Anotace

Tato práce se zabývá problematikou optimalizace fulltextového vyhledávání pro případ vyhledávání v dokumentaci a kódu. V textu je, mimo jiné, také popsáno, jak se fulltextově vyhledávalo v minulosti a jaké přístupy se využívají dnes. V co nejsrozumitelnější podobě jsou zde popsány jednotlivé frameworky, které se v současnosti využívají. Více je pak specifikována a rozebrána knihovna Apache Lucene, kterou jako svůj základ tyto frameworky využívají. V hlavní části se tato práce věnuje obecnému nasazení technologie na produkční prostředí a jaké způsoby lze využít pro optimalizaci vyhledávání v dané oblasti. Na základě obsahu této práce by měl být každý, kdo se chce informovat o současných technologiích v oblasti fulltextového vyhledávání nebo si neví rady s výběrem té správné technologie, schopen pro svůj současný nebo budoucí projekt vybrat tu nejlepší technologii a měl by být schopen ji implementovat a optimalizovat pro svůj projekt. Pro mé účely se ale nejvíce hodil framework Elasticsearch, který bude v práci mnohokrát zmiňován a spousta věcí je vysvětlována právě na něm.

**Obsah**

Úvod do současného světa internetu

Co je to fulltextové vyhledávání a k čemu slouží

Vyhledávací algoritmy

Brute Force Algoritmus (Naivní algoritmus)

Boyer-Moor Algoritmus

Rabin-Karp Algoritmus

Fragmentacni Algoritmus

Dotazy

Optimalizace

Indexovací fáze

Vyhledávací fáze

Regulární výrazy

Sugesce

Synonyma

Strojové učení

Technologie fulltextového vyhledávání

Apache Lucene

Výběr technologie

ElasticSearch

Cluster mamagement - Zen Discovery

Uzel

Cluster

Fragment

Ukládání dat

Index

Typ

Dokument

Pole

Token

Dotazování

Komuniakce

JAVA API

Restful API

Rozsireni

X-pack

Kibana

Logstash

Beats

TBD = Implementace a nasazení

1. Současný internet a proč je důležité v něm vyhledávat

Svět se topí v datech. Na internetu každou sekundu přibyde nepřeberné množství (okolo 5 milionů) nových stránek a tato rychlost se neustále zvyšuje. Není v lidských silách se v těchto datech vyznat, umět v nich vyhledat požadovanou informaci. Proto vznikly a k tomuto účelu nám slouží full-textové vyhledávače.

Proces vyhledávání je na první pohled možná takovou černou skříňkou, která vykoná cosi na pozadí, co nám je skryto a vrátí nám požadovaný výsledek. Účelem této práce je vysvětlit, že to není jen magie, jak se může na první pohled zdát, ale spíše důmyslný a komplexní systém procesu, který je zodpovědný za odpověď na náš dotaz.

2. Fulltextové vyhledávání

Fulltextové vyhledávání nebo zkráceně fulltext je obecné označení pro vyhledávače fungující na základě porovnávání fráze se všemi ostatními slovy v daném dokumentu. Fulltext je tedy 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 porovnán námi zadaný termín s dokumenty v databázi a nám se vrátí požadovaný výsledek. Ten je závislý hned na několika faktorech. Složitostí vyhledávacích algoritmů, schopností relevance, scoringu, atd. Existuje celá řada vyhledávacích algoritmů.

Současné fulltexty mívají 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. Běžné je také schopnost skloňování a časování.

Slovo fulltext si můžeme spojit s vyhledávači, které prohledávají celý internet (např. Google) ale také s vyhledávači fungujícími v rámci jedné webové aplikace, kde se z fulltextu stává neocenitelný pomocník např. v případě, že naší webovou aplikací je rozsáhlá dokumentace a prosté prohledávání stránky po stránce je časově velice náročné až nemožné.

Internetové fulltexty bývají značně složitější, neboť zpracovávají větší množství dokumentů. Pro představu společnost Google na prohledání celého internetu využívá přes 300 000 webových serverů, které neustále pracují a shromažďují data. Odpovědi na naše dotazy, které se chystáme položit v nejbližších dnech má tedy Google uložené ještě dříve než se na ně vůbec zeptáme.

Elasticsearch i další frameworky, které jsou v této práci zmíněny, patří mezi “jednodužší” vyhledávače fungujícím 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 fulltext známe několik. Od nejjednodužšího naivního algoritmu až po např. daleko propracovanější a složitější fragmentační algoritmus, který používá vyhledávací knihovna Apache Lucene, nad kterou je vystavěna většina dnešních předních vyhledávacích frameworků.

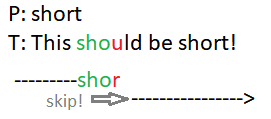
3.1. Brute Force Algoritmus (Naivní algoritmus)

Při tomto způsobu zadanou frází zkoušíme přímo porovnat s dokumentem ve kterém hledáme. Pokud jsme našli shodu vrátíme výsledek. Pokud ne, posuneme se o znak doprava a celý proces opakujeme do doby než narazíme na konec dokumentu.

Jedná se pravděpodobně o nejjednodužší algoritmus na porovnání textu. V algoritmu úplně chybí jakákoli předpříprava dokumentů, tak je jeho složitost z pohledu výpočetní náročnisti je velice vysoká. 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 B-F spočívá v porovnávání textu znak po znaku. Pokud nalezne shodu prvního characteru v patternu s textem, přejde na další character a takto algoritmus pokračuje až do doby, kdy dorazí na konec patternu (v takovém případě se jedná o shodu) nebo do doby než narazí na neshodu. V takovém případě porovnáme character v textu (ten který neseděl do sekvence characterů v patternu) s kompletním patternem a pokud se v patternu nevyskytuje tak víme, že další porovnání můžeme přeskočit a algoritmus začne běžet opět po daném nevyhovujícím characteru.



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.

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, u Rabin-Karp Algoritmu sice předpříprava probíhá, ale ne příliš propracovaná.

Základní kontejner pro data se nazývá index, který bychom mohli přirovnat k tradiční relační databázi. Při předpřípravě se dokumenty (objekty) ukládají do databáze ve formátu JSON (říkáme, že dokument indexujeme). 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 námi požadovaného textu 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 i například 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 fragmentů (= Shards). Každý fragment je pak považován za takový vlastní index uvnitř indexu, dokáže plně manipulovat se svými daty stejně jako by byl pouze jeden.

Tento přístup nám dovoluje rozšiřitelnost obsahu, můžeme obsah ukládat na více místech. Také nám dovoluje distribuovat a paralelizovat operace napříč jednotlivými fragmenty díky čemuž dosáhneme vyššího výkonu a tím pádem i rychlosti. [2]

Dotazy

V předchozích kapitolách byly přiblíženy jednotlivé vyhledávací algoritmy, které se i dnes v mnohých případech stále využívají. Moderní vyhledávací nástroje (Elasticsearch nevyjímaje) ale ke svému fungování využívají právě fragmentačního algoritmu a z toho důvodu se dále budeme zabývat pouze tímto algoritmem a veškeré následující kapitoly budou popisovat kroky, doporučení a postupy jednotlivých částí samotného indexu, předzpracování dokumentu i následné vyhledávání v indexovaných dokumentech.

Jako na každý dotaz v běžném životě, tak i na dotaz v kontextu fulltextového vyhledávání očekáváme odpověď. Jakkoli se dokážou vyhledávací algoritmy postarat o vyhledání patternu 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 se ale v moderním vyhledávání nemusíme omezit pouze na takto triviální způsob odpovědi, ale to jaké výsledky nebo v jakém pořadní dle relevantnosti vyhledaného objektu nám dané výsledky budou navráceny, můžeme sami ovlivnit. Lze využít řadu optimalizačních kroků, které se pokusím v následujících kapitolách objasnit.

Optimalizace

Jak již bylo zmíněno, tak to jak nám bude na dotaz odpovězeno, můžeme do značné míry ovlivnit 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 se nemusíme omezovat na základní nastavení, ale můžeme chování přizpůsobit našemu problému.

Optimalizace Indexovací fáze

V indexovací fázi probíhá proces indexování, který spočívá v uložení (= Zaindexování) dokumentu do Indexu. Pouze indexované dokumenty mohou být následně vyhledány.

Nejpodstatnějším krokem této fáze je tedy vypracování samotného modelu našeho indexu. Do jakých fieldů budou data ukládána (kam uložíme content stránky, kam doplňující informace) nebo v jakém formátu mají být data pro daný field. Model je v ideálním případě třeba připravit na rozšiřitelnost, protože málokterá dnešní aplikace neočekává rozvoj. (Rozšiřitelnost může spočívat např. V přípravě na multijazyčnosti, přidávání různých dalších fieldů, které budou ovlivňvoat relevantnost výsledků apod.)

Indexování je proces ukládání dokumentů (jeho jednotlivých polí složených z textových dat) do databáze ve formátu, který umožňuje jeho efektivní zpětně vyhledávání. Lucene ukládá data ve formátu, kterému se říká invertovaný index (z anglického překladu Inverted index).[3] Nad invertovaným indexem později probíhá vyhledávání. Analýzu textu mají na starost analyzátory, které mohou být pro každý field různé. Toho se v praxi velmi využívá, když je potřeba k jednotlivým fieldům přistupovat různě. V pozdějších kapitolách o implementcai se budeme analyzátory a jejich optimalizací více zabývat.

Další metodou optimalizace už při indexovací fázi může být přidání speciálního fieldu (Obsahuje float hodnotu), který bude později při vyhledávání použit k rekalkulaci relevantnosti daného výsledku. Tuto metodu lze využít také při vyhledávací fázi, ale v indexovací fázi se toho využívá například v případě, že jednotlivé dokumenty mají různou důležitost a později bychom to nemohli určit.

Optimalizace Vyhledávací fáze

Vyhledávání je proces, kdy se uživatelem zadaný text (případně další omezující parametry) porovnávají s indexovanými dokumenty v indexu. Jako výsledek je pak vrácen soubor dokumentů u nichž byla nalezena shoda. Výsledek má dokumenty seřazené dle jejich relevance.

V moderním internetu může hrát i několik málo milisekund výraznou roli. Pakliže vyhledávání trvá příliš dlouho, uživatel může stratit trpělivost a stránku opustit. Z toho důvodu je potřeba na tuto fázi dávat velký pozor.

Ve vyhledávací fázi se formuluje dotaz, na základě kterého je text od uživatele porovnáván s indexovanými dokumenty. A právě to, jakým způsobem je text s dokumety porovnáván, má velký vliv na výkonnost a rychlost poskytnutí výsledků. Je třeba zvážit jaké shody jsou pro nás 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 zpomalit.

Některé kroky, které svádí dělat právě při vyhledávací fázi lze uskutečnit již při fázi indexovací. To pak sice znamená, že déle trvá než se objekt zaindexuje, ale jeho pozdější vyhledání je o to rychlejší. Obecně platí, že některé kroky je lepší učinit jednou v indexovací fázi než opakovaně ten samý krok při fázi vyhledávací. Více o tomto způsobu optimalizace a dalších bude konkrétně popsáno v dalších kapitolách.

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 veliký vliv na výkon.

Mají ale také jiné využití než k vyhledávání a to již při indexovací fázi. Konkrétně v analyzátoru je možné použít regulární výraz k vytvoření tokenů. Tímto způsobem je například možné za pomoci správně definovaného patternu pro CamelCase oddělit jednotlivá slova napsaná v řetězci tímto způsobem.

Je třeba optimalizovat samotný regulární výraz. Špatně definovaný 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 uživateli vrátil prázdný výsledek nám sugesce umožňuje pořádně zkontrolovat index a najít někde podobnost.

Sugesce se dá také považovat za způsob optimalizace jelikož díky ní dokážeme vrátit i výsledek pro například špatně napsaný dotaz místo toho, aby musel být dotaz podán znovu. Stejně tak jako ohlídání uživatelských překliků můžeme suggesci 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 pozdějších kapitolách 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

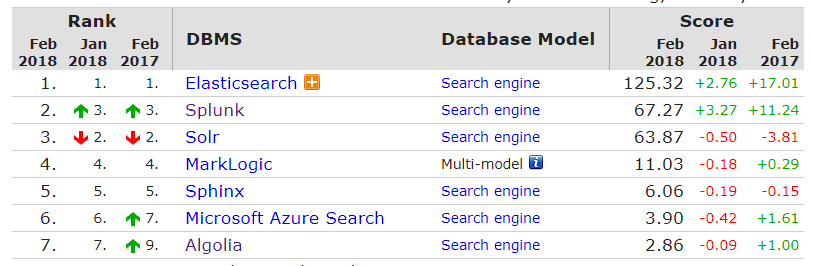
Hlavním cílem strojového učení je navrhnout a vytvořit algoritmy, které umožňují systému používat empirická data, vyvíjet se na základě vlastních zkušeností a připravovat se na změny, které mohou nastat v našem systému. [5] Toho se dá zajisté využít i v případě fulltextového vyhledávání a tak může být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

Technologie fulltextového vyhledávání

Technologií, které nám umožňují fulltextově vyhledávat je mnoho a opravdu je z čeho vybírat. Existuje několik technologií, které jsou tzv. Open-source (otevřený zdrojový kód) a ty jsou z převážné většiny zdarma. (Zpoplatněny mohou být funkcionality, které základní knihovna Apache Lucene nepodporuje, jako je například Machine learning u Elasticsearche). Open-source projekty jsou většinou nadstavbami knihovny Apache Lucene a zpřístupňují tedy její myšlenky a funkcionality v jednodušší podobě než kdybychom používali samotnou knihovnu. 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 jejichž hlavní výhodu se dá považovat to, že nejsou závislé na Lucene knihovně, která ačkoli přináší skvělé myšlenky a podporu velice rychlého indexování a vyhledávání, nepřináší takový výkon jako novější modernější technologie, které jsou lépe optimalizované pro dnešní dobu a poskytují vyhledávání, což mnohé testy dokazují, daleko 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.

Na obrázku je porovnání v současnosti nejvyužívanějších vyhledávacích enginů.

<https://db-engines.com/en/ranking/search+engine>



Ačkoli Splunk, Algolia nebo ostatní nejnovější technologie mohou být v některých ohledech lepším řešením, Lucene technologie dokáží poskytnou stále velice vysoký výkon, který dalece přesahuje nároky některých menších projektů (jako je například téma této práce, Vyhledávání v dokumentaci interního frameworku). A z toho důvodu jsme i pro tento projekt zvolili Lucene technologii. Blíže o jednotlivých nadstavbách Lucene v další kapitole.

4. Apache Lucene

Apache lucene je fulltextový open-source engine napsaný v jazyce JAVA. Nefunguje jako standalone aplikace. Je to knihovna JAVY, kterou může každý použít pro vyhledávací účely. Pokud se v naší vlastní aplikaci rozhodneme použít fulltextové vyhledávání postavené této knihovně, máme tedy tu možnost. Nicméně veškeré nastavení clusteru, uzlů, fragmentů, atd. zůstává na nás. 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 ním. Framework jako např. Elasticsearch má svůj vlastní Zen Discovery modul pro cluster management, naproti tomu Solr používá externí ZooKeeper.

Technologie nad Apache Lucene

Pokud budeme chtít použít některou z technologii postavených na základech Apache lucene, máme hned několik možností. Pro příklad můžeme jmenovat již zmíněný Elasticsearch, Solr nebo také technologie Sphinx. Tyto technologie patří (jak je vidět na obrázku) mezi nejpopulárnější v oblasti vyhledávání a existují všechny ze stejného důvodu. Tím je zprostředkovat uživateli všechny výhody rychlého až real-time indexování a vyhledávání. Některé z nich fungují na separátně běžícím standalone serveru, se kterým nějakým způsobem komunikujeme, obvykle přes http protokol a poskytují API, přes které dokážeme jednodušeji využívat knihovnu Lucene. Bližší specifikace, která může sloužit i jako pomůcka při výběru technologie, je v tabulce.

Tato práce se zabývá technologií Elasticsearch, kterou využívá mnoho velkých a významných projektů. Je to například Wikipedie, StackOverflow nebo GitHub, který díky Elasticsearchi dokáže prohledávat přes 130 bilionů řádků kódu.

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Framework** | | |
| **ElasticSearch** | **Solr** | **Sphinx** |
| **Komunita a vývojáři** | Stará se o něj společnost Elastic. Přestože je projekt open-source a je zveřejněn 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. Nicméně je to také open-source projekt. |
| **Poslední release** | 6.1.2, Leden 2018 | 7.2.1, Leden 2018 | 3.0.1, Prosinec 2017,  Předchozí Červenec 2016 |
| **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 v závislé na čase. Od nové verze Elastic dokáže rozlišit zda iterovat přes všechny dokumenty nebo jen část. | 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 |

6. Elasticsearch

Elasticsearch je vyhledávací a analytický engine. Dovoluje 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 vlastně nepřináší nic nového. Žádná z jeho částí není nijak převratná nebo objevná. To co Elastic dělá jedinečným, je kombinace malých funkčních částí pospojovaných do fungujícího systému.

Každá databáze dokáže selektovat data na základě přesných údajů - dokáže filtrovat na základě času, třídit. Ale co nedokáže je fulltextové vyhledávání, nahrazování synonymy, “scoring” dokumentů na základě jejich relevantnosti, nedokáže agregovat data a generovat analýzu. A co už vůbec ne je dělat toto všechno v reálném čase.

Elasticsearch je stejně jako vyhledávací knihovna Apache Lucene, nad kterou je vytvořen, open-source. Apache Lucene je, jak již bylo zmíněno, pravděpodobně nejvíce rozšířenou a plně podporovatelnou vyhledávací knihovnou dneška. Ale je to jen knihovna. K jejímu využití je potřeba pracovat s Javou a integrovat ji přímo do aplikace. Lucene je velice komplexní a složitá na pochopení, chtěli-li bychom použít její čistou verzi. ElasticSearch je také napsán v jazyce JAVA a využívá Lucene knihovnu pro veškeré vyhledávání a indexování. To vše zabaluje na standalone server a díky tomu jsou veškeré funkcionality Apache lucene daleko přístupnější. Se serverem pak můžeme komunikovat několika způsoby. Skrze Restful API z webového prohlížeče, z vlastní aplikace nebo jednoduše z příkazového řádku.

Elasticsearch je dokumentově orientovaný. Objekty v naší aplikaci jsou málokdy pouze jednoduché objekty s atributy klíč - hodnota. Většinou máme objekty více komplexní. Obsahují různá pole nebo jiné objekty a dříve nebo později je budeme chtít uložit do databáze, což je i přes dnešní mapovací frameworky, viz Hibernate například, docela zdlouhavý proces. Naštěstí Elasticsearch je dokumentově orientovaný. To znamená, že ukládá a indexuje celé objekty. Indexováním se objekty stanou dohledatelnými.

Elasticsearch hledá, řadí nebo třídí celé dokumenty, ne jako je to u databází jen řádky 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, Cluste, Node, Shard---------------------------------------------

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 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 a v případě překročení všech je za potřebí zvolit nový hlavní 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. Jednak je to detekce chyb u Master uzlu. Ostatní uzly odesílají Ping na Master uzel, aby zjistily, zda je stále aktivní. Druhak Master uzel odešle Ping do ostatních uzlů k detekci, 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, když spustíme instanci Elasticu, spustíme 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 (Skrze 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. “Vertikálni rozšíření” (= Rozšíření ve smyslu nákupu větších, výkonějších zařízení).

Jeden uzel dokáže zároveň sloužit více účelům. Záleží na konfiguraci k jakému účelu bude uzel určen. Máme několik typů uzlů. Hlavní uzel, Datový uzel, Ingest uzel a Kmenový uzel.

Cluster obsahuje právě jeden Hlavní uzel. Je velice důležité mít v clusteru stabilní hlavní uzel. 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 dalšího typu uzlu 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 potřebujeme například 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 nějakého 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, který nám v podobě barvy dokáže říct zda jsou všechny primární a replikované fragmenty aktivní.

Fragment (=Shard)

Fragmenty se využívají k rozdělení indexu na menší části, do více fragmentů, kde každý fragment je samostatná fungující jednotka (samostatná instance Apache Lucene), která dokáže plně manipulovat se svou částí dat. Naše dokumenty jsou indexovány do fragmentů, ale naše 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 kontejnerů pro data. Dokumenty jsou ukládány do fragmentů, kde každý fragment náleží nějakému z uzlu v našem clusteru. Jak náš cluster roste, Elastic automaticky migruje fragmenty mezi uzly a tím nám 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.

------------------------------------Základní termíny elasticu, proces indexování---------------------------------------

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.)

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

Index

Index je kolekce dokumentů, které mají podobnou charakteristiku. Například můžeme mít jeden index pro zaměstnance a druhý index pro objednávky. Index je definován jménem, podle kterého je pak správný index vybrán během průběhu všech procesů (indexace, vyhledávání, atd). Jméno tedy musí být unikátní.

Typ

V rámci jednoho indexu můžeme definovat jeden nebo více typů. Počet typů je zcela na nás. Slouží nám akorát pro přehlednost. Pro představu jsme v indexu zaměstnanci, kde si můžeme vytvořit typ např. 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 může být ještě dále rozdělená 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.

---------------------------------------------------Dotazování, Query-----------------------------------------------------------

Query

Elasticsearch podporuje robustní a velmi flexibilní query jazyk Querry DSL, který umožňuje psát I velice komplikované a robustní dotazy. Dotazy jsou psány ve formátu JSON.

Uvnitř query jsou napsány jednotlivé dotazy nebo případně filtry. Dotaz nám počítá výsledné “score” dokumentu, které nám určuje jak moc je tento vyhledaný dokument relevantní vzhledem k ostatním indexovaným dokumentům. Dotazy a jejich score se vykonají vždy, když uvnitř napsaného dotazu použijeme parametr “query”.

Naproti tomu Filtr nám výsledné score nepočítá. Odpovídá pouze na otázku, zda dokument sedí na náš dotaz či nikoliv. Filtr použijeme, když uvnitř napsaného dotazu použijeme parametr “filter” nebo “must/must\_not” uvnitř parametru “bool”.

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

--------------------------------------------------------Komunikace--------------------------------------------------------------

Komunikace

To jak budeme komunikovat s elasticem záleží na tom jaký používáme jazyk. Odkud voláme naše dotazy na server. Komunikace může probíhat dvěma způsoby. Skrze JAVA API (port 9300) nebo přes Restful API (port 9200).

Java API

Jestliže máme naší aplikaci napsanou v jazyce JAVA, pak lze s Elasticem komunikovat přes JAVA API. Elastic 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 (“Coordination only”) uzel jak bylo popsáno v dřívější kapitole. 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 (“lehči”) 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 Elasticem je přes restful API. Odkudkoli můžeme s Elasticem komunikovat za použití http protokolu přes port 9200. Můžeme posílat requesty z aplikace v kterémkoli jazyce, z webového prohlížeče nebo klidně z příkazového řádku.

-----------------------------------------------------------Rozšíření---------------------------------------------------------------

Rozšířeni

Jak již bylo zmíněno, Elastic nepřináší nic nového ani objevného. Vše co dokáže už bylo dokázáno dříve než byl Elastic v únoru roku 2010 vytvořen. To, co ho ale staví na špičku mezi full-textovými vyhledávači je spojení maličkostí ve skvěle fungující celek. Nejen, že spoustu užitečných funkcionalit má Elasticsearch implementovaných přímo v sobě, ale existuje i celá řada externích rozšíření, které základní funkcionalitu rozšiřují o další užitečné funkce.

Existuje celá řada různých nástrojů, které nám usnadňují nebo vylepšují práci s Elasticem. Díky nim dokážeme lépe monitorovat stav našeho serveru a víme kde je potřeba co upravit nebo vylepšit, abychom maximalizovali výkon.

X-pack

X-pack je balíček rozšíření pro elastic. X-pack zabaluje řadu pluginů do jednoho balíčku, který zbývá pouze nainstalovat. Sám se stará o správu a kompatibilitu jednotlivých pluginů a odpadá tedy starost s různými verzemi pluginů apod.

Mezi pluginy můžeme nalézt nástroje starající se o bezpečnost, monitoring, reporting, schopnost pracovat s grafy nebo dokonce od verze elasticu 5.4 i Machine learning.

Kibana

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

Logstash

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

Beats

Beasts je opět open-source projekt, který je nainstalován jako agent na serveru naší 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. Packet beat je síťový analyzer, který posílá informace o transakcích provedených mezi aplikačními servery. Filebeat posílá log soubory z našeho serveru. Metricbeat je zase monitorovací agent, který periodicky sbírá data z operačního systému a z procesů běžících na serveru.

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.