Masarykova univerzita
Fakulta informatiky


# Implementace fulltextového vyhledávání v systému správy požadavků

Bakalářská práce

**Jiří Holuša**

Brno, Jaro 2014

## Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Jiří Holuša

**Vedoucí práce:** Mgr. Filip Nguyen

# Poděkování

TODO: poděkování

## Shrnutí

TODO: abstrakt

## Klíčová slova

TODO: klíčová slova

# Obsah

# 1 Úvod

Úvod

# 2 Vyhledávání

Tato kapitola stručně popisuje způsob vyhledávání skrze SQL v nejčastějším datovém úložišti – relačních databázích – a uvádí jeho nedostatky. Poté se detailněji věnuje jednou z možností jejich řešení, a to fulltextovým vyhledáváním. Uvádí nezbytnou teorii k pochopení principů, jak fulltextové vyhledávání funguje.

## 2.1 Vyhledávání pomocí SQL

Relační databáze poskytují vysoce výkonný přístup k datům a široké možnosti pro jejich správu. Díky svých schopnostem se staly nejpoužívanější technologií pro datové uložiště. Pro komunikaci s nimi se využívá jazyk SQL, který nabízí pro vyhledávání v datech pouze dva způsoby: porovnání obsahu buňky a operátor `LIKE` [1].

Porovnání obsahu buňky funguje na velice jednoduchém principu úplné shody obsahu. Na obrázku refSQLexample1 vidíme dotaz v jazyce SQL, který vybere právě ty záznamy z tabulky `People`, které mají hodnotu atributu `name` rovnou „Bruce Banner“.

```
SELECT * FROM People WHERE name = 'Bruce Banner'
```

**Obrázek 2.1:** Jednoduché použití SQL pro vyhledávání pomocí úplné shody obsahu pole

Nebudou vybrány žádné jiné záznamy, přestože by obsah atributu `name` byl např. „Bruce Banners“ či dokonce ani „Bruce Banner “ (přebytečná mezera na konci). Výhodou tohoto řešení je efektivita a jednoduchost – jediná nutná operace je pouze porovnání dvou řetězců, žádné dodatečné zpracování není potřeba.

Trochu více sofistikovaným způsobem je operator `LIKE`, který umožňuje (v omezené míře) používat vyhledávání pomocí vzoru (*pattern matching*). Podporovány jsou tzv. zástupné symboly (*wildcards*), jež mohou mít v tomto kontextu jiný význam než jen právě daný znak, např. symbol % (procento) zastupuje libovolnou sekvenci znaků (třeba i žádnou) nebo znak _ (podtržítko) libovolný, ale právě jeden znak. Obrázek 2.2 ukazuje příklad SQL dotazu, jenž nám vrátí všechny záznamy z tabulky `People`, které jejich jméno končí na „Banner“.

Nyní již dokážeme tímto dotazem získat jak lidi se jménem „Bruce Banner“, tak i „Richard Banner“.

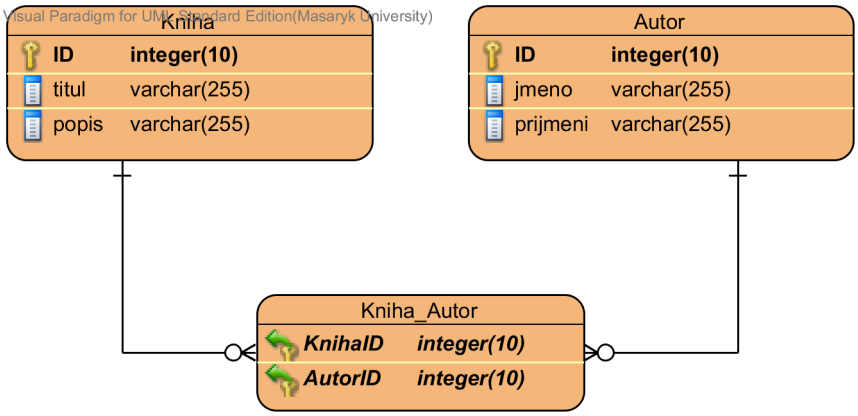
```
SELECT * FROM People WHERE name LIKE '%Banner'
```

**Obrázek 2.2:** Použití SQL operátoru LIKE

## 2.2 Problémy vyhledávání pomocí SQL

Předchozí kapitola představila základní způsoby vyhledávání pomocí SQL. Nyní se podíváme na případy, kde nám tyto způsoby nestačí nebo si s danou situací nedokáží poradit buď vůbec, nebo pouze neefektivně.

