TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Lauri Keel 773667IDDR

DOKUMENDIKOGU LAIENDUSE DISAIN JA TEOSTUS POSTGRESQL ANDMEBAASILE

Diplomitöö

Juhendaja: Priit Raspel
MSc

Tallinn 2019

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Lauri Keel

20.05.2019

Annotatsioon

Käesoleva töö eesmärk on võimaldada PostgreSQL-i kasutamist täisväärtusliku dokumendikoguna. Lisaks vaadeldakse erinevate süsteemiseadete mõju jõudlusele.

Eesmärgi saavutamiseks esmalt analüüsitakse erinevaid andmeformaate, mis võiksid sobida dokumendikogus kasutamiseks ning realiseeritakse neist ühe tugi koos vastavate operaatorite ja indeksite toega PostgreSQL-i laiendusena.

Järgnevalt analüüsitakse dokumentide sisu kohta statistika kogumise võimalusi ning lisatakse vastav funktsionaalsus loodud laiendusele.

Viimaks analüüsitakse erinevaid jõudlust mõjutavaid tegureid, realiseeritakse jõudlustestide raamistik ning vaadeldakse selle abil erinevate tegurite mõju PostgreSQL-ile ja MongoDB-le.

Töö tulemiks on dokumendikogu, mis oma funktsionaalsuselt, jõudluselt ja laiendatavuselt on märkimisväärselt parem seni populaarseimast dokumendiandmebaasist.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 72 leheküljel, 9 peatükki, 35 joonist, 6 tabelit.

Abstract

Design and Implementation of a Document Store Extension for PostgreSQL

The objective of this thesis is to enable using PostgreSQL as a full fledged document store. Additionally, the effect of different system configurations to performance is evaluated.

The base issue is the limited availability of data types and lack of logical types in JSON, thus limiting the usefulness of the *jsonb* data type of PostgreSQL in a (partially) schemaless scenario.

This thesis resolves this by implementing support for a JSON-like data serialization format that supports adding logical types to all values. For instance, it will be possible to store a timestamp as a Unix timestamp (integer) and tag it accordingly which allows the application to later correctly deserialize it into the timestamp object used by the particular programming language.

Firstly, different data serialization formats that may be suitable for a document store are assessed. Support for one of them along with the relevant operators and index support is implemented as a PostgreSQL extension.

Secondly, options for gathering statistics of complex documents are studied and the relevant functionality is added to the previously implemented extension.

Finally, different aspects that affect performance are analyzed, a performance testing framework is implemented and the effect of different system configurations to the performance of PostgreSQL and MongoDB is evaluated.

The result of this thesis is a document store that from its functional, performance and extensibility aspects is significantly superior to the current most popular document database.

The thesis is in Estonian and contains 72 pages of text, 9 chapters, 35 figures, 6 tables.

Lühendite ja mõistete sõnastik

BSON	MongoDB kasutatav andmete jadastuse formaat
dokumendikogu	semi-struktureeritud (näiteks JSON formaadis) andmete talletamiseks mõeldud andmebaas või selle osa
GitHub	Git repositooriumite majutuskeskkond
GIN indeks	üldistatud pöördindeks
Gnuplot	vabavaraline diagrammide loomise tarkvara
Grafana	aegreaandmete visualiseerimise tarkvara
kaudne jagamine	ressursihalduse meetod, mis võimaldab efektiivselt muudetavatest andmetest koopiaid luua
MongoDB	populaarne dokumendiandmebaas
NVMe	liides, mis võimaldab ligipääsu ketastele kasutades PCI Express siini
ObjectId	MongoDB kasutatav 12-baidine primaarvõti
PipelineDB	vabavaraline aegreaagregatsiooni lisa PostgreSQL-ile
PostgreSQL	vabavaraline relatsiooniline andmebaas
PostgreSQL-XL	mitmeid servereid korraga kasutav PostgreSQL-i lahendus, mis peidab kliendi eest vastava keerukuse
selektiivsus	(andmebaaside kontekstis) iseloomustab päringu võimet ridu filtreerida
ZFS	kaudse jagamise põhimõttel töötav failisüsteem
Zipfi jaotus	(andmebaaside kontekstis) Zipfi seaduse põhine jaotus, mille järgi viimati sisestatud read on populaarsemad

Jooniste loetelu

Joonis 1. Andmete suurus kettal erinevate formaatide lõikes. 58

Joonis 2. Failisüsteemi pakkimise suhe failisüsteemi taseme pakkimisel. 59

Joonis 3. Genereeritud andmetega andmebaaside suurus kettal failisüsteemi pakkimist kasutades (v.a. tärniga märgitud, mis kasutavad vaid andmebaasi taseme oma ja *jsonb*, mis kasutab mõlemat). 60

Joonis 4. GitHub-i andmetega andmebaaside suurus kettal failisüsteemi taseme pakkimist kasutades. 61

Joonis 5. Failisüsteemi tegelikult kasutatud blokisuurused 128KiB blokisuurusel töökoormuse 1 korral. 62

Joonis 6. Päeviku failisüsteemi blokisuuruse mõju jõudlusele, töökoormus 1. . . . 63

Joonis 7. Päeviku failisüsteemi blokisuuruse mõju jõudlusele, töökoormus 14. . . 64

Joonis 8. Andmebaasi failisüsteemi blokisuuruse mõju jõudlusele, töökoormus 1. 64

Joonis 9. Andmebaasi failisüsteemi blokisuuruse mõju jõudlusele, töökoormus 6. 65

Joonis 10. Andmebaasi failisüsteemi blokisuuruse mõju kettaliidese andmemahule, töökoormus 6. 66

Joonis 11. Andmebaasi failisüsteemi blokisuuruse mõju kettaliidese operatsioonide arvule, töökoormus 6. 67

Joonis 12. Andmebaasi mällu lugemise kiiruse seos blokisuurusega, töökoormus 6. 67

Joonis 13. Vähendatud turvalisusgarantii mõju jõudlusele, töökoormus 1. 68

Joonis 14. Vähendatud turvalisusgarantii mõju PostgreSQL-i jõudlusele, töökoormus 10. 69

Joonis 15. Vähendatud turvalisusgarantii mõju MongoDB jõudlusele, töökoormus 10. 70

Joonis 16. Andmebaasi puhvrite suuruse mõju PostgreSQL-i jõudlusele, töökoormus 6. 70

Joonis 17. Andmebaasi puhvrite suuruse mõju PostgreSQL-i jõudlusele, töökoormus 11. 71

Joonis 18. Andmebaasi puhvrite suuruse mõju PostgreSQL-i jõudlusele, töökoormus 14. 71

Joonis 19. Lõimede arvu mõju jõudlusele, töökoormus 6. 72

Joonis 20. Lõimede arvu mõju jõudlusele, töökoormus 1. 73

Joonis 21. Lõimede arvu mõju jõudlusele, töökoormus 14. 73

Joonis 22. *logbias=throughput* andmebaasi köitel kasutamise mõju PostgreSQL-i jõudlusele, töökoormus 6. 74

Joonis 23. *allocation_size=8* mõju MongoDB jõudlusele. 75

Joonis 24. Erinevate MongoDB talletusmootorite jõudlus, töökoormused 6 ja 11. . 76

Joonis 25. MongoDB residentsete talletusmootorite anomaalia, töökoormus 1. . . . 76

Joonis 26. Erinevate andmebaaside jõudlus, töökoormused 1 ja 6. 77

Joonis 27. Erinevate andmebaaside jõudlus, töökoormused 13 ja 14. 77

Joonis 28. Mälusõlmede mõju jõudlusele, töökoormused 1 ja 6. 78

Joonis 29. Mälusõlmede mõju jõudlusele, töökoormused 13 ja 14. 79

Joonis 30. ZIL-i kiiruse mõju jõudlusele, töökoormus 1. Tärniga märgitutel kasutati *zfs_nocacheflush=0*. 80

Joonis 31. Kirjete suuruse mõju PostgreSQL-i jõudlusele, töökoormus 10. 81

Joonis 32. PostgreSQL-i seaded: sünkroonne kirjutamine 91

Joonis 33. PostgreSQL-i seaded: asünkroonne kirjutamine 92

Joonis 34. MongoDB seaded: sünkroonne kirjutamine 93

Joonis 35. MongoDB seaded: asünkroonne kirjutamine 94

Tabelite loetelu

Tabel 1. Sobivad ja mittesobivad andmeformaadid 24

Tabel 2. Realiseeritavad operaatorid. Tärniga märgitutel puudub *jsonb* ekvivalent, need realiseeritakse parema jõudluse huvides. 46

Tabel 3. Erinevate andmebaaside/talletusmootorite latentsus, töökoormus 1. 81

Tabel 4. Erinevate andmebaaside/talletusmootorite latentsus, töökoormus 6. 82

Tabel 5. Erinevate andmebaaside/talletusmootorite latentsus, töökoormus 11. . . . 82

Tabel 6. Erinevate andmebaaside/talletusmootorite latentsus, töökoormus 14. . . . 82

Sisukord

1 Sissejuhatus 14

2 Ülesande püstitus 16

3 Uuring 18

3.1 Sissejuhatus 18

3.2 Seotud töid 20

3.3 Andmeformaadid 20

3.3.1 Kahend- ja tekstipõhised 21

3.3.2 Skeemiga ja skeemita 22

3.3.3 Kiire alamväärtuse juurdepääs 22

3.3.4 Spetsiifilised ja keerukad 23

3.3.5 Puuduvad tüübi metaandmed 23

3.3.6 Kokkuvõte 24

3.4 Andmetüübid PostgreSQL-is 24

3.5 Statistika 25

3.5.1 Efektiivsus 25

3.5.2 Olulisus 25

3.5.3 Talletamine 26

3.6 Jõudlustestid 26

3.6.1 Andmebaaside seadistused 27

3.6.1.1 PostgreSQL 27

3.6.1.2 MongoDB 28

3.6.2 Failisüsteemi seadistused 29

3.6.3 Operatsioonisüsteemi seaded . . . 30
3.6.4 Olemasolevad avatud koodiga jõudlustestid . . . 31
3.6.5 Informatsiooni kogumine . . . 31
3.6.6 Riistvara ja operatsioonisüsteem . . . 32
3.6.7 Andmeallikad . . . 32
4 Analüüs . . . 33
4.1 Sobiva formaadi valik . . . 33
4.1.1 Apache Arrow . . . 33
4.1.2 Amazon Ion . . . 34
4.1.3 CBOR . . . 34
4.1.4 VelocyPack . . . 34
4.1.5 Kokkuvõte . . . 35
4.2 Statistika . . . 35
4.2.1 Kogumine . . . 36
4.2.2 Talletamine . . . 36
4.2.3 Kasutamine . . . 36
4.3 Jõudlustestid . . . 37
4.3.1 Testiraamistiku valik . . . 38
4.3.2 Testiraamistiku valideerimine . . . 39
4.3.3 Testiraamistiku disain . . . 39
4.3.3.1 Töökoormused . . . 39
4.3.3.2 Algandmed . . . 40
4.3.4 Jõudlustestide läbiviimine . . . 41
5 Projekt . . . 43
5.1 Andmeformaat . . . 43
5.1.1 PostgreSQL . . . 44
5.1.2 *CREATE TYPE* . . . 44
5.1.3 *CREATE CAST* . . . 45

5.1.4 *CREATE OPERATOR* . . . 45
5.1.5 *CREATE OPERATOR CLASS* . . . 45
5.1.6 Lisafunktsioonid . . . 46
5.2 Statistika . . . 46
5.2.1 Kogumine ja talletamine . . . 47
5.2.2 Kasutamine . . . 47
5.3 Testiraamistik . . . 48
5.4 Jõudlustestide läbiviimine . . . 48
5.4.1 Süsteem . . . 48
5.4.2 Keskkond . . . 49
5.4.3 Aspektide valik . . . 50
6 Realisatsioon . . . 51
6.1 Andmeformaat . . . 51
6.1.1 Väärtustüübi lisamine . . . 52
6.1.2 Mikrotestid . . . 52
6.2 Statistika . . . 52
6.2.1 Paikapidavuse testid . . . 53
6.3 Testiraamistik . . . 53
6.4 Jõudlustestide läbiviimine . . . 54
6.4.1 Süsteem . . . 54
6.4.2 Keskkond . . . 55
6.4.3 Läbiviimine . . . 55
6.4.4 Testiraamistiku valideerimine . . . 56
6.5 Edasisi töid . . . 56
7 Jõudlustestide tulemuste analüüs . . . 57
7.1 Andmete kompaktsus . . . 57
7.1.1 Andmete suurus kettal . . . 57
7.1.2 Andmebaaside pakkimise efektiivsus . . . 58

7.1.3 Andmebaaside suurus kettal 60
7.2 Blokisuurus 62
7.2.1 Päevikul 63
7.2.2 Andmebaasil 64
7.3 Vähendatud turvalisusgarantii 68
7.3.1 PostgreSQL 68
7.3.2 MongoDB 69
7.4 Andmebaasi puhvrid 70
7.5 Lõimede arv 72
7.6 *logbias=throughput* 74
7.7 MongoDB 75
7.7.1 *allocation_size=8* 75
7.7.2 Residentne 75
7.8 Erinevate lahenduste võrdlus 77
7.9 NUMA 78
7.10 Sünkroonse kirjutamise kiirus 79
7.11 Kirjete suurus 80
7.12 Latentsus 81
8 Kokkuvõte 84
9 Summary 86
Kasutatud kirjandus 88
Lisa 1 – Andmebaaside seadistused 91

1 Sissejuhatus

PostgreSQL on vabavaraline relatsiooniline andmebaasisüsteem, mis omab suurel hulgal erinevaid laiendusi. Paljud neist pole relatsioonilise andmemudeliga üldse seotud, kuid lisavad kogu süsteemile suure lisandväärtuse. Üheks selliseks laienduseks on *hstore*, mis loob uue sõnastiku andmetüübi ja võimaldab võti-väärtus paaride talletamist ehk teisisõnu kasutada PostgreSQL-i dokumendikoguna.

Võrreldes mõnede spetsiaalsete dokumendikogudega, näiteks MongoDB-ga, on sellel aga üks suur puudus: talletatavate andmetüüpide hulk on piiratud. *hstore* esimeses versioonis toetati vaid tekstipõhiseid väärtusi, teises versioonis (tuntud ka kui *jsonb* andmetüüp) kasutatakse andmevahetuseks JSON formaati, mis ei võimalda läbi metaandmete väärtustele lisatähendusi anda, näiteks pole võimalik eristada numbrit ja numbriliselt talletatud ajatemplit. Sarnane piirang eksisteerib ka mitmetes teistes andmebaasides, näiteks MySQL-is ja Oracle-s.

Teine *hstore* ja sarnaste andmetüüpide puudus PostgreSQL-is on statistika puudumine. Andmete omaduste kohast statistikat kasutatakse sobiva päringuplaani valimisel. Näiteks võib päringuplaneerija eeldada, et päringu tulemuste arv on suhteliselt väike ja leida need indeksi põhjal, samas kui tegelikult on tulemusi mitme suurusjärgu võrra rohkem ning oleks märgatavalt kiirem kasutada kirjete täielikku skaneerimist.

Käesoleva töö raames kõrvaldatakse nimetatud puudused ning võrreldakse tulemust olemasolevate lahendustega.

Töö koosneb sissejuhatusest, ülesande püstitusest, uuringust, analüüsist, projektist, realisatsioonist, tulemuste analüüsist, kokkuvõttest ja lisadest.

Ülesande püstituses määratletakse töö väljundid. Lisaks loetletakse erinevad piirangud, millele töö tulem vastama peab.

Uuringus antakse esmalt ülevaade probleemi taustast: mis on dokumendikogud, skeemi valikulisus, millised on olemasolevad lahendused ja miks nad ei sobi. Seejärel uuritakse erinevaid andmeformaate ning analüüsitakse nende sobivust dokumendikogus kasuta-

miseks. Lisaks käsitletakse andmete statistika olulisust ühe Eestis üldtuntud näite põhjal, kus rakenduse jõudlus kadus mõne minutiga puuduliku statistika tõttu. Peatüki viimases osas uuritakse kuidas dokumendikogude lahendusi võrrelda ning kuidas erinevad andmebaaside, operatsioonisüsteemi ja failisüsteemi seadistused jõudlust mõjutavad.

Analüüsis valitakse sobiv andmeformaat PostgreSQL-i dokumendikogu laienduses kasutamiseks. Lisaks käsitletakse kuidas on võimalik taaskasutada olemasolevat statistikasüsteemi, luuakse reeglid dokumentide destruktureerimiseks ning leitakse sobiv viis uue statistikasüsteemi integreerimiseks. Peatüki kolmandas osas analüüsitakse erineviad jõudlust mõjutavaid tegureid, määratletakse nõuded testiraamistikule ja jõudlustestide läbiviimisele, valitakse sobiv testiraamistik ning valideeritakse selle jõudlust.

Projektis projekteeritakse valitud andmeformaadile vajalikud muudatused, samuti PostgreSQL-i laiendus uue andmetüübi, selle operaatorite, indekseerimise ja dokumentide statistika jaoks. Lisaks käsitletakse testiraamistiku ehitust ning kuidas testimiskeskkonda võimalikult automaatselt käidelda ja minimaalsete väliste mõjutuste saavutamiseks üles seada.

Realisatsioonis käsitletakse teostatud PostgreSQL-i laienduse täpsemat ülesehitust ning kuidas selle korrektsust kontrolliti. Lisaks kirjeldatakse detailsemalt testiraamistiku ülesehitust, jõudlustestide läbiviimise süsteemi ja läbiviimist, samuti kuidas valideeriti testiraamistiku tulemuste õigsust.

Tulemuste analüüsis võrreldakse käesoleva töö tulemit ühe populaarse dokumendikogu lahendusega ning vaadeldakse erinevate andmebaasi ja operatsioonisüsteemi parameetrite mõju jõudlusele.

Kokkuvõttes esitatakse käesoleva töö järeldused ning valik ideid edasisteks töödeks.

Lisades on toodud kasutatud andmebaaside seadistused.

2 Ülesande püstitus

PostgreSQL-i kasutamiseks täisväärtusliku dokumendikoguna tuleb realiseerida uue andmetüübi tugi, mis võimaldab andmeid rikkalikumalt iseloomustada kui JSON formaat.

Käesoleva töö raames leitakse sobivad andmeformaadid, võttes seejuures arvesse ka jõudlusparameetreid.

Teise väljundina leitakse sobiv meetod statistika kogumiseks ning luuakse vastav formaat selle talletamiseks andmebaasis.

Kolmanda väljundina realiseeritakse valitud andmeformaat koos statistikaga PostgreSQL-i laiendusena, tehes vajadusel minimaalseid muudatusi ka PostgreSQL-i tuumas.

Neljanda väljundina võrreldakse tulemi jõudlust olemasolevate lahendustega ning vaadeldakse erinevate andmebaasi, failisüsteemi ja operatsioonisüsteemi seadistuste mõju jõudlusele.

Töö tulemina tekib põhifunktsionaalsus PostgreSQL-i kasutamiseks täisväärtusliku dokumendikoguna, mis võimaldab mugavalt koos kasutada relatsioonilist ja skeemivaba andmemudelit. Selle eelis spetsiaalsete dokumendikogude, näiteks MongoDB kasutamise ees on PostgreSQL-i stabiilsus, kiirus ja lai funktsionaalsus, eriti arvestades laienduste pakutavaid võimalusi.

Piiranguteks on:

1. Andmeformaadil:
   (a) dünaamilisus – massiivide ja sõnastike sama võtmega väärtuste tüübid ei tohi olla fikseeritud, näiteks peab saama luua massiive, mille kõik väärtused on erinevast tüübist;
   (b) tüübi metaandmed – kõigile väärtustele peab saama lisada metaandmeid neile lisatähenduse andmiseks, näiteks, et numbrina talletatud väärtus on ajatempel;
   (c) lai andmetüüpide tugi – talletada peab olema võimalik tõeväärtusi, lõpmata suuri arve, ujukomaarve, teksti, kahendväärtusi ja tühimärke ning nendest koosnevaid sõnastikke ja massiive.

2. Teostataval realisatsioonil (olulisuse järjekorras):
   (a) jõudlus – tulemi jõudlus peab olema sarnane PostgreSQL-i *jsonb* andmetüübi jõudlusega;
   (b) realiseerimise kiirus – tuleb eelistada valmislahendusi ning taaskasutada võimalikult palju olemasolevat PostgreSQL-i koodi;
   (c) edasised arendused – lahendus peab olema sobivaks baasiks edasisteks arendusteks;
   (d) väliste süsteemide tugi – võimalusel tuleb eelistada populaarseimaid olemasolevaid lahendusi.
3. Teostatava jõudluse võrdlusel:
   (a) objektiivsus – võrreldavad lahendused peavad olema ressursikasutuselt ja andmete turvalisusgarantiidelt maksimaalselt sarnase seadistusega, erisused tuleb selgelt välja tuua.

3 Uuring

Selles peatükis antakse esmalt ülevaade probleemi taustast: mis on dokumendikogud, skeemi valikulisus, millised on olemasolevad lahendused ja miks nad ei sobi.

Seejärel uuritakse erinevaid andmeformaate ning analüüsitakse nende sobivust dokumendikogus kasutamiseks.

Lisaks käsitletakse andmete statistika olulisust ühe Eestis üldtuntud näite põhjal, kus rakenduse jõudlus kadus mõne minutiga puuduliku statistika tõttu.

Peatüki viimases osas uuritakse, kuidas dokumendikogude lahendusi võrrelda ning kuidas erinevad andmebaasi, operatsioonisüsteemi ja failisüsteemi seadistused jõudlust mõjutavad.

3.1 Sissejuhatus

Andmete haldamise viise on väga palju ning sobiva valik sõltub konkreetsest rakendusest, andmete olemusest ning andmemahtudest.

Struktureeritud andmeid talletatakse traditsiooniliselt relatsioonilistes andmebaasides, kus pannakse tavaliselt suurt rõhku andmete usaldusväärsele töötlemisele, mis on realiseeritud ACID põhimõttega [15, 8]:

- A – atomaarsus: andmebaasitehingu mingi osa nurjumine tähendab kogu tehingu nurjumist;
- C – konsistentsus: iga tehing viib andmebaasi ühest lubatud olekust teise, kõik andmed vastavad igal ajahetkel ettemääratud kitsentustele;
- I – isoleeritus: rööbiti sooritatavate tehingute vastastikune sõltumatus;
- D – püsivus: sooritatud tehingute püsimajäämine ka vigade, süsteemi tõrgete jms korral.

ACID ei paku käideldavusgarantiid: eelistatud on teenuse käideldamatus põhimõtete rik-

kumisele.

Sellele vastandub BASE, mis paneb esikohale käideldavuse [13]:

- BA – sisuliselt käideldav: andmebaas tagab käideldavuse, kuid tagastatavad andmed võivad olla ajutiselt vananenud või ligikaudsed;
- S – pehme olek: andmed ei ole püsivad, kuid neid saab uuesti arvutada või lugeda;
- E – lõpuks konsistentne: vananenud andmeid lubatakse eeldusel, et konsistentsus saavutatakse lühikese aja jooksul kõigis koopiates.

Sellega said populaarseks ka dokumendikogud, mis vastanduvad relatsioonilistele andmebaasidele andmete talletamise põhimõttelt. Kui relatsioonilises andmemudelis on sama objekti andmed erinevates tabelites laiali ning seosed kirjeldatud võõrvõtmetega, siis dokumendikogudes on kõik vastavad andmed samas dokumendis koos. See võib olenevalt kasutusjuhust olla märgatavalt mugavam ja kiirem, eriti suuremate andmehulkade töötlemisel ja/või kui kõiki andmeid on vaja korraga.

Dokumendikogude teiseks eripäraks on võimalus kasutada määratlemata skeemiga (skeemivaba) või osaliselt määratletud skeemiga andmemudelit. Skeemiga andmemudel eeldab, et enne andmeid talletama asumist defineeritakse mudeli struktuur ja kasutatavad andmetüübid. Skeemivaba andmemudeli korral mingeid piiranguid ei ole ning dokumendi struktuuri, mis võib igal dokumendil erinev olla, määrab kasutaja. Lisaks on ka vahepeale variant, kus dokumendi andmebaasi sisestamisel kontrollitakse selle täielikku või osalist skeemile vastavust, näiteks kohustuslike väljade olemasolu.

Andmeskeemi dünaamilisus võib olla nii eeliseks (täielik vabadus talletada mida ja kuidas parasjagu vaja andmebaasis muudatusi tegemata) kui ka puuduseks (tarkvara kood muutub keerulisemaks, kuna tuleb kontrollida kõiki võimalikke variante, suurem võimalus vigadeks). Nii nagu mõneks rakenduseks on parem ACID semantika ja mõneks teiseks BASE, on ka relatsioonilise ja dokumendi ning (osalise) skeemiga ja skeemita andmemudeliga. Nende aspektide käsitlemine ei ole käesoleva töö skoobis.

Skeemivaba andmemudeli puhul on oluline, et transpordiks ja talletamiseks kasutatav andmeformaat oleks ennastkirjeldav ning suudaks väljendada võimalikult paljusid andmetüüpe. See võimaldab tagastatud andmeid õigesti objektida vastavalt kasutatava programmeerimiskeele iseärasustele.

Näiteks on tavaliselt olemas kindel klass ajatemplite haldamiseks, mille väärtus jadastamisel talletatakse kindlas formaadis, harilikult Unix-i ajatemplina ehk sekunditena 1. jaanuarist 1970. Selle väärtuse hilisemal objektimisel on oluline tuvastada, et tegemist oli

ajatempliga. Selleks võib formaadil olla eraldi ajatempli tüüp või võimalus anda väärtustele lisatähendusi siltide lisamise abil. Viimane neist on parem, kuna võimaldab lihtsamini lisada uusi tüüpe ning olemasolevatele (rakendusespetsiifilisi) lisatähendusi anda.

Kuigi PostgreSQL on ennekõike relatsiooniline andmebaas, saab teda kasutada ka dokumendikoguna. Selleks on tal sõnastiku andmetüübid *hstore, json* ja *jsonb*, mis võimaldavad võti-väärtus paaride talletamist. *hstore* võimaldab väärtustena talletada vaid teksti, *json* ja *jsonb* kasutavad andmevahetuseks JSON formaati ning toetavad väärtustena kõiki selles formaadis lubatud andmetüüpe.

Paraku ei ole see piisavalt ennastkirjeldav ning eelnevalt toodud ajatempli näite korral ei ole objektimisel võimalik eristada numbrit ja numbrina talletatud ajatemplit. Ainus võimalus oleks kasutada skeemi, kuid see ei pruugi olla selgelt defineeritud või vastavas süsteemi osas saadaval.

PostgreSQL-i kasutamiseks täisväärtusliku dokumendikoguna on vajalik realiseerida uus andmetüüp, mis võimaldab lisatähenduste andmiseks väärtustele siltide lisamist.

3.2 Seotud töid

Varasemalt on PostgreSQL-ile lisatud BSON andmetüübi tugi, mis andis märgatava jõudluseelise võrreldes tekstipõhise JSON andmetüübiga [18]. *jsonb* kahendformaadi tuge sel hetkel veel realiseeritud ei olnud.

Samuti on võrreldud *hstore* ja võti-väärtus paaride jõudlust [25] ning olem-attribuut-väärtus meetodi asendamist JSON dokumentidega [29]. Mõlemas leiti, et dokumendimudeli jõudlus on parem.

3.3 Andmeformaadid

Andmeformaate ning tarkvara nende jadastamiseks ja objektimiseks on mitmeid. [19] nimetab oma võrdluses formaadid JSON, XML, XStream, Protostuff, Java Serialization, Protocol Buffers, Thrift ja Apache Avro, samuti jadastajana Jakcson-i, mis omakorda toetab lisaks formaate CBOR, CSV, Amazon Ion, Java Properties, Smile, YAML, BEncode, BSON, Efficient XML Interchange, MessagePack, HOCON, Rison ja VelocyPack [16]. Lisaks on Apache Tarkvarafond arendamas formaati Arrow [3].

Seega on kokku 22 potensiaalselt sobivat andmeformaati. Kuna otsitaval formaadil peavad olema väga spetsiifilised omadused on tõenäoline, et valdav osa neist erinevatel põhjustel ei sobi.

