Technical Report of Index-manager

Yingfeng Zhang

December 15, 2008

Contents

1 Introduction

The design aim of IndexManager is to provide a high performance indexing framework that can work well together with other components of SF1 v5.0, including DocumentManager, ConfigurationManager, and LAManager. This framework should support the index's incremental updating, index compression, index's deletion, multi-collection management, query ability during index construction (on-line index), range query using a certain property, and can work distributively.

2 Design Issue

2.1 Index Construction

The Dynamic Balancing Tree algorithm [1] is used for index construction, which provides an index merging algorithm that supports on-line index maintenance. During past years, great advances have been achieved to build an off-line index, which could not provide query service during the process of index construction. However, with the boom of the web pages' count number, to provide search ability during indexes construction has been more and more urgent, therefore how to maintain on-line index is the current research hotspot on indexing problems. There are three kinds of index construction method: In-place, Re-build, and Re-merge[3]. For In-place update strategy, documents are accumulated in main memory. Then, once main memory is exhausted, the existing on-disk index is combined with the in-memory index by appending each posting list from the in-memory index to the corresponding list in the on-disk index; The Re-build algorithm constructs a new index from the current entire collection; For the Re-merge update strategy, once main memory is exhausted, a merge event is triggered, and the entire on-disk index is read sequentially and merged with the in-memory index, then written to a new location to form a new on-disk index that immediately replaces the old one. According to the experiments of [3], in most cases, Re-merge strategy would perform better than the other two approaches.

DBT[1] index construction strategy is a kind of Re-merge algorithm and performs the best among all available schemes. A DBT is an m-way tree, each node of which is a sub-index. The tree is divided into H layers from bottom to top. At layer k, the number of nodes is either zero, or is less than m. Let $\mathbf{E}_{k,j}$ be the capacity of node $j, 0 \leq j < m-1$, at layer $k, 0 \leq k < H$, a constraint of the number of documents in one node of layer k is given by:

$$c^k \leq \epsilon_{k,j} < c^{k+1} \tag{1}$$

When the size of each node in layer k satisfy the above equation, the tree is balanced. When the number of nodes in the layer k is equal to or greater than m, a merge event is triggered, resulting a new sub-index. The newly created sub-index will be placed into layer $k+1$. If the tree is balanced and a sub-index merge operation is only performed on one layer, the efficiency of merging process can be guaranteed. According to the experiments in Ref[1], choosing the param value of $m = c = 3$ offers better indexing performance.

In the implementation, the index is organized into several barrels. One barrel refers to one node in the DBT tree. When constructing the index, the postings will be built up in memory at first, when the memory has been exhausted, these postings will be flushed to one barrel, the layer of which in the DBT tree could be computed by the above equation according to the document number that have been indexed in this barrel. If the number of barrels of a certain layer satisfy the merge condition described above, these barrels will be merged together to form a new barrel, and the old barrels will be deleted. Therefore, merge operation has been limited within several barrels since it is an expensive process, and the total barrels could be controlled because too many barrels will effect the query performance. What's more, all of the barrels could be merged into one monad manually to provide better query performance.

2.2 Index Compression

Index compression is very useful, because it can accelerate the speed of extracting postings and reduce the index size. There exists a trade off between the compressibility and decoding speed. Higher compression ratio would always lead to slower decoding, if it is too slow, then query performance will be effected. D-gap together with variable length are adopted to compress the index, because it is very fast to decode, and can reduce the index size to about $\frac{1}{2}$ to $\frac{1}{4}$ of the original. It is a mature scheme and has been proven effectively.

2.3 Index Deletion

There exists a simple and effective approach to remove an indexed document from the index: Just generate a bitmap file for each barrel of index, each of its bit indicates that document has been removed if the bit is set to 1. In order to avoid of allocating a too much big bitmap caused by the huge number of document id, the document id should be allocated by the IndexManager and could change when index merge happens, in that case, each bit of the bitmap only indicates the gap value of the document id relative to its previous document.

However, it could not be available for IndexManager because there is a limitation that, as soon as the documents id has been assigned, it could not be changed anymore. Therefore, this has inevitably brought a problem that the bitmap can only map the real document id, leading to a impractical size of the bitmap file (document id is an unsigned 32-bit integer, anyone can infer the largest size that the bitmap file could be). What's more, it has brought another fatal problem: how to update a document? The updating operation is composed of removing a document followed by inserting that document again. If the bitmap indicates the document to be deleted, and if the document id could not be reassigned, how to distinguish between the removed old document and the newly added document with the same id? Therefore, IndexManager has to choose to delete the postings of a document physically although it is expensive.

2.4 B-Tree Index

When indexing documents, some kinds of data are not suitable to be stored into inverted index, such as date and time, number, etc, because range query ability would be provided, B-Tree is suitable for dealing with it. The data to be stored in B-Tree is a pair whose key is a three tuple: [collection-id, field-id, key value], where collection-id refers to the id of the collection that include the document to be indexed, field-id refers to the id of the certain field whose data is needed to be stored in B-Tree index. Therefore, the query will involve a multi-dimensional search operation. According to the research results, choosing composite-key solution while not a multi-dimensional search tree could provide better query performance and is much easier to be implemented.

2.5 Distributed Index

There are two approaches to implement distributed index: document-partition and term-partition. For document-partition, when indexing a collection, the document are dispatched to a certain server according to its document id, and all the distributed servers will search their indexes when there is a search request. This strategy can provide better load-balance, but each search operation will have to involve searching all machines, it has strengthen the system burden. For term-partition, when indexing a collection, the document is dispatched to all the servers, and each server will determine whether to store a posting according to its term id. When there is a search request, only servers have the term to be searched are required to deal with the search request. This strategy could provide less burden, but worse load balance. As to the IndexManager, both of these strategy are supported and it can be configured to adopt either of them.

