Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta

Distribuované databázové systémy

Diplomová práce

Bc. Jakub Geyer

Vedoucí práce: Mgr. Miloš Prokýšek, Ph.D.

České Budějovice 2015

Geyer, J., 2015: Distribuované databázové systémy.
[Distributed database systems. Mas. Thesis, in Czech.] – 79 p.,
Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Tato diplomová práce se snaží přispět k problematice distribuce velkých dat na databázové úrovni. Jednotlivé způsoby distribuce dat jsou zde analyzovány zejména s ohledem na možné decentralizované využití. Získané poznatky jsou následně použity při tvorbě konceptu systému pro ukládání a sdílení biologických vzorků vzniklých měřením na hmotnostním spektrometru.

Abstract

This thesis has for its aim to contribute to the problematics of big data distribution on the database level. Particular means of data distribution are analysed, especially in regard to its possible decentralised usage. Acquired findings are consequently used for the creation of the database system concept that enables saving and sharing of biological samples created by measuring with the aid of mass spectrometer.

Poděkování

Na prvním místě bych rád velice poděkoval vedoucímu mé diplomové práce panu Mgr. Miloši Prokýškovi, Ph.D. za cenné rady a čas strávený při konzultacích.

Rovněž bych rád poděkoval společnosti ČD Cargo, a. s. za zapůjčení hardwaru na testování a dále pak panu Ing. Janu Feslovi za odborné konzultace. Rovněž děkuji všem učitelům a kolegům z Ústavu Aplikované informatiky a v neposlední radě své rodině za podporu v průběhu studia i při psaní diplomové práce.

Prohlášení

Prohlašuji, že svoji diplomovou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své diplomové práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

České Budějovice, 24. 04. 2015

Jakub Geyer

Obsah

1. Úvod

Vývoj databázových systémů a způsobů ukládání dat obecně, je významným, dynamicky se rozvíjejícím odvětvím informatiky. Vznikají nové struktury a způsoby ukládání dat, jejichž cílem je umožnit uchovávat a zpracovávat velké množství dat (tzv. „big data“). [1] [2]

Kdysi téměř výhradně používané relační databáze [3] se stále častěji ukazují jako nevhodné pro práci s některými typy dat, proto vnikají databázové systémy založené na jiných principech a způsobech práce s daty. V posledních 15 letech jsou stále častěji nasazovány databáze, dnes označované jako NoSQL [4], bez pevné vnitřní struktury (schema-less), především s ohledem na flexibilitu obsahu a snadnou škálovatelnost. [5] [6]

Právě škálovatelnost dat (Scaling) nabývá na významu při práci s databázemi [7], neboť uchovávání velkých dat na jednom fyzickém stroji se ukazuje nejen jako finančně náročnou, ale v případě exponenciálně narůstajícího množství dat a nároků na výkon prakticky nerealizovatelnou variantou.

Ke slovu tak přicházejí distribuovaná centralizovaná řešení v podobě rozsáhlých diskových polí, clusterů a cloudů. Zde je však nutné brát v úvahu problém distribuce a práce s daty. Podle Brewerova CAP theoremu (Brewer’s Theorem) [8] je totiž pro distribuovaný systém (a potažmo tedy i pro distribuovanou databázi) možné splňovat pouze 2 ze 3 kritérií: konzistence, dostupnost, odolnost vůči chybám sítě (partition tolerance).

Na moderní databáze je tak kladeno množství různých požadavků, kromě škálovatelnosti například práce s binárními daty – soubory, podpora transakcí, možnost nasazení na cloudu, podpora pro BI (Business Intelligence) apod. Jen těžko si lze představit databázový model, který by dokázal vyhovět všem požadavkům (často i protichůdným) a současně ještě poskytoval dostatečný výkon. Vznikají různé nové databázové platformy, ať už se jedná o klasický relační model, rozšířený NewSQL [9], objektově-orientované databáze, NoSQL a další.

Jednotlivé nové i existující databázové platformy se snaží specializovat na podporu konkrétních požadavků, a volba vhodného řešení je tak pro každý projekt vyžadující databázi klíčová.

1.1. Formulace problému

Představme si, že chceme vytvořit systém pro sdílení dat založených na databázi, která nám podle specifikace vrátí požadovaná data (soubory v binární podobě). Tento systém může sloužit pro sdílení souborů různých specifických typů, například dokumentů, hudebních souborů, nebo unikátních dat pro potřeby konkrétní aplikace.

Při velkém množství dat by pro takový systém bylo jistě vhodné distribuované řešení, především z důvodů rozložení dat, konektivity, a v případě složitějších dotazů i rozložení výpočetní zátěže.

Představme si rovněž, že nechceme spoléhat na centralizované řešení v podobě datového centra či cloudu, ale chceme vytvořit decentralizovaný systém založený na lokálních uzlech s databázemi, do kterých budou uživatelé data ukládat a nad nimiž, včetně jejich sdílení, budou mít přímou kontrolu. Počet těchto uzlů bude přitom možné flexibilně měnit.

Ačkoliv některé databázové platformy (především z řad NoSQL a NewSQL) podporují možnost distribuovaných řešení, jen málo z nich počítá s možností decentralizace. I v případě podpory decentralizovaného nasazení se navíc jedná převážně o realizace geografického clusteru, s předem známým počtem, umístěním serverů, a pravidly pro distribuci dat.

1.2. Výzkumné otázky

Je možné realizovat decentralizovaný systém pro sdílení dat založený na lokálních databázích s možností flexibilního počtu serverů?

Dílčí otázky:

1) Jaké jsou existující způsoby distribuovatelnosti databází?
2) Jaké jsou výhody a nevýhody proti centralizovanému řešení?
3) Jak realizovat distribuovanou databázi s flexibilním počtem úložišť?

1.3. Cíl práce

Hlavním cílem práce je vytvoření funkčního konceptu decentralizovaného systému pro ukládání a sdílení dat. Systém bude tvořen lokálními databázemi s možností změny počtu těchto databází a klientskou aplikací umožňující vyhledávání dat.

K naplnění hlavního cíle je rovněž nutné splnit následující dílčí cíle a úkoly:

1) Identifikovat současné způsoby distribuovatelnosti různých databázových modelů.
2) Nalézt vhodné kandidáty (existující databázové platformy), které je možné použít s ohledem na distribuovatelnost, flexibilitu a výkon při velkém množství dat.
3) Popsat teoretický model pro sdílení dat v distribuovaném systému s nestálým počtem databází.
4) Ověřit model distribuovaného systému při řešení konkrétního problému (sdílení a prohledávání dat vzniklých měřením na hmotnostním spektrometru)

1.4. Použité nástroje a metody

Při plnění úkolů je nejdříve využito teoretických metod. Jedná se především o studium odborné literatury, odborných diskuzí a analýzy dostupných databázových produktů.

Teoretické poznatky budou poté využity k provedení experimentu. Vytvořený modelový systém pro ukládání a sdílení biologických vzorků bude otestován z hlediska výkonu při dotazování, s ohledem na různé množství dat a počet úložišť.

2. Škálovatelnost (Scaling)

Škálovatelnost značí schopnost systému reagovat na zvyšující se nároky (zvyšující se množství vstupů/výstupů, počet dotazů apod.). Se škálovatelností často souvisí i paralelizace jednotlivých operací za účelem zvýšení výkonu a snížení doby potřebné k provedení úkonů. [7]

2.1. Vertikální škálovatelnost

Vertikální škálovatelností se rozumí hardwarové zlepšení/rozšíření současného systému (typicky jednoho počítače). V praxi se jedná především o přidávání nebo výměnu za výkonnější komponenty (procesory, paměti, disky). Paralelizace může být do vertikálního škálování zahrnuta na úrovni jednotlivých hardwarových komponent (např. v podobě vícejádrových procesorů, diskových polí, apod.) za předpokladu, že je s nimi v systému počítáno.

Tento druh škálovatelnosti obvykle nelze uplatnit na dynamicky se rozvíjející systémy, s předpokládaným nárůstem objemu dat či požadavků na výkon. To je dáno fyzickým omezením každého systému.

I kdyby teoreticky existoval systém, který by bylo možné donekonečna hardwarově rozšiřovat (např. přidávat další a další procesory), takový systém by byl velmi náročný na údržbu, náchylný k chybám, a bylo by velmi obtížné integrovat nové technologie. [8]

2.2. Horizontální škálovatelnost

Horizontální škálovatelností se rozumí přidávání celých nových systémových prvků (např. dalších počítačů), které převezmou část činnosti současného systému nebo vykonávají jinou návaznou činnost. V praxi se přistupuje k horizontálnímu škálování zejména v případě, že vertikální již není dále možné, nebo je finančně příliš nákladné. [8] [9]

Ani při horizontálním škálování nemusí docházet k paralelizaci – každá část systému může provádět závislou činnost, která je však odlišná od činnosti jiné části systému (princip více závislých front). U dynamicky se rozrůstajících systémů se však dříve nebo později paralelizaci některých činností vyhnout nelze, jinak by docházelo k nárůstu doby potřebné k vyřízení požadavků. Vzniká zde však nový problém se způsobem rozložení a distribuce dat.

2.3. Vertikální nebo horizontální škálování

Vztah mezi očekáváným a skutečným zvýšením výkonu při paralelizaci se řídí především Amdahlovým zákonem [12]. Z něj je možné vyvodit, že v žádném systému není možné paralelizovat všechny operace, vyhnout se vzájemným čekáním paralelních procesů, a zajistit nulovou režii (obslužné procesy apod.). Teoretický výkon systému při paralelizaci části výpočtů je tedy možné vyjádřit jako:

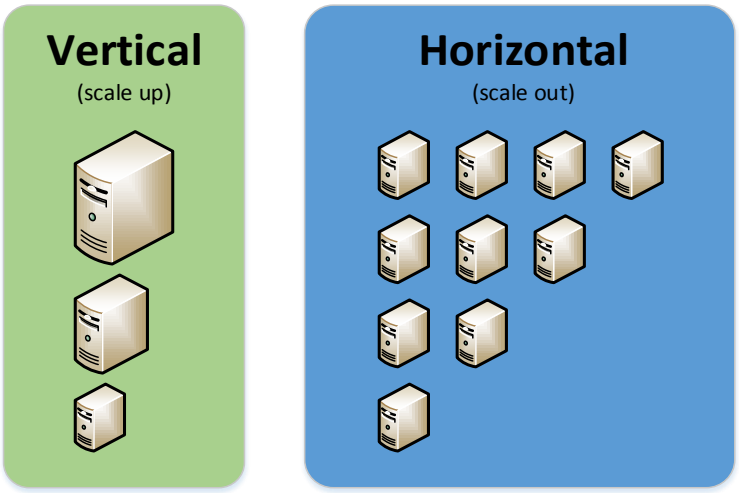
$$V_p = \frac{1}{S + \frac{1 - S + L}{P}} - R$$

kde V_p značí teoretický výkon, S je část vyžadující sekvenční zpracování a P je počet paralelních procesů. Proměnná L představuje nutnost čekání procesů a proměnná R režii vniklou například přidělováním dat paralelním procesům a shromažďováním výsledných výpočtů.

Pokud je tedy například 60 % činnosti systému možné paralelizovat a v rámci jednoho počítače zvažujeme výměnu 4-jádrového procesoru za 8-jádrový, maximální zvýšení výkonu (i bez možného vzájemného čekání a režie) bude pouze necelých 16 %.

$$V_4 = 1{,}818 \qquad V_8 = 2{,}105 \qquad S = \frac{V_8}{V_4} = 1{,}158$$

Navyšování hardwarového výkonu zde tedy nemá velký vliv, a je tak vhodné uvažovat o horizontálním škálování.



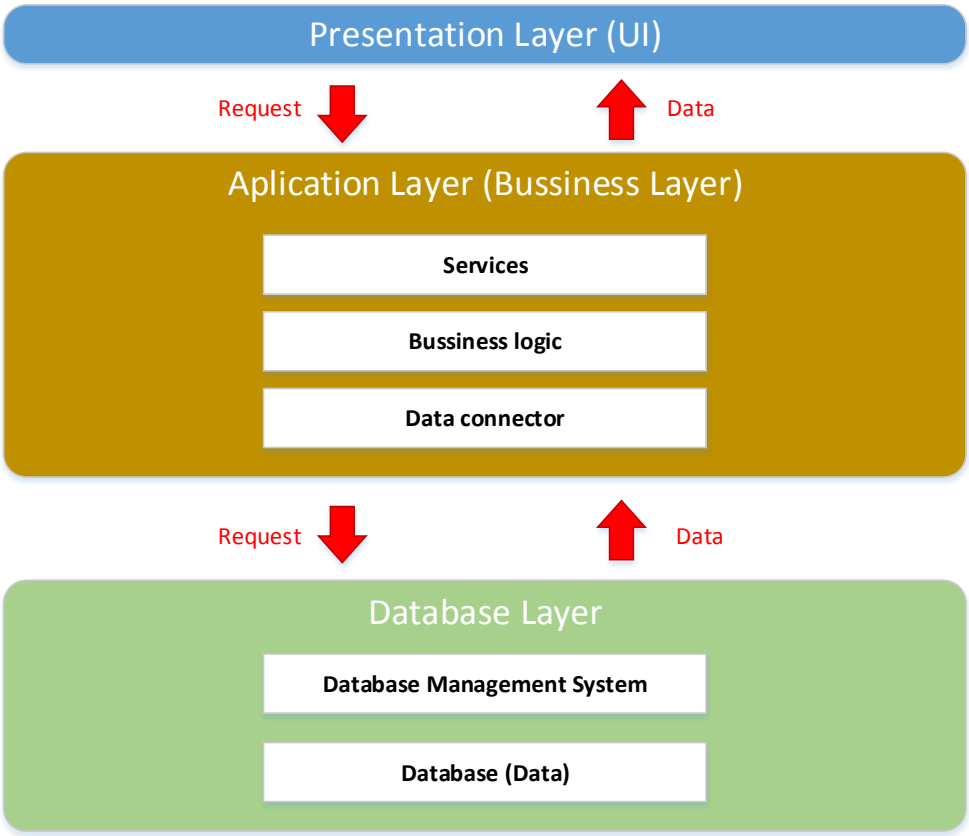
Obrázek 1 - Vertikální a horizontální škálování

3. Distribuovatelnost databází

U každé databáze, kterou je nutné rozložit na více částí (více disků, serverů, geografických clusterů apod.), je nutné vyřešit způsob rozmístění dat a jejich vzájemného provázání či striktních oddělení. Způsob, jakým bude tento problém řešen, bude mít zásadní vliv na výkon a způsob rozšiřování této databáze.

3.1. Na aplikační vrstvě (Database Switching)

Jedním z řešení rozložení dat v databázích je dělení již na aplikační vrstvě, které vytvoří programátor s ohledem na konkrétní požadavky. [11] V aplikaci je přesně definována množina oddělených databází a způsob rozložení dat mezi jednotlivé databáze (obvykle na úrovni konektoru). [10] U tohoto řešení je však nutno předem vědět, že data, která budou uchovávána v databázích, lze opravdu takto logicky rozdělit. Příkladem této distribuce může být databáze uživatelů, rozdělená podle počátečního písmene uživatelského jména.



Obrázek 2 - Schéma 3 vrstvé architektury

Pokud je tedy v aplikaci nutné přistupovat k údajům uživatele „novakj“, bude se pracovat s databází „N“, kde budou rovněž všechny související údaje.

Distribuci na aplikační vrstvě lze dále rozdělit na statickou a dynamickou podle toho, zda je počet databází pevně daný, nebo je možné ho měnit (obvykle zvyšovat).

3.1.1. Staticky

V případě statického rozdělování na aplikační vrstvě má veškerou odpovědnost za rozdělení programátor. Ten pevně specifikuje rozdělení a databáze (potažmo servery), do kterých budou data a dotazy směřovány. Příkladem může být zmíněná databáze uživatelů, rozdělená na servery podle abecedy.

Problém zde představuje mimo jiné velmi nerovnoměrné rozložení dat (databáze uživatelů začínajících na „N“ bude pravděpodobně mnohem větší než databáze uživatelů začínajících na „Q“).

3.1.2. Dynamicky

U dynamického rozdělování na aplikační vrstvě se předpokládá rovnoměrnější rozložení dat mezi databáze. Dochází zde k přidávání dalších databází (opět vyžaduje zásah programátora – přidání informací o dalším serveru do aplikace). [11] Příkladem může být databáze faktur s přírůstkem 1 serveru ročně, kde faktury jsou vždy ukládány dle příslušnosti k odpovídajícímu roku.

3.1.3. Nevýhody a omezení

Tento postup má celou řadu omezení a prakticky se téměř nepoužívá. Není zde například možné přímo realizovat vazby mezi daty v různých databázích (jen přes aplikační úroveň). Jakákoli operace s daty musí mít přesně určené databáze a i minimální chyba může mít velký dopad. Pozdější zásah do vnitřní či vnější struktury databází je prakticky nemožný (vyjma přidávání a odebírání celých databází), nebo vyžaduje velké změny v kódu i databázích. Jakoukoli jinou aplikaci třetí strany (například aplikace pro reporting) nelze použít.

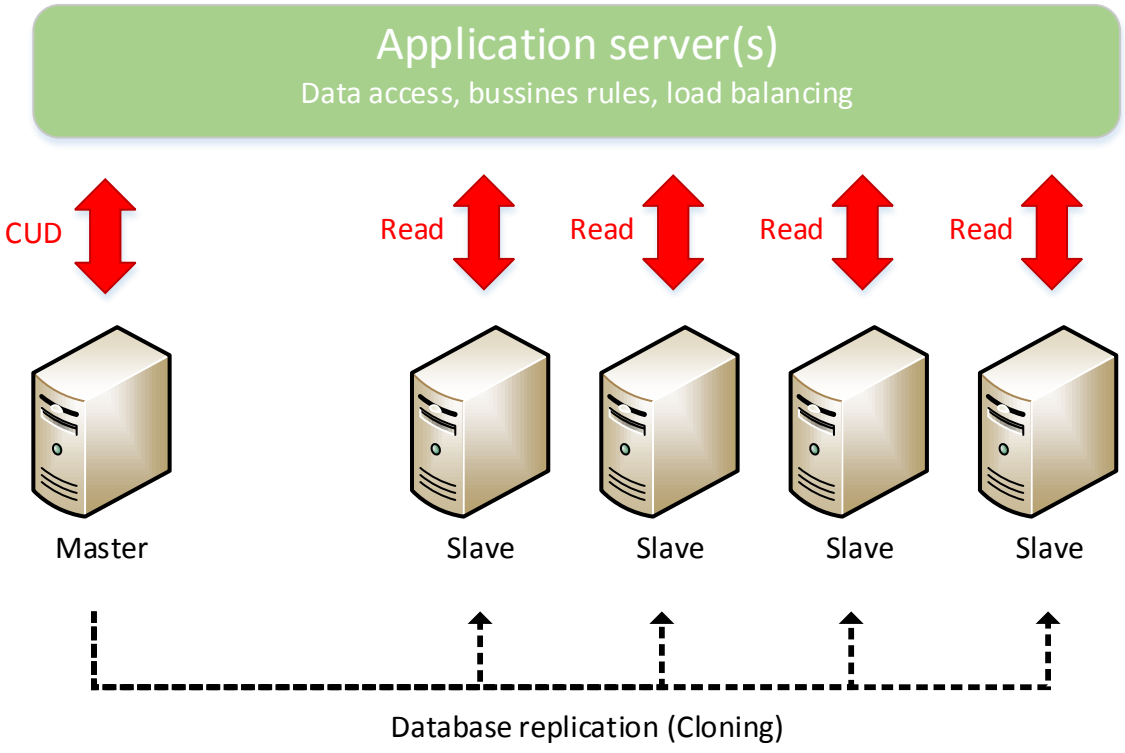
3.2. Database Mirroring

Mezi jednoduché způsoby jak zvýšit výkon databáze patří zrcadlení databází, které přestože dokáže při relativně nízkých nákladech zvýšit výkonnost databáze, příliš neřeší problém s dynamickým nárůstem dat.

3.2.1. Master – Slave

Princip Master – Slave spočívá v seskupení 1 Master serveru a *n* Slave serverů, které udržují co nejaktuálnější kopie databáze z Masteru. Master se přitom stará o všechny operace CUD (create, update, delete), zatímco Slave servery provádí pouze operace R (read). [12]

Tento princip je velmi závislý na online replikaci (musí být obvykle téměř real-time), a je tak vhodná především pro řešení, kde je předpoklad velkého množství dotazů a malé množství CUD operací. Dalším významným omezením je i fakt, že Master má vždy pevně daná omezení z hlediska kapacity (i za použití diskových polí), kterou musí současně splňovat i všechny Slave servery.



Obrázek 3 - Schéma replikace a dotazování v architektuře Master - Slave

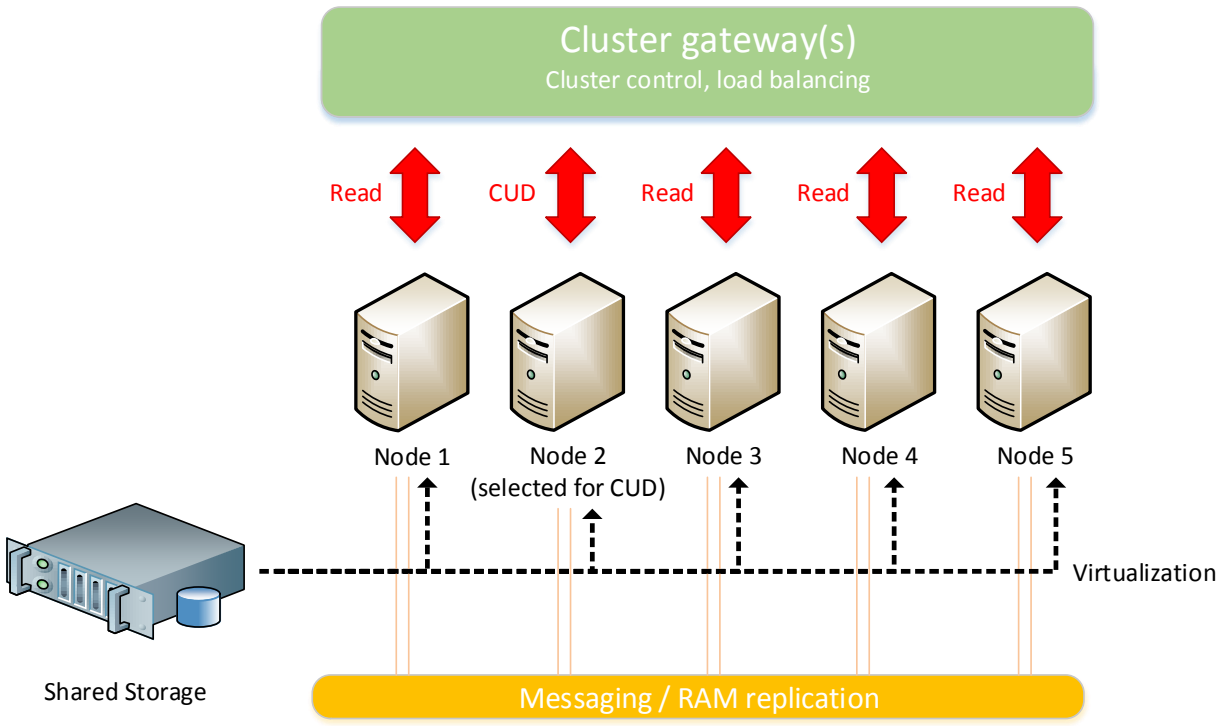
3.2.2. Basic cluster database

Dalším principem využívajícím zrcadlení je basic cluster database. Jedná se o cluster složený z jednotlivých serverů (nodů), které každý spravují vlastní instanci databáze a komunikují mezi sebou pomocí zpráv. Toto řešení často využívá externí sdílené síťové diskové pole (SAN – Storage Area Network), na kterém se databáze nachází. [12]

Obvykle se 1 server stará o operace CUD, zatímco ostatní pouze o Read. Na rozdíl od provedení Master-Slave může však operace CUD převzít libovolný server (např. při výpadku konkrétního nodu).

