Elasticsearch

REAL-TIME SEARCH AND ANALYTICS ENGINE

Elasticsearch

- Open-source search engine
  - Built on top of Apache Lucene
    - Fulltext search engine library
  - Written in Java
- It can also be described as follows:
  - A distributed real-time document store where every field is indexed and searchable
  - A distributed search engine with real-time analytics
  - Capable of scaling to hundreds of servers and petabytes of structured and unstructured data

Talking to Elasticsearch

- Java API
  - Two built-in clients:
    - Node client
    - Transport client
- RESTful API with JSON over HTTP

Indices

- An Elasticsearch cluster contain multiple indices
  - Wich in turn contain multiple types
    - Types hold multiple documents
      - Each document has multiple fields.

JSON Documents

- Objects in an application are seldom just a simple list of keys and values
- Best way to serialize objects is JavaScript Object Notation:
  - All languages handles JSON in same and easy manner
  - It's only text, then they came through HTTP channels without any problem in security and protocol
  - It's human-readable
- It hold properties with a name-value convention, where
  - name (key) is the name of property or field
  - value is the instance value of property

Elasticsearch Document

- A document doesn’t consist only of data
  - It also has metadata
    - Information about the document
- _index
  - Like as database in RDBMS, is the place we store and index related data
- _type
  - Strong representation of data, linked with a mapping
- _id
  - The unique identifier in specified index and type
    - The key of document is tuple (_index, _type, _id)

Request a Document

- Simple query:
  - GET index_name/_search
    - Search all document in given index
- Elasticsearch provides a rich, flexible, query language called the query DSL,
  - which allows us to build much more complicated, robust queries
  - The domain-specific language (DSL) is specified using a JSON request body

elasticsearch

nr

Relevance Score

- By default, Elasticsearch sorts matching results by their relevance score,
  - that is, by how well each document matches the query
- This concept of relevance is important to Elasticsearch,
  - and is a concept that is completely foreign to traditional relational databases,
    - in which a record either matches or it doesn't

Life Inside a Cluster

Nodes and Cluster

- Node is a running instance of Elasticsearch
- Cluster consists of one or more nodes with the same cluster.name that are working together to share their data and workload
- One node in the cluster is elected to be the master node
  - is in charge of managing cluster-wide changes like creating or deleting an index, or adding or removing a node from the cluster
- Every node knows where each document lives and can forward our request directly to the nodes that hold the data we are interested in
  - Whichever node we talk to manages the process of gathering the response from the node or nodes holding the data and returning the final response to the client
- It is all managed transparently by Elasticsearch.

Cluster Health

- Many statistics can be monitored in an Elasticsearch cluster, but the single most important one is cluster health, which reports a status of either green, yellow, or red:
  - GET /_cluster/health
- The returned status field is the one we're most interested in
  - The status field provides an overall indication of how the cluster is functioning
  - The meanings of the three colors are provided here for reference:
    - 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

Sharding

- To add data to Elasticsearch, we need an index
  - a place to store related data
- index is just a logical namespace that points to one or more physical shards
  - a low-level worker unit that holds just a slice of all the data in the index
- Our documents are stored and indexed in shards,
  - but our applications don’t talk to them directly, instead, they talk to an index
- Shards are how Elasticsearch distributes data around your cluster
  - Elasticsearch will automatically migrate shards between nodes so that the cluster remains balanced

Replicas

- A replica shard is just a copy of a primary shard
- Replicas are used to provide redundant copies of your data to protect against hardware failure,
  - and to serve read requests like searching or retrieving a document
- The number of primary shards in an index is fixed at the time that an index is created,
  - but the number of replica shards can be changed at any time

Create an Index

- PUT /index_name
  - Options:
    - settings:
      - numbers_of_shards
      - number_of_replicas

Query Data

Searching

- A search can be any of the following:
  - A structured query on concrete fields like gender or age, sorted by a field like join_date, similar to the type of query that you could construct in SQL
  - A full-text query, which finds all documents matching the search keywords, and returns them sorted by relevance
  - A combination of the two

Basic Concepts

- The Empty Search
  - The most basic form of the search API is the empty search,
    - which doesn’t specify any query but simply returns all documents in all indices in the cluster:
  - GET /_search
- In the response we found:
  - hits: total number of documents that matched query and and array containing first 10
  - took: total milliseconds took
  - shards: total number of shards involved in query
  - timeout: tells if query timed out

Multi-index, Multitype

- If we want to search within one or more specific indices,
  - and probably one or more specific types,
- We can do this by specifying the index and type in the URL
- When you search within a single index, Elasticsearch forwards the search request to a primary or replica of every shard in that index,
  - and then gathers the results from each shard
- Searching within multiple indices works in exactly the same way
  - there are just more shards involved

- /_search
  - Search all types in all indices
- /gb/_search
  - Search all types in the gb index
- /gb,us/_search
  - Search all types in the gb and us indices
- /g*,u*/_search
  - Search all types in any indices beginning with g or beginning with u
- /gb/user/_search
  - Search type user in the gb index
- /gb,us/user,tweet/_search
  - Search types user and tweet in the gb and us indices
- /_all/user,tweet/_search
  - Search types user and tweet in all indices

Pagination

- Preceding empty search return all documents in the cluster match our
- (empty) query
  - But there were only 10 documents in the hits array
- How can we see the other documents?
  - Elasticsearch accepts the from and size parameters:
    - size
      - Indicates the number of results that should be returned, defaults to 10
    - from
      - Indicates the number of initial results that should be skipped, defaults to 0

Deep Pagination

- Deep paging is problematic
  - Let's imagine that we are searching within a single index with five primary shards
    - When we request the first page of results (results 1 to 10),
    - each shard produces its own top 10 results and returns them to the requesting node,
    - which then sorts all 50 results in order to select the overall top 10.
  - Now imagine that we ask for page 1,000—results 10,001 to 10,010.
    - Everything works in the same way except that each shard has to produce its top 10,010 results.
    - The requesting node then sorts through all 50,050 results and discards 50,040 of them!
- You can see that, in a distributed system, the cost of sorting results grows exponentially the deeper we page.
  - There is a good reason that web search engines don't return more than 1,000 results for any query

Search Lite

- There are two forms of the search API:
  - a “lite” query-string version
    - that expects all its parameters to be passed in the query string
    - Useful for running ad hoc queries from the command line
  - the full request body version
    - That expects a JSON request body
    - and uses a rich search language called the query DSL

