Database Management Systems

ElasticSearch

Search in a nutshell

1. Take a query string
2. Match it against a document collection
   - Perform full text search, handle synonyms
3. Calculate a set of relevant results
   - Score documents by ***relevance***
4. Display a sorted list

ElasticSearch

- Real-time distributed search and analytics engine
- Scalable and efficient data exploration
  - Full-text search
    - Highlighted search snippets, search as you type, did-you-mean, more-like-this
  - Structured search
  - Analytics
    - Real-time query answers on mixed data types (e.g., text and structured data)

elastic

ElasticSearch

- Popular examples
  - *GitHub* uses ElasticSearch to query 130+B lines of code.
  - *Wikipedia* provides full-text search with highlighted snippets
  - *StackOverflow* combines both full-text and geolocation queries for recommending related questions and answers

ElasticSearch

- Document-oriented (JSON) search engine
  - Complex data structures that may contain dates, geo locations, text, other objects, arrays of values
- Built on Lucene search engine library
  - Documents are indexed and searchable
- Highly available and horizontally scalable

ElasticSearch strong points

- The **horizontal scaling** capabilities of ElasticSearch make it suitable for a great variety of applications

- Its **RESTful API** allows programmers to write most of the operations in any programming language

- **JSON-based** interactions make it machine- and human-friendly

- Any modification to stored (indexed) documents is recorded in transaction logs that are replicated in multiple nodes to **avoid data loss**

Data representation

Parallel to relational representation

ElasticSearch: `field` ⇔ **SQL:** `column`

- Data is stored in *named entries* belonging to a variety of data types
- SQL calls such an entry a `column` while in ElasticSearch it is called `field`

In ElasticSearch (similarly to other NoSQL databases) a field can contain *multiple* values of the same type (list of values)

Parallel to relational representation

ElasticSearch: `document` ⟺ **SQL:** `row`

- Data objects are represented as rows (SQL) or documents (ElasticSearch)
- Columns and fields are part of a row (SQL) or a document (ElasticSearch)

Row format in SQL is **strict** and follows a predefined schema.
Documents are more **flexible** and can contain a variety of fields (they do not follow a strict schema)

Parallel to relational representation

ElasticSearch: `index` ⇔ **SQL:** `table`

An `index` is like a `table` in a relational database

ElasticSearch: `cluster` ⇔ **SQL:** `database`

In Elasticsearch the indices are grouped in a cluster.

Recap

ES	cluster	index	document	field
SQL	database	table	row	column

The *index* term

The *index* term is overloaded

- *(noun)* An index stores a collection of documents
- *(verb)* To index a document means to insert a document in an index
  - If the document already exists, it is replaced

The *index* term

The *index* term is overloaded

- *(inverted index)* Additional structure that accelerates data retrieval
  - Similar to a traditional relational index
  - *Every field* in a document is indexed in ES
    - All inverted indices are used during search
  - Non-indexed fields (if any) are not searchable

Document

- A document is the top-level (root) object serialized into JSON
- It is uniquely identified by the pair
  - *Index:* where the document (object) is stored
  - *Id:* the identifier of the document
    - Can be provided or uniquely generated by ES

Querying and searching

Searching

Search options

- **[Hard]** Structured query on specific fields, possibly sorted
  - similar to SQL query
- **[Soft]** Full-text query
  - finds all documents matching the search keywords
  - returns them sorted by *relevance*
- Combination of the two

Key concepts

- Mapping
  - How the data in each field is interpreted
  - ES dinamically generates a mapping by "guessing" data types
    - E.g. it may recognize a `date` type
- Analysis
  - How full text is processed to make it searchable
- Query DSL (Domain Specific Language)
  - Elastic Search query language

Exact values

- Traditional data types (e.g., integer, float, date, but also string)
  - A value must *match exactly* the query
    - Similar to SQL
  - Examples: date, user ID, but also exact strings such as username or email address
- Question answered

  "Does this document match the query?"

Full text

- Textual data, usually written in some human language
  - Typical search is within the textual field
  - Examples: text of a tweet, body of an email
- Question answered

  "*How well* does this document match the query?"
  - Notion of *relevance* of a document for a query

