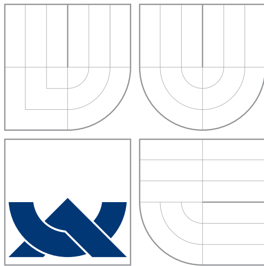
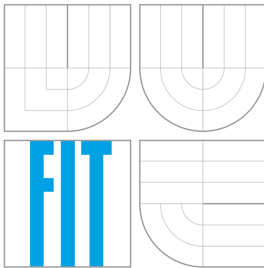


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NOSQL: PODPORA PRO UKLÁDÁNÍ VÍCEROZMĚR-NÝCH DAT

NOSQL: SUPPORT OF MULTIDIMENSIONAL DATA STORAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAN HRIVNÁK
AUTHOR

VEDOUCÍ PRÁCE Ing. VOLF TOMÁŠ
SUPERVISOR

BRNO 2014

Abstrakt

Tato bakalářská práce se zabývá možnostmi uložení vícerozměrných dat v NoSQL databázích. Je zde diskutována vhodnost vybraných databází pro ukládání trajektorií pohybujících se 2D objektů. Nad vybranou databází MongoDB jsou poté provedeny experimenty s cílem nalézt nejlepší možný způsob uložení a indexování dat trajektorií. V rámci práce je také popsán jednoduchý demonstrační program pro práci s MongoDB API v jazyce Python. V závěru jsou navrženy možnosti dalšího rozšíření této práce.

Abstract

This bachelor's thesis deals with posibilities of storing multidimensional data in NoSQL databases. It compares suitability of several different databases for storing trajectories of moving 2D objects. Experiments with MongoDB were carried out to find the best way to store trajectories and create indexes over them. This thesis also provides a simple demonstration program in Python which works with MongoDB API.

Klíčová slova

NoSQL, databáze, vícerozměrná data, trajektorie, MongoDB

Keywords

NoSQL, database, multidimensional data, trajectories, MongoDB

Citace

Jan Hrivnák: NoSQL: Podpora pro ukládání vícerozměrných dat, bakalářská práce, Brno, FIT VUT v Brně, 2014

NoSQL: Podpora pro ukládání vícerozměrných dat

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Tomáše Volfa a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Jan Hrivnák
21. května 2014

Poděkování

Rád bych poděkoval vedoucímu své práce Ing. Tomáši Volfovi za jeho čas a cenné rady, které mi poskytl během konzultací.

© Jan Hrivnák, 2014.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Vícerozměrná data **4**
- 2.1 Obecná vícerozměrná data . . . 4
- 2.2 Prostorové objekty . . . 5
- 2.3 Časoprostorové objekty . . . 6
- 2.4 Trajektorie . . . 6

3 NoSQL **8**
- 3.1 Not only SQL . . . 8
- 3.2 Odlišnost NoSQL od relačních databází . . . 8
- 3.3 Dělení NoSQL databází . . . 9
- 3.4 Související pojmy . . . 11
  - 3.4.1 ACID . . . 11
  - 3.4.2 BASE . . . 11
  - 3.4.3 CAP teorém . . . 12

4 Ukládání vícerozměrných dat **14**
- 4.1 Možné způsoby uložení času . . . 14
- 4.2 Indexování . . . 15

5 Zkoumané vícerozměrné databáze **16**
- 5.1 Rasdaman . . . 16
- 5.2 GT.M . . . 16
- 5.3 Globals . . . 17
- 5.4 MongoDB . . . 17
  - 5.4.1 Vícerozměrné indexy . . . 17
  - 5.4.2 Prostorové operátory . . . 18
- 5.5 CouchDB . . . 18
  - 5.5.1 Map-reduce funkce a pohledy . . . 18
  - 5.5.2 GeoCouch . . . 19

6 Návrh řešení **20**
- 6.1 Vstupní data . . . 20
- 6.2 Návrh řešení v MongoDB . . . 20
  - 6.2.1 Vkládání dat . . . 22
  - 6.2.2 Indexování . . . 23
  - 6.2.3 Dotazování . . . 24

6.3 Návrh řešení v CouchDB 25
6.3.1 Vytvoření pohledu a možnosti dotazování 26
6.3.2 Výběr CouchDB pro experimenty 27

7 Experimenty 28
7.1 Experimenty nad MongoDB 28
7.2 Zhodnocení experimentů 31

8 Demostrační program 33

9 Závěr 34

A Obsah CD 37

Kapitola 1

Úvod

Vícerozměrná data jsou v dnešní době používána ve stále větším počtu aplikací či služeb. Spolu se zvyšujícím se počtem uživatelů a zvětšováním okruhu služeb poskytovaných pomocí informačních technologií, jsou tak stále častější požadavky na vysokou výkonnost databázových systémů zpracovávajících právě tato vícerozměrná data.

Práce si klade za cíl seznámit čtenáře s problematikou vícerozměrných dat, konkrétně prostorových a časoprostorových objektů. Blíže je v ní rozebrán popis trajektorií, zachycujících pohyb zmíněných objektů v čase a jejich možné uložení v NoSQL databázích. Je zde přiblížen princip fungování těchto databází, základní rozdělení a rozdíly oproti známějším relačním databázím.

V kapitole 2 jsou teoreticky popsána vícerozměrná data a trajektorie. V kapitole 3 je přiblížena problematika zmíněných NoSQL databází a v kapitole 4 se blíže podíváme na důležité pojmy související s ukládáním vícerozměrných dat. Patří mezi ně různé možnosti práce s časem a především možnosti indexování. Následně je v kapitole 5 představeno několik vybraných databází a je diskutována jejich vhodnost pro práci s popsanými daty. Nad databázemi, které byly označeny jako vhodné, je poté v kapitole 6 proveden konkrétní návrh možného řešení ukládání dat trajektorií a dotazování nad nimi.V kapitole 7 je provedena řada experimentů pro ověření vhodnosti navrženého řešení nad MongoDB a v kapitole 8 je popsán jednoduchý demonstrační program ukazující možnosti práce s MongoDB API.

V závěru jsou shrnuty získané poznatky z experimentální části a jsou navrženy možnosti dalšího rozšíření této práce.

Kapitola 2

Vícerozměrná data

Společně se značným rozšířením informačních technologií do všech oblastí lidské práce se také zvyšuje komplexnost dat, která je potřeba ukládat v počítačových systémech a vyhledávat v nich. Většina těchto nestrukturovaných, případně semi-strukturovaných, dat ovšem již nemá přesně danou strukturu. Jsou totiž získávána nejrozličnějšími způsoby. Upouští se od ručního zadávání a nahrazuje se získáváním dat z kamer, snímačů a automatických systémů.

S rozšířením nejrůznějších mobilních zařízení se změnila i samotná data, která se v počítačových systémech musejí ukládat a pracovat s nimi. Stále více se jedná o data, která jsou nějakým způsobem vázána na geografickou polohu. Ať už se jedná o GPS souřadnice, na kterých se aktuálně uživatel nebo měřící senzor nachází, souřadnice odkud kam chce být uživatel navigován, přesné označení místa jisté události apod. Vzhledem k tomu, že se tyto informace často ukládají do databází a jejich množství se neustále zvyšuje, je potřeba zkoumat i nové způsoby, jak je zde efektivně ukládat, především s ohledem na rychlost vkládání a vyhledávání.

Výsledkem výše zmíněných příčin je, že při vyhledávání v těchto datech si již nevystačíme jen s dotazy porovnávající pouze jeden parametr, ale je potřeba porovnávat zároveň více parametrů. K tomuto účelu se nejčastěji používají data o více rozměrech. V této kapitole si představíme, co to taková vícerozměrná data jsou, jak se dají přibližně rozdělit a k čemu je můžeme využívat.

2.1 Obecná vícerozměrná data

Dle čistě teoretické definice lze za vícerozměrná data považovat veškerá data, která se skládají z více částí, položek či parametrů. V případě, že budeme například vytvářet databázi pro ukládání informací o klientech, tak i při vyhledávání pomocí „jména“+„příjmení“+„rodného čísla“ se jedná o vícerozměrný dotaz, neboť každá z vyhledávaných položek nám vytváří nový rozměr, ve kterém hledáme jistá data.

V této práci však dále budeme pojem „rozměr dat“ chápat pouze ve smyslu prostorového rozměru či času. Za vícerozměrná data tak budeme považovat pouze taková data, která mají jistou spojitost s dvou- a více-dimenzionálním prostorem. Jako jediné možné rozšíření tohoto prostorového pojmu „rozměr dat“ budeme používat čas, využívaný v trajektoriích.

2.2 Prostorové objekty

V každém případě, kdy chceme zaznamenávat polohu nějakých objektů, musíme nejdříve umět popsat tyto objekty. Za prostorový (*spatial*) považujeme každý objekt, u kterého je nějakým způsobem popsán jeho tvar, velikost a poloha v prostoru. Při práci s prostorovými objekty nejčastěji narazíme na data dodaná v podobě vícerozměrných objektů na 2D podkladu. Mezi základní 2D objekty poté patří:

- Bod
- Čára
- Polygon

kde bod je zadán pouze souřadnicemi X a Y a čára je spojnice dvou bodů. V případě, že čára prochází více než dvěma body a každý z těchto bodů je v ní zahrnut pouze jedenkrát, mluvíme o lomené čáře. Polygon je složen z lomené čáry, která tvoří jistou ohraničenou oblast o nenulovém obsahu a má tedy počáteční i koncový bod shodný. Polygony do sebe mohou být dále vnořovány. Nejčastější využití nachází množina více základních objektů. Využívají se tak multi-množiny bodů, případně čar či polygonů.

Podobným způsobem je možné definovat také prostorové 3D objekty. Další informace případná zájemce nalezne například v [1] (kapitola 1.11.)

Důležitou vlastností při prací s prostorovými objekty je jejich vzájemná poloha. Například dva polygony mohou být navzájem: disjunktní, dotýkající se, protínající se, shodné či vnořené. Velmi častý je pak také dotaz na vzdálenost dvou objektů.

> „Vzdálenost mezi dvěma objekty je rovná minimální vzdálenosti mezi kterýmikoliv jejich body“ [1]

Znalost vlastností vzájemné polohy objektů a jejich vzdálenosti nám umožňují poté sestavovat řadu dotazů nad takovýmito daty. Příklady takových dotazů mohou být:

- Najdi všechny body, které jsou uvnitř 2 zadaných polygonů.
- Jaká je vzdušná vzdálenost mezi těmito městy (polygony)?
- Kolik uživatelů je od daného bodu vzdáleno méně než 150 m?

Další vlastností u každého prostorového objektu je informace, v jakém souřadnicovém systému jsou uvedeny jeho vlastnosti. V praxi se setkáme se třemi typy souřadnicových systémů:

- **Kartézský souřadnicový systém** – pozice se počítá podle os z daného středu. Jednotlivé osy jsou na sebe kolmé.
- **Geodetický systém** (neboli Zeměpisné souřadnice) – zobrazení vycházející ze zeměpisné šířky a délky.
- **Systém promítaných souřadnic** (*Projected coordinates*) - Jedná se o využití kartézského souřadnicového systému v 3D, do kterého je zasazena Země. Jednotlivé body jsou matematicky namapovány z povrchu Země do 2D roviny.