There are two issues of designing the distributed index:How to dispatch a request to some distributed servers and how to combine the results got from them; How to deal with the servers' failure or turnover. The first issue is to design a map/reduce framework essentially, it will be discussed later when describing the IndexManager's architecture. Now let us illustrate how to deal with the second issue.

When dispatching a request to some server, how does the system know which server the request will be dispatched? Traditionally, a Modulo hashing algorithm could be adopted according to the request's key, in the IndexManager, the key could be either document-id or term-id. For example there are N servers in the system, then the server with id of key mod N will be chosen to process the request. If a new server is coming, then all the data stored in the N servers will be obsolete because the hash value of mod N completely changed, therefore a limitation has been added that the number of maximum servers should be fixed. If a certain server is down because of some reason, then all requests that would have been dispatched to that server will be rejected. Therefore we introduce the Consistent Hashing algorithm[2] - consistently maps objects to the same server, as far as is possible, at least.

The basic idea behind the consistent hashing algorithm[2] is to hash both objects and servers using the same hash function. The reason to do this is to map the server to an interval, which will contain a number of object hashes. If the server is removed then its interval is taken over by a server with an adjacent interval. All the other servers remain unchanged. Let's look at this in more detail. The hash function actually maps objects and servers to a number range. Imagine mapping this range into a circle so the values wrap around. Here's a picture of the circle with a number of objects (1, 2, 3, 4) and servers (A, B, C) marked at the points that they hash to.

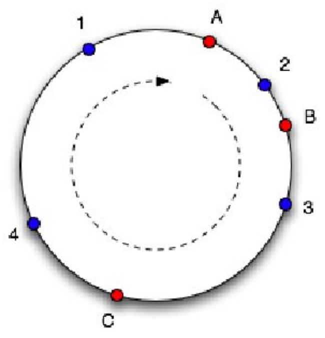


Figure 1: Consistent hashing circle

To find which server an object goes in, we move clockwise round the circle until we find a server point. So in Figure 1, we see object 1 and 4 belong in server A, object 2 belongs in server B and object 3 belongs in server C. Consider what happens if server C is removed: object 3 now belongs in server A, and all the other object mappings are unchanged. If then another server D is added in the position marked it will take objects 3 and 4, leaving only object 1 belonging to A, shown in Figure 2.

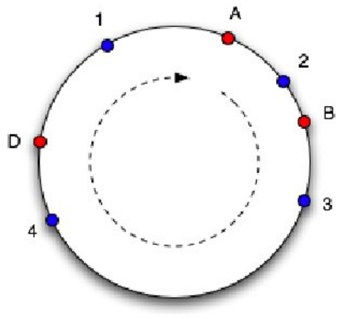


Figure 2: Consistent hashing circle without object C

This works well, except the size of the intervals assigned to each server is pretty hit and miss. Since it is essentially random it is possible to have a very non-uniform distribution of objects between servers. The solution to this problem is to introduce the idea of ”virtual nodes”, which are replicas of server points in the circle. So whenever we add a server we create a number of points in the circle for it.

When applying consistent hashing in IndexManager, the working flow is: The request is dispatched to servers by the consistent hashing value according to document id or term id. When new server is added or removed, we record the adjacent server next to the changed server, then requests that would have been dispatched to that changed server will only be forwarded to these two servers because data exists only on both of them. This design has conquered the limitations of Modulo hashing and could provide very high availability and scalability.

3 Architecture

3.1 Indexer

Indexer is the main interface class of IndexManager. See Figure 3.

It interacts with ConfigurationManager to get parameters, and pass them to IndexWriter and IndexReader to process the index writing(insert, remove) and reading operation(search). Indexer also interacts with DocumentManager to get the Document objects. There are three modes that Indexer works: working locally, working as the client of distributed index, and working as the server node of distributed index. IndexBroker is responsible for dispatching all requests and wait for the results.

3.2 Document and Collection Management

Document is the key interface among IndexManager, DocumentManager and LAManager. At the beginning, a document is an object from DocumentManager which has just initialized itself using the web page content. Then it is passed to IndexManager, which calls LAManager

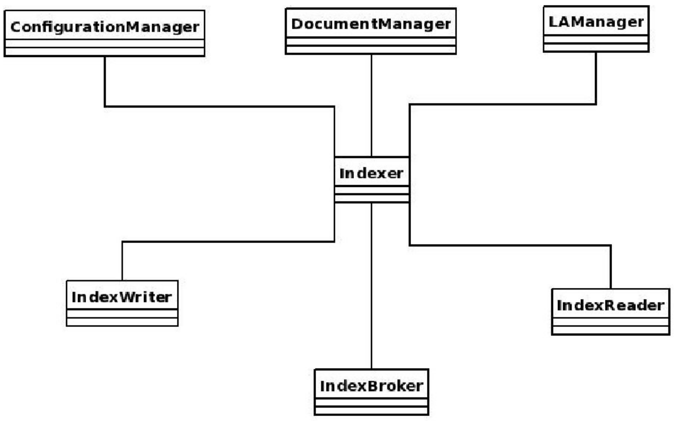


Figure 3: Indexer Class

to tokenize its content. After this procedure, the content of the document has been converted into a serial tokens, which will be put into the inverted index soon after. To standardize this whole flow, a new class DocumentSchema is introduced. The relationships among them could be shown in Figure 4.

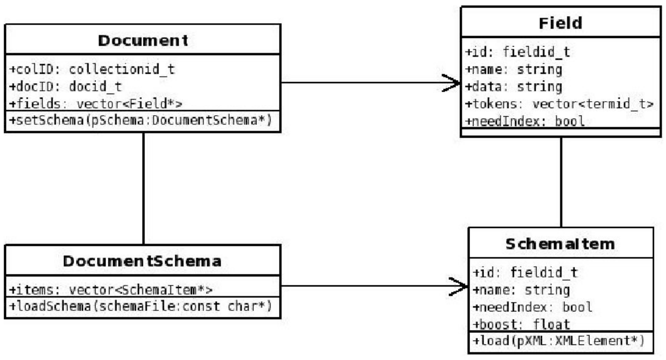


Figure 4: Document and DocumentSchema Class

The DocumentSchema initializes itself using a configruation xml file. With such a mechanism, the indexing process of the documents could be standardized and also configurable. A Document is composed of several Fields, whose meta information such as the field-id, field-name, field-type could be got from the above referred configuration file. The introduction of the concept of field-type is to differentiate those Fields whose data is not suitable to put into inverted index, such as date and time, digital number, etc, these data is more suitable to put into a B-tree.

