elastic

The capabilities and applications of the elastic stack technology are limited only by your imagination.

Who is the Elastic Stack for?

Is this just for Security Analysts?

No, it is for everyone!

Community Driven

Open Source First

Security Data is also just known as data, it is NOT exclusive to security operations

Custom non relational database for use in development and custom data search

Predefined support for hundreds of services with pre-built data processing and dashboards

Heavily used in community security monitoring projects as well as a significant investment into a native security solution

and the beat goes on and on…

Connect with community

https://community.elastic.co/

https://community.elastic.co/united-states-and-canada-virtual/

https://www.meetup.com/topics/elasticsearch/

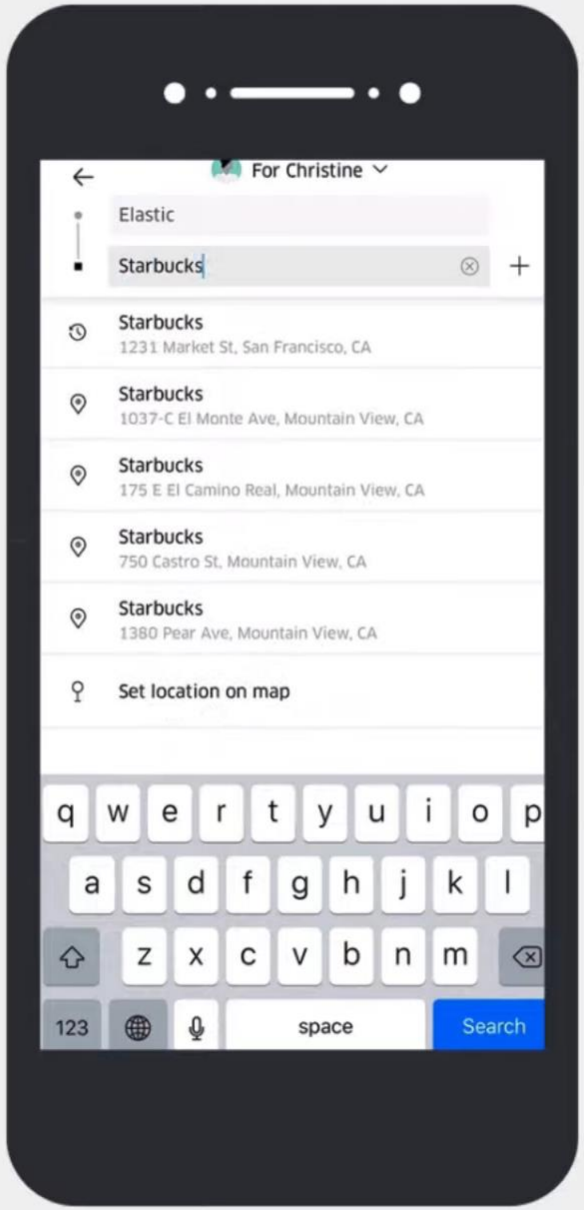
https://communityinviter.com/apps/elasticstack/elastic-community

Elastic Contributor Program

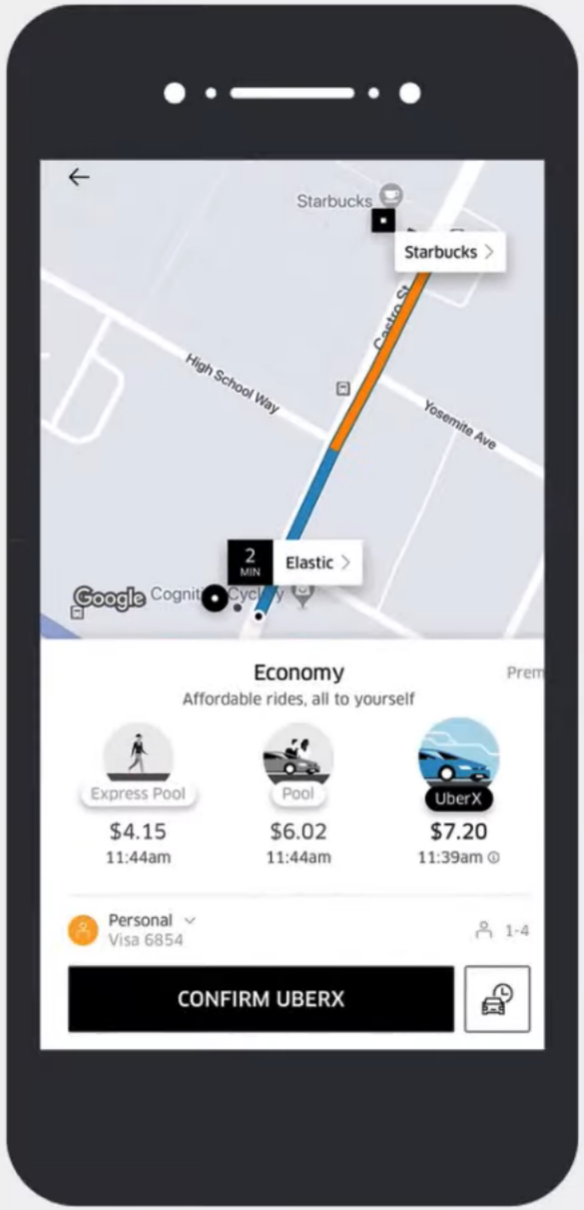
Join the Elastic Contributor Program, which recognizes the hard work of our awesome contributors!

Start contributing code, presentations, tutorials, and more today to earn yourself a spot on the leaderboard and the chance to win free training, Elastic Swag, bragging rights, and more...

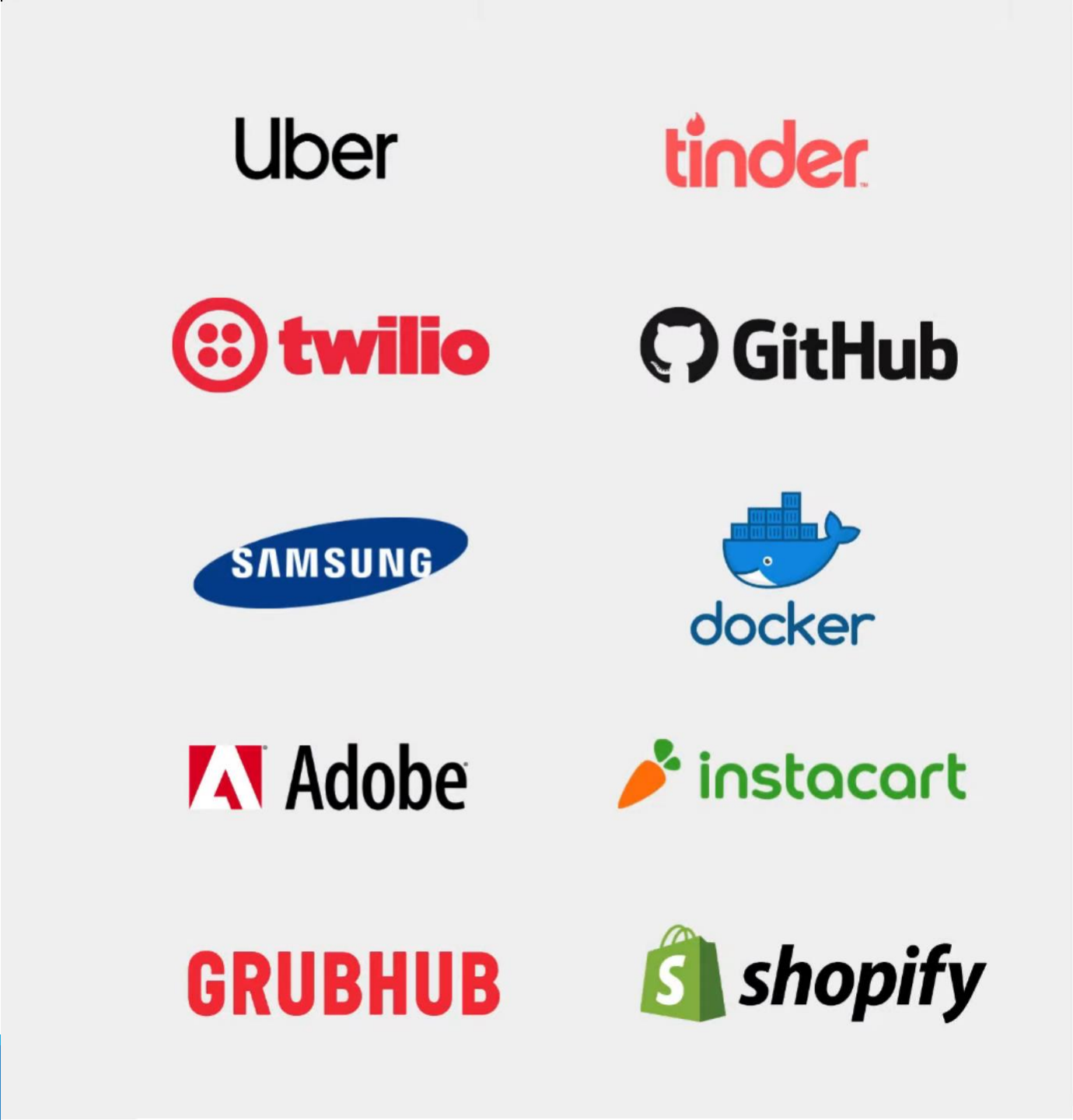
https://elastic.co/community/contributor



Searching for **Rides**



Searching for **Rides**

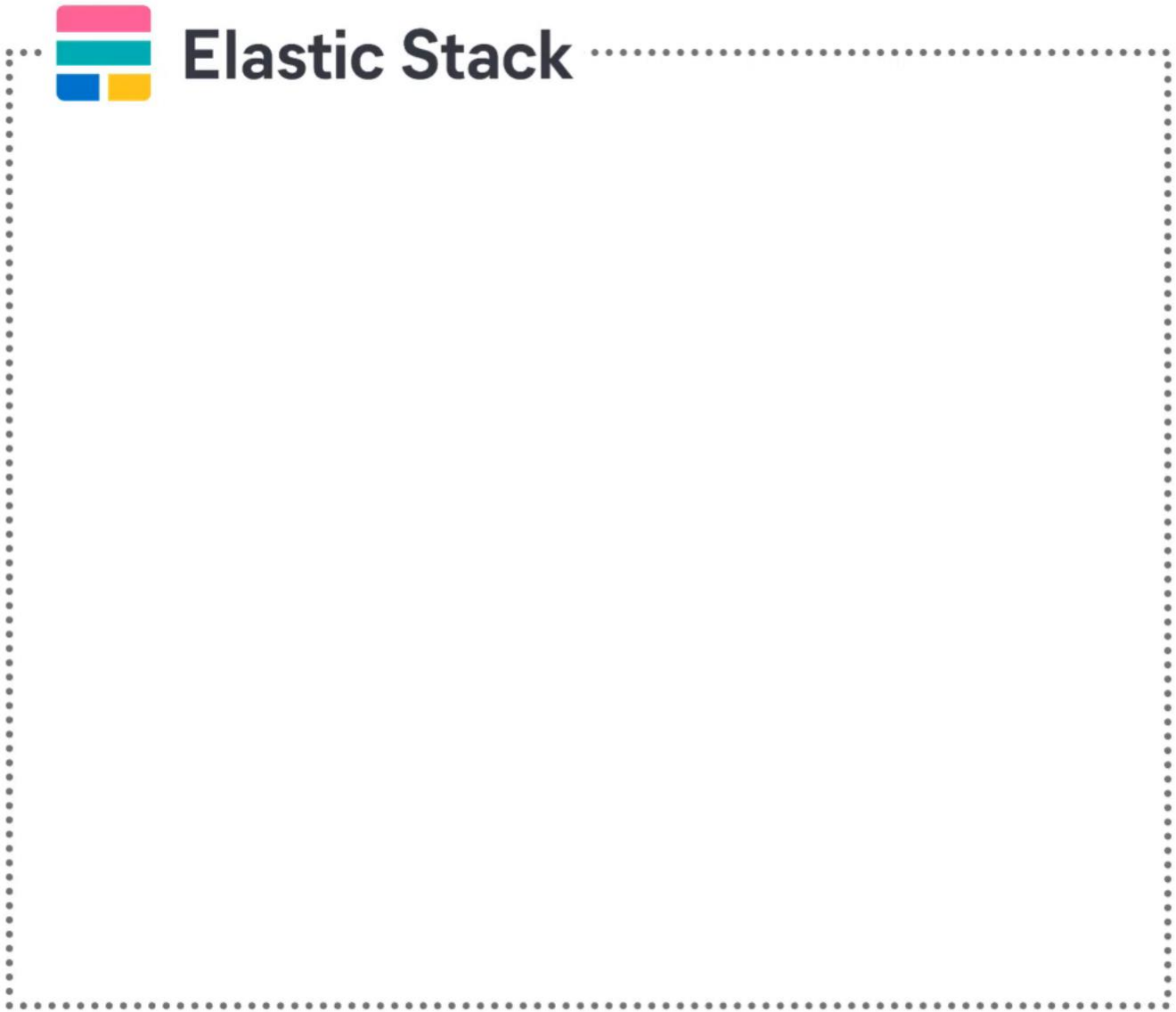


Searching for

- Restaurants
- Love
- Flights
- Jobs
- Groceries
- News
- Social Media
- Shopping
- Files
- Emails
- Tickets
- Errors
- Bugs
- Videos
- Threats
- Victims
- Adversaries
- Fraud
- Malware
- Anomalies

What Is the Elastic Stack?

Elastic Stack



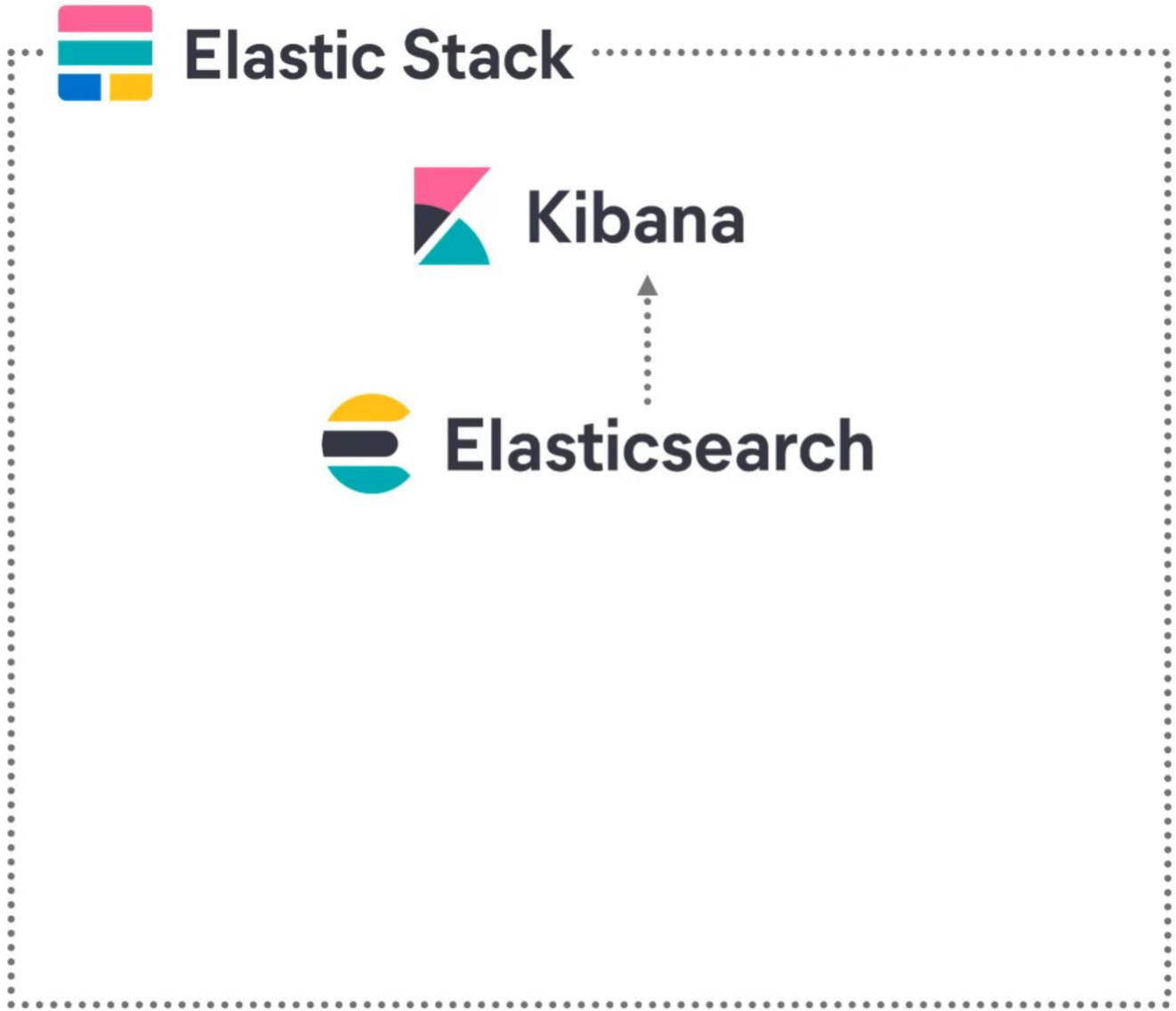
Elastic Stack

Elasticsearch

Elastic Stack

Elasticsearch

Kibana

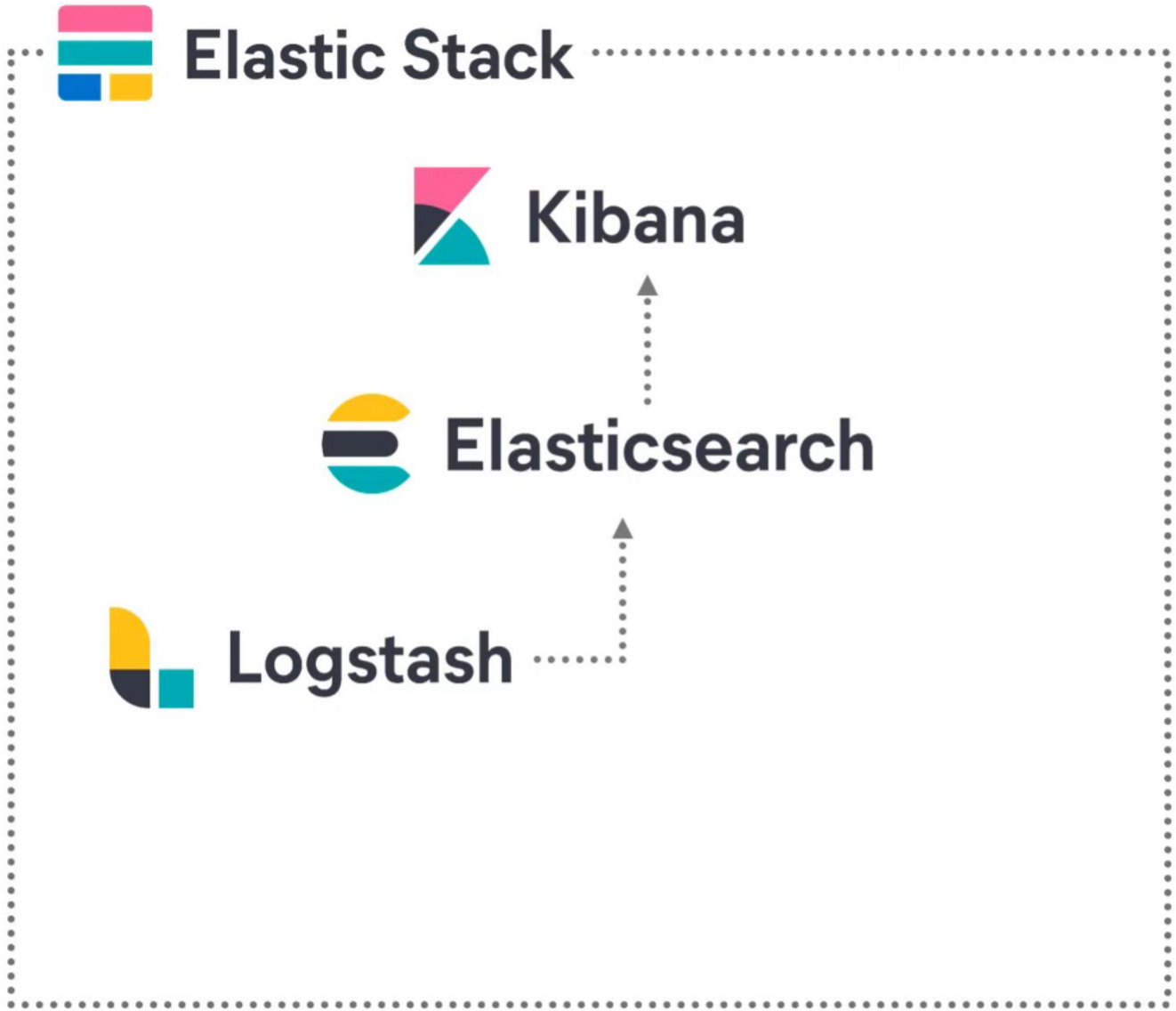


Elastic Stack

Elasticsearch

Kibana

Logstash



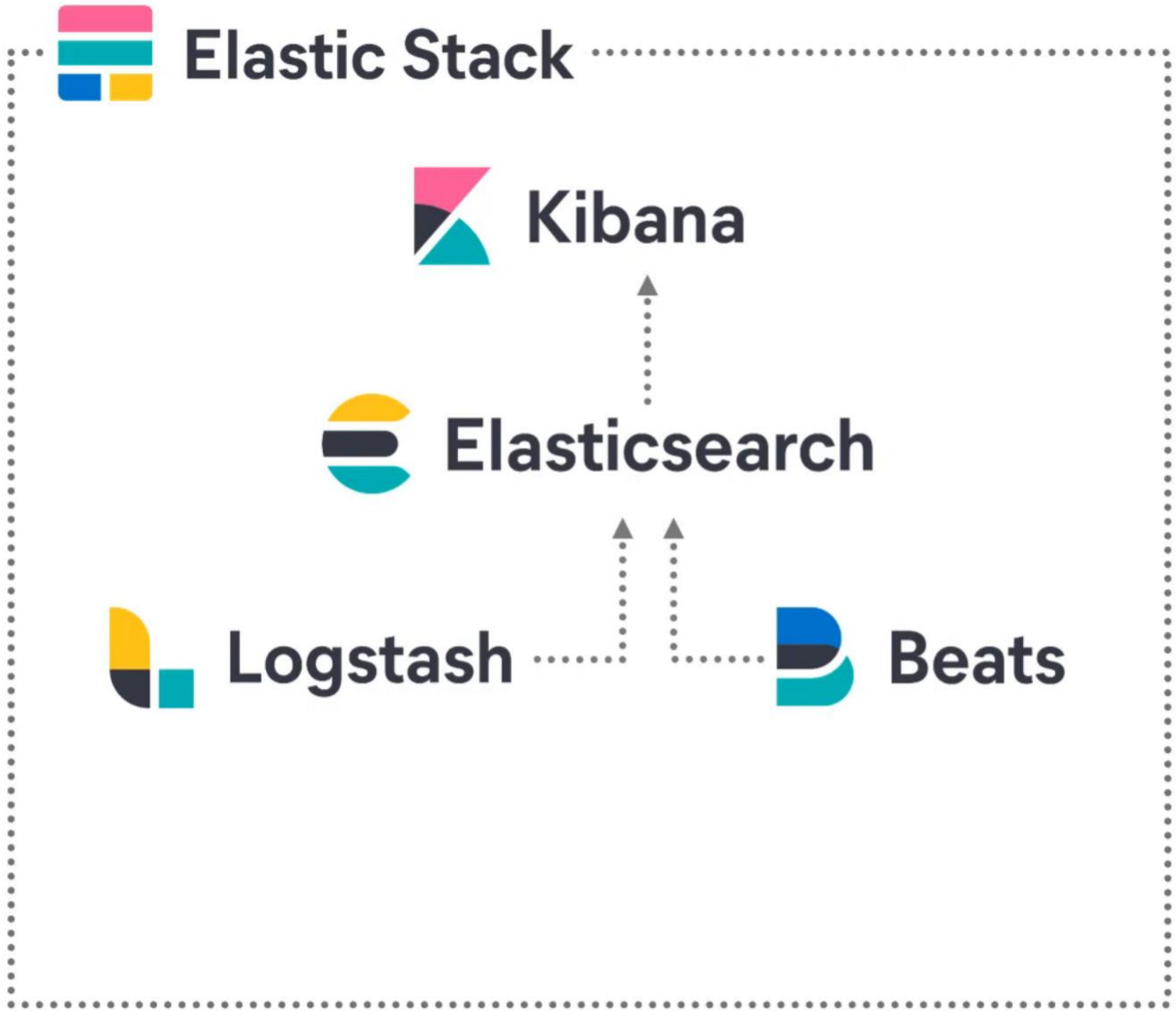
Elastic Stack

Elasticsearch

Kibana

Logstash

Beats



Elastic Stack

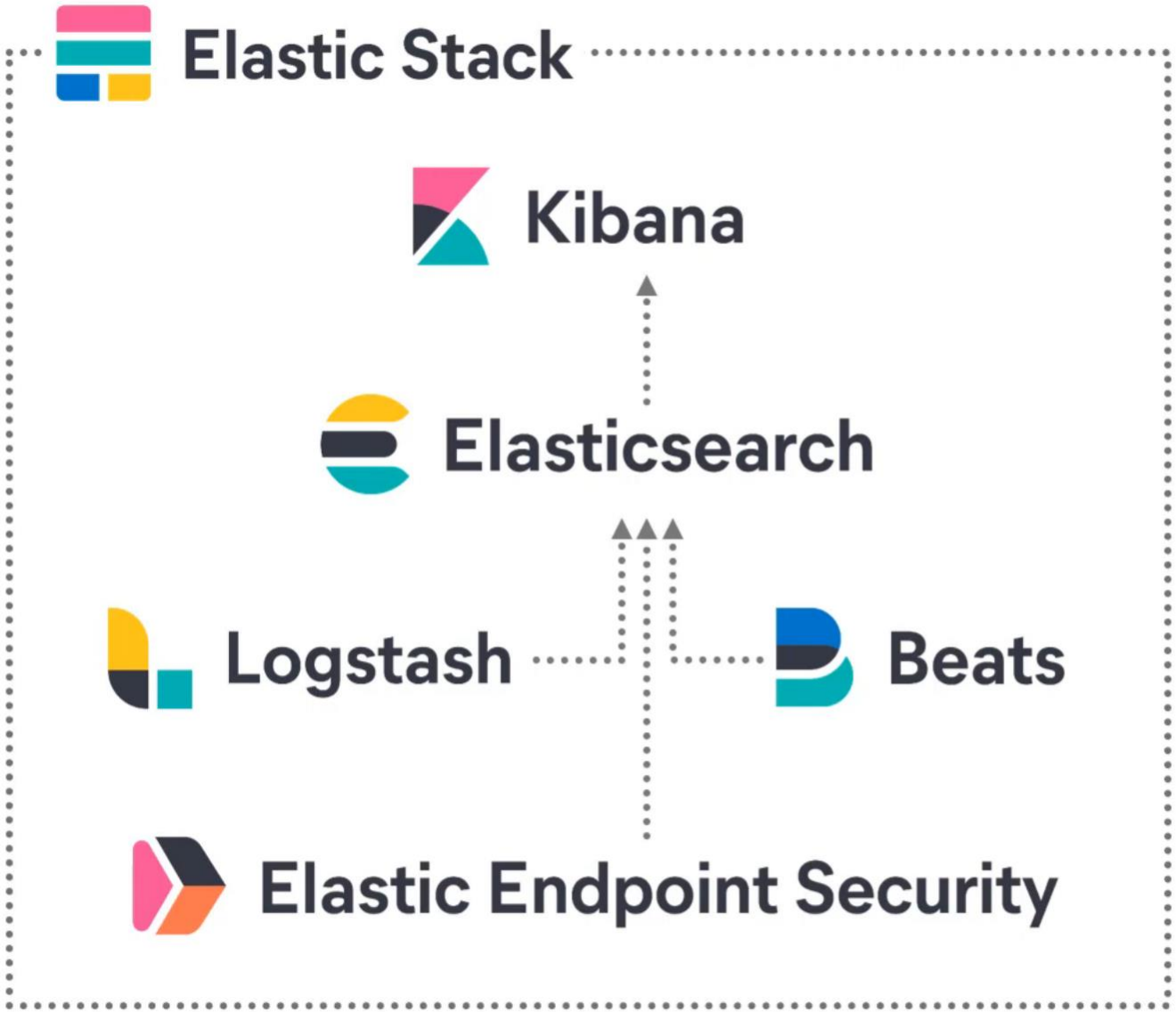
Elasticsearch

Kibana

Logstash

Beats

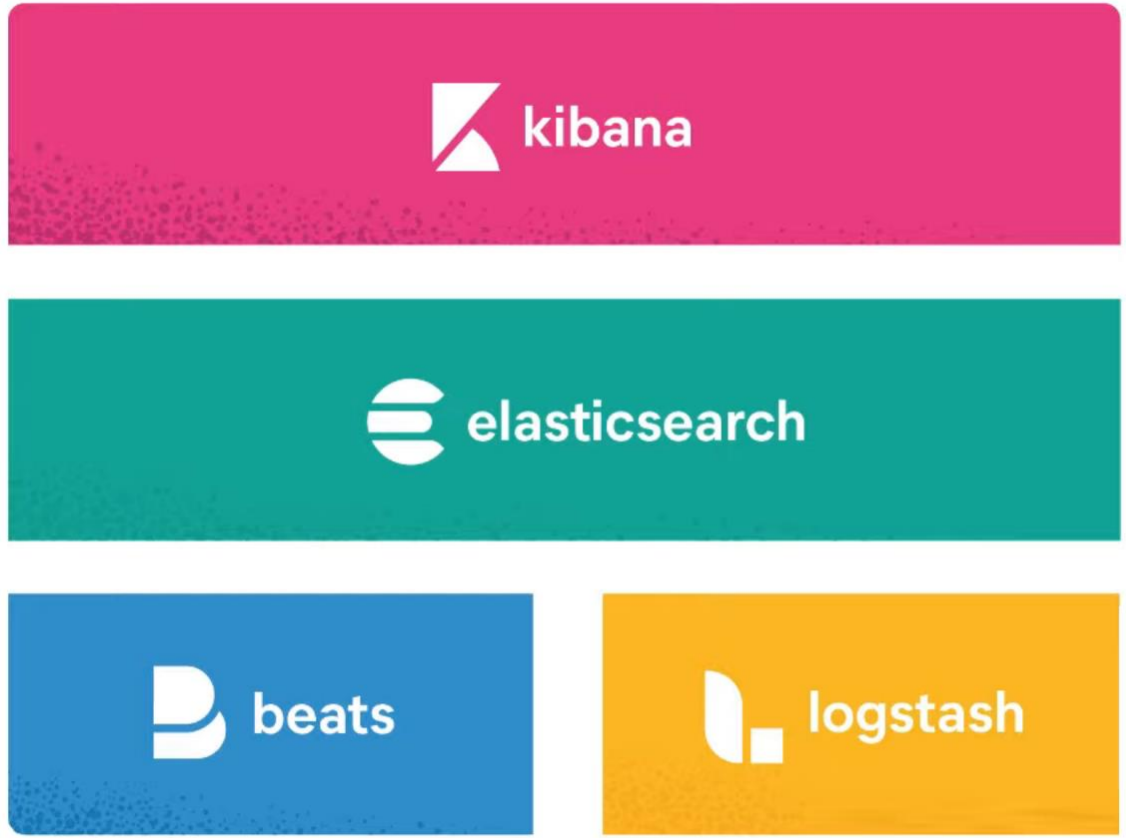
Endpoint Security



https://www.elastic.co/products/

The Elastic Stack

Reliably and securely take data from any source, in any format, then search, analyze, and visualize it in real time.



What Is the Elastic Stack?

