NewSolrCloudDesign

New SolrCloud Design

This was a scratchpad for ideas and was not used as is - this page does not describe the SolrCloud design as it was implemented and this page is only around for historical reasons.

What is SolrCloud?

SolrCloud is an enhancement to the existing Solr to manage and operate Solr as a search service in a cloud.

Glossary

- **Cluster** : Cluster is a set of Solr nodes managed as a single unit. The entire cluster must have a single schema and solrconfig
- **Node** : A JVM instance running Solr
- **Partition** : A partition is a subset of the entire document collection. A partition is created in such a way that all its documents can be contained in a single index.
- **Shard** : A Partition needs to be stored in multiple nodes as specified by the replication factor. All these nodes collectively form a shard. A node may be a part of multiple shards
- **Leader** : Each Shard has one node identified as its leader. All the writes for documents belonging to a partition should be routed through the leader.
- **Replication Factor** : Minimum number of copies of a document maintained by the cluster
- **Transaction Log** : An append-only log of write operations maintained by each node
- **Partition version** : This is a counter maintained with the leader of each shard and incremented on each write operation and sent to the peers
- **Cluster Lock** : This is a global lock which must be acquired in order to change the range -> partition or the partition -> node mappings.

Guiding Principles

- Any operation can be invoked on any node in the cluster.
- No non-recoverable single point of failures
- Cluster should be elastic
- Writes must never be lost i.e. durability is guaranteed
- Order of writes should be preserved
- If two clients send document "A" to two different replicas at the same time, one should consistently "win" on all replicas.
- Cluster configuration should be managed centrally and can be updated through any node in the cluster. No per node configuration other than local values such as the port, index/logs storage locations should be required
- Automatic failover of reads
- Automatic failover of writes
- Automatically honour the replication factor in the event of a node failure

Zookeeper

A ZooKeeper cluster is used as:

- The central configuration store for the cluster
- A co-ordinator for operations requiring distributed synchronization
- The system-of-record for cluster topology

Partitioning

The cluster is configured with a fixed max_hash_value (which is set to a fairly large value, say 1000) ‘N’. Each document’s hash is calculated as:

```
hash = hash_function(doc.getKey()) % N
```

Ranges of hash values are assigned to partitions and stored in Zookeeper. For example we may have a range to partition mapping as follows

```
range  : partition
------  ----------
0 - 99 : 1
100-199: 2
200-299: 3
```

The hash is added as an indexed field in the doc and it is immutable. This may also be used during an index split

The hash function is pluggable. It can accept a document and return a consistent & positive integer hash value. The system provides a default hash function which uses the content of a configured, required & immutable field (default is unique_key field) to calculate hash values.

Using full hash range

Alternatively, there need not be any max_hash_value - the full 32 bits of the hash can be used since each shard will have a range of hash values anyway. Avoiding a configurable max_hash_value makes things easier on clients wanting related hash values next to each other. For example, in an email search application, one could construct a hashcode as follows:

```
(hash(user_id)<<24) | (hash(message_id)>>>8)
```

By deriving the top 8 bits of the hashcode from the user_id, it guarantees that any users emails are in the same 256th portion of the cluster. At search time, this information can be used to only query that portion of the cluster.

We probably also want to view the hash space as a ring (as is done with consistent hashing) in order to express ranges that wrap (cross from the maximum value to the minimum value).

shard naming

When partitioning is by hash code, rather than maintaining a separate mapping from shard to hash range, the shard name could actually be the hash range (i.e. shard "1-1000" would contain docs with a hashcode between 1 and 1000).

The current convention for solr-core naming is that it match the collection name (assuming a single core in a solr server for the collection). We still need a good naming scheme for when there are multiple cores for the same collection.

Shard Assignment

The node -> partition mapping can only be changed by a node which has acquired the Cluster Lock in ZooKeeper. So when a node comes up, it first attempts to acquire the cluster lock, waits for it to be acquired and then identifies the partition to which it can subscribe to.