Abychom si mohli tyto nedostatky demostrovat na příkladech, uvažujme existenci jednoduché relační databáze s následujícím schématem (obrázek 2.3).




**Obrázek 2.3:** Datový model ukázkové databáze

**Vyhledávání přes několik tabulek** Představme si, že uživatel zadá do vyhledávacího políčka nějaký řetězec, na jehož základě očekává odpovídající výsledky. Vyvstává otázka, kde bychom měli zadanou frázi hledat. V našem případě pravděpodobně v nadpisu, popisu, ve jméně a příjmení autora, tam všude by se mohly nacházet informace, které uživatel hledal.

SQL nyní musí prohledat všechny zadané sloupce, které se však mohou nacházet v různých tabulkách, což vede ke spojování tabulek. Možný příklad výsledného dotazu vidíme na obrázku 2.4.

Je vidět, že i při relativně jednoduchém požadavku (vyhledáváme pouze ve čtyřech sloupcích) je výsledný dotaz poměrně složitý. Pokud bychom chtěli uživateli dát možnost využívat komplexnější dotazy, je otázka generování

```
SELECT *
FROM Kniha kniha
LEFT JOIN kniha.autor autor
WHERE kniha.titul = ? OR kniha.popis = ? OR
autor.jmeno= ? OR autor.prijmeni= ?
```

**Obrázek 2.4:** SQL dotaz vyhledávající přes několik tabulek

odpovídajích SQL dotazů netriviální. Navíc uvažme, že musíme spojit více tabulek, což může vést k problémům s efektivitou [2, s. 9].

**Vyhledávání jednotlivých slov** Kapitola 2.1 ukázala, že SQL dokáže vyhledat v jednotlivých sloupcích přesně zadanou frázi. Je ovšem velice nepravděpodobné, že sloupce v databázi budou obsahovat přesně stejnou danou frázi, hledání jednotlivých slov by nám velice zvýšilo pravděpodobnost nálezu [2, s. 9]. SQL však žádnou takovou funkcionalitu na dělení vět neposkytuje, je tedy nutné si větu předpřipravit explicitně (tj. rozdělit na slova), a poté spouštět vyhledávácí dotaz pro každé slovo zvlášť. Následně výsledky nějakým způsobem sloučit. Takové řešení však nebude dostatečně efektivní [2, s. 10].

**Filtrace šumu** Některá slova ve větách nenesou vzhledem k vyhledávání žádnou informační hodnotu, např. spojky či předložky či ještě lepším příkladem mohou být anglické neurčité členy. Taková slova se nazývají šum (*noise*). Dále se pak některá slova v určitém kontextu šumem stávají, např. slovo „kniha“ v našem internetovém knihkupectví [2, s. 9]. Jelikož šum nenese žádnou informační hodnotu, měl by být při hledání ignorován. SQL nám opět neposkytuje žádný prostředek k řešení tohoto problému.

**Vyhledávání příbuzných slov** Je velice žádoucí, abychom se při vyhledávání mohli zaměřit pouze na význam hledaného slova, nikoliv na jeho tvar. Nemělo by záležet na tom, zda hledáme frázi „fulltextové hledání“ nebo „fulltextových vyhledávání“, význam těchto frázi je stejný. Jinak řečeno, vyhledávání by mělo brát v potaz i slova odvozená, se stejným kořenem. Ještě pokročilejším požadavkem by mohla být možnost zaměňovat slova s jejich synonymy, např. „upravit“ a „editovat“ [2, s. 10].

SQL nám nenabízí možnost k řešení těchto požadavků, klíčem by mohl být slovník příbuzných slov a synonym a pokusit se vyhledávat i podle něj. Takové řešení však přináší nezanedbatelné množství práce, nehledě na nutnost existence takového slovníku.

**Oprava překlepů** Uživatel je člověk a jako člověk je omylný a dělá chyby. Vyhledávání by to mělo brát v potaz a snažit se tyto překlepy opravit či uhodnout, co měl uživatel na mysli. Když v našem internetovém knihkupectví uživatel hledá knihu „Fulltextové vyhledávání“ a omylem zadá do vyhledávacího pole „Fulltetové vyhledávání“, je žádoucí, aby i přes tento překlep knihu našel [2, s. 10].

