Improving Consistency in SolrCloud

Darsh Shah
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA
darshs@andrew.cmu.edu

Yifan Li
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA
yifanl@andrew.cmu.edu

ABSTRACT

SolrCloud is the distributed search engine from Apache Solr project. There is a known bug in the SolrCloud community that results in lost writes in the SolrCloud system. In this report, we studied the architecture of the SolrCloud system and used a simulation system to address the consistency problems around this bug. We have proposed a quorum based writing strategy to handle the writes. In this case, we have favored consistency over write availability. In the evaluation part, we have shown that this new quorum based writing strategy is worth the price of availability, as it will improve the query success rate in a large-scale real world deployment under heavy write workloads.

1. INTRODUCTION

Apache Solr (pronounced "solar") is an open source enterprise search platform, written in Java, from the Apache Lucene project. Its major features include full-text search, hit highlighting, faceted search, real-time indexing, dynamic clustering, database integration, NoSQL features and rich document handling. SolrCloud, which is a part of Solr, makes it possible to set up a cluster of Solr servers that combines fault tolerance and high availability. These capabilities provide distributed indexing and search capabilities.[1][2]

In the current SolrCloud implementation, the leader will accept writes even if all the replicas are down. And if the leader goes down and a replica rejoins and becomes the leader, it will not have the recent writes from the old leader (which is dead). Hence, all the queries for this missing data will fail. Here, the design favors write availability over consistency. This bug SOLR-5468 [3] is known in the Solr community and we have reproduced the bug in the next section. Currently, as a solution to this bug, SolrCloud puts the onus on the client/user to check if there exists a quorum and to risk of losing the writes in case of no quorum.

We propose a quorum based write design, where the write requests are accepted only when a majority of replicas are alive. This proposed method would always favor consistency over write availability. We created a simulation system to show that our solution works. We simulate almost every aspect of the SolrCloud design and do evaluations to prove that our design achieves consistency over availability although it takes a minor hit in write performance.

2. DESCRIPTION

2.1 Apache SolrCloud

To better understand the problem we are addressing, we reproduced the bug on the actual Solr cloud distribution. We downloaded Solr 5.0.0 version from [4] and using the inbuilt command line interface, we created a 2-shard 3-replica system. Each shard has a leader and two replicas. We wrote data so that each shard has some amount of data to be queried initially. SolrCloud provides a web interface to see the system state. Fig 1 shows the current state. We killed two replicas on shard2 as seen in Fig 2 and added new writes to the system. The leader on shard2 accepted the writes. Specifically, the docs with id – darsh, cmu and daiict were accepted and stored on shard2 even though other two replicas on shard2 were dead. We queried the newly written data and got back the new writes successfully as shown in Fig 4. We queried for id=darsh,daiict,cmu and we got 3 docs in response. Then we killed the leader on shard2 and started an existing node that became the leader of shard2 as seen in Fig 3. Now when we query shard2 for the same docs, we get zero results as shown in Fig 5. This is because the writes on the old leader (which died) were not synced with the new leader. Hence, we lose writes here.

More detailed explanation of the steps and corresponding code can be found at [5][6][7].

2.2 Proposed Solution

The solution we propose for the above problem is to enforce a quorum based writing. Quorum based writing will ensure that the replication factor (live nodes) is greater than half the total nodes in the system. For example, in a 3-node system containing one leader and two replicas, there should be at least two nodes alive (including the leader) to accept the writes.

