VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

MONGODB JAKO DATOVÉ ÚLOŽIŠTĚ PRO GOOGLE APP ENGINE SDK

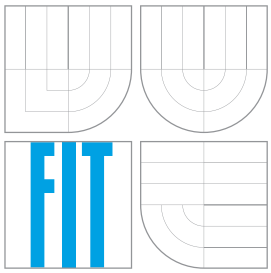
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. STANISLAV HELLER

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

MONGODB JAKO DATOVÉ ÚLOŽIŠTĚ PRO GOOGLE APP ENGINE SDK

MONGODB AS A DATASTORE FOR GOOGLE APP ENGINE SDK

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. STANISLAV HELLER

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ VOLF

BRNO 2013

Abstrakt

Tato práce se zabývá možnostmi použití NoSQL databáze MongoDB jako úložiště pro data, která jsou v Google App Engine SDK ukládána databázovými stuby - rozhraními pro simulaci produkčního databázového prostředí v Google App Engine. Existující stuby nejsou optimalizovány pro větší zátěž a při větším množství uložených dat značně zpomalují vývoj a testování celé aplikace. Práce analyzuje vlastnosti MongoDB a platformy Google App Engine se zaměřením na datové úložiště Google Datastore. Předmětem další analýzy je rozhraní pro implementaci databázových stubů v SDK. Výsledkem práce je návrh a implementace nového výkonnějšího stubu využívajícího MongoDB, který je plně integrovatelný do Google App Engine SDK.

Abstract

In this thesis, there are discussed use-cases of NoSQL database MongoDB implemented as a datastore for user data, which is stored by Datastore stubs in Google App Engine SDK. Existing stubs are not very well optimized for higher load; they significantly slow down application development and testing if there is a need to store larger data sets in these storages. The analysis is focused on features of MongoDB, Google App Engine NoSQL Datastore and interfaces for data manipulation in SDK - Datastore Service Stub API. As a result, there was designed and implemented new datastore stub, which is supposed to solve problems of existing stubs. New stub uses MongoDB as a database layer for storing testing data and it is fully integrated into Google App Engine SDK.

Klíčová slova

Google App Engine, NoSQL databáze, Google App Engine SDK, MongoDB, Datastore stub, Python, cloud, web.

Keywords

Google App Engine, NoSQL database, Google App Engine SDK, MongoDB, Datastore stub, Python, Cloud, Web.

Citace

Stanislav Heller: MongoDB jako datové úložiště pro Google App Engine SDK, diplomová práce, Brno, FIT VUT v Brně, 2013

MongoDB jako datové úložiště pro Google App Engine SDK

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Volfa.

.....
Stanislav Heller
13. května 2013

Poděkování

Tímto bych rád poděkoval panu Ing. Tomáši Volfovi za mnoho cenných rad, které mi poskytl při zpracování této diplomové práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Vývoj trhu v oblasti webových aplikací a služeb směřuje ke stále masivnějšímu nasazení distribuovaných technologií. Významným průkopníkem v této oblasti je bezesporu společnost Google, která již od roku 2008 provozuje službu *Google App Engine* (GAE) - platformu pro vývoj a hostování webových aplikací nad velmi výkonnou infrastrukturou, na které jsou vystavěny služby jako např. Gmail nebo Google Calendar. Tato služba umožňuje webovým vývojářům soustředit se na vývoj vlastní aplikace a její funkcionality bez nutnosti spravovat vlastní HW a SW infrastrukturu. Aplikace hostované na Google App Engine se vyznačují vysokou škálovatelností, vysokou dostupností a jednoduchou údržbou.

Z hlediska datových úložišť je GAE primárně orientován na NoSQL databáze[1], které (také díky Google) v poslední době zažívají obrovský nárust popularity a využití. Hlavním úložištěm dat v GAE je v dnešní době *High Replication Datastore* (HRD), který je vystavěn nad BigTable. Protože není možné toto úložiště provozovat lokálně[2], nastává problém, jak vyvíjet a testovat aplikace, které jsou určeny pro běh na GAE. Vývojáři Google proto vytvořili sadu nástrojů - SDK (Software Development Kit), která slouží pro simulaci prostředí, ve kterém aplikace poběží. Součástí Google App Engine SDK jsou tzv. *stuby*, což jsou rozhraní napodobující chování služeb provozovaných na GAE. Pro ukládání testovacích dat existují v SDK tzv. *Datastore service stuby*, které umožňují používat lokální persistentní úložiště (např. SQLite) a současně simulovat chování HRD, zvláště co se týče vlastností týkajících se eventuální konzistence dat.

V prvních verzích SDK byla k dispozici pouze jedna oficiální implementace datastore stubu - *Datastore File Stub*. Tento stub ukládá data i indexy do jednoho souboru, ale současně udržuje všechna data v paměti RAM, dotazování je neefektivní a v případě většího množství testovacích dat jsou operace nad touto databází velmi pomalé. V březnu 2010 byl po vyslyšení požadavků vývojářské komunity GAE vytvořen alternativní stub pro Python využívající SQLite jako databázové vrstvy (*Datastore SQLite Stub*). Toto řešení je na podstatně vyšší úrovni a řeší mnohé dětské nemoci Datastore File Stubu. Přesto tato implementace nedosahuje potřebných výkonnostních parametrů, zejména co se týká rychlosti vkládání a úpravy dat. Nabízí se tedy možnost vytvořit nový stub, který by využíval NoSQL databázi MongoDB.

Jedním z cílů této práce je zdokumentovat programové rozhraní (API) Datastore service stubů v GAE SDK pro Python - toto rozhraní není nikde (veřejně) zdokumentováno a proto je vývoj uživatelských stubů velmi obtížný.

[1] Alternativně lze použít Google Cloud SQL - úložiště postavené na základech MySQL.

[2] Implementace BigTable, příp. HRD nebyla a pravděpodobně nebude nikdy zveřejněna.

Hlavním cílem práce je návrh a implementace konektoru (stubu), pomocí kterého by bylo možné při vývoji nad GAE Python SDK používat databázi MongoDB. Práce tím navazuje na již existující knihovnu MongoDB AppEngine Connector [6], která umožňovala použití MongoDB jako úložiště pro starší verzi knihovny pro přístup ke GAE Datastore. Značnou nevýhodou tohoto konektoru je fakt, že nepodporuje knihovnu ndb, která je součástí nových verzí GAE SDK, nepodporuje transakční zpracování, nad novým SDK nefunguje práce s indexy a není plně integrován do SDK pro širší použití v testovacím prostředí. Nově vytvořený stub by měl tyto chyby odstranit a poskytnout tak celé Google App Engine komunitě možnost rychlejšího a efektivnějšího vývoje aplikací.

1.1 Struktura dokumentu

Na začátku práce je pozornost věnována obecně NoSQL databázím, kde jsou rozebrány jejich základní typy, vlastnosti, výhody a případné nevýhody. Dále se text specializuje výhradně na NoSQL databázi MongoDB. Ve třetí kapitole je analyzováno prostředí Google App Engine, jeho architektura, vlastnosti a poskytované služby. Nejvíce pozornosti je věnováno úložišti Google App Engine Datastore, které je pro tuto práci stěžejní. Práce dále popisuje principy a problémy Google App Engine SDK, kde se zaměřuji na existující *Datastore service stuby*. Druhá část práce je zaměřena na problematiku tvorby nového datastore stubu pomocí MongoDB. V následujících kapitolách je popsán návrh a vývoj *Datastore MongoDB Stub*u, jeho optimalizace a způsoby testování. Výkon nového řešení je vyhodnocen pomocí sady testů, jejichž výsledky jsou porovnány proti existujícím datastore stubům. V sedmé kapitole je popsán způsob integrace stubu do SDK a principy vývoje, které by měly být dodržovány pro kompatibilitu s dalšími verzemi SDK. V závěru jsou diskutovány hlavní výhody nového stubu, možnosti jeho rozšíření a směr dalšího vývoje.

Kapitola 2

NoSQL databáze

V posledních letech je v oblasti databázových systémů zřetelný trend využívání úložišť, která jsou označována jako NoSQL. Zkratka, jejíž význam je interpretován jako „Not Only SQL“, v sobě zahrnuje několik typů databází a velké množství jejich implementací. Základně jsou do této třídy databází zahrnuty databáze nerelační, vysoce škálovatelné, distribuovatelné a velmi výkonné[1].

NoSQL databáze se zásadně liší od relačních jak způsobem komunikace s databází, způsobem organizace dat v DB, tak i poskytovanými možnostmi transakčního zpracování nebo integritních omezení. Velmi často se jedná o databáze bez schématu (tzv. *schema-free*). Na rozdíl od relačních DB, každá NoSQL databáze je primárně určena pro určitý typ dat - je zde tedy většinou opuštěn koncept obecnosti a db se stávají více specializované.

Ve většině NoSQL databází nenalezneme vestavěné transakční zpracování[2], triggery nebo vestavěné procedury. Pro udržení integrity dat jsou v NoSQL databázích často jen jednoduché integritní omezení na unikátnost primárního klíče (ID), cizí klíče zde většinou vůbec neexistují. Neexistují zde také operace typu JOIN. V některých typech jednodušších NoSQL db neexistují dokonce ani indexy - je ale možné je velmi efektivně implementovat.

NoSQL databáze disponují odlišným systémem dotazování; některé se snaží přiblížit SQL a implementují vlastní podmnožinu jazyka SQL ke kterým přidávají specifické DML[3] a DQL[4] operace (Cassandra CQL, Google Datastore GQL atp.), jiné používají vlastní dotazovací jazyk nebo využívají HTTP/REST rozhraní (např. *CouchDB, Riak*).

2.1 Rozdělení NoSQL databází

NoSQL databáze lze klasifikovat jak podle jejich možného (či určeného) využití pro jisté typy úloh, ale také podle způsobu uložení dat v db. Každý takový druh databáze pak implementuje nějaký kompromis mezi množstvím a kvalitou poskytovaných vlastností, rychlostí daných operací, škálovatelností a pohodlím databázového programátora nebo adminstrátora.

NoSQL databáze lze základně rozdělit do pěti kategorií[13]:

- **Key-value store** jsou velmi jednoduché databáze založené na datovém modelu aso-

[1] Výkonnost NoSQL databází je nutné vztáhnout vždy ke specifické úloze, ke které jsou vytvořeny - např. rychlé vkládání nebo dotazování na nerelační data.

[2] Některé NoSQL databáze podporují transakční zpracování - např. Redis nebo Neo4j.

[3] DML (Data Manipulation Language) je rodina jazyků pro vkládání, úpravu a mazání dat z databáze.

[4] DQL (Data Query Language) je rodina jazyků pro výběr dat z databáze.

ciativního pole a vytvářejí tedy vždy dvojice klíč/hodnota. Výhodou těchto úložišť je vysoká rychlost operací zápisu a čtení podle klíče, obtížné jsou všechny operace s rozsahem klíčů (UPDATE ... WHERE), agregace atp. Ve většině těchto databází neexistují transakce, triggery, indexy ani integritní omezení. Obecně lze rozlišit dva hlavní typy databází klíč-hodnota:

1. úložiště uchovávající data v paměti RAM
2. úložiště uchovávající data na pevném disku

Databáze, které uchovávají data v paměti RAM, jsou nejčastěji využívány jako velmi rychlé cache nebo úložiště pro webové session. Příkladem těchto databází jsou *Redis*[5] nebo *memcached*[6].

Pro uchovávání persistentních dat ve formě klíč-hodnota jsou určeny databáze, jejichž obsah je uložen na pevném disku. Mezi tyto databáze patří např. *BigTable*[7] nebo *DynamoDB*[8] od společnosti Amazon.

- **Column store** ukládají data do skupin po sloupcích. Této architektury je využito často pro datové sklady nebo pro CRM systémy (*Customer Relationship Management*), kde je častou operací dotaz na množinu hodnot v atributu, nikoliv na záznam (řádek). Sloupcově orientované databáze jsou výrazně rychlejší v agregačních operacích, vložení záznamu je však oproti řádkově orientovaným databázím mnohonásobně pomalejší. Mezi reprezentanty lze řadit *Apache Cassandra*[9] nebo *HBase*[10].

- **Document store** databáze staví svůj datový model na pojmu „dokument“. Implementace dokumentu se napříč databázemi liší, ale koncepčně se jedná o strukturu zachycující formátovaná data s metadaty. Způsob reprezentace a uložení může být různý - XML, YAML, BSON (proprietální formát využívaný databází MongoDB založený na JSONu), existují i databáze uchovávající dokumenty v binární formě. Pro organizaci dat využívají dokumentové databáze kolekce (např. MongoDB), což jsou struktury podobné tabulkám, nebo využívají hierarchické uspořádání záznamů či tagování. Mezi typické představitele dokumentových databází patří *MongoDB*[11] a *CouchDB*[12].

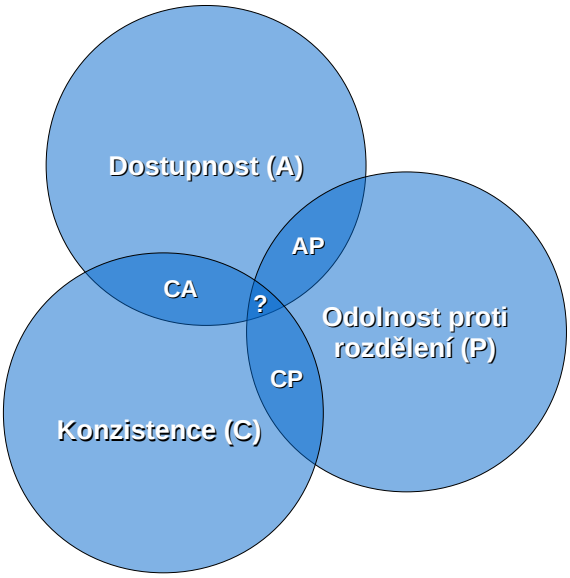
- **Grafové databáze** jsou primárně určeny pro data, která je obtížné popsat relační algebrou a jejichž vztahy mohou být vyjádřeny grafem. Typickým příkladem aplikace těchto databází jsou sociální sítě, kde uživatel je vrcholem a vztahy mezi uživateli lze popsat hranami. Dalším příkladem využití je popis topologie počítačových a inženýrských sítí. Jako nevýhodu grafových databází lze vnímat jejich horší škálovatelnost. Mezi nejrozšířenější databáze tohoto druhu patří *Neo4j*[13] a *HypergraphDB*[14].

[5] http://redis.io/
[6] http://memcached.org/
[7] http://research.google.com/archive/bigtable.html
[8] http://aws.amazon.com/dynamodb/
[9] http://cassandra.apache.org/
[10] http://hbase.apache.org/
[11] http://www.mongodb.org/
[12] http://couchdb.apache.org/
[13] http://neo4j.org/
[14] http://www.hypergraphdb.org/index

- **Objektové databáze** jsou konceptem ukládání dat ve formě instancí tříd (objektů) velmi blízko objektově orientovanému návrhu a programování. Tyto systémy často poskytují dědičnost přímo na úrovni databáze. Je třeba zmínit, že objektové databáze jsou většinou úzce svázané s použitým programovacím jazykem a kvůli použitému paradigmatu je není možné využít v součinnosti s jazyky, které toto paradigma nepodporují. Dnes existuje mnoho implementací objektových databází, z nichž nejpoužívanější jsou *ZODB (Zope Object Database)*[15], *ObjectDB*[16] nebo *Objectivity*[17].

Dalším velmi zajímavým pohledem na NoSQL databáze může být jejich zařazení z hlediska CAP teorému. CAP teorém (někdy označovaný také jako Brewerův teorém), který byl při poměrně striktních předpokladech dokázán [7], popisuje omezení mezi třemi základními vlastnostmi distribuovaného systému: konzistencí (*Consistency*), dostupností (*Availability*) a odolností proti rozdělení sítě (*Partition tolerance*). Stručně lze jeho význam popsat jako nemožnost dosáhnout všech tří vlastností současně v jednom systému. Většinu databází lze tedy zařadit pod dvojici písmen, které vyjadřují třídy **C** - silná konzistence, **A** - vysoká dostupnost dat a **P** - vysoká odolnost proti rozdělení sítě. Průniky těchto tříd jsou znázorněny na obrázku 2.1. Databáze lze tedy zařadit takto:

- **AP** - mezi tyto databáze patří ty, které poskytují pouze případnou konzistenci dat (případně tzv. *read-your-writes consistency*). Lze mezi ně zařadit *Riak, Apache Cassandra, CouchDB* a *DynamoDB*.
- **CP** - do této skupiny patří databáze, které obětovaly vysokou dostupnost výměnou za silnou konzistenci dat. Jejich příkladem může být *MongoDB*, *HBase* nebo *Redis*.
- **CA** - v této kategorii leží většina relačních databází (*PostgreSQL, MySQL* atp.).



Obrázek 2.1: Grafické znázornění CAP teorému. Kruhy značí diskutované třídy databází.

[15] http://www.zodb.org/
[16] http://www.objectdb.com/
[17] http://objectivity.com/

2.2 MongoDB

MongoDB je nerelační, platformově nezávislá, dokumentová databáze, vyvíjená společností *10gen*. Vývoj databáze započal v roce 2007, v roce 2009 vyšel první stabilní release. V současnosti je MongoDB jednou z nejoblíbenějších a nejužívanějších NoSQL databází. Komunita tento fakt většinou přisuzuje velmi snadné ovladatelnosti a některým shodným konceptům s SQL databázemi, což z MongoDB dělá ideální databázi pro přechod z SQL na NoSQL. Kritika MongoDB většinou poukazuje na velmi agresivní marketing a dříve nedostatečnou dokumentaci některých vlastností databáze. Jednou z kritizovaných vlastností byla nízká trvanlivost dat ve verzích, které nepodporovaly žurnálování [17].

2.2.1 Datový model

Z hlediska architektury MongoDB se jedná o dokumentovou databázi (angl. *Document store*) bez schématu. Základní datovou jednotkou je tedy *dokument*, který je ekvivalentem řádku tabulky v SQL databázích. Dokument je reprezentován záznamem (řetězcem) ve formátu BSON[18]. Záznam v notaci BSONu nesoucí data o osobě může vypadat následovně:

```
record = { _id  : ObjectId(``507f1f77bcf86cd799439016''),
           name : ``Stanislav Heller'',
           age  : 23 }
```

Z tohoto formátu vychází i nativní datové typy, které lze do MongoDB ukládat. Nejzajímavější z nich je tzv. „vnořený záznam“ (angl. *Embedded Document*), který umožňuje vložit do záznamu jiný strukturovaný záznam. Technika vkládání souvisejících záznamů do sebe (angl. *embedding*) je pak často využívána jako náhrada neexistující operace JOIN, kdy při vyšším stupni denormalizace databázového schématu jsou data již implicitně uložena ve spojené podobě a lze je z db získat pomocí jednoho dotazu bez nutnosti spojování klíčů na aplikační vrstvě. Vnořování záznamů je ale omezeno limitem maximální velikosti záznamu, který je ve výchozím stavu stanoven na 16MB (ve verzi 2.2).

Pro organizaci záznamů v databázi využívá MongoDB koncept kolekcí. *Kolekce* je množina záznamů, které (nejčastěji, ale ne nutně) spadají do téže entitní množiny. Kolekce nedefinuje žádné schéma (na rozdíl od tabulky v SQL databázích), a tedy žádná omezení na atributy záznamu a jejich hodnoty. Zde se nejvíce projevuje absence schématu v MongoDB: do libovolné kolekce lze uložit záznam s jakýmikoliv atributy a typy jejich hodnot. Této vlastnosti lze velmi dobře využít při evoluci databáze v průběhu času vývoje a údržby aplikace, v praxi je ale při zneužívání této vlastnosti velmi obtížné udržet databázi konzistentní. Protože kolekce a jejich schémata nejsou pevně definována, neexistuje ani DDL[19], který by zajišťoval jejich vytvoření a formát. Kolekce je automaticky vytvořena poté, co je do ní zapsán první záznam.