**Relevance** Pravděpodobně největším problémem v SQL je absence jakéhokoliv mechanismu pro určení míry shody (*relevance*) záznamu se zadaným dotazem [2, s. 10]. Předpokládejme, že v našem knihkupectví napsal autor „John Smith“ 100 knih, jednu o fulltextovém vyhledávání a zbytek naprosto nesouvisející s informatikou. Dále několik dalších autorů rovněž napsalo publikace na téma fulltextového vyhledávání.

Pokud jako uživatel víme, že je autorem John Smith a kniha je o fulltextovém vyhledávání, očekáváme, že na vyhledávací dotaz „John Smith fulltextové vyhledávání“ obdržíme nejdříve právě chtěnou knihu, a poté teprve knihy ostatní od našeho autora či další knihy o fulltextovém vyhledávání, jelikož naše kniha „nejvíce“ odpovídala položenému dotazu.

## 2.3 Fulltextové vyhledávání

Předchozí kapitola demostrovala, jaké problémy má vyhledávání pomocí SQL. Nyní si bude představeno možné řešení – fulltextové vyhledávání.

### 2.3.1 Úvod do fulltextového vyhledávání

Fulltextové vyhledávání (někdy také *fulltext* nebo *full-text*) je speciální způsob vyhledávání informací v textu. Vyhledávání probíhá porovnáváním s každým slovem v hledaném textu. Jelikož počet slov v textu může teoreticky neomezený a jelikož je nutné, aby vyhledávání bylo co nejrychlejší, funguje fulltextové vyhledávání ve dvou fázích: *indexace* a *hledání* [2, s. 11].

### 2.3.2 Indexace

Indexace je hlavním krokem ve fulltextovém vyhledávání. Jedná se o proces předpřipravení vstupních dat, jejich přeměnu na co nejvíce efektivní datovou strukturu, aby se v ní dalo snadno a rychle vyhledávat. Této datové struktuře, která je výstupem indexace, se říká *index* [3, s. 11].

Index si lze představit jako datovou struktu umožňující přímý přístup ke slovům v něm obsažených. Základním úkolem je rozdělit text do slov a

pomocí přímého přístupu umožnit velice efektivně zjistit, kde se dané slovo vyskytuje. Toho je typicky (např. v Apache Lucene) dosaženo *invertovaným indexem* [3, s. 35].

Pouhým rozdělením do slov však možnosti předpřipravení textu nekončí a může být zapojena složitá analýza. V praxi (např. v Apache Lucene [3, s. 35]) je celý text předáván analyzátoru, který může index libovolně budovat, a tím ho lépe připravit na nadcházející dotazování, a umožnit mu odpovídat na složitější dotazy. Typickým příkladem možné analýzy je úprava podstatných jmen do základního tvaru (např. z množného čísla na jednotné), přidání synonym do indexu či získávání statistiky o četnosti výskytu daného slova.

### 2.3.3 Hledání

Samotné vyhledávání v textu je ve fulltextovém vyhledávání realizováno nikoliv nad textem samotným, ale nad předpřipraveným indexem z procesu indexace [2, s. 15]. Vyhledávácí nástroj tedy může využít doplňkových informací o textu, které dokáží vyhledávání zrychlit. Jakým způsobem je index budován a jak se nad ním následně vyhledává, záleží pak již na konkrétní technologii.

# 3 Dostupné technologie

Pro platformu Java existuje řada dostupných volně širitelných vyhledávacích technologií. Nyní si představíme tři z nich: *Apache Lucene*, *Hibernate Search* a *Elasticsearch*.

## 3.1 Apache Lucene

Apache Lucene je vysoce výkonná, škálovatelná, volně širitelná vyhledávací knihovna napsána v jazyce Java [3, s. 6]. Autorem projektu, který vznikl v roce 1997, je Doug Cutting. Zajímavostí je, že jméno Lucene bylo vybráno podle druhé jména manželky autora [3, s. 6]. V roce 2000 zveřejnil Lucene na stránkách serveru SourceForge.com a uvolnil ji tak zdarma pro komunitu. O rok později byla adoptována organizací *Apache Software Foundation*. Od té doby se knihovna neustále vyvíjela a v dubnu roku 2014 je aktuálně dostupná ve své nejnovější verzi 4.7.1 [3, s. 6].

