OceanRealtimeSearch

Introduction

Ocean enables realtime search written in Java using Lucene. It is currently in patch phase at LUCENE-1313. Ocean offers a way for Lucene based applications to take advantage of realtime search. Realtime search makes search systems more like a database. Realtime search offers similar functionality to that of a relational database such as atomicity and transactional updates. Like an RDBMS, where after an insert one can immediately select what was just added without incurring any kind of extra cost, updated documents are also immediately searchable.

There is a good article written by Adam Bosworth. I think many of his points are quite valid. It is worth mentioning the main points of the article here as they also define the positive attributes of the Ocean open source search system.

- It is worth making things simple enough that one can harness Moore's law in parallel
- It is acceptable to be stale much of the time
- Be as loosely coupled as possible
- SQL databases have problems harnessing Moore's law in parallel (whereas distributed search systems do not)
- SQL databases have problems allowing users to evolve a schema over time. Lucene offers the ability to define any fields at any time including the ability to define multiple values for a field something SQL databases do not offer.
- SQL queries are more complex than search queries which are more suitable for end users

Background

From a discussion with Karl Wettin:

I was an early user of Solr when GData came out. They were similar in that they were both search exposed as XML. GData however offered realtime search and Solr offered batch processing. I worked for a social networking company that wanted the updates available as fast as possible. It was hard to achieve anything below a couple of minutes as the queries the company wanted used a sort. In Lucene a sort loads the field cache into RAM which on a large index is expensive. There are ways to solve this but they were not available. In any case I wanted to figure out a way to allow updates to be searchable in a minimal amount of time as possible while also offering functionality like SOLR of replication and facets. The one thing GData had over Solr was realtime updates or the ability to add, delete, or update a document and be able to see the update in search results immediately. With Solr the company had decided on a 10 minute interval of updating the index with delta updates from an Oracle database. I wanted to see if it was possible with Lucene to create an approximation of what GData does. The result is Ocean.

The use case it was designed for is websites with dynamic data, some of which are social networking, photo sites, discussions boards, blogs, wikis, and such. More broadly it is possible to use Ocean with any application that requires the database like feature of immediate updates. Probably the best example of this is all of Google's web applications, outside of web search, uses a GData interface. Meaning the primary datastore is not mysql or some equivalent, it is a proprietary search based database. The best example of this is Gmail. If I receive an email through Gmail I can also search on it immediately, there is no 10 minute delay. Also in Gmail I can change labels, a common example being changing unread emails to read in bulk. Presumably Gmail is not reindexing the entire email for each label change.

Most highly trafficked web applications do not use the relational facilities like joins because they are too expensive. Lucene does not offer joins so this is fine. The only area Lucene is currently weak in is range queries. Mysql uses a btree index whereas Lucene uses the time consuming TermEnum and Term Docs combination. This is an area Tag Index addresses.

The way Ocean is designed there should be no limitations to using it compared to using Lucene IndexWriter. It offers the same functionality. If one does not want to use the transaction log Ocean offers because one simply wants to index 1 million documents at once, Ocean offers what is a called a LargeBatc . It is a way to perform a large number of updates taking advantage of the new IndexWriter speedup, combined with transactional semantics.

What I Learned

Merging is expensive and detrimental to realtime search. The more merging that occurs during the update call, the longer it takes for the update to become available. Using IndexWriter.addDocument, committing and then calling IndexReader.reopen takes time because a merge must occur during the commit call, which would be called after each transaction. I learned that I needed to design a system that would not perform merging in the foreground during the update call, and have the merging performed in a background thread. Karl Wettin had created InstantiatedIndex and it took some time to figure out that it was the right object to use to create an in memory index of document(s) that would be immediately searchable. The issue of losing data is solved by the standard method Mysql uses which is a binary transaction log of the serialized documents and deletes.

Lucene uses a snapshot system that is embodied in the IndexReader class. Each IndexReader is a snapshot of the index with associated files. Ocean uses an IndexReader per snapshot however the IndexReaders are created more often. This means the IndexReaders are also disposed of much more quickly than in a system like SOLR. A lot of design work went into creating a system that would allow the IndexReaders to be created and then to remove them when they are no longer required. A referencing system was created for each snapshot where Java code may lock a snapshot, do work and unlock it. Only a set number of snapshots need to be available at a given time and the older unlocked snapshots are removed. Deletes occur in ram directly to the bitvector of the IndexReader with no flush to disk. This is because it was found to be prohibitively expensive to flush a new deletes file to disk on each update transaction. The file would then need to be cleaned only a few transactions later. Because there is a transaction log there is no need to write the deletes to disk twice and it is much faster to append to a file than create a new one and delete it possibly seconds later.

