UNICORN COLLEGE

Katedra informačních technologií

BAKALÁŘSKÁ PRÁCE

Porovnání řešení fulltextového vyhledávání v aplikacích vyvíjených ve webovém frameworku Ruby on Rails

Autor: David Horsák

Vedoucí bakalářské práce: Ing. Miroslav Žďárský

2012 Praha

Čestné prohlášení

Prohlašuji, že jsem svou bakalářskou práci na téma *Porovnání řešení fulltextového vyhledávání ve webovém frameworku Ruby on Rails* vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím výhradně obdorné literatury a dalších informačních zdrojů, které jsou v práci citovány a jsou také uvedeny v seznamu literatury a použitých zdrojů.

Jako autor této bakalářské práce dále prohlašuji, že v souvislosti s jejím vytvoření jsem neporušil autorská práva třetích osob a jsem si plně vědom následků porušení ustanovení § 11 a následujících paragrafech autorského zákona č. 121/2000 Sb.

V Praze dne 1. 5. 2012 David Horsák

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Miroslavu Žďárskému za pomoc s výběrem tématu této práce a za rady, které mi velice ulehčily psaní této práce. Dále mu děkuji za nadšení pro fulltextové vyhledávací nástroje a databáze, které mi předal na svých přednáškách předmětu Návrh relačních databází.

Porovnání řešení fulltextového vyhledávání v aplikacích vyvíjených ve webovém frameworku Ruby on Rails

The comparison of full text search solutions for applications developed in Ruby on Rails web application framework

UNICORN COLLEGE

Abstrakt

Cílem této práce je porovnat vybrané fulltextové vyhledávací nástroje, které je možné jednoduše integrovat do Ruby on Rails aplikací. Nejprve jsou fulltextové nástroje představeny a poté je jejich použití demonstrováno na zvoleném příkladu. Čtenář je postupně seznamován s principy, konfigurací i prácí s těmito nástroji a jejich omezeními. V závěru je doporučen jeden fulltextový vyhledávací nástroj a tato volba je odůvodněna.

Klíčová slova: fulltextový vyhledávací nástroj, Apache Solr, ElasticSearch, PosgtreSQL, Ruby on Rails

Abstract

The goal of this thesis is to compare chosen full text search engines that are easy to integrate into Ruby on Rails application. The search engines are introduced in the beginning and demonstrated later on an example application. The reader is gradually presented the principles, required configuration, ease of work and constaraints of these engines. At the very end, one full text search engine is recommended and the reasons of this choice are given.

Keywords: full text search engine, Apache Solr, ElasticSearch, PosgreSQL, Ruby on Rails

Obsah

1. Úvod 8
2. Představení použitých technologií 9
  2.1. Ruby on Rails 9
  2.2. Fulltextové vyhledávací nástroje 11
    2.2.1. Apache Solr 11
    2.2.2. ElasticSearch 13
    2.2.3. PostgreSQL 16
3. Konfigurace nástrojů a infrastruktura 22
  3.1. Datový model aplikace 22
  3.2. Infrastruktura 24
  3.3. Zadání pro fulltextové vyhledávání 25
  3.4. Apache Solr 26
    3.4.1. Příprava Apache Solr a klienta 26
    3.4.2. Nastavení modelů 30
    3.4.3. Vyhledávání 32
  3.5. ElasticSearch 36
  3.6. PostgreSQL 43
  3.7. ElasticSearch 2 48
4. Závěr 52
5. Seznam použitých zdrojů 54
6. Seznam příkladů 57
7. Seznam obrázků 60
8. Seznam tabulek 60

1. Úvod

Fulltextové vyhledávání je činnost, bez které se od doby příchodu Googlu na trh neobejdeme. Většina z nás něco fulltextově vyhledává na Internetu prakticky každý den a pro mě, jako pro vývojaře, je to nutnost. Díky fulltextovému vyhledávání a správně položenému vyhledávacímu dotazu jsem schopný najít řešení problému, který mi brání v další práci, v rámci několika sekund, maximálně minut. Fulltextové vyhledávání tím skoro úplně vytlačilo internetové katalogy, které se ještě před několika lety používaly právě pro procházení Internetu a vyhledávání požadovaných informací.

Má motivace proč jsem si vybral toto téma byla velice jednoduchá. Vyvíjím webové aplikace ve framewoku Ruby on Rails a sám jsem zakladatelem společnosti, která provozuje sociální síť jménem Artvasion určenou pro umělce, galerie a lidi se zájmem o umění.

Dle KISSMETRICS uživatelé tráví na jedné webové stránce v průměru 4 minuty a 50 sekund a během návštěvy si prohlédnout 4,5 stránky [1]. Tyto hodnoty se budou určitě meziročně snižovat s tím, jak přibývá informací na Internetu. Jedinými možnostmi, jak zařídit, aby se uživatel zdržel na webových stránkách déle, je vylepšit jejich obsah, nabídnout uživateli jednoduchout navigaci a dát mu to, co ho zajímá a co chce vědět.

Role fulltextového vyhledávání na webové stránce z předchozího textu jasně vyplývá. Je potřeba uživatelům zobrazovat obsah, který chtějí vidět a který je zajímá. Pro mě to znamená to, že pokud uživatel přijde na Artvasion, které obsahuje profily uživatelů a jimi nahrané umění, a zadá do vyhledávání např. „street art“, musí se mu zobrazit umění, které je Street artem. Dále musí mít možnost upravit parametry vyhledávání o řazení, např. od nejnovějšího umění po nejstarší, od nejlépe hodnoceného umění po nejhůře, podle relevance dotazu, atd.

V rámci této bakalářské práce představím několik fulltextových vyhledávacích nástrojů, datový model vzorové aplikace napsané v Ruby on Rails, provedu potřebná nastavení a zhodnotím složitost implementace jednotlivých řešení. V závěru práce doporučím nástroj, se kterým se mi nejjednodušeji pracovalo, vyžadoval nejméně konfigurace a také jak dlouho jej trvalo zavézt do provozu.

2. Představení použitých technologií

V této teoretické částí mé práce představím důležité části webového frameworku Ruby on Rails potřebné pro pochopení a účely této práce. Následovat budou vybrané fulltextové vyhládávací nástroje. V závěru části ukážu kousek datového modelu Artvasion, na kterém budu provádět fulltextové výhledávání.

2.1. Ruby on Rails

Ruby on Rails (zkr. RoR) je framework pro tvorbu webových aplikací. Celý je napsán v programovacím jazyce Ruby. Ruby on Rails silně prosazuje přístup *convention over configuration* (konvence místo konfigurace), a tudíž jediný možný návrhový vzor, který používá, je Model-View-Controller.

```
application_root
|-- app
|     |-- assets
|     |-- controllers
|     |-- helpers
|     |-- models
|     +-- views
|
|-- config
|     |-- environments
|     +-- initializers
|
|-- db
|     +-- migrate
|
|-- lib
|-- log
|-- public
|-- script
|-- spec
|-- tmp
+-- vendor
```

Příklad č. 1 - Základní struktura složek Ruby on Rails aplikace

Tento přístup také silně ovlivňuje to, jak jsou třídy a metody pojmenovávány. Díky tomu se mohou vývojaři velice rychle zorientovat v cizích

projektech a jednoduše číst cizí kód. Vše umocňuje ActiveRecord, který RoR používá pro práci s databází a pro vytváření SQL dotazů.

Ve svém základu RoR poskytuje kompletní API pro vývoj, počínaje např. zmíněným ActiveRecordem, přes ActionMailer, který se stará o posílání emailů, až po ActionView, který pracuje s šablonami pro view. Tyto a další části tvoří Ruby on Rails jako celek.

RoR se rozšiřují pomocí tzv. gemů. Pod pojmem gem si můžeme představit balíček, jež obsahuje kód, který se integruje do RoR aplikace. Pokud chci, aby moje webová aplikace postavená na RoR uměla komunikovat s databází PostgreSQL, nainstaluji si patřičný gem (v mém případě to je gem s názvem pg). Tento gem pro mě bude klientem a ActiveRecord jej použije pro spojení s databází.

Na stejném nebo podobném principu fungují všechny gemy. Gemy se řídí strukturou složek, kterou RoR používá, takže při spouštění aplikace se soubory gemů začlení do RoR aplikace.

Gemy zmiňuji a zdůrazňuji hlavně z toho důvodu, že je budu používat pro komunikaci jak s databází, tak případně i s fulltextovými vyhledávacími nástroji. V každém případě to bude fungovat stejně. Zvolím si vhodný gem, který bude sloužit jako klient pro dané řešení a já nebudu muset používat nebo psát přímo API daného nástroje. Pokaždé použiji DSL daného gemu, které naimplementuji do modelů a kontrolerů.

Příklad č. 1 ukazuje základní strukturu Rails aplikace. V této práci budu pracovat převážně se soubory ve složce *app/models*, kde jsou uloženy modely, a v nich budu provádět konfigurace pro indexování a vyhledávání. Pokud budu v práci měnit nějaký soubor, uvedu jeho cestu ve struktuře v aplikaci vždy na prvním řádku.

Mimo modely budu také upravovat kontrolery, které budou přijímat vyhledávací parametry a dotazy, které pak použijí pro vyhledávání. Kontrolery se nachází ve složce *app/controllers.*

2.2. Fulltextové vyhledávací nástroje

Pro účely této práce jsem si vybral tři fulltextové vyhledávací nástroje. Prvním je Apache Solr, se kterým už mám zkušenosti a je aktivně používán v projektu Artvasion. Druhým je ElasticSearch, který je relativně novinkou na trhu. Jako poslední jsem vybral PostgreSQL. PostgreSQL mě zajímá mimojiné také z důvodů, protože jej používáme jako databázi pro ukládání dat a nabízí možnost využítí jako fulltextový vyhledávací nástroj.

2.2.1. Apache Solr

Apache Solr je open source fulltextový vyhledávácí nástroj postavený nad Apache Lucene. Apache Lucene je sám o sobě fulltextový vyhledávácí nástroj a určitě by šel použít bez Apache Solr. Nicméně, Apache Solr je nádstavbou nad Apache Lucene, který používán jako jádro, a rozšiřuje jej o funkce, jako jsou např. JSON nebo XML API, administrační rozhraní, facety a mnoho dalšího [2]. Díky svému API vznikla pro Ruby on Rails spousta klientů, kteří mohou s Apache Solr jednoduše komunikat a posílat mu data pro indexování.

Apache Solr patří do skupiny fulltextových vyhledávacích nástrojů, která potřebuje pro indexování dat a jejich následné vyhledávání schéma. Schéma definuje vnitřní pole, do kterých se indexují data. V případě Apache Solr mohu například jmenovat typy řetězec, text, celé číslo, datum a čas. Schéma přesně udává, jakým způsobem budou tato data indexována, ale také vyhledávána.

