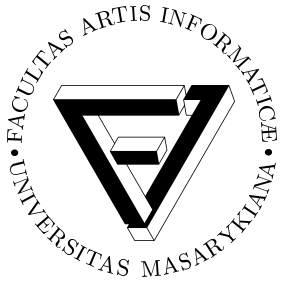
Masarykova univerzita
Fakulta informatiky



NoSQL úložiště pro data v podobě časových řad

Diplomová práce

Michal Dúbravčík

Brno, 2013

Prehlásenie

Prehlasujem, že táto diplomová práce je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Vedúci práce: doc. RNDr. Tomáš Pitner, Ph.D.

Poďakovanie

Rád by som poďakoval vedúcemu práce doc. RNDr. Tomášovi Pitnerovi, Ph.D. a hlavne konzultantovi Mgr. Danielovi Tovarňákovi za odborné vedenie. Taktiež sa chcem poďakovať kolegom zo spoločnosti Mycroft Mind za rady pri vývoji praktickej časti. V neposlednej rade patrí vďaka mojej rodine a priateľke.

Zhrnutie

Táto práca sa zaoberá analýzou možností NoSQL databáz s využitím ako perzistentné úložisko dát v podobe časových rad. Popisuje existujúce nástroje fungujúce nad týmito databázami. Súčasťou je popis a výkonnostné porovnanie implementovaného rozhrania nad MongoDB, ktoré je základným nástrojom pre monitorovacie účely.

Kľúčové slová

databázy, NoSQL, časové rady, agregácie, Map Reduce, MongoDB

Obsah

Kapitola 1

Úvod

Neustály vývoj informačných technológií vytvára platformu pre nárast objemu dát a rýchlosti, s akou pribúdajú. Tento fakt vynucuje vytváranie nových metód spracovávania a uchovávania dát.

Ak sa vydáme po stopách vzniku NoSQL hnutia, narazíme na dva hlavné zdroje. Spoločnosti Google a Amazon. Ako jejich online služby rýchlo rástli, potrebovali nové spôsoby ako ukladať veľké objemy dát naprieč tisíckami serverov. Výsledkom ich práce boli platformy BigTable v Google a Dynamo v Amazon. Na základe článkov, ktoré zverejnili, začali vznikať nové databázy, ktoré sa nimi inšpirovali [1]. Výsledkom bol nástup NoSQL databáz, ktoré obrátili pohľad na prácu a ukladanie dát. Vzniklo množstvo špecializovaných databáz, ktoré spája pôvodný zámer, a to bežať na tisíckach serverov.

Vďaka týmto veľkým spoločnostiam, ktoré sú naďalej nútené vyvíjať stále lepšie systémy, aby uspokojili potreby svojich zákazníkov a užívateľov, vieme dnes spracovávať analýzy dát v takmer reálnom čase, či dokonca v reálnom, ktoré pred pár rokmi vyžadovali niekoľko minút alebo hodín spracovania. Nie je problém, aby nami sledovaný systém, či zariadenie dokázalo produkovať veľké množstvo informácií, ktoré by sme chceli uchovať a analyzovať, a s využitím týchto databáz už nie je ani problém s ich uložením a rýchlym čítaním.

Je bežné , že chceme disponovať tými najlepšími dostupnými informáciami, a preto sa snažíme z každého zdroja vydolovať všetky možné dáta, aké poskytujú a čo nám kapacity výpočetných technológií dovoľujú. NoSQL databázy tak vytvárajú stabilný základ pre uspokojovanie týchto potrieb.

Táto práca si dáva za cieľ zanalyzovať druhy NoSQL databáz a preskúmať možnosti týchto databáz a existujúcich nástrojov nad nimi, v prípade, že potrebujeme sledovať, analyzovať a ukladať dáta, ktoré majú monitorovací charakter, či už ako pravidelné merania, alebo ako záznamy udalostí. Druhým cieľom je vytvorenie a popis vlastného nástroja nad NoSQL databázou, ktorý bude základom pre monitorovacie účely. Kapitola 2 uvádza do ďalšieho kontextu použitia NoSQL databáz.

---

1. FINLEY, K. *NoSQL: The Love Child of Google, Amazon*

Kapitola 3 bližšie vysvetľuje, aké databázy myslíme pod pojmom NoSQL a popisuje ich jednotlivé druhy. Popis existujúcich systémov určených na spracovanie časových rad, ktoré vznikli na základe využitia NoSQL, je obsahom kapitoly 4. Ďaľej práca pokračuje so zameraním na vyvinuté vlastné dotazovacie rozhranie. V kapitole 5 uvádzam, pre aké prípady bolo rozhranie vyvinuté a v kapitole 6 nad akou databázou je postavené. Kapitola 7 popisujem podobný nástroj, z ktorého rozhranie čiastočne vychádzalo a bol obohatený o novú funkcionalitu. Kapitola 8 opisuje samotné dotazovacie rozhranie a vysvetľuje jeho použitie. Na záver v kapitole 9 porovnávam výkonnosť tohto rozhrania voči implementácii za použitia relačnej databázy. Poslednou kapitolou je záver, zhodnotenie práce a možné pokračovanie v budúcnosti.

Kapitola 2

Big Data

Relačné databázové systémy (RDBMS) začali vznikať na základe relačného modelu vynájdeného v 70. rokoch E. F. Coddom v IBM. Vznikli kvôli potrebe centrálneho úložiska dát. No v tom období takéto úložisko predstavovalo jedno zariadenie, s kapacitou maximálne v desiatkách MB a pripájali sa naň len jednotky klientov. Za nasledujúcich 40 rokov sa ich vlastnosti zlepšovali a prispôsobovali sa aktuálnym potrebám. Stále aj v súčastnosti sú RDBMS číslom jedna pre ukladanie dát pre väčšinu potrieb. No existujú prípady, kedy RDBMS nedokážu zvládať dátový nápor ani pri využití toho najvýkonnejšieho hardvéru. Štatistiky spoločnosti Gupta uvádzajú: "Na YouTube je nahraných 72 hodín videa za minútu, prevedené na objem, je to 1 TB každé 4 minúty. 500 TB nových dát pribudne denne do databázy Facebooku. Urýchľovač častíc v CERN vygeneruje 1 PB dát za sekundu (avšak nie všetky sú uložené aj do databázy)."[1] Vtedy je potrebné poohliadnúť sa po alternatívnych spôsoboch ukladania dát. Jedným z nich sú NoSQL databázy.

No nemusíme byť veľkými spoločnostiami ako Google, Amazon, Twitter či Facebook, aby sa ukázalo, že relačné databázy nie sú vždy vhodné pre všetky typy dát. Moderné je hovoriť v tejto súvislosti o Big Data (veľké dáta). Spoločnosť Gartner charakterizuje veľké dáta pomocou 3V - High Volume, High Velocity, High Variety (veľký objem, vysoká rýchlosť, veľká rozmanitosť). Veľké dáta teda nie sú len o množstve, ale aj o veľkej rýchlosti zapisovani a čítania, a o rozmanitosti dátových štruktúr, čo vyžaduje použitie nových foriem spracovávania dát.

Oblasti ako senzorové siete, sociálne siete, indexové vyhľadávanie na Internete, vedecké a výskumné práce, Internet vecí, dokumentové multimediálne archívy, RFID, elektronické obchodovanie, predpovede v rôznych oblastiach, dohľad nad systémami a mnoho ďalších vyžadujú nové, rýchlejšie a účinejšie metódy spracovania.

---

1. HIGGINBOTHAM, S. *As data gets bigger, what comes after a yottabyte?*

Kapitola 3

NoSQL

Na začiatok je vhodné definovať, aké databázy máme na mysli pod termínom NoSQL. Viacero zdrojov uvádza, že tento pojem vychadza zo slov "no SQL"(ne-SQL databázy) alebo "not only SQL"(nielen SQL). Oba však mierne zavádzajú, pretože existujú databázy, ktoré spĺňajú taký predpoklad, a napriek tomu ich neradíme medzi NoSQL. Hlavnou ideou pri vzniku takýchto databáz bolo podporiť riešenie problémov moderných webových aplikácií, ktoré prerastajú schopnosti tradičných databázových systémov. Základnými charakteristikami, ktoré sú spoločné pre väčšinu NoSQL databáz sú:

- nepoužívajú relačný model (navyše ani SQL)
- sú open source
- sú navrhnuté na prácu v klastri
- nepoužívajú žiadne schéma (záznamy nie sú obmedzované presnou štruktúrou)
- sú horizontálne škálovateľné[1].

Ďalej mnoho z nich ponúka jednoduchú replikáciu (duplikovanie serverov), jednoduché programovacie rozhranie (API) a schopnosť spracovávať veľké objemy dát. NoSQL databázy začali vznikať na začiatku roka 2009 a v súčasnosti ich môžeme napočítať okolo 150[2]. Všetky majú za sebou zatiaľ len krátky vývoj, ktorý neustále odhaľuje nedokonalosti či pridáva novú funkcionalitu.

Triedu NoSQL databáz môžme ďalej rozčleniť podľa dátového modelu. Definovali sme, že je nerelačný. Ten môžeme ďalej rozdeliť na model kľúč-hodnota (key-value), dokumentový model, model rodiny stĺpcov (column-family), grafový a ďalšie.

---

1. FOWLER, M. *NosqlDefinition*
2. *NoSQL databases.* http://nosql-database.org/

3.1 Model kľúč-hodnota

Na tento model sa dá pozerať ako na tabuľku v RDBMS s dvoma stĺpcami: kľúč a hodnota. Hodnota v tomto úložisku môže byť ľubovoľný blok dát bez potreby analyzovania obsahu na strane databázy. To má na starosť až aplikačná vrstva. Môže to byť jednoduchá hodnota, text, JSON, XML či iné štruktúry. Databázy s týmto modelom predstavujú z pohľadu prístupu najjednoduchšiu triedu NoSQL. Klient môže hodnotu pre daný kľúč vkladať, vyberať alebo zmazať.

Všetky databázy kľúč-hodnota však môžu k dátam pristupovať len prostredníctvom kľúča. Ak potrebujeme dotazovať podľa nejakého atribútu uloženého v poli hodnota, musíme zvoliť databázy s iným modelom. Výnimkou sú databázy, ktoré podporujú sekundárne indexy (napr. Riak). Ich vytvorením môžme potom dotazovať uložené dáta okrem kľúča aj podľa indexovaného atribútu.

Výhodou takýchto databáz je dobrá škálovatelnosť a vysoký výkon vďaka jednoduchému modelu. Najznámejšími databázami v tejto triede sú Riak, Redis, Memcached, Berkeley DB, Amazon DynamoDB Project Voldemort, Aerospike.

Databázy kľúč-hodnota sú vhodné pre prípady ukladania informácií o reláciach (ang. session - v zmysle spojenia klient - server), kde jednoduchou operáciou put ukladáme zoskupené dáta v jednom zázname pod kľúčom (zvyčajne identifikátor užívateľa a jeho relácie) a neskôr k ním rýchlo pristupujeme pomocou operácie get. Databáza Memcached sa špecializuje presne na takéto použitie a je používaná mnohými webovými aplikáciami ako cache[3].

3.2 Model rodiny stĺpcov

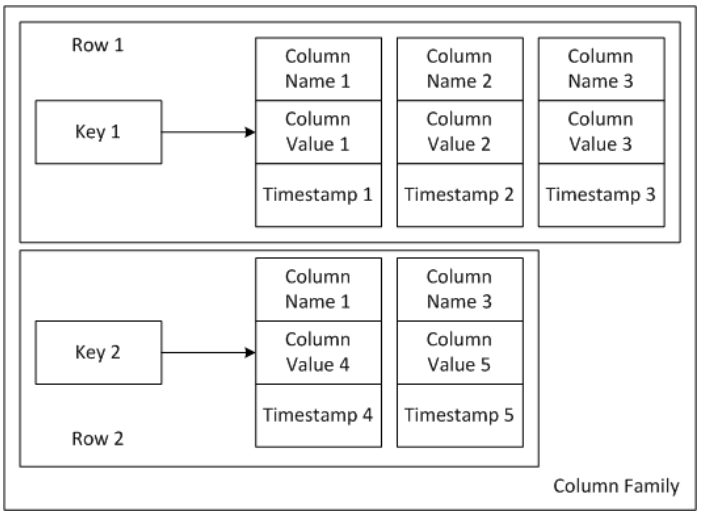
V databázach s týmto modelom sú dáta uložené ako viacrozmerné mapy. Záznam obsahuje kľúč namapovaný na hodnoty. Hodnoty sú zoskupené do rôznych takzvaných rodín stĺpcov (column-family) a každá rodina stĺpcov obsahuje mapu dát.

V mnohých zdrojoch sa názov týchto databáz zamieňa so stĺpcovo orientovanými [4]. Tento názov je ale nesprávny. Stĺpcovo orientované databázy sú relačné, ktoré ale narozdiel od väčšinou používaných riadkovo orientovaných RDBMS (Microsoft SQL Server, MySQL, Oracle...) ukladajú dáta po stĺpcoch. Príkladom sú systémy Sybase IQ, Teradata, MonetDB, Vertica. Nemožno ich teda zaraďovať do triedy NoSQL.

---

3. SADALAGE, P. J., FOWLER, M. *NoSQL distilled* str. 81 - 87
4. ABADI, D. Distinguishing Two Major Types of Column-Stores

Typický predstaviteľ databáz s modelom rodiny stĺpcov, Cassandra, ukladá dáta ako riadky (row), ktoré majú viacero stĺpcov (column), združené s kľúčom riadka (key). Stĺpce, ku ktorým často pristupujeme spoločne, zoskupujeme do jednej rodiny stĺpcov. Riadok má voľnú schému, takže môže obsahovať rôzne stĺpce. Každá bunka (obsah stĺpca pre daný kľúč riadku) okrem samotnej hodnoty obsahuje aj časovú známku (timestamp), ktorá slúži na odhaľovanie konfliktov pri zápise, expirovanie atď. Celú štruktúru zachytáva (obr. 3.1), platný je aj pre iné databázy tohoto modelu. Cassandra dovoľuje navyše od štandardnej rodiny stĺpcov, ukladať super-stĺpce (super-column). Takýmto stĺpcom je mapa ďalších dvojíc kľúč-hodnota. Cassandra tak vytvára v rámci záznamu viacdimenzionálnu mapu.



Obr. 3.1: Dátový model v databáze Cassandra Zdroj: http://10kloc.wordpress.com/2012/12/25/cassandra-chapter-three-data-model/

Okrem databázy Cassandra sú v tejto triede známe ďalšie ako HBase a Hypertable. Väčšina databáz tohto modelu sú klony databázy BigTable od spoločnosti Google[5]. Tieto databázy poskytujú lineárnu škálovatelnosť. Dokazuje to aj benchmark s Cassandra databázou, kde zvýšenie počtu uzlov v klastri z 50 na 300, zvýšilo priepustnosť zápisu na šesťnásobok [6].

5. SADALAGE, P. J., FOWLER, M. *NoSQL distilled* str. 99
6. COCKCROFT, A., SHEAHAN, D. Benchmarking Cassandra Scalability

3.3 Dokumentový model

Základným prvkom databáz s dokumentovým modelom sú dokumenty. Dokumentom môžu byť štruktúry ako XML, JSON, BSON a ďalšie. Tento model je rozšírením modelu kľúč-hodnota s tým, že ako hodnota je uložený dokument. Pre uloženie dokumentu je teda potrebné dodržať formát dokumentu, ktorý databáza podporuje. K záznamom môžme pristupovať okrem kľúča aj cez obsah dokumentov a upravovať ich na strane servera. Takéto databázy sa vyznačujú bohatším dotazovacím jazykom ako databázy typu kľúč-hodnota. Ukladané dokumenty môžu mať navzájom rôznu schému a nie je nutné ju vopred definovať.

