Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Relační a nerelační modelování pro portál elektrofyziologických experimentů

Plzeň, 2014 Martin Bydžovský

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

.....

V Plzni dne 1. 5. 2014 Martin Bydžovský

Poděkování

Rád bych poděkoval Jakubovi Jindrovi za pomoc s instalací databáze Elasticsearch, Lukáši Drbalovi za konzultace ohledně optimalizace Elasticsearch dotazů a v neposlední řadě Evě Janouškové za velkou morální podporu.

Abstract

The goal of this thesis is to analyze current situation of NoSQL databases with focus on fulltext searching. The data layer of EEG/ERP Portal has to be evaluated and key parts identified as suitable candidates for storing in one of the storage engines. The thesis compares several NoSQL databases with respect to the needs of the Portal. After one is chosen, the database will be deeply described – from installation, through configuration to final integration with current application.

Another target of the thesis is to find a full-featured replacement for actual Oracle relational database, as the licensing conditions cease to be sufficient for the needs of the Portal. A quick comparison of relational databases will be made and the most appropriate will be chosen. Finally, all (relational) data should be migrated to this database.

Obsah

1. Úvod

Potřeba efektivně ukládat digitální data existuje od doby, kdy byl vytvořen první počítač. Postupem času z této problematiky vzniklo celé technologické odvětví, kdy v roce 1970 Edgar F. Codd definoval pojem "relační databáze". Tyto databáze mají schraňovat data nezávisle na použitém aplikačním jazyce a dokonce i na tom, jak jsou data fyzicky uložena. V osmdesátých letech potom IBM přichází s jazykem SQL a vznikají první databáze: Oracle nebo IBM se svojí DB2.

V poslední době začíná být velmi populární databázový koncept NoSQL, který data ukládá jinak než tyto tradiční databáze. Často se jedná o specificky zaměřená, optimalizovaná úložiště typu klíč-hodnota, přičemž jako hodnota může být jakýkoliv objekt, obsahující množství, předem nespecifikovaných atributů (hovoří se o schema-free databázích). Další z výhod těchto systémů je jednoduché (většinou implicitně podporované) vertikální škálování. Využití těchto systémů je vhodné v oblasti tzv. Big Data[1] nebo např. při fulltextovém hledání.

Hlavním cílem této práce je průzkum aktuální nabídky NoSQL databází s primárním zaměřením na fulltextové vyhledávání. Za tímto účelem bude provedena analýza stávajícího datového modelu EEG/ERP portálu. Budou identifikovány nejdůležitější entity a jejich vazby na ostatní části systému. Dále se určí data, která jsou vhodnými kandidáty na přesunutí do nějaké NoSQL databáze. Jedna z klíčových vlastností, proč je nutné NoSQL databázi implementovat, je požadavek na fulltextové vyhledávání. Poté, co se zvolí nejvhodnější řešení, vybraná data se přesunou do tohoto nového úložiště. Bude nutné provést instalaci, vhodnou konfiguraci a nastavit specifika indexování tak, aby vyhledávání poskytovalo relevantní výsledky pro potřeby EEG/ERP portálu. Integrace do stávající aplikace by měla probíhat pouze na datové vrstvě tak, aby bylo ovlivněno co nejméně vrstev vyšších. Z pohledu třívrstvé architektury by se vyšších vrstev vlastně tato změna vůbec neměla dotknout.

Dalším bodem této práce je změnit úložiště relačních dat, protože stávající databáze Oracle přestává být z licenčních důvodů vhodná. Bude nutné najít databázový

[1] Big Data: většinou se myslí data o objemech v řádu TB až PB.

systém, který se bude co nejvíce podobat svojí funkcionalitou původní Oracle databázi a všechna relační data do této databáze zmigrovat.

V neposlední řadě se tato práce věnuje celkovému refaktoringu[2] datové vrstvy aplikace. V projektu existuje několik problémových míst, které se postupem času tak, jak na aplikaci pracovalo množství různých studentů, rozrostlo a kód začíná nepřehledný a do budoucna špatně udržitelný. Část práce je tedy věnována pokusu o napravení těchto neduhů.

[2] Refaktoring: úprava kódu za účelem lepší přehlednosti a jednoduší rozšiřitelnosti při zachování totožné funkcionality.

2. Aktuální stav

2.1. Popis aplikace

EEG[3]/ERP[4] portál (dále jen Portál) je webová aplikace, vyvíjená na Katedře informatiky a výpočetní techniky při Západočeské univerzitě v Plzni, která slouží jako centrální místo pro výzkumné pracovníky, kteří provádějí, zpracovávají nebo analyzují výsledky měření EEG/ERP. Dovoluje registrovaným uživatelům nahrávat tyto měřené experimenty, přidávat k nim metainformace týkající se daného měření, kategorizovat je a v neposlední řadě nad nimi vyhledávat podle různých kritérií. Obsahuje propracovaný systém uživatelských rolí a oprávnění, sdružuje jednotlivé uživatele do tzv. výzkumných skupin (Research Groups), v rámci které mohou sdílet jednotlivé experimenty. Dále aplikace slouží jako agregátor vědeckých článků, týkající se problematiky výzkumu měření ERP.

Kompletní funkcionalita portálu je následující [1]:

- Vytváření a správa uživatelských účtů.
- Vytváření výzkumných skupin a jejich správa.
- Ukládání, analýza a zpracování dat a metadat experimentů.
- Vytváření a správa licencí a jejich následné přiřazování k experimentům.
- Sdílení dat a metadat mezi uživateli a výzkumnými skupinami.
- Přidávání článků a jejich komentovaní pro uživatele a skupiny.
- Vyhledávání v EEG databázi.
- Přihlášení do Portálu pomocí sociálních sítí Facebook a LinkedIn.

V tuto chvíli Portál používají převážně lidé ze Západočeské univerzity, ale mohl by však dobře sloužit např. v laboratořích, které se zabývají výzkumem ERP. Z tohoto

[3] Elektroencefalogram (zkráceně EEG) je záznam časové změny elektrického potenciálu způsobeného mozkovou aktivitou. Tento záznam je pořízen elektroencefalografem.

[4] Event Related Potencial (ERP, evokovaný potenciál) je měřená odpověď mozku na konkrétní vyvolanou událost nebo podnět.

důvodu se do Portálu přidávají funkce, které umožňují licencování naměřených experimentů a jejich následný prodej ostatním uživatelům portálu. Na jedné straně by tedy mohly být výzkumné instituce, které provádějí měření a zveřejňují získaná data. Na straně druhé potom firmy, které provádějí analýzu těchto dat a budou za publikované experimenty platit.

Jde o aplikaci, která si klade za cíl být centralizovaným místem pro odbornou (jak akademickou, tak z praxe) komunitu lidí, kteří se aktivně zajímají o tuto oblast.

2.2. Měření experimentů

Stěžejní vlastností Portálu je ukládání experimentů z oblasti měření evokovaného potenciálu. Měření se skládá jednak z binárních dat, které reprezentují konkrétní naměřená napětí na jednotlivých elektrodách v čase, a pak z metadat, což jsou textové údaje, které detailně popisují experiment. Jde například o popis testovaného subjektu (věk, váhu, výšku, jeho nemoci), prostředí (teplota, počasí, denní doba) nebo výčet použitých pomůcek (model měřící čepičky, popis zapojených elektrod). Na následujícím obrázku je vidět, jak probíhá takové měření ERP na katedře.

Obrázek 1: Ukázka měření ERP

2.3. Architektura

Jedná se o webovou aplikaci, která napsaná v jazyce Java a používající vlastnosti platformy Java EE (Java Enterprise Edition). Je to standardní aplikace, která používá třívrstvou architekturu, kde je oddělená datová vrstva, aplikační a prezentační, přičemž každá z těchto vrstev se skládá ještě z několika dalších.

Vzhledem k tomu, že Portál je klasická enterprise aplikace, používá osvědčené frameworky a postupy, jak psát rozsáhlé webové projekty v Javě. Proto je použit framework Spring, který se primárně stará o dependency injection a správné vytváření a propojení jednotlivých komponent aplikace. Jako persistentní úložiště dat je použita databáze Oracle 11g a pro komunikaci mezi aplikací a databází je nasazen framework Hibernate. Ten zajišťuje tzv. objektově-relační mapování (ORM) mezi databází a objekty, se kterými umí aplikace pracovat.

Nedílnou součástí každé aplikace je také prezentační vrstva. Zde je použit framework Wicket, resp. v době psaní této práce se na něj zrovna přecházelo.

2.4. Datová vrstva

Z pohledu této práce je zajímavé hlavně to, jak funguje datová vrstva. Proto je v této kapitole podrobněji rozebrána. Nad těmito vrstvami potom stojí controllery (aktuálně se používá Spring MVC) a dále nad nimi již zmíněný Wicket, přes který se generují formuláře a celé uživatelské rozhraní.

2.4.1. POJO Entity

Třídy POJO (Plain Old Java Object) jsou obecně Java třídy, které splňují určité specifické podmínky. Jedná se např. o bezparametrický konstruktor nebo žádné výkonné metody (s výjimkou getterů a setterů). Zde se používají pro reprezentaci persistentních entit, které se ukládají do databáze. Hibernate se stará o mapování jejich atributů na konkrétní sloupce v tabulkách a jejich automatické načítání a zápis z/do databáze. Samy o sobě nemají žádnou funkcionalitu, pouze reprezentují data.

2.4.2. DAO

DAO, neboli Database Access Object je skupina tříd, které jsou svázané s nějakým (většinou jedním) POJO objektem. Obsahují jednak metody pro zápis, aktualizaci a mazaní záznamů v databázi (vztahující se právě k tomuto POJO), a pak metody pro jejich získávání, např. na základě id, filtrování podle hodnot atributů a podobně. DAO vrstva se stará o vytváření POJO objektů.

2.4.3. Service

DAO třídy obsahují metody pro práci s jednou entitou a provádějí z hlediska fungování celé aplikace velmi jednoduché a dílčí operace. A právě z tohoto důvodu

existuje servisní vrstva, která má k dispozici reference na několik DAO objektů a je schopná provádět komplexnější operace. Např. registrace uživatele znamená z pohledu celé aplikace tyto následující kroky:

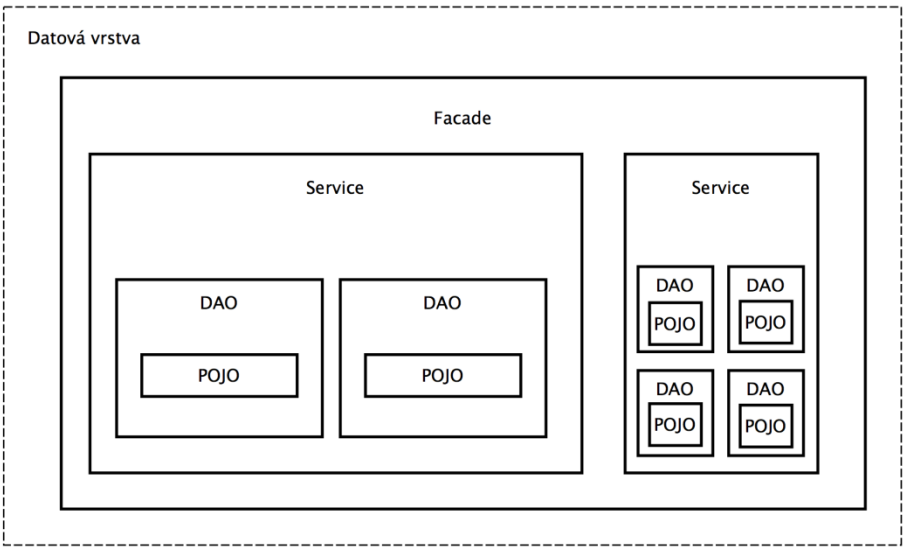
- zašifrovat jeho heslo (bezpečnost)
- uložit objekt uživatele do databáze
- přidělit mu roli běžného uživatele
- pokud byla součástí registrace pozvánka do nějaké výzkumné skupiny, připojit ho do ní
- odeslat potvrzovací email

V neposlední řadě tato vrstva obstarává transakce. To znamená, že databázové transakce jsou drženy na úrovni nějakého service objektu. Celá operace prováděná nad service se potom jeví navenek jako atomická operace.

2.4.4. Facade

Facade plní podobnou úlohu jako vrstva service. Je to další úroveň abstrakce, která může provádět velmi rozsáhlé operace s daty. Přitom ke své funkčnosti může využívat jednu a více service objektů. V tuto chvíli není tato vrstva tolik využívaná a víceméně jenom předává řízení podřízené service.

Kompletní přehled členění datové vrstvy ilustruje následující obrázek:



Obrázek 2: Architektura datové vrstvy

2.5. Důležité entity a vazby

2.5.1. Experiment

Jedná se o nejrozsáhlejší entitu, co do počtu vazeb na ostatní objekty a také co do důležitosti významu. Reprezentuje skutečné experimenty naměřené v laboratoři. Obsahuje metainformace o provedeném experimentu (ať už jednoduché datové typy nebo reference na ostatní entity), jako například:

- startTime, endTime, subjectPerson, researchGroup, ownerPerson, dataFile, scenario, electrodeConf

Dále experiment obsahuje množství velmi podobných parametrů, které jsou všechny vedeny jako samostatné persistentní entity, přičemž jsou strukturou úplně stejné. Jedná se o tyto entity:

- digitization, weather, temperature, hardware, software, disease, projectType, pharmaceutical, experimentOptParam

Jejich struktura je následující:

název	datový typ	význam
id	int	primární klíč
title	String	hodnota parametru
type	String	volitelný typ, prakticky nepoužívaný
description	String	delší textový popis parametru
isDefault	boolean	příznak, jestli je tento parametr defaultně přítomen u nového experimentu
experiments	Set<Experiment>	reference na experimenty obsahující tento parametr

Tabulka 1: Shodné položky parametrů experimentu

Je zřejmé, že všechny výše uvedené entity by šly spojit do jedné obecnější, která by byla doplněna o vlasnost parameterType, který by rozlišoval, o jaký parametr se jedná. Významně by se tím zjednodušila databáze, odpadlo by několik vazebních tabulek a značně by se zjednodušilo přidávání nových typů parametrů - nyní se musí založit nová persistentní entita, vygenerovat tabulka(y) v databázi, vytvořit DAO, upravit několik service a facade objektů a na závěr rozšířit GUI[5], aby tuto entitu umělo vytvářet. Místo toho by se pouze rozšířil navrhovaný enum parameterType o jednu hodnotu, a ta by se začala používat.

[5] GUI: Graphical User Interface. Grafické rozhraní, přes které koncový uživatel aplikaci ovládá.

2.5.2. DataFile

Entita propojená s experimentem. Mimo referenci na experiment, ke kterému patří, krátký popis, název souboru a mimetype obsahuje hlavně binární data získaná z měřicího přístroje. Vlastní obsah binárních dat není pro tuto práci zajímavý, ale potenciální velikost této entity (může šplhat až k jednotkám GB) ano.