Existuje zde i varianta, kdy mohou všechny servery provádět všechny operace (CRUD). Aktuálnost/stejnost instancí databáze na jednotlivých nodech je v tomto případě zajištěna pomocí zpráv a real-time replikace RAM. Hlavním nedostatkem je zde právě množství komunikace mezi nody, tato varianta je tak použitelná spíše u clusterů s menším počtem nodů.

Nevýhodou obou variant basic clusteru je také velká závislost na výkonu sdíleného úložiště.



Obrázek 4 - Schéma fungování a dotazování v basic cluster database

3.3. Table Partitioning

Distribuovatelnost databází založená na table partitioningu je řešení vhodné pro uložení velkého množství dat v rámci jedné instance databáze napříč většímu množství disků. Podle způsobu dělení dat je možné ji dále dělit na horizontální a vertikální (výjimečně i kombinaci obou). Hlavní nevýhodou tohoto řešení je možnost právě 1 instance databáze. Teoreticky lze uvažovat i o kombinaci s principem Master-Slave, ale náročnost na udržení aktuálnosti dat je přímo úměrná množství použitých disků a četnosti CUD operací.

3.3.1. Horizontální Partitioning

U horizontálního partitioningu jsou data (zde tabulky relační databáze) rozděleny na bloky řádků. Realizace je náročná především na udržení přehledu primárních klíčů, které určují umístění dat v konkrétní části tabulky na konkrétním disku, případně serveru. Další komplikací je zde také udržení správnosti hodnot u polí s auto-inkrementací. [13]

Disk 1

ID	First name	Surname	E-mail	Address 1	Address 2	City	Post code	State
...	...	...	...	...	...	...	...	...
999997	Jan	Novak	novak@gmail.com	Horni	21	Praha	110 00	27
999998	Franta	Lála	lala@gmail.com	Dolni	7	Praha	120 00	27
999999	John	Snow	snow@gmail.com	Brown	452	Dallas	3047	2

Disk 2

ID	First name	Surname	E-mail	Address 1	Address 2	City	Post code	State
1000000	Marta	Mladá	mlada@gmail.com	Pravíkov	42/2	Pravíkov	394 70	27
1000001	Oliver	Patrik	patrik@gmail.com	Green	24	Dallas	3047	2

Tabulka 1 - Horizontální partitioning

3.3.2. Vertikální Partitioning

V případě vertikálního partitioningu jsou data rozdělena do tabulek vertikálně. Na první pohled toto řešení může připomínat klasickou relaci 1:1 (a principiálně to tak skutečně je), kde jako klíč slouží číslo řádku. Hlavním nedostatkem tohoto řešení je náročnost na cross-join dotazy, kde primární klíč je v jedné části tabulky a cizí klíč pro JOIN v jiné (na jiném disku). Řešení je tak vhodné především pro databáze s malým počtem velkých tabulek. [12] [13]

Disk 1

ID	First name	Surname	E-mail
1	Jan	Novak	novak@gmail.com
2	Franta	Lála	lala@gmail.com
3	John	Snow	snow@gmail.com
4	Marta	Mladá	mlada@gmail.com
5	Oliver	Patrik	patrik@gmail.com

Disk 2

Address 1	Address 2	City	Post code	State
Horni	21	Praha	110 00	27
Dolni	7	Praha	120 00	27
Brown	452	Dallas	3047	2
Pravíkov	42/2	Pravíkov	394 70	27
Green	24	Dallas	3047	2

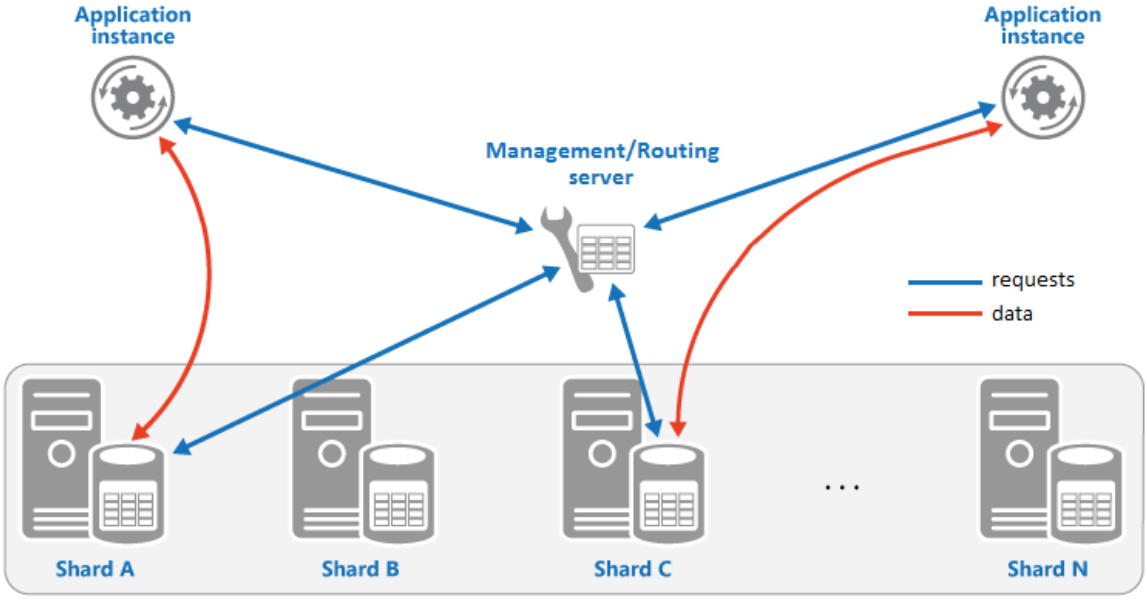
Tabulka 2 - Vertikální partitioning

3.3.3. Sdružené tabulky (Federated Tables / Functional Partitioning)

Mezi table partitioning lze do určité míry zařadit i sdružené tabulky, přestože se prakticky o žádné dělení tabulek nejedná. [13] Sdružení tabulek (často užívané např. v MySQL [14]) umožnuje propojení dat (relace) mezi tabulkami na různých serverech. Toto propojení je velmi závislé na konektivitě serverů a mnohdy funkčně omezené (v případě MySQL např. neumožňuje handlery a transakce). V praxi je proto vhodné spíše jako nástroj pro provázání tabulek databází, které jinak fungují nezávisle, ale jejichž data jsou do určité míry provázána (předejde se tak zbytečné duplikaci dat). Jedná se ale víceméně o zjednodušení, neboť tyto potřeby lze pochopitelně ošetřit i na aplikační vrstvě.

3.4. Sharding

Sharding[1] je metoda založená na horizontálním partitioningu, kde jednotlivé části databáze (shardy) jsou umístěny na různých serverech (nodech). Každý server přitom tuto část spravuje nezávisle – udržuje svou individuální instanci. O optimální umisťování dat na jednotlivé servery se starají zpravidla řídící (management) servery, které následně i směrují požadavky na konkrétní servery při dotazování (více v kapitole *Sharding Strategies*). Z pohledu klientské aplikace se však celý systém tváří jako jediná databáze.

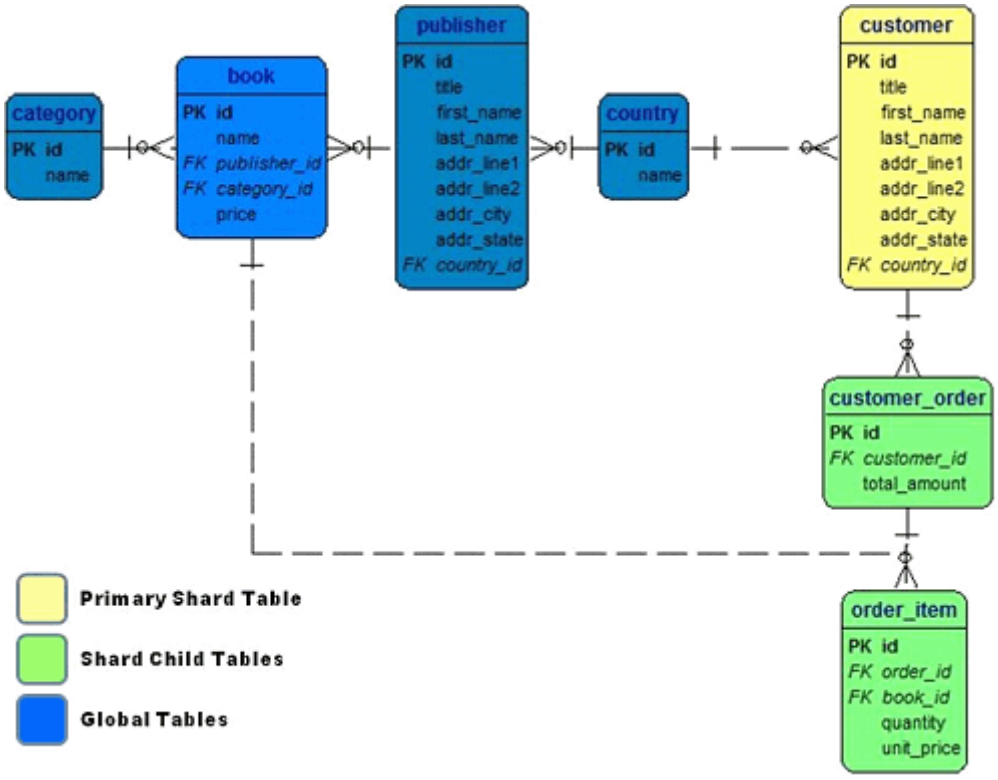


Obrázek 5 - Princip shardingu[2]

[1] Sharding je ve zdrojích často úzce spojován pouze s NoSQL databázemi, nicméně pro účely této práce je výraz použit jako obecné označení pro partitioning mezi více servery.

[2] Převzato z *https://msdn.microsoft.com/en-us/library/dn589797.aspx*

Při použití shardingu nemusí být vždy děleny všechny části databáze (viz obr. xy???), je však nezbytné, aby byly společně (na stejném serveru) ukládány všechny závislé části („child tables“ v případě relačních databází). V opačném případě by docházelo při dotazování k prodlevám způsobeným postupným procházením více serverů, než je nezbytně nutné. Vazby mezi serverem a shardem přitom nemusí být nezbytně 1:1, ale jeden server může hostovat i více shardů (pokud je to vzhledem ke struktuře databáze a výkonu konkrétního serveru výhodné). [15]



Obrázek 6 - Ukázka závislosti tabulek při shardingu v relačních databázích[3]

V extrémním případě přímé vazby mezi daty na různých serverech neexistují vůbec – princip striktního oddělení (Shared-nothing architecture). Tento princip je základem pro NoSQL databáze.

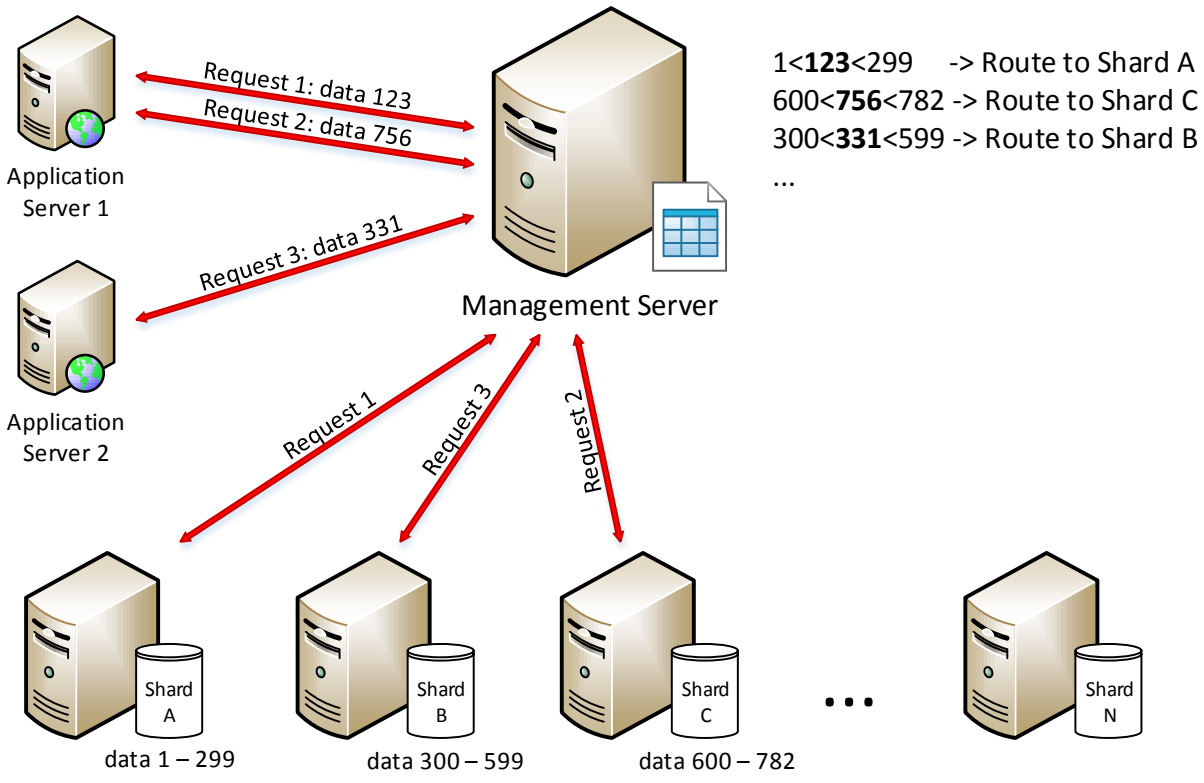
[3] Převzato z *http://codefutures.com/database-sharding/*

3.4.1. Sharding Strategies

Existují různé strategie, jak rozdělit data do jednotlivých shardů. Volba vhodné strategie je pro konečný výkon klíčová. Strategie určují především to, kam budou ukládána nová data, jak určit shard s daty která jsou dotazována a jakým způsobem bude řešena zpětná optimalizace (rebalancing). Mezi ty nejrozšířenější základní strategie patří [9] [15]:

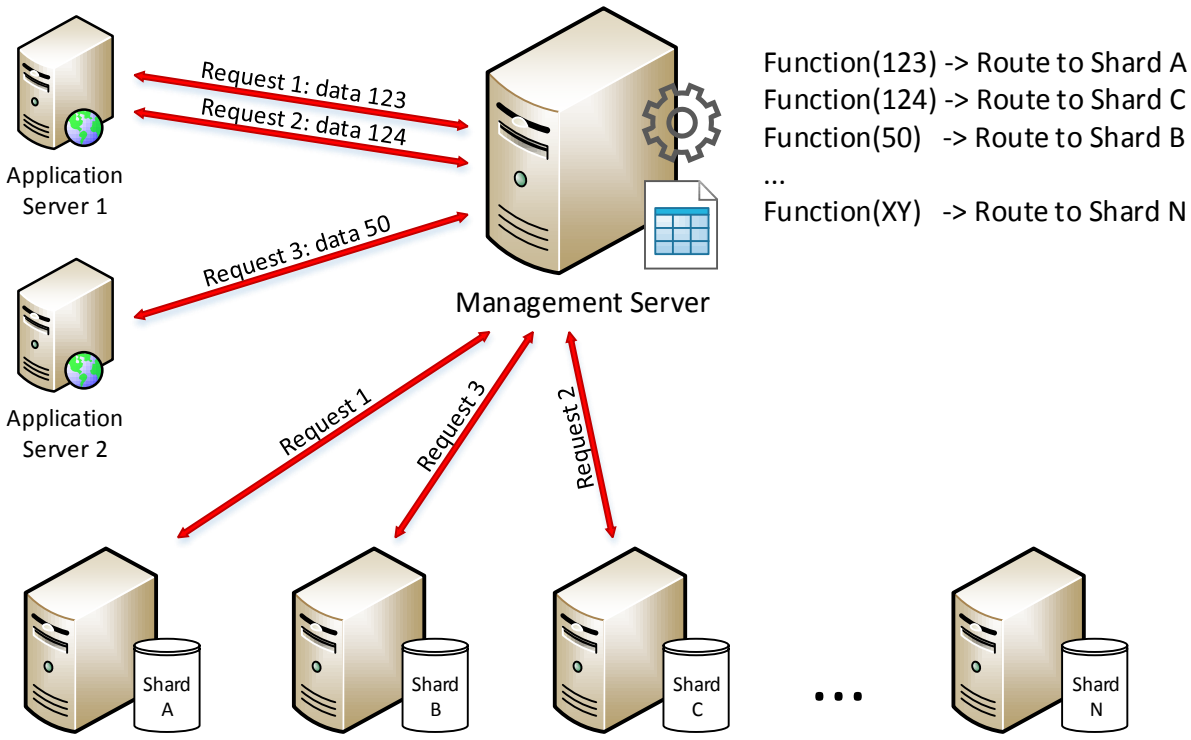
- Shard by a primary key
- Shard by function(key)
- Shard by range
- Master shard index table (key-value)

Shard by primary key je strategie velmi podobná horizontálnímu partitioningu s tím rozdílem, že k dělení dat dochází napříč servery (resp. shardy) a je tedy zapotřebí směrovací prvek (management server). Tato strategie nijak neřeší vhodné rozložení jak zátěže, tak vzájemně souvisejících dat.



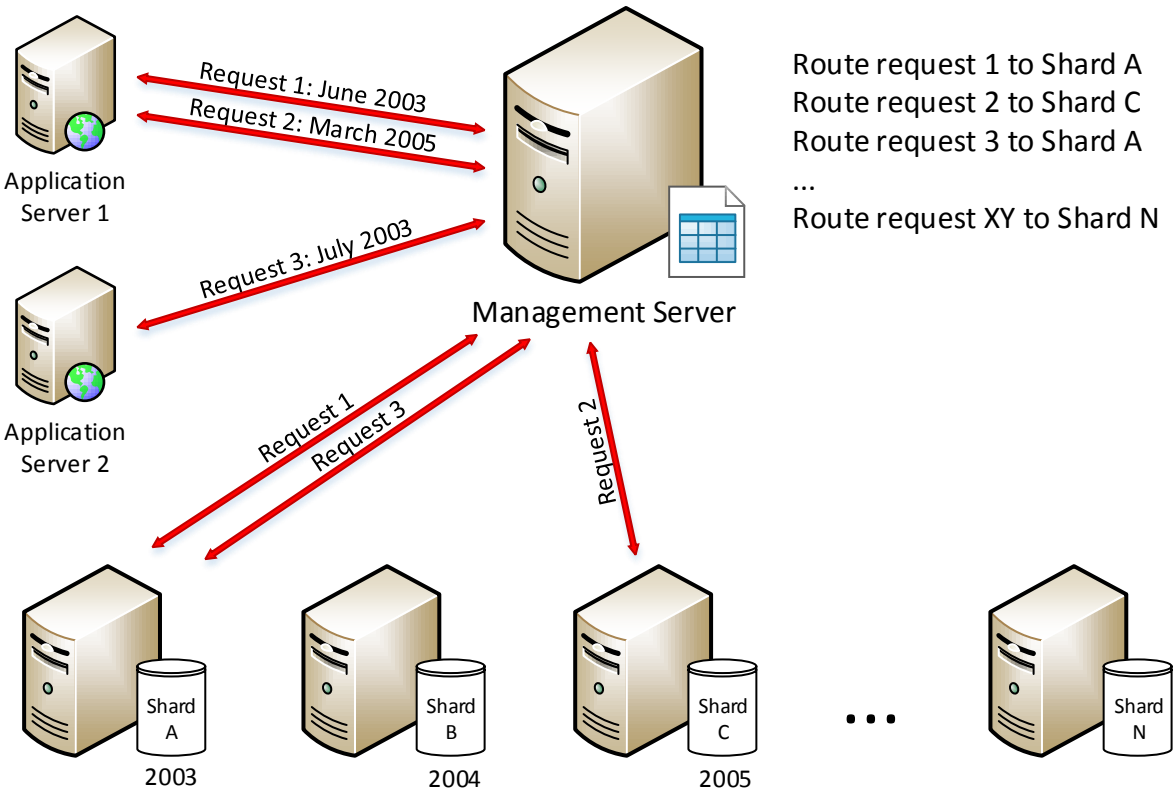
Obrázek 7 - Shard podle primárního klíče

Shard by function(key) je strategie obdobná shard by primary key, která se snaží o rovnoměrnější rozložení zátěže pomocí aplikování různých matematických funkcí na klíč (obv. primární klíč). Data tak již nejsou rozložena za sebou podle jejich přidání, ale rovnoměrně pomocí funkce rozložena mezi shardy. To však zvyšuje zátěž směrovacího prvku, který musí funkci provádět při každém požadavku. Je zde také vhodné znát předem počet shardů, protože pozdější přidávání vyžaduje úpravu funkce a náročný rebalancing.



Obrázek 8 - Shard podle funkce

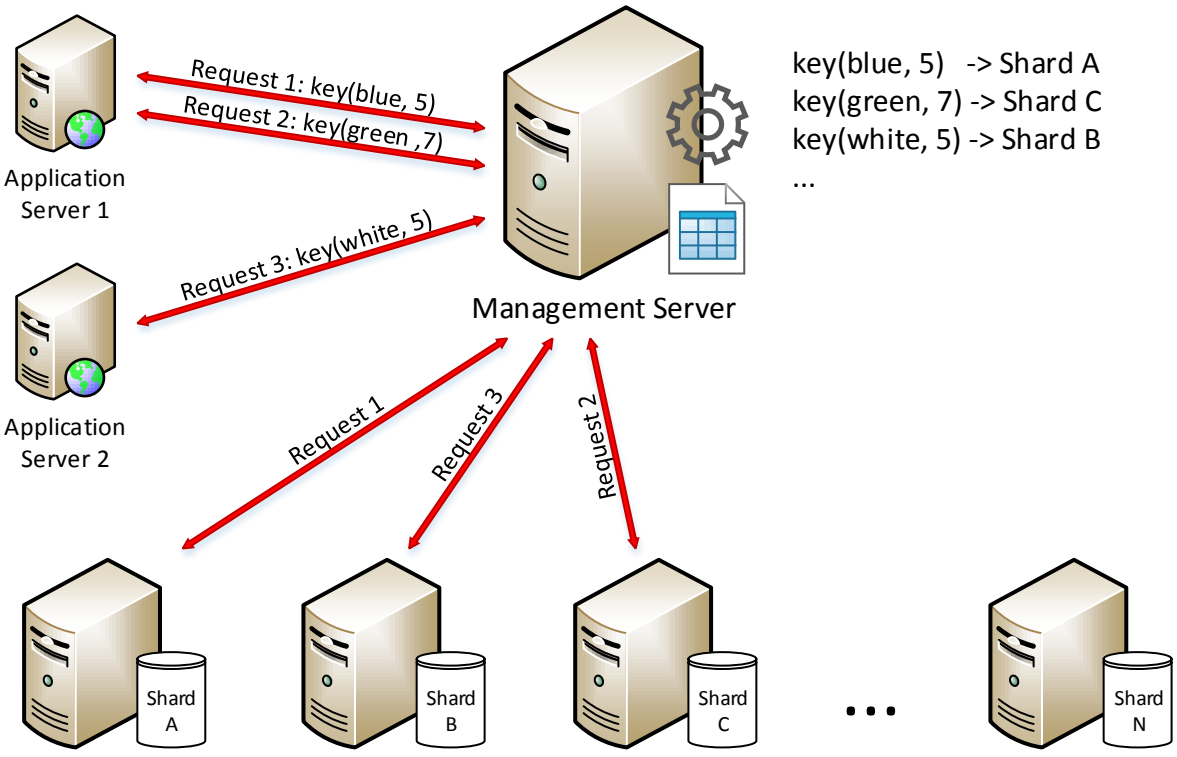
Shard by range je založena na seskupování dat podle určité společné vlastnosti. Princip je téměř shodný s dynamickým řešením na aplikační vrstvě, jen je rozhodování/směrování na jednotlivé shardy přesunuto z aplikační vrstvy na samostatný management server. To je výhodné, především pokud s databází pracuje více aplikací. Příkladem může být stejně jako v případě řešení na aplikační vrstvě databáze dělená na jednotlivé shardy podle příslušnosti dat k roku/měsíci nebo např. podle zákazníka (máme-li několik málo stálých zákazníků).