Již několik let je Lucene nejpopulárnější vyhledávací technologií zdarma. Díky své popularitě se však dočkala i přepsání do jiných jazyků než je Java jako například Perl, Python, Ruby, C/C++, PHP a C# (.NET) [3, s. 3]. Projekt je stále aktivně vyvíjen s širokou komunitní základnou.

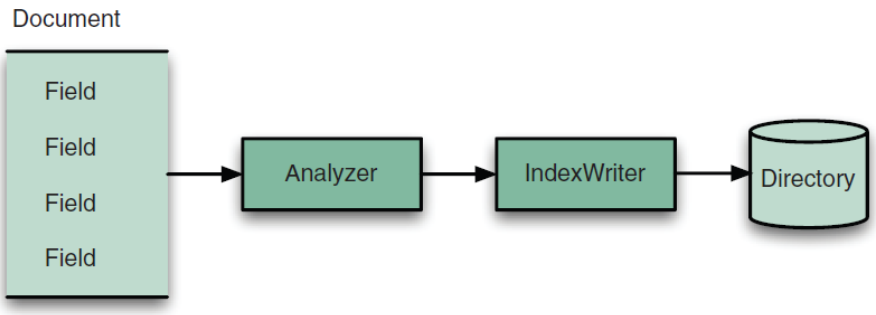
Apache Lucene není hotová vyhledávací aplikace, je to knihovna, nástroj, poskytující všechny potřebné prostředky, aby mohla být taková aplikace pro vyhledávání naprogramována. Nabízí rozhraní pro vytváření, úpravu indexu, zpracování dat před indexací a tvorbu, úpravu dotazů a mnoho dalšího. O zbytek úkonů se musí programátor postarat sám, z čehož vyplývají hlavní výhoda (robustnost, univerzálnost použití), ale také hlavní nevýhoda (složitost nasazení) [3, s. 7].

Používání Apache Lucene je poměrně náročné, což vychází z její univerzálnosti [5] – uživatel (programátor) má mnoho možností, jak výslednou vyhledávací aplikaci nakonfigurovat, a tím i vyladit. Kvůli této složitosti začaly vznikat další technologie, které staví na Apache Lucene, snaží se schovat podrobná, a tedy i méně často používaná, nastavení do pozadí a umožnit tak vývojáři se v technologii rychle zorientovat se zachováním původní síly Apache Lucene. Takových technologií existuje více (Apache Solr, Hibernate Search, Elasticsearch a další) a je dobré při jejich používání vědět, jak funguje Apache Lucene na nižší úrovni, neboť tyto technologie ji přímo využívají. Z toho důvodu si podrobněji představíme architekturu Apache Lucene, abychom poznali její sílu a možnosti.

### 3.1.1 Architektura

Abychom pochopili, jak Apache Lucene funguje, představíme si zběžně její architekturu. Níže následuje výčet základních tříd, které se podílejí na procesu indexace [3, s. 26]:

- `IndexWriter`
- `Directory`
- `Analyzer`
- `Document`
- `Field`




**Obrázek 3.1:** Architektura indexační části Apache Lucene, převzato z [3, s. 26]

Třída `IndexWriter` je vstupní bod indexace. Je zodpovědná za vytváření nového indexu a přidávání dokumentů do indexů existujících. Neslouží k vyhledávání ani modifikaci indexu. `IndexWriter` musí znát umístění, kam má svůj index uložit a k tomu slouží `Directory`.

`Directory` je abstraktní třída reprezentující fyzické umístění indexu.

Předtím než je text indexován, je předán analyzéru, implementaci abstraktní třídy `Analyzer`. Analyzér je zodpovědný za extrakci *tokenů* – jednotek, které následně budou skutečně uloženy do indexu [3, s. 116] – a eliminaci všeho ostatního. Analyzér je patrně nejdůležitější komponenta indexace, rozhoduje, které tokeny budou uloženy a dokáže je libovolně modifikovat. Apache Lucene obsahuje již některé praktické implementace třídy `Analyzer`, které jsou nejběžnější. Některé z nich se například zabývají odstraněním šumu z textu, další převedením všech písmen na malá apod. Proces analýzy

podrobněji rozebíráme v další kapitole, neboť je to klíčová vlastnost Apache Lucene, kterou dědí i ostatní technologie na ní postavené.

