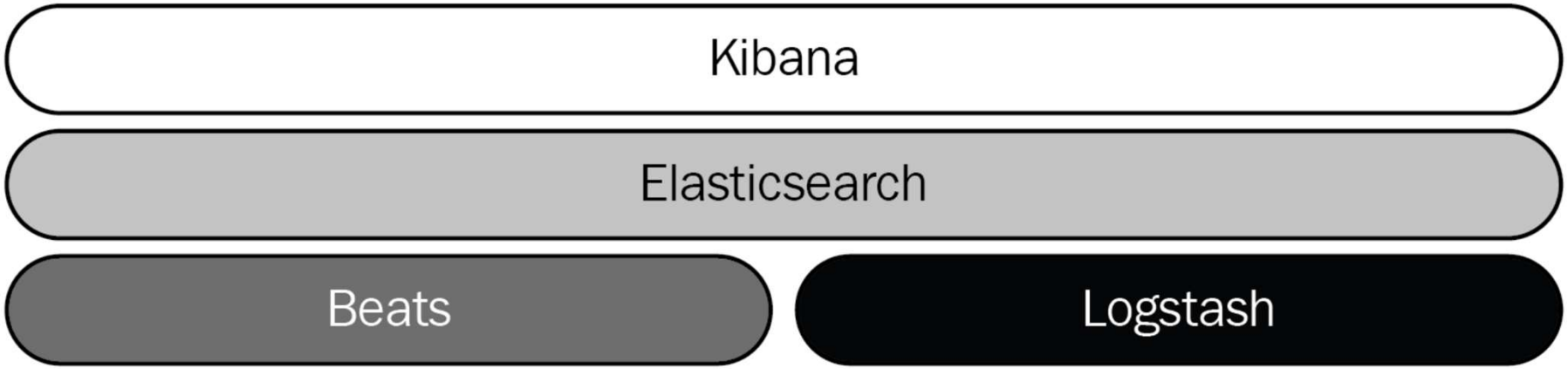
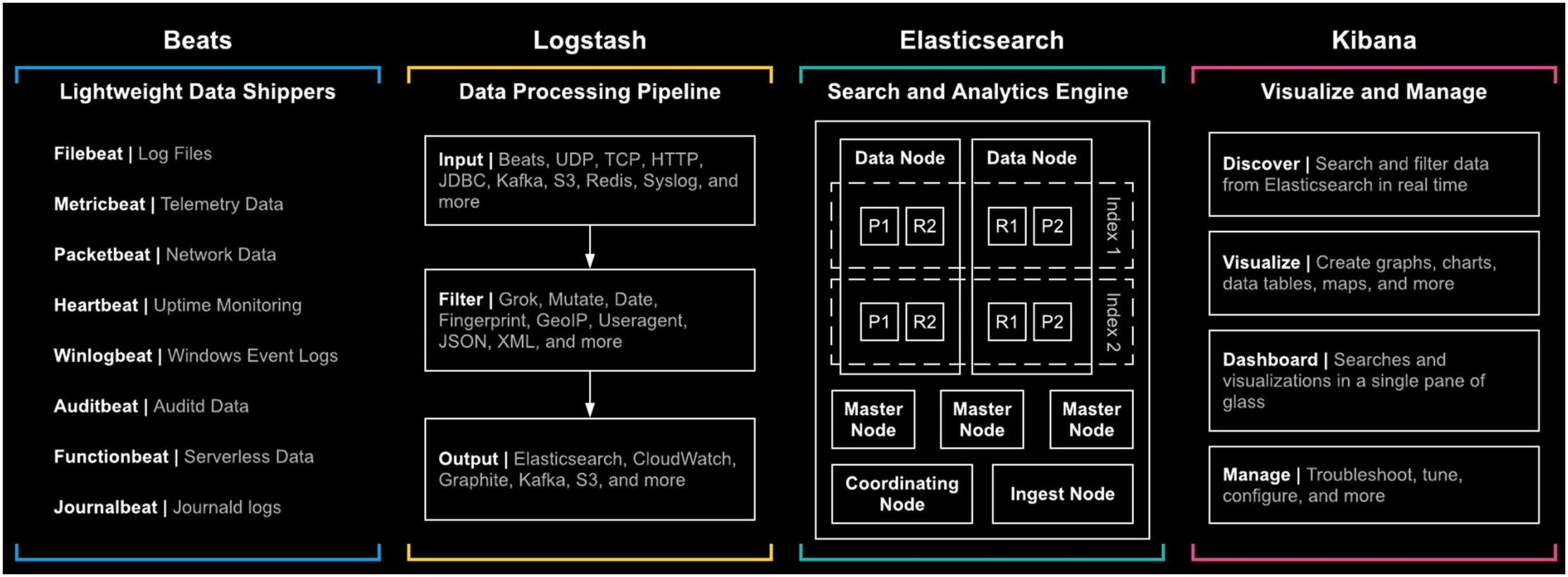
Elasticsearch

Getting Started

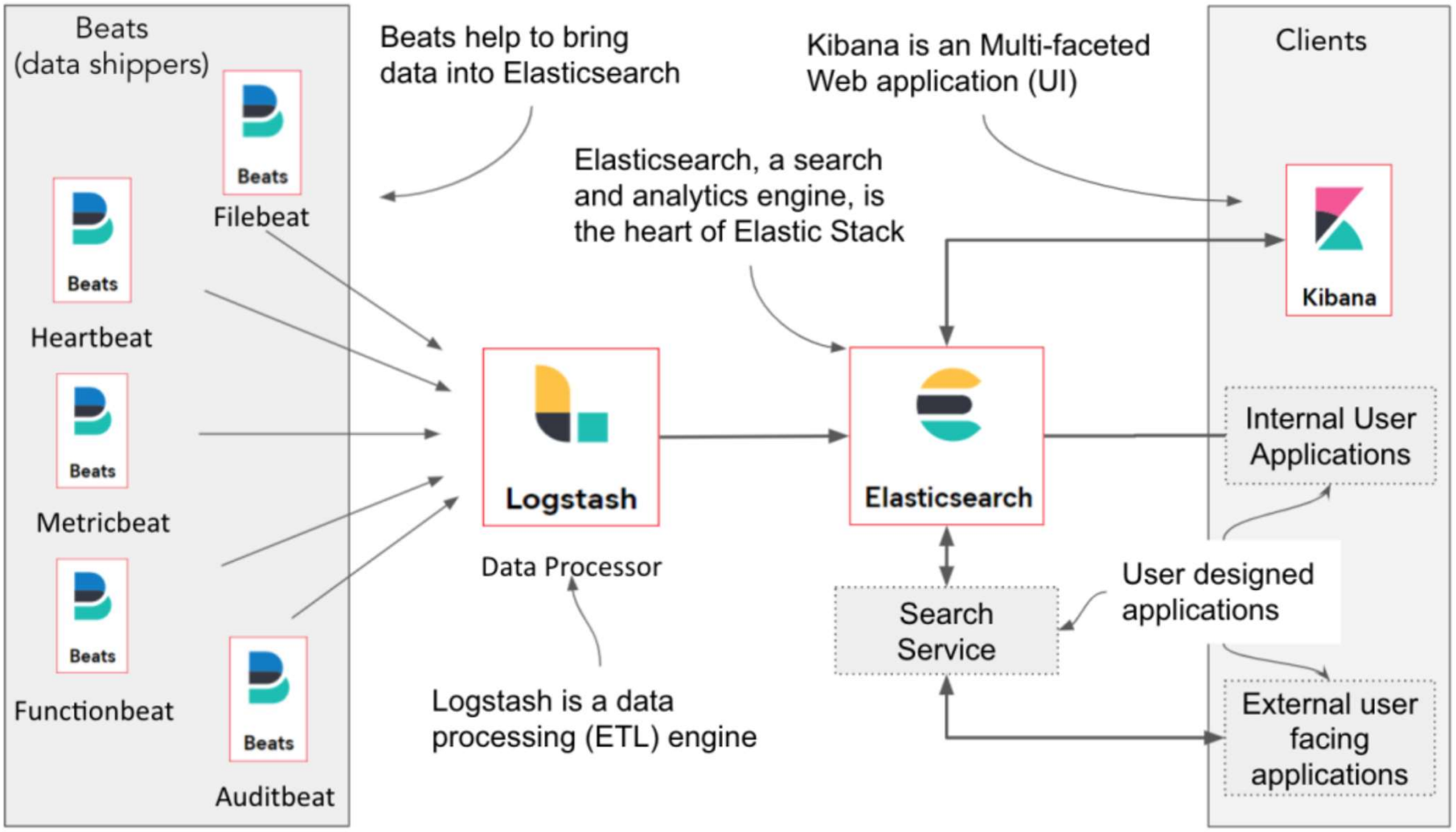
Components of the Elastic Stack



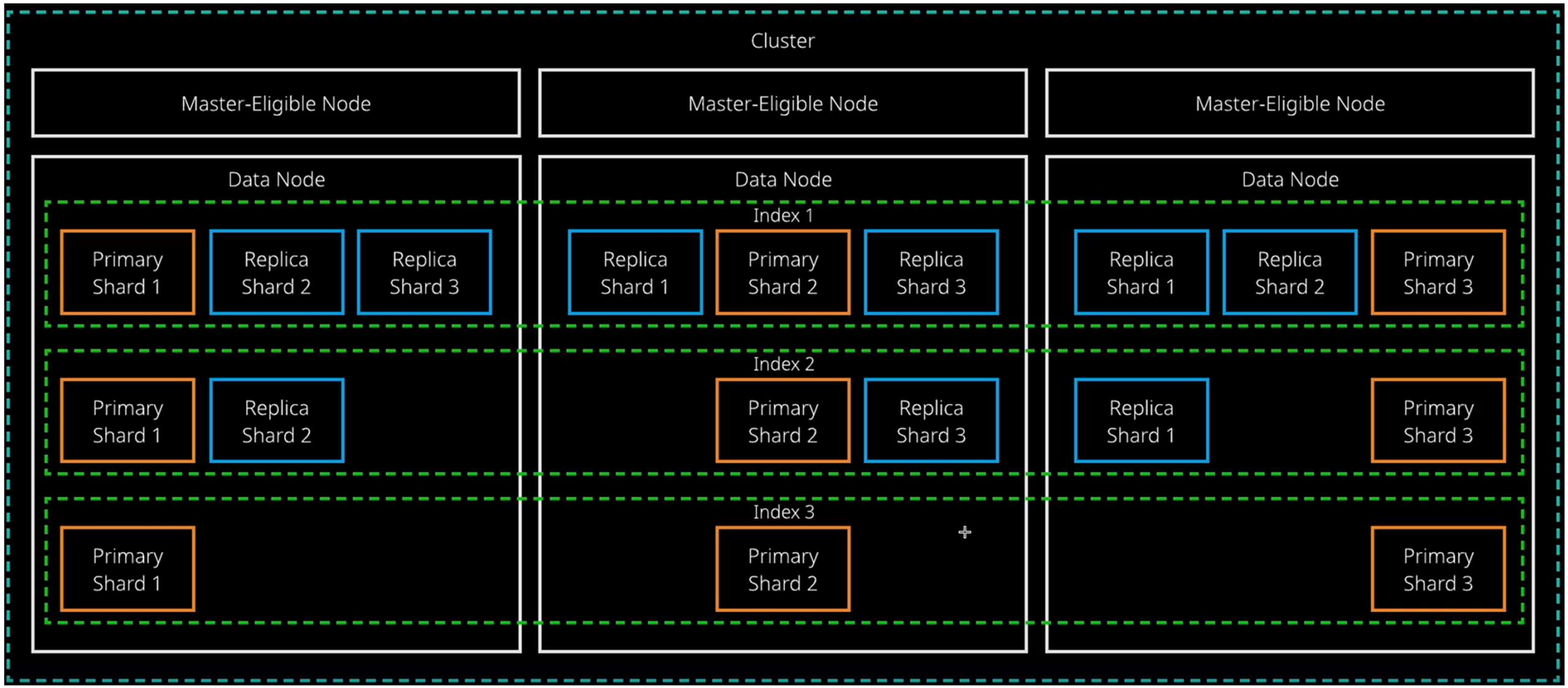
Components of the Elastic Stack



Elastic Stack ecosystem



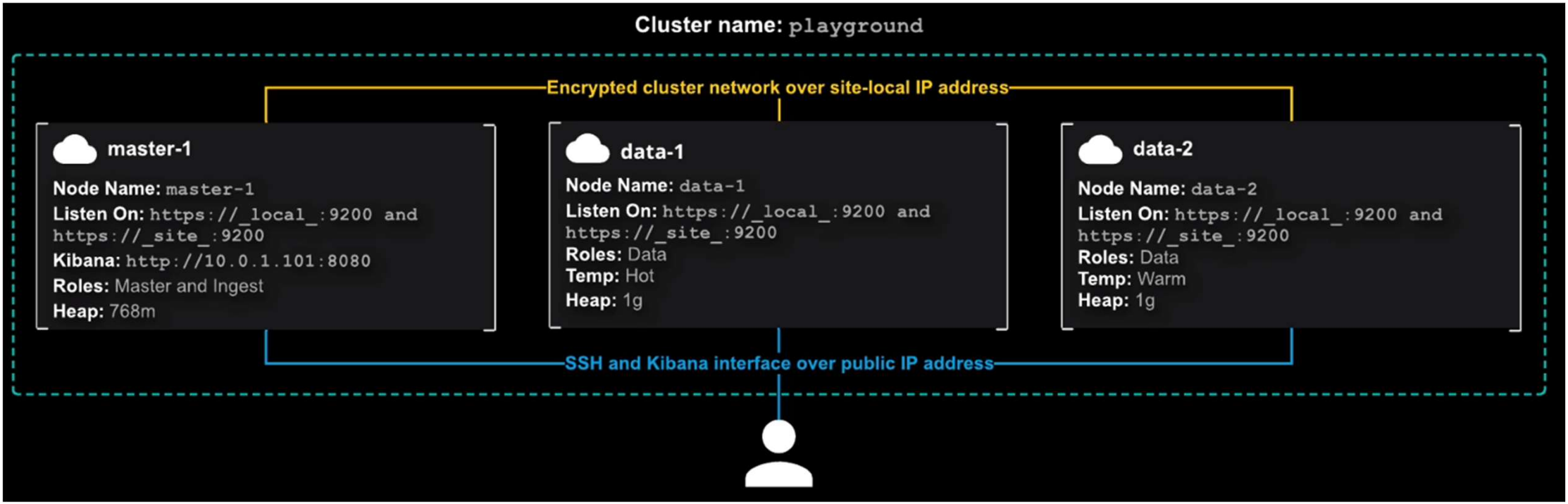
Elasticsearch



Hands-on

Installation and Configuration

Our deployment Architecture



Hands-on

Security

Hands-on

Index and Data Model

Data model for a book represented in a tabular format

Field	Explanation	Example
`title`	The title of a book	`"Effective Java"`
`author`	The author of the book	`"Joshua Bloch"`
`release_date`	Data of release	`01-06-2001`
`amazon_rating`	Average rating on Amazon	`4.7`
`best_seller`	Flag that qualifies the book as a best seller	`true`

`prices`	Inner object with individual prices in three currencies	`"prices":{`
		`  "usd":9.95,`
		`  "gbp":7.95,`
		`  "eur":8.95`
		`}`

A JSON representation of a book entity

Indexing two more documents

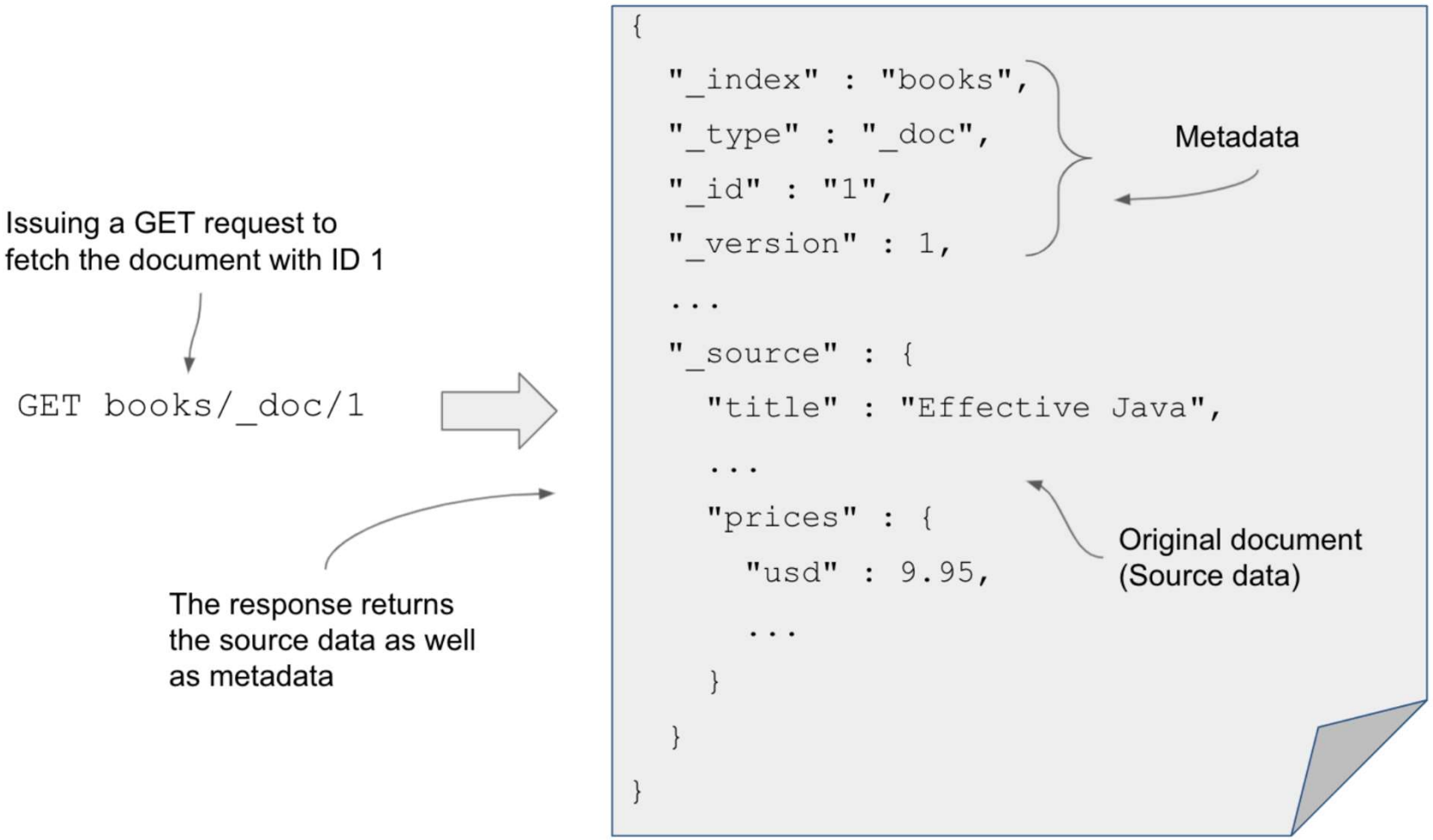
```
PUT books/_doc/2
{
  "title":"Core Java Volume I -
Fundamentals",
  "author":"Cay S. Horstmann",
  "release_date":"2018-08-27",
  "amazon_rating":4.8,
  "best_seller":true,
  "prices": {
    "usd":19.95,
    "gbp":17.95,
    "eur":18.95
  }
}
```

