OpenSearch: everything you need to know for the perfect setup

4 August 2023 - 10 min. read

Data and Analytics | Data Visualization | OpenSearch

Data is all around.
Thus, every organization - regardless of size or industry - faces the challenge of managing and extracting valuable information from huge amounts of information. In this scenario, fast and effective data handling and processing arises as a priority for companies to adapt, react, and keep pace with today's fast-changing environments. For this reason, selecting the perfect tool that best meets organization's needs is crucial.

In this blog post, we present OpenSearch, a highly scalable toolset providing fast access and response to large volumes of data.

Let's dive deep!

Cluster fundamentals

OpenSearch is an open-source search and analytics suite used to query large volumes of data using API calls or an integrated Dashboard. OpenSearch offers features such as Full-text querying, Autocomplete, Scroll Search, customizable scoring and ranking, fuzzy matching, phrase matching, and more. Responses can be returned in jdbc, csv, raw, or JSON format.

Let's have a brief description of the fundamental componets of an OpenSearch cluster:

- Indexes
- Shards
- Node

- Types of node

To search data, you must organize it into **indexes**. Indexes store documents (sets of fields with key-value pairs) and optimize them. Optimization is possible because each field has a specific type. You can specify field types. Otherwise, OpenSearch can try to determine the type automatically.

Another form of optimization is splitting the index into several **shards**. Each shard contains a subset of the documents inside the index. When you search for data, queries run across different shards in parallel if each shard is located on a different node. The size of shards should be around 10-30 GB for workloads requiring low search latency and 30-50 GB for write-heavy workloads such as storing logs.

OpenSearch instances are called **nodes**. OpenSearch can operate as a single-node or multi-node cluster. When creating a multi-node cluster, the number of nodes, the node types, and their hardware depend on your use case.

The **types of nodes** are:

- **Master**: the master node manages tasks such as indexes management, keeping track of the cluster nodes, doing health checks, and allocating shards
- **Master-eligible**: master-eligible nodes can be promoted to master through a voting process
- **Data**: data nodes perform all-data related operations on local shards, such as indexing, searching, and aggregating
- **Ingest**: ingest nodes run pipelines to transform data before storing it
- **Coordinating**: coordinating nodes delegate client requests to Data Nodes and aggregate the results into one before sending it to the client.

A node can have multiple types. Each node is a master-eligible, data, ingest, and coordinating node by default.

During the creation of an AWS OpenSearch cluster, you can customize Data Nodes and Dedicate Master Nodes.

For Data Nodes, you can specify the number of nodes, the instance type, the volume type, and the volume size.

Dedicated Master Nodes are nodes that do not contain data and are dedicated to cluster management. You can choose to not have any Dedicate Master Node in development or test environments. Since they do not contain data, you must specify only the number of nodes and the instance type.

Provisioning types

On AWS, you can deploy an OpenSearch cluster in two different ways:

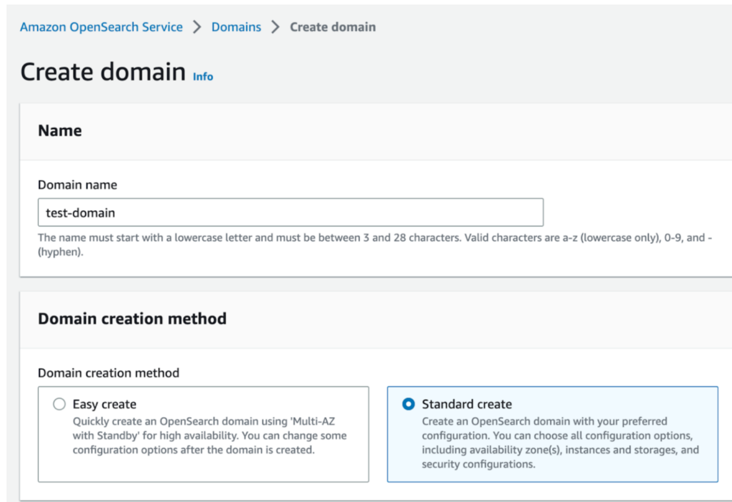
- Servefull
- Serverless

Serverfull

The classic approach is the Serverfull one in which you have to choose how many nodes you want to create, specifying the node types, sizing, and other properties.

Let's look at the AWS Console and go through the Amazon OpenSearch service. The first step we will do is create a domain, in other words, an OpenSearch cluster.

You will have to choose between two options: **Easy create** or **Standard create.**



We strongly recommend choosing the Standard Create option since it is more flexible, while you could use the **Easy create** one just for short demos or POCs.

Going deep into the Standard creation, you can select from two different templates: Production or dev/test. We suggest skipping this choice and going through the other parameters.

The most important selection is the kind of redundancy of the cluster:

Pay attention to this option since it will drastically change your pricing.

In fact, the option **Domain with standby** will create a cluster with minimum of 3 data nodes, and you can increment them only by multiples of 3.

