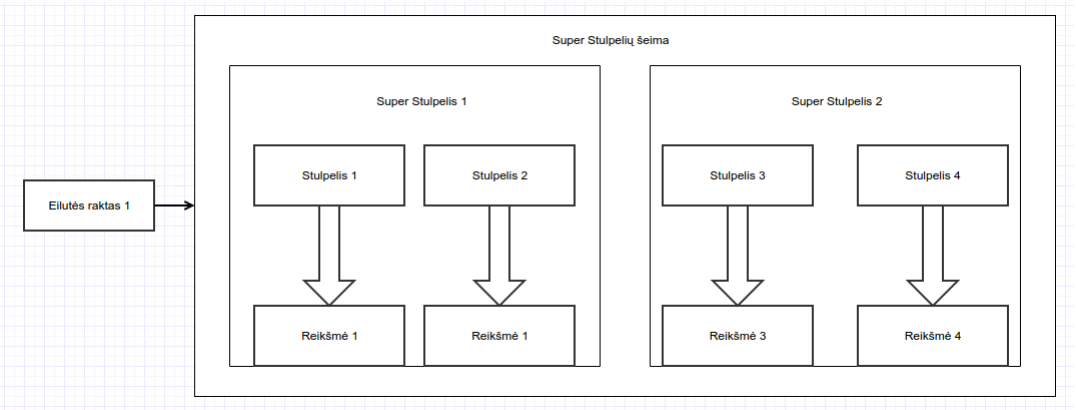
8 pav. Dviejų dimensijų duomenų saugojimas [EJ11]

Tai išspręstų aukščiau minėtas problemas, kadangi mūsų masyvo reikšmės turėtų pavadinimus. Tiesa, ši struktūra kol kas veikia tik vienam įrašui, tai reiškia, kad mes galėtume turėti tik vieną, sakykim, žmogaus įrašą. Tuo tarpu mūsų tikslas yra saugoti daug įrašų vienodoje struktūroje ir juos analizuoti. Taigi, reikia kažko, kas leistų mums grupuoti stulpelių reikšmes kartu ir leisti jas laisvai pasiekti. Reikalingas identifikatorius, kuris leistų grupę stulpelių laikyti visuma – eilute. Būtent tai yra “Cassandra” duomenų bazės pats didžiausias skirtumas. Ji save apibūdina kaip eilutiniam duomenų saugojimui orientuotą, tačiau kolonėlinę, duomenų bazę. Taigi, įgyvendinę minėtą pakeitimą galėtume gauti vieną eilutę turinčia informaciją apie visus stulpelius. Tai atitiktų tokią struktūrą paveiksliuke žemiau.



9 pav. Stulpelių grupavimas ir priskyrimas eilutėms [EJ11]

Toks duomenų modelis leidžia duomenis įsivaizduoti šiek tiek panašiai kaip ir reliaciniame modelyje, kadangi mes patogiai galime pasiekti visą eilutę, tačiau lygiai taip mes galime pasiekti ir visą stulpeliuose esančią informaciją ignoruojant eilutes.
Be to, “Cassandra” leidžia stulpelius grupuoti į reikšmiškai susijusias stulpelių grupes. Taigi, galutinė duomenų modelio schema atrodo štai taip.



10 pav. Pilna duomenų, esančių ”Cassandra” duomenų bazėje, struktūra [EJ11]

Štai čia yra bene didžiausias “Cassandra” duomenų bazės “kabliukas”. Duomenys esantys skirtingose stulpelių grupėse negalės būti pasiekti viena užklausa, todėl labai svarbu įterpiant duomenis į duomenų bazę jau turėti viziją ką su jais vėliau bus norima atlikti. Kita šios savybės ypatybė – tai leidžia užtikrinti greitesnį duomenų įrašymą (kadangi žinoma ir tiksli vieta į kurią duomenys

yra įrašomi ir susiję laukai, kuriuos gali tekti atnaujinti) bei greitesnį duomenų skaitymą, kuomet skaitomi duomenys yra vienoje stulpelių grupėje (kadangi jie saugomi vienoje vietoje diske ir nereikia atlikti pertraukimų skaitant informaciją) [AP09]

1.4.2. Blokai (clusters)

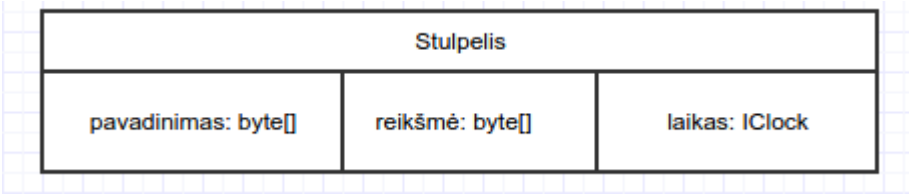
"Cassandra" skirta dirbti su keliais serverių blokais vienu metu, todėl tai nėra geriausias pasirinkimas, jei ketinama dirbti su mažu kiekiu duomenų ir vienu bloku. "Cassandra" daug dėmesio skyrė patogiam, patikimam ir lanksčiam klasterių pridėjimui ir atėmimui, kadangi ji skirta dirbti su dideliais duomenų kiekiais, todėl naudojantis tik vienu bloku bus neišnaudojama didelė dalis "Cassandra" duomenų bazės našumo galimybių. [EJ11]

1.4.3. Stulpelių šeimos

Iš esmės, stulpelių šeimos primena reliacinio duomenų modelio lenteles, tačiau tai nėra visiškai tas pats. Pagrindinis skirtumas yra tas, kad reliacinės duomenų bazės lentelėj visų laukų pavadinimai yra žinomi, kai tuo tarpu stulpelių šeimoj bet kuriuo metu galima pridėti naują stulpelį. Rašant duomenis į stulpelių šeimą galima rašyti tiek į vieną stulpelį, tiek į visus, taip pat galima rinktis ir eilutes į kurias rašyti.
Pagrindinis stulpelių šeimos privalumas – jie leidžia išsitraukti kelių stulpelių reikšmes viena užklausa, kadangi diske jie guli vienas šalia kito. [EJ11]

1.4.4. Stulpeliai

Tai pagrindinė duomenų struktūros dalis. Stulpelį sudaro trys dalys – vardas, reikšmė ir laiko reikšmė. Skirtingai nei reliacinėse duomenų bazėse, stulpeliai gali būti kuriami duomenų bazės kliento bet kuriuo veikimo metu. [EJ11]



11 pav. Stulpelio struktūra [EJ11]

Nuo "Cassandra" 0.6 versijos, eilutės, priklausančios tai pačiai stulpelių šeimai, saugomos vienoje vietoje diske, tam, kad būtų greičiau pasiekiamos.

1.4.5. Super stulpeliai

Super stulpeliai skiriasi nuo paprastų stulpelių tuo, kad jų reikšmė nėra baitų masyvas. Jų reikšmė sudaryta iš paprastų stulpelių. Jų pagrindinė funkcija yra grupuoti stulpelius, kurie užklausose būna kartu. Tai reiškia, tiems stulpeliams, kuriuos iš duomenų bazės traukiame vienu metu yra gerai priklausyti super stulpeliams arba stulpelių šeimoms.