Node to a shard assignment

The node which is trying to find a new node should acquire the cluster lock first. First of all the leader is identified for the shard. Out of the all the available nodes, the node with the least number of shards is selected. If there is a tie, the node which is a leader to the least number of shard is chosen. If there is a tie, a random node is chosen.

Boot Strapping

Cluster Startup

A node is started pointing to a Zookeeper host and port. The first node in the cluster may be started with cluster configuration properties and the schema /config files for the cluster. The first node would upload the configuration into zookeeper and bootstrap the cluster. The cluster is deemed to be in the “bootstrap” state. In this state, the node -> partition mapping is not computed and the cluster does not accept any read/write requests except for clusteradmin commands.

After the initial set of nodes in the cluster have started up, a clusteradmin command (TBD description) is issued by the administrator. This command accepts an integer “partitions” parameter and it performs the following steps:

1. Acquire the Cluster Lock
2. Allocate the “partitions” number of partitions
3. Acquires nodes for each partition
4. Updates the node -> partition mapping in ZooKeeper
5. Release the Cluster Lock
6. Informs all nodes to force update their own node -> partition mapping from ZooKeeper

Node Startup

The node upon startup, checks ZooKeeper if it is a part of existing shard(s). If ZooKeeper has no record of the node or if the node is not part of any shards, it follows the steps in the New Node section else it follows the steps in the Node Restart section.

New Node

A new node is one which has never been part of the cluster and is newly added to increase the capacity of the cluster.

If the “auto_add_new_nodes” cluster property is false, the new nodes register themselves in ZooKeeper as “idle” and wait until another node asks them to participate. Otherwise, they proceed as follows:

1. The Cluster Lock is acquired
2. A suitable source (node, partition) tuple is chosen:

    a. The list of available partitions are scanned to find partitions which has less then “replication_factor” number of nodes. In case of tie, the partition with the least number of nodes is selected. In case of another tie, a random partition is chosen.
    b. If all partitions have enough replicas, the nodes are scanned to find one which has most number of partitions. In case of tie, of all the partitions in such nodes, the one which has the most number of documents is chosen. In case of tie, a random partition on a random node is chosen.
    c. If moving the chosen (node, partition) tuple to the current node will decrease the maximum number of partition:node ratio of the cluster, the chosen tuple is returned.Otherwise, no (node, partition) is chosen and the algorithm terminates
    d. The node -> partition mapping is updated in ZooKeeper
3. The node status in ZooKeeper is updated to “recovery” state
4. The Cluster Lock is released
5. A “recovery” is initiated against the leader of the chosen partition
6. After the recovery is complete, the Cluster Lock is acquired again
7. The source (node, partition) is removed from the node -> partition map in ZooKeeper
8. The Cluster Lock is released
9. The source node is instructed to force refresh the node -> partition map from ZooKeeper
10. Goto step #1

Node Restart

A node restart can mean one of the following things:

- The JVM crashed and was manually or automatically restarted
- The node was in a temporary network partition and either could not reach ZooKeeper (and was supposed to be dead) or could not receive updates from the leader for a period of time. A node restart ine node failure.
- Lifecycle of a Write Operation this scenario signifies the removal of the network partition.
- A hardware failure or maintenance window caused the removal of the node from the cluster and the node has been started again to rejoin the cluster.

The node reads the list of partitions for which it is a member and for each partition, starts a recovery process from each partition’s leader respectively. Then, the node follows the steps in the New Node section without checking for the auto_add_new_nodes property. This ensures that the cluster recovers from the imbalance created by th

Writes are performed by clients using the standard Solr update formats. A write operation can be sent to any node in the cluster. The node uses the hash_function , and the Range-Partition mapping to identify the partition where the doc belongs to. A zookeeper lookup is performed to identify the leader of the shard and the operation is forwarded there. A SolrJ enhancement may enable it to send the write directly to the leader

The leader assigns the operation a Partition Version and writes the operation to its transaction log and forwards the document + version + hash to other nodes belonging to the shard. The nodes write the document + hash to the index and record the operation in the transaction log. The leader responds with an ‘OK’ if at least min_writes number of nodes respond with ‘OK’. The min_writes in the cluster properties can be overridden by specifying it in the write request.

