ELK - ELASTICSEARCH

What's Elasticsearch?

Elasticsearch is a **search server** based on **Lucene** (free open source information retrieval software library). It's 'elastic' in the sense that it's easy to scale horizontally-simply add more nodes to distribute the load. Today, many companies, including Wikipedia, eBay, GitHub, and Datadog, use it to store, search, and analyze large amounts of data on the fly. Elasticsearch represents data in the form of structured JSON documents, and makes full-text search accessible via RESTful API and web clients for languages like PHP, Python, and Ruby. In Elasticsearch, related data is often stored in the same **index**, which can be thought of as the equivalent of a logical wrapper of configuration.

Each index contains a set of related **documents** in JSON format.

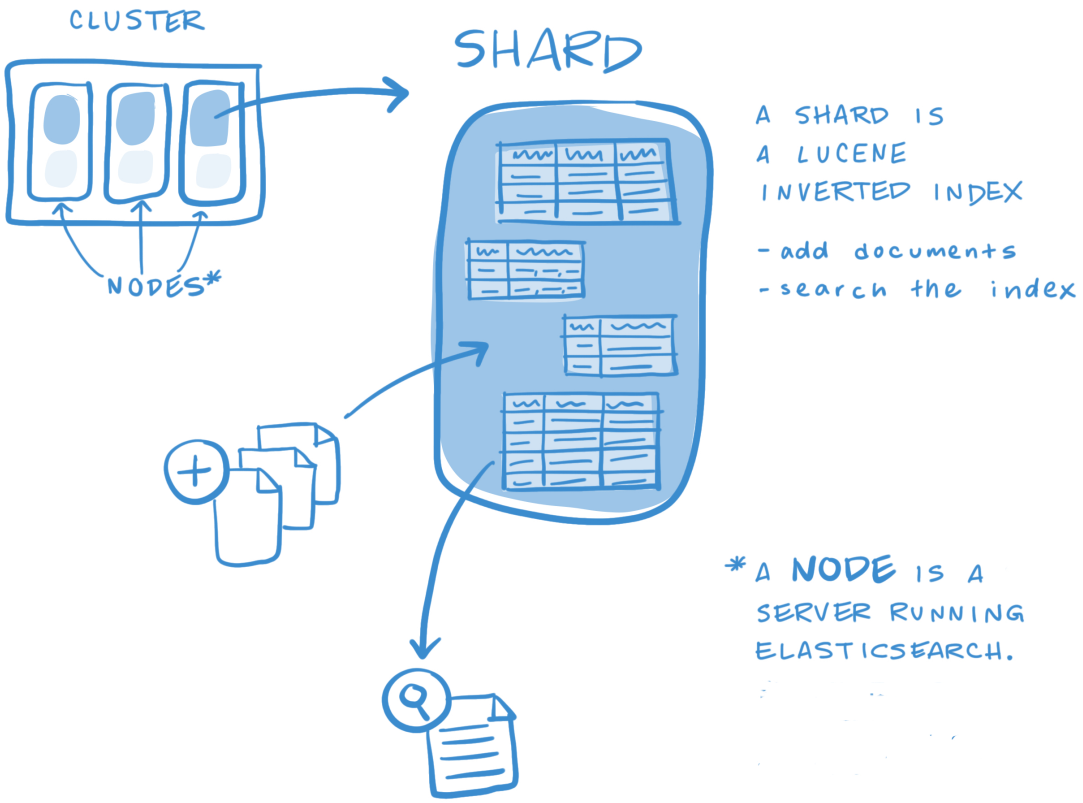
Elasticsearch's secret sauce for full-text search is Lucene's inverted index.

When a document is indexed, Elasticsearch automatically creates an inverted index for each field; the inverted index maps terms to the documents that contain those terms.

Elasticsearch is developed in Java and is released as open source.

1. Distributed and Scalable search engine.
2. Based on Lucene.
3. Hide Lucene complexity by exposing all services : HTTP/REST/JSON
4. Horizontal scaling, replication, fail over, load balancing.
5. Fast!
6. It's a search engine NOT a search tool.