IndexManager is able to support indexing different collections together. Since documents in different collections could have the same document id, therefore, to identify a document requires both collection id and document id. If we put both collection id and document id into the posting lists, the index's size will contain too much redundancy. In order to differentiate documents with same document id but belongs to different collections in the index, we store the vocabulary of a barrel according to the sequence of collections, and each term in the vocabulary of each collection have its own pointer to its according posting. Therefore, the inverted index is composed of several barrels, each barrel contains indexes of different collections, each collection index is composed of field indexes, if the field data is number or date and time, then they are put into B-trees alone.

3.3 IndexWriter

3.3.1 Building a Single Barrel Index

IndexWriter is responsible for index construction and updating. Figure 5 shows the hierarchy of IndexWriter of how to form a single barrel index. And Figure 6 shows us the Class call sequence when inserting a collection to construct index. And Figure 7 shows us the Class call sequence when deleting a collection.

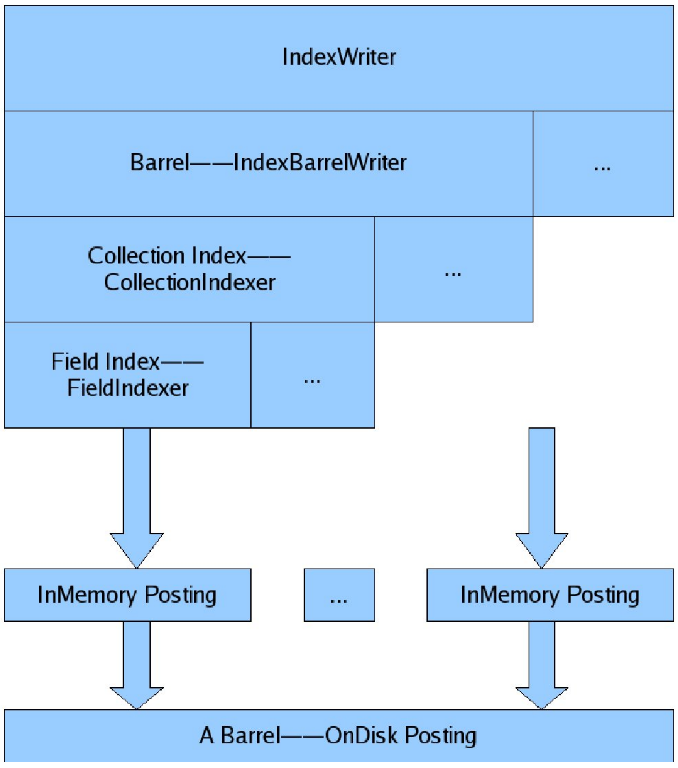

IndexWriter
Barrel——IndexBarrelWriter
...
Collection Index—— CollectionIndexer
...
Field Index—— FieldIndexer
...
InMemory Posting
...
InMemory Posting
A Barrel——OnDisk Posting

Figure 5: Hierarchy of IndexWriter

As has been shown above, IndexBarrelWriter is in charge of flushing index into one barrel. Internally, it contains a serial instances of CollectionIndexer, each of which corresponds to one collection solely. Inside a CollectionIndexer, there are several FieldIndexers, which takes charge of building index within its corresponding Field. After Indexer starts up, the instances of CollectionIndexer and FieldIndexer will be created according to DocumentSchema that have been initialized by the ConfigurationManager which was referred in the previous section. B-Tree indexer is built based on SequentialDB which is not included in this technical report.

FieldIndexer will build index of one field of a collection. Before an index barrel is formed, the index exists with the form of InMemoryPosting. There are two kinds of posting lists in the system: document-frequency posting and position posting. The former stores document id and frequency of a certain term that has appeared in that document. The latter stores all the term position information of a certain term in the document. When the memory pool has been exhausted, these in-memory index is flushed to disk.Figure 8 gives a detailed description of what a single barrel index contains.

Let us illustrate these files one by one.

