

Agenda

- ▶ Tuesday, the 11th, morning (3h)
 - ▶ General information on NoSQL database
 - ▶ Installation of what is required
 - ▶ Discover the cluster
- ▶ Wednesday, the 12th, morning (3h)
 - ▶ Interacting with the cluster
- ▶ Wednesday, the 12th, afternoon (3h)
 - ▶ Personal manipulation, use case.

Understanding NoSQL Databases

A Comprehensive Introduction

Introduction to NoSQL Databases

► Définition

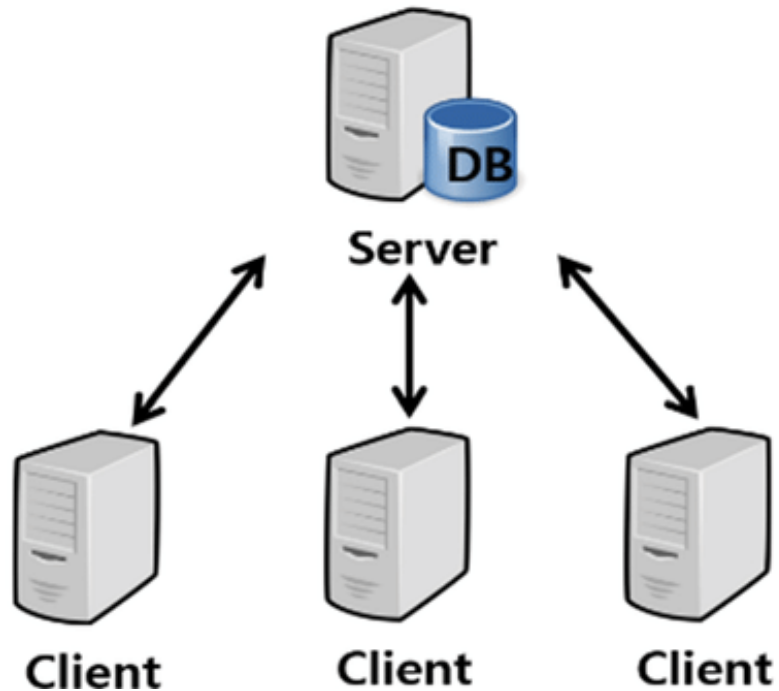
- « Not Only SQL » Designed for high scalability and increased flexibility in handling various data types.

► Context of Emergence

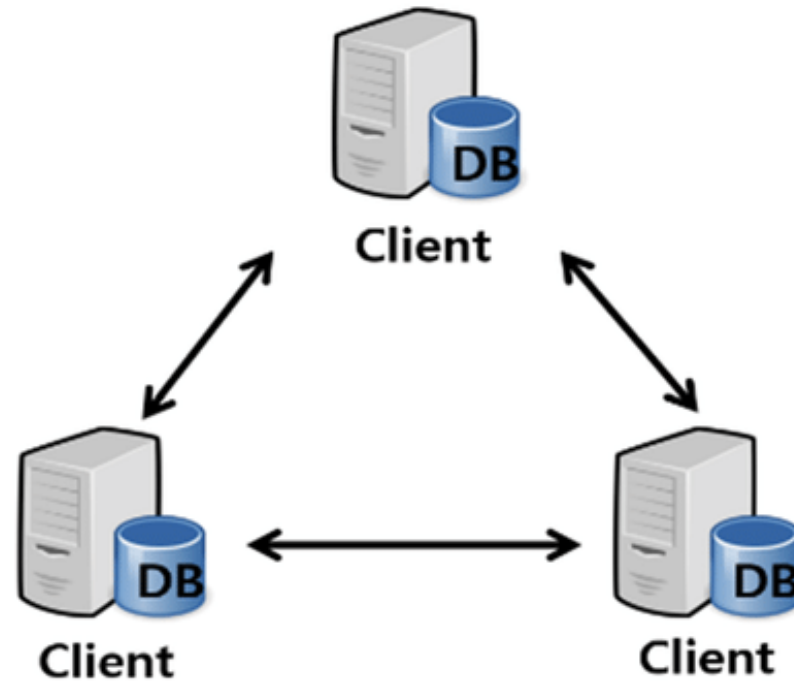
- emerged in the 2000s
- horizontal scalability
- large volumes of diverse data and high traffic

Centralized / Versus Distributed

Centralized



Distributed (Blockchain)



Scalability

- The scalability of an application is the measure of the number of client requests it can simultaneously handle



	Vertical	Horizontal
Data	Data is executed on a single node	Data is partitioned and run on multiple nodes
Management	Easy to manage, share data reference	Complex task as there is no shared address space
Downtime	Downtime while upgrading the machine	No downtime
Upper limit	Limited by machine specifications	Not limited by machine specification
Cost	Lower license Higher material	Higher licence Lower Material

Redefinition of Databases

- ▶ Redefinition of ACID Constraint to BASE
 - ▶ ACID
 - ▶ Atomicity
 - ▶ Consistency
 - ▶ Isolation
 - ▶ Durability
 - ▶ BASE
 - ▶ Basically Available
 - ▶ Soft State
 - ▶ Eventual Consistency

Greater flexibility and improved performance in distributed environments by somewhat sacrificing immediate consistency for availability and fault tolerance.

SQL vs. NoSQL Query Language

- ▶ SQL (Structured Query Language) in Traditional Databases
 - ▶ Complex querying
 - ▶ Data Integrity
 - ▶ Scalability and Flexibility
 - ▶ Standardization
 - ▶ Security
- ▶ Why NoSQL Databases Do Not Always Use SQL
 - ▶ Schema-less Data
 - ▶ Scalability concerns
 - ▶ Different Data Access Patterns
 - ▶ Performance optimization

Requête SQL

- ▶ Database with two tables

- ▶ Clients (id_client, nom, email)
- ▶ Commandes (id_commande, id_client, montant, date_commande)

```
SELECT p.categorie,  
COUNT(c.id_commande) AS  
nombre_commandes, SUM(c.montant) AS  
total_montant  
FROM Commandes c  
JOIN Produits p ON c.id_commande =  
p.id_commande  
WHERE c.status = 'livré'  
AND c.date_commande BETWEEN '2024-01-  
01' AND '2024-12-31'  
GROUP BY p.categorie  
ORDER BY total_montant DESC;
```

catégorie	nombre_commandes	total_montant
Électronique	120	50000
Maison	80	32000
Vêtements	60	15000