Obrázek 9 - Shard podle rozsahu

Master shard index table je dnes pravděpodobně nejpoužívanější strategií využívající principu klíč-hodnota (key-value). Vybrané vlastnosti (atributy) dat jsou zpracovány předem definovaným způsobem, na základě kterého jsou data přidělena do konkrétního shardu. Důsledkem je tvorba vlastních strategií vysoce přizpůsobeným konkrétnímu problému, kterou je možno i později poměrně snadno upravovat.

Klíč odkazující na konkrétní data je buďto uchováván na řídicím/směrovacím management serveru či serverech (v takovém případě odkazuje i na konkrétní shard, ale zvyšuje se zatížení management serverů) nebo je ukládán společně s daty a indexován přímo na databázových serverech. Nevýhodou je pak skutečnost, že při dotazování musí být osloveny všechny tyto servery s dotazem, zda obsahují data s konkrétním klíčem. Klíč přitom může být jak virtuální hodnota (ID), tak častěji určující prvek pro konkrétní data (primární klíč, číslo zákazníka, klíčová slova, apod.), přičemž tento klíč může být i poměrně složitý výraz, obsahující větší množství parametrů.

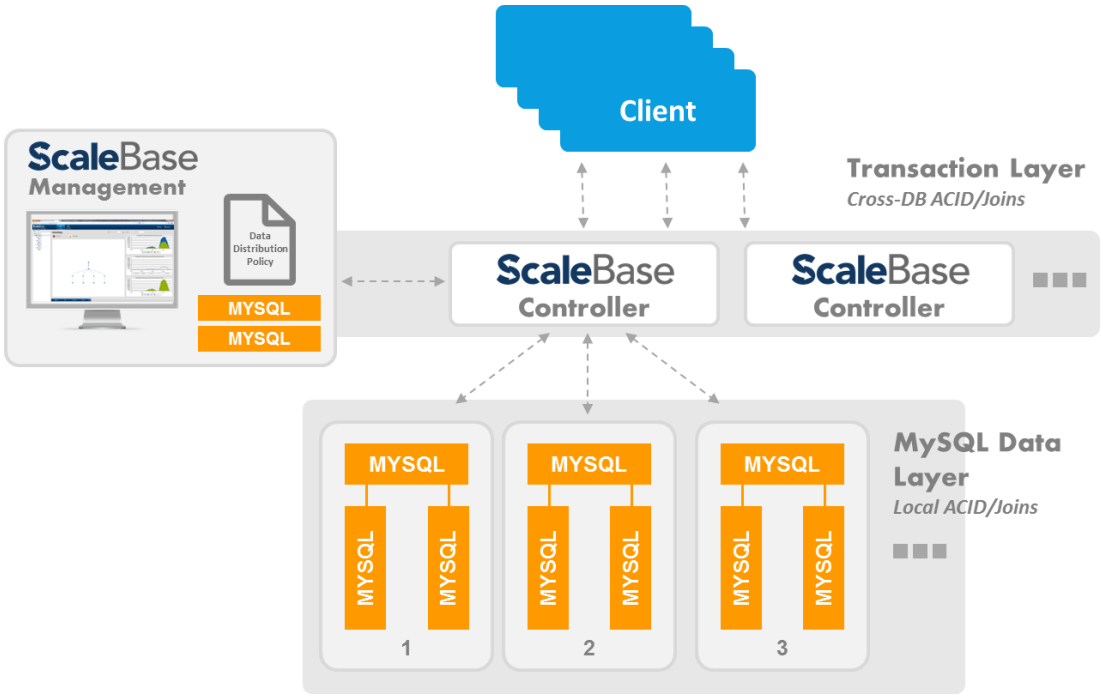


Obrázek 10 - Shard založený na principu klíč - hodnota

3.4.2. Nadstavbové management systémy

Existují distribuované databázové management systémy (DDBMS), rozšiřující možnosti běžných databází, které nejsou přímo určeny pro distribuovaná řešení. Princip spočívá v přidání nadstavbové vrstvy mezi databáze (databázi rozdělenou na menší databáze - shardy) a mezi aplikační vrstvu. DDBMS poté stejně jako v případě klasických distribuovaných databází založených na shardingu řídí veškerou činnost, rozložení dat a provádí rebalancing, navenek se přitom pro aplikaci tváří jako jediná databáze.

Mezi nejznámější příklady patří například ScaleBase [16] distributed database management system, který je nadstavbou pro MySQL.



Obrázek 11 - Architektura nadstavbového systému ScaleBase[4]

[4] Převzato z *https://www.scalebase.com/technology/*

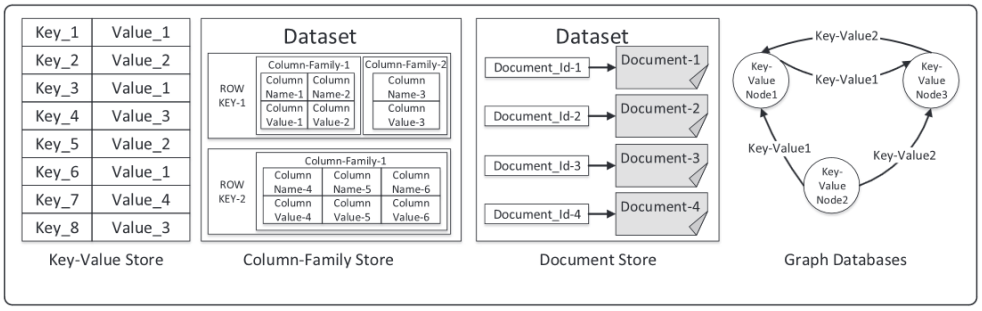
3.5. NoSQL

Mezi NoSQL (Not Only SQL) řadíme ty databáze, které nejsou založeny na relačním modelu a nevyužívají pro přístup a práci jazyk SQL (nebo ho nepodporují vůbec). Cílem NoSQL databází je snadná škálovatelnost a distribuce dat, přičemž data budou co nejméně omezena schématem/strukturou (schema-less přístup). Distribuce dat v NoSQL databázích vychází z horizontálního partitioningu se striktním oddělením dat (share-nothing architecture). [17] [4]

Obrázek 12 - Rozdíl v přístupu k vazbám u relačních a NoSQL databází[5]

Typy NoSQL databází

Existují různé způsoby dělení NoSQL databází do skupin [17] [18], mezi nejrozšířenější však patří dělení na základě datového modelu na: Key-Value, Column-Oriented, Document-Oriented, Graph Database. [17] [19] [20]



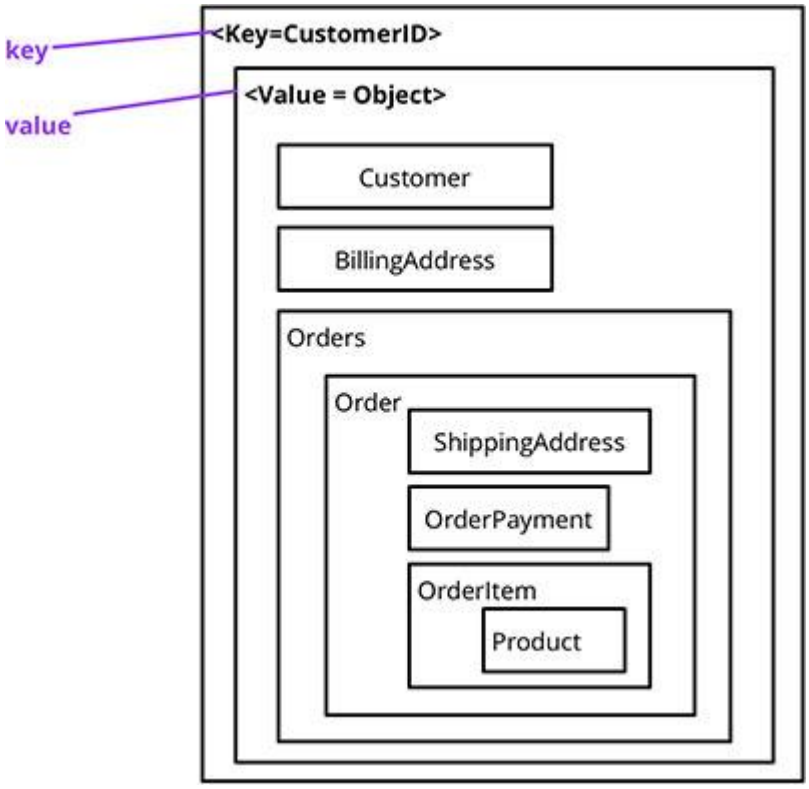
Obrázek 13 - Typy NoSQL databází[6]

[5] Převzato z „MongoDB vs. MySQL with ScaleBase Comparison" dostupné na *https://www.scalebase.com/resources/whitepapers/*

[6] Převzato z *http://www.journalofcloudcomputing.com/content/pdf/2192-113X-2-22.pdf*

3.5.1. Key-Value databases

Datový model key-value, jak je patrno již z názvu, využívá pro ukládání a přístup k datům (hodnotám) unikátní klíč, dohromady tvořící dvojici klíč-hodnota. Neexistují zde žádné vazby mezi objekty či struktura, resp. struktura existuje až ve formě hodnoty. V praxi se vyskytuje klíč nejčastěji ve formě množiny výrazů, zatímco hodnota má obvykle strukturu objektů s vazbami v podobě zapouzdřování (objektový přístup k programování). Hodnota však představuje spíše útržkovou informaci, která je vzhledem k závislosti na klíči sama o sobě bez širších znalostí nepoužitelná (na rozdíl od relačních databází, kde jsou vazby jasně vidět ze struktury databáze). V případě tohoto modelu tedy nelze žádným způsobem vyhledávat či pracovat s hodnotami bez předchozího vyhledání pomocí klíče.



Obrázek 14 - Příklad objektů v Key-Value NoSQL databázi[7]

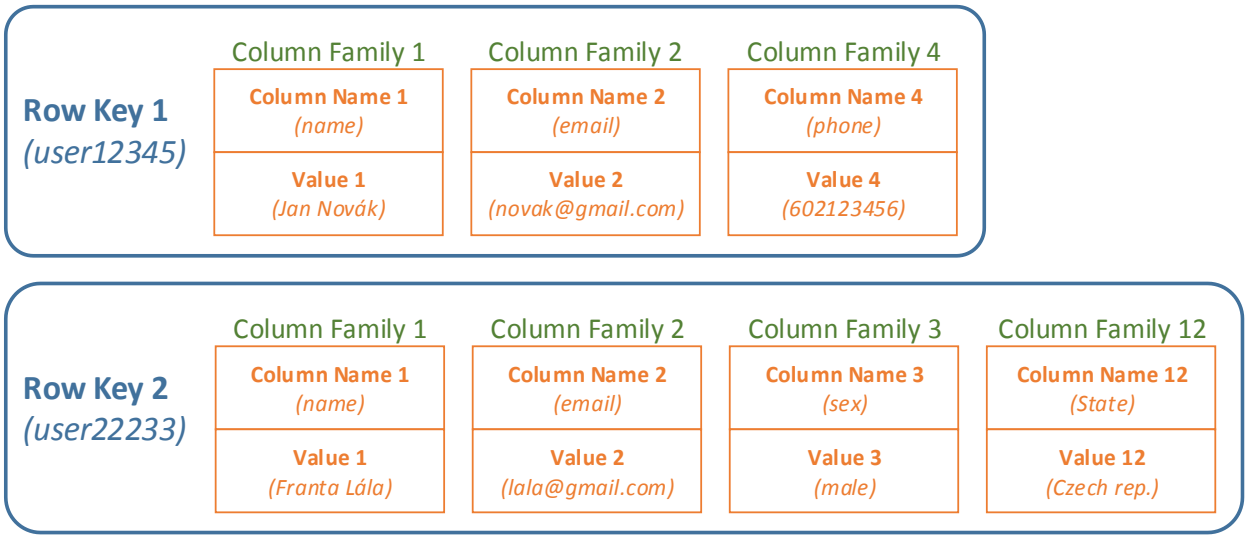
[7] Převzato z *http://www.thoughtworks.com/insights/blog/nosql-databases-overview*

3.5.2. Column-Oriented databases

Column-Orineted databázový model (nebo také Column-Family model) na první pohled připomíná tabulky z relačních databází. Uložení dat si lze představit jako dvourozměrné pole, kde pro vyhledání záznamu musíte znát řádek a pro konkrétní data také sloupec. Jedná se o rozšíření modelu Key-Value, kde klíč slouží opět k vyhledávání (řádku), a hodnotu pak tvoří data ve sloupcích tohoto řádku. Tento princip byl poprvé použit u BigTable od Googlu. [21]

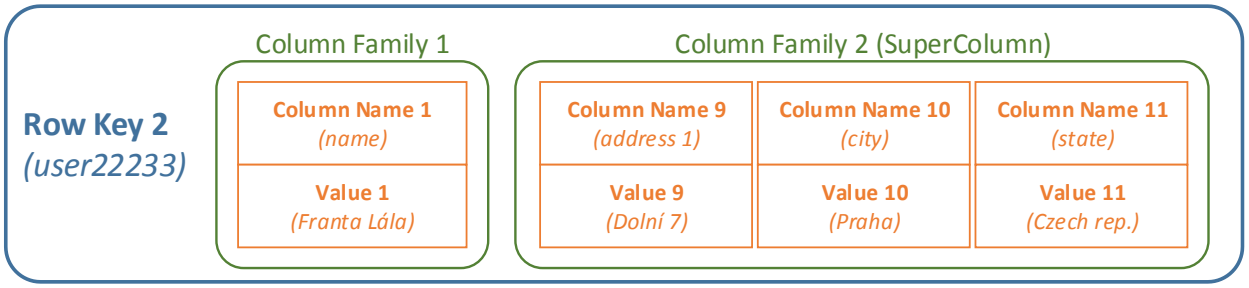
Na rozdíl od relačních databází zde řádky nemusí mít shodné sloupce co do počtu, ani typu. Dva řádky tak mohou mít teoreticky společný i jeden nebo žádný sloupec. Sloupce lze rovněž k libovolným řádkům přidávat či odebírat bez ohledu na jiné řádky (jedná se o objektový přístup, nikoliv o zápis prázdných hodnot, jak by byl stejný případ řešen v relačních databázích).

Jednotlivé řádky jsou obvykle ukládány jako celky na serverech/nodech (celý řádek na 1 nodu), případně na více serverech s přímou indexací (obdoba vertikálního partitioningu). To je výhodné vzhledem k faktu, že tato data (1 řádek) jsou obvykle vyžadována společně. Pokud je požadavek pouze na část hodnoty (na konkrétní informaci, např. email), je nutné znát kromě klíče i název sloupce (další párování klíč-hodnota, resp. jméno-hodnota).



Obrázek 15 - Příklad objektů ve sloupcově orientované NoSQL databázi (bez vnořování)

Dalším možným rozšířením Column-Oriented modelu je vnořování sloupců, tedy 1 sloupec obsahuje více dalších sloupců v rámci jednoho řádku – vzniká tzv. super sloupec (Super Column). Data v tomto super sloupci jsou opět přístupná společně (jsou ukládána společně).

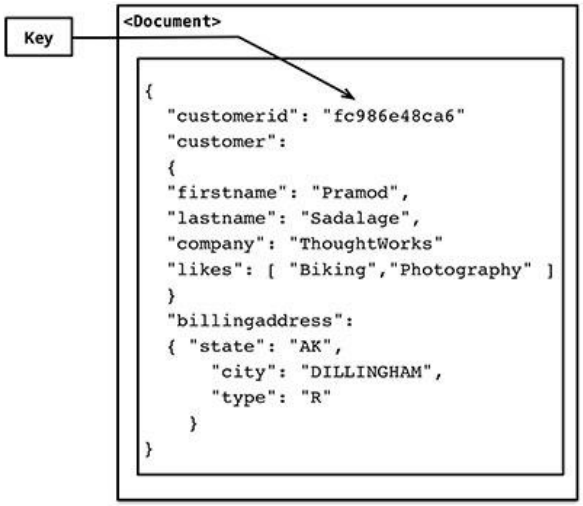


Obrázek 16 - Příklad objektů ve sloupcově orientované databázi (s vnořováním)

3.5.3. Document-Oriented databases

Model založený na ukládání strukturovaných dokumentů je dalším derivátem Key-Value modelu. Tyto dokumenty využívají nejčastěji formáty JSON, BSON, XML, apod. Tyto jednoduché formáty umožňují indexaci nejen pomocí klíče, ale částečně i podle hodnot (díky faktu, že i samotné struktury dokumentů mají prakticky podobu klíč-hodnota).

Jednotlivé dokumenty nemusí mít shodnou vnitřní strukturu, často však mají. Některá řešení tak nabízí možnosti sdružování dokumentů do kolekcí za účelem společného ukládání a rychlejšího vyhledávání.



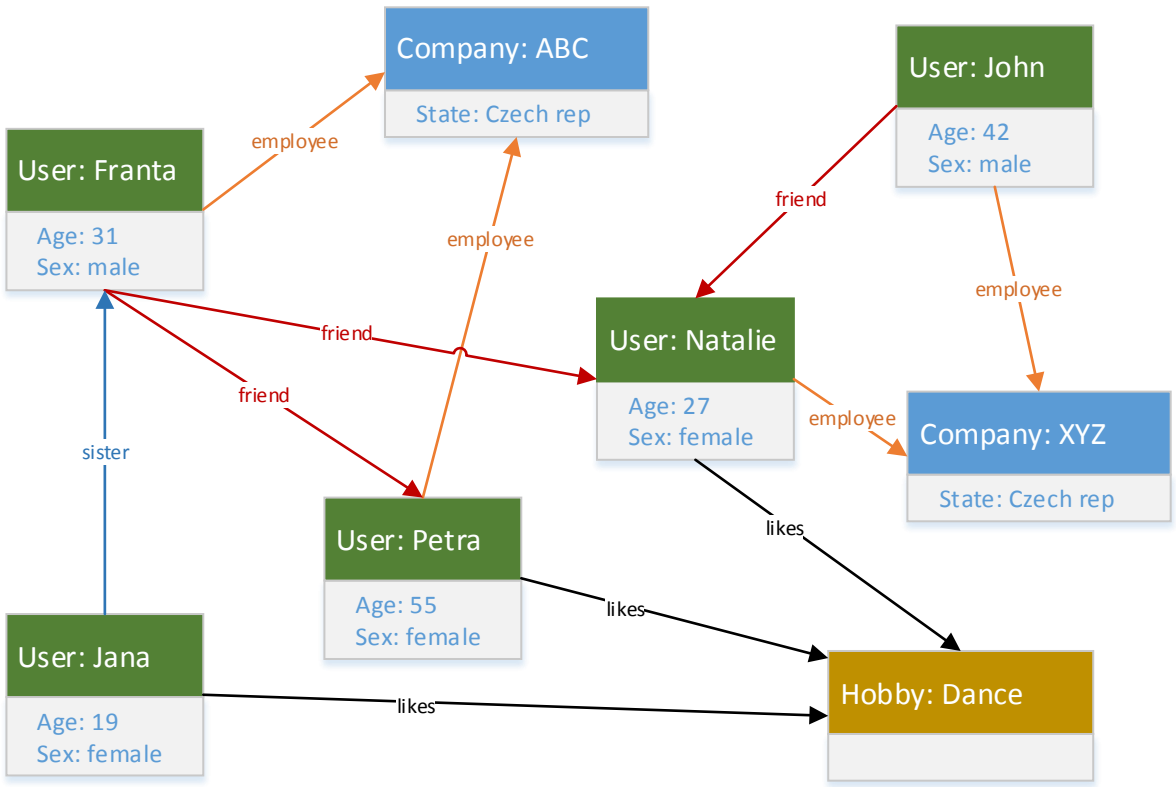
Obrázek 17 - Příklad objektů v dokumentově orientované NoSQL databázi[7]

[8] Převzato z *http://www.thoughtworks.com/insights/blog/nosql-databases-overview*

3.5.4. Graph Databases

Databázový model založený na matematické teorii grafů je oproti předchozím 3 modelům značně odlišný. Databáze se skládá z entit (nazývané také objekty, vrcholy či nody[9]) a ze vztahů (relationships) mezi nimi (nazývané také hrany=edges nebo propojení=links). Entity a vztahy mají pak další vlastnosti (atributy) určující jejich povahu.

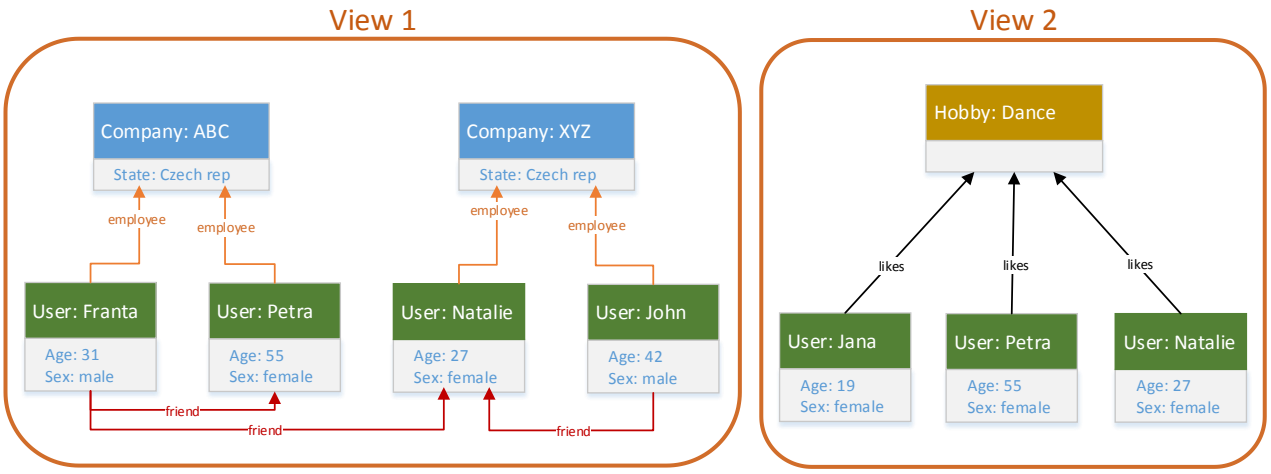
Samotnou databázi si lze představit jako orientovaný graf, do kterého lze dynamicky přidávat nové entity a jejich vztahy, nebo vztahy mezi již existujícími entitami. To je velký rozdíl proti relačním databázím, kde je pro přidání nové relace zapotřebí zásah do struktury, zatímco u grafových databází je struktura vytvářena průběžně. Nevýhodou však může být to, že stejný typ vztahu se může vyskytovat na různých místech, a pokud bychom tento chtěli upravit (nebo nahradit jiným), je nutné všechny postupně jednotlivě vyhledat a upravit. To samé platí pro entity (např. pokud chceme všem entitám *User* přidat novou vlastnost *From*).



