Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Techniky virtuálně synchronních replikací v relačních databázích

Bc. Radim Špigel

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Radim Špigel

Informatika

Název práce

Techniky virtuálně synchronních replikací v relačních databázích

Název anglicky

Virtually synchronous replication techniques in relational databases

---

Cíle práce

Diplomová práce je tematicky zaměřena na technologie a architekturu replikací v ekosystému relačních databází. Hlavním cílem diplomové práce je navrhnout a realizovat migraci existujícího aplikačního systému využívající synchronně replikovanou databázovou infrastrukturu na systém virtuálně synchronních replikací mezi čtyřmi a více databázovými uzly. Dílčí cíle diplomové práce jsou:
- vytvoření kritické literární rešerše k problematice konceptů replikací dat v relačních databázích, především v MySQL ekosystému,
- analyzovat současný stav a požadavky pro návrh a optimalizaci relačního databázového uložiště,
- navrhnout řešení z oblasti virtuálně synchronních replikací,
- syntetizovat výsledky práce a formulovat přínosy a závěry práce.

Metodika

Metodika řešené problematiky diplomové práce vychází ze studia a analýzy odborných informačních zdrojů. Praktická část práce je zaměřena na vypracování případové studie analyzující materiály dostupné a získané z vybrané společnosti, ve které probíhá implementace reklamního systému, které se student účastní. Na základě syntézy teoretických poznatků a výsledků praktické části práce budou formulovány závěry diplomové práce.

Doporučený rozsah práce

60-80

Klíčová slova

relační databáze, synchronní replikace, mysql, galera

Doporučené zdroje informací

Baron Schwartz, Peter Zaitsev, Vadim Tkachenko: High Performance MySQL, 3rd Edition Optimization, Backups, and Replication. 2012. ISBN:978-1-449-31428-6

Bill Wilder: Cloud Architecture Patterns. 2012. ISBN: 9781449357979

DYER, Russell J. T. MySQL in a nutshell. 2nd ed. Sebastopol, CA: O'Reilly, c2008. ISBN 978-0-596-51433-4.

MAVRO, PIERRE. Mariadb High Performance. 23. září 2014. Lightning Source UK, 2014. ISBN 9781783981601.

Pedone, F., Guerraoui, R. & Schiper, A. Distributed and Parallel Databases (2003) 14: 71. doi:10.1023/A:1022887812188

Pokorný J.: Database Architectures: Current Trends and their Relationships to Requirements of Practice , in Advances in Information Systems Development: New Methods and Practice for the Networked Society , Springer Science+Business Media, ISBN: 978-0-387-70760-0, pp. 267-277, 2007

Pokorný J.: Database technologies in the world of Big Data, in Proceedings of the 16th International Conference on Computer Systems and Technologies, Dublin, ACM, ISBN: 978-1-4503-3357-3, pp. 1-12, 2015

Předběžný termín obhajoby

2017/18 LS – PEF

Vedoucí práce

Ing. Jan Tyrychtr, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 31. 10. 2017

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 1. 11. 2017

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 28. 03. 2018

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Techniky virtuálně synchronních replikací v relačních databázích" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 28.3. 2018 _____

Poděkování

Rád bych touto cestou poděkoval vedoucímu diplomové práce Ing. Janu Tyrychtrovi, Ph.D. za ochotu a odborné vedení této diplomové práce. Dále osobním konzultantům Ing. Tomášovi Komendovi a Michalu Fizkovi za trpělivé vysvětlování a konzultace, principu replikací a databázových systémů a rady při implementaci.
V neposlední řadě také firmě Seznam.cz a.s., která umožnila případovou studii.

Techniky virtuálně synchronních replikací v relačních databázích

Abstrakt

Účelem této diplomové práce je seznámení s problematikou replikací v relačních databázových clusterech a možnostmi škálování velkých databázových systémů a navržení rozšíření pro současné řešení škálování dat systému Sklik.cz. V práci jsou porovnávány přístupy asynchronních replikací a nových implementací synchronních replikací. Dále zahrnuje porovnání různých implementací synchronních replikací. Je zde nastíněno, jaké architektury je možné použít při stavbě databázových clusteru. Práce popisuje analýzu a návrh rozšíření serveru o vlastní řešení, které umožňuje škálování dat v systému Sklik.cz. Získané poznatky z teoretických východisek jsou aplikovány při návrhu rozšíření stávajícího řešení škálování dat v systému Sklik.cz. Práce obsahuje také popis pseudokódu implementace, testování asynchronního databázového clusteru a databázového clusteru využívající Galera cluster technologii. V neposlední řadě jsou zde uvedeny výstupy z testování implementované verze řešení, kdy se porovnávalo řešení single master a multi master, na simulaci práce se systémem Sklik.cz.

Klíčová slova: relační databáze, synchronní replikace, mysql, galera, škálování dat

Virtually synchronous replication techniques in relational databases

Abstract

The goal of this thesis is to acquaint one with replication problematic in relational database clusters and also with the possibilities of scaling database systems and designe extention for current scaling solution for Sklik.cz system. This thesis includes a comparison of asynchronous replication and a new implementation of synchronous replication. It also contains an overview about types of architectures possible to use with building database clusters. The thesis decribes the analysis and design of extension of the server with an author's own solution which allows one to scale the data in Sklik.cz system. The knowledge gained from theoretical base is applied for design extension of current sharding solution on Sklik.cz. Furthermore, the work itself contains a description in pseudocode, a comparison of a certain database cluster - which uses asynchronous replication - with a cluster that uses synchronous Galera cluster technology. The ending is dedicated to the results of the test of an implementation of the extension to the solution, in which there are the single master and multi master solutions on a simulated work with Sklik.cz system compared.

Keywords: relational database, synchronous replication, mysql, galera, data scaling

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Zaznamenávání a uchovávání dat v konzistentní podobě je v současné době velice důležitým faktorem pro všechny systémy či aplikace.

Systémy, které využívají relační databázové systémy a v nichž může rozsah dat narůst do obrovských rozměrů, potřebují optimalizovat přístup k datům. K tomuto problému slouží několik řešení, jako je použití vertikálního škálování, které je ovšem velice omezené. Další možností je využití master-slave architektury za použití asynchronních replikací, rozdělení dat do více relačních databázových systémů, škálování dat nebo kombinace.

Relační databázové systémy využívající asynchronní replikace naráží na několik problémů. Prvním velice závažným problémem je nemožnost škálovat zápisové operace. To způsobuje, že databázové systémy využívající asynchronní replikace mají pouze jeden hlavní bod, na který jsou směřovány všechny zápisové operace a ostatní členové si pouze aplikují změny často s vysokou latencí.

Druhý zásadní problém je vznik nekonzistence dat při použití asynchronních replikací napříč dvěma datovými centry, kdy je přepínán zápis z jednoho datového centra do druhého. Při tomto úkonu se může stát, že vzniknou nekonzistence, kdy aplikace či servery připojené do databázového clusteru budou zapisovat do obou instancí zároveň. Změny provedené tímto způsobem je často nemožné zpětně seskupit a nepřijít o důležitá data, jež již byla potvrzena uživatelům. Tento problém může být pro systémy, které spoléhají na konzistenci dat, jako jsou bankovní systémy, fatální.

Aby byly tyto problémy eliminovány, vznikaly různé implementace synchronních replikací. Ty ovšem nebyly výkonnostně dostačující, proto se databáze stále stavěly do asynchronně replikovaných clusterů.

Na přelomu milénia byl představen nový postup, jak implementovat synchronní replikace s minimálním dopadem na výkon databázového clusteru. Díky tomuto postupu bylo implementováno několik řešení, která budou v této práci představena a porovnána.

Využití škálování dat či rozdělení dat vyžaduje přídavnou aplikaci či server, který bude definovat, kde se data nachází. Tato přídavná aplikace může být řešena buď externě, nebo vlastní implementací.

V rámci úvodu byly zmíněny některé pojmy týkající se problematiky rozložení dat, jejich replikací či jejich uchovávání. Je zřejmé, že se jedná o velmi komplexní oblast

informačních technologií, která ovliňuje nejen správu databáze, ale i návrh databází, aplikací a systémů. Jednotlivým oblastem se budou věnovat kapitoly, jež budou zaměřeny především na problematiku týkající se distribuce a škálování dat v databázovém clusteru, možnosti jejich replikací a přístupu k těmto datům.

V praktické části bude pospáno řešení implementace serveru, který umožňuje směřovat data do databázových clusterů. Dále zde bude popsáno, jak tento server upravit, aby mohl využívat výhod synchronních replikací databázového clusteru. V neposlední řadě zde bude porovnáno řešení využívající potenciál synchronně replikovaného clusteru (multi master) a přístupu, kdy se přistupuje k takto replikovanému clusteru jako při asynchronních replikacích, tedy pouze k masteru, tzn. single master konfigurace.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této diplomové práce je formulovat teoretická východiska týkající se principů a možností synchronních replikací u relačních databázových clusterů porovnat různé technologie synchronních replikací. Vyhodnotit rozdíly mezi asynchronními a synchronními replikacemi. Formulovat různé způsoby jak, optimalizovat databázové systémy obsahující obrovské množství dat a do jakých architektur se relační databázové clustery dají stavět.

Hlavním úkolem je aplikovat nastudované informace pro návrh rozšíření stávajícího řešení manažeru, který spravuje komunikaci mezi databázovými clustery.

2.2 Metodika

Metodika diplomové práce vychází ze studia a analýzy odborných informačních zdrojů. Praktická část je zaměřena na případovou studii systému Sklik.cz a rozšíření stávajícího řešení. Dále pak porovnává různé techniky replikací databázových systémů.

2.2.1 Zpracování teoretických východisek

V rámci teoretické části je nejprve nutné definovat základní teorémy, jako je ACID a CAP teorém. Dále pak popsat problémy, které mohou nastat při vícenásobném přístupu, a definice úrovně izolovanosti transakcí, kdy má databázový systém jednu či více instancí.

Na tyto teorémy navazuje představení replikací a toho, do jakých architektur je možné stavět databázové clustery. Bude zde popsáno, jakým způsobem je možné optimalizovat rozsáhlé databázové systémy za pomoci technik rozložení dat. V teoretické části budou také porovnány různé implemntace synchronních replikací, a to Galera cluster a Group replikace.

2.2.2 Postup úpravy

Praktická část se zaměří na porovnání počtu zpracovaných transakcí databázového clusteru za využití asynchronních replikací a jiného typu databázového clusteru, který využívá virtuálně synchronní řešení Galera cluster.

Další důležitou částí je analýza stávajícího řešení optimalizace dat systému Sklik.cz, které využívá horizontálního škálování, kdy jsou data rozložena do více databázových systémů. Aby bylo možné přistupovat k těmto datům, bude nutné rozšířit server, který směřuje dotazy na základě předem dané logiky do správného databázového systému. Pro rozšíření tohoto řešení za účelem využití možností, které mají virtuálně synchronní replikace, bude v této práci navrženo několik možností, jak stávající server rozšířit. Práce bude obsahovat i popis několika alternativ, tedy externích řešení, které by bylo možné použít pro rozložení zátěže.

Vybraná možnost bude naimplementována v jazyce C++. V práci bude popsán pouze pseudokód této implementace.

Tato práce také obsahovat testy originálního a nového řešení, které bylo zvoleno a aplikováno. Tyto testy se zaměří na simulaci provozu nad systémem Sklik.cz, a to převážně vytvářecí a listovací operace nad kampaněmi, sestavami a inzeráty. Každý tento úkon bude proveden pro deset a více uživatelů, kteří přistupují k systému paralelně v rámci testů.

3 Teoretická východiska

3.1 Základní principy relačních databází

V této kapitole jsou popsány základní pojmy týkající se relačních databází, jejich architektury a druhy replikací. Zaměřím se na problematiku databází z pohledu běžící transakce, která se vykonává nad stejnými daty souběžně. Dalším tématem, které zde bude probráno je téma týkající se nejběžnějšími druhy architektur, do kterých je možné stavět databázové clustery.

Relační databázové systémy slouží k uchovávání dat, která jsou důležitá pro širokou škálu informačních systému jako jsou bankovní systémy, e-shopy či další systémy poskytující internetové služby. Pro tyto data je nutné, aby byli uloženy ve spolehlivých a rychle dostupných uložištích. [1]

3.1.1 Základní teorémy

Databáze jako takové pro účel uchovávání a dostupnosti dat by měly splňovat podmínky dané teorémy. Bavíme-li se o jedné instanci relační databáze musí splňovat teorém ACID. V kontextu clusteru musí jednotlivé instance plnit podmínky teorému ACID, a zároveň celý cluster plnit podmínky teorému CAP.

3.1.1.1 ACID teorémy

Skrze požadavek velké dostupnosti validních dat, nekonzistencí a možnosti zpracování více transakcí je nutné, aby databáze plnily ACID teorém. Tento teorém popisuje, jaké vlastnosti musí mít databáze při zpracování transakcí, aby nedošlo k poškození dat nebo uložení dat, které nejsou žádaná. [2]

- **Atomicita (A - atomic) –** Atomicita umožňuje, aby transakce byla provedena jako celek. Znamená to tedy, že jsou dva výsledky transakce. První možnost znamená, že transakce je provedena úspěšně jako celek, tedy všechny změny jsou potvrzeny a uloženy. Druhá možnost je, že při provádění transakce vznikla chyba, buď v celku, nebo i její části a to znamená, že jsou data ve stavu, v jakém byla před spuštěním transakce. Jako by transakce vůbec neběžela.
- **Konzistence (C - consistency) –** Transakce, která mění data, musí tyto data vždy měnit konzistentně, tedy splňovat integritní omezení (Integritní omezení určují typy

dat, rozsahy hodnot a struktury, kterých mohou data uložená v databázi nabývat). Pokud nastane situace, kdy by transakce měla porušit nějaké integritní omezení, tato vlastnost zajišťuje, že tato transakce bude ukončena s chybou.

- **Izolovanost (I - isolation)** – Tato vlastnost zajišťuje, že transakce pracují s daty, které byly uloženy před ukončenou transakcí. Druhů izolovanosti je více a podrobněji jsou popsány ve vlastní podkapitole (Úrovně izolovanosti transakcí v rámci jedné instanci relační databáze).
- **Trvalost (D - durability) -** Trvalost zajišťuje, že změny nad daty, které byly úspěšně provedeny v rámci transakcí, jsou trvalé, a to i v případě krizových situací jako je selhání softwaru či hardwaru. Znamená to, že je zajištěno permanentní uložení na perzistentních uložištích.

Databázový systém, který běží pouze v jedné instanci, dokáže zpracovat vždy jen určité množství dotazů v závislosti na druhu dotazů, softwarových a hardwarových prostředcích, které jí byli dány. Při přiblížení se k limitům, které databáze vyžaduje pro odbavení provozu, který přichází do dané instance, je nutné tuto situaci řešit.

Existuje několik způsobů, jak řešit problém s vytížením. Pokud dokážeme data rozdělit do více částí, můžeme využít tzv. partition přístup, tedy rozdělíme rozsáhlé tabulky na menší celky a rozdělíme je na více samostatných instancí. Nad těmito instancemi poté můžeme balancovat pomocí databázového systému, který má uložené informace, kde se jaká část tabulek (informací) nachází. [2]

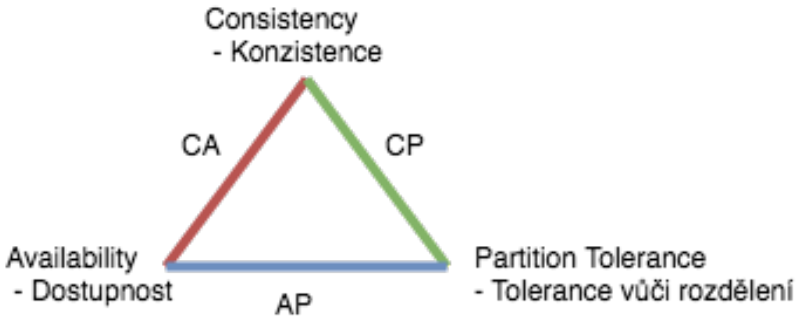
Pokud ovšem máme stav kdy data není lehké rozdělit nebo je nutné přistupovat k nim jako k celku, je nutné stavět databáze do tzv. clusterů. Stejná data jsou uložena na více instancí a nad těmito instancemi balancujeme. Tedy rozkládáme zátěž, kterou by musela zpracovat pouze jedna instance. Dále tomuto přístupu se zvyšuje spolehlivost databáze z pohledu dostupnosti. Pokud je cluster dobře navržený umožní nám odbavovat dotazy, i když jedna z instancí vypadne.

3.1.2 CAP teorém u relačních databázových clusterů

Tento teorém byl definován v roce 2000 Ericem Brewerem. Může být znám i pod jménem Brewerův teorém. Definuje vlastnosti, které by měly splňovat distribuované databázové sklady. Přesněji řečeno mohou splňovat pouze dvě ze tří. [3]

- **Konzistence (C - consistency) –** Pro distribuované systémy je chápání konzistence odlišné oproti konzistenci u ACID teorému. Tato vlastnost znamená, že pro čtení je garantováno navrácení posledního zápisu pro daného klienta nebo je navrácena chyba.
- **Dostupnost (A - availability) -** Dostupnost nám říká, že pro každý uzel, který není v chybovém stavu, vrátí na každé čtení odpověď, která není chybou. Tuto odpověď vrátí v rozumném časovém intervalu. Ovšem není zajištěno, že data budou nejaktuálnější.
- **Tolerance vůči rozdělení (P – partition tolerance) –** Tato vlastnost nám zaručuje, že systém funguje nezávisle na počtu chyb nebo zpožděných zpráv v rámci sítě daného clusteru.

Obrázek 1: CAP teorém



Zdroj: Vlastní zpracování [3]

Při distribuování relačních databází do clusterů víme, že v rámci jedné instance musí relační databáze zajistit teorém ACID, ale v rámci clusteru musím zajistit dvě ze tří vlastnosti CAP teorému.

Vlastnosti tolerance vůči rozdělení v clusteru kvůli rozložení dat a zátěže do více instancí je vždy naplněna. To znamená, že jsou dvě možnosti a to, že zajistíme konzistenci, anebo dostupnost.

- **AP systém -** Pokud systém zaměříme na dostupnost, získáme pro každý dotaz nechybovou odpověď a to i v případě rozdělením dat na více částí na různé stroje. Databáze, které využívají AP systém CoucheDb, Cassandra.
- **CP systém –** Systémy, které mají rozptýlená data a zároveň garantují to, že čtení nám vrátí vždy nejčerstvější data nebo je navrácena chyba. Tento systém používá například HBase, Redis, MongoDB.

- **AC systém –** Tyto systémy jsou založeny na dostupnost a zároveň na konzistenci dat to znamená, že je zde nulová tolerance vůči rozdělení dat. AC systémy jsou vlastně všechny relační systémy o jedné instanci, které splňují ACID teorém.

3.1.3 Problémy nastávající při vícenásobném přístupu

Pokud se bavíme o ideálním světě, můžeme se dívat na databázový systém jako na systém, který v jeden čas odbavuje pouze jedinou transakci, proto v tomto případě nemůže dojít k žádné anomálii.

Reálný svět se právě z rychlostních důvodů liší. Pro práci s rozsáhlými daty potřebujeme, aby databázový systém odbavoval více transakcí najednou, a to dokonce nad stejnými řádky tabulek. Díky tomuto přístupu vznikají anomálie jako jsou například tzv. špinavé čtení a další. [4]

3.1.3.1 Anomálie při kolizi souběžných transakcí nad stejnými daty

· **Špinavé čtení** - Máme transakci ***A*,** která čte data, která upravuje zároveň transakce ***B*,** ale změny transakce ***B*** ještě nebyly potvrzeny. Pokud tedy transakce ***B*** skončí s chybou, navrátí se změněná data na původní hodnoty. Transakce ***A*** v tomto případě přečetla nekonzistentní data.

· **Neopakovatelné čtení –** Pokud transakci ***A*** čte opakovaně hodnoty ze stejného řádku, ale dostává rozdílné hodnoty, aniž by se data měnila. Tato situace nastává, když daná data zároveň mění jiné transakce.

· **Vznik fantomů –** Fantomy se například objevují například, pokud se v transakci vícenásobně počítá počet záznamů, ale vrácené výsledky jsou rozdílné díky činnosti souběžných transakcí nad stejnými daty.

· **Dvojité čtení –** Nastává v případě, pokud transakci ***A*** čte data pomocí indexu, a mezitím transakci ***B*** změní hodnotu na řádku, který již transakci ***A*** měla přečtený a tím jej při aktualizaci zařadila nakonec indexu, kde jej opět přečte transakci ***A*.**

Pokud máme systém zaměřený na správnost vydávaných a zapisovaných dat a chceme paralelní vykonávání transakcí, je nutné určit meze jak moc se mohou transakce ovlivňovat.

Pokud bychom uvažovali o nejstriktnější úrovni izolovanosti tak dosáhneme vlastně sériového zpracování transakcí. To ovšem může mít obrovský dopad pro určité typy

dotazů, a proto je nutné dělat kompromisy. K tomuto účelu byli definovány různé stupně izolovanosti, ve kterých se objevují různé typy anomálií.

3.1.3.2 Úrovně izolovanosti transakcí v rámci jedné instanci relační databáze

Před popsáním úrovně izolovanosti v clusteru je potřeba znát úrovně izolovanosti v rámci jedné instance relačního databázového systému. [5]

- **Čtení nepotvrzených dat (Read uncommitted) –** Nejméně striktní úroveň izolace transakcí. Transakce může číst data měněná souběžnými transakcemi, které ještě nebyla potvrzena. Dochází ke špinavému a neopakovatelnému čtení. [6]
- **Čtení potvrzených dat (Read committed) –** Při této úrovni je povoleno jen čtení dat, která jsou potvrzena, aby to bylo možné je nutné použít zámky. Při tomto stupni izolovanosti může nastat tzn. uváznutí, kdy dvě transakce vzájemně čekají na uvolnění zamčených dat druhou transakcí. [6]
- **Opakovatelné čtení (Repeatable Read) –** Opakovatelné čtení zajišťuje to, že stejný dotaz vrátí vždy stejné výsledky. Pokud dotaz přečte data, zajistí zachování aktuálního stavu pro následná použití (nejčastěji zámkem nebo verzováním). [6]
- **Sériové zpracování (Serializable) –** Tato úroveň má největší důraz na izolovanost transakcí. Paralelní běžící transakce nemají možnost navzájem ovlivňovat data se kterými pracují. [6]

3.1.3.3 Úrovně izolovanosti transakcí v rámci clusteru relační databáze

Zabýváme-li se úrovněmi izolovanosti v rámci clusteru, musím rozšířit úrovně izolovanosti o další úroveň díky škálovatelnosti dotazů napříč clusterem.
Díky balancovaní konkurenčních transakcí nad jednotlivými instancemi databázového systému, mohou nastat dvě situace. [6]

1. **Konkurenční transakce balancované na stejnou instanci v clusteru**

   Transakce se potkají na stejné instanci clusteru. To jak se k případným kolizním operací nad stejnými daty přistoupí, určí nastavení izolační úrovně databáze. Stejně jako v případě, kdy by databázový systém byl jen o jediné instanci.