MongoDB poskytuje také speciální variantu kolekcí, tzv. *Capped collection*. Jsou to kolekce, které mají omezenou, uživatelsky definovanou velikost a vytvářejí kruhový buffer; záznamy se vkládají za sebe a pokud je kolekce plná, přemaže se nejstarší záznam. Výhodou tohoto druhu kolekcí je vysoký zápisový výkon a garance čtení dat bez indexu v pořadí podle vložení. Typickým případem užití může být logování nebo implementace velmi rychlé FIFO[20] cache typu klíč-hodnota.

[18] Binary JSON - varianta populárního formátu JSON, ve které lze uchovávat i binární data.

[19] Data Definition Language - v kontextu databází se jedná o jazyk pro definici databázového schématu.

[20] FIFO (First In, First Out) je způsob organizace a manipulace s daty v datové struktuře typu fronta, kde dříve vložený záznam vyjmut z fronty jako první.

2.2.2 Dotazování, agregace

Systém dotazování na data v MongoDB je značně odlišný od klasického dotazovacího jazyka SQL. Pro získání dat je použit vlastní dotazovací jazyk využívající notaci formátu JSON, ve kterém lze specifikovat dotazy na záznamy a jejich atributy včetně vnořených záznamů a seznamů. Dotaz má vždy následující schéma (použita notace jazyka Javascript):

```
db.collection.find([{specifikace filtru}][.metody řazení a omezení výběru])
```

MongoDB poskytuje velkou sadu dotazovacích operací, mezi něž patří projekce (výběr jen některých atributů ze záznamu), filtry (dotazování podle hodnoty atributu), regulární výrazy a dotazy typu `LIKE`, řazení a základní agregace. Neexistují zde však žádné spojovací operátory typu `JOIN` a jeho varianty.

Pro ilustraci způsobu dotazování lze použít následující příklad. Dotaz v jazyce SQL na tabulku osob `person`, kdy chceme získat jména osob, které vydělávají více než 30 000 Kč, seřazená podle platu vzestupně, by zněl například takto:

```
SELECT name FROM person WHERE salary > 30000 ORDER BY SALARY ASC
```

Tento dotaz by byl v notaci MongoDB zapsán takto:

```
db.person.find({salary: {$gt: 30000}}, {name: 1}).sort({salary: 1})
```

Z příkladu je patrný způsob zápisu dotazu; funkce `find` je parametrizována dvěma dokumenty - první specifikuje filtry a omezení, kde operátor `$gt` má funkci filtru „větší, než“, druhý dokument `{name: 1}` definuje projekci, tedy výběr pouze atributu `name`. Parametr řazení (funkce `sort`) obsahuje dokument `{salary: 1}`, kde jednička definuje vzestupné řazení.

Agregační operace jsou v MongoDB implementovány většinou pomocí map-reduce, jehož vlastnosti programátor specifikuje ve formátu JSON. Podrobná analýza Map-reduce v MongoDB však přesahuje rozsah této práce. Aby programátor nemusel pro každý dotaz na sumu či průměr z hodnot psát celý map-reduce, vytvořili vývojáři z *10gen* agregační framework, který zápis takových operací podstatně zjednodušuje. Některé operace (`count`, `distinct` a `group`) mají dokonce vlastní rozhraní (lze se přímo dotázat `db.collection.group(...)`).

Pro programátora, který přechází z SQL databází na MongoDB, může být matoucí sémantika operací `min` a `max` - ty zde nevyjadřují agregaci, resp. získání záznamu s minimální nebo maximální hodnotou daného atributu nad množinou dat, ale představují filtry, které jsou ekvivalentní s operátory `$lt` a `$gt`. Pokud aplikace vyžaduje operaci, jejíž chování je shodné s funkcemi `MIN` a `MAX` v SQL, lze využít seřazení hodnot atributu a výběr prvního prvku:

```
db.collection.find().sort({attr : 1}).limit(1)
```

2.2.3 Indexování

Na rozdíl od některých NoSQL databází, MongoDB umožňuje uživateli klást na data dotazy filtrující i podle atributů, které nejsou indexovány. Přestože je tento způsob poměrně pomalý (je nutný sekvenční průchod dat), umožňuje tak pokládání ad-hoc dotazů.

Uživatel může v MongoDB využít několik druhů indexů, které jsou implementovány jako B-stromy:

- *Primární index* - vždy definován nad primárním klíčem (atribut `_id`)

- *Sekundární indexy*:
  - jednoduchý index - lze definovat i nad vnořeným dokumentem nebo seznamem
  - složený index - záznam indexu se skládá z více atributů
  - geografický index - určen pro optimalizaci dotazů na 2D souřadnice

Indexům je možné definovat různé vlastnosti:

- unikátní - povoluje pouze jeden záznam s danou hodnotou atributu
- řídký - je definován pouze nad záznamem, který obsahuje indexovaný atribut
- TTL index - speciální index pro omezení doby platnosti záznamu v databázi

V případě, že je třeba vytvořit index nad velkou kolekcí dat (>100 MB), MongoDB umožňuje konstrukci indexu na pozadí (tzv. *Background index*). V případě tvorby indexu klasicky v popředí, je proces `mongod` (MongoDB démon) blokován tvorbou indexu a nelze se na databázi po tuto dobu dotazovat. Vytváření indexu na pozadí tento problém řeší, je však pomalejší. Vytváření indexu je vždy synchronní - nelze tedy zadat příkaz pro tvorbu indexu a asynchronně ve stejném procesu začít konat jinou práci - tvorba indexu blokuje proces do doby, než je index plně vytvořen.

2.2.4 Trvanlivost dat

Trvanlivost dat v MongoDB byla od počátku velmi diskutovaným tématem[21]. Od verze 2.0 má MongoDB implicitně nastavené žurnálování, tedy proces, kdy se záznam operace zapisuje do zvláštího souboru na disku a řízení je navráceno teprve poté, až je záznam o operaci uložen. To pak slouží pro obnovení dat při náhlém výpadku systému (například při výpadku dodávky el. proudu). Data ze žurnálu jsou periodicky zapisována do databázových souborů (délka periody je volitelná a lze nastavit). Žurnál je implementován jako *write-ahead log*.

Při vkládání dat má vývojář k dispozici čtyři možnosti, ze kterých může vybrat v závislosti na požadavcích na trvanlivost vkládaných dat:

1. data vložena do RAM a nepotvrzena (`w=0`)
2. data vložena do RAM a potvrzena (případně potvrzena na N replikách) (`w>0`)
3. data zapsána do žurnálu (`j=true`)
4. data zapsána do databázových souborů (`fsync=true`)

Tyto možnosti se od sebe samozřejmě liší úrovní trvanlivosti dat (poslední dvě jsou na stejné úrovni) a také rychlostí. Zatímco vložení dat pouze do RAM bez nutnosti potvrzení je velmi rychlé, s potvrzováním je přibližně 3x pomalejší, s žurnálováním až 500x pomalejší [10].

[21] Příkladem takové diskuze jsou komentáře k velmi zajímavému článeky Mikeala Rogerse porovnávající výkonnost a trvanlivost dat v MongoDB a CouchDB [17].

2.2.5 Transakce

MongoDB v současné verzi (2.2) neposkytuje možnost transakčního zpracování. Jedinou možností, jak na databázové vrstvě dosáhnout atomické, izolované a trvanlivé operace, je uložení všech transakčních dat do jednoho dokumentu a provádění operací nad ním.

Pokud je v aplikaci třeba použít transakce nad více než jedním dokumentem, je třeba na aplikační vrstvě implementovat dvoufázový commit [5]. Ten je však ve většině případů velmi pomalý (přidává 6 a více aditivních operací `insert` a `update` pro udržení transakčního kontextu) a měl by být využíván pouze v nutných případech.

Další možností, jak na aplikační vrstvě implementovat obecné transakční zpracování, by byla implementace MVCC[22], pro kterou by bylo v distribuovaném prostředí pravděpodobně třeba využít Oplog (tj. vyhrazená kolekce, která uchovává záznamy o operacích, které provedly změny obsahu databáze). Výkonnost databáze by se ale rapidně snížila a vyvstává otázka, zda není jednodušší místo složité implementace MVCC použít již relační databázi, která obsahuje plné transakční zpracování.

[22] Multiversion Concurrency Control - metoda kontroly souběžného přístupu založená na uchovávání starších verzí záznamů a aplikaci databázových operací na danou verzi záznamu.

Kapitola 3

Google App Engine

Google App Engine (GAE) je platforma pro vývoj a hostování webových aplikací. Jedná se o tzv. PaaS (Platform as a Service), což je druh IT služby založený na cloud computingu, který poskytuje jak hardwarovou a softwarovou platformu (servery, úložiště, OS), tak softwarové vybavení potřebné pro vývoj aplikací (prostředí programovacích jazyků, webové frameworky, API pro poskytované služby).

Google App Engine poskytuje webovým vývojářum plně vybavené prostředí s těmito vlastnostmi:

- dynamický webhosting, podpora WSGI[1] a CGI[2]
- podpora jazyků Python, Java a Go[3] včetně vybraných knihoven třetích stran
- plná podpora všech běžných webových technologií
- automatické škálování podle provozu a vyvážení zátěže
- perzistentní NoSQL a SQL úložiště
- množství velmi kvalitních API pro používání služeb Google (Gmail, Users, Images)
- plně vybavené SDK pro simulaci prostředí GAE na lokálním vývojovém stroji
- možnost spouštění dlouhých procesů pomocí front (task queue)
- možnost plánování a spouštění událostí v určitém čase nebo intervalu (cron jobs)

Výhody

Největší výhodou Google App Enginu je bezesporu vysoká škálovatelnost. Škálovat lze velmi jednoduše jak ve smyslu zátěže (výpočetní výkon versus počet obsloužených požadavků), ale také z hlediska velikosti uložených dat, kde Google uživatele nijak nelimituje (kromě ceny úložiště). Velkou konkurenční výhodou GAE je možnost bezplatného vyzkoušení služby – je možné tedy provozovat web zcela zdarma, ale po překročení předem definovaných volných kvót na využívání CPU času, úložišť atp. pak aplikace ztratí možnost používat daný zdroj do

[1] Web Server Gateway Interface je definice rozhraní, pomocí kterého webové aplikace v Pythonu komunikují s webovým serverem.

[2] Common Gateway Interface je metoda (protokol) pro komunikaci mezi webovým serverem a tzv. CGI skriptem, což je spustitelný program nebo skript napsaný v některém ze skriptovacích jazyků.

[3] Programovací jazyk vyvíjený od roku 2007 společností Google.

obnovení kvóty (což je ve většině případů jeden den). Touto politikou Google dokáže získat mnoho potenciálních zákazníků-programátorů, kteří vytvoří novou aplikaci, za jejíž provoz zpočátku neplatí a v případě, že je aplikace na trhu úspěšná, provoz vzroste a provozovatel musí začít platit, aby byla aplikace vždy dostupná a spolehlivá.

Nevýhody

Množství kvalitních služeb, které GAE poskytuje, může hrát v případě potřeby migrovat aplikaci do jiného prostředí (jiná cloudová služba, vlastní hosting) velmi negativní roli. Většina těchto služeb (memcache, url fetch, blobstore, task queue aj.) není dostupná mimo prostředí Google App Enginu, a tedy všechny funkcionality používající tyto služby je při migraci nutné přepsat. Google tímto přístupem může nepříznivě ovlivňovat projektové manažery, kteří mohou při volbě platformy zvolit raději otevřenější prostředí s jednodušší možností migrace, než velmi výkonné, ale proprietální služby od Googlu.

Příklady

Z úspěšných aplikací, které využívají Google App Engine, lze jmenovat např. web neziskové organizace *Khan Academy*[4], která poskytuje bezplatné vzdělávací on-line kurzy. Jako příklad úspěšně hostované webové stránky, na níž se projevila síla škálovatelnosti App Enginu, lze uvést *The Royal Wedding*[5] - web, který obsahoval podrobnosti o nedávné svatbě v britské královské rodině. V den svatebního ceremoniálu stránku navštívilo přes 5,6 mil. návštěvníků a v době největšího vytížení stránky atakovalo až 2000 požadavků za sekundu [15].

Konkurence

Google App Engine není na trhu s cloudovým hostingem jediným produktem. Konkurentem je například *Heroku*[6], který poskytuje prostředí pro Ruby, Javu a Python. V porovnání s GAE nezavádí velké množství omezení na to, co lze v rámci vyřizování HTTP požadavku provádět. Jako úložiště může sloužit PostgreSQL, případně Neo4j. Nevýhodou Heroku je chybějící administrátorské rozhraní a poměrně hrubý cenový rámec, ve kterém se pohybuje cena za provoz aplikace – GAE poskytuje podstatně jemnější granularitu.

Dalším konkurentem je *Amazon EC2*[7] patřící pod obecný souhrn služeb *Amazon Web Services* a poskytující velkou volnost v oblasti konfigurace, nastavení a volby programovacích jazyků. Jako možné úložiště je možné zvolit SimpleDB/DynamoDB, případně MySQL nebo Oracle jako relační databázi. Nevýhodou EC2 je nemožnost si službu vyzkoušet zdarma, jak je tomu v Google App Engine.

Společnost Microsoft nabízí v této oblasti produkt *Windows Azure*[8], který je zaměřen na vývojáře, kteří pracují v .NET technologiích, ale poskytuje také prostředí pro PHP a Javu. Operačním systémem na výpočetních instancích je Windows, jako úložiště zde Microsoft nabízí tzv. *Azure Table Storage* nebo vlastní relační databázi - *Azure SQL Server*. Nevýhodou je, stejně jako v případě Amazon EC2, nemožnost si službu vyzkoušet zdarma a také chybějící funkcionalita automatického škálování výpočetních zdrojů.

Poměrně novým produktem (2011) je *OpenShift*[9] od společnosti Red Hat. OpenShift

[4] http://www.khanacademy.org
[5] http://www.officialroyalwedding2011.org/
[6] http://www.heroku.com
[7] http://aws.amazon.com/ec2/
[8] http://www.windowsazure.com/
[9] https://openshift.redhat.com/

vytváří otevřenou platformu, kde veškeré služby lze použít i mimo cloud. Výpočetní jednotkou je tzv. *gear*, který je obdobou instance v GAE. Škálování výpočetního výkonu probíhá automaticky podle provozu na webové stránce. Platforma OpenShift poskytuje vývojářům značné množství programovacích jazyků (Java, Ruby, Python, PHP, Perl) a aplikační služby (tzv. *cartridge*), jako např. úložiště MongoDB, MySQL nebo PostgreSQL. Výhodou OpenShiftu je možnost si platformu vyzkoušet zdarma.

3.1 Architektura

Google App Engine lze konceptuálně rozdělit na tři části:

1. **HW a infrastruktura**: výpočetní jednotky, systém pro vyvážování zátěže, proxy cache
2. **SW vybavení instancí**: prostředí programovacích jazyků, frameworky a knihovny, rozhraní pro správu aplikace
3. **služby**: úložiště a služby (Google Services)

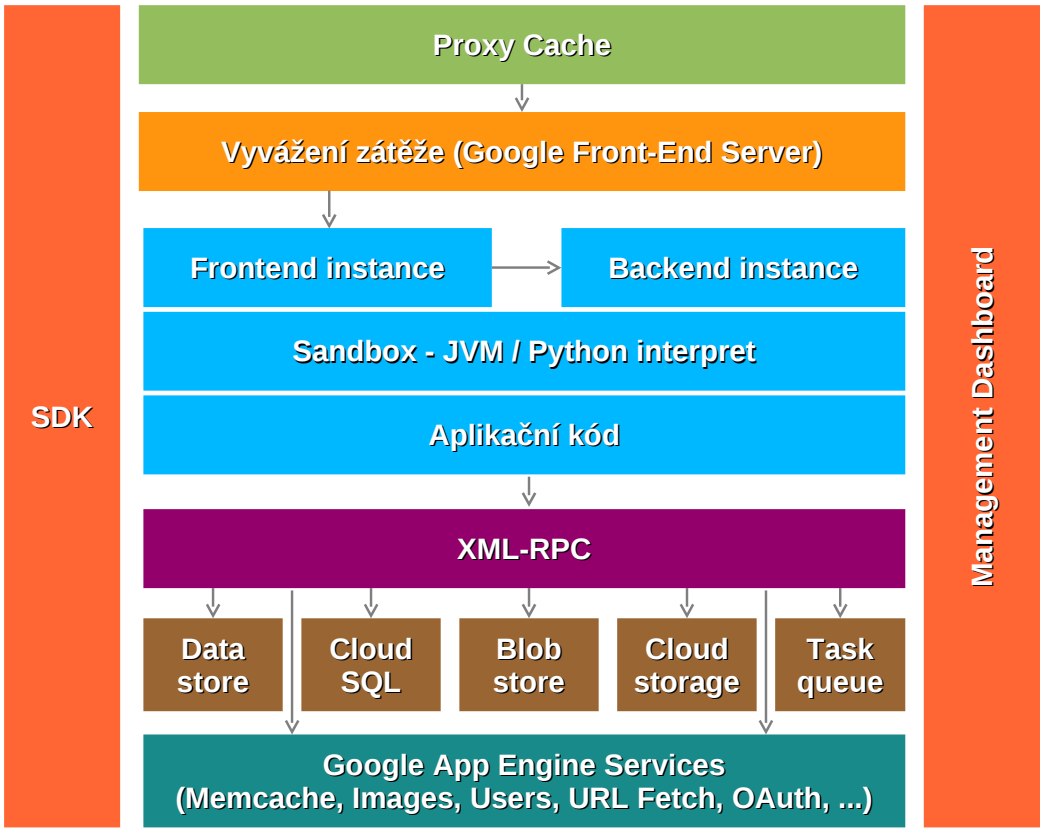
Podrobněji popisuje architekturu platformy App Engine schéma na obrázku 3.1. HTTP požadavky vždy nejprve prochází přes proxy cache, která je umístěna blízko uživatelům (je tedy předsazena před datacentra Googlu) a kterou lze využívat pro cachování statických stránek a dat. Pokud požadavek projde přes proxy cache, je směrován na některou z výpočetních jednotek (instance) podle jejich momentálního vytížení. K tomu slouží tzv. *Load Balancer*. Na každé instanci běží interpret Pythonu/JVM a uživatelská aplikace. Aplikační kód může využívat datových úložišť (Datastore, Blobstore, Cloud SQL) a služeb, ke kterým poskytuje App Engine velmi propracovaná API. S těmito úložišti a službami aplikace komunikuje pomocí XML-RPC volání.

3.1.1 Instance a škálování

Každá aplikace běží na tzv. *instancích*. Instance je jednotkou výpočetního výkonu, podle které Google pak počítá zdroje spotřebované aplikací, které se pak zohlední v účtované částce.

App Engine rozděluje instance na tzv. *front-endové* a *back-endové*. Front-endové instance jsou primárně určeny pro vyřizování HTTP požadavků. Google nabízí 3 typy různě výkonných instancí - začínají u 600MHz CPU se 128MB RAM a končí u 2,4GHz CPU s 512MB RAM. Back-endové instance jsou určeny pro déle trvající procesy. Opět je možné využít různě výkonných instancí: od 600MHz CPU se 128MB RAM do 4,8GHz CPU s 1024MB RAM.

Škálování výpočetního výkonu je automatické (na rozdíl např. od Windows Azure) - App Engine automaticky měří vytížení instancí a v případě, že dosáhne jisté meze, automaticky startuje nové instance, které slouží pro odbavení požadavků. Pokud vytížení instance po určité době klesne, GAE zajistí její odpojení, aby zákazník platil jen za čas, po kterou je instance zapojena do provozu. Uživatel samozřejmě může i ručně ovlivňovat počet běžících instancí.



Obrázek 3.1: Architektura platformy Google App Engine, převzato z [16] a rozšířeno.