Svarbu žinoti, kad “Cassandra” neindeksuoja stulpelių esančių super stulpeliuose. Tai reiškia, kad užkrovus super stulpelį į atminį, visi jam priklausantys stulpeliai užkraunami kartu. [EJ11]

1.4.6. Antriniai ir kombinuoti indeksai

“Cassandra” duomenų bazė neturi antrinių indeksų, tai reiškia, kad norint galėti ieškoti duomenų pagal kitą stulpelio reikšmę reikia turėti papildomą super stulpelį, kuris turėtų tą vienintelį stulpelį kartu su nuoroda (ID) į originalią eilutę. [EJ11]

1.4.7. Apibendrinimas

Iki šiol dirbus su reliacinėmis duomenų bazėmis galbūt nebuvo tekę susidurti su problemomis kurios atsiranda dirbant su dideliais duomenų kiekiui, tačiau teoriniame lygmenyje apsvarsčius, atrodo, kad esant poreikiui pereiti prie kolonėlinės duomenų bazės “Cassandra” atrodo kaip iš ties geras sprendimas vien dėl to, kad ji yra tarsi vidurys, tarp visiškai kolonėlinių duomenų bazių skirtų tik duomenų skaitymui bei reliacinių duomenų bazių. Taip, ji neturi lentelių sąjungų, neturi laikinų lentelių (nors stulpelių šeimos tai šiek tiek primena), neturi tranzakcijų, tačiau iš esmės visas duomenų modelis yra šiek tiek panašus į realiacinį. Bene didžiausias skirtumas – duomenis reikia saugoti denormalizuotus, tai yra, be jokių reliacinių sąsajų tarp stulpelių šeimų. Žinoma, saugoti nuorodas į susijusias eilutes (sudarytas iš stulpelių) galima, tačiau tokių funkcijų kaip, pavyzdžiui, duomenų vientisumo užtikrinimas duomenų bazės lygyje (ištrynus įrašą A pašalinti visus įrašus B su nuorodomis į jį) “Cassandra” duomenų bazė tiesiog nepalaiko. Vietoje to ji siūlo išties greitai veikiančią, patogią naudoti ir lengvai plečiamą duomenų bazę, kuri skirtingai nuo kitų tokio tipo duomenų bazių pasižymi ne tik greičiu, skaitant duomenis, tačiau ir juos rašant bei savo modelio panašumu į reliacinį [AP09] [EJ11]

1.5. ”Druid” duomenų bazė

“Druid” duomenų bazė yra kolonėlinė duomenų bazė skirta duomenų analizavimui. Problema paskatinusi “Druid” atsiradimą – itin lėtas itin didelių .log plėtinio failų skaitymas. Tai vis dar nauja duomenų bazė (šiuo metu jos versija tėra 0.9.0), tačiau jos našumo rodikliai yra vienareikšmiškai geriausi, kuomet užduoties tikslas yra išanalizuoti duomenis. [FEX+12]

1.5.1. Duomenų saugojimo formatas

Visi duomenys saugojant yra dalijami į tris skirtingus tipus:

Įrašo laiko duomuo. Kiekvienas įrašas (stulpelių rinkinys) turi turėti bent vieną stulpelį, kurio formatas bus kažkokio tipo data, kadangi tai yra pagrindinė ašis, kurios pagrindu duomenys yra saugomi.

Dimensijų stulpeliai. Tai tokie stulpeliai, kurie skirti duomenims skaidyti. Pagal juos vėliau duomenys yra filtruojami bei grupuojami.

Metrikų stulpeliai. Tai tie stulpeliai, su kuriais reikės atlikti skaičiavimus (pavyzdžiui vidurkius, sumas, maksimalias reikšmes) Svarbu žinoti, kad nors indeksuojant duomenis į duomenų bazę ir reikia nurodyti dimensijas bei preliminarias metrikas, užklausų vykdymo metu galima tai ignoruoti ir pakeisti reikšmes, tačiau tokiu atveju nukentės užklausos atlikimo laikas.

Pasak, F.Yang , analitinių duomenų savybė yra ta, kad vienas pats duomuo iš esmės nelabai kam įdomus, todėl buvo priimtas sprendimas jau įvedant duomenis į duomenų bazę nurodyti kokia preliminari bus tų duomenų reikšmė ir paskirtis. Tai yra pirmojo lygio agregacija, kurią F.Yang prilygina pseudokodui.
Toks duomenų išankstinis agregavimas sprendžia ir kitą problemą – dauguma duomenų iš esmės nesudaro jokios reikšmės ir galbūt net neverti saugoti. Pavyzdžiui, jei norime skaičiuoti tik sumą, tai pavieniai rezultatai mums nėra svarbūs, todėl gali būti net nesaugomi duomenų bazėje, taip sutaupant disko vietos. [FEX+12]

1.5.2. Duomenu skaidymas

Duomenys "Druid" duomenų bazėje skaidomi segmentais, kurie yra dalijami pagal laiką. Pavyzdžiui, galima skaidyti duomenis į segmentus po savaitės laiko informaciją, taip prireikus konkretaus rėžio informacijos ji visa bus vienoje vietoje ir taip sumažės laikas, per kurią ji bus gaunama. Be to, segmentai gali būti sunaikinami praėjus tam tikram (nurodytam) laikui.
Vykdant užklausas tik reikalingi duomenų segmentai yra kviečiami, o be to, tik reikalingų stulpelių informacija yra ištraukiama. Taip užtikrinama, kad bus nuskaitytas kuo didesnis reikalingos skaityti informacijos kiekis.

1.5.3. Duomenu įrašymas

Duomenis į "Druid" duomenų bazę galima įrašyti dvejais būdais:

- Užkrauti duomenų rinkinį (pavyzdžiui iš JSON arba CSV failo). Šis būdas šiuo metu veikia stabiliai ir korektiški rezultatai yra garantuoti.

- Realaus laiko duomenų užkrovimas (taip, kaip veikia dauguma reliacinių duomenų bazių). Kol kas "Druid" kūrėjai, pasak F.Yang, negali užtikrinti, kad visi duomenys yra pridedami sėkmingai. Siūlomas sprendimas norintiems naudoti realaus laiko duomenų įterpimą – periodiškai importuoti ir duomenų rinkinius, o realaus laiko duomenis laikyti kaip apytikslius iki tol, kol bus pakeltas stabilumas. [FEX+12]

1.5.4. Užklausų vykdymas

Užklausos į "Druid" duomenų bazę pateikiamos HTTP POST metodu, siunčiant JSON failą, tačiau, yra įvairių alternatyvų, kurios leidžia užklausas rašyti kalbomis, itin panašiomis į SQL. Reiktų žinoti, kad užklausos "Druid" bloke keliauja per servisus, kurių kiekvienas atlieka konkrečią užduotį.