Pro lepší čitelnost se snažíme prostorové objekty zobrazovat uživateli také graficky a nikoliv pouze jako množinu čísel. Často se graficky vizualizují data spojená s geografickou polohou na povrchu Země. Systémy pracující s takovýmito daty se souhrnně označují jako GIS (geografické informační systémy). V těchto systémech však již nejsou vícerozměrné objekty pouze v podobě prostorových objektů, ale jsou doplněny o řadu dalších parametrů a vlastností. Využití tak zde nacházejí prostorové objekty doplněné například o nadmořskou výšku apod. Za neprostorový atribut se zde chápe vlastnost přímo nesouvisející s prostorem samotným. V systému pro zaznamenávání počasí by takovýmto neprostorovým atributem oblasti (polygonu) mohl být úhrn srážek za časové období.

2.3 Časoprostorové objekty

Za časoprostorové (*spatio-temporal*) objekty považujeme prostorové objekty, u kterých je kromě zmíněných prostorových vlastností také údaj o čase. Bod v 2D prostoru je tak místo souřadnic $[x, y]$ popsán trojicí $[x, y, t]$, kde x a y jsou prostorové souřadnice a t je čas. Při práci s takovým objektem nás tak přidání času k dvourozměrnému objektu přesouvá k výpočtům v třírozměrném prostoru. Obdobným způsobem jako body jsou rozšířeny o další parametr i zbývající objekty (čáry i polygony).

V případě bodu v 3D prostoru je místo popisu $[x, y, z]$, určujícím polohu objektu, použit zápis $[x, y, z, t]$, kde t je je opět proměnná času. Výpočet s pohybujícím se objektem v 3D prostoru se tak spolu s časem mění ve výpočet v čtyřrozměrném prostoru.

Čas je možné u časoprostorových objektů zapisovat ve dvou podobách:

- **Absolutní čas** – Jedná se o čas zachycení daného objektu. Často je zachycen pomocí takzvaného UNIX Timestamp[1]. Je možné se setkat s případy, kdy je požadována větší přesnost a kromě vteřin jsou použity i nižší řády času. Při práci s časem v této podobě je možno jednoduše srovnávat data z různých systémů.

- **Relativní čas** – Jedná se časovou značku, ukazující časovou vzdálenost od místa počátku měření. Lze jej využít v systémech, kde se jednotlivé záznamy v čase snímají v pravidelných intervalech. V případě přenosu dat mezi různými systémy je nutné řešit synchronizaci času.

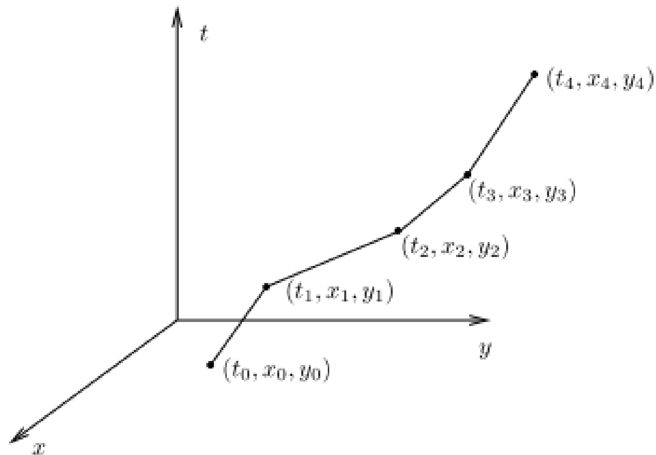
2.4 Trajektorie

Za pohybující se objekty považujeme časoprostorové objekty měnící svoji pozici v čase. Trajektorií nazýváme popis pohybu takového objektu [2]. Matematicky je za trajektorii bodu v 2D prostoru považována množina trojic, které můžeme zapsat jako

$$T = (t_1, x_1, y_1), (t_2, x_2, y_2), \dots, (t_n, x_n, y_n), \quad (2.1)$$

kde $x_i, y_i, t_i \in \mathbb{R} \wedge t_1 < t_2 < \dots < t_n$, dvojice $[x_1, y_1], \dots [x_n, y_n]$ jsou prostorové souřadnice daného bodu a $t_1, \dots, t_n$ je čas.

[1] Celočíselný počet vteřin uplynulých od 1.1. 1970 00:00:00



Obrázek 2.1: Grafické znázornění trajektorie bodu pohybujícího se v 2D prostoru v čase. Zdroj [2]

Trajektorii bodu pohybujícího se v 3D prostoru můžeme definovat obdobným způsobem jako množinu čtveřic

$$T = (t_1, x_1, y_1, z_1), (t_2, x_2, y_2, z_2), \ldots, (t_n, x_n, y_n, z_n), \tag{2.2}$$

kde $x_i, y_i, z_i, t_i \in \mathbb{R} \wedge t_1 < t_2 < \ldots < t_n$, trojice $[x_1, y_1, z_1], \ldots [x_n, y_n, z_n]$ jsou prostorové souřadnice daného bodu a $t_1, \ldots, t_n$ je čas [2].

Za trajektorii v uvedeném smyslu považujeme záznam pohybu objektu, reprezentovaného pouze jako bod v prostoru. Tento přístup se využívá ve valné většině systémů, neboť má nižší nároky na výpočetní výkon a práce s ním je jednodušší než při práci s celými pohybujícími se polygony. Příkladem použití může být monitorování pohybu vozů.

Druhou možností jsou záznamy pohybujících se polygonů. V tomto případě se ovšem již nejedná o trajektorie dle dříve uvedené definice. Takový pohyb řady bodů a čar, tvořících jeden polygon, je nutno zapisovat a ukládat složitějším způsobem. Polygony mohou v průběhu času měnit nejenom svoji polohu, ale také tvar či velikost.

Poslední a v aplikacích zachycující reálný svět (*real-world application*) často využívanou možností práce s trajektoriemi je přístup, kde se objekt pohybuje pouze po vytyčené trase. Jedná se například o sledování automobilů jedoucích po cestě, vlak jedoucí po kolejích či letadlo létající v koridoru. Ukládání těchto dat se dá převést na dva podproblémy. Nejdříve se dané trasy rozdělí na menší celky, jejichž vlastnosti se uloží. Poté se pohyb objektu po takovéto trase ukládá vždy jako záznam o čase a odkaz na danou část trasy, na které se objekt v dané chvíli nacházel. Samotné vyhledávání v takto uložených trajektoriích se poté také rozdělí, neboť se nejdříve naleznou části trasy, ve kterých chceme hledat, a poté se v záznamech o trajektoriích pouze hledají objekty nacházející se v daném čase na těchto částech trasy. Místo prostorového porovnávání polohy každého bodu se tak problém zjednoduší na jedno vyhledání částí trasy v prostoru a poté na vyhledávání podle jednoduchého klíče či indexu (kterým může být například číselné ID dané části trasy), což je výpočetně mnohem méně náročné. Tímto přístupem ukládání trajektorií se dále v této práci nebudeme zabývat.

Kapitola 3

NoSQL

Vzhledem k rostoucím požadavkům na výkonnost informačních systémů se výrazným způsobem dostávají do popředí zájmu distribuované systémy. Ty jsou schopny běžet na větším množství levnějšího hardwaru. V následující kapitole se tak blíže seznámíme s NoSQL databázemi jako zástupcem distribuovaných systémů pro ukládání dat. Rozebereme jednotlivé pojmy se kterými budeme dále v textu pracovat, vysvětlíme způsob práce těchto databází, zmíníme jejich rozdělení a budeme se také snažit důraznit rozdíly oproti známějším relačním databázím.

3.1 Not only SQL

Název NoSQL poprvé použil Carlo Strozzi v roce 1998 [3]. V jeho případě se ovšem nejednalo o NoSQL databázi z pohledu, jak ji vnímáme dnes. Názvem NoSQL chtěl Strozzi tehdy zdůraznit, že jeho databáze Strozzi NoSQL nepoužívá dotazovací jazyk SQL. K prvnímu použití pojmu NoSQL v dnešním smyslu došlo na meetingu v San Franciscu 11. června 2009[1], na němž byly probírány různé přístupy k volně šiřitelným distribuovaným nerelačním databázím[2] a kde jej v názvu meetingu použil pořadatel Johan Oskarsson.

Název NoSQL zde budeme chápat ve smyslu „Not only SQL“ [4], odkazující na to, že se nejedná o technologii, která by měla nahradit relační databáze, nýbrž by měla nabídnout alternativu. Měla by nabízet možnost pracovat s uloženými daty jinak než pouze v matematických relacích. Důležitou vlastností NoSQL databází je, že data v nich uložená nemají žádnou předem pevně danou pevnou strukturu (*schema-free*). To je velmi výhodné pokud potřebujeme mít u velkého množství objektů řadu parametrů, ale často jsou některé z nich nevyplněné. Stejně tak případné přidání nového parametru k nějaké položce v NoSQL databázi nijak neovlivní ty již existující. To by v opačném případě, kdy máme v systému například miliony uživatelů, byl poměrně velký problém.

Jedním z významných důvodů, proč se začaly vyvíjet a poté i rozšiřovat NoSQL databáze, je lepší možnost horizontálního škálování a replikace než v relačních databázích.

3.2 Odlišnost NoSQL od relačních databází

První databázové systémy, se kterými se v dnešní době prakticky každý čtenář seznámí již při studiu či samostudiu, jsou relační databáze. Tento pojem poprvé definoval Edgar F.

[1] `http://www.eventbrite.com/e/nosql-meetup-tickets-341739151`

[2] Originál z pozvánky: *open source, distributed, non relational databases*

Codd v roce 1970 [5] a jedná se o DBMS (dále jen DBMS[3]) založené na relačním modelu přístupu k datům. Data jsou uložena v tabulkách s pevně danou strukturou, kde je každý záznam uložen v podobě jednoho řádku. Vztahy mezi jednotlivými záznamy jsou uloženy v podobě matematických relací mezi jednotlivými záznamy. Tyto relační vazby, které daly vzniknout názvu tohoto konceptu ukládání dat jsou označovány jako cizí klíče.

Rozdílů mezi relačními databázemi a NoSQL je celá řada. Plynou především z toho, že se relační databáze používaly dříve a při vzniku NoSQL se tak jejich tvůrci snažili vytvořit systém odlišný od relačních databází, ve smyslu že doplňovali a zaměřili se právě na ty funkce, jenž jim v relačních databázích chyběly či nevyhovovaly.

Mezi tyto vlastnosti patří například zmíněná volná struktura dat v NoSQL. To byla vlastnost u relačních databází téměř[4] neřešitelná, neboť data v nich se ukládají v podobě řádků do tabulek, které mají strukturu přesně stanovenou.

Další vlastností je možnost zanořování dokumentů v NoSQL. V případě uložení komplexnějších dat v relačních databázích se neobejdeme bez výkonově náročných JOIN dotazů. V NoSQL se sice lze také odkazovat na jiné dokumenty, nicméně je tak potřeba činit mnohem méně často, a jelikož se na cizí dokumenty odkazuje přímo jejich indexem, je nalezení a připojení takového dokumentu v dotazu výkonově mnohem levnější řešení. Abychom ale NoSQL nepopsali jako naprosto dokonalý systém, tak je potřeba také zmínit, že nevýhodou tohoto přístupu „vše v jednom dokumentu“ může být jistá redundance dat. V relačních databázích se totiž zmiňovaný příkaz JOIN v dotazech nemusím provádět pokaždé. Mohou být dotazy, u kterých postačuje vrátit data z jedné tabulky. V tomto případě je to výkonově levnější řešení než v NoSQL, neboť ty musí zbytečně načítat mnohem větší dokument obsahující řadu, v tu chvíli, zbytečných dat.