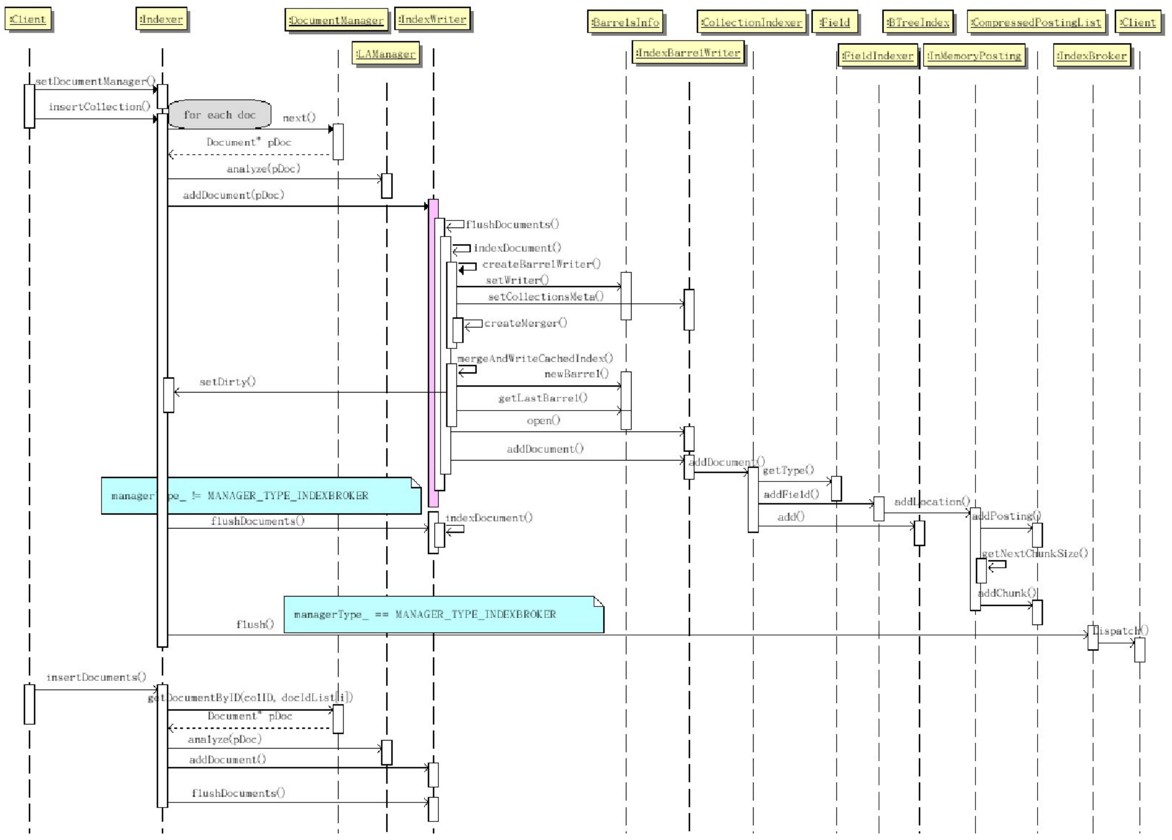


Figure 6: Index construction sequence

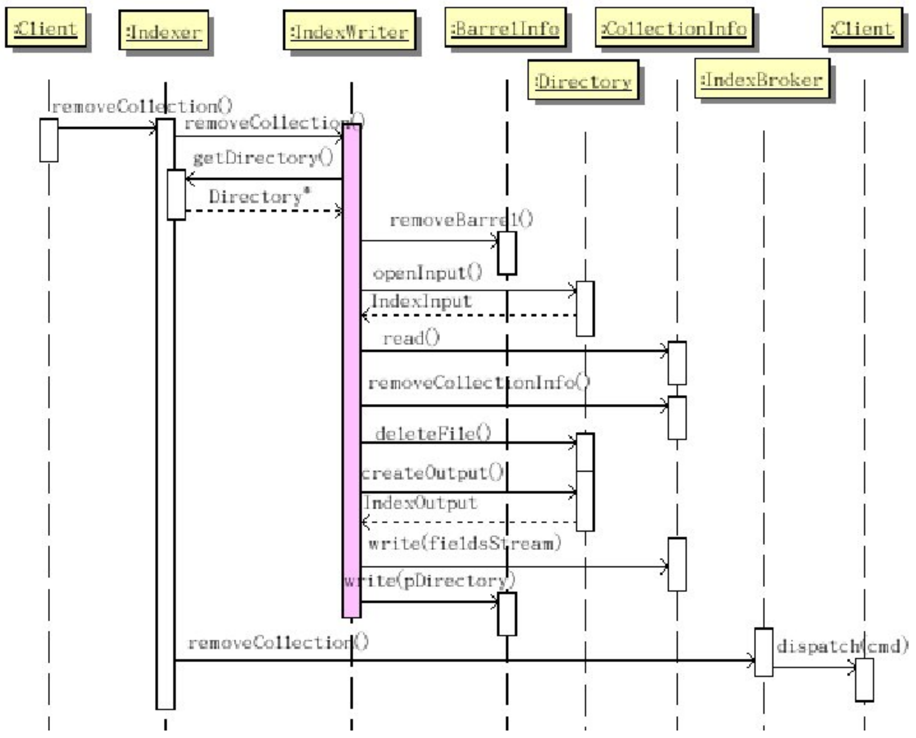


Figure 7: Index deletion sequence

.fdi stores information of all fields.