Also known as the ELK Stack

Elasticsearch

- Provides a distributed, JavaScript Object Notation (JSON)-based search and analytics engine

Logstash

- Data processing pipeline to move and transform

Kibana

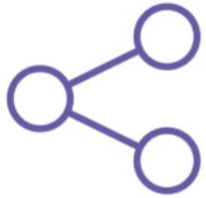
- An extensible user interface

Beats

- Data shippers

Why Use Elasticsearch

Very scalable and distributed search and analytics platform



Comes with search, aggregation and sharding capabilities

Used by many companies, from start-ups to top global companies such as Netflix and Microsoft

Provides you with the ability to log, index and search massive amounts of data

Use Cases

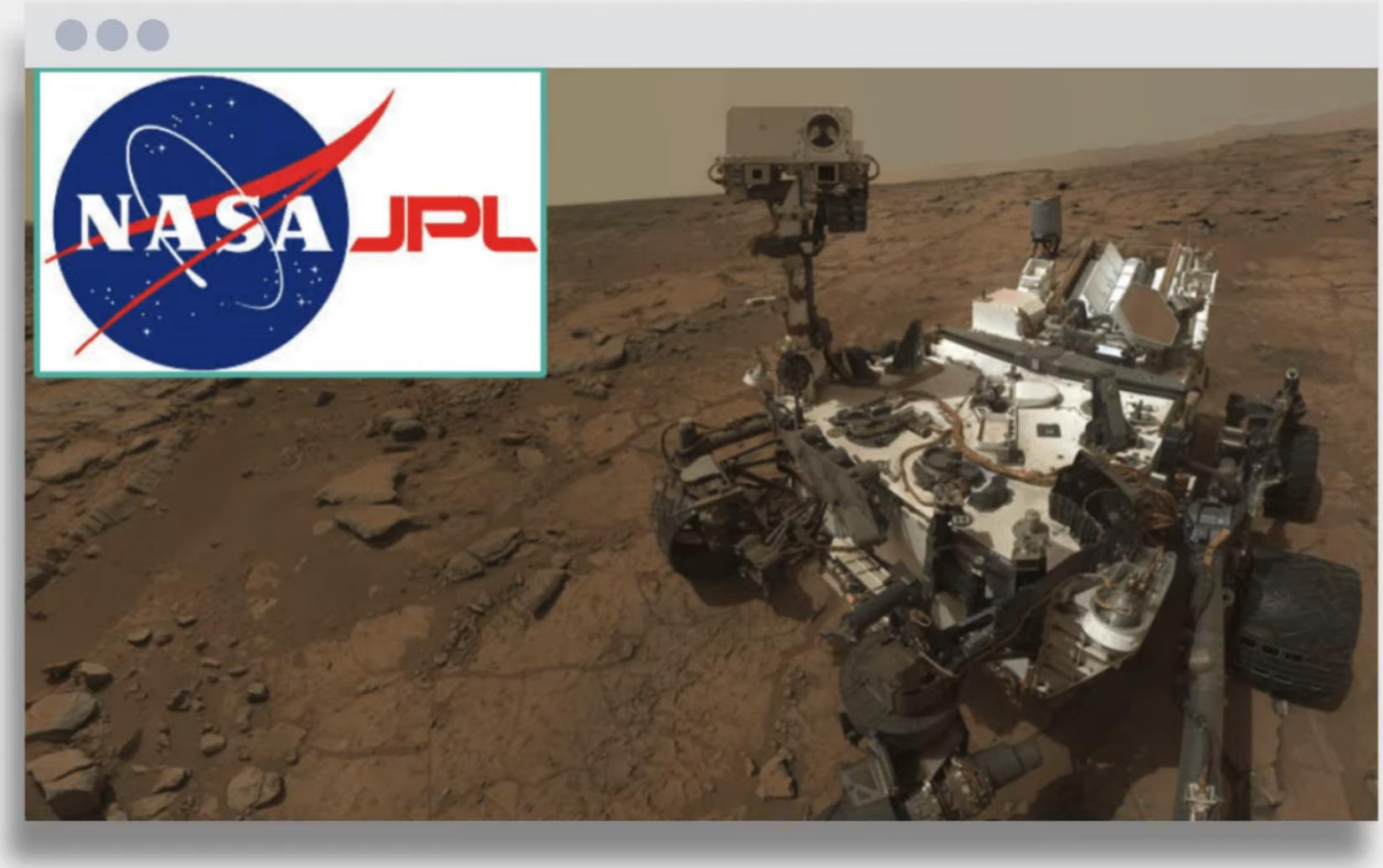
- **Logging**
- **Metrics**
- **Security Analytics**
- **Business Analytics**

Use Case: Logging

https://www.reddit.com/r/gaming/comments/4lhm69/overwatch_blocked_pharahs_rocket_with_hanzos_arrow/

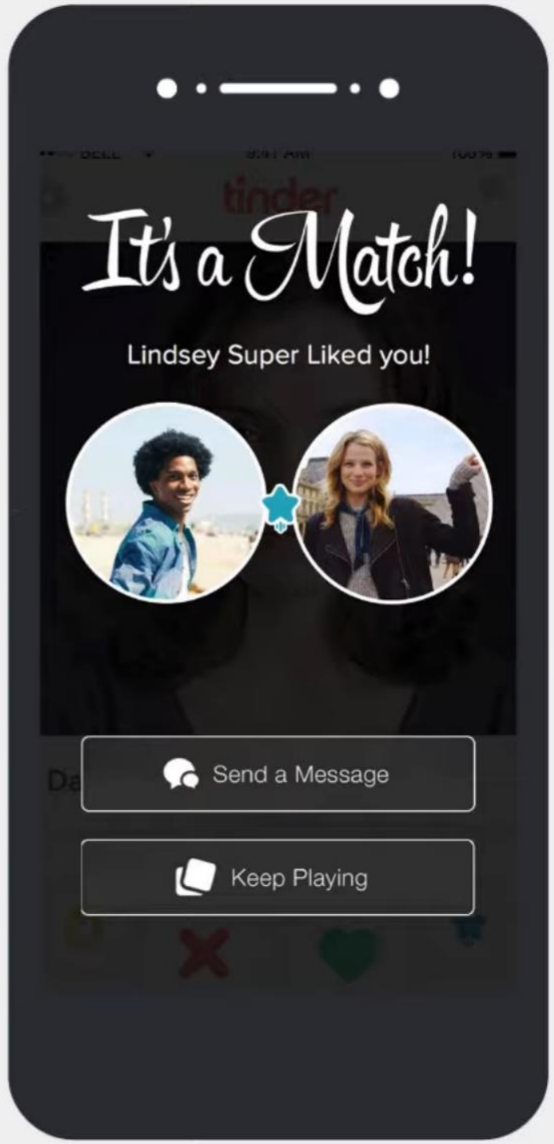
ACTIVISION BLIZZARD

Use Case: Metrics



Use Case: Security Analytics

Use Case: Business Analytics

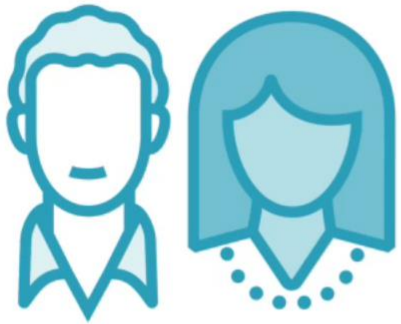


tinder™

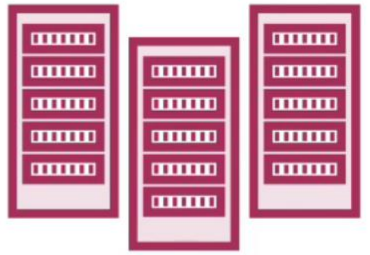
Your Role Today

DevOps / IT / Security

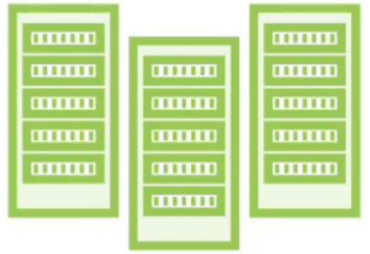
Your Role Today



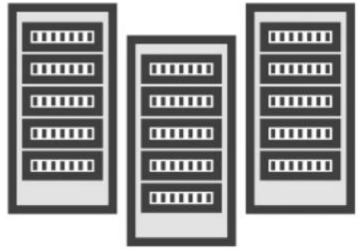
DevOps / IT / Security



United States



Europe

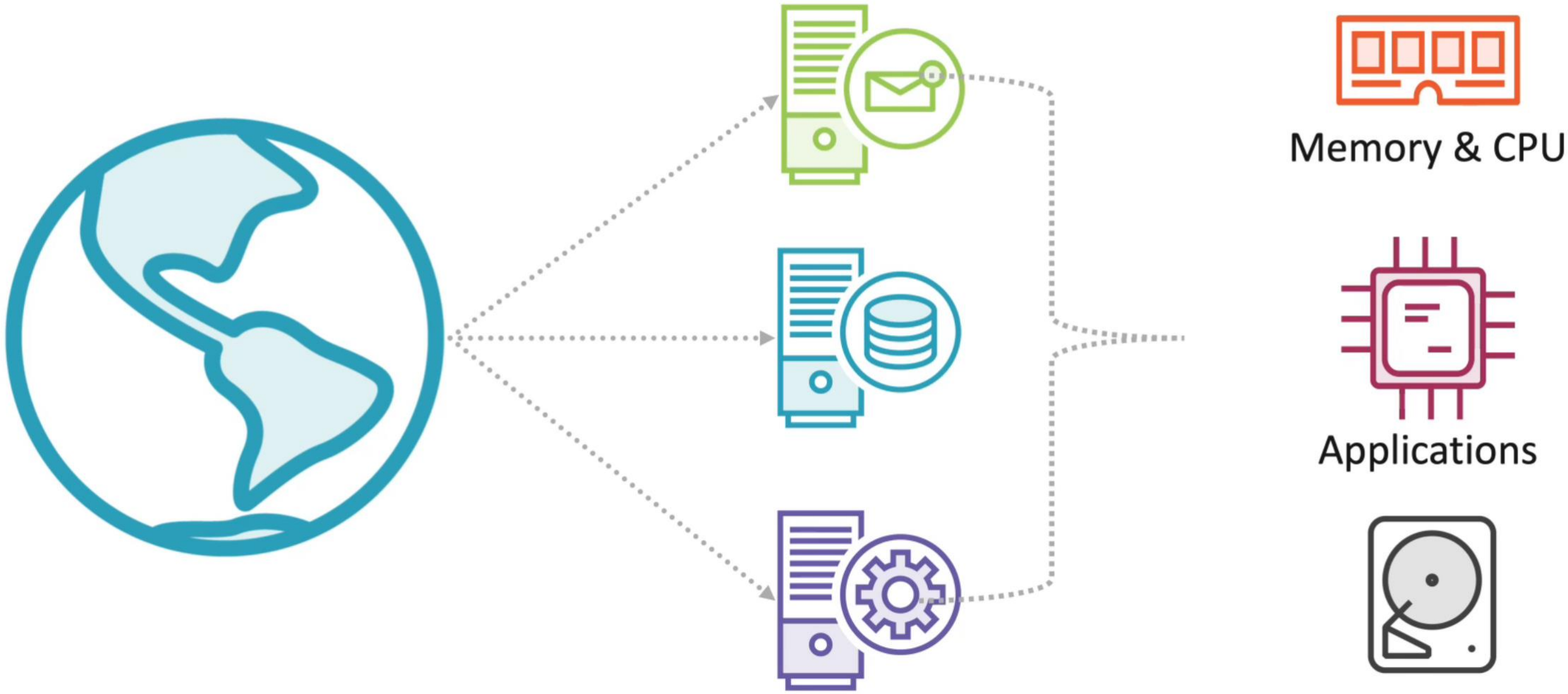


Asia

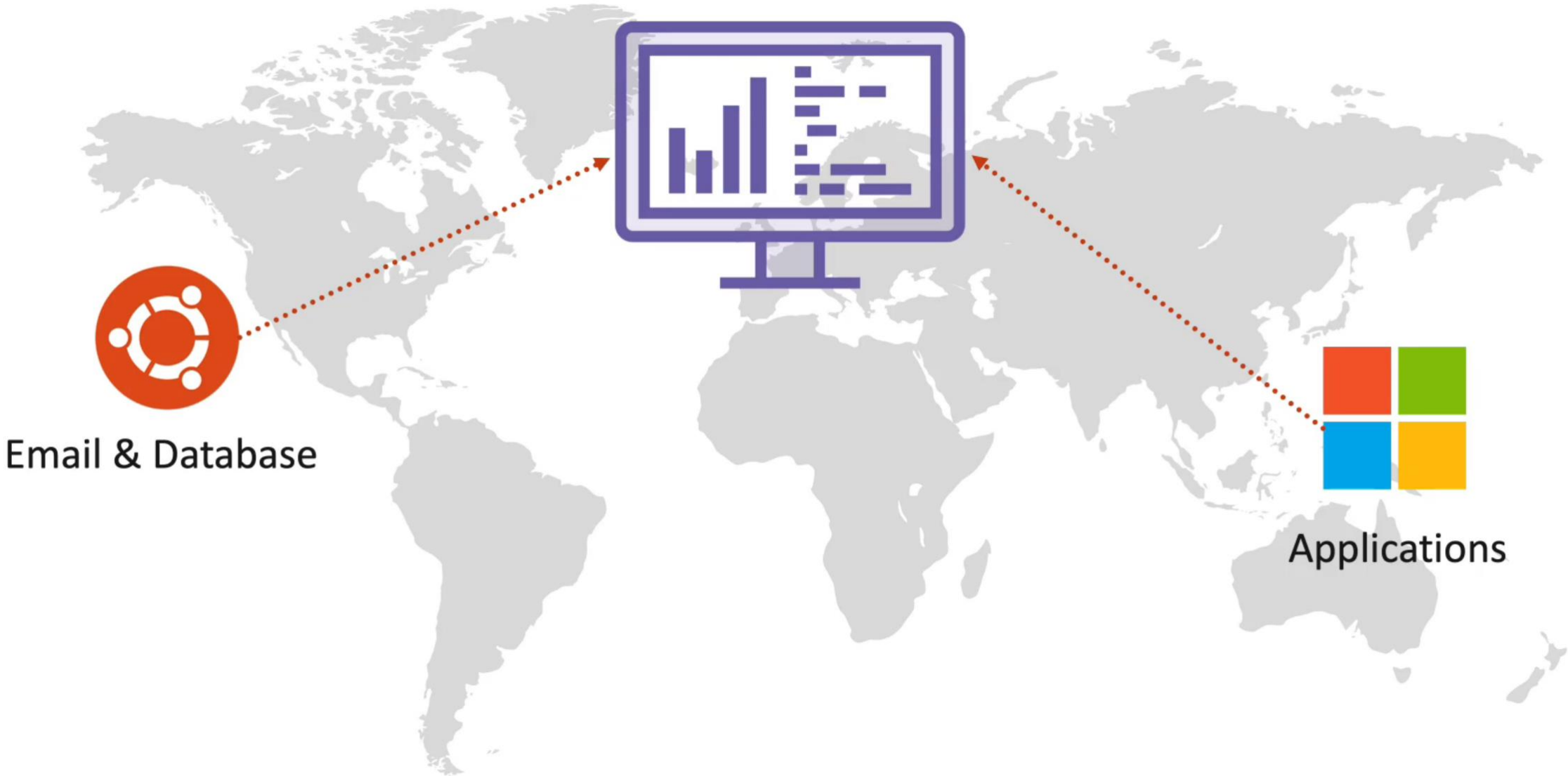
GLOBOMANTICS

Collecting Data

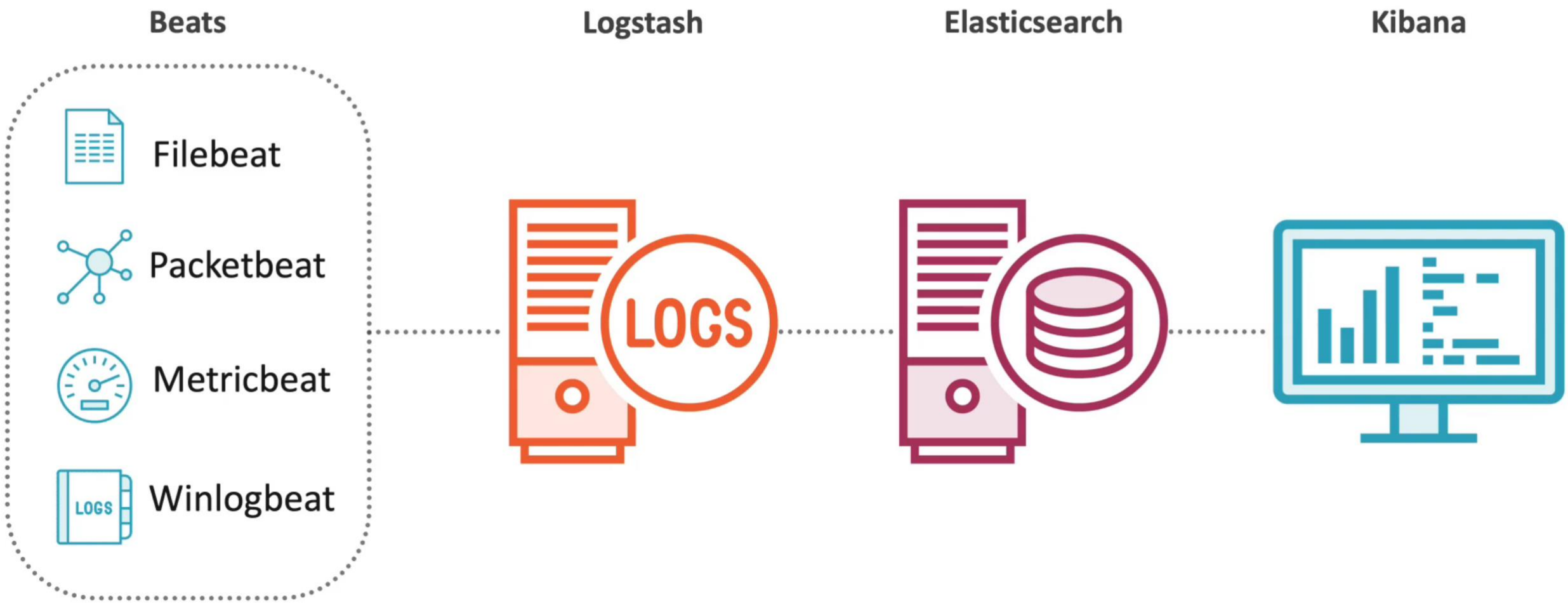
Collecting Data



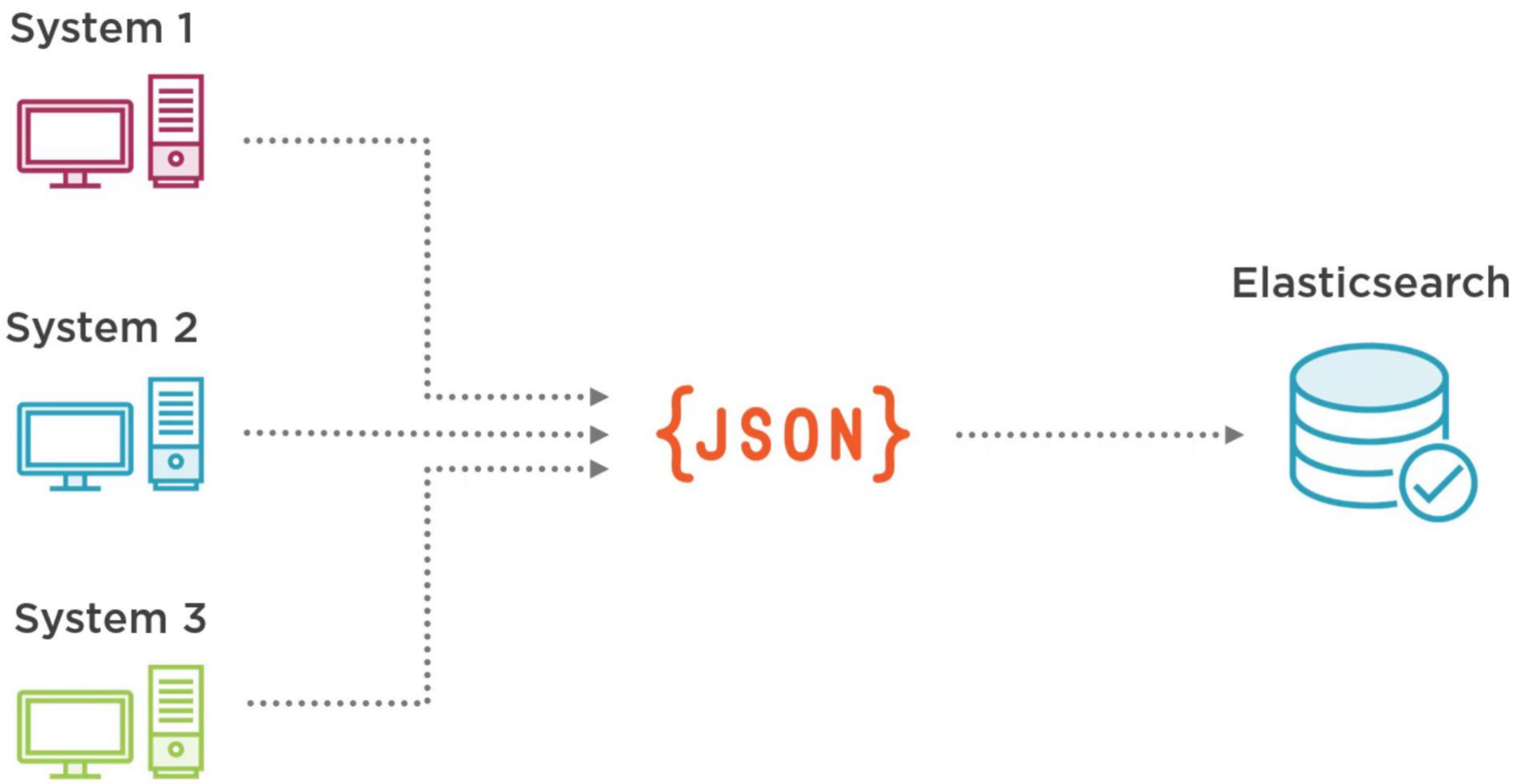

Email & Database
Applications


Email & Database
Applications

Building with Beats



Data Flow and Conversion



More use cases:

https://www.elastic.co/customers/?ultron=B-Stack-Trials-APJ-IND-Exact&gambit=Stack-UseCase&blade=adwords-s&hulk=paid&Device=c&thor=elasticsearch%20use%20cases&gclid=Cj0KCQiAyMKbBhD1ARIsANs7rEHS_YSpxcGpDWG19JNxXcRbjHBvwLJqQVx2bO6Wt8EzLk9zq9NVMegaAppFEALw_wcB

https://www.elastic.co/customers/success-stories?usecase=enterprise-search&industry=All

Apache Lucene

Full-text Search

Scaleable

Elasticsearch

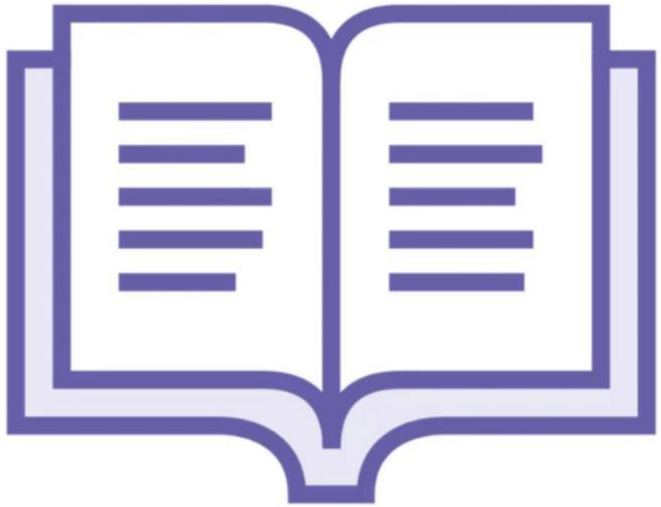
Lucene is a Java library. You can include it in your project and refer to its functions using function calls. Elasticsearch is built over Lucene and provides a JSON based REST API to refer to Lucene features. Elasticsearch provides a distributed system on top of Lucene

Data is stored as documents!

```
{
name: "Clementines(3lb bag)",
category: "Fruits",
brand: "Cuties",
price: "$4.29",
 }
```

I am a document, a JSON object that is stored in Elasticsearch under a unique ID!

Apache Lucene



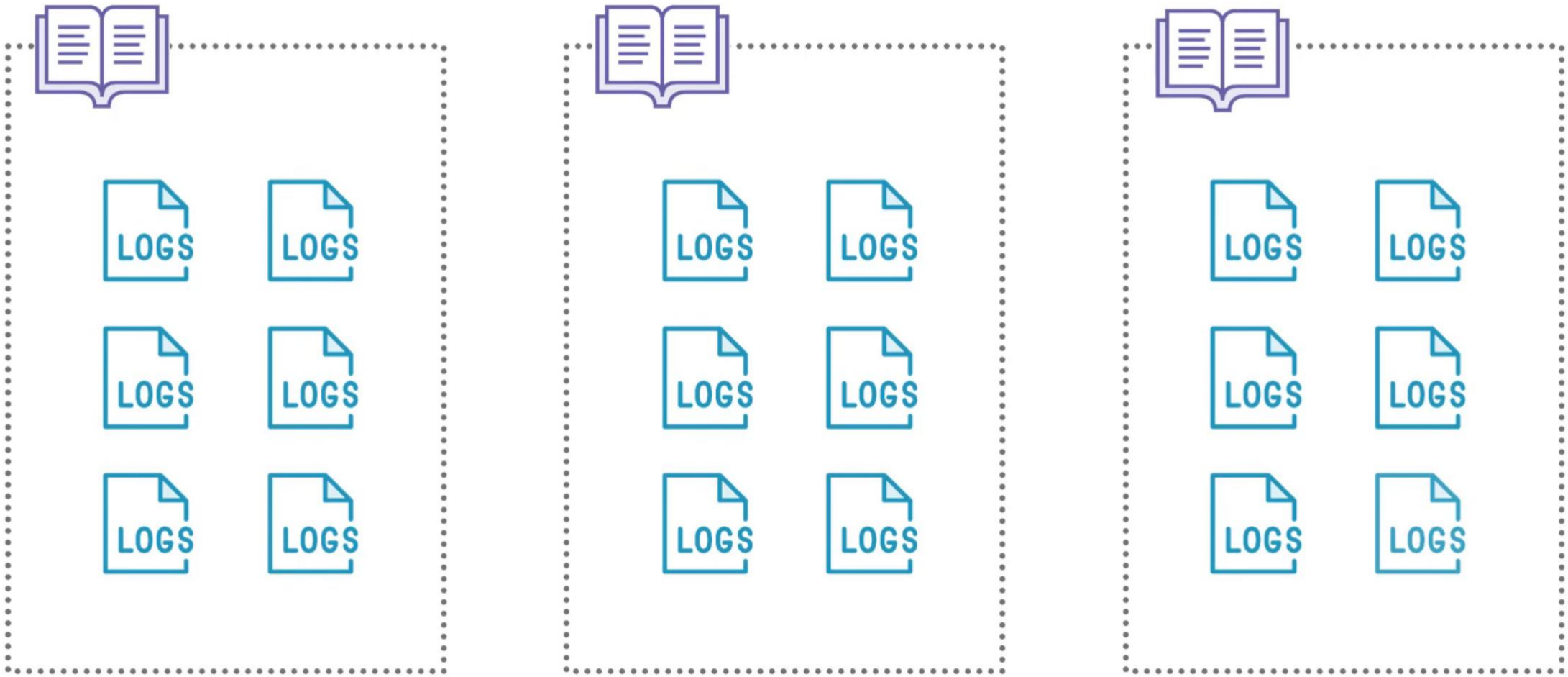
Inverted Index

Like a book index; fast searches!

Document

Event, network, metric, or security logs

Apache Lucene



Documents and Fields

Windows log 1

- @timestamp: [time]
- source.ip: 192.168.2.10
- log.level: ...

Suricata log 1

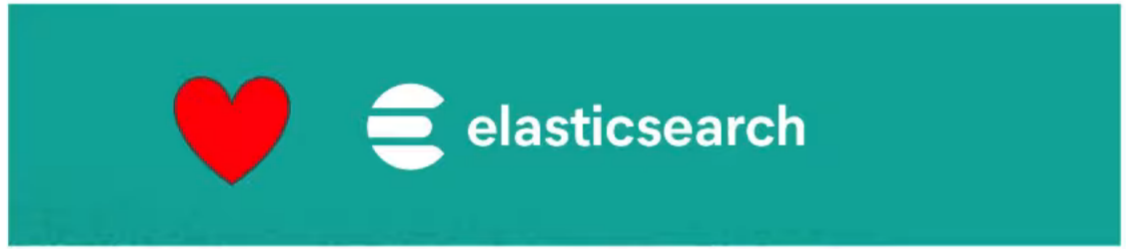
- @timestamp: [time]
- source.ip: 192.168.2.10
- event.id: ...

Zeek log 1

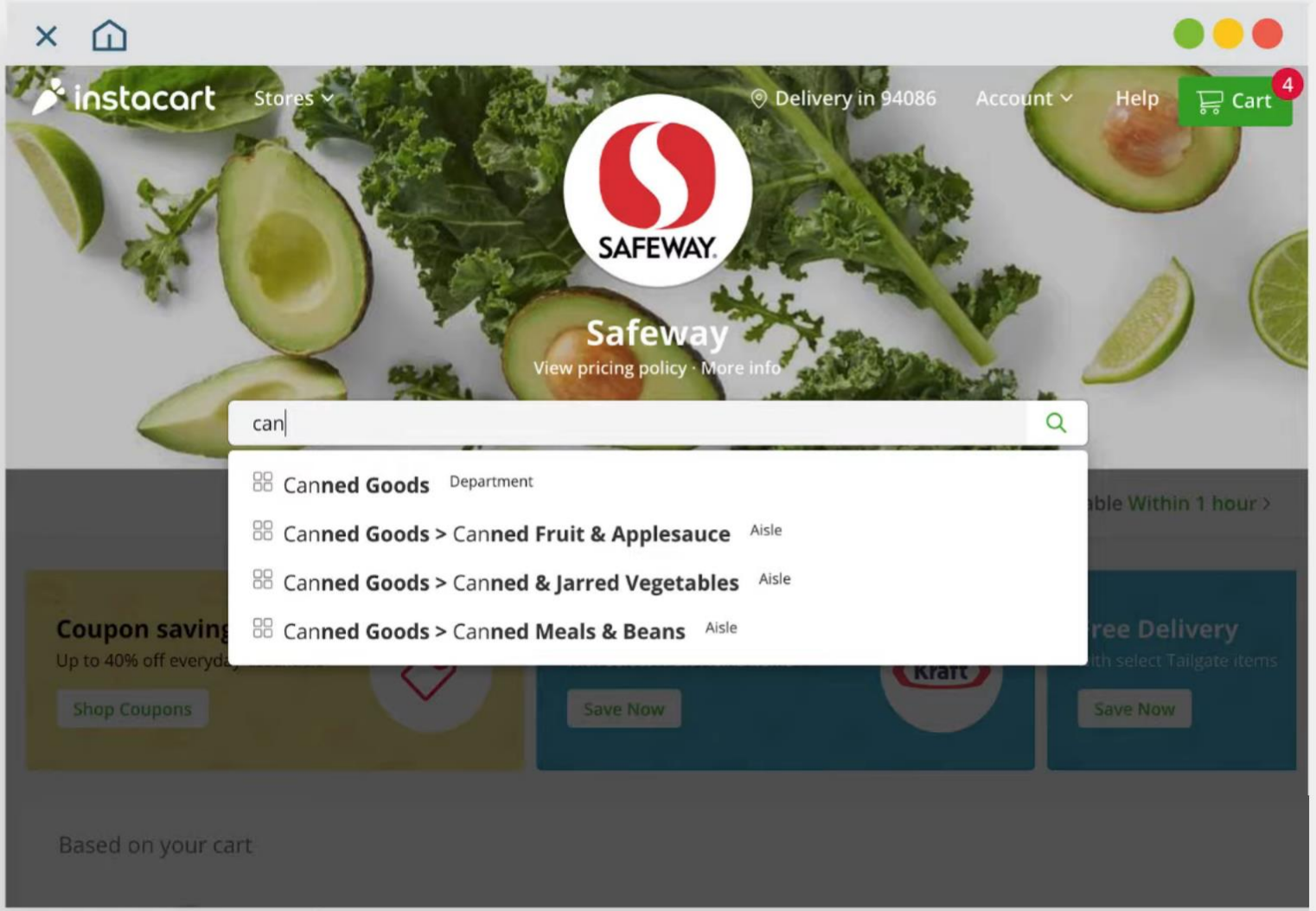
- @timestamp: [time]
- source.ip: 192.168.2.10
- conn.duration: ...