Další nevýhodou NoSQL databází proti relačním je časté ustoupení od dodržování ACID teorému. Nevýhodou NoSQL může být také často menší předpřipravená funkčnost těchto databází. To je způsobeno především „volným stylem“ těchto databází, kdy nejenom díky bezschémovému přístupu, ale i dalším vlastnostem si každý vývojář pro svůj projekt může databázi upravit a využívat pouze to, co nezbytně potřebuje. V případě relačních DBMS, server poskytoval veškerou dostupnou funkčnost i v případě, že ji vývojář v daném případě vůbec nepotřeboval. To samozřejmě snižovalo výkon celé takové sestavy. NoSQL jsou tak často značně rychlejším řešením, ovšem za cenu toho, že si vývojář mnohem více věcí a vlastností musí hlídat nebo případně doprogramovat sám.

3.3 Dělení NoSQL databází

Hlavními propagátory a prvními tvůrci NoSQL databází byly přední IT společnosti. Každá z nich měla ale jiné požadavky na tvořený DBMS. Například firma Google potřebovala DBMS umožňující efektivní ukládání velkého množství dat a rychlé vyhledávání. Další společnosti potřebovaly rychlé ukládání, neboť u nich převažoval počet zápisů nad počtem čtení z databáze. Většinu společností však spojovala potřeba efektivnějšího škálování. Z ekonomického hlediska je pro tyto společnosti lepší provozovat větší množství levnějších serverů, než několik vysoce výkonných. K tomu však nebyly dříve používané DBMS v čele s relačním přístupem optimalizovány a horizontální škálování se tak stalo společným cílem velkého množství vznikajících DBMS. Z pohledu bezpečnosti a zálohování se stala dalším často jmenovaným parametrem replikace.

[3] z originálního *DataBase Management System*

[4] Existují systémy, které ukládají nestrukturovaná data například do BLOB objektů v relačních databázích, nicméně tento přístup na druhou stranu přináší řadu dalších nevýhod a příliš se nerozšířil.

V této kapitole se seznámíme se čtyřmi skupinami, do kterých lze NoSQL databáze zařadit především na základě jejich vnitřního fungování způsobu práce s daty a jejich ukládáním

1. Databáze typu **Klíč: hodnota** (*Key-value*)

   Tyto databázové systémy mají nejjednodušší datový model. Data ukládají pouze v podobě kolekce dvojic klíčů a hodnot. V těch lze vyhledávat pouze podle daného klíče, nikoliv podle uložené hodnoty. Výhoda těchto databází je v jejich velkém výkonu a jednoduché škálovatelnosti. Je v nich jednoduché provedení shardingu (rozdělení databáze na více serverů), kde se data mohou rozdělit na několik menších částí podle určitých intervalů hodnot například podle abecedy. Nevýhodou je někdy až příliš omezená funkcionalita. Nejčastěji se využívá v systémech, kde je požadavek na co nejlepší výkonnost při čtení a zápisu.

   Příklad: DynamoDB, Redis, Voldemort , Riak

2. **Dokumentově orientované** databáze (*Document oriented*)

   Jak již název napovídá, skládají se tyto databáze z dokumentů. V každém dokumentu jsou data organizována v podobě množin klíč:hodnota, kde hodnotou může být i další dokument. Tím do sebe mohou být dokumenty dále zanořovány a vzniká tak hierarchická struktura. Na rozdíl od databází typu klíč-hodnota dokumentové databáze podporují i vyhledávání podle samotných hodnot. Je taktéž možné indexovat a vyhledávat přes více položek současně(tzv. multi-index). Díky možnosti zanořování parametrů se pro ukládání dat využívají formáty jako JSON, BSON, XML, YAML apod.

   Příklad: MongoDB, CouchDB, SimpleDB.

3. **Sloupcově orientované** databáze (*Column oriented*)

   Průkopníkem tohoto typu databáze byla společnost Google a její BigTable. Tento typ NoSQL je optimalizován na vysokou výkonnost a paralelní zpracování při dotazování. Princip ukládání je podobný jako u klasické relační databáze, nicméně klasická tabulka s řádky a sloupci je zde uložena po sloupcích a nikoliv po řádcích. Díky tomuto způsobu řazení získávají sloupcově orientované databáze jednu ze svých hlavních předností - velmi rychlé vyhledávání podle daného klíče.

   Zástupcem toho typu je zmíněný BigTable, Cassandra nebo HBase.

4. **Grafové** databáze (*Graph databases*)

   Grafové databáze jsou inspirovány teorií grafů a jsou vhodné pro ukládání velmi komplexních dat. Data jsou v těchto databázích ukládána jako uzly a hrany grafu. Každý uzel i hrana mohou obsahovat další atributy. Značné uplatnění v současné době nacházejí díky rozšíření sociálních sítí. V případě grafu, kde uzly reprezentují jednotlivé uživatele a vztahy mezi nimi značí vzájemné přátelství. Díky teorii grafů a metodě nalezení nejkratší cesty grafem tak lze velmi efektivně vyhledávat „osoby, které možná znáte“, případně se dotazovat co mají dva lidé společného. Tyto dotazy nad klasickými relačními tabulkami by se spoustou JOIN dotazů trvaly mnohonásobně déle a grafové databáze jsou zde již nepostradatelné. Příklad využití v prostředí české služby nalezneme například na službě Jízdomat[5]

[5] `http://www.slideshare.net/janmittner/neo4j-jzdomat`

Příklady: Neo4J, HyperGraphDB

3.4 Související pojmy

3.4.1 ACID

Pojem ACID se sice u NoSQL databází používá pouze pro vysvětlení odlišností od relačních databází, nicméně pro úplnost této práce zde krátce rozebereme, co tento pojem znamená.

ACID je množina vlastností používaná v kontextu transakčního zpracování dat a jedná se o složeninu z anglických názvů čtyř těchto vlastností:

- **Atomicita** (***A**tomicity*) – zajišťuje, že skládá-li se transakce z více dotazů, budou provedeny všechny, nebo žádný z nich
- **Konzistence** (***C**onsistency*)- zajišťuje, že každá transakce po dokončení přivede databázi opět do konzistentního stavu, který bude odpovídat všem integritním omezením
- **Nezávislost** (***I**solation*) – operace prováděné jednou transakcí nejsou viditelné ostatním transakcím, dokud nejsou kompletně dokončené a potvrzené
- **Trvanlivost** (***D**urability*) – změny, které transakce úspěšně provede, jsou v databázi uloženy persistentně. Není možné, aby se změny provedené úspěšně dokončenou transakcí, například po výpadku proudu, ztratila

Většina post-relačních[6] databázových systémů ACID v plné míře nepodporuje, avšak získává díky tomu jiné výhody jako je například lepší horizontální škálování či replikace.

3.4.2 BASE

Z důvodu častého upouštění od ACID teorému v NoSQL databázích se začal využívat pojem BASE. Ten není tak striktní jako ACID a pro DBMS není tak obtížné jej dodržovat. BASE za cenu částečného ustoupení od konzistence a izolace dává přednost dostupnosti a výkonu. Jedná se o zkratku z anglických termínů:

- **Basic Availability** – systém je vždy dostupný. Klient dostane vždy odpověď na svůj dotaz
- **Soft-state** – aplikace nemusí být v každém okamžiku konzistentní
- **Eventual consistency** – aplikace se v konečném čase dostane opět do konzistentního stavu

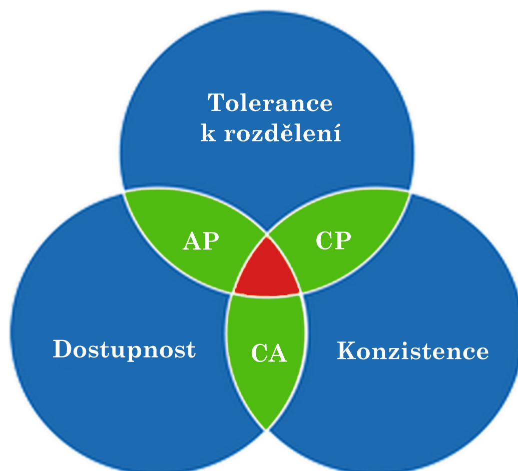
Databáze využívající BASE tedy nemusí vždy poskytovat konzistentní data. Například při aktualizaci dat na některém z uzlů nemusí všichni klienti dostat stejnou odpověď na shodný dotaz. Důležité je, aby po dokončení všech změn byla databáze opět v konzistentním stavu. Teorii přístupu BASE poprvé prezentoval v roce 2000 Eric A. Brewer [6].

[6] další možné označení systémů používajících jiný než relační přístup k datům

3.4.3 CAP teorém

Myšlenka CAP teorému byla poprvé prezentována také Ericem A. Brewerem[7] [6]. V jeho práci poprvé zazněla myšlenka, že žádný distribuovaný systém nemůže splnit všechny tři požadavky na popsaný CAP teorém současně. Tato teoretická myšlenka byla později formálně dokázána S. Gilbertem a N. Lynchem [7]. Těmito požadavky jsou:

- **Konzistence** (***C****onsistency*) – Všechny uzly v jednu chvíli mají k dispozici stejná data. Tj. pokud klient položí dotaz ve stejný čas na jakékoliv dva uzly, dostane vždy stejnou odpověď.
- **Dostupnost** (***A****vailability*) – Klient vždy obdrží nějakou odpověď. To znamená, že některý z uzlů klientovi vždy na jeho dotaz pošle odpověď.
- **Tolerance k rozdělení** (***P****artition tolerance*) – Systém bude schopen fungovat i v případě, že se spojení mezi některými uzly přeruší. Například při výpadku sítového spojení.



Obrázek 3.1: Grafické znázornění CAP teorému. Význam zkratek: C-konzistence, A-dostupnost, P-tolerance k rozdělení

Důsledkem tohoto teorému je, že každý distribuovaný systém, a tedy i NoSQL DBMS, se dle svého zaměření musí zařadit pouze do dvou z těchto pomyslných kruhů a třetí z nich částečně či úplně omezit. Díky tomu se tyto systémy dělí na tři skupiny:

- **CA** – data v těchto systémech jsou vždy dostupná a konzistentní. Tyto systémy však nepodporují toleranci k rozdělení. Patří mezi ně například relační databázové systémy jako MySQL apod.
- **CP** – patří sem systémy postavené na tom, že data jsou vždy konzistentní a mohou být oddělena. Cenou za tyto vlastnosti je, že některý uzel nemusí být v každé chvíli dostupný. Patří sem například BigTable, HBase, MongoDB nebo Redis.

[7] Někdy se tak lze setkat s pojmem Brewerův teorém

- **AP** – data v těchto systémech jsou stále dostupná, a to i v případě že dojde k přerušení komunikace mezi jednotlivými uzly. Nicméně tyto systémy negarantují, že všechny uzly budou mít ve stejnou chvíli stejná data. Tento přístup se využívá u systémů, které jsou velmi rozsáhlé, a udržení striktní konzistence by vyžadovalo příliš velkou režii. Patří sem například Cassandra, CouchDB , SimpleDB nebo také systém DNS.

Dle tohoto teorému nemůže existovat žádný počítačový systém, a tedy ani databáze, která by byla vždy dostupná, konzistentní a zároveň byla plně odolná vůči rozdělení. Často jsou distribuovaných systémy tvořeny tak, že třetí z požadavků CAP teorému úplně nevypouštějí, ale pouze jej omezují. V takovém případě se pak můžeme bavit o přístupu BASE zmíněného v kapitole 3.4.2.