Sobiv formaat:

- on kiire ja ressursisäästlik (piirang 2a) – kuna suurem osa andmetega teostatavatest operatsioonidest hõlmab nende lugemist ja nendest otsimist on väga oluline just nende aspektide jõudlus;
- toetab vajalikke andmetüüpe (piirang 1c);
- toetab väärtustele metaandmete lisamist (piirang 1b);
- ei sea dokumendi struktuuri järgi piiranguid väärtuste andmetüüpidele (piirang 1a);
- omab C või C++ realisatsiooni (piirang 2b);
- eelistatavalt omab realisatsioone paljudes programmeerimiskeeltes (piirang 2d).

Järgnevalt vaadeldakse neid tingimusi ja leitud andmeformaatide neile vastavust lähemalt.

3.3.1 Kahend- ja tekstipõhised

Andmeformaate saab üldjoontes jagada kaheks: tekstipõhisteks ja kahendformaatideks. Tekstipõhised formaadid on kodeeritud tekstina, näiteks JSON ja XML. Kahendformaadid on kodeeritud bitijadana ning on üldjuhul kompaktsemad ja töötlemiseks kiiremad, kuna kõvakettalt loetav ja sellele kirjutatav andmemaht on väiksem, samuti võtavad nad mälus vähem ruumi. Erinevalt tekstipõhistest formaatidest, mis on inimloetavad ja mida saab tekstiredaktoriga lihtsasti muuta, on kahendformaadis andmete muutmiseks vaja spetsiaalset tarkvara konkreetse formaadi jaoks. Olenevalt rakendusest võib kompaktsusest saadav kasu selle üle kaaluda.

Näiteks „123456789” kodeerimiseks tekstina kulub 9 baiti, kuid kahendarvuna vaid 4 baiti. Protsessori sõnast suuremate numbrite korral saab kasutada kahend-kümnendkoodi, mis on poole kompaktsem kui tekstipõhine kodeering, kuna ühte baiti kodeeritakse kaks numbrit. Lisaks on võimalik teha erinevaid formaadipõhiseid optimisatsioone, mis suurendavad kompaktsust veelgi.

Käesoleva töö raames teostatavas realisatsioonis on esikohal jõudlus, seega kaalutakse vaid kahendformaatide kasutamist. Kuigi jõudlus oleneb suuresti ka realisatsioonist ja on võimalik, et näiteks mõni JSON formaadi jadastaja või objektija on kiiremad, kui mõne kahendformaadi omad [43], eeldatakse, et realiseerijad on andnud oma parima. Alles jääb 13 potensiaalselt sobivat formaati.

3.3.2 Skeemiga ja skeemita

Dokumendikogus võivad dokumentide väljade arv ja sama võtmega väljade tüübid varieeruda. Kui puuduvaid ja lisanduvaid välju toetavad enamik skeemipõhiseid formaate ning skeeme oleks võimalik ka sisendandmete põhjal automaatselt genereerida, siis varieeruvaid sama välja tüüpe mitte. Sellest tulenevalt ei ole koguülese ühtse skeemi loomine võimalik või kaoks dünaamilisus, mis on käesoleva töö tulemis vajalik. Samas on skeemiga andmemudel on kompaktsem ja jadastuseks ning objektimiseks kiirem [43].

Üheks lahenduseks oleks skeemi talletamine iga dokumendi kohta eraldi. See eeldab sisendandmete töötlemist skeemi loomiseks ning seejärel uuesti andmete talletamiseks, mis on ressursimahukam kui skeemivaba formaadi kasutamine, kus andmeid tuleb töödelda vaid üks kord. Lisaks sellele ei pruugi olla skeemiga formaatide tarkvara optimiseeritud kiireks skeemide loomiseks, kuna tavaolukorras tuleb seda ette võrdlemisi harva. Samuti kaob kompaktsusest saadav eelis ja võib juhtuda, et tulenevalt üldkuludest on kasutatav andmemaht suurem, kui skeemita formaatide puhul.

Nimetatud puudustest tulenevalt käesolevas töös eraldiseisva skeemiga formaate ei käsitleta. Alles jääb 10 potensiaalselt sobivat andmeformaati.

3.3.3 Kiire alamväärtuse juurdepääs

Lugemise jõudluse aspektist on väga oluline kiire juurdepääs alamväärtustele, mis eeldab, et kõigi väärtustega on talletatud nende pikkus baitides. Otsides sõnastikust või massiivist väärtusi on võimalik ebasobivad vahele jätta iga elementi läbi töötamata. Paljudel formaatidel on olemas sõnastike ja massiivide elementide arv, kuid see ei ole piisav (v.a. juhul, kui massiivi väärtused on fikseeritud pikkusega) ning järgmise võtmeni jõudmiseks tuleb läbi käia kõik vastava elemendi alamväärtused, mida võib olla väga palju.

Sellest jõudluskaost tulenevalt jäävad kõrvale formaadid, mille sõnastikel ja massiividel ei talletata nende pikkust baitides. Alles jääb 7 potensiaalselt sobivat andmeformaati.

CBOR formaat on populaarne, standardiseeritud (IETF RFC 7049), lihtsasti laiendatav ning omab realisatsioone väga paljudes programmeerimiskeeltes [6]. Kuigi ta ei vasta käesoleva punkti tingimusele, on võimalik talle vastav tugi luua. See võib tema üldisi häid omadusi arvestades otstarbekaks osutuda ja seega kaalutakse ka selle formaadi kasutamist.

3.3.4 Spetsiifilised ja keerukad

Paljud formaadid on disainitud spetsifiiliste kasutusjuhtude jaoks, mis teeb nende kasutamise antud kontekstis keeruliseks, samuti esineb mitmetel neist muid käesolevas osas käsitletavaid puudusi. Näiteks programmeerimiskeelte sisseehitatud jadastusformaadid on tugevalt mõjutatud vastava keele ülesehitusest ning on ennekõike mõeldud selle andmestruktuuride ühtselt talletamiseks, mitte neist andmete otsimiseks.

Tulenevalt käesoleva töö raames teostatava realisatsiooni piirangutest 2a, 2b ja 2d selliseid formaate ei käsitleta. Alles jääb 3 potensiaalselt sobivat formaati.

3.3.5 Puuduvad tüübi metaandmed

Valitava andmeformaadi kõige olulisem aspekt on võimalus väärtustele metaandmeid lisada. Ilma selleta ei ole võimalik sama andmetüübina jadastatud kuid tegelikult eri tüüpi andmeid eristada, näiteks numbrit ja numbriliselt talletatud ajatemplit. See on põhjus, miks *jsonb* formaat dokumendikogu kasutusjuhuks ei sobi.

Jättes kõrvale ebasobivad formaadid jääb alles vaid 1 potensiaalselt sobiv formaat: Amazon Ion. Lisaks sellele on kaks formaati, mille kasutamist oma eriliste omaduste tõttu kaaluda tasub: Apache Arrow ja VelocyPack.

Amazon Ion on hierarhiline andmete jadastuse formaat, mis on optimiseeritud lugemisoperatsioonideks, toetab paljusid erinevaid andmetüüpe ja väärtustele annotatsioonide lisamist. Olemas on nii tekstipõhine kui ka kahendversioon ning Java, C, Python-i ja JavaScript-i realisatsioonid. [2]

Apache Arrow on tulp-orienteeritud formaat, mis on optimiseeritud efektiivsetele analüütilistele operatsioonidele moodsal riistvaral. Sama tulba andmete koos talletamine võimaldab kasutada protsessorite vektorisatsioonioptimisatsioone, mis ühe instruktsiooniga töötlevad paralleelselt mitut andmepunkti korraga. Formaat võimaldab andmetele kopeerimisvaba ligipääsu, omab realisatsioone paljudes programmeerimiskeeltes ning seda toetavad või on sellele üle minemas paljud Apache projektid, näiteks Cassandra, Hadoop, HBase, Pandas ja Spark [3]. Tulp-orienteeritud formaadid on teatud operatsioonideks kuni mitu suurusjärku kiiremad, kui rida-orienteeritud formaadid [5].

VelocyPack on spetsiaalselt andmebaasisüsteemi silmas pidades arendatud kompaktne, laia andmetüübi toega, laiendatav ja andmete kiirele leidmisele optimiseeritud formaat. Pea kõik andmetega teostatavad operatsioonid on kopeerimisvabad ning sõnastike ja mas-

siivide alamväärtusi saab kasutada eraldiseisvatena igasuguste muudatusteta. Olemas on täielikud C++ ja Java ning poolikud Go, PHP ja NodeJS realisatsioonid. [44]

3.3.6 Kokkuvõte

Sobivad ja mittesobivad andmeformaadid on toodud tabelis 1. Näeme, et vaadeldud 22-st formaadist jäi erinevatel põhjustel kõrvale 21, ainus kõigile nõuetele vastav formaat on Amazon Ion. Lisaks sellele kaalutakse kolme nõuetele mittevastavat, kuid muus osas oluliste eelistega formaati, mille puudused saab potensiaalselt kõrvaldada realiseerimise kiiruse nõuet rikkumata.

Tabel 1: Sobivad ja mittesobivad andmeformaadid

Mittesobivuse põhjus	**Andmeformaat**
Tekstipõhine	JSON, XML, CSV, YAML, Java Properties, XStream, BEncode, HOCON, Rison
Vajab skeemi	Apache Avro, Protocol Buffers, Thrift
Sõnastikel ja massiividel puudub pikkus baitides	CBOR*, MessagePack, Smile
Liiga spetsiifiline ja/või keerukas	BSON, Efficient XML Interchange, Java Serialization, Protostuff
Puuduvad tüübi metaandmed	Apache Arrow*, VelocyPack*
–	Amazon Ion

3.4 Andmetüübid PostgreSQL-is

PostgreSQL toetab vaikimisi paljusid erinevaid andmetüüpe. Lisaks on kasutajatel ja laiendustel võimalik neid lisada, nii olemasolevaid kombineerides kui ka uusi realiseerides, seejuures on paljud vaikimisi saadaolevatest tüüpidest, näiteks *jsonb*, realiseeritud seda sama infrastruktuuri kasutades [30].

Uue andmetüübi realiseerimiseks tuleb luua kaks funktsiooni: üks andmete teisendamiseks kliendi kasutatavast kahendformaadist andmebaasisiseseks kahendformaadiks, mida kasutatakse andmete töötlusel ja talletamisel ning teine vastupidiseks operatsiooniks [30]. Käesoleva töö realisatsioonis on need formaadid samad, seega tagastavad nad oma sisendi muutmata kujul. Seejärel saab *CREATE TYPE* käsuga uue tüübi lisada.

Kui lisaks talletamisele on vajalik andmete töötlemine andmebaasis, tuleb selleks realiseerida vastavad funktsioonid, näiteks väärtuste võrdlemiseks ja samasuse tuvastamiseks.

3.5 Statistika

Päringuplaneerija on andmebaasi komponent, mille ülesandeks on kasutajalt saadud päringu põhjal võimalikult kiire plaani koostamine andmete töötlemiseks. Arvestada tuleb serveri jõudlust (näiteks ketaste parameetreid), andmete hulka, erinevate operatsioonide jõudlust ja ressursivajadust (näiteks kas andmeid sorteerida mälus või kõvakettal) jms.

Neist kõige suurema mõjuga on päringu selektiivsus ehk igale päringu tingimusele vastavate ridade arv [30]. Näiteks indeksi kasutamise efektiivsus oleneb tulemuste arvust ning teatud hetkest on parem teostada terve tabeli skaneerimine. Indeksi kasutamisel loetakse andmeid suvalises järjekorras, kuid tabeli skaneerimise korral järjestikuliselt, mis on suuremate andmemahtude korral märgatavalt kiirem.

Selektiivsuse arvutamiseks kogub PostgreSQL andmete kohta statistikat: võetakse juhuslik valim ridu, võrreldakse vastavate tulpade väärtusi ning (olenevalt tulemustest) salvestatakse iga tulba kohta kõige populaarsemad väärtused, histogramm, puuduvate väärtuste protsent, erinevate väärtuste osakaal jms [30].

Tulemuse umbkaudne ridade arv leitakse selektiivsuse korrutamisel kardinaalsusega ehk tabeli ridade arvuga. Seejuures ei ole oluline täpne tulemus, vaid suurusjärk.

3.5.1 Efektiivsus

Vaikimisi kogutud statistika eeldab, et tulpade väärtused ei ole omavahel korrelatsioonis, vastasel juhul võib see anda mitme tingimuse kasutamisel vigase tulemuse. See on lahendatud võimalusega koguda statistikat selliste väljade kohta koos, kuid vastavad kombinatsioonid tuleb käsitsi defineerida, sest automaatselt kõigi kombinatsioonide kohta statistika kogumine oleks liiga ressursimahukas [30].

Sõnastiku tüüpi väljade kohta statistikat ei koguta, vaid kasutatakse vaikimisi selektiivsust.

Statistika kogumiseks tuleb käivitada *ANALYZE* käsk või lubada *autovacuum* teenus, mis seda iga tabeli jaoks perioodiliselt käivitab.

3.5.2 Olulisus

Ehe näide antud teema olulisusest on 2011. aasta Eesti kohalike valimiste tulemuste kuvamise infosüsteem, mis valimispäeva õhtul pooleteise tunni jooksul kuvas vananenud

tulemusi [33, 32].

Süsteemi haldaja ja arendaja oli veidi parema jõudluse nimel välja lülitanud *autovacuum* teenuse, mis tähendas ka automaatse statistika kogumise puudumist. Teatud andmebaasi täituvuse hetkest ei olnud kasutatud päringuplaan enam optimaalne, mistõttu päringud võtsid märgatavalt rohkem aega.

Seejuures väärib mainimist asjaolu, et statistika genereerimine tabeli kohta võtab üldjuhul alla sekundi, kuid infosüsteem oli kättesaamatu poolteist tundi.

3.5.3 Talletamine

PostgreSQL talletab statistikat *pg_statistic* (tavaline) ja *pg_statistic_ext* (mitme muutujaga) süsteemikataloogides, mis tehniliselt on tavalised tabelid [30]. Kuna nendes käesolevas töös realiseeritava andmetüübi statistikat talletada pole võimalik, tuleb selleks luua sobiva formaadiga kataloog.

3.6 Jõudlustestid

Sobiva andmebaasi valik oleneb ennekõike konkreetsest rakendusest või rakenduse osast, arendusmeetodist, andmete mahust, oodatavast jõudlusest ja turvalisusgarantiidest. Kuna nende parameetrite spekter on väga lai ning paljud aspektid on teineteist välistavad (näiteks ACID ja BASE) ei ole olemas kõigeks sobivat andmebaasi ning isegi sarnaste rakenduste korral võib sobiv andmebaas erineda.

Korretkselt läbi viidud jõudlustestid aitavad õige andmebaasi valikul, kuigi tihti ei ole jõudlus esmatähtis. Võrdlusi PostgreSQL-i ja teiste andmebaasilahenduste vahel on erinevatest aspektidest tehtud mitmeid [11, 20, 40, 4, 42, 1, 39, 38, 10].

MongoDB on kõige populaarsem dokumendikogu [9] ja seega antud töö kontekstis väga oluline. Paraku ei ole Tallinna Tehnikaülikooli 2015. aasta magistritöös MongoDB-ga läbi viidud võrdlus [20], kus leiti, et PostgreSQL on märkimisväärselt aeglasem, korrektne, kuna andmebaase võrreldi vaikimisi seadistustega. Seetõttu kaasatakse MongoDB ka käesoelva töö jõudlustestidesse.

3.6.1 Andmebaaside seadistused

Andmebaaside jõudlust mõjutavad oluliselt kaks aspekti: ressursikasutus ja antav turvalisusgarantii.

Ressursikasutus puudutab ennekõike kasutatava mälu hulka. PostgreSQL kasutab puhvriteks vaikimisi olenevalt versioonist 32-128MB mälu [30], samas MongoDB pool mälust miinus 1GB, minimaalselt 256MB [21]. Näiteks 16GB suuruse mäluga serveri korral oleks MongoDB puhvrite mälukasutus kuni 7GB. Mõlemad kasutavad ka operatsioonisüsteemi puhvreid ja seega ei ole alati selle seadistuse mõju väga suur.

Andmete turvalisusgarantii tähendab seda, et kasutajale päringu vastuse andmise hetkel on teostatud muudatused talletatud püsival andmekandjal ning tõrke või voolukatkestuse korral nad ei hävi. MongoDB teostab vaikimisi parema jõudluse nimel kõrvakettale sünkroniseerimist iga 100ms järel [21], mis tähendab, et umbes 100ms jooksul võivad justkui edukalt sooritatud muudatused hävida. Samas PostgreSQL vaikimisi ei anna vastust enne, kui andmed on turvaliselt kettale talletatud.

Seetõttu ei ole PostgreSQL-i ja MongoDB vaikimisi seadistusega kõrvutamine aus võrdlus ning selle tulemused on tühised.

3.6.1.1 PostgreSQL

PostgreSQL-i puhvrite suuruse määrab parameeter *shared_buffers*, mille soovituslik suurus on 25% saadaolevast mälust [30]. Lisaks on mälukasutusega seotud parameetrid:

- *work_mem* – sorteerimiseks ja paisketabelite jaoks kasutatava mälu hulk, suurema mahuga andmed vastavad operatsioonid sooritatakse kettal;
- *maintenance_work_mem* – erinevate siseste operatsioonide, näiteks VACUUM ja indeksite loomine, kasutatava mälu hulk;
- *wal_buffers* – päeviku puhvrite suurus;
- *effective_cache_size* – umbkaudne erinevate vahemälude kogusuurus, kasutatakse päringute planeerimisel.

Andmete turvalisusgarantiid mõjutavad parameetrid *fsync* ja *synchronous_commit*. Määrates *fsync=off* ei sunnita operatsioonisüsteemi andmeid puhvrist kettale kirjutama ning tõrke või voolukatkestuse korral võib andmebaas jääda vigasesse olekusse ja tekkida täielik andmekadu. Seetõttu on tavaliselt selline seadistus kasulik vaid näiteks andmete esialgsel laadimisel varukoopiast [30].

synchronous_commit=off vastab MongoDB vaikimisi käitumisele, kus kasutajale antakse positiivne vastus enne andmete püsivale andmekandjale kirjutamist ning kolmekordse *wal_writer_delay* (vaikimisi 200ms) perioodi jooksul (kuid maksimaalselt *wal_writer_flush_after* mahus, vaikimisi 1MB) võib tekkida andmekadu [30]. *synchronous_commit* parameetrit saab ka kasutaja transaktsioonipõhiselt määrata [30].

full_page_writes=on kindlustab, et tõrke korral ei jää kettale osaliselt kirjutatud andmeid. Käesolevas töös kasutatava failisüsteemi puhul see ei ole võimalik ja seega võib parameetri välja lülitada, mis mõjub jõudlusele positiivselt [30].

zheap on uus PostgreSQL-i andmete talletamise mootor [12], mida käesoleva töö kirjutamise hetkel integreeritakse PostgreSQL-i tuuma [31]. Kuna tegemist on potensiaalselt suure edasiminekuga on otstarbekas seda võimalusel ka käesoleva töö jõudlustestides käsitleda.

PostgreSQL talletab andmed kettal kasutades 8KiB suurusi lehti. Seda on võimalik muuta vaid PostgreSQL-i kompileerimise ajal. Talletatud andmed pakitakse kasutades kiiret LZ perekonna tihendusalgoritmi [30].

PGTune on veebiteenus ja programm, mille abil on võimalik genereerida serveri parameetrite põhjal sobivad PostgreSQL-i seadistused [28].

3.6.1.2 MongoDB

MongoDB puhvrite suuruse määrab parameeter *storage.wiredTiger.engineConfig.cacheSizeGB,* mille soovituslikku suurust dokumentatsioonis toodud ei ole [21].

Andmete turvalisusgarantiid mõjutab parameeter *storage.journal.commitIntervalMs* (vaikimisi 100ms), mis määrab perioodi, mille jooksul võib tekkida andmekadu [21].

Lisaks sellele saab kasutaja iga päringu korral sundida MongoDB-d kirjutama enne vastuse andmist muudatused püsivale andmekandjale määrates päringu parameetri *j* väärtuseks *true* [21].

Ka MongoDB toetab erinevaid andmete talletusmootoreid: MMAPv1, WiredTiger ja residentne. MMAPv1 tugi varsti kaob ja seega seda käesolevas töös ei käsitleta [21]. Residentne talletusmootor on saadaval vaid MongoDB Enterprise versioonis ning Percona Server for MongoDB-s [27].

WiredTiger-i talletusmootoril on võimalik muuta andmete talletamiseks kasutatavate leh-

tede suurusi järgnevate parameetritega [45]:

- allocation_size – andmebloki suurus, vaikimisi 4KiB;
- internal_page_max – maksimaalne B-puu siseste lehtede suurus, vaikimisi 4KiB;
- leaf_page_max – maksimaalne B-puu lehe suurus, vaikimisi 32KiB.

Talletatud andmed pakitakse vaikimisi kasutades Snappy tihendusalgoritmi [21].

3.6.2 Failisüsteemi seadistused

Lisaks andmebaaside seadistustele on väga olulised ka kasutatava failisüsteemi seadistused.

ZFS on kaudse jagamise põhimõttel toimiv failisüsteem, mida on võimalik detailselt seadistada, võimaldab andmetest koheste tõmmiste ja kirjutatavate kloonide loomist, toetab andmete pakkimist ja krüpteerimist jpm [22].

Failisüsteemi seadistustes on otstarbekas eristada andmete ja päeviku köite seadeid, kuna nende olemus on erinev. Päeviku puhul on oluline kirjed võimalikult kiirelt püsivale andmekandjale talletada ning üldjuhul loetakse neid vaid pärast tõrget või voolukatkestust andmete taastamiseks. Andmete puhul on oluline ennekõike lugemise kiirus ning kirjutamise latentsus jõudlust oluliselt ei mõjuta.

ZFS-i olulisemad parameetrid on [23, 24]:

- *compression* (*on*/*off*) – kas lubada failisüsteemi taseme andmete pakkimine;
- *recordsize* (KiB) – andmebloki suurus; PostgreSQL kasutab 8KiB suurusi andmeblokke, kuid parema pakkimise huvides võib olla otstarbekas kasutada suuremat blokisuurust;
- *atime* (*on*/*off*) – kas pöördumise aega salvestatakse (pole vajalik);
- *primarycache* (*all*, *metadata*, *none*) – kas talletada puhvris täielikke andmeid, metaandmeid või mitte midagi, sobiv väärtus oleneb andmete hulgast;
- *zfs_nocacheflush* (*0*, *1*) – kas sundida ketas puhvreid tühjendama või mitte; pole vajalik, kui kettal on toite katkemise vastane kaitse või puhvrid puuduvad;
- *sync* (*standard, always, disabled*) – kas kasutada sünkroonset kirjutamist kui rakendus seda nõuab, alati või mitte kunagi.

Lisaks tasub eraldi välja tuua parameeter *logbias* (*latency*, *throughput*), mille väärtuseks soovitatakse andmebaasi andmete talletamiseks tihti määrata *throughput* selle olemust täielikult mõistmata.

ZFS kirjutab andmed kettale transaktsioonidena, mille ulatus võib olla olenevalt seadistusest ja andmevoost mõnest sekundist minutini. Sünkroonsete päringute haldamiseks on ZFS Intent Log (ZIL), mis on oma olemuselt päevik – sealt loetakse andmeid vaid süsteemitõrkest taastumisel [22]. Transaktsiooni lõpus kirjutatakse kõik andmed kettale, seejuures talletatakse andmed ning nende juurde kuuluvad metaandmed koos (otsene sünkroniseerimine).

Erandiks on juhud, kus:

- *logbias* väärtuseks on *throughput;*
- ZFS Intent Log ei asu eraldiseisval kettal ning
  - andmete suurus on suurem või võrdne parameetri *zfs_immediate_write_sz* väärtusega;
  - andmete suurus on suurem või võrdne andmebloki suurusega ning andmebloki suurus on suurem, kui parameetri *zvol_immediate_write_sz* väärtus.

Siis kirjutatakse andmed koheselt lõplikult kettale ning ZIL-i talletatakse vastav viide (kaudne sünkroniseerimine) [47]. Transaktsiooni lõpus kirjutatakse kettale ka metaandmed, kuid neid ei talletata andmetega koos ja tekib fragmenteerumine, mis mõjub halvasti lugemise jõudlusele, kuna andmeid tuleb lugeda kahest erinevast kohast.

Mida kiirem on ZIL, seda suurem kirjutamise jõudlus saavutatakse. Kettad, millel andmeid talletatakse ei pea seejuures sarnase jõudlusega olema. ZIL peab talletama vaid ühe transaktsiooni ulatuses andmeid, seega tema reaalne suurus jääb üldjuhul alla 10GB.

3.6.3 Operatsioonisüsteemi seaded

Operatsioonisüsteemil tuleb määrata järgnevad parameetrid [17]:

- *kernel.shmmax* – maksimaalne jagatud mälu segmendi suurus;
- *kernel.shmall* – jagatud mälu lehtede koguarv;
- *vm.overcommit_memory* – kas lubada rakendustele rohkema mälu eraldamine, kui tegelikult saadaval on;
- *vm.overcommit_ratio* – maksimaalne ühele rakendusele lubatud mälu protsent, kui *vm.overcommit_memory=2*.

Lisaks on vajalik enne teste eemaldada operatsioonisüsteemi ja ZFS-i vahemälud. Seda saab teha kirjutades „3” faili */proc/sys/vm/drop_caches* [17]. Tasub tähele panna, et selleks tuleb määrata ka ZFS-i parameeter *zfs_arc_min*, mis vaikimisi on pool parameetri

zfs_arc_max väärtusest.

3.6.4 Olemasolevad avatud koodiga jõudlustestid

Yahoo Cloud Serving Benchmark (YCSB) on Yahoo! loodud testiraamistik NoSQL andmebaaside võrdlemiseks, mida on laialdaselt kajastatud ka akadeemilises kirjanduses [46]. Päringuid sooritatakse kuue erineva andmete lugemise ja uuendamise mustri järgi. Näiteks töökoormus A päringutes on 50% lugemisi ja 50% uuendamisi ning töökoormus B päringutes 95% lugemisi ja 5% uuendamisi.

MongoDB/PostgreSQL JSON Benchmark Tool testiraamistik võrdleb ühteteist erinevat andmete kirjutamise ja lugemise töökoormust [10].

ArangoDB loodud testiraamistik võrdleb lisaks lugemise ja kirjutamise operatsioonidele ka agregaat- ja graafioperatsioone [4]. See toetab ka mitmeid graafiandmebaase, kuid nende vaatlemine ei ole antud töö skoobis.

Enim meediakajastust sai EnterpriseDB 2014. aastal sooritatud võrdlus MongoDB-ga, kus PostgreSQL oli kõigis operatsioonides märkimisväärselt kiirem [11]. Tulenevalt selle testi potensiaalsetest puudustest [20] ning operatsioonide kaetusest teiste testiraamistikega seda testiraamistikku käesolevas töös ei käsitleta.

3.6.5 Informatsiooni kogumine

Nii PostgreSQL kui ka MongoDB koguvad statistikat andmebaasis toimuvast, sh erinevad jõudlusnäitajad [30, 21], mis võimaldab hiljem toimunut analüüsida ja vastavat jõudluskäitumist põhjendada.

Prometheus on vabavaraline monitoorimissüsteem [35], millel on olemas lisad PostgreSQL-i [37], MongoDB [26] ja operatsioonisüsteemi (sh ZFS) [7, 34] mõõtude perioodiliseks kogumiseks. Paraku puudub tal andmete impordi ja ekspordi võimalus ning keerulisemate päringute tugi [35], seetõttu kasutatakse käesolevas töös Timescale aegreaandmebaasi, mis on realiseeritud PostgreSQL-i laiendusena [41] ning toetab andmete importi Prometheus-i lisadest Prometheus-i kasutamata.

Eelnevalt mainitud tarkvara ei kogu infot protsesside kohta eraldi, kuid see on oluline pudelikaelade tuvastamisel. Neid andmeid saab koguda Linux-i *ps* käsu abil [36]. Samuti puudub informatsioon kettakasutuse kohta, mida saab koguda käsuga *zpool iostat* [48].

3.6.6 Riistvara ja operatsioonisüsteem

Testimiseks kasutatava riistvara omadused on järgnevad:

- kaks Intel Xeon Gold 5118 protsessorit (2.30GHz, 12 tuuma, 24 lõime);
- 96 GB mälu;
- kaks kõvaketast HGST HUS724040ALA640 (4TB, 7200 rpm);
- kaks pooljuhtketast WDC CL SN720 SDAQNTW-512G-2000 (512GB, NVMe).

Tegemist on NUMA süsteemiga. Non-Uniform Memory Access (NUMA) on mäludisain, kus mälu poole pöördumise aeg oleneb mälu asukohast protsessori suhtes – lokaalsest mälusõlmest on andmete lugemine märgatavalt kiirem, kui teistest.

