Cloudera Runtime 7.1.9

Deployment Planning for Cloudera Search

Date published: 2015-05-05
Date modified: 2024-02-07

https://docs.cloudera.com/

Legal Notice

Contents

Things to consider before deploying Cloudera Search

Deployment planning for Cloudera Search is a complex procedure, covering several aspects from resource considerations to defining a schema.

Dimensioning guidelines for deploying Cloudera Search

Think about the type of workload your future Search deployment is expected to handle and plan resources accordingly.You need to consider several aspects from the amount of data ingested and the frequency of updates, through the number and complexity of concurrent queries to security and resilience.

Cloudera Search provides interactive search and scalable indexing. Before you begin installing Cloudera Search:

- Select the machines on which you want to install Cloudera Search, as well as any other services you are collocating with Search.
- Identify the tasks and workloads you need to manage and the types of data you need to search. This information can help guide the deployment process.

Important: Cloudera Search does not support contrib modules, such as DataImportHandler.

Spark, MapReduce and Lily HBase indexers are not contrib modules but part of the Cloudera Search product itself, therefore they are supported.

Memory considerations

Default CDP deployments use a Java virtual machine (JVM) size of 1 GB, but this is insufficient for most use cases. Consider the following when determining an optimal JVM size for production usage:

- The more searchable data you have, the more memory you need. In general, 10 TB of searchable data requires more memory than 1 TB of searchable data.
- Indexing more fields requires more memory. For example, indexing all fields in a collection of logs, email messages, or Wikipedia entries requires more memory than indexing only the Date Created field.
- If the system must be stable and respond quickly, more memory might help. If slow responses are acceptable, you might be able to use less memory.

To ensure an appropriate amount of memory, consider your requirements and experiment in your environment. In general:

- Machines with 16 GB RAM are sufficient for some smaller loads or for evaluation.
- Machines with 32 GB RAM are sufficient for some production environments.
- Machines with 96 GB RAM are sufficient for most situations.

Define the use case

To determine how best to deploy Search in your environment, define use cases. You can use a sample application to benchmark different use cases and data types and sizes to help you identify the most important performance factors.

Consider the information given here as guidance instead of absolute requirements.

The same Solr index can have different hardware requirements, depending on the types of queries performed. The most common variation in hardware requirements is memory. For example, the memory requirements for faceting vary depending on the number of unique terms in the faceted field. Suppose you want to use faceting on a field that has 10 unique values. In this case, only 10 logical containers are required for counting. No matter how many documents are in the index, memory overhead is almost nonexistent.

Conversely, the same index could have unique timestamps for every entry, and you want to facet on that field with a : -type query. In this case, each index requires its own logical container. With this organization, if you had a

large number of documents—500 million, for example—then faceting across 10 fields would increase the RAM requirements significantly.

For this reason, you must consider use cases and data characteristics before you can estimate hardware requirements. Important parameters to consider are:

- Number of documents. For Cloudera Search, sharding is almost always required.
- Approximate word count for each potential field.
- What information is stored in the Solr index and what information is only for searching? Information stored in the index is returned with the search results.
- Foreign language support:
  - How many different languages appear in your data?
  - What percentage of documents are in each language?
  - Is language-specific search supported? This determines whether accent folding and storing the text in a single field is sufficient.
  - What language families will be searched? For example, you could combine all Western European languages into a single field, but combining English and Chinese into a single field is not practical. Even with similar languages, using a single field for different languages can be problematic. For example, sometimes accents alter the meaning of a word, and in such cases, accent folding loses important distinctions.
- Faceting requirements:
  - Be wary of faceting on fields that have many unique terms. For example, faceting on timestamps or free-text fields typically has a high cost. Faceting on a field with more than 10,000 unique values is typically not useful. Ensure that any such faceting requirement is necessary.
  - What types of facets are needed? You can facet on queries as well as field values. Faceting on queries is often useful for dates. For example, “in the last day” or “in the last week” can be valuable. Using Solr Date Math to facet on a bare “NOW” is almost always inefficient. Facet-by-query is not memory-intensive because the number of logical containers is limited by the number of queries specified, no matter how many unique values are in the underlying field. This can enable faceting on fields that contain information such as dates or times, while avoiding the problem described for faceting on fields with unique terms.
- Sorting requirements:
  - Sorting requires one integer for each document (maxDoc), which can take up significant memory. Additionally, sorting on strings requires storing each unique string value.
- Paging requirements. End users rarely look beyond the first few pages of search results. For use cases requiring deep paging (paging through a large number of results), using cursors can improve performance and resource utilization. For more information, see Pagination of Results on the Apache Solr wiki.
- Is advanced search capability planned? If so, how will it be implemented? Significant design decisions depend on user requirements:
  - Can users be expected to learn about the system? Advanced screens can intimidate e-commerce users, but these screens can be more effective if users can be expected to learn them.
  - How long will users wait for results? Data mining or other design requirements can affect response times.
- How many simultaneous users must your system accommodate?
- Update requirements. An update in Solr refers both to adding new documents and changing existing documents:
  - Loading new documents:
    - Bulk. Will the index be rebuilt from scratch periodically, or will there only be an initial load?
    - Incremental. At what rate will new documents enter the system?
  - Updating documents:
    - Can you characterize the expected number of modifications to existing documents?
    - How much latency is acceptable between when a document is added to Solr and when it is available in Search results?
- Security requirements. Solr has no built-in security options, although Cloudera Search supports authentication using Kerberos. In Solr, document-level security is usually best accomplished by indexing authorization tokens