Kapitola 4

Ukládání vícerozměrných dat

V této kapitole rozebereme možnosti ukládání vícerozměrných dat a blíže popíšeme několik důležitých pojmů a vlastností, se kterými budeme dále pracovat. Jsou jimi možnosti indexování a různé způsoby práce s časem.

4.1 Možné způsoby uložení času

V případě, že pracujeme s trajektoriemi pohybujících se objektů, je velmi důležité kromě výběru samotné databáze také způsob uložení dat v ní. Kromě uložení prostorových parametrů je nutné nějakým způsobem reprezentovat také další rozměr - čas. Stejně tak je důležité tento údaj správně indexovat, neboť se často využívá v dotazech nad těmito trajektoriemi.

Rozebereme zde několik možností uložení času v případě, že pracujeme pouze s bodem v prostoru. V případě ukládání pozice celého polygonu by popisované možnosti byly obdobné. Přibylo by však více variant, plynoucích z více možností, jak lze uložit pozici polygonu, jenž může měnit svoji polohu, velikost i počet bodů, ze kterých se skládá.

Příklady jsou ukázány pro uložení bodu v 2D prostoru, nicméně v 3D prostoru by tyto možnosti byly téměř shodné až na přidání dalšího rozměru (místo souřadnic $[x, y]$ by se využívalo $[x, y, z]$) a veškeré dimenzionální argumenty by se o jednu dimenzi zvýšili. K uložení času a souřadnic je možné přistoupit několika způsoby:

- **[x,y] + t** – Čas je uložen odděleně od prostorových souřadnic. Tento způsob využijeme především v případě, kdy máme k dispozici operátory, které umějí pracovat s daty v 2D prostoru. Čas se v takovém případě kontroluje až pomocí dalšího vyhledávacího parametru. V literatuře se lze setkat také s označením „2+1 rozměrný prostor"

- **[x,y,t]** – Čas je přidán jako další rozměr. Nad trajektoriemi v 2D prostoru je poté nutné použít dotazy a operátory určené pro třírozměrný prostor. V případě takového uložení času se veškerá práce s dvourozměrnými daty přesouvá do práce s třírozměrnými daty. Tento přístup je možné použít pouze v systémech, které umějí pracovat s třírozměrným prostorem, avšak tyto systémy mají připravenou řadu funkcí či operátorů a mají taky optimalizované algoritmy pro indexování takto rozměrných dat.

- **x + y + t** – V tomto přístupu jsou všechny parametry uloženy samostatně, jako pouhá čísla. Tento přístup nabízí nejmenší možnosti v podobě předpřipravených funkcí či operátorů. Na rozdíl od předchozích dvou ale neklade příliš velké nároky na zvolený DBMS a lze jej tedy využít prakticky ve všech systémech. Tento způsob je užitečný

především v případech, kdy použitý DBMS nedisponuje prostorovými indexy, případně disponuje pouze indexy pro méně rozměrný prostor.

4.2 Indexování

Dotazování nad vícerozměrnými daty je výkonově velmi drahou operací. Je zde tedy velmi důležité správně indexovat ukládaná data. Správným indexováním můžeme několikanásobně zkrátit dobu běhu některých vyhledávacích dotazů. Nevýhodou ovšem je jistá režie navíc při ostatních operacích (vkládání, odstraňování atd.).

> „Index je mechanismus pro lepší přístup k datům bez potřeby měnit strukturu dat samotných“ [8]

Pro samotnou práci s trajektoriemi v databázích není sice znalost vnitřního fungování indexů nezbytná, nicméně díky takové znalosti je možné navrhnout výkonově mnohem levnější řešení. Dále se v této práci budeme věnovat několika možným způsobům indexování dat trajektorií. Zaměříme se na prostorové indexy, avšak v rámci srovnání budeme také zkoumat uložení dat a jejich indexování bez využití specializovaných prostorových indexů. Ve všech případech se neobejdeme bez využití složených indexů.

Indexování neprostorových dat se nejčastěji v DBMS provádí pomocí B-stromu (*B-tree*). Ten je však pro prostorová či časoprostorová data nevhodný. Velmi často se tak využívá R-strom (*R-tree*), případně nějaká jeho upravená verze. R-strom byl navržen A. Guttmanem [9] v roce 1984 a jedná se o rozšíření B-stromu v n-dimenzionálním prostoru. Z jeho rozšíření zaměřených na prostorová a časoprostorová data jmenujme například: *3D R-tree*, *Spatio-Temporal R-tree* (STR-tree) či *Trajectory Bundle Tree* (TB-tree). Pro více informací o indexaci časoprostorových dat odkážeme čtenáře například na [10] nebo [11].

Kapitola 5

Zkoumané vícerozměrné databáze

V této kapitole si představíme DBMS, které byly zkoumány v rámci této práce. U každé z nich je na závěr zhodnoceno, zda a proč je vhodná či nevhodná pro ukládání dat trajektorií. Stejně tak je vždy zmíněno, zda byla daná databáze vybrána pro rozbor v dalších kapitolách této práce, kde se věnujeme podrobněji návrhu ukládání trajektorií a dotazování nad nimi.

5.1 Rasdaman

Rasdaman[1] je DBMS určený pro správu velkého množství (*big data*) GEO dat. Data ukládá v podobě multidimenzionálních polí o neomezené velikosti. Jedná se o vícerozměrnou databázi ve všech ohledech, neboť nemá žádné omezení na počet dimenzí a je v ní tedy možné efektivně ukládat i třídimenzionální prostorová data, případně doplněná i o čtvrtý časový rozměr.

Z dalšího zkoumání v této práci byl Rasdaman vyřazen z důvodu, že jsou v něm veškerá data ukládána v podobě polí (*arrays*) do BLOB objektů v relační databázi PostgreSQL[2] a nepovažujeme jej tedy za NoSQL databázi. Dalším důvodem byla vstupní testovací data pro experimenty v této práci, která obsahují trajektorie v 2D prostoru a neobsahují žádné GEO informace, pro které je tento DBMS určen.

V případě nasazení na některý z praktických projektů, který nemá důvod požadovat přesnou shodu s definicí NoSQL databáze, ale hlavním kritériem pro výběr bude vhodnost daného DBMS k řešenému problému, by ovšem využití databáze Rasdaman mohlo být vhodné.

5.2 GT.M

GT.M[3] je NoSQL databáze typu key-value podporující ACID teorém pro transakce. Jedná se o velmi rozsáhlý projekt určený a hojně využívaný k ukládání terabajtů dat. Používá se především v bankovním sektoru, kde se využívá podpory vícerozměrných indexů. Konkrétně je ale tento DBMS určen pro ukládání vícerozměrných dat, která jsme popsali v úvodu kapitoly 2.1, a není tedy určen pro ukládání prostorových vícerozměrných dat. Z tohoto důvodu nebyla tato databáze vybrána k dalším experimentům.

[1] http://www.rasdaman.com/

[2] http://www.postgresql.org/

[3] http://www.fis-gtm.com

5.3 Globals

Globals[4] je NoSQL databáze vyvíjená firmou InterSystems a hlavní využití nachází v komerčním produktu InterSystems Caché. Taktéž se jedná o DBMS pracující s multidimenzionálními poli.

Nevýhodou tohoto DBMS je, že poskytuje pouze Java API a plug-in do Eclipse. To bereme jako možnou poměrně velkou nevýhodu v porovnání s ostatními DBMS, které poskytují API pro řadu programovacíích jazyků zároveň. Dalším nevýhodou je, že DBMS Globals je sice možné stáhnout a používat zdarma, ovšem stále je vyvíjen jednou firmou, což je další nedostatek v porovnání s dalšími open-source řešením.

5.4 MongoDB

MongoDB je open-source dokumentově orientovaná NoSQL databáze, kterou začala v roce 2007 vyvíjet společnost 10gen. Databázový systém byl poté v roce 2009 uvolněn open-source komunitě pod licencí GNU AGPL a může být tedy veřejností využívána a modifikována.

S daty pracuje ve formátu JSON a ukládá je ve formátu BSON (binárně zakódovaný formát JSON [12]). Základní jednotkou je zde semistrukturovaný dokument, zhruba odpovídající řádku v relačních databázích, který však může obsahovat další zanořené dokumenty a pole. Dokumenty nemají žádnou pevnou strukturu a jsou dále seskupovány do kolekcí (*collections*), které se dají přirovnat k tabulkám v relačním modelu. Každý dokument má unikátní identifikátor *_id*, který je zároveň povinným unikátním indexem. Tento identifikátor generuje MongoDB automaticky, případně je možné jej nastavit ručně, ovšem v tom případě kontrola unikátnosti přechází na návrháře.

MongoDB má velmi dobře zpracovaný manuál[5], vysvětlující většinu důležitých vlastností i s řadou příkladů. Nebudeme zde tedy rozepisovat způsob instalace ani základní příkazy pro práci s MongoDB. V této práci byla použita aktuální verze MongoDB 2.6. K pohodlnějšímu prohlížení dokumentů můžeme využít některý z volně dostupných programů[6]. Konkrétně bylo při tvorbě této práce využito programu MongoVision ve verzi 1.1.

5.4.1 Vícerozměrné indexy

Z pohledu této práce byla MongoDB vybrána k dalšímu zkoumání díky tomu, že v ní jsou již naimplementovány tři typy indexů pro práci s vícerozměrnými daty. Jsou jimi *2dsphere*, *2d* a *haystack index*. Tyto indexy jsou vytvořeny pro práci s objekty v 2D prostoru.

Index „*2dsphere*“ je geo-prostorový (*geospatial*) index, sloužící k práci s daty v GeoJSON[7] formátu. Jedná se tedy o objekty, jež jsou definovány pomocí zeměpisné délky a šířky. V případě indexu „*2d*“ se jedná také o prostorový index, který ale šířku a výšku nepočítá na ploše koule, nýbrž na 2D rovině. Výchozí rozsah je $< -180, 180 >$, nicméně se dá upravit dle potřeby. Stejně tak se dá parametrem nastavit požadovaná přesnost rozsahu (Výchozí přesnost 26 bitů při přepočítání originálního rozsahu na plochu Země odpovídá přesnosti 60cm v reálném prostředí).

Třetím typem indexu je „*Haystack index*“. Ten je určen pro prostorová data, která jsou sdružována do menších celků/skupin. Při vytváření indexu se tak udává také velikost ob-

[4] http://globalsdb.org/
[5] http://docs.mongodb.org/manual/
[6] http://docs.mongodb.org/ecosystem/tools/administration-interfaces/
[7] http://geojson.org/geojson-spec.html

lasti, do které se mají objekty sdružovat (jeden objekt může být uvnitř několika skupin). Vyhledání objektů patřících do jedné skupiny je poté velmi rychlé. V této práci ovšem s tímto indexem nebudeme pracovat, neboť kvůli nutnosti předem definovat rozsah jednotlivých skupin je tento index vhodný spíše pro data, u nichž se dotazuje často stejnými či podobně rozsáhlými dotazy. Zároveň by bylo nutné pracovat s daty seřazenými nikoliv po trajektoriích, jako v předchozích dvou případech, nýbrž seřazených dle času.