- ▶ Join (JOIN Produits p ON c.order_id = p.order_id)
 - ▶ Retrieves product categories associated with orders.
- ▶ Filtering (WHERE c.status = 'delivered' AND order_date BETWEEN '2024-01-01' AND '2024-12-31')
 - ▶ Selects only orders from 2024 that have been delivered.
- ▶ Aggregation (GROUP BY p.categorie) Groups orders by product category. Count (COUNT(c.id_commande) AS nombre_commandes)
 - ▶ Returns the number of orders per category.
- ▶ Sum (SUM(c.amount) AS total_amount)
 - ▶ Calculates total sales by category.
- ▶ Sort (ORDER BY total_amount DESC)
 - ▶ Sort results from highest to lowest total amount. Translated with DeepL.com (free version)

Requête NoSQL

```
{
  "size": 0,
  "query": {
    "bool": {
      "filter": [
        { "term": { "status": "livré" } },
        { "range": { "date_commande": { "gte": "2024-01-01", "lte": "2024-12-31" } } }
      ]
    }
  },
  "aggs": {
    "par_categorie": {
      "terms": {
        "field": "produits.categorie.keyword",
        "size": 10
      },
      "aggs": {
        "total_montant": {
          "sum": {
            "field": "montant"
          }
        }
      }
    }
  }
}
```



```
{
  "aggregations": {
    "par_categorie": {
      "buckets": [
        { "key": "Électronique", "doc_count": 120, "total_montant": { "value": 50000 } },
        { "key": "Maison", "doc_count": 80, "total_montant": { "value": 32000 } },
        { "key": "Vêtements", "doc_count": 60, "total_montant": { "value": 15000 } }
      ]
    }
  }
}
```



Differences from Relational Databases

- ▶ Data Storage Structure
 - ▶ Relational databases use a tabular structure with fixed schemas
 - ▶ NoSQL databases allow for a variety of storage structures
 - ▶ Key/Value
 - ▶ Columns
 - ▶ Graph
 - ▶ Documents

Database Structure : Traditionnal

- ▶ Relational databases use a structured data storage model organized into tables. Each table is defined by a schema that prescribes the structure and constraints of the data:
 - ▶ **Tables:** Each table represents a type of entity (like customers, orders, etc)
 - ▶ **Schemas:** The schema of a relational database defines the columns of each table Schemas are fixed.
 - ▶ **Constraints:** Relational databases enforce data integrity through constraints such as primary keys, foreign keys, and unique constraints.

Database Structure : NoSQL

Key-Value Stores: These are the simplest form of NoSQL databases, designed for storing data as a collection of key-value pairs. They are highly efficient for scenarios where quick access to data is required, such as in caching and session storage, where the key is known.

Column Stores: Columnar databases store data in columns rather than rows, making them ideal for analytics and data warehousing scenarios where operations are typically performed on column data. This structure allows for efficient data compression and fast aggregation.

Graph Databases: These databases are optimized for storing and querying data that is interconnected, making them suitable for social networks, recommendation engines, and fraud detection systems where relationships between data points are crucial.

Document Stores: Document-oriented databases store data in document formats like JSON, XML, etc. They are versatile and flexible, making them suitable for content management systems and e-commerce applications where each document can be unique and evolve over time.

How to choose the correct database ?

ORACLE

mongoDB



Azure



Microsoft
SQL Server



IBM
DB2



redis



SAP HANA



Key-Value Databases (Redis)

What is a Key-Value Database?

- - A NoSQL database storing data as key-value pairs
- - Highly optimized for fast lookups using keys
- - Data structures: strings, lists, sets, hashes

Why Use Redis?

- ✓ Ultra-fast in-memory storage (low-latency operations)
- ✓ Supports caching, pub/sub, and session storage
- ✓ Can persist data with snapshotting & AOF logs
- ✓ Scales horizontally using clustering

When Not to Use Redis?

- ✗ Complex queries & relationships needed (Use relational/graph DB)
- ✗ Large datasets exceeding RAM capacity

Graph Databases (Neo4j)

What is a Graph Database?

- - A database designed for relationships between data
- - Stores data as nodes (entities) and edges (relationships)
- - Uses graph traversal algorithms for queries

Why Use Neo4j?

- ✓ Perfect for connected data (e.g., social networks, fraud detection)
- ✓ Fast relationship-based queries
- ✓ Uses Cypher Query Language (CQL) for intuitive graph operations
- ✓ Supports deep link analysis

When Not to Use Neo4j?

- ✗ Flat/tabular data structure needed (Use SQL/NoSQL DB)
- ✗ Heavy transactional workloads with strict ACID compliance

Time-Series Databases (InfluxDB)

What is a Time-Series Database?

- - A database optimized for storing & querying time-stamped data
- - Handles high-ingestion rates efficiently
- - Stores metrics, events, logs, and monitoring data

Why Use InfluxDB?

- ✓ Optimized for time-based queries
- ✓ Supports real-time analytics & monitoring
- ✓ Efficient data compression & retention policies
- ✓ SQL-like query language (InfluxQL, Flux)

When Not to Use InfluxDB?

- ✗ Relational data models needed (Use SQL DB)
- ✗ Transactional consistency required

Document Database : OpenSearch

- ▶ Most popular document database : MongoDB
- ▶ OpenSearch is related somehow to ElasticSearch
- ▶ Introduction to Opensearch
 - ▶ Open-source search and analytics engine
 - ▶ Forked from Elasticsearch 7.10 after Elastic changed its license
 - ▶ Maintained by AWS and a growing open-source community
 - ▶ Used for full-text search, log analytics, and observability

The background is a solid light green color. On the right side, there are several overlapping, semi-transparent geometric shapes in various shades of green, creating a dynamic, abstract design. These shapes include triangles and quadrilaterals that intersect to form a complex pattern. A thin white line also runs diagonally across the lower right portion of the image.