Note : Elasticsearch is a near real time search platform. What this means is there is a slight latency (normally one second) from the time we index a document until the time it becomes searchable.
This is an important distinction from other platforms like SQL wherein data is immediately available after a transaction is completed.
In Elasticsearch, a cluster is made up of one or more nodes, as shown below.
An index is stored across one or more **primary shards**, and zero or more **replica shards**, and each shard is a complete instance of Lucene, like a mini search engine.

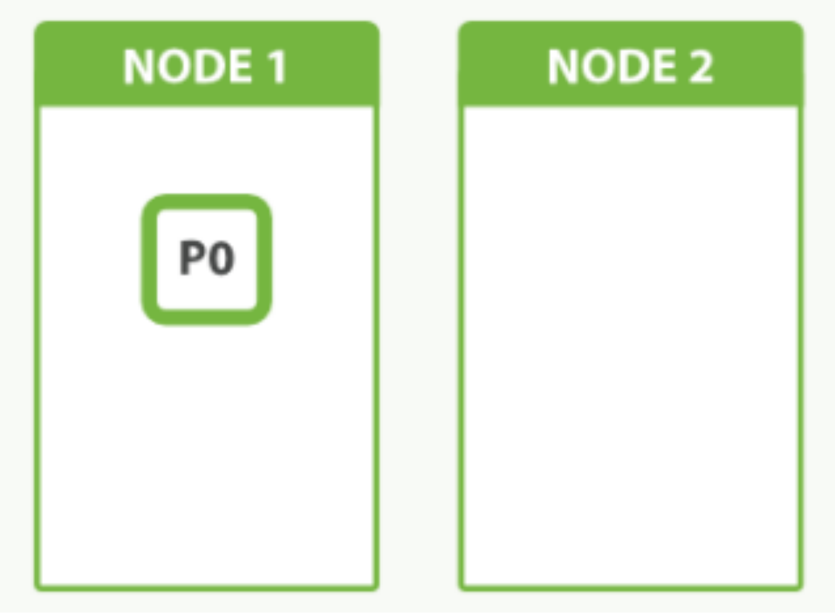


Other key concepts of Elasticsearch are **replicas** and **shards**, the mechanism Elasticsearch uses to distribute data around the cluster. The index is a logical namespace which maps to one or more primary shards and can have zero or more replica shards. A shard is a Lucene index and that an Elasticsearch index is a collection of shards. Our application talks to an **index**, and Elasticsearch routes our requests to the appropriate **shards**.

The smallest index we can have is one with a single shard. This setup may be small, but it may serve our current needs and is cheap to run. Suppose that our cluster consists of one node, and in our cluster we have one index, which has only one shard: an index with one primary shard and zero replica shards.

```
PUT /my_index
{
  "settings": {
    "number_of_shards":   1,
    "number_of_replicas": 0
  }
}
```

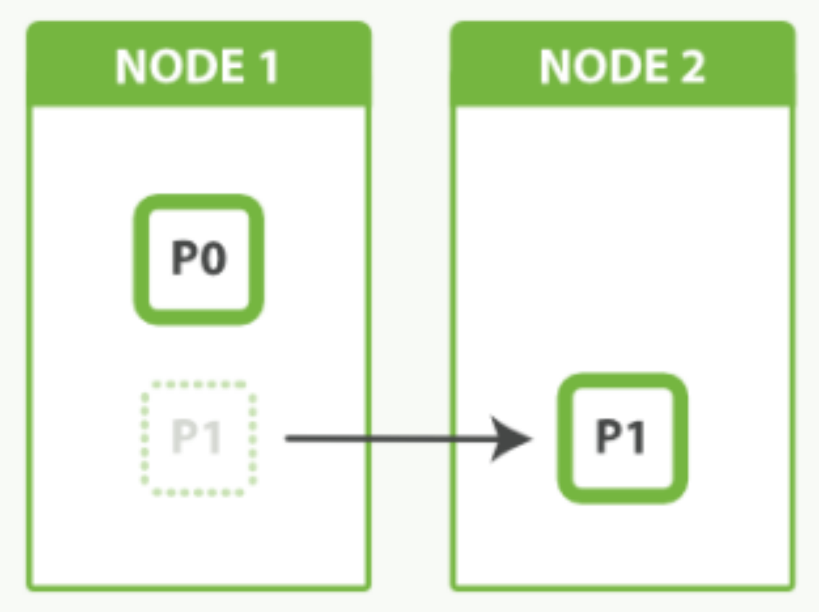
However, as time goes on, a single node just can't keep up with the traffic, and we decide to add a second node. What will happens? Nothing.



