Datenbanken-Projekt Elasticsearch

Inhaltsverzeichnis

Elasticsearch

Allgemein

Elasticsearch ist ein verteilter Suchserver bzw. eine Suchmaschine und Analytics Engine, deren Möglichkeiten über die einer einfachen Datenbank hinausgehen. Eine wachsende Zahl von Anwendungsfällen kann damit abgedeckt werden. Elasticsearch wurde in Java geschrieben und basiert auf Lucene, einer Open Source Programmbibliothek zur Volltextsuche. Die Kommunikation zwischen Clients und Server erfolgt über eine REST-Schnittstelle. Eine weitere Besonderheit ist, dass diese Schnittstelle JSON erwartet und in JSON antwortet. Im selben Format werden ebenfalls alle Daten gespeichert.

Zwischen den verschiedenen Versionen von Elasticsearch existieren in der Verwendung, unter anderem der Suchabfragen und Mappings, mitunter große Unterschiede. In der offiziellen Dokumentation wird dies schnell ersichtlich. In unserem Fall wurde Elasticsearch 6.5 verwendet. Alle weiteren Ausführungen beziehen sich daher auf diese Version.

Architektur der Datenbank

In *Elasticsearch* werden die Daten in Indizes abgelegt. Diese sind Vergleichbar mit Tabellen in relationalen Datenbanken. Innerhalb der Indizes werden dann Dokumente abgelegt, die man mit einer einzelnen Zeile in relationalen Datenbanken vergleichen kann. Dokumente enthalten wiederum verschiedene Felder, von denen jedes einen Datentyp hat.
Anders als in relationalen Datenbanken sind aufgrund der Speicherung im JSON-Format auch Datentypen wie Arrays oder Subdokumente (“nested Object”) möglich. Außerdem müssen nicht alle Dokumente eines Index die gleichen Felder haben. Anzahl und Typen von Feldern sind für jedes Dokument frei definierbar, allerdings müssen gleiche Felder den gleichen Typ haben.
Weiterhin gibt es auch die Möglichkeit, den Feldern Typen zuzuordnen. Dies macht man im sogenannten Mapping für einen Index. Allerdings kann jedes Dokument eines Indexes weitere Felder enthalten, die nicht im Mapping vordefiniert wurden.

Für die Speicherung teilt Elasticsearch jeden Index in mehrere Teile auf (sogenannte *shards*). Diese können auf mehrere Server verteilt werden, um eine Lastverteilung zu ermöglichen oder sich gegen Ausfälle abzusichern.
Bei der Indexierung nach dem Senden eines Dokumentes werden außerdem die Texte an vordefinierten Stellen wie Kommas und Leerzeichen in Wörter aufgetrennt. Zusätzlich werden weitere Schritte zur Bereinigung der Daten unternommen; beispielsweise werden alle Buchstaben in Kleinbuchstaben umgewandelt. Es besteht weiterhin die Möglichkeit, weitere eigene Schritte zu definieren. Anschließend werden im Anschluss die Ergebnisse und die Ursprungsdaten für die einzelnen Dokumente gespeichert.

Vorteile und Möglichkeiten

Das allgemein verfolgte Ziel ist es, eine stets hohe Verfügbarkeit bei einer Gewährleistung von schnellen Suchanfragen zu ermöglichen. Neben der generellen Architektur der Datenbank wird dies unter anderem durch eine automatische Indexierung sowie durch Unterstützung für Volltextsuchen gewährleistet. Durch die Speicherung der Daten in mehreren *shards* wird darüber hinaus eine gute Skalierbarkeit garantiert. Durch clusterübergreifende Replikation sowie durch die Verwendung in einer verteilten Umgebung soll dabei eine hohe Resilienz sichergestellt werden. Dabei wird versucht, stets eine hohe Flexibilität zu bewahren. Dies wird durch verschiedene Möglichkeiten für verschiedene Daten und durch die Verfügungstellung unterschiedlicher Erweiterungen erreicht, wie zum Beispiel *App Search,* Security Analytics, Metriken und Logging. Eine Großzahl gängiger Programmiersprachen wie Java und Python werden dabei unterstützt.

