LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

Bezservera iebūvētas datubāzes kā alternatīva SQL

MAGISTRA DARBS

Autors: **Bogdans Ozerkins**
Studenta apliecības Nr.: bo12001
Darba vadītāja: prof., Dr.dat. Laila Niedrīte

RĪGA 2018

ANOTĀCIJA

Darba tēma – Bezservera iebūvētas datubāzes kā alternatīva SQL

Darbu veido 88 lappuses. Izmantotās literatūras sarakstu veido 21 avoti.

Pētījuma jautājumi:

- Relāciju datubāzes arhitektūru un tehnoloģiju pārskats, iespējamas problēmsituācijas, to risinājumi un alternatīvas datu pārvaldības pieejas
- Datu patvaldības metodoloģiju apkopojums un iespējama risinājuma īstenojums

Pētījuma mērķis – Izpētīt SQL valodas īpašības, iedomāto funkcionalitāti un īstenošanas variantus un noskaidrot, vai ir iespējams definēt datu pārvaldības pamata metodes un operācijas un vienkāršot tās no konceptuālā un tehniskā viedokļa, lai izveidotu elastīgākus datu pārvaldības principus un metodes.

Šī mērķa sasniegšanai ir nepieciešams:

- Izpētīt SQL valodas pamata principus un īstenošanas variantus
- Definēt ar SQL saistītas datu pārvaldības operācijas, saistītas datu struktūras un koncepcijas
- Definēt un apkopot datu pārvaldības metodes
- Piedāvāt vienkāršas datu pārvaldības īstenošanas piemēru, kas demonstrētu iespējamos ieguvumus lietotnes ietvaros

Darbu veido ievads, četras nodaļas, rezultātu apkopojums, secinājumi un izmantotās literatūras saraksts.

Zinātniski pētnieciskais darbs “Bezservera iebūvētas datubāzes kā alternatīva SQL” varētu tikt izmantots datu pārvaldības procesu plānošanai un relāciju datubāžu tehnoloģiju izpratnei, jo skar vairākus datu pārvaldības jautājumus un mēģina izsecināt alternatīvus risinājumus. Darba rezultātā tika izpētītas vairāki datu pārvaldības risinājumi, no tehniskas un arhitektūras puses, apkopoti datu pārvaldības paņēmieni, to pielietošanas priekšrocības un trūkumi, uz tiem balstoties izveidota datu pārvaldības bibliotēka un lietotne, kura bibliotēku pielieto. Secinājumus veido strukturētas autora izpratnes par pētnieciska darba tematu un iespējamus tālāku pētījumi virzienu aprakstu.

Atslēgvārdi: datu pārvaldība, relāciju datubāzes, SQL, bezservera iebūvētas datubāzes, modulāra pieeja

ABSTRACT

In this study “Serverless embeded databases as an alternative to SQL” the author presented a 88 pages paper, with 21 literature sources.

The questions discussed in this paper are:

- Overview of relational databases architecture and technologies, possible problematic situations, solutions to those and alternative data management approaches
- Summary of data management methodologies and implementation example

The aim of the research is to study SQL qualities, intended functionality and implementation variations, and find out if it is possible to define basic data management methods and operations, additionally simplifying those from conceptual and technical points of view, with the solo purpose of creating a flexible approach for data management.

To achieve the aim the following tasks has been put:

- Research the main principles and implementation variations of the SQL
- Define basic, in SQL defined, data management operations, related data structures and concepts
- Define and generalize data management methods
- Create a simplistic data management implementation example, which would demonstrate possible benefits of proposed approach to data management

The study consists of introduction, four chapters, results, conclusions and literature sources. The paper “” can be used in data management processes planning and understanding of relational database technologies, because it touches multiple data management aspects and tries to define possible approach alternatives. The paper is the result of research of multiple data management solutions, from both technical and architectural views, summing data management methods, possible benefits and problems concerning those. The resulting practical solution is and implementation of data management methodologies overview, which is a data management library and an example application using the library. Conclusion chapter describes author’s acquired knowledge and possible directions for future research.

Keywords: data management, relational databases, SQL, serverless embeded database, modular approach

AUTOREFERĀTS

Darba pamatā ir vēsturiskas relāciju datubāžu attīstības un pašreizējo relāciju datubāžu datu pārvaldības pieeju izpēte un dekonstrukcija, un alternatīvu datu pārvaldības pieeju meklējumi. Darba novitāti definē pieeja. Pašlaik datu pārvaldības diskusijas centrā ir relāciju datubāzes un alternatīvas NoSQL datubāzes, šīs darbs piedāvā alternatīvu diskusijas pamata definējumu, kas ir datu pārvaldības metodes un pieejas, kuras definē datubāzes funkcionalitāti un dažreiz arī piekļuves interfeisu, tas ir pielietošanas sfēru. Tas koncentrē uzmanību nevis uz datu pārvaldības efektīvas pielietošanas gadījumiem, bet uz datu pārvaldības pielietošanas iespēju paplašinājumu.

Darbā tika izmantoti vairāki resursu tipi: grāmatas un raksti par datu pārvaldības īstenojumiem, datu pārvaldības sistēmu īstenojumi, pasaules datu pārvaldības standarti un datu pārvaldības sistēmu atvērti pieejams programmas kods. Grāmatas un raksti veido pusi no kopējā literatūras avotu skaita, un veido pētījuma pamatu, aprakstot vēsturiskās tendences un pašreizējas arhitektūras un pieejas. Dokumentācijas un programmatūras kods tika izpētīts lai iedziļinātos datu pārvaldības īstenojumā, noskaidrot saskaņotību ar standartiem, precīzāk definēt datu pārvaldības metodes.

Datu pārvaldība ir plašs temats, jo tiek skarts gandrīz katrās lietotnes izstrādes un apraksta laikā. Darba ietvaros šīs temats tika maksimāli precīzi ierobežots, lai skartu tikai aktuālas metodes relāciju datubāzes ietvaros. Sakarā ar to, ka darbs ietver gan konceptuālus spriedumus, gan īstenošanas nianses, tika pievērsta īpaša uzmanība informācijas detalizācijas kontrolei. Vēsturiskas situācijas apraksts un arhitektūras pētījumi tika izklāsti pietiekami detalizēti, bet bez tehniskām detaļām, bet īstenojuma nianses tika papildinātas ar tehniskām detaļām un piemēriem.

Pētījuma laikā tika apgūtas iemaņas par datubāzes arhitektūras variācijām un datu pārvaldības īstenojuma detaļām, bez kā praktiskais darbs nebūtu iespējams īstenot. Datu pārvaldības bibliotēka un uz tā balstītā lietotne tika īstenota vairāku mēnešu laikā, jo prasīja papildus tehniskas un matemātiskas zināšanas izstrādei. Tomēr tehniski par praktiskām darbam veltīto laiku var uzskatīt pētījuma darba izstrādes laiku, jo vairākas iemaņas ir cieši saistītas un starp atkarīgas.

Darbā ir rezultāti, kur katrs sniedz inovatīvu skatu uz datu pārvaldības konceptu. Darbs apkopo datu pārvaldības metodes relāciju datubāžu arhitektūras un tehnoloģiju ietvaros, papildus parakstot metožu starp saiknes, atkarības un īstenojuma nianses. Šīs pārskats ļauj iegūt priekšstatu datu pārvaldībās jomā un strukturē datu pārvaldības pamata pieejas. Darbs satur datu pārvaldības

īstenojumu augstā programmēšanas valodā un uz tā balstīto lietotnes. Šis ir piemērs datu pārvaldības alternatīvai pieejai, kura piedāvā gan vairāku pieeju izmantošanas, gan paplašināšanas iespējas. Alternatīvas literatūras meklējumos netika atrasti līdzīgi pētījumi, kuri piedāvātu līdzīgu skatu datu pārvaldības jomā.

Darbs tika izstrādāts patstāvīgi, atbilst kvalitātes, struktūras un noformējuma prasībām, kas veikti pēc atbilstošām Universitātes norādījumiem.

SATURA RADĪTĀJS

APZĪMĒJUMU SARAKSTS

Atzvanīšanas funkcija – funkcija, kura tiek izsaukta pēc iepriekš definēta notikuma

SQL (Structured Query Language) – pieprasījumu instrukciju valoda relāciju modeļa īstenojumam, tiek atbalstīta vairākās datubāzes sistēmās

ISO (International Organization for Standardization – organizācija, kura izveido un atbalsta vairākus internacionālus standartus, iekļaujot SQL ISO standartus

MySQL – atvērtā koda relāciju datubāze, kura tiek plaši izmantota pasaulē

Mutex bloķēšanu – datnes vai atmiņas segmenta bloķēšanas pieeja

Datnes slēgšana – paņēmiens, kas ļauj kontrolēt procesu piekļuves pie datnes iespējamību

Vienlaicīgie procesi – procesi, kuri vienas sistēmas ietvaros eksistē vienlaicīgi un spēj konkurēt resursu piekļuvē

Metadati – informācija par datiem, parasti apraksta datu struktūras, tipus vai iespējamas operācijas

IEVADS

Mūsdienās tehnoloģijas attīstās ātri, arvien parādās jaunas paradigmas, programmēšanas valodas, satvari, datubāzes, operētājsistēmas un ar informācijas tehnoloģijām saistītas profesijas. Tomēr programmatūras jauninājumi un izmaiņas joprojām tiek aktualizēti, to nosaka programmētāju sabiedrības viedoklis un uzņēmumu vajadzības. Šī darba ietvaros uzmanība tiks pievērsta datu pārvaldības un datubāzes attīstības tendencēm, mēģinot veidot pamatojumu pašlaik populāriem risinājumiem un prognozēt nākotnes tendences.

Sākot no 2011. gada Stack Overflow interneta vietne organizē ikgadējas aptaujas par izstrādātāju demogrāfisko sastāvu, izmantotajām un vēlamajām tehnoloģijām, programmēšanas valodām utml. Tas tiek darīts, lai piedāvātu izstrādātajam izvērtētu un pamatotu informāciju atbilstošā jomā, kas atbilst profesionālas sabiedrības tendencēm un industrijai kopumā. Informācijas sistematizācijas un apkopošanas mērķis ir arī izglītot uzņēmējus, informējot tos par izstrādātāju profesionālajām prasmēm un tehnoloģiju attīstības virzieniem. Dati ir pieejami lejupielādei pēc ODbL (Open Database License). 2017. gadā aptaujā iesaistījās 64 tūkstoši izstrādātāju no visas pasaules [1].

Fokusējoties uz datubāzēm, MySQL (55.6% cilvēku pielieto šo risinājumu) ir populārākais risinājums pēc šīs aptaujas sniegtajiem datiem, tam seko SQL Server (38.6%), SQLite (26.6%), PostgreSQL (26.5%) risinājumi. Tie visi ir relāciju datubāzes serveri, izņemot SQLite, kurš ir bezservera relāciju datubāze un dažreiz tiek klasificēts kā iebūvēts serveris. Nākamās sarakstā ir MongoDB (21.0%), Oracle (16.5%), Redis (14.1%) un Cassandra (3.1%) datubāzes. Abas MongoDB un Redis datubāzes ir NoSQL risinājumi, kuri tika izveidoti samērā nesen un jau ir sasnieguši lielu pielietojamību. Šīs aptaujas ietvaros ir vēl viena sadaļa, kas ir saistīta ar datubāzēm - iecienītākās datubāzes un neērtākās datubāzes. Interesanti, ka pirmās iecienītāko datubāžu sarakstā ir Redist, PostgreSQL un MongoDB, par neērtākajām ir nosauktas Oracle, MySQL un SQLite [1].

NoSQL datubāzes (precīzāk būtu - ne relāciju datubāzes) fokusējas uz citu, salīdzinot ar RDBMS, risinājumu problemātiku. Ir vairākas NoSQL datubāžu struktūras (Key-Value, Document, Graph, RDF), katrā tiek optimizēta konkrētas problēmas vai problēmu grupas risināšanā. NoSQL datu struktūras var daļēji (vai pilnība) īstenot, izmantojot klasiskus SQL un relāciju modeļa rīkus, piemēram, vienkāršs Key-Value modulis, kas varētu tikt izveidots uzbūvējot Relāciju tabulu ar vienu PRIMARY KEY kolonnu un vienu “value” kolonnu. NoSQL ar īstenotu

Key-Value krātuvi, satur optimizācijas ātrdarbības iespēju, kā arī tieši šai struktūrai pielāgotu lietošanas interfeisu. Specifisks interfeiss ļauj programmētājiem jau izstrādes sākumposmā izmantot funkcionalitāti attiecīgi esošai datu struktūras paradigmai, nevis jaukt klasiskas programmēšanas koncepcijas ar Relāciju modeli [2].

Relācijas datubāzēs daļēji vai pilnīgi integrē NoSQL funkcionalitāti, paplašinot relācijas modeli ar papildus datu struktūrām un Datubāzes SQL dialektu ar papildus konstrukcijām. Programmētājam lietotnes izstrādēs laikā ir pieejamas vairākas datu struktūras vairāku algoritmu atbalstam, līdz ar to datubāzes mēdz īstenot vairākus SQL paplašinājumus, lai sniegtu atbalstu biežāk sastopamajiem gadījumiem. Viens no spilgtākajiem piemēriem ir XML un JSON formātu atbalsts RDBMS, kur katra datubāze ievieš savas integrācijas nianses, optimizācijas papildinājumus utml. Tas rada relāciju datubāzes modeļa paplašinājumu, kas katrai datubāzei tas ir savs. Piemēram, MySQL atbalsta darbu ar JSON kolonnas formātu, ievieš vairākas SQL funkcijas darbam ar JSON tipu, struktūras nolasei un apstrādei, paplašinot SQL dialektu. JSON dokuments ir SQL dialekta daļa, līdz ar to Relāciju modeļa daļa, tas nozīmē, ka viena datu struktūra jeb datu koncepcija tiek iebūvēta otrā, kas padara sarežģītākus gan konceptuālus, gan tehniskus darbus [3].

Maģistra darba problēmas definīcija ir - relāciju datubāzes nav pietiekami elastīgas komplekso datu pārvaldības procesu īstenošanai, jo tiek optimizētas entītiju relāciju modeļa atbalstam un netiek optimizētas citu modeļu atbalstam. Sarežģītākas datu struktūras parasti tiek pielāgotas vienotam pārvaldības interfeisam (SQL), līdz ar to tiek pielīdzinātas entītiju relāciju modelim un modeļa reprezentācijai, kas savukārt padara datu struktūras pārvaldību sarežģītāku un dziļāk slēpj tehniskās īstenošanas nianses. No tā var secināt, ka trūkst vienotas pieejas vairāku datu struktūru un modeļu pārvaldībai, pieejas, kura varētu piedāvāt elastīgu datu pārvaldību un datu pārvaldības elementu struktūru.

Maģistra darba mērķis ir izpētīt SQL valodas īpašības, SQL standartos ieteikto funkcionalitāti un īstenošanas variantus un noskaidrot, vai ir iespējams definēt datu pārvaldības pamata metodes un operācijas un vienkāršot tās no konceptuāla un tehniska viedokļa, lai izveidotu elastīgākus datu pārvaldības principus un metodes.

Lai sasniegtu mērķi, tika definēti šādi uzdevumi:

- Izpētīt SQL valodas pamata principus un īstenošanas variantus
- Izpētīt ar SQL saistītas datu pārvaldības operācijas, saistītas datu struktūras un koncepcijas

- Izpētīt un apkopot datu pārvaldības metodes
- Piedāvāt vienkāršas datu pārvaldības īstenošanas piemēru, kas demonstrētu iespējamos ieguvumus lietotnes ietvaros

Pētījuma struktūra: Pirmajā nodaļā tiek apskatīti SQL valodas un Relāciju datubāzes pirmsākumi, ka arī aprakstīti Relāciju datubāzes pamata principi, konceptuālais modelis un datu pārvaldības variācijas. Otrajā nodaļā tiek aplūkotas datubāzes arhitektūras variācijas, SQL standartizētas un pielāgotas ne-relāciju modeļa iespējas un izveidošanas konceptuālie paņēmieni un problēmas. Trešajā nodaļā tiek apkopota informācija, kas ir saistīta ar datu pārvaldības metodēm un piedāvāti tās vienkāršas īstenošanas varianti un variantu priekšrocības. Ceturtajā nodaļā tiek aprakstīts datu pārvaldības bibliotēkas un lietotnes piemērs, ar to saistītās problēmsituācijas un to iespējamie risinājumi. Tiek prezentēti pētījuma darba rezultāti, pētījuma darba secinājumi un iespējamie tālākie pētījuma virzieni.

1. SQL INFORMĀCIJAS PĀRVALDĪBAS APSKATS

Šajā nodaļā tiks apskatīti datu un datubāzes jēdzieni, kā arī tiks izsekots datubāzes klasifikācijai un to atslēgu funkcionalitāte, kuras definē datubāzes piederību. Tiks apskatīts tas, kā datubāzes evolucionējušas un kāds bijis to izmaiņu pamatojums, kā arī izpētīti SQL pamata koncepcijas, operācijas un esošos standartus, papildus apskatot to standartu datu pārvaldības tendences. Papildus tiks analizēts un dekonstruēts MySQL datubāzes vairāku operāciju īstenojums.

Nodaļas mērķis ir radīt priekšstatu par SQL iesaistītām koncepcijām un to īstenojuma variācijām, izpētīt datubāzes funkcionāla vienkāršojuma iespējas un to iespējamo pamatojumu.

1.1. Dati un informācijas glabātuves pamatojums

Pirms sākt spriest par datu pārvaldību ir svarīgi izprast, kas ir dati un kādā veidā dati tiek iekļauti informācijas sistēmās.

Dati ir reprezentācija faktam, notikumam vai idejai, tie var būt cipars, vārds (vai vairāki vārdi), attēls, utml. Dati nosaka kvalitatīvu vai kvantitatīvu atribūtu (vai atribūtu kopu) mainīgajam vai mainīgo kopai. Tomēr dati, kas tiek definēti bez konteksta, neietver sevī nozīmes, jo tie ir tikai simbolu virknes. Piemēram, cipari 5 un 10 varētu nozīmēt jebko - datumu, vecuma grupas vai ciparu sekvences definīciju. Tomēr tāds datu definējums kā 5 zēni un 10 meitenes nosaka situāciju, kuru varētu saistīt un klasificēt kā noteiktu cilvēku grupu apzīmējumu. Šis konteksts tiek definēts kā meta dati, kuri pēc savas būtības ir dati par datiem, kas palīdz definēt aprakstīto un saglabāt datos situāciju [4].

Datus, kuri ir iesaistīti kontekstā, var izmantot vairākiem nolūkiem, tomēr tas ir iespējams tikai tad, ja dati ir organizēti, saglabāti un ir pieejami pēc pieprasījuma. Par datubāzi parasti sauc datu kolekciju/ kopumu. Datubāze parasti satur gan datus, gan meta datus, un tie ir integrēti datubāzes struktūrā, kā arī parasti piesaistīti pie fiziskiem vai konceptuāliem objektiem. Datubāze satur savas struktūras meta datus, kuri apraksta datu ierakstus un attiecības starp tiem [4].

Darbu ar datiem definē cilvēks, līdz ar to visas izveidotās struktūras ir iespējams mainīt vai adaptēt konkrētu procesu izveidošanai, adaptācijas vai optimizācijai. Protams, sistēmai, kura piedāvā datu pārvaldību augstākā līmenī, ir nepieciešams atbalstīt informācijas rediģēšanas iespējas, no kā ir atkarīgas datubāzes pārvaldnieka iespējas pielāgot datus saviem mērķiem.

1.2. Datubāzes koncepciju un parādību attīstība

Datubāzes koncepcijas vēsturiskā attīstība ir svarīga datu pārvaldības procesu izprašanai. Datubāzes attīstījās iteratīvi, ar katru iterāciju uzlabojot jau esošus modeļus un pieejas.

Pirmās datubāzes parādījās ap divdesmitā gadsimta sešdesmitajiem gadiem, kad datori un cietie diski kļuva populāri uzņēmumos, tie tika izmantoti ar mērķi palielināt glabātuves kapacitāti. Cietie diski deva piekļuvi datiem, kas atļāva lasīt tikai nepieciešamos ierakstus, kontrolēt darbu ar datiem un bija alternatīva tolaik populārajām uz lentām bāzētajām glabātuvēm, kuras piedāvāja datu apstrādi partijās. Pirmie modeļi, kas konceptualizēja darbu ar datiem datubāzē bija tīmekļa modelis (network model) CODASYL (Conference on Data System Language) un hierarhisks modelis (hierarchical model) IMS (Information Management System). Šīs sistēmas tika projektētas, lai efektīvi apstrādātu datus, nevis aprakstītu datu struktūru un atvieglotu darbu ar to. Lietotājiem bija jāzina datu struktūras un piekļuves metodoloģijas, lai strādātu ar šiem modeļiem [4].

E.F. Codd izstrādāja relācijas datubāzes modeļus divdesmitā gadsimta septiņdesmitajos gados. Modelis aprakstījās datu organizācijas terminoloģiju un paņēmienus, kas atļāva strādāt ar datiem konceptuālā, nevis tehniskā līmenī. Šo modeli 1976. gadā uzlaboja P. Chen un nosauca to par Entity-Relationship (ER) modeli. Parādījās pirmās relāciju datubāzes - INGRES un System R, kuras definēja funkcionalitāti un paņēmienus relāciju datu glabātuves un apstrādes realizācijai, kā arī iesāka SQL valodas attīstību [4,5].

Relāciju datubāzes kļuva populāras un tika plaši izmantotas. Parādījās vairāki komerciāli produkti, kuri popularizēja ER modeli, piemēram, DB2, Dbase, Oracle, utml. Nākamais solis datubāzes koncepta evolūcijā bija objektu orientētas datubāzes, kas tika izveidotas astoņdesmitajos gados, līdz ar objektu orientētu programmēšanu. To mērķis bija piedāvāt iespēju strādāt ar datiem, kuri ir strukturēti pēc Entity-Relationship modeļa un objektu orientēti, t.i., definēt tabulas kā objektu kopas, tabulu rindas kā objektus un tabulu kolonnas kā objektu atribūtus. Tādas datubāzes ir projektētas, lai labi strādātu ar tādam valodām kā, piemēram, C++, Objective-C, Delfi, Java [4].

Konceptuāli darbs ar datubāzēm attīstījās un datubāzes (un lietotnes) tika diferencētas pēc Online Transaction Processing (OLTP) un Online Analytical Processing (OLAP) metodoloģijas. OLTP orientētas lietotnes realizēja klasisko darbu ar datiem, SQL valodu, relāciju koncepcijas un ACID principus. OLAP tika izmantots datu krātuves veidošanai un padziļinātai datu izpētei. Līdz ar OLAP parādījās tādas koncepcijas kā Zvaigznes shēma un Sniegpārslas shēma, fakti un

dimensijas, kas atļāva konceptualizēt un optimāli projektēt datu krātuves. No OLTP lietotnēm tika popularizēta object-relational mapping (ORM) pieeja datu strukturēšanai. ORM atļāva integrēt objektu orientētas lietotnes ar relāciju datubāzēm, kas ir līdzīgi Object Oriented Databases (OOD), bet tas tika realizēts lietotnes līmenī. ORM ir programmatūras slānis, kas tiek īstenots starp lietojumprogrammatūras un datubāzes draiveri, papildus datu definīcijai un validācijai [4,6].

Datu apjomiem palielināties, parādījās vairākas NoSQL datubāzes, kuras neatbalsta SQL interfeisu, neatbalsta vai daļēji atbalsta relāciju starp entītijām, neatbalsta vai daļēji atbalsta ACID (Atomicity, Consistency, Isolation, Durability), un tika optimizēti konkrēti tehniskā vai konceptuālā līmeņa modeļi. NoSQL datubāzes tipi ir - key-value storage, BigTable koncepta realizācija, dokumentu glabātuve, grafu datubāzes [4].

Protams, relāciju modelis ir "de fakto" standarts darbam ar datiem, līdz ar to relāciju datubāzes ir primārā izvēle izstrādātas lietotnes datu pārvaldības nolūkiem. Praktiski tas nozīmē, ka jaunas lietotnes mēģina īstenot savus procesus un datu pārvaldības struktūras relāciju datubāzes ietvaros, un, ja datubāzes piedāvātā SQL variācija neļauj izveidot vajadzīgās struktūras, vai to struktūru pārvaldība ir nepietiekami optimāla, tiek meklētas datu pārvaldības alternatīvas. Tas nozīmē vairāku datubāžu iestatīšanu, piemēram izmantot relāciju datubāzi klasiskās funkcionalitātes ieguvei un NoSQL datubāzi dokumentu glabātuves piesaistei.

1.3. Relācijas datubāzes pamata īpašību apraksts

Relāciju datubāzes īsteno un papildina SQL standartu vairākos veidos, tomēr katra datubāze piedāvā pamata operācijas un koncepcijas, kuras konceptuāli un fiziski definē darbu ar datiem ER modelī.

