If using java to talk to Elasticsearch, there are two clients

Node Client, Transport client

Both Java clients talk to the cluster over port 9300, using the native Elasticsearch transport protocol. The nodes in the cluster also communicate with each other over port 9300. If this port is not open, your nodes will not be able to form a cluster.

**What is a B-tree index? Traditional databases use this**

(1)Search Lite - query-string search

curl -XGET 'localhost:9200/megacorp/employee/\_search?q=last\_name:Smith&pretty'

equivalent query DSL to (1)

(2)Search query DSL

curl -XGET 'localhost:9200/megacorp/employee/\_search?pretty' -H 'Content-Type: application/json' -d'

{

"query" : {

"match" : {

"last\_name" : "Smith"

}

}

}

'

Rules for index name in Elasticsearch

All we have to do is choose an index name. This name must be lowercase, cannot begin with an underscore, and cannot contain commas. Let’s use website as our index name.

Rules for type in Elasticsearch

A \_type name can be lowercase or uppercase, but shouldn’t begin with an underscore or period. It also may not contain commas, and is limited to a length of 256 characters. We will use blog for our type name.

Rules for id in Elasticsearch

The ID is a string that, when combined with the \_index and \_type, uniquely identifies a document in Elasticsearch. When creating a new document, you can either provide your own \_id or let Elasticsearch generate one for you.

Retrieving Part of a Documentedit

By default, a GET request will return the whole document, as stored in the \_source field. But perhaps all you are interested in are the title and text fields. Individual fields can be requested by using the \_source parameter. Multiple fields can be specified in a comma-separated list:

GET /website/blog/123?\_source=title,text

Or if you want just the \_source field without any metadata, you can use the \_source endpoint:

GET /website/blog/123/\_source

Checking Whether a Document Existsedit

If all you want to do is to check whether a document exists—you’re not interested in the content at all—then use the HEAD method instead of the GET method. HEAD requests don’t return a body, just HTTP headers:

curl -i -XHEAD http://localhost:9200/website/blog/123

Elasticsearch will return a 200 OK status code if the document exists:

Creating a New Documented

How can we be sure, when we index a document, that we are creating an entirely new document and not overwriting an existing one?

Use POST method

However, if we already have an \_id that we want to use, then we have to tell Elasticsearch that it should accept our index request only if a document with the same \_index, \_type, and \_id doesn’t exist already. There are two ways of doing this, both of which amount to the same thing. Use whichever method is more convenient for you.

The first method uses the op\_type query -string parameter:

PUT /website/blog/123?op\_type=create

{ ... }

And the second uses the /\_create endpoint in the URL:

PUT /website/blog/123/\_create

{ ... }

If the request succeeds in creating a new document, Elasticsearch will return the usual metadata and an HTTP response code of 201 Created.

On the other hand, if a document with the same \_index, \_type, and \_id already exists, Elasticsearch will respond with a 409 Conflict response code,

**Life Inside a Cluster**

A node is a running instance of Elasticsearch, while a cluster consists of one or more nodes with the same cluster.name that are working together to share their data and workload.

**curl -XGET 'localhost:9200/\_cluster/health?pretty'**

**Add Failover**

Starting a second node.

[https://www.elastic.co/guide/en/elasticsearch/guide/master/\_add\_failover.html#\_add\_failover](https://www.elastic.co/guide/en/elasticsearch/guide/master/_add_failover.html" \l "_add_failover)

[https://www.elastic.co/guide/en/elasticsearch/guide/master/important-configuration-changes.html#unicast](https://www.elastic.co/guide/en/elasticsearch/guide/master/important-configuration-changes.html" \l "unicast)

**Data In, Data Out**

What is an inverted index

**What is a Document**

Field names can be any valid string, but may not include periods.