Schéma lze upravovat přesně podle potřeb aplikace a přizpůsobovat indexovaným datům. V následujícím příkladu ukážu, jak pomocí schématu Apache Solr indexuje data a jak je používá ke zpracování dotazu ze strany klienta.

```
<fieldType name="text" class="solr.TextField" omitNorms="false">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StandardFilterFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

Příklad č. 2 - Standardní nastavení schéma Apache Solr pro datový typ text

Ukázka kódu zobrazuje standardní nastavení pro datový typ text. V definici

field type je jasně patrné, že atribut name, určuje jméno pole v indexu a třída určuje chování tohoto pole zděděného ze třídy *solr.textField*.

Jelikož blok *analyzer* neobsahuje žádné atributy, bude nastavení uvnitř tohoto bloku platit pro indexovaná i vyhledávaná data. Celý proces bude do podrobna rozveden později v této kapitole.

Tokenizer se stará o zpracování textu na tzv. tokeny. Pokud dáme tokenizeru blok textu, rozdělí text na několik bloků (tokenů) a podle třídy, ze které dědí chování, provede nad tokeny určité operace, např. převede všechny tokeny na řetezce složené z malých písmen. Třída *solr.StandardTokenizerFactory* dělí text pouze na slova a odstraňuje formátovací znaky a neindexovatelné symboly. Tokenizer může být v analyzer bloku pouze jednou.

Opačným případem jsou filtry. Ty s textem rozlečleněným na tokeny dále pracují a může jich být mnoho, záleží na potřebách aplikace. Příkladem může být filter třídy *solr.StandardFilterFactory,* který zbaví text opakujících se slov, nebo filter třídy *solr.LowerCaseFilterFactory*, který text převede do malých písmen [3].

V praxi bude toto nastavení fungovat následovně:

1. Solr přijme požadavek na zaindexování textu typu text.
2. Dle zaslaných dat, Apache Solr najde ve svém schématu, jakým způsobem má text typu text zaindexovat.
3. Text tokenizer rozčlení text na slova.
4. První filtr odebere opakující se slova.
5. Druhý filter převede slova do malých písmen.
6. Výsledek je zaindexován.

Vyhledávání bude fungovat úplně stejně:

1. Uživatel zadá textový dotaz a ten aplikační klient pošle Apache Solr.
2. Podle dotazu Apache Solr začne vyhledávat nad indexovanými daty. Pro každé pole najde ve schématu způsob, jakým má dotaz zanalyzovat než začne porovnávat.
3. Text tokenizer rozčlení text na slova.
4. První filter odebere opakující se slova.
5. Druhý filter dotaz převede na malá písmena.

6. Zpracovaný dotaz je porovnán s indexem a pokud je nalezena shoda, výsledek je vrácen klientovi.

Jelikož budeme Apache Solr instalovat jako balíček, pracovat s ním budeme pouze přes aplikačního klienta a konfigurovat budeme pouze schéma, nebudu jej již dále představovat.

2.2.2. ElasticSearch

ElastciSearch jsem si všimnul teprve nedávno. V jako každé komunitě, všude existují osobnosti, jejichž názory jsou slyšet a jejichž práce je vidět více než ostatních. Příkladem může být Karel Minařík, který se jako jeden z Čechů aktivně podílí na rozvoji i18n backendu pro Ruby on Rails. Mě zaujal gem s názvem Tire, který je klientem pro ElasticSearch a pochází právě z dílny Karla Minaříka a dalších spolupracovníků.

ElasticSearch si klade několik cílů [14]:

- Rychlé nasazení
- Co nejmenší konfigurace
- Žádné schéma
- Vysoká dostupnost
- Vysoká škálovatelnost
- Rychlé vyhledávání
- RESTful přístup
- Komunikace pomocí JSON přes HTTP protokol

ElasticSearch, stejně jako Apache Solr, je napsaný v programovacím jazyce Java a stejně tak používá i Apache Lucene jako své jádro. Narozdíl od Apache Solr, kde nevíme, co si Solr ukládá do indexu, ElasticSearch to prezentuje úplně jasně. ElasticSearch vychází z modelu NoSQL Document Oriented databází a jejich rychlosti. Tyto NoSQL databáze používají JSON pro ukládání informací. Stejně tak to dělá i ElasticSearch. Dá se říct, že data ve formátu JSON, která na server pošleme, se tam také v této podobě uloží.

Jako vývojáři Ruby on Rails mi velice imponuje RESTful přístup.

ElasticSearch ve své dokumentaci nabízí široký přehled svého API [15]. Protože budu k ElasticSearch přistupovat především přes klienta, uvedu pouze základní příklady. V těchto příkladech předpokládejme, že mám ElasticSearch nainstalovaný na svém počítači (adresa je tudíž *localhost*) a server poslouchá na portu 9200.

```
# in console

$ curl -XPUT 'http://localhost:9200/artvasion/'
```

Příklad č. 3 - Vytvoření nového indexu s názvem Artvasion

```
# in console

$ curl -XPUT 'http://localhost:9200/artvasion/user/1' -d '{
  username: "DaveTsunami",
  name: "David",
  surname: "Horsák"
}'
```

Příklad č. 4 - Uložení uživatele do indexu

```
# in console

$ curl -XGET 'http://localhost:9200/artvasion/user/1'

# Response

{
  "_index": "artvasion",
  "_type": "user",
  "_id": "1",
  "_source": {
    "username": "DaveTsunami",
    "name": "David",
    "surname": "Horsák"
  }
}
```

Příklad č. 5 - Získání záznamu z indexu

```
# in console
```

```
$ curl -XGET 'http://localhost:9200/artvasion/user/_search?q=David

# Response

{
  "_shards": {
    "total": 5,
    "successfull": 5,
    "failed": 0
  },
  "hits": {
    "total": 1,
    "hits": [
      {
        "_index": "artvasion",
        "_type": "user",
        "_id": "1",
        "_source": {
          "username": "DaveTsunami",
          "name": "David",
          "surname": "Horsák"
        }
      }
    ]
  }
}
```

Příklad č. 6 - Vyhledávání nad uživateli

```
# in console

$ curl -XDELETE 'http://localhost:9200/artvasion/user/1
```

Příklad č. 7 - Smazání uživatele z indexu

Předchozí příklady jasně ukazují REST přístup v případě ElasticSearch. Za prvním lomítkem hned za adresou serveru je jméno indexu (v našem případě *artvasion*). Následuje jméno modelu. U ElasticSearch bych to spíše nazval jmenným prostorem. Na konci adresy se nachází buď ID nebo jméno metody, jako např: _search, která dále s požadavkem pracuje.

V představení Apache Solr jsem žádnou ukázku komunikace neukazoval. Narozdíl od ElasticSearch, Apache Solr způsob komunikace nijak nevyzdvihuje. Co má ovšem Apache Solr a ElasticSearch společného je analýza indexovaného obsahu a dotazů [16]. Vše je dáno společným jádrem, kterým je Apache Lucene.

Proto stejně jako v případě Apache Solr musíme nadefinovat analyzer. To, co bylo řešeno o tokenizerech a filtrech v případě Apache Solr, platí i pro ElasticSearch. Akorát konfigurace vypadá jinak.

```
index:
  analysis:
    analyzer:
      standard:
        tokenizer: standard
        filter: [standard, lowercase]
```

Příklad č. 8 - Nastavení indexu pro analýzu indexovaného obsahu

Podle příkladu je jasná podobnost s nastavením schématu v Apache Solr. Po lehkém přezkoumání se dá poznat, že bude použit standardní tokenizer a při filtrování standardní filtr odstraní anglické stop slova a změní všechny písmena na malé. Upravené nastavení indexu je potřeba nahrát do ElasticSearch pomocí požadavku na server ve formátu JSON [17].

ElasticSearch má velmi rozsáhlé RESTful API, které umožňuje všechny oprace potřebné pro komunikaci s fulltextovým vyhledávacím nástrojem. Tento nástroj je mi velice sympatický a těším se na práci s ním v praktické části.

2.2.3. PostgreSQL

O PostgreSQL jsem se začal zajímat až po absolvování předmětu Návrh relačních databází, kde právě vedoucí této bakalářské práce mluvil o fulltextových vyhledávacích nástrojích a zmínil se i o modulech, které se do relačních databázích dají přidat, aby mohly fungovat jako fulltextové vyhledávací nástroje. Od verze PostgreSQL 8.3. jsou fulltextové vyhledávací funkce implementovány přímo v jádře PostgreSQL [18].

Databáze PostgreSQL je používána jako hlavní databáze pro ukládání dat v Artvasion. Část třídního modelu v kapitole 3.1 je obrazem i datového modelu, který Artvasion používá.

Co chápu jako obrovskou výhodu využití PostgreSQL jako fulltextového vyhledávacího nástroje je zjednodušení. Pokud použiji svou databázi pro fulltextové vyhledávání, nemusím instalovat a konfigurovat další software a také se o tento software starat.

Práce s databází při vyhledávání probíhá pomocí jazyka SQL a to stejným

způsobem jako při klasickém SQL. Databázi je položen SQL dotaz a buď PostgreSQL prochází řádek po řádku a aplikuje na tyto řádky funkce, které pracují s textem, anebo prochází indexem a hledá [19].

Jak již bylo zmíněno, PostgreSQL obsahuje již ve svém jádru všechny funkce potřebné pro fulltextové výhledávání, resp. od verze 8.3. implementuje modul tsearch2, který se dříve používal jako modul pro fulltextové vyhledávání. tsearch2 značně rozšířil PostgreSQL, SQL příkazy pro PosgtreSQL a také vnitřní funkce, které pracují s textem. Také implenetuje tzv. parsery, které slouží jako tokenizery a filtery v Apache Lucene. PostgreSQL je s jejich pomocí schopný rozdělovat text na slova, zmenšovat na malá písmena, odstraňovat diakritiku atd.

Mimo tsearch2 obsahuje PostgreSQL i tzv. contrib mobuly, což jsou jednoduché funkce, které pracují s textem a mohou zastávat funkci jednoduchého fulltextového vyhledávače. Příkladem může být funkce *lower* nebo *unaccent*.

```
unaccent(Horsák) -> Horsak
lower(Horsák) -> horsák
unaccent(lower(Horsák)) -> horsak
```

Příklad č. 9 - Použití contrib modulů v PostgreSQL

Contrib moduly se přidávají do databáze pomocí příkazu *CREATE EXTENSION* [20]. Funkce, které přináší, mohou být použity třeba pro jednoduché vyhledávání jmen bez háčků a čárek, vhodné např. pro autocomplete jmen pro odesílatelé zpráv apod.

tsearch2 má obrovské možnosti použití a určitě se dá postavit na úroveň Apache Lucene. Základní použití může vypadat např. následovně:

```
# psql

artvasion=# select 'street art' @@ 'art'; ?column?
----------
 t
(1 row)

artvasion=# select 'street art' @@ 'painting';
 ?column?
----------
 f
(1 row)
```

```
artvasion=# select to_tsvector('street art') @@ to_tsquery('art'); ?
column?
----------
 t
(1 row)
```

Příklad č. 10 - Fulltextové vyhledávání v řetězci

V příkladu jsem použil statický řetězec jako vstup a nechal jsem pomocí symbolu @@ PostgreSQL vyhledat řetezec "art". Dle odezvy od databáze bylo vyhledávání úspěšné. Když jsem se snažil vyhledat řetězec "painting", mé vyhledávání bylo neúspěšné. Symbol @@ mimo to, že spouští fulltextové vyhledávání, tak „obaluje“ parametr nalevo a napravo do funkcí *to_tsvector* a *to_tsquery*, viz příklad č. 10.

To, co potřebuji, není vyhledávání nad řetezcem, ale nad tabulkou nebo celou databází. Proto je potřeba můj SQL dotaz rozšířit.

```
# psql

artvasion=# select surname from users where surname @@ 'horsák';
 surname
---------
 Horsák
(1 row)
```

Příklad č. 11 - Základní fulltextové vyhledávání podle uživatelského jména

Použil jsem fulltextového vyhledávání, abych našel své příjmení. Pokud chci vyhledávat bez diakritiky, musím modifikovat svůj dotaz, aby použil již zmíněných contrib modulů.

```
# psql

artvasion=# select surname from users where unaccent(surname) @@
unaccent('horsák');
 surname
---------
 Horsák
(1 row)
```

Příklad č. 12 - Vyhledávání bez diakritiky

tsearch2 nabízí jedinou možnost jak vyhledávat bez diakritiky, a to pomocí tzv. dictionary - česky slovníku. Slovníky mohou mít několik úkolů. Může jim být např. vyhledávání synonym, kde vložíme hledaný výraz a databáze vezme v potaz i synonyma. Pokud bych chtěl vyhledávat bez diakritiky, musel bych si sám připravit slovník, který by filtroval slova a odstraňoval by z nich diakritku, tzv. filtering dictionary [26]. Přijde mi jednodušší použít pro tyto účely contrib modul unaccent a proto se touto možností nebudu zabývat.

Vyhledávání uvnitř řezezců tsearch2 sám neumí, musí používat další contrub modul s názvem pg_trgm.

```
# psql

artvasion=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
```

Příklad č. 13 - Instalace pg_trgm do databáze

Jeho jméno je zkratkou pro trigram. Jeho úkol je „rozdrobit“ vstupní slovo na části 3 znaky dlouhé.

