Универзитет у Београду

Математички факултет

Мастер рад

Хоризонтално скалирање релационих система за управљање базама података

Студент:
Милан Јеремић

Ментор:
проф. др Саша Малков

Београд, 2022.

Комисија:

Ментор:	проф. др Саша Малков Математички факултет, Универзитет у Београду
Председник:	проф. др Ненад Митић Математички факултет, Универзитет у Београду
Члан:	др Ивана Томашевић Математички факултет, Универзитет у Београду

Датум одбране: _____

Апстракт

Значај теме и области

Релациони системи за управљање базама података (релациони СУБП или РСУБП) су веома битни у данашњем развоју софтвера, зато што омогућавају корисницима да приликом приступа подацима описују 'шта' желе да постигну, а не 'како'. Ради се о чврсто математички заснованом моделу података, што омогућава висок ниво аутоматизације. Скалирање база података подразумева повећавање капацитета и снаге СУБП кроз унапређивање рачунарског система на коме СУБП ради и практично је неопходно за велике базе података. У основне циљеве скалирања РСУБП спадају одржавање основних карактеристика РСУБП и реализација која је транспарентна у односу на корисника (тј. РСУБП се користи на исти начин као пре скалирања).

Скалирање РСУБП се може вршити на два начина: вертикално и хоризонтално. Вертикално је једноставније и обухвата повећање перформанси физичке (или виртуелне) машине на којој се софтвер извршава (додавање дискова за већи капацитет, додавање додатних чипова радне меморије или замена процесора процесором већих перформанси). Хоризонтално скалирање подразумева укључивање већег броја рачунара у рад јединственог СУБП и зато је сложеније - захтева промену архитектуре софтвера СУБП да би могао да искористи ресурсе више рачунара без нарушавања карактеристика РСУБП и уз очување транспарентности.

Специфични циљ рада

Циљ рада је истраживање и поређење карактеристика различитих приступа проблему хоризонталног скалирања СУБП. Биће разматрано више система, укључујући ‘Azure SQL Database *Hyperscale*’, ‘Azure SQL Synapse’ и ‘Google Spanner’. Рад ће обухватити и имплементацију прототипа механизма за очување конзистентности при физичком хоризонталном скалирању СУБП. Прототип ће покушати да очува и неке друге ACID карактеристике СУБП.

Овај рад посвећујем:

мојој Ленки и мојој Јаници,
родитељима Мири и Драганчету,
сестрама Милеви, Нати, **Кићи** и Неци,
остатку породице коју не можеш пребројати,
драгом професору који ме је трпео,
драгом професору који ме је подстакао,
као и многим другима...

Садржај

1. Увод

База података (енгл. *database*) је организована колекција података која се најчешће чува у рачунару. Систем за управљање базама података (СУБП, енгл. *database management system – DBMS*) је софтвер који омогућава скуп услуга за управљање базом података. Неке од битнијих услуга су:

- Чување, приступ и мењање података
- Подршка за трансакције и више корисника
- Опоравак базе података у случају наглог престанка рада СУБП-а или губитка базе података (на пример, губитак диска на коме су датотеке базе података)
- Сигурност кроз могућност дефинисања права приступа различитим корисницима система

Модел података дефинише логичку структуру података, односно објашњава како се подаци чувају, како им се приступа и како се њима манипулише. Примери модела података су: релациони модел (енгл. *relational model*), модел кључ-вредност (енгл. *key-value data model*), модел докумената (енгл. *document model*), хијерархијски модел (енгл. *hierarchical model*), модел објеката (енгл. *object model*), итд.

Релациони СУБП (РСУБП) је СУБП заснован на релационом моделу. Релациони модел је математички заснован модел података који је дефинисао Едгар Код 1970. године, као истраживач ИБМ лабораторије [1]. РСУБП омогућава корисницима да описују 'шта' би требало да се постигне са подацима, а не 'како', често кроз декларативни упитни језик *SQL* (*Structured Query Language*).

У контексту база података, трансакција је скуп операција, која је део једне логичке операције над подацима, који задовољава *ACID* својства. *ACID* својства гарантују исправност података у случају грешака, наглог престанка рада СУБП-а, итд. *ACID* својства су:

1. **Атомичност** (енгл. *atomicity*) – скуп операција дефинисана трансакцијом се примењује у целости на податке или се не примењује уопште. На пример, у случају наглог престанка рада СУБП-а, база података мора бити у стању као да се недовршене трансакције (у тренутку наглог престанка рада СУБП-а) нису ни десиле.
2. **Конзистентност** (енгл. *consistency*) – трансакција мора да преводи базу података из једног валидног стања у друго валидно стање.
3. **Изолација** (енгл. *isolation*) – постоји више нивоа контроле изолације. Најстриктнији ниво дефинише да истовремено извршавање трансакција оставља базу у истом стању као да су те трансакције извршене секвенцијално.
4. **Трајност** (енгл. *durability*) – када се трансакција заврши, њене промене у бази података ће наставити да постоје чак и у случају отказа СУБП-а (на пример нестанак струје на рачунару на коме је СУБП покренут). Ово се обично постиже уписивањем промена на бази података у трајну меморију.

1.1 Скалирање СУБП-а

Скалирање СУБП-а подразумева повећавање капацитета и снаге СУБП-а кроз унапређивање рачунарског система на коме СУБП ради и практично је неопходно за велике базе података. Постоје два типа скалирања СУБП-а:

1. Вертикално скалирање
2. Хоризонтално скалирање.

Вертикално скалирање обухвата повећање перформанси физичке машине на којој се софтвер извршава. Ако администратори СУБП-а имају приступ физичкој машини, вертикално скалирање би могли урадити тако што би променили процесор на процесор са више језгара, додали нови процесор уколико то матична плоча подржава, додали још чипова радне меморије, додали још дискова, итд. У овом процесу често има доста мануелног посла, посебно код одржавања расположивости СУБП-а. У облаку (енгл. *cloud*) су ствари мало другачије зато што је ниво физичке машине апстрахован и корисници (администратори или инжењери СУБП-а) имају приступ само виртуелној машини. Виртуелне машине такође подржавају скалирање рачунске и складишне моћи, с тим што је то значајно лакше него 'ручно' на физичком нивоу. У облаку се кроз пар минута виртуелној машини може доделити други процесор, већа количина радне меморије или диск већег капацитета и перформанси на виртуелној машини.

Међутим, проблем настаје када је потребно скалирати преко могућности једне виртуелне машине. У том случају је очигледно да се дизајн СУБП-а мора променити да би се на прави начин искористила моћ више виртуелних машина.

Хоризонтално скалирање подразумева укључивање већег броја чворова у рад јединственог СУБП-а па је зато и сложеније [2]. Односно, захтева промену архитектуре софтвера СУБП-а да би се искористили ресурси више рачунара. Један од честих циљева хоризонталног скалирања СУБП-а је одржавање основних *ACID* својства трансакција. Чест циљ је и да скалирање СУБП-а буде транспарентно, односно да се СУБП користи на исти начин као пре скалирања.

Приликом хоризонталног скалирања СУБП-а, различити чворови система могу имати приступ свим подацима, али и не морају. Такође, подаци могу имати више копија на различитим чворовима система. За овакве податке се каже да су реплицирани, односно свака копија се зове реплика. У овом контексту, постоје следеће архитектуре СУБП-а:

1. архитектуре **без дељења података** (енгл. *shared nothing*) – чворови система могу садржати:
    a. **потпуну копију података** – чворови система садрже целу копију базе података.
    b. **фрагменте података** (енгл. *fragment*) – чворови система немају све податке, већ само неки њихов фрагмент. Поделу на фрагменте корисник може спецификовати, а може и сам систем имати неки алгоритам којим одређује аутоматску поделу података на фрагменте. У зависности од имплементације система, чворови могу приступити подацима других чворова, али то неретко доводи до значајно мањих перформанси целокупног система.[1]
2. архитектуре **дељених података** (енгл. shared data) - у овим системима постоји засебан чвор или скуп чворова који су задужени за складиштење података. Остали чворови система који раде обраду података контактирају чворове за складиштење да би дошли до података.

Један од видова хоризонталног скалирања јесте додавање нових чворова система који садрже још једну копију података у кластер ради високе расположивости. Примарна реплика увек омогућава писање, док остале, секундарне, реплике могу омогућавати:

1. **читање** – примарна реплика шаље све промене над базом података свим секундарним репликама.
2. **писање** – такозвани мулти-мастер системи (енгл. *multi master*). Главни проблеми у оваквим системима су разрешавање конфликта (више чворова мења исти ресурс у исто време) и скалирање конзистентног читања.

Када реплика врши писање, односно жели да промени нешто на бази података, мора да комуницира са осталим чворовима ову намеру да би и други чворови применили промену и тиме имали конзистентне податке. Промена на бази података се може проследити свим репликама и тек када све реплике потврде промену операција се може сматрати успешном. Међутим, због перформанси се често чека потврда само одређеног број чворова, који се назива кворум (енгл. *quorum*). Број чворова у кворуму се често бира као број који је већи од половине укупног броја чворова.

Реплике се могу укључити у кластер у различитим режимима репликације. Режими репликације могу бити:

1. **синхрони** – ове реплике учествују у кворуму
2. **асинхрони** – ове реплике не учествују у кворуму и оне прихватају само промене које је кворум прогласио валидним. Ово значи да овакве реплике често немају најновије податке, чега корисници морају бити свесни приликом читања са њих. Стандардна пракса је да реплике које се налазе на другој географској локацији раде у асинхроном режиму репликације.

[1] Често овакви системи захтевају да се наведе кључ под којим су подељени подаци, што повлачи то да корисник мора променити своју апликацију у случају да апликација већ не ради са оваквом претпоставком. Ово ограничење је често велики камен спотицања за кориснике због нетривијалног посла промене апликација.

По нивоу апстракције, репликација приликом хоризонталног скалирања се може радити на следећим нивоима:

1. **физички ниво** – пример овог вида репликације је кластер високе расположивости. Овде се репликација ради на нивоу страница података и дневника трансакција, односно дневник трансакција је тај који се шаље осталим репликама. Пошто реплике имају исту копију података, морају имати и страницу на коју дневник трансакција реферише, па тај дневник може и применити.
2. **логички ниво** – ниво корисничког погледа на базу података и то је ниво упита, редова, колона, табела, шема, итд. Пример логичког вида репликације јесте трансакциона репликација компаније *Microsoft*. Уместо дневника трансакција, на секундарне чворове се шаљу редови који су промењени [3].

1.2 Примери хоризонталног скалирања

У наставку овог поглавља ће бити представљени примери различитих продукционих система из индустрије који имају различите приступе хоризонталном скалирању са идејом да се наведу неке кључне разлике, јер детаљнији опис сваког од система би превазишао обим овог рада. Такође, издвајање ових система не значи да слични системи који покривају исте сценарије не постоје и од других компанија. Главни производ који ће бити дискутован је *Azure SQL Database Hyperscale*, о коме ће бити много више речи у поглављу 2.

1.2.1 *Azure SQL Synapse*

Azure SQL Synapse је фамилија од два производа компаније *Microsoft* која је прављена за аналитички тип коришћења база података (енгл. *OLAP – On-Line Analytical Processing*) [4]. У овим производима постоје различити типови чворова:

1. **Контролни чвор** (енгл. *control node*) обрађује корисничке упите и прикупља резултате од осталих чворова. Уколико се извршавање упита процени јефтиним, контролни чвор ће извршити упит, а у супротном ће упит бити преусмерен на арбитра.
2. **Арбитар** је специјализован чвор за прављење дистрибуираног плана извршавања и корисник њему нема приступ. Када направи план, арбитар шаље рачунским чворовима делове посла који треба одрадити у виду упита. Такође, дистрибуираним планом извршавања је урачунато евентуално прослеђивање резултата једног рачунског чвора другом. Арбитар има специјализовано име зависно од производа.
3. **Рачунски чвор** (енгл. *compute node*) – корисник нема директан приступ овим чворовима и они врше већину процесирања упита у систему по инструкцијама арбитра.

Производи који постоје су:

1. ***Dedicated SQL Pool*** [4] [5] – рачунски чворови су посвећени једној бази података и корисник је тај који дефинише број рачунских чворова. Архитектура је без дељења података где сваки рачунски чвор има приступ неком фрагменту података.
2. ***Serverless SQL Pool*** [4] [6] – рачунски чворови се додељују упитима по потреби. Архитектура је архитектура дељених података.

1.2.2 *Apache Cassandra*

Apache Cassandra је иницијално направљена од стране компаније *Facebook*, и сада је база података отвореног кода [7]. Производ је направљен по узору на производе *Dynamo* компаније *Amazon* и *BigTable* компаније *Google*. Подаци су подељени на фрагменте. Сваки фрагмент има више реплика и све реплике су распоређене на различите чворове система који могу бити и на другој географској локацији. Систем је веома добар у хоризонталном скалирању на глобалном нивоу где је очувано линеарно скалирање. Систем је оријентисан ка томе да корисник наведе кључ који је коришћен приликом дељења података на фрагменте. Подржан је мулти-мастер где је стратегија разрешавања конфликта 'последњи упис побеђује'. Систем не поштује *ACID*. Тачније, не поштује стриктну конзистентност, већ поштује карактеристику одложене конзистентности (енгл. *eventual consistency*). Такође, упити изван једне партиције нису подржани, као ни страни кључ, односно референтна конзистентност.

1.2.3 *Google Spanner*

Spanner је производ компаније *Google* где је главни циљ глобално хоризонтално скалирање СУБП-а [8]. Као и у другим системима, подаци су подељени на фрагменте који су дистрибуирани на више географских локација. Унапређење у односу на производ *Apache Cassandra* је да су овде подржане трансакције које обухватају више фрагмената, као и страни кључ, док су остале карактеристике концептуално исте.

Spanner одржава својство екстерне конзистентности на свим географским локацијама. Односно, ако се једна трансакција која врши упис успешно заврши на једном чвору, та промена ће одмах бити видљива свим чворовима система. Ово својство је у дистрибуираном систему тешко одржати. На пример, у кластеру високе расположивости са асинхроним режимом репликације ово својство није испоштовано.

Један од главних компоненти преко којих *Spanner* постиже екстерну конзистентност јесте систем *TrueTime*. Овај систем омогућава да највећа разлика времена на свим чворовима СУБП-а буде максимално 7 милисекунди. Ради поређења, комуникација између чворова система који су на различитим географским локацијама би била око 100 милисекунди. Свака географска локација има засебан *TrueTime* систем који се синхронизује са другим географским локацијама преко глобалног позиционог система. У оквиру истог центра за податке (енгл. *datacentar*) постоје атомски часовници који одржавају тачно време које се често проверава са временом глобалног позиционог система. Сваки чвор система дохвата од свог локалног атомског часовника тачно време. Ако систем зависи од тачног времена, онда веома битна метрика јесте несигурност у тачно време, односно одступање од тачног времена. Несигурност у тачно време је збир несигурности:

1. атомског часовника и његовог освежавања од глобалног позиционог система[2],
2. временски распон освежавања чвора, односно време трајања позива за синхронизацију времена од атомског часовника до чвора[3],
3. кварцног часовника на самом чвору, односно нека функција времена протеклог од последњег освежавања времена са атомским часовником[4].

---

[2] Веома мала несигурност. Несигурност се мери у микросекундама.
[3] Ред величине је 1 милисекунда.
[4] Кварцни часовник који се налази на сваком чвору има релативно велику непрецизност, која је око 200 микросекунди по секунди.

За решавање проблема дистрибуираног писања на више фрагмената *Spanner* користи познати алгоритам уписа из две фазе (енгл. *two-phase commit*). Један од чворова система се одабере као координатор. Као део прве фазе фрагменти који учествују у трансакцији закључају потребне редове и обећају да могу да одраде трансакцију. Након што сви обећају, почиње друга фаза, када координатор трансакције потврди свима да трансакцију заврше. Ово би било довољно за одржање екстерне конзистентности да *Spanner* подржава само операције писања и читања са закључавањем.