1.3.1. Datu pārvaldības pamata operācijas

Relāciju datubāzes ir populārākais datubāzes veids un aizstāj tīmekļa un hierarhiskās datubāzes, tas galvenokārt ir saistītas ar to, ka relāciju datubāzes koncentrēja darbu ar datiem loģiskā līmenī. Par pamatu tika ņemts "rindas" jēdziens, kas apzīmē vienu datubāzes ierakstu (entitiju). Lietotajam tiek piedāvāts loģisks skats, kura ietvaros ir iespēja veikt CRUD (Create, Read, Update, Delete) operācijas un pārvaldīt datubāzes loģisko struktūru. Datubāzes shēmas tiek sadalītas trijās daļās - konceptuālā shēma (Conceptual Schema), kurā tiek aprakstīti visi dati, kas ir

datubāzē, fiziskā shēma (Physical Schema), kurā ir detaļas par fizisko datu glabātuvi, un lietotāja (jeb ārējā) shēma (User View), kur tiek piedāvāti dati, kurus ievadījis un var izmantot lietotājs [4,6].

1.3.2. Pārvaldības struktūras

Datubāzes pārvaldību var sadalīt trīs daļās: DDL (Data Definition Language), DML (Data Manipulation Language) un DCL (Data Control Language). DDL valoda tiek lietota, lai pārvaldītu datubāzes tabulu struktūru no lietotāja skata. DML ir domāts datu pārvaldībai, t.i., CRUD operācijām ar datiem. DCL dod iespēju pārvaldīt piekļuvi datiem - lietotājus un viņu piekļuves konfigurācijas, lomas un lietotājus grupas [4].

1.3.3. ACID īpašības un to ietekme uz datubāzes attīstību

Viens no populārākajiem principiem, kas plaši tiek izmantots relāciju datubāzēs, ir ACID (Atomicity, Consistency, Isolation and Durability). Šis termins apraksta transakcijas jēdzienu un īpašības. Pirmās transakcijas sāka parādīties ap divdesmita gadsimta septiņdesmitajiem gadiem, IMS sistēma atbalstīja transakcijas 1973. gadā. ACID termina komponentes nav formāli definētas, kā arī nav matemātiskas aksiomas, tomēr tiek izmantotas, lai uzsāktu un definētu dialogu par transnacionālām sistēmām. Terminu neformāla definīcija ir šāda:

- *Atomicity* ir operāciju grupas izpildes garantija. Vai nu visas operācijas grupā tiek izpildītas, vai neviena operācija nav izpildīta. Tiek realizēta, izmantojot datnes slēgšanu (locking) un reģistrēšanu (logging) procesu.
- *Consistency* ir garantijas datubāzes ietvaros, SQL integritātes ierobežojumi tiek lietoti DBMS konsistences definīcijā - transakcija vai beigties tikai tad, ja tā atstāj datubāzi konsistentā stāvoklī. Tiek pārvaldīta ar datu pārbaudes mehānismiem koda / pieprasījumu izpildes laikā.
- *Isolation* ir garantija, ka vienas transakcijas ietvaros lietotājs neredzēs citu transakciju izmaiņas, kamēr tās nav izpildītas. Izolācija lietotnes izstrādātājiem ļauj neanalizēt nekorektus datus, kuri tiek veidoti citos procesos. Tiek realizēts kā datnes slēgšanas (locking) un reģistrēšanas (logging) process.

- *Durability* ir garantija, ka pēc transakcijas veiksmīgas izpildes dati tiek saglabāti energoneatkarīga atmiņā un netiek pazaudēti. Tiek realizēts reģistrēšanas (logging) un datu atjaunošanas (recovery) protokolā.

Ja transakcijas laikā parādās kļūdas vai nesakarības, tā tiek atcelta un tiek parādīts kļūdas kods[7].

ACID īpašību alternatīva ir BASE (Basic Availability, Soft-state, Eventual consistency) īpašības, kuras parasti tiek iebūvētas NoSQL datubāzēs. BASE koncepcijas ietvaros dati var eksistēt neskaidrā stāvoklī (soft state), t.i. dati var atrasties nekonsistentā vai pieprasījuma laikā mainīgā stāvoklī. Kad beidzās ienākošie pieprasījumi, ar laiku, datu stāvoklis kļūst konsistents [8].

1.3.4. Datubāzes apraksta formulējums

Izmantojot noteiktus terminus, ir iespējams aprakstīt datubāzes pamata funkcionalitāti un pielietojuma sfēras. Galvenie jautājumi, kurus nepieciešams uzdot, izvēloties datubāzi, ir:

1. Kādas no datu pārvaldības operācijām (CRUD) tiek atbalstītas? Dažas datubāzes atbalsta tikai CRD, bez datu mainīšanas iespējām.
2. Cik detalizēti tiek dokumentēti / aprakstīti datu pārvaldības skati (Conceptual Schema, Physical Schema, User View) ? Dažreiz lietotāja skats ir piedāvāts tikai SQL veidā, kā arī konceptuālā skata definējums ir izplūdis, neskatoties uz to, ka tas ir līdzīgs relāciju modelim.
3. Cik attīstītas ir DML, DCL, DDL?
4. Vai datubāze atbalsta ACID vai BASE principus, un cik lielā mērā?

Parasti relāciju datubāzes atbalsta visas CRUD iespējas, īsteno visus pārvaldības skatus ar vairākām alternatīvām, pietiekami īsteno DML, DCL, DDL moduļus, ka arī pilnīgi īsteno ACID principus.

1.4. Secinājumi par datu pārvaldības vēsturiskajām izmaiņām, pielietošanas un attīstības tendencēm

Datubāzes sākotnēji tika izveidotas, lai atvieglotu darbu ar datiem. Pirmās datubāzes maksimāli tuvu atspoguļoja cietā diska interfeisa realizāciju operētāj sistēmā un bija orientētas uz datu glabāšanas un piekļuves optimizāciju. Tomēr laika gaitā parādījās dalījums fiziskajā un

loģiskajā skatā. Darbs ar datu loģisko skatu kļuva populārs un datubāzes sāka realizēt relāciju modeli, SQL interfeisu un veidot papildus funkcionalitāti ap šādām parādībām.

Pirmās relāciju datubāzes tika izveidotas kā slēgtās sistēmas un izvietotas uz atsevišķiem serveriem, jo tolaik relācijas datubāzes funkcionalitātes īstenošanai vajadzēja vairākus resursu, bija jāminimizē citu lietotņu resursu izmantojums. Datubāzes tika izolētas un bija definēts SQL piekļuves interfeiss ar lietotāju piekļuves kontroli. Tehniski sarežģītas lietas, t.i., detaļas par datu glabāšanu, ierakstu fragmentāciju un transakciju (un citu papildus funkcionalitātes) realizāciju tika paslēptas zem loģiskā datu skata.

Ar laiku datoru resursi kļuva lētāki un pieejami uzņēmumiem, tas atļāva vākt un digitalizēt reālās pasaules informāciju, savukārt ER modelis, ACID principi un SQL interfeiss kļuva par rīkiem, kuri palīdzēja šo informāciju kontekstualizēt. Līdzīgs kontekstualizācijas process parādījās arī kā programmēšanas paradigma - objekt orientēta programmēšana, kas izveidoja vairākus papildus rīkus (un pat datubāzes), kuras mēģināja savienot lietojumprogrammatūras (koda) un datubāzes (datu) slāņus, SQL valoda tika integrēta ar objekt orientētu pieeju.

Ar laiku parādījās NoSQL datubāzes, kuras neīstenoja (vai daļēji īstenoja) relāciju modeli, tomēr koncentrējās uz datu apstrādes efektivitātes paaugstināšanas, jo relāciju datubāzes nespēja piedāvāt pietiekami augstas veiktspējas kopā ar relāciju datubāzes funkcionalitāti. Parādījās vairākas datubāzes, kuru mērķis bija paātrināt veiktspējas vai vienkāršot darbu ar dokumentiem, grafiem vai notikumu rindām.

Dalītas datubāzes bija veids kā optimizēt datu glabāšanu un apstrādi. Tas fakts, ka gan SQL, gan NoSQL datubāzēm ir definēts ārējas komunikācijas interfeiss (SQL, JavaScript vai cits) un lietotājs strādā tikai ar loģisko modeli, palīdzēja dalītas datubāzes funkcionalitātes veidošanas procesā. Starp-datubāzes komunikācijā arī ir nepieciešams vienots ārējais interfeiss, kas dotu iespēju koordinēt veicamās operācijas, kā arī īstenot ACID principus un kontrolēt CAP teorēmas ietekmi uz klastera stabilitāti. Tomēr ne visas NoSQL datubāzes iebūvē ACID funkcionalitāti, bet īsteno BASE, jo konceptuāli vai / un tehniski BASE pieeja ir vieglāk pārvaldāmā, realizējama.

Pašlaik datubāzes ir slēgtas sistēmas, kuras tiek uzstādītas kā instalējamā programmatūra, tomēr darbam ar datnēm un datiem trūkst kopējas metodoloģijas, kura būtu pieejama gan zema, gan augsta līmeņa programmēšanas valodās. Efektīvāk vienas lietotnes ietvaros ir pieslēgt (vai īstenot) nepieciešamo funkcionalitāti darbam ar datiem, tā atbilst izvēlētai loģiskai datu uztveres paradigmai, nevis uzstādīt de fakto vispārinātu loģisku un tehnisku realizāciju un mēģināt to pielāgot lietotnes vajadzībām. Datu pārvaldība kā komponents ir izdevīgāks par datu pārvaldību kā

servisu, jo serviss ietver sevī vairākas abstrakcijas, kuras, datu uztverei mainoties, izveido papildus interfeisa slāņus, un modernai lietotnei, lai realizētu darbu ar datiem, ir jāizmanto ORM (vai cita konceptualizācijas) bibliotēka, kura transformē savas entītijas (vai citus ierakstu reprezentācijas elementus) SQL valodas pieprasījumos, kuri tiek padoti caur valodā iebūvētu datubāzes draiveri, kurš padod komandas datubāzes SQL kompilatoram, lai palaistu vienkāršas operācijas ar datiem.

1.5. Relāciju datubāzes pirmsākumi

Šajā nodaļā ir aprakstīts viena no pirmajiem relācijas datubāzes arhitektūras variācijām, koncentrējot uzmanību uz sistēmas dizaina pieņēmumiem un sistēmas izstrādes laika fiziskajiem ierobežojumiem. Nodaļas mērķis ir aprakstīt risinātās problēmas un pamatojumus, kā arī analizēt SQL izveidošanas motivāciju.

1.5.1. Sistēmas R apraksts un pamatojums

Sistēma R ir datubāzes sistēma, kura tika izveidota septiņdesmitajos gados Sanhosē pētījumu laboratorijā. Sistēma R bija pirmā sistēma, kurā tika īstenota SQL (tolaik SEQUEL) valoda, kā arī vairākas pamata arhitektūras, relācijas modeļa un transakciju algoritmi tika izveidoti Sistēmas R veidošanas laikā. Vairākās datubāzēs implementējas Sistēmas R pieejas un optimizācijas, piemēram DB2 [9].

Sistēma R bija pirmā sistēma, kura piedāvāja augstā līmeņa datnes piekļuves interfeisu, izolējot gala lietotāju, cik vien iespējams vairāk no datu pārvaldības dzinēja. Augsta līmeņa datu kontroles un pārvaldības komponentes bija - relāciju skatu definīcija, lietotāju autorizācija, integritātes pārbaudes, transakcijas, reģistrēšanu (logging) un atjaunošanas (recovery) apakšsistēmas. Reģistrēšanas un atjaunošanas sistēmas nodrošināja datu konsistenci paralēlu datu modificēšanas vidē. Pirmās relāciju pieejas idejas aprakstīja E.F. Codd 70-jos gados, un “Sistēma R” projekts tika uzsākts kā pētījuma projekts, kurš izpētītu iespējas relāciju modeļa īstenošanai un ātrdarbībai vismaz līdzīgai eksistējošām datubāzes sistēmām, nevis kā patentēts gala produkts [10].

1.5.2. Sistēmas R arhitektūra un pamatidejas

Sistēmas R arhitektūra sastāv no trim pamata elementiem, tie ir - datubāzes piekļuves interfeisu kolekcija, relācijas datu interfeiss (RDI, relatioinal data interaface) ar relāciju datu sistēmu (RDS, relational data system) un relācijas krātuves interfeiss (RSI, relational storage interface) ar relācijas krātuves sistēmu (RSS, relational storage system). Arhitektūras vizualizācija ir pieejama pielikumā 1. RDI ir ārējais interfeiss, kuru ir iespējams izsaukt tieši no programmēšanas valodas vai izmantojot ārējas programmas vai emulatorus, kuri atbalsta RDI interfeisu. RDS atbalsta RDI un nodrošina autorizāciju, lietotāja sesijas integritātes, atbalstu alternatīviem skatiem un meta datiem. Komunikācijas valoda Sequel ir iebūvēta RDI interfeisā nu tiek lietota kā pamats visām datu definīcijām un manipulācijām. Papildus RDS apakšsistēmā ir iebūvēts optimizators, kurš nosaka piekļuves un izpildes ceļu, kurus atbalsta RSS sistēma, katram ārējam pieprasījumam [10].

Datubāze tika paredzēta lietošanai kopā ar VM/370 operētāj sistēmu, kura virtualizēja pieeju resursiem. Virtualizācija tika izvēlēta kā veids, kas nodrošinātu vairāku lietotāju piekļuvi. Katram ielogotām lietotājam tiek veltīta vesela virtuāla datubāzes mašīna, kura satur visas tabulas un kodu, lai nodrošinātu datu pārvaldības funkcionalitāti. Papildus pastāv uzraudzības serviss, kurš pārvalda virtuālās vides un nodrošina datubāzes resursu lietošanas statistikas vākšanu [10]. Arhitektūras vizualizācija ir redzama pielikumā 2.

1.5.3. Sistēmas R pieprasījuma valodas mērķis un motivācija

SEQUEL valodas izveidošanu ietekmēja strukturētas programmēšanas metodoloģija un salasāmas pieprasījumu valodas meklējumi. SEQUEL valodai bija alternatīva - SQUARE, kura tika izveidota daudz agrāk un uz kuru SEQUEL valoda sākotnēji tika bāzētā. SQUARE valoda bija labi strukturēta, bet grūti salasāma. SEQUEL valodas izstrādātāji mēģināja pielīdzināt valodu angļu valodai un uzlabot lasāmību, lai vienkāršotu darbu ar datiem tiem lietotājiem, kuri iepriekš nav lietojuši datubāzes un pieprasījumu valodas, bet kuriem ir jāstrādā ar datiem.

Oriģinālais “SEQUEL: A Structured English Query Language” koncentrējās uz valodas datu pieprasījumu nodrošināšanu, tikai pavirši aplūko pārejās datu pārvaldības nianses. Datu izmaiņas un piekļuves kontrole, kā arī kļūdu konceptuāla apstrādē tiek aplūkota vēlāk, citos rakstos. Pieprasījumus pēc datiem izdevīgāk veikt ar kopas izteikumiem, kas ļauj datubāzei optimizēt

pieprasījumu, nevis izmantojot viens-pēc-viena ierakstu iterācijas. Neskatoties uz loģiskā un fiziskā skata datu pārvaldības niansēm, ir svarīgi saprast kā un kādā veidā lietojamās sistēmas ir projektētas un kā tās pārvalda datus [11].

1.5.4. Secinājumi par Sistēmas R izstrādes ieguldījumu datubāzes koncepta attīstībā

Relāciju modelis tiek popularizēts kopā ar SQL (SEQUEL) valodu, kura strukturē un apraksta relāciju modeli tehniskā valodā, tomēr tā ir salasāma netehniskiem lietotājiem. SQL valodas lasāmība bija viens no valodas dizaina galvenajiem punktiem. Sistēmas R izveidošanas laikā pastāvēja SQL alternatīvas, tomēr lasāmības dēļ SQL valoda kļuva populārāka par pārējām. Papildus faktors bija SQL pielāgojamība jauniem lietotājiem, jo valoda nebija sarežģīta un pārklājās ar angļu valodu.

Sistēmas R tika veidota ar mērķi pierādīt, ka Relāciju modelis var būt izmantots produkcijas vides datu pārvaldībai un mēģināja risināt šādas problēmas:

1. Īstenot relāciju modeli, kas palīdzētu atdalīt konceptuālo un tehnisko skatu. Iepriekš datubāzes īstenoja visu kopā, kas netehniskiem cilvēkiem apgrūtināja datu struktūru un pārvaldības procesus.
2. Izveidot lasāmu datu pārvaldības interfeisu, kas dotu iespēju netehniskiem cilvēkiem pārvaldīt datus.
3. Īstenot vairāku lietotāju vienlaicīgu datubāzes lietošanu caur doto piekļuves interfeisu.
4. Nodalīt metadatu no datu pārvaldības.

Rezultāta Sistēma R piedāvāja gan piekļuves interfeisu, gan lietotāju sesiju un piekļuves pārvaldību, izmantojot piekļuves interfeisu, kas papildus veicināja loģiskā skata atdalīšanu no fiziskā skata. Ir svarīgi pieminēt, ka Sistēmas R datiem bija arī tehniskā piekļuve, izmantojot programmas interfeisu, kas tika domāta programmēšanas uzdevumu risināšanai, tehniska interfeisa izmantošanai bija nepieciešamas tehniskas zināšanas par datubāzes uzbūvi un iekšējiem procesiem.

1.6. Datubāzes SQL ISO standarts un īstenošanas piemērs

SQL valodai tika definēti ISO standarti, kuri apraksta SQL konceptuālo satvaru un definē SQL gramatiku un datu apstrādes instrukciju rezultātus, daļēji definējot īstenošanas detaļas. Standartu pamatnes ir pieejamas atvērtā piekļuvē un tiek pielietotas vārākajos mūsdienu datubāzes

projektos, piemēram MySQL, PostgreSQL, MSSQL, SQLite, utml. Šī nodaļa aprakstīs pamata datubāzes funkcionalitāti no SQL ISO un analizēs standarta īstenošanas nianses vienā no populārākajām atvērta koda datubāzēm MySQL.

1.6.1. Datubāzes metadatu pārvaldība

SQL standartā tiek definētas vairākas sistēmu tabulas metadatu pārvaldībai. Ir nepieciešams izveidot un uzturēt SQL shēmu INFORMATION_SCHEMA, papildus šai shēmai ir nepieciešams būt pieejamai no SQL valodas. Shēma satur informāciju par visiem datu struktūru objektiem, t.i. tabulām, kolonnām, skatiem, procedūrām un funkcijām, utml. Shēma ir publiski pieejama, un lietotāji var to izmantot, lai programmatiski spriestu par datu struktūrām, veidot dinamiskus pieprasījumus vai arī implementēt ORM funkcionalitāti lietotnē. Papildus tiek definēta DEFINITION_SCHEMA shēma, kura satur tehnisko informāciju par tabulām un ir slēgta SQL pieprasījumiem. Definīciju shēma satur datus, kuri tiek piedāvāt Informācijas shēmas skatos. Lietotāju un pieejas pārvaldība notiek arī izmantojot INFORMATION SCHEMA [12].

Praksē gandrīz visas MySQL komandas izmanto INFORMATION_SCHEMA tabulu. Ir vairākas SQL komandas, kuras maina standartu SQL sintaksi, lai īstenotu vieglāku pieeju shēmas tabulai. Piemēram, komanda “SHOW TABLES” parāda visas tabulas no sesijā izmantotās datubāzes, tomēr pilns un no SQL sintakses viedokļa korekts pieprasījums izskatītos šādi “SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_SCHEMA = ‘<current_database>’”. Saīsinātas komandas piedāvā vienkāršāku alternatīvu darbam ar informācijas shēmu, tomēr sarežģi SQL sintaksi [13].

Viena no pamata MySQL datubāzes struktūrām ir tabula, kura tiek definēta “sql/table.h” failā kā “TABLE”, papildus tiek definēta “TABLE_SHARE” klase, kura īsteno tabulas datu daļēju sadalījumu starp procesiem. Tabula tiek izmantota vairākos koda moduļos, piemēram, SQL interpretatorā, optimizētajā, piekļuves kontrolē vai pieprasījumu kešā, jo reprezentē tabulas entītiju no vairākiem programmas slāņiem. Tabulas klase satur vairākus metadatus par tabulu, kā arī īsteno vairākas datu struktūras pieprasījumu izpildei. Piemēram, lauku deskriptorus (kolonnas), tabulas nosaukumu, primāras atslēgas un indeksus, iespējamās operācijas ar tabulu, ierobežojumus, asociētas īslaicīgas tabulas, utml. Tabulas struktūras objekti tiek ielikti tabulas kešā, lai optimizētu metadatu izmantošanu nākamajiem tabulas pieprasījumiem [14].

Struktūra, kura ir cieši saistīta ar tabulas definējumu un operācijas ar datiem, ir lauks, kodā atrodas “sql/field.h” failā un tiek definēta kā “Field” abstraktā klase. Šī klase tiek izmantota kā pamats lauku definējumam. Katram lauku tipam, t.i. datu tipam, ir sava klase, kura manto lauka abstrakto klasi un aizpilda to ar saviem nosacījumiem un atribūtiem. Piemēram, lauka tips, tabulas nosaukums un lauka nosaukums, vērtība, lauka garums, kā arī lauka operācijas, pievienot vērtību, nolasīt vērtību, iztīrīt lauku [14].

MySQL datubāze piedāvā datu krātuves interfeisu, kuru manto vairāki datu pārvaldības moduļi, piemēram, InnoDB, MyISAM, Memory. Katrs datu pārvaldnieks īsteno interfeisā definētas operācijas, pie kura piekļuve tiek īstenota caur vienotu SQL interfeisu. Papildus moduļiem ir iespēja paplašināt SQL valodas iespējas kontētā pārvaldnieka izmantošanas gadījumā. Tomēr dažas operācijas tiek veidotas ar vienotu SQL komandu moduli un pārvaldniekam tiek padota notikumu informācija, šādi pārvaldniekam ir tikai daļēja kontrole par datubāzes operācijām. Piemēram, tabulas veidošana ir definēta datu pārvaldības interfeisā, bet kolonnas pievienošana ir īstenota tabulas struktūras uzturēšanas modulī [15].

1.6.2. Datubāzes transakcijas un datu pārvaldība

SQL standarts definē transakciju kā SQL priekšrakstu kopu, kuru ir iespējams atkopt ar vienu komandu. SQL transakcijas notiek sesijas ietvaros, tomēr izmanto kopīgus datus. Pēc transakcijas veiksmīgas izpildes dati ir pieejami pārējām transakcijām un sesijām. Pēc transakcijas atkopšanas, dati netiek mainīti pamata datu krātuvē. Kļūdas gadījumā uzvedība netiek definēta standartā [12].

Papildus standarts definē vairākus transakciju izolācijas līmeņus, no kuriem ir atkarīgi transakciju iespējamie fenomeni. Transakciju fenomeni ir neizpildītu datu lasīšana (dirty read, P1), neatkārtojamā datu lasīšana (non-repeatable read, P2), fantomā datu lasīšana (phantom, P3).

Transakciju veidu un iespējamie fenomeni ir pieejami attēlā 1.6.2.1. Pēc noklusējuma standarts definē realizējamu transakciju tipu (serializable). Tiek definēti COMMIT un ROLLBACK atslēgvārdi transakciju izpildes un atkopšanas operācijām [12].

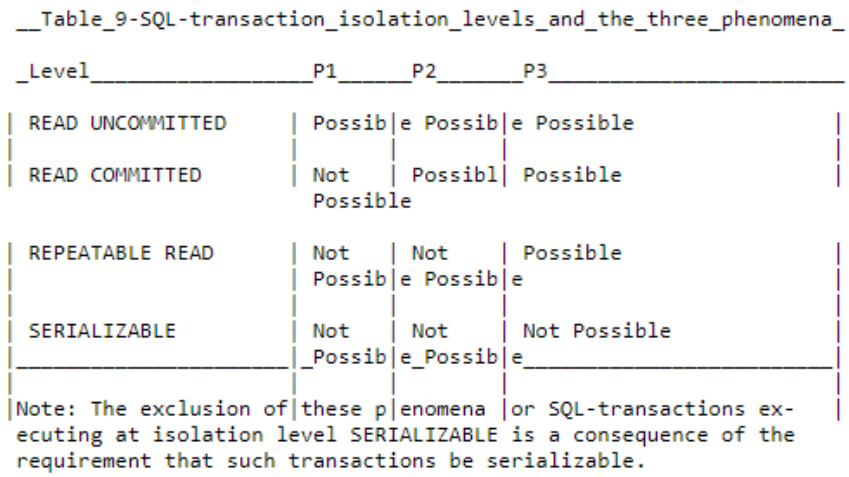

__Table_9-SQL-transaction_isolation_levels_and_the_three_phenomena_

_Level	P1	P2	P3
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

Note: The exclusion of these phenomena for SQL-transactions executing at isolation level SERIALIZABLE is a consequence of the requirement that such transactions be serializable.

att. 1.6.2.1. **Transakciju tipu un fenomenu atkarība [12]**

MySQL datubāzes transakciju īstenojums ir cieši saistīts ar datu krātuves interfeisu mehāniku. Datubāze īsteno vienu interfeisu, ko dēvē par - "handler" un satur vairākas virtuālas metodes, kuras tiek izsauktas datubāzes darbības laikā. Metodes piemēri ir update_auto_increment, write_row, update_row, delete_row. Katra krātuve īsteno handler interfeisu un tas metodes. Tomēr ar laiku paradījās arī papildus process, kas īsteno līdzīgu komunikāciju. Datubāzei ir vairāki procesi, kuri notiek ārpus krātuves objekta un ir pamatā nemainīgi krātuves tipu mainot. Tika izveidota "handlerton" datu struktūra, kura satur katrai krātuvei atzvanīšanas funkcijas datubāzes procesiem. Šīs atzvanīšanas funkciju modelis tiek lietots, lai īstenot transakcijas. Transakciju atbalsts ir atkarīgs no krātuves moduļa, piemēram, MyISAM neatbalsta transakcijas, InnoDB atbalsta transakcijas. MySQL satur atzvanīšanas funkciju izsaukumus vietās, kur sākas vai beidzās datu lasīšana / rakstīšana un ģenerē bloķēšanas / atbloķēšanas notikumus, tad krātuves pašas īsteno saistītas ar tiem notikumiem darbības. Tāda pati pieeja tiek izmantota datu indeksācijas atbalstam, papildus informācija tiek padota metodēm, kuras maina datus, un atzvanīšanas funkcijas tiek lietotas, lai informētu optimizatoru par indeksu esamību [14].