Dokumenty predstavujú jednoduchý a intuitívny spôsob reprezentácie zložitejších dátových štruktúr a väčšinou sú dobre mapovateľné z objektovo orientovaných jazykov. Populárnymi dokumentovými databázami sú MongoDB, CouchDB, OrientDB (grafovo-dokumentová databáza), Terrastore a RavenDB. Viac o MongoDB v kapitole 6.

3.4 Grafový model

Grafový model sa značne odlišuje od ostatných typov modelov. Kým záznamy v databázach s predchádzajúcimi modelmi sa snažili byť bez vzťahov k ďalším záznamom, tak v týchto dátabázach záznamy môžu mať veľké množstvo vzťahov. Tento model reprezentuje dátové štruktúry ako grafy. Vo väčšine prípadov ide o orientovaný, ohodnotený graf. Záznamy sú nazývané uzlami. Vzťahy medzi nimi sú hranami, ktoré sú smerovo orientované a doplnené o atribúty vzťahu.

Výhodou takých databáz sú možnosti riešenia klasických grafových problémov. Databáza Neo4j má implementované grafové algoritmy na hľadanie najkratšej vzdialenosti medzi uzlami, nájdenie všetkých ciest, či Dijkstrov algoritmus. Jasným využitím sú sociálne siete, ktoré vyžadujú evidovanie veľkého počtu a typu vzťahov.

Existuje mnoho grafových databáz, medzi tie najznámejšie patria Neo4j, Infinite Graph, OrientDB alebo FlockDB.

3.5 Časové rady

Časová rada je podľa definície postupnosť pozorovaní (dát), ktoré sú jednoznačne usporiadané z hľadiska času v smere od minulosti do prítomnosti.

Seger (1995) uvádza, že časové rady môžme rozlišovať podľa ich charakteru na:

- časová rada intervalových ukazateľov (napr. dátové prenosy v sieti za mesiac)
- časová rada okamžikových ukazateľov (napr. vyťaženie procesora)
- časová rada odvodených charakteristík (odvodená z predchádzajúcich rad, často súčtové alebo pomerové rady, napr. kumulatívny súčet prenesených dát od určitého času v minulosti)

Zatriedenie rady do správnej kategórie má vplyv na to ake analýzy, výpočty a samotné spôsoby výpočtu na nich môžme vykonať. Nemalo by zmysel počítať sumu napätia v sieti alebo priemer kumulovanej spotreby. Ďôležité je taktiež zvážiť odlišnosti časových intervalov a potreby vážených výpočtov podľa veľkosti intervalu a pod.

Určitú výhodu prináša používanie odvodenej rady namiesto intervalovej. Predstavme si, že zaznamenávame spotrebu elektriny za hodinu. Ak by sme odosielali do systému na spracovanie hodinovú spotrebu, vystavujeme sa riziku, že ak sa tento údaj stratí, nie je možné ho odvodiť. Vhodnejšie je preto vytvoriť čítač a každú hodinu odosielať jeho hodnotu. Z rozdielu susedných meraní sa spoľahlivo dopočíta absolútna spotreba a v prípade, že by sa nejaké meranie stratilo, je ho možné priemerne dopočítať. Takúto funkcionalitu natívne poskytuje OpenTSDB (viac v kapitole 4.1).

Elementárnu časť časovej rady – záznam merania/pozorovania – ďalej budeme označovať ako prvok časovej rady. Takýmto prvkom býva kvantitatívna hodnota veličiny. Niektoré ďalej uvádzané systémy a databázy síce podporujú spracovanie aj kvalitatívnych veličín, no ak je to možné, efektívnejšie sa pracuje s transformovanými radami na numerické vyjadrenie, ktoré podporujú nutne všetky systémy.

Pre ďalšiu prácu je potrebné definovať, čo je to udalosť. Etzion (2011) hovorí: Ùdalosť je jav v rámci systému alebo oblasti. Je to niečo, čo sa stalo alebo je to považované, akoby sa malo stať v danej oblasti. Slovo udalosť je tiež používané vo význame programovej enitity, ktorá reprezentuje výskyt udalosti vo výpočetnom systéme.Ùdalosťou môže byť napríklad v počítačovej oblasti: prekročenie určitej hranice zaplnenia disku, neautorizovaný prístup do systému, zlyhanie zariadenia, v oblasti monitorovania domov: vznik požiaru, spustenie alarmu, nízka teplota, v oblasti obchodu: príjazd dodávky, platba zákazníka atď. Udalosti majú však komplexnejší charakter ako merania. Merania väčšinou zaznamenávame pomocou jednoduchých dátových typov (číselných či znakových), kdežto udalosti sú objekty. Platba zákazníka môže byť jednoduchým "meraním"hodnoty platby, ale aj udalosťou – objektom obsahujúcim identifikáciu platiteľa, nakúpené položky, metódu platby a mnoho ďalšieho.

Merania časovej rady mávajú väčšinou pravidelný charakter. Každú minútu, hodinu či inú časovú jednotku zaznamenávame tú istú veličinu. Naproti tomu udalosti majú náhodnostný charakter. Nevieme povedať kedy, ani ako často nastávajú. Udalosti vznikajú na základe situácii, ktoré sú neplánované, kdežto merania vznikajú úmyselne. Napriek tomu úložiska časových rad tento aspekt neberú do úvahy. Ukladať do nich môžeme pravidelné časové rady, ale aj úplne náhodné.

Systémy pre ukladanie časových rad sú primárne zamerané na klasické časové rady, ktoré obsahujú prvky jednoduchých dátových typov. Nepodporujú teda ukladanie žiadnych zložitejších štruktúr. Nasledujúca kapitola predstavuje tie najznámejšie.

Kapitola 4

Systémy na spracovanie časových radov

4.1 OpenTSDB

OpenTSDB (Open Time Series DataBase)[1] je open-source databáza postavená na HBase. Implementuje techniky, ktoré optimalizujú systém ukladania dát vhodný pre časové rady. Vytvorená bola pre potrebu ukladania, indexovania a pracovania s metrikami zozberanými z počítačových systémov vo veľkom rozsahu. Pomocou nej je taktiež možné časové rady vykresľovať do grafov. Na to je potrebné doinštalovať program gnuplot.

Ponúka RESTful HTTP API pre všetky operácie nad databázou. Dáta môžu byť vkladané vo formáte JSON, XML a iných. Časové rady sú označované pomocou názvov, značiek a metadát. Čítanie rad prebieha pomocou dotazov, kde môžeme filtrovať prvky pomocou značiek. Ak definujeme zoskupovací interval, databáza podporuje výpočet základných štatistík: suma, priemer, maximum a minimum.

V prípade, že sme uložili do OpenTSDB časovú radu, ktorej prvky majú rastúci charakter (napr. čítač), užitočnou operáciou je diferencia, ktorá vypočíta rozdiel navzájom susedných prvkov. Medzi viacerými časovými radami môžme previesť operáciu sčítania, priemeru, maxima alebo minima. V takom prípade môže nastať situácia, že prvky nemajú vždy zhodné časové značky. Vtedy je prevedená lineárna interpolácia okolných hodnôt.

Jednotlivé prvky časovej rady môžu obsahovať anotáciu. Tie predstavujú elementárny spôsob zaznamenávania udalostí. Systém ďalej nepodporuje žiadny nástroj na spracovávanie takýchto “udalostí”. Cieľom anotácii je skôr pridávať dodatkové informácie k radám. Vhodnejšie je ich spracovávanie pomocou externých nástrojov. Anotácie sú interne využívané v systéme ako popisky bodov vo vykreslených grafoch.

Optimalizácia schémy dát pre časové rady spočíva v zoskupovaní prvkov po hodinách do jedného riadku. Kľúčom riadku je dátum bez ödstrihnutých"minút. Prvok je v tomto riadku uložený pod stĺpcom, ktorý má názov vytvorený z od-

1. http://opentsdb.net/

strihnutej časti dátumu. Akonáhle skončí ukladanie prvkov v danej hodine (kedy už nie je predpoklad, že by nové prvky pribúdali) na pozadí sa spustí proces, ktorý zoskupí všetky stĺpce na riadku do jednej bunky. To má za následok rapídne zvýšenie rýchlosti. Dáta nie sú počas jednej hodiny ukladané do jednej bunky, z toho dôvodu, že by to vyžadovalo čítanie bunky a jej prepísanie, čo by spôsobilo nízky výkon.

OpenTSDB hovorí na svojich stránkach, že dokáže ukladať 2 tis. prvkov časovej rady za sekundu aj pri použití jedného uzlu tvoreného spotrebným PC. Podobné výsledky dosahuje aj ďalší benchmark s uložením 10 mil. prvkov a dosiahnutím viac ako 8 tis. zápisov/sek. na jednom štvorjadrovom uzle.

4.2 Chuckwa

Chukwa je open-source systém určený na monitorovanie veľkých distribuovaných systémov. Postavený je na frameworku Pig, Hadoop a databáze HBase. Zber dát prebieha pomocou agentov, ktorý sú súčasťou monitorovaných zariadení, ktoré preposielajú dáta na kolektor, ktorý ich ukladá do dočasného úložiska v HDFS a pomocou Hadoop Map Reduce procesov sú spracované a uložené natrvalo. Tieto dáta sú potom pomocou analytického webového rozhrania v štýle webových portálov vizualizované v grafoch.

4.3 TempoDB

TempoDB je cloudová databáza (databáza ako služba) určená pre časové rady. Použité dátové úložisko ani kód nie je zverejnený. K databáze sa pristupuje cez REST API. Webové rozhranie poskytuje základné analytické funkcie a vykreslovanie grafov. Pri komunikácii sa používa JSON formát.

Prvok časovej rady je dvojica časová značka a hodnota, ktorou môže byť integer alebo float (viď obr. 4.1).

```
[{"t": 2014-01-03T14:23:33.000+0600, "v": 52.113},
{"t": 2014-01-03T14:23:34.000+0600, "v": 51.762}]
```

Obr. 4.1: Prvok časovej rady v TempoDB

Pomocou API môžme uložiť jeden alebo dávku prvkov časovej rady spolu s listom značiek. Čítať môžme viacero časových radov naraz a výstupom je JSON objekt,

ktorý obsahuje metadata časovej rady, sumárne štatististiky a pole dátových bodov v rozmedzí dvoch dátumov. Voliteľné je definovanie veľkosti intervalu (od 1 minúty vyššie), na aký je "rozsekaný"celý časový rozsah a na dátových bodoch v každom intervale je vypočítaná štatistika. Môže ňou byť priemer, suma, minimum, maximum, štandardná odchýľka, suma štvorcov, počet, rozsah (maximum-minimum) a percentil. Ak už nie sú dáta potrebné, môžu byť z časovej rady odstránené podľa časové rozsahu alebo kľúča časovej rady.

4.4 Ďalšie systémy na spracovanie časových radov

Ďalej spomínané systémy nie sú len databázy, ale platformy poskytované ako služby (PaaS). Uvádzam ich ako alternatívy NoSQL databáz, pretože všetky tieto systémy v sebe obsahujú špecializované databázy prispôsobené na ukladanie časových rad. Značne odlišnou službou je Squwk [http://squwk.com/], ktorý poskytuje SEP (jednoduché spracovanie udalostí) a akceptuje dáta všetkých typov od číselných až po dokumenty. Xively [https://xively.com] je platforma smerovaná na vývojárov produktov v súvislosti s “Internetom vecí”, rovnako ako ThingSpeak [https://www.thingspeak.com/] (podpora analytických funkcií a vizualizácie s bohatou dátovou štruktúrou) a Nimbits [http://www.nimbits.com/] (úložisko časových radov s podporou trigrov a upozornení). Posledným je systém Cube, ktorému je vyhradená kapitola 7.

Kapitola 5

Dotazovacie rozhranie

Vyvinuté rozhranie má za cieľ poskytnúť nástroj primárne pre oblasť Inteligentného merania a Zaznamenávania logov a monitorovania výpočetných systémov. Ako úložisko bola zvolená dokumentová trieda NoSQL, kvôli možnosti vkladať rozmanité dátové štruktúry a následne s nimi aj pracovať na strane databázy. Takúto funkcionalitu ostatné triedy NoSQL neposkytujú. Z tejto triedy bolo vybrané MongoDB, ktoré je súčasne najpoužívanejšia databáza zo všetkých NoSQL. MongoDB obsahuje aj agregačné nástroje, ktoré sú dobre použiteľné pre štatistické výpočty nad numerickými atribútmi záznamov. Viac o vlastnostiach a výhodnosti použitia MongoDB pojednáva kapitola 6.

5.1 Prípady použitia

5.1.1 Inteligentné meranie

"Inteligentný merací systém" znamená elektronický systém, ktorý je schopný merať spotrebu energie a pridávať k tomu viac informácií než bežný merač a ktorý je schopný vysielať a prijímať dáta s využitím niektorej formy elektronickej komunikácie. Inteligentné meranie predstavuje platformu pre "Inteligentnú sieť", ktorá vytvára zdokonalenú energetickú sieť, ku ktorej bola pridaná obojsmerná digitálna komunikácia medzi dodávateľom a spotrebiteľom, inteligentné meranie a monitoring a riadiace systémy[1].

V prípade ČR je plánované nasadenie 3,5 mil. meracích zariadení, ktoré by každých 15 minút odosielali údaje. V závislosti od množstva zaznamenávaných ukazateľov má jedno meranie 3-9 kB. To predstavuje v ideálnom prípade 4 000 operácii za sekundu, za deň celkovo 336 mil. meraní a pri 9kB meraniach 2,8 TB dát/deň. Na takéto objemy a rýchlosť nie sú RDBMS pripravené, je preto nutné využiť lepšie škálovatelné databázy a nasadenie databázového klastra.

1. 2012/148/EÚ: Odporúčanie Komisie z 9. marca 2012 o prípravách na zavádzanie inteligentných meracích systémov

V najjednoduchšom prevedení by prvky časovej rady obsahovali plochý dokument s jednou nameranou hodnotou. Keďže ale predmetom merania elektriny nie je len spotreba, ale aj mnoho ďalších metrík, v širšej variante by mohlo jedno meranie pozostávať z viacerých hodnôt v stále plochom dokumente. V najširšej variante by záznam obsahoval aj profil meraných hodnôt počas 15 minút. V takom prípade by záznam plne využíval bohatosť tohoto dokumentového formátu.

Hlavnou výhodou nasadenia NoSQL je v tomto prípade spracovanie veľkých dátovych objemov.

5.1.2 Zaznamenávanie logov a monitorovanie výpočetných systémov

Spravovanie rozsiahlých výpočetných systémov a dosahovanie neustálej dostupnosti vyžaduje sledovanie infraštruktúry a okamžité zásahy v prípade kritických udalostí. Veľké množstvo metrík zaznamenaných od hardvéru až po užívateľskú vrstvu pri sekundovom rozlíšení vygeneruje obrovské množstvo dát, ktoré musia byť spracované a uložené. Okrem pravidelného merania stavu zariadení môžme zaznamenávať udalosti generované týmto systémom (tzv. logy).

Relačné databázy vyžadujú ukladanie dát s rovnakou štruktúrou. Logy sú ale často variabilné a nepredikovateľné štruktúry a s vývojom systému pribúda ich bohatosť. NoSQL databázy s voľnou schémou sú preto vhodnejšie pre ukladanie logov. Logy naformátované v JSON môžu byť priamo uložené bez potreby zložitého mapovania do dokumentových databáz s plným využitím všetkých vlastností tohoto formátu.

Výhodou nasadenia NoSQL je v tomto prípade okrem spracovania veľkých dátovych objemov aj schopnosť ukladať rozmanité typy logov.

