Cloudera Runtime 7.2.15

Cloudera Search Overview

Date published: 2015-05-05
Date modified: 2022-05-12

https://docs.cloudera.com/

Legal Notice

Contents

What is Cloudera Search....................4

How Cloudera Search works....................5

Cloudera Search and CDP.................... 6

Cloudera Search and other Cloudera Runtime components.................... 6

Cloudera Search architecture....................8

Local file system support....................11

Cloudera Search tasks and processes....................13

Ingestion....................13
Indexing....................14
Querying....................15

Extracting, transforming, and loading data with Cloudera Morphlines....................15

Backing up and restoring data using Search....................17

What is Cloudera Search

Learn about how Cloudera Search is different from Apache Solr and the added value it provides.

Cloudera Search is Apache Solr fully integrated in the Cloudera platform, taking advantage of the flexible, scalable, and robust storage system and data processing frameworks included in Cloudera Data Platform (CDP). This eliminates the need to move large data sets across infrastructures to perform business tasks. It further enables a streamlined data pipeline, where search and text matching is part of a larger workflow.

Cloudera Search provides easy, natural language access to data stored in or ingested into Hadoop, HBase, or cloud storage. End users and other web services can use full-text queries and faceted drill-down to explore text, semi-structured, and structured data as well as quickly filter and aggregate it to gain business insight without requiring SQL or programming skills.

Using Cloudera Search with the CDP infrastructure provides:

- Simplified infrastructure
- Better production visibility and control
- Quicker insights across various data types
- Quicker problem resolution
- Simplified interaction and platform access for more users and use cases beyond SQL
- Scalability, flexibility, and reliability of search services on the same platform used to run other types of workloads on the same data
- A unified security model across all processes with access to your data
- Flexibility and scale in ingest and pre-processing options

The following table describes Cloudera Search features.

Table 1: Cloudera Search Features

Feature	Description
Unified management and monitoring with Cloudera Manager	Cloudera Manager provides unified and centralized management and monitoring for Cloudera Runtime and Cloudera Search. Cloudera Manager simplifies deployment, configuration, and monitoring of your search services. Many existing search solutions lack management and monitoring capabilities and fail to provide deep insight into utilization, system health, trending, and other supportability aspects.
Simple cluster and collection management using the solrctl tool	Solrctl is a command line tool that allows convenient, centralized management of: • Solr instance configurations and schemas in ZooKeeper • Solr collection life cycle including low level control (Solr cores) • Collection snapshots, Backup and restore operations • SolrCloud cluster initialization and configuration
Index storage in HDFS	Cloudera Search is integrated with HDFS for robust, scalable, and self-healing index storage. Indexes created by Solr/Lucene are directly written in HDFS with the data, instead of to local disk, thereby providing fault tolerance and redundancy. Cloudera Search is optimized for fast read and write of indexes in HDFS while indexes are served and queried through standard Solr mechanisms. Because data and indexes are co-located, data processing does not require transport or separately managed storage.
Batch index creation through MapReduce	To facilitate index creation for large data sets, Cloudera Search has built-in MapReduce jobs for indexing data stored in HDFS or HBase. As a result, the linear scalability of MapReduce is applied to the indexing pipeline, off-loading Solr index serving resources.
Easy interaction and data exploration through Hue	A Cloudera Search GUI is provided as a Hue plug-in, enabling users to interactively query data, view result files, and do faceted exploration. Hue can also schedule standing queries and explore index files. This GUI uses the Cloudera Search API, which is based on the standard Solr API. The drag-and-drop dashboard interface makes it easy for anyone to create a Search dashboard.

Feature	Description
Simplified data processing for Search workloads	Cloudera Search can use Apache Tika for parsing and preparation of many of the standard file formats for indexing. Additionally, Cloudera Search supports Avro, Hadoop Sequence, and Snappy file format mappings, as well as Log file formats, JSON, XML, and HTML. Cloudera Search also provides Morphlines, an easy-to-use, pre-built library of common data preprocessing functions. Morphlines simplifies data preparation for indexing over a variety of file formats. Users can easily implement Morphlines for Kafka, and HBase, or re-use the same Morphlines for other applications, such as MapReduce or Spark jobs.
HBase search	Cloudera Search integrates with HBase, enabling full-text search of HBase data without affecting HBase performance or duplicating data storage. A listener monitors the replication event stream from HBase RegionServers and captures each write or update-replicated event, enabling extraction and mapping (for example, using Morphlines). The event is then sent directly to Solr for indexing and storage in HDFS, using the same process as for other indexing workloads of Cloudera Search. The indexes can be served immediately, enabling free-text searching of HBase data.
Spark-Solr connector	You can use the Spark-Solr connector to index data into Solr in multiple ways, both batch and streaming. With the connector you get the benefits of Spark while you can access Solr easily and in a familiar way, from both Scala and Java.

How Cloudera Search works

Learn about the operation of Cloudera Search

In near real-time indexing use cases, such as log or event stream analytics, Cloudera Search indexes events that are streamed through Apache Kafka, Spark Streaming, or HBase. Fields and events are mapped to standard Solr indexable schemas. Lucene indexes the incoming events and the index is written and stored in standard Lucene index files in HDFS.

The indexes by default are loaded from HDFS to Solr cores, but Solr is also able to read from local disk. The difference in the design of Cloudera Search is the robust, distributed, and scalable storage layer of HDFS, which helps eliminate costly downtime and allows for flexibility across workloads without having to move data. Search queries can then be submitted to Solr through either the standard Solr API, or through a simple search GUI application, included in Cloudera Search, which can be deployed in Hue.

