Anotace

Práce se zabývá problematikou optimalizace fulltextového vyhledávání. V textu je, mimo jiné, také popsáno, jak se fulltextově vyhledávalo v minulosti a jaké přístupy se využívají dnes. Je zde detailněji specifikována a rozebrána knihovna Apache Lucene, kterou jako svůj základ používá většina dnešních předních vyhledávacích technologií. Tyto technologie jsou zde popsány a je provedeno jejich porovnání, na základě kterého je vybrána jedna konkrétní technologie Elasticsearch, jejímž směrem se práce orientuje a na které je otestována řada navržených způsobů optimalizace, které se dají využít pro optimalizaci fulltextového vyhledávání širokého spektra aplikací. Kromě jednotlivých metod optimalizace si práce klade za cíl více přiblížit proces vyhledávání od zpracování nového dokumentu, přes jeho indexaci a následné fulltextové vyhledávání v něm. Na základě obsahu práce by měl být každý, kdo se chce informovat o současných technologiích v oblasti fulltextového vyhledávání nebo má v plánu do svého projektu použít některý fulltextový vyhledávač, schopen vybrat tu nejlepší technologii a měl by být schopen ji implementovat a optimalizovat pro svůj projekt.

1. Současný internet a proč je důležité v něm vyhledávat

Na internetu existuje ohromné množství stránek a jejich počet se neustále zvyšuje. Podle [ 6 ] počet stránek na internetu od dob svého vzniku exponenciálně roste. Z čehož plyne, že každým dnem roste rozdíl mezi nově vzniklými webovými stránkami a těmi, již nějakou dobu fungujícími, starými. O sběr dat z nových stránek, aktualizaci dat z upravených starších stránek a umožnění v nich vyhledat požadovanou informaci se starají fulltextové vyhledávače.

Fulltextové vyhledávače jsou zodpovědné za odpověď na položený dotaz a jedním z cílů práce je objasnit proces vyhledávání se zaměřením na vnitřní fungování vyhledávače. Jinými slovy, popsat vnitřní komplexní systém procesu, který je zodpovědný za odpověď položený dotaz.

2. Fulltextové vyhledávání

Vyhledávače, které jsou označeny jako fulltextové, jsou vyhledávače fungující na základě porovnávání fráze se všemi ostatními slovy v daném dokumentu. Je známo více druhů vyhledávačů. Na jedné straně jsou to vyhledávače pracující přímo s textem. Neprobíhá tedy žádná předpříprava porovnávaného dokumentu, ale daná fráze (fráze, kterou chceme vyhledat) je v dokumentu hledána přímo za pomoci některého z vyhledávacích algoritmů, které jsou v práci zmíněny v kapitole “Vyhledávací algoritmy” nebo kterýmkoli jiným algoritmem pracujícím přímo nad textem. Na straně druhé existují algoritmy, konkrétněji souhrn [algoritmů](http://www.adaptic.cz/znalosti/slovnicek/algoritmus/), které dokáží z daného dokumentu vytvořit podrobnou statistiku výskytu jednotlivých pojmů a tu uložit do vlastní [databáze](http://www.adaptic.cz/znalosti/slovnicek/databaze/) (říkáme, že dokument naindexují). Při vyhledávání je porovnána zadaná fráze s dokumenty v databázi. Vrácený výsledek je závislý na několika faktorech. Složitostí vyhledávacích algoritmů, schopností relevance, scoringu, atd. Existuje celá řada vyhledávacích algoritmů.

Současné fulltextové vyhledávače mají algoritmy natolik propracované, že dokaží sledovat nejen počet slov v textu, ale i jejich hustotu, [sémantiku](http://www.adaptic.cz/znalosti/slovnicek/semantika/) (např. umístění v nadpisech, titulcích…) a mnoho dalších parametrů. Běžná je také schopnost skloňování a časování.

Fulltextové vyhledávání, zkráceně fulltext, je možné si spojit s vyhledávači, prohledávajícími celý internet (např. Vyhledávač společnosti Google) ale také s vyhledávači fungujícími v rámci jedné webové aplikace. Aplikací může být například rozsáhlá dokumentace nebo e-shop, kde fulltextový vyhledávač bývá vyžadován a s roustoucí velikostí aplikace se stává nezbytným, jelikož prosté prohledávání stárnky po stránce k nalzení informace je časově velice náročné.

Internetové fulltexty bývají značně složitější, neboť zpracovávají větší množství dokumentů. Pro představu, společnost Google a jejich produkt Google Search, který je od roku 2009 podle “comScore” (Americká společnost, která poskytuje měření a analýzu médií) nejvíce využívaným vyhledávacím enginem ve spojených státech. Indexuje biliony webových stránek pro umožnění uživatelům získat z nich požadovanou informaci skrze zadání klíčových slov. [ 7 ][ 8 ]

Elasticsearch i další frameworky, které jsou v této práci zmíněny, patří mezi “jednodužší” vyhledávače fungujícími nad výrazně menším objemem dat. Převážně v rámci jedné webové aplikace.

3. Vyhledávací algoritmy

Jednoduše se jedná o algoritmy, jejichž cílem je porovnat námi zadanou frázi se všemi dokumenty a vrátit výsledek, který obsahuje shody. Algoritmů pro fulltextové vyhledávání existuje několik. Od nejjednodužšího naivního algoritmu až po např. daleko propracovanější a složitější fragmentační algoritmus, který používá vyhledávací knihovna Apache Lucene, nad kterou je vystavěna řada dnešních předních vyhledávacích frameworků.

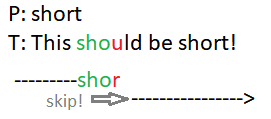
3.1. Brute Force Algoritmus (Naivní algoritmus)

Při tomto způsobu se zkouší zadaná fráze přímo porovnat s dokumentem ve kterém je fráze vyhledávána. Podstatou je cyklus, který běží od začátku dokumentu až do doby, kdy se narazí na konec dokumentu. Cyklus začína na prvním řádku a prvním znaku dokumentu a je porovnávána fráze se vzorem, který tvoří textový řetězec o délce fráze. V průběhu cyklu, pokud nastane shoda, je výsledek zaznamenán a v každém kole cyklu se vzor posune o jeden znak doprava.

Jedná se pravděpodobně o nejjednodužší algoritmus na porovnání textu. V algoritmu zcela chybí jakákoli předpříprava dokumentů, proto je jeho složitost rovna *O(m\*n)*, kde *m* je délka fráze a *n* je délka textu v dokumentu. Při miniaturních projektech dokáže být při rychlostech dnešních počítačů efektivní. Jeho efektivita, ale značně klesá při větším a obsáhlejším množství dokumentů. [ 1 ]

3.2. Boyer-Moore Algoritmus

Tento algoritmus dotváří myšlenku Brute-Forcova Algoritmu a tím zvyšuje jeho efektivitu. Stejně jako naivní algoritmus spočívá v porovnávání textu znak po znaku. Jeho reálná složitost je tedy je stejná jako u Naivního algoritmu *O(m\*n)*, ale díky optimalizaci tato složitost může teoreticky klesat. Pokud nalezne shodu prvního znaku ve frázi s textem, přejde na další znak a takto algoritmus pokračuje až do doby, kdy dorazí na konec fráze (v takovém případě se jedná o shodu) nebo do doby než narazí na neshodu. V takovém případě je porovnán znak v textu (ten který neseděl do sekvence znaků ve frázi) s kompletní frázi a pokud se ve frázi nevyskytuje tak je zřejmé, že další porovnání je možné přeskočit a algoritmus začne běžet opět po daném nevyhovujícím znaku. (viz Obr. 1)

Obr. 1: Boyer-Moore algoritmus

3.2. Rabin-Karp Algoritmus

Algoritmus, který používá hashovací funkci k nalezení shody v textu dokumentu. Hashovací funkce spočívá v převedení každého textového pole na číselnou hodnotu. Rabin-Karp algoritmus pak předpokládá, že jsou-li slova stejná, pak mají i stejnou hash hodnotu.

Fungování tohoto algoritmu tedy spočívá v předpřípravě dokumentu, který je převeden na pole hash hodnot. Při vyhledávací fázi je pak i vyhledávaná fráze převedena na hash hodnotu a porovnávají se pouze tyho numerické hodnoty. Pro text, který má délku *n* a má *p* vzorků společné délky *m* je nejlepší složitost Rabin-Karpova Algoritmu *O(n+m) a n*ejhorší *O(n\*m)*.

3.2. Fragmentační algoritmy

Současné fulltexty využívají daleko důmyslnější prohledávání dokumentů. Ten spočívá v jeho předpřípravě, která např. u Naivního nebo Boyer-Moorova algoritmu zcela absentovala.

Základní kontejner pro data se nazývá index, který lze přirovnat k tradiční relační databázi. Při předpřípravě se dokumenty (objekty) ukládají do databáze ve formátu JSON (dokument je indexován). JSON je zkratka pro JavaScript Object Notation, což je datový formát, se kterým dokáže většina současných nástrojů pracovat. Hlavní výhodou JSONu je jeho jednoduchost. Hodnoty objektů jsou v něm uloženy ve formátu klíč - hodnota, kde hodnoty nemají žádný datový typ.

Samotné hledání shod zadané fráze k vyhledání (=Patternu) s dokumentem pak nemusí probíhat nad celým dokumentem, ale pouze nad jednotlivými částmi JSON objektu (=Fields). Moderní vyhledávače umožňují vyhledávat jak přesné shody, tak dokáží pracovat se skloňováním, časováním pro jednotlivé jazyky, částečnou shodou, regulárními výrazy apod.

Jeden index může potenciálně uchovávat obrovské množství dat, které mohou i překročit kapacitu disku nebo mohou být z důvodu své velikosti přišlis pomalé. Index je proto možné rozdělit do několika částí, tzv. fragmentů (= Shards). Každý fragment je pak považován za vlastní index uvnitř indexu a dokáže plně manipulovat se svými daty stejně, jako by byl pouze jeden. Kromě manipulace s daty ale umí také spolupracovat a zprostředkovat informace ostatním fragmentům.

Tento přístup umožňuje ukládat obsah ukládat na více místech. Také dovoluje distribuovat a paralelizovat operace napříč jednotlivými fragmenty díky čemuž vyhledávač dosahuje vyššího výkonu a tím pádem i rychlosti. [ 2 ]

Moderní vyhledávací nástroje

V předchozích kapitolách byly přiblíženy jednotlivé vyhledávací algoritmy, které se i dnes v mnohých případech stále využívají. Moderní vyhledávací nástroje ale ke svému fungování využívají právě fragmentačního algoritmu a z toho důvodu se práce v dalších kapitolách věnuje pouze tomuto algoritmu.

Následující kapitoly popisují části fulltextového indexu a jednotlivé kroky a postupy při předzpracování dokumentů i následné vyhledávání v indexovaných dokumentech.

Dotazy

Jako na každý dotaz v běžném životě, i na dotaz v kontextu fulltextového vyhledávání je očekávána odpověď. Jakkoli se dokážou vyhledávací algoritmy postarat o vyhledání zadané fráze ve své databázi, tak výsledky nemusí být vždy optimální. Jako ukázka neoptimální odpovědi může vypadat odpověď typu “ANO, danému dotazu odpovídají tyto objekty” nebo “NE, tomuto dotazu neodpovídá žádný z dokumentů”. Naštěstí ale v moderním vyhledávání není nutné se omezit pouze na takto triviální způsob odpovědi. Dnešní vyhledávače jako návratovou hodnotu vrací celé objekty. Obvykle sadu vyhovujících dokumentů u nichž lze ovlivnit v jakém pořadní například dle relevantnosti nalezeného objektu budou seřazeny. Lze využít řadu optimalizačních kroků, které jsou v následujících kapitolách objasněny.

Optimalizace

Práce vychází z předpokladu, že čím výše je správný výsledek ve výsledkové sadě, tím spíše je uživatelem nalezen. Účelem optimalizace je tedy zajištění toho, aby požadovaný výsledek ve výsledkové sadě, kterou vrátí fulltextový vyhledávač, byl co možná nejvýše. V ideálním případě na prvním místě. Toho lze docílit řadou optimalizačních kroků. Fragmentační algoritmy jsou postavené na propracované předpřípravné fázi. Práce s dokumenty je tak rozdělena na Indexovací fázi a Vyhledávací fázi, kde ani v jedné fázi není nutné se omezovat na základní nastavení, ale je možné chování přizpůsobit dané problematice.

Optimalizace Indexovací fáze

V indexovací fázi probíhá proces indexování, který spočívá v uložení (= zaindexování) dokumentů do indexu. Pouze mezi indexovanými dokumenty může být následně vyhledáváno.

Nejpodstatnějším krokem této fáze je tedy vypracování samotného modelu indexu. Do jakých polí budou data ukládána (kam bude uložen obsah stránky, kam doplňující informace, atd.). Model je v ideálním případě třeba připravit pro budoucí rozšíření, které může spočívat například v přípravě na multijazyčnost, přidávání dalších polí, které budou ovlivňvoat relevantnost výsledků apod. Je tedy třeba dobře zvážit nejen požadavky, které apliakce v danou chvíli má, ale i požadavky, které mohou přijít. Každá aplikace je unikátní a není tedy možné definovat přesný, univerzální a zároveň optimální model pro všechny aplikace.

Indexování je, jak bylo zmíněno, proces ukládání dokumentů do databáze ve formátu, který umožňuje jeho efektivní zpětné vyhledávání. Knihovna Apache Lucene indexuje data do speciálního typu databáze, tzv. invertovaného indexu (z anglického překladu inverted index). [ 3 ] Nad invertovaným indexem později probíhá vyhledávání.

Procesu indexování předchází analýza textu, kterou mají na starost analyzátory. Ty mohou být pro každé jednotlivé pole různé. Toho se dá v praxi velmi dobře využívt. Například když je třeba ke každému jednotlivému poli objektu přistupovat různě. V návrhové části práce, kde je navržena také optimalizace analyzátorů, je jejich chování detailněji popsáno.

Další metodou optimalizace již při indexovací fázi může být přidání speciálních polí (obvykle obsahujících číselnou hodnotu), které budou později při vyhledávání použity k rekalkulaci relevantnosti daného výsledku. Danou metodu lze využít také při vyhledávací fázi. V indexovací fázi se této metody ale využívá v případě, že jednotlivé dokumenty mají různou důležitost, kterou by později již nebylo možné určit.

Optimalizace Vyhledávací fáze

Vyhledávání je v kontextu fulltextového vyhledávání proces, kdy se zadaná fráze (případně i další omezující parametry) porovnává s indexovanými dokumenty v invertovaném indexu. Jako výsledek je pak vrácena sada dokumentů u nichž byla nalezena shoda. Výsledková sada má dokumenty seřazené dle jejich relevance.

V dnešní době může hrát i několik málo milisekund výraznou roli. Pakliže vyhledávání trvá příliš dlouho, uživatel může stratit trpělivost a stránku opustit. Přestože vyhledávací nástroje jsou v dnešní době natolik výkonostně optimalizované, že dokáží i složité dotazy nad relativně velkým počtem dat v indexu provést v řádech milisekund, jak je vidět v tabulce (Tab. 2), je potřeba na tuto fázi dávat také pozor.

Ve vyhledávací fázi se formuluje dotaz, na základě kterého je fráze porovnávána s indexovanými dokumenty. A právě to, jakým způsobem je fráze s dokumety porovnávána, má velký vliv jednak na výkon a rychlost poskytnutí výsledků, ale hlavně na požadovanou kvalitu výsledků. Protože ale každý projekt má na kvalitu jiné požadavky a měřítka, nelze obecně určit jaké optimalizační kroky je třeba v daném případě učinit. V návrhové části práce je ale navržena a otestována řada optimalizačních kroků, které lze převzít a přizpůsobit danému projektu.

Při snaze o optimalizaci vyhledávací fáze je třeba zvážit jaké shody jsou pro daný projekt ještě relevantní a jaké už ne. Při malém projektu a malém množství indexovaných objektů nemusí být rozdíl příliš znatelný, ale s jejich roustoucím počtem se proces vyhledávání dokáže výrazně zhoršit.

Regulární výrazy

Regulární výrazy slouží k najití shody na základě definovaného patternu (schéma), které popisuje určité množství textu. Jméno regulární výraz dostal od matematické disciplíny, která je na nich založena. [ 4 ]

Regulární výrazy se dají při fulltextovém vyhledávání použít několika způsoby. Základním využitím regulárního výrazu je vyhledání všech různých řetězců, které odpovídají zadanému patternu. Avšak použití regulárního výrazu při vyhledávací fázi může mít z důvodu velkého množství nutných porovnávání velký vliv na výkon.

Lze je také využít již při indexovací fázi. Konkrétně jako součást analyzátoru, kde je možné regulární výraz použít k vytvoření sady tokenů podle výrazem definovaného pravidla. Tímto způsobem je například možné za pomoci správně definovaného patternu pro *CamelCase* oddělit jednotlivá slova napsaná v dokumentu tímto způsobem.

Je třeba optimalizovat samotný regulární výraz. Špatně definovaný výraz dokáže výrazně zpomalit nebo i shodit celou aplikaci.

Sugesce

Ne vždy uživatel napíše přesně to, co chtěl nebo záměrně nedokončí celý dotaz. Místo toho, aby se jako odpověď vrátila prázdná výsledková sada, je možné zkontrolovat index více detailněji a nalézt podobnosti. Tomuto způsobu se říká sugesce jež se dá také považovat za jeden ze způsobů optimalizace, protože díky ní je možné vrátit i výsledek pro nepřesně napsaný dotaz místo toho, aby musel být dotaz podán znovu. Stejně tak jako ohlídání uživatelských překliků je možné sugesci použít také pro tzv. “autocomplete” nebo “search-as-you-type” sugesci, která vyhledává ještě před tím, než je celý dotaz zformulován. Více o jednotlivých metodách sugesce v návrhové části práce v rámci implementace suggesterů.

Synonyma

Existuje spousta slov, která se dají říci několika různými způsoby. Proto synonyma hrají při fulltextovém vyhledávání významnou roli a lze tímto způsobem velmi dobře optimalizovat.

Fulltextový vyhledávač dokáže jako výsledek vrátit i ty dokumenty, které neobsahují přímo uživatelem zadané slovo ale jen některé z jeho synonym.

Strojové učení

Hlavním cílem strojového učení je navrhnout a vytvořit algoritmy, které umožňují systému používat empirická data, vyvíjet se na základě vlastních zkušeností a připravovat se na změny, které mohou v systému nastat. [ 5 ] Toho se dá využít i v případě fulltextového vyhledávání a tak může být strojové učení velice mocným nástrojem v oblasti optimalizace a analýzy indexu i relevantnosti navrácených výsledků ve výsledkové sadě. Řada současných vyhledávacích technologií strojové učení podporuje a lze ho nějakým způsobem použít a přizpůsobit danému problému. Velikost tohoto tématu ale dalece přesahuje obsah této práce a je zde uvedeno pouze jako další možnost optimalizace.

Technologie fulltextového vyhledávání

Technologií, které umožňují fulltextově vyhledávat je mnoho. Existuje řada technologií, které jsou tzv. “open-source” (otevřený zdrojový kód) a ty jsou z převážné většiny zdarma. Zpoplatněny mohou být služby, které rozšiřují základní funkcionalitu, jako je například Machine learning u technologie Elasticsearch. Jedním z typů open-source projektů jsou technologie, které jsou nadstavbou knihovny Apache Lucene a zpřístupňují tedy její myšlenky a funkcionality v jednodušší podobě než při použití samotné knihovny. Další variantou je zvolit jednu z několika technologií, které jsou “closed-source” a poskytují tzv. “SaaS” (= Search as a Service). Takovou technologií může být například technologie Splunk nebo Algolia. Za hlavní výhodu closed-source technologií se dá považovat jejich nezávislost na Lucene knihovně, která ačkoli přináší skvělé myšlenky a podporu velice rychlého indexování a vyhledávání, nepřináší takový výkon jako novější a modernější technologie, které jsou lépe optimalizované pro dnešní dobu a poskytují indexování a vyhledávání podle [ 11 ] až 200x rychleji než právě Lucene technologie. Za jejich další výhodu se dá považovat pokročilá podpora Machine Learningu, který už nyní začíná mít, a v budoucnu jistě mít bude, obrovské využití.

Ačkoli Splunk, Algolia nebo ostatní novější technologie mohou být v některých ohledech lepším řešením, Lucene technologie dokáží stále poskytnou velice vysoký výkon a podle [ 12 ] je to právě Elasticsearch technologie, která jako nadstavba knihovny Apache Lucene je nejvíce využívanou technologií pro fulltextové vyhledávání. Z tohoto důvodu a z důvodů které vyplývají z tabulky (Tab. 1), kde jsou jednotlivé Lucene technologie porovnány, je i pro projekt, nad kterým je tato práce sepsána, zvolena technologie Elasticsearch.

4. Apache Lucene

Apache lucene je fulltextový open-source engine napsaný v jazyce JAVA, který je přístupný jako knihovna, kterou je možné využít pro vyhledávací účely.

V jakékoli JAVA aplikaci je možnost využití vyhledávání postaveného pouze na této knihovně. Nicméně veškeré nastavení (nastavení clusteru, uzlů, fragmentů, atd.) je potřeba ošetřit. Toto je hlavní nevýhodou a většinou rozhodujícím faktorem, proč si nevybrat pouze tuto knihovnu ve své čisté podobě bez nadstavby, kterou poskytují frameworky vystavěné nad ní. Framework jako např. Elasticsearch má svůj vlastní Zen Discovery modul pro cluster management, naproti tomu Solr používá externí ZooKeeper a díky tomu není nutné se o Cluster management starat vždy od začátku.

Technologie nad Apache Lucene

Technologií postavených na základech Apache Lucene existuje celá řada. Pro porovnání v tabulce (Tab. 1) jsou zvoleny technologie Elasticsearch, Solr a Sphinx. Tyto technologie patří podle [ 12 ] mezi nejpopulárnější technologie v oblasti fulltextového vyhledávání a existují všechny ze stejného důvodu. Tím je zprostředkovat uživateli všechny výhody rychlého až real-time indexování a vyhledávání.

Tab. 1: Porovnání nejvyužívanějších technologií pro fulltextové vyhledávání.

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Framework** | | |
| **ElasticSearch** | **Solr** | **Sphinx** |
| **Komunita a vývojáři** | Společnost Elastic. Open-source projekt. Veřejně na GitHubu. Každý se může podílet na vývoji, ale Elastic tým má poslední slovo | Community over Code = Kdokoli, kdo projeví zájem na vývoji může commitovat do projektu. Solr nemá centrální kontrolní společnost. | Sphinx Developers = Malá skupina vývojářů, především studenti. Open-source projekt. |
| **Poslední release** | 6.1.2, Leden 2018 | 7.2.1, Leden 2018 | 3.0.1, Prosinec 2017 |
| **Napsán v jazyce** | JAVA | JAVA | C++ |
| **Cluster management** | Vlastní produkt Zen - poskytuje úplný node management | Apache Lucene Zookeeper - Vyžaduje externí soubor | Nemá vlastní implementaci, vyžadována externí. Př. BroControl |
| **Cache** | Pro každý segment = Lepší pro dynamicky se měnící data | Globální = Při změně segmentu je třeba změny celé cache | Od verze 2.3.1 |
| **Query optimalizace** | Základní optimalizace + Rychlejší query závislé na čase. | Základní | Základní |
| **Rychlost** | Rychlá - Inverted Index, Rychlejší pro dynamicky se měnící data | Rychlá - Inverted Index, Rychlejší pro statická data. | Rychlá - 7. září 2017 první úspěšné testy s Indexy, od verze 3.0.1 Podpora Indexu. Dříve potřeba, externí databáze pro ukládání a získávání dat. (MySQL) |
| **Featury** | Highlighting, Rescoring, Jedena zjednodušená Suggest implementace (snadnější na použití, možné více využití) | Více suggesterů: Spell Checkers, Highlighting | Autocomplete, Correction Suggestion, Highlighting |
| **Query** | JSON, URL parametry | JSON, XML, URL parametry | JSON, Podpora XML, Query s nastavením vytvořená v některém z podporovaných jazyků |
| **Základní podporované jazyky** | .Net, Clojure, Erlang, Go, Groovy, Haskell, JAVA, JavaScript, Lua, Perl, PHP, Pzthon, Ruby, Scala | .Net, Erlang, Java, JavaScript, Perl, PHP, Pzthon, Ruby, Scala | C++, Java, Perl, PHP, Python, Ruby |

Pro tuto práci byla zvolena technologie Elasticsearch, která podporuje všechny základní funkcionality fulltextového vyhledávání, je stále ve vývoji a navíc tuto technologii využívá mnoho velkých a významných projektů, což je zárukou, že její podpora bude ještě nejakou dobu pokračovat. Mezi nejznámější firmy využívající technologii Elasticsearch patří například Wikipedie, StackOverflow nebo GitHub.

6. Elasticsearch

Elasticsearch je vyhledávací a analytický engine, který umožňuje prohledávat data téměř v reálném čase. Jeho použití je na fulltextové vyhledávání, strukturované vyhledávání, analýzu nebo kteroukoli kombinaci těchto 3 možností. Je vhodný pro velké korporace a obrovské projekty, ale lze ho přizpůsobit i malému domácímu projektu. Stejně tak jako dokáže běžet na jednom notebooku, dokáže běžet na desítkách spolupracujících separátních serverů.

Elasticsearch nepřináší nic nového. Žádná z jeho částí (v open-source variantě zdarma bez rozšíření v podobě například zpoplatněného Machine learningu) není novinkou, jelikož pouze zpřístupňuje funkcionalitu knihovny Apache Lucene, která je pravděpodobně nejvíce rozšířenou a plně podporovatelnou vyhledávací knihovnou dneška. Ale je to jen knihovna. K jejímu využití je potřeba pracovat s jazykem JAVA a integrovat ji přímo do aplikace. Lucene je velice komplexní a složitá na pochopení a správu v případě jejího využití bez nadstavby. ElasticSearch je také napsán v jazyce JAVA a využívá Lucene knihovnu pro veškeré vyhledávání a indexování. Navíc veškerou funkcionalitu zabaluje na standalone server a díky tomu je veškerá funkčnost knihovny Apache lucene daleko přístupnější. Se serverem je možné komunikovat několika způsoby. Skrze Restful API z webového prohlížeče, z vlastní aplikace nebo jednoduše z příkazového řádku.

Každá databáze dokáže selektovat data na základě přesných údajů. Dokáže filtrovat na základě času, třídit, atd. Ale co nedokáže je fulltextové vyhledávání, nahrazování synonymy, hodnocení dokumentů dle jejich relevantnosti, nedokáže agregovat data a generovat analýzu. Toto vše se v případě technologie Elasticsearch děje, navíc téměř v reálném čase.

Elasticsearch je dokumentově orientovaný. To znamená, že jednak ukládá a indexuje celé objekty, ale také celé dokumenty hledá, řadí nebo třídí. Tím se liší od běžných relačních databází, u kterých se to děje jen v rámci sloupcových údajů. Ke správě dokumentů Elastic používá formát JSON, který je jednoduchý a dobře čitelný. Navíc práce s ním je velice rychlá a efektivní.

Cluster management - Zen Discovery

Elasticsearch používá pro správu clusteru a jeho jednotlivých uzlů vlastní vestavěný modul nazvaný Zen discovery. Ten má na starosti volbu hlavního uzlu, detekci chyb a správu globálního stavu.

Ke zjištění stavu a objevování nových uzlů v clusteru používají jednotlivé uzly proces *Ping*. Ten je také základním procesem Zen discovery modulu a na základě něj probíhá komunikace mezi jednotlivými uzly. Definice toho, kam budou uzly “pingovat”, jsou definovány jako list hostů v Unicast discovery modulu. Jedním z úkolů modulu, a také jedním z využití procesu ping je detekce hlavního (“master”) uzlu. Takový uzel musí existovat právě jeden a je automaticky zvolen, pokud již neexistuje. Rozhodnutí o tom, zda se uzel připojí k existujícímu hlavnímu uzlu nebo proběhne volba nového, probíhá na základě 3 *pingů*, které jednotlivé uzly broadcastem odesílají a jejichž timeout je definovatelný. V případě překročení všech je za potřebí zvolit nový master uzel. Pokud ale příjde odpověď před timeoutem, uzel vyšle tzv. *Join request*, díky kterému se připojí k hlavnímu uzlu.

Zen discovery se dále stará o detekci chyb. Za prvé se stará o detekci chyb u master uzlu. Ostatní uzly odesílají *ping* na master uzel, aby zjistily, zda je stále aktivní. Zároveň ale také master uzel odesílá *ping* do ostatních uzlů za účelem zjištění, zda jsou stále aktivní. Timeout a časový interval mezi jednotlivými *pingy* jsou modifikovatelné.

Dalším úkolem Zen discovery je udržování a aktualizace stavu. Jedině master uzel může držet globální stav celého clusteru a stará se o aktualizace. Pouze jedna aktualizace může v jeden čas běžet. Master uzel aplikuje požadované změny a odešle *publish* všem ostatním uzlům. Ty tuto zprávu obdrží a odešlou *acknowledge* zprávu zpět do master uzlu. Rozhodnutí o změně učiní master uzel až v případě příchodu *acknowledge* zprávy od více než definovaného minima uzlů. Poté je aktualizace buď zamítnuta nebo je rozhodnuto o jejím aplikování a každý z uzlů si aktualizuje svůj lokální stav.

Uzel (=Node)

Kdykoli je spuštěna instance Elasticu, je spuštěn jeden uzel. Každý uzel je schopen samostatné existence a v základu dokáže příjmat HTTP a Transportní requesty. HTTP pro komunikaci s restful API a transportní pro komunikaci navzájem mezi uzly a pro příjem requestů z JAVA API (za pomoci JAVA Transport Client). Všechny uzly uvnitř jednoho clusteru musí mít jedinečné jméno, navzájem o sobě vědí a dokáží si mezi sebou přeposílat zprávy. Funkčnost vzájemné komunikace je velice důležitá, protože díky ní je možné zátěž rozdělit mezi několik uzlů. Díky této komunikaci je možné při nedostatku výkonu nebo místa provést tzv. “Horizontální rozšiření” (= Rozdělená zátěž mezi několik uzlů, které je možné provozovat i na více zařízeních) a není nutné pouze radikální tzv. “Vertikální rozšíření” (= Rozšíření ve smyslu nákupu větších, výkonějších zařízení).

Jeden uzel dokáže zároveň sloužit více účelům. Záleží na konfiguraci k jakému účelu je uzel určen. Existují 4 typy uzlů. Je to hlavní uzel, datový uzel, ingest uzel a kmenový uzel.

Cluster obsahuje právě jeden hlavní uzel. Je velice důležité mít v clusteru stabilní hlavní uzel. Ten je zodpovědný za rovnováhu celého clusteru, monitoruje jednotlivé uzly a podle zátěže přiděluje *fragmenty* (“shards”). Kterýkoli uzel může být zvolen jako hlavní během volebního procesu, který je, jak bylo popsáno v kapitole Cluster management, automaticky prováděn k nalezení hlavního uzlu.

Jedním z dalších typů uzlů je datový uzel. Uzel, který je nastaven jako datový, dokáže udržet fragmenty, které obsahují indexovaná data, a provádět nad nimi operace jako jsou CRUD, vyhledávání nebo aggregace. Tyto uzly jsou velice vytěžovány, je potřeba je monitorovat a v případě přetížení rozdělit zátěž mezi více datových uzlů.

V případě, že je potřeba, aby byl dokument před indexací modifikován, použijeme tzv. ingest uzel. Ten definuje pipelinu, která zapříčiní za pomoci definovaných ingest procesorů modifikaci dokumentu (například odebrání nebo právě přidání některého pole). Tyto pipeliny jsou poté uloženy ve stavu clusteru.

Při větší velikosti clusteru je možné připojit tzv. pouze koordinační (“coordination only”) uzel. Takový uzel pak přebírá koordinační roli od datového a hlavního uzlu a tím tak ulevuje jejich zátěži. Po připojení takového uzlu obdrží kompletní stav celého clusteru a distribuuje requesty přímo na místo kam patří.

Cluster

Cluster je balík, který obsahuje jeden nebo více uzlů. Jak jsou uzly přidávány nebo odebírány, cluster se sám dokáže reorganizovat a rovnoměrně rozdělovat data.

Díky tomu, že si cluster uchovává stav, můžeme jednotlivé clustery monitorovat. Sledovat můžeme například počet uzlů, kolik uzlů je datových nebo třeba status, jehož barva dokáže sdělit zda jsou všechny primární a replikované fragmenty aktivní.

Fragment (=Shard)

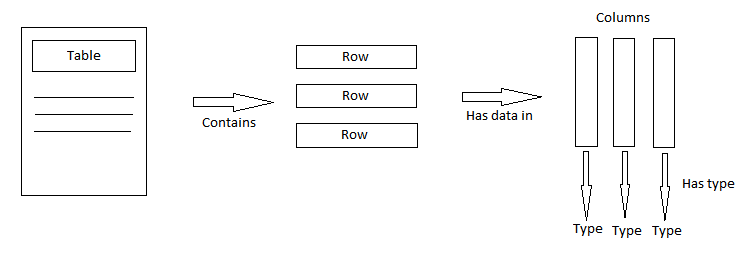
Fragmenty se využívají k rozdělení indexu na menší části. Každá část (každý fragment) je samostatná fungující jednotka (samostatná instance Apache Lucene), která dokáže plně manipulovat se svou částí dat. Dokumenty jsou indexovány do fragmentů, ale aplikace nekomunikuje přímo s nimi. Namísto toho komunikuje s indexem, který se stará o distribuci požadavku do správného fragmentů.

O fragmentech se dá přemýšlet jako o takovém kontejneru pro data. Dokumenty jsou ukládány do fragmentů, kde každý fragment náleží nějakému z uzlu v clusteru. Jak cluster roste, Elastic automaticky migruje fragmenty mezi uzly a tím zůstává cluster v rovnováze. [2]

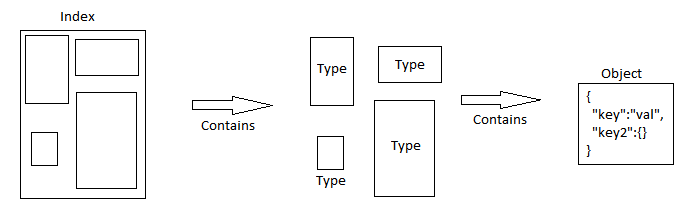
V primárním fragmentu jsou uložena indexovaná data. Počet takových fragmentů je pevně dán při vytváření indexu a později ho již nelze změnit. K tomu, aby nedošlo ke ztrátě indexovaných dat v primárním fragmentu, je možné využít až několika replikovaných fragmentů. Replikované fragmenty existují pro případ, že primární fragmenty nebudou z nějakého důvodu k dispozici. Uzel, ve kterém byli vytvořeny náhle zkolabuje nebo je zkrátka jen offline a není přístupný. Replikované fragmenty pak suplují za primární fragment. Proto je důležité replikované fragmenty mít a mít je zároveň uložené v jiném uzlu. Počet replikovaných fragmentů je možné dynamicky měnit.

Ukládání dat

Tradiční relační databáze se skládá z navzájem provázaných tabulek pomocí primárních a cizích klíčů. Tabulky obsahují data uložená v řádcích a jednotlivá data mají své hodnoty uloženy ve sloupcích, které mají přesně definovaný typ. (Obr. 1)

Obr. 2: Struktura dat v relační databázi

Naproti tomu Elasticsearch se dá spíše přirovnat k některé NoSQL databázi, jako je například MongoDB. Místo databáze má Elasticsearch index, který obsahuje jednotlivé typy. V každém Typu jsou obsaženy Dokumenty, jejichž data jsou rozčleněna do netypových Polí. (Obr. 2)

Obr. 3: Struktura dat ve fulltextovém indexu

Index

Index je kolekce dokumentů nebo kolekce typů, které mají podobnou charakteristiku. Například je možné mít jeden index pro zaměstnance a druhý index pro objednávky. Index je definovaný jménem podle kterého je správný index během průběhu všech procesů (indexace, vyhledávání, atd.) vybírán, proto musí být jméno unikátní.

Typ

V rámci jednoho indexu je možné definovat jeden nebo více typů. Počet typů je naprosto libovolný. Slouží zejména pro přehlednost. Pro představu v indexu zaměstnanci, kde je možné vytvořit typ “Manažer” nebo “Prodejce”.

Dokument

Dokument je základní informační jednotkou pro Elasticsearch. Nese v sobě informace o objektu, který popisuje. Například to může být jeden Zaměstnanec. Dokument, nebo spíše virtuální dokument, jelikož se nemusí jednat o reálný dokument, je uložen ve formátu JSON jako množina polí. Apache Lucene, tedy i Elasticsearch, využívá dokumenty jako objekty, které se indexují a mezi kteými je možné vyhledávat.

Pole

Dokumenty knihovny apache lucene jsou složeny z polí. Pole je obyčejný set, který se skládá z klíče a hodnoty. Všechny hodnoty jsou netypové. Knihovnu nezajímá, zda pracuje s číslem nebo textem. Všechny hodnoty si knihovna převádí na sérii bajtů.

Token

Hodnota pro jednotlivá pole je dále rozdělena na tzv. tokeny. Token si můžeme představit jako například jedno slovo ve větě. Jejich tvorba je nedílnou součástí indexovacího procesu. Při fulltextovém vyhledávání se vyhledává právě mezi tokeny.

Query

Elasticsearch podporuje velmi flexibilní dotazovací jazyk Query DSL, který umožňuje psát i velice komplikované a robustní dotazy, které jsou psány ve formátu JSON.

Uvnitř query jsou napsány jednotlivé dotazy, případně filtry. Elasticsearch vyhodnotí query a vypočítá výsledné score dokumentu, které určuje jak moc je vyhledaný dokument relevantní vzhledem k ostatním indexovaným dokumentům. Dotazy se vykonají a jejich score se vypočítá vždy, když uvnitř napsaného dotazu použijeme parametr *query*.

Naproti tomu *filtr* výsledné score nepočítá. Odpovídá pouze na otázku, zda dokument sedí na položený dotaz či nikoliv. Filtr se použije v případě, že je uvnitř dotazu použit parametr *filter* nebo *must/must\_not* uvnitř parametru *bool*.

Často používané dotazy a filtry si elasticsearch dokáže cachovat a tím tak umožnit pozdější efektivnější a rychlejší použití toho samého dotazu.

Komunikace

To jak probíhá komunikace s elasticsearch serverem záleží na tom, odkud jsou volány dotazy. Komunikace může probíhat dvěma způsoby. Za použití JAVA API přes port 9300 nebo přes restful API s použitím portu 9200.

Java API

Jestliže je aplikace, ve které je za potřebí fulltextové vyhledávání, napsána v jazyce JAVA, pak lze s Elasticem komunikovat přes JAVA API. Elasticsearch při této formě komunikace přichází se dvěma vestavěnými klienty. Jedním je Node Client a druhým Transport Client. Oba klienti komunikují s clusterem prostřednictvím http protokolu na portu 9300. Stejně tak jednotlivé uzly uvnitř clusteru komunikuji přes tento port mezi sebou.

Node client se připojí k lokálnímu clusteru jako bezdatový uzel. Jedná se o tzv. *coordination only* uzel, jehož funkce je popsána v podkapitole Uzel kapitoly Cluster management. Tento uzel neobsahuje žádná data, ale pouze ví, jaká data jsou v jednotlivých uzlech clusteru a posílá dotazy přímo do správných uzlů, kterým dotaz náleží.

Naproti tomu transportní klient je jednodužší (“lehčí”) klient, který může být použit k odesílání requestu na vzdálený cluster. Nepřipojuje se ke clusteru přímo ale jednoduše přeposílá requesty do uzlů v clusteru, jejichž stav si udržuje díky tzv. Sniffing technologii, která spočívá v pravidelných requestech ke zjištění všech datových uzlů a na základě jejích výsledků je aktualizován list těchto datových uzlů. Requesty jsou poté přeposílány přímo těmto datovým uzlům.

Restful API

Druhou možností jak komunikovat s elasticsearch serverem je přes restful API. Odkudkoli je možné s Elasticem komunikovat za použití http protokolu přes port 9200. Je možné posílat requesty z aplikace v kterémkoli jazyce, z webového prohlížeče nebo klidně z příkazového řádku.

Rozšířeni

Elasticsearch kromě řady vestavěných funkcionalit podporuje také několik externích rozšíření, které základní funkcionalitu doplňují o další užitečné funkce. Existuje celá řada nástrojů, které usnadňují nebo vylepšují práci s Elasticem. Díky nim je možné například lépe monitorovat stav serveru, díky čemuž je snadnější detekce chyb nebo omezení, která by mohla snižovat výkon.

X-pack

X-pack je balíček rozšíření pro elastic, který v sobě zahrnuje řadu pluginů, který zbývá pouze nainstalovat. Sám se stará o správu a kompatibilitu jednotlivých pluginů a díky tomu tedy odpadá starost s různými verzemi pluginů apod.

Mezi pluginy můžeme nalézt nástroje starající se o bezpečnost, monitoring, reporting nebo schopnost vizualizace dat pomocí práce s grafy.

Kibana

Kibana je open-source analytická a vizualizační platforma vytvořena pro práci s Elasticem. Dokáže přehledně zprostředkovat data uložená v elasticsearch indexech, provádět nad nimi analýzu, případně je zanášet do grafu, map apod. Díky prostředku Kibana je možné také snadno interagovat s indexovanými daty, dále je k dispozici vizualizace dat posbíraných z pluginů nainstalovaných v balíčku X-Pack.

Logstash

Logstash je open-source datový engine, díky kterému je možné dynamicky, v reálném čase sjednotit data z různých zdrojů a podle požadavků je normalizovat. Díky tomuto nástroji je možné snadno pročistit nebo přizpůsobit data pro různorodé případy užití.

Beats

Beasts je opět open-source projekt, který je nainstalován jako agent na serveru aplikace a zprostředkovává Elasticu různé informace (buď přímo nebo za pomoci Logstashe), dle toho jaký beat máme nakonfigurován. Máme několik různých typů Beatů. Například to mohou být Packetbeat, Filebeat nebo Metricbeat. Packetbeat je síťový analyzátor, který posílá informace o transakcích provedených mezi aplikačními servery. Filebeat posílá log soubory ze serveru. Metricbeat je zase monitorovací agent, který periodicky sbírá data z operačního systému a z procesů běžících na serveru.

**Návrh řešení**

Hlavní otázkou je, jak dosáhnout optimálních výsledků vyhledávání. Podle [ 9 ] je programová nebo také softwarová optimalizace process modifikace systému za účelem učinit systém více efektivní. Efektivitu systému lze chápat z několika úhlů. Optimalizovaný systém je takový systém, který k fungování potřebuje méně zdrojů, systém který pracuje rychleji nebo systém pracující s daty, který vrací například pouze požadované výsledky navíc ve správném pořadí, které určí uživatel nebo systém sám. Optimalizační proces se skládá z několika úrovní. Podle [ 10 ] existuje několik úrovní optimalizace, z nichž minimálně 2 úrovně jsou v kontextu fulltextového vyhledávání podstatné a ty je třeba optimalizovat. Je to úroveň designová a úroveň Algoritmů a datových struktur. Existují i další úrovně optimalizačního procesu. Je možné optimalizovat zdrojový kód aplikace nebo v neposlední řadě optimalizovat hardware pro zajištění efektivního běhu aplikace, ale tyto kroky bývají velice obtížné, velice nákladné až nemožné. Mnohdy navíc mohou být zbytečné. To v případě, špatného designu aplikace nebo špatně navržené datové struktury, která může být právě důvodem horšího výkonu aplikace.

Tuto práci lze brát jako soubor doporučení, které lze obecně aplikovat pro široké spektrum aplikací, které chtějí využívat fulltextového vyhledávání. V následujících příkladech a případech užití jsou veškeré obrázky a ukázky kódu demonstrované pro technologii Elasticsearch. Doporučení lze převzít i pro ostatní technologie. Uvedené ukázky se mohou ale pro odlišné technologie lišit, jelikož každý framework užívá odlišnou syntaxi a také výsledky pro jednotlivé příklady nemusí být zcela přesné, protože každá technologie má implementované jiné algoritmy a modifikátory pro výpočet významnosti vyhledaných dokumentů, pokud tuto hodnotu vůbec počítají.

**Implementace**

Jak již bylo v dřívějších kapitolách zmíněno pro aplikaci sloužící jako podklad pro práci byla zvolena technologie Elasticsearch, která je velice dobře optimalizovaná z hlediska zdorojového kódu co do časové efektivnosti a paměťové náročnosti. (viz Tab. 2) Optimalizace hardwaru, ve smyslu nákupu lepších a výkonějších zařízení, bývá často až poslední a ne vždy nezbytnou možností. Mezi hlavní úkoly vývojáře, který se chystá implementovat fulltextové vyhledávání, je tedy správný designový návrh aplikace, návrh datových struktur a rozhodnutí o správném použití té dané vlastnosti systému, který implementuje.

Tab. 2: Znázornění výkonostní optimalizace. Minimalizace času, minimalizace potřebné paměti

**Představení aplikace**

Aplikace, nad kterou je tato práce sepsána je apliakce uuFulltextovéVyhledávání (uuFulltextSearch).

Je to aplikace vytvořená jako služba, jejíž cílem je umožnit aplikacím pod zadaným URI zaindexovat svůj obsah pro fulltextové vyhledávání a zpřístupnit tak vyhledávání podle vytvořeného indexu.

Služba využívá kromě již zmíněné technologie Elasticsearch určené pro fulltextové vyhledávání také technologii MongoDB, což je NoSQL databáze, sloužící pro zálohování podstatných údajů. Elasticsearch instance je nasazena na 3 uzlech kvůli zajištění stability. Jelikož data z těchto uzlů nejsou zálohovaná, tak při výpadku všech 3 uzlů dojde ke ztrátě dat. Ovšem díky databázi, která drží podstatná data jako jsou uri a další, je možné data v Elasticsearch instanci obnovit.

Další technologií použitou v aplikaci je Sellenium a Chrome driver. Tyto technologie se staraji o vyrenderování stránky definované danou uri na serveru v textové podobě. Odtud si aplikace vezme textová data, která zaindexuje.

**Business procesy**

Mezi základní a klíčové procesy, které aplikace přináší, patří indexování, vyhledávání, monitoring indexu a správa indexu. Procesy jsou vyvolávané buď přímo uživatelem (například přímé odeslání uri k zaindexování) nebo jsou volány systémem samým (například automatické renderování stránky do paměti a následná indexace do fulltextového indexu) jednou za definovaný interval nebo jako reakce na událost.

Proces indexování zajišťuje uložení dat do fulltextového indexu pro definovanou uri. Proces probíhá tak, že na server přide request s několika uri k zaindexování. Aplikace tyto údaje uloží do noSQL databáze a processor, který jedou za definovaný časový interval tuto databázi prohlíží, zda-li neobsahuje dokumenty, které čekají na zaindexování, v případě že nalezne dokument, spustí process renderování stránky a její následné zaindexování.

Proces vyhledávání umožňuje vyhledávání dat uložených pod definovanou uri ve fulltextovém indexu. Opět na server přijde request, který obsahuje údaje, jež mají být vyhledány v indexu. Aplikace složí všechna známá data a vytvoří dotaz, který odešle, za pomoci transportního klienta na server Elasticsearche, odkud se vrátí odpověď. Odpověď obsahuje sadu dokumentů seřazených podle hodnoty *score*, která určuje relevantnost daného dokumentu.

V poslední řadě procesy Monitoring indexu a správa indexu slouží pro zjišťování informací o fulltextovém indexu. Jeho velikosti, limitech, nastavování limitů, apod.

**Business role**

Pro zajištění správného fungování a organizovatelnosti indexu, aplikace dokáže pracovat s různými typy uživatelů (různými rolemi). Kde každá role má nebo naopak nemá právo využívat některou z funkcionalit.

**Klíčové myšlenky**

Mezi klíčové vlastnosti kromě realizace základních business procesů a práce s business rolemi, v případě aplikace uuFulltextovéVyhledávání, patří podpora multijazyčnosti, přímé indexování dokumentů bez jejich renderování a jejich záloha, rescoring dokumentů na základě vlastností dokumentu nebo filtrování výsledků na základě práv uživatele, který vyhledávání inicializuje.

Tyto klíčové myšlenky mohou být důležité konkrétně pro aplikaci uuFulltextovéVyhledávání, ale ne tak moc pro ostatní aplikace. Ovšem každá aplikace může mít jiné specifické myšlenky, proto je třeba dbát na správném designovém návrhu a návrhu správné datové struktury, ve které se budou data do Indexu a do zálohované databáze ukládat. Objekt datové struktury by měl v ideálním případě obsahovat vše podstatné a být snadno udržitelný, modifikovatelný a hlavně dále jednoduše rozšiřitelný.

**Úvod do základního nastavení**

Pro splnění požadavků 2 základních business procesů, mezi které patří indexování dokumentů a fulltextové vyhledávání v nich, což je zároveň i klíčovou myšlenkou celé služby pro fulltextové vyhledávání, postačuje základní návrh datové struktury. Sturktura (Code 1), je strukturou objektu v databázi, která jak již bylo zmíněno, slouží jako držitel podstatných údajů. Údaje jako jsou *index, uri* a *indexInfo*, slouží jednak jako záloha v případě neočekávaného výpadku a spadnutí běžící Elasticsearch instance, ale zároveň slouží také jako mezikrok v procesu objektů k indexaci. Poté co aplikace obdrží tyto objekty, uloží data do mongo databáze se stavem, který indikuje čekání na indexaci do Elasticsearche. Na tento stav dokáže zareagovat procesor, který údaje převezme, nechá stránku, za pomoci technologií k tomu určených, vyrenderovat a její obsah zaindexuje ve formátu, který je vidět na ukázce (Code 2).

Code 1: Struktura objektu v mongo databázi. Základní nastavení.

{

"index": "exampleIndex",

"uri": "http://www.example.com",

"code": "example",

"indexInfo": [

{

"language": "cs",

"state": "IN\_PROGRESS/COMPLETED/FAILED",

}

]

}

Code 2: Struktura fulltextového indexu. Základní nastavení.

{

"\_source": {

"cs": "Lorem ipsum dolor sit amet",

"code": "example",

"uri": "http://www.example.com",

"mts": "2018-04-13T09:21:44.835"

}

}

Objekty při indexaci Elasticsearch ukládá do pojmenovaného indexu. Jak je ze struktury (Code 2) patrné, tak objekt obsahuje několik polí, mezi kterými může být dále fulltextově vyhledáváno. To umožňuje analyzátor textu, který daný text zpracuje, rozdělí na tokeny a uloží do invertovaného indexu. Elasticsearch podporuje širokou škálu již implementovaných analyzátorů s již předdefinovaným chováním. Mezi ty patří, mimo jiné, například jazykové analyzátory pro analýzu textu s ohledem na jazyk, ve kterém je text napsán. Je možné použít pro každé pole jiný analyzátor nebo také odlišný analyzátor pro indexaci a pro vyhledávání, ale pokud není přesně specifikováno, použije se základní analyzátor pro každé pole.

Pro specifikaci analyzátorů pro jednotlivá pole slouží mapovací dokument, který obsahuje souhrn všech polí v daném typu u kterých chceme specifikovat chování při analýze textu. V dokumentu je také možné definovat vlastní pojmenované analyzátory nebo tokenizory, které analyzátory využívají. Vlastní analyzátory je možné definovat v případě potřeby více specifikovat zpracování textu. Základní mapovací dokument pro aplikaci uuFulltextovéVyhledávání je zobrazen na ukázce (Code 3).

Code 3: Základní mapovací dokument pro aplikaci uuFulltextovéVyhledávání

{

"mappings": {

"page": {

"properties": {

"mts": {

"type": "date"

},

"code": {

"type": "keyword"

},

"uri": {

"type": "keyword"

},

"cs": {

"type": "text",

"analyzer": "cs",

},

"en": {

"type": "text",

"analyzer": "en",

}

}

}

}

**Optimalizace**

Do indexu, který je definovaný jak je popsáno v předešlé kapitole, lze data vkládat a lze v něm také vyhledávat. Díky výkonostní optimalizaci technologie Elasticsearch se navíc vše děje velice rychle, jak je vidět v tabulce (Tab. 2) a téměř není prostor pro optimalizaci v tomto směru.

Hlavním cílem pro optimalizování je přispůsobení výsledků, které Elasticsearch vrací jako odpověď na položený dotaz.

Výsledek, který Elasticsearch vrací, je sada objektů, kde každý objekt nějakým způsobem vyhovuje položenému dotazu. Jak moc je daný objekt relevantní specifikuje hodnota *score*, což je hodnota, kterou má každý objekt přiřazen a podle které je výsledné pole objektů seřazeno. Algoritmy pro výpočet score jsou již v rámci Elasticsearch technologie implementovány. Existuje ale celá řada možností jak výsledné score modifikovat.

Práce vychází z předpokladu, že čím výše je správný výsledek ve výsledkové sadě, tím spíše je uživatelem nalezen. Protože jsou výsledky ve výsledkové sadě technologie Elasticsearch řazeny podle hodnoty score je hlavním optimalizačním cílem modifikace této hodnoty takovým způsobem, aby reflektovala relevantnost daného objektu ve výsledkové sadě.

V dalších kapitolách jsou popsány jednotlivé optimalizační kroky, které se v aplikaci uuFulltextovéVyhledávání osvědčili. Ty jsou doložené v tabulce “” a je možné je převzít jako doporučení pro optimalizaci výsledků širokého spektra aplikací.

**Optimalizace vyhledávacího dotazu**

K dotazování na data uložená v Elasticsearch indexech se používá flexibilní jazyk Query DSL, díky kterému je možné psát i velice komplikované a robustní složené dotazy, které jsou psané ve formátu JSON. Query DSL obsahuje 2 typy klauzulí. Klauzule, jež hledají shody nebo částečné shody v jednotlivých polích. Mezi takové query se řadí například “match” nebo “term” query. Dalším typem klauzule jsou klazule složených dotazů. Ty se dají použít k logické kombinaci více dotazů. Řadí se mezi ně například “bool” query, která rozhoduje zda objekt z indexu odpovídá dotazu, nebo query jež ovlivňují chování ostatních dotazů, jako je například “functionScoreQuery”.

Některé dotazy jsou důležitější než ostatní, proto lze každému dotazu přiřadit jeho váhu a tím do značné míry ovlivnit výsledné *score* objektů z výsledkové sady.

**Popisek stránky**

V případě vyhledávání v rámci jedné aplikace může nastat situace, kdy uživatel přesně ví co na stránce hledá, ale v kontextovém menu to není na první pohled zřejmé. V takovém případě je pravděpodobně do vyhledávače zadáno klíčové slovo, které definuje přesně tuto stránku.

Výsledky podle předešlého návrhu indexu (Code 2) mohou být ale zavádějící, jelikož dané klíčové slovo se na různých stránkách může také vyskytovat, mnohdy v hojnějším počtu. Z tohoto důvodu ve výsledkové sadě stránka, kterou uživatel doopravdy hledal nemusí být v popředí. Je dobré proto k indexovanému objektu přiřadit i pole, jehož obsah definuje co je v daném objektu obsaženo.

V případě aplikace uuFulltextovéVyhledávání byl, jako pole shrnující obsah stránky, zvolen popisek (label) s dodatkem jazyku, pro který je ten daný popisek. Objektové struktury v mongo databázi i Elasticsearch indexu se o dané pole rozšířily jak je vidět na ukázkách datových struktur mongo databáze (Code 4) a elasticsearch indexu (Code 5).

Code 4: Struktura objektu v mongo databázi. Přidání pole "label"

{

"index": "exampleIndex",

"uri": "http://www.example.com",

"code": "example",

"indexInfo": [

{

"language": "cs",

"state": "IN\_PROGRESS/COMPLETED/FAILED",

"label": "Lorem"

}

]

}

Code 5: Struktura fulltextového indexu. Přidání pole "label"

{

"\_source": {

"cs": "Lorem ipsum dolor sit amet",

"label\_cs": "Lorem",

"code": "example",

"uri": "http://www.example.com",

"mts": "2018-04-13T09:21:44.835"

}

}

Elasticsearch podporuje tzv. MultiMatchQuery, díky které je možné vyhledávat přes více polí a díky tomu dokáže zobrazit i objekty, které v případě obyčejného dotazu vůbec zobrazené nebyly nebo je zobrazí s odlišnou vážností.

Samotná relevance se může i výrazně lišit v závislosi na specifičnosti daného popisného pole. Čím specifičtější pole je (čím obsahuje méně klíčových slov) tím relevantnější daný objekt bude ve výsledkové sadě. Odlišnost relevance v závislostech na jednotlivých popiscích je vidět v tabulce (Tab. 3), ze které také vyplývá, že score je do značné míry ovlivněno celkovým počtem dokumentů v indexu, proto je potřeba dané dodatečné pole dále modifikovat a získat tak lepší kontorlu nad výsledkovou sadou. Více je popsáno v kapitole Optimalizace vyhledávacího dotazu.

Tab. 3: Relevantnost výsledků v závislosti na specifičnosti popisku a celkového počtu dokumentů.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | Výsledek | | | | | |
| Počet dokumentů | Popisek | Počet stránek obsahujících popisek | Score | Počet stránek obsahujících popisek | Score | Počet stránek obsahujících popisek | Score |
| 10 | "Lorem" | 1 | 2.356 | 2 | 1.812 | 1 | 1.816 |
| "Lorem Ipsum" | 1 | 2.154 | 1 | 1.646 | 2 | 1.656 |
| "Example Label" | 8 | 0.046 | 7 | 0.046 | 7 | 0.046 |
| 1000 | "Lorem" | 1 | 9.484 | 2 | 8.951 | 1 | 8.951 |
| "Lorem Ipsum" | 1 | 8.613 | 1 | 8.129 | 2 | 8.129 |
| "Example Label" | 998 | 4.90E-04 | 997 | 4.90E-04 | 997 | 4.90E-04 |

**Boost hodnota**

V situaci, kdy je potřeba objekty ve výsledkové sadě vždy upřednostnit před ostatními, lze jeho *score* dále přepočítat na základě definované hodnoty. Takovou hodnotou může být například popularita (inkrementována při každém výběru daného objektu z výběrové sady) nebo statická hodnota přiřazená objektu, která říká, že daný objekt je důležitější než ostatní a ve výsledkové sadě to musí být uváženo.

Pro případ modifikace *score* na základě číselné hodnoty zaindexované v poli objektu, Elasticsearch přináší tzv. FunctionScoreQuery. FunctionScoreQuery dokáže modifikovat *score* různýmy funkcemi. Každá funkce přijímá výsledkovou sadu z předchozí query a každému objektu z ní dokáže různýmy způsoby upravit *score.* Jedním ze způsobů je například tzv. “field\_value\_factor”, který převezme indexovanou hodnotu daného pole a tou podle definice upraví původní score. Je možné definovat jak velkou váhu bude mít hodnota v daném poli vzhledem k velikosti hodnoty. Elasticsearch přináší více způsobů modifikace score vedle již zmíněného “field\_value\_factor”. Dá se použít vlastní funkce, náhodné číslo nebo složitější funkce rozkladu jako jsou například gaussova funkce nebo lineární funkce. Tyto funkce hodnotí dokumenty v závislosti na vzdálenosti hodnoty od původně uživatelem zadané hodnoty a hodí se například při udělování relevance objektům vzhledem ke vzdálenosti od uživatelem zadaného místa.

Funkce s modifikátorem typu “field\_value\_factor” je pro optimalizaci výsledkové sady, vzhledem k zaindexovaným hodnotám v polích “boost” a “popularity”, použita i v aplikaci uuFulltextovéVyhledávání. Hodnota “boost” obsahuje statickou hodnotu, která určuje důležitost objektu vzhledem k ostatním a hodnota v poli “popularity” říká, kolikrát byl daný objekt z výsledkové sady vybrán. Struktura objektu v mongo databázi po přidání polí “boost” a “popularity” je zobrazena na ukázce (Code 6), pro fulltextový index (Code 7). Jako způsob výpočtu váhy hodnoty “field\_value\_factor” modifikátoru je v případě pole “popularity” použita funkce Log1P. Díky této funkci, jak vyplývá z tabulky (Tab. 4), má růst grafu výsledného score logaritmický průběh. To zmanená, že změny při nižších hodnotách v poli “popularity” mají na výsledné score dokumentu větší vliv než při hodnotách vyšších. Další možností pro logaritmický průběh nárůstu výsledného score je použití Ln modifikátoru hodnoty. Tento způsob má však větší dopad na výsledné score, než je v případě aplikace uuFulltextovéVyhledávání žádoucí. Pole “boost” přímo určuje důležitost daného objektu, proto není žádoucí, aby váha této hodnoty byla nějakým způsobem upravována a žádná funkce není tedy modifikátoru typu “field\_value\_factor” přiřazena.

Code 6: Struktura objektu v mongo databázi. Přidání polí "boost" a "popularity"

{

"index": "exampleIndex",

"uri": "http://www.example.com",

"code": "example",

"boost": 1.0,

"popularity": 1.0,

"indexInfo": [

{

"language": "cs",

"state": "IN\_PROGRESS/COMPLETED/FAILED",

"label": "Lorem"

}

]

}

Code 7: Struktura fulltextového indexu. Přidání pole "boost" a "popularity".

{

"\_source": {

"cs": "Lorem ipsum dolor sit amet",

"label\_cs": "Lorem",

"code": "example",

"uri": "http://www.example.com",

"mts": "2018-04-13T09:21:44.835",

"boost": 1.0,

"popularity": 1.0

}

}

Tab. 4: Růst score podle hodnoty. Modifikátory nárůstu score.

|  |  |  |  |
| --- | --- | --- | --- |
| Hodnota | Modifikátor hodnoty | | |
| Žádný | Ln | Log1P |
|  |  |  |  |
| 1 | 1.052 | 0.789 | 1.052 |
| 5 | 2.761 | 1.424 | 1.096 |
| 10 | 4.734 | 1.697 | 1.199 |
| 50 | 20.515 | 2.332 | 1.46 |
| 100 | 40.242 | 2.605 | 1.579 |
| 1000 | 395.32 | 3.514 | 1.972 |

**Rescore**

Díky tomu, že Elasticsearch používá dotazovací jazyk Query DSL, který podporuje skládání dotazů, existuje celá řada možností díky kterým je možné kromě váhy jednotlivých dotazů, výsledné *score* modifikovat. Pro dosažení, pro danou aplikaci, opravdu optimálních výsledků je možné zřetězit řadu dotazů uvnitř tzv. “rescore” query. Ta obsahuje pole dalších, obvykle více specifičtějších dotazů, jako je například “phrase” query, která vyhledává přímo zadanou frázi bez možnosti rozdělení fráze na jednotlivá slova nebo “phrase” query s definovaným “slop” atributem, jež získává na vážnosti podle toho, jak jsou dané tokeny z vyhledávané fráze v textu od sebe navzájem vzdálené.

Tato optimalizační metoda nemá vliv na počet nalezených objektů ve výsledkové sadě, ale dokáže přepočítat přiřazené score objektů vrácených jako odpověď na prvotní dotaz a tím tak dosáhnout požadovaného setřídění dané výsledkové sedy.

**Analýza textu**

Analýza textu je z pohledu optimalizace podstatným krokem. Reálné dokumenty se stanou dohledatelnými po jejich zaindexování, což je proces ukládání dat do invertovaného indexu, kterému předchází právě proces analýzy.

Veškerá snaha o optimalizaci relevantnosti objektů takovým způsobem, aby byl hledaný výsledek co možná nejvýše ve výsledkové sadě, by mohla být zbytečná, pokud se hledané objekty do výsledkové sady ani nedostanou.

Hlavním myšlenkou optimalizace fáze analýzy je tedy zajištění zpracování textu takovým způsobem, aby byl invertovaný index následně naplněn co možná nejobecnějšími tokeny.

**Optimalizace analyzátorů**

Analyzátor je objekt, který je přiřazen každému jednotlivému poli dokumentu (je použit defaultní, přenastavený analyzátor, pokud v mapovacím dokumentu indexu není definováno jinak) a obsahuje v sobě definici procesu, který se stará o předpřípravu textu.

Ta spočívá v rozdělení daného textu na pole tokenů. Pole tokenů je dále možné na základě definovaných filtrů upraveno tak, aby obshaovalo tokeny pouze v takovém tvaru, díky kterému budou co nejsnáze dohledatelnými. Ideálně by měly být v základním tvaru a bez interpunkce (lze použít matematické algoritmy nebo slovníková pravidla pro hledání kořenů slova) nebo navíc ještě rozděleny po částech, tzv. “n-gramech”. To umožňuje nalezení slova i v případě, že je zadána jen jeho část. Pole tokenů by mělo dále obsahovat co možná nejméně tzv. “stopwords”, což jsou slova pro daný jazyk hojně se vyskytující a pro vyhledávání nepodstatná. Může se jednat například v českém jazyce o předložky, spojky nebo v angličtině “the”, “a”, apod. Tato slova by mohla mít z důvodu své četnosti jednak vliv na výkon, ale zároveň by mohla mít podstatný vliv na relevantnost výsledků.

Jak již bylo zmíněno, pro každé pole je v základním nastavení použit defaultní analyzátor, který ve svém nastavení používá standartní tokenizer, který dělí text na tokeny pokaždé když narazí na mezeru, pomlčku, podtržítko nebo jiný speciální znak. Na pole tokenů je aplikován pouze filtr, který všechny tokeny převede na malá písmena.

Optimalizace analýzy textu tedy spočívá v přenastavení základního chování analyzátorů, jejich tokenizerů a použití a posloupnosti jednotlivých filtrů.

**Optimalizace tokenizerů**

Tokenizer je objekt, který předepisuje chování analyzátoru při prvotním zpracování textu a jeho rozdělení na pole tokenů, se kterým je dále pracováno. Jeho optimalizace spočívá v zajištění správné identifikace jednotlivých tokenů, čehož lze dosáhnout několika způsoby. Lze vybrat z řady již předdefinovaných tokenizerů nebo lze vytvořit tokenizer s vlastním chováním a využít tak různé metody pro rozdělení textu na tokeny. Například je možné využít regulárních výrazů pro rozdělení frází psaných ve formátu “camelCase”.

Další metodou jak optimalizovat pomocí tokenizerů je pomocí rozdělení tokenů po částech, tzv. “n-gramech”, čehož se využívá pro odhad kompletní fráze po jejím neúplném zadání.

Po použití n-gramu s minimální velikostí tokenu 3 a maximální velikostí 5 se do pole tokenů pro frázi například “index” uloží tokeny “ind”, “inde”, “index”. Po následném vyhledání nekompletní fráze “ind” index dokáže zareagovat a vrátit tak shodu s dokumentem, který v textu obsahuje “index” nebo jiné fráze začínající na danou posloupnost znaků.

**Optimalizace filtrů**

Analyzátor může obsahovat pole filtrů. Na všechny tokeny, vytvořené pomocí tokenizoru, jsou aplikována pravidla v posloupnosti jak jsou jednotlivé filtry uvedeny. Hlavním účelem filtrů je upravit jednotlivé tokeny na co možná nejobecnější tvar a umožnit tak jejich vyhledání i po zadání nepřesné fárze.

Hlavním optimalizačním cílem je vybrat správné token filtry a z důvodu zajištění optimálního chování a výkonu dané filtry využít ve správném pořadí.

Na aplikaci uuFulltextovéVyhledávání byly otestovány filtry v pořadích jež jsou vidět na ukázkách (Code 8) a (Code 9), kde výsledky pro indexování a vyhledávání pro jednotlivé analyzátory jsou zaneseny do tbaulky (Tab. 5), která znázorňuje čas, za který byly stejné dotazy zpracovány, a počet výsledků, které byly navráceny. Pro daný příklad bylo zaindexováno 37 stránek reálné aplikace a následné vyhledání frází podle kategorií.

Z výsledků jsou patrné nepatrné odlišnosti ve výkonu ve prospěch algoritmického zpracování. Hlavním rozdílem jsou ale navrácené výsledky, kde po vyhledání netypické fráze “zaindexování” byl v případě slovníkového zpracování navrácen pouze jeden výsledek. To je z důvodu, že český slovník slovo “zaindexovat” nezná a není tedy možné ho převést na základní tvar. Proto není možné nalézt v inverovaném indexu i slova “zaindexovat”, “zaindexováno”, apod.

V případě analyzátoru, používajícím algoritmické zpracování, jsou postupně použity filtry “lowercase”, který všechny tokeny převede na malá písmena. Filtr “czech\_stop”, který má za cíl z pole tokenů odstranit všechna česká “stopwords”, což jsou slova, která jsou pro daný jazyk nepodstatná a není třeba mezi nimi vyhledávat nebo je dále zpracovávat. Pro hledání základního tvaru slova je použit obecný “czech\_stemmer” token filtr, který využívá proces stemantizace s algoritmem stemmer. Ten se využívá pro nalezení kmene slova. Obvykle ale kmen slova není správným kmenem, proto se ve výsledkové sadě jako odpověď na dotaz mohou objevit i shody, které nebyly hledány. Stemantizace ale narozdíl od Lemantizace (nalezení základního tvaru slova podle slovníkem daných pravidel pro daný jazyk) zpracuje i slova, která nejsou gramaticky správně nebo v daném jazyce neexistují. Dalším použitým filtrem je “asciifolding”, což je Elasticsearchem předdefinovaný filtr a má za cíl převést písmenné, číselné nebo speciální znaky, které nejsou mezi prvním 127 znaky ascii tabulky, na jejich ASCII equivalent. V poslední řadě je použit filtr “unique\_on\_same\_position” jehož cílem je odstranit duplicitní tokeny, které během analýzy textu vzniknou na stjené pozici.

Code 8: Algoritmické hledání kmenů slov

"custom\_czech\_stemmer\_analyzer": {

"type": "custom",

"tokenizer": "standard"

"filter": [

"lowercase",

"czech\_stop",

"czech\_stemmer",

"asciifolding",

"unique\_on\_the\_same\_position"

]

}

V případě použití analyzátoru s definicí podle (Code 9), je použit “czech\_hunspell” token filtr. Czech\_hunspell je označení pro filtr, který k nalezení základního tvaru slova využívá slovník všech slov daného jazyka a definici pravidel pro jejich skloňování a časování. Při využití tohoto způsobu je zajištěna gramatická správnost a přesnost vyhledávání. Nezpracují se ale slova, jež nejsou definována ve slovníku. V poli filtrů je definován dvakrát “czech\_stop” filtr. To je z důvodu, že “stopwords” daného jazyka nemusí být definovány ve všech tvarech, proto se v prvním pokusu odstraní “stopwords”, které již v základním tvaru jsou, poté jsou tokeny za pomoci “czech\_hunspell” filtru převedeny na základní tvar a následně jsou tokeny opět promazány. Místo “asciifolding” filtru je v prípadě daného analyzátoru použit filtr “icu\_folding”, který funguje na podobném principu jako “asciifolding” filtr, ale zahrnuje také specifická pravidla pro daný jazyk. Pro češtinu například rozpozná, že sekvence znaků “c” a “h” znamená “ch” a díky tomu dokáže například výsledky správně setřídit.

Code 9: Slovníkové hledání základního tvaru slov.

"custom\_czech\_hunspell\_analyzer": {

"type": "custom",

"tokenizer": "standard"

"filter": [

"lowercase",

"czech\_stop",

"czech\_hunspell",

"czech\_stop",

"icu\_folding",

"unique\_on\_the\_same\_position"

]

}

Tab. 5: Porovnání testovaných analyzátorů.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Typ fráze** | **Fráze** | **Stemmer (Code 8) analyzer** | | **Hunspell (Code 9) Analyzer** | |
| **Čas** | **Nalezené shody** | **Čas** | **Nalezené shody** |
| **Popisek** | Obsah | 92 | 37 | 12 | 37 |
| **Titulky na více stárnkách** | Vychozi hodnoty | 65 | 19 | 16 | 19 |
| **Často používané slovo** | Nazev | 88 | 22 | 84 | 22 |
| **Několik používaných slov** | nazev hodnoty informace atribut init workspace inicializace model business | 85 | 37 | 15 | 37 |
| **Netypické slovo** | Zaindexování | 15 | 1 | 9 | 4 |

Zdroje:

**[ 1** **]** JANANI, R.; VIJAYARANI, S. An Efficient Text Pattern Matching Algorithm for Retrieving Information from Desktop. Indian Journal of Science and Technology, 2016, 9.43.

**[ 2 ]** Clinton Gormley and Zachary Tony (O’Reilly). Elasticsearch: The Definitive Guide. Copyright 2015 Elasticsearch BV, ISBN: 978-1-449-35854-9.

**[ 3 ]**BALIPA, Mamatha; BALASUBRAMANI, R. Search Engine using Apache Lucene. International Journal of Computer Applications, 2015, 127.9: 27-30.

**[ 4 ]** Goyvaerts Jan, 8.11.2017 [citováno dne 14.11.2017]. [on-line zdroj] Dostupné z: http://www.regular-expressions.info/tutorial.html

**[ 5 ]** KAPITANOVA, Krasimira; SON, Sang H. Machine learning basics. Intelligent Sensor Networks: The Integration of Sensor Networks, Signal Processing and Machine Learning, 2012, 13.

**[ 6 ]** HUBERMAN, Bernardo A.; ADAMIC, Lada A. Internet: growth dynamics of the world-wide web. Nature, 1999, 401.6749: 131.

**[ 7 ]** [comScore Releases November 2009 U.S. Search Engine Rankings](http://www.comscore.com/Press_Events/Press_Releases/2009/12/comScore_Releases_November_2009_U.S._Search_Engine_Rankings). December 16, 2006. Retrieved July 5, 2010.

**[ 8 ]** ARLINGTON, M. Google’s misleading blog post: the size of the web and the size of their index are very different. Available on http://techcrunch. com, 2008.

**[ 9 ]** SEDGEWICK, Robert. Algorithms. Series in Computer Science. 1984.

**[ 10 ]** PEYMANDOUST, Armita; SIMUNIC, Tajana; DE MICHELI, Giovanni. Low power embedded software optimization using symbolic algebra. In: Proceedings of the conference on Design, automation and test in Europe. IEEE Computer Society, 2002. p. 1052.

**[ 11 ]** Algolia technical documentation. 2016. Available on [https://www.algolia.com/doc].

**[ 12 ]** Database-Engines Ranking. 2018. Available on [https://db-engines.com/en/ranking/search+engine].