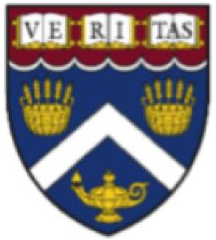
CSCI E-88 Principles Of Big Data Processing

Harvard University Extension, Fall 2019

Marina Popova



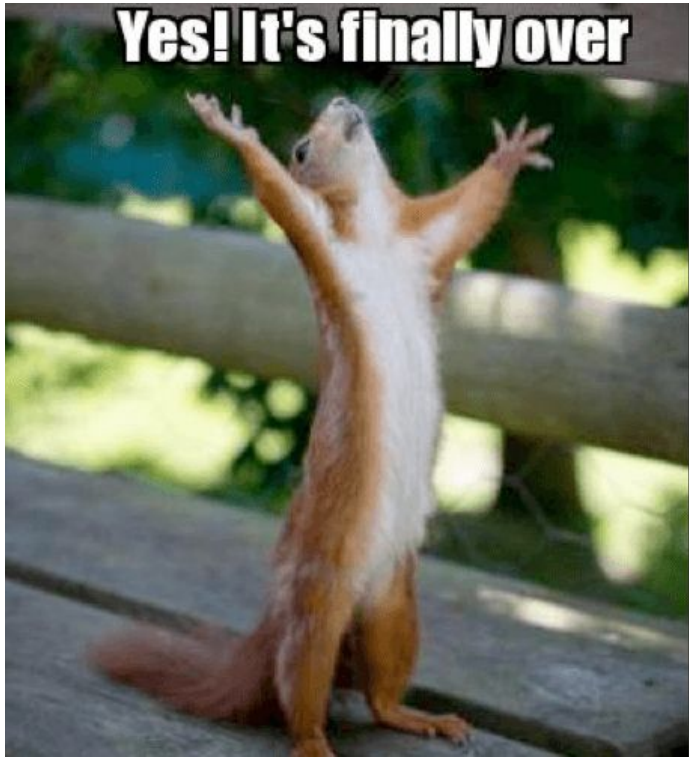
Lecture 12 Distributed Indexing and Search

Agenda

- Admin info
- Distributed Indexing Concepts
- ElasticSearch: an Illustration of Distributed Indexing and Search Systems

Admin Info

almost ...



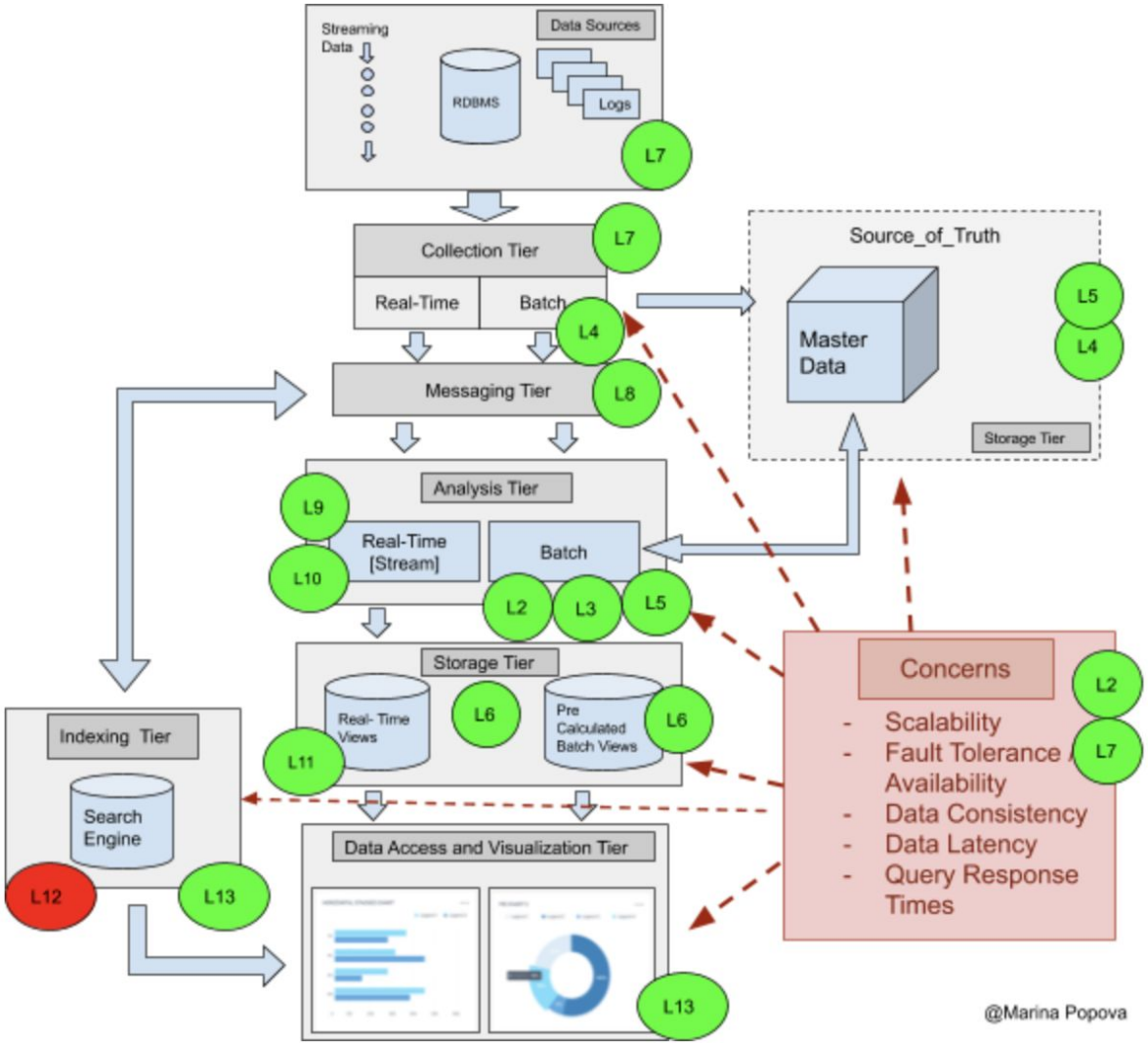
Important dates:

- HW11: - the last HW !!!
  - **Due: Tue, Nov 26, noon, EST - NO LATE DAYS !! - start working on Final Project proposals**
- Lab 11: ElasticSearch and Kibana by Andriy Pyshchyk Time: Wed, Nov 20, 7:30AM EST
- Final Project Proposal: Due: Sun, Dec 1, noon, EST - **no Late Days or extensions!**
- Final Project: Due: Sat, Dec 14, midnight, EST - **no Late Days, no extensions!**

All details about the Final Project and Proposal will be posted early next week

@Marina Popova

Where Are We?


Streaming Data
Data Sources
RDBMS
Logs
L7
Collection Tier
L7
Real-Time
Batch
L4
Source_of_Truth
Master Data
L5
L4
Storage Tier
Messaging Tier
L8
Analysis Tier
L9
Real-Time [Stream]
Batch
L10
L2
L3
L5
Storage Tier
Real- Time Views
L6
Pre Calculated Batch Views
L6
L11
Indexing Tier
Search Engine
L12
L13
Data Access and Visualization Tier
L13
Concerns
- Scalability
- Fault Tolerance / Availability
- Data Consistency
- Data Latency
- Query Response Times
L2
L7
@Marina Popova

Where Are We?

What have we learned so far:

- How to collect data
- How to model/store data in the Master Datastore
- how to write massively parallel jobs to run queries on the batch/Master data stores - and store the results into Batch Views storage
- How to process streaming data and do analyses on "windowed" batches
- How to store results of the streaming/windowed analyses in RealTime Views data storage

How do we utilize collected data?

Master Datasets:

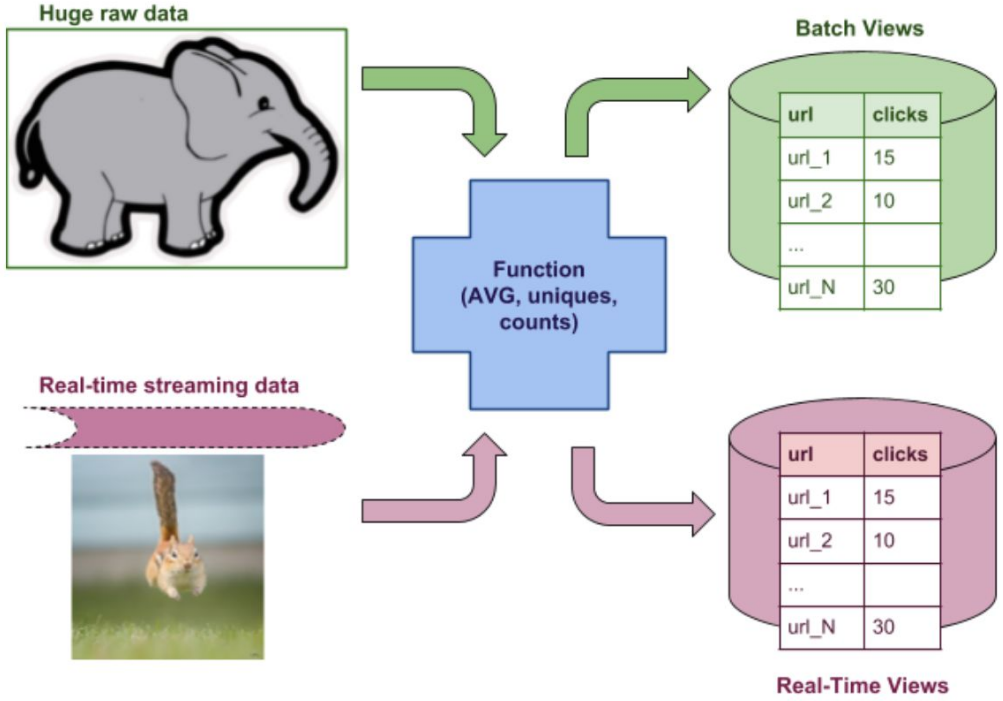
- use for bulk GETs and ETL-type jobs (batch processing/ HDFS)
- Single or bulk key-value retrievals/searches - NoSQL storage

Batch + Real-Time Views:

- can be used as final results - as input for further Analytics/ML training/ AI
- can also be used as is for Visualization/Monitoring (next lecture)
- Beware of limitations! (aggregates only)

Limitations?