Indexing a document with ID 2

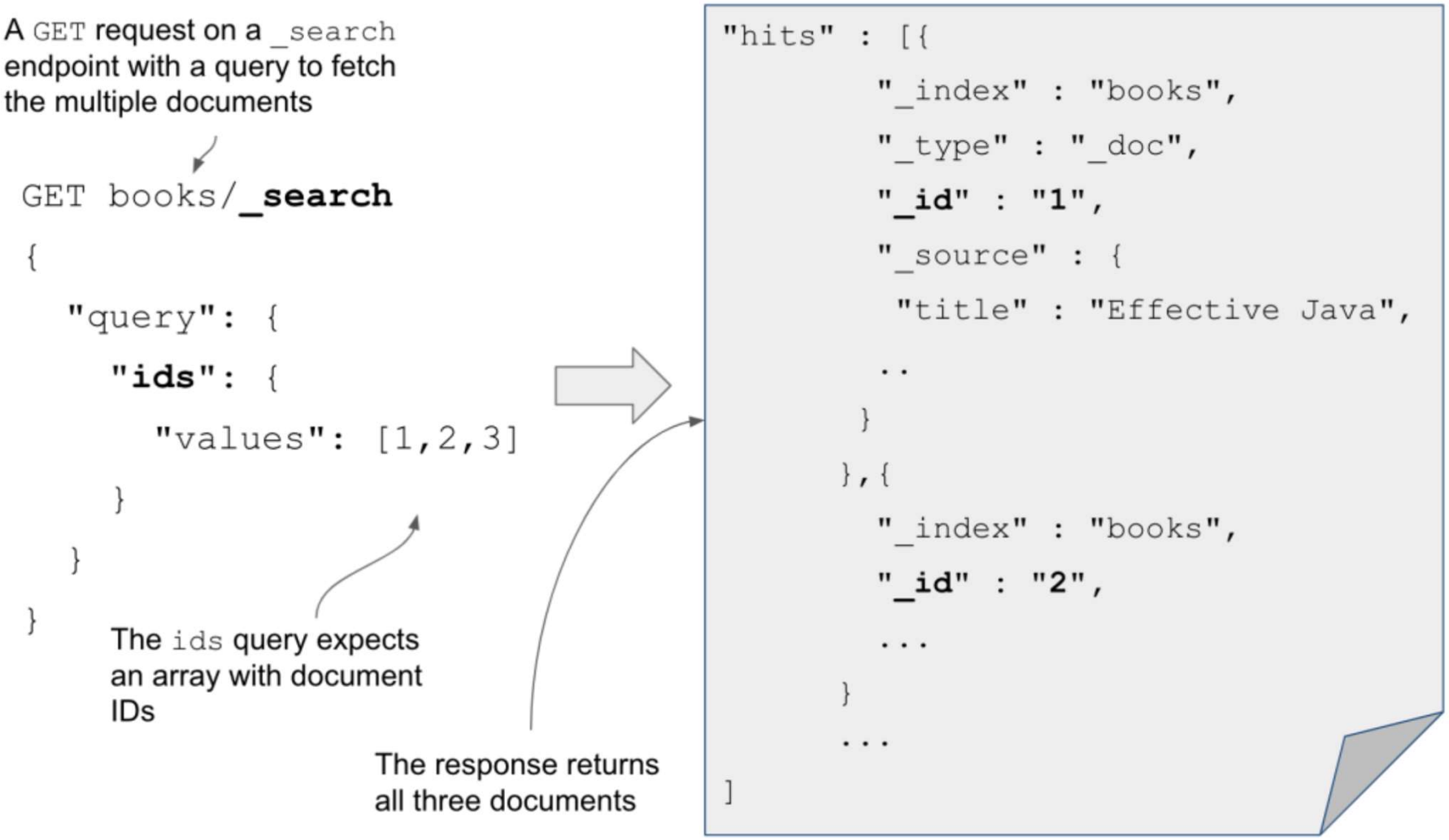
```
PUT books/_doc/3
{
  "title":"Java: A Beginner's
Guide",
  "author":"Herbert Schildt",
  "release_date":"2018-11-20",
  "amazon_rating":4.2,
  "best_seller":true,
  "prices": {
    "usd":19.99,
    "gbp":19.99,
    "eur":19.99
  }
}
```

Indexing a document with ID 3

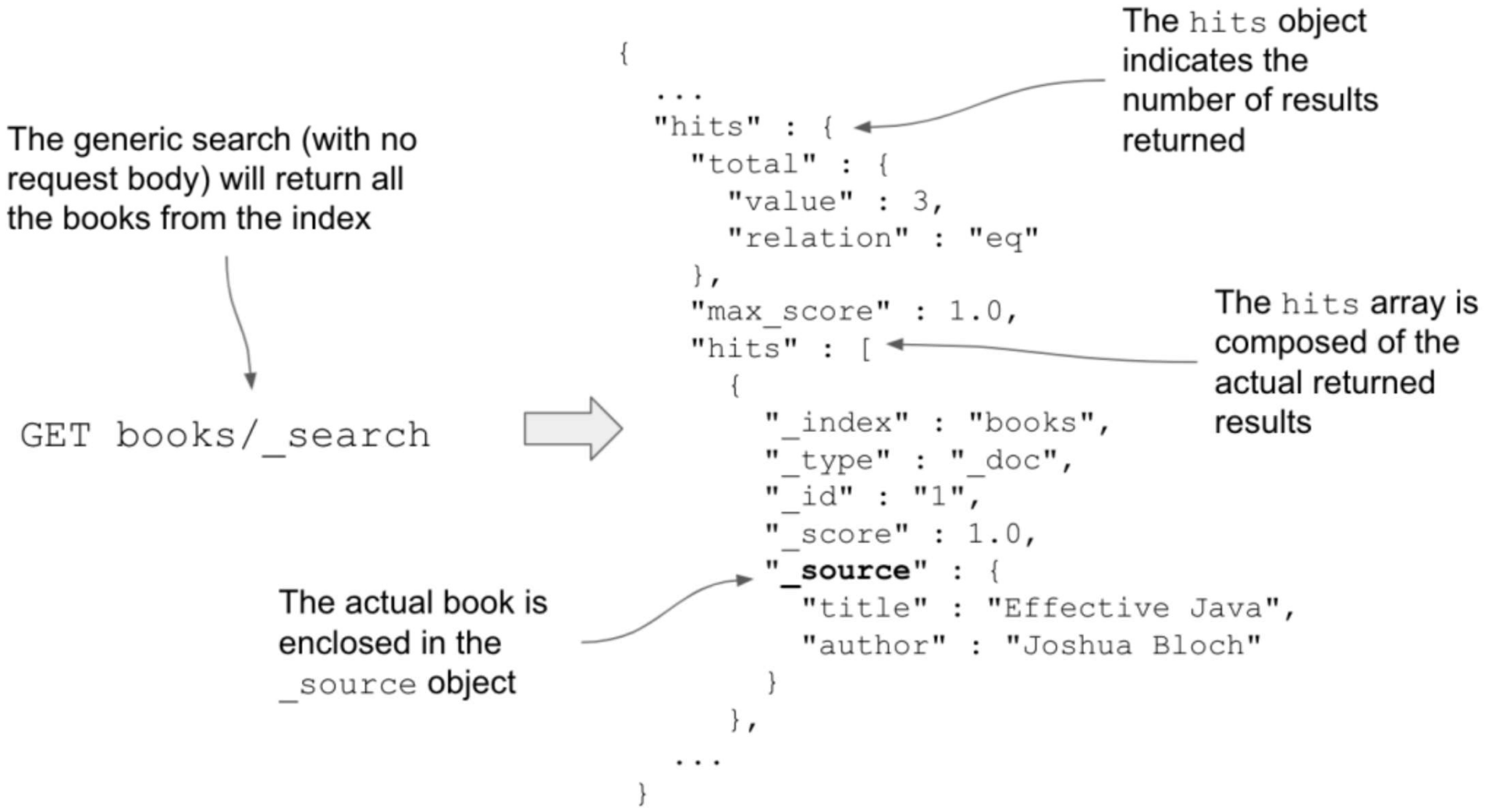
Fetching a book document by an ID



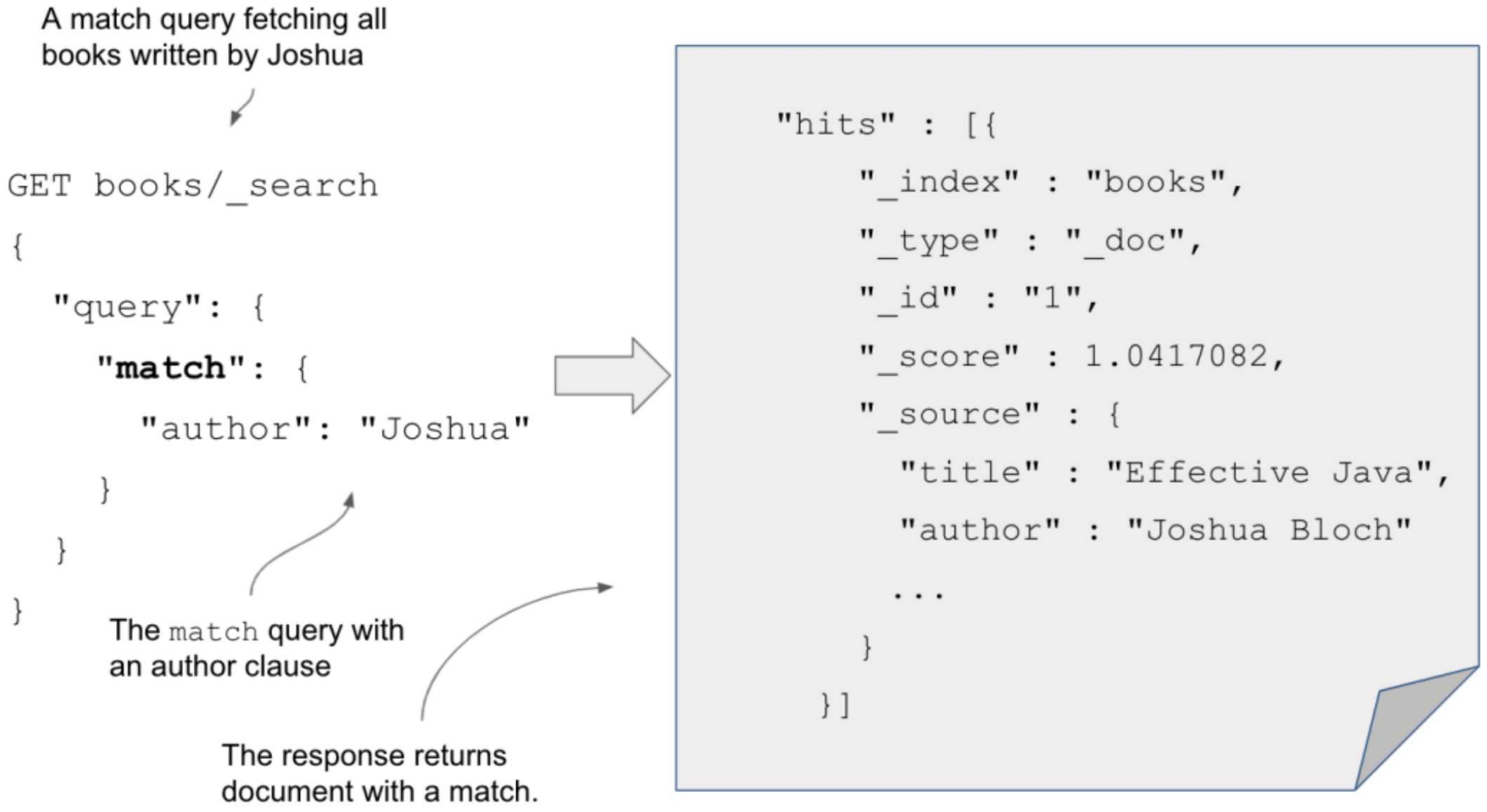
Retrieving documents given a set of IDs using an ids