```
# psql

artvasion=# select show_trgm('davetsunami');
                        show_trgm
---------------------------------------------------------
 {"  d"," da",ami,ave,dav,ets,"mi ",nam,sun,tsu,una,vet}
(1 row)
```

Příklad č. 14 - Jak funguje pg_trgm

pg_trgm neumí slova vyhledávat, ale při fulltextovém vyhledávání je možné jej použít pro určení nejvíce relevantního výsledku. Následující příklad č. 15 demonstruje použítí pg_trgm pro určení relevatnosti výsledku. SQL dotaz vrátí nalezená uživatelská jména seřazená podle velikosti shody s nalezeným výsledkem, kde 0 je absolutní shoda a jedna označuje žádnou shodu.

```
# psql

artvasion=# select username, username <-> 'davetsu' AS dist FROM
users order by dist;
     username      |   dist
-------------------+----------
 DaveTsunami       | 0.461538
 darthdeus         |    0.875
 Savel             | 0.923077
 Denisa            | 0.928571
 pavelhouf         | 0.941176
 dominickous       | 0.947368
 Zavorka           |        1
 mafia666          |        1
 Esh               |        1
 cabbage           |        1

...
```

Příklad č. 15 - Vyhládávní pomocí pg_trgm seřazené podle shody

Aby fulltextové vyhledávání fungovalo efektivně a databáze nemusela procházet při každém vyhledávání tabulku řádek po řádku, je zapotřebí použít indexy. Apache Solr a i ElasticSearch se o tuto problematiku starají samy. U PostgreSQL je tomu přesně naopak.

Fulltextové vyhledávání nad PostgreSQL potřebuje pro své efektivní fungování GiST nebo GIN indexy [27] vytvořené nad sloupci typu ts_vector. Tento druh sloupce může obsahovat prakticky jakékoliv hodnoty a tsearch2 používá parsery pro určení typu obsahu. Pokud bych měl dva sloupce, kde jeden by byl typu string a druhý právě ts_vector, vyhledávání nad sloupce typu ts_vector by bylo rychlejší.

Do sloupců ts_vector většinou nezapisuje přímo aplikace. Dělá se to tak, že nad sloupci typu string nebo text se vytvoří funkce (TRIGGER), která aktualizuje data ve sloupci ts_vector podle hodnot v normálním sloupci.

Pro kompeletní výčet možností indexů a rozdílů GiST a GIN odkážu čtenáře raději na referenci. Jediné, co bych rád zmínil, jsou jejich základní rozdíly.

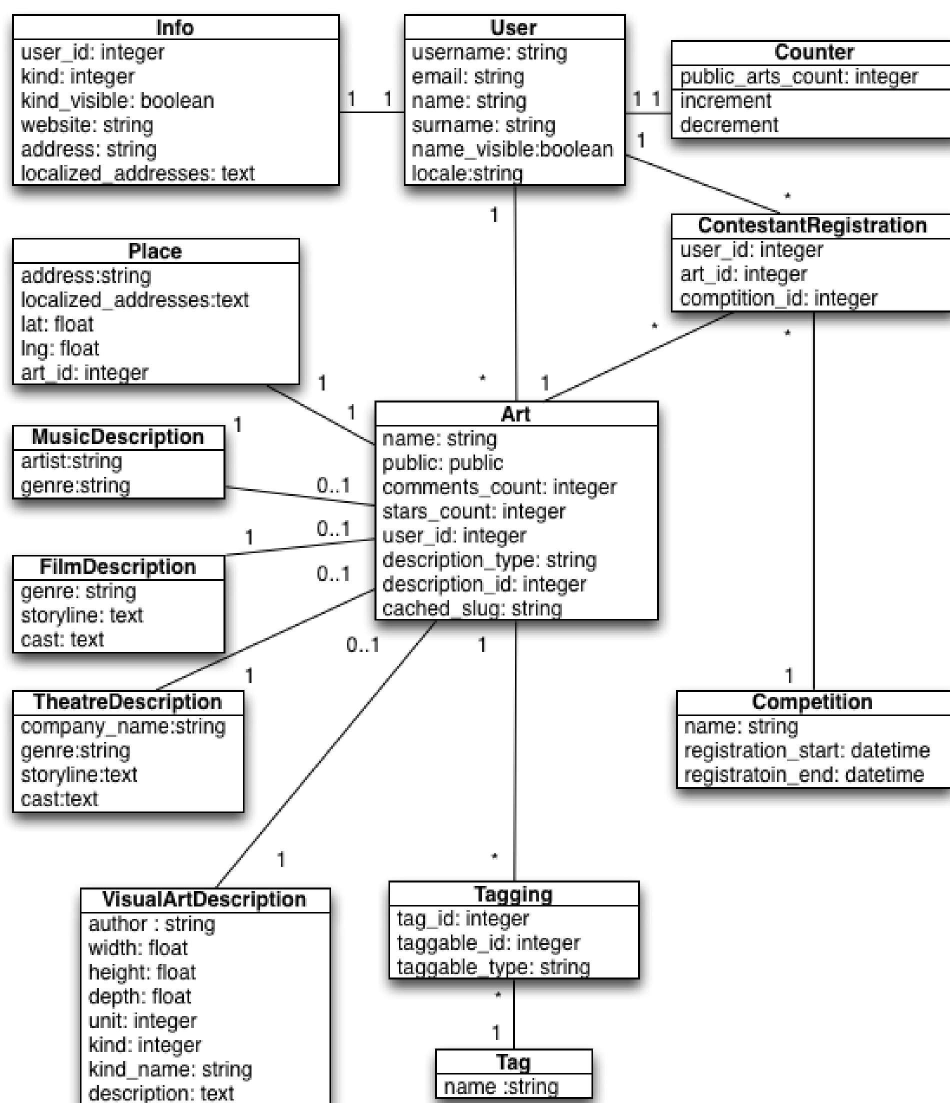
Podle reference je GIN index lepší pro statická data, protože se v něm rychleji vyhledává. Pro dynamická data je lepší GiST, protože umí rychleji aktualizovat data, ale pouze má-li v indexu méně než 100 000 unikátních slov. Pokud počet unikátních slov přesahuje zmíněnou hranici, je lepší použít GIN.

První co mě napadne je, že až bude mít Artvasion 100 000 uživatelů a v databázi 100 000 unikátních uživatelských jmen, budu mít dost peněz na to, abych investoval do výzkumu a technologie, která se o fulltextové vyhledávání postará. Proto bych zatím volil GiST, protože uživatelská data jsou spíše dynamická než statická.

3. Konfigurace nástrojů a infrastruktura

3.1. Datový model aplikace

Jak již bylo zmíněno, Artvasion je sociální síť pro umělce, galerie a lidi se zájmem o umění. Centrem datového modelu je tudíž uživatel (*User*). Na něj je navázáno umění (A*rt*). Na umění a uživatele je dále navázáno ještě několik tříd, které jsou hlavně nositeli informací.



Obrázek č. 1 - Část datového modelu aplikace Artvasion, nad kterou bude probíhat fulltextové vyhledávání.

Jméno třídy	**Popis**
User	Třída *User* je stavebním kamenem datového modelu, od které se všechno odvíjí. Její atributy obsahují informace potřebné pro uživatelovo přihlašování a jeho vystupování v systému - email, uživatelské jméno, jméno a příjmení.
Info	Třída *Info* je doplňkovou třídu k uživateli. Obsahuje informace, které se používájí k vyhledávání nebo k zobrazování na profilu uživatele. Informace v této třídě nejsou součástí třídy *User*, aby nedocházelo k neustálemu načítání těchto informací. Hlavním adaptérem, který Ruby on Rails používá pro komunikaci s databází, je ActiveRecord.
Counter	Třída *Counter* sdružuje informace počtech asociací třídy User, aby se nemusel volat SQL dotaz pokaždé, když je potřeba tyto údaje zobrazovat.
Art	Třídu *Art* bych spíše než za samostatnou třídu, označil jako za vezební třídu. Sama obsahuje pouze jméno a stav, jestli je umění veřejné nebo ne. Ostatní jsou už jenom vazby na popisy, tagy a místo. Popisy jsou vázány polymorfní vazbou na třídu, která obsahuje atributy potřebné pouze pro daný druh umění.
Place	Třída *Place* je nositelem informací o fyzické poloze umění. Zajímavým atributem je *localized_addresses*, kde jsou uloženy adresy lokalizované pro dané lokále. Tyto adresy jsou uloženy v podobě serializovaného hashe ve formátu YAML.
MusicDescription	Třída *MusicDescription* obsahuje atributy, do kterých se ukládají informace o hudbě.
TheatreDescription	Třída *TheatreDescription* obsahuje atributy, do kterých se ukládají informace o divadelních představeních.
FilmDescription	Třída *FilmDescription* obsahuje atributy, do kterých se ukládají informace o filmech.
Tagging	*Tagging* je vazební třída s polymorfní vazbou na cokoliv (prozatím používané pouze pro umění). Na její "druhý konec" se připojuje třída *Tag*.
Tag	Třída *Tag* je nositelkou pouze jména štítku.

Competition	Třída *Competition* zastupuje soutěž. Ta má jméno a také odkdy do kdy je možné se do ní registrovat.
ContestantRegistration	*ContestantRegistration* slouží pouze jako vazební tabulka mezi uměním, registrovaným uživatelem a soutěží.

Tabulka č. 1 - Popis tříd v datovém modelu

3.2 Infrastruktura

Při prvotním vývoji Artvasion byla jako databáze zvolena PostgreSQL, která při posledním nasazováním Artvasion byla upgradována na verzi 9.1. Jako hlavní operační systém produkčního prostředí bylo zvoleno Ubuntu. Sám mám výborné zkušenosti s linuxí distribucí Debian a Ubuntu je jeho odštěpením z vývojové větve. Oba operační systémy používají balíčkovací nástroj APT.

Pro účely této práce budu tudíž vždy v základu používat virtuální stroj běžící v programu VirtualBox s nainstalovaným operačním systémem Ubuntu 10.04 LTS Lucid Lynx. Podle potřeby budu do tohoto operačního systému instalovat potřebné komponenty vždy jeden fulltextový vyhledávací nástroj. Jako databázi budu mít nainstalovanou PostgreSQL 9.1.4.

Pro běh Ruby on Rails aplikací budu používat programovací jazyk Ruby ve verzi 1.9.3 a Ruby on Rails 3.2.3. Ruby on Rails aplikaci budu vždy pouštět na svém MacBooku Pro.

Ubuntu má všechny potřebné softwarové knihovny pro svůj běh v sobě. V případě PostgreSQL se o knihovny postará APT.

Pro instalaci Apache Solr použiju APT balíček se jménem *solr-jetty*. Tento balíček webový server Jetty a Apache Solr. Je to doslova "One-Click Installer", protože po nainstalování stačí akorát spustit a Apache Solr je připraven. Je potřeba dodat, že zmíněný balíček obsahuje Apache Solr ve verzi 1.4. Tato verze není aktuální. Nejnovější verze je 3.5. I přesto budu pracovat s verzí 1.4, protože je nasazená v produkčním prostředí na Internetu.

Pro instalaci ElasticSearch použiji manuál na jejich oficiálních stránkách [5]. Jelikož ElasticSearch neexistuje v podobě žádného APT balíčku, je toto asi ta jediná správná cesta.

Třetím nástrojem pro fulltextové vyhledávání je samotná databáze PostgreSQL. Tu nebudu muset instalovat, protože ji mám v základní sestavě.

3.3. Zadání pro fulltextové vyhledávání

Následující požadavky na vyhledávání vycházejí z datového modelu a specifikace:

- Vyhledávání uživatelů:
  - Hledání bez dotazu vrátí všechny uživatele.
  - Řadit výsledky musí být možné podle relevatnosti dotazu, podle uživatelského jména sestupně, podle počtu umění sestupně, podle data registrace sestupně.
  - Pokud zadá uživatel do dotazu „umělec“, „umelec“, „artist“, „kunstler“, musí mu vyhledávání vrátit pouze umělce.
  - Pokud uživatel zadá do dotazu „galerie“, „gallery“, musí mu vyhledávání vrátit pouze galerie.
  - Při vyhledávání se bude hledat v uživatelských jménech, jmenéch a příjmeních pokud jsou veřejná.
  - Pokud uživatel chce hledat umělce pocházející z Prahy, dotaz "praha" nebo "prague" nebo jakákoli jiná lokalizovaná varianta by měla být nalazena. To samé platí pro jiná města i země.
  - Hledané dotazy by neměly brát v potaz malá a velká písmena ani diakritiku.
  - Pokud je uživatel přihlášen do soutěže, dotaz se jménem soutěže by měl vrátit všechny uživatele do této soutěže přihlášené.
  - Vyhledávání musí být možné také pouze zadáním částečného jména, např. pokud se uživatel jmenuje "CeskyTucnak", po zadaní výrazu "tucnak", by tohoto uživatele mělo vyhledávání najít.
- Vyhledávání umění:
  - Hledání bez dotazu vrátí všechno umění.
  - Řadit výsledky musí být možné podle relevance, jména, datum vytvoření, počtu hvězdiček (takto hodnotí uživatelé umění) a podle počtu komentářů.
  - Uživatelé musí mít možnost vybrat, jaký druh umění hledají. Zde mohou vybírat mezi možnostmi všechno, výtvárné umění, film, divadlo a hudba.

- ◦ Pokud uživatel zadá jméno umění, vrátí mu vyhledávání všechna umění nesoucí toto jméno.
- ◦ Vyhledávání musí být možné také pouze zadáním částečného jména, např. pokud se umění jmenuje "Illustrations", po zadaní výrazu "lustrations", by toto umění mělo vyhledávání najít.

3.4 Apache Solr

Apache Solr je na Artvasion již aktuálně používaný. V době prvního spuštění existovaly pouze dva gemy, které mohly sloužit jako klient pro propojení s Ruby on Rails aplikací. První je Acts_as_solr [6] a druhý Sunspot [7].

Vývoj u Acts_as_solr nebyl příliš aktivní a ani komunita, která se kolem něj vytvořila, nebyla příliš rozsáhlá. Aktuálně už vývoj tohoto gemu napokračuje [8]. Jelikož byl gem publikován pod MIT licencí, někteří uživatelé převzali vývoj. Jde to moc pěkné vidět v git webovém repozitáři Github, kde je možnost se podívat na graf vytvořených větví. Nicméně v žádné větvi vývoj dostatečně nepokročil a převážně jde o práci jednotlivců. Proto nemá smysl se tímto gemem dále zabývat.

Sunspot je stálicí v Ruby on Rails světe. Používá jej převážná většina aplikací pracující s Apache Solr. Gem je dobře zdokumentovaný a při každém posunu ve vývoji Ruby on Rails je gem aktualizován, aby nezůstával pozadu za vývojem. S gemem se dobře pracuje a mám s ním výborné zkušenosti.

Novým gemem je Solrsan. Gem vypadá vcelku dobře. I to jak se integruje do Ruby on Rails a jeho modelů [10]. Nicméně opět jde o práci jednotlivce, takže vývoj není dostatečně stabilní pro použití v našem případě.

Po opětovném úvážení všech možností pro integraci Artvasion s Apache Solr opět vítězí Sunpost. Infrastruktura je už je jasná z předchozí části prácie Proto se mohu plně vrhnout na konfiguraci aplikace.

3.4.1. Příprava Apache Solr a klienta

Do svého Gemfile přidám následující řádek, kde říkam Bundleru [30], aby tento gem nainstaloval. Poté Bundler spustím.

