Софийски университет „Св. Климент Охридски”
Факултет по математика и информатика
Катедра „Компютърна информатика”

Дипломна работа

за получаване на образователно-квалификационна степен „магистър”
на Веселин Димитров Николов, фак. № М-22528,
студент от магистърска програма „Информационни системи”

на тема:

NoSQL бази от данни – възможности и приложение

Научен ръководител: доц. д-р Калинка Калоянова

София, 2010 г.

1. Увод

През последните години има сериозен подем в разработката на бази данни, които не поддържат SQL и/или не са релационни. Общият принцип на тези бази данни е, че SQL трябва да бъде жертван, в името на по-добра производителност, отказоустойчивост или възможност за дистрибутиране върху голям брой евтини системи, но не едновременно всяка една от тези цели. Течението се обединява от идеята, че SQL е пречка пред качествената услуга, като се налага терминът NoSQL.

В момента съществуват достатъчно сериозни алтернативи на RDBMS. Всяка NoSQL база от данни е възникнала около определена различна потребност и се разработва, за да я задоволи, в тясна връзка с потребителите си. Въпреки споделянето на общи принципи, отделните NoSQL бази данни са технологично и архитектурно много разнообразни. Познаването им е важно за вземането на правилен избор каква база от данни да се ползва за нов или разрастващ се софтуерен проект.

Мотивация за избор на тази тема

В процеса на работа като разработчик на сайтовете 911.bg и topbloglog.com се сблъсках с проблемите на денормализацията и загубата на консистентност. Изправен пред необходимостта да правя промени в архитектурата на базата от данни, в противоречие на изучаваното в университета, реших да отделя време в търсене на алтернативен надежден метод за съхраняване и достъп до големи обеми от информация, при който изчерпването на системните ресурси няма да доведе до нужда от нови промени в архитектурата.

Цел на разработката

Целта на дипломната работа е да изследва необходимостта от NoSQL бази от данни, практическото им приложение, да опише основните характеристики на тези бази от данни, като прави референции за сравняване с няколко популярни RDBMS – MySQL, MS SQL Server и Oracle, като основно за сравнение се използва MySQL.

Ще бъдат разгледани както общите принципи и различията между отделните NoSQL бази данни, както и специфичните решения и какви предимства носят. Целта е настоящето изследване да бъде полезно не само като дипломна работа, но и за професионалисти, които не са запознати с NoSQL и имат желание да научат повече за технологията и архитектурата на тези бази от данни.

Структура на разработката

Увод

Въвежда в предметната област, описва целта, структурата и целите на дипломната работа.

NoSQL срещу RDBMS

Целта на този раздел е да обясни исторически нуждата от NoSQL бази данни. Върху кратко описание на релационния модел са направени референции към възникващите проблеми и предоставените от NoSQL решения. Разделът въвежда и някои термини като Partition и Sharding, които ще бъдат силно употребявани в останалата част от разработката.

Общи принципи на работа на NoSQL базите от данни

Целта на този раздел е да въведе читателя в принципите на теоремата на Брюър, MapReduce и въпросите на евентуалната консистентност, както и да представи REST и JSON.

Съвременни NoSQL бази от данни

Целта на този раздел е да представи на читателя някои от най-разпространените бази от данни, заедно с техните специфични особености и начин на работа. Избрани бази от данни са разгледани в детайли.

Тестов пример за работа с NoSQL

Целта на този раздел е еднотипни операции да бъдат извършени с различни NoSQL бази от данни върху 2 различни машини – Windows и Ubuntu Linux. Включва тестове за писане, четене, mapReduce в MongoDB и CouchDB, както и примерно свързване на две системи в Replica Set с MongoDB.

Приложения

Към дипломната работа вървят файловете couch6.py, connect.py и threadpool.py, с които са извършени тестовете.

Литература

Наличните материали за темите, касаещи NoSQL са голямо количество, но са неравномерно разпределени. Има голям обем информация за въведение и сравнително малко информация, която покрива проблемите в дълбочина. За някои бази данни, като CouchDB, има 2 издадени книги. За други има единствено помощ в официалните им страници.

Дипломната работа използва за основни източници книгите „CouchDB: The Definitive Guide", „Hadoop: The Definitive Guide", материалите „Amazon Dynamo Paper" и „Bigtable: A Distributed Storage System for Structured Data", както и онлайн документациите на различните бази от данни, особено тази на MongoDB и Riak.

Използвани са множество други книги, презентации, лекции и онлайн статии, предимно на участници в разработката на базите от данни, които касаят. Приложен е пълен списък на цитираните източници.

2. NoSQL срещу RDBMS

NoSQL е отрицание на релационния модел, не толкова на SQL като език. За това е необходимо на кратко да се разгледат RDBMS (Relational Database Management Systems – релационни системи за управление на бази от данни), за да може от техните недостатъци да стане ясно защо има NoSQL.

RDBMS и нормални форми

Основна конструкция в релационния модел е релацията. Релацията е двумерна таблица, в която се помещават данните и се основава на математическото понятие за релация. Атрибутите служат за имена на колони. Името на релацията и множеството на атрибутите й формират схемата на релацията. Множеството от всички схеми на релации в една база от данни е схемата на базата от данни. Ако A и B са множества, релацията е подмножество на произведението от AxB. [63]

Математическата дефиниция за релация е следната: нека r е релация, а $R(A_1, A_2, ..., A_n)$ е нейната схема. Тогава r(R) е математическа релация от степен n върху домейните $dom(A_1)$, $dom(A_{2)}$, ..., $dom(A_n)$, подмножество на декартовото произведение на домейните, които дефинират R. Релацията r(R) е множество от енторки, $r = \{t_1, t_2, ..., t_m\}$. Всяка енторка (n-tuple) е подреден списък от *n* стойности, като всяка отделна стойност v_i принадлежи на домейна $dom(A_i)$ или има специалната стойност NULL. [63]

Схемата на една конкретна база от данни, изградена с релационния модел, се изгражда чрез прилагане на правилата за нормализация. Нормалните форми имат за цел да съхранят данните от аномалии при добавяне, обновяване и прочит на информация.

Съществуват дефиниции за първа нормална форма (Код, 1970 г.), втора и трета нормална форма (Код, 1971 г.), Бойс-Код нормална форма (1974 г.), също и по-висши нормални форми, от които с практическо приложение е шеста нормална форма на К. Дейт.

Първа нормална форма, по деф. на Код гласи: Всеки компонент на всеки кортеж съдържа атомарно значение [63].

К. Дейт 2003 г. дава широко интерпретиране на дефиницията. Данните са в първа нормална форма, ако:
- Колоните не са изрично подредени;
- Редовете не са изрично подредени;
- Няма дублирани редове;
- Всяко сечение между ред и колона съдържа точно една стойност от съответния домейн;
- Към колоните няма скрити стойности, от вида на скрити дати или идентификатори;

Втора нормална форма гласи, че релацията R е във втора нормална форма, ако е изпълнена първа нормална форма и всеки неключов атрибут е в пълна функционална зависимост от ключа, т.е. зависи от целия ключ, а не от някакво подмножество на ключа [63].

Трета нормална форма.
Атрибут, който е част от ключа се нарича първичен атрибут. Релацията R е в трета нормална форма тогава и само тогава, когато за всяка функционална зависимост, или лявата страна е суперключ, или дясната страна е първичен атрибут. Трета нормална форма изисква атрибутите да не са транзитивно зависими от първичния ключ. Обикновено се приема, че базата от данни е нормализирана, ако е в трета нормална форма.

Бойс-Код нормална форма (BCNF). Релацията R е в BCNF тогава и само тогава, когато за всяка нетривиална зависимост $A_1, A_2, ..., A_n$ -> B от R, съответното множество от атрибути $\{A_1, A_2, ..., A_n\}$ е суперключ за R. [63]

Някои предимства на RDBMS:

- Простота, гъвкавост и продуктивност при разработка;
- Данните са в лесно разбираем вид;
- Лесно обучение, неограничен обем документация за съществуващите СУБД (системи за управление на бази от данни);
- SQL е стандартизиран и има голяма степен на припокриване между SQL за различните бази данни;

- SQL дава възможност да се правят всевъзможни справки.

Ограничения на RDBMS

Ограниченията на релационните бази от данни, мотивиращи разработката на NoSQL бази от данни, са по отношение няколко практически проблема – евтина скалируемост, без ангажиране на приложенията с допълнителна логика; нужда от евтина промяна на структурата на данните; нужда от метод за управление на обобщения и свързани данни при голям обем на информацията; отразяване на гъвкави структури от данни.

1. Скалируемост

Има два основни аспекта на скалируемостта – възможност за справяне с нарастващ обем данни и с нараснало натоварване. Най-прозрачното решение е добавянето на хардуер към съществуващите машини и на нови машини в клъстери.

Всички по-известни RDBMS поддържат репликация и клъстеризация. MySQL Cluster дори предлага близка до линейната скалируемост. Въпреки това и MySQL, и SQL Server 2008, и Oracle RAC имат ограничения за броя отделни сървъри, които могат да участват в клъстера и високи системни изисквания за отделните нодове (възли, nodes).

Първият основен проблем, поставен за решаване пред разработчиците на NoSQL бази данни е постигането на масивна скалируемост при ниска цена и при употреба на евтина техника.

При RDBMS има няколко метода за работа, които дават възможност за разрастване, с цената на измяна в логиката на приложенията.

2. Partitioning

Представлява разделяне на една база от данни на части - MySQL дава възможност за разпределяне на базите данни върху множество дискове чрез partitioning по първичен ключ (primary key, PK). Чрез storage engine NDB Cluster, MySQL дава възможност 1 база да не

присъства физически на 1 машина. NDB дава възможност и да се ползват индекси върху диска с размери далеч отвъд наличната памет.

Partitioning-ът при MySQL изисква данните да са разпределени на части по първичен ключ и не работи задоволително, при положение че данните с индекси, различни от PK.

NoSQL DB предлагат богат набор от решения за разпределяне и дублиране на данните между отделните сървъри в рамките на един клъстер, разгледани по-подробно към всяка база от данни по отделно.

3. Sharding (Шардинг)

Шардингът е метод за разделяне на данните от една база от данни (една таблица) на части, разположени на различни устройства на база някакъв принцип, с цел намаляване на латентността. Разделянето може да бъде по дата на постъпване, по user_id % X или друг избран критерий. Отделните части се наричат Shards – ще бъде използвана думата „шардове". Приложенията, ползващи RDBMS, правят sharding чрез логика в приложението, не в базата. В NoSQL шардингът или е автоматизиран, или е ненужен, в зависимост от различния вид база от данни.

Sharding-ът оскъпява разработката на приложения и прави поддържането им много по-трудно и скъпо. Това от своя страна мотивира търсенето на решения, при които шардинг не е необходим или се реализира автоматизирано.

4. Кеширане извън DB сървъра.

DB сървърите предлагат вградено кеширане на заявките. При 1 сървър, Query cache забавя изпълнението на всички заявки, за които няма cache hit (визирам MySQL, където такъв Query cache съществува). При повече от 1 сървър, Query cache е различен спрямо всеки слейв.

В MS SQL Server няма query cache, но има кеширане на последно прочетените блокове с информация (data page), както и на query plans (анализирана и оптимизирана SQL заявка,

преди изпълнението). За разлика от MySQL, тази форма на кеширане не изисква настройка от администратора на база от данни, за постигане на оптимална производителност.

В Oracle аналогът е „Buffer cache" и подлежи на настройване, като съществува механизъм да се анализира до колко заделения обем памет е достатъчен или прекомерен.

Във всички бази данни има и ограничение за обема на кеша и колкото по-голям е, толкова по голямо е забавянето при търсене на cache hit. За това при разработката на натоварени приложения се практикува и кеширане извън DB сървъра на application ниво, чрез употреба:

- Асоциативен масив / речник в паметта (Application object, APC, shared memory, дори файлове)
- Специализиран софтуер – Memcached

Възможно е да се използват един или повече отделни сървъри за целта.

Основен проблем с кеширането е т.нар. студен старт (cold start). Натоварено приложение може да бъде рестартирано, в резултат от което кешът да се изгуби. Докато се натрупат данни в кеша, базата данни ще понася натоварване, което може да не е в състояние да обслужи.

Това от своя страна насърчава употребата на бързи in-memory персистентни бази данни за кеш, т.е. асоциативни масиви, които при рестартиране поддържат състоянието си. Такива са NoSQL базите данни MemcacheDB, Redis.

5. Възможност за адаптиране и измяна.

Twitter посочват като основна причина за дългогодишните им опити да мигрират от MySQL към Cassandra невъзможността да добавят нова колона в съществуваща таблица, предвид обема на данните. Този проблем е общ за RDBMS. Измененията в схемата при 50-100K записа в RDBMS са лесни и бързи. Измененията при ТВ информация са близки до невъзможност и отнемат дни, при много сложни схеми за измяна (потвърдено от Twitter и WordPress.com за MySQL).

Повечето NoSQL бази данни управляват лесно, тъй като не разполагат с фиксирана схема. При липса на схема, добавянето на нова колона в колонна база от данни или нов атрибут в документ за документна база от данни е тривиално и се отнася само до новите данни.

6. Нарушаване на релационния модел като метод за оптимизация на RDBMS

RDBMS са лесни, удобни и интуитивно правилни за работа при малък обем данни, когато е възможно да се поддържат данните в нормализиран вид. При нарастване на натоварването, първата жертва са консистентността и нормализацията. Записват се обобщени данни в полета и се поддържат при обновяване на данните или периодично. Този подход дава възможност за връщане на невярна информация при заявка.

Над определен обем данни става нецелесъобразна, дори невъзможна употребата на JOIN. При MySQL JOIN не може да се ползва, когато данните са разположени физически на няколко различни сървъра. JOIN е скъпа операция още при работа с 10-100 милиона записа в рамките на 1 сървър и употребата в натоварени справки не е разумна. Това води до употреба на денормализирани DB схеми, в които данните от много таблици се съхраняват в една, често многократно повторени в отделните шардове (с нарушена първа нормална форма). Типичен пример за това е Twitter, чиято архитектура с MySQL работи при тежка денормализация на данните (към май 2010). Всяко потребителско съобщение се копира по 1 път за всеки негов получател, като съдържа цялата нужна информация в 1 запис. Този подход увеличава заеманото пространство хиляди пъти над необходимото.

Получаването на обобщена информация, когато данните са съхранени по подобен начин е трудна задача.

NoSQL базите данни са създадени да управляват този тип проблеми.

CouchDB и MongoDB съхраняват данните, като заемат много повече място от MySQL MyISAM. Това е, защото конструктивно тези бази данни са така устроени, че денормализират информацията, в съответствие с потреблението ѝ. Това става чрез методиката MapReduce. Този подход ще бъде подробно обяснен в дипломната работа, както и отделната му реализация в някои от най-популярните бази данни.

Релационните бази данни се разработват от 70-те г., след въвеждането на термина от Едгар Код и предоставят доста усъвършенстван метод за управление и достъп до данни, както и за разработка на натоварени и големи по обем приложения. Липсата на SQL е сериозен недостатък за NoSQL базите данни, като удобство за работа. Излизането от релационния модел обаче не е точно проблем, а по-скоро решение на възникващите в RDBMS проблеми.

3. Общи принципи за работа на NoSQL базите данни

Изграждането на нерелационни бази данни се възражда с нарасналия обем данни за обработване – данни с мащаб Петабайти и появата на теория за работа с такива обеми информация. Важната терминология за работа с NoSQL включва изясняване на теоремата CAP, евентуалната консистентност, MapReduce и др., изяснени в тази глава.

Теоремата CAP (теорема на Брюър)

Теоремата CAP твърди, че в една разпределена система трябва да се избере между две от Консистентност (Consistency (C)), Наличност (Availability (A)) и възможност за разделяне на части (Partition tolerance (A)). Т.е. едновременното осигуряване на трите не е възможно.

Теоремата е представена от Ерик Брюър, 2000 г.

Из “CouchDB: The Definitive Guide” [1]:

> *Консистентност:*
> *Всички клиенти на базата от данни виждат една и съща информация, даже при конкурентно обновяване.*
>
> *Наличност:*
> *Всички клиенти на базата от данни могат да достъпват някоя версия на информацията.*
>
> *Възможност за разделяне:*
> *Базата от данни може да се разделя върху множество сървъри.*
>
> *Изберете две.*

Решението на повечето NoSQL бази данни е да изберат наличност и възможност за разделяне, за сметка на консистентността, като се търси максимално бърза и надеждна синхронизация между отделните сървъри.

Ако трябва да се търси гарантирана консистентност при разпределена база от данни, това означава да се изчака кворум между отделните сървъри и това ще бъде задължителен bottleneck (ботълнек, тясно място) в системата.

Осигуряването на консистентност при единствен сървър обикновено не е проблем при всички бази данни, за това въпросите, касаещи теоремата CAP се отнасят предимно до опериращите върху множество сървъри NoSQL бази данни, освен в частта с обобщаване на информацията.

Amazon Dynamo имплементира Consistent hashing (осигурено от Java)

> *Основната идея на консистентния хеш алгоритъм е да хешира едновременно обектите и кеша чрез една и съща хеш функция. Причината за това е да се асоциира кеша към интервал, който съдържа списък с хешове. Ако кешът бъде премахнат (рестартиране на машина примерно), неговият интервал ще бъде поет от съседен интервал. По този начин се избягва при падане на 1 сървър, да има пиково натоварване на точно 1 друг сървър.*
>
> *[20]: http://weblogs.java.net/blog/2007/11/27/consistent-hashing*

Следователно Amazon Dynamo дава приоритет на наличността пред всичко останало.