Retrieving all documents using the search API



Fetching books authored by Joshua

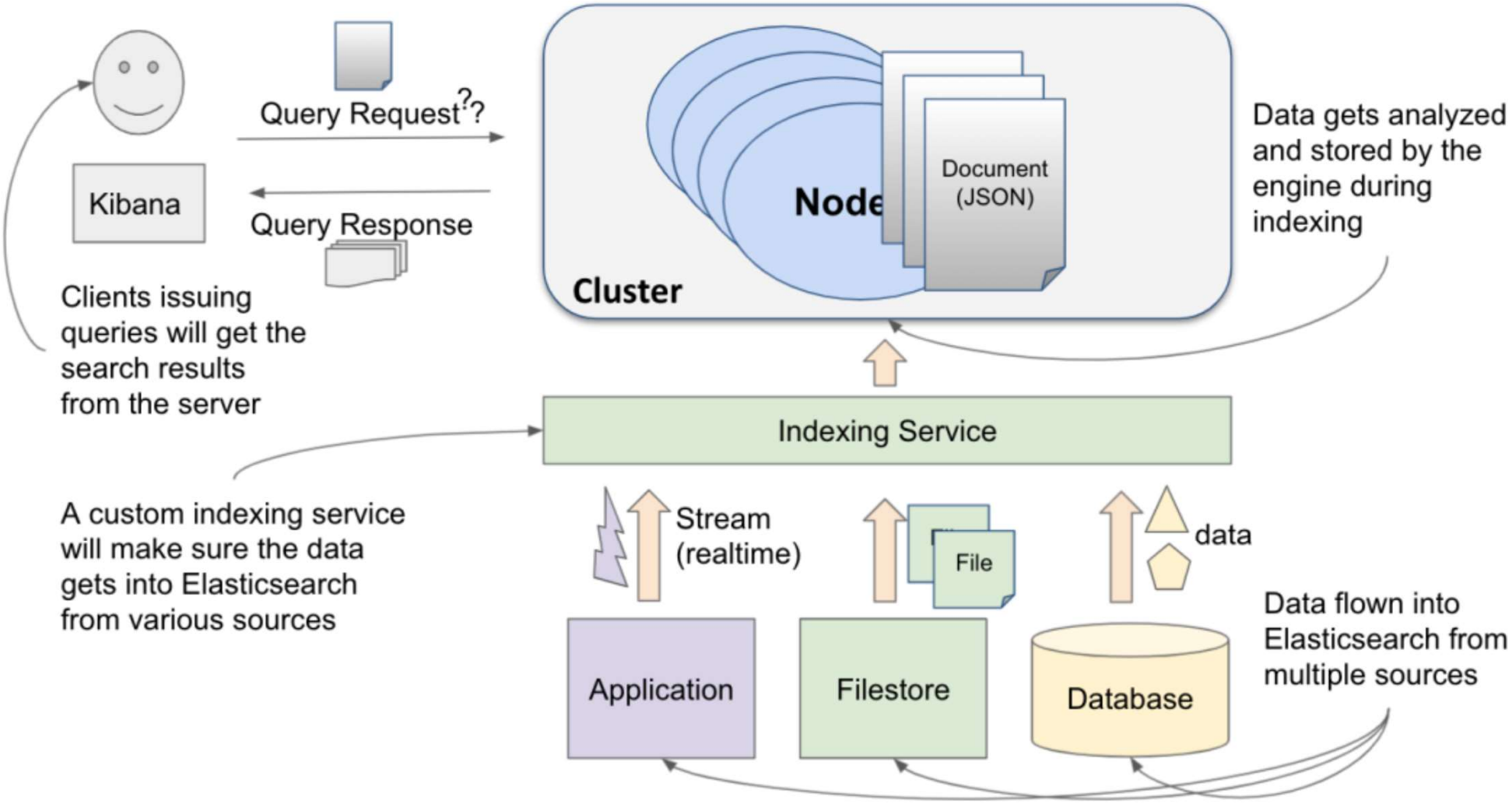


Fetching an exact match for a title using the and operator

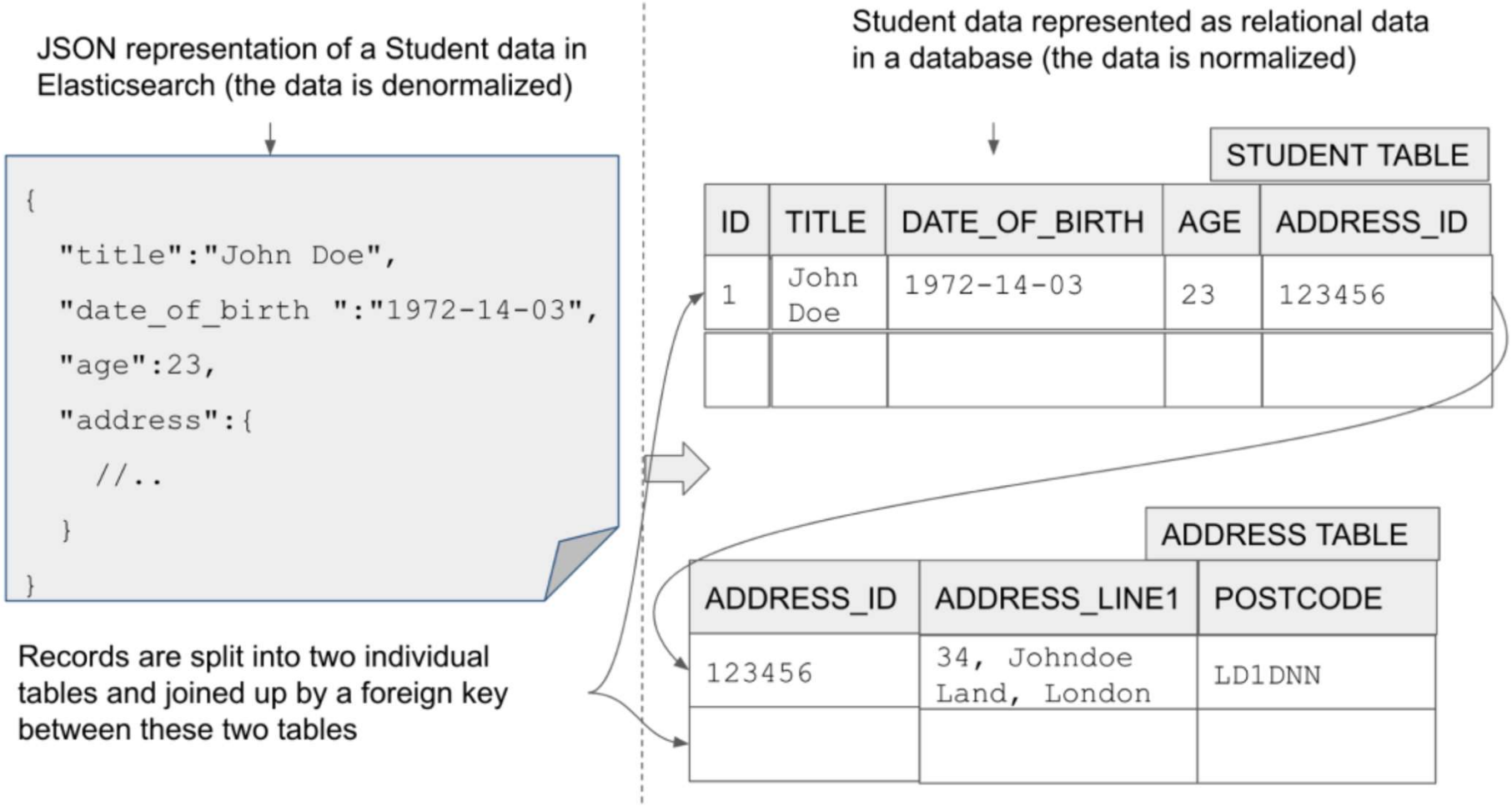
A match query fetching a specific title. The `and` operator searches for a title with both words in the title (“Effective and Java”)

```
GET books/_search
{
  "query": {
    "match": {
      "title": {
        "query": "Effective Java",
        "operator": "and"
      }
    }
  }
}
```

Elasticsearch with data



A JSON document vs. a relational database table structure



Removing document types from Elasticsearch

The url consists of the type (car) of the document too (which is deprecated and be removed in version 8.0)

```
PUT cars/car/1
{
  "make":"Toyota",
  "model":"Avensis"
}
```

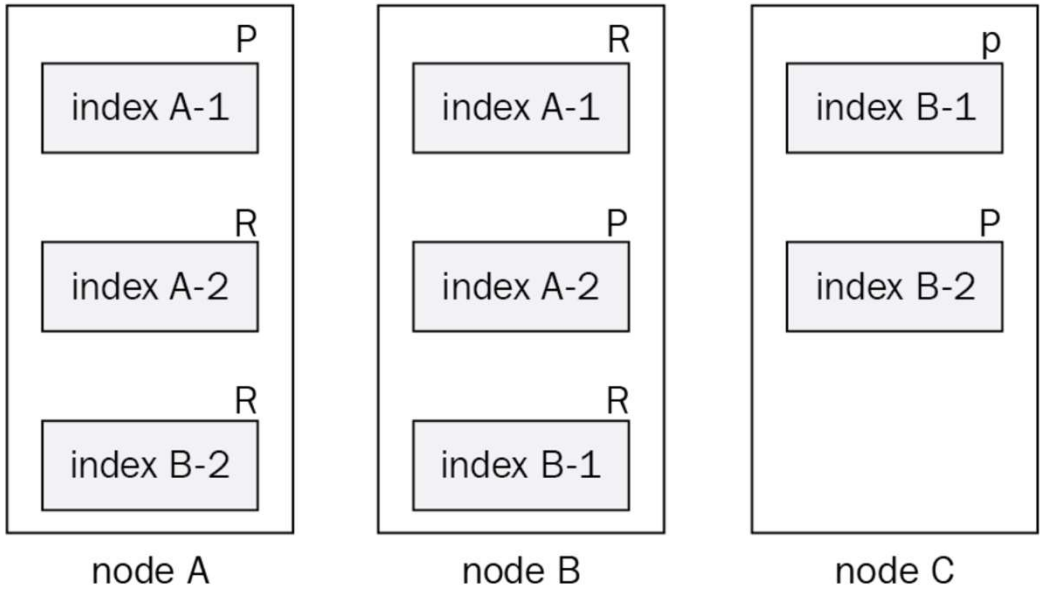
> V7.0: The explicit type is replaced with an endpoint named `_doc` (not document type)

```
PUT cars/_doc/1
{
  ...
}
```

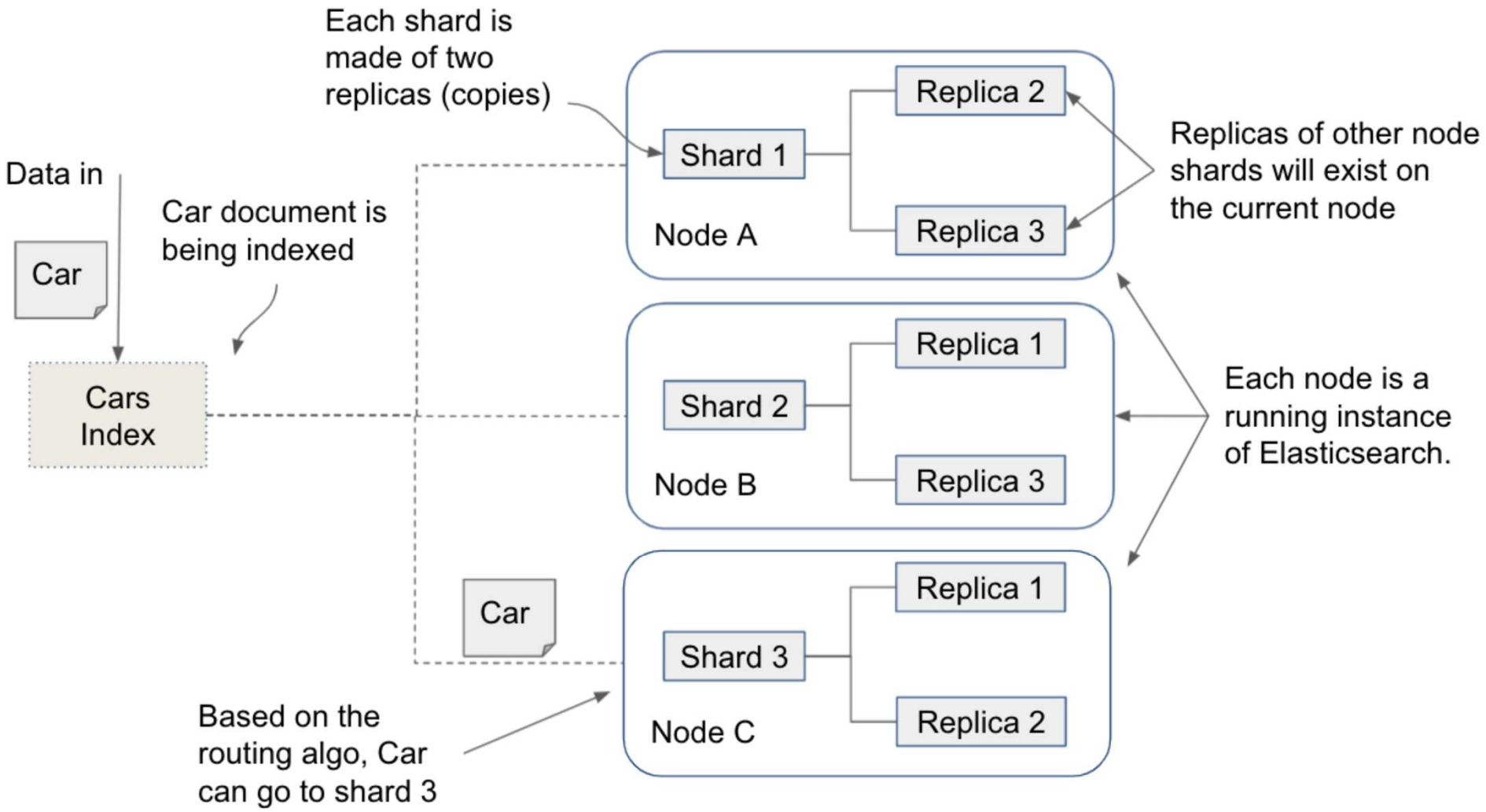
```
#! [types removal] Specifying types in
document index requests is deprecated, use the
typeless endpoints instead
(/{index}/_doc/{id}, /{index}/_doc, or
/{index}/_create/{id}).

{
  "_index" : "cars",
  "_type" : "car",
  "_id" : "1",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 2,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}
```

While using the type upto 7.x is allowed (you will receive a warning as shown here), it is advisable to drop the type completely

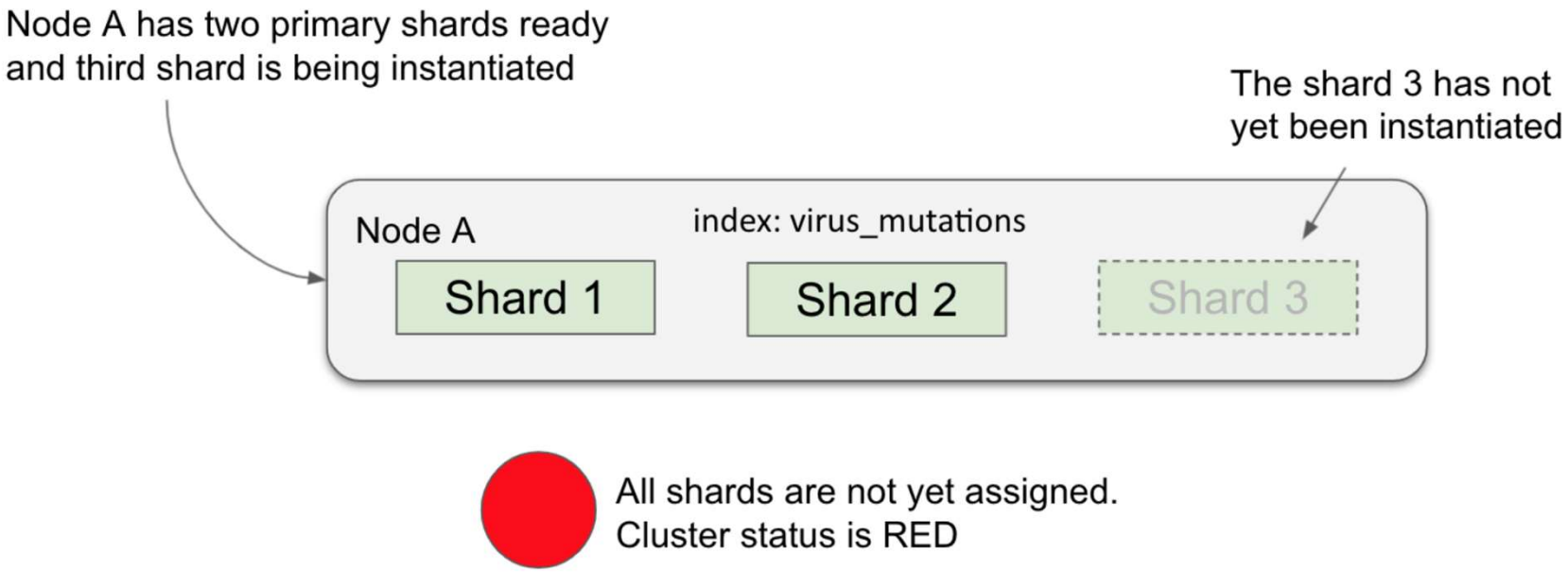
Shards of data distributed across nodes



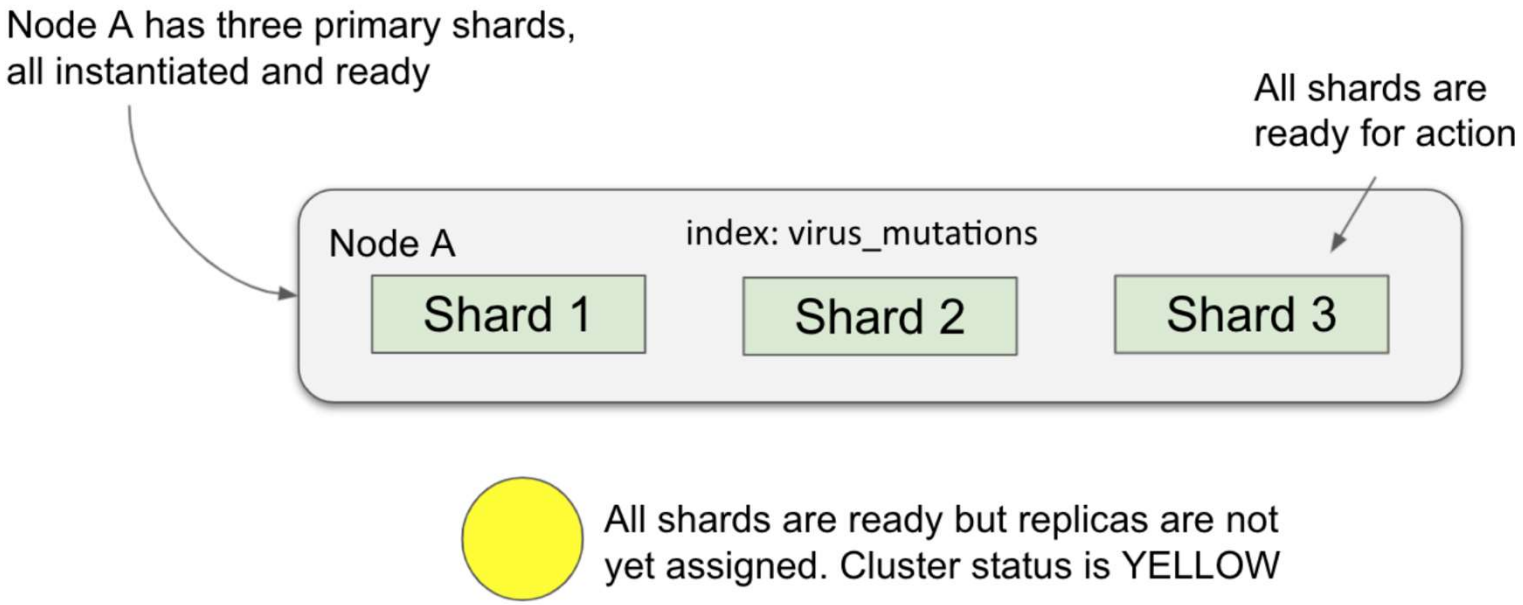
An index designed with three shards and three replicas