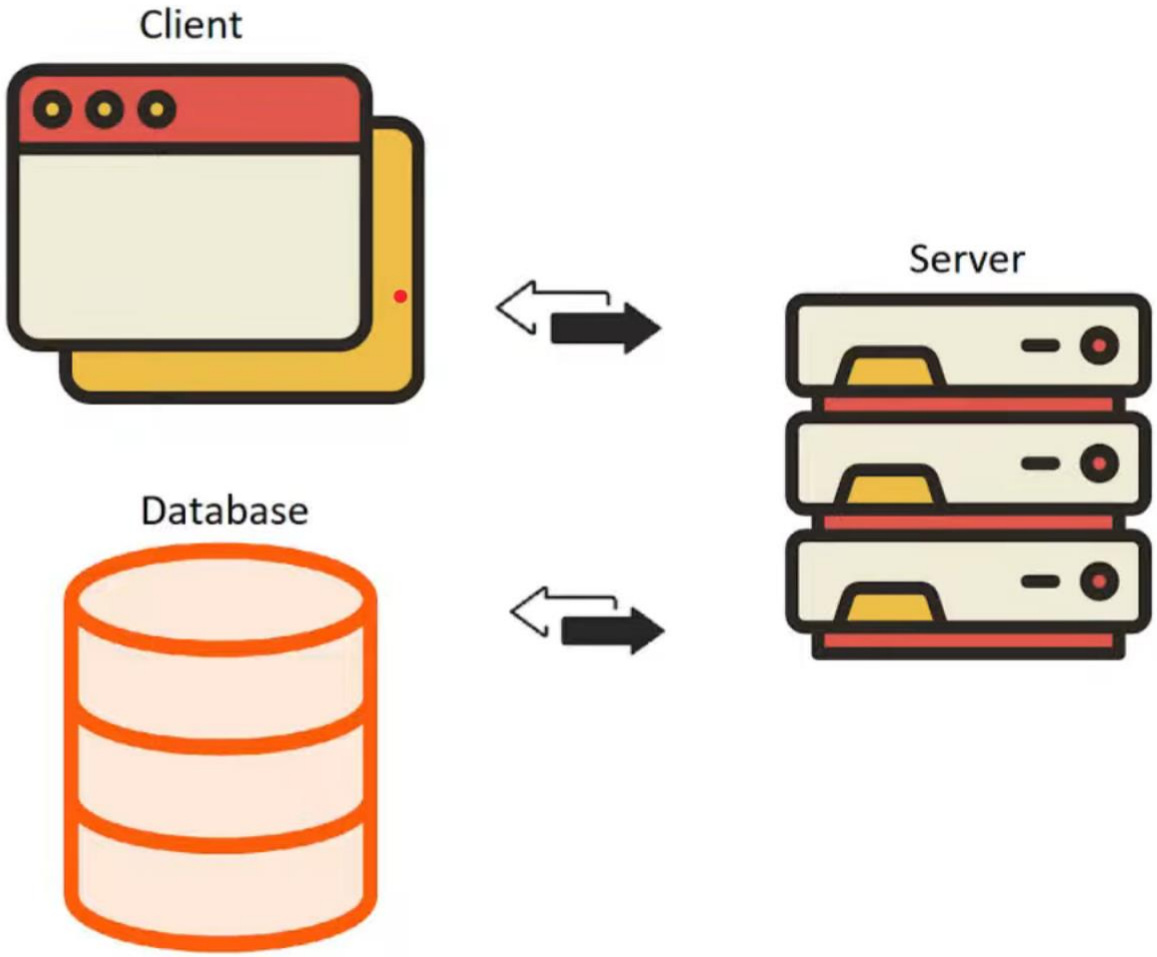
Elasticsearch

Store | Search | Analyze



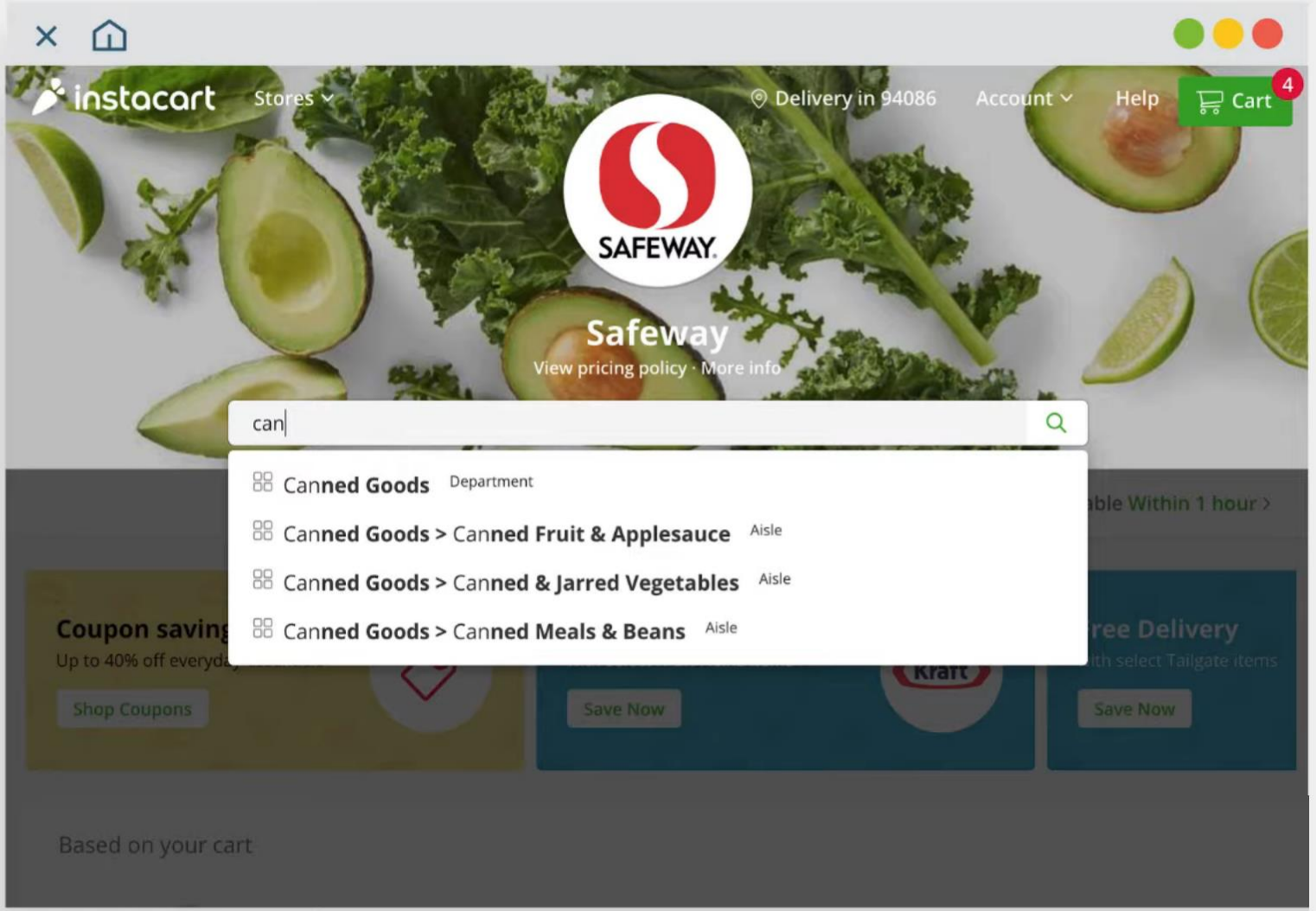
instacart


instacart
Stores
Delivery in 94086
Account
Help
Cart
4
SAFEWAY
Safeway
View pricing policy · More info
can
Canned Goods Department
Canned Goods > Canned Fruit & Applesauce Aisle
Canned Goods > Canned & Jarred Vegetables Aisle
Canned Goods > Canned Meals & Beans Aisle
Coupon saving
Up to 40% off everyda
Shop Coupons
Save Now
Kraft
Save Now
Based on your cart


Client
Server
Database

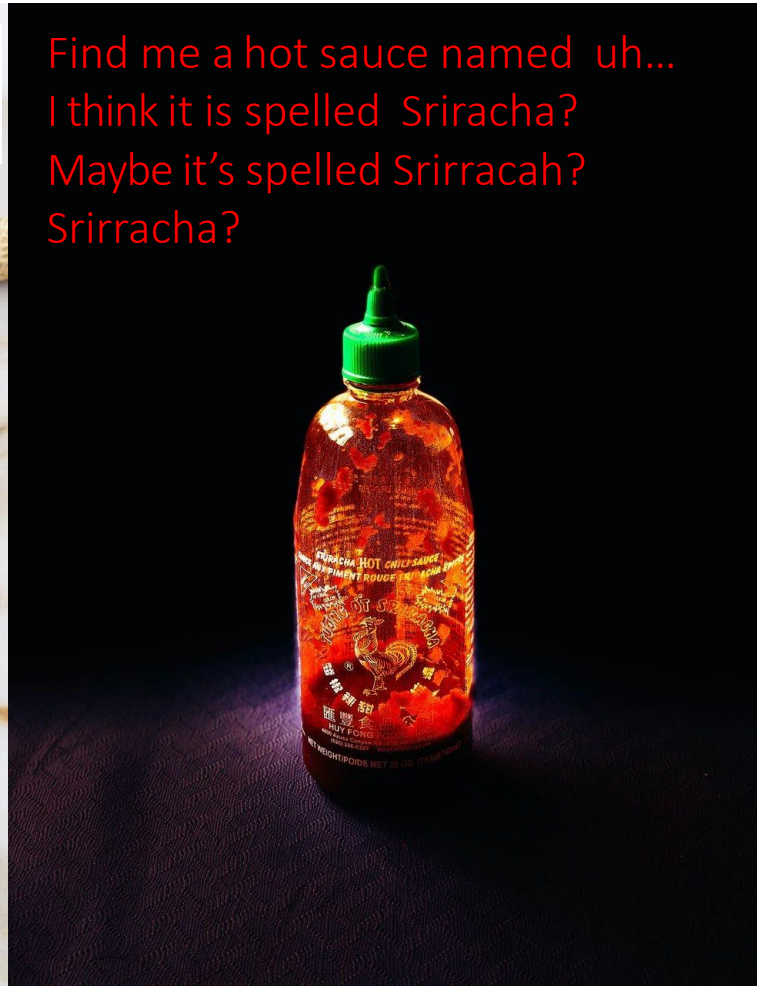
Great Search Experience = Get fast and relevant results, no matter the scale.

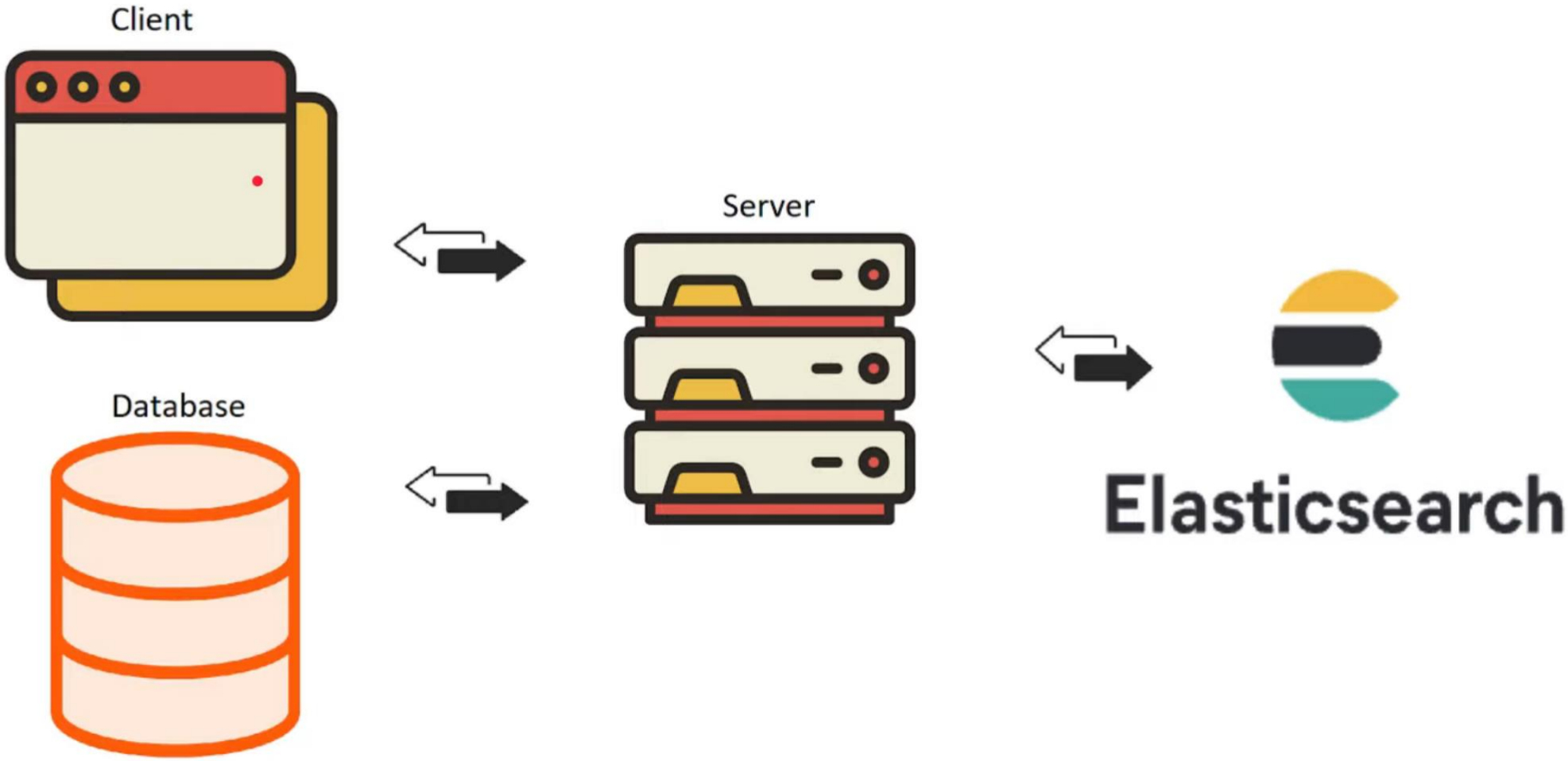
instacart



Find me a list of peanut butter brands. I want the highest rated brands at the top.

Find me a hot sauce named uh…
I think it is spelled Sriracha?
Maybe it's spelled Srirracah?
Srirracha?




Client
Server
Database
Elasticsearch

Document grouped into an index!

Produce Index

```
{
name: "Baby Carrots(1lb bag)",
category: "Vegetables",
brand: "365",
price: "$0.99",
 }
```

```
{
name: "Clementines(3lb bag)",
category: "Fruits",
brand: "Cuties",
price: "$4.29",
 }
```

Wine & Beer Index

```
{
name: "Unanime Malbec(750ml)",
brand: "Mascota Vineyards",
country/state: "Argentina",
region: "Mendoza"
wine_type: "Red Wine",
ABV: "14%",
price: "$22.99",
 }
```

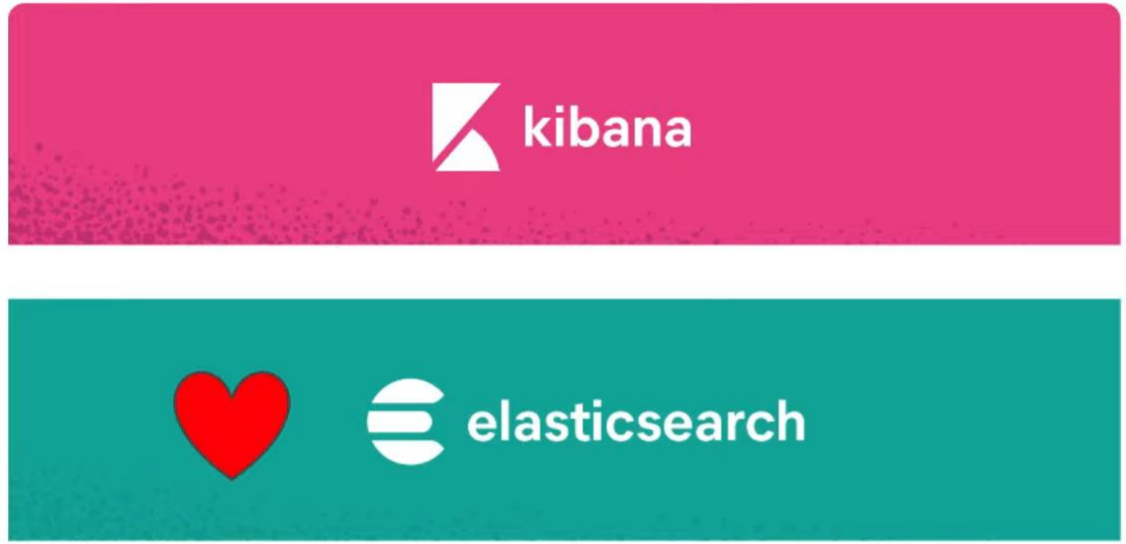
```
{
name: "Hazy Little Thing IPA(750ml)",
country: "US",
state: "California",
beer_type: "Ale",
beer_style: "India Pale Ale"
ABV: "6.7%",
price: "$14.99",
 }
```

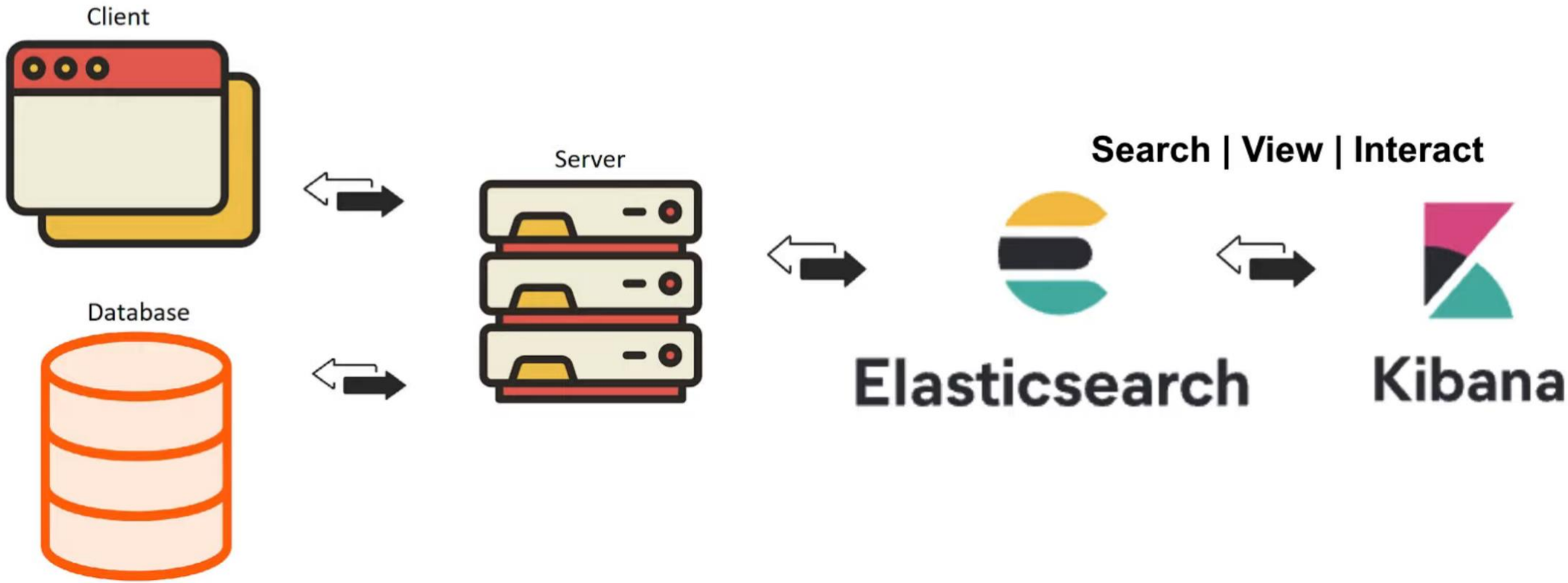
Kibana

Visualize & Manage

Elasticsearch

Store | Search | Analyze



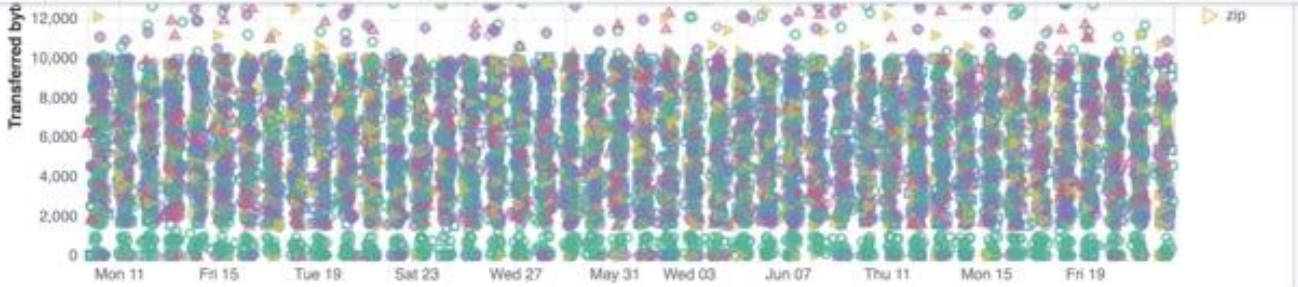

Client
Database
Server
Search | View | Interact
Elasticsearch
Kibana

Elastic

Search Elastic

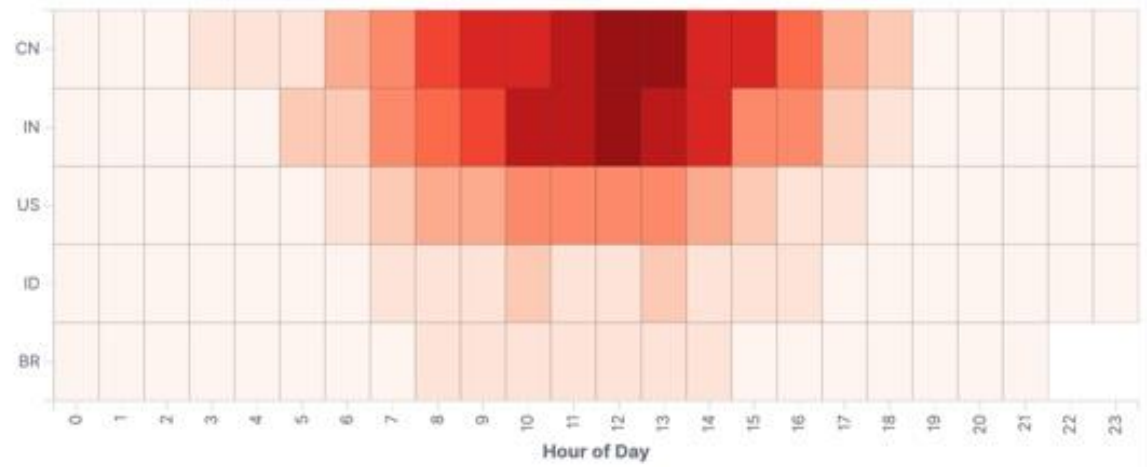
Dashboard / [Logs] Web Traffic 2

Full screen Share Clone Edit



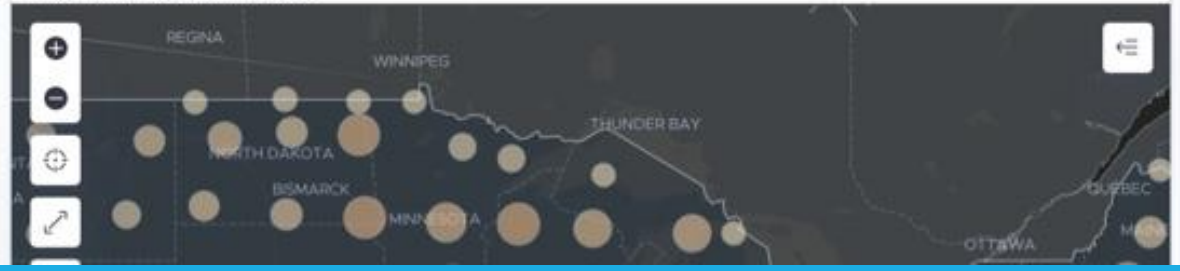
css	11.8MB	19.7KB	2,218 ↓	5 ↓
zip	9.8MB	6.4KB	1,644 ↓	2 ↑
deb	9.6MB	35.4KB	1,574 ↓	6 ↑
rpm	3.3MB	0B	554 ↓	0 ↓

[Logs] Heatmap



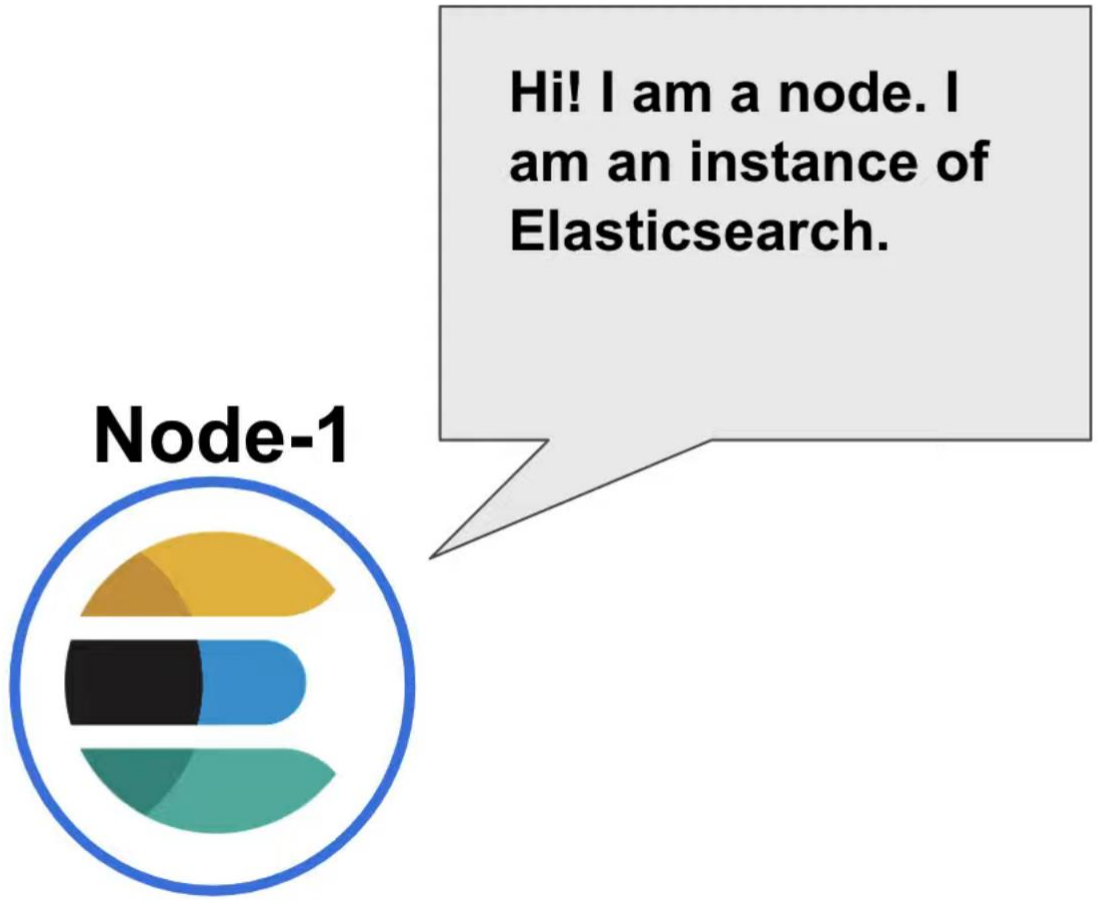
Records: Geo & Tags

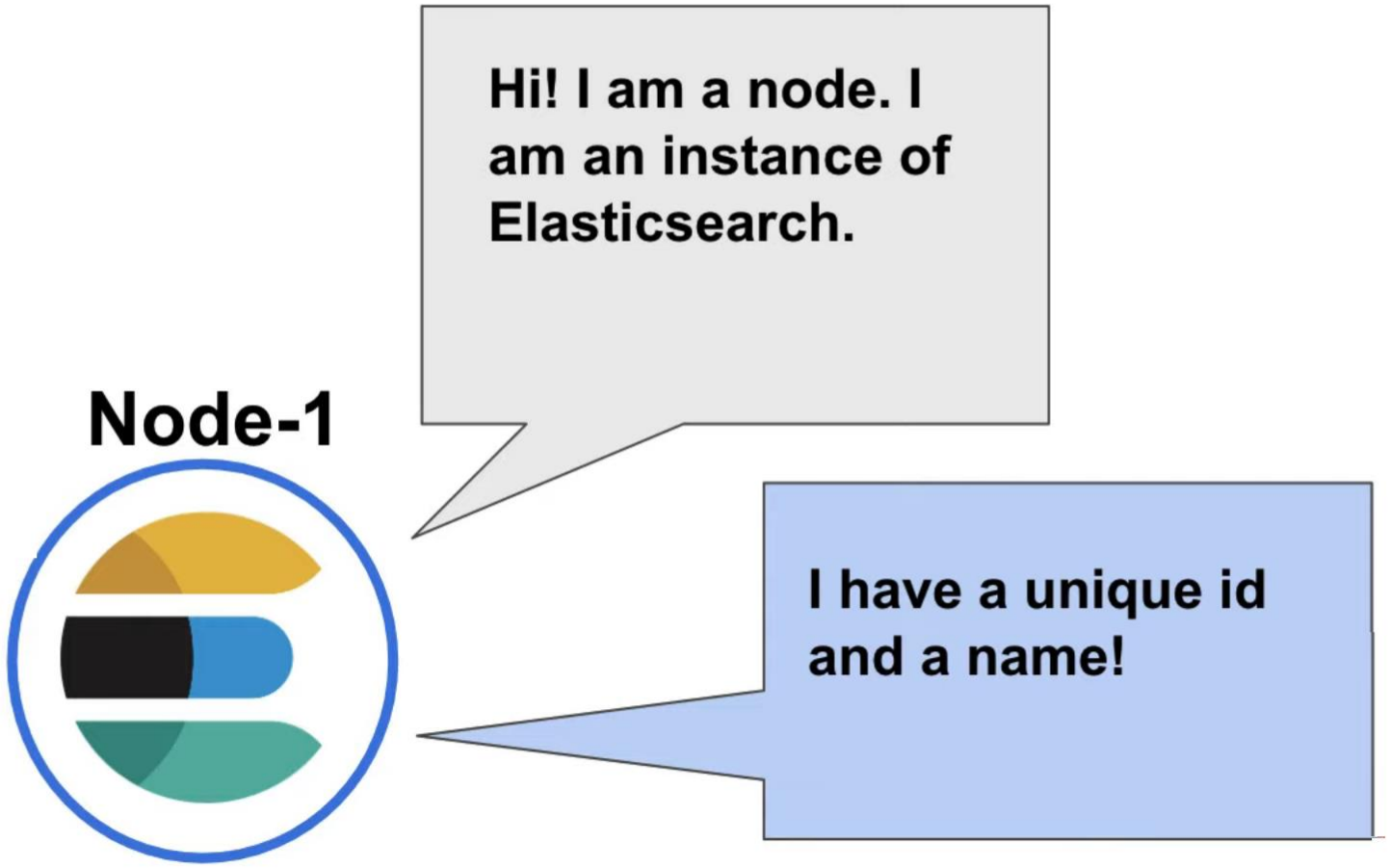
[Logs] Total Requests and Bytes

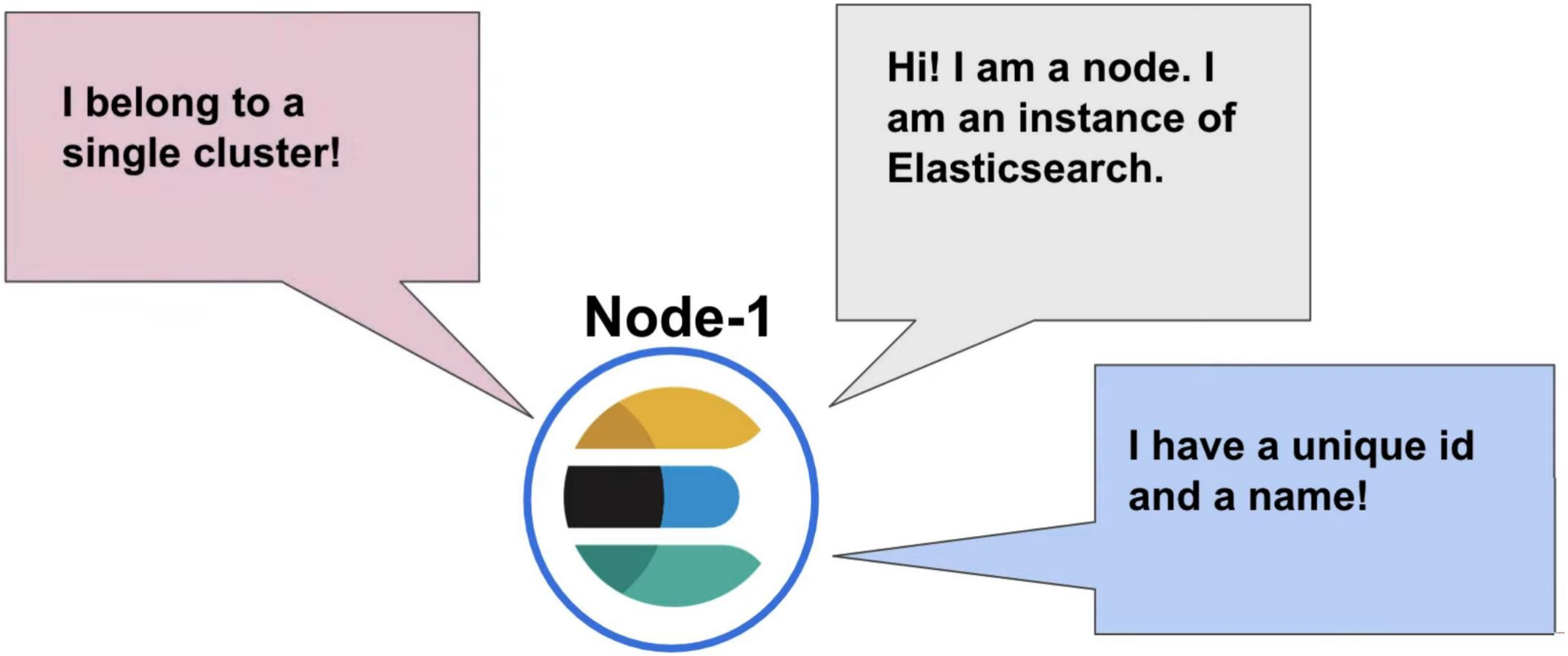


[Logs] Source and Destination Sankey Chart

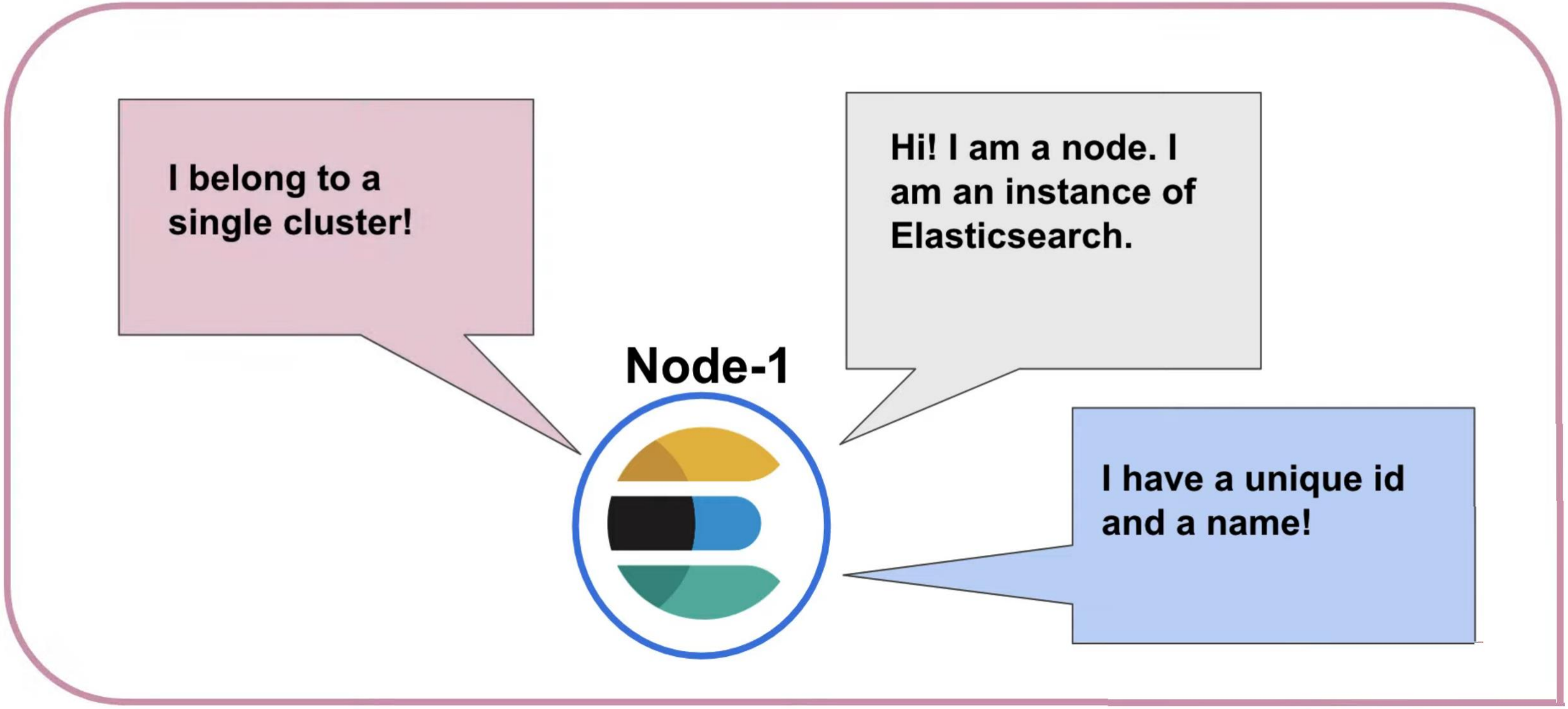
Node-1


Hi! I am a node. I am an instance of Elasticsearch.
Node-1

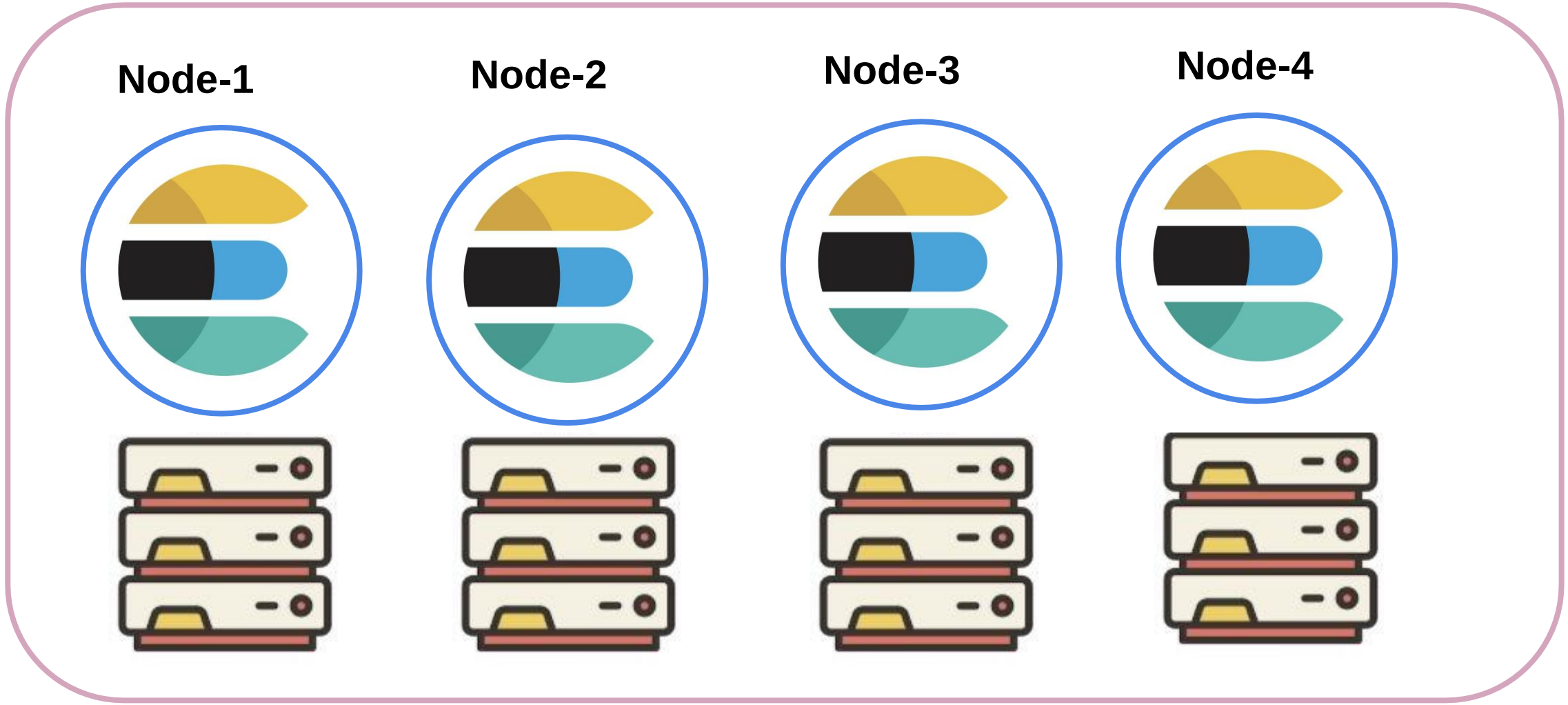

Hi! I am a node. I am an instance of Elasticsearch.
Node-1
I have a unique id and a name!