`Document` reprezentuje kolekci polí (*fields*), je to kontejner pro objekty `Field`, které nesou textová data.

`Field` je základní jednotka, která obsahuje vlastní indexovaný text.

Jak je uvedeno v kapitole 2.3.1, fulltextové vyhledávání má dvě části – indexaci a vyhledávání. Protože však technologie postavené na Apache Lucene poskytují své vlastní vyhledávací API, a tím skrývájí vyhledávání v Apache Lucene úplně, nebudeme se detaily architektury vyhledávání v Apache Lucene dále zabývat.

### 3.1.2 Indexace

Předchozí kapitola stručně popisuje architekturu indexační části Apache Lucene. Nyní si vysvětlíme, jak spolu jednotlivé části spolupracují.

Základní jednotkou indexu Apache Lucene jsou *dokumenty* (*directories*) a *pole* (*fields*) [3, s. 32]. Dokument je kolekcí polí, které pak obsahují „skutečný“ obsah. Každé pole má své jméno, textovou nebo binární hodnotu a seznam operací, které popisují, co má Apache Lucene dělat s hodnotou pole při vytváření indexu. Abychom mohli indexovat naše uživatelská data (položky z databáze, PDF dokumenty, HTML stránky apod.), je potřeba je převést do formátu Apache Lucene dokumentu. Apache Lucene nemá ponětí o sématice obsahu, který indexuje. Převedením struktury našeho obsahu do struktury Lucene dokumentů, do dvojic klíč:hodnota, se zabývá *denormalizace*.

**Denormalizace** Denormalizace je proces převedení libovolné struktury dat do jednoduchého formátu klíč:hodnota [3, s. 34]. Například v databázi jsou jednotlivé záznamy spojovány cizími klíči mezi různými tabulkami, vzniká mezi nimi vztah, jednotlivé záznamy se na sebe odkazují. V dokumentech Apache Lucene však žádná možnost odkazu či spojení není, jediný akceptovaný formát je, jak jsme si řekli, klíč:hodnota. Programátor musí vyřešit problém, jak data, ve kterých chce vyhledávat, denormalizuje. Apache Lucene nechává tuto část zcela na programátorovi, na rozdíl od na ní postavených technologiích jako např. Hibernate Search.

Jednou z dalších důležitých věcí, které je potřeba vědět o Apache Lucene dokumentech, je absence jakéhokoliv pevného schématu jako např. u databází. Tato vlastnost se někdy označuje jako *flexibilní schéma* [3, s. 34]. Umožňuje nám například iterativně budovat index, protože nově nahraný index může být naprosto rozdílný, obsahovat jiná pole, od předchozího. Rovněž můžeme do jednoho dokumentu uložit indexy reprezentující zcela jiné entity.

### 3.1.3 Analýza

V předchozích kapitolách jsou uvedeny fundamentální základy, na kterých Apache Lucene staví indexy, nyní se blíže podíváme nejdůležitější část indexačního procesu – analýzu.

Předpokládejme, že vstupní data máme již denormalizována do dokumentů, které jsou naplněny poli. Analýza v Apache Lucene je proces převedení textových polí do základní indexované podoby – do termů [3, s. 28]. Analyzérem nazýváme komponentu, která zajišťuje analýzu. Ukažme si několik typických příkladů, co analyzéry dělají [3, s. 110]:

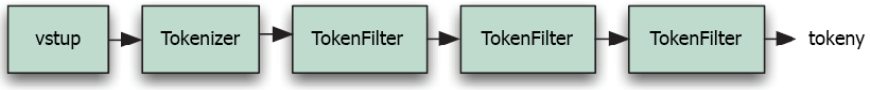
- extrakce slov
- zahození interpunkce
- převod na malá písmena (*normalizace*)
- redukce šumu
- převod slova na jeho kořen (*stemming*)
- převod slova na základní tvar (*lemmatizace*)
- a další

Samozřejmě je možné naprogramovat vlastní analyzér, některé úkony jsou však natolik běžné (jako například výše uvedené), že Apache Lucene přichází s několika zabudovanými analyzéry. Analyzéry pro svou funkčnost využívají dva další typy komponent: *tokenizéry* (potomky třídy `Tokenizer`) a *filtry* (potomky třídy `TokenFilter`) [3, s. 115]. Obě dědí od abstraktní třídy `TokenStream`, zabývají se však rozdílnou částí zpracování vstupu. Tokenizér čte vstup a vytváří tokeny. Filtr bere jako vstup tokeny a na jejich základě vrátí nově vytvořený seznam tokenů. Tento seznam může vzniknout přidáním nových tokenů, úpravou existujících či odstraněním některých z nich.