with the document. The number of authorization tokens applied to a document is largely irrelevant; for example, thousands are reasonable but can be difficult to administer. The number of authorization tokens associated with a particular user should be no more than 100 in most cases. Security at this level is often enforced by appending an fq clause to the query, and adding thousands of tokens in an fq clause is expensive.

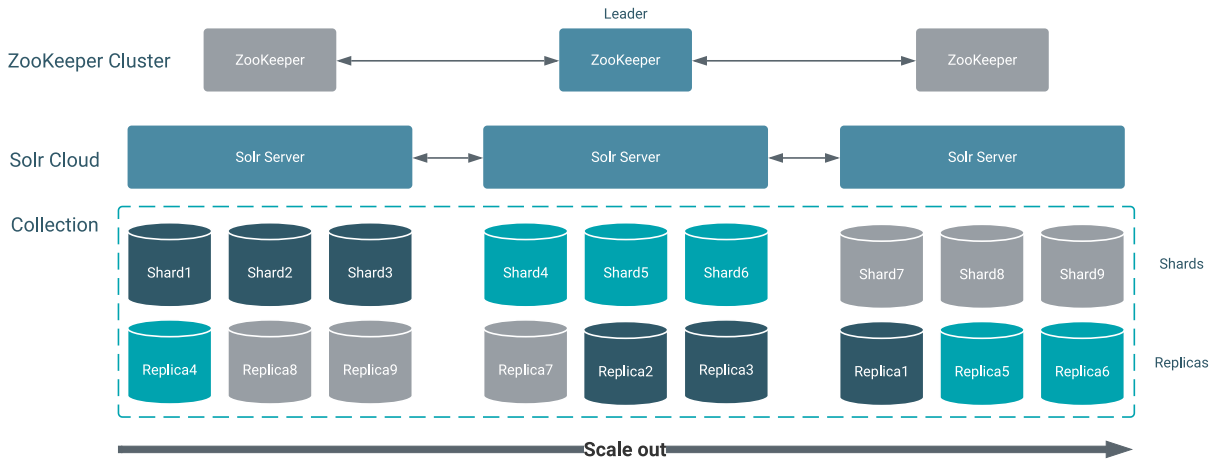
  - A post filter, also known as a no-cache filter, can help with access schemes that cannot use an fq clause. These are not cached and are applied only after all less-expensive filters are applied.
  - If grouping, faceting is not required to accurately reflect true document counts, so you can use some shortcuts. For example, ACL filtering is expensive in some systems, sometimes requiring database access. If completely accurate faceting is required, you must completely process the list to reflect accurate facets.
- Required query rate, usually measured in queries-per-second (QPS):
  - At a minimum, deploy machines with sufficient hardware resources to provide an acceptable response rate for a single user. Some types of queries can utilize the system so much that performance for even a small number of users is unacceptable. In this case, resharding can help.
  - If QPS is only somewhat lower than required and you do not want to reshard, you can often improve performance by adding replicas to each shard.
  - As the number of shards in your deployment increases, the general QPS rate starts to slowly decrease. This typically occurs when there are hundreds of shards.