Cloudera Search batch-oriented indexing capabilities can address needs for searching across batch uploaded files or large data sets that are less frequently updated and less in need of near-real-time indexing. It can also be conveniently used for re-indexing (a common pain point in stand-alone Solr) or ad-hoc indexing for on-demand data exploration. Often, batch indexing is done on a regular basis (hourly, daily, weekly, and so on) as part of a larger workflow.

For such cases, Cloudera Search includes a highly scalable indexing workflow based on MapReduce or Spark. A MapReduce or Spark workflow is launched for specified files or folders in HDFS, or tables in HBase, and the field extraction and Solr schema mapping occurs during the mapping phase. Reducers use embedded Lucene to write the data as a single index or as index shards, depending on your configuration and preferences. After the indexes are stored, they can be queried by using standard Solr mechanisms, as previously described above for the near-real-time indexing use case. You can also configure these batch indexing options to post newly indexed data directly into live, active indexes, served by Solr. This GoLive option enables a streamlined data pipeline without interrupting service to process regularly incoming batch updates.

The Lily HBase Indexer Service is a flexible, scalable, fault tolerant, transactional, near real-time oriented system for processing a continuous stream of HBase cell updates into live search indexes. The Lily HBase Indexer uses Solr to index data stored in HBase. As HBase applies inserts, updates, and deletes to HBase table cells, the indexer keeps Solr consistent with the HBase table contents, using standard HBase replication features. The indexer supports flexible custom application-specific rules to extract, transform, and load HBase data into Solr. Solr search results can contain columnFamily:qualifier links back to the data stored in HBase. This way applications can use the Search result set to directly access matching raw HBase cells. Indexing and searching do not affect operational stability or write throughput of HBase because the indexing and searching processes are separate and asynchronous to HBase.

Cloudera Search and CDP

Discover how Search fits into Cloudera's Data Platform offering.

Cloudera Search fits into the broader set of solutions available for analyzing information in large data sets. Cloudera Data Platform (CDP) provides both the means and the tools to store the data and run queries. You can explore data through:

- MapReduce or Spark jobs
- Impala queries
- Cloudera Search queries

CDP provides storage for and access to large data sets by using MapReduce jobs, but creating these jobs requires technical knowledge, and each job can take minutes or more to run. The longer run times associated with MapReduce jobs can interrupt the process of exploring data.

To provide more immediate queries and responses and to eliminate the need to write MapReduce applications, you can use Apache Impala. Impala returns results in seconds instead of minutes.

Although Impala is a fast, powerful application, it uses SQL-based querying syntax. Using Impala can be challenging for users who are not familiar with SQL. If you do not know SQL, you can use Cloudera Search. Although Impala, Apache Hive, and Apache Pig all require a structure that is applied at query time, Search supports free-text search on any data or fields you have indexed.

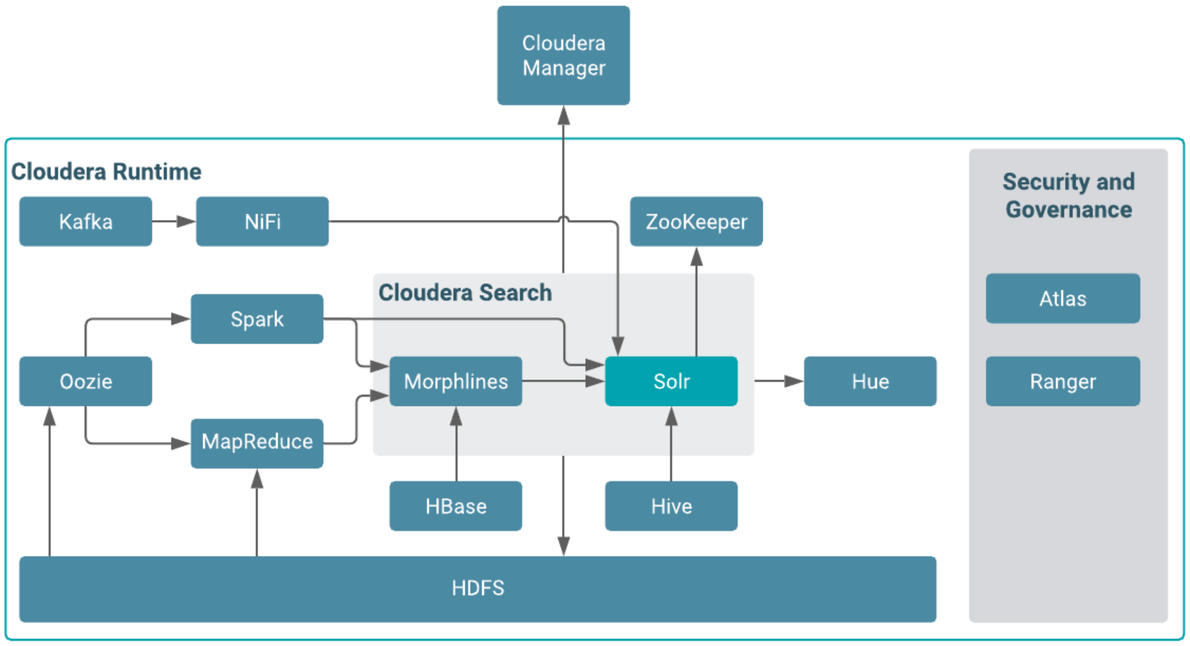
How Search uses existing infrastructure

Any data already in a CDP deployment can be indexed and made available for query by Cloudera Search. For data that is not stored in CDP, Cloudera Search provides tools for loading data into the existing infrastructure, and for indexing data as it is moved to HDFS or written to Apache HBase.