CouchDB с неговата master-master инкрементална репликация, обяснена по-долу, дава също приоритет на наличността.

Cassandra, подобно на CouchDB, има за приоритет наличността и възможността за разделяне на части. Възможно е да се избира между време на достъп и консистентност, като с тази настройка може да се избере максимална стойност за консистентност. Cassandra не осигурява възможност за row lock (заключване на запис).

[21] http://wiki.apache.org/cassandra/ArchitectureOverview

MongoDB ползва auto-sharding и replication cluster, което по същество означава един мастър сървър за всеки шард. Това поставя MongoDB в лагера на релационните бази данни по отношение на теоремата CAP – с приоритет на консистентността. MongoDB жертва устойчивостта на данните (Durability) – няма гаранция, при употребата на един сървър, че системата ще работи надеждно.

[22] http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1

Hbase, подобно на MongoDB дава приоритет на Консистентността и Partition tolerance.

Доказателството на теоремата има на адрес [23]:
http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

ACID и BASE

ACID означава Atomicity, Consistency, Isolation, Durability. Съвременните RDBMS следват тези 4 правила практически без изключения.

Atomicity – означава, че транзакциите в база от данни трябва да следват правилото всичко или нищо. Ако една част от транзакцията се провали, цялата транзакция се проваля.

Consistency – консистентност. Означава, че всяка транзакция променя базата от данни от едно състояние с консистентна информация в друго такова състояние.

Isolation – изолация. Това е изискване, че не може да се достъпва информация, която е в незавършила операция. Това правило в съвременните бази данни се поддържа с известно заобикаляне, чрез няколко различни вида ниво на изолация и се допускат компромиси в името на намаляване броя dead locks (READ UNCOMMITED isolation level).

BASE означава Basically Available, Soft-state, Eventual consistency.

BASE е противоположност на ACID. BASE приема, че в края на операцията ще има консистентност, без изрично да я осигурява, в името на по-добра производителност, скалируемост и наличност (Basically Available).

Soft State по отношение състоянието на данните означава, че:

- Състоянието се управлява с вероятности
- Състоянието не е непременно консистентно
- Потребителят поема отговорност за управление на състоянието на данните
- Състоянието може да бъде прекратено от мрежата / сървъра, но може да бъде прекратено и от потребителя
- Състоянието не е критично за услугата

Оригиналната презентация, с която Ерик Брюър представя CAP теоремата си и BASE е достъпна на адрес: [10]
http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

Soft-state подробна дефиниция: [25]
http://mercury.lcs.mit.edu/~jnc//tech/hard_soft.html

На евентуалната консистентност е посветен следващия раздел.

Евентуална консистентност

Този термин се описва и въвежда в практиката от Amazon Dynamo paper (2007), въпреки че може да се срещне и в разработки от преди това. Означава, че ако няма обновяване, евентуално всички нодове (възли, инстанции) на базата от данни ще връщат актуалната и консистентна стойност на данните.

Google BigTable и Amazon Dynamo

Google BigTable е фокусирана върху консистентност и наличност, при нея не може да се говори за евентуална консистентност. В BigTable репликацията се осъществява от файловата система (този подход ще бъде разгледан в раздела за Hadoop).

Dynamo е система, която поставя като приоритетни Наличност и Разделяне на части от теоремата CAP. Тя е изключително налична, дори когато отделните части на данните са разделени върху мрежови partition-и и е евентуално консистентна. Данните се репликират в рамките на единичен клъстер. Въпреки това, версията на данна върху един възел може да се различава от тази в съседен. Може да се гарантира, че евентуално всяко изменение ще бъде получено от всеки клиент.

Според дефиницията на Amazon, публикувана в блога на Вернер Вогелс, консистентността бива 2 вида, според гледната точка – от гледна точка на клиента и от гледна точка на сървъра.

От гледна точка на клиента има storage system (разпределена система за съхраняване на данните) и няколко независими процеса, които пишат и четат от системата.

От гледна точка на клиента

Клиентската консистентност трябва да се справи с това как отделните процеси, комуникиращи със системата, виждат обновяванията. Ако процес А направи обновяване, имаме следните видове консистентност:

- Силна консистентност. След приключване на обновяването, всеки един клиент ще вижда обновената стойност.

- Слаба консистентност. Системата не гарантира, че последващите клиенти ще получат обновената стойност, освен ако не са изпълнени някакви условия. Периодът до изпълняване на условията е период на неконсистентност.

- Евентуална консистентност. Това е форма на слаба консистентност, при която системата гарантира, че ако няма направени нови обновявания на данната, евентуално всички точки на достъп ще върнат вярната стойност.

Евентуалната консистентност от своя страна бива:

- Обикновена консистентност – ако процес А информира с процес Б, че е обновил данна, следваща заявка към процес Б ще върне обновената стойност и следващ запис ще е с по-висок приоритет от по-ранния.

- Read-your-write consistency – процес А, след като обнови стойност на данна, никога няма да види стара версия на стойността.

- Session consistency – форма на предишната консистентност, при която имаме гаранция за консистентност в рамките на една сесия.

- Monotonic read consistency – ако процес А види дадена стойност, никога няма да види предишна версия на тази стойност на данната.

- Monotonic write consistency – гарантира, че записът на данната ще е приключил, преди следващ запис от същия процес със същите данни да бъде извършен.

От сървърска гледна точка

Настройваемост на консистентността (Динамо и Касандра) – ако W + R > N, има консистентност.
N – брой реплики на базата от данни
R – брой реплики, които трябва да потвърдят стойността
W – броя реплики, които ще бъдат блокирани за запис

Как да се конфигурират тези параметри зависи от задчата. Ако R=1 и N=W, имаме оптимизиране за четене. Ако W=1 и R=N, оптимизацията е за бързо писане. Ако W < (N+1)/2, има възможност за конфликт при писане. Евентуалната консистентност възниква, когато W+R <= N, което означава, че има възможност записаните и прочетени данни да не съвпадат.

> *Обяснено в презентацията на Stu Hood, Rackspace за Касандра, но се отнася и за Динамо [26]:*
> *http://thestrangeloop.com/sites/default/files/slides/StuHood_Cassandra.pdf*
> *Вернер Вогелс – за Динамо [62]:*
> *http://www.allthingsdistributed.com/2008/12/eventually_consistent.html*
> *Източник за терминологията [27]:*
> *http://www.cis.upenn.edu/~lee/07cis505/Lec/lec-ch7b-replication-v3.pdf*

Евентуална консистентност в CouchDB

Ако имаме 2 или повече сървъри, информацията трябва постоянно да се уточнява чрез постоянна комуникация между сървърите. CouchDB постига евентуална консистентност между сървърите чрез incremental replication (последователна репликация). Това е процес, при който измененията на документите се копират между сървърите периодично (всъщност инстанцията може да бъде CouchDB инстанция на преносим компютър без достъп до Интернет, ползвана с тестова цел).

Сървър 1 праща промените н сървър 2, той на 3 и т.н. Когато се получи конфликт между две ревизии на документ, печели по-новата от двете. [1]

Евентуална консистентност в MongoDB

MongoDB дефинира няколко вида евентуална консистентност:

- Single Writer Eventual Consistency – евентуална консистентност с един източник на промени. При 1 мастър и множество слейвове, това е ситуация, в която клиентът може да прочете остаряла информация или да получи изменения в неправилен ред.
- Monotonic Read Consistency – евентуална консистентност, в която не може да се получат изменения в неправилен ред.
- Read-your-own-writes Consistency – евентуална консистентност, при която клиентът достъпва собствените си обновявания. В уеб приложение това може да означава потребителя, а не клиента. При употребата на load balancer (приложение за разпределяне на натоварването) лесно може да бъде нарушена.

  Източник е официалния блог на MongoDB [28]:
  http://blog.mongodb.org/post/498145601/on-distributed-consistency-part-2-some-eventual

В зависимост от това какъв тип компромис с консистентността може да бъде допуснат в приложението, трябва да се осигури конфигурацията на MongoDB.

Евентуална консистентност в Cassandra

В Касандра консистентността подлежи на настройка в момента на заявка. Можем да подадем ниво на консистентност при подаване на заявка за получаване на данни:

```
get(keyspace, key, [column_family, column_name], consistency_level)
```

consistency_level може да бъде
ONE – връща стойността в първия достъпен нод
QUORUM – 50%+1 нода трябва да потвърдят стойността
ALL – изчаква всички нодове да върнат в стойност.

В случая със запис на данни, опциите са:
ZERO – без блокирания
ONE – изчакване на 1 нод да потвърди записа
QUORUM – изчакване на 50%+1 нода

ALL – блокиране на всички.

JSON & BSON

JavaScript Object Notation или JSON е удобен текстов формат за обмяна на информация. JSON е базиран на масивите в JavaScript, но е изцяло езиково независим. Чете се лесно от разработчиците и се парсва лесно от различните езици за програмиране. JSON се използва често за сериализация и пренасяне на структурирана информация, която функция изпълнява с много по-малък овърхед (код, участващ във формата) от XML.

Първоначално асинхронното предаване на информация между клиент и сървър в Web (AJAX – Asynchronous JavaScript and XML) се подразбира, че се е извършвало чрез XML, но поради сложността и различния начин на работа с XML в различните браузъри, JSON измества XML и в последствие се разпространява извън Web.

В контекста на NoSQL, JSON е най-популярния формат за комуникация с NoSQL базите данни, въпреки че някои NoSQL бази поддържат и други. JavaScript е основен език за заявки към NoSQL базите данни, но се използва значително по-рядко от JSON като формат за приемане и връщане на нформация (HBase).
BSON е бинарен JSON и се използва от някои бази данни за съхраняване на информация (CouchDB).

Пример, подобен на дадения в JavaScript: The Definitive Guide, 5-то издание, раздел 20.1.9 [6]

```
<!-- XML – наличие на затварящи тагове и 5-6 символа кодиране на специалните
символи като & в &amp; -->
<author>
  <name>David Flanagan</name>
  <books>
    <book>JavaScript &amp; The Definitive Guide</book>
    <book>JavaScript for noobs</book>
  </books>
</author>

// JSON
{
  "name": "David Flanagan",
  "books": [
    "JavaScript & The Definitive Guide ",
```

```
    "Javascript for noobs"
  ]
}
```

Допълнително удобство дава C++ стила за ескейпване на символни низове. JSON поддържа числа, стрингове, булеви променливи, null, масиви и обекти.

REST

REST е съкращение на Representional State Transfer и представлява модел на софтуерна архитектура за разпределени системи като Web, при който при комуникацията между клиентите и сървърите се използват представяния на ресурсите. В контекста на Web, REST услуга е такава уеб услуга, при която HTTP методите, които се ползват, отразяват реалните операции, изисквани от клиента към сървъра, а URL адресите отразяват самия ресурс.

HTTP е богат на смислени методи. REST използва GET, PUT, POST, DELETE.

GET се използва за извличане на информация, без промяна на състоянието на информацията.
POST и PUT се използват за обновяване и добавяне на информация (кое от двете, варира в зависимост от стила и системата).
DELETE се използва за изтриване на информация.

REST означава също така избор на система за адресиране, което съответства на ресурсите на сървъра. В NoSQL базите, ползващи REST интерфейси, това означава в общия случай адрес от вида, като адресите варират и са разгледани допълнително за всяка база от данни:
http://host:port/DB_NAME/TABLE?some_filter=1

Наличието на такова API означава, че обичайно почти всички Database операции – включително създаване на бази данни, иницииране на репликация и други, са достъпни през HTTP. От друга страна, посредством употребата на REST API, могат да бъдат изградени уеб програми, които не използват междинен слой (като PHP). Клиентът може да бъде JavaScript+HTML, сървърът – примерно CouchDB или Riak.

NoSQL бази данни, поддържащи REST интерфейси (REST API) са CouchDB, Riak, Hbase и др.

Map / Reduce

Това е метод за извличане на информация. Включва 2 функции, които се прилагат спрямо списък от данни – map и reduce. Map функцията връща обработен резултат от 0, 1 или повече записа за всеки един запис в списъка, докато reduce обобщава списъка в 1 стойност. Map/reduce може да работи разпределено, в много процеси, едновременно върху много машини, като разбива информацията на парченца, обобщава ги и отново ги обобщава.

Map / Reduce е метод за извеждане на изгледи и обработена, обобщена информация, на периодичен принцип, чрез еднократно писане и многократно четене в последствие. Поддържа се от почти всички NoSQL бази данни (без някои key/value pairs бази).

Типично map и reduce функциите са на JavaScript, но някои бази данни поддържат множество езици. Независимо от езика, на който са написани, map функциите съдържат в себе си emit – с нея към резултата се добавя нов запис. Причината да не се ползва return е, че за 1 входящ запис може да имаме няколко изходящи.

Примерна map функция в CouchDB:

```
function(doc) {
  if (doc.comments && doc.comments.length) {
    emit(doc._id, doc.comments.length);
  }
}
```

Примерна reduce функция в CouchDB:

```
function(key, values, rereduce) {
    return sum(values);
}
```

И документните, и колонните бази данни използват map/reduce. Все пак, някои key/value pairs, като Redis, не предлагат тази възможност.

Употреба на мрежови файлови системи

NoSQL базите данни имат изключително разнообразни подходи по отношение на съхраняване на данните. Някои бази данни, като Google BigTable ползват за разпределяне специализирани файлови системи, други вървят в пълен софтуерен пакет – като HBase + Hadoop – база от данни и отделен storage engine. При трети файловата система не е от значение, тъй като за разпределянето на данните между нодовете отговаря самата NoSQL база от данни (MongoDB, CouchDB, Cassandra, Riak и т.н.)

Употреба на мрежови файлови системи в RDBMS

MySQL

MySQL работи надеждно върху NFS (network file system) и това е проверен метод за съхраняване на бинарни логове с цел бързо възстановяване на системата при авария. Все пак правилният начин за клъстъризация е чрез употребата на NDB Cluster.

> *Тестове в тази посока са проведени от MySQL performance blog:*
> *http://www.mysqlperformanceblog.com/2010/07/30/storing-mysql-binary-logs-on-nfs-volume/ [29]*

NDB Cluster е изграден от множество хостове. Те съдържат NDB Management Server, множество MySQL сървъри, както и NDB data nodes на отделни сървъри. Всеки data node съхранява копие (replica) на определен partition.

> *Подробна информация има в "High Performance MySQL"*
> *http://oreilly.com/catalog/9780596003067 [4]*
> *Също и на официалния сайт на MySQL:*
> *http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster-overview.html [30]*

MS SQL Server

MS SQL Server има вградена опция за клъстеризация, както и съвместна работа между множество SQL Server-и които са свързани един с друг. Файловите системи на отделните сървъри са видими, което е осигурено от Windows.

Допълнителна информация: SQL Server 2005 – наръчник на администратора, гл. 12 и 13. [5]

Oracle

Oracle разполага със клъстерна файлова система (Oracle Cluster File System, OCFS), която дава възможност за обща файлова система в рамките на всички сървъри от един клъстер. OCFS дава възможност на администраторите да работят с физически файлове на базата от данни върху целия клъстер, с цел улесняване на администрацията.

[31] http://oss.oracle.com/projects/ocfs/

Употреба на мрежови файлови системи от NoSQL базите данни

Google Bigtable и GFS

Google File System (GFS) е скалируема, разпределена файлова система, предназначена за големи обеми разпределена информация. Осигурява отказоустойчивост върху евтин хардуер и осигурява възможност за сумиране на производителността на голямо количество сървъри.

GFS е оптимизиран за работа с голямо количество (милиони) файлове с информация, всеки от които е с голям размер (100MB+). Изграден е по начин да улеснява типичните операции за работа с данни – както четенето на големи обеми от последователна информация, така и четенето на случайна информация и записването и разширяване на файловете. Особено внимание е отделено на надеждността и отказоустойчивостта.

1 GFS клъстер разполага с 1 мастър сървър и множество chunk servers. Мастър сървъра съхранява мета информацията, а chunk сървърите съхраняват самата информация, като за

всеки chunk има записан 64 битов идентификатор (chunk handle) и 64MB размер. Мета информацията се съхранява в паметта.

Промените в мета информацията се съхраняват в лог, подобен на RDBMS транзакшън логовете, с който може да се прави възстановяване до точка от времето (point in time recovery).

GFS поддържа консистентност на информацията при конкурентен запис на един chunk, като прилага измененията в еднакъв ред за всяка реплика и като използва версия на всеки chunk, за да открие евентуални неактуални реплики, в резултат на период, в който chunk server не е работил. Теоретично е възможно подобна неактуална реплика да се използва, тъй като има кеширане на прочетената информация, но възможността за връщане на некоректна информация е логически ограничена и обичайно се връща грешка, вместо остарели данни.

Отказоустойчивостта се постига чрез:

- Бързо възстановяване при грешка;
- Репликация на всеки chunk върху множество chunk servers;
- Репликация на мастъра.

Google BigTable се възползва максимално от възможностите на GFS и не осигурява вътрешно функции като репликация. GFS и BigTable са разработвани едновременно. Конструктивно разделянето на таблицата на таблети от ~200MB в BigTable е направено, с цел оптимална употреба на GFS.