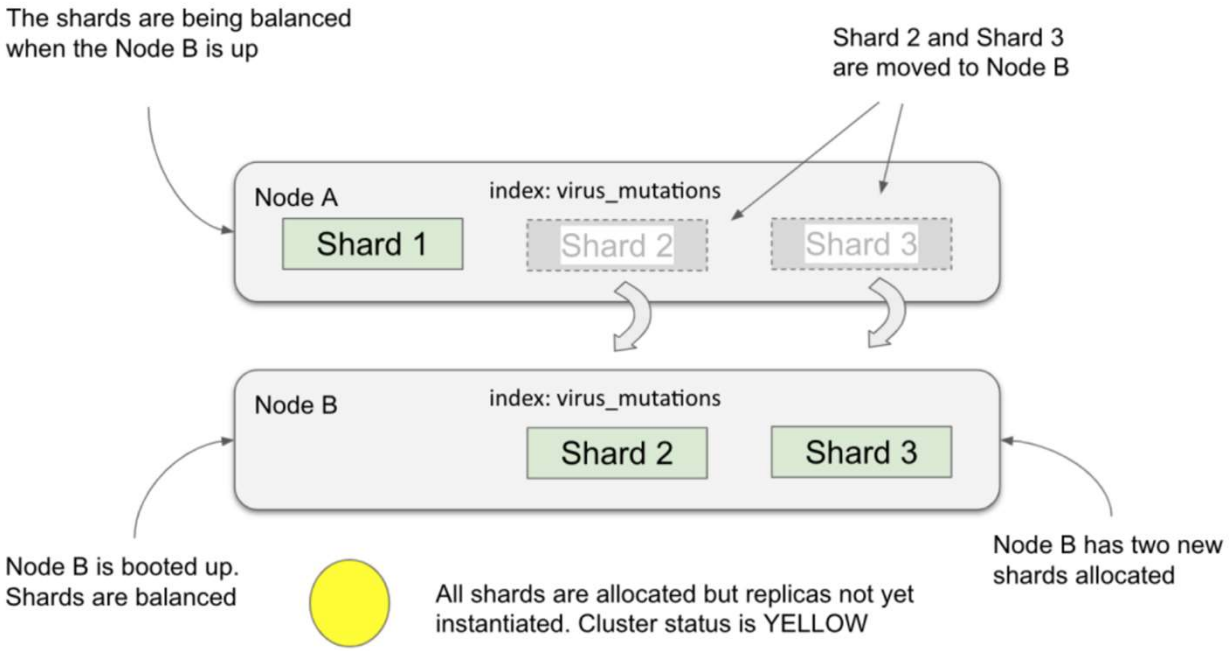
Engine is not ready



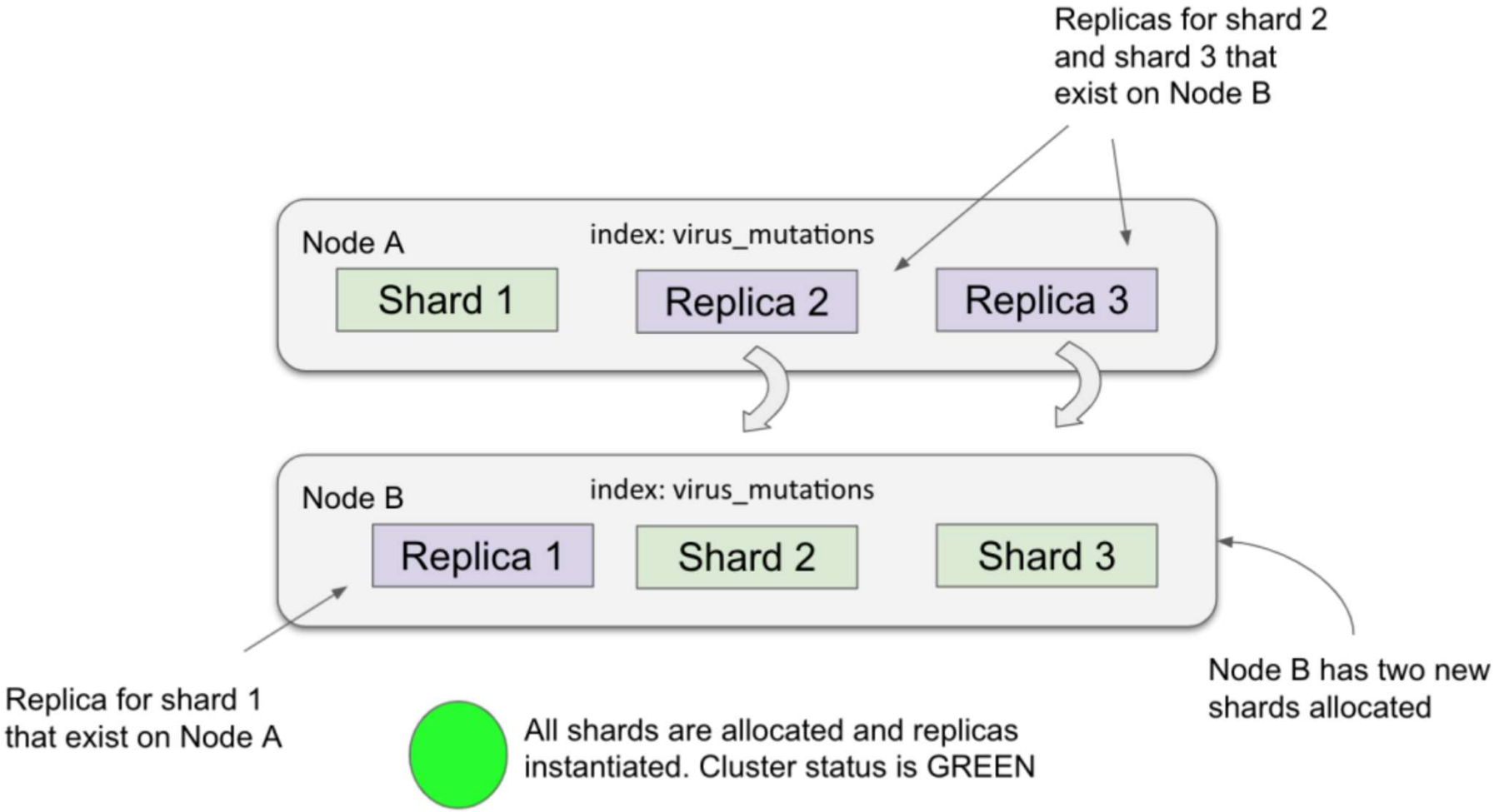
Single node with three shards joining a single-node cluster



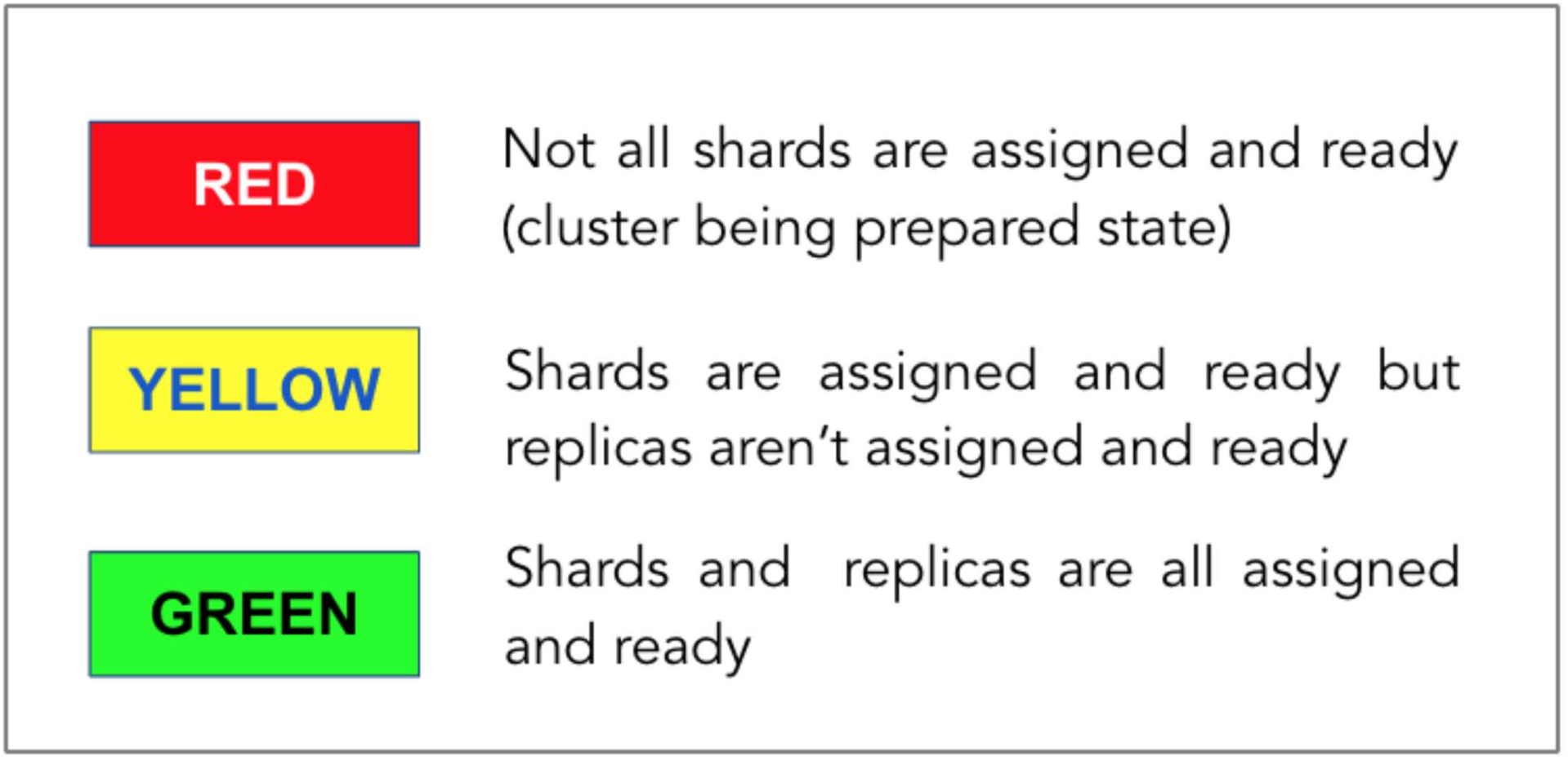
Shards were balanced on to the new node but replicas aren’t assigned yet (yellow status).



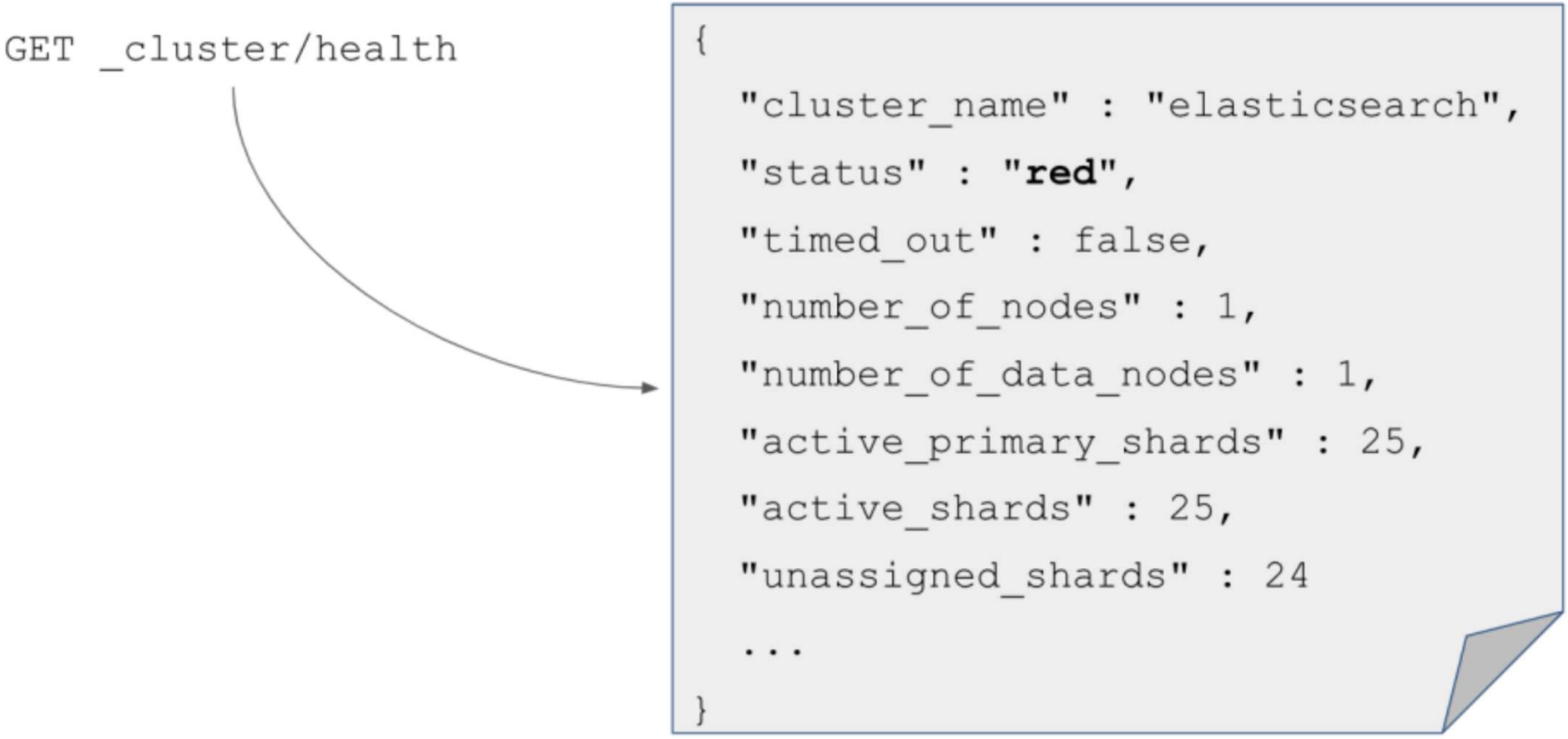
All shards and replicas are allocated



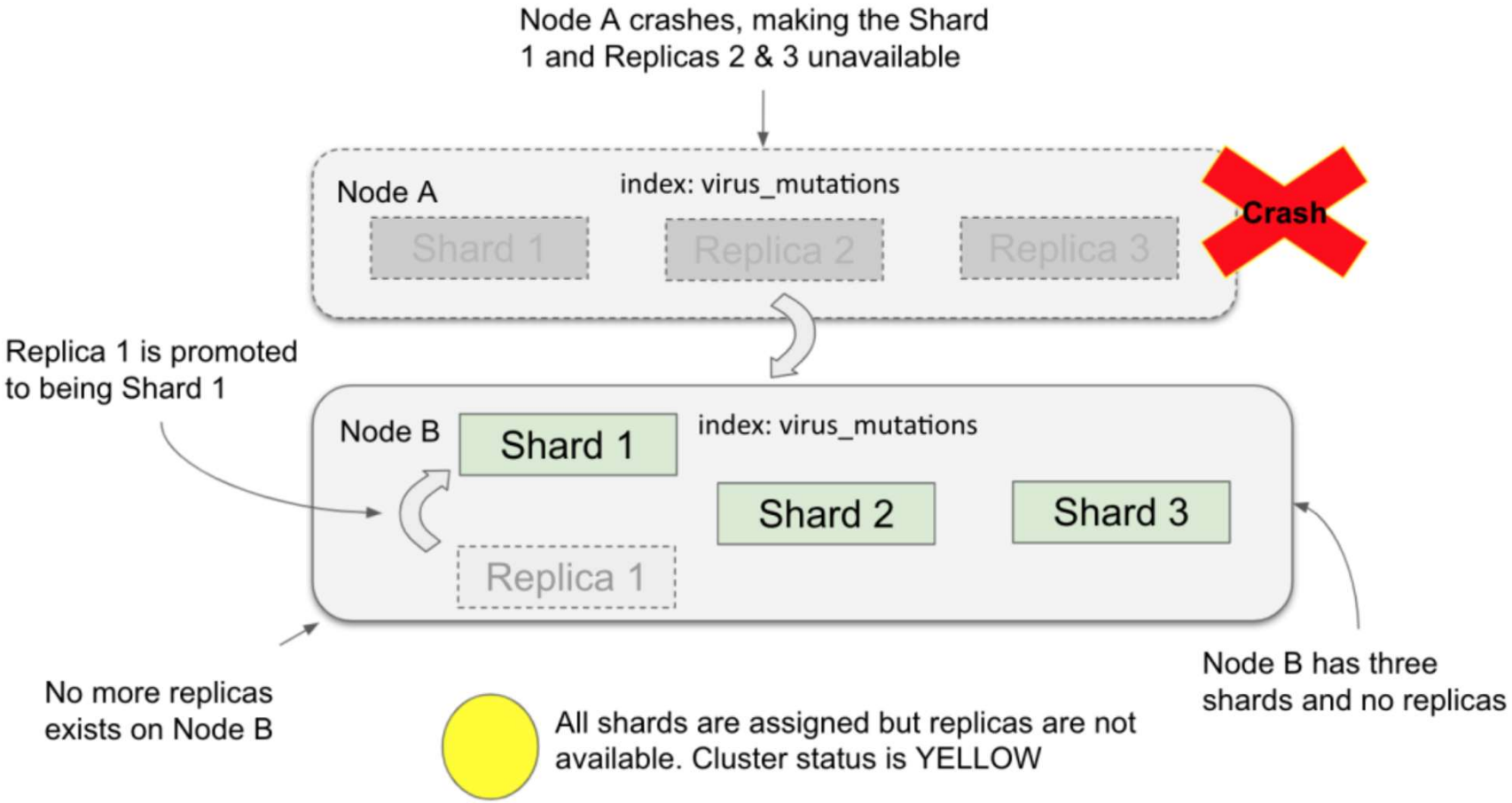
Health of shards using a traffic light signal board



