MySQL Load balancing and Solr Performance Tuning

Zhuyang Wang
515030910543
wzy1997@sjtu.edu.cn

May 28, 2018

1 Solr Performance Tuning

Solr[1] is highly reliable, scalable and fault tolerant, providing distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration and more. Solr powers the search and navigation features of many of the world's largest internet sites.

However, configuring Apache Solr properly is critical for production system stability and performance. It can be hard to find the right balance between competing goals. There are also multiple factors, implicit or explicit, that need to be taken into consideration.

In this project, we investigate a running Solr instance, which provides searching and faceting utilities for the website Acemap[4], and we make some improvements on different aspects.

It is easy to setup a simple workable Solr instance through the following steps:

1. Define a schema. The schema tells Solr about the format of documents it will be indexing. For example, `year` should be of type `int`, `title` is `string`, and `keyword` is a list of `string`.

2. Index data for which our users will search. Solr will generate several index files for further usage.

And now we can search via RESTful API provided by Solr from the backend or frontend using simple HTTP request URLs.

It is easy to see that there are few critical points regard to performance. First is a well defined schema. Even a simple modification of schema can largely influence searching performance and the index size. We will discuss schema design in 1.1. The index file size also matters. For small index, Solr can directly load it into memory. But as the index size grows we may not have enough memory to hold the whole index. We will talk about this in 1.2. Another important factor is cache size for different searching requests, which will be discussed in 1.3. What's more, since Solr is running on JVM, it is also important to tune JVM parameters. See 1.4 for details. Finally in 1.5 we give an example in which we speed up the loading process of some charts by faster faceting and asynchronous fetching.

1.1 Schema Design

Solr's schema is a single XML file that stores the details about the fields and field types Solr is expected to understand. The schema defines not only the field or field type names, but also any modifications that should happen to a field before it is indexed. For example, we may want to normalize all the words in a paper, or we may want to categorize searching result by keywords. These rules are defined in schema.

Solr has a schemaless way to index data. Schemaless means Solr will try to guess the field itself so we can start using Solr without having to define all the fields we trying to index. However there exist limitations. If it guesses wrong, we can do nothing but reindex the whole data. It may be preferable to use schemaless way on small dataset, but in our project it is totally not recommended and we must define a suitable schema ourselves.

After some investigation we found some parts of original schema can be further improved. For each field, Solr has two attributes `indexed` and `stored` which are important. Indexed fields are fields which undergo an analysis phase, and are added to the index. If a field is not indexed, it cannot be searched on. And the purpose of the stored attribute is to tell Solr to store the original text in the index somewhere so the actual value of the field can be retrieved by queries. These two attributes are independent and orthogonal. By carefully turning on or off thse two attributes we can reduce the index file size and increase the overall searching speed.

Another problem is that in the original schema, there exist two isomor-

phic fields called `_text_` and `_entext_`, which means they contain the same and duplicate data. These two `copyFields`, which is prepared for matching multiple default fields, result in large index file.

Besides, fields need faceting should turn on the attribute `docValues`, which can dramatically increase faceting speed. This attribute is turned on defaultly for field with type `int` and `string`, hence we do not have to manually configure it.

1.2 In-memory Index

Solr uses custom `DirectoryFactory` to handle index. The default implementation `solr.StandardDirectoryFactory` is filesystem based, and tries to pick the best implementation for the current JVM and platform. We can force a particular implementation specifying `solr.MMapDirectoryFactory`, `solr.NIOFSDirectoryFactory`, or `solr.SimpleFSDirectoryFactory`.

The one used in our project is `NRTCachingDirectoryFactory`, which wraps `solr.StandardDirectoryFactory` and caches small files in memory for better NRT(Near Real Time) performance. After a look at the code[3], we found that `StandardDirectoryFactory` should use mmap if the OS and Java version support it. If support isn't there, it will use conventional file access methods. As far as we know, all 64-bit Java versions and 64-bit operating systems will support mmap.

Reasons to use `MmapDirectory` on 64-bit platforms are detaily discussed in [2]. Using mmap we can allocate as less as possible heap space for Java since our index is in OS's disk cache, which is also very friendly to the Java garbage collector.

Because `NRTCachingDirectoryFactory` is best suited for frequent soft commits, which is not needed by our project, we may directly force the mmap implementation in the future and compare performance.

1.3 Cache

Solr caches are associated with an Index Searcher — a particular 'view' of the index that doesn't change. So as long as that Index Searcher is being used, any items in the cache will be valid and available for reuse. Caching in Solr is unlike ordinary caches in that Solr cached objects will not expire after a certain period of time; rather, cached objects will be valid as long as the Index Searcher is valid.