Durch die Unterstützung von Spark und Hadoop kann Elasticsearch weiterhin als Suchmaschine und Analyse-Tool für Big Data genutzt werden. Auch eine Abfrage von Geo-Daten mit komplexen Formen, wie definiert in *GeoJSON*, können abgefragt werden. Ein beispielhafter Anwendungsfall wäre die Suche nach Restaurants in einem bestimmten Umkreis um einen Nutzer, welche dann zusätzlich nach Preis, Distanz und Bewertung gefiltert werden können.

Kibana

Kibana ist eine Open-Source-Analyseplattform, die auf Elasticsearch aufbaut. Sie ist browserbasiert und ermöglicht eine Visualisierung und Suche der in Elasticsearch enthaltenen Daten. Weiterhin werden hilfreiche Entwicklertools mitgeliefert, die es dem Nutzer beispielsweise über den Browser ermöglichen, Queries zu verfassen und zu verschicken. In einem Dashboard zeigt nützliche Informationen über Elasticsearch und den darin enthaltenen Daten.
In diesem Projekt wurde Kibana häufig als praktisches Tool für die Bedienung von Elasticsearch genutzt, also um Queries zu bilden und auszuprobieren, die etwa Daten löschen, hinzufügen, oder Mappings erstellen.

Kibana, *Elasticsearch* und *Logstash* bilden zusammen den *Elastic Stack.*

Mappings in Elastic

Es gibt dynamische und explizite Mappings. Ein dynamisches Mapping wird automatisch generiert aus den Dokumenten und kann sich ständig ändern, wogegen ein explizites Mapping für einen Index im Vorwege Felder mit Namen, Datentypen und weiteren gewünschten Eigenschaften definiert. Wenn man kein Mapping festlegt, wird automatisch das dynamische Mapping verwendet.
Mappings können genutzt werden, um festzulegen welche String-Felder als Volltextfelder behandelt werden sollen, welche Felder Zahlen, Daten oder Geo-Daten enthalten oder welches Format zum Beispiel ein Datumsfeld haben soll.

Als Datentypen gibt es folgende Auswahlmöglichkeiten:

- Einfache Typen: text, keyword, date, long, double, boolean, ip, usw.
- Hierarchische Typen: object (für ein JSON Object) oder nested (für mehrere JSON Objekte im Array)
- Spezielle Datentypen: Geo-Koordinaten, alias, join

Um “mapping explosions” zu verhindern, die zu Speicherfehlern führen können, gibt es verschiedene Einstellungen, die im mapping festgelegt werden können:

- maximale Anzahl von Feldern in einem Dokument
- die maximale Tiefe eines Dokumentes
- die maximale Anzahl an “nested fields” in Dokumenten

Suchanfragen in Elastic

Suchanfragen an den Suchserver werden ebenfalls im JSON-Format formuliert. Mit diesen Queries können Dokumente innerhalb eines Indexes gesucht, gefiltert und aggregiert werden. Mit Hilfe von Score-Werten können hier die passendsten Dokumente ausgewählt werden. Dies trägt auch zur Analyse der Daten bei.

Dabei gibt es verschiedene Parameter, die man nutzen kann:

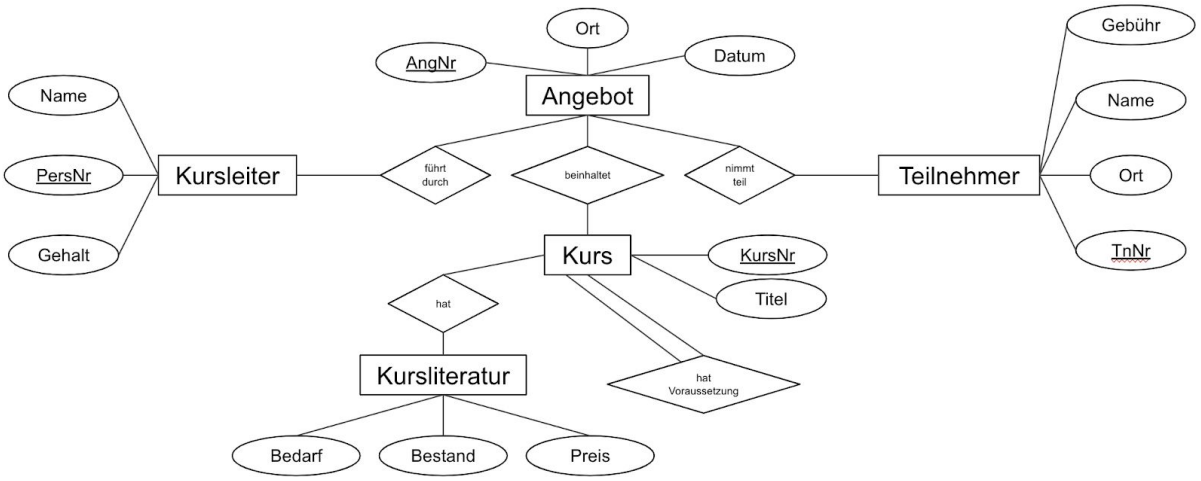
- “query” bestimmt welche Dokumente (mit welchem Inhalt) gefunden werden und auch in welcher Reihenfolge. Um dies zu realisieren hat dieser Parameter verschiedene mögliche Unterobjekte:
    - “match_all”: { }
      → findet alle Dokumente im Index
    - “match”: {
      “<*Feldname*>” : “<*Suchbegriff*>”
      }
      → **findet alle Dokumente mit dem Suchbegriff im gewünschten Feld**
    - “multi_match”: {
      “query”: “<*Suchbegriff*>”,
      “fields”: [“<*Feldname*>”, “<*Feldname*>”]
      }
      → **wie “match” für mehrere Felder**
    - “bool”: {
      “must”: {
      “match”: { “<*Feldname*>”: “<*Suchbegriff*>” }
      },
      → **Feld muss Suchbegriff enthalten**
      “must_not”: {
      “match”: { “<*Feldname*>”: “<*Suchbegriff*>” }
      },
      → **Feld darf Suchbegriff nicht enthalten**
      “should”: {
      “match”: { “<*Feldname*>”: “<*Suchbegriff*>” },
      “match”: { “<*Feldname*>”: “<*Suchbegriff*>” }
      }
      → **eines der Felder muss den Suchbegriff enthalten**
      }
- “size” legt das maximum an angezeigten Dokumenten fest
- “from” ermöglicht eine Aufteilung langer Trefferlisten
- “_source” ermöglicht die Auswahl einzelner Felder, damit nur diese im Ergebnis angezeigt werden
- “sort” ermöglicht eine gewünschte Sortierung.

Eine weitere Möglichkeit, die Elasticsearch bietet, sind Aggregationen. Dabei kann man einzelne Felder nach deren verschiedenen Ergebnissen zusammenfassen und bekommt zusätzlich eine Häufigkeit ausgegeben. Des weiteren sind damit Durchschnitts-, Minimum- und Maximum-Berechnungen möglich, jeweils für ein aggregiertes Feld.

Durch die Nutzung des *script*-Feldes können außerdem kurze Scripte in die Query eingebunden werden, um beispielsweise Felder mit Rechenoperationen verknüpft oder mit den gängigen Operatoren verglichen werden. Diese Felder können sowohl zum Sortieren genutzt als auch ausgegeben werden.

Nutzung von Elastic im konkreten Anwendungsfall

ER-Modell und Mapping



Unser grundlegendes Datenmodell hat sich im Vergleich zu den gegeben SQL-Statements nicht verändert. Jedoch besteht trotzdem ein großer Unterschied zwischen der Ablage der Daten in SQL-Tabellen und in Elasticsearch-Indizes.

Elasticsearch bietet uns zusätzlich die Optionen, Arrays und Objekte als Felder zu nutzen. Nun können nicht mehr nur ein Wert in einem Feld, sondern mehrere Werte in mehreren Ebenen gespeichert werden. Aus diesem Grund denormalisieren wir die Daten für das Elasticsearch-Mapping nicht. Hier liegt der große Unterschied in der abgelegten Struktur.