Search Lite

- GET /_all/tweet/_search?q=tweet:elasticsearch
- GET /_search?q=%2Bname%3Ajohn+%2Btweet%3Amary
  - To send params +name:john +tweet:mary
    - Where + indicates conditions that must be satisfied
      - Similarly – indicates conditions that must not match
- _all field:
  - The query-string search uses the _all field unless another field name has been specified
    - This match on all fields

More Complicated Queries

- In query we can use operators to compare or aggregate
  - In this case the query must be encoded with percentage (%) encoding
    - +name:(mary john) +date:>2014-09-10 +(aggregations geo)

Query String Topics

- Lite query-string search is surprisingly powerful
  - Its query syntax allows us to express quite complex queries succinctly
  - This makes it great for throwaway queries from the command line or during development
  - However, you can also see that its terseness can make it cryptic and difficult to debug
    - And it's fragile: a slight syntax error in the query string, such as a misplaced -, :, /,or ", and it will return an error instead of results
- Finally, the query-string search allows any user to run potentially slow, heavy queries on any field in your index, possibly exposing private information or even bringing your cluster to its knees!

Mapping And Analysis

- Elasticsearch interprets our document structure, by requesting the mapping (or schema definition) for the data type
  - So fields of type date and fields of type string are indexed differently, and can thus be searched differently
  - You might expect that each of the core data types—strings, numbers, Booleans, and dates—might be indexed slightly differently
    - And this is true: there are slight differences
    - But by far the biggest difference is between fields that represent exact values (which can include string fields) and fields that represent full text
      - This distinction is really important
        - it’s the thing that separates a search engine from all other databases

Exact Values vs Full Text

- Data in Elasticsearch can be broadly divided into two types:
  - exact values
  - full text

Exact Values vs Full Text

- Exact values are exactly what they sound like
  - Examples are a date or a user ID,
    - but can also include exact strings such as a username or an email address
      - The exact value Foo is not the same as the exact value foo
      - The exact value 2014 is not the same as the exact value 2014-09-15
- Full text refers to textual data
  - usually written in some human language

like the text of a tweet or the body of an email

Exact Values

- Exact values are easy to query
  - The decision is binary; a value either matches the query, or it doesn't. This kind of query is easy to express with SQL:

```
WHERE name = "John Smith"
AND user_id = 2
AND date > "2014-09-15"
```

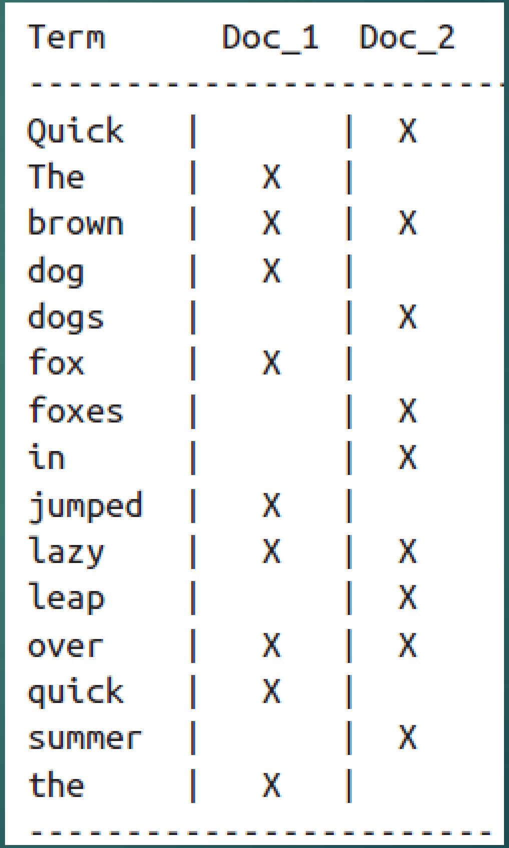
- Querying full-text data is much more subtle
  - We are not just asking, "Does this document match the query" but "How well does this document match the query?"
  - In other words, how relevant is this document to the given query?
    - We seldom want to match the whole full-text field exactly. Instead, we want to search within text fields. Not only that, but we expect search to understand our intent:
      - • A search for UK should also return documents mentioning the United Kingdom
      - • A search for jump should also match jumped, jumps, jumping, and perhaps even leap
      - • johnny walker should match Johnnie Walker, and johnnie depp should match Johnny Depp
      - • fox news hunting should return stories about hunting on Fox News, while fox hunting news should return news stories about fox hunting
  - To facilitate these types of queries on full-text fields, Elasticsearch first analyzes the text, and then uses the results to build an inverted index

Inverted Index

- Elasticsearch uses a structure called an inverted index, which is designed to allow very fast full-text searches
- An inverted index consists of a list of all the unique words that appear in any document, and for each word, a list of the documents in which it appears
  - For example, let’s say we have two documents, each with a content field containing the following:
    - 1. The quick brown fox jumped over the lazy dog
    - 2. Quick brown foxes leap over lazy dogs in summer
- To create an inverted index, we first split the content field of each document into separate words (which we call terms, or tokens), create a sorted list of all the unique terms, and then list in which document each term appears

Inverted Index

- Suppose to have a content field containing the following
  1. The quick brown fox jumped over the lazy dog
  2. Quick brown foxes leap over lazy dogs in summer
- An inverted index may looks like this to the right



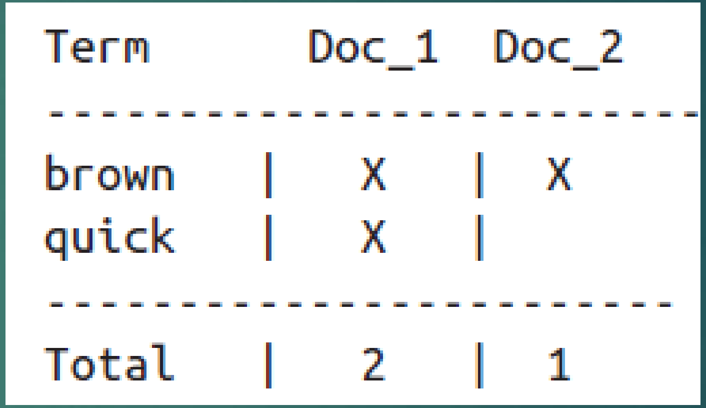
```
Term      Doc_1  Doc_2
-------------------------
Quick   |       |  X
The     |   X   |
brown   |   X   |  X
dog     |   X   |
dogs    |       |  X
fox     |   X   |
foxes   |       |  X
in      |       |  X
jumped  |   X   |
lazy    |   X   |  X
leap    |       |  X
over    |   X   |  X
quick   |   X   |
summer  |       |  X
the     |   X   |
------------------------
```