1.5.4.1. Istorinių duomenų segmentų servisas

Servisas skirtas istorinių duomenų pasikrovimui atmintyje ir užklausų susijusių su jais duomenų vykdymui. Pavyzdžiui, jei neseniai daryta užklausa tikrino visus vartotojus, tai tas segmentas išliks atmintyje ir per nustatytą laiko tarpą jo iš naujo išsitraukinėti neteks.

1.5.4.2. ”Broker” servisas

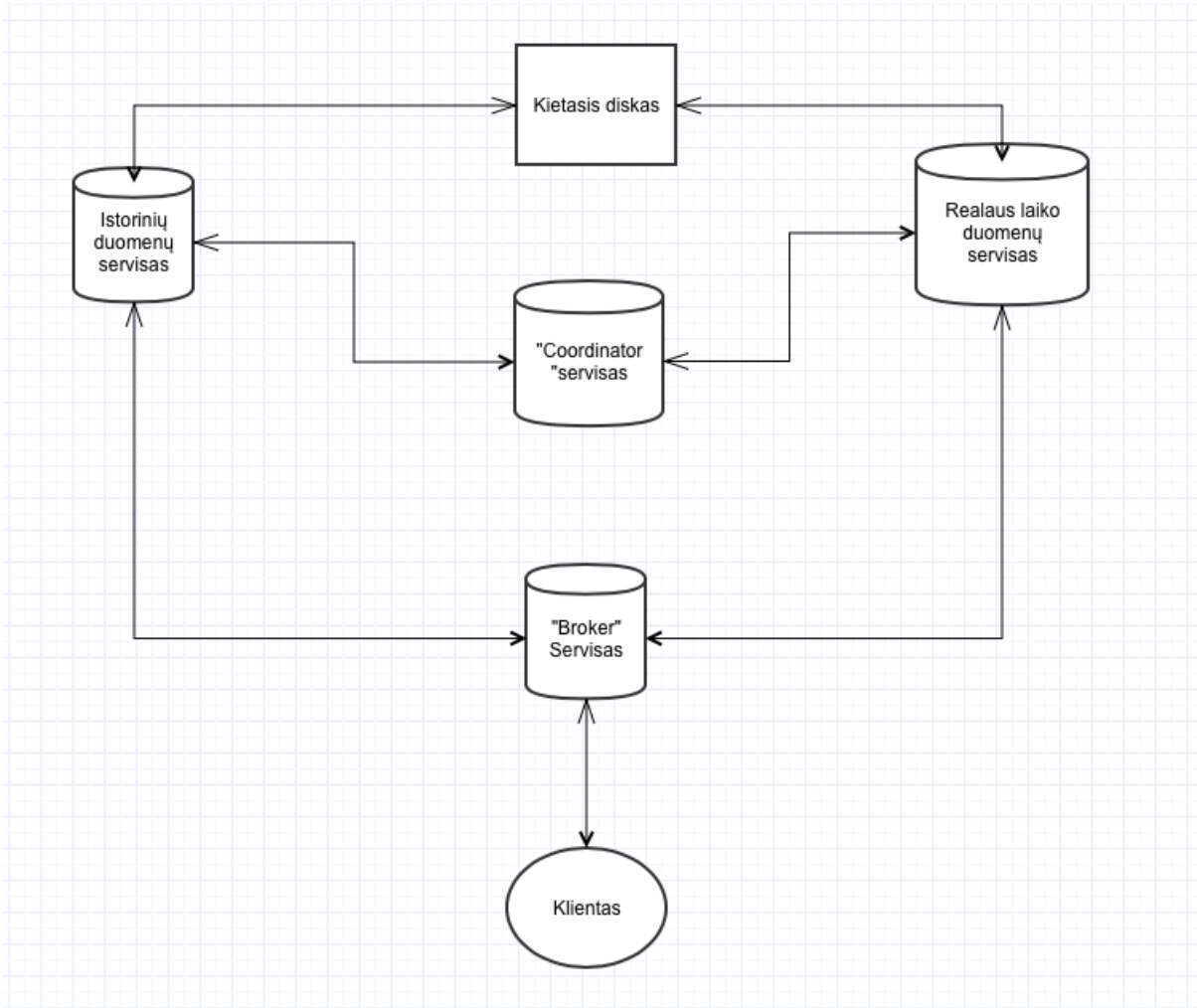
Tai servisas, kuris atlieka maršrutizatoriaus funkciją. Jis surenka informaciją iš visų likusių servisų ir pateikia ją klientui taip, kad rezultatas būtų kuo greitesnis

1.5.4.3. ”Coordinator” servisas

Servisas valdantis istorinių duomenų servisą. Jis nurodo, kuomet vienus duomenis atmintyje jau reikia pakeisti kitais.

1.5.4.4. Realaus laiko duomenų servisas

Tai servisas skirtas duomenų ištraukimui bei įdėjimui praleidžiant visus likusius servisus. Į jį galima kreiptis tiesiogiai.



12 pav. ”Druid” duomenų bazės užklausų vykdymo mechanizmo schema [FEX+12]

Norint užtikrinti, kad “Druid” duomenų bazė veikia taip, kaip ji save pristato, reikia turėti bent du skirtingus blokus (tai užtikrintų, kad yra bent du veikiantys kiekvieno tipo servisai).

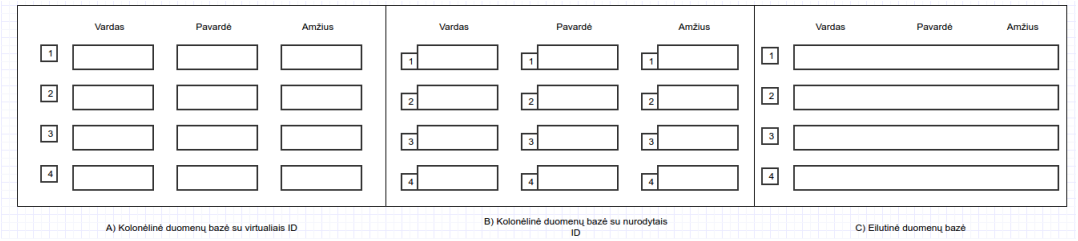
1.6. Abiejų kolonėlinių duomenų bazių palyginimas

Nors abi duomenų bazės ir yra kolonelinės ir skirtos greitam duomenų skaitymui/agregavimui/analizavimui, tačiau “Cassandra” duomenų bazė šiuo atveju stengiasi būti universalesnė ir būti panašesnė į reliacinę. Tai, galbūt, ir yra gerai, kadangi duomenų bazės projektavimo klausimas tampa mažesne problema, kuomet ketinama keisti naudojamą duomenų bazę, tačiau kita vertus, atrodo, jog tikroj paskirtis, dėl kurios buvo pradėtas naudoti principas, kuomet duomenys saugomi stulpeliais, o ne eilutėmis, yra geriau įgyvendinta “Druid” duomenų bazėje. Be viso to, “Druid” duomenų bazė pasižymi tuo, kad ji turi nemažai jau įgyvendintų duomenų agregacinių funkcijų, kai tuo tarpu “Cassandra” duomenų bazė sako, kad šias funkcijas apsirašyti turi pats vartotojas pagal savo poreikius.