Източници и допълнителна информация за GFS:
http://labs.google.com/papers/gfs.html [32]
и
http://static.googleusercontent.com/external_content/untrusted_dlcp/labs.google.com/bg//papers/gfs-sosp2003.pdf [33]
За работата на BigTable върху GFS: [34]
http://static.googleusercontent.com/external_content/untrusted_dlcp/labs.google.com/bg//papers/bigtable-osdi06.pdf

Hadoop Distributed File System - HDFS

HDFS е файлова система, проектирана за съхраняване на много големи файлове с възможност да бъдат подавани поточно (в стрийм). Файловата система е предназначена да работи върху клъстери от евтин хардуер, като осигурява наличност на данните без загуби, дори когато част от сървърите не са налични. HDFS, подобно на GFS, е оптимизирана за append (добавяне) към файловете. Редакцията на файлове не се поддържа.

> *Особено голям файл за HDFS – от стотици MB до PB.*

Размерът на единичен блок в HDFS е многократно по-голям от този в обичайна файлова система и равен на този в GFS – минимум 64MB, опционално 128 или повече. Това е с цел да се намали времето на търсене на блок (seek time).

В един клъстер има два типа нодове, според шаблона master-worker – namenode (мастър) и datanode (worker).

Namenode управлява пространството от имена, дървото на файловата система и мета информацията за всички файлове. Данните са съхранени под формата на файлове – с текущото състояние и с лог на измененията. Данните за това къде се намира всеки блок по отделно не се съхраняват постоянно, а се изграждат от datanodes всеки път, когато системата се стартира. Namenode сървъра е критичен за работата на HDFS, поради което в конфигурацията е нужно да се осигури отказоустойчивост – Hadoop може да се конфигурира така, че всяка операция на Namenode да се съхранява едновременно в няколко различни файлови системи. Опционално може да се стартира вторичен Namenode сървър, чиято функция е да бъде резервен.

> *Под евтин хардуер все още се има предвид употреба на RAID5. Типичен Datanode сървър е 2и с 6 евтини SATA диска, според „Pro Hadoop”, Джейсън Венер (7).*

За не-Java достъп до HDFS се използва Thrift service (също като Cassandra, в раздел Cassandra има пояснения за Thrift).

HDFS поддържа компресия на файловете, като може да се избира опционално между DEFLATE, gzip, ZIP, bzip2, и LZO. За големи файлове са подходящи единствено ZIP и bzip2, тъй като поддържат разделяне на части.

Важно за HDFS е, че при заявка се калкулира най-подходящия datanode, на база дистанцията до съответната реплика на търсения блок, като дистанцията се измерва в bandwidth (ширина на мрежовия канал) между отделните datanode сървъри. Този проблем възниква и при писане, при определяне на това къде да бъдат разположени репликите на всеки блок, като за писане се използва специфичен balancer, чиято цел е да разпредели данните равномерно из datanode сървърите.

Важно е да се знае, че след създаване на файл, неговото съдържание може да не е непременно достъпно веднага след създаването му, дори след flush операция на потока (stream flush). Това е жертва в името на по-добра производителност.

HDFS има сериозно сходство с GFS, но е достъпна за употреба и е с отворен код.

> *Допълнителна информация: "Hadoop: The Definitive Guide", основен източник за този раздел [2]. Също Hadoop user guide [35]:*
> *http://hadoop.apache.org/common/docs/r0.18.3/hdfs_user_guide.html*

4. Съвременни NoSQL бази данни

NoSQL базите данни са съвременна технология, която е в етап на доказване. Флагманите са Google BigTable и Amazon Dynamo, които при все че са със затворен код, дават доказателство, че този метод на работа работи. Те въвеждат и част от терминологията и са важни, до колкото повлияват много от останалите бази данни. От Dynamo и BigTable се заражда Cassandra, която свързваме с Facebook. От архитектурата на Google с GFS + BigTable тръгва Hadoop, а от архитектурата на Dynamo - Riak. Съхранението на данни ключ/стойност еволюира от една страна до документния модел и базите данни CouchDB, MongoDB и по-късно RavenDB, а от друга до кеш-ориентираните бази като Redis и в по-малка степен граф базата Neo4j. Избрах също да представя решението на Yahoo – Sherpa, свързаните с Hadoop решения – Pig, Hive и няколко други съществени технологии.

Класификация и обхват на дипломната работа

Според http://nosql-databases.org и http://en.wikipedia.org/wiki/NoSQL NoSQL базите данни се класифицират грубо на документни, граф, key-value store (с няколко под вида), таблични, обектни и други, като общо различните бази от данни са десетки. Това наложи за целите на дипломната работа да бъдат избрани някои характерни представители и да бъдат игнорирани други, като подборът е субективен, на база основно случаи на употреба. Предпочетени са базите данни, използвани в популярни услуги, както и тези с повече налична информация в Интернет.

Списък с базите от данни по видове, като с получерен шрифт са отбелязани онези, които са разгледани в дипломната работа:

Документни бази от данни:
CouchDB, **MongoDB**, Apache Jackrabbit, **RavenDB,** Terrastore и др.

Граф-бази данни:
AllegroGraph, **Neo4j**, InfoGrid, DEX, FlockDB, InfiniteGraph и др.

Key/value с персистентност:
BigTable, **MemcacheDB**, **Redis**, SimpleDB, **Kyoto Cabinet,** Tokyo Cabinet и др.

Key/value cache в паметта:
Memcached, Velocity, **Redis** и др.

Евентуално консистентни key/value pairs:
Dynamo, **Cassandra**, Project Voldemort, **Riak** и др.

Подредени key/value pairs:
Memcachedb, Berkeley DB.

Обектни:
Db4o, GemStone/S, Objectivity/DB, **ZODB** и др.

Таблични:
BigTable, Mnesia, **Hbase**, Hypertable и др.

Други:
Sherpa, FluidDB и др.

В този списък са подбрани по-съществените.

Въпреки наличието на тази и други класификации, трябва да се отбележи, че всяка от базите от данни може да бъде поставена в 4-5 от групите. Евентуално консистентни key/value pairs са и документните бази от данни, а практически всички NoSQL бази от данни са подредени key/value pairs, без “key-value cache”.

По-добра класификация може да се направи според приложението – на пълнофункционални бази от данни и специализирани, като специализираните биват бази данни за кеширане (Redis, Memcachedb, Kyoto Cabinet и др.), за графи и други.

Според поддръжката на шардинг и репликация – такива, които поддържат и такива, които не; според езика, на който са написани (Java, Erlang, C++, C, C#); според наличието на REST API и според поддръжката на собствен shell (MongoDB, HBase Pig Grunt, Cassandra); според поддръжката на MapReduce; според необходимостта от специална мрежова файлова система (HBase, BigTable срещу останалите); дори според това, дали са с отворен код или не (Dynamo, BigTable, Sherpa, Lotus Notes не са с отворен код, повечето от останалите са).

Google BigTable

2004 г. Google представя BigTable – система за управление на подредена информация, чиито дизайн поддържа и управлява надеждно и устойчиво петабайти информация, върху хиляди нодове.

Google внедрява BigTable в много от услугите си – Google Analytics, Google Reader, Blogspot, Google Earth и т.н.

BigTable не поддържа релационния модел изцяло. Вместо него поддържа по-прост модел, в който има динамичен контрол на данните и структурата им. BigTable третира всички данни като неинтерпретирани стрингове, съхранени бинарно, в които може да има, а може и да няма сериализирани обекти. BigTable дава възможност за контрол, дали информацията да се връща от диска или от паметта на сървъра.

Моделът на BigTable представлява дистрибутиран, подреден многомерен масив. Масивът е индексиран по ред, колона и дата.

Ключовете по редове са с размер до 64 KB, типично между 10 и 100 байта. Всяко четене и запис на информация е независимо. Данните са подредени по азбучен ред на ключовете. По същия критерий, с цел load balancing се извършва автоматизиран partitioning на данните.

Google BigTable съхранява данните под формата на **Column families.** Семейството от колони е група от колони, обединени от някакъв общ принцип. Данните за 1 ред от едно семейство колони се съхраняват заедно.

Датата, под форма на 64 битов timestamp, с точност до микросекунди, е определяща за версията на дадена данна. Различните версии се съхраняват едновременно. Това поле може да се използва за разрешаване на конфликти, като timestamp полето е достъпно за запис от приложението.

Достъпът до данните е през API. Интерфейсът дава функции за създаване и изтриване на таблици и семейства колони, за промяна на клъстър, таблици, мета информация за семейства колони и промяна на права за достъп. Клиентските програми могат да записват или изтриват стойности в BigTable, да извличат стойност за отделен ред или да обходят списък с информацията в таблица.

В Интернет е достъпен REST-подобен тестов интерфейс, базиран на HBase Thrift API. Въпреки че не е достъпна за download, BigTable може да бъде пробвана онлайн на адрес, като за идентификация се използват google потребителско име и парола:

http://bigtable.appspot.com/

Достъпни са Python и Java библиотеки.

BigTable поддържа транзакция върху единичен ред – прочит – промяна – запис, но не и върху по-комплексни групи данни.

BigTable ползва Google File System – GFS. Информацията е в SSTable формат. SSTable е конструирана, така че всеки блок да се достъпва с 1 seek, а според конфигурация може да бъде достъпна в паметта.

BigTable ползва 2 нива на кеширане – Block cache и Scan cache. Block cache кешира прочетен 64 кб-тов блок от данни, който е бил прочетен – с цел да се улесни достъпа до близки данни. Scan cache улеснява достъпа до данни, които се четат многократно.

Данните се компресират чрез бързи алгоритми (BMDiff и Zippy).

[18] http://blogoscoped.com/archive/2005-10-23-n61.html

Google BigTable е разработен на C++.

Източник и допълнителна информация (официалния документ за Google BigTable):

http://static.googleusercontent.com/external_content/untrusted_dlcp/labs.google.com/bg//papers/bigtable-osdi06.pdf [9]

Amazon Dynamo

Амазон Динамо е вътрешна технология на Амазон, разработена за да задоволи нуждата от лесно скалируема, високо налична key-value база от данни. Технологията е с такава архитектура, че да даде възможност на потребителите си да избират между цена, консистентност, устойчивост и производителност, като гарантира висока наличност.

Динамо предлага прости интерфейси за put (запис) и get (извличане) на информация. Всеки put изисква ключ, контекст и обект. Контекстът е базиран на обекта е и се използва от Динамо да валидира обновяванията. Всеки обект се съхранява като бинарен blob. Интерфейсът не предлага възможност за работа с групи от обекти.

Сървърите на Динамо са организирани в ринг. За изваждане на хардуер от ринга могат да се използват логически нодове. Цялостната архитектура на инсталация на Динамо може да се види на адрес: http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html

Амазон Динамо е вдъхновен от начина, по който се осигурява баланс в природата – всеки път, когато част от веригата отпадне, натоварването автоматично се преразпределя. Динамо извършва се логическо разпределяне на данните по нодовете, като логиката е комплексна. Извършва се автоматично преразпределяне, когато ще се вади устройство от ринга. Всеки обект се съхранява едновременно на множество нодове, но не непременно на всички. Обновяването на информацията става асинхронно. Консистентността се поддържа след извършване на Put/Get.

Всеки нод може да се използва за изискване или запис на всеки ключ.

Динамо е разработено на Java.

> *Подобно на Google BigTable, Amazon са публикували доста подробен идеологически документ:*
> *http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf*

Динамо на Амазон въвежда терминологията за евентуална консистентност: "the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value." – базата от данни гарантира, че ако няма нови обновявания към обекта, евентуално всички точки на достъп ще връщат актуалната стойност.

Cassandra

Касандра е разпределена система за управление на бази данни с отворен код, разработвана под егидата на Apache Software Foundation. Предназначена е за управление на особено големи обеми информация, пръснати върху множество сървъри, при осигуряване на висока наличност, без единична точка на отказ.

Динамо и BigTable не се използват вън от техните създатели. Тяхната техническа документация обаче е публично достъпна и вдъхновява серия от бази данни. Един от бившите разработчици на Amazon Dynamo започва разработката на Cassandra, на база на идеологията и опита на BigTable и Dynamo.

Разработката на Cassandra е инициирана от Facebook и публикувана с отворен код в Google code през 2008 г. До преминаването на проекта към Apache Software Foundation януари 2009 г., няма външни commit-ери (хора с право да променят кода) и развитието и документирането е затруднено. Текущата версия (към август 2010) е 0.6.4, но вече се използва масирано във Facebook и някои други големи сайтове.

Основни принципи на работа на Cassandra са: DHT (distributed hash table), евентуална консистентност, и настройваема латентност срещу консистентност (по отношение теоремата

CAP). Ключовете са разпределени върху много нодове, които образуват ринг (подобно на Динамо).

> *Distributed hash tables (DHTs) – представлява разпределена услуга, в която се съхраняват ключове и стойности, като ключовете са под формата на хеш. Всеки участващ възел в системата може да получи всяка стойност по даден ключ. Връзката между ключ и стойност е разпределена между възлите, независимо колко точно участват в даден момент.*

Касандра е колонна база от данни, донякъде идеологически сходна с BigTable, но въвежда ново измерение на масива си, след колона и семейство от колони.

Колона. Суперколона. Семейство колони. Суперсемейство от суперколони.

Колоната е най-малката единица на натрупване на информация в Cassandra. Съдържа име, стойност и дата (timestamp). Представя се с JSON, но се съхранява като byte[]. UTF-8 се сериализира.

Примерна колона:

```
{
 "name": "emailAddress",
 "value": "dzver@abv.bg"
 "timestamp": "1234567890"
}
```

Суперколона (supercolumn) е двойка (tuple) с име и стойност, която съдържа няколко колони. Тя е първата голяма структурна разлика между Cassandra и Amazon Dynamo & Google BigTable.

```
{
    name: "homeAddress",
    value: {
        street: {name: "street", value: "Mladost", timestamp: 123456789},
```

```
        city: {name: "city", value: "Sofia", timestamp: 123456789},
        postalcode: {name: "postalcode", value: "1000", timestamp: 123456789},
    }
}
```

Семейство от колони - всеки ред в ColumnFamily съдържа набор от колони. Може да се направи аналог с асоциативен масив (съкратени са timestamp стойностите в примера):

```
{
   "Ivan Ivanov":{
      "Users":{
         "emailAddress":{"name":"emailAddress", "value":"foo@bar.com"},
         "webSite":{"name":"webSite", "value":"http://bar.com"}
      },
      "Stats":{
         "visits":{"name":"visits", "value":"243"}
      }
   },
   "Petar Petrov":{
      "Users":{
         "emailAddress":{"name":"emailAddress", "value":"user2@bar.com"},
         "twitter":{"name":"twitter", "value":"user2"}
      }
   }
}
```

[36] http://arin.me/blog/wtf-is-a-supercolumn-cassandra-data-model

Семейство колони може да съдържа супер колони също. Ако обектът „статия” може да се опише със семейство от колони, коментарът към статия има нужда от 1 допълнително измерение – ключа на отделната статия, към която се отнася коментара. Това представлява по същество суперсемейството от суперколони.

Всеки отделен ред няма дефинирана схема, т.е. може да съдържа разнообразни колони.

Keyspace е първото измерение на Касандра хешове и съдържа семействата колони. Keyspace са Конфигурациите на семействата колони и дават структурата, на база на която се извършва добавянето на информация (insert).

Реален пример с архитектура от digg.com Arin Sarcissian дава в https://nosqleast.com/2009/slides/sarkissian-cassandra.pdf [19]

```
Friend_Diggs { // Column Family – гласовете на приятелите
      12345 : { // story_id as Row key
            user_id: { // SuperColumns are User's IDs
                  friend_id1: true,
                  friend_id2: true,
            }
      }
}
```

Keyspace, семействата колони и семействата суперколони се дефинират в конфигурационния файл storage-conf.xml на Cassandra и не могат да се променят, докато програмата работи. Заедно с тях се дефинират настройките за кеширане, типа на ключа, подреждането на отделните записи. Липсата на възможност за промяна на точно тази конфигурация не е сериозен проблем, предвид липсата на фиксирана схема за самите колони. Аналог в RDBMS би било дефиниране на таблиците в RDBMS в конфигурационен файл и невъзможност да се изменят в runtime режим (към Cassandra 0.6.4).

В Cassandra 0.7 ще съществува възможност да се промени схемата чрез миграция.

Обикновено има само един Keyspace в рамките на 1 сървър.

Ограничения:

1. Една колона трябва да се побере физически на 1 машина.
2. Една стойност (value) не може да бъде по-голяма от 2^31-1 байта (2GB).
3. Една двойка ключ-стойност трябва да се побере в RAM паметта, за да се обработи.

Интерфейс за достъп и начин на работа

Касандра ползва Thrift и има API на 12 различни езика.

> *Thrift е RPC framework за разработка на услуги между различни езици за мащабна скалируемост. Поддържа сериализация при по-нисък овърхед спрямо SOAP, тъй като е бинарна. Поддържа езици като Java, C++, Python, Ruby.*

Записите на информация в Касандра са изключително бързи. Това е постигнато чрез следния алгоритъм:

- Първа стъпка на запис е той да влезе в commit log.
- След това се записва в MemTable (съхранено в паметта състояние).
- След това се праща към диска и се записва в SSTable (аналогично на BigTable)

Няма read before write, sequential. Няма seeks. Може да се пише върху всеки нод, данните са винаги writeable.