Inverted Index

- If we want to search for

  quick brown

- we just need to find the documents in which each term appears as to the right



```
Term      Doc_1  Doc_2
-------------------------
brown   |   X   |  X
quick   |   X   |
------------------------
Total   |   2   |  1
```

Inverted Index

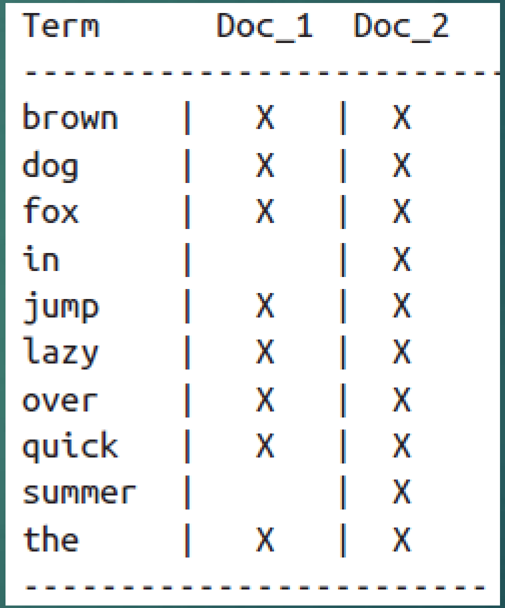
- Both documents match, but the first document has more matches than the second
  - If we apply a naive similarity algorithm that just counts the number of matching terms, then we can say that the first document is a better match than the second document
  - But there are a few problems with our current inverted index:
    - Quick and quick appear as separate terms, while the user probably thinks of them as the same word
    - fox and foxes are pretty similar, as are dog and dogs; They share the same root word
    - jumped and leap, while not from the same root word, are similar in meaning. They are synonyms

Inverted Index

- With the preceding index, a search for +Quick +fox wouldn't match any documents
  - Both the term Quick and the term fox have to be in the same document in order to satisfy the query
    - but the first doc contains quick fox and the second doc contains Quick foxes
- We reasonably expect both documents to match the query
  - If we normalize the terms into a standard format, then we can find documents that contain terms that are not exactly the same as the user requested,
    - but are similar enough to still be relevant
      - For instance:
        - Quick can be lowercased to become quick
        - foxes can be stemmed--reduced to its root form—to become fox. Similarly, dogs could be stemmed to dog
        - jumped and leap are synonyms and can be indexed as just the single term jump

Inverted Index

- The index now is like this to the right
- This process of tokenization and normalization is called analysis



```
Term        Doc_1   Doc_2
-------------------------
brown     |   X   |   X
dog       |   X   |   X
fox       |   X   |   X
in        |       |   X
jump      |   X   |   X
lazy      |   X   |   X
over      |   X   |   X
quick     |   X   |   X
summer    |       |   X
the       |   X   |   X
-------------------------
```

Analysis and Analyzers

- Analysis is a process that consists of the following:
  - • First, tokenizing a block of text into individual terms suitable for use in an inverted index,
  - • Then normalizing these terms into a standard form to improve their "searchability," or recall
- This job is performed by analyzers.
- An analyzer is really just a wrapper that combines three functions into a single package:
  - Character filters
    - First, the string is passed through any character filters in turn. Their job is to tidy up the string before tokenization.
      - A character filter could be used to strip out HTML, or to convert & characters to the word and
  - Tokenizer
    - Next, the string is tokenized into individual terms by a tokenizer. A simple tokenizer might split the text into terms whenever it encounters whitespace or punctuation
  - Token filters
    - Last, each term is passed through any token filters in turn, which can change terms (for example, lowercasing Quick), remove terms (for example, stopwords such as a, and, the) or add terms (for example, synonyms like jump and leap)
- Elasticsearch provides many character filters, tokenizers, and token filters out of the box
  - These can be combined to create custom analyzers suitable for different purposes

Built-in Analyzers

- Standard analyzer
- Simple analyzer
- Whitespace analyzer
- Language analyzer

Standard Analyzer

- The standard analyzer is the default analyzer that Elasticsearch uses
  - It is the best general choice for analyzing text that may be in any language
  - It splits the text on word boundaries, as defined by the Unicode Consortium, and removes most punctuation
  - Finally, it lowercases all terms
    - It would produce set, the, shape, to, semi, transparent, by, calling, set_trans, 5

Simple Analyzer

- The simple analyzer splits the text on anything that isn't a letter, and lowercases the terms
- It would produce
  - set, the, shape, to, semi, transparent, by, calling, set, trans

Whitespace Analyzer

- The whitespace analyzer splits the text on whitespace
- It doesn’t lowercase
- It would produce
  - Set, the, shape, to, semi-transparent, by, calling, set_trans(5)

Language Analyzer

- Language-specific analyzers are available for many languages
- They are able to take the peculiarities of the specified language into account
  - For instance, the english analyzer comes with a set of English stopwords (common words like and or the that don't have much impact on relevance), which it removes
  - This analyzer also is able to stem English words because it understands the rules of English grammar
  - The english analyzer would produce the following:
    - set, shape, semi, transpar, call, set_tran, 5
    - Note how transparent, calling, and set_trans have been stemmed to their root form

Analyzers

- When we index a document, its full-text fields are analyzed into terms that are used to create the inverted index
- However, when we search on a full-text field, we need to pass the query string through the same analysis process, to ensure that we are searching for terms in the same form as those that exist in the index
- Full-text queries understand how each field is defined, and so they can do the right thing:
  - When you query a full-text field, the query will apply the same analyzer to the query string to produce the correct list of terms to search for
  - When you query an exact-value field, the query will not analyze the query string, but instead search for the exact value that you have specified

Mapping

- In order to be able to treat date fields as dates, numeric fields as numbers, and string fields as full-text or exact-value strings, Elasticsearch needs to know what type of data each field contains
- This information is contained in the mapping.
  - Each document in an index has a type
  - Every type has its own mapping, or schema definition
  - A mapping defines the fields within a type, the datatype for each field, and how the field should be handled by Elasticsearch
  - A mapping is also used to configure metadata associated with the type