2. Duomenų bazių palyginimas

Taigi, jau išsiaiškinta, kuo pasižymi tiek reliacinės duomenų bazės su “Postgresql” priešaky tiek ir kolonėlinės su “Cassandra” ir “Druid”. Sekantis tikslas – atrasti pagrindinius skirtumus tarp duomenų bazių ir nuspręsti, kurie iš tų skirtumų yra esminiai bei svarbūs turimos praktinės užduoties atlikimui.

2.1. Duomenų saugojimo struktūros skirtumai



13 pav. Trys skirtingi duomenų saugojimo tipai [APS+12]

“Postgresql” duomenys saugomi tokiu principu, koks atvaizduojamas c lentelėje, kai tuo tarpu “Cassandra” duomenų bazėje duomenys saugomi struktūra pavaizduota b dalyje, o “Druid” – a.

Esminis skirtumas tarp duomenų saugojimo tipų yra pastebimas skaitant duomenis. Jeigu mums reikia pasiimti vienos eilutės kelias reikšmes, tuomet kolonėlinės duomenų bazės užtruks ilgiau ir ilgiau, kuo daugiau stulpelių toje eilutėje bus, kadangi kiekvienam stulpeliui bus daroma papildoma užklausa (nebent naudojant “Cassandra” duomenų bazė tie stulpeliai bus super stulpelyje). Tuo tarpu reliacinėje duomenų bazėje būtų įvykdyta viena užklausa. Kadangi užklausa yra skaitymas iš disko, daugiau atskirų skaitymų reiškia lėtesnį veikimą.
Taip pat skiriasi ir duomenų kompresija. Kuomet duomenys saugomi atskiruose stulpeliuose juos galime skirtingai suspausti (priklausomai nuo jų tipo), kas, vėlgi, padidina skaitymo greitį.[D.E10]

2.2. Komandų skirtumai

- Tiek “Cassandra” duomenų bazė tiek ir “Druid” neturi užklausos duomenims atnaujinti. Norint atnaujinti duomenis tenka rašyti naują reikšmę į tą patį stulpelį. Jeigu bandysime įterpti keletą skirtingų įrašų į skirtingus stulpelius, tuomet egzistuojantys bus pakeisti, o nauji pridėti.

- Tiek “Cassandra” tiek ir “Druid” duomenų bazės užtikrina, kad įrašymo operacija yra atominė, tačiau skirtingai nei Postresql duomenų bazė ji negali užtikrinti izoliavimo, kuomet atnaujinamos visos tam tikro stulpelio reikšmės.

- “Postgresql” duomenų bazė suteikia galimybes operacijas atlikti tranzakcijose, kai tuo tarpu abi nagrinėtos kolonėlinės duomenų bazės sako, kad tai tektų atlikti kliento pusėje.

- “Postgresql” leidžia turėti kelias reikšmes su vienodu raktu (jei jis nėra unikalus), kai tuo tarpu “Cassandra” duomenų bazė tokios galimybės neturi. Duomenys būtų tiesiog perrašyti. Tuo tarpu “Druid” duomenų bazė leidžia turėti kiek norima vienodų reikšmių vienam raktui, tačiau atliekant agregacines funkcijas būtina nurodyti, ar naudoti tik unikalias reikšmes ar visas.

- “Postgresql” iš ties greitai rašo duomenis į duomenų bazę, tačiau “Cassandra” šioje vietoje neatsilieka. Taip yra todėl, kad įrašymas į duomenų bazę kolonėlinėje architektūroje “Cassandra” nereikalauja skaitymo iš disko, o įrašymo vieta žinoma iš anksto pagal stulpelių grupes. Tačiau sudėtinga eilučių struktūra įgyvendinta kartu su stulpeliais nors ir prideda greičio įrašant duomenis, tačiau šiektiek sumažiną pačios kolonėlinės duomenų struktūros privalumus. “Druid” duomenų bazė yra orientuota vienareikšmiškai į duomenų skaitymo operacijas.

- Tiek “Postgresql” tiek ir “Cassandra” užtikrina, kad duomenys naudojami užklausos metu yra validūs. Tai reiškia, kad jei esama papildomų duomenų bazės serverių, sutikrinama, ar nėra naujos reikšmės su naujesne įrašymo data. Jei tokia yra – imama pati naujausia reikšmė. Tiesa, “Cassandra” leidžia duomenų vientisumo lygį konfiguruoti. Tuo tarpu “Druid” duomenų bazę nurodo, kad pasitikėti galima tik duomenimis, kurie yra sukeliami ne realiu laiku. Tai reiškia, kad kurį laiką, duomenų kėlimas realiu laiku dar liks orientacinių duomenų šaltiniu.

[EJ11] [Gro11] [FEX+12]

2.3. Apibendrinimas

Kadangi reliacinių duomenų bazių teikiami privalumai (indeksai, lentelių jungimai) iš esmės nėra aktualūs mūsų analitinio tipo uždaviniui, taip pat kaip ir rašymo į duomenų bazę greitis, todėl galima teigti, kad reliacinės duomenų bazės nepasiūlo nieko, kas priverstų rinktis jas, vietoje kolonėlinių. Tuo tarpu kolonėlinės duomenų bazės siūlo didelį greitį traukiant vieno stulpelio reikšmes (ko, užduočiai ir reikės, kadangi vesime vidurkius). Dar viena svarbi savybė – “Postgresql” plečiasi vertikaliai, kai tuo tarpu “Cassandra” ar “Druid” duomenų bazės auga Horizontaliai. Tai ir yra itin svari šių duomenų bazių stiprybė – galima pridėti tiek papildomų blokų, kiek tik jų reikia, o tai reiškia, kad galėsime dirbti su praktiškai bet kokio dydžio duomenimis. Kadangi uždavinys analitinio tipo ir pagrindinis tikslas yra įrodyti, kad būtent su tokiais duomenimis dirbti ir yra patogiausia naudojant kolonėlines duomenų bazes, todėl darome išvadą, kad prielaida buvo teisinga. Bandysime tai įrodyti ir praktiniu darbu.