Последователност от действия при четене на информация:
- Първо се проверява Memtable за value, след което и SSTable. Извършват се повече операции от writes, за това са по-бавни.
- Read repair е термин, при осъществяване на quorum read. В зависимост от използваното ниво консистентност се връщат данни на база определен кворум или от първия нод, но във фонов режим данните на всички нодове се сравняват и изравняват.

Cassandra разполага с конзолен шел за достъп до данните. В Linux обичайно това е командата:

```
/usr/bin/cassandra-cli –host localhost –port 9160
```

В Windows аналогът е cassandra-cli.bat

Базови команди

Зписване на стойност:

```
cassandra> set Blog.Posts['post1']['title'] = 'Hello World'
```

```
Value inserted.
```

Прочитане на стойност:

```
cassandra> get Blog.Posts['post1']
=> (column=title, value=Hello World, timestamp=1283360210609000)
```

Преброяване на броя колони в 1 ключ:

```
cassandra> count Blog.Posts['post1']
1 columns
```

За употребата на Cassandra е налична много малко документация към момента. Налични са примери за получаване на списък от стойности, които изискват предварително всеки ключ да е записан в списък. Това означава, че не съществува аналог на SELECT * FROM TABLE.

Индекси в Cassandra

Създаването на индекс представлява създаване на ново семейство колони, в което ключът е полето, по което желаем да поставим индекс и това трябва да се управлява от приложната логика. За всяка връзка между семейства трябва ново семейство, описващо връзката.

[24] http://www.royans.net/arch/cassandra-inverted-index/

Измежду останалите разгледани NoSQL БД това е най-трудното за реализация решение.

MapReduce

MapReduce се поддържа единствено при работа с Hadoop и чрез употреба на вградения в Hadoop, вж. раздел Hadoop. Поддръжка за това е добавена във версия 0.6:

[37] http://architects.dzone.com/news/cassandra-adds-hadoop

Репликация и синхронизация

Касандра разполага с gossip протокол за синхронизация на измененията. Това, което зависи от администратора на базата от данни е да определи броя копия на всяка данна и броя възли, които трябва да върнат успех при запис и четене (вж. в раздел консистентност).

Обяснение за gossip protocol, [38]: http://en.wikipedia.org/wiki/Gossip_protocol

[39] http://www.slideshare.net/benjaminblack/introduction-to-cassandra-replication-and-consistency

Има два подхода за разпределяне на данните между отделните инстанции. Това са методите RandomPartitioner (RP) и OrderPreservingPartitioner (OPP). С OPP могат да се правят нодове по диапазон, но разпределянето на натоварването може да стане неравномерно. Обратно – с RP се постига равномерно натоварване, но не може да се изградят отделни нодове по диапазон.

[40] http://ria101.wordpress.com/2010/02/22/cassandra-randompartitioner-vs-orderpreservingpartitioner/

Разпространение

1. Digg използва Cassandra, като времето за цялостна миграция е било 5 месеца.
2. Twitter – разполагат с копие, работещо на Cassandra, но към момента не е известно да го използват.
3. Facebook – използва Cassandra за някои функции. Налични са данни има към 2008 г. - 100TB, 160 nodes, 500M записа на ден.
4. Rackspace ползват Cassandra за инфраструктурата си.
5. Reddit е разработен с Cassandra.

CouchDB

Apache CouchDB е безплатна документна база от данни с отворен код, разработена на Erlang. CouchDB е повлияна от Lotus Notes и е в страни от разгледаните до тук бази данни, въпреки някои сходства. Данните се съхраняват без фиксирана схема. Вместо това се съхраняват в документи и е осигурен HTTP интерфейс за достъп до тях, чрез REST API и JavaScript – като език за създаване на изгледи.

CouchDB означава Cluster Of Unreliable Commodity Hardware - предназначен да работи върху множество евтини сървъри, отдалечени един от друг географски, дори несвързани с Интернет.

CouchDB е разработена на Erlang - fault tolerant език и работи върху Unix, като с тестова цел може да се ползва и Windows инсталация (в тестовия раздел Windows инсталацията се справи задоволително).

> *Fault Tolerance - Никоя комбинация от проблеми, освен пълен срив на мрежата, не е допустимо да причини некоректна работа на системата.*
>
> *Дефиниция на Сет Гилберт и Нанси Линч [41]:*
> *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.6951&rep=rep1&type=pdf*

CouchDB има за цел да отрази реалното състояние на данните - под формата на документи, не на таблици. В CouchDB една база данни е серия от документи, съдържащи множество полета - ключ и няколко стойности. Всеки документ е независим от останалите, няма схема, но документът носи в себе си информация от кой тип (doc_type) е.

Предварителното обмисляне на схемата е много важно в релационния модел. ALTER на големи таблици е бавно и неприятно. В документните бази това не е точно нужно да се прави. Като резултат, CouchDB не може да се осигури консистентност на данните спрямо външен ключ (foreign key, FK). Ако добавим ново поле в CouchDB серия от сходни документи, то ще съществува само за онези, в които полето е записано. Не е необходимо да се съхраняват NULL стойности за останалите документи.

Привеждането на данните във вид, удобен за различни видове употреба, става във фонов режим, чрез мощни средства за това – MapReduce и материални изгледи (view). Този подход е създаден с мисъл за решаване на големи проблеми и задачи.

Основен недостатък на документния подход за съхраняване на данни е, че се налага се ненужно повтаряне на информация между документите.

Всеки документ в CouchDB има уникално ID (уникалността е в рамките на една база от данни) и съдържа полета с имена. Стойностите на данните в документите могат да бъдат стрингове, числа, дати, булеви, списъци и др., без ограничения за броя. Освен данните, дсеки документ съдържа мета-информация, примерно ревизията на документа. Ревизиите са важни, защото помагат за разрешаването на конфликти при едновременна редакция от 2 потребителя. Ревизиите не са налични продължително време - изтриват се с db compact-ване от db администратора.

Бихме могли да използваме уникалните идентификатори на CouchDB и управляемо. Има REST API и за това:

curl -X GET http://127.0.0.1:5984/_uuids?count=5

Ще върне 5 уникални идентификатора за употреба:

```
{"uuids":["163271278f438b6ccb9c87879df9d226","163271278f438b6ccb9c87879df9d25e",
"163271278f438b6ccb9c87879df9e03d","163271278f438b6ccb9c87879df9ed59","163271278
f438b6ccb9c87879df9f605"]}
```

Примерен документ в CouchDB

Следващият документ е изграден с визуалното средство за създаване на документи Futon.

Futon е средство за управление на CouchDB, което е достатъчно удобно за разработка, но слабо използваемо в реален клъстър от CouchDB сървъри.

Документът съдържа автоматично генерирани полета _id и _rev за ревизия, като предишните ревизии са достъпни също за определено време.

```
{
   "_id": "163271278f438b6ccb9c87879d0001c7",
   "_rev": "3-29830772003eb7f6e49f55834d28c640",
   "url": "http://dzver.com/blog/?p=2000",
   "author": "Eric Brewer",
   "title": "The CAP Theorem",
   "comments": [
       {
           "author": "Veselin Nikolov",
           "published": "2010-08-17 15:00:00",
           "body": "Cool Dude!"
```

```
        },
        {
            "author": "Mark Twain",
            "published": "2010-08-17 15:01:00",
            "body": "Nice Theorem"
        }
    ]
}
```

_rev полето е изключително важно. При обновяване на документ, ревизията от _rev трябва да бъде подадена. След обновяването, _rev придобива нова стойност. Ако се опитаме повторно да обновим стара версия, ще се получи конфликт и CouchDB ще върне грешка.

ACID в CouchDB. MVCC

CouchDB спазва принципите Атомарност, Консистентност, Изолация, Устойчивост на релационните бази (ACID) при изпълнението на транзакции и осигурява устойчивост на данните в рамките на единичен сървър.

CouchDB не подържа locks (заключване на данните). Вместо това разполага с Multi-Version Concurrency Control (MVCC) за управление на конкурентния достъп до данните. Основната идея е версията на всеки документ да бъде подадена в момента на обновяване и след обновяване, ревизията на активния документ да се промени. Старата ревизия не може повече да се обновява и може да изчезне след известно време.

RESTFUL API

Данните и практически всички функции на CouchDB са достъпни през уеб услуги през HTTP, като се използват HTTP методи, съответстващи на извършваните операции върху данните. Самите данни се предават във формат JSON - JavaScript Object Notation. Този метод на работа дава възможност на всеки език и платформа, поддържащ HTTP, да работи с CouchDB.

GET се използва за извличане на информация, PUT или POST за обновяване и добавяне на данни, DELETE - за изтриване. PUT се предпочита пред POST.

Примерна употреба през linux shell с curl (GET заявка, JSON отговор):

```
dzver@localhost:~$ curl http://127.0.0.1:5984
{"couchdb":"Welcome","version":"1.0.0"}
```

Създаване на база от данни foobar:

```
curl -X PUT http://127.0.0.1:5984/foobar
{"ok":true}
```

Добавяне на данни:

```
curl -X PUT http://127.0.0.1:5984/contacts/johndoe -d '{}'
{"ok":true,"id":"johndoe","rev":"1-795358690"}
```

Дори при създаването им през уеб интерфейса за администрация, заявките представляват HTTP requests с параметри, подадени като част от адреса или подаден JSON. В обратната посока CouchDB може да връща и други формати, освен JSON – XML, HTML и други, посредством show функции и шаблони.

CouchDB ядро

Ядрото на CouchDB е B-tree storage engine. Б-дървото е структура от подредена информация, която позволява търсене, добавяне и изтриване. CouchDB използва B-tree за цялата си вътрешна информация, както и за всички документи и изгледи.

Рестрикцията за достъп до документите е по ключ или диапазон на ключа. Целта на ограничението е да се постигне максимална производителност. Идентични ограничения има в множество от NoSQL базите данни – пример BigTable, Hadoop, SimpleDB, memcache.

Map, Reduce и изгледи в CouchDB

CouchDB предлага Spidermonkey базиран JavaScript engine за достъп до данните. Дава възможност да се създават изгледи, които извършват обобщения и JOIN-ове. Те се генерират при поискване, подобно на RDBMS изгледите. Базирани са на map/reduce. Могат да бъдат съхранени и временни, като временните са по-бавни за употреба.

При всяка заявка от клиент, CouchDB проверява, дали съответното view (изглед) е актуално. Ако не е, извършва нужните промени и връща информацията.

CouchDB прилага map и reduce при промяна на документ само върху онези документи, за които е необходимо, за да калкулира новите стойности (използваният в литературата термин е incremental computing). Прилага ги в изолация. Възможността да изолира тези операции дава възможност изчисленията на изгледите да става паралелно и инкрементално.

Съвкупността от map и reduce функция произвежда изглед. В CouchDB изгледите са индексирани и подредени по ключа, който се подава с emit в map функцията.

Всички map функции в CouchDB имат един входящ параметър – doc, който представлява единичен документ в базата, но могат да върнат повече от един документ като резултат, чрез употреба на функцията emit(key, value).

Изгледите се съхраняват като „Design documents” – това е специален документ, чието id започва с „_design/”. Те съдържат map и reduce функции под формата на символен низ в JSON тялото-то на документа. Резултатът е нов изглед, който се попълва при първоначалното изпълнение и допълва при всяко обновяване на документ, касаещ изгледа. Резултатът от това изпълнение се съхранява под формата на B-tree, в отделен файл. За по-бърза работа може да се ползва отделен диск за едно или множество view-та.

Примерен изглед:

```
{
   "_id": "_design/test",
   "_rev": "1-230141dfa7e07c3dbfef0789bf11773a",
   "views": {
       "foo": {
```

```
            "map": "function(doc){ emit(doc.price, doc)}"
        }
    }
}
```

За да направим филтриране, трябва ключа на emit да е полето, по което ще филтрираме. Заявката ще бъде:

```
GET /db/_design/test/_view/by_date?startkey=100&endkey=200&descending=true
```

– за всички документи с цена между 100 и 200, в обратен ред спрямо цената.

Reduce функциите работят с подредени редове, върнати от emit на map функция в изглед. Reduce използва фундаменталните свойства на B-дърветата – за всеки краен елемент (подреден ред) има последователност от възли, достигащи до корена. Всеки краен елемент (leaf node) в B-tree съхранява няколко десетки реда, в зависимост от размера, и всеки вътрешен възел може да е свързан с няколко крайни елемента или с други вътрешни възли. Reduce функцията се изпълнява към всеки възел в дървото, за да калкулира окончателната стойност. Крайният резултат е reduce функция, която може да се обновява последователно при изменения от map функцията, като преизчисляването на reduce се прави за минимален брой възли. Единствено първоначално редукцията се изчислява за всеки възел.

Базова reduce функция:

```
function(keys, values, rereduce) {
      return sum(values);
}
```

Когато работи върху крайни възли, съдържащи реални редове от изгледа, reduce функцията има трети параметър „rereduce = false”. Аргументите в този случай keys и values са резултатите от map функцията. Функцията връща 1 стойност, която се съхранява за последваща reduce (rereduce) калкулация, съответно тогава параметърът е “rereduce = true”. Ако дълбочината на B-дървото е по-голяма, rereduce се изпълнява неколкократно, докато се достигне корена.

Show функции

CouchDB поддържа show функции, които представляват JavaScript с вход – документ и изход – HTML или нещо друго, например конкатениран символен низ. Могат да се използват за форматирани справки директно от CouchDB във HTML, XML или друг формат.

```
GET /db/_design/test2/_show/someshow/1234
```

Ще извика someshow javascript функцията, дефинирана в дизайн документа _design/test2, приложена спрямо документ с _id = 1234. Като допълнение, CouchDB поддържа HTML шаблони (templates).

[42] http://books.couchdb.org/relax/design-documents/shows

CouchDB репликация и sharding

CouchDB репликация от един хост към друг може да се инициира с GET заявка, указваща източника и целта. Като получи такава заявка, двете бази се сравняват и се трансферират само измененията и изтриванията. В заявката може да се укаже тя да е постоянна – тогава CouchDB ще следи за изменения и ще извършва синхронизацията автоматично.

CouchDB Lounge е прокси-базирана програма за разделяне на части на базите данни и клъстеризация, разработена за Meebo. Lounge съдържа два основни компонента – един за обработка на GET и PUT заявки за документи, и друг за разпределяне на заявките към изгледи.

Първият модул е dumbproxy – представлява модул за nginx сървър. Върши работа да осигури сигурност, кодиране, разпределяне на натоварването, компресиране и кеширане на ресурсите.

Вторият модул е smartproxy – разпределя заявките към изгледи върху възли от клъстера.

Тези два прокси модула трябва да бъдат дублирани неколкократно, за да се гарантира отказоустойчивост.

Шардингът се пави чрез consistent hashing – използва фрагмент от идентификаторите на документите _id, за да ги разпределя разпределя по отделните шардове. За да се улесни скалируемостта, CouchDB дава възможност да се прави oversharding – разпределяне на по-малки порции от нужното, като множество шардове са на един и същи сървър, с оглед това, че преместването им е по-малък проблем от разделянето на данните.

MongoDB

Думата Mongo идва от huge + monstrous, името е намек за функцията – да борави с чудовищно-огромни бази данни.

MongoDB е документна база, поддържаща JSON, динамични заявки, индекси, репликация, автоматичен шардинг, map/reduce, профилиране за оптимизация на заявките, по подобие на SQL. Разработена е на C++.

Текущо е по-разпространена от CouchDB поради няколко важни причини. Работи бързо и надеждно върху 1 или няколко сървъра (устойчивост на данните се постига при повече от един сървър). В някои случаи MongoDB се справя по-добре върху единичен сървър, отколкото MySQL. Функционира в стабилна версия, за разлика от повечето от конкурентните бази данни. Работата с индекси е идентична с познатата от RDBMS. За обобщаване се използва MapReduce, но не е необходимо да се пише map функция за получаване на резултатен списък, както в CouchDB.

Времето за инсталация и реализация на примерите е няколко минути. Базата дани е пълнофункционална.

> *http://www.slideshare.net/mongosf/going-from-zero-to-mongo-in-about-2-days-ryan-angilly - Ryan Angilly дава някои от причините, [43].*

Осъществяването за връзка с MongoDB става чрез connection на определен mongodb порт, за разлика от CouchDB, с нейния RESTful интерфейс. Това предполага, че за употреба на MongoDB при разработка с някакъв език за програмиране са необходими driver-и или библиотеки за достъп.

Библиотеки за MongoDB съществуват на разпространените езици за програмиране, като са добре развити за Python, Ruby, PHP, Java.

MongoDB разполага със конзолен shell и дава възможност за работа с JavaScript, като след всяка изписана команда директно се вижда резултата. Тази среда е много подобна на ruby irb shell-а или python shell, с разлика в езика и това, че данните се съхраняват трайно. Предимствата на този shell е, че намалява драстично времето за усвояване на базата от данни, тъй като много бързо може да бъде видян резултата от създаденото, без познаване на базата от данни в дълбочина.

Езикът за работа е JavaScript, но JavaScript с особености. Съдържа някои специфични функции. Съответно JSON, който се ползва е Mongo Extended JSON и поддържа конвертиране до JSON по спецификация.

MongoDB документ. ObjectID() и DBRef()

Документът в MongoDB е споменатият разширен JSON, почти идентичен с показаното в CouchDB. Съществените разлики са няколко.

Начинът на представяне на уникалния идентификатор _id е чрез ObjectID, което е специален тип. В бинарния BSON, ObjectID е 12 байтова стойност, съдържаща 4 байта за дата, 3 байта за идентификатор на сървъра, 2 байта за идентификатор на процеса и 3 байта за брояч. Други специални типове са Date() и DBRef() за дата и указател към данна.