2.5.3. Scenario a ScenarioSchema

Každý experiment se v laboratoři provádí podle určitého plánu. Takový plán zahrnuje například délku trvání, co se testovanému subjektu promítá za obraz, jaká hudba hraje na pozadí, atd. Součástí tohoto schématu jsou opět binární data (případně data ve formátu XML). S formátem a jeho způsoby ukládání se dříve experimentovalo, zkoušely se různé Oracle-proprietární databázové XML typy, datové typy blob a podobně. Jedním z cílů této práce je pokus o sjednocení a zjednodušení použití těchto schémat.

2.6. Mapování datové vrstvy

Jak již bylo zmíněno, portál používá ORM pro propojení databázových tabulek s POJO objekty, se kterými pak umí pracovat samotná aplikace. Hibernate nabízí dvojí způsob, jak toto mapování realizovat [2]:

Externí konfigurační soubory *.hbm.xml

Jedná se o XML soubory, kde každá POJO entita má definováno pomocí XML tagů, jak se která property mapuje do konkrétní tabulky a sloupce. Jde o původní (nyní už zastaralý způsob), jak toto mapování realizovat. Problém je jednak v celkem složité struktuře XML souboru a pak v nutnosti udržovat související informace na dvou oddělených místech. Příklad takového mapování vypadá následovně:

```
<hibernate-mapping>
  <class name="cz.zcu.kiv.eegdatabase.data.pojo.Experiment" table="EXPERIMENT">
    <id name="experimentId" type="int">
      <column name="EXPERIMENT_ID" precision="22" scale="0"/>
      <generator class="increment"/>
    </id>
    <many-to-one class="cz.zcu.kiv.eegdatabase.data.pojo.Weather" fetch="select"
lazy="false" name="weather">
      <column name="WEATHER_ID" not-null="true" precision="22" scale="0"/>
    </many-to-one>
    <set inverse="true" name="artifactRemoveMethods"
table="ARTEFACT_REMOVING_METHODS_REL">
      <key>
        <column name="EXPERIMENT_ID" not-null="true" precision="22" scale="0"/>
      </key>
```

```
      <many-to-many entity-
name="cz.zcu.kiv.eegdatabase.data.pojo.ArtifactRemoveMethod">
        <column name="ARTEFACT_REMOVING_METHOD_ID" not-null="true" precision="22"
scale="0"/>
      </many-to-many>
    </set>
  </class>
</hibernate-mapping>
```

Java anotace, které jsou součástí POJO entit

Toto je modernější a čitelnější způsob zápisu mapování. Přímo u každé proměnné v kódu POJO třídy je anotacemi nadefinováno mapování. Zápis takového kódu je výrazně kratší a související informace jsou pohromadě. Mapování pak vypadá následovně:

```
@Entity
@Table(name = "EXPERIMENT")
public class Experiment implements Serializable {
  @GenericGenerator(name = "generator", strategy = "increment")
  @Id
  @GeneratedValue(generator = "generator")
  @Column(name = "EXPERIMENT_ID", nullable = false, precision = 22)
  private int experimentId;
  @ManyToOne(fetch = FetchType.EAGER)
  @JoinColumn(name = "WEATHER_ID", nullable = false)
  private Weather weather;
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "RESEARCH_GROUP_ID", nullable = false)
  private ResearchGroup researchGroup;
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "ELECTRODE_CONF_ID", nullable = false)
  private ElectrodeConf electrodeConf;
  @Column(name = "START_TIME", length = 7)
  private Timestamp startTime;
}
```

Problém portálu z pohledu tohoto mapování je to, že se v konfiguraci projektu míchají oba přístupy dohromady a navíc některé třídy mají jak anotační mapování, tak k nim existuje i hbm.xml soubor. Je těžko dohledatelné, co je aktuální, co (ne)platí a který přístup vlastně Hibernate načte a použije. Poslední problém tohoto hybridního přístupu je ten, že se musí po jedné entitě vyjmenovávat (ve speciálním konfiguračním souboru), jak se která třída mapuje a uvést odkaz na tento mapovací zdroj.

3. Nutné změny a revize funkčnosti portálu

Protože se projekt postupem času vyvíjí a dynamicky reaguje na požadavky uživatelů, a zároveň se pracovníci katedry snaží formovat určité vize a plány, kam by se portál měl, co se funkčností týče ubírat, dochází často k zásahům do architektury aplikace a rozsáhlým změnám v kódu. Navíc se na projektu podílí střídající se týmy studentů, kteří plní zadané úkoly a implementují rozličné funkčnosti jako semestrální práci svého předmětu. Tyto zmíněné faktory se nezřídka negativně podepisují na kvalitě výsledného kódu. Proto je součástí této práce identifikace největších nedostatků, a pokus o jejich nápravu, jejich odstranění a refaktoring tak, aby kód byl dále lehce udržitelný a čitelný.

3.1. Přesun části dat do NoSQL

Entity a vazby kolem parametrů experimentu jsou zbytečně složité, opakuje se velké množství kódu a přidávání dalších typů parametrů je složité. Navíc by vlastníci portálu nad těmito atributy chtěli fulltextově vyhledávat, včetně pokročilých technik analýzy textu jako je stemming, rozpoznávání synonym apod. Moderní nerelační databáze toto umožňují v kombinaci s dynamickou strukturou ukládaných dat, takže se jeví jako ideální úložiště pro tuto oblast dat.

3.2. Nahrazení databáze Oracle

Jak již bylo popsáno v předchozích kapitolách, katedra má v plánu do budoucna portál zpřístupnit i komerčním firmám, které by za přístup k experimentům platili. Bohužel stávající licence databázového systému Oracle 11g neumožňuje takové použití a bylo by potřeba zakoupit nějakou z komerčních licencí. Ceny za tyto licence se pohybují v řádech jednotek až desítek tisíc dolarů [3], což je v tuto chvíli pro univerzitu neakceptovatelné. Je tedy třeba nalézt jinou vhodnou SQL databázi, kde budu uložena a kam se přesunou všechna stávající data.

3.3. Úprava konfigurace Hibernate

V minulé kapitole byly popsány nelogičnosti při konfiguraci mapování entit do databáze. Je více než vhodné toto mapování sjednotit. Navíc nové verze Springu umožňují specifikovat pouze cestu ke všem entitám a on si je automaticky načte a zpracuje. Nemusí se tedy vyjmenovávat všechny entity zvlášť.

Dalším problémem s konfigurací celého portálu je existence asi šesti konfiguračních souborů, kde se některé informace (přihlašovací údaje k databázi, konfigurace Java Beans, nastavení verbozity logování a další) opakují a není vůbec jasné, které properties se odkud načítají.

4. NoSQL databáze

Portál obsahuje data (převážně kolem experimentů), která svojí podstatou se nehodí do schématu standardních relačních databází. Jednak se vyvíjí jejich struktura, přibývají další typy parametrů, případně různé položky jsou u různých parametrů povinné/volitelné. Navíc by bylo vítané mít nad těmito daty možnost fulltextově vyhledávat, čímž se nemyslí pouhé SQL …WHERE value LIKE '%val%'…, ale použití technik jako je stemming (nalezení základního tvaru slova), rozpoznávání synonym, posuzování relevance (např. kde a kolikrát se hledané slovo v textu vyskytuje) atd.

Implementace výše zmíněných postupů jsou obsaženy i ve standardních relačních databázích, ale je často problematické nastavení takových dotazů, a záznamy obecně vyžadují specifický postup při ukládání (indexaci). V neposlední řadě je škálování těchto databází nesnadné při rostoucí velikosti indexovaných dat.

Termín NoSQL byl poprvé použit v roce 1998 a měl zdůraznit rozdíl oproti tradičním databázím v tom, že k přístupu k datům se nepoužívá zaběhnutý jazyk SQL, ale nějaký jiný, specifický pro danou databázi. V dalších letech se potom začíná vyvíjet mnoho databází tohoto typu.

4.1. SQL vs. NoSQL

Tradiční relační (SQL) databáze se používají již od osmdesátých let. Základním prvkem těchto databází jsou relace - databázové tabulky. Tabulka se skládá ze záznamů (řádky), které mají jasně dané atributy (sloupce). Atributy jsou pak definované konkrétním datovým typem a rozsahem hodnot, kterých může atribut nabývat. Každý záznam potom obsahuje primární klíč - jednoznačný identifikátor každého řádku v tabulce. Může se jednat o jeden nebo kombinaci více sloupců, které dohromady tvoří unikátní kombinaci. Další podstatnou vlastností jsou cizí klíče. Ty slouží právě pro vyjádření oněch vztahů/vazeb (relací), které mezi sebou jednotlivé záznamy z různých tabulek mají. Jde opět o jeden nebo více sloupců, které jednoznačně identifikují záznam v jiné tabulce. V neposlední řadě je významným prvkem relačních databází integrita dat.

V posledních letech se začínají hojně využívat databázové systémy, které mají poměrně odlišné vlastnosti a s tím související i případy užití. Řeč je o tzv. NoSQL

(nerelačních) databázích. Jejich označení plyne z toho, aby byl vyzdvižen jejich zásadní rozdíl oproti klasickým SQL databázím.

Důležitá je vysoká dostupnost dat a snaha o co největší propustnost, ať už při čtení nebo při zápisu. Aby tyto požadavky bylo možné splnit, většinou se rezignuje na některé principy známé z RDBMS[6]. Typicky je to pravidlo 3NF[7], data nejsou atomická, neexistují transakce a není zaručena konzistence databáze v každém okamžiku. V některých případech toto nemusí být problém: Např. nevadí, že vědecký článek po vložení do databáze nebude dostupný pro fulltext vyhledávání okamžitě, ale až za jednu nebo dvě sekundy a navíc si s sebou nese kompletní údaje o svém autorovi, takže při jeho zobrazení odpadá nutnost JOIN spojení několika tabulek.

Vlastnost	NoSQL	SQL
Styl ukládání dat	Dokumenty, dvojice klíč-hodnota, grafové struktury…	Tabulky.
Organizace dat	Dynamické schéma nestrukturovaných dat.	Předem dané, jasně definované schéma.
Škálování (zvyšování výkonu)	Horizontální – vyššího výkonu se dosahuje přidáním více běžících serverů.	Vertikální – zvyšuje se výkon běžícího serveru (více RAM, silnější procesor, SSD disky…).
Zpracovávaná data	Vhodné pro hierarchická data s různě zanořenými položkami na způsob JSON nebo XML dokumentů	Komplexní data s množstvím složitých vazeb mezi jednotlivými tabulkami. Méně vhodné pro hierarchicky vrstvená data.
Dotazovací jazyk	Každá NoSQL databáze používá vlastní dotazovací jazyk. Většinou pro získávání jasně daných dat z databáze podle jednodušších kritériích.	Používá se standardizovaný SQL pro pokládání složitých dotazů. Vhodný k tvorbě různých komplexních projekcí nad zdrojovými daty.
Provázanost dat	Protože neexistuje efektivní možnost spojování souvisejících entit (dokumentů), související data se často kopírují jako vnořené sub-dokumenty v rámci ukládané entity.	Relace se definují cizími klíči – záznam obsahuje referenci na související entitu v jiné tabulce.
Bezpečnost dat	V (distribuovaných) NoSQL databázích nelze zajistit požadavky ACID známé z relačních databází.	Podporují tzv. ACID přístup – Atomicita operací (uzavřených v transakci), Konzistence – k každý okamžik jsou data v bezchybném stavu, Izolovanost – jak (ne)jsou transakce viděny okolnímu prostředí, Trvanlivost – jakmile jsou data zapsána, už se bez další operace sama „neztratí“.

[6] RDBMS: Relational database management system – relační databáze.

[7] 3NF: Třetí normální forma: soubor pravidel a doporučení pro návrh databázových struktur, aby byly optimálně využity vlastnosti relačních databází.

Tabulka 2: Srovnání SQL a NoSQL databází

4.2. Využití NoSQL

Z výše popsaného srovnání vlastností SQL a NoSQL úložišť vyplývá i způsob využití a hlavní případy užití. NoSQL se hodí:

- Na ukládání velkého množství dat, ke kterému je třeba distribuovaný přístup z různých částí světa (horizontální škálování).
- V aplikacích, kde je jednodušší datový model (není třeba tolik JOIN operací).
- V aplikacích, kde je důraz na rychlý vývoj.
- Pro jednoduchou a automatickou redundanci dat (zotavení z pádu určitého množství uzlů).
- Pro vytvoření rychlé cachovací vrstvy.
- Kdy nepatrná latence v dostupnosti nově uložených dat nepředstavuje problém.
- Jako flexibilní úložiště dat (nestrukturovaná/semistrukturovaná data).

Naopak ne úplně vhodné nebo dokonce špatné využití NoSQL čítá tyto případy:

- Je klíčová atomicita a konzistence ukládaných dat (např. seznam online plateb zákazníků).
- Ukládaná data mají typicky relační charakter.
- Aplikace, které vyžadují transakční zpracování.
- Je nezbytná složitá analýza, agregace a další zpracování uložených dat.
- Je vyžadován propracovanější systém oprávnění a přístupu na úrovni databázových uživatelských účtů.
- Požaduje se stoprocentní odladěnost a otestovanost použité databáze.

4.3. Rozdělení NoSQL databází

Základní klasifikaci NoSQL databází lze provést na základě typu ukládaných dat:

- Dokumentové

Základním konceptem dokumentových databází je pojem dokumentu. Ten obsahuje částečně strukturovaná data, kde každý dokument může obsahovat libovolné položky (klíče). Nad těmito klíči se pak dají vytvářet indexy a lze podle nich dokumenty vyhledávat. Pro kódování dokumentů se používá spousta formátů, např. JSON, XML, YAML, BSON (= binární JSON). Mezi typické představitele těchto databází patří: MongoDB, Elasticsearch, Neo4J, CouchDB, Solr a další.

- Klíč-hodnota

Jde o často velmi jednoúčelové databáze, které jsou pro daný úkol vysoce optimalizované. Data lze získávat jen pomocí primárního klíče. Často se snaží veškerá data držet v RAM, čímž dosahují velmi vysokých rychlostí zápisu a čtení. Zástupci této technologie jsou např.: Memcached, Redis.

- Grafové

Tyto databáze používají grafové struktury jako uzly a hrany k reprezentaci dat. Uzly jsou přímo propojeny se svými sousedními prvky, takže není třeba dělat žádné JOIN operace a prohledávání indexu. Využívají se například k reprezentaci dat na sociálních sítích, map nebo třeba síťových topologií.

Existuje samozřejmě mnoho dalších typů úložišť (objektové, sloupcově orientované, XML…), které jsou ale nad rámec této práce a proto zůstanu jen u této klasifikace.