3.1.2 Kvóty a limity

Obchodní filozofie App Enginu je založena na principu „plať jen za to, co používáš“ (pay only for what you use). Google poskytuje pro každou aplikaci volné denní kvóty, do kterých je možné zdroje (CPU čas, úložiště, linku, volání API) používat. Pokud uživatel nemá u Googlu založen účet a neplatí, aplikace po spotřebování volné kvóty nemůže daný zdroj využívat (což je v případě např. CPU času velmi zásadní nedostatek). Jakmile uživatel povolí placení, je inkasován pouze za zdroje, které využívá nad volnou kvótu. Tyto kvóty jsou pro většinu malých projektů dostatečně vysoké a při malém provozu na vytvořené webové stránce většinou nejsou přesaženy.

Aplikace na GAE je limitována také z hlediska možnosti použití některých programových prostředků a služeb operačního systému:

- nelze používat souborový systém (kvůli bezpečnosti)
- nelze používat sockety (pouze do verze SDK 1.7.6)
- nelze používat SSL certifikáty mimo doménu appspot.com
- nelze provozovat aplikaci na tzv. *naked domain* (doména bez prefixu www)
- v prostředí pro Python nelze používat knihovny, které jsou napsané v C (CPython)
- každý HTTP požadavek musí být vyřízen do 60 sekund, jinak je automaticky ukončen

Google se snaží mírnit dopady těchto omezení poskytnutím poměrně velkého množství služeb a vestavěných knihoven (např. lxml, jejíž části jsou napsány v C), pro déle trvající procesy poskytuje službu Task Queue, případně back-endové instance. Pro přístup k HTTP ze strany serveru lze využít místo socketů službu URL Fetch a jako náhradu souborového systému lze pro ukládání souborů (dokumenty, obrázky) použít úložiště Blobstore.

3.2 Produkční prostředí

App Engine podporuje ve svém produkčním prostředí (GAE Runtime) programovací jazyky Python, Java a také vlastní jazyk Go. V této práci je uvažován jako hlavní jazyk pro práci s GAE Python. Pokud není výslovně uvedeno jinak, veškeré konstrukce, případně informace týkající se GAE SDK se vztahují vždy k jazyku Python.

Webové aplikace je možné provozovat s využitím rozhraní WSGI (což je preferováno) a nebo pomocí CGI. Pokud aplikace komunikuje pomocí WSGI, je možné využívat vícevláknového zpracování dat (threading). Použití je však třeba dobře uvážit, protože threading se vyplatí pouze z hlediska rychlosti zpracování požadavku (spotřebovaný CPU čas je stejný, ne-li nepatrně vyšší) a to jen v aplikacích, které pro provoz používají velké množství vstupně-výstupních operací (komunikace s datastore či blobstore, URL fetch atp.).

3.2.1 Webové frameworky

Z webových frameworků má vývojář k dispozici nerelační mutaci[10] frameworku Django (tzv. *Django nonrel*) a také Googlem podporovaný webový framework *webapp2*. Webapp2 se řadí mezi tzv. microframeworky, které na rozdíl od full-stack frameworků (Django, web2py) obsahují pouze jádro[11] - neposkytují databázovou vrstvu, ORM[12] ani šablonovací systém a omezují se výhradně na vyřizování HTTP požadavků.

V prostředí Google App Engine lze provozovat i jiné než nativně podporované frameworky. Velmi dobrou podporu má na GAE také dnes populární *Flask*, *web2py* a méně známý *Pyramid*. Další možností je využití frameworku *tipfy*, který je založený na webapp (předchůdce webapp2), ale jeho vývoj byl pozastaven.

3.2.2 Knihovny

V Google App Engine verze 1.7.6 je v prostředí pro Python k dispozici velká většina vestavěných (builtin) knihoven. Mezi výjimky patří knihovny *socket* a *select*. Dále je v produkčním prostředí k dispozici mnoho knihoven třetích stran; pro parsování formátů XML, YAML a JSON lze používat knihovny *lxml, yaml a simplejson*. Pro vykreslování grafů poskytuje App Engine knihovnu *graphy*. Pro kodéry je velmi výhodné použít vestavěných knihoven *jinja2, markupsafe a prettytable*. Další knihovny obecného použití jsou - *grizzled, webob* a pro autentizaci *oauth2*.

[10] Django ORM je orientováno na relační (SQL) databáze. V mutaci, která je k dispozici na GAE, je ORM vyměněno vrstvu, která za pomoci velmi podobného rozhraní poskytuje přístup k nerelační databázi Google Datastore.

[11] Při pohledu na framework jako na implementaci MVC modelu většinou jádro microframeworku obshauje jen integrovaný controller a metody pro předání dat šablonám.

[12] ORM (Object-relational mapping) je technika pro mapování záznamů relační databáze na objekty a jejich atribut v objektově orientovaném programovacím jazyku.

3.2.3 Nasazení, administrace

Poté, co vývojář vytvoří novou aplikaci, je nutné ji nasadit do provozu. Proces nasazení je odlišný od klasických PHP hostingů, kdy se přes protokol FTP nahrají data a skripty na vzdálený server; na GAE pro tento účel existuje zvláštní skript (v linuxovém prostředí je to `appcfg.py`, který je součástí SDK). Po spuštění tohoto skriptu je nastartován proces nasazení, kdy se zkontroluje verze aplikace a změněné soubory se nahrají na GAE deployment server.

V produkčním prostředí může běžet více, než jedna verze aplikace. Za zmínku stojí fakt, že verze jedné aplikace mohou sdílet data v úložišti (Datastore), ale jednotlivé aplikace mezi sebou data sdílet nemohou.

Při procesu nasazení aplikace do provozu je většinou nutné do databází načíst data. Vývojaři většinou uloží základní data do lokálního úložiště v SDK a poté pomocí nástroje `bulkloader` nahrají data do vzdáleného úložiště u Googlu.

Pro každou aplikaci na GAE je k dispozici administrační rozhraní. V tomto rozhraní může administrátor/provozovatel stránek zjišťovat vytížení instancí, míru spotřebovaných zdrojů a statistiky úložišť. Lze zde také upravovat chování aplikace ve vztahu k používaným službám (nastavení počtu a typu běžících instancí, chování back-endových instancí, front a opakovaně spouštěných procesů atp.).

3.3 Poskytované služby (Google Services)

Velkou konkurenční výhodou App Enginu je množství velmi kvalitních služeb, které se označují souhrnným názvem *Google Services*. Mezi tyto služby patří: [4]

- **Channel** - posílání zpráv javascriptovým klientům přes persistentní připojení
- **Images** - ukládání obrázků a manipulace s grafickými daty
- **LogService** - logování událostí v rámci běhu aplikace na GAE
- **Mail** - programový přístup ke službě GMail
- **OAuth** - autentizace pomocí standardu OAuth
- **Fulltext Search** - implementace fulltextového vyhledávání v dokumentech uložených v Datastore (implementace je však velmi jednoduchá, rozhodně nedosahuje kvalit Google Search)
- **URL Fetch** - velmi rychlý přístup k datům dostupným přes HTTP protokol (web)
- **Users** - autentizace uživatelů na základě údajů, které jsou uloženy u Google (přístup do GMailu atp.)
- **XMPP** - používání XMPP protokolu (Jabber)
- a další

Google počítá k službám také služby Blobstore, Task Queue a memcache, které jsou diskutovány níže.

3.3.1 Datová úložiště a cache

Google Datastore

Základním úložištěm, které je určeno pro strukturovaná aplikační data, je NoSQL Google Datastore. Velikost uložených dat v Datastore je omezena na 1MB/entitu, což určuje použitelnost Datastore jako úložiště pro metadata, případně pro persistenci stavu bussiness logiky aplikace. Tématu tohoto úložiště se podrobnější věnuje kapitola 3.4.

Blobstore

Blobstore je primárně určen pro nestrukturovaná (např. binární nebo textová) data. Každý záznam má svůj klíč, z hlediska architektury DB se jedná o úložiště typu klíč-hodnota. Uložená data nelze měnit (lze je pouze smazat), není zde žádný systém pro dotazování (kromě získání hodnoty podle klíče), neexistují zde transakce ani jiné vlastnosti typické pro relační databáze.

Záznam v blobstore není limitován hranicí 1MB/záznam, čímž se z Blobstore stává idální úložiště pro velké binární soubory. Nejčastějším případem použití je ukládání obrázků, videa, XML souborů a dokumentů obecně. Obzvláště výhodné je využití společně se službou *High Performance Image Serving System*, která ukládá obrázky do Blobstore a pomocí parametrů URL je možné velmi rychle tato data vyčítat a efektivně transformovat (otočení, ořezání, změna velikosti atp.).

Google Cloud SQL

Distribuovanou databázi Cloud SQL je možné využít jako úložiště pro strukturovaná data, podobně jako v Google Datastore. Cloud SQL je však relační databáze založená na MySQL; mohou ji tedy využít uživatelé, kteří si nechtějí zvykat na nová návrhová paradigmata při užití NoSQL databází. Existují však jistá omezení pro použití:

- velikost databáze na jedné instanci je maximálně 10GB
- nejsou podporovány uživatelské funkce
- nelze využít replikace na úrovni MySQL
- a jiné

Google Cloud Storage

Cloud Storage, jak lze již z názvu usoudit, není databází, ale jednoduchým úložištěm, které je určeno pro ukládání velkého množství dat (řádově terabyty), ke kterým pak aplikace přistupuje přes REST[13] rozhraní. Typicky zde lze ukládat data z logů, kdy můžeme s výhodou využít dalších služeb od Google (*Big Query*, *Prediction API*). Dalším případem užití je úložiště pro multimediální data, kdy aplikace využijí možnosti přerušitelného stahování nebo čtení určité části souboru. Cloud Storage poskytuje pro přístup k datům nastavitelná práva na úrovni osob, projektů a skupin (OAuth 2.0).

[13] REpresentational State Transfer - architektura rozhraní pro přístup ke zdrojům v distribuovaném prostředí přes protokol HTTP.

Memcache

Služba Memcache je inspirována původní technologií *memcached*, která byla vyvinuta společností Danga Interactive pro portál LiveJournal[14]. Jedná se o distribuovanou asociativní paměť, která udržuje data v RAM memcache serverů, které jsou s produkčními instancemi propojeny vysokorychlostí linkou.

Použití memcache je přímočaré: uživatel ukládá dvojice klíč-hodnota, ke kterým může přiřadit časový údaj o předpokládané expiraci záznamu. Tento údaj však není závazný z hlediska trvanlivosti dat v cache; systém pro správu a řízení memcache obsahuje funkcionalitu pro automatickou realokaci zdrojů. Tento systém odstraňuje z cache staré a nepoužívané záznamy (LRU[15]), pokud objem dat uložený danou aplikací přesáhne stanovenou mez. Bohužel, Google neposkytuje přesné informace o použitém algoritmu a o stanovených limitech.

3.3.2 Zpracování dat

Protože prostředí Google App Engine uvaluje na každý HTTP požadavek limit 60 sekund pro jeho vyřízení, existují mezi nabízenými službami také různé druhy jiných výpočetních jednotek a mechanismů:

- **Task Queue** (fronta úloh) je hlaví strukturou pro ukládání záznamů o procesech, které mají být spuštěny. Fronta limituje délku běhu procesu na 10 minut, což je dostačující pro většinu běžných operací ve webovém prostředí.
- **Scheduled Jobs** jsou marketingovým názvem pro procesy naplánované v programu *cron*, které jsou spouštěny periodicky v předem zadaný čas.
- **Backendové instance** (viz kapitolu 3.1.1) - je možné využít pro náročné výpočty, jejichž očekávaná doba trvání výpočtu je delší, než 10 minut. V některých případech je možné proces rozdělit do několika podprocesů a pro výpočet využít několika front, ale pro rozsáhlé výpočty je výhodnější použít samostatnou výpočetní jednotku (instanci).
- **Map-reduce** - výpočetní framework, který je dnes standardem v distribuovaných systémech, je využíván pro náročné výpočty nad nerelačními databázemi.

3.4 Google App Engine Datastore

Datastore je obecný název pro základní nerelační databázi, která je určena pro ukládání dynamických aplikačních dat v GAE. Datastore se chová jako objektová[16] databáze bez schématu, poskytuje ACID[17] transakce a sekundární indexy. Pro dotazování zde existuje rozhraní, které však neposkytuje některé operace a vlastnosti, které jsou běžné v relačních databázích: nelze provádět spojovací operace typu JOIN, vyhledávání pomocí LIKE, neexistují zde uložené procedury, triggery a neexistují zde explicitní integritní omezení na úrovni db (kromě PRIMARY KEY).

[14] http://www.livejournal.com

[15] Least Recently Used - algoritmus, který odstraňuje z cache ty záznamy, které byly nejméně kvantitativně využívány.

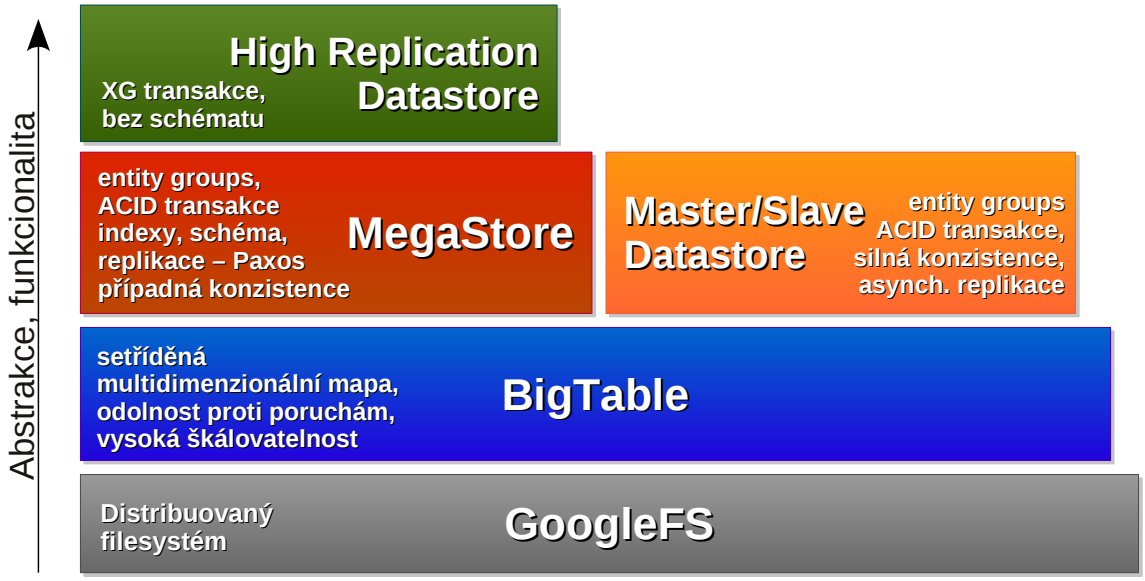
[16] Některé vlastnosti běžné u objektových databází jsou v Datastore simulovány až na úrovni knihovny *ndb*.

[17] Transakce splňující podmínky: Atomicity (atomičnost), Consistency (konzistence), Isolation (izolace), Durability (trvanlivost).

Implementace Datastore se v historii GAE měnila: v době spuštění (květen 2008) byl v provozu tzv. *Master/Slave Datastore*, což je úložiště, které se vyznačuje architekturou replikace dat, kdy jediné datové centrum obsahuje hlavní kopii (master copy) všech dat. Data zapisovaná do hlavní kopie byla pak asynchronně replikována do dalších (slave) datacenter. Tento přístup je nevýhodný z hlediska dostupnosti, protože zde existuje čas, po který jsou data nedostupná, ať už z důvodu údržby nebo neplánovaných poruch. Od května 2012 Google doporučuje místo Master/Slave Datastore používat novější úložiště s názvem **High Replication Datastore**. Architektuře a vlastnostem této databáze jsou věnovány další kapitoly v této sekci.

3.4.1 Architektura HRD

High Replication Datastore je hierarchicky vystavěn nad několika staršími technologiemi vyvinutými společností Google. Každá vrstva přidává nové funkcionality včetně vyšší úrovně abstrakce, jak je možné nahlédnout na schématu 3.2.



Obrázek 3.2: Hierarchická architektura High Replication Datastore. Pro srovnání je ve schématu obsažen i starší Master/Slave Datastore.

HRD je postaven na základech distribuovaného souborového systému GFS (*Google File System*). Nad tímto filesystémem je implementováno úložiště *BigTable*, které lze charakterizovat jako seřazené multidimenzionální asociativní pole [3, s.1]. Dimenzemi v tomto případě jsou klíč řádku, klíč sloupce a časový otisk. Tyto klíče pak jsou asociovány s hodnotou[18], která může mít jakýkoliv binární obsah:

`(row:string, column:string, time:int64)` → `string`

Je tedy možné velmi jednoduše uchovávat verzovaná data (díky časovému údaji), čehož také úložiště, které BigTable využívají jako nižší vrstvy, často využívají. Data jsou v BigTable

[18] V některých zdrojích, které se zabývají tematiku NoSQL databází, se BigTable zařazuje do oblasti úložišť typu klíč-hodnota.

uchovávána v souborech, které mají formát *SSTable* [3, s.3]. Podrobnější rozbor architektury a chování BigTable je již mimo rozsah popisné části této práce.

V roce 2011 zveřejnila společnost Google nový databázový systém *Megastore* založený na základech BigTable, který zachovává škálovatelnost NoSQL databází, ale přidává funkcionality SQL databází, které v BigTable chybí [2]. Megastore poskytuje ACID transakce v rámci databázových oddílů (partitions) a synchronní replikaci, která zaručuje odolnost proti poruchám a vysokou dostupnost za cenu vyšší zápisové latence, která se, podle Google, ve většině aplikací pohybuje v rozmezí 100-400 ms [2, s.232]. Megastore poskytuje transakce na úrovni entitní skupiny (každá entitní skupina má svůj WAL[19], do které jsou změny zapsány), pro atomické změny napříč entitními skupinami je možné využít dvoufázového commitu. Tyto transakce mají však vyšší latenci a obecně není doporučeno je používat.

High Replication Datastore má mnoho vlastností shodných s Megastore. Nejsou to však identické technologie; HRD využívá vlastností Megastore (např. schéma), ale implementuje odlišný datový model typu *entity-attribute-value*. Stojí za povšimnutí, že nad BigTable (bez schématu) je postaven Megastore se silně typovaným schématem a nad ním je vystavěn HRD, který je opět bez schématu. Data jsou v HRD replikována napříč datacentry synchronně; pro dosažení konsenzu mezi replikami je v HRD využíváno algoritmu Paxos [14]. Opakovanou aplikací tohoto algoritmu na vstupní data je zajištěno, že ve všech replikách jsou konzistentní data.

3.4.2 Datový model

Základní jednotkou informace je tzv. *entita*, která je v programovém rozhraní reprezentována objektem. Entita má svůj klíč a zpravidla několik atributů[20], každý atribut může nabývat hodnoty nebo několika hodnot v seznamu (tzv. *Repeated Property*). Každá entita patří nějaké entitní skupině (tzv. *entity group*), která je v Datastore základní jednotkou transakcionality a konzistence. Analogií ke vztahu entitní množina - entita je v relačních databázích vztah tabulka - řádek. Není však vhodné tyto pojmy zaměňovat, protože jejich význam, funkcionalita a implementace jsou odlišné. Entitní skupiny by měly uchovávat sémanticky podobné objekty, které v objektově orientovaném vyjádření spadají do stejné třídy - příkladem entitních skupin v kontextu modelu skladu zboží mohou být tyto: produkt, dodavatel, výrobce atp.

Zvláštností datového modelu úložiště HRD jsou tzv. předci (angl. *ancestors*). Každá entita může mít 0 - N předků. Klíč každé entity je složen z názvu entitní množiny, ke které přísluší, a z identifikátoru (většinou číselného). Pokud má entita předky, je jejím klíčem klíč jejího předka a její vlastní klíč. Díky této rekurzivní vlastnosti je možné při znalosti klíče entity získat kteréhokoliv jejího předka. Předci jsou využívání převážně při transakčním zpracování, protože entita je umístěna vždy v entitní množině nejvyššího předka (a tedy ve stejné lokalitě, resp. ve stejném tabletu BigTable).