Typické využití, kterého se drží i zabudované analyzéry, vypadá následovně. Analyzéru je předán vstup. Ten je rozdělen na tokeny pomocí jednoho tokenizéru. Následně jsou tokeny předány jednomu či více filtrům, čímž vznikne finální kolekce tokenů, která je předána jako výsledek analýzy (obrázek 3.2).

Uveďme si příklady zabudovaných tokenizérů [3, s. 118]:

- `WhitespaceTokenizer` - nový token je ohraničen bílými znaky
- `KeywordTokenizer` - předá celý vstup jako jeden token




**Obrázek 3.2:** Použití tokenizéru a filtrů, převzato z [3, s. 117]

- `LowerCaseTokenizer` - nový token je ohraničen jinými znaky než písmeny
- `StandardTokenizer` - pokročilý tokenizér založený na sofistikovaných gramatických pravidel, dokáže rozpoznat např. e-mailové adresy a předat je jako jediný token

Představme si rovněž i několik základních filtrů [3, s. 118]

- `LowerCaseFilter` - převede token na malá písmena
- `StopFilter` - odstraní tokeny, které se nacházejí v předaném seznamu
- `PorterStemFilter` - převádí tokeny na jejich kořen (*stemming*)
- `LengthFilter` - akceptuje tokeny, jejichž délka spadá do určitého rozsahu
- `StandardFilter` - navržen pro spolupráci s tokenizérem StandardTokenizer, odstraňuje tečky z akronymů a „'s“ (apostrof následovaný písmenem s)

Aby byl výčet kompletní, následuje přehled zabudovaných analyzérů. Zabudované analyzéry jsou v podstatě kombinací tokenizérů a filtrů, z čehož je následně jasná jejich funkce [3, s. 112].

- `WhitespaceAnalyzer` - dělí text na tokeny pomocí tokenizéru `WhitespaceTokenizer`
- `SimpleAnalyzer` - zpracovává vstup pomocí tokenizéru `LowerCaseTokenizer`
- `StopAnalyzer` - kombinace tokenizéru `LowerCaseTokenizer` a filtru `StopFilter`, kterému je předán seznam často se vyskytujících nevýznamových slov v angličtině (členy *a*, *an*, *the*, apod.)

```
WhitespaceAnalyzer:
[The] [quick] [brown] [fox] [jumped]
[over] [the] [lazy] [dog]

SimpleAnalyzer:
[the] [quick] [brown] [fox] [jumped]
[over] [the] [lazy] [dog]

StopAnalyzer:
[quick] [brown] [fox] [jumped] [over] [lazy] [dog]

StandardAnalyzer:
[quick] [brown] [fox] [jumped] [over] [lazy] [dog]
```

**Obrázek 3.3:** Použití zabudovaných analyzérů pro větu *„The quick brown fox jumped over the lazy dog“*

- `StandardAnalyzer` - nejpropracovanější zabudovaný analyzér, využívá `LowerCaseTokenizer`, `StopFilter`, navíc však přidává i propracovanou logiku, která dokáže např. rozeznat e-mailové adresy, názvy společností atd.

Popis analýzy zakončeme ukázkami, jaké tokeny jednotlivé zabudované analyzéry vytvoří ze dvou anglických vět *„The quick brown fox jumped over the lazy dog“* (obrázek 3.3) a *„XY&Z Corporation - xyz@example.com“* (obrázek 3.4).

## 3.2 Hibernate Search

Po rozmachu technologie *objektově relačního mapování* (ORM, *Object-Relational Mapping*) na platformě Java a její nejznámější implementace Hibernate Core [2, s. 29] bylo nutné rovněž dát tomuto nástroji možnosti fulltextového vyhledávání, o což se postarala právě knihovna Hibernate Search. Hibernate Search je volně šiřitelná knihovna napsaná Emmanuelem Bernardem, která doplňuje Hibernate Core o možnosti fulltextového vyhledávání pomocí kombinace s Apache Lucene [2, s. 29]. Hibernate Search se snaží zabalit komplexnost Apache Lucene do jednoduší podoby a integrovat funkčnost do Hibernate ORM. S minimálním úsilím za nás řeší převod objektového datového modelu do podoby přijatelné pro Apache Lucene, čímž výrazně usnadňuje její použití.

```
WhitespaceAnalyzer:
[XY&Z] [Corporation] [-] [xyz@example.com]

SimpleAnalyzer:
[xy] [z] [corporation] [xyz] [example] [com]

StopAnalyzer:
[xy] [z] [corporation] [xyz] [example] [com]

StandardAnalyzer:
[xy&z] [corporation] [xyz@example.com]
```