Obrázek 18 - Příklad objektů v grafové NoSQL databázi

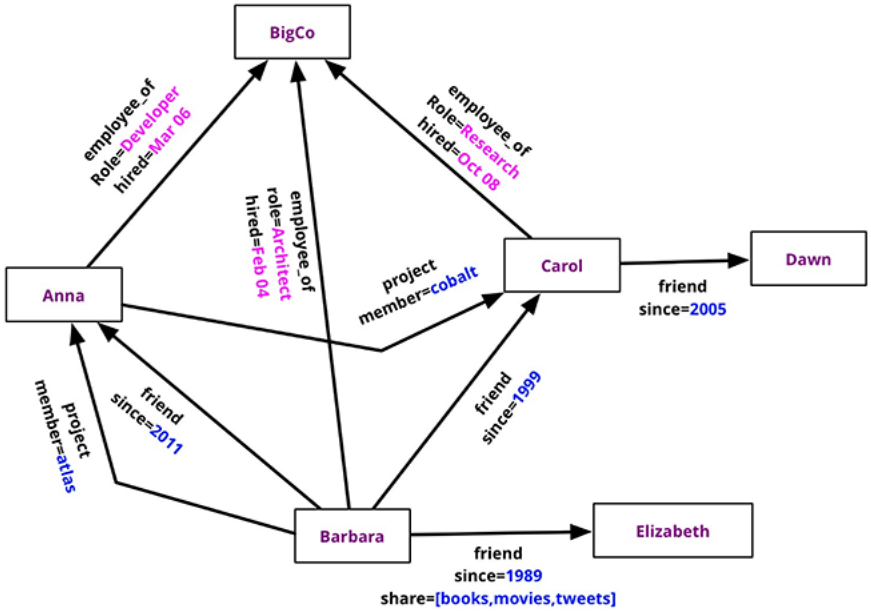
[9] Nijak nesouvisí s *nody* u clusterů.

Vzhledem k tomu, že v grafových databázích je možné pomocí vztahů spletitě propojit (přes 1 či více skoků) i zdánlivě na první pohled nesouvisející entity, vznikají tak velmi složité struktury (na rozdíl od přehledně modelovaných relačních databází). To ale na druhou stranu umožňuje velkou flexibilitu a komplexnost celé databáze, kde na jedinou databázi je možné nahlížet z různých zcela odlišných pohledů.



Obrázek 19 - Příklad různých pohledů na data v grafové NoSQL databázi

Počáteční vyhledávání entity (nebo vztahu) probíhá opět nejčastěji pomocí metody key-value, poté následuje vyhledávání „do hloubky“ pomocí vztahů a modelování pohledu. Vztahy přitom mohou stejně jako entity mít další upřesňující parametry. Vhodně tvořené vztahy, jejich parametry a indexace, jsou tak klíčové pro výkon tohoto typu databáze.



Obrázek 20 - Ukázka parametrů na hranách NoSQL databáze[10]

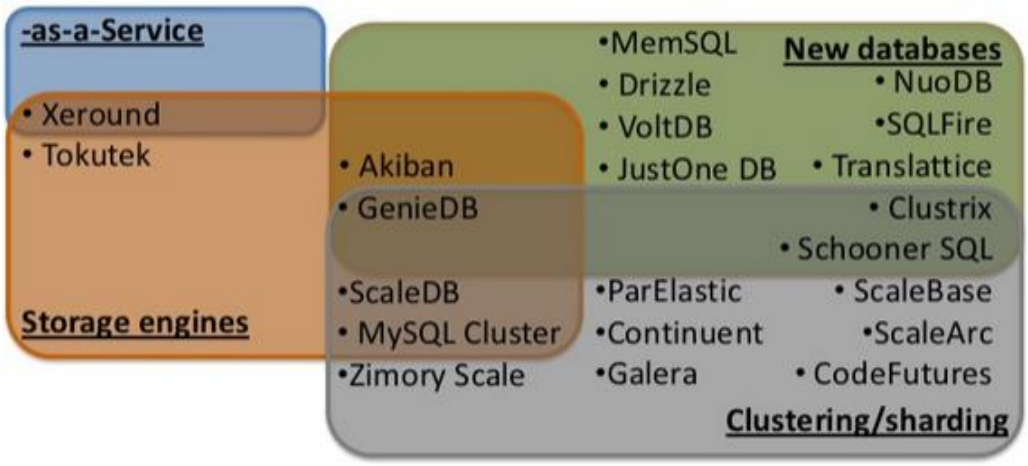
Distribuovatelnost grafových databází je technicky proveditelná (vztahy napříč servery), nicméně vzhledem k posloupnému prohledávání při dotazech („vyhledávání do hloubky“), je rozložení na více serverů (navíc bez jasné představy o struktuře) značným výkonnostním omezením. Je zde také značné riziko bouncingu (skákaní při dotazu přes vztahy mezi 2 servery tam a zpět).

Grafové databáze jsou velmi specifické, a nejsou tak vhodné pro všechny typy situací, například pro databáze s malým množstvím vazeb a velkým množstvím záznamů (např. logy) jsou prakticky nemyslitelné. Naopak velmi vhodné jsou pro data v situacích jako závislostní analýzy, rozpoznávání vzorů, trasové podklady (navigace) a v neposlední řadě sociální sítě. Vzhledem k uvedeným komplikacím při distribuovatelnosti na více serverů jsou často použita jako prvotní dotazovací databáze, kde podrobnější data k entitám jsou uložena v jiném typu databáze. Tento princip více databází je používán např. u sociálních sítí, kde je díky grafovým databázím možné rychlé vyhledávání pomocí vazeb (přátelé, skupiny, likes, apod.), zatímco konkrétnější data (články, galerie, podrobnosti uživatelů, atd.) jsou uložena v jiné databázi (relační, objektové, jiné než grafové NoSQL).

[10] Převzato z *http://www.thoughtworks.com/insights/blog/nosql-databases-overview*

3.6. NewSQL

Mezi NewSQL databáze řadíme takové databáze, které jsou založeny na relačním modelu (jejich primárním dotazovacím jazykem je SQL), ale současně umožňují dobrou horizontální škálovatelnost. Na rozdíl od NoSQL databází jsou závislé na vnitřní struktuře a obvykle plně podporují ACID a transakce. [20]



Obrázek 21 - NewSQL ecosystem[11]

3.6.1. Principy distribuce dat

Distribuce dat je v NewSQL databázích založena na horizontálním partitioningu, existují zde však velké odlišnosti v konkrétních implementacích. Je to dáno i tím, že některé databáze byly vytvořeny nově zcela od začátku a vnitřními procesy se od tradičních relačních databázi výrazně liší. Jiné platformy jsou naopak rozšířením či modifikací existujících relačních databází, které byly původně navrhovány s ohledem na vertikální škálovatelnost. Většina NewSQL databází se zakládá na některém (nebo kombinaci více) níže uvedených principů:

Messaging a konstantní hashovací algoritmus

Messaging je princip založený na tom, že každý z nodů ví, jaká konkrétní data obsahují ostatní nody. To je realizováno tak, že ukládání nových dat na konkrétní node se řídí konstantním hashovacím algoritmem a nody si mezi sebou předávají informace o změnách. Každý node pak

[11] Převzato z *http://www.slideshare.net/mattaslett/mysql-vs-nosql-and-newsql-survey-results-13073043*

může přijmout dotaz, a pokud se týká dat na jiném nodu, pak zprostředkuje dotazování. Na tomto principu je založena například databáze Clusterix[12].

Multiversioning

Aby nedocházelo k omezenému přístupu k datům při zamykání, které představuje při práci s distribuovanými daty ještě větší problém než u nedistribuovaných databází (může se vyskytovat častěji a ve větší míře), používá se častěji než zamykání řešení pomocí metody multiversioning. Ten je sice náročnější z pohledu výpočetního výkonu a udržení konzistence, umožňuje však vyšší dostupnost dat.

Oddělení transakční logiky a dat

Dalším možným principem nakládání s daty je striktní oddělení dotazování a transakční logiky od dat. Dotazování probíhá na vrstvě, která pracuje pouze s projekcemi dat, k samotnému uložení dochází až po provedení celé transakce odesláním na datovou vrstvu. Tuto práci s daty lze připodobnit k paměti RAM a pevnému disku počítače. Potřebná data jsou načtena do paměti RAM, následně jsou provedeny úpravy a po skončení jsou data uložena na disk. Tato analogie je ve skutečnosti velmi přesná, neboť transakční servery v ideálním případě pracují s daty pouze v paměti bez nutnosti pevného disku. Tento princip využívá například databáze NuoDB[13].

Obrázek 22 - Architektura NuoDB[12]

[12] *http://www.clustrix.com/* (*https://www.youtube.com/watch?v=PUq1fYZlNPs*)

[13] http://www.nuodb.com/explore/newsql-cloud-database-how-it-works

In-Memory

Princip uchovávání dat pouze v paměti není použit jen při oddělení transakční logiky, ale mnohdy je využíván jako nástroj pro replikaci či pro dávková zpracování (mnohdy jsou totiž data ukládaná naposled dotazována častěji).

Replikace a rebalancing

Důležitými součástmi distribuovanosti za použití horizontálního partitioningu jsou replikace a rebalancing. Pokud nepracujeme s daty metodou delete-nothing (žádná data nejsou nikdy odstraněna, nepotřebná jsou pouze označená příznakem), dochází časem k nerovnoměrnému rozložení dat mezi nody. Aby se zabránilo snížení výkonu databáze, je tak nutné provádět rebalancing dat mezi jednotlivými nody. Další možností je vzájemná replikace dat mezi vytíženějšími a méně vytíženými servery – většina NewSQL databází stejně již replikaci mezi nody využívá pro případ jejich selhání, vhodně zvolená replikace tam může pomoci udržení výkonu.

Prefixy klíčů

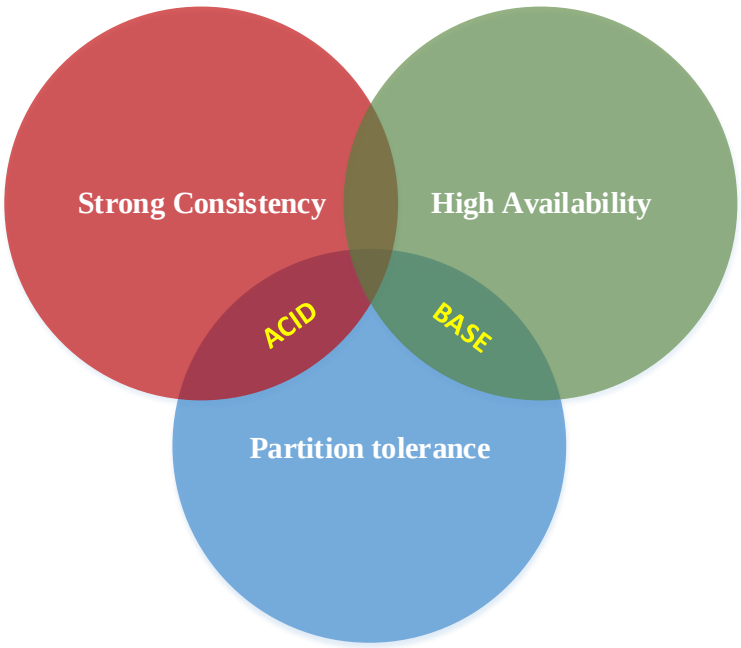
Některé databázové platformy přidělují při distribuci dat jejich primárním klíčům prefixy, pomocí nichž je možné identifikovat související skupiny dat napříč nody. Tento princip je podobný row-key známému z *Column-Oriented databases*.

4. CAP (Brewer's Theorem)

V roce 2000 byl na sympóziu o principech distribuovaných výpočtů (Symposium on Principles of Distributed Computing - PODC) představen CAP Theorem (někdy též nazývaný Brewer's Theorem, podle jeho autora Erica Brewera). Podle tohoto teorému není možné, aby distribuovaný systém splňoval zároveň všechny následující vlastnosti, ale pouze 2 ze 3 [8]:

- Strong **C**onsistency (úplná konzistence): Všechna data musí splňovat úplnou konzistenci, a všichni klienti tak vidí vždy stejná data.
- High **A**vailability (vysoká dostupnost): Klienti musí mít vždy přístup k datům, a to i v případě, že dojde k selhání některého z nodů clusteru.
- **P**artition tolerance (odolnost vůči chybám sítě): Databázové operace musí být odolné vůči chybám síťové komunikace (zpoždění, výpadky, apod.) mezi některými nody.

Později se ukázalo (CAP Twelve Years Later [25], Perspectives on the CAP Theorem [26]), že distribuované databáze, které by zcela splňovali úplnou konzistenci a vysokou dostupnost, prakticky není možno realizovat (toto splňují nedistribuované databáze). Z toho plynou dva možné modely distribuovaných databází: databáze splňující CP (consistency + partition tolerance) a databáze splňující AP (availability + partition tolerance).



Obrázek 23 - CAP Theorem

5. Transakce

Transakcí se rozumí databázová operace (často skupina operací), kterou provádí DBMS. Mezi nejběžnější operace patří zápis, čtení, úprava dat v databázi, a dále pak potvrzení (commit/save changes) či zrušení transakce (rollback). [22] [23]

Způsob provádění transakcí a její vlastnosti velmi výrazně ovlivňuje chování a funkce databáze, stejně jako její výkon. V současné době existují dva základní modely transakcí, označované podle jejich vlastností jako ACID a BASE.

5.1. ACID

ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability) je transakční model existující společně s relačními databázemi již od sedmdesátých let 20. století, kdy jeho vlastnosti poprvé definoval James Gray. [24] Samotný model pak popsal a dal mu název Andreas Reuter. [25] [26]

ACID model je charakterizován 4 vlastnostmi [26] [27]:

- **Atomicity** (atomicita) značí, že skupina operací musí proběhnout způsobem „všechno nebo nic“. Každá operace je ve skupině jednotlivá („atomická“) a při selhání jediné této operace musí být celá transakce zrušena a nesmí mít dopad na data v databázi.
- **Consistency** (konzistence) značí, že před i po provedení transakce musí být databáze ve validním stavu, nesmí být porušena žádná integritní či jiná pravidla a omezení (neexistující reference, redundance dat, vlastní definované podmínky, apod.).
- **Isolation** (izolovatelnost) znamená, že transakce musí být na sobě nezávislé i v případě, že vyžadují pro svou činnost stejná data (při pouhém čtení to není problém). Nesmí docházet ke konfliktům způsobeným neaktuálnosti části dat, systém musí takové transakce provádět postupně.
- **Durability** (trvalost) zaručuje, že všechny změny provedené transakcí, která je úspěšně dokončena, budou skutečně uloženy do databáze, a to i v případě následných systémových chyb. V praxi je obvykle využíváno transakčních logů.

V praxi jsou transakce splňující ACID řešeny nejčastěji pomocí zámků (Lock) nebo verzování (Multiversioning).

5.2. BASE

BASE **(B**asic **A**vailability, **S**oft State, **E**ventual Consistency) je transakční model vniklý v důsledku obtížné udržitelnosti ACID u distribuovaných databází. Pro mnoho projektů navíc ACID (resp. konzistence) nejsou zcela nezbytné a důraz je zde kladen hlavně na dostupnost (v reálném nebo přiměřeném čase).

Base model je charakterizováván těmito 3 vlastnostmi [31]:

- **Basic Availability** (základní dostupnost) znamená upřednostňování dostupnosti dat před jejich úplnou správností (aktuálností, úplností).
- **Soft State** (přechodný stav) značí, že databáze nemusí být vždy v konzistentním stavu a údržba nekonzistentních dat je přenesena na aplikační vrstvu.
- **Eventual Consistency** (eventuální konzistence) zaručuje, že databáze musí mít možnost přejít do konzistentního stavu, pokud je to požadováno (může to ovšem vyžadovat více času). Pro přechod do konzistentního stavu je rovněž nutné omezit po určitou dobu databázové operace (kromě čtení).

5.3. Distribuované transakce

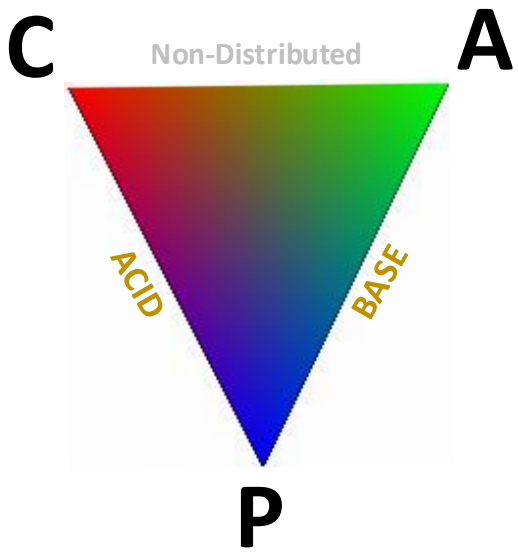
Distribuované databáze splňující ACID pochopitelně musí plně podporovat transakce. Obzvláště udržení konzistence je však při práci s více databázemi mnohem obtížnější a vyžaduje větší spolupráci mezi jednotlivými instancemi. Všechny operace vyžadující více databází (databázových částí) v rámci jedné transakce, je tak nutno operace provádět postupně a vždy musí být potvrzeny oběma zúčastněnými stranami.

Výrobci distribuovaných databázových platforem (především z řad NoSQL) často na první pohled vzbuzují zdání, že splňují ACID [14]. Při podrobnější analýze však vyjde najevo, že se to týká pouze určitých částí systému (například dat), a jiné (například indexace nebo replikace) přitom odpovídají modelu BASE. Přesto i mezi NoSQL existují databáze plně podporující ACID (například CouchDB). [20]

[14] *http://web.archive.org/web/20150320053809/https://foundationdb.com/acid-claims*

Modely ACID a BASE si tak lze představit ne jako dvě striktně oddělené možnosti, ale jako hraniční body, mezi kterými existují různé varianty nakloněné více k jedné či k druhé variantě. [32]

Při volbě databázové platformy tak nestačí jen zvážit, zda je pro cílový systém zásadní konzistence dat, či jejich vysoká dostupnost (i za cenu možných nepřesností), ale do jaké míry jsou pro nás tyto vlastnosti důležité.



Obrázek 24 - CAP (rozložení ACID a BASE)

6. Praktická část

Hlavní náplní praktické části práce je vytvoření aplikace v podobě decentralizovaného systému pro ukládání a sdílení biologických vzorků (vzniklých měřením na hmotnostním spektrometru), založená na lokálních databázích. Tento systém by měl umožňovat jak lokální činnost v rámci jedné organizace, tak sdílení dat mezi různými organizacemi (např. různými univerzitami) zapojenými do systému (prostřednictvím internetu).

Jedná se pouze o koncept pro ověření možností decentralizované distribuované databáze s proměnným počtem uzlů/úložišť. Nebude zde proto příliš brána v úvahu grafická stránka projektu a některé další stránky projektu (např. zabezpečení) budou popsány pouze teoreticky bez implementace.

Vzniklý systém bude následně testován z hlediska výkonu při dotazování s ohledem na různé množství dat, počet úložišť a složitost dotazu.

6.1. Analýza

Tato kapitola obsahuje analýzy nezbytně předcházející implementaci konceptu. Jedná se o analýzu struktury dat biologických vzorků a analýzu funkčních i nefunkčních požadavků, na jejichž základě je následně zvolena vhodná databázová platforma.

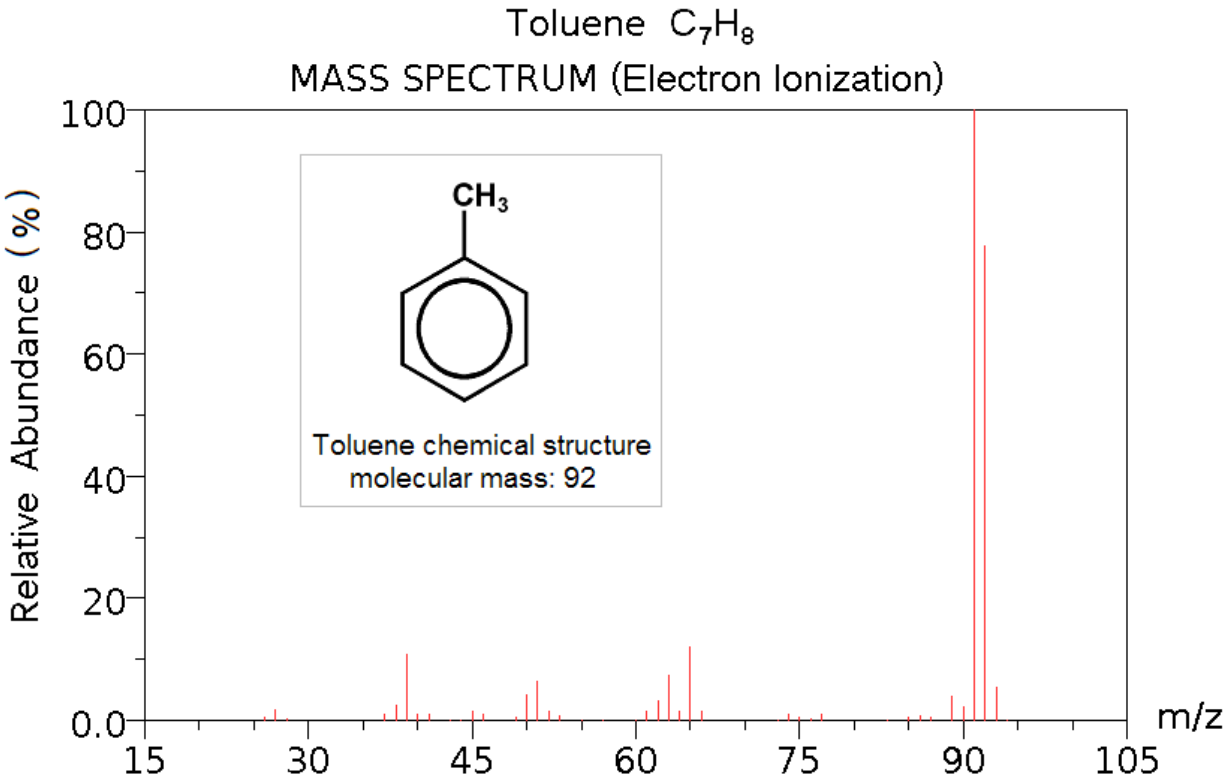
Současně je zde provedena analýza výhod decentralizovaného řešení v porovnání s centralizovaným řešením nebo použití tradičnějšího přístupu komunikace prostřednictvím aplikace (nikoli přímé práce s databázemi).