Focus on Opensearch

History & Origin

- ▶ 2004: Elasticsearch was created as a distributed search engine based on Apache Lucene
- ▶ 2021: Elastic changed its license from Apache 2.0 to SSPL (Server Side Public License)
- ▶ 2021: AWS forked Elasticsearch 7.10, creating OpenSearch to keep it open-source
- ▶ Present: OpenSearch continues to be developed independently with its own features and governance



Key Differences Between OpenSearch & Elasticsearch

Feature	OpenSearch	Elasticsearch (post 7.10)
License	Apache 2.0	SSPL (not fully open-source)
Governance	Open community-driven	Elastic company-led
Security features	Built-in (free)	Paid in Elastic stack
Machine Learning	Open-source ML plugins	Paid X-Pack ML
Future Development	Independent roadmap	Elastic-controlled

OpenSearch Roadmap

- ▶ Scalability improvements: More efficient indexing and search performance
- ▶ Better observability: Native support for logs, traces, and metrics
- ▶ Security enhancements: Role-based access control, authentication integrations
- ▶ Machine Learning & AI: Expanded ML capabilities for anomaly detection
- ▶ Kibana replacement: OpenSearch Dashboards evolving with better visualization tools

When and Where to Use OpenSearch

► Best Use Cases

- ✓ Log & Event Analytics
 - Centralized logging (ELK alternative)
 - Security monitoring (SIEM)
 - Infrastructure observability
- ✓ Full-Text Search
 - E-commerce product search
 - Website & application search engines
 - Document indexing and retrieval
- ✓ Monitoring & Observability
 - Metrics, traces, and logs analysis
 - Performance monitoring (APM replacement)
- ✓ Business Intelligence & Data Exploration
 - Real-time dashboards and analytics
 - Large-scale data visualization

► When Not to Use OpenSearch?

- ✗ Strong ACID Compliance Needed → Use relational databases (e.g., PostgreSQL, MySQL)
- ✗ Transactional Workloads → Not optimized for OLTP operations
- ✗ Graph Database Needs → Use Neo4j or AWS Neptune

Opensearch Platform

Opensearch Platform



Build freely with OpenSearch's integrated components

Data Prepper



Ingest and enrich your data through custom pipelines

OpenSearch Core



Utilize our versatile search and analytics engine

OpenSearch Dashboards



Gather meaning to drive insights

What is a document

Data is stored as a document, as a **json** object with a unique Id

```
{  
  "_id": 2,  
  "name": "Jane Smith",  
  "email": "jane.smith@example.com",  
  "age": 25  
}
```

Index

Documents that are related to each others are grouped into index

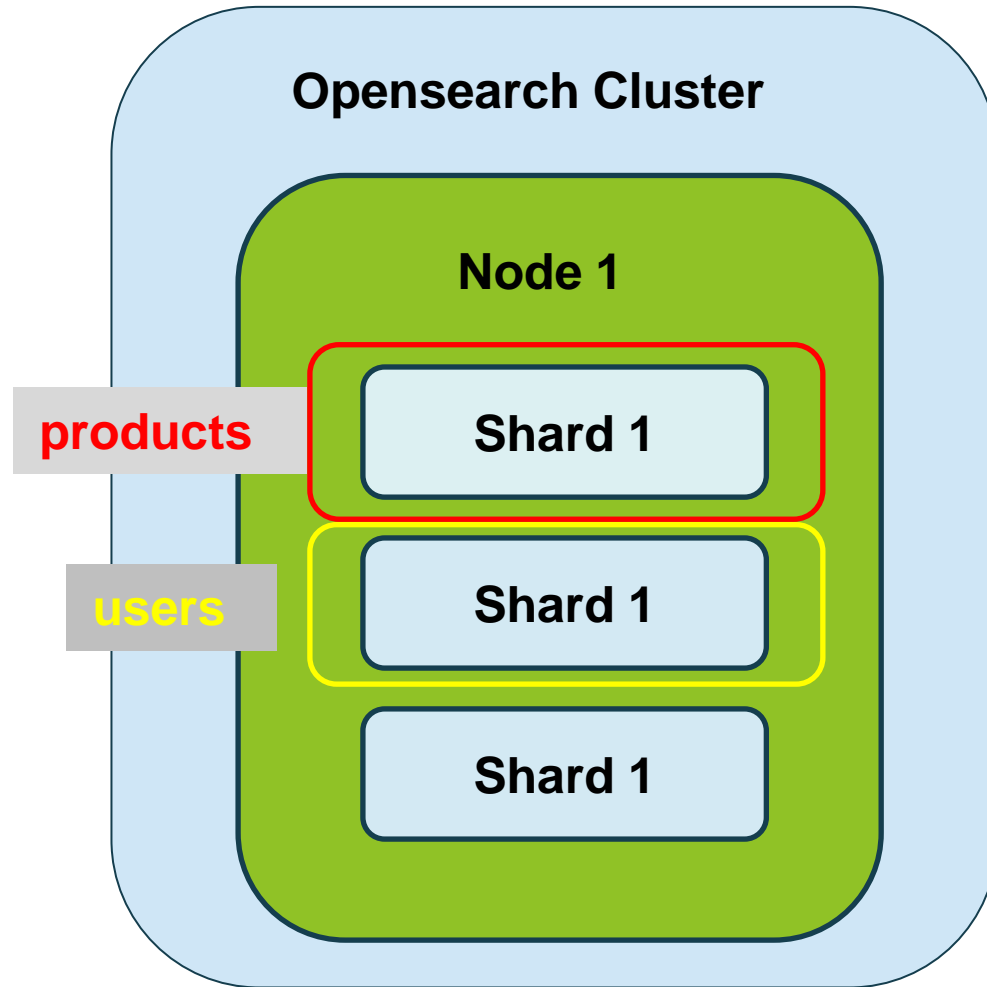
Index : products

```
{
  "_id": 1,
  "name": "Smartphone",
  "price": 800,
  "brand": "XYZ Tech
}
{
  "_id": 2,
  "name": "Camera",
  "price": 500
  "brand": "CameroCo"
}
```