- slow AdHoc query response times
- pre-aggregated data by fixed dimensions only
- limited SQL-like capabilities



@Marina Popova

What's Next?

Next Goals:

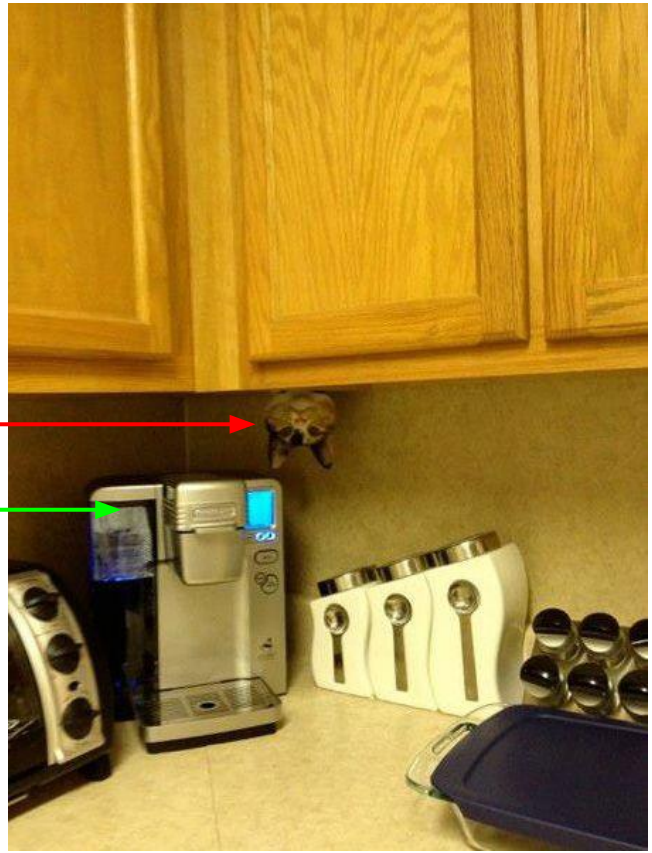
- deeper insight into the raw data
- faster AdHoc queries
- greater flexibility in query options
- more complex data analyses
- richer interactive visualization

This usually requires use of specialized Search systems !

Beyond Simple Search

We want to be able to **quickly find something that is of interest**, given a proper description:

- "sneaky cat"
- "fancy coffee machine"



@Marina Popova

How?

By **parsing, analyzing and categorizing** your information and storing the discovered metadata in a form of an **"index"**

The "analyzing and categorizing" part is a huge area of research - AI/NLP/ML etc.

Distributed Indexing

What is an index?

A database where information is stored and organized to allow for **quick information retrieval**

What is Indexing?

A process of collecting, parsing and extracting meaningful information and storing the resulting data into the index structures

Why do we need Indexing at all ??

- Speed up access to the required information by some criteria (search queries) - without scanning the full corpus (amount) of data
- To improve accuracy of the search results
- To perform different types of queries:
  - exact/partial match
  - Range; Top-K; similarity; keyword

@Marina Popova

It's not new!

All storage systems do some kind of indexing!

- Block-level meta info : File-based systems
  - S3, HDFS, ObjectStore (Google Cloud Storage, others)
- Key-level meta info: Key-Value
  - Key: Document [extra meta for some fields]
  - Key: Wide-Column [extra meta for PKs/ partitions]
- Graph DBs - not indexed storage, but "index-free adjacency"
- Document/Lucene-based (fully indexed)
  - ES, Solr
- NoSQL - SQL: hybrid between NoSQL and Lucene-based
  - Kudu [column-based, like Parquet], Druid, CockroachDB, BigQuery, Presto, Impala ...

Distributed Indexing

Indexing Requirements and Considerations:

- Speed of access:
  - Search response latency
  - Update latency
  - Index generation time - ingestion latency
- Parallel processing - for all of the above areas
- Size of index - much smaller than the original data
- Storage options - support for encryption and compression
- Multi-dimensional queries
  - Ability to search/combine results by multiple parameters
- Fault tolerance - how reliable should the service be?
  - Index corruption / machine failures, bad data handling strategies

In fact, all our usual Concerns are at play here - see our Master Plan

Index Types

With so many different requirements - **many types of indexing data structures are used**:
From the Wikipedia: https://en.wikipedia.org/wiki/Search_engine_indexing

Suffix tree

Built by storing the suffixes of words. Used for searching for patterns in DNA sequences and clustering.

Inverted index

Stores a list of occurrences of each atomic search criterion, typically in the form of a hash table or binary tree

Citation index

Stores citations or hyperlinks between documents to support citation analysis, a subject of Bibliometrics.

Ngram index

Stores sequences of length of data to support other types of retrieval or text mining

Document-term matrix

Used in latent semantic analysis, stores the occurrences of words in documents in a two-dimensional sparse matrix.

Indexing Strategies

We will use the following research, which has a great analyses of the characteristics of the most popular indexing strategies:

http://www.slac.stanford.edu/pubs/slacpubs/16250/slac-pub-16460.pdf

"A Survey On Big Data Indexing Strategies" by:
Fatima Binta Adamu , Adib Habbal , Suhaidi Hassan1 , R. Les Cottrell , Bebo White2 , Ibrahim Abdullahi
InterNetworks Research Laboratory, School of Computing, Universiti Utara Malaysia, 06010 UUM, Sintok, Kedah, Malaysia. 1 SLAC National Accelerator Lab

Indexing Strategies

In general:

Indices are a list of tags, names, subjects, etc. of a group of items which references where the items occur

An **indexing strategy** is :

- the design of an access method to a searched item
- as well as the organization of data in the storage system.

Indexing strategies categories:

- Artificial Intelligence (AI) approach
- Non-Artificial Intelligence (NAI) approach

How do we pick an indexing strategy??

The type of indexing strategy to be used in processing a *specific dataset* **depends on the type of queries** that will be performed on the dataset, such as:

- **similarity queries (nearest neighbor search)**
- **range queries**
- **point queries (equality search)**
- **keyword queries**
- **ad-hoc queries (dynamic, with variable parameters)**

Very important!!
Indexing strategy can (and should!) be different for different datasets and/or different requirements for queries!

Indexing Strategies

Artificial Intelligence (AI) Strategies: have an ability to detect unknown behavior in Big Data. They *establish relationships between data items by observing patterns and categorizing items or objects with similar traits*

- Latent Semantic Indexing (LSI)
  - "Latent" - **present but hidden**
  - "Semantic" - meaning
- Hidden Markov Model (HMM)

Challenges:

Scalability 🙂 performance

(of course)

what is hidden here???

@Marina Popova

Indexing Strategies

Non-Artificial Intelligence Strategies:

- index design and creation **does not depend on:**
  - the meaning of the data item
  - the relationship between items
- index design **does depend on:**
  - which items are most queried or searched for in a particular data set
- Strategies:
  - Tree-based (B-tree, R-tree and X-tree)
    - retrieval of data is done in a sorted order, following branch relations of the data item - best for **nearest neighbor queries**
  - inverted indices
  - Hash - best for **point queries**
  - custom (GiST - Generalized Search Tree, and GIN - Generalized Inverted Index)

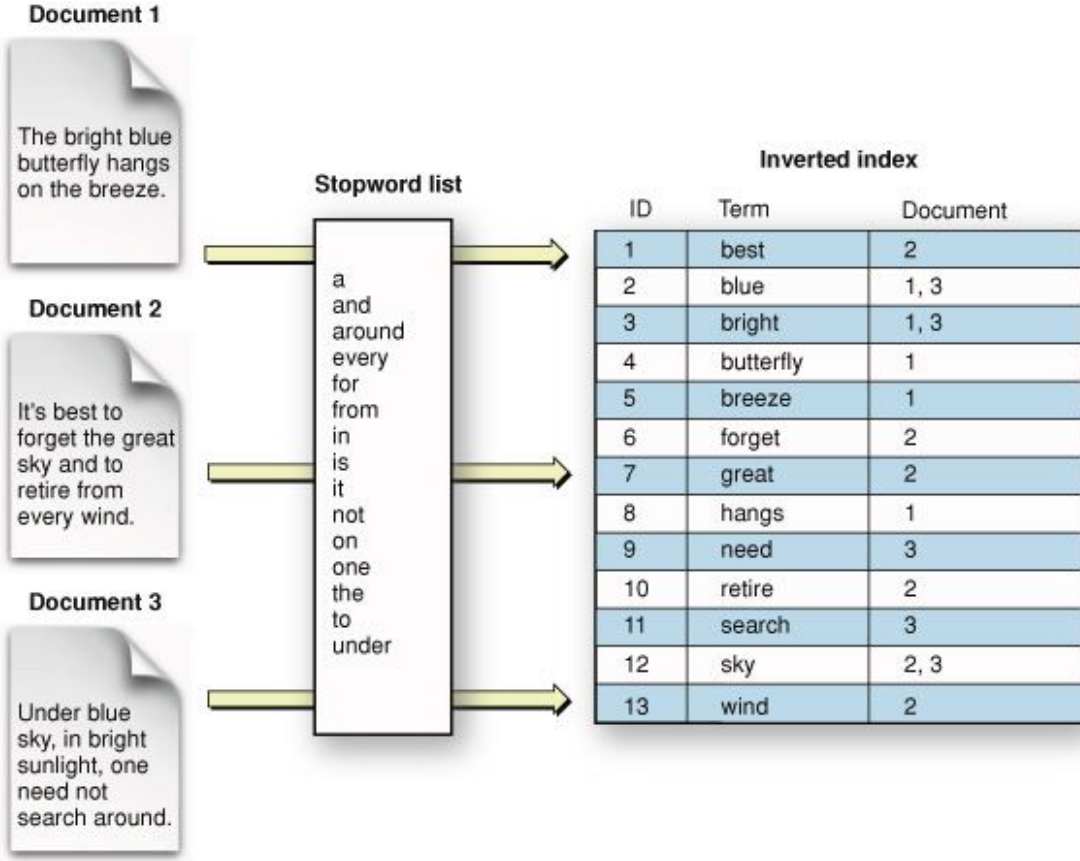
Inverted Indices

What is a Term?

- **a word or set of words to search by**

What is Inverted Index?