Operatsioonisüsteemina on kasutusel Debian Stretch (Linux 4.9.0).

3.6.7 Andmeallikad

GH Archive talletab kõik GitHub-is toimuvad avalikud tegevused ja avaldab need JSON formaadis iga päeva iga tunni kohta [14]. Tegemist on komplekssete kirjetega, mis sobivad hästi dokumendikogude testimiseks.

4 Analüüs

Selles peatükis leitakse analüüsi tulemusena sobiv andmeformaat PostgreSQL-i dokumendikogu laienduses kasutamiseks.

Lisaks käsitletakse kuidas on võimalik taaskasutada olemasolevat statistikasüsteemi, luuakse reeglid dokumentide destruktureerimiseks ning leitakse sobiv viis uue statistikasüsteemi integreerimiseks.

Peatüki kolmandas osas analüüsitakse erineviad jõudlust mõjutavaid tegureid, määratletakse nõuded testiraamistikule ja jõudlustestide läbiviimisele, valitakse sobiv testiraamistik ning valideeritakse selle jõudlust.

4.1 Sobiva formaadi valik

Järgnevalt analüüsitakse lähemalt eelmises peatükis potensiaalselt sobivaks peetud andmeformaate ning valitakse neist üks käesoleva töö realisatsioonis kasutamiseks.

4.1.1 Apache Arrow

Apache Arrow on sisseehitatud skeemiga formaat ning talle laienevad punktis 3.3.2 toodud puudused, seega ta käesoelva töö eelistatud formaadiks ei sobi. Samuti puudub võimalus lisada väärtustele tüübi metaandmeid. Selle tuge ei ole võimalik lihtsalt lisada, samuti kaoks sel juhul potensiaalselt koostalitlusvõime originaalformaati kasutava Apache tarkvaraga, mis on tulp-orienteerituse kõrval selle formaadi teiseks suureks eeliseks.

Samas on tal potensiaali dokumendikogu või tabeli tasandi formaadina, kuna tulenevalt tulp-orienteeritusest on ta teatud operatsioonideks märgatavalt kiirem. Olemas on tugi andmete voona talletamiseks, mis teeb ta sobivaks pidevalt lisanduvate andmete salvestamiseks. Dokumendi tasandi formaadina tuleks tulp-orienteerituse eelis esile vaid suuremate keeruka struktuuriga dokumentide puhul.

Paraku pole hetkel PostgreSQL-is tuge terve tabeli talletamismeetodi muutmiseks, kuid vastav realisatsioon on käesoleva töö kirjutamise hetkel juba peaaegu valmis. Tulenevalt oma headest omadustest ja laiast toest erinevas Apache tarkvaras on tõenäoline, et Apache Arrow saab PostgreSQL-is eelistatud formaadiks tulp-orienteeritud kasutusjuhtudeks.

4.1.2 Amazon Ion

Amazon Ion on ainus formaat, mis rahuldab kõiki tehnilisi nõudmisi, lisaks on ta eeliseks tekstipõhise versiooni olemasolu, mida saab kasutada näiteks andmebaasipäringute kirjutamisel ja mujal, kus inimloetavus on vajalik.

Paraku puudub ta C realisatsioonil igasugune dokumentatsioon ja näited (v.a. testid), mis potensiaalselt tähendab suurt ajakulu selle integreerimisel ning erinevate probleemide lahendamisel. Seetõttu võib osutuda paremaks lahenduseks mõne teise formaadi puuduste likvideerimine.

4.1.3 CBOR

CBOR formaadi eelis on realisatsioonide olemasolu väga paljudes programmeerimiskeeltes, kuid tal puudub sõnastikel ja massiividel pikkus baitides. Laiendusega on võimalik lisada vastavad uued tüübid, mis selle talletavad, kuid on muus osas identsed tavaliste realisatsioonidega.

Formaadil on mitmeid iseärasusi, mille puudumist tuleb enne andmete talletamist kontrollida. Lubatud on määramata pikkusega sõnastikud, massiivid ja stringid, mis ei vasta piirangule 2a. Sõnastike võtmete tüübid ei ole piiratud stringidega, mis muudab keeruliseks päringute koostamise, kuna see vajab inimloetavust ning võib tekitada segadust arvuliste võtmete korral, mille täpne tüüp peab päringu koostajale teada olema.

4.1.4 VelocyPack

VelocyPack on spetsiaalselt andmebaasile disainitud formaat. C++ realisatsioonil on enamiku operatsioonide kohta toodud näited ja dokumentatsioon formaadi integreerimiseks. Lisaks on see objekt-orienteeritud, mis annab koodist ja formaadi ülesehitusest hea ülevaate. Seetõttu on realiseerimiseks kuluv hinnanguline aeg märgatavalt väiksem teistest sobivatest formaatidest.

VelocyPack paraku kõigile tehnilistele nõuetele ei vasta, kuna puudub võimalus lisada väärtustele metaandmeid. Samas on selle võimaluse lisamine tänu realisatsiooni ülesehitusele lihtne ja vähe aeganõudev. Tehniliselt on see sisuliselt identne Amazon Ion-i vastava realisatsiooniga.

4.1.5 Kokkuvõte

Analüüsi tulemusena jäi alles kolm sobivat formaati: Amazon Ion, CBOR ja VelocyPack.

Jõudluse aspektist on nad analoogse ülesehituse tõttu sarnased. Erinevused tulenevad ennekõike realisatsiooni kvaliteedist ning vajalikust valideerimise tasemest. Realisatsiooni kvaliteeti on eksperimentaalsete tulemusteta keeruline hinnata, kuid kuna Amazon Ion ja VelocyPack on jõudlusele orienteeritud ning ka CBOR-il on olemas ressursisäästlik realisatsioon, on alust arvata, et nende jõudlus on sarnane.

Kõigi nende kasutamiseks käesoleva töö realisatsioonis tuleb kulutada aega mõne puuduse likvideerimiseks: Amazon Ion-i puhul selle realisatsiooni testide põhjal kasutama õppimiseks ning silumiseks, CBOR-i ja VelocyPack-i korral laienduste lisamiseks. Autori hinnangul on kõige väiksem ajakulu VelocyPack-i kasutamisel.

Edasised arendused formaadi valikust ei sõltu, kuna kõigil neil on sarnased omadused.

Väliste süsteemide toe aspektist on eelis CBOR-il, kuna ta on standardiseeritud (sh andmetüüpide laiendused) ja omab realisatsioone paljudes programmeerimiskeeltes, enamikes mitu.

Tulenevalt üldistest headest omadustest, heast dokumentatsioonist ja väikseimast hinnangulisest realiseerimise ajakulust otsustati valida käesoleva töö eelistatud formaadiks VelocyPack.

4.2 Statistika

Dokumendi välju on võimalik käsitleda kui relatsioonilise andmemudeli tabelite tulpi, mis võimaldab täielikult taaskasutada PostgreSQL-i olemasolevaid statistika kogumise ja selektiivsuse arvutamise funktsioone.

4.2.1 Kogumine

Statistika kogumiseks tuleb leida vaadeldavates dokumentides esinevate absoluutteede hulk ning arvutada statistika kõigi nende teede väärtuste kohta eraldi. Kuna teid võib olla väga palju on otstarbekas muuta see kasutaja poolt seadistatavaks kasutades valget ja musta nimekirja. Käesolevas töös selliseid võimalusi ei realiseerita.

Järgnevalt defineeritakse erinevate teede käsitlemise reeglid.

Kui tee väärtus puudub, tuleb tagastada tühimärk. Tähenduslikult ei erine see kuidagi relatsioonilise andmemudeli tühimärgilisest väärtusest.

Kui tee väärtus on objekt või massiiv, tuleb tagastada vastava tüübi märk. Selliste väärtuste täielik töötlemine ei ole efektiivne, kuid väärtuse olemasolu indikatsioon võimaldab toetada vastava tee kohta *IS NULL* ja *IS NOT NULL* päringuid.

Kui tees sisaldub massiiv, tuleb väärtust käsitleda kui massiivi ning tagastada kõigi massiivi elementide vastavate väljade väärtused. Teatud juhtudel on otstarbekas koguda lisaks statistika massiivi elementide kohta eraldi, kuid see on ressursimahukam ja seega jäetakse selle lubamine teepõhiselt valikuliseks.

Muudel juhtudel tuleb tagastada vastava tee väärtus. Kuna statistika genereerimise funktsioonid eeldavad kindlatüübilisi väärtusi, viiakse kõik väärtused üle tekstikujule.

4.2.2 Talletamine

pg_statistic süsteemikataloogil puudub võimalus teede talletamiseks, seega tuleb luua uus. Kuna süsteemikataloogide lisamine eeldab muudatusi PostgreSQL-i tuumas, kasutatakse selleks esialgu spetsiaalset tabelit kasutatavas andmebaasis.

Struktuurilt on loodav tabel sama, mis *pg_statistic*, kuid tabeli lõpus on lisaks väli tee salvestamiseks. Selline lahendus võimaldab taaskasutada olemasolevat koodi ja andmestruktuure, mida statistika genereerimise funktsioonid sisendina vajavad.

4.2.3 Kasutamine

Statistikat kasutatakse selektiivsuse arvutamisel. Näiteks selektiivsuse 0.01 (1%) puhul on miljoni kirjega tabelist saadav umbkaudne tulemuste arv 10000.

Selektiivsuse arvutamise defineerimiseks on kaks võimalust:

1. määrata selektiivsuse arvutamise funktsioon operaatori loomisel;
2. kasutada *get_relation_stats_hook* sisendit.

Mõlema puuduseks on asjaolu, et neid saab kasutada vaid predikaatoperaatorite puhul. Kaetud ei ole olukord, kus muude operaatoritega võeti dokumendi osa ja päringus kasutatakse filtreerimisel selle tulemust.

Selle lahendamiseks tuleb luua uus sisend *get_relation_node_stats_hook*, mis päringu filtri täielikku avaldist analüüsides leiab sealt kasutatava tee ja predikaatoperaatori ning tagastab võimalusel vastava selektiivsuse. See on ainus käesoleva töö raames PostgreSQL-i tuumas tehtav muudatus.

4.3 Jõudlustestid

Jõudlust mõjutavad paljud aspektid, mille osakaalu saab täpsemini vaid eksperimentaalselt hinnata. Testides tuleb arvesse võtta:

1. kõvaketta tüüpi (pooljuhtketas või kõvaketas) – pooljuhtkettad on üldjuhul kiiremad, kuid andmebaas peab rahuldavalt toimima ka kõvaketastega;
2. failisüsteemi andmebloki suurust (4–1024KiB) – mida suurem andmeblokk, seda parem andmete pakkimise suhe, samas tähendab see potensiaalselt paljude ebavajalike andmete lugemist ja korduvat kirjutamist;
3. andmebaasi andmebloki suurust (vaikimisi või sama, mis kõvakettal);
4. andmete pakkimist (failisüsteemis või andmebaasis) – efektiivsus oleneb ennekõike andmebaasi andmete pakkimise realisatsioonist ning failisüsteemi andmebloki suurusest;
5. puhvrite suurust – lisaks andmebaasile puhverdab operatsioonisüsteem kettalt loetud andmeid ja tekib andmete topeltpuhverdamine, mis võib olla ebaefektiivne;
6. failisüsteemi puhverdamise tüüpi (täielik või ainult metaandmed) – kui kõik andmed mahuvad andmebaasi puhvrisse, ei ole mõtet operatsioonisüsteemi tasandil neid uuesti puhverdada, piisab nende metaandmetest;
7. failisüsteemi puhvrite olekut – kui andmed on eelnevalt (osaliselt) puhvrisse loetud, ei ole neid vaja uuesti kettalt lugeda ning see annab jõudluseelise;
8. saadaoleva mälu suurust – lisaks puhvritele kasutab andmebaas mälu ka päringute teostamisel ja taustaprotsessides;
9. operatsioonisüsteemi mäluhalduse efektiivsust – mida väiksemate lehtedena mälu

eraldatakse, seda rohkem ressurssi kulub nende haldamiseks;
10. andmete turvalisusgarantii (täielik või osaline) – mida suurem on andmete potensiaalse hävimise aken, seda parem on jõudlus;
11. andmete mahtu – mida rohkem andmeid, seda paremini tulevad esile jõudlust mõjutavad asjaolud;
12. kirjete mahtu – mida suuremad kirjed, seda efektiivsem on pakkimine, kuid ka seda rohkem on vaja andmete töötlemiseks mälu;
13. parallelset klientide arvu – mida rohkem kliente, seda rohkem nad teineteist segavad ressursidele konkureerides (ketas, protsessor);
14. talletusmootori tüüpi (PostgreSQL: tavaline või *zheap*; MongoDB: WiredTiger või residentne) – tulenevalt ehituse erinevustest võivad kummalgi olla töökoormusest tulenevad eelised;
15. transaktsioonide osakaalu ja kestust – transaktsioonid võivad jõudlusele mõjuda nii positiivselt kui ka negatiivselt;
16. ZFS-i otsest ja kaudset sünkroniseerimist – andmete fragmenteerumine mõjub jõudlusele negatiivselt;
17. ketta puhvreid tühjendama sundimist – puhvrite mittetühjendamine mõjub jõudlusele positiivselt.

See nimekiri ei ole lõplik ja kindlasti on võimalik leida veel jõudlust mõjutavaid aspekte, näiteks erinevaid vähemtuntud ZFS-i parameetreid.

4.3.1 Testiraamistiku valik

YCSB testiraamistikul puudub PostgreSQL-i tugi ning selle lisamine ja testimine on potensiaalselt ajamahukas.

ArangoDB loodud testiraamistik ei toeta VelocyPack formaati, kuna puudub teek JavaScript-i jaoks ning NodeJS realisatsioon on vananenud, samuti ei väljasta see täielikke algandmeid, mis on oluline nende võrdlemiseks muude kogutud andmetega.

MongoDB/PostgreSQL JSON Benchmark Tool (*mpjbt*) toetab põhilisi YCSB koormusmustreid, kuid puudub transaktsioonide tugi. Selle lisamine on lihtsa ülesehituse tõttu triviaalne.

4.3.2 Testiraamistiku valideerimine

Enne edasist arendust ja testimisega alustamist otsustati kontrollida testiraamistiku tulemuste õigsust. Selleks loodi lihtne programm, mis tegi tsüklis ükshaaval andmebaasi andmete sisestamise päringuid ning selle tulemust võrreldi *mpjbt* väljundiga.

Testprogramm suutis PostgreSQL-i sekundis sisestada keskmiselt 2000 päringut enam kui *mpjbt*, seejuures ei olnud testi ajal *mpjbt* ega PostgreSQL-i protsessorikasutus 100% lähedal, mis viitab sisemisele probleemile *mpjbt* kasutatavas PostgreSQL-i teegis.

Seetõttu otsustati ka *mpjbt* testiraamistik kõrvale jätta ning luua uus.

4.3.3 Testiraamistiku disain

Testiraamistik peab:

- olema kiire ja võimalikult väikese ressursinõudlikkusega;
- võimaldama operatsioonide kestuse mõõtmist mikrosekundilise täpsusega;
- väljastama kõigi operatsioonide täpse aja ja kestuse;
- mõjutama võimalikult vähe operatsioonide kestust;
- võimaldama erinevate sisendandmete kasutamist;
- kasutama andmebaasi või selle laienduse omaformaate vältimaks andmete teisendamist operatsioonide ajal;
- mälutihendust vajavate keskkondade puhul olema võimalusel mälutihendusvaba;
- olema võimalikult lihtne ja mugav erinevate töökoormuste realiseerimiseks.

Testiraamistik otsustati realiseerida Java programmeerimiskeeles, kuna see rahuldab piiranguid 2a ja 2b. Töökoormuste realisatsioonid on mugavuse aspektist hea luua kasutades Groovy programmeerimiskeelt, kuid see osutus testimisel 5 korda aeglasemaks (konkreetse realisatsiooni puhul, see järeldus ei laiene Groovy-le üldisemalt).

4.3.3.1 Töökoormused

Realiseeritavad töökoormused:

1. andmete sisestamine;
2. andmete sisestamine ning sisestatud rea kohene lugemine;
3. andmete sisestamine ning sisestatud rea kohene uuendamine;

4. andmete sisestamine ning rea lugemine Zipfi jaotuse põhjal (YCSB töökoormus D);
5. andmete sisestamine ning rea uuendamine Zipfi jaotuse põhjal;
6. andmete lugemine juhusliku ID järgi (YCSB töökoormus C);
7. andmete lugemine juhusliku ID järgi ning vastava rea kohene uuendamine (YCSB töökoormus E);
8. andmete lugemine Zipfi jaotuse põhjal ning vastava rea uuendamine;
9. andmete lugemine lühikese vahemiku põhjal (YCSB töökoormus E);
10. andmete uuendamine juhusliku ID järgi;
11. 50% päringutest andmete sisestamine, 50% päringutest andmete lugemine (YCSB töökoormus A);
12. 5% päringutest andmete sisestamine, 95% päringutest andmete lugemine (YCSB töökoormus B);
13. agregaatoperatsioon: välja erinevate väärtuste ning neile vastavate kirjete arvu leidmine;
14. transaktsioon: juhusliku dokumendi lugemine esimesest tabelist, rea sisestamine teise tabelisse, esimesest tabelist loetud dokumendi olemasoleva välja uuendamine, rea sisestamine teise tabelisse ja esimesest tabelist loetud dokumendile uue välja lisamine;
15. dokumendi alam-alamobjekti uuendamine juhusliku ID järgi;
16. andmete sisestamine, sisestatud rea lugemine ID järgi, uuendamine, lugemine ID järgi ja vahemiku põhjal ning tulemuste õigsuse valideerimine;
17. andmete järjestikune lugemine ID järgi.

4.3.3.2 Algandmed

Testides kasutatakse kahte erinevat tüüpi algandmeid:

1. juhuslikult genereeritud dokumente;
2. GitHub-i arhiivi.

Genereeritavad dokumendid sisaldavad erinevat tüüpi välju:

1. ID;
2. 5-15 tähemärgine tekst;
3. 0-99 vahemikus täisarv;
4. tõeväärtus;
5. ajatempel;
6. objektide massiiv;

7. täisarvude massiiv;
8. objekt.

Selline formaat võimaldab vajadusel sooritada kompleksseid päringuid dokumendi kohta, mis iseloomustavad hästi dokumendiformaadi jõudlust.

4.3.4 Jõudlustestide läbiviimine

Testide läbiviimise lahtuamatu osa on süsteem nende juhtimiseks ning jõudlusandmete kogumiseks. See võimaldab mugavalt läbi viia palju erinevate seadistuste kombinatsioonidega teste ning hiljem kogutud andmete põhjal tulemusi analüüsida.

Süsteem peab:

1. üles seadma vastavate seadetega failisüsteemi;
2. üles seadma vastavate seadetega konteineri, milles andmebaasisüsteemi käitada;
3. üles seadma andmebaasisüsteemi vastavate seadete ja andmetega;
4. enne iga jõudlustesti tühjendama operatsioonisüsteemi ja failisüsteemi vahemälu vältimaks mõjutusi eelnevatest testidest;
5. käivitama jõudlusandmete kogumise;
6. läbi viima vastavad jõudlustestid;
7. peatama jõudlusandmete kogumise;
8. eemaldama kasutatud konteineri ja failisüsteemi;
9. jätkama algusest järgmiste seadistustega.

Jõudlusandmeid peab koguma vähemalt iga viie sekundi tagant:

1. käimasolevate protsesside kohta (protsessori ja mälu kasutus, staatus, magamise aadress);
2. failisüsteemi kohta (Prometheus-i vastava lisa väljastatavad andmed, samuti *zpool iostat* käsu tulem talletamaks kettale kirjutatavate ning sealt loetavate andmete blokisuurused, kiiruse ja operatsioonide arvu);
3. andmebaasi kohta (Prometheus-i vastavat lisa kasutades).

Tasub tähele panna, et protsesside mälukasutuse hulka arvestatakse ka jagatud mälu. See tähendab, et sama jagatud mälu blokki kasutatavate protsesside summaarne mälukasutus võib näida märkimisväärselt suurem tegelikust. Seetõttu tuleb lisaks talletada ka konteineri mälukasutus, mis näitab tegelikku summaarset kasutatava mälu hulka.

Enne ja pärast testide sooritamist tuleb lisaks koguda:

1. failisüsteemi seaded ning kasutus;
2. andmebaasi seaded;
3. süsteemi sisendid.

Sama serveri kasutamisel nii andmebaasi- kui ka testiraamistiku käitamiseks peab kasutama vähemalt kahe protsessoriga serverit, millest ühel käitatakse testiraamistikku ja operatsioonisüsteemi ning teis(t)el vaid testitavat andmebaasi ning sellega seotud tarkvara. Latentsuse testimisel peavad kõik testiraamistiku teste läbi viivad lõimed olema eraldatud oma protsessorilõimele.

Sarnaselt ei tohi kasutada operatsioonsüsteem ja testiraamistik ning sellega seotud tarkvara sama ketast, kus asub andmebaas. Nõnda hoitakse ära kõrvalmõjud testitavale andmebaasisüsteemile.

5 Projekt

Selles peatükis projekteeritakse VelocyPack formaadile vajalikud muudatused, samuti PostgreSQL-i laiendus uue andmetüübi, selle operaatorite, indekseerimise ja dokumentide statistika jaoks.

Lisaks käsitletakse testiraamistiku ehitust ning kuidas testimiskeskkonda võimalikult automaatselt käidelda ja minimaalsete väliste mõjutuste saavutamiseks üles seada.

5.1 Andmeformaat

Kõigi piirangute täitmiseks tuleb VelocyPack formaadile lisada täiendus väärtustele siltide lisamise võimaldamiseks. See tähendab, et kõigile väärtustele saab lisada ka loogilise tüübi informatsiooni: näiteks, et täisarvuna salvestatud number on ajatempel või et kahendarv on MongoDB ObjectId tüüpi. Siltide tähendused on seejuures kokkuleppelised ning dokumendis talletatakse vaid täisarvuline ID.

Kõik VelocyPack-i väärtused on iseseisvad. Nõnda võib võtta näiteks massiivi või objekti elemendi ning seda eraldiseisvana kasutada midagi muutmata või lisamata. Erinevaid väärtuste tüüpe eristatakse nende esimese baidi järgi, näiteks 0x18 on tühimärk, 0x02 massiiv ja 0x1C Unix-i ajatempel ning nende edasine töötlemine oleneb konkreetsest tüübist.

Siltide lisamise võimaldamiseks tuleb luua uus väärtustüüp, mis sisaldab päisena tüübi täisarvulist ID-d, millele järgnevad andmed tavapärases VelocyPack formaadis. Suurema kompaktsuse saavutamiseks defineeritakse kaks väärtustüüpi: 0xEE, kus tüübi ID on 1-baidine täisarv ja 0xEF, kus tüübi ID on 8-baidine täisarv.

Modifitseeritud VelocyPack formaati nimetatakse edaspidi VPack formaadiks.

5.1.1 PostgreSQL

PostgreSQL-i laiendused koosnevad neljast põhilisest osast:

1. *.control* failist, kus on kirjeldatud teatud parameetrid, näiteks mooduli asukoht, vaikimisi versioon ja kommentaar;
2. *.sql* failist, kus defineeritakse kasutatavad funktsioonid, andmetüübid, operaatorid, operaatorite klassid jms;
3. vajadusel funktsioonide C realisatsioonist;
4. testidest, mis võrdlevad andmebaasipäringute tulemusi etteantud tulemustega.

Käesolevas töös realiseeritakse kõik vajaminevad funktsioonid C programmeerimiskeeles. Enamik neist on funktsionaalsuselt samad vastavate *jsonb* tüübi omadega, millest paljusid on võimalik VelocyPack-i formaadile kohandada.

5.1.2 *CREATE TYPE*

Esmalt tuleb luua uus tüüp *CREATE TYPE* käsuga, tüübi nimeks määratakse *vpack*. Selle eelduseks on 5 funktsiooni:

- *input* ja *output* – tekstipõhise esituse teisendamine tüübi sisemisse formaati ja vastupidi;
- *send* ja *receive* – kahendesituse teisendamine tüübi sisemisse formaati ja vastupidi;
- *analyze* – statistika kogumiseks (vt punkt 5.2).

Käesoleva töö realisatsioonis on tüübi sisemine formaat ja kliendi kasutatav kahendformaat samad. See annab jõudluseelise, kuna puudub vajadus andmete erinevate esituste vahel teisendamiseks, vajalik on vaid sisendandmete väljade pikkuste valideerimine vältimaks puhvri ületäitumise rünnakuid.

Tekstipõhise esituse jaoks kasutatakse prototüübis JSON formaati, mis ei võimalda kõiki andmetüübi aspekte väljendada, kuid lihtsustab silumist ning võimaldab tüübi inimloetava esituse.

Seejärel on võimalik *vpack* tüüpi andmeid andmebaasis talletada.

5.1.3 *CREATE CAST*

Järgnevalt on otstarbekas defineerida teisendused erinevate andmetüüpide vahel kasutades *CREATE CAST* käsku:

- *vpack–bytea* ja *bytea–vpack* – teisendamine *bytea* (kahendandmed) ja *vpack* vahel – võimalike kliendi andmebaasiteegi probleemide puhuks, sisult on mõlemad ekvivalentsed;
- *vpack–text* ja *text–vpack* – teisendamine JSON formaadis teksti ja *vpack* vahel – testimiseks ja silumiseks;
- *jsonb–vpack – jsonb* andmete teisendamiseks *vpack* formaati.

Seejärel on võimalik *vpack* tüüpi mugavalt testida ja siluda, samuti teisendada JSON ja *jsonb* formaadis andmeid *vpack* formaati.

5.1.4 *CREATE OPERATOR*

Senine võimaldab andmeid talletada ning neid erinevate formaatide vahel teisendada. Andmetega erinevate operatsioonide sooritamiseks tuleb defineerida operaatorid. Käesolevas töös realiseeritavad on toodud tabelis 2. Oma funktsionaalsuselt ja käitumiselt on nad täpselt samad, mis *jsonb* omad.

Seejärel on võimalik *vpack* tüüpi andmete kohta sooritada erinevaid päringuid.

5.1.5 *CREATE OPERATOR CLASS*

PostgreSQL-i indeksite süsteemi kasutamiseks tuleb *CREATE OPERATOR CLASS* käsuga operaatoritest luua operaatorite klassid:

1. räsiindeksite jaoks ning realiseerida *vpack_hash* funktsioon, mis tagastab *vpack* väärtuse räsi 32-bitise täisarvuna;
2. B-puu indeksite jaoks ning realiseerida *vpack_cmp* funktsioon, mis võrdleb kahte *vpack* väärtust;
3. GIN indeksite jaoks ning realiseerida 7 vastavat funktsiooni, mis oma funktsionaalsuselt on samad *jsonb* analoogidega.

Seejärel on võimalik *vpack* tüüpi andmeid indekseerida, mis võimaldab paljusid päringuid sooritada märkimisväärselt kiiremini.

Tabel 2: Realiseeritavad operaatorid. Tärniga märgitutel puudub *jsonb* ekvivalent, need realiseeritakse parema jõudluse huvides.

Operaator	**Kirjeldus**
=	ekvivalentsus
<>	mitteekvivalentsus
<	väiksem
<=	väiksem või võrdne
>=	suurem või võrdne
>	suurem
||	kahe väärtuse ühendamine
- >	välja võtmine väärtusest
- >>	välja võtmine väärtusest tekstina
@>	tee järgi eksistentsi või ekvivalentsuse kontroll
<@	eelmise pööratud argumentidega variant
?	võtme olemasolu kontroll väärtuses
?&	võtmete olemasolu kontroll väärtuses konjunktsiooniga
?|	võtmete olemasolu kontroll väärtuses disjunktsiooniga
#>	tee järgi välja võtmine väärtusest
#>>	tee järgi välja võtmine väärtusest tekstina
- >>>	välja võtmine väärtusest täisarvuna*
#>>>	tee järgi välja võtmine väärtusest täisarvuna*

5.1.6 Lisafunktsioonid

Lisaks on otstarbekas realiseerida funktsioonid mõnede ebamugavate operatsioonide lihtsustamiseks:

- *vpack_set* – objekti alamväärtuse tee põhjal määramiseks, *jsonb_set* analoog;
- *to_vpack* – väärtuste (sh andmebaasiridade) teisendamiseks *vpack* formaati, *to_jsonb* analoog.

Turvalisuskaalutlustel tuleb valideerida kasutaja sisestatud dokumentide väljade pikkusi. Vastasel juhul oleks võimalik lihtsasti sooritada puhvri ületäitumise ründeid.