6.1.1. Struktura biologického vzorku

Hmotnostní spektrometrie je založena na rozdílné hmotnosti peptidů a proteinů. Měřený vzorek je nejdříve odpařen, ionizován, separován na základě hmotností iontů a poté je měřeno zastoupení jednotlivých skupin iontů. [30]

Výsledkem měření je tedy množina hmotností iontů (resp. poměru váhy a náboje) a jejich intensita. Tento vztah je obvykle znázorňován pomocí dvourozměrného grafu, kde osa *x* představuje poměr hmotnosti k náboji (m/z) a osa *y* značí intensitu (v procentech). Z pohledu aplikace tato data lze uchovávat například v podobě dvourozměrného pole nebo kolekce s prvky o 2 proměnných.

Výslednou podobu naměřených dat ovlivňuje jak čistota vzorku, tak přesnost použitého hmotnostního spektrogramu. Z tohoto důvodu je nutnost chápat množinu m/z jako množinu reálných čísel, bez zaokrouhlování či jiných matematických funkcí. Nelze tak vytvořit konečnou množinu všech možných hodnot (číselník).



Obrázek 25 - Příklad vzorku hmotnostní spektrometrie[15]

[15] Převzato z *http://cs.wikipedia.org/wiki/Hmotnostn%C3%AD_spektrometrie*

6.1.2. Funkční požadavky

Modelový systém pro ukládání a sdílení biologických vzorků by měl splňovat následující základní funkční požadavky:

- Systém umožňuje vkládání dat biologických vzorků.
- Společně s daty je možné uložit i binární soubor.
- Systém umožňuje vyhledávání biologických vzorků na základě podobnosti (s předloženým vzorkem).
- Systém zobrazuje náhled vzorků ve smysluplném formátu (např. graf).
- Systém umožňuje přidávat a odebírat databázová úložiště.
- Systém je nezávislý na momentální dostupnosti úložišť.
- Systém je možné využívat paralelně více uživateli současně.

V případě praktického nasazení by bylo vhodné, aby výsledný produkt zahrnoval i následující funkční požadavky:

* Práce se systémem je zabezpečena autentizací jménem a heslem.
* Uživatelé mohou kontrolovat, kdo smí přistupovat k jejich vzorkům (případně zpoplatnění).
* Import/export většího množství vzorků, případně celého jednoho úložiště.
* Kopírování vzorků mezi úložišti.

6.1.3. Nefunkční požadavky

Základními nefunkčními požadavky jsou:

- Databázová platforma je šířena pod open-source licencí. [16]
- Aplikace boudou implementovány v jazyce C# za použití .NET Frameworku. Pro provoz aplikací je tedy nezbytný operační systém Windows s odpovídajícím .NET Frameworkem a webovým serverem IIS.
- Aplikace pro dotazování bude mít podobu webové aplikace ASP.NET MVC 5 a současně bude umožňovat komunikaci prostřednictvím Web API 2.0.
- Databázová platforma pro ukládání informací o úložištích, uživatelích, apod. je shodná s databázovou platformou pro ukládání biologických vzorků.
- Zdrojový kód je srozumitelný a dokumentovaný.

V případě praktického nasazení by bylo vhodné, aby výsledný produkt zahrnoval i následující funkční požadavky:

* Úložiště jsou klonována nebo pravidelně zálohována.
* Komunikace s úložišti je zabezpečena (např. protokolem SSL/TLS).
* Komunikace uživatelů při dotazování je zabezpečena (např. protokolem SSL, resp. HTTPS).

[16] *http://opensource.org/licenses*

6.1.4. Výhody decentralizovaného databázové řešení

Lze předpokládat, že problematiku sdílení dat (zde biologických vzorků) by bylo možné úspěšně řešit pomocí centralizovaného úložiště, nebo by komunikace mezi jednotlivými uzly mohla být řešena na aplikační vrstvě (každý uzel by měl svou aplikaci, která by pracovala s lokální databází). Jaké jsou tedy výhody distribuované databáze s použitím lokální uzlů?

Hlavní výhodou proti centrálnímu řešení je možnost flexibilního vytváření lokálních databází, nad kterými máme plnou kontrolu, a do kterých máme obvykle velmi slušnou konektivitu (pravděpodobně řádově GB/s). Databáze tak můžeme plně spravovat, přidávat a odebírat, rovněž můžeme snadno vytvářet i vlastní logiku - můžeme mít např. oddělenou databázi pro různé typy dat (např. můžeme oddělit rostlinné/živočišné vzorky nebo video/audio soubory apod.), což umožní rychlejší vyhledávání. Díky lokální konektivitě můžeme data nahrávat pohodlně a bez zbytečného čekání. Ve většině případů nás navíc při vyhledávání zajímá, zda data máme my (lokálně), a až v případě, že jimi nedisponujeme, nás zajímá, zda je má někdo jiný (pokud je máme v databázi my, opět s výhodou používáme lokální konektivitu).

V případě komunikace přes aplikační vrstvu můžeme přijít o některé funkčnosti databázové vrstvy, nebo ji budeme problematicky zprostředkovávat. Jedná se především o Map/Reduce, Transformery, nebo pokročilé operace s indexy, které by bylo možné zprostředkovávat jen velmi obtížně. Nadstavbová aplikace by navíc zvýšila obtížnost nasazení a údržby, kde namísto jednoduché instalace databáze a nastavení oprávnění by byly nutné ještě i instalace a nastavení aplikace, při úpravě funkčnosti by pak bylo nutné, aby všichni účastníci své aplikace aktualizovali. Při rozsáhlých výpočtech nad velkým množstvím databází by rovněž další vložený prvek mohl znamenat zbytečné zpomalení komunikace a výpočtů.

Architektura Klient-Databáze (dále v kapitole 6.3) umožňuje velkou variabilitu, kterou tradiční model Klient-Server-Databáze neumožňuje, ale současně tento ani nevylučuje v podobě zprostředkovatele dotazování (přenesení funkce databázového klienta na centrální prvek). Stejně tak není vyloučena implementace aplikací pracujících pouze lokálně (bez znalosti shardingu). Příkladem může být aplikace pro zpracování a přímé ukládání vzorků do konkrétní databáze.

6.1.5. Volba databázové platformy

Existuje nepřeberné množství databázových platforem různých typů, každoročně vznikají nové a většina těch současných se neustále vyvíjí. [31] [20] Klíčovou roli tak hraje v první řadě výběr vhodného typu podle požadavků konkrétního systému.

Vzhledem k funkčním požadavkům a struktuře dat (biologických vzorků) je možné vyvodit následující závěry pro použití konkrétních databázových modelů:

- Jedná se o ukládání spíše oddělených, neprovázaných dat. Databáze, které jsou zaměřeny především na podporu a uspořádání dat pomocí vazeb (např. relační, grafové), či zapouzdření (např. objektově orientované), tak nejsou vhodné s ohledem na vysoké požadavky na škálovatelnost. Naopak většina NoSQL databází bez pevné vnitřní struktury (schema-less) je přímo navržena pro distribuovaná řešení, nebo je aktivně podporuje.

- Data biologických vzorků (pole/kolekce) nemají pevný počet prvků, ani nelze vytvořit číselník hodnot, což komplikuje možnost vyhledávání způsobem klíč-hodnota. Klíčem může být pochopitelně jméno nebo chemický vzorec látky, to nám však neumožní vyhledávat na základě podobnosti. K tomu potřebujeme přístup a práci se samotnými daty jednotlivých vzorků. To umožňují mezi NoSQL nativně téměř výhradně dokumentové databáze, díky vnitřní struktuře ukládaných dokumentů.

Přestože pro tento modelový případ by jistě bylo možné s úspěchem použít různé databázové platformy, byla, po prostudování vlastností různých databází, dokumentací a dalších materiálů, především z webů CodeProject[17] a StackOverflow[18], vybrána databáze RavenDB[19].

[17] *http://www.codeproject.com*

[18] *http://stackoverflow.com*

[19] *http://ravendb.net/*

Klíčové vlastnosti pro výběr RavenDB:

- Dokumentová databáze s možností ukládání binárních dat (jako přílohy nebo pomocí virtuálního souborového úložiště).
- Podpora shardingu řešena tak, že každou databázi (resp. každé úložiště) je možno použít jak samostatně, tak jako součást shardingového clusteru (nebo i více clusterů).
- Kromě komerční a vývojové licence i open-source včetně komunitní i přímé podpory. Minimální omezení u nekomerčních verzí (omezeno zabezpečení databází).
- .Net Client i samotný server napsaný v jazyce C#. Vzhledem k dostupnosti zdrojových kódů je tak zde potenciálně možnost vlastních pluginů, či jiných přímých úprav.

Přestože v různých diskuzích jsou zejména starší verze označovány za pomalejší vůči svým konkurentům (především kvůli platformě .Net a programovacímu jazyku C#), je RavenDB oceňována zejména pro své široké množství funkcí, škálovatelnost a flexibilitu. [32]

6.2. RavenDB

RavenDB (v době psaní této práce ve verzi 3.0.3599) je dokumentově orientovaná databáze od společnosti Hibernating Rhinos Ltd, existující od roku 2010. Mezi její základní vlastnosti a specifika patří [32] [33]:

- Komerční, vývojová a Open-Source (AGPLv3) verze
- .Net Client API, Java Client API a RESTful WebAPI
- Webové rozhraní (Studio) pro správu úložišť a dat
- Podpora transakcí (ACID pro dokumentové operace, BASE pro indexaci a dotazování), podpora distribuovaných transakcí (DTC)
- Ukládání binárních souborů (jako přílohy nebo na samostatném file serveru)
- Podpora LINQ a Transformerů
- Podpora shardingu a replikace
- Podpora Map/Reduce
- Fulltextové vyhledávání za pomocí Lucene engine
- Embedded mode
- InMemory mode (pro testování)

6.2.1. Instalace

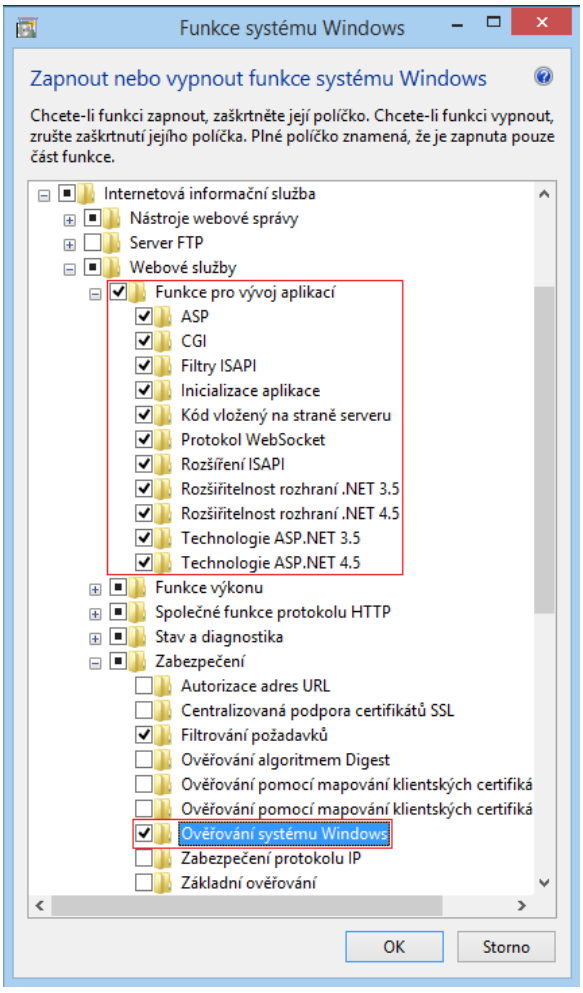
Způsob instalace je závislý na požadovaném instančním módu. RavenDB může být použita v jednom ze 4 instančních módů: Debug mode, Embedded mode, Service mode, IIS aplication mode. [34]

Debug mode je spuštění serveru bez instalace pomocí souboru *Start.cmd* dostupného v instalačním balíku. Po spuštění je databáze zaregistrována na nejbližším volném portu od 8080 výše a k dispozici jsou všechny funkce včetně Studia. Tento mód není vhodný pro ostré nasazení.

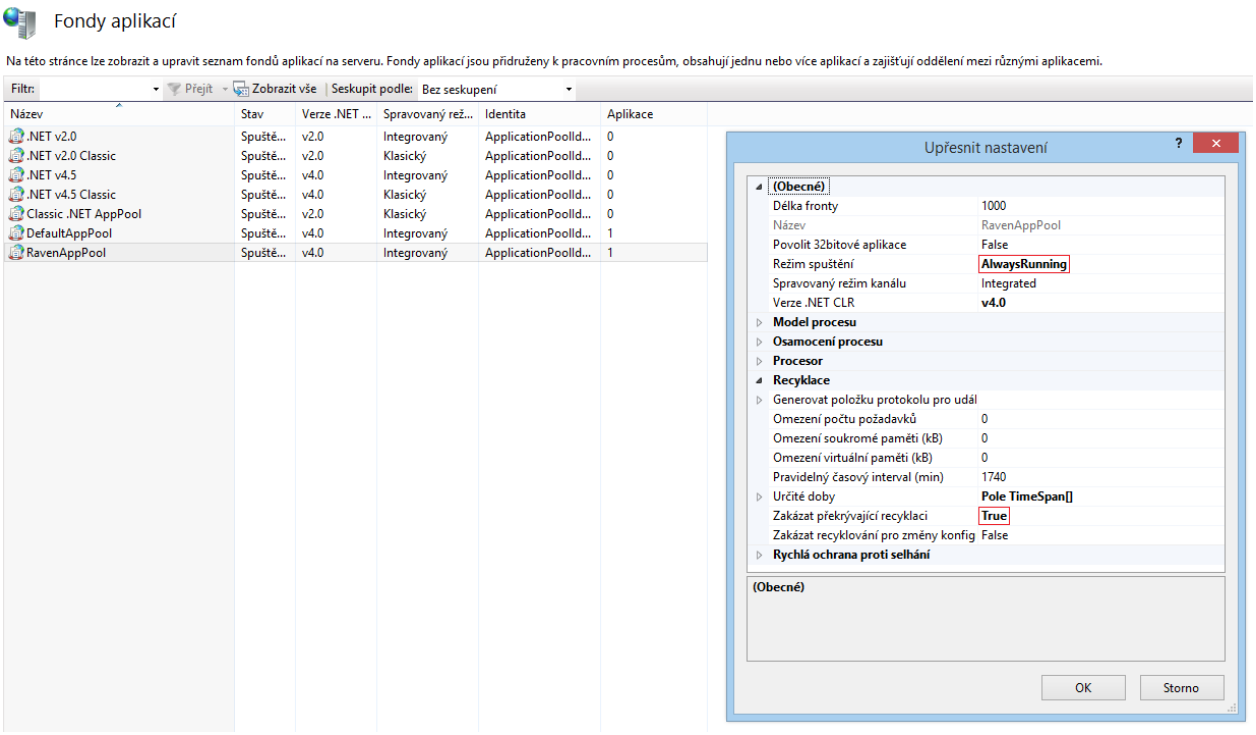
Embedded mode (vložená instance) umožňuje vytvoření instance přímo v rámci konkrétní aplikace. Instalace zde spočívá v nastavení potřebných referencí na knihovny a poskytnutí cesty k fyzickému uložení databáze. Přestože jsou v embedded módu ve výchozím nastavení WebAPI pomocí HTTP protokolu vypnuty, je možné je povolit, a využívat tak většinu funkcí včetně Studia a shardingu.

Service mode umožňuje spuštění databázové instance jako služby systému Windows. K instalaci je zde možné použít instalační soubory a instalaci provést pomocí příkazové řádky nebo je možné využít instalátor. V případě Service módu je implicitně vytvořen vlastní webový server pro běh WebAPI a Studia.

IIS application mode slouží pro správu instance databáze v rámci existujícího IIS Serveru a jeho application pools. Tato varianta je ekvivalentní k Service mode s tím rozdílem, že lze využít dalších vlastností, správy a nastavení (např. zabezpečení) IIS serveru. Pro funkční instalaci je však nutné nainstalovat některé funkce IIS a provést dodatečná nastavení, přičemž některé tyto kroky nejsou bohužel v dokumentaci uvedeny (nebo jen částečně).



Obrázek 26 - Součásti IIS



Obrázek 27 - IIS nastavení fondu aplikací (application pool)

6.2.2. Sharding v RavenDB

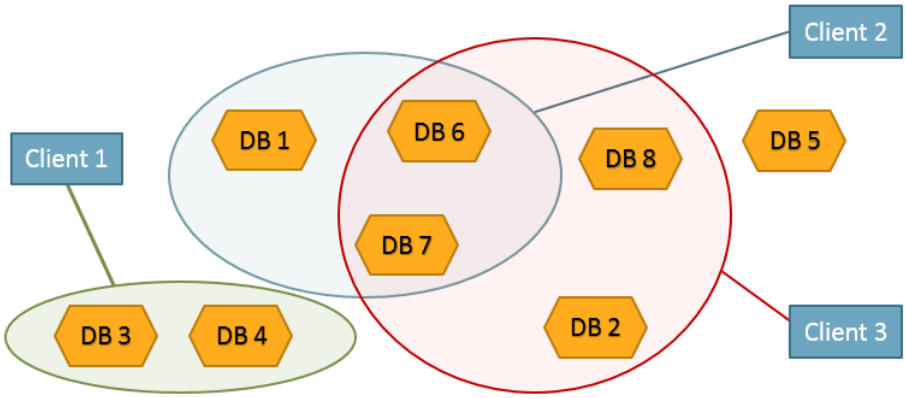
Jednou ze zásadních vlastností RavenDB, jakožto NoSQL databáze, je podpora shardingu[20], kterou lze navíc kombinovat s replikací (mirroringem). Odlišností proti jiným databázovým systémům je zde to, že každá databáze může prakticky sloužit jak individuálně, tak jako shard – součást shardingového clusteru.

Dalším podstatným rozdílem je, že zde není zapotřebí žádný management server, ale jeho funkci přebírá klientská aplikace. To umožňuje si „na požádání“ vytvořit shardingový cluster z jednotlivých databází a začít s ním pracovat. Jedna databáze/shard může být navíc součástí více clusterů, které mohou velmi snadno vznikat a zanikat. Pro vytvoření clusteru je prakticky zapotřebí pouze patřičné oprávnění a definice shardingové strategie (lze použít pomocnou metodu ShardingOn, využívající jednoduchou *master shard index table* s indexací, nebo je možné vytvořit zcela vlastní strategii).

Nutné je však podotknout, že sharding v RavenDB nikdy nebyl primárně určen pro takovéto použití (předpokládané využití je tradiční geografický cluster s víceméně pevným počtem

[20] *http://ravendb.net/docs/article-page/3.0/csharp/server/scaling-out/sharding/how-to-setup-sharding*

shardů). Velmi neobvyklé je rovněž využití jednoho shardu v teoreticky více shardingových clusterech ve stejný okamžik. Pro funkční řešení je tak minimálně nutné, aby databáze obsahovaly očekávaný stejný typ dat se známou indexací (statický index). Přestože jednoduchou indexaci je možné vytvářet automaticky na základě dotazu (dynamický auto-index[21]), v případě složitějších indexů je nutné je do databáze nahrát. Nová indexace (včetně auto-indexace) na větším množství dat pak pochopitelně vyžaduje čas na zpracování, po který není možné se dotazovat (resp. protože indexace probíhá paralelně na pozadí, je možné se dotazovat, ale jen na již indexované položky).



Obrázek 28 - Shardingový cluster "na požádání"

6.2.3. Binární data

RavenDB umožňuje od verze 3.0 dva způsoby ukládání binárních dat (souborů), které mohou být použity i současně.

První metodou jsou přílohy (attachments[22]), které umožňují připojení souboru přímo k ukládanému objektu (strukturovaná data). Tato metoda je však již považována za zastaralou a funkčně omezenou. Je tak vhodná spíše pro drobná data, s kterými nechceme jinak příliš pracovat. Tyto přílohy, jak již název napovídá, je možné si představit například jako přílohy v emailu. Pro více detailů o použití je v dokumentaci nutné přepnout na starší verzi RavenDB (např. 2.5).

[21] *http://ravendb.net/docs/article-page/3.0/csharp/indexes/creating-and-deploying*

[22] *http://ravendb.net/docs/article-page/3.0/http/client-api/commands/attachments/put*

Druhou metodou, která byla představena právě ve verzi 3.0 je samostatná souborová databáze (File System[23]). Jedná se o virtuální souborový systém, který představuje samostatnou databázi/databáze na stejné úrovni jako databáze pro strukturovaná data. Uložené soubory je možné vyhledávat pomocí defaultních (jméno, velikost, datum vytvoření, apod.) nebo vlastních metadat. Na tyto soubory je možné pochopitelně odkazovat ze strukturovaných dat. Rovněž je možné se soubory (na rozdíl od příloh) provádět další operace, například synchronizaci mezi úložišti.

Aktuálně není bohužel možné použít sharding přímo na souborové úložiště, i když se lze domnívat, že by tato funkcionalita mohla být implementována. Rozložení dat (shardingová strategie) by přitom bylo založeno na uživatelských metadatech, která mají stejný formát (JSON) jako strukturovaná data v dokumentovém úložišti. V nejnovější verzi je již dokonce sharding podporován v podobě shardingového clienta (AsyncShardedFilesServerClient[24]). Ten má ovšem v současné době poměrně omezené funkce a jeho dokumentace se teprve připravuje.

6.3. Architektura Klient – Databáze

V systému postaveném na lokálních databázích s klienty, kteří mohou sami sestavovat shardingové clustery (každý klient zde zastupuje roli management serveru), můžeme specifikovat následující prvky:

Databáze[25] představuje jednu instanci databázové platformy (v tomto případě RavenDB). Databáze může obsahovat více dokumentových úložišť (document storage) či souborových úložišť (file storage/file system), které se podílejí na shardingu.

Databázový server obsahuje jednotlivé databáze, přičemž na jednom serveru může běžet i více instancí databázové platformy, dostupných na různých portech, a každá může obsahovat více úložišť. Konečné množství úložišť je závislé na výkonu serveru a množství dostupného úložného

[23] *http://ravendb.net/docs/article-page/3.0/csharp/file-system/what-is-raven-fs*

[24] *https://github.com/ayende/ravendb/blob/master/Raven.Tests.FileSystem/Shard/SimpleSharding.cs*

[25] Pojmy databáze a úložiště jsou často zaměňovány, v tomto kontextu však databáze značí skupinu jednoho či více dokumentových a souborových úložišť.

prostoru. Ekvivalentně je tak možné využít v závislosti na potřebách a možnostech různých organizací jak výkonné „super servery“, například s použitím virtualizace, tak rozložení na větší množství méně výkonných serverů s nižším počtem databází.