Full text

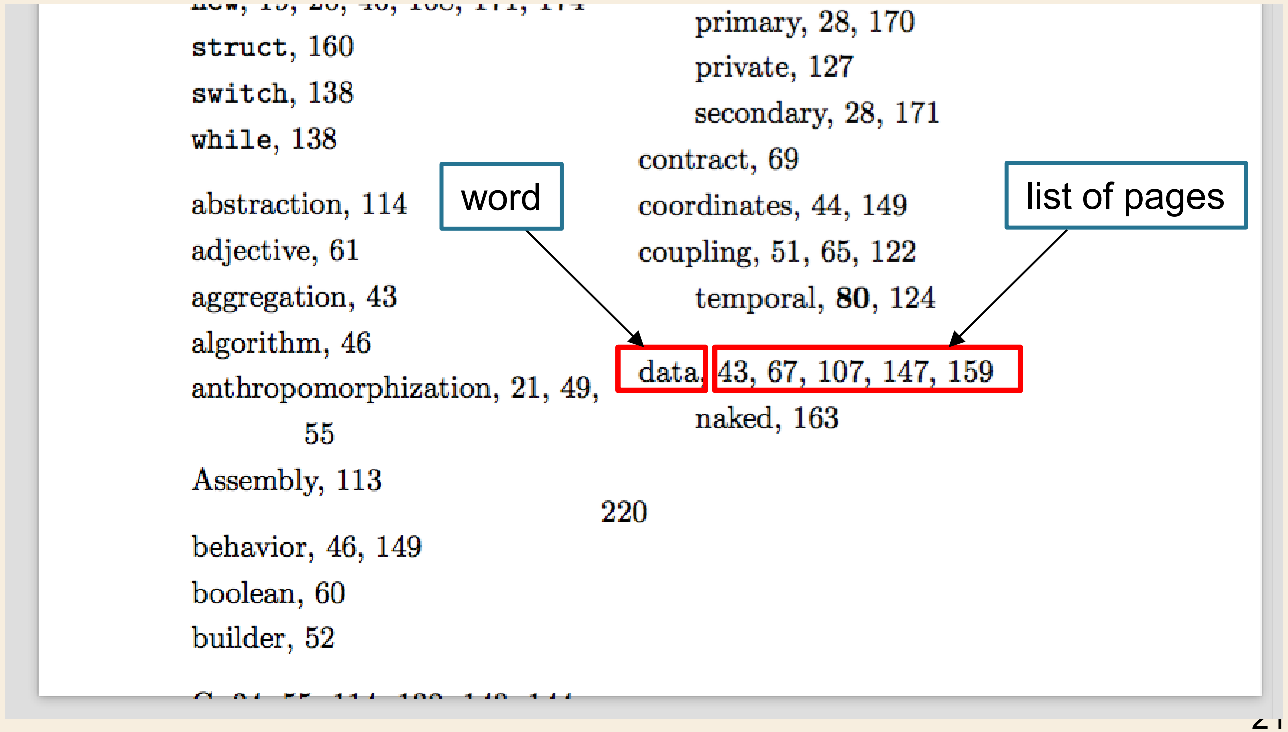
- For full text queries, it is also important understanding the intent
  - abbreviations
    - e.g., USA vs United States of America
  - singulars/plurals, verb conjugation
    - e.g., cat vs cats, does vs did vs to do
  - synonyms
    - e.g., game vs competition
  - order of words building a context
    - e.g., fox news hunting vs fox hunting news

Indexing full text

- ES builds an *inverted index* on every full-text field
  - Designed for fast full-text search
- Inverted index
  - List of all the unique words that appear in any document in the collection
  - For each word: list of the documents in which it appears

Indexing full text

- Similar to analytic indices in books


word
list of pages

struct, 160
switch, 138
while, 138

abstraction, 114
adjective, 61
aggregation, 43
algorithm, 46
anthropomorphization, 21, 49, 55
Assembly, 113

behavior, 46, 149
boolean, 60
builder, 52

primary, 28, 170
private, 127
secondary, 28, 171
contract, 69
coordinates, 44, 149
coupling, 51, 65, 122
temporal, **80**, 124
data, 43, 67, 107, 147, 159
naked, 163

220

Analysis

1. Tokenization of a block of text into individual terms suitable for an inverted index
2. Normalization into a standard form to improve their retrieval (or recall) in queries
   - Terms are not exactly the same, but similar enough to be still relevant
     - Lowercase vs uppercase
     - Stemming, i.e., reduction to the root form
       - e.g., cats vs cat
     - Synonym management

- For searching, both indexed text and query string must be analyzed in the same way

Analyzer

- Analyzers provide the following functions
  - Character filters: cleans the string before tokenization
    - e.g., converts & to and
  - Tokenizers: split the string into individual words
    - e.g., by considering white spaces or punctuation as separators
  - Token filters: operate on single terms
    - change terms (e.g., to lowercase)
    - remove (e.g., stopwords)
    - add terms (e.g., synonyms)
- Built-in analyzers are provided by ES

The concept of token will return with vector databases!

Filter vs Query

- A *filter* is used for fields containing exact values
  - It provides a boolean *matches/does not match* answer for every document
- A *query* is (typically) used for full-text search
  - It also asks the question: How *well* does this document match?
    - calculates how *relevant* each document is to the query
    - assigns it a relevance _score, which is later used to sort matching documents by relevance
  - The concept of relevance is well suited to full-text search
    - there is seldom a completely "correct" answer

Filter vs Query

- Filter execution is more efficient
- Filters are typically used to reduce the number of documents that have to be examined by a query
- Hint
  - use query clauses for *full-text* search or for any condition that should affect the *relevance score*
  - use filter clauses for everything else