4.4. Požadavky Portálu

Hlavní motivací, proč začít používat nerelační databázi, je mít možnost jednoduše a dynamicky přidávat další metainformace o měřených experimentech. Tím se myslí jednak úplně nové typy parametrů, ale také snadná rozšiřitelnost stávajících parametrů o dodatečné sub-parametry. O nutnosti vyhledávání nad těmito daty už bylo psáno dříve v této práci.

Použití bezschémové databáze se tedy nabízí jako ideální řešení. V další části této kapitoly se pokusím o základní srovnání několika nejznámějších NoSQL databází, které se specializují na fulltext vyhledávání. Z tohoto srovnání by mělo vyjít vhodné řešení pro ukládání metadat k experimentům. Z podstaty NoSQL databází je jasné, že se do této databáze nepřesunou všechna data (např. autor experimentu nebo výzkumná skupina, do které experiment patří), protože se jedná o data s úzkou vazbou na ostatní entity Portálu a bylo by velice komplikované udržovat tato data v obou databázích synchronizovaná a aktuální.

4.5. Použitelné databáze

4.5.1. MongoDB

Jedná se o NoSQL, dokumentově orientovanou databázi napsanou v jazyce C++ [4]. Umožňuje data dělit do databází, přičemž každá databáze se skládá z jedné nebo více kolekcí. Kolekce by měly (ale nemusí) sdružovat dokumenty se stejnou nebo podobnou strukturou. Každý dokument musí obsahovat unikátní primární klíč. Ten je možné nechat si vygenerovat buď databází při vkládání dokumentu nebo ho specifikovat ručně. Nad jednotlivými klíči v dokumentu lze následně definovat indexy, které urychlují hledání dokumentů při jejich čtení. Na druhou stranu zpomalují zápisové operace, stejně jako v klasických SQL systémech.

MongoDB podporuje tzv. ReplicaSety, což je princip master-slave replikace. Jde o to, že existuje několik instancí MongoDB na různých strojích, které jsou spojeny do jednoho ReplicaSetu. Jedna z instancí potom vystupuje jako primární uzel, který umí zpracovávat jak čtecí, tak i zápisové operace. Pokud dojde k zápisu, interně potom tuto změnu přepošle i sekundárním uzlům. Ty umí zpracovávat pouze čtecí operace a snižují tak zátěž primárního uzlu.

Jako formát pro ukládání dokumentů je použit JSON. Vlastně celá databáze má k Javascriptu poměrně blízko - řádkový mongo shell je interaktivní (REPL) klient Javascriptu. Na následujícím kódu je vidět ukázka použití Mongo konzolového klienta, připojení se do databáze, vložení tří dokumentů (u druhého je ručně specifikován primární klíč _id) a nakonec vypsání všech záznamů:

```
bydga@bydga-mac ~ $ mongo
MongoDB shell version: 2.4.9
connecting to: test
> use eeg
```

```
switched to db eeg
> db.users.insert({name:"Martin",surname:"Bydžovský", birth: new Date("1988-09-08")})
> db.users.insert({name: "Jan", surname: "Černý", _id: "jancerny"})
> db.users.insert({name: "Petr", surname: "Novák", friends: [{name:
"Honza"},{name: "Martin"},{name: "Jan"}]})
> db.users.find().pretty()
{
        _id : ObjectId("534afe2d94be6eee9330d001"),
        name : "Martin",
        surname : "Bydžovský",
        birthday : ISODate("1988-09-08T00:00:00Z")
}
{ _id : "jancerny", name : "Jan", surname : "Černý" }
{
        _id : ObjectId("534afe8494be6eee9330d002"),
        name : "Petr",
        surname : "Novák",
        friends : [{name : "Honza"},{name : "Martin"},{name: "Jan" }]
}
> db.users.find({_id: "jancerny"})
{ _id : "jancerny", name : "Jan", surname : "Černý" }
```

Bohužel Mongo v době psaní této práce nemá podporu pro fulltextové vyhledávání. Od verze 2.4 (duben 2013) [5] je sice přidána podpora pro Text Search, ale i po téměř roce od vydání je stále ve verzi beta a není připravena pro produkční nasazení.

4.5.2. Elasticsearch

Jde o poměrně nový projekt (únor 2010) [6], vyvíjený v jazyce Java. Elasticsearch je dokumentově orientovaná databáze, se kterou se plně komunikuje prostřednictvím jejího REST API. Jak už je patrné z názvu, jeho primární zaměření je analýza textu a fulltextové vyhledávání. Je postaven nad Lucene indexem.

Komunikační formát je také JSON. Ten se používá jak k ukládání a získávání dat (query language), tak ke konfiguraci celé databáze - počet uzlů v clusteru, způsob replikace, nastavení mapování atd. Struktura ukládaných dat je následující: Definuje se index (obdoba relační databáze nebo schématu). V každém indexu může být několik typů (obdoba tabulky), které sdružují pohromadě dokumenty se stejným nebo podobným obsahem.

Jako Mongo, tak i Elasticsearch v základu podporuje replikaci. Více instancí se spojí do tzv. clusteru a každému indexu se nadefinuje v kolika replikách (na kolika instancích) má existovat. Elasticsearch se poté automaticky stará o rovnoměrnou distribuci dat mezi jednotlivými uzly.

Na následujícím obrázku je ukázka práce s Elasticsearchem z příkazové řádky - vytvoření nového indexu, zaindexování dvou dokumentů (typu people) a pak vyhledání konkrétní osoby podle _id a nakonec vyhledání všech osob:

```
bydga@bydga-mac ~ $ curl -XPUT http://eeg2.kiv.zcu.cz:9200/testindex -d '{ "settings"
: {"index" : {"number_of_shards" : 5,"number_of_replicas" : 0 }}}'
{"acknowledged":true}
bydga@bydga-mac ~ $ curl -XPUT http://eeg2.kiv.zcu.cz:9200/testindex/people/bydga -d
'{ "name" : "martin", "birth":"1988-08-09T00:00:00", "surname" : "bydzovsky" }'
{"_index":"testindex","_type":"people","_id":"bydga","_version":1,"created":true}
bydga@bydga-mac ~ $ curl -XPOST http://eeg2.kiv.zcu.cz:9200/testindex/people -d '{
"name": "Jan", "birth":"1992-11-03;2DT00:00:00", "surname" : "Černý", "friends":
[{"name": "Honza"},{"name" : "Martin"},{"name": "Jan" }] }'
{"_index":"testindex","_type":"people","_id":"LqVERoGeQd2FwR7sd0acVA","_version":1,"c
reated":true}
bydga@bydga-mac ~ $ curl -XGET
http://eeg2.kiv.zcu.cz:9200/testindex/people/bydga?pretty=true
{
  "_index": "testindex",
  "_type": "people",
  "_id": "bydga",
  "_version" : 1,
  "found": true, "_source": { "name": "martin", "birth":"1988-08-09T00:00:00",
"surname": "bydzovsky" }
}
bydga@bydga-mac ~ $ curl -XGET http://eeg2.kiv.zcu.cz:9200/testindex/people/_search
{
  "hits" : [ {
    "_index": "testindex",
    "_type": "people",
    "_id": "bydga",
    "_source":
      { "name": "martin", "birth":"1988-08-09T00:00:00", "surname": "bydzovsky" }
    }, {
      "_index": "testindex",
      "_type": "people",
      "_id": "LqVERoGeQd2FwR7sd0acVA",
      "_source": { "name" : "Jan", "birth":"1992-11-03;2DT00:00:00", "surname" :
"Černý", "friends": [{"name" : "Honza"},{"name" : "Martin"},{"name": "Jan" }] }
    } ]
  }
}
```

Je vidět, že veškerá komunikace probíhá přes HTTP protokol - jak vytvoření indexu, uložení dokumentů, tak i jejich získání.

4.5.3. Solr

Solr je technologicky podobný projekt jako Elasticsearch. Jedná se o starší projekt - vyvíjený je od roku 2004 [7], také napsaný v jazyce Java jako dokumentové úložiště. Solr stejně tak jako Elasticsearch používá fulltextové jádro jádro Lucene. Pro komunikaci se Solrem se používá REST API. Solr podporuje ukládání dokumentů ve formátu JSON, XML, CSV.

Podpora pro nějakou formu replikace byla přidána až později, ve verzi Solr 4 (říjen 2012) [8]. Do té doby se Solr profiloval jako jednoinstancový. I v rámci jedné instance bylo potřeba oddělovat navzájem nesouvisející data (dokumenty). Pro tuto situaci má Solr pojem jádro (Core). Každé jádro má vlastní konfiguraci, mapování dokumentů a funguje jako oddělený celek. Do tohoto jádra se potom nahrávají jednotlivé dokumenty. Jemnější granularitu (jako např. Elasticsearch definuje index-typ-dokument) Solr nemá.

4.6. Výběr NoSQL databáze

Protože jedním ze základních požadavků bylo fulltextové vyhledávání, databáze MongoDB byla vyřazena rovnou, protože nic, co by se alespoň přibližovalo tomuto požadavku, nenabízí. Oproti tomu jak Elasticsearch, tak Solr, obě databáze poskytují fulltextové vyhledávání se všemi aspekty, co k tomu patří. V této kapitole bude provedeno podrobné srovnání vlastností i výkonu obou databází. Na základě výsledků, které zahrnují i subjektivní zhodnocení pohodlnosti při práci s oběma systémy, bude zvolena vhodnější databáze.

4.6.1. Lucene a invertovaný index

Již několikrát bylo v textu zmíněno, že jak Elasticsearch, tak Solr pracují s Lucene indexem. Lucene je open source projekt napsaný v jazyce Java a spadající pod Apache Software Foundation. Jde o engine, který slouží pro indexaci a vyhledávání textových dat. K organizaci dat používá datovou strukturu označovanou jako invertovaný index. Lucene samotný obsahuje pouze nízkoúrovňové příkazy pro ukládání a získávání dat. K tomu poskytuje komplexní a složité api. Dále neřeší škálování, distribuovaný přístup a další užitečné vlastnosti pro budování komplexních aplikací.

Invertovaný index je struktura vhodná právě na fulltextové vyhledávání. Místo, aby se vzal vstupní text a uložilo se, která slova obsahuje, index se sestavuje opačně. Projde se vstupní text a všechna slova se přidají do indexu (pokud už tam nejsou) s informací, že jsou obsaženy (také) v právě indexovaném dokumentu. Například pro následující dokumenty:

```
DOC1: This is a test.
DOC2: Test example that is funny.
DOC3: It is weird
```

Bude sestaven takovýto invertovaný index:

```
this    => 1
is      => 1,2,3
a       => 1
test    => 1,2
example => 2
that    => 2
funny   => 2
it      => 3
weird   => 3
```

Při vyhledávání fráze „test is“ poté bude výsledkem:

```
{1,2} && {1,2,3} = {1,2}
```

Čili nalezené jsou dokumenty DOC1 a DOC2.

Do takovéhoto indexu lze dále ukládat informace o pozici slova v daném dokumentu a další metainformace. Dalším důležitým aspektem je ukládání slov ne přímo tak, jak jsou v textu, ale v lemmatizovaném[8] tvaru. Tím se jednak sníží velikost indexu a navíc pro potřeby fulltextového vyhledávání se sloučí slova se stejným významem (množná čísla, přivlastňovací tvary, minulý čas, atd.).

4.6.2. Teoretické srovnání

Jde o dva na první pohled velmi podobné systémy. Oba jsou napsány v jazyce Java, oba jsou nadstavbou nad Lucene a oba se profilují jako datové úložiště s primárním zaměřením na fulltext vyhledávání.

Vyzrálost

Elasticsearch je mnohem novější a tedy i modernější systém, který lépe reflektuje současné požadavky. Od začátku je programován s důrazem na škálovatelnost (nasazení na více serverech). Solr je oproti tomu starší projekt, což může znamenat výhodu ve smyslu stabilnějšího kódu, daleko hlubší otestovanosti celého projektu a celkově vyzrálejším ekosystému. Např. stabilní verze Elasticsearche (1.0.0) byla oficiálně vydána až v průběhu psaní této práce. Do té doby existovala pouze vývojová verze 0.90. Stáří projektu má také zásadní vliv na velikost uživatelské základny a aktivních členů, kteří se podílejí, ať už přímo na vývoji databáze, nebo na jejím používání - což je zde plus pro Solr.

Konfigurace

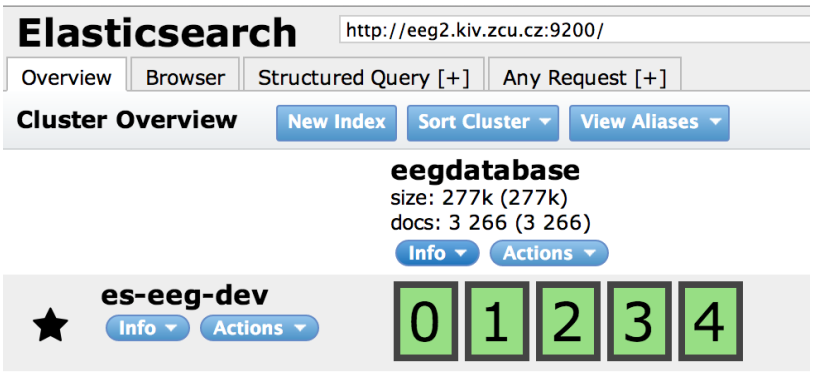
[8] Lematizace: Nalezení základního tvaru slova.

Solr má ve srovnání s Elasticsearchem složitější konfiguraci, která je navíc roztříštěna do několika souborů ve formátu XML. Zvlášť se konfiguruje celý server a zvlášť potom každé jádro (databáze). Problém těchto souborů je hlavně ten, že XML je velmi komplexní formát (něco se zapisuje jako XML vnořený element, něco jako atribut elementu) a je nutné stále kontrolovat dokumentaci, jak se která vlastnost zapisuje. Elasticsearch má jen jeden konfigurační soubor ve formátu YAML, který se navíc drží ploché struktury, takže je velice jednoduché s takovým souborem pracovat. Konfigurace jednotlivých indexů (databází) se poté provádí přes REST API, takže odpadá nutnost dalších souborů. S tím souvisí i další výhoda Elasticsearche. Většina změn v konfiguraci indexu se dá dělat za běhu - zavoláním specifického endpointu[9]. Pro změnu takové konfigurace není třeba pak restartovat server, což u Solru možné není. Tam i změna (přidání) nového mapování restart vyžaduje.

Sledování stavu

Elasticsearch v základu nedisponuje žádným přehledným monitoringem stavu (pouze JSON výstupem z REST endpointu status). Ale existují dva oficiálně podporované pluginy: HEAD a Paramedic.