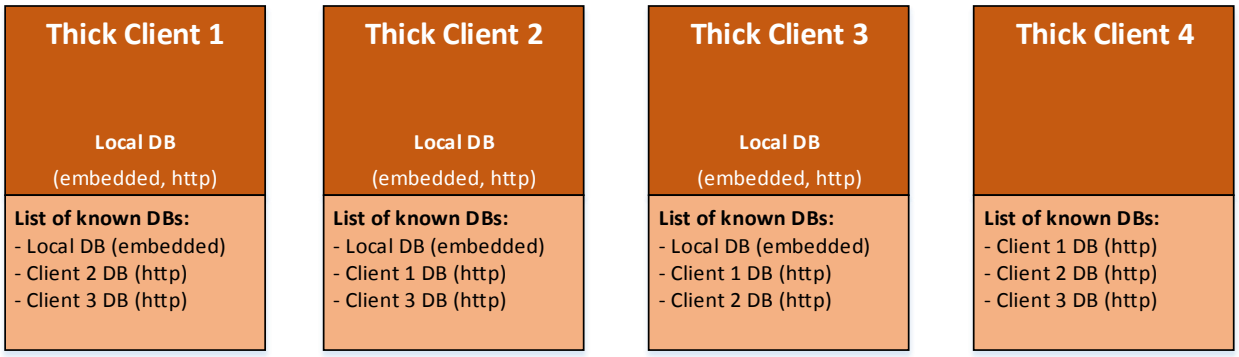
Klient (databázový klient) poskytuje základní operace vkládání dat a vyhledávání. Vzhledem k absenci management serveru musí právě klient znát údaje o dostupných databázích a jejich úložištích/shardech a musí mu být umožněno s těmito pracovat. Klient může mít jak podobu desktopové aplikace (thick client), tak aplikace webové (thin client). Klient může rovněž vykonávat funkci serveru zprostředkovávající požadavky jiných klientů – typickým příkladem může být webová aplikace s WebAPI.

6.3.1. Modelové situace

Díky možnosti sestavování shardingového clusteru přímo na úrovni aplikace je výsledná architektura systému velmi variabilní. Záleží tak především na konkrétních požadavcích, zda je potřeba ukládat data přímo u klientů (embedded databáze), zda budou databáze přístupné přímo (klient - databáze) nebo zprostředkovaně (koncový klient – databázový klient – databáze), což se více podobá klasické architektuře klient – server – databáze. Dále je nutné vzít v úvahu zabezpečení a dostupnost jednotlivých databází. Níže jsou popsány příklady modelových situací.

Model 1

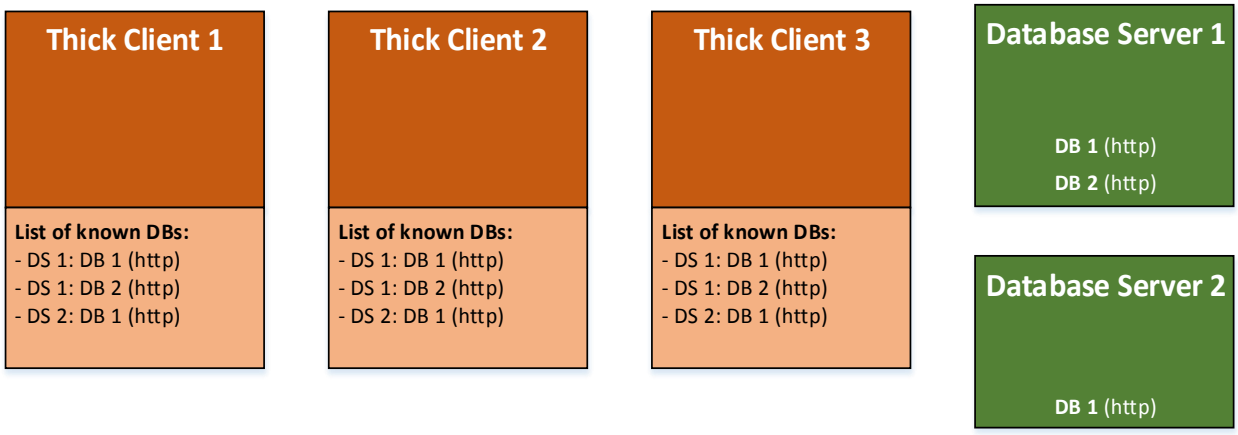
Nejjednodušším modelem jsou klienti s embedded databázemi a povoleným WebAPI. Každý klient musí pochopitelně znát údaje o ostatních databázích a ne všichni klienti musí mít vlastní databáze. Typickým příkladem může být desktopová aplikace s konfiguračním souborem (například xml), obsahující údaje o databázích (a jejich úložištích).



Obrázek 29 - Model 1

Model 2

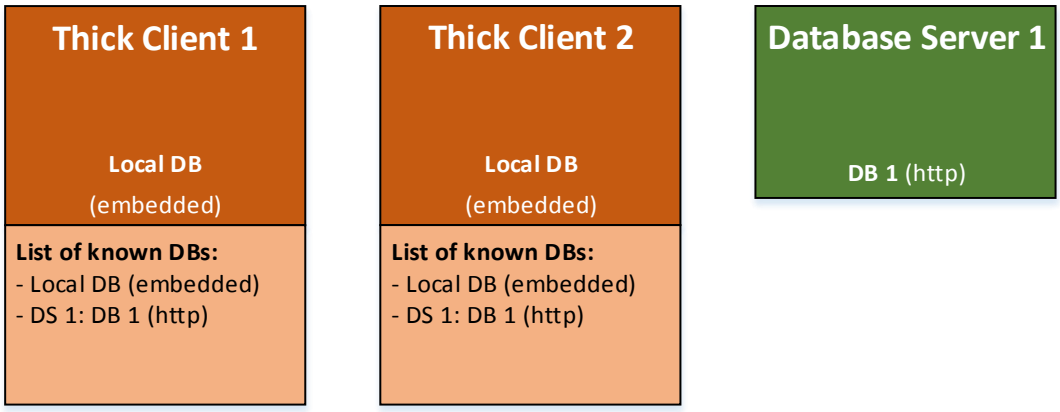
Dalším jednoduchým modelem jsou klienti s oddělenou databází/databázemi (Service / IIS app mode). Databáze může být přitom spuštěna na stejném fyzickém počítači jako některý klient, nebo na zcela odděleném serveru.



Obrázek 30 - Model 2

Model 1+2

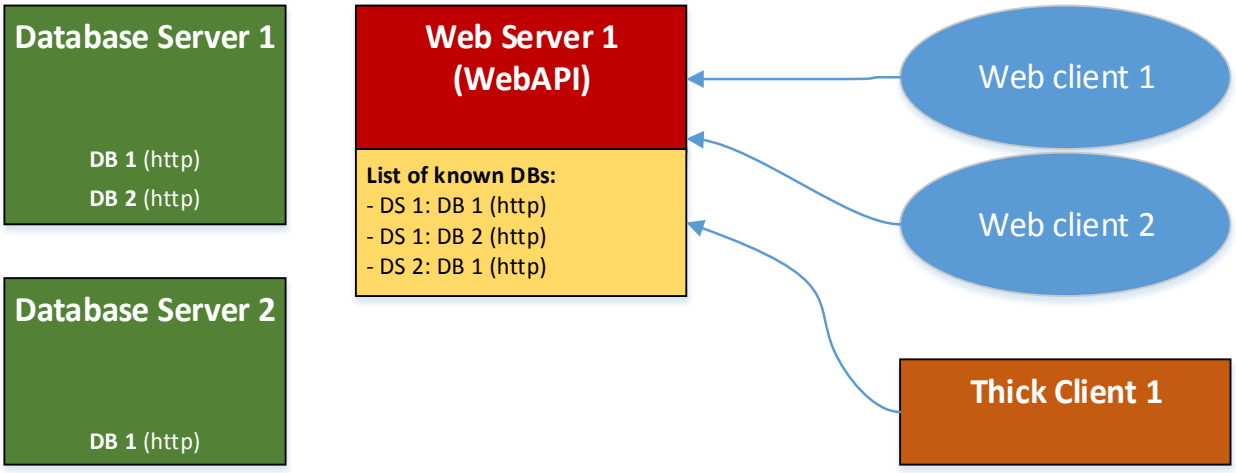
Modelové situace 1 a 2 je možné kombinovat. Příkladem zde může být situace, kdy si uživatel chce uchovávat svá data nezávisle (nesdílet), ale současně chce využívat firemní databázi.



Obrázek 31 - Model 1+2

Model 3

Klient (resp. databázový klient) může mít rovněž podobu serveru, který zprostředkovává požadavky koncových klientů. Typickým příkladem může být webová aplikace. Koncový klient přitom může mít jak podobu webového prohlížeče, tak při použití WebAPI i desktopové aplikace.



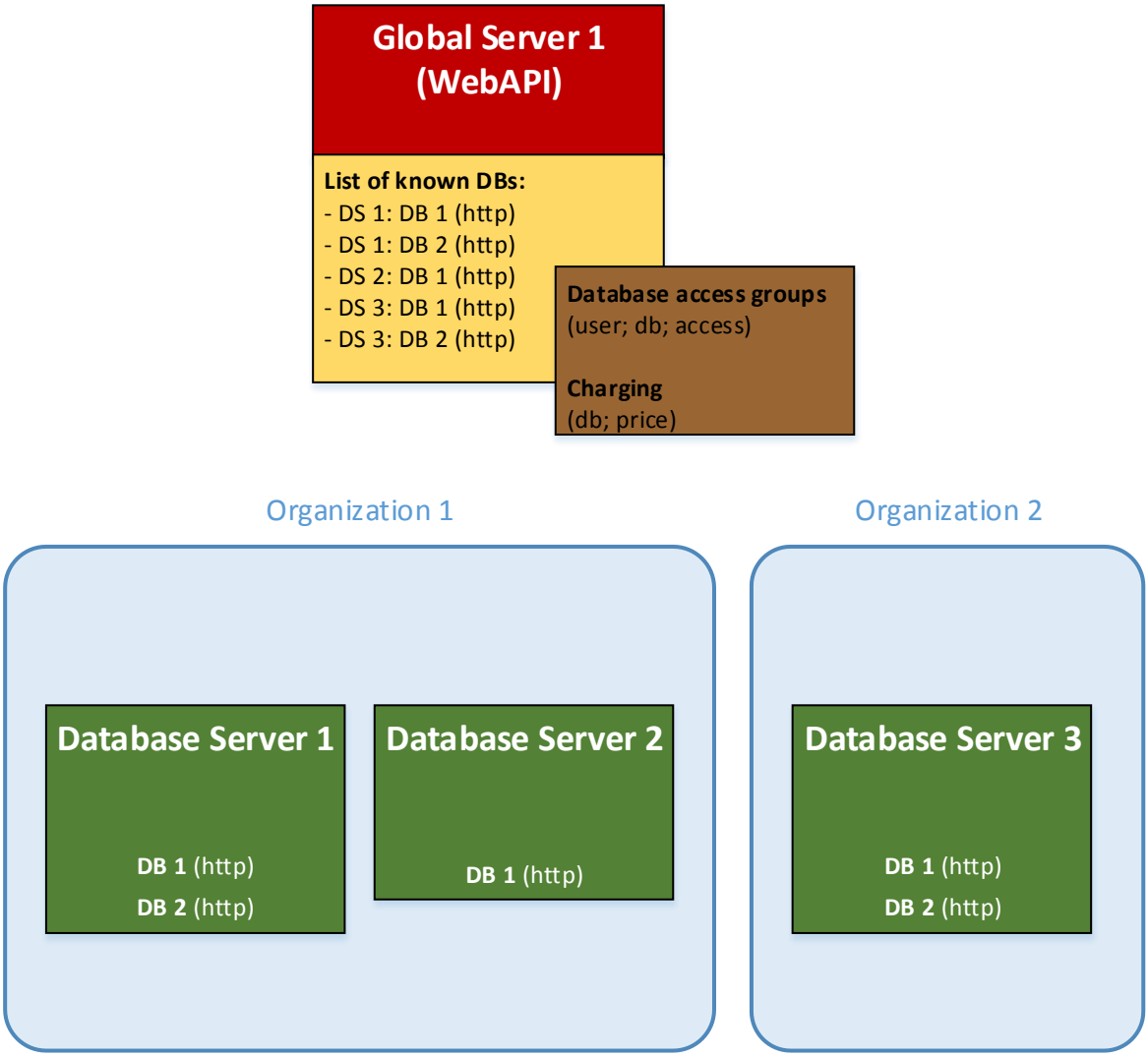
Obrázek 32 - Model 3

Desktopová aplikace by opět mohla obsahovat i embedded databázi. Ta by ale musela být používána nezávisle na databázích, s kterými pracuje webový server (sharding přes prostředníka není možný).

Teoreticky by bylo možné, aby webový server poskytl údaje o databázových serverech, to je ale obvykle nežádoucí, a představuje to bezpečnostní riziko. Koncový klient by navíc musel mít k databázovým serverům přímý přístup (databázové servery by musely být veřejně dostupné).

Model 4

Vezmeme-li v úvahu, že data potřebujeme sdílet napříč organizacemi, můžeme tuto situaci řešit pomocí společného klienta (důvěryhodné autority), které povolíme přístup do naší databáze. Tento globální server bude zprostředkovávat dotazy různých organizací, současně může obsahovat i údaje o tom, kdo má mít které databáze k dispozici (případně může obsahovat i zpoplatnění pro přístup apod.).



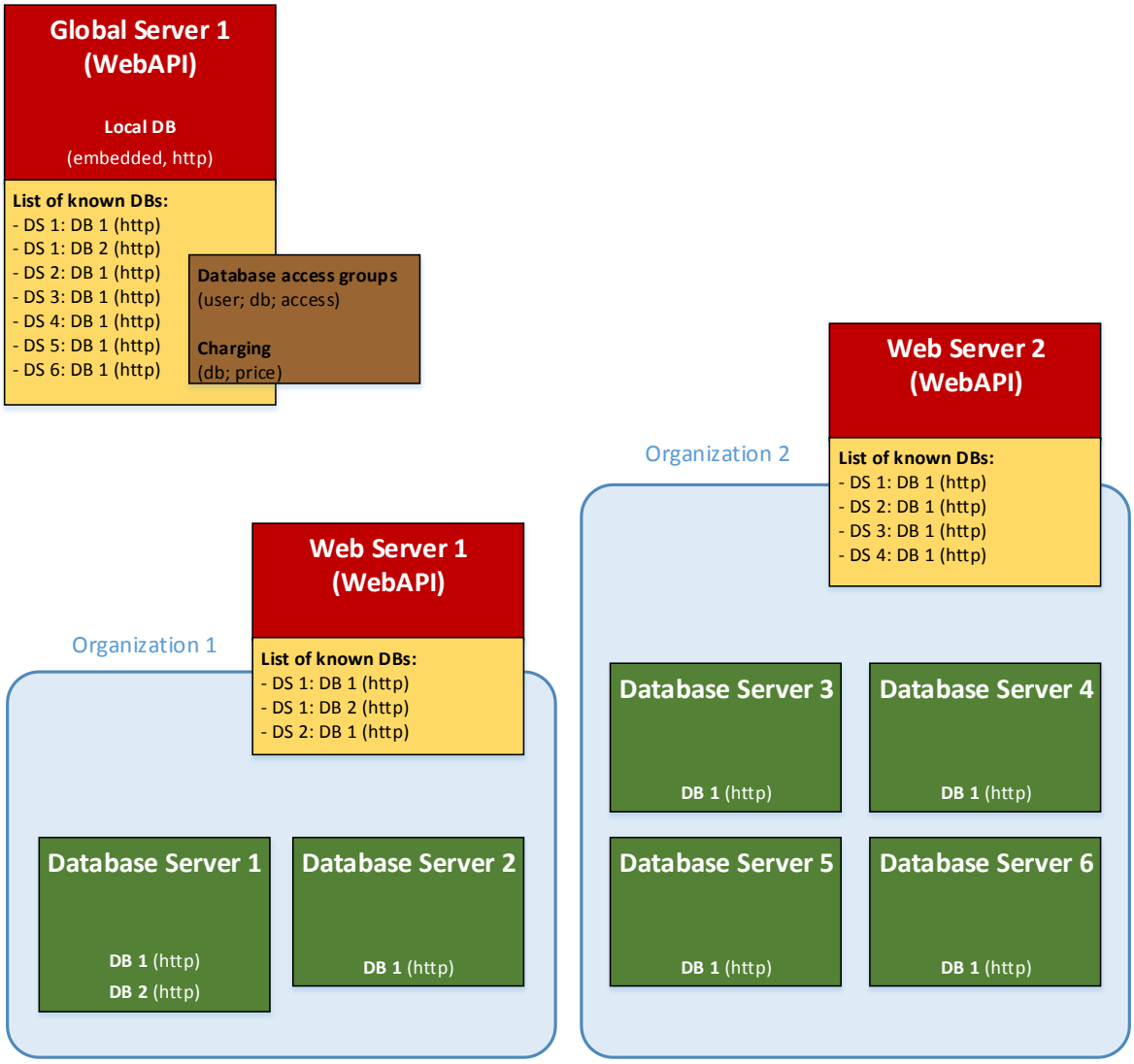
Obrázek 33 - Model 4

Tento přístup již pochopitelně vyžaduje, aby databázové servery byly veřejně dostupné (pravděpodobně přes internet). Je zde nicméně možné omezit přístup k databázím například podle IP adresy. Samotné dotazování na globální server je opět realizováno pomocí webového prohlížeče nebo desktopové aplikace za použití WebAPI.

Model 4+

Použití globálního serveru nevylučuje použití místního serveru, který bude zprostředkovávat dotazy pouze nad databázemi konkrétní organizace (nebo skupiny – např. fakult). Přestože by měl globální server umožňovat zúžení procházených databází podle potřeby (navíc kromě zúžení dle povoleného přístupu uživatelů), může být často vhodné současné použití vlastního zprostředkovatelského serveru, především v případě většího množství vlastních databází.

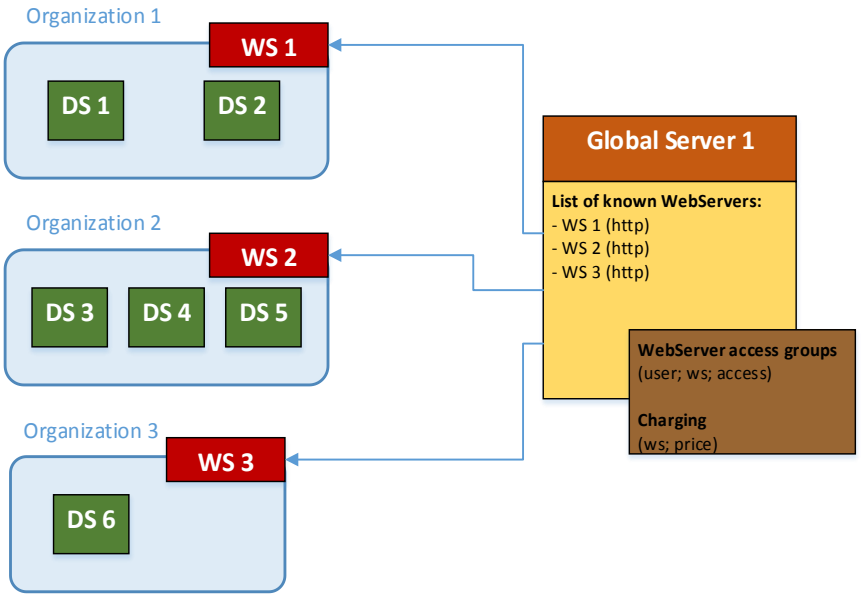
Globální server může rovněž obsahovat vlastní databázi/databáze, která usnadní sdílení dat, případně poslouží jako úložiště pro organizace s malým množstvím dat (bez vlastního databázového serveru). Tento model je pravděpodobně nejvhodnější pro obvyklé reálné nasazení.



Obrázek 34 - Model 4+

Model 5

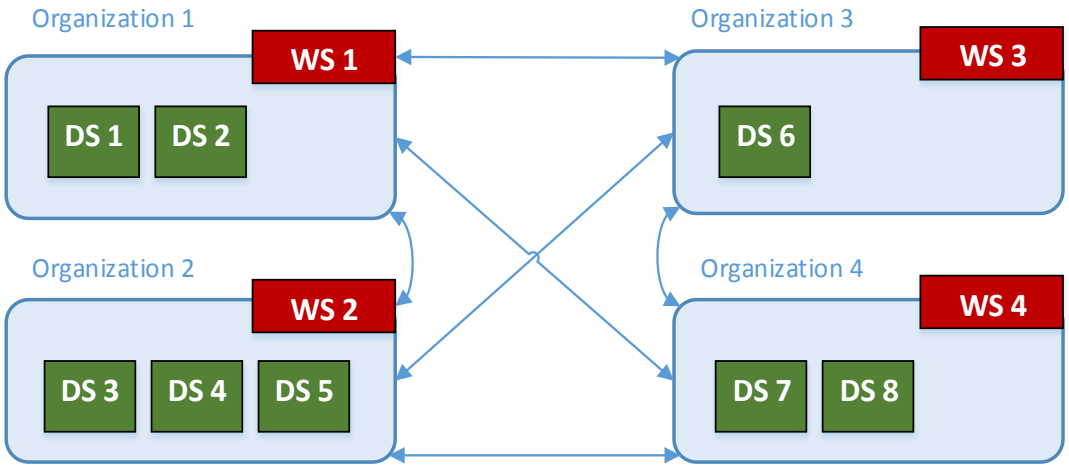
Chceme-li se vyhnout použití globálního serveru, můžeme i při potřebě sdílení dat napříč organizacemi využít možnosti zprostředkování pouze přes vlastní servery (*Model 3*), které budou veřejně přístupné. V tomto případě je však v rámci uživatelského komfortu vhodné při větším množství organizací (serverů) nějakým způsobem zprostředkovat dotazování a výsledky agregovat. To je možné buď přímo pomocí koncové desktopové aplikace (ta by ale musela mít k dispozici aktuální údaje o serverech), nebo opět pomocí globálního serveru. Ten by ovšem (na rozdíl od *Model 4*) neměl přístup k samotným databázím, ale pouze by komunikoval se servery jednotlivých organizací.



Obrázek 35 - Model 5 (s globálním serverem)

Další možností je aplikování některého z peer-to-peer algoritmů pro procházení na úrovni místních serverů (například obdobu floodingu). Tento postup by však vyžadoval tvorbu dalších struktur a logiky.

Zásadní nevýhodou tohoto modelu zůstává nemožnost přímého přístupu k databázím, což může výrazně omezit některé funkce poskytované na databázové úrovni, nebo může tento postup vést k výraznému zpomalení při práci s velkým množstvím databází.



Obrázek 36 - Model 5 (Peer-to-Peer)

6.3.2. Shrnutí

Výše uvedené modelové situace zdaleka nepokrývají všechny způsoby užití architektury „klient – databáze“, potažmo „koncový klient – databázový klient (server/zprostředkovatel) – databáze“. Další variabilitu přitom umožňuje aplikace různých peer-to-peer mechanizmů a jiných aplikačních nadstaveb nad databázové klienty. Ty přitom nemusí mít nezbytně podobu webových aplikací a komunikovat prostřednictvím http protokolu.