Im ersten Schritt bestand unser Ziel darin, doppelte Datenhaltung zu vermeiden und mehrere Indizes anzulegen, die über die IDs (PersNr, AngNr usw.) miteinander verbunden werden sollten, also ähnlich wie das Vorgehen bei Fremdschlüsseln in SQL. In früheren Elasticsearch-Versionen wäre dies direkt im Mapping über Referenzfelder möglich gewesen. Dies ist in der verwendeten Version 6.5 nicht mehr möglich. Aus diesem Grund sollten anschließend die verschiedenen Indizes in den Abfragen verknüpft werden. Bei dieser angedachten Struktur wäre jeweils ein Index für die Teilnehmer, einer für die Kursleiter und einer für den Kurs gewählt worden. Im Kursindex wären dann die Kursliteratur als *nested Object* und die Voraussetzungskurse als Array enthalten. Der Index *Angebot* sollte all diese Daten zusammenfassen und ein Feld mit der *PersNr* des Kursleiters, sowie ein Feld mit der *KursNr* und ein Array mit Objekten aus *TnNr* sowie der zugehörigen Gebühr enthalten. Die zugrunde liegende Idee, diese Daten in den Abfragen mit einem *Join* zu verknüpfen, wie man es aus SQL gewohnt ist, ist in Elasticsearch allerdings nicht umsetzbar. Aus diesem Grund mussten wir unser Vorgehen überdenken und kamen zu dem Schluss, dass wir nicht

ohne doppelte Datenhaltung auskommen können. Alle Daten wurden daher im Angebotsindex zusammengeführt. Die jeweiligen ID-Felder, die als Verknüpfung gedacht waren,wurden durch den kompletten Datensatz ersetzt. Dies macht alle Abfragen auf diesen Daten auch mit Elasticsearch möglich und ähnelt eher dem Gedanken, in Elasticsearch schnelle Suchen auf großen Datenmengen durchzuführen und dabei auf Normalisierung zu verzichten.

Unser verwendetes Mapping lautet wie folgt:

```
{
    "settings": { "number_of_shards": 1 },
    "mappings": {
        "_doc": {
            "properties": {
                "AngNr": { "type": "keyword" },
                "Kurs": {
                    "type": "nested",
                    "properties": {
                        "KursNr": { "type": "keyword" },
                        "Titel": { "type": "keyword" },
                        "KursLit": {
                            "type": "nested",
                            "properties": {
                                "Bestand": { "type": "integer" },
                                "Bedarf": { "type": "integer" },
                                "Preis": { "type": "double" } } },
                        "Voraus": {
                            "type": "nested",
                            "properties": { "VorNr": { "type": "keyword" } } } } },
                "Datum": {
                    "type": "date",
                    "format": "dd.MM.yyyy" },
                "Ort": { "type": "keyword" },
                "Kursleiter": {
                    "type": "nested",
                    "properties": {
                        "PersNr": { "type": "keyword" },
                        "Name": { "type": "keyword" },
                        "Gehalt": { "type": "double"  } } },
                "Teilnehmer": {
                    "type": "nested",
                    "properties": {
                        "TnNr": { "type": "keyword" },
                        "Name": { "type": "keyword" },
                        "Ort": { "type": "keyword" },
                        "Gebuehr": {
                            "type": "double",
                            "null_value": 0 } } } } } } }
```

Aufgaben-Queries

Create

Die *Create*-Queries waren mit Hilfe eines Bulk-Inserts problemlos möglich. Die benötigten JSON-Objekte müssen lediglich konstruiert werden, anschließend gibt man den Index an, in den man dieses Objekt einfügen will. Den genauen Bulk-Insert sieht man in unserer Query-Textdatei.

Read

Bei den *Read*-Queries sind dagegen einige Probleme aufgetreten. Einige Abfragen waren verglichen mit äquivalenten SQL-Queries viel umständlicher, bei wiederum anderen mussten "Abstriche" bei der Lösung gemacht werden. Es war beispielsweise nicht immer möglich, das Ergebnis wie gefordert zu sortieren. Auch die einzelnen Ergebniswerte waren durch die JSON-Struktur nicht immer gut lesbar und mussten zusätzlich gefiltert werden, um die Anzeige einzugrenzen, da Elasticsearch standardmäßig immer das gesamte Objekt ausgibt, das als Match für eine Abfrage gefunden wurde.