2. **Konkurenční transakce rozložené na různé instance v clusteru**

   Pokud nastane stav, že se konkurenční transakce spustí na různých instancích clusteru, navzájem na sebe nevidí, neřeší tedy, že současně přistupují ke stejným datům.

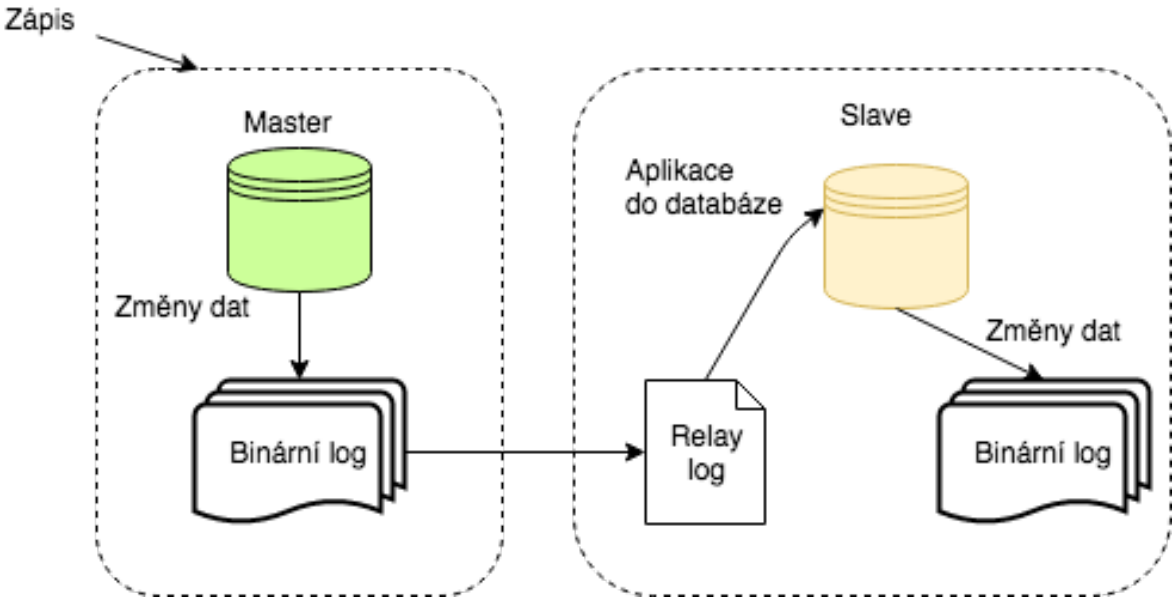
Jakmile jsou prováděny transakce nad stejnými daty na různých místech v clusteru, je nutné řešit, jak se ke kolizím zachovat. Buď obě transakce skončí s chybou a budou navráceny jejich změny, nebo první transakce požádala o potvrzení a bude zachována, a všechny ostatní skončí s chybou.

Na základě tohoto chování je vhodné, aby kolizní transakce v rámci stejného dotazu balancovaly nad stejnou instancí, kde se korektně provedou, než aby polovina končila s chybou a musela se opakovat.

3.1.4 Replikace

Replikace je mechanismus, který umožňuje, že všechny změny potvrzených transakcí provedené v rámci jedné instance jsou přeneseny na další členy clusteru. Replikační mechanismy se dělí do dvou základních skupin, a to asynchronní replikace a synchronní replikace. [5]

Obrázek 2: Asynchronní replikace, Master - Slave



Zdroj: Vlastní zpracování

Pro předání dat z instance na instanci se používají tyto dva způsoby stavově založené replikace, řádkově založení replikace nebo jejich kombinace.

· **Stavově založené replikace (State-base) –** Změny jsou přenášeny mezi instancemi v podobě příkazů, které instance vykonala. Tyto příkazy přebírá druhá instance a vykoná je nad svými daty. [7]

· **Řádkově založení replikace (Row-base) –** Instance vykonává příkazy, které jsou na ni zaslány. Přenášeny na ostatní instance clusteru jsou pouze výsledné změněné řádky tabulky. Ostatní instance daný dotaz nevykonávají. Nahrazují své řádky v tabulce, řádkami přijatými. [7]

3.1.5 Asynchronní replikace

Při asynchronních replikací se zápisové operace provádí pouze nad jedinou instancí, která je jako jediná pro tyto operace určena. Pokud je zápisová operace úspěšná, je její podoba nebo výsledek uložen do souboru zvaného ***binární log.*** Tento soubor si následovně další členové clusteru přenesou k sobě, uloží jej do souboru zvaného ***relay log.***

Poté postupně vykonávají na svých datech operace popsané v tomto ***relay logu.*** Jak již bylo zmíněno, všechny zápisové operace od klienta jsou prováděny nad jedinou master instancí, a proto se nepředává informace o tom, zda slave instance dokázala replikovaná data aplikovat. [7]

3.1.6 Synchronní replikace

Zda se zápisové operace mohou provádět nad více instancemi daného clusteru, závisí na principu synchronních replikací. Jednou z možností je, že se během vykonávání transakce instance zasílá žádost o zamknutí dat, se kterými chce manipulovat, všem instancím clusteru.

Další možností je, že se změny pokusí vykonat a pokud se podaří změny vykonat, dotazuje se ostatních instancí v clusteru, zda změny provedené jsou možné. Pokud jsou změny potvrzeny od ostatních členů jsou přeneseny na ostatní členy v clusteru, anebo jsou všechny změny navráceny a transakce skončí s chybou.

3.2 Replikace a její problémy u relačních databází

Asynchronní replikace mezi instancemi master a slave, a to i mezi clustery postavěnými ve více lokalitách, přináší několik problému a omezení, nejzásadnější z nich popíši v této kapitole.

Hlavní problémy vznikají, když máme architekturu relačních clusterů postavenou napříč dvěma datovými centry. Jedním z těchto problémů je možnost škálování databázového systému, při údržbě master instancí, a to především z pohledu jednoho zápisového replikačního bodu nebo při údržbě clusteru napříč data centry.

Pro provedení údržby je nutné zamykaní pro zápis nebo vypnutí instance, což způsobuje výpadek služby, která využívá tuto instanci pro své fungování. Z těchto důvodů je údržba dopředu plánovaná a je prováděna v době, kdy je systém, který danou instanci využívá, nejmíň vytížen. Problémy ovšem nastávají při neplánovaných výpadcích, ať už z chyby administrátora nebo selhání hardwaru.

Údržba jako taková i případná řešení problému s hardwarem je starost administrátorů. V této práci a kapitole se spíše zaměřím na problémy, které vznikají v datech při různých výpadcích a chybách administrátorů při přepínání zápisu do clusteru mezi mastery napříč data centry.

3.2.1 Vznik nekonzistence nebo ztráty dat

Replikace jako takové nejsou velmi odolné vůči pádům, výpadkům proudu, chybám na síti nebo chybám způsobeným nekonzistencí na disku či paměti. Při používání relačních databází a jejich replikací může vzniknout spousta problémů jako jsou neočekávané vypnutí master nebo slave instance, nekonzistentní Binary log na master nebo slave instanci, použití tabulek, které se zamykají pro zápis („nontransactional tables“), nebo jejich kombinace enginu pro tabulky a spousty dalších zapeklitostí.

Popsat všechny tyto problémy a jejich řešení je nad rámec této práce a jsou spíše administrátorskou záležitostí či záležitostí databázového specialisty. [5]

3.2.1.1 Zápis do obou master instancí při bidirectionální architektuře

Tento scénář je možný, ovšem přináší velká rizika, a proto není dobrým nápadem. Pokud zapisujeme do obou masteru ve stejný čas, vznikají nám problémy. Některé mají řešení jiné ovšem ne.

Představme si situaci, kdy máme postavený cluster ve dvou lokalitách a využíváme asynchronní replikace. Chceme přepnout zápis z jednoho masteru do druhého. Tato situace nastává například z důvodu provádění údržby, kdy je například potřeba jedno data centrum síťově odstavit kvůli výměně routeru. Při tomto přepínání může nastat problém při paralelním zápisu dat do obou masteru v mezičase přepínání provozu napříč data centry.

Další možnost, kdy se nám může stát, že zapisujeme do obou master instancí, když přepneme DNS záznam. Vzniká totiž latence mezi tím, kdy jej klienti, kteří databázový cluster využívají, znovu obnoví. V tento moment se může stát, že zápisové operace jdou do obou master instancí ve stejný čas, vzniknou vlastně dva rozdílné záznamy se stejnými

unikátními hodnotami. To způsobí porušení integritního omezení napříč clusterem a daná replikace nesmí být vykonána. Na chyby vzniklé tímto zápisem můžeme narážet ještě dlouho po jejím vzniku. V tomto případě má některý uživatel systému těžce odhalitelné chyby. Dohledání těchto chyb je velice náročné.

Řešení těchto problémů je ve většině případů velice složitou a náročnou činností. Jedním z jednodušších řešení, pokud chceme zapisovat do dvou master instancí a používáme AUTO_INCREMENT na primární klíče, je použití takzvaného ***auto_increment_offset***. Tedy posuneme si klíče na každém serveru jinak, například na prvním budeme mít pouze liché klíče, na druhém masteru budeme mít pouze sudé klíče. Ovšem toto řešení není univerzální, lze aplikovat pouze na omezenou množinu problémů, a navíc může způsobit další problémy jako je velikost klíče a mrhání čísli. [5]

3.2.2 Nedeterministické dotazy

Dotazy, které mění nedeterministicky[1] data, můžou způsobit problémy při replikacích. Můžou nám způsobit, že data budou jiné na master uzlu a jiné na slave uzlu. Tohle se může stát, pokud používáme stavové replikace (viz podkapitola Replikace).

Nejčastějšími příklady těchto dotazů je UPDATE s LIMIT klauzulí spoléhající na pořadí řádků tabulky vyhledané tímto dotazem. Není-li garantováno stejné pořadí na masteru i slavu.

Dalšími dotazy, které mohou mít toto chování, jsou dotazy REPLACE a INSERT IGNORE na tabulku s více než jedním unikátním indexem, jelikož server může zvolit jiný na masteru a jiný na slavu. Nesmíme zapomenout na funkci NOW, která může také způsobit velké nepříjemnosti v rozdílných hodnotách na master a slave instanci při použití stavových replikací. [5]

Pokud ovšem použijeme řádkově založené replikace (viz podkapitola Replikace), tyto omezení a problémy nevznikají.

3.2.3 Chybějící dočasné tabulky

Dalším problémem, na který můžeme narazit při použití stavových replikací, je použití dočasných tabulek („temporary tables“). Pokud replikace spadne nebo je vypnuta

[1] Nedeterministický znamená, že při stejném vstupu bude vracet jiné výsledky. V kontextu relační databáze jsou to hlavně funkce NOW(), AUTO_INCREMENT, ale samozřejmě další.

všechny dočasné tabulky zmizí. To znamená, že pokud zašleme dotaz na repliku, která odkazuje na dočasnou tabulku, tak nám databázový systém vrátí chybu.

Nezáleží, jestli dočasné tabulky existují pouze krátkou dobu, mohou nám způsobit problémy. Dočasné tabulky se využívají, pokud potřebujeme v rámci jednoho připojení vytvořit tabulku nebo si něco spočítat, ale nechceme, aby tabulka byla viditelná pro ostatní připojení do databáze. Výhodou dočasných tabulek je také to, že se tabulka automaticky po ukončení spojení smaže, tedy odpadá režie s mazáním.

Vyřešit problém s chybějícími dočasnými tabulkami se dá řešit tak, že simulujeme tuto tabulku normální ("skutečnou") tabulkou, která obsahuje potřebná data, a navíc sloupeček s identifikátorem připojení. Tuto tabulku můžeme po ukončení aplikace smazat nebo vyčistit. Toto použití přináší další plus, a to je snadnější ladění aplikace, jelikož dočasná tabulka zaniká při konci připojení, "skutečná" tabulka existuje nadále dokud ji nesmažeme. "Skutečné" tabulky ovšem přináší zpomalení při vytváření souborů, tuto režii dočasné tabulky nemají. [5]

3.3 Druhy architektur

Druh architektury, do které je cluster zapojen, je silně vázán na výběr replikačního mechanismu. V této podkapitole nastíním základní Master-slave architekturu, bidirectional a multi-master architekturu.

3.3.1 Master-slave architektura

Tato architektura je vyobrazena na obrázku (Obrázek 2: Asynchronní replikace, Master - Slave) je to základní pohled na asynchronní replikační mechanismus. Máme jednu instanci, která obstarává všechny zápisové operace takzvaného mastera. Všechny tyto změny jsou přenášeny pomocí replikačního mechanismu na instanci v pozici slave, který je na master instanci napojený.

Jak již bylo řečeno, instance slave nepředává vůbec žádnou informaci o tom, zda se přenos dat podařilo zpracovat, ani v jaké je kondici. Master instance zná pouze počet slavu, které má na sobě navázány.

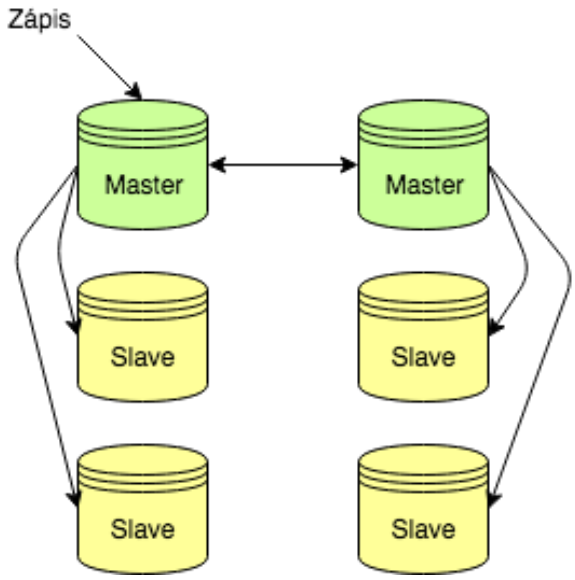
Instance slave slouží převážně jako záloha, kdyby master například hardwarově selhal. Umožňuje ovšem i pomoc s provozem, kdy jsou na něj směřovány například agregační dotazy, které nepotřebují nejaktuálnější data. [2]

3.3.2 Bidirectional architektura

Zapojení v bidirectionalní architektuře znamená, že jsou dvě instance zapojeny do master-slave architektury popsané výše. Zapojení je ale v obou směrech. Takové zapojení znamená, že master je pro druhou instanci slave a replikuje z něj případné změny, a druhá instance je master pro první instanci, tzn. data se replikují z první instance na druhou a z druhé na první. Z tohoto popisu je jasné, že by mohlo docházet ke konfliktům.

Toto zapojení se využívá napříč datovými centry, kdy samozřejmě zapisujeme do jednoho mastra a veškeré zápisové operace se replikují do druhého data centra, ovšem princip funguje oboustranně. V tomto zapojení je možné přepínat provoz, tedy zápisové operace z jednoho mastra na druhého, kde se role vůči sobě vymění.

Obrázek 3 Bidirectional architektura



Zdroj: Vlastní zpracování [2]

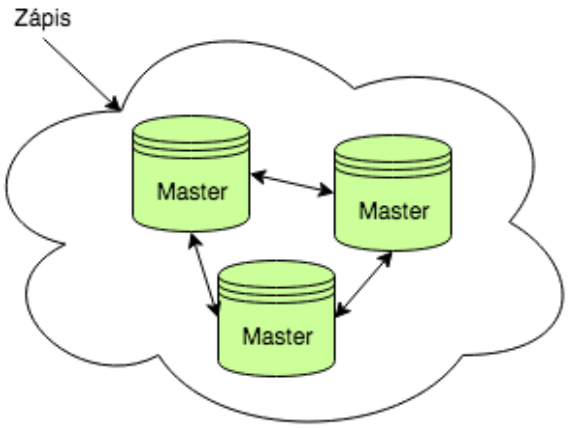
Na schématu (Obrázek 3 Bidirectional architektura) je vidět, že na mastry je možné navázat další slave instance, které mohou sloužit jako záložní instance nebo jsou určeny pro čtení, aby čtecí operace nezatěžovali master. [2]

3.3.3 Multi-master architektura

Multi-master architektura je znázorněna na obrázku (Obrázek 4: Multi-master architektura). Tato metoda je založena na principu synchronních replikacích. Při tomto zapojení jsou všechny instance clusteru mastery. V tomto případě jsou všechny zápisové i

čtecí operace povoleny na všechny instance clusteru. Instance si mezi sebou synchronizují měněná data a domlouvají se, které transakce jsou povoleny, a které je nutné vrátit. [2]

Obrázek 4: Multi-master architektura

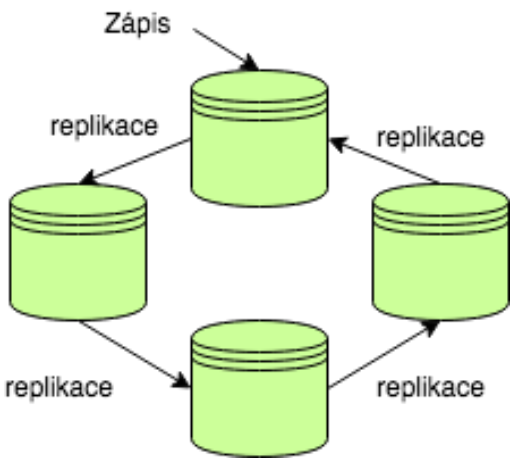


Zdroj: Vlastní zpracování [2]

3.3.4 Kruhová (Ring) architektura

Je to speciální případ multi-master replikací (Obrázek 5: Kruhová architektura). Při kruhové architektuře máme tři a více master instancí. Každý server je vlastně replikou serveru předchozího a master instací následujícího. Bohužel tato architektura nepřináší žádný přínos, je silně závislá na dostupnosti ostatních uzlů. Při odebrání jednoho uzlu je možné dostat nekonečnou smyčku replikačního cyklu (změny na odebraném uzlu budou neustále probíhat). Z těchto důvodů je vhodné vyhnout se této architektuře. [2]

Obrázek 5: Kruhová architektura

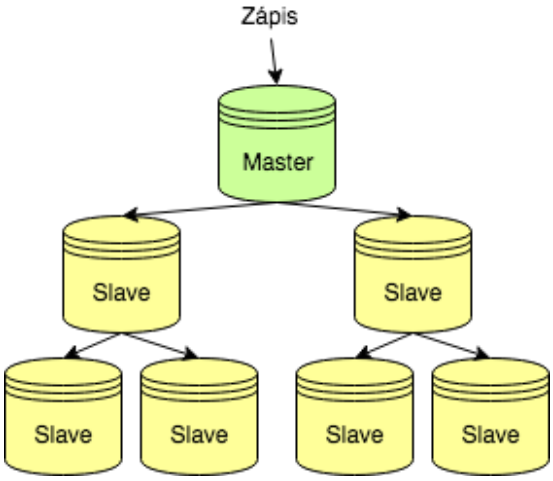


Zdroj: Vlastní zpracování [2]

3.3.5 Stromová nebo pyramidová architektura

Pokud replikujeme master instanci na spousty slavu (Obrázek 6: Stromová architektura), ať už z důvodu rozložení geograficky dat nebo skrze vysokou dostupnost čtení. Můžeme využít architekturu zvanou stromová. Tato architektura přináší odlehčení masteru skrze distribuování dat na repliky. Přináší ovšem také rizika spojená s tím, že je tato architektura hierarchická, to znamená že pokud selže master instance je nutné rozhodnout, která instance převezme otěže a čím víc rozsáhlejší strom máme, tím složitější a komplikovanější je zvládání chyb. [2]

Obrázek 6: Stromová architektura



Zdroj: Vlastní zpracování [2]

3.4 Škálování v relačních databázích

Při systémech, kdy nám data mohou růst velice rychle, je nutné tyto data rychle a efektivně zpracovávat a uchovávat. K tomuto účelu existují různé metodiky jak v relačních databázových systémech, ale nejen v nich, data škálovat.

Data lze škálovat dvěma způsoby prvním z nich je takzvané vertikální škálování, při tomto způsobu škálování je nakoupen více výkonný server/y. Pravým opakem je horizontální škálování, kdy se práce rozloží mezi více serverů.

3.4.1 Vertikální škálování

Pro spoustu aplikací je upgrade stávajícího hardwaru za výkonnější velice vhodným řešením problémů. Tato strategie má spousty výhod. Například jeden server se snáze

spravuje a vyvíjí oproti více serverům. Zálohování a obnova dat pro aplikaci je mnohem jednodušší, jelikož máme jeden data set a neřešíme proto ne-konzistence napříč clusterem.

Ovšem řešit tímto způsobem škálování relační databázové systémy nejde do nekonečna. Pokud aplikační data bobtnají, je dost možné, že narazíme na limity, které již nedovolí škálovat dále. Tyto limity se mohou týkat například i velké finanční náročnosti. Je také velice důležité zamyslet se, zda udržíme slave uzel v podobném hardwarovém stavu, kdyby nám master náhodou selhal. V neposlední řadě nesmíme zapomenout na to, že i nejvýkonnější počítače mají své limity.

3.4.2 Horizontální škálování

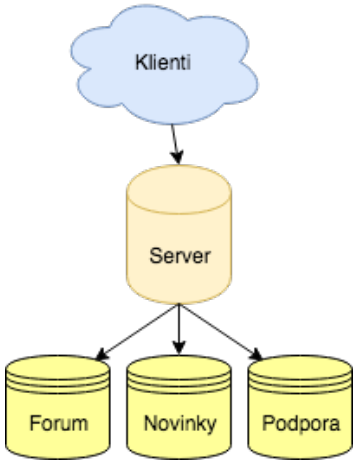
Pokud se budeme bavit o horizontálním škálování, existují zde tři hlavní skupiny jak data dělit: replikace, rozdělení dat („partitioning“), škálování („sharding“).

Nejjednodušším, a také nejčastějším řešením, jak distribuovat data skrze více serverů, je použití replikací. Replikace se používají pro čtecí operace. Tato možnost je velice vhodnou pro systémy, které jsou velice zatěžovány čtecími operacemi. Přináší ovšem i pár nevýhod popsaných v podkapitole Vznik nekonzistence nebo ztráty dat.

Další velice časté řešení jak škálovat je rozdělení dat („partitioning“). To znamená, že provoz a dotazy jsou rozprostřeny skrze celý cluster. Toto řešení je velice podobné škálování dat („shardingu“). [5]

3.4.2.1 Rozdělení dat („Partitioning“)

Obrázek 7: „Partitioning“ - Rozdělení dat



Zdroj: Vlastní zpracování [5]

Rozdělení dat je někdy také nazýváno jako rozdělení povinností, znamená že dedikujeme různé uzly na různé úkoly. Pro použití této strategie je velice důležité rozmyslet, jak budou data dělena, jelikož každá část clusteru bude obstarávat svou část. Z těchto důvodů je velice nutné připravit aplikaci na práci a manipulaci s daty jiným způsobem, než je nejčastější použití.