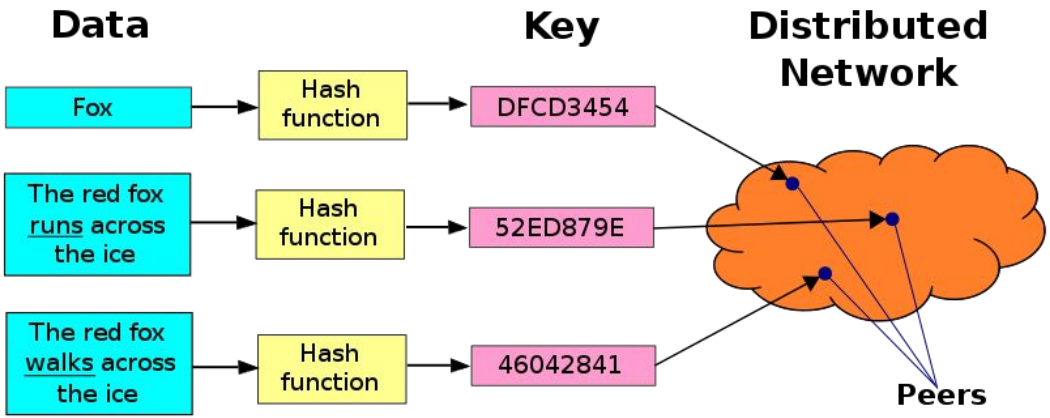
- it is an index which has terms marked as keys
- the terms map to the document they appear in.
- the index is sorted by its keys and works well with Boolean operators (AND,OR, AND NOT)
- we find the documents by matching the terms – this is why we say it is inverted



(image source: https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/SearchKitConcepts/searchKit_basics/searchKit_basics.html)

Inverted Indices

- The inverted index is a sparse matrix, since not all words are present in each document
- The inverted index can be considered a form of a hash table
- In some cases the index is a form of a binary tree, which requires additional storage but may reduce the lookup time
- In larger indices the architecture is typically a distributed hash table (ES - shards!)



Inverted and Forward Indices

- inverted index is usually filled via a merge or rebuild - and has to support parallel operations for these actions
- often, Inverted index is filled via sorting of the Forward Index

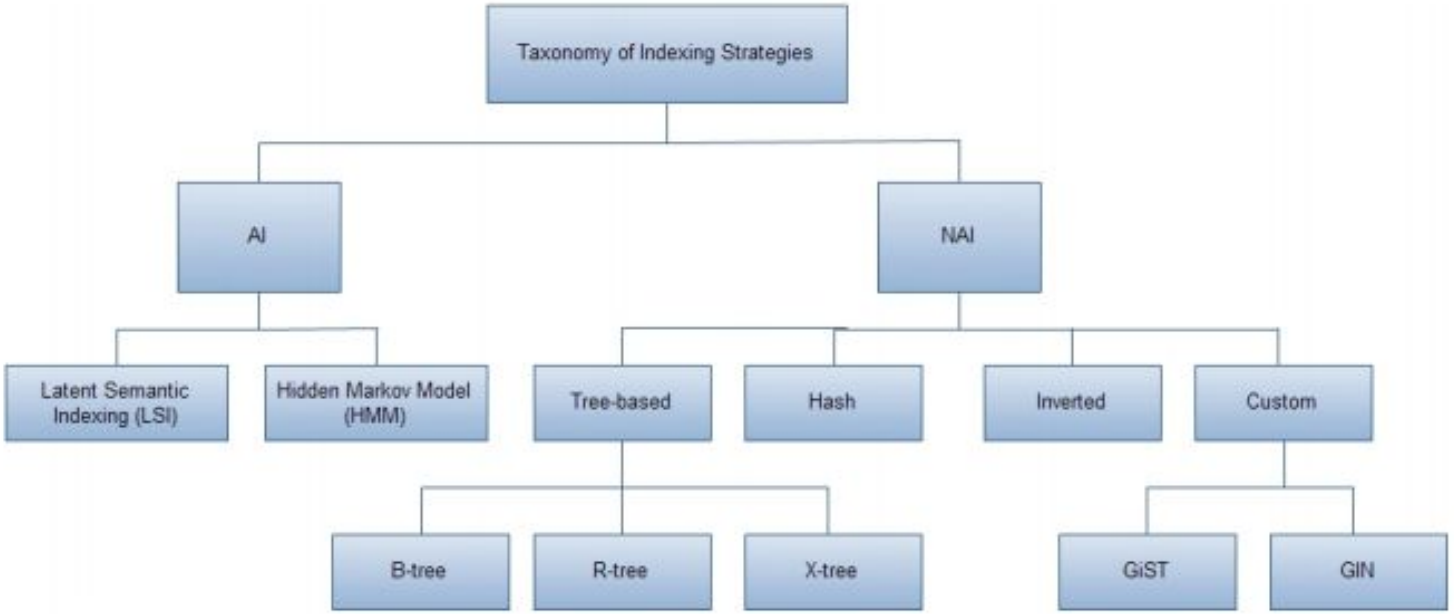
what is Forward Index?

- an index that uses Document ID as a key and a list of words for that document as the values

Forward Index

Document	Words
Document 1	the,cow,says,moo
Document 2	the,cat,and,the,hat
Document 3	the,dish,ran,away,with,the,spoon

Indexing Strategies



@Marina Popova

Table I
INDEXING STRATEGIES AND QUERY-TYPES

Indexing Strategies	Data-type	Query-type
B-tree	Log data, multimedia data	Range queries, similarity queries (NNS): 1 dimension
R-tree	Spatial data e.g Geographical coordinates, multimedia data	Spatial or range query: 2-3 dimensions
X-tree	Spatial data	Spatial or range query: Multiple dimensions
Hash	Log data, multimedia data	Point query (Equality search)
GiST	Spatial data, log data	Range query, ad-hoc query
GIN	Log data, spatial data	Similarity query, ad-hoc query
Inverted	Multimedia data, documents	Keyword queries
LSI	Multimedia data, spatial data (textual data)	Keyword queries
HMM	Multimedia data, unstable signals etc.	Ad-hoc query

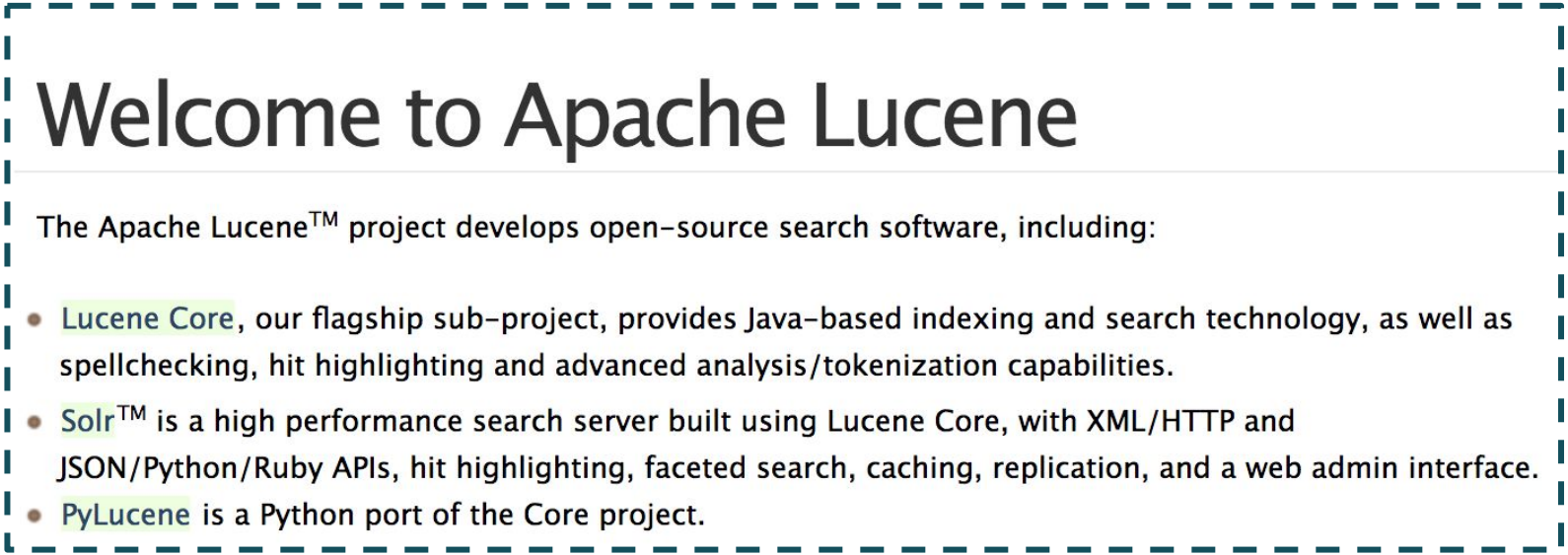
Table II
CHARACTERISTICS OF INDEXING STRATEGIES

Indexing Strategies	Properties	Challenges
B-tree	- One dimensional access method -Tree structure with nodes and pointers - Scales linearly	- Waste storage space - Not suitable for multidimensional access -Consumes huge computing resources
R-tree	- More scalable than the B-tree - 2 to 3 dimensional access method	- Index consumes more memory space
X-tree	- Multidimensional access method	- Consumes memory space
Hash	- Presents the exact answer (uses '=' operator) - Quick information retrieval	- Computational overhead
GiST	- Arbitrary indexes - Based on the tree structures	- Slower query response
GIN	- Arbitrary indexes - Based on the tree structures	- Longer processing time

Inverted	- Index consumes less space - Full text search (keyword search)	- Longer data processing time - Limits the search space, not necessarily producing the exact answer - Can present wrong answers due to synonyms and polysemy
LSI	- Uses data and meaning of data for indexing - Presents accurate query results (since it uses more information)	- Demands high computational performance - Consumes more memory space
HMM	- Based on the Markov model - Recognizes relationships between data	- Demands high computational performance

Apache Lucene

Foundation for many search frameworks, including ElasticSearch!
http://lucene.apache.org/



Apache Lucene - Core Features

Scalable, High-Performance Indexing

- over 150GB/hour on modern hardware
- small RAM requirements -- only 1MB heap
- incremental indexing as fast as batch indexing
- index size roughly 20-30% the size of text indexed

Powerful, Accurate and Efficient Search Algorithms

- ranked searching -- best results returned first
- many powerful query types: phrase queries, wildcard queries, proximity queries, range queries and more
- fielded searching (e.g. title, author, contents)
- sorting by any field
- multiple-index searching with merged results
- allows simultaneous update and searching
- flexible **faceting**, **highlighting, joins and result grouping**
- fast, memory-efficient and typo-tolerant **suggesters**
- pluggable ranking models, including the Vector Space Model and Okapi BM25
- configurable storage engine (codecs)

Apache Lucene - Core Features

Concurrent Query Execution:

http://blog.mikemccandless.com/2019/10/concurrent-query-execution-in-apache.html#comment-form

- Lucene indexes may be composed of multiple sub-indexes, or **segments**
- each segment is a fully independent index, which could be searched separately
- Indexes evolve by:
  - creating new segments for newly added documents
  - merging existing segments
- searches may involve multiple segments and/or multiple indexes, each index potentially composed of a set of segments

Concurrent Query execution mode allows searches to be performed in parallel over multiple segments!