Kapitola 6

MongoDB

MongoDB[1] je open-source dokumentovo orientovaná databáza vyvinutá spoločnosťou 10gen, jedna z najpoužívanejších NoSQL databáz. Navrhnutá bola pre dátové a škálovacie potreby súčasných internetových aplikácií. Podporuje indexy (jednoduché aj zložené), má relatívne pestré možnosti dotazovania, jednoduché škálovanie prostredníctvom replikácie a automatického shardovania v LAN aj WAN, a dokáže ukladať veľké binárne súbory prostredníctvom GridFS.

Ako bolo spomenuté, MongoDB používa dokumentový model. Dokumentom je množina dvojíc (kľúč, hodnota), ktorý je ukladaný do kolekcie v databáze. Terminologicky je kolekcia obdoba tabuľky v RDBMS a dokument obdoba riadku tabuľky. Takéto dokumenty majú v MongoDB formát JSON[2]. Príkladom dokumentu je štruktúrovaný log (obr. 6.1).

Dáta su ukladané a taktiež prenášané v sieti v BSON[3] formáte, ktorý je vytváraný binárnym kódovaním serializácie JSON dokumentov. BSON rovnako ako JSON podporuje vkladanie subdokumentov (dokument v dokumente), polia dokumentov a polia jednoduchých typov. BSON navyše obsahuje rozšírenia, ktoré nie sú v špecifikácií JSON (napr. dátový typ Date, Binary Data alebo JavaScript Code).

Výhodou takéhoto formátu a teda aj celkovo DB je voľné schéma dokumentov. Narozdiel od SQL úložísk, v ktorých je vopred potrebné definovať štruktúru tabuliek, do MongoDB môžme ukladať dokumenty bez toho, že by sme vopred definovali, akú štruktúru má dokument a navyše, každý dokument môže mať aj rôznu štruktúru. Takýto prístup je veľmi výhodný pre ukladanie logov a podobných objektov, ktoré majú variabilný obsah. Druhotnou výhodou je vhodnosť pre agilný vývoj, kedy sa očakávajú zmeny štruktúry dát. Dokumenty s novou štruktúrou môžu byť ukladané bez toho, aby sme vykonali nejaké zmeny v DB.

---

1. odvodené od "humongous"v preklade extrémne veľký
2. JSON (JavaScript Object Notation), http://www.json.org/
3. Binary JSON, http://bsonspec.org/

```
{
      _id : ObjectId("50a8240b927d5d8b5891743c"),
         occurrenceTime : ISODate("2012-01-01T01:00:00+01:00"),
         host : "192.168.1.100",
         type : "GET",
         application : "Apache Server",
         process : "httpd",
         processId : 4219,
         responseTime : 31,
         log : {
                  file : "http://httpd.apache.org/",
                  resource : "/apache_pb.gif",
                  protocol : "HTTP/1.0"
                  responseCode : 200
               }
}
```

Obr. 6.1: Log uložený v MongoDB

Denormalizované schéma, teda použitie vnorených dokumentov (subdokumentov) a polí znižuje potrebu spájania, ako ho poznáme z RDBMS, čo je kľúčová vlastnosť pre vysoký výkon a rýchlosť[4].

MongoDB využíva všetkú voľnú pamäť RAM, ktorú zariadenie poskytuje, ako cache pre dáta, aby nemuselo neustále pristupovať na disk. Najvyšší výkon tak dosahuje nasadenie, kedy sa celá dátová množina, s ktorou pracujeme (najčastejšie používané kolekcie a indexy), vojde do RAM[5].

MongoDB je pripravené na zvyšujúce sa výkonostné požiadavky prostredníctvom horizontálneho škálovania. Na tieto účely slúži sharding, ktorý rozdeľuje kolekciu na menšie časti a distribuuje ju v rámci clustera. V prípade, že databáza nadobudne väčší objem než poskytuje existujúce úložisko, stačí pridať ďalšie zariadenie a sharding zabezpečí rovnomerné rozloženie medzi tieto zariadenia. Samotné rozdeľovanie medzi jednotlivé shards (útržky databázy) prebieha na základe sharding key ("deliaci kľúč"). Ten by mal byť volený s ohľadom na, čo najrovnomernejšie rozloženie dát medzi uzlami. V prípade logov (ale aj pri iných

4. MENEGAS, G. What is NoSQL, and why do you need it?
5. 10gen. MongoDB Manual, Fundamentals

použitiach) ním bude najčastejšie položka zdroj, na ktorom daný log vznikol, čo zabezpečí, že logy z jedného zdroja budú uložené na rovnakom shard. Takáto distribúcia nám uľahčí bezproblémové paralelné vykonávanie agregačných dotazov, ktoré budú často zoskupované podľa zdroja (v jazyku SQL : GROUP BY source).

Sharding by mal byť nasadzovaný až ako jedna z posledných možností. V prípade, že dátové potreby presahujú možnosti jedného stroja, alebo pracovná množina prevyšujú objem RAM, alebo systém generuje veľké požiadavky na zápis, ktorý nedokáže jedna inštancia MongoDB obslúžiť a ostatné metódy nedosiahli požadované výsledky. V iných prípadoch nasadenie sharding neprinesie väčšinou žiadne prínosy, naopak sa tým len zvýši komplexnosť systému[6].

Sharding vyžaduje v produkčnom nasadení, aby bol každý shard zložený z replica set. Replica set je cluster MongoDB inštancií, ktoré sú navzájom replikované. To znamená, že všetky inštancie obsahujú rovnaké dáta a tým zabezpečujú automatickú obnovu v prípade zlyhania. Väčšina replica set obsahuje 2 a viac inštancií, z ktorých jedna je primárna a ostatné sekundárne. Klient vždy zapisuje dáta na primárnu a sekundárne inštancie asynchrónne replikujú tieto dáta. V situácii, keď zlyhá primárny uzol, systém automaticky zabezpečí zvolenie nového primárneho uzlu, ktorým sa stane jeden zo sekundárnych.

Replikácia v MongoDB pridáva redundanciu, pomáha udržiavať vysokú dostupnosť, zjednodušuje zálohovanie a môže zvyšovať kapacity na čítanie z DB (prostredníctvom čítania zo sekundárnych uzlov).

Dôležité je upozorniť na to, že je potrebné sledovať zaplnenie kapacity disku a RAM. V prípade zaplnenia disku, MongoDB blokuje všetky operácie a nie je možné vyvolávať ani zmazanie. V prípade, že sa pracovná množina nevôjde do RAM môže nastať spomalenie celého systému až na úroveň rýchlosti disku. V prípade systému s vysokými nárokmi na výkonnosť, tak môže dôjsť k dramatickému poklesu rýchlosti až výpadkom, ako sa stalo v prípade populárnej geolokačnej socialnej siete Foursquare, ktorá používa MongoDB, kde došlo z tejto príčiny k 17 hod. výpadku[7].

6.1 Funkcie MongoDB využiteľné pre štatistické výpočty

Narozdiel od ostatných NoSQL databáz, MongoDB poskytuje širšie možnosti dotazovania a vykonávania výpočtov na strane databázy. Tie síce nedosahujú tak obrovské spektrum, ako ponukajú RDBMS, no v porovnaní s inými NoSQL,

6. 10gen. MongoDB Manual, Sharded Cluster Overview
7. HOFF, T. Troubles with Sharding

pokrývajú väčšinu potrieb, ktoré kladieme na DB a nie je teda nutné všetky výpočty suplovať aplikačnou vrstvou.

Väčšina NoSQL poskytuje len CRUD[8] operácie a zložitejšie dotazy, napr. agregačné, je potrebné doprogramovať. Nepripadá mi vhodné, ani efektívne neustále vynaliezať koleso, preto používanie MongoDB a jeho nástrojov považujem za dobrú voľbu.

Taktiež komunikácia medzi aplikáciou a databázou nás stojí nezanedbateľný čas, preto vo vykonávaní výpočtov nad dátami, čo najbližšie k jejich úložisku, vidím úsporu zdrojov. Tento aspekt má ešte väčší význam v prípade databázového clusteru, kde môže dochádzať k paralelným výpočtom priamo na uzloch.

MongoDB má implementované 2 nástroje pomocou, ktorých je možné agregovať. Prvým jednoduchším je Agregačný Framework s preddefinovanými funkciami, vhodný na jednoduché úlohy ako sčítavanie či priemerovanie a druhý Map Reduce, ktorý je vhodný na zložitejšie úlohy. Oba tieto nástroje sú schopné fungovať aj pri využití horizontálneho škálovania - shardingu.

6.2 Agregačný Framework

Hlavným využitím tohto nástroja sú pre nás štatistické funkcie: suma, počet, priemer, maximum a minimum. Navrhnutý je tak, aby bol výpočet agregácií jednoduchý a zároveň výkonný, bez potreby zložitého Map Reduce. Agregačný framework podporuje výpočet vo viacerých vláknach a je napísaný v jazyku C++, preto je ho možné natívne vykonávať na všetkých uzloch.

Agregačný framework funguje na princípe pipeline (množina zoradených operátorov, cez ktoré prechádzajú dokumenty). Pipeline vytvára prúd dokumentov, pričom na začiatku sú dokumenty čítané z kolekcie a predané na vrchol - prvý operátor pipeline a nasledné sú posúvané postupne k ďalším operátorom vo vopred definovanom poradí. Operátory v pipeline sa môžu opakovať a v prúde dokumentov môžu vznikať nové dokumenty alebo môžu niektoré zanikať filtrovaním.

Pipeline sa môže skladať z týchto operátorov:

- $group (zoskupovanie)
- $project (projekcia)
- $match (filtrovanie)

8. z angličtiny Create, Read, Update, Delete, teda operácie na vytváranie, čítanie, aktualizovanie a mazanie

- $limit (obmedzenie počtu)
- $skip (preskočenie počtu)
- $unwind (rozvoj pola na nové dokumenty)
- $sort (zoradenie)

Výpočet prostredníctvom Agregačného framweroku sa spúšta volaním funkcie aggregate() s parametrom pipeline - pole s usporiadanými operátormi (prvý prvok pola je prvý operátor v pipeline):

```
db.people.aggregate( [<pipeline>] )
```

Každý operátor musí obsahovať výraz v JSON formáte. Najdôležitejším operátorom pre agregovanie je $group.

Predpokladajme, že máme kolekciu log naplnenú udalosťami formátu ako v ukážke na obr. 6.1, potom pre výpočet počtu udalostí, priemernej a maximálnej doby odpovede zoskupením podľa pôvodu udalosti (host), môžme využiť Agregačný framework (obr. 6.2).

```
db.log.aggregate(
    [ $group : {
        _id : "$host",
        count : { $sum : 1 },
        avg : { $avg : "$responseTime" },
        max : { $max : "$responseTime"}
    }]
);
```

Obr. 6.2: Agregácia pomocou Agregačného frameworku

V tomto príklade pipeline obsahuje jeden operátor $group, ktorý zabezpečuje zoskupovanie podľa položky _id. Názvy položiek dokumentov sa uvádzajú so znakom dolára ($). Položka _id môže obsahovať jednoduchú hodnotu ako v príklade $host- alebo zloženú, napr. {host: "$host", type : "$type"}, ak by sme okrem zoskupovania podľa zdroja chceli navyše rozlišovať typ požiadavky.

Operátor $group musí obsahovať aspoň jeden agregačný výraz z:

- $max (maximum)

- $min (minimum)
- $avg (priemer)
- $sum (suma alebo počet)
- $first (prvá hodnota)
- $last (posledná hodnota)
- $addToSet (hodnotu vloži do pola bez duplicít)
- $push (hodnotu vloží do pola s duplicitami)

Ďalším dôležitým operátorom je $project, pomocou ktorého je možné určiť položky dokumentov, ktoré chceme ponechať a ktoré zahodiť (napr. {host : true, processId: false}), premenovať položky (napr. {source: "$host"}), presunúť položku zo subdokumentu (napr. {file:"$events.log.file"}) alebo prepočítať pomocou aritmetickej operácie (súčet, rozdiel, násobenie, delenie, zvyšok po delení), textovej operácie (zreťazenie, lexikografické porovnanie, extrakcia podreťazca, zmena na veľke alebo malé písmená), dátumovej operácie (extrakcia samostnanej časovej jednotky z dátumu od milisekundy až po rok) alebo podmienený výraz fungujúci ako ternárny operátor (?:).

Napríklad príkaz na obr. 6.3 pomocou operátora $year z časovej položky dokumentu môžme extrahovať rok a pomocou $dayOfYear deň z tohoto dátumu. Na číselnej položke dokumentu je vykonaná aritmetická operácia delenie. Ako je vidieť, pri všetkých operáciach sa používa prefixový tvar. V prípade, že má operácia viac operandov, uvádzajú sa v poli.

```
db.log.aggregate(
    [ $project : {
        year : {$year : "$occurenceTime"},
        day : {$dayOfYear : "$occurenceTime"},
        responseInSeconds :
                {$divide: ["$responseTime",1000]}
    ]
);
```

Obr. 6.3: Projekcia v Agregačnom frameworku

Operátor $match filtruje dokumenty, ktoré sú posúvané k ďalšiemu operátoru. Používať môže štandardné operácie používané aj v CRUD príkazoch: porovnávacie - eq (rovnosť), lt (menšie ako), le (menšie alebo rovné), gt (väčšie ako), ge (väčšie alebo rovné), ne (nerovnosť), in (rovný jednej hodnote z pola). pravdivostné - and (konjunkcia), or (disjunkcia), not (negácia)

Ďalšie operátory v pipeline môžu byť: $limit, ktorý obmedzuje počet dokumentov na začiatku prúdu, ktoré sa pošlú ďalej v pipeline, $skip spôsobí vynechanie daného počtu dokumentov zo začiatku prúdu, $sort zoraíi dokumenty vzostupne alebo zostupne $unwind vytvorí rozvoj subdokumentu alebo pola. Tým vznikne pre každý prvok subdokumentu celý nový samostatný dokument s tým, že namiesto subdokumentu je jeden jeho prvok (z 1 dokumentu, ktorý obsahuje subdokument s 10 položkami, sa vytvorí 10 dokumentov).

Nevýhodou Agregačného frameworku je momentálne to, že nepodporuje uloženie výstupu agregácie naspäť do kolekcie, teda je potrebné ho manuálne uložiť. Ďalším obmedzením, ktoré je možné pocítiť len pri rozsiahlých agregáciach, je limit 16 MB na celkový výstup z Agregačného frameworku.

6.3 Map Reduce

Map Reduce je programovací model, ktorý bol predstavený spoločnosťou Google v roku 2004. Jeho implementácia slúži na spracovávanie veľkých dátových súborov, ktorá je použiteľná v širokom spektre reálnych úloh[9]. Úživateľ definuje funkcie map a reduce a systém zabezpečuje automatickú paralelizáciu výpočtu a komunikáciu medzi viacerými zariadeniami. Tento koncept je maximálne využiteľný v rozsiahlom clusteri. Napriek tomu nám nič nebráni v jeho použití aj na jednom zariadení. V MongoDB je tento model implementovaný príkazom db.collection.mapReduce(map, reduce, doc), ktorej parametrami sú JavaScript funkcie map, reduce a dokument s ďalšími parametrami.