By leveraging existing infrastructure, Cloudera Search eliminates the need to create new, redundant structures. In addition, Cloudera Search uses services provided by CDP and Cloudera Manager in a way that does not interfere with other tasks running in the same environment. This way, you can reuse existing infrastructure without the cost and problems associated with running multiple services in the same set of systems.

Cloudera Search and other Cloudera Runtime components

Cloudera Search interacts with other Cloudera Runtime components to solve different problems. Learn about Cloudera components that contribute to the Search process and see how they interact with Cloudera Search.



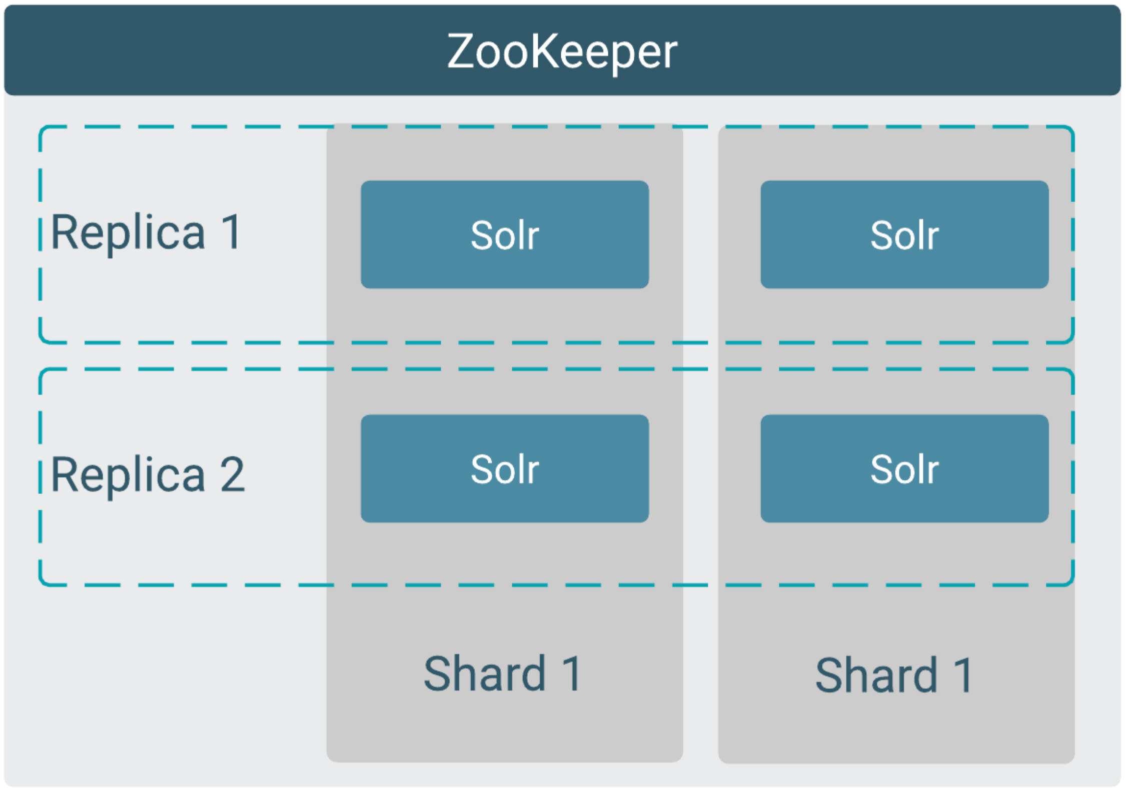
Component	Contribution	Applicable To
HDFS	Stores source documents. Search indexes source documents to make them searchable. Files that support Cloudera Search, such as Lucene index files and write-ahead logs, are also stored in HDFS. Using HDFS provides simpler provisioning on a larger base, redundancy, and fault tolerance. With HDFS, Cloudera Search servers are essentially stateless, so host failures have minimal consequences. HDFS also provides snapshotting, inter-cluster replication, and disaster recovery.	All cases
MapReduce	Search includes a pre-built MapReduce-based job. This job can be used for on-demand or scheduled indexing of any supported data set stored in HDFS. This job uses cluster resources for scalable batch indexing.	Many cases
Hue	Hue includes a GUI-based Search application that uses standard Solr APIs and can interact with data indexed in HDFS. The application provides support for the Solr standard query language and visualization of faceted search functionality.	Many cases
Morphlines	A morphline is a rich configuration file that defines an ETL transformation chain. Morphlines can consume any kind of data from any data source, process the data, and load the results into Cloudera Search. Morphlines run in a small, embeddable Java runtime system, and can be used for near real-time applications such as the Lily HBase Indexer as well as batch processing applications such as a Spark job.	Many cases
ZooKeeper	Coordinates distribution of data and metadata, also known as shards. It provides automatic failover to increase service resiliency.	Many cases
Spark	The CrunchIndexerTool can use Spark to move data from HDFS files into Apache Solr, and run the data through a morphline for extraction and transformation. The Spark-Solr connector framework enables ETL of large datasets to Solr using spark-submit with a Spark job to batch index HDFS files into Solr.	Some cases
NiFi	Used for real time and often voluminous incoming event streams that need to be explorable (e.g. logs, twitter feeds, file appends etc).	
HBase	Supports indexing of stored data, extracting columns, column families, and key information as fields. Although HBase does not use secondary indexing, Cloudera Search can facilitate full-text searches of content in rows and tables in HBase.	Some cases
Cloudera Manager	Deploys, configures, manages, and monitors Cloudera Search processes and resource utilization across services on the cluster. Cloudera Manager helps simplify Cloudera Search administration.	Some cases
Atlas	Atlas classifications drive data access control through Ranger.	Some cases