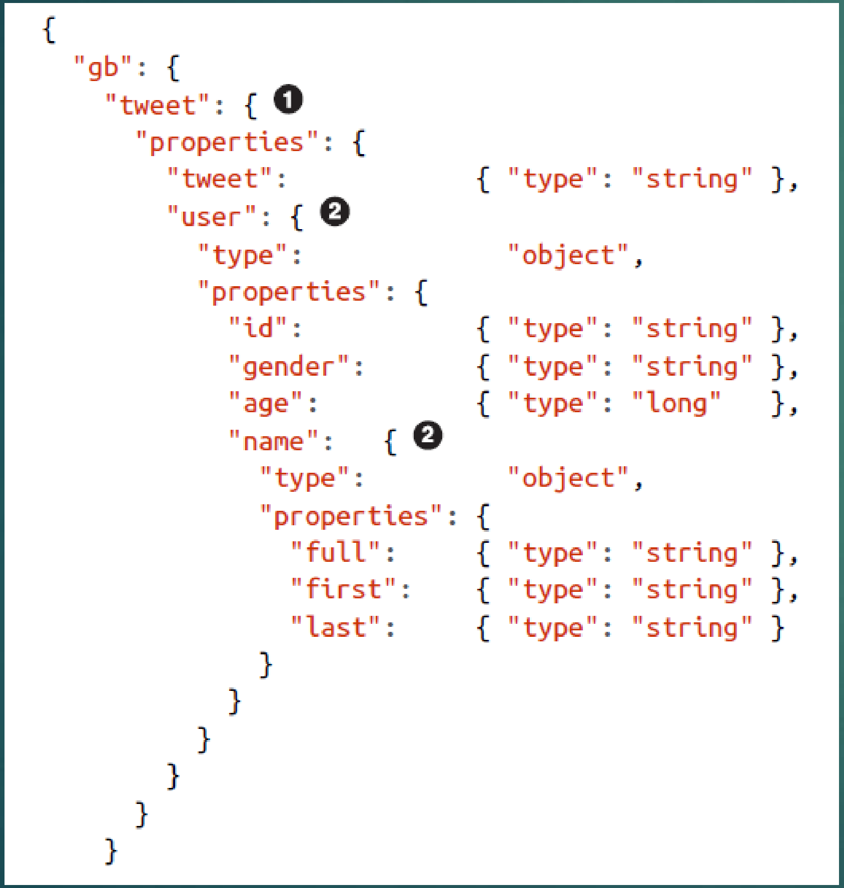
Mapping Simple Field Type

- Elasticsearch supports the following simple field types:
  - String: string
  - Whole number: byte, short, integer, long
  - Floating-point: float, double
  - Boolean: boolean
  - Date: date
- When you index a document that contains a new field Elasticsearch will use dynamic mapping to try to guess the field type from the basic datatypes available in JSON

View the Mapping

- To view mapping you can get it:
  - GET /index/_mapping
- To set mapping type, we can send PUT with mappings attribute
  - Each field can be set by attribute analyzer (i.e. english)
- If we cannot analyze a field, we can set on field the attribute
  - index = "not_analyzed"

Mapping for Inner Objects



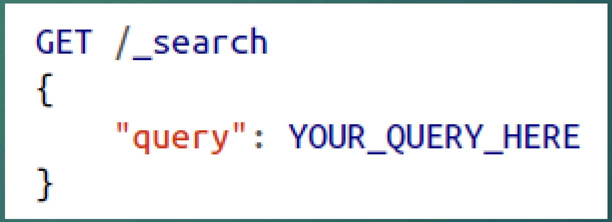
```
{
  "gb": {
    "tweet": { ❶
      "properties": {
        "tweet":            { "type": "string" },
        "user": { ❷
          "type":             "object",
          "properties": {
            "id":           { "type": "string" },
            "gender":       { "type": "string" },
            "age":          { "type": "long"   },
            "name":   { ❷
              "type":         "object",
              "properties": {
                "full":     { "type": "string" },
                "first":    { "type": "string" },
                "last":     { "type": "string" }
              }
            }
          }
        }
      }
    }
  }
```

1. Root object
2. Inner objects on different deep

- Indexing
  - Lucene doesn’t understand inner objects
    - Inner fields can be referred by name with punctation path

Query DSL

- The query DSL is a flexible, expressive search language that Elasticsearch uses to expose most of the power of Lucene through a simple JSON interface
- It is what you should be using to write your queries in production
- It makes your queries more flexible, more precise, easier to read, and easier to debug



```
GET /_search
{
    "query": YOUR_QUERY_HERE
}
```

GET Request with a Body?

- RFC 7231does not define what should happen to a GET request with a body!
  - As a result, some HTTP servers allow it, and some — especially caching proxies — don't
- The authors of Elasticsearch prefer using GET for a search request because they feel that it describes the action — retrieving information — better than the POST verb
- However, because GET with a request body is not universally supported, the search API also accepts POST requests
- The same rule applies to any other GET API that requires a request body

Query DSL

- Query DSL is the Domain Specific Language based on JSON to define queries
  - Think of the Query DSL as Abstract Syntax Tree of queries consisting of two types of clauses that behave differently depending on whether they are used in query context or filter context:
    - Leaf query clauses
      - Leaf query clauses look for a particular value in a particular field, such as the match, term or range queries
      - These queries can be used by themselves
    - Compound query clauses
      - Compound query clauses wrap other leaf or compound queries and are used to combine multiple queries in a logical fashion (such as the bool or dis_max query), or to alter their behaviour (such as the constant_score query)

Query DSL

- A query clause typically has this structure:

```
{
    QUERY_NAME: {
        FIELD_NAME: {
            ARGUMENT: VALUE,
            ARGUMENT: VALUE,...
        }
    }
}
```

Query Clause

- Query clauses are simple building blocks that can be combined with each other to create complex queries
- Clauses can be as follows:
  - Leaf clauses (like the match clause) that are used to compare a field (or fields) to a query string
  - Compound clauses that are used to combine other query clauses

Queries and Filters

- There are two DSLs:
  - the query DSL
  - the filter DSL
- A filter asks a yes | no question of every document and is used for fields that contain exact values:
  - Is the created date in the range 2013 - 2014?
  - Does the status field contain the term published?
  - Is the lat_lon field within 10km of a specified point?
- A query is similar to a filter, but also asks the question:
  - How well does this document match?