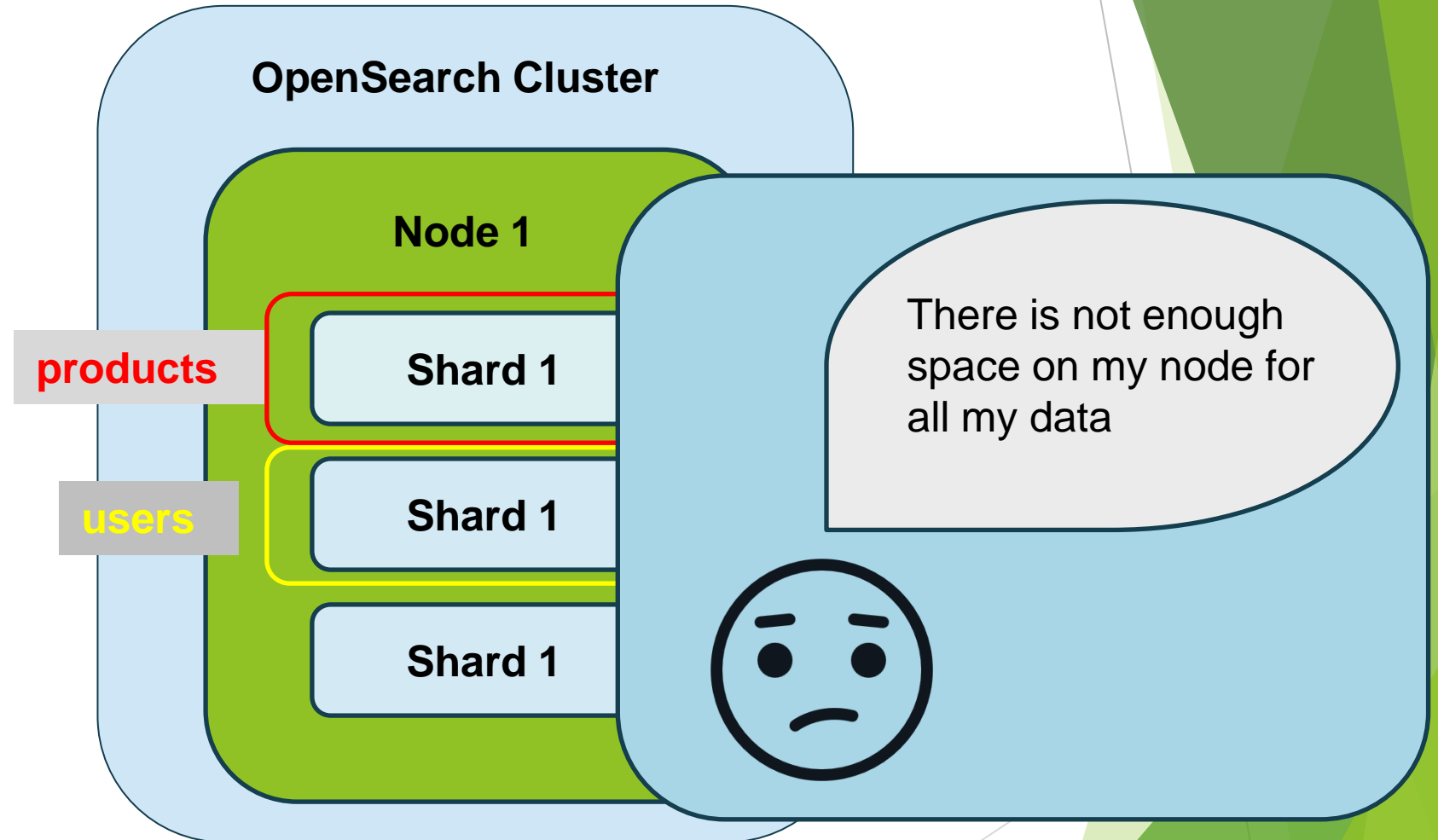
Index : users

```
{
  "_id": 2,
  "name": "Jane Smith",
  "email": "jane.smith@example.com",
  "age": 25
}
{
  "_id": 3,
  "name": "Tom Durand",
  "email": "tom.durand@example.com",
  "age": 42
}
```