Range queries: deeper dive

Most Search Frameworks/Systems are Lucene based: what Lucene can do - they can do

Original numeric range implementation in Lucene:
https://www.elastic.co/blog/apache-lucene-numeric-filters

Numbers are stored the same as text tokens, but zero-padded :

5 --> "005"
10 --> "010"
234 --> "234"

Limitations:

- Have to know the max size of your numbers range
- Have to re-index all data if larger numbers happen in the future
- Slow
- Index bloating

ElasticSearch: range queries

ElasticSearch: made a great contribution to Lucene for range and similarity queries:
https://www.elastic.co/blog/numeric-and-date-ranges-in-elasticsearch-just-another-brick-in-the-wall

Uwe Schindler added a better solution, called ***numeric tries***

This approach also indexed numbers as if they were textual tokens (the inverted index), but it did so at multiple precision levels for each indexed number, so that entire ranges of numbers were indexed as a single token

This made the index larger, because a single numeric field is indexed into multiple binary encoded tokens. But at search time it meant there were often far fewer terms to visit for a given range, making queries faster, because the requested query range could be recursively divided into a union of already indexed ranges (tries). Indices got larger and queries got faster.

ElasticSearch: illustration

The 'numeric tries' approach was further improved, and later replaced, by using new BKD trees!

To improve geo-spacial searches, ES added a new feature to Lucene (after 6.0 version):

BKD-trees (**B k-d**imensional trees) data structures for fast multi-dimensional point search

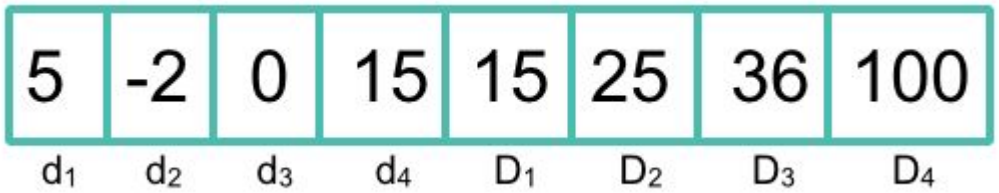
Instead of continuing to represent numeric data using a structure specifically designed and tuned for text, the Bkd implementation introduced the **first flexible tree structure designed specifically for indexing discrete numeric points**

It works great for both numeric and date range queries

ElasticSearch: illustration

Indexing:
for each dimension, represent the dimensional numeric range as a 2 dimensional point where the encoding is stored as an array of minimum values, d, (for each dimension) followed by an array of maximum values, D.

Essentially, to Lucene a 1-dimension range simply looks like a 2-dimension point. This encoding is then passed to the Bkd indexer which proceeds to build a hierarchical tree of d*2-dimensional points; where d is the dimension of the range, or 4 in the illustration:

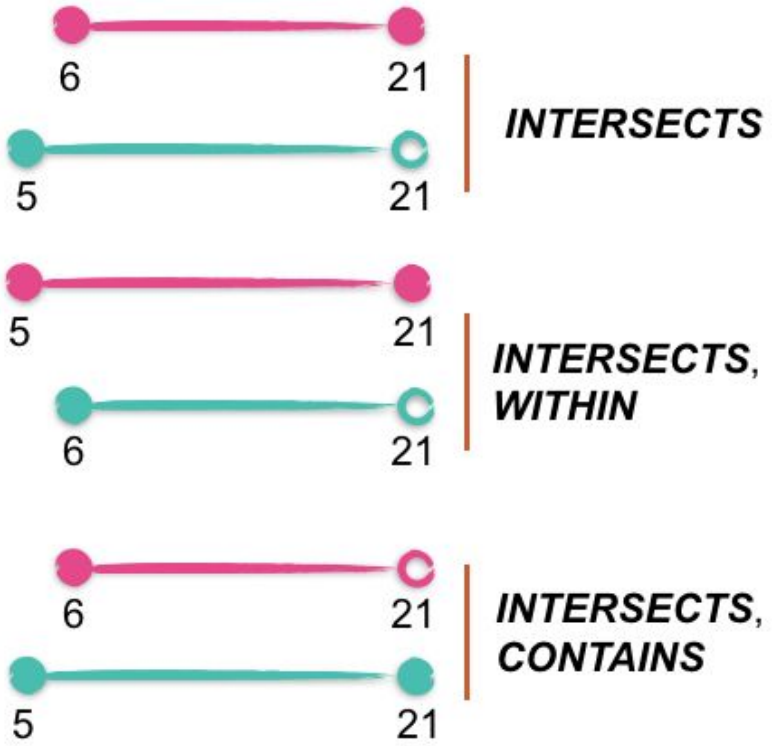


ElasticSearch: illustration

Search:
Is implemented by computing the spatial relation between the user provided search range (the target) and the indexed ranges (or minimum bounding ranges at each level of the tree).

ES implemented a **dimensional range comparator** that evaluates the range relation at each dimension. The spatial relations implemented in the comparator include: INTERSECTS, CONTAINS, and WITHIN.

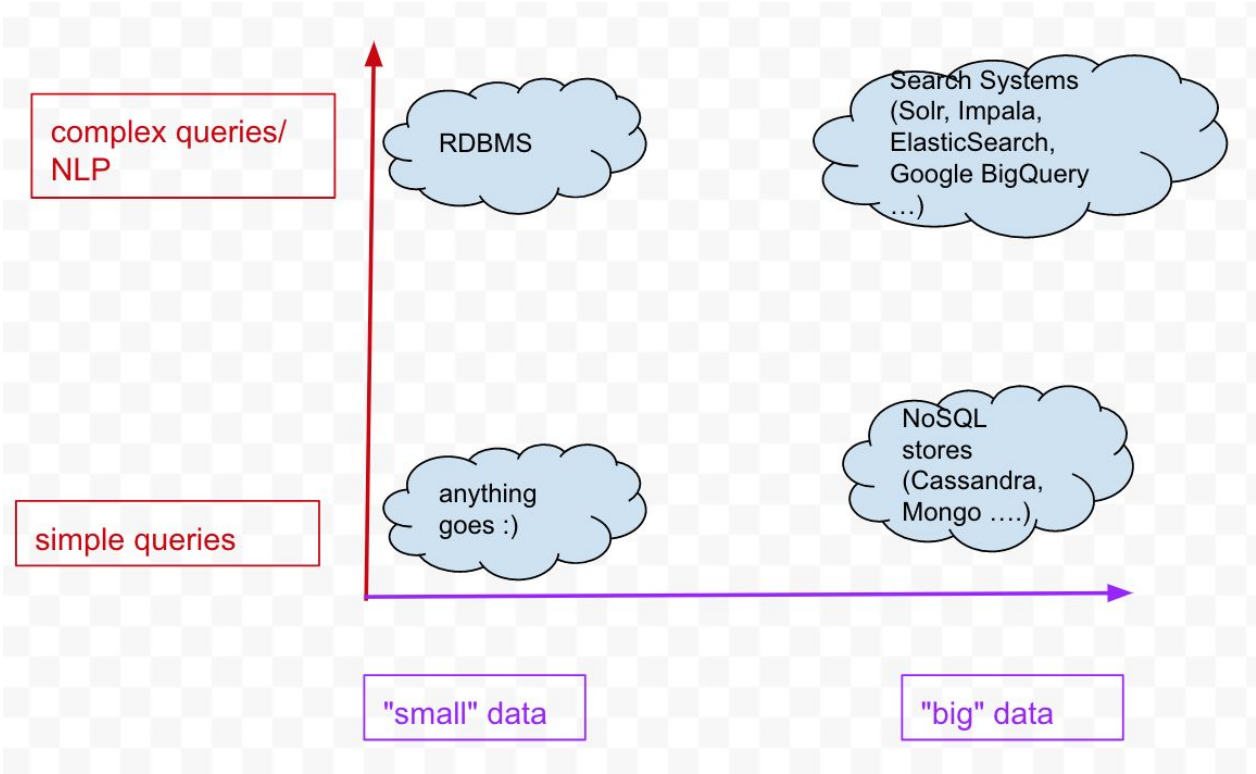
DISJOINT queries are then accomplished using an INTERSECTS query contained in a boolean query with a MUST_NOT clause.