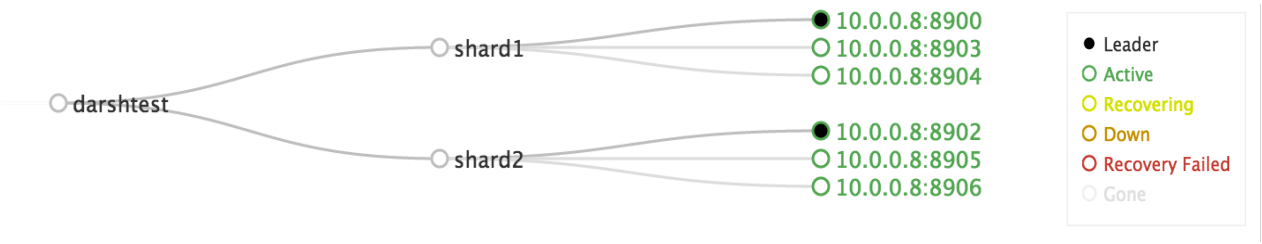

Figure 1: All nodes are healthy and running

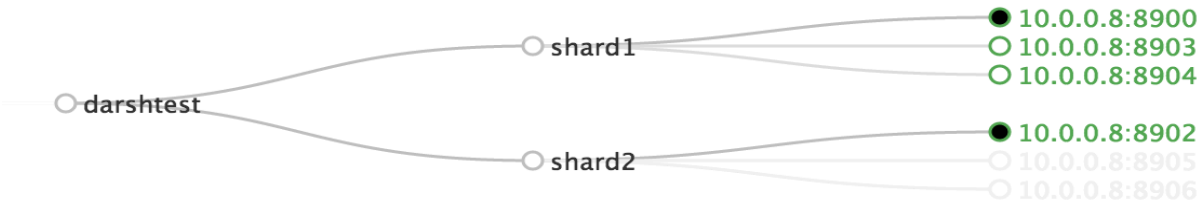

Figure 2: Only the leader is alive on shard2. Both shard2 replicas are dead

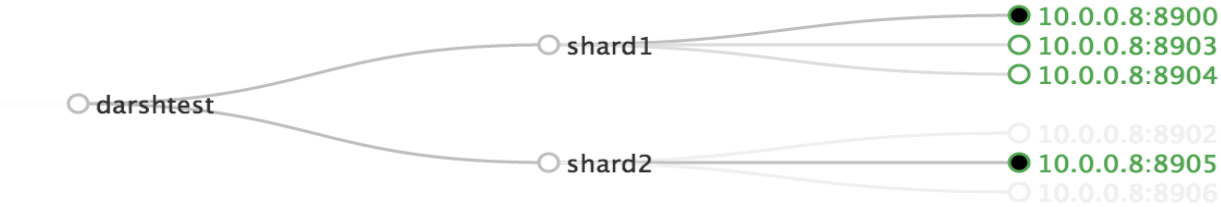

Figure 3: New leader on shard2. Other two nodes are dead

```
  <lst name="params">
    <str name="q">id=darsh,daiict,cmu</str>
    <str name="distrib">false</str>
    <str name="indent">true</str>
    <str name="wt">xml</str>
    <str name="_">1428298834793</str>
  </lst>
</lst>
<result name="response" numFound="3" start="0">
```

Figure 4: Three docs found

```
  <lst name="params">
    <str name="q">id=darsh,daiict,cmu</str>
    <str name="distrib">false</str>
    <str name="indent">true</str>
    <str name="wt">xml</str>
    <str name="_">1428298915017</str>
  </lst>
</lst>
<result name="response" numFound="0" start="0">
```

Figure5: Zero docs found

If there is no quorum, then there is a possibility that the writes diverge. For example, assume that the leader and two replicas are both alive and up to date with each other and assume the last common write has id 70. Suppose the two replicas die and the leader accepts 10 new writes. So, now the leader is at id 80. Now, replica 1 comes up and sends a sync message to the leader. While syncing, the leader dies such that only 5 writes (id 71-75) are synced to the replica 1. Now, since the old leader died, replica 1 is the new leader and will start accepting writes and storing them after id 75. Hence, the writes from id 76-80 will be different on the current leader from the old leader. This divergence is caused because there is no write quorum.

Our proposed solution provides consistency over write availability and we show that our designed system achieves the desired result over the existing system using a simulation as described in the next few sections.

3. SIMULATION SYSTEM DESIGN

To simulate the original SolrCloud system, we designed the new system as shown in the Figure 6. The three major parts of the system are client, LoadBalancer and nodes. The client is simulating the SolrCloud search engine client, where it is indexing new data into the replicas and also queries data. The LoadBalancer is playing a different role in the simulation system. It serves requests from clients, maintain status of nodes and manage election process of nodes. There are two types of nodes. One is leader, which will accept write and query, and sync data with replica node. Another one is replica, which will accept sync writes from leader and also be prepared to be potential leader. The design principles

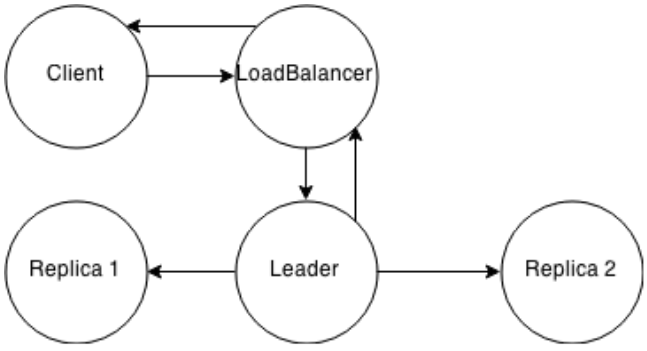

Figure 6: Simulation System Design

we use during the process are simple. We are creating this system mainly to recover the original write consistency bug in the SolrCloud system. We want to keep other components simple and abstract. Following this principle, we have simplified the index and query into key value pair query and writes. The detail design of each part is discussed below.