**Obrázek 3.4:** Použití zabudovaných analyzérů pro větu „*XY&Z Corporation - xyz@example.com*“

Obrázek 3.5 demonstruje, jak snadno lze s využitím Hibernate Search zpřístupnit entitu pro fulltextové vyhledávání. Entita musí být označena anotací `@Indexed` [2, s. 38]. Dále přidáme anotaci `@DocumentId` k primárnímu klíči, a poté označíme atributy, podle kterých chceme vyhledávat anotací `@Field` [2, s. 38]. V momentě uložení entity za nás Hibernate Search vyřeší přidání uvedených atributů do indexu, tedy denormalizuje entitu. Jelikož je to však pod povrchem stále Apache Lucene, máme k dispozici všechny možnosti, které nám nabízí, nyní v přístupnější formě.

Integrace s Hibernate Core za nás elegantně řeší jeden podstatný problém, který máme s použitím čistě Apache Lucene - synchronizaci fulltextového indexu a obsahu databáze. Jsou to v zásadě dvě zcela oddělená datová úložiště, která spolu úzce souvisí. Pokud používáme přímo Apache Lucene, je nutné se po manipulaci s objektem v databázi explicitně postarat o úpravu příslušného indexu, což je pro programátora práce navíc. Oproti tomu Hibernate Search je navázán na události Hibernate Core, tudíž při úpravě objektu v databázi je automaticky spuštěn proces aktualizace indexu, aby spolu byla data v databázi a fulltextovém indexu synchronizována [2, s. 24].

## 3.3 Elasticsearch

Elasticsearch je distribuovaný vyhledávácí a analytický nástroj v reálném čase [5]. Historie této technologie se začala psát v roce 2004, kdy Shay Banon vytvořil *Compass*. Postupným vývojem a změnou požadavků však dospěl k názoru, že aby se mohl Compass stát distribuovanou technologií, bylo

```
@Entity
@Indexed
public class Person {

        @Id     @GeneratedValue
        @DocumentId
        private Long id;

        @Field
        private String firstName;

        @Field
        private String lastName;
}
```

**Obrázek 3.5:** Zpřístupnění entity pro vyhledávání v Hibernate Search

by zapotřebí ho značnou část přepsat. Rozhodl se proto naprogramovat zcela nový nástroj, který měl být již od počátku distribuovaný. První verze Elasticsearch byla vydána v únoru 2010 [6].

Elasticsearch je rovněž postaven na dříve představené technologii Apache Lucene, k níž však přidává další klíčové vlastnosti. Řeč je zejména o celé architektuře. Elasticsearch není na rozdíl od Apache Lucene knihovnou, Elasticsearch tvoří samostatný distribuovaný systém serverů, které na pozadí používají Apache Lucene, ovšem skrývají její složitost a poskytují služby v mnohem jednodušším uživatelském API. Velký důraz je kladen právě na distribuovanost celého systému, proto je Elasticsearch vysoce škálovatelný, schopný vytvořit klastr několika stovek serverů, a tím zajistit vysoký výkon i při několika petabajtech dat, což patří mezi hlavní přidanou hodnotu navrch k Apache Lucene [5].

Základním způsobem komunikace se serverem Elasticsearch je REST (*Representational State Transfer*) API posílající JSON (*JavaScript Object Notation*) objekty. Tím získáváme naprostou nezávislost na programovacím jazyku, komunikace může probíhat přímo i z příkazové řádky. Do některých jazyků, jako například Java, PHP, Python, však byli napsáni klienti, kteří umožňují komunikaci přímo z onoho jazyka. Stejně jako Hibernate Search je tedy i Elasticsearch zaobalená knihovna Apache Lucene s několika přidanými hodnotami [5].

