OceanComponents

Ocean Components

Replication

There are two ways to do replication and I have been leaning towards a non master slave architecture. I looked at the Paxos at War algorithm for master slave failover. The problem is, I did not understand it, and found it too complex to implement. I tried other more simple ways of implementing master slave failover and it still had major problems. This led me to look for another solution.

In a master slave architecture the update is submitted to the master first and then to the slaves which is not performed in parallel. Configuration changes such as turning semi-sync on or off would require restarting all processes in the system.

The ideal architecture would allow any node to act as the proxy for the other nodes. This would make every node a master. The transaction would be submitted to all nodes and the client would determine on how many nodes the transaction needs to be successful. In the event a transaction fails on a node, nodes are always executing a polling operation to all other nodes that rectifies transactions. This does not need to run too often, however if a node is just coming back online, it needs to reject queries until it is up to date. The node may obtain the latest transactions from any other node.

When a new node comes online, it will need to simply download the entire set of Lucene index files from another node. The transaction log will not always have all transactions contained in it's indexes because there is no need. It is faster for a new node to download the indexes first, then obtain the transactions it does not have from another node's transaction log.

Because the Ocean system stores the entire document on an update and there is no support for update of specific fields like SQL, it is much easier to rectify transactions between nodes. Meaning that deletes and updates of objects are less likely to clobber each other during the rectification process.

Crowding

GBase mentions a feature that is perhaps somewhat interesting and this is crowding. It is similar to what in Solr is referred to as Field Collapsing however the implementation for Ocean could be a little bit easier and more efficient. Solr's Field Collapse code performs a sort on the results first and then seems to perform another query. GBase allows only 2 fields to be crowded making the implementation seem a bit easier. Also it would seem to be easier to simply obtain more results than are needed and crowd a field similar to how the NutchBean uses a dedupField. I have tried to implement this feature into Ocean and have been unable to get it quite right.

Facets

I wanted facets to work in realtime because it seemed like a challenging thing to do. The way I came up with to do this is a copy on read versioned LRU cache embodied in the BitSetLRUMap. The bit sets for faceting need to be cached. The problem is, each transaction may perform deletes and the bit set needs to reflect this to be accurate during an intersection call. Rather than perform deletes on all of the cached bit sets for each transaction (which would consume a large amount of RAM and create a lot of garbage) a copy on read is used (deletes are applied only when the value is read). The bit set cache stores the deletes docs of each snapshot/transaction. If a given bit set is required and the value is out of date then the deletes are applied to a new one. Each value in the cache stores multiple versions of a bit set. Periodically as snapshots are released by the system the older bit sets are also released. This system is efficient because only the used bit sets are brought up to date with the latest snapshot.

Facet caching needs to be handled per segment and merged during the search results merging.

Distributed Search

Distributed search with Ocean will use the http://issues.apache.org/jira/browse/LUCENE-1336 patch. It provides RMI functionality over the Hadoop IPC protocol. Using Hadoop IPC as a transport has advantages over using Sun's RMI because it is simpler and uses NIO (non blocking sockets). In large systems using NIO reduces thread usage and allows the overall system to scale better. LUCENE-1336 allows classes to be dynamically loaded by the server from the client on a per client basis to avoid problems with classloaders and class versions. Using a remote method invocation system for me is much faster to implement functionality than when using Solr and implementing XML interfaces and clients or using namedlists. I prefer writing distributed code using Java objects because they are what I am more comfortable with. Also I worked on Jini and Sun and one might say it is in the blood. The idea to create a better technique for classloading comes from my experiences and the failures of trying to implement Jini systems. Search is a fairly straightforward non-changing problem and so the dynamic classloading is only required by the server from the client. By having a reduced scope problem the solution was much easier to generate compared to working with Jini which attempted to solve all potential problems even if they most likely do not exist.

In the future it is possible to write a servlet wrapper around the Ocean Java client and expose the Ocean functionality as XML possibly conforming to Open Search and/or GData.

An object is localized to a cell. Meaning after it is created it usually remains in the same cell over it's lifespan. This is to insure the searches remain consistent. The object contains the cellid of where it originated from. This allows subsequent updates to the object (in Lucene a deleteDocument and then addDocument are called) to occur in the correct cell.

Name Service

Name services can become quite complex. For example it may be possible in the future to use Zookeeper which is a lock based service. However even by Zookeeper's own admission these types of lock services are hard to implement and use correctly. I think for Ocean it should be good enough in the first release to have an open source SQL database that stores the nodes and the cells the nodes belong to. Because there is no master there is no need for a locking service. The columns in the node table would be id, location, status (online/offline), cellid, datecreated, datemodified. The cell table would simply be id, status, datecreated, datemodified. Redundant name services may be created by replicating these 2 tables. I am also pondering an errors table where clients may report outages of a node. If there are enough outages of a particular node the name service marks the node as offline. Clients will be able to listen for events on a name service related to cells, mainly the node status column. This way if a node that was online goes offline, the client will know about it and not send requests to it any longer.

Location Based Services

LocalLucene provides the functionality for location based queries. It is possible to optimize how LocalLucene works and I had code that implemented Local Lucene functionality directly into Ocean that I may put back in at some point. The optimization works by implementing a subclass of ScoreDoc that has a Distance object as a member variable. This removes the need for a map of the document to the distance value from the DistanceFilter. I would like to see D istanceFilter use the new Lucene Filter code that returns DocIdSet.

Tag Index

The tag index patch is located at LUCENE-1292. I had seen people mention using a ParallelReader to have an index that is static and an index that is dynamic appear as one index. The challenge with this type of system is to get the doc numbers to stay aligned. Google seems to have a realtime tag index system. I figured there must be some way using the Lucene architecture to achieve the same thing. The method I came up with is to divide the postings list into blocks. Each block contains a set number of documents, the blocks are not divided by actual byte size but by document number. The blocks are unified using a TagMultiTermDocs class. When a block is changed it is written to RAM. Once the RAM usage hits a certain size, the disk and memory postings are merged to disk. There needs to be coordination between this process and the merging of the segments. Each Tag Index is associated with a segment. In Ocean the merging of segments is performed by the Ocean code and not IndexWriter so the coordination does not involve hooking into Index Writer. Currently there needs to be a way to obtain the doc id from an addDocument call from IndexWriter which needs a patch still.