Cloudera Runtime 7.1.0

Troubleshooting Cloudera Search

Date published: 2019-11-19
Date modified:

https://docs.cloudera.com/

Legal Notice

Contents

Troubleshooting Cloudera Search

After installing and deploying Cloudera Search, use the information in this section to troubleshoot problems.

Troubleshooting

The following table contains some common troubleshooting techniques.

Note: In the URLs in the following table, you must replace entries such as <server:port> with values from your environment. The port defaults value is 8983, but see /etc/default/solr or /opt/cloudera/parcels/CDH-*/etc/default/solr for the port if you are in doubt.

Symptom	Explanation	Recommendation
All	Varied	Examine Solr log. By default, the log can be found at /var/log/solr/solr.out.
No documents found	Server may not be running	Browse to http://*SERVER*:*PORT*/solr to see if the server responds. Check that cores are present. Check the contents of cores to ensure that numDocs is more than 0.
No documents found	Core may not have documents	Browsing http://server:port/solr/[collection name]/select?q=*:*&wt=json&indent=true should show numFound, which is near the top, to be more than 0.
The secure Solr Server fails to respond to Solrj requests, but other clients such as curl can communicate successfully	This may be a version compatibility issue. Httpclient 4.2.3, which ships with solrj in Search 1.x, has a dependency on commons-codec 1.7. If an earlier version of commons-codec is on the classpath, httpclient may be unable to communicate using Kerberos.	Ensure your application is using commons-codec 1.7 or higher. Alternatively, use httpclient 4.2.5 instead of version 4.2.3 in your application. Version 4.2.3 behaves correctly with earlier versions of commons-codec.

Dynamic Solr Analysis

Any JMX-aware application can query Solr for information and display results dynamically. For example, Zabbix, Nagios, and many others have been used successfully. When completing Static Solr Log Analysis, many of the items related to extracting data from the log files can be seen from querying Solr, at least the last value (as opposed to the history which is available from the log file). These are often useful for status boards. In general, anything available from the Solr admin page can be requested on a live basis from Solr. Some possibilities include:

- numDocs/maxDoc per core. This can be important since the difference between these numbers indicates the number of deleted documents in the index. Deleted documents take up disk space and memory. If these numbers vary greatly, this may be a rare case where optimizing is advisable.
- Cache statistics, including:
  - Hit ratios
  - Autowarm times
  - Evictions
- Almost anything available on the admin page. Note that drilling down into the “schema browser” can be expensive.

Other Troubleshooting Information

Since the use cases for Solr and search vary, there is no single solution for all cases. That said, here are some common challenges that many Search users have encountered:

- Testing with unrealistic data sets. For example, a users may test a prototype that uses faceting, grouping, sorting, and complex schemas against a small data set. When this same system is used to load of real data, performance issues occur. Using realistic data and use-cases is essential to getting accurate results.
- If the scenario seems to be that the system is slow to ingest data, consider:
  - Upstream speed. If you have a SolrJ program pumping data to your cluster and ingesting documents at a rate of 100 docs/second, the gating factor may be upstream speed. To test for limitations due to upstream speed, comment out only the code that sends the data to the server (for example, SolrHttpServer.add(doclist)) and time the program. If you see a throughput bump of less than around 10%, this may indicate that your system is spending most or all of the time getting the data from the system-of-record.
  - This may require pre-processing.
  - Indexing with a single thread from the client. ConcurrentUpdateSolrServer can use multiple threads to avoid I/O waits.
  - Too-frequent commits. This was historically an attempt to get NRT processing, but with SolrCloud hard commits this should be quite rare.
  - The complexity of the analysis chain. Note that this is rarely the core issue. A simple test is to change the schema definitions to use trivial analysis chains and then measure performance.
  - When the simple approaches fail to identify an issue, consider using profilers.

Cloudera Search Configuration and Log Files

Cloudera Search (powered by Apache Solr) configuration is managed using Cloudera Manager. You can view log files on the individual Solr server or using Cloudera Manager.

Identifying problems in a Cloudera Search deployment

Learn about common issues affecting Search performance and what you can do about them.

To investigate your Cloudera Search deployment for performance problems or misconfigurations, inspect the log files, schema files, and the actual index for issues. If possible, connect to the live Solr instance while watching log files so you can compare the schema with the index. For example, the schema and the index can be out of sync in situations where the schema is changed, but the index was never rebuilt. See the following list for some common issues and what you can do about them:

- A high number or proportion of 0-match queries. This indicates that the user-facing part of the application is making it easy for users to enter queries for which there are no matches. In Cloudera Search, given the size of the data, this should be an extremely rare event.
- Queries that match an excessive number of documents. All documents that match a query have to be scored, and the cost of scoring a query goes up as the number of hits increases. Examine any frequent queries that match millions of documents. An exception to this case is “constant score queries”. Queries, such as those of the form ":" bypass the scoring process entirely.
- Overly complex queries. Defining what constitutes overly complex queries is difficult to do, but a very general rule is that queries over 1024 characters in length are likely to be overly complex.
- High autowarm times. Autowarming is the process of filling caches. Some queries are run before a new searcher serves the first live user request. This keeps the first few users from having to wait. Autowarming can take many seconds or can be instantaneous. Excessive autowarm times often indicate excessively generous autowarm parameters. Excessive autowarming usually has limited benefit, with longer runs effectively being wasted work.
  - Cache autowarm. Each Solr cache has an autowarm parameter. You can usually set this value to an upper limit of 128 and tune from there.