InnoDB atbalsta visus SQL ISO:1992 definētos transakciju tipus, tomēr pēc noklusējuma tiek izmantots atkārtojams nolasījums (repeatable read), kas tiek pamatots kā optimāls izolācijas līmenis. Transakciju dati tiek saglabāti atmiņas buferī, tomēr dažreiz atrodas arī datu tabulā, katra

krātuves datu manipulācijas operācija notiek caur transakcijas procesu. Atmiņas buferis tiek izmantots, lai uzkrātu lappuses (page) datus pirms rakstīšanas uz diska. Ieraksti lappusē tiek saglabāti secīgi, tomēr lappuse satur papildus metadatus segmenta sākumā un beigās. Lappuses sākums ir galvene, kas satur informāciju par tagadējo lappusi, kur atrodas nākama lappuse un iepriekšējā, kā arī informāciju par pirmajiem / pēdējiem ierakstiem lappusē, kopējo ierakstu skaitu, utml. Beigās tiek definētas mapes sloti un lappuses kontrolsumma. InnoDB krātuves ieraksts sastāv no lauku atrašanas nobīdēm (offset), informāciju par ierakstu, kas sevī ietver vai ieraksts ir nodzēsts, cik lauku ir ierakstā, utml., un paši ieraksta dati [13, 15].

MyISAM krātuve neizmanto lappuses, līdz ar to tām ir daudz vienkāršāka struktūra. Ieraksts sastāv no bita, kas definē, vai ieraksts tika nodzēsts vai ne, tad tiek definēta bitu secība, kura definē kādiem laukiem ir NULL vērtības, un tad tiek ierakstīti pasi lauki. Papildus dinamiskiem laukiem glabājas lauka garums (piemēram BLOB tipam). Transakciju vietā tiek izmantota vienkāršā failu bloķēšanas metode [13, 15].

1.6.3. Datubāzes vienlaicīgie procesi

SQL ISO:1992 standarts apraksta vienlaicīgus procesus tikai transakcijas koncepta ietvaros, t.i. realizējams transakcijas tips nodrošina secīgu transakciju izpildi, citiem transakcijas tipiem vienlaicīgu procesu laikā var parādīties viens to trim fenomeniem (P1, P2, P3). Tomēr tehniski ir vairāki veidi, kā nodrošināt konsistenci vienlaicīgu procesu vidē. Ir vērts pieminēt, ka datu konsistences un operāciju korektas secības nodrošināšana ir viena no svarīgākajām datu pārvaldības īstenošanas problēmām [12].

MySQL izmanto uz pavedieniem bāzētu pieprasījumu apstrādi. Tiek palaists galvenais MySQL datubāzes process, kurš inicializē datubāzes konfigurācijas un sāk klausīties servera portus, kad ienāk jauns pieprasījums, datubāzes process izveido jauno pavediena procesu, kurš strādā dotā pieprasījuma (vai arī klienta sesijas) ietvaros. Ir divi paņēmieni paralēlas apstrādes datubāzes izveidošanai - izmantot pavediena procesus vai apakš procesus. Pavediena procesu izmantošana ļauj lēti (no resursu izmantošanas viedokļa) veidot jaunas instances, pamata izdevumi ir saistīti ar pavediena procesu pārvaldnieka resursu pieprasījumu. Papildus uzreiz ir pieejami visi servera dati, un vienlaicīga piekļuve tiek nodrošināta izmantojot mutex bloķēšanu. Pavediena procesi tiek ātri izveidoti un nepieprasa daudz resursus konteksta maiņai, kas labi strādā lielas slodzes sistēmās. Tomēr pavediena sistēmas ir grūti testēt, bet kļūdas pavediena procesu pārvaldībā

spēj korumpēt visu procesu, arī ir ierobežojumi 32-bitu sistēmās, kur vienam procesam ir veltīts līdz 4-iem gigabaitiem [14].

Salīdzinot ar apakš procesu pieeju, apakš procesu izveidošanai nepieciešams vairāk atmiņas un dažreiz daļa no izveidošanas procesā klonētas informācijas nav nepieciešama. Ir nepieciešami speciāli paņēmieni, lai nodrošinātu apakš procesu datu piekļuvi galvenā procesa datiem, kā arī vienlaicīgu piekļuvi pie datiem. Tomēr tos ir vieglāk testēt, kā arī procesi un apakš procesi nav tik cieši saistīti savā starpā, lai kļūdas gadījumā nedarbotos visa sistēma [14].

MySQL satur THD struktūru, kura nodrošina darbības ar pavediena procesiem. Kad THD objekts tiek izveidots, tas tiek reģistrēts pavediena pārvaldības modulī. Visu pavediena procesu sarakstu ir iespējams apskatīt ar "SHOW PROCESSLIST" komandu jeb arī izpētot INFORMATION_SCHEMA tabulas. Tomēr vairāku pavedienu procesi var vienlaicīgi pieprasīt piekļuvi kopējiem datiem. Datu pieejas sadalījums tiek nodrošināts ar vairākiem paņēmieniem, galvenais no kuriem ir mutex izmantošana. Mutex ir datu pieejas ekskluzīvā bloķēšana, kura ir ļoti ātra un vienkārša, tomēr mutex pieeja nav efektīva gadījumā, kad vairākiem procesiem ir nepieciešams nolasīt datus (bez rakstīšanas), jo teorētiski datus var nolasīt paralēli bez bloķēšanas. Tad tiek izmantoti read-write bloku mehānisms, kurš atļauj gan ekskluzīvus, gan kopējus datu blokus, tomēr izmanto vairāk CPU resursu. MySQL izmanto abus, atkarībā no situācijas. Vienu MySQL pieprasījumu var vienlaicīgi izpildīt tikai viens process, kas savukārt veicināja modinātajā moduļa īstenojumu. MySQL dažreiz apstādina pavediena procesus, lai atbrīvotu CPU resursus citiem procesiem, svarīgākajiem pieprasījumiem [14].

1.6.4. Secinājumi par MySQL datubāzes SQL realizāciju un izmantotām pieejām

SQL ISO:1992 standarts definē pieprasījumu valodas konceptuālus aspektus un sintaksi. Standarts detalizēti apraksta metadatu glabātuves, konceptuālu datubāzes struktūru, t.i. tabulas, kolonnas, operācijas ar tām struktūrām un iespējamus parametrus. Bieži daļa no SQL definētiem procesiem netiek īstenota, jo ir atkarīga no īstenošanas niansēm, ierobežojumiem. Papildus standarts apraksta klienta\ servera koncepciju, sesijas un kāda veidā tas saistās ar relāciju modeli.

Pamata paradigmas un datubāzes uzvedības modeļus MySQL īsteno pēc SQL standartiem, tomēr bieži datubāzes optimizācijas dēļ vai tehnisko ierobežojumu dēļ standarts netiek ievērots. MySQL datubāze ir daļēji modulāra datubāze, kura atbalsta vairākus krātuves moduļus, izmantojot vienotu datu manipulāciju interfeisu. MySQL datubāze atbalsta pamata SQL sintaksi, tomēr

sarežģītākās lietas, piemēram, transakcijas, tiek definētas krātuves programmas līmenī kā programmas un SQL valodas paplašinājums. MySQL nevar īstenot SQL standartu pilnā mērā, jo programmas arhitektūras līmenī neatbalsta vienotu sintaksi un uzvedību. Tomēr tā SQL valodas un tehniskas realizācijas dažādība MySQL datubāzes ietvaros ir laba lieta, jo ļauj izvēlieties krātuves programmu, kura labāk atbilst konkrētā gadījuma datu pārvaldības uzdevumiem un mērķiem. Piemēram, apskatot InnoDB transakciju programmas kodu, ir redzama procesu sarežģītība un vietām liekas operācijas, salīdzinot ar MyISAM pieeju, kur transakciju nav. Pat salīdzinot ierakstu formātu, MyISAM krātuve ir vienkāršāka, un loģiski pieņemt ka arī ātrāka. Speciālā gadījumā, kad problēmas risinājumam nav nepieciešamas transakcijas, bet gan operāciju ātrdarbība, ir vērts izmantot ne-transakciju MyISAM krātuvi. Modularitāte mēdz veicināt konkrētu datubāzes lietotāju mērķu sasniegšanu un piedāvā elastīgāku funkcionalitāti, kas tiks izpētīts dziļāk nākamajās nodaļās.

Tomēr no lietotnes arhitektūras un programmas koda viedokļa MySQL satur vairākas problēmas un ne konsistences. MySQL savā pamata funkcionalitātē nesatur transakciju atbalstu un transakcijas tiek īstenotas lokāli krātuves līmenī, tomēr SQL pieprasījuma interpretācija notiek ārpus krātuves, tas nozīmē, ka transakcijas faktiski ir SQL valodas paplašinājums, papildus, transakciju kontrole notiek caur datu bloķēšanas notikumiem, jo ne-transakciju sistēmās datnes bloķēšana ir primāras konkurentes pieejas nodrošinājuma rīks, un vēsturiski MySQL neatbalstīja transakcijas. Konceptuāli tas pielīdzina transakcijas pie datu bloķēšana, kad tomēr transakcijas gan konceptuālais, gan tehniskais apraksts ir daudz sarežģītāks un plašāks.

Neskatoties uz modularitātes pirmsākumus MySQL datubāzē, kodā trūkst konceptu un procesu sadalījuma, kā arī interfeisu un pieejas integritātes. SQL interpretācijā tiek iesaistīti vairāki lietotnes līmeni, kas, no vienas puses, ļauj paplašināt esošās pieprasījumu valodas funkcionalitāti, tomēr padara pieprasījumu valodu granulārāku, kas ierosina tikai daļēju standartu atbalstu starp krātuves programmām, kas savukārt rosina jautājumus par SQL standarta esamības nepieciešamību.

Datubāzes krātuves programmas interfeiss ir definēts vienā failā, un ir potenciāli monolīts. Papildus integrācijas tiek īstenotas, izmantojot atzvanīšanas funkcijas, tas nozīmē, ka pamata daļa, kura tiek īstenota, izmantojot vienu realizāciju, un papildus funkcionalitāte tiek definēta otrā veidā, kas liecina par konceptuālas integrācijas pieejas trūkumu un atstāj krātuves programmu konceptuāli sadalītu. Tomēr integrācijas interfeisam un atzvanīšanas konstrukcijām trūkst datu struktūru dažādību. Gandrīz visas darbības tiek īstenotas caur vienotu datu struktūru. Ir ne konsistences arī

atbildības sadalījumā starp moduļiem, piemēram, tabulas izveidošana ir definēta krātuves programmā, bet lauku izmaiņas ir ārpus tās, bet tiek kontrolētas caur atzvanīšanas funkcijām. Neskatoties uz modularitātes iemaņām konceptuālā datubāzes līmenī, koda arhitektūrā tās modularitātes pietrūkst.

2. RELĀCIJAS DATUBĀZES ARHITEKTŪRA, ALTERNATĪVĀS UN PAPILDUS IESPĒJAS

Šīs nodaļas mērķis ir veidot priekšstatu par relāciju datubāzes arhitektūru un iespējamām datubāzes uzbūves principu variācijām. Papildus izpētīt, cik efektīvi relāciju datubāzē tiek integrētas papildus, neatbilstošas relāciju modelim.

2.1. Relācijas datubāzes arhitektūra

Relāciju datubāzes īstenošanas pamatā parasti tiek ņemti SQL ISO standarti. Datubāzes platformas ierobežojumi definē, kādā mērā SQL standartiem seko un kādas papildus variācijas un konfigurācijas parādās datubāzes SQL dialektā. Tomēr pamatā ir komponentes un to iekšējā komunikācija. Šajā nodaļā tiks aprakstīta relācijas datubāzes uzbūve un tās izmantošanas iespējas.

2.1.1. Relācijas datubāzes galvenie komponenti un pieprasījumu izpildes process

Relācijas datubāzes arhitektūra sastāv no pieciem pamata komponentiem: komunikācijas ar klientiem pārvaldības, procesu pārvaldības, relāciju pieprasījumu apstrādes, transakciju krātuves pārvaldības un vispārinātas komponentu kopas (kopīgas komponentes un utilītprogrammas). Arhitektūras detalizēts skats ir atrodams pielikumā 3 [16].

Datubāzes pieprasījuma ceļš sākas ar ārējo klientu. Ārējā programmatūra, izmantojot klienta bibliotēku, datubāzes draiveri vai cita veida API, uzstāda savienojumu ar datubāzes komunikācijas pārvaldības sistēmu, kur savienojums un sazināšanās notiek caur tīmekļa savienojumu. Kad sazināšanās notiek caur ODBC vai JDBC savienojuma protokoliem, paradigma tiek nosaukta klients / serveris jeb divlīmeņa sistēma, tomēr nereti starp klientu un datubāzi tiek ievietota papildus sistēma, kura kā starpnieks reģistrē visus pieprasījumus un pārsuta datus starp sistēmām, lai vāktu statistikas datus, ieviestu papildus komunikācijas kontroles procesus, utml. Komunikācijas pārvaldības sistēma kontrolē lietotāja sesiju un pārsuta pieprasījumu tālāk procesu pārvaldības sistēmai, kā arī tiek izmantota datu pārsūtīšanai un korektu atbildes kodu saņemšanai [16].

Procesu pārvaldības sistēma nodrošina resursu izdalīšanas kontroli pieprasījuma izpildei. Pirmkārt, procesu pārvaldības sistēma definē konkrēta pieprasījuma izpildes kārtību, ir iespēja, ka

pašlaik datubāzes sistēmai trūkst resursu un pieprasījums ir jāatliek, kamēr resursi neparādīsies, vai pat pieprasījums ir jāatceļ. Papildus ir jānodrošina, lai izdalītais pavediens konkrētā pieprasījuma izpildei ir savienots ar klienta sesiju, lai vēlāk atgrieztu rezultātus [16].

Pēc pavediena piešķiršanas, pieprasījuma izpildi apstrādā relāciju pieprasījumu apstrādes sistēma. Sistēma pārbauda vai lietotājam ir tiesības piekļūt pie pieprasītajiem datiem un izpildīt doto pieprasījumu, pieprasījuma SQL teksts tiek transformēts pieprasījuma plānā, kurš definēts, izmantojot datubāzes iekšējās datu struktūras. Pēc SQL teksta kompilācijas, rezultējošais plāns tiek palaists ar pieprasījuma plāna izpildes sistēmu. Šī sistēma satur darbību kolekciju, kura īsteno select, join, unit, utml. operatorus, kuri tiek koordinēti, tajā skaitā arī pieprasījuma optimizatoru moduli [16].

Transakcionālas krātuves modulis ir lietots, lai tieši strādātu ar datu glabāšanas ierīci, t.i., lasītu datus un veiktu datu manipulācijas operācijas (create, update, delete). Papildus šis modulis kontrolē datu glabāšanas metodes, pārvietojot nepieciešamos datus starp diska glabātuvi un atmiņas buferiem. Datu manipulācijas tiek pielietotas "ACID" transnacionālās īpašības, kuru īstenošanai tiek pielietoti datnes slēgšanas (locking) un reģistrēšanas (logging) moduļi [16].

2.1.2. Paralēla piekļuve datiem relāciju datubāzēs

Viens no svarīgiem datubāzes aspektiem ir piekļuves datiem pārvaldība. Tomēr vairākiem procesiem strādājot ar tiem pašiem datiem, būs problēmas operāciju prioritātes un secības noskaidrošanas procesā, pieprasījumu akceptēšanas un datu izvadīšanas no sistēmas procesā. Paralēlas piekļuves un darba ar kopējo atmiņu organizācijā tiek lietoti vairāki buferizācijas paņēmieni. Piemēram, uz diska bāzēts datubāzes I/O pieprasījumu buferis, kas ir uz diska organizēta rinda, pie kuras ir piekļuve no vairākiem procesiem. Dati, kas ienāk datubāzes sistēmā, tiek translēti caur šo rindu. Kad ir nepieciešams nolasīt datus no datubāzes, tiek izveidots pieprasījums, kurš nosaka, kur datiem vajadzētu būt nolasītiem un kur buferī ielikt rezultātu, tas pats notiek ar datu rakstīšanu datubāzē, tiek definēta vieta buferī, kur tiek ielikti ienākošie dati, un vieta datubāzē, kur šos datus ierakstīt. Šis buferizācijas paņēmiens tiek plaši lietots arī komunikācijas īstenošanā un datnes slēgšanas (locking) īstenošanā, izmaiņu reģistrēšanas (logging) īstenošanā, utml [16].

2.1.3. Relāciju datubāzes arhitektūras īstenošanas piemērs

Viens no relāciju datubāzes īstenošanas piemēriem ir MySQL, kura arhitektūra sastāv no vairākiem moduļiem. Moduļu savstarpējās komunikācijas tendences ir atrodamas pielikumā 4. Startējot MySQL datubāzi, pirmkārt, tiek palaists inicializācijas modulis, kurš nolasa konfigurācijas failus, piešķir atmiņas buferus, inicializē globālos mainīgos un struktūras, un organizē citus inicializācijas uzdevumus. Pēc inicializācijas beigām tiek palaists savienojuma modulis, kurš sāk pieņemt klienta savienojuma pieprasījumus, veiksmīgas autorizācijas gadījumā pieprasījumi tiek nodoti komandu dispečeram, kurš vai nu izpilda komandas pats, vai nu nodot komandas citam, piemērotākam modulim [14].

MySQL koda terminoloģijā "komanda" ir pieprasījuma formulējums, tipisks komandas piemērs, ir aktīvas datubāzes shēmas maiņa, atskaite par datubāzes vai replikācijas statusu, pieslēguma aizvēršanu. Otrs pieprasījuma veids ir "vaicājums" (query), tas ir jebkura komanda, kura var tikt pārsūtīta caur vaicājumu parsētāju. Vaicājumu piemēri ir tādi SQL apgalvojumi kā SELECT, DELETE, INSERT, utml [14].

Komandu dispečers pārsūta vaicājumus uz vaicājumu parsētāju, tomēr komunikācija starp šiem diviem moduļiem notiek caur vaicājumu keša moduli, kurš pārbauda, vai vaicājumu ir iespējams iekešot un vai pieprasītā vaicājuma rezultāti jau ir kešā un aktuāli. Ja vaicājumu ir nepieciešams izpildīt (aktuālie rezultāti netika atrasti kešā), vaicājuma parsētājs, pēc pieprasījuma parsēšanas, var tālāk deleģēt izpildes kontroli vienam no vairākiem moduļiem. SELECT pieprasījumi tiek pārsūtīti Optimizēšanas modulim. UPDATE, INSERT, DELETE, tabulu veidošanas un izmaiņas komandas tiek apstrādātas ar tabulu modifikāciju moduļiem. Indeksu uzturēšanas vai datu defragmentācijas komandas tiek apstrādātas ar tabulu uzturēšanas moduli. Arī ir speciāls modulis, kura funkcija ir replikācijas uzturēšana un datubāzes statistikas un statusu pārvaldība [14].

Katrs no iepriekš minētajiem vaicājumu apstrādes moduļiem tiek kontrolēts ar centrālo kontroles moduli, kurš savukārt komunicēs ar tabulu pārvaldības moduli, atverot nepieciešamās tabulas un iegūstot datu / tabulu vai ierakstu bloku. Tālāk tiks izsūtīti vairāki pieprasījumi abstraktam krātuves dzinējam zemā līmeņa manipulācijām ar datiem [14].

2.1.4. Relāciju datubāzes datnes slēgšanas (locking) īstenošanas piemērs

MySQL un datubāzes dzinēji atbalsta trīs pamata datnes slēgšanas veidus - tabulas līmeņa slēgšana (table-level lock), ieraksta līmeņa slēgšana (row-level lock) un bloka līmeņa slēgšana (page-level lock). Datnes slēgšanas tips ietekmē šādas datubāzes piekļuves īpašības - paralēlas datu piekļuves ātrdarbība, strupsaķeres (deadlock) varbūtība un atmiņas apjoma lietojums. Tabulas līmeņa datnes slēgšana ir vieglākais veids paralēlas datu piekļuves realizācijai, jo tabulu slēgšanai netiek izmantots daudz atmiņas, strupsaķeres gandrīz nekad nenotiek, tomēr paralēlas piekļuves gadījuma lietotnes ātrdarbība samazināsies. Ieraksta līmeņa datnes slēgšana ļoti paātrina paralēlas piekļuves ātrdarbību, tomēr paaugstina strupsaķeres varbūtību un paaugstina izlietotās atmiņas apjomu, kas varētu būt neefektīvi gadījumos, kad netiek plānota paralēla piekļuve šiem datiem. Bloka līmeņa datnes slēgšana ir starpgadījums, kurš piedāvā kompromisu starp tabulas un ierakstu līmeņa bloķēšanu [14].

MySQL datubāzē datnes slēgšana tiek īstenota caur speciālo tabulas slēgšanas moduli, kurš kontrolē datnes slēgšanu vairākos līmeņos, tie ir globāla tabula līmeņa slēgšanas kontrole un lokāla slēgšanas kontrole, kura tiek īstenota un atbalstīta ar izmantoto krātuves dzinēju. Datnes slēgšanai tiek izmantotas notikumu rindas, tās datubāzes datnes slēgšanas pārvaldībā ir četras: aktīva datu slēgšana lasīšanai (current read-lock), sagaidāmo datu slēgšana lasīšanai (pending read-lock), aktīvo datu slēgšana rakstīšanai (current write-lock), sagaidāmo datu slēgšana rakstīšanai (pending write-lock). Datu slēgšanas tipi MySQL datubāzei ir pieejami pielikumā 5, kur tiek aprakstīti vairāki gadījumi, specifiski datu pārvaldības gadījumiem un SQL valodas realizācijas speciālgadījumiem [14].

Strupsaķeres varbūtības samazināšanai ir ieteicams paredzēt strupsaķeres lietojumprogrammatūras līmenī. Datubāzes līmenī nav speciālu algoritmu, kuri samazinās strupsaķeres varbūtību, tomēr ir speciāli procesi, kuri automātiski meklēs strupsaķeres un mēģinās problēmsituācijas atrisināt [14].

2.1.5. Relāciju datubāzes papildus iespējas

Viens no datu formātiem, kas kļuva populārs līdz ar WWW popularizāciju, ir XML datu formāts. XML (eXtensible Markup Language) ir datu formāts, kas ļauj iekodēt informāciju gan cilvēkam, gan datoram salasāmā formā. XML formātā ir iespējams veidot datnes ar stingri definētu

struktūru, kurās var aprakstīt objektus, atribūtus un vairākus objektu hierarhijas līmeņus. XML ir standartizēts pēc W3C un joprojām tiek plaši izmantots lietotņu komunikācijas datu iekodēšanai. Vairākas datubāzes piedāvā XML formātu kā speciālu kolonnu tipu, kurā var meklēt informāciju ar standartizētiem rīkiem, piemēram, XPath / XQuery. XML formātam ir vairākas versijas, kuras tiek definētas dokumenta metadatos. Šī formāta ieviešana pievienoja klasiskām relāciju datubāzēm dokumentu orientētas pieejas iemaņas, kas atrisināja vairākas lietošanas problēmas [4,6].

MySQL datubāze atbalsta darbu ar XML dokumentiem funkciju līmenī, tas nozīmē, ka XML dokumentiem nav veltīts atsevišķs datu tips vai tabulas tips, dati tiek saglabāti kā simbolu virknes atbilstošos tipos un tiek piedāvātas divas funkcijas meklēšanai un saglabāšanai. XML dokumenta atribūtu nolasīšanai tiek izmantots XML ceļa valoda (XPath) gan parametru nolasīšanai gan saglabāšanai. Operācijas ar JSON formātu (kas ir XML alternatīva) tiek īstenotas līdzīgā veidā, tiek definētas vairākas funkcijas, kuras spēj veikt iepriekš definētas operācijas ar simbolu virknēm, kuras tiek saglabāti dati JSON formātā. JSON operācijām ir īstenotas vairākas funkcijas darbam ar masīviem, piemēram pievienot elementu masīvam vai apvienot divus masīvus. Datu meklēšana arī notiek caur speciālām funkcijām. Priekš JSON formātu arī ir izveidots XPath valodas analogs, kurš ļauj definēt datus JSON struktūrā, tomēr pašas operācijas tiek noteiktas ārpus tas valodas, izmantojot funkcijas. XPath valoda papildus dod iespēju izpildīt operācijas ar datiem [13].