Bei der Abfrage einzelner Felder gab es mit Elasticsearch keine Probleme. Diese waren leicht zu aggregieren. Somit waren die Abfragen aus Aufgabenteil a), b), d), e) und k) relativ leicht zu realisieren. Für andere Teilaufgaben mit Abfragen bestand eines der Problem darin, dass Aggregationen nicht einfach sortiert ausgegeben werden können. So existierte bei Aufgabenteil c) die Problematik, dass die Kursleiter nur entweder sortiert oder aggregiert und damit ohne Duplikate ausgeben werden konnten. Auch bei Teil m) konnte zwar der Durchschnitt der Gehälter ausgegeben, aufgrund der Aggregation aber nicht sortiert werden. Eine weitere Problemstellung war die fehlende Möglichkeit *Joins* zu nutzen. Somit bestand bei Aufgabenteil f) nur die Möglichkeit, die *VorNr* der vorausgesetzten Kurse auszugeben. Auch die Deklarierung eines Feldes als *NULL* ist in dieser Datenstruktur mit Elasticsearch nicht möglich, da hier immer ein leeres Array ausgegeben wird, sobald es keine Voraussetzung für einen Kurs gibt.

In Aufgabenteil g) konnten wir von der Möglichkeit profitieren, Scriptfelder zu nutzen. Dies eröffnete uns die Möglichkeit, Vergleiche zwischen Feldern aufzustellen:

```
GET /angebot/_search?filter_path=hits.hits.*.Teilnehmer
{
    "query": {
        "bool": {
            "must": [ {
                "script": {
                    "script": {
                        "source": "doc['Ort'].value == doc['Teilnehmer.Ort']" } } } ] } } }
```

Hier werden alle Datensätze ausgegeben, bei denen *Ort* und *Teilnehmer.Ort* identisch sind. Durch *filter_path* wird auf die Teilnehmerdaten gefiltert.

Aufgabenteil j) bestand darin, alle Teilnehmer und Kursleiter mit dem Namen “Meier” zu finden. Dabei trat die Problematik auf, dass Elasticsearch immer den ganzen Ergebnis-Datensatz anzeigt und somit bei einem einem Match mit einem Teilnehmer oder einem Kursleiter immer auch die dazugehörigen anderen Teilnehmer sowie den Kursleiter ausgegeben werden. Wir können mit Elasticsearch also nur die Kursleiter und Teilnehmer von Angeboten abfragen, bei denen entweder ein Teilnehmer oder der Kursleiter Meyer heißt. Für diesen Anwendungsfall bietet Elasticsearch nicht die benötigten Möglichkeiten von SQL, wie *distinct* und *union*. Eine mögliche Lösung für dieses Problem wären zwei aggregierte Abfragen, die dann in einem darüberliegenden Programm zusammengefügt und weiterverarbeitet werden.

Weiterhin als problematisch zu betrachten ist die Tatsache, dass Anfragen schnell sehr tief geschachtelt und unübersichtlich werden. Bei Aufgabenteil l) zum Beispiel wurde mithilfe von Scriptfeldern folgende Lösung erzielt:

```
GET angebot/_search?filter_path=aggregations.*.*.buckets.key,
aggregations.*.*.buckets.Vorauscount.value
{
    "aggs": {
        "Kurs.Titel": {
            "nested": { "path": "Kurs" },
            "aggs": {
                "Kurs.Titel": {
                    "terms": { "field": "Kurs.Titel" },
                    "aggs": {
                        "Vorauscount": {
                            "bucket_script": {
                                "buckets_path": {
                                    "vCnt": "Kurs.Voraus._count",
                                    "kCnt": "_count" },
                                "script": "params.vCnt / params.kCnt" } },
                        "Kurs.Voraus": {
                            "nested": {
                                "path": "Kurs.Voraus" } },
                        "voraus_bucket_filter": {
                            "bucket_selector": {
                                "buckets_path": {
                                    "vCnt": "Kurs.Voraus._count",
                                    "kCnt": "_count" },
                                "script": "params.vCnt / params.kCnt >= 2"
                             } } } } } } } }
```

Auch hier fehlt erneut die Sortierung. Auch das Zählen der Voraussetzungen war schwierig zu realisieren, da vergleichbare Operationen wie *countdistinct* nicht genutzt werden konnten, sondern die Berechnungen mittels Script selbst durchgeführt werden musste.