Queries and Filters

- A typical use for a query is to find documents
  - Best matching the words full text search
  - Containing the word run, but maybe also matching runs, running, jog, or sprint
  - Containing the words quick, brown, and fox—the closer together they are, the more relevant the document
  - Tagged with lucene, search, or java—the more tags, the more relevant the document
- A query calculates how relevant each document is to the query, and assigns it a relevance _score, which is later used to sort matching documents by relevance
- This concept of relevance is well suited to full-text search, where there is seldom a completely "correct" answer

Performance Differences

- The output from most filter clauses—a simple list of the documents that match the filter—is quick to calculate and easy to cache in memory, using only 1 bit per document
  - These cached filters can be reused efficiently for subsequent requests
- Queries have to not only find matching documents, but also calculate how relevant each document is, which typically makes queries heavier than filters
  - Also, query results are not cachable
- Thanks to the inverted index, a simple query that matches just a few documents may perform as well or better than a cached filter that spans millions of documents
- In general, however, a cached filter will outperform a query, and will do so consistently
- The goal of filters is to reduce the number of documents that have to be examined by the query

Queries and Filters

- As a general rule, use
  - query clauses for full-text search or for any condition that should affect the relevance score
  - use filter clauses for everything else

Most Important Queries and Filters

- Filters:
  - term, terms
    - Filter by exact values
  - range
    - find numbers or dates that fall into a specified range
    - Operators can accept:
      - gt, gte, lt, lte
  - exists and missing
    - used to find documents in which the specified field either has one or more values (exists) or doesn't have any values (missing)
    - These filters are frequently used to apply a condition only if a field is present, and to apply a different condition if it is missing

Most Important Queries and Filters

- Filters:
  - bool
    - Used to combine multiple filter clauses using boolean logic
    - Accepts three parameters:
      - must
        - must match, like and
      - must_not
        - musto not match, like not
      - should
        - At least one of clauses must match, like or

Most Important Queries and Filters

- Queries:
  - match_all
    - matches all documents
    - it is the default query that is used if no query has been specified
  - match
    - should be the standard query that you reach for whenever you want to query for a full-text or exact value in almost any field
    - if you run a match query against a full-text field, it will analyze the query string by using the correct analyzer for that field before executing the search
    - if you use it on a field containing an exact value, such as a number, a date, a Boolean, or a not_analyzed string field, then it will search for that exact value

Most Important Queries and Filters

- Queries:
  - multi_match
    - allows to run the same match query on multiple fields
  - bool
    - like the bool filter, is used to combine multiple query clauses
      - However, there are some differences
        - while filters give binary yes/no answers, queries calculate a relevance score instead. The bool query combines the _score from each must or should clause that matches
    - This query accepts the following parameters:
      - must
        - match for the document to be included
      - must_not
        - must not match for the document to be included
      - should
        - If these clauses match, they increase the _score; otherwise, they have no effect
        - They are simply used to refine the relevance score for each document.

Combining Queries with Filters

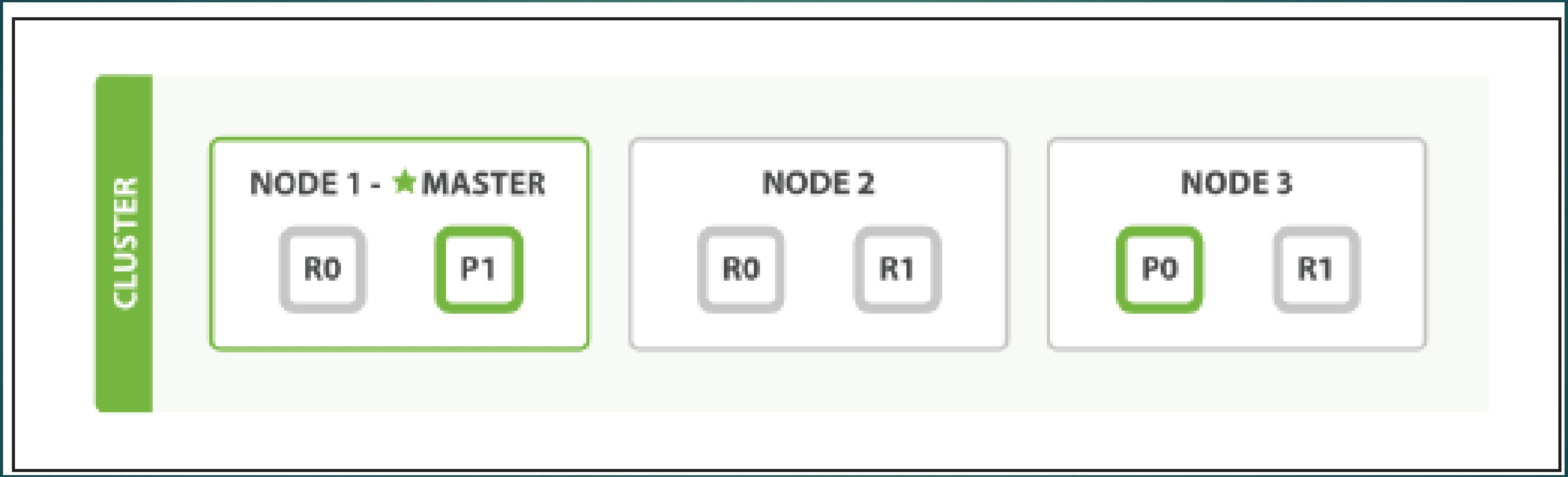
- Queries can be used in query context, and filters can be used in filter context
  - Throughout the Elasticsearch API, you will see parameters with query or filter in the name
  - These expect a single argument containing either a single query or filter clause respectively
  - In other words, they establish the outer context as query context or filter context
  - Compound query clauses can wrap other query clauses, and compound filter clauses can wrap other filter clauses
    - However, it is often useful to apply a filter to a query or, less frequently, to use a full-text query as a filter
    - To do this, there are dedicated query clauses that wrap filter clauses, and vice versa, thus allowing us to switch from one context to another
    - It is important to choose the correct combination of query and filter clauses to achieve your goal in the most efficient way