Component	Contribution	Applicable To
Ranger	Enables role-based, fine-grained authorization for Cloudera Search. Ranger can apply restrictions to various actions, such as accessing data, managing configurations through config objects, or creating collections. Restrictions are consistently enforced, regardless of how users attempt to complete actions. Ranger offers both resource-based and tag-based access control policies.	Some cases
Oozie	Automates scheduling and management of indexing jobs. Oozie can check for new data and begin indexing jobs as required.	Some cases
Hive	Further analyzes search results.	Some cases
Kafka	Search uses this message broker project to increase throughput and decrease latency for handling real-time data.	Some cases
Sqoop	Ingests data in batch and enables data availability for batch indexing.	Some cases

Cloudera Search architecture

Cloudera Search runs as a distributed service on a set of servers, and each server is responsible for a portion of the searchable data. The data is split into smaller pieces, copies are made of these pieces, and the pieces are distributed among the servers. This provides two main advantages:

- Dividing the content into smaller pieces distributes the task of indexing the content among the servers.
- Duplicating the pieces of the whole allows queries to be scaled more effectively and enables the system to provide higher levels of availability.



Each Cloudera Search server can handle requests independently. Clients can send requests to index documents or perform searches to any Search server, and that server routes the request to the correct server.

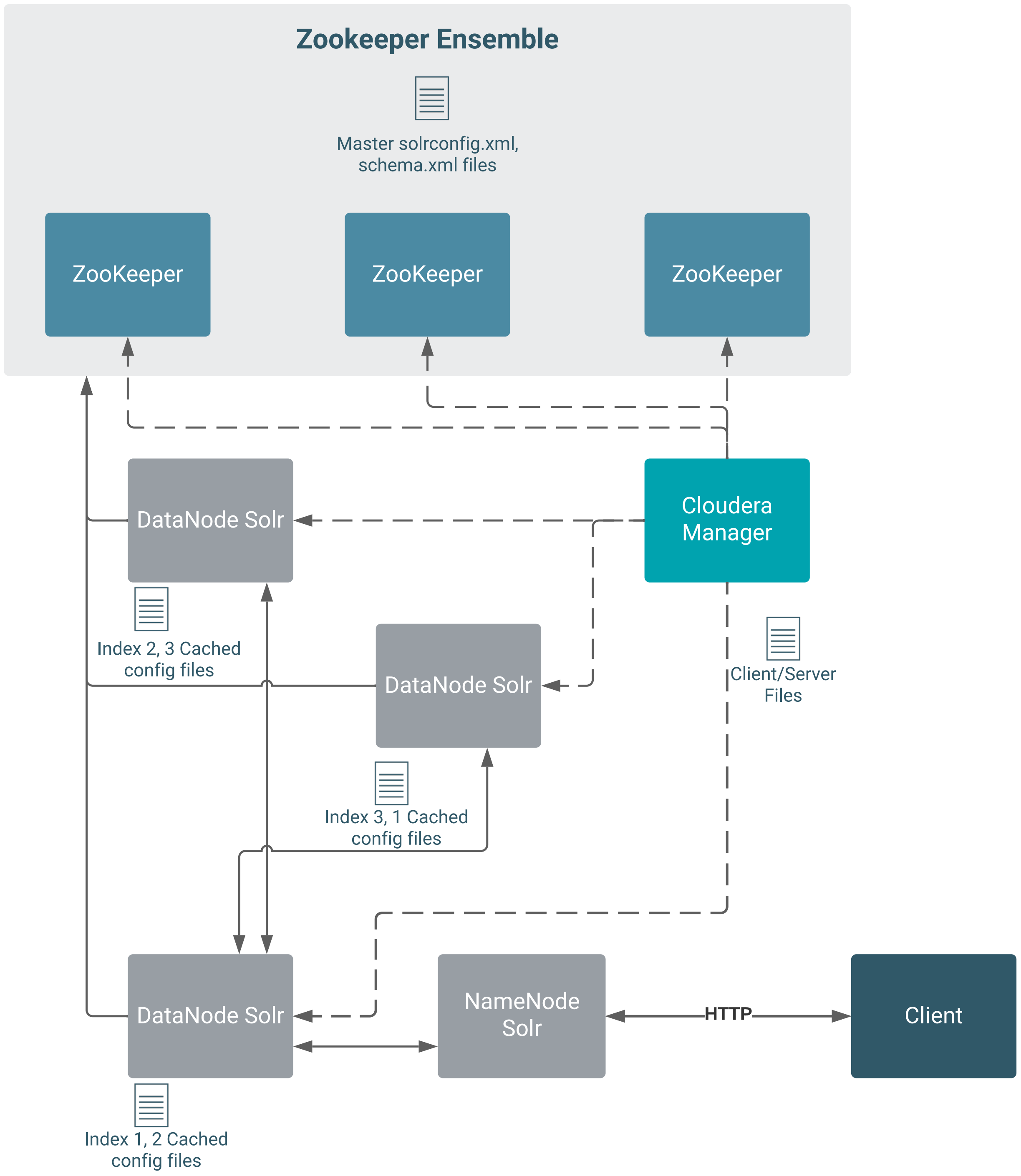
Each Search deployment requires:

- ZooKeeper on at least one host. You can install ZooKeeper, Search, and HDFS on the same host.
- HDFS on at least one, but as many as all hosts. HDFS is commonly installed on all cluster hosts.
- Solr on at least one but as many as all hosts. Solr is commonly installed on all cluster hosts.