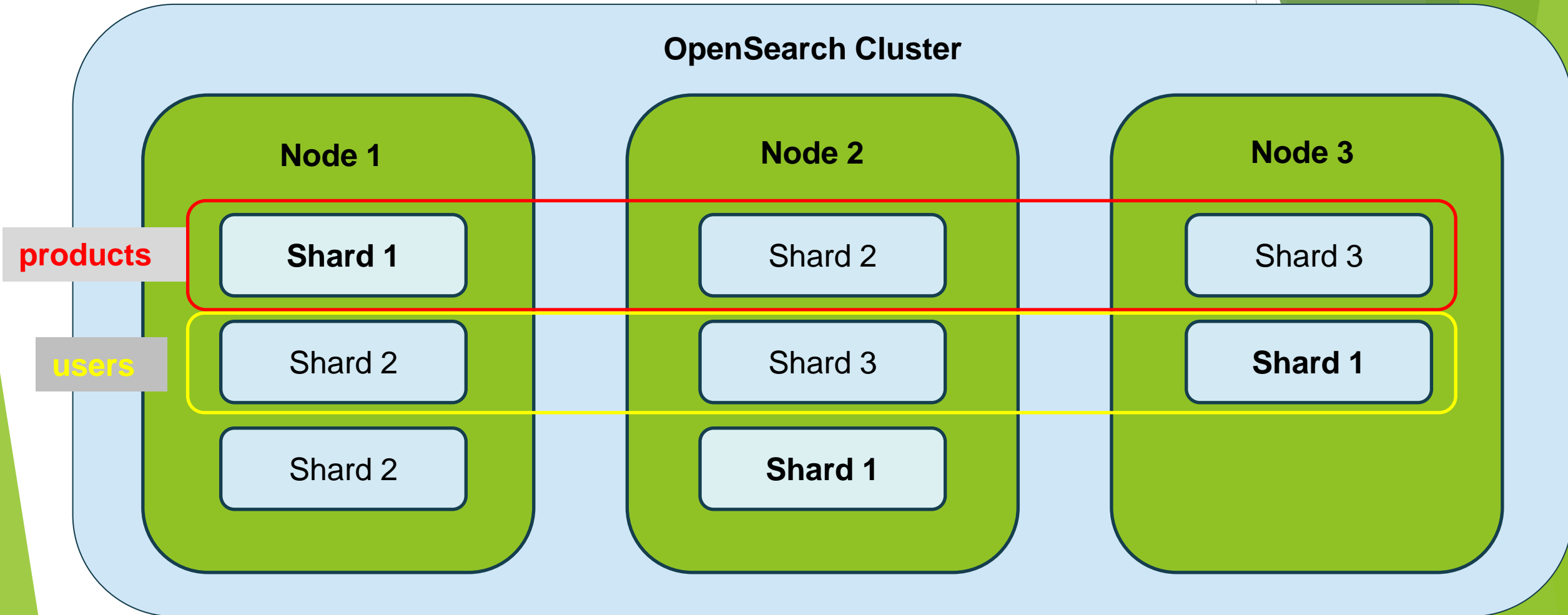
Opensearch Cluster



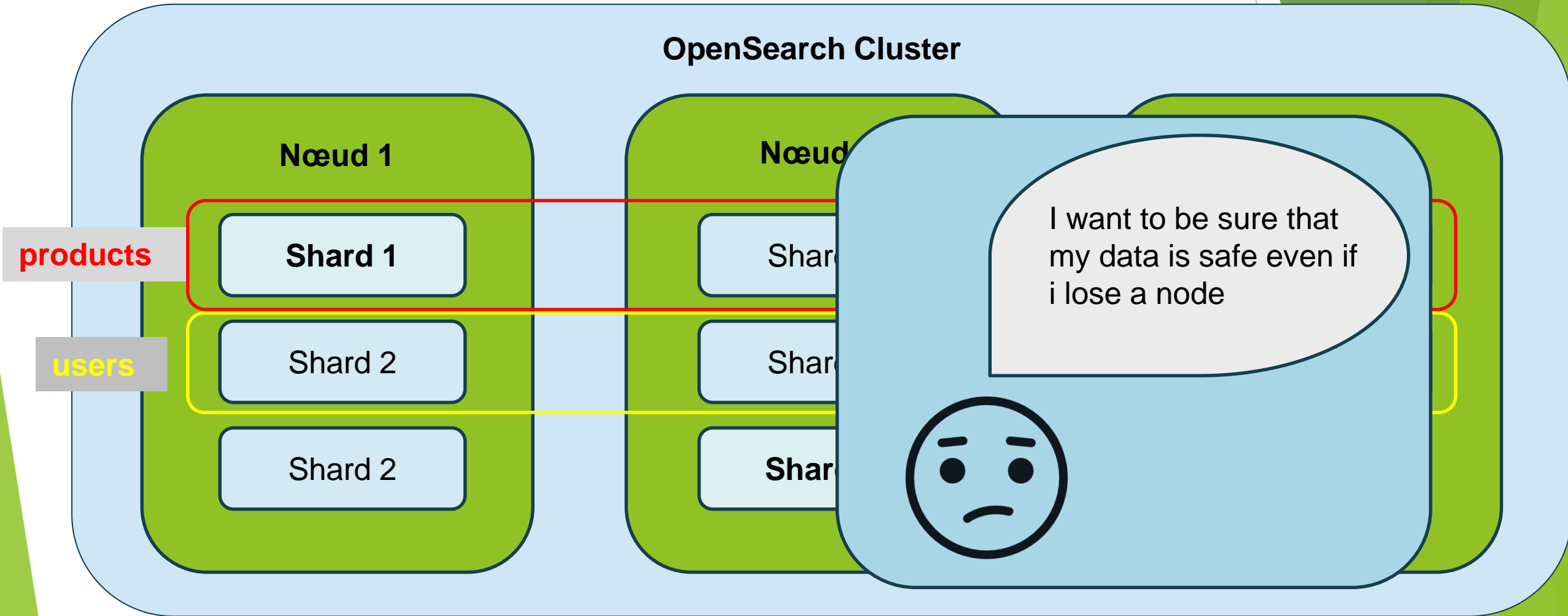
OpenSearch Cluster



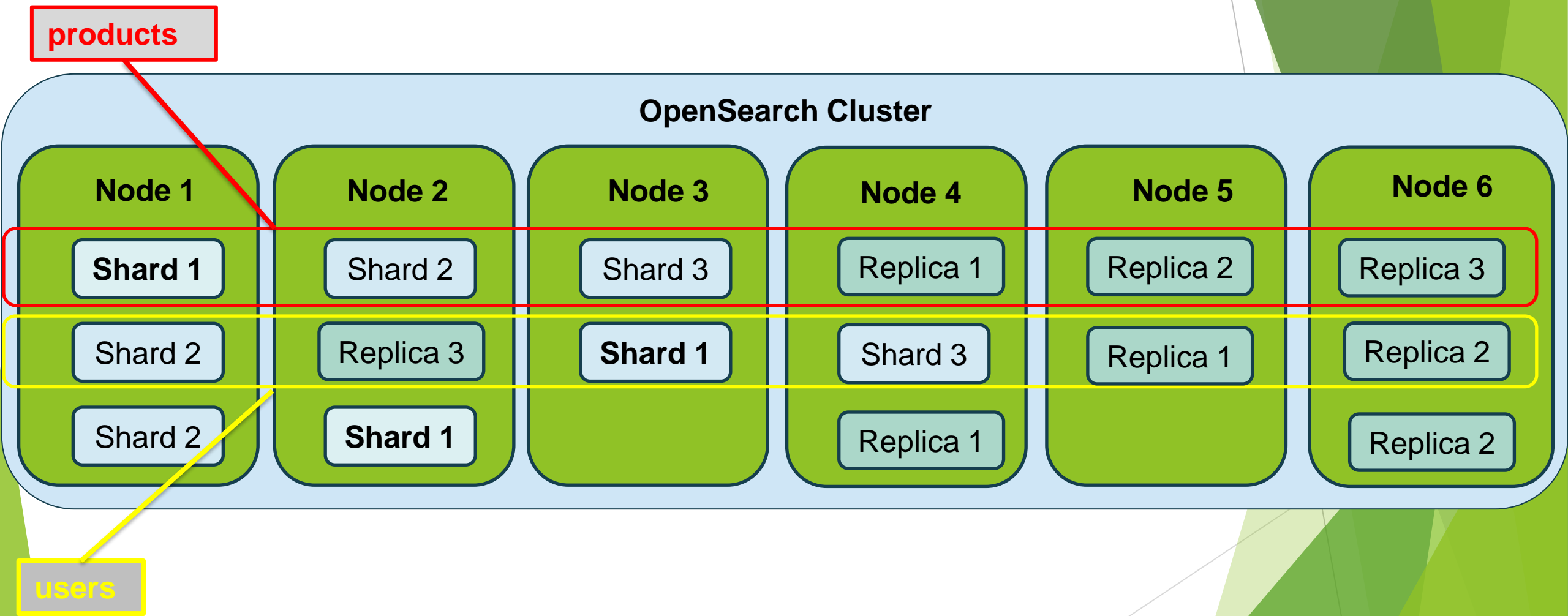
OpenSearch Cluster



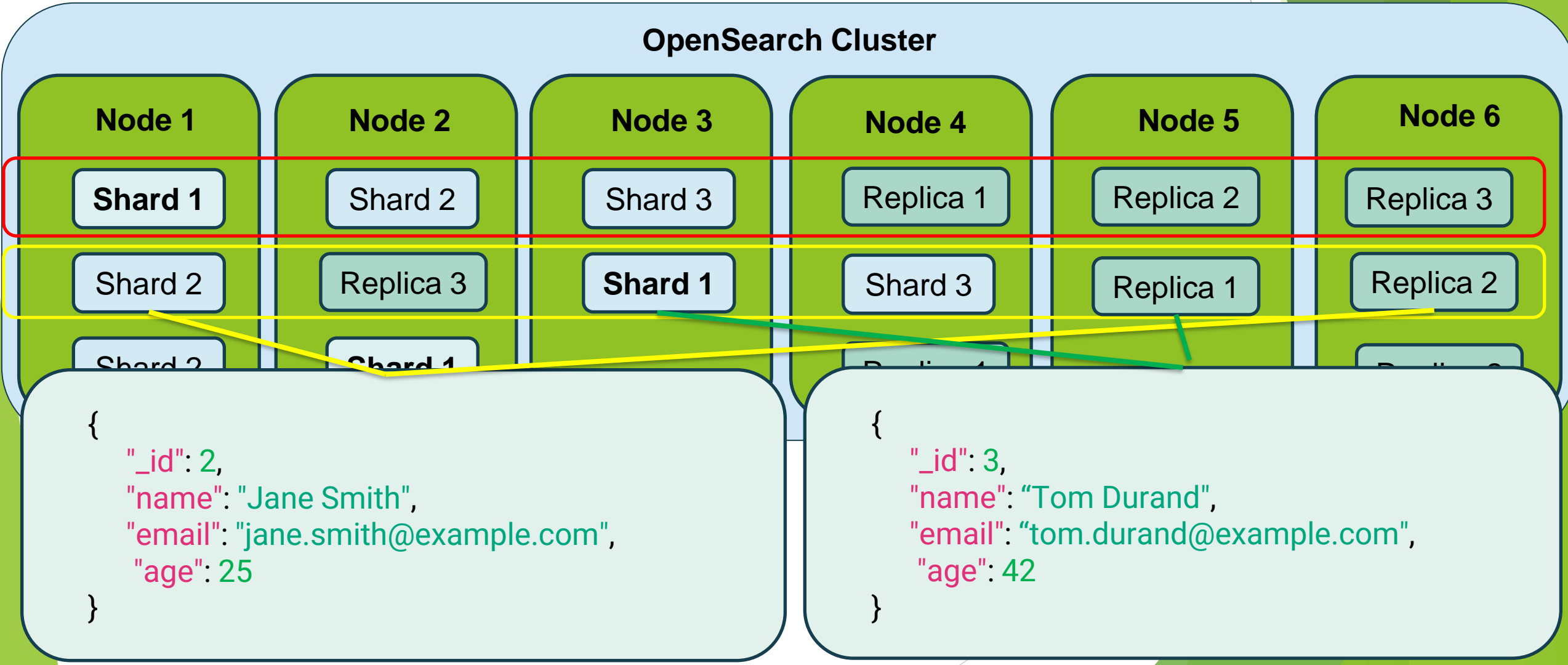
OpenSearch Cluster



OpenSearch Cluster



OpenSearch Cluster



Opensearch Cluster



OpenSearch operates as a distributed system, and its primary components include nodes, shards, and replicas.



Nodes: Individual instances of OpenSearch that make up the cluster.



Shards: Divisions of data within an index that allow for parallelism and scalability.



Replicas: Duplicate copies of primary shards to ensure data availability and fault tolerance.

Interact With the Cluster

- ▶ CRUD operations
- ▶ Filters and Queries
- ▶ Mapping
- ▶ Full Text Queries
- ▶ Geospatial Data

<https://github.com/codingexplained/complete-guide-to-elasticsearch/tree/master>

CRUD operations



C - Create

Syntax:

PUT Name-of-the-Index



Create an index

Example:

PUT favorite_candy

CRUD operations

C - Create

Index a document

When indexing a document, both HTTP verbs POST or PUT can be used. Use POST when you want Opensearch to autogenerate an id for your document.

Syntax:

```
POST Name-of-the-Index/_doc
{
  "field": "value"