Index naming

this name must be lowercase, cannot begin with an underscore, and cannot contain commas. Let’s use website as our index name.

type naming

A \_type name can be lowercase or uppercase, but shouldn’t begin with an underscore or period. It also may not contain commas, and is limited to a length of 256 characters. We will use blog for our type name.

id

The ID is a **string**

**Retrieving a document**

<https://www.elastic.co/guide/en/elasticsearch/guide/master/get-doc.html>

Checking whether a document exists

If all you want to do is to check whether a document exists—you’re not interested in the content at all—then use the HEAD method instead of the GET method. HEAD requests don’t return a body, just HTTP headers:

curl -i -XHEAD http://localhost:9200/website/blog/123

Elasticsearch will return a 200 OK status code if the document exists:

HTTP/1.1 200 OK

Content-Type: text/plain; charset=UTF-8

Content-Length: 0

And a 404 Not Found if it doesn’t exist:

Search Lite - Query string search

GET /megacorp/employee/\_search?q=last\_name:Smith

Query DSL (domain specific language)

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"last\_name" : "Smith"

}

}

}

Query to match last name

GET /megacorp/employee/\_search

{

"query" : {

"bool" : {

"must" : {

"match" : {

"last\_name" : "smith" https://www.elastic.co/guide/en/elasticsearch/guide/2.x/images/icons/callouts/1.png

}

},

"filter" : {

"range" : {

"age" : { "gt" : 30 } https://www.elastic.co/guide/en/elasticsearch/guide/2.x/images/icons/callouts/2.png

}

}

}

}

}

Query to match last name AND age greater than 30

Difference between match and match\_phrase

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"about" : "rock climbing"

}

}

}

Match will return results that contain rock and/or climbing and arrange results by a “relevance” score

Match\_phrase will return exact matches for “rock climbing”

Red, Yellow, Green status – CLUSTER HEALTH

green

All primary and replica shards are active.

yellow

All primary shards are active, but not all replica shards are active.

red

Not all primary shards are active.

LIFE INSIDE A CLUSTER

A *node* is a running instance of Elasticsearch, while a *cluster* consists of one or more nodes with the same cluster.name that are working together to share their data and workload. As nodes are added to or removed from the cluster, the cluster reorganizes itself to spread the data evenly.

PUT /blogs

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}

the above actually means – create an index called “blogs”, assign it to 3 primary shards, and also assign 1 replica shard for each primary shard. There for there would be a total of 6 shards : 3 primary and 3 replicas. Additionally 3 primary shards and 3 replica shards operating on the same node is leaves us at risk. The replica nodes will not be active and our cluster health will be YELLOW. Only after adding nodes ( Elastic search will move shards around accordingly) that the replica shards can spread themselves out evenly to provide failover protection and change cluster health to green. i.e. all primary and replica shards are active

Data In, Data Out

In Elasticsearch, *all data in every field* is *indexed by default*. That is, every field has a dedicated inverted index for fast retrieval. And, unlike most other databases, it can use all of those inverted indices *in the same query*, to return results at breathtaking speed.

When using POST to create a document. eg.

POST /website/blog/

{

"title": "My second blog entry",

"text": "Still trying this out...",

"date": "2014/01/01"

}

the id is auto generated.

Autogenerated IDs are 20 character long, URL-safe, Base64-encoded GUID strings. These GUIDs are generated from a modified FlakeID scheme which allows multiple nodes to be generating unique IDs in parallel with essentially zero chance of collision.

Retrieving a document

If we were to request a document that doesn’t exist, we would still get a JSON response, but found would be set to false.

Also, the HTTP response code would be 404 Not Found instead of 200 OK. We can see this by passing the -i argument to curl, which causes it to display the response headers:

curl -i -XGET http://localhost:9200/website/blog/124?pretty

also you can use curl **-I** -XGET <http://localhost:9200/website/blog/124?pretty>

the -i and -I are very subtly different.