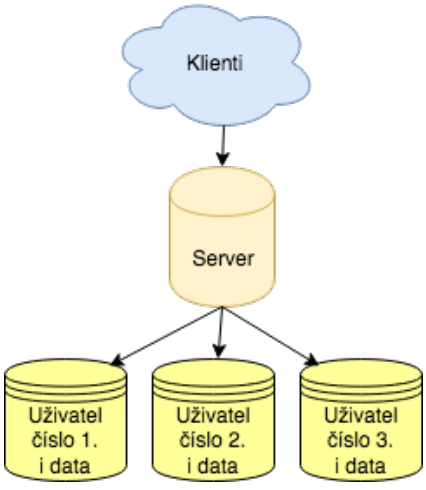
Jedním z příkladů, kdy se používá rozdělení dat je například webová stránka obsahující několik sekcí, které nemají nic společného, například stránka fórum, novinky a podpůrná stránka. Díky tomu můžeme mít data logicky i fyzicky oddělené na jiných serverech.

Další možné využití je rozdělení jedněch aplikačních dat na sety tabulek, o kterých víme, že se nikdy nebudou spojovat dohromady nebo se budou spojovat velice výjimečně. Nevadilo by, pokud bychom ztráceli výkon, jelikož bychom museli tyto data museli spojovat, až na úrovni aplikace. Toto využití není moc běžné, jelikož je velice náročné dělit efektivně data a nepřináší nám zas tak moc výhod.

Ovšem pokud se zamyslíme, dělení dat také nejde do nekonečna. Jelikož každý sub set aplikačních dat musí být na jednom databázovém serveru. Může se i stát, že jeden sub set nám přeroste a budeme muset zvolit jinou strategii. [5]

3.4.2.2 Škálování dat („Sharding“)

Obrázek 8: Škálování dat



Zdroj: Vlastní zpracování[5]

Škálování dat je v dnešní době jedním z nejúspěšnějších řešení zefektivnění obrovských aplikaci. Aplikační data jsou rozložena na malé kousky, které jsou uloženy napříč celým clusterem.

Škálování dat („sharding“) jako takové velice dobře funguje i s kombinací s rozdělením dat („partitioning“). Většina systémů má tzv. globální data, která nejsou škálovaná vůbec. Tyto globální data jsou obvykle uložena v single node instanci, často v cachi jako je **memcached[2].**

Ve skutečnosti většina aplikačních „shardů“ obsahuje data, která byly škálovaná, tedy typicky data sety, které jsou obrovské a rostou velice rychle. Představte si stránku, kterou navštěvuje denně miliony uživatelů a přispívají do diskuzí, nebo pouze příspěvky.

Velké aplikace mohou mít několik logických data setů, které se mohou škálovat různě. Je možné je uložit na různé stroje nebo je možné mít stejná data několikrát. Záleží, jakým způsobem přistupujeme k datům a jak s nimi potřebujeme pracovat.

Návrh databáze a aplikace tak, aby podporoval horizontální škálování, je velice odlišný od běžných návrhů. Pokud bychom uvažovali o horizontálním škálování existujícího monolitického datové uložiště, může to být velice složité, až téměř nemožné. Proto je mnohem snazší stavět aplikaci se škálovanými daty od samého začátku, pokud máme i jen malé podezření, že bychom je mohli potřebovat.

Horizontální škálování dat může mít několik přístupu, například už nám nestíhá optimalizace řešená replikacemi, kdy jsme měli několik strojů pouze pro čtení. Můžeme rozdělit data na logické celky jako jsou uživatelé, příspěvky a komentáře, a ty uložit na různé servery.

V neposlední řade je možné dělit data podle návaznosti. Například víme, že příspěvky a komentáře jsou navázány na uživatele, tak škálujeme příspěvky i komentáře po celém clusteru a uživatele budeme mít jako globální data.

Horizontálně škálované aplikace často mají knihovnu, která obsahuje abstrakci databáze a zjednodušuje komunikaci mezi aplikaci a rozdělenými daty v clusteru. Takové knihovny obvykle úplně neskrývají to, že jsou data škálovaná, protože aplikace obvykle ví něco o dotazech, které databázový server neví. Příliš mnoho abstrakce může vést k velké

[2] Memcached jak již název vypovídá je distribuovaný cachovací nástroj, který je open-source. Ukládá si data do „hash“ tabulek distribuovaných napříč clusterem. Data jsou uložena pouze v RAM paměti.

neefektivitě, může se například stát, že dotaz bude rozeslán na všechny uzly clusteru, i když data existují pouze na jednom.

Škálování nebo rozdělení dat i přes nevýhody složitosti návrhu je tedy vhodné použít v případě kdy chceme zefektivnit zápis, jelikož je nemožné mít efektivní zápis, pokud máme jeden master uzel, to i s libovolným počtem replik. [5]

3.5 Synchronní replikace a druhy jejich implementací

V této kapitole bude popsáno fungování synchronní replikace, a to převážně z pohledu globálního zpracování transakce napříč clusterem založeným na databázovém stavovém automatu. Dále to budou druhy synchronních replikací a budou zde představeny způsoby jejich využití velkými společnostmi.

Různá řešení i implementace synchronních replikací existují již řadu let. V praxi byly tradičně implementovány tzv. dvoufázovým potvrzováním nebo distribuovaným zamykáním tabulek nebo řádků. Tato řešení se však ukázala jako pomalé a složité. Díky nízkému výkonu a složité implementaci synchronních replikací se staly asynchronní replikace dominantními pro stavění databázových replikačních clusterů i navzdory nedostatkům, viz kapitola Problémy nastávající při vícenásobném přístupu.

Na přelomu milénia byl představen zcela nový alternativní přístup k paralelnímu zpracování transakcí napříč clusterem. Tento přístup je takový, že se využívá certifikačního testování, skupinové komunikace a přístupu ke clusteru jako ke stavovému automatu.

Nový přístup nám umožňuje stavět databázové clustery do multi-master architektury replikované pomocí synchronních replikací, a díky tomu výrazně eliminovat problémy popsané v kapitole Zápis do obou master instancí při bidirectionální architektuře. Princip synchronních replikací založených na databázovém automatu definuje mnoho vlastností a řešení. [8]

3.5.1 Databázový stavový automat

Z uživatelského pohledu databázový stavový automat umožňuje vysokou dostupnost a dobrý výkon. Nabízí také silnou konzistenci. Ze systémového hlediska je to mechanismus, který obstarává replikace v clusteru pomocí standardní komunikace skrze síť.

Komunikace probíhá skrze takzvané atomické vysílání („Atomic broadcast“ viz Komunikace). Při porovnání s přístupy ostatních relačních databází, které splňují dostupnost, databázový stavový automat neobětuje výkon (minimalizuje interní synchronizaci a eliminuje deadlocky[3]). [8]

- Databázový cluster zpracovává transakce na základě stavů. Automat přechází do stavů pro každou zpracovávanou transakci.
- Je definován způsob, jakým instance clusteru mezi sebou komunikují a předávají si replikovaná data. V komunikaci je potřebné rozhodnutí, zda jednotlivé zápisové transakce bude napříč clusterem potvrzená nebo zamítnutá je zde využito certifikační testování.
- Pro každou zápisovou transakci je potřebné potvrzení takzvaná certifikace. Ta nám určí, zda na každé instanci v clusteru je zápisová transakce zpracovatelná či nikoliv.
- Transakce i jejich certifikační testování jsou napříč clusterem seřazené. Díky tomuto přístupu dosahují výborného výkonu. Transakce jsou podle principů serializace řazené tak, aby replikované transakce byly zamítány co nejméně.
- Potvrzení transakcí (certifikace) se provádí v první řadě na základě toho, zda měněné řádky již mají být změněny jinou transakcí. Tento způsob umožňuje opožděné aplikování již potvrzených replik a zvyšuje výkonnost databázového clusteru. Hlavně proto, že cluster nemusí čekat, než instance daná data zapíší, aby byla detekována kolize s jinou transakcí.
- Při synchronních replikací založených na stavovém automatu, skupinové komunikaci a certifikačním testování vyhrává transakce, která získá první potvrzení.

3.5.1.1 Princip opožděného zpracování dat

Tento princip popisuje techniku, kdy se transakce provádí lokálně na jedné straně databáze. Během aplikace změn zde není žádná interakce s ostatními instancemi. Transakce jsou lokálně synchronizovány na straně databáze podle pravidel konkurence.

[3] Deadlock – Situace kdy dokončení první akce je podmíněno dokončením akce předchozí, ale akce předchozí závisí na akci první.

Další předpoklad je, že kontrola konkurence využívá na všech uzlech databázového clusteru k lokální synchronizaci transakcí dvou-fázové zamykací pravidlo.

Když klient zašle transakci, tak update operace, co transakce vykoná, a další kontrolní struktury jsou propagovány na všechny uzly, kde je transakce certifikována a popřípadě provedena. Transakce a certifikace jsou propagovány na databázové uzly. Opoždění zpracování dat může být implementováno jako stavový automat. [8]

3.5.1.2 Stavy transakcí

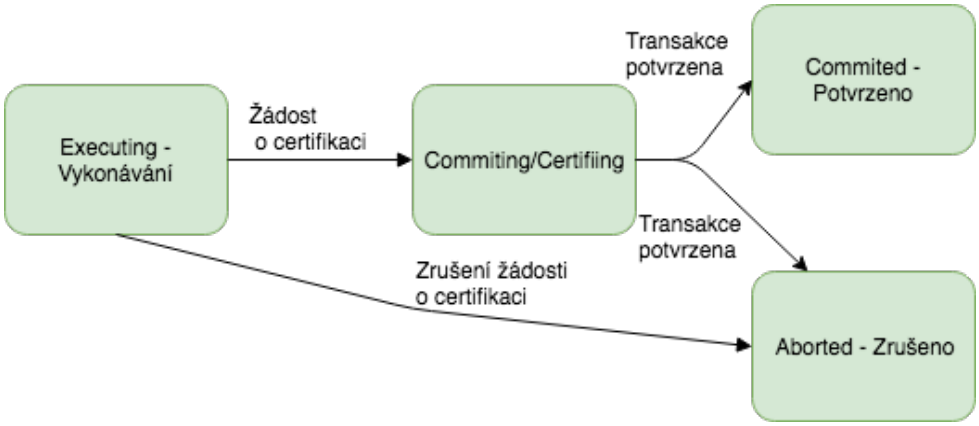
Zpracování transakce přechází z několika definovaných stavů (Obrázek 9: Stavový automat transakce). Transakce začíná pro všechny čtecí i zápisové transakce ve stavu Vykonávání („Executing“).

Jakmile dojde k potvrzení operace na jedné instanci jde vlastně o izolační mechanismus založený na dvou-fázovém zamykacím pravidle, jak bylo popsáno při opoždění zpracování dat (viz Princip opožděného zpracování dat). Jakmile je iniciováno potvrzení transakce instance je možné přejít do dvou stavů ze stavu Vykonávání.

Pokud transakce selže již na první instanci přesunuje se transakce do stavu Zrušeno („Aborted“). Pokud ovšem je vše v pořádku změní se stav transakce na Potvrzeno („Commiting“), neboli potvrzování v systémech postavených na certifikačním testováním. Může být tento stav také označován jako Certifikace („Certification“).

O tom, co se stane s transakcí rozhoduje databázový serveru, buď přejde do stavu Potvrzeno, je tedy konzistentní a transakce jsou přeneseny do tzv. Zamykacího manažeru („Lock Manageru“) nebo do stavu Zrušeno. [8]

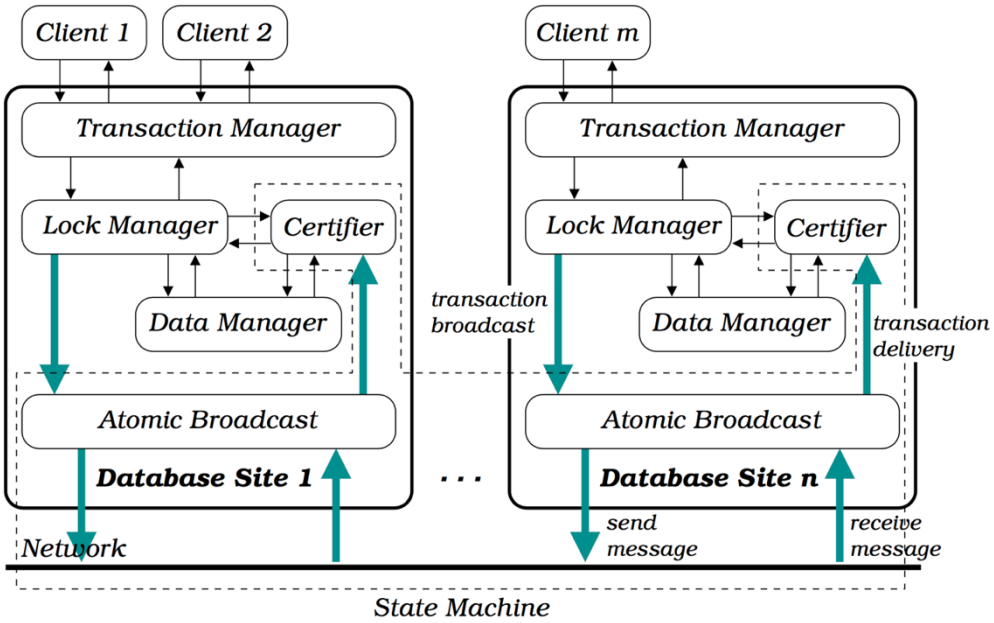
Obrázek 9: Stavový automat transakce



Zdroj: Vlastní zpracování[8]

3.5.1.3 Zpracování transakce

Obrázek 10: Podrobnější schéma zpracování transakce



Zdroj: [8]

Na schématu Obrázek 10: Podrobnější schéma zpracování transakce, můžeme vidět podrobnější postup zpracování transakci. Pokud se transakce dostanou do stavu Potvrzeno, jsou změny zaslány na všechny instance clusteru. Každá instance clusteru zpracovává transakci podle předem daných pravidel. Transakce jsou zpracovány pomocí Správce Transakcí („Transaction Managera“), Zamykacího manažeru („Lock Manageru“) a Data Manažeru.

Certifikační manažer („Certifier“) vykonává certifikační test pro všechny příchozí transakce. Tyto transakce jsou zaslány pomocí Atomického Vysílacího („Atomic Broadcast“) modulu. Všechny úspěšně certifikované transakce jsou přesunuty do Zamykacího manažeru, který garantuje, že změny budou provedeny v budoucnu. [8]

3.5.1.4 Seřazení transakcí

Jak již bylo zmíněno, transakce jsou setříděny při jejich certifikaci. Toto třídění je prováděno ve stavu Seřadit („Reorder“). Certifikační test tedy probíhá odlišně oproti automatu, viz podkapitola Databázový stavový automat. Certifikovaná transakce je porovnávaná oproti transakcím, které byly už aplikovány a potvrzenými transakcemi

v Seřazeném seznam transakcí („Reorder List“). Ten uchovává potvrzené transakce, které mají garantované, že budou aplikovány, tedy nebyly ještě provedeny za běhu aktuální transakce. Počet transakcí v Seřazeném Seznamu transakcí je ovšem omezený. Jakmile je dosáhnut limit nejstarší transakce je aplikována. [8]

3.5.1.5 Komunikace

Pro komunikaci v databázovém clusteru je nutné využívat zpráv. Máme dva základní typy přenosů zpráv: [8]

1. **Spolehlivý přenos** - Negarantuje pořadí zpráv, tudíž může zapříčinit nekonzistenci dat.
2. **Atomický přenos** - Garantuje přesně takové pořadí zpráv v jakém byly odeslány, ale přináší větší zátěž na přepočty a kontroly.
3. **Optimisticky atomický přenos** - Je využíván databázovým stavovým automatem, umožňuje nastavovat různé stupně kontroly pravidel při přenosu. Je zde nutné zamyslet se jakou úroveň pro jaký účel použijeme. Pokud nám jde více o konzistenci, zpřísníme pravidla, ale pokud jde o rychlost, zvolníme.

3.5.2 Typy synchronních replikací

Existuje více druhů synchronních replikací lišících se podle toho, jakým způsobem se transakce potvrdí a popřípadě zapisují do instancí v clusteru.

3.5.2.1 Plně synchronní

Tyto replikace čekají na potvrzení, že jsou všechna aplikovaná data v pořádku od všech instancí v clusteru. Jak již vyplývá z této logiky tento přístup je značně pomalý, velice závislý na počtu instancí v clusteru a zatížení sítě v clusteru a hardwaru pro databázové uzly, je totiž nutné čekat, než se všechny změny promítnou na všech uzlech v clusteru. [8]

Toto chování je založeno na takzvaném dvojitém potvrzení transakce, které se skládá, jak již název vypovídá, ze dvou fází. V prvním kroku se instance, která vykonává operace, které mění data, ptá ostatních členů v clusteru, zda tyto transakce nejsou v kolizi s jinou právě vykonávanou transakcí. Členové clusteru potvrdí, a tedy přijmou změny, pokud není v kolizi s jinou transakcí a zamknou dané řádky, popřípadě tabulky napříč clusterem. [9]

Druhý krok spočívá v tom, že se transakce dokončí a vyšle se ostatním členům v clusteru žádost o druhé potvrzení, že byly změny aplikovány.

3.5.2.2 Virtuálně synchronní

Popis fungování těchto replikací je popsán v kapitole Databázový stavový automat. Je zde stavový automat, který přechází ze stavů a používá se zde certifikace, které zaručují, že replikační změny budou někdy konzistentní, tedy se promítnou do dat v rámci celého clusteru, ovšem není definováno kdy. Během doby po potvrzení transakce až po její zápis do databáze se s daty nepracuje, pouze se kontrolují nově příchozí transakce, zda neobsahují konfliktní změny. Pokud jsou v pořádku jsou zařazeny do fronty čekajících transakcí. [8]

3.5.2.3 Polo synchronní („Semi-synchronní“)

Pro polo synchronní replikace je nutné definovat počet instancí v clusteru, které musí data potvrdit, aby byla data úspěšně potvrzena. To znamená menší režie, ovšem mohou nám zde vzniknout konflikty. Stále se jedná o master-slave architekturu, kdy master čeká na potvrzení z daného počtu instancí.

Díky tomuto přístupu je zvýšená perzistence dat (existence změn je na více než jednom místě). Tento přístup je označen jako synchronní replikace například v PostgreSQL. [10]

3.6 Implementace synchronních replikací

V současnosti existují dvě hlavní implementace synchronních replikací založených na distribuovaném stavovém automatu.

Prvním řešení je Galera cluster, která byla vytvořena společností Codership v roce 2007. Galera cluster je kompletní nadstavba nad serverem MySQL v podobě modulu. Tento modul využila a vydala společnost Percona a MariaDB.

Druhé řešení je od společnosti Oracle zveřejněné na konci roku 2016. Oracle implementoval svůj vlastní přístup k multi-master clusteru, který využívá tradičních binárních logu pro přenos replikací pojmenovaný jako Group replikace.

V této kapitole bude zmíněna i další technologie od firmy Oracle, a to Oracle RAC, která umožňuje vytvořit z více databázových serverů jeden cluster. Tato technologie

využívá sdílení zdrojů, neimplementuje tedy certifikaci, jako je tomu u Galery či Group replikací.

3.6.1 Galera cluster

Je to multi-master řešení, které je prověřené již delší dobu. Poskytuje kompletní nástavbu nad replikacemi v MySQL. Každá spuštěná transakce v clusteru získá jedinečné číslo zvané „Global Transaction Identifikator“ (GTID[4]). MySql verze 5.6 již obsahovala takový identifikátor, ovšem Galera definuje svůj vlastní. Tento identifikátor slouží k synchronizaci napříč clusterem a v případě výpadku umožňuje toto GTID obnovu.

Galera požaduje, aby replikace probíhaly striktně v řádkově orientovaném formátu kromě DDL[5] příkazů (viz podkapitola Replikace). Vyžaduje dále InnoDB engine pro uložení dat. [6]

3.6.1.1 Stavy instancí

Stav instance se odvíjí podle toho, s jakými daty právě pracuje či disponuje. [6]

Otevřený („Open“) - Instance se spustí a pokusí se navázat komunikaci s plně funkční instancí v clusteru.

Primární („Primary“) - Pokud uspěje s žádostí o přenos dat, začne ukládat tzv. „*Write-sety“* probíhajících transakcí.

Připojení („Joiner“) - Instance získala celý obraz dat pomocí SST od dárce a může začít aplikovat uložené „Write Sety“ transakcí, které proběhly během získávání dat.

Připojený („Joined“) - Je ukončeno aplikování „Write-setů“ a instance získává stav ve kterém je zbylý cluster. Jeho mezi paměť je prázdná.

Synchronizovaný („Synced“) - V tomto stavu je instance, pokud se stane plnohodnotnou instancí clusteru. Umožňuje tedy zpracovávat transakce.

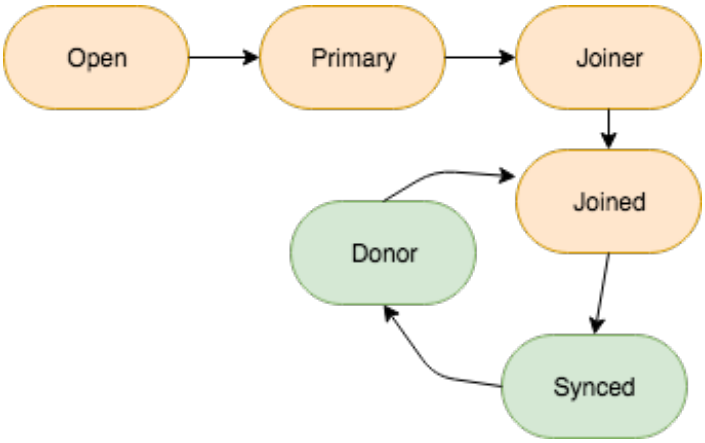
Dárce („Donor“) - Pokud je tato instance v tomto stavu, obdržela žádost o poskytnutí svého obrazu dat jiné instanci. Stává se zní tedy dárce.

---

[4] GTID – Skládá se z, Stavového UUID (unikátní identifikátor pro stav a sekvence změn, kterými prochází) a pořadové číslo (seqno, 64-bitové číslo sloužící k označení v sekvenci).

[5] DDL - Data Defining Language, jsou to příkazy, které mění strukturu dat (CREATE, ALTERm TRUNCATE, COMMENT, RENAME). [6]

Obrázek 11: Stavy instancí v Galera clusteru



Zdroj: Vlastní zpracování [6]

3.6.1.2 Replikace založené na certifikaci („Certification-based replication“)

Metoda replikací založená na certifikaci využívá skupinovou komunikaci („Group communication“) a techniku řazení transakcí k dosažení synchronních replikací.