I belong to a single cluster!
Hi! I am a node. I am an instance of Elasticsearch.
Node-1
I have a unique id and a name!

Cluster


I belong to a single cluster!
Hi! I am a node. I am an instance of Elasticsearch.
Node-1
I have a unique id and a name!

Cluster



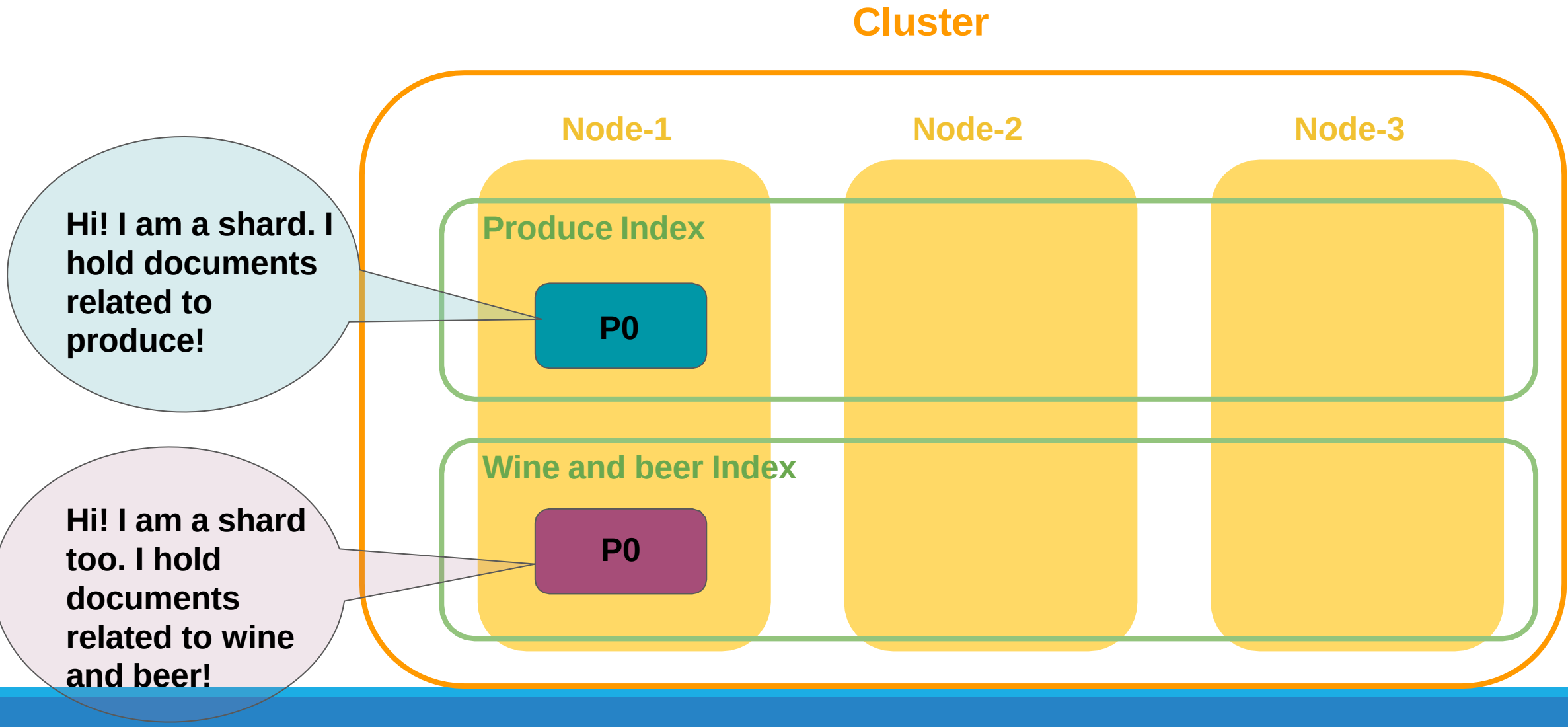
Produce Index


```
{
  "name": "Baby Carrots(1lb bag)",
  "category": "Vegetables",
  "brand": "365",
  "price": "$0.99"
}

{
  "name": "Clementines(3lb bag)",
  "category": "Fruits",
  "brand": "Cuties",
  "price": "$4.29"
}
```

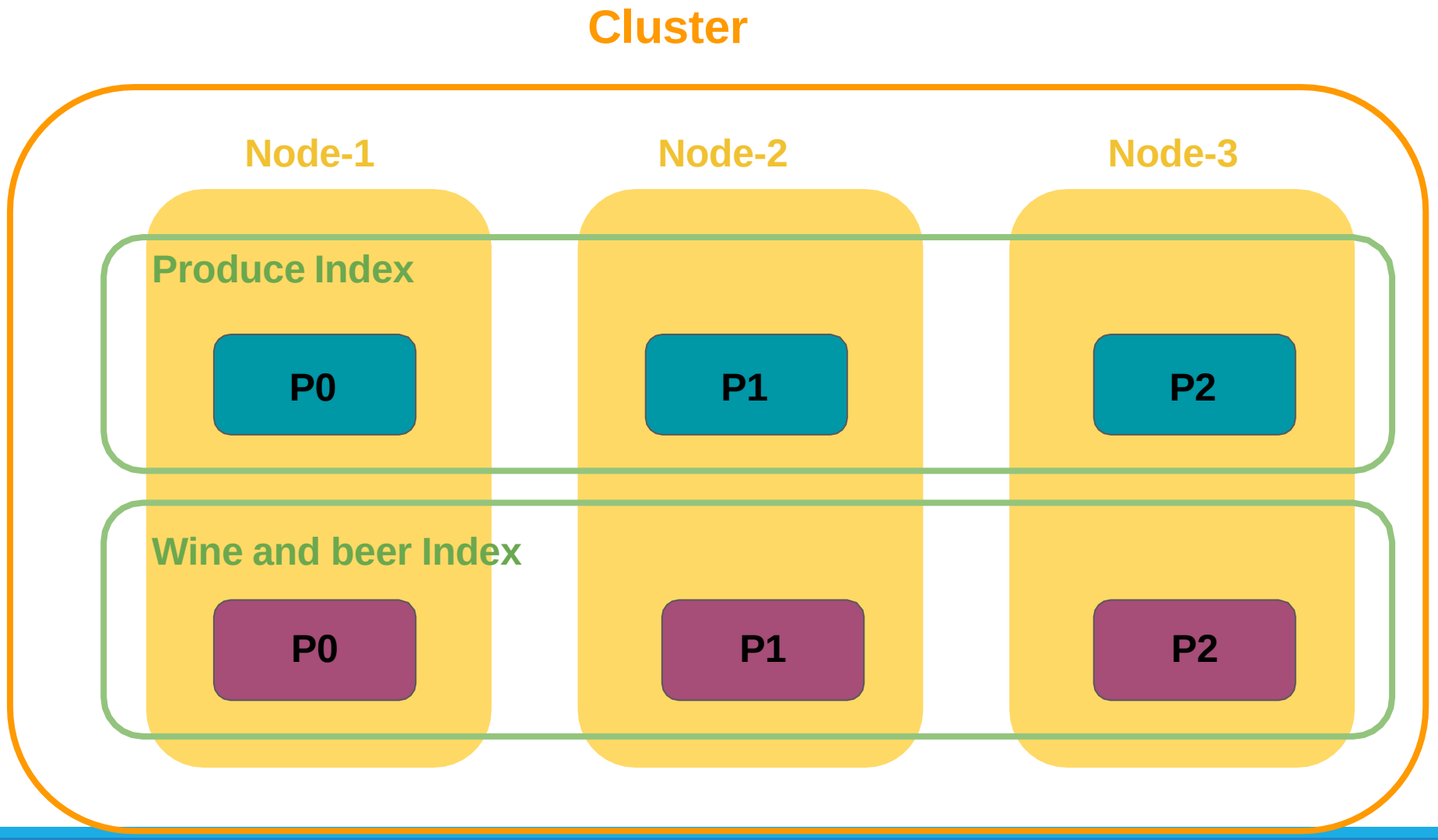

Wine & Beer Index


```
{
  "name": "Unanime Malbec(750ml)",
  "brand": "Mascota Vineyards",
  "country": "Argentina",
  "region": "Mendoza",
  "wine_type": "Red Wine",
  "ABV": "14%",
  "price": "$22.99"
}

{
  "name": "Hazy Little Thing IPA",
  "brand": "US",
  "country": "California",
  "beer_type": "Ale",
  "beer_style": "India Pale Ale",
  "ABV": "6.7%",
  "price": "$14.99"
}
```

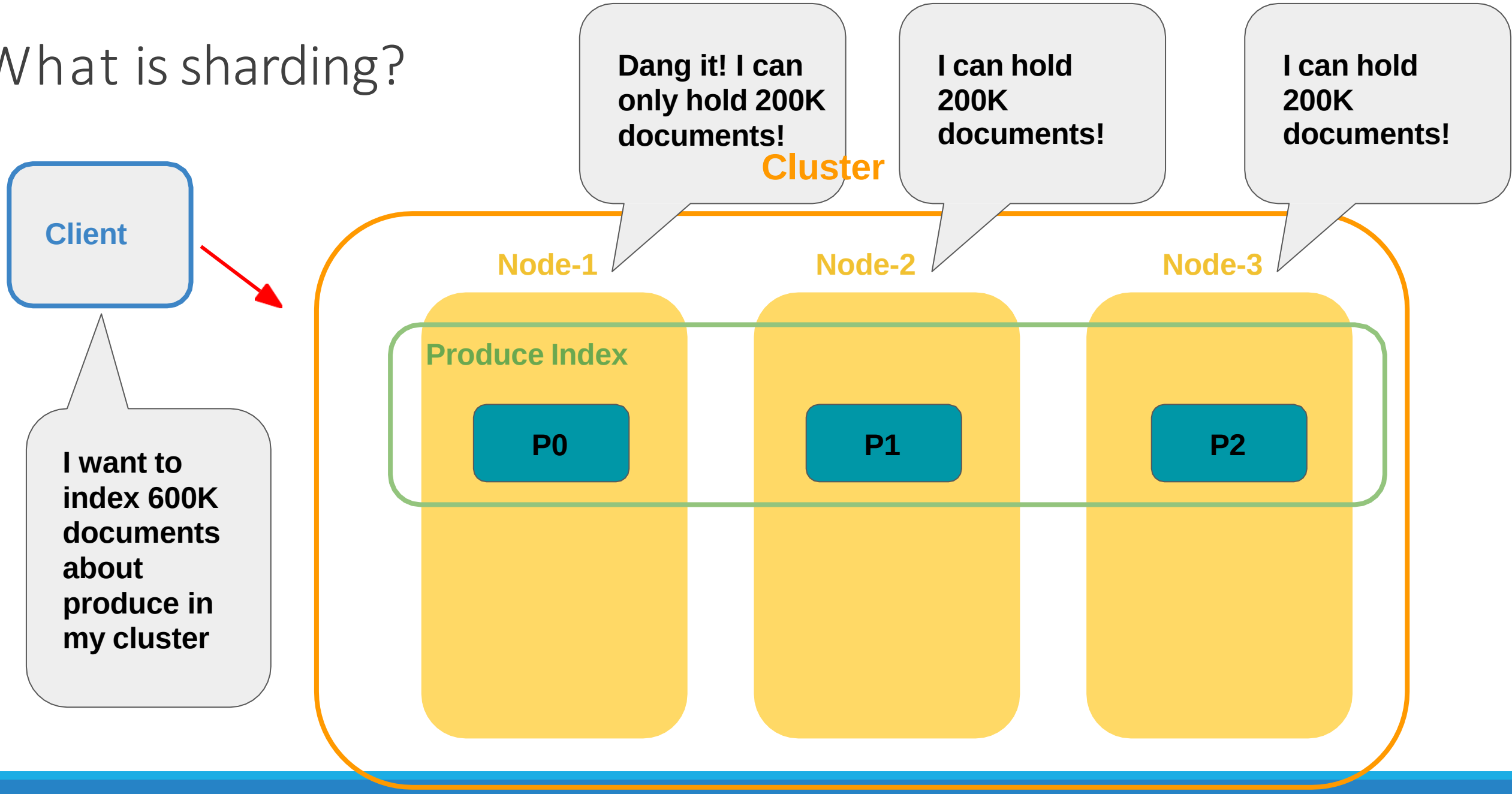
What is a shard?



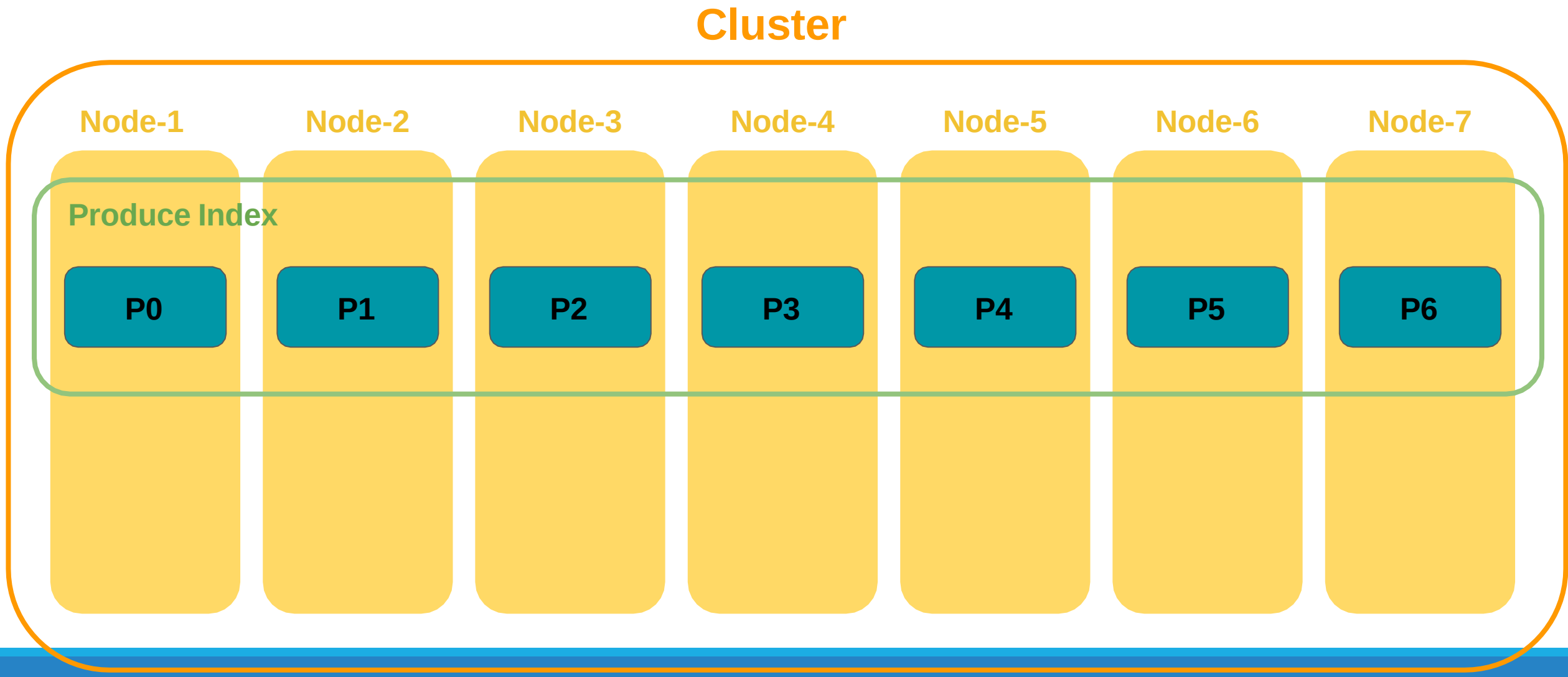
What is sharding?



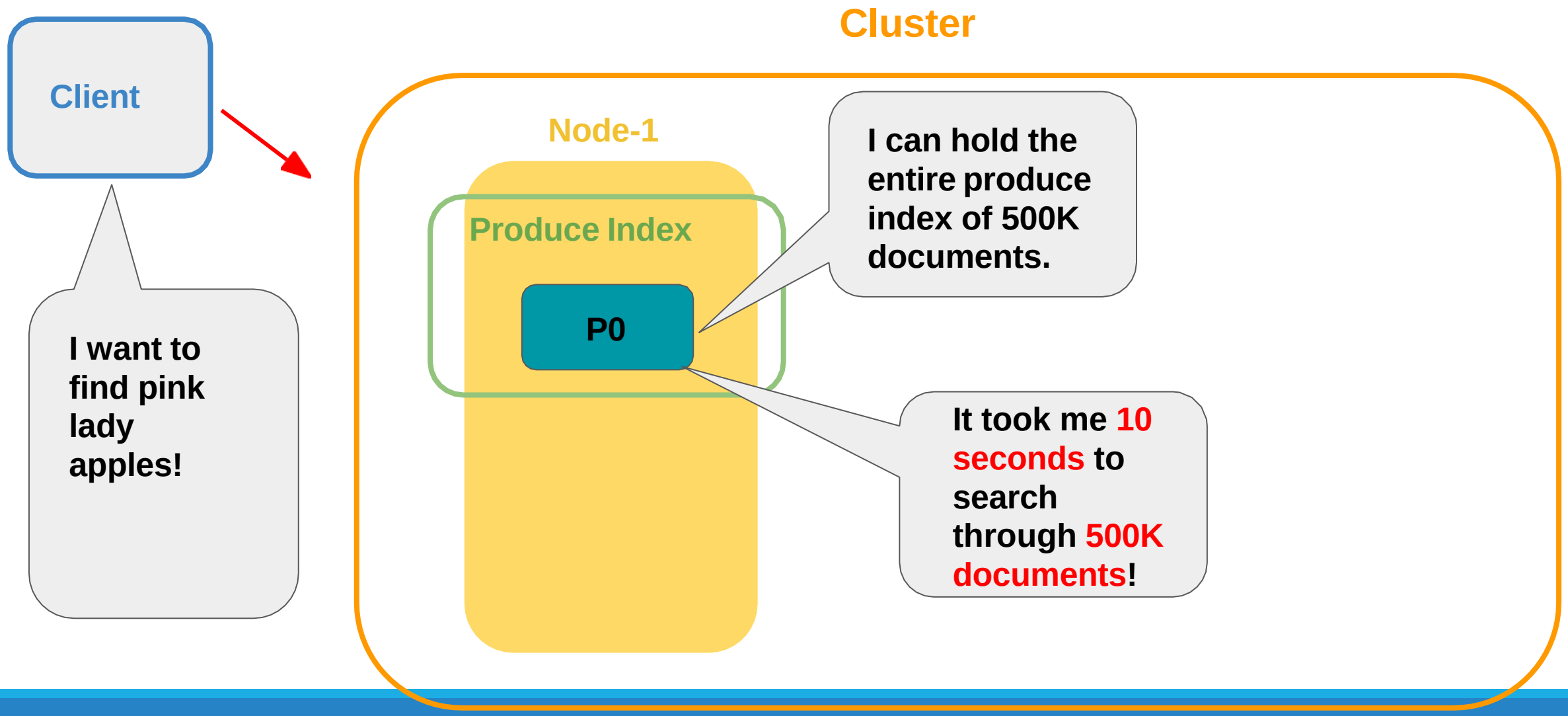
What is sharding?



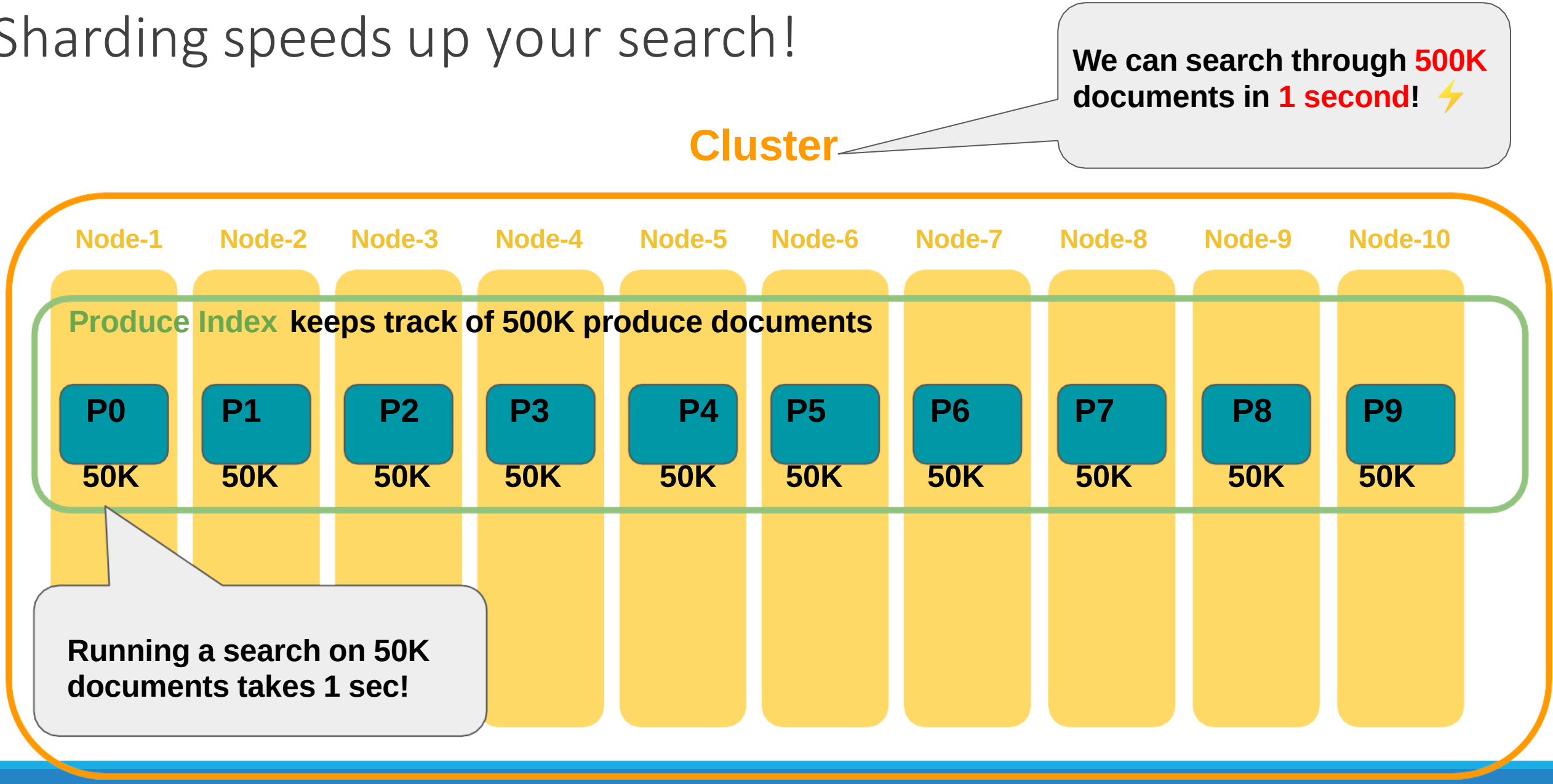
What is sharding?



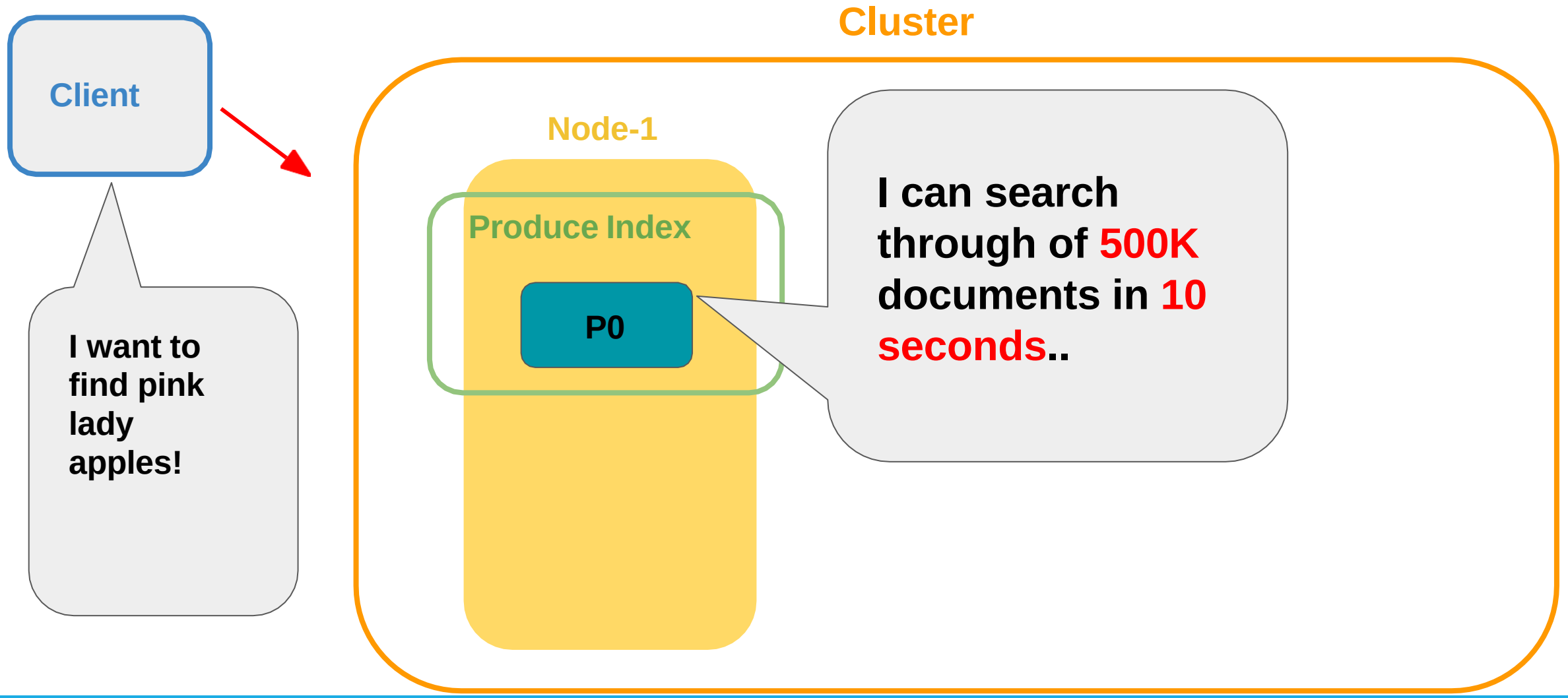
What is sharding?



Sharding speeds up your search!



What is sharding?



Sharding speeds up your search!

We can search through 500K documents in 1 second! ⚡

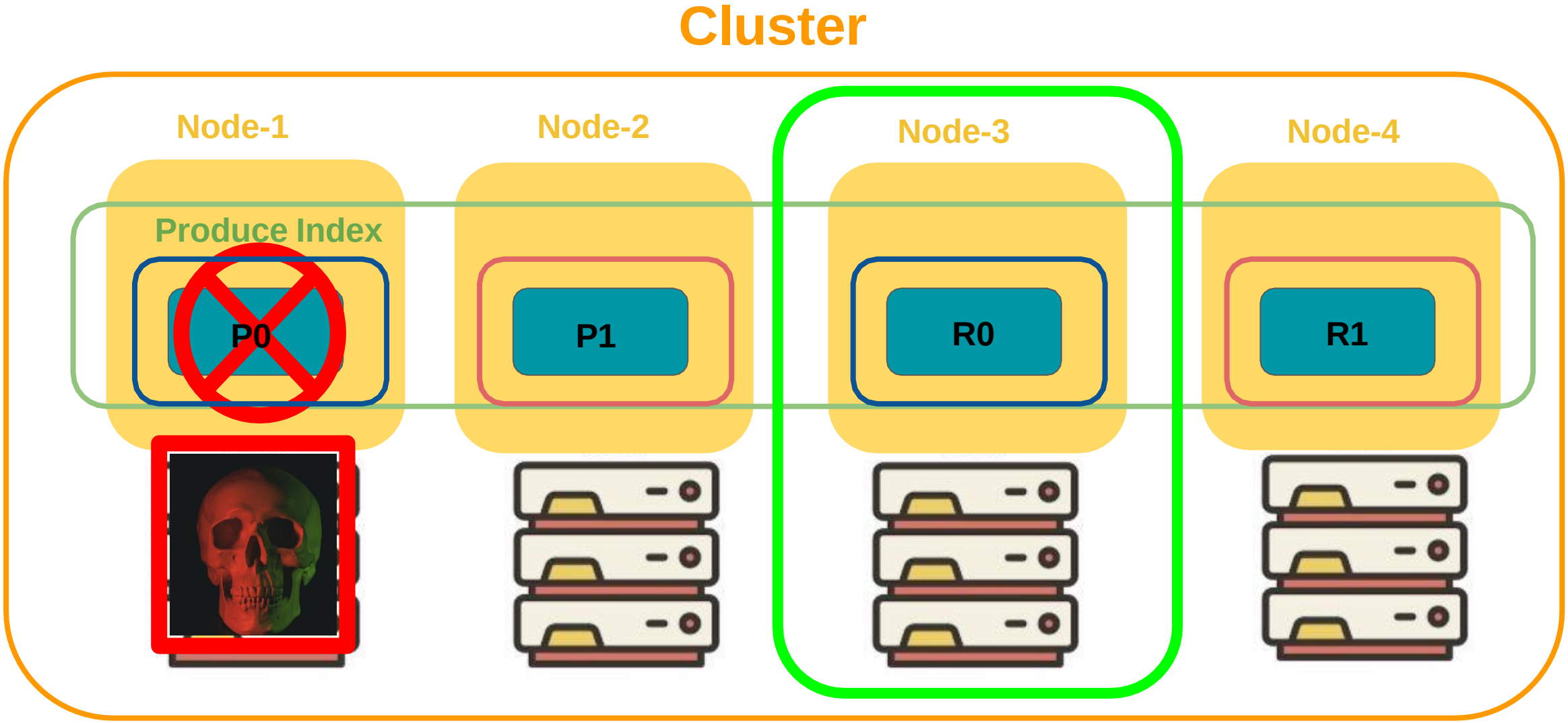
Cluster

Node-1 Node-2 Node-3 Node-4 Node-5 Node-6 Node-7 Node-8 Node-9 Node-10

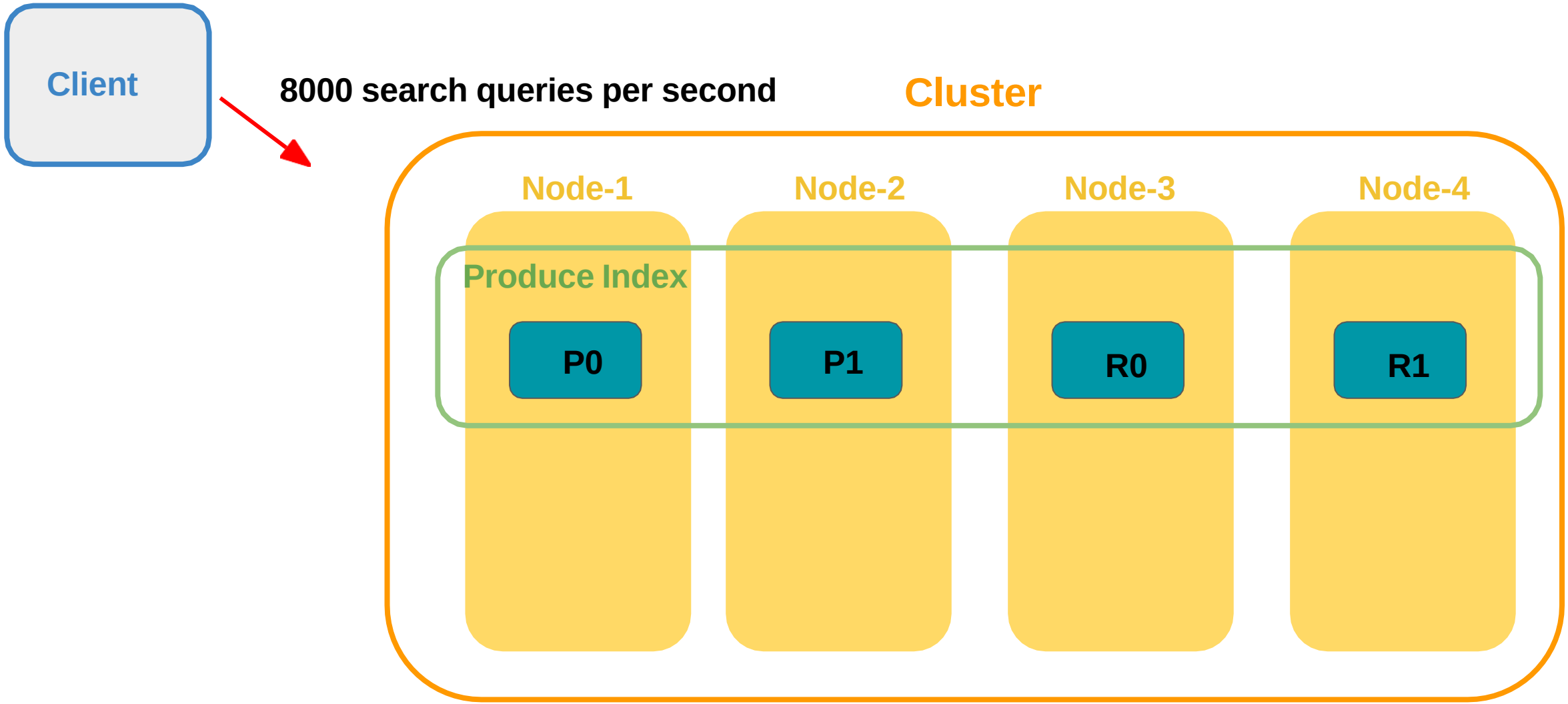
Produce Index

P0 50K | P1 50K | P2 50K | P3 50K | P4 50K | P5 50K | P6 50K | P7 50K | P8 50K | P9 50K

What are replica shards?