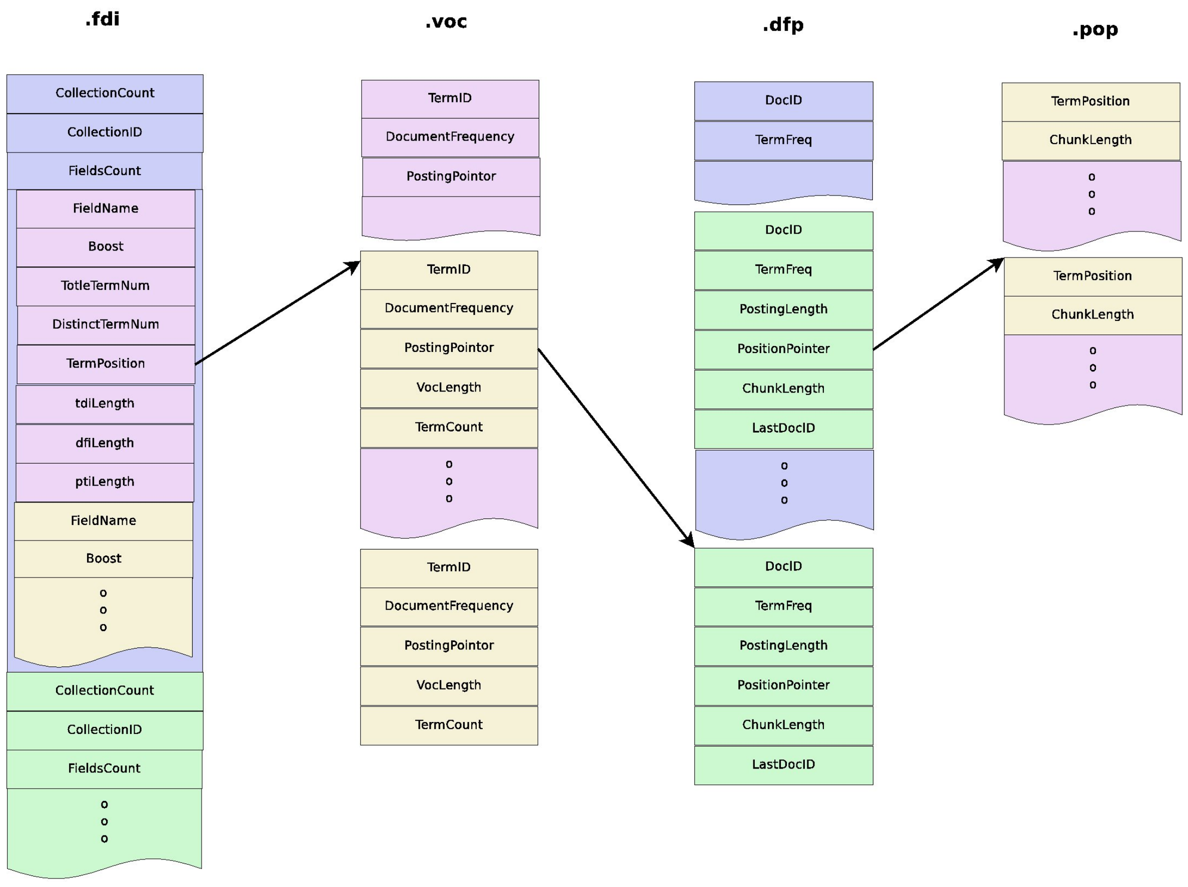


Figure 8: Index format of a barrel

Property	**Type**	**Description**
CollectionCount	Int32	The count number of collections in this barrel
CollectionID	Int32	The collection id of this collection
FieldsCount	Int32	The count number of fields in this collection
FieldName	string	Field name
Boost	Byte	Boost value of this Field
TotalTermNum	Int64	Total term number of this Field
DistinctTermNum	Int64	Total distinct term number of this Field
TermPosition	Int64	File offset of this Field's vocabulary information in the .voc file
tdiLength	Int64	Length of this Field's vocabulary information in .voc file
dfiLength	Int64	Length of this Field's document-frequency postings in the file .dfp
ptiLength	Int64	Length of this Field's position postings in the file .pop

.voc vocabulary information of all fields.

Property	**Type**	**Description**
TermID	Int32	Term ID, it is stored with d-gap encoded.
DocumentFrequency	Int32	The Document frequency, which means how many documents that this term has appeared.
PostingPointer	Int64	File offset of this term's document-frequency posting in the .dfp file
VocLength	Int64	Same as tdiLength in .fdi
TermCount	Int64	Same as DistinctTermNumin .fdi

.dfp document-frequency posting.

Property	**Type**	**Description**
DocID	Int32	Document ID, which is stored with d-gap encoded.
TermFrep	Vint64	Term frequency in this document.
PostingLength	Vint64	Length of this posting
PositionPointer	Vint64	File offset of the corresponding position posting in the .pop file.
ChunkLength	Vint64	Same as PostingLength
LastDocID	Vint32	Last document id of this posting.

.pop position posting.The positions of a term in a document is written in the .pop file sequentially.

Figure 9 shows the class diagram of Posting. Either document-frequent posting list or position posting list is an instance of CompressedPostingList and they are contained by InMemoryPosting, which takes charge of allocating memory from memory pool for them and flushing them to files. Both OnDiskPosting and InMemoryPosting are inherited from Posting, they are responsible for decoding the posting lists locating in memory or on disk.