MongoDB позволява в документ да има референция DBRef() към друг документ, както и колекция от документи да се съдържат в определен документ. Ако имаме статии и коментари, колекцията на статиите може да съдържа в себе си както списъци с указатели към коментари, така и самите коментари. В случая с коментарите, препоръчано от MongoDB е коментарите да са част от документа на отделната статия, предвид употребата и това, че референциите работят по-бавно, ако колекцията н е в паметта.

За отношения много-към-много се използват единствено референции.

Пример от http://www.mongodb.org/display/DOCS/Database+References [44]:

```
x = { name : 'Oracle' } //примерът е за Mongo shell
db.courses.save(x) //x придобива _id = ObjectId("4b0552b0f0da7d1eb6f126a1")
student = { name : 'Veselin', classes : [ new DBRef('courses', x._id) ] } //JSON
с референция
db.students.save(student)
```

И е създаден обект student с име ‘Veselin’ в колекцията students, в който има DBRef към курс ‘Oracle’.

Индекси

MongoDB поддържа индекси по полета, различни от ключа. За тях важат сходни правила с тези в RDBMS – трябва да има индекс по полетата, които се ползват за сортиране и филтриране.

[45] http://www.mongodb.org/display/DOCS/Indexes

Индекс се гарантира, че съществува и ако не съществува – се създава с ensureIndex():

```
db.things.ensureIndex({somekey:1});
```

Индекс може да се постави дори, ако документът е йерархичен:

```
db.things.ensureIndex({subdocument.somekey:1});
```

Поставянето на индекс не спестява нуждата от писане на map и reduce функции, но спестява необходимостта от създаване на изглед за всяка една справка. Обичайният синтаксис за справка е:

```
db.things.find({somekey: somevalue}); //търсене с употреба на индекс
```

MapReduce

Map/reduce са JavaScript функции, изпълнявани на сървъра, чрез команда към базата от данни. Базата от данни създава временна колекция за да съхранява резултата от операцията. Колекцията се изчиства със затварянето на клиента или когато бъде изрично изтрита.

Възможно е да се укаже име за трайно запазване на резултата (има пример за това в практическата част на дипломната работа).

Map функцията използва променливата this, за да достъпне текущия обект (в CouchDB документът се подава като входяща променлива). Тя връща резултата чрез emit(key,value) един или повече пъти.

Reduce функцията приема входящи параметри ключ и масив от стойности и връща стойност с return. MapReduce в MongoDB изпълнява reduce функциите итеративно.

Съществува и finalize функция, която може да бъде изпълнена след приключване работата на reduce. Finalize не е задължителна за употреба. Тя връща окончателна стойност.

Пример 1, за работа с конзолния shell mongo. Примерът е изпълнен под Windows XP, с mongodb 1.6.1. За 4 статии с ID от 1 до 4 са дефинирани тагове, които са JavaScript масиви и MapReduce ги преброява и обобщава:

```
MongoDB shell version: 1.6.1
connecting to: test
> db.diplomna.insert( { _id:1, tags: ['bar', 'baz'] } );
> db.diplomna.insert( { _id:2, tags: ['foo', 'bar'] } );
> db.diplomna.insert( { _id:3, tags: ['foo', 'baz', 'node', 'code'] } );
> db.diplomna.insert( { _id:4, tags: ['pipi', 'ripi', 'code', 'test'] } );
> m = function() {
...     this.tags.forEach(
...             function(z) {
...                     emit( z, {count:1} );
...             }
...     );
... };
function () {
    this.tags.forEach(function (z) {emit(z, {count:1});});
}
> r = function( key, values ) {
...     var total = 0;
...     for (var i=0; i<values.length; i++) total += values[i].count;
```

```
...     return {count: total};
... };
function (key, values) {
    var total = 0;
    for (var i = 0; i < values.length; i++) {
        total += values[i].count;
    }
    return {count:total};
}
> res = db.diplomna.mapReduce(m, r);
{
        "result" : "tmp.mr.mapreduce_1282132609_3",
        "timeMillis" : 0,
        "counts" : {
                "input" : 4,
                "emit" : 12,
                "output" : 8
        },
        "ok" : 1,
}
> db[res.result].find()
{ "_id" : "bar", "value" : { "count" : 2 } }
{ "_id" : "baz", "value" : { "count" : 2 } }
{ "_id" : "code", "value" : { "count" : 2 } }
{ "_id" : "foo", "value" : { "count" : 2 } }
{ "_id" : "node", "value" : { "count" : 1 } }
{ "_id" : "pipi", "value" : { "count" : 1 } }
{ "_id" : "ripi", "value" : { "count" : 1 } }
{ "_id" : "test", "value" : { "count" : 1 } }
```

Към момента MapReduce задачите се изпълняват в 1 нишка от всеки mongod процес, поради ограничение в текущия JavaScript engine. Ако MongoDB работи в клъстер, за всеки сървър има по 1 нишка.

[52] http://www.mongodb.org/display/DOCS/MapReduce

MongoDB - репликация и sharding

В MongoDB репликацията се използва за осигуряване на постоянна наличност и шардинг – за хоризонтална скалируемост.

При запис на информация, информацията може да се приеме за commit-ната, когато е записана на повече от половината от отделните реплики. Ако данната е записана в master сървъра, може да бъде видима и преди този истински commit да е настъпил.

MongoDB използва термина “replica sets” за група от сървъри с еднаква информация на тях. Един от тях е мастър. Ако мастъра падне, се излъчва нов мастър, автоматично. След това при възстановяване на падналия сървър, той става слейв.

[47] http://www.slideshare.net/mongodb/replica-sets

За хоризонтална скалируемост MongoDB използва автоматичен шардинг (от версия 1.6, налична от август 2010). Шардът може да работи с replica sets, както и с единични сървъри. Всяка база с MongoDB може лесно да се разбие върху хиляди сървъри, като лесно могат да се добавят нови сървъри. Load balancing-ът (разпределяне на натоварването – включва и разпределянето на данни, и на заявките в последствие) е автоматичен. При изменения в натрупването на данни, преразпределянето също е автоматично. Шардингът на Mongo има 0 единични точки на отказ.

За да работи се изискват сървъри, които работят като шардове, поне един сървър за конфигурация и разпределящ mongos процес.

[48] http://www.mongodb.org/display/DOCS/Configuring+Sharding

Riak

Riak е разпределена key-value база от данни, повлияна от Динамо. Поддържа висока степен на наличност и настройваема консистентност. Riak работи като набор от добре свързани физически хостове. Всеки хост в клъстера има една инстанция на Riak. Всяка инстанция има

набор от виртуални нода или „vnodes”, като всеки един е отговорен за съхраняване на част от key space (пространство от ключове, обяснено в раздела с Cassandra).

Отделните нодове не са копия един на друг, нито участват в обработката на всяка заявка. Делът, до който датата е репликирана, стратегията за обединяване и моделът за възстановяване при грешка се конфигурира.

Клиентският интерфейс на Riak говори за „buckets” и “keys”. Riak изчислява 160 битов бинарен хеш за двойка bucket/key и свързва тази стойност към позиция в пръстена (the ring) от нодове. Пръстенът е разделен на partitions (части). Всеки виртуален нод (vnode) отговаря за отделен partition (част).

> *Bucket е контейнер и keyspace за информацията в Riak, с набор от общи свойства за съдържанието си (броя копия, например). Bucket се достъпват като първ част на URL йерархията след /riak. Т.е. при n_val = 5, броя реплики на дадения data object ще бъде 5.*
> ***[Error! Reference source not found.]***
> *https://wiki.basho.com/display/RIAK/REST+API#RESTAPI-BucketOperation*

За съхраняване на информацията на диска се използва специално разработен storage – Bitsack. До скоро Riak е работил с InnoDB (на MySQL) чрез Innostore връзка на Erlang с InnoDB.

Riak имплементира consistent hashing за дистрибуция на данните между отделните нодове, следвайки идеологията на Динамо. Това поддържа клъстъризацията проста – добавянето на нов нод в клъстъра е изпълнение на едноредова команда.

> *https://wiki.basho.com/display/RIAK/Basic+Cluster+Setup#BasicClusterSetup-AddaSecondNodetoYourCluster [49]*

Riak е разработен на Erlang и C.

Интерфейс за достъп

Riak разполага с REST API, подобно на CouchDB, но и с Erlang API.

Специфичното за REST API, спрямо разгледаните по-подробно REST API е, че е възможно подаването на изискване за кворум, като параметър на заявката и че е задължително подаването на HTTP request-header “Riak-ClientId”, идентифициращ клиента.

Пълно описание на REST API има на адрес:
https://wiki.basho.com/display/RIAK/REST+API

Riak разполага с Gossip протокол за синхронизация, подобно на Cassandra, чрез който отделните нодове информират останалите за настъпили при тях изменения, както и за това, че съществуват.

Vector clocks

За синхронизация на това кой нод с коя версия данна разполага, се използват Vector clocks. Vector clocks представлява информация към всяка данна за актьорите, които са я променяли (X-ClientId от предния раздел) и номера версия, които са видели. Всеки път, когато актьор види информация и я измени, той добавя себе си и текущия vector clock и го променя. По този начин към различните актьори може да съществуват данни с различна стойност и различен vector clock, но може да бъде направена максимално добра синхронизация. Допустимо е да се създават siblings – едновременно съществуващи актуални версии на една данна.

[50] http://blog.basho.com/2010/01/29/why-vector-clocks-are-easy/
Очаква се подобен метод за синхронизация да бъде въведен в Cassandra.

Riak MapReduce

В map функцията в Riak се подават двойка от bucket и key. За всяка двойка, Riak ще изпрати функцията към partition-а, който отговаря за съхраняването на информацията. Vnode ще я изпълни и ще върне стойността.

Reduce фазата приема списък с информация като аргумент и връща като аргумент друг списък от информация, който трябва да е съвместим с първия, тъй като reduce може да бъде

изпълнен многократно. Ако списъкът е различен, това трябва да е обработено в reduce функцията.

Sherpa

Sherpa е key/value store, разработен от Yahoo. Sherpa е с RESTful API, работи с JSON. Създаден е за хоризонтална скалируемост. Притежава собствен модел за консистентност на информацията - клиентът винаги трябва да може да получи информация, която е консистентна, но е допустимо да е остаряла. Консистентността и наличността на базата от данни могат да се настройват - едното, за сметка на другото.

Sherpa работи с таблици.

Примерен GET request:

```
$ curl http://sherpa.yahoo.com/SherpaWebService/V1/get/AddressTable/yahoo
{"sherpa":{
     "status":{"code":200,"message":"OK"},
     "metadata":{"seq_id":"abc-4990E052:abc-5",
                 "modtime":1234231551},
     "fields": {
                     "addr":{"value":"700 First Ave"},
                     "city":{"value":"Sunnyvale"},
                     "state":{"value":"CA"}
     }
}}
```

REST API на Sherpa дава възможност прости SQL заявки да бъдат лесно заменени с GET заявки [17]:

```
SELECT (*|col1,col2,…) FROM table WHERE pkcol = pkval; -- става
$curl http://…/V1/get/table/pkval[field=col1&field=col2]
```

Подреждането на таблиците е по PK и те са разпределени в шардове по PK, като базовата единица е подреден фрагмент от таблица, наречен таблет. Sherpa има автоматизиран шардинг, вграден в приложението, така че не е нужно приложния слой да се грижи за това. В Sherpa sharding и partitioning се върши от Load balancer приложението YAK.

Load Balancer (лоуд балансър) е услуга или сървър, който разпределя натоварването.

Notification streams - Sherpa разполага с поток от информация за информиране за промени. Работи на принципа publisher-subscriber. Дава възможност на приложенията, ползващи Sherpa, да се абонират за събития в една таблица и по този начин да управляват своя кеш и външни индекси. [15], [16], [17]

Hadoop

Какво е Hadoop

Hadoop не е NoSQL база от данни, а колекция от софтуерни пакети, които съвместно могат да се използват за изграждане на такава. Тези проекти са под шапката на Apache Software Foundation. Най-известните под проекти са MapReduce, файловата система HDFS и базата от данни HBase.

Разработката на Hadoop започва от Nutch и GFS (Google File System). Nutch е уеб-търсачка с отворен код. След публикуването на документи от Google за архитектурата на GFS и MapReduce, разработчиците вземат решение да променят подхода в разработка на Nutch. С подкрепата на Yahoo, частта на Nutch, която е за дистрибутиране на изчислителна мощ между различни сървъри се отделя в самостоятелен проект.

Проблемът, който таргетира Hadoop е следния. Голям обем информация, примерно 1TB може да бъде съхранен на един хард диск (HDD - hard disk drive), но четенето и записването на 1TB от 1HDD би отнело часове. Това време може да се ускори, ако едновременно парченца от информация се четат от 100 HDD, вместо от 1. Ако информацията е разпространена върху 100 HDD, е необходимо по специфичен начин да се управляват отказите на дискове (повреди).

Така пред Hadoop стоят 2 задачи:
- Разпределяне и употреба на информация върху множество дискове. Разпределянето става чрез файловата система на Hadoop - HDFS, Hadoop Distributed File System, а употребата - чрез MapReduce.

- Управление на повредите по начин, различен от RAID.

Предимства на Hadoop пред RDBMS

MapReduce разрешава проблема със Seek time - времето за откриване на конкретното място на диска, върху което да се пише или от което да се чете информация. MapReduce ползва Sort/Merge, вместо B-Tree, което дава предимство при големи изменения върху базата от данни.

Таблица 1. Сравнение на RDBMS и Hadoop MapReduce [2]

	RDBMS	MapReduce
Данни	GB	PB
Достъп	Интерактивен и неинтерактивен	Неинтерактивен
Обновяване	Многократно четене и запис	Еднократен запис, многократно четене
Структура	Статична схема	Динамична схема
Data Integrity *	Висока степен	Ниска степен
Скалируемост	Нелинейна	Линейна

Data Integrity - Цялостност и логическа свързаност на данните. В RDBMS се управлява от Primary keys и Foreign keys + data types.

Друга разлика между MapReduce и RDBMS е количеството структура в данните.

Структурираната информация е онази, която е организирана в полета с дефиниран формат, като примерно XML документи или таблици от база от данни, които имат предварително дефинирана схема.

Полу-структурираната информация дори да има схема, тя може да не се спазва - пример, таблица, в която всяка клетка може да има произволен тип данни.

Неструктурираната информация няма никаква вътрешна структура - примерно текстов файл. MapReduce работи и върху полуструктурирана информация.

Компоненти на Hadoop

Core (Ядро) – набор от компоненти и интерфейси за разпределена файлова система, сериализация, Java RPC, persistent data structures.

> *Persistent data structutre е структура, която съхранява в себе си своите предишни версии.*

Avro – система за сериализация и съхраняване на данни.
MapReduce – модел за разпределена обработка на информация върху множество сървъри.

HDFS – разпределена файлова система, която работи върху клъстери. Ще бъде разгледан самостоятелно в раздела за файлови системи.

Pig – SQL подобен език за конструиране на MapReduce заявки върху големи обеми от данни. Изисква Mapreduce и HDFS.

HBase – разпределена колонна база от данни, работеща върху HDFS и Mapreduce, предмет на следващия раздел.

ZooKeeper – услуга за координиране – дистрибутирано заключване (distributed locks).

Hive – осигурява SQL подобен език за достъп до данните върху HDFS, data warehousing.

Chukwa – разпределена система за анализ и обработка на данни върху HDFS и MapReduce. [2]

MapReduce

Може да се реализират map и reduce функции с всички езици, които поддържат работа със стандартен вход и изход. Въпреки че Hadoop е реализиран на Java, работата с map и reduce е

удобна на ruby, python и други езици, поддържащи работа със stdin, както и на C++, чрез Hadoop pipes.

Пример на Python, модифициран от Hadoop: The Definitive Guide p. 35 (2):

```
#!/usr/bin/env python
import re, sys

for line in sys.stdin: #за всеки ред в стандартния вход
      val = line.strip() #като игнорираме интервалите в края
      (year, temp) = (val[15:19], val[87:92]) #година и температура са символите
от 15 до 19-ти и от 87 до 92-и в stdin.
```

С усложняване на задачата, логиката на работа е да се добавят нови map и reduce функции, а не да се усложнява логиката на съществуващите. В Hadoop обичайният начин за това е чрез използване на някой от езиците за създаване на map и reduce, а именно pig и hive.

Map и reduce функциите се изпълняват периодично в задачи (jobs), като предвид възможното бавно действие, има метод за получаване на текущо състояние за това каква част от информацията е обработена.

PIG

Pig е комбинация от среда за изпълнение и език за обработка на данните, които дават възможност да се работи удобно с Hadoop MapReduce. Езикът Pig се нарича Pig Latin. Pig разполага с шел, наречен Grunt, в който в реално време могат да се прилагат Pig команди (както примерно MongoDB Shell-a, python, irb).

Примерна работа с Pig в Grunt, из Hadooop: The Definitive Guide [2]:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int); #зарежда данните в списък
grunt> filtered_records = FILTER records BY temperature != 9999 AND
(quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
#филтрира списъка
grunt> grouped_records = GROUP filtered_records BY year;
```

```
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature); #групира списъка
grunt> DUMP max_temp; #отпечатва резултата на екрана на шела
```

Pig Latin работи с Pig Bags, което е колекция от енторки (tuples). Поддържа SQL подобни операции и функции между Bags с Tuples – JOIN, GROUP, UNION, ORDER BY, MIN, MAX и т.н., по различен начин:

foo = ORDER foo BY bar DESC;

Собствените функции в PIG се пишат на Java, след което са достъпни като User defined functions (UDF).