Replica shards can improve the performance of your search



Near Realtime Search

Very low latency, ~1 second from the time a document is indexed until it becomes searchable

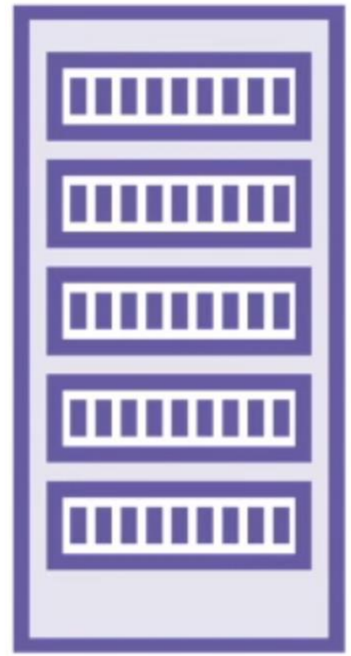
Node

Single server

Performs indexing

Allows search

Has a unique id and name



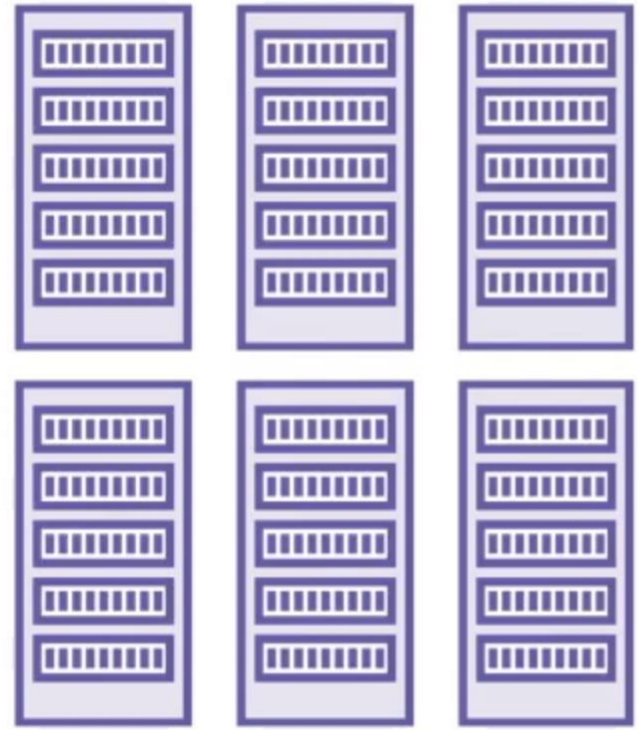
Collection of nodes

Holds the entire indexed data

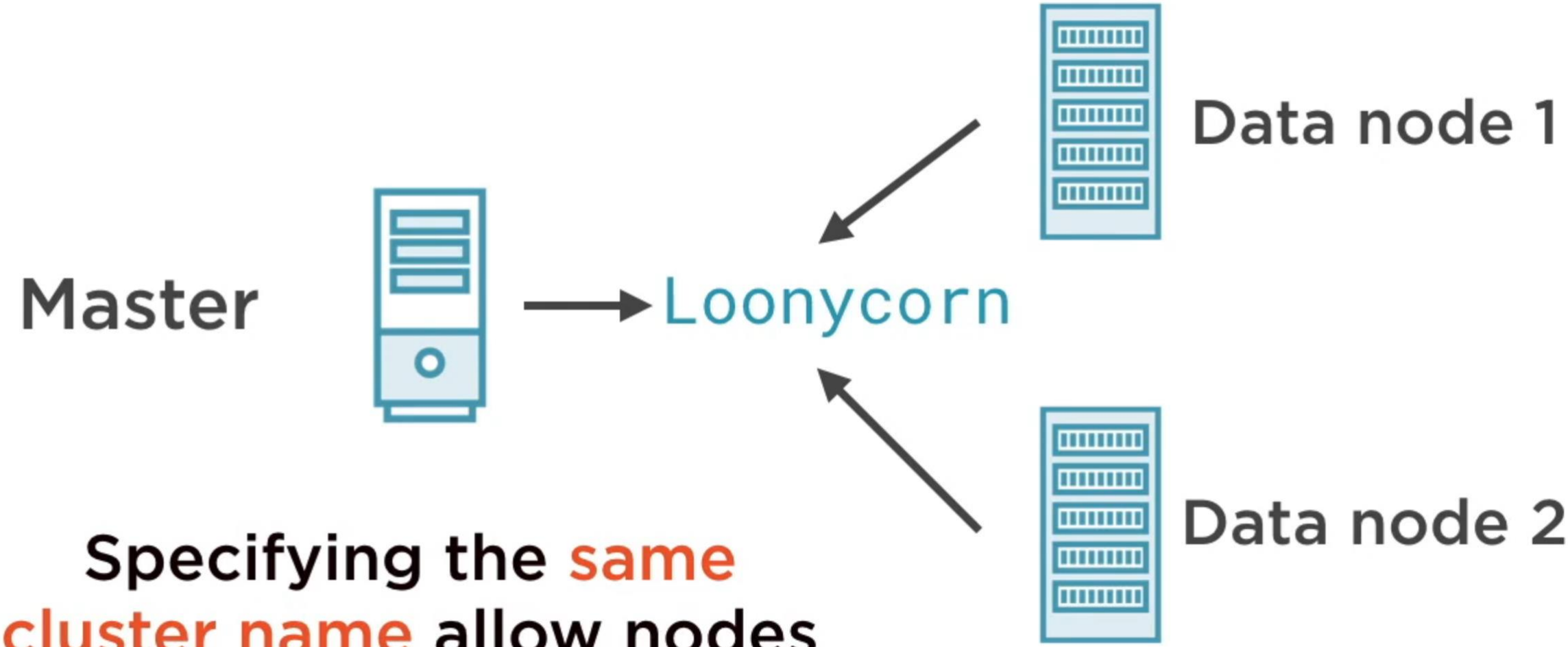
Has a unique name

Nodes join a cluster using the cluster name

Cluster

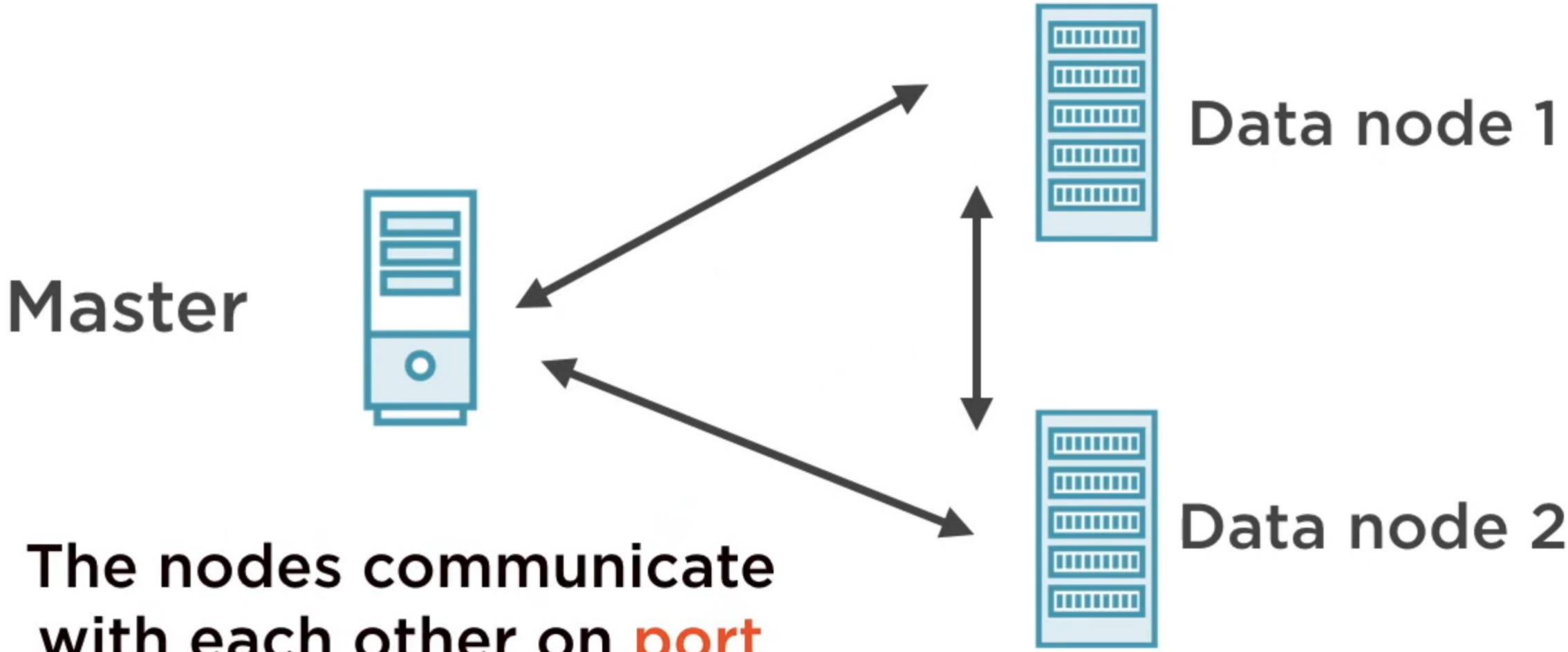


Elasticsearch Cluster



Specifying the same cluster name allow nodes to join the same cluster

Elasticsearch Cluster



The nodes communicate with each other on port 9300 to discover each other

Elasticsearch Cluster

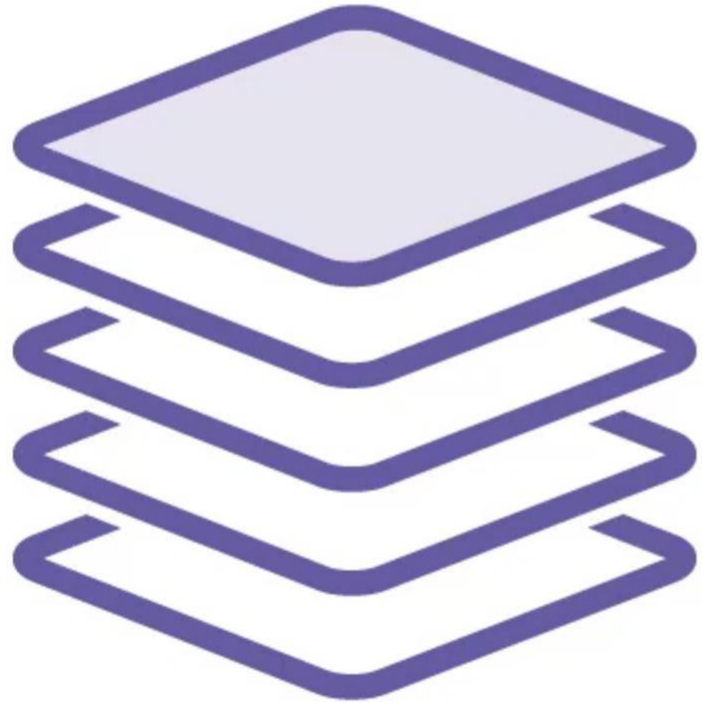
The cluster exposes HTTP endpoints at port 9200 for external communication

Document

Basic unit of information to be indexed

Expressed in JSON

Resides within an index



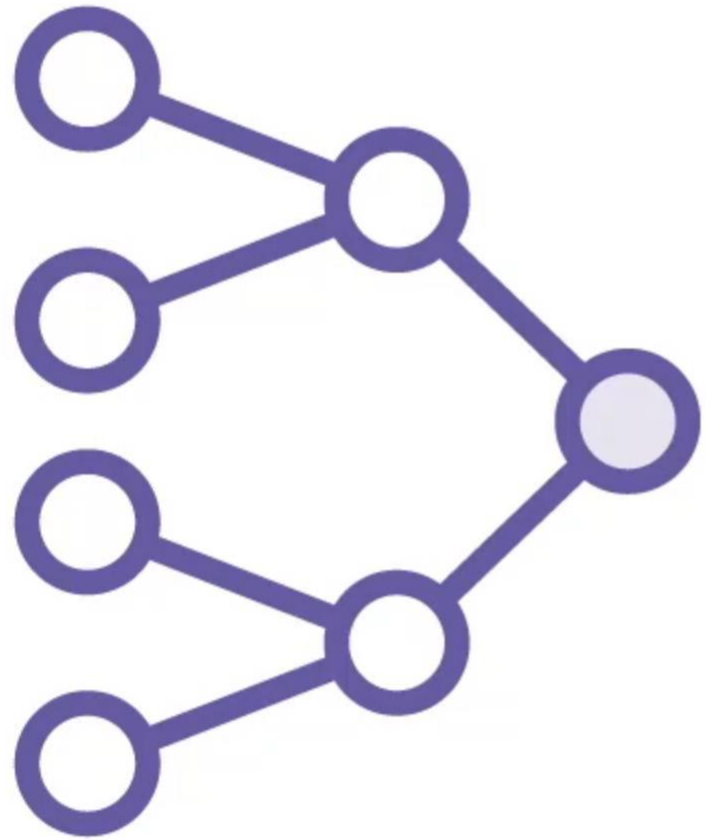
Index

Collection of similar documents

Identified by name

Any number of indices in a cluster

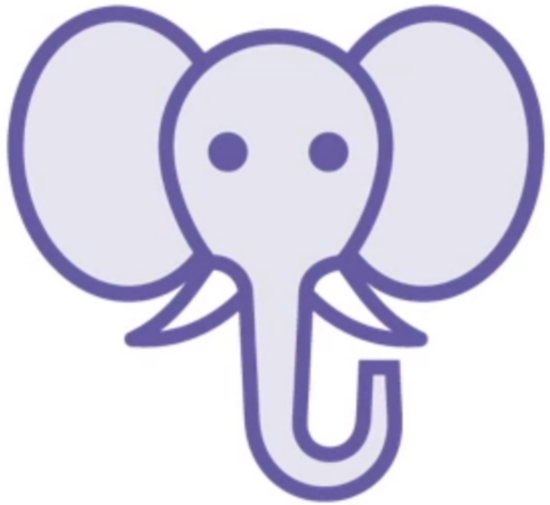
Different indices for different logical groupings



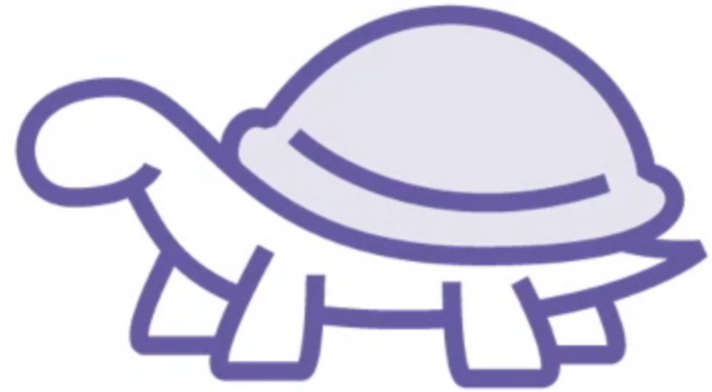
Type: Logical category of a document within an index

Multiple types in one index deprecated in 6.0.0

Documents in an Index

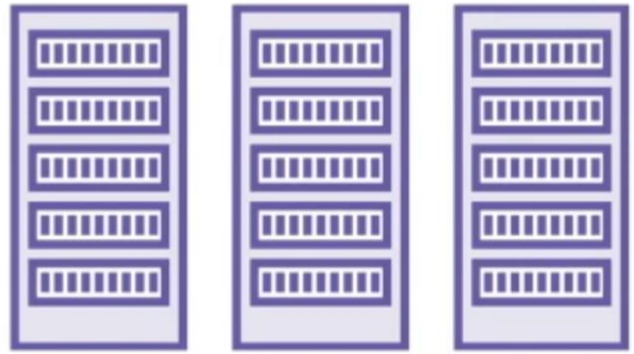


Too large to fit in the hard disk of one node



Too slow to serve all search requests from one node

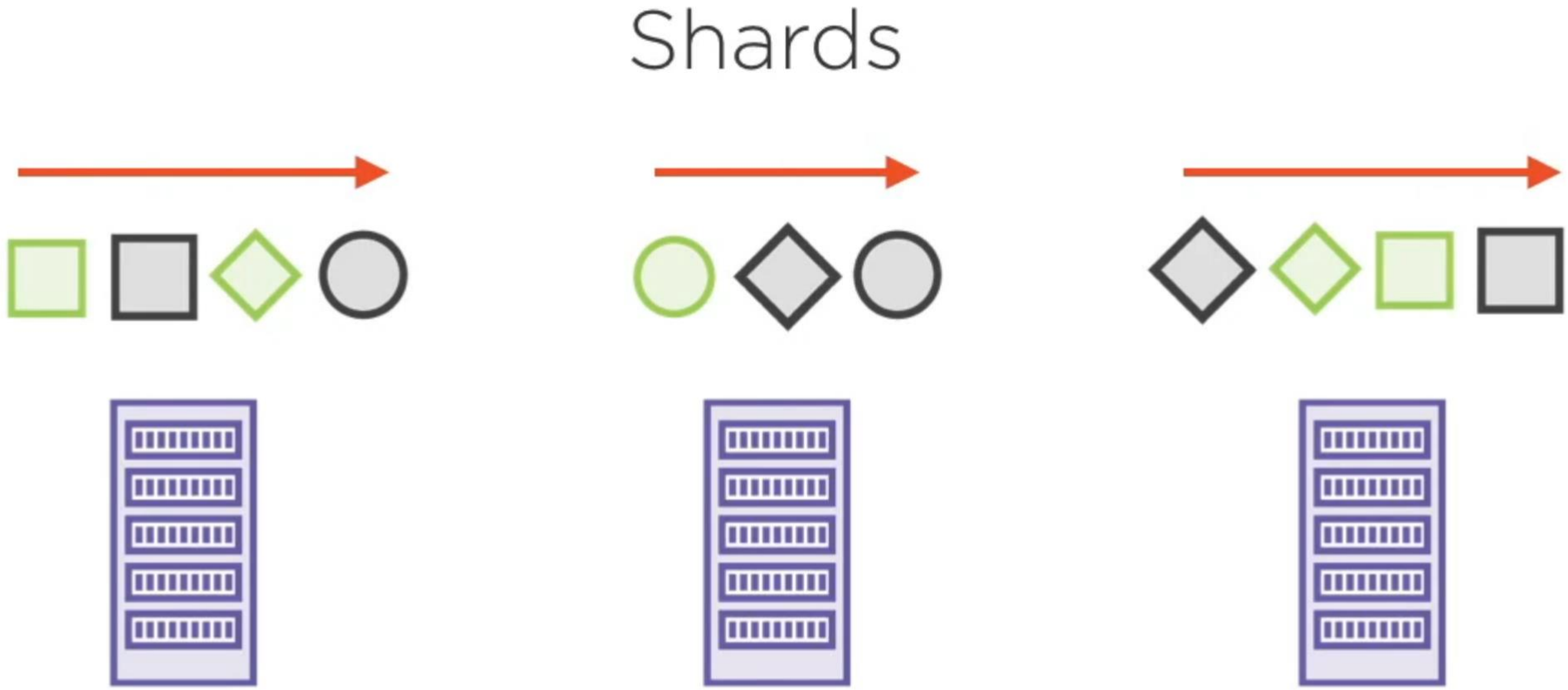
Shards



Split the index across multiple nodes in the cluster

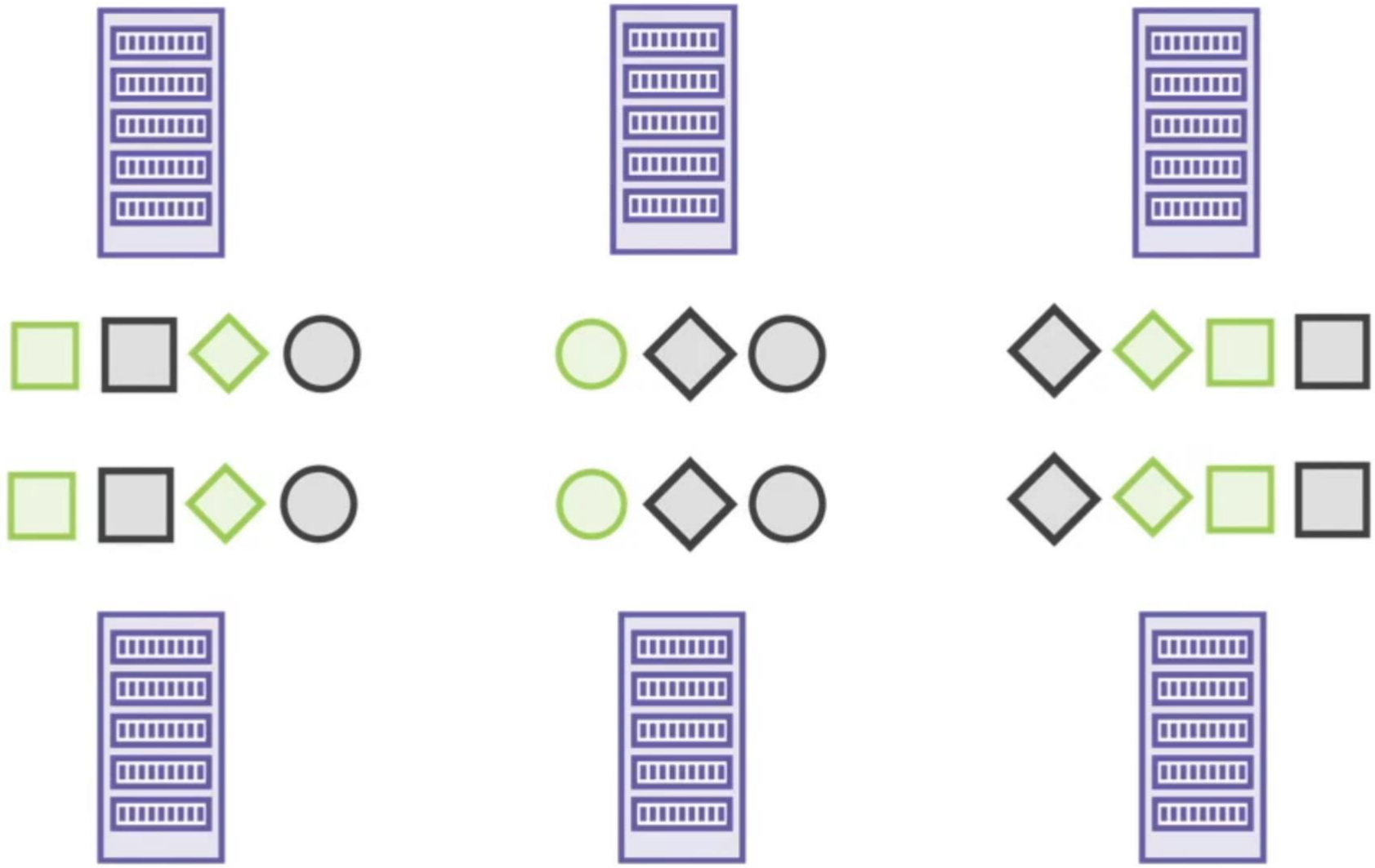
Shards

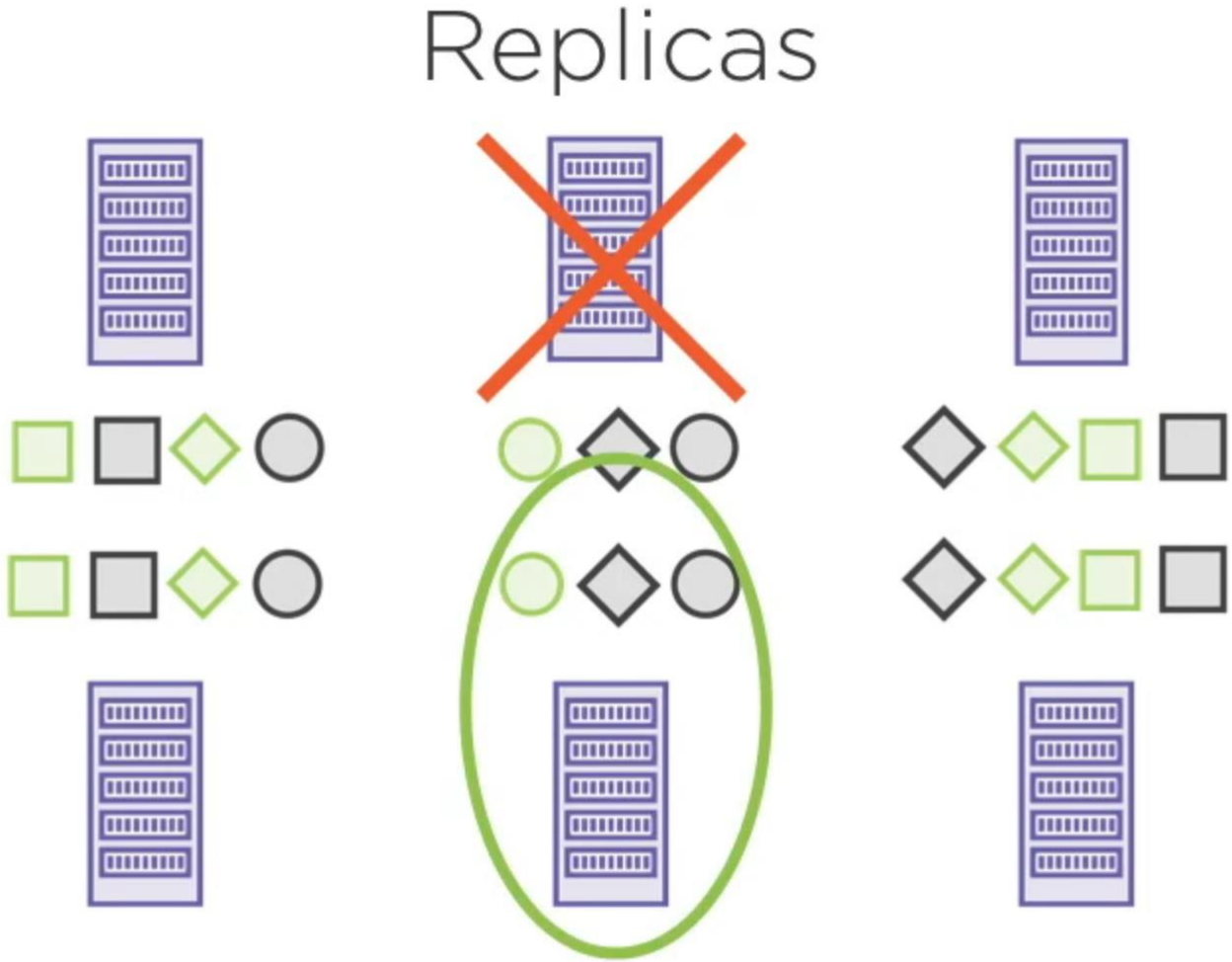
Sharding an index



Search in parallel on multiple nodes

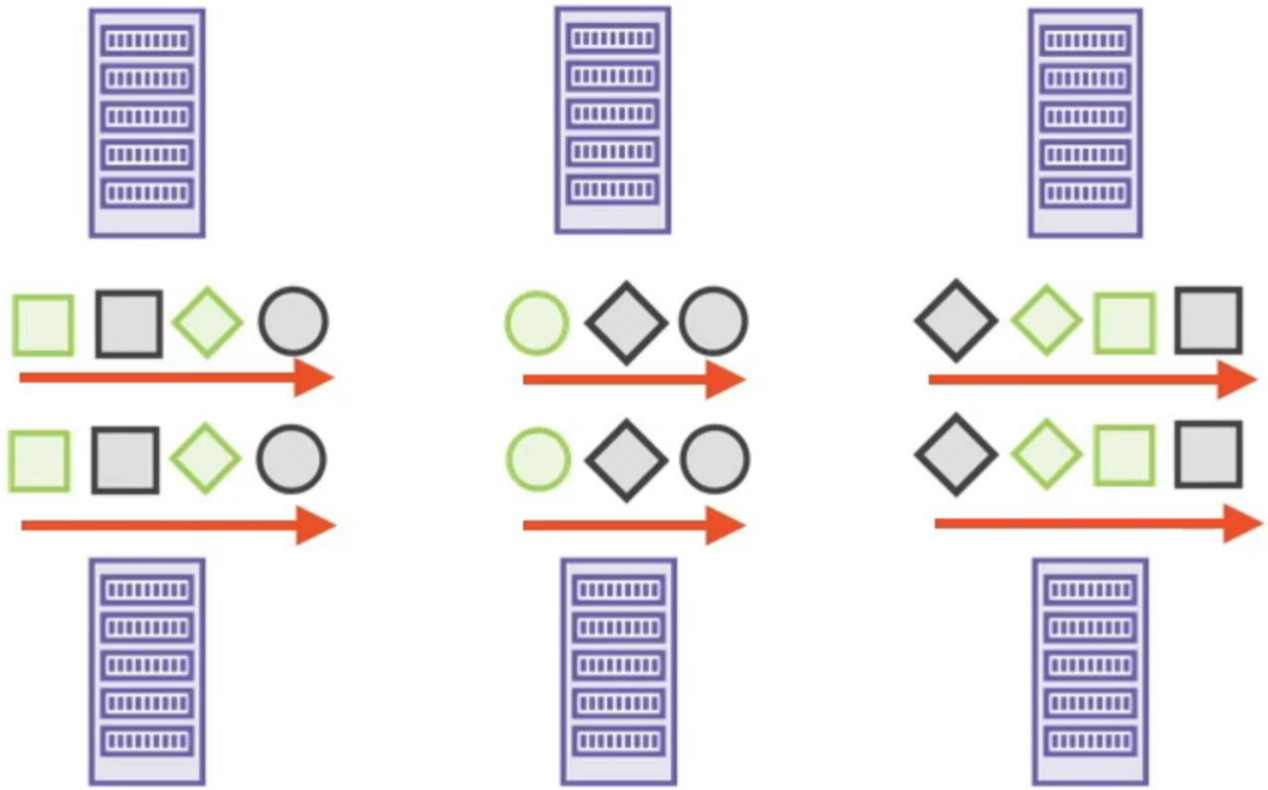
Replicas





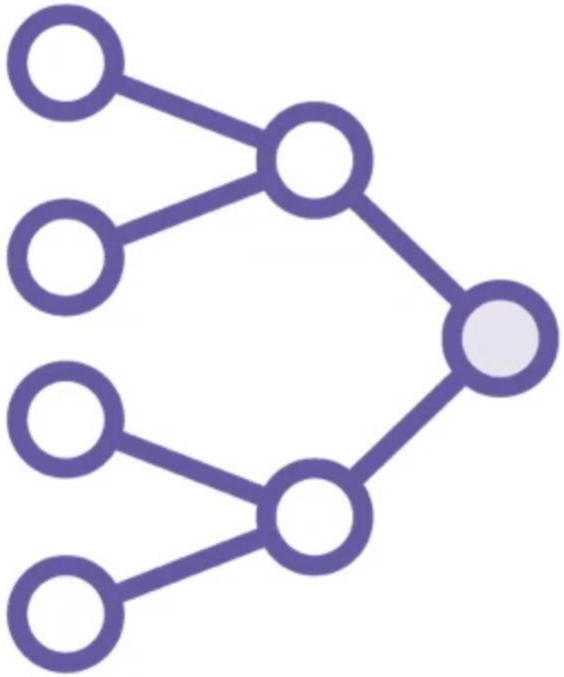
High availability in case a node fails

Replicas



Scale search volume/throughput by searching multiple replicas

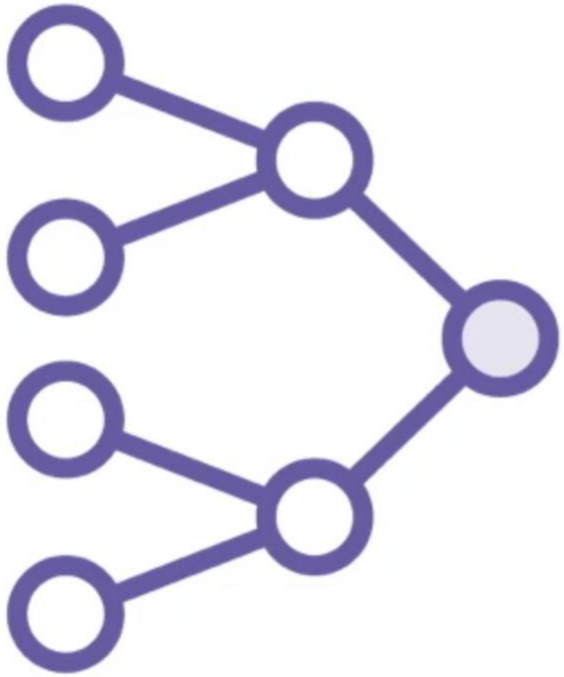
Shards and Replicas



An index can be split into multiple shards

A shard can be replicated zero or more times

An index in Elasticsearch has 5 shards and 1 replica by default



Shards and Replicas

Specifying the number of shards is static

- can only be done at index creation time

Specifying the number of replicas is dynamic

- can be changed after the index is created and populated

Environment Setup

https://cloud.elastic.co/registration?elektra=en-elasticsearch-page&storm=hero

https://www.elastic.co/downloads/elasticsearch

https://www.elastic.co/downloads/past-releases#elasticsearch

https://www.elastic.co/guide/en/elasticsearch/reference/current/targz.html

---

https://www.elastic.co/guide/en/elasticsearch/reference/current/es-release-notes.html

https://www.elastic.co/elasticsearch/features

https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-types.html

Deploy your way

Select a distribution model for your unique needs

Self-Managed

Install a single package

Elastic Cloud

Deploy instantly on AWS, Azure or Google Cloud

amazon web services™

Microsoft Azure

Elastic Cloud Enterprise

Elastic Cloud on Kubernetes

Centrally manage multiple deployments on your infra

ELASTICSEARCH QUERIES

What You Should Know

Basic understand of distributed systems

Comfortable with Linux command line interfaces

Understanding of JSON files

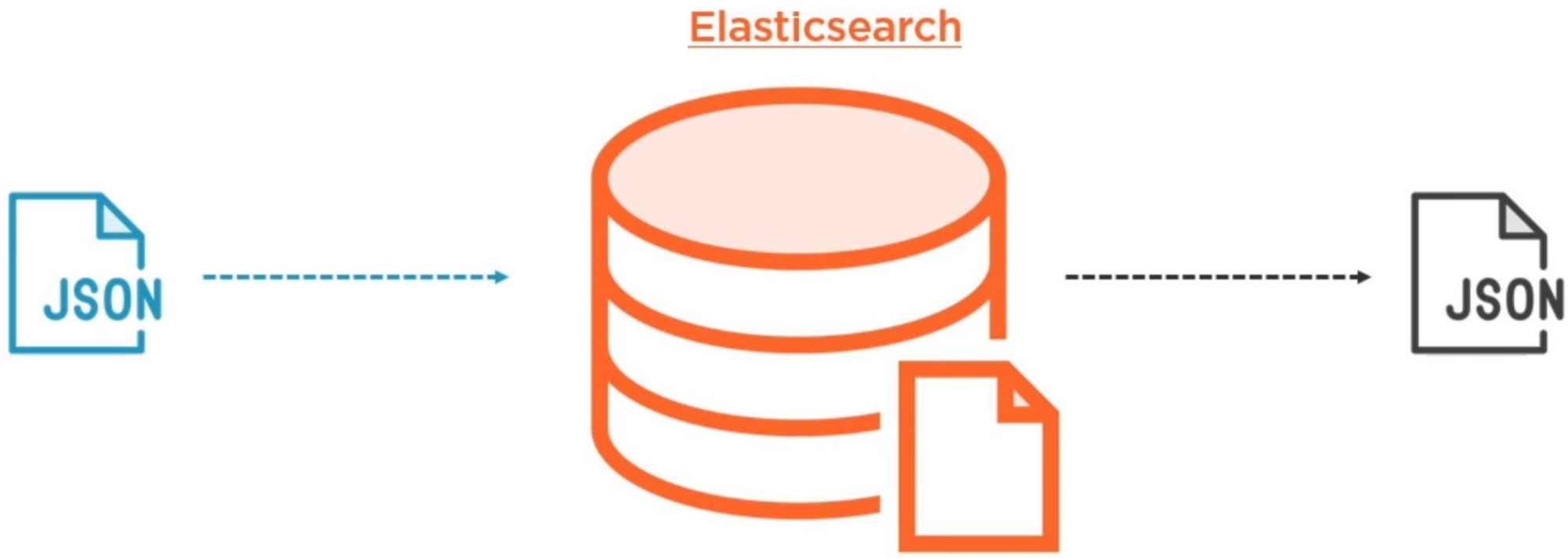
Opensource

ELK Stack

Based of Lucene

Java

Store data as JSON documents


Elasticsearch
JSON
JSON

Inverted Database

Indexing system designed to allow for fast full-text search. List are based on word/numbers and associated documents.