3. Praktinė užduotis

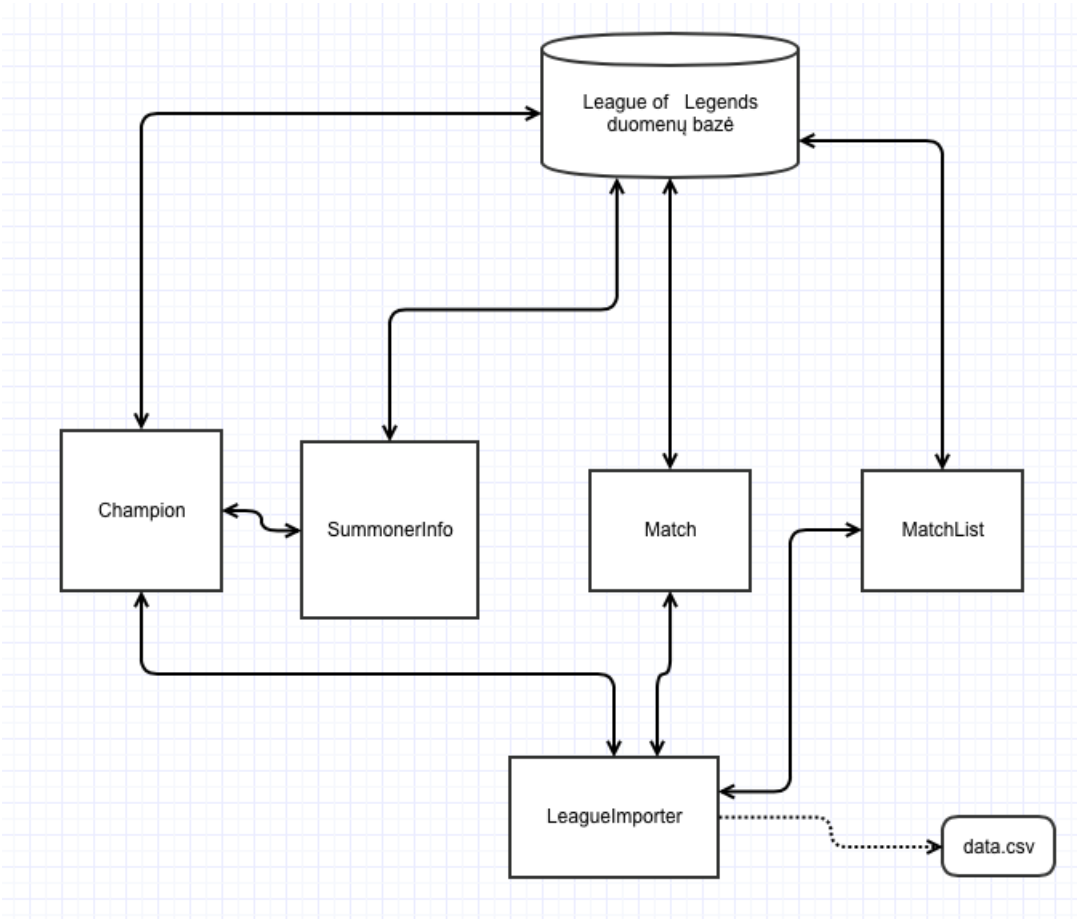
Praktinė užduotis skirta patikrinti prieitą išvadą, jog kolonėlinio tipo duomenų bazės yra geriausias duomenų saugojimo būdas, kuomet jų yra daug (bent jau šimtai tūkstančių įrašų) bei juos ketinama agreguoti/analizuoti. Kadangi buvo poreikis surasti realių duomenų, kuriuos būtų galima analizuoti, buvo pasirinkta naudotis žaidimo pateikiamais statistiniais duomenimis. Žaidimas vadinasi "League of Legends". Tai strateginis realaus laiko žaidimas, kurio esmė – komandiškai įveikti kitą komandą ir nugriauti jos tvirtovę. Jis pasirinktas todėl, kad suteikia prieigą prie itin didelio kiekio žaidimų istorijos duomenų, kuriuos išanalizavus įvairiais būdais (sudėties, vidurkių) galima prieiti tam tikrų išvadų. Žaidime egzistuoja dvi komandos po 3 arba 5 žmones (tik po vienodai). Taigi, iš viso – arba 6 arba 10 žaidėjų viename žaidime. Žaidimo tikslas – nugalėti priešininkų komandą. Trumpai apie patį modelį – žaidime yra kaunamasi dėl bokštų, tvirtovių ir kitų strateginių resursų. Kiekvienas iš 10 žaidėjų valdo vieną veikėją (kiekvienas veikėjas unikalus ir daro kažką neįprasto). Žaidėjo pasirodymas vertinamas pagal tai, kiek priešininkų jis nugalėjo, kiek kartų padėjo komandos draugams nugalėti priešininkus ir kiek kartų pats buvo nugalėtas. Pasirodymo indekso formulė – (Nugalėjimai + Pagelbėjimai) / Mirčių skaičiaus. Užduoties tikslas – įrodyti, kad analizuojant tokio tipo duomenis dideliais kiekiais, kolonėlinės duomenų bazės darbą atlieka greičiausiai, bei pagaminti tai įliustruojančią sistemą, leidžiančią analizuoti praeities žaidimų istorijos rezultatus.

Užduotis susideda iš trijų dalių:

- Susigeneruoti kliento aplikaciją, kuri iš "RIOT GAMES API" išsitrauktų žaidėjo ir visų su juo žaidusių žaidėjų statistinius duomenis.
- Atlikti duomenų bazių testus ir išsiaiškinti, kuri iš duomenų bazių geriausiai tvarkosi su tokio tipo duomenimis ir pavyzdinėmis užklausomis.
- Pasirinkus geriausią variantą – pagaminti mažos apimties sistemą, leidžiančią matyti kaip atrodo galutinis rezultatas.