    - FirstSearcher/NewSearcher. The solrconfig.xml file contains queries that can be fired when a new searcher is opened (the index is updated) and when the server is first started. Particularly for firstSearcher, it can be valuable to have a query that sorts relevant fields.

Note: The aforementioned flags are available from solrconfig.xml

- Exceptions. The Solr log file contains a record of all exceptions thrown. Some exceptions, such as exceptions resulting from invalid query syntax are benign, but others, such as Out Of Memory, require attention.
- Excessively large caches. The size of caches such as the filter cache are bounded by maxDoc/8. Having, for instance, a filterCache with 10,000 entries is likely to result in Out Of Memory errors. Large caches occurring in cases where there are many documents to index is normal and expected.
- Caches with low hit ratios, particularly filterCache. Each cache takes up some space, consuming resources. There are several caches, each with its own hit rate.
    - filterCache. This cache should have a relatively high hit ratio, typically around 80%.
    - queryResultCache. This is primarily used for paging so it can have a very low hit ratio. Each entry is quite small as it is basically composed of the raw query as a string for a key and perhaps 20-40 ints. While useful, unless users are experiencing paging, this requires relatively little attention.
    - documentCache. This cache is a bit tricky. It's used to cache the document data (stored fields) so various components in a request handler don't have to re-read the data from the disk. It's an open question how useful it is when using MMapDirectory to access the index.
- Very deep paging. Users seldom go beyond the first page and very rarely to go through 100 pages of results. A &start=<pick your number> query indicates unusual usage that should be identified. Deep paging may indicate some agent is completing scraping.

Note: Solr is not built to return full result sets no matter how deep. If returning the full result set is required, explore alternatives to paging through the entire result set.

- Range queries should work on trie fields. Trie fields (numeric types) store extra information in the index to aid in range queries. If range queries are used, it's almost always a good idea to be using trie fields.
- fq clauses that use bare NOW. fq clauses are kept in a cache. The cache is a map from the fq clause to the documents in your collection that satisfy that clause. Using bare NOW clauses virtually guarantees that the entry in the filter cache is not be re-used.
- Multiple simultaneous searchers warming. This is an indication that there are excessively frequent commits or that autowarming is taking too long. This usually indicates a misunderstanding of when you should issue commits, often to simulate Near Real Time (NRT) processing or an indexing client is improperly completing commits. With NRT, commits should be quite rare, and having more than one simultaneous autowarm should not happen.
- Stored fields that are never returned (fl= clauses). Examining the queries for fl= and correlating that with the schema can tell if stored fields that are not used are specified. This mostly wastes disk space. And fl=* can make this ambiguous. Nevertheless, it's worth examining.
- Indexed fields that are never searched. This is the opposite of the case where stored fields are never returned. This is more important in that this has real RAM consequences. Examine the request handlers for "edismax" style parsers to be certain that indexed fields are not used.
- Queried but not analyzed fields. It's rare for a field to be queried but not analyzed in any way. Usually this is only valuable for "string" type fields which are suitable for machine-entered data, such as part numbers chosen from a pick-list. Data that is not analyzed should not be used for anything that humans enter.
- String fields. String fields are completely unanalyzed. Unfortunately, some people confuse string with Java's Stri ng type and use them for text that should be tokenized. The general expectation is that string fields should be used sparingly. More than just a few string fields indicates a design flaw.
- Whenever the schema is changed, re-index the entire data set. Solr uses the schema to set expectations about the index. When schemas are changed, there's no attempt to retrofit the changes to documents that are currently indexed, but any new documents are indexed with the new schema definition. So old and new documents can have the same field stored in vastly different formats (for example, String and TrieDate) making your index inconsistent. This can be detected by examining the raw index.

- Query stats can be extracted from the logs. Statistics can be monitored on live systems, but it is more common to have log files. Here are some of the statistics you can gather:
  - Longest running queries
  - 0-length queries
  - average/mean/min/max query times
  - You can get a sense of the effects of commits on the subsequent queries over some interval (time or number of queries) to see if commits are the cause of intermittent slowdowns
- Too-frequent commits have historically been the cause of unsatisfactory performance. This is not so important with NRT processing, but it is valuable to consider.
- Optimizing an index, which could improve search performance before, is much less necessary now. Anecdotal evidence indicates optimizing may help in some cases, but the general recommendation is to use expungeDeletes, instead of committing.
  - Modern Lucene code does what optimize used to do to remove deleted data from the index when segments are merged. Think of this process as a background optimize. Note that merge policies based on segment size can make this characterization inaccurate.
  - It still may make sense to optimize a read-only index.
  - Optimize is now renamed forceMerge.