Distributed Document Store

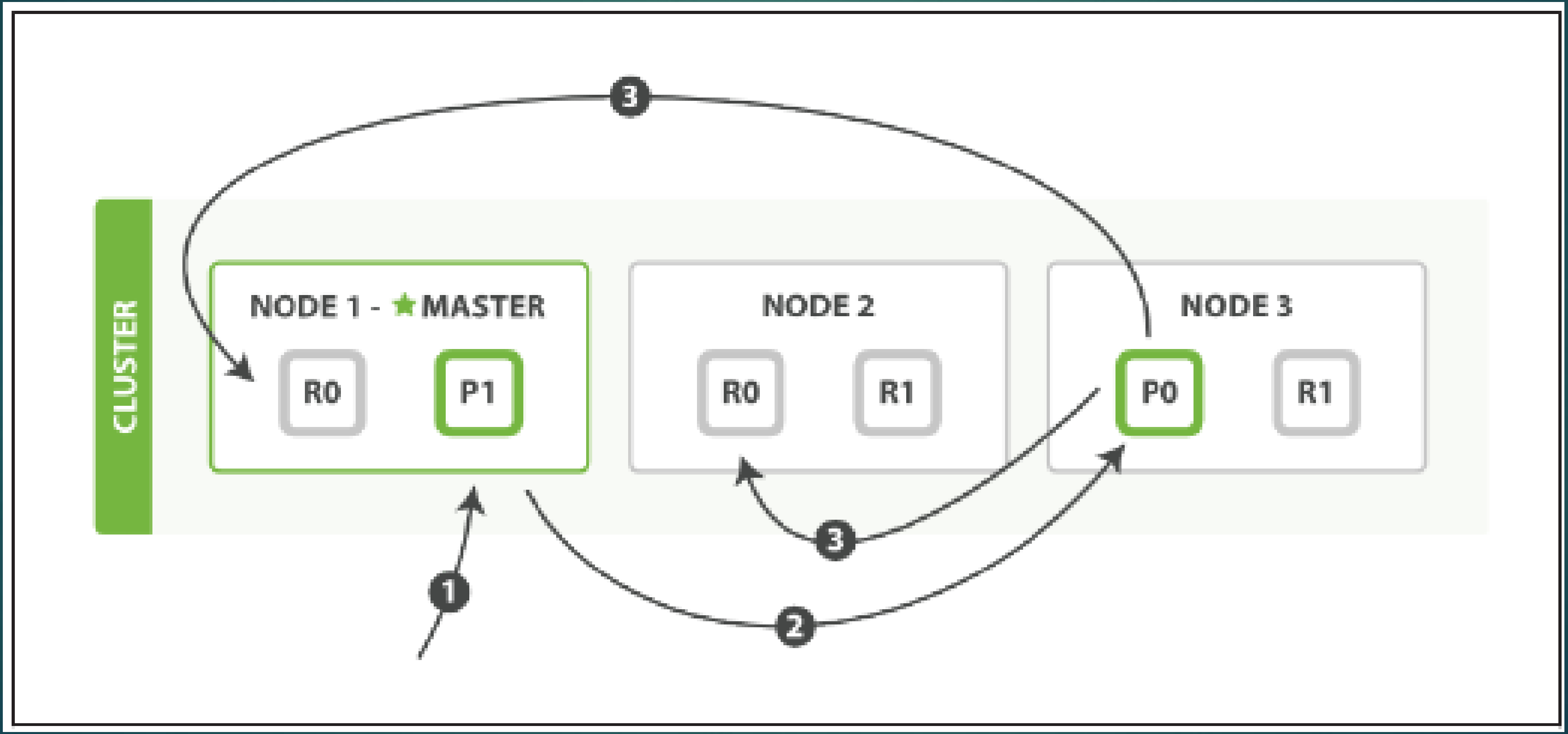
Distributed Document Store

- Document is stored on a single primary shard
  - The storage process can’t be random
    - Since we may need to retrieve the document in future
      - shard = hash(routing) % n_primary_shards

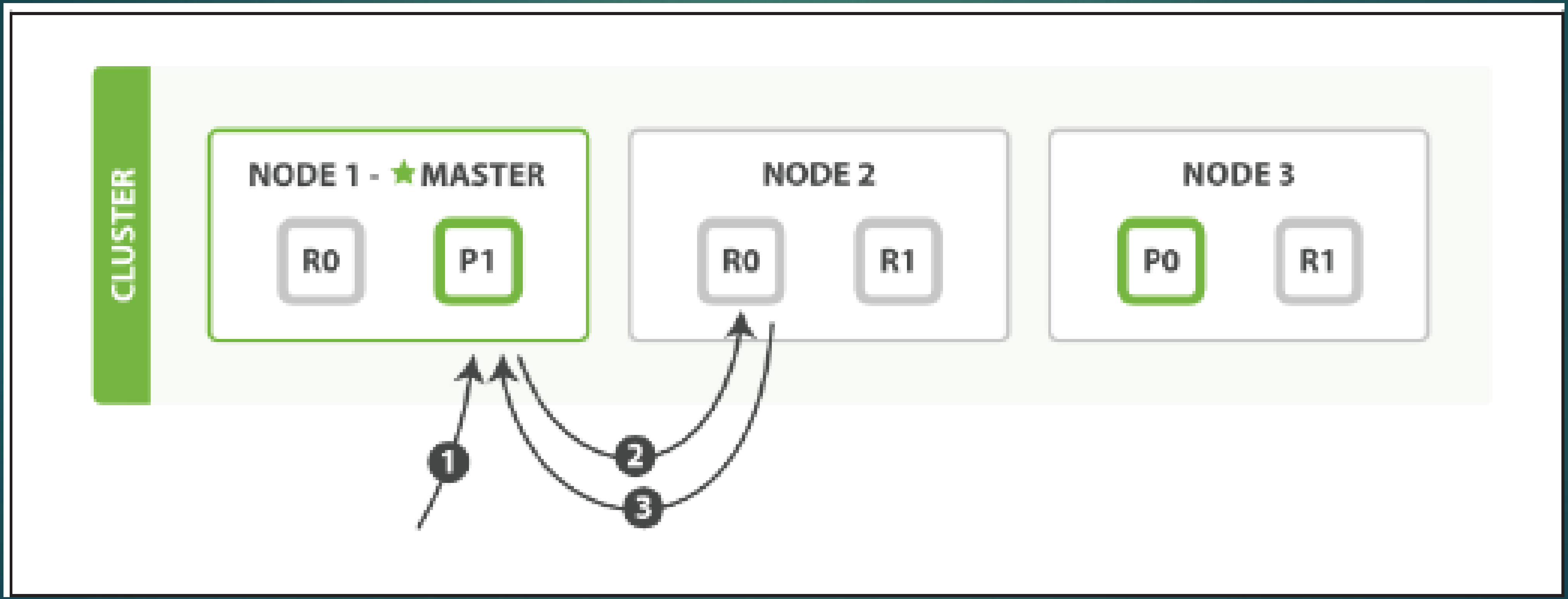
Interaction Between Primary and Replica Shards