3.1 Important System Components

In this section, we will describe in depth about the LoadBalancer, Nodes and Client design.

3.1.1 LoadBalancer Design

LoadBalancer in our system is a machine that handles client requests, communicate with nodes, and maintain election process and nodes status with heartbeat. It is a relatively centralized part in our system as compared to other parts. Leader election and heartbeat will be discussed later in the report. In this part, we will cover more design detail of client requests handling and communication between nodes.

A client server thread that is part of LoadBalancer process is mainly responsible to listen to the requests from the client server. There are three kinds of requests that are client joining, query request and write request from the client. Whenever a new client joins, the LoadBalancer creates a new socket dedicated for this new client. The query requests are always be relayed to the leader node. For write requests, LoadBalancer first checks the configuration to see whether we are using the quorum based writing scheme or not.

If we are not using the quorum based scheme, the writing requests are directly sent to the leader node. Otherwise, the writing requests will be sent to the leader only if there are more than one nodes alive (say 3 is the total number of nodes). We reject the writes to the client if there is no consensus existing among nodes.

An independent node hiring server thread is mainly responsible to listen to the join requests from the nodes. Once joined, a new dedicated node listening server thread is created for the node. On this server thread, we listen the heartbeats, write and query acknowledges from the nodes. The messages sent to the nodes are sent from the dedicated socket. The LoadBalancer assigns a unique positive id to each

joining node. If a node rejoins after dying, it will get a new id.

3.1.2 Node Design

A Node in our simulation system is a machine that stores and serves data. The data essentially is a key-value pair. This data is written as a log on the disk. More information about writes and log is described in the next section. Each node in our simulation can be an independent machine on the Internet. The node can be a leader or a replica. Whenever a node starts, it requires the port number on which it is going to run and the LoadBalancer IP address and port for initialization. Each node has a local log file on disk on which the writes are saved. More information about the design of log file is in the next section. A node can join, die and rejoin at any point of time in the system.

3.1.2.1 ID Assignments and Initialization

The first step for any joining node is to ping the LoadBalancer with the *nodeinitialization* message to register itself in the system. As described above, the LoadBalancer will send a reply, which contains a positive unique id assigned to the node and the information regarding the leader. This information includes the leader id, leader's IP address and port. This information is useful for the joining node to decide whether it is the leader (assigned id is same as leader id) and if not, then how to connect to the leader. The leader election algorithm is described in the next section. If the node is new i.e. it is not a rejoining node, then the log file will be initialized on the disk to save the writes.

3.1.2.2 Heartbeats to LoadBalancer

After initialization message, the node sets up a persistent connection with the LoadBalancer. It sends a heartbeat signal to the LoadBalancer at regular intervals. More information about the heartbeat algorithm is described in the next section. The node also waits for any message from the LoadBalancer. Messages can be of type write, query or leader re-election.

3.1.2.3 Node as Leader

If it is the leader, it will start a server and wait for incoming replica connections. It will also create a local data cache to store writes. The main task for the leader is to accept writes and server queries. More information about this is in the next section. If an existing node becomes a leader after re-election, it will create a local data cache from its log file to serve queries.

3.1.2.4 Node as Replica

If it is a replica, it will connect to the leader by using the information that it received from the LoadBalancer during initialization. The first message sent to the leader is *syncwithleader* message. Using this message, the replica gets all the missing writes and comes up to speed with the current system. More information about this is in the next section. On getting a leader re-election message, it starts a server if it is the new leader or it connects to the new leader if some other node has been chosen as the new leader.

3.1.3 Client design

Client process in this simulation system is designed as a command line interaction program to simulate the behaviors of real SolrCloud clients. Two important components of the client are command line console and LoadBalancer listening server. The system input stream is taken as the input for the commands. Mainly 4 types of commands are implemented right now including: help, write, query, and quit. LoadBalancer listener server thread is created mainly to deal with the communication with the LoadBalancer.