How it Works

Ocean writes updates to a transaction log and an in memory index. A transaction consists of document adds and deletes. If a transaction consists of (default: 100) or less documents, the documents are serialized to the transaction log. If greater than (default: 100), the documents are encoded into a Lucene segment by using an IndexWriter to write the documents to a RAMDirectory that is serialized to the transaction log. The latter reduces redundant analyzing if the transaction log is being replicated.

The in memory index is actually a series of indexes that are periodically merged in memory. When documents are first added, they are placed into a WriteableMemoryIndex that uses the Lucene contrib project InstantiatedIndex. InstantiatedIndex provides an in memory index where all of the objects are stored as is which makes for fast insert times because there is no serialization to bytes as with a RAMDirectory. Once the InstantiatedIndex reaches a predefined limit, it is converted into a RamIndex. The RamIndex uses a RAMDirectory and is an intermediary step before the in memory index is written to disk. RamIndexes are periodically merged in the background as well. The DiskIndex utilizes FSDirectory.

Ocean uses a different than usual process for writing indexes to disk. Instead of merging on disk, meaning reading from indexes on disk and writing to the new index at the same time, the merge process occurs in RAM. This happens with the RamIndex where it is in RAM and simply written to disk. When multiple DiskIndexes are merged, the new index is first created in RAM using RAMDirectory and then copied to disk. The reason for creating the index first in RAM is to save on rapid hard drive head movement. Usually DiskIndexes are partially in the system file cache. The normal merging process therefore is fast for reads and slow for the incremental write process. Hard drives are optimized for large sequential writes which is the described mechanism Ocean performs by first creating the index in RAM. The large segment which is typically 64MB in size is written all at once which should take 5-10 seconds.

Each transaction internally is recognized as a snapshot. A snapshot (org.apache.lucene.ocean.Snapshot) consists of a series of IndexSnapshots (org.apache.lucene.ocean.Index.IndexSnapshot). The parent class of DiskIndex and RamIndex is DirectoryIndex. DirectoryIndex uses IndexReader.clone http://issues.apache.org/jira/browse/LUCENE-1314 in the creation of an IndexSnapshot. IndexReader.clone creates a copy of an IndexReader that can be modified without altering the original IndexReader like IndexReader.reopen does. DirectoryIndexSnapshots never have documents added to them as they are single segment optimized indexes. DirectoryIndexSnapshots are only deleted from. Each transaction with deletes does not result in an IndexReader.flush call because this process is expensive. Instead, because the transaction is already stored on disk in the transaction log, the deletes occur only to the SegmentReader.deletedDocs.

Facets and filters need to be cached per Index. Each Index is really the same as a Lucene segment. However due to the way Lucene is designed, if one is merging outside of IndexWriter then each segment needs to be in it's own physical directory. This creates some extra files such as the segmentinfos file. Ocean manages deleting the old index directories when they are no longer necessary.

Transaction Log

Each transaction is recorded in the transaction log which is a series of files with the file name format log00000001.bin. The suffix number and a new log file is created when the current log file reaches a predefined size limit. The class org.apache.lucene.ocean.log.LogFileManager is responsible for this process.

The transaction record consists of three separate parts, the header, document bytes, and other bytes. The other bytes can store anything other than the documents, usually the deletes serialized. Each part has a CRC32 check which insures integrity of data. The transaction log can become corrupted if the process is stopped in the middle of a write. There is a CRC32 check with each part because they are loaded separately at different times. For example during the recovery process on the Ocean server startup the documents are the first to be loaded and in memory indexes are created. Then the deletes from the transactions are executed. Then the indexes are optimized to remove the deleted documents. The process described is much faster than performing each transaction incrementally during recovery. It is important to note that internally each delete, especially the delete by query is saved as the actual document ids that were deleted when the transaction was committed. If the system simply re-executed the delete by query, then the transaction would create inconsistent results.

Snapshot Log

The snapshot log is a set of rolling log files that contain the snapshot information. Each transaction generates a new snapshot entry in the current snapshot log file. The snapshot element contains the index elements. The log is in XML for human viewing which is useful for debugging.