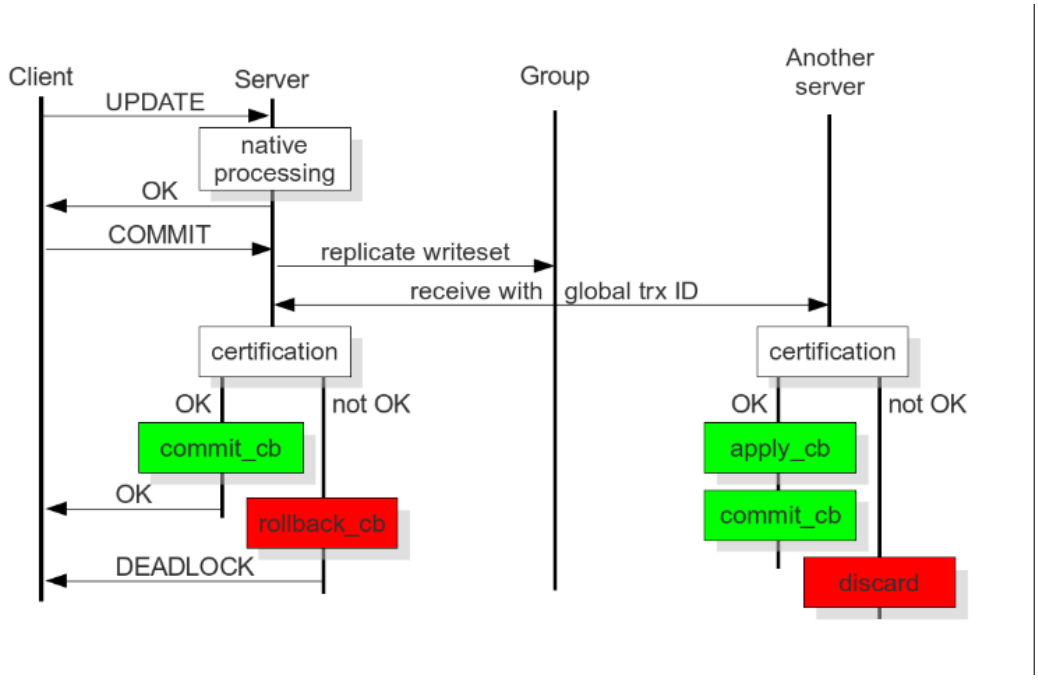
Transakce jsou vykonány na jednom uzlu, poté v čase potvrzení („commit“) běží certifikační proces, který zajistí globální konzistenci v clusteru. Této konzistence je dosaženo pomocí vysílání „*write-setů*“ k potvrzení. Je zajištěno správné pořadí i v rámci konkurenčních transakcí.

Není možné zajistit replikace založené na certifikaci pro všechny databázové systémy. Abychom mohli využít této funkcionality je nutné, aby databázový systém podporoval tyto podmínky:

- **Transakční databáze** – Je nutné, aby databázový systém uměl pracovat v transakcích, tedy uměl operaci „rollback“ – navrácení do předchozího stavu. [6]
- **Atomické změny** – Další podmínkou jsou atomické změny. Je tedy nutní zajistit, že série operací se provede jako celek nebo vůbec. [6]
- **Globální seřazení** – Replikace jsou seřazeny a aplikovány v daném pořadí na všechny instance clusteru. [6]

Hlavní myšlenkou replikací založených na certifikaci je provádění transakcí, dokud není dosaženo potvrzovacího bodu za předpokladu, že není konflikt. Tomu se říká optimistické vykonání. [6]

Obrázek 12: Certification-based replication



Zdroj: [6]

Všechny primární klíče změněných řádků jsou po zaslání potvrzovacího příkazu sloučeny do „*write-setu*“. Ještě před tím, než je opravdové databázové potvrzení provedeno. Databázový server poté rozešle tyto „*write-sety*“ na všechny uzly. Na všech uzlech jsou primární klíče z „*write-setu*“ porovnány, pokud mohou být aplikovány. Pokud certifikační test selže, uzel zahodí „*write-set*“ a cluster zašle navrácení změn („rollback“) do původního stavu. Pokud test uspěje transakce jsou potvrzeny a „*write-sety*“ se aplikují na zbytek clusteru.

Galera cluster implementuje toto řešení. Využívá globálního seřazení transakcí. V Galera clusteru je každé transakci přiřazeno globálně unikátní sekvenční číslo během replikace.

Pokud transakce dosáhne stavu, kdy má být potvrzena, uzel zkontroluje sekvenční číslo oproti poslední úspěšné transakci. Jsou zkontrolovány všechny „*write-sety*“, které jsou v ***GCachy*** a pokud zde není konflikt primárních klíčů mezi již zařazenými „*write-sety*“, zařadí „*write-set*“ do fronty a zašle potvrzení. Pokud certifikace selže, je zaslán příkaz na navrácení do předchozího stavu. Proces je deterministický a všechny uzly přijímají transakce ve stejném pořadí. To znamená, že všechny uzly dosáhnou stejného

výsledků příchozí transakce. Uzel, který začal transakci, zašle klientovi upozornění, pokud transakce byla nebo nebyla provedena.

3.6.1.3 Způsoby přenosu dat napříč clusterem

V předchozí podkapitole byly zmiňovány „*write-sety*“, pro replikaci dat uvnitř clusteru. Pokud transakce vykonává některou ze zápisových či update operací, jsou změněné řádky rozdistribuovány a certifikovány napříč clusterem právě pomocí „*write-setů*“, jak již bylo popsáno v kapitole Synchronní replikace a druhy jejich implementací. „*Write-set*“ si můžeme představit jako buffer, neboli vyrovnávací paměť pro změněné řádky.

Zmíněné SST je zkratka pro Snapshot State Transfare, tedy Přenos Obrazu Stavu. Je to vlastně jeden z druhů řešení výpadku, kdy se přenáší celý obraz stavu databázové instance. Druhým způsobem je inkrementální přenos „*write-setů*“ tzv. IST tedy Incremental State Transfer, což je Inkrementální Přenos Stavů. [6]

Všechny potvrzené „*write-sety*“ se ukládají do speciálního souboru tzv. GCache (viz Galera Cache), která slouží k následnému do replikování dat pomocí IST.
Pokud se jedna instance odmlčí, je odstraněna z clusteru. Cluster pak ustanoví kvórum[6] a běží dále bez výpadku. Jakmile se odmlčená instance vrátí do clusteru, vybere si jednu instanci, kterou využije jako svého dárce. Pokud dárce má ve své GCache „*write-sety*“, které odmlčená transakce nemá aplikované, spustí se IST.

Ovšem pokud v GCache tyto „*write-sety*“ nemá, tzn. instance byla odmlčena mimo komunikaci v clusteru delší dobu. Nebo je úplně nová a je spuštěno SST, které udělá pomocí zvolené metody kompletní zálohu. Přenese tuto zálohu k odmlčené či nové instanci a do replikuje data. Tímto způsobem vytvoří plnohodnotnou instanci.

SST je plně automatický, což je oproti asynchronním a i jiným multi-master řešením obrovský přínos. Pro vytvoření obrazu existuje více způsobů. Některé způsoby

[6] Ustanovení kvóra se využívá v distribuovaných systémech. Kde se volebním systémem anebo systémem vah ustanoví většina, která dále bude dostupná. Zbylá část clusteru, i kdyby byla dostupná, se kvůli nedostatku hlasů anebo velikosti součtu vah uzamkne. (3)

jsou blokující, například ***rsync***[7] nebo ***mysqldump***[8]. Blokující přenos odstaví aplikování „*write-setů“* pro danou instanci dárce. Jiné způsoby jsou neblokující např. ***xtrabackup***[9]. Instance, která je dárce, dál zaznamenává a aplikuje „*write-sety“*. [6]

3.6.1.4 Kontrola toku („Flow control“)

Tento mechanismus slouží k zajištění konzistence dat napříč clusterem. Všechny instance clusteru přijímají „*write-sety“*, které certifikují a postupně aplikují do své databáze. V případě, že některá instance nestíhá a její fronta s „*write-sety“* dosáhne předem určeného maxima velikosti, je spuštěna kontrola toku. Kontrola toku zajistí, že ostatní instance clusteru pozastaví replikace a počkají na to, dokud se neuvolní předem určená část fronty. Galera implementuje několik druhů kontroly toků. [6]

3.6.1.5 Ochrana při rozpojení sítě napříč clusterem

Pokud máme cluster rozdělený do dvou lokalit a dojde k síťovému rozpojení clusteru, může nastat nekonzistence dat způsobená zápisem do obou lokalit v jednom čase. Galera má mechanismus, který zajišťuje, aby k tomu nedocházelo. Zavádí ustanovení fungování na principu kvóra.

Cluster ví z kolika instancí je složen. Pokud nastane situace, že některé instance jsou nedostupné, spustí se kontrola, aby zjistil kolik instancí odpoví.

Pokud většina instancí odpoví, skupina ustanoví kvórum a otevře se pro zápis a čtení. Jestli skupina zjistí, že je v menšině oproti stavu předtím, zamkne se pro čtení i zápis. Díky tomuto mechanismu se zabrání nekonzistentnímu stavu, tzv. ***splitbrainu*** zmíněním v kapitole Zápis do obou master instancí při bidirectionální architektuře.

Instancím clusteru lze nastavovat váhy, jež ovlivní hlasování. V případě dvou lokalit, ve kterých je stejný počet instancí a dojde k rozpojení napříč lokalitami, obě lokality se zamknou pro čtení i zápis, jelikož již ani jedna lokalita nezíská většinu.

---

[7] Rsync – Velmi rychlí kopírovací/synchronizační nástroj pro soubory. Umí minimalizovat přenosy dat, tím že přenáší pouze rozdílové informace (delta). (5)

[8] Mysqldump slouží k zálohování MySql databáze, kopíruje obraz databáze v SQL jazyku. (9)

[9] Xtrabackup – je to specializovaný nástroj od Percony, který umožňuje kopírování záloh bez blokování databáze. (6)

V případě, že bychom jedné lokalitě navýšili hodnotu vah, při rozpojení by tato lokalita zůstala otevřená zápisu i čtení a druhá se zamkne.

Způsob navýšení vah je dobrý, pokud jsou výpadky plánované. Pro případ náhlých nebo neplánovaných výpadků existuje lepší řešení, a to ustanovení kvóra minimálně ve třech lokalitách nebo využít tzv. arbitra. Díky tomuto systému je pravděpodobnost výpadku ve dvou lokalitách mnohem nižší.

Arbitr určí, která lokalita je dostupná zvenku. Určí vybrané lokalitě, že se má otevřít pro čtení a zápis. Využití této vlastnosti může být i v případě kdy jedna lokalita je odpojena od internetu, například výpadkem páteřní sítě. [6]

3.6.1.6 Kauzální čtení

Na jedné instanci jsou data zapsána jednou transakcí. Další transakce může tyto změny číst bezprostředně po potvrzení předchozí transakce. Právě u virtuálně synchronních clusterů může nastat situace, že se potvrzená replikovaná data ještě nepromítla do databáze instance. V případě, že je zapnuté kauzální čtení transakce vyčkávají, dokud se potřebné změny neprojeví do databáze, a až poté může s daty pracovat. [6]

3.6.1.7 SWOT analýza

SWOT matice nám přehledně znázorní silné a slabé stránky, hrozby a příležitosti Galera clusteru.

Silné stránky

- Zamezení nekonzistence dat
- Kontrola toku („Flow Control“)
- Kauzální čtení
- Certifikované příspěvky („commity“)
- Vyvíjena a používaná více jak 10 let
- Skupinová komunikace pro potvrzení transakcí

Slabé stránky

- Může docházet k „DeadLockům“ a to při zpracování více dotazů nad stejnými daty
- Složitější změny tabulek (ALTER)

Příležitosti

- Automatické začleňování nových členů do clusteru
- Automatické zotavení z chyb
- Více způsobu, jak replikovat data (Rsync, Xtrabackup, Mysqldump)

Hrozby

- Nepatrné zpomalení oproti asynchronním replikacím.
- Skrze funkce kontroly toku musí mít servery stejnou hardware konfiguraci.

3.6.2 Skupinové replikace („Group replication“)

Společnost Oracle vydala toto nové řešení na konci roku 2016. Oracle vytvořil přídavný modul pro MySQL, který je možné použít. Ve výchozí konfiguraci je zabalený a distribuovaný v serverových balíčcích. Skupinové replikace jsou dostupné a podporované na všech platformách od Windows přes Linux, až po Mac OS a FreeBSD.
Tyto replikace doporučují, aby se cluster používal co nejvíce, nejlépe neustále, jako single-master, tedy cluster, do kterého se zapisuje pouze do jedné instance. Výchozí stav se jako single-master určí sám pro zápis. Skupinové replikace vyžadují, aby transakce, až na DDL, probíhaly striktně v řádkově založeném („row base“) formátu. Dále je vyžadováno uložení dat pomocí enginu InnoDB. [10]

3.6.2.1 Stavy instancí

Instance mohou nabývat různých stavů na základě toho, zda jsou plnohodnotnými členy clusteru, anebo nastala některá z možných chyb. [10]
Online – Stav, kdy je instance plnohodnotným členem clusteru. Může tedy vykonávat transakce.
Zotavování („Recovering“) – Instance přijímá data od ostatních instancí, aby se z ní stal plnohodnotný člen clusteru.
Offline – Instance je načtena korektně, ale již není součástí clusteru.
Chyba („Error“) – Pokud nastane chyba při zpracování transakce, anebo při aplikaci replikací, je instance přepnuta do tohoto stavu.
Nedostupná („Unreachable“) – V tomto stavu je instance oddělená od zbytku clusteru.

3.6.2.2 Způsob přenosu dat napříč clusterem

Skupinové („Group“) replikace využívají pro přenos změny provedených transakcí již zavedenou infrastrukturu MySQL. Pro přenos a uložení výsledků transakcí se využívají binární logy a transakce jsou napříč clusterem číslovány pomocí GTID („Global Transaction Identifikator“).

Když nastane stav, kdy se jedna instance na určitý čas odmlčí, je vyřazena z clusteru, tzn. ztratí některé replikace. Po opětovném připojení vybere od svého dárce, kterého si vybere sama. Pomocí dárce naváže replikace podle jeho binárních logů a svého GTID. V tomto případě se skupinové replikace chovají ve své podstatě stejné jako IST u Galery.

Jakmile žádný z dárců nevlastní tak stará binární data, neexistuje tu nic jako SST u Galery. Je nutné, aby tuto situaci vyřešil administrátor, který ručně vytvoří plnou zálohu některého z aktivních instancí. Vytvoří plnou zálohu některé z aktivních instancí. Vytvoří nad ní nový server a následně jej připojí do clusteru, kde instance naváže na GTID a do replikuje se. [10]

3.6.2.3 Ochrana při rozpojení sítě napříč clusterem

Pokud dojde k síťovému rozpojení napříč clusterem, mají skupinové replikace ošetření na základě ustanovení kvóra podle počtu členů. Administrátor musí určit, která z částí se má povolit pro čtení a zápis. V nynějším stavu nemá vytvořený arbitr a nelze nastavovat různé váhy jednotlivým instancím. [10]

3.6.2.4 Kontrola toku („Flow control“)

Transakce je potvrzená, pokud ji potvrdí a akceptuje majoritní skupina a domluví se na pořadí přijatých transakcí. Ovšem v situaci, kdy některý z členů zaostává za ostatními, spustí se kontrola toku, která zpomalí ostatní členy, dokud si pomalý člen do replikuje data.

Skupinové replikace poskytují dva způsoby, jak předcházet zpomalování clusteru.

Omezení toku replikací – Podobný přístup jako u Galery. Pokud je dosažena určitá míra naplnění fronty neaplikovaných transakcí, ostatní členové pozastaví posílání replikovaných dat a odbavování transakcí, dokud tento člen určitou část fronty neaplikuje své databáze.

Sondování a vytváření statistik – Členové si předávají informace o velikosti fronty, potvrzených transakcí a jiných ukazatelů. Na základě předávaných informací ostatní členové s určitou rychlostí odbavují transakce, aby se nestalo to, že některý z členů výrazně zaostane za ostatními. Informace si mezi sebou instance clusteru předávají jednou za sekundu. [10]

3.6.2.5 SWOT analýza

SWOT matice nám přehledně znázorní silné a slabé stránky, hrozby a příležitosti Group Replication.

Silné stránky

- Certifikované příspěvky („commity“)
- Skupinová komunikace pro potvrzení transakcí
- Synchronní řešení bez nové vrstvy logiky

Slabé stránky

- Výpadky musí řešit administrátor
- Nové servery musí přidávat administrátor

Příležitosti

- Relativně nová technologie
- Využívá existujícího MySQL serveru
- Multiplatformní
- Vyvíjen firmou Oracle

Hrozby

- Vyvíjen firmou Oracle
- Využívají binárních logů při replikacích
- V základním nastavení je single-master

3.6.3 Porovnání rozdílů mezi Galera clusterem a MySQL Group replikacemi

Oba tyto přístupy jsou založeny na podobném principu. Liší se ovšem v částech implementace. V této podkapitole budou popsány rozdílnosti přístupů.

3.6.3.1 Podobnosti

Galera i Skupinové (Group) replikace využívají řádkově založený („row-base“) formát binárních logů.

Využívají také skupinovou komunikaci pro ustanovení kvóra, které umožňuje předávání informací o stavu, v jakém se nachází, a také pro potvrzení transakcí.

Další podobnost je v unikátním označováním transakcí globálním sekvenčním číslem. Galera má vlastní GTID a Skupinové replikace využívají GTID, které bylo přidáno do MySQL ve verzi 5.6 viz kapitola Galera.

Obě řešení využívají certifikace, ovšem Skupinovým replikacím stačí potvrzení od většiny, kdežto Galera vyžaduje potvrzení od všech členů.

3.6.3.2 Rozdíly

Galera využívá pro komunikaci kruhovou topologii s předáváním tokenu. Skupinové replikace oproti tomu jsou založeny na peer to peer Paxos algoritmu[10].

Dalším rozdílem je to, že Galera potřebuje přidat navíc vrstvu nad MySQL, jež umožňuje ukončit lokální transakci za předpokladu, že nastane certifikační konflikt. Skupinové replikace využívají Vysokou prioritu transakcí (High Priority Transaction[11]) systém v InnoDB, který umožňuje detekování a zachycení konfliktu.

Skupinové replikace používají pro replikace binární logování, oproti tomu Galera toto binární logování povolené mít nepotřebuje. Využívá vlastní soubor gcache s předem určenou velikostí, jež funguje jako kruhový zásobník.

Výpadky jsou řešeny v rámci Galery téměř automaticky. Skupinové replikace vyžadují větší interakci s administrátorem, například při rozpojení lokalit je nutné, aby administrátor určil, která lokalita bude dostupná. Při výpadku, kdy není možné inkrementálně do replikovat data, má Galera systém SST, který tento stav řeší. Skupinové replikace SST nemají. Administrátor musí nové instance vytvářet sám, samozřejmě za předpokladu, že existují všechny binární logy.

Galera je multi-master ve výchozím stavu, skupinové replikace jsou ve výchozím stavu single-master a doporučují tuto architekturu pro produkční použití. Skupinové replikace určí master instanci při výpadku sama.

---

[10] Paxos algoritmus – algoritmus, které řeší consensus, tedy postup přijímání a zamítání zpráv v distribuovaném prostředí při skupinové komunikaci. [13]

[11] Hight Priority Transaction – musí zajistit všechny zápisové zámky a popřípadě zamítnout všechny probíhající transakce, které by pracovali s řádky, které jsou zamknuté.

Galera má v základu implementovaný arbitr a možnost nastavovat různé váhy jednotlivým instancím v clusteru. Má také implementovanou možnost vypnutí či zapnutí kauzálního čtení.

3.6.4 Oracle RAC

Oracle RAC používá Oracle Clusterware software k provázání více serverů. Oracle Clusterware je software, který je integrovaný do Oracle databází, může ovšem běžet nezávisle na databázové instalaci. Umožňuje provádět operace nad provázanými servery, chová se tak ke skupině serverů jako k jedné instanci. Automaticky hlídá instance serveru a umožňuje restart při pádu. [15]

Oracle RAC spoléhá na distribuovanou architekturu. Soubory jsou přístupné pro každý uzel v clusteru. Je zde možnost konfigurovat sdílené uložiště. Oracle ASM poskytuje virtualizovanou vrstvu mezi databázovým serverem a uložištěm. Chová se ke skupině disků jako k jednomu disku. Pro správné fungování technologie Oracle RAC je nutné dodržet několik podmínek, a to stejný hardware, stejná konfigurace operačního systému a aplikace ve stejné verzi na všech členech, jelikož data jsou uloženy na všech serverech.

Pro zpracování a distribuci transakcí jsou definovány dvě vyrovnávací paměti Globální Obsluha Cache „Global Cache Service (GCS)“ a Globální Obsluha Fronty „Global Enqueue Service (GES)“. GES obsahuje zámky cache, zámky pro transakce i zámky pro tabulky. GCS obstarává bloky, které jsou distribuovány mezi instancemi. Zřejmé je tedy, že díky těmto dvěma vyrovnávacím pamětem vzniká režie, kdy jsou transakce nejprve zapsány do paměti prvního databázového serveru, z ní jsou distribuovány do vyrovnávacích pamětí všech členů. Dále pak jsou tyto transakce aplikovány, díky tomuto principu může docházet k vyšší latenci. [16]

Vlastní práce

3.7 Případová studie Sklik

Tato kapitola se zabývá popisem implementace praktické časti. V rámci implementace bude popsán Sklik.cz use case, dále bude popsáno několik druhů řešení a následné řešení v pseudokódu.

Společnost Seznam.cz a.s. již existuje více než 20 let. Spravuje několik desítek služeb, každá služba obsahuje několik stovek serverů a několik stovek databázových clusterů. Všechny MySQL řešení využívají bidirectional master-slave architektury a jsou založené na řešení od Percony.

Seznam.cz a.s. disponuje dvěma data centry. Tato data centra se v poslední době stávají nedostatečnými, proto se uvažuje nad třetí lokalitou. Tato lokalita by ovšem znamenala, že bude nutné rozdělit MySQL clustery.

Firma se potýká s problémy popsaných v kapitole Vznik nekonzistence nebo ztráty dat. Tyto problémy způsobují situace, jenž musí řešit administrátoři a programátoři řešit či opravovat velice dlouhou dobu.