2.1.6. Secinājumi par relācijas datubāzes arhitektūru un papildus iespējām

Relāciju datubāzēm attīstoties, arhitektūra kļuva sarežģītāka par Sistēmas R arhitektūru, tomēr tas bija saistīts ar fizisko tehnoloģiju attīstību un relāciju datubāzes (kopā ar SQL valodu) koncepciju ģeneralizāciju. Arhitektūras ziņā relāciju datubāzēm palielinājās komponentu skaits - gan komponentu slānim, kurš pārvalda konceptuālo datubāzes daļu, gan komponentu slānim, kurš pārvalda fiziskās piekļuves un stabilitātes nodrošināšanas funkcionalitāti.

Relāciju datubāzes pārsvarā realizē vairākas pamata īpašības - vairāku lietotāju atbalstu, vairāku lietotāju vienlaicīgas piekļuves kontroli pie kopējiem datiem, transakcionālo piekļuvi pie datu pārvaldības, datu izkliedēšanas iespējas. Šie koncepti sāka veidoties ap Sistēmas R publikācijas laiku un ir joprojām aktuāli un turpina attīstīties.

Relāciju datubāzes pamatā ir SQL valodas atbalsts, un SQL valodu definē SQL ISO standarti. Tie standarti ir monolīti pēc savas būtības, t.i. standarti definē ne tikai konceptuālo pieeju, bet arī ārpus datu pārvaldības koncepcijas, piemēram datu tipus, sesijas un pieslējuma jēdzienus. Relāciju

datubāzes arhitektūras un funkcionalitātes paplašinājumi mēdz mantot SQL standarta monolīto pieeju.

Tas traucē veidot jaunās struktūras bez SQL struktūru iesaistīšanas, līdz ar to samazinot jauno struktūru izmantošanas iespējas un elastīgumu, piemēram MySQL īsteno XML un JSON datu tipu atbalstu, tomēr nedefinē tos kā atsevišķus tipus, bet kā simbolu virknes, kas savukārt samazina iespējamās darbības ar dokumentu struktūrām, piemērs - nav iespējams izveidot relāciju starp dokumentiem tieši, tas ir iespējams tikai piesaistot papildus SQL lauku un izveidojot SQL relāciju.

Tomēr ciešā datubāzes piesaiste pie SQL valodas un relāciju koncepcijām ir pamatota. Piesaiste pie standartiem ļauj veidot sarežģītākus risinājumu datu pārvaldības un konsistences jomā, kā arī veidot vairākus piekļuves bibliotēkas, izmantojot vienot SQL interfeisu. Tas padara datubāzi stabilāku, bet samazina datu struktūru iespējamas variācijas. Izveidojot jaunās struktūras relāciju datubāzes ietvaros, vai pat, piemēram, MySQL gadījumā, izveidojot krātuves (vai cita datubāzes aspekta) modularitātes iespējas, datubāze kļūst mazāk par relāciju datubāzes un pietuvinās vispārīgas datu pārvaldības sistēmas statusam. Ir skaidrs, ka relāciju datubāzei ir nepieciešams saglabāt monolītu pieeju interfeisa izstrādei un datu apstrādei, samazinot datu struktūru variācijas vai / un ierobežojot darbības ar ne-relāciju datu struktūrām, t.i. samazināt datu pārvaldības elastīgumu, lai konceptuāli atrastos relāciju datubāzes lomā.

2.2. Iebūvētas datubāzes kā alternatīvais datu pārvaldības rīks

Salīdzinot ar servera datubāzēm, iebūvētas datubāzes ir vieglākais veids piedāvāt modulāru pieeju, jo, lai sasniegtu modulāru reprezentāciju, servera datubāzei ir nepieciešams to uzturēt gan interfeisam, gan tehniskai daļai (programmas kodam), tomēr iebūvētas datubāzes ietvaros piekļuves interfeiss cieši saistīts ar programmas kodu, tad jāuztur tikai viena daļa. Šīs nodaļas mērķis ir izsekot iebūvēto datubāžu parādībai un izpētīt tas pieejas pielāgojamību.

2.2.1. Iebūvētās datubāzes jēdziens un pamatojums

Iebūvētas datubāzes tika izveidotas, lai nodrošinātu iespēju pārvaldīt datus lokāli, izmantojot lietojumprogrammatūras fizisko telpu, pretstatā datubāzes atdalīšanai atsevišķā serverī. Tradicionālās datubāzes pamata mērķi ir nodrošināt caurlaidspēju, mērogojamību, elastīgumu un funkcionalitāti, pie tam izmērs un resursu izmantojums nav ļoti svarīgi, jo ir samērāmi un viegli

iegūstami. Iebūvētai datubāzei pamata īpašības palielina veiktspēju un samazina resursu izmantošanu. Bieži vien iebūvētā datubāze ir viena no iebūvētām komponentēm sistēmas ietvaros [17].

Iebūvētās datubāzes saskaņojas ar komponentbāzētu programmatūras izstrādes (CBSE) paradigmu, kuras ietvaros, lai uzlabotu programmatūras kvalitāti un nodrošinātu uzticamu ilglaicīgu atbalstu un izstrādi, programmatūra tiek sadalīta komponentēs. Šai pieejai ir vairākas priekšrocības, salīdzinot ar monolītlietotnes izstrādes paradigmu:

- Izstrādes izmaksas samazinās, jo sistēmas tiek izveidotas, pieslēdzot jau eksistējošas, atkārtoti lietojamas komponentes;
- Sistēmas atjauninājumi notiek vieglāk un ir fragmentētāki, jo jaunas komponentes vai jaunas versijas var būt pieslēgtas esošai sistēmai;
- Programmatūras kvalitāte palielinās, pieņemot, ka katra komponente tiek atsevišķi pārbaudīta izstrādes laikā. Tā sistēmas izstrāde tiek fokusēta uz arhitektūras projektēšanu vairāk nekā uz izstrādes niansēm;
- Uzturēšanas izmaksas pazeminās, jo komponentes tiek izmantotas vairākās vietās un ir projektētas tieši šim nolūkam.

Tomēr kļūdas komponentu izstrādē vai sistēmas arhitektūras projektēšanas laikā var radīt problēmas sistēmas kopējā uzticamībā un integritātē. Iebūvēta datubāze, kurā tiek definēti komponenti, parasti iebūvē vairākus interfeisus, paradigmas vai vienas funkcionalitātes īstenojumus. Kad iebūvētā datubāze ir bibliotēka ir papildus priekšrocības, piemēram, iespēja pamata lietotnei izmantot datubāzes iekšējas konstrukcijas un moduļus [17].

2.2.2. Iebūvētās datubāzes paradigma un pielietošanas sfēra

Iebūvētās datubāzes parasti īsteno vienu vai abus datubāzes pārvaldības sistēmu modeļus - klienta / servera modeli vai / un bibliotēkas (jeb lietojumprogrammatūras komponentes) modeli. Klienta / servera modelis ļauj pienest papildus abstrakcijas līmeni, kas atvieglo veikt konceptuāli sarežģītākus programmatūras risinājumus, tomēr parasti pienes papildus resursu izlietojuma izmaksas. Parasti uz servera bāzēta datubāze, kura tiek izmantota kā iebūvēta datubāze, izmanto tos pašus resursu kā lietojumprogrammatūra, kas decentralizē resursu kontroli lietotnes ietvaros un var radīt resursu rezervācijas konfliktu.

Savienojumi un informācijas apmaiņa starp lietotni un datubāzes iebūvēto serveri izmanto vairākus lietotnes kontekstus, tādēļ ir sarežģītāk paredzēt pieprasījuma izpildes laiku un resursu patēriņu. Konteksta vizuālais skaidrojums ir redzams attēlā 2.2.2.1. Tomēr iebūvētā datubāze kā bibliotēka parasti nesatur standartizētas piekļuves pie datiem. Pārskats par iebūvētās datubāzes īstenotiem modeļiem ir atrodams pielikumā 6 [17].

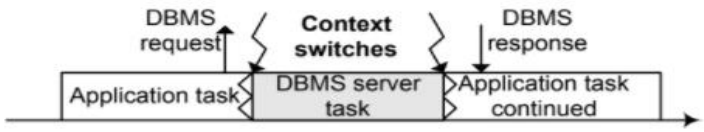

Figure 2.1: *In an embedded client/server DBMS at least two context switches are necessary for each transaction.*

att 2.2.2.1. **Iebūvētas uz servera bāzētas datubāzes konteksta vizuālais skaidrojums [17]**

Parasti iebūvētās datubāzēs tiek izmantotas reālā laika iebūvētas sistēmās, kur resursu lietošana, procesu automatizācija un stabilitāte ir prioritātes. Ir vairāki piemēri iebūvētām reālā laika sistēmām - transportēšanas sistēmas, kuras ir iebūvētas automašīnās, lidmašīnās un vilcienos, satiksmes kontroles sistēmās, ražošanas automatizācijas sistēmās, telefona un radio komunikācijas sistēmās, ēku pārvaldības sistēmās, utml. Prasības iebūvētai datubāzei ir atkarīgas no lietojumprogrammatūras, kura to izmantos, un programmatūras funkcionalitātes. Piemēram, eksistē divas datu konsistences prasību paradigmas - absolūtā konsistence, kas nodrošina sinhronizāciju starp vides stāvokli un stāvokļa reprezentāciju datubāzes datos, un relatīvā konsistence, kura nodrošina konsistenci tikai datubāzes datņu ietvaros. Tieši pēc prasību mainīguma atšķiras iebūvētās datubāzēm arhitektūra, īstenošanas paradigma un darbības principi. Tas nosaka datubāzes resursu un datubāzes interfeisa paradigmas izvēli [17].

2.2.3. Iebūvētās datubāzes piemērs

Iebūvētās datubāzes piemēra meklēšanai tika izvēlēta Berkeley DB bibliotēka, kura kā iebūvēta datubāze pēc ideju kopuma ir piemērotāka šī pētījuma pamatideju aprakstam [18].

Pirmā īstenošanas ideja, kas aprakstīta Berkeley DB arhitektūras aprakstā, ir modularitāte. Ir nepieciešams sadalīt bibliotēku vairākās komponentēs un katrai komponentei definēt piekļuves

interfeisu, kurš arī definēs moduļa funkcionālās robežas. Tas palīdzēja izvairīties no Berkeley DB datubāzes pārveidošanas monolītā lietotnē. Ar laiku bibliotēka attīstījās, vairāki moduļi tika sadalīti, citi moduļi novērsti, tomēr interfeisu lietošana atviegloja arhitektūras izmaiņas procesus [18].

Berkeley DB datubāzes sastāv no četriem pamata komponentiem - bufera pārvaldnieks, datnes slēgšanas pārvaldnieks, reģistrēšanas (logging) pārvaldnieks, transakciju pārvaldnieks. Katrai komponentei ir definēts savs ārējas piekļuves interfeiss un šie komponenti nav atkarīgi savā starpā, t.i., katru no četriem pamata komponentiem var izmantot saviem nolūkiem, piemēram, ja lietotnei ir nepieciešams, buferizēt datnes kopīgā atmiņā, var izmantot atbilstošu komponentu [18].

Berkeley DB datubāzes bufera pārvaldnieks ir Mpool, tā ir apakšsistēma, kura pārvalda datu blokus (file pages) brīvpiekļuves atmiņā caur bufera sistēmas īstenošanu. Tas palīdz optimizēt bloku pārvietošanu uz un no cietā diska, kad bloku apjoms ir lielāks par bufera izmēru. Aplūkojot Mpool apakšsistēmas funkcionalitāti detalizētāk, ir implikācijas, ka sistēma strādā līdzīgi kešatmiņas sistēmai, kur dati tiek saglabāti ātrākai piekļuvei. Piemēram, caur Mpool apakšsistēmu datubāzes indeksi tiek daļēji saglabāti brīvpiekļuves atmiņā, kas paātrina darbības ar indeksiem. Mpool strādā ar failu sistēmas abstrakciju, tas nozīmē abstraktu interfeisu operāciju ar datiem pārvaldību. Šis risinājums ļauj apakšsistēmai veikt darbības ar datiem gan uz cietā diskā, gan brīvpiekļuves atmiņā, palielinot apakšsistēmas pielietošanas gadījumus [18].

Slēgšanas pārvaldnieks ir lock, un tas tika izveidots ar objektu hierarhijas slēgšanas iespēju atbalstu, tie ir individuālu datu entītiju objekti, bloki, kur dati atrodas, faili, kur atrodas bloki vai pat failu kolekcijas.

Datnes slēgšana notiek, izmantojot konfliktu matricu, kuru lietotnes ietvaros var brīvi mainīt. Pēc noklusējuma iebūvēta matrica ir atrodama attēlā 2.2.3.1. Matrica apraksta visas iespējamas situācijas datnes slēgšanas ietvaros un vizualizē operāciju sakarības [18].

	Holder							
Requester	**No-Lock**	**Read**	**Write**	**Wait**	**iWrite**	**iRead**	**iRW**	**uRead**	**wasWrite**
No-Lock								
Read		✓		✓		✓		✓
Write	✓	✓	✓	✓	✓	✓	✓	✓
Wait								
iWrite	✓	✓					✓	✓
iRead		✓						✓
iRW	✓	✓					✓	✓
uRead		✓		✓		✓		
iwasWrite	✓	✓		✓	✓	✓		✓

Table 4.2: Read-Writer Conflict Matrix.

att. 2.2.3.1. **Konfliktu matrica Berkeley DB sistēmai [18]**

Reģistrēšanas (logging) pārvaldnieks ir Log, un tā ir apakšsistēma, kura piedāvā abstrakciju strukturētā failā, kuram piemīt tikai klāt rakstīšanas funkcionalitāte. Pēdējais komponents ir transakciju pārvaldnieks, tas ir Txn, un atbild par ACID principu uzturēšanu, kā arī kontrolpunktu veidošanu (vairāku operāciju drošībai) un datu atgūšanu pēc transakciju vai datu kļūdām [18].

2.2.4. Secinājumi par iebūvētās datubāzes pielāgojamība

Datubāze kā koncepts sastāv no vairāku ideju kopas, un tiek formēts, sākot ar vairākiem arhitektūras lēmumiem, turpinot ar kvalitatīvu īstenošanu un beidzot ar pielietošanas iespējamību. Datubāze nav vienkārši tehniska datu piekļuves organizācija, Tas ir sarežģīts informācijas apkopošanas, uzturēšanas un izmantošanas mehānisms, kuru, bez šaubām, varētu sadalīt vairākos slāņos un komponentēs.

Pirmais un galvenais jautājums datubāzes definēšanai ir, kādu datu loģisko modeli datubāze piedāvā pārvaldībai. Pastāv vairāki modeļi, no tiem populārākais, protams, ir relāciju datu pārvaldības modelis, tomēr relāciju modelis tika popularizēts kopā ar SQL (SEQUEL) valodu, kura strukturē un apraksta relāciju modeli tehniskā valodā, tomēr tā ir salasāma netehniskiem

lietotājiem. SQL valodas lasāmība bija viens no valodas dizaina galvenajiem punktiem. Sistēmas R izveidošanas laikā pastāvēja SQL alternatīvas, tomēr salasāmības dēļ SQL valoda kļuva populārāka par pārējām. Papildus faktors bija SQL pielāgojamība jauniem lietotājiem, jo valoda nebija sarežģīta un pārklājās ar angļu valodu.

Iebūvētās datubāzes atšķiras ar savu pielāgojamību konkrētiem datu pārvaldības speciālgadījumiem. Tas tiek izpausts caur loģisko modeļu dažādību, kas realizējas iebūvētajās datubāzēs. Katrs loģiskais modelis definē nepieciešamību pēc vairākām datu pārvaldības paradigmām, kas savukārt nosaka tehniskās īstenošanas nianses un arhitektūru.

Gan klienta / servera paradigmas datubāzes, gan bibliotēkas veido koda pamatu un arhitektūru no vairākiem komponentiem. Visām datubāzēm vairāki komponenti un / vai to funkcionalitāte pārklājas, piemēram, MySQL datubāzē ir "Parser" komponents, kurš pārbauda SQL valodā iesūtīto pieprasījumu un pārsūta instrukcijas tālāk pēc pieprasījuma apstrādes loģikas ķēdes, Barkeley DB datubāzē šī komponente ir ārējais API, pārbauda ārējā API izsaukuma korektumu un pārsūta pieprasījumu tālāk.

Datubāzēm (īstenotām kā bibliotēkas) ir vieglāk definēt un piedāvāt komponent-bāzētu pieeju datubāzes izstrādei, jo datubāze kā bibliotēka konceptuāli nosaka iekšējo elementu pieejamību pamata lietotnei. Tomēr relāciju datubāzes arī satur abstrakcijas parādības un komponentu definējumus, piemēram, MySQL arhitektūrā ir abstraktas krātuves dzinēja interfeiss, kurš definē zema līmeņa darba ar datiem funkcionalitāti. Komponentu definējums arhitektūras paraksta nolūkiem ir sabalsojams ar to, cik komponenti ir neatkarīgi savā starpā un cik tos ietekmēt gan datubāzes veiktspējas, gan tālākas izstrādes iespējas.

2.3. Secinājumi par pielāgojamības uzlabojuma iespējām datubāzēs

Vēsturiski datubāzes tika veidotas ap datu reprezentācijas konceptu jeb loģisko datu skatu, aiz kura tika izvietotas tehniskās datu pārvaldības īstenošanas detaļas, tas atļāva pārvaldīt datus ar datubāzes loģiskā skata pārvaldības rīkiem. Ar laiku loģiskais skats attālinājās no tehniskās realizācijas, jo datu bāzu arhitektūra kļuva sarežģītāka, piedāvājot tai vairāk funkcionalitātes.

Pēc vairāku datu bāzu koncepciju un paņēmienu izpētes var secināt, ka tām ir vairāki kopīgi elementi, piemēram, bufera izmantojums datu pārsūtīšanai uz un no diska, indeksu algoritmi un to pielietojumi, pieprasījumu aktīvais kešs, reģistrēšanas paņēmieni. Šos elementus ir iespējams izmantot datu pārvaldības metodoloģijas veidošanai, tomēr vadlīniju īstenošanai konkrētajā

programmēšanas valodā ir svarīgi izmantot komponent bāzētu pieeju, jo tā atļauj kombinēt datu pārvaldības metodes, kas palīdz optimizēt datu pārvaldību konkrētam gadījumam. Datu pārvaldības metodes nav cieši saistītas ar datu konceptuālo skatu, tāpēc, tehniskās metodes kombinējot, ir iespējams rast jaunas datu koncepcijas, kas būtu aktuālas konkrētiem mērķiem.

Arī no arhitektūras viedokļa ir vairāki kopējie elementi – ārējais komunikācijas interfeiss (kas varētu būt SQL vai programmatūras API, vai cits), vairāku veidu piekļuves kontrole (ja tā ir nepieciešama), nodalīta meta datu un datu krātuve, transakciju un procesu pārvaldība. Tās pētījuma ietvaros ir iespējams unificēt un strukturēt, veidojot funkcionālas komponentu izvietošanas iespējas arhitektūras ietvaros.

3. DATU PĀRVALDĪBAS PAŅĒMIENI UN TO MOTIVĀCIJA DATUBĀZES ĪSTENOJUMA IETVAROS

Tika apskatīti vairāki datu pārvaldības paņēmieni un īstenojumi, kur katrs ir pielāgots konkrētiem mērķiem un optimizēts iepriekš definētiem speciālgadījumiem. Ir vairākas datu pārvaldības jomas, kuras ietekmē gala risinājuma pielāgojamību datu pārvaldības interfeiss, moduļi un paņēmieni, operāciju reģistrēšana un datu atjaunošana, u.c. Dažās jomās ir nepieciešams izvēlieties konceptuālo pieeju, citās – pielāgot un integrēt vairākus paņēmienus elastīga risinājuma izveidei. Šīs nodaļas mērķis ir strukturēt datu pārvaldības jomas, iespējamos lēmumus īstenošanas procesā un definēt katra paņēmiena pielietošanas pamatojumu, lai atvieglot datu pārvaldības elementu izvēli lietotnes īstenošanai.

3.1. Arhitektūras paņēmienu pārskats datu pārvaldībā

Pirms datu pārvaldības procesu aspektu definēšanas ir svarīgi izvēlēties metodes, kuras ļauj tiem procesiem piekļūt, kontrolēt un pielietot tos. Datu pārvaldības arhitektūras lēmumi mēdz definēt datu pārvaldības operāciju kvalitāti un pielāgojamības iespējas, jo tiek cieši saistīti ar konceptuāliem datu pārvaldības aspektiem un pielietošanas iespējamības aspektiem. Šī nodaļa apskatīs pamata arhitektūras iespējas, priekšrocības un trūkumus.

3.1.1. Datu pārvaldības interfeisi un to pielietojumu iespējas

Ir vairākas variācijas datu pārvaldības interfeisiem, tomēr tās ir cieši saistītas ar sistēmas piekļuves modeli. Populārākais variants ir klienta / servera modelis, kur dati tiek uzturēti servera pusē un piekļuvē tiek īstenota caur tīkla savienojumu, kura tehniskā realizācija ir atkarīga no procesu pārvaldības pieejas, kura tiks aplūkota šīs nodaļas ietvaros. Klienta / servera modelis ļauj pilnīgi izolēt lietotnes procesus no datu pārvaldības aspektiem resursu ziņā, jo datu pārvaldībai tiek lietoti atsevišķi serveri, t.i. atsevišķi CPU, RAM, cietā diska resursi. Trūkumi - klienta esamības problēma, jo lai lietotne spētu datu pārvaldīt šī modeļa ietvaros, ir nepieciešams definēt klientu un nodrošināt komunikāciju ar serveri, kas iekļauj sevī tīkla savienojuma nodrošināšanu un parasti interfeisa dublējumu. Ir nepieciešams klientu pieprasījumu interpretators, piemēram, SQL parsētājs servera pusē, kas būtībā dublēs datu pārvaldības iekšējo interfeisu, kas savukārt negatīvi ietekmēs

datu pārvaldības elastīgumu interfeisa jomā. Teorētiski ir iespējams klienta / servera komunikāciju padarīt dinamisku, t.i. klientam definēt interfeisu, kas pilnīgi sakrīt ar servera iekšējo datu pārvaldības interfeisu un pārsūtīt pieprasījumu ar minimālām validācijām un izmaiņām, tomēr praksē tāda pieeja tiek izmantota lielo datu apstrādes satvaros (piem. hadoop, spark), kas ir ārpus šī pētījuma sfēras. Relāciju datubāzēs parasti tiek definēta komunikācijas valoda (SQL, vai cits) un interpretēta servera pusē.

Alternatīva pieeja ir datu glabāšana lokālā datņu sistēmā, tad datu pārvaldībai un lietotnes vajadzībām tiks izmantoti kopējie procesi, tomēr dati būs tieši pieejami. Klienta / servera gadījumā, lai tiktu pie datiem, ir divi varianti: izmantot servera interfeisu, kas parasti iekļauj sevī lietotāju un sesiju pārvaldību, kā arī tehniskos ierobežojumus, vai pieslēgties pie servera un tikt pie datņu sistēmas. Lokālas krātuves gadījumā datnes jau no pašā sākuma ir pieejamas, tā veidojas trešais pārvaldības variants – piekļūt pie datiem programmatiski uzrakstot savas pārvaldības programmas vai modificēt jau esošo interfeisu un pielāgot pārvaldību saviem nolūkiem.

Pašam piekļuves interfeisam arī ir vairāki varianti: statiskas vai dinamiskas semantikas piekļuves programmēšanas valoda, kura tiek pārsūtīta un interpretēta starp lietotni un datu pārvaldība sistēmu, vai datu pārvaldības modulis kā daļa no lietotnes, kas ļauj piekļūt gan augstāka, gan zemāka līmeņa pārvaldības interfeisam. Abus variantus ir iespējams izmantot klienta / servera modelī un lokālas krātuves modelī, piemēram, MySQL serveris izmanto SQL valodu datu pārvaldībai, kas ir attālināta datu pārvaldība, un SQLite bibliotēka arī izmanto SQL valodu datu pārvaldībai, tomēr ir lokālais datu pārvaldības modulis. Interfeisa izvēle definē datu pārvaldības izolācijas līmeni no lietotnes un datu pārvaldības elastīguma.

3.1.2. Datu piekļuves kontrole

Datu piekļuves kontrole ir atkarīga no datu pārvaldības interfeisa izolācija līmeņa. Parasti, kad tiek izveidota atsevišķa lietotne datu pārvaldībai, piemēram, MySQL serveris, tad lietotnes un interfeisa ietvaros tiek definēta sesijas pārvaldība, un līdz ar to lietotāju pārvaldība. Tiek aprakstīti visi lietotāji, un piekļuves tiesības konkrētām datnēm. Tā pieeja ļauj kontrolēt gan programmas lietotājus, jo programmas klients, kas veido pieprasījumu, arī ir datu pārvaldības sistēmas lietotājs, gan datu pārvaldniekus (cilvēkus), kuri pieslēdzās pie datubāzes, izmantojot to pašu klienta / servera interfeisu un speciāli pielāgotu grafisko vidi.