5.4.2 Prostorové operátory

Dalším důležitou vlastností, díky které byla MongoDB vybrána, je existence operátorů umožňující jednodušší dotazování a práci s prostorovými souřadnicemi. Jsou jimi:

- **$geoWithin** – vyhledává objekty uvnitř jiného objektu, který může být definován jako obdélník ($box), polygonu ($polygon) nebo kruhu ($center)
- **$geoIntersects** – pro dotazy vracející prvky, které mají společný neprázdný průnik
- **$geoSearch** – pro dotazy využívající *Haystack index*
- **$geoNear** – pro dotazy vracející prvky patřící do jisté vzdálenosti od bodu na základě geo souřadnic
- **$near** – pro dotazy vracející objekty v blízkosti bodu
- **$nearSphere** - pro dotazy vracející objekty v blízkosti bodu na kouli

5.5 CouchDB

Apache CouchDB je dokumentově orientovaná NoSQL databáze napsaná v jazyce Erlang (první implementace v roce 2005 byla napsána v C++), která pracuje s daty ve formátu JSON. Jedná se o open-source řešení dostupné pod licencí Apache 2.0.

Amazon CouchDB splňuje vlastnosti pro ACID transakce. Dosahuje toho díky používání vlastnosti MVCC (*Multi-Version Concurrency Control*). Ta pracuje na principu , že nepřepisuje data, ale tvoří nové revize dokumentů při každé změně. Toto ukládání revizí má nevýhodu ve větším využití diskového prostoru databáze, k čemuž je ovšem určena funkce stlačení (*compaction*), která maže starší revize dokumentů a tím šetří místo. Pro jednodušší správu dat uložených v CouchDB je k dispozici webové grafické uživatelské rozhraní nazvané Futon[8].

5.5.1 Map-reduce funkce a pohledy

Stejně jako MongoDB se i v případě CouchDB jedná o dokumentově orientovaný DBMS. I když v tomto případě s o něco jednodušší datovou sadou, neboť jednotlivé dokumenty nesdružuje do kolekcí. Hlavním rozdílem je ovšem přístup k indexování. V MongoDB jsme si popsali (v kapitole 5.4.1) několik možností, jak indexovat samotné dokumenty a poté v nich vyhledávat. CouchDB k tomuto problému přistupuje zcela jinak. Za využití funkcí Map a Reduce tvoří nad samotnými dokumenty pohledy (*view*) a ty se teprve indexují.

Výhodou tohoto přístupu je fakt, že lze vytvořit celou řadu pohledů. Pokud pracujeme pouze s několika, často se opakujícími dotazy, je velmi výhodné si pro každý takový

[8]`http://couchdb.readthedocs.org/en/latest/intro/futon.html`

dotaz připravit vhodný pohled a nad tím teprve provádět vyhledávání. Výhoda proti klasickému vyhledávání nad celými dokumenty je zřejmá: Dokument obsahuje všechny informace, ovšem upravený pohled může obsahovat pouze ta data, která potřebujeme pro efektivní vyhledávání v daném dotazu a to ještě nejrůzněji upravené či seřazené.

Nevýhodou je, že každý takový pohled se musí při prvním použití dotazu spočítat a je nutné ho přepočítávat při každém vkládání či upravování dat v příslušných dokumentech, což může být při velkém množství dat časově náročnou operací (i přesto, že se nepřepočítává celý index, ale pouze položky, které se v něm změnily nebo byly přidány). Ovšem v případě, že provádíme analýzu nad již v podstatě statickou databází, která už všechna svá data obsahuje a provádíme tak minimální počet vkládacích a editačních dotazů, tento problém odpadá a pohled i s indexem se spočítá pouze při prvním použití.

Hlavní odlišnost tohoto DBMS od klasického přístupu k indexování a dotazování, tedy využití funkcí Map-Reduce, však není jen výhodou, ale může se stát také nevýhodou. Možnost napsat si vlastní pohled pro každý typ dotazu totiž není jenom možností, nýbrž také povinností. V CouchDB je tak časově i logicky mnohem náročnější vytváření i jednoduchých dotazů. Jako nevýhodu[9] to například zmiňuje i Nicholas Knize ve své přednášce o práci s prostorvými daty a R-stromem v NoSQL databázích v [13]

5.5.2 GeoCouch

GeoCouch[10] je rozšíření pro CouchDB. Přidává do funkčnosti CouchDB prostorový index založený na R-stromu. Díky použití tohoto rozšíření, je možné v CouchDB pracovat s prostorovými souřadnicemi ve formátu *[x, y]* a mít k tomu k dispozici vhodný index. Můžeme díky němu například ve funkci Map používat zápis:

```
1       function(doc) {
2               if (doc.loc) {
3                       emit(
4                               {type: "Point", coordinates: [doc.loc[0], doc.loc[1]]},
5                               [doc._id, doc.loc]
6                       );
7               }
8       }
```

[9] Z originálu: „Heavily dependent in Map-Reduce model (complicated design)“
[10] `https://github.com/couchbase/geocouch/`

Kapitola 6

Návrh řešení

V této kapitole budou rozebrány možné způsoby uložení trajektorií ve vybraných DBMS a budou blíže popsány možnosti indexování a použití předpřipravených funkcí pro vyhledávání. V případě všech řešení bude využíván kartézský souřadnicový systém a relativní čas.

6.1 Vstupní data

Jelikož je tato práce zaměřena na vícerozměrná data, s důrazem na časoprostorová data, byly jako vhodná testovací data použity záznamy trajektorií pohybujících se 2D objektů. Data pocházejí ze systému i-LIDS[1] a záběrů z londýnského letiště LGW airport. Jedná se konkrétně o data trajektorií získaná rozborem záznamu statické kamery grafickými algoritmy. Kamera měla rozlišení 800x600px a to jsou tedy i maximální hodnoty, které lze nalézt na souřadnicích X:Y. Z každého objektu byl vytvořen pouze bod v jeho středu a celá experimentální část se tedy zabývá trajektorií bodů pohybujících se ve 2D rovině.

Pro samotné testování byla využita sada 10 663 trajektorií zaznamenaných z videa „LGW_20071108_E1_CAM3“.

Vstupní data byla původně uložena v databázi PostgreSQL. Pro účely této práce byla z dat zachována pouze informace o trajektoriích a vstupní data tak byla dodána ve formátu tabulky, která se v PostgreSQL tvořila SQL příkazem:

```
1  CREATE TABLE tracks (
2     frames integer[],
3     positions point[]
4     );
```

Tento způsob uložení je ovšem nevhodný pro ukládání v NoSQL databázích, a tak byl formát dat upraven pomocí skriptů do podoby popsané u jednotlivých DBMS.

6.2 Návrh řešení v MongoDB

Pro potřeby experimentů byl v této práci zvolen, vzhledem k povaze vstupních dat, jako vhodný index „*2d*“. Původně je sice určen pro data zeměpisné šířky a délky, nicméně

[1] https://www.gov.uk/imagery-library-for-intelligent-detection-systems

je to nejbližší podoba podporovaného indexu k našim vstupním datům. Jako rozdělení dat trajektorií jsem zvolil model, kdy jedna trajektorie = jeden dokument. Na nevýhodu zvoleného přístupu je možné narazit pouze v případě, kdy by některá z trajektorií byla příliš rozsáhlá a celková velikost souboru dokumentu této trajektorie přesáhla 16MB (maximální velikost podporovaná MongoDB pro jeden dokument). Nicméně tento problém je řešitelný použitím GridFS[2], který rozdělí velký dokument do více menších celků a tudíž toto řešení považuji za dostatečně vhodné. Všechny dokumenty s trajektoriemi jednotlivých objektů jsou zahrnuty do jedné jediné kolekce.

Výsledná navržená struktura pro jednu trajektorii s využitím prostorového *2d* indexu:

```
{_id:ObjectId("generateID"),
        track_id: <tID>,
        pos: [
                {time: <t_1>,
                loc: [<x_1>,<y_1>]
                },
                {time: <t_2>,
                loc: [<x_2>,<y_2>]
                },
                {time: <t_3>,
                loc: [<x_3>,<y_3>]
                },
                ...
        ]
}
```

kde <x_1>, <x_2>, <x_3> jsou prostorové souřadnice na ose X, <y_1>, <y_2>, <y_3> jsou prostorové souřadnice na ose y, <t_1>, <t_2>, <t_3> jsou hodnoty času a <tID> je ID trajektorie.

Pokrývající dotaz

Za pokrývající dotaz (*covered query*) se v MongoDB považuje takový dotaz, který ke svému vyhodnocení nemusí načítat žádný dokument s uloženými daty a pro vyhodnocení dotazu mu stačí použít data z indexu. Vyhodnocení takového dotazu je rychlejší, neboť se vše vyhodnocuje podle indexovacího stromu, který je optimalizován na rychlé vyhledávání, a není nutné načítat celé soubory (dokumenty) z disku pro další parametry vyhledávání.

Vzhledem k tomu, že dotazy nad prostorovými indexy v MongoDB nelze použít u pokrývajících dotazů, bylo v rámci experimentů vyzkoušeno také uložení trajektorií bez využití prostorových indexů. Pouze s využitím základních datových typů, kdy souřadnice bodů byly rozděleny na samostatné hodnoty, tak byla navržena struktura:

[2] http://docs.mongodb.org/manual/reference/gridfs/

```
1       {_id:ObjectId("generateID"),
2               track_id: <tID>,
3               pos: [
4                       {time: <t_1>,
5                       x: <x_1>,
6                       y: <y_1>
7                       },
8                       {time: <t_2>,
9                       x: <x_2>,
10                      y: <y_2>
11                      },
12                      {time: <t_3>,
13                      x: <x_3>,
14                      y: <y_3>
15                      },
16                      ...
17              ]
18      }
```

Dotazy nad takto organizovanými daty jsou samozřejmě náročnější na programování. Neposkytují totiž možnost použití operátorů $geoWithin, $near apod. I pokud by experimenty ukázaly, že se jedná o nejrychlejší řešení, bylo by potřeba při použití takovéhoto uložení dat brát v potaz tuto značně omezenou funkcionalitu a využít jej pouze u specializovaných problémů, kde by chybějící operátory nebyly zásadním problémem. Na druhou stranu tuto funkcionalitu by nám poskytla v podstatě každá NoSQL databáze, neboť se jedná pouze o uložení pole čísel a indexů nad nimi a tudíž by se možný výběr vhodného reprezentanta z řad NoSQL mohl rozhodovat podle dalších parametrů jako je škálovatelnost, replikace apod.

Toto řešení bylo testováno také z důvodů, že MongoDB nepodporuje žádné další indexy pro vícerozměrná data. Pro práci s objekty v 3D prostoru nebo pro práci s objekty v 2D, kde bychom ale chtěli pracovat s časem, jak bylo popsáno v kapitole 4.1, v podobě $[x, y, t]$ by bylo nutné využít právě tento přístup k uložení dat.

Díky tomu, že MongoDB podporuje indexování prostorových dat a je velmi často doporučováno[3] právě pro použití v kombinaci s těmito daty, tak byl vybrán jako vhodný reprezentant pro další experimentování v této práci. Dále se tak v práci zaměřím na porovnání několika přístupů k ukládání a indexování dat, a to jak z pohledu teoretického tak i praktického, kdy budu provádět rychlostní srovnání zmíněných přístupů.