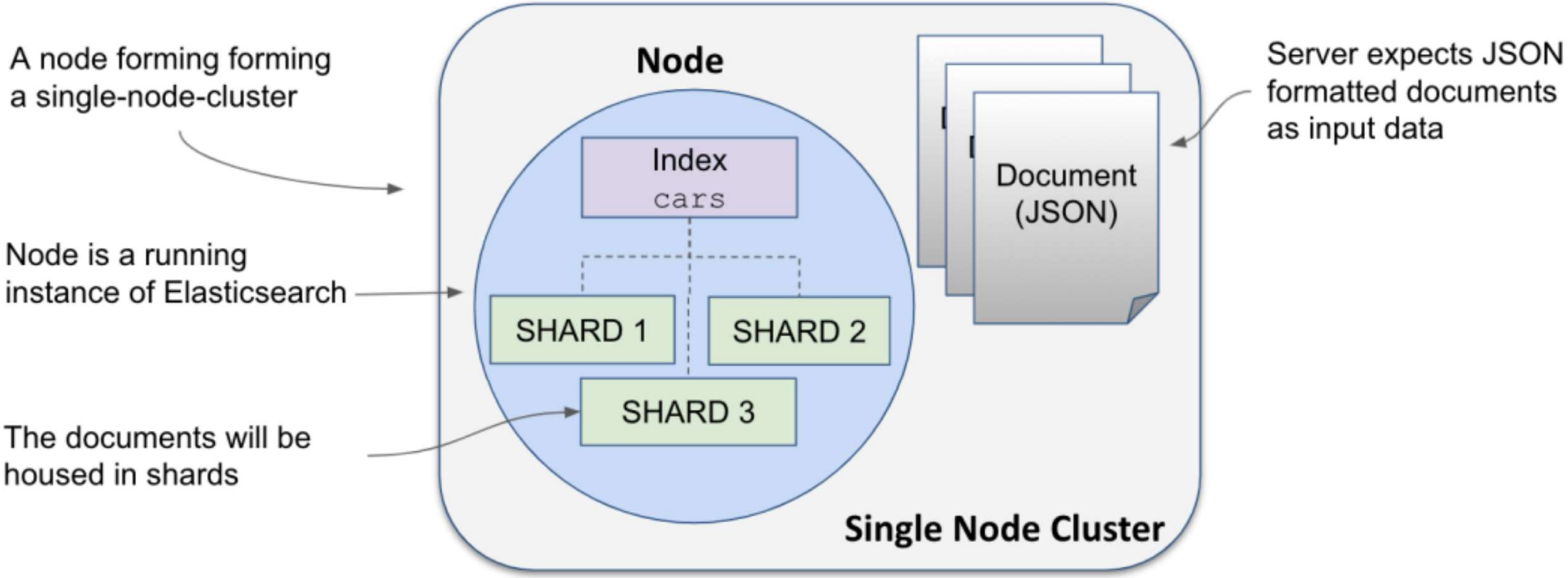
Fetching the status of a cluster



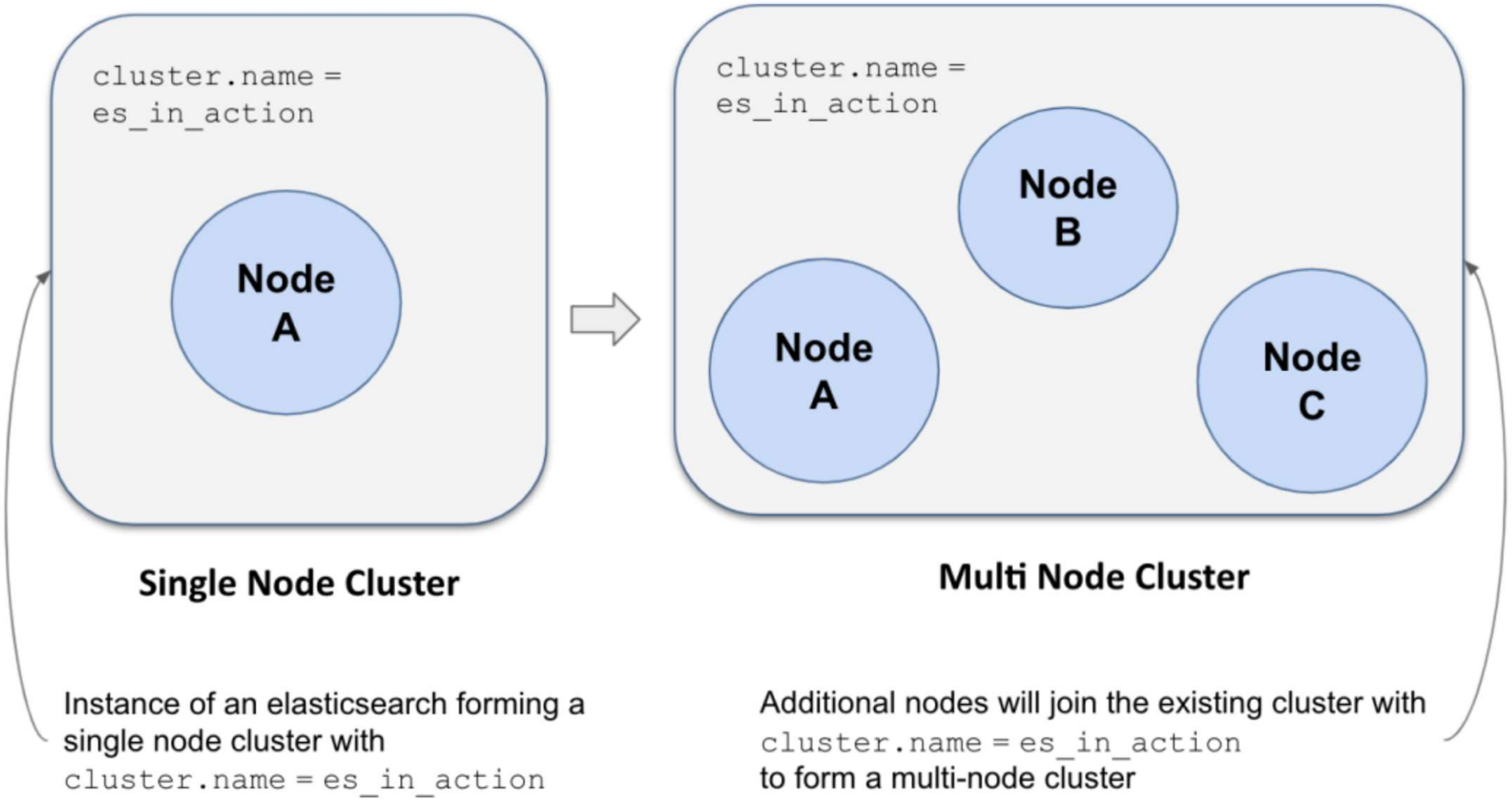
Replicas were lost (or promoted to a shard) when a node crashed.



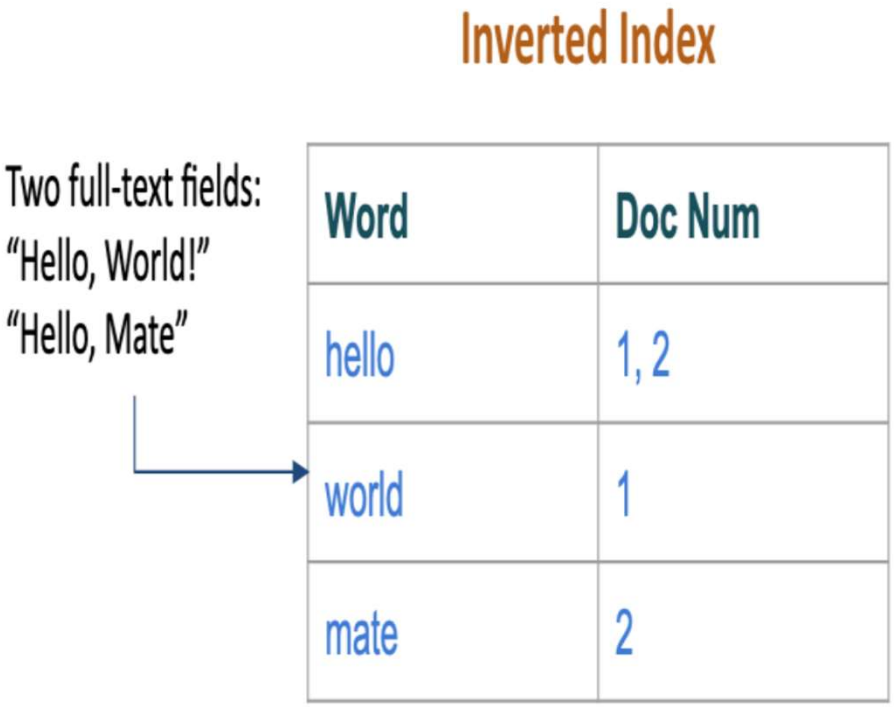
A single node Elasticsearch cluster



Cluster formation from single node to multi-node cluster

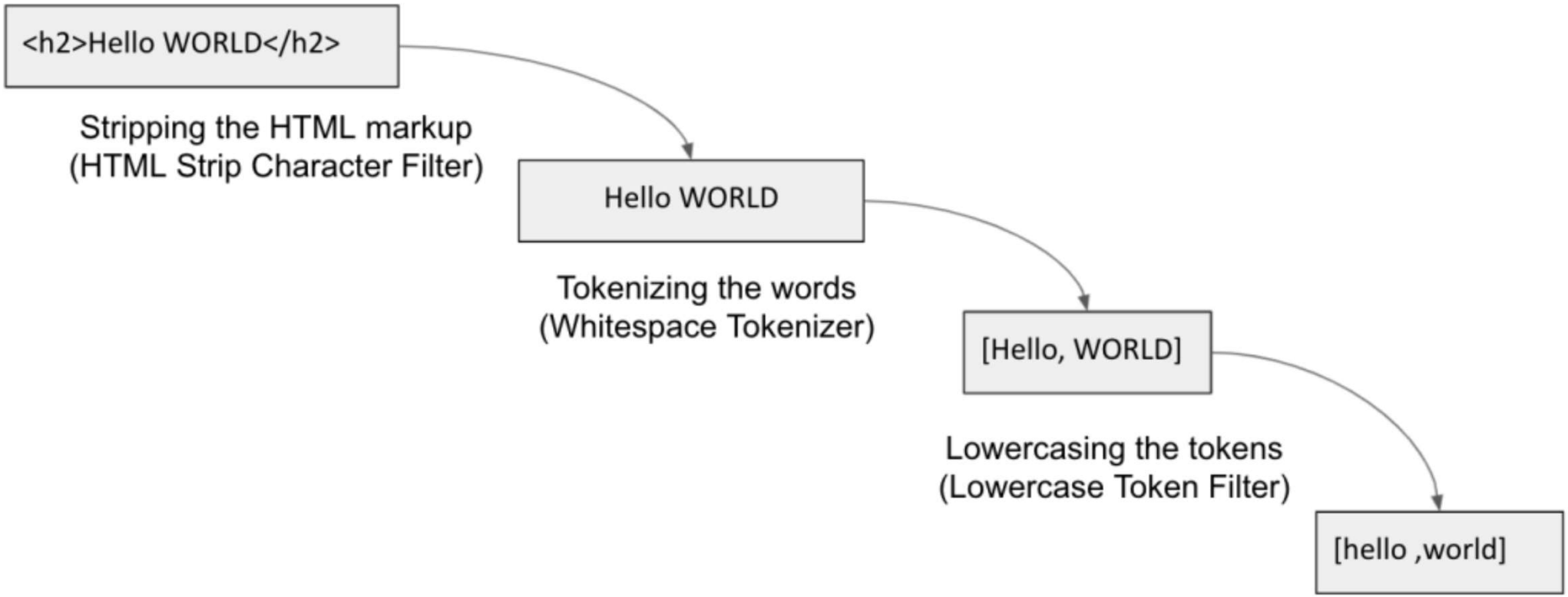


An inverted index data structure



- This inverted index is the key to faster retrieval of documents during the full-text search phase
- For each document that consists of full-text fields, the server creates the respective inverted indices.

Text analysis procedure



The tokenized words for “Hello, World” and the documents they are found in

Word	Frequency	Document ID
hello	1	1
world	1	1

The tokenized words for “Hello, Mate” and the documents they are found in

Word	Frequency	Document ID
hello	2	1,2
world	1	1
mate	1	2

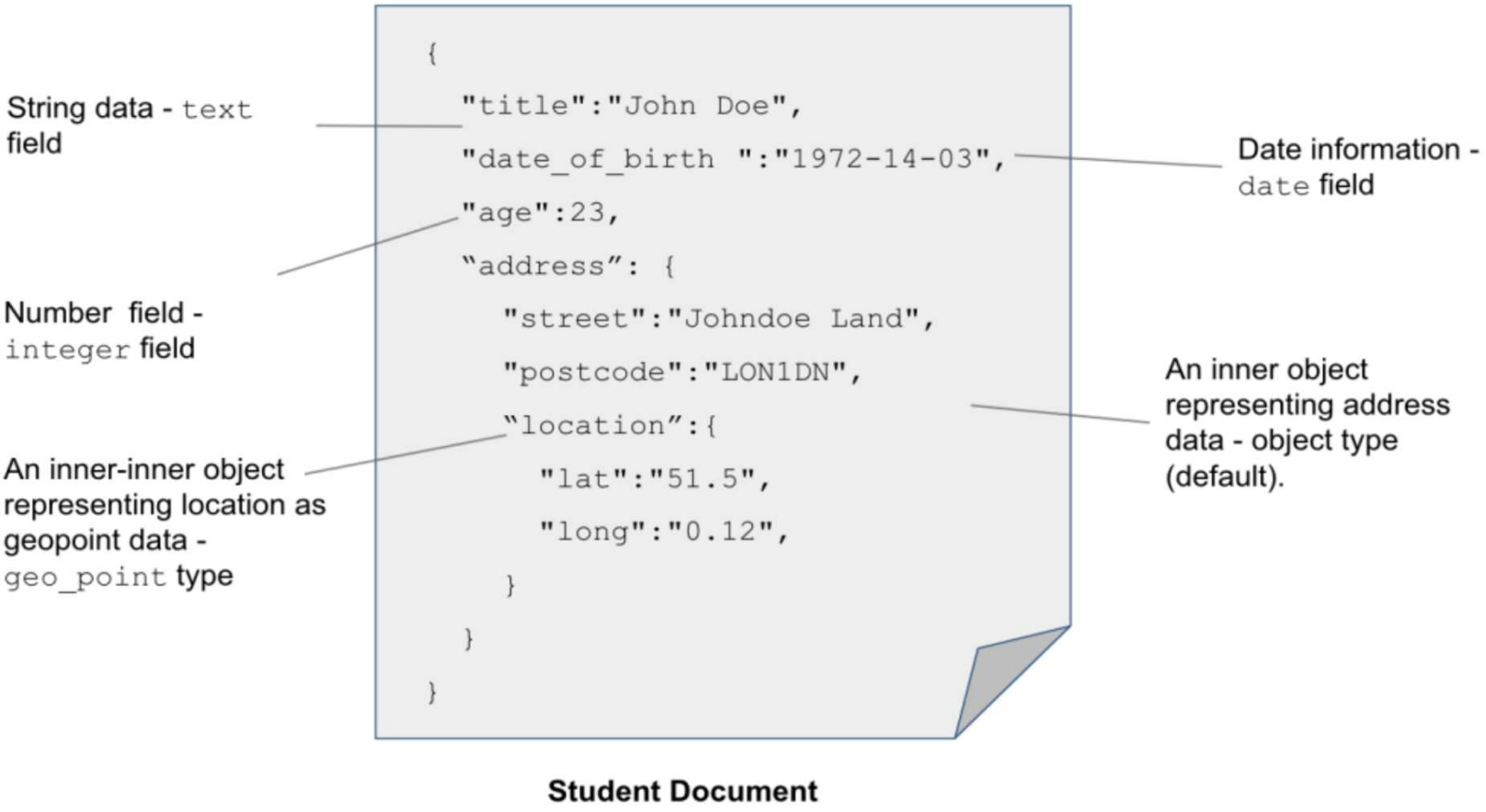
Relevant results for Java in a title search

```
GET books/_search
{
  "_source": "title",
  "query": {
    "match": {
      "title": "Java"
    }
  }
}
```

```
"hits" : {
     …
    "max_score" : 0.33537668,

    "hits" : [{
        "_score" : 0.33537668,
        "_source" : {  "title" : "Effective
Java" }
      },
      {
        "_score" : 0.30060259,
        "_source" : { "title" : "Head First
Java" }
      },
      {
        "_score" : 0.18531466,
        "_source" : { "title" : "Test-Driven:
TDD and Acceptance TDD for Java Developers"}
      }…]
    }
```