3.4.3 Interakce s databází

Google App Engine poskytuje dva způsoby, jak se lze dotazovat v HRD:

[19] Write-Ahead Log - struktura, do které jsou zapisovány započaté transakce a která slouží jako žurnál. Některé databáze používají tohoto termínu (PostgreSQL, SQLite), jiné používají pro podobnou technologii název *Redo log* (Oracle).

[20] Lze vytvořit prázdnou entitu bez atributů, její informační hodnota je však nulová - využití takové entity je až na velmi ojedinělé případy nemožné.

- pomocí knihovního rozhraní (knihovny `ndb` nebo `db`)
- pomocí jazyka GQL (Google Query Language)

Knihovní rozhraní, které používá podobných principů, jako ORM/ODM[21] vrstvy, mapuje jednotlivé operace (filtry, řazení) na třídní metody, které ovlivňují tvar dotazu.

Druhou možností je použití jazyka GQL: část jazyka je podmnožinou SQL (neexistují zde však operace typu `JOIN`) a zavádí několik operátorů, které SQL nedefinuje - např. `ANCESTOR IS`. Ukázkový dotaz používající tento operátor může vypadat následovně[22]:

```
SELECT * WHERE ANCESTOR IS KEY('Person', 'Amy')
```

Tento dotaz navrátí všechny entity (bez ohledu na entitní množinu), které mají v klíči předka - osobu (`Person`) s klíčem `Amy`.

Pro programátory zvyklé na jazyk SQL je existence tohoto jazyka velkou výhodou. Nevýhodou je však mírné zpomalení operací, protože GQL se na straně knihovny parsuje a překládá na knihovní volání, resp. dále RPC volání.

Zápis dat je možný pouze pomocí knihovního rozhraní.

3.4.4 Konzistence

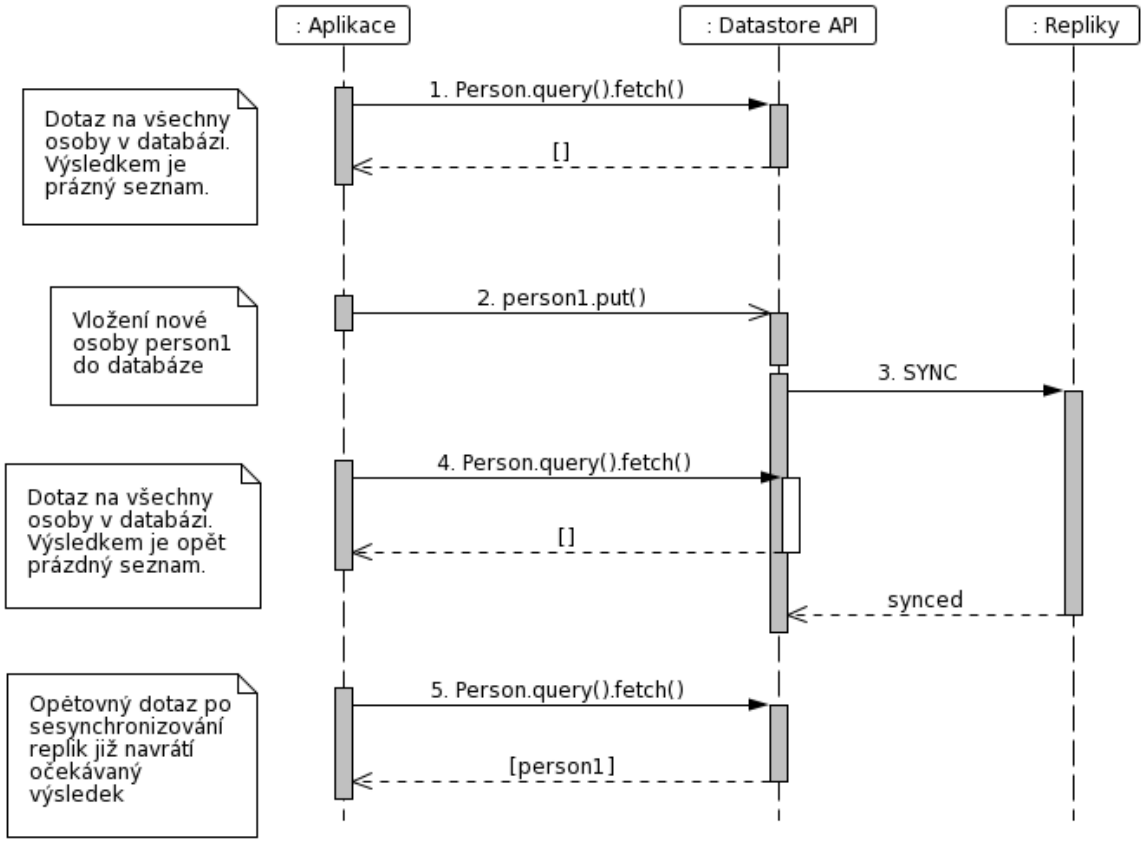
Z hlediska CAP teorému, který postihuje všechny distribuované systémy, zvolili vývojáři společnosti Google v HRD *dostupnost* a *odolnost proti rozdělení* na úkor *konzistence*, která musela být kvůli vysokému stupni replikace částečně degradována; HRD poskytuje pouze případnou konzistenci dat (angl. *eventual consistency*). Z hlediska rozdělení případné konzistence distribuovaných databází (Vogels, [18]) lze u chování HRD nalézt *Monotonic read consistency* a *Monotonic write consistency*, avšak nikoliv již *Read-your-writes consistency*.

Chování Datastore z hlediska případné konzistence ukazuje diagram 3.3; po uložení dat do datastore (na obrázku fáze 2) přes rozhraní, které poskytuje knihovna *ndb* (viz kap. 4.2), je započat mechanismus, kdy se data kopírují na všechny repliky (fáze 3). Protože tento proces může být poměrně časově náročný, řízení se vrací okamžitě po potvrzení ve všech replikách se data objeví až po určitém čase. Jestliže je okamžitě po uložení dat (resp. po navrácení řízení) knihovnou proveden dotaz, který by měl zahrnovat uložená data (fáze 4), většinou je směrován na jinou repliku, která o nově uložených datech neví a je tedy navrácen zastaralý výsledek. Po dokončení synchronizace replik (což může trvat v některých případech až 2 sekundy), by měl dotaz již poskytovat aktuální, konzistentní data (fáze 5).

Pro programátora zvyklého na silně konzistentní databáze může být toto chování velmi nepříjemné; při práci s HRD je třeba aplikaci vždy testovat v SDK s módem simulace případné konzistence pro odhalení chyb vzniklých návrhem, který s touto vlastností databáze nepočítá.

Jednou z možností, jak se vyhnout problémům kvůli případné konzistenci, je strukturování dat do větších celků, které jsou uloženy v jedné entitní skupině. Toho lze docílit buď tím, že jsou související entity uloženy přímo do nadřazené entity pomocí vnoření (*angl. embedding*) nebo pomocí skládání entit do stromu předků. Pak jsou všichni potomci entity, která je kořenem stromu, uloženi do jedné entitní skupiny, která je definována kořenem. Dotazování pak probíhá na bázi filtru podle předků (tzv. *ancestor queries*), které poskytuje v HRD již silně konzistentní výsledky. Nevýhodou tohoto řešení je však limit omezující

[21] Object Document Mapper - vrstva pro mapování dokumentů v nerelačních databázích na objekty.
[22] Příkad je převzat z https://developers.google.com/appengine/docs/python/datastore/gqlreference.



Obrázek 3.3: Sekvenční diagram popisující případnou konzistenci dat v HRD. Okamžitě po uložení nejsou data synchronizována mezi replikami a výsledek tedy obsahuje nekonzistentní data.

frekvenci zápisu do dané entitní skupiny - ten je nastaven přibližně 1 zápis za sekundu. Pokud však aplikace potřebuje častější zápisy včetně silné konzistence, je nutné využít jiných technik. Některé z nich popisuje přímo oficiální dokumentace Google App Engine [8]:

- vyžití memcache,
- cachování dat v cookie,
- uchovávání stavu v URL nebo
- při HTTP POST požadavku navracení nově uložené entity.

Cílem je většinou nalézt způsob cachování, které poskytne data uživateli, který právě upravuje nebo vkládá data do aplikace - v podstatě se jedná o způsob, jak dosáhnout úrovně *Read-your-writes consistency*.

3.4.5 Indexy

Google Datastore využívá pro všechny typy dotazů indexy - nelze položit dotaz, který by sekvenčně prohledával všechny položky. Pokud je explicitně definováno, že některý atribut entity nemá být indexován, není pak možné podle tohoto atributu vyhledávat nebo filtrovat.

Některé typy atributů (blob, dlouhý text atp.) jsou v HRD implicitně neindexované. Pokud je indexován řetězec, je jako klíč indexu brán celý obsah řetězce, do procesu není vložen žádný druh předzpracování jako při fulltextovém indexování (tokenizace, stemming atp.).

Každá entita obsahuje ve výchozím stavu záznamy v 3 indexech:

- `EntitiesByKind` slouží pro získání entit podle daného druhu (entitní množiny),
- `EntitiesByProperty_DESC` je určen pro získání entit podle hodnoty některého z atributů (z vlastností entity) seřazených sestupně,
- `EntitiesByProperty_ASC` je ekvivalent předchozího typu indexu, avšak seřazený vzestupně.

Programátor může dále definovat „uživatelské indexy“ (*user-defined*), které deklarativně popíše v konfiguračním souboru `index.yaml`. Lze vytvořit indexy i nad seznamy hodnot (tzn. pokud je atribut označen jako *repeated*) a složené indexy nad atributem a seznamem. Pokud je definován složený index nad dvěma seznamy, Datastore pravděpodobně odmítne položky indexovat a označí jej jako *exploding index* kvůli kombinatorické prostorové složitosti takového indexu.

3.4.6 Projekce

Mezi další velmi neobvyklé vlastnosti HRD lze zařadit přístup k databázové projekci - Google Datastore k projekci využívá pouze indexů. Největší rozdíl je u projekce atributů, které obsahují seznam hodnot. V tom případě Google Datastore navrací parciální entity (ochuzené objekty), které v projektovaném atributu obsahují jen jeden prvek z celého seznamu. Následující příklad ilustruje uvedenou situaci.

Do Datastore je uložena entita reprezentující produkt - kalhoty. V entitě je uložen také seznam barev, ze kterých si zákazník při koupi může vybrat.

```
e = Product(title='pants', colors=['white', 'grey', 'black'])
e.put()
```

Na Google Datastore je položen dotaz, který projektuje název a současně barvy (pro přehlednost je dotaz zapsán ve formátu GQL):

```
SELECT title,colors FROM Product WHERE title='pants'
```

Datastore pak navrátí tři parciální entity:

```
Product(title='pants', colors=['white'])
Product(title='pants', colors=['grey'])
Product(title='pants', colors=['black'])
```

3.4.7 Transakce

HRD poskytuje atomické, konzistentní a trvanlivé transakce v rámci jedné entitní skupiny na úrovni izolace, která je označována jako *serializable*[23]. V rámci jedné transakce všechna čtení a dotazy vidí jeden snímek (snapshot) dat, který odpovídá stavu při započetí transakce a trvá po celou dobu transakce.

[23] Nejvyšší úroveň izolace, kdy každá transakce je kompletně izolována. Transakce nad jednou entitní skupinou jsou prováděny sériově.

Uvnitř transakcí jsou povoleny dotazy pouze s definovaným předkem (tzv. *ancestor queries*) a každý dotaz musí být omezen pouze na jednu entitní skupinu. Transakce (resp. její zápisové operace) může zasahovat více entit, které mohou patřit maximálně k pěti entitním skupinám.

Datastore používá pro správu transakcí model *optimistické souběžnosti*: implicitně se předpokládá, že transakce mohou běžet paralelně, aniž by byl ovlivněn jejich výsledek. Pokud se dvě či více transakcí snaží souběžně upravit nebo vložit entitu do stejné entitní skupiny, první z transakcí (časově) data uloží a ostatní transakce selžou.

Transakce nad více entitními skupinami

Transakce nad entitami, které přísluší k různým entitním skupinám, je označována jako tzv. *cross-group* (XG) transakce. Transakce nad více entitními skupinami lze provádět pouze za předpokladu, že žádná jiná souběžná transakce neoperuje nad některou z entitních skupin, která je první transakcí ovlivněna. Podobně jako v transakcích nad jednou entitní množinou, není v XG transakci možné provádět jiné dotazy než ty s definovaným předkem.

Kapitola 4

Google App Engine SDK

Google poskytuje vývojářům SDK ve verzi pro Python a pro Javu. Mezi těmito verzemi jsou jisté odlišnosti (nejen v programovacím jazyku), které odráží stav služeb poskytovaných v GAE runtime. Některé služby jsou vyvíjeny nejdříve pro Python, a proto jsou přidány do SDK dříve než pro Javu.

Vývojový kit (SDK) pro GAE implementuje potřebné vybavení pro vývoj aplikací pro prostředí Google App Engine. Jeho základní architektura a obsah je naznačen na obrázku 4.1. SDK se skládá z těchto složek:

- databázové stuby,
- simulace služeb (service stuby) poskytovaných v GAE runtime (Blobstore, memcache, Task Queue, URL Fetch aj.),
- vybrané knihovny třetích stran,
- vývojový HTTP server,
- nástroje pro deployment aplikace do GAE Runtime (`appcfg`),
- testovací rozhraní (tzv. *testbed*) pro provoz service stubů mimo vývojový server (tzn. např. pro testování pomocí unit-testů).

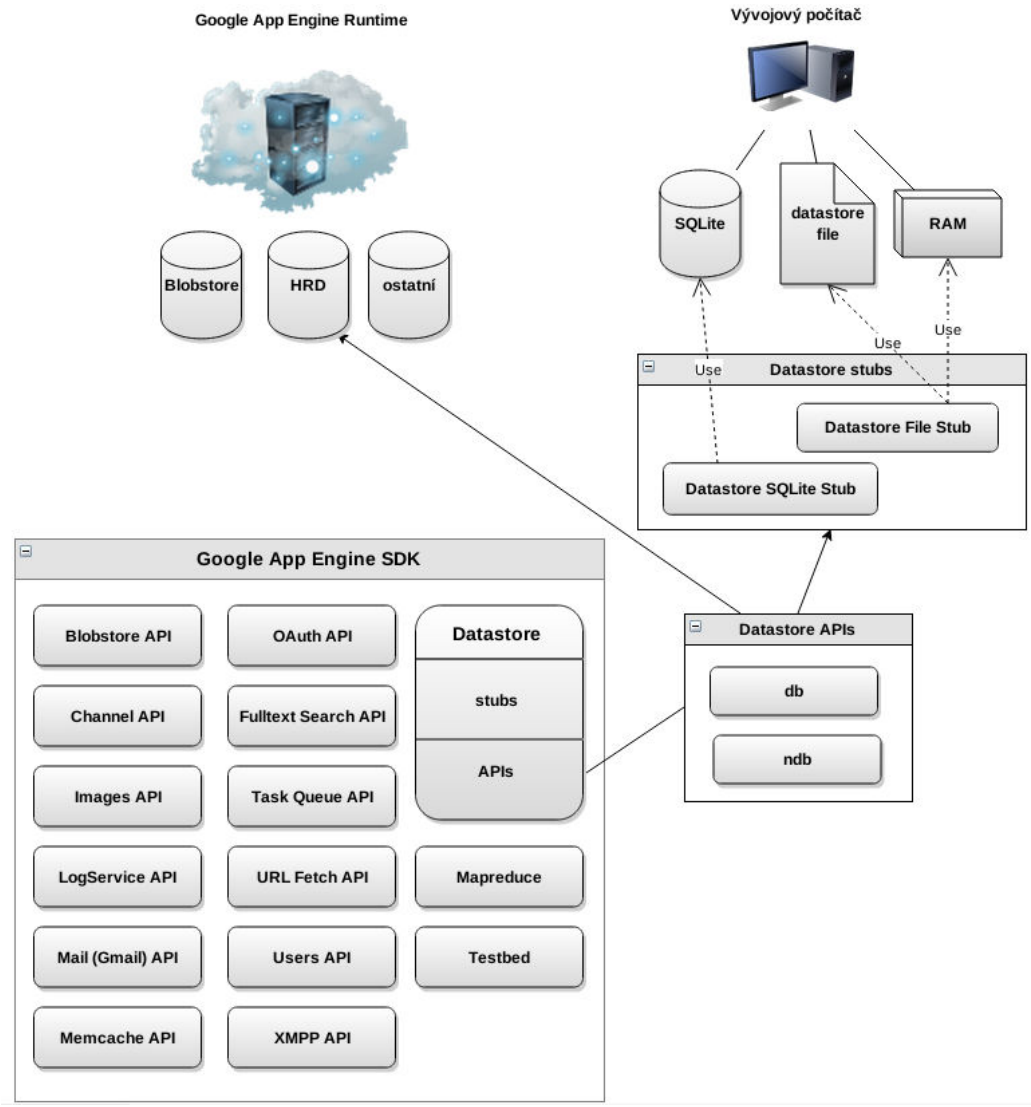
Všechny nástroje SDK využívají buď přímo vestavěných knihoven jazyka, knihoven třetích stran (např. PIL pro zpracování obrázků v implementaci stubu služby Google Images), nebo jsou implementovány přímo jako knihovna, která je daným stubem používána.

4.1 Problémy SDK

Z důvodu náročné implementace některých služeb, které GAE runtime nabízí, je práce s SDK mírně omezena, případně implementace nevytváří dokonalé prostředí, kde by byly veškeré vlastnosti služeb precizně simulovány. Při používání SDK se tedy lze setkat s následujícími problémy:

4.1.1 Rychlost služeb

Výkonnost a rychlost služeb ve vývojovém prostředí se v porovnání s GAE runtime často výrazně liší. Výkon vývojového stroje je na podstatně nižší úrovni než výkon, který poskytuje infrastruktura Google. V některých stubech jsou také opomenuty vlastnosti služeb,



Obrázek 4.1: Náhled na architekturu Google App Engine SDK (Python, verze 1.7.6) a použití datastore service stubů při lokálním vývoji a testování.

které souvisí s jejich distribuovaností a paralelismem. Nejzřetelnější je tento rozdíl na úrovni databází, kde reálnou rychlost prakticky nelze změřit jinak než v produkčním prostředí. Příkladem může být úprava dat v databázi: Google Datastore provádí úpravy indexů paralelně s úpravou/vložením entity do datastore. Tohoto efektu není možné dosáhnout na běžných pracovních stanicích s použitím poměrně jednoduchého software, jakým je SDK.

Je třeba vzít také v potaz celkovou odlišnost technologií a architektur, které jsou použity na straně SDK a v GAE runtime; databázové stuby využívají pro simulace NoSQL databází často relační databáze, jejichž vlastnosti jsou značně odlišné: kvůli MVCC nedokáží často poskytnout pro jednoduché operace potřebný výkon a naopak některé typy operací (např. selektivní update některých sloupců v řádku tabulky) nejsou v GAE Datastore vůbec možné.