Distributed Search and Indexing Systems - Landscape

Summary:

- As amount of data grows, data processing, indexing and search systems become more and more complex
- As search requirements become more complex - so are the systems that can fulfil them
- Search Systems - a specialized type of Big Data Processing frameworks focusing on distributed indexing and search



ElasticSearch: as a Distributed Indexing and Search Framework

Next:

- Review ElasticSearch - and example of a Distributed Indexing and Search System
  - Based on the original Presentation by ElasticSearch Core engineer, Igor Motov
- Why not Lucene ?
  - ES is More feature-reach
  - Lucene - the engine under the hood
  - Lucene - not distributed (Solr is)

Elasticsearch

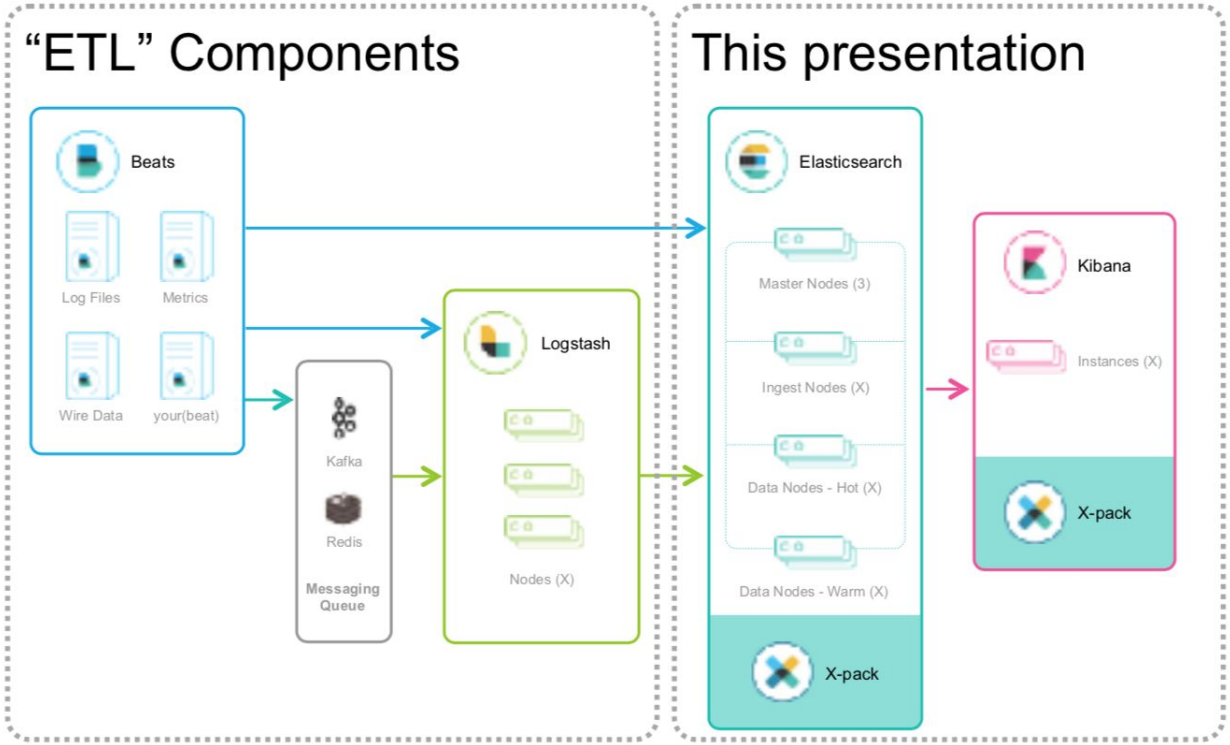
a distributed search and analytics engine

- Created by Shay Banon
- Store and retrieve JSON documents
- Supports REST API
- Search fields and return a subset of these documents
- Run analytics queries on all documents or a subset
- Scalable: more documents than can fit on a machine
- Fault-tolerant: any machine can be lost at any time

Near Realtime (NRT)

Elasticsearch is a near-realtime search platform. What this means is there is a slight latency (normally one second) from the time you index a document until the time it becomes searchable.

Elastic Stack



All data is stored as JSON documents:

JSON Document

```
{
    "text": "You know, for search...",
    "author": "Shay Banon",
    "tags": ["elasticsearch", "search", "computers"],
    "year": 2010
}
```

ElasticSearch: Core Concepts

Cluster
a collection of one or more nodes (servers) that together holds your entire data and provides federated indexing and search capabilities across all nodes. A cluster is identified by a unique name which by default is "elasticsearch"

Node
a single server that is part of your cluster, stores your data, and participates in the cluster's indexing and search capabilities:

- A Java application
- One node per JVM process, usually
- Listens on port 9200 for REST requests
- Nodes communicate with each other (in the cluster) via a binary protocol on port 9300

ElasticSearch: Core Concepts

Index
a collection of documents that have somewhat similar characteristics. An index is identified by a name (that must be all lowercase) and this name is used to refer to the index when performing indexing, search, update, and delete operations against the documents in it.

Type - Deprecated in 6.0.0 version!
used to be a logical category/partition of your index to allow you to store different types of documents in the same index

Document
a basic unit of information that can be indexed; is expressed in JSON

ElasticSearch: Core Concepts

Shards and Replicas
An index can potentially store a large amount of data that can exceed the hardware limits of a single node. Elasticsearch provides the ability to subdivide your index into multiple pieces called shards. Each shard is in itself a fully-functional and independent "index" that can be hosted on any node in the cluster

- this allows you to horizontally split/scale your content volume
- This also allows you to distribute and parallelize operations across shards (potentially on multiple nodes) thus increasing performance/throughput

Elasticsearch allows you to make one or more copies of your index's shards into what are called **replica shards**, or replicas for short. This enables:

- high availability in case a shard/node fails. For this reason, it is important to note that a replica shard is never allocated on the same node as the original/primary shard that it was copied from.
- It allows you to scale out your search volume/throughput since searches can be executed on all replicas in parallel.

Each Elasticsearch shard is a Lucene index!

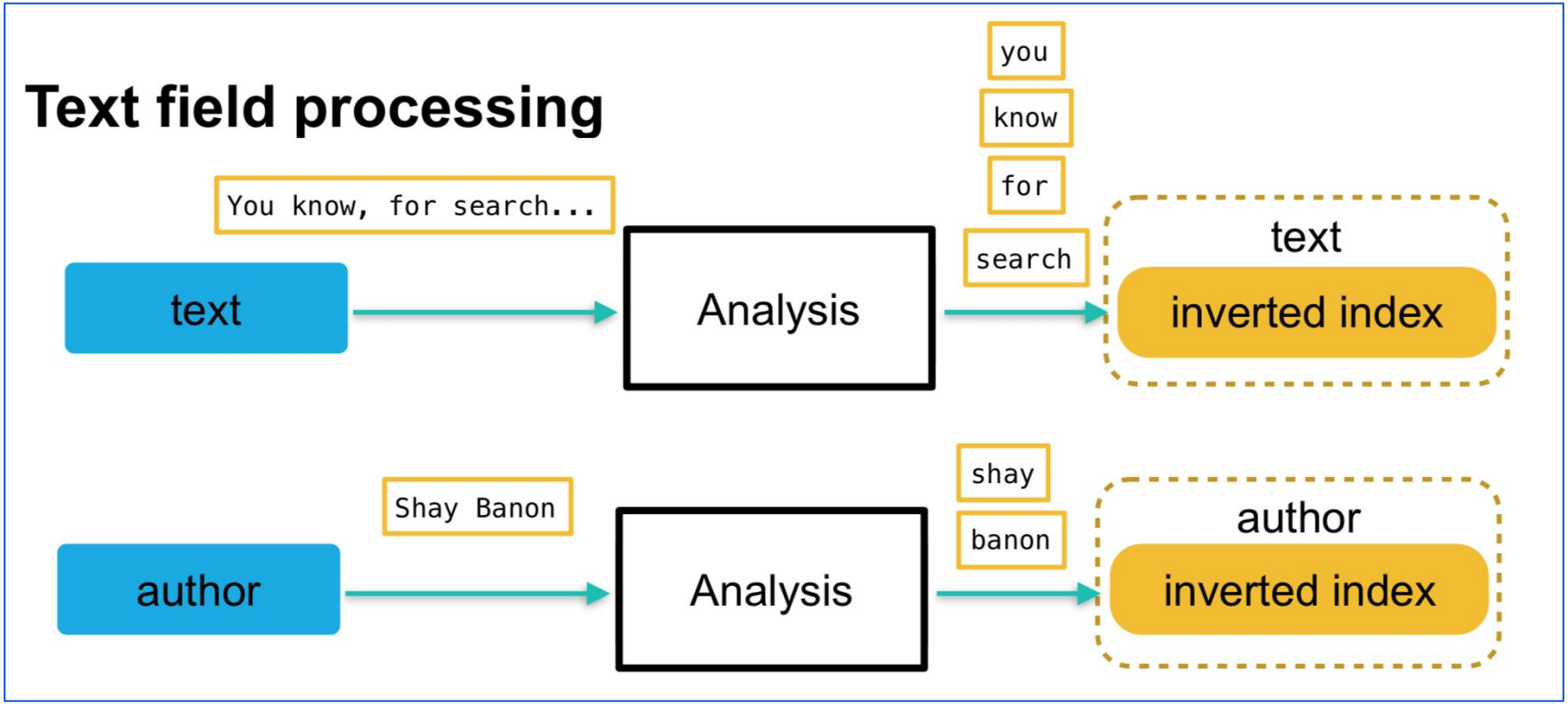
Cassandra → Elasticsearch (roughly)

- Table → Index + Type
- Schema → Mapping
- Row → Document
- Column → Field
- Node → Node
- Cluster → Cluster

ElasticSearch: Indexing

How is indexing done, exactly ?

By analyzing documents first!


Text field processing
You know, for search...
text
Analysis
you
know
for
search
text
inverted index
Shay Banon
author
Analysis
shay
banon
author
inverted index

What is Analysis?

Analysis is the process of converting text, like the body of any email, into tokens or terms which are added to the inverted index for searching. Analysis is performed by an analyzer which can be either a built-in analyzer or a custom analyzer defined per index.

Analysis

- Preprocess text, for example convert emoji to text (char_filter)
- **Split text into tokens and remove punctuation marks (tokenization)**
- **Lowercase all tokens**
- Remove frequently used words (stop words)
- Convert words to stems
- Replace words with their synonyms
- …

Results:

Inverted index for the “author” field

TOKEN	DOC #
b	5
banon	0, 2
heraclitus	4
johnson	5
lyndon	5
mitchell	3

TOKEN	DOC #
plato	1
shay	0, 2, 3

More Info for Indexing:

Mappings (Schema)

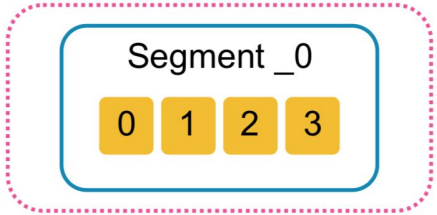
- Defines how individual fields should be indexed
- Types of new fields are automatically detected
- Once a field is defined, it's type can no longer be changed
- Multiple types are supported
  - Strings (text and keyword)
  - Numbers (byte, integer, long, float, double ….)
  - Date
  - Inner and Nested objects
  - Other special types (geo shapes, percolation queries, suggestions…)