ElasticSearch Query

- Expressed in Query DSL
- Submitted as formatted JSON in the body of an HTTP request
- Example: empty query
  - returns all documents in all indices

```
POST /_search
{}
```

- Search on a specific index

```
POST index1/_search
{}
```

Query DSL

- The top level field in an ElasticSearch query is always **`query`**
  - the query type is specified one level below

```
POST departments/_search
{
   “query”: {
       “match” : { “name” : ”John” }
   }
}
```

Query DSL

- The top level field in an ElasticSearch query is always **`query`**
  - the query type is specified one level below

```
POST departments/_search
{
   "query": {
       "match" : { "name" : "John" }
   }
}
```

  - the query operates on the department index
    - specified in the URI
  - it performs the **`search`** operation

Query DSL

- Query

  Find all the documents in the department index that have a field name containing the term John in it.

- Query type: match query

```
POST departments/_search
{
   “query”: {
      “match” : { “name” : ”John” }
   }
}
```

Compound queries

Compound queries are complex queries specifying multiple matching criteria

```
POST departments/_search
{
   "query": {
      "bool": {
         "should": [
            {"match": {"name": "John"}},
            {"match": {"name": "Mark"}}
         ],
            "minimum_should_match":1,
         "must":{
            {"match": {"title": "developer"}}
         }
         "must_not":{
            {"match": {“lastname": “Smith"}}
         }
      }
   }
}
```

bool specifies the compound query

Compound queries

```
POST departments/_search
{
   "query": {
      "bool": {
         "should": [
            {"match": {"name": "John"}},
            {"match": {"name": "Mark"}}
         ],
            "minimum_should_match":1,
         "must":{
            {"match": {"title": "developer"}}
         }
         "must_not":{
            {"match": {“lastname": “Smith"}}
         }
      }
   }
}
```

should specifies the OR condition

must corresponds to the AND condition

must_not specifies the NOT condition

The match query

- Can be used for both full-text and exact queries
- On a full-text field
  - it analyzes the query string with the correct analyzer before executing the search
  - it returns a relevance score `_score` for the search
- On an exact field or a `not_analyzed` string field
  - it searches the exact value
  - it returns a relevance score `_score` of 1
- When a bool query is specified on full-text fields
  - It combines the `_score` from each must or should clause that matches

Query DSL

- It is possible to specify multiple indices to be searched in the query URI

```
POST rooms,students/_search {...}
```

- When a number of documents can be returned as query result, by default the top 10 relevant results are returned

Earlier versions of ElasticSearch include index types that have been deprecated since version `7.0`.

Data modifications

Insert

Insert of a new single document is performed by means of a PUT operation
- Name of index
- JSON document to be indexed

```
PUT /index_name/<id>
{
    JSON document
}
```

`index_name:` name of the index in which the document should be inserted
`<id>:` optional parameter that associates the document with a specific identifier
- If the ID is not provided, ElasticSearch creates a unique identifier for the document (e.g., `W0tpsmIBdwcYyG50zbta`)

Update

- Documents in ES are immutable
  - To update a document, it is reindexed
- When a document is updated, ES
  1. Retrieves the old document
  2. Modifies the retrieved copy
  3. Deletes the old document
  4. Indexes the new document (the copy)
- Internally, the old version of the document is not deleted immediately
  - It is not accessible
  - Deleted documents are cleaned in background

Update

The update of a document is performed using a PUT request

- Name of the index
- Unique ID of the document
- the fields to be updated and the associated new values

```
PUT index_name/123/_update
{
  "color" : "red",
}
```

This update request modifies the document with ID=123 setting the value of the "color" field to "red".

Delete

The deletion of a document is performed using a DELETE request

- Name of the index
- Unique ID of the document

```
DELETE index_name/id
```

This operation removes a JSON document from the specified index

- Document removal is not immediate

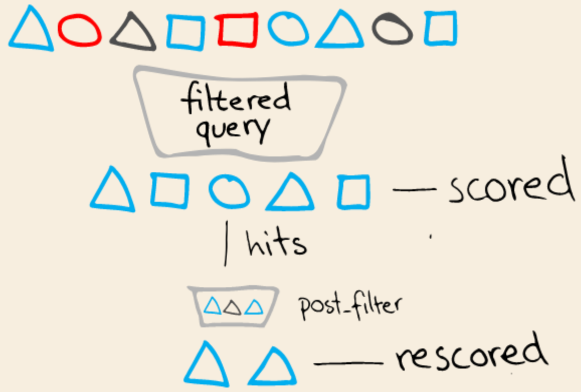
Results scoring

Relevance

- In ElasticSearch, relevance is represented by a value
  - floating-point number
  - computed for each document matching the query
  - stored as **`_score`** for each document in the search result
  - higher **`_score`** values correspond to more relevant documents
- Sorting by relevance is performed by considering the **`_score`** variable
  - by default, documents in a query result are sorted by descending value of the **`_score`** field

ElasticSearch Scoring

1. Compute matching results for the query
   - Compute relevance score for all documents in the query result
2. Select top relevance documents (hits)
   - Default is 10 hits (documents)
3. (optional) Rescore documents
   - more computationally expensive algorithm



Relevance score computation

- Need to compute the similarity between
  - The query
  - Each document
- Each document may contain a (different) subset of the query terms

Relevance score computation

1. Select documents matching the query
   - Boolean model
   - Fast computation
2. Evaluate the importance (weight) of each term in a document with respect to the query
   - Term importance evaluated with TF/IDF (Term Frequency/Inverse Document Frequency) score
   - Document and query are represented in vector form (Vector Space Model)
3. Evaluate the similarity of the vector representation of the query and the document

TF/IDF scoring function

The TF/IDF scoring function takes into account

Term frequency

- How often does the term appear in the field? The more often, the *more* relevant.