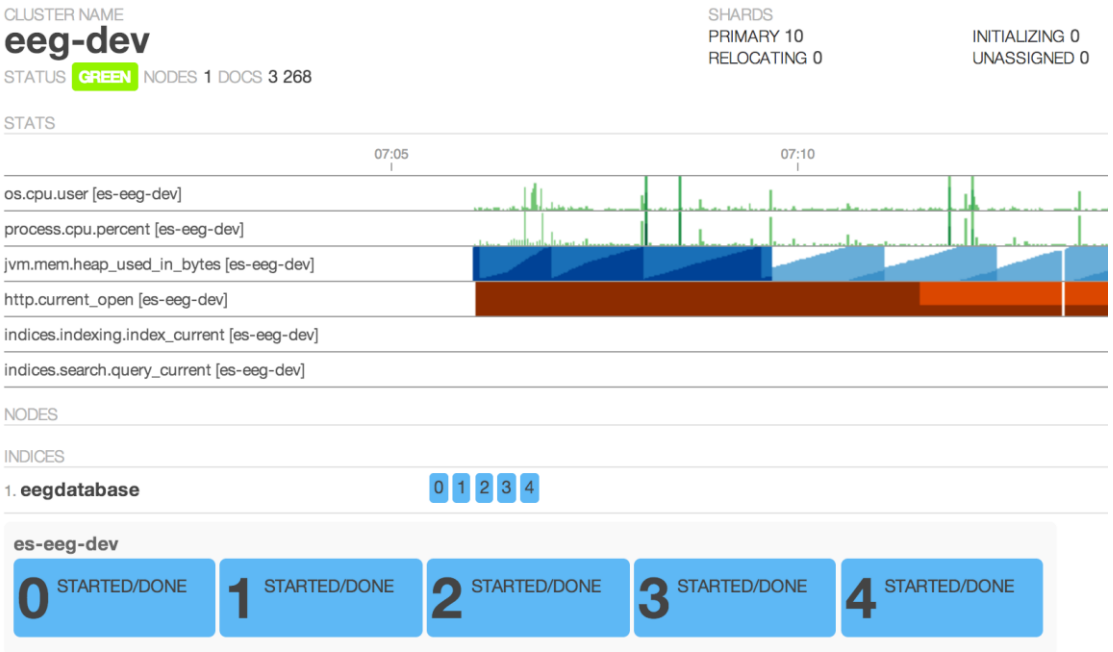
Head je plugin spíše pro běžnou administraci databáze, kontrolu základního stavu, prohlížení uložených dat a testování dotazů skrz zabudovaný REST klient. Ten si navíc pamatuje všechny poslané příkazy, takže je možné kdykoliv vyvolat již dříve položený dotaz.



Obrázek 3: Ukázka pluginu pro správu databáze

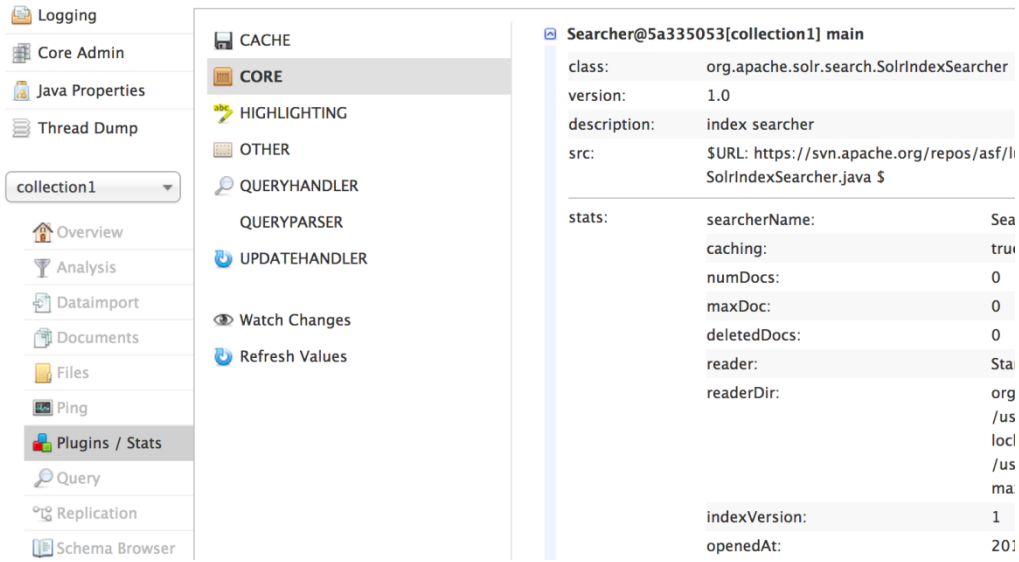
[9] Endpoint: Označení používané v REST službách pro URL adresu, kam se posílají požadavky na jednotlivé úkony.

Paramedic potom umí zobrazovat přehledně v grafech různé realtime statistiky serveru jako počet dotazů, využití paměti RAM, CPU, stav jednotlivých indexů atd.



Obrázek 4: Elasticsearch monitoring

Dle dokumentace a popisů na internetu by Solr měl poskytovat monitorovací rozhraní přes Java Management Konzolu, ale bohužel se mi toto řešení nepodařilo zprovoznit. Nějaké základní statistiky sice poskytuje webové rozhraní Solr Admin, ale v porovnání s monitorovacími nástroji Elasticsearche Solr značně zaostává.



Obrázek 5: Solr monitoring

Zanořené objekty

Jedním z velkých nedostatků Solru je nemožnost indexovat zanořené (vnořené) dokumenty. Příklad takového dokumentu může být třeba:

```
{
  _id: "76102734",
  title: "Some beautiful heading",
  author_info: {
    name: "John",
    surname: "Doe",
    birth: "1980-01-01"
  }
}
```

A právě položka author_info je vnořený dokument. Existuje sice možnost, jak tuto situaci řešit, a sice převést vnořený dokument na plochou strukturu, např. takto:

```
{
  _id: "76102734",
  title: "Some beautiful heading",
  author_info_name: "John",
  author_info_surname: "Doe",
  author_info_birth: "1980-01-01"
  }
}
```

S takovýmto dokumentem by již Solr uměl pracovat. Bohužel v momentě, kdy by (v tomto konkrétním případě) bylo např. autorů více, již by takováto struktura byla těžko realizovatelná. Musely by se začít jednotlivé fieldy číslovat: author_info_name1, author_info_name2,…, což vede k nemožnosti poté podle takovéto položky vyhledávat - těžko se položí dotaz ve smyslu: WHERE author_info_name1="john" OR author_info_name2 = "john". Solr má možnost označit field jako multiValued, který se ale hodí pro jiné účely - třeba pro jednoduchý seznam klíčových slov. Pro složitější informace vhodný není. Elasticsearch oproti tomu s vnořenými dokumenty nemá nejmenší problém.

Split-brain problém

Tento jev může nastat v případě, kdy databáze (ať už Elasticsearch nebo Solr) běží distribuovaně ve více instancích. Jeden uzel vždy vystupuje jako master a ten jediný může přijímat požadavky na zápis nových dokumentů. Ostatní uzly (slave) si poté interně replikují zapsaná data a obsluhují požadavky na čtení dat. V momentě, kdy se z jakéhokoliv důvodu rozpadne síť mezi těmito uzly, slave uzly usoudí, že se něco stalo s master uzlem a převezmou jeho roli. Tím dojde k fatální nekonzistenci dat, kdy existují dva master uzly a každý přijímá část požadavků na zápis. Navíc se aplikace může tvářit jako funkční, čímž se způsobí ještě hlubší problémy.

Solr toto řeší zavedením jednoho rozhodčího uzlu - ZooKeeper [9], který existuje jen v jedné instanci a má na starosti určování master uzlů. S výpadkem nějakého uzlu je tedy schopen se vypořádat přímo ZooKeeper a při výpadku jeho samotného se poté celá aplikace stane nefunkční, čímž zamezí vzniku nekonzistentních dat.

Elasticsearch sice ve výchozí konfiguraci není odolný proti split-brain problému, ale vše stačí vyřešit úpravou jednoho řádku v konfiguraci. Jde o položku discovery.zen.minimum_master_nodes a její hodnota říká, kolik uzlů mezi sebou musí být minimálně viditelných, aby si mohli zvolit svého mastera. Nastavení této direktivy na hodnotu N/2+1, kde N je celkový počet uzlů v clusteru, poté zajistí, aby k split-brain situaci vůbec nemohlo dojít.

4.6.3. Srovnání výkonu

Pro účely srovnání výkonu obou databází byl sestaven následující test: Z volně dostupných zdrojů [10] bylo staženo množství anglicky psaných knih v textovém formátu. Celkem se jedná o 25MB čistého textu. Knihy byly spojeny do jednoho souboru a rozděleny po odstavcích. Vzniklo tak necelých 500 000 odstavců s průměrnou délkou 150slov/odstavec. Každý odstavec byl potom použit při vkládání do databáze jako jeden indexovaný dokument. Následně se pro každou databázi sestavilo několik různých scénářů pro vkládání dokumentů, jejich čtení a také vzájemné paralelní provádění těchto operací s různými intenzitami. Měřenou veličinou byl buď počet dokumentů, kolik je schopna databáze přečíst/zaindexovat za určitý čas nebo v případě vkládání záznamů čas, za který se podařilo určitý počet dokumentů vložit. Každý scénář byl opakován vícekrát pro přesnější výsledky.

Referenční stroj, na kterém byly testy spouštěny měl následující konfiguraci:
Macbook Pro Retina (Early 2013)
Intel Core i5, 2,6GHz
8GB RAM
256GB SSD disk

Softwarové vybavení:
OSX 10.9 Mavericks
Java: 1.6.0_65

Solr: 4.5.1

Elasticsearch 0.90.5

Scénář 1

Čas vložení jednoho milionu dokumentů do prázdné databáze, žádné čtení.

Pokus/Databáze	Elasticsearch [ms]	Solr [ms]
1.	322 693	313 453
2.	305 222	298 107
3.	315 745	315 742

Tabulka 3: Výkonnostní srovnání 1

Scénář 2

V databázi je milion dokumentů, čtení v jednom vlákně po dobu 30 sekund. Měří se počet přečtených dokumentů.

Pokus/Databáze	Elasticsearch [počet]	Solr [počet]
1.	3 827	11 038
2.	4 150	10 970
3.	4 145	11 172

Tabulka 4: Výkonnostní srovnání 2

Scénář 3

V databázi je milion dokumentů, 3 vlákna paralelně z databáze čtou po dobu 20 sekund. Měří se počet přečtených dokumentů.

Pokus/Databáze	Elasticsearch [počet]	Solr [počet]
1.	3x1 320	3x10 400
2.	3x1 610	3x9 980
3.	3x1 480	3x10 950

Tabulka 5: Výkonnostní srovnání 3

Scénář 4

V databázi je milion dokumentů, doba běhu testu je 20 sekund, tři vlákna čtou, jedno vlákno zapisuje rychlostí 2 dokumenty/sekundu.

Pokus/Databáze		Elasticsearch [počet]	Solr [počet]
1.	**zapsáno**	40	40
	přečteno	3x1 200	3x6 500
2.	**zapsáno**	40	38
	přečteno	3x1 190	3x6520
3.	**zapsáno**	40	39
	přečteno	3x1220	3x6490

Tabulka 6: Výkonnostní srovnání 4

Scénář 5

V databázi je milion dokumentů, doba běhu testu je 20 sekund, tři vlákna čtou, jedno vlákno zapisuje rychlostí 20 dokumentů/sekundu.

Pokus/Databáze		Elasticsearch [počet]	Solr [počet]
1.	zapsáno	386	262
	přečteno	3x1 150	3x2230
2.	zapsáno	369	258
	přečteno	3x1180	3x2 220
3.	zapsáno	376	261
	přečteno	3x1200	3x2230

Tabulka 7: Výkonnostní srovnání 5

Scénář 6

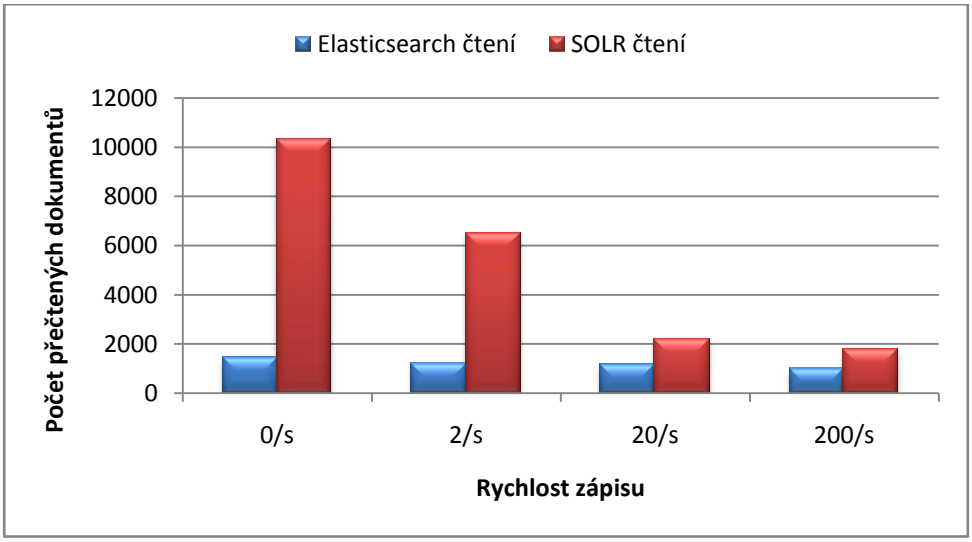
V databázi je milion dokumentů, doba běhu testu je 25 sekund, tři vlákna čtou, jedno zapisuje rychlostí 200 dokumentů/s.

Pokus/Databáze		Elasticsearch [počet]	Solr [počet]
1.	zapsáno	860	685
	přečteno	3x1 010	3x1830
2.	zapsáno	842	682
	přečteno	3x1 030	3x1820
3.	zapsáno	864	692
	přečteno	3x1 010	3x1820

Tabulka 8: Výkonnostní srovnání 6

Zhodnocení výsledků

Z měření je patrné, že indexování (vkládání) do klidné databáze (bez probíhajícího čtení) je v obou databázích prakticky totožné. Nepatrné odchylky v řádu jednotek procent jsou zanedbatelné. Oproti tomu při čtení z klidné databáze (bez simultánních zápisů) je rozdíl naprosto markantní ve prospěch Solru. Za stejný čas je Solr schopen vrátit přibližně třikrát tolik dokumentů než Elasticsearch. Zajímavé ale je, že v momentě, kdy se do databáze přidá i zápis, výkon Solru jde prudce dolů. Při rostoucí zátěži z hlediska zápisů do databáze si Elasticsearch drží takřka konstantní počty přečtených dokumentů, zatímco propustnost Solru klesne přibližně šestkrát. Na druhou stranu, i přes tento pokles pořád Solr vyhrává, byť už ne tolik, z hlediska čtení. Elasticsearch si ale ve velké zátěži vede o malinko lépe v počtu zaindexovaných dokumentů.