Mapping

An example of a student document with the data represented in JSON



Student Document

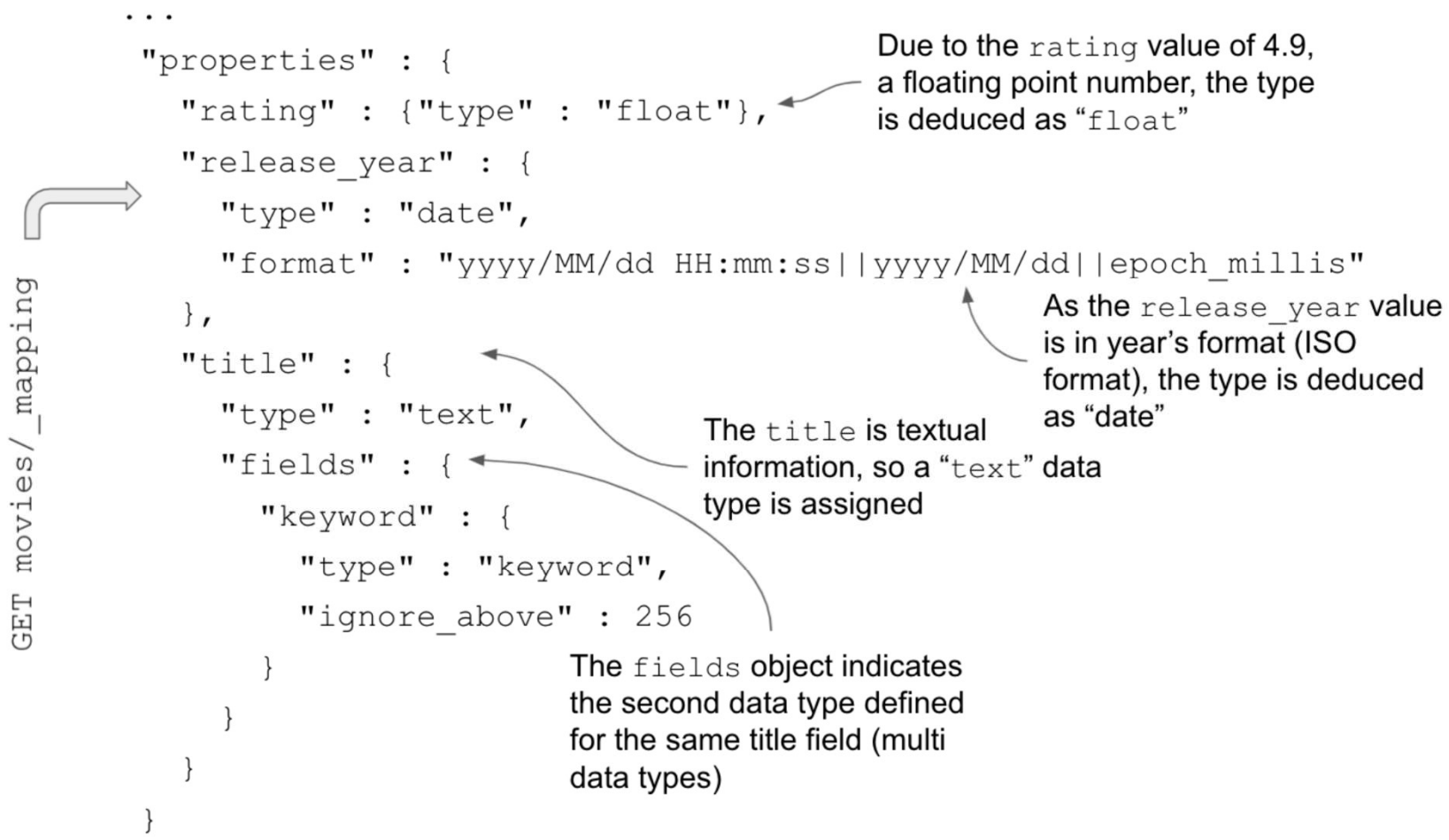
Indexing a document for the first time

- PUT movies/_doc/1
- {
- "title":"Godfather",#A The title of the movie
- "rating":4.9,#B The rating given to the movie
- "release_year":"1972/08/01" #C Movie’s release year(note the date format)
- }

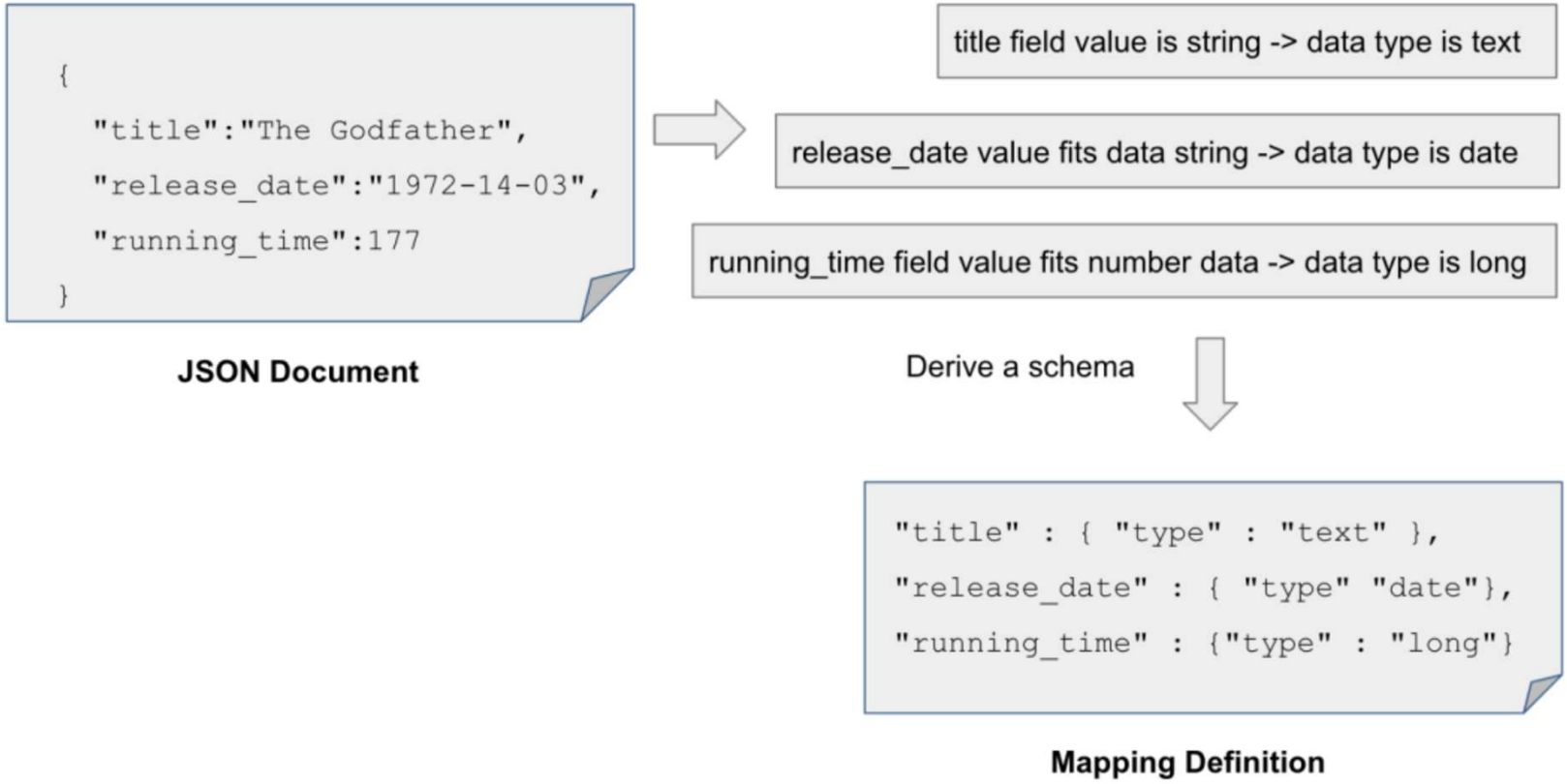
What happens when this document hits the engine?

- A new index (movies) is created automatically with default settings.
- A new schema is created for the movies index with the data types deduced from this document's fields
- The document gets indexed and stored in the Elasticsearch data store
- Subsequent documents get indexed without undergoing the previous steps as Elasticsearch consults the newly created schema for further indexing

Movie index mapping derived from the document values



Dynamically deriving the indexing schema



Creating and updating schema using indexing and mapping APIs

Creating a `movies` index with initial set of fields and their types (using Indexing API)

```
PUT movies
{
 "mappings": {
    "properties": {
      "title":{
        "type": "text"
      }
    }
  }
}
```

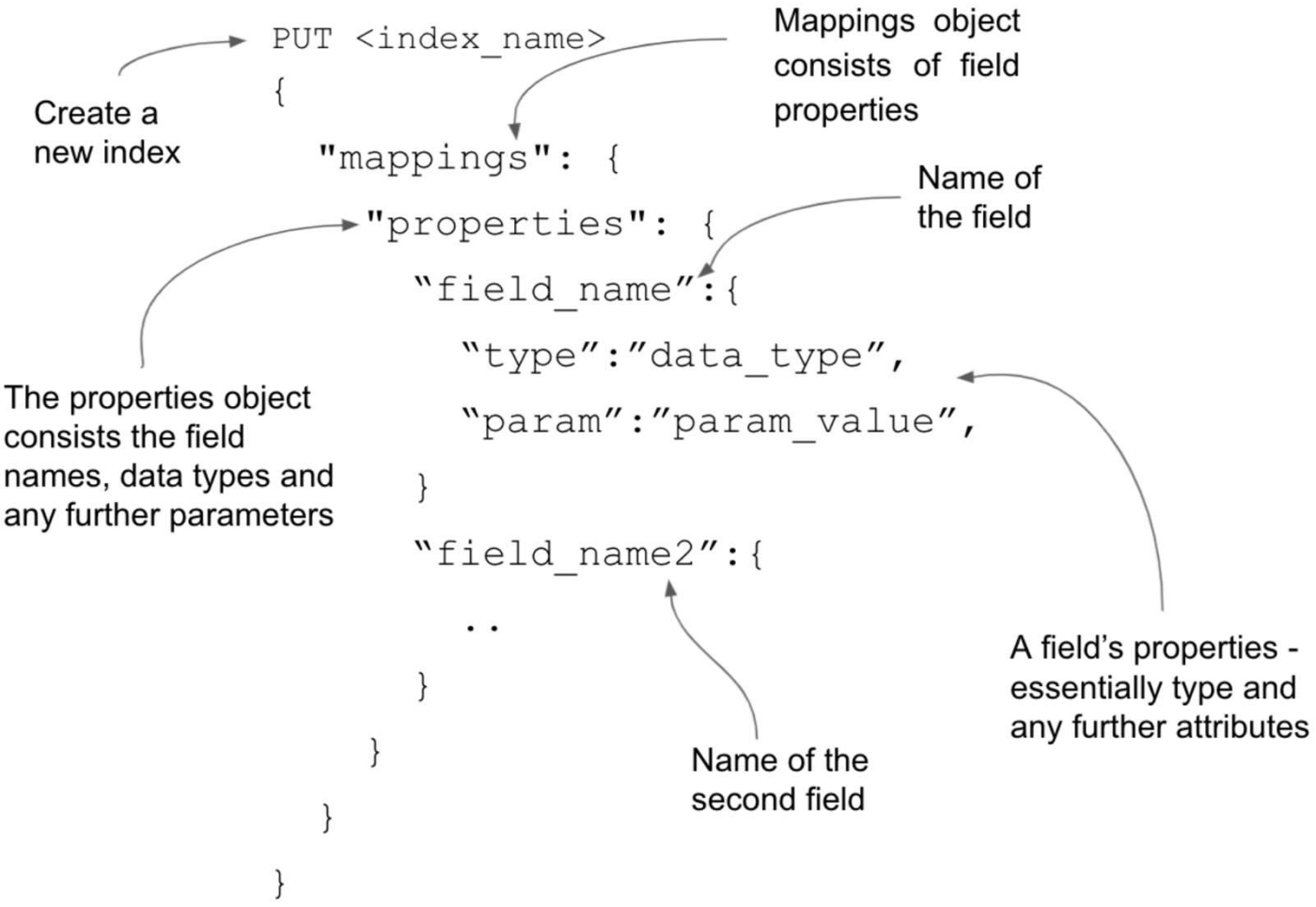
Updating the `movies` index with additional set of fields and their types (using Mapping API)

```
PUT movies/_mapping
{
  "properties":{
    "release_date":{
      "type":"date",
      "format":"dd-mm-yyyy"
    }
  }
}
```

title	text
release_date	date

Final movies index consisting of fields added during the indexing time and then updated later

Creating a mapping definition during index creation



Common data types with examples

Type	Description	Examples
`text`	Represents textual information (such as string values); unstructured text	Movie title, blog post, book review, log message
`integer/long/short/byte`	Represents a number	Number of infectious cases, flights canceled, products sold
`float/double`	Represents a floating-point number	Student's grade point, moving average of sales, reviewer ratings
`boolean`	Represents a binary choice, true or false	Is the movie a blockbuster? Are COVID cases on the rise? Has the student passed the exam?
`keyword`	Represents structured text, text that must not be broken down or analyzed	Error codes, email addresses, phone numbers, Social Security numbers
`object`	Represents a JSON object	(In JSON format) employee details, tweets, movie objects
`nested`	Represents an array of objects	An employee's address, email's routing data, a movie's technical crew

Creating the mapping definition (left) and retrieving the schema (right)

```
PUT students
{
  "mappings": {
    "properties": {
      "name":{
        "type": "text"
      },
      "age":{
        "type": "byte"
      }
    }
  }
}
```

The `students` index is defined with a schema consisting of name (`text`) and age (`byte`)

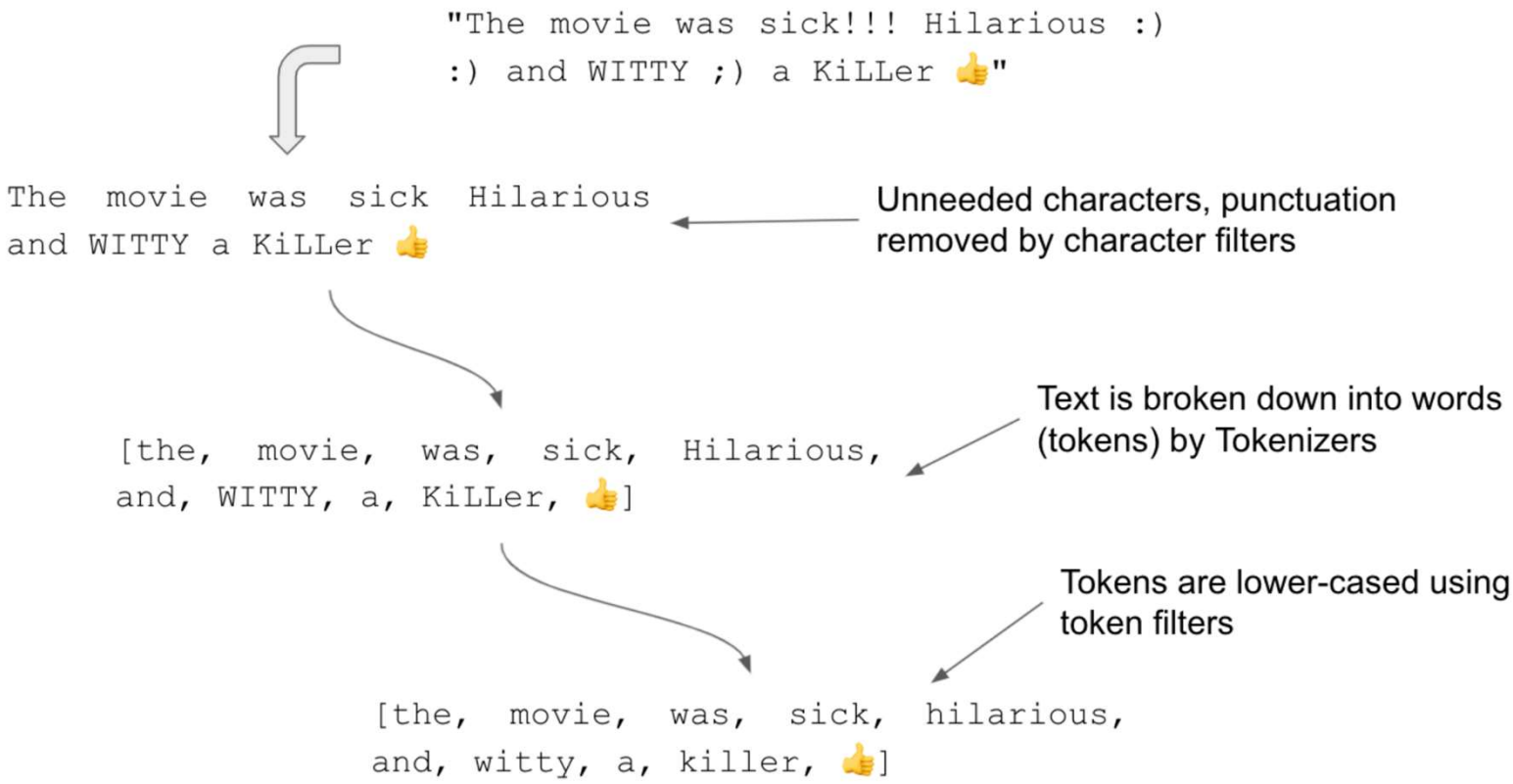
```
GET students/_mapping
```

Fetching the mappings for students index

```
{
  "students" : {
    "mappings" : {
      "properties" : {
        "age" : {
          "type" : "byte"
        },
        "name" : {
          "type" : "text"
        }
      }
    }
  }
}
```