Kad datu piekļuve nav pilnīgi izolēta, ir iespējams kontrolēt piekļuvi pie datnēm izmantojot vietējos piekļuves resursus, t.i. operētāj sistēmas datņu sistēmas operācijas. Parasti operētāj sistēmas atbalsta POSIX standartu, kur ir definētas lietotāju grupas un lietotāji, kā arī operāciju uz datnēm kontroles rīki. Šī kontrole nav ļoti detalizēta, salīdzinot ar klasisko SQL piekļuves kontroli, jo ļauj kontrolēt nevis konceptuālās operācijas, bet fiziskās operācijas ar datnēm, tomēr programmas līmenī šī kontrole ir pietiekama. Lokālai datu pārvaldības sistēmai arī ir iespējams definēt SQL (vai citu) valodu datu piekļuvei, līdz ar to ir iespējams arī ieviest papildus kontroles rīkus datu piekļuvei datu pārvaldniekiem (cilvēkiem).

3.1.3. Procesu organizācija datu pārvaldības sistēmā

Procesu organizācija ir atkarīga no sistēmas, kura ir atbildīga par datu pārvaldības procesiem. Gadījumā ar klienta / serveri modeli serveris ir izolēta instance ar atdalītiem resursiem un definētiem procesu pārvaldības metodēm. Ja datu pārvaldība tiek īstenota ar lokāliem rīkiem, tad atbildība par procesu pārvaldību tiek nodota lietotnei. Tieši lietotne, kura izmanto datu pārvaldības bibliotēku / moduli definē procesu organizāciju, ja vien datu pārvaldības modulim nav stingri definētas procesu saskaņošanas sistēmas.

Ir divi pamata veidi kā organizēt datu pārvaldības procesus: definēt vienu procesu, kura ietvaros notiks datu pārvaldība, vai veidot vairākus neatkarīgus procesus. Centralizēta procesu pārvaldība ļauj ciešāk kontrolēt servera resursus, kā arī piedāvā vairākas iespējas resursu atbrīvošanas prioritātes pārvaldībai, pamata procesa ietvaros. Papildus ir vieglāk pārbaudīt resursu izmantošanu, jo ir jāseko līdzi tikai vienam procesam. Tomēr, mainot datu pārvaldības konfigurācijas, parasti procesu ir jāpalaiž no jauna, papildus no programmēšanas viedokļa apakš procesu vai pavedien procesu pārvaldība nav triviāls uzdevums, it īpaši vienlaicīgu procesu pārvaldības nodrošināšanai. Alternatīva ir veidot atsevišķu procesu vienas datu pārvaldības operāciju virknes izpildei. Decentralizācijas pieeja ļauj elastīgāk pārvaldīt sistēmas konfigurācijas, jo katrs jauns process var daļu no resursiem inicializēt no jauna, tomēr kopēju resursu izmantošanai ir nepieciešami papildus rīki.

3.1.4. Metadatu novietojums datu pārvaldībai

Efektīvai informācijas izmantošanai ir nepieciešams glabāt un uzturēt metadatus. Relāciju datubāzēs metadati glabājas servera pusē un ir pieejami caur INFORMATION_SCHEMA datubāzi, kura piedāvā vairākus skatus uz datubāzē glabātajām struktūrām un procesiem. Šī pieeja piedāvā centralizētu metadatu glabāšanu un pārvaldības interfeisu. Tomēr praksē lietotnes īstenošanas laikā var tikt veidotas papildus metadatu uzturēšanas struktūras, piemēram, ORM, kura pieeja ir nolasīt datubāzes metadatus un ģenerēt programmas koda struktūras, caur kurām ir iespējams pārvaldīt datus. Šī pieeja satur vairākas problēmas, viena no kurām ir metadatu dublējums, jo tad ir nepieciešams uzturēt un pārvaldīt metadatus nevis vienā vietā, bet vairākās vietās. Migrācijām un citiem datu struktūru uzturēšanas procesiem ir nepieciešams ņemt vērā lietotnē glabātas reprezentācijas par pārvaldītiem datiem. ORM ir vēl viens metadatu pārvaldības un abstrakcijas slānis, kurš ievieš papildus loģiku metadatu procesos, tas nozīmē sarežģītāku datu konsistences nodrošinājumu lietotnes ietvaros, kā arī procesu pārvaldību. Lietotne un datu pārvaldības sistēma tiek uzturētas kā vienota lietotne, jo procesi tiek tuvāk piesaistīti viens otram, ieviešot papildus datu pārvaldības konstrukcijas.

Alternatīva ir uzturēt metadatus vienā vietā un nepieciešamības gadījumā dublēt informāciju tikai kešatmiņā, un saglabāt rādītāju metadatu atrašanās vietā, lai būtu iespējams atjaunot lokālo metadatu glabātuvi. Klienta / servera modelim šī pieeja nav efektīva, jo starp klientu un serveri parasti ir tīkla savienojuma slānis, tas nozīmē papildus resursu nepieciešamību metadatu nolasīšanai. Otrais variants ir glabāt metadatus kā daļu no lietotnes, un datu pārvaldības procesos izmantot lietotnes metadatu krātuvi. Šī pieeja ļauj optimāli uzturēt datus un nodrošināt metadatu esamību, tomēr pieprasa pēc iespējas lielāku integrāciju starp datu pārvaldības sistēmu un lietotni, ideālā gadījumā lietotne un datu pārvaldības procesi ir vienota sistēma.

3.1.5. Secinājumi par arhitektūras variācijām datu pārvaldībā

Datu pārvaldības arhitektūras jautājumu risinājums ir atkarīgs no lietotnes pieņēmumiem un resursu izvietošanas infrastruktūrā. Pamatjautājums ir – cik cieši datu pārvaldības sistēma tiek integrēta ar lietotnes sistēmu, no tā ir atkarīgi abu sistēmu komunikācijas interfeisi, piekļuves kontroles rīki, procesu organizācijas mehānismi un metadatu novietošanas un pārvaldības paņēmieni. Lietotnes un datu pārvaldības pilnas integrācijas gadījumā, ir iespējams novērst datu

pārvaldības procesu dublēšanu, tas ir iespējams arī atdalītu sistēmu arhitektūrā, bet parasti netiek īstenots.

Arhitektūras lēmumus iespējams kombinēt, un pat piedāvāt vairākas alternatīvas vienas sistēmas ietvaros, tomēr ir vairākas tendences, populārākas arhitektūras variācijas, kuras parasti nosaka datu pārvaldības sistēmas risinājumu izvēli. Tie ir relāciju datubāzes, kuri īsteno klienta / serveri modeļi un iebūvētas datubāzes, kuras piedāvā programmas koda bibliotēku datu pārvaldībai. Relāciju datubāzes ir monolītas sistēmas, kuras īsteno iepriekš definētas darbības ar datiem, fokusējoties uz datu konsistenci un procesu stabilitāti. Iebūvētas datubāzes īsteno gan augsta, gan zema līmeņa datu pārvaldības procesus, kas ļauj pielāgot risinājumu lietotnes vajadzībām.

3.2. Datu pārvaldības tehniskais īstenojums

Datu pārvaldības tehniskais īstenojums sākas no konceptuāla modeļa izvēles, t.i. kādā veidā dati tiks reprezentēti caur interfeisu. Populārākās variācijas ir relāciju modelis, kur dati tiks glabāti tabulu un kolonnu struktūrās, un dokumentu krātuve, kur dati tiks glabāti simbolu virknes formā, izmantojot XML, JSON, vai citu datu reprezentācijas formātu. Protams, tehniski gan datu uzturēšanas tehnikas, gan optimizācijas paņēmieni ir līdzīgi, tomēr jāsāk ar vienu reprezentācijas modeli, lai pakāpeniski izveidotu datu pārvaldības elementus. Šīs nodaļas mērķis ir definēt datu pārvaldības tehniskos aspektus un veidotu priekšstatu par datubāzes datu uzturēšanas procesiem.

3.2.1. Ieraksta koncepcija un īstenojuma piemēri

Ieraksti tiek glabāti bitu formātā, jo bitos dati aizņem mazāk vietas atmiņā vai cietā diskā, un ir papildus iespēja iepriekš definēt konkrētas bitu virknes datu tipus (integer, character, float). Ieraksta formāts parasti sastāv no iepriekš definētiem bitiem, kuri sastāda meta informāciju par konkrēto ierakstu, un bitu virknēm kuros ievieto ieraksta saturu. Vieglāk strādāt ar ierakstiem, kuros saturs ir ierobežots pēc garuma, vai citiem parametriem, jo ir iespējams ātrāk meklēt ierakstus failā, zinot katra ieraksta statisku un nemainīgu izmēru failā.

Datus failā var brīvi rakstīt, pārrakstīt un dzēst no faila beigām līdz konkrētai faila vietai, tas nozīmē, ka faila sākumā saglabātus ierakstus nav iespējams nodzēst, nemainot vēlāk saglabātus ierakstus. Datus rakstot, parasti jau iepriekš ir zināms ieraksta formāts un izmērs. Pieņemsim, ka

mums ir tabula ar trim laukiem: *id* ar tipu integer, *title* ar tipu simbolu virkne un maksimālu garumu 10 simboli, *price* ar tipu float. Kā piemēru varētu izmantot MySQL MyISAM krātuves programmas ieraksta formātu, t.i. viens bits, kurš definē vai ieraksts ir nodzēsts, trīs biti, pa vienam bitam uz katru ieraksta lauku, kuri definē vai laukā ir ierakstīts NULL tips (pieņemam ka tādu tipu mēs izmantojam), tad 4 baiti integer lauka tipam, 10 baiti simbolu virknei un 4 baiti float tipam. Viena ieraksta kopējais garums failā ir 1+3+4+10+4=22 baiti. Caur tabulām ar nemainīgu ieraksta garumu ir viegli veikt meklēšanas iterācijas vai meklēt konkrētus ierakstus. Piemēram, lai nolasītu trešo pēc kārtas ierakstu, ir jānolasa 22 baiti, sākot no 22*3=66 faila pozīcijas.

Ar ierakstiem, kuriem ir lauki ar dinamisku izmēru strādāt nedaudz sarežģītāk, papildus lauka saturam ir jāglabā satura garums, citādi programma nezinās, cik simbolu ir nepieciešams nolasīt, nezinās, kad apstādināt lasīšanu. Dinamiskus laukus ir iespējams glabāt kopā ar statiska izmēra laukiem, tomēr tad nebūs iespējams bez papildus metadatiem izrēķināt konkrēta ieraksta pozīciju. Alternatīvi ir iespējams dinamiskā izmēra laukus glabāt atsevišķā failā kopā ar katra ieraksta izmēru, un failā ar statiskiem laukiem saglabāt rādītāju uz otra faila ieraksta atrašanās vietu.

3.2.2. Ieraksta lauku izmaiņas un datnes uzturēšana

Statiska izmēra laukiem mainīt vērtību ir iespējams vienkārši pārrakstot lauka bitus. Simbolu virknes lauku vērtību maiņa nedaudz atšķiras no citiem tipiem, jo simbolu virknes lauks ar nemainīgo izmēru ir izveidots, aizpildot trūkstošus simbolus ar tukšiem simboliem. Piemēram, ir lauks ar 10 simboliem, tiek aizpildīts ar 6 simbolu vārdu, tad papildus 4 simboli tiek aizpildīti ar tukšiem simboliem, ar mērķi saglabāt katra lauka garumu. Relāciju datubāzēs SQL valodā šīs lauku tips tiek nosaukts par CHAR, un ir optimāls no datu operāciju viedokļa, jo palīdz uzturēt statiska izmēra ieraksta struktūru.

Mainīt lauka ar dinamisku garumu vērtību ar vienu operāciju neizdosies, jo lauka ieraksta vērtība ir glabāta starp citiem ierakstiem, kas padara neiespējamu vērtības garuma izmaiņas bez kaimiņ ierakstu pārrakstīšanas. Ir vairākas pieejas šīs operācijas īstenojumam, viena no tām ir, dinamiska lauka vērtībai mainoties, dzēst veco ierakstu un ierakstīt jauno ierakstu ar jauno vērtību. Ieraksta dzēšana notiks ar dzēsta ieraksta indikatora maiņu, un jaunais ieraksts tiks pievienots datnes beigās. Papildus gadījumā, kad jaunās vērtības garums ir mazāks par tagadējo garumu, ir iespējams ierakstīt jauno vērtību un atlikumu aizpildīt ar tukšiem simboliem.

Ar laiku vienas datnes ietvaros vairāki ieraksti tiek dzēsti un pārrakstīti, līdz ar to faila izmērs pieaug un satur lieku informāciju, piemēram, nodzēstus ierakstus vai tukšus simbolus. Lai optimizētu vietas izmantošanu tiek izmantota daļējas datnes pārrakstīšanas metode. Datne tiek sadalīta fiksēta garuma segmentos, jaunie ieraksti tiek pievienoti segmentos. Kad viena segmentā ietvaros dati tiek glabāti neoptimāli, to var optimizēt. Visus ierakstus saglabāt pēc kārtas, fiziski nodzēšot atzīmētus kā nodzēstus ierakstus, atbrīvojot vietu jaunas informācijas ierakstīšanai. Tos segmentus sauc par lappusēm, un aktīvi izmantot relāciju datubāzes īstenošanā. Papildus segmentiem ir iespējams pierakstīt metadatus, kur nodrošina vieglāku datu lasīšanu, uzturēšanu un integritāti. Pēc izmēra lappuses parasti ir tik lielas, cik atļauj RAM un CPU resursi, jo lappuse var tikt optimizēta datu pārvaldības operācijas laikā, tas nozīmē, ka optimizācijas procesam ir nepieciešams būt pietiekami ātram.

Lai vieglāk būtu sekot datnes ierakstiem, ir iespējams uzturēt papildus datnes ar metadatiem par ierakstiem. Pieņemsim, ka mums ir datne ar vairākiem ierakstiem un daļa no tiem tiek dzēsta. Programma varētu saglabāt nodzēstu ierakstu lokācijas atsevišķā datnē, un, kad atnāk jauni ieraksti, tos pievienot nevis datnes beigās, bet gan pārrakstot nodzēstus ierakstus. Tā pieeja ļauj minimizēt nepieciešamību pēc pilnas vai daļējas datnes restrukturēšanas, jo tas notiks dinamiski.

Ar mērķi optimizēt konkrētu ierakstu meklēšanu datnēs pēc parametriem tiek izmantotas indeksācijas struktūras, tomēr to struktūru īstenojums ir ļoti atkarīgs no datnes struktūras. Indeksācijas struktūras parasti tiek glabātas atsevišķās datnēs un izmantotas kopā ar oriģināliem ierakstiem. Indeksu izveidošanai tiek paņemti klasiskie indeksu algoritmi un pielāgoti konkrētiem specialiem gadījumiem. Parasti indeksa datnes struktūra un principi ir līdzīgi oriģinālu ierakstu datnes struktūrai un principiem, jo tas padara vieglāku kopēju izmantošanu un uzturēšanu.

3.2.3. Atomiskās operācijas un transakciju atbalsts

Vienlaicīgu procesu vidē ir nepieciešams nodrošināt atomiskās operācijas uz datnēm. Parasti tiek izmantoti vairāki datnes informācijas slēgšanas paņēmieni, ar mērķi kontrolēt datnes pārvaldības procesus un novērst datu korupcijas notikumus. Kļūdas operācijas ar datiem var parādīties, kad vairāki procesi konfliktē vienlaicīgu operāciju izpildē, piemēram, divi vienlaicīgi procesi raksta vienā lokācijā dažādus bitus, tad viens process pārrakstīs otra procesa bitus un katra bita gadījumā tas var būt cits process, tā var tikt saglabāts ieraksts, kurš netika apzināti ierakstīts.

Vienlaicīgu procesu datu pārvaldības organizācijai ir vairāki pamata principi, kuri nodrošina datu kļūdu novēršanu:

- Vienlaicīgi konkrētā vietā datnē ir tiesības rakstīt tikai vienam procesam
- Datu rakstīšana ir atomiskā, t.i., kamēr ieraksta rakstīšana nav pabeigta, citiem procesiem nav tiesību lasīt to ierakstu
- Datu lasīšana ir atomiskā, t.i. ierakstu nolasīšanas laikā citiem procesiem nav tiesību mainīt to ierakstu

Tehniski vienkāršākais veids kā nodrošināt atomiskās operācijas ir izmantot lasīšanas / rakstīšanas slēgšanu datnes līmenī. Datne var tikt slēgta lasīšanai vai rakstīšanai. Kad datne tiek slēgta lasīšanai, tad vairākiem procesiem ir tiesības lasīt ierakstus no datnes tomēr nevienam procesam nav tiesību mainīt datni. Kad datne tiek slēgta rakstīšanai, tad viens process spēj mainīt datni un citi procesi (gan lasīšanai, gan rakstīšanai) gaida, kamēr pirmais process pabeigs savas darbības. Šīs pieejas galvenās priekšrocības ir tā, ka tā ir vienkāršā izprašanai un operāciju rezultātu ir viegli paredzēt.

Tomēr ir vairākas nianses, kuras padara šo pieeju neefektīvu lielas slodzes lietotnēm. Strupsaķeres izveidošana ir triviāla, piemēram, ir divas datnes (tabulas) A un B, gadījuma gan viens process slēdz A un tad mēģina slēgt B, bet tajā pašā laikā otrs process slēdz tabulu B un mēģina slēgt tabulu A notiek strupsaķere. Šo gadījumu var likvidēt tikai kontrolējot datnes slēgšanas secību visā sistēma, t.i. lai visi procesi slēgtu un atbrīvotu datnes vienā kārtība. Otrkārt, datnes slēgšanas pieeja padara neefektīvu vienlaicīgo procesu izmantošanu, jo teorētiski slēdz pieeju savstarpēji izslēdzošām vienlaicīgām operācijām, piemēram, kad vienam procesam ir nepieciešams mainīt ierakstu datnē un otram nolasīt citu ierakstu, tad teorētiski šie divi procesi nepārklājās, tomēr vienkāršas slēgšanas modeļa dēļ operācijas tiek palaistas secīgi.

Ir iespējams īstenot lappuses vai ierakstu līmeņa slēgšanas mehānismus. Ierakstu līmeņu slēgšana reti tiek izmantota, jo tās pārvaldībai ir nepieciešams vairāk resursu, salīdzinot ar pārējām metodēm. Lappuses līmeņa slēgšana strādā līdzīgi datnes slēgšanai, tomēr slēgšanas mehānismus jāīsteno programmas ietvaros, jo nav gatavu protokolu daļējai datnes slēgšanai. Lai īstenot lappuses līmeņa slēgšanas metodi ir nepieciešams uzturēt lappuses struktūras datnēs un uzturēt slēgšanas indikatoru sadalītā atmiņā vai kopējās metadatu datnēs. Uz slēgtām lappusēm ir jāsatur informāciju par slēgšanas metodi un procesu, kurš izpilda operācijas, papildus ir nepieciešams uzturēt procesu, kurš seko līdzi strupsaķerēm un laicīgi neatbrīvotam lappusēm.

Transakciju īstenojums parasti ir esošas krātuves paplašinājums, izmantojot buferizācijas mehānisma pieeju. Transakcijas sākumā tiek izveidots jaunais ieraksts ar transakcijas informāciju un izveidots jaunais krātuves segments darbam ar datiem, kas var atrasties atmiņā vai arī transakciju datnēs. Atkarībā no izvēlētā transakcijas tipa ir iespējams īstenot vairākus datu nolasīšanas mehānismus. Piemēram, ja transakcijas tips pieļauj "dirty read" fenomenu, tad transakciju krātuves ietvaros procesiem ir atļauts nolasīt datus pirms tie ir pievienoti pamata datu krātuvei, bet pretējā gadījumā dati tiek nolasīti tikai no konkrēta transakcijas datnes segmenta un pamata datnes krātuves, kurai dati tiek pievienoti pēc transakciju akceptēšanas. Datu izmaiņu gadījumā transakcijas datnei tiek pievienots mainīts ieraksts un pēc transakcijas akceptēšanas ieraksts tiek pārkopēts uz pamata krātuvi, pārrakstot iepriekšēju ieraksta versiju. Tas pats mehānisms var tikt izmantots ierakstu dzēšanai. Šos procesus var optimizēt, izmantojot papildus metadatu datnes.

Ir alternatīva pieeja transakciju īstenošanai, kura lieto uzticamības nodrošināšanas rīkus lai nodrošināt transakciju atrites. Ieraksti tiek mainīti galvenā ierakstu krātuvē, tomēr, kad atnāk atrites komanda, vecas ierakstu versijas tiek nolasītas no datu pārvaldības žurnāliem un atjaunināti krātuvē. Ar šo pieeju ir iespējams īstenot transakcijas, tomēr tā piedāvā mazāk elastīguma transakciju tipu kombinēšanai.

3.2.4. Datu pārvaldības atjaunošana un rezerves kopijas

Datu atjaunošana var tikt nepieciešama vairākos gadījumos, tomēr galvenais no tiem ir datnes korupcija. Datne var tikt korumpēta nepabeigtas pierakstīšanas operācijas vai arī vienlaicīgiem procesiem mainot vienādu ierakstu. Jo sarežģītāki datu pārvaldības paņēmieni tiek izmantoti, jo vairāk iespējamu datnes korupcijas variantu, piemēram, ja tiek atbalstītas transakcijas, tad, rakstot vairākas izmaiņas no transakciju datnes uz krātuvi, daļa no datiem var tikt korumpēta.

Ir viens pamata datu atjaunošanas paņēmiens – ja dati netika korekti saglabāti / mainīti, t.i. ja datu izmaiņas operācija netika korekti pabeigta, tad komandu ir nepieciešams atcelt, un, ja tas ir iespējams, pamēģināt operāciju atkārtot. Vienkāršākā gadījuma nepareizi pierakstītu ierakstus var atrast pēc ieraksta garuma, kad garums ir statisks. Piemēram, programma apzinās ka datnes ieraksta garum ir 22 baiti, tomēr datnes pilns izmērs ir 50 baiti, tas nozīmē, ka datnē ir vismaz viens nepareizi pievienots ieraksts, un tas parasti ir pēdējais ieraksts. Sarežģītāku algoritmu atjaunošanas nodrošināšanai tiek rakstīti datnes izmaiņu žurnāl faili. Populārākais veids žurnāl failu izveidošanai

ir pierakstīt žurnālā operācijas un papildus informāciju pirms palaist operācijas uz galvenās datnes. Šī pieeja ļauj ļoti efektīvi atjaunot trūkstošus datus, jo pēc būtības satur visas operācijas uz datiem un to rezultātus. Žurnāl failus ir viegli uzturēt kad ir kopējais operāciju formāts, piemēram, SQL, gadījumā, kad tas ir atbalstīts, vai lokāli izveidots formāts, kad SQL vai cits standarts nav atbalstīts.

Rezerves kopijas veidošanu var īstenot lietojot žurnāl failus, jo tas datus nav nepieciešams kopēt no galvenās datnes, kura tiek aktīvi izmantota ikdienu datu pārvaldībā, bet gan inkrementālus datnes izmaiņu žurnālus. Gadījumā, kad ir nepieciešams izveidot rezerves kopiju no galvenās datnes, parādās datu konsistences problēma. Kad ir vairākas saistītas datnes (tabulas), tad rezerves kopijas veidošanas laikā nedrīkst datnes mainīt, citādi rezerves kopijas dati nebūs konsekventi savā starpā. Tomēr ir vairākas sarežģītākas pieejas rezerves kopijas izveidošanai, viens no tiem kopēt datus, izmantojot lappuses un ļaut mainīt tās lappuses, kuras jau tika nokopētas. Tad, protams, vajag uzturēt metadatus par nokopētām lappusēm un koordinēt lappušu izmaiņas.

3.2.5. Secinājumi par datu pārvaldības tehniskajiem aspektiem

Datu pārvaldības īstenošanas tehniskus paņēmienus var sadalīt divās grupās – tie, kuri nodrošina pamata funkcionalitāti, t.i. darbs ar datni un ierakstu reprezentāciju, un tie, kuri ir atkarīgi no datnes formāta. Vienkāršās datu pārvaldības sistēmas īstenošanai ir iespējams lietot pamata funkcionalitāti darbam ar datnēm, kas nesīs paredzamus datu pārvaldības operāciju rezultātus un iespējamas problēmas. Sarežģītos gadījumos ir jāīsteno pielāgotus ierakstu slēgšanas moduļus, ka arī vairāku datņu integrāciju vienotā datu pārvaldības procesā.

Datu pārvaldības paņēmienus pārskatot, skaidri redzamas sistēmas modularitātes un pakāpeniskās funkcionalitātes uzkrāšanas iespējas. Tehniski vairākas sarežģītākas iespējas definē ieraksta formāts un datnes organizācija. Indeksi, transakcijas un datnes pārvaldības optimizācijas ir papildus funkcionalitāte, kura balstās uz ieraksta formāta atbalsta un evolūcijas. Papildus ir iespējams sadalīt datnes pārvaldības paņēmienus vairākos moduļos un slāņos, kas piedāvātu vairākus zemākā līmeņa bibliotēkas, kuras īsteno vairākus ierakstu formātus, vai arī datnes uzturēšanas paņēmienus, kurus varētu kombinēt konceptualizējot sistēmas interfeisu augstākajā līmenī. Datu pārvaldībā ir atrodamas komponent bāzētās pieejas paradigmas un tendences.