Doc 1
Helmet
Rope
Cams

Doc 2
Ax
Helmet
Climbing Shoes

Frequency	Term	Document
1	cams	1
1	rope	1
2	helmet	1,2
1	climbing	2
1	shoes	2

Uniform Resource Identifier

Commonly “URI”

Example

- http:
- mailto:

Simple URI request to Elasticsearch

```
curl -X GET "localhost:9200/customers/_search?pretty"
```

URI Query

Return the query contained in the index

Domain Specific Language

Based of JSON

- JavaScript Object Notation

Types of Clauses

- Leaf query
- Compound query

Complex request to Elasticsearch

```
GET /customers /_search
{
    “query”: {
        “match_all”:{}
    }
}
```

DSL Query

Return the query contained in the index

Large existing customer base

Delays in search impacting sales

Objectives of Proof of Concept

Ease of use for existing Data Team

Will Elasticsearch help Carved Rock to fix search challenges

Elasticsearch enable future use cases

Development Data

Carved Rock Fitness POC

accounts.json


```
{"index":{"_id":"1"}}

{"account_number":1,"balance":39225,"firstname":"Amber","lastname":"Duke","age":32,
"gender":"M","address":"880 Holmes
Lane","employer":"Pyrami","email":"amberduke@pyrami.com","city":"Brogan","state":"I
L"}
```


Sample data set of customer accounts *(not real data)*

```
Put accounts



Post accounts/_bulk

{"index":{"_id":"1"}}

{"account_number":1,"balance":39225,"firstname":"Amber","lastname":"Duke","age":32,
"gender":"M","address":"880 Holmes
Lane","employer":"Pyrami","email":"amberduke@pyrami.com","city":"Brogan","state":"I
L"}
```

Import Data into Elasticsearch

Bulk uploading our account data into Elasticsearch using bulk

Relational Database vs. Elasticsearch

Relational Database	Elasticsearch
SQL	DSL
Rows	Documents
Table	Index
Column	Field
3rd Normal Form	Inverted Database

Executing Queries

Relational Databases vs. Elasticsearch

Example.sql

```
SELECT *
FROM accounts
```

Example.json

```
GET /accounts /_search
{
    “query”: {
        “match_all”:{}
    }
}
```

Domain Specific Language

The Elasticsearch DSL was a built to give developers a high-level language to write both basic and complex queries for Elasticsearch. Abstract away the complexity.

```
GET /customers /_search
{
    “query”: {
        “match_all”:{}
    }
}
```

DSL Query

Return the query contained in the index

Leaf Query Clause

The Elasticsearch leaf clause is used to search for a particular field in the DSL query.

First Leaf Query

Leaf-by-age.json

```
GET /customers /_search
{
    “query”: {
        “match_all”:{
            “age”: “32”
        }
    }
}
```

Full Text Queries

Elasticsearch queries where analysis is conducted on query before execution.

Term Queries

Elasticsearch queries where no analysis is conducted on query before execution.

Full Text Query vs. Term Query

Full Text

```
GET /accounts/_search
{
      “query”: {
            “match”:{
                  “firstname”: “huff dale”
            }
      }
}
```

Term

```
GET /accounts/_search
{
      “query”: {
            “term”:{
                  “firstname”: “huff dale”
            }
      }
}
```

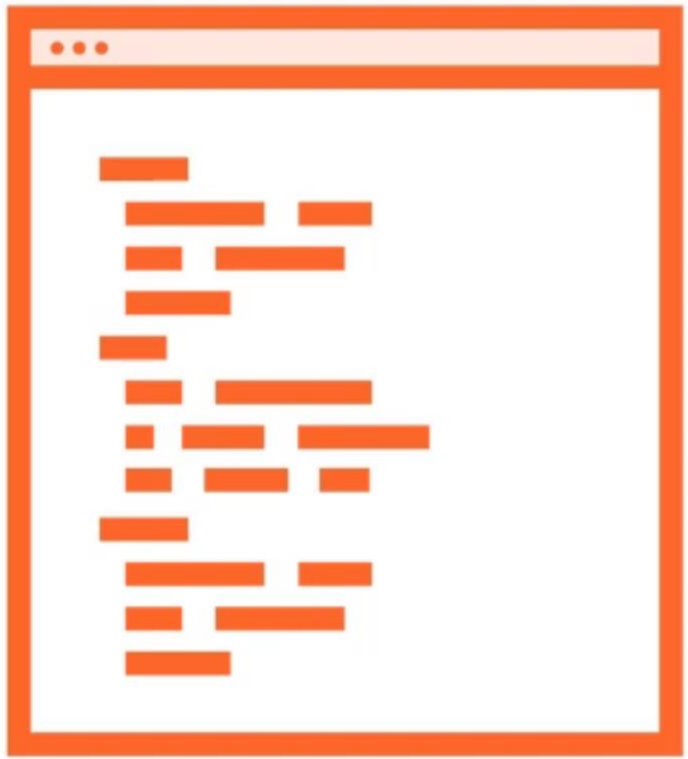
Mutli-Match

- Full-text
- Allows multiple fields

Multi-Match

Leaf-by-age.json

```
GET /accounts/_search
{
    “query”: {
        “multi_match”:{
            “query”: “ford”
            “fields”: [“firstname”, “address”]
        }
    }
}
```



Mutli-Match

- Full-text
- Allows multiple fields

Match_phrase

- Full-text
- Matches phrases in a field

Match_phrase

Match_phrase

```
GET /accounts/_search
{
    “query”: {
        “match_phrase”:{
            “address”:“sedgwick street”
            }
        }
    }
}
```

Wildcards

Wildcards Query

```
GET /accounts/_search
{
    “query”: {
        “wildcard”:{
            “firstname”: {
                “value”: “h*ll”
            }
        }
    }
}
```

Compound Query

Compound Queries

In Elasticsearch, compound queries are combinations of leaf query clauses and other compound queries for search results.

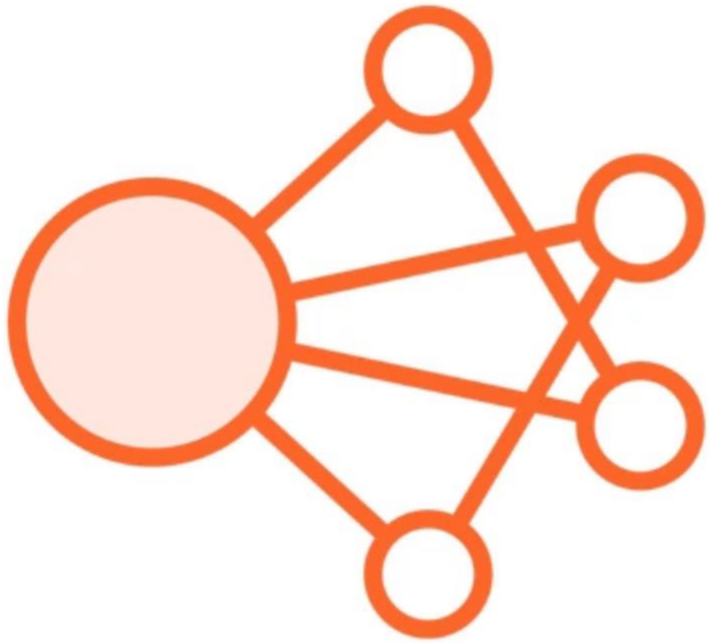
```
GET /accounts /_search
{
      “query”: {
            “bool”:{
                  //clause here//
            }
      }
}
```

Example Boolean Query

Default query for combining multiple leaf or compound queries.

Accounts.json


```
{"index":{"_id":"1"}}

{"account_number":1,"balance":39225,"firstname":"Amber","lastname":"Duke","age":32,
"gender":"M","address":"880 Holmes
Lane","employer":"Pyrami","email":"amberduke@pyrami.com","city":"Brogan","state":"I
L"}

{"index":{"_id":"6"}}

{"account_number":6,"balance":5686,"firstname":"Hattie","lastname":"Bond","age":36,
"gender":"M","address":"671 Bristol
Street","employer":"Netagy","email":"hattiebond@netagy.com","city":"Dante","state":
"TN"}

{"index":{"_id":"13"}}
```



Must

- Must appear in document

Should

- Should appear in document

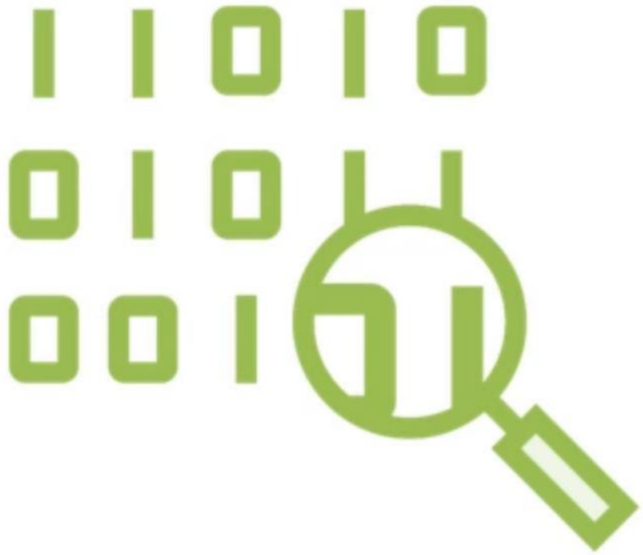
Must_not

- Not appear in document

Filter

- Must appear but score ignored

Using Scoring with Elasticsearch



WRONG WAY

Employer: Quility
Address: River

"index":{"_id":"44"}}
{"account_number":44,"balance“.....

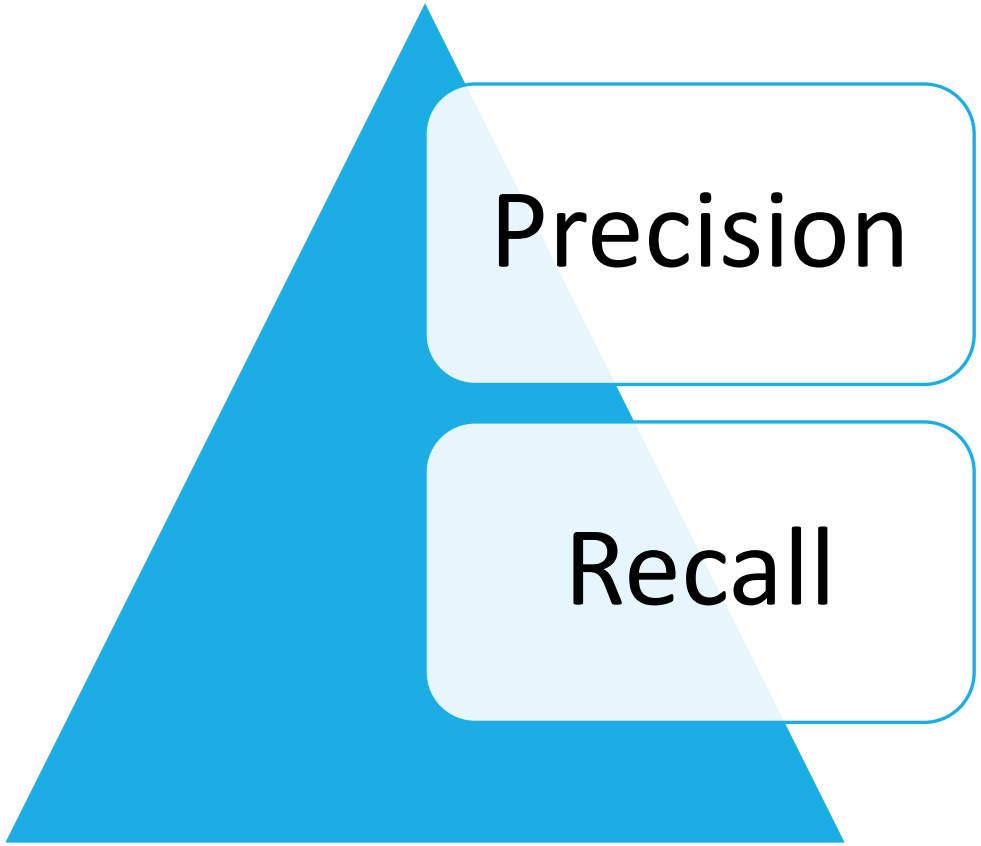
Speed, Scale,

Relevance

Elastic is a search company.

We focus on value to users by producing fast results that operate at scale and are relevant. This is our DNA. We believe search is an experience. It is what defines us, and makes us unique.

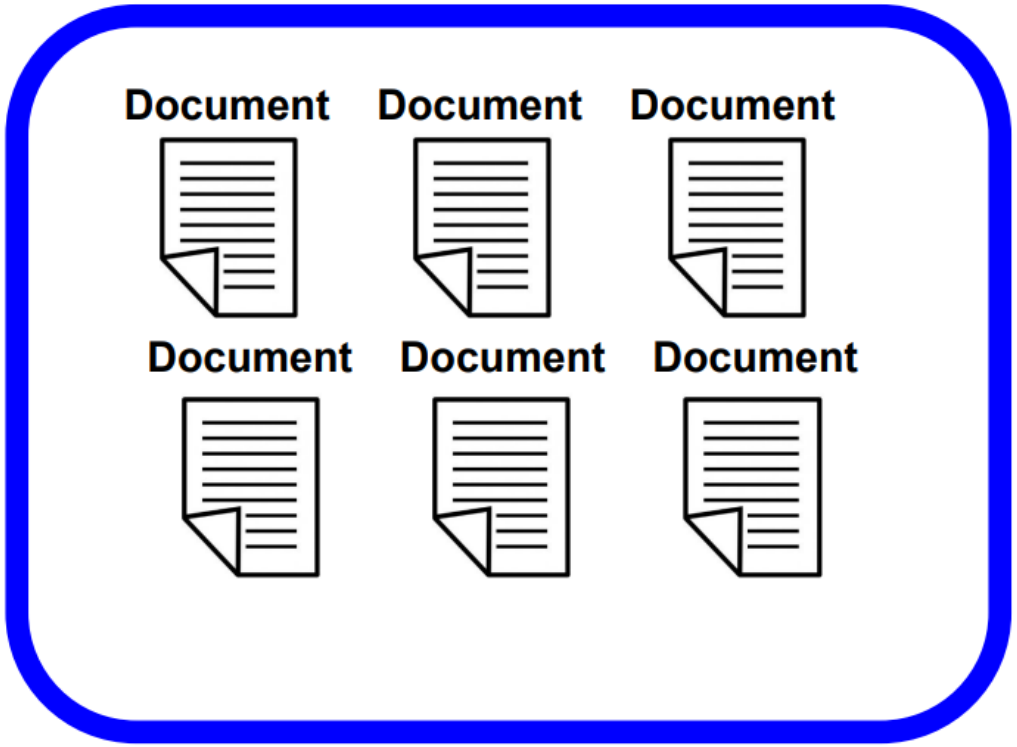
How to measure the relevance?



I store data as documents!

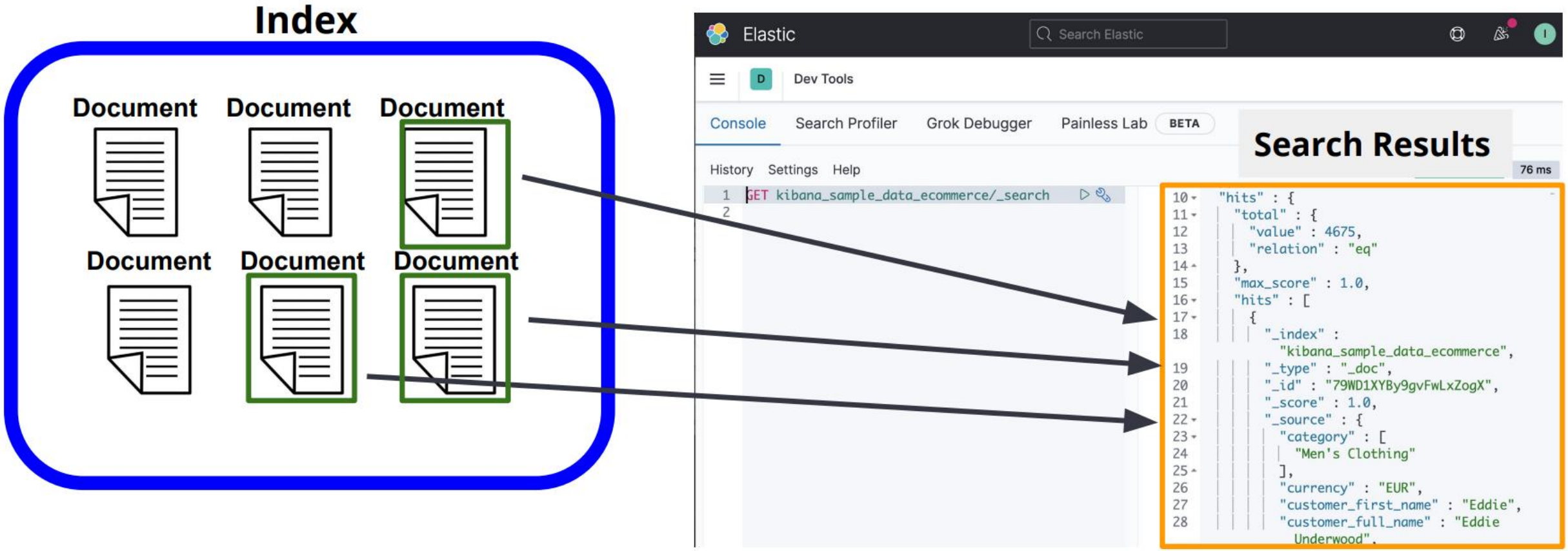
Elasticsearch

Store | Search | Analyze

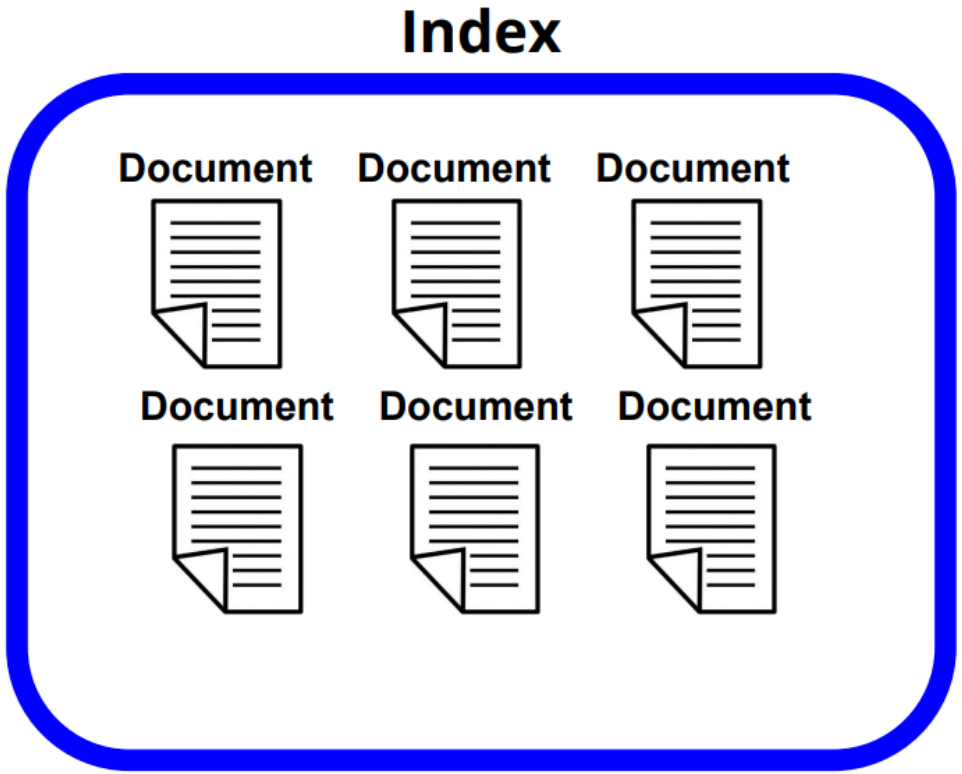


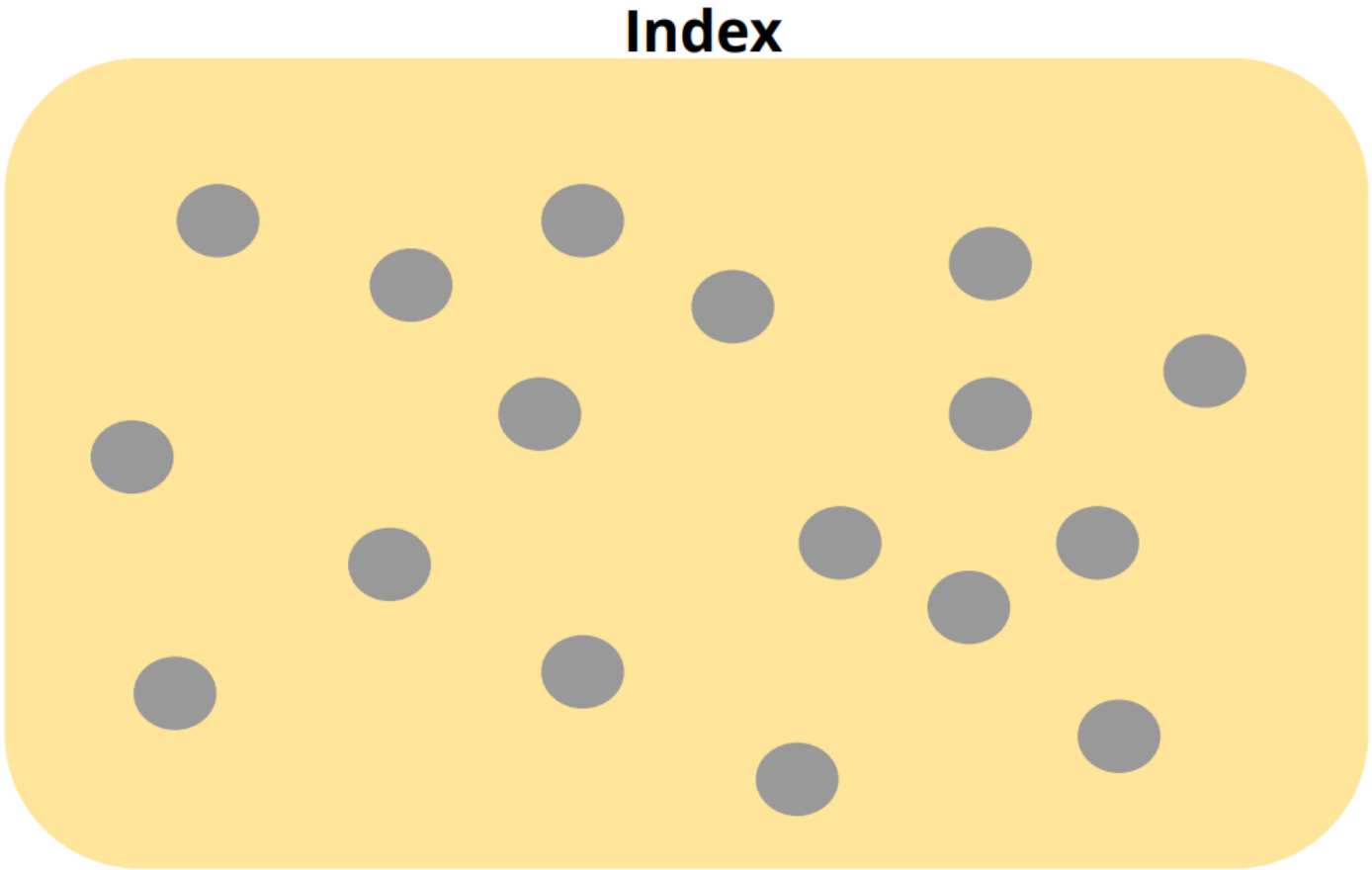
Documents with similar traits are grouped into an index!

When search query is sent, Elasticsearch retrieves relevant documents and presents the documents as search results.

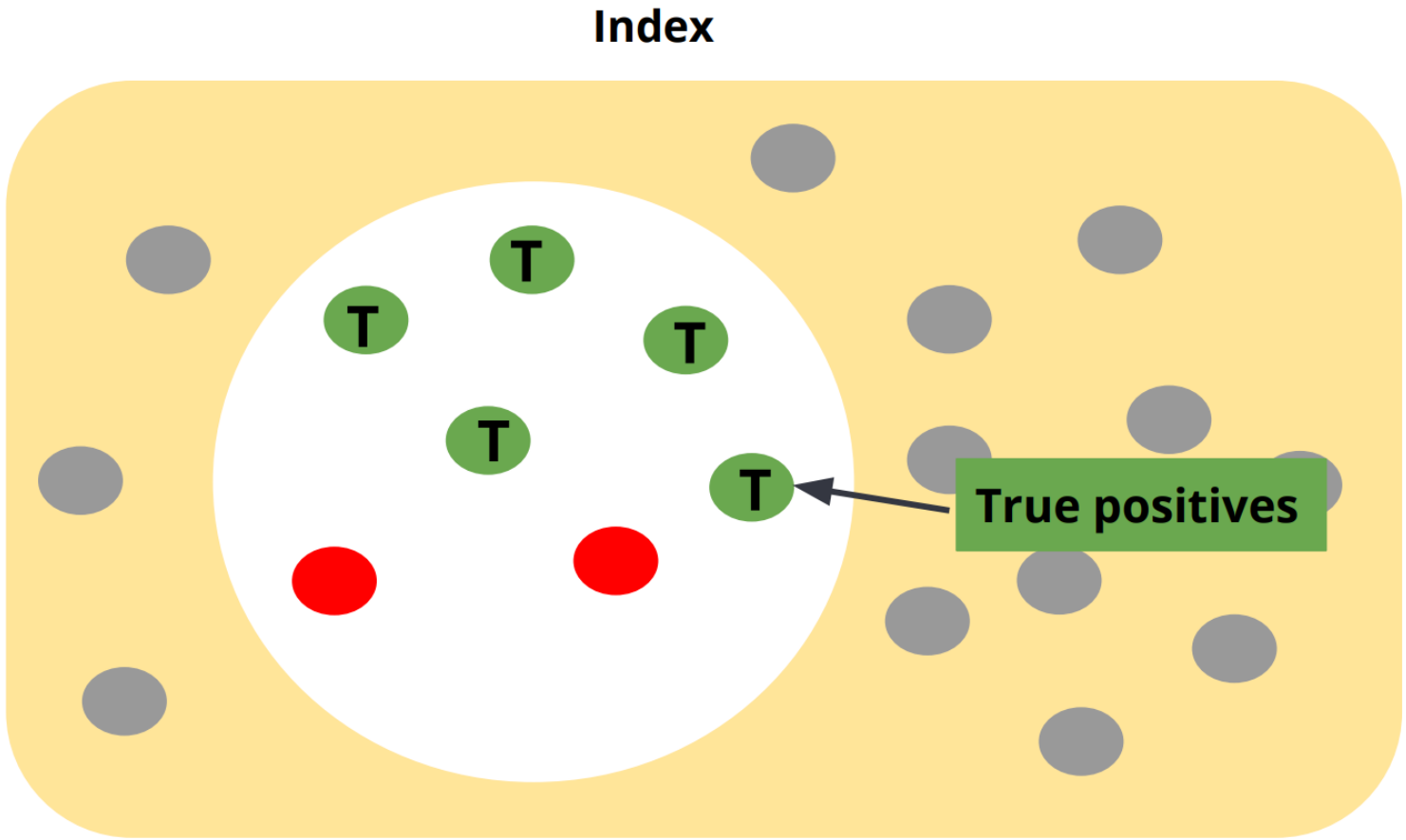


These two diagrams depict the same thing!