3.1. Kliento aplikacijos generavimas

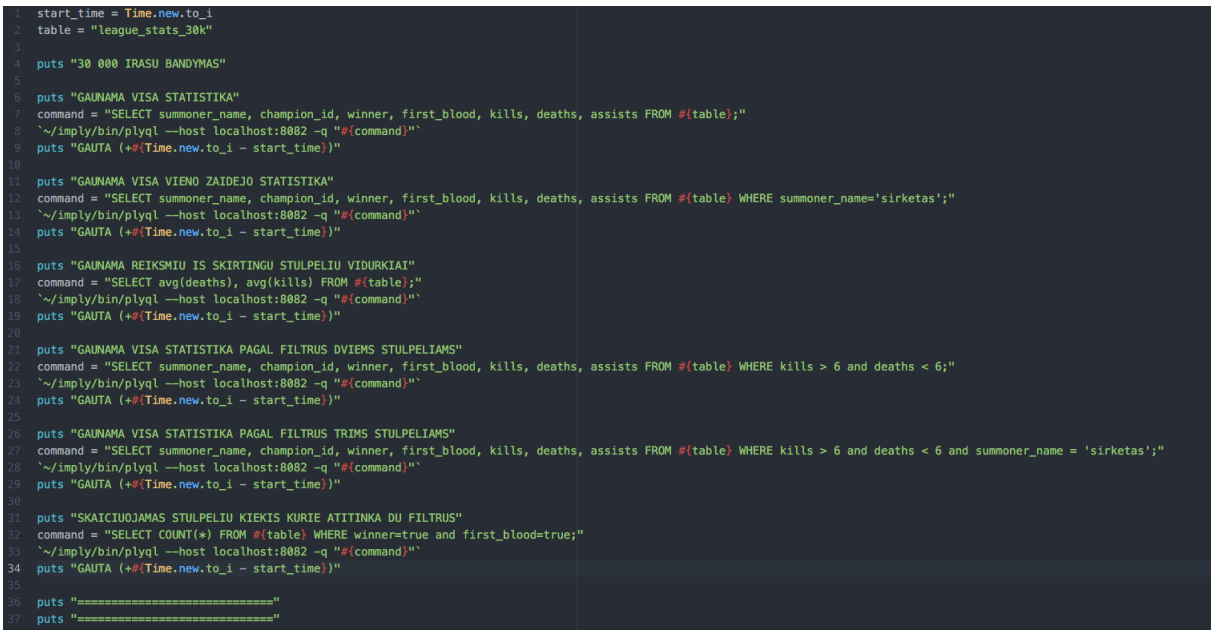
Aplikaciją kurta su Ruby programavimo kalba, pasinaudojant ApiSmith biblioteka, skirta daryti HTTP užklausas. Buvo parašytos keturios klasės – SummonerInfo (Atvaizduojanti žaidėjo vardą bei kitus identifikacinius duomenis), MatchList(Atvaizduojanti žaidėjo sužaistų partijų sąrašą), Match (Atvaizduojanti konkretaus žaidimo informaciją su visų žaidėjų pasirodymais (6 arba 10), bei Champion, skirta atvaizduoti veikėjams, už kuriuos buvo žaista. Implementacijas žiūrėti prieduose.
Šias klases naudojo LeagueImporter klasė, kurioje buvo nurodoma kiek iš kiekvieno žaidėjo istorijos išsitraukti įrašų. Gautus duomenis įrašius į CSV failą, juos vėliau buvo galima suimportuoti į skirtingas duomenų bazes.



14 pav. Duomenų importavimo schema

3.2. Duomenų bazių testavimas su skirtingo dydžio duomenų failais

Tam, kad duomenys būtų ištestuoti kuo panešesnėmis sąlygomis, jokios papildomos bibliotekos nebuvo naudotos. Duomenys buvo sukelti standartiniu būdu – užkraunant iš CSV failo (“Druid” duomenų bazėje kartu ir nurodant galimus reikšmių panaudojimo būdus)
Į visas duomenų bazes buvo atliktos užklausos, kurių rezultatą galima matyti žemiau. Skliausteliuose rašomas laikas, kiek truko užklausos vykdymo laikas nuo pat pirmosios užklausos pradžios.



```
start_time = Time.new.to_i
table = "league_stats_30k"

puts "30 000 IRASU BANDYMAS"

puts "GAUNAMA VISA STATISTIKA"
command = "SELECT summoner_name, champion_id, winner, first_blood, kills, deaths, assists FROM #{table};"
`~/imply/bin/plyql --host localhost:8082 -q "#{command}"`
puts "GAUTA (+#{Time.new.to_i - start_time})"

puts "GAUNAMA VISA VIENO ZAIDEJO STATISTIKA"
command = "SELECT summoner_name, champion_id, winner, first_blood, kills, deaths, assists FROM #{table} WHERE summoner_name='sirketas';"
`~/imply/bin/plyql --host localhost:8082 -q "#{command}"`
puts "GAUTA (+#{Time.new.to_i - start_time})"

puts "GAUNAMA REIKSMIU IS SKIRTINGU STULPELIU VIDURKIAI"
command = "SELECT avg(deaths), avg(kills) FROM #{table};"
`~/imply/bin/plyql --host localhost:8082 -q "#{command}"`
puts "GAUTA (+#{Time.new.to_i - start_time})"

puts "GAUNAMA VISA STATISTIKA PAGAL FILTRUS DVIEMS STULPELIAMS"
command = "SELECT summoner_name, champion_id, winner, first_blood, kills, deaths, assists FROM #{table} WHERE kills > 6 and deaths < 6;"
`~/imply/bin/plyql --host localhost:8082 -q "#{command}"`
puts "GAUTA (+#{Time.new.to_i - start_time})"

puts "GAUNAMA VISA STATISTIKA PAGAL FILTRUS TRIMS STULPELIAMS"
command = "SELECT summoner_name, champion_id, winner, first_blood, kills, deaths, assists FROM #{table} WHERE kills > 6 and deaths < 6 and summoner_name = 'sirketas';"
`~/imply/bin/plyql --host localhost:8082 -q "#{command}"`
puts "GAUTA (+#{Time.new.to_i - start_time})"

puts "SKAICIUOJAMAS STULPELIU KIEKIS KURIE ATITINKA DU FILTRUS"
command = "SELECT COUNT(*) FROM #{table} WHERE winner=true and first_blood=true;"
`~/imply/bin/plyql --host localhost:8082 -q "#{command}"`
puts "GAUTA (+#{Time.new.to_i - start_time})"

puts "=========================="
puts "=========================="
```

15 pav. Atliekamų užklausų pavyzdžiai

3.2.1. Rezultatai

Pateikiama visų užklausų trukmės suma sekundėmis (neįskaičiuojant duomenų bazių pildymo duomenimis laiko)

1 lentelė. Užklausų atlikimo rezultatai

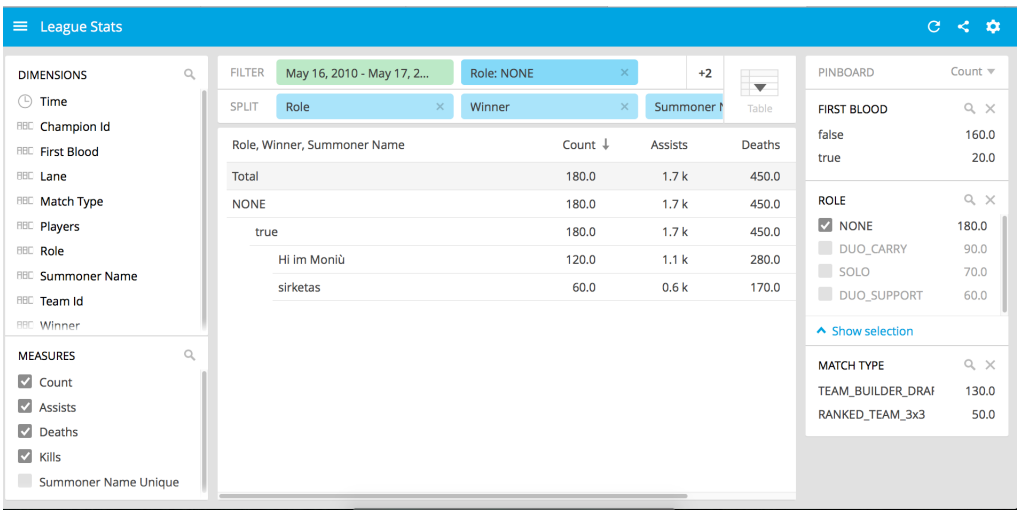
Duomenų bazė	30 000 įrašų	300 000 įrašų	3 000 000 įrašų
PostgreSQL	2s	6s	23s
Cassandra	6s	13s	20s
Druid	1s	1s	3s