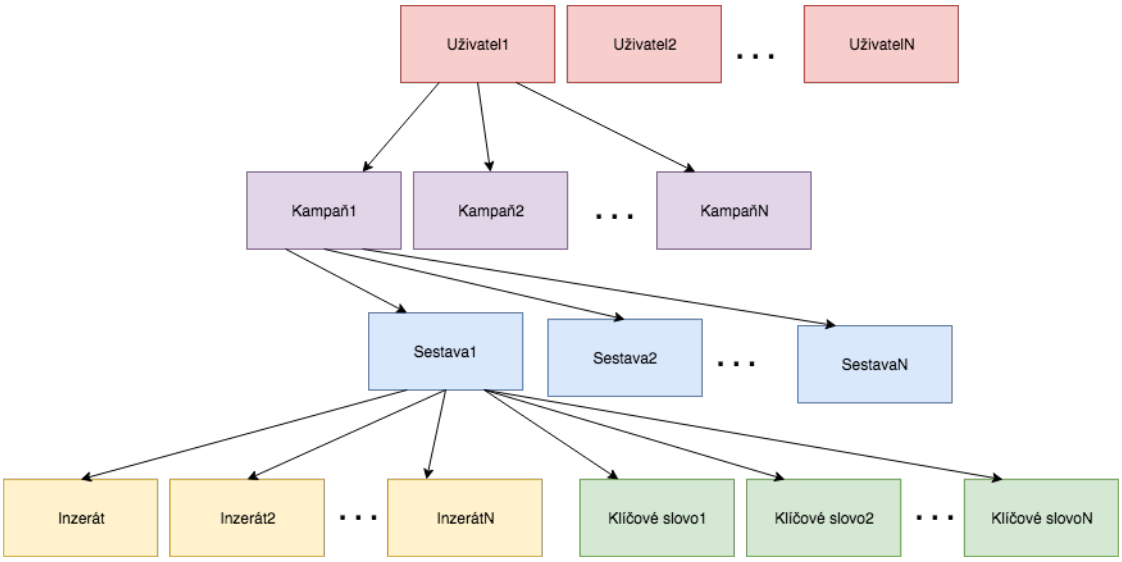
Seznam.cz a.s. se rozhodla pro Galera cluster řešení a její virtuálně synchronní replikace, převážně kvůli jejímu dlouholetému vývoji i komunitě, která tento systém používá a automatické správě při výpadku či přidávání nových členů do clusteru.

3.8 Struktura dat v systému Sklik

Sklik je služba firmy Seznam.cz a.s.. Tato služba se zaměřuje na vytváření a vydávání reklamy. Tento systém je velice komplexní, a proto bude nastíněna pouze část, která se týká této diplomové práce. Seznam.cz a.s. je českou společností, a proto se reklamní systém zaměřuje na české uživatele a český internet.

Reklamní systém jako takový má tisíce uživatelů, kteří chtějí inzerovat. Každý uživatelský účet obsahuje hierarchickou strukturu. Tato hierarchická struktura se skládá z kampaní, které obsahují sestavy a tyto sestavy následně obsahují další entity jako jsou samotné reklamy, bannery, klíčová slova, odkazy, zájmy, retargeting (speciální typ cílení), cílení, produktové skupiny a další entity. Zjednodušená struktura je znázorněna na obrázku Obrázek 13 Struktura účtu.

Obrázek 13 Struktura účtu



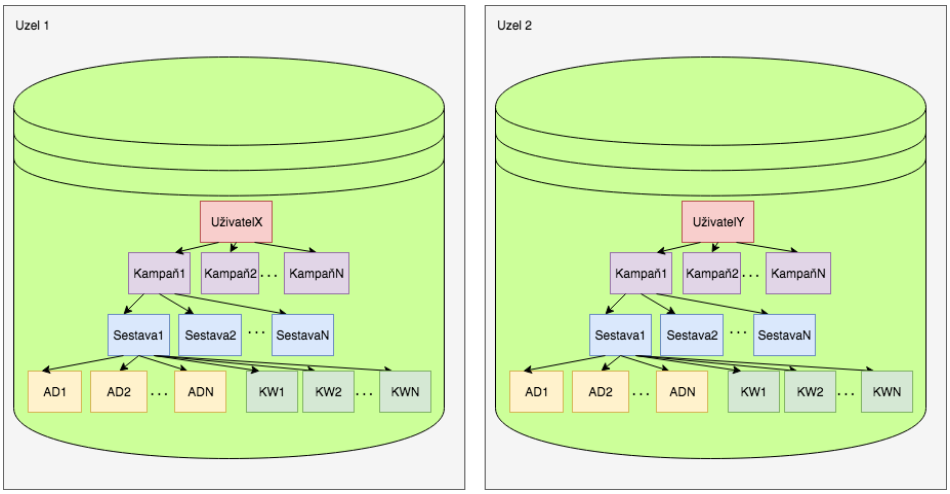
Zdroj: Vlastní zpracování

Struktura účtu je velice rozsáhla, a pro větší klienty může celá struktura jejich účtu dosahovat řádu giga bytů. Pokud by všechny účty byly uloženy v jedné databázi, docházelo by při jakýchkoliv úpravách jako jsou UPDATE, INSERT, ALTER a další k výpadkům celého systému. I operace SELECT, která je ve své podstatě neblokující, by nad takovou databází byla velice pomalá a neefektivní, proto bylo nutné vymyslet, jak řešit tuto situaci.

Dalším častým problémem v dříve navrhovaných databázích bylo použití základního (defaultního) databázového enginu. Databázový engine, který byl v dřívějších verzích MySQL, byl MyISAM. Tento engine přináší velkou nevýhodu v zamykání celé tabulky při zpracování jakéhokoliv dotazu na modifikaci dat jako je UPDATE, INSERT a další. Skrze tuto vlastnost se začal používat engine InnoDB, který zamyká pouze měněné řádky. InnoDB je také odolnější proti poškození tabulek a obsahuje transakce.

Velikost databáze samozřejmě způsobila to, že bylo nutné využít bidirectionální architektury (viz kapitola Bidirectional architektura). Tedy byla vytvořena master instance, do které se zapisuje a několik slave instancí, které umožní škálovat čtecí operace.
Pro zajištění dostupnosti i v případě výpadků byla tato architektura postavena ve dvou lokalitách, tím se dosáhlo možnosti přepínání provozu v případě složitějších údržby databáze či sítí v daném data centru.

Obrázek 14: Uživatelé škálovaní napříč uzly



Zdroj: Vlastní zpracování

Použití bidirectionální architektury ovšem nevyřešilo zápisové operace. Pro toto řešení bylo nutné využít metodiky z kapitole Škálování v relačních databázích.

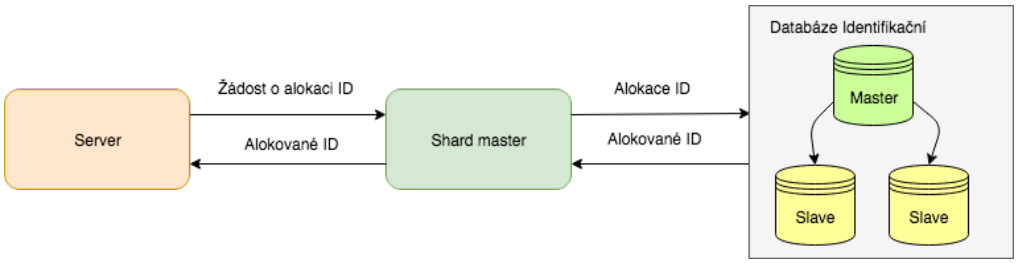
Pro vyřešení škálování zápisových operací lze využít škálování dat („shardování“) v kombinaci s rozdělením dat (viz Obrázek 14: Uživatelé škálovaní napříč uzly). Toto řešení umožňuje využít více master instancí v kombinaci s bidirectionální architekturou, jak bylo zmíněno v popisku této kapitoly.

Tohle řešení ovšem vyžaduje přidanou logiku. Tato logika je implementována serverem, který určuje kam jaká data budou zapsána.

To znamená, že přibyla další vrstva pro práci s MySQL. Jak již bylo zmíněno, struktura účtu je jasně hierarchická. To znamená, že je nutné mít pro jednoho uživatele data pohromadě. Škálování dat probíhá z těchto důvodů na úrovni uživatelů. To tedy znamená, že máme X uzlů, každý uzel sestává z master instance databázového systému a několika slave instancí. Uživatelé i všechny jejich data jsou rovnoměrně rozloženi mezi daný počet uzlů.

Získávání dat i alokace unikátních identifikátorů pro jakoukoliv entitu probíhá za pomoci „Shard - mastra“ (viz Obrázek 15: Alokace primárního klíče v clusteru). Vytváření nové entity probíhá tak, že se nejdříve alokuje unikátní klíč v rámci celého clusteru pomocí „Shard mastra“. Ten tento klíč nejprve zapíše do databáze identifikátorů, která je společná pro všechny uzly a poté jej vrací aplikaci, která jej zapíše do uzlu ke správnému uživateli.

Obrázek 15: Alokace primárního klíče v clusteru

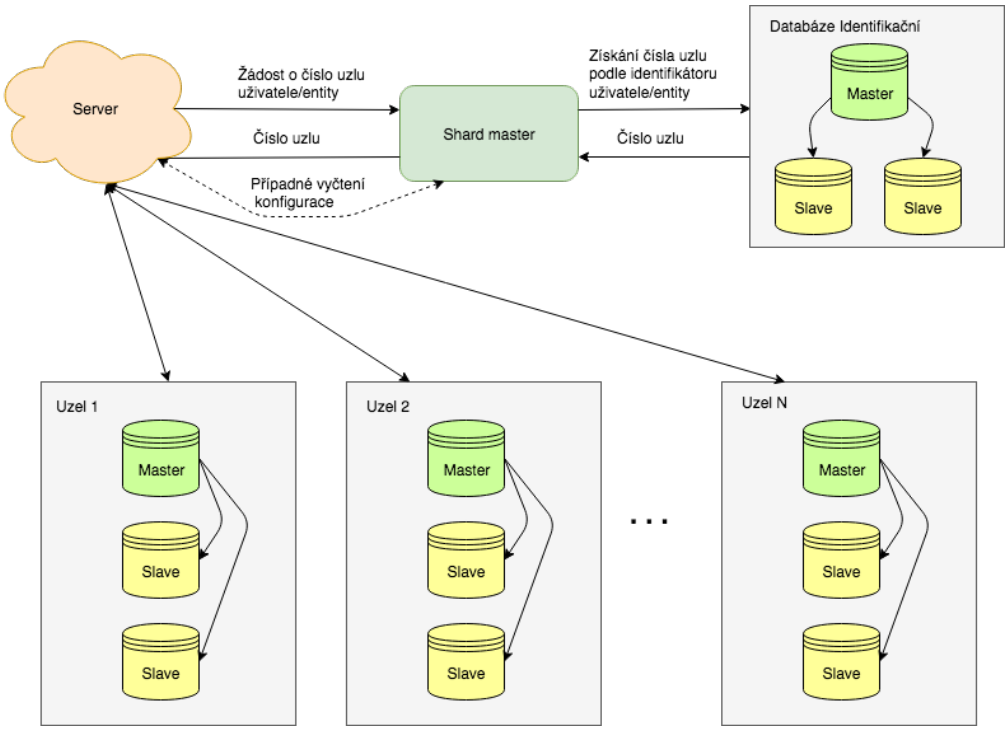


Zdroj: Vlastní zpracování

Čtení i úpravy probíhají podobným způsobem nejprve se za pomoci „Shard mastra“ zjistí, kterému uživateli tento identifikátor patří (viz Obrázek 16: Škálovana data skrze více uzlů a distribuovaná skrze Shard master). Pak se zjistí, na kterém uzlu je pro danou entitu záznam. „Shard master“ poté vrací číslo uzlu, kde se nachází veškeré informace pro daného uživatele. Aplikace buď má uloženou konfiguraci pro dané číslo uzlu, nebo si jej získá opět z „Shard - mastra“ a může vyčítat informace pro danou entitu.

Škálování dat umožnilo škálovat i zápisové operace ovšem v omezené podobě, jelikož není využit plný potenciál clusteru, protože máme pouze jednu master instanci pro daný uzel a využíváme asynchronní replikace.

Obrázek 16: Škálovana data skrze více uzlů a distribuovaná skrze Shard master

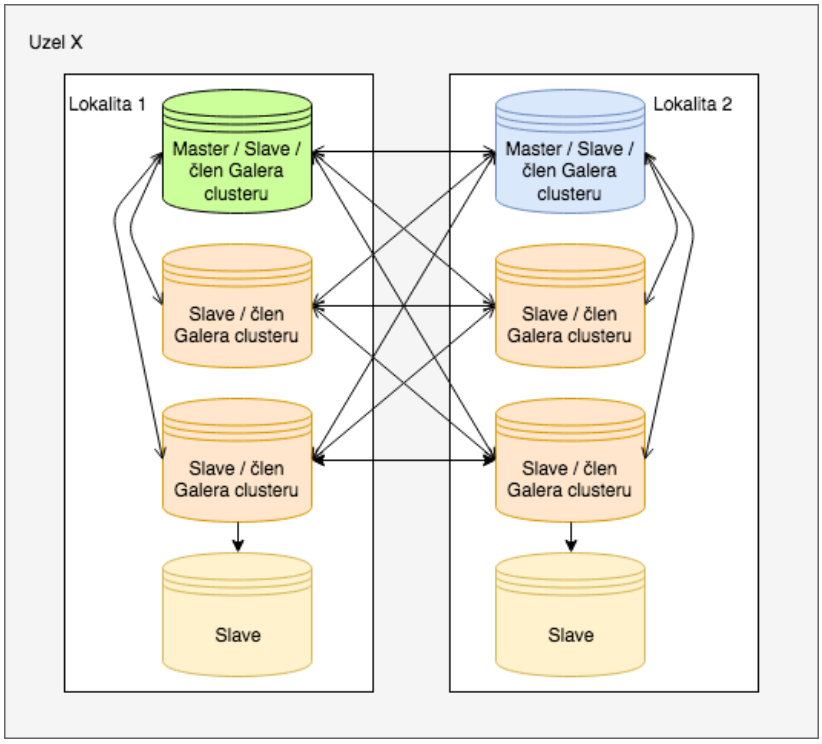


Zdroj: Vlastní zpracování

Pro využití plného potenciálu clusteru je nutné použít některé z řešení synchronních replikací. Toto řešení umožní nejen to, že data budou synchronní na všech instancích daného uzlu. Tudíž je možné využít škálování čtení bez opožděných replikací. Další výhodou je možnost zapisovat do všech instancí daného uzlu, a to i bez hrozby vzniku nekonzistence dat.

„Shard – master“, který je implementovaný interně, ovšem nebyl připraven na toto řešení. Proto bylo nutné navrhnout řešení, které nám umožní využít potenciálu, který nabízí synchronní replikace.

Obrázek 17: Uzel využívající Galera cluster



Zdroj: Vlastní zpracování

3.9 Prerekvizity pro využití Galera clusteru

Jelikož jsme se rozhodli využít pro virtuálně synchronní řešení Galera cluster, na základě několika výhod. Galera využívá již několik let a má velkou uživatelskou i vývojářskou komunitu. Další výhodou je takzvaný „automatic fail-over“, tedy to, že se Galera cluster umí automaticky při správné konfiguraci přepnout i v případě výpadku. Umí

také automatické začlenění nových členů clusteru pomocí IST a SST i skrze další výhody, které byly popsány v kapitole Galera.

Pro otestování navrhovaných řešení je v této kapitole popsáno abstraktní řešení Sklik use-case, využívající XMLRPC/FASTRPC protokol.

3.9.1 Server pro zpracování reklamy (AD server)

Server, který se obstarává abstraktní funkcionalitu pro zpracování inzerátů podle struktury na obrázku Obrázek 13 Struktura účtu . Obsahuje základní funkcionalitu pro manipulaci a listování kampaní, sestav a inzerátů.

Funkce:

`campaign.create` - Metoda, která vytvoří kampaň

`campaign.update` - Metoda, která upraví kampaň

`campaign.list` - Metoda, která vylistuje kampaň/e

`group.create` - Metoda, která vytvoří sestavu

`group.update` - Metoda, která upraví sestavu

`group.list` - Metoda, která vylistuje sestavu/y

`ad.create` - Metoda, která vytvoří inzerát

`ad.update` - Metoda, která upraví inzerát

`ad.list` - Metoda, která vylistuje inzerát/y

`user.list.account` - Metoda, která vylistuje na základě parametrů účet uživatele

Funkce pro vytváření kampaní, sestav a inzerátů mají podobnou strukturu viz psedokód vytváření.

```
def create(user_id -> int, attributes -> list(dict)) -> dict:
   # Kontrola duplicit a správnosti vstupu a přístupu k entitám.
   ok = check(user_id, attributes)
   if not ok:
      return diagnostics
   length_new_ids = len(attributes)
   # Alokace nových identifikátorů skrze „shard master".
   ids = id_manager_proxy.allocate_ids(user_id,
            entity, length_new_ids)
   # Získání připojení do databáze.
   shard_database = get_shard(user_id)
```

```
    # Provedeni samotného dotazu.
    result = shard_database.execute(
     "INSERT INTO entity (attributes,…) VALUES (%s, …)",
         attributes)
     return result # Nové identifikátory.
```

Funkce pro editaci kampaní, sestav a inzerátů mají podobnou strukturu viz psedokód editace.

```
def update(user_id -> int, attributes -> list(dict)) -> dict:
    # Kontrola duplicit a správnosti vstupu a přístupu k entitám.
     ok = check(user_id, attributes)
    if not ok:
       return diagnostics
    # Získání připojení do databáze.
    shard_database = get_shard(user_id)
    # Provedeni samotného dotazu.
     result = shard_database.execute(
     "UPDATE entity SET attribute=%s, …", attributes)
     return result # Informace o změně.
```

Funkce pro získání informací o kampaních, sestavách a inzerátech mají podobnou strukturu viz psedokód editace.

```
def list(user_id -> int, restriction_filter -> dict,
       display_option -> dict) -> dict:
    # Ověření přístupu k entitám.
     ok = check_access(user_id, restriction_filter)
    if not ok:
       return diagnostics
    # Získání připojení do databáze.
    shard_database = get_shard(user_id)
    # Získání sloupců, podmínek a třídících omezení z atributů.
    select_attributes = get_select_attributes(restriction_filter,
                         display_columns)
    # Provedeni samotného dotazu.
     result = shard_database.execute(
         "SELECT {display_columns} FROM entity {joins} {condition}
```

```
{ordering_and_limits}",
        select_attribute))
    return result # Informace získané danými filtry.
```

3.9.2 Shard master (Id manažer)

Srdcem celého systému je takzvaný Id (identifikátor) manažer neboli „Shard-master“ (viz Obrázek 16: Škálovana data skrze více uzlů a distribuovaná skrze Shard master). Tento server se stará o přiřazování unikátních identifikátorů pro veškeré entity uživatele napříč celým clusterem, kde jsou data rozdělena na několik „shardů“ (viz Obrázek 14: Uživatelé škálovaní napříč uzly a Obrázek 15: Alokace primárního klíče v clusteru). Tato komponenta je nejkritičtějším místem toho, aby celý systém fungoval správně. Popíši zde opět pouze abstraktně (pseudo kódem) jak funguje daný server.

Funkce:

`id.manager.allocate` - Metoda, která alokuje (rezervuje) unikátní identifikátory.

`node.config.listAll` - Metoda, která vylistuje všechny konfigurace pro databáze v clusteru

`node.config.get` - Metoda, která vrací pouze konfiguraci pro dané číslo „shardu“

`user.findById` - Metoda, vrací číslo „shardu“ podle uživatelského identifikátoru

`user.findByEntityId` - Metoda, která vrací číslo „shardu“ podle identifikátoru entity.

Funkce pro alokaci unikátních identifikátorů pro jednotlivé entity viz psedokód alokace.

```
def allocate_ids(user_id -> int, entity -> str, count -> int) -> list:
   # Všechny změny jsou prováděny v transakci.
   id_db_connection.start_transaction()
   try:
        # Získání posledního unikátního klíče (čísla).
        max_id = id_db_connection.execute(
        "SELECT max_id FROM id_manager WHERE entity={entity}", entity)
        new_max_id = max_id + count
        # Příprava nových identifikátorů.
        new_ids = [(user_id, new_id)
                   for new_id in range(max_id, new_max_id)]
        # Zapsaní nových identifikátorů do databáze.
```

```
            id_db_connection.execute(
                "INSERT INTO entity_ids VALUES ({user_id}, {new_id}), …",
        new_ids)
            id_db_connection.execute(
                "UPDATE id_manager SET max_ids = {new_max_id} WHERE
entity={entity}",
                new_max_id)
            id_db_connection.commit()
    except:
        # Něco se pokazilo vracíme všechny změny.
            id_db_connection.rollback()
    return new_ids
```

Funkce pro získání konfigurace pro všechny uzly nebo jeden jsou téměř stejné viz psedokód listování konfigurací.

```
def list_cofiguration(shard_id -> optional int) -> list:
    # Listujeme konfigurace pro dané „shardy"/uzly.
    # Podmínky se aplikují pouze pokud je zadán shard_id, kdy se
    # získává konfigurace pouze pro jeden „shard".
    result = id_db_connection.execute(
     "SELECT * FROM node_conf {where_condition}", shard_id)
    return result
```

Funkce pro získání čísla „shardu“ neboli uzlu na základě identifikátoru uživatele či identifikátoru entity jsou téměř stejné viz psedokód vyhledání čísla nodu.

```
def find_shard_id(user_ids -> list(int), entity -> str,
        ids -> optional list(int)) -> list(dict):
    # Získáme číslo „shardu"/uzlu na základě identifikátoru
    # uživatele. Pokud máme zadánu entitu je poskládán složitější
    # dotaz za pomoci JOIN klauzulí na tabulky entit.
    result = id_db_connection.execute(
     "SELECT shard_id, user_id FROM user_node {joins} {where_condition}",
         (ids, entity, user_ids))
    return result
```

Pro správné fungování v rámci celého systému bylo nutné napsat několik knihoven, které zaobalují funkcionalitu a volání právě tohoto Id manažeru. Při návrhu využití potenciálu synchronních replikací bylo zvažován i scénář s téměř nulovými úpravami právě těchto knihoven, ale i s většími úpravami.

3.9.3 Galera cluster proměnné

Použití Galera clusteru znamená, že máme další vrstvu logiky v relačním databázovém systému. Právě z tohoto důvodu přidává Galera další stavy a globální proměnné, které je možné vyčítat a vyhodnocovat. Díky těmto stavům a proměnným je možné zjistit stav v jakém je člen Galera clusteru, jak je člen či cluster vytížen, v jakém stavu jsou replikace a další informace. Nejdůležitější proměnné budou popsány v této podkapitole.

Proměnné, které udávají v jakém stavu je člen clusteru.

- **wsrep_ready** – Proměnná, která ukazuje, zda člen může přijímat dotazy, je ve stavu ON, pokud přijímá dotazy a tedy „*write-sety*“. Ve stavu OFF je, pokud dotazy zahazuje.
- **wsrep_connected** – Pokud je připojení jeden či více členů clusteru v pořádku je tato proměnná nastavená na ON jinak je ve stavu OFF. Stav, kdy není připojení na jeden či více členů clusteru je nejčastěji špatnou konfigurací členů například konfigurační hodnoty ***wsrep_cluster_address*** nebo ***wsrep_cluster_name***.
- **wsrep_local_state** – Tato proměnná udává, v jakém stavu je daný člen (viz Obrázek 11: Stavy instancí v Galera clusteru). Úzce spojená je s touto proměnnou proměnná ***wsrep_local_state_comment*** kde je tento stav popsán v čitelné podobě.