Students mapping definition

Processing a full text field during indexing by standard analyzer module



Numeric data types

Integer types	`byte`	Signed 8 bit integer
	`short`	Signed 16 bit integer
	`integer`	Singed 32 bit integer
	`long`	Signed 64 bit integer
	`unsigned_long`	64 bit unsigned integer
Floating Point types	`float`	32 bit single precision floating-point number
	`double`	64 bit double precision floating-point number
	`half_float`	16 bit half precision floating-point number
	`scaled_float`	Floating-point number backed by `long`

Working with documents

Indexing a document with an identifier

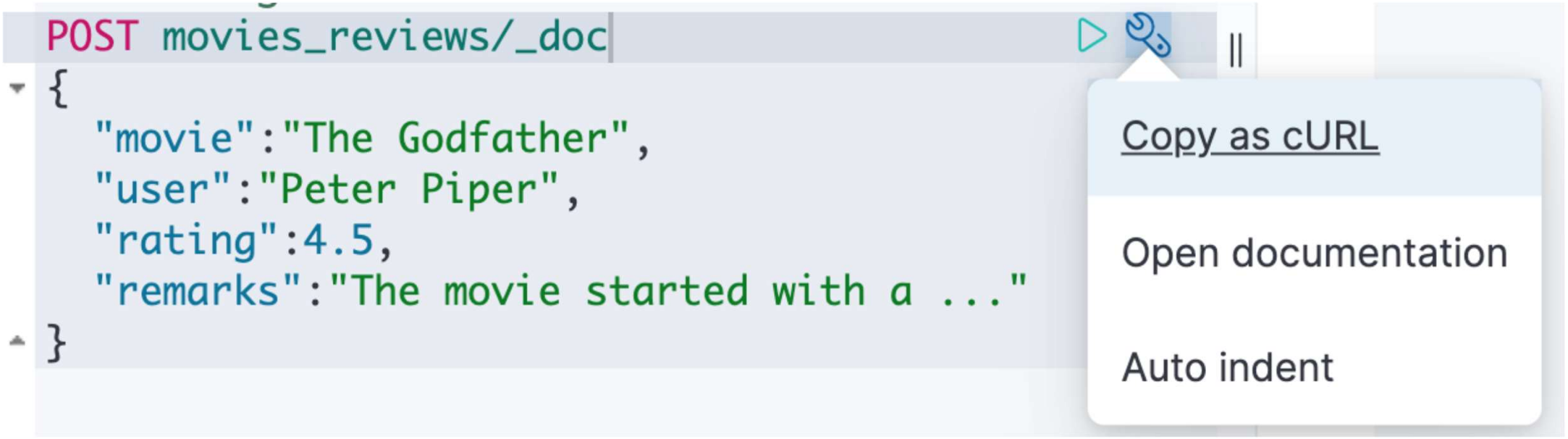
```
PUT movies/_doc/1
{
  "title":"The Godfather",
  "synopsis":"The aging patriarch
   of an organized crime ..”
}
```

A request to the server to index a movie document with an id as 1. The body of the request is a JSON document

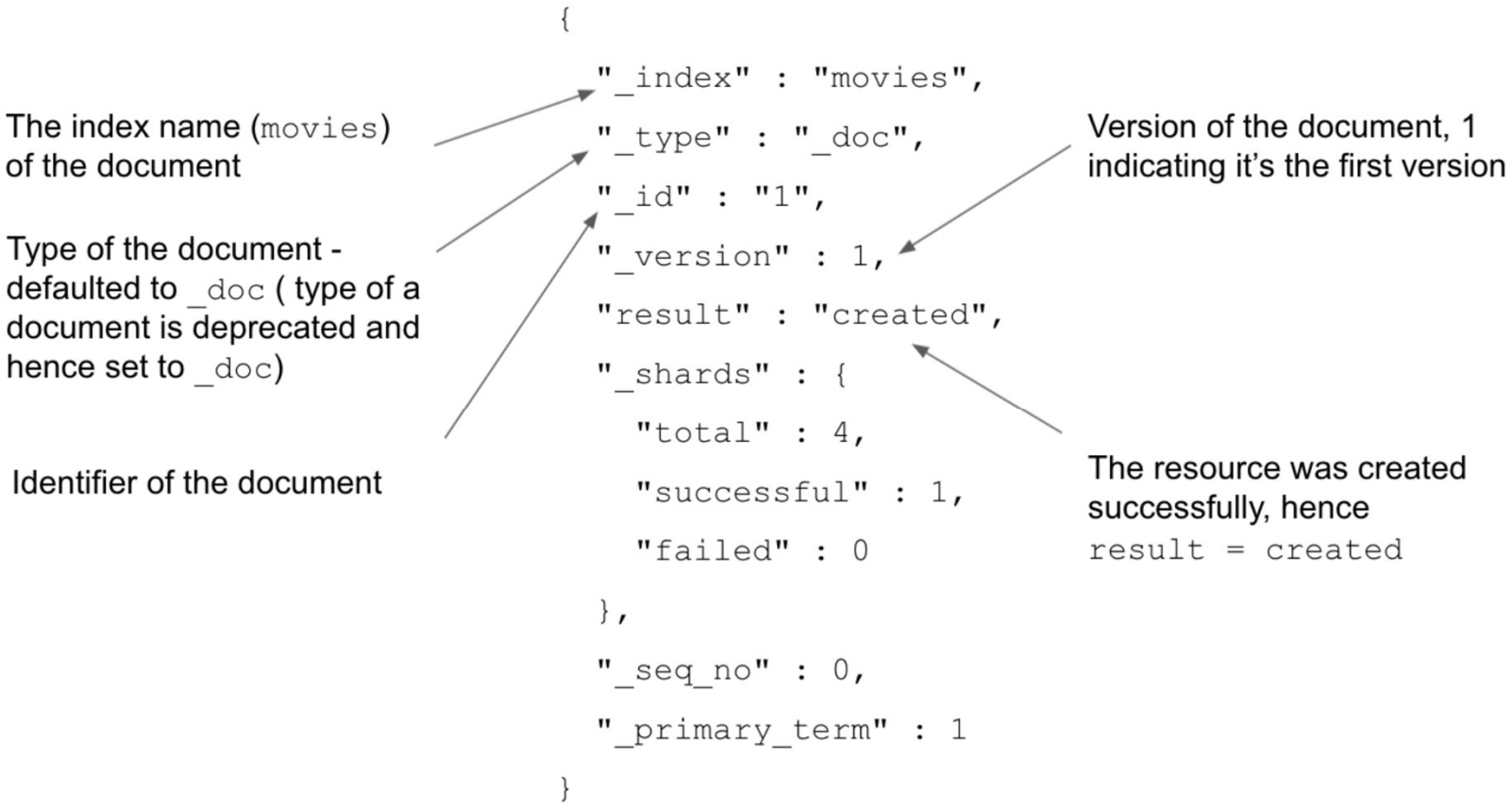
```
{
  "_index" : "movies",
  "_type" : "_doc",
  "_id" : "1",
  "_version" : 1,
  "result" : "created",
  "_shards" : {
    "total" : 4,
    "successful" : 1,
    "failed" : 0
  },
  "_seq_no" : 0,
  "_primary_term" : 1
}
```

Response from the server indicating that the movie document was indexed successfully

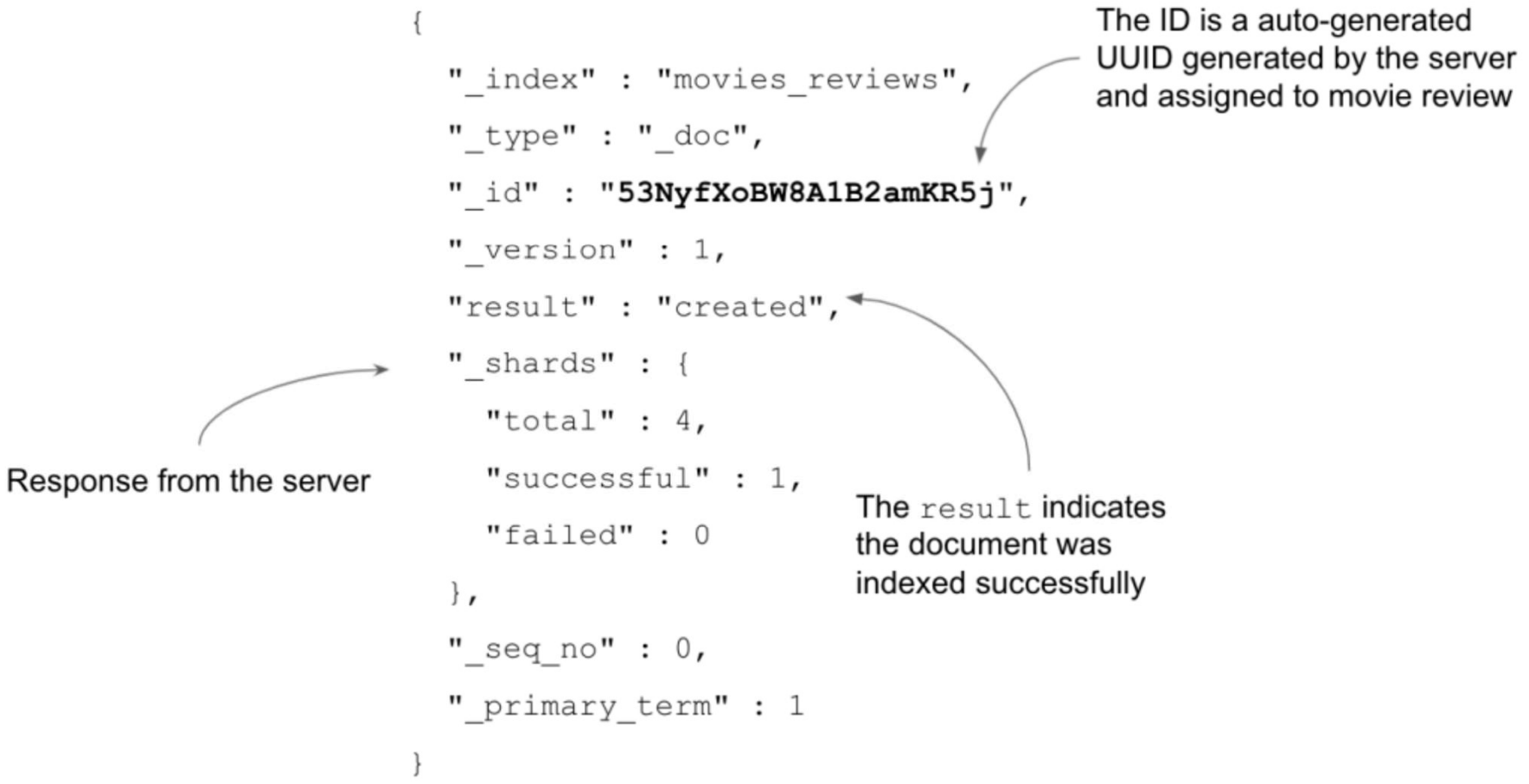
Exporting the query as cURL



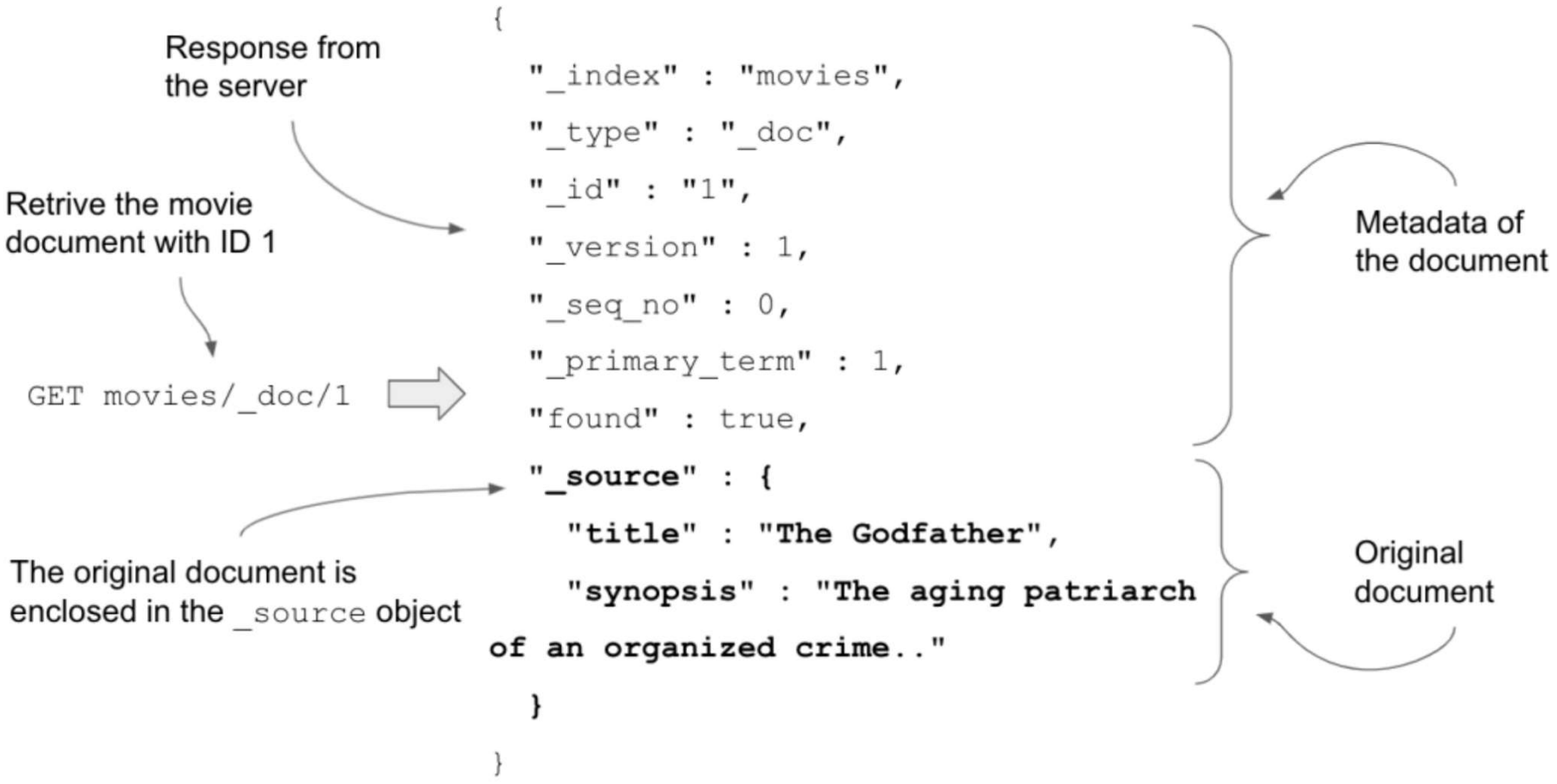
Response from the server when a document is indexed successfully



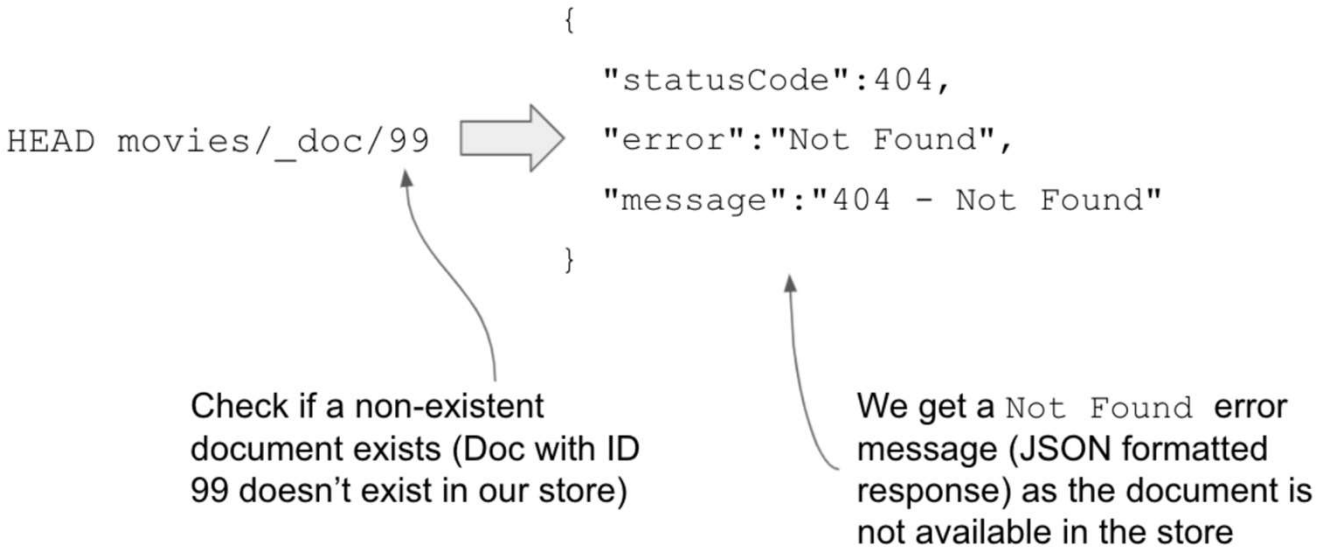
The server creates and assigns a randomly auto generated ID to the document