Inverse document frequency

- How often does each term appear in the index? The more often, the *less* relevant.

Field-length norm

- How long is the field? The longer it is, the less likely it is that words in the field will be relevant.

Term frequency

- Term frequency is defined by

$$\text{Tf(t in d)} = \sqrt{frequency}$$

  - frequency is the number of times the term t appears in document d

Inverse document frequency

- Inverse document frequency is defined by

$$Idf(t) = 1 + \log(numDocs/(docFreq + 1))$$

  - numDocs is the number of documents in the index
  - docFreq is the number of documents that contain the term

Field-length norm

- Field-length norm is defined by

$$\text{norm}(d) = 1 / \sqrt{numTerms}$$

  - numTerms is the number of terms in the field

TF/IDF scoring function

- The scoring function is based on a combination of the three factors
  - Term frequency
  - Inverse document frequency
  - Field-length norm
- They are calculated and stored at index time
- They are used to calculate the weight of a single term in a document
  - Other methods can be used
- Queries usually contain more than one term
  - Need a way to combine multiple terms

Vector Space Model

- It represents both query and document as (term) vectors
- It provides a way to compare a multi-term query against a document
- A query (or document) is represented as a vector
  - The vector size is the number of terms in the query
  - Each vector element is the weight of one term, calculated with TF/IDF scoring
- Vectors can be compared by measuring the angle between them

Measuring similarity

- Vectors can be compared by measuring the angle between them
  - Cosine similarity
- The angle between a document vector and a query vector is used to compute the similarity between a document and a query
  - It assigns to the document its relevance score for the query

VSM Example

Consider the query

- happy hippopotamus

Considering the TF-IDF heuristics

- happy is a common word and should have low weight (e.g., 2)
- hippopotamus is uncommon and should have a higher weight (e.g., 5)

The 2-dimensional vector associated to the query is

```
[2, 5]
```

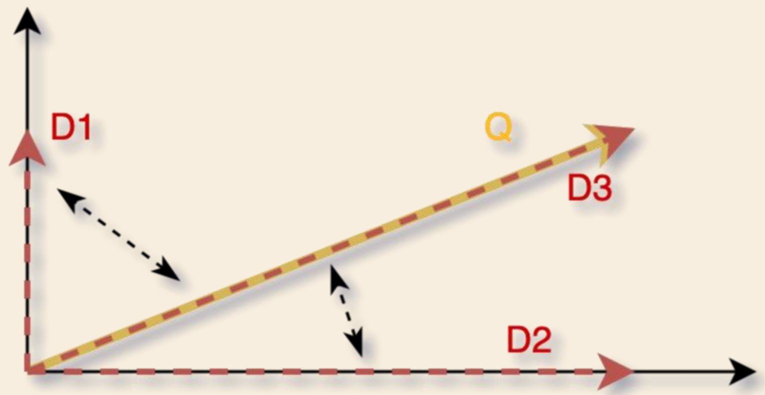
VSM Example

Consider the three documents:

- I am *happy* in summer.
- After Christmas I'm a *hippopotamus*.
- The *happy hippopotamus* helped Harry.

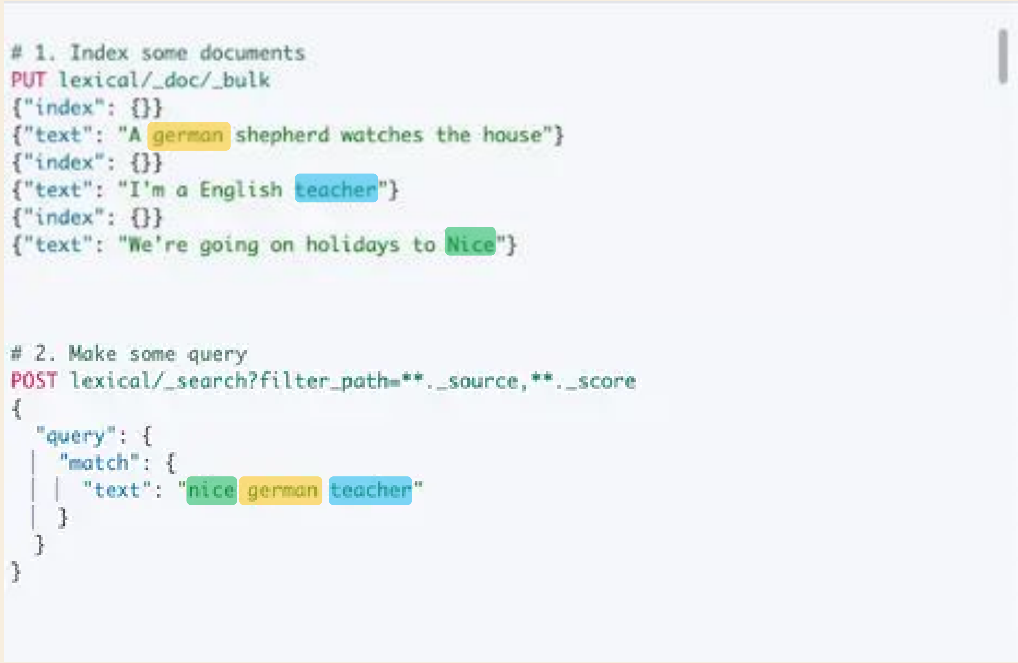
It is possible to create a vector for each document

