Elasticsearch internals

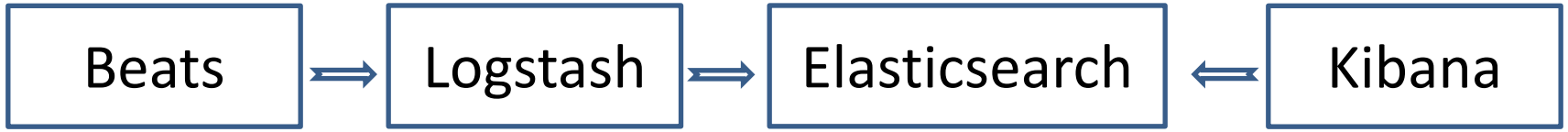
Martin Toshev

Agenda

- The ELK stack
- Elasticsearch architecture
- Extending Elasticsearch

The ELK stack

Overview



data collection

data aggregation and processing

Indexing and storage

analysis and visualization

Elasticsearch

- A web server build on top of the Lucene Java library
- Another way to describe it is a document-oriented database
- Provides more functionality not provided by Lucene such as:
  - Caching
  - Clustering
  - JSON-based REST API

Elasticsearch

- The basic data structure used by Elasticsearch is an **inverted index**
- Indexes are stored on disk in separate files
- Search can be performed on multiple indexes
- Documents in an index are logically grouped by type (deprecated in 7.0.0)

Elasticsearch

- In order to ensure result relevancy Elasticsearch uses a few algorithms to calculate relevant scores

- The default one used is **BM25** (prior to 5.0 it was **tf-idf)**

Elasticsearch

- Provides faster retrieval of documents for more scenarios than a traditional RDBMS
- A traditional RDBMS uses indexes implemented using a B-tree or hash table structures
- The RDBMS indexes pose significant limitations on the types of queries where they can be applied

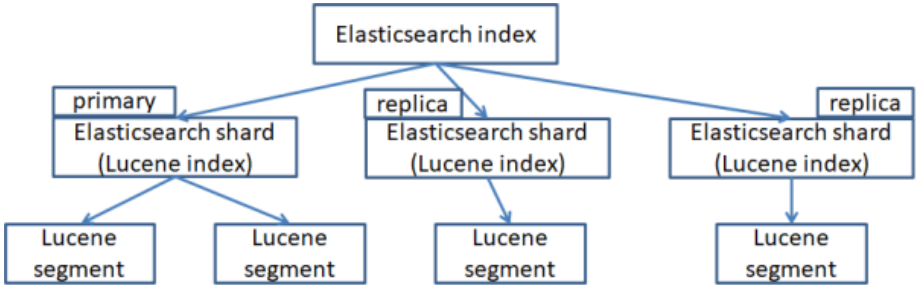
Elasticsearch

- Documents might not have an explicit schema specified
- An explicit schema (mapping) for certain fields can be specified
- Certain fields can match a pattern that identifies their field types (dynamic mapping)
- The same field can be indexed multiple times using different mechanisms

Elasticsearch architecture

Clustering

- Elasticsearch is designed with clustering in mind
- By default each node starts with 1 shard (5 in versions < 7.0.0)
- An Elasticsearch shard is a Lucene index



Clustering

- The more nodes are added to the cluster shards get distributed among nodes
- By default, Elasticsearch tries to balance the number of shards across your nodes so the load is evenly spread
- Partial results can be returned from shards that are still available

Clustering

- The shard for a document is determined based on the following formula:

```
shard = hash(<rounting_key>) % number_of_primary_shards
```

- By default the routing key is the document ID

- A custom routing key can be set during indexing to enable shard routing

Clustering

- By default, new nodes discover existing clusters via multicast
- If a cluster is discovered, the new node joins it only if it has the same cluster name
- If a node on the same instance already runs on the specified port Elasticsearch tries to pick the next available port

Clustering

- Two ways to discover nodes:
  - multicast: automatic discovery of nodes on the network, multicast ping send by default to 224.2.2.4:54328
  - unicast: explicitly specify cluster nodes in the Elasticsearch configuration

```
discovery.seed_hosts = ["10.0.0.3", "10.0.0.4:9300",
                        "10.0.0.5[9300-9400]"]
```