Because we have only one shard, there is nothing to put on the second node. We can't increase the number of shards in the index, because the number of shards is an important element in the algorithm used to route documents to shards:

shard = hash(routing) % number_of_primary_shards

We should have planned like this:

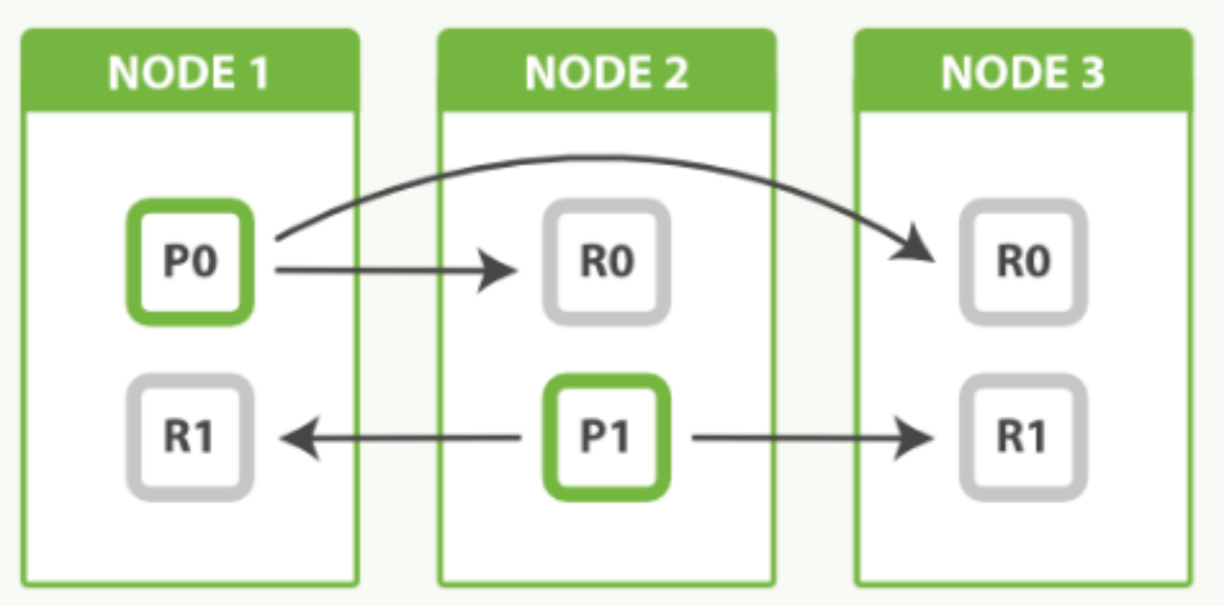


Our only option now is to reindex our data into a new, bigger index that has more shards, but that will take time that we can ill afford. By planning ahead, we could have avoided this problem completely by Shard Overallocation.

The main purpose of **replicas** is for failover: if the node holding a primary shard dies, a replica is promoted to the role of primary.

At index time, a replica shard does the same amount of work as the primary shard. New documents are first indexed on the primary and then on any replicas. Increasing the number of replicas does not change the capacity of the index.