3.2 Implementation Details

3.2.1 Heartbeats

We use heartbeat techniques to keep track of the status of nodes. Basically, each node will have an independent thread that is sending heartbeat message to the LoadBalancer at a fixed rate. The LoadBalancer is maintaining a map data structure in memory that contains node ID and its status. Each node listener in the LoadBalancer will create a timer thread after initialization. This timer will be refreshed after receiving the heartbeat information. If the timer hasn't received heartbeat information for a long time, the timer will time out and change the status of the node in the status map. This part is trying to simulate some functionality of ZooKeeper in the original SolrCloud system.

3.2.2 Leader Election

As mentioned in previously, whenever a node joins the system, it is assigned a unique positive increasing

id by the LoadBalancer. The LoadBalancer keeps track of all the live nodes along with their ids. The ids are used for leader election. The node with the smallest id is the leader. When a node with the smallest id (i.e. the current leader) dies, the node with the smallest id among existing live nodes becomes the new leader. This algorithm works well as we always assign a new id to a node that is rejoining the system. For example, if there are 3 live nodes in the system and suppose we kill node A whose id is 0. So now, node B with id 1 is the leader. When node A rejoins (i.e. we restart node A), it will get id 3 (0-2 were already used). When node B dies, node C with id 2 will be the new leader. This leader election method is same as what SolrCloud currently uses. This algorithm works fine with quorum. When there is no quorum, it is possible that the replica can be ahead of the leader. Hence as a future work, we plan to implement a new leader election algorithm where on every new node join, the node with the highest timestamp in the log file will become the leader.

3.2.3 Write, Query, Sync and Log

In this simulation, our writes are key value pair objects. The reason to choose key value pair is because it easy to store and query data. Currently, the writes are of type strings but it can be any object par se. The client sends write/query to the LoadBalancer and the LoadBalancer forwards it to the leader. More description is given below.

3.2.3.1 Log Implementation

As mentioned previously, each node has a log file on disk to save writes. Each line in a log file is a write and it contains the key and value objects and a positive increasing logical timestamp.

3.2.3.2 Write Implementation

Whenever the leader gets a write from the LoadBalancer (*writeData*), it puts the data in local cache, writes it to log file and forwards it to the replicas. The leader doesn't wait for the replicas to reply and ACKs the write instantly. We assume that since the LoadBalancer checks for quorum and replica liveliness before sending a write to the leader, the writes will always succeed on the replicas. This assumption also leads to better performance, as waiting for replicas to ACK does not block the leader. Also, the leader doesn't do a local quorum check as the LoadBalancer does it before forwarding the write to it. In future, we plan to add this feature achieve more robust behavior. The leader copies the log entry of the new write into a new message and sends it to the replicas. The replicas will just put this received log entry in their log file. Hence, the timestamp in the log file will be from the leader and not the local timestamp of the replica. Only the leader maintains the timestamp and not the replicas. Whenever a node becomes a leader, it will update its own timestamp from its last log entry. This is because the leader always has the latest write.

3.2.3.3 Query Implementation

Query is essentially a key object. The purpose of the query is to get the value object for the given key. The LoadBalancer sends the query (*queryData*) to the leader only. This design decision was made to ensure that queries are not sent to recovering/syncing replicas, which in turn may lead to query failure. The leader sends the *queryAck* message to the LoadBalancer if it finds the Key Value pair in the local cache.

3.2.3.4 Sync Implementation

A replica initiates sync whenever it joins the system. The replica will send the last timestamp id in the local log file to leader (-1 if the log file has no writes yet). The leader will send the writes after the requested timestamp id to the replica. The replica will save the missing writes in its log file and hence come up to speed with the leader.

3.3 Tunable Control variables for the simulation system

According to our reproduction process, we have the chance to utilize different control variables out of the simulation system to control the result of final query success rates. In our design, the process we want to simulate is:

1. Three nodes are all healthy. Clients write a set of records into the system. (set of records is considered as w1)
2. Two replica nodes die after writing in the first step. After two nodes are dead, clients write another set of records into the system. (the set of records is considered as w2)
3. After the second set of writes is finished, one of the dead replicas comes alive and it starts

to sync data from the leader node. We kill the leader node after they have synced certain percentage of the data. (the set of synced data is considered as S)

As you can see from the above process, there are three variables to control. We could change any of them to adjust the simulation results.