# 4 Návrh a implementace

Předchozí kapitoly představily fulltextové vyhledávání a dostupné technologie pro jeho implementaci na platformě Java.

Nyní se věnujeme skutečné implementaci fulltextového vyhledávání v systému správy požadavků eShoe. Celou implementaci lze rozdělit na několik částí: výběr technologie, indexace, vyhledávání a import dat.

## 4.1 Specifikace požadavků

Specifikace

## 4.2 Návrh

Návrh

## 4.3 Použité technologie

### 4.3.1 JBoss AS

### 4.3.2 Maven

### 4.3.3 Git

### 4.3.4 Jackson

### 4.3.5 ANTLR

## 4.4 Implementace

### 4.4.1 Indexace

První z problémů, který je potřeba vyřešit, je zvládnutí procesu indexace, tedy denormalizaci entit do formátu, jež se dá přímo předat Elasticsearch serveru. Jak již víme, Elasticsearch příjímá JSON-like objekty, tedy dvojce klíč - hodnota. Jedním z prvních možných řešení indexace je prostá manuální tvorba indexu z entity pomocí get metod, tedy pro každou třídu je vytvořen mechanismus, který v předem daném pořadí předem dané atributy získá a vytvoří z nich index.

Nevýhoda tohoto řešení je zjevná - nulová flexibilita. Při každé úpravě entity je nutné dopsat odpovídající mechanismus, který upravený atribut denormalizuje. Navíc je toto řešení udělané přesně na míru tomuto projektu, resp. přesně danému modelu entit, tudíž není znovupoužitelné do budoucna.

Výhoda je ovšem rovněž zřejmá - jednoduchost. K naprogramování takového mechanismu není potřeba víc než základní znalost jazyka Java.

Rozhodli jsme se však ubírat jinou, sofistikovanější, cestou. Denormalizace je řešena pomocí anotací velice podobně jako v Hibernate Search. Myšlenkou bylo vytvořit samostatný projekt, který máambice se později začlenit přímo do projektu Elasticsearch. To však vyžaduje vytvořit robustní a široce použitelný nástroj, nikoliv na míru vytvořený pouze pro potřeby projektu eShoe.

Základem je použití anotací, což zaručuje vysokou eleganci a jednoduchost použití. Inspirace vzešla právě z Hibernate Search, tedy oanotovat datový model (entity) nově vytvořenými anotacemi tak, aby po předání takové entity našemu indexačnímu manažeru nástroj sám vytvořit příslušný JSON dokument, který by následně automaticky odeslal do Elasticsearch serveru. Tím se z pohledu klienta redukuje proces tvorby indexu na pouhé vybrání vlastností entit, podle kterých se následně bude vyhledávat a zavolání jediné metody, o vše ostatní se již nástroj postará sám.

Přesně podle této myšlenky byl založet nový projekt elasticsearch-annotations. Elasticsearch-annotations poskytuje velice malé veřejné API, což usnadňuje jeho použití. Jádrem projektu je třída IndexManager.

### 4.4.2 Vyhledávání

### 4.4.3 Uživatelské rozhraní

### 4.4.4 Import dat

# 5 Závěr

Závěr

# Literatura


[1] KOFLER, Michael. *Mistrovství v MySQL 5*. Vyd. 1. Překlad Jan Svoboda, Ondřej Baše, Jaroslav Černý. Brno: Computer Press, 2007, 805 s. ISBN 978-80-251-1502-2.

[2] BERNARD, Emmanuel a John GRIFFIN. *Hibernate search in action*. Greenwich, CT: Manning, c2009, xxiv, 463 p. ISBN 19-339-8864-9.

[3] MCCANDLESS, Michael, Erik HATCHER, Otis GOSPODNETIC a Otis GOSPODNETIC. *Lucene in action*. 2nd ed. Greenwich: Manning, c2010, xxxviii, 488 p. ISBN 19-339-8817-7.

[4] Lucene FAQ. *Lucene-java Wiki* [online]. 2004 [cit. 2014-05-04]. Dostupné z: `http://wiki.apache.org/lucene-java/LuceneFAQ`

[5] *Elasticsearch: The Definitive Guide* [online]. 2014 [cit. 2014-05-04]. Dostupné z: `http://www.elasticsearch.org/guide/en/elasticsearch/guide/current/`

[6] Elasticsearch. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-05-04]. Dostupné z: `http://en.wikipedia.org/wiki/Elasticsearch`