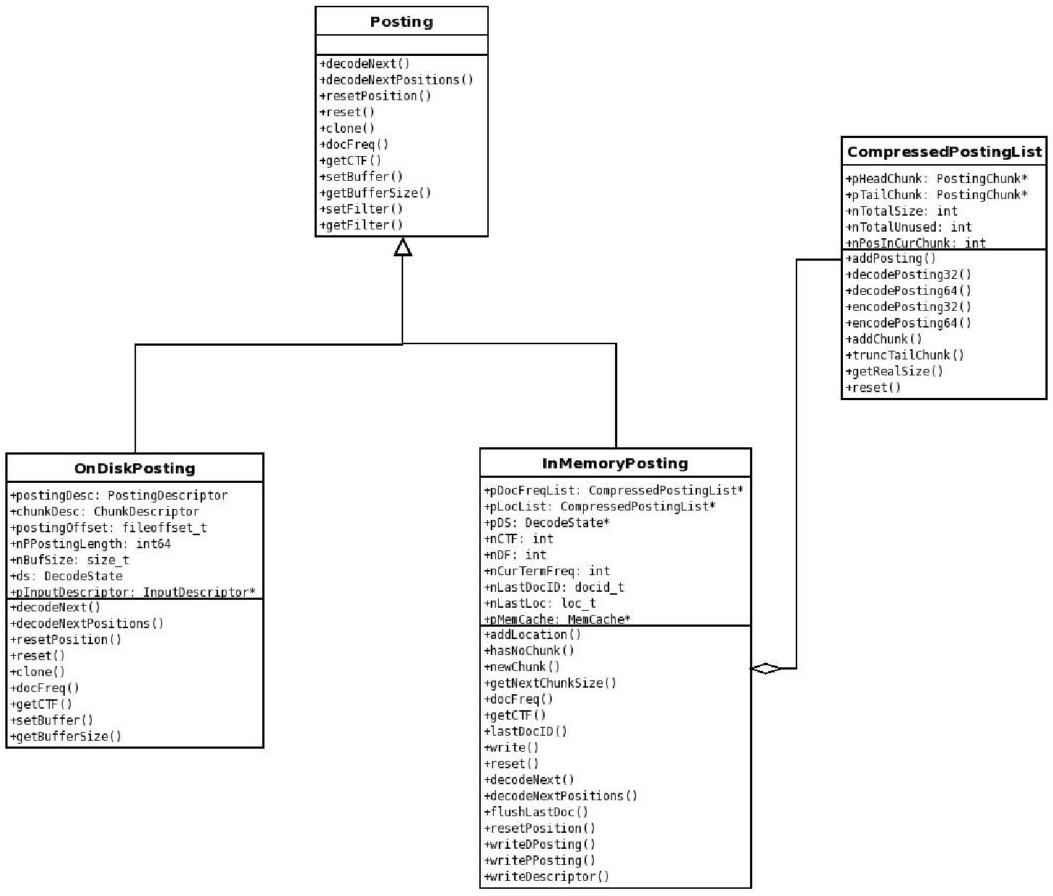


Figure 9: Class Posting

3.3.2 Merging Index of Different Barrels

After a single barrel has been formed, if the index construction process has not been finished, the index will be flushed to new barrels and if the merge event is triggered as mentioned in section 2.1, IndexMerger will start to merge different barrels into a new barrel, old barrels will then be removed. Similarly, IndexMerger also has a hierarchy which is shown in Figure 10. When a merge event happens, all the barrels that are needed to be merged are ordered according to the document number that each barrel has contained, then the merging process will be proceeded field by field and collection by collection. FieldMerger is responsible for merging same field of a collection in different barrels, and PostingMerger will merge postings at the term level.

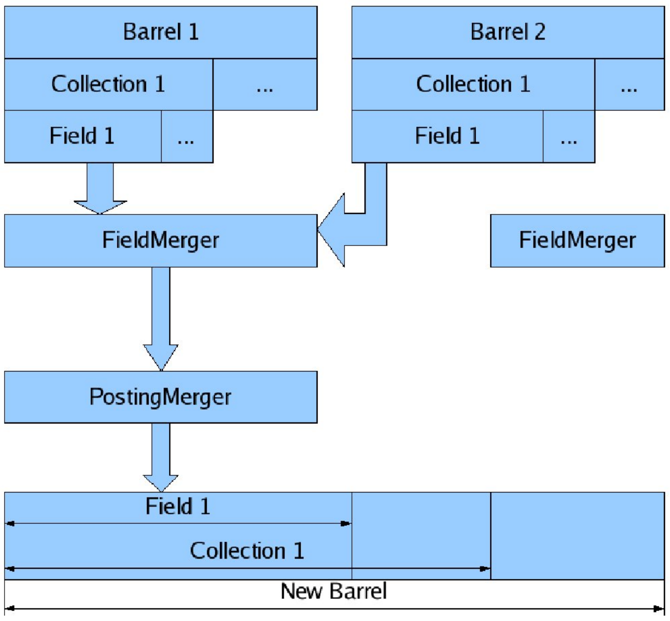


Figure 10: Hierarchy of IndexMerger

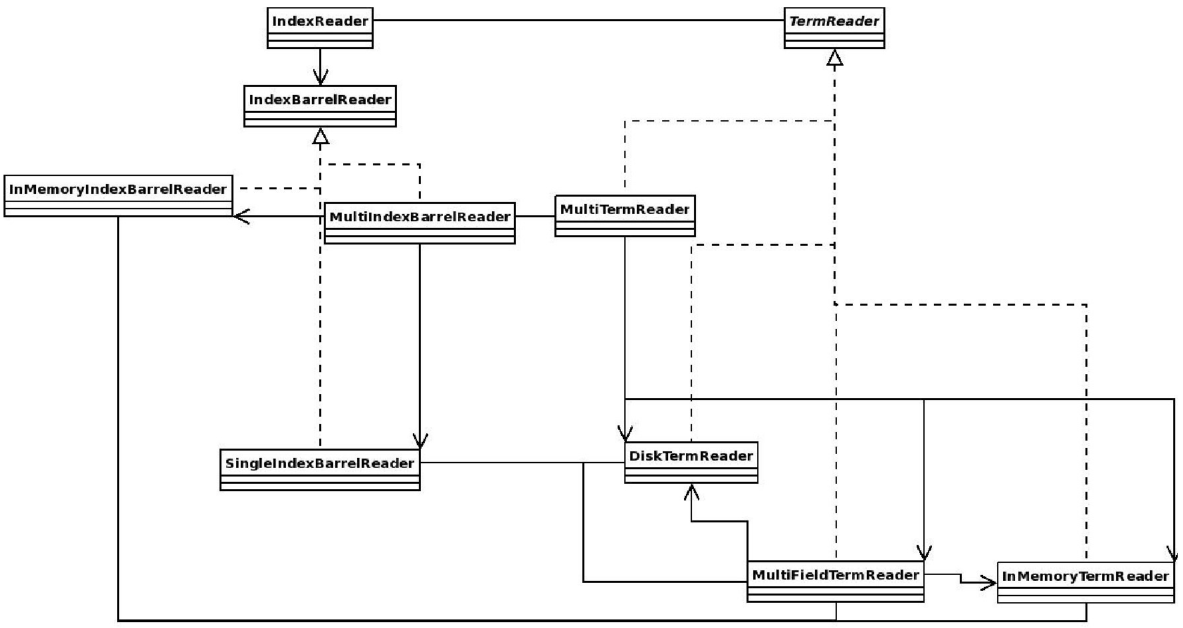


Figure 11: Components of IndexReader

3.4 IndexReader

IndexReader is the interface to read inverted indexes and search them. Figure 11 gives a detailed class diagram of the components of IndexReader. Figure 12 shows us the Class call sequence when query request happens. When IndexReader is created, it will open the index barrels and return an instance of TermReader to users. TermReader takes charge of seeking a term in the vocabulary of the indexes, and then iterating its corresponding posting list according to the user's requests. SingleIndexBarrelReader is used to open a single index