```
REGISTER /src/myfunc.jar
A = LOAD 'students';
B = FOREACH A GENERATE myfunc.MyEvalFunc($0);
```

Паралелното изпълнение на reduce функциите е изключително лесно:

`grouped_records = GROUP records BY year PARALLEL 30;` -- осигурява изпълнение в 30 паралелни процеса, а това зависи от наличните сървъри и слотове за reducer-и в отделните сървъри.

Пълна референция към синтаксиса на Pig е достъпна на адрес:
http://hadoop.apache.org/pig/docs/r0.5.0/piglatin_reference.html

HIVE

Hive е data warehousing средство, изградено върху Hadoop. Дава механизъм за изграждане на структурирани файлове, както и за правенето на заявки към такива файлове, които включват големи обеми от данни. Hive разполага със собствен език, HiveQL, който е много близък като синтаксис до SQL. HiveQL дава възможност да се ползват MapReduce UDF, написани на Java.

Заявките, разработени с Hive не са бързи. Hive не е средство за осигуряване на натоварен достъп в реално време до голям обем от данни. Обичайно се изпълняват за минути.

Hive разполага със собствен hive shell.

Пример за работа с hive shell има наличен на: [12]
http://www.cloudera.com/videos/hive_tutorial

Примерни заявки за създаване на таблица с Hive, зареждане на информацията от файл и след това селектиране:

```
CREATE TABLE students (fno INT, name STRING, age INT) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' STORED AS TEXTFILE;

LOAD DATA  INPATH '/my/students' INTO TABLE students;
```

Трябва да се отбележи, че въпреки големите прилики в синтаксиса за създаване на таблица и този за SELECT, INSERT е в контекста на директории и файлове и не е нито логически, нито синтактично сходен.

```
SELECT age, count(*) as broi FROM students GROUP BY age SORT BY broi LIMIT 5;
```

Hive превръща тази заявка в серия от map и reduce функции, за да калкулира резултата, като предоставя адрес, на който може да се следи статуса процентът изпълнение, като връща резултата, когато приключи.

Подробно описание на езика е достъпно на адрес:
http://wiki.apache.org/hadoop/Hive/LanguageManual

Case study за употребата на Hive във Facebook [13]:
http://www.slideshare.net/zshao/hive-data-warehousing-analytics-on-hadoop-presentation

Ненужна употреба

В контекста на единствен сървър, работата с файлове по подобен начин е изключително ненужна. Pig и Hive са средства за работа в разпределени бази данни, с големи масиви от информация.

1 GB файл в някакъв формат може да бъде обработен с добре познатите cat, awk, grep и sed и при все голямото удобство да бъде управляван с Hive, това най-вероятно не е необходимо.

Допълнителна информация "sed & awk" - Dale Dougberty, Arnold Robbins.

HBase

HBase е разпределена колонна NoSQL база от данни, изградена върху HDFS. HBase е програмата от Hadoop, която трябва да се използва в случай на натоварен достъп за четене и писане до особено големи обеми данни.

Част от Hadoop е от 2008 г.

Общо описание на начина на работа на HBase

Данните се съхраняват в таблици. Таблиците съдържат редове и колони. Клетките на таблицата имат версия, която по подразбиране е timestamp (дата с голяма точност). Съдържанието е неинтерпретиран масив от байтове. Първичните ключове (PK) са също масиви от байтове, а редовете са подредени по тези ключове (PK). Достъпът до данните става по PK.

Колоните са групирани в семейства (column families, вж. Dynamo). Всички членове на едно семейство имат общ префикс (prefix:column). Съществува схема на таблицата, която описва всички семейства колони, но самите колони могат да се добавят при нужда

Таблиците се разделят на части от HBase в отделни региони. Всеки регион съдържа част от редовете на таблицата. Регионът е дефиниран от своя първи и последен ред плюс

идентификатор на региона, който е случаен. Регионите са най-малката единица, която се дистрибутира в клъстера.

Съществува модел за заключване на ниво ред.

Интерфейси за достъп

HBase поддържа REST API, както и Thrift API.

В случай, че е избран REST, в зависимост от отделната заявка, резултатът се връща в JSON или XML.

> *Подробна информация за това как работи REST API на Hbase има на адрес: http://wiki.apache.org/hadoop/Hbase/HbaseRest*
>
> *Въпреки съществените разлики спрямо Riak и CouchDB, принципът е същия и REST API е разгледано основно за CouchDB, като най-подробно документирано.*

MemcacheDB

MemcacheDB е персистентната версия на софтуера за кеширане memcache. Memcache представлява система за съхраняване и достъп до двойки ключ-стойност в паметта на сървъра. MemcacheDB осигурява записване на тези двойки трайно и надеждно. MemcacheDB използва Berkeley DB за съхраняване на данните. Като резултат от това поддържа транзакции и репликация.

Кеш решението е Memcache. MemcacheDB е база от данни, а не кеш решение, но поддържа изцяло протокола на memcache и клиентите, разработени за работа с memcache могат идентично да използват и базата от данни.

Клиентски библиотеки съществуват на практически всички езици за разработка на уеб приложения - Perl, C, Python, Java и др.

Основни поддържани операции за работа с данни:
- get – извличане на 1 или повече двойки;
- set – съхраняване на информация;
- add – записване на информация, само ако не съществува;
hold data for this key";
- replace – замяна, ако такъв ключ съществува;
- delete – изтриване на данна по ключ;
- incr/decr – инкрементиране или декрементиране на integer;

MemcacheDB може да бъде използвана в случаите, когато е необходима голяма бързина на действие и всички свързани компромиси могат да бъдат направени. Основният конкурент е Redis, по-популярен последната година.

[56] - http://memcachedb.org/memcachedb-guide-1.0.pdf

Redis

Редис е база от данни, съхраняваща данните в паметта на машината, с осигурена персистентност. Написан е на C и е финансиран от VMWare.

Логиката на работа на Redis, като key/value storage, е подобна на memcachedb. Съхранява данните изцяло в паметта, като прави периодични записи на измененията на диска.

Redis може да се използва за съхраняване и извличане на атомарни стойности по особено бърз начин. Redis може да се ползва единствено, ако данните са достатъчно малки по обем, за да се побират в паметта на сървъра. Загубата на малък обем информация при спиране на ток или подобен отказ е допустима, за това не бива да се използва за приложения, в които това е проблем.

Редис поддържа в полето за стойност да се записват списъци. Списъците на Redis не са масиви, а свързани списъци. Това дава възможност дори при милиони елементи в един списък, добавянето на нов елемент с LPUSH командата да има константно време. Поддържат се също подредени списъци, както и SET обект – уникален списък със стрингове.

[58] - http://code.google.com/p/redis/wiki/IntroductionToRedisDataTypes

Разработва се в момента поддръжка на виртуална памет – чрез swap файл, което ще бъде представено във версия 2.0 на базата.

Дистрибуцията върху множество сървъри е възможна чрез Master/Slave репликация. Мастър сървъра може да има много слейвове, като слейвовете могат да се свързват помежду си. Репликацията не блокира мастър сървъра. Слейвовете могат да се използват за четене и извършване на тежки операции, като сортиране.

[57] – http://code.google.com/p/redis/wiki/ReplicationHowto

Redis може да се използва както за кеширане, така и за пълно функционален проект, благодарение на възможностите си за сортиране и заявки с диапазон. Примерните проекти с Redis използват описателни съставни ключове, за изграждане на модел на данните. Това не е особено удобен метод за работа, но е възможен.

Допълнителна информация:
http://rfw.posterous.com/a-redis-powered-newsfeed-implementation
http://nosql.mypopescu.com/post/522925057/redis-powered-facebook-like-newsfeeds
http://www.engineyard.com/blog/2009/key-value-stores-for-ruby-part-4-to-redis-or-not-to-redis/

Kyoto Cabinet

Kyoto Cabinet е база от данни, която по същество представлява текстов файл със записи, от които всеки следващ притежава уникален ключ и стойност. Всеки ключ и стойност е последователност от байтове с променлива дължина. Поддържа както символни низове, така

и бинарна информация. Не съдържа типове данни, нито таблици. Данните са подредени в хеш таблица или B+ дърво. Поддържа транзакции.

Kyoto Cabinet е текстова база от данни, чиято функция е да работи много бързо. Според официалния сайт, 0.9 секунди е средното време за добавяне на 1 милион записа в хеш база и 1.1 секунди в B+ дърво.

Производителността при употреба на хеш-база данни на теория би трябвало да е константна с нарастването на информацията. На практика, производителността се определя от дисковете. Ако свободната за употреба памет на сървъра е по-голяма от размера на базата от данни, ограничението идва от скоростта на паметта. Съществува ограничение за обема данни, което е 8 EB.

При употреба на B+ дърво производителността е намаляваща с логаритмична функция, но достъпът при четене е по-ефективен.

Kyoto Cabinet е написан на C++. Налични са библиотеки в C++, C, Java, Python, Ruby, Perl и други езици. Работи и под Linux, и под Windows.

[59] *http://fallabs.com/kyotocabinet/*

Neo4j

Neo4j е граф-база от данни, която съхранява информацията в графи, вместо в таблици. Neo4j е разработена на Java. Изисква всички операции да бъдат извършени в транзакция, която се описва с Java код. За индексиране се ползва отделно приложение, което не е вградено в базата от данни. Препоръчаното приложение в документацията е Lucene. Този софтуер осигурява възможността за търсене по диапазон (range query), както и сортирането на резултатите.

[61] *http://neo4j.org*

Базовия модел на Neo4j се състои от възли, връзки между възлите и свойства на възлите и връзките. Заявки за намиране на връзки могат да се правят чрез употреба на 2 различни езика – SPARQL и Gremlin.

[60] http://www.slideshare.net/directi/neo4j-and-the-benefits-of-graph-dbs-3-3325734

Neo4j е актуална база от данни, в контекста на социалните връзки в социалните мрежи. Може да извършва специфични задачи многократно по-бързо от RDBMS.

RavenDB

RavenDB е нова документна база от данни, забележителна с това, че е реализирана на C# - .net 4, Visual Studio 2010. Достъпна е от април 2010 г, след над 2 години разработка. RavenDB е с отворен код.

RavenDB работи под Windows, разполага с REST API което използва IIS уеб сървър. Разполага с шардинг и репликация. Евентуално консистентна е и на единичен нод, и в клъстер.

RavenDB разполага с достъпен през уеб административен интерфейс.

RavenDB поддържа индекси, които въпреки името си са подобни на изгледите в CouchDB и се състоят от map и reduce функции. Специфично в RavenDB е, че при създаването на такъв индекс може да се дефинира какво да вижда клиента, докато данните в изгледа се изравнят с данните в базата от данни. Опциите на "WaitForNonStaleResults" са 3:

- Клиентът да изчака обновяването на индекса при обновяване на данна;
- Клиентът да изчака прочитането на данни от индекс, ако не е текущия;
- Клиентът да не чака, просто да бъде информиран (по подразбиране).

[55] http://www.codeproject.com/KB/cs/RavenDBIntro.aspx

Допълнителна информация:
http://skillsmatter.com/podcast/open-source-dot-net/ayende-rahien-introduction-into-raven-db
http://weblogs.asp.net/britchie/archive/2010/08/17/document-databases-compared-mongodb-couchdb-and-ravendb.aspx

ZODB

ZODB е обектна база от данни, предназначена за трайно съхраняване на Python обекти. ZODB е ACID база от данни, работи в транзакции и поддържа "savepoints". Разполага с история на промените, както и с MVCC. ZODB е предшественик на съвременните NoSQL бази данни, но има практическо приложение и в момента.

Идеята за работа на ZODB е, Python обектите от класове, наследяващи persistent.Persistent клас, да са персистентни - т.е. състоянието им автоматично да се съхранява в базата от данни.

[54] http://www.zodb.org/documentation/guide/index.html

ZODB може да използва RDBMS за съхраняване на Python обекти.

ZODB се използва от Zope application server и Plone, и има практическо приложение, както и дял в популяризирането на Python.

Предимства и недостатъци:
http://www.coactivate.org/projects/topp-engineering/blog/2009/03/20/to-zodb-or-not-to-zodb/

Приложение на NoSQL DB. Перспективи за развитие.

NoSQL базите данни споделят общи принципи, но решават различни потребности. Поради това често в практиката на NoSQL DB се възлага изпълнението на една или няколко специфични задачи от проект, а не на целия проект и основната функционалност се реализира с RDBMS или друга NoSQL база от данни.

Yahoo използват Sherpa, но части от функционалността им са реализирани с Hadoop + Hbase. Google разполагат със собствената BigTable + GFS, но използват и свободната алтернатива Hadoop, както и MySQL.

При избор на база от данни трябва да се отговори на няколко основни въпроса. Може ли да се ползва евентуална консистентност (ако не, да бъде ползвана ACID база или RDBMS)? Кое е по-натоварващо – четенето или писането (някои БД, като RDBMS и CouchDB записват данните по-бавно)? Какви обобщения ще се правят и трябва ли данните да са актуални в реално време (нужна ли е поддържаща MapReduce база от данни)? Колко са данните и как ще нарастват във времето (redis не е решение за петабайти информация)? Какви изменения в модела могат да възникнат (нужна ли е schema-free база от данни)?

Няма прост отговор на въпроса коя е подходящата БД, но информираният избор изисква познаване на алтернативите. Към момента вероятно не е препоръчително да се използват NoSQL бази от данни, ако съществува решение с RDBMS, осъществимо в рамките на не-скъп хардуер и ако липсват перспективи за сериозно нарастване на обема данни и натоварване.

В перспектива, вероятно основните NoSQL DB ще достигнат стабилно състояние, при което ще е възможна мащабна употреба. Въпрос на време е да бъдат издадени първите книги за MongoDB и Cassandra, които имат потенциал за широко разпространение. Те ще бъдат приложени дори на места, на които RDBMS биха свършили работата достатъчно евтино и бързо.

Програмистите, използващи ORM слой (object-relational mapping), ще бъдат в състояние да мигрират от SQL към NoSQL и обратно, като изменят малка част от кода си, единствено в модел класовете (междинен слой за работа с документи се нарича ODM – object-document mapper). Поддръжката за различни NoSQL DB в популярни frameworks се разработва в момента, като за някои вече е реализирана.

5. Тестов пример за работа с NoSQL

Цел на тестовия пример е да се провери начина на работа с NoSQL архитектура при прост модел и натоварване, което може да възникне с реален проект и да се съпостави с идентичната ситуация при RDBMS.

Избрано бе в NoSQL базите от данни да се записват статии и коментари. Статиите имат автор, дата на публикуване, заглавие, текст и адрес. Коментарите имат автор, дата на публикуване, съдържание и се отнасят към статия.

Първоначалната цел беше да се направи сравнение между 1 документна база от данни, 1 k/v и MySQL при следните тестове – последователно записване на записи, конкурентно записване на записи от много нишки, последователно изчитане на записи в случаен ред, конкурентно изчитане на записи в много нишки. В процеса на тест се установи, че броят записи, с които е оправдано да се тества с наличния хардуер е в диапазона 1000 – 10 000.

За реализация на теста беше избран Python, като разработката беше извършена при едновременна употреба на Windows XP и Ubuntu Linux 10.04.

Използван хардуер

1. Laptop AMD Turion 1800, 1GB RAM, Windows XP – за писане на дипломната работа и пускане на тестовия скрипт + проба кои от базите данни и как ще работят под Windows. Тази машина ще бъде наричана по-долу “XP”
2. Desktop AMD Sempron 3000 384MB RAM, Ubuntu Linux 10.04 32bit – за тест в по-реална среда. Тази машина ще бъде наричана по-долу “node1”.

Използван софтуер и бележки по инсталацията

Беше избран Ubuntu Linux заради лесната инсталация, голямото общество, наличието на много помощна документация и package management системата с debian пакети – apt. Не беше заменена Windows операционната система на работния компютър, поради липса на дисково пространство неудобство в превключването между операционните системи. Интересна задача беше да се установи дали работещите под Windows NoSQL бази данни ще дадат сходна производителност с производителността им под Linux. Данните не биха били смислени при употреба на виртуални машини.

За всяка база от данни бяха извършени по (минимум) 4 инсталации – базата от данни, клиентска библиотека за Python за работа с нея – веднъж под Windows и веднъж под Linux.

1. Python + setuptools

Под Windows се наложи инсталация на стара версия – Python 2.6.5, тъй като с актуалната 2.7 не бе възможно с разумни усилия в рамките на 1 ден да се пусне библиотеката mysql-python.

2.6 Python е вграден в Ubuntu Linux, не се налага да се инсталира нищо допълнително.

И за двете OS е необходима python package management системата easy_install, която се инсталира чрез:

```
$> sudo apt-get install setuptools
```

Под Windows:

http://pypi.python.org/pypi/setuptools

2. MySQL 5.1.50 и MySQLdb

Под Linux:

```
$> sudo apt-get install mysql-server
$> sudo apt-get install mysql-python
$> sudo vi /etc/mysql/conf.d/bind.conf

[mysqld]
bind-address            = 0.0.0.0
```

```
$> sudo /etc/init.d/mysql restart
```

Това съдържание е необходимо, за да е видим MySQL от Windows. Не бива да се поставя в my.cnf, за да няма препокриване на конфигурацията при обновяване на софтуера.

Под Windows (беше наличен):

http://dev.mysql.com/downloads/mysql/ - MySQL 5.1 беше предварително инсталиран.

http://www.codegood.com/ - за MySQLdb, инсталатор

бяха добавени пътищата в към инсталациите на mysql bin, python bin и easy_install Path на Windows, за да са достъпни директно в конзолата Command Prompt.

Посоченото по-горе решение с codegood.com и downgrade към python 2.6.5 беше взето след преминаване през десетки различни грешки и употребата на различни компилатори.

На всички таблици, collation-и, бази данни default charset, connection charsets, навсякъде трябва изрично да е указано енкодинг utf8, за да може mysqldb да работи с utf8 под Windows. Един неудачен пропуск и няма да работи коректно. MySQLdb.escape_string в Python не проработи с utf8. Методът cursor() връща инстанции на String, не на Unicode, което в Python 2.6 са различни типове.

3. CouchDB и couchdbkit

Под Linux:

```
$> sudo apt-get install couchdb
$> sudo easy_install couchdbkit
$> sudo vi /etc/couchdb/local.ini
bind_address = 0.0.0.0
```

отново, промяната е нужда за видимост в мрежата.