4. DATU PĀRVALDĪBAS KOMPONENTU IZSTRĀDE UN INFORMĀCIJAS SISTĒMAS IZVEIDOJUMS

Šī nodaļa aprakstīs datu pārvaldības komponentu izstrādes procesu un iegūtos rezultātus. Papildus tiks apskatītas priekšrocības un trūkumi izveidotai bibliotēkai un pieejai kopumā. Datu pārvaldības bibliotēku izmantojot tika izveidota lietotne kura īsteno uzdevuma saraksta pārvaldību tīmekļa vietnē. Lietotnes frontend daļa tika paņemta no "todomvc.com" atvērtā koda apmācības resursa. Nodaļas mērķis ir izmantojot apkopotu informāciju no iepriekšējas nodaļas izpētīt komponent bāzētās pieejas praktisko daļu datu pārvaldībā īstenošanai.

4.1. Izstrādes vides apraksts un arhitektūras pieņēmumi

Izstrādes videi tika izmantota Ubuntu 16.04 LTE operētājsistēma un Apache2 tīmekļa serveris, jo tas tehnoloģijas ir labi pielāgotas tīmekļa vietnes uzturēšanai. Tika izmantota datnes sistēma, kas tika uzstādītā pēc noklusējumф – ext4. Par programmēšanas valodu tika izmantota augsta līmeņa valoda – PHP, kura arī ir pielāgota tīmekļa lietotnes izstrādei. Vides uzstādīšanai un uzturēšanai tika izmantota Vagrant lietotne, kura ir virtuālas vides konfigurāciju automatizācijas rīks.

Datu pārvaldības bibliotēka ir pieejama lietotnē kā datu pārvaldības modulis un tiek uzskatīta kā bezservera, jo datu pārvaldībai netiek izdalīts centrālais process. PHP valoda ir dinamiski tipizēta skriptu valoda un ir labi integrēta ar Apache2. Tīmekļa vietnes pieprasījumi tiek veidoti kā jaunie procesi, tas nozīmē procesu decentralizācija ir galvenais veids informācijas pārvaldībai, ko arī izmantos datu pārvaldības bibliotēka. Abi lietotne un datu pārvaldība tika rakstītas PHP valodā, kas padara PHP programmas interfeisu par labāku integrācijas variantu. Datu piekļuves kontrole tika īstenota izmantojot operētājsistēmas līdzekļus, jo datu pārvaldībai būs tikai viens lietotājs – lietotnes programmas kods.

Ar mērķi īstenot labas kvalitātes bibliotēku tika izmantoti sekojošie papildus rīki:

- composer– bibliotēkas pakešu pārvaldībai
- symfony/options-resolver – iekšējo konfigurāciju pārvaldībai
- symfony/console – papildus termināla komandu īstenošanai
- phpunit/phpunit – vienību testu īstenošanai

4.2. Datu pārvaldības bibliotēkas izstrāde

Datu pārvaldības bibliotēka tika izstrādāta, izmantojot MIT atvērtā koda licenci un ir ievietota atvērtai pieejai tīmekļa vietnē GitHub - saite https://github.com/bozerkins/dmc. Izstrādes progresu ir iespējams apskatīt koda nodošanas žurnālā, kur papildus ir atrodami vēsturiskie eksperimenti ar interfeisu un bibliotēkas darbību. Šīs nodaļas mērķis ir aprakstīt bibliotēkas izstrādes procesu, īstenošanas problēmas un to risinājumus, kā arī izpētīt pielietošanas iespējas tāda tipa bibliotēkai. Šīs nodaļas novitāte ir pats fakts datu pārvaldības bibliotēkas īstenošanai PHP valodā, jo parasti tiek izvēlētas kompilējamas valodas, ņemot vērā koda ātrdarbību.

4.2.1. Izstrādes plāns un bibliotēkas pamata funkcionalitāte

Bibliotēkas mērķis ir piedāvāt vairākas variācijas datu pārvaldībai, protams, tiks īstenots entītiju relāciju modelis, tomēr papildus tiks atbalstīts arī analītiskā tipa modelis statistikas apstrādei. Galvenais pieņēmums šīs bibliotēkai izstrādei ir – bibliotēka būs pielāgota lietošanai produkcijas vidē. To ņemot vērā tika saplānota šāda funkcionalitāte:

- entītiju relāciju modeļa pamata īstenojums
- entītiju relāciju modeļa ACID principu īstenojums
- kolonnas analītiskā modeļa īstenojums
- kolonnu saspiešana
- kolonnas analītiskā modeļa BASE principu īstenojums
- tabulas izveide
- kolonnas definēšanas iespējas
- integer, float un string tipu atbalsts (statiska garuma)
- relāciju atbalsts
- viena indeksu tipa atbalsts
- CRUD operācijas

Šo bibliotēku nebūs iespējams izmantot lielas slodzes lietotnes datu pārvaldībai, tomēr tā būs pietiekama funkcionalitātes vidējas slodzes (10-20 vienlaicīgas piekļuves pie vienas datnes) lietotnes korektai operēšanai.

4.2.2. Metadatu glabāšana un tabulas struktūras definēšanas interfeiss

Datu struktūru metadati tiek glabāti iekš lietotnes programmas koda. Katrai tabulai tiek definēta datne ar instrukciju, kura apraksta, kādas kolonnas ir definētas tabulā un kāda tipa tās ir, kā arī kolonnu nosaukums un izmērs. Ir jādefinē izmērs, integer un float datu tipi, un tas, cik baitu aizņems lauks. Piemēram, integer ar garumu 1 baits būs skaitlis intervālā no -128 līdz 127, divu baitu integer būs intervālā no -32,768 līdz 32,767, utt. Simbolu virknes tipam tiek izmantots papildus baits, lai apzīmētu virknes beigas, kas nozīmē, ka, definējot simbolu virkni ar garumu 100 simboli, ir pieejami 99 simboli un 1 tiek rezervēts sistēmā.

Tika īstenoti divi datu pārvaldības konceptuālie modeļi, tomēr abiem pamata struktūra ir tabula ar kolonnām. Tabulas interfeiss ir līdzīgs abām pieejām, un nozīmīgi atšķiras tikai fiziskajā īstenošanas līmenī. Entītiju relāciju modelim tabulas dati tiek glabāti vienā failā, bet analītiskā moduļa dati tiek glabāti vienā mapē, katrai kolonnai pienākas sava failu struktūra, papildus kolonnām tika ieviesta sadalīšanas sistēma statistikas piegādes optimizācijai.

```php
return [
    # location of the table file
    'location' => '~/storage/my-table',
    # table structure
    'structure' => [
        array (
            'name' => 'ID',
            'type' => 1,   # integer
            'size' => 4,   # size of integer
        ),
        array (
            'name' => 'Profit',
            'type' => 2,   # decimal
            'size' => 8,   # size of decimal
        ),
        array (
            'name' => 'ProductTypeReference',
            'type' => 1,   # integer
            'size' => 4,   # size of integer
        ),
        array (
            'name' => 'ProductTitle',
            'type' => 3,   # string
            'size' => 100, # size of string
        ),
    ]
];
```

att. 4.2.2.1. **Tabulas struktūras metadatu glabāšanas piemērs**

Tabulas definēšanas formāts ir PHP vārdnīcas (masīva) struktūra ar diviem parametriem – lokācija un struktūra. Lokācijā entītiju relāciju modelī tiek definēta datnes lokācija, ja datne neatrodas lokācijā, tad tā tiks automātiski izveidota pēc tabulas objekta inicializācijas.

Tabulas struktūras metadatu piemērs ir atrodams attēlā 4.2.2.1., kur tiek definēta tabula my-table, kura atrodas mapē storage lietotāja mājas mapē (/home/<user>/storage/). Piemērā tiek definētas četras kolonnas – ID ar tipu integer, Profit ar tipu decimal, ProductTypeReference ar tipu integer un ProductTitle ar tipu string un izmēru 100 simboli (no kuriem datu ierakstīšanai ir pieejami ir 99).

```
return [
    'location' => '~/storage/my-table',
    'structure' => [
            array (
                'name' => 'ID',
                'type' => 1,
                'size' => 4,
                'partition' => 0
            ),
            array (
                'name' => 'Date',
                'type' => 3,
                'size' => 11,
                'partition' => 1
            ),
    ]
];
```

att. 4.2.2.2. **Tabulas struktūras metadatu glabāšanas piemērs analītiskā modulī**

Analītiskā modeļa tabulas definējums ir līdzīgs, tomēr papildus ir nepieciešams definēt sadalījuma kolonnu, jo pēc tam tas tiks fiziski sadalīts kolonnu datnēs. Tabulas definējuma piemērs ir redzams attēlā 4.2.2.2.Tabulas mape atrodas lietotāja storage mapē un satur datnes par divām kolonām – ID ar tipu integer un Date ar tipu string un izmēru 11. Date kolona tiks izmantota datu sadalījumam.

```
use DataManagement\Model\EntityRelationship\Table;
use DataManagement\Model\TableHelper;

$table = new Table();
$table->addColumn('ID', TableHelper::COLUMN_TYPE_INTEGER);
$table->addColumn('Profit', TableHelper::COLUMN_TYPE_FLOAT);
$table->addColumn('ProductType', TableHelper::COLUMN_TYPE_INTEGER);
$table->addColumn('ProductTypeReference', TableHelper::COLUMN_TYPE_INTEGER);
$table->addColumn('ProductTitle', TableHelper::COLUMN_TYPE_STRING, 100);

$structure = var_export($table->structure(), true);
```

att. 4.2.2.3. **Tabulas struktūras definējums programmas kodā**

Tabulas struktūras tiek definētas, izmantojot Tabulas interfeisu, ir divas metodes tabulu struktūras izveidošanai: 1) inicializēt tabulas objektu, izmantojot tabulas struktūras instrukcijas, 2)

eksportēt tabulas struktūras instrukcijas mainīgos no jau definēta objekta. Attēlā 4.2.2.3. var redzēt jaunas tabulas struktūras izveidošanu un eksportu. Izmantojot tabulas metodi addColumn tiek izveidotas 5 kolonnas, kuras tiek piesaistītas pie tabulas struktūras. Struktūra tiek eksportēta izmantojot structure metodi un iebūvētu PHP valodā var_export funkciju, kura konvertē PHP mainīgu programmas kodā. Tas ļauj automatizēt tabulu izveidošanu un vienkāršot tabulas objektu inicializāciju no metadatiem. Skripts tiek izveidots kā jauns tabulas objekts, tiek pievienoti vairāki lauki, izmantojot tabulas kolonnas tipu konstantes un struktūra tiek eksportēta instrukciju datnē. Pilns tabulas metadatu struktūras izveidošanas piemēra skripts ir atrodams pielikumā 7.

```php
# ER Table
$instructionFileDestinationER = '/project-root/my-table-instruction-er.php';
$erTable = \DataManagement\Model\EntityRelationship\Table::newFromInstructionsFile($instructionFileDestinationER);

# Columnar table
$instructionFileDestinationColumnar = '/project-root/my-table-instruction-columnar.php';
$columnarTable = \DataManagement\Model\Columnar\Table::newFromInstructionsFile($instructionFileDestinationColumnar);
```

att. 4.2.2.4. **Tabulas struktūras ielādē no instrukciju datnes**

To datnēs var vēlāk izmantot, lai inicializētu tabulas objektu un turpinātu ar to strādāt, piemēru ir iespējams atrast attēlā 4.2.24. Tiek nolasītas instrukcijas no datnes, kura atrodas "/project-root/my-table-instruction-er.php" un ir PHP skripts, kurš inicializē PHP valodā struktūru un atgriež to kā skripta iekļaušanas rezultātu. Tabulas struktūras pašlaik nav iespējams mainīt, ir jāraksta atsevišķs migrācijas skripts, kas izveidos jaunu tabulu un eksportēs tajā datus no vecās tabulas.

4.2.3. ACID / BASE principu nodrošinājums

Lai īstenotu entītiju relāciju modeli, ir nepieciešams nodrošināt ACID principus, ar mērķi piedāvāt atomiskās operācijas bibliotēkas ietvaros. Tabulu operācijām tiek izmantota PHP valodā iebūvētā datnes mīkstā (advisory) slēgšana, kura nodrošina sadalīto datnes bloku datu nolasīšanu un ekskluzīvu bloku datnes maiņu. Tas nozīmē, ka datnes netiek aizsargātas ar blokiem ārpus lietotnes procesiem, tomēr procesa ietvaros tiks nodrošināti ACID principi, izmantojot blokošanas funkcionalitāti.

BASE principi tiek atbalstīti analītiskā modeļa īstenojumā. Datnes pārvaldība notiek tādā veidā, ka datnes rakstīšanas un lasīšanas operācijas nepārklājas, tas nozīmē optimālas operācijas vienlaicīgu procesu vidē. Katra rakstīšanas operācija izveido atsevišķus failus, kuri tiek saglabāti tabulas mapē. Faili tiek pieejami, kad speciālais process ievietos to informāciju pamata krātuvē, tas

nozīmē, ka informācija nav pieejama uzreiz, bet gan tiek sagatavota un optimizēta analītisku pieprasījumu veikšanai.

4.2.4. CRUD operācijas entītiju relāciju modelī un ieraksta struktūra

Entītiju relāciju tabulas tiek veidotas ar vienkāršāku iespējamu struktūru, kas atbalsta datu dzēšanu. Tiek atbalstīti tikai fiksēta izmēra datu tipi. Katra tabula sastāv no ierakstiem, katram ierakstam ir atvēlēts viens bits statusa vērtībai, iespējamie ieraksta statusi ir aktīvs vai nodzēsts. Aktīvie ieraksti piedalās datu pārvaldībā, nodzēstie ieraksti nav fiziski nodzēsti, tomēr nepiedalās datu pārvaldībā un tiek likvidēti, izmantojot atsevišķus procesus. Ieraksts ir reprezentēts kā PHP valodas vārdnīca, kurā atslēgas ir kolonnu nosaukumi un vērtības ir ieraksta atbilstošas vērtības.

Tabulas objekts īsteno CRUD operācijas caur četrām metodēm: create, read. update, delete. Ieraksta izveidošanas metode pieņem ierakstu kā pirmo argumentu un pievieno sagatavotu ieraksta bitu virkni datnes beigās. Ieraksta pievienošanas piemērs ir atrodams attēlā 4.2.3.1. Tiek izveidots tabulas objekts to "/project-root/my-table-instruction.php" tabulas struktūras un izsaukta create metode, kurai tiek padots produkta ieraksts. Pārējām operācijām ir nepieciešama ierakstu meklēšanas un maiņas funkcionalitāte, kura tiek īstenota, izmantojot atzvanīšanas funkcijas.

```php
use DataManagement\Model\EntityRelationship\Table;

$instructionFile = '/project-root/my-table-instruction.php';
$table = Table::newFromInstructionsFile($instructionFile);
# create
$table->create([
    'ID' => 1,
    'Profit' => 2.11,
    'ProductTypeReference' => 0,
    'ProductTitle' => 'My first product, yay!'
]);
```

att. 4.2.3.1. **Ieraksta pievienojums datnei**

Meklēšana notiek, izmantojot iekšēju iterācijas konstrukciju, kura secīgi nolasa datni un izsauc atzvanīšanas funkciju uz katru ierakstu, padodot ierakstu kā funkcijas parametru. Lietotājs definē atzvanīšanas funkcijas dinamiski, un tās mēdz atgriezt speciālas konstantes, kuras kontrolē iterācijas procesu.

Meklēšanas piemērs ir atrodams attēlā 4.2.3.2. Pirmais read metodes izsaukums atgriež visus ierakstus no tabulas. Otrais read metodes izsaukums atgriež tikai ierakstus, kuriem ID lauks ir 5.

```php
# read all the table
$result = $table->read(function(){
    return Table::OPERATION_READ_INCLUDE;
});
# read the record with specific ID
$result = $table->read(function(array $record){
    if ($record['ID'] === 5) {
        return Table::OPERATION_READ_INCLUDE_AND_STOP;
    }
});
```

att. 4.2.3.2. **Ieraksta no datnes nolasīšanas piemēri**

Katrai operācijai (update, read, delete) meklēšanas daļa pieņem vairākas komandas: iekļaut tagadēju ierakstu rezultējošām masīvām, iekļaut ierakstu un pabeigt iterācijas procesu, vienkārši pabeigt iterācijas procesu. Datu izmainīšanas metode pieņem otru argumentu, kas arī ir atzvanīšanas funkcija, kura atgriež laukus, kurus ir nepieciešams mainīt un jaunās vērtības. Dzēšanas metode deaktivizē rezultējošus ierakstus datnē. Operāciju sintakse ir pieejama pielikumā 8.

Katra operācija automātiski izmanto atbilstošu datnes bloka operāciju, tomēr to ir iespējams definēt arī manuāli. Piemēram, ja ir zināms, ka tiks veiktas datnes nolasīšanas un maiņas operācijas, ir iespējams bloķēt datni uzreiz lasīšanai un rakstīšanai. Tabulā tiek definēti trīs statusi, kuri paraksta pieejamos bloka veidus: rakstīšanai, lasīšanai, rakstīšanai un lasīšanai. Gadījumā, kad datne tika nobloķēta lasīšanai, bet kods mēģina datni rediģēt, tik izmesta kļūda un process izslēgsies, automātiski atbrīvojot datni.

4.2.5. Datnes iterāciju modulis, zemāka līmeņa ierakstu pārvaldība

Tika izveidota papildus programmas struktūra, kura īsteno CRUD operācijas zemākajā līmenī, izmantojot iterāciju pieeju. Tabulas iterāciju klase īsteno interfeisu, kurš atļauj veikt operācijas ar ierakstiem, izmantojot rādītāju, kurš pēc savas būtības ir līdzīgs vienkāršai failu pārvaldībai. Šī klase atšķiras tikai ar to, ka mazāka pārvaldības vērtība ir nevis viens simbols, bet viens ieraksts. Interfeiss īsteno šādas operācijas ar datni: rādītāja pārvietošana, kur viena iterācijas vērtība ir ieraksts, ir iespējams pievienot jaunu ierakstu vai pārrakstīt eksistējošo ierakstu (kad rādītājs nav novietots uz datnes beigām), ir iespējams nodzēst ierakstus vai arī nolasīt tos.

Piemēram dzēšanas operācija ar iterāciju ir atrodama attēlā 4.2.5.1. Tabula tiek rezervēta rakstīšanai, tad tiek izveidots iterator objekts, kurš pārvieto rādītāju uz 5-ta ieraksta pozīciju un nodzēš to ierakstu, pēc tam tabula tiek atbrīvota.

```
# reserve the table
$table->reserve(Table::RESERVE_WRITE);
# get iterator
$iterator = $table->newIterator();
# jump to the record index
$iterator->jump(5);
# update record
$iterator->delete();
# release the table
$table->release();
```

att. 4.2.5.1. **Zemāka līmeņa ierakstu dzēšanas piemērs**

Pēc katras operācijas objekts palielina rādītāju uz vienu vērtību, tas tika izdarīts, lai nodrošinātu operāciju uzvedības konsistenci. Šī klase tiek izmantota Tabulas klasē, un visas operācijas notiek caur iterācijām, tomēr iterācijas ir iespējams izmantot arī atsevišķi, tomēr tās neīsteno datnes slēgšanas funkcionalitāti, tomēr to pieprasa. Papildus iterāciju objekts tiek padots kā papildus arguments atzvanīšanas funkciju konstrukcijās, lai piedāvātu lietotājam vairāk kontroles iterāciju laikā. Operāciju piemēri ir pieejami pielikumā 9.

4.2.6. Entītiju relāciju modeļa relāciju īstenojums

Tabulas iterāciju modulis ir izdevīgs, jo ļauj tieši piekļūt un strādāt ar ieraksta pozīciju datnē. Tabulu relāciju funkcionalitāte ir ļoti vienkārša, jo ir iespējama, izmantojot jau esošus interfeisus, bez papildus kodēšanas.

Ieraksta relācijas pievienošanai netiek izmantots unikāla identifikatora lauks, jo tas nav nepieciešams, ieraksta pozīcija jau ir unikāla un ir pieejama datnes iterācijas procesā. Lai nolasītu ieraksta unikālo rādītāju iterācijas laikā, var vienkārši pieprasīt iterācijas pozīciju kontroles objektam. Pēc tam to var saglabāt integer tipa laukā pamata tabulā, jo rādītājs datnē ir integer tipa cipars.

Kad ir jānolasa dati no abām tabulām, var izmantot tabulas iterācijas objektu, norādot konkrētu nolasīšanas vietu papildus tabulā un nolasot ierakstu. To var tikt izdarīts izmantojot vienu (!) nolasīšanas iterāciju, bez papildus indeksācijas konstrukcijām.

```
# read the record
$record = $table->read(function($record) {
    if ($record['ID'] === 5) {
        return Table::OPERATION_READ_INCLUDE_AND_STOP;
    }
})[0];

# read the related record with 1 (!) operation
$iterator = $tableRelation->newIterator();
$iterator->jump($record['ProductTypeReference']);
$productTypeRecord = $iterator->read();
```

att. 4.2.6.1. **Relāciju īstenojuma piemērs**

Relācijas izmantošanas piemērs ir atrodams attēlā 4.2.6.1., kur tiek izmantotas divas tabulas – viena pamata tabula ar ārējo atslēgu ProductTypeReference, un otra tabula, ar kuru tā atslēga saistās. Attēlā ir redzama read operācija, kura nolasa ierakstu ar ID=5 un saglabā to mainīgā $record. Tad no otras tabulas tiek izveidots iterator objekts, kurš pārvieto rādītāju uz ierakstu, kura atrāšanas vietas pozīcija ir saglabāta $record mainīgā. Tad iterator objekts nolasa to ierakstu no datnes un saglabā mainīgā $productTypeRecord.

4.2.7. Ierakstu indeksācijas īstenojums

Indeksu pārvaldības funkcionalitāte tiek definēta ārpus tabulas pārvaldības moduļa. Indeksi ir atsevišķas struktūras ar iepriekš definētām operācijām. Šī darba ietvaros tika izdarīts mēģinājums īstenot B-tree indeksu ar automātisku zaru balansēšanu. Par pamata krātuvi indeksa īstenošanai tika izmantots entītiju relāciju modeļa tabulas interfeiss. Indeksu tika nolemts īstenot viena tabulas ietvaros, tā struktūra ir atrodama pielikumā 10.

B-tree indekss tiek definēts kā koks, kuram katrs ieraksts satur rādītāju uz oriģinālu ierakstu datnē, kuru indekss optimizē. Indeksa ieraksti tiek sadalīti grupās, kur katra grupa ir binära koka mezgls. Katram mezglam ir rādītājs uz iepriekšējā mezgla pirmo ierakstu. Katram indeksa ierakstam ir rādītājs uz nākamo ierakstu grupā, kā arī uz zemāk esošiem ierakstiem kokā, ja tādi ir. B-tree koka katrs ieraksts satur rādītājus uz ierakstiem, kas atrodas pa kreisi (mazāka vērtība) un pa labi (lielāka vērtība). Indeksa sākumā tiek novietots pamata ieraksts, kurš ir vienīgais pamata

grupā ar tukšo vērtību. Šī indeksa fiziskā uzbūve ļoti aktīvi izmanto ierakstu rādītāju sistēmu datnes ietvaros un tabulas iterācijas interfeisu.

```
# reserve
$table->reserve( mode: Table::RESERVE_READ);
# read index with value 2
$node = $tree->read( value: 2);
# create iterator
$iterator = $table->newIterator();
# jump to the record
$iterator->jump($node->location());
# read the record
$record = $iterator->read();
# release lock
$table->release();
```

att. 4.2.7.1. **Indeksa konstrukcijas izmantošanas piemērs**

Indeksa struktūras piemēra un indeksu īstenošanas apraksta trūkumā dēļ tabulas indeksa struktūras tika īstenotas tikai daļēji. Pētījuma rakstīšanas brīdi B-tree indeksa struktūra pilnīgi atbalsta ierakstu pievienošanu, indeksāciju pēc viena lauka ar integer tipu, un konkrēta ieraksta meklēšanu. Teorētiski, izmantojot izveidotu struktūru, ir iespējams pievienot citu tipu atbalstu, vairāku lauku indeksāciju, papildus meklēšanas operācijas, kā arī ierakstu dzēšanu no indeksa koka struktūras. Indeksa izmantošanas piemērs ir atrodams attēlā 4.2.7.1., kur mainīgais $table satur tabulas objektu un mainīgais $tree satur indeksa objektu. Piemērs apraksta ieraksta ar indeksa lauka vērtību 2 meklēšanu. Sākumā tiek rezervēta tabula lasīšanai, tad tiek izsaukta read metode indeksam ar vērtību 2, un tiek atgriezts indeksa elementa objekts, kurš satur informāciju par meklējamā ieraksta atrašanas vietu. Tad tiek izmantots tabulas $iterator objekts lai nolasītu ierakstu pēc atrāšanas vietas rādītāja datnē.

4.2.8. Kolonnas krātuves atbalsts

Kolonnas krātuves īstenojums ir domāts analītisko pieprasījumu izpildei. Interfeiss ir līdzīgs entītiju relāciju modeļa interfeisam, tomēr atbalsta citu analītiskas krātuves funkcionalitāti. Tika īstenotas divas operācijas: create, read. Ierakstu izveidošanas operācija veido jaunas datnes, kuras pēc ierakstīšanas nav pieejamas nolasīšanai. Tiek definēta speciāla operācija, kura pievieno jaunas datnes pamata datnēm, pēc tam jaunie ieraksti ir pieejami nolasīšanai. Nolasot datus, ir nepieciešams norādīt kolonnas, kuras tiks izmantotas operācijai. Datu nolasīšanai tiek izmantota līdzīgā tabulas iterācijas struktūra.