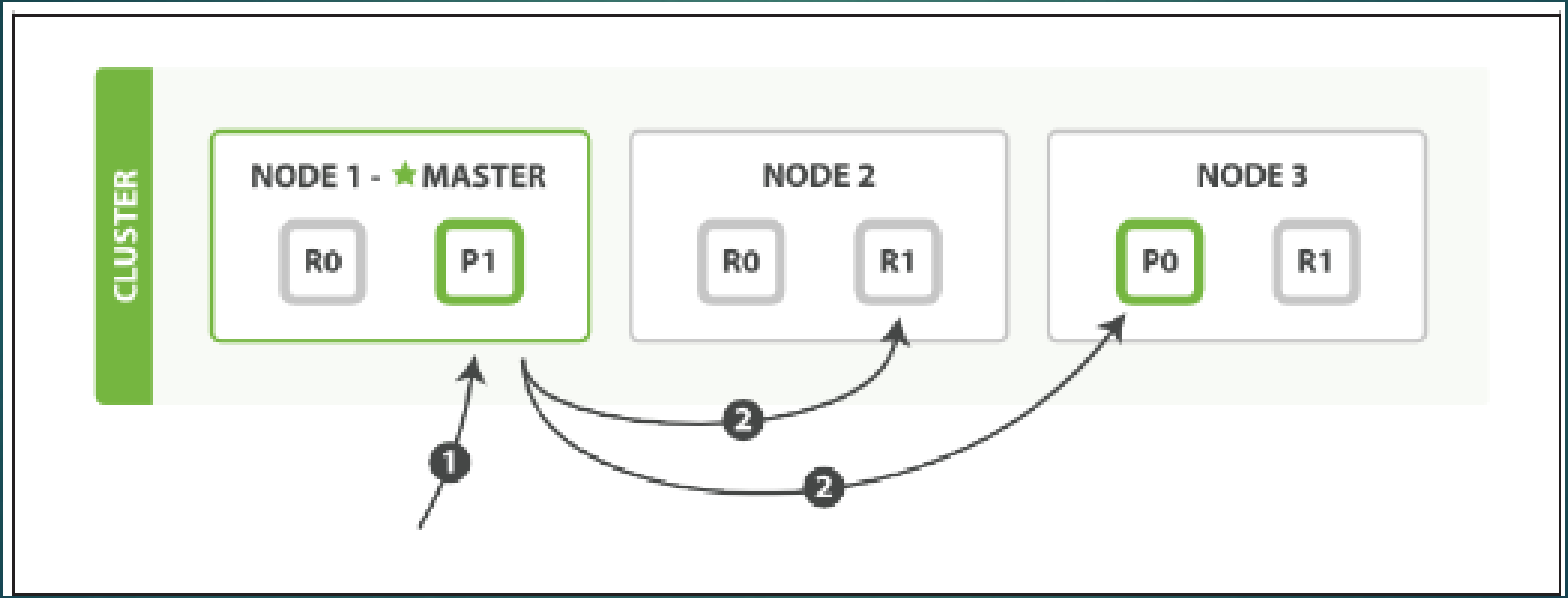
CRUD



Retrieving Document



Retrieving Documents



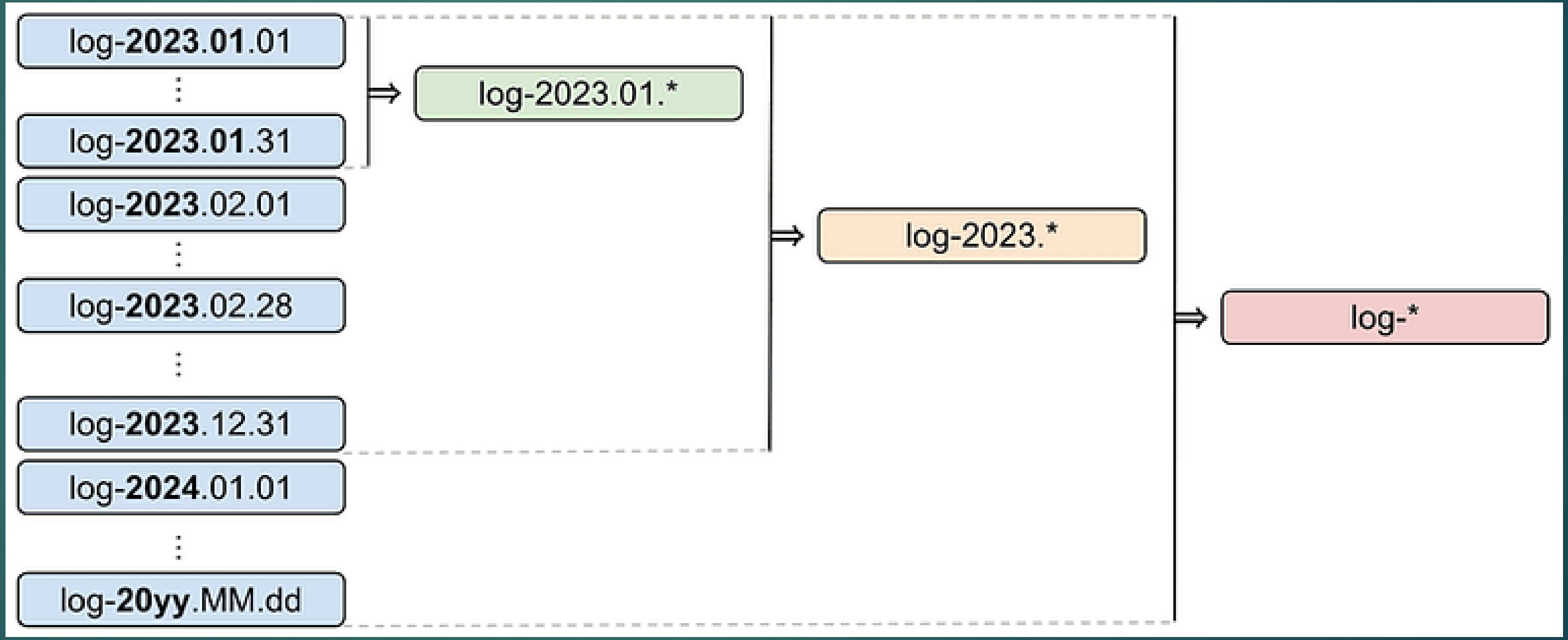
Index Pattern

Elastichsearch Index Pattern

- Allows users to define how to match and interact with multiple indices
  - key feature when working with data in Elasticsearch because it allows users to focus exactly on the data set that they want to work on