3.4 Development Environment

We developed the whole system using Java programming language on Mac OS 10.10 with JDK 1.8. Both computers are using Intel Core i5 2.6Ghz CPU.

4. EVALUATION

We ran the query experiments on both the original SolrCloud simulation and improved SolrCloud simulation. The results have shown positive support for quorum based writing scheme. The original design, where write availability is preferred, causes serious consistency problem. We show the trade-off between the write availability and write consistency and argue that the proposed quorum based writing scheme is better.

For evaluation, we have designed our writes as a serial sequence of key value pairs. The write command in client will generate a write set. For example, if we say that we want to write one hundred records at beginning into the simulation system, the client will generate keys starting from 0 to 99. Values will be the same as keys for simplicity. The keys will keep increasing if we want to write more. The query traffic will be generated with 2 main parameters. One is the starting id of the key and another is the amount of queries we want to write. For example, if we want to generate 100 query starting from key id 100, then the query set will be 100, 101, …199.

If we set w1 = 100 (keys: 0~99), w2 = 100 (keys: 100~199), S = 24(keys: 100~123), we could have the result of query success rate in the figure 7. The R ratio in traffic is defined as the ratio of lost data over synced data inside a fixed size of query set. Intuitively, if R ratio is larger, it means there are more data lost due to the writing scheme. Figure 7 clearly shows that the query success rate is going to be always 1 if we are using quorum based writing scheme. If we are not using quorum based writing scheme, then under certain traffic, the query success rate is going to be very low, which will cause a lot of problems for the users due to these lost writes.

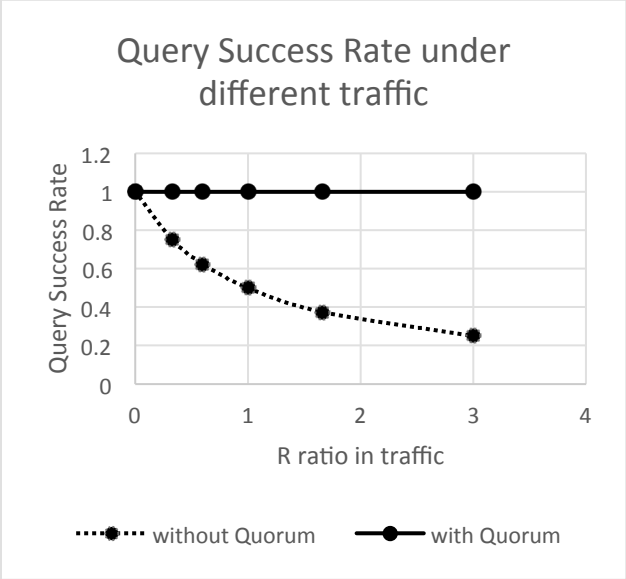

Figure 7

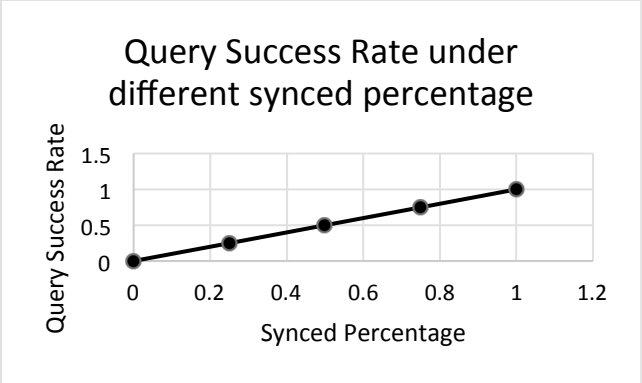

Figure 8

If we set w1 = 100(keys: 0~99), w2 = 100(keys: 100~199), Query set to be 100~200, and we vary the synced percentage, we will get the result of Figure 8. In Figure 8, we could basically see the trend of increasing query success rate with increasing synced percentage. This figure implies that if nodes that were dead for a long time may have lower query success rate since they have longer syncing process and the expected synced percentage for them may be relatively lower.