Segments

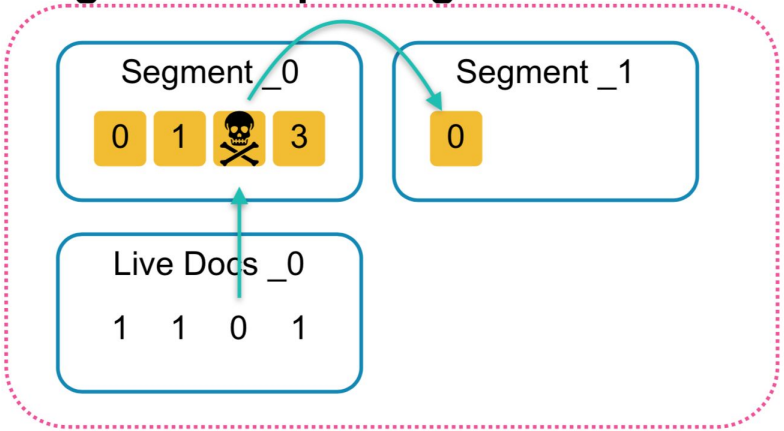
- Each time we index documents they are stored in segments
- Before segment is generated documents are buffered in memory
- Segments are generated on “refresh” operation which happens every second by default
- Documents in segments are enumerated from 0 to N

- Contain all information about added records in form of
  - Stored fields
  - Inverted indices
  - Doc values
  - Token Positions (for phrase searches)
  - BKD Trees (for numbers, ranges, and geo data)
- Immutable (updated and deleted documents are marked in a bitmap)
- Multiple segments can be searched sequentially
- Merged automatically in bigger segments when we get too many of them

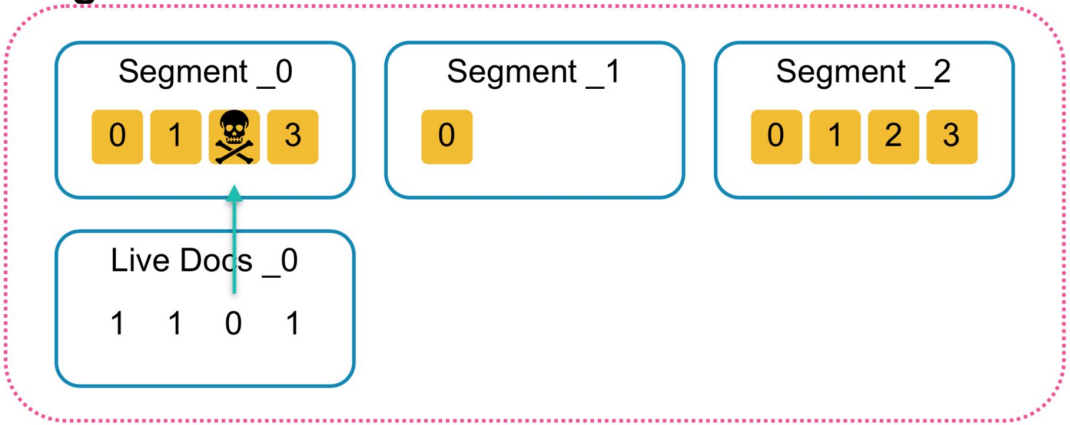
Lets see them in action!



Segments - updating document 2

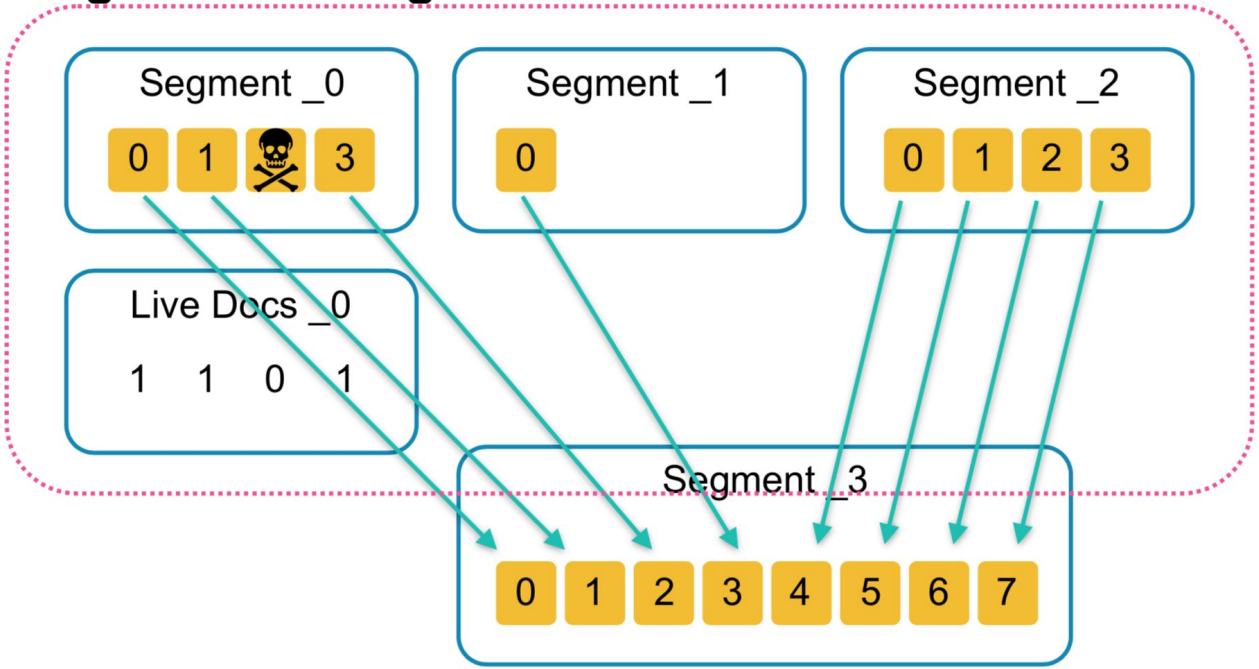


Segments - more documents indexed

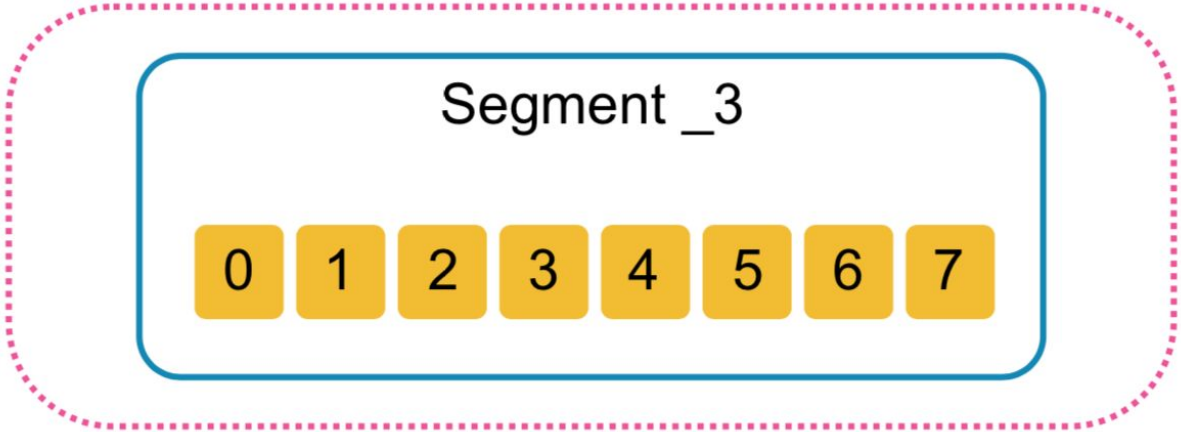


@Marina Popova

Segments - merge



Final Cleanup:



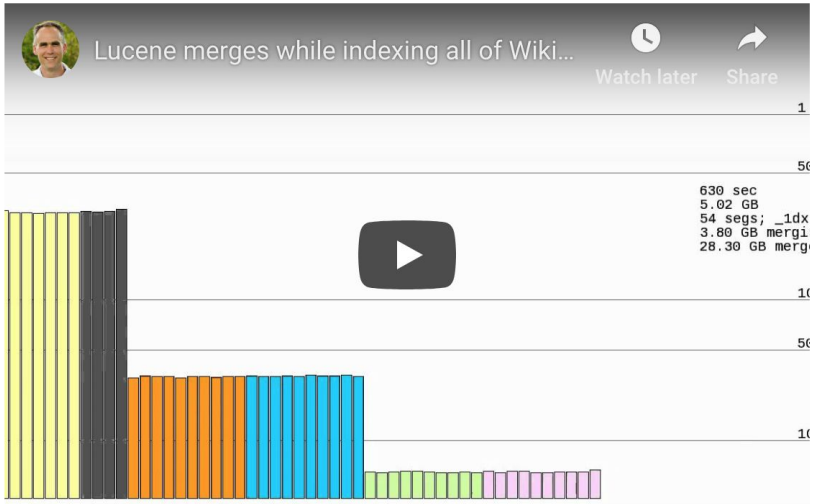
All this is happening while other operations take place…..

Here is a video of how this is happening [in Lucene but its the same in ES …]

http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html

Visualizing Lucene's segment merges

If you've ever wondered how Lucene picks segments to merge during indexing, it looks something like this:



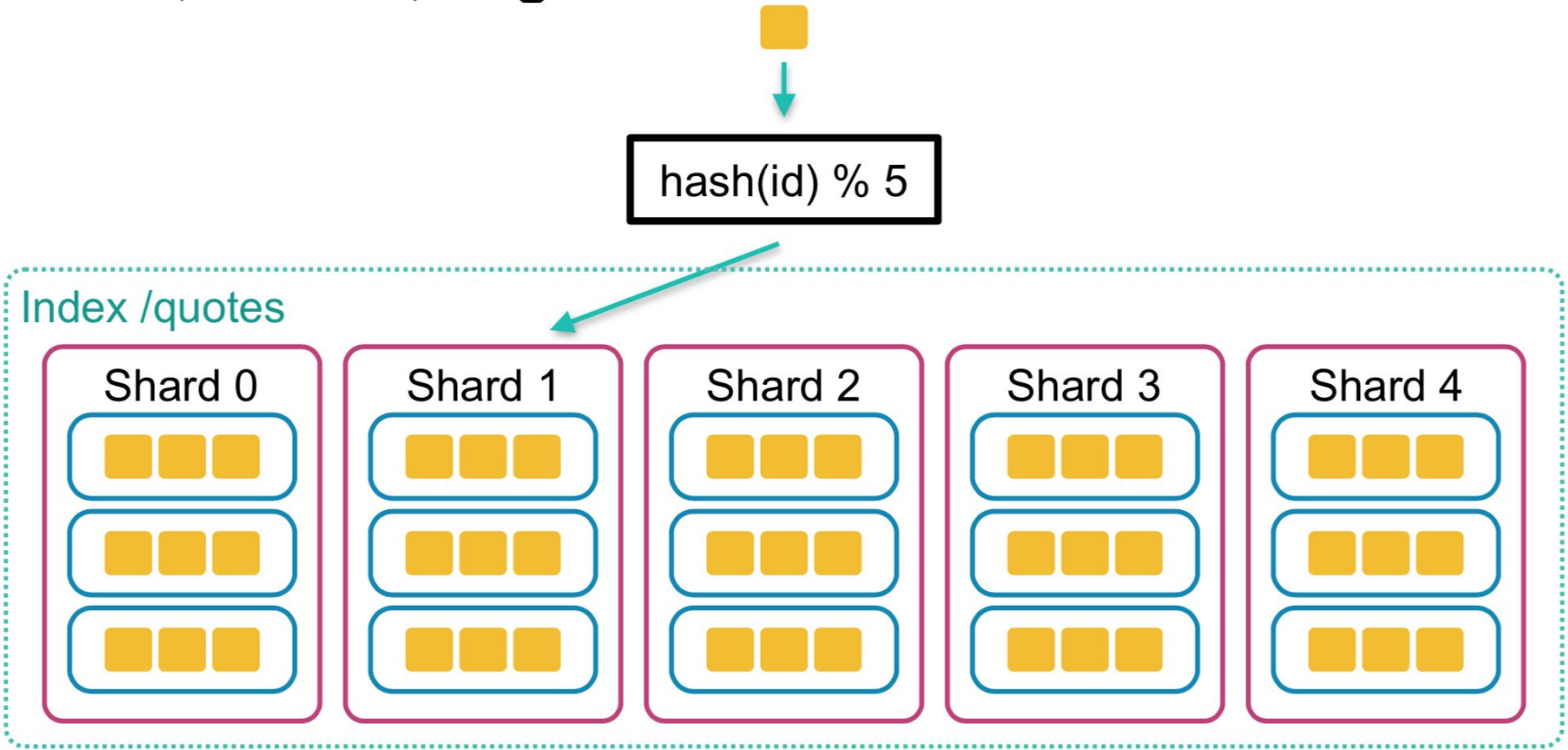
That video displays segment merges while indexing the entire Wikipedia (English) export (29 GB plain text), played back at ~8X real-time.

@Marina Popova

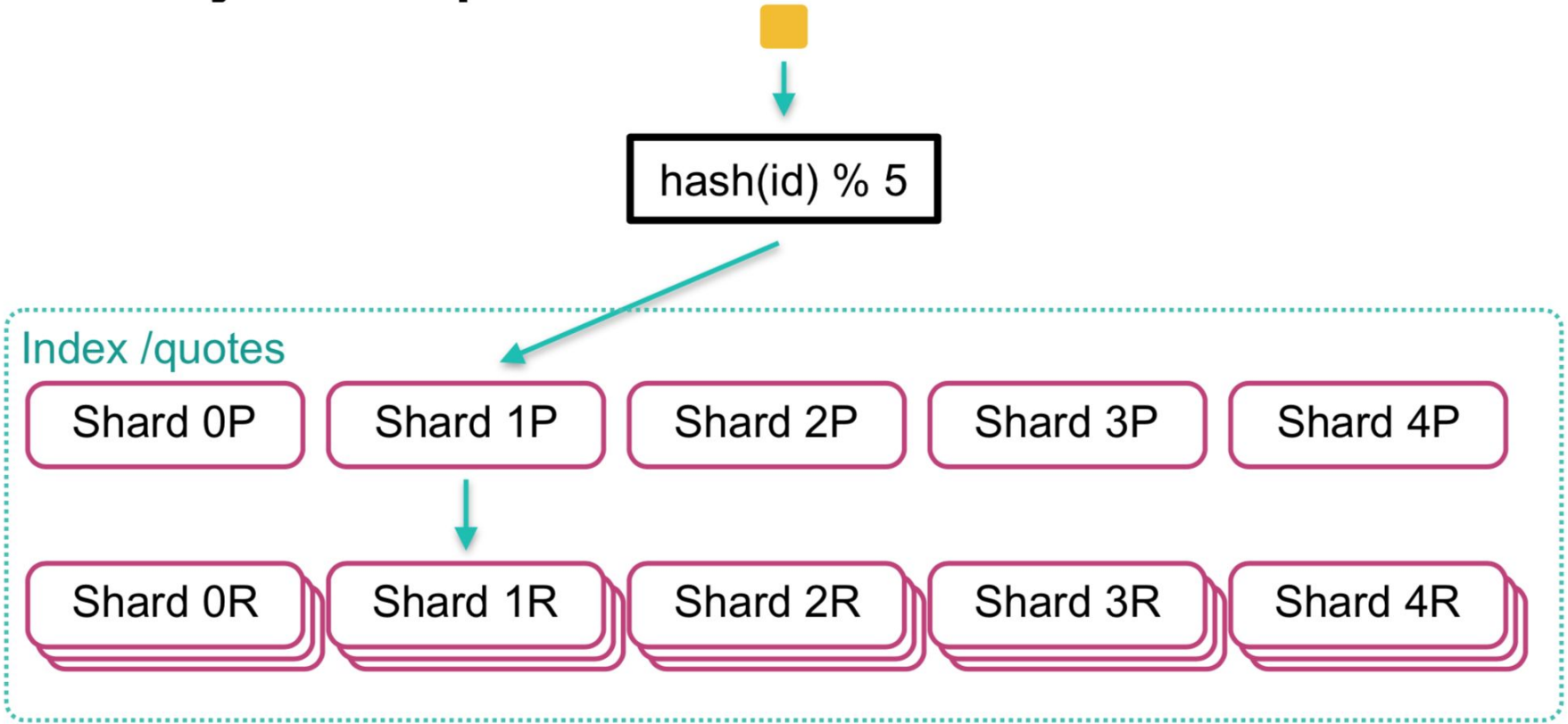
Shards

- segments are stored in index partitions called **shards**
- an index can be divided into one or more shards
- be default, in index is divided into 5 shards - but it is configurable
- each document is assigned to a shard based on the document ID

Index, Shards, Segments and Documents



Primary and Replica Shards

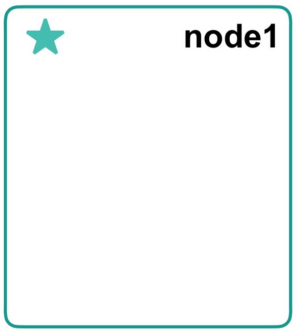


Creating two new indices

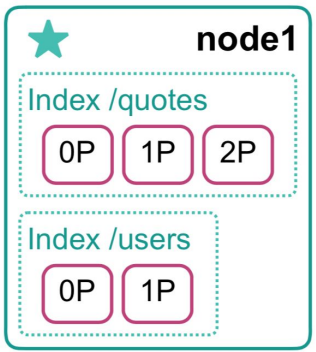
```
PUT /quotes
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 1
  }
}

PUT /users
{
  "settings": {
    "number_of_shards": 2,
    "number_of_replicas": 1
  }
}
```