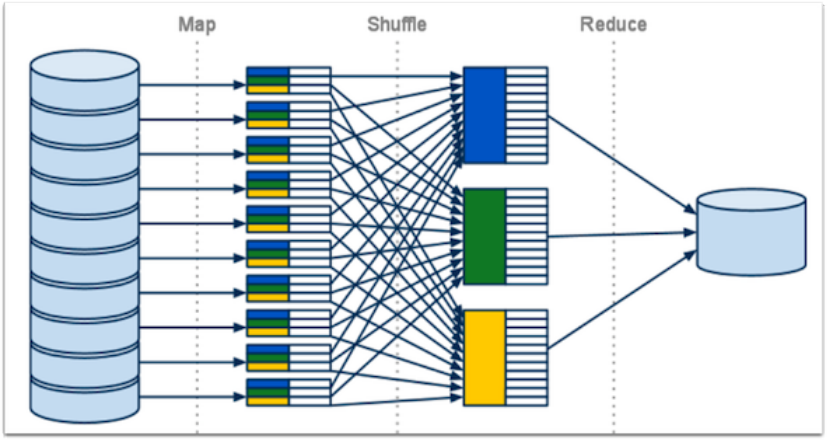
Dokument s ďalšími parametrami špecifikuje:

- názov kolekcie a metódu spojenia s existujúcou kolekciou (povinná položka out), do ktorej bude uložený výsledok map-reduce operácie (ak chceme výsledok priamo bez uloženia do DB, uvádza sa paramater inline),
- filter, ktorý obmedzuje dokumenty vstupujúce do výpočtu (query),
- zoradenie výstupu (sort),

9. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Str. 1

- obmedzenie počtu dokumentov na výstupe (limit),
- JS funkciu finalize, ktorá je volaná po funkcii reduce, pri tejto funkcii je garantované volanie jedenkrát, kďežto funkcia reduce môže byť volaná viackrát na jednej skupine dokumentov,
- globálne premenné použiteľné vo funkciach map, reduce, finalize (scope),
- jsMode transformácia dokumentov na BSON pri spracovávaní a
- pridanie štatistiky spracovávania (verbose).

Podstata celej operácie Map Reduce spočíva vo vygenerovaní dvojíc (kľúč, hodnota) pre každý dokument (obr. 6.4 - krok [Map]). Následne sú automaticky zoskupené tie dvojice (kľúč, hodnota), ktoré majú rovnaký kľúč [krok Shuffle]. Nakoniec je každá takáto skupina dvojíc zredukovaná na jednu hodnotu [krok Reduce].



Obr. 6.4: Priebeh Map Reduce

Funkcia map je volaná na každom dokumente kolekcie alebo na selektovaných dokumentoch (určených parametrom query). Funkcia map musí obsahovať volanie funkcie emit(k,v) s parametrami kľúč a hodnota. V prípade zložitejších agregácii nemusí byť odosielaná z map funkcie len hodnota jednoduchého typu, ale môže to byť aj zložitejší dokument. Dvojice (kľúč, hodnota) sú potom zoskupené do skupín majúcich rovnaký kľúč. Pre každý kľúč teraz existuje pole hodnôt resp. dokumentov. Nasleduje volanie funkcie reduce(k,[v]) s parametrami kľúč a pole hodnôt

(resp. dokumentov), kde dochádza k agregácii a vystupuje jedna hodnota (resp. dokument). Dôležité je dodržať tento prototyp funkcií, počet a typ parametrov a výstupu.

Predpokladajme, že máme kolekciu test obsahujúcu merania spotreby uložené ako hodnoty (value) z rôznych zdrojov (source), ktoré majú nasledujúcu schému:

```
{
 _id:          ObjectId("50a8240b927d5d8b5891743c"),
 source:       "13333@CT",
 date:         ISODate("2013-01-01T12:00:00+01:00"),
 consumption:  100
}
```

Použitím Map Reduce sme schopní zoskupiť merania podľa zdroja a pre takéto skupiny meraní vypočítať sumu hodnôt z jednotlivých zdrojov. Ekvivalentný dotaz v SQL pre riešenie tejto otázky by mal nasledujúci tvar : SELECT source, sum(consumption) FROM test GROUP BY source.

Pre Map Reduce zadefinuje funkcie map a reduce (obr. 6.5).

```
> map = function() {
      emit(this.source, this.consumption);
      };
> reduce = function(source, values){
         return Array.sum(values);
         };
```

Obr. 6.5: JavaScript funkcie Map a Reduce

Zavolaním príkazu mapReduce (obr. 6.6) sa na výstup vypíšu jednotlivé sumy z každého zdroja.

```
> db.test.mapReduce(map, reduce, {out: {inline: true}});
[{
 _id:    "X",
 value: 100000
},{
 _id:    "Y",
 value: 200000
}}]
```

Obr. 6.6: Prevedenie Map Reduce operácie

Na to aby bol zabezpečený korektný výpočet pomocou modelu Map Reduce, sú na redukčnú funkciu kladené nasledujúce podmienky:

- výstupný typ objektu je rovnaký ako typ objektov vo vstupnom poli, a súčasne redukované pole môže byť ľubovolne rozdelené na časti a zredukované tak, aby platilo:

$$reduce(key,[C,reduce(key,[A,B])]) == reduce(key,[C,A,B])$$

- funkcia je idempotentná[10]:

$$reduce(key,[reduce(key,valuesArray)]) == reduce(key,valuesArray)$$

- nezáleží na poradí prvkov v poli[11]:

$$reduce(key,[A,B]) == reduce(key,[B,A]).$$

6.4 Zhodnotenie

Významným rozdielom medzi Map Reduce a Agregačným frameworkom spočíva v spracovaní príkazov na strane databázy. V prípade Map Reduce sú JavaSript príkazy interpretované JS enginom, kdežto Agregačný framework je skompilovaný C++ kód[12].

10. taká, že platí $f(f(x)) = f(x)$ pre všetky x
11. 10gen. MongoDB Manual, mapReduce
12. Z., W. Map-Reduce performance in MongoDb 2.2 and 2.4

Vo verzii MongoDB 2.4.3 (4/23/2013) došlo k zlepšeniu a ako JS engine bol nasadený V8 od Google, ktorý dokáže spracovaváť JS operácie vo viacerých vláknach (voči predchádzajúcemu jednovláknovému SpiderMonkey), čo je značná podpora v zrýchlení Map Reduce.

Napriek tomu stále JS engine vyžaduje, aby bol každý dokument z MongoDB preformátovaný z BSON do JSON a pri ukladaní do DB zasa preformátovaný naspäť. Spolu s interpretovaním kódu v Map Reduce, tak celkovo dochádza k nižšiemu výkonu ako pomocou Agregačného frameworku.

Implementácia Map Reduce tak nie je úplne vhodná pre použitie na výpočty v reálnom čase. V prípade, že je potrebné vypočítavať agregácie v reálnom čase, je vhodné najprv zvážit použitie Agregačného frameworku a v prípade, že danú funkcionalitu ním nie je možné dosiahnuť, je vhodnou alternatívou predpočítavať agregácie online (s každou novou udalosťou uloženou do DB aktualizovať agregované štatistiky), prípadne použitie inkrementálneho Map Reduce.

Pre rýchle porovnanie doby výpočtov jednotlivých nástrojov uvádzam výsledky rýchlosti vykonania jednoduchej agregácie nameraný v JS konzole v MongoDB pre rovnaké dotazy, ktoré spočítajú súčet hodnôt zoskupených podľa mesiaca (kód v prílohe B). Agregačný framework tento príkaz spracuje za 20ms a Map Reduce 150ms. Z tohto výsledku je jasne vidieť značný rozdiel aj pri tak jednoduchej agregačnej úlohe. 150 ms je čas, ktorá je pre mnoho úloh v reálnom čase nepoužiteľný.

Kapitola 7

Cube

Hotovým produktom určeným na prácu s časovými radami je systém Cube[1]. Beží na platforme Node.js[2], napísaný v jazyku JavaScript a je určený na ukladanie udalostí s časovými známkami a výpočet štatistík z uložených udalostí. Cube používa ako úložisko MongoDB. Obsahuje 2 hlavné aplikačné časti: Collector a Evaluator.

Collector prijíma udalosti od klientov a ukladá ich do MongoDB. Collector funguje ako server a udalosti môžme naň posielať 3 možnými protokolmi: HTTP POST, WebSockets, UDP a navyše Cube podporuje integráciu s collectd[3].

Udalosti sa na Collector posielajú v JSON objekte, ktorý musí obsahovať položky type, time a data.

Druhá časť, Evaluator, zabezpečuje obsluhu dotazov na uložené udalosti a na štatistiky nad udalosťami. Evaluator funguje tiež ako server a podporuje HTTP GET a WebSocket požiadavky. Samotné dotazy na Evaluator je potrebné formulovať v jednoduchom jazyku, ktorý ma Cube definovaný.

Dotazovací jazyk v Cube pozostáva z redukčných a filtorvacích funkcií. V súčasnosti Evaluator obsahuje 5 redukčných funkcií: sum (súčet alebo počet), min, max, median a distinct (počet navzájom rôznych). Ďalšie je možné naprogramovať plne v JavaScripte. Každá redukčná funkcia je JS funkcia, ktorá má parameter pole hodnôt a vracia jednu hodnotu.

Filtrovacie funkcie slúžia na obmedzenie udalostí, ktoré budú vstupovať do výpočtu štatistík. Sú nimi: eq (rovnosť), lt (menšie ako), le (menšie alebo rovné), gt (väčšie ako), ge (väčšie alebo rovné), ne (nerovnosť), re (regulárny výraz), in (rovný jednej hodnote z pola).

---

1. http://square.github.io/cube/
2. Node.js je serverový systém navrhnutý pre škálovateľné webové aplikácie, využíva JavaScript engine V8 vyvinutý spoločnosťou Goggle (http://nodejs.org/)
3. Collectd je program bežiaci na pozadí systému, ktorý v pravidelných intervaloch zaznamenáva výkonnostné štatistiky systému (http://collectd.org/)

Pomocou Evaluatora môžeme jednoducho pristupovať k uloženým udalostiam a s využitím filtrovacích funkcií podľa potreby špecifikovať obmedzenia na položky udalostí. Dotazovať môžme najjednoduchšie cez HTTP GET, ktorý obsahuje zreťazené parametre: expression (dotaz v špecialnom jazyku Evaluatora), start (dátum od), stop (dátum do) a limit (maximálny počet vrátených udalostí). Druhou možnosťou na dotazovanie je WebSocket pomocou JSON súboru, v ktorom sú rovnaké položky ako uvedené parametre HTTP GET.

Dotaz na štatistiky nad udalosťami má rovnaký formát ako pre udalosti, ale navyše obsahuje položku step - časová granularita, hodnota v milisekundách, podľa ktorej sa zoskupujú udalosti. Cube podporuje 5 možných časových granularít, podľa ktorých sa dajú zoskupovať udalosti pre výpočet štatistík: 1e4 (10 sekúnd), 6e4 (1 minúta), 3e5 (5 minút), 36e5 (1 hodina) a 864e5 (1 deň). Samozrejme tentokrát dotaz musí obsahovať použitie aspoň jednej redukčnej funkcie. Pevným obmedzením systému je, že dokáže spracovať maximálne 1000 výstupných udalostí.

Predpokladajme, že máme pomocou Collectora uložené udalosti rovnakého typu ako v ukážke. Dotaz na posledných 10 požiadaviek na webový server, ktoré smerovali na adresu "/search" by vyzeral nasledovne:

```
{
    "expression": "request.eq(path, '/search')",
    "limit": 10
}
```

Druhý typ dotazov na štatistiky nad udalosťami môžme využiť na agregačný dotaz, napríklad na počet webových požiadaviek v jednotlivých dňoch (1 den = 864e5 milisekúnd) od 1.1 do 31.1 by vyzeral nasledovne:

```
{
  "expression": "sum(request)",
  "start": "2013-01-01T00:00:00.000Z",
  "stop": "2013-01-31T00:00:00.000Z",
  "step": 86400000
}
```

Prvé riadky z výstupu:

```
{"time": "2013-01-01T00:00:00.000Z", "value": 1512}
{"time": "2013-01-02T00:00:00.000Z", "value": 1817}
{"time": "2013-01-03T00:00:00.000Z", "value": 1725}
```

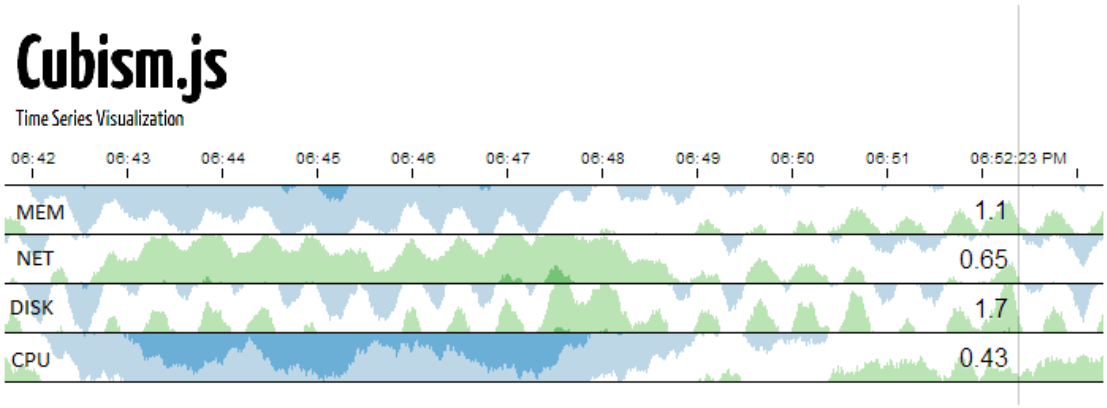
Zložitejšie dotazy môžme vytvoriť z viacerých redukčných a zreťazených filtrovacích funkcií. Dotazy môžu taktiež obsahovať aritmetické operácie na položkách

udalostí. Napríklad výpočet priemernej dĺžky odpovede v sekundách na požiadavku smerovanú na úvodnú stránku s kódom odpovede HTTP 200:

```
sum(request(duration/1000).eq(status, 200).eq(path, "/"))
 /sum(request.eq(status, 200).eq(path, "/"))
```

Štatistické výstupy sú súčasne ukladané aj do cache. Ak klient požaduje výstup s identickým dotazom, je vyberaný priamo z tejto cache. Na to je využívaná limitovaná kolekcia v MongoDB, ktorá má definovanú pevnú velkosť, ktorú nemôže presiahnuť a v prípade, že sa zaplní, nové udalosti nahrádzjú tie, ktoré boli do nej uložené ako prvé.

Ďalšia implementovaná metóda, ktorá zrýchľuje výpočet štatistík je hierarchická agregácia. Každá granularita tvorí pomyselnú vrstvu od najnižšej – 10 sekúnd po najvyššiu – 1 deň, a štatistiky na vyšších vrstvách sú vypočítavané z najvyššej možnej vrstvy pod aktuálnou. To znamená, že napríklad v situácii, že dotazujeme štatistiky s granularitou 1 hodina a súčasne je v cache uložený výsledok z totožného dotazu, ale s granularitou 5 minút, štatistiky sa vypočítajú z existujúcich 5 minutových a nie až zo samotných udalostí na nultej úrovni.



Obr. 7.1: Vizualizácia výstupu Cube pomocou knižnice Cubism.js

Výhodou systému Cube je integrovaná knižnica Cubism, postavená na grafickej JS knižnici D3, určená na vizualizáciu časových radov. Na vykreslovanie numerických výsledkov do grafu stačí len napísať niekoľko riadkov JS kódu a požadovaný dotaz, a Cubism zabezpečí pravidelné dotazovanie a vykreslovanie grafov (obr. 7.1) s dátami v takmer reálnom čase.

Z uvedených vlastností systému Cube vyplýva, že je to jednoduchý, ale použiteľný nástroj v prípade, že si vystačíme so základnými operáciami, teda uloženie

a dotazovanie udalostí (s možnosťou filtrovania), a výpočet štatistík počet, súčet, minimum, maximum, medián nad uloženými udalosťami a neprekáža nám komunikácia prostredníctvom webových protokolov. Vhodným využitím je ilustrovaná vizualizácia štatistík priamo vo webovom prehliadači. Na rozšírenie tohto systému o novú funkcionalitu je ale potrebné disponovať pokročilými znalosťami jazyka JavaScript.