The current Index Searcher serves requests and when a new searcher is opened, the new one is auto-warmed while the current one is still serving external requests. When the new one is ready, it will be registered as the current searcher and will handle any new search requests. The old searcher will be closed after all request it was servicing finish. The current Searcher is used as the source of auto-warming. When a new searcher is opened, its caches may be prepopulated or "autowarmed" using data from caches in the old searcher.

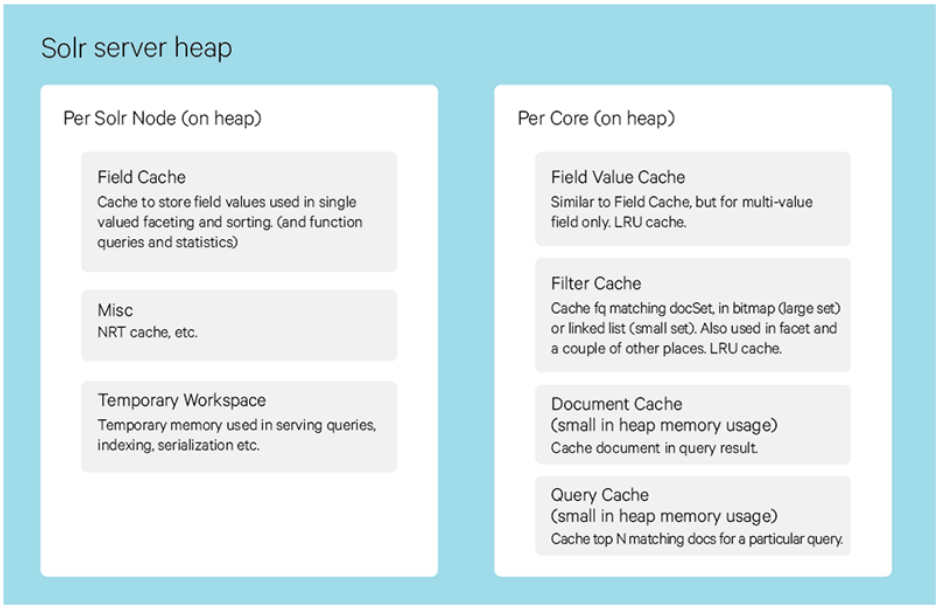

Figure 1: Solr Server Memory

There are few caches available for faster searching. Filter cache is used by SolrIndexSearcher for filters (DocSets), unordered sets of all documents that match a query. When a new searcher is opened, its caches may be prepopulated or autowarmed using data from caches in the old searcher. Query result cache caches results of searches - ordered lists of document ids (DocList) based on a query, a sort, and the range of documents requested. Document cache caches Lucene Document objects (the stored fields for each document). Since Lucene internal document ids are transient, this cache will not be autowarmed.

We can increase corresponding cache size to achieve better search performance. For facet cache, the related caches are field value cache and filter

Category	Keyword	Conference	Journal	Author	Year
Total Time/s	**2.34**	0.33	0.15	0.77	0.18
Facet Time/ms	30.5	25.96	24.17	**459.81**	24.86

Table 1: Time of fetching data of different categories

cache. Since field value cache is not declared in current `solrconfig.xml`, it is generated automatically with an initial size 10, a max size of 10000, and no auto warming.[5] We can increase initial size and enable auto warming to speed up facet.

1.4 JVM tuning

Solr instance needs a lot of allocated memory in our situation, and the Java virtual machine will sometimes be out of memory. Whence we should increase the initial memory size and maximal memory size for Java virtual machine. However, too large memory for JVM may lead to slow garbage collection. We should test different garbage collectors like parallel and generational in the future.

1.5 Faster Faceting

Facet functionality is used heavily in searching result page[4]. It will count papers published in different years and present the result in a chart, as well as other charts with various types of facet information.

Though these charts are fantastic, it upsets us that the loading time is too long, usually taking three or more seconds to fetch data and render charts 2. The problem is that facet is often slower than plain query since it must iterate over all matching documents to count. What's more, in addition to facet result, the keyword chart needs much more data from MySQL database. Actually it takes 2.34 seconds on average to fetch keyword related data.

We conduct some experiments about fetching time and the result is shown in Table 1. We randomly choose 100 words from dictionary, make query for different categories and calculate the average time it takes. It is suggested that the reason keyword chart loads slowly is database operations but not Solr facet. We also discover that faceting author ID takes a lot of time, because there are too many authors in our system.