More hosts with Solr and HDFS provides the following benefits:

- More Search servers processing requests.
- More Search and HDFS co-location increasing the degree of data locality. More local data provides faster performance and reduces network traffic.

The following graphic illustrates some of the key elements in a typical deployment.



This graphic illustrates:

1. A client submits a query over HTTP.
2. The response is received by the NameNode and then passed to a DataNode.
3. The DataNode distributes the request among other hosts with relevant shards.
4. The results of the query are gathered and returned to the client.

Also notice that the:

- Cloudera Manager provides client and server configuration files to other servers in the deployment.
- ZooKeeper server provides information about the state of the cluster and the other hosts running Solr.

The information a client must send to complete jobs varies:

- For queries, a client must have the hostname of the Solr server and the port to use.
- For actions related to collections, such as adding or deleting collections, the name of the collection is required as well.
- Indexing jobs, such as MapReduceIndexer jobs, use a MapReduce driver that starts a MapReduce job. These jobs can also process morphlines and index the results to Solr.

Local file system support

Learn about the advantages and limitations of using local File System (FS) to store indexes of Cloudera Search.

Apache Lucene-Solr supports various file systems to store index data, including the Hadoop Distributed File System (HDFS) or file systems which are local to the server.

Solr queries perform the best when the index data is available in memory. The HDFS data storage implementation includes various performance tweaks, for example data locality, Short Circuit Data Reads and caching of the HDFS data blocks in memory. For most of the cases this yields acceptable performance, however there are some cases where sub-second response time of queries is critical. In these cases a local index storage (especially using performant Solid State Disks, SSDs) paired with the OS caching, can yield better performance. In the upstream Apache Solr, local FS storage is a widely used and supported feature that is considered stable.

Cloudera curently supports local FS storage for new installations only. The global default behavior of Cloudera Search remains storing indexes on HDFS. If you want to store your indexes locally, you can configure it at collection level, by generating a collection configuration using the solrctl instancedir --generate command with the -localfs argument. This replaces HdfsDirectoryFactory with NRTCachingDirectoryFactory, and HDFS lock factory with native lock factory in the generated solrconfig.xml file. When Solr loads such a collection, it writes the index into a subdirectory of the local data directory (by default var/lib/solr).

Note: If you opt for storing indexes on local FS, it is your responsibility to set quotas on OS level, preventing Solr from consuming all available space under /var, negatively impacting the performance of other services.

Benefits of using local file system for index storage

No need to worry about data locality

The HDFS directory factory attempts to place each HDFS data block on a Data Node which is local to a given Solr server, provided that such a Data Node is up and operational. When the data node goes down, or an HDFS rebalancing operation is triggered in the cluster, the overall data locality of Solr can be ruined which can negatively affect performance. At the moment there is no easy way to restore locality, either you can optimize an index which can have bad side effects, or you use the ADDREPLICA of the Collections API to re-add each replica and then apply DELETEREPLICA on the old replica with bad locality.

With local FS there are no such issues.

Faster disk reads and better caching

The HDFS block cache provides a great help when you have to read data blocks which are not available locally. For the fast execution of a query it is critically important to ensure that a block is available locally - a cache miss can be costly due to the network round trips involved.

Better Query Response Times

Cloudera performed benchmarking to measure document indexing performance, and found that the indexing was roughly two times faster on local FS than on HDFS even when using a magnetic spinning disk.

Table 2: Indexing time in seconds for 10 million records

	HDFS	Local FS	Percentage
2x5 million records, 2x1 thread	3265	2853	87.38
2x5 million records, 2x5 thread	1645	856	52.04
2x5 million records, 2x10 thread	1603	715	44.6
2x5 million records, 2x20 thread	1524	695	45.6
Spark Crunch Indexer, 20 executors	1652	766	46.37
Spark Crunch Indexer, 40 executors	1559	762	48.88
Spark Crunch Indexer, 100 executors	1471	752	51.12

In another case NVMe SSDs were used with a local FS to store the Solr index. In this case the query executions turned out to be three times faster than on HDFS.

Note:

The downsides of this configuration are that replacing an NVMe SSD might require full server downtime (since they are connected directly to the PCI Express bus).

You must take care regarding the RAID configurations, as the local FS does not provide an FS level replication, so the data redundancy has to be provided at the disk level by using a RAID setup and/or at the Solr level by adding more replicas to the collection.

No "Small files problem"

HDFS is sensitive to having a large number of small files stored. Even in spite of Solr periodically merging the segments in the background, an index directory can have many small files. NameNode RPC queues may be periodically overloaded when having a Solr collection with a large number of shards (100+).

Dynamic caching

When using a local FS, caching of the files is managed by the underlying operating system. The operating system can dynamically manage the amount of memory used for caching based on the current memory pressure, whereas the HDFS block cache allocation is more static.

Less impact from other services

By storing the files locally, query performance is not affected by the load on HDFS caused by other services. This is especially important if you have certain query response time SLAs to meet.

Less storage overhead

The index files on HDFS are stored with the same replication factor as configured for the underlying HDFS service. The current Solr implementation allows specifying a custom replication factor for the transaction log files, but for the index files it always uses the replication factor configured for the entire HDFS service even if you want to go with a smaller replication factor.

CDP features that are only available on HDFS

Index data encryption

CDP provides robust, transparent encryption of the HDFS data where the encryption key management is external to HDFS. Only the authorized HDFS clients can access the data, even

HDFS and the operating system interact using encrypted data only. The solution also supports various Hardware Security Modules and CPU instruction sets to accelerate the encryption.