Obrázek 6: Srovnání výkonu Elasticsearch vs. SOLR

Z hlediska absolutních čísel si lépe stojí Solr, ale je velice znepokojující jeho klesající výkonnost při stoupající zátěži.

4.6.4. Subjektivní pohled

S oběma systémy se pracuje vcelku podobně, princip přístupu z Javy je prakticky stejný až na rozdílné názvy tříd a metod. Oba systémy se velmi liší v kvalitě a přehlednosti dokumentace. Solr používá dokumentaci na způsob wiki stránek a intuitivnost ovládání je na nízké úrovni (viz např.: https://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters). Na první pohled je samotná stránka strukturovaná do sekcí, bez rušivých elementů a v kontrastních barvách. Chybí ale jakákoliv drobečková navigace, propojení s příbuznými tématy, menu, které by pomohlo v orientaci, kde se uživatel právě nachází, dokonce i URL je nestrukturovaná a cesty všech stránek jsou ve formátu http://wiki.apache.org/solr/Clanek. To, že design stránek a nefixní šířka obsahu se špatně čte (na širokoúhlých monitorech s vysokým rozlišením - text se roztáhne na celou šířku monitoru) je už ve výčtu výše popsaných neduhů spíše drobnost. Něco v této dokumentaci hledat a rychle jí procházet je poměrně náročné. Elasticsearch oproti tomu netrpí ani jedním z výše popsaných problémů. Ale ani dokumentace Elasticsearche není bezchybná - kvůli právě aktuálnímu přechodu z vývojové verze 0.90 na finální 1.0, byla jeho dokumentace občas nejasná ve smyslu, ke které verzi se vlastně článek vztahuje.

Dále, u Solru jsou obecně složitější všechny úkony spojené s administrací a konfigurací databáze. Nastavení jsou komplikovanější a všude na očích jsou parametry,

které slouží pro jemné ladění určitých indexovacích parametrů, nastavení Lucene indexu, které ale nejsou pro počáteční zprovoznění nezbytné a je třeba je řešit až při konkrétních výkonnostních problémech.

4.6.5. Výběr databáze

Vlastnost	MongoDB	Elasticsearch	Solr
Dynamické schéma	+	+	-
Dokumentace	+++	++	+
Fulltext-search	-	+++	+++
Škálování (replikace)	+++	+++	+
Uživatelská komunita	++	++	+++
Výkon	?	++	+++
Jednoduchost konfigurace	++	+++	+
Změny konfigurace za běhu	?	++	-

Tabulka 9: Srovnání vlastností Mongo vs. Elasticsearch vs. Solr

Po zhodnocení všech výše uvedených faktorů, zvážení informací z dostupné dokumentace, výkonnostního srovnání a i subjektivního pohledu bylo nutné vybrat databázi pro implementaci fulltextového vyhledávání. Solr vychází lépe ve výkonnostním benchmarku, nejmarkantnější rozdíl byl ale v umělém testu, kdy se z databáze pouze četlo. Při přidávání zápisových operací a simultánním čtení (o čemž se dá tvrdit, že se velmi blíží skutečnému scénáři použití) se výkonnostní rozdíl začal smazávat. Zásadní problém Solru jsou problémy s vnořenými objekty, kde by se situace musela složitě řešit na aplikační úrovni. Složitosti s konfigurací databáze v kombinaci s ne úplně přívětivou dokumentací také nepřináší kladné body pro Solr. Co ale Solr má nepochybně lepší je velikost projektu a s tím související množství zainteresovaných vývojářů. Tento rozdíl by se ale mohl postupem času zmenšovat, jak kolem Elasticsearche vzniká množství zajímavých projektů (např. Logstash, Kibana, integrace s nástrojem Puppet…).

Vzhledem k všem těmto skutečnostem bylo rozhodnuto, že se jako primární NoSQL úložiště použije Elasticsearch.

5. Elasticsearch

5.1. Základní pojmy

V této kapitole bude popsána základní terminologie tak, jak se používá v projektu Elasticsearch [11].

Node (uzel)

Jde o jednu běžící instanci procesu Elasticsearch, která patří do určitého clusteru. Lze nastartovat více nodů na jednom serveru, ale obecně by měla běžet na jednom stroji jedna instance. Při startu se proces pokusí na síti vyhledat další uzly ze stejného clusteru a spojit se s nimi.

Cluster

Cluster je seskupení jednoho nebo více uzlů, které mají nadefinovaný stejný název clusteru. Každý cluster obsahuje právě jeden master uzel, který je automaticky vybrán a který může být nahrazen v případě jeho poruchy.

Index

Jedná se o ekvivalent databáze v relačních databázích. Index obsahuje mapování, které definuje různé typy. Index je rozprostřen na jeden nebo více primary shardů a dále na žádný nebo více replica shardů.

Shard

Shard je přímo jedna instance Lucene indexu, která je naprosto automaticky spravovaná Elasticsearchem. Index je logické seskupení, které se skládá z primary a replica shardů. Uživatel pak pouze definuje množství těchto shardů a dále s nimi nikdy přímo nepracuje. Stačí reference na index. Elasticsearch poté distribuuje shardy přes všechny uzly v clusteru a přesouvá je podle potřeby: při přidání nového uzlu do clusteru nebo při jeho selhání.

Primary shard

Každý dokument je při indexaci nejprve uložen do jednoho primary shardu a poté na všechny repliky tohoto shardu. Ve výchozí konfiguraci index obsahuje pět primary shardů. Tento počet lze nastavit v době vytváření indexu a nelze později nijak změnit.

Replica shard

Každý primary shard může mít žádnou nebo více replik. Replika je identická kopie dat a slouží jednak k záloze primárního shardu (může se z ní stát nový primary při výpadku původního), a pak ke zvýšení výkonu při vyhledávání.

Typ dokumentu

Jde o analogickou entitu k tabulce v databázi. Sdružuje dokumenty se stejným významem a tím pádem stejnou nebo velmi podobnou strukturou. Každý typ obsahuje mapování (mapping), které definuje, jak se bude každá položka dokumentu analyzovat a indexovat.

Mapping (mapování)

Mapování je podobné jako definice schématu tabulky v relační databázi. Říká, jaké položky (fieldy) bude dokument obsahovat a jak se budou zpracovávat při indexaci. Mapování může být explicitně definováno, nebo se vytvoří automaticky při vložení prvního dokumentu.

Dokument

Jde o JSON dokument, který je uložen do Elasticsearche, do nějakého indexu a je nějakého typu (typ = analogický pojem jako řádek v tabulce). Pokud je specifikováno ID dokumentu při jeho indexaci, je použito to, v opačném případě ho databáze vygeneruje. Kombinace index-typ-ID je unikání identifikátor každého dokumentu.

5.2. Instalace

V této kapitole bude popsána instalace aktuální verze Elasticsearch (1.0.1) na Linuxový server Debian 7 (Wheezy).

Jedna z možností by byla stáhnout přímo spustitelnou binární aplikaci a jednoduše ji spustit. Musel by se ale ručně konfigurovat init skript a celkově by systém o databázi Elasticsearch vůbec nevěděl. Následné aktualizace by také znamenaly problém. Proto nejjednodušší a také nejvhodnější je instalace jako hotový balíček z centrálního repositáře.

Protože se v oficiálním (Debian) repositáři balíčků Elasticsearch nenachází, musí se přidat další zdroj aplikací. Nejprve se přidá podpis repositáře:

```
wget -O - http://packages.elasticsearch.org/GPG-KEY-elasticsearch | apt-key add -
```

Následně samotný repositář Elasticsearch:

```
echo "deb http://packages.elasticsearch.org/elasticsearch/1.0/debian stable main" >>
/etc/apt/sources.list
```

Nyní už lze jednoduše Elasticsearch nainstalovat jako jakýkoliv jiný balíček:

```
apt-get update && apt-get install elasticsearch
```

Tento sled příkazů zajistí jednak stažení příslušného programu, vytvoření systémového uživatele elasticsearch, umístění konfiguračního souboru na správné místo a také vytvoří init skript - který slouží pro spouštění a zastavování Elasticsearche jako služby na pozadí.

5.3. Konfigurace

Po instalaci se Elasticsearch rovnou nespustí, protože je potřeba nastavit několik věcí - bylo by nemilé, kdyby se ihned po instalaci uzel spustil, připojil do nějakého clusteru a ihned začal replikovat jeho data. Hlavní (a jediný) konfigurační soubor se nachází v /etc/elasticsearch/elasticsearch.yml.

V něm je potřeba změnit následující hodnoty:

```
cluster.name: eeg-dev
```

Název clusteru, do kterého chceme tuto instanci připojit. Tím, že zatím s daty Portálu žádný Elasticsearch neběží, musí toto jméno být dostatečně unikátní, aby nehrozilo, že v rámci školní sítě už někdo Elasticsearch se stejným názvem používá, a tato instance se s ním spojí nebo dokonce začne replikovat jeho data.

```
node.name: es-eeg-dev
```

Jedinečné jméno této instance. Pokud bychom více instancí zformovali do stejného clusteru (eeg-dev), každý musí mít jiný název.

```
path.conf: /etc/elasticsearch
path.data: /home/elasticsearch/data
path.logs: /home/elasticsearch/logs
path.plugins: /home/elasticsearch/plugins
```

Jedná se o cesty, kde je uložena konfigurace tohoto nodu, kde budou fyzicky uložená data, logy a dodatečné pluginy. Protože na serveru je automaticky zálohován adresář /home, navíc automaticky vytvořenému uživateli elasticsearch byl vytvořen i vlastní domovský adresář, bylo vhodné veškeré cesty nasměrovat právě sem.

```
discovery.zen.ping.multicast.enabled: false
```

Po spuštění instance Elasticsearch bylo z logů zjištěno, že v rámci univerzitní sítě již někdo Elasticsearch používá. Tato instance pak přijímala multicast informace o onom cizím uzlu. Stále dokola tak naše instance logovala (do souboru), že vidí v síti další node s rozdílným cluster name a tudíž se s ním nespojí. Tato direktiva tedy zaručí to, že Elasticsearch bude podobná příchozí spojení ignorovat.

Na závěr již jenom stačí nastavit, aby se Elasticsearch spouštěl se startem systému:

```
sudo update-rc.d elasticsearch defaults 95 10
```

5.4. Proces indexace

V této kapitole bude rozebráno, jak probíhá proces indexace dokumentu při jeho vložení do databáze. Celému tomuto procesu se říká analýza (analysis). Pro každý field dokumentu je v mapování definován analyzer, který specifikuje, jak se vstupní text rozpadne na jednotlivé termy, nad kterými potom probíhá samotné vyhledávání. Na vstupu celého procesu je tedy původní text, který postupně projde několika fázemi, aby na výstupu byla množina termů, které se uloží do indexu.

Char filtr

Jako první v sérii zpracování je char filtr. Může jich být definováno více za sebou, ale také žádný. Jejich úlohou je příprava textu než přijde do tokenizeru. Typickým zástupcem char filtru je např. odstranění html tagů nebo třeba nahrazení znaku & za "and".

Tokenizer

Další v pořadí je právě jeden tokenizer. Rozděluje souvislý text (upravený charfiltery) na jednotlivé tokeny. Elasticsearch jich obsahuje několik předpřipravených, ale poskytuje možnost vytvořit si i vlastní. Tokeny jsou většinou jednotlivá slova, ale různé tokenizery se liší např. v přístupu, jak oddělují slova s pomlčkami, apostrofy a jinými speciálními znaky - třeba emailovou adresu, která obsahuje tečky, pomlčky a zavináč, většinou není cílem rozdělovat na více tokenů.

Tokenfilter

Tokenfiltrů může být v procesu opět nadefinováno více. Na vstupu mají seznam tokenů, které vyprodukoval předchozí tokenizer. Jejich úkolem je například převést všechna slova na malá písmena, odstranit nevhodné tokeny, např. stopslova: předložky, spojky, členy a další slova, která nenesou žádnou nebo minimální významovou hodnotu a pro účely vyhledávání jsou zbytečná. Token filtry mohou také upravovat slova do jejich základního tvaru (stemming).

Zajímavým zástupcem token filtrů je Edge ngram token filtr. Ten z jednoho vstupního tokenu o délce N znaků udělá N výstupních. Např. z tokenu "ahoj" vyrobí tokeny: ["a", "ah", "aho", "ahoj"]. Tento přístup se používá pro různá našeptávání během psaní (searchboxy a podobně), kde postupně, jak uživatel píše další písmena, tak mu systém vrací stále relevantnější výsledky.

5.5. Struktura uložení dat

To, jak budou data v databázi uložena, se bude zásadně odrážet v možnostech a rychlosti následného vyhledávání. NoSQL sice zjednodušeně znamená, že databáze je bez pevně daného schématu, ale to je velice dvojsečná záležitost. Stejně tak jako klasické databáze, i tady se staví indexy nad uloženými daty, které následně usnadňují jejich vyhledávání. Proto je nezbytné formát ukládaného JSON dokumentu dobře navrhnout.

Nejjednodušší přístup

Základní přístup, který se jako první nabízí, je jednoduše každý parametr zavést jako klíč JSON dokumentu. V případě, že parametrů stejného názvu bude více, tento klíč se rozpadne do pole. Tento přístup rozpadu na pole je pro Elasticsearch poměrně častý, protože Elasticsearch při vyhledávání nerozlišuje mezi polem skalárních hodnot a skalární hodnotou. Pokud parametr by se parametr skládal z dalších sub-atributů, změní se na objekt a opět jako klíč bude veden název tohoto atributu a hodnota bude jeho obsah.

```
{
  _id: "123",
  temperature: 23,
  disease: "blindness",
  software: ["Corel", "Brainvision", "Windows 7"],
  hardware: [
    {title: "intel i5", description: "cool processor"},
    {title: "intel i3", description: "not so cool processor", note: "really slow
one"},
```

```
  ]
}
```

První výhoda tohoto přístupu je ta, že se data dobře prohlížejí (lidským pohledem na JSON). Další výhoda je ve velikosti JSON dokumentu, jedná se o nejkratší možný styl zápisu, který lze vymyslet.

Tento přístup je navíc jednoduchý na implementaci, vyžadoval by nejméně zásahů v kódu aplikace (POJO by zůstalo prakticky stejné, JSON serializer by se nakonfiguroval tak, aby se jednotlivé fieldy ukládaly pod klíči, které odpovídají názvům těchto fieldů). Problém by ale nastal už s definicí mapování pro Elasticsearch. To by bylo pro každý field úplně stejné (jsou to vlastně podobné parametry), při přidání nového parametru by se muselo toto mapování upravovat.