```
$> sudo /etc/init.d/couchdb restart
```

При първоначалната инсталация couchdb не бе възможно да се рестартира със стартовия скрипт в /etc/init.d.

Под Windows:

http://couchdb.apache.org/downloads.html

```
easy_install.py couchdbkit
```

> *Важно е да се отбележи, че някои от примерите, дадени в официалната страница на couchdbkit не работят под Windows. Като цяло библиотеката работи без особени проблеми, ако се заобиколи неработещата функционалност. Библиотеката не е добре документирана и не функционира съвсем интуитивно, но е изчерпателна и богата на функционалност. Вероятно следващи нейни версии ще са използваеми в истински проект.*

Библиотеката разполага с възможност да изгражда документи като наследници на класа Document, както и с работа чрез вградения асоциативен масив dict. По-богатият на възможности и удобен вариант е чрез Document.

4. Redis и redis-py

Redis работи под Unix. На Windows беше инсталирана единствено клиентската библиотека, която функционира без забележки.

```
$> sudo apt-get install redis-server
$> sudo easy_install redis
$> vi /etc/redis/
bind 0.0.0.0
$> sudo /etc/init.d/redis-server restart
```

Windows:

```
Easy_install.py redis
```

5. MongoDB, pyomongo

За инсталация трябва да се следват инструкциите на:

http://www.mongodb.org/display/DOCS/Ubuntu+and+Debian+packages

```
$> sudo easy-install pymongo
```

Инсталацията, включена в apt по подразбиране не беше функционална към момента на тест.

Windows:
Download на инсталацията, unzip и стартиране на mongod като процес (поставянето Services не е необходимо с тестова цел) - http://www.mongodb.org/downloads

```
Easy_install.py pymongo
```

MongoDB заема място на блокове по 64 MB по подразбиране и не може да работи с над 2.5 GB данни на компютри като тези, с които разполагаме. Ползваната библиотека pymongo за MongoDB документ използва вградения dict обект в Python и не изисква създаването на собствени инстанции на Document.

6. Cassandra

Инсталация под Linux:

```
sudo apt-get install cassandra
sudo apt-get install python-dev
sudo easy_install python-cassandra
sudo easy_install lazyboy
sudo vi /etc/cassandra/storage-conf.xml
и редактиране на <ThriftAddress>0.0.0.0</ThriftAddress>
sudo /etc/init.d/cassandra stop
sudo /etc/init.d/cassandra start
```

Инсталация на клиентската библиотека под Windows – поради неуспешната инсталация на thrift, беше използвана стара версия thrift, компилирана предварително:
http://www.dreamcubes.com/webdrive/thrift/thrift-python-2.6.4-win32.zip
разрхивирана в site-packages. След това
easy_install.py lazyboy

Първоначално Cassandra отказа да работи с Python и за това бяха проведени тестове с алтернативната й Riak. В последствие беше открит начин за заобикаляне на проблема и тестът беше проведен. Cassandra беше най-трудна за инсталация от всички БД.

7. Riak + riak

Информация как се инсталира Riak на Ubuntu има на адрес:
https://wiki.basho.com/display/RIAK/Installing+on+Debian+and+Ubuntu

Инсталация под Windows не е налична.

Riak са първата база от данни от тестваните, която предлага Python библиотека на официалния си сайт:
http://github.com/basho/riak-python-client

Python клиентът е с package name "riak", въпреки името си "riak-python-client". В google нямаше наличен нито един пример за работа, което вероятно значи слабо популярна комбинация от база от данни и клиент. Инсталира се с easy_install (аналогично на по-горе)

В /etc/riak/app.config 127.0.0.1 е посочено на 2 места трябва да се замени с 0.0.0.0 (или с външното IP на тестовия сървър).

Модел на базата от данни

Първоначално беше избрана следната схема:
Posts: ID PK, Author, URL, Title, Body, Published, post_id – от оригиналната таблица.
Comments: ID PK, POST_ID FK, Author, Body

В процеса на разработка на теста се оказа, че за тест за записване на информация не са нужни 2 таблици. За това бяха използвани само статиите.

По отношение на коментарите – основната цел беше изграждането на йерархичен документ в CouchDB. След разговор със специалист в областта, тази идея се оказа подходяща единствено за тест, но не за реална работа, въпреки че е широко изтъквана като предимство във форумите.

```
class Post(Document):
    author = StringProperty()
```

```
    body = StringProperty()
    url = StringProperty()
    title = StringProperty()
    published = DateTimeProperty()
    comments = ListProperty() #това е списък с коментари
```

Какво се случва в CouchDB с така избрана схема? При всяко добавяне на коментар ще се запише нова ревизия на документа в append режима на файловете. Ако има 2000 коментара към 1 статия, това би означавало първия коментар да се дублира 2000 пъти, без да е изменян. Йерархичен документ в CouchDB е добре да се ползва там, където природата на обекта изисква подобна йерархия, а измененията са редки или отсъстват, пример - фактура. Вероятно бъдещи публикации ще изяснят този въпрос. Това не е валидно за MongoDB. В MongoDB могат да се използват.

В Cassandra моделът на базата от данни се описва в конфигурационен файл, в който бе добавено следното:

```
      <ColumnFamily Name="Posts" CompareWith="BytesType" RowsCached = "10000"
KeysCached="100%" />
      <ColumnFamily Name="Comments" CompareWith="BytesType" ColumnType="Super"
RowsCached="10000" KeysCached="100%" />
```

И беше преименуван keyspace на Blog1.

Тест 1 – запис на 1000 документа в една нишка

Тестът представлява на записването на един кеширан dataset от redis в някоя от базите данни, запис по запис, последователно, в една нишка. Стартирането (с DB Cassandra) става така:

```
D:\html\diplomna>couch6.py -h 192.168.1.5 -d cassandra -t write -r 1 -p 1000 -e
192.168.1.5
ThreadedDocsTest(connect_to = 'cassandra', dbhost = '192.168.1.5', test = 'write
-test', add_comments = 'False')
Tasks: 1, Thread count: 1
total_time: 0:00:08.078000
```

Целта на този тест е определен брой реални записи да бъдат вложени в няколко различни бази данни и да се направи сравнение на времената за изпълнение. Кодът на теста е наличен в

приложението. Първоначалният план от 100 000 документа се оказа неразумен, предвид неочаквано високите времена, които дадоха някои от DB сървърите на остарелия хардуер (или поради пропуски в конфигурацията).

За източник бяха ползвани реални статии от блогове от http://topbloglog.com , с реални адреси и реално съдържание, средно 2.3 KB на запис. Някои от постовете съдържаха corrupt-нати utf-8 символи, поради което се наложи навсякъде да се извършва допълнителна стъпка по почистването им.

Резултати от теста, всеки тест е извършен по 3 пъти, в общия случай с еднакъв резултат (по-ниският резултат е по-добър):

Таблица 1. Резултати от тест за запис на 1000 статии в NoSQL DB

База от данни	XP->XP	XP->Linux	Linux->Linux
MySQL	0:40 min	0:49	
CouchDB 1.0	0:13 min		
CouchDB 0.10		0:28 min	45.0 s
Redis		0:02.6 min	
MongoDB	0:04 min	0:06.3 min	
Riak		1:13 min	
Cassandra		0:08.0 min	

Възможен е пропуск в конфигурацията на Riak поради неочаквано по-слабия резултат. В теста бяха следвани инструкциите на basho.com. Пробван беше и различен bucket. Вероятно има начин да се постигне повече.

Версията CouchDB под XP и Ubuntu е различна, разликите във времената между CouchDB на XP и на Linux могат да се дължат на това.

За такъв тип натоварване, от тестваните бази данни се справи най-добре MongoDB. Обичайно в подобни тестове MongoDB не участва, а се тестват Cassandra и Hbase. Hbase може да се пусне с Ubuntu:

http://wiki.apache.org/hadoop/Running_Hadoop_On_Ubuntu_Linux_(Single-Node_Cluster)
но инсталацията на “Single node cluster” не изглежда като подходяща за тестове.

Бързият резултат на MongoDB се обяснява с нивото на надеждност при писане – MongoDB връща, че данните са записани, още преди те да са попаднали където и да е, и е възможно с тях да се случи нещо, с по-ниска защита от INSERT DELAYED в MySQL. Подробна статия, от разработчик на CouchDB, има на адрес:

[51] http://www.mikealrogers.com/2010/07/mongodb-performance-durability/

Според Майкъл Роджърс, надеждността при запис на MongoDB е нулева (което не може да се каже дори за redis). Ако направим сравнение с RDBMS, ACID гарантира единствено, че транзакцията прехвърля базата от данни от едно консистентно състояние в друго, не че недозаписания резултат може да бъде спасен, ако хардуера откаже. Т.е. разликата е в това, че RDBMS няма да върнат потвърждение, преди да има реално записана информация.

Извод от теста – за база от данни, в която времето за писане е решаващо, MongoDB вероятно е добър избор. Разликите във времето за изпълнение спрямо MySQL са от порядъка на 8-10 пъти, въпреки че има резерви за оптимизация, чрез компромиси.

Наблюдение 1. Изменение на заеманото дисково пространство за 1000 записа

Първоначално планирано беше да се наблюдава колко място заемат отделните бази данни. След стартирането на първите 3 теста, този показател се оказа несъстоятелен, поради несравнимост на това, което извършват различните бази данни:

1. CouchDB е append-only, т.е. DB файловете нарастват до извършване на compact опреация. От чиста база, заетото място беше 6.6 MB.
2. MongoDB расте на много големи фрагменти (64MB), тест може да се направи с милиони документи, не е обоснован с хиляди.
3. redis заема RAM + 1.1MB disk space (след първоначалния тест беше използван за кеширане).
4. riak с bitsack изгради 66 директории, които заеха 9.1 MB.

```
$> du –ch /var/lib/riak
```

5. MySQL от INFORMATION_SCHEMA.TABLES върна 4.7 MB за обем на данните.
6. Cassandra е commit-log based и при нея заеманото пространство също не е измеримо, но след първите 1000 записа в директорията имаше 3.9MB (част от тях вероятно заради извършените преди това тестове).

Изводът – целта на употреба на NoSQL не е пестене на дисково пространство, а пестене на seek операции на HDD. Методиките за борба с това са различни, но се употребява по-голямо пространство от заеманото в RDBMS.

Тест 2 – записване на 5000 записа в 50 нишки

Тестът представлява записване на 5000 записа със статии, в 50 конкурентни нишки, на страници от MySQL по 100 записа, с кеширане в redis и засичане на времето до приключване на най-бавната нишка.

Стартирането на теста с базата Cassandra:

```
D:\html\diplomna>couch6.py -h 192.168.1.5 -d cassandra -t write -r 50 -e 192.168
.1.5
ThreadedDocsTest(connect_to = 'cassandra', dbhost = '192.168.1.5', test = 'write
-test', add_comments = 'False')
Tasks: 50, Thread count: 50
total_time: 0:00:23.360000
```

Таблица 2. Резултати от тест за запис на 5000 записа в 50 нишки по 100 записа за всяка.

База от данни	XP	Ubuntu
MySQL	1:02 min	1:05
CouchDB	0:55	1:08
MongoDB	0:18	0:21
Riak	N/A	неуспешен
Redis	N/A	0:14.2
Cassandra	N/A	0.23

Riak python библиотеката не е thread safe и този тест не проработи с riak. Трябва да се отбележи, че описанието как се инсталира riak на Ubuntu е невярно. След рестартиране riak не функционираше и всеки опит да се пусне беше неуспешен. Неуспешният тест е след преинсталация и повторен опит.

Многонишковият тест за писане показва, че вероятно част от забавянето в теста с 1 нишка на MySQL и CouchDB е причинено от опит да осигурят консистентност на базата от данни. При 5-кратно увеличаване на записите, времето за изпълнение се променя в по-малък мащаб. Тестът не се опитва да изолира времето за запис от времето за обработка. Времето за обработка е от порядъка на 0.3 сек/1000 записа и не е пренебрежимо само в тестовете на Redis и MongoDB.

CouchDB върна грешка в записването на няколко статии при част от повторенията на теста под XP, като не възприе датата на публикуване за валидна. Идентични проблеми имаше с библиотеката в конзолен режим при четене на изглед. Не беше изолиран модел, в който се връщат такива грешки. В общия случай CouchDB работи коректно и вероятно става въпрос за специфични особености под XP, като вероятно не е желателно тази база от данни да се използва под Windows OS за цели, различни от разработка.

Извод от теста – MongoDB и Cassandra показаха до 3 пъти по бърза производителност при описаното натоварване. Redis показа около 4 пъти по-бърза работа. CouchDB, въпреки демонстрираното по-високо време на запис на данни, се представи по-добре от MySQL.

Тест 3 – прочитане на 1000 статии с коментарите им

Тестът представлява реалистична справка, като се изчитат и зареждат в dataset по id последователно 1000 различни статии, както и коментарите към тях. За целта бяха добавени фиксиран брой коментари към статиите, така че на всяка статия да съответства равен брой коментари във всяка от тестваните бази.

В CouchDB беше изтрита базата от данни и добавен следния изглед, преди теста:

```
{
   "_id": "_design/single",
```

```
   "views": {
       "by_post_id": {
           "map": "function(doc) { if (doc.doc_type=='Post' ||
doc.doc_type=='Comment') emit(doc.post_id, doc) }"
       }
   }
}
```

Този изглед дава комбинирано пост + всички статии с 1 GET request, индексирано по колоната post_id, която не е id от couchdb, а прието от mysql.

```
GET /couch3test/_design/single/_view/by_post_id?key=375123
```

В MongoDB бяха добавени през mongodb конзолата индекси на колекциите posts и comments по post_id:

```
> use diplomna_test
> db.posts.ensureIndex({post_id:1})
> db.comments.ensureIndex({post_id:1})
```

В MySQL беше добавен индекс на полето post_id в comments:

```
alter table comments add index ixp(post_id);
```

След което бяха попълнени базите данни с 1000 поста и 4493 коментара.

В Cassandra беше използван post_id от MySQL за ключ (в отсъствието на MySQL ключ, Касандра вероятно е удачно да ползва външна система за генериране на приблизително последователни ID-та, от вида на Snowflake).

Тестът представлява прочитането на 1000-та поста с коментарите към всеки, 1 по 1, в 10 нишки едновременно.