O kontrolu stavu „zdraví“ replikací se starají tyto proměnné.

- **wsrep_local_recv_queue_avg** – Průměrná velikost fronty příchozích „*write-setů*“. Pokud je tohle číslo větší nežli 0.0 znamená to, že člen nestíhá aplikovat změny tak rychle jak přichází bude tedy docházet k opožďování.
- **wsrep_flow_control_paused** – Doba, kdy byl člen clusteru pozastaven z důvodu Flow Control neboli kontroly toku, tedy kdy člen nezpracovával žádné „*write-sety*“. Pokud je hodnota 0.0 znamená to, že člen nebyl vůbec pozastaven v průběhu kontroly toku. Z toho tedy vyplývá, že hodnota by měl být co nejblíže 0. Pokud je

hodnota vyšší je možné upravit proměnnou ***wsrep_slave_threads***, která definuje kolik vláken může aplikovat „*write-sety*“

- **wsrep_cert_deps_distace** – Hodnota, která udává průměrnou vzdálenost mezi nejnižším a nejvyšším sekvenčním číslem (seqno). Je to tedy hodnota, která udává kolik změn je možné aplikovat paralelně.

Detekce, zda dochází v clusteru ke zpomalení z důvodu sítě je udána v těchto proměnných.

- **wsrep_local_send_queue_avg** – Průměrná hodnota, která ukazuje délku fronty na příchozí dotazy před posledním vyprázdněním `(FLUSH STATUS)`.

Další důležitá proměnná.

- **wsrep_incoming_addresses –** V této proměnné jsou uloženy adresy všech členů v clusteru. Hodnoty jsou zde jako v textové podobě (IP adresa/host:port) oddělené čárkou např. 127.0.0.1:3306,172.1.17.123:3306

3.10 Shard master vrací virtuální uzly

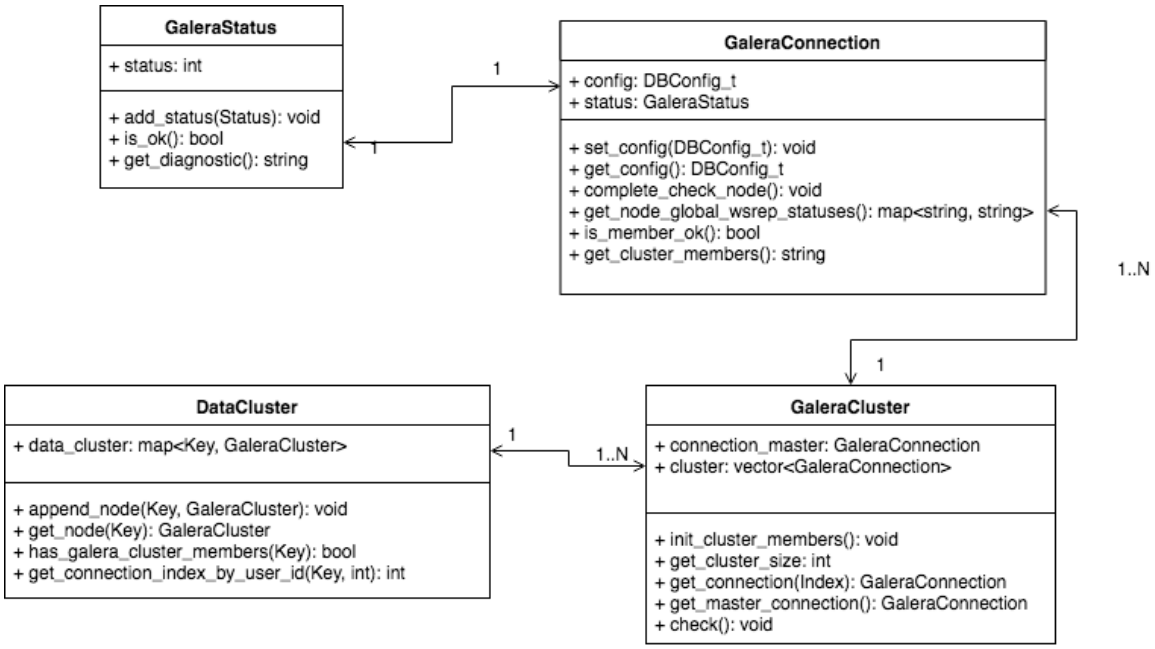
Jak si můžeme povšimnout na ilustraci Obrázek 16: Škálovana data skrze více uzlů a distribuovaná skrze Shard master a i v pseudokódu v podkapitole Shard master (Id manažer), kde se popisuje získávání čísla uzlu či „shardu“, máme všechny informace o daném uzlu uloženy v MySQL, do které má přístup pouze „Shard-master“.

Pokud použijeme Galera cluster a budeme chtít využít potenciál, který nabízí, tedy využít automatického začleňování nových členů do clusteru, musíme upravit stávající „work flow“. Pro tento případ bude muset vzniknout nová funkcionalita (vrstva), která obstará komunikaci s uzly („shardy“). Jelikož každý jednotlivý uzel obsahuje vlastní Galera cluster, tedy obsahuje několik MySQL instancí, které mají nainstalovanou Galeru a jsou propojeny virtuálně synchronními replikacemi (viz Obrázek 17: Uzel využívající Galera cluster).

Tohle řešení umožňuje použít ostatní servery, které potřebují komunikovat s uzly, bez úprav knihoven. Další výhodou tohoto řešení je garance získání vždy stejného člena Galera clusteru pro daný uzel, pokud je cluster stabilní.

Pseudokód obsahuje definice tříd a několik důležitých metod, jako je kontrola stavů, výběr člena clusteru. Další metody jsou v části přílohy skrze rozsah definic (Pseudokód tříd zaštiťující datový cluster).

Obrázek 18: Diagram tříd obsluhující datový cluster



Zdroj: Vlastní zpracování

Tento pseudokód popisuje rozšířenou funkcionalitu, která umožňuje využít plný potenciál Galera clusteru. Jakmile tento kód zaintegrujeme do získávání čísla uzlu („shardu“), bude pseudokód vypadat.

```
def list_cofiguration(shard_id -> optional int) -> list:
    # Získáme velikost clusteru, tedy počet "shardu".
    size_of_cluster = get_size_of_cluster()
    galera_member_id = 0
    # Pokud je číslo uzlu větší než-li velikost všech uzlů, jedná
    # se o člena galera_clusteru.
    if shard_id > size_of_cluster:
        galera_member_id = shard_id
        # Abychom získali správný uzel musím použít modulo počtem
        # všech uzlů (masterů) v clusteru.
        # Příklad převodu:
        # Při maximálním počtu uzlů 8 a hledaném číslu uzlu 17.
        # 17 / 8 = 2 a pomocí modula můžeme získat číslo člena = 1
        # tedy při maximálním počtu uzlů 8 a hledaném číslu uzlu 20
        # 20 / 8 = 2 číslo člena 20 % 8 = 4
        shard_id = shard_id / size_of_cluster
```

```
# Listujeme konfigurace pro dané „shardy“/uzly.
# Podmínky se aplikují pouze pokud je zadán shard_id, kdy se
# získává konfigurace pouze pro jeden „shard“.
result = id_db_connection.execute(
    "SELECT * FROM node_conf {where_condition}",
    shard_id)
all_configs = []
for shard in result:
    # Získáme instanci třídy GaleraCluster.
    member = shard.get_member(shard.id)
    # Získáme celý cluster pro daný "shard".
    cluster = member.get_cluster()
    # Násobíme indexem proto je nutné začínat od 1.
    index = 1
    # Iterujeme přes všechny členy clusteru, abychom upravili
    # konfiguraci a vytvořili, tak virtuální uzly ("shardy").
    for galera_connection in cluster:
        config = galera_connection.get_config()
        # vypočítáme číslo virtuálního "shardu" výpočet
        # je založen na modulu
        # (počet všech "shardu" * index) + číslo shardu
        # Tímto výpočtem zajistíme, že pro každý člen clusteru,
        # získá své unikátní číslo
        # Příklad:
        # Uzel 1, má 4 členy, počet všech uzlu je 8.
        # člen 1 => (8 * 1) + 1 = 9
        # člen 2 => (8 * 2) + 1 = 17 ...
        # Uzel 4, má 4 členy, počet všech uzlu je 8.
        # člen 1 => (8 * 1) + 4 = 12
        # člen 2 => (8 * 2) + 4 = 20 ...
        config.id = (size_of_cluster * index) + shard.id
        if galera_member_id and config.id == galera_member_id:
            all_configs.append(config)
            break
        index += 1
return all_configs
```

Pro získání čísla uzlu („shardu“) na základě identifikátoru entity či uživatele bude upravená funkce vypadat následovně.

```
def find_shard_id(user_ids -> list(int), entity -> str,
      ids -> optional list(int)) -> list(dict):
    # Získáme číslo „shardu"/uzlu na základě identifikátoru
    # uživatele. Pokud máme zadánu entitu je poskládán složitější
    # dotaz za pomoci JOIN klauzulí na tabulky entit.
    result_shards = id_db_connection.execute(
        "SELECT shard_id, user_id FROM user_node {joins} {where}",
        (ids, entity, user_ids))
    # Projdeme všechny výsledky a upravíme uzel.
    for shard in result_shards:
        # Získáme "index" člena Galera clusteru pro daný uzel
        # viz get_index_by_user_id().
        shard.galera_member_id = cluster.get_index_by_user_id(
            shard.shard_id, shard.user_id)
        # Vypočítáme virtuální uzel, který je vypočítán z maximálního
        # počtu uzlů, který je vynásoben existujícím uzlem a je
        # přičten "index" člena clusteru daného uzlu.
        shard.nodeId = (max_nodes * u.nodeId)  + shard.galera_member_id
    return result_shards
```

Další důležitá část se týká kontroly stavů členů, a to pomocí metody zvané „health check". Je nutné, aby tato metoda periodicky kontrolovala stavy proměnných daných členů Galera clusteru viz Galera cluster a metoda v pseudokódu `GaleraConnection`, `complete_check()`.

Tato kontrola („healthckeck") bude probíhat každých 10 sekund a bude nastavovat stav instance `GaleraStatus` pro všechny `GaleraConnection`. Díky této periodické kontrole zajistíme, že se nastaví stav a poté se již kontroluje při jakémkoliv získávání pouze proměnná `GaleraStatus`, tedy není nutné připojování do databázových systémů jednotlivých členů.

Další velice důležitou součástí tohoto řešení je automatické vyhodnocování začlenění člena clusteru na základě dalších proměnných popsaných v kapitole Galera cluster . Hodnoty proměnných jako je ***wsrep_local_recv_queue_avg, wsrep_flow_control_paused, wsrep_cert_deps_distace, wsrep_local_send_queue_avg.***

Těmto proměnným budou přiřazeny určité váhy a na základě těchto vah bude systém rozhodovat, zda daného člena zařadit do výběru možných členů daného uzlu či nikoliv.

Možnost automatického vyhodnocování začlenění člena clusteru, nastavování vah i samotné kritérium, které říká, jak velkou část (všechny členy, polovinu, čtvrtinu, jednoho či žádný) začlenit do výběru možných členů clusteru daného uzlu.

3.11 Shard master vrací speciální typ

Dalším možných řešením by bylo upravit při vyčítání návratovou hodnotu, tedy konfiguraci uzlu. Úprava by zahrnovala přidáním speciálního typu, který by jasně definoval, že se jedná o člena Galera clusteru.

Tohle řešení ovšem znamená, že bude nutné upravit knihovny, které komunikují s Id manažerem, abychom mohli využít potenciál Galera clusteru pro daný uzel. Vycházíme z řešení a pseudokódu v podkapitole Shard master vrací virtuální uzly. Jediným rozdílem je to, že se nemusí vypočítávat virtuální číslo uzlu.

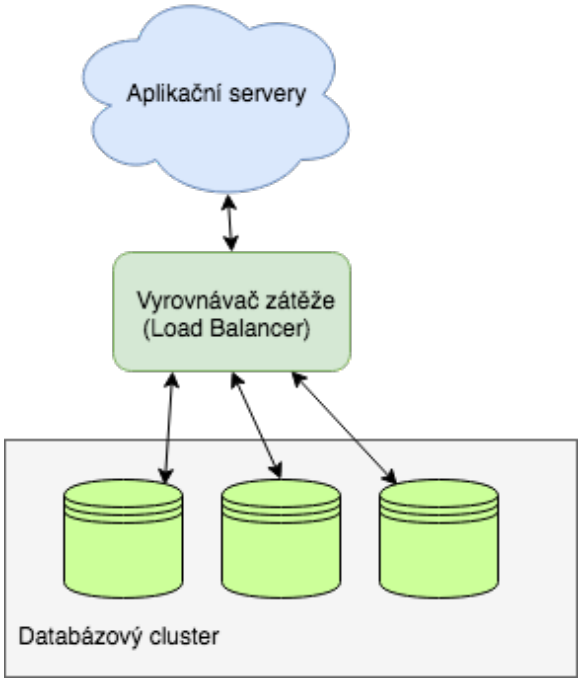
Ovšem jak již bylo řečeno, logika směrování na členy Galera clusteru uzlu, aby byl využit potenciál by se musel přesunout do knihoven, a i některé servery, které komunikují s Id manažerem napřímo.

Právě z těchto důvodu bylo, od tohoto řešení zamítnuto. Vývoj by byl náročnější, mohl by přinést více bugů a problémů na více místech, tzn. větší údržba a dohledávání chyb. Řešení popsané v kapitole Shard master vrací virtuální uzly, se týká pouze úpravy Id manažeru, proto je z pohledu implementačního a údržbového z dlouhodobějšího hlediska jednodušší a prospěšnější. Pokud by se vyskytla chyba oprava by se týkala pouze jednoho bodu, a ne více serverů a knihoven.

3.12 Externí řešení

Musím zde zmínit i externí řešení, tedy řešení, které je vyvíjeno a spravováno jinou firmou či skupinou (komunitou) vývojářů. Externí řešení jsou výbornou volbou, pokud se navrhuje nové řešení či není problém rozšířit stávající řešení. Jedním z těchto řešení je ProxySQL a dalším řešením je MaxScale. Obě tato řešení fungují na principu vyrovnávání zátěže (viz Obrázek 19: Vyrovnávaní zátěže) tzn. chovají se ke skupině instancí databázových serveru jako k jednomu celku.

Obrázek 19: Vyrovnávaní zátěže



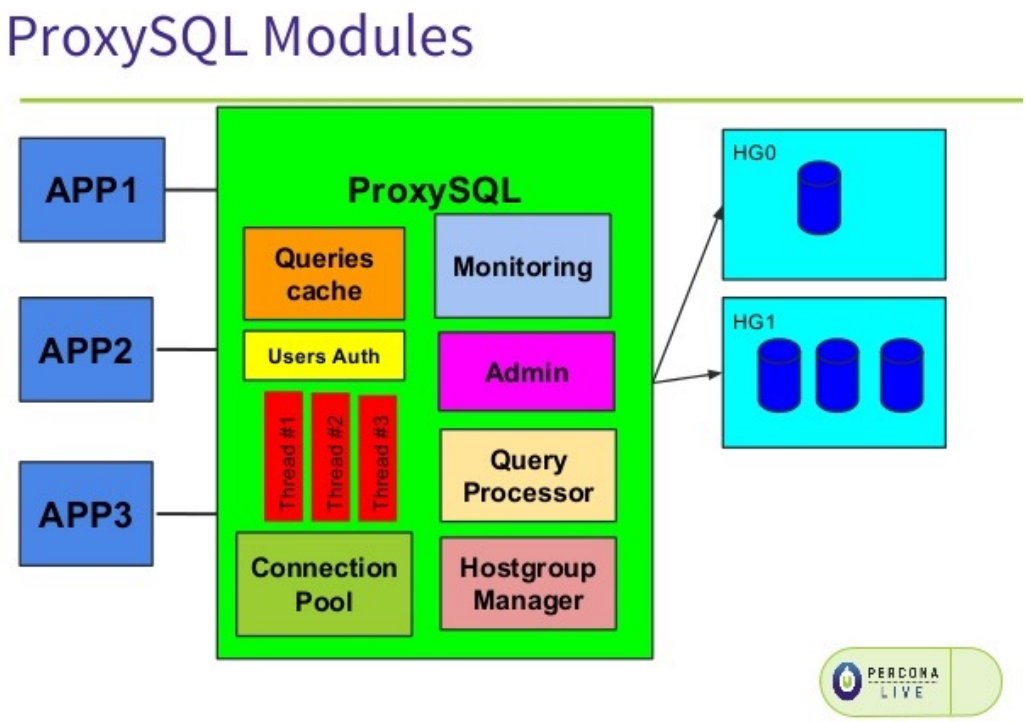
Zdroj: Vlastní zpracování

Pro řešení problému pro Sklik by bylo možné využít tato řešení, ovšem za podmínek několika ProxySQL či MaxScale. Pro každý uzel by musela být vlastní, aby bylo možné směřovat dotazy do clusteru. Jedním z požadavků na implementaci bylo směřovat operace, pokud to je možné vždy na jednoho člena Galera clusteru uzlu, tohle řešení je díky virtuálním uzlu popsaným v kapitole Shard master vrací virtuální uzly je díky výpočtu virtuálního uzlu zajištěno.

3.12.1 ProxySQL

Tento nástroj umožnil databázovým systémům, které byly rozložené do více instancí, tedy škálované, přistupovat jako k celku. Umožňuje také kontrolovat dotazy i vytíženost nad databázovými systémy. Přináší několik benefitů, jako je inteligentní rozdělení zátěže nad více databázovými systémy. Umí rozlišit zápisové i čtecí dotazy, tedy pokud je použita architektura Master-Slave zápisové dotazy směřuje pouze na master a čtecí rozděluje na slavy. Umožňuje analyzovat operace nad clusterem a díky tomu optimalizovat zvýšit efektivitu práce s uzlem. ProxySQL zabezpečuje vysokou dostupnost (High Availability). Zvládá také zotavení se z pádů databázového systému. [11]

Obrázek 20: ProxySQL



Zdroj: [11]

ProxySQL (viz Obrázek 20: ProxySQL) obsahuje vlastní Connection Pool, tedy připojení do MySQL, aby je bylo možné využít více krát. Je zde manažér skupiny serverů, na které se vyrovnává zátěž a sleduje jejich stav. Uživatelská data jsou zde zašifrovaná a uložena. Umožňuje také sledování a sbírání metrik.

3.12.2 MaxScale

Je to nástroj pro škálované („shardování“) relační databázové systémy od firmy MariaDB, která tento nástroj spravuje a vyvíjí. Umožňuje přistupovat k více databázím zapojených do jednoho clusteru jako celku. Také umožňuje vysokou dostupnost („High Availability“), zabezpečení a škálovatelnost bez změny chování aplikace.

Opět je snahou minimalizovat čas, kdy jsou cluster nebo databáze nedostupné („downtime“). Zabezpečení přístupu do databázových systémů umožňuje nastavit firewall, kde je možné některé dotazy zakázat (black a white list) či limitovat a ochránit tak databáze před DDoS útoky. [12]

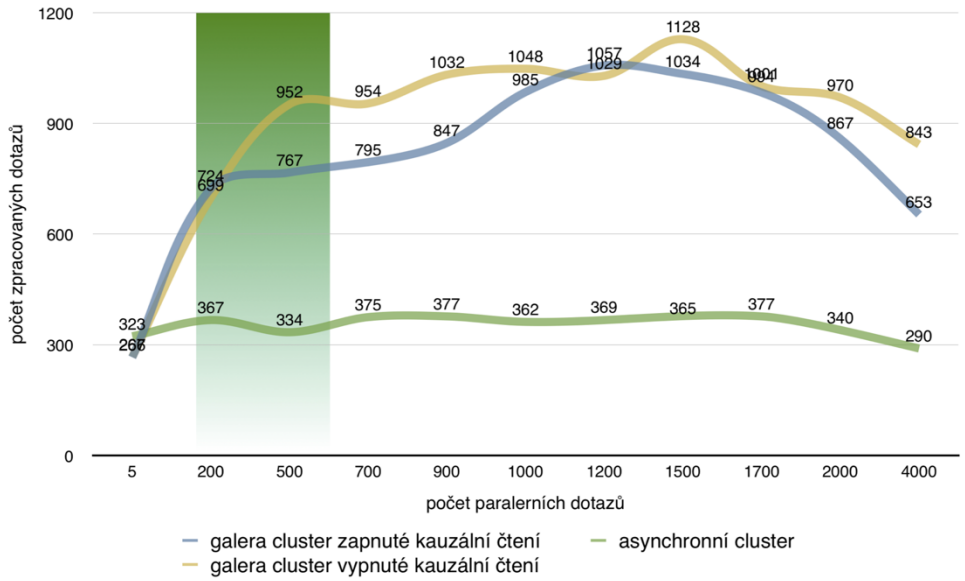
Výsledky a diskuse

Jak již bylo zmíněno v popisu kapitoly Struktura dat v systému Sklik, asynchronní replikace byly nahrazeny virtuálně synchronními replikacemi Galera clusteru. Toto řešení muselo být otestováno, zda tento převod byl výhodný.

Hlavním kritériem pro přechod byla nutnost mít konzistentní data v clusteru. Tuto konzistenci zajišťují právě virtuálně synchronní replikace. Dalším benefitem je možnost škálování čtení i zápisů, kdy můžeme využít i slave uzly, a to bez latence, která byla typická pro clustery, které využívali asynchronní replikace. Tyto výhody se projevují převážně při vysokém vytížení master instance databázového systému.

3.13 Porovnání řešení

Obrázek 21: Graf, kde je zobrazen počet dotazů, které jsou úspěšně zpracování v závislosti na vytíženosti porovnávaných typu cluster.



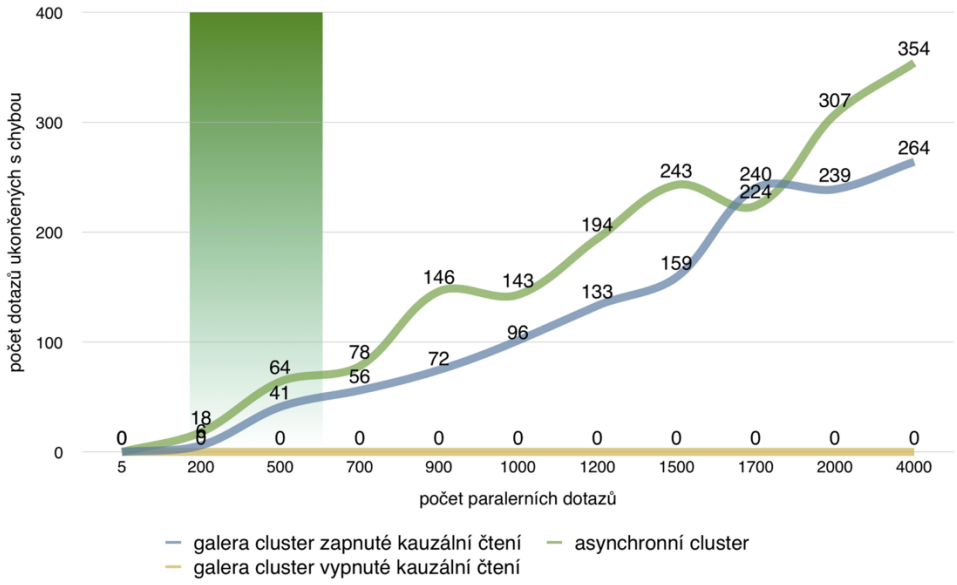
Zdroj: [13]

Na obrázku Obrázek 21: Graf, kde je zobrazen počet dotazů, které jsou úspěšně zpracování v závislosti na vytíženosti porovnávaných typu cluster. Je vidět, že počet zpracovaných dotazů při asynchronních replikací je několikanásobně nižší nežli při použití Galera clusteru.