- In unicast discovery not all nodes need to be listed
- The ones that are listed need to know of the other nodes in the cluster

Scaling considerations

- When planning an Elasticsearch cluster several aspects need to be considered:
  - sharding
  - splitting data between indexes
  - maximizing throughout

Sharding considerations

- Too small number of shards introduces a scalability bottleneck
- Too many shards introduces performance and management overhead
- Determining the number of shards should be based on an upfront planning

Data considerations

- Avoid putting huge amounts of data in a single index
- Index allocation strategies might be adopted such as a daily/weekly/yearly index. For example: **orders-20200106**
- Aliases can be used to avoid changing the name of the index

Concurrency control

- Elasticsearch uses optimistic locking for concurrency control
- When indexing a document the **version** attribute can be specified
- If the document already has the specified version the operation is rejected from Elasticsearch
- Concurrency control can also be achieved using **the if_seq_no** and **if_primary_term** parameters of an index request

High availability

- To increase availability, you can create one or more copies (called replicas) for each of your initial shards (called primaries)

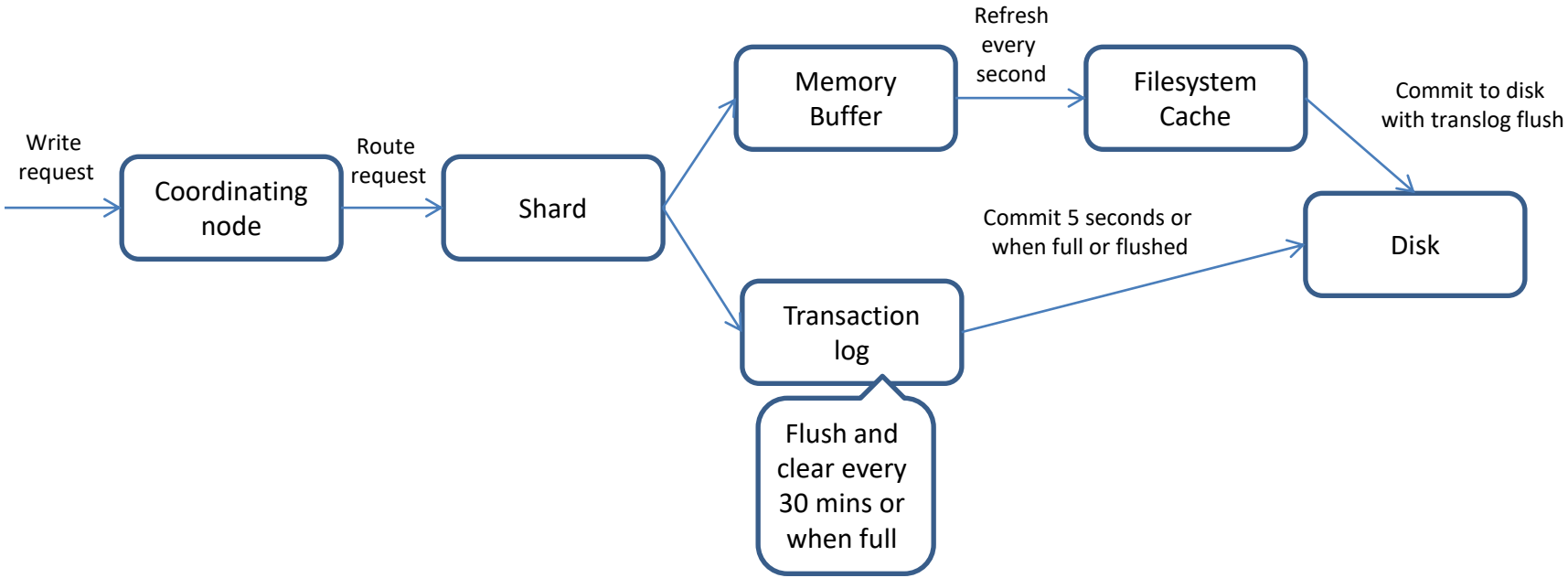
- Once an indexing request is send to a particular shard (determined from a hash of the document's ID) the document being indexes is also sent to the shard's replicas