```
# Gemfile
```

```
gem 'sunspot_rails', '1.3.1'

# in console
$ bundle
```

Příklad č. 16 - Instalace gemu sunspot do Ruby on Rails aplikace

Po instalaci gemu je potřeba ještě spustit rails generátor, který vytvoří ve soubor *config/sunspot.yml* s konfiguračními údaji pro připojení k Apache Solr.

```
# in console
$ rails generate sunspot_rails:install
```

Příklad č. 17 - Spuštění generátoru pro vytvoří souboru sunspot.yml

Takto vypadá nastavení mého sunspotu:

```
# config/sunspot.yml

development:
  solr:
    hostname: 192.168.1.138
    port: 8080
    path: '/solr
    log_level: INFO

test:
  solr:
    hostname: 192.168.1.138
    port: 8080
    path: '/solr'
    log_level: INFO
```

Příklad č. 18 - Vzorové nastavení gemu sunspot

Na nejnižší úrovni ve stromovém zápisu YAML souboru určuji prostředí, pro které nastavení platí. Rails v základu rozlišuje prostředí tři - development (vývojové), test (testovací) a production (produkční). Hostname uvozuje adresu Apache Solr, port na kterém běží a důležitá je path (cesta). Solr může mít tzv. více jader - více spuštěných subinstancí sama sebe s vlastními indexy. U těch jader se

pak cesta liší. Já více jader nekonfiguruji, proto si vystačím s tím jedním hlavním, a to je dostupné na cestě specifikovaném v ukázce konfigurace.

Z pohledu třídního modelu budeme indexovat atributy typu string, text, datetime a boolean. Solr používá stejné datové typy až na datový typ datetime - Solr má ekvivalent se jménem time.

Apache Solr má kompletní schéma pro všechny datové typy. Pokud bych schéma nijak neměnil, indexování a vyhledávání by probíhalo podle již definovaných tokenizerů a filtrů, které vychází z dědičnosti třídy, jež pole definuje.

U datových typů jako např. time, boolean a string si vystačím se standardním nastavením (vycházím ze zadání). Pole text si budu muset přizpůsobit tak, aby indexovalo text převedený do malých písmen, bez diakritiky a rozdělené na malé tokeny, abych mohl vyhledávat i řetězce uvnitř dlouhých slov.

Pro již zmíněné účely jsem si ve schématu nadefinoval nový typ pole, který nazývám *text_pre* a vypadá následovně:

```
<--! /etc/solr/conf/schema.xml -->

<schema name="sunspot" version="1.0">
  <types>
    <fieldtype class="solr.TextField" name="text_pre">
      <analyzer type="index">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.NGramFilterFactory" minGramSize="2"
maxGramSize="15"/>
      </analyzer>
      <analyzer type="query">
        <tokenizer class="solr.WhitespaceTokenizerFactory"/>
        <filter class="solr.ASCIIFoldingFilterFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
      </analyzer>
    </fieldtype>
  </types

  <--! ... -->
</schema>
```

Příklad č. 19 - Nastavení pole se jménem text_pre

Při představování Apache Solr jsem popsal ukázku jednoduchého

textového pole. Tento příklad je složitější v tom, že indexace textu probíhá jiným způsobem, než zpracování.

Analyzer s typem *index* zpracovává text pro indexování. Pro rozdělení slova na tokeny jsem použil tokenizer třídy *solr.WhitespaceTokeanizerFactory*. Pokud ten dostane na vstup text, např. „Český tučňák“, rozdělí jej na dvě slova. Jeho název napovídá, co tento tokenizer dělá - rozděluje na řetezce podle mezer.

```
"Český tučňák" -> "Český", "tučňák"
```

Příklad č. 20 - Použítí tokenizeru třídy solr.WhitespaceTokeanizerFactory

Filtr třídy *solr.ASCIIFoldingFilterFactory* vymění všechny Unicode symboly, jako např. diakritiku, za jejich ASCII ekvivalenty.

```
"Český", "tučňák" -> "Cesky", "tucnak"
```

Příklad č. 21 - Použítí filterusolr.ASCIIFoldingFilterFactory

Filtr třídy *solr.LowerCaseFilterFactory* se naopak stará o to, aby se indexoval text, který je složen pouze z malých písmen.

```
"Cesky", "tucnak" -> "cesky", "tucnak"
```

Příklad č. 22 - Použítí filtru třídy solr.LowerCaseFilterFactory

Celý proces zpracování textu před uložením do indexu zakončuje filtr třídy *solr.NGramFilterFactory*, který slova rozdělí na několik řetezců tak, aby se zaindexovaly i vnořené řetezce (substrings). Příklad předvedu pro zjednodušení pouze na slově „cesky“.

```
"cesky" -> "ce", "es", "sk", "ky", "ces", "esk", "sky", "cesk",
"esky", "cesky"
```

Příklad č. 23 - Použítí filtru třídy solr.NGramFilterFactory

Při dotazování proces probíhá úplně stejně, akorát chybí poslední NGram filtr. Kdyby se tento filtr použil i pro analýzu dotazu, mohlo by to mít za následek nejen zbytečnou zátěž pro Apache Solr, ale také spoustu nerelevatních výsledků.

Pochybuji, že je pro uživatele relevantní, když hleda pomocí dotazu „Cesky“, ale vyhledávání mu vrátí také všechno, co v sobě obsahuje řetezec „ky“.

Tímto nastavením jsem dosáhl toho, že uživatel „CeskyTucnak“ bude nalazen, i když uživatel zadá pouze dotaz „Tučňák“.

Aby bylo nastavení schématu kompletní, je potřeba do schématu přidat ještě jedno dynamické pole. Toto dynamické pole definuje ve svém atributu *name* postfix, kterým se budu v kódu odkazovat na toto dynamické pole a také definuje typ *type*, který určuje, jaký druh pole bude použit pro indexování a vyhledávání. Moje dynamické pole vypadá následovně:

```
<--! /etc/solr/conf/schema.xml -->

<schema name="sunspot" version="1.0">

  <fields>
    <dynamicField name="*_textp" stored="false" type="text_pre"
multiValued="true" indexed="true"/>
  </fields>

  <--! ... -->

</schema>
```

Příklad č. 24 - Dynamické pole pro text_pre

3.4.2. Nastavení modelů

Nastavení modelů se děje přímo v jejich souborech pomocí jazyka Ruby a DSL, které gem sunspot nabízí. Model User jsem nastavil následujícím způsobem:

```
# app/models/user.rb

class User < ActiveRecord::Base
  searchable do
    text :username, :as => :username_textp, :boost => 2.0
    string :username
    text :name, :as => :name_textp
    text :surname, :as => :surname_textp
    text :addresses, :as => :address_textp do
      info.localized_addresses.keys.collect { |key|
user.info.localized_addresses[key] }
    end
```

```
    integer :competition_ids, :references => Competition, :multiple
=> true do
      contests.pluck("competitions.id")
    end
    text :kind, :as => :kind_textp, :boost => 2.0 do
      case info.kind
      when Info::ARTIST
       "artist umelec kunstler"
      when Info::GALLERY
        "gallery galerie"
      when Info::OTHER
        info.kind_name
      end
    end
    integer :no_of_art do
      counter.public_arts_count
    end
    time :created_at

  end
  # ...
end
```

Příklad č. 25 - Nastavení modelu User pro vyhledávání

DSL Sunspotu je velice jednoduché a používá konvencí pro snížení jinak potřebné konfigurace. Sunspot se integruje do modelu pomocí volání metody *searchable*. Tato metoda pak přebírá blok, který se v případě indexace volá na každém objektu. Jak to vypadá v bloku metody *searchable* je jasně zřejmé. Vždy se volá metoda, která má stejné jméno jako datový typ, do které se hodnota bude ukládat. Jako parametr se jí posílá název atributu a hash s dalšími možnostmi pro konfiguraci.

U každého metody *text* předávám také hash s *:as => :jmeno_attributu_textp*, kde „_textp" je zmíněný postfix. Apache Solr postfix rozpoznává pomocí dynamického pole a zaindexuje do pole *text_pre*. U všech ostatních datových typů se použije konvence, tzv. že volání metody *string* bude ukládat do pole pro řetězec. Stejně tak to udělá i metoda *time*.

Mezi další konfigurační možnosti patří i klíč *:references*, kde Sunspotu říkám, na jaký model budou odkazovat indexované hodnoty. Klíč *:multiple* definuje, že se do jednoho pole se bude ukládat více hodnot. *:boost* zvyšuje váhu výsledku v případě, že se dotaz shoduje s nalezeným modelem pomocí tohoto

atributu.

Poslední věcí, která se mi na Sunspotu velice líbí, je možnost předat blok místo jména atributu. V případě, že volám metodu *text* a jako první atribut předám symbol, Sunspot tento symbol vezme a pomocí reflexe zavolá metodu se jménem symbolu. Na druhou stranu v bloku si můžu napsat potřebnou logiku a vrátit mnou připravené hodnoty. Dobrým příkadem je blok, kde se odkazuji na model *Competition*, Sunspotu říkám, že budu indexovat více položek a z bloku vracím pole se identifikačními ID soutěží [11].

Od teď při každém vytvoření nebo uložení objektu třídy *User* se bude Sunspot starat o indexaci.

3.4.3. Vyhledávání

Vyhledávání se Sunspotem mi přijde velice jednoduché. Jde o pouhé volání třídy *Sunspot* a její statické metody *search*, která si bere jako parametr třídy, nad kterými se bude vyhledávat, a blok s metodami, které vyhledávání konfigurují. Metoda vrací objekt s parametry vyhledávání, na které když se zavolá metoda *results*, proběhne vyhledávání, které vrací pole s výsledky.