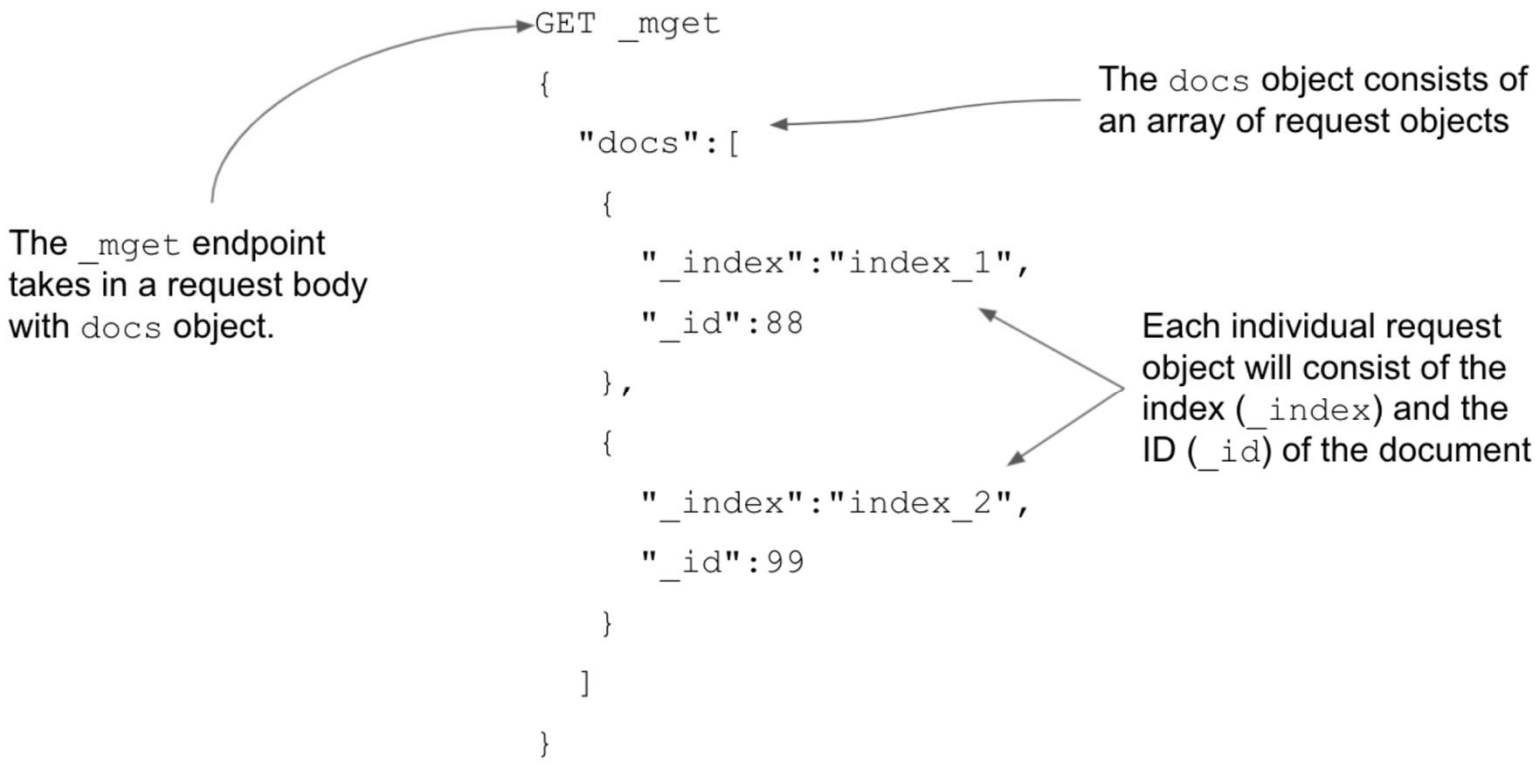
Retrieving a document using the GET API call



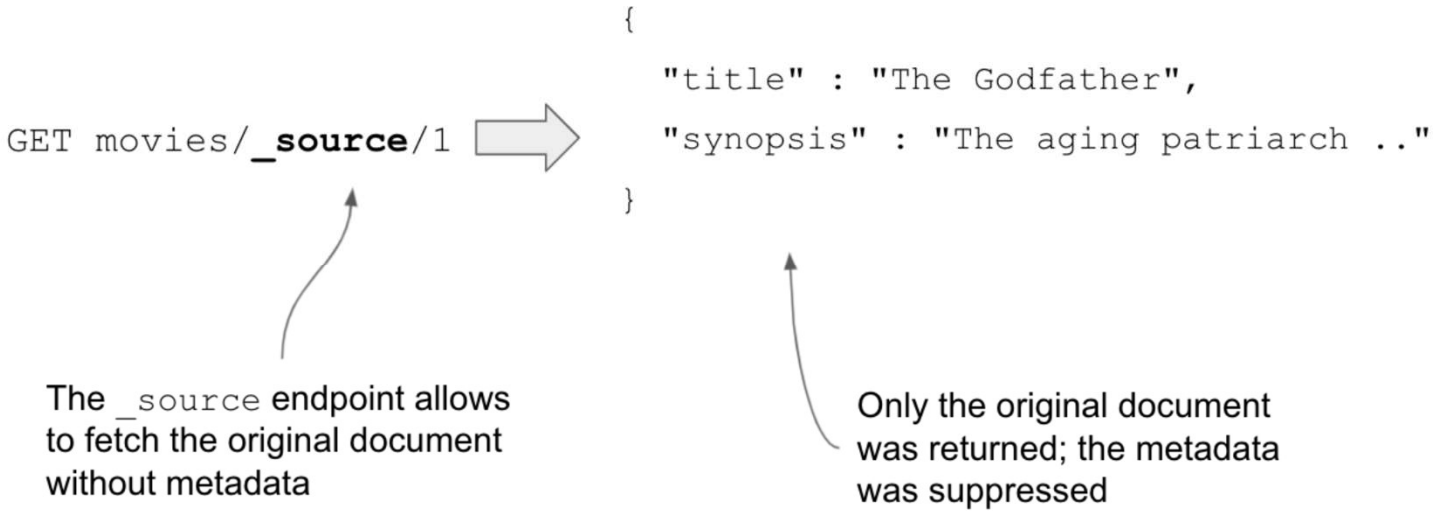
Fetching a non-existent document returns a Not Found message



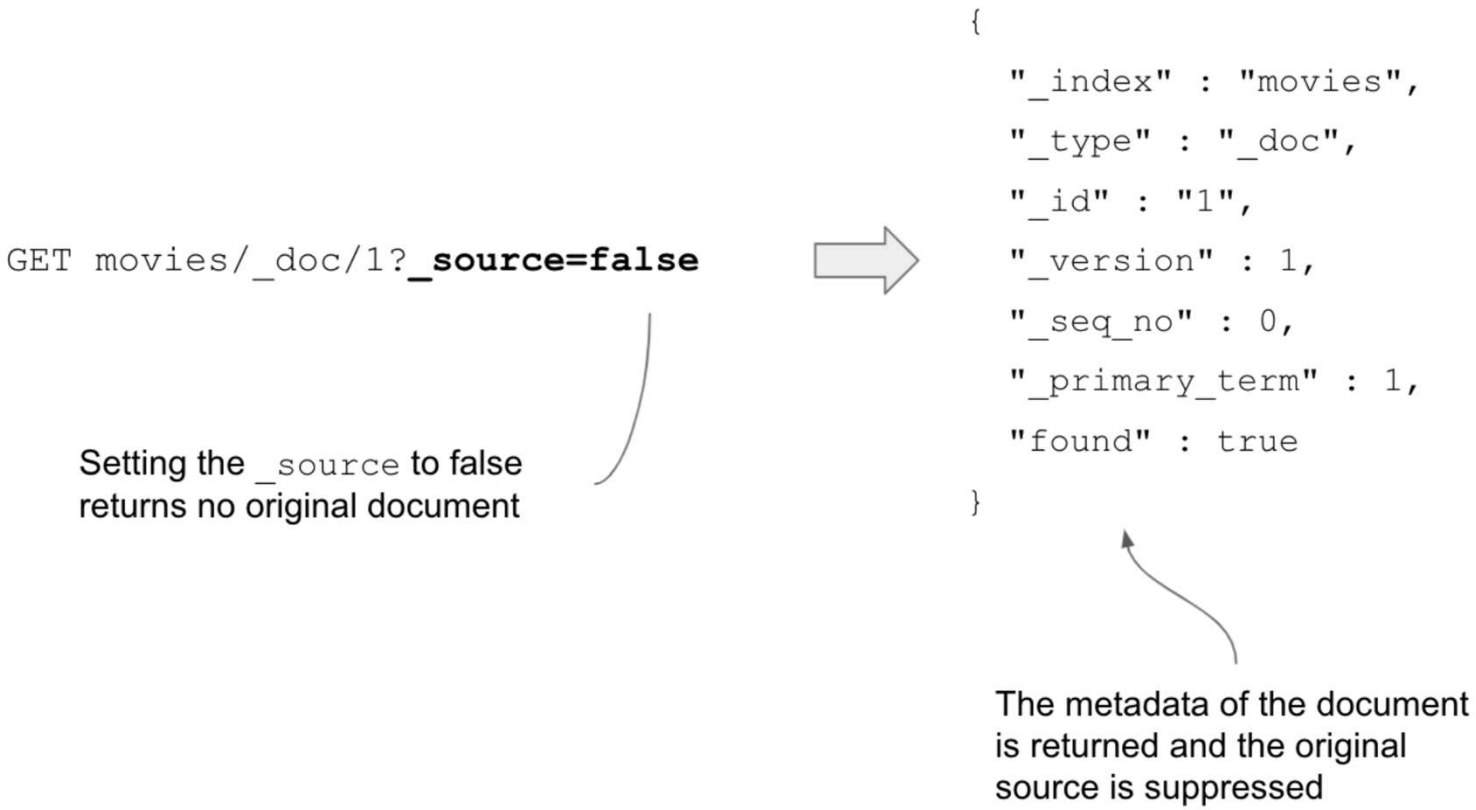
Fetching multiple documents using the _mget API



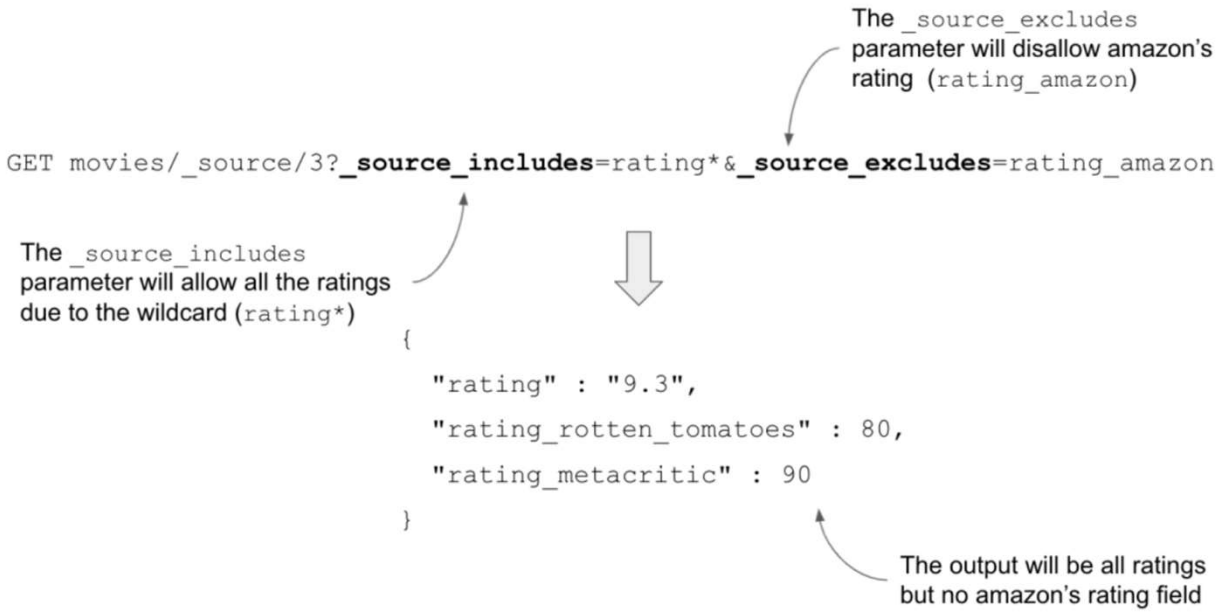
Invoking the _source endpoint returns the original document with no metadata



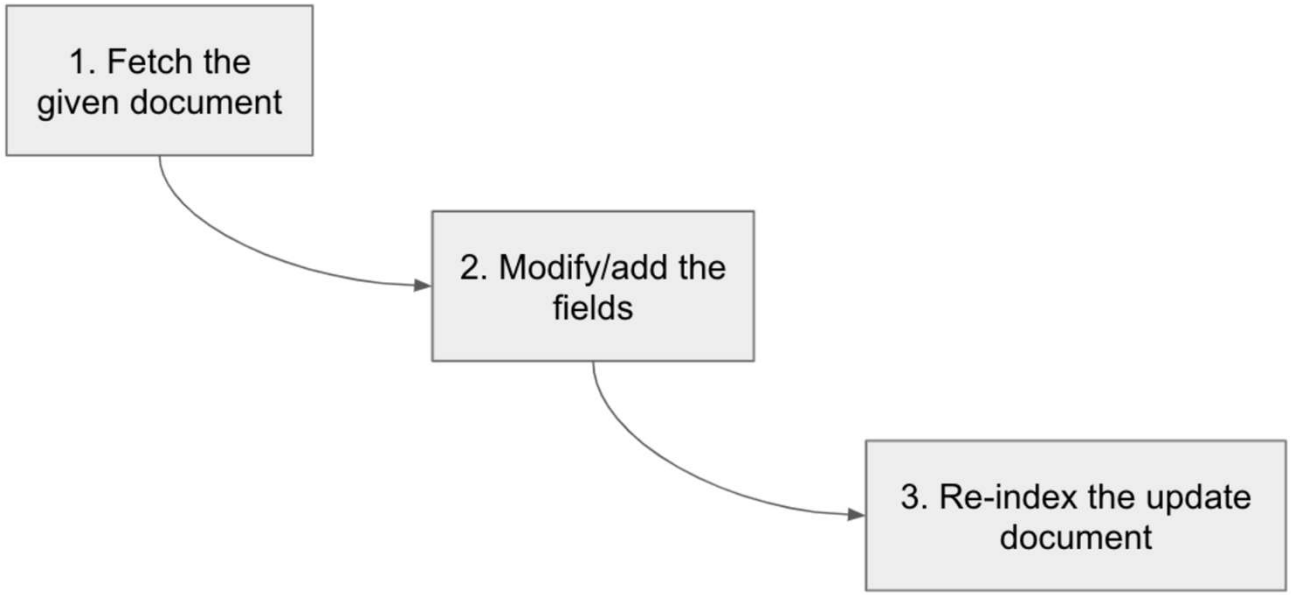
Setting the _source flag in the request returns metadata only



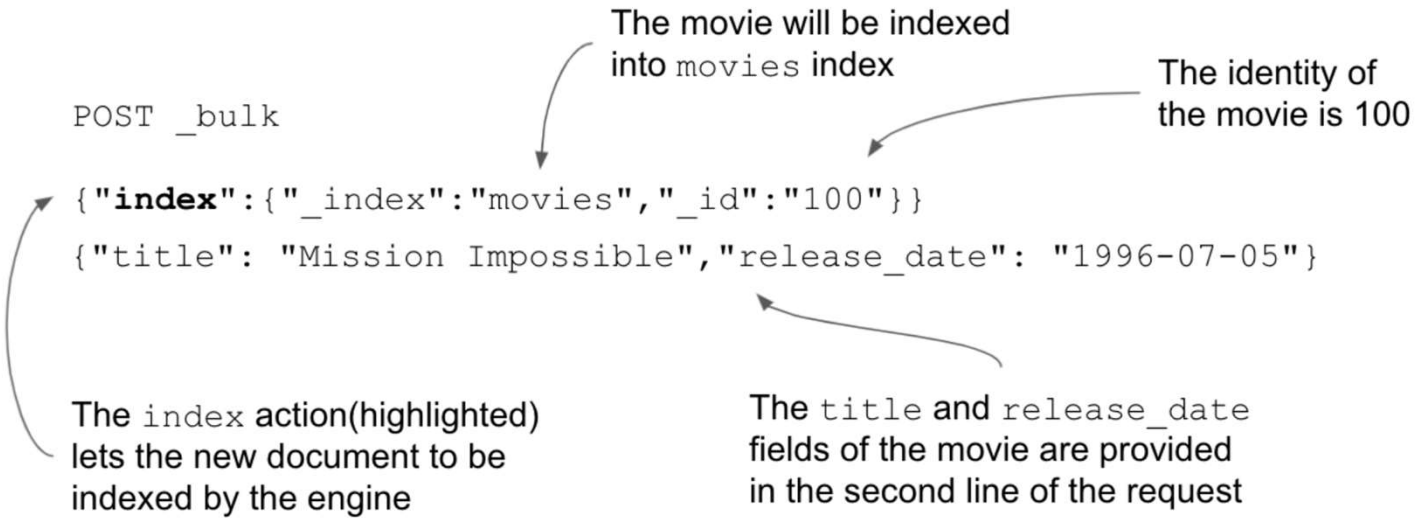
Tweaking what attributes can and cannot be as part of the return result



Updating or modifying the document is a three-step process



Indexing the new movie Mission Impossible using the _bulk API



Thanks