Při reálném nasazení by pravděpodobně přibyly i další systémy přímo nesouvisející s distribucí dat. Například pro výhradně lokální nahrávání dat nemá sharding význam, a data tak mohou být nahrávána přímo z aplikace mimo klienta pro dotazování. Naopak pokud bychom vyžadovali často nějakou operaci nad větším množstvím/všemi databázemi (operacemi mohou být různé druhy reportingu, analýz apod.), bude nezbytné uchovávat centrálně údaje o jednotlivých shardech, včetně dalších souvisejících informací (příslušnost k databázi a organizace, typ dat, aj.). Vzhledem k tomu, že dokumentově-orientované databáze jsou pro ukládání takových dat nevhodné, bylo by výhodnější přidat do systému (na úroveň globálního serveru/serverů) i další typ databáze (například relační/grafovou).

6.4. Implementace

Protože cílem této práce je vytvoření decentralizovaného distribuovaného modelu pro více organizací, byl pro implementaci zvolen zjednodušený *Model 4*, kde organizace budou simulovány pomocí různě umístěných skupin databázových serverů, při komunikaci s webovou aplikací (databázovým klientem), přes různé poskytovatele připojení k internetu (více podrobností v části *Testování*).

Přidávání a odebírání úložišť je uživatelům umožněno v rámci webové aplikace, která rovněž umožňuje vyhledávání v těchto úložištích (operace jsou dostupné i přes WebAPI).

Tato kapitola dále popisuje hlavní body implementace modelového systému pro sdílení biologických vzorků s použitím platformy RavenDB.

6.4.1. Shardingová strategie

Přestože RavenDB umožňuje vytvoření vlastní shardingové strategie, většina projektů si pravděpodobně vystačí s pomocnou metodou ShardingOn, která umožňuje distribuci dat mezi shardy na základě jednoho z atributů ukládaného objektu/dokumentu. [26] Tímto atributem může být pochopitelně i proměnná vytvořená pouze za účelem určení cílového shardu (v našem případě se jedná o atribut Shard v objektu Sample – viz. *Struktura dokumentu*).

Hlavní problém v rámci shardingu představuje fakt, že všechny databáze nebudou vždy k dispozici, a vzhledem k tomu, že databáze si sami spravují jednotlivé organizace, nemá nad nimi klient žádnou kontrolu. Nabízí se zde možnost testovat dostupnost před samotným dotazováním, to ale není spolehlivé (k výpadku může dojít i v průběhu dotazu), navíc by tato akce představovala zbytečnou prodlevu.

Efektivnějším řešením je níže znázorněná úprava metod OnError a OnAsyncError, která umožní pokračování dotazu i v případě selhání/nedostupnosti shardu. [35] Tento postup je v rozporu s tradičním použitím shardingu (geografický cluster), kde nedostupnost shardu znamená nedostupnost některých dat a tudíž nekonzistenci.

```
// Apply sharding strategy.
shardStrategy = new ShardStrategy(shardStores)
	.ShardingOn<Sample>(s => s.Shard);

// Sequential or parallel shard access.
shardStrategy.ShardAccessStrategy = new ParallelShardAccessStrategy();

// Setting OnError to continue when shard is unavailable.
shardStrategy.ShardAccessStrategy.OnError += (failingCommands, requestData, exception) =>
{
	if (requestData.Query == null) // query by id (cannot be handled)
	{
		return false; // raise error
	}
	ErrorLog(exception);
	return true; // continue (return and continue with other shard)
};

// Setting OnError for asynchronous shard access.
shardingStrategy.ShardAccessStrategy.OnAsyncError += (commands, request, exception) =>
	request.Query != null;
```

[26] *http://ravendb.net/docs/article-page/3.0/csharp/server/scaling-out/sharding/how-to-setup-sharding*

6.4.2. Struktura dokumentu

Z analýz *Struktura biologického vzorku* a *Funkční požadavky* je možné vyvodit následující strukturu biologického vzorku ukládaného v podobě dokumentu ve formátu JSON:

Sample

```
{
	"Name": "name_of_sample",
	"Formula": "chemical_formula",
	"Author": null,
	"Company": null,
	"Shard": "shard_to_store",
	"OriginShard": „original_shard“,
	"Peaks":
	[
		{"M":10.51,"V":128},
		{"M":14.2,"V":90},
		{"M":15.3,"V":171},
		{"M":29.95,"V":194},
		{"M":36.62,"V":3},
		{"M":37.1,"V":62},
		{"M":39.48,"V":78},
		{"M":58,"V":48},
		{"M":71,"V":167},
		{"M":79.13,"V":85}
	],
	„File_server“ : „file_server“
	„File“ : „fs_reference“
}
```

Atribut *Shard* zde představuje konkrétní úložiště, na kterém je vzorek uložen. *OriginShard* pak slouží pro informativní účely například při použití replikace mezi nody.

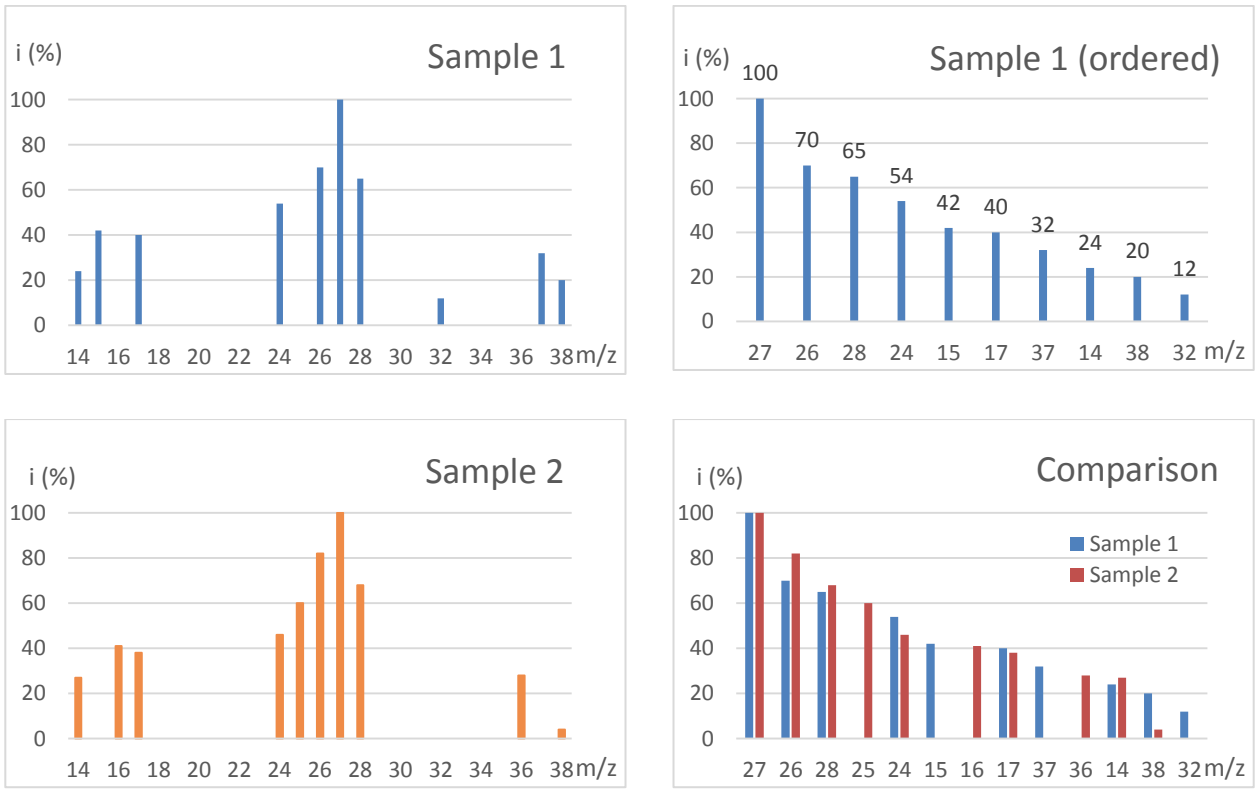
Hlavní složkou je atribut-pole *Peaks*, které představuje jednotlivé vrcholy vzorku. Tyto vrcholy budou použity jako klíč při vyhledávání. V závislosti na metodě vyhledávání nemusí *Peaks* obsahovat všechny vrcholy, ale například jen x nejvyšších vrcholů nebo vrcholy s definovanou minimální výškou. Hodnoty zde mohou být rovněž zaokrouhleny či jinak upraveny.

Atributy *File_server* a *File* slouží jako reference na souborové úložiště a soubor představující skutečná data z měření v podobě binárního souboru.

6.4.3. Indexace a dotazování

Prakticky jediným způsobem vyhledávání dokumentů v dokumentové databázi je za pomoci indexovaných atributů a metody klíč - hodnota. V případě dat v podobě dvourozměrného pole, kde navíc nemůžeme ani pro jedno pole vytvořit číselník, je ale použití této metody obtížnější. Můžeme pochopitelně indexovat a vyhledávat podle jména (atribut *Name*) nebo chemického vzorce látky (atribut *Formula*), tato indexace je však nedostačující, pokud chceme vyhledávat na základě podobnosti (podobnosti naměřených dat).

Existují různé způsoby porovnávání dat hmotnostní spektrometrie. [36] [37] Pro potřeby této práce byl použit zjednodušený princip založený na podobnosti mezi nejvyššími vrcholy. Ty jsou seřazeny od nejvyššího, kde prvních *x* je indexováno, resp. je indexován jejich poměr hmotnosti k náboji (nikoli jejich intensita). Současně je také indexován poměr mezi těmito vrcholy. Dotazování je poté založeno na vyhledávání podobných hodnot se zvolenou tolerancí.



Obrázek 37 - Ukázka porovnávání vzorků

Počet indexovaných vrcholů záleží na zvolené implementaci, nicméně při vyhledávání není nutné využít všechny indexované vrcholy. Příklad kódu pro tuto indexaci 3 vrcholů je následující:

```
Map = samples => from sample in samples
let toppeaks = sample.Peaks.OrderByDescending(p => p.V).Take(3).ToList()
select new
{
       P1 = toppeaks[0].M,
       P2 = toppeaks[1].M,
       P3 = toppeaks[2].M,
       R1 = toppeaks[0].V / toppeaks[1].V,
       R2 = toppeaks[1].V / toppeaks[2].V,
};
```

Indexace probíhá paralelně na pozadí databáze. Při ukládání většího množství dat nebo při nahrání nové indexace tak můžeme provádět vyhledávání, musíme však počítat s možností nedostupnosti dat, která ještě nebyla indexována. Tyto výsledky jsou označeny přívlastkem *Stale*[27]. Právě tento aspekt způsobuje, že přestože data splňují zásady ACID modelu, indexace nesplňuje vždy konzistenci, a odpovídá tak modelu BASE. Nicméně pokud vyžadujeme konzistentní výsledek, můžeme ho dosáhnout pomocí metody *WaitForNonStaleResults…()* za předpokladu, že jsme ochotni na výsledek čekat (podle změn v databázi může být doba čekání od řádově vteřin až po dny).

RavenDB obsahuje 2 hlavní metody pro dotazování na indexované položky. Jedná se o *Query* založeném na dotazovacím jazyce LINQ a o *DocumentQuery*, pracující přímo s API RavenDB databáze.[28] V případě složitějších dotazů je obvykle nutné (nebo vhodné z hlediska množství a srozumitelnosti kódu) použít pokročilé dotazovaní pomocí *DocumentQuery*. Dotaz na výše vytvořenou indexaci pak vypadá takto:

```
simSample => (simTopPeaks, simR1, simR2);
results = session.Advanced  // Advanced is used for direct NonLINQ DocumentQuery methods.
.DocumentQuery<Sample, SamplesByTopPeaks>()
.WhereBetweenOrEqual("P1_Range", simTopPeaks[0].M - tolerance, simTopPeaks[0].M + tolerance)
.WhereBetweenOrEqual("P2_Range", simTopPeaks[1].M - tolerance, simTopPeaks[1].M + tolerance)
.WhereBetweenOrEqual("P3_Range", simTopPeaks[2].M - tolerance, simTopPeaks[2].M + tolerance)
.WhereBetweenOrEqual("R1_Range ", simR1 - ratioTolerance, simR1 + ratioTolerance)
.WhereBetweenOrEqual("R2_Range ", simR2 - ratioTolerance, simR2 + ratioTolerance)
.UsingDefaultOperator(QueryOperator.And) // DocumentQuery uses OR as default
.ToList();
```

[27] *http://ravendb.net/docs/article-page/3.0/csharp/indexes/stale-indexes*

[28] *http://ravendb.net/docs/article-page/3.0/csharp/indexes/querying/query-vs-document-query*

Za povšimnutí zde stojí, že přestože indexujeme proměnné P1 až R2, nyní se dotazujeme na klíč P1_Range až R2_Range. To je dáno tím, že RavenDB používá jako základ indexace Lucene engine [38], a pokud při indexaci objektu zpracovává číselné proměnné (int, double, float, decimal, apod.), indexuje je pod zadaným názvem jako *String* (sloužící především při dotazování na přesnou hodnotu). Současně vytváří index s příponou „_Range“ pro vyhledávání v rozsahu. Přestože by RavenDB měla již od verze 2.5, při použití *DocumentQuery*, rozpoznat dotazy vyžadující ke zpracování číselnou proměnnou (nikoliv *String*), ve výše uvedeném kódu toto fungovalo pouze pro klíče Pn, zatímco Rn zpracovávalo textovou hodnotu. Důvodem je pravděpodobně použití různých číselných proměnných. Příponu „_Range“ je tak vhodné raději uvádět vždy.

Příklady dotazů pomocí Lucene syntaxe (použita rovněž v RavenDB Studiu)[29]:

P1: 21.*	(zpracováno textově)
P1: 21.?5	(zpracováno textově)
P1: [Dx20 TO Dx21]	(zpracováno textově – vrátí hodnoty začínající na 20 nebo 21)
P1_Range: [Dx20 TO Dx21]	(vrátí hodnoty v rozsahu 20 - 21)

Finální podoba dotazu při indexaci 3 vrcholů po přeložení do Lucene syntaxe:

P1_Range: [Dx*P1min* TO Dx*P1max*]

AND P2_Range: [Dx*P2min* TO Dx*P2max*]

AND P3_Range: [Dx*P3min* TO Dx*P3max*]

AND R1_Range: [Dx*R1min* TO Dx*R1max*]

AND R2_Range: [Dx*R2min* TO Dx*R2max*]

* Kurzívou psané řetězce (*P1min*, …) představují hodnoty vzniklé po přičtení tolerance k datům vzorku (simSample), k němuž hledáme shodné/podobné vzorky v databázi.

[29] Více podrobností na *http://lucene.apache.org/core/2_9_4/queryparsersyntax.html*

6.4.4. MapReduce

MapReduce (často uváděno i jako Map/Reduce či Map-Reduce) je paralelní způsob zpracování dat, založený na principu rozdělení výpočtu na menší části (zpracovány paralelně) a následné opakované redukci, sloučení a zpracování výsledků. [43] V RavenDB je funkce MapReduce[30] podporována jak v rámci jednoho úložiště, tak při použití shardingového clusteru.[31]

Mapovací funkce je prováděna pomocí indexace, což mimo jiné umožňuje průběžné zpracování požadovaných hodnot v případě, že se jedná o počítané parametry. Funkce reduce poté při dotazu zpracuje tyto indexované hodnoty (nejprve je zredukuje, a poté provede konečný výpočet[32]).

Níže naleznete jednoduchý příklad, který by umožnil dotazem zjistit například:

- která organizace pořídila nejvíce vzorků konkrétního jména
- kolik je na konkrétním shardu vzorků stejných jmen
- kolik je na jednotlivých shardech vzorků se specifickým nejvyšším vrcholem

```
Map = samples => from sample in samples
                 select new
                 {
                      sample.Name,
                      sample.Company,
                      sample.Shard,
                      TopPeak = sample.Peaks.OrderByDescending.First().M,
                      Count = 1
                 };

Reduce = results => from result in results
                    group result by new
                    {
                        result.Name,
                        result.Company,
                        result.Shard,
                    } into g
                    select new
                    {
                        Name    = g.Key.Name,
                        Company = g.Key.Company,
                        Shard   = g.Key.Shard,
                        TopPeak = g.Sum(x => x.TopPeak),
                        Count   = g.Sum(x => x.Count)
                    };
```

[30] *http://ravendb.net/docs/article-page/3.0/csharp/indexes/map-reduce-indexes*

[31] *http://ayende.com/blog/155361/ravendb-sharding-map-reduce-in-a-cluster*

[32] *http://ayende.com/blog/4435/map-reduce-a-visual-explanation*

Vzhledem k tomu, že s MapReduce se pracuje stejně jako s běžným indexem, je možné ho do databáze nahrát kdykoli a prakticky ihned se začít dotazovat. V případě velkého množství dotazů a složité funkce reduce (či počítaných parametrů) budou výsledky pochopitelně *Stale*, nicméně po indexaci všech existujících dat (asynchronně na pozadí) budou k dispozici kompletní údaje.

Požadovanou MapReduce funkci je rovněž v případě shardingu nutné mít nahranou na každém shardu, s kterým chceme takto pracovat. To lze nicméně zajistit z aplikace po sestavení shardingového cluster snadno. Shardy přitom mohou pojmout teoreticky neomezené množství jak klasických indexů, tak funkcí MapReduce (resp. omezení je dáno výpočetním výkonem serverů).

MapReduce tedy představuje velmi užitečný nástroj, který umožňuje převedení některých výpočtů na databázovou úroveň. Jejich zpracování se provádí paralelně a s předem připravenými parametry. Rovněž tak předchází zbytečně složitým a dlouho zpracovávaným dotazům

6.5. Bezpečnost

Při reálném nasazení by bezpečnost systému pro sdílení dat, založená na databázové platformě RavenDB, byla závislá na zvoleném modelu a zvolené licenci. Obecně však lze shrnout následující možnosti z hlediska zabezpečení komunikace s databázemi a přístupu do nich.

Vzhledem k předpokládanému nasazení RavenDB je defaultně použit protokol http, nicméně při použití certifikátu je možné použít i zabezpečeného protokolu https – v současné době však jen při běhu databáze jako služby (Service mode).

Dalším možným bezpečnostním prvkem je použití firewallu či dalších pokročilých funkcí a zabezpečení, které nabízí operační systém a IIS. V případě modelu s globálním serverem se může jednat například o omezení přístupu podle IP adresy apod.

Samotný přístup do databází je možné zabezpečit autentifikací pomocí jména a hesla. To představuje problém z hlediska správy při sdílení dat mezi více organizacemi, bez použití jednotného, důvěryhodného zprostředkovatele pro dotazování. Další omezení zde plyne z licenčních podmínek, kde vývojová verze je omezena pouze na uživatele admin, open-source pak není možné v některých situacích distribuovat mezi organizace. V některých případech by tak organizace museli jednotlivě žádat o open-source licence nebo zakoupit licenci komerční.

6.6. Testování

Testování probíhalo za použití 18 zapůjčených počítačů a 3 různých bran pro komunikaci, které s použitím VPN simulovali jejich geografické rozložení. Pro zjednodušení byla na jednom PC spuštěna vždy pouze 1 databázová instance s jedním úložištěm.

Konfigurace počítačů (HP Compaq 6000 Pro Small Form Factor PC):

Procesor:	Core 2 Duo E7600
Paměť RAM:	2GB DDR 3 (1333Mhz)
Pevný disk:	ST3320418AS

6.6.1. Generování dat

Vzhledem k tomu, že v průběhu psaní aplikace nebyla k dispozici v dostatečné míře reálná data, a současně vzhledem k zátěžové povaze testu by jejich použití ani nebylo vhodné, byl vytvořen algoritmus pro jejich generování s použitím pseudonáhodných čísel.

Od systému se však očekává vyhledávání na základě podobnosti, z tohoto hlediska by zcela nahodile generovaná data nebyla vhodná. Generování vzorků proto umožňuje ke každému náhodnému vzorku vytvořit libovolné množství podobných (mírně upravených) vzorků.

Popis generování dat:

1) Vytvoření vzorku s nahodilým počtem vrcholů (průměrně kolem 15 vrcholů) s pozicí z rozsahu <0;200>.
2) Přiřazení náhodných, postupně klesajících hodnot vrcholům (nejvyšší má hodnotu 100).
3) Vytvoření podobných vzorků přičtením či odečtením náhodných hodnot z rozsahu <0;2> pro pozici vrcholu a z rozsahu <0;5> pro jeho velikost. Tyto vzorky (včetně zdrojového) mají shodný generovaný název.
4) Uložením vzorků do listu a opakováním od kroku 1, dokud není dosaženo požadovaného počtu vzorků.
5) Uložení všech vzorků z listu do databáze v náhodném pořadí.

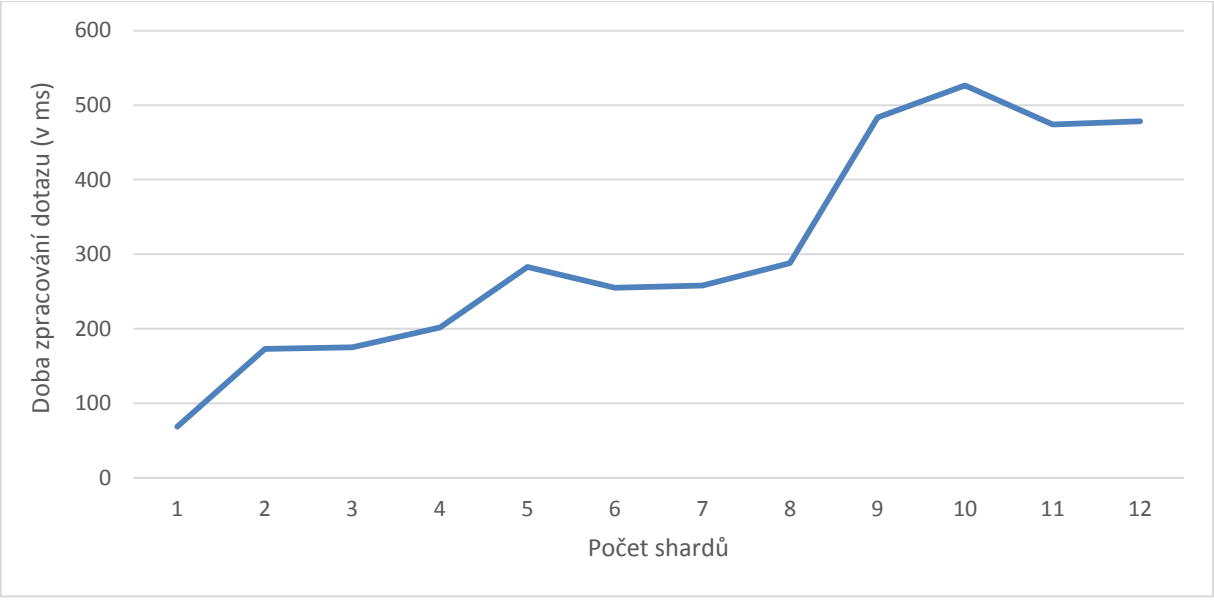
6.6.2. Výsledky měření

Tato kapitola obsahuje souhrnné informace o průběhu a výsledcích testů. Konkrétní data měření je možno nelézt v příloze.

Závislost na počtu shardů

V tomto testu byla zkoumána závislost doby zpracování dotazu na počtu dotazovaných shardů (každý shard je umístěn na jiném počítači). Nejprve byly přidávány počítače s rychlejším připojením. Databázová úložiště obsahovaly stejná data (kopie).