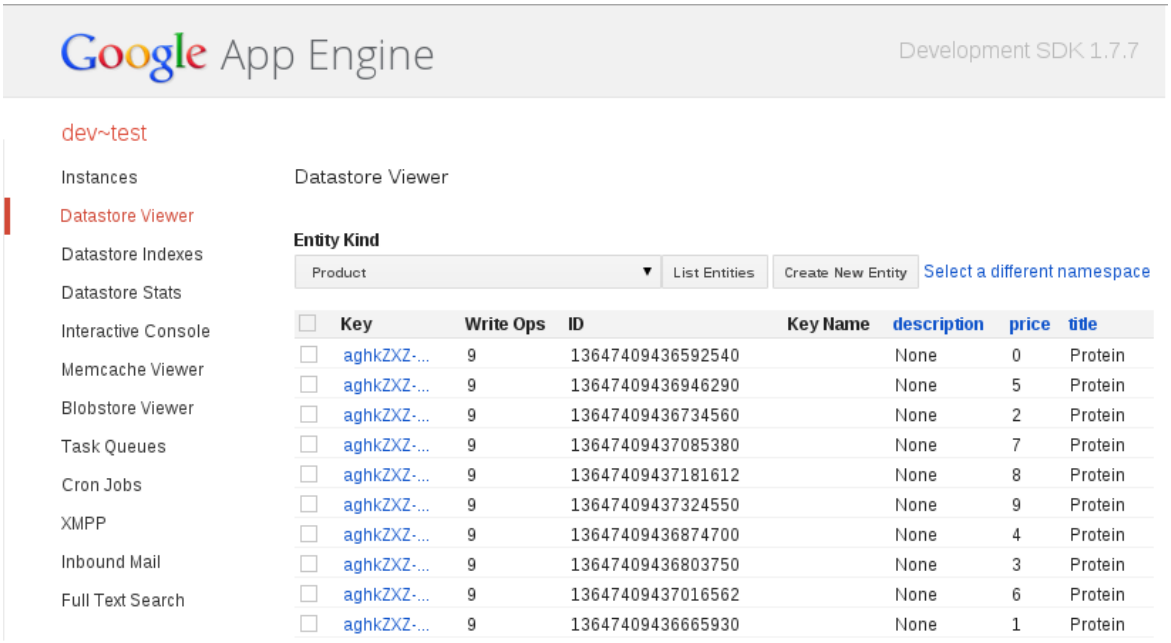
4.1.2 Jednovláknový vývojový server

Vývojový `dev_appserver` byl do vydání verze 1.7.6 (březen 2013) dalším nezanedbatelným problémem - načítání statických souborů zde probíhalo sériové, což je v případě webových stránek s většinou statického obsahu velmi pomalé. Limitující pak je také fakt, že pokud na serveru probíhá nějaká delší operace (vyřízení delšího HTTP požadavku), není možné spustit jinou stránku ze stejné aplikace. Jediným řešením tohoto problému je nastartovat další instanci serveru (na jiném portu), což značně zpomaluje vývoj a testování celého systému. Ve verzi 1.7.6 byl tento server označen za zastaralý a byl vytvořen nový `devappserver2`, který je architektonicky od základu změněn. Starý `dev_appserver` byl přejmenován na `old_dev_appserver` a v červenci 2013 bude ze SDK vyřazen.

Nový server startuje 3 samostatné procesy:

- server pro přístup ke službám (tzv. API server),
- server pro přístup k administračnímu rozhraní (tzv. admin server) a také
- vícevláknový webový server pro aplikaci.

V neposlední řadě také `devappserver2` obsahuje graficky modernizované, přehlednější administrační rozhraní, které je zobrazeno na obrázku 4.2. V tomto administračním rozhraní je kromě prohlížení obsahu databáze a databázových indexů možné také kontrolovat stav memcache a blobstore či vygenerovat statistiky uložených dat v datastore. Mezi velmi praktické nástroje administračního rozhraní patří interaktivní konzole, kde lze rychle interpretovat kód v jazyce Python, který využívá prostředí a služby SDK.



Obrázek 4.2: Webová administrace vývojového prostředí, která je tvořena v SDK 1.7.7.

4.1.3 Neexistující stuby

Přestože SDK je velmi dobře vybavené, existují služby, které nejsou v SDK zastoupeny. Neexistence příslušbého stubu je pak velmi omezujícím faktorem, pokud je na nich vystavěna

značná část funkcionality aplikace. Příkladem takové služby je Google Conversion[1], která je určena pro převod formátů dokumentů; mezi podporované formáty patří HTML, PDF, čistý text nebo obrázky včetně podpory OCR[2]. Kvůli značné složitosti systému, který by musel být v SDK implementován, a velkému výpočetnímu výkonu, který by byl potřebný pro takové zpracování dokumentů, nebylo API pro Google Conversion v SDK vůbec dostupné.

4.2 Knihovna ndb

Ndb je hlavní knihovnou pro přístup ke Google Datastore. Je k dispozici jak v SDK, tak v GAE Runtime a výhradně obstarává veškerý kontakt s tímto úložištěm - k úložišti nelze přistupovat přímo (pomocí XML-RPC volání), protože na této úrovni je komunikace poměrně složitá (a nezdokumentovaná). Knihovna automaticky převádí volání metod objektů modelu na sadu XML-RPC volání, obstarává transakční zpracování, zabezpečuje serializaci a deserializaci[3] strukturovaných dat a implementuje funkcionalitu triggerů na aplikační úrovni[4].

Ndb nahrazuje starší knihovnu *db*, která byla vytvořena původně pro starší *Master/Slave Datastore*. Knihovna db neposkytuje možnost asynchronní komunikace, neintegruje v sobě cache (ndb automaticky používá kontextovou cache pro ukládání dat během vyřizování HTTP požadavku a dále využívá memcache) a umožňuje vývojářům podstatně pohodlnější práci s transakcemi.

Google Datastore je databáze bez schématu; knihovna ndb však nad ní staví schéma na aplikační úrovni. Lze tedy tvořit databázové modely s přesně určenou strukturou včetně typů atributů. Vývojáři je ale umožněno používat i modely s volnějších schématem, kdy je možné nadefinovat jen určité atributy a některé je možné volně přidávat a odebírat. V ndb je možné využít vestavěné modely, které podporují dědičnost, což dává vývojáři možnost pracovat s Datastore jako s nativním objektovým úložištěm.

Na rozdíl od jiných ORM/ODM vrstev, indexy nejsou definovány na aplikační úrovni pomocí ndb, ale v konfiguračním souboru `index.yaml`. Transakce jsou definovány v ndb pomocí dekorátoru `@transactional` (Python), který je aplikován na funkci, která představuje jednotku transakčního zpracování.

4.3 Google App Engine Service Stuby

Service stuby byly vyvinuty za účelem poskytnout programátorům možnost testovat na lokálním stroji aplikace vyvinuté pro Google App Engine.

Service stub (dále jen stub) je komponenta SDK, která simuluje chování služby, kterou zastupuje. Poskytuje identické API a co nejvěrněji napodobené vlastnosti originálních Google služeb. Důraz je kladen převážně na správnost odpovědí stubu na kladené požadavky (tzn. stub a reálná služba musí dávat stejné výsledky). Některé druhotné vlastnosti služeb (rychlost, vnitřní struktura) není možné často nasimulovat tak, aby byly ekvivalentní. Vý-

[1] Google App Engine odstranil službu Conversion ze SDK a z runtime v listopadu 2012.

[2] OCR (Optical Character Recognition) je technika převodu textu v obrázcích do textu s metadaty o jeho velikosti a umístění.

[3] Metodou serializace dat v Google Datastore jsou tzv. *Protocol Buffers*, kde výsledkem je kompaktní binární formát, který je úspornější než XML. Google jej používá ve svém speciálním XML-RPC systému.

[4] Jedná se o tzv. pre-put, post-put, pre-delete a post-delete hooky, které jsou součástí základní modelové třídy ndb.Model.

vojáři Google se snažili o maximální pohodlí programátora, ale z časových důvodů nejsou některé stuby odladěné nebo jsou implementovány poměrně jednoduše.

V kontextu této práce jsou nejzajímavější stuby implementující chování Google Datastore. V současnosti existují dvě implementace: *Datastore File Stub* a *Datastore SQLite Stub*, jejichž vlastnostem jsou věnovány následující kapitoly. Pro srování zápisové rychlosti těchto stubů byl proveden test, jehož výsledky jsou vyneseny v grafu na obrázku 4.3. V testu byly porovnány tyto varianty stubů a jejich nastavení:

- Datastore File Stub s ukládáním dat na HDD (datová řada „File Stub HDD“)
- Datastore File Stub s ukládáním dat jen do paměti RAM (řada „File Stub RAM“)
- Datastore SQLite Stub s implicitním nastavením (datová řada „SQLite3“)
- Datastore SQLite Stub s vypnutou okamžitou synchronizací transakčních dat na disk - nastavení `PRAGMA synchronous=OFF` (datová řada „SQLite3 synch=OFF“)
- Datastore SQLite Stub s vypnutou okamžitou synchronizací transakčních dat na disk a ukládáním žurnálu pouze do paměti RAM - nastavení `PRAGMA synchronous=OFF` a `PRAGMA journal=MEM` (datová řada „SQLite synch=OFF + journal=MEM“)

Testovaný proces vložil do datastore vždy 10000 entit o průměrné velikosti 100 Bytů (což simuluje vložení většího počtu záznamů např. o produktech v databázi e-shopu, které obsahují název a metadata produktu), bez aditivních indexů. Parametry testovacího stroje: 1,2 GHz Intel Celeron, 80GB SSD disk, 16GB RAM, OS Linux - Ubuntu 12.04 LTS.

Výsledky tohoto testu jsou diskutovány v následujících kapitolách.

4.3.1 Datastore File Stub

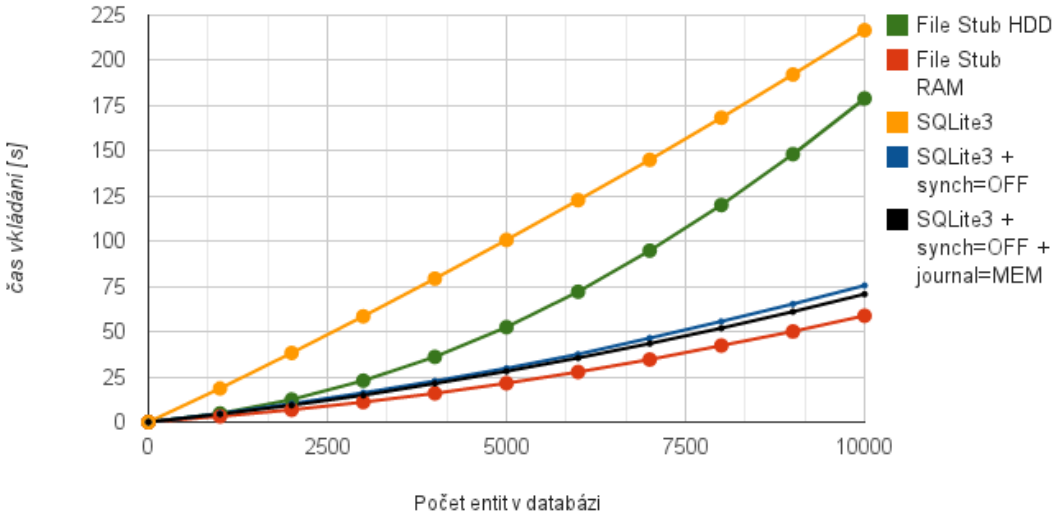
První implementací datastore stubu byl *Datastore File Stub*, který (navzdory svému názvu) ukládá všechna data do paměti RAM, ale lze je současně kopírovat i na pevný disk do souboru.

Velkou výhodou ukládání všech dat do RAM je vysoká rychlost zápisu a dotazování při malém objemu dat. Pokud je použito ukládání současně i na pevný disk, zápisová rychlost značně degraduje, jak je zřetelné na grafu v obrázku 4.3 - viz datové řady „File Stub RAM“ (na grafu červeně) vs. „File Stub HDD“ (na grafu zeleně). Další výhodou File stubu je fakt, že pro provoz není třeba žádných externích knihoven nebo databází.

Mezi nevýhody *Datastore File Stubu* patří nízký výkon pro větší objemy dat. V paměti RAM ani v databázovém souboru nejsou udržovány žádné druhy indexů - v případě filtrování dat je nutné vždy provést sekvenční prohledávání (SCAN) dat, což vede k lineární složitosti. Tento způsob je výhodný pro menší testovací data, ale v případě, že je nutné testovat nad větším objemem entit (řádově tisíce), stává se tento způsob velmi neefektivní a pomalý. Další značnou nevýhodou je nutnost při startu stubu vždy celý obsah databáze načíst ze souboru do paměti, což je zvláště při testování limitující.

4.3.2 Datastore SQLite Stub

Další implementací stubu pro simulaci úložiště Datastore je tzv. *Datstore SQLite Stub*, který zveřejnil v březnu 2010 Nick Johnson [12], jeden z někdejších hlavních vývojářů Google App Engine. Jako úložiště je zde použita volně šiřitelná databáze SQLite. Ta řeší problém množství dat v paměti RAM a vyhledávání v rozsáhlejších databázích rychlejší: třída



Obrázek 4.3: Test zápisové rychlosti Datastore File Stubu a Datastore SQLite stubu.

asymptotické časové složitosti většiny dotazovacích operací nad databází se díky použití indexů zlepšila z $O(N)$ (sekvenční vyhledávání) na $O(log_2 N)$. Pro malé datové sady je ale *Datastore File Stub* stále rychlejší jak při čtení, tak zápisu do databáze.

I přes chytřejší implementaci a použití databáze (byť relační) lze najít v SQLite stubu nedostatky. Rychlost vkládání entity je samozřejmě nižší než v případě Datastore File Stubu - je zřejmé, že při použití SQLite není možné dosáhnout zápisové rychlosti ekvivalentní s RAM, ale zpomalení není extrémní a pro potřeby testování je rychlost dostačující. Podstatně horší je situace při hromadném vkládání, kde se jako limitující faktor projevily transakce; není zde použita jedna transakce pro vložení všech dat, ale každá operace INSERT/UPDATE je obalena transakcí, což je velmi pomalé, jak lze vyčíst z grafu na obrázku 4.3 - datová řada SQLite3 (oranžově). Této vlastnosti lze jen stěží zabránit, protože je implementována hluboko ve stubu, ale je možné odladit SQLite tak, aby tato vlastnost nezpůsobovala takové problémy. Jednou z možností je nastavení `PRAGMA synchronous=OFF`, což sice zapříčiní, že trvanlivost transakce není zaručena (tzn. že po skončení transakce nejsou všechny změny zapsány na povrch disku), ale výkonnostní nárůst je obrovský (na obrázku 4.3 viz datovou řadu „SQLite3 + synch=OFF“ - modře). Chování a rychlost lze dále ladit pomocí nastavení velikost bufferů nebo pomocí udržování žurnálu v paměti RAM - toho lze docílit direktivou `PRAGMA journal=MEM`. Zde se nejedná již o tak značný nárůst výkonu - na grafu na obrázku 4.3 je toto nastavení reprezentováno datovou řadou „SQLite3 + synch=OFF + journal=MEM“.

4.4 Možnosti využití MongoDB pro nový stub

4.4.1 Aplikační rozhraní stubu

Jedním z úskalí, které je třeba při návrhu stubu překonat, je chybějící dokumentace programového rozharaní pro databázové stuby v GAE SDK. Proto je jedním z cílů této práce zdokumentovat programové rozhraní (API), které musí každý datastore stub v SDK implementovat, aby jej mohly využívat vyšší vrstvy SDK (např. knihovna ndb).

Největší problém při analýze rozhraní způsobuje již programovací jazyk Python. Zdro-

jový kód v tomto jazyce neobsahuje datové typy parametrů a návratových hodnot metod a funkcí - je dynamicky typovaný. Z toho plynou i velmi závažné důsledky pro případ zpětné tvorby API ze zdrojového kódu. Je možné sice zjistit názvy tříd a metod, počet parametrů a jejich názvy, ale je již velmi obtížné zjistit typ parametrů a návratových hodnot. Ten lze zjistit pouze experimentálně pomocí debuggeru.

Výsledkem analýzy je rozhraní, které je popsané v dodatku A. Rozhraní bylo zkoumáno na různých typech dotazů (jak z testovacího protředí, tak z vývojového serveru), ale i přesto nelze s jistotou určit, zda není v některých případech využito dynamičnosti jazyka a nejsou předávány parametry jiného typu, ale stejných vlastností (tzv. *duck typing*[5]).

4.4.2 Rozdíly mezi úložišti

Nerelační podstata a absence schématu databáze MongoDB je úložišti Google Datastore podstatně blíže, než např. SQLite. Přesto však existují mezi těmito databázemi značné rozdíly. Mezi ty nejzásadnější patří:

- MongoDB je dokumentová databáze, zatímto Google Datastore je databáze objektová. Datastore ukládá serializované objekty do BigTable pod klíč objektu - není zde tedy možný parciální update záznamu.
- MongoDB používá sadu datových typů, která vychází z jazyka JSON a kterou obohacuje o binární data, ObjectID, datum atp. Tato sada je z nemalé části odlišná od souboru typů v Google Datastore, který na úrovni *protocol bufferu* reprezentuje navíc geografické body, záznamy o uživatelích aj. Na druhou stranu Datastore neumí pracovat např. s vnořenými strukturovanými záznamy.
- MongoDB nepoužívá hierarchický klíč - neexistuje nativní způsob ukládání stromových struktur pomocí klíčů.
- MongoDB poskytuje možnost sekundárních indexů. Datastore je simuluje na úrovni BigTable (vytváří novou tabulku pro index).
- MongoDB neumožňuje indexování dvou seznamů současně - tzn. není možné vytvořit složený index nad dvěma atributy, jejichž hodnoty jsou pole. Google Datastore tvorbu tohoto indexu umožní, ale označí jej za tzv. *exploding index*.
- Chování databázové projekce v Google Datastore se kvůli použití indexů zásadně liší od chování MongoDB, které v tomto případě kopíruje vlastnosti SQL databází. Největší rozdíl lze nalézt u projekce atributu, který obsahuje seznam hodnot - zde Google Datastore navrací zvláštní parciální entity obsahující vždy pouze jeden prvek seznamu.
- MongoDB nepodporuje transakční zpracování. Google Datastore (resp. High Replication Datastore) poskytuje možnost transakcí i mezi entitními skupinami (tzv. *cross-group* transakce).

Velkou většinu těchto rozdílů bude muset stub vyrovnávat vhodným návrhem způsobu uložení dat a simulací správného chování na aplikační vrstvě tam, kde je není možno simulovat na úrovni databáze.

[5]Duck typing je označení způsobu programování v dynamickém jazyce, kdy není prováděna explicitní kontrola typu, ale postačuje fakt, že objekt dané třídy zná požadované metody a jejich protokol.

Kapitola 5

Návrh a implementace

Po analýze existujících stubů v SDK a programového vybavení nižších vrstev, které SDK obsahuje (viz dodatek A), byl navržen nový stub, který byl po vzoru *Datastore File Stubu* a *Datastore SQLite Stubu* pojmenován *Datastore MongoDB Stub*.

V následujících kapitolách budou popsány možnosti využití existujících komponent SDK pro návrh a tvorbu stubu, včetně způsobu řešení problémů plynoucích z nekompatibility mezi architekturami MongoDB a Google Datastore. Dále je uveden výčet použitých technologií, metody vývoje a výsledky optimalizace, při které byl zkoumán a zlepšen výkon výsledného řešení.