3.2.2. Rezultatų išvada

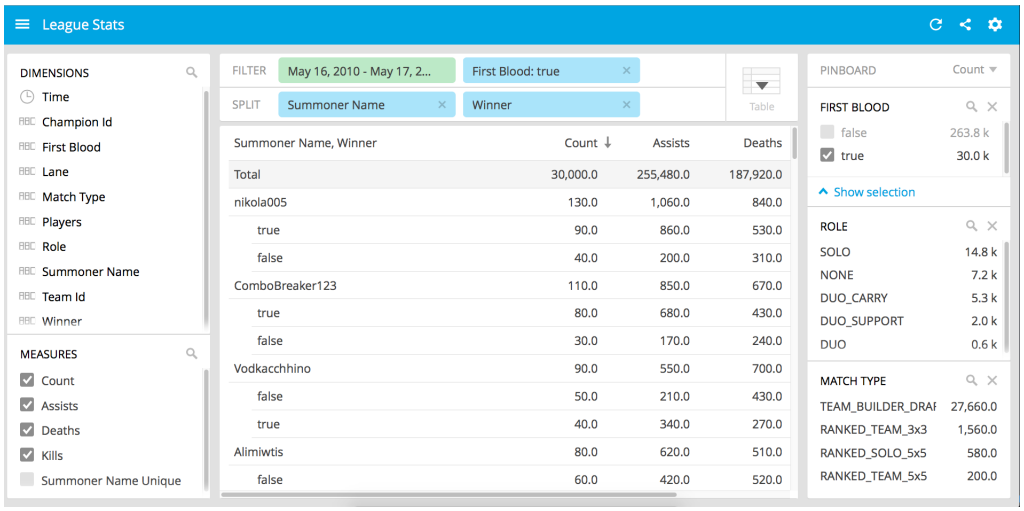
Kadangi neįskaičiuojamas laikas, kuris truko užpildyti duomenų bazes duomenimis, todėl realus duomenų apdorojimo laikas šiek tiek skiriasi, tačiau aiškiai matome, kad jei turimas didelis duomenų kiekis yra iš anksto paruoštas ir norima jį analizuoti pagal daug kriterijų – “Druid” duomenų bazė yra nepralenkiama. Grįžtant prie darbo temos – galima matyti, kad “Cassandra” duomenų bazė persvėrė rezultatą tik esant tikrai dideliems duomenimis. Kita vertus, “Cassandra” nėra visiškai kolonėlinės duomenų bazės standartus atitinkanti duomenų bazė. Tuo tarpu “Druid” yra stipriai optimizuotas būtent duomenų analizavimui, todėl tokie puikūs rezultatai, iš tiesų, nestebina.

3.3. Praktinės užduoties įgyvendinimas

Pasinaudojus Pivot ir UI javascript bibliotekomis, kartu su “Druid” duomenų baze, pavyko pasileisti veikiančią sistemą, kuri leidžia visus turimus duomenis filtruoti pagal begalę aspektų tiesiai iš naršyklės lango, o rezultatai grąžinami akimirksniu.



16 pav. Sistemos atvaizdas



17 pav. Sistemos atvaizdas su daugiau duomenų

Sistema jau yra tinkama naudoti, kadangi iš analizuojamų duomenų galima išsivesti tam tikras prielaidas. Tarkim, žaidžiant vienoje pozicijoje yra žymiai didesni šansai laimėti, arba tai, kad bloga pradžia nebūtinai reiškia pralaimėtą žaidimą, kadangi net 263 000 iš 290 000 ištirtų žaidimų buvo laimėti, kuomet pradžia buvo pralaimėta.

Rezultatai ir išvados

Šio darbo metu ištyrinėjau tris duomenų bazes. Vieną – pilnai reliacinę, kitą – kolonėlinę, tačiau besiorientuojančią į eilutines duomenų bazes (turinčią daug bendrų principų) ir paskutinę – analitiniams duomenims skirtą kolonėlinio tipo duomenų bazę. Pagal tyrimo rezultatus ir tai, kaip sekėsi atlikti praktinį darbą, galiu teigti, kad duomenų įterpimas greičiausiai veikia reliacinėse duomenų bazėse ir kuo daugiau duomenų įterpiama tuo skirtumas labiau jaučiamas. Tačiau, grįžtant prie užduoties tikslo – išsiaiškinti kokie yra principiniai skirtumai tarp kolonėlinių ir reliacinių duomenų bazių, kuomet norima saugoti ir analizuoti didelius kiekius – manau, kad praktinio darbo rezultatai kalba už save. Kadangi analitinių duomenų įprastai būna labai daug, todėl galime laikytis prielaidos, kad mūsų bandytas 3 000 000 duomenų imties variantas tikrai nėra didžiausias įmanomas, o dirbant su didesniais duomenimis skirtumas tar reliacinių ir kolonėlinių duomenų bazių tik didėtų. Taip pat, darbo metu pavyko išsiaiškinti, kad kolonėlinio tipo duomenų bazėse, kuomet planuojama duomenis analizuoti, yra laikomasi praktikos, kad duomenis bent šiek tiek paruošti analizavimui jau juos įterpiant. Tai reiškia, kad teisinga duomenų struktūra, teisingi pirminiai raktai ar agregacinių funkcijų atributai, esant didesniam duomenų kiekiui, rezultatus gerins vis pastebimiau. Svarbu žinoti, kad esant kraštutiniems atvejams (pavyzdžiui, nedideliam analitinių duomenų kiekiui, arba, dideliam, tačiau tokiam, kurio skaidyti neįmanoma ir visuomet reikalinga visa eilutės informacija) reliacinės duomenų bazės taip pat gali būti tinkamas variantas. O pagrindinės dvi priežastys(arba du dižiausi skirtumai), kodėl kolonėlinės duomenų bazės didelėms analitinėms užduotims yra labiau tinkamos yra tokios:

- Be papildomo kompiuterinės įrangos atnaujinimo išgaunams didesnis nuskaitomų duomenų kiekis, kadangi skaitoma tik ta informacija, kuri yra reikalinga. Taip sumažinamas brangių įvesties ir išvesties iš disko operacijų skaičius.
- Duomenys gali būti dalinami horizontaliai, kuomet reliacinės duomenų bazės auga vertikaliai. Horizontalus augimas yra praktiškai neribojamas ir tai suteikia teoriškai neribotus resursus darbui su dideliu duomenų kiekiu.

Šaltiniai

[AP09] A.Lakshman and P.Malik. Cassandra - a decentralized structured storage system. https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf. 2009.

[APS+12] D. Abadi, P.Boncz, S.Harizopoulos, et al. The design and implementation of modern column-oriented database systems. http://db.csail.mit.edu/pubs/abadi-column-stores.pdf. 2012.