When storing data on a local FS, encryption is also possible but needs to be implemented and managed external to CDP.

Quota support

CDP have built in solutions to define quotas for the data stored on HDFS.

When using a local FS, the disk space taken by Solr indexes is not controlled by CDP, so monitoring the disk usage and ensuring that index files do not eat up all the space has to be done externally.

File system level redundancy

Solr allows collection level fault tolerance using the replica concept. Provided that the replicas of each shard are stored on different machines, this provides enough protection against the failure of a single server. Additionally to this, HDFS is distributed by nature and it provides block level redundancy. This provides more opportunities for disaster recovery - in case of any server going down, the affected replicas can be restarted on any other server, pointing to the same data set.

Flexible cross-cluster backups

The backup implementation provided by Cloudera Search is relying on the hadoop distcp tool to export index snapshots to HDFS, and it also allows transferring them to remote clusters.

Using the MapReduceIndexerTool (MRIT)

MRIT does not work with local file systems as an indexing target, as it creates temporary indexes on the HDFS. After the MapReduce jobs have completed, the client part of the indexer invokes the MERGEINDEXES action of the Solr CoreAdmin API. This only works when the source and target index use the same directory factory.

Using the HBASE MapReduceIndexerTool (HBASE MRIT)

HBASE MRIT does not work with local file systems if the number of reducers are greater than 0 as an indexing target, as it creates temporary indexes on the HDFS. After the MapReduce jobs have completed, the client part of the indexer invokes the MERGEINDEXES action of the Solr CoreAdmin API. This only works when the source and target index use the same directory factory.

Note: HBASE MRIT works correctly with local file systems if the number of reducers are set to 0 (--reducers 0) because in this case documents are sent directly from the mapper tasks to live Solr servers instead of using MERGEINDEXES.

Related Information

Generating Solr collection configuration using instance directories

solrctl Reference

Cloudera Search tasks and processes

For content to be searchable, it must exist in Cloudera Data Platform (CDP) and be indexed. Content can either already exist in CDP and be indexed on demand, or it can be updated and indexed continuously. To make content searchable, first ensure that it is ingested or stored in CDP.

Ingestion

Ingestion is about making data available in Cloudera Data Platform (CDP).

You can move content to CDP by using:

- Apache NiFi, a flexible data streaming framework.
- A copy utility such as distcp for HDFS.
- Sqoop, a structured data ingestion connector.

Related Concepts

Extracting, transforming, and loading data with Cloudera Morphlines

Indexing

Content must be indexed before it can be searched. Learn about how indexing in Cloudera Search happens.

Indexing comprises the following steps:

1. Extraction, transformation, and loading (ETL) - Use existing engines or frameworks such as Apache Tika or Cloudera Morphlines.
   a. Content and metadata extraction
   b. Schema mapping
2. Index creation using Lucene.
   a. Index creation
   b. Index serialization

Indexes are typically stored on a local file system. Lucene supports additional index writers and readers. One HDFS-based interface implemented as part of Apache Blur is integrated with Cloudera Search and has been optimized for CDP-stored indexes. All index data in Cloudera Search is stored in and served from HDFS.

You can index content in the following ways:

Batch indexing using MapReduce

To use MapReduce to index documents, run a MapReduce job on content in HDFS to produce a Lucene index. The Lucene index is written to HDFS, and this index is subsequently used by Search services to provide query results.

Batch indexing is most often used when bootstrapping a Search cluster. The Map phase of the MapReduce task parses input into indexable documents, and the Reduce phase indexes the documents produced by the Map. You can also configure a MapReduce-based indexing job to use all assigned resources on the cluster, utilizing multiple reducing steps for intermediate indexing and merging operations, and then writing the reduction to the configured set of shard sets for the service. This makes the batch indexing process as scalable as MapReduce workloads.

NRT indexing using the API

Other clients can complete NRT indexing. This is done when the client first writes files directly to HDFS and then triggers indexing using the Solr REST API. Specifically, the API does the following:

1. Extract content from the document contained in HDFS, where the document is referenced by a URL.
2. Map the content to fields in the search schema.
3. Create or update a Lucene index.

This is useful if you index as part of a larger workflow. For example, you could trigger indexing from an Oozie workflow.

Indexing using the Spark-Solr connector

Using the Spark-Solr connector you have two options to index data into Solr: batch index documents or index streaming data.

For batch indexing you have multiple options, firstly you can use the spark-shell tool to reach Solr with Scala commands. Cloudera recommends this solution mostly for experimenting or smaller indexing jobs.

The second option is to use spark-submit with your spark job, for this you need to create a class which implements SparkApp.RDDProcesor interface. This way you can access the robustness of Spark with the phrases and concepts well-known from Solr. This works in Scala and Java as well.

If you want to index streaming data, you can do it by implementing the SparkApp.StreamingProcessor interface. With this solution you gain access to all the benefits of SparkStreaming and send on the data to Solr.

Querying

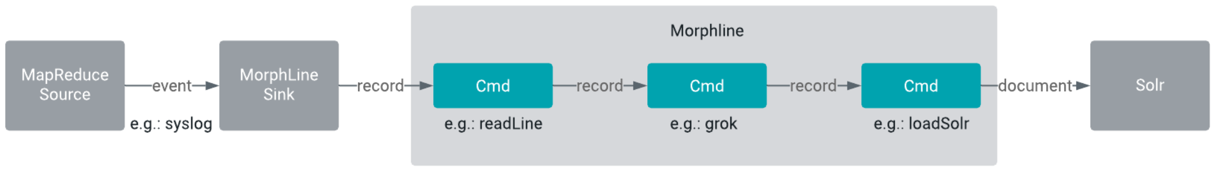
After data is available as an index, you can run queries against it.