Kapitola 8

Dotazovacie rozhranie

Súčasťou tejto práce je aj hotové dotazovacie rozhranie, ktorého základom sú metódy vytvárajúce štatistické dotazy nad dátami typu časových radov. Implementované je v jazyku Java a využíva úložisko MongoDB.

8.1 Štatistické dotazy

Rozhranie používa metódu Fluent interface - reťazenie metód, ktorého cieľom je vytvoriť čitateľnejší a plynulejší kód[1]. V našom prípade je to skôr sekvenčné tvorenie dotazu na DB, ktorý je obmedzený na voľbu operácií, ktoré môžu za sebou nasledovať a sú ukončené spustením agregačnej funkcie. Pre lepšiu prehladnosť je súčasťou práce je aj analytický class diagram v prílohe C.

Projekt obsahuje konfiguračný súbor config.properties, v ktorom je potrebné definovať adresu, port a názov databázy MongoDB a ďalej výber implementácie výpočtu dotazov. Súbor js_command.js obsahujúci preddefinované JS map a reduce funkcie, ktoré sa pri spustení uložia do DB a v Java rozhraní sa predáva v textovom reťazci odkaz na tieto funkcie. Tento JS súbor je možné ďalej rozširovať a pridávať tak novú funkcionalitu.Trieda Manager slúži na vytvorenie spojenia s DB, inicializáciu DB, vytváranie inštancií Query a inštancií Preaggregate.

Trieda Query aktuálne obsahuje štatistické funkcie počet, súčet, priemer, minimum a maximum. Rozhranie ďalej obsahuje metódu causeFor() určenú na jednoduché zisťovanie príčin udalostí prostredníctvom Agregačného frameworku. Využiteľná je v situáciach, kedy predpokladáme kauzalitu udalostí. Pre potreby vyhľadania týchto príčinných udalostí, je potrebné identifikovať viacero udalostí, ktoré sú následkami, pomocou selekčného výrazu (dokumentom) v MongoDB. Ďalej zvolíme atribút udalosti, na základe ktorého budú zoskupené udalosti predchádzajúce vzniknutým udalostiam. Výstupom je množina hodnôt zvoleného atribútu, zoradená zostupne podľa početnosti výskytu. Tento postup vyjadruje len

1. Martin Fowler. Fluent Interface

naivnú metódu hľadania príčin a výsledky je nutné prehodnotiť, pretože môžu vzhľadom na rôzne typy udalostí podávať správne výsledky, ale aj úplne zavádzajúce.

Trieda Query slúži na hromadný výpočet štatistík v skupinách udalostí, ktoré vzniknú zoskupovaním na základe delenia dátumu prevedeného na milisekundy. Takáto metóda je rozšírením odstrihávania časových zložiek z dátumu. V nej môžme dosiahnúť odstrihnutie minút a potom využívame zoskupenie udalostí majúcich rovnaký hodinový údaj a ľubovolný minútový , ale nedovoľuje zoskupenie v necelých časových jednotkách napr. polhodiny [viac pozri výpočet step v tab. 8.1]. Na výpočet štatistických funkcií je na výber z 3 implementácií definovateľných v konfigurácií alebo v Manager aj v Query metódou setImplementation():

AggregationFramework(1) MapReduce(2) MapReduceCached(3)

Implementácia 1 používa Agregačný framework, ktorý je najrýchlejší, ale má obmedzené možnosti práve implementovanými funkciami v Agregačnom frameworku [odkaz]. Implementácia 2 využíva metódu Map Reduce, ktorý bol pridaný kvôli širším možnostiam pre výpočet zložitejších funkcií ako napr. rozptyl, kovariancia či korelácia, či iných požiadaviek, ktoré sa nedajú dosaihnúť predchádzajúcou implementáciou.

Implementácia 3 je rozšírením 2 o ukladanie výsledkov späť do DB.

V implementácii 3 (MapReduceCached) prebieha ukladanie výsledkov dotazu do kolekcie v MongoDB (tzv. cache), takže pri totožnom dotaze nie je spúšťaný výpočet znova, ale výsledok sa číta priamo z cache. Takýto dotaz musí byť totožný s dotazom v minulosti, ktorý uložil výsledky do cache, okrem časových hraníc. Ak je v cache výsledok totožného dotazu, ale nie v úplne celom časovom rozsahu, aký aktuálne požadujeme, výpočet sa spúšťa len na nevypočítanom rozsahu. Potom sú výsledky uložené do cache a nakoniec je kompletný výsledok vrátený z cache. Tento spôsob je určený pre súbory udalostí, v ktorých už nepribúdajú nové udalosti. V prípade, že by boli ukladané nové udalosti, bolo by potreba invalidovať cache, aby sa zabezpečil výpočet aj s novou udalosťou. Túto funkcionalitu táto implementácia zatiaľ nepodporuje.

Inštanciu Query je potrebné vytvoriť cez Manager s uvedením názvu kolekcie, na ktorej budú vykonávané výpočty.

```
Manager m = new Manager();
Query q = m.createQueryOnCollection("events");
```

V Query je možné filtrovať dokumenty, na ktorých prebehne výpočet. Napr. ak chceme len dokumenty, ktoré majú položku responseMs menšiu ako 10 a protocol rovný “GET”, pomocou fluent api zapíšeme príkaz:

```
q.field("responseMs").lessThan(10).field("protocol").equal("GET");
```

Filtrovacími metódami na položke dokumentu môžu byť :

- exists() //položka existuje v dokumente
- doesNotExist() //položka neexistuje v dokumente
- equal(obj) //položka je rovná hodnote
- notEqual(obj) //položka nie je rovná hodnote
- lessThan(obj) //položka je menšia ako
- lessThanOrEq(obj) //položka je menšia alebo rovná
- greaterThan(obj) //položka je väčšia ako
- greaterThanOrEq(obj) //položka je väčšia alebo rovná
- hasOneOf([obj]) //položka je rovná jednému prvku z poľa [obj]
- hasAllOf([obj]) //položka obsahuje pole a sú v ňom všetky prvky z poľa [obj]
- hasNoneOf([obj]) //položka obsahuje pole a nie sú v ňom žiadne prvky z poľa [obj]
- hasThisOne(obj) //položka obsahuje pole a niektorý prvok je rovný obj
- hasThisElement(obj) //položka obsahuje pole a niektorý prvok spĺňa všetky podmienky obj

Dôležitým údajom pre výpočet je step - hodnota v milisekundách, podľa ktorej sa budú zoskupovať udalosti. Nastaviť je ju potrebné pomocou metódy setStep(long), inak sa použije prednastavená hodnota 1000 ms. Narozdiel od Cube, step nie je obmedzený len na výčet možných granularít, ale je možné definovať ľubovolnú kladnú hodnotu. Kvôli čitateľnosti je vhodné použiť triedu TimeUnit na prepočet vyšších časových jednotiek na milisekundy. Napr. 1 deň na milisekundy:

```
long oneDay = TimeUnit.DAYS.toMillis(1); //86400000
```

Vo výpočte potom do jednej skupiny udalostí spadnú tie, ktorých dátum reprezentovaný ako počet milisekúnd, ktoré uplynuli od počiatku UNIX letopočtu, po odčítania zvyšku po celočíselnom delení step-om, je rovnaký.

kluc = datum ms - datum ms mod step

Tabuľka 8.1: Výpočet kľúča

Dátum	Dátum v ms	Kľúč v ms	Klúč ako dátum
2013-01-01T22:12:52	1357078372000	1356998400000	2013-01-01T00:00:00
2013-01-02T02:45:05	1357094705000	1357084800000	2013-01-02T00:00:00
2013-01-02T13:05:41	1357131941000	1357084800000	2013-01-02T00:00:00

Napríklad z nasledujúcich 3 dátumov (v tabuľke 8.1), kde posledné 2 majú rovnaký deň, spadnú do rovnakej skupiny, pretože majú rovnaký kľúč pri použití step 86400000 ms = 1 deň.

Z tohoto dôvodu v celej práci, ak je uvedený dátum ako charakteristický pre interval, je ním myslený začiatok intervalu, a koncom intervalu je daný dátum + step.

Časové hranice vrámci ktorých chceme vypočítať štatistiky definuje pomocou metód fromDate(date) a toDate(date). Hranicu je možné obmedziť aj jednostranne, vtedy výpočet prebieha na všetkých udalostiach v DB od, resp. do daného dátumu. V prípade, že neurčíme žiadnu hranicu, výpočet prebieha na všetkých uložených udalostiach v DB.

Výstup je možné zoradiť podľa dátumu alebo vypočítaných hodnôt prostredníctvom metód orderDateDesc(), orderDateAsc() alebo orderAsc(field), orderDesc(field).

Jednoduché načítanie udalosti z DB, ktoré spĺňajú podmienky definované spomínanými metódami, sa vykoná pomocou metódy find(), ktorá ukončí Query. V takom prípade je možné použiť radenie podľa viacerých položiek zavolaním metódy order viackrát. Vtedy sa intuitívne radí primárne podľa položky z prvého volania order, sekundárne podľa ďalšej atď.

Napr. príkaz výberu maximálne 10 udalostí, ktoré majú položku value väčšiu ako 1000 zoradených zostupne podľa 2 položiek :

```
q.field("value").greaterThan(1000).ordereDesc("type")
    .orderDesc("response").limit(10).find();
```

Výpočet štatistík nad udalosťami sa vykoná pomocou nasledujúcich metód, ktoré ukončia Query s položkou, na ktorej má byť vypočítaná daná štatistika:

- count(field)
- sum(field)
- avg(field)

- max(field)
- min(field)

8.1.1 Štatistické dotazy implementované pomocou Aggregačného frameworku

Query v prvej implementácii zabezpečí vytvorenie pipeline, ktorá sa následne pošle ako paramater Agregačného frameworku do MongoDB. Táto pipeline pozostáva z operátorov match, group, sort a limit v tomto poradí. Operátory match, sort a limit sú len jednoduché dokumenty zostavené na základe nastavovacích metód volaných na Query. Najdôležitejší je operátor group (obr. 8.1), ktorý zabezpečuje zoskupovanie podľa času. Agregačný framework podporuje síce funkcie pre prácu s dátumami, no patria medzi ne len extrakcie jednotlivých časových jednotiek z dátumu, preto je pre vytvorenie zoskupovacieho kľúča _id potrebné dátum udalosti previesť na milisekundy po častiach a následne odčítať zvyšok po delení step-om. Druhá časť operátora group value obsahuje už samotný subdokument s agregačnou funkciou aplikovanou na položke udalosti (konkrétne súčet { $sum: "$value "}).

```
{ "$group" : {
  "_id" : {
    "$subtract" : [ "$date" , { "$mod" : [ {
    "$add" : [
       { "$multiply" : [ 31536000000 , {"$year":"$date"}]},
       { "$multiply" : [ 86400000,{"$dayOfYear":"$date"}]},
       { "$multiply" : [ 3600000 , { "$hour" : "$date"}]},
       { "$multiply" : [ 60000 , { "$minute" : "$date"}]},
       { "$multiply" : [ 1000 , { "$second" : "$date"}]},
       { "$millisecond" : "$date"}]}
     , step ]}]} ,
  "value" : { "$sum" : "$value"}}}
```

Obr. 8.1: Operátor group pre pipeline v Agregačnom framweworku

Jednotlivé štatistické metódy v Query sa líšia práve spomínaným subdokumentom value, ktorý obsahuje príslušné aplikované agregačné funkcie (ukážka zobrazuje operátor pre metódu count()). Pre komplexnejšie štatistické funkcie

je možné využiť univerzálnu štatistickú metódu aggregate(). Pomocou ktorej je možné vypočítať viacero štatistických funkcií naraz alebo vykonať aritmetické operácie na položke pred agregáciou.

Napr. pre výpočet všetkých štatistík naraz je potrebné vytvoriť nasledujúci dokument a predať ho metóde aggregate() (obr. 8.2).

```
{
 "pocet" :    {"$sum" : 1},
 "suma" :     {"$sum" : "$value"},
 "maximum" : {"$max" : "$value"},
 "minimum" : {"$min" : "$value"},
 "priemer" : {"$avg" : "$value"}
}
```

Obr. 8.2: Výpočet všetkých štatistických operácii v Agregačnom frameworku

8.1.2 Štatistické dotazy implementované pomocou Map Reduce

Query v druhej implementácii využíva metódu mapReduce(), kde je pre každú štatistickú metódu dosádzaná príslušná JS funkcia reduce a ostatné parametre sú identické. Všetky JS funkcie sú pri vytvorení inštancie Manager vložené do špecialnej kolekcie db.system.js v MongoDB, ktoré je potom možné využívať pri Map Reduce (inak by museli byť vždy celá definícia funkcie súčastou príkazu Map Reduce).

Tieto funkcie je možné jednoducho meniť aj počas behu priamo v spomínanej kolekcii štandardnou operáciou update a nie je potreba meniť dotazovacie rozhranie. Predmetom pridávania alebo zmeny funkcionality sú väčšinou funkcie Map a Reduce. Map pre situácie kedy je napr. potrebné vyberať viacero položiek alebo medzi položkami, či na samotných položkách vykonávať aritmetické, dátumové, textové operácie. Práve tieto operácie môžu byť rozhodujúce pri uprednostnení Map Reduce pred Agregačným frameworkom. V AF sme vo veľkej miere limitovaný operáciami, ktoré má implementované, kdežto v Map Reduce je možné využiť všetky možnosti jazyka JavaScript. Zmena Map následne vyžaduje zmenu Reduce funkcie tak, aby rátala s novými položkami. Druhým typom zmeny Reduce funkcie môže byť aj iný spôsob redukovania odlišný od klasických štatistických agregácií.

Napr. niektoré aplikácie môžu vyžadovať hľadanie extrémne odlišných hodnôt od priemerných (tzv. outlier), v takom prípade map funkcia len odosiela hodnoty a reduce zodpovedá za výpočet priemeru a smerodatnej odchýlky a identifikuje outlier - často sa ním považuje hodnota väčšia ako trojnásobok smerodatnej odchýlky. Tieto 3 hodnoty by boli výstupom reduce funkcie. V prípade, že outlier-ov je viac, je možné ich vložiť do pola.

V prípade, že potrebujeme do výpočtu predávať ďalšie premenné, ako v prípade tejto implementácie, kedy vkladáme do výpočtu konštanty step a názov položky na redukovanie, je možné využiť globálne premenné príkazu mapReduce.

Bohužial Map Reduce nemôže byť použitý na výpočet mediánu (všeobecne žiadneho kvantilu) v jednoduchej implementácii. Tou býva nasledujúci algoritmus: 1. zoraď pole, 2. vyber stredný prvok, ak je dĺžka pola nepárna inak priemer 2 stredných prvkov[2]. Redukčná funkcia nespĺňa podmienku, ktorá je kladená na ňu, kvôli možnosti behu funkcie reduce na jednej skupine hodnôt viackrát (po častiach):

$$reduce(key,[1[reduce(key,[2,3])]) == reduce(key,[1,2,3])$$

Pre reduce implementovanú na výpočet mediánu:

$$reduce(key,[1,2.5]) \neq 2$$

$$1{,}75 \neq 2$$

8.1.3 Štatistické dotazy implementované pomocou Map Reduce s cache

V tejto implementácií štatistický dotaz funguje rovnako ako v predchádzajúcej, s tým rozdieľom, že výsledok každého Map Reduce je ukladaný do cache kolekcie. Súčasne sú do ďalšej kolekcie vložené 2 dokumenty: začiatok so značkou start a koniec so značkou end, ktoré reprezentujú interval, v ktorom bol vypočítaný aktuálny dotaz.