}
```

Example:

```
POST favorite_candy/_doc
{
  "first_name": "Lisa",
  "candy": "Sour Skittles"
}
```

CRUD operations

C - Create

Index a document

Use PUT when you want to assign a specific id to your document (i.e. if your document has a natural identifier - purchase order number, patient id, & etc).

Syntax:

```
PUT Name-of-the-Index/_doc/your-id
{
  "field": "value"
}
```

Example:

```
PUT favorite_candy/_doc/1
{
  "first_name": "John",
  "candy": "Starburst"
}
```

CRUD operations

R - READ

Read a document

Syntax:

GET Name-of-the-Index/_doc/id

Example:

GET favorite_candy/_doc/1

CRUD operations

U - UPDATE

Update a document

Syntax:

```
POST Name-of-the-Index/_update/id
{
  "doc": {
    "field1": "value",
    "field2": "value",
  }
}
```

Example:

```
POST favorite_candy/_update/1
{
  "doc": {
    "candy": "M&M's"
  }
}
```

CRUD operations

D - DELETE

Delete a document

Syntax:

DELETE Name-of-the-Index/_doc/id

Example:

DELETE favorite_candy/_doc/1

CRUD operations

C - Create

Index a document

When you index a document using an id that already exists, the existing document is overwritten by the new document. If you do not want an existing document to be overwritten, you can use the `_create` endpoint!

With the `_create` Endpoint, no indexing will occur and you will get a 409 error message.

Syntax:

```
PUT Name-of-the-Index/_create/your-id
{
  "field": "value"
}
```

Example:

```
PUT favorite_candy/_create/1
{
  "first_name": "John",
  "candy": "Starburst"
}
```



Queries

Queries

The query DSL is a flexible, expressive search language that Opensearch uses to expose most of the power of Lucene through a simple JSON interface.

Syntax:

```
GET name-of-the-Index/_search
{
  "query": YOUR_QUERY_HERE
}
```

Query structures

A query clause typically has this structure:

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

If it references one particular field, it has this structure:

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

Example:

GET favorite_candy/_search

```
{  
  "query": {  
    "match": {  
      "candy": "M&M's"  
    }  
  }  
}
```

Create complex queries

Query clauses are simple building blocks that can be combined with each other to create complex queries.

- ▶ *Leaf clauses* (like the match clause) that are used to compare a field to a query string.
- ▶ *Compound clauses* that are used to combine other query clauses. For instance, a bool clause allows you to combine other clauses that either must match, must_not match, or should match if possible:

Example:

GET favorite_candy/_search

```
{  
  "bool": {  
    "must": { "match": { "candy": "M&M's" } },  
    "must_not": { "match": { "first_name": "mary" } },  
  }  
}
```

Compound clause can combine *any* other query clauses, including other compound clauses. This means that compound clauses can be nested within each other, allowing the expression of very complex logic.

Query vs Filter

Filter asks a yes|no question of every document and is used for fields that contain exact values.

A **query** is similar to a filter, but also asks the question: How well does this document match?

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.

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

Most common queries and filters

term Filter

The term filter is used to filter by exact values, be they numbers, dates, Booleans, or not_analyzed exact-value string fields:

```
{
  { "term": { "age": 26 }}
  { "term": { "date": "2014-09-01" }}
  { "term": { "public": true }}
  { "term": { "tag": "full_text" }}
}
```


Most common queries and filters

terms Filter

The terms filter is the same as the term filter, but allows you to specify multiple values to match. If the field contains any of the specified values, the document matches:

```
{  
  "terms": { "tag": [ "search", "full_text", "nosql" ] }  
}
```

Most common queries and filters

range Filter

The `range` filter allows you to find numbers or dates that fall into a specified range:

```
{  
  "range": {  
    "age": {  
      "gte": 20,  
      "lt": 30  
    }  
  }  
}
```

The operators that it accepts are as follows:

`gt` : Greater than

`gte` : Greater than or equal to

`lt` : Less than

`lte` : Less than or equal to

Most common queries and filters

exists and missing Filters

The exists and missing filters are used to find documents in which the specified field either has one or more values (exists) or doesn't have any values (missing)

```
{  
  "exists": {  
    "field": "title"  
  }  
}
```

Most common queries and filters

Bool Filters

The bool filter is used to combine multiple filter clauses using Boolean logic. It accepts three parameters:

`must` : These clauses must match, like `and`.

`must_not` : These clauses must not match, like `not`.

`should` : At least one of these clauses must match, like `or`.

Search for information

There are two main ways to search in Opensearch:

Queries : to retrieve documents that match the criteria.

Aggregations : to summarizes your data as metrics, statistics, and other analytics.

Queries

Syntax:

```
GET name_of_the_index/_search
{
  "query": {
    "Specify the type of query here": {
      "Enter name of the field here": {
        "gte": "Enter lowest value of the range here",
        "lte": "Enter highest value of the range here"
      }
    }
  }
}
```

Opensearch displays a number of hits and a sample of 10 search results by default.

Example:

```
GET favorite_candy/_search
```

Index mappings

Mapping

- ▶ Mapping is the process of defining the fields/structure that documents follow and how they are stored and indexed on ES.
- ▶ Mapping is generally used to define :
 - ▶ which string fields should be treated as full text fields
 - ▶ Which fields contain numbers, dates or geolocation
 - ▶ Whether the values of all fields in the document should be indexed in the `_all` field
 - ▶ Date format
 - ▶ Etc.
- ▶ Each field has a data type, which can be :
 - ▶ A simple data type such as string, date, long, double, boolean or ip.
 - ▶ A type that supports the hierarchical nature of JSON, such as object or nested.
 - ▶ A special type such as `geo_point`, `geo_shape` or `completion`.

Mapping

Create a new index with a mapping :


```
PUT /my_index?pretty
{
  "settings": {
    "number_of_shards": 1
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "age": {
        "type": "integer"
      }
    }
  }
}
```

Create a mapping on an existing index :

```
PUT /my_index/_mapping?pretty
{
  "properties": {
    "email": {
      "type": "keyword"
    }
  }
}
```

Retrieve the mapping :

Curl -XGET localhost:9200/my_index/_mapping

The background features a series of overlapping, semi-transparent green triangles and polygons that create a dynamic, layered effect. The colors range from a light, pale green to a deep, forest green. The shapes are primarily oriented diagonally, with some pointing towards the top right and others towards the bottom left. The overall composition is modern and minimalist.

Full text
queries

Information Retrieval

Applications

Web search, used daily by billions of users

Searching messages in your mailbox

Searching for files on your computer

Searching for sources in a public or private document database

etc.



Information Retrieval (IR) is the process of finding documents in a large documents with little or no structure, in a large collection, according to an information need.



IR develops models for :



interpret documents and needs, and bring them together



calculate answers quickly, even in the presence of very large collections



search engine" systems providing sophisticated, ready-to-use solutions.