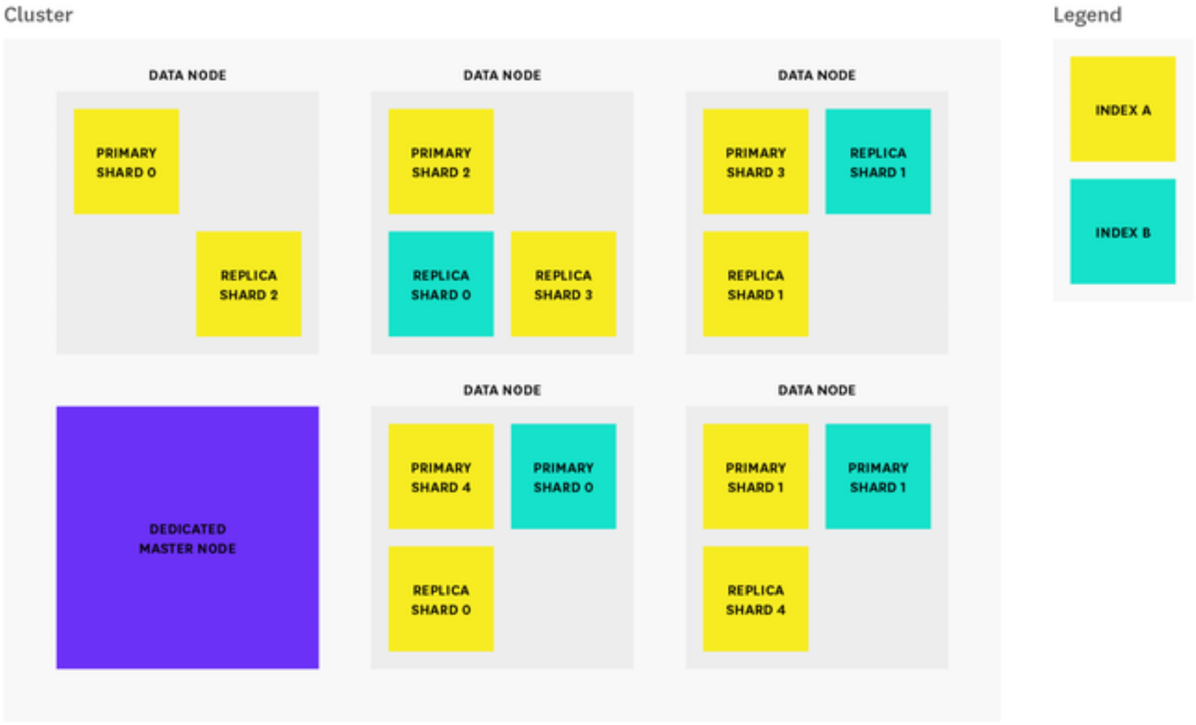
However, replica shards can serve read requests. If, as is often the case, our index is search heavy, we can increase search performance by increasing the number of replicas, but only if we also add extra hardware.



In the picture above, we have 3 nodes with 2 primary and 2 replicas. The fact that node 3 holds two replicas and no primaries is not important. Replicas and primaries do the same amount of work; they just play slightly different roles. There is no need to ensure that primaries are distributed evenly across all nodes.

Each node is a single running instance of Elasticsearch, and its configuration

file (**elasticsearch.yml**) designates which cluster it belongs to (**cluster.name**) and what type of node it can be.
The diagram below shows that we constructed 1 **dedicated master node** and 5 **data nodes**.