6.2.1 Vkládání dat

Vzhledem k podstatně jiné struktuře dat v PostgreSQL, sloužících jako vstupní data pro experimenty v této práci, jsem v Pythonu 3 naimplementoval skript pro převod do formátu JSON se strukturou blíže popsanou v předchozí podkapitole. Skript je k nalezení na přiloženém CD v `/scripts/sql_to_json.py`. Detaily použití jsou zobrazeny po spuštění s parametrem *−−help*.

Vložení dat bylo rozděleno na dvě části. První částí je nahrání samotných dat a druhou částí je vytvoření indexů nad těmito daty.

[3]např. `http://stackoverflow.com/a/19231732`

V této kapitole budeme zkratkou <collection> označovat název kolekce, se kterou se aktuálně pracuje. V samotných testech ji samozřejmě nahradíme za skutečný název kolekce. Vložení dat lze provádět pro každý záznam zvlášť příkazem:

```
db.<collection>.insert({ ... data ve formátu JSON ... })
```

Pro potřeby této práce jsem ale využil hromadný import dat (*bulk insert*), pro který jsme si již připravili data ve správném formátu. K hromadnému importu je v MongoDB připraven program *mongoimport*[4]. Import se provádí i s příslušnými nutnými parametry (v nápovědě nalezneme řadu dalších parametrů, které lze případně využít, například pokud nepracujeme s databází běžící na lokálním stroji) příkazem:

```
mongoimport.exe --db <collection> --type json --file <file>
```

kde <file> je cesta k souboru s daty ve formátu JSON. Jelikož kolekce před zahájením importu ještě neexistuje, není možné hned při vkládání data indexovat. Je potřeba to tedy provést až po vložení všech záznamů.

6.2.2 Indexování

Z návrhu struktury jednotlivých dokumentů popsané v kapitole 6.2 budeme do indexu zahrnovat následující položky:

- **track_id** - pro vyhledání konkrétního jednoho záznamu
- **pos.time** - pro vyhledání záznamů podle času
- **pos.loc** - zde využijeme *2d* index popsaný v kapitole 5.4.1
- **pos.x** a **pos.y** - pro „ruční“ prostorové vyhledávání bez použití prostorových indexů

Pro *2d* index určíme rozsah 0-800, jak jsme si určili v kapitole 6.1 popisující vstupní data. To vše za předpokladu, že počítáme pouze s celými čísly a přesnost tedy můžeme podstatně snížit z výchozích 26 bitů na nejnižší možnou hodnotu, tj. 1 bit. Takovýto složený index vidíme na prvním řádku a na druhém vidíme vytvoření indexu bez použití prostorového indexu.

```
db.<collection>.ensureIndex({"pos.loc":"2d", "pos.time":1, track_id:1},{min:0, max:
    800,bits:1})
db.<collection>.ensureIndex({"pos.time":1, "pos.x":1, "pos.y":1, "track_id":1 })
```

kde nad polem „pos.loc“ vytvoříme zmíněný *2d* index, nad položkou času „pos.time“ vytvoříme index řazený vzestupně a stejně tak „track_id“ řadíme vzestupně. Směr řazení určuje číslo za dvojtečkou, kdy hodnota 1 je pro vzestupné a -1 pro sestupné řazení.

[4] http://docs.mongodb.org/manual/reference/program/mongoimport/

Prvotní tvorbu indexu je možné spustit také na pozadí běžící a používané databáze. Ovšem z podstaty testovacích dat nepředpokládám, že by šlo o real-time aplikaci, nýbrž že dojde k vytvoření DB, vytvoření indexů a až poté bude databáze používána. Z toho důvodu jsem tuto možnost ani netestoval na rychlost. V případě, že vložíme nějaká data do kolekce, vytvoříme nad nimi index a poté vkládáme další data, tak tato nová data sama způsobí přepočítání indexu a prakticky je tedy samozřejmě možné takovou databázi využívat i v real-time aplikacích.

6.2.3 Dotazování

MongoDB je možno používat přes řadu dostupných API. Ty jsou dostupná[5] pro jazyky C, C++, Java, PHP, Python a řadu dalších. Kvůli co nejmenší možnosti ovlivnění výsledků použitým jazykem, budeme veškeré testy provádět přímo v příkazové řádce s využitím JavaScript API.

Dotazování se v MongoDB provádí pomocí konstrukce:

```
1    db.<collection>.find({<parameters>},{<return>})
```

kde <parameters> je seznam parametrů ve formátu JSON, podle kterých určujeme parametry vyhledávání, a <return> je popis prvků, které chceme, aby nám dotaz vracel.

V případě dotazů, kde parametrem je pouze vyhledávání podle času, využijeme konstrukci:

```
1    db.<collection>.find({"pos.time": {$gt:t1, $lt:t2}}, { track_id:1, _id:0})
```

která nám vrátí ID všech trajektorií objektů, které se nacházeli na záběru v době mezi t1 a t2 snímkem. Druhým parametrem dotazu je formát vracených dat. V tomto konkrétním případě se nám vrátí pouze track_id seřazené vzestupně. Konstrukcí *_id:0* říkáme, že nechceme vracet unikátní identifikátor dokumentu, který jinak MongoDB k výsledkům každého dotazu vypisuje automaticky. Za zmínku stojí také operátory $gt s $lt sloužící k omezení hledaných hodnot. K dispozici je v MongoDB řada dalších operátorů[6].

Při vyhledávání bodů procházejících daným polygonem (v tomto případě obdélníkem) využijeme, v případě, že pracujeme se souřadnicemi uloženými v podobě *[x, y]*, funkci *$geoWithin*. V opačném případě, kdy máme pouze základní datové typy, musíme jednotlivé rozměry kontrolovat ručně. V takovém případě využijeme také operátor *$elemMatch*, který zajistí, že se dané rozměry X a Y budou párovat pouze uvnitř pole v pořadí, v jakém jsou zde uloženy. Bez použití tohoto operátoru by se vyhodnotila první část a až poté druhá. Dotaz by tak vracel všechny trajektorie, které někdy vyhověly podmínce na rozsah X a někdy, ne nutně ve stejný čas, také podmínce pro rozsah na ose Y.

Na prvním řádku vidíme dotaz nad daty trajektorií, kde jsou souřadnice uloženy v podobě *[x,y]* a na druhém řádku vyhledávací dotaz s hodnotami souřadnic uloženými odděleně.

[5] http://api.mongodb.org/

[6] http://docs.mongodb.org/manual/reference/operator/query-comparison/

```
1   db.<collection>.find({"pos.loc": $geoWithin:{$box: [[x1,y1], [x2, y2]] }}, { track_
        id:1, _id:0})
2   db.<collection>.find({"pos": {$elemMatch:{ "x": {$gt:x1, $lt:x1}, "y": {$gt: y1, $lt
        : y2} } }}, { track_id:1, _id:0})
```

kde v prvním případě využijeme funkci *$box*, která definuje obdélník. Pro jeho specifikaci se zadává levý-dolní a pravý-horní roh. Hodnoty $x1$ a $x2$ označují rozsah hledaných hodnot na ose X a hodnoty $y1$ a $y2$ rozsah hodnot na ose Y.

V případě posledního dotazu, kdy se dotazujeme na prostorovou část dat a současně na čas, tedy asi nejčastější využití vyhledávání v trajektoriích pohybujících se objektů, využijeme operátor *$elemMatch* v obou případech uložení dat.

První řádek opět popisuje vyhledávací dotaz s použitím souřadnic uložených jako dvojice *[x, y]* a druhý řádek s uložením souřadnic odděleně. Význam použitých proměnných je shodný s výše uvedenými případy.

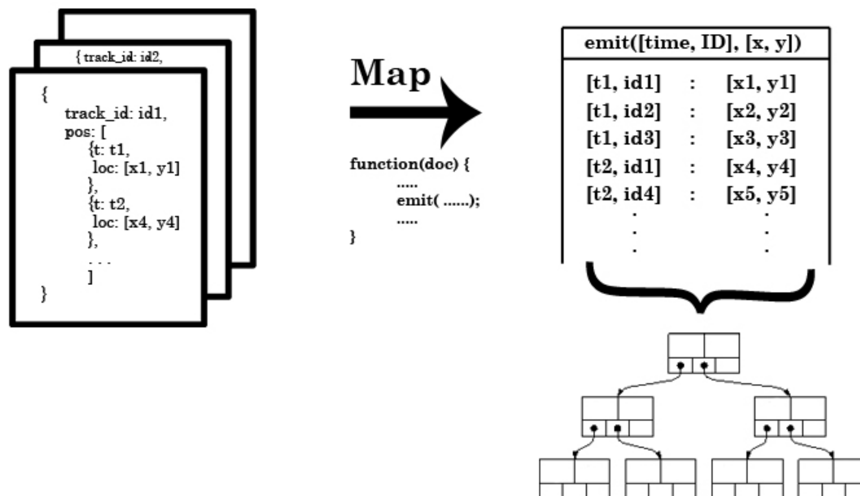
```
1   db.<collection>.find({"pos": {"$elemMatch": {"loc": {$geoWithin: { $box: [ [x1,y1],
        [x2, y2] ] }}, "time": {$gt: t1, $lt:t2}}}}, {track_id:1, _id:0})
2   db.<collection>.find({"pos": {"$elemMatch": {"x": {$gt:x1, $lt:x2}, "y": {$gt: y1,
        $lt:y2} , "time": {$gt: t1, $lt:t2}}}}, {track_id:1, _id:0})
```

6.3 Návrh řešení v CouchDB

Jak jsem již zmínili v kapitole 5.5.1, návrh každého dotazu nad CouchDB musí být prováděn přes pohledy. K vytváření pohledů pomocí funkcí Map-Reduce je nejrychlejší možností využití rozhraní Futon. V tom si vytvoříme dále popsané funkce v podobě „dočasných pohledů“ (*temporary view*) a ty poté uložíme jako klasické pohledy.

Pro návrh možného přístupu k řešení ukládání a vyhledávání v CouchDB si vybereme jeden konkrétní dotaz. Tím bude „Které objekty procházely ve stejný čas jistým polygonem?“ tj. dotaz hledající objekty, které se v daném čase potkaly. Ovšem i v případě ostatních dotazů by byl navržený postup velmi podobný.

6.3.1 Vytvoření pohledu a možnosti dotazování



Obrázek 6.1: Návrh vytváření pohledů nad CouchDB

Na obrázku 6.1 vidíme v levé části naznačenou řadu dokumentů s jednotlivými trajektoriemi. Nad těmito dokumenty navrhneme pohled. Tento pohled vytvoříme pomocí funkce Map naznačené ve střední části obrázku.

```
1     function(doc) {
2           for(var point in pos) {
3                 emit([point.time, track_id], [point.x, point.y]);
4           }
5     }
```

Obrázek 6.2: Návrh funkce Map pro vytvoření pohledu nad daty trajektorií seřazeného dle času

Výsledkem práce této mapovací funkce bude pohled, který bude obsahovat jako klíč dvojici *[čas, track_id]* a jako hodnotu dvojici souřadnic *[x, y]*. Z principu funkčnosti mapovací funkce bude tento pohled seřazen podle klíče a tedy v našem případě podle času a id dané trajektorie. Tento pohled je naznačen v pravé části obrázku. Nad tímto pohledem se vytvoří index přes *klíč*, který z pohledu vytvoří B-strom. Provedení výše popsaného dotazu poté bude značně rychlejší než vyhledávání nad samotnými dokumenty. V prvním kroku se ze seřazeného pohledu vybere pouze určitý rozsah časů a nad takovými daty se provede prostorové vyhledání daných objektů (za využití rozšíření GeoCouch).