5.2 Statistika

Statistika realisatsioon koosneb kahest põhilisest osast: statistika kogumisest ja talletamisest ning selle päringutes kasutamise võimaldamisest.

5.2.1 Kogumine ja talletamine

Statistika kogumiseks tuleb luua võimalus selle talletamiseks, kuna *pg_statistic* süsteemikataloogil puudub tee attribuut. Selleks luuakse uus tabel *vpack_statistic*, mis on struktuurilt sama, mis *pg_statistic*, kuid mille lõpus on lisaks väli tee salvestamiseks.

ANALYZE käsuga käivitatakse teistehulgas *vpack_analyze* funktsioon, millele antakse valim tabelis leiduvatest kirjetest nende põhjal statistika kogumiseks. Valimi suurus on seadistatav, kuid vaikimisi on see kuni 30000.

Esmalt tuleb leida kirjete absoluutteede hulk, misjärel saab leida statistika iga tee kohta eraldi vastavalt punktis 4.2.1 toodud reeglitele. Selleks tuleb iga tee jaoks luua kohandatud *VacAttrStats* objekt, mis on PostgreSQL-i statistika arvutamise funktsioonide sisendiks ja sisaldab statistika genereerimiseks vajalikku metainformatsiooni.

Seejärel saab käivitada vastava statistika genereerimise funktsiooni selleks spetsiaalselt loodud ajutises mälukontekstis ning talletada tulemused *vpack_statistic* tabelis.

Statistika kogumise ja talletamise lihtsustamiseks on vaja PostgreSQL-i tuumas teha kolm täiendust:

1. lisada *pg_statistic* süsteemikataloogile tee väli;
2. lisada *VacAttrStats* objektile lisaandmete edastamiseks valikulise viite väli;
3. luua uus süsteemikataloog statistika kogumise seadete talletamiseks, näiteks ignoreeritavate teede ja erineva lisastatistika jaoks.

5.2.2 Kasutamine

Keerulisemate avaldiste selektiivsuse leidmiseks tuleb realiseerida *get_relation_node_stats_hook* sisend nagu kirjeldatud punktis 4.2.3.

Predikaatoperaatorite @>, <@, ?, ?& ja ?| selektiivsuse arvutamiseks realiseeritakse funktsioon *vpackcontsel*, mis tee põhjal olenevalt sellest, kas tees sisalduvad massiivid või ei, arvutab olemasoleva *calc_arraycontsel* või *var_eq_const* funktsiooni abil selektiivsuse.

Lisaks realiseeritakse selektiivsuse arvutamise funktsioonid operaatorite = ja <> jaoks. Sisult on nad ekvivalentsed *jsonb* analoogidega.

5.3 Testiraamistik

Testiraamistik realiseeritakse lähtuvalt punkti 4.3.3 analüüsist. Raamistikul on kolm põhilist komponenti:

1. töökoormuse täitja, mis käivitab vastavad operatsioonid ning mõõdab nende kestust;
2. andmebaasiabstraktsioon, mis võimaldab töökoormuste täitjal läbi ühtse liidese andmebaasidega suhelda arvestades seejuures maksimaalset lubatud ühenduste arvu;
3. dokumendivarustaja, mis vajadusel varustab töökoormuse täitjat dokumentidega.

Töökoormuse täitja operatsioonid realiseeritakse sulunditena, kuna see võimaldab kõige paremini vaid vastavate operatsioonide kestust mõõta. Mõõdetud andmed lisatakse Java *ConcurrentLinkedQueue* loendisse, mis võimaldab andmeid lukuvabalt selle algusest lugeda mõjutamata andmete lisamist selle lõppu. Nõnda on võimalik mõõtmistulemusi jooksvalt andmebaasi talletada. Töökoormuse täitjaid võib töötada mitu tükki paralleelselt.

Dokumendivarustaja realiseeritakse failist loetavate (JSON või omaformaadis) ja genereeritavate (omaformaadis) andmete jaoks. Selle oluline osa on puhver, mis võimaldab vältida andmete lugemisest ja genereerimisest tulenevaid mõjutusi operatsioonidele. Puhver realiseeritakse samuti kasutades Java *ConcurrentLinkedQueue* loendit.

Mugavamaks kasutamiseks luuakse käsurealiides, mille abil saab määrata erinevaid parameetreid, näiteks paralleelsuse astet, sooritatavate päringute arvu ja ajalõppu.

5.4 Jõudlustestide läbiviimine

Jõudlustestide läbiviimisel on lisaks testisüsteemile väga oluline ka läbiviimise keskkonna õige seadistamine vältimaks väliseid mõjutusi.

5.4.1 Süsteem

Jõudlustestide läbiviimise süsteem tuleb realiseerida nagu kirjeldatud punktis 4.3.4. Testide juhtimise programm realiseeritakse kasutades Groovy programmeerimiskeelt, kuna antud komponent pole jõudluskriitiline, vaid peab võimaldama võimalikult kiirelt ja mugavalt muudatuste tegemist.

Testides kasutatavaid andmebaase ja testandmeid hallatakse ZFS-i tõmmiste abil. Esmalt tuleb üles seada konteinerid igale andmebaasile ning neisse vastav andmebaas installeerida ja seadistada.

Järgnevalt tuleb teha tõmmised iga andmebaasi andmefailidest ja päevikust ning täita need vastava hulga testandmetega. Enne testi käivitamist tehakse neist omakorda tõmmised testis kasutamiseks. Nõnda on võimalik tagada testide sõltumatus ning välditakse liigset andmete kopeerimist. Pärast testi loodud tõmmised eemaldatakse.

Testide andmed talletatakse spetsiaalses PostgreSQL-i andmebaasis, mis aegreaandmete jaoks kasutab TimescaleDB laiendust. Andmete visualiseerimiseks kasutatakse Grafana-t, milles luuakse uus koondpaneel, kuhu kuvatakse testide parameetrid ja tulemused ning graafikud päringute latentsusest, päringute arvust sekundis, mälukasutusest, failisüsteemi parameetritest jt kogutud andmetest.

5.4.2 Keskkond

Operatsioonisüsteem installeeritakse ühele NVMe ketastest, teine jääb testitava andmebaasi kasutusse. Enne installeerimist tuleb *nvme-cli* abil muuta nende blokisuurus 512 baidilt 4 KiB-le, kuna see on väikseim testitav blokisuurus, ühtib ketta sisese blokisuurusega ja vähendab adresseerimise lisakulu.

ZFS Intent Log-i (ZIL) talletamiseks on otstarbekas kasutada eraldi ketast, kuna see võimaldab märgatavalt paremat läbilaskevõimet. Kuna käesolevas töös kasutatakse teist pooljuhtketast operatsioonisüsteemi ja tulemuste salvestamiseks, on selle kasutamisel ZIL-i jaoks jõudlustestidele negatiivne kõrvalmõju tulenevalt läbilaskevõime pidevast varieeruvusest. Seetõttu tuleb selle asemel kasutada mälus olevat virtuaalset ketast, mis kaotab sünkroonsete kirjete turvalisusgarantii, kuid antud juhul võimaldab jõudlustestid kõrvalmõjudeta läbi viia.

Serveri üks protsessoritest eraldatakse testitavale andmebaasile, millele jääb seega 12 tuuma ja 24 lõime. Selleks kasutatakse Linux-i *cgroups* alamsüsteemi, millega luuakse vastavalt kaks *cpuset*-i ehk protsessoriruumi, millele eraldatakse kasutamiseks kindlad protsessorilõimed ning mälusõlmed: üks andmebaasile ja üks ülejäänud süsteemile.

Lisaks tuleb testiraamistiku testi läbi viivad lõimed eraldada ülejäänud süsteemist. Selleks luuakse kaheksa *cpuset*-i, millest igaühele eraldatakse üks järjestikune protsessori lõim. Järgnevalt luuakse operatsioonisüsteemi tuuma lõimedele oma *cpuset*, milles on kõik protsessori lõimed peale testiraamistiku kasutatavate. See tagab võimalikult väikese interfe-

rentsi ja latentsuse.

Lisaks tuleb *cpuset*-ide loomisel arvestada ka mälusõlmi ning vajadusel need lokaalsega piirata.

5.4.3 Aspektide valik

Kuna punktis 4.3 toodud erinevate aspektide kombinatsioone on lõpmata palju on otstarbekas valida teatud aspekt baastestiks ning realiseerida järgnevad testid vastavalt selle ja järgmiste testide vaheanalüüsidele.

Baastestiks sobib hästi failisüsteemi blokisuurus, kuna see mõjutab jõudlust märkimisväärselt läbi pakkimise efektiivsuse, metaandmete ülekulu ja kirjutamise võimenduse.

Tulenevalt piiratud serveri kasutamise ajast viiakse enamik teste läbi 60 sekundi pikkuses kolmes suuruses andmetega: 10^6 kirjet (mahub täielikult mällu), 10^7 (mahub enamuses mällu) ja 10^8 (ei mahu mällu). Pikemate testide korral peaks andmete hulk ka vastavalt suurem olema, kuna sel juhul jõutakse rohkem andmeid mällu lugeda ning need ei vastaks enam oma kasutusjuhule.

6 Realisatsioon

Selles peatükis käsitletakse teostatud PostgreSQL-i laienduse täpsemat ülesehitust ning kuidas selle korrektsust kontrolliti.

Lisaks kirjeldatakse detailsemalt testiraamistiku ülesehitust, jõudlustestide läbiviimise süsteemi ja läbiviimist, samuti kuidas valideeriti testiraamistiku tulemuste õigsust.

6.1 Andmeformaat

Andmeformaat realiseeriti vastavalt projektile kasutades peamiselt C ning üksikutes funktsioonides ka C++ programmeerimiskeelt.

Realisatsioon koosneb järgnevatest osadest:

- VelocyPack-i teek;
- *vpack_gin.cpp* (753 rida) – GIN indeksite funktsioonid, kohandatud vastavatest *jsonb* omadest;
- *vpack_ops.cpp* (1177 rida) – punktis 5.1.4 kirjeldatud operaatorite realisatsioonid;
- *vpack.h* ja *vpack.cpp* (1019 rida) – toetavad funktsioonid, utiliidid, *to_vpack* realisatsioon, laienduse valmendus;
- *CMakeLists.txt* (63 rida) – CMake koostesüsteemi seadistusfail;
- *FindPostgres.cmake* (39 rida) – CMake koostesüsteemi laiendus, mis võimaldab leida kompileerimiseks vajalikud PostgreSQL-i teegid;
- *vpack.control* (5 rida) – *.control* fail nagu kirjeldatud punktis 5.1.1;
- *vpack–0.1.0.sql* (450 rida) – *.sql* fail, kus on defineeritud kõik kasutajale saadaval olevad funktsioonid, operaatorid jms nagu kirjeldatud punktis 5.1.1.

6.1.1 Väärtustüübi lisamine

VelocyPack formaadile uue väärtustüübi lisamiseks tuli muuta peamiselt *Slice.h* ja *Slice.cpp* faile, mis sisaldavad väärtuse dekodeerimise realisatsiooni klassi *Slice*. Sisendina kasutab see vaid viita andmete esimesele baidile, mis määrab väärtuse tüübi. Vajadusel toimub edasine dekodeerimine vastavalt konkreetse tüübi eripärale.

Kuna enamikel juhtudel töötatakse otseste väärtustüüpidega ja sildid ei ole olulised, oli otstarbekas muuta *Slice* realisatsiooni nõnda, et lisatav väärtustüüp on läbipaistev. See võimaldas ka vältida muudatusi teegi teistes osades.

Selleks tuli luua uus meetod, mis kontrollib kas tegemist on sildi väärtustüübiga või ei ning tagastab vastavalt sellele viida andmete asukohale. Seejärel asendati kõik viited esimese baidi viida muutujale selle funktsiooni väljakutsumisega. Nõnda käitub *Slice* klass lisatud väärtustüübi korral täpselt nagu selles sisalduva väärtuse puhul.

Lisaks realiseeriti meetodid sildi olemasolu kontrolliks ja selle võtmiseks.

6.1.2 Mikrotestid

Andmeformaadi realisatsiooni valideerimiseks sooritati erinevatel testandmetel päringuid kõigi realiseeritud ning *IS NULL* ja *IS NOT NULL* operaatoritega. Kasutati nii ühe- kui ka mitmeosalisi teid, nii dokumente, kus tee koosnes vaid sõnaraamatutest kui ka dokumente, kus tees sisaldus(id) massiiv(id). Ebakõlasid ei leitud.

6.2 Statistika

Statistika realisatsioon loodi vastavalt projektile ning on osa andmetüübiga samast laiendusest lisades sellele järgnevad osad:

- *catalog.h* ja *catalog.c* (450 rida) – kataloogi realisatsioon statistika talletamiseks, kohandatud PipelineDB laienduse vastavast osast;
- *analyze.cpp* ja *analyze.h* (620 rida) – statistika kogumise realisatsioon nagu kirjeldatud punktis 5.2.1;
- *PathInfo.cpp* ja *PathInfo.h* (52 rida) – klass, mis talletab tee ja kas seal sisalduvad massiivid, kasutatakse dokumentidest absoluutteede hulga leidmisel;
- *PathValue.cpp* ja *PathValue.h* (35 rida) – eelmisega analoogne klass, mis lisaks tal-

letab tee väärtuse, kasutatakse @> ja <@ operaatorite statistika realisatsioonis;
- *vpack_sel.cpp* ja *vpack_sel.h* (1468 rida) – selektiivsuse arvutamise ja vastava statistika võtmise funktsioonid.

Statistika ja selektiivsuse arvutamiseks kasutatakse samu funktsioone, mida tavaliste väljade puhul. Algandmete leidmiseks luuakse kõigi dokumentide põhjal nende absoluutteede hulk ja arvutatakse statistika iga tee kohta eraldi. Sisuliselt ei erine see tabelist, millel on dokumendi iga absoluuttee jaoks vastavat tüüpi väli.

6.2.1 Paikapidavuse testid

Statistika testimiseks kasutati *EXPLAIN ANALYZE* käsku. Eristati kolme juhtu:

1. tee statistika, kus vastavad dokumendi elemendid olid sõnaraamatud;
2. tee statistika, kus vastavates dokumendi elementides sisaldus massiiv;
3. *IS NULL* ja *IS NOT NULL* päringud osalise tee kohta.

Kasutati erinevate suuruste ja omadustega andmehulki, sh erineva struktuuriga dokumentidest koosnevaid. Kõigi testide tulemused olid tegelike kirjete arvudega samas suurusjärgus ning üldjuhul oli erinevus alla saja kirje.

Laiendatud statistikat (näiteks korrelatsioonis olevate väärtuste jaoks) ei realiseeritud ja seega vastavaid päringuid ka ei testitud, kuid vastava funktsionaalsuse saab realiseerida analoogselt olemasoleva *CREATE STATISTICS* käsu tulemile.

6.3 Testiraamistik

Testiraamistik realiseeriti vastavalt projektile peamiselt Java programmeerimiskeeles mitmelõimelise rakendusena, kokku 5121 rida koodi.

Süsteemi keksekseteks osadeks on testid, mis tsükliliselt sooritavad andmebaasis teatud operatsiooni, mõõdavad selle kestust ja talletavad tulemuse spetsiaalsesse loendisse ning testide haldur, mis loob määratud arvu paralleelseid teste ja talletab nende tulemused jooksvalt andmebaasi.

Enne testide käivitamist tuleb vajadusel andmebaasist lugeda seal olevate dokumentide ID-d. Kuna suurte andmebaaside korral on see protsess väga aeglane realiseeriti vastav vahemälu, mida saab kasutada korduval sama andmebaasi kasutamisel.

Samuti on osades testides vajalik sisestada dokumente. Selleks realiseeriti juhuslike dokumentide generaator ning failipõhine dokumendilugeja JSON ja andmebaaside omaformaatides. Kuna dokumendigeneraatorist või -lugejast andmete lugemine võib viibida või osutuda aeglaseks realiseeriti ka dokumendipuhver, mis vajadusel kasutab mitut lõime allikast dokumentide lugemiseks (näiteks, kui ühe protsessorituumaga ei suudeta piisavalt dokumente genereerida või andmebaasi omaformaati jadastada).

Java virtuaalmasin sõltub mäluhalduses suuresti mälutihendusest. Algselt üritati süsteemi kasutada tihendusvabalt, kuid pikemate testide korral sai mälu otsa. Lahendusena kasutati Z mälutihendajat, mis on spetsiaalselt optimiseeritud tekitama võimalikult vähe ja võimalikult lühikesi pause virtuaalmasina töös. Testide läbiviimisel talletati ka mälutihendaja logid, kus on muuhulgas kirjas ka kriitiliste pauside pikkused.

Mugavamaks kasutamiseks realiseeriti käsurealiides kasutades Groovy programmeerimiskeelt. Selle eesmärk on luua määratud dokumendiallikas, käivitada testide haldur ning vajadusel katkestada testid ajalõpu saabumisel.

6.4 Jõudlustestide läbiviimine

Järgnevalt kirjeldatakse detailsemalt jõudlustestide läbiviimise süsteemi ning kuidas valideeriti testiraamistiku korrektsust.

6.4.1 Süsteem

Jõudlustestide läbiviimise süsteem realiseeriti vastavalt punkti 4.3.4 kirjeldusele Groovy programmeerimiskeeles, kokku 985 rida koodi.

Esmalt kontrollib süsteem kõigi vajalike sisendandmete õigsust ning vajadusel katkestab töö vastava veateatega.

Järgnevalt luuakse vajalikud failisüsteemi tõmmised andmebaasi andmetest ja konteinerist, koostatakse konteineri seadistuste fail, käivitatakse konteiner ning oodatakse andmebaasi käivitumist. Kõik vastavad operatsioonid realiseeriti sulunditena ning spetsiaalsesse pinusse talletatakse ka operatsiooni pöördoperatsioon, näiteks failisüsteemi tõmmise loomise puhul selle eemaldamine. Jõudlustesti lõppemisel või vigade tekkimisel käivitatakse kõik seni pinus talletatud operatsioonid vastavas järjekorras. See tagab, et alati eemaldatakse testi jooksvad andmed ning jäetakse süsteem puhtasse olekusse.

Seejärel käivitatakse perioodiliseks süsteemiandmete kogumiseks Prometheus-i lisad, protsesside info (*ps, smem* ja *lxc-info*) ja failisüsteemi info (*zfs iostat*) kogumise programmid. Andmed salvestatakse selleks spetsiaalselt loodud andmebaasi.

Enne jõudlustesti kogutakse lisaks failisüsteemi atribuudid (sh kettakasutus ja bloki suurus), andmebaasi seadistused ja süsteemi sisendid.

Lõpuks saab käivitada jõudlustesti vastavalt süsteemi sisendparameetritele.

Pärast jõudlustesti kogutakse uuesti failisüsteemi atribuudid, mis võimaldab hiljem võrrelda näiteks kettakasutuse muutust. Seejärel käivitatakse tagasipöörde protsess ning käivitatakse kõik vastavas pinus olevad käsud.

Erindite korral või mõne programmi väljumisel nullist erineva staatuskoodiga käivitatakse samuti tagasipöördeprotsess ning vastav märge tehakse ka andmebaasis.

6.4.2 Keskkond

Serveri operatsioonisüsteemi installeerimiseks kasutati selle omaniku pakutavat päästesüsteemi võimaldamaks läbivat ZFS failisüsteemi kasutamist, mis vaikimisi seadistusena ei ole toetatud. Edasine seadistus tehti vastavalt punkti 5.4.2 kirjeldusele.

6.4.3 Läbiviimine

Testisüsteemi automaatseks käivitamiseks vastavate parameetritega kasutati BASH skripte, mis loodi iga testitava aspekti jaoks eraldi.

Tulemusi nägi Grafana-st ning käesolevasse töösse graafikute lisamiseks realiseeriti spetsiaalne programm, mis võttis andmebaasist vastavad andmed ning viis need Gnuplot-i jaoks sobivasse formaati.

Kasutatud andmebaasid olid järgnevad:

- PostgreSQL kompileeriti selle *git* repositooriumist 17. veebruari 2019 seisuga;
- MongoDB versioon 4.0.9;
- MongoDB Enteprise versioon 4.0.9 (residentse talletusmootori testid);
- Percona Server for MongoDB versioon 4.0.9-4 (residentse talletusmootori testid).

6.4.4 Testiraamistiku valideerimine

Testiraamistiku tulemuste õigsuse kontrolliks kasutati võrdlust YCSB raamistikuga. Selleks seati üles MongoDB andmebaas ning täideti see 10^7 dokumendiga. Järgnevalt käitati 16 lõimega YCSB C töökoormust ja käesoleva töö koormust nr 6 kuni tulemused olid stabiilsed.

YCSB luges keskmiselt 83000 dokumenti sekundis ja käesoleva töö testiraamistik keskmiselt 81000 dokumenti sekundis. Seega on alust arvata, et testiraamistik on realiseeritud korrektselt.

6.5 Edasisi töid

Käesolevas töös realiseeriti põhifunktsionaalsus PostgreSQL-i kasutamiseks täisväärtusliku dokumendikoguna. Samas on see alles algus ning järgnevalt teostamist ootavaid töid on mitmeid:

1. erinevate toetatavate funktsioonide, näiteks *jsonb_agg* ja *json_populate_recordset* analoogide realiseerimine;
2. *jsonpath* päringukeele toe realiseerimine VelocyPack andmetüübile;
3. *pg_statistic* täiendamine nagu kirjeldatud punktis 5.2.1;
4. valikulise ja laiendatud statistika realiseerimine, sh vastavad *CREATE STATISTICS* käsu täiendused;
5. erinevate andmetüüpide loogiliste tüüpide identifikaatorite määramine ning vastava registri loomine sarnaselt CBOR formaadi omale;
6. VelocyPack formaadi teekide loomine erinevates programmeerimiskeeltes;
7. populaarsete dokumendiformaatide (MessagePack, CBOR, BSON) andmete transpordiks kasutamise võimaldamine;
8. populaarsete MongoDB teekide kohandamine PostgreSQL-i jaoks või nende laiendamine PostgreSQL-i toega;
9. ühilduvuskiht MongoDB-ga võimaldamaks olemasolevaid rakendusi lihtsamini PostgreSQL-ile üle viia;
10. PostgreSQL-XL jõudluse võrdlemine MongoDB-ga mitme andmebaasiserveri kasutamisel.

7 Jõudlustestide tulemuste analüüs

Käesolevas töös realiseeritud jõudlustestide eesmärk oli ennekõike erinevate ZFS failisüsteemi, operatsioonisüsteemi ja andmebaasiseadistuste mõju hindamine jõudlusele, mitte parima andmebaasisüsteemi leidmine. Kuna erinevaid võimalikke muutujate kombinatsioone on väga palju ning serveri kasutamise aeg oli piiratud, realiseeriti neist valik, mis suuresti kujunes testide vaheanalüüsi käigus.

Testide kestus oli 60 sekundit kui pole märgitud teisiti. Testid viidi läbi 10^6, 10^7 ja 10^8 kirjega: esimene loetakse hetkeliselt mällu, teine oli algselt mõeldud peaaegu täielikult mällu mahtuma, kuid reaalsuses loeti täielikult mällu hiljemalt testi keskpaigaks (vt joonis 12) ning kolmas simuleeris olukorda, kus andmed mällu ei mahu ning pea alati tuleb neid kettalt lugeda.

Peamiselt kasutati töökoormusi 1 (andmete sisestamine), 6 (andmete võtmine), 10 (andmete uuendamine), 11 (50% sisestamine, 50% võtmine), 13 (agregaatoperatsioon) ja 14 (transaktsioon). Testide varieeruvus on umbes 2%.

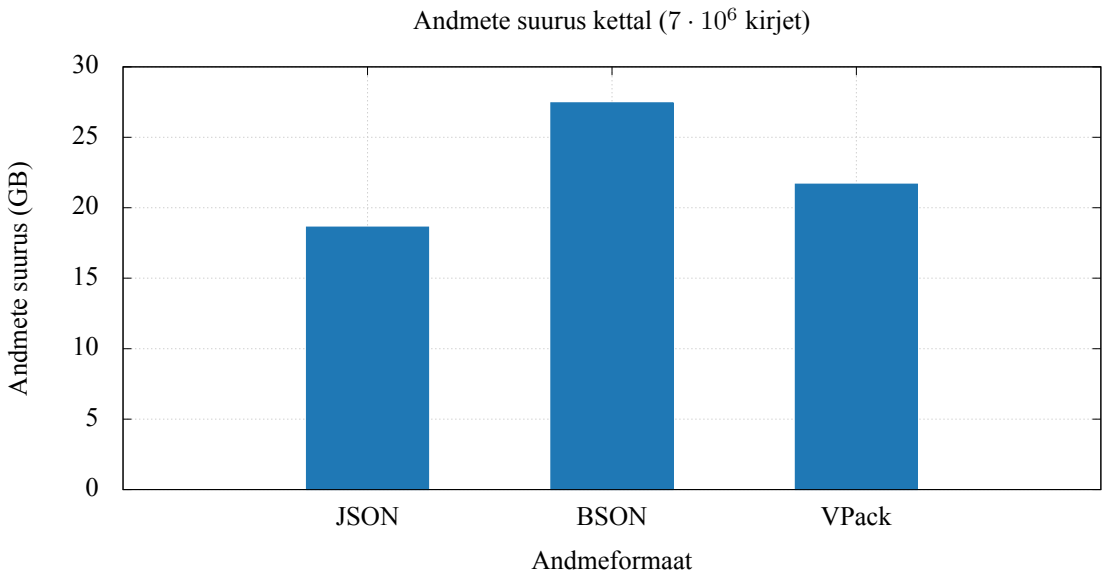
7.1 Andmete kompaktsus

Jõudlus oleneb suuresti andmete kompaktsusest, kuna mida kompaktsemad on andmed, seda rohkem neid mällu mahub ning seda kiirem on neid kettale kirjutada ja sealt lugeda.

7.1.1 Andmete suurus kettal

Joonisel 1 on toodud $7 \cdot 10^6$ GitHub-i dokumendi suurus erinevates andmeformaatides.

Näeme, et JSON formaat on kõige väiksemamahulisem, mis tuleneb sellest, et ei talletata väljade pikkusi. Talle järgneb VPack formaat, mis vajab 10% rohkem ruumi ning MongoDB BSON formaat, mis vajab 30% enam ruumi kui JSON formaadis andmed. *jsonb* formaadis andmete suurust toodud ei ole, kuna tegmeist on andmebaasisisese formaadiga.



Joonis 1: Andmete suurus kettal erinevate formaatide lõikes.

7.1.2 Andmebaaside pakkimise efektiivsus

Joonisel 2 on toodud andmete pakkimise suhe failisüsteemi taseme pakkimisel blokisuuruse järgi.