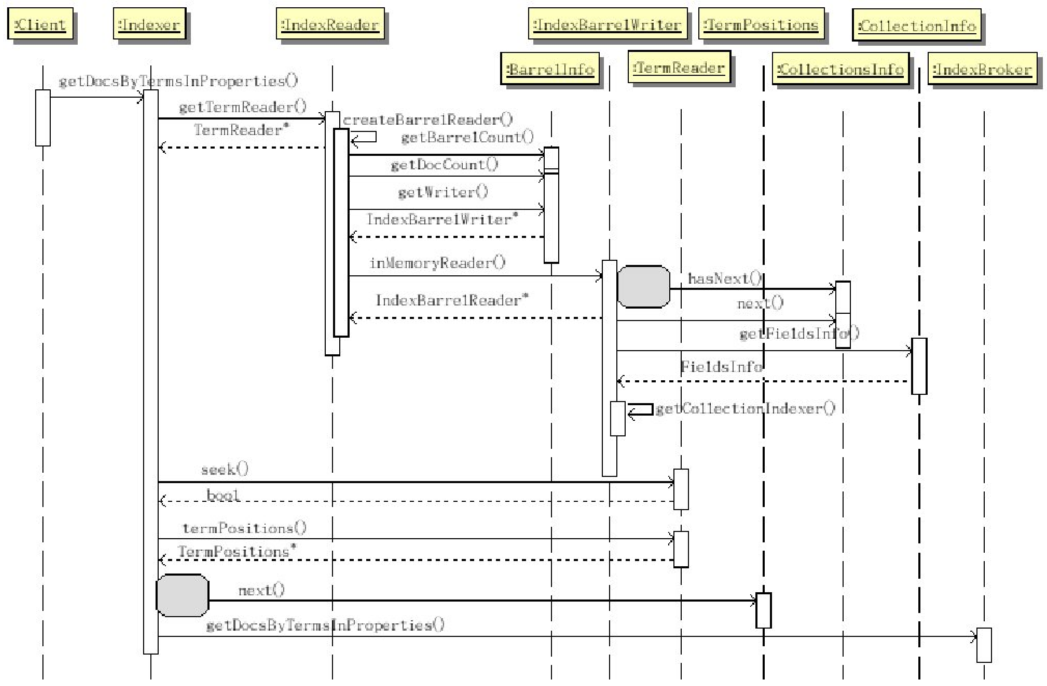


Figure 12: Index querying sequence

barrel and return its TermReader, here it is the DiskTermReader. Because IndexManager should support on-line indexing, which means the indexes could be searched during index construction, therefore, those indexes that have not been flushed to disk should also allow being searched, InMemoryIndexBarrelReader takes this responsibility. What's more, since there may exist several barrels in the system, then we have MultiIndexBarrelReader which contains several instances of SingleIndexBarrelReader or InMemoryIndexBarrelReader, each of which takes charge of reading their corresponding sub-index. Accordingly, the TermReader got by InMemoryIndexBarrelReader should be InMemoryTermReader, and the TermReader got from MultiIndexBarrelReader should be MultiTermReader, which is composed of several instances of DiskTermReader and InMemoryReader similarly.

After getting an instance of TermReader, the user could use it to search the inverted indexes: if the term to be queried could be found by TermReader in the indexes, then TermReader could return an index iterator TermDocFreqs or TermPositions. Just as Figure 10 shows. TermDocFreqs will iterate the document-frequent postings and TermPositions will iterate document-frequent postings and position postings both, therefore TermPositions is inherited from TermDocFreqs.

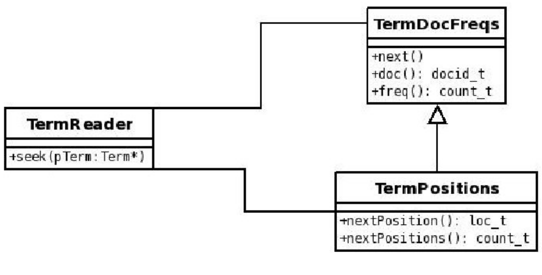


Figure 13: TermDocFreqs and TermPositions

3.5 Working Distributively

When working under distributed environment, there are three roles in the system: Client, Broker, and Datanode, as shown in Figure 14.