High availability

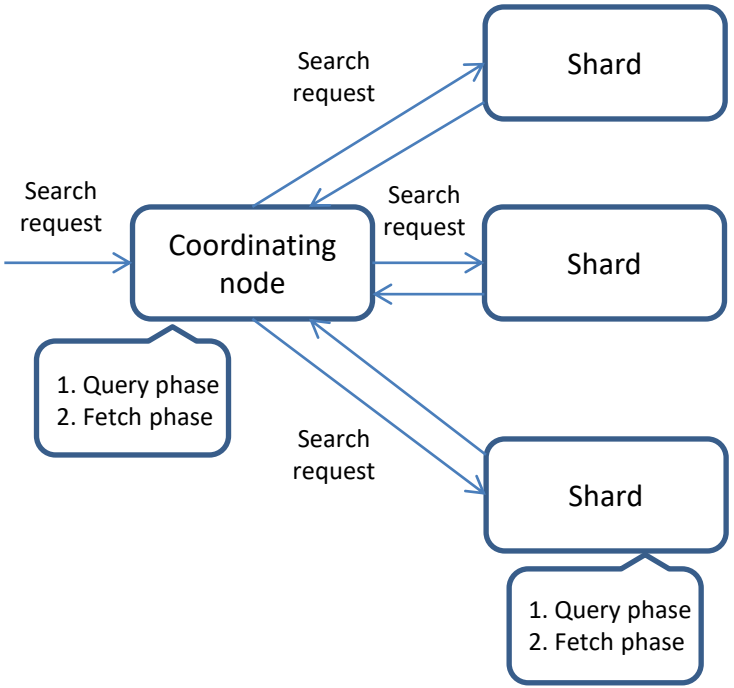
- When you perform a search request Elasticsearch distributes the load among the shards and the replicas

- In that manner replicas are also used to improve performance and not only provide a mechanism for fault-tolerance