Sofistikovaný přístup

Jak již bylo řečeno v předchozí kapitole: pokud není dopředu nic známo o struktuře a formátu ukládaných dat, je velice problematické nad takovými daty postavit správně index a efektivně v nich vyhledávat. Proto byla navržena následující struktura:

```
{
  _id: "123",
  parameters: [
    { name: "temperature", value: 23 },
    { name: "disease", value: "blindness" },
    { name: "hardware", value: "intel i5",
      attributes: [
        {name: description, value:"cool processor"}
      ]
    },
    { name: "hardware", value: "intel i3",
      attributes: [
        {name: "description", value:"not so cool processor"}
        {name: "notes", value:"really slow one"}
      ]
    }
  ]
}
```

Tento přístup je nepatrně delší na zápis, ale přináší velikou výhodu v tom, že je předem jasně známo, jaké fieldy dokument obsahuje a struktura těchto dokumentů je víceméně předem známa. Při přidání nového typu parametru není nutné vytvářet nový klíč dokumentu a s tím související úprava mapování. Jediný problém, který tento přístup má je ten, že se v klíči value (kde je hodnota nějakého parametru) může objevovat jak číslo (např. u teploty), tak řetězec (nad kterým kvůli fulltextu probíhá textová analýza). V mapování musí být tedy tento field definován jako string s odpovídajícím

analyzerem. Tím se ale vyloučí možnost provádět range-queries nad číselnými hodnotami (např. „získat všechny dokumenty, kde teplota je větší než 22).

Finální struktura

Protože každý field může být mapován pouze na jeden konkrétní datový typ, bylo třeba zavést následující přístup:

```
{
  _id: "123",
  parameters: [
    { name: "temperature", valueInteger: 23 },
    { name: "disease", valueString: "blindness" },
    { name: "hardware", valueString: "intel i5",
      attributes: [
        {name: description, value:"cool processor"}
      ]
    }
  ]
}
```

V tomto přístupu jsou rozděleny číselné a textové hodnoty do vlastních fieldů. Každý pak má nastavené specifické mapování – valueString jako analyzovaný řetězec a valueInteger jako obyčejné číslo. Díky tomu je pak možné na aplikační vrstvě podle toho, co uživatel hledá, sestavit dotaz a zeptat se buď na jeden, nebo druhý field.

5.6. Definice analyzeru

Analyzer v Elasticsearchi popisuje celý proces indexace konkrétního fieldu při vkládání do databáze. Elasticsearch sice v základu poskytuje několik předem definovaných analyzerů, ale pro potřeby Portálu bylo nutné vytvořit vlastní. Ten lze nadefinovat např. při vytváření nového indexu a jeho skladba vypadá následovně:

```
curl -XPOST http://localhost:9200/eegdatabase
{
  index: { number_of_shards: 5, number_of_replicas: 0 },
  analysis: {
    analyzer: {
      standard_snowball_analyzer: {
        type: "custom",
        tokenizer: "standard",
        filter: ["standard", "lowercase", "stop", "eeg_synonym", "eng_snowball"]
      }
    },
    filter: {
      eng_snowball: { type: "snowball", language: "English" },
      eeg_synonym:{type: "synonym",synonyms_path: "/home/elasticsearch/synonyms.txt"}
    }
  }
}
```

Je zde vidět definice vlastního analyzeru, který se skládá z jednoho tokenizeru a několika token filtrů.

5.6.1. Tokenizer

Je použit standardní tokenizer. Ten rozděluje slova podle mezer, pomlček a dalších znaků, specifikovaných v Unicode Text Segmentation algoritmu [12] a hodí se pro většinu evropských jazyků. Proto je použit v tomto analyzeru. Výstupem tokenizeru jsou jednotlivé tokeny, na které jsou postupně aplikovány jednotlivé token filtry v pořadí tak, jak jsou definovány.

5.6.2. Token filtry

Token filtry jsou postupně aplikovány na výstupní proud tokenizeru a upravují, případně úplně odebírají tokeny.

Standard

Jde o jednoduchý filtr, který z tokenů odebírá znaky 's a tečky ze zkratek, takže např. z Apple's C.E.O. se stane Apple CEO.

Lowercase

Toto je opět jeden z předdefinovaných filtrů, který pouze všechna písmena převádí za malá.

Stop

Stop filtr má na starosti odstranění slov (tokenů), které nemají žádný sémantický význam – jde hlavně o předložky, spojky, členy a další slova, která nemají vliv na podstatu hledaného textu a ve většině případů spíše zhoršují nalezené výsledky.

Eeg_synonym

Jedná se o implementaci předdefinovaného filtru synonym. Jeho úkolem je projít vstupní tokeny a nahradit je podle definice synonym. Synonyma lze definovat buď přímo v konfiguračním JSON dokumentu nebo odkazem na externí soubor – což je preferovaný a zde i použitý způsob. Jsou podporovány dva formáty: Solr a WordNet [13]. Jeden vstupní token pak může být po průchodu tímto filtrem nahrazen jedním nebo více synonymickými tokeny.

Eng_snowball

Poslední v sérii filtrů je pak snowball filtr. Jedná se o filtr, který hledá základní tvary tokenů (stemming) pomocí Sommeru. Je důležité, aby tento filtr byl až jako poslední. Pokud bychom například prohodili pořadí synonymového filtru a tohoto, musela by i synonyma být v lemmatizovaném tvaru. Získat takovou databázi nebo definovat slova v tomto tvaru by bylo velice obtížné.

5.7. Konfigurace mapování

Definice mapování je esenciální pro to, aby šlo nad daty po jejich indexaci vyhledávat. Mapování říká, jakého datového typu budou jednotlivé fieldy dokumentu, případně jak bude probíhat textová analýza nad těmito fieldy.

5.7.1. Automatické mapování

Elasticsearch sice nabízí automatické rozpoznávání datových typů na základě vkládaných dokumentů – mapování pro konkrétní field se vytvoří při indexaci prvního dokumentu, který tento field obsahuje. Tato funkce se ale hodí jen pro seznamování se s Elasticsearchem a pro jednoduché ukázkové příklady. Pro jakékoliv reálné aplikace je tato funkcionalita spíše nežádoucí. Může se totiž stát, že se do databáze začnou ukládat dokumenty obsahující fieldy, ke kterým se vygeneruje (většinou špatné) automatické mapování, se špatnou definicí analyzeru a tyto dokumenty se pak samy (nevhodně) indexují. Následná změna není jednoduchá, protože Lucene nedovoluje modifikaci již indexovaných dokumentů a je tedy třeba všechny dokumenty smazat a vložit znovu.

5.7.2. Ruční mapování

Z výše uvedeného plyne, že je nutné nadefinovat mapování pro ukládané dokumenty (resp. typy dokumentů) ihned po vytvoření indexu a před tím, než je vložen první dokument. K tvorbě tohoto mapování Elasticsearch poskytuje tzv. PUT Mapping API. To bylo využito pro definici následujícího mapování, které reflektuje zvolenou strukturu z předchozí kapitoly.

```
curl -XPUT http://localhost:9200/eegdatabase/experiment/_mapping
{
  experiment: {
    dynamic: "strict",
    properties: {
      params: {
        type: "nested",
        properties: {
          name: { type: "string", index: "not_analyzed" },
          valueInteger: { type: "double" },
          valueString: { type: "string", analyzer: "standard_snowball_analyzer" },
```

```
          attributes: {
            type: "nested",
            properties: {
              name: { type: "string" },
              value: { type: "string", analyzer: "standard_snowball_analyzer" }
            }
          }
        }
      }
    }
  }
}
```

Na začátku je uvedeno, že se vytváří mapování pro dokumenty typu experiment. Direktiva dynamic znamená to, že vkládaný dokument musí obsahovat pouze fieldy, které jsou specifikovány v tomto mapování. Dokumenty obsahující neznámé fieldy budou odmítnuty. Další direktiva properties popisuje, ze kterých sub-položek se bude dokument skládat. Je uvedeno, experiment obsahuje field s názvem params, což je pole reprezentující jednotlivé parametry (Elasticsearch nerozlišuje mezi JSON objektem a polem objektů). Každý parametr (param) se pak skládá ze čtyř dalších položek[10]: name, valueInteger, valueString a attributes. U fieldu valueString je pak vidět definice specifického analyzeru – který byl vytvořen při tvorbě indexu. Velice důležité je nastavení mapování params jako tzv. typu nested.

5.7.3. Nested mapování

Nested mapování říká, jak se bude při indexování nakládat s vnořeným objektem. Nejlépe bude vše vysvětleno na následujícím příkladu.

Mějme následující JSON dokument připravený k uložení do databáze:

```
{
  experimentId: "123",
  params: [
    { name: "hardware", value: "intel"},
    { name: "software", value: "corel"}
  ]
}
```

Pokud bychom položili tento dotaz (nyní bude uveden pouze v náznaku, skutečná syntaxe Elasticsearch dotazů bude rozebrána v následující kapitole):

```
GET documents WHERE type=experiment AND param.name=hardware AND param.value=corel
```

Bez definice mapování nested (výchozí chování) bychom mezi výsledky dostali i výše uvedený dokument. Je třeba svázat společně jednotlivé klíče jednoho sub-

[10] Jejich význam je detailně popsán v předchozích kapitolách.

dokumentu. Proto je nutné v definici mapování nastavit jednotlivé parametry jako nested-dokumenty. Elasticsearch je poté interně indexuje jako samostatné dokumenty (tak, jako by byly do databáze vloženy separátně), ale je postaráno o to, aby byly uloženy ve stejném bloku jako jejich nadřazený dokument a byly tak rychle vyhledány [14].

5.8. Tvorba dotazů

Pokládání dotazů do Elasticsearch probíhá opět formou http požadavků na server, kde databáze běží (standardně na portu 9200). Nejjednodušší dotaz, který vrátí všechny dokumenty, vypadá takto:

```
POST localhost:9200/eegdatabase/experiment/_search
{
  query: { match_all: {} }
}
```

Pro potřeby Portálu bylo nutné namodelovat sofistikovanější dotaz, který bude reflektovat požadavky na vyhledávání. Po ujasnění zadání se zadavatelem bylo rozhodnuto, že z aplikace půjde položit dotaz, který bude filtrovat experimenty, přičemž na vstupu bude jednak seznam parametrů (ve formátu typ-hodnota), které se v dokumentu musí vyskytovat a pak seznam parametrů, které se v něm vyskytovat nesmí. Musí být reflektováno výše rozebrané nested mapování, aby nedocházelo k nechtěnému slučování neodpovídajících si parametrů. Výsledný dotaz pak vypadá následovně:

```
{
filter: {
  and: {
    filters: [ {
      nested: {
        path: "params",
        query: { bool: { must: [
          { term: {params.name : "hardware"}},
          { match: {params.valueString : {query: "intel"}}}
        ]}}
      }
    }, {
      nested: {
        path: "params",
        query: { bool: { must: [
          { term: { params.name : "software" }},
          { match: { params.valueString : { query: "corel“ }}}
         ]}},
      }
    }, {
      bool: { must_not: {
         nested: {
           query: {bool: {must: [
             { term: { params.name : "hardware" }},
             { match: { params.valueString : { query: "brainvision" }}}
           ]}},
         }
       }
     }
```

```
}]}}
```

Elasticsearch dotazy jsou často poměrně těžko čitelné a komplikované. Je to vykoupeno tím, že existuje možnost přesně popsat, jaká data jsou potřeba a jak se získají z prohledávaného indexu, což vede k možnosti optimalizovat dotaz podle zamýšleného způsobu použití. Zde je vidět dotaz, který hledá celkem podle tří kritérií (ta musí platit všechna společně – AND):

- Experiment obsahuje parametr, kde hardware JE intel
- Experiment obsahuje parametr, kde software JE corel
- Experiment neobsahuje parametr, kde hardware JE brainvision

Ve všech případech se jedná o nested queries, přičemž poslední z nich je negovaná za pomoci boolean query a typu must_not.

5.9. Integrace do stávajícího systému

V jenom ze základních požadavků na rozšíření Portálu znělo, aby se co nejvíce změn odehrávalo pouze na datové vrstvě tak, aby zbytek aplikace vůbec nemusel zjišťovat, že přibylo nové datové úložiště, které entity případně jejich části jsou v něm uloženy a jak s ním komunikovat. Veškerá integrace tedy probíhala pouze na nejnižší vrstvě Portálu.

5.9.1. Spring Data Elasticsearch

Aby se nemuselo komunikovat s Elasticsearchem přímo, posílat http požadavky, parsovat JSON odpověď a pak ručně mapovat jednotlivé JSON fieldy na konkrétní Java property, bylo nutné najít vhodný ovladač, který tuto práci obstará automaticky. Volba padla na relativně nový projekt spring-data-elasticsearch [15], který řeší všechny výše zmíněné operace. Výhodné na něm je to, že je psán jako rozšíření frameworku Spring (který portál využívá), takže jeho připojení do projektu je velice snadné.

Funguje na stejném principu jako stávající projekt, nadefinuje se POJO třída s nepatrně odlišnými anotacemi než při využití JPA[11] a nad ní poté operuje tzv. Repository (což je obdoba DAO entity v relačním schématu).

[11] JPA: Java Persistence API – rozhraní, které implementuje například Hibernate.

5.9.2. Tvorba Elasticsearch entity

Jak již bylo výše popsáno, jedná se o obdobu POJO entity z relační databáze. Anotace @Document specifikuje, že se jedná o Elasticsearch entitu a @Field pak určuje, jak se jednotlivé properties mapují při serializaci do JSON dokumentu.

```
@Document(indexName = "eegdatabase", type = "experiment", replicas = 0, shards = 5)
public class ExperimentElastic implements Serializable {

    @Id
    private String experimentId;
    @Field(type = FieldType.Nested)
    private List<GenericParameter> params = new ArrayList<GenericParameter>();
    @Field(type = FieldType.Integer)
    private int userId;
    @Field(type = FieldType.Integer)
    private int groupId;

    public ExperimentElastic() {}
}
```

5.9.3. Úprava POJO

Do POJO třídy reprezentující (relační) část experimentu přibyla reference na výše popsanou třídu ExperimentElastic, přičemž gettery a settery na získávání a nastavování konkrétních parametrů fungují tak, že transparentně toto volání přeposílají na podřízenou entitu do Elasticsearche. Anotace @Transient slouží k tomu, aby se Hibernate nepokoušel ukládat tyto properties do relační databáze. Na venek tedy vše vypadá jako jedna entita, ale vnitřní implementace je naprosto skryta.

```
@Entity
@Table(name = "EXPERIMENT")
public class Experiment implements Serializable {
    private int experimentId;
    private ExperimentElastic elasticExperiment = new ExperimentElastic();

    public void setElasticExperiment(ExperimentElastic e) {
        this.elasticExperiment = e;
    }

    @Transient
    public ExperimentElastic getElasticExperiment() {
        return this.elasticExperiment;
    }

    @Transient
    public List<GenericParameter> getGenericParameters() {
        return this.elasticExperiment.getParams();
    }

    public void setGenericParameters(List<GenericParameter> params) {
        this.elasticExperiment.setParams(params);
   }
}
```

5.9.4. Interceptor

Výše popsaný algoritmus sice při práci s parametry experimentu interně zaznamenává tyto informace do Elasticsearch POJO entity, ale ta se nijak automaticky nesynchronizuje s Elasticsearch databází. Bylo třeba nalézt způsob, jak během ukládání (aktualizaci) entity v relační databázi automaticky uložit i nerelační část – aby volající nemusel zvlášť volat Hibernate.saveEntity() a Elasticsearch.saveEntity(). A právě pro tento případ existuje Interceptor.