```
# app/controllers/browsers_controller.rb

class BrowsersController < ApplicationController

  def search_people
    ordering = people_order(params[:order])
    search = Sunspot.search(User) do
      keywords params[:query]
      order_by(ordering[0], ordering[1]) if ordering
      with(:competition_ids, params[:competition_id].to_i) if
params[:competition_id]
      paginate(:page => params[:page], :per_page => PAGINATE_PER)
    end
    @users = search.results
  end

  private

  def people_order(attribute)
    case attribute.to_s.to_sym
    when :relevance
      nil
    when :username
```

```
      [:username, :asc]
    when :no_of_art
      [:no_of_art, :desc]
    when :member_since
      [:created_at, :desc]
    else
      nil
    end
  end
end
```

Příklad č. 26 - Implementace kontroleru pro vyhledávání nad modelem User

Před začátkem vyhledávání volám metodu people_order, která nastavuje řazení podle poslaného parametru (relevance, username, no_of_art, member_since).

Nejzákladnější metodou v bloku pro vyhledávání je metoda *keywords*, která bere jako parametr řetězec s dotazem. Další metoda je např. order, která si bere jako první atribut symbol se jménem atributu, podle kterého se bude řadit, a jako druhý atribut, jestli to bude vzestupně nebo sestupně.

S metodou with mohu vybírat, co musí nalezené objekty obsahovat. V mém případě, pokud požadavek na server obsahuje identifikační číslo soutěže, chci vybrat pouze takové výsledky, které obsahují dané číslo soutěže. Metoda with se dá různými zbůsoby dále ohýbat a dají se na vráceném objektu volat daślí metody, jako např. any_of, all_of, equal_to, které ještě více specifikují hledaní [12].

Blok ve vyhledávání ukončuje metoda paginate, která Sunspotu říká, že má výsledky stránkovat. Podle nainstalovaného gemu Sunspot vrací buď objekt typu Kaminari nebo WillPaginate, což jsou gemy, které se o stránkování v Rails aplikacích mohou starat [13].

Pro kompletnost a možnost porovnání s ostatními technologiemi přidávám nastavení umění a také implementaci kontroleru, který nad uměním vyhledává.

```
# app/models/Art

class Art
  searchable do
    integer :user_id, :references => User
    string :name
    text :name, :as => :name_textp
```

```
    integer :competition_ids, :references => Competition, :multiple
=> true do
      contests.pluck("competitions.id")
    end
    text :description_names, :as => :description_names_textp do
      case description_type
      when "VisualArtDescription" then "vytvarno vytvarne umeni"
      when "FilmDescription" then "film movie motion"
      when "TheatreDescription" then "theatre divadlo"
      when "MusicDescription" then "music hudba muzika"
      end
    end
    text :tags, :as => :tags_textp do
      tags.map { |tag| tag.name }
    end
    string :description do
      case description_type
      when "VisualArtDescription" then "visual_art"
      when "FilmDescription" then "film"
      when "TheatreDescription" then "theatre"
      when "MusicDescription" then "music"
      end
    end
    boolean :public
    integer :stars_count
    integer :comments_count
    date :created_at
  end

  # ...
end
```

Příklad č. 27 - Nastavení modelu Art pro vyhledávání

Třída Art si pro vyhledávání drží referenci na uživatele a soutěže. Jako textové hodnoty indexuje své jméno a štítky. Aby bylo možné vyhledávat jednoduše divadlo, indexují se u všech umění s typem divadlo i slova „theatre“ a „divadlo”. Zadání dotazu „divadlo“ a pak vrací všechna divadla.

Atributy, které se indexuji jako typ celé číslo, řetězec a Boolean, indexuji jenom proto, abych podle nich mohl později řadit. Atribut boolean indexuji, abych mohl vyhledávat pouze nad uměním, jehož atribut public ma hodnotu true.

```
# app/controllers/browsers_controller.rb

class BrowsersController < ApplicationController
```

```
  def search_art
    ordering = art_order(params[:order])
    search = Sunspot.search(Art) do
      keywords params[:query]
      with(:description).equal_to(params[:type]) if
params[:type].present? && params[:type] != "all"
      with(:public).equal_to(true)
      with(:competition_ids, params[:competition_id].to_i) if
params[:competition_id]
      order_by(ordering[0], ordering[1]) if ordering.present?
      paginate(:page => params[:page], :per_page => 9)
    end
    @art = search.results
    respond_to do |format|
      format.html
      format.js
    end
  end

  private

  def art_order(attribute)
    case attribute.to_s.to_sym
    when :relevance
      nil
    when :name
      [:name, :asc]
    when :date_of_creation
      [:created_at, :desc]
    when :number_of_stars
      [:stars_count, :desc]
    when :number_of_comments
      [:comments_count , :desc]
    else
      nil
    end
  end
end
```

Příklad č. 28 - Implementace kontroleru pro vyhledávání nad modelem Art

Princip fungování této části implementace je stejná jako pro uživatele. Tímto nastavením a implementací jsem splnil zadání pro vyhledávání.

3.5. ElasticSearch

ElasticSearch jsem nainstaloval podle jejich návodu [5]. Jediným rozdílem byla verze, kterou jsem si stáhl. V návodu instalují verzi 0.17.2, já použil verzi 0.19.2. Pro instalaci jsem tudíž použil následující skript a nevytvářel jsem žádné složky ani soubory ve složce */dev*, protože konfigurační soubory má ElasticSearch ve své instalační složce. Zbytek konfigurace jsem provedl podle ElasticSearch reference [22].

```
# in console

$ curl -OL
http://github.com/downloads/elasticsearch/elasticsearch/elasticsearch
-0.19.2.zip
$ unzip elasticsearch-* && rm elasticsearch-*.zip
$ mv elasticsearch-* /usr/local/elasticsearch
```

Příklad č. 29 - Stažení aktuální verze ElasticSearch

ElasticSearch v základu běží na portu 9200, proto jsem hned po instalaci chtěl otestovat jestli funguje a proto jsem použil jeho API pro vytvoření nového indexu s názvem "artvasion".

```
# in console

$ curl -XPUT 'http://192.168.1.138:9200/artvasion'

# Reponse
{"ok":true,"acknowledged":true}
```

Příklad č. 30 - Test funkčnosti ElasticSearch vytvořením nového indexu

Výběr gemu byl velice jednoduchý. Jasnou velbou byl gem Tire od Karla Minaříka [22]. ElasticSearch na svých webových stránkách zvěřejněné ještě dva gemy - klienty, které se hodí pro integraci s Ruby nebo Ruby on Rails, ale buď postrádají aktivní komunitu anebo jsou špatně zdokumetované. V tomto má Tire naprosto navrch a proto jsem si jej nainstaloval.

```
# Gemfile
gem 'tire'

# in console
$ bundle
```

Příklad č. 31 - Test funkčnosti ElasticSearch vytvořením nového indexu

Po instalaci je potřeba integrovat ElasticSearch s modely. Prvním krokem je zahrnutí (include) modulů, se kterými ElasticSearch přichází. Podle tvůrců Tire jsem nyní připraven a mohu indexovat a vyhledávat. Tire automaticky posílá všechny atributy modelu k indexování, proto v menších aplikacích a ve vývojovém prostředí už není potřeba dál nic řešit. Modely Artvasion mají atributů hodně a ne všechny je potřeba indexovat. Proto můj model v základu vypadá takto:

```
# app/models/user.rb

class User
  # ...

  # include ElasticSearch modules
  include Tire::Model::Search
  include Tire::Model::Callbacks
  # setup attributes for indexing
  mapping do
    indexes :username
    indexes :name
    indexes :surname
    indexes :address
    indexes :kind
    indexes :created_at
  end
end
```

Příklad č. 32 - Nastavení modelu User pro vyhledávání

Když se podíváte na moje nastavení, spousta atributů, které potřebuji indexovat, chybí. Pokud bych to ponechal tak, Tire by při indexování serializoval objekt User a zaindexoval by pouze atributy v bloku mapping. Tire nemumožňuje

do bloku mapping vkládat další bloky k atributům, které by umožňovaly jejich další modifikaci, jako třeba u Apache Solr. Na druhou stranu dovoluje vytvořit si vlastní metodu pro serializaci, kde již atributy mohu přizpůsobit. Výsledný JSON hash je pak poslán k indexování.

```
# app/models/user.rb

class User
  # ...

  mapping do
    indexes :username
    indexes :name
    indexes :surname
    indexes :addresses
    indexes :kind
    indexes :competitions_ids
    indexes :no_of_art
    indexes :created_at
  end

  def to_indexed_json
    user_kind = case info.kind
    when Info::ARTIST then "artist umelec kunstler"
    when Info::GALLERY then "gallery galerie"
    when Info::OTHER then info.kind_name
    end
    user_competitions_ids = contests.pluck("competitions.id")

    {
      :username => username,
      :name => name,
      :surname => surname,
      :addresses => info.localized_addresses,
      :kind => user_kind,
      :competitions_ids => user_competitions_ids,
      :no_of_art => counter.pubic_arts_count,
      :created_at => created_at
    }.to_json
  end
end
```

Příklad č. 33 - Rozšířené nastavení modelu User pro vyhledávání

Atributy *addresses* a *competitions_ids* indexuji v serializované formě. Nikde jsem nenašel žádnou referenci na to, jak pracovat s tímto druhem atributů. Uvidíme, jak se ElasticSearch zachová, a jestli bude možné se dotazovat na takového atributy.

Když jsem chtěl začít zkoušet aplikaci, narazil jsem na problém s propojením Rails aplikace. Došlo mi, že nikde nenastavuji adresu serveru, na kterém beží ElasticSearch, proto jsem podle dokumentace vytvořil soubor ve složce *config/initializers*. Do této složky se ukládají soubory, které konfigurují Rails aplikaci nebo gemy.

```
# config/initializers/elasticsearch.rb

Tire.configure do
  url "http://192.168.1.138:9200"
end
```

Příklad č. 34 - Konfigurace IP adresy ElasticSearch pro gem Tire

Nicméně po dalším spuštění Rails konzole jsem dostal opět stejnou hlášku, že se Tire nezdařilo připojení k ElasticSearch. Po chvíli procházení jsem si našel metodu, kterou jsem zjistil nastavenou URL k ElasticSearch serveru.

```
#in console

$ bundle exec rails c
Skipping index creation, cannot connect to ElasticSearch
(The original exception was: #<Errno::ECONNREFUSED: Connection
refused - connect(2)>)
Loading development environment (Rails 3.2.3)
1.9.3p0 :001 > Tire::Configuration.url
 => "http://192.168.1.138:9200"
1.9.3p0 :002 >HTTParty.get("http://192.168.1.138:9200/_stats")
 => #<HTTParty::Response:0x7fde3459bec8 @parsed_response=....
```

Příklad č. 35 - Spuštění Rails konzole a test konektivity s ElasticSearch pomocí gemu HTTParty

Vše je nastavené správně a přes Curl jsem schopný se serverem komunikovat. Když jsem použil gem HTTParty, také jsem získal odezvu od serveru v pořádku. Když jsem zkoušel importovat User model do indexu

ElasticSearch pomocí Rake úkolu, opět jsem dostal zprávu, že se spojení nezdařilo, ale celá procedura doběhla do konce v pořádku a vypadá to, že data se zaindexovaly tak, jak měly. Základní vyhledávání funguje v pořádku.