Táto pomocná kolekcia slúži k tomu, aby sme vedeli identifikovať intervaly, v ktorých sú už výsledky uložené a doplnky k týmto intervalom, v ktorých bude potrebné spúšťať výpočet.

Celý proces určenia podintervalov k aktuálne dotazovanému intervalu, na ktorých ešte neprebehol výpočet a spustenie map reduce na ňom, prebieha podľa nasledujúceho pseudokódu (obr 8.3).

---

2. funkcia, ktorá vracia 1 hodnotu - medián. Na korektný výpočet mediánu pomocou Map Reduce by bolo potreba vracať všetky hodnoty a súčasne medián, čo je pamäťovo náročné

```
znacky = najdiZnackyVIntervale(zaciatok, koniec)
if |znacky| = 0 then
        if najdiZnackuPred(od) = start then
                skonci
        else
                mapReduceNaIntervale(zaciatok, koniec)
                skonci
else for znacka in znacky do
        if znacka = start then
                if pom = null then
                    mapReduceNaIntervale (zaciatok, znacka)
                else
                    mapReduceNaIntervale(pom, znacka)
                    pom = null
        else pom = znacka
if pom != null then
        mapReduceNaIntervale(pom, koniec)
vlozNoveZnacky(zaciatok, koniec)
```

Obr. 8.3: Pseudokód zistenia neprepočitaných intervalov v cache

Po skončení tohoto algoritmu sú v cache uložené výsledky štatistík v celom intervale definovanom v dotaze. Taktiež je zabezpečené, že nad žiadnou udalosťou neprebehol výpočet viackrát. Táto implementácia je vhodná pre také nasadenie, kde prevláda dotazovanie tých istých štatistík (nie nutne v totožnom časovom intervale). Takýmto prípadom je napr. vizualizácia grafu štatistík s možnosťou časového posunu. Napr. užívateľ požiada o hodinové štatistiky v týždni od pondelka do nedele a potom znova požiada o rovnaké štatistiky, ale s posunom jeden deň, teda od utorka do pondelka. Určite by nebolo efektívne, aby boli takmer rovnaké dotazy vypočítavané vždy znova. Táto implementácia zabezpečí výpočet jedenkrát a pri ďalšom dotaze sa pristupuje k hotovým výsledkom v cache.

Pri definovaní príkazu je potrebné rozhodnúť sa medzi dostupnosťou alebo konzistenciou pre okamžik, kedy sa ukladá výsledok Map Reduce do cache. Prednastaveným chovaním je, že sa výstupná kolekcia zamkne a nie je možné z nej čítať až kým sa nezapíše kompletný výstup. Ak je ale potreba zabezpečiť dostupnosť neustále, v Map Reduce je možné pridať parameter {nonAtomic: true}. Tento krok

zabráni v zamknutí kolekcie a užívateľ môže z kolekcie čítať aj počas Map Reduce, avšak môže dôjsť k tomu, že budú načítané prechodné stavy výstupnej kolekcie.

8.2 Predpočítavanie agregácií

V prípade, že rýchlosť výpočtu štatistický metód cez Query nie je dostatočná, je možné využiť koncept predpočítavania. Tento prístup však vyžaduje, aby sme vopred vedeli množinu dotazov, ktoré budeme klásť na dáta v databáze. Tzn. že v čase dotazu už budú výsledky pripravené k načítaniu. Negatívnym efektom je, že mnoho výpočtov je prevedených zbytočne a jejich výsledky zaberajú priestor v DB, aj keď nie sú nikdy z DB čítané. To je ale daň za okamžitú odpoveď na dotaz na agregácie. Takýto prístup väčšinou vyžadujú real-time systémy, ktoré potrebujú rýchle a jednoduché metódy dotazovania.

Dôležitým prvkom je voľba schémy agregačných dokumentov, tak aby efektivne využívali možnosti dokumentov. Namiesto ukladania jednotlivých štatistík ako samostatné dokumenty, je lepšie zoskupovať štatistiky do komplexnejšieho dokumentu. Samozrejme voľba správnej schémy pre danú aplikáciu sa nedá zovšeobecniť a vždy je potrebné posúdiť jej štruktúru vzhľadom na účel, jednoduché dotazovanie a hlavne rýchlosť odpovede na pôvodnú otázku, ktorá stojí za dotazom. V prípade, že vytvárame schéma pre aplikáciu, ktorá slúži na vizualizáciu štatistík v určitom časovom období - napr. počet udalostí v každom dni z daného roku, nebude bude vhodné ukladať denný sučet ako samostatný dokument a potom dotazom vyhladať a vybrať 365 dokumentov, ale efektívnejšie bude zoskupiť všetky dni do jedného dokumentu (obr. 8.4), prípadne zvoliť jednotku niekde medzi týmito extrémami napr. mesiac.

Voľba vhodnej schémy je práve o tom, ktoré položky zahrnúť do jedného dokumentu a ktoré naopak vyčleniť do iného dokumentu. Zoskupovaním vytvoríme jednotky, ktorých obsah úzko súvisí a veľká časť obsahu je relevantná pri čítaní alebo zapisovaní daného dokumentu. Cieľom je tak zjednodušenie, koncentrovanie súvisiacich dát a najmä zrýchlenie prístupu.

Zoskupenie štatistík do jedného dokumentu nemusí byť ale len z jednej časovej úrovne. Pri predpočítavaní vo viacerých časových úrovniach napr. v minútach, 2 minútach, 5 minútach, 30 minútach a hodinách súčasne, by mohli byť všetky agregácie zlúčené v jednom dokumente. Ten by výsledne obsahoval 60 položiek (minúty) + 30 (2 minúty) + 12 (5 minút) + 2 (30 minút) + 1 (hodina) položiek, celkovo 105 položiek. Takýto dokument je dátovo stále malý a prináša veľkú výhodu - s každou novou udalosťou nám stačí aktualizovať len tento jeden dokument,

```
{
   "_id" : ISODate("2013-01-01T00:00:00Z"),
   "yearTotal" : 203931,
   "day" : {
      "1" : {"count" : 52},
      "2" : {"count" : 110},
         ...
      "365" : {"count" : 64}
    }
}
```

Obr. 8.4: Ročný dokument pre ukladanie predpočítaných denných štatistík

teda postačuje jedna operácia upsert[3]. Na druhej strane tiež pri čítaní a vytváraní hodinového prehľadu, postačuje načítať rovnako len jeden dokument.

Táto metóda má však jednu prekážku a tou je rastúca velkosť dokumentu s novými udalosťami, ktoré postupne pridávajú nové časové položky. To by spôsobilo, že mnohokrát počas danej hodiny velkosť dokumentu presiahne priestor, ktorý MongoDB pre neho alokovalo a musel byť realokovaný nový priestor, dokument presunutý a aktualizované indexy. Jednoduchým riešením je prealokácia. Pred prvým upsertom dokumentu sa vytvorí celý dokument s inicializovanými hodnotami všetkých položiek na 0[4]. Tento krok zaistí, že dokument už nikdy nenarastie a nie je potrebné alokovať viac, ako má dokument na začiatku, čo výsledne zefektívňuje využitie priestoru na disku a zrýchľuje zápisy.

V implementovanom rozhraní funkcionalitu predpočítavania zabezpečuje trieda Preaggregate, ktorá poskytuje metódu saveEvent(). Tá zodpovedá za uloženie udalosti do DB a následne aktualizovanie štatistík. Taktiež zabezpečuje prealokáciu pri vytváraní nového dokumentu. Parametrom tejto metódy sú okrem udalosti aj dvojíca časová granularity (step) a počet agregácií, ktoré budú uložené v jednom dokumente. Podporované je vytváranie agregácií na viacerých časových úrovniach, preto je akceptované aj pole dvojíc (granularita, počet).

---

3. update or insert - operácia, ktorá aktualizuje dokument a v prípade, že neexistuje, vytvorí nový dokument
4. 10gen. MongoDB Manual, Pre-Aggregated Reports

Predpokladajme, že vytvárame týždňové agregácie, ktoré ukladáme do mesačného[5] dokumentu (obr. 8.5). Metódu saveEvent voláme s ukladanou udalosťou v nasledujúcimi formáte:

```
saveEvent(TimeUnit.Day, new int[][]{{7,4}}, event);
```

Z tejto definície vyplýva, že dokument obsahuje 4 položky pre agregácie. Tie udalosti, ktoré nastali v rovnakom týždni sú zoskupené a zredukované do jednej hodnoty. Udalosť s dátumom 6.2 bude zredukovaná s ostatnými v rovnakom týždni 31.1 - 6.2 a aktualizuje mesačný dokument, ovplyvní tak jednu položku.

```
{
  "_id" : ISODate("2013-01-31T00:00:00Z"),
  "week" :{
        "0" : {"count" : 1},
        "1" : {"count" : 0},
        "2" : {"count" : 0},
        "3" : {"count" : 0},
   }
}
```

Obr. 8.5: Mesačný dokument pre ukladanie predpočítaných týždenných štatistík

Doposiaľ boli štatistiky vytvárané v preskakujúcom časovom okne. Každá udalosť jednoznačne spadala vždy len do jedného okna - časového intervalu. Rozšírením tohto konceptu sú kĺzajúce štatistiky. Najčastejšie používaným je kĺzajúci priemer. Ten je najviac používaný v technickej analýze v časových radách za účelom vyhladenia dát, ktoré odstraňujú krátkodobé odchýlky a zvýrazňujú trendy či cykly.

Týždňovú agregáciu budeme zaznamenávať každý deň. Mesačný dokument tak bude obsahovať namiesto 4 až 30 položiek, a každá z nich reprezentuje týždňovú agregáciu – daný deň $\pm$ 3 dni [odkaz tabulka.]. Udalosť s dátumom 3.2 bude zredukovaná s príslušnými v daných týždňoch postupne od 31.1 až po 6.2. Výsledne teda ovplyvní 7 položiek v mesačnom dokumente.

Kĺzajúce štatistiky je možné vytvárať pridaním 2 prvkov na koniec pola, ktoré vyjadrujú rozsah kĺzajúcej granularity. Agregácie na obr. 8.6 vytvárame volaním:

5. kvázi mesiac, ktorý má vždy 28 dní a nie je prispôsobený premenlivým mesiacom v gregoriánskom kalendári

```
{
  "_id" : ISODate("2013-01-31T00:00:00Z"),
  "weekInDay" :{
        "0" : {"count" : 1},
        "1" : {"count" : 1},
        "2" : {"count" : 1},
        "3" : {"count" : 1},
        "4" : {"count" : 1},
        "5" : {"count" : 1},
        "6" : {"count" : 1},
        "7" : {"count" : 0},
        ...
        "29" : {"count" : 0},
   }
}
```

Obr. 8.6: Mesačný dokument predpočítaných kĺzajúcich týždenných štatistík

```
saveEvent(TimeUnit.Day, new int[][]{{1,30,3,3}}, event);
```

Paramater by {1,30} symbolizoval vytváranie denných agregácii ukladaných do mesačného dokumentu a {1,30,3,3} značí zväčšenie rozsahu, v ktorom je zaznamenávaná jedna agregácii o ± 3 dni, čo je celkovo 1 týždeň.

8.2.1 Predpočítavanie pomocou aktualizácie - upsert

Koncept predpočítavania pomocou operácie upsert je jednoduchý. S každou novou zaznamenanou udalosťou sa aktualizuje príslušný dokument obsahujúci štatistiky. Nevýhodou update operácie v MongoDB je, že podporuje len pričítanie alebo odčítanie pevnej hodnoty od hodnoty položky v dokumente. Nie je tak možné jednou operáciou aktualizovať položku hodnotou inej položky alebo napr. zmeniť ju na násobok. Jednoduchá update operácia je teda použiteľná iba na predpočítavanie sumy (použijeme inkrementovanie o hodnotu z udalosti) alebo počtu (inkrementovanie o 1). Pre ostatné operácie je nutné najprv načítať štatistiku z DB, v aplikácii ju prepočítať a naspať uložiť do DB. Súčasne je potrebné ošetriť situáciu, keď môže súbežne prebiehať iný výpočet, ktorý načítal tú istú štatistiku a chce ju aktualizovať a vznikol by tak nekonzistentný stav.

Proces komunikácie by mal prebiehať nasledovne:

1. načítaj dokument so štatistikou z DB, ktorá ma byť aktualizovaná
2. preveď v aplikácií jej aktualizáciu o hodnotu z novej udalosti
3. aktualizuj v DB taký dokument, ktorý je presne zhodný s tym, ktorý bol načítaný v bode 1
4. ak neexistuje zhodný dokument na aktualizáciu v bode 3, znamená to, že iný proces prebehol súbežne a aktualizoval hodnotu skôr a celý proces je potrebné začať znova od bodu č.1.

Alternatívnou metódou je verzovanie dokumentov. Ku každému dokumentu pridáme novú položku version: 1 a pri každej aktualizácií dokumentu, túto hodnotu inkrementujeme. Predchádzajúci proces sa zmení v bode č.3, kedy aktualizujeme taký dokument v DB, ktorý ma rovnaku hodnotu verzie ako verzia, ktorú sme pred tým načítali a vtedy môžme dokument aktualizovať, ak je hodnota verzie väčšia, dokument bol už iným procesom zmenený a je potrebné začať odznova. Tento proces všeobecne popísal, ako zabezpečiť atomicky sériu operácií čítanie a zápis, ak ho DB nepodporuje.

8.2.2 Predpočítavanie pomocou inkrementálneho Map Reduce

Napriek tomu, že Map Reduce nie je odporúčaný pre použitie vo výpočtoch v reálnom čase, inkrementálna varianta predstavuje alternatívu k jednoduchému aktualizovaniu štatistík. Inkrementálny Map Reduce sa vo všeobecnosti používa v prípadoch, keď dáta, ktoré budú agregované, neustále pribúdajú. Potom namiesto neustáleho vykonávania Map Reduce cez celú množinu dát, spúšťame inkrementálny Map Reduce len na nových dátach.

Túto operáciu by sme mohli spustiť s každou novou udalosťou, no efektívnejšie bude spúšťat výpočet po väčších dávkach, ak to aplikácia dovoľuje.

Tento postup vyžaduje, aby sme vždy vedeli identifikovať v kolekcii len nové udalosti, ktoré ešte nevstúpili do výpočtu Map Reduce a potom ich označili ako spracované, aby neboli započítané viackrát. Na to využijme vlastnosti unikátneho identifikátora ObjectId, ktorý je primárnym kľúčom dokumentov v MongoDB.

ObjectId je 12 bajtový BSON typ pozostávajúci z: 4 bajty počet sekund od začiatku UNIX letopočtu 3 bajty identifikátor zariadenia 2 bajty id procesu 3 bajty čítač začínajúci na náhodnej hodnote.

Identifikátor vytvára rastúcu postupnosť, preto nám stačí zapamätať si prvý a posledný doposiaľ nespracovaný dokument. Tieto 2 identifikátory uložíme ako dokument do ďalšej kolekcie a s každou novou udalosťou aktualizujeme 2. z nich až

dovtedy, kým počet aktualizácií neprevýši veľkosť určenej dávky. Potom spustíme Map Reduce na dokumentoch, ktorých identifikátor je v uzatvorenom intervale prvého a posledného dokumentu, ktorý sme uložili.

Aby sme docielili inkrementálneho Map Reduce, v príkaze [ukážka dole] je potrebné zadať akciu reduce na výstupnej kolekcií. To spôsobí, že ak vo výstupnej kolekcii existuje dokument s rovnakým identifikátorom ako nový dokument z výsledku Map Reduce, je na tejto dvojíci nový a starý dokument zavolaná funkcia reduce a dokument vo výstupnej kolekcií je prepísaný.