Interceptor je třída, kterou poskytuje Hibernate framework proto, aby se dalo reagovat na různé akce, kdy aplikace interaguje s databází (práce s Hibernate session). Dovoluje registrovat obsluhu na množství událostí (nejčastěji se toto používá při logování práce s databází). Lze pracovat například s těmito akcemi:

- onLoad: Metoda je zavolána pro každou událost, kdy je načítána jedna entita z databáze (např. voláním DAO.read()).
- onDelete: Tato metoda je volána ve všech případech, kdy jde přes Hibernate požadavek na smazání nějaké entity (DAO.delete()).
- onSave: Metoda je zavolána při ukládání nové entity do relační databáze.
- onFlushDirty: Metoda je zavolána při aktualizaci stávající entity v relační databázi.

Interceptor poskytuje ještě množství dalších událostí, které ale nejsou pro potřeby synchronizace Hibernace (relační DB) a Elasticsearche podstatené. Konkrétní implementace Interceptoru vypadá následovně (pro zjednodušení je uvedena implementace pouze metody load – dodatečné načtení nerelačních dat z Elasticsearche při načítání experimentu z relační databáze a jejich spojení do jedné výsledné entity):

```
public class ElasticSynchronizationInterceptor extends EmptyInterceptor {

  @Resource
  private ElasticsearchTemplate elasticsearchTemplate;

  @Override
  public boolean onLoad(Object entity, Serializable id, Object[] state, String[]
propertyNames, Type[] types) {
    boolean res = super.onLoad(entity, id, state, propertyNames, types);

    if (entity instanceof Experiment) {
      Experiment e = (Experiment) entity;
      SearchQuery searchQuery = new NativeSearchQueryBuilder().withQuery(new
```

```
IdsQueryBuilder("experiment").addIds("" + e.getExperimentId())).build();
      List<ExperimentElastic> elastic =
elasticsearchTemplate.queryForList(searchQuery, ExperimentElastic.class);
      if (elastic.size() > 0 && elastic.get(0) != null) {
        e.setElasticExperiment(elastic.get(0));
      }
    }

    return res;
  }
```

V kódu je vidět, že se reaguje na událost načítání Hibernate entity. Pokud se jedná o entitu typu experiment, sestaví se dotaz do Elasticsearche, který vyhledá odpovídající data podle ID experimentu a získají se tak nerelační parametry. Nakonec se nalezená data z Elasticsearche propíší i do Hibernate entity, čímž (díky transparentnímu provolávání metody Experiemnt.getParameters()), začne fungovat přístup popsaný v kapitole „Úprava POJO“.

5.9.5. Úprava Experiment DAO

V DAO třídě pro experiment (SimpleExperiemntDAO.java) nakonec stačilo přidat jen konkrétní metody pro fulltextové vyhledávání nad parametry uloženými v Elasticsearchi. Implementace takové metody vypadá následovně (naštěstí na rozdíl od surových JSON dotazů je kód mnohem čitelnější):

```
public class SimpleExperimentDao {

  @Autowired
  private ElasticsearchTemplate elasticsearchTemplate;
  @Override
  @Transactional(readOnly = true)
  public List<Experiment> searchByParameters(GenericParameter[] contains,
GenericParameter[] notContains) {

    AndFilterBuilder and = new AndFilterBuilder();
    for (GenericParameter p : contains) {
      BoolQueryBuilder b = new BoolQueryBuilder();
      Object value = p.getValueString() == null ? p.getValueInteger() :
p.getValueString();
      String fieldName = p.getValueString() == null ? "params.valueInteger" :
"params.valueString";
      b.must(termQuery("params.name", p.getName())).must(matchQuery(fieldName,
value));
      and.add(new NestedFilterBuilder("params", b));
    }

    for (GenericParameter p : notContains) {
      BoolQueryBuilder b = new BoolQueryBuilder();
      Object value = p.getValueString() == null ? p.getValueInteger() :
p.getValueString();
      String fieldName = p.getValueString() == null ? "params.valueInteger" :
"params.valueString";
      b.must(termQuery("params.name",
             p.getName())).must(matchQuery(fieldName, value));
      BoolFilterBuilder not = new BoolFilterBuilder();
      not.mustNot(new NestedFilterBuilder("params", b));
```

```
      and.add(not);
    }

    SearchQuery searchQuery = new NativeSearchQueryBuilder().withFilter(and).build();
    List<ExperimentElastic> list =
this.elasticsearchTemplate.queryForList(searchQuery, ExperimentElastic.class);
    return this.transformEsResultToHibernate(list);
  }
}
```

5.10. Shrnutí

Těmito kroky je zajištěna celková integrace databáze Elastiscearch do stávající aplikace Portálu. Jde jednak o kompletní propojení entit datové vrstvy – relační experiment POJO entita obsahuje referenci na nerelační entitu ExperimentElastic a každá si obsluhuje property, které jsou uloženy v „její“ části. Navenek se pracuje jen s relační entitou a při požadavku na nerelační data je tento požadavek předán interní nerelační entitě. Dále, protože POJO třídy samy o sobě nemají žádnou funkčnost ve smyslu výkonného kódu (pouze sdružují data), byl přidán Interceptor, který se automaticky stará o dodatečné načtení (případně uložení) dat z nerelační entity do Elasticsearch databáze. Nakonec byla rozšířena DAO třída o několik metod, které jsou schopné fulltextového vyhledávání nad Elasticsearch daty a na výstupu vrací kompletní entity tak, jako by byly načteny z relační databáze.

5.11. Automatické testy

Portál obsahuje množství různých testů (jednak unit testy, několik integračních a dokonce i selenium[12] testy), které by se měly spouštět před každým commitem do centrálního repositáře a také před každým nasazením nové verze aplikace. Tímto by mělo být zajištěno, v kombinaci s iterativním způsobem vývoje, že aplikace bude stále v konzistentním a funkčním stavu.

Bohužel v době psaní této práce automatické testy vůbec nefungovaly. Na vině byl nejspíše neaktuální konfigurační soubor pro Spring (test-context.xml). Navíc, protože testy takto nefungovaly delší dobu, dá se předpokládat, že původní testy budou neaktuální a buď nebudou procházet vůbec, nebo budou testovat pouze omezenou část kódu.

[12] Selenium testování: Specifický druh testů, kdy je simulováno klikání uživatele do aplikace v prohlížeči a je kontrolována odezva aplikace (přechod na další stránku, zobrazení pop-up dialogu apod.).

Proto byl vytvořen jednoduchý skript (samozřejmě v rámci kódu Portálu), který je nutné spouštět ručně (není tedy zasazen do kontextu ostatních testů) po spouštění aplikace. Jedná se o integrační test, který zkusí vytvořit nový experiment s několika parametry, které mají být uloženy v databázi Elasticsearch. Poté je proveden dotaz přímo do Elasticsearche a zkontroluje se shoda uložených parametrů. Následně se experiment upraví a opět se zkontrolují hodnoty jednotlivých parametrů. Na závěr se tento testovací experiment smaže a ověří se, že je smazán i z databáze Elasticsearch.

Testovací skript je napsán tak, aby v momentě, kdy se zprovozní celá sada testů, které jsou nyní v Portálu, šel jednoduše integrovat mezi ostatní testovací scénáře.

6. Změna relační databáze

Oracle je jedna z největších “enterprise” databází. Někdy se řadí mezi tzv. velkou trojku, společně s IBM DB2 a Microsoft SQL Serverem. Nabízí pokročilé možnosti jako zamykání tabulek, transakční zpracování, sekvence pro generování primárních klíčů, nástroje pro replikaci a horizontální škálování, triggery, psaní uložených procedur v jazyce PL/SQL. Bohužel z hlediska licence přestává být pro portál použitelná [3].

Je tak třeba najít nějakou adekvátní náhradu, ideálně zdarma řešení, které bude dostatečně suplovat původní funkce Oracle DB. V úvahu přicházejí tyto čtyři databázové systémy:

- MySQL
- SQLite
- MariaDB
- PostgreSQL

6.1. Vhodní kandidáti

6.1.1. MySQL

Jedná se asi o nejrozšířenější zdarma databázové řešení [14]. Je de facto standardem na většině serverech, které poskytují levné hostování webových stránek a jednodušších aplikací. Jedná se o široce podporovanou databázi. Většinou je nasazována v malých aplikacích na platformě LAMP (Linux + Apache + MySQL + PHP).

Drobný problém MySQL spočívá v ne úplně přesném dodržování SQL standardů (chybí např. windowing functions) a několik dalších odchylek [15]. Ty sice nejsou pro většinu aplikací esenciální, ale roli při rozhodování to může sehrát. Dále používá několik tzv. “storage enginů”, přičemž každý se hodí na jiné použití, některý podporuje transakce, jiný se hodí na archivaci a jen občasné čtení dat. Je poměrně složité určit, který se hodí pro jaký use-case. Nejpoužívanější storage enginy jsou následující [16]:

- MyISAM - výchozí engine, velmi rychlý, bez podpory transakcí a referenční integrity (cizí klíče)
- InnoDB - transakce podporující engine (příkazy commit, rollback), umí cizí klíče, ne tak rychlý
- Memory - Všechna data ukládá v paměti RAM, extrémně rychlé úložiště, při restartu databáze jsou ale data ztracena.

Úplně chybí podpora pro sekvence a místo toho se vytváří AUTO_INCREMENT sloupec pro automatické generování primárních klíčů pro každou tabulku zvlášť.

Nezanedbatelný problém s MySQL je ten, že při přidání nového sloupce zamkne celou tabulku a celý proces může (v závislosti na velikosti dat) trvat až několik hodin. Během této operace je samozřejmě tabulka nepřístupná jak pro čtení, tak i pro zápis. Toto může hrát významnou roli při rozhodování, kterou databázi použít.

Dalším důležitým faktorem ke zvážení je fakt, že Oracle v roce 2009 koupil MySQL (společně s akvizicí Sun Microsystems, která byla předchozím vlastníkem MySQL) [17]. Existuje spousta obav a názorů, že Oracle může časem z této databáze udělat uzavřený SW a celou databázi tak značně znehodnotit (kvůli prosazování své hlavní Oracle DB).

6.1.2. SQLite

Jedná se o jednu z nejjednodušších SQL databází. Nejedná se o databázový server v pravém slova smyslu - každá databáze je uložena jako jeden samostatný soubor. Ačkoliv implementuje většinu standardu SQL-92, transakční zpracování přibylo až nedávno, nemá například plnou podporu triggerů a upravování stávajících tabulek má určitá omezení [18].

Největším problémem je ale rychlost. Protože neexistuje centrální server, který by řídil synchronizaci přístupu, musí se pro každou operaci otevírat a zavírat již zmíněný databázový soubor, což je časově velice drahá operace. Tato databáze se tedy spíše hodí na jednoduché aplikace s maximálně stovkami záznamů. Často se používá jako datové úložiště pro mobilní aplikace nebo jednoduché desktopové aplikace, kde se nepředpokládá enormní zátěž a velký růst objemu dat.

6.1.3. MariaDB

Právě z obav o otevřenost systému MySQL vzešel fork [19] této databáze, který dostal pojmenování MariaDB. Obsahuje ovšem stejné neduhy co byly zmíněny u MySQL, navíc přidává některé další storage enginy (např. TokuDB), čímž situaci s jejich výběrem ještě více komplikuje.

6.1.4. PostgreSQL

Asi nejkomplexnější a nejvíce enterprise-like řešení z vybraných databází [20]. Podporuje sekvence (navíc má i speciální syntaxi, která práci s nimi zjednodušuje), triggery, možnost psaní uložených procedur v modifikovaném jazyce PL/pgSQL. Dále pracuje s modelem ukládání dat ve struktuře databáze-schéma-tabulka (podobně jako Oracle), čímž se této databázi ještě více přibližuje. V neposlední řadě je zajímavostí databáze Postgres sloupec typu OID (Object Identifier). Jedná se techniku, jak ukládat velká binární data mimo samotnou tabulku, kam daný sloupec patří. V tomto sloupci je pak pouze reference (identifikátor) do interní systémové tabulky. Dobrou zprávou je také to, že Hibernate s tímto typem umí dobře spolupracovat a umožňuje na něj aplikovat tzv. Lazy Loading - data se načtou až v momentě, kdy jsou skutečně potřeba. Toto by bylo vhodné u všech entit, které obsahují binární data (Scenario, Experiment…).

6.2. Výběr

Vlastnost	MySQL	SQLite	MariaDB	PostgreSQL
Podpora standardu SQL	+	-	+	++
Práce s binárními daty	++	-	++	++
Jednoduchost konfigurace	+	+++	+	++
Open source projekt	+	+++	+++	+++
Struktura databáze-schéma-tabulka	-	-	-	+
Škálování při zátěži	+	-	++	++

Vzhledem k výše popsaným skutečnostem je vidět, že všechny databáze jsou velmi vyrovnané a obstojně by zvládly uchovávat data z Portálu. Protože nejvíce kladných vlastností a výhod oproti ostatním databázím má systém PostgreSQL, byl jako primární relační datové úložiště zvolen právě on.

6.3. Příprava migrace

6.3.1. Úprava Hibernate mapování

Jak již bylo popsáno, existují dva přístupy mapování entit - XML a Java anotace, přičemž je dlouhodobá snaha, aby se upustilo od XML a přešlo se na nové, jednodušší a přehlednější anotační konfigurace.

Problém byl v také v tom, že (nejspíše jako semestrální projekt) se v minulosti už někdo o tento přepis pokoušel, a to tak, že část existujících POJO entit obsahovala anotace. Ty se ale bohužel nepoužívaly (až na pět entit) a byly často neaktuální nebo neúplné - neodpovídaly stávajícímu datovému modelu v databázi. Nějaké slučování by bylo velice náročné a ruční přepis by zabral mnoho času (Portál obsahuje přes 60 entit). Proto bylo nutné najít nějaké automatizované řešení. Naštěstí existuje aplikace JBoss Developer Studio, která poskytuje přesně tuto funkcionalitu. Nebylo možné jakkoliv sloučit anotace a XML konfiguraci, navíc použité anotace stejně byly v době psaní této práce neaktuální. Velkou výhodou bylo také to, že díky správnému používání architektury a POJO tříd, entity neobsahovaly žádný výkonný kód, pouze strukturu dat. Bylo tedy možné tyto stávající entity smazat a kompletně je nahradit vygenerovanými třídami z Developer Studia.