- Document 1: `(happy, ____________) -> [2,0]`
- Document 2: `( ___ , hippopotamus) -> [0,5]`
- Document 3: `(happy, hippopotamus) -> [2,5]`

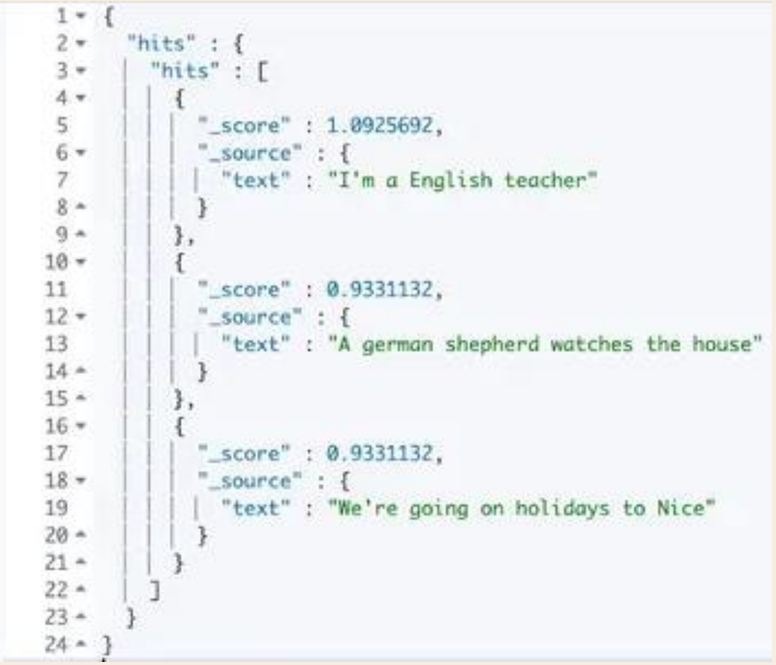


From Lexical to “Semantic” search

- Lexical search doesn’t try to understand the real meaning of what is indexed and queried: it just quantify the matches between the literals of the words or variants of them (stemming, synonyms, etc.)



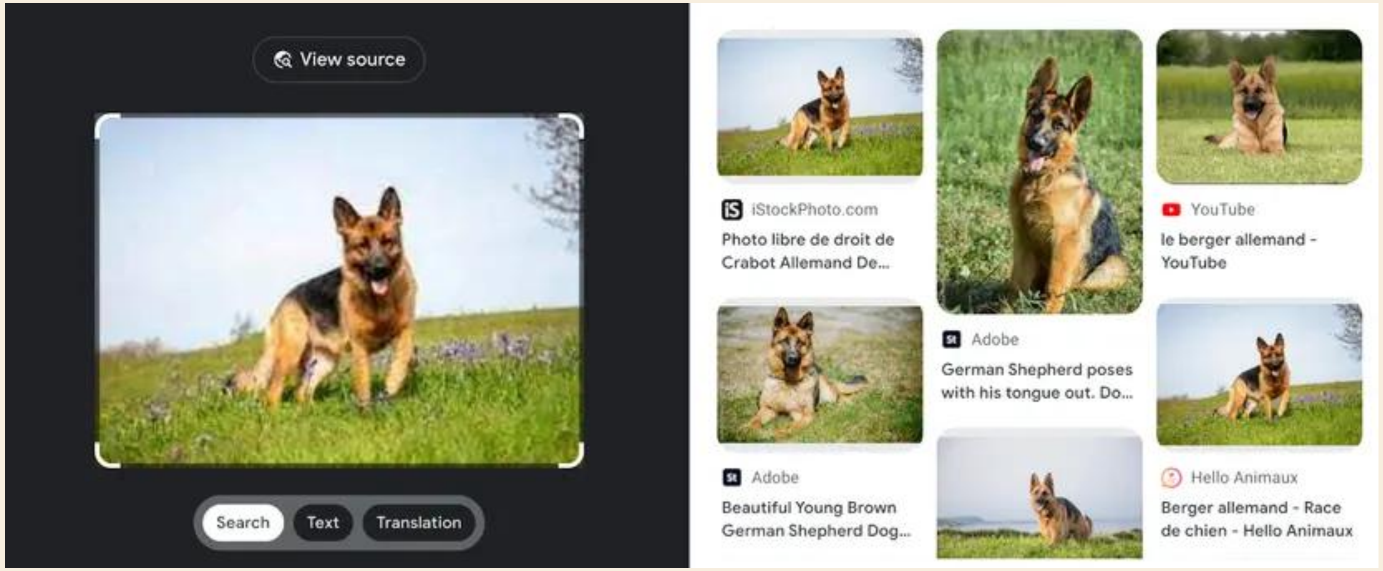
```
# 1. Index some documents
PUT lexical/_doc/_bulk
{"index": {}}
{"text": "A german shepherd watches the house"}
{"index": {}}
{"text": "I'm a English teacher"}
{"index": {}}
{"text": "We're going on holidays to Nice"}


# 2. Make some query
POST lexical/_search?filter_path=**._source,**._score
{
  "query": {
    "match": {
      "text": "nice german teacher"
    }
  }
}
```



```
{
  "hits" : {
    "hits" : [
      {
        "_score" : 1.0925692,
        "_source" : {
          "text" : "I'm a English teacher"
        }
      },
      {
        "_score" : 0.9331132,
        "_source" : {
          "text" : "A german shepherd watches the house"
        }
      },
      {
        "_score" : 0.9331132,
        "_source" : {
          "text" : "We're going on holidays to Nice"
        }
      }
    ]
  }
}
```