```
# in console

$ bundle exec rake environment tire:import CLASS='User'
Skipping index creation, cannot connect to ElasticSearch
(The original exception was: #<Errno::ECONNREFUSED: Connection
refused - connect(2)>)
[IMPORT] Creating index 'users' with mapping:

#  ...

[IMPORT] Starting import for the 'User' class
----------------------------------------------------------------------
80/80 | 100% #######################################################
======================================================================
Import finished in 1.14302 seconds

$ bundle exec rails c
Skipping index creation, cannot connect to ElasticSearch
(The original exception was: #<Errno::ECONNREFUSED: Connection
refused - connect(2)>)
Loading development environment (Rails 3.2.3)
 > User.tire.search do
 >     query { string "davetsunami" }
 >   end
 => #<Tire::Results::Collection:0x007f9455cc9ea8
@response={"took"=>81, "timed_out"=>false, "_shards"=>{"total"=>5,
"successful"=>5, "failed"=>0}, "hits"=>{"total"=>1,
"max_score"=>0.5550905, "hits"=>[{"_index"=>"users", "_type"=>"user",
"_id"=>"4", "_score"=>0.5550905,
"_source"=>{"username"=>"DaveTsunami", "name"=>"David",
"surname"=>"Horsák", "addresses"=>{"original"=>""}, "kind"=>"artist
umelec kunstler", "competitions_ids"=>[], "no_of_art"=>2,
"created_at"=>"2011-08-30T21:20:21Z"}}]}}, @options={}, @time=81,
@total=1, @facets=nil, @wrapper=Tire::Results::Item>
```

Příklad č. 36 - Import dat do databáze a test vyhledávání

Chování ElasticSearch mi přijde opravdu zvláštní, ale pokračuji dál. Chybu jsem reportoval a uvidíme, jak se k tomu tvůrci Tire postaví [24].

V tuto chvíli se mi indexují uživatelé. Nicméně nejsou nastavené tokenizery

a filtry pro vychledávání bez háčků a čárek. Proto jsem nastavení modelu uživatel změnil následujícím způsobem:

```
# app/models/user.rb

class User

  # ...

  settings :analysis => {
    :filter => {
      :text_ngram => {
        "type" => "nGram",
        "max_gram" => 15,
        "min_gram" => 2
      }
    },
    :analyzer => {
      :text_analyzer => {
        :tokenizer => "whitespace",
        :filter => ['asciifolding', 'lowercase', 'text_ngram']
      }
    }
  } do
    mapping do
      indexes :username, :type => "string", :analyzer =>
"text_analyzer"
      indexes :name, :type => "string", :analyzer => "text_analyzer"
      indexes :surname, :type => "string", :analyzer =>
"text_analyzer"
      indexes :addresses, :analyzer => "text_analyzer"
      indexes :kind, :type => "string", :analyzer => "text_analyzer"
      indexes :competitions_ids, :type => "array"
      indexes :no_of_art, :type => 'integer'
      indexes :created_at, :type => "date"
    end
  end

  def to_indexed_json
     # ...
  end
end
```

Příklad č. 36 - Aktualizované nastavení modelu User pro podporu vyhledávání bez diakritiky

V modelu jsem udělal několik změn. Přidal jsem *settings* blok, ve kterém jsem nadefinoval potřebné nastavení pro vyhledávání. Dá se říct, že to co jsem nastavoval pro Solr v jeho schema.xml, to jsem přepsal zde v zápisu pro Tire. Princip zůstává stejný. Blok mapping je nyní jako vnitřní blok bloku settings.

Ke každému z atributů jsem v závěru explicitně nadefinoval datový typ. ElasticSearch uvádí, že datový typ atributů se mapuje podle indexováného JSON řetězce. Tudíž si vystačí bez explicitně definovaných datových typů [23]. Já jsem i přesto datové typy nadefinoval.

V závěru jsem u atributů, které chci, aby se indexovaly pomocí mého nového analyzeru, explicitně nadefinoval tento analyzer.

```
$ bundle exec rake environment tire:import CLASS='User' FORCE=true
```

Příklad č. 37 - Import všech modelů třídy User do indexu ElasticSearch, kde force, pokud je true, najdříve index smaže a pak jej znovu vytvoří

Když jsem tímto způsobem nakonfiguroval svůj model, použil jsem kódu v příkladu č. 37 pro zaindexování dat. Data se zaindexovaly, ale bez toho, aby se použilo mé nastavení a mapování atributů.

Když jsem se pokusil o to samé z Rails konzole, tak se mi ani index nepodařilo vytvořit.

```
Tire.index("users").delete
User.create_elasticsearch_index
 => false
```

Příklad č. 38 - Smazání indexu se jménem users a vytvoření nového indexu pomocí volání třídní metody na modelu

Opravdu jsem se dostal do bodu, kdy nevím, co je špatně. Index se mi nedaří vytvořit. Metoda v příkladu č. 38 vrací true nebo false, takže nemám ani možnost zjistit, co je špatně. Je možné, že problém je opravdu s připojením. Tire používá gem RestClient pro komunikaci s ElasticSearch. Pokud selže RestClient, tak pak Tire používá jako záchranu Curl. Proto možná některé operace fungují a jiné ne. Abych vyvrátil to, co jsem zrovna řekl, pokoušel jsem se použít vnitřní metody Tire, které používají RestClient, pro volání ElasticSearch a ty fungovaly.

Než jsem úplně ztratil všechny naděje v Tire, pokusil jsem se odstranit

všechno nastavení a nechat pouze blok mapping. I přesto se mi z konzole index nepodařilo vytvořit a importovací úloha sice data naimporotovala, ale bez mého mapování. Proto ElasticSearch nyní opouštím. Strávil jsem nad ním 3 dny a nebyl jsem schopný k tomuto problému shromáždit skoro žádné informace.

Tento gem je teprve rok starý a stále prochází intenzivním vývojem. Vypadá to, že je potřeba ještě rok počkat.

3.6. PostgreSQL

Pro fulltextové vyhledávání nad PostgreSQL jsou podle mého průzkumu nejpoužívanější dva gemy - Texticle a pg_search.

Texticle pochází z dílny Aarona Pattersona, tvůrce ActiveRecord pro Rails. Tento gem je výborný pro svou jednoduchost a ne náhodnou podobnost s dotazy pro ActiveRecord. Jeho výhodou je jednoduchost, ale jak sami jeho tvůrci přiznávají, méně konfigurovatelný než pg_search [28].

pg_search na první pohled překvapí možnostmi, které nabízí. Vše je dobře vidět na jejich domovské stránce na Githubu [29]. Právě pro to, že tento gem nabízí takovéto možnosti, si jej vybírám a vyzkouším na něm naimplementovat řešení pro zadání. Jako ve všech ostatních případech, instaluji tento gem vložením jeho jména do Gemfile a spuštěním příkazu bundle.

```
# Gemfile

gem 'pg_search'

# in console

bundle
```

Příklad č. 39 - Smazání indexu se jménem users a vytvoření nového indexu pomocí volání třídní metody na modelu

Abych rozšířil možnosti modelů o fulltextové vyhledávání, je potřeba přidat následující řádek kódu.

```
# app/models/user.rb

class User
  include PgSearch

  # ...

end
```

Příklad č. 40 - Zahrnutí pg_search do modelu

pg_search podporuje dvě techniky vyhledávání. Jedna je vyhledávání nad všemi modely, resp. nad všemi označenými modely, a druhá technika vyhledává vždy nad jedním modelem. Pro případy globálního vyhledávání si pg_search vytváří novou tabulku v databázi, do které ukládá data potřebná pro vyhledávání nad všemy modely. Mě se týká případ druhý.

Ze zadání vím, že potřebuji nad modelem vyhledávat bez háčků a čárek, podřetězce a nad vybranými atributy, tak pg_search nad modelem User naimplementuji.

```
# app/models/user.rb

class User
  include PgSearch
  pg_search_scope :search_model, :against => [:username, :name,
:surname], :using => [:tsearch, :trigram], :ignoring => :accents

  # ...

end
```

Příklad č. 41 - Zahrnutí pg_search do modelu

Prozatím jsem nastavil model tak, aby vyhledával pouze nad uživatelským jménem, jménem uživatele a jeho příjmením, protože jsem si chtěl nejdříve vyzkoušet chování vyhledávání v konzoli.

Na první pohled se všechno chovalo v pořádku. Při zadání slov jako „davetsunami“, „dave“, „ceskytucnak“, „tucnak“ nebo „tsunami“, vyhledávání nachází patřičné uživatele. Nicméně když jsem zkrátil výrazy na slova jako např. "tsun", "tuc" nebo "tucna", nic nebylo nalezeno. Protože vím, jak vyhledává Apache Solr, přišlo mi toto chování opravdu divné a musím říct, že mi to hodně vadilo.

```
# in Rails console

> User.search_model("dave")
  User Load (48.5ms)  SELECT "users".*,
(ts_rank((to_tsvector('simple', unaccent(coalesce("users"."username",
'')))), (to_tsquery('simple', ''' ' || unaccent('dave') || ' ''')),
0)) AS pg_search_rank FROM "users" WHERE (((to_tsvector('simple',
unaccent(coalesce("users"."username", '')))) @@ (to_tsquery('simple',
''' ' || unaccent('dave') || ' '''))) OR
((unaccent(coalesce("users"."username", ''))) % unaccent('dave')))
ORDER BY pg_search_rank DESC, "users"."id" ASC
 => [#<User id: 4, email: "david.horsak@nevico.eu" ...
```

Příklad č. 42 - Test vyhledávání pomocí pg_search

Napadlo mě, že vyhledávání nebere v potaz výsledky, které nesplňují určitou hranici shody nebo relevantnosti. Bohužel jsem nenašel nikde nápovědu, ani můj průzkum nenašel nic, jak toto ovlivnit nebo změnit.

Než jsem začal s další implementací, hodně jsem přímýšlel nad rozdíly mezi použitím PostgreSQL a Apache Solr s ElasticSearch. V klasických fulltextových vyhledávačích jsem musel indexovat i pole, podle kterých jsem sice nechtěl vyhledávat, ale musel jsem podle nich řadit. V případě PostgreSQL, použiju do SQL datazu funkci ORDER.

Pokud chci provést vyhledávání nad modelem Info, kde mám uložené informace o adrese ve formě serializovaného pole, musím nejprve provést JOIN s tabulkou modelu User. S gemem pg_search to vůbec provést nejde. Důvod je jednoduchý. Rails ActiveRecord ve svých SQL dotazech vždy uvádí u atributu i jmeno tabulky, proto i když v implementaci modelu User uvedu i atribut localized_attributes, vždy je k němu připojeno i jméno tabulky users, takže JOIN pomocí ActiveRecord je s tímto gemem nepoužitelný.

Gem umožňuje variantu globálního vyhledávání nebo vyhledávání nad jedním modelem. Mohl bych pro tuto možnost využít globalního vyhledávání, čímž bych pak ztratil možnost využít stejného „triku" i pro model Art, se kterým je asociováno také více modelů, které ukládají jeho data.

Dalším možným řešením tohoto problému je denormalizace. Ta je pro mě nepřijatelná, protože z důvodu výkonosti Artvasion jsem právě sáhli k tomu, že jeden model bude mít k sobě navázané další modely a vše se bude načítat pouze

tam, kde je potřeba. V převážné většině systému si uživatel vystačí pouze s uživatelským jménem anebo jménem. Informace o jeho narozeninách, pohlaví a místě, kde se nachází, se zobrazují pouze na jeho profilu nebo ve vyhledávání.

Dalším háčkem jsou pro mě lokalizované adresy v modelu Info, podle kterých také potřebuju vyhledávat. Rails serializuje objekty do formátu YAML. Adresy jsou uloženy ve formě hashe, a pokud udělám SQL dotaz v psql konzoli, dostanu následující:

```
artvasion=# select localized_addresses from infos where user_id = 5;
    localized_addresses
----------------------------
 ---                       +
 :original: Česká republika+
 :en: Czech Republic       +
 :cs: Česká republika      +
 :de: Tschechische Republik+
 :ru: Чехия                +

(1 row)
```

Příklad č. 43- Serializovaný hash jako výsledek SQL dotazu

Důležité pro mě je, aby šlo i nad takto serializovanými dotazy vyhledávat. Apache Solr a i ElasticSearch měly toto vyřešené.