Drobná chyba nastala ve vygenerovaném kódu ve všech vazbách, kde se alespoň na jedné straně vyskytuje kolekce (vazby typu 1:N a M:N). Tyto vygenerované entity totiž obsahovaly netypovanou kolekci Set, namísto generické Set<OtherEntity>. Změna nastavení se skrývala pod celkem neočekávaným a skrytým tlačítkem “Používat syntaxi Java 5”. Po této úpravě se již POJO třídy s anotacemi vygenerovaly správně.

6.3.2. Přechod na anotace

Stávající konfigurace vypadala tak, že ty třídy, kde se používaly anotace, musela být jedna po druhé zvlášť vyjmenována následujícím způsobem v souboru hibernate.cfg.xml:

```
<mapping class="cz.zcu.kiv.eegdatabase.data.pojo.License">
<mapping class="cz.zcu.kiv.eegdatabase.data.pojo.PersonalLicense">
```

Všechny ostatní třídy, kde se používalo XML mapování, potom byly souhrnně nastaveny v souboru persistence.xml takto:

```
<property name="classpath*:cz/zcu/kiv/eegdatabase/data/pojo/*.hbm.xml">
```

Již zde je vidět první problém. Používají se dva odlišné způsoby konfigurace, které se liší jak syntaxí, tak i svým umístěním. Navíc i definice toho, který způsob je pro jaké entity použit, se nachází na jiném místě.

Vzhledem k tomu, že se XML konfigurace úplně přestaly používat, bylo možné druhý řádek úplně odstranit. Bylo ale třeba vyřešit další neefektivní řešení - a sice vyjmenovávání každé anotované třídy zvlášť. Naštěstí Spring framework je na takovýto požadavek připraven [21] a stačilo pouze změnit implementaci Java beany sessionFactory, kde se používala následující (soubor persistence.xml):

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
annotation.LocalSessionFactoryBean">
```

na tuto:

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
annotation.AnnotationSessionFactoryBean">
```

která obsahuje property packagesToScan, která říká, kde jsou obsaženy veškeré anotované třídy:

```
<property name="packagesToScan" value="cz.zcu.kiv.eegdatabase.data.pojo"/>
```

Zároveň se podařilo všechny údaje o připojení do databáze, velikosti connection poolu, cache a dalším nastavení Hibernate přesunout do souboru persistence.xml (který automaticky načítá proměnné ze souboru project.properties), čímž se celý soubor hibernate.cfg.xml stal nepotřebným a mohl být odstraněn.

6.3.3. Dlouhé řetězce

Při přecházení na databázi Postgres vznikl problém s ukládáním dlouhých textových řetězců. Postgres zná celkem tři datové typy pro práci s textem:

- character (n): řetězec pevné délky n, doplněn o prázdný padding
- character varying (n): řetězec proměnlivé délky do maximální délky n
- text: řetězec neomezené délky

Pokud se nechá vše v základní konfiguraci a proměnná se v POJO třídě označí:

```
@Column(name = "TEXT")
private String text;
```

Hibernate pro ni vygeneruje v databázi sloupec typu character varying(255) – textový řetězec o maximální délce 255 znaků. Toto omezení je v pořádku pro většinu nadpisů, názvů apod., ale v případě delších popisů experimentů nebo scénářů jsou řetězce často delší. V dokumentaci Hibernate je uvedeno, že pro tyto případy se má použít anotace @javax.persistence.Lob. Vygenerování schématu zafunguje správně, vytvoří se sloupec typu text, ale při pokusu o čtení této hodnoty je vyvolána výjimka org.postgresql.util.PSQLException: Bad value for type long. Je to způsobeno tím, že při čtení se Hibernate snaží tuto hodnotu interpretovat jako dříve popsaný datový typ OID, kde je v sloupci pouze číselný identifikátor do externí tabulky. Přestože se jedná určitě o chybu, ať už v samotném Java-Postgres driveru nebo v Hibernate interním mapování, vývojáři toto označili za vlastnost [22].

Jediné řešení je přidat ještě anotaci @Type, kde se specifikuje přímo datový typ, který se má pro sloupec použít a Hibernate pak začne fungovat podle očekávání. Kompletní zápis potom vypadá takto:

```
@Lob
@Type(type = "org.hibernate.type.TextType")
@Column(name = "TEXT")
private String text;
```

Je možné, že časem bude tato chyba opravena a anotace @Type, která navíc zanáší závislost na Hibernate datovém typu (od kterého by aplikace měla být odstíněna díky JPA), by mohla být odstraněna.

6.3.4. Binární data

Stejně jako pro řetězce, tak i pro binární hodnoty poskytuje Postgres více datových typů:

- bytea: Zkratka pro byte array, úložiště pro jednodušší binární data do max. velikosti 1GB. Do Hibernate se mapuje jako byte[] dataFile;
- OID: Již několikrát zmiňovaný Object Identifier, data se ukládají v speciální systémové tabulce, max. velikost do 4GB. V Hibernate se mapuje jako datový typ java.sql.Blob.

Velikostní omezení bytea by asi nebyl pro potřeby Portálu problém, ale více nepříjemností přináší nutnost v Java kódu použít primitivní typ byte[]. Sice je kód jednoduchý a čitelný, ale Hibernate bohužel nezvládá tento datový typ Lazy loadovat.

Načtení přehledové tabulky několika experimentů pak trvá příliš dlouho (i když binární data nejsou vůbec třeba), protože se všechna data musí přenést z databázového serveru na aplikační. Ve většině případů pak aplikace spadne na nedostatku volné paměti.

Proto pro implementaci binárních dat pro Experiment a Scenario je použit typ OID. Definice takové proměnné potom vypadá následovně:

```
@Basic(fetch=FetchType.LAZY)
@Lob
@Column(name = "FILE_CONTENT", nullable = false)
private Blob fileContent;
```

6.4. Instalace PostgreSQL

Pro instalaci databáze Postgres byl vyčleněn virtuální stroj běžící na operačním systému Debian 7 (Wheezy). Postgres (aktuálně ve verzi 9.1) se nachází v oficiálním repositáři aplikací, takže instalace přes balíčkovací systém je jednoduchá záležitost:

```
sudo apt-get install postgresql-9.1
```

Po instalaci je třeba v souboru /etc/postgresql/9.1/main/pg_hba.conf povolit příchozí spojení z vnějšího světa. Jedná se o úpravu rozsahu IP adres kolem řádku 92:

```
# IPv4 local connections:
host    all             all             0.0.0.0/0           md5
```

Databáze se rovnou nainstaluje jako služba a nastaví se spouštění při startu systému. Není třeba nic víc řešit.

6.5. Migrace dat

Jak původní Oracle databáze procházela postupem času změnami datového modelu, tak v ní zůstalo několik tabulek, které již s aplikací nijak nesouvisely. Navíc proběhl upgrade verze samotné databáze a během tohoto procesu bylo vytvořeno mnoho (nejspíše) dočasných tabulek (začínajících jménem SYS_*). Tyto tabulky jsou nyní nepotřebné a není nutné je přenášet i do nové databáze.

Nejlepší řešení bylo nastavit aplikaci připojení do nově vytvořené (prázdné) databáze Postgres a přepnutí Hibernate do módu hibernate.hbm2ddl.auto=create. Tato direktiva se postará při zavádění aplikace o vytvoření všech nezbytných databázových tabulek podle aktuálního mapování jednotlivých entit. Tím je zaručeno, že se vytvoří naprosto všechny tabulky, které bude aplikace kdykoliv při svém běhu potřebovat.

Dalším krokem pak bylo napsat jednoúčelový skript, který projde všechny takto vytvořené tabulky a pro každou z nich vybere všechny záznamy z původní Oracle databáze a vloží tyto řádky do Postgresu. Zajímavostí je SQL dotaz, jak získat všechny tabulky z určité Postgres databáze:

```
SELECT * FROM information_schema.tables WHERE table_schema = 'public' AND
table_catalog='eeg'
```

Drobný problém byl s referenční integritou, kdy by se musely migrovat tabulky ve správném pořadí podle vzájemných závislostí. Sestavování takové posloupnosti by bylo náročné, proto bylo nejjednodušší dočasně po dobu migrace vypnout kontrolu cizích klíčů. Toho lze v Postgresu dosáhnout následujícím dotazem (pro každou tabulku):

```
ALTER TABLE experiments DISABLE TRIGGER ALL
```

7. Zhodnocení výsledků

Stěžejním úkolem této diplomové práce bylo analyzovat situaci kolem fulltextového vyhledávání, provést srovnání několika databázových systémů a nalézt nejvhodnější. Dále bylo třeba tento systém integrovat do stávající aplikace EEG/ERP portálu tak, aby bylo možné provádět fulltextové hledání nad parametry a metadaty měřených experimentů. Jako nejvhodnější úložiště byla po teoretickém, výkonnostním a i subjektivním srovnání vybrána databáze Elasticsearch.

Dále se práce věnovala přechodu na jiné databázové úložiště všech ostatních (relačních) dat. Bylo nutné opustit databázi Oracle a nahradit ji jinou, co nejpodobnější, aby se při přechodu muselo udělat co nejméně zásahů v kódu aplikace. Jako vhodná alternativa k Oraclu byla vybrána databáze PostgreSQL, do které byla přesunuta všechna relační data Portálu.

Během plnění těchto hlavních cílů se autor práce věnoval refaktoringu a celkovému zpřehlednění datové vrstvy. Podařilo se mimo jiné sjednotit konfiguraci aplikace (přihlašovací údaje do databází, nastavení Hibernate a další věci) do jednoho souboru nebo přejít na anotační mapování pro Hibernate, čímž se značně zjednodušil kód aplikace pro budoucí rozšiřování.

Do budoucna by bylo vhodné, aby se přepsala prezentační vrstva Portálu tak, aby používaly přímo generické parametry z databáze Elasticsearch. Nyní se stále přistupuje k pojmenovaným parametrům načítaným z relační části. Také by bylo vítané, kdyby se rozrostl počet experimentů uložených v Portálu. V současné chvíli je v databázi asi 200 měřených experimentů. Aby šel naplno využít potenciál databáze Elasticsearch, případně otestovat funkčnost navrženého řešení v reálné situaci, hodilo by se mít dat alespoň 100x více.

Seznam použitých zkratek

API: Application Programming Interface, rozhraní pro programování aplikací.

DAO: Database Access Object, objekt poskytující abstraktní rozhraní pro databázi nebo jiné perzistentní úložiště.

EEG: Elektroencefalogram, záznam časové změny elektrického potenciálu způsobeného mozkovou aktivitou. Tento záznam je pořízen elektroencefalografem.

ERP: Event Related Potencial, evokovaný potenciál je měřená odpověď mozku na konkrétní vyvolanou událost nebo podnět.

ORM: Objektově relační mapování, programovací technika v softwarovém inženýrství, která zajišťuje automatickou konverzi dat mezi relační databází a objektově orientovaným programovacím jazykem.

POJO: Plain Old Java Object, Jednoduché Java třídy, které splňují určité podmínky.

REST: Representational State Transfer, architektura rozhraní, navržená pro distribuované prostředí.

SQL: Structured Query Language, standardizovaný dotazovací jazyk používaný pro práci s daty v relačních databázích.

XML: Extensible Markup Language , rozšiřitelný značkovací jazyk. Umožňuje snadné vytváření konkrétních značkovacích jazyků pro různé účely a různé typy dat.

YAML: Yaml Ain't Markup Language je formát pro serializaci strukturovaných dat. Výhodou tohoto formátu je, že je čitelný nejen strojem, ale i člověkem.

Literatura

1. **Rinkes, Jakub.** *Prezenční vrstva EEG/ERP Portálu.* 2013.

2. Basic O/R Mapping. *Hibernate Community Documentation.* [Online] 2014. http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/mapping.html.

3. Oracle Price Lists. *Oracle.* [Online] Oracle, 2014. http://www.oracle.com/us/corporate/pricing/technology-price-list-070617.pdf.

4. About Mongo. *MongoDB - NoSQL & Big Data Database.* [Online] 10gen, 2012. https://www.mongodb.org/.

5. Release Notes for MongoDB 2.4. *MongoDB.* [Online] 10gen, 2013. http://docs.mongodb.org/manual/release-notes/2.4/#text-search.

6. you know, for search. *Elasticsearch.* [Online] Elasticsearch BV, 2010. http://www.elasticsearch.org/blog/you-know-for-search/.

7. About Solr. *Apache Solr.* [Online] Apache Software Foundation, 2012. https://lucene.apache.org/solr/.

8. Solr News. *Apache Solr.* [Online] Apache Software Foundation, 2014. https://lucene.apache.org/solr/solrnews.html.

9. ZooKeeper. *Apache ZooKeeper.* [Online] Apache Software Foundation, 2014. http://zookeeper.apache.org/.

10. Free ebooks - Project Gutenberg. *Project Gutenberg.* [Online] 2014. http://www.gutenberg.net.

11. glossary of termsedit. *Elasticsearch.* [Online] 2014. http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/glossary.html.

12. Unicode Text Segmentation. *unicode.org.* [Online] 2013. http://unicode.org/reports/tr29/.

13. About WordNet. *WordNet - a lexical database for English.* [Online] 2013. http://wordnet.princeton.edu/.

14. What is MySQL. *MySQL Documentation.* [Online] Oracle Corporation, 2014. http://dev.mysql.com/doc/refman/5.1/en/what-is-mysql.html.

15. MySQL Standards Compliance. *MySQL.* [Online] Oracle Corporation, 2014. https://dev.mysql.com/doc/refman/5.0/en/compatibility.html.

16. An Introduction To MySQL Storage Engines. *Linux.org.* [Online] 2013. http://www.linux.org/threads/an-introduction-to-mysql-storage-engines.4220/.

17. Oracle Press Release. *Oralce.* [Online] Oracle Corporation, 2009. http://www.oracle.com/us/corporate/press/018363.

18. About SQLite. *SQLite.* [Online] 2014. https://sqlite.org/about.html.

19. About MariaDB. *MariaDB.* [Online] SkySQL Corporation, 2013. https://mariadb.com/about.

20. About PostgreSQL. *PostgreSQL.* [Online] The PostgreSQL Global Development Group, 2014. http://www.postgresql.org/about/.

21. Hibernate's AnnotationSessionFactoryBean. *My Journey Through IT.* [Online] DINUKA ARSECULERATNE, 2010. http://dinukaroshan.blogspot.cz/2010/06/hibernates-annotationsessionfactorybean.html.

22. PostgreSQL problem when using @Lob on String field. *Hibernate JIRA.* [Online] Atlassian, 2011. https://hibernate.atlassian.net/browse/HHH-6105.