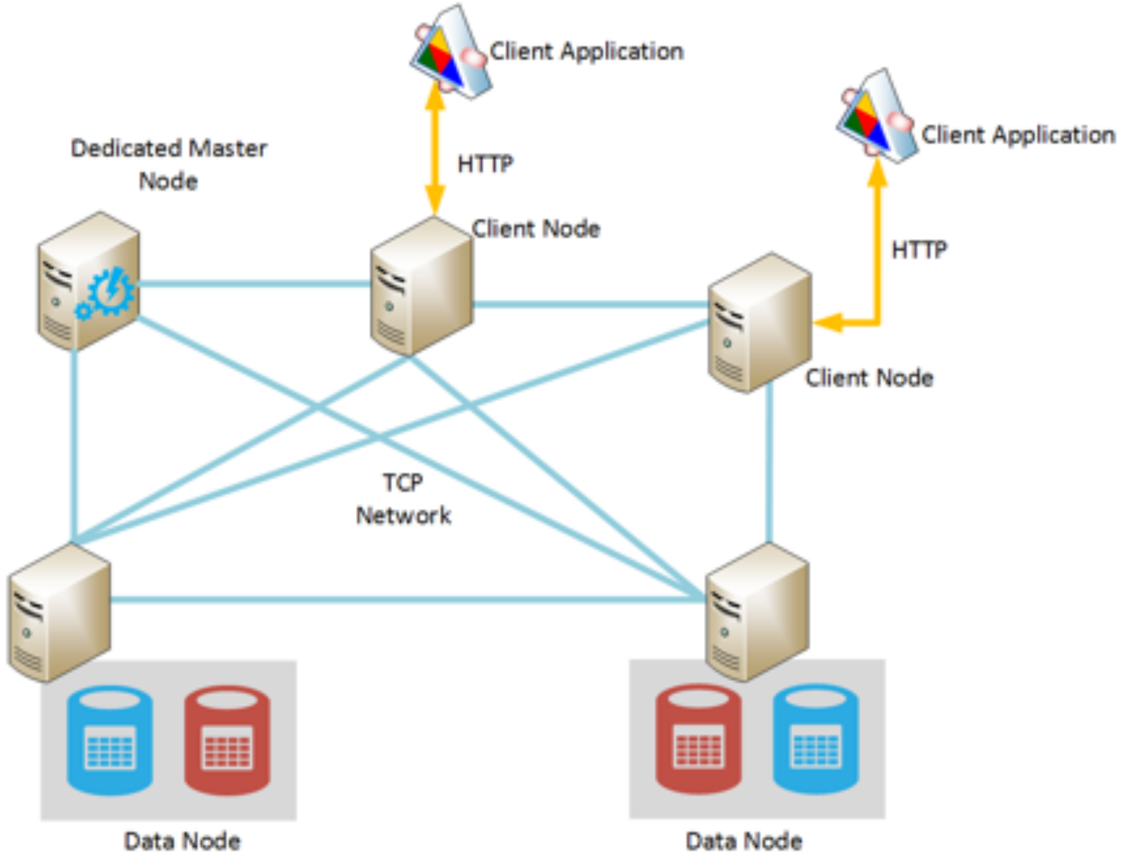
Note: By default, each index in Elasticsearch is allocated 5 primary shards and 1 replica which means that if we have at least two nodes in our cluster, our index will have 5 primary shards and another 5 replica shards (1 complete replica) for a total of 10 shards per index.

The number of primary shards cannot be changed once an index has been created, so choose carefully, or we will likely need to reindex later on.

The number of replicas can be updated later on as needed. To protect against data loss, the master node ensures that each replica shard is not allocated to the same node as its primary shard.

Types of nodes

Elasticsearch has three types of nodes:



1. **Master-eligible nodes**: By default, every node is master-eligible unless otherwise specified. Each cluster automatically elects a master node from all of the master-eligible nodes. In the event that the current master node experiences a failure (such as a power outage, hardware failure, or an out-of-memory error), master-eligible nodes

elect a new master.

The master node is responsible for coordinating cluster tasks like distributing shards across nodes, and creating and deleting indices.

Any master-eligible node is also able to function as a data node. However, in larger clusters, users may launch dedicated master-eligible nodes that do not store any data (by adding **node.data: false** to the config file), in order to improve reliability.

In high-usage environments, moving the master role away from data nodes helps ensure that there will always be enough resources allocated to tasks that only master-eligible nodes can handle.

2.

3. **Data nodes**: By default, every node is a data node that stores data in the form of shards (more about that in the section below) and performs actions

related to indexing, searching, and aggregating data.

In larger clusters, we may choose to create dedicated data nodes by adding **node.master: false** to the config file, ensuring that these nodes have enough resources to handle data-related requests without the additional workload of cluster-related administrative tasks.