Takové prostorové vyhledání však bude velmi rychlé, neboť se již vyhledává pouze mezi objekty, které byly v daném čase v záběru. Odpadá tak prostorové prohledávání nad objekty, které nebyly v daný čas viditelné a tedy ani nemohou být výsledkem daného dotazu.

6.3.2 Výběr CouchDB pro experimenty

V předchozí podkapitole jsme si navrhli teoretické fungování systému pro ukládání a práci s trajektoriemi v CouchDB. Pro další experimentování v této práci jsem však CouchDB nevybral. Hlavní příčinou tohoto rozhodnutí je popisovaná nutnost pracovat s funkcemi Map-Reduce. Tento přístup je značně odlišný od klasického přístupu ad-hoc dotazů aplikovatelných jak v relačních databázích a jazyce SQL, tak v MongoDB. Domnívám se, že nelze kvalitně otestovat všechny možnosti tvoření pohledů a experimentovat s jejich rychlostí v jedné podkapitole v experimentální části této práce. Proto bych experimentování a praktické ověření teorie nastíněné v kapitole 6.3 doporučil jako možné pokračování této práce.

V případě další práce nad trajektoriemi v CouchDB, by však bylo více než vhodné mít kromě vstupních dat zadány i rámcové dotazy, na které má být databáze připravena odpovídat. Z podstaty fungování pohledů není CouchDB vhodná pro uložení trajektorií a pokládání „jakýchkoliv“ dotazů nad nimi. Ovšem v případě, že bude stanovena jistá sada často používaných dotazů, tak věřím, že po připravení jednotlivých pohledů by databáze mohla poskytovat výsledky dotazů velmi rychle.

Posledním důvodem, proč jsem se rozhodl CouchDB v této práci dále experimentálně netestovat je, že dle vyjádření hlavního vývojáře rozšíření GeoCouch Volkera Mische z naší emailové konverzace v březnu 2014, je vícedimenzionální indexování funkční, ovšem v současné době se jedná o první verzi implementace, která zatím není optimalizována na rychlost. Byl jsme tak požádán, abych tuto informaci uvedl ve své práci jako známý nedostatek (*known bug*) na jehož odstranění se bude pracovat. V případě následování této práce a pokračování ve zkoumání možností CouchDB pro vícerozměrná data by tak mohl být tento výkonnostní problém již ze strany vývojářů odstraněn.

Kapitola 7

Experimenty

V předchozích kapitolách jsem popsal vícerozměrná data, provedl jsme seznámení s několika typy NoSQL databází, vyzdvihl jsme některé vlastnosti těchto databází a zhodnotil jsem jejich vhodnost pro ukládání trajektorií pohybujících se 2D objektů.

Původním plánem v této práci bylo provést stejné experimenty nad několika NoSQL databázemi a porovnat jejich rychlost práce s daty. Po důkladném seznámení s řadou vybraných NoSQL databází v kapitole 5 a návrhu možných přístupů k řešení ukládání a indexování dat trajektorií v kapitole 6 jsem se však rozhodl, že jako nejlepší řešení pro uložení a práci se vstupními daty, popsanými v kapitole 6.1, bude MongoDB. V tomto přesvědčení mě utvrdila jak řada názorů v nejrůznějších odborných fórech, tak také velmi aktivní komunita vývojářů[1], kteří implementovali různé prostorové indexy v MongoDB [14] ještě před oficiální implementací těchto indexů do databáze samotné. Díky této aktivní podpoře a řadě již reálně provozovaných projektů [13] využívající právě prostorové indexy z MongoDB považuji řešení v této databázi jako dostatečně funkční a předpokládám i jeho další rozšiřování a vylepšování ze strany tvůrců.

V této kapitole se tak zaměřím především na měření rychlostí dotazů nad různě vytvořenými indexy či různě uloženou strukturou dat v MongoDB a zkusím nalézt nejvhodnější řešení pro práci s popsanými testovacími daty.

Jelikož veškeré testy byly prováděny na lokálním stroji, tak zde pro úplnost uvádím také parametry tohoto počítače:

- Procesor: Intel Core i5-3210M, 2,50 GHz
- RAM: 8 GB
- Disk: SSD Samsung 840 EVO
- OS: Windows 8.1

7.1 Experimenty nad MongoDB

Jak již bylo řečeno v kapitole 4.2, vkládání dat a jejich indexování je velmi důležitou součástí návrhu práce v databázích. Z toho důvodu jsem vyzkoušel několik možností jak vložit data a vytvořit indexy nad strukturami popsanými v kapitole 6.2. Každý z uvedených testů jsem provedl vždy minimálně 10x a uvedený čas je aritmetickým průměrem naměřených

[1] `https://groups.google.com/forum/?hl=cs_US#!forum/mongodb-user`

hodnot. Jednotlivá měření byla prováděna ihned za sebou, nicméně nedocházelo k žádnému cachování, neboť jsem po každém dotazu použil příkaz pro vyčištění cache:

```
1    db.<collection>.getPlanCache().clear()
```

Dále v této kapitole budu zkratkou *„formát [x,y]“* označovat případ, kdy jsou data strukturována s využitím zápisu „...loc: [x,y]...“ a zkratkou *„formát čísel“* případ, kdy jsou hodnoty X a Y u jednotlivých souřadnic uloženy odděleně.

Převod SQL do JSON formátu Nejdříve jsem musel převést vstupní data, popsaná v kapitole 6.1, do vhodné podoby formátu JSON. Výsledné časy běhu skriptu pro 10 664 trajektorií můžeme vidět v tabulce 7.1.

Výstupní formát	**Průměrný čas převodu**
Formát [x,y]	11.23 s
Formát čísel	11.44 s

Tabulka 7.1: Převod formátu SQL do JSON

Hromadný import + odstranění všech dokumentů Dalším krokem byl import dat do databáze, jehož detaily jsem popsal v kapitole 6.2.1. V případech kdy během importu není vytvářen index nad prostorovými daty není potřeba rozlišovat mezi formátem [x,y] a formátem čísel. Jedná se o stejné množství dat a jejich import je tak shodně náročný.

Nejdříve jsem otestoval hromadné vložení dat do neexistující kolekce. Následně jsem ručně vytvořil kolekci s jedním záznamem a hromadný import provedl do ní. Pro otestování odstraňování dokumentů z kolekce jsem využil data naimportovaná v předchozím měření. V dalším případě jsem pro otestování rychlost vkládání do zaplněné kolekce a odstraňování většího množství dokumentů spustil import nad jednou kolekcí 11x se stejnými vstupními daty. Každý dokument se tak v této kolekci opakoval, ovšem díky tomu, že každý dokument při vkládání dostane své unikátní *_id*, se z pohledu vkládání i odstraňování jednalo vždy o 11 různých dokumentů.

Prováděná akce	**Průměrný čas**
Hromadný import 10 664 trajektorií do neexistující kolekce	9.24 s
Import do existující kolekce s jedním vytvořeným záznamem	8.21 s
Import do kolekce s vytvořenými desetitisíci záznamy	8.13 s
Odstranění 10 664 dokumentů z jedné kolekce	151 ms
Odstranění 117 304 dokumentů z jedné kolekce	1 480 ms

Tabulka 7.2: Experimentálně naměřené časy pro hromadné vkládání a odstraňování dokumentů

Pro odstraňování jsem použil příkazy:

```
1        use <database> //pro výběr aktuální databáze
2        db.<collection>.remove({}) //pro odstranění dokumentů z kolekce
3        db.<collection>.drop() //pro odstranění celé kolekce
4        db.dropDatabase(); //pro odstranění celé databáze
```

Z naměřených hodnot v tabulce 7.2 vidíme, že import do neexistující kolekce je zhruba o 1 sekundu pomalejší, než poté další přidávání stejného množství dokumentů do již vytvořené kolekce. To je způsobeno tím, že MongoDB nad každou kolekcí samo tvoří jeden povinný index *_id*. V případě kdy kolekce dosud neexistuje, se tak musí tento index nejdříve vytvořit a to způsobuje ono zpomalení. Sama instance programu mongo, která vypisuje počet připojených klientů a další důležité informace, nás o tom informuje pomocí:

```
1        build index on: ibp.tracks properties: { v: 1, key: { _id: 1 }, name: "_id_", ns: "
             ibp.tracks" }
```

kde „*ibp*“ je název naší databáze a „*tracks*“ název vytvářené kolekce. Z těchto naměřených hodnot taktéž plyne, že v případě bez vytváření vlastních indexů **je rychlost vkládání do existující kolekce nezávislá** na tom, zda vkládáme do téměř prázdné kolekce obsahující pouze 1 dokument či do zaplněné kolekce obsahující desetitisíce záznamů.

Další zjištěnou vlastností je lineární časová složitost odstraňování dokumentů. V případě mazání 10 tisíc dokumentů z kolekce, i v případě mazání téměř 120 tisíc dokumentů, byla rychlost odstraňování shodně zhruba 7500 dokumentů za 100 ms. Smazání celé kolekce a smazání celé databáze není ani v tabulce uvedeno, neboť jejich provedení proběhlo během pár ms a tedy předpokládám, že tyto příkazy fyzicky neodstraňují vlastní data, ale pouze odstraní referenci na danou kolekci či databázi z MongoDB. Nemělo tudíž význam testovat rychlost jejich provádění nad různě zaplněnými kolekcemi či databázemi.

Vytváření indexů Dalším testem, který jsem nad MongoDB provedl, bylo vytvoření indexů.

Vytvářený index	Naměrný čas
formát [x, y]: vytvoření „*2d*“ indexu a indexu nad „pos.time“	107.96 s
formát čísel: složený index nad [x, y], time a track_id	289.37 s
formát čísel: index pouze nad časem	102.64 s
formát čísel: složený index nad souřadnicemi X a Y	191.17 s

Tabulka 7.3: Experimentálně naměřené časy pro vytváření indexů nad kolekcí o 10 664 dokumentech

Při vytváření *2d* indexu jsem narazil na problém, kdy si databáze nedokázala poradit s takovým rozsahem testovacích dat. Při zahrnutí „pos.loc“ do prostorového indexu a spolu s ním do složeného indexu i „pos.time“, skončilo vytváření chybou „index too large“. Tyto dvě pole tedy při mém množství použitých vstupních dat nelze zahrnout do jednoho složeného indexu. Nad takovou kolekcí jsem tedy vytvořil dva samostatné indexy.

Nad formátem čísel jsem vyzkoušel, kromě jednoho složeného indexu obsahujícího všechny položky, vytvořit více indexů nad jednotlivými položkami samostatně.

Vyhledávací dotazy Vyhledávací dotazy jsem vyzkoušel v několika podobách a nad různě indexovanými kolekcemi. Výsledky těchto pokusů jsou zobrazeny v tabulce 7.4. Ve všech případech se prováděly stejné dotazy, tj. dotazy se stejnými parametry, které vracely stejné výsledky. V rámci dotazování jsem pokládal dva typy dotazů. První, který vracel správně pouhý 1 vyhovující objekt (tedy jeho track_id) a druhý, který vracel 100 track_id. Každý z těchto testů byl vždy proveden 10x a uvedený čas v tabulce je aritmetickým průměrem výsledků nad prvním i druhým dotazem. Za vhodného reprezentanta pro měření rychlosti dotazů jsem vybral ten, kdy se dotazujeme na prostorové souřadnice i na čas.