Lucene index size limitation

A Solr collection can consist of multiple Lucene indexes because of sharding and replication. Splitting a collection into multiple shards (based on the order of magnitude of the number of documents to be indexed) is a very common setup to scale out Solr.

This is illustrated in figure *Solr scaling*. Each shard and replica represents a Lucene index. One such component can contain a maximum of ~2 billion documents.

Figure 1: Solr scaling



This allows Solr to handle more than 2 billion documents. Although for collections with billions of documents, there is a good chance that you will run into some other bottlenecks that require further tuning; especially if the size of each document is large or the schema is complex.

Related Information

Package org.apache.lucene.codecs.lucene40 Limitations

Schemaless mode overview and best practices

Schemaless mode removes the need to design a schema before beginning to use Search. This can help you begin using Search more quickly, but schemaless mode is typically less efficient and effective than using a deliberately designed schema.

Note: Cloudera recommends pre-defining a schema before moving to production.

With the default non-schemaless mode, you create a schema by writing a schema.xml file before loading data into Solr so it can be used by Cloudera Search. You typically write a different schema definition for each type of data being ingested, because the different data types usually have different field names and values. Writing a custom schema is done by examining the data to be imported so its structure can be understood, and then creating a schema that accommodates that data. For example, emails might have a field for recipients and log files might have a field for IP addresses for machines reporting errors. Conversely, emails typically do not have an IP address field and log files typically do not have recipients. Therefore, the schema you use to import emails is different from the schema you use to import log files.

Cloudera Search offers schemaless mode to help facilitate sample deployments without the need to pre-define a schema. While schemaless is not suitable for production environments, it can help demonstrate the functionality and features of Cloudera Search. Schemaless mode operates based on three principles:

1. The schema is automatically updated using an API. When not using schemaless mode, users manually modify the schema.xml file or use the Schema API.
2. As data is ingested, it is analyzed and a guess is made about the type of data in the field. Supported types include Boolean, Integer, Long, Float, Double, Date, and Text.
3. When a new field is encountered, the schema is automatically updated using the API. The update is based on the guess about the type of data in the field.

For example, if schemaless encounters a field that contains "6.022", this would be determined to be type Float, whereas "Mon May 04 09:51:52 CDT 2009" would be determined to be type Date.

By combining these techniques, Schemaless:

1. Starts without a populated schema.
2. Intakes and analyzes data.
3. Modifies the schema based on guesses about the data.
4. Ingests the data so it can be searched based on the schema updates.

To generate a configuration for use in Schemaless mode, use solrctl instancedir --generate path -schemaless. Then, create the instancedir and collection as with non-schemaless mode.

Best practices

Give each collection its own unique Instancedir

Solr supports using the same instancedir for multiple collections. In schemaless mode, automatic schema field additions actually change the underlying instancedir. Thus, if two collections are using the same instancedir, schema field additions meant for one collection will actually affect the other one as well. Therefore, Cloudera recommends that each collection have its own instancedir.

Advantages of defining a schema for production use

Although schemaless mode facilitates quick adoption and exploratory use, you are recommended to implement a schema for production use of Cloudera Search. This ensures that ingestion of data is consistent with expectations and query results are reliable.

Schemaless Solr is useful for getting started quickly and for understanding the underlying structure of the data you want to search. However, Schemaless Solr is not recommended for production use cases. Because the schema is automatically generated, a mistake like misspelling the name of the field alters the schema, rather than producing an error. The mistake may not be caught until much later and once caught, may require re-indexing to fix. Also, an unexpected input format may cause the type guessing to pick a field type that is incompatible with data that is subsequently ingested, preventing further ingestion until the incompatibility is manually addressed. Such a case is rare, but could occur. For example, if the first instance of a field was an integer, such as '9', but subsequent entries were text such as '10 Spring Street', the schema would make it impossible to properly ingest those subsequent entries. Therefore, Schemaless Solr may be useful for deciding on a schema during the exploratory stage of development, but Cloudera recommends defining the schema in the traditional way before moving to production.

When constructing a schema, use data types that most accurately describe the data that the fields will contain. For example:

- Use the tdate type for dates. Do this instead of representing dates as strings.
- Consider using the text type that applies to your language, instead of using String. For example, you might use text_en. Text types support returning results for subsets of an entry. For example, querying on "john" would find "John Smith", whereas with the string type, only exact matches are returned.
- For IDs, use the string type.