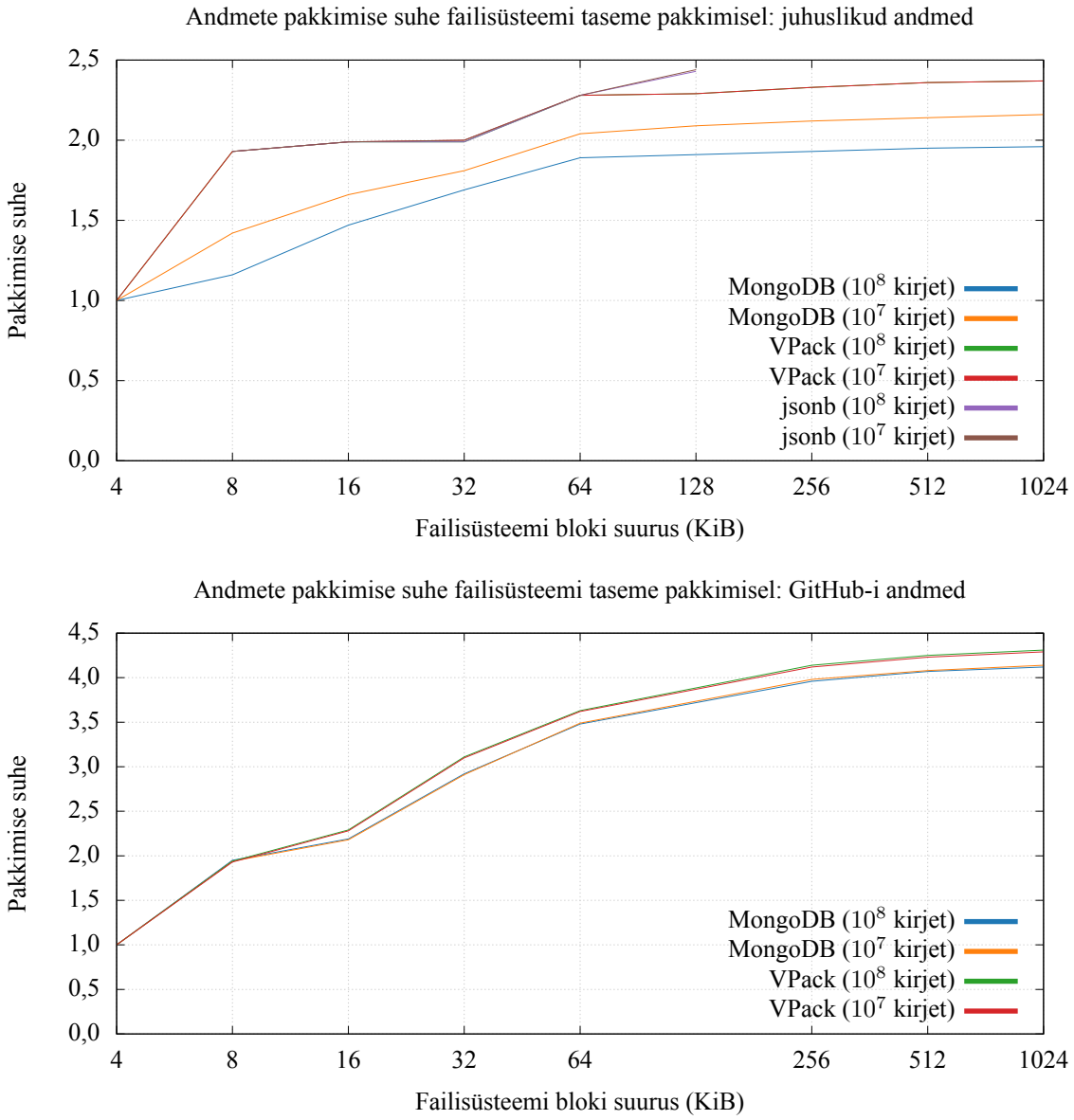
Näeme, et 4KiB blokisuurusel pakkimist ei toimu, kuna andmeid on liiga vähe.

GitHub-i andmete korral on nii MongoDB kui ka VPack-i pakkimise suhe sarnane, VPack-i oma veidi parem. Suhe tõuseb 8KiB blokisuurusel 2-ni ning kasvab kiirelt kuni 64KiB blokisuuruseni, mil suhe on 3,5, sealt edasi on tõus aeglasem.

Juhuslike andmete puhul on näha selged erinevused MongoDB ja PostgreSQL-i vahel. PosgreSQL-i korral tõuseb 8KiB blokisuurusel pakkimise suhe 2-ni olles stabiilne 64KiB blokisuuruseni, mil suhe tõuseb 2,3-ni. Sealt edasi on suhe VPack-i korral stabiilne, kuid *jsonb* korral tõuseb veel 128KiB blokisuurusel 2,4-ni. MongoDB puhul on tõus märgatavalt aeglasem, jõudes umbes 2-ni alles 64KiB blokisuuruse korral.

Seega on andmete kompaktsuse aspektist üldjuhul otstarbekas kasutada 64KiB blokisuurust. PostgreSQL-i puhul peaks see alati olema vähemalt 8KiB, mis ühtib ka vaikimisi kasutatava lehe suurusega.

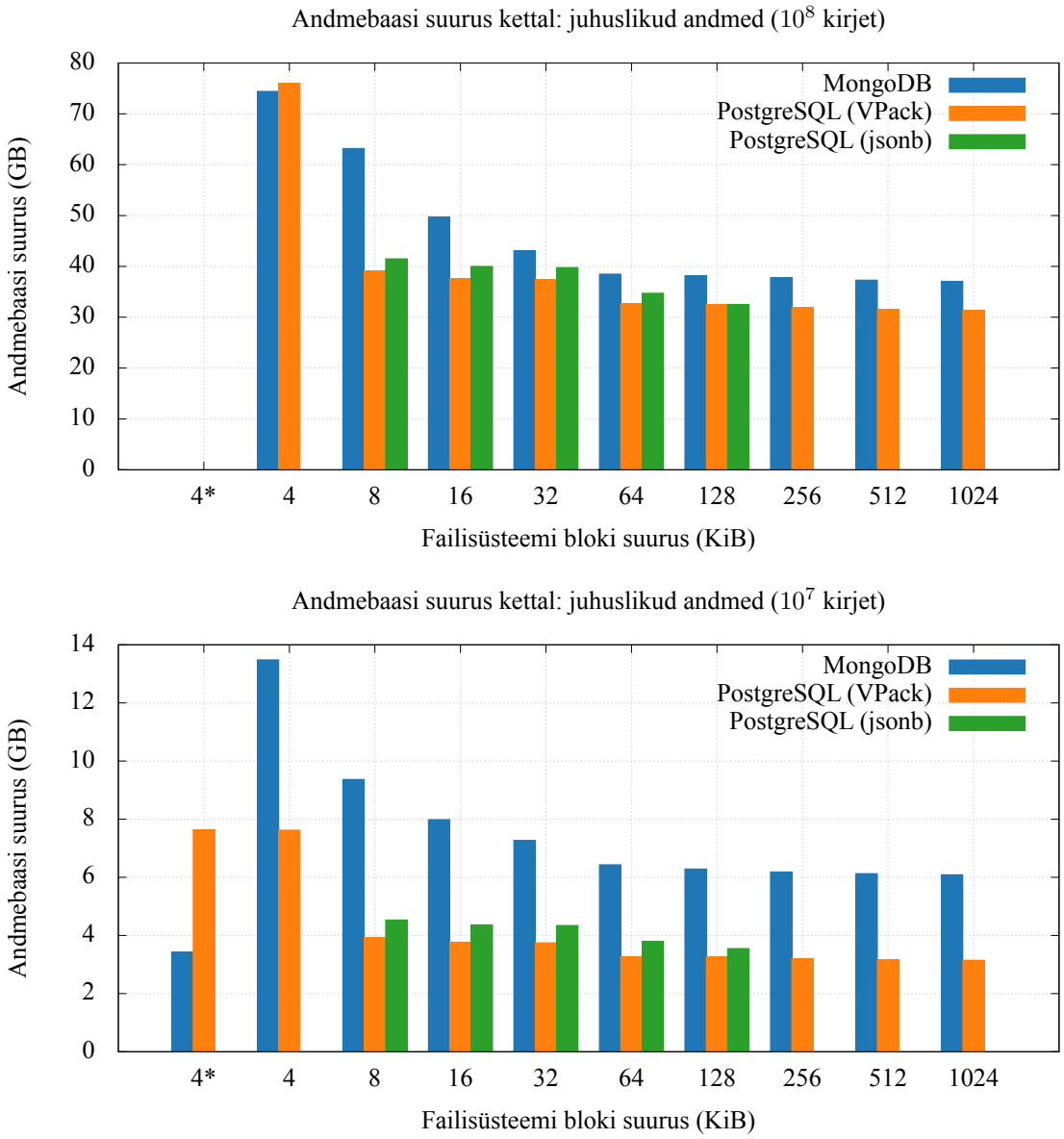
Käesoleva töö jõudlustestide tulemuste talletamise andmebaas kasutas 512KiB blokisuurust ja *gzip* pakkimisalgoritmi, millega saavutati pakkimise suhe 7,58, andmeid oli kokku üle 2TB. Enamjaolt on kasutatud relatsioonilist mudelit, vaid Prometheus-i lisade andmete tabel kasutab atribuutide jaoks ka *jsonb* formaadis välja.



Joonis 2: Failisüsteemi pakkimise suhe failisüsteemi taseme pakkimisel.

7.1.3 Andmebaaside suurus kettal

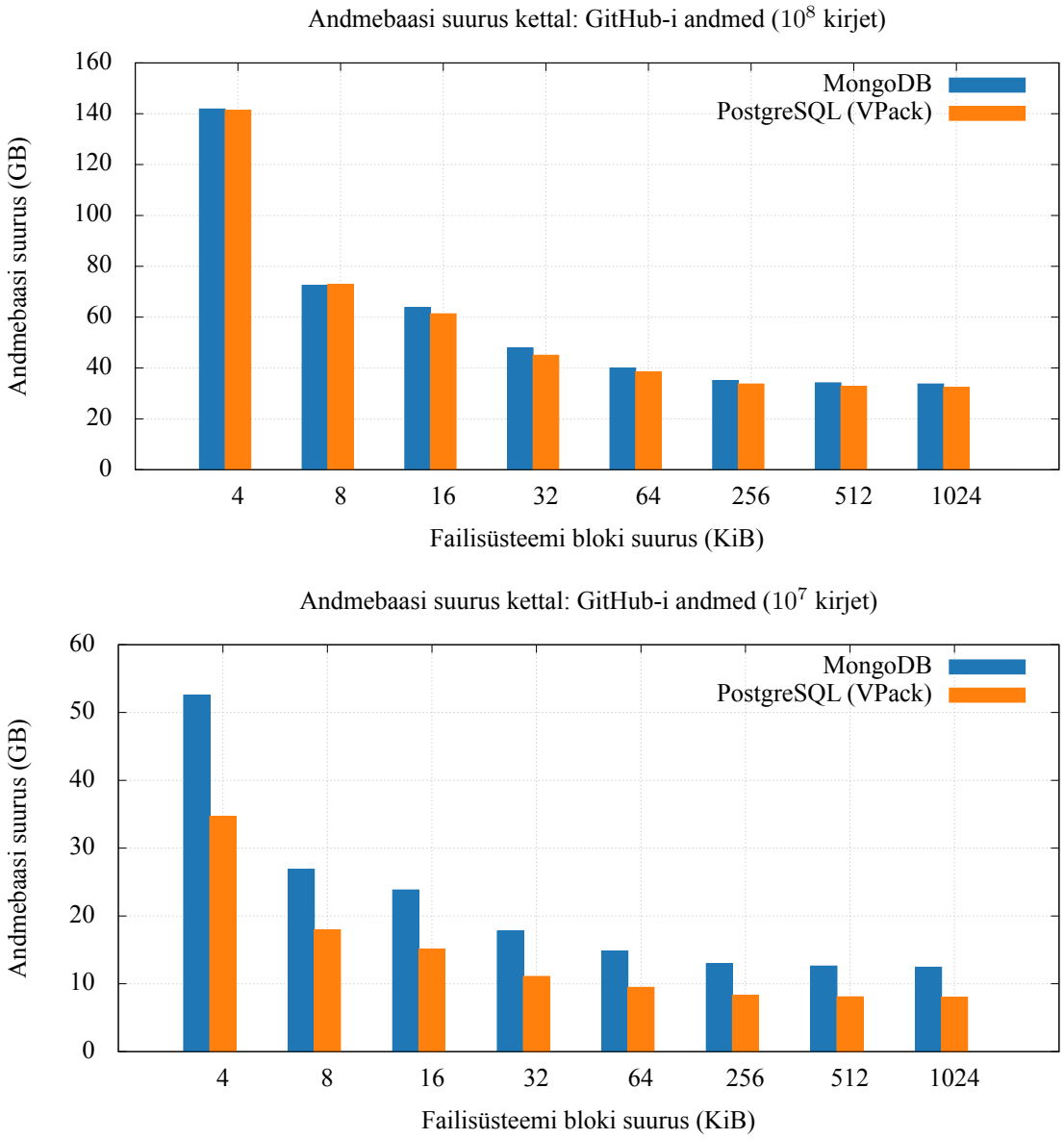
Joonisel 3 on toodud juhuslike andmetega ja joonisel 4 GitHub-i andmetega andmebaaside suurus kettal failisüsteemi ja andmebaasi taseme pakkimist kasutades blokisuuruse järgi. MongoDB kasutab Snappy pakkimisalgoritmi, PostgreSQL ja ZFS LZ perekonna algoritme.



Joonis 3: Genereeritud andmetega andmebaaside suurus kettal failisüsteemi pakkimist kasutades (v.a. tärniga märgitud, mis kasutavad vaid andmebaasi taseme oma ja *jsonb*, mis kasutab mõlemat).

Näeme, et andmebaasi tasandi pakkimine VPack formaadi korral mingil põhjusel ei toimi, kuigi *CREATE TYPE* käsu kohaselt on pakkimine lubatud.

MongoDB andmebaasi tasandi pakkimine on väga efektiivne: 10^7 kirjet mahutatakse



Joonis 4: GitHub-i andmetega andmebaaside suurus kettal failisüsteemi taseme pakkimist kasutades.

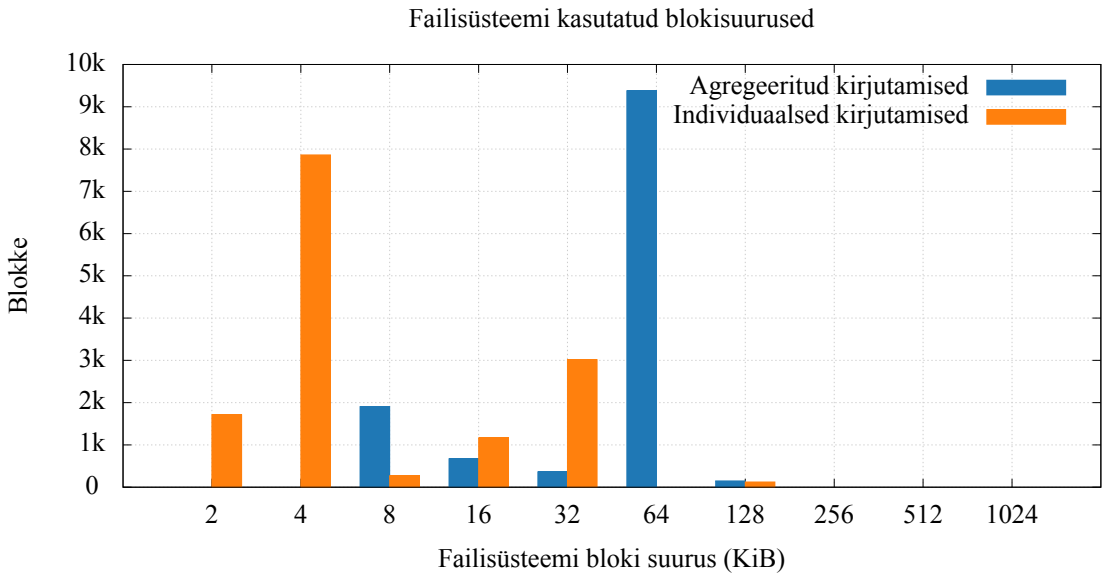
3,5GB sisse, samas kui failisüsteemi taseme pakkimise korral on minimaalne maht 6GB. VPack ja *jsonb* formaadi korral saavutatakse sama pakkimise suhe 64KiB blokisuuruse korral.

Arvestades punkti 7.1.2 tulemusi on tõenäoline, et need erinevused tulenevad andmete olemusest ning GitHub-i andmete korral oleks MongoDB andmebaasi tasandi ja failisüsteemi tasandi pakkimise suhted sarnased.

7.2 Blokisuurus

Andmete kompaktsus oleneb suuresti blokisuurusest, kuna mida suurem blokisuurus, seda parem pakkimise suhe saavutatakse. Samas on suurte blokisuuruste miinuseks kulukus väikeste muudatuste tegemisel – minimaalne loetav ja kirjutatav suurus on andmeid sisaldava bloki suurus isegi vaid ühte biti muutmisel, lisaks tuleb vastava bloki andmed eelnevalt kettalt mällu lugeda. Samuti võib see tekitada ketta andmesiini küllastumist ning vähendada pooljuhtketaste eluiga. Väikeste andmeblokkide (ennekõike 4KiB ja väiksemad) miinuseks on jällegi suur metaandmete ülekulu.

Tasub tähele panna, et määratud blokisuurus on maksimaalne lubatud suurus. Tegelikkuses on suur osa kirjutatavaid blokke sellest väiksemad, nagu illustreerib joonis 5, kus on kujutatud tegelikult kasutatud blokisuurused 128KiB blokisuurusel töökoormuse 1 korral. Erandiks on juhud, kus andmed ei mahu ühte blokki – siis peavad kõik blokid olema sama suured.



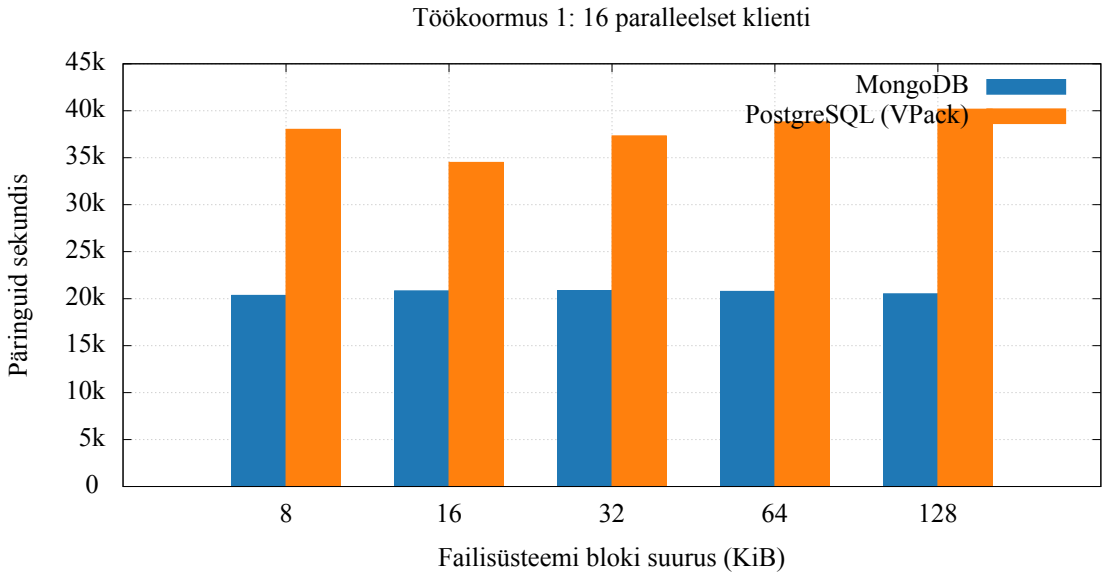
Joonis 5: Failisüsteemi tegelikult kasutatud blokisuurused 128KiB blokisuurusel töökoormuse 1 korral.

7.2.1 Päevikul

PostgreSQL kasutab päeviku jaoks vaikimisi 8KiB andmeblokke, MongoDB päeviku bloki suurust ei ole dokumentatsioonis välja toodud.

Päevikut kirjutatakse järjestikuliselt teatud failisuuruseni, misjärel jätkatakse uue failiga. Kuna päevikusse kirjutatakse pärast iga muudatust on selle viimased kirjutatud andmeblokid pea alati mälus olemas. Päevikus olevaid andmeid loetakse üldjuhul vaid tõrkest taastamisel, seega pole selle lugemise aspekt tavaolukorras oluline.

Joonistel 6 ja 7 on toodud päeviku failisüsteemi blokisuuruse mõju jõudlusele.

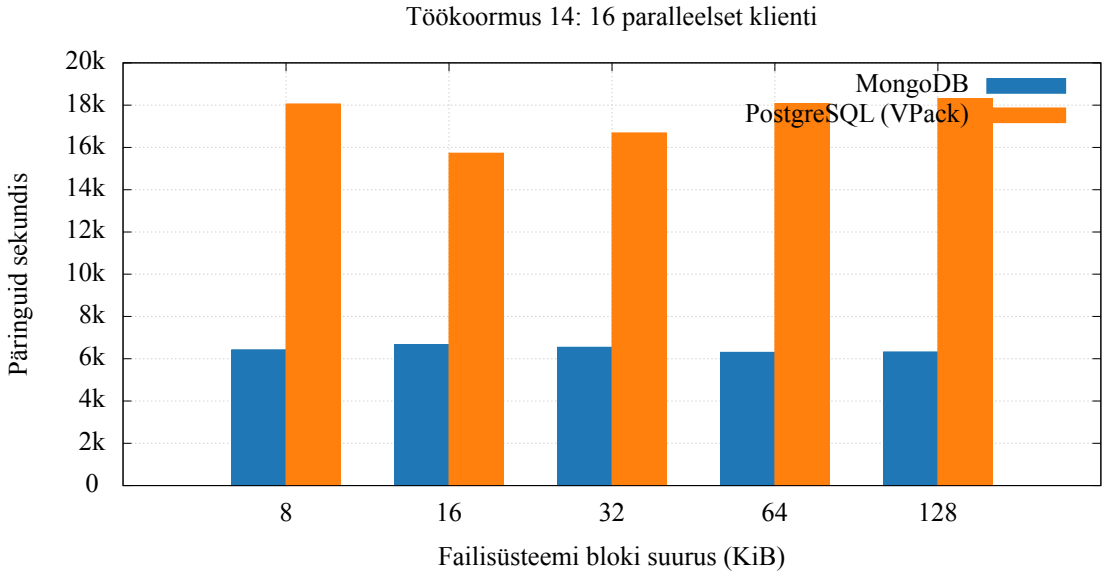


Joonis 6: Päeviku failisüsteemi blokisuuruse mõju jõudlusele, töökoormus 1.

Näme, et PostgreSQL-i korral saavutatakse parim jõudlus 8KiB, 64KiB ja 128KiB blokisuurustel. 16KiB blokisuurusel on jõudlus töökoormuste 1 ja 14 puhul 14% parimast tulemusest madalam. 32KiB blokisuurusel on sarnaselt jõudlus keskmiselt 7% parimast madalam. Kui töökoormuse 14 korral pole 8KiB ja 128KiB blokisuuruste kasutamisel märkimisväärset vahet, siis töökoormuse 1 puhul on 8KiB blokisuuruse kasutamine 5% aeglasem.

Seega tuleks PostgreSQL-i päeviku jaoks alati kasutada 8KiB või 128KiB blokisuurust.

MongoDB korral märkimisväärsed erinevused puuduvad.

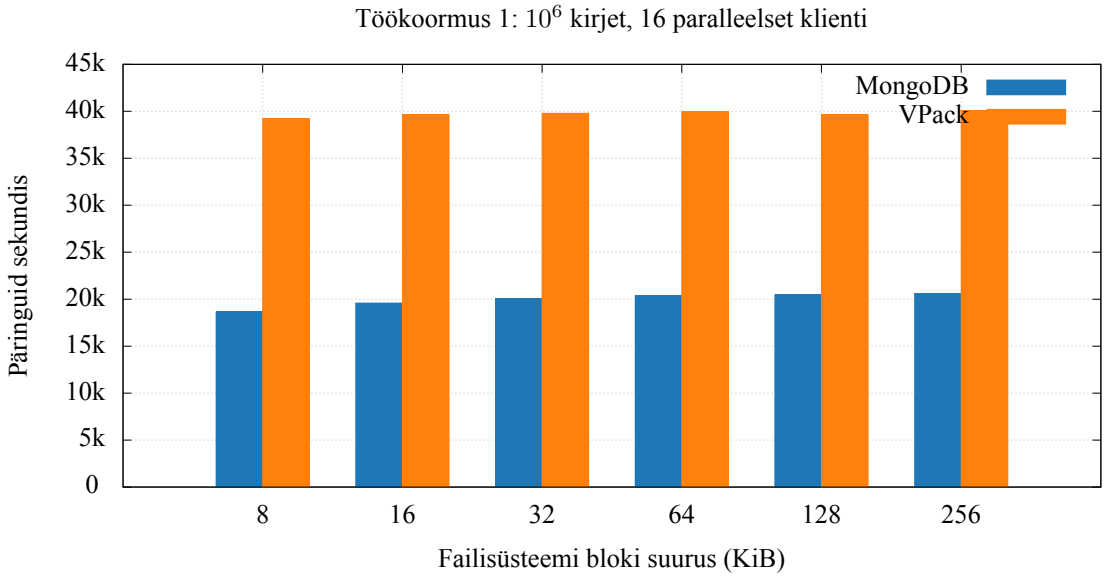


Joonis 7: Päeviku failisüsteemi blokisuuruse mõju jõudlusele, töökoormus 14.

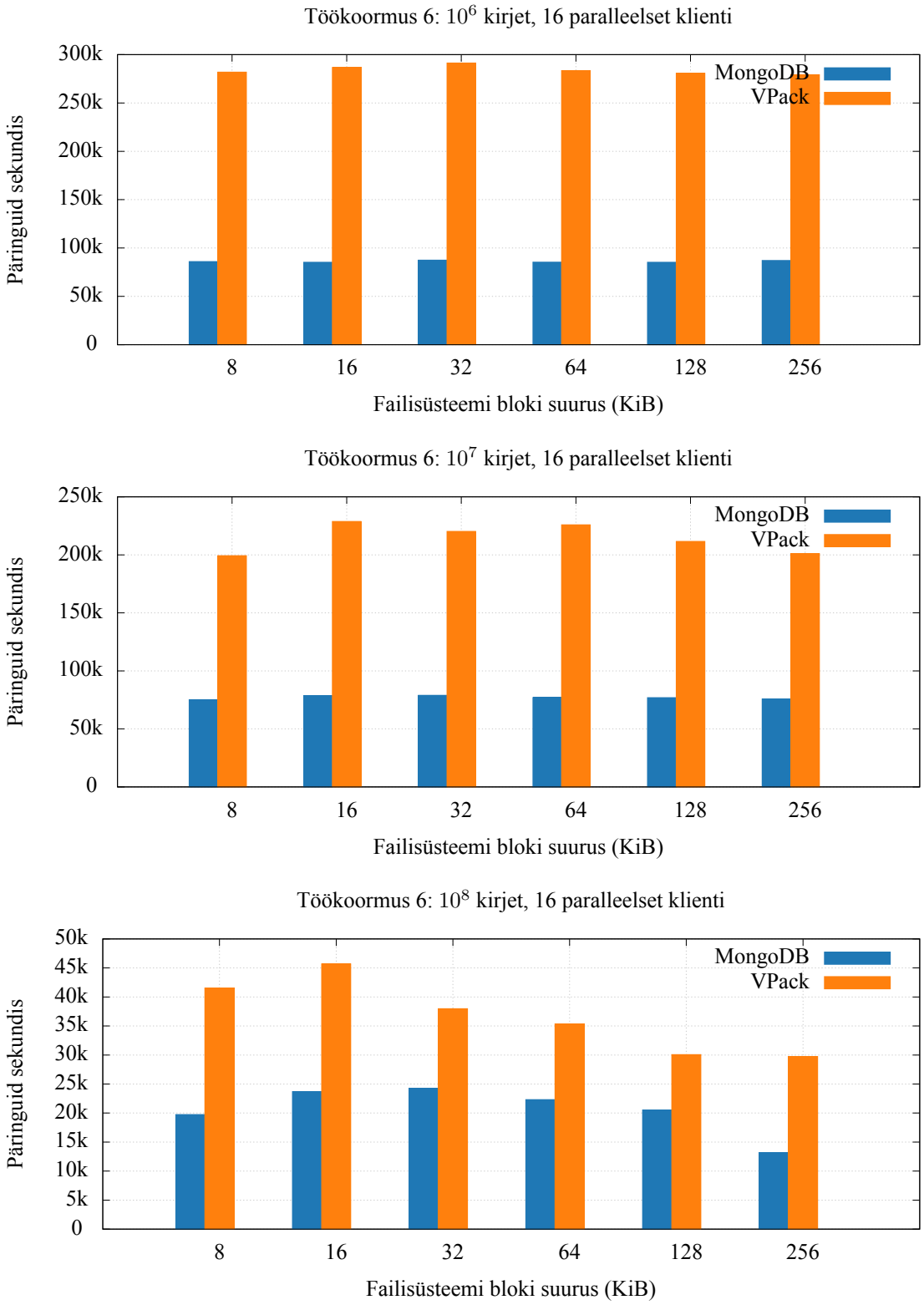
7.2.2 Andmebaasil

Andmebaasi blokisuuruse mõju dokumendikogule on erinev relatsioonilise mudeli omast, kuna tavaliselt puudub võimalus osaliseks dokumentide uuendamiseks kettal – muudetud dokumendid kirjutatakse alati täielikult uuesti, mis suuremate dokumentide korral võib tähendada märkimisväärset ülekulu.

Joonistel 8 ja 9 on toodud andmebaasi blokisuuruse mõju jõudlusele.



Joonis 8: Andmebaasi failisüsteemi blokisuuruse mõju jõudlusele, töökoormus 1.