The cloud mode would not offer any explicit commit/rollback operations. The commits are managed by auto-commits at intervals (commit_within) by the leader and triggers a commit on all members on the shard. The latest version available to a node is recorded with the commit point.

Transaction Log

- A transaction log records all operations performed on an Index between two commits
- Each commit starts a new transaction log because a commit guarantees durability of operations performed before it
- The sync can be tunable e.g. flush vs fsync by default can protect against JVM crashes but not against power failure and can be much faster

Recovery

A recovery can be triggered during:

- Bootstrap
- Partition splits
- Cluster re-balancing

The node starts by setting its status as ‘recovering’. During this phase, the node will not receive any read requests but it will receive all new write requests which shall be written to a separate transaction log. The node looks up the version of index it has and queries the ‘leader’ for the latest version of the partition. The leader responds with the set of operations to be performed before the node can be in sync with the rest of the nodes in the shard.

This may involve copying the index first and replaying the transaction log depending on where the node is w.r.t the state of the art. If an index copy is required, the index files are replicated first to the local index and then the transaction logs are replayed. The replay of transaction log is nothing but a stream of regular write requests. During this time, the node may have accumulated new writes, which should then be played back on the index. The moment the node catches up with the latest commit point, it marks itself as “ready”. At this point, read requests can be handled by the node.

Handling Node Failures

There may be temporary network partitions between some nodes or between a node and ZooKeeper. The cluster should wait for some time before re-balancing data.

Leader failure

If node fails and if it is a leader of any of the shards, the other members will initiate a leader election process. Writes to this partition are not accepted until the new leader is elected. Then it follows the steps in non-leader failure

Non-Leader failure

The leader would wait for the min_reaction_time before identifying a new node to be a part of the shard. The leader acquires the Cluster Lock and uses the node-shard assignment algorithm to identify a node as the new member of the shard. The node -> partition mapping is updated in ZooKeeper and the cluster lock is released. The new node is then instructed to force reload the node -> partition mapping from ZooKeeper.

Splitting partitions

A partition can be split either by an explicit cluster admin command or automatically by splitting strategies provided by Solr. An explicit split command may give specify target partition(s) for split.

Assume the partition ‘X’ with hash range 100 - 199 is identified to be split into X (100 - 149) and a new partition Y (150 - 199). The leader of X records the split action in ZooKeeper with the new desired range values of X as well as Y. No nodes are notified of this split action or the existence of the new partition.

1. The leader of X, acquires the Cluster Lock and identifies nodes which can be assigned to partition Y (algorithm TBD) and informs them of the new partition and updates the partition -> node mapping. The leader of X waits for the nodes to respond and once it determines that the new partition is ready to accept commands, it proceeds as follows:
2. The leader of X suspends all commits until the split is complete.
3. The leader of X opens an IndexReader on the latest commit point (say version V) and instructs its peers to do the same.
4. The leader of X starts streaming the transaction log after version V for the hash range 150 - 199 to the leader of Y.
5. The leader of Y records the requests sent in #2 in its transaction log only i.e. it is not played on the index.
6. The leader of X initiates an index split on the IndexReader opened in step #2.
7. The index created in #5 is sent to the leader of Y and is installed.
8. The leader of Y instructs its peers to start recovery process. At the same time, it starts playing its transaction log on the index.
   a. Once all peers of partition Y have reached at least version V:
   b. The leader of Y asks the leader of X to prepare a FilteredIndexReader on top of the reader created in step #2 which will have documents belonging to hash range 100 - 149 only.
   c. Once the leader of X acknowledges the completion of request in #8a, the leader of Y acquires the Cluster Lock and modifies the range -> partition mapping to start receiving regular search/write requests from the whole cluster.
   d. The leader of Y asks leader of X to start using the FilteredIndexReader created in #8a for search requests.
   e. The leader of Y asks leader of X to force refresh the range -> partition mapping from ZooKeeper. At this point, it is guaranteed that the transaction log streaming which started in #3 will be stopped.
9. The leader of X will delete all documents with hash values not belonging to its partitions, commits and re-opens the searcher on the latest commit point.
10. At this point, the split is considered complete and leader of X resumes commits according to the commit_within parameters.