Lemmatization & Stemming

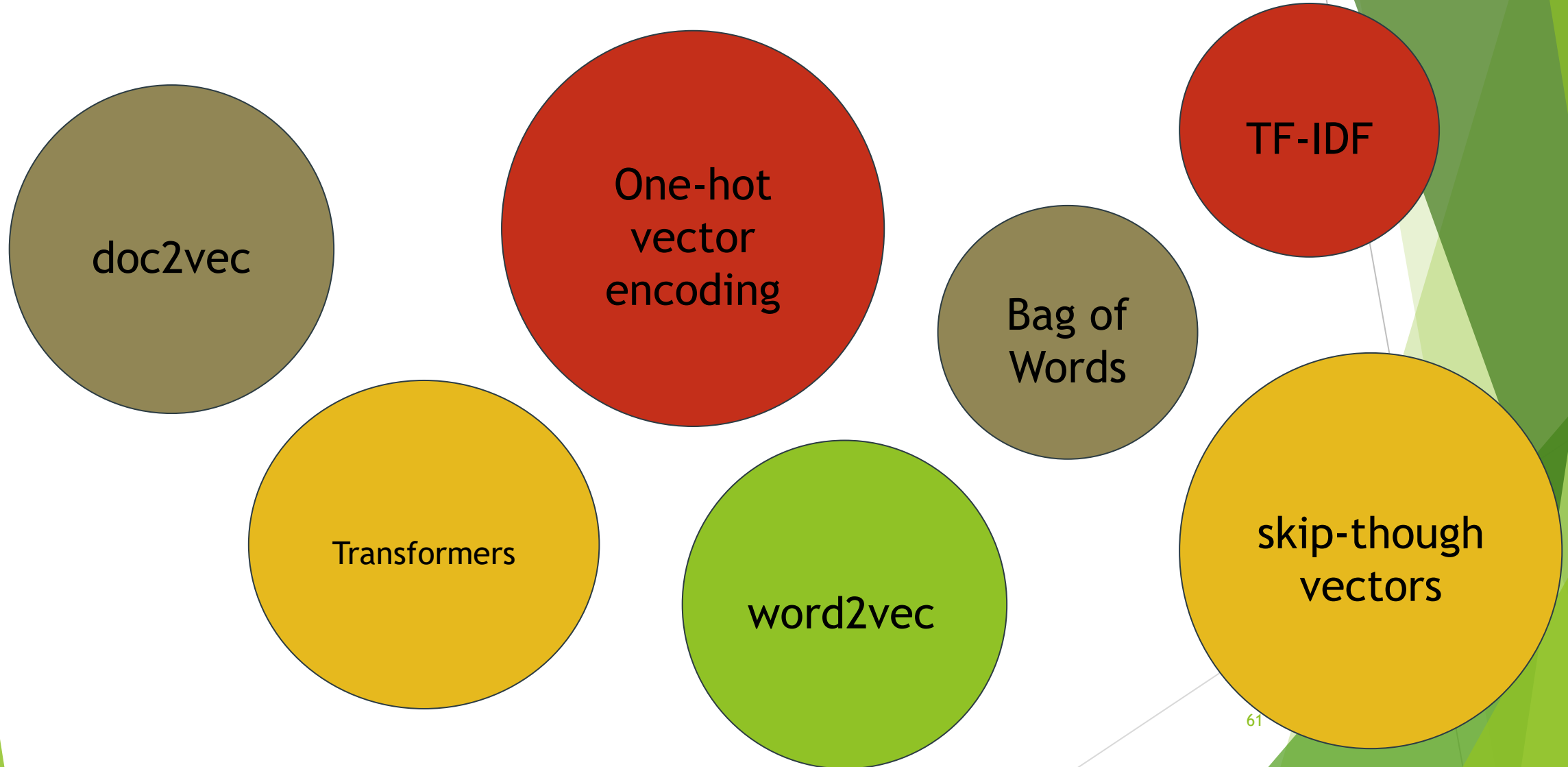
Lemmatization

- Transforms a word into a canonical form (lemma) by standardizing gender (masculine or feminine), number (one or more), person (me, you, them...), mode (indicative, imperative...).
- seeks → seek
- dogs → dog

Stemming

- Transforms a word into a radical or root. The affixes (suffixes, prefixes, postfixes, antefixes) are removed, leaving only the root.
- Sometimes same result as the lemmatization:
- fishing, fished, fish, fisher → fish
- Source of error sometimes: university, universe → univers

How to represent a document?



Opensearch for geospatial data

Geospatial data representation

- ▶ To represent **geospatial** data in Opensearch, you have two data types:
 - ▶ “geo_point”
 - ▶ “geo_shape”
- ▶ In OpenSearch, the geo_distance function is based on the **great-circle distance**, which is calculated using the Earth's reference sphere. This distance is determined using **Haversine's formula** or other spherical approximations.

Geo Point

► geo_point

► fields accept latitude-longitude pairs, which can be used:

- to find geo-points within a **bounding box**, within a certain **distance** of a central point, or within a **polygon**.
- to aggregate documents **geographically** or by **distance** from a central point.
- to integrate distance into a document's **relevance score**.
- to **sort** documents by distance.

Geo shape

► geo_shape

► With this data type you can have more representation **geospacial** data :

- **Point**: defined by an one point with latitude and longitude.
- **Linestring**: defined by an array of two or more positions.
- **Polygon**: defined by a list of a list of points.
- **Multipoint**: a list of geojson points.
- **Multilinestring**: is an example of a list of geojson linestrings.
- **Multipolygon**: list of geojson polygons.
- **Geometrycollection**: collection of geojson geometry objects.
- **Envelope**: consists of coordinates for upper left and lower right points of the shape to represent a bounding rectangle in the format `[[minLon, maxLat], [maxLon, minLat]]` .
- **Circle**: consists of a center point with a radius.

Geospatial queries

- ▶ You can search by:
 - ▶ **geo_bounding_box**: query to finds documents with geo-points that fall into the specified rectangle.
 - ▶ **geo_distance**: query to finds documents with geo-points within the specified distance of a central point.
 - ▶ **geo_polygon**: query to find documents with geo-points within the specified polygon.
 - ▶ **geo_shape query**: to finds documents with geo-shapes which either intersect, are contained by, or do not intersect with the specified geo-shape.

Geospatial search

Filtre par Geo Point

Geo distance, geo distance range et geo bounding box

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance": {
          "distance": "1km",
          "distance_type": "plane",
          "location": {
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance_range": {
          "gte": "1km",
          "lt": "2km",
          "location": {
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```

```
GET /attractions/restaurant/_search
```

```
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.7,
              "lon": -73.0
            }
          }
        }
      }
    }
  }
}
```