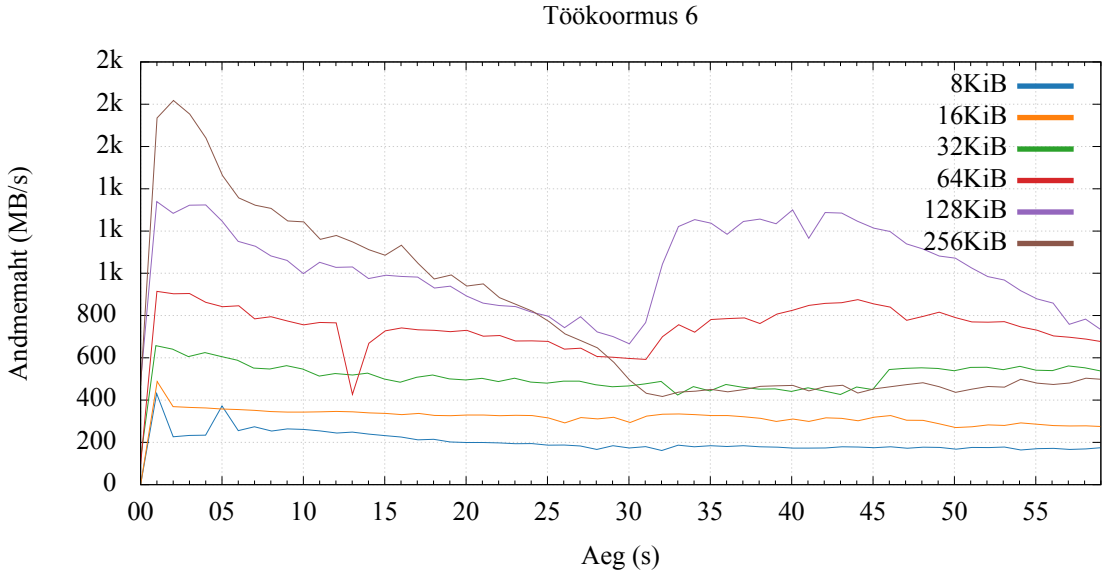


Joonis 9: Andmebaasi failisüsteemi blokisuuruse mõju jõudlusele, töökoormus 6.

Näeme, et PostgreSQL-i korral blokisuurus andmete sisestamise jõudlust ei mõjuta, kuid MongoDB puhul on eelis suurematel blokisuurustel: 8KiB blokisuuruse kasutamine on 11% aeglasem, kui 64KiB kasutamine. See tuleneb arvatavasti asjaolust, et WiredTiger-i *leaf_page_max* väärtus on vaikimisi 32KiB.

Andmete lugemine on nii PostgreSQL-i kui ka MongoDB korral parima jõudlusega 16KiB blokisuurusel. Suuremate andmemahtude korral tuleb esile suurte blokisuuruste negatiivne mõju, kuna kettalt loetakse vajalikust märgatavalt rohkem andmeid, millest enamik pole vajalikud: 256KiB blokisuuruse kasutamine on PostgreSQL-i korral 34% aeglasem ning MongoDB puhul 44% aeglasem.

Joonistel 10 ja 11 on toodud failisüsteemi blokisuuruse mõju ketta kasutusele.

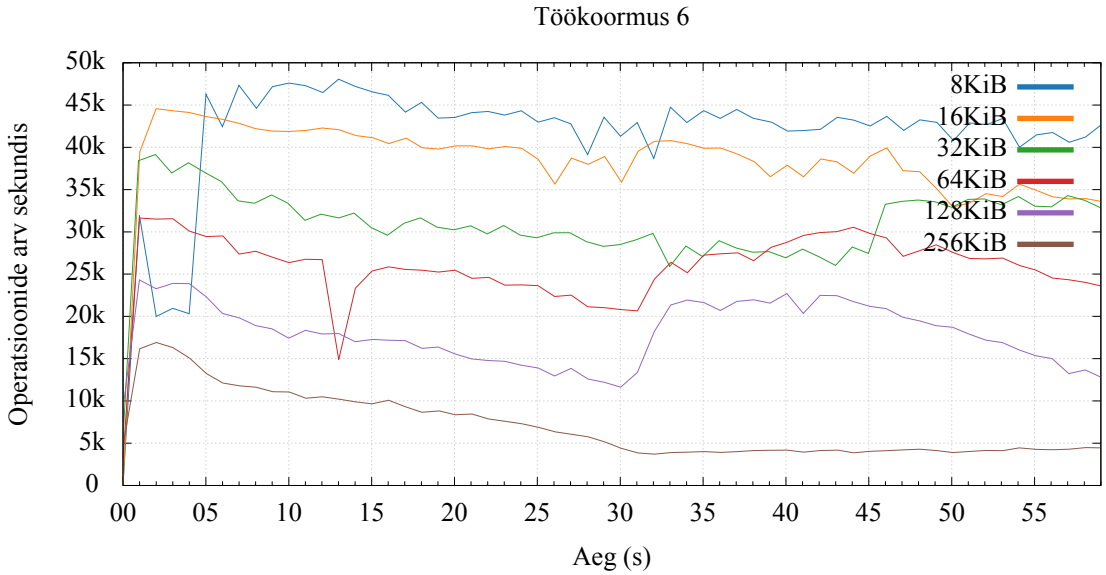


Joonis 10: Andmebaasi failisüsteemi blokisuuruse mõju kettaliidese andmemahule, töökoormus 6.

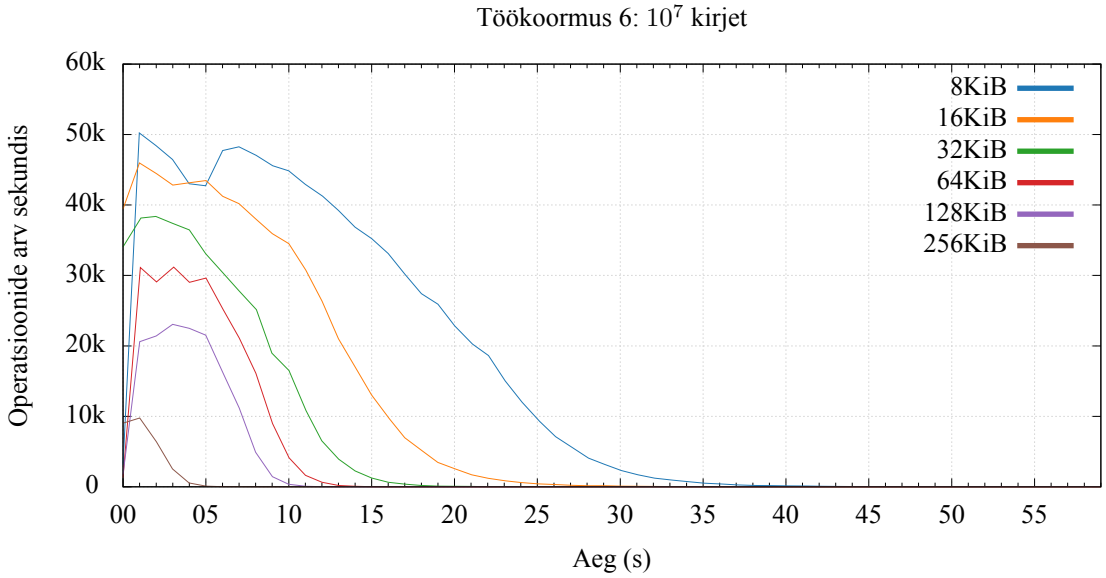
Näeme, et suuremate blokisuuruste korral on ketta operatsioonide arv väiksem, kuid edastatav andmemaht vastavalt suurem.

See tähendab ka seda, et andmete failisüsteemi puhvrisse laadimine võtab märgatavalt vähem aega, mis võib anda jõudluseelise järjestikuste või lähedal asuvate andmete lugemisel, näiteks aegreaandmebaasides. Seda illustreerib joonis 12, kus on toodud andmebaasi mällu lugemise kiiruse sõltuvus blokisuurusest kirjete juhusliku lugemise korral. Näeme, et 256KiB blokisuuruse korral loetakse terve andmebaas mällu vaid 5 sekundiga, samas kui 8KiB korral läheb selleks pea 40 sekundit. Aeglasemate ketaste või liideste korral võib suurematel blokisuurustel tekkida küllastumine, mis jõudlusele negatiivselt mõjub.

Seega on andmebaasi blokisuuruse valik kompromiss pakkimise suhte ja lugemise jõudluse vahel.



Joonis 11: Andmebaasi failisüsteemi blokisuuruse mõju kettaliidese operatsioonide arvule, töökoormus 6.



Joonis 12: Andmebaasi mällu lugemise kiiruse seos blokisuurusega, töökoormus 6.

Antud testi andmebaasid kopeeriti enne testi vastava blokisuurusega köitele, mille tulemusena saavutati vastava suurusega blokkide kasutamine. Tavaolukorras see nii ei ole, nagu illustreeritud joonisel 5. PostgreSQL-i puhul on seda töösüsteemis eeldatavasti võimalik saavutada *pg_repack* utiliidiga, mis tabelid neid tihendades ümber kirjutab.

Täiendavalt tasub uurida andmebaasi blokisuuruse mõju ajutistele tabelitele, mille asukohta saab PostgreSQL-is vajadusel *temp_tablespaces* parameetri abil määrata.

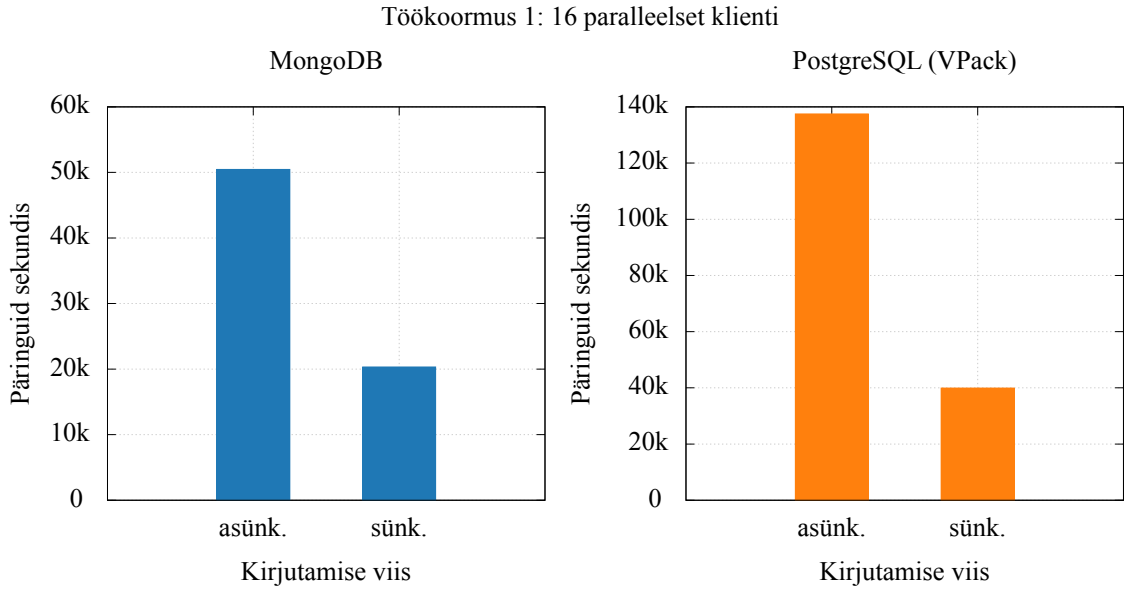
7.3 Vähendatud turvalisusgarantii

Vähendatud turvalisusgarantii mõjutab vaid kirjutamise ja sellega seotud operatsioonide jõudlust. Olenevalt seadistusest on mõnesaja millisekundiline aken, mille jooksul võib tõrke korral tekkida andmekadu.

7.3.1 PostgreSQL

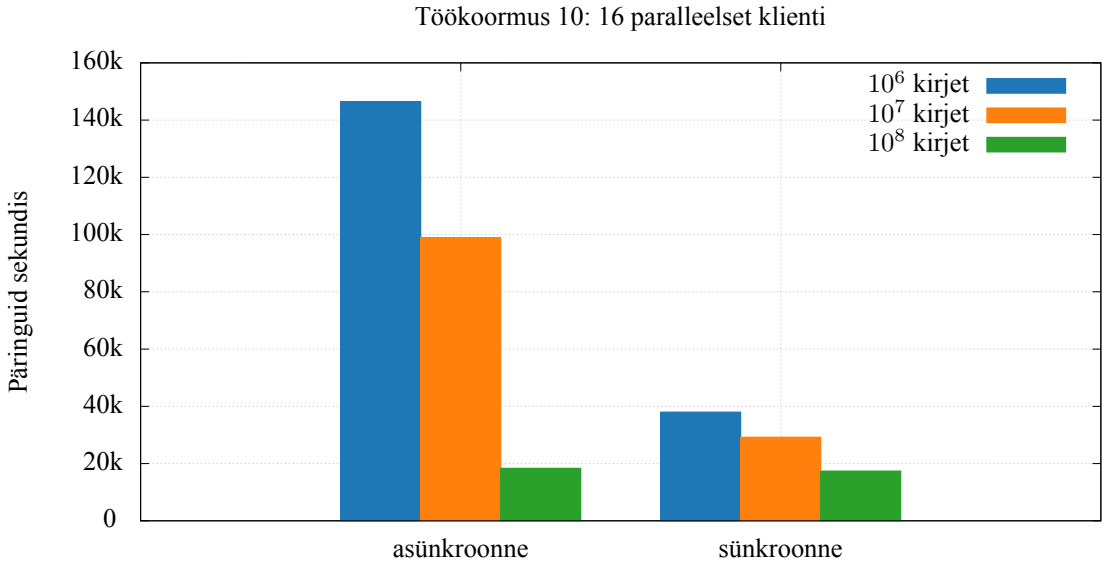
PostgreSQL-i test viidi läbi parmeetritega *wal_writer_delay=100* ja *wal_writer_flush_after=1MB*, mis tähendab, et andmed kirjutati päevikusse maksimaalselt iga 300ms järel või 1MB andmete kogunemisel.

Joonistel 13 ja 14 on toodud vähendatud turvalisusgarantii mõju PostgreSQL-i jõudlusele.



Joonis 13: Vähendatud turvalisusgarantii mõju jõudlusele, töökoormus 1.

Näeme, et täieliku turvalisusgarantiiga (sünkroonne) andmete sisestamine on 70% aeg-



Joonis 14: Vähendatud turvalisusgarantii mõju PostgreSQL-i jõudlusele, töökoormus 10.

lasem, kui vähendatud turvalisusgarantiiga (asünkroonne). Andmete sünkroonne uuendamine on 74% aeglasem, kui asünkroonne juhul, kui kõik andmed mahuvad mällu ja 5% aeglasem, kui andmed ei mahu mällu.

Seega tasub võimalusel kindlasti kaaluda vähendatud turvalisusgarantii kasutamist, mis võib samuti pakkuda ajutist leevendust jõudlusprobleemide korral olemasolevates süsteemides.

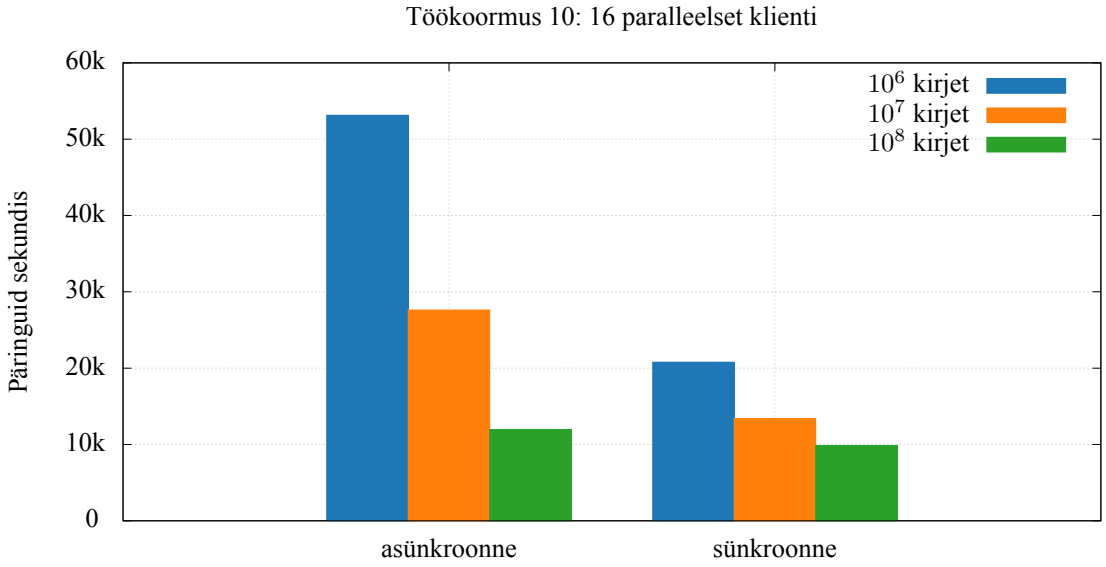
7.3.2 MongoDB

MongoDB test viidi läbi parameetriga *storage.journal.commitIntervalMs=300*, mis tähendab, et andmed kirjutati päevikusse iga 300ms järel.

Joonistel 13 ja 15 on toodud vähendatud turvalisusgarantii mõju MongoDB jõudlusele.

Näeme, et täieliku turvalisusgarantiiga (sünkroonne) andmete sisestamine on 60% aeglasem, kui vähendatud turvalisusgarantiiga (asünkroonne). Andmete sünkroonne uuendamine on 60% aeglasem, kui asünkroonne juhul, kui kõik andmed mahuvad mällu ja 17% aeglasem, kui andmed ei mahu mällu.

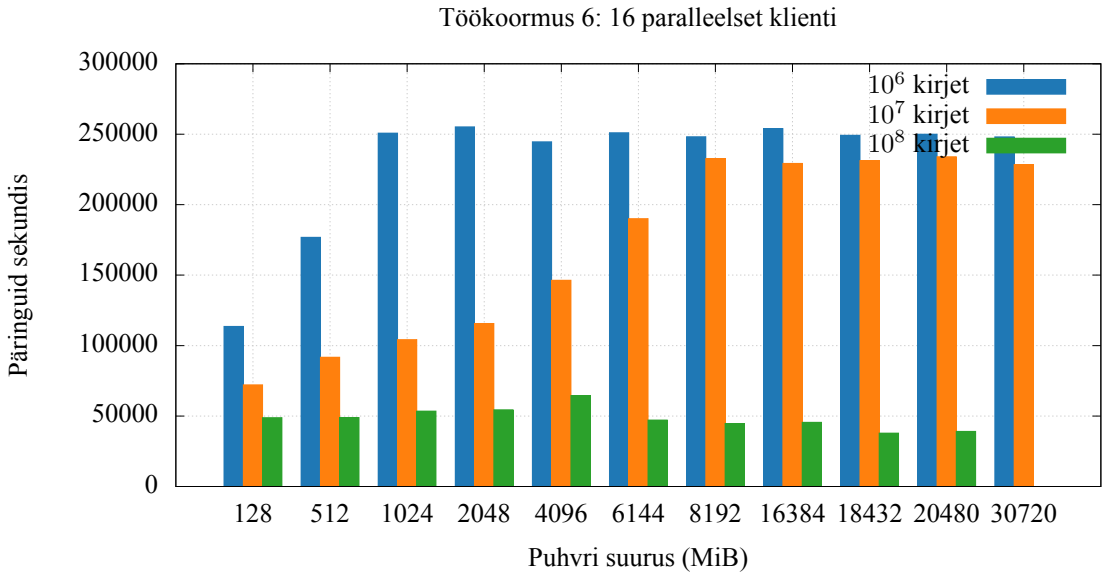
Seega on MongoDB vähendatud turvalisusgarantiiga käitlemisel pisut kiirem, kui PostgreSQL täieliku turvalisusgarantiiga, kuid võrdsetel alustel võrdlemisel siiski märkimisväärselt aeglasem.



Joonis 15: Vähendatud turvalisusgarantii mõju MongoDB jõudlusele, töökoormus 10.

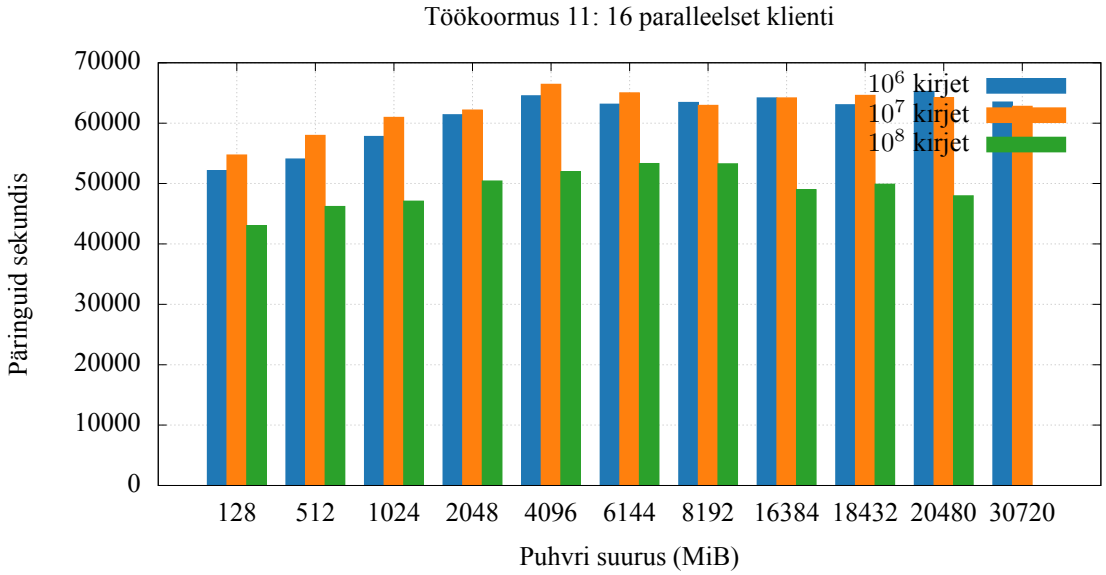
7.4 Andmebaasi puhvrid

Joonistel 16, 17 ja 18 on toodud PostgreSQL-i *shared_buffers* seade mõju mõju jõudlusele. Testi pikkus oli 3 minutit.

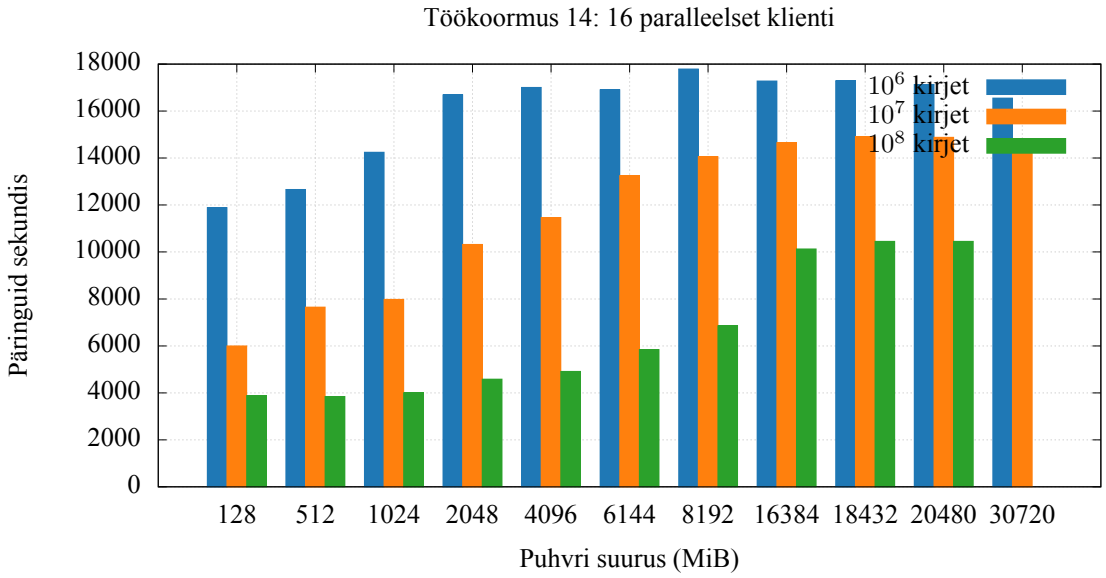


Joonis 16: Andmebaasi puhvrite suuruse mõju PostgreSQL-i jõudlusele, töökoormus 6.

Näeme, et töökoormuse 6 puhul saavutati maksimaalne kasu 10^6 kirje korral 1GiB juures ja 10^7 kirje korral 8GiB juures, mis vastab nende andmebaaside suurusele kettal ning 10^8 kirje korral 4GiB juures. Kui esimesel kahel juhul jäi pärast seda jõudlus stabiilseks, siis viimasel juhul hakkas pärast seda jõudlus hoopis langema, mis tuleneb puhvrite haldamise



Joonis 17: Andmebaasi puhvrite suuruse mõju PostgreSQL-i jõudlusele, töökoormus 11.



Joonis 18: Andmebaasi puhvrite suuruse mõju PostgreSQL-i jõudlusele, töökoormus 14.

ülekulust ja asjaolust, et puhvrite efektiivsus on sel juhul minimaalne, kuna valdav osa andmeid tuleb kettalt lugeda.

Töökoormuse 11 korral saavutati maksimaalne kasu 10^6 ja 10^7 kirje korral 4GiB juures ning 10^8 kirje korral 6GiB juures. Viimasel juhul hakkas suuremate puhvrite juures jällegi jõudlus langema.

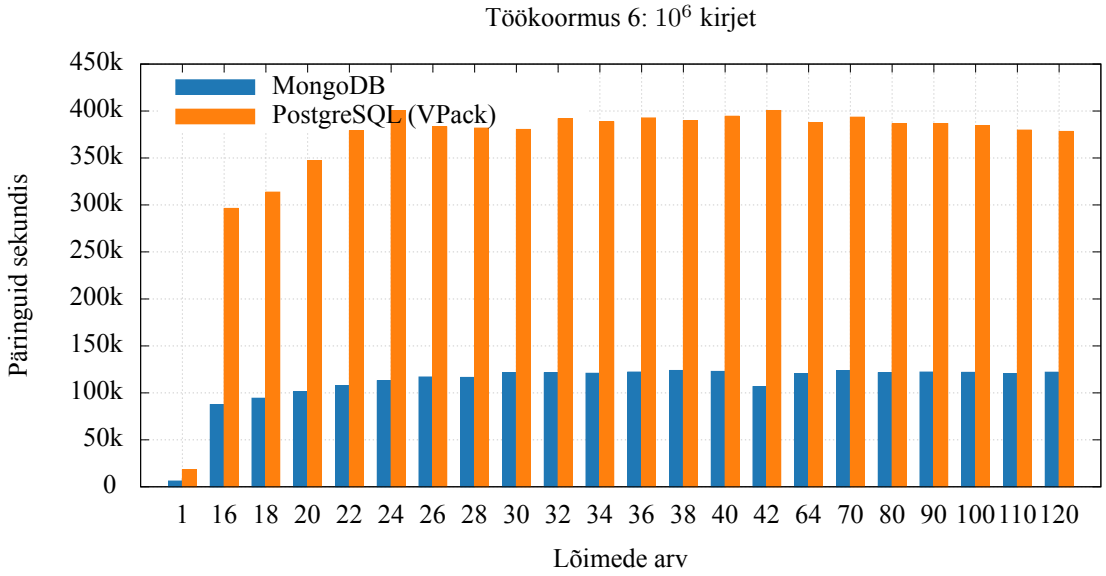
Töökoormuse 14 korral saavutati maksimaalne kasu 10^6 kirje korral 4GiB juures ning 10^7 ja 10^8 kirje korral 18GiB juures. Erinevus eelneva kahe töökoormusega tuleneb arvatavasti asjaolust, et antud juhul on andmete võtmise päringute arv sekundis märgatavalt väiksem ning negatiivne mõju ei jõudnud testi jooksul esile tulla.

Seega tuleb puhvrite suurus valida vastavalt nende efektiivsusele – madala tabamuskiiruse korral mõjub liiga suurte puhvrite kasutamine jõudlusele negatiivselt. Samas ei saa toetuda vaid failisüsteemi puhvritele, kuna see on üle 50% aeglasem.