Update

Für die Update-Queries war es einfach, die Datensätze abzufragen, die verändert werden mussten. Die Schwierigkeit bestand vielmehr darin, diese zu verändern.
Im Aufgabenteil b), hier sollten lediglich die Orte “Kiel” durch “Lübeck” ersetzt werden, war dies kein Problem. Da Elasticsearch jedoch keine Funktionen für die Datumsmanipulation mitliefert, ist es nur sehr kompliziert mit einigen Extrafunktionen möglich, das Jahr von 2018 auf 2019 zu verändern. Dies sieht man auch an unserem Ergebnis zu diesem Aufgabenteil. Es wäre einfach möglich, lediglich das Jahr in das Feld zu schreiben und somit mehrere Felder für das vollständige Datum zu haben. Wie man an der obigen Abfrage sieht, ist es für Feldupdates einfach, die Datensätze zu finden und zu überschreiben. Eine Manipulationsmöglichkeit der bisherigen Datensätze bietet Elasticsearch nicht direkt an.

```
POST angebot/_update_by_query
{
    "query": {
        "bool": {
            "must": {
                "match": {
                    "Ort": "Kiel" } } } },
    "script": "ctx._source.Ort = 'Lübeck'" }
```

Delete

Die hier existenten Probleme waren eher indirekte Resultate aus den Abfragen der zu löschenden Datensätze. Diese Abfragen weisen natürlich die gleichen Problematiken auf, wie zuvor schon die *Read*-Queries. Ein Beispiel für eine *Delete*-Query aus Aufgabenteil a):

```
POST angebot/_update_by_query
{
    "query": {
        "nested": {
            "path": "Kurs",
            “query": {
                "bool": {
                    "must": [ {
                        "match": {
                            "Kurs.Titel": "C-Programmierung" } } ] } } } },
    "script": "ctx._source.Kurs.remove('KursLit')" }
```

Erfahrungen und Probleme

Einige Probleme und Erfahrungen wurden bereits im vorherigen Kapitel detailliert erklärt. In diesem Kapitel sollen diese noch einmal allgemein zusammengefasst werden.

Für die Übertragung von Daten aus einem relationalen Datenbankmodell ist eine deutliche Ummodellierung nötig, um ansatzweise die gleichen Abfragemöglichkeiten zu erlangen, wie sie SQL bietet.
Hier mussten mehrere Versuche mit verschiedenen Ansätzen unternommen werden, bis ein für diese Datenbank sinnvolles Datenmodell gefunden wurde, auch wenn dieses schlussendlich für diese Art der Daten nicht ideal ist.
Desweiteren muss man sich zunächst einmal bewusst werden, dass in Elasticsearch keine *Joins* möglich sind. Es gibt keine ähnlichen Konstrukte, die man zur Umgehung des Problems anwenden könnte. Somit wird eine doppelte Datenhaltung unvermeidlich, um sich bei den Queries nicht zu stark einzuschränken.

Wir haben weiterhin die Erfahrung gemacht, dass die Daten nicht so übersichtlich, wie man es aus SQL kennt, zurückgegeben werden, was die Lesbarkeit erschwert. Das von Elasticsearch verwendete Datenformat bietet aber andere Vorteile. Es liefert mehr Informationen, ermöglicht Arraystrukturen sowie *nested objects* und kann in Programmen gut weiterverarbeitet werden.

Problematisch war auch, dass sich die Funktionalitäten und Möglichkeiten zwischen den verschiedenen Versionen zum Teil stark unterscheiden, sodass man häufig auf Lösungen stößt, die nicht für die aktuelle Version geeignet sind. Elasticsearch hat im vergleich zu SQL noch keine festen Standards, sondern durchlebt (noch) stetige Veränderungen.