Použité PC: 12 pro databáze, 1 pro dotazování
Způsob připojení: 4 PC lokálně (100Mpbs), 4 PC přes VPN s rychlostí 3,5Mbps a 4 PC přes VPN s rychlostí 2Mbps
Složitost dotazu: Podobnost na základě 3 nejvyšších vrcholů



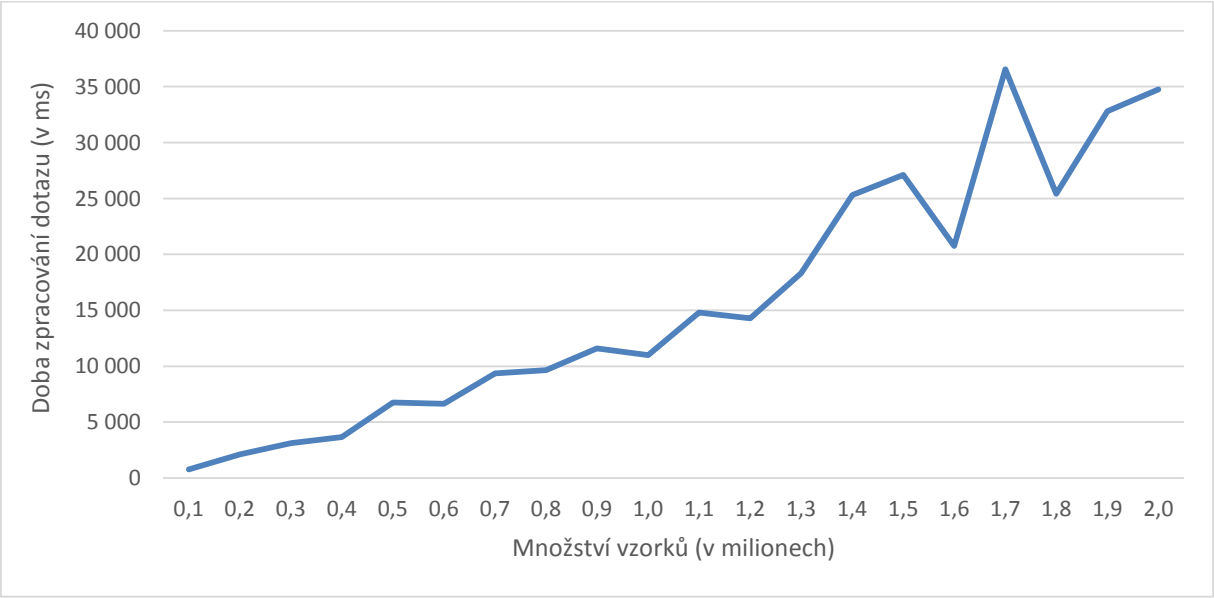
Obrázek 38 - Graf závislosti doby zpracování na počtu shardů (s různou konektivitou)

Z grafu je patrno, že doba zpracování není příliš závislá na počtu shardů, nicméně vzhledem k tomu, že se vždy čeká na výsledky z nejpomalejšího shardu (nebo vypršení maximální stanovené doby), závisí zpracování částečně i na konektivitě, a především pak na rychlosti odpovědi jednotlivých serverů.

Závislost na množství vzorků

V tomto testu byla zkoumána závislost doby požadavku na množství dat v databázích. Maximální doba zpracování požadavku zde byla nastavena na 1 minutu (defaultně 5 sekund), aby nedošlo k ukončení dotazů z důvodu vypršení tohoto limitu. Všechna úložiště obsahovala vždy stejné množství různých vzorků.

Použité PC: 10 pro databáze, 1 pro dotazování
Způsob připojení: 4 PC lokálně (100Mpbs), 4 PC přes VPN s rychlostí 3,5Mbps a 2 PC přes VPN s rychlostí 2Mbps
Složitost dotazu: Podobnost na základě 3 nejvyšších vrcholů



Obrázek 39 - Graf závislosti doby zpracování na množství vzorků

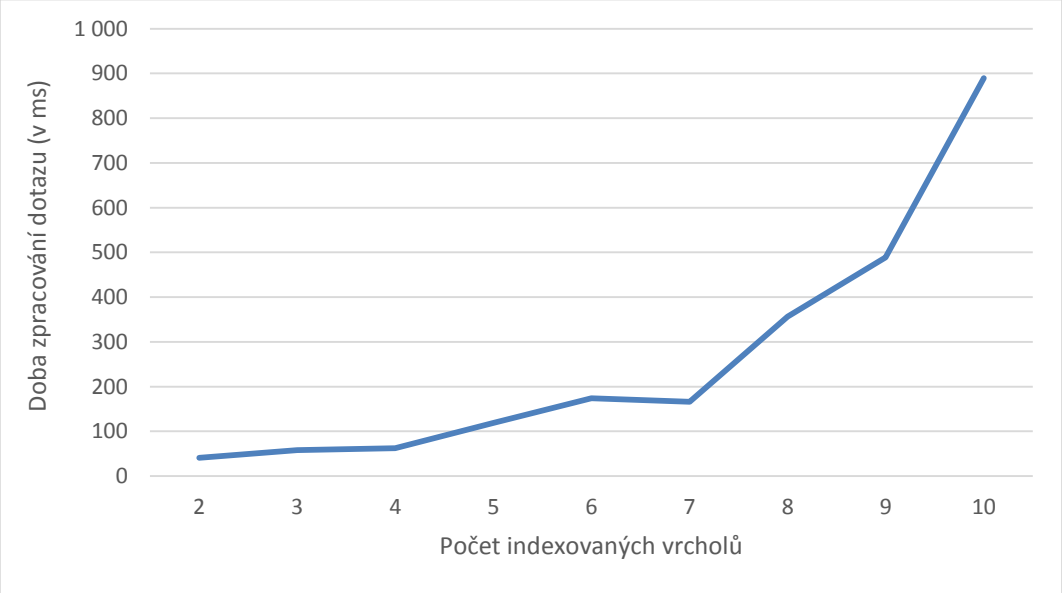
Při množství dat nad 1,3 milionu vzorků se objevily velké rozdíly jak u různých množství dat, tak ale i mezi jednotlivými opakovanými měřeními (každé měření bylo při stejném množství dat provedeno 5x s jiným vzorkem pro vyhledávání). Důvodem je pravděpodobně nedostatek operační paměti, neboť při pozdějším testování na jednom PC s navýšeným množstvím paměti se tento úkaz objevil až při množství dat okolo 2 milionů.

Dotazování s různým počtem indexovaných vrcholů

Tento test se zaměřuje na testování při různé obtížnosti dotazování. Použita je opět metoda dotazování na základě podobnosti, přičemž je množství indexovaných vrcholů a poměru mezi nimi postupně navyšováno (*x* vrcholů + *(x-1)* poměrů mezi každými dvěma sousedními vrcholy).

Použité PC: 5 pro databáze, 1 pro dotazování

Způsob připojení: 5 PC lokálně



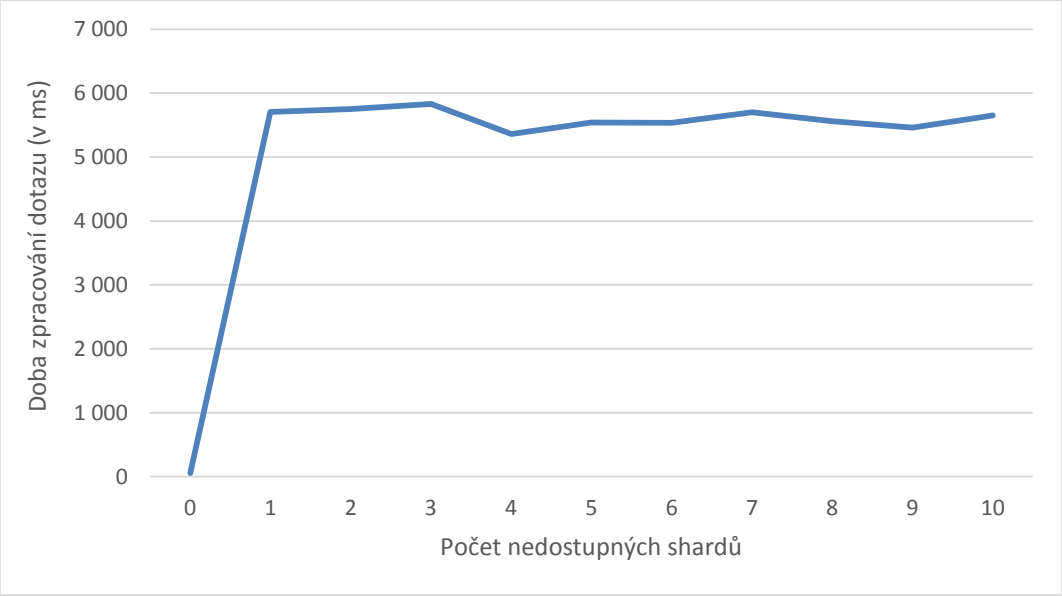
Obrázek 40 - Graf závislosti doby zpracování na počtu indexovaných vrcholů

Z grafu je patrná exponenciální tendence růstu hodnot, což je pravděpodobně způsobeno zvyšující se náročností vyhledávání nad indexovanými hodnotami se složitostí *n*(2n-1)*.

Simulace nedostupnosti shardů

Tento test zkoumá chování při dotazování na shardingový cluster, v kterém není v době dotazování několik shardů dostupných. Tato situace může nastat z důvodu vypnutí serveru, přerušení komunikační linky, softwarové chyby, apod. Maximální doba zpracování požadavku zde byla nastavena na 5 sekund.

Použité PC: 10 pro databáze, 1 pro dotazování
Způsob připojení: 10 PC lokálně (100Mpbs)
Složitost dotazu: Podobnost na základě 3 nejvyšších vrcholů



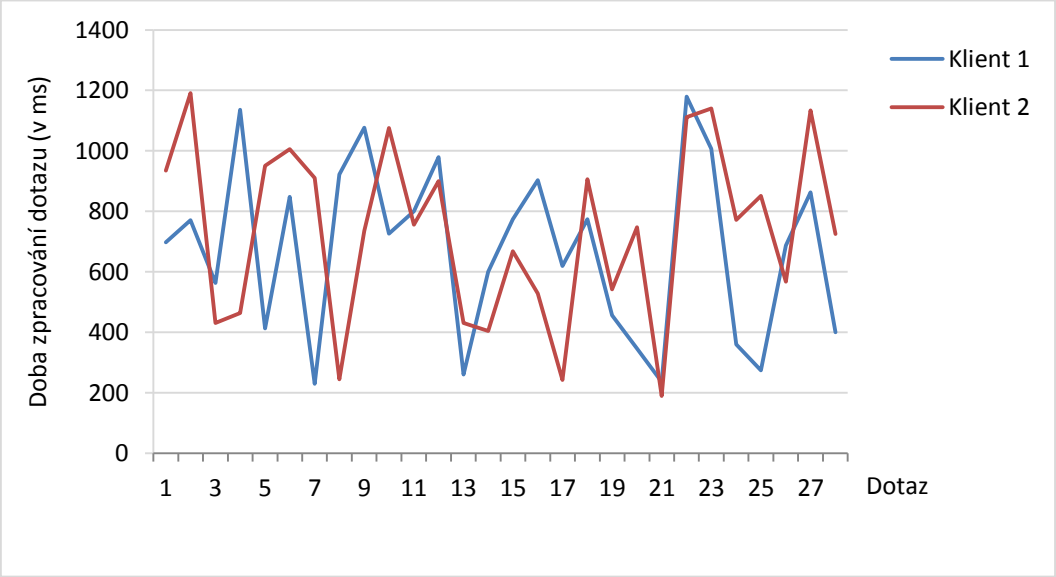
Obrázek 41 - Graf závislosti doby zpracování na počtu nedostupných shardů

Zde je asi nejvíce patrna jedna z nevýhod použití shardingu, a to čekání na odpověď všech dotazovaných databází. Omezení maximální doby zpracování tento problém řeší jen částečně, v praxi by proto bylo vhodné, aby systém například udržoval informace o dostupnosti databází (nebo by test na dostupnost mohl předcházet dotazu na data). Řešení tohoto problému a jeho důležitost je závislá především na konkrétní podobě cílového systému.

Paralelní dotazování klientů na stejný shard

Tento test zkoumá chování databáze při dotazování více klientů současně na stejnou databázi/úložiště. To může nastat velmi často vzhledem k tomu, že každý klient si vytváří svůj shardingový cluster podle potřeby.

Aby bylo možné data porovnat, byla tato situace simulována následujícím způsobem. Jediná aplikace s připravenými dotazy paralelně zadávala současně vždy 2 dotazy (představující 2 klienty) asynchronně každou sekundu, bez ohledu na to, zda databáze již odpověděla na předchozí dotazy. Aby se co nejvíce zabránilo zkreslení síťovou komunikací (aby dotazy byly přijaty téměř současně), byla aplikace i úložiště na stejném počítači. Odpověď na každý dotaz přitom obsahovala jediný vzorek (dotazy a vzorky byly předem připraveny), který navíc obsahoval i číslo shodné s pořadím dotazu a číslo identifikující vlákno (klient 1 nebo 2). Díky tomu bylo možné zpětně určit dobu odpovědi každého požadavku.



Obrázek 42 - Graf znázorňující paralelní dotazování 2 klientů

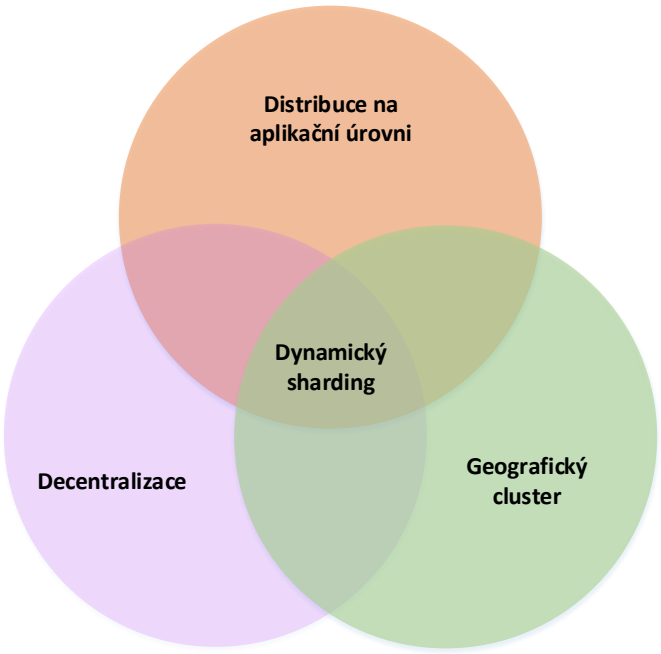
Z grafu je možno vyvodit, že ani paralelní dotazování nepředstavuje problém. Doba zpracování pak bude závislá především na vytíženosti a výkonu serveru s databází.

7. Závěr

Tato práce se věnuje problematice distribuce velkých dat na databázové úrovni. Jejím hlavním cílem bylo vytvoření modelu decentralizovaného systému pro ukládání a sdílení dat s flexibilním počtem databází, a následné ověření tohoto modelu vytvořením konceptu systému pro sdílení a prohledávání biologických vzorků, vzniklých měřením na hmotnostním spektrometru.

V teoretické části práce byly popsány a analyzovány současné možnosti distribuovatelnosti různých databázových modelů (především tradičních relačních databází, NoSQL a NewSQL). Rovněž zde byl popsán vztah distribuovatelnosti těchto modelů a transakcí v souvislosti s CAP Theoremem.

Poznatky z teoretické části byly následně použity pro vytvoření teoretického modelu architektury Klient – Databáze, a popisu modelových případu užití této architektury. Tato architektura byla následně aplikována za použití databázové platformy RavenDB. Použitý model dynamického shardingu (sharding „na požádání“) lze považovat za funkční spojení principů decentralizace, geografického clusteru a distribuce na aplikační (klientské) úrovni.



Obrázek 43 - Dynamický sharding

Výsledkem je vzorový systém umožňující ukládání biologických vzorků, jejich distribuci a sdílení napříč organizacemi. Aplikace rovněž umožňuje vyhledávání na základě podobnosti vzorků - pro tuto funkci bylo nutné vyřešit problém indexace dat s obtížnou definicí klíče.

Vzhledem k úspěšnému otestování na generovaných datech je možné tento koncept považovat za plně funkční a připravený na reálnou implementaci. Rovněž cíle této práce je tak možné považovat za splněné.

8. Definice pojmů a zkratek

DB	Databáze, pro potřeby této práce ve smyslu počítačové databáze.
DBMS	Database Management System (systém řízení báze dat) je software zajišťující práci s databází.
DDBMS	Distributed Database Management Systém.
CRUD	Create, Read, Update, Delete (vytváření, čtení, úpravy, mazání) jsou základní databázové operace prováděné pomocí DBMS.
Datové proměnné	Jednoduché datové typy (string, int, char, bool, apod.).
NoSQL	Not Only SQL (Johan Oskarsson, 2009) [20], původní význam NeSQL (*http://www.dataversity.net/the-nosql-movement-what-is-it/*)
NewSQL	Moderní relační databáze s podporou distribuovatelnosti.

9. Seznam obrázků

10. Seznam tabulek

11. Použité zdroje

1. Big Data. IBM. [Online] http://www.ibm.com/big-data/us/en/.

2. Big Data. System On Line. [Online] http://www.systemonline.cz/clanky/big-data.htm.

3. SELECT * FROM SQL History. FairCom. [Online] http://www.faircom.com/ace/enl_22_s12_t.php.

4. DATAVERSITY. The NoSQL Movement — What is it? [Online] 2012. http://www.dataversity.net/the-nosql-movement-what-is-it/.

5. When NoSQL Makes Sense. InformationWeek. [Online] 2013. http://www.informationweek.com/big-data/software-platforms/when-nosql-makes-sense/d/d-id/1111811.

6. What the heck are you actually using NoSQL for? High Scalability. [Online] 2010. http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html.

7. Charles B. Weinstock, John B. Goodenough. On System Scalability. [Online] 2006. http://www.sei.cmu.edu/reports/06tn012.pdf.

8. Brewer, E. Towards Robust Distributed System. [Online] 2000. https://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

9. NewSQL: what's this? Sogeti Labs. [Online] 2014. http://labs.sogeti.com/newsql-whats/.

10. DNS Made Easy. Vertical And Horizontal Scaling. [Online] 2013. http://www.dnsmadeeasy.com/blog/vertical-and-horizontal-scaling/.

11. ScaleBase. MySQL database sharding. [Online] https://www.scalebase.com/technology/database-sharding/.

12. Amdahl's law. Wikipedia. [Online] http://en.wikipedia.org/wiki/Amdahl%27s_law.

13. MacVittie, Lori. Infrastructure Scalability Pattern: Sharding Sessions. [Online] 2010. https://devcentral.f5.com/articles/infrastructure-scalability-pattern-sharding-sessions.

14. ORACLE. Partitioning Concepts. ORACLE Help Center. [Online] http://docs.oracle.com/cd/B28359_01/server.111/b32024/partition.htm.

15. CodeFutures. Cost-effective Database Scalability using Database Sharding. [Online] 2008. http://codefutures.com/database-sharding-white-paper/.

16. Agarwal, Abhay. Database Sharding. [Online] 2014. http://www.developeriq.in/articles/2014/nov/20/database-sharding/.

17. MySQL Documentation Library. MySQL FEDERATED Storage Engine. [Online] 2010. http://dev.mysql.com/doc/refman/5.0/en/federated-storage-engine.html.

18. Microsoft. Sharding Pattern. MSDN Library. [Online] https://msdn.microsoft.com/cs-cz/library/dn589797.aspx.

19. ScaleBase. [Online] https://www.scalebase.com/.

20. Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, Miriam AM Capretz. Data management in cloud environments: NoSQL. Journal of Cloud Computing. [Online] 2013. http://www.journalofcloudcomputing.com/content/pdf/2192-113X-2-22.pdf.

21. NoSQL. [Online] http://nosql-database.org/.

22. DigitalOcean. A Comparison Of NoSQL Database Management Systems And Models. [Online] 2014. https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models.

23. Sadalage, Pramod. NoSQL Databases: An Overview. ThoughtWorks. [Online] 2014. http://www.thoughtworks.com/insights/blog/nosql-databases-overview.

24. Google, inc. Bigtable:A Distributed Storage System for Structured Data. [Online] 2006. http://static.googleusercontent.com/media/research.google.com/cs//archive/bigtable-osdi06.pdf.

25. Brewer, Eric. CAP Twelve Years Later: How the "Rules" Have Changed . [Online] 2012. http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed.

26. Seth Gilbert, Nancy Lynch. Perspectives on the CAP Theorem. [Online]

27. Microsoft. What is a Transaction? Microsoft Developer Network. [Online] https://msdn.microsoft.com/en-us/library/aa366402%28VS.85%29.aspx.

28. Oracle. Database Concepts - Transaction Management. Oracle Help Center. [Online] http://docs.oracle.com/cd/B19306_01/server.102/b14220/transact.htm.

29. Biographical Memoirs - James N. Gray. National Academy of Sciences (NAS). [Online] http://www.nasonline.org/publications/biographical-memoirs/memoir-pdfs/gray-james.pdf.

30. Principles of transaction-oriented database recovery. A. Reuter, T. Haerder. místo neznámé : ACM Computing Surveys, 1983.

31. Understanding ACID Properties. Beginning C# 2008 Databases: From Novice to Professional. 2008.

32. Microsoft. Microsoft Development Network. ACID properties. [Online] https://msdn.microsoft.com/en-us/library/aa480356.aspx.

33. Pritchett, Dan. Base: An Acid Alternative. [Online] 2008. http://queue.acm.org/detail.cfm?id=1394128.

34. Matematická biologie: E-learningová učebnice. Analýza dat hmotnostní spektrometrie. [Online] http://portal.matematickabiologie.cz/index.php?pg=analyza-genomickych-a-proteomickych-dat--analyza-genomickych-a-proteomickych-dat--analyza-dat-hmotnostni-spektrometrie.

35. gmbh, Solid IT. DB-Engines. [Online] http://db-engines.com/en/.

36. Document Databases : A look at them. CodeProject. [Online] 2012. http://www.codeproject.com/Articles/388982/Document-Databases-A-look-at-them.

37. RavenDB. [Online] http://ravendb.net/.

38. RavenDB. RavenDB Documentation. [Online] http://ravendb.net/docs/.

39. Eini, Oren. Ayende Rahien - Blog. API Design: Sharding Status for failure scenarios. [Online] http://ayende.com/blog/search?q=Sharding+Status+for+failure+scenarios.

40. The NIST Mass Spectrometry Data Center. The National Institute of Standards and Technology. [Online] http://chemdata.nist.gov.

41. AMDIS. Automated Mass Spectral Deconvolution and Identification System. [Online] http://www.amdis.net.

42. Apache Lucene. [Online] https://lucene.apache.org/.

43. Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters . Google Research Publications. [Online] 2004. http://static.googleusercontent.com/media/research.google.com/cs//archive/mapreduce-osdi04.pdf.

44. A full list of all the new/popular databases and their uses? Stack Overflow. [Online] http://stackoverflow.com/questions/1270321/a-full-list-of-all-the-new-popular-databases-and-their-uses.

12. Přílohy

1. Hodnoty naměřené při testování. (pouze v elektronické podobě)

2. Zdrojový kód aplikací dostupný na *http://ddbs.codeplex.com*.