Na dalším obrázku Obrázek 22: Graf zobrazující počet dotazů, které skončily s chybou v závislosti na vytíženosti porovnávaných clusterů. Jsou vidět chyby, které daný

databázový systém vrátil při zpracování paralelních dotazů. Z grafu je vidět, pokud máme zapnuté kauzální čtení dosahujeme 0 chyb ovšem při zapnutém kauzálním čtení je počet chyb („rollbacků“) vysoký. To je způsobeno tím, že při zpracování paralelních transakcí může docházet ke kolizím, a proto je nutné aplikace upravit, aby počítali s tímto scénářem a případně provedli opakování dotazu.

Obrázek 22: Graf zobrazující počet dotazů, které skončily s chybou v závislosti na vytíženosti porovnávaných clusterů.

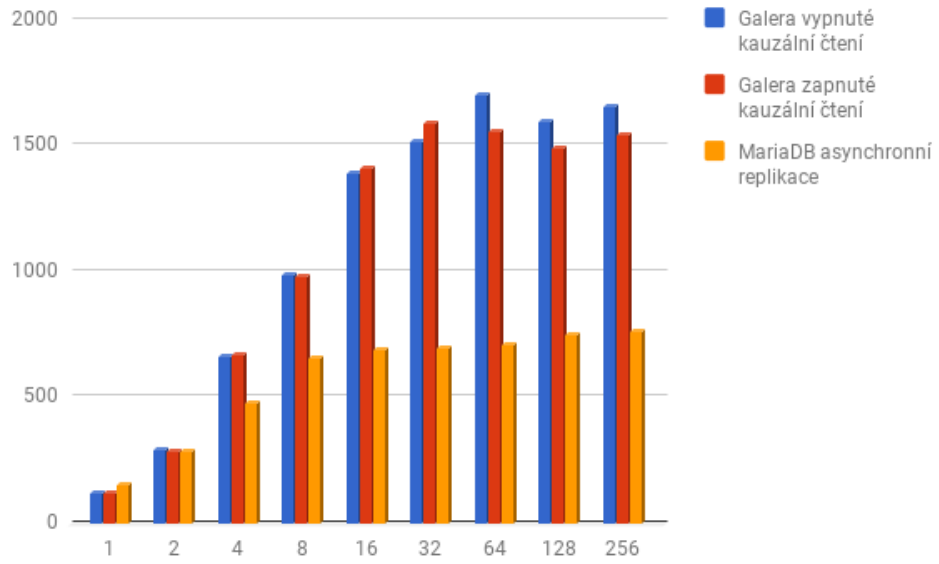


Zdroj: [13]

Grafy popsané viz výše jsou výsledkem testu nad databázovými servery, které jsou v provozních clusterech. Pro další porovnání jsem využil dvou strojů a testoval jsem pomocí nástroje *sysbench* výkon, kdy máme dvě instance databázového serveru. Instance databázového serveru běžely v Dockeru, obrazy („image“) a nastavení použité pro test jsou v přílohách Nastavení Docker Image.

Porovnal jsem opět viz grafy výše tři způsoby Galera cluster se zapnutým kauzálním čtením, Galera cluster s vypnutým kauzálním čtením a asynchronní replikace. Konfigurace databázového systému jsou v kapitole přílohy tak i konfigurace nástroje *sysbench* (viz 6.2).

Obrázek 23: Počet zpracování transakcí za sekundu

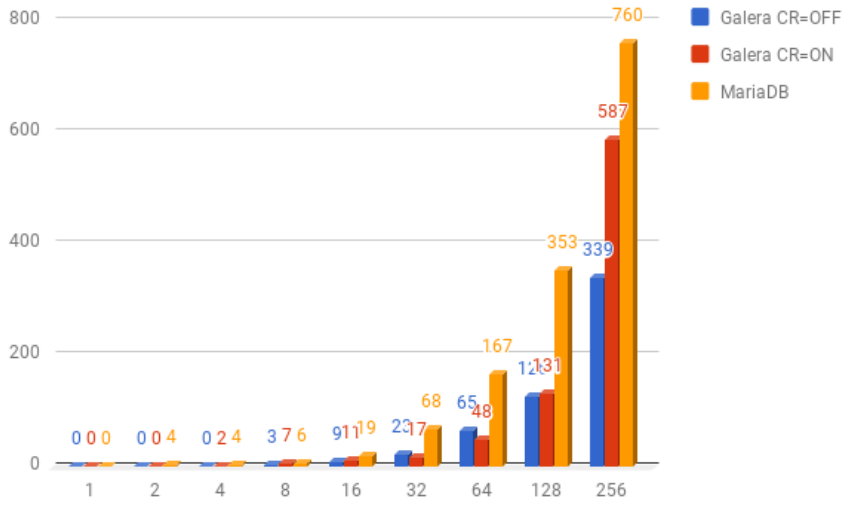


Zdroj: Vlastní zpracování

Na grafu Obrázek 23: Počet zpracování transakcí za sekundu je jasně vidět, že opět Galera zpracovala více paralelních dotazů za sekundu. Při vyšší konkurenci je z grafu čitelné, že když má Galera vypnuté kauzální čtení má o něco vyšší výkon než Galera s vypnutým kauzálním čtením. Asynchronní řešení má dá se říci stabilní výkon při vyšší konkurenci.

Další graf zobrazuje chyby, respektive počet ignorovaných chyb jako je chyba „deadlock“ (ER_LOCK_DEADLOCK, MySQL kód 1213), „timeout“ neboli vypršení času kdy se čeká na zámek (ER_LOCK_WAIT_TIMEOUT, MySQL kód 1205) a chyba kontroly čtení (ER_CHECK_READ, MySQL kód 1020). Tyto chyby jsou častým úkazem při konkurenčním zpracování dotazů. Proto je nutné, pokud počítáme s vysokou konkurencí v naší aplikaci, se na tyto chyby připravit a mít implementovanou funkcionalitu, která umožní provedení transakce znovu při těchto chybách. Galera má implementovaný tzv. „auto retry“, tedy funkcionalitu, kdy transakci zkusí automaticky provést znovu, jak již bylo popsáno v kapitole Galera cluster.

Obrázek 24: Počet ignorované chyb, při testu sysbench nástroje



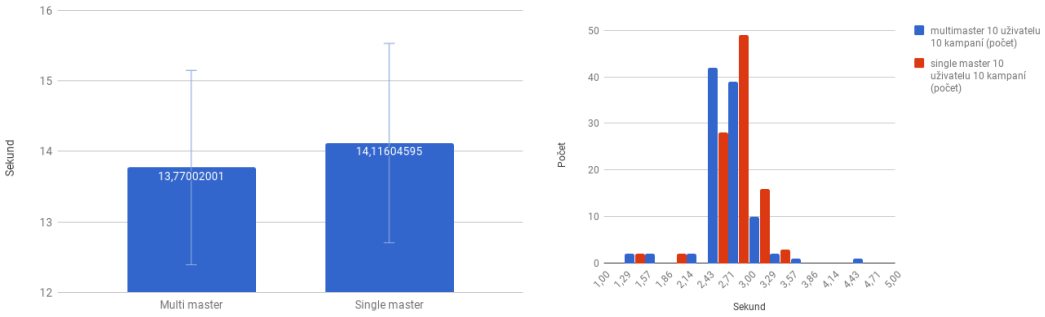
Zdroj: Vlastní zpracování

Řešení popsané v kapitole Shard master vrací virtuální uzly bylo porovnáno se stávajícím řešením. Server, který spravuje identifikátory a přístupy na uzly pro uživatele je programován v programovacím jazyce C++. Obě tyto řešení, jak přístup single master, tak i přístup multi master jsou implementovány jazyce C++. Byly tedy porovnán přístup single master a multi master. Hlavním kritériem bylo získání minimálně stejného výkonu pro zpracování požadavků jako při použití single master.

Testovací skript, který musel vzniknout, aby bylo možné otestovat přibližný scénář chování uživatele obsahuje vytváření kampaní, sestav a inzerátů. Obsahuje také listování všech těchto entit. Skript je volán paralelně za využití procesů. Testováno bylo několik scénářů. Scénář malé zatížení, kdy pro 10 uživatelů bylo vytvořeno pro každého uživatele 10 kampaní. Další test byl, kdy opět pro 10 uživatelů bylo vytvořeno pro každého uživatele 20 kampaní. Bylo otestován také vytváření 50 kampaní pro 10 uživatelů. Poslední test byl takový, který vytvářel 50 kampaní pro 50 uživatelů.

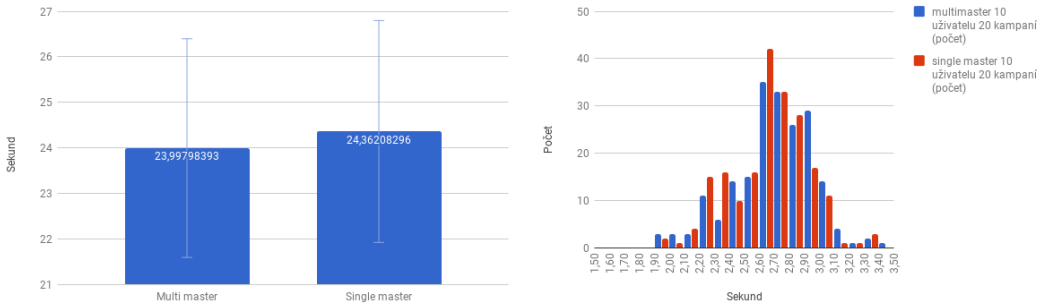
Při těchto testech byla měřena dobo vykonávání dotazů, to tedy znamená, že čím kratší odezva tím lepší výkon. Galera cluster obsahoval tři členy, každá instance byla na jiném serveru. Dále obsahoval dva servery jeden, který měl implementovanou vytvářecí a listovací funkcionalitu a druhý, který prováděl funkci „Shard mastra“. Mezi single a multi master instancí byl přepínán pouze „Shard master“, který vracel pro uživatele buď jednu instanci (single master) nebo všechny tři (multi master).

Obrázek 25: Graf celkové doby vytváření 10 kampaní pro 10 uživatelů a histogram počtu podobných trvání vytvoření jedné entity nebo skupiny entit.



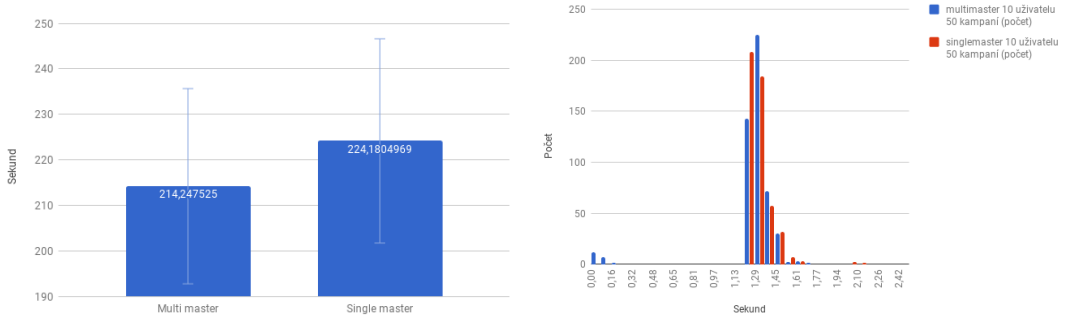
Zdroj: Vlastní zpracování

Obrázek 26 Graf celkové doby vytváření 20 kampaní pro 10 uživatelů a histogram počtu podobných trvání vytvoření jedné entity nebo skupiny entit.



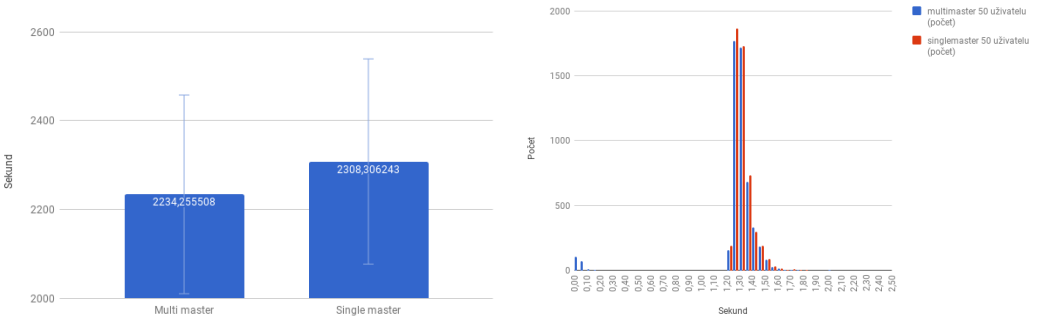
Zdroj: Vlastní zpracování

Obrázek 27: Graf celkové doby vytváření 50 kampaní pro 10 uživatelů a histogram počtu podobných trvání vytvoření jedné entity nebo skupiny entit.



Zdroj: Vlastní zpracování

Obrázek 28: Graf celkové doby vytváření 50 kampaní pro 50 uživatelů a histogram počtu podobných trvání vytvoření jedné entity nebo skupiny entit.



Zdroj: Vlastní zpracování

Pokud se podíváme na tyto grafy vytváření 10, 20, 50 kampaní pro 10 a 50 uživatelů, je vidět, že celkové vytváření účtu je o něco rychlejší při použití multi master řešení. Je zde

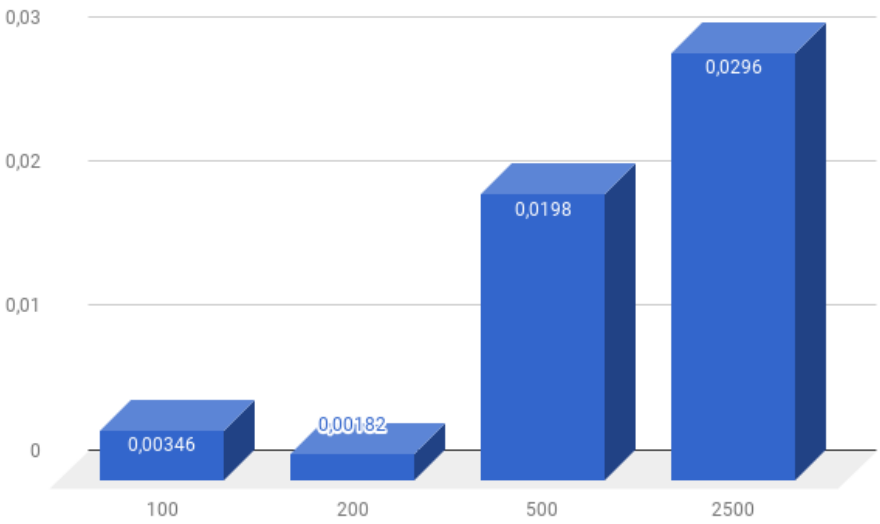
Rozdíl mezi single a multi master řešením se s větším počtem uživatelů a vyšším počtem vytváření entity vzrůstá. Při prvním případu testování (10 kampaní pro 10 uživatelů) je rozdíl mezi single a multi master 0,3460 sekund. Rozdíl u druhého případu je o něco vyšší 0,3640 sekund. Při třetím případu 50 kampaní pro 10 uživatelů byl rozdíl 9,3297 sekund. U posledního testovacího scénáře je to 74,0507 sekund. Tyto testy potvrzují předpoklad, že při vyšším zatížení má vyšší výkon přístup multi master, a tedy neztratíme výkon. V tabulce i na grafu si můžeme povšimnout rostoucího trendu rozdílu mezi řešením single a multi master.

Tabulka 1: Rozdíl a poměr rozdílu testu při single a multi master řešení.

Počet vytvořených kampaní (počet uživatelů * počet kampaní)	Rozdíl doby vytváření (sekund)	Poměr rozdílu a počtem vytvořených kampaní
100	0,346	0,00346
200	0,364	0,00182
500	9,9327	0,0198
2500	74,0507	0,0296

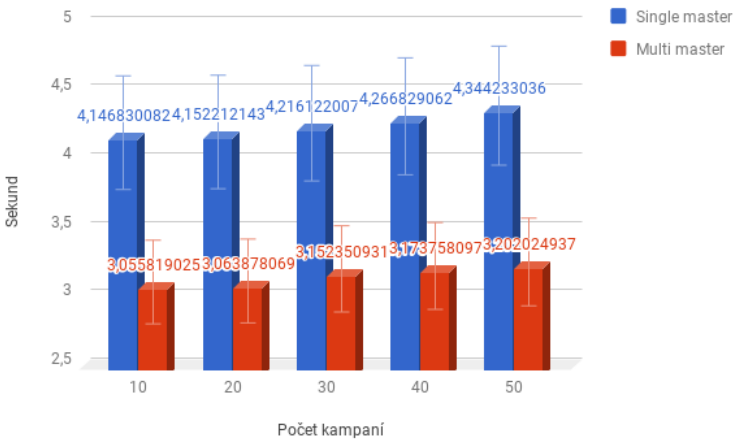
Zdroj: Vlastní zpracování

Obrázek 29: Poměr rozdílu mezi single a multi master řešením.



Zdroj: Vlastní zpracování

Obrázek 30: Porovnání listování kampaní pro 50 uživatelů v single a multi master módu



Zdroj 1: Vlastní zpracování

Tyto testy prokázaly, že řešení splňuje očekávání a výkon při vytváření nebude ztracen. Navíc díky škálování zápisových a čtecích operací je možné využít hardwarových zdrojů slave instancí databázového clusteru, to v single master konfiguraci nebylo možné. Význam využití více zdrojů je významný, až v případě, že master instance databázového serveru nebude stíhat zpracovávat požadavky. Tato situace, díky rozprostření dotazů a transakcí na celý cluster, by v řešení multi master neměla nastat.

Další velkou výhodou je fakt, že řešení multi master může ušetřit firmě Seznam.cz a.s. nemalé peníze na pořizování dalších databázových serverů. Dříve pro navýšení výkonu

systému, bylo pořízeno několik hardwarových serverů. Několik z důvodu master slave architektury, kdy slave instance databázového clusteru byly vytěžovány pouze výdejovými komponentami. Právě z důvodu vytěžování některých slave instancí výdejovými komponentami, bylo nutné implementovat systém, který vyhodnocuje vytíženost databázového serveru a na základě metrik a vah rozhoduje, zda tuto instanci daného clusteru zobrazí pro využití dalším komponentám systému Sklik.cz.

4 Závěr

Spolehlivost služeb je v dnešní době velice důležitým prvkem. Nedostupnost služby způsobená selháním systému může mít okamžitý dopad na zákazníka i na provozovatele dané služby. Data, která jsou uložena v databázových clusterech, jsou pro běh aplikací a služeb ve velké míře kritická. Pokud při uchovávání či manipulaci s těmito daty vznikne nějaká nekonzistence, může zapříčinit velké finanční ztráty. Nehledě na to, že návrat dat do konzistentního stavu může být mnohdy nemožný.

Odvětví informačních technologií je velice dynamické a posouvá se stále vpřed. Díky tomu se rozšiřují možnosti v místech, kdy to ještě před pár lety bylo otázkou vize. Databázové clustery se historicky stavěly do asynchronně replikovaných clusterů, aby bylo možné rozložit čtecí operace. Využívání asynchronních replikací s sebou nese značná rizika, která se při využívání těchto clusterů mohou projevovat.

V současné době existuje možnost stavět databázové clustery do synchronně replikovaných clusterů. Toto řešení přináší několik benefitů, jako je rozložení zátěže i odstranění nemalých problémů, např. vznik nekonzistence dat, vyskytující se u asynchronně replikovaných clusterů. Synchronně replikované clustery přináší další možnosti, jak škálovat databáze.

Cílem práce bylo nastudování a prozkoumání domény databázových clusterů. Dále popsat, jaké problémy se mohou vyskytovat při stavbě klasických asynchronních clusterů, a jaké naopak při využití synchronních replikací. Dalším důležitým tématem bylo prozkoumání techniky škálování uložení dat, jako je vertikální škálování a horizontální škálování. V neposlední řadě bylo nutné tyto nabité vědomosti aplikovat, analyzovat a navrhnout možná zlepšení pro systém Sklik.cz.

Práce byla zaměřena převážně na porovnání synchronních a asynchronních řešení replikací v databázových clusterech, následně na možnosti stavění databázových clusterů, analyzování problémů, které různé techniky přináší a možnosti řešení těchto problémů. Virtuálně synchronní replikace přináší převratné řešení zpracování transakcí a přístup ke stavovému automatu. Toto řešení eliminovalo problémy zpomalení, které bylo problémem u starších metod založených na metodě distribuovaných zámku atp. Toto řešení velice zjednodušilo práci s databázovým clusterem a také umožnilo eliminovat problém nekonzistence dat.

V práci je popsána implementace několika řešení moderních synchronních replikací, která jsou zde porovnána. Dále zde bylo popsáno několik možností, do jakých architektur lze databázové clustery stavět. Následně bylo představeno několik technik, popisujících, jakým způsobem lze data škálovat horizontálně pomocí techniky rozdělení dat, techniky škálování dat nebo jejich kombinací. Byla zde popsána úskalí, které různé techniky mají možná řešení.

V případové studii byl porovnán virtuálně synchronní cluster s asynchronním. Důležitým úkolem bylo analyzovat stávající řešení a navrhnout možnosti zlepšení. Po analýze bylo navrženo několik možností, jak přizpůsobit server, který obstarává škálování dat, aby využil možnosti, které má synchronně replikovaný cluster. Tato řešení byla popsána pseudokódem. Vybraný postup využití virtuálních uzlů byl implementován v jazyce C++. Dále pak bylo zvolené řešení otestováno na abstraktních příkladech provozu nad systémem Sklik.cz. Testy potvrdily, že systém neztratí na výkonu, dokonce při vyšším zatížení může být využito benefitu rozložení zátěže na více strojů. Tento benefit umožní také ušetření finančních prostředků na nákup nových serverů, jelikož rozložení zátěže dovolí využít hardwarové zdroje, která byly využity pouze komponentami, které neměly problém s latencí asynchronní replikace. Bylo zde také navrženo několik technik, jak kontrolovat stav synchronního clusteru a možnosti omezení velikosti viditelného clusteru pro server na základě vytíženosti databázového systému. V neposlední řadě zde byla nastíněna externí řešení, která je možné použít pro škálování a rozkládání zátěže nad databázovým clusterem.