```
artvasion=# select localized_addresses from infos where
localized_addresses @@ 'czech republic';
                             localized_addresses
----------------------------------------------------------------------
 ---                                  +
 :original: Prague, Czech Republic    +
 :en:                                 +
 :cs:                                 +
 :de: Prag, Tschechische Republik     +
 :ru: Прага, Чехия                    +

 ...
```

Příklad č. 44 - Fulltextové vyhledávání nad serializovaným hashem

Jak ukazuje předchozí příklad, fulltextově vyhledávat na serializovanými objekty lze, protože se serializují do sloupců typu text. Bohužel jak ukazuje příklad

č. 45, vyhledávání probíhá i nad klíči serializovaného hashe. Což znamená, že po vyhledání slova „ru“, dostává uživatel absolutně nerelevatní výsledky.

```
artvasion=# select localized_addresses from infos where
localized_addresses @@ 'ru';
                            localized_addresses
--------------------------------------------------------------------
 ---                                   +
 :original: Prague, Czech Republic     +
 :en:                                  +
 :cs:                                  +
 :de: Prag, Tschechische Republik      +
 :ru: Прага, Чехия                     +

 ...
```

Příklad č. 45 - Fulltextové vyhledávání nad serializovaným hashem s dotazem na slovo „ru“

Tento problém by opět vyřešila denormalizace. Dříve Artvasion opravdu ukládalo každý jazyk do svého sloupce. Toto řešení bylo komplikované v tom, že když se přidával nový jazyk, musely se přidat do několika tabulek nové sloupce. Řešení pomocí serializovaného hashe je elegatní v tom, že se žádné sloupce přidávat nemusí. Prostě se rozšíří v systému dostupné lokále a při ukládání adres si systém sám dohledá překlady pro používané lokále. Navíc když se přecházelo mezi různými vývojovými větvemi, hodněkrát se stávalo, že větev, které měla více jazyků v nastavení, při spuštění aplikace vyhazovala výjimky, protože jí chyběly právě sloupce pro každý jazyk.

Dostal jsem se do momentu, kdy jsem zjistil, že nemohu použít žádný existující gem, který by splňoval potřeby aplikace, abych byl schopný splnit zadání pro fulltextové vyhledávání.

Gem je určitě použitelný pro realizaci vyhledávání nad jedním modelem bez serializovaných atributů nebo při použití globálního vyhledávání nad vybranými modely. Určitým řešení je i denormalizace, která by umožnila použití gemu za cenu snížení rychlosti aplikace.

Pokud bych chtěl realizovat fulltextové vyhledávání a splnit tak zádání bez změny datového modelu, musel bych si všechny dotazy pro vyhledávání napsat v SQL sám.

Posledním celkem zásadním faktem je to, že pokud by se realizovalo fulltextové vyhledávání nad PostgreSQL, znamenalo by to pro aplikaci velice úzké svázání s touto databází, a o to pak může být migrace na jiné řešení složitější, ať už se jedná o relační databázi nebo NoSQL databázi.

Fulltextové vyhledávání pomocí PostgreSQL má určitě smysl tam, kde jsou aplikace jednoduché, není dostupné jiné řešení nebo na to z určitých důvodů nejsou peníze. Pro složité aplikace fulltextové vyhledávání není určitě dobrým nápadem, pokud vlastník aplikace nechce věnovat velké usílí nebo sumu peněz na jeho implementaci. Mě to přijde jako zbytečná ztráta času, když mohu mít řešení jako Apache Solr za jedno odpoledne naimplementované, integrované v aplikaci a nemusím se starat o žádnou optimalizaci.

3.7. ElasticSearch 2

Poté co jsem se rozhodl neimplemenovat fulltextové vyhledávání nad PostgreSQL z důvodů velké náročnosti a nemožnosti použít pro řešení gem, upnul jsem se zpět k ElasticSearch.

Po několika hodinách zkoumání jsem našel problém, který bránil připojení Tire k ElasticSearch při inicializaci. Tím problémem byl gem Devise. Tento gem zajišťuje komplexní řešení autentifikace a při své vlastní inicializaci načítá moduly pro ActiveRecord. Při načítání těchto modulů se nejspíše spouštěl i nějaký callback Tire, který navazoval komunikaci s ElasticSearch. Díky tomu se Tire snažilo komunikovat s ElasticSearch, i když samo nemělo správně nastavenou IP adresu ElasticSearch serveru. Pořádně jsem si oddechl a vrhl jsem se zpět do řešení fulltextového vyhledávání pomocí ElasticSearch.

Obnovil jsem nastavení ve svých modelech a zkusil data znovu naimportovat. Vše vypadalo v pořádku. Moje mapping modelu bylo uloženo v indexu a objekty byly naimportovány.

Moje nadšení, že pokračuji dál, rychle opadlo. Nefunovalo vyhledávání bez háčků a čárek ani částečných výrazů. Musel jsem zadat celé své uživatelské jméno, abych našel svůj záznam. Začal jsem hledat chybu v nastavení. Ještě jednou jsem si dotazem na server zkontroloval svoje mapping:

```
# in console

$ curl -XGET 'http://192.168.1.138:9200/users/_mapping'

{"users":{"user":{"properties":{"addresses":
{"type":"string","analyzer":"my_analyzer"},"competitions_ids":
{"type":"string"},"created_at":
{"type":"date","format":"dateOptionalTime"},"kind":
{"type":"string","analyzer":"my_analyzer"},"name":
{"type":"string","analyzer":"my_analyzer"},"no_of_art":
{"type":"integer"},"surname":
{"type":"string","analyzer":"my_analyzer"},"username":
{"type":"string","analyzer":"my_analyzer"}}}}}
```

Příklad č. 46 - Kontrola mapping u uživatele

Mapping vypadalo v pořádku. Přesně odpovídalo mému nastavení v modelu. Otázkou stále zůstavá settings:

```
# in console

$ curl -XGET 'http://192.168.1.138:9200/users/_settings'

{"users":{"settings":
{"index.analysis.analyzer.my_analyzer.filter.0":"asciifolding","index
.analysis.analyzer.my_analyzer.filter.1":"lowercase","index.analysis.
analyzer.my_analyzer.filter.2":"ngram","index.analysis.analyzer.my_an
alyzer.tokenizer":"lowercase","index.number_of_shards":"5","index.num
ber_of_replicas":"1","index.version.created":"190099"}}}
```

Příklad č. 47 - Kontrola settings u uživatele

Odezvu jsem nechápal. Proč mapping vypadá prostě jako správný JSON, klíče v hashi skrývají další hash až po hodnoty? V settings jsou klíče tečkovou notací poskladané za sebou. Je to takto správně?

Snažil jsem se hledat v dokumentaci ElasticSearch, ale nikde jsem odpověď na svou otázku nenašel. Samozřejmě jsem i googloval, ale jediný příklad jsem našel na neznámém fóru, kde uživatel ukazoval odezvu při dotazu na nastavení naformátovanou jako mapping (klíče bez tečkové notace).

Jak teď poznám, že moje nastavení funguje správně? Vyhledávat pomocí částí hledaného řetěce mi nešlo, nevím, jestli moje nastavení platí. Na ElasticSearch jsem se velice těšil, protože se teď celkem hodně začíná používat v kombinaci

s gemem Tire v Ruby on Rails komunitě. Bohužel moje cesta je spíše trnitá s hlubokými příkopy než asfaltová rovinka.

ElasticSearch nabízí v rámci svého API také možnost vyzkoušet analyzery a také jim jde poslat text, který vrátí zanalyzovaný [31]. Vyzkoušel jsem si pomocí toho API zavolat index s indexovanými uživately a na něm zavolat můj nakonfigurovaný analyzer, aby mi zanalyzoval řetězec textu (po dlouhých hodinách debuggování jsem text_analyzer v obavě před kolozí jmen přejmenoval na my_analyzer).

```
# in console

$ curl -XGET '192.168.1.138:9200/users/_analyze?analyzer=my_analyzer'
-d 'Horsák'

{"tokens":
[{"token":"h","start_offset":0,"end_offset":1,"type":"word","position
":1},
{"token":"o","start_offset":1,"end_offset":2,"type":"word","position"
:2},
{"token":"r","start_offset":2,"end_offset":3,"type":"word","position"
:3},
{"token":"s","start_offset":3,"end_offset":4,"type":"word","position"
:4},
{"token":"a","start_offset":4,"end_offset":5,"type":"word","position"
:5}, ...
```

Příklad č. 48 - Kontrola analyzeru u nastavení uživatele

Toto fungovalo. Odezva mi vrátila slovo rozčleněné na tokeny, zmenšené, bez diakritiky a rozčleněné na tokeny od 2 znaků délky po délku celého mého příjmení. Proč teda nefunguje metoda vyhledávání?

```
# in Rails console

> User.search("horsak")

 => #<Tire::Results::Collection:0x007fcdae136d88
@response={"took"=>1, "timed_out"=>false, "_shards"=>{"total"=>5,
"successful"=>5, "failed"=>0}, "hits"=>{"total"=>0, "max_score"=>nil,
"hits"=>[]}}, @options={}, @time=1, @total=0, @facets=nil,
@wrapper=Tire::Results::Item>
```