Међутим, једна од главних могућности које *Spanner* пружа јесте скалирање читања под изолацијом снимка стања (енгл. *snapshot isolation*). Овај систем у традиционалном РСУБП-у се ради тако што се сачува верзија базе података у тренутку када је трансакција започета. Међутим, у дистрибуираном систему, по истом принципу ово би било могуће само ако један чвор јави свим осталима коју верзију базе података да сачувају, али то није оптимално. Уместо тога, у *Spanner* систему сваки фрагмент чува све своје верзије за неки претходни период. Када се покрене трансакција која врши само читање под изолацијом снимка стања на неком чвору, тај чвор проследи време када је та трансакција започета и сви остали чворови испоручују верзију података из задатог времена (што је и могуће због чувања претходних верзија фрагмената).

Међутим, ово није довољно за одржање екстерне конзистентности. На крају друге фазе уписа, не постоји гаранција да ће се на свим фрагментима упис десити у исто време. Ако се на једном чвору потврдила трансакција уписа, а на другом чвору није још увек, онда може постојати друга трансакција која врши само читање (под изолацијом снимка) и чита са оба чвора истовремено. Оваква трансакција ће прочитати вредности које нису конзистентне. Односно, једна вредност ће бити пре краја трансакције уписа, а друга након краја трансакције уписа.

Да би се претходни проблем решио, приликом уписа координатор трансакције прослеђује време завршетка трансакције уписа свим чворовима система. Време завршетка трансакције се мора одабрати након завршетка прве фазе уписа, односно када су сви чворови обећали да могу да потврде трансакцију и, што је још битније, држе тражене редове закључаним. Главни трик којим се добија екстерна конзистентност јесте да сваки од чворова који учествује у писању одложи потврду трансакције до (претходно одабраног) времена завршетка трансакције уписа. Приликом овог одлагања се мора урачунати и несигурност у тачно време. Овим се ефективно трансакција потврђује у прошлости, зато што увек постоји нека несигурност у тачно време. Главна идеја је да се овим чекањем осигура да било која трансакција (која врши само читање под изолацијом снимка) на било ком чвору, буде засигурно након тачке потврде трансакције уписа, што је и дефиниција екстерне конзистентности.

1.2.4 *Amazon Aurora*

Aurora је производ компаније *Amazon* и оно што га издваја од осталих производа јесте подршка за мулти-мастер [9] [10]. Мулти-мастер се постиже тако што је успостављен кластер високе расположивости између реплика и приликом уписа свака реплика мора то најавити свим другим репликама. Упис се ради директно у складиште и то на 6 независних реплика одједном. Када 4 од тих 6 реплика одговори потврдно, сматра се да је тај упис успешан. Након ове потврде складишног слоја, прослеђује се промена осталим чворовима система да освеже свој бафер страница. Проблем уписа над једном страницом података од стране више чворова се решава тако што је логика за разрешавање конфликта у складишном слоју. Пошто је складишни слој одговоран за странице и увек има најновије верзије страница, може дати одговор да је одређена страница већ на новијој верзији и да одбије упис. У случају одбијања трансакције враћа се грешка да се десило узајамно блокирање (енгл. *deadlock*), да би корисник могао да реагује.

Негативна страна производа *Aurora* јесте да корисник мора да преусмери уписе са једног чвора на други у случају престанка расположивости једног од чворова, пошто чворови система имају засебне крајње тачке (енгл. *endpoint*). Такође, корисник мора да брине о избегавању конфликта, тиме што ће трансакције које гађају исту табелу груписати на један чвор. Ако се иста табела мења са више чворова, може доћи до узајамног блокирања а тиме и неуспешних трансакција које имају утицај на корисничку апликацију.

2. *Azure SQL Database Hyperscale*

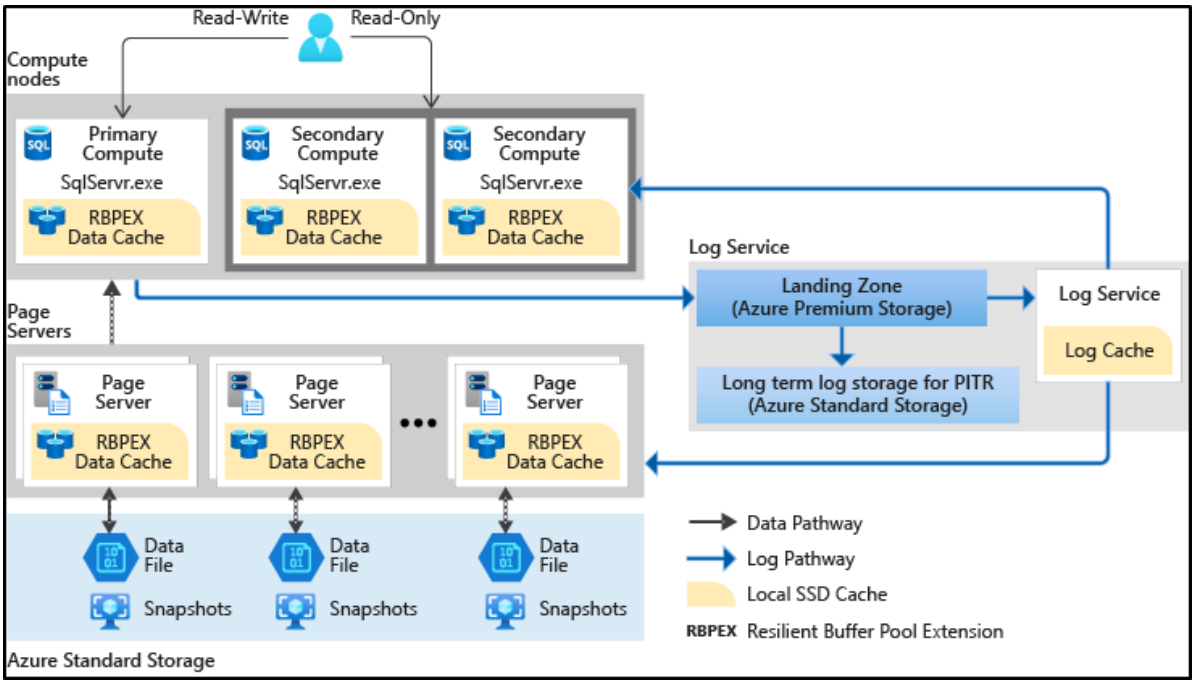
Azure SQL Database Hyperscale је производ компаније *Microsoft* који је направљен као надоградња на традиционални РСУБП исте компаније, *SQL Server*. Главни циљ је повећање флексибилности, односно независно скалирање рачунске (енгл. *compute*) и складишне (енгл. *storage*) моћи до стотину терабајта података [11] [12]. Флексибилност је битна зато што би неки корисници желели да имају базу података са веома малом рачунском моћи, а са огромном складишном моћи за податке који се ретко користе, а подаци морају бити расположиви. И обратно, са огромном рачунском моћи, а веома малом складишном моћи за извршавање рачунски захтевних упита. Битно је нагласити да је овај производ прављен за трансакциони тип коришћења база података, а не за аналитички тип коришћења података.

2.1 Архитектура

Да би се постигло ефективно скалирање на више виртуелних машина *Hyperscale* уводи концепт различитих типова чворова система:

- **рачунски чвор** (енгл. compute node),
- **сервис дневника трансакција** (енгл. log service)
- **сервер страница** (енгл. page server)

Треба нагласити да постоји **дељена област непосредног писања дневника трансакција** (енгл. *landing zone*, у даљем тексту **област писања**) која формално припада сервису дневника трансакција, али је најобичнија датотека на дељеном складишту којој има директан приступ за упис и примарни рачунски чвор. Дијаграм 1 показује архитектуру *Hyperscale* која ће бити детаљно разматрана у наредним одељцима.



Дијаграм 1. Архитектура Hyperscale.

2.1.1 Рачунски чвор

Рачунски чвор је главна улазна тачка у СУБП *Hyperscale*. Кориснику је коришћење система транспарентно, у смислу идентично као да је конекција направљена ка традиционалном РСУБП-у. Корисник система не зна да ли ће се његов упит извршити само на рачунском чвору или ће рачунски чвор контактирати још неки чвор система, што указује на очување транспарентности.

Рачунски чвор је пандан традиционалном РСУБП-у уз неколико разлика. Кључна разлика је то што рачунски чвор не чува стање. Тачније, он нема приступ трајној меморији за читање (диск или *Azure* складиште - решење компаније *Microsoft* за складиште у облаку, у наставку текста 'складиште у облаку'), односно нема датотеку из које би стање било враћено у случају наглог престанка рада рачунског чвора.

У случају традиционалног РСУБП-а упис новог записа у дневник трансакција би се вршио директно у трајну меморију. Слично, у случају система *Hyperscale*, тај позив би се одрадио директно у област писања. Област писања је датотека на складишту која чува дневник трансакција. Међутим, читање дневника трансакција се не ради директно из те датотеке на складишту у облаку, већ би рачунски чвор дохватио потребан део дневника трансакција од сервиса дневника трансакција. У чистијој архитектури сервис дневника трансакција би био сервис који искључиво комуницира са облашћу писања, али је због перформанси одабрано да примарни рачунски чвор директно уписује у област писања пошто је упис у дневник трансакција једна од најкритичнијих операција за брзину СУБП-а.

Традиционални РСУБП има на располагању трајну меморију у коју врши упис страница података. Са друге стране, примарни рачунски чвор у архитектури *Hyperscale* не врши упис странице у трајну меморију, јер није неопходно пошто исте увек може дохватити од сервера страница. Више о томе како се врши упис страница у трајну меморију у архитектури *Hyperscale* ће бити у поглављу 2.1.3.

Након наглог престанка рада СУБП-а, као део опоравка стања (енгл. *recovery*) потребне су две главне операције: дохватање активног дела дневника трансакција и дохватања потребних страница из трајне меморије које су неопходне активном делу дневника трансакција. У традиционалним РСУБП-има, активни део дневника трансакција и потребне странице би биле допремљене са диска или складишта у облаку, али с обзиром да то није доступно у архитектури *Hyperscale*, активни део дневника трансакција се допрема од сервиса дневника трансакција, а потребне странице се дохватају од одговарајућих складишних сервера. Да би се избегла висока цена дохватања страница од сервера страница, рачунски чворови имају бафер страница (енгл. *buffer pool*) (као и традиционални РСУБП), али имају и отпорну екстензију бафера страница (енгл. *resilient buffer pool extension – RBPEX*, у наставку текста 'бафер страница на диску'). Бафер страница на диску користи локални диск као екстензију радне меморије за странице базе података попут оперативних система који то раде за странице радне меморије (енгл. *paging*) [13].

У систему *Hyperscale* може бити један рачунски чвор или их може бити више. Систем тренутно подржава само један рачунски чвор који подржава писање (примарна реплика), док остали рачунски чворови подржавају само читање (секундарне реплике). Да би секундарне реплике одржавале конзистентност, у традиционалном РСУБП-у би примарна реплика слала све промене на све остале реплике. У систему *Hyperscale* је могуће додати овакве реплике и оне се називају реплике високе расположивости (енгл. *high availability replica*) [14]. Међутим, код система *Hyperscale* се могу додати и такозване именоване реплике (енгл. *named replica*) [15]. Именоване реплике имају засебне крајње тачке. Свака именована реплика је задужена да одржава најновију верзију дневника трансакција, тиме што ће га дохватати од сервиса дневника трансакција и потребне странице од складишних сервера. На овај начин именоване

реплике имају мање утицаја на перформансе примарног рачунског чвора. Именоване реплике садрже трансакционо конзистентне податке, али не морају да садрже најновије податке, односно читање се врши под изолацијом снимка стања (енгл. *snapshot isolation*) [16].

2.1.2 Сервис дневника трансакција

Главна функција сервиса дневника трансакција је да обрађује дневник трансакција СУБП-а. Обрада подразумева читање дневника трансакција из области писања, испоручивање дневника трансакција осталим чворовима система којима је дневник трансакција потребан, као и отпремање дневника трансакција са брзог и скупог складишта у облаку на спорије и јефтиније складиште у облаку.

Претплатници сервиса дневника трансакција су сви остали чворови система. Овај концепт већ постоји и у традиционалном РСУБП-у, где се различити сегменти СУБП-а претплате на дневник трансакција (резервна копија дневника трансакција, репликација на секундарни чвор, прављење резервне копије базе података, итд.) [17]. Као што је раније поменуто, примарном рачунском чвору је потребан активни део дневника трансакција након наглог прекидања рада да би могао да одради опоравак. Секундарним чворовима је потребан дневник трансакција за нормално оперисање и њега дохватају од сервиса дневника трансакција. Такође, сервери страница дохватају дневник трансакција од сервиса дневника трансакција да би одржавали најновије верзије страница података које припадају њима, о чему ће бити више речи у поглављу 2.1.3.

Сервис дневника трансакција је задужен да прави резервне копије дневника трансакција на јефтинијем и спорије складишту у облаку, што је еквивалент резервној копији дневника трансакција у традиционалном РСУБП-у (енгл. *log backup*), иако се ради на мало другачији начин. У СУБП-у *SQL Server*, корисник система мора да изврши команду за прављење резервне копије дневника трансакција. Такође, у облаку где је платформа задужена за резервне копије дневника трансакција, платформа најчешће има засебан сервис који је задужен за резервне копије дневника трансакција. Када неки део дневника трансакција више не би био потребан свим претплатницима, онда би тај део дневника трансакција био обрисан [17]. Део дневника трансакција који се брише мора бити најстарији, односно почетак дневника трансакција. Поред прављења резервне копије дневника трансакција, сервис дневника трансакција има додатну дужност да обрише дневник трансакција коме је направљена резервна копија. То може бити одрађено одмах по прављењу резервне копије, али често је повољније да сви претплатници назначе који им је део дневника трансакција потребан да би сервис дневника трансакција могао да направи исправнију одлуку која би избегла плаћање цене приступа спорије складишту у облаку. У случају активне базе, обрисани део дневника трансакција биће потребан само у ретким случајевима[5].

Да би дневник трансакција могао бити испоручен другим чворовима система, сервис дневника трансакција мора прво да учита сегменте дневника трансакција у своју радну меморију интерпретирањем области писања. Пошто два чвора система (примарни рачунски чвор и сервис дневника трансакција) имају приступ области

[5] У СУБП-у *SQL Server*, као и у систему *Hyperscale*, уколико секундарни чвор изгуби претплату на дневник трансакција, на поновном успостављању претплате ће морати да крене изнова са копирањем података у случају да је потребни део дневника трансакција обрисан на примарном чвору.

писања, може доћи до неправилног коришћења овог дељеног ресурса из два независна процеса. Ово се решава једноставно тиме што се успостави инваријанта да је примарни рачунски чвор једини који уписује и то на крај дневника трансакција (енгл. *append only*), а сервис дневника трансакција само чита дневник трансакција и брише неактивне сегменте дневника трансакција са почетка дневника трансакција (након што је тај део дневника трансакција отпремљен на складиште у облаку).

Ради оптимизације времена испоруке дневника трансакција другим чворовима система, сервис дневника трансакција има кеш дневника трансакција у меморији, али и на локалном диску, слично бафер страница на диску.

2.1.3 Сервер страница

Главна дужност сервера страница је да испоручује странице података које су потребне рачунским чворовима. Када рачунски чвор тражи једну или више страница од сервера страница, он мора навести и редни број дневника трансакције (енгл. *log sequence number*) на којем тај рачунски чвор оперише. Свака страница има редни број дневника трансакција који је последњи примењен на ту страницу. Ако сервер страница већ поседује страницу са редним бројем дневника трансакција већим или једнаким траженом редном броју странице, онда сервер страница може одмах испоручити ту верзију странице. У супротном случају сервер страница мора да контактира сервис дневника трансакција и затражи најновији сегмент дневника трансакција, који ће применити на страницу коју има, да би је ажурирао до верзије која је потребна рачунском чвору.

Складишта сервера страница су ограничена на 128 гигабајта или на 1 терабајт. Да би се постигле величине база од 100 терабајта (односно веће од величине једног сервера страница) потребно је да постоји више сервера страница. Раст складишне моћи базе података се своди на додавање нових сервера страница, а смањивање складишне моћи се своди на избацивање сервера страница. Пре избацивања неког сервера страница, подаци морају бити премештени са сервера страница који се избацује на неки други активан сервер страница. У случају више сервера страница, рачунски чворови морају да знају које странице су на ком серверу страница.

Перформансе система би биле ослабљене уколико би се допремање најновијег дневника трансакција од сервиса дневника трансакција радило само када је то потребно неком рачунском чвору. Због тога сервери страница периодично довлаче део дневника трансакција који је битан за њих.

Сервер страница не мора да примењује логичке записе из дневника (трансакција је започета, трансакција је завршена, итд.), зато што је задужење сервера страница да испоручује странице, а не да проверава конзистентност трансакција и друге логичке операције. Због тога, сервер страница само примењује физичке записе из дневника који мењају саме странице.

Складишта у облаку која користе сервери страница за трајно чување података су јефтино и споро складиште. Ради перформанси испоручивања страница рачунским чворовима, сервери страница имају бафер страница на диску (слично рачунским чворовима), који садржи све странице које су додељене том серверу страница. Идеја је да се цена приступа спором складишту надомести брзим кешом. Занимљиво је да приликом примењивања дневника трансакција, сервер страница не мора да уписује резултат директно на складиште, већ га може уписати у бафер страница (било у меморији или на локалном диску) и испоручити га рачунским чворовима.

Након наглог престанка рада сервера страница, ако се сервер страница није покренуо на истој виртуелној машини и нема приступ претходном баферу страница на

диску, онда се бафер страница на локалном диску мора поново попунити да би се убрзао опоравак. Због тога је упис у складиште и даље неопходан. Да би се додатно поправиле перформансе губљења бафера страница на диску, *Hyperscale* за сваки сервер страница има две потпуно независне реплике. У случају наглог или планираног престанка рада једне од реплика, друга ће бити расположива. Ова додатна реплика није потребна за одржавање конзистентности, већ је ту углавном због перформанси.

Сервер страница држи резервацију дневника трансакција код сервиса дневника трансакција са редним бројем дневника на коме он оперише. Ова резервација се може померити са једног редног броја дневника трансакција на други само након што сервер страница упише промене у складиште.

2.1.4 Резервне копије и опоравак стања

Прављење резервних копија (енгл. *backup*) и опоравак стања (енгл. *restore*) [12] [18] су веома битне функционалности СУБП-а. У случају да корисник случајно обрише неку табелу или дође до нарушавања конзистентности података или губитка диска или складишта у облаку где се налазе подаци, могућност опоравка је веома битна. Да би опоравак био могућ, неопходно је прављење резервних копија базе података на независан диск или складиште података у облаку.

СУБП *SQL Server* ово постиже кроз више операција. Прво направи резервну копију базе података (енгл. *full backup*), која се узима ређе, на пример на седам дана. Потом се прави резервна копија дневника трансакција (енгл. *log backup*). Та операција се ради значајно чешће, рецимо на пет минута. Након тога се може направити и копија промењених страница од претходне резервне копије целе базе, односно може се направити инкрементална резервна копија (енгл. *differential backup*), која се ради, на пример, на дванаест сати. У СУБП-у *SQL Server* инкрементална резервна копија се увек ради у односу на резервну копију целе базе података[6]. Све резервне копије једне базе чине ланац резервних копија (енгл. *backup chain*). У облаку се датотеке базе података чувају на брзом и скупом складишту, док се резервне копије чувају на спором и јефтином складишту због мање учесталости приступа.

Опоравак стања у одређену тачку у времену се врши тако што се примени прва доступна резервна копија базе података пре тражене тачке у времену. Након тога се примени последња инкрементална резервна копија пре тражене тачке у времену, а након одабраних резервних копија базе података (ако постоји). Затим се на основу редног броја дневника трансакција, који се меморише у мета подацима сваке датотеке резервне копије, идентификује резервна копија дневника трансакција која следи по редном броју дневника трансакција. Затим се наставља примена низа резервних копија дневника трансакција све док се не достигне тражена тачка опоравка. Овако формиран низ се зове план опоравка стања (енгл. *restore plan*). Применом резервних копија из плана опоравка добијамо верзију базе података која је била у тренутку тражене тачке у времену. Треба нагласити да је прављење инкременталних резервних копија и њихова примена опциони и постоје због подизања перформанси операције опоравка. Операција опоравка може функционално радити и ако су на располагању само резервна копија базе података и резервна копија дневника трансакција.

Hyperscale прави резервне копије на мало другачији начин. Уместо преписивања података из датотека базе података у датотеке резервних копија на другом складишту, у овом систему се користи механизам снимка стања складишта (енгл. *snapshot*). Ово је

[6] У општем случају се инкременталне резервне копије могу радити и у односу на претходне инкременталне резервне копије.

могућност коју пружа складиште у константном времену, зато што се операција своди на операцију над мета подацима од стране складишта и нема физичког преписивања података. Снимак стања на складишту постаје нови ентитет из погледа корисника (сервера страница), а подаци се у позадини не дуплирају, већ се методом показивача одржавају одговарајуће слике података [19]. Као што је већ установљено, датотеке базе података су на јефтином и спором складишту (са неколико нивоа кеширања испред), онда ће и датотека снимка стања бити на јефтином складишту. Пошто резервне копије имају потенцијално много података оне би и требало да буду на јефтином и спором складишту, што се снимком стања и добија.

Као што је већ поменуто, сервис дневника трансакција је задужен за отпремање дневника трансакција на спорије складиште и он је задужен за расположивост дневник трансакција. Корисник система може да одреди колико дуго ће се чувати резервне копије. То имплицитно дефинише и колико дуго би требало сервис дневника трансакција да чува дневник трансакција. Брисање непотребног дела дневника трансакција са спорог складишта у облаку је још једна дужност сервиса дневника трансакција која није поменута раније.

Инкременталне резервне копије не постоје, зато што се оне имплицитно добијају од складишта кроз узимање снимака стања. Снимци стања по дефиницији чувају само шта се променило у односу на претходни снимак стања или оригиналну датотеку [19].

Приликом операције опоравка стања, сваки сервер страница проналази први старији снимак стања који припада њему и тај снимак преписује на одговарајућу локацију у новој *Hyperscale* бази података. Предност преписивања снимка стања је што се оно обавља у константном времену, јер је то још једна од могућности које складиште у облаку пружа. Иза сцене ће складиште преписати датотеку, али са стране корисника (сервера страница) се операција завршава у константном времену. Сервис дневника трансакција мора дохватити део дневника трансакција који је релевантан за опоравак. Релевантни део дневника трансакција почиње најстаријим редним бројем трансакција одабраних снимака стања свих сервера страница, а завршава се одабраном тачком опоравка. Релевантни део дневника трансакција се преписује на област писања новокреиране базе података са сервиса дневника трансакција изворне базе података. Уколико изворни сервис дневника трансакција не постоји, у сценарију неповратног престанка рада или опоравка обрисане базе података, сервис дневника трансакција новокреиране базе података приступа директно датотекама који садрже дневник трансакција и преписује га у датотеке које припадају њему. У том тренутку сервери страница имају неку верзију података, а сервис дневника трансакција релевантни део дневника трансакција да ти сервери страница уз примену дневника трансакција дођу до тражене тачке опоравка. Једино преостало је да се дода рачунски чвор у систем да би ово постала функционална *Hyperscale* база података. Додавање рачунског чвора је поприлично једноставно и не разликује се много од сценарија када је систем већ функционалан и постојећи рачунски чвор нагло престане са радом и поново се покрене (енгл. *crash recovery*).

2.2 Одржавање *ACID* карактеристика трансакција

2.2.1 Атомичност

Систем *Hyperscale* одржава атомичност по истом принципу као и СУБП *SQL Server*. Пошто је рачунски чвор задужен за процесирање упита, на њему остаје и одговорност одржавања атомичности. Осталим чворовима система (сервис дневника трансакција и сервер страница) концепт трансакција ни не треба да буде познат, пошто је то логичка операција, а не физичка.

2.2.2 Конзистентност

Конзистентност треба посебно посматрати на логичком, а посебно на физичком нивоу.

Конзистентност на логичком нивоу по дефиницији је карактеристика да се база података мора превести из једног валидног стања базе у друго валидно стање. Одржавање ове карактеристике се ослања на одржавање атомичности и изолације, али подразумева и мноштво других провера као што су провере окидача, каскадне примене окидача, ограничења и њихове комбинације. У систему *Hyperscale* то није много другачије, пошто се све провере и дешавају на примарном рачунском чвору.

Једна логичка операција која је дефинисана трансакцијом, као што је рецимо додавање једног реда у табелу, се може превести у више физичких операција. Тај ред се мора уписати у физичку структуру где се подаци чувају, што је често једна физичка операција. Међутим, у случају да на тој табели постоји више индекса, ти индекси се такође морају ажурирати, што су додатне физичке операције.

Конзистентност на физичком нивоу се одржава тако што се приликом примене дневника трансакција СУБП мора постарати да се трансакције примењују на странице у редоследу у којем су иницијално направљене. У СУБП-у *SQL Server* се ово лако постиже зато што је тачно један процес задужен за опоравак стања. У систему *Hyperscale* сличну централну улогу има примарни рачунски чвор. Међутим, додатна сложеност је то што примарни рачунски чвор тражи од других чворова система странице и дневник трансакција, па се мора више пазити приликом имплементације оваквих система. Када рачунски чвор не би тражио страницу са одређеним редним бројем трансакције и сервер страница не би гарантовао да је то страница на траженом (или новијем), односно ажурном, редном броју трансакција, рачунски чвор би лако могао доћи до погрешних закључака о валидности достављене странице. То би скоро сигурно брзо довело до физичког нарушавања конзистентности, што повлачи и логичко нарушавање конзистентности. Такође, сви чланови система морају да раде оно што гарантују. На пример, сервис дневника трансакција неће испоручивати дневник трансакција са недостајућим трансакцијама у неком распону. Ово је неопходно да би се осигурало да је целокупан систем, а са њим и база, из перспективе корисника система конзистентна.

2.2.3 Изолација

Hyperscale одржава изолацију на исти начин као и *SQL Server* кроз системе закључавања на различитим нивоима. Пошто је рачунски чвор тај који је задужен за процесирање упита, на њему остаје и одговорност одржавања изолације. Осталим чворовима система (сервис дневника трансакција и сервер страница) концепт закључавања ни не треба бити познат, пошто је то логичка операција, а не физичка. Секундарни рачунски чворови омогућавају само читање у изолацији снимка, односно омогућавају читање само потврђених трансакција[7] у верзији података коју тренутно имају.

[7] Секундарни чворови добијају информацију о томе које су трансакције потврђене преко дневника трансакција.

2.2.4 Трајност

Традиционални РСУБП одржава трајност тако што се ослања на одговорност слоја испод себе да ће се упис у трајну меморију десити. У случају СУБП-а са локалним фајловима, слој испод СУБП-а је систем датотека оперативног система, док у систему *Hyperscale* су различити чворови система одговорни за одређени део одржавања трајности.

Примарни рачунски чвор је задужен за трајност најновијих записа у дневнику трансакција. Међутим, сервис дневника трансакција такође дели одговорност трајности дневника трансакција, зато што је његова одговорност да измести дневник трансакција са брзог на споро складиште у облаку и при тим операцијама мора гарантовати трајност података. Што се тиче трајности самих датотека које садрже странице података, за њих су задужени сервери страница, који рачунају на складиште у облаку на којем су саме датотеке да гарантују трајност података.

Механизам којим традиционални РСУБП одржава перформансе приликом одржавања критеријума трајности је да се промене прво упишу у дневник трансакција и да се странице промене у меморији. Након тога се странице спуштају у трајну меморију независно од корисничког коришћења базе. Ово се обично ради кроз другу нит или процес, мањег приоритета. У систему *Hyperscale* дневник трансакција се уписује директно у област писања од стране примарног рачунског чвора, а за трајност страница је заслужан сервер страница који то ради у асинхроном режиму рада што концептуално подсећа на традиционални РСУБП.

2.3 Предности и мане

Главна предност система *Hyperscale* у односу на традиционалне РСУБП-е је већа флексибилност, односно подршка за независно скалирање рачунске и складишне моћи. У оба случаја користећи могућности више виртуелних машина. Друге предности су прожете кроз остатак рада и укључују предности као што су опоравак који временски не зависи од величине базе података и постојање именованих реплика које мање утичу на корисничко коришћење базе података.

Главна мана система *Hyperscale* у односу на традиционалне РСУБП је сложеност. Цену дефинитивно треба поменути, али то је релативан критеријум, зато што зависи у односу на који систем се пореди. Компанија *Microsoft*, поред система *Hyperscale*, има још две главне понуде релационих база података, и то су:

1. *General Purpose* [20] – *SQL Server* са једном репликом где се подаци чувају у брзом складишту у облаку. У поређењу са системом *Hyperscale*, ово је једноставнији и јефтинији систем, али мање флексибилан и углавном мањих перформанси.
2. *Business Critical* [21] – *SQL Server* са четири реплике у кластеру високе доступности. У поређењу са системом *Hyperscale*, ово је једноставнији систем и систем већих перформанси, али је скупљи (због четири копије података и четири пута више процесора у поређењу са једном репликом, јер свака реплика користи исти број процесора) и мање флексибилан.

Такође, једна битна мана у односу на конкурентне производе других компанија јесте непостојање подршке за мулти-мастер. Друге мане су прожете кроз остатак текста и укључују мане као што су спорији одзив у случају непостојања тражене странице у кешу на разним нивоима (што може бити неретко у случајевима великог оптерећења система), као и коришћење спорије складишта за складиштење страница.

3. Систем *Simplified Hyperscale-like Database*

Систем *Simplified Hyperscale-like Database* (у наставку текста "систем *SHYLD*") је развијен у оквиру овог мастер рада ради демонстрације принципа функционисања система *Hyperscale* у веома поједностављеном облику са једним рачунским чвором и једним сервером страница. Циљ система *SHYLD* је да прикаже да је могуће релативно једноставно имплементирати минимални скуп функционалности СУБП-а са архитектуром налик на *Hyperscale*. Фокус је стављен на одржавање физичке конзистентности. Архитектура система *SHYLD* је обликована по узору на систем *Hyperscale*, са неким разликама, углавном упрошћавањима о којима ће касније бити речи.

3.1 Подржани скуп функционалности

3.1.1 Рад са табелама

Систем *SHYLD* подржава прављење табеле (али не и њено брисање) која садржи само једну неименовану колону нумеричког типа (енгл. *int*). Иако ово није табела у правом релационом смислу, већ скуп бројева, у даљем тексту ће се називати табелом јер је идеја да се симулира прави СУБП, а и подршка за комплексније шеме се може накнадно додати. Прављење табеле се ради командом:

```
CREATE TABLE <table_name>
```

где `<table_name>` представља име табеле коју треба направити и мора садржати само мала слова (енгл. *lowercase*). Ова команда ће избацити изузетак при извршавању у случају да табела са траженим именом већ постоји.

Након тога, могуће је унети нове вредности у табелу командом:

```
INSERT INTO <table_name> VALUES <integer> [, <integer>]
```

при чему `<integer>` представља нумеричку вредност. Као што синтакса показује, могуће је навести листу нумеричких вредности које би се уписале у табелу. Ова команда ће избацити изузетак при извршавању у случају да вредност у табели већ постоји, односно нису дозвољени дупликати унутар једне табеле. Такође, изузетак ће се избацити и у случају да табела на коју се команда односи не постоји, што важи и за све наредне команде које оперишу са табелама.

Такође, могуће је и обрисати постојеће вредности из табеле следећом командом:

```
DELETE FROM <table_name> VALUES <integer> [, <integer>]
```

Ова команда ће избацити изузетак при извршавању у случају да вредност која је спецификована за брисање не постоји.

Систем *SHYLD* има могућност читања садржаја табеле следећом командом:

```
SELECT FROM <table_name>
```

Такође, могуће је и проверити садржај табеле следећом командом:

```
CHECK <table_name> VALUES
    <integer> [, <integer>]
  | <empty>
```

где `<empty>` представља празну листу нумеричких вредности, односно празну табелу. Ова команда ће избацити изузетак при извршавању у случају да очекивани садржај не одговара стварном (било да неки елементи нису наведени или су вишак). Ова команда је направљена да би се при тестирању лако проверио садржај табеле, односно да би се избегло парсирање резултата команде за читање.

3.1.2 Логичке операције

Један од главних механизама за одржавање атомичности и конзистентности СУБП-а јесте механизам трансакција. У систему *SHYLD* је подржан само један корисник у једном тренутку, тако да може постојати тачно једна активна трансакција у датом моменту. Следећим командама корисник може започети, потврдити или поништити трансакцију:

```
BEGIN TRANSACTION
COMMIT TRANSACTION
ROLLBACK TRANSACTION
```

Једна од главних операција СУБП-а јесте опоравак након наглог престанка рада. У систему *SHYLD*, ово се постиже процедуром која се састоји из неколико фаза:

1. **фаза опоравка** – опоравак критичних меморијских структура као што је управљач трансакција.
2. **фаза примене** (енгл. *redo phase*) – примена дневника трансакција до краја.
3. **фаза поништавања** (енгл. *undo phase*) – поништава трансакцију која је била активна али није довршена. Записи у дневнику трансакција који одговарају недовршеној трансакцији се морају поништити, пошто се током нормалног рада СУБП-а све промене одмах уписују у дневник трансакција (чак и пре успешног завршетка трансакције). Фаза поништавања се ради тако што се поништавају промене везане за недовршену трансакцију обрнутим редоследом од редоследа којим су промене настајале до тренутка док се не дође до записа који одређује почетак трансакције. Поништавање промена се врши применом супротне операције. За додавање вредности у табелу супротна операција је брисање вредности, и обратно супротна операција за брисање вредности је додавање вредности. Поништавање промена се одмах уписује у трајну меморију дневника трансакција као засебни тип записа који у себи садржи информације о оригиналној промени, који се зове запис поништавања. Запис поништавања се може само применити (у случају наглог престанка рада система током незавршеног опоравка), али се не може поништити.

Најједноставнији начин за имплементацију опоравка стања јесте да се крене од настанка базе података, прескочи прва фаза, а одради само фаза примене и фаза поништавања. Овакав приступ би могао да ради на систему малих размера на коме се овде и оперише (што и јесте била иницијална имплементација). Тренутна имплементација опоравка стања је боља и то је да се упише меморијско стање кључних структура СУБП-а у запис дневника трансакција који се зове контролна тачка (енгл.

checkpoint)[8]. Затим се фаза примене ради од те тачке надаље, док се фаза поништавања ради по истом принципу као и раније. Пошто систем *SHYLD* подржава само једну трансакцију у једном тренутку, у контролној тачки постоји само једна бинарна вредност која означава да ли је трансакција била активна у тренутку узимања контролне тачке.

Као део узимања контролне тачке мора се осигурати да се прљаве странице (енгл. *dirty page*) из меморије упишу у трајну меморију. Прљава страница је страница која је промењена само у меморији, али та промена није пропагирана на диск. Ово одлагање уписа страница је главни трик којим СУБП-и одржавају перформансе, односно уписују успешно завршене трансакције само у трајну меморију дневника трансакција, а промене на страницама одлажу за касније и те уписе раде под мањим приоритетом. У систему *SHYLD* не постоји оваква оптимизација и све промене се уписују директно у дневник трансакција чим се направе, па чак и промене недовршене трансакције. Када се прљава страница упише у трајну меморију, та страница се у меморији означи да више није прљава. У СУБП-у *SQL Server*, као део узимања контролне тачке, могу се уписати у трајну меморију и странице непотврђених трансакција. Током опоравка стања се ове странице враћају на конзистентну верзију поништавањем промена одговарајућих недовршених трансакција [22]. Систем *SHYLD* функционише по истом принципу.

Комбинацијом механизама из претходна два пасуса се добија сигурност да се фаза примене може радити од контролне тачке па надаље. Уписивањем прљавих страница у трајну меморију се добија сигурност да су странице на верзији која је била у тренутку узимања контролне тачке. Чувањем вредности критичних меморијских структура СУБП-а се добија сигурност да је СУБП у истом меморијском стању као и када би се на празну базу података применио сав дневник трансакција до тренутка узимања контролне тачке.

Узимање контролне тачке је могуће следећом командом:

```
CHECKPOINT
```

3.1.3 Конфигурационе команде

У систему *SHYLD* је омогућено бирање између режима рада традиционалног СУБП-а и система *Hyperscale*, где је *Hyperscale* предефинисана опција. Традиционални СУБП у контексту система *SHYLD* је имплементација СУБП-а који има исти скуп функционалности подржан, али се све извршава унутар једног процеса. Односно, не постоји сервис дневника трансакција, нити сервер страница. Тачније, обрађивање дневника трансакција и обрађивање страница се ради у истом процесу као и процесирање упита. Опција за традиционални СУБП је развијена ради провере рада основних функционалности СУБП-а, као што је опоравак, што је касније надограђено у систем *Hyperscale*. Опција да се врати на традиционални СУБП је остављена у случају потребе да се одради претходно поменуте провере, на пример у случају грешке у имплементацији. Могуће команде су:

```
CONFIGURE DATABASE TRADITIONAL
CONFIGURE DATABASE HYPERSCALE
```

[8] Треба нагласити да се под контролном тачком подразумева имплементација налик СУБП-у *SQL Server* где се контролна тачка односи на целу базу података и не узрокује потврду трансакције.

Командом за прекидање рада се може симулирати нагли престанак рада одређеног процеса. У режиму рада система *SHYLD* са традиционалним СУБП-ом, није могуће убити сервер страница и сервис дневника трансакција, јер ни не постоје у том контексту. Уколико се не спецификује другачије, подразумевана је опција `ALL`, која прекида рад свих процеса система. Прекидање рада се може постићи следећом командом:

```
KILL [DATABASE | STORAGE | LOG | ALL]
```

Могуће је конфигурисати систем *SHYLD* да не уписује дневник трансакција у трајну меморију. Ово је коришћено у сврхе тестирања исправности имплементације, па је и остављено да се може поново искористити у исту сврху уколико се укаже потреба. Команде су:

```
CONFIGURE LOGGING OFF
CONFIGURE LOGGING ON
```

3.1.4 Тестови

Тестови се могу покренути један по један или сви одједном где је `<test_name>` име теста који треба покренути. Примери тестова су приказани у поглављу 3.3.4. Команде су:

```
RUN <test_name>
RUN ALL
```

3.2 Недостаци

У систему *SHYLD* не постоји концепт колоне, индекса, окидача, као ни многе друге функционалности које би биле очекиване од продукционог СУБП-а. Страница података није исте величине као страница диска што има негативног утицаја на перформансе система, мада на количини података на којој систем *SHYLD* оперише овај утицај је занемарљив.

У систему *SHYLD* подржан је само један корисник у датом тренутку, тако да подршка за катанце и комплексније механизме за одржавање атомичности и конзистентности у конкурентном систему није подржана.

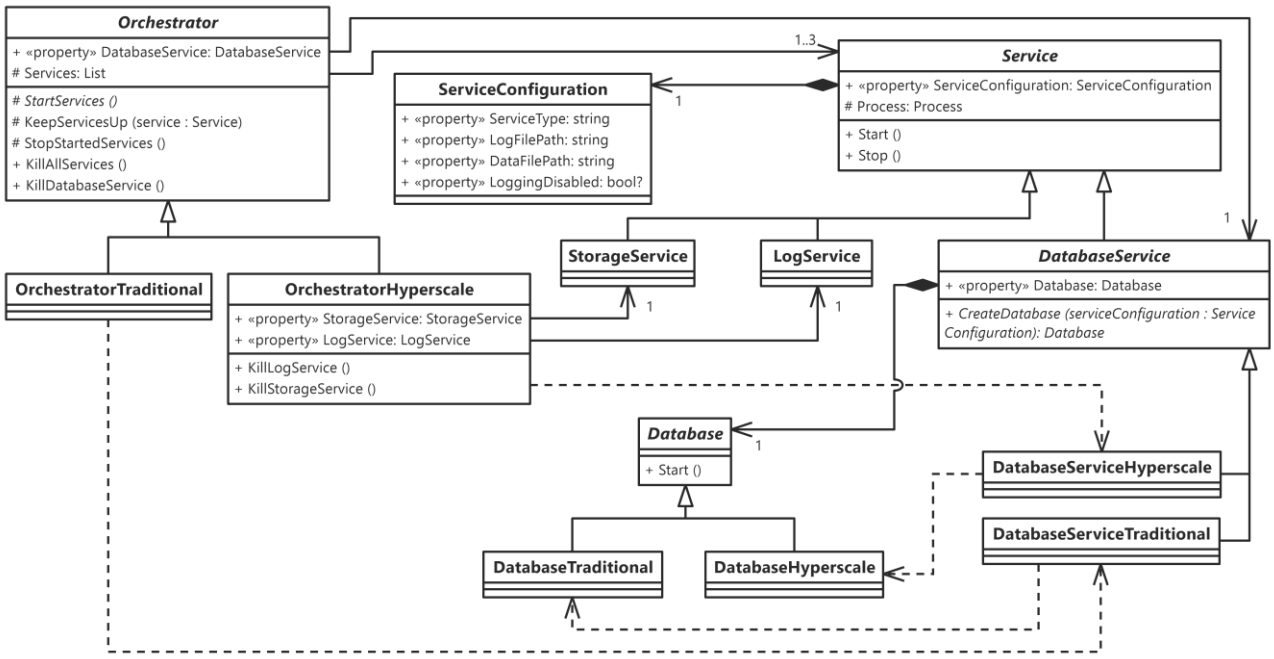
Као што смо већ установили, код система *Hyperscale*, примарни рачунски чвор има право уписа на област писања, што је датотека у трајној меморији која чува дневник трансакција. У систему *SHYLD* је то поједностављено тиме што сервис дневника трансакција има искључив приступ датотеци дневника трансакција на трајној меморији. Односно, уместо да примарни рачунски чвор уписује директно у област писања он шаље поруку сервису дневника трансакција да упише дневник трансакција у трајну меморију, што сервис дневника трансакција и уради. На овај начин је имплементација поједностављена тиме што не мора да се брине о проблемима конкурентног приступа и чистија је јер се не нарушава организација класа у имплементацији. У овим условима перформансе продукционог система би засигурно биле лошије, али је овакав приступ концептуално чистији, што је оправдано у случају система *SHYLD*.

3.3 Имплементација

Имплементација система *SHYLD* се може наћи на следећем веб линку: https://github.com/jeremicmilan/Database.

3.3.1 Оркестрација

Да би систем *SHYLD* могао да ради, мора постојати начин покретања процеса који представљају чворове система. Дијаграм 2 илуструје како је оркестрација постигнута у случају система *SHYLD*.



Дијаграм 2. Оркестрација покретања чворова система.
https://github.com/jeremicmilan/Database/tree/master/Diagrams/Orchestration

Постоји оркестратор (*Orchestrator*) који је задужен за покретање (*StartServices*), заустављање (*KillAllServices*) и поновно покретање осталих чворова система (*KeepServiceUp*). Приликом свог покретања, оркестратор покрене и остале чворове система. У коду се чворови система именују сервисима и то:

- рачунски чвор је *DatabaseService*
- сервер страница је *StorageService*
- сервис дневника трансакција је *LogService*.

Сва три наслеђују класу *Service* која представља сервис, односно чвор система. Сервис подржава покретање (*Start*) и стопирање (*Stop*), као и референцу ка класи која представља сам процес (*Process*).

Такође, сервис има референцу на конфигурацију сервиса (*ServiceConfiguration*), у коме може наћи неке релевантне информације о себи. Конфигурација сервиса се серијализује (претвара у *json* текст) и шаље преко цеви процеса (енгл. *process pipe*) новокреираном процесу, да би тај процес знао коју функцију у систему врши. Ово је ради једноставности имплементације, пошто је коришћена једна имплементација процеса за све чворове система. Од битнијих поља, конфигурација сервиса садржи

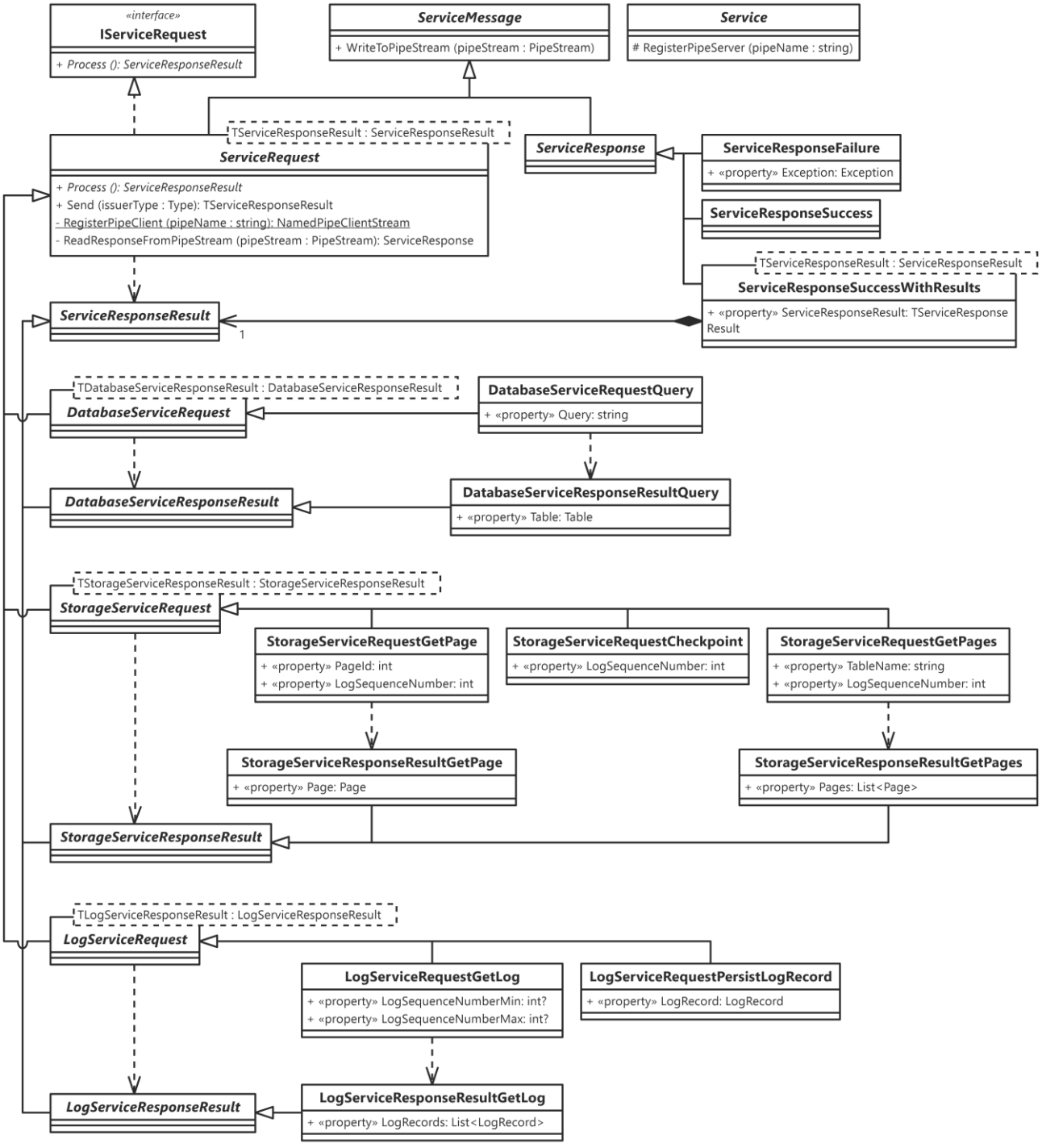
путању датотеке страница (*DataFilePath*), путању датотеке дневника трансакција (*LogFilePath*) и тип сервиса (*ServiceType*). Путања датотеке страница је иницијализована само за *StorageService* и *DatabaseServiceTraditional*. Путања датотеке дневника трансакција је иницијализована само за *LogService* и *DatabaseServiceTraditional*.

Постоје две верзије оркестратора који наслеђују главну класу *Orchestrator*:

1. **традиционални** (*OrchestratorTraditional*) - наслеђује све од базне класе која има све што му је потребно за оркестрацију и комуницирање са самим СУБП-ом чији је тип *DatabaseServiceTraditional*. Овај оркестратор има могућност прекидања рада сервиса базе података (*KillDatabaseService*).
2. ***Hyperscale*** (*OrchestratorHyperscale*) оркестратор - такође наслеђује све из базне класе, међутим њему *DatabaseService* представља рачунски чвор и типа је *DatabaseServiceHyperscale*. Поред тога, *Hyperscale* оркестратор има и додатне информације специфичне за њега и то су референце ка серверу страница (*StorageService*) и дневнику трансакција (*LogService*). Овај оркестратор има могућност прекидања рада свих осталих чворова система (*KillDatabaseService, KillStorageService, KillLogService*).

Једно од задужења класе *DatabaseService* јесте да направи (*CreateDatabase*) инстанцу класе СУБП-а (*Database*). За традиционални СУБП одговарајућа класа је *DatabaseTraditional*, док је за *Hyperscale* одговарајућа класа *DatabaseHyperscale*. Задужење различитих класа СУБП-а је да инстанцирају одговарајуће објекте за свој тип СУБП-а.

Дијаграм 3 показује како сви ови сервиси комуницирају између себе.



Дијаграм 3. Комуникација између сервиса.
https://github.com/jeremicmilan/Database/tree/master/Diagrams/Communication

Главна класа која представља поруку између сервиса је **порука сервиса** (*ServiceMessage)* која има могућност да њен садржај буде уписан преко цеви процеса (*WriteToPipeStream*). Постоје два различита типа порука које се размењују између сервиса:

1. **захтев сервиса** (*ServiceRequest*)
2. **одговор сервиса** (*ServiceResponse*).

Обе класе су изведене из поруке сервиса. Сам сервис је задужен да слуша на цеви процеса као сервер (*RegisterPipeServer*), док је захтев сервиса задужен да уписује преко цеви процеса, пре чега се мора регистровати као клијент (*RegisterPipeClient*) као иницијатор комуникације. Захтев сервиса такође чека на одговор и чита га преко цеви процеса (*ReadResponseFromPipeStream*).

Одговор сервиса може бити:

1. **успешан са резултатом** (*ServiceResponseSuccessWithResults*) - генеричка класа (енгл. *generic class*) која очекује одређени тип резултата одговора сервиса (*ServiceResponseResult*). Кроз механизам генеричке класе се намеће коришћење одређеног типа приликом враћања одговора, што олакшава имплементацију нових порука тиме што се провере типова раде за време компајлирања. Он има поље *ServiceResponseResult* који представља сам резултат одговора.
2. **успешан одговор без резултата** (*ServiceResponseSuccess*)
3. **неуспешан одговор** (*ServiceResponseFailure*) - има поље типа изузетак (*Exception*) са свим информацијама о изузетку који се догодио приликом обраде текућег захтева.

Захтев сервиса је генеричка класа која захтева да се наведе тип резултата одговора сервиса за који ће тај захтев сервиса бити везан. Ово значи да ако дефинишемо захтев сервиса на прави начин, тај захтев ће бити у обавези да врати одговарајући одговор сервиса. Ово олакшава имплементацију сервиса иницијатора захтева и сервиса који обрађује захтев, зато што се провере типова раде током компајлирања. Слање (*Send*) захтева се извршава на страни иницијатора захтева, док се процесирање (*Process*) захтева врши на страни сервиса који ради обраду захтева.

Хијерархије захтева сервиса и резултата одговора сервиса су синхронизоване. Односно, за сваки захтев сервиса постоји одговарајући резултат одговора сервиса, осим у случајевима када резултата одговора нема (успешан одговор без резултата).

Захтеви ка рачунском чвору морају наслеђивати класу *DatabaseServiceRequest*, а одговарајући резултат мора наслеђивати класу *DatabaseServiceResponseResult*. Постоји само један захтев ка рачунском чвору који иницира клијент базе података и то је *DatabaseServiceRequestQuery*, који представља захтев за извођење упита над базом података и прима параметар који се зове упит (*Query*) који је типа ниске карактера. Одговор на овај захтев је *DatabaseServiceResponseResultQuery*, који у себи садржи табелу (*Table*).

Захтеви ка серверу страница морају наслеђивати класу *StorageServiceRequest*, а одговарајући резултат мора наслеђивати класу *StorageServiceResponseResult*. Постоје три типа захтева:

1. захтев од рачунског чвора да се испоручи страница (*StorageServiceRequestGetPage*). Параметри за овај захтев су редни број странице (*PageId*) и редни број дневника трансакција (*LogSequenceNumber*) са којим се страница мора испоручити (може бити и новија верзија). Одговор на овај захтев је *StorageServiceResponseResultGetPage*, који у себи садржи саму страницу (*Page*).
2. захтев од рачунског чвора да се испоруче све странице за једну табелу (*StorageServiceRequestGetPages*). Параметри за овај захтев су име табеле (*TableName*) и редни број дневника трансакција (*LogSequenceNumber*) са којим се странице морају испоручити (може бити и новија верзија). Одговор на овај захтев је *StorageServiceResponseResultGetPages*, који у себи садржи листу страница (*Pages*).
3. захтев за узимање контролне тачке (*StorageServiceRequestCheckpoint*), који нема одговарајући резултат одговора, већ у случају успешног узимања контролне тачке враћа успешан одговор без резултата.

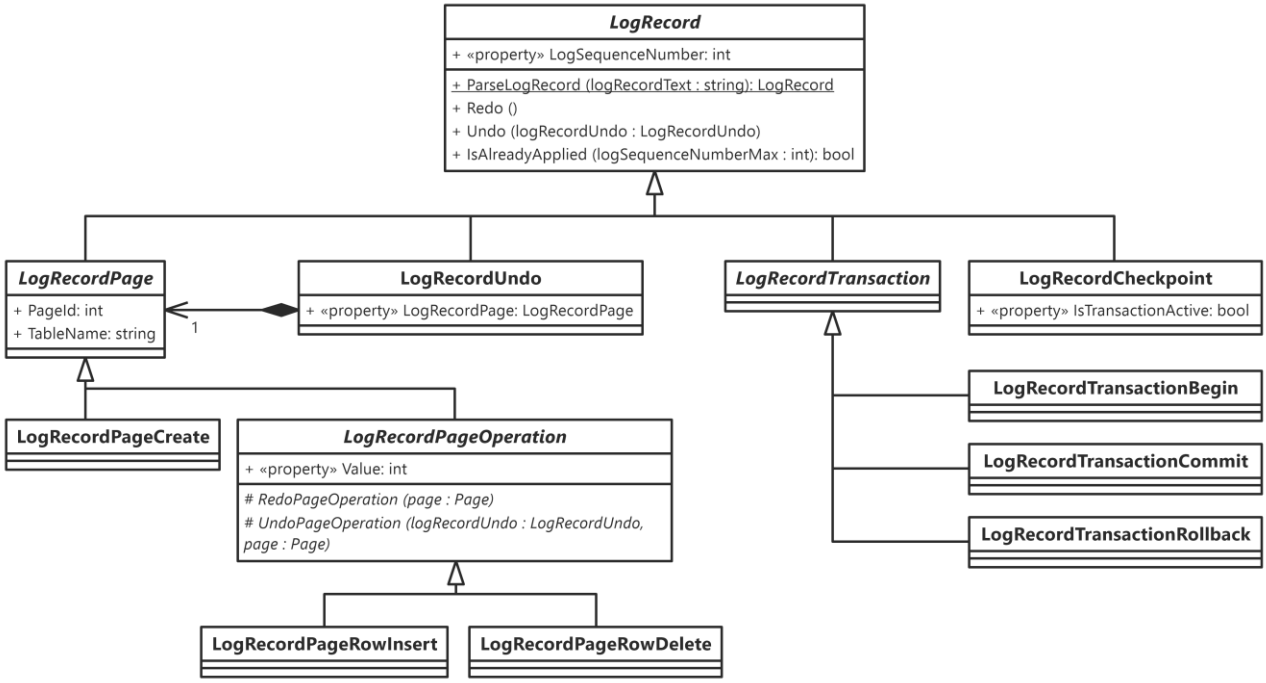
Захтеви ка сервису дневника трансакција морају наслеђивати класу *LogServiceRequest*, а одговарајући резултат мора наслеђивати класу *LogServiceResponseResult*. Постоје два типа захтева:

1. захтев од рачунског чвора или сервера страница да се испоручи дневник трансакција (*LogServiceRequestGetLog*).
2. захтев за уписивање записа из дневника трансакција у трајну меморију (*LogServiceRequestPersistLogRecord*), који нема одговарајући резултат одговора, већ у случају успешног уписа враћа успешан одговор без резултата.

Параметри за захтев испоручивања дневника трансакција су два опциона параметара: минимални (*LogSequenceNumberMin*) и максимални (*LogSequenceNumberMax*) редни број дневника трансакција. Ако минимални редни број дневника трансакција није наведен, вратиће се дневник трансакција од почетка, а у случају да максимални није наведен, вратиће се дневник трансакција до краја. Ако ни један није наведен, вратиће се цео дневник трансакција. Пожељно је да ови параметри буду наведени због оптимизације перформанси, али у систему *SHYLD*, те оптимизације нису увек рађене. Одговор на овај захтев је *LogServiceResponseResultGetLog*, који у себи садржи листу записа из дневника трансакција (*LogRecords*).

3.3.2 Систем за управљање базама података

Дијаграм 4 приказује хијерархију записа из дневника трансакција (*LogRecord*).



Дијаграм 4. Записи из дневника трансакција.
https://github.com/jeremicmilan/Database/tree/master/Diagrams/LogRecord

Сви записи подржавају операције примене (*Redo*) и поништавања (*Undo*). Такође, сваки запис мора имати свој уникатни редни број дневника трансакција (*LogSequenceNumber*)[9].

Постоје 4 изведене класе из главне базне класе записа из дневника трансакција и то су:

1. **запис странице** (*LogRecordPage*)
2. **запис поништавања** (*LogRecordUndo*)
3. **запис трансакције** (*LogRecordTransaction*)
4. **запис контролне тачке** (*LogRecordCheckpoint*).

Прва два, запис странице и запис поништавања, су физички записи, док су друга два, запис трансакције и запис контролне тачке, логички записи.

Записи страница садрже редни број странице (*PageId*) и име табеле (*TableName*). Разликују се следећи типови записа страница:

1. запис за прављење странице (*LogRecordPageCreate*)
2. запис операције над вредностима странице (*LogRecordPageOperation*) са два типа операција које чувају стање у виду поља *Value* које чува саму вредност:
   a. додавање вредности у страницу (*LogRecordPageRowInsert*)
   b. брисање вредности из странице (*LogRecordPageRowDelete*)

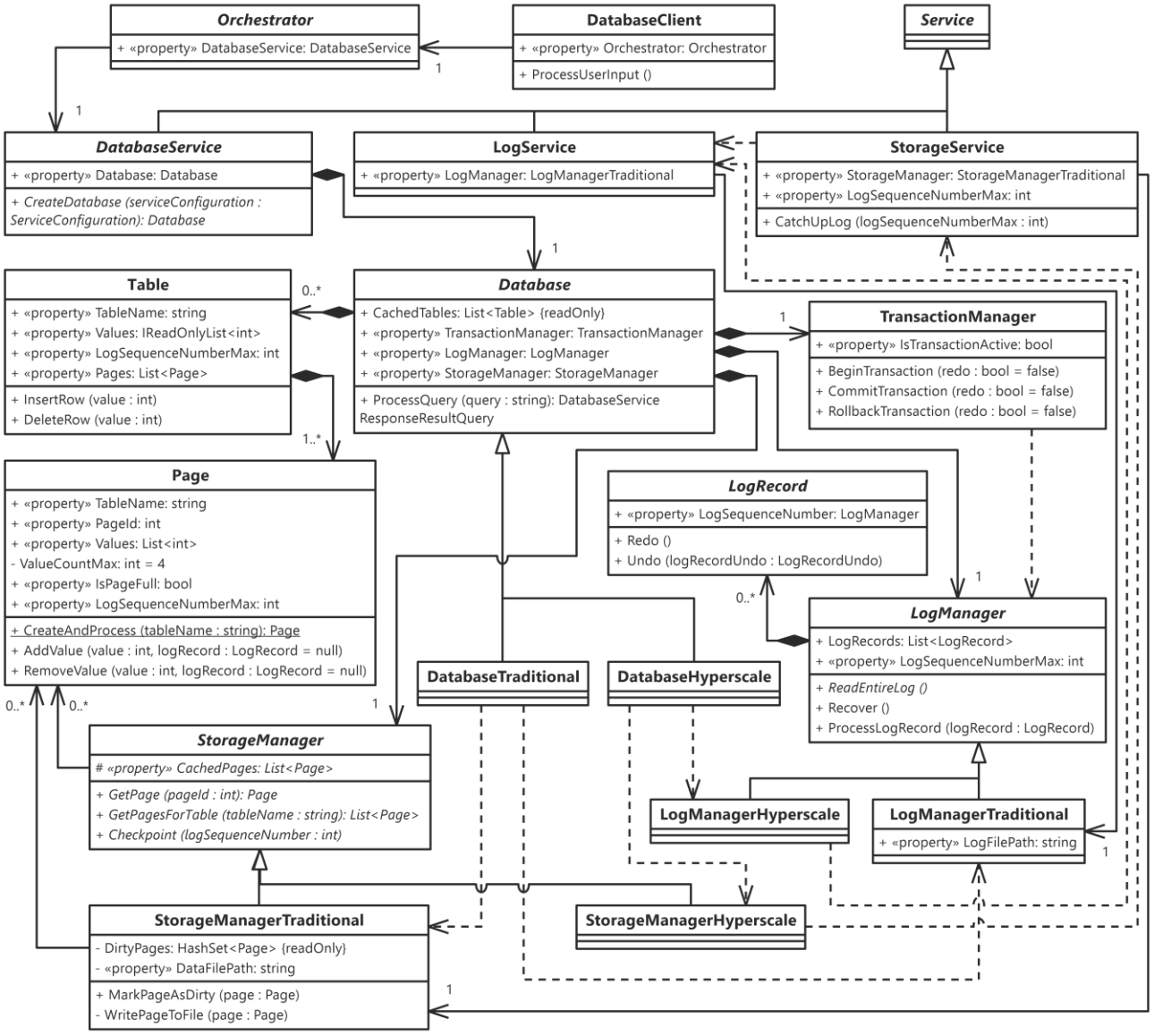
[9] Да би систем *SHYLD* подржао више трансакција у паралели базна класа записа из дневника трансакција би садржала и идентификатор трансакције који би био коришћен у фази поништавања.

Запису поништавања, пошто је физички запис, је потребан идентификатор странице који добија индиректно јер има референцу на сам запис странице (*LogRecordPage*) који поништава. Запис поништавања је запис који се користи приликом фазе поништавања током опоравка стања објашњене у 3.1.2.

Постоје три типа **записа трансакција** и то су започињање трансакције (*LogRecordTransactionBegin*), успешно завршавање (*LogRecordTransactionCommit*) и поништавање (*LogRecordTransactionRollback*).

Запис контролне тачке садржи поље *IsTransactionActive* која указује да ли је трансакција била активна у тренутку узимања контролне тачке.

Дијаграм 5 показује архитектуру имплементације система *SHYLD*.



Дијаграм 5. Архитектура СУБП-а у систему SHYLD.
https://github.com/jeremicmilan/Database/tree/master/Diagrams/Database

Главна функција коју клијентска компонента СУБП-а (*DatabaseClient*) ради јесте процесирање улазног текста од стране корисника (*ProcessUserInput*). Линија текста може бити идентификована као:

1. контролна команда - она се преусмерава оркестратору да је спроведе.
2. команда за извршавање теста - клијентска компонента СУБП-а сама настави интерпретацију и извршавање теста
3. све друго се претпоставља да је у питању упит и прослеђује се бази података (*Database*) преко захтева *DatabaseServiceRequestQuery* на процесирање (*ProcessQuery*). Како процесирање и извршавање упита изгледа у пракси биће приказано у поглављу 3.4.

Основне три компоненте базе података у функционалностима које систем *SHYLD* подржава јесу:

1. **управљач складишта** (*StorageManager*)
2. **управљач дневника трансакција** (*LogManager*)
3. **управљач трансакција** (*TransactionManager*).

Управљач складишта има бафер страница у меморији, који је на дијаграму приказан као листа страница (*CachedPages*). Подржане методе су дохватање страница (*GetPage* и *GetPagesForTable*) и узимање контролне тачке (*Checkpoint*)[10]. Постоје два типа (изведене класе) управљача складишта:

1. **традиционални** (*StorageManagerTraditional*) - има додатне дужности, као што је вођење рачуна око прљавих страница (*MarkPageAsDirty* и *DirtyPages*). Такође, он има и поље које представља физичку путању до датотеке страница (*DataFilePath*) из које се ради дохватање страница из трајне меморије имплементацијом функције *GetPagesForTable* из наткласе, као и упис страница (*WritePageToFile*). Узимање контролне тачке се ради по принципу објашњеном у поглављу 3.1.2.
2. ***Hyperscale*** (*StorageManagerHyperscale*) – његова инстанца се прави само на рачунском чвору. Он све потребне радње прослеђује серверу страница. Прослеђене операције су дохватање страница из трајне меморије преко захтева *StorageServiceRequestGetPage* и *StorageServiceRequestGetPages*, као и узимање контролне тачке имплементацијом функције *Checkpoint* преко захтева *StorageServiceRequestCheckpoint*.

Да би обрадио захтеве, сервер страница има референцу ка управљачу складишта (*StorageManager*) који је традиционалног типа. Главна идеја је искоришћавање већ постојеће имплементације приступа трајној меморији за странице[11]. Сервер страница има поље специфично за њега, а то је тренутни редни број дневника трансакција на коме оперише (*LogSequenceNumberMax*). Метода којом сервер страница дохвата потребан дневник трансакција од сервиса дневника трансакција је *CatchUpLog* користећи захтев *LogServiceRequestGetLog*.

Страница (*Page*) има јединствени идентификатор и то је редни број странице (*PageId*). Страница може припадати само једној табели и зато чува име табеле (*TableName*) које служи као идентификатор приликом читања свих страница за једну табелу из трајне меморије (*GetPagesForTable*). Страница такође има листу вредности

[10] Класа *Database* која је позивалац узимања контролне тачке најпре забележи операцију позивајући управљач дневника трансакција. Запис који се користи у овом случају је *LogRecordCheckpoint*.

[11] У случају да систем *SHYLD* има више сервера страница, имплементација би била захтевала да сваки сервер страница обрађује странице додељене њему, што би била компликованија имплементација. Али чак и у том случају оваква архитектура би била могућа и поновна искористивост кода максимална.

(*Values*) које припадају тој страници. Такође, има и редни број трансакције која је последња примењена на страницу (*LogSequenceNumberMax*). Ово је потребно да би чворови система знали да раде са очекиваном верзијом странице. На пример, рачунски чвор мора знати коју верзију му сервер страница испоручује да би могао да направи исправну одлуку. Максималан број вредности у страници је четири[12] (*ValueCountMax*), а бинарна вредност *IsPageFull* је тачна уколико је овај лимит достигнут. Вредности се додају функцијом *AddValue*[13], а бришу функцијом *RemoveValue*[14]. Такође, постоји статичка функција *CreateAndProcess* која је главна функција за прављење нових страница приликом прављења или ширења табела[15].

Табела пружа могућност додавања (*InsertRow*) и брисања (*DeleteRow*) вредности из ње. Ове функције ће бити преусмерене на одговарајућу страницу, осим у случају када су све странице пуне приликом додавања вредности када се прави нова страница и операција се прослеђује њој. Табела садржи листу вредности (*Values*) која може само да се чита и онда представља унију вредности свих страница везаних за табелу. Табела има информацију на ком редном броју дневника трансакција (*LogSequenceNumberMax*) се десила последња промена на њој. Последње поље табеле јесте име табеле (*TableName*).

Управљач дневника трансакција има кеш дневника трансакција у меморији, који је на дијаграму приказан као листа записа из дневника трансакција (*LogRecords*). Неке од подржаних метода су читање дневника трансакција (*ReadEntireLog*) и опоравак стања (*Recover*). Опоравак стања се ради по принципу објашњеном у поглављу 3.1.2. Постоје два типа (изведене класе) управљача дневника трансакција:

1. традиционални (*LogManagerTraditional*) - има поље која представља физичку путању до датотеке дневника трансакција (*LogFilePath*) из кога се ради дохватање дневника трансакција из трајне меморије (имплементација виртуелне функције *ReadEntireLog*), као и процесирање записа из дневника трансакција (*ProcessLogRecord*). Процесирање подразумева уписивање дневника трансакција у трајну меморију, као и у меморијски кеш (*LogRecords*)[16].
2. *Hyperscale* (*LogManagerHyperscale*). – његова инстанца се прави само на рачунском чвору. Он све потребне радње прослеђује сервису дневника трансакција. Прослеђене операције су дохватање дневника трансакција из трајне меморије преко захтева *LogServiceRequestGetLog*, као и његово процесирање преко захтева *LogServiceRequestPersistLogRecord*.

Да би обрадио захтеве, сервис дневника трансакција има референцу ка управљачу дневника трансакција (*LogManager*) који је традиционалног типа. Слично као и са сервером страница и традиционалним типом управљача складишта, идеја је поновна искористивост кода.

---

[12] У продукционим системима овај лимит би био значајно већи, али је у систему *SHYLD* значајно смањен да би се са једноставним тестовима могли тестирати гранични случајеви који укључују више страница. Такође, у продукционим системима као што је *SQL Server,* лимит није фиксан у броју вредности које стају у страницу, него је величина странице фиксна, а онда величина једне вредности (или реда) одређује колико их може стати у једну страницу.

[13] Бележи се у дневнику трансакција записом *LogRecordPageRowInsert*.

[14] Бележи се у дневнику трансакција записом *LogRecordPageRowDelete*.

[15] Бележи се у дневнику трансакција записом *LogRecordPageCreate*.

[16] Процесирање се позива из разних компоненти СУБП-а. У овом раду је углавном обележено фуснотама са објашњењем да се одређена операција бележи у дневнику трансакција са одређеним записом.

Управљач трансакција је главни арбитар за трансакције. Подржане операције над трансакцијама су почетак (*BeginTransaction*)[17], успешни завршетак (*CommitTransaction*)[18] и прекидање (*RollbackTransaction*)[19]. Прекидање трансакције ради по сличном принципу као фаза поништавања приликом опоравка стања која је објашњена у поглављу 3.1.2. Управљач трансакција има бинарну вредност *IsTransactionActive* која представља да ли је трансакција у току. С обзиром да систем *SHYLD* подржава само једну трансакцију у датом тренутку, једна бинарна вредност је довољна[20].

3.3.3 Радни директоријум

Систем *SHYLD*, као и систем *Hyperscale*, мора имати датотеке у трајној меморији и у случају система *SHYLD* је, ради једноставности имплементације, то увек диск. Датотеке страница података *database.data* и дневника трансакција *database.datalog* се чувају у привременом директоријуму *WorkingDirectory* који представља радни директоријум.

Такође, у радном директоријуму се налазе и датотеке које садрже трагове извршавања процеса. Трагови извршавања који завршавају у отвореним конзолама током извршавања система *SHYLD* завршавају у претходно поменутим датотекама и служе да се установе проблеми са радом тих процеса. Те датотеке служе за опоравак трагова извршавања конзола након наглог престанка рада конзола. У том случају се додаје хоризонтална линија која означава ново покретање. Ово је ради лакшег увида у то шта се дешавало пре наглог престанка рада процеса. Назив ових датотека је име конзоле, а екстензија је *trace*.

3.3.4 Тестови

Тестови су додавани како је имплементирана подршка за нове функционалности система *SHYLD*. Често покретање ових тестова је осигуравало да се не направи грешка у претходно подржаним функционалностима система *SHYLD*, док сада служе као репрезентативни примери који показују подржану функционалност.

Приликом извршавања теста, прави се привремени радни директоријум у самом тест директоријуму, да би тестови задржали изолацију један од другог, али и од главног радног директоријума. Односно, ако постоји проблем са једним тестом, то не би требало да утиче на остале тестове.

Да би се додао нови тест потребно је додати нови директоријум теста у директоријуму *Tests* и унутар тог директоријума је потребно додати датотеку *test.txt* која садржи листу команди које представљају сам тест. У директоријуму теста се могу наћи и две датотеке: *expected.datalog* датотека која представља очекивани изглед датотеке дневника трансакција и *expected.data* датотека која представља очекивани изглед датотеке страница података након извршавања теста. Након завршетка извршавања теста ради се провера претходно поменутих датотека са завршним датотекама у привременом радном директоријуму теста.

---

[17] Бележи се у дневнику трансакција записом *LogRecordTransactionBegin*.
[18] Бележи се у дневнику трансакција записом *LogRecordTransactionCommit*.
[19] Бележи се у дневнику трансакција записом *LogRecordTransactionRollback*.
[20] У озбиљнијим системима би се овде нашла барем листа идентификатора трансакција које су у току.

У наставку следи неколико репрезентативних тестова. Нису сви тестови који су направљени за систем *SHYLD* излистани овде, пошто би заузели превише простора, а не доносе превише вредности. Тестови који нису приказани у раду се могу наћи на следећој локацији: *https://github.com/jeremicmilan/Database/tree/master/Tests*.

Пример 1 је тест са више табела, повременим додавањем вредности у табеле, као и брисање вредности из табела, са повременом контролном тачком, као и неколико симулација наглог заустављања процеса.

```
CREATE TABLE first
INSERT INTO first VALUES 1, 2, 3
DELETE FROM first VALUES 2
CHECK first VALUES 1, 3

CREATE TABLE second
INSERT INTO second VALUES 4, 5
DELETE FROM second VALUES 4
CHECK second VALUES 5

CHECKPOINT

KILL

CHECK first VALUES 1, 3
DELETE FROM first VALUES 1
CHECK first VALUES 3

CHECK second VALUES 5
INSERT INTO second VALUES 6
CHECK second VALUES 5, 6

KILL

CHECK first VALUES 3

CHECK second VALUES 5, 6
```

Пример 1. https://github.com/jeremicmilan/Database/tree/master/Tests/04.CheckpointKillMultipleTables

Пример 2 је пример теста са трансакцијом која је започета, где је узета контролна тачка, а затим прекинута наглим престанком рада процеса. Очекивање је да се оваква трансакција поништи, што тест и проверава.

```
CREATE TABLE first
INSERT INTO first VALUES 1, 2, 3

BEGIN TRANSACTION

INSERT INTO first VALUES 4

CHECKPOINT

KILL

CHECK first VALUES 1, 2, 3

KILL

CHECK first VALUES 1, 2, 3
```

Пример 2.
https://github.com/jeremicmilan/Database/blob/master/Tests/06.TransactionBeginCheckpointKillKill/test.txt

Пример 3 је тест који покреће и завршава једну трансакцију, затим покреће другу трансакцију (за коју очекује да ће се поништити) и, пре него што је заврши, симулира нагло заустављање СУБП-а, са уписима и проверама вредности за време и ван трансакција, као и повременим узимањем контролне тачке.

```
CREATE TABLE first
INSERT INTO first VALUES 1, 2, 3

CHECKPOINT

BEGIN TRANSACTION
INSERT INTO first VALUES 4
COMMIT TRANSACTION

INSERT INTO first VALUES 5

BEGIN TRANSACTION
INSERT INTO first VALUES 6

KILL

CHECK first VALUES 1, 2, 3, 4, 5

CHECKPOINT

CHECK first VALUES 1, 2, 3, 4, 5
```

Пример 3.
https://github.com/jeremicmilan/Database/blob/master/Tests/09.CheckpointTransactionCommitTransactionBeginKillCheckpoint/test.txt

3.4 Пример извршавања система *SHYLD*

Прво ће се направити табела звана *test*. Слика 1 показује да када клијент тражи нову табелу од система, тај захтев прво долази до рачунског чвора. Пошто рачунски чвор није сигуран да ли та табела постоји, прво мора консултовати сервер страница, на шта и добија одговор да табела заиста не постоји индиректно кроз чињеницу да не постоје странице за ту табелу. У продукционом систему би се одиграла слична ситуација, јер би рачунски чвор морао да дохвати страницу (у случају да је нема у баферу страница) на којој се налазе мета подаци који чувају информацију које табеле постоје у бази података. Пошто је рачунски чвор сада сигуран да табела не постоји, прави ту табелу у меморији. Табела мора имати барем једну страницу (па макар и празну), па се страница такође прави у меморији. Одмах затим се формира дневник трансакције за прављење странице и прослеђује се сервису дневника трансакција. Сервис дневника трансакција тај дневник трансакција уписује у кеш и трајну меморију[21]. Након потврдног одговора од сервиса дневника трансакција, потврдан одговор се прослеђује и клијенту.

DatabaseClient

```
2022-08-17 19:35:37.5906  ::  Starting up DatabaseServiceHyperscale...
2022-08-17 19:35:37.6216  ::  Starting up StorageService...
2022-08-17 19:35:37.6223  ::  Starting up LogService...
2022-08-17 19:35:38.0650  ::  LogService started.
2022-08-17 19:35:38.0666  ::  StorageService started.
2022-08-17 19:35:38.0687  ::  DatabaseServiceHyperscale started.
> CREATE TABLE test
2022-08-17 19:35:42.9175  ::  Initiating request DatabaseServiceRequestQuery.
2022-08-17 19:35:43.0335  ::  Request DatabaseServiceRequestQuery finished.
```

DatabaseServiceHyperscale

```
--------------------------------------------------------------------------------
2022-08-17 19:35:38.7670  ::  Starting DatabaseServiceHyperscale...
2022-08-17 19:35:38.7832  ::  Initiating request LogServiceRequestGetLog.
2022-08-17 19:35:38.8873  ::  Request LogServiceRequestGetLog finished.
2022-08-17 19:35:38.8901  ::  DatabaseServiceHyperscale started.
2022-08-17 19:35:42.9277  ::  Creating table: test
2022-08-17 19:35:42.9300  ::  Reading pages for table test from cache.
2022-08-17 19:35:42.9315  ::  Pages for table test not found in cache.
2022-08-17 19:35:42.9333  ::  Initiating request StorageServiceRequestGetPages.
2022-08-17 19:35:42.9845  ::  Request StorageServiceRequestGetPages finished.
2022-08-17 19:35:42.9868  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:43.0005  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:43.0017  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:43.0026  ::  Page with id 1 not found in cache.
2022-08-17 19:35:43.0039  ::  Created table: test
```

StorageService

```
--------------------------------------------------------------------------------
2022-08-17 19:35:38.7941  ::  Starting StorageService...
2022-08-17 19:35:38.8079  ::  StorageService started.
2022-08-17 19:35:42.9487  ::  Getting pages for table test with LSN -1.
2022-08-17 19:35:42.9509  ::  Reading pages for table test from cache.
2022-08-17 19:35:42.9524  ::  Pages for table test not found in cache.
2022-08-17 19:35:42.9540  ::  Reading pages from disk for table: test
2022-08-17 19:35:42.9553  ::  Read pages for table test from disk.
2022-08-17 19:35:42.9563  ::  No pages to return.
```

LogService

```
--------------------------------------------------------------------------------
2022-08-17 19:35:38.7554  ::  Starting LogService...
2022-08-17 19:35:38.7721  ::  Reading log from disk...
2022-08-17 19:35:38.7743  ::  No log found in file.
2022-08-17 19:35:38.7769  ::  LogService started.
2022-08-17 19:35:38.8452  ::  Getting entire log.
2022-08-17 19:35:38.8466  ::  Nothing to return.
2022-08-17 19:35:42.9960  ::  Persisting log record: 1,PageCreate,1,test
2022-08-17 19:35:42.9972  ::  Writing log record to disk: 1,PageCreate,1,test
2022-08-17 19:35:42.9980  ::  Written log record to disk: 1,PageCreate,1,test
2022-08-17 19:35:42.9989  ::  Persisted log record: 1,PageCreate,1,test
```

Слика 1. Стање конзола након прављења табеле.

[21] Формат записа дневника трансакција је следећи *1,PageCreate,1,test*. Делови су одвојени запетом следећим редоследом: редни број дневника трансакција, тип записа дневника трансакција и специфичне вредности по запису (у овом случају су редни број странице и име табеле).

Након креирања табеле уписују се две вредности у табелу (1 и 2). Слика 2 приказује да клијент шаље захтев рачунском чвору, који у свом баферу страница проналази прву празну страницу наведене табеле и закључује да је додавање ових вредности могуће. За сваку вредност понаособ, рачунски чвор уписује вредност у страницу у баферу страница и шаље запис дневника трансакција[22] сервису дневника трансакција, који то уписује у трајну меморију и враћа потврдан одговор. На крају се потврдни одговор прослеђује и клијенту.

DatabaseClient

```
> INSERT INTO test VALUES 1,2
2022-08-17 19:35:43.0349  ::  Initiating request DatabaseServiceRequestQuery.
2022-08-17 19:35:43.2468  ::  Request DatabaseServiceRequestQuery finished.
```

DatabaseServiceHyperscale

```
2022-08-17 19:35:43.1174  ::  Adding [1, 2] to table test
2022-08-17 19:35:43.1217  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:43.1292  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:43.1300  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:43.1314  ::  Read the page with id 1 from cache.
2022-08-17 19:35:43.1325  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:43.2425  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:43.2437  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:43.2445  ::  Read the page with id 1 from cache.
2022-08-17 19:35:43.2455  ::  Added [1, 2] to table test
```

StorageService

LogService

```
2022-08-17 19:35:43.1254  ::  Persisting log record: 2,PageRowInsert,1,test,1
2022-08-17 19:35:43.1265  ::  Writing log record to disk: 2,PageRowInsert,1,test,1
2022-08-17 19:35:43.1273  ::  Written log record to disk: 2,PageRowInsert,1,test,1
2022-08-17 19:35:43.1282  ::  Persisted log record: 2,PageRowInsert,1,test,1
2022-08-17 19:35:43.2387  ::  Persisting log record: 3,PageRowInsert,1,test,2
2022-08-17 19:35:43.2398  ::  Writing log record to disk: 3,PageRowInsert,1,test,2
2022-08-17 19:35:43.2407  ::  Written log record to disk: 3,PageRowInsert,1,test,2
2022-08-17 19:35:43.2416  ::  Persisted log record: 3,PageRowInsert,1,test,2
```

Слика 2. Стање конзола након уписивања две вредности у табелу.

Након овога се симулира нагло прекидање рада рачунског чвора. Слика 3 показује стање након. Пошто је клијент у систему *SHYLD* задужен и за оркестрацију осталих процеса система, он детектује да је рачунски чвор престао са радом и покреће га поново. Приликом стартовања рачунски чвор прво дохвата дневник трансакција од сервиса дневника трансакција. Приликом примене тог дневника трансакција долази до прве операције која је прављење нове странице. У овом тренутку је рачунски чвор сигуран да је ова операција валидна и да та табела не постоји, па прави табелу у меморији без консултације сервера страница. У наставку, рачунски чвор примењује остатак дневника трансакција и након примене долази до стања базе као и пре наглог престанка рада.

DatabaseClient

```
2022-08-17 19:35:44.1017  ::  DatabaseServiceHyperscale exited.
2022-08-17 19:35:44.2163  ::  Starting up DatabaseServiceHyperscale...
2022-08-17 19:35:44.2918  ::  DatabaseServiceHyperscale started.
```

DatabaseServiceHyperscale

```
--------------------------------------------------------------------------------
2022-08-17 19:35:44.6478  ::  Starting DatabaseServiceHyperscale...
2022-08-17 19:35:44.6609  ::  Initiating request LogServiceRequestGetLog.
2022-08-17 19:35:44.7180  ::  Request LogServiceRequestGetLog finished.
2022-08-17 19:35:44.7204  ::  Redoing log record 1,PageCreate,1,test
2022-08-17 19:35:44.7220  ::  Redone log record 1,PageCreate,1,test
2022-08-17 19:35:44.7229  ::  Redoing log record 2,PageRowInsert,1,test,1
2022-08-17 19:35:44.7250  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:44.7262  ::  Read the page with id 1 from cache.
2022-08-17 19:35:44.7276  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:44.7283  ::  Read the page with id 1 from cache.
2022-08-17 19:35:44.7289  ::  Redone log record 2,PageRowInsert,1,test,1
2022-08-17 19:35:44.7295  ::  Redoing log record 3,PageRowInsert,1,test,2
2022-08-17 19:35:44.7303  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:44.7310  ::  Read the page with id 1 from cache.
2022-08-17 19:35:44.7315  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:44.7330  ::  Read the page with id 1 from cache.
2022-08-17 19:35:44.7366  ::  Redone log record 3,PageRowInsert,1,test,2
2022-08-17 19:35:44.7394  ::  DatabaseServiceHyperscale started.
```

StorageService

LogService

```
2022-08-17 19:35:44.6906  ::  Getting entire log.
2022-08-17 19:35:44.6921  ::  Returning log records:
2022-08-17 19:35:44.6932  ::  -- 1,PageCreate,1,test
2022-08-17 19:35:44.6941  ::  -- 2,PageRowInsert,1,test,1
2022-08-17 19:35:44.6950  ::  -- 3,PageRowInsert,1,test,2
```

Слика 3. Стање конзола након наглог престанка рада рачунског чвора.

[22] Специфичне вредности записа дневника трансакција су редни број странице, име табеле и вредност која се додаје у страницу.

Након овога се покреће трансакција, уносе се нове вредности, узима се контролна тачка и проверава стање табеле. Слика 4 показује стање конзола након тих корака. Започињање трансакције рачунски чвор пре свега убележи код себе у управљачу трансакција, а потом прослеђује сервису дневника трансакција да се запис о тој операцији упише у трајну меморију. Што се тиче додавања нових вредности у табелу (3 и 4), то се дешава идентично као и раније. Узимање контролне тачке је урађено између додавања вредности да би се приказала функција контролне тачке у остатку система. Рачунски чвор прослеђује серверу страница сигнал да се контролна тачка треба узети[23], односно сервер страница то интерпретира као уписивање свега до одређеног редног броја дневника трансакција у трајну меморију. Серверу страница је потребан део дневника трансакција да би испунио овај захтев па потребни део дневника трансакција дохвата од сервиса дневника трансакција. Затим сервер страница примењује дневник трансакција па страницу уписује на диск. Након тога се проверава стање базе података из текућег клијента који је под отвореном трансакцијом и његово очекивано стање табеле јесте да су све четири вредности (1, 2, 3 и 4) присутне што и јесте случај.

DatabaseClient

```
> BEGIN TRANSACTION
2022-08-17 19:35:46.9069  ::  Initiating request DatabaseServiceRequestQuery.
2022-08-17 19:35:47.0725  ::  Request DatabaseServiceRequestQuery finished.
> INSERT INTO test VALUES 3
2022-08-17 19:35:47.0763  ::  Initiating request DatabaseServiceRequestQuery.
2022-08-17 19:35:47.2240  ::  Request DatabaseServiceRequestQuery finished.
> CHECKPOINT
2022-08-17 19:35:47.2259  ::  Initiating request DatabaseServiceRequestQuery.
2022-08-17 19:35:47.4780  ::  Request DatabaseServiceRequestQuery finished.
> INSERT INTO test VALUES 4
2022-08-17 19:35:47.4806  ::  Initiating request DatabaseServiceRequestQuery.
2022-08-17 19:35:47.5990  ::  Request DatabaseServiceRequestQuery finished.
> SELECT FROM test
2022-08-17 19:35:47.6009  ::  Initiating request DatabaseServiceRequestQuery.
2022-08-17 19:35:47.7529  ::  Request DatabaseServiceRequestQuery finished.
2022-08-17 19:35:47.7548  ::  test:1,2,3,4
```

DatabaseServiceHyperscale

```
2022-08-17 19:35:47.0476  ::  Transaction started.
2022-08-17 19:35:47.0519  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:47.0698  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:47.1830  ::  Reading pages for table test from cache.
2022-08-17 19:35:47.1892  ::  Read pages for table test from cache.
2022-08-17 19:35:47.1945  ::  1:test:1,2:3
2022-08-17 19:35:47.2006  ::  Adding [3] to table test
2022-08-17 19:35:47.2054  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:47.2131  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:47.2193  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:47.2211  ::  Read the page with id 1 from cache.
2022-08-17 19:35:47.2224  ::  Added [3] to table test
2022-08-17 19:35:47.3360  ::  Checkpoint started.
2022-08-17 19:35:47.3403  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:47.3476  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:47.3497  ::  Initiating request StorageServiceRequestCheckpoint.
2022-08-17 19:35:47.4690  ::  Request StorageServiceRequestCheckpoint finished.
2022-08-17 19:35:47.4734  ::  Checkpoint done.
2022-08-17 19:35:47.5814  ::  Adding [4] to table test
2022-08-17 19:35:47.5846  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:47.5930  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:47.5942  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:47.5964  ::  Read the page with id 1 from cache.
2022-08-17 19:35:47.5978  ::  Added [4] to table test
2022-08-17 19:35:47.7041  ::  Selecting table: test
2022-08-17 19:35:47.7085  ::  Returning table: test:1,2,3,4
```

StorageService

```
2022-08-17 19:35:47.3639  ::  Processing checkpoint with LSN 6.
2022-08-17 19:35:47.3658  ::  Catching up log from LSN -1 to LSN 6.
2022-08-17 19:35:47.3675  ::  Initiating request LogServiceRequestGetLog.
2022-08-17 19:35:47.4389  ::  Request LogServiceRequestGetLog finished.
2022-08-17 19:35:47.4407  ::  Redoing log record 1,PageCreate,1,test
2022-08-17 19:35:47.4423  ::  Redone log record 1,PageCreate,1,test
2022-08-17 19:35:47.4436  ::  Redoing log record 2,PageRowInsert,1,test,1
2022-08-17 19:35:47.4453  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:47.4463  ::  Read the page with id 1 from cache.
2022-08-17 19:35:47.4485  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:47.4494  ::  Read the page with id 1 from cache.
2022-08-17 19:35:47.4501  ::  Redone log record 2,PageRowInsert,1,test,1
2022-08-17 19:35:47.4509  ::  Redoing log record 3,PageRowInsert,1,test,2
2022-08-17 19:35:47.4515  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:47.4523  ::  Read the page with id 1 from cache.
2022-08-17 19:35:47.4531  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:47.4539  ::  Read the page with id 1 from cache.
2022-08-17 19:35:47.4549  ::  Redone log record 3,PageRowInsert,1,test,2
2022-08-17 19:35:47.4557  ::  Redoing log record 5,PageRowInsert,1,test,3
2022-08-17 19:35:47.4566  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:47.4574  ::  Read the page with id 1 from cache.
2022-08-17 19:35:47.4582  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:47.4602  ::  Read the page with id 1 from cache.
2022-08-17 19:35:47.4609  ::  Redone log record 5,PageRowInsert,1,test,3
2022-08-17 19:35:47.4617  ::  Caught up log from LSN 6 to LSN 6.
2022-08-17 19:35:47.4635  ::  Writing page to disk: 1:test:1,2,3:5
2022-08-17 19:35:47.4666  ::  Written page to disk: 1:test:1,2,3:5
2022-08-17 19:35:47.4676  ::  Processed checkpoint with LSN 6.
```

LogService

```
2022-08-17 19:35:47.0626  ::  Persisting log record: 4,TransactionBegin
2022-08-17 19:35:47.0642  ::  Writing log record to disk: 4,TransactionBegin
2022-08-17 19:35:47.0666  ::  Written log record to disk: 4,TransactionBegin
2022-08-17 19:35:47.0679  ::  Persisted log record: 4,TransactionBegin
2022-08-17 19:35:47.2082  ::  Persisting log record: 5,PageRowInsert,1,test,3
2022-08-17 19:35:47.2094  ::  Writing log record to disk: 5,PageRowInsert,1,test,3
2022-08-17 19:35:47.2105  ::  Written log record to disk: 5,PageRowInsert,1,test,3
2022-08-17 19:35:47.2120  ::  Persisted log record: 5,PageRowInsert,1,test,3
2022-08-17 19:35:47.3440  ::  Persisting log record: 6,Checkpoint,True
2022-08-17 19:35:47.3452  ::  Writing log record to disk: 6,Checkpoint,True
2022-08-17 19:35:47.3460  ::  Written log record to disk: 6,Checkpoint,True
2022-08-17 19:35:47.3468  ::  Persisted log record: 6,Checkpoint,True
2022-08-17 19:35:47.3800  ::  Getting entire log.
2022-08-17 19:35:47.3816  ::  Returning log records:
2022-08-17 19:35:47.3830  ::  -- 1,PageCreate,1,test
2022-08-17 19:35:47.3839  ::  -- 2,PageRowInsert,1,test,1
2022-08-17 19:35:47.3847  ::  -- 3,PageRowInsert,1,test,2
2022-08-17 19:35:47.3921  ::  -- 4,TransactionBegin
2022-08-17 19:35:47.4006  ::  -- 5,PageRowInsert,1,test,3
2022-08-17 19:35:47.4073  ::  -- 6,Checkpoint,True
2022-08-17 19:35:47.5871  ::  Persisting log record: 7,PageRowInsert,1,test,4
2022-08-17 19:35:47.5888  ::  Writing log record to disk: 7,PageRowInsert,1,test,4
2022-08-17 19:35:47.5901  ::  Written log record to disk: 7,PageRowInsert,1,test,4
2022-08-17 19:35:47.5917  ::  Persisted log record: 7,PageRowInsert,1,test,4
```

Слика 4. Стање конзола након започете трансакције.

[23] Овај корак је сувишан и додато је да би у систему *SHYLD* контролна тачка имала више функција, али и да би се показала асинхрона природа сервера страница.

Наредни корак је нагло прекидање рачунског чвора ради потврде да ће се трансакција поништити, што Слика 5 и показује. Рачунски чвор пролази кроз сличну процедуру као и након претходног наглог прекидања рада. Пошто је направљено још пар промена на бази редни број дневника трансакција на којој рачунски чвор оперише је већи (7), и приликом испоруке странице сервер страница ће морати поново да дохвати потребан део дневника трансакција. Сада то не мора радити од почетка дневника трансакција, већ само онај део који му је потребан. То је од 6 до 7, пошто је узимање контролне тачке већ подигло верзију сервера страница на редни број дневника трансакција 6. Страница која је испоручена рачунском чвору је '*1:test:1,2,3,4:7*'[24], што је и очекивано. Када рачунски чвор примени сав дневник трансакција након контролне тачке[25], рачунски чвор долази до закључка да је у тренутку наглог престанка рада трансакција била у току и започиње процедуру поништавања недовршене трансакције. Поништавање трансакције се ради тако што се обрнутим редоследом од промена у бази ради поништавање појединих корака[26]. Односно, прво се брише вредност 4 из странице, прескаче се запис из дневника трансакција који означава контролну тачку, јер је то логичка операција и њу нема смисла поништавати, па тек након тога се брише 3. Да би се одређени корак поништио, прави се нови запис у дневнику трансакција који трајно памти поништавање у трајној меморији[27]. Након што се пониште све промене до почетка трансакције, стање управљача трансакција се враћа у првобитно стање. У овом случају то је маркирање као да трансакција није у току и проглашавање опоравка базе готовим. Након тога ће се нови упити процесирати, односно од тог тренутка корисник може поново користити базу података.

[24] У формату странице делови су одвојени двотачком следећим редоследом: редни број странице, име табеле, листа вредности одвојених запетом и редни број трансакције која је последња примењена на страницу.

[25] СУБП зна да је све пре контролне тачке уписано у трајну меморију, па не мора примењивати дневник трансакција пре тога, јер ће ионако бити прескочено.

[26] У СУБП-у који подржава конкурентност, односно више паралелних трансакција, поништавање свих промена на бази од краја дневника трансакција до почетка трансакције није коректна акција, већ морају да се циљају промене које су битне само за актуелну трансакцију. Очигледно сваки запис у дневнику трансакција би морао да садржи неки идентификатор трансакције. Као додатну оптимизацију, сваки запис у дневнику трансакција би могао имати показивач на претходни запис те трансакције, где би крај те листе био запис у дневнику трансакција који обележава почетак те трансакције.

[27] Специфична вредност записа дневника трансакција је цела страница која се поништава.

DatabaseClient

```
2022-08-17 19:35:50.3593  ::  DatabaseServiceHyperscale exited.
2022-08-17 19:35:50.4697  ::  Starting up DatabaseServiceHyperscale...
2022-08-17 19:35:50.5468  ::  DatabaseServiceHyperscale started.
> SELECT FROM test
2022-08-17 19:35:53.2225  ::  Initiating request DatabaseServiceRequestQuery.
2022-08-17 19:35:53.3974  ::  Request DatabaseServiceRequestQuery finished.
2022-08-17 19:35:53.3983  ::  test:1,2
```

DatabaseServiceHyperscale

```
--------------------------------------------------------------------------------
2022-08-17 19:35:50.9576  ::  Starting DatabaseServiceHyperscale...
2022-08-17 19:35:50.9690  ::  Initiating request LogServiceRequestGetLog.
2022-08-17 19:35:51.0232  ::  Request LogServiceRequestGetLog finished.
2022-08-17 19:35:51.0299  ::  Redoing log record 6,Checkpoint,True
2022-08-17 19:35:51.0329  ::  Redone log record 6,Checkpoint,True
2022-08-17 19:35:51.0354  ::  Redoing log record 7,PageRowInsert,1,test,4
2022-08-17 19:35:51.0402  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:51.0433  ::  Page with id 1 not found in cache.
2022-08-17 19:35:51.0476  ::  Initiating request StorageServiceRequestGetPage.
2022-08-17 19:35:51.0834  ::  Request StorageServiceRequestGetPage finished.
2022-08-17 19:35:51.0872  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:51.0882  ::  Page with id 1 not found in cache.
2022-08-17 19:35:51.0899  ::  Skipping apply of log record: 7,PageRowInsert,1,test,4
2022-08-17 19:35:51.0910  ::  Redone log record 7,PageRowInsert,1,test,4
2022-08-17 19:35:51.0943  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:51.1259  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:51.1277  ::  Redoing log record 8,Undo,7,PageRowInsert,1,test,4
2022-08-17 19:35:51.1303  ::  Undoing log record 7,PageRowInsert,1,test,4
2022-08-17 19:35:51.1320  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:51.1350  ::  Read the page with id 1 from cache.
2022-08-17 19:35:51.1372  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:51.1386  ::  Read the page with id 1 from cache.
2022-08-17 19:35:51.1397  ::  Undone log record 7,PageRowInsert,1,test,4
2022-08-17 19:35:51.1410  ::  Redone log record 8,Undo,7,PageRowInsert,1,test,4
2022-08-17 19:35:51.1420  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:51.2477  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:51.2502  ::  Redoing log record 9,Undo,5,PageRowInsert,1,test,3
2022-08-17 19:35:51.2515  ::  Undoing log record 5,PageRowInsert,1,test,3
2022-08-17 19:35:51.2529  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:51.2559  ::  Read the page with id 1 from cache.
2022-08-17 19:35:51.2574  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:51.2586  ::  Read the page with id 1 from cache.
2022-08-17 19:35:51.2604  ::  Undone log record 5,PageRowInsert,1,test,3
2022-08-17 19:35:51.2619  ::  Redone log record 9,Undo,5,PageRowInsert,1,test,3
2022-08-17 19:35:51.2633  ::  Initiating request LogServiceRequestPersistLogRecord.
2022-08-17 19:35:51.3716  ::  Request LogServiceRequestPersistLogRecord finished.
2022-08-17 19:35:51.3734  ::  DatabaseServiceHyperscale started.
2022-08-17 19:35:53.3600  ::  Selecting table: test
2022-08-17 19:35:53.3678  ::  Reading pages for table test from cache.
2022-08-17 19:35:53.3743  ::  Read pages for table test from cache.
2022-08-17 19:35:53.3814  ::  1:test:1,2:9
2022-08-17 19:35:53.3876  ::  Returning table: test:1,2
```

StorageService

```
2022-08-17 19:35:51.0593  ::  Getting page with id 1 with LSN 7.
2022-08-17 19:35:51.0605  ::  Catching up log from LSN 6 to LSN 7.
2022-08-17 19:35:51.0613  ::  Initiating request LogServiceRequestGetLog.
2022-08-17 19:35:51.0664  ::  Request LogServiceRequestGetLog finished.
2022-08-17 19:35:51.0675  ::  Redoing log record 7,PageRowInsert,1,test,4
2022-08-17 19:35:51.0694  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:51.0707  ::  Read the page with id 1 from cache.
2022-08-17 19:35:51.0717  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:51.0728  ::  Read the page with id 1 from cache.
2022-08-17 19:35:51.0738  ::  Redone log record 7,PageRowInsert,1,test,4
2022-08-17 19:35:51.0748  ::  Caught up log from LSN 7 to LSN 7.
2022-08-17 19:35:51.0760  ::  Reading the page with id 1 from cache.
2022-08-17 19:35:51.0769  ::  Read the page with id 1 from cache.
2022-08-17 19:35:51.0778  ::  Returning page:1:test:1,2,3,4:7
```

LogService

```
2022-08-17 19:35:50.9910  ::  Getting entire log.
2022-08-17 19:35:50.9976  ::  Returning log records:
2022-08-17 19:35:51.0007  ::  -- 1,PageCreate,1,test
2022-08-17 19:35:51.0017  ::  -- 2,PageRowInsert,1,test,1
2022-08-17 19:35:51.0026  ::  -- 3,PageRowInsert,1,test,2
2022-08-17 19:35:51.0036  ::  -- 4,TransactionBegin
2022-08-17 19:35:51.0047  ::  -- 5,PageRowInsert,1,test,3
2022-08-17 19:35:51.0072  ::  -- 6,Checkpoint,True
2022-08-17 19:35:51.0083  ::  -- 7,PageRowInsert,1,test,4
2022-08-17 19:35:51.0621  ::  Getting log with min LSN: 6
2022-08-17 19:35:51.0639  ::  Returning log records:
2022-08-17 19:35:51.0652  ::  -- 7,PageRowInsert,1,test,4
2022-08-17 19:35:51.1151  ::  Persisting log record: 8,Undo,7,PageRowInsert,1,test,4
2022-08-17 19:35:51.1176  ::  Writing log record to disk: 8,Undo,7,PageRowInsert,1,test,4
2022-08-17 19:35:51.1192  ::  Written log record to disk: 8,Undo,7,PageRowInsert,1,test,4
2022-08-17 19:35:51.1209  ::  Persisted log record: 8,Undo,7,PageRowInsert,1,test,4
2022-08-17 19:35:51.2374  ::  Persisting log record: 9,Undo,5,PageRowInsert,1,test,3
2022-08-17 19:35:51.2434  ::  Writing log record to disk: 9,Undo,5,PageRowInsert,1,test,3
2022-08-17 19:35:51.2447  ::  Written log record to disk: 9,Undo,5,PageRowInsert,1,test,3
2022-08-17 19:35:51.2459  ::  Persisted log record: 9,Undo,5,PageRowInsert,1,test,3
2022-08-17 19:35:51.3622  ::  Persisting log record: 10,TransactionRollback
2022-08-17 19:35:51.3654  ::  Writing log record to disk: 10,TransactionRollback
2022-08-17 19:35:51.3679  ::  Written log record to disk: 10,TransactionRollback
2022-08-17 19:35:51.3698  ::  Persisted log record: 10,TransactionRollback
```

Слика 5. Стање конзола након прекинуте трансакције.

4. Закључак

У данашњем развоју софтвера, хоризонтално скалирање СУБП-а је неопходно и широко коришћено. Релациони модел је стриктан и одржавање његових карактеристика постаје за ред величине теже приликом хоризонталног скалирања. У зависности од других ограничења, ово може бити и немогуће. Многи универзитети, компаније и појединци су посветили време да реше комплексне проблеме хоризонталног скалирања, али често се морао направити компромис из једног или другог разлога. Због тога, постоји много различитих врста СУБП-а који су прилагођени одређеним сценаријима. Често у тим сценаријима постоји конкуренција између различитих компанија која природно води ка иновацијама.

У овом раду, детаљно је описан један вид хоризонталног скалирања *Azure SQL Database Hyperscale*, што је испраћено и имплементацијом система *SHYLD*. Да би се омогућило независно скалирање процесорске и складишне моћи приказано је да је било неопходно направити компромисе који су резултовали већом комплексношћу и ценом, као и споријим перформансама у одређеним случајевима. Идеално скалирање СУБП-а не постоји, већ се прави за одређену намену, односно скалирање се дизајнира на такав начин да реши проблем корисницима и увек има неке компромисе.

Референце

[1] E. F. Codd, „A relational model of data for large shared data banks,“ *Communications of the ACM,* т. 13, бр. 6, pp. 377-387, јун 1970.

[2] M. T. Ozsu и P. Valduriez, Principles of Distributed Database Systems, Springer, 2011.

[3] Microsoft, „Transactional Replication,“ 29 11 2021. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/sql/relational-databases/replication/transactional/transactional-replication?view=sql-server-ver16. [Последњи приступ 31 7 2022].

[4] Microsoft, „Azure Synapse SQL architecture,“ 25 5 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/synapse-analytics/sql/overview-architecture. [Последњи приступ 30 7 2022].

[5] Microsoft, „What is dedicated SQL pool (formerly SQL DW) in Azure Synapse Analytics?,“ 18 2 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-overview-what-is?context=%2Fazure%2Fsynapse-analytics%2Fcontext%2Fcontext. [Последњи приступ 30 7 2022].

[6] Microsoft, „Serverless SQL pool in Azure Synapse Analytics,“ 26 4 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/synapse-analytics/sql/on-demand-workspace-overview. [Последњи приступ 30 7 2022].

[7] Apache, „Cassandra Documentation - Architecture - Overview,“ [На мрежи]. Доступно: https://cassandra.apache.org/doc/latest/cassandra/architecture/overview.html. [Последњи приступ 31 7 2022].

[8] J. C. Corbett и е. а. , „Spanner: Google’s Globally-Distributed Database,“ *Proceedings of OSDI 2012,* pp. 1-14, 2012.

[9] Amazon, „Working with Aurora multi-master clusters,“ 2022. [На мрежи]. Доступно: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-multi-master.html. [Последњи приступ 31 7 2022].

[10] Amazon, „Build highly Доступно MySQL applications using Amazon Aurora Multi-Master,“ 8 8 2019. [На мрежи]. Доступно: https://aws.amazon.com/blogs/database/building-highly-available-mysql-applications-using-amazon-aurora-mmsr/. [Последњи приступ 31 7 2022].

[11] Microsoft, „Hyperscale distributed functions architecture,“ 22 6 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/azure-sql/database/hyperscale-architecture?view=azuresql. [Последњи приступ 8 7 2022].

[12] Microsoft Mechanics, „What is Azure SQL Database Hyperscale?,“ 8 7 2019. [На мрежи]. Доступно: https://www.youtube.com/watch?v=Z9AFnKI7sfI. [Последњи приступ 9 7 2022].

[13] R. Arpaci-Dusseau и A. Arpaci-Dusseau, „Beyond Physical Memory: Mechanisms,“ у *Operating Systems: Three Easy Pieces*, 2013.

[14] Microsoft, „High Availability replica,“ 22 6 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tier-hyperscale-replicas?view=azuresql&tabs=portal#high-availability-replica. [Последњи приступ 9 7 2022].

[15] Microsoft, „Named replica,“ 22 6 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tier-hyperscale-replicas?view=azuresql&tabs=portal#named-replica. [Последњи приступ 9 7 2022].

[16] Microsoft, „Azure SQL Database Hyperscale named replicas FAQ,“ 22 6 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tier-hyperscale-named-replicas-faq?view=azuresql#how-much-delay-is-there-between-the-primary-and-secondary-compute-replicas-. [Последњи приступ 9 7 2022].

[17] Microsoft, „The Transaction Log (SQL Server),“ 9 10 2020. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/sql/relational-databases/logs/the-transaction-log-sql-server?view=sql-server-ver16. [Последњи приступ 8 7 2022].

[18] Microsoft, „Quickstart: Backup and restore a SQL Server database on-premises,“ 31 5 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/sql/relational-databases/backup-restore/quickstart-backup-restore-database?view=sql-server-ver16. [Последњи приступ 8 7 2022].

[19] Microsoft, „Blob snapshots,“ 7 7 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/storage/blobs/snapshots-overview. [Последњи приступ 9 7 2022].

[20] Microsoft, „General Purpose service tier - Azure SQL Database and Azure SQL Managed Instance,“ 17 6 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tier-general-purpose?view=azuresql. [Последњи приступ 14 8 2022].

[21] Microsoft, „Business Critical tier - Azure SQL Database and Azure SQL Managed Instance,“ 9 6 2022. [На мрежи]. Доступно: https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tier-business-critical?view=azuresql. [Последњи приступ 14 8 2022].

[22] K. Delaney, SQL Server 2012 Internals, Microsoft Press, 2013.