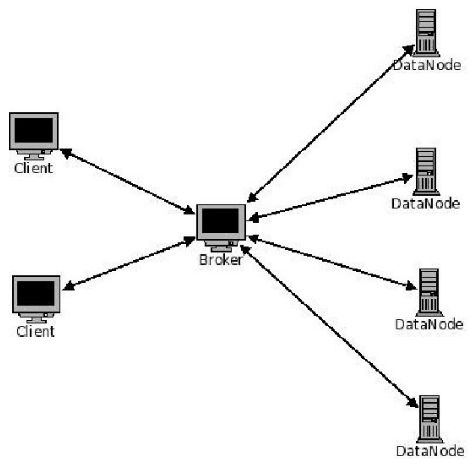
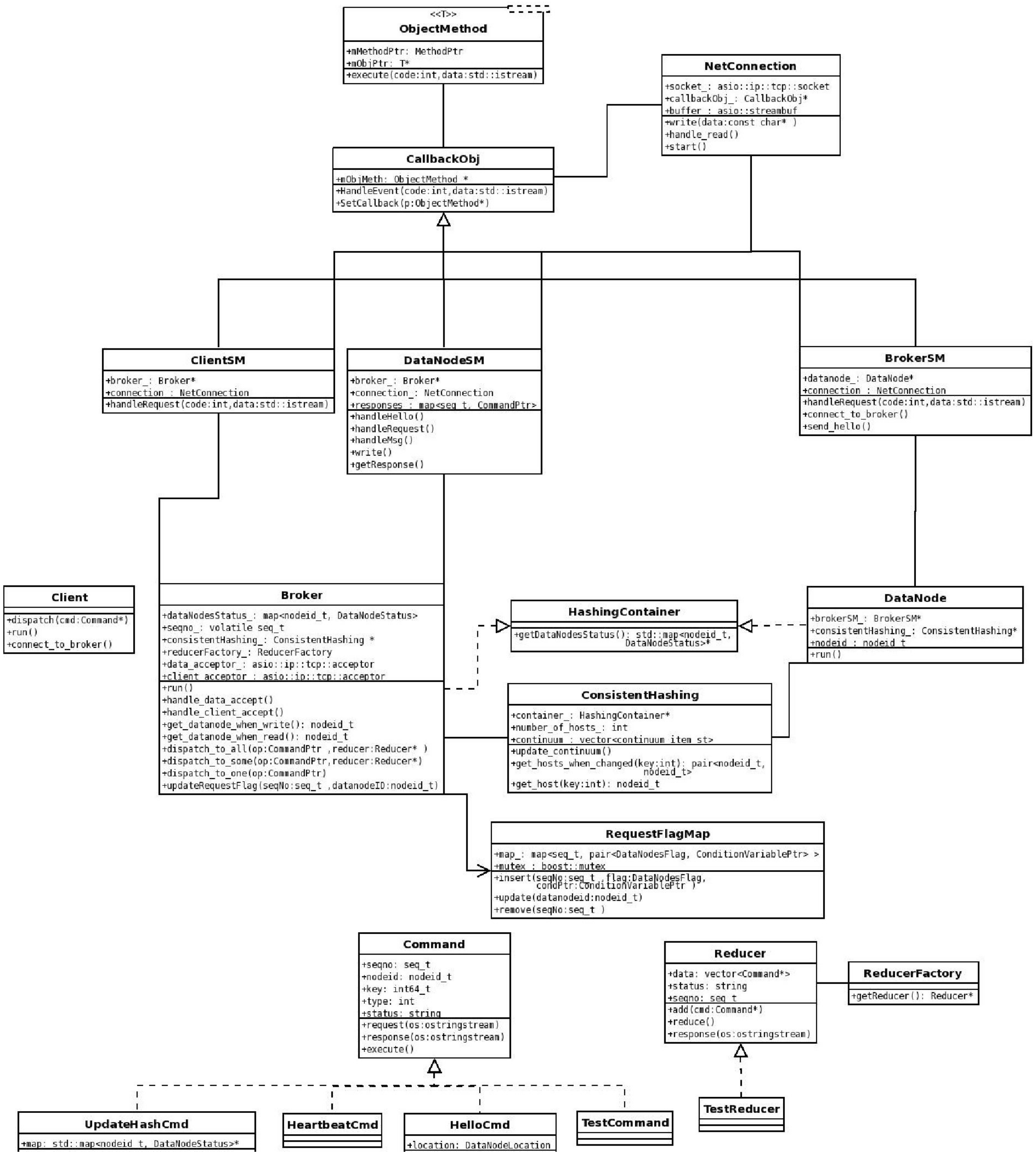


Figure 14: Topology of Distributed Index

Broker should work alone and act as a server. DataNode is the entity who operates on indexes at its local machine and connects to the Broker when it starts up. The connection between Broker and DataNodes is based on TCP, in order to reduce the communication's overhead, the connection should be kept alive, to satisfy this requirement, the Broker will send a heartbeat detection command to all the DataNodes every 60 seconds. The Client is the interface to users, it is responsible for sending user's requests to the Broker and wait for the results. The Client, Broker, and DataNode could be extended to support any kinds of distributed applications, it has implemented a basic map/reduce framework with high availability and scalability because of the introduction of Consistent Hashing described in section 2.5. Figure 15 gives a detailed class diagram of this map/reduce framework.

One of the design idea of the framework is to reuse the codes about the socket communication. We adopt the Boost::Asio toolkit because it has given a good encapsulation of sockets under a portable environment, what's more, Asio has adopted an event based communication design pattern Proactor[5], as the fundamental layer, which is the best solution to process tasks with high concurrency. Class NetConnection is an encapsulation of Boost::Asio. At the Broker side, class ClientSM takes charge of processing communication between Client and Broker and class DataNodeSM takes charge of the communication between Broker and DataNode, while at the DataNode side, class BrokerSM is responsible for the communication to the Broker. NetConnection is reused by all these three classes, and through setting the callback object handle, when NetConnection receives a message, it will pass the message to the callback object to let it handle at application layer, ClientSM, DataNodeSM, and BrokerSM are all callback objects. Through this callback mechanism, we have reached the aim of code reusing, as well as the logic of application has been separated from the communication layer, which is much clear and easy to maintain or scale.

Class Broker and DataNode are the main frame of Broker and DataNode respectively. They are both inherited from HashingContainer, because both Broker and DataNode are required to maintain the consistent hashing. As to the Broker, it needs to decide which DataNode to dispatch when it receives the Client's request, as to the DataNode however, the consistent hashing is only useful when the term-partition distribute strategy is adopted, because, the DataNode needs to know whether or not the term is needed to be indexed



<<T>>
ObjectMethod
+mMethodPtr: MethodPtr
+mObjPtr: T*
+execute(code:int,data:std::istream)
NetConnection
+socket_: asio::ip::tcp::socket
+callbackObj_: CallbackObj*
+buffer_: asio::streambuf
+write(data:const char*)
+handle_read()
+start()
CallbackObj
+mObjMeth: ObjectMethod *
+HandleEvent(code:int,data:std::istream)
+SetCallback(p:ObjectMethod*)
ClientSM
+broker_: Broker*
+connection_: NetConnection
+handleRequest(code:int,data:std::istream)
DataNodeSM
+broker_: Broker*
+connection_: NetConnection
+responses_: map<seq_t, CommandPtr>
+handleHello()
+handleRequest()
+handleMsg()
+write()
+getResponse()
BrokerSM
+datanode_: DataNode*
+connection_: NetConnection
+handleRequest(code:int,data:std::istream)
+connect_to_broker()
+send_hello()
Client
+dispatch(cmd:Command*)
+run()
+connect_to_broker()
Broker
+dataNodesStatus_: map<nodeid_t, DataNodeStatus>
+seqno_: volatile seq_t
+consistentHashing_: ConsistentHashing *
+reducerFactory_: ReