Úvod do Elasticsearch a knihovny Elasticsearch-DSL

whys

O mně

- Standa (pohodář ;)
- 4. rokem ve Whys
- backend developer
  (nabrán původně na React, že jen s něčím vypomůžu na backendu :D)
- technologie:
  - Django,
  - Django - REST,
  - Elasticsearch
- do budoucna snad full-stack
  (a konečně i ten React :D)

Cíl přednášky

- získat představu, co je to elasticsearch, jaká je jeho struktura
- naučit se základní teoretické principy
- naučit se pracovat s knihovnou elasticsearch-dsl
- vidět celé workflow a umět vše převést do praxe

Obsah

- **Elasticsearch**
  - co to je, jak a kde se dá využít
  - základní pojmy a struktura

- **Elasticsearch-DSL**
  - motivace a popis aplikace
  - propojení aplikace s elasticsearch
  - založení indexu a modelace jeho struktury
  - rozdíly mezi datovými typy a jev object's flattening
  - indexace dat
  - sestavení query, spuštění a práce s response
  - filter a query kontext, analýza textu, full-text search
  - filtrování
  - agregace

Elasticsearch

- full-text vyhledávácí engine (Java)
- open-source
- NO-SQL databáze
- založený na knihovně apache Lucene (Java) - jádro
- rychlý, škálovatelný, odolný proti výpadkům, ztrátě dat
- komunikace ve formátu JSON
- kdo používá: Netflix, Slack, LinkedIn

Proč elasticsearch?

- obecně výhodný pro vyhledávání:
  - **rychlost**
  - podpora **full-text search**
  - multijazyčnost (široká škála analyzátorů)
  - podpora pro **filtrování, agregace**
  - **horizontální škálovatelnost** (přidávání dalších serverů)
- horší (ve srovnání s ostatními databázemi) v uchovávání dat:
  - integrita dat
  - náročnější správa - návrh, modelace, indexování

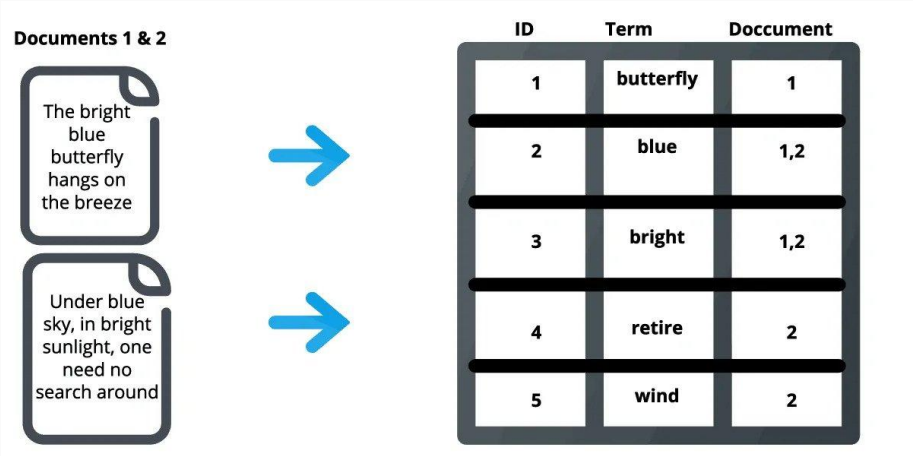
=> **nejvýhodnější kombinace obou přístupů**: PostgreSQL | MongoDB | … + Elasticsearch

Dokument, Index

- **dokument** = strukturované uspořádání dat určitého typu (zákazník, produkt, …)
  - unikátní ID
  - “jeden řádek v tabulce”
- **index** = soubor dokumentů určitého typu
  - “databáze”

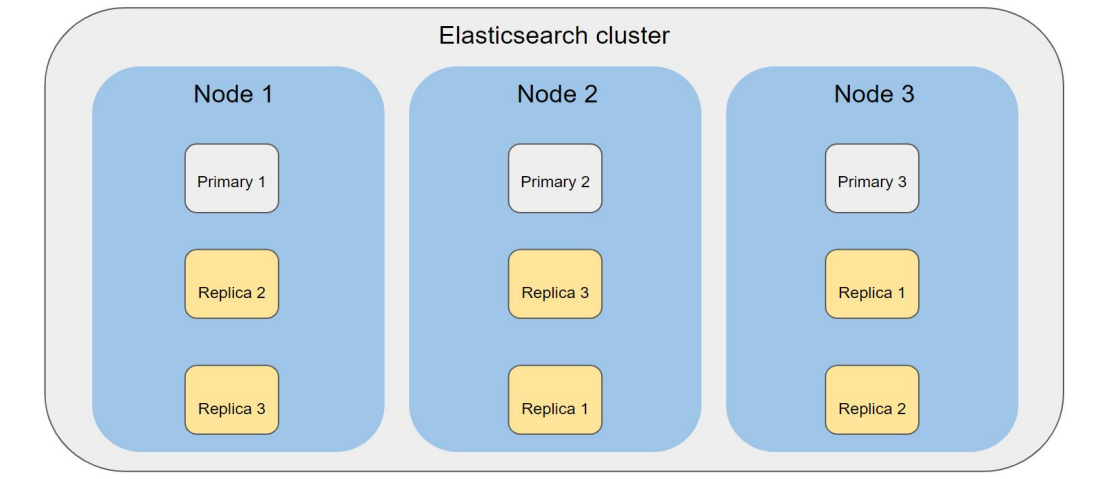
Inverted index

- způsob, kterým elastic ukládá data
- obsah dokumentů není indexován přímo, ale vytváří se z nich hashmap “slov” tzv. termů s odkazy na obsahující dokumenty



Struktura backendu

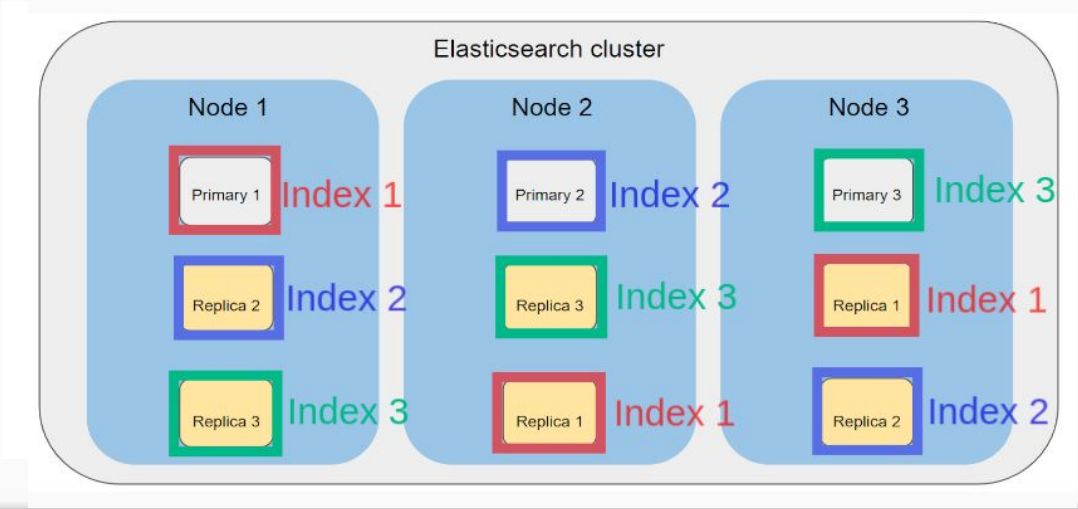
- 3 základní prvky - **shard, node, cluster**
- shard < node (server) < cluster
- shard: **primary** (čtení + zápis), **replica** (pouze čtení)



whys

Struktura backendu a index

- rozprostřen na 1 nebo více shardů (zároveň může více indexů na 1 shardu)
- elastic automaticky re-balancuje podle potřeby



whys

Elasticsearch DSL

6.4.0

Motivace

- mnoho build-in funkcí, abstrakce od formátu JSON, zjednodušení práce

elasticsearch-py

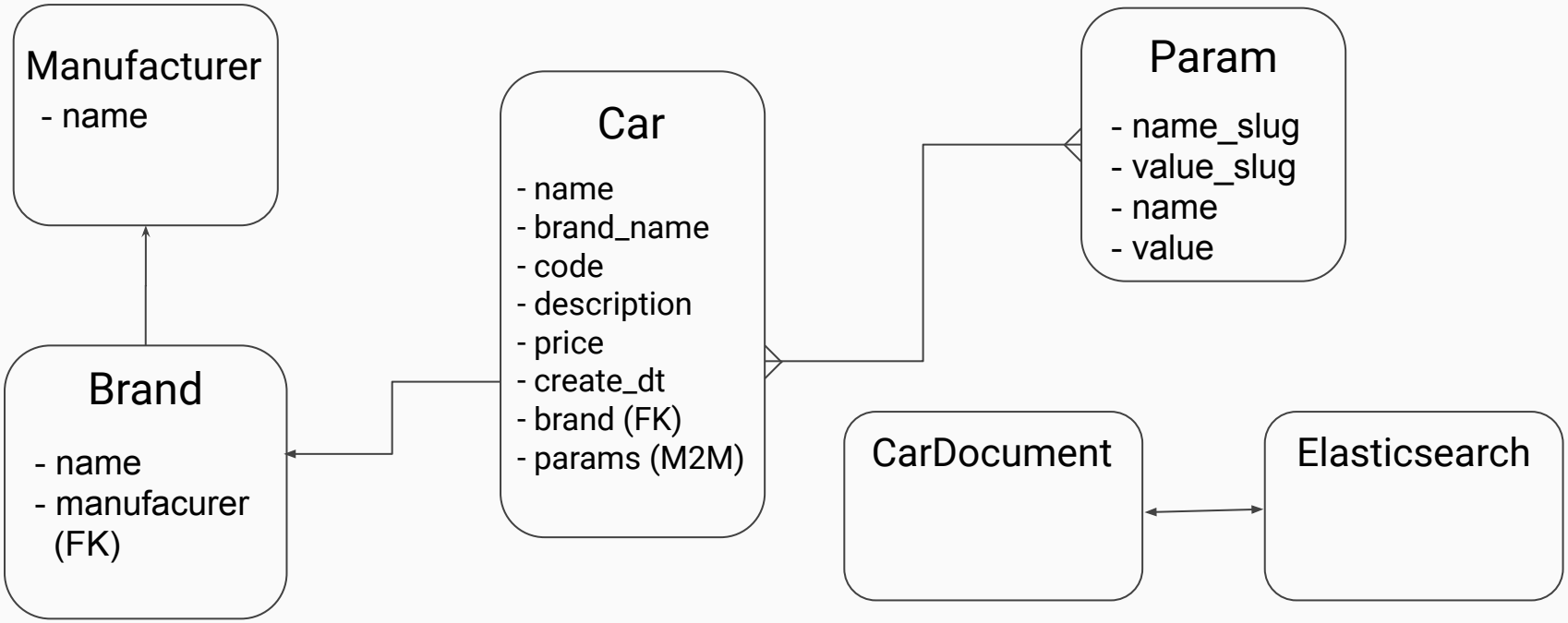
```
{
 "query": {
   "bool": {
     "filter": [
       {
         "term": {
           "brand_name": "BMW"
         }
       }
     ]
   }
 }
}
```

elasticsearch-dsl

```
CarDocument
   .search()
   .filter(Q("term", brand_name="BMW"))
   .execute()
```

- ve skutečnosti se instalují dvě knihovny:
  - **elasticsearch-py =** low level knihovna, práce hlavně s formátem JSON
  - **elasticsearch-dsl** = high level knihovna, wrapper nad elasticsearch-py

Naše aplikace - Auta



whys

Propojení aplikace s elasticsearch

- zajišťuje třída **Elasticsearch** == klient
- 3 možnosti:
  - a) `Elasticsearch([{'host': 'localhost', 'port': 9200}])`
  - b) `connections.create_connection(alias='default', hosts=['localhost'])`
  - c)
```
connections.configure(
    **{
        'default': {
            'hosts': [{
                'host': 'localhost',
                'port': '9200',
    }]}}
)
```

- ověření:
  - `Index("elastic-cars")._get_connection().info()`
  - `http://localhost:9200/`

Vytvoření a nastavení indexu

- zdědění třídy **Document**
- definice Meta třídy **Index**
  - zde alespoň name = “nazev_indexu”
  - volitelně i aliases, settings, …
- 3 možnosti:
  - `Elasticsearch.indices.create(index="elastic-cars")`
  - `Index("elastic-cars").create()`
  - `CarDocument().init()`
- ověření:
  - `Index("elastic-cars").get_settings()`
  - `http://localhost:9200/elastic-cars/doc/_settings`

Název indexu vs alias

- **název** = obvykle delší název obohacený např. o časové razítko
  - unikátní
  - elastic-cars_2021_03_16_19_51_36
- **alias** = zástupný název, obvykle jednodušší, můžeme využívat pro práci s DSL
  - lze více aliasů na jeden index
  - může mít filtr pro různé pohledy na index (např. dokumenty vytvořené v roce 2019)
  - ```
    index = Index("elastic-cars_2021_03_16_19_51_36")
    index.aliases(**{"elastic-cars": {}})
    ```
- ověření:
  - `index.get_alias()`
  - `http://localhost:9200/alias_name/_alias`
    ```

Struktura indexu a mapping

- definice polí indexu dle potřeb aplikace
- při založení indexu se vytvoří v elasticku popis polí indexu tzv. **mapping**
- ověření:
  - `Index(index="elastic-cars").get_mapping()`
  - `http://localhost:9200/elastic-cars/doc/_mapping`

Typy polí

- **Keyword**
- **Text**
- **Object**
- **Nested**
- Double
- Integer
- Range
- Join field type (parent/child)
- ...

Keyword vs Text

- **Keyword** = elastic hledá přesnou shodu s vyhledávanou frází
- **Text** = full-text vyhledávání
  - umí také Keyword
  - záleží na **analyzátorech**
  - podpora multivalue polí (např. Text může být také Keyword)

Object vs Nested

- **Object** = **inner object** = klasický objekt (python slovník/json objekt, …) nebo pole o jednom objektu
  - **pouze one-to-one relace**
  - podléhá jevu ***objects flattening***
- **Nested =** pole objektů se stejnými daty
  - **many-to-one, many-to-many**
  - každý 1 objekt je **nezávislý** na ostatních

Object’s flattening

- **elasticsearch** má plochou/přímou hierarchii
- pole typu **Object** jsou vnitřně napřímeny:

```
{
 "params" : [
   {
     "name" : "engine-type",
     "value" : "diesel"
   },
   {
     "name" : "top-speed",
     "value" : 240
   }
 ]
}
```

```
{
  "params.name" ["engine-type", "top-speed"],
  "params.value" [diesel, 240],
}
```

Indexace dat

- přenos uložených (DB) dat do elasticsearch v nadefinované (Dokument) podobě
- uložení po jednotlivých dokumentech:
  - pro měnší počet dat

```
CarDocument(name="BMW").save()
```

- všechny dokumenty najednou:

```
elasticsearch.helpers.bulk(
    client,
    "index",
    [CarDocument(name="BMW"), ...]
)
```

Sestavení query

- různé způsoby (ekvivalentní výsledek):
  - `Match(description={"query": "year model"})`
  - `Q("match", description={"query": "year model"})`
  - `Q("match", description="year model")`

- záleží na použití - konfigurace vs přehlednost
- zápis strukturovaných dat (Object, multifield):
  - `Q("match", description__en="year model")`
  - `Q("match", **{"description.en": "year model"})`

- lze řetězit s operátory **&** (and), | (or), ~ (negate):
  - `Q(...) & Q(...) | ~Q(...)`

Spuštění hledání a response

- základem je **Search()** objekt
- sestavenou query vložíme do **.query()** funkce
- spuštění funkcí **.execute()**
- celý výraz můžeme také řetězit: **Search().query().execute()**
- výsledky vyhledávání jsou poté v poli **response.hits**
- ```
  query = Q(...) & Q(...) ...

  search = Search(index="elastic-cars").query(query)

  response = search.execute()

  response.hits
  ```
  ```

Řazení a paginace

- defaultně se řadí **dle skóre**
- názvy polí do funkce **.sort()**
    - **-** .. sestupně
- defaultní stránkování **10 výsledků**
- vlastní stránkování jako slice indexing v pythonu **[od:do]**

```
search = Search(index="elastic-cars").sort('price','-brand_name')

search_paginated = search[0:20]

response = search.execute()
```

Příklad

elasticsearch-dsl

```
Search(index="elastic-cars") \
        .query(
            Q("match", brand_name="BMW") &
            ~Q("range", price={"lte":
1500000})
        ) \
        .sort(
            "-price"
        )[0:10]
```

elasticsearch-py

```
{
 "query": {
   "bool": {
     "must": [
       {
         "match": {
           "brand_name": "BMW"
         }
       }
     ],
     "must_not": [
       {
         "range": {
           "price": {
             "lte": 1500000
           }
         }
       }
     ]
   }
 },
 "sort": [
   {
     "price": {
       "order": "desc"
     }
   }
 ],
 "from": 0,
 "size": 10
}
```

Filter a query kontext

- defaultně řazení výsledků podle vhodnosti tzv. **skóre relevantnosti**
  - čím vyšší skóre, tím je dokument vhodnější
  - v REST api field **_score**
- **query kontext**
  - **ovlivňuje skóre**
  - otázka: Jak dobře dokument splňuje query klauzuli? -> odpověď v číslu = skóre
- **filter kontext**
  - **neovlivňuje skóre**
  - elastic cachuje
  - otázka: Splňuje dokument query klauzuli? -> odpověď ANO/NE
  - klíčová slova v klauzulích: **filter, must_not, filter - constant_score, filter - aggregation**

Filter a query kontext - příklad

elasticsearch-dsl

```
Search(index="elastic-cars") \
   # QUERY CONTEXT
   .query(
        Q("match", brand_name="BMW")
   ) \
   # FILTER CONTEXT
   .filter(
         Q("match", tags__cs="diesel")
   )
```

elasticsearch-py

```
{
 "query": {
    // QUERY CONTEXT
    "bool": {
      // QUERY CONTEXT
      "filter": [
        // FILTER CONTEXT
        {
          "match": {
            "tags.cs": "diesel"
          }
        }
      ],
      // QUERY CONTEXT
      "must": [
        {
          "match": {
            "brand_name": "BMW"
          }
        }
      ]
    }
 }
}
```

Analýza textu

- důležité pro full text search
- **probíhá ve dvou fázích: indexace, full text search**
- **tokenizace** = rozdělení vstupní věty na menší části - termy (tokeny)
- **normalizace** = převedení dílčích částí na standardizované formáty
  - normalizéry - produkují pouze jeden znak
- **analyzér** = soubor pravidel pro tokenizaci a normalizaci
  - index analyzér, search analyzér
  - často stejné, někdy se hodí různé
- příklad: “**Koupil jsem si krásné nové auto.**”
  - po tokenizaci: [**Koupil, jsem, si, krásné, nové, auto**]
  - po normalizaci: [**koupil, krasne, nove, auto**]

Full-text search 1

- vyhledávání v celém obsahu polí
- zpravidla datový typ **Text**
- typy queries: **match, multi_match, …**
  - **probíhá analýza vyhledávaných dat - uplatňují se analyzátory**
- sestavená query do **.query()** funkce

```
query = Q("match", brand_name="Mer")

response = Search(index="elastic-cars").query(query).execute()
```

Full-text search 2

- **boostování** = zvýšení skóre dokumentu na základě splnění určitých kritérií
  - boostovat můžeme na úrovni celé query nebo dílčích polí
- **fuzziness =** míra nepřesnosti

```
multi_match_query = FunctionScore(
    query=Q("multi_match", query="Mercedes", fields=['name']), fuzziness=2),
    boost=5,
    functions=[
        {"filter": Q("match", name="Mercedes")}, "weight": 5},
    ],
    boost_mode="multiply"
)
response = Search.query(multi_match_query).execute()
```

Filtrování

- datové typy např. **Keyword**, **Range**, **Integer**, ale i **Text(raw=Keyword())**
- typy queries: **term, range, nested, ...**
  - **neprobíhá analýza vyhledávaných dat - neuplatňují se analyzátory**
- může využít ale i **match** - uplatní se analyzátory, ale nezapočítá se skóre

  (filter context)

- výsledek **do .filter() funkce:**

```
filter_query = Q("term", price=1200000)

response = Search().filter(filter_query).execute()
```

Agregace

- sumarizace dat, statistiky a další analýzy
- 3 druhy:
  - **Metric** = suma, průměr, …
  - **Bucket** = shlukování dokumentů k sobě dle zadaného kritéria
  - **Pipeline** = propojuje agregace, vstupem výsledek jiné agregace
- **agregace jsou spuštěny až po full text search a filtrování**
- sestavená query se **nevkládá** zpět do objektu
- **lze řetězit**
- výsledek v **response.aggregations[<agg_name>]**

```
search = Search()
search.aggs.bucket("price_range", A("ranges", field="price", from=150000, to=200000))
response = search.execute()
response.aggregations["price_range"]
```

Literatura

- Elasticsearch 6.4.0 DSL. *Elasticsearch DSL* [online]. 2020 [cit. 2021-03-17]. Dostupné z: https://elasticsearch-dsl.readthedocs.io/en/6.4.0/
- Elasticsearch 6.8.2. *Elasticsearch PY* [online]. 2021 [cit. 2021-03-17]. Dostupné z: https://elasticsearch-py.readthedocs.io/en/6.8.2/
- Elasticsearch Reference 6.8. *Elasticsearch* [online]. 2021 [cit. 2021-03-17]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/6.8/index.html

Děkuji za pozornost :)