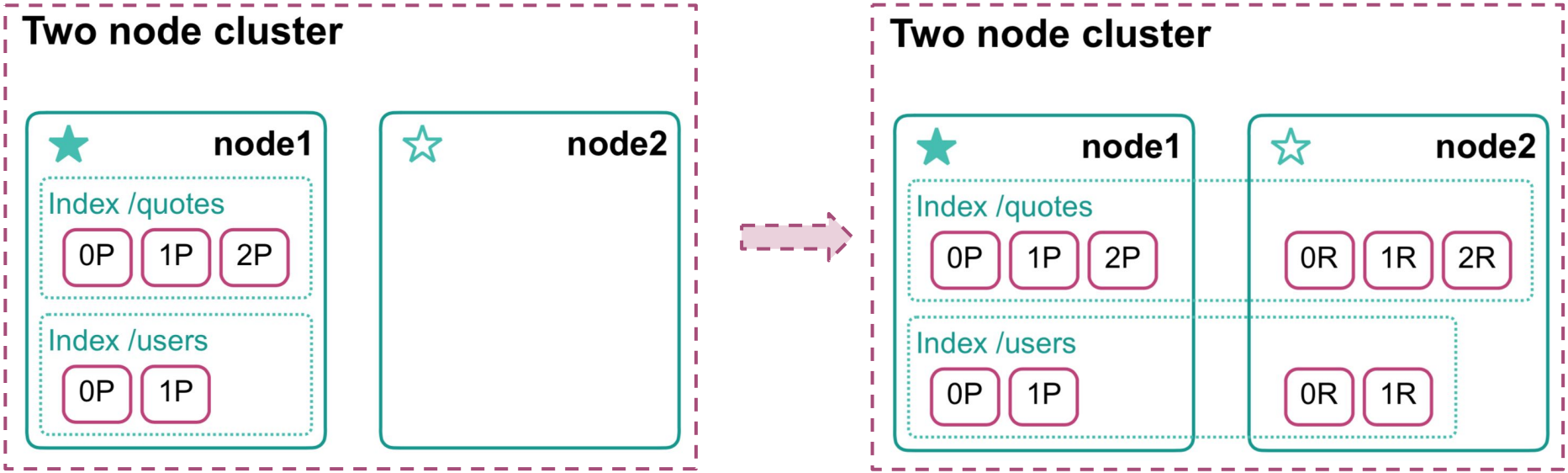
Empty one node cluster



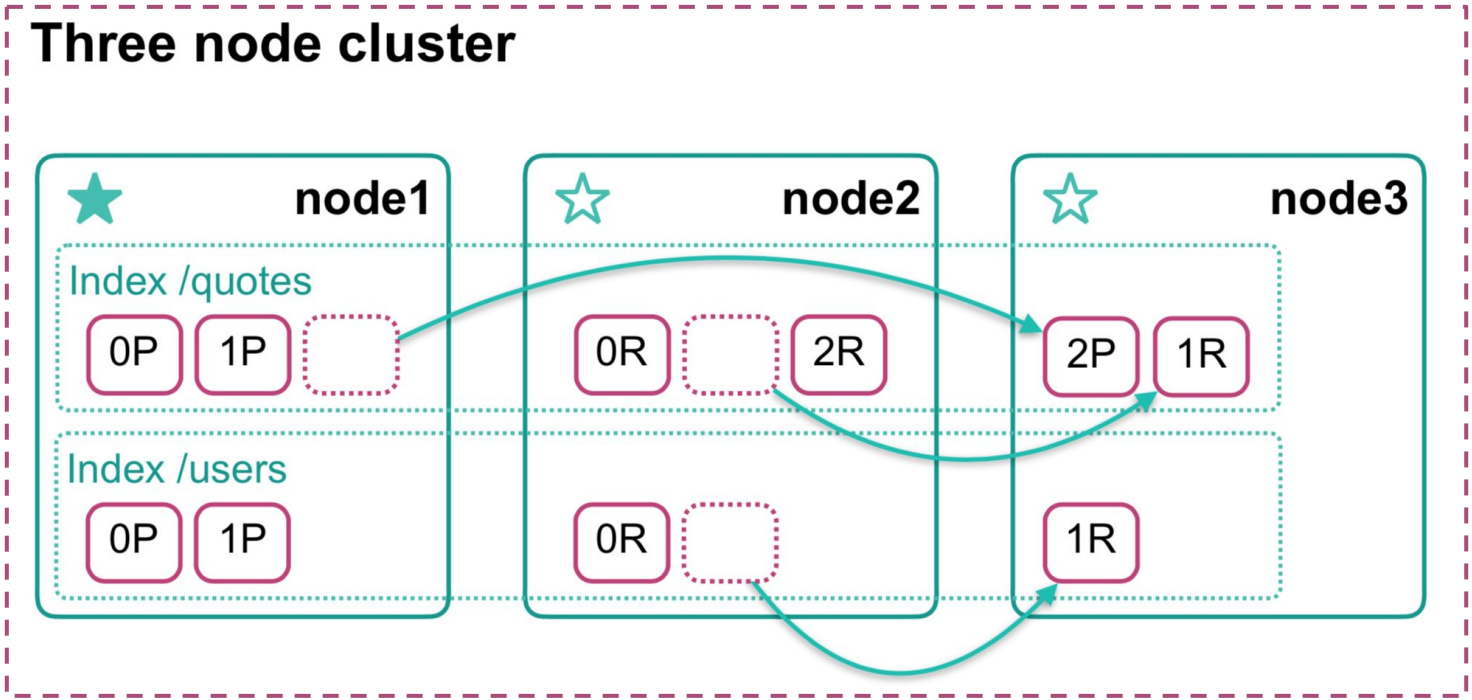
One node cluster



Adding one more node to the cluster:



Adding one more node to the cluster:



Cluster State

- Contains information about
  - Nodes in the cluster
  - Indices, their settings and mappings
  - Shards, their location and state
  - …
- Maintained by a single node in the cluster (master node)
- Published to every node in the cluster on every change

Cluster Health

- GREEN - all shards are accounted for
- YELLOW - all primary shards are available, but at least one replica is missing
- RED - at least one primary shard is missing

How do we interact with ElasticSearch?? - via REST APIs !!

Create Index:

```
PUT /quotes
{
  "settings": {
    "number_of_shards": 3
  }
}
```

Index a document:

```
PUT /quotes/doc/plato1
{
    "text": "Man - a being in search of meaning.",
    "author": "Plato"
}
```

Search for a document:

```
GET /quotes/doc/_search
{
  "size": 4,
  "query": {
    "match": {
      "text": "find search",
      "operator": "and"
    }
  },
  "sort": [{
      "author.keyword": {"order": "asc"}
  }]
}
```

The magic is in: Parallelization!!

- Shards are a unit or parallelization for all operations
- One operation runs on one core - better have lots of cores !!

Remember Lucene?

Concurrent Query execution mode allows searches to be performed in parallel over multiple segments! - *this is not available in ES yet ...*

Thread Pools

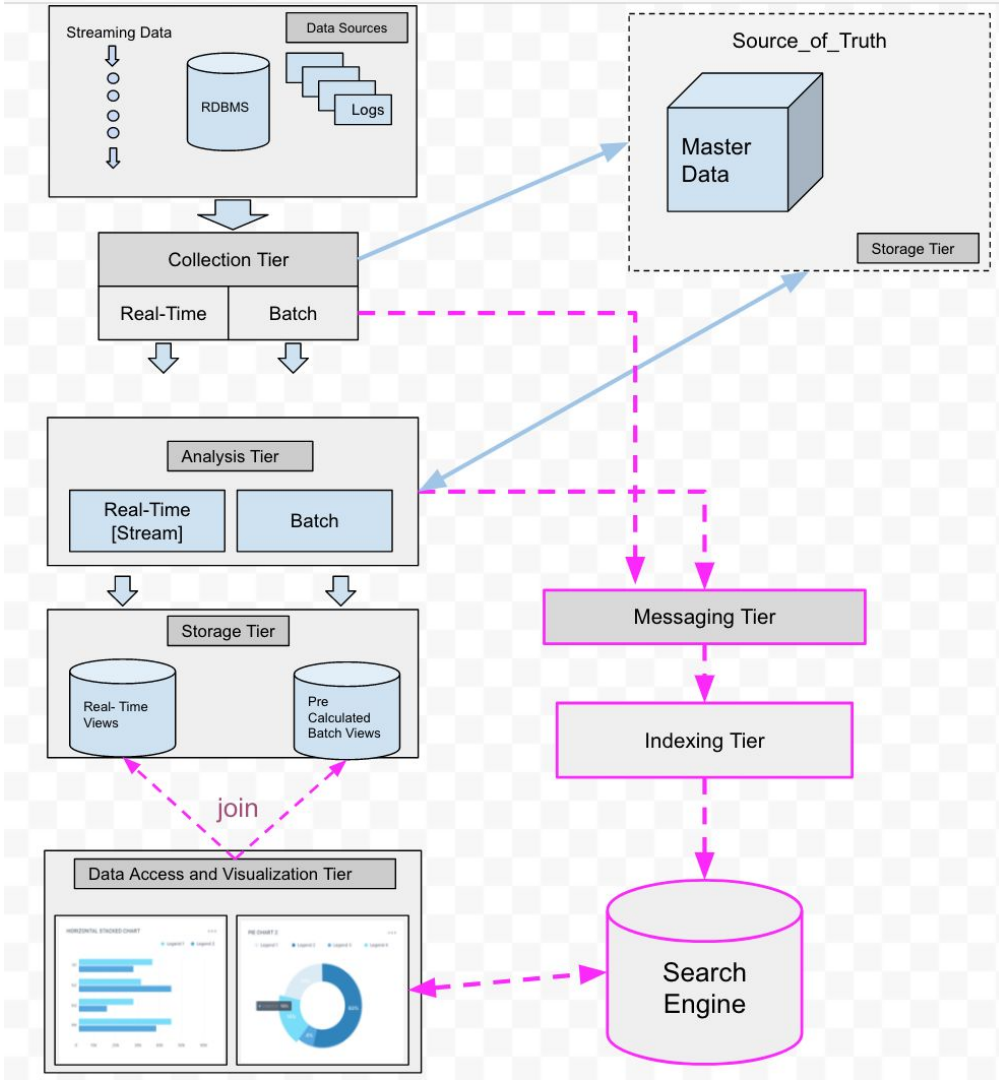
- All network operations are asynchronous
- Operation specific
  - Indexing
  - Searching
  - Fetch
  - Cluster state update (1 thread)
  - etc.

Stepping back …. Where does it fit?

Who and how exactly is going to index the data into a Search System?
What is the Indexing Tier?

Many options here as well:

- Elastic: Logstash
  - https://www.elastic.co/guide/en/logstash/current/deploying-and-scaling.html
- Kafka Connect
- Other connector-like frameworks (Flume plug-ins, etc.)
- Custom application that uses Search System's APIs to index data



@Marina Popova

Custom Applications

Custom applications can be developed to interact with Search systems:
In case of ElasticSearch: can use ES APIs to index/query data

Java API docs:
https://www.elastic.co/guide/en/elasticsearch/client/java-rest/7.4/java-rest-high.html

Python APIs:
https://www.elastic.co/guide/en/elasticsearch/client/python-api/current/index.html

Custom Applications: Java APIs:

Java High Level REST Client supports the following Document APIs:

Main interface:
RestHighLevelClient

A `RestHighLevelClient` instance needs a REST low-level client builder to be built as follows:

```
RestHighLevelClient client = new RestHighLevelClient(
        RestClient.builder(
                new HttpHost("localhost", 9200, "http"),
                new HttpHost("localhost", 9201, "http")));
```

Single document APIs

- Index API
- Get API
- Exists API
- Delete API
- Update API
- Term Vectors API

Multi-document APIs

- Bulk API
- Multi-Get API
- Reindex API
- Update By Query API
- Delete By Query API
- Rethrottle API
- Multi Term Vectors API

Java APIs: Index Management APIs

Index Management APIs:
Can be used to create/get/delete indices and mappings

Create Index Request

A `CreateIndexRequest` requires an `index` argument:

```
CreateIndexRequest request = new CreateIndexRequest("twitter"); (1)
```

(1) The index to create

Index settings

Each index created can have specific settings associated with it.

```
request.settings(Settings.builder() (1)
    .put("index.number_of_shards", 3)
    .put("index.number_of_replicas", 2)
);
```

(1) Settings for this index

Individual Document Indexing:

```
IndexRequest request = new IndexRequest("posts"); (1)
request.id("1"); (2)
String jsonString = "{" +
        "\"user\":\"kimchy\"," +
        "\"postDate\":\"2013-01-30\"," +
        "\"message\":\"trying out Elasticsearch\"" +
        "}";
request.source(jsonString, XContentType.JSON); (3)
```

1. Index
2. Document id for the request
3. Document source provided as a `String`

BulkRequestBuilder:

To index multiple documents
(into multiple indices) in one request

```
BulkRequest request = new BulkRequest(); (1)
request.add(new IndexRequest("posts").id("1")  (2)
        .source(XContentType.JSON,"field", "foo"));
request.add(new IndexRequest("posts").id("2")  (3)
        .source(XContentType.JSON,"field", "bar"));
request.add(new IndexRequest("posts").id("3")  (4)
        .source(XContentType.JSON,"field", "baz"));
```

1. Creates the `BulkRequest`
2. Adds a first `IndexRequest` to the Bulk request. See Index API for more information on how to build `IndexRequest`.
3. Adds a second `IndexRequest`
4. Adds a third `IndexRequest`

Custom Applications

Example application:

Java app: https://github.com/BigDataDevs/kafka-elasticsearch-consumer