5.1 Použití datových struktur SDK

Každý service stub v GAE SDK je třída implementující metodu `MakeSyncCall`, pomocí které všechny knihovny vyšších vrstev se stubem komunikují. Takto komunikující třída je pak v kontextu SDK nazývána *službou* (*service*). Vlastní funkce stubu (volání služby) jsou mapovány na metody, které musí splňovat následující jmennou konvenci a parametry:

```
_Dynamic_<call>(request, response)
```

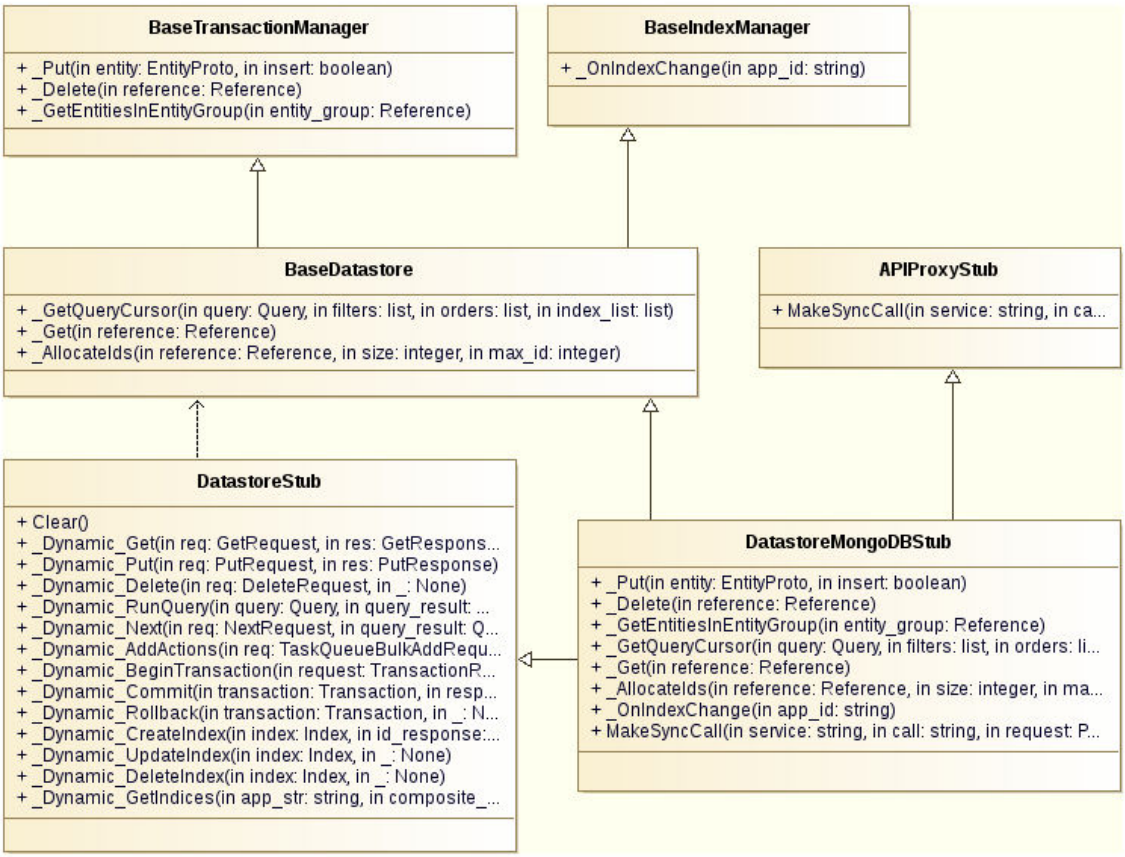
kde `<call>` je název volané funkce, `request` je vstupní parametr, který je povinný, a `response` je výstupní parametr, který často není využíván.

Nový datastore stub může implementovat přímo rozhraní[1] třídy `DatastoreStub`, kde však nastává problém s obsluhou transakcí, protože transakční vrstva je tvořena třídou `BaseTransactionManger`, která je na jiné úrovni v SDK a nezasahuje přímo do DatastoreStubu. Tento způsob byl použit pro implementaci prototypu, který měl potvrdit, zda je mapování mezi datovými formáty databází správně navrženo a netrpí nějakou závažnou chybou. Tomuto tématu se věnuje kapitola 5.2.

Druhou možností, která implicitně využívá již vytvořenou transakční vrstvu včetně simulace případné konzistence, je implementace rozhraní třídy `BaseDatastore` a podědění všech metod stubu od třídy `DatastoreStub`, která většinou požadavky validuje a deleguje

[1] V jazyce Python neexistují rozhraní ve smyslu protokolu, který musí daná třída implementovat (jako např. v jazycích Java nebo .NET). Rozhraním jsou zde myšleny metody, které dané abstraktní třídy neimplementují a dle konvence v Pythonu vyvolávají výjimku *NotImplementedError*. Přepisem těchto metod ve zděděné třídě je docíleno implementace daného „rozhraní“.

na nižší vrstvu (`BaseDatastore`). Tento způsob je sice méně přímočarý, vyžaduje implementaci speciálních kurzorů a dalších struktur pro simulaci chování Google Datastore, ale zato výsledný stub již plně zastupuje stávající implementace v SDK. Grafické znázornění tohoto řešení v podobě třídního diagramu je uvedeno na obrázku 5.1



Obrázek 5.1: UML diagram tříd[2] znázorňující stavbu Datastore MongoDB Stubu.

5.2 Mapování Google Datastore na MongoDB

Následující kapitoly se věnují způsobům řešení problémů, které vychází z odlišností v architektuře databází MongoDB a Google Datastore. Ve vybraných případech jsou uvedeny i alternativní přístupy, kde jsou diskutovány jejich výhody a nevýhody.

5.2.1 Reprezentace entit

Entita, která je základní datovou jednotkou v Google Datastore, je mapována na MongoDB dokument. Každá entitní sada (*entity group*) je uložena v kolekci s odpovídajícím jménem.

[2]Deklarace některých metod byly zkráceny kvůli množství parametrů. Cílem diagramu není plně dokumentovat programové rozhraní, ale ilustrovat architekturu stubu.

Klíč entity

Protože klíč je v Datastore hierarchický, mohou být v entitní sadě uloženy i entity jiného typu; entita je uložena vždy do té entitní sady, které odpovídá její nejzazší rodič.

Například pokud existuje klíč `(Child-123)`, spadá pod entitní skupinu *Child*. Pokud ale má dítě rodiče a jeho klíč je `(Parent-456, Child-123)`, pak je celá entita uložena v entitní skupině *Parent*.

V prvních verzích návrhu byl klíč entity v MongoDB reprezentován jako řetězec: pro jednoduchost byl klíč serializován do formátu, který lze popsat s použitím syntaxe regulárních výrazů takto:

```
key = (<GROUP>-<ID>)(-<GROUP>-<ID>)*
```

Serializovaný klíč z posledně uvedeného příkladu má tedy následlesledující tvar:

```
key = 'Parent-456-Child-123'
```

Tento způsob se později ukázal jako nevhodný z hlediska výkonnosti řešení. Proto byl navržen nový formát, využívající seznam klíčů, který je indexován:

```
key = [<GROUP>-<ID>(, <GROUP>-<ID>)*]
```

Tvar ukázkového klíče je tedy následující:

```
key = ['Parent-456', 'Child-123']
```

Nárust výkonnosti při použití tohoto řešení je podrobněji popsán v kapitole 5.4.

Atributy entity

Všechny názvy atributů objektů v Google Datastore jsou mapovány přímo na atributy dokumentů MongoDB s jednou výjimkou: pokud se v názvu atributu vyskytuje tečka, není tento název převoditelný do MongoDB, protože tečka je v MongoDB vyhrazena pro oddělení kolekce a atributu, případně atributu a vnořených atributů mezi sebou. Pokud například je názvem atributu `product.price`, musí být tečková notace přepsána tak, aby bylo možné atribut a jeho hodnotu uložit. Pro tento účel byla vybrána sekvence znaků `#!#`, která se v běžně užívaných názvech atributů nevyskytuje a která nahrazuje tečkový zápis. Při zpětném převodu je pak název atributu přepsán zpět na tečku.

Tento přepis je velmi důležitý kvůli knihovně `ndb`, která tečku používá pro přístup k tzv. strukturovaným atributům (*StructuredProperty*). Následující příklad blíže popisuje mapování strukturovaných atributů.

Mějme datový model pro adresář. Ten obsahuje seznam kontaktů, každý kontakt obsahuje jméno a má k dispozici 0-N kontaktních adres. Kontaktní adresa se skládá z názvu ulice a města.

Námět příkladu je převzat z oficiální dokumentace knihovny `ndb` [9]. S použitím knihovny `ndb` bude uvedený příklad modelován pomocí dvou tříd, seznam adres bude modelován s použitím *repeated properties* takto:

```
class Address(ndb.Model):
    street = ndb.StringProperty()
    city = ndb.StringProperty()

class Contact(ndb.Model):
    name = ndb.StringProperty()
    addresses = ndb.StructuredProperty(Address, repeated=True)
```

Příklad bude demonstrován na záznamu o kontaktu s více adresami, který je uložen do databáze:

```
guido = Contact(name='Guido',
                addresses=[Address(city='Amsterdam'),
                           Address(street='Spear St', city='SF')])

guido.put()
```

Knihovna `ndb` provádí speciální formu mapování těchto vnořených záznamů (adres) na paralelní seznamy. V Datastore bude tedy entita vypadat takto:

```
name = 'Guido'
addresses.street = [None, 'Spear St']
addresses.city = ['Amsterdam', 'SF']
```

Zde je zřetelná tečková notace, pod kterou jsou uloženy „sloupce“ vnořených entit. Do MongoDB je pak tato entita uložena jako dokument v následujícím formátu[3]:

```
{
   name: 'Guido',
   addresses#!#street: [None, 'Spear St'],
   addresses#!#city: ['Amsterdam', 'SF']
}
```

5.2.2 Převod mezi datovými typy

Vzhledem k faktu, že Google Datastore a MongoDB poskytují odlišnou sadu nativních datových typů, bylo třeba zavést mezi nimi mapování, aby bylo možné typy z datastore spolehlivě ukládat do MongoDB. Na toto mapování jsou ze strany Google Datastore (a jeho chování) kladeny tyto požadavky:

1. Mapování (zobrazení) musí být bijektivní: každý datový typ Datastore musí být reprezentován v MongoDB tak, aby byl možný zpětný překlad.
2. Mapování musí umožnit indexování na straně MongoDB a současně musí respektovat neindexovatelné typy podle Datastore (Blob, Text, EmbeddedEntity).

[3] Náhled je pro přehlednost zjednodušen. Ve skutečnosti jsou hodnotami atributů slovníky (vnořené dokumenty), které uchovávají informaci o typu.

3. Mapování musí respektovat specifika vyšších vrstev, které stub používají. Mezi tyto specifika patří např. reprezentace strukturovaného záznamu v knihovně `ndb` (GAE SDK).

Některé typy, které MongoDB poskytuje nativně, jsou ukládány ve svém původním formátu. Aby byly splněny všechny uvedené požadavky a bylo možné ukládat i nativně nepodporované typy, byl navržen strukturovaný formát hodnoty tak, aby u ní byl vždy obsažen i její typ:

```
{'t': <typ>, 'v': <hodnota>}
```

Toto schéma obsahuje zástupné symboly:

- **<typ>** - řetězec nesoucí název typu (v přehledové tabulce 5.1 leží ve sloupci „Název typu“). Tento název vychází z typu podle API Google Datastore.
- **<hodnota>**- reálná hodnota, kterou je možné vložit do MongoDB - tzn. vše, co je akceptováno formátem BSON (typ této hodnoty je v přehledové tabulce 5.1 ve sloupci „Typ reálné hodnoty v MongoDB“).

Atributy obsahující seznam hodnot není v MongoDB třeba serializovat, čímž bychom se připravili o možnost přímého indexování: lze využít vestavěného datového typu pole (`array`), jehož položky je možné indexovat nativně.

V následujících podkapitolách jsou diskutovány vybrané typy, pro které bylo třeba vytvořit zvláštní formát reprezentace. Celkový přehled mapování je pak uveden v tabulce 5.1.

Typ dle Datastore API	Název typu	Typ reálné hodnoty v MongoDB
datetime.datetime	`datetime`	string
datastore_types.Blob	`blob`	bson.Binary
datastore_types.GeoPt	`geo`	JSON
datastore_types.Key	`key`	JSON
datastore_types.BlobKey	`blobkey`	string
datastore_types.Text	`text`	string
datastore_types.EmbeddedEntity	`local`	bson.Binary
api.users.User	`user`	JSON
ostatní	nemá	stejná hodnota

Tabulka 5.1: Převod Datastore typů na odpovídající typy v MongoDB.

GeoPt je reprezentace geografického bodu. MongoDB nedefinuje pro tento datový typ zvláštní hodnotu, ale manuál MongoDB definuje způsob, jak tento typ reprezentovat [1]. Je možné použít seznam o dvou prvcích obsahující zeměpisnou délku (angl. *longitude*) a zeměpisnou šířku (angl. *latitude*):

```
geopt = [longitude, latitude]
```

nebo slovník mapující atribut `x` na zeměpisnou délku a `y` na zeměpisnou šířku:

```
geopt = {x: longitude, y: latitude}
```

Výhodou druhého způsobu reprezentace je fakt, že není nutné hlídat pořadí vkládaných hodnot a je tedy nižší šance chyby při manipulaci s geografickými body v aplikaci. Z tohoto důvodu byl pro implementaci stubu vybrán formát slovníku.

Key je datový typ pro klíč entity. Využívá se nejčastěji pro ukládání referencí na objekt (entitu). Formát pro ukládání je stejný jako v případě primárního klíče entity (viz kapitolu 5.2.1).

User je jedním ze speciálních složených datových typů v Google Datastore. Instance třídy `User` obsahuje atributy `email`, `federated_identity`, a `federated_provider` - všechny jsou datového typu `string`. Proto byl vybrán přímý způsob reprezentace v JSONu: dokument obsahující dané atributy a jejich hodnoty.

DateTime je typ pro reprezentaci data a času. MongoDB sice podporuje datový typ `Date` a `ISODate`, ten ale rozlišuje pouze milisekundy. Google Datastore pracuje s rozlišením v řádu mikrosekund. Proto je každé datum převedeno a uloženo do MongoDB jako řetězec ve formátu ISO8601 [11] s přídavkem 6 desetinných míst sekundy. Například datum 19.4.2013, 18:46:33 (včetně desetiného místa vteřin) je serializováno takto:

```
2013-04-19T18:46:33.517885
```

Formát ISO8601 je vytvořen tak, aby lexikografické řazení odpovídalo uspořádání datumů, takže je funkce `sort()` na straně MongoDB při porovnávání řetězců seřadí správně.

5.2.3 Dotazování, indexování a transakce

Google Datastore rozlišuje indexovatelné a neindexovatelné datové typy. Indexovatelné jsou implicitně všechny, ze kterých výjimku tvoří tyto:

- `Text` - rozsáhlá textová data,
- `Blob` - binární data,
- `EmbeddedEntity` - vnořená entita (v `ndb` je označena jako `LocalStructuredProperty`) s neindexovatelným obsahem.

Aby nebylo možné takové entity vůbec navrátit, má každý dotaz do MongoDB implicitní filtr ve tvaru `{$nin : ['blob', 'text', 'local']}`, který zaručuje, že se ve výsledcích nebudou vyskytovat žádné hodnoty těchto datových typů.

Filtrování a řazení

Mezi další odlišnosti databází MongoDB a Google Datastore patří chování, které vykazují při řazení navrácených položek podle daného atributu. Pokud daný atribut v záznamu neexistuje, pak:

- **MongoDB** záznam zahrne do výsledků,
- **Google Datastore** entitu vůbec nenavrátí.

Proto je třeba každý dotaz, ve kterém probíhá řazení, navíc vybavit filtrem `{$exists: 1}`, který zaručuje existenci atributu.

Projekce

Za jeden z největších rozdílů mezi MongoDB a Google Datastore lze považovat chování úložišť při vyhodnocování dotazů obsahujících projekci. Oproti MongoDB, jehož chování je velmi podobné SQL databázím a které navrací jednu entitu s projektovanými atributy, Datastore navrátí množství parciálních entit (více je tento princip popsán v kapitole 3.4.6).

Protože toto chování nelze simulovat na úrovni MongoDB, vše je simulováno na aplikační vrstvě prostým kopírováním entity a vkládáním jednotlivých variant do projektovaných atributů. Stejně tak filtrování hodnot těchto atributů není možné a musí být simulováno přímo ve stubu.

Transakce

Transakce jsou zajištěny na aplikační vrstvě v SDK. O simulaci transakcí se stará `BaseTransactionManager`, jehož funkcionalitu `DatastoreMongoDBStub` dědí. Není třeba tedy implementovat transakční vrstvu nad MongoDB pomocí dvoufázového commitu nebo jiných podobných technik.

5.3 Implementace

Vývoj *Datastore MongoDB Stubu* probíhal ve dvou fázích. V první fázi byl vytvořen prototyp, který implementoval pouze rozhraní `DatastoreStub`u bez transakční vrstvy. Na tomto prototypu byla ověřena funkčnost navrženého mapování typů na MongoDB a byly opraveny chyby plynoucí z neošetřených krajních stavů a kombinací hodnot a datových typů. Ve druhé fázi byl pak implementován finální stub, včetně transakční vrstvy a možnosti simulace případné konzistence.

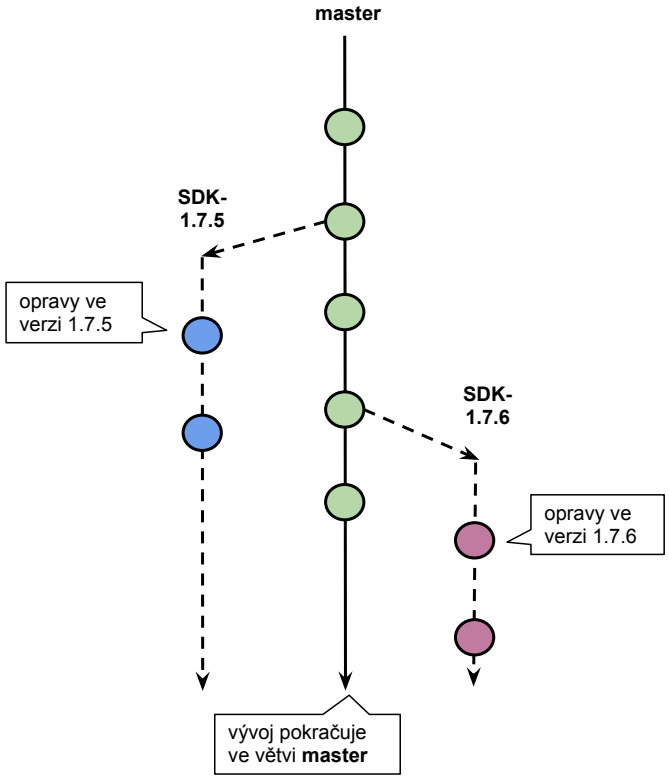
5.3.1 Sdílení zdrojových kódů

Protože cílem diplomové práce je vytvořit volně šiřitelný nástroj, který by využívala komunita Google App Engine, jsou veškeré zdrojové kódy umístěny ve veřejném DVCS[4] repozitáři GitHub[5]. Základní schéma repozitáře je následující:

- ve větvi **master** jsou implementována veškerá vylepšení a postupná implementace nových vlastností pro poslední verze SDK,
- každá verze má svoji větev, která je pojmenována ve tvaru **SDK-<verze>**. Verze stubu odpovídající verzi SDK nejsou opatřeny štítkem[6], ale mají své větve, aby bylo možné je zpětně spravovat a opravovat chyby.

Toto schéma je zobrazeno i na obrázku 5.2. Každý commit je znázorněn barevným kruhem, větve jsou reprezentovány vertikální čarou.

[4] Distributed Version Control System - systém pro správu verzí, jehož repozitáře jsou umístěny v cloudu.
[5] https://github.com/hellerstanislav/appengine-datastore-mongodb-stub
[6] Štítkem je myšlen git tag.



Obrázek 5.2: Schéma git repozitáře pro Datastore MongoDB Stub.

5.3.2 Vývoj

V průběhu implementace bylo třeba držet krok s rychle se vyvíjejícím SDK, který je vydáván jednou za měsíc. Nejzásadněji se změny projevují v serverové části, kde je třeba pro každou verzi vytvořit nový integrační patch (viz kapitolu 7.1). Funkcionalita stubů se naštěstí nemění tak rychle, nejčastěji jsou opravovány chyby na vyšších vrstvách, tedy např. v knihovně `ndb`.

V polovině března 2013 byla vydána nová verze GAE SDK 1.7.6, kde byl původní jednovláknový vývojový webserver `dev_appserver` nahrazen novější vícevláknovou implementací se jménem `devappserver2`. Tento fakt značně zkomplikoval vývoj nového stubu, protože veškeré implementované funkcionality stubu musely být testovány i nad novým serverem, do něhož musel stub být integrován kvůli jeho rychle narůstající popularitě.

5.4 Optimalizace

Optimalizace výkonnosti stubu probíhala za pomocí profilovacího nástroje `cProfile`, který je integrován do standardní knihovny jazyka Python.

5.4.1 Ancestor query

Jak již bylo v kapitole 5.2.1 zmíněno, klíč entity v MongoDB byl před optimalizací reprezentován jako řetězec, který propojoval skupiny a klíče pomocí pomlček.

Po provedení testů rychlosti a zkoumání výsledků profilování se tento způsob ukázal být z hlediska výkonnosti velmi nevhodný. Veškeré dotazy, které filtrovaly entity podle předka (tzv. *ancestor queries*), musely probíhat pomocí regulárních výrazů, které jsou poměrně pomalé. Proto byl navržen nový formát, který umožňuje indexování jednotlivých částí klíče a tedy rychlejší dotazování na předky dané entity. Jednotlivé dvojice (entitní skupina, id) jsou uloženy do seznamu, jehož položky jsou indexovatelné:

```
key = [<GROUP>-<ID>(, <GROUP>-<ID>)*]
```

Výsledný formát primárního kliče každého záznamu (každé entity) v MongoDB je tedy:

```
_id: {'dskey': ['Parent-456', 'Child-123', ...]}
```

Celý klíč je obalen slovníkem, protože MongoDB neumožňuje, aby primárním klíčem bylo pole. Do parametru `_id` je však možné umístit slovník, který již může mít toto pole jako jednu z hodnot a MongoDB zaručuje možnost jeho indexování.

Výsledné zrychlení bylo u všech typů dotazů až dvojnásobné, protože vyšší vrstvy nad stubem si ukládají pomocná metadata a volají mnoho vedlejších dotazů, které byly kvůli regulárním výrazům zpomaleny.

5.4.2 Cache

Dalším výkonnostním problémem, který značně zpomaloval dotazování, bylo velké množství volání metody `_GetEntitiesInEntityGroup`, která navrací všechny prvky z entitní skupiny, což je poměrně náročná operace. Tento dotaz je volán opět vyšší vrstvou SDK a nelze jej (kromě implementace) nijak ovlivnit. Proto byla vytvořena cache, která uchovává výsledky této metody v paměti RAM.

Princip cache je odvozen od *Datastore File Stubu*, který je na funkcionalitě této cache postaven. Entity jsou zde mapovány do dvouúrovňového slovníku, který mapuje entitní skupiny a klíče na entity takto:

```
<entity group> → <entity key> → <entity>
```

Pokud je pak výsledek v cache nalezen, není položen dotaz na databázi, což výrazně urychluje činnost stubu. Cache je samozřejmě invalidována při smazání nebo přepisu entity.

Kapitola 6

Testování

Všechny testy probíhaly na HW a SW v následující konfiguraci:

- CPU: Intel Pentium B980 2,4 GHz (2 jádra, 64bit),
- RAM: 4GB DDR3 1600 MHz, HDD: 500GB 7200 ot./min.,
- OS: Fedora 17, Linux 3.6.11-5.fc17.x86_64 (64bit)
- MongoDB 2.2.3 (64bit), testováno i na verzi 2.0.9,
- Python 2.7.3 [GCC 4.7.0 20120507 (Red Hat 4.7.0-5)] (64bit),
- pymongo >= 2.0 (testováno na verzích 2.0, 2.1, 2.2, 2.3 a 2.4)

6.1 Testy funkcionality

Celý vývoj probíhal metodikou TDD - test driven development, která byla pro účel tvorby nového stubu velmi vhodná. Vždy byla tedy napsána nejprve sada unit testů pro danou vlastnost stubu a teprve poté byly inkrementálně implementovány nové funkcionality. Pro implementaci testů byla použita knihovna `unittest`.

Správnost testů (tedy zda testují správné chování HRD) je zajištěna tím, že každý test probíhá pokaždé jak nad Datastore File Stubem, který je zvolen jako referenční implementace stubu, tak nad Datastore MongoDB Stubem, jehož vlastnosti jsou testovány. Tento způsob testování je mírně pomalejší, ale zajišťuje okamžité zjištění regrese při vývoji a včasné řešení problematických částí systému.

Celkově bylo implementováno 75 unit testů, z čehož využíváno je pouze 74. Jeden z testů obsahuje kontrolu funkcionality, která není ve stubu zahrnuta (projekce nad dvěma atributy, jejichž hodnoty jsou seznamy - tohoto chování je velmi obtížné dosáhnout, protože MongoDB nepodporuje filtrování podle dvou seznamů). Proto je tento test přeskakován.

V konečné formě trvá celá sada testů 16 sekund. Nejdelší čas stráví testovací skript při kontrole chování krajních (velmi vysokých) hodnot limitu a offsetu. Tento druh dotazů je jak u *Datastore File Stubu*, tak i u *Datastore MongoDB Stubu* velmi pomalý. Je samozřejmé, že ve výsledném řešení byly všechny testy napříč testovanými konfiguracemi SW úspěšné.

Dále byla implementována sada testů, která testuje funkčnost integrace stubu do SDK, zvláště do testovacího nástroje `testbed`.

6.2 Testy rychlosti

Rychlostní testy probíhaly vždy nad náhodně vygenerovanou datovou sadou. Tato sada obsahovala při každém testu entity, jejichž datový model lze v `ndb` popsat takto:

```
class Inner(ndb.Model):
    z~= ndb.FloatProperty()

class Entity(ndb.Model):
    x = ndb.StringProperty()
    y = ndb.IntegerProperty()
    d = ndb.DateTimeProperty()
    i = ndb.StructuredProperty(Inner)
```

Jedná se tedy o entity s jedním řetězcem, který obsahoval 100 tisknutelných znaků, náhodný integer (uniformní rozložení), aktuální datum a vnořenou entitu, která obsahovala jeden atribut - náhodný float (opět uniformní rozložení).

Pro testování byl vytvořen skript, který provede testy rychlosti vkládání dat a dotazování v deseti na sobě nezávislých sériích pro různě velkou datovou sadu - od 1000 do 10000 vložených entit. Testy proběhly vždy s vyprázdněním paměti RAM MongoDB (`mongod` démon byl vždy restartován). Výsledky těchto sérií byly zprůměrovány (aritmeticky). Stejně, jako tomu bylo i v předchozích testech při porovnávání vlastností existujících stubů, byl SQLite stub optimalizován pro co nejvyšší výkon (zákaz synchronizace transakcí na disk, bez žurnálování).

Pro měření délky dotazů a operací nad databázovými stuby bylo využito vestavěné funkce `time.time()`, která navrací aktuální UNIXový čas takto:

```
t_start = time.time()
result = Entity.query(...)
t_total = time.time() - t_start
```

6.2.1 Zápis dat

Při prvním měření byl porovnáván zápisový výkon stubů při větší datové sadě. Vkládání dat do MongoDB probíhalo bez čekání na zprávu o případné chybě (`safe=false`) a bez čekání na zápis do žurnálu (`j=false`).

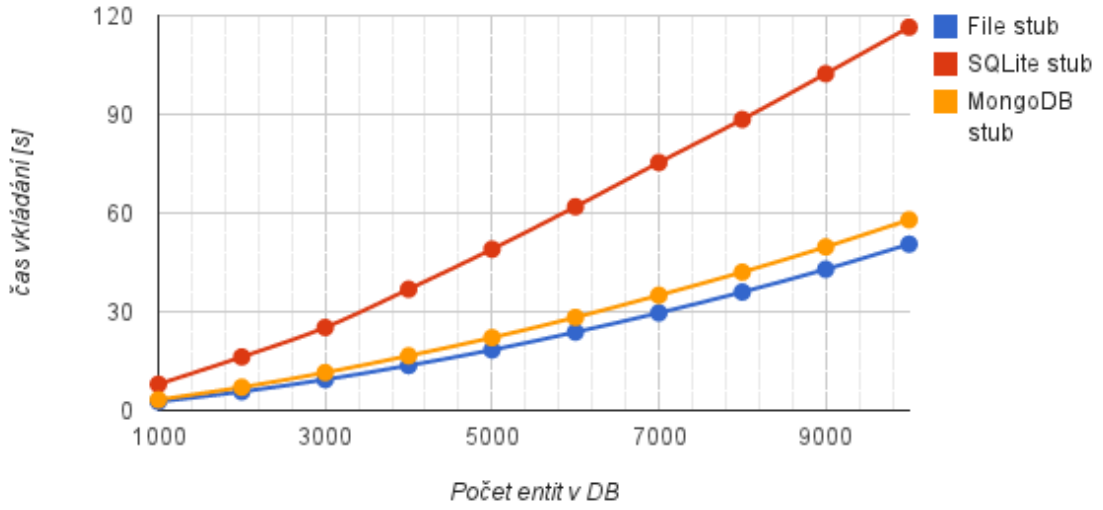
Kumulativní graf na obrázku 6.1 zobrazuje závislost času potřebného pro vložení dat na množství vkládaných dat (na počtu entit vkládaných do DB). Je zřetelné, že ukládání do MongoDB je jen o několik procent pomalejší, než ukládání dat do paměti RAM File Stubem.

6.2.2 Filtrování

Testy na filtr byly prováděny v každé sérii vždy 100x a byl vypočten aritmetický průměr z výsledných hodnot. Dotaz obsahující testovaný filtr je zapsán v GQL takto:

```
SELECT * FROM Entity WHERE y <= 100 LIMIT 100
```

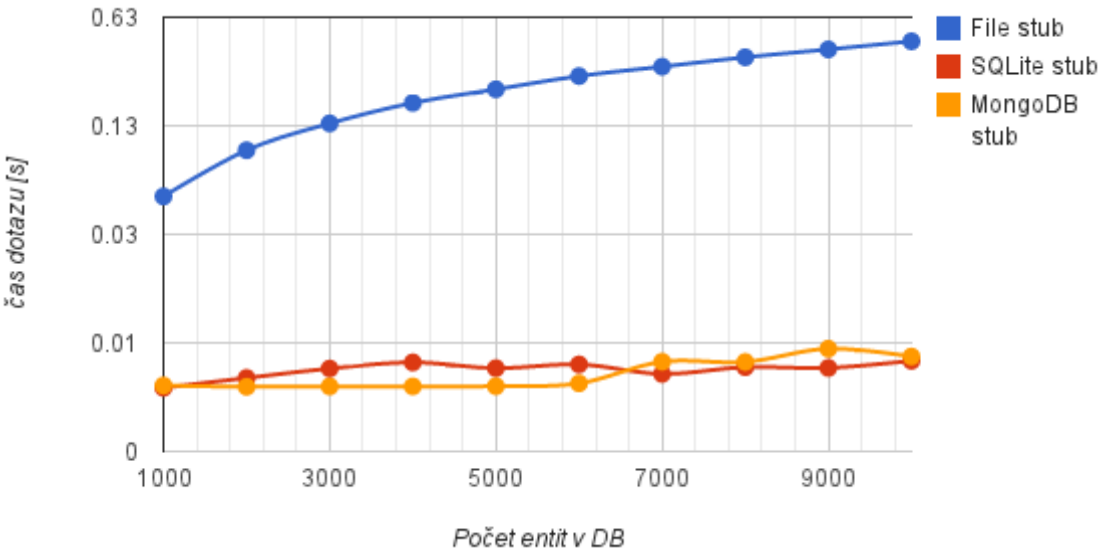
Přestože dotaz může v každé sérii navracet jiný počet záznamů (integery v atributu y jsou generovány podle uniformního rozložení), variance při tak velké datové sadě není natolik



Obrázek 6.1: Srovnání zápisové rychlosti Datastore MongoDB Stubu s existujícími stuby.

zásadní, aby ovlivnila výsledky všech sérií testování stejným směrem a tedy zpochybnila relevanci testu.

Výsledky tohoto testu byly vyneseny do grafu, který je zobrazen na obrázku 6.2. Měřítko osy y, která zobrazuje průměrný čas dotazu, je logaritmické, protože rozdíly mezi výsledky SQLite a MongoDB byly pod rozlišovací schopnost při tak velkém rozdílu od výsledků File stubu. Ten je při 10000 vložených entitách až 100 pomalejší, než SQLite stub. Výkonnost MongoDB stubu je srovnatelná se SQLite, což je velmi dobrý výsledek.



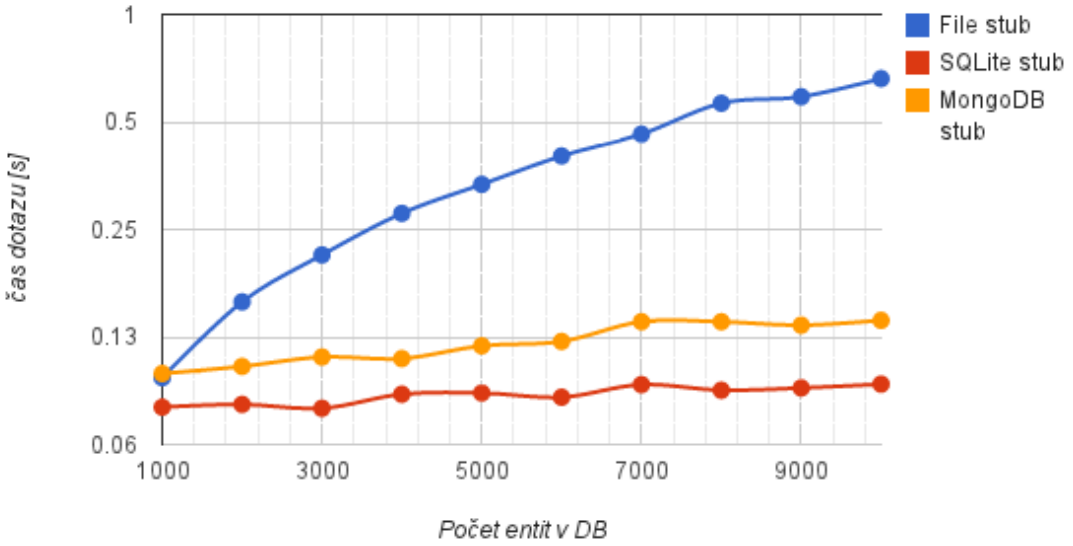
Obrázek 6.2: Srovnání rychlosti filtrování Datastore MongoDB Stubu s existujícími stuby.

6.2.3 Řazení

Výkonnost řazení byla testována také v každé sérii vždy 100x a stejně jako v případě filtrování byl vypočten aritmetický průměr z výsledných hodnot. Dotaz pro testy řazení je zapsán v GQL takto:

```
SELECT * FROM Entity ORDER BY d DESC LIMIT 100
```

Z grafu na obrázku 6.3, do kterého byly vyneseny výsledky testování, lze určit, že řazení zvládá nejlépe SQLite stub. MongoDB je pomalejší řádově o několik setin sekundy na dotaz, zatímco křivka File stubu stoupá až na hodnotu kolem 0.65 sekund při 10000 vložených entitách. Měřítko osy `y` je opět kvůli velkému rozdílu logaritmické.



Obrázek 6.3: Srovnání rychlosti řazení Datastore MongoDB Stubu s existujícími stuby.

6.2.4 Projekce

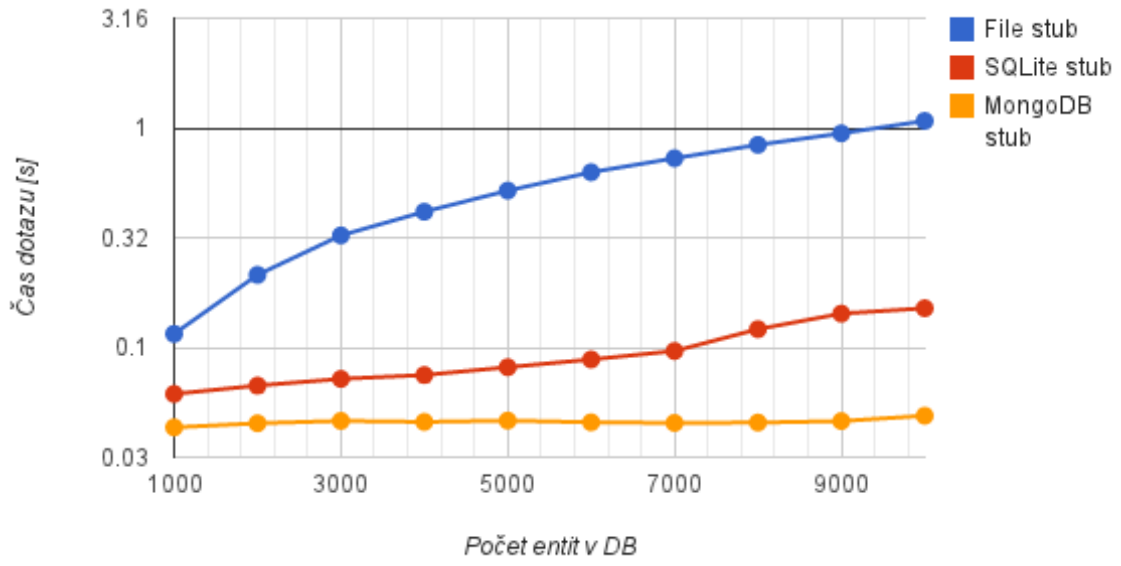
Posledním typem testu byla projekce na atribut vnořené entity `Inner`. Zde je testována jak schopnost stubu dotázat se jen na vybraný „sloupec“ záznamu, tak i rychlost dotazu na vnořenou entitu.

Dotaz, který testuje výběr atributu `z` (float) z vnořené entity, je zapsán v GQL takto:

```
SELECT i.z FROM Entity LIMIT 100
```

Hodnoty časů, které jednotlivé stuby potřebovaly pro vyhodnocení dotazů s projekcí, jsou znázorněny v grafu na obrázku 6.4. Graf zobrazuje závislost doby trvání dotazu na množství entit uložených v dotazované kolekci. Jako jediný z testů skončil tento vítězstvím MongoDB stubu, který při 10000 vložených entitách dokáže navrátit projektované hodnoty o desetinu sekundy rychleji, než SQLite stub.

File stub byl stejně jako v předchozích testech na dotazování výrazně pomalejší - hodnoty času potřebného pro vyhodnocení dotazu lineárně stoupají až za hodnotu jedné sekundy při 10000 záznamech v kolekci. Měřítko osy `y` je stejně jako v předchozích testech logaritmické.



Obrázek 6.4: Srovnání rychlosti databázové projekce Datastore MongoDB Stubu s existujícími stuby.

Kapitola 7

Integrace do SDK

Datastore MongoDB Stub je plně integrovatelný do Google App Engine SDK. Podporovány jsou všechny verze od 1.7.4. Pro podporu dané verze je třeba stáhnout z GitHubu vždy verzi z příslušné větve. Primární orientace celého stubu je na Linux, případě OSX, ale poskytována je i podpora pro Windows v podobě batch skriptu.

7.1 Integrace do vývojového serveru

Integrace do starého vývojového serveru (`dev_appserver`, `old_dev_appserver`) je založena na úpravě části SDK, převážně skriptů obsahujících implementaci serveru. Tato úprava je realizována patchem, jehož funkce je popsána v kapitole 7.3. Na straně stubu bylo pak nutné implementovat funkcionality, které používá pouze server, nikoliv knihovna `ndb`. Mezi tyto funkcionality patří:

- práce s uživatelsky definovanými indexy,
- podpora kurzorů pro generování statistik,
- podpora kurzorů pro pseudotypy (`__kind__`, `__property__`, `__namespace__`, ...),
- a jiné.

Kvůli vydání nového `devappserveru2` v březnu 2013 musela být do patche přidána podpora i pro tento nový server. Protože implicitním datastore stubem pro `devappserver2` je SQLite stub, byly vyřazeny z provozu všechny přepínače ovlivňující typ datastore stubu. Proto patch `devappserveru2` natvrdo přepisuje inicializaci SQLite stubu; přidání nového přepínače by znamenalo rozsáhlejší úpravy v několika modulech, což je pro údržbu podstatně náročnější. Tak rozsáhlé úpravy pak za situace, kdy se SDK rychle vyvíjí a mění, znamenají údržbu vlastní verze devappserveru2, což by ve výsledku bylo kontraproduktivní, protože každá změna by vyžadovala mnoho práce navíc a vydání nové verze stubu by bylo zpožděné.