Formát	**Typ dotazu**	**Průměrný čas**	**Změna**
Formát čísel			
	dotaz bez indexu	527.55 ms	
	dotaz nad složeným indexem	413.90 ms	-22 %
	dotaz nad jednotlivými indexy	281.90 ms	-46 %
Formát [x, y]			
s $geoWithin	dotaz bez indexu	551.25 ms	
	dotaz s *2d* indexem	4877.75 ms	+785 %
	dotaz s jednoduchým indexem	217.15 ms	-61 %

Tabulka 7.4: Experimentálně naměřené časy pro dotazování nad MongoDB

Jak je vidět, nejdříve jsem prováděl měření nad formátem čísel, kde se použitím složeného indexu zvýšila rychlost vyhledávacích dotazů o 22 %. Nicméně po vytvoření každého indexu zvlášť, jak je popsáno v tabulce 7.3, se čas pro vyhodnocení stejného dotazu snížil průměrně až o 46 %.

Následně jsem vyzkoušel vyhledávací dotazy nad formátem *[x, y]* s pomocí operátoru *$geoWithin* a *$box* pro vyhledávání uvnitř obdélníku. Jak je vidět v tabulce, tak tento způsob dotazování bez indexů je o 5 % pomalejší než v případě formátu čísel. Nicméně se jedná malý rozdíl, který může být způsoben odchylkou při měření. Ovšem v případě, kdy jsem vytvořil obyčejné indexy nad „pos.loc“, a poté nad „pos.time“, se mi podařilo dosáhnout výsledků až o 61 % rychlejších než původní dotazy.

Největšího překvapení jsem se dočkal po vytvoření prostorového *2d* indexu. Ve výpise programu *mongo*, popisující aktuální činnost samotné databáze, se změnil popis z klasického dotazu na „*GEO_2D*“ a vyhledávání se mnohonásobně zpomalilo. To si vysvětluji pouze tím, že popisovaný *2d* index je určen pro geoprostorová data obsahující souřadnice Longitude a Latitude (zeměpisná šířka a délka). Bohužel jsem se pokusil tento index využít i pro indexování souřadnic určených pouze celými čísly. V případě takto malého rozlišení je ovšem počítání pomocí GeoHash-e[2], který je využívám v případě *2d* indexu, velmi neefektivní.

7.2 Zhodnocení experimentů

Z výsledků experimentů nad MongoDB lze vyčíst několik zajímavých informací. Dle mého názoru nejdůležitější z nich je ta, týkající se rychlosti dotazů nad různě uloženými daty

[2] `http://docs.mongodb.org/manual/core/geospatial-indexes/#geospatial-indexes-geohash`

a indexy nad nimi vytvořenými. A to především z důvodu, že import dat, vytvoření indexu i hromadné mazání dokumentů se bude pravděpodobně provádět pouze jedenkrát či velmi výjimečně. Na druhou stranu dotazování nad uloženými daty považuji za velmi častou operaci a primární důvod, proč tyto databáze vůbec používáme.

Z experimentálně měřených vyhledávacích dotazů vyšlo jako jednoznačně nejlepší řešení to, kdy byly souřadnice bodu ukládány v podobě dvojice *[x, y]*, nad touto dvojicí byl vytvořen samostatný index, a dále byl vytvořen samostatný index nad položkou času. V tomto případě se totiž jedná nejenom o časově nejrychlejší řešení, ale máme také k dispozici operátor *$geoWithin*, díky kterému je dotazování na prostorové souřadnice velmi pohodlné. Lze také využít kromě operátoru *$box*, také další jako *$center* a *$polygon* a vyhledávat tak objekty procházející kruhem či polygonem. To by, v případě kdy bychom definovali rozsah ručně, bylo velmi obtížné. Pro mě docela překvapivě toto řešení dokonce porazilo případ, kdy jsou souřadnice x a y uloženy odděleně a jejich rozsah v dotazu kontrolujeme sami. Jedním z možných vysvětlení je nějaká optimalizace dotazu ze strany MongoDB v případě použití operátoru *$geoWithin*. Druhé, a možná pravděpodobnější, vysvětlení je nepříliš optimalizovaná práce se složenými indexy. V případě obou formátů dat se totiž jako rychlejší řešení ukázalo vytvoření několika samostatných indexů, než jednoho složeného. A jelikož v případě dotazování nad souřadnicemi uloženými jako *[x, y]* šlo o jednoduchý index a v případě, kdy byly souřadnice rozděleny do dvou položek, o index složený, tak je to docela pravděpodobné vysvětlení.

Poslední experimentálně zjištěnou informací je téměř naprostá nepoužitelnost prostorového indexu *2d*, a prostorových indexů v MongoDB obecně, pro mnou testovaná vstupní data. Tyto indexy se ukázaly jako nevhodné pro použití nad objekty, které se pohybují pouze po celočíselných souřadnicích s poměrně malým rozsahem hodnot. V případě, kdy by vstupními daty byl například GPS záznam pohybujících se vozidel, tak se domnívám, že výsledek těchto experimentů by byl zcela jiný. V takovém případě by teprve vynikl smysl prostorového indexu (ať už *2d* nebo *2dSphere*) a naopak by velmi začala ztrácet na výkonu ostatní popisovaná řešení, která by místo práce s nízkými celými čísly, musela pracovat s poměrně rozsáhlými desetinnými čísly používanými pro zeměpisnou šířku a délku.

Kapitola 8

Demostrační program

Kromě experimentování s ukládáním dat v MongoDB, jsem v rámci této práce vytvořil také velmi jednoduchý demonstrační program. Ten je napsána v jazyce Python 3 a je k nalezení na přiloženém CD v `/scripts/mongodb.py`. Detaily použití jsou zobrazeny po spuštění s parametrem $--$*help*.

Příklad použití:

```
1    >mongodb.py --x1 383 --x2 385 --y1 120 --y2 150 --t1 20000 --t2 25000
2
3    Vyhledávací parametry: x=(383, 385), y=(120, 150), t=(20000, 25000)
4    Nalezených objektů: 14
5    Track_id: 20, 687, 694, 713, 738, 742, 747, 758, 763, 778, 785, 813, 831, 839
6    Process finished with exit code 0
```

Program má jednoduchou funkčnost, kdy podle zadaných parametrů umí poslat do MongoDB jeden ze tří dotazů (porovnávající pouze čas, porovnávající pouze prostorové souřadnice nebo porovnávající čas i prostorové souřadnice) a zobrazit počet vrácených dokumentů a jejich ID. Jelikož se má jednat pouze o čistě demonstrační program bez možnosti praktického nasazení, tak je v kódu napevno vybrána databáze a kolekce.

Cílem tohoto programu je čtenáři ukázat základy práce s MongoDB pomocí API. V tomto případě konkrétně pomocí knihovny *PyMongo*[1]. Sám jsem si tak prakticky ověřil, že pokládání dotazů je velmi podobné JavaScript API, které jsem používal v příkazové řádce pro experimentování.

[1] `http://api.mongodb.org/python/`

Kapitola 9

Závěr

Cílem mé bakalářské práce bylo seznámit se s problematikou trajektorií vícerozměrných objektů a NoSQL databázemi. Následně poté bylo mým úkolem prozkoumat či navrhnout možné způsoby ukládání trajektorií pohybujících se 2D objektů a dotazování se nad nimi v NoSQL databázích.

Důležitá část práce spočívala v prozkoumání několika NoSQL databází, které jsou schopny pracovat s vícerozměrnými daty a indexy nad nimi. Na základě tohoto rozboru z kapitoly 5, jsem provedl návrh možného způsobu ukládání a práce s daty trajektorií nad vybranými databázemi. Vzhledem k tomu, že byla jako nejlepší možné řešení vybrána jediná databáze - MongoDB, jsem s ní v kapitole 7.1 provedl řadu experimentů nad dodanými trajektoriemi. Cílem těchto experimentů bylo nalézt nejvhodnější způsob, jak lze uložit zmíněné trajektorie a vytvořit nad nimi indexy. V kapitole 7.2 jsem provedl zhodnocení těchto experimentů, ze kterých jako nejzajímavější výsledek vyšla informace o nevhodnosti popisovaného prostorového indexu pro moje vstupní data. Také zde ale bylo dokázáno, že MongoDB má prostředky pro vhodné uložení a práci nad těmito vstupními daty, kdy se mi podařilo vhodným uložením a indexováním zrychlit vyhledávací dotazy o více než 60 %.

Přínos této práce vidím kromě seznámení čtenáře s problematikou a návrhu několika možných způsobů ukládání a práce s daty trajektorií, také v praktickém ověření možností použití prostorových indexů v MongoDB a nalezení nejvhodnějšího řešení, vhodného i pro reálné nasazení.

Jako možné rozšíření této práce bych navrhoval praktické ověření teoretických návrhů nastíněných v kapitole 6.3, neboť srovnání výkonnosti CouchDB a MongoDB nad reálně používanými dotazy, by mohlo přinést zajímavé poznatky do problematiky ukládání vícerozměrných dat.

Literatura

[1] Oracle. Spatial developer's guide. http://docs.oracle.com/cd/E11882_01/appdev.112/e11830/sdo_intro.htm#SPATL010, 2009 [cit. 2014-03].

[2] Elias Frentzos. Trajectory data management in moving object databases - phd thesis. http://infolab.cs.unipi.gr/pubs/theses/Frentzos_PhD_Thesis_EN.pdf, 2008-07 [cit. 2014-04].

[3] Carlo Strozzi. Nosql - a relational database management system. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page, 1998 [cit. 2014-04].

[4] List of nosql databases. http://nosql-database.org/, 2009 [cit. 2014-04].

[5] Edgar F. Codd. A relational model of data for large shared data banks. http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf, 1970-07 [cit. 2014-04].

[6] Eric A. Brewer. Towards robust distributed systems. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf, 2000-07-19 [cit. 2014-04].

[7] Seth Gilbert, Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf, 2002 [cit. 2014-05].

[8] Paul Beynon-Davies. *Database Systems.* Palgrave Macmillan, 2004. ISBN 1-4039-1601-2.

[9] Antotin Guttman. *R-trees: a dynamic index structure for spatial searching.* ACM New York, 1984. ISBN:0-89791-128-8.

[10] Tomáš Volf. Indexování časoprostorových dat. http://www.fit.vutbr.cz/study/courses/VPD/public/1112VPD-Volf.pdf, 2012 [cit. 2014-04].

[11] Nick Johnson. Damn cool algorithms: Spatial indexing with quadtrees and hilbert curves. http://blog.notdot.net/2009/11/Damn-Cool-Algorithms-Spatial-indexing-with-Quadtrees-and-Hilbert-Curves, 2009-09-11 [cit. 2014-05].

[12] Binary json. http://bsonspec.org/, Dostupné online [cit. 2014-05].

[13] Nicholas Knize. Rtree spatial indexing with mongodb. http://www.slideshare.net/nknize/rtree-spatial-indexing-with-mongodb-mongodc, 2012-07-11 [cit. 2014-05].

[14] Samuel Bosch. Spatial indexing a mongodb with rtree. http://www.samuelbosch.com/2009/06/spatial-indexing-mongodb-with-rtree.html, 2009-06-06 [cit. 2014-05].

Dodatek A

Obsah CD

- Skript pro převod formátu SQL do JSON
- Demonstrační program
- Tabulka s naměřenými hodnotami použitými v kapitole 7.1
- Vstupní data
- Vstupní data převedená do formátu [x, y] i do formátu čísel