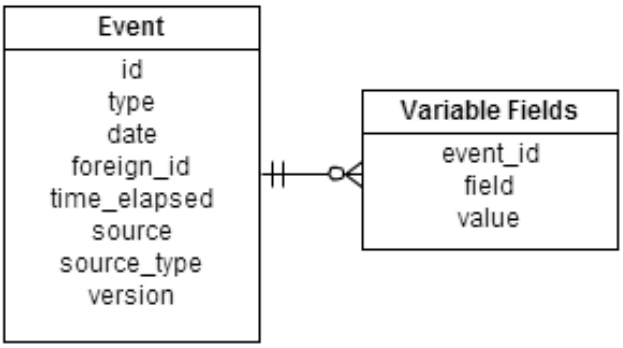
```
mapReduce(map, reduce, {out: { reduce: "stats" }});
```

Bohužial vykonanie Map Reduce na 1 dokumente, je drahšia operácia ako jednoduché aktualizovanie z prechádzajucej podkapitoly. Preto ikrementálny Map Reduce má význam len vtedy, ak je vykonaný na dávke aspoň 5 dokumentov, vtedy je táto operácia rýchlejšia ako 5 aktualizácií [viac test].

Kapitola 9

Porovnanie výkonu

Na testovanie výkonnosti bol použitý open-source framework Caliper v0.5[1] a v niektorých prípadoch manuálne zaznamenávanie času spracovania a následné sprie-merovanie. Pre porovnanie MongoDB voči RDBMS bol použitý systém PostgreSQL. Testy su vykonávané práve na týchto databázových systemóch, pretože cieľom bolo overiť, či je MongoDB schopné nahradiť po výkonnostnej stránke. Využívané dáta a výsledky týchto testov boli podkladom pre moju stáž v spoločnosti Mycroft Mind.



Obr. 9.1: ERD udalosti

Prvé 2 testy merajú rýchlosť uloženia a čítania. Schéma uložených udalostí v PostgreSQL bola vopred definovaná. Charakterizuje ju obr. 9.1. Každá udalosť obsahuje definované atribúty entity Event a ďalšie atribúty, ktoré su premenlivé podľa typu udalosti. Tie sú uložené v druhej tabľke s cudzím kľúčom na id udalosti. V MongoDB som zvolil ekvivalentnú dokumentovú schému, ktorá udržiava rov-

1. https://code.google.com/p/caliper/

```
{
 "_id": 1,
 type:"type",
 date: ISODate("2013-02-01T00:00:00Z"),
 foreign_id: 1,
 time_elapsed: "100",
 source: "local",
 source_type: "system",
 version: 1
 fields: {
        field1: value1,
        field2: value2
        }
}
```

Obr. 9.2: Ekvivalent udalosti v dokumente

naké dáta kompaktnejšie v jednom dokumente (obr. 9.2). Pre variabilné atribúty udalostí využíva voľné schéma v rámci subdokumentu.

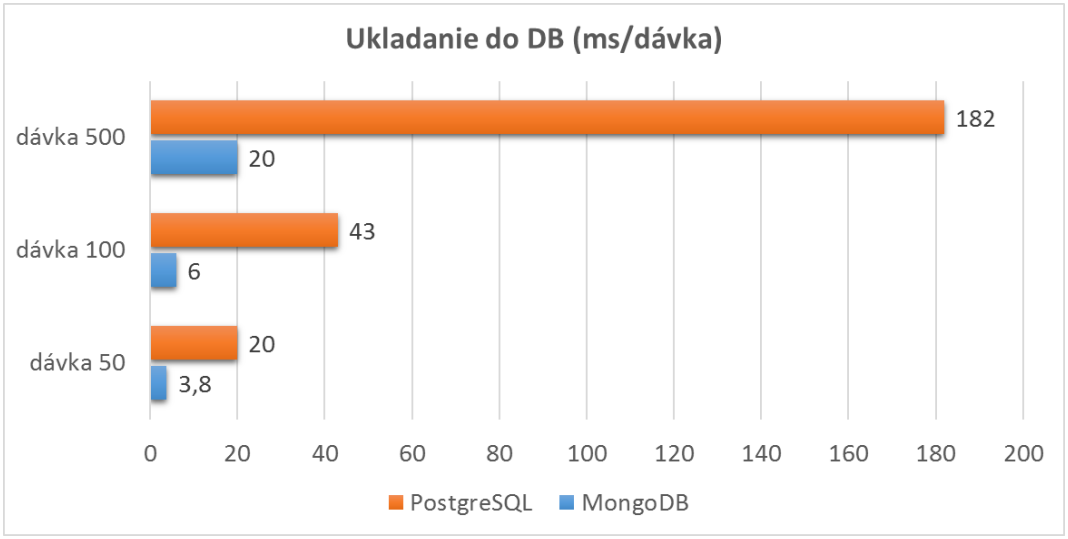
9.1 Ukladanie

Graf (obr. 9.3) zobrazuje porovnanie rýchlosti ukladani týchto údalostí. Ukladané boli rovnaké udalosti po dávkach 50, 100 a 500 objektov. Driver MongoDB podporuje dávkové odosielanie dokumentov, rovnako ako Postgre s využitím transakcií. Vo východzej kolekcii resp. tabuľke bolo uložených 2,5 mil. udalostí s indexami na 2 atributoch.

Pri výbere najrýchlejšej varianty, dávka 500 udalostí, vychádza priepustnosť v MongoDB 25 tis. udalostí za sekundu a PostgreSQL len 2 800 udalostí za sekundu. To predstavuje približne 9x väčšiu rýchlosť MongoDB.

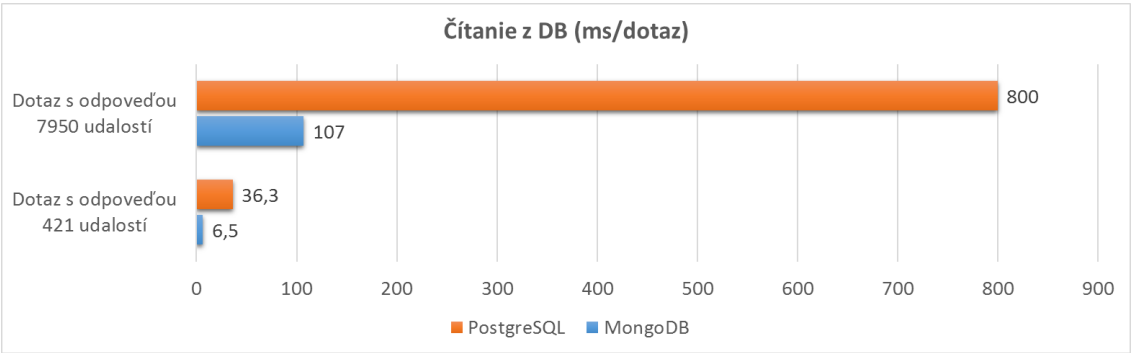
9.2 Čítanie

Na rovnakých dátach potom prebiehal test rýchlosti čítania dát. Použité boli 2 reálne dotazy používané v aplikácii, ktora pracovala s SQL databázou, ktorých výsledkom boli množiny 421 resp. 7950 udalostí (výsledky viď obr. 9.4). Tieto



Obr. 9.3: Ukladanie do databázy (ms/dávka)

dotazy vyberali udalosti na základe časového atribútu. V oboch DB bol index na atribúte podľa, ktorého sa vyberalo.



Obr. 9.4: Čítanie z databázy (ms/dotaz)

Z týchto výsledkov vychádza priemerná priepustnosť 74 tis. udalostí za sekundu v MongoDB a 11,6 tis. udalostí za sekundu v PostgreSQL. Pri čítaní dát je tak MongoDB zhruba 6x rýchlejšie.

Tieto testy bolo vykonané aj na kolekcii/tabuľke s 1000 udalosťami. V takom prípade sa príkazy zrýchlili maximálne o 10 %, čo je znakom dobrého využitia indexov v predchádzajúcom teste. Na druhú stranu ukladanie udalostí úplne bez

indexov je samozrejme rýchlejšie, no čítanie dát z úložiska s niekoľkými milionmi entít metódou table scan (čítanie celej tabuľky) následne zaberie desiatky sekúnd až minút, a nie je teda reálne použiteľné.

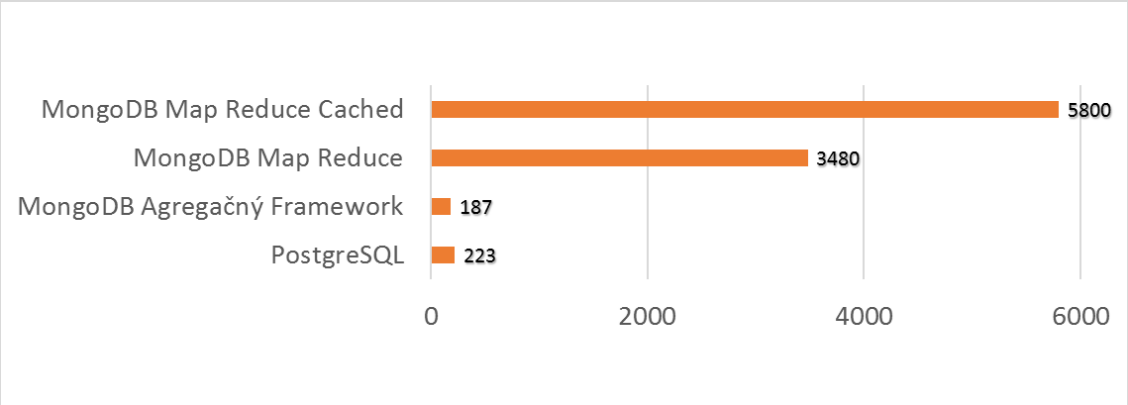
9.3 Query

Nasledujúci test sa týka vlastnej implementácie, konkrétne triedy Query. Porovnáva dĺžku výpočtu súčtu hodnôt v udalostiach, ktoré boli zaznamenané v rovnakej sekunde (ignorujeme milisekundy). Udalosti mali tentokrát jednoduchšiu plochú schému. V oboch systémoch mala udalosť len 3 atribúty: id, dátum a hodnotu. Výpočtet prebiehal na kolekcii/tabuľke, ktorá obsahovala celkovo 100 tis. udalostí, ktoré vytvorili 200 skupín, ktoré sa líšia v sekunde. V každej skupine existovalo 500 udalostí, ktorých hodnoty boli sčítané do jedného čísla a spolu s dátumom odoslané na výstup.

V PostgreSQL je tento výpočet vyjadriteľný SQL príkazom na obr. 9.5.

```
sum(event_value) as sum
FROM meterevent GROUP BY date LIMIT 1000;
```

Obr. 9.5: SQL dotaz ekvivalentný Query



Obr. 9.6: Query (ms)

Z výsledkov je vidieť, že obe databázy v tomto teste obstáli približne rovnako. PostgreSQL ho spracuje za 223 ms a MongoDB 187 ms. Aj pri ostatných štatistických funkciach sú obe databázy zrovnateľné.

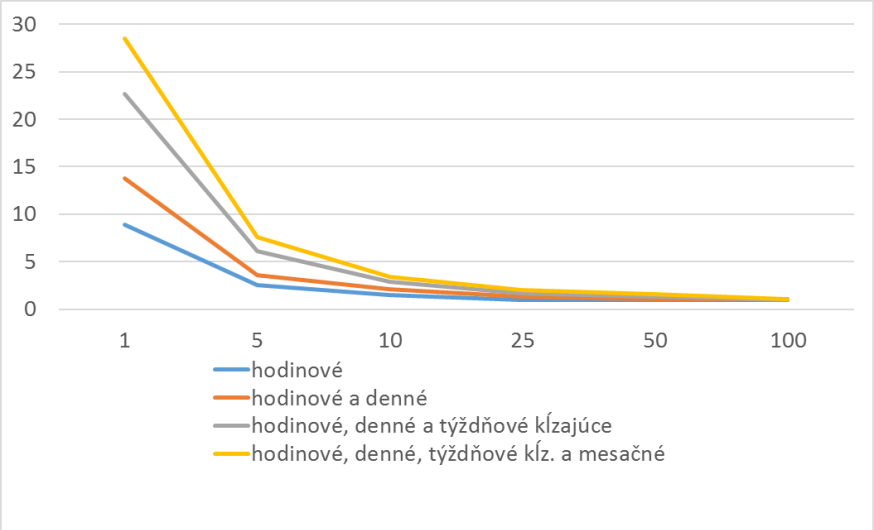
Ak ale zmeníme implementáciu výpočtu Query na Map Reduce, výpočet sa znateľne spomalý až na niekoľko sekúnd. Jednoznačne z toho vyplýva, že Map Reduce v MongoDB nie je použiteľný pre real-time dotazy a je nutné využiť Agregačný Framework, ktorý dosahuje dobrých výsledkov.

Tretia implementácia Query používa Map Reduce rozšírený o trvalé ukladanie výsledkov. Ten vykazuje najdlhší výpočet, takmer až 6 sekúnd. Na druhú stranu ale po vykonaní tejto časovo náročnej úlohy sú výsledky uložené v DB a hneď nasledujucí dotaz ich môže využiť. Takýto dotaz potom trvá priemerne 4ms.

9.4 Preaggregate

Posledné testy merajú výkonnosť predpočítavania (Preaggregate). V každej granularite sú predpočítavané štatistiky súčet, počet a priemer. Tie boli vybrané kvôli tomu, že bývajú najčastejšiou analytickou otázkou.

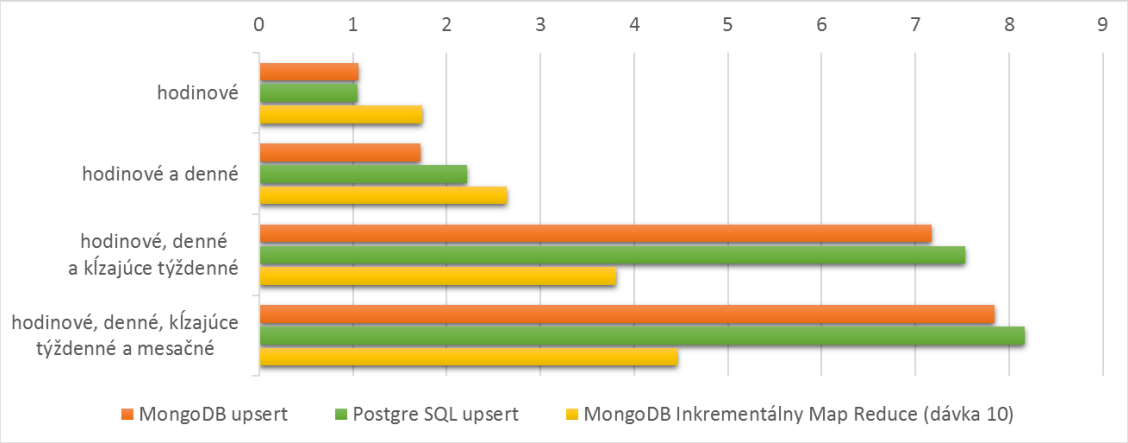
V implementácii Preaggregate s inkrementálnym Map Reduce je však potrebné špecifikovať, v akej veľkej dávke sa bude spúšťať výpočet (nový inkrement). Graf (obr. 9.7) zobrazuje primernú dĺžku výpočtu na jednu udalosť. Pri dávke o veľkosti 1, čo značí, že s každou udalosťou sa spúšta Map Reduce je výpočet neefektívny a je lepšie použiť implementáciu s aktualizáciou štatistík pomocou upsert. Približne od dávky 5 udalostí sa stáva táto implementácia najrýchlejšia. Vhodné je upozorniť, že namerané hodnoty sú priemerné. Takže pri dávke 10 udalostí, v prvých 9 operáciach prebehne len uloženie udalosti pod 1 ms a pri 10. udalosti sa spúšťa inkrementálny výpočet na všetkých 10 udalostiach. Ako je aj vidieť z grafu, ak sa tento výpočet spúšta pri každých 100 udalostiach, priemerne výpočet dosahuje 1 ms aj pri predpočítavaných štatistík vo veľkom množstve časových úrovní (granularít).



