

RealtimeSearch

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 called a [LargeBatch](#). 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

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.

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

- Test case for [LargeBatch](#)

The previous sections referencing Ocean is now listed at [OceanComponents](#).