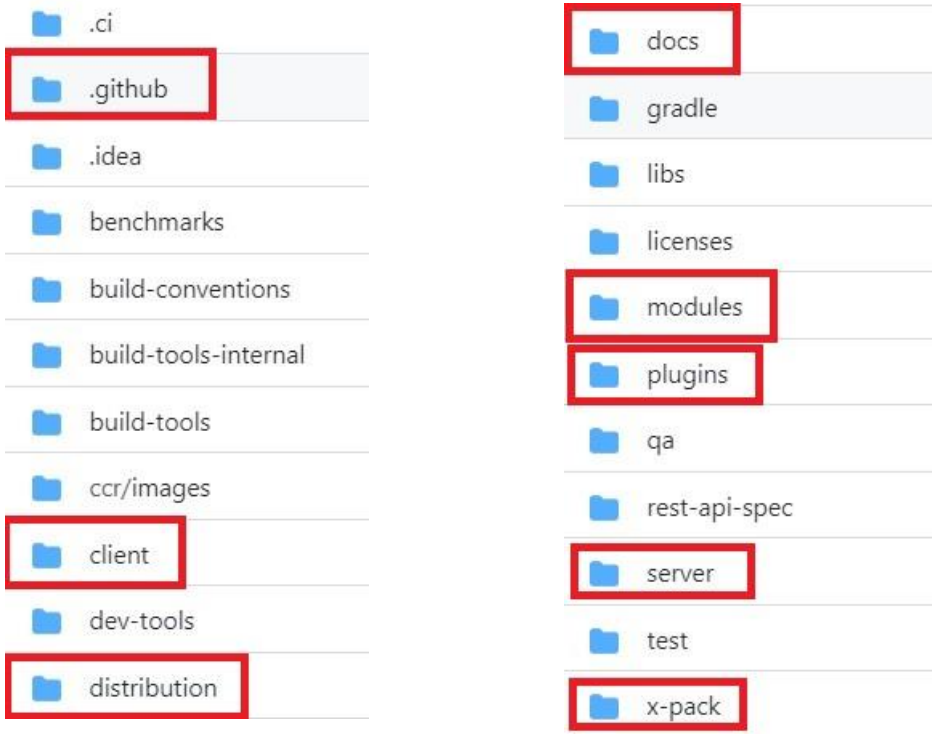
Shard request processing



Shard request processing



Repository structure



Elasticsearch modules

- Internally Elasticsearch is comprised of different modules
- Modules are loaded during Elasticsearch instance startup
- Elasticsearch used a modified version of Google Guice for the module binding

```
// b is a Guice binder
modules.add(b -> {
        b.bind(Node.class).toInstance(this);
        b.bind(NodeService.class).toInstance(nodeService);
        b.bind(PluginsService.class).toInstance(pluginsService);
        b.bind(ThreadPool.class).toInstance(threadPool);
            …
}
```

Elasticsearch startup

org.elasticsearch.bootstrap.Elasticsearch#main()

org.elasticsearch.bootstrap.Elasticsearch#start()

org.elasticsearch.node.Node#start()

Indexing

org.elasticsearch.index.IndexService#createShard() ⇦ shard must be created upfront

org.elasticsearch.index.shard.IndexShard#applyIndexOperationOnPrimary()

⇩

org.elasticsearch.index.shard.IndexShard#index()

⇩

org.elasticsearch.index.engine.InternalEngine#index()

⇩

org.elasticsearch.index.engine.InternalEngine#indexIntoLucene()

⇩

org.elasticsearch.index.engine.InternalEngine#addDocs()

⇩

org.apache.lucene.index.IndexWriter#addDocuments()

Extending Elasticsearch

Elasticsearch plug-ins

- Plug-ins extend the functionality of Elasticsearch
- Located under the **plugins** directory
- The **elasticsearch-plugin** utility can be used to manage plugins

```
bin/elasticsearch-plugin install [core_plugin_name]
```

```
bin/elasticsearch-plugin install [URL]
```

```
bin/elasticsearch-plugin list
```

```
bin/elasticsearch-plugin remove [plugin_name]
```

Elasticsearch plug-ins

- Elasticsearch plug-ins are bundled in a ZIP archive along with their dependencies
- Elasticsearch plug-ins must implement the **org.elasticsearch.plugins.Plugin** class
- An instance of **org.elasticsearch.plugins.PluginService** is responsible to load plug-ins

Elasticsearch plug-in example

- Let’s create a simple plug-in that provides an ingest processor to filter words from a field of an indexed document …

Elasticsearch plug-in example

```
public class FilterIngestPlugin extends Plugin implements IngestPlugin {

        public static final Setting<String> FILTER_WORD = Setting.simpleString("filter.word", Setting.Property.IndexScope,
                                        Setting.Property.Dynamic);
        public static final Setting<String> FILTER_FIELD = Setting.simpleString("filter.field", Setting.Property.IndexScope,
                                        Setting.Property.Dynamic);

        @Override
        public List<Setting<?>> getSettings() {
                return Arrays.asList(FILTER_WORD, FILTER_FIELD);
        }
        @Override
        public Map<String, Processor.Factory> getProcessors(Processor.Parameters parameters) {
                String filterWord = FILTER_WORD.get(parameters.env.settings());
                String filterField = FILTER_FIELD.get(parameters.env.settings());

                Map<String, Processor.Factory> processors = new HashMap<>();
                processors.put(FilterWordProcessor.TYPE, new FilterWordProcessor.Factory(filterWord, filterField));
                return processors;
        }

}
```

Elasticsearch plug-in example

```
public class FilterWordProcessor extends AbstractProcessor {

        …

        @Override
        public IngestDocument execute(IngestDocument ingestDocument) throws Exception {
                IngestDocument document = ingestDocument;
                String value = document.getFieldValue(field, String.class);
                String clearedValue = value.replace(filterWord, "");
                document.setFieldValue(field, clearedValue);
                return document;
        }

        public static final class Factory implements Processor.Factory {
                …
                public Processor create(Map<String, Processor.Factory> registry, String processorTag,
                                        Map<String, Object> config) throws Exception {
                        return new FilterWordProcessor(processorTag, filterWord, field);
                }
        }

}
```

Q&A

Thank you