Na tuto práci by bylo možné navázat analyzováním a převedením databáze, která spravuje unikátní identifikátory a má ji pod správou pouze server („Shard master“), který umožňuje škálovat data a rozkládat zátěž. Tato databáze stále využívá techniky asynchronních replikací a je jedním z nejvíce kritických bodů, který způsobuje nekonzistenci dat, tzv. splitbrain. Existuje několik možných způsobů, jak tuto databázi optimalizovat, aby přechod z asynchronních replikací na synchronní neubral na výkonu. Například rozdělit databázi na dvě či více databází, aby bylo jasné, kde jsou číselníky a kde jsou uchovány identifikátory. Další možností je převést správu identifikátoru na bitová pole, která by umožnila zrychlení certifikace při virtuálně synchronních replikacích a i alokace při větším počtu identifikátorů za cenu menší režie na aplikační vrstvě.

Další možnost pokračování by mohlo být prozkoumání a navržení možnosti automatického škálování databáze napříč cloudovým řešením. Prozkoumat, zda by tato možnost neumožnila firmě ušetřit finance za současné řešení, kdy je nutné spravovat několik desítek serverů, a to i v případě malé zátěže. Tohle řešení by mohlo být využito i v případě krizových situací, kdy by bylo zcela nedostupné jedno datové centrum.

5 Seznam použitých zdrojů

[1] J. &. V. M. Pokorný, Databázové systémy., České vysoké učení technické. editor, Praha, 2013.

[2] B. Wilder, Cloud Architecture Patterns, O'Reilly Media, 2012.

[3] B. Erb, „Concurrent Programming for Scalable Web Architectures," 2012. [Online]. Available: http://www.benjamin-erb.de/ thesis.

[4] J. D. a. T. K. George F Coulouris, Distributed systems: concepts and design. pearson education, 2005.

[5] V. T. B. S. P. Zaitsev., High Performance MySQL, 3rd Edition, O'Reilly Media, 2012.

[6] C. Ltd, „Galera cluster documentation," [Online]. Available: http://galeracluster.com/.

[7] P. LLC, „Percona Server 5.6 - Documentation," [Online].

[8] F. Pedone, „The database state machine and group communication issues," 1999.

[9] J. Lechtenbörger, „Two-phase commit protocol," v *Encyclopedia of Database Systems*, Springer, 2009.

[10] O. Corporation, „MySQL 5.6 Reference Manual," [Online]. Available: https://dev.mysql.com/doc/refman/5.6/en.

[11] „ProxySQL Tutorial - Database Load Balancing Tutorial for MySQL & MariaDB with ProxySQL," serverlnines, [Online]. Available: https://severalnines.com/resources/tutorials/proxysql-tutorial-mysql-mariadb.

[12] J. E. Boritz, IS practitioners' views on core concepts of information integrity, n: International Journal of Accounting Information Systems, 2005.

[13] P. J., Database Architectures: Current Trends and their Relationships to Requirements of Practice , in Advances in Information Systems Development, Springer Science+Business Media, 2007, pp. 267-277.

[14] L. Lamport, „Paxos Made Simple," 1 11 2001. [Online]. Available: https://lamport.azurewebsites.net/pubs/paxos-simple.pdf.

[15] A. Nair, „Oracle Database 12c Release 2, Oracle Real Application Clusters", 2017. [Online]. Available: http://www.oracle.com/technetwork/database/options/clustering/rac-wp-12c-1896129.pdf

[16] B. Oles, „Comparing Oracle RAC HA Solution to Galera Cluster for MySQL or MariaDB", 2018. [Online]. Available: https://severalnines.com/blog/comparing-oracle-rac-ha-solution-galera-cluster-mysql-or-mariadb

6 Přílohy

6.1 Pseudokód tříd zaštiťující datový cluster

```
class GaleraStatus:
    # atributy
    statuses = {"OK": 1, "NOT_RESPONDING": 2,
        "NOT_READY": 3,"NOT_CONNECTED": 4,
        "NOT_SYNCED": 5, "UNKNOWN": 6}
    member_status = 0
    # metody
    # přidá status do statusů
    def add_status(self, status): ...
    # Navrátí aktuální status.
    def get_member_status(self): ...
    # Vrátí řetězec, který obsahuje diagnostiku.
    def get_diagnostics(self): ...
    # Zjistí v jakém stavu je dany člen vrací bool hodnotu.
    def is_member_ok(self): ...

class GaleraConnection:
    # atributy
    connection = Connection(...)
    status = GaleraStatus()
    nickname = "nickname"
    configuration = Config(...)
    # Metoda umožní vylistovat všechny proměnné,
    # které se týkají Galera clusteru tedy wsrep_*.
    def get_node_global_wsrep_status(self): ...
    # Metoda pro získání členů clusteru z proměnné
    # wsrep_incoming_addresses.
    def get_cluster_members(self): ...
    # Vrací konfiguraci pro dané připojení.
    def get_configuration(self): ...
    # Zkontroluje zvolené parametry wsrep_local_state,
    # wsrep_connected, wsrep_ready.
    def complete_check(self): ...
    # Tahle metoda je tzv. fallback na GaleraStatus
    # tedy volaní metody GaleraStatus.is_member_ok,
    # vrací tedy bool hodnotu.
    def is_member_ok(self): ...
    # Kontrola, zda se nejedna o stejné připojení
    def same_connection(self, configuration ->Config): ...

class GaleraCluster:
    # atributy
    connection = GaleraConnection(...)
```

```
members = [GaleraConnection(...)]
# Konfigurovatelná hodnota, která udává kolik
# členů clusteru bude využito jako multimaster.
galera_ratio = "all" # none, half, quarter, one
decision = "normal" # auto_optimalize
# Inicializace clusteru.
def init_cluster(self):
    # Získáme pomocí připojení do MASTER instance
    # informace o clusteru.
    members_hosts = self.connection.get_cluster_members()
    # Získáme pole ip(host):port.
    hosts = members_hosts.splite(",")
    # Pokud nemáme žádné adresy vracíme se není
     # zde co dělat.
    if not hosts:
        return
    selected_hosts = []
    # Pokud jsme získali adresy musím rozhodnout
    # kolik adres vlastně použijeme, a to
    # na základě konfigurace.
    if self.galera_ratio == "all":
        selected_hosts = hosts
    elif self.galera_ratio == "half":
        selected_hosts = hosts[:(len(hosts)/2)]
    elif self.galera_ratio == "quarter":
        selected_hosts = hosts[:(len(hosts) / 4)]
    elif self.galera_ratio == "one":
        selected_hosts =  [hosts[0]]

    for host_and_port in selected_hosts:
       # Nejsou specifikováni žádní členové
       # clusteru pokračuji.
        if host_and_port == "unspecified":
            continue
        # Získáme host a port.
        host, port = host_and_port.split(":")
        # Kopie konfigurací.
        configuration = copy(
               self.connection.get_configuration())
        # Nastavíme nový host a port.
        configuration.host = host
        configuration.port = port
        # Kontrola stejných připojení.
        self.connection.same_connection(configuration)
        # Přidání připojení Galera clusteru
      # do listu/vectoru/pole.
        self.members.append(
               GaleraConnection(configuration))
# Získání clusteru.
def get_cluster(self):
   # Musíme si vytvoří novou instanci
```

```
        # listu/vektoru/pole, jelikož chceme připojení,
        # která jsou v pořádku.
        result_cluster = [
              galera_connection
                for galera_connection in self.members
                  if galera_connection.is_member_ok()]
        return result_cluster

    # Získá velikost clusteru, je spočítaná způsobem,
    # kdy se porovnává jestli
    # je připojeni (GaleraConnection) v pořádku, tedy pokud
    # is_member_ok vrátí True.
    def get_cluster_size(self): ...

    # Zkontroluje se cely cluster příslušné
    # se nastaví hodnoty.
    def check(self): ...

    # Zkontroluje, zda cluster, resp. proměnná
    # members obsahuje členy clusteru.
    def has_members(self): ...

class DataCluster:
    # atributy
    cluster = {shard_id: GaleraCluster}
    # Přidání nového nodu, kontrola stavů a vlastností.
    def append_member(self, member -> GaleraCluster): ...
    # Získá index virtuálního shardu na
    # základě identifikátoru uživatele.
    def get_index_by_user_id(
             self, key -> Key, user_id -> int) -> int:
        # Získáme shard podle klíče.
        shard = self.get_member(key)
        # Pokud člen není definován vracíme 0, což znamená
        # použití master instanci daného uzlu.
        if not shard:
            return 0
        # získáme velikost clusteru, ta záleží
         # na parametrech, které jsou nakonfigurované
        # viz GaleraCluster.init_cluster.
        size = shard.get_cluster_size()
        # Pokud je velikost 0, což znamená použití
        # master instance.
        if not size:
            return 0
        # Na základě modula získáme vždy, pokud se nezmění
        # velikost clusteru, stejné připojení.
        id = user_id % size
        return id + 1

    # Získáme uzel, a tím i všechny členy Galera clusteru,
```

```
    # pokud splňuje podmínky toho, že jeho stav je v pořádku.
    def get_member(Key key) -> GaleraCluster: ...
```

6.2 Konfigurace databázových serverů při testu

6.2.1 Nastavení Docker Image

Byl vytvořen Docker image, pro Galera cluster instanci, za pomoci již existujícího image pro MariaDB.

```
FROM mariadb:10.2

RUN apt-get update && apt-get install -y galera-arbitrator-3
ssh rsync lsof && \
    rm -rf /var/lib/apt/lists/*

COPY galera-init.sh /

EXPOSE 3306 4444 4567 4567/udp 4568

ENTRYPOINT ["/galera-init.sh"]
CMD ["mysqld"]
```

V popisu vytváření je vidět použití shell skriptu, tento skript vytváří pouze základní konfiguraci pro Galera cluster.

```
#!/usr/bin/env bash

[ "$DEBUG" == 'true' ] && set -x

set -eo pipefail

if [ -n "$WSREP_CLUSTER_ADDRESS" -a "$1" == 'mysqld' ]; then

    echo '>> Creating Galera Config'
    export MYSQL_INITDB_SKIP_TZINFO="yes"
    export MYSQL_ALLOW_EMPTY_PASSWORD="yes"

    echo """
    # Galera Cluster Auto Generated Config
    [galera]
    wsrep_on="on"
    wsrep_provider="${WSREP_PROVIDER:-
/usr/lib/libgalera_smm.so}"
    wsrep_provider_options="${WSREP_PROVIDER_OPTIONS}"
    wsrep_cluster_address="${WSREP_CLUSTER_ADDRESS}"
```

```
    wsrep_cluster_name="${WSREP_CLUSTER_NAME:-
my_wsrep_cluster}"
    wsrep_node_name="${WSREP_NODE_NAME:-$(hostname -s)}"
    wsrep_sst_auth="${WSREP_SST_AUTH}"
    wsrep_sst_method="${WSREP_SST_METHOD:-rsync}"
    [mysqld]
    default_storage_engine = innodb
    binlog_format = row
    """ >  /etc/mysql/conf.d/galera-auto-generated.cnf

    if [ -n "$WSREP_NODE_ADDRESS" ]; then
            echo wsrep_node_address="${WSREP_NODE_ADDRESS}" >>
/etc/mysql/conf.d/galera-auto-generated.cnf
    fi

elif [ -n "$WSREP_CLUSTER_ADDRESS" -a "$1" == 'garbd' ]; then

    echo '>> Configuring Garbd'
    set -- "$@" --address=$WSREP_CLUSTER_ADDRESS --
group=${WSREP_CLUSTER_NAME:-my_wsrep_cluster} --
name=$(hostname)

fi

exec /docker-entrypoint.sh "$@"
```

Pro chod Docker Image, byl použit nástroj *docker-compose,* který umožňuje orchestraci těchto instancí.

```
db-master:
  image: mariadb
  volumes:
    - "/tmp/masterdb:/var/lib/mysql:rw"
    - "./my_master.cnf:/etc/mysql/conf.d/00-my.cnf:ro"
  environment:
    - "MYSQL_DATABASE=test"
    - "MYSQL_USER=user"
    - "MYSQL_PASSWORD=passwd"
    - "MYSQL_ROOT_PASSWORD=rootpasswd"
    - ""
  ports:
    - "3306:3306" #Povolení portu pro MariaDB/MySQL.


db-galera-node:
  image: galera-node
  environment:
   - "MYSQL_DATABASE=test"
   - "MYSQL_USER=user"
   - "MYSQL_PASSWORD=passwd"
```

```
  - "MYSQL_ROOT_PASSWORD=rootpasswd"
  - "WSREP_CLUSTER_NAME=node_gn_test_cluster"
  - "WSREP_NODE_NAME=galera-node-gn"
  - "WSREP_NODE_ADDRESS=0.0.0.0"
  - "WSREP_CLUSTER_ADDRESS=gcomm://0.0.0.0"
 volumes:
  - "./my_galera.cnf:/etc/mysql/conf.d/00-my.cnf:ro"
  - "/tmp/masterdb:/var/lib/mysql:rw"
 ports:
  - "3306:3306" # Povolení portu pro MariaDB/MySQL a také
                # SST, pokud se použije mysqldump metoda.
  - "4567:4567" # Povolení portu pro wsrep replikace.
  - "4444:4444" # Povolení portu pro SST, pro ostatní metody.
  - "4568:4568" # Povolení portu pro IST
```

6.2.2 Konfigurace nástroje sysbench

Nástroj sysbench byl spuštěn s touto konfigurací. [14]

`sysbench --db-driver=mysql --mysql-host=10.1.18.125 --mysql-port=3306 --mysql-user=sklik --oltp-table-size=20000 --oltp-tables-count=10 --oltp-test-mode=complex --threads=$thread --mysql-user=user --mysql-password=pass`

`--oltp-table-size=20000` znamená kolik záznamů (řádků) bude vytvořeno.

`--oltp-tables-count=10` kolik tabulek bude vytvořeno.

`--oltp-test-mode=complex` mód, který bude použit pro oltp. Byly použity složitější transakce.

`-- threads=$thread` počet vláken neboli konkurence.

6.2.3 Konfigurace Galera clusteru

Konfigurace byla uložena v *my.cnf*, odkud si databázový systém při startu získává nastavení svých globální proměnných. [6]

```
[galera]
wsrep_retry_autocommit = 5 # počet opakování při kolizi
(„first commit wins“)
wsrep_causal_reads = ON # zapnutí/vypnutí kauzálního čtení
wsrep_slave_threads = 8 # počet vláken pro replikaci
# nastavení gcache a váhy člena clusteru
wsrep_provider_options =
"gcache.size=1GB;gcache.recover=yes;pc.weight=1"

[mysqld]
innodb_buffer_pool_size = 1500M # Velikost buffer poolu.
```

```
innodb_buffer_pool_instances = 2 # Kolik instancí bude
spravovat buffer pool.
innodb_flush_log_at_trx_commit = 2 # Zápis do binárního logu
po potvrzení transakce, flush 1 sekundě.

# Konfigurace jména, velikosti a atributů InnoDB prostoru pro
datové soubory.
innodb_data_file_path = ibdata1:10M:autoextend:max:1000M
# Počet spuštěných vláken naráz, které mohou zapisovat.
innodb_write_io_threads = 8
# Počet naráz spuštěných vláken, které čtou.
innodb_read_io_threads = 8
# Počet konkurenčních vláken nad InnoDB soubory.
innodb_thread_concurrency = 16
# Maximální počet spojení.
max_connections = 10000

# Nastavení bufferů.
sort_buffer_size = 64M
read_buffer_size = 64M
join_buffer_size = 64M
# Nastavení cache pro každé vlákno.
thread_cache_size = 32
# Neukládáme slow_query_log.
slow_query_log = 0
# Kódování znaků.
collation-server=utf8_general_ci
character-set-server=utf8
# Timeout pro zámek
innodb_lock_wait_timeout = 120
# Uložení InnoDB souborů, každá tabulka má vlastní soubor.
innodb_file_per_table = 1
innodb_file_format = barracuda
```

6.2.4 Konfigurace MariaDB asynchronní replikace

```
[mysqld]
innodb_buffer_pool_size = 1500M # Velikost buffer poolu.
innodb_buffer_pool_instances = 2 # Kolik instancí bude
spravovat buffer pool.
innodb_flush_log_at_trx_commit = 2 # Zápis do binárního logu
po 1 sekundě.

# Konfigurace jména, velikosti a atributů InnoDB prostoru pro
datové soubory.
innodb_data_file_path = ibdata1:10M:autoextend:max:1000M
# Počet spuštěných vláken naráz, které mohou zapisovat.
innodb_write_io_threads = 8
# Počet naráz spuštěných vláken, které čtou.
innodb_read_io_threads = 8
# Počet konkurenčních vláken nad InnoDB soubory.
```

```
innodb_thread_concurrency = 16
# Maximální počet spojení.
max_connections = 10000

# Nastavení bufferů.
sort_buffer_size = 64M
read_buffer_size = 64M
join_buffer_size = 64M
# Nastavení cache pro každé vlákno.
thread_cache_size = 32
# Neukládáme slow_query_log.
slow_query_log = 0
# Kódování znaků.
collation-server=utf8_general_ci
character-set-server=utf8
# Timeout pro zámek
innodb_lock_wait_timeout = 120
# Uložení InnoDB souborů, každá tabulka má vlastní soubor.
innodb_file_per_table = 1
innodb_file_format = barracuda
```

6.3 Skript pro simulaci vytváření a listování entit

Skript pro simulaci zjednodušeného chování uživatele. Je vytvořena kampaň, každá kampaň obsahuje několik sestav a každá sestava zaštiťuje několik inzerátů. Tento skript simuluje přístupy pro čtení, zápis nebo jejich kombinaci.

```
try:
   from fastrpc import ServerProxy
except:
   from xmlrpclib import ServerProxy
import time
import multiprocessing
import os


def build_campaigns(proxy, user_id, count=5, offset=0):
   print("Building campaign for  user={},
count={}".format(user_id, count))
   campaigns = [{"name":"kampane test RW {}".format(i+offset),
"dayBudget": 400} for i in range(count)]
   print(campaigns)
   start = time.time()
   result = proxy.campaigns.create(user_id, user_id,
campaigns)
   print("Building campaign took:<{}>
[{}]".format(time.time()-start, os.getpid()))
   if result["status"] != 200:
      print("ERROR: params={}, result={}".format(campaigns,
```

```
result))
        return []
    print("Kampane: {}".format(result))
    return result["campaignIds"]


def build_groups(proxy, user_id, campaigns, count=5,
offset=0):
    print("Building group for  user={}, campaigns={},
count={}".format(user_id, campaigns, count))
    groups = [{"campaignId": c, "name": "Sestava
{}".format(i+offset), "cpc": 500.0} for c in campaigns for i
in range(count)]
    print("ATTRS: ", groups)
    start = time.time()
    result = proxy.groups.create(user_id, user_id, groups)
    print("Building groups took:<{}> [{}]".format(time.time()-
start, os.getpid()))
    if result["status"] != 200:
        print("ERROR: params={}, result={}".format(groups,
result))
        return []
    return result["groupIds"]


def build_ads(proxy, user_id, groups, count=5, offset=0):
    print("Building ads for  user={}, groups={},
count={}".format(user_id, groups, count))
    ads = [{"groupId": g,
            "headline1": "testovaci {}".format(i+offset),
            "headline2": "headline 2 {}".format(i+offset),
            "description": "description {}".format(i+offset),
            "finalUrl": "https://seznam.cz",
            "adType": "eta"
            } for g in groups for i in range(count)]
    start = time.time()
    result = proxy.ads.create(user_id, user_id, ads)
    print("Building ads took:<{}> [{}]".format(time.time()-
start, os.getpid()))
    if result["status"] != 200:
        print("ERROR: params={}, result={}".format(ads, result))
        return []
    return result

def build_ads2(proxy, user_id, groups, count=5, offset=0):
    print("Building ads for  user={}, groups={},
count={}".format(user_id, groups, count))
    ads = [{"groupId": g,
            "headline1": "headliny1 {}".format(i+offset),
            "headline2": "headliny2 {}".format(i+offset),
            "description": "description {}".format(i+offset),
```

```
            "finalUrl": "https://seznam.cz",
            "adType": "eta"
            } for g in groups for i in range(count)]
    start = time.time()
    result = proxy.ads.create(user_id, user_id, ads)
    print("Building ads took:<{}> [{}]".format(time.time()-
start, os.getpid()))
    if result["status"] != 200:
        print("ERROR: params={}, result={}".format(ads, result))
        return []
    return result


def list_campaigns(proxy, user_id):
    start = time.time()
    result = proxy.campaigns.list(user_id, user_id)
    print("Listing campaigns took:<{}>
[{}]".format(time.time()-start, os.getpid()))
    cids = [r["id"] for r in result["campaignList"]]
    return cids

def list_groups(proxy, user_id, campaigns):
    start = time.time()
    result = proxy.groups.list(user_id, user_id,
{"campaignIds": campaigns})
    print("Listing groups took:<{}> [{}]".format(time.time()-
start, os.getpid()))
    return [g["id"] for g in result["groups"]]

def list_ads(proxy, user_id, groups):
    start = time.time()
    result = proxy.ads.list(user_id, user_id, {"groupIds":
groups})
    print("Listing ads took:<{}> [{}]".format(time.time()-
start, os.getpid()))
    return result["ids"]

def build_acc(args):
    user_id, offset = args
    proxy = ServerProxy("http://mainserver:port/RPC2")
    start = time.time()
    campaigns = build_campaigns(proxy, user_id, offset=offset)
    groups = build_groups(proxy, user_id, campaigns)
    ads = build_ads(proxy, user_id, groups)
    ads = build_ads2(proxy, user_id, groups)
    print("Building whole took:<{}> [{}]".format(time.time()-
start, os.getpid()))

def list_accout(args):
    user_id, offset = args
    proxy = ServerProxy("http://mainserver:port/RPC2")
```

```
    start = time.time()
    campaigns = list_campaigns(proxy, user_id)
    groups = list_groups(proxy, user_id, campaigns)
    ads = list_ads(proxy, user_id, groups)
    print("Listing whole took:<{}> [{}]".format(time.time()-
start, os.getpid()))


def multiproces(user_ids, multi=True, write=True, read=False):
    pool =
multiprocessing.Pool(processes=multiprocessing.cpu_count())
    start = time.time()
    if multi:
        if write:
            rw = pool.map_async(build_account, user_ids)
        if read and user_ids_list:
            rl = pool.map_async(list_account, user_ids)
        if write:
            rw.wait()
        if read:
            rl.wait()
    else:
        for item in user_ids:
            if write:
                build_account(item)
            if read and user_ids_list:
                list_account(item)
    print("WHOLE EXEC took:<{}> [{}]".format(time.time()-
start, os.getpid()))
```