```
D:\html\diplomna>couch6.py -h 192.168.1.5 -d cassandra -t read -r 10 -e
192.168.1.5
ThreadedDocsTest(connect_to = 'cassandra', dbhost = '192.168.1.5', test = 'read-
test', add_comments = 'False')
Tasks: 10, Thread count: 10
total_time: 0:00:05.406000
```

Таблица 3. Време за прочитане на 1000 статии и коментарите им в 10 нишки.

База от данни	XP	Ubuntu – node1
MySQL	5.9 sec	2.4 sec
CouchDB	0.6 sec	0.57 sec
MongoDB	1.15 sec	1.95 sec
Redis	N/A	2.3 sec
Cassandra – без cache, ONE	N/A	5.4 sec
Cassandra – с cache, ONE	N/A	3.0 sec

Cassandra бе стартирана в 2 различни конфигурации, с и без активиран кеш.

Документната архитектура на CouchDB спечели безкомпромисно срещу останалите бази данни. Резултатът на Redis е по-слаб от очакваното, възможно е да се дължи на неправилно конфигуриране. По-слабият резултат на MongoDB спрямо CouchDB може да се дължи на това, че в конкретния случай за MongoDB би била по-подходяща архитектура с йерархичен документ, а не на два отделни документа за статия и коментари, поради липсата на MVCC. Друга възможна причина е библиотеката pymongo, която консумира повече оперативна памет от python-mysql, а тестовите системи имат недостиг на памет.

Конфигурацията на Cassandra позволява да се определи колко и какво да се кешира от всяко семейство колони. Резултатите при наличие и отсъствие на кеш се различават при повторно изпълнение. Независимо от това, резултатът е по-лош от този на MySQL. Вероятна причина – библиотеката lazyboy може да подава заявките последователно, а не паралелно.

Извод – употребата на NoSQL може да доведе до сериозно освобождаване на ресурс при четене на случайни данни, както и до намаляване на времето за изпълнение на заявки. В теста резултатите варират – от равно време, до 10 пъти разлика в полза на CouchDB 1.0 под Windows спрямо MySQL.

Тест 4. Статия + брой коментари

Беше поставена цел да се създаде справка, аналогична на Group by в SQL, за да се изясни колко лесно се постига това в NoSQL.

Със следния дизайн документ се постига групиране в CouchDB:

```
{
   "_id": "_design/all",
   "views": {
       "by_post_id": {
           "map": "function(doc) { if (doc.doc_type=='Comment')
emit(doc.id_post, 1) }",
           "reduce": "function(key, values, rereduce) { return sum(values); }"
       }
   }
}
GET /couch3test/_design/all/_view/by_post_id?group=true

{"rows":[
{"key":"00bcb73b5bbb75f9dc7fe4130d000f6d","value":5},
{"key":"00bcb73b5bbb75f9dc7fe4130d003a1f","value":4},
{"key":"00bcb73b5bbb75f9dc7fe4130d006118","value":5},
...
```

Подреждане по броя резултати обаче се оказа изпълнимо единствено с хак за CouchDB 1.0. Очевидният начин на реализация на подреждане по брой коментари е чрез записване на поле в posts с брой коментари, което би могло да се реализира и в MySQL. Хакът е чрез записване на резултатите от горния изглед в нова база от данни ordered_results и създаване на _list функция за визуализацията им, като ключ е броя повторения.

Информация: *http://stackoverflow.com/questions/2817703/sorting-couchdb-views-by-value*
http://geekiriki.blogspot.com/2010/08/couchdb-using-list-functions-to-sort.html

MongoDB има друг подход за работа с MapReduce. В MongoDB MapReduce е команда, която връща колекция. Ако колекцията се дефинира да е постоянна, тя може да се използва като таблица, да й се постави индекс и да се подреди по брой коментари.

[53] http://www.slideshare.net/gabriele.lana/couchdb-vs-mongodb-2982288

Ето как бе реализирано това в процеса на разработка на настоящата работа:
1. Map и reduce функции. Map дава emit на полето, по което ще се групира и 1. Reduce сумира единиците.

```
> map = function() {
... emit(this.post_id, 1)
```

```
... }
function () {
    emit(this.post_id, 1);
}

> reduce = function(previous, current) {
... var count = 0;
... for (i in current) {
... count += current[i];
... }
... return count;
... }
function (previous, current) {
    var count = 0;
    for (i in current) {
        count += current[i];
    }
    return count;
}
```

Това е ключовата разлика с CouchDB – групирания резултат може да се запише като постоянна колекция ‘groupedComments’:

```
> res = db.comments.mapReduce(map, reduce, {'out': 'groupedComments'})
{
        "result" : "groupedComments",
        "timeMillis" : 599,
        "counts" : {
                "input" : 4493,
                "emit" : 4493,
                "output" : 903
        },
        "ok" : 1,
}
> show collections
comments
groupedComments
posts
system.indexes
```

След това може в последствие да се сортира, филтрира, индексира и т.н.

```
> db.groupedComments.find().sort({value:-1})
{ "_id" : 374339, "value" : 9 }
{ "_id" : 374369, "value" : 9 }
{ "_id" : 374389, "value" : 9 }
{ "_id" : 374399, "value" : 9 }
```

Т.е. MongoDB може да се използва и за сложни групирания, каквито се правят в SQL.

В Cassandra преброяването на статиите към 1 пост е много просто:

```
cassandra> count Blog1.Comments['374333']
3 columns
```

Но изграждането на подреден изглед изисква приложна логика, която не е непременно трудна за реализация. Подобни обобщения се извършват от digg.com именно с библиотеката lazyboy, с която е реализиран теста в дипломната работа.

Извод от теста – извършването на обобщения в документните бази от данни CouchDB и MongoDB е възможно, но по по-сложен начин от този в SQL. За разлика от RDBMS обаче, тези обобщения могат да бъдат приложени върху данни, които биха представлявали трудност с всяка релационна база от данни.

Тест 5. MongoDB Replica Sets

Реализирана беше с тестова цел успешно репликация с MongoDB, между Win32 и Linux. Тези бази от данни са разпределени, биха могли да се реализират всякакви клъстери.

Последователността на свързване на тестовите машини node1 и xp беше следната. Първо изключване на двата MongoDB процеса, изтриване от node1 на файловете с данни от всички предишни тестове и стартиране наново на двата mongod процеса с опцията –replSet:

```
D:\path-to-mongo\mongod –replSet xp
./mongod –replSet xp
```

Следващата стъпка е създаване на конфигурация за replica set в конзолата на master-а, XP:

```
> cfg = {
        "_id" : "xp",
```

```
        "members" : [
                {
                        "_id" : 0,
                        "host" : "192.168.1.5"
                },
                {
                        "_id" : 1,
                        "host" : "192.168.1.6"
                }
        ]
}

> rs.initiate(cfg)
> use reptest
switched to db reptest
> db.a.insert ( { hello: 'world'} );
```

в конзолата на slave server-а:

```
> db.a.find({})
error: { "$err" : "not master", "code" : 10107 }
```

IP адресите са използваните по време на теста.

По подразбиране slave server-ите са единствено за backup и с преминава към тях, когато master сървъра откаже. Това е дублаж (redundancy) за отказоустойчивост. Ако се иска slave server-а да поеме част от натоварването, може да се активира четене от него с команда към master server-а XP:

```
> use admin
> db.getMongo().setSlaveOk()
```

и съответно в конзолата на node1:

```
> db.a.find({})
{ "_id" : ObjectId("4c713d191620000000002348"), "hello" : "world" }
```

Добавянето на slave сървъри може да се автоматизира.

Подробна информация за това има налична на:
http://www.snailinaturtleneck.com/blog/category/mongodb/

Извод от теста – изграждането на Replica Set в MongoDB е тривиално, а добавянето на нови сървъри може да се автоматизира.

Извод от практическата част

Най-добре представилата се база от данни в периода на тестове е MongoDB. Инсталира се най-лесно, конзолният й шел е най-интуитивен и функционира най-разбираемо. Тя е най-богата на функционалности и най-близка като начин на мислене до RDBMS, въпреки че липсва достатъчно образователна информация. Изграждането и стартиране на тест отне минути.

Трябва да се има предвид случилото се с Foursquare:
http://blog.foursquare.com/2010/10/05/so-that-was-a-bummer/
Базата от данни е нова и са възможни инциденти.

Като архитектура, най-добре се представи CouchDB. Тя се справи най-добре с изграждането на изглед, разполага с REST интерфейс и уеб администрация, през която лесно може да се следят резултатите. Изграждането и стартиране на тест отне часове в няколко последователни дни, но веднъж преодолени различията, работата би вървяла по-лесно.

Redis свърши своята работа да кешира MySQL добре, но би могъл да се използва за много повече. Използването му като самостоятелна база от данни означава усложнена приложна логика, което не е добра идея.

Популярните бази данни Cassandra и Riak създадоха трудности на всеки етап от опитите с тях. Отне повече от 2 седмици от първоначалната инсталация на Cassandra до успешно стартиране на последния тест, заради липса на документация и необясними грешки. Тези две бази данни не изглеждат приложими към момента, без допълнителна документация и специална мотивация за употреба.

6. Заключение

NoSQL базите данни са създадени да решават специфични проблеми, като мотивацията на създателите на различните бази е била различна и като резултат всяка от тях е оптимална за употреба в различни случаи. RDBMS са универсални, но универсалността им идва на висока цена – в хардуер, електричество и лицензи. Това мотивира разработката на NoSQL бази данни, върху които функционират много от най-големите Уеб услуги в света.

Oracle и MS SQL Server не са алтернатива за големи по размер проекти поради високата си цена, надхвърляща 50 000 долара за сървър. MySQL се използва за натоварени проекти, като Google Adsense, но за да бъде накаран да работи, обикновено се свежда до key/value store и се ползва бедния на възможности MyISAM storage engine.

NoSQL DB правят възможно съществуването на уеб услуги, които не биха могли да съществуват по друг начин. Hadoop и Cassandra движат Facebook, Twitter, отделни услуги в Yahoo. CouchDB стои зад bbc.co.uk. SourceForge и Foursquare използват MongoDB. Основни услуги на Yahoo са Sherpa, на Google – BigTable, на Amazon – Dynamo, а всички те ползват и други NoSQL решения в определени случаи.

Моментът, в който най-големите услуги в света са преминали от RDBMS към NoSQL вероятно вече е настъпил.

В практическата част бе демонстрирано, че въпреки различната си архитектура, NoSQL базите от данни са използваеми и способни да дадат сериозно подобрение на производителността – независимо, дали използвани за цял проект, или за някаква негова част. Има какво да се желае по отношение обезпеченост с библиотеки, богатство на функционалност и удобство на работа, но пречките са преодолими и поставените задачи могат да бъдат изпълнени.

Изводи от разработката:

- NoSQL не дава възможност за по-лесно писане на заявки за програмистите.

- NoSQL дава възможност малка услуга да стане голяма и после да стане световна, единствено с добавяне на евтина техника.
- NoSQL е възможност обобщенията и изгледите да са бързи, разпределени, материални и управлявани от базата от данни.
- NoSQL е възможност да се обобщават паралелно данни на множество сървъри.
- NoSQL ДБ в момента са необходими, но не достатъчни, за изграждането на комплексен и натоварен уебсайт.

Свършено бе най-много, за най-малко време с MongoDB. Постигнат бе най-добър резултат с изгледите на CouchDB. Личният избор на автора на настоящата работа за NoSQL база, която да вложа в реален проект, като алтернатива на RDBMS пада върху тези две бази данни. Все пак, обхватът на разработката и проведените тестове е твърде малък за категорични изводи. При необходимост от внедряване на такава система трябва да се проведат по-специализирани и продължителни тестове, като се включи и Hbase, задължително върху клъстер от поне 3 машини в подходяща Linux дистрибуция (вероятно не Debian/Ubuntu).

Няма ясна тенденция за налагане на доминираща NoSQL база от данни или дори NoSQL модел. Има тенденция една от силните страни на NoSQL базите данни, MapReduce, да бъде внедрена в RDBMS, под различни форми, както и обратното - да се изграждат SQL-подобни езици за генериране на MapReduce заявки. NoSQL е в процес на бурно развитие.

Надявам се дипломната работа да е полезна на всички, които имат желание да навлязат в NoSQL.

Веселин Николов
София, октомври 2010 г.
M22528

7. Използвана литература

Учебници:

1. CouchDB: The Definitive Guide – J. Anderson, J. Lehnardt, N. Slater (основен източник за CouchDB и общите части за NoSQL)
2. Hadoop: The Definitive Guide – Tom White (основен източник за Hadoop и HBase)
3. Beginning CouchDB – Joe Lennon (употребен за CouchDB раздела)
4. High Performance MySQL – B. Schwartz, P. Zaitsev, V. Tkachenko, J. Zawodny, A. Lentz, D. Balling. (за MySQL клъстъризация)
5. Microsoft SQL Server 2005 – Наръчник на администратора (по въпросите, касаещи MS SQL Server)
6. JavaScript: The Definitive Guide – D. Flanagan (JSON дефиниция, обща част)
7. Pro Hadoop - Джейсън Венер

Публикации:

8. Amazon Dynamo Paper
9. Bigtable: A Distributed Storage System for Structured Data
10. Towards Robust Distributed Systems - Dr. Eric A. Brewer (презентация) http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
11. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services - Seth Gilbert, Nancy Lynch

Статии в блогове, презентации, online уроци, официални сайтове, документации:

12. Hadoop Training: Hive Tutorial - http://www.cloudera.com/videos/hive_tutorial
13. Hive - http://www.slideshare.net/zshao/hive-data-warehousing-analytics-on-hadoop-presentation
14. Презентация на #nosqleu 2010 за MongoDB - http://github.com/RedBeard0531/Mongo_Presentations/raw/master/20100420-nosql-eu.pdf
15. http://research.yahoo.com/files/10RamakrishnanSherpa.pdf
16. http://developer.yahoo.net/blog/archives/2009/06/sherpa.html
17. http://developer.yahoo.net/blog/archives/2010/06/sherpa_update.html
18. Google's BigTable by Philipp Lenssen - http://blogoscoped.com/archive/2005-10-23-n61.html
19. https://nosqleast.com/2009/slides/sarkissian-cassandra.pdf
20. http://weblogs.java.net/blog/2007/11/27/consistent-hashing
21. http://wiki.apache.org/cassandra/ArchitectureOverview
22. On Distributed Consistency — Part 1 - http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1
23. http://www.julianbrowne.com/article/viewer/brewers-cap-theorem
24. Cassandra inverted index - Royans K Tharakan. http://www.royans.net/arch/cassandra-inverted-index/
25. http://mercury.lcs.mit.edu/~jnc//tech/hard_soft.html

26. Apache Cassandra - Pragmatic Consistency for Massive Scale - Stu Hood. http://thestrangeloop.com/sites/default/files/slides/StuHood_Cassandra.pdf
27. Lecture Note on Consistency and Replication - Insup Lee. http://www.cis.upenn.edu/~lee/07cis505/Lec/lec-ch7b-replication-v3.pdf
28. On Distributed Consistency - Part 2 - Some Eventual Consistency Forms - http://blog.mongodb.org/post/498145601/on-distributed-consistency-part-2-some-eventual
29. Storing MySQL Binary logs on NFS Volume – Peter Zaitsev. http://www.mysqlperformanceblog.com/2010/07/30/storing-mysql-binary-logs-on-nfs-volume/
30. http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster-overview.html
31. http://oss.oracle.com/projects/ocfs/
32. GFS: http://labs.google.com/papers/gfs.html
33. GFS: http://static.googleusercontent.com/external_content/untrusted_dlcp/labs.google.com/bg//papers/gfs-sosp2003.pdf
34. За работата на BigTable върху GFS: http://static.googleusercontent.com/external_content/untrusted_dlcp/labs.google.com/bg//papers/bigtable-osdi06.pdf
35. http://hadoop.apache.org/common/docs/r0.18.3/hdfs_user_guide.html
36. Arin Sarkissian - An Intro to the Cassandra Data Model. http://arin.me/blog/wtf-is-a-supercolumn-cassandra-data-model
37. http://architects.dzone.com/news/cassandra-adds-hadoop
38. http://en.wikipedia.org/wiki/Gossip_protocol
39. Introduction to Cassandra: Replication and Consistency – Benjamin Black. http://www.slideshare.net/benjaminblack/introduction-to-cassandra-replication-and-consistency
40. Cassandra: RandomPartitioner vs OrderPreservingPartitioner - Dominic Williams. http://ria101.wordpress.com/2010/02/22/cassandra-randompartitioner-vs-orderpreservingpartitioner/
41. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.6951&rep=rep1&type=pdf
42. http://books.couchdb.org/relax/design-documents/shows
43. Zero to Mongo in 60 Hours – Ryan Angilly. http://www.slideshare.net/mongosf/going-from-zero-to-mongo-in-about-2-days-ryan-angilly
44. http://www.mongodb.org/display/DOCS/Database+References
45. http://www.mongodb.org/display/DOCS/Indexes
46. https://wiki.basho.com/display/RIAK/REST+API#RESTAPI-BucketOperation
47. Replica Sets Dwight Merriman http://www.slideshare.net/mongodb/replica-sets
48. MongoDB Documentation: Configuring Sharding http://www.mongodb.org/display/DOCS/Configuring+Sharding
49. Basic Cluster Setup – Riak https://wiki.basho.com/display/RIAK/Basic+Cluster+Setup#BasicClusterSetup-AddaSecondNodetoYourCluster
50. Why Vector Clocks are Easy – Bryan Fink. http://blog.basho.com/2010/01/29/why-vector-clocks-are-easy/
51. MongoDB performance and durability – Mikael rogers. http://www.mikealrogers.com/2010/07/mongodb-performance-durability/
52. http://www.mongodb.org/display/DOCS/MapReduce#MapReduce-NoteonPermanentCollections

53. CouchDB Vs MongoDB – Gabriele Lana. http://www.slideshare.net/gabriele.lana/couchdb-vs-mongodb-2982288
54. Официална документация на Zodb - http://www.zodb.org/documentation/guide/index.html
55. RavenDB - An Introduction - http://www.codeproject.com/KB/cs/RavenDBIntro.aspx
56. Memcachedb: The Complete Guide - http://memcachedb.org/memcachedb-guide-1.0.pdf
57. Redis: Replication Howto- http://code.google.com/p/redis/wiki/ReplicationHowto
58. Redis: A fifteen minutes introduction to Redis data types - http://code.google.com/p/redis/wiki/IntroductionToRedisDataTypes
59. Kyoto Cabinet - Overview http://fallabs.com/kyotocabinet/
60. Neo4j And The Benefits Of Graph Dbs - Emil Eifrem. http://www.slideshare.net/directi/neo4j-and-the-benefits-of-graph-dbs-3-3325734
61. Официален сайт на Neo4j - http://neo4j.org
62. Eventually Consistent – Revisited, Werner Vogels – за раздел евентуална консистентност и раздел Dynamo. http://www.allthingsdistributed.com/2008/12/eventually_consistent.html
63. Лекции основи на БД, доц. К. Калоянова 2007 г.

За MongoDB към момента на писане на дипломната работа нямаше издадена литература. Подготвят се 3 издания, едно от които може да бъде достъпно към момента на дипломна защита. Едно от изданията вече излезе в Rough cuts вариант, без да даде съществена нова информация.