Obr. 9.7: Predpočítavanie štatistík s novou udalosťou (ms)

Nasledujúci test porovnáva rýchlosť predpočítavania voči PostgreSQL. Keďže PostgreSQL nepodporuje jednoduchú operáciu upsert, bolo potrebné implementovať túto funkcionalitu pomocou uloženej procedúry [Príloha C]. Vo všetkých množinách časových úrovní predpočítavania je implementácia pomocou aktualizácie v MongoDB (upsert) približne rovnako rýchla ako v PostgreSQL (upsert)

(obr. 9.8). Navyše ale ak je možné využiť Inkrementálny Map Reduce s dávkou väčšou ako 1 (zvolená bola 10), tak v prípade väčšie počtu úrovní dosahuje ešte lepšie výsledky ako upsert v oboch DB. V porovnaní s PostgreSQL v najväčšej množine časových úrovní - hodinových, denných, kĺzajúcich týždenných a mesačných štatistík - inkrementálna implementácia redukuje čas výpočtu na 4 ms.



Obr. 9.8: Predpočítavanie štatistík s novou udalosťou (ms)

9.5 Využitie priestoru

Podstatnou informáciou, nie tak z pohľadu aktuálnej rýchlosti, ale z pohľadu využitia priestoru na disku je celkový priestor, ktorý zaberajú rovnaké dáta v jednotlivých databázach spolu s ďalšími dátami (indexy, interná reprezentácia…).

Na otestovanie využitia priestoru na disku bol do oboch databáz vložený 1 mil. udalostí s 3 atribútmi (integer id, dátum, hodnota double). Tabuľka 9.1 sumarizuje obsadený priestor v týchto databázach. Celkovo MongoDB zabralo o 9 % priestoru viac. Uloženie takýchto plochých dát, teda zaberie približne rovnaký priestor. Rozdiel nastáva v prípade, že ukladáme bohatšie objekty, ktorých normalizáciou vytvoríme viacero tabuliek v RDBMS, ale v MongoDB tieto objekty dokážeme namapovať priamo na jeden dokument. Vtedy MongoDB zaberá menší priestor, pretože v RDBMS potrebujeme duplikovať atribút (cudzí kľúč), na ktorom sa spájajú tabuľky a taktiež vytvoriť nový index na tomto atribúte.

Tabuľka 9.1: Sumarizácia obsadeného priestoru

	dáta udalostí	index na id a dátume	celkovo
MongoDB	64 000 016 B (61 MB)	50 347 808 B (48 MB)	114 347 824 B (109 MB)
PostgreSQL	60 235 776 B (57 MB)	45 006 848 B (43 MB)	105 242 624 B (100 MB)

Kapitola 10

Záver

Cieľom tejto práce bolo analyzovať NoSQL databázy, preskúmať ich možnosti v prípade, že ich využívame ako úložisko monitorovacích dát s charakterom časových rad a taktiež preskúmať nástroje nad nimi postavené. Ukázalo sa, že väčšina hotových nástrojov nahliada na časové rady čisto numerickým pohľadom. Časovú radu chápu ako postupnosť dvojic čas a hodnota. Najčastejšie preto využívajú ako úložisko databázu s modelom rodiny stĺpcov. Z tejto triedy používajú najpopulárnejšiu databázu HBase. Jejich optimalizácia pre časové rady spočíva hlavne v zafixovaní veľkosti prvku časovej rady. Nedovoľujú nám tak ukladať zložitejšie, bohatšie a variabilné dátove štruktúry ako napr. udalosti.

Druhý cieľ práce, vytvorenie dotazovacieho rozhrania nad časovými radami pre monitorovacie účely, bol úspešne naplnený. Podarilo sa vytvoriť systém nad dokumentovou databázou, ktorý disponuje bohatšími možnosťami pre ukladanie udalostí aj meraní. Poskytuje jednak výpočet štatistík nad prvkami uložených časových rad a predpočítavanie dopredu známych štatistík. Pri tvorbe tohoto rozhrania bolo dbané na to, aby mohlo byť jednoducho upravené pre potreby výpočtu iných ako preddefinovaných štatistík.

Výkonostné testy ukázali nevhodnosť použitia RDBMS pri ukladaní rozmanitých dátových štruktúr, ako sú udalosti či logy. Akonáhle sme ale použili ploché dátové štruktúry, RDBMS a NoSQL dosahovali približne rovnakých výsledkov pri použití jedného servera. Pokračovaním práce by bolo vhodné overiť, aké výsledky by táto dvojica dosiahla pri nasadení distribuovaných variant s desiatkami uzlov.

Literatúra

[1] ABADI, D. *Distinguishing Two Major Types of Column-Stores* [online]. 2010-03-29 [cit. 2013-04-15]. Dostupné z: <http://dbmsmusings.blogspot.cz/2010/03/distinguishing-two-major-types-of_29.html>

[2] COCKCROFT, A., SHEAHAN, D. *Benchmarking Cassandra Scalability on AWS - Over a million writes per second* [online]. 2011-11-2 [cit. 2013-04-15]. Dostupné z: <http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html>

[3] DEAN, J. GHENEMAWATT, S. *MapReduce: Simplified Data Processing on Large Clusters*. In OSDI'04, 6th Symposium on Operating Systems Design and Implementation, Sponsored by USENIX, in cooperation with ACM SIGOPS,

[4] ETZION, O. NIBLETT, P. *Event processing in action*. Greenwich: Manning, c2011, xxiv, 360 s. ISBN 9781935182214.

[5] FINLEY, K. *NoSQL: The Love Child of Google, Amazon and … Lotus Notes*. 2012-12-05 [cit. 2013-05-22]. Dostupné z: <http://www.wired.com/wiredenterprise/2012/12/couchdb/all/>

[6] FOWLER, M. *FluentInterface* [online]. 2005-12-20 [cit. 2013-05-10]. Dostupné z: <http://martinfowler.com/bliki/FluentInterface.html>

[7] FOWLER, M. *NosqlDefinition* [online]. 2012-01-09 [cit. 2013-05-15]. Dostupné z: <http://martinfowler.com/bliki/NosqlDefinition.html>

[8] HIGGINBOTHAM, S. *As data gets bigger, what comes after a yottabyte?* [online]. 2012-10-30 [cit. 2013-05-20]. Dostupné z: <http://gigaom.com/2012/10/30/as-data-gets-bigger-what-comes-after-a-yottabyte/>

[9] HOFF, T. *Troubles with Sharding - What can we learn from the Foursquare Incident?* [online]. 2010-10-15 [cit. 2013-04-15]. Dostupné z: <http://highscalability.com/blog/2010/10/15/troubles-with-sharding-what-can-we-learn-from-the-foursquare.html>

[10] MENEGAS, G. *What is NoSQL, and why do you need it?* [online]. 2012-10-01 [cit. 2013-04-15]. Dostupné z: <http://www.zdnet.com/what-is-nosql-and-why-do-you-need-it-7000004989/>

[11] MoreDevs. *OpenTSDB and HBase rough performance test* [online]. 2013-03-28 [cit. 2013-05-09]. Dostupné z: <http://www.moredevs.ro/opentsdb-and-hbase-rough-performance-test/>

[12] SADALAGE, P. J., FOWLER, M. *NoSQL distilled : a brief guide to the emerging world of polyglot persistence.* Upper Saddle River : Addison-Wesley, 2013. 164 s. ISBN 9780321826626

[13] SEGER, Jan a Richard HINDLS. *Statistické metody v tržním hospodářství*. 1. vyd. Praha: Victoria Publishing, 1995. 435 s. ISBN 80-7187-058-7.

[14] Z., W. *Map-Reduce performance in MongoDb 2.2 and 2.4* [online]. 2013-04-06 [cit. 2013-05-09]. Dostupné z: <http://stackoverflow.com/questions/12678631/map-reduce-performance-in-mongodb-2-2-and-2-4>

[15] 10gen. *MongoDB Manual, Fundamentals* [online]. 2013 [cit. 2013-04-15]. Dostupné z: <http://docs.mongodb.org/manual/faq/fundamentals/>

[16] 10gen. *MongoDB Manual, mapReduce* [online]. 2013 [cit. 2013-04-15]. Dostupné z: <http://docs.mongodb.org/manual/reference/method/db.collection.mapReduce/>

[17] 10gen. *MongoDB Manual, ObjectId* [online]. 2013 [cit. 2013-04-15]. Dostupné z: <http://docs.mongodb.org/manual/reference/object-id/>

[18] 10gen. *MongoDB Manual, Pre-Aggregated Reports* [online]. 2013 [cit. 2013-04-15]. Dostupné z: <http://docs.mongodb.org/manual/use-cases/pre-aggregated-reports/>

[19] 10gen. *MongoDB Manual, Sharded Cluster Overview* [online]. 2013 [cit. 2013-04-15]. Dostupné z: <http://docs.mongodb.org/manual/core/sharded-clusters/>

[20] 2012/148/EÚ: Odporúčanie Komisie z 9. marca 2012 o prípravách na zavádzanie inteligentných meracích systémov

Dodatok A

Zdrojový kód

QueryAPI pre MongoDB

```
git clone https://github.com/ngmon/ngmon-mongo-queryapi.git
```

Záťažové testy a SQL implementácia QueryAPI

```
git clone https://github.com/ngmon/time-series-db.git
```

Dodatok B

Porovnanie Agregačného frameworku a Map Reduce v MongoDB

Predpokladajme, že kolekcia test obsahuje 5 000 dokumentov s indexom na položke date v nasledujúcom formáte :

```
{
        "_id" : ObjectId("5187cf0bfa35f5ee064a9c77"),
        "date" : ISODate("2013-02-01T00:00:00Z"),
        "value" : 10
}
```

Dotaz v Agregačnom frameworku:

```
start = new Date();
db.test.aggregate([
    {$group: {_id: {$month: "$date"}, sum: {$sum: "$value"}}},
    {$sort: {_id: 1}}]);
end = new Date();
print(end.getTime() - start.getTime());
```

Dotaz s rovnakým výsledkom pomocou Map Reduce:

```
start = new Date();
map = function (){
     emit(this.date.getMonth(),this.value);}
reduce = function (id,values){
     return Array.sum(values);}
db.test.mapReduce(map,reduce,{out:{inline:1},sort:{_id:1}});
end = new Date();
print(end.getTime() - start.getTime());
```

Priemerný čas spracovania pomocou Agregačného frameworku 20 ms, pomocou Map Reduce 150 ms.

Dodatok C

Analytický Class diagram


StatisticsAggregationFramework
Attributes
Operations
public StatisticsAggregationFramework( )
Operations Redefined From Statistics
public DBObject count( )
public DBObject avg(String field)
public DBObject sum(String field)
public DBObject max(String field)
public DBObject min(String field)
StatisticsMapReduce
Attributes
Operations
public StatisticsMapReduce( )
Operations Redefined From Statistics
public DBObject count( )
public DBObject avg(String field)
public DBObject sum(String field)
public DBObject max(String field)
public DBObject min(String field)
StatisticsMapReduceCached
Attributes
Operations
public StatisticsMapReduceCached( )
Operations Redefined From Statistics
public DBObject count( )
public DBObject avg(String field)
public DBObject sum(String field)
public DBObject max(String field)
public DBObject min(String field)
<<interface>>
Statistics
Attributes
Operations
public DBObject count( )
public DBObject avg(String field)
public DBObject sum(String field)
public DBObject max(String field)
public DBObject min(String field)
Field
Attributes
package String field
Operations
public Field(Query query, String field)
public Query exists( )
public Query doesNotExist( )
public Query equal(Object value)
public Query notEqual(Object value)
public Query lessThan(Object value)
public Query lessThanOrEq(Object value)
public Query greaterThan(Object value)
public Query greaterThanOrEq(Object value)
public Query hasOneOf(Iterable<> value)
public Query hasAllOf(Iterable<> value)
public Query hasNoneOf(Iterable<> value)
public Query hasThisOne(Object value)
public Query hasThisElement(Object value)
public DBObject sum( )
public DBObject avg( )
public DBObject min( )
public DBObject max( )
Query
Attributes
private DBCollection col
Operations
public Query(DBCollection col)
public DBCollection getCollection( )
public void append(String field, Object value)
public Field field(String field)
public Query orderAsc(String field)
public Query orderDateAsc( )
public Query orderDesc(String field)
public Query orderDateDesc( )
public DBObject getOrder( )
public Query limit(int num)
public int getLimit( )
public void setImplementation(Statistics stat)
public Statistics getImplementation( )
public DBObject getMatchQuery( )
public DBObject getMatchQueryWithSubTime(Date qStart, Date qEnd)
public Query fromDate(Date date)
public Date getFromDate( )
public Query toDate(Date date)
public Date getToDate( )
public Query setStep(long step)
public long getStep( )
public DBObject find( )
public DBObject distinct(String field)
public int countAll( )
public DBObject count( )
public DBObject avg(String field)
public DBObject sum(String field)
public DBObject min(String field)
public DBObject max(String field)
public DBObject median(String field)
public DBObject causeFor(DBObject effect, int limit, int timeBefore)
Manager
Attributes
private Mongo m
Operations
public Manager(String host, int port, String dbName)
public Manager( )
private void connect( )
private void loadProperties( )
public Query createQueryOnCollection(String collectionName)
public Query createQuery( )
public Preaggregate createPreaggregate(String collectionName)
public DB getDb( )
public DBCollection getCollection( )
public Manager setCollection(String collectionName)
public Manager dropCollection(String collectionName)
public void executeJSFromFile(String path)
public void executeJSSaveFromDefaultFile( )
public String readFile(String path)
public void executeJS(String cmd)
public void setMode(Mode mode)
public void setDB(String name)
Event
Attributes
private ObjectId id
private Date date
private double value
private String source
Operations
public ObjectId getId( )
public void setId(ObjectId id)
public String getSource( )
public void setSource(String source)
public Date getDate( )
public void setDate(Date date)
public double getValue( )
public void setValue(double value)

Dodatok D

Upsert procedúra pre aktualizáciu štatistík v PostgreSQL

```
CREATE OR REPLACE FUNCTION upsertclassic(tabled varchar(20),
dated TIMESTAMP,field varchar(10),
valued DOUBLE PRECISION) RETURNS integer AS
$$
DECLARE
sumf varchar(10);
countf varchar(10);
avgf varchar(10);
sql varchar(200);
counter integer;
BEGIN
    sumf:='sum' || field;
    avgf:='avg' || field;
    countf:='count' || field;
    LOOP
     sql := 'UPDATE '||tabled||' SET ('||sumf||', '||countf||',
      '||avgf||') = ('||sumf||'+'||valued||','||countf||'+1,
      ('||sumf||'+'||valued||')/(1+'||countf||')) WHERE date
      ='||quote_nullable(dated);
          EXECUTE sql;
          GET DIAGNOSTICS counter = ROW_COUNT;
       IF counter>0 THEN
           RETURN 1;
       END IF;
       BEGIN
            sql:='INSERT INTO '||tabled||'('||avgf||',
            '||countf||','||sumf||',date) VALUES
            ('||valued||',1, '||valued||','||
            quote_nullable(dated)||')';
```

```
            EXECUTE sql;
            RETURN 1;
            EXCEPTION WHEN unique_violation THEN
            END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;
```