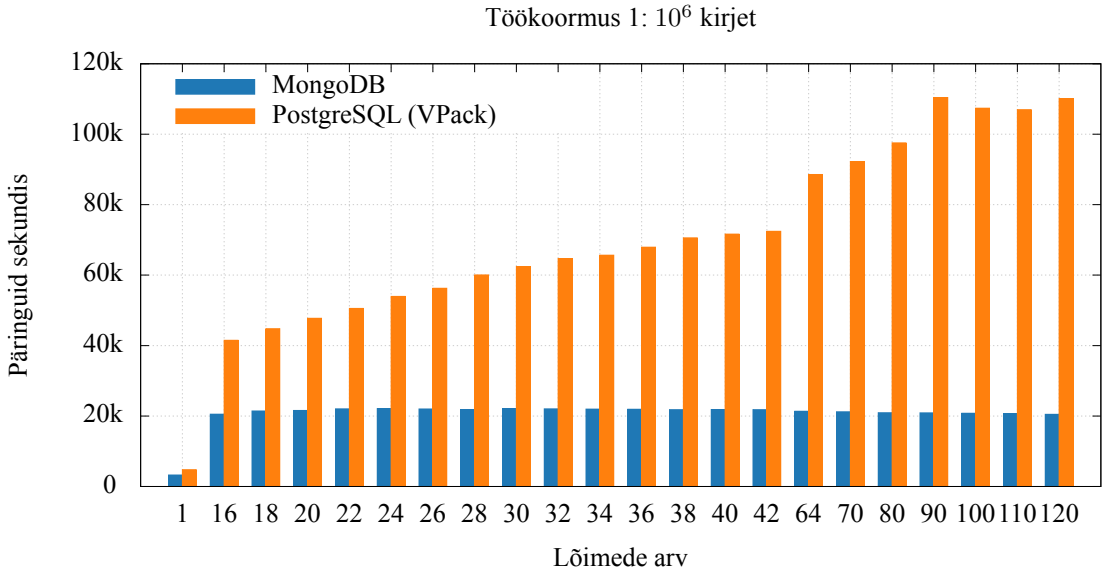
7.5 Lõimede arv

Suurem lõimede arv mõjub jõudlusele positiivselt, kuid teatud hetkel jõutakse küllastumiseni ning jõudlus võib hoopiski langema hakata. Sel juhul on otstarbekas kasutada ühenduste puuli.

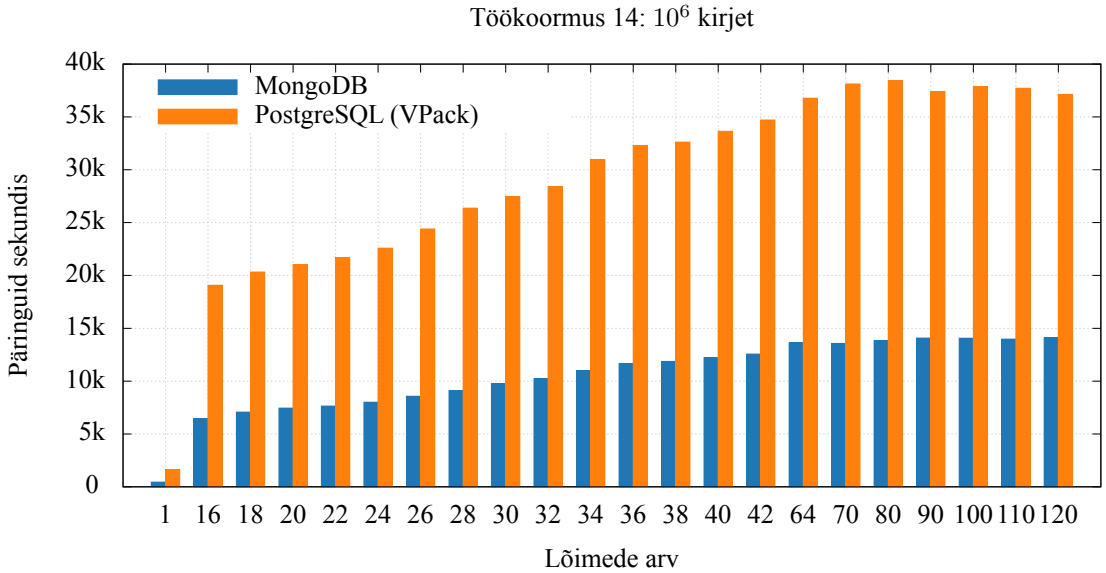
Joonistel 19, 20 ja 21 on toodud lõimede arvu mõju jõudlusele.



Joonis 19: Lõimede arvu mõju jõudlusele, töökoormus 6.



Joonis 20: Lõimede arvu mõju jõudlusele, töökoormus 1.



Joonis 21: Lõimede arvu mõju jõudlusele, töökoormus 14.

Näeme, et PostgreSQL-i korral saavutab andmete lugemine maksimaalse jõudluse 24 lõime juures, mis ühtib saadaolevate protsessori lõimede arvuga. Töökoormuste 1 ja 14 jõudlus kasvab lineaarselt ja maksimaalne jõudlus saavutatakse andmete sisestamisel 90, uuendmisel 120 ja töökoormuse 14 korral 80 lõime juures.

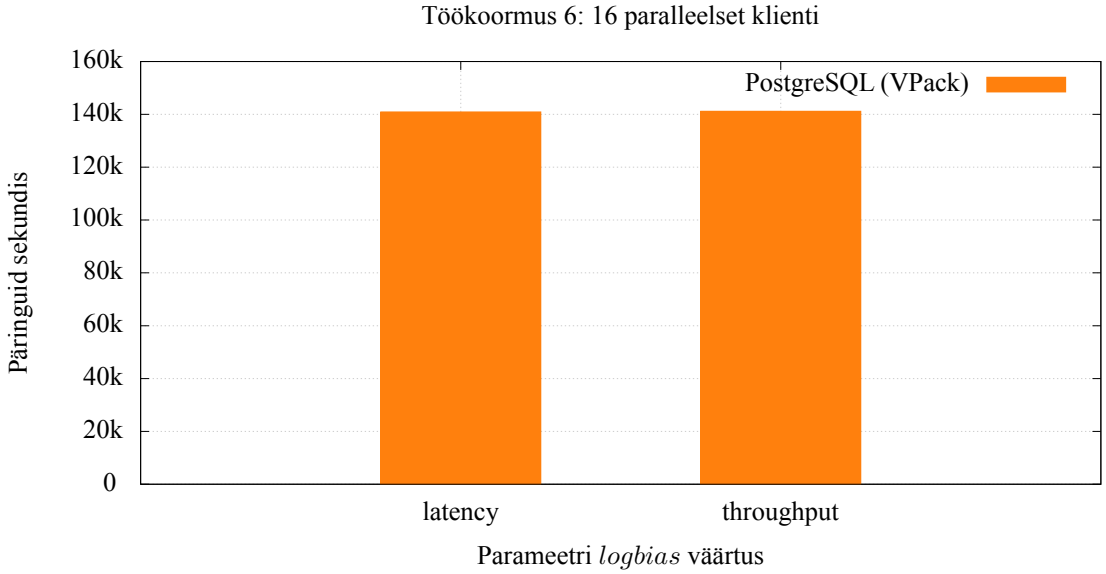
MongoDB puhul saavutab andmete sisestamine maksimaalse jõudluse 24 lõime juures ja andmete lugemine 30 lõime juures. Koormuse 14 jõudlus kasvab lineaarselt kuni 64 lõimeni.

Ühelgi juhul ei jõua MongoDB sarnase jõudluseni kui PostgreSQL 16 lõime korral.

7.6 *logbias=throughput*

ZFS-i seadet *logbias* on lähemalt kirjeldatud punktis 3.6.2. *logbias=throughput* kasutamisel andmebaasi köitel tekib potensiaalselt fragmenteerumine, mis hakkab lugemise jõudlust negatiivselt mõjutama.

Joonisel 22 on toodud parameetri mõju PostgreSQL-i jõudlusele. Testis kasutatud andmebaasid täideti esmalt vastavat seadet ja 8KiB blokisuurust kasutaval köitel andmetega ning viidi läbi koormustestid. ZIL-ina kasutati teist pooljuhtketast.



Joonis 22: *logbias=throughput* andmebaasi köitel kasutamise mõju PostgreSQL-i jõudlusele, töökoormus 6.

Näeme, et negatiivset stsenaariumit tekitada ei õnnestunud ning lugemise jõudlus ei muutunud. Ka kettaoperatsioonide arvus ja andmemahus märgatavaid erinevusi ei esinenud.

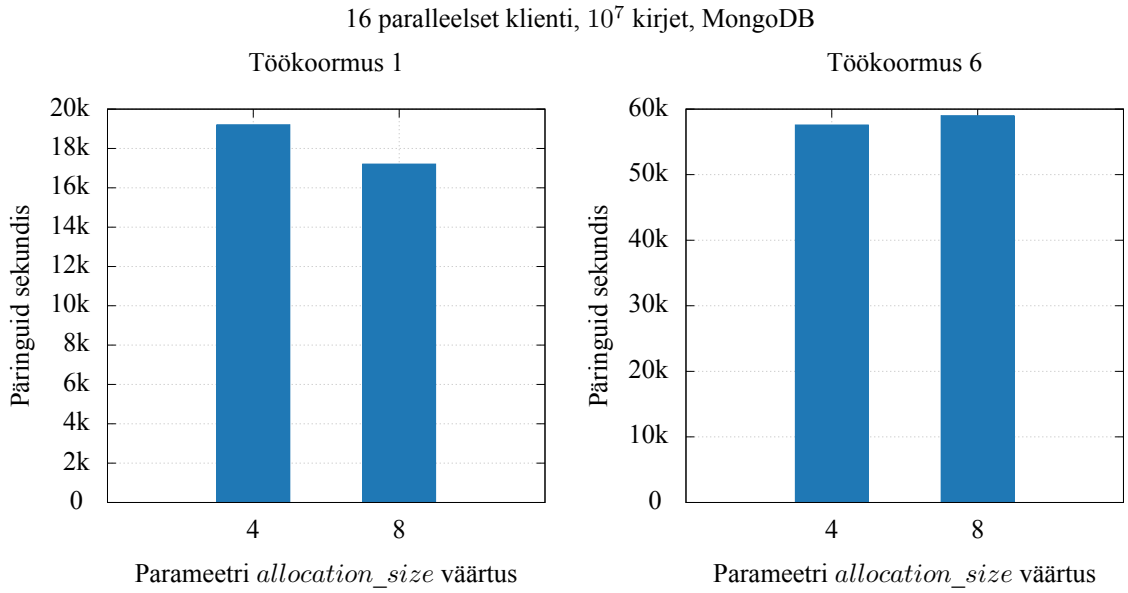
7.7 MongoDB

MongoDB WiredTiger-i talletusmootor kasutab vaikimisi 4KiB blokisuurust ning seda on võimalik tõsta. Kuigi on vähetõenäoline, et selle tõstmine jõudlust märgatavalt (kui üldse) parandab, tasub seda proovida.

Teine MongoDB eripära on residentsete talletusmootorite olemasolu, mis arendaja kohaselt peaks tagama prognoositava jõudluse.

7.7.1 *allocation_size=8*

Joonisel 23 on toodud WiredTiger-i talletusmootori *allocation_size* parameetri 8-le tõstmise mõju.



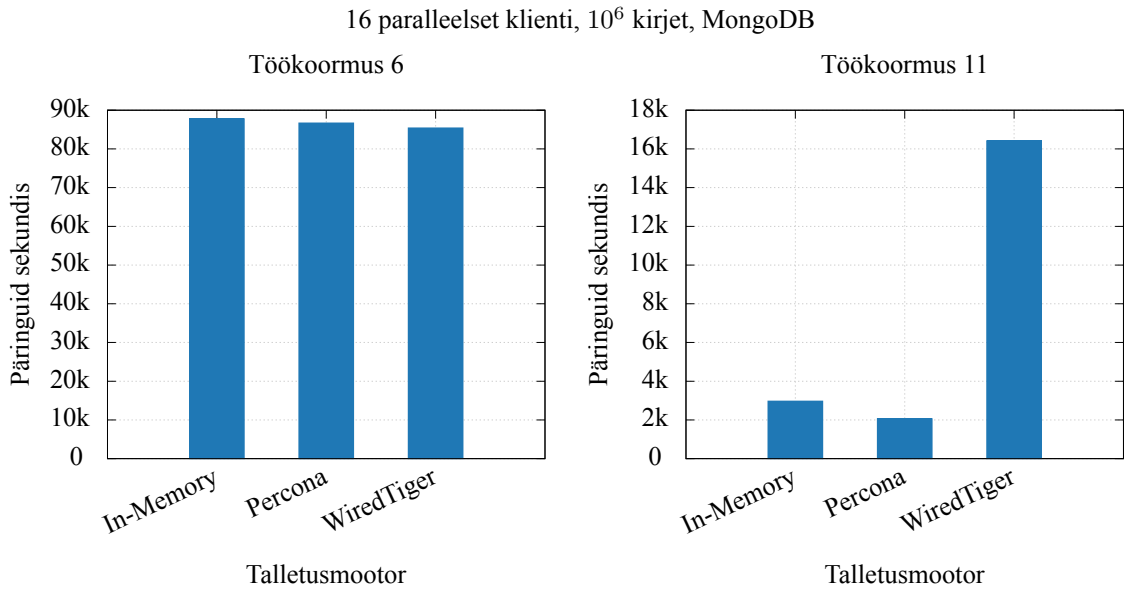
Joonis 23: *allocation_size=8* mõju MongoDB jõudlusele.

Näeme, et andmete sisestamisel on mõju negatiivne: jõudlus langes 10%. Andmete lugemisel mõju puudub.

7.7.2 Residentne

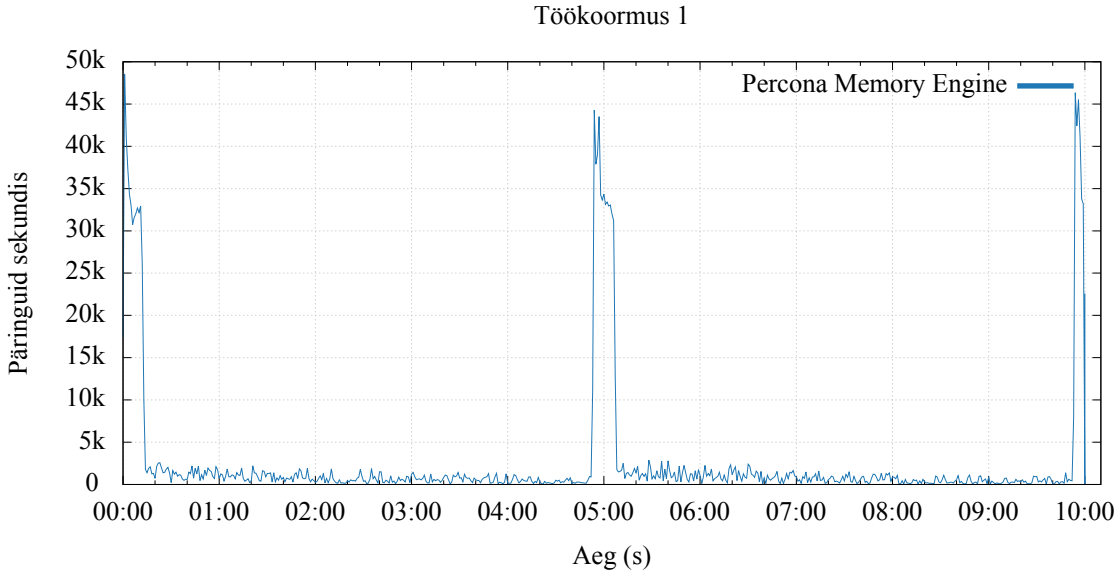
Joonisel 24 on toodud erinevate MongoDB talletusmootorite – MongoDB In-Memory, Percona Memory Engine ja WiredTiger – jõudluse võrdlus.

Näeme, et andmete lugemise jõudlus on kõigil sama, kuid töökoormuse 11 korral on Wi-



Joonis 24: Erinevate MongoDB talletusmootorite jõudlus, töökoormused 6 ja 11.

redTiger märkimisväärselt kiirem. Lähemal uurimisel selgub, et andmete sisestamisega töökoormuste korral avaldus anomaalia, mis on kujutatud joonisel 25.



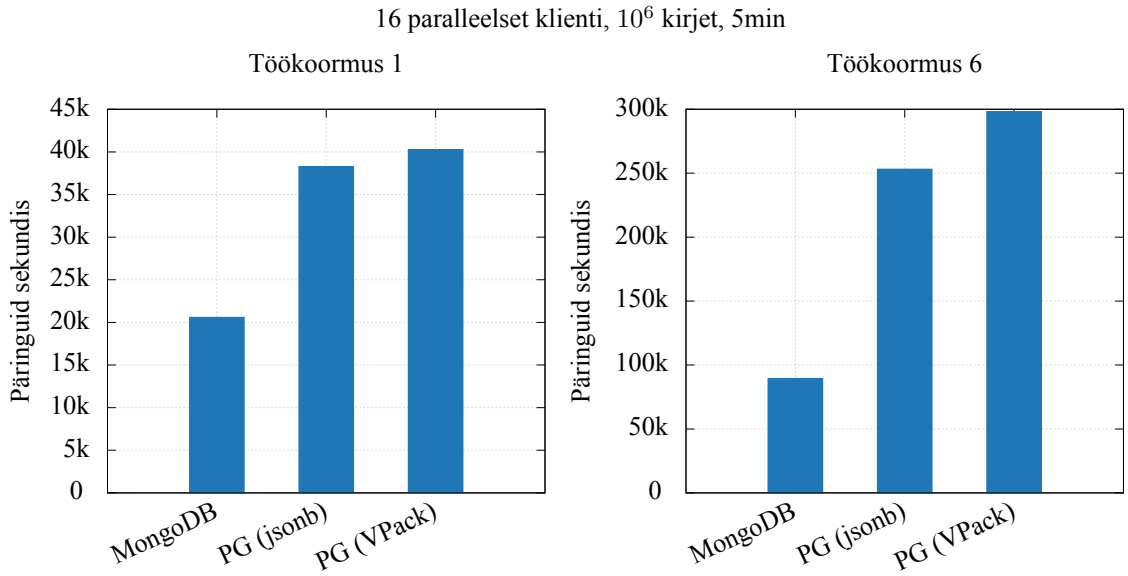
Joonis 25: MongoDB residentsete talletusmootorite anomaalia, töökoormus 1.

Näeme, et testi alguses on jõudlus umbes 45000 päringut sekundis, mis seejärel järsult langeb ning tõuseb uuesti mõnekümneks sekundiks täpselt viieminutilise intervalliga. Protsessorikasutus tõusis testi jooksul lineaarselt 92%-lt 619%-le, mälukasutuses olid järsud hüpped kõrge jõudluse ajal. Jääb selgusetuks, miks kumbki MongoDB residentne talletusmootor ei suuda mälu eraldada, kuigi seda on piisavalt saadaval ning WiredTiger-i kasutamisel pole see probleemiks.

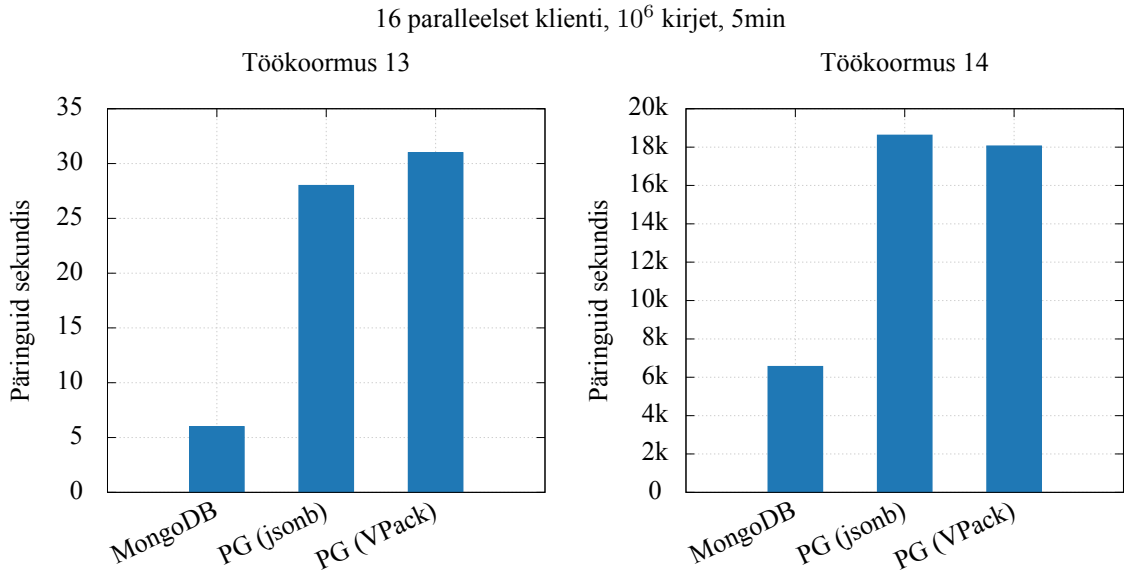
Latentsust on analüüsitud punktis 7.12.

7.8 Erinevate lahenduste võrdlus

Joonistel 26 ja 27 on toodud erinevate andmebaaside jõudluse võrdlus.



Joonis 26: Erinevate andmebaaside jõudlus, töökoormused 1 ja 6.



Joonis 27: Erinevate andmebaaside jõudlus, töökoormused 13 ja 14.

Näeme, et käesoleva töö realisatsioon on kiireim. *jsonb* on sellest andmete lugemisel 15% ning andmete sisestamisel 4% aeglasem. See tuleneb kahend- ja tekstipõhise formaadi

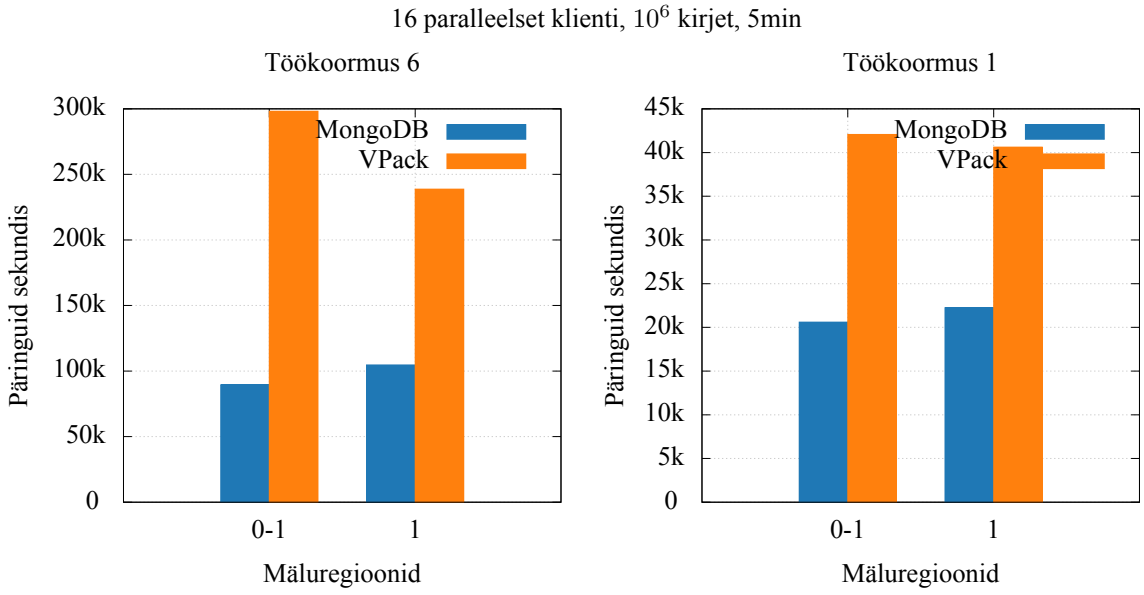
vahelise teisendamise vajadusest. Transaktsioonide jõudlus on mõlemal sama, kuid agregaatoperatsioonides on *jsonb* 9% aeglasem. See tuleneb tõenäoliselt asjaolust, et VPack formaadist on andmete otsimine kiirem.

MongoDB on andmete lugemisel 69% ning andmete sisestamisel 48% aeglasem, kui käesoleva töö realisatsioon, transaktsioonide jõudlus on 64% ning agregaatoperatsioonid tervelt 80% aeglasemad.

Seega on käesoleva töö tulemus edukas ning selle arendusega tasub edasi minna.

7.9 NUMA

Joonistel 28 ja 29 on toodud mälusõlmede valiku mõju jõudlusele. Testis lubati andmebaasil vaid mälusõlme 1 ning sellele lähimate protsessorilõimede (12) kasutamine.

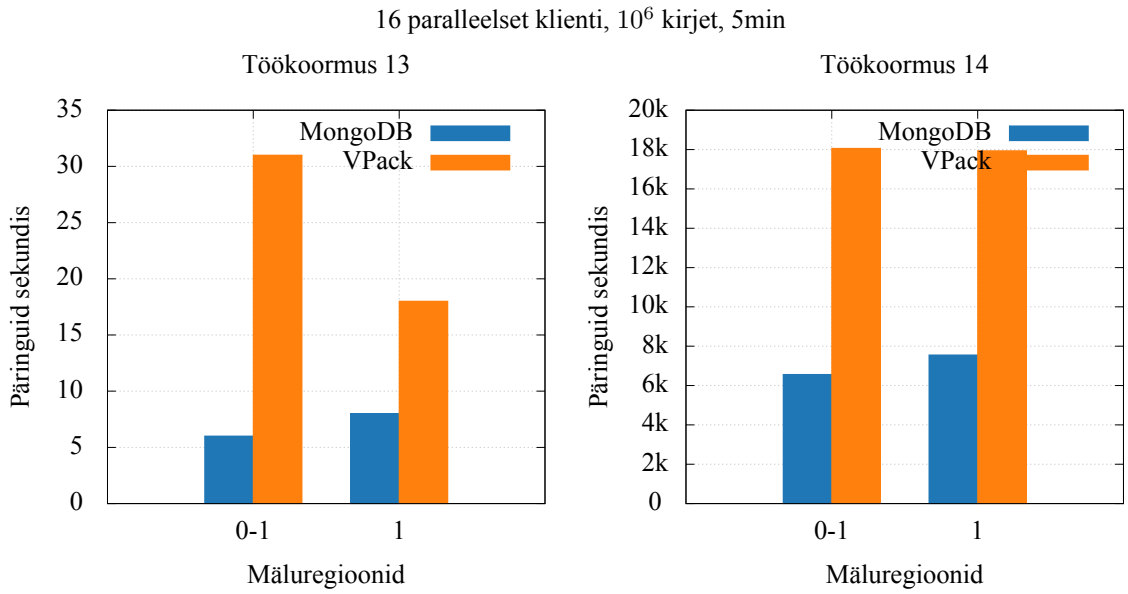


Joonis 28: Mälusõlmede mõju jõudlusele, töökoormused 1 ja 6.

Näeme, et MongoDB korral on protsessorile lähima mälusõlme kasutamine märkimisväärse positiivse mõjuga, samas kui PostgreSQL-i korral on mõju negatiivne.

PostgreSQL-i jõudluse langus on tõenäoliselt põhjustatud hoopis väiksemast saada olnud protsessorilõimede arvust, paraku ei sooritatud vastavaid teste.

MongoDB korral olid kõigi mälusõlmede kasutamisel agregaatoperatsioonid 25%, transaktsioonid 13%, andmete sisestamised 7% ja andmete võtmised 14% aeglasemad kui lähima mälusõlme kasutamisel.



Joonis 29: Mälusõlmede mõju jõudlusele, töökoormused 13 ja 14.

Seega on MongoDB puhul mõistlik võimalusel lubada vaid lokaalse mälusõlme kasutamine. PostgreSQL-i kohta tuleb test uuesti sooritada ning vaadelda ka protsessorilõimede arvu mõju.

