Data Engineering

Elasticsearch

Elasticsearch

- Suchmaschinen & Dokumentenorientierte, verteilte Datenbank
- Basiert auf Apache Lucene (opensource Search Engine Library)
- Speichert Schema-Freie JSON-Dokumente
- Geschrieben in Java
- Meist genutzte Suchmaschine
- Open source Variante: OpenSearch
- Teil des „ELK-Stacks“: Elasticsearch, Logstash, Kibana

Anwendungsfälle

- **Observability**: Überwacht Logs, Metriken und Anwendungsergebnisse in Echtzeit.
- **Suche**: Unterstützt Volltext-, Vektor-, semantische und geobasierte Suche sowie KI-unterstützte Suchen.
- **Security**: Wird für Sicherheitsinformations- und Ereignismanagement (SIEM), Endpointsecurity & threat hunting genutzt.

Konzepte

Indices

- „Index“ ist in Elasticsearch vergleichbar mit einer Tabelle in einem RDBMS
- Logische Einheit in der Daten mit ähnlicher Charakteristik gespeichert werden
- Indizes stellen eine Sammlung von **Dokumenten** dar
- Besitzen einen einzigartigen Namen
- Später mehr zu Indices

Documents

- Elasticsearch speichert Daten in JSON-Documents
- Ein Document entspricht in etwa einer Zeile in einem RDBMS
- Ein Dokument besteht aus beliebig vielen *Fields*
- *Fields* sind Key-Value Paare
- Jedes Dokument besitzt eine eindeutige ID, die selbst erstellt werden kann

Metadata fields

- Jedes Dokument besitzt neben *Fields* in denen die Nutzdaten gespeichert sind auch *Metadata Fields*
- Sind durch einen „_“ gekennzeichnet
- Typische Metadata fields sind: _id, _index

Mappings

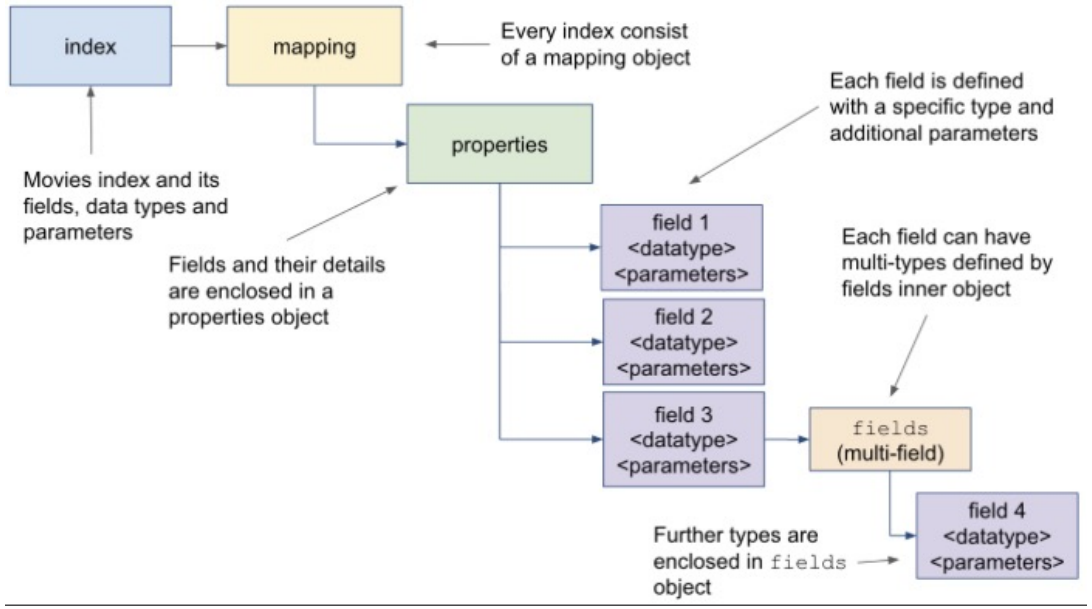
- Elasticsearch ist schemaless
- Die Definition eines Schemas kann trotzdem sinnvoll und notwendig sein => geschieht mit Hilfe von *Mappings*
- Mappings stellen keine Konsistenzbedingung dar => Dokumente mit abweichender Datenstruktur können ohne Probleme gespeichert werden
- Mapping bezieht sich auf einen Index

Mappings – Warum?

- Mappings geben vor wie Felder in einem Index indexiert werden
- Beschreiben die zugrundeliegende Datentypen
- Beispiel: Eine IP-Adresse kann als String oder im Datentyp „IP“ gespeichert und indexiert werden.
- Besonders wichtig bei genesteten Dokumenten

Mappings

- **Dynamisches Mapping**: Elasticsearch erkennt automatisch Datentypen und erstellt die Zuordnungen. Dies ermöglicht einen schnellen Einstieg, kann aber zu weniger optimalen Ergebnissen führen, da die Felder automatisch zugewiesen werden.
- **Explizites Mapping**: Benutzer legen Datentypen manuell fest. Dies wird für produktive Anwendungsfälle empfohlen, da man vollständige Kontrolle darüber hat, wie Daten indexiert werden, um sie optimal auf spezifische Anforderungen abzustimmen.



Quick Start

API

- Die Elastic API basiert auf REST
- Darüber lässt sich der Cluster verwalten, Daten lesen, schreiben und Indizes erstellen
- Die Query Language von Elasticsearch heißt Query DSL
- Seit diesem Jahr gibt es eine neue Query Language namens ES|QL
- Es gibt auch eine SQL Variante für Elastic

Index anlegen

- Ein Index kann einfach mittels eines PUT Requests angelegt werden
- Hier legen wir einen Index mit dem Namen „Books“

```
PUT /books
```

Daten schreiben

- Elasticsearch basiert auf JSON
- Jeder Datensatz ist ein “Dokument“
- Jedes Dokument lebt in einem Index
- REST-Typisch nutzt man POST

```
POST books/_doc
{
  "name": "Snow Crash",
  "author": "Neal Stephenson",
  "release_date": "1992-06-01",
  "page_count": 470
}
```

BULK Requests

- HTTP Request haben einen gewissen Overhead, deswegen ist die Nutzung von BULK Requests sinnvoll um mehrere Dokumente in einem Request anzulegen

```
curl -X POST "localhost:9200/_bulk?pretty" -H 'Content-Type: application/json' -d'
{ "index" : { "_index" : "books" } }
{"name": "Revelation Space", "author": "Alastair Reynolds", "release_date": "2000-03-15", "page_count": 585}
{ "index" : { "_index" : "books" } }
{"name": "1984", "author": "George Orwell", "release_date": "1985-06-01", "page_count": 328}
{ "index" : { "_index" : "books" } }
{"name": "Fahrenheit 451", "author": "Ray Bradbury", "release_date": "1953-10-15", "page_count": 227}
{ "index" : { "_index" : "books" } }
{"name": "Brave New World", "author": "Aldous Huxley", "release_date": "1932-06-01", "page_count": 268}
{ "index" : { "_index" : "books" } }
{"name": "The Handmaids Tale", "author": "Margaret Atwood", "release_date": "1985-06-01", "page_count": 311}
'
```

Mappings Anfrage

- Elastic erstellt automatisch Mappings für die Felder
- Der Mappings-Endpunkt gibt die Möglichkeit die erstellten Mappings anzufragen und zu verändern
- Mappings existieren immer auf Index-Ebene

```
GET /books/_mapping
```

Mappings

```
POST books/_doc
{
  "name": "Snow Crash",
  "author": "Neal Stephenson",
  "release_date": "1992-06-01",
  "page_count": 470
}
```

```
{
  "books": {
    "mappings": {
      "properties": {
        "author": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
        "name": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        },
```

Mappings festlegen

- Mappings können unter anderem beim Anlegen eines Index festgelegt werden
- HTTP Request legt einen neuen Index mit definiertem Mapping an

```
PUT /my-explicit-mappings-books
{
  "mappings": {
    "dynamic": false,  (1)
    "properties": {  (2)
      "name": { "type": "text" },
      "author": { "type": "text" },
      "release_date": { "type": "date", "format": "yyyy-MM-dd" },
      "page_count": { "type": "integer" }
    }
  }
}
```

Daten lesen

- Queries werden in Elasticsearch ebenfalls über HTTP Requests durchgeführt
- REST Typisch geschieht das lesen mittels GET Request auf den Index Endpunkt

```
GET books/_search
```

Match Query

- Eine von vielen Query-Typen ist die sogenannte “Match“ Query
- Sie liefert Dokumente zurück die definierten Bedingungen entsprechen
- Vergleichbar mit *where* in SQL

```
GET books/_search
{
  "query": {
    "match": {
      "name": "brave"
    }
  }
}
```

Verteiltes System

Verteiltes System

- Elasticsearch ist ein verteiltes System, welches keine Skalierungslimits hat
- Ein Elasticsearch Cluster benötigt **mindestens** drei Nodes
- In Elasticsearch gibt es verschiedenen Rollen, die ein Node einnehmen kann: Ein Node kann mehrere Rollen gleichzeitig einnehmen
- Die wichtigste Rolle: Den Master Node gibt es nur einmal
- Der Master wird von den anderen Nodes gewählt
- Neben dem Master Node muss jeder Cluster über *Data Nodes* verfügen

Rollen

- Ein Node kann mehrere Rollen gleichzeitig einnehmen
- Wird in der Konfiguration des Nodes festgelegt
- Die wichtigste Rolle: Den Master Node gibt es nur einmal
- Der Master wird von den anderen Nodes gewählt
- Neben dem Master Node muss jeder Cluster über *Data Nodes* verfügen
- *Rollen sind wichtig für die Cluster Architektur und sind elementar für die Optimierung großer Cluster!*

Rollen – Überblick

- master
- data
- data_content
- data_hot
- data_warm
- data_cold
- data_frozen
- ingest
- ml
- remote_cluster_client
- transform

Master Node

- Zuständig für Cluster-weite Aufgaben wie:
  - Erstellen und Löschen eines Index
  - Hinzufügen oder entfernen von Nodes
  - Management des Cluster Sates
  - Shard allocation
- Jedes Cluster besitzt einen Master Node
- Wird von den Master-Eligible Nodes des Clusters gewählt

Master-eligible Nodes

- Nodes die zum Master gewählt werden können
- Nodes die an der Wahl des Masters teilnehmen
- In großen und komplexen Elastic Installationen kann es sinnvoll sein dedizierte master-eligible Nodes zu definieren.
  - Einzige Aufgabe: Eventuell Master zu werden: Halten keine Daten und haben auch keine weitere Rollen (außer Coordinating dazu gleich mehr)
- Voting-only master-eligible nodes
  - Nehmen an der Wahl des Masters teil, stehen selbst aber nicht zur Wahl
  - Dies ist wichtig weil die Wahl nur dann möglich ist, wenn es ein Quorum gibt (gleich mehr)

Data Nodes

- Data Nodes speichern Shards (teile eines Index), welche die Daten halten
- Data Nodes sind für CRUD Operationen, Suchen und Aggregationen zuständig
- Data Nodes benötigen viel I/O, CPU & RAM
- Dedizierte Data Nodes ergeben in großen Clustern Sinn
- Multi-Tier Architecture: Spezialisiertere Data Node Rollen wie:
  - data_content, data_hot, data_warm, data_cold, data_frozen

Coordinating Nodes

- Coordinating Nodes sind für die Koordination von Queries verantwortlich
- Load Balancer: Verteilt Anfragen an die Nodes, die Daten halten
- Koordiniert Anfragen, die Daten aus mehreren Shards benötigen
- Führt Aggregationen auf den Teilergebnissen der Nodes aus
- **Jeder Node ist automatisch Coordinating Node**
- Nodes ohne zugewiesene Rolle sind **ausschließlich** Coordinating Node (kann in manchen Fällen sinnvoll sein)

Discovery and Cluster Formation

- Ein Cluster besteht aus einer Summe von Elasticsearch Nodes (prod ready min 3) Doch wie formt sich ein Cluster?
- Bei der Formierung eines Clusters werden folgende Schritte durchlaufen:
  - Discovery: Die Nodes die den Cluster formen müssen sich initial finden
  - Master Election: Ein Master muss gewählt werden
  - Bootstrapping: Händische Definition der Master-eligible nodes
  - Publishing Cluster State: Master Node informiert regelmäßig alle Nodes über den Cluster State
  - Cluster Fault Detection: Health Checks um fehlerhafte Nodes zu detektieren

Cluster State

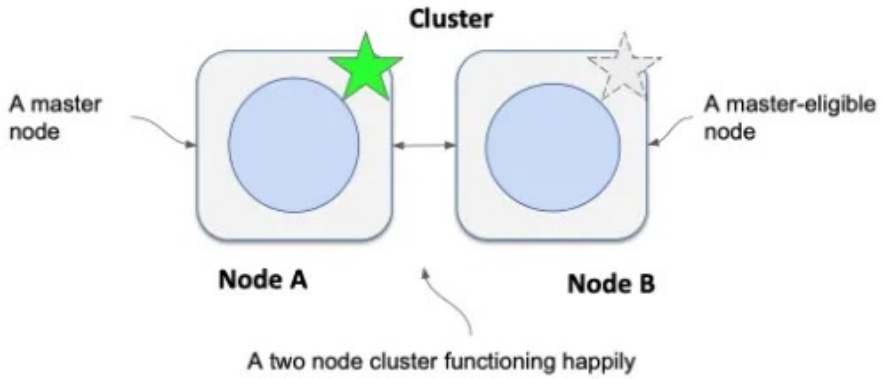
- **Cluster State**: Eine interne Datenstruktur, die Informationen über alle Knoten im Cluster, Cluster-Einstellungen, Index-Metadaten und den Status aller Shard-Kopien speichert.
- **Master-Knoten**: Sorgt dafür, dass jeder Knoten den gleichen Cluster-Zustand hat.
- **Cluster State API**: Ermöglicht das Abrufen dieses Zustands zu Diagnosezwecken. Standardmäßig werden Anfragen an den Master-Knoten weitergeleitet.
- **Hinweis**: Bei großen Clustern kann die API-Antwort viel Daten enthalten und bei wiederholter Nutzung Instabilität verursachen.

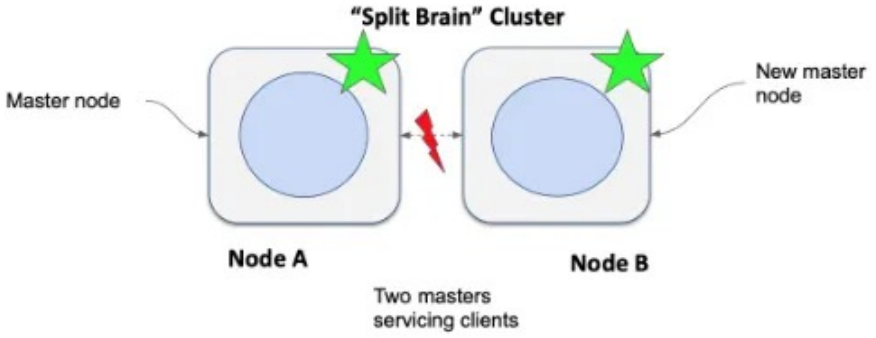
Quorum

- **Quorum-basiertes Entscheidungssystem**: Mindestens die Mehrheit der Master-wählbaren Knoten (N/2 + 1) muss erreichbar sein, um einen neuen Master zu wählen.
- Quorum wird automatisch gebildet aber: Anzahl der master-eligible Nodes muss stimmen:
  - `Minimum number of master nodes = (number of master-eligible nodes / 2) + 1`
- Jeder Cluster muss also mindestens drei master-eligible nodes besitzen => Wieso?

Split Brain

- **Definition**: Das Split-Brain-Problem tritt auf, wenn eine Netzwerkpartition dazu führt, dass zwei getrennte Master-Knoten in verschiedenen Cluster-Partitionen gewählt werden.
- **Auswirkungen**: Jede Partition akzeptiert unabhängig Anfragen, was zu inkonsistenten Datenzuständen führt, da verschiedene Kopien der Daten bearbeitet werden.
- **Beispiel**: Ein Cluster mit zwei Knoten, bei dem beide Knoten nach einer Trennung glauben, sie seien Master, und unterschiedliche Indexdaten verarbeiten.
- Deswegen: Mindestens drei potenzielle Master-Nodes pro Cluster





Verhalten im Fehlerfall

- **Fehlendes Quorum**: Ohne Quorum lehnt das Cluster Anfragen ab, um Dateninkonsistenzen zu vermeiden.
- **Empfohlene Überwachung**: Der /nodes-Endpunkt hilft, Split-Brain-Situationen frühzeitig zu erkennen, indem Unterschiede im Cluster-Zustand der Knoten sichtbar werden.

Wie werden Daten verteilt gespeichert?

Zur Erinnerung

- Das verteilte Speichern von Daten nennt sich Sharding
- Sharding passiert in elasticsearch auf Basis des Index
- **Gründe für Sharding:**
  - **Skalierbarkeit:** Ein Datenbanknode kann nur eine bestimmte Anzahl an Daten halten
  - **Performance:** Viele Anfragen können verteilt schneller beantwortet werden
  - **Availability:** Bei Teilausfällen des Systems sind Daten weiterhin verfügbar
- These der letzten Vorlesung: Dokumentenorientierte Datenbanken sind optimal zu skalieren => **Wieso?**

Shards in elasticsearch

- Shards existieren auf Index Level
- Halten einen Teil der Daten eines Indexes
- Jeder Shard ist eine Lucene Instanz (Suchmaschine unter Elasticsearch)
- Shards werden über Nodes verteilt gespeichert

Primaries and Replicas

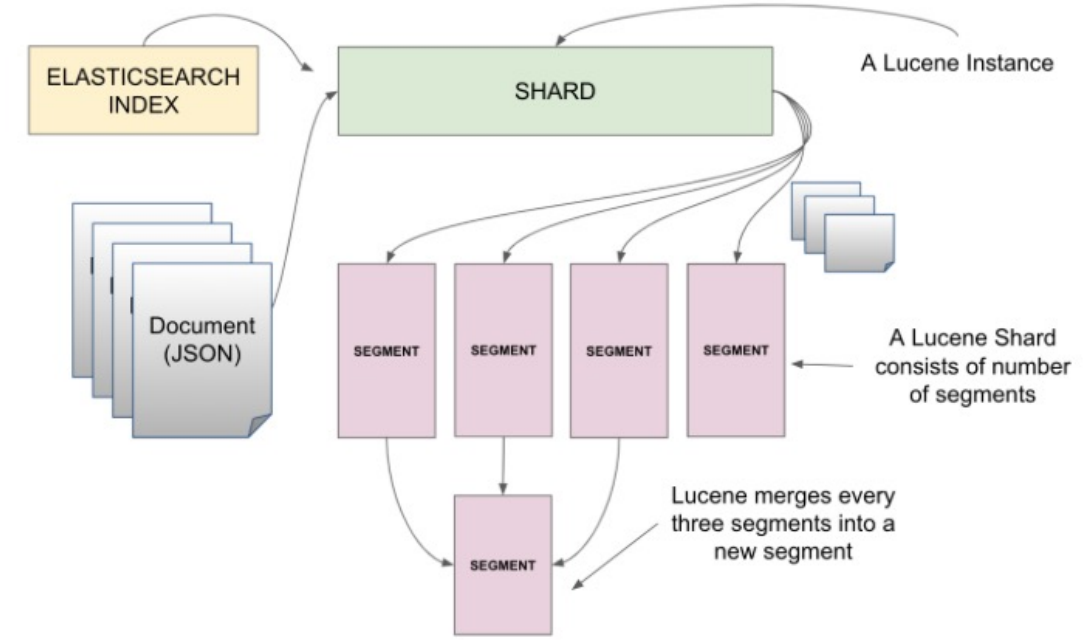
- **Primaries:** Primäres Abbild der Daten nur hier wird geschrieben
- **Replicas:** Beliebige Anzahl an Kopien der Primaries, lesende Zugriffe möglich. Primär für Availability
  - Ergeben nur dann Sinn wenn sie auf einem anderen Node liegen
- Anzahl der Primaries und Replikas muss zwingend auf die Anzahl der Nodes im Cluster abgestimmt sein => Sonst unnötiger Overhead ohne Mehrwert
- Definition der Shards bei Erstellung der Indizes

Bestandteile eines Shards

- Besteht aus Segments (Lucene Indices)
- Segements sind Immutable → können nicht geupdated werden
- Segments enthalten mehrere Dokumente

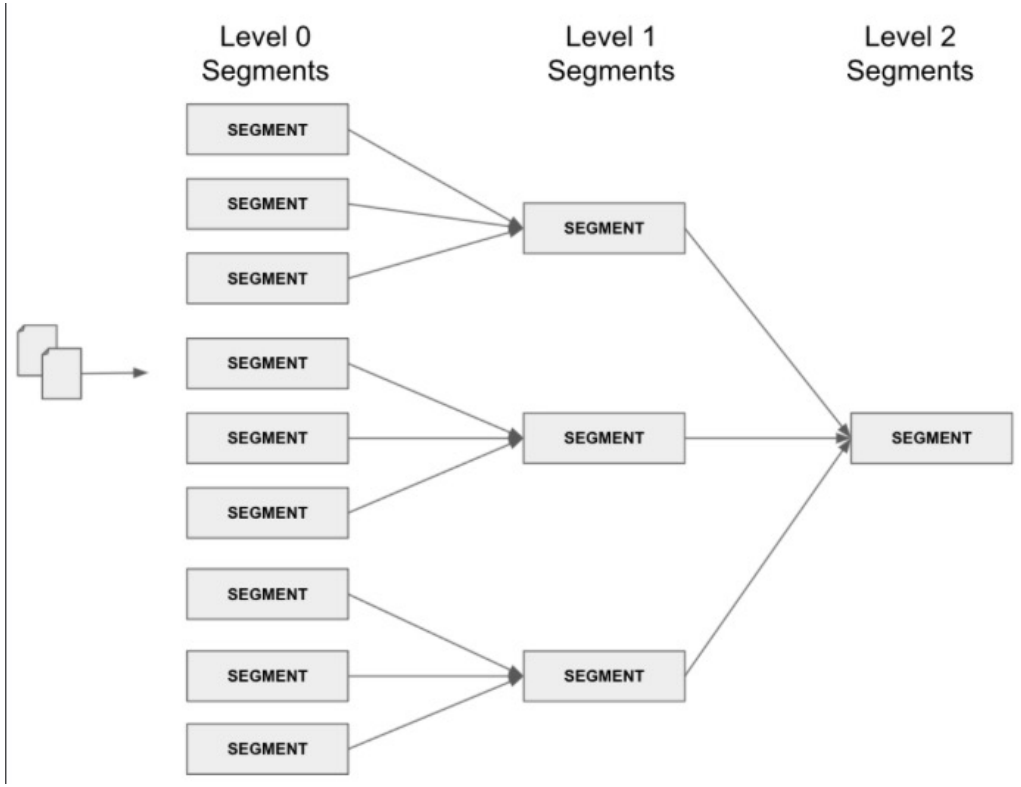
Segments

- Da Segments immutable sind: Kein Update existierender Segments
- Daten werden zunächst in einen In Memory Buffer geschrieben
- I/O ist eine teure Operation: Lucene führt periodisch (default 1 Sekunde) Updates aus um neue Daten zu persistieren
- Bei Update: Neue Daten werden aus Buffer in neue Segments geschrieben



Segments

- Segments sind zu Beginn sehr klein => schlecht für die Performance
- Werden mit der Zeit in größere Segments zusammengefasst
- Das verhalten ist auch in anderen Dokumentenorientierten Datenbanken üblich. Siehe: LSM Tree
- Segments enthalten neben den Daten auch den Inverted-Index (später mehr)



Löschen eines Dokuments in einem Segment

- Wird ein Dokument gelöscht, wird das korrespondierende Segment als gelöscht markiert (Tombstone)
- Ein neues Segment mit den übrig gebliebenen Daten erstellt
- Das übrig gebliebene Segment wird periodisch in cleanups gelöscht
- Löschen von Daten führt temporär zu mehr Speicherbedarf

Update eines Dokuments in einem Segment

- Segments sind immutable: Keine Updates in Place
- Kopie des Segments notwendig für ein Update
- Löschen des alten Segments analog zu Löschoperation
- **Updates sind sehr teuer**