Datu nolasīšanas piemērs ir redzams attēlā 4.2.8.1., kur $table mainīgais satur tabulas objektu. Tiek izmantota read metode, kura kā pirmo argumentu sagaida masīvu ar nepieciešamiem

kolonas nosaukumiem, un kā otro – atzvanīšanas funkciju. Dotais piemērs nolasa ierakstus, kuriem ID kolonas vērtība ir vienāda ar 5.

```
# using read operator
$result = $table->read(
   ['ID', 'Date'], # columns
   function(array $record){ # search callback
       if ($record['ID'] === 5) {
           return Table::OPERATION_READ_INCLUDE_AND_STOP;
       }
   }
);
```

att. 4.2.8.1. **Ieraksta nolasīšana no analītiskās tabulas**

Katrai kolonnai tiek izveidota atsevišķa datne, papildus datnes tiek sadalītas pēc konkrētas izvēlētas kolonnas vērtībām. Katra datne tiek saspiesta ar pēc noklusējuma pieejamiem PHP valodā rīkiem, kuri izmanto GZIP datnes formātu, īstenojot variāciju no LZ77 un Deivida Hafmena saspiešanas algoritmu [21]. Pašlaik kolonnas modulis ļauj nolasīt tikai tās kolonnas, kuras tika pieprasītas, un saspiež informāciju datnēs, lai izmantotu mazāk vietas uz pastāvīgās atmiņas ierīces.

4.2.9. Papildus komandas fona procesu īstenojumam

Tika izveidotas vairākas papildus komandas datu pārvaldības optimizācijai. Tā kā datu pārvaldība tika īstenota bez vienota procesa atbalsta, komandas ir domātas lietošanai kopā ar crontab lietotni, kura ļauj definēt komandas periodiskas palaišanas intervālus. Atbalstītas komandas un pielietošanas gadījumi:

- Entītiju relāciju tabulas izveidošana – pieņem faila ar tabulas metadatiem atrašanās vietu kā argumentu, inicializē datni tabulas pārvaldībai
- Entītiju relāciju tabulas dzēšana – pieņem faila ar tabulas metadatiem atrašanās vietu kā argumentu, nodzēš tabulas datni
- Entītiju relāciju tabulas optimizācija – pieņem faila ar tabulas metadatiem atrašanās vietu kā argumentu, optimizē tabulas datni, nodzēšot neaktīvus ierakstus no datnes. Kā papildus opcija – saskaitīt neaktīvo / aktīvo ierakstu skaitu datnē
- Kolonnas tabulas dzēšana – pieņem faila ar tabulas metadatiem atrašanās vietu kā argumentu, nodzēš tabulas mapi un visas piesaistītas datnes
- Kolonnas tabulas sadaļas optimizācija – pieņem faila ar tabulas metadatiem atrašanās vietu kā argumentu, optimizē tabulas datnes, pievienojot jaunus ierakstus pamata

krātuvei. Kā papildus arguments – optimizēt tikai konkrētas, piemēram pēdējās, sadaļas datnes

- Kolonnas tabulas sadaļas pārskats – pieņem faila ar tabulas metadatiem atrašanās vietu kā argumentu, parāda visas eksistējošās tabulā sadaļas

Kā papildus funkcionalitāti ir iespējams pievienot daļējo SQL atbalstu netehniskajiem lietotajiem, lai nodrošinātu datu izpētes iespējas lietotnē.

4.2.10.Secinājumi datu pārvaldības sistēmas izstrādei

Datu pārvaldības bibliotēkas īstenojums augstā līmeņa programmēšanas valodā ir unikāla pieredze. Bibliotēka īsteno pamata funkcionalitāti datu pārvaldībai un piedāvā vairākus rīkus efektīvai datu pārvaldībai, rīki ir integrēti izvēlētajā programmēšanas valodā. Protams, ir nepieciešami vairāki uzlabojumi un papildinājumi, lai šo bibliotēku varētu ērti izmantot, tomēr izstrādes laikā kļuva skaidrs, ka to ir iespējams izdarīt. Tehniskā līmenī PHP valoda piedāvā pietiekamu funkcionalitāti datnes pārvaldībai. Konceptuālā līmenī moduļu sadalījuma pieeja mēdz nodrošināt vairākas iespējas gan ērtai datu pārvaldībai, gan arī zema līmeņa efektīvai datu pārvaldībai.

Kopā tika īstenoti šādi moduļi:

- Entītiju relāciju modeļa tabulas modulis
- Entītiju relāciju modeļa tabulas iterācijas modulis
- Entītiju relāciju modeļa kolonnas indeksa modulis (daļēji)
- Kolonnas modeļa tabulas modulis
- Kolonnas modeļa tabulas kolonnas modulis
- Datnes pārvaldības interfeiss (zemāka līmeņa modulis, klase operācijām ar datnēm)

Tos ir iespējams pielietot atsevišķi, gan arī to pielietošanu var elastīgi kombinēt. Efektīvas un elastīgas datu pārvaldības atrašana bija viens no šī pētījuma mērķiem, un praktiskās izstrādnes liecina, ka iebūvētas bezservera bibliotēkas praktisko īstenojumu ir iespējams pielāgot efektīvai lietošanai produkcijas vidē.

4.3. Lietotnes īstenošana, izmantojot datu pārvaldības bibliotēku

Datu pārvaldības bibliotēkas integrācijas ar lietotnes programmu iespējām tika definēts kā uzdevumu pārvaldības projekts. Projekts piedāvā funkcionalitāti - pārvaldīt sava lietotāja uzdevumu sarakstu. Lietotājam ir iespēja piereģistrēties un pieslēgties, izmantojot savu lietotāja vārdu un paroli. Lietotājs var pievienot uzdevuma sarakstam jaunu ierakstu, spēj rediģēt ieraksta tekstu, atzīmēt uzdevumu kā izpildītu un iztīrīt vienu vai vārākus izpildītus uzdevumus no saraksta. Darba un šīs nodaļas mērķis ir noskaidrot, vai izveidoto datu pārvaldības bibliotēku ir iespējams izmantot lietotnes īstenošanai. Lietotne tika izstrādāta, izmantojot MIT atvērtā koda licenci un tika ievietota atvērtai pieejai tīmekļa vietnē GitHub pēc saites https://github.com/bozerkins/dmc-example-app.

4.3.1. Lietotnes datu modelis, tabulas un relācijas

Lietotnes īstenošanai tika izvēlēts entītiju relāciju modelis, jo tas ir klasisks modelis dotās lietotnes izveidošanai. Tika izveidotas divas tabulas: lietotāju tabula un uzdevumu tabula. Lietotāju lauki ir: unikālais identifikators, lietotāja vārds, lietotāja parole, ieraksta izveidošanas datums un laiks. Uzdevumu tabulā ir šādi lauki: unikālais identifikators, uzdevuma teksts, mārkeris, kurš norāda vai uzdevums ir izpildīts, norādītājs uz lietotāja ierakstu. Ierakstu unikalitāte un relācijas netiek reģistrētas, izņemot izpildes kodu un migrāciju failus (kuri veido tabulu struktūru). Unikalitāte un identifikatoru automātiskais palielinājums tiek nodrošināts lietotnes kodā, nevis datu pārvaldības bibliotēkā. Papildus, lai nodrošinātu automātisko identifikatora palielinājumu, tika izveidota papildus tabula ar identifikatoru skaitļotājiem, ar laukiem: tabulas nosaukums, pēdējais tabulas identifikators. Tabulu veidošanas skripti ir atrodami pielikumā 11.

4.3.2. Lietotāju izveide un autentifikācija

Lietotne atbalsta reģistrāciju un pieslēgšanu uzdevumu saraksta pārvaldībai. Lietotājam reģistrācijai ir nepieciešams unikāls lietotāja vārds un parole. Lietotājs tiek saglabāts lietotāju tabulā, papildus tiek pievienots unikāls lietotāja identifikators un lietotāja izveidošanas datums un laiks. Lietotāja paroles tiek šifrētas, izmantojot bcrypt algoritmu. Kad lietotājs mēģina ielogoties, tiek pārbaudīts, vai tāds lietotājs un parole eksistē datubāzē. Vispirms lietotājs tiek meklēts pēc

lietotāja vārda un tad tiek pārbaudīta parole. Ja lietotājam parole sakrīt un tāds lietotājs eksistē datnē, lietotāja identifikators tiek ierakstīts sesijā, citādi tiek izmesta autentifikācijas kļūda. Autentifikācijas un reģistrācijas piemēra skripti ir atrodami pielikumā 12.

4.3.3. Lietotnes funkcionāla nodrošinājums

Lietotne īsteno vairākas operācijas ar datiem. Pirms katras operācijas izpildes tiek pārbaudīts lietotāja autentifikācijas status. Ja lietotājs ir joprojām pieslēgts, lietotāja ieraksts tiek nolasīts no datubāzes, izmantojot lietotāja identifikatoru. Relāciju funkcionalitātes nodrošinājumam tiek izmantots lietotāja ieraksta pozīcijas rādītājs datnē. Tiek nodrošinātas šādas operācijas ar uzdevumiem: uzdevuma ieraksta pievienošana, uzdevumu saraksta nolasīšana vienam lietotājam, uzdevumu dzēšana, visu izpildīto uzdevumu dzēšana, visu lietotāja uzdevumu atzīmēšana par izpildītiem, uzdevuma teksta maiņa viena lietotāja ietvaros. Lietotnes skriptu piemēri ir pievienoti pielikumā 13.

4.3.4. Lietotnes izstrādes secinājumi

Vienkāršās lietotnes izstrāde, izmantojot izveidotu datu pārvaldības bibliotēku, nav triviāls uzdevums, tomēr tas ir izpildāms. Datu pārvaldības komponentes nodrošina stabilu datu pārvaldību, pietiekami salasāmu un pielietojamu operāciju interfeisu. Izpildes laikā pietrūka datu izpētes funkcionalitātes, jo dažreiz bija jāpārbauda vai dati tika ierakstīti pareizi. Pietrūka iebūvētas unikalitātes pārbaudes un identifikatoru automātiskas palielināšanas funkcionalitātes, tomēr to bija iespējams īstenot, izmantojot esošos datu pārvaldības bibliotēkas rīkus. Izstrādes laikā datu pārvaldības operāciju izpildes ātrdarbības problēmas netika atrastas, teorētiski ar tādu pieeju problēmas ir iespējamas risināt vienlaicīgu procesu skaitam palielinoties.

REZULTĀTI

Darbā tika izpētīti datu pārvaldības pamatprincipi, vairāku datu pārvaldības sistēmu arhitektūras, kas tiek plaši izmantotas lietotnes pārvaldības īstenošanai pasaulē, kā arī aprakstītas vairākās grāmatās un pētījumos. Tika izpētīti datu pārvaldības tehniskie un algoritmiskie paņēmieni, kas tiek izmantoti datubāzes funkcionalitātes nodrošināšanai.

SQL nodrošina informācija pārvaldības konceptus relāciju modeļa ietvaros, kura pirmsākumi un pamata principi tika izpētīti un konceptualizēti. Tika apskatīta relāciju datubāzes un SQL pamata funkcionalitāte un abu jēdzienu mijiedarbības tendences. Maģistra darba izstrādes gaitā, analizējot literatūru, tika konstatēts, ka relāciju datubāzes pirmsākumi ir Sistēmas R konceptuālais un tehniskais apraksts, kas nodrošina šādus datu pārvaldības problēmu risinājumu:

Sistēmas R tika veidotas ar mērķi pierādīt, ka Relāciju modelis var būt izmantots produkcijas vides datu pārvaldībai un darba gaitā tika risinātas šādas problēmas:

1. konceptuālo un tehnisko skatu atdalījums datubāzes ietvaros
2. salasāma datu pārvaldības interfeisa izveide
3. vienlaicīgas piekļuves nodrošinājums
4. metadatu atdalījums no datu pārvaldības

SQL standarti un to īstenojums MySQL datubāzē tika izpētīti gan no konceptuālā viedokļa, gan arī no tehniskā. SQL pamata funkcionalitāte tika izpētīta reāla, atvērtā koda, relāciju datubāzes ietvaros un tika konstatētas vairākas SQL valodas un datubāzes monolītas arhitektūras izmaņas. Šis secinājums lika aizdomāties par datu pārvaldības sistēmas izveidošanas arhitektūras problēmām un pieprasījumu valodas nozīmi tādas sistēmas ietvaros.

Tika izpētītas mūsdienīgas relācijas datubāzes arhitektūras, pamata komponentes un paņēmieni SQL atbalstam, datu pārvaldības konsistences nodrošināšanai. Tas iekļauj vairāku paralēlu procesu pārvaldību, datnes slēgšanu, procesu prioritizāciju un papildus iespējas, piemēram, ne-relāciju konceptu atbalsts, piemēram, XML vai JSON datu tipu darbības. Datu pārvaldības arhitektūras variāciju izpētes laikā tika atrastas bezservera sistēmas, kuras nodrošina modulāru pieeju izstrādē un datu pārvaldībā, kuras funkcionalitāte ir pieejama lietotnes izstrādei.

Tehniskas un arhitektūras uzbūves elementi tika apkopoti un analizēti. Pamata datu pārvaldības paņēmieni ir cieši saistīti ar izvēlēto fiziskas datnes formātu. Ierakstu organizācija datnēs iekļauj sevī metadatu pārvaldības procesu atbalstu un papildus datu operāciju īpašības.

Tomēr tieši datnes pārvaldības fiziskās operācijas un šo operāciju optimizācijas paņēmieni ir pamatā nemainīgi.

Datu pārvaldības sistēmas izstrādē tika definēts fiziskas datnes struktūras atbalsta segments, konceptuālas datu uzbūves paņēmieni, t.i. dinamiska izmēra lauku atbalsts, vairāku transakcijas tipu atbalsts, datu konsistences, atjaunošanas un rezerves kopēšanas izstrāde, kas ir atkarīga no: 1) definētas datnes struktūras, 2) no iedomātās datnes pārvaldības funkcionalitātes. Pētījuma laikā tika atrastas interesantas un efektīvas pieejas.

Datu pārvaldības bibliotēka īstenoja daļu no ieplānotās funkcionalitātes, izmantojot iepriekš izpētītus paņēmienus un datu struktūras. Bibliotēka piedāvā gan relāciju, gan ne-relāciju datu pārvaldības pieeju. Relāciju pieeja nodrošina datu konsistenci un atomisku operāciju izpildi. Tika īstenotas create, read, update, delete operācijas un vairākas metadatu pārvaldības koncepcijas. Modulāras datu pārvaldības mēģinājumi parādīja plašākas lietotnes izstrādes iespējas, jo piekļuve vairākiem datu pārvaldības bibliotēkas moduļiem ļauj tos izmantot tikai daļēji un ārpus pamata datu pārvaldības funkcionalitātes. Papildus, datu pārvaldības operāciju rezultāti kļuva paredzamāki, jo izmantojot zemā līmeņa operācijas un programmēšanas valodas sintakses interfeisu, no operāciju konteksta ir saprotams cik diska nolasīšanas operācijas un programmas iterācijas tiek veiktas izpildes laikā.

Bibliotēka tika izmantota vienkāršas lietotnes izstrādei. Diemžēl tika pielietotas tikai relāciju bibliotēkas iespējās un analītiska moduļa integrācijai pietrūka laika. Bija īpaši interesanti izmantot alternatīvu SQL-am datu pārvaldības pieeju, kura piedāvā elastīgu un vienkāršu datu pārvaldības konceptu. Tomēr vairāku datu pārvaldības iespēju pietrūka, daļu no tiem bija iespējams īstenot ar esošiem rīkiem, tomēr SQL ir efektīvāks interfeiss datu manuālai izpētei. Darba gaitā tika iegūtas būtiskas iemaņas datu pārvaldības sistēmu uzbūvē un zināšanas par tehniskām problēmām, kuras datubāzes sistēmas mēģina risināt, piedāvājot konceptuālas datnes pārvaldības rīkus.

SECINĀJUMI

Datubāzes sistēmas īsteno vairākas pieejas datu pārvaldībai un datu konsistences nodrošināšanai. Datnes tehniskās robežas un operētājsistēmu datnes pārvaldības interfeisi diktē pastāvīgas ierīces pārvaldības pieejas. Apskatot konceptuālas prasības datnes pārvaldībai un tehniskus ierobežojumus, datu operācijas ir iespējams segmentēt uz vienkārši īstenojamiem un tehniski netriviāliem. Datubāzes mēdz piedāvāt tikai konceptuālo skatu un konceptuālus ierobežojumus, kas, no vienas puses, palīdz koncentrēt izstrādes laiku uz problēmu risināšanu, no citas puses, slēdz pieeju elastīgākai datu pārvaldībai, piemēram iespējas kombinēt vairākus pārvaldības moduļus optimālākai konceptuālo operāciju izpildei, vai indeksa struktūras paplašinājuma iespējas.

Relāciju datubāzes nodrošina uz tabulām un relācijām balstītas koncepcijas pārvaldību, kura ir cieši saistīta ar SQL definējumu un problēmām, kuras šī valoda mēdz risināt. SQL ir monolīta valoda, kuras ietvaros nav triviāla veida kā mainīt konceptuālu pieeju, piemēram MySQL datubāzē JSON tipa atbalsts tiek īstenots izmantojot speciālas pārvaldības metodes, kuras pamata ir simbolu virknes noklusētais datu tips, kas nozīme JSON atbalst ir relācijas koncepta paplašinājums tehniski un konceptuāli.

Relāciju datubāzes sistēmas izmanto monolītas pieejas, tendences programmas īstenojumā, kur pamatā tiek izmantotas relāciju modeļa koncepcijas. Relāciju koncepts nav pielāgots izmaiņām, tomēr atbalsta vairāku veidu paplašinājumus, kas palīdz saglabāt aktualitāti un novitāti pielietošanas pieejās. Ir skaidrs, ka pašreizējie relāciju datubāžu un SQL problēmas un mērķi stipri atšķiras no tiem, ar ko strādāja datu pārvaldības pirmsākumu sistēmas.

Datu pārvaldības sistēmu izstrāde nav triviāls uzdevums, tomēr ir vairāki izplatīti paņēmieni pamata pārvaldības īpašību nodrošināšanai. Datu pārvaldības uzlabojumi un papildus funkcionalitāte tiek nodrošināta ar ieraksta pārvaldības dizaina paplašinājuma. Datu pārvaldības koncepciju attīstība ir pakāpeniska, tas ļauj īstenošanas laikā korekti un efektīvi sadalīt datu pārvaldības funkcionalitāti vairākos moduļos. Modulāra pieeja spēj piedāvāt vairāk iespēju un optimizāciju datu pārvaldībai, vienkāršot operācijas uz datiem un padara operāciju rezultātu paredzamāku. Modularitātes īstenošanas tendences ir redzamas mūsdienu datubāzes sistēmu īstenojumos (gan relāciju, gan iebūvētu), tomēr piesaiste pie SQL mēdz sarežģīt modularitātes attīstību un atbalstu.

Šīs darbs piedāvā pārskatu par iebūvētas bezservera datubāzes koncepta īstenojumu kā alternatīvu klasiskai relāciju SQL datubāzei konceptuālā un praktiskā pamata līmenī. Ir vairāki virzieni tālākajiem pētījumiem: šīs pieejas ātrdarbības jautājums augstākā un zemākā līmeņa programmēšanas valodas īstenojumā, sarežģītāku datu pārvaldības struktūru īstenojums un modulāras pieejas priekšrocības, izkliedēta datu pārvaldība iebūvētā un modulārā koncepta ietvaros.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Tehnoloģiju aptauja 2017 gads StackOverflow vietne [tiešsaiste] – [pārbaudīts 22.01.2018]. Pieejams: https://insights.stackoverflow.com/survey/2017
2. NoSQL definējums un jēdziena apraksts [tiešsaiste] – [pārbaudīts 26.04.2018]. Pieejams: https://en.wikipedia.org/wiki/NoSQL
3. MySQL dokumentācijas darbam ar JSON tabulas kolonnas tipu [tiešsaite] – [pārbaudīts 26.04.2018]. Pieejams: https://dev.mysql.com/doc/refman/8.0/en/json.html
4. Kristi L. Berg, Tom Seymour, Richa Goel, History Of Databases, International Journal of Management & Information Systems – First Quarter 2013 Volume 17, Number 1
5. Donald D. Chamberlin, Early History of SQL, IEEE Annals of the History of Computing, 2012
6. Peter Lake, Paul Crowther, Concise Guide to Databases, A Practical Introduction, 2013
7. Joseph M. Hellerstein, Michael Stonebraker, James Hamilton, Architecture of a Database System, 2007
8. Mohamed A. Mohamed, Obay G. Altrafi, Mohammed O. Ismail, Relational vs. NoSQL Databases: A Survey, 2014
9. IBM System R [tiešsaiste] – [pārbaudīts 13.01.2018]. Pieejams: https://en.wikipedia.org/wiki/IBM_System_R
10. M. M. ASTRAHAN, ht. W. BLASGEN, D. D. CHAMBERLIN, K. P. ESWARAN, J. N. GRAY, P. P. GRIFFITHS, W. F. KING, R. A. LORIE, P. R. A&JONES, J. W. MEHL, G. R. PUTZOLU, I. L. TRAIGER, B. W. WADE, AND V. WATSON, System R: Relational Approach to Database Management, 1975
11. Comments on “Chamberlin and Boyce - SEQUEL: A Structured English Query Language” [tiešsaiste] – [pārbaudīts 13.01.2018]. Pieejams: http://www.cs.grinnell.edu/~rebelsky/Courses/CS302/2007S/Readings/chamberlin-sequel.html
12. Digital Equipment Corporation Maynard, Massachusetts, Information Technology - Database Language SQL (Proposed revised text of DIS 9075), July 30, 1992. [tiešsaiste] – [pārbaudīts 08.05.2018]. Pieejams: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt
13. MySQL datubāzes dokumentācija, versija 8.0 [tiešsaiste] – [pārbaudīts 08.05.2018].