Notes:

- The partition being split does not honor commit_within parameter until the split operation completes
- Any distributed search operation performed starting at the time of #8b and till the end of #8c can return inconsistent results i.e. the number of search results may be wrong.

Cluster Re-balancing

The cluster can be rebalanced by an explicit cluster admin command.

TBD

Monitoring

TBD

Configuration

solr_cluster.properties

This are the set of properties which are outside of the regular Solr configuration and is applicable across all nodes in the cluster:

- **replication_factor** : The number of replicas of a doc maintained by the cluster
- **min_writes** : Minimum no:of successful writes before the write operation is signaled as successful . This may me overridden on a per write basis
- **commit_within** : This is the max time within which write operation is visible in a search
- **hash_function** : The implementation which computes the hash of a given doc
- **max_hash_value** : The maximum value that a hash_function can output. Theoretically, this is also the maximum number of partitions the cluster can ever have
- **min_reaction_time** : The time before any reallocation/splitting is done after a node comes up or goes down (in secs)
- **min_replica_for_reaction** : If the number of replica nodes go below this threshold the splitting is triggered even if the min_reaction_time is not met

- **auto_add_new_nodes** : A Boolean flag. If true, new nodes are automatically used as read replicas to existing partitions, otherwise, new nodes sit idle until the cluster needs them.

Cluster Admin Commands

All cluster admin commands run on all nodes at a given path (say /cluster_admin). All nodes are capable of accepting the same commands and the behavior would be same. These are the public commands which a user can use to manage a cluster:

- **init_cluster** : (params : partition) This command is issued after the initial set of nodes are started. Till this command is issued, the cluster would not accept any read/write commands
- **split_partition** : (params : partition$_{\text{optional}}$). The partition is split into two halves. If the partition parameter is not supplied, the partition with the largest number of documents is identified as the candidate.
- **add_idle_nodes** : This can be used if auto_add_new_nodes=false. This command triggers the addition of all 'idle' nodes to the cluster.
- **move_partition** : (params : partition, from, to). Move the given partition from a given node from to another node
- **command_status** :(params : completion_id$_{\text{optional}}$) . All the above commands are asynchronous and returns with a completion_id . This command can be used to know the status of a particular running command or all the current running commands
- **status** : (params : partition$_{\text{optional}}$) Shows the list of partitions and for each partition, the following info is provided
  - leader node
  - nodes list
  - doc count
  - average reads/sec
  - average writes/sec
  - average time/read
  - average time/write

Migrating from Solr to SolrCloud

A few features may be redundant or not supported when we move to cloud such as. We should continue to support the non cloud version which supports all the existing features

- Replication. This feature is not required anymore
- CoreAdmin commands. Explicit manipulation of cores will not be allowed. Though cores may exist internally and they meay be managed implicitly
- Multiple schema support ? Should we just remove it from ver 1.0 for simplicity?
- solr.xml . Is there a need at all for this in the cloud mode?

Alternative to a Cluster Lock

It may be simpler to have a coordinator node (we avoid the term master since that is associated with traditional index replication) that is established via leader election. Following a pattern of treating the zookeeper state as the "truth" and having nodes react to changes in that state allow for more future flexibility (such as allowing an external management tool directly change the zookeeper state to control the cluster). Having a coordinator (as opposed to grabbing a lock every time) can be more scalable too. A hybrid model where a cluster lock is used only in certain circumstances can also make sense.

Single Node Simplest Use Case

We should be able to easily start up a single node and start indexing documents. At a later point in time, we should be able to start up a second node and have it join the cluster.

1. start up a single node, upload it's configuration (the first time) to zookeeper, and create+assign the node to shard1.
   - in the absence of other information when the config is created, a single shard system is assumed
2. index some documents
3. start up another node and pass it a parameter that says "if you are not already assigned, assign yourself to any shard that has the lowest number of replicas and start recovery process"
   - avoid replicating a shard on the same host if possible
   - after this point, one should be able to kill the node and start it up again and have it resume the same role (since it should see itself in zookeeper)