Meanwhile, with the **Domain without standby,** you can customize the number of nodes, but none of them will be reserved as standby nodes.

Which is the best-fitting solution for your use case? As a rule of thumb, choose the Domain with a standby option for mission-critical workloads. For other scenarios, just choose the other option.

Serverless

AWS provides OpenSearch Serverless, an on-demand, auto-scaling configuration for Amazon OpenSearch Service. OpenSearch Serverless provides collections: groups of indexes with a common use case. Collections are the serverless counterpart of clusters, but collections don't require manual provisioning. OpenSearch Serverless allows two types of collections: **Time series collections** and **Search collections**.

The main difference is that in search collections, all data is stored in hot storage to ensure fast query response times; instead, time series collections use a combination of hot and warm caches to optimize query response times for more frequently accessed data.

Also, you can index by custom ID only in search collections.

OpenSearch Serveless compute capacity is measured in OpenSearch Compute Units (OCUs). Each OCU is a combination of 6 GiB of memory, corresponding virtual CPU, and data transfer to Amazon S3.

The first Serverless collection instantiates a total of four OCUs (2 used for ingesting, primary, and standby; 2 used for searching, and one active replica for high availability). Subsequent collections can share these OCUs. OpenSearch scales OCUs based on the indexing and searching workloads. These four initial OCUs are always active, even when there's no indexing or search activity, so you are billed for at least four OCUs. You can set a maximum number of OCUs to contain costs. You are also billed for storage retained in Amazon S3.

Query types

Ok, now we know everything about the creation of an OpenSearch cluster, but how can we use it? Through **Queries**.

OpenSearch provides a search language called query domain-specific language (DSL) which provides a JSON interface. OpenSearch also supports SQL to write queries rather than use DSL.

There are two main types of queries: **leaf queries** and **compound queries**.

Leaf Queries

Leaf queries search for a specified value in a certain field or fields. Leaf queries can be further categorized into different sub-types.

Full-text queries

Full-text queries are used to search text documents. There are different types of full-text queries.

You can match a specific value in a specific field or search for the values in a list of fields. You can assign a weight to each field to boost its value in the result score. For instance, while searching for a book, finding the value in the “title” field could have a bigger impact than finding it in the “abstract” field.

A type of query allows you to search for different terms in a field, and the last term of the input string will be used as a prefix to have documents that contain either any of these terms or terms starting with the prefix.

Some full-text queries support a “fuzziness” parameter useful in case of inputs written with missing characters or characters whose position is exchanged.

Others support a “slop” parameter that controls how words can be misordered and still be considered a match.

```
GET _search
{
  "query": {
    "multi_match": {
      "query": "proud",
      "fields": ["title^2", "body"]
    }
  }
}
```

Term-level queries

Term-level queries are used to search documents for an exact specified term.

You can search for exact or multiple terms in a field. When searching for multiple terms, term-level queries also allow you to specify the minimum number of matches required. You can search for an ID, a value contained in a range, or terms containing a specific prefix. Fuzzy queries search for terms similar to the search term using the Levenshtein distance. Term-level queries offer the possibility to search using wildcard or Lucene regular expression. You can also search for documents containing a specific field instead of a value.

```
GET blog/_search
{
  "query": {
    "wildcard": {
      "title": {
        "value": "*cloud"
      }
    }
  }
}
```

XY queries

XY queries search for documents that contain geometries using xy_point and xy_shape fields. xy_point fields support points. xy_shape fields support points, lines, circles, and polygons.

You can search for documents whose points or shapes intersect the provided shape or those whose shapes do not intersect, are contained, or contain the provided shape.

```
GET index/_search
{
  "query": {
    "xy_shape": {
      "geometry": {
        "shape": {
          "type": "circle",
          "coordinates": [1.0, 5.0],
          "radius": 3
        },
        "relation": "INTERSECTS"
      }
    }
  }
}
```

Geographic queries

Geographic queries are used to search documents containing geospatial geometries. Geographic queries support points and shapes like lines, circles, and polygons.

You can search for documents whose geopoint values are within a bounding box or a polygon. Geoshape queries return documents that contain geopoints or geoshapes that intersect the provided shape, or documents that contain geoshapes that do not intersect, are contained, or contain the provided shape. You can also query for documents with geopoints within a specified distance from a provided geopoint.

```
GET index/_search
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
        "geo_distance": {
          "distance": "10km",
          "pin.location": {
            "lat": 10,
            "lon": 50
          }
        }
      }
    }
  }
}
```

Joining queries

Joining queries can be used to query a nested object as an independent document or retrieve parent or child documents linked through a field of “join” type.

```
GET /_search
{
  "query": {
    "has_child": {
      "type": "child",
      "query": {
        "match_all": {}
      }
    }
  }
}
```

Compound queries

Compound queries wrap multiple leaf queries to combine their results or modify their behavior. You can join multiple query clauses with boolean logic or assign a higher score to documents that match multiple clauses. Compound queries also offer the ability to create a function to recalculate returned documents' scores. Lastly, you can combine a "positive" query with a "negative" query, documents found through the positive query will have an increased score, while documents found through the negative query will have a decreased score. This is useful in case of synonyms that you want to remove from your results.