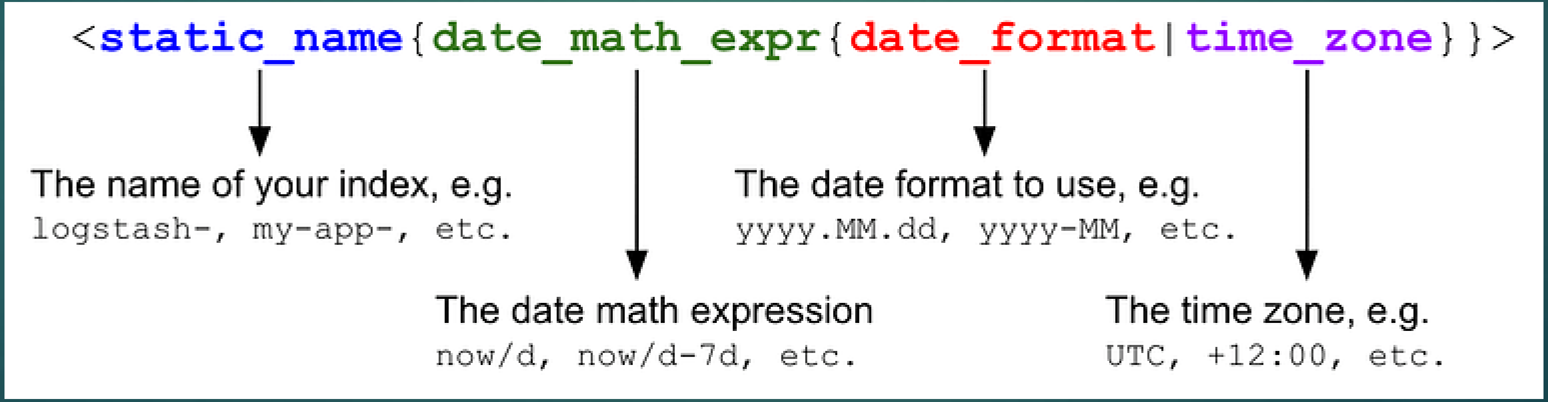
Index Patterns – Best Practices

- Use wildcards to match multiple indices



Index Patterns – Best Practices

- Utilize Date Math in Index Patterns
  - Date math expressions allow users to have Elasticsearch dynamically calculate dates based on the current date, which can be useful when working with time-based indices
  - Users can use date math in index patterns to match indices based on a specific time range



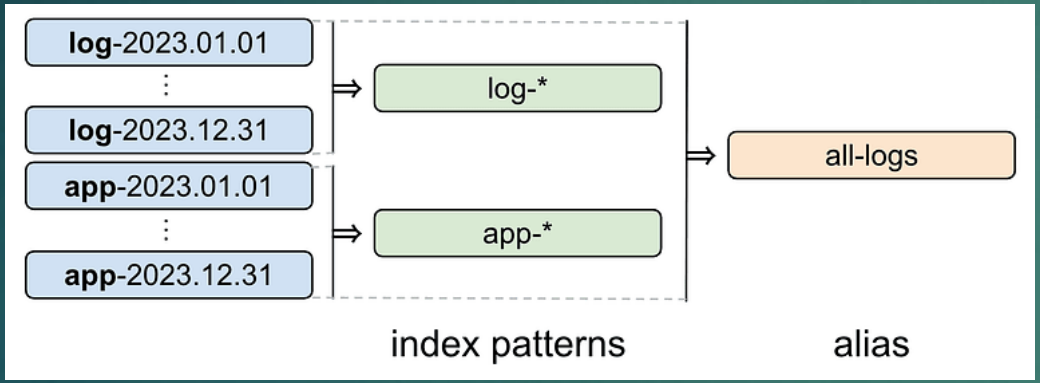
```
<static_name{date_math_expr{date_format|time_zone}}>
```

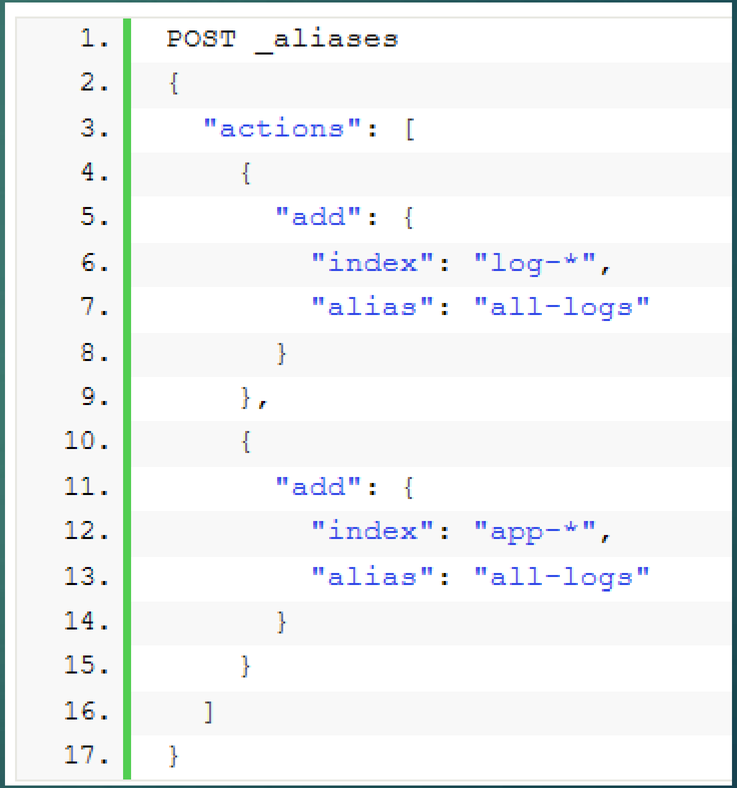
Index Patterns – Best Practices

- Keep Index Patterns Up-to-date
  - . Regularly review index patterns and update them as needed to maintain accurate search results and analytics

Index Patterns – Best Practices

- Use Aliases for Flexibility





```
POST _aliases
{
  "actions": [
    {
      "add": {
        "index": "log-*",
        "alias": "all-logs"
      }
    },
    {
      "add": {
        "index": "app-*",
        "alias": "all-logs"
      }
    }
  ]
}
```