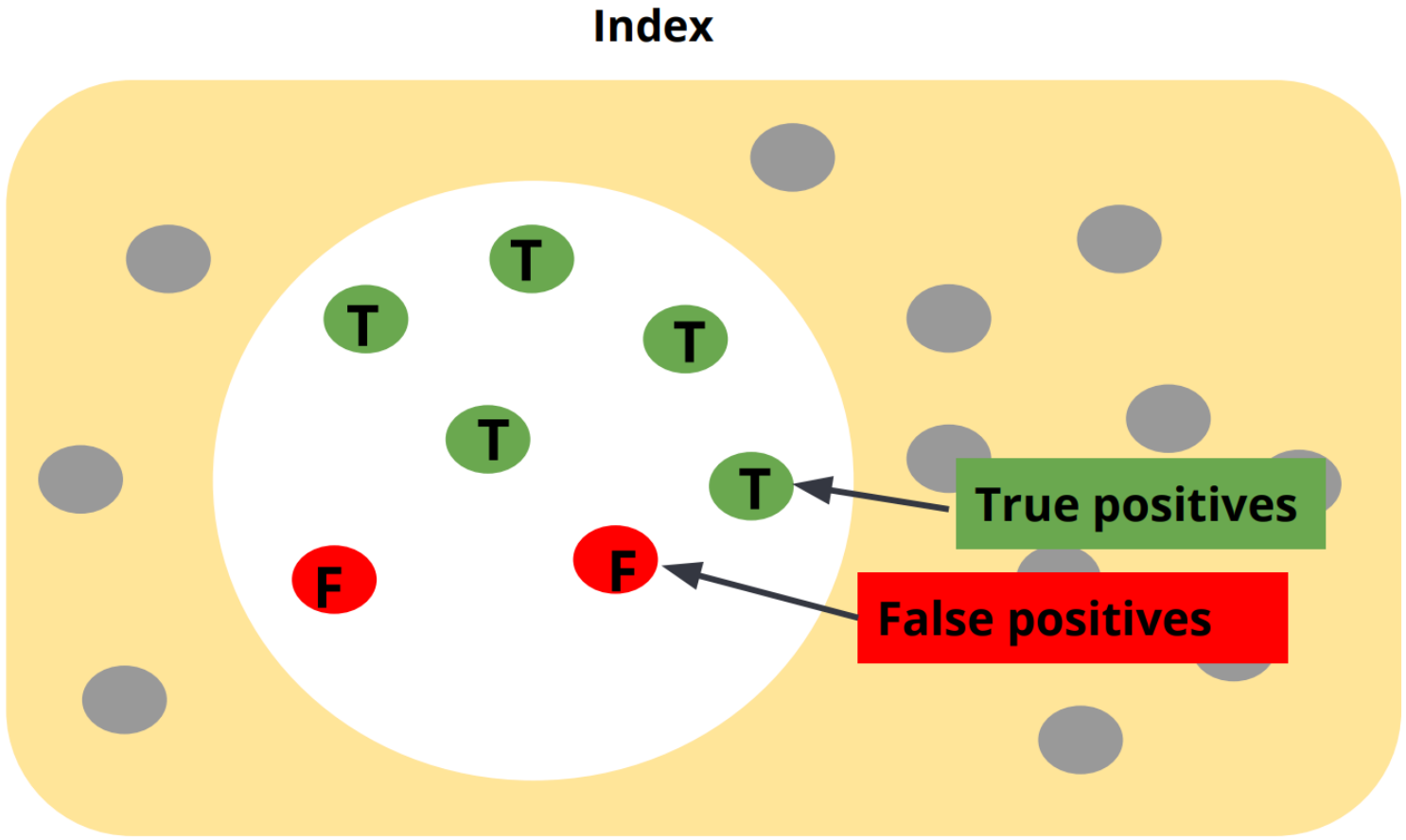




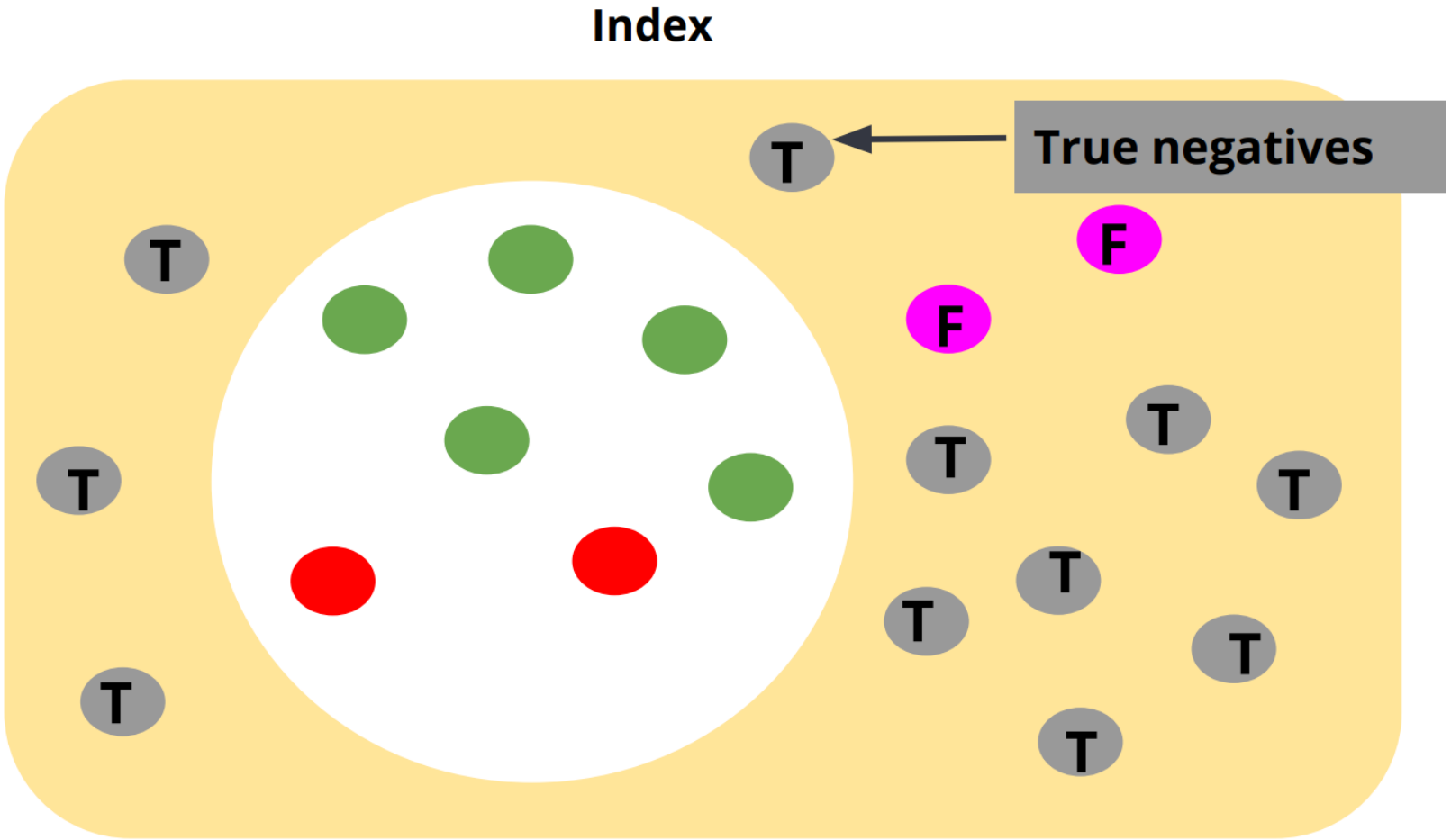
True positives are relevant documents that are returned to the user.



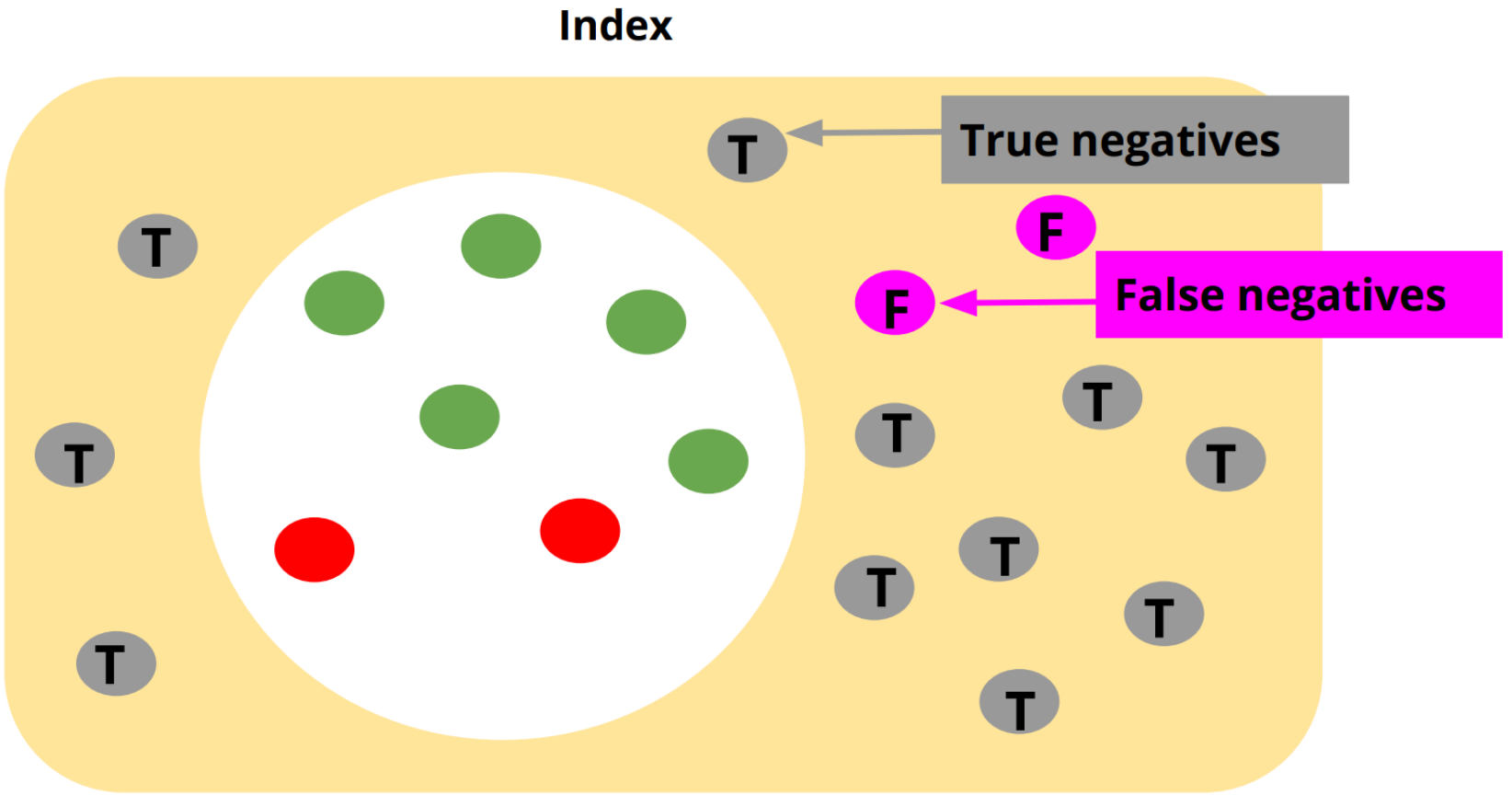
False positives are irrelevant documents that are returned to the user.



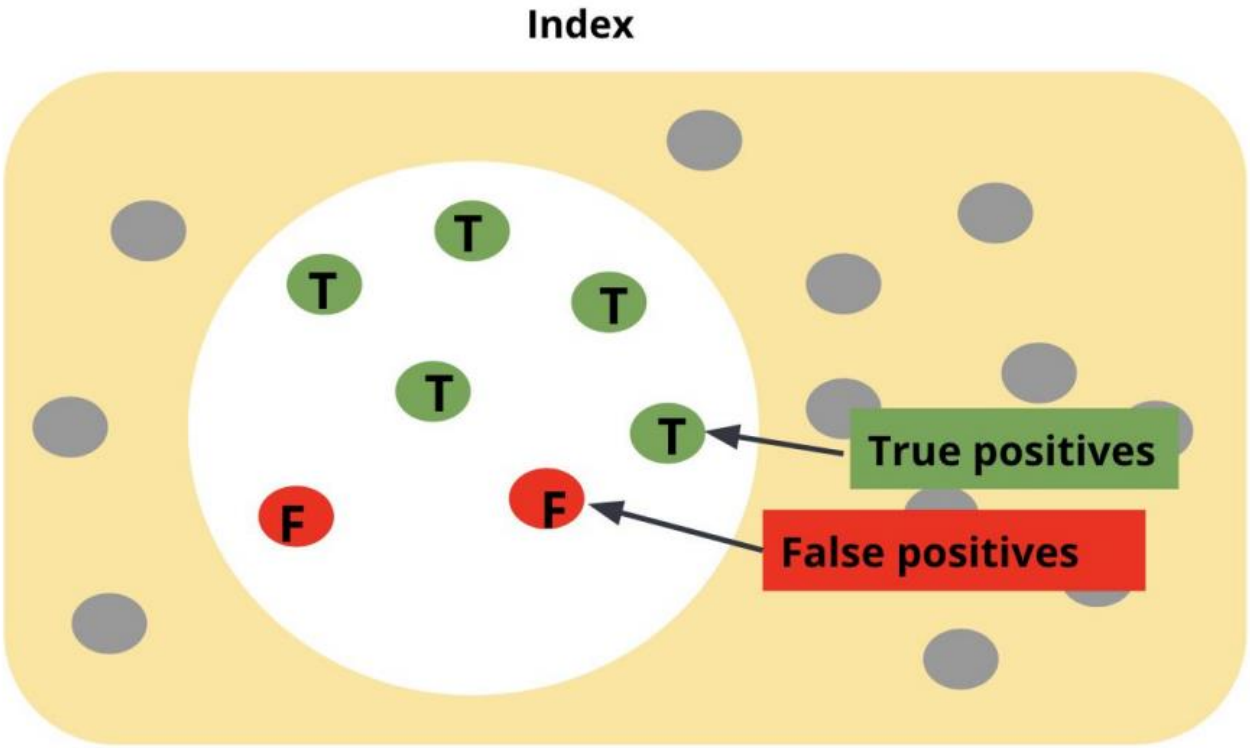
True negatives are irrelevant documents that are not returned to the user.



False negatives are relevant documents that were not returned to the user.



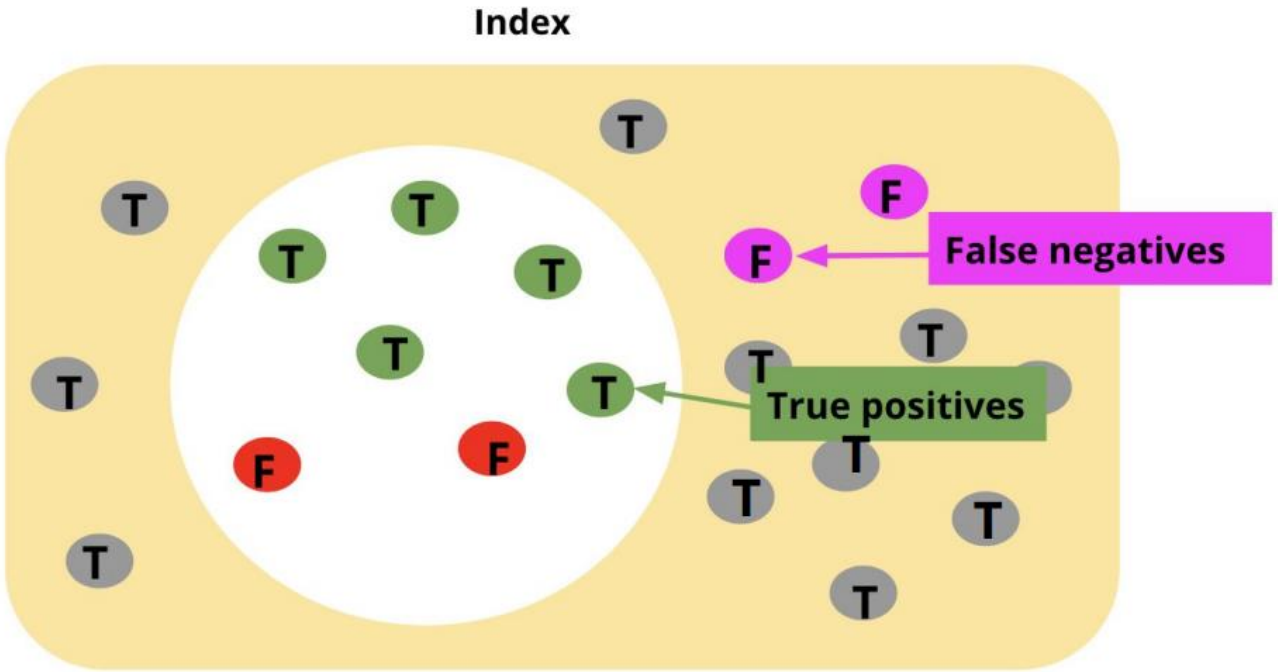
What is precision?



$$\text{Precision} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

What portion of the retrieved data is actually relevant to the search query?

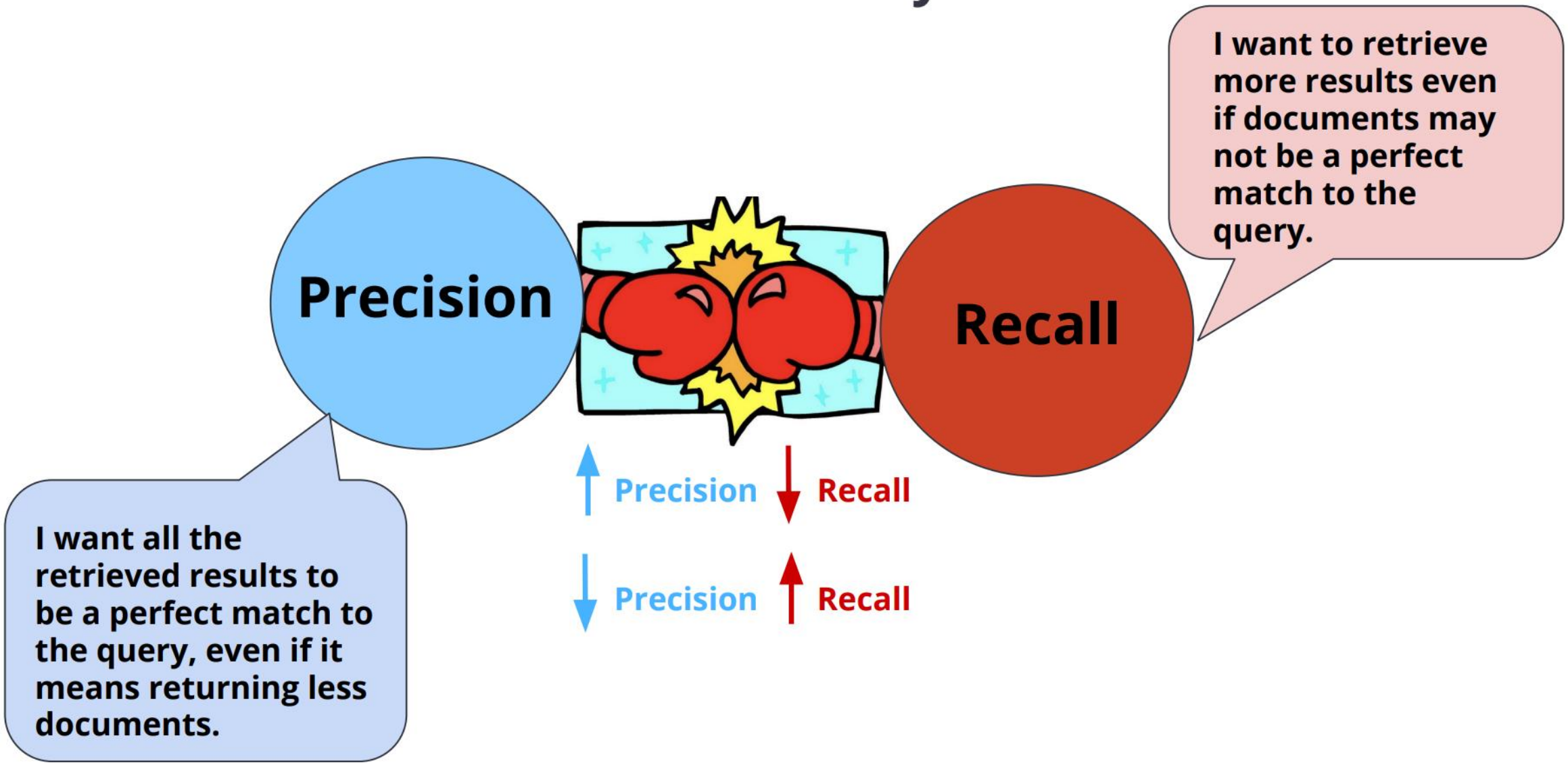
What is recall?



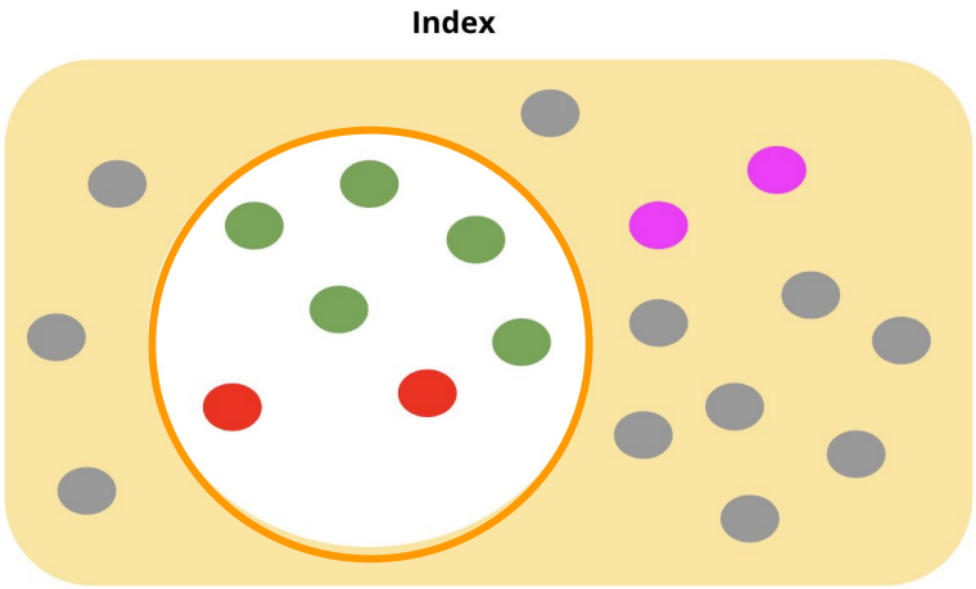
$$\text{Recall} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}}$$

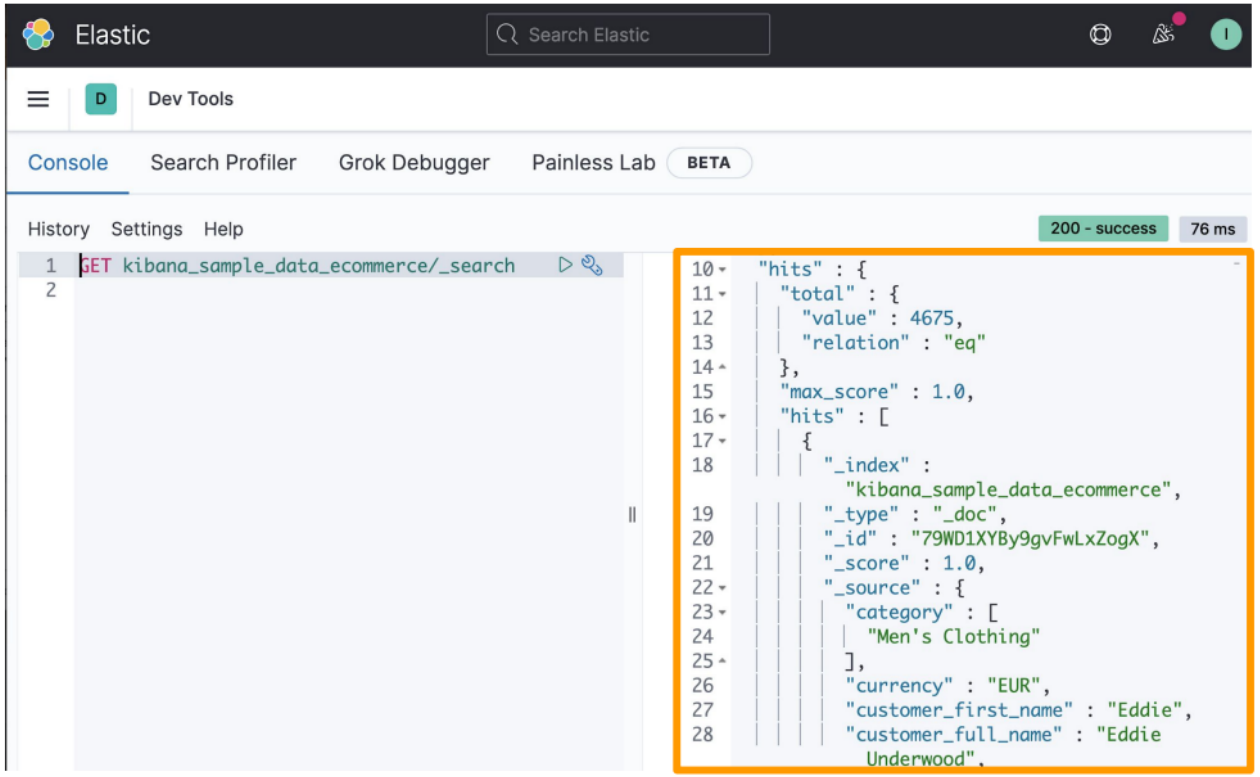
What portion of relevant data is being returned as search results?

Precision and Recall are inversely related



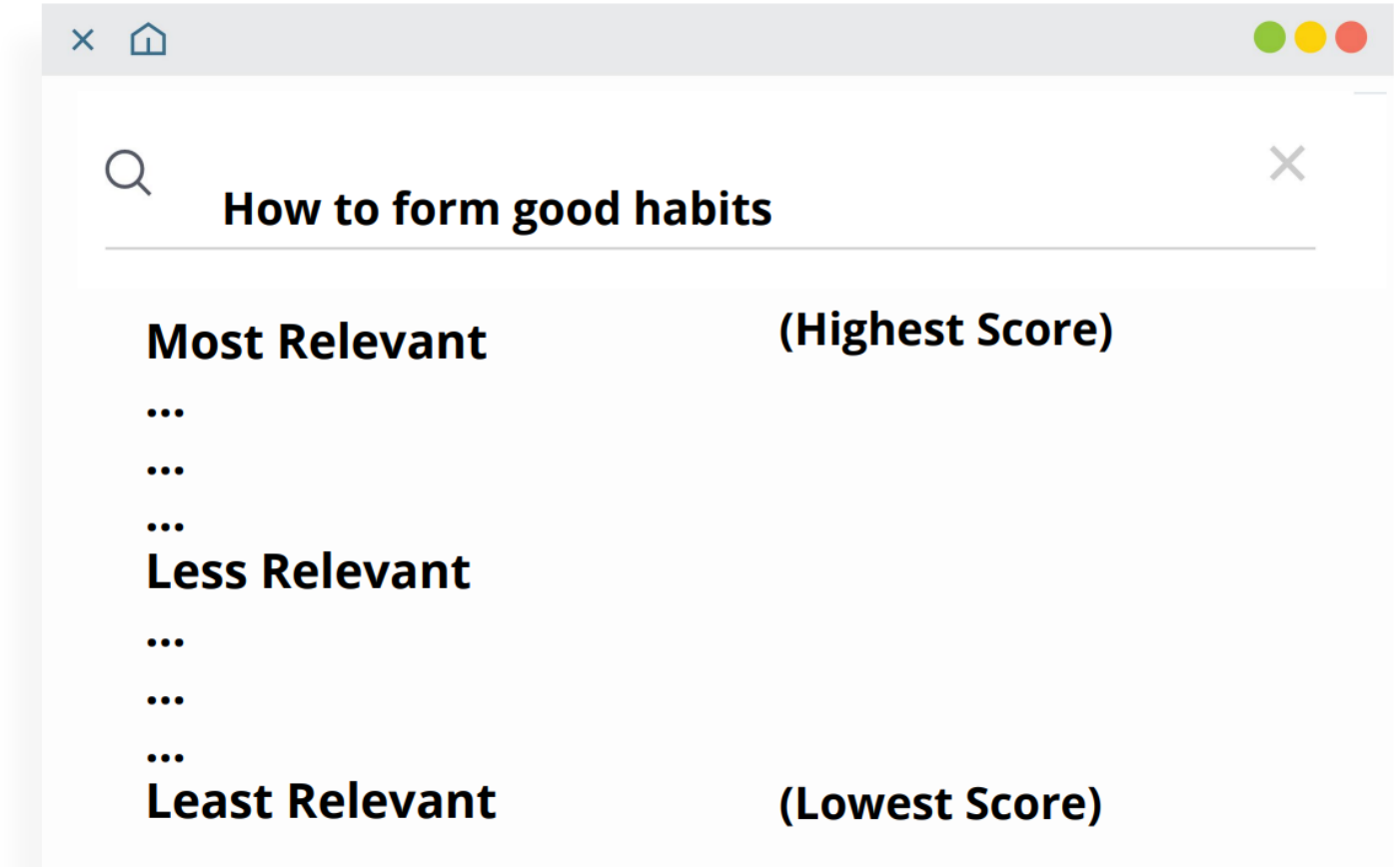
Precision and recall determine which documents are included in the search results.





Precision and recall do not determine which of the returned documents are more relevant compared to the other!

Ranking refers to ordering of the results (from most relevant results at the top, to least relevant at the bottom).



What is score?

- The score is a value that represents how relevant a document is to that specific query
- A score is computed for each document that is a hit

What is score?

- Term Frequency(TF)
- Inverse Document Frequency(IDF)

Scoring

Elasticsearch uses scoring to assign a value of the relevance of the results to the query. The higher the score the better fit the document to the query.

TF/IDF Relevance Algorithm

Term frequency

How often does the term appear in the field?

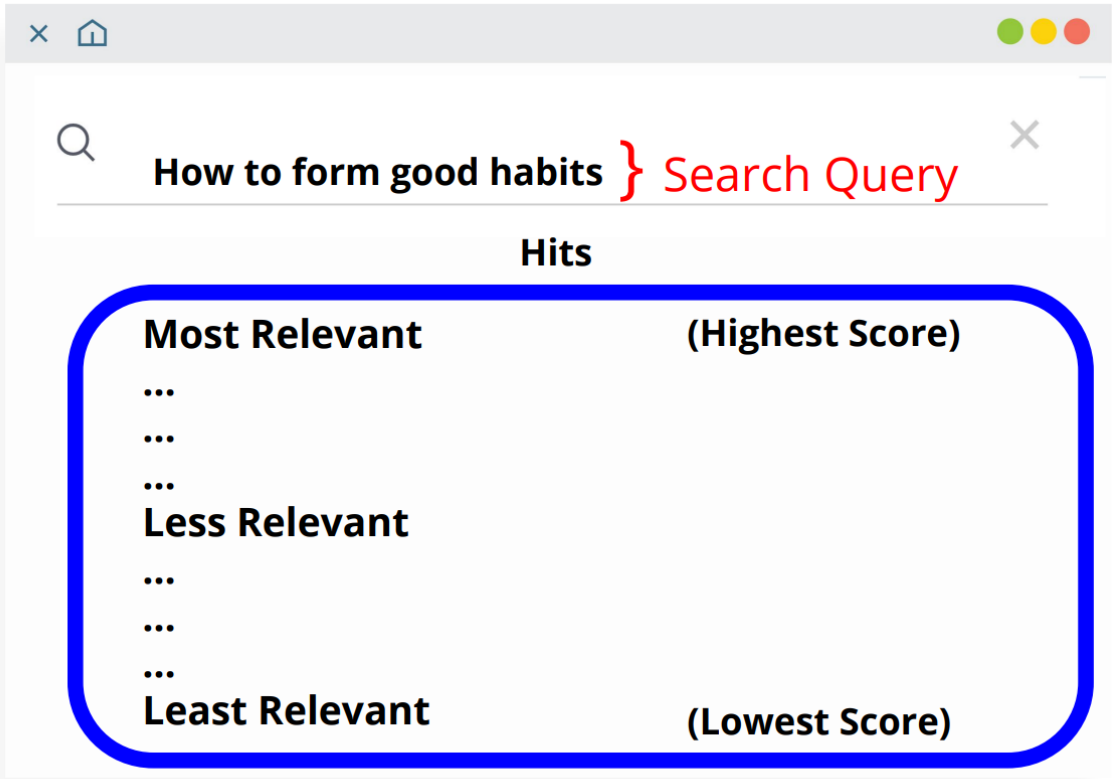
Inverse document frequency

How often does the term appear in the index?

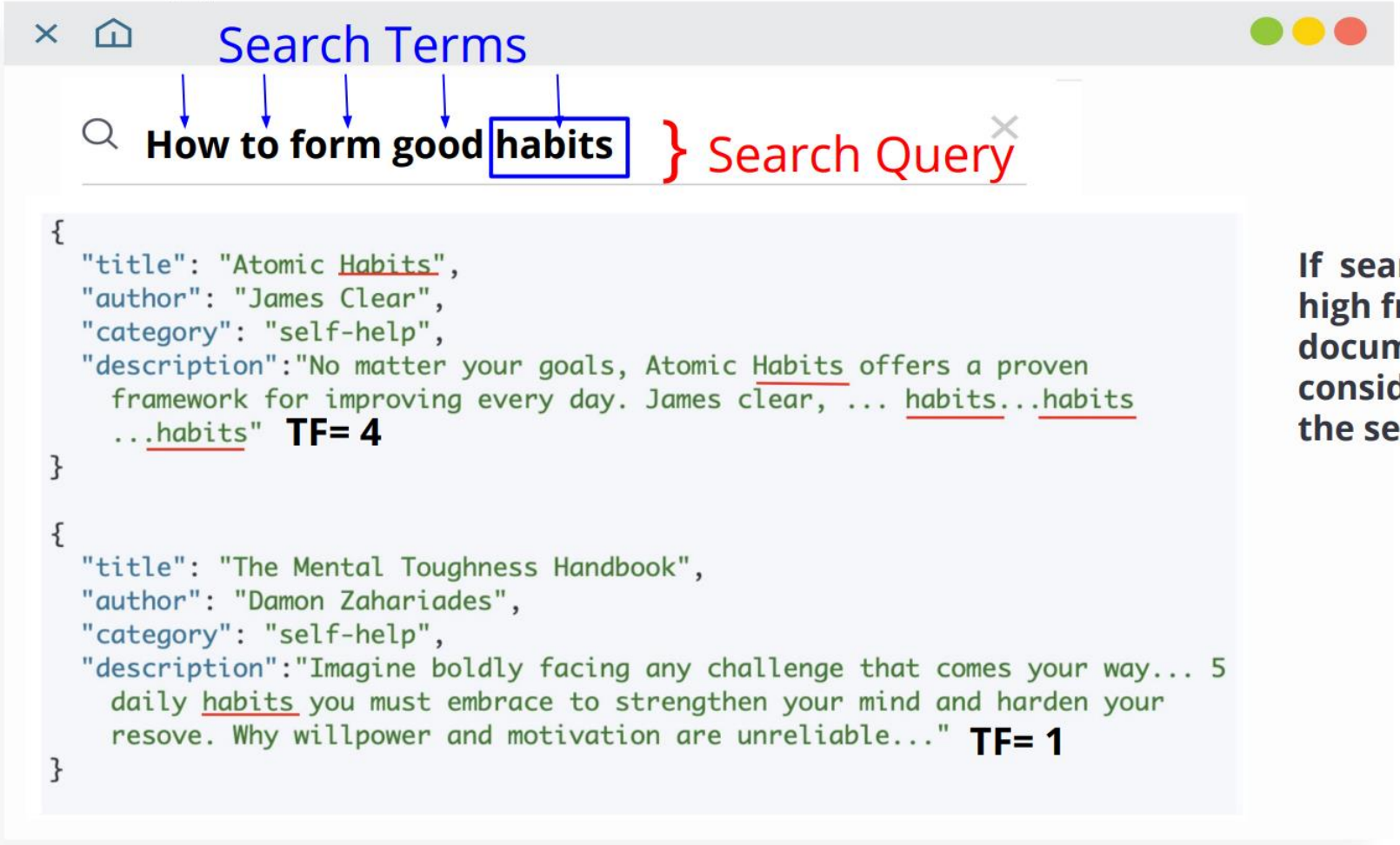
Field-length norm

How long is the field which was searched?

What is score?