Example:

```
<snapshot id="29.02" numDocs="10" maxDoc="25" deletedDocs="15">
<index snapshotid="974" id="787" segmentGeneration="401" type="disk" maxDoc="466" numDocs="442" deletedDoc="95" minDocumentId="117" maxDocumentId="483" minSnapshotId="693" maxSnapshotId="116" deleteFlushId="876" lastAppliedId="780" /></snapshot>
```

Name	Value
snapshotid	The id of the snapshot
id	The index id
segmentGeneration	Segment generation of the index as reported by IndexReader.
type	The type of the index
maxDoc	The max doc value of the index
numDocs	Number of documents in the index
deletedDoc	Number of deleted documents in the index
minDocumentId	The minimum document id in the index
maxDocumentId	The maximum document id in the index
minSnapshotId	The minimum snapshot id in the index
maxSnapshotId	The maximum snapshot id in the index

deleteFlushId	Snapshot id the last time the deleted docs were flushed to disk
lastAppliedId	The last snapshot id that affected this index

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.

Storing the Data

SOLR uses a schema. I chose not to use a schema because the realtime index should be able to change at any time. Instead the raw Lucene field classes such as Store, TermVector, and Indexed are exposed in the OceanObject class. An analyzer is defined on a per field per OceanObject basis. The process of serializing analyzers is not slow and is not bulky over the network as serialization performs referencing of redundant objects in the data stream. GData allows the user to store multiple types for a single field. For example, a field named battingaverage may contain fields of type double and text. I am really not sure how Google handles this underneath. I decided to use Solr's NumberUtils class that encodes numbers into sortable strings. This allows range queries and other enumerations of the field to return the values in their true order rather than string order. One method I came up with to handle potentially different types in a field is to prepend a letter signifying the type of the value for untokenized fields. For a string the value would be "s0.323" and for a long "l845445". This way when sorting or enumerating over the values they stay disparate and can be modified to be their true value upon return of the call. Perhaps there is a better method.

Since writing the above I came up with an alternative mechanism to handle any number in an Field.Index.UN_TOKENIZED field. If the field string can be parsed into a double, then the number is encoded using SOLR's NumberUtils into an encoded double string. There may be edge cases I am unaware of that make this system not work, but for right now it looks like it will work. In order to properly process a query, terms with fields in the query that are Field.Index.UN_TOKENIZED will need to be checked for having a number by attempted parsing. If the value can be parsed into a double then it is encoded into a double string and replaced in the term. A similar process will be used for date strings which will conform to the ISO 8601 standard. The user may also explicitly define the type of a number by naming the field such as "battingaverage_double" where the type is defined by an underscore and then the type name. This is similar to Solr's dynamic fields construction.

The following are default document fields. Some are currently placed by TransactionSystem, some by OceanDatabase. It is probably best for most applications to use OceanDatabase so all of them are mentioned here.

Name	Value
_id	Unique type long object id persistent across updates to an object assigned by Ocean.

_documentid	Unique type long document id to uniquely identify the document. This is useful when performing deletes and the exact documents deleted need to be saved in the transaction.
_version	Type long version of an object.
_datecreated	Date the object was created.
_datemodified	Date the object was last modified.

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.

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 DistanceFilter use the new Lucene Filter code that returns DocIdSet.

Configuration Options

Name	Description
serverNumber	The server number differentiates servers in a cell and is encoded into the uniquely generated ids on a server as the first 2 digits of the id.
memoryIndexMaxDocs	The maximum number of documents the InstantiatedIndex holds. Internally known as the WriteableMemoryIndex.
maybeMergeDocChanges	Number of changes to the overall index before the system checks to see if any indexes need to be merged. Executing this on every transaction would be a waste.

maxRamIndexesSize	Size in bytes all ram indexes after which they are written as a single optimized index to disk.
maxSnapshots	The maximum number of snapshots the system keeps around. The system will only remove the snapshot if it is unlocked.
mergeDiskDeletedPercent	Disk indexes that have too many deleted documents need to be merged to remove the deleted documents. This is the percentage of deleted documents a DiskIndex needs to have in order to be considered for merging.
snapshotExpiration	Duration in milliseconds after which a snapshot is considered for removal.
deletesFlushThresholdPercent	The percentage of deleted docs after which the deleted docs file is written for an index/segment.
maybeMergesTimerInterval	The maybe merges call is sometimes started in the background after a transaction, this milliseconds value is for running it in general in the background according to a timer.
logFileDeleteTimerInterval	Timer value in milliseconds for checking on deleting old log files.
diskIndexRAMDirectoryBufferSize	This is the buffer size in bytes to be used for the RAMDirectory when multiple DiskIndexes are merged. Ocean merges the DiskIndexes into a RAMDirectory first, then flushes the RAMDirectory to disk. This is to perform a large sequential write which is much faster than an incremental write process which usually used in Lucene merges.

To Do

- Finish and write test cases for OceanDatabase
- Facets
- Filter caching
- Distributed updates
- Name service
- Write the node asynchronous conflict resolution code and test cases
- Test case for LargeBatch