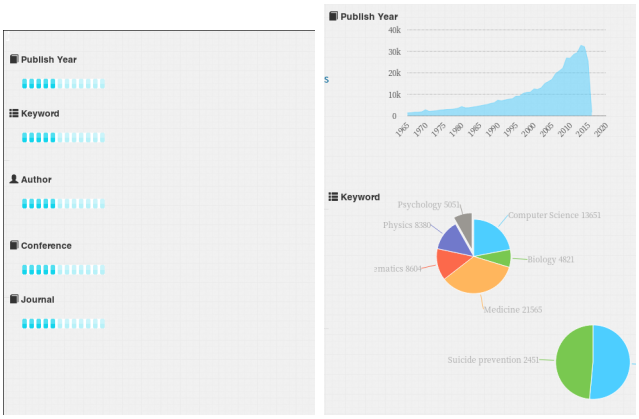

Figure 2: Loading charts

To address this problem, we first separate that one huge faceting request into 5 asynchronous requests. This trivial modification dramatically improve the loading speed of some charts like paper published year. Further more improvements can be witnessed after facet performance tuning.

2 MySQL Load balancing

Modern high-traffic websites, like Acemap, must serve hundreds of thousands, if not millions, of concurrent requests from users or clients and return the correct text, images, video, or application data, all in fast and reliable manner. To cost-effectively scale to meet these high volumes, modern computing best practice generally requires adding more servers.

There exist many load balancing algorithms. Different algorithms provide different benefits:

Random Choice Requests are randomly sent to server.

Round Robin Requests are distributed across the group of servers sequentially.

Weighted Round Robin Requests are distributed sequentially with some weight.

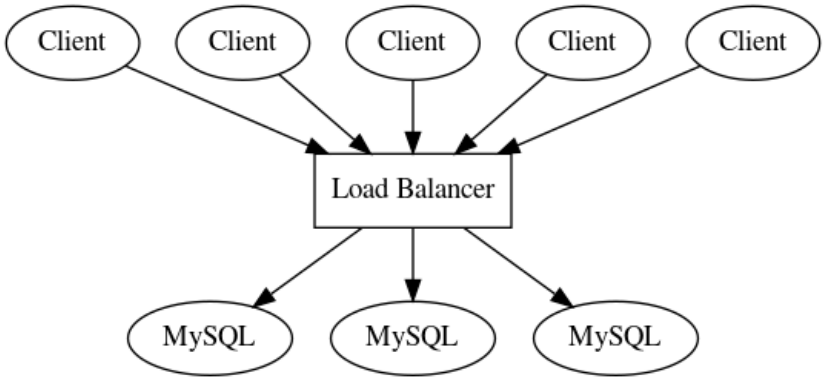

Figure 3: MySQL cluster and load balancer

Least Connections A new request is sent to the server with the fewest current connections to clients. The relative computing capacity of each server is factored into determining which one has the least connections.

IP Hash The IP address of the client is used to determine which server receives the request.

Currently we use round robin algorithm to implement load balancing. The ultimate goal is to consider each MySQL server's load, capacity, CPU, memory status. Until now we have done load balancing for one table, which will be extended to all the tables in the future. Failure detection is also a must for load balancing, as well as master-slave auto synchronization.

Acknowledgements

I would like to express my sincere thanks to Prof. Xinbing Wang and Luoyi Fu for thier inspiring lectures in this semester. They showed me the golden principles to do great research. I am also extremely thankful to Yuting Jia and Xiaoyang Huo under whose guidance and assistance I finally accomplished this project and I came to know lots of new things. I would also like to thank people who has ever helped me during this project. Finally I would not forget Apache Solr team for their excellent wiki and detailed reference.

References

[1] The Apache Software Foundation. *Apache Sorl*. URL: https://lucene.apache.org/solr/.

[2] Uwe Schindler. *Use Lucene's MMapDirectory on 64bit platforms, please!* URL: http://blog.thetaphi.de/2012/07/use-lucenes-mmapdirectory-on-64bit.html.

[3] Eric Torti. *Is solr.StandardDirectoryFactory an MMapDirectory?* URL: http://grokbase.com/t/lucene/solr-user/15a79gakvs/is-solr-standarddirectoryfactory-an-mmapdirectory.

[4] Acemap Inc. Shanghai Jiao Tong University. *Acemap*. URL: http://acemap.sjtu.edu.cn/.

[5] Solr Wiki. *SolrCaching*. URL: https://wiki.apache.org/solr/SolrCaching.