The query API provided by the Search service allows direct queries to be completed or to be facilitated through a command-line tool or graphical interface. Hue includes a simple GUI-based Search application, or you can create a custom application based on the standard Solr API. Any application that works with Solr is compatible and runs as a search-serving application for Cloudera Search because Solr is at its core.

Extracting, transforming, and loading data with Cloudera Morphlines

Cloudera Morphlines is an open-source framework that reduces the time and skills required to build or change Search indexing applications.

A morphline is a rich configuration file that simplifies defining an extraction, transformation, and loading (ETL) transformation chain. Use these chains to consume any kind of data from any data source, process the data, and load the results into Cloudera Search. Executing in a small, embeddable Java runtime system, morphlines can be used for near real-time applications as well as batch processing applications. The following diagram shows the process flow:



Morphlines can be seen as an evolution of Unix pipelines, where the data model is generalized to work with streams of generic records, including arbitrary binary payloads. Morphlines can be embedded into Hadoop components such as Search, MapReduce, Hive, and Sqoop.

The framework ships with a set of frequently used high-level transformation and I/O commands that can be combined in application-specific ways. The plug-in system allows you to add new transformations and I/O commands and integrates existing functionality and third-party systems.

This integration enables the following:

- Rapid Hadoop ETL application prototyping
- Complex stream and event processing in real time
- Flexible log file analysis
- Integration of multiple heterogeneous input schemas and file formats
- Reuse of ETL logic building blocks across Search applications

The high-performance Cloudera runtime compiles a morphline, processing all commands for a morphline in the same thread and adding no artificial overhead. For high scalability, you can deploy many morphline instances on a cluster in many MapReduce tasks.

The following components run morphlines:

- MapReduceIndexerTool
- Lily HBase Indexer

Cloudera also provides a corresponding Cloudera Search Tutorial.

Data Morphlines support

Morphlines manipulate continuous or arbitrarily large streams of records. The data model can be described as follows: A record is a set of named fields where each field has an ordered list of one or more values. A value can be any Java Object. That is, a record is essentially a hash table where each hash table entry contains a String key and a list of Java Objects as values. (The implementation uses Guava's ArrayListMultimap, which is a ListMultimap). Note that a field can have multiple values and any two records need not use common field names. This flexible data model corresponds exactly to the characteristics of the Solr/Lucene data model, meaning a record can be seen as a SolrInpu tDocument. A field with zero values is removed from the record - fields with zero values effectively do not exist.

Not only structured data, but also arbitrary binary data can be passed into and processed by a morphline. By convention, a record can contain an optional field named _attachment_body, which can be a Java java.io.InputStream or Java byte[]. Optionally, such binary input data can be characterized in more detail by setting the fields named _att achment_mimetype (such as application/pdf) and _attachment_charset (such as UTF-8) and _attachment_name (such as cars.pdf), which assists in detecting and parsing the data type.

This generic data model is useful to support a wide range of applications.

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.

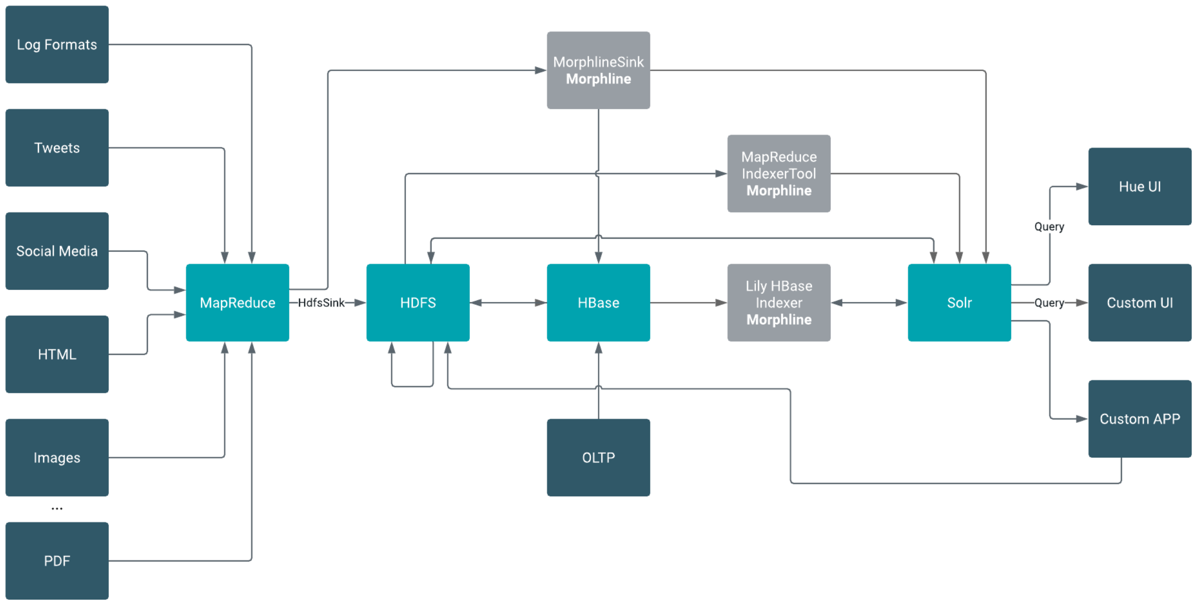
How Morphlines act on data

A command transforms a record into zero or more records. Commands can access all record fields. For example, commands can parse fields, set fields, remove fields, rename fields, find and replace values, split a field into multiple fields, split a field into multiple values, or drop records. Often, regular expression based pattern matching is used as part of the process of acting on fields. The output records of a command are passed to the next command in the chain. A command has a Boolean return code, indicating success or failure.

For example, consider the case of a multi-line input record: A command could take this multi-line input record and divide the single record into multiple output records, one for each line. This output could then later be further divided using regular expression commands, splitting each single line record out into multiple fields in application specific ways.

A command can extract, clean, transform, join, integrate, enrich and decorate records in many other ways. For example, a command can join records with external data sources such as relational databases, key-value stores, local files or IP Geo lookup tables. It can also perform tasks such as DNS resolution, expand shortened URLs, fetch linked metadata from social networks, perform sentiment analysis and annotate the record accordingly, continuously maintain statistics for analytics over sliding windows, compute exact or approximate distinct values and quantiles.

A command can also consume records and pass them to external systems. For example, a command can load records into Solr or write them to a MapReduce Reducer or pass them into an online dashboard. The following diagram illustrates some pathways along which data might flow with the help of morphlines:



Morphline characteristics

A command can contain nested commands. Thus, a morphline is a tree of commands, akin to a push-based data flow engine or operator tree in DBMS query execution engines.

A morphline has no notion of persistence, durability, distributed computing, or host failover. A morphline is basically just a chain of in-memory transformations in the current thread. There is no need for a morphline to manage multiple processes, hosts, or threads because this is already addressed by host systems such as MapReduce or Storm. However, a morphline does support passing notifications on the control plane to command subtrees. Such notifications include BEGIN_TRANSACTION, COMMIT_TRANSACTION, ROLLBACK_TRANSACTION, SHUTDOWN.

The morphline configuration file is implemented using the HOCON format (Human-Optimized Config Object Notation). HOCON is basically JSON slightly adjusted for configuration file use cases. HOCON syntax is defined at HOCON github page and is also used by Akka and Play.

How Morphlines are implemented

Cloudera Search includes several maven modules that contain morphline commands for integration with Apache Solr including SolrCloud, flexible log file analysis, single-line records, multi-line records, CSV files, regular expression based pattern matching and extraction, operations on record fields for assignment and comparison, operations on record fields with list and set semantics, if-then-else conditionals, string and timestamp conversions, scripting support for dynamic Java code, a small rules engine, logging, metrics and counters, integration with Avro, integration with Apache Tika parsers, integration with Apache Hadoop Sequence Files, auto-detection of MIME types from binary data using Apache Tika, and decompression and unpacking of arbitrarily nested container file formats, among others.

Backing up and restoring data using Search

Cloudera Search includes a backup and restore mechanism primarily designed to provide disaster recovery capability for Apache Solr. You can create a backup of a Solr collection by creating and exporting a snapshot and restoring from this backup to recover from scenarios, such as accidental or malicious data modification or deletion.

Important: Backing up and restoring Cloudera Search is based on the SolrCloud implementation of Apache Solr. Cloudera Search does not support backups using the Solr replication handler.

Cloudera Search includes a backup/restore mechanism primarily designed to provide disaster recovery capability for Apache Solr. You can create a backup of a Solr collection and restore from this backup if the index is corrupted due to a software bug, or if an administrator accidentally or maliciously deletes a collection or a subset of documents. This procedure can also be used as part of a cluster migration (for example, if you are migrating to a cloud environment), or to recover from a failed upgrade.

At a high level, the steps to back up a Solr collection are as follows:

1. Create a snapshot.
2. If you are exporting the snapshot to a remote cluster, prepare the snapshot for export.
3. Export the snapshot to either the local cluster or a remote one. This step uses the Hadoop DistCP utility.

The backup operation uses the built-in Solr snapshot capability to capture a point-in-time, consistent state of index data for a specified Solr collection. You can then use the Hadoop DistCp utility to copy the index files and the associated metadata for that snapshot to a specified location in HDFS or a cloud object store (for example, Amazon S3).

The Solr snapshot mechanism is based on the Apache Lucene IndexDeletionPolicy abstraction, which enables applications such as Cloudera Search to manage the lifecycle of specific index commits. A Solr snapshot assigns a user-specified name to the latest hard-committed state. After the snapshot is created, the Lucene index files associated with the commit are retained until the snapshot is explicitly deleted. The index files associated with the snapshot are preserved regardless of document updates and deletions, segment merges during index optimization, and so on.

Creating a snapshot does not take much time because it only preserves the snapshot metadata and does not copy the associated index files. A snapshot does have some storage overhead corresponding to the size of the index because the index files are retained indefinitely until the snapshot is deleted.

Solr snapshots can help you recover from some scenarios, such as accidental or malicious data modification or deletion. They are insufficient to address others, such as index corruption and accidental or malicious administrative action (for example, deleting a collection, changing collection configuration, and so on). To address these situations, export snapshots regularly and before performing non-trivial administrative operations such as changing the schema, splitting shards, or deleting replicas.

Exporting a snapshot exports the collection metadata as well as the corresponding Lucene index files. This operation internally uses the Hadoop DistCp utility to copy the Lucene index files and metadata, which creates a full backup at the specified location. After the backup is created, the original Solr snapshot can be safely deleted if necessary.

Important: If you create a snapshot and do not export it, you do not have a complete backup, and cannot restore index files if they are accidentally or maliciously deleted.

Related Information

Apache Lucene IndexDeletionPolicy

Backups using the Solr replication handler