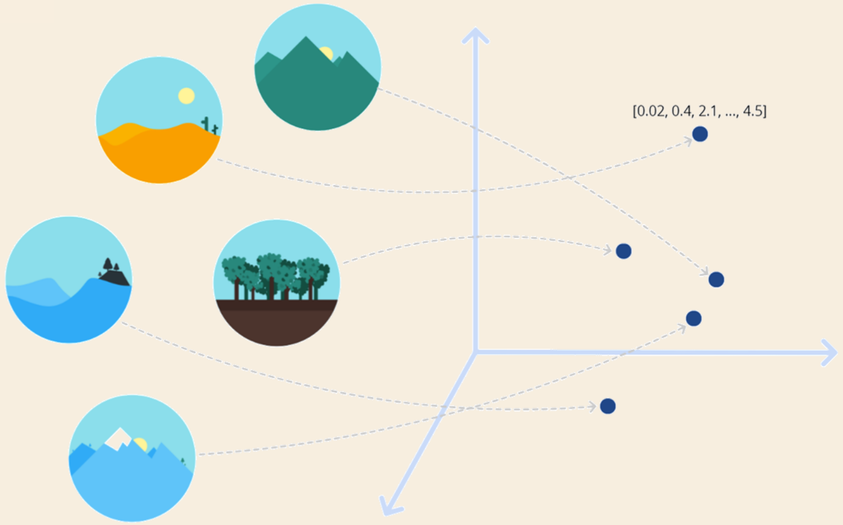
Let’s Unstructure

- Lexical search tries to push unstructured data (text) into a structure (set of occurrences)
- Besides text, even images, audio, and videos embed **unstructured semantic information**


View source
Search
Text
Translation
iStockPhoto.com
Photo libre de droit de Crabot Allemand De...
Adobe
German Shepherd poses with his tongue out. Do...
YouTube
le berger allemand - YouTube
Adobe
Beautiful Young Brown German Shepherd Dog...
Hello Animaux
Berger allemand - Race de chien - Hello Animaux

Semantical embeddings

- **Embeddings** are **numerical representations** of data (text, images, audio, etc.) in a **high-dimensional space**.
- Each piece of data is converted into a **vector,** a list of numbers that captures its **semantic meaning**.
  - Similar meanings → vectors close together.
  - Different meanings → vectors far apart.



How Embeddings Are Created

Embeddings come from **machine learning models** trained on large datasets.
Examples:

- **BERT / OpenAI Embeddings** – Language models that capture **contextual meaning**, event on long sentences.
- **CLIP** – Model that learns image embedding, aligned with their descriptions

Each model learns to **map meaning** into a **fixed-sized vector** of numbers based on context.

VSM vs Semantic embeddings

- The Vector Space Model creates a vector that depends on the user query, both in size and content.
  - It must be recomputed every time for each text in the knowledge base
  - Each dimension represents a word in the query, all the other words in the text are neglected, regardless the fact that they can be useful
- Semantic embeddings are generated independently from the user query, their size depends on the embedding model.