```
> User.search { query { string "hors" } }

 => #<Tire::Results::Collection:0x007fe18683e5b8
@response={"took"=>2, "timed_out"=>false, "_shards"=>{"total"=>5,
"successful"=>5, "failed"=>0}, "hits"=>{"total"=>0, "max_score"=>nil,
"hits"=>[]}}, @options={}, @time=2, @total=0, @facets=nil,
@wrapper=Tire::Results::Item>
```

Příklad č. 49 - Testování vyhledávání

Netuším, kde by mohl být problém. Z dostupných zdrojů nejsem schopný zjistit, co dělám špatně. Během celé práce s ElasticSearch mám neustále nějaký problém. Končím...

4. Závěr

Z celé práce jasně vyplývá jeden vítěz a tím je Apache Solr v kombinaci s gemem Sunspot. Apache Solr se nainstaluje jako balíček a kromě malé konfigurace na serveru a změny schématu všechna práce probíhá v Ruby. Řešení i gem jsou na trhu už dlouho, oba jsou výborně zdokumentované, mají rozsáhlou komunitu uživatelů, a když jsem narazil na nějaký problém, vše bylo v jednotkách minut vyřešeno. Stačilo jedno odpoledne a měl jsem výsledek dle zadaní. Tuto kombinaci vřele doporučuji a nemám žádnou negativní připomínku.

Svůj názor na PostgreSQL a fulltextové vyhledávání pomocí integrovaného modulu tsearch2 a contrib modulů už jsem naznačil v praktické části. Je to skvělé řešení tam, kde není možné použít externí vyhledávácí nástroj nebo v případech, kde máme jednoduchou aplikaci, která neobsahuje žádné speciální řešení. Konfigurace je poté za pomocí gemu pg_search jednoduchá a rychlá a opět je to řešení hotové za pár minut. Pro složité aplikace je to prozatím řešení nepoužitelné. Při výběru tohoto řešení je také důležité brát v potaz optimalizaci, tzn. každý dotaz si vykoušet, pustit jej v rámci EXPLAIN funkce a v případě potřeby kopírovat data do sloupců typu ts_vector a vytvářet GiST nebo GIN indexy pro rychlé vyhledávání. Může to být také skvělé řešení tam, kde chybí peníze na provoz fulltextového nástroje, ale součástí aplikace již databáze PostgreSQL je.

ElasticSearch na rozdíl od Apache Solr a PostgreSQL je pro mne velkým zklamáním. Možná je to také tím, že jsem před jeho implementací s gemem Tire měl velká očekávání a celkem jsem se na práci s ním těšil a také ho měl za svého favorita. Rychle jsem byl vyveden z míry a opraven. Těžko mohu říct, jestli je chyba na straně ElasticSearch nebo gemu, který jsem používal. Na ElaticSearch mi vadí, že mu chybí minimálně webové rozhraní, kde bych si mohl přehledně zkontrolovat, jak mám jednotlivé indexy nastavené. To třeba Apache Solr má a také mi dovoluje si vyzkoušet všechny analyzery a ukazuje jejich výstupy. Na Tire mi nejvíce vadí, že vůbec nekomunikuje s uživatelem a nepropaguje výjimky. Když jsem na úplném začátku práce špatně pojmenoval tokenizer a chtěl jsem skrze Tire vytvořit nový index, Tire mi neustále vracelo pouze false. Když jsem to samé pak chtěl udělat přímo pomocí programu Curl na ElasticSearch, odezva mi vrátila chybový kód se zprávou. Tyto zprávy Tire nijak nepropaguje dál, a to je podle mě velká chyba. Kdyby mi Tire tuto zprávu vrátilo, nestrávil bych desítky

minut hledáním chyb. V neposlední řadě okolo ElasticSearch a Tire není tak velká komunita jako v porovnání s Apache Solr a Sunspot. Řešení problému není jednoduché a rychlé. Proto v tuto chvíli ElasticSearch a Tire nedoporučuji.

I přesto že jsem již vynesl ortel na vybranými řešeními, zůstávám optimistou. Obliba ElasticSearch a Tire roste a po roce bude jejich vývoj zase podstatně dál. Těším se za rok, až se k němu vrátím a vyzkouším jej.

V této práci jsem porovnával tři řešení fulltextového vyhledávání a to pomocí Apache Solr a gemu Sunspot, PostgreSQL a gemu pg_search a ElasticSearch a gemu Tire. Je potřeba mít na paměti, že existují i další dvě velice oblíbené řešení a to Spinx a gem Thinking Spinx a Ferret a gem acts_as_ferret. Pokud se vám ani jedno z mých porovnávaných řešení nelíbilo, určitě se podívajte na tyto jmenované dvojice.

Na úplný závěr se sluší říct, co mi tato práce dala a co si odnáším dál do své kariéry vývojáře. Především jsem se utvrdil v tom, že při výběru fulltextového vyhledávacího nástroje při mě nejspíše stála ruka boží, když jsem jako první řešení zvolil Apache Solr, které jsem okamžitě naimplemenoval do Artvasion a spustil s ním celou aplikaci před začátkem registrace umělců do soutěže Český tučňák 2012. Kdybych začal ElasticSearch, asi by pořadatelé museli odložit registraci. To samozřejmě není vše. Pochopil jsem základní principy fulltextového vyhledávání a i to, jakým způsobem je realizováno nad relační databází. Fulltextové vyhledávání je určitě budoucnost, což se do značné míry prokazuje stále více, ale i přesto si myslím, že je tento obor stále na velkém začátku a čeká ho ještě dlouhý vývoj.

5. Seznam použitých zdrojů

1. KISSMETRICS: The 2012 Web Analytics Review [online]: Dostupné z URL <http://blog.kissmetrics.com/2011-web-analytics-review/>
2. APACHE SOLR: Features [online]: Dostupné z URL <http://lucene.apache.org/solr/>
3. SOLR WIKI: Analyzers, Tokenizers, and Token Filters [online]: Dostupné z URL <http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters>
4. RVM: Quick Install [online]: Dostupné z URL <https://rvm.io/>
5. ELASTICSEARCH: Setting up elasticsearch on Debian [online]: Dostupné z URL <http://www.elasticsearch.org/tutorials/2010/07/02/setting-up-elasticsearch-on-debian.html>
6. ACTS_AS_SORL: Documentation [online]: Dostupné z URL <http://acts-as-solr.rubyforge.org/>
7. SUNSPOT: Homepage [online]: Dostupné z URL <http://sunspot.github.com/>
8. ACTS_AS_SOLR: Git repozitář [online]: Dostupné z URL <https://github.com/onemorecloud/acts_as_solr>
9. ACTS_AS_SORL: Network graph [online]: Dostupné z URL <https://github.com/onemorecloud/acts_as_solr/network>
10. SOLRSAN: Git repozitář [online]: Dostupné z URL <https://github.com/tc/solrsan>
11. SUNSPOT WIKI: Setting up Classes for Search and Indexing [online]: Dostupné z URL <https://github.com/sunspot/sunspot/wiki/Setting-up-classes-for-search-and-indexing>
12. SUNSPOT WIKI: Scoping by Attribute Fields [online]: Dostupné z URL <https://github.com/sunspot/sunspot/wiki/Scoping-by-attribute-fields>
13. SUNSPOT WIKI: Ordering and Paginations [online]: Dostupné z URL <https://github.com/sunspot/sunspot/wiki/Ordering-and-pagination>

14. ELASTICSEARCH: Homepage [online]: Dostupné z URL <http://www.elasticsearch.org/>

15. ELASTICSEARCH: API [online]: Dostupné z URL <http://www.elasticsearch.org/guide/reference/api/>

16. ELASTICSEARCH: Analysis [online]: Dostupné z URL <http://www.elasticsearch.org/guide/reference/index-modules/analysis/>

17. ELASTICSEARCH: Indices Update Settings API [online]: Dostupné z URL <http://www.elasticsearch.org/guide/reference/api/admin-indices-update-settings.html>

18. POSTGRESQL 9.1 DOCUMENTATION: F.43. tsearch2 [online]: Dostupné z URL <http://www.postgresql.org/docs/9.1/static/tsearch2.html>

19. POSTGRESQL 9.1 DOCUMENTATION: 12.2. Tables and Indexes [online]: Dostupné z URL <http://www.postgresql.org/docs/9.1/static/textsearch-tables.html>

20. POSTGRESQL 9.1 DOCUMENTATION: Create extension [online]: Dostupné z URL <http://www.postgresql.org/docs/9.1/static/sql-createextension.html>

21. ELASTICSEARCH: Configuration [online]: Dostupné z URL <http://www.elasticsearch.org/guide/reference/setup/configuration.html>

22. GITHUB: karmi/tire [online]: Dostupné z URL <https://github.com/karmi/tire>

23. ELASTICSEARCH: Clients & Integrations [online]: Dostupné z URL <http://www.elasticsearch.org/guide/appendix/clients.html>

24. GITHUB: karmi/tire - #<Errno::ECONNREFUSED: Connection refused - connect(2)> on boot - Issue #324 [online]: Dostupné z URL <https://github.com/karmi/tire/issues/324>

25. ELASTICSEARCH: Core Types [online]: Dostupné z URL <http://www.elasticsearch.org/guide/reference/mapping/core-

types.html>

26. POSTGRESQL 9.1 DOCUMENTATION: Dictionaries [online]: Dostupné z URL <http://www.postgresql.org/docs/9.1/static/textsearch-dictionaries.html>
27. POSTGRESQL 9.1 DOCUMENTATION: GiST and GIN Index Types [online]: Dostupné z URL <http://www.postgresql.org/docs/9.1/static/textsearch-indexes.html>
28. TEXTICLE: Homepage [online]: Dostupné z URL <http://tenderlove.github.com/texticle/>
29. GITHUB: Casecommons / pg_search [online]: Dostupné z URL <https://github.com/Casecommons/pg_search>
30. BUNDLER: Homepage [online]: Dostupné z URL <http://gembundler.com/>
31. ELASTICSEARCH: Analyze API [online]: Dostupné z URL <http://www.elasticsearch.org/guide/reference/api/admin-indices-analyze.html>

6. Seznam příkladů

Příklad č. 1 - Základní struktura složek Ruby on Rails aplikace 9

Příklad č. 2 - Standardní nastavení schéma Apache Solr pro datový typ text 11

Příklad č. 3 - Vytvoření nového indexu s názvem Artvasion 14

Příklad č. 4 - Uložení uživatele do indexu 14

Příklad č. 5 - Získání záznamu z indexu 14

Příklad č. 6 - Vyhledávání nad uživateli 15

Příklad č. 7 - Smazání uživatele z indexu 15

Příklad č. 8 - Nastavení indexu pro analýzu indexovaného obsahu 16

Příklad č. 9 - Použití contrib modulů v PostgreSQL 17

Příklad č. 10 - Fulltextové vyhledávání v řetězci 18

Příklad č. 11 - Základní fulltextové vyhledávání podle uživatelského jména 18

Příklad č. 12 - Vyhledávání bez diakritiky 19

Příklad č. 13 - Instalace pg_trgm do databáze 19

Příklad č. 14 - Jak funguje pg_trgm 19

Příklad č. 15 - Vyhládávní pomocí pg_trgm seřazené podle shody 20

Příklad č. 16 - Instalace gemu sunspot do Ruby on Rails aplikace 25

Příklad č. 17 - Spuštění generátoru pro vytvoří souboru sunspot.yml 27

Příklad č. 18 - Vzorové nastavení gemu sunspot 27

Příklad č. 19 - Nastavení pole se jménem text_pre 28

Příklad č. 20 - Použítí tokenizeru třídy solr.WhitespaceTokeanizerFactory 29

Příklad č. 21 - Použítí flterusolr.ASCIIFoldingFilterFactory 29

Příklad č. 22 - Použítí fltru třídy solr.LowerCaseFilterFactory 29

Příklad č. 23 - Použítí fltru třídy solr.NGramFilterFactory 29

Příklad č. 24 - Dynamické pole pro text_pre 30

Příklad č. 25 - Nastavení modelu User pro vyhledávání 31

Příklad č. 26 - Implementace kontroleru pro vyhledávání nad modelem User 33

Příklad č. 27 - Nastavení modelu Art pro vyhledávání 34

Příklad č. 28 - Implementace kontroleru pro vyhledávání nad modelem Art 35

Příklad č. 29 - Stažení aktuální verze ElasticSearch 36

Příklad č. 30 - Test funkčnosti ElasticSearch vytvořením nového indexu 37

Příklad č. 31 - Test funkčnosti ElasticSearch vytvořením nového indexu 37

Příklad č. 32 - Nastavení modelu User pro vyhledávání 37

Příklad č. 33 - Rozšířené nastavení modelu User pro vyhledávání 38

Příklad č. 34 - Konfgurace IP adresy ElasticSearch pro gem Tire 39

Příklad č. 35 - Spuštění Rails konzole a test konektivity s ElasticSearch pomocí gemu HTTParty 39

Příklad č. 36 - Import dat do databáze a test vyhledávání 40

Příklad č. 36 - Aktualizované nastavení modelu User pro podporu vyhledávání bez diakritiky 41

Příklad č. 37 - Import všech modelů třídy User do indexu 42

ElasticSearch, kde force, pokud je true, najdříve index smaže a pak jej znovu vytvoří

Příklad č. 38 - Smazání indexu se jménem users a vytvoření nového indexu pomocí volání třídní metody na modelu 42

Příklad č. 39 - Smazání indexu se jménem users a vytvoření nového indexu pomocí volání třídní metody na modelu 43

Příklad č. 40 - Zahrnutí pg_search do modelu 44

Příklad č. 42 - Test vyhledávání pomocí pg_search 45

Příklad č. 43- Serializovaný hash jako výsledek SQL dotazu 46

Příklad č. 44 - Fulltextové vyhledávání nad serializovaným hashem 46

Příklad č. 45 - Fulltextové vyhledávání nad serializovaným hashem s dotazem na slovo „ru“ 47

Příklad č. 46 - Kontrola mapping u uživatele 49

Příklad č. 47 - Kontrola settings u uživatele 49

Příklad č. 48 - Kontrola analyzeru u nastavení uživatele 50

Příklad č. 49 - Testování vyhledávání 50

7. Seznam obrázků

Obrázek 1: Část datového modelu aplikace Artvasion, nad kterou bude probíhat fulltextové vyhledávání 22

8. Seznam tabulek

Tabulka 1: Popis tříd v datovém modelu 23