CHECK THAT A DOCUMENT EXISTS

curl -i **-XHEAD** http://localhost:9200/website/blog/123

notice use of the http verb HEAD

RETRIEVING PART OF A DOCUMENT

What if you only want some of the fields in your document

GET /website/blog/123?**\_source=title,text**

**GET** requests will return meta data + \_source

GET /website/blog/123?**\_source=title,text –** will return only the fields you specify

GET /website/blog/123/**\_source –** will return no metadata just srouce

GET /website/blog/123/**\_source?\_source=title,text –** will return no metadata just source AND only those fields specified

Update a **whole** document

after we update the ES response will look like the following

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : "123",

"\_version" : 2,

"created": false 

}

CREATE NEW DOCUMENT

We have decided to create a new document and also have an id we want to use (as opposed to letting ES auto generate it for us)

To ensure we don’t overwrite an existing document and thus tell ES to only accept our document if and only if there is no document with the same index/type/id…………… to do this there are two ways

The first method uses the op\_type query -string parameter:

PUT /website/blog/123?op\_type=create

{ ... }

And the second uses the /\_create endpoint in the URL:

PUT /website/blog/123/\_create

{ ... }

If the request succeeds in creating a new document, Elasticsearch will return the usual metadata and an HTTP response code of 201 Created.

On the other hand, if a document with the same \_index, \_type, and \_id already exists, Elasticsearch will respond with a 409 Conflict response code, and an error message like the following:

**When deleting a document**

Even though the document doesn’t exist (found is false), the \_version number has still been incremented. This is part of the internal bookkeeping, which ensures that changes are applied in the correct order across multiple nodes.

Concurrency and dealing with conflicts

In the database world, two approaches are commonly used to ensure that changes are not lost when making concurrent updates:

*Pessimistic concurrency control*

Widely used by relational databases, this approach assumes that conflicting changes are likely to happen and so blocks access to a resource in order to prevent conflicts. A typical example is locking a row before reading its data, ensuring that only the thread that placed the lock is able to make changes to the data in that row.

*Optimistic concurrency control*

Used by Elasticsearch, this approach assumes that conflicts are unlikely to happen and doesn’t block operations from being attempted. However, if the underlying data has been modified between reading and writing, the update will fail. It is then up to the application to decide how it should resolve the conflict. For instance, it could reattempt the update, using the fresh data, or it could report the situation to the user.

Suppoe we create a document for the first time. It will have a version number of 1.

Then suppose we want to update it, naturally we will assume its version number is still 1.

So to update we can do something like the following

PUT /website/blog/1?version=1 

{

"title": "My first blog entry",

"text": "Starting to get the hang of this..."

}

if the version number is still 1 (i.e. no one else updated the document before we updated) then the above operation succeeds.

A failed message (when there is a conflict looks like

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[blog][1]: version conflict, current [2], provided [1]",

"index": "website",

"shard": "3"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[blog][1]: version conflict, current [2], provided [1]",

"index": "website",

"shard": "3"

},

"status": 409

}

All APIs that update or delete a document accept a version parameter, which allows you to apply optimistic concurrency control to just the parts of your code where it makes sense.

### Using Versions from an External System[edit](https://github.com/elastic/elasticsearch-definitive-guide/edit/2.x/030_Data/40_Version_control.asciidoc)

A common setup is to use some other database as the primary data store and Elasticsearch to make the data searchable, which means that all changes to the primary database need to be copied across to Elasticsearch as they happen. If multiple processes are responsible for this data synchronization, you may run into concurrency problems similar to those described previously.

If your main database already has version numbers—or a value such as timestamp that can be used as a version number—then you can reuse these same version numbers in Elasticsearch by adding version\_type=external to the query string. Version numbers must be integers greater than zero and less than about 9.2e+18--a positive long value in Java.

The way external version numbers are handled is a bit different from the internal version numbers we discussed previously. Instead of checking that the current \_version is *the same* as the one specified in the request, Elasticsearch checks that the current \_version is *less than* the specified version. If the request succeeds, the external version number is stored as the document’s new \_version.

External version numbers can be specified not only on index and delete requests, but also when *creating* new documents.

For instance, to create a new blog post with an external version number of 5, we can do the following:

PUT /website/blog/2?version=5&version\_type=external

{

"title": "My first external blog entry",

"text": "Starting to get the hang of this..."

}

Partial Update to document

For instance, we could add a tags field and a views field to our blog post as follows:

POST /website/blog/1/**\_update**

{

"doc" : {

"tags" : [ "testing" ],

"views": 0

}

}

**Using Scripts to Make Partial Updates**