7.10 Sünkroonse kirjutamise kiirus

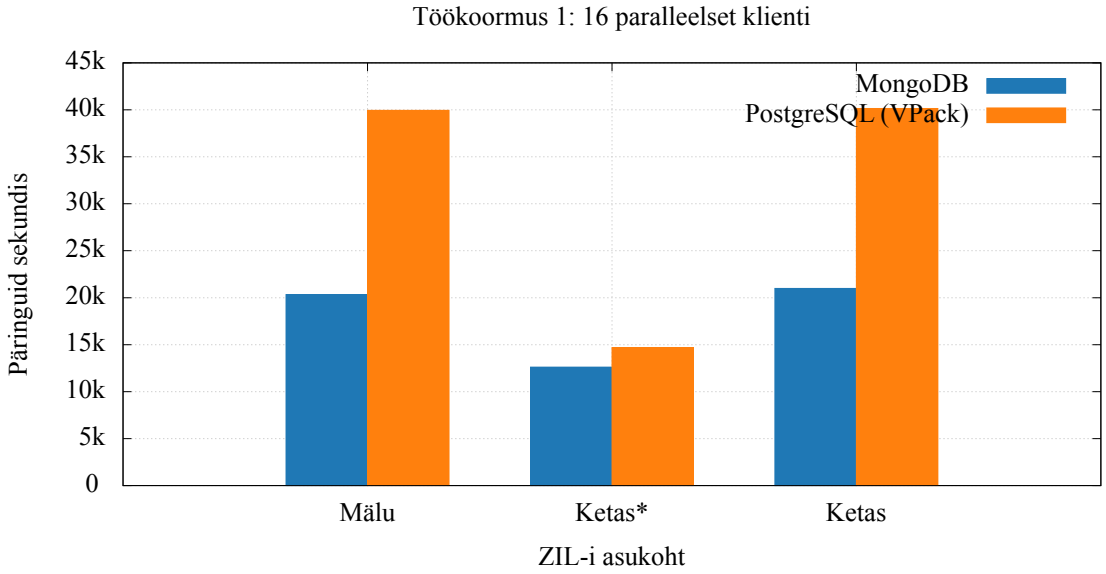
ZFS failisüsteemi kasutamisel oleneb sünkroonsete kirjutamiste kiirus ennekõike ZIL-i kiirusest, kuna mida kiiremini andmed sellele talletatakse, seda kiiremini saab kasutajale vastava kinnituse anda.

Joonisel 30 on toodud ZIL-i kiiruse mõju kirjutamise jõudlusele kui ZIL on mälukettal ja pooljuhtkettal erinevate *zfs_nocacheflush* parameetri väärtuste korral.

Näeme, et *zfs_nocacheflush=0* kasutamisel on andmete sisestamine PostgreSQL-i korral 63% ning MongoDB puhul 39% aeglasem. *zfs_nocacheflush=1* korral on kiirus sama, kui mälus asuva ZIL-i puhul, mis tähendab, et kirjutamise kiirust limiteeris ennekõike transaktsioonide kirjutamise jõudlus.

Seega on otstarbekas panna ZIL kettale, millel puhver puudub või mis kindlustab puhvris olevate andmete talletamise ka voolukatkestuste korral ning kasutada *zfs_nocacheflush=1*.

Muude ketaste puhul võib selle seade kasutamine tekitada andmekao, kuna ketas ei pruugi voolukatkestuse korral jõuda kõiki puhvris olevaid andmeid talletada. Sisuliselt on see



Joonis 30: ZIL-i kiiruse mõju jõudlusele, töökoormus 1. Tärniga märgitutel kasutati *zfs_nocacheflush=0*.

sarnane *sync=disabled* kasutamisele, mis lülitab välja sünkroonse kirjutamise ning umbes kirjutamise transaktsiooni ulatuses võib tekkida andmekadu. Mõnikord võib see arvestades saavutatavat võitu jõudluses olla aktsepteeritav, just nagu andmebaasi vähendatud turvalisusgarantiiga käitlemine.

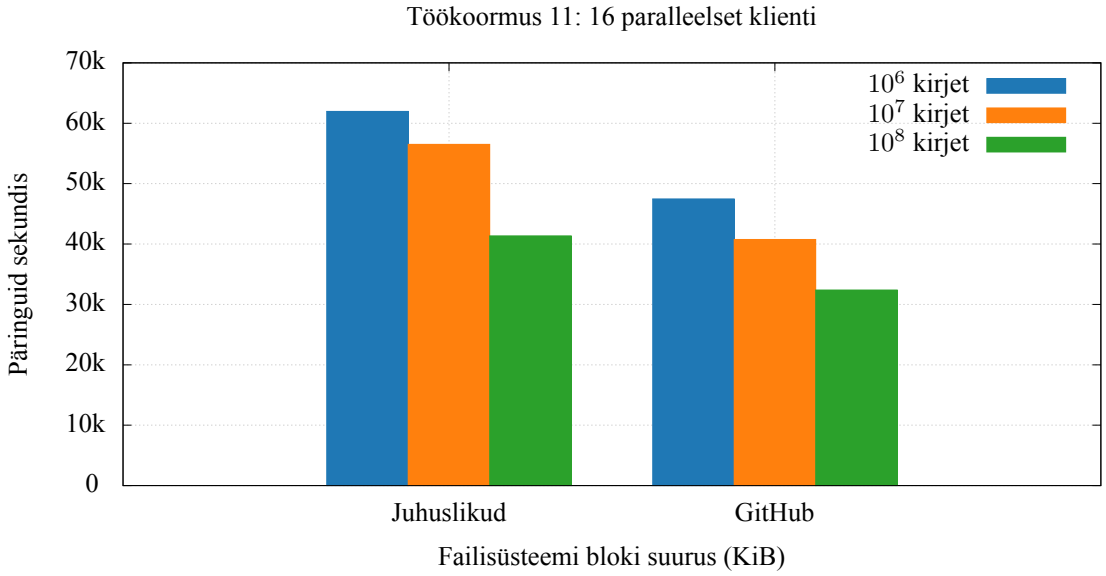
Tasub vaid tähele panna, et *sync=disabled* tekitab andmekao lisaks ka süsteemi tõrgete korral ning see võib jäädavalt rikkuda andmebaasi failid, mistõttu on parem andmebaasi käitlemine vähendatud turvalisusgarantiiga.

7.11 Kirjete suurus

GitHub-i kirjete 90-protsentiili suurus oli 6020 ja mediaan 1019 baiti ning kõige suurema kirje maht 393KB. Juhuslikult genereeritud kirjete 90-protsentiili suurus oli 958 ning mediaan 607 baiti.

Joonisel 31 on toodud PostgreSQL-i jõudluse sõltuvus kirjete suurusest.

Näeme, et töökoormus 10 oli GitHub-i kirjete puhul keskmiselt 25% aeglasem, kui juhuslike andmete korral, mis tuleneb sellest, et GitHub-i kirjed olid keskmiselt 30% suuremad.



Joonis 31: Kirjete suuruse mõju PostgreSQL-i jõudlusele, töökoormus 10.

7.12 Latentsus

Latentsuse ja päringute arvu iseloomustamiseks sobivad protsentiilid paremini kui keskmine. Näiteks joonisel 25 toodud juhul on keskmine päringute arv sekundis 3020, kuid 90-protsentiil vaid 126, mis tähendab, et 10% ajast teostati alla 126 päringu sekundis. Järgnevalt toodud tulemused on saavutatud 5-minutiliste testide jooksul, v.a. tärniga märgitud, mis on 1-minutilistest testidest.

Tabelis 3 on toodud erinevate andmebaaside/talletusmootorite latentsus andmete sisestamisel.

Tabel 3: Erinevate andmebaaside/talletusmootorite latentsus, töökoormus 1.

Andmebaas/Talletusmootor	**Latentsus (µs)**			**Päringuid sekundis**		
	keskm.	**99-p.**	**90-p.**	**keskm.**	**99-p.**	**90-p.**
WiredTiger	776	1580	1030	20590	17690	19200
VPack	379	808	504	42080	**38330**	39980
jsonb	417	859	536	38290	34940	37150
WiredTiger (NUMA)	717	1430	940	22250	19240	20960
WiredTiger (asünk.)*	309	686	438	50670	27170	47090
VPack (asünk.)*	**113**	**438**	**142**	**136160**	33980	**136740**

Näeme, et väikseima ja stabiilseima latentsuse (ning seega ka suurima päringute arvu sekundis) saavutab PostgreSQL (VPack andmetüübiga), seda nii sünkroonsel kui ka asünkroonsel kirjutamisel.

Tabelis 4 on toodud erinevate andmebaaside/talletusmootorite latentsus andmete võtmisel.

Tabel 4: Erinevate andmebaaside/talletusmootorite latentsus, töökoormus 6.

Andmebaas/Talletusmootor	**Latentsus (µs)**			**Päringuid sekundis**		
	keskm.	**99-p.**	**90-p.**	**keskm.**	**99-p.**	**90-p.**
WiredTiger	177	280	206	89490	81440	86170
Percona ME	183	284	210	86570	81770	83550
MongoDB In-Memory	181	281	214	88000	80660	83620
VPack	**53**	**70**	**59**	**298150**	**285580**	**292260**
jsonb	63	85	69	253180	246080	249680
WiredTiger (NUMA)	152	260	189	104680	92990	98990

Näeme, et väikseima ja stabiilseima latentsuse saavutab jällegi PostgreSQL VPack andmetüüpi kasutades. Seejuures on latentsus stabiilsem, kui MongoDB residentsete talletusmootorite korral.

Tabelis 5 on toodud erinevate andmebaaside/talletusmootorite latentsus töökoormusel 11.

Tabel 5: Erinevate andmebaaside/talletusmootorite latentsus, töökoormus 11.

Andmebaas/Talletusmootor	**Latentsus (µs)**			**Päringuid sekundis**		
	keskm.	**99-p.**	**90-p.**	**keskm.**	**99-p.**	**90-p.**
WiredTiger	679	1.26ms	1010	23470	12210	21840
VPack	247	688	425	64790	26450	61110
jsonb	257	722	422	62150	30660	58730
WiredTiger (NUMA)	624	1210	937	25630	14890	24070
WiredTiger (asünk.)*	537	1510	747	29580	18250	28600
VPack (asünk.)*	**95**	**259**	**130**	**163090**	**62660**	**155650**

Näeme, et väikseima ja stabiilseima latentsuse saavutab PostgreSQL VPack andmetüüpi kasutades, seda nii sünkroonse kui ka asünkroonse kirjutamise korral.

Tabelis 6 on toodud erinevate andmebaaside/talletusmootorite latentsus transaktsioonide korral.

Tabel 6: Erinevate andmebaaside/talletusmootorite latentsus, töökoormus 14.

Andmebaas/Talletusmootor	**Latentsus (µs)**			**Päringuid sekundis**		
	keskm.	**99-p.**	**90-p.**	**keskm.**	**99-p.**	**90-p.**
WiredTiger	2430	3660	3020	6570	5140	6080
VPack	836	3190	1020	18060	**16590**	17400
jsonb	858	2290	1060	18620	16440	17240
WiredTiger (NUMA)	2120	3060	2340	7550	6420	7000
WiredTiger (asünk.)*	2200	3470	2870	7230	5490	6670
VPack (asünk.)*	**504**	**1540**	**623**	**30450**	15380	**27950**

Näeme, et väikseima latentsuse ja suurima päringute arvu sekundis saavutab jällegi PostgreSQL VPack andmetüübiga. Stabiilseima latentsuse saavutab aga MongoDB lokaalsele mälusõlmele piiratud variant, mis oli ühtlasti märgatavalt stabiilsem MongoDB tavalisest variandist.

Seega tasub uurida PostgreSQL-i kasutusvõimalusi ka töökoormuste korral, kus on vajalik stabiilse latentsuse saavutamine.

8 Kokkuvõte

Käesoleva töö eesmärgiks oli võimaldada PostgreSQL-i kasutamine täisväärtusliku dokumendikoguna ning vaadelda erinevate süsteemiseadete mõju jõudlusele.

Töö esimeses osas anti ülevaade probleemi taustast, uuriti erinevaid andmeformaate ning nende sobivust dokumendikogus kasutamiseks. Samuti käsitleti andmete statistika olulisust, uuriti erinevaid võimalusi dokumendikogude jõudluse võrdlemiseks ning anti ülevaade erinevatest jõudlust mõjutavatest andmebaasi, failisüsteemi ja operatsioonisüsteemi seadetest.

Töö teises osas analüüsiti leitud andmeformaate ning valiti nende seast sobiv käesolevas töös realiseerimiseks. Sobivaid formaate oli täpselt üks – Amazon Ion – kuid tulenevalt selle C realisatsiooni puudulikust dokumentatsioonist otsustati ajakasutuse efektiivsust arvestades täiendada teist peaaegu sobivat formaati – VelocyPack-i. Lisaks käsitleti kuidas oli võimalik taaskasutada olemasolevat PostgreSQL-i statistikasüsteemi ning loodi reeglid dokumentide destruktureerimiseks, mis olid statistika genereerimise süsteemi aluseks. Peatüki kolmandas osas leiti, et ükski olemasolev testiraamistik ei sobi käesolevas töös kasutamiseks ning otsustati luua uus, mistõttu tuli lisaks määratleda nõuded testiraamistikule ning jõudlustestide läbiviimisele.

Töö kolmandas osas projekteeriti VelocyPack-ile vajalikud muudatused ning PostgreSQL-i laiendus VelocyPack-i andmetüübi, vastavate operaatorite, indekseerimise ja dokumentide statistika jaoks. Lisaks käsitleti testiraamistiku ehitust, kuidas testimisprotsessi maksimaalselt automatiseerida ning teste minimaalsete väliste mõjutustega läbi viia.

Töö neljandas osas kirjeldati teostatud PostgreSQL-i laienduse täpsemat ülesehitust ning kontrolliti selle korrektsust. Lisaks käsitleti täpsemalt testiraamistiku ülesehitust ning jõudlustestide automaatse läbiviimise süsteemi. Samuti valideeriti testiraamistiku tulemusi YCSB raamistiku abil ning leiti, et tulemus on rahuldav.

Töö viiendas osas analüüsiti jõudlustestide tulemusi ning leiti, et PostgreSQL on igas aspektis märkimisväärselt kiirem, kui seni populaarseim dokumendiandmebaas – MongoDB. Kuid jõudlustestide eesmärk ei olnud selle üldteada fakti kontrollimine, vaid eri-

nevate andmebaasi, operatsioonisüsteemi ja failisüsteemi seadete mõju uurimine ning suuniste andmine edasisteks testideks. Leiti, et:

- failisüsteemi blokisuurus tuleb hoolikalt valida vastavalt soovitud pakkimise suhtele ning ketta ja selle liidese läbilaskevõimele;
- PostgreSQL-i päeviku failisüsteemi blokisuurusena on soovitav kasutada 8 või 128KiB;
- vähendatud turvalisusgarantii kasutamine tõstab kirjutamise jõudlust üle kolme korra;
- liiga suured andmebaasi puhvrid toovad väikese tabamuskiiruse korral kaasa jõudluse märgatava languse;
- PostgreSQL-i jõudlus püsib stabiilne ka suure arvu lõimede korral;
- lõimede arvu tõstmine ei too kaasa märkimisväärset MongoDB jõudluse kasvu;
- *logbias=throughput* kasutamine andmebaasi köitel ei too kaasa muutusi jõudluses ega kettakasutuses;
- MongoDB residentsed talletusmootorid on kasutuskõlbmatud, kuna nendesse andmete lisamisel avaldub anomaalia, mis latentsuse väga kõrgeks tõstab;
- *jsonb* andmetüübi kasutamine on 4-15% aeglasem kui käesoleva töö realisatsiooni oma;
- PostgreSQL on kõigis testides märkimisväärselt kiirem kui MongoDB;
- MongoDB saab märgatavat kasu mälupöördumise optimiseerimisest;
- tasub kindlasti investeerida võimalikult kiiresse ja puhvrivabasse või voolukatkestuste kaitsega pooljuhtketasse ZFS Intent Log-i jaoks – jõudluse vahe on enam kui kaks korda;
- kirjete suurus mõjutab PostgreSQL-is märgatavalt nii andmete kirjutamise kui ka võtmise jõudlust;
- PostgreSQL saavutab testitud tarkvarast madalaima ja stabiilseima latentsuse.

Iga rakendus ja nende töökoormused on erinevad ning erineva komplekssuse astmega, seega on otstarbekas seadeid testida konkreetsete rakenduste ja koormuste jaoks. Käesoleva töö jõudlustestide tulemused pakuvad suunised, milliseid aspekte on mõtet testida ning kuidas baasstsenaariumi valida. Mitmete testitud aspektide puhul oli nende mõju juba eelnevalt teada, kuid puudus konkreetne hinnang.

Saavutati kõik püstitatud eesmärgid: PostgreSQL-is on võimalik talletada komplekksseid dokumente säilitades nende erinevate väärtuste algsed andmetüübid, päringuplaneerijal on statistika abil võimalik teha arukamaid valikuid päringuplaani valimisel ning lisaks on teostatud lahendus olemasolevatest parema jõudlusega.

9 Summary

The objective of this thesis was to enable using PostgreSQL as a full fledged document store and evaluate the effect of different system configurations to its performance.

In the first chapter an overview was given about the background of the problem, different data serialization formats were studied and their suitability for using in a document store evaluated. The importance of data statistics was explained, different ways to evaluate the performance of document stores were studied and an overview was given about different database, filesystem and operating system configuration parameters that affect performance.

In the second chapter the relevant data serialization formats were analyzed and one chosen for implementation. Only one suitable format was found – Amazon Ion – but due to the lack of documentation for its C implementation it was found to be more efficient to modify another one – VelocyPack. Additionally, it was studied how to reuse the existing PostgreSQL statistics system and rules for destructuring documents were created that are the basis of the statistics generation system. In the third part of the chapter it was found that none of the existing performance testing frameworks are suitable for the current use case and it was decided to create a new one, so the requirements for the testing framework and conducting the tests were specified.

In the third chapter the required changes to VelocyPack and the PostgreSQL extension for the new data type, relevant operators, indexing and document statistics were engineered along with the testing framework and tests automation system. Additionally, it was explained how to carry out the tests with minimal external interference.

In the fourth chapter the implementation of the PostgreSQL extension, testing framework and the tests automation system were described in more detail. Additionally, the results of the testing framework were validated by a comparison with those of the YCSB framework and the results found to be satisfying.

In the fifth chapter the results of the performance tests were evaluated and found that PostgreSQL is at all aspects significantly faster than the current most popular document

database – MongoDB. But the objective of the performance tests was not confirming that well-known fact, but evaluating the effect of different database, filesystem and operating system configurations to performance and give guidelines for further testing. It was found, that:

- the record size of the ZFS filesystem must be carefully selected according to the desired compression ratio and the throughput of the hard disk and the interface used;
- for PostgreSQL, the recommended record size for write-ahead log is 8KiB or 128KiB;
- using reduced data security (*synchronous_commit=off*) gives a write performance benefit of more than 3 times;
- too big *shared_buffers* cause a significant performance decline when their hit rate is low;
- the performance of PostgreSQL stays stable under high concurrency;
- raising concurrency does not bring any significant gain to MongoDB performance;
- using *logbias=throughput* for the database dataset does not bring any performance benefits nor changes in disk usage;
- the MongoDB in-memory storage engines are not usable due to an anomaly appearing when inserting data that causes high latency;
- using the *jsonb* format is 4-15% slower compared to the one implemented in this thesis;
- PostgreSQL is significantly faster than MongoDB in all tests;
- it does make sense to invest in a fast buffer-less or supercapacitor-backed SSD for storing the ZFS Intent Log (SLOG) – the difference in performance is more than two times;
- document size significantly affects the read and write performance of PostgreSQL;
- PostgreSQL achieves the lowest and most stable latency of all tested software.

Each application and their workloads are different and with differnet level of compexity, so the settings need to be evaluated for those particular workloads and applications. The results of the performance tests of this thesis give guidelines which aspects should be tested and how to choose the baseline scenario. The effect of many configuration options was already known, but the exact estimation was missing.

All objectives were achieved: PostgreSQL can store complex documents while keeping the original data types of its values, the query planner can make smarter decisions based on data statistics and the implementation has the best performance of the compared solutions.

Kasutatud kirjandus

[1] "A Scalability Comparison Study of Data Management Approaches for Smart Metering Systems". *2016 45th International Conference on Parallel Processing (ICPP)*.

[2] *Amazon Ion*. [WWW] http://amzn.github.io/ion-docs/ (2019-03-08).

[3] *Apache Arrow*. [WWW] https://arrow.apache.org (2019-03-08).

[4] ArangoDB. *NoSQL Performance Benchmark 2018 – MongoDB, PostgreSQL, OrientDB, Neo4j and ArangoDB*. [WWW] https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/ (2019-03-08).

[5] Peter A. Boncz, Marcin Zukowski ja Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". 2005.

[6] *CBOR*. [WWW] http://cbor.io/ (2019-03-08).

[7] CloudFlare. *Introducing ebpf_exporter*. 2018. [WWW] https://blog.cloudflare.com/introducing-ebpf_exporter/ (2019-03-08).

[8] Cybernetica AS. *Andmekaitse ja infoturbe leksikon*. [WWW] https://akit.cyber.ee (2019-03-08).

[9] *DB-Engines Ranking of Document Stores*. [WWW] https://db-engines.com/en/ranking/document+store (2019-03-08).

[10] Dominic Dwyer ja Wei Shan Ang. *MongoDB/PostgreSQL JSON Benchmark Tool*. 2017. [WWW] https://github.com/domodwyer/mpjbt (2019-03-08).

[11] EnterpriseDB. *Postgres Outperforms MongoDB and Ushers in New Developer Reality*. [WWW] https://www.enterprisedb.com/blog/postgres-outperforms-mongodb-and-ushers-in-new-developer-reality (2019-03-08).

[12] EnterpriseDB. *Zheap*. [WWW] https://github.com/EnterpriseDB/zheap (2019-03-08).

[13] Armando Fox et al. "Cluster-based Scalable Network Services". *SIGOPS Oper. Syst. Rev.* 31.5 (1997), lk.-d 78–91. [WWW] http://doi.acm.org/10.1145/269005.266662.

[14] *GH Archive*. [WWW] https://www.gharchive.org/ (2019-03-08).

[15] Jim Gray. "Readings in Database Systems". Toim. Michael Stonebraker. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988. Peatükk The Transaction Concept: Virtues and Limitations, lk.-d 140–150. [WWW] http://dl.acm.org/citation.cfm?id=48751.48761.

[16] *Jackson*. [WWW] https://github.com/FasterXML/jackson (2019-03-08).

[17] *Linuxi kerneli dokumentatsioon*. [WWW] https://www.kernel.org/doc/Documentation/sysctl/ (2019-03-08).

[18] Geoffrey Litt, Seth Thompson ja John Whittaker. "Improving performance of schemaless document storage in PostgreSQL using BSON" (2013).

[19] Kazuaki Maeda. "Performance evaluation of object serialization libraries in XML, JSON and binary formats" (2012).

[20] Dmitri Maksimov. "Performance Comparison of MongoDB and PostgreSQL with JSON types". Magistritöö. 2015.

[21] *MongoDB: Documentation*. [WWW] https://docs.mongodb.com/manual/ (2019-03-08).

[22] *OpenZFS dokumentatsioon*. [WWW] http://open-zfs.org/wiki/Main_Page (2019-03-08).

[23] Oracle. *Oracle Solaris Administration: ZFS File Systems*. 2012. [WWW] https://docs.oracle.com/cd/E23824_01/html/821-1448/index.html (2019-03-08).

[24] Oracle. *Sun Flash Accelerator F20 PCIe Card Product Notes*. 2013. [WWW] https://docs.oracle.com/cd/E19682-01/E21357/index.html (2019-03-08).

[25] Michel Ott. "Key/Value Pair versus hstore" (2011).

[26] Percona. *Percona MongoDB Exporter*. [WWW] https://github.com/percona/mongodb_exporter (2019-03-08).

[27] Percona. *Percona Server for MongoDB*. [WWW] https://www.percona.com/software/mongo-database/percona-server-for-mongodb (2019-03-08).

[28] *PGTune*. [WWW] https://pgtune.leopard.in.ua/ (2019-03-08).

[29] Mateusz Piech ja Robert Marcjan. "A new approach to storing dynamic data in relational databases using JSON". *Computer Science* 19 (2018), lk. 5.

[30] *PostgreSQL: Documentation*. [WWW] https://www.postgresql.org/docs/11/ (2019-03-08).

[31] *PostgreSQL mailing list*. [WWW] https://www.postgresql.org/message-id/flat/20180703070645.wchpu5muyto5n647@alap3.anarazel.de (2019-03-08).

[32] Postimees. *Audit: valimiste infosüsteemi tõrke põhjustas ASi Helmes viga*. 2011. [WWW] https://majandus24.postimees.ee/408091/audit-valimiste-infosusteemi-torke-pohjustas-asi-helmes-viga (2019-03-08).

[33] Postimees. *Eksperdid ei usu Helmese seletusi andmebaasi vea kohta*. 2011. [WWW] https://www.postimees.ee/400387/eksperdid-ei-usu-helmese-seletusi-andmebaasi-vea-kohta (2019-03-08).

[34] Prometheus. *Node exporter*. [WWW] https://github.com/prometheus/node_exporter (2019-03-08).

[35] *Prometheus*. [WWW] https://prometheus.io/ (2019-03-08).

[36] *ps*. [WWW] https://linux.die.net/man/1/ps (2019-03-08).

[37] Will Rouesnel. *PostgreSQL Server Exporter*. [WWW] https://github.com/wrouesnel/postgres_exporter (2019-03-08).

[38] Nico Rutishauser. "TPC-H applied to MongoDB: How a NoSQL database performs" (2012).

[39] Valeriia Shpychka. "Analysis of Performance and Complexity of Building a Web Application Based on Couchbase and PostgreSQL with jsonb Type". 2017.

[40] Luca Sinico. "Graph databases and their application to the Italian Business Register for efficient search of relationships among companies." Magistritöö. 2017.

[41] Timescale. *Timescale*. [WWW] https://timescale.com (2019-03-08).

[42] Timescale. *Why bother with NoSQL databases? Choose PostgreSQL for IoT.* [WWW] https://blog.timescale.com/choose-postgresql-for-iot-19688efc60ca/ (2019-03-08).

[43] Jan Vanura ja Pavel Kriz. "Perfomance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats". 2018, lk.-d 166–175.

[44] *VelocyPack*. [WWW] https://github.com/arangodb/velocypack/blob/master/README.md (2019-03-08).

[45] *Wired Tiger Reference Guide*. [WWW] http://source.wiredtiger.com/3.1.0/tune_page_size_and_comp.html (2019-03-08).

[46] "YCSB and TPC-H: Big Data and Decision Support Benchmarks". *2014 IEEE International Congress on Big Data*.

[47] *ZFS-i lähtekood*. 2019. [WWW] https://github.com/zfsonlinux/zfs/blob/a8577bdb32e091645df901d8501e44ef50748389/include/sys/zil.h (2019-03-08).

[48] *zfs*. [WWW] https://linux.die.net/man/8/zpool (2019-03-08).

Lisa 1 – Andmebaaside seadistused

```
max_connections = 300
shared_buffers = 10GB
effective_cache_size = 30GB
maintenance_work_mem = 2GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 100
random_page_cost = 1.1
effective_io_concurrency = 200
work_mem = 1456kB
min_wal_size = 1GB
max_wal_size = 2GB
max_worker_processes = 24
max_parallel_workers_per_gather = 12
max_parallel_workers = 24

autovacuum = off
full_page_writes = off
```

Joonis 32: PostgreSQL-i seaded: sünkroonne kirjutamine

```
max_connections = 300
shared_buffers = 10GB
effective_cache_size = 30GB
maintenance_work_mem = 2GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 100
random_page_cost = 1.1
effective_io_concurrency = 200
work_mem = 1456kB
min_wal_size = 1GB
max_wal_size = 2GB
max_worker_processes = 24
max_parallel_workers_per_gather = 12
max_parallel_workers = 24

autovacuum = off
full_page_writes = off

synchronous_commit = off
wal_writer_delay = 100
```

Joonis 33: PostgreSQL-i seaded: asünkroonne kirjutamine

```
storage:
  dbPath: /var/lib/mongodb
  journal:
    enabled: true
  directoryPerDB: true
  wiredTiger:
    engineConfig:
      cacheSizeGB: 20
      directoryForIndexes: true
      journalCompressor: none
    collectionConfig:
      blockCompressor: none
    indexConfig:
      prefixCompression: true

systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log

net:
  port: 27017
  bindIp: 0.0.0.0

processManagement:
  timeZoneInfo: /usr/share/zoneinfo

setParameter:
  wiredTigerConcurrentReadTransactions: 256
  wiredTigerConcurrentWriteTransactions: 256
```

Joonis 34: MongoDB seaded: sünkroonne kirjutamine

```
storage:
  dbPath: /var/lib/mongodb
  journal:
    enabled: true
    commitIntervalMs: 300
  directoryPerDB: true
  wiredTiger:
    engineConfig:
      cacheSizeGB: 20
      directoryForIndexes: true
      journalCompressor: none
    collectionConfig:
      blockCompressor: none
    indexConfig:
      prefixCompression: true

systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log

net:
  port: 27017
  bindIp: 0.0.0.0

processManagement:
  timeZoneInfo: /usr/share/zoneinfo

setParameter:
  wiredTigerConcurrentReadTransactions: 256
  wiredTigerConcurrentWriteTransactions: 256
```

Joonis 35: MongoDB seaded: asünkroonne kirjutamine