4.
5. **Client nodes**: If we set **node.master** and **node.data** to **false**, we will end up with a client node, which is designed to act as a load balancer that helps route indexing and search requests.

   Client nodes do not hold index data but that handle incoming requests made by client applications to the appropriate data node.

   Client nodes help shoulder some of the search workload so that data and master-eligible nodes can focus on their core tasks. Depending on our use case, client nodes may not be necessary because data nodes are able to handle request routing on their own. However, adding client nodes to our cluster makes sense if our search/index workload is heavy enough to benefit from having dedicated client nodes to help route requests.

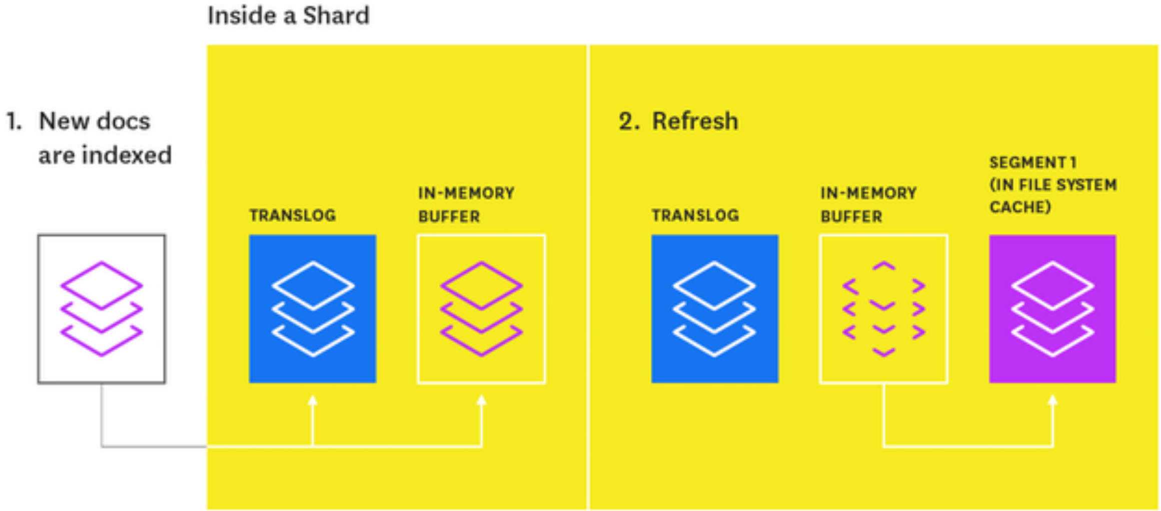
Index update process

In this section, we'll explore the process by which Elasticsearch updates an index. When new information is added to an index, or existing information is updated or deleted, each shard in the index is updated via two processes: **refresh** and **flush**.

Index refresh

Newly indexed documents are not

immediately made available for search. First, they are written to an in-memory buffer where they await the next index refresh, which occurs once per second by default. The refresh process creates a new in-memory segment from the contents of the in-memory buffer (making the newly indexed documents searchable), then empties the buffer:



Shards of an index are composed of multiple **segments**. The core data structure from Lucene, a segment is essentially a change set for the index.

These segments are created with every refresh and subsequently merged together over time in the background to ensure efficient use of resources.

Every time an index is searched, a primary or replica version of each shard must be searched by, in turn, searching every segment in that shard.

A segment is immutable, so updating a document means:

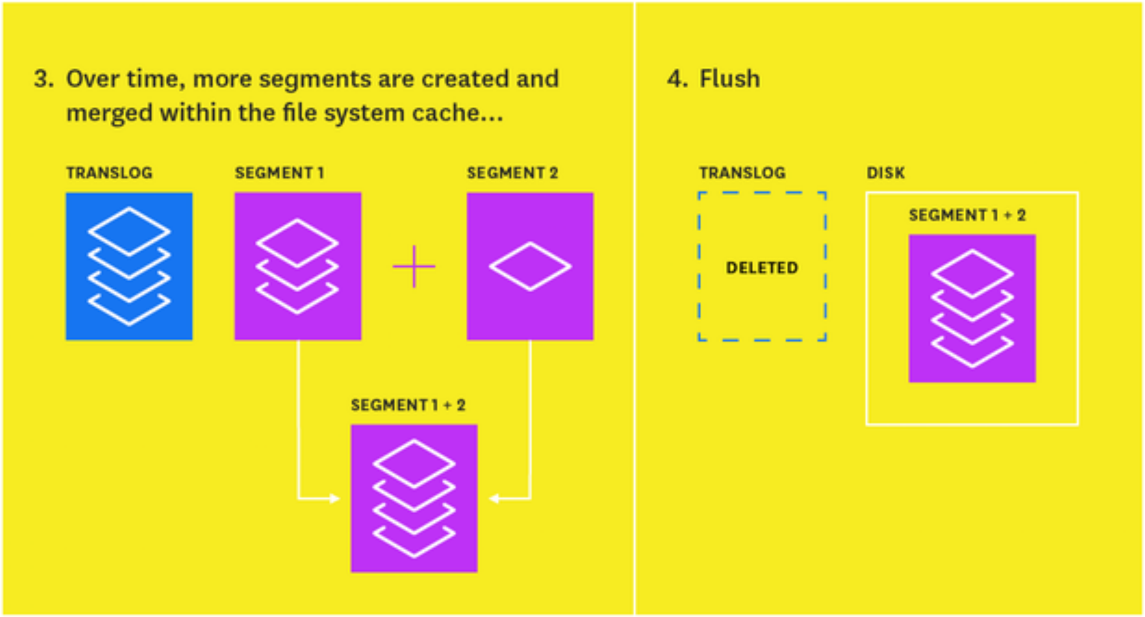
1. writing the information to a new segment during the refresh process
2. marking the old information as deleted

The old information is eventually deleted when the outdated segment is merged with another segment.

Index flush

At the same time that newly indexed documents are added to the in-memory buffer, they are also appended to the shard's translog: a persistent, write-ahead transaction log of operations. Every 30 minutes, or whenever the translog reaches a maximum size (by default, 512MB), a flush is triggered. During a flush, any documents in the in-memory buffer are refreshed (stored on new segments), all in-memory segments are committed to disk, and the translog is cleared.

Inside a Shard



3. Over time, more segments are created and merged within the file system cache...

4. Flush

The translog helps prevent data loss in the event that a node fails. It is designed to help a shard recover operations that may otherwise have been lost between flushes. The log is committed to disk every 5 seconds, or upon each successful index, delete, update, or bulk request (whichever occurs first).

When do we want to use ES?

Here are a few sample use-cases of Elasticsearch from Getting Started.

1. You run an online web store where you allow your customers to search for products that you sell. In this case, you can use Elasticsearch to store your entire product catalog and inventory and provide search and autocomplete suggestions for them.
2. You want to collect log or transaction data and you want to analyze and mine this data to look for trends, statistics, summarizations, or anomalies. In this case, you can use Logstash (part of the Elasticsearch/Logstash/Kibana stack) to collect, aggregate,

and parse your data, and then have Logstash feed this data into Elasticsearch. Once the data is in Elasticsearch, you can run searches and aggregations to mine any information that is of interest to you.

3. You run a price alerting platform which allows price-savvy customers to specify a rule like "I am interested in buying a specific electronic gadget and I want to be notified if the price of gadget falls below $X from any vendor within the next month". In this case you can scrape vendor prices, push them into Elasticsearch and use its reverse-search (Percolator) capability to match price movements against customer

queries and eventually push the alerts out to the customer once matches are found.

4. You have analytics/business-intelligence needs and want to quickly investigate, analyze, visualize, and ask ad-hoc questions on a lot of data (think millions or billions of records). In this case, you can use Elasticsearch to store your data and then use Kibana (part of the Elasticsearch/Logstash/Kibana stack) to build custom dashboards that can visualize aspects of your data that are important to you. Additionally, you can use the Elasticsearch aggregations functionality to perform complex business intelligence queries against your data.