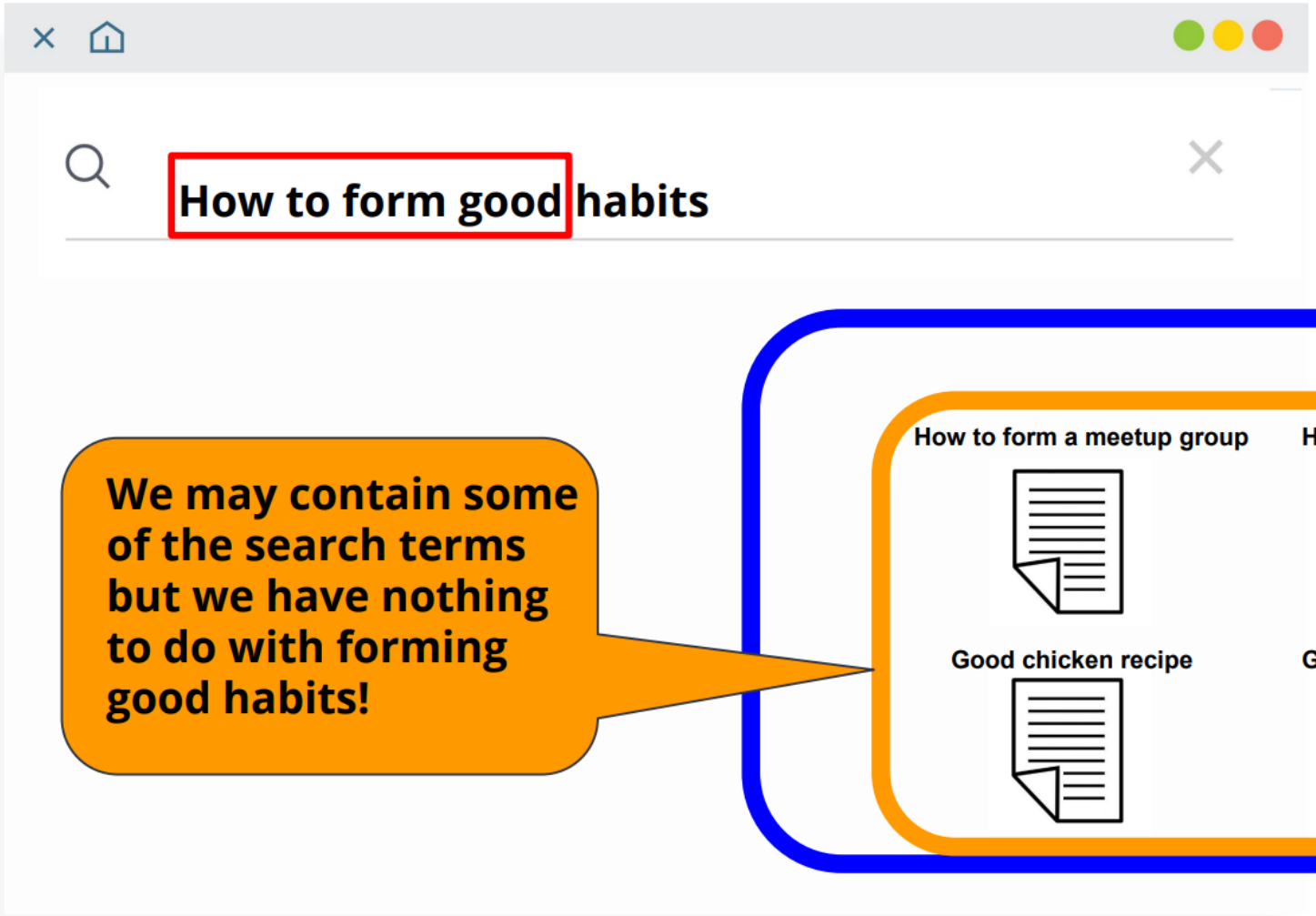


Term Frequency(TF) determines how many times each search term appears in a document.



If search terms are found in high frequency in a document, the document is considered more relevant to the search query.

What is Inverse Document Frequency(IDF)?



IDF diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely!

We may contain some of the search terms but we have nothing to do with forming good habits!

Hits

How to form a meetup group

How to form a band

Good habits 101

Good chicken recipe

Good times rolling

Good habits are easy to master!

Scoring Query Results

```
GET /accounts/_search
{
  "query": {
    "match_all": {}
  }
}
```


```
{
  "took" : 14,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 2000,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "accounts",
        "_type" : "_doc",
        "_id" : "owQyw3YBddLz9mGwhUAE",
        "_score" : 1.0,
        "_source" : {
          "index" : {
            "_id" : "1"
          }
        }
      },
      {
        "_index" : "accounts",
        "_type" : "_doc",
        "_id" : "pAQyw3YBddLz9mGwhUAh",
        "_score" : 1.0,
```

4 Types of Compound Queries

Boosting Query – boosting

Constant Score – constant_score

Distance Max – dis_max

Function Score – function_score

bool query

- The default query for combining multiple leaf or compound query clauses, as must, should, must_not, or filter clauses. The must and should clauses have their scores combined — the more matching clauses, the better — while the must_not and filter clauses are executed in filter context.

boosting query

- Return documents which match a positive query, but reduce the score of documents which also match a negative query.

constant_score query

- A query which wraps another query, but executes it in filter context. All matching documents are given the same “constant” _score.

dis_max query

- A query which accepts multiple queries, and returns any documents which match any of the query clauses. While the bool query combines the scores from all matching queries, the dis_max query uses the score of the single best- matching query clause.

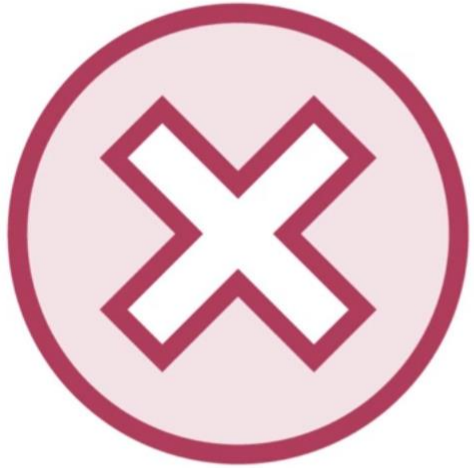
function_score query

- Modify the scores returned by the main query with functions to take into account factors like popularity, recency, distance, or custom algorithms implemented with scripting.

Types of Boolean

Must

Should



Must Not

Filter
(ignore score)

Boolean Query:

A query that matches documents matching boolean combinations of other queries. The bool query maps to Lucene BooleanQuery. It is built using one or more boolean clauses, each clause with a typed occurrence.

The occurrence types are:

Occur	Description
must	The clause (query) must appear in matching documents and will contribute to the score.
filter	The clause (query) must appear in matching documents. However unlike must the score of the query will be ignored. Filter clauses are executed in filter context, meaning that scoring is ignored and clauses are considered for caching.
should	The clause (query) should appear in the matching document.
must_not	The clause (query) must not appear in the matching documents. Clauses are executed in filter context meaning that scoring is ignored and clauses are considered for caching. Because scoring is ignored, a score of 0 for all documents is returned.

```
GET /kibana_sample_data_ecommerce/_search
{
     “query”: {
          “bool”:{
               “must”:{
                    “term”: { “name” : “Bill” }
          }
     }
}
```

Must

The query must appear in the document and impacts score

```
GET /kibana_sample_data_ecommerce/_search

{
     “query”: {
          “bool”:{
               “should”:{
                    “term”: { “name” : “Bill” }
          }
     }
}
```

Should

The query should appear in the document

```
GET /kibana_sample_data_ecommerce/_search
{
     “query”: {
          “bool”:{
                    “must_not”:{
                         “term”: { “name” : “Bill” }
          }
     }
}
```

Must Not

The query should appear in the document

```
GET /kibana_sample_data_ecommerce/_search
{  "query":
     {    "bool":
{      "filter": [
          {"term": {"manufacturer": "Pyramidustries"}},
          {"term": {"category": "Women's Clothes"}}

          ] }
     }
}
```

Filter

Must appear in document but score ignored

Elasticsearch Scoring Queries

Boosting Query

Returns documents matching a positive query while reducing the relevance score of documents that also match a negative query.

You can use the boosting query to demote certain documents without excluding them from the search results.

```
GET /_search
{
  "query": {
    "boosting": {
      "positive": {
        "term": {
          "text": "apple"
        }
      },
      "negative": {
        "term": {
          "text": "pie tart fruit crumble tree"
        }
      },
      "negative_boost": 0.5
    }
  }
}
```

Top-level parameters for boosting

Positive:

(Required, query object) Query you wish to run. Any returned documents must match this query.

Negative:

(Required, query object) Query used to decrease the relevance score of matching documents.

If a returned document matches the positive query and this query, the boosting query calculates the final relevance score for the document as follows:

1. Take the original relevance score from the positive query.
2. Multiply the score by the negative_boost value.

Negative_boost:

(Required, float) Floating point number between 0 and 1.0 used to decrease the relevance scores of documents matching the negative query.

```
"boosting": {
    "positive": {
        "term": {
            "text": "Dillard"
}       },
    "negative": {
        "term": {
            "text": "Mooney"
}
},
"negative_boost": 0.5
```

- Boosting
- Return document which match positive query
- Reducing score of documents with negative match
- Score

Elasticsearch Scoring Queries

Boosting Query

Constant Score

```
“constant_score": {
    “filter": {
        "term":{ “lastname”: Dillard”}
        },
    "boost": 1.1
```

- Constant wraps filter query
- Any results in document must match query
- Boost for relevance score

Wraps a filter query and returns every matching document with a relevance score equal to the boost parameter value.

```
GET /_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": { "user.id": "kimchy" }
      },
      "boost": 1.2
    }
  }
}
```

Top-level parameters for constant_score

Filter:

(Required, query object) Filter query you wish to run. Any returned documents must match this query.

Filter queries do not calculate relevance scores. To speed up performance, Elasticsearch automatically caches frequently used filter queries.

Boost:

(Optional, float) Floating point number used as the constant relevance score for every document matching the filter query. Defaults to 1.0.

Elasticsearch Scoring Queries

Boosting Query

Constant Score

Disjunction Max

```
“dis_max": {
   “queries": [
    {"term":{ “lastname”: “Dillard”}},
    {"term":{ “age”: “32”}},
    ],
    “tie_breaker": 0.6
```

- Disjunction query allow for many query clauses
- Query allows for adding clauses
- Option for increasing score (relevance)

Returns documents matching one or more wrapped queries, called query clauses or clauses.
If a returned document matches multiple query clauses, the dis_max query assigns the document the highest relevance score from any matching clause, plus a tie breaking increment for any additional matching subqueries.

You can use the dis_max to search for a term in fields mapped with different boost factors.

```
GET /_search
{
  "query": {
    "dis_max": {
      "queries": [
        { "term": { "title": "Quick pets" } },
        { "term": { "body": "Quick pets" } }
      ],
      "tie_breaker": 0.7
    }
  }
}
```

Top-level parameters for dis_max

Queries:
(Required, array of query objects) Contains one or more query clauses. Returned documents must match one or more of these queries. If a document matches multiple queries, Elasticsearch uses the highest relevance score.

tie_breaker:
(Optional, float) Floating point number between 0 and 1.0 used to increase the relevance scores of documents matching multiple query clauses. Defaults to 0.0.

You can use the **tie_breaker** value to assign higher relevance scores to documents that contain the same term in multiple fields than documents that contain this term in only the best of those multiple fields, without confusing this with the better case of two different terms in the multiple fields.

If a document matches multiple clauses, the **dis_max** query calculates the relevance score for the document as follows:

Take the relevance score from a matching clause with the highest score.
Multiply the score from any other matching clauses by the **tie_breaker** value.
Add the highest score to the multiplied scores.
If the **tie_breaker** value is greater than 0.0, all matching clauses count, but the clause with the highest score counts most.

Elasticsearch Scoring Queries

Boosting Query

Constant Score

Disjunction Max

Function Score

```
“function_score": {
   “query": {“match_all”:{}};
    “boost”: “5”
    //query clauses…
    “random_score”:{},
    “boost_mode”: “multiply”
```

- Function Score allows for modification of the score of documents.
- Query allows for adding clauses
- Boost score for nested query clauses
- Option for increasing score (relevance)

The **function_score** allows you to modify the score of documents that are retrieved by a query. This can be useful if, for example, a score function is computationally expensive and it is sufficient to compute the score on a filtered set of documents.

To use **function_score**, the user has to define a query and one or more functions, that compute a new score for each document returned by the query.

function_score can be used with only one function like this:

```
GET /_search
{
  "query": {
    "function_score": {
      "query": { "match_all": {} },
      "boost": "5",
      "random_score": {}, (1)
      "boost_mode": "multiply"
    }
  }
}
```

If no filter is given with a function this is equivalent to specifying `"match_all": {}`

First, each document is scored by the defined functions. The parameter `score_mode` specifies how the computed scores are combined:

`multiply`	scores are multiplied (default)
`sum`	scores are summed
`avg`	scores are averaged
`first`	the first function that has a matching filter is applied
`max`	maximum score is used
`min`	minimum score is used

Furthermore, several functions can be combined. In this case one can optionally choose to apply the function only if a document matches a given filtering query

```
GET /_search
{
  "query": {
    "function_score": {
      "query": { "match_all": {} },
      "boost": "5", (1)
      "functions": [
        {
          "filter": { "match": { "test": "bar" } },
          "random_score": {}, (2)
          "weight": 23
        },
        {
          "filter": { "match": { "test": "cat" } },
          "weight": 42
        }
      ],
      "max_boost": 42,
      "score_mode": "max",
      "boost_mode": "multiply",
      "min_score": 42
    }
  }
}
```

https://www.elastic.co/guide/en/elasticsearch/reference/7.11/query-dsl-function-score-query.html

Explaining the Cost of Queries

Doc 1
Helmet
Rope
Cams

Doc 2
Ax
Helmet
Climbing Shoes

Frequency	Term	Document
1	cams	1
1	rope	1
2	helmet	1,2
1	climbing	2
1	shoes	2

Expensive Queries

Queries that tend to search large portions of indexes. Common systems are slow search results, high CPU, or long-time lapses.

```
GET /accounts/_search
{
     “query”: {
          “wildcard”:{
               “firstname”: {
                    “value”: “t*”
```

Elasticsearch Expensive Query

Search all account index for first name for t*

How will this scale with millions of documents?

Fuzzy Queries

Fuzzy Query

Returns documents that contain terms similar to search terms.

Uses Levenshtein Distance

City: Edenburg

City: Edenburg

City: Edenborg

City: Edenburg

City: Edenburg

City: Edenborg

Replace “bor” → “bur” in Edenburg

```
GET /accounts/_search

{

      "query": {

            "fuzzy":{

                  "city": {

                        "value": "edenborg"

            }

      }

}
```

Example Fuzzy Query

Finds similar terms for "edenborg" within city field

Elasticsearch Regular Expressions

https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-regexp-query.html

```
GET /accounts/_search

{

      "query": {

            "regexp":{

                  "city": {

                        "value": "e.*g"

            }

      }

}
```

Elasticsearch Regular Expression

Finds similar terms for both Edinburg and Edenburg within city field

.

?

+

*

|

- Matches any character
- Repeat the preceding character or one times
- Repeat the preceding character one or more times
- Repeat proceeding character zero or more times.
- Or operator to split left and right side

Full List of Supported Parameters in Documentation!

Building Queries with Ranges

What are all the balances over 10K?

How many accounts do we have over 1 million?

```
GET /accounts/_search
{
“query”: {
    “range”:{
    // greater than, less than,…
 }
}
```

Elasticsearch Range

Returns documents within the specific range

```
“gt”: 10

“gte”: 10

“lt”: 10

“lte”: 10
```

- Greater than 10
- Greater than or equal to 10
- Less than 10
- Less than or equal to 10

Elasticsearch Filters

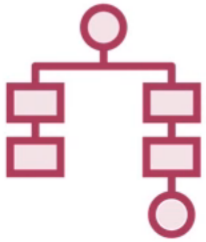
Filter

In Elasticsearch filters give users the ability to query mostly structured terms within DSL query. Example timestamp falling between 2010 – 2011.

Where to Use Filters

Combined with query clause

Asking questions of data with yes/no answers



Structured data flow. Scoring generally not relevant

Does the category equal Women's Shoes?

Does the product ID fall between 6000 – 1200?

Product id	Category	Sku
6283	Men's Clothing	ZO0549605496
11283	Women's Clothing	ZO0489604896
22794	Women's Shoes	ZO0374603746

```
GET /kibana_sample_data_ecommerce/_search
{  "query":
    {    "bool":
      {       "filter": [
              {"term": {"manufacturer": "Pyramidustries"}},
              {"term": {"category": "Women's Clothes"}}

          ] }
    }
}
```

Example Filter

Use Boolean to filter on manufacturer and category on returned results

Elasticsearch Joins

Product id	Category	Sku
6283	Men’s Clothing	ZO0549605496
11283	Women's Clothing	ZO0489604896
22794	Women's Shoes	ZO0374603746

Products

Product id	Manufacturer
6283	New Phase
11283	Nelly Mae’s
22794	Cato

Manufacturer

Join on Product ID

Product id	Category	Sku	Manufacturer
6283	Men’s Clothing	ZO0549605496	New Phase
11283	Women's Clothing	ZO0489604896	Nelly Mae’s
22794	Women's Shoes	ZO0374603746	Cato

SQL Joins Traditional

4 types of SQL Joins

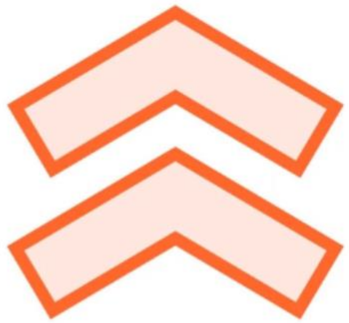
- Inner
- Left
- Right
- Full Outer

Relationships matter

ElasticSearch Joins

Not RDBS

Flat Structure



Nested Approach

Parent-Child

Application Side

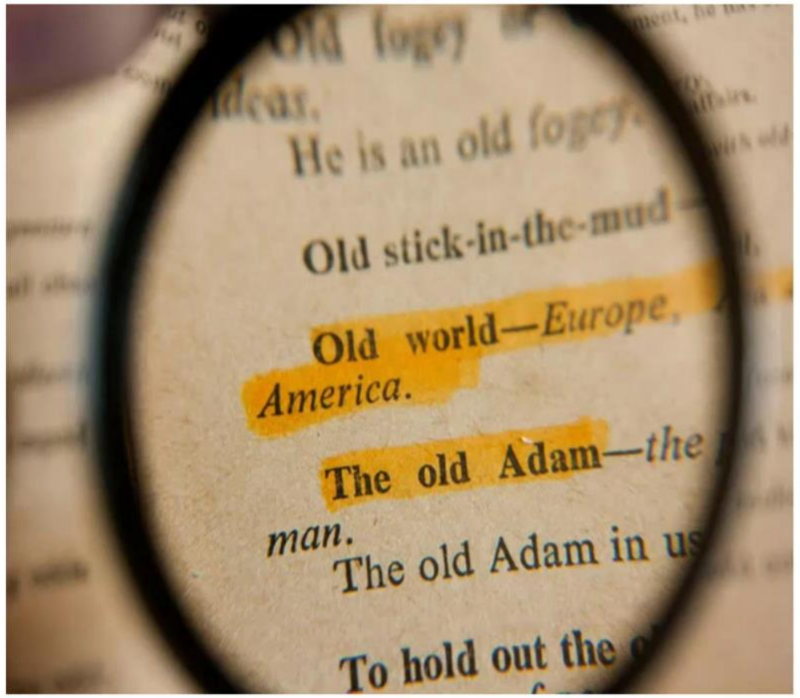
SQL Joins Traditional

4 types of SQL Joins

- Inner
- Left
- Right
- Full Outer

Relationships matter

Full Text Search

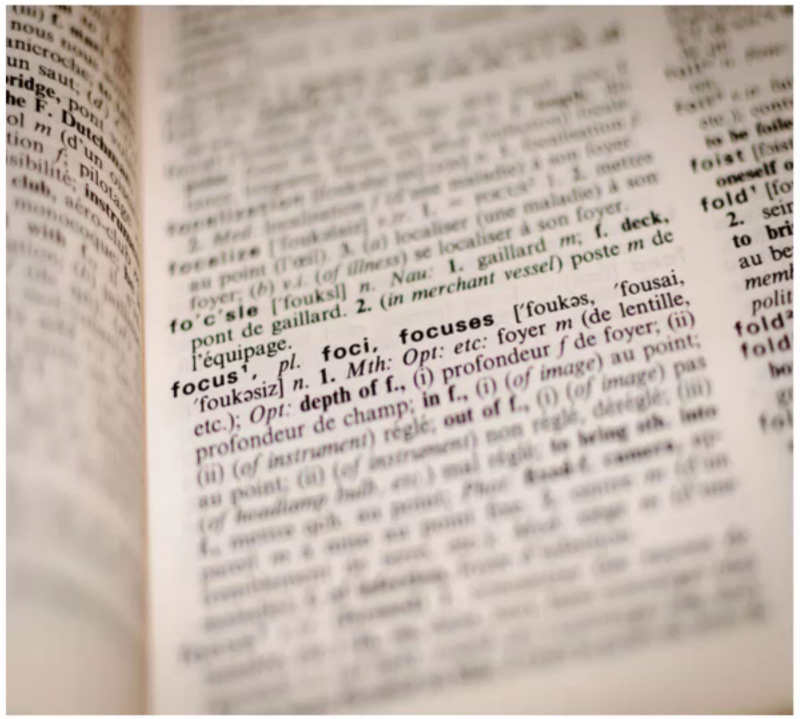


Search matching is term based

It can search for:

- Simple Term
- Prefix Term
- Generation Term
- Proximity Term
- Thesaurus
- Weighted Term

Intervals Query



Uses matching rules constructed from a small set of definitions

For example, to lay down conditions such as:

- Search for a *pattern* after a given *query*
  - "Kids Toys" as a *query* and "Lego Puzzles" as a *pattern*
- Search for a *pattern(**Hot pizzas**)* where two words of the *pattern* are a distance apart
  - "*Hot* food items are pretty good at the new café which got recently inaugurated, especially their *pizzas* are super yum"

Implement a Specific Search-based Query Based on the Given Pattern

Query Analytics based on Matching Pattern

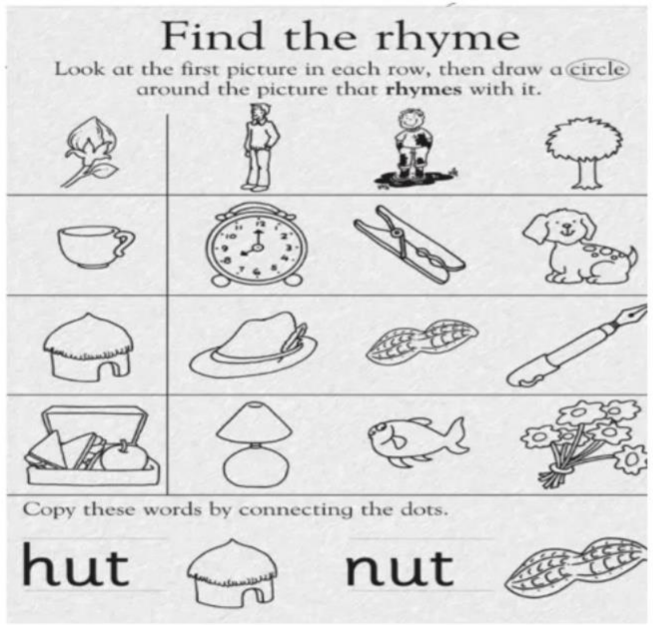
Standard query for performing a full-text search

Apply Boolean conditions to the text search

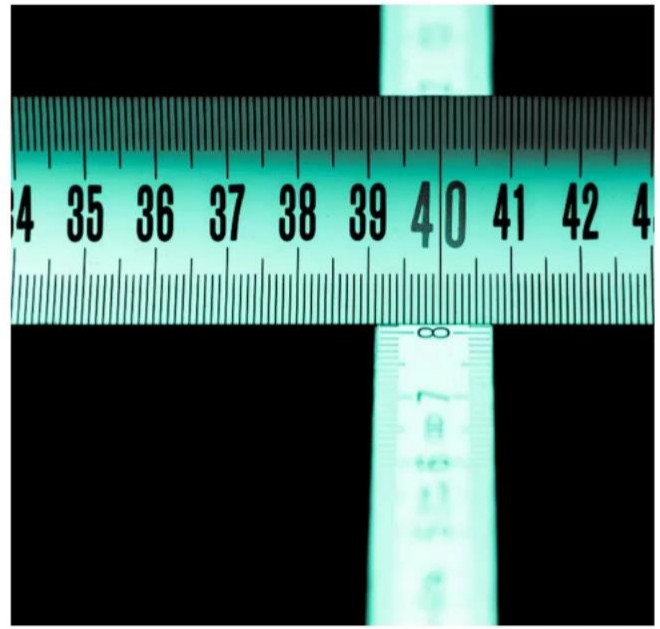
Ignore stop words while searching

Ignore format-based errors while searching

Allow Fuzziness



Transpositions



Levenshtein Edit Distance

Imperfect queries

Levenshtein Edit Distance

String metric for measuring the difference between two sequences

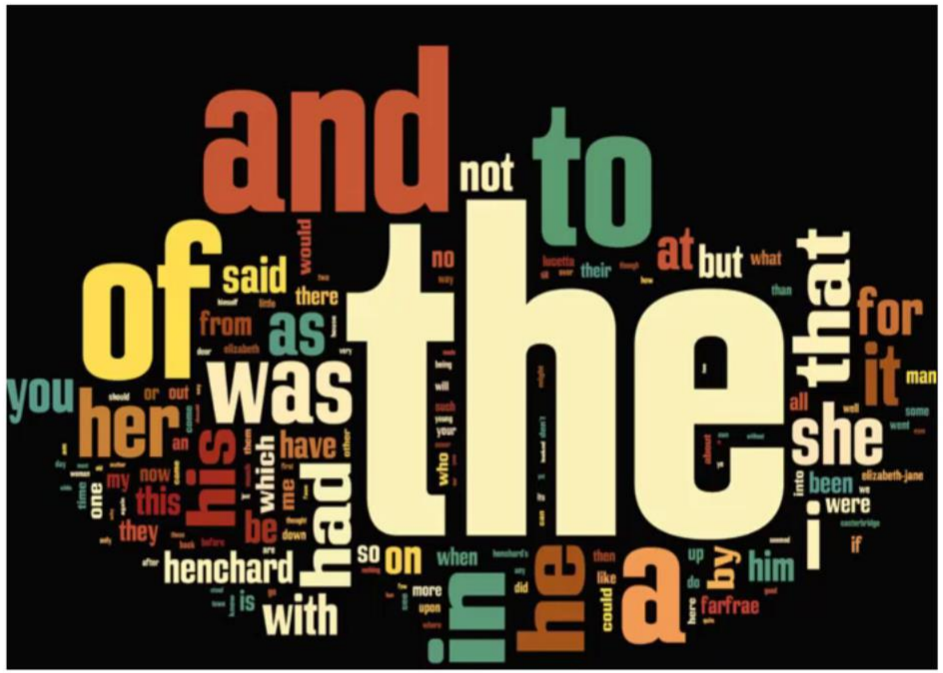
It's defined as the minimum number of character edits (insertions, deletions, or substitutions) required to change one word into another

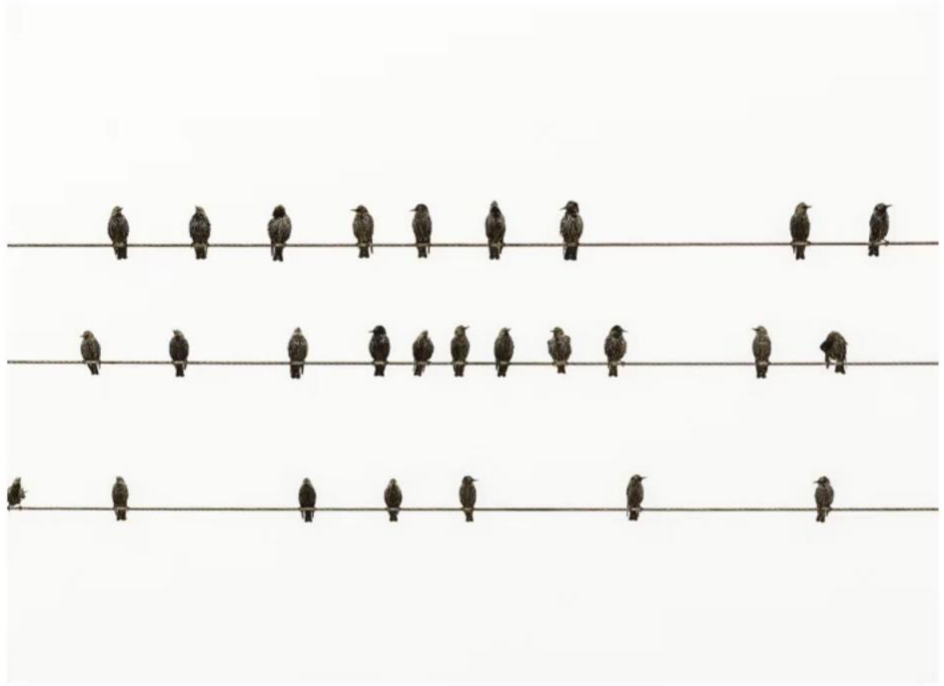
For example, "kitten" → "sitting":

- *k*itten → *s*itten (substitute "s" for "k")
- sitt*e*n → sitt*i*n (substitute "i" for "e")
- sittin → sittin*g* (insert "g" at the end)

So here the distance is 3

Special Parameters for the Search Field





zero_terms_query

Removal of stop words or applying a custom filter with the analyzer

minimum_should_match

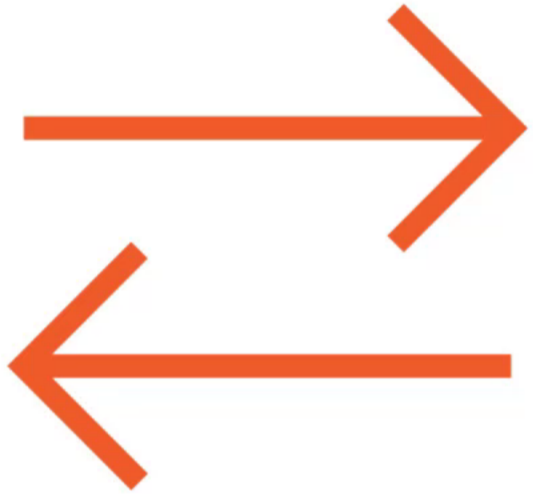
At least specific number of words required to match the query

Extracting Multiple Patterns Using Multi-match Querying Mechanism

Custom Search for Multi-match Query

Weightage

Importance given to specific fields while searching



Cross-field

All terms must be present in at least one field for a document to match

Boolean Combinations

The more the merrier☺

Phrase & Phrase Prefix Query

Match Phrase

A phrase query that matches up to a configurable slop in any order

Slop parameter tells the match_phrase query how far apart terms are allowed to be

Match Phrase Prefix

Last term of the provided phrase is considered to be pre-fix

Example:

"quick brown f" will match "quick brown fox" or "two quick brown ferrets"

Slop

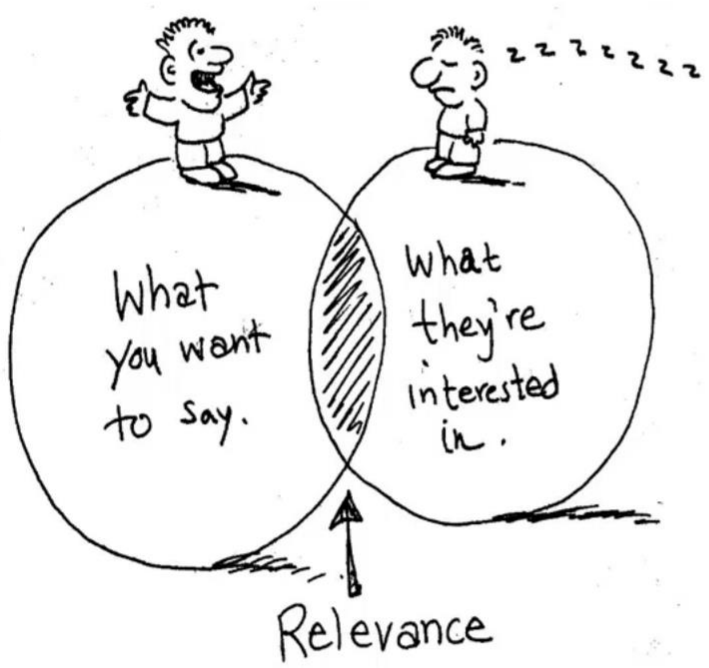
How far apart the terms are allowed to be

For the query “hot pizza” matching a document “hot freshly baked pizza” we need a *slop* of 2 in either way so that the re-order of terms would look like:

- “freshly baked *hot* pizza”
- “hot *pizza* freshly baked”

Either the term hot or pizza could be shifted two places

Relevant Score



Measures how well each document matches a query

Returned as _score metadata field

Higher the _score, more relevant the document is

tie_breaker can be used to create a custom score

Implementing Common Terms Querying Mechanism

Common Terms Query

Relevance

Based on the relevance score calculation the search results appear

Domain-Specific

Youtube Stop Words – “Like”, ”Share”,”Subscribe”

Cutoff Frequency

Terms are allocated to the high or low frequency groups

Each term is associated with the cost

Brute force approach of removing stop words is not always a good idea

- Unable to distinguish between “good” and “not good”

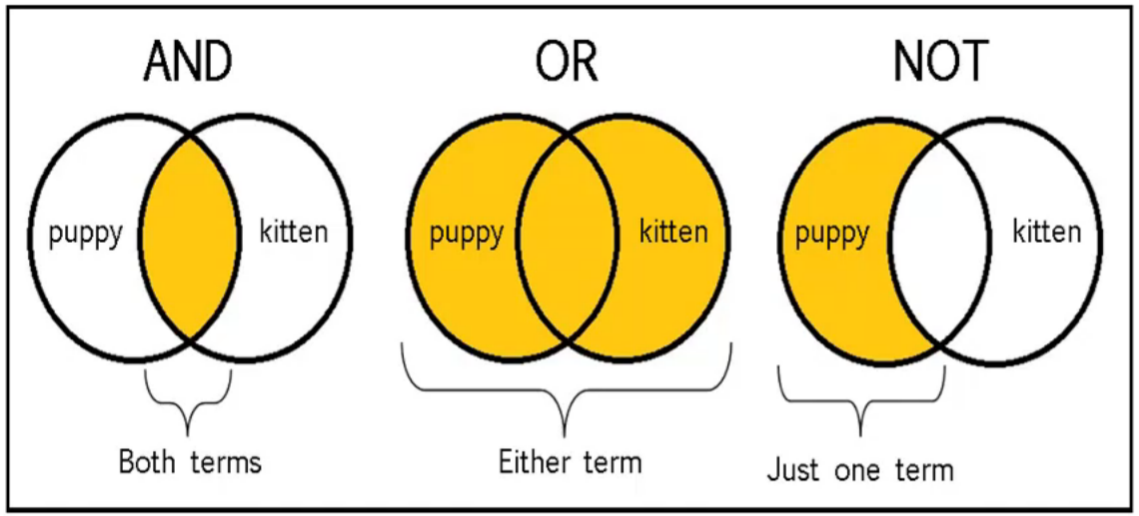
Common terms is the solution:

- It divides the query terms into two groups “more important” and “less important”

Properties like *minimum_should_match* and *cutoff_frequency* helps in fine-tuning

Discovering Boolean patterns within the String Based Search

Query String Search: Parameters



Regular Expression

Pattern based Search

Conditional Search

Search with Boolean conditions