Die Vorteile, die Elasticsearch bietet, sind für die hier verwendeten Daten völlig irrelevant. Die Zielsetzung ist durchaus eine Andere. Elasticsearch bietet eine sehr gute Volltextsuche und schnelle Suchabfragen mit Scoring-Werten, also die Möglichkeit zu entscheiden, wie gut ein Datensatz zu der gestellten Abfrage passt - also nicht ob es ein Ergebnis richtig ist oder nicht. Unsere gegebenen Daten sind allerdings sehr strukturiert. Wichtiger ist, dass die Datenbank “gut zu pflegen” ist und die Daten normalisiert gespeichert werden. Volltextsuche oder besonders schnelle Abfragen auf großen Datenmengen werden hier gar nicht benötigt.

Beim Updaten der Daten kam zusätzlich das Problem hinzu, dass die vorhandenen Daten nicht gut manipuliert, sondern nur gut ersetzen werden können.

Fazit

Zusammenfassend kann man sagen, dass diese Datenbank bzw. NoSQL-Datenbanken eine Alternative zu relationalen Datenbanken mit SQL darstellen, diese aber nicht gleichwertig ersetzen können. Sie ergänzen sich eher. Es gibt Anwendungsfälle in denen einzelne NoSQL-Datenbanken Vorteile bringen, die in einem spezifischen Anwendungsfall benötigt werden. Es gibt aber auch Fälle, in denen gerade das relationale Modell gut genutzt werden kann. In diesen Fällen sollte man keine NoSQL-Datenbank benutzen werden, sondern auf die ausgereifteren, relationalen Datenbanken zurückgreifen.
In jedem Fall muss man sich im Vorfeld genaue Gedanken darüber machen, was genau man mit den jeweiligen Daten machen möchte und welche Eigenschaften die Datenbank dafür haben muss.

Die uns vorliegenden, sehr gut strukturierten Daten sind ein optimaler Fall für relationale Datenbanken. Sie benötigt keine Volltextsuchen oder Ähnliches. Das Ziel sollte es sein, möglichst einfache Verknüpfungen zwischen den einzelnen Teilen herzustellen, ohne Daten doppelt zu speichern und den Pflegeaufwand somit erhöhen zu müssen.

Aus dieser ausgeprägten Struktur sowie den recht spezifischen Aufgabestellungen resultieren auch unsere Probleme mit den komplexen Queries. Fehlende *Joins* bzw. *Unions* machen es unmöglich, die Daten auf mehrere Indizes zu verteilen. Die Kombination von Aggregation, Sortierung und Abfrage ist nur bedingt möglich und für Abfragen mit Elasticsearch fehlt eine klare Struktur der Datenbank wie in SQL mit entsprechenden Operationen wie *Select*, *From* und *Where*, wodurch Anfragen schnell verschachtelt und unübersichtlich werden.

Elasticsearch bietet aber auch viele Vorteile. So kann man Daten schnell und einfach in die Datenbank laden und Datensätze über einzelne Felder leicht finden. Es sollte betont werden, dass Elasticsearch eine Suchmaschine und keine klassische Datenbank im eigentlichen Sinne ist. Sie ermöglicht daher ein sehr leichtes Suchen von Datensätzen. Auch die Volltextsuche bietet viele Möglichkeiten. Aufgrund der Art unserer Daten kämen wir jedoch nie dazu, diese Vorteile zu nutzen. Viele einfache Operationen - aggregieren mit *Count*, *Min*, *Max* oder einem Durchschnitt oder auch Datumsangaben und der Möglichkeit, nach Feldern sortieren - sind zwar gegeben; Wenn es jedoch darum geht, Daten aufzubereiten, zu kombinieren oder auch die oben genannten Möglichkeiten gleichzeitig zu verwenden, stößt man auf viele Problemstellungen und kann diese entweder gar nicht oder nur unvollständig lösen. Außerdem bekommt man stets ganze Datensätze als Ergebnis, sodass erweiterter Aufwand für das Filtern der Resultate erzeugt wird, was den Arbeitsaufwand deutlich erhöht.

Quellen

- https://de.wikipedia.org/wiki/Elasticsearch
- https://de.wikipedia.org/wiki/Apache_Lucene
- Kuc, R., & Rogozinski, M. (2013). *Elasticsearch server*. Packt Publishing Ltd.
- https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started.html# (elastic docs)
- https://www.elastic.co/de/products/elasticsearch