Pieejams: https://dev.mysql.com/doc/refman/8.0

14. Sasha Pachev, Understanding MySQL Internals, 2007
15. MySQL datubāzes programmas kods, versija 8.0 [tiešsaiste] – [pārbaudīts 08.05.2018]. Pieejams: https://github.com/mysql/mysql-server/tree/8.0
16. Joseph M. Hellerstein, Michael Stonebraker, James Hamilton, Architecture of a Database System, 2007
17. Aleksandra Tešanovic, Dag Nyström, Jörgen Hansson, Christer Norström, Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach, 2002
18. Amy Brown, Greg Wilson, Margo Seltzer, Keith Bostic, The Architecture of Open Source Applications - Berkley DB, 2011
19. PHP dokumentācija [tiešsaiste] – [pārbaudīts 13.05.2018]. Pieejams: http://php.net/manual/en/
20. Datu pārvaldības bibliotēkas īstenojums [tiešsaiste] – [pārbaudīts 13.05.2018]. Pieejams: https://github.com/bozerkins/dmc
21. GZIP datnes formāta apraksts [tiešsaiste] – [pārbaudīts 13.05.2018]. Pieejams: https://en.wikipedia.org/wiki/Gzip

PIELIKUMI

Pielikums 1. Sistēmas R arhitektūra [10]

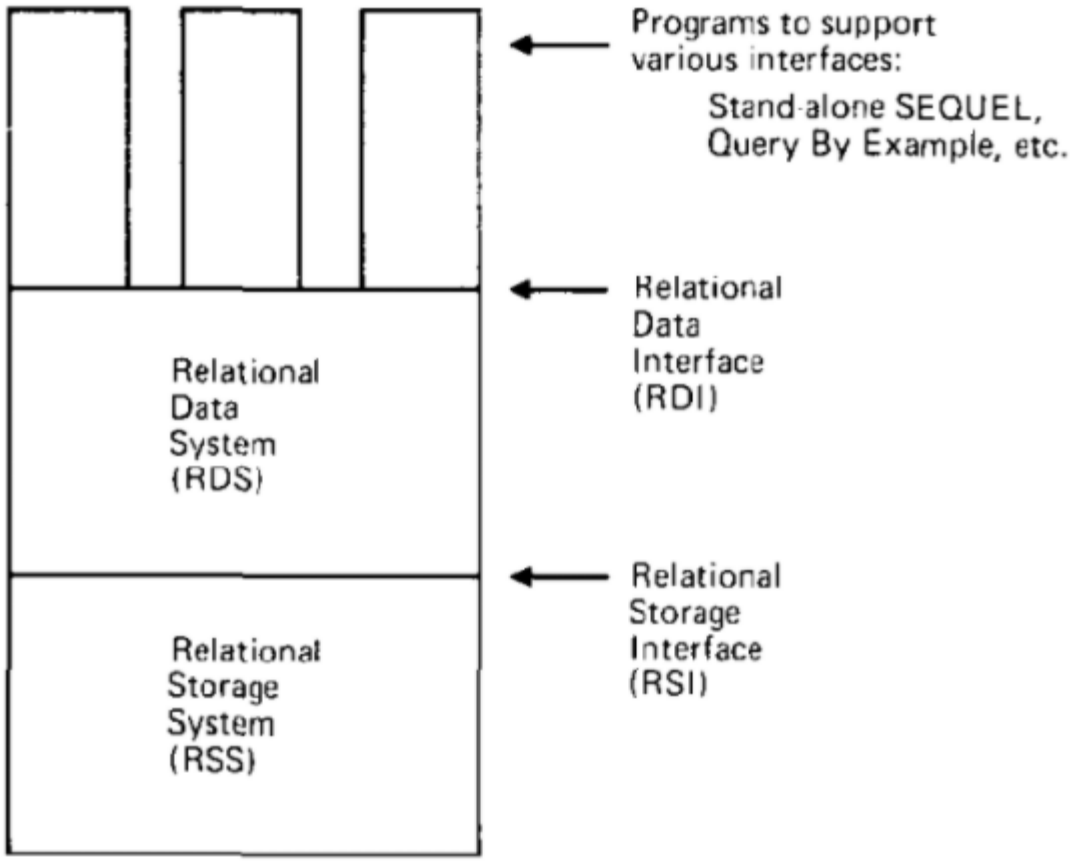

FIG. 1. Architecture of System R

Pielikums 2. Sistēmas R arhitektūra virtuālajā vidē [10]

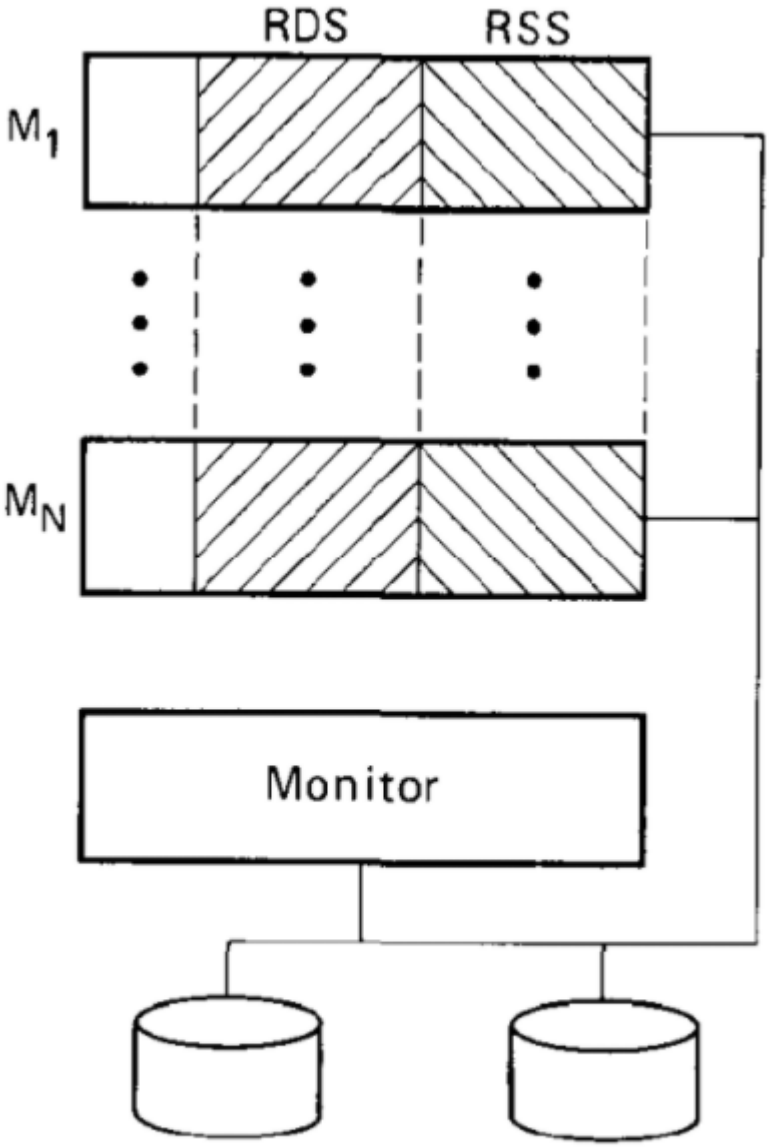

FIG. 2. Use of virtual machines in System R

Pielikums 3. Relāciju datubāzes arhitektūra [16]

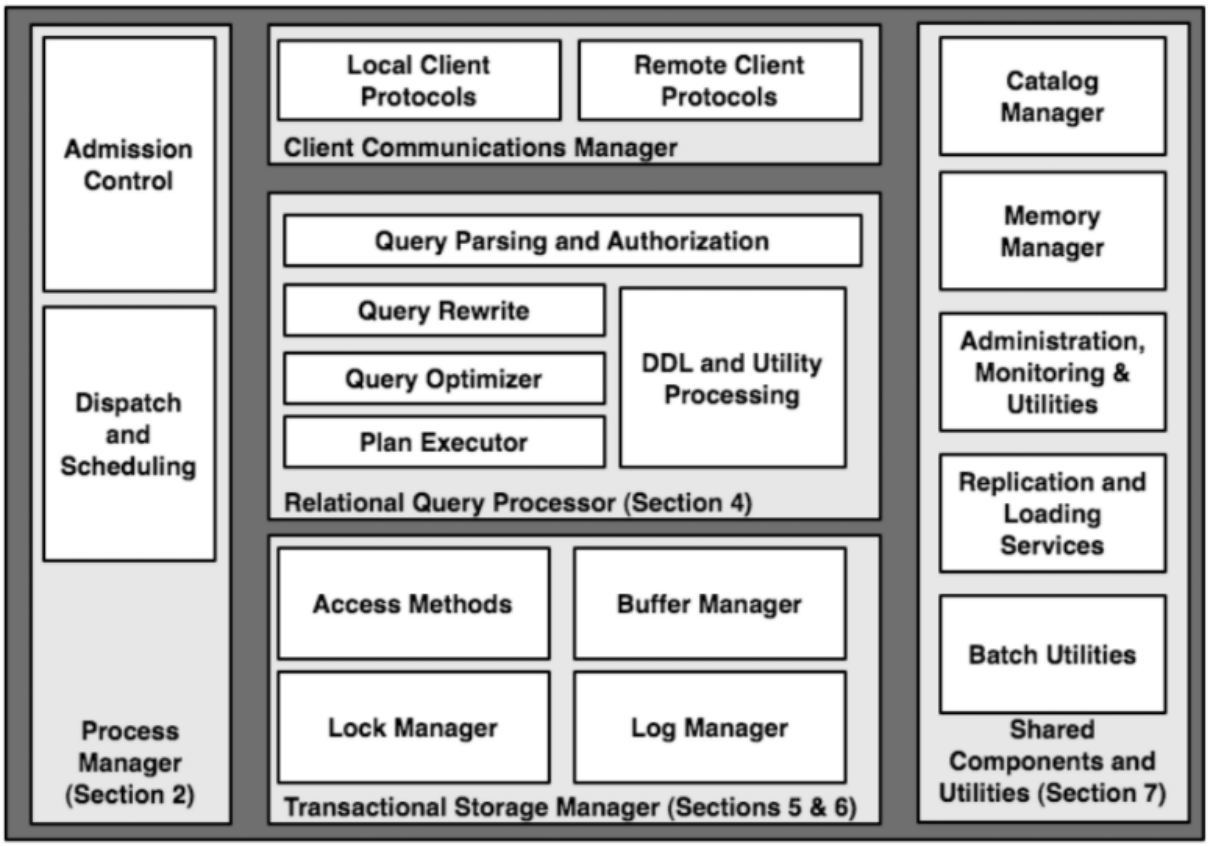

Fig. 1.1 Main components of a DBMS.

Pielikums 4. Relāciju datubāzes arhitektūras piemērs - MySQL [14]

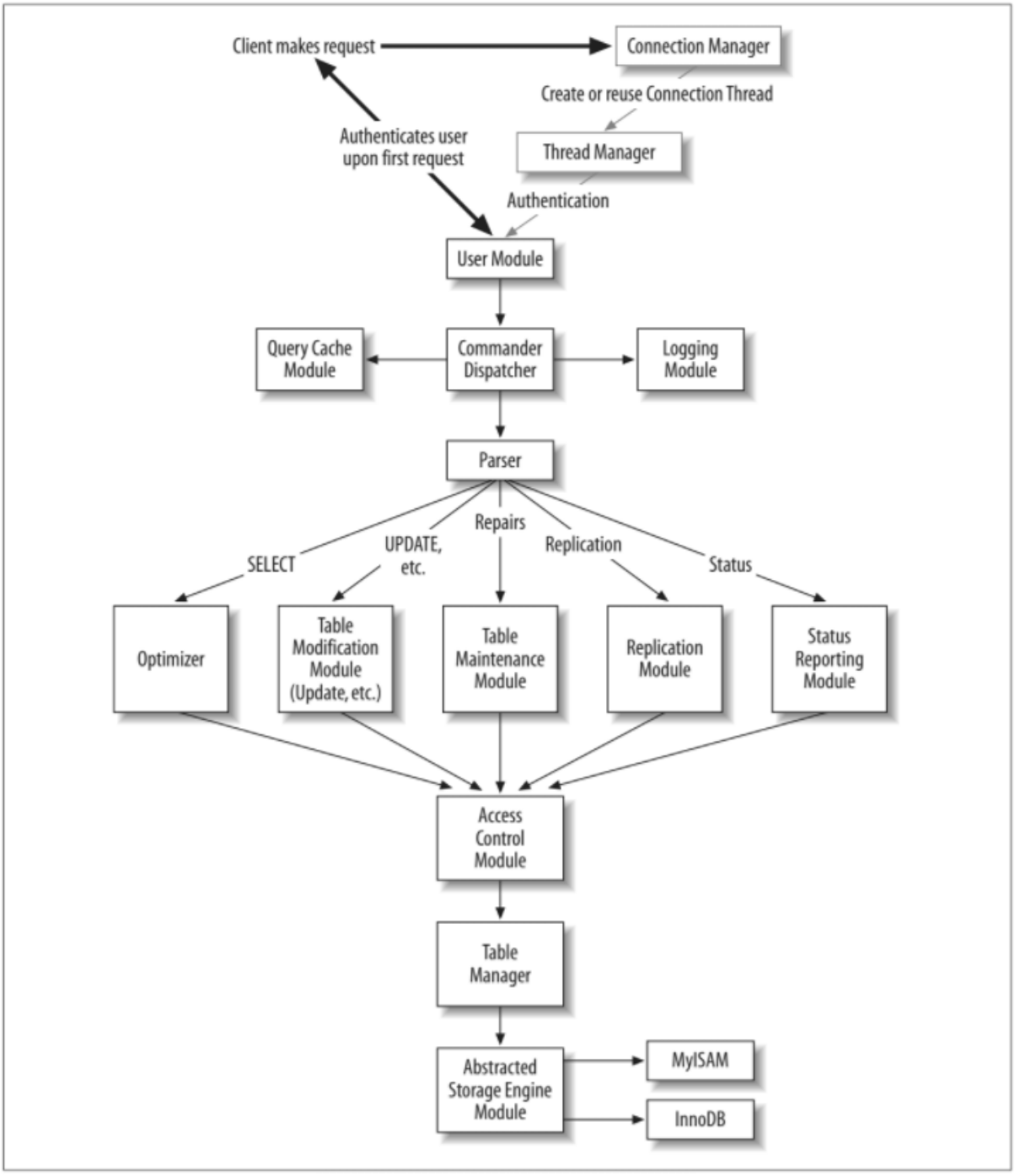

Figure 1-1. High-level view of MySQL modules

Pielikums 5. MySQL datnes slēgšanas tipi [14]

Table 8-1. Types of locks in MySQL

Lock type	Description
TL_IGNORE	A special value used in locking requests to communicate that nothing should be done in the lock descriptor structures.
TL_UNLOCK	A special value used in locking requests to communicate that a lock should be released.

Table 8-1. Types of locks in MySQL (continued)

Lock type	Description
TL_READ	A regular read lock.
TL_READ_WITH_SHARED_LOCKS	A higher priority lock used by InnoDB for SELECT... LOCK IN SHARE MODE.
TL_READ_HIGH_PRIORITY	A high priority read lock used by SELECT HIGH_PRIORITY....
TL_READ_NO_INSERT	A special read lock that does not allow concurrent inserts.
TL_WRITE_ALLOW_WRITE	A special lock used by storage engines that take care of locking on their own. Other threads are allowed to acquire read and write locks while this lock is being held.
TL_WRITE_ALLOW_READ	A special lock for ALTER TABLE. Altering a table involves creating a temporary table with the new structure, populating it with new rows, and then renaming it to the original name. Thus a table can be read while being altered during most of the operation.
TL_WRITE_CONCURRENT_INSERT	The write lock used by concurrent inserts. If this type of lock is already placed on the table, read locks are granted to other threads immediately unless TL_READ_NO_INSERT is requested.
TL_WRITE_DELAYED	A special lock used by INSERT DELAYED....
TL_WRITE_LOW_PRIORITY	A low-priority lock used in UPDATE LOW_PRIORITY... and other queries with the LOW_PRIORITY attribute.
TL_WRITE	A regular write lock.
TL_WRITE_ONLY	An internal value used when aborting old locks during operations that require closing tables.

Pielikums 6. Iebūvētas datubāzes modeļu īstenošanas pārskats [17]

DBMS system	Client/server	Library
Pervasive.SQL	x	
Polyhedra	x	
Velocis	x	
RDM		x
Berkeley DB		x
TimesTen	x	

Table 2.1: DBMS models supported by different embedded database systems.

Pielikums 7. Datu pārvaldības bibliotēkas tabulu izveidošanas skripts

```
use DataManagement\Model\EntityRelationship\Table;
use DataManagement\Model\TableHelper;

$table = new Table();
$table->addColumn('ID', TableHelper::COLUMN_TYPE_INTEGER);
$table->addColumn('Profit', TableHelper::COLUMN_TYPE_FLOAT);
$table->addColumn('ProductType', TableHelper::COLUMN_TYPE_INTEGER);
$table->addColumn('ProductTypeReference', TableHelper::COLUMN_TYPE_INTEGER);
$table->addColumn('ProductTitle', TableHelper::COLUMN_TYPE_STRING, 100);

$structure = var_export($table->structure(), true);
$location = '~/storage/my-table';

$instructions = <<<EOT
<?php
return [
    'location' => '{$location}',
    'structure' => {$structure}
];
EOT;
$instructionFileDestination = __DIR__ . '/../instructions/users.php';
# create the file
touch($instructionFileDestination);
# make it writable by everyone
chmod($instructionFileDestination, 0777);
# write the contents
file_put_contents($instructionFileDestination, $instructions);
```

Pielikums 8. Datu pārvaldības bibliotēkas CRUD operācijas

```php
use DataManagement\Model\EntityRelationship\Table;

$instructionFile = '/project-root/my-table-instruction.php';
$table = Table::newFromInstructionsFile($instructionFile);
# find and update the table
$table->update(
    # search the record
    function(array $record) {
        if ($record['ID'] === 5) {
            return Table::OPERATION_UPDATE_INCLUDE_AND_STOP;
        }
    },
    # update each record that's included
    function(array $record) {
        return ['ProductTitle' => 'My first product update!'];
    }
);
```

```php
use DataManagement\Model\EntityRelationship\Table;

$instructionFile = '/project-root/my-table-instruction.php';
$table = Table::newFromInstructionsFile($instructionFile);
# find and update the table
$table->delete(
    # search the record
    function(array $record) {
        if ($record['ID'] === 5) {
            return Table::OPERATION_DELETE_INCLUDE_AND_STOP;
        }
    }
);
```

```php
$instructionFile = '/project-root/my-table-instruction.php';
$table = Table::newFromInstructionsFile($instructionFile);
# read all the table
$result = $table->read(function(){
    return Table::OPERATION_READ_INCLUDE;
});
# read the record with specific ID
$result = $table->read(function(array $record){
    if ($record['ID'] === 5) {
        return Table::OPERATION_READ_INCLUDE_AND_STOP;
    }
});
# read only first two record
$result = [];
$table->iterate(function(array $record) use (&$result) {
    $result[] = $record;
    if (count($result) === 2) {
        return Table::OPERATION_READ_STOP;
    }
});
# read id and pointer of all the records
$result = [];
$table->iterate(function(array $record, TableIterator $iterator) use (&$result) {
    $result[] = ['id' => $record['ID'], 'pointer' => $iterator->position()];
});
```

Pielikums 9. Datu pārvaldības bibliotēkas CRUD operāciju zemā līmeņa piemēri

```php
use DataManagement\Model\EntityRelationship\Table;

$instructionFile = '/project-root/my-table-instruction.php';
$table = Table::newFromInstructionsFile($instructionFile);
# reserve the table
$table->reserve(Table::RESERVE_READ_AND_WRITE);
# get iterator
$iterator = $table->newIterator();
# reset pointer
$iterator->jump(0);
# loop over table
while ($iterator->endOfTable() === false) {
    # read the record
    $record = $iterator->read();
    # check if it's not deleted
    if ($record === null) {
        continue;
    }
    # check if record found
    if ($record['ID'] === 5) {
        # rewind pointer to the record location
        $iterator->rewind(1);
        # update
        $iterator->update(['ProductTitle' => 'Updating in the iterator']);
        # exit the loop
        break;
    }
}
# release the table
$table->release();
```

Pielikums 10. Datu pārvaldības bibliotēkas indeksa struktūras definējums

```
return array (
    array (
        'id' => 1,
        'name' => 'parent_node',
        'type' => TableHelper::COLUMN_TYPE_INTEGER,
        'size' => 4,
    ),
    array (
        'id' => 2,
        'name' => 'next_in_bucket',
        'type' => TableHelper::COLUMN_TYPE_INTEGER,
        'size' => 4,
    ),
    array (
        'id' => 3,
        'name' => 'left_node',
        'type' => TableHelper::COLUMN_TYPE_INTEGER,
        'size' => 4,
    ),
    array (
        'id' => 4,
        'name' => 'right_node',
        'type' => TableHelper::COLUMN_TYPE_INTEGER,
        'size' => 4,
    ),
    array (
        'id' => 5,
        'name' => 'value',
        'type' => $valueType,
        'size' => $valueSize,
    ),
);
```

Pielikums 11. Īstenotas lietotnes tabulu izveidošanas skripti

```php
#ids.php

require_once __DIR__ . '/../../vendor/autoload.php';

use DataManagement\Model\EntityRelationship\Table;
use DataManagement\Model\TableHelper;

$table = new Table();
$table->addColumn( name: 'table', type: TableHelper::COLUMN_TYPE_STRING, size: 100);
$table->addColumn( name: 'last_id', type: TableHelper::COLUMN_TYPE_INTEGER);

$structure = var_export($table->structure(), return: true);
$location = __DIR__ . '/../../data/ids';

$instructions = <<<EOT
<?php
return [
    'location' => '{$location}',
    'structure' => {$structure}
];
EOT;
$instructionFileDestination = __DIR__ . '/../instructions/ids.php';
# create the file
touch($instructionFileDestination);
# make it writable by everyone
chmod($instructionFileDestination, mode: 0777);
# write the contents
file_put_contents($instructionFileDestination, $instructions);

# create table
$table = Table::newFromInstructionsFile($instructionFileDestination);
$table->storage()->create();
```

```php
# todos.php

require_once __DIR__ . '/../../vendor/autoload.php';

use ...

$table = new Table();
$table->addColumn( name: 'ID', type: TableHelper::COLUMN_TYPE_INTEGER);
$table->addColumn( name: 'Title', type: TableHelper::COLUMN_TYPE_STRING, size: 255);
$table->addColumn( name: 'Completed', type: TableHelper::COLUMN_TYPE_INTEGER);
$table->addColumn( name: 'UserReference', type: TableHelper::COLUMN_TYPE_INTEGER);

$structure = var_export($table->structure(), return: true);
$location = __DIR__ . '/../../data/todos';

$instructions = <<<EOT
<?php
return [
    'location' => '{$location}',
    'structure' => {$structure}
];
EOT;
$instructionFileDestination = __DIR__ . '/../instructions/todos.php';
# create the file
touch($instructionFileDestination);
# make it writable by everyone
chmod($instructionFileDestination, mode: 0777);
# write the contents
file_put_contents($instructionFileDestination, $instructions);

# create table
$table = Table::newFromInstructionsFile($instructionFileDestination);
$table->storage()->create();
```

```php
# users.php

require_once __DIR__ . '/../../vendor/autoload.php';

use ...

$table = new Table();
$table->addColumn( name: 'ID', type: TableHelper::COLUMN_TYPE_INTEGER);
$table->addColumn( name: 'Username', type: TableHelper::COLUMN_TYPE_STRING, size: 100);
$table->addColumn( name: 'Password', type: TableHelper::COLUMN_TYPE_STRING, size: 200 );
$table->addColumn( name: 'CreatedAt', type: TableHelper::COLUMN_TYPE_STRING, size: 20);

$structure = var_export($table->structure(), return: true);
$location = __DIR__ . '/../../data/users';

$instructions = <<<EOT
<?php
return [
    'location' => '{$location}',
    'structure' => {$structure}
];
EOT;
$instructionFileDestination = __DIR__ . '/../instructions/users.php';
# create the file
touch($instructionFileDestination);
# make it writable by everyone
chmod($instructionFileDestination, mode: 0777);
# write the contents
file_put_contents($instructionFileDestination, $instructions);

# create table
$table = Table::newFromInstructionsFile($instructionFileDestination);
$table->storage()->create();
```

Pielikums 12. Īstenotas lietotnes pieslēgšanas un reģistrācijas skripti

```php
# login.php

require_once __DIR__ . '/bootstrap.php';

use \DataManagement\Model\EntityRelationship\Table;

session_start();

$username = array_key_exists( key: 'username', $_REQUEST) ? $_REQUEST['username'] : null;
$password = array_key_exists( key: 'password', $_REQUEST) ? $_REQUEST['password'] : null;

if ($username === null) {
    echo 'empty username';
    exit;
}
try {
    $users = Table::newFromInstructionsFile( file: __DIR__ . '/database/instructions/users.php');
    $result = $users->read(function($record) use ($username, $password) {
        if ($record['Username'] === $username && password_verify($password,$record['Password'])) {
            return Table::OPERATION_READ_INCLUDE_AND_STOP;
        }
    });
    $user = null;
    if ($result) {
        $user = $result[0];
    }
    if ($user === null) {
        $_SESSION['message_bad'] = 'Authentication error';
        header( string: 'Location: index.php');
        exit;
    }
    // set user
    $_SESSION['user_id'] = $user['ID'];
    // redirect
    header( string: 'Location: app/index.php');
    exit;
} catch (\Exception $ex) {
    echo 'Unexpected error';
    exit;
}
```

```php
if ($password !== $password_confirm) {
    $_SESSION['message_bad'] = 'Passwords does not match';
    header( string: 'Location: index.php');
    return;
}

try {
    $users = Table::newFromInstructionsFile( file: __DIR__ . '/database/instructions/users.php');
    $result = $users->read(function($record) use ($username) {
        if ($record['Username'] === $username) {
            return Table::OPERATION_READ_INCLUDE_AND_STOP;
        }
    });
    $user = null;
    if ($result) {
        $user = $result[0];
    }
    if ($user !== null) {
        $_SESSION['message_bad'] = 'Username already exists';
        header( string: 'Location: index.php');
        exit;
    }

    $user = [];
    $user['ID'] = getNextId( table: 'users');
    $user['Username'] = $username;
    $user['Password'] = password_hash($password,  algo: PASSWORD_BCRYPT);
    $user['CreatedAt'] = date( format: 'Y-m-d H:i:s');
    $users->create($user);

    $_SESSION['message_good'] = 'User created. You can login now!';
    header( string: 'Location: index.php');
    exit;

} catch (\Exception $ex) {
    echo 'Unexpected error';
    exit;
}
```

Pielikums 13. Īstenotas lietotnes skriptu piemēri

```php
# add.php

require_once __DIR__ . '/../bootstrap.php';

use \DataManagement\Model\EntityRelationship\Table;

$user = initializeUserOrExit();

$title = array_key_exists( key: 'Title', $_REQUEST) ? $_REQUEST['Title'] : null;
$completed = array_key_exists( key: 'Completed', $_REQUEST) && $_REQUEST['Completed'] ? 1 : 0;

$todos = Table::newFromInstructionsFile( file: __DIR__ . '/../database/instructions/todos.php');
$record = [];
$record['ID'] = getNextId( table: 'todos');
$record['Title'] = $title;
$record['Completed'] = $completed;
$record['UserReference'] = $user['Reference'];

$todos->create($record);

sendOk($record);
```

```php
# toggle_all.php

require_once __DIR__ . '/../bootstrap.php';

use \DataManagement\Model\EntityRelationship\Table;

$user = initializeUserOrExit();

$completed = array_key_exists( key: 'Completed', $_REQUEST) && $_REQUEST['Completed'] ? 1 : 0;

$todos = Table::newFromInstructionsFile( file: __DIR__ . '/../database/instructions/todos.php');

$todos->update(function($record) use ($user) {
    if ($record['UserReference'] === $user['Reference']) {
        return Table::OPERATION_UPDATE_INCLUDE;
    }
}, function($record) use ($completed) { return ['Completed' => (int) $completed]; });

sendOk([]);
```

Dokumentārā lapa

Maģistra darbs “ Bezservera iebūvētas datubāzes kā alternatīva SQL ” izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota 21.05.2018.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____

(Autora paraksts un datums)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _____

(Vadītāja paraksts un datums)

Darbs iesniegts **maģistratūras sekretariātā** _____.

(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā. Studiju metodiķe: _____.

(Metodiķes paraksts)

Recenzents: _____

(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____

(Sekretāra paraksts)