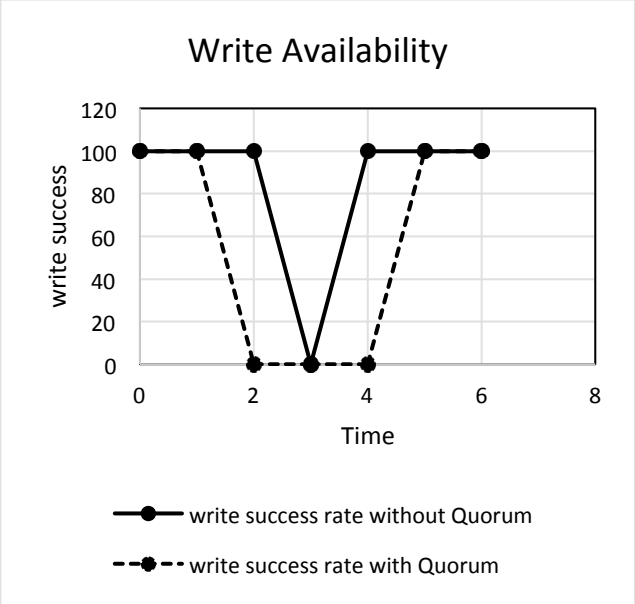

Figure 9

Even though quorum based writes has benefits on the consistency, there are still drawbacks in this scheme. The main concern is the reduced write availability. As shown in figure 9, if at time 0 there are 3 nodes alive, time 1 with 2 nodes alive, time 2 with 1 node alive, time 3 with all nodes dead, time 4 with 1 node alive, time 5 with 2 nodes alive, time 6 with 3 nodes alive. Then, the time period length of write without quorum will be longer than with quorum. However, we argue that during time 1 and time 2 when there is 1 node alive, the writes you gain from writing without quorum have serious writes loss problem that makes the consistency very bad. Thus we conclude that writing with quorum is worth the price of write availability in favor of write consistency.

5. RELATED WORK

As mentioned previously, this issue of SolrCloud is known in the community and they have provided a workaround in order to solve it. In the workaround, Solr supports the optional min_rf parameter on write requests that cause the server to return the achieved replication factor for a write request in the response. For the example scenario described above, if the client application included min_rf >= 1, then Solr would return rf=1 in the Solr response header because the request only succeeded on the leader. The write request will still be accepted as the min_rf parameter only tells Solr that the client application wishes to know what the achieved replication factor was for the write request. In other words, min_rf does not mean Solr will enforce a minimum replication factor. On the client side, if the achieved replication factor is less than the acceptable level, then the client application can take additional measures to handle the degraded state. For instance, a client application may want to keep a log of which update requests were sent while the state of the collection was degraded and then resend the updates once the problem has been resolved. In short, min_rf is an optional mechanism for a client application to be warned that an update request was accepted while the collection is in a degraded state. [3]

To design our simulation system, we take reference design hints from already existing large scale systems like Chubby[8], Apache Zookeeper[9] and Apache Solr [10]

6. SUMMARY AND CONCLUSION

Starting from a consistency related bug issue found in SolrCloud, we have reproduced the bug in the SolrCloud 5.0 system, simulated the bug in our simulation system, designed an improved solution to solve the bug, and evaluated the effectiveness of our improved solution against the older system. We have shown that having a quorum based write system will lead to better consistency and writes will not be lost. We have also created a flexible simulation system that could be used to explore more consistency and fault tolerance related issues inside the SolrCloud system and distributed systems in general.

We hope that the question we started out to address – whether consistency is important over availability – is successfully addressed and understood using our simulation. We hope that people who read this report will get more ideas and insights into tradeoff between consistency and availability in a distributed system.

7. REFERENCES

[1]Wikipedia http://en.wikipedia.org/wiki/Apache_Solr
[2] SolrCloud Wiki
https://cwiki.apache.org/confluence/display/solr/SolrCloud
[3] SOLR-5468 bug report
https://issues.apache.org/jira/browse/SOLR-5468
[4] Apache Solr homepage http://lucene.apache.org/solr/
[5] SolrCloud tutorial
https://cwiki.apache.org/confluence/display/solr/Getting+St

arted+with+SolrCloud
[6] SolrCloud tutorial http://heliosearch.org/solrcloud-assigning-nodes-machines/
[7] SolrCloud experiments https://drive.google.com/file/d/0Byq25hMG-bUxcnRUNVZKSWJtZ28/view?usp=sharing
[8] M. Burrows, *The Chubby Lock Service for Loosely-Coupled Distributed Systems*, in Proceedings of the Seventh Symposium on Operating System Design and Implementation, December, 2006
[9] ZooKeeper: Wait-free coordination for Internet-scale systems
[10] Solr in action book by Trey Grainger and Timothy Potter. ISBN-1617291021