Use cases

There are two main purposes for using the OpenSearch service: one is to store and query/analyze application and infrastructure **logs & metrics**, and the other is to build a **search engine**.

Logs & Metrics

This is one of the main uses of OpenSearch. Thanks to the fast query response time in a lot of data, it allows very fast searching for specific application or infrastructure logs.

The main purpose of this solution is to have a centralized log storage where you can find all the applications and infrastructure logs and build near real-time dashboards that visualize different metrics to understand your application's healthiness.

Let's imagine a web application composed of a Single Page Application front-end and a back-end hosted on AWS using serverless services like Amazon API Gateway and AWS Lambda. What you can do is send all the logs to AWS OpenSearch, structuring your log payload with at least this information:

- Request ID: a unique identifier for the request made by your front-end.
- Type of Log: the log type like INFO, ERROR, DEBUG, WARNING.
- Message: your log message
- Timestamp: The timestamp of the log

And then add different attributes based on what you are logging for. For example, we can imagine having at least two logs for every request: one when the request starts that will add the event source as the additional attributes and one when the request ends, with an attribute that defines the response status code. If an error happens, we can imagine logging the same payload with an error status code.

Just with this information, we can build a simple Dashboard that shows us the following:

- The number of requests per second
- The number of successful requests
- The number of errors with the associated error message
- The approximate response time average of the application

And we can filter all the logs for a specific time range or even for the request id showing us the entire log of the single request.

Search Engine

Another main purpose of using OpenSearch is to build a Search Engine. Thanks to all the different types of queries that the engine supports, you can simply create a search bar that uses fuzziness to search all the data that look like what your user has typed.

You can even use the match_phrase_prefix function to autocomplete your user input based on the data that you have in the database or use the suggest function to correct your user input.

Conclusion

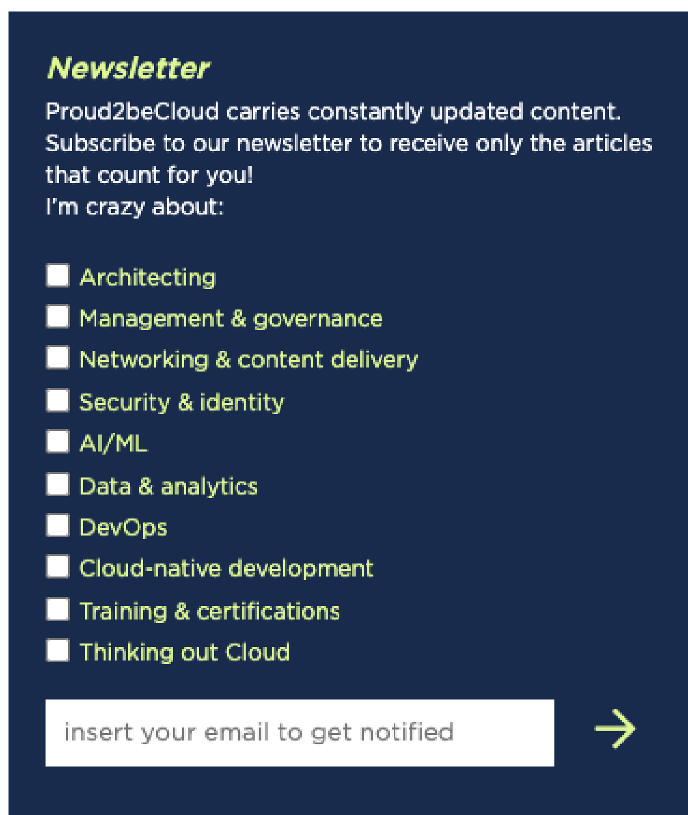
In this article, we have given a general view of what Opensearch is, its fundamentals, and how to leverage AWS services to manage a cluster of this kind. Nowadays, so many

workloads require such data sources, so you must know it!

Many topics have not been taken into consideration, such as how to authenticate and authorize to an OpenSearch cluster, how to leverage Amazon S3 as warm and cold data storage for the cluster, enable Cognito as Identity Provider, or how to manage Disaster Recovery for business-critical applications.

If you are interested in these or other aspects of OpenSearch implementations, please leave a comment below!

More content like this? Choose your topic and subscribe to our newsletter!



About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Alessandro Bertini

DevOps Engineer @ beSharp. I deal with Cloud-Native software development, strongly oriented to the serverless paradigm!Passionate about board games and video games (as the best geeks do!)

Daniele Papa

DevOps Engineer and backend developer @ beSharp. I like playing video games and board games in my free time. In the last few years, I approached the Cloud environment, and now I switch between IAM roles and role-play games.

Copyright © 2011-2023 by beSharp spa - P.IVA IT02415160189