7.2 Integrace do testovacích nástrojů

Další částí SDK, kam bylo třeba zavést podporu *Datastore MongoDB Stubu*, je nástroj *testbed*[1]. Ten slouží pro jednodušší inicializaci celého prostředí v unit testech. Jedná se

[1] V SDK se nachází v umístění google.appengine.ext.testbed.

o poměrně jednoduchou třídu `Testbed`, která obsahuje metody pro aktivaci a deaktivaci jednotlivých stubů v SDK a metody pro konfiguraci prostředí. Inicializace mongodb stubu je zpřístupněna metodou `init_datastore_v3_stub()`, které patch přidává nový parametr `use_mongodb`. Použití je osvětleno na následující ukázce:

```
from google.appengine.ext import testbed
tb = testbed.Testbed()
# aktivace testbedu
tb.activate()
# inicializace datastore mongodb stubu
tb.init_datastore_v3_stub(use_mongodb=True)
```

7.3 Instalace

Instalace *Datastore MongoDB Stub*u přímo do SDK probíhá automatizovaně. Pro tento účel byly vytvořeny skripty `install.sh` (Linux) a `install.bat` (Windows), které po spuš–tění se správnými parametry automaticky nakopírují stub do SDK (do umístění, kde se nachází i *Datastore SQLite Stub* - `google.appengine.datastore`) a integrují stub do obou vývojových serverů.

Pro instalaci stubu do SDK je třeba nainstalovat potřebnou (ideálně poslední) verzi GAE SDK pro Python a stáhnout z GitHubu adekvátní verzi stubu. Následující podkapitoly názorně ukazují přesný postup.

Linux

```
$ wget https://github.com/hellerstanislav/\
      appengine-datastore-mongodb-stub/archive/master.zip
$ unzip master.zip
$ cd appengine-datastore-mongodb-stub-master
$ bash install.sh /CESTA/K/SDK/
```

Windows

Pro instalaci na MS Windows je zapotřebí mít nainstalovanou utilitu `patch`[2], která přináší funkcionalitu unixového patche na Windows. Poté je nutné v příkazové řádce přidat cestu k nástroji `patch` do proměnné PATH a nainstalovat stub do umístění, kde se nachází SDK pomocí skriptu `install.bat` (ukázka pro 64bitovou verzi OS):

```
> PATH=%PATH%;''C:\Program Files (x86)\GnuWin32\bin\''
> install.bat ''C:\Program Files (x86)\Google\google_appengine\''
```

7.4 Použití

Jednovláknový server

Patch, který upravuje starý jednovláknový server (`old_dev_appserver`), do něj přidává nový parametr `--use_mongodb`, pomocí kterého je zpřístupněno použití MongoDB stubu.

[2]Ke stažení na http://gnuwin32.sourceforge.net/packages/patch.htm

Spuštění serveru je pak velmi přímočaré:

```
$ python ./google_appengine/old_dev_appserver.py --use_mongodb $PROJECTDIR
```

Zde je uveden způsob spuštění serveru v SDK verze 1.7.6. Ve starších verzích je server pojmenován `dev_appserver.py`.

Vícevláknový server

Jak již bylo v předchozí kapitole zmíněno, `devappserver2` ruší veškeré přepínače týkající se nastavení datastore stubu. Proto *Datastore MongoDB Stub* přímo nahrazuje SQLite stub, který po patchi již není možné používat. Spuštění serveru je tedy možné bez dodatečných parametrů:

```
$ python ./google_appengine/dev_appserver.py $PROJECTDIR
```

Kapitola 8

Závěr

V rámci teoretické části této práce byly analyzovány vlastnosti NoSQL databází, zvláštní pozornost byla věnována databázi *MongoDB*. Dále bylo analyzováno cloudové prostředí *Google App Engine* se zaměřením na NoSQL databázi *Google Datastore* a databázové *stuby* v Google App Engine SDK (verze pro Python).

Výsledkem těchto analýz je porovnání výkonnosti existujících databázových stubů a dokumentace programového vybavení, které stávající stuby využívají (tzv. Datastore Service Stub API, viz dodatek A). Tato dokumentace nebyla společností Google zveřejněna, což se projevuje velmi nízkou aktivitou komunity při spolupráci na vývoji nových stubů. Nutno podotknout, že popis API byl tvořen experimentálně, protože jazyk Python je dynamicky typovaný a neuvádí typy u parametrů metod a u návratových hodnot. Na základě analýzy použití MongoDB jako úložiště pro SDK byl vytvořen výčet nekompatibilních vlastností MongoDB a Google Datastore. Tyto poznatky byly použity pro návrh nového a výkonnějšího stubu.

Ve výsledku byl navržen a implementován nový *Datastore MongoDB Stub*, jako referenční implementace byl použit existující *Datastore File Stub*. V průběhu vývoje bylo nutné kvůli častým změnám ve vývojového prostředí stub stále přizpůsobovat novým verzím SDK. Rozsáhlé změny byly zapotřebí také po vzniku nového vývojového serveru `devappserver2` v březnu 2013.

Za hlavní výhody oproti existujícím stubům lze považovat optimální rychlost databázových operací (vkládání dat, dotazování). Ve srovnání se stávajícími řešeními nemá nový stub nedostatky, které v sobě skrývá ukládání dat do paměti RAM (*Datastore File Stub*) - velmi pomalé dotazování a dlouhé načítání většího objemu dat při inicializaci. Oproti *Datastore SQLite Stubu* nově implementovaný stub netrpí pomalým vkládáním, které je způsobeno tím, že SQLite každou operaci obaluje transakcí. U některých typů dotazů je MongoDB stub v porovnání se SQLite mírně pomalejší (řádově procenta), naopak v případě databázové projekce je MongoDB Stub rychlejší, než oba existující stuby.

V současnosti je *Datastore MongoDB Stub* přizpůsoben pro 4 nejnovější verze SDK (jedná se o verze 1.7.4 - 1.7.7) a je plně integrovatelný do obou vývojových serverů (jak starého `old_dev_appserveru`, tak do `devappserver2`). Důraz je kladen na kompatibilitu s různými verzemi MongoDB a knihovny `pymongo`. Správná funkčnost stubu v různých konfiguracích programového vybavení byla testována na sadě unit-testů. Finální verze stubu byla revidována a oceněna i jedním ze zaměstnanců společnosti Google - Takashi Matsuem (Google Developers Advocate)[1].

[1] https://groups.google.com/forum/?fromgroups=#!topic/google-appengine/2bq8eImTULE

V budoucnosti bude třeba zajistit stálý vývoj a přizpůsobení novým verzím SDK, které vychází každý měsíc. Další práce na stubu by měla směřovat k širšímu pokrytí testy (v oblasti multithreadingu) a k vyřešení jediné nepodporované vlastnosti - možnosti tvorby složeného indexu nad seznamy.

Aby měla celá práce smysl i v budoucnu, bude třeba větší propagace stubu mezi komunitou Google App Engine. Cílem propagace je zapojení vývojářů do údržby stubu a jeho příprav pro nové verze. V případě většího zájmu komunity bude experimentálně vytvořené API zveřejněno, aby podnítilo tvorbu nových a rychlejších stubů.

Literatura

[1] 10gen: MongoDB Manual: 2d Geospatial Indexes. 2013, [cit. 2013-4-9].
URL http://docs.mongodb.org/v2.2/core/geospatial-indexes/

[2] Baker, J.; Bond, C.; Corbett, J. C.; aj.: Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, s. 223–234.

[3] Chang, F.; Dean, J.; Ghemawat, S.; aj.: Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, Berkeley, CA, USA: USENIX Association, 2006.

[4] Chun, W. J.: *Core Python Applications Programming*. Prentice Hall, třetí vydání, 2012, s. 612–613.

[5] Copeland, R.: *MongoDB Applied Design Patterns*. O'Reilly, 2013, 30-33 s.

[6] Dirolf, M.: MongoDB AppEngine Connector. 2009, [cit. 2012-11-22].
URL https://github.com/mdirolf/mongo-appengine-connector

[7] Gilbert, S.; Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services. In *In ACM SIGACT News*, 2002.

[8] Google App Engine Developers Team: Structuring Data for Strong Consistency. 2011, [cit. 2013-4-14].
URL https://developers.google.com/appengine/docs/python/datastore/structuring_for_strong_consistency

[9] Google App Engine Developers Team: The Python NDB API. 2012, [cit. 2013-2-5].
URL https://developers.google.com/appengine/docs/python/ndb/properties

[10] Gulati, S.: How MongoDB Different Write Concern Values Affect Performance On A Single Node. 2011, [cit. 2012-12-4].
URL http://whyjava.wordpress.com/2011/12/08/

[11] ISO: Data elements and interchange formats – Information interchange – Representation of dates and times. ISO 8601:2004, International Organization for Standardization, Geneva, Switzerland, 2013.

[12] Johnson, N.: Announcing the SQLite datastore stub for the Python App Engine SDK. 2010, [cit. 2013-4-14].
URL http://blog.notdot.net/2010/03/Announcing-the-SQLite-datastore-stub-for-the-Python-App-Engine-SDK

[13] Katsov, I.: NoSQL Data Modeling Techniques. 2012, [cit. 2012-11-27].
URL http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/

[14] Lamport, L.: Paxos Made Simple, Fast, and Byzantine. In *OPODIS*, 2002, s. 7–9.

[15] Magnusson, P. S.: Royal Wedding Bells in the Cloud. 2011, [cit. 2012-11-14].
URL http://googleappengine.blogspot.cz/2011/05/royal-wedding-bells-in-cloud.html

[16] Nakov, S.: Software Development for the Public Cloud Platforms: Azure vs. App Engine vs. Amazon. Telerik Software Academy, 2012.

[17] Rogers, M.: MongoDB Performance & Durability. 2010, [cit. 2012-11-18].
URL http://redbeard0531.s3.amazonaws.com/mikeals_blog_backup/MongoDB+Performance+&+Durability.html

[18] Vogels, W.: Eventually Consistent. *ACM Queue*, ročník 6, č. 6, 2008: s. 14–19.

Dodatek A

Datastore service stub API

Všechny následující třídy se nacházejí v GAE SDK, v balíčku `google.appengine`. Uvedené API se vztahuje k verzi SDK 1.7.7.

A.1 api.apiproxy_stub.APIProxyStub

APIProxyStub je základní třída pro API proxy stuby a tvoří hlavní přístupový bod pro RPC volání. Každý stub by ji měl dědit a implementovat servisní metody - funkcionalitu stubu. Jedinou metodou, kterou může datastore stub s výhodou přepisovat, je `MakeSyncCall`.

`__init__(service_name, max_request_size=MAX_REQUEST_SIZE, request_data=None)`
Konstruktor.

- **service_name** *(str)*: jméno služby očekávané u všech volání.
- **max_request_size** *(int)*: maximální povolená velikost požadavku. Defaultně 1MB.
- **request_data** *(request_info.RequestInfo)*: v datastore stubu nevyužíváno.

`MakeSyncCall(service, call, request, response, request_id=None)`
Hlavní přístupová metoda pro RPC volání.

- **service** *(str)*: jméno služby, které je uvedeno v konstruktoru.
- **call** *(str)*: jméno funkce, která má být zavolána. Ve stubu pak musí existovat obslužná metoda s názvem ve tvaru `_Dynamic_<call>`.
- **request** *(google.net.proto.ProtocolBuffer.ProtocolMessage)*: protocol buffer dotazu korespondující s typem funkce.
- **response** *(google.net.proto.ProtocolBuffer.ProtocolMessage)*: protocol buffer pro odpověď korespondující s typem funkce.
- **request_id** *(str)*: unikátní řetězec identifikující požadavek.

A.2 datastore.datastore_stub_util.BaseDatastore

Třída, která dědí z `datastore.datastore_stub_util.BaseTransactionManager` a z `datastore.datastore_stub_util.BaseIndexManager`. Definuje základní chování datastore, včetně transakcionality a indexování. Není to však sám o sobě stub, protože nekomunikuje přes RPC proxy API.

`__init__(require_indexes=False, consistency_policy=None, use_atexit=True, auto_id_policy=SEQUENTIAL)`

Konstruktor.

- **require_indexes** *(bool)*: příznak indikující, zda je vyžadováno, aby byly v souboru `index.yaml` nakonfigurovány složené indexy pro dotazy, které je vyžadují.
- **consistency_policy** *(datastore.datastore_stub_util.BaseConsistencyPolicy)*: způsob chování konzistence ve stubu. Mezi možnosti patří objekty tříd MasterSlaveConsistencyPolicy, TimeBasedHRConsistencyPolicy a PseudoRandomHRConsistencyPolicy.
- **use_atexit** *(bool)*: pokud je nastaven, obsah datastore je synchronizován na disk při ukončení činnosti.
- **auto_id_policy** *(str)*: konstanta definující chování automatického generování ID. Možné volby jsou `SEQUENTIAL` a `SCATTERED`.

`_GetQueryCursor(self, query, filters, orders, index_list)`

Provede daný dotaz a navrátí `QueryCursor`.

- **query** *(datastore.datastore_pb.Query)*: definice parametrů dotazu.
- **filters** *(list)*: seznam objektů definujících filtry, které mají vyšší prioritu, než ty, které jsou uvedeny v parametru dotazu.
- **orders** *(list)*: seznam objektů definujících řazení, které má vyšší prioritu, než to, které je uvedeno v parametru dotazu.
- **index_list** *(list)*: seznam indexů, které dotaz používá.

`_Get(reference)`

Navrátí entitu definovanou klíčem v parametru.

- **reference** *(datastore.entity_pb.Reference)*: klíč entity.

`_Put(entity, insert)`

Vloží entitu do datastore. Pokud v datastore existuje a je nastaven příznak `insert`, selže.

- **entity** *(datastore.entity_pb.EntityProto)*: entita určená k uložení do datastore.
- **insert** *(bool)*: příznak, zda má metoda selhat, pokud již vkládaný objekt v datastore existuje.

`_Delete(reference)`
Smaže entitu asociovanou s klíčem v parametru.

- **reference** *(datastore.entity_pb.Reference)*: klíč entity, která má být smazána.

`_GetEntitiesInEntityGroup(entity_group)`
Navrátí obsah entitní skupiny (specifikované v parametru) ve formě slovníku mapujícího datastore_types.ReferenceToKeyValue(key) na entity_pb.EntityProto.

- **entity_group** *(datastore.entity_pb.Reference)*: klíč entitní skupiny.

`_AllocateIds(reference, size=1, max_id=None)`
Alokuje ID pro danou referenci. Navrací dvojici (min, max), která definuje alokovaný rozsah integerů.

- **reference** *(datastore.entity_pb.Reference)*: klíč entity, pro kterou je alokováno ID.
- **size** *(int)*: velikost rozsahu alokovaných ID.
- **max_id** *(int)*: vyšší hranice rozsahu alokace.

A.3 datastore.datastore_stub_util.DatastoreStub

DatastoreStub je třída delegující volání služby na backend, který je reprezentován instancí třídy implementující rozhraní BaseDatastore (viz výše). Na tento backend nejsou kladena žádná omezení a jako úložiště lze využít prakticky libovolnou databázi.

`__init__(datastore, app_id=None, trusted=None, root_path=None)`
Konstruktor.

- **datastore** *(datastore.datastore_stub_util.BaseDatastore)*: nízkoúrovňová obsluha úložiště a jeho operací.
- **app_id** *(str)*: string definující ID aplikace. Pro aplikace vyvíjené v SDK má ID prefix `dev~`.
- **trusted** *(bool)*: bit indikující, že aplikace je důveryhodná. Důveryhodná aplikace může zapisovat do dat jiných aplikací.
- **root_path** *(str)*: cesta vedoucí k adresáři, kde se nachází SDK.

`Clear()`
Smaže celý datastore, včetně cache stubu a lokálně uložených nastavení.

`_Dynamic_Get(request, response)`
Navrátí entity pro klíče, které jsou v `request.key_list()`.

- **request** *(datastore.datastore_pb.GetRequest)*: objekt protocol bufferu obsahující data dotazu, mimo jiné klíče entit, které mají být načteny.

- **response** *(datastore.datastore_pb.GetResponse)*: objekt protocol bufferu obsahující data odpovědi. V ní by se měly nacházet načtené entity.

`_Dynamic_Put(request, response)`
Uloží do datastore entity, které jsou v `request.entity_list()`. V odpovědi pak navrací klíče uložených entit.

- **request** *(datastore.datastore_pb.PutRequest)*: objekt protocol bufferu obsahující data dotazu, mimo jiné entity, které mají být uloženy.
- **response** *(datastore.datastore_pb.PutResponse)*: objekt protocol bufferu obsahující data odpovědi, mimo jiné klíče uložených entit.

`_Dynamic_Delete(request, response)`
Smaže z datastore entity, které mají klíče v `request.key_list()`.

- **request** *(datastore.datastore_pb.DeleteRequest)*: objekt protocol bufferu obsahující data dotazu, mimo jiné klíče entit, které mají být smazány.
- **response**: Nevyužito. Parametr se zde nachází pouze kvůli konvenci dvojice parametrů u ostatních metod.

`_Dynamic_RunQuery(query, query_result)`
Provede dotaz na databázi a navrátí kurzor.

- **query** *(datastore.datastore_pb.Query)*: objekt protocol bufferu obsahující parametry dotazu.
- **query_result** *(datastore.datastore_pb.QueryResult)*: objekt protocol bufferu obsahující parametry odpovědi, mimo jiné také ID kurzoru pro iteraci nad navrácenými entitami.

`_Dynamic_Next(next_request, query_result)`
Získání další skupiny výsledků z kurzoru. Id kurzoru je v dotazu obsaženo v `next_request.cursor().cursor()`.

- **next_request** *(datastore.datastore_pb.NextRequest)*: objekt protocol bufferu obsahující parametry dotazu.
- **query_result** *(datastore.datastore_pb.QueryResult)*: objekt protocol bufferu obsahující načtené entity.

`_Dynamic_AddActions(request, _)`
Asociuje tvorbu jedné nebo více úloh s transakcí.

- **request** *(api.taskqueue.taskqueue_service_pb.TaskQueueBulkAddRequest)*: objekt protocol bufferu obsahující úlohy, které mají být vytvořeny, když je transakce potvrzena pomocí operace commit.

- _: Nepoužívá se.

Následují metody pro operaci s transakcemi:

`_Dynamic_BeginTransaction(request, transaction)`

Založí novou transakci.

- **request** *(datastore.datastore_pb.BeginTransactionRequest)*: objekt protocol bufferu obsahující parametry dotazu na transakci.
- **transaction** *(datastore.datastore_pb.Transaction)*: objekt protocol bufferu obsahující parametry transakce.

`_Dynamic_Commit(transaction, response)`

Potvrdí změny provedené v transakci - provede operaci commit.

- **transaction** *(datastore.datastore_pb.Transaction)*: objekt protocol bufferu obsahující parametry transakce.
- **response** *(datastore.datastore_pb.CommitResponse)*: objekt protocol bufferu obsahující odpověď.

`_Dynamic_Rollback(transaction, _)`

Zneplatní změny provedené v transakci - provede operaci rollback.

- **transaction** *(datastore.datastore_pb.Transaction)*: objekt protocol bufferu obsahující parametry transakce.
- _: Nepoužívá se.

Dále se ve třídě nachází metody pro operaci s indexy, které však nemusí být nutně implementovány, pokud není podporován přepínač `--require_indexes` vývojového serveru. Z toho důvodu zde uvádím jen výčet hlaviček těchto metod.

```
_Dynamic_CreateIndex(index, id_response)
_Dynamic_GetIndices(app_str, composite_indices)
_Dynamic_UpdateIndex(index, _)
_Dynamic_DeleteIndex(index, _)
```

Dodatek B

Seznam použitých zkratek

Zkratka	Význam
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BSON	Binary JSON
CGI	Common Gateway Interface
DDL	Data Definition Language
DML	Data Manupulation Language
DQL	Data Query Language
DVCS	Distributed Version Control System
FIFO	First In First Out
GAE	Google App Engine
GFS	Google File System
GQL	Google Query Language
HRD	High Replication Datastore
LRU	Least Recently Used
MVCC	Multiversion Concurrency Control
OCR	Optical Character Recognition
ODM	Object-Document Mapping
ORM	Object-Relational Mapping
REST	Representational State Transfer
SDK	Software Development Kit
WAL	Write Ahead Log
WSGI	Web Server Gateway Interface