Query:	Proemio dell'Iliade, lite tra **Achille** e Agamennone
Expected text:	Cantami o diva del Pelide **Achille** l'Ira funesta
VSM:	[0, 0, 0, 8, 0, 0]
Embedding (Bert):	$\underbrace{[\ 0.02,\ 0.034,\ \ldots,\ 0.09,\ 0.012\ ]}_{\text{769 features}}$

Analysis (v2)

Traditional analysis

1. Tokenization of a block of text into individual terms
2. Normalization into a standard form
   - Lowercase vs uppercase
   - Stemming, i.e., reduction to the root form
   - Synonym management
3. Inverse indexing update
4. (When querying) generating the vector for the scoring function

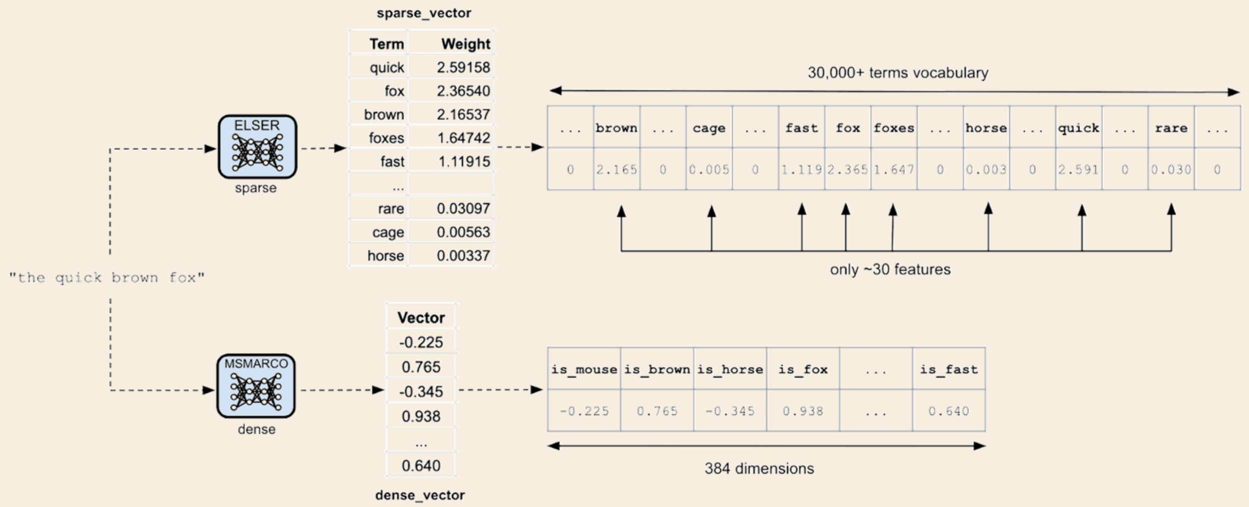
Dense search preprocessing

1. Normalization
   - Lowercase, bad characters removal…
   - No need for stemming and synonym management
2. Tokenization: tokens here are sub-words used as input for the embedding model
3. Creating and storing embedding once for all
4. (When querying) compare the embeddings with the query one

Hybrid Search with Elastic

Elastic provides hybrid search support to combine lexical search and vector search (both sparse and dense):

- Embeddings are extracted and combined to the traditional search to address semantic similarity
- Scoring can be combined, e.g. linearly



sparse_vector

Term	Weight
quick	2.59158
fox	2.36540
brown	2.16537
foxes	1.64742
fast	1.11915
...	
rare	0.03097
cage	0.00563
horse	0.00337

...	brown	...	cage	...	fast	fox	foxes	...	horse	...	quick	...	rare	...
0	2.165	0	0.005	0	1.119	2.365	1.647	0	0.003	0	2.591	0	0.030	0

Vector
-0.225
0.765
-0.345
0.938
...
0.640

dense_vector

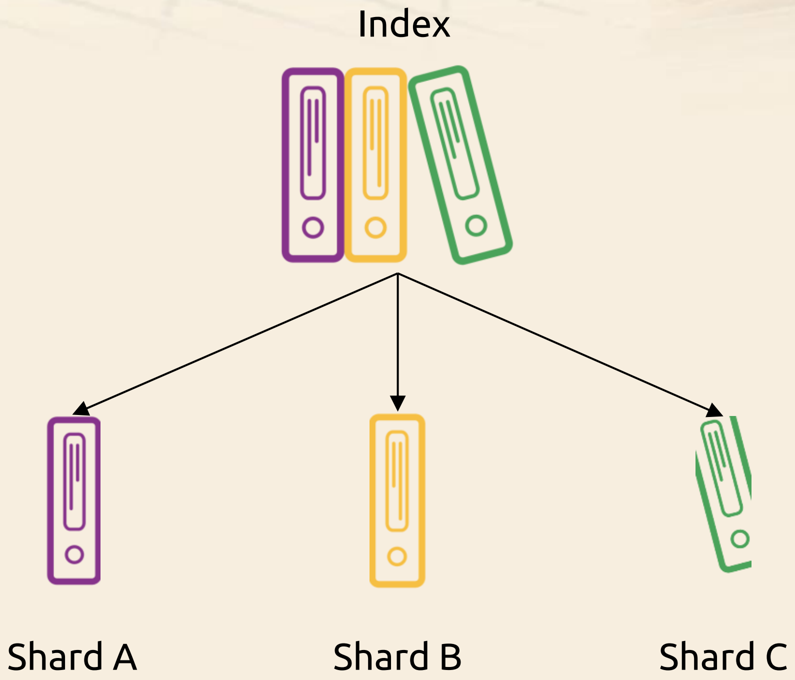
is_mouse	is_brown	is_horse	is_fox	...	is_fast
-0.225	0.765	-0.345	0.938	...	0.640

Horizontal scalability

Sharding

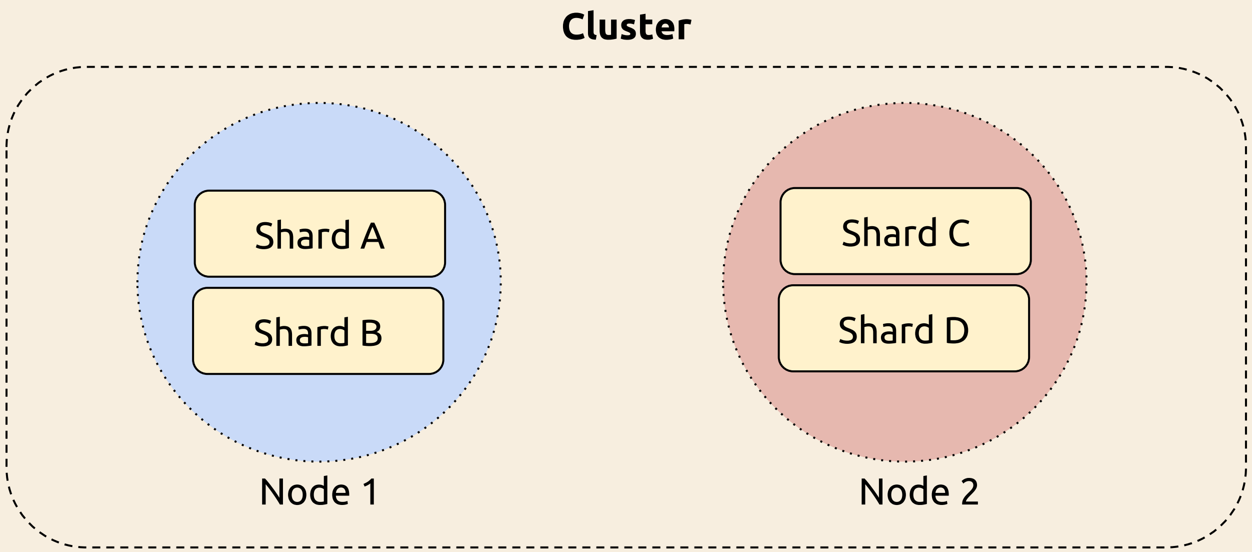
- Sharding is a technique to divide an index in smaller partitions
  - Each partition is a shard
- Each document belongs to a single shard
  - Each shard is an instance of a Lucene index
- When data is written to a shard
  1. It is periodically (every 1 second) written into a new immutable Lucene segment on disk
  2. It becomes available for querying
- Shards are the elementary units in which data is distributed on nodes in a cluster

Shards



Clusters

- A *cluster* is a collection of multiple machines (nodes in the cluster)
- Shards can be stored in any node within the cluster



Sharding

Why is sharding important?

- It allows splitting data in smaller chunks, and thereby scaling on large volumes of data
  - Data may be distributed across multiple nodes within a cluster
  - Shards can be stored on smaller disks
    - E.g., it is possible to store 1TB of data even without a single node with that disk capacity
- Operations can be distributed across multiple nodes and thereby parallelized
  - Performance is increased, because multiple machines can potentially work on the same query.
- Shards may be replicated on different nodes to increase availability

Document versioning

Optimistic concurrency control

- ElasticSearch uses optimistic concurrency control
  - It assumes that conflicts are unlikely to happen
  - However, if the underlying data has been modified between reading and writing, the update will fail
- Different from ACID transactions that need locking
- The process is “simple” for centralized data management

Modification propagation

- ElasticSearch data may be distributed on different nodes in a cluster
  - Shards may be replicated on different nodes (replica shards)
- When documents are created, updated, or deleted, the new version of the document has to be replicated to other nodes in the cluster
  - The primary copy is always written first
  - The replication requests are sent in parallel and may arrive at their destination *out of sequence*

Document versions

- Elasticsearch needs a way of ensuring that an older version of a document never overwrites a newer version
  - Every document has a `_version` number that is incremented whenever a document is changed
- Elasticsearch uses this `_version` number to ensure that changes are applied in the correct order
  - if an older version of a document arrives after a new version, it can be ignored
  - the `_version` number is used to ensure that conflicting changes made by applications do not result in data loss
- APIs that update or delete a document accept a version parameter
  - can apply optimistic concurrency control only when needed