[D.E10] D.Engines. System properties comparison cassandra vs. postgresql. http://db-engines.com/en/system/CassandraPostgreSQL. 2010.

[EJ11] E.Hewwit and J.Ellis. *Cassandra: the definitive guide*. O'Reilly Media, Gravenstein Highway North, Sebastopol, 2011.

[FEX+12] F.Yang, E.Tschetter, X.Léauté, et al. Druid - a real-time analytical data store. http://static.druid.io/docs/druid.pdf. 2012.

[Gro11] Postgresql Global Development Group. Postgresql 9.0 official documentation - volume iv. reference. https://books.google.lt/books?id=07a1DiW0F8wC. 2011.

[Tra13] Paper Trail. Columnar storage. http://the-paper-trail.org/blog/columnar-storage. 2013.

Priedai

```ruby
class LeagueClient
  class SummonersInfo
    include APISmith::Client

    base_uri "https://eune.api.pvp.net/"
    endpoint "api/lol/eune/v1.4/summoner"

    attr_reader :names, :ids

    def initialize(name: nil, id: nil)
      @names = Array(name) if Array(name).any?
      @ids = Array(id) if Array(id).any?
    end

    def base_query_options
      { api_key: 'b310845a-ff3b-45d0-b632-75c260697437' }
    end

    def information_by_names
      get("/by-name/#{summoner_names.join(',')}").parsed_response
    end

    def information_by_ids
      get("/#{summoner_ids.join(',')}").parsed_response
    end

    def information
      @information ||= begin
        if names.present?
          information_by_names
        elsif ids.present?
          information_by_ids
        else
          []
        end
      end
    end

    def summoners
      @summoners ||= begin
        information.values.each_with_object({}) do |item, hash|
          hash[item['name']] = item['id']
        end
      end
    end

    def summoner_names
      @names ||= information.values.map { |summoner| summoner['name'] }.uniq
    end

    def summoner_ids
      @ids ||= information.values.map { |summoner| summoner['id'] }.uniq
    end
  end
end
```

18 pav. SummonerInfo klasės implementacija

```ruby
class LeagueClient
  class MatchList
    include APISmith::Client

    base_uri "https://eune.api.pvp.net/"
    endpoint "/api/lol/eune/v2.2/matchlist"

    attr_reader :summoner_id

    def initialize(summoner_id)
      @summoner_id = summoner_id
    end

    def result
      @result ||= begin
        get("/by-summoner/#{summoner_id}").parsed_response
      end
    end

    def match_ids
      result['matches'].map { |item| item['matchId'] }
    end

    def base_query_options
      { api_key: 'b310845a-ff3b-45d0-b632-75c260697437' }
    end
  end
end
```

19 pav. MatchList klasės implementacija

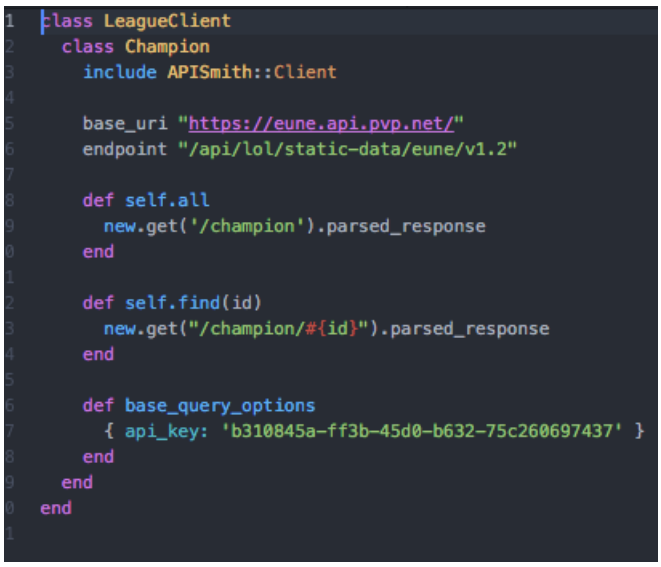
```ruby
class LeagueClient
  class Match
    include APISmith::Client

    base_uri "https://eune.api.pvp.net/"
    endpoint "/api/lol/eune/v2.2"

    def self.find(id)
      response = new.get("/match/#{id}")
      response.parsed_response unless response.code == 404
    end

    def base_query_options
      { api_key: 'b310845a-ff3b-45d0-b632-75c260697437' }
    end
  end
end
```

20 pav. Match klasės implementacija



```ruby
class LeagueClient
  class Champion
    include APISmith::Client

    base_uri "https://eune.api.pvp.net/"
    endpoint "/api/lol/static-data/eune/v1.2"

    def self.all
      new.get('/champion').parsed_response
    end

    def self.find(id)
      new.get("/champion/#{id}").parsed_response
    end

    def base_query_options
      { api_key: 'b310845a-ff3b-45d0-b632-75c260697437' }
    end
  end
end
```

21 pav. Champion klasės implementacija

```ruby
class LeagueImporter
  NUMBER_OF_MATCHES = 1;

  attr_reader :performances, :summoners, :matches, :fetched_summoners

  def initialize(summoners)
    @summoners = LeagueClient::SummonersInfo.new(name: summoners).summoners
    @performances = {}
    @matches = []
    @fetched_summoners = []
  end

  def result
    puts 'SCRIPT STARTED'
    gather_matches
    gather_performances
    performances
  end

  private

  def gather_matches
    puts 'GATHERING MATCHES STARTED'
    summoners.each_value do |summoner_id|
      next if fetched_summoners.include?(summoner_id)
      puts "GATHERING MATCHES FROM SUMMONER #{ summoner_id }"
      LeagueClient::MatchList.new(summoner_id).match_ids.first(NUMBER_OF_MATCHES).each do |match_id|
        matches << match_id unless matches.include?(match_id)
      end
      fetched_summoners << summoner_id
    end
    puts 'GATHERING MATCHES FINISHED'
  end

  def gather_performances(dive: true)
    puts 'GATHERING PERFORMANCES STARTED'

    puts matches
    matches.each do |match_id|
      next if performances.key?(match_id)
      puts "GATHERING PERFORMANCES FROM MATCH #{ match_id }"
      response = LeagueMatchReformatter.new(match_id).result
      performances[match_id] = response if response.present?
    end
    puts 'GATHERING PERFORMANCES FINISHED'

    matches.each do |match_id|
      gather_additional_performances(match_id) if dive
    end
  end

  def gather_additional_performances(match_id)
    puts "GATHERING ADDITIONAL PERFORMANCES FROM MATCH #{ match_id }"
    related_summoners = performances[match_id].each_with_object({}) do |performance, hash|
      hash[performance[:summoner_name]] = performance[:summoner_id] unless summoners.key?(performance[:summoner_name])
    end

    summoners.merge!(related_summoners)
    gather_matches
    gather_performances(dive: false)
    puts "GATHERING ADDITIONAL PERFORMANCES FINISHED"
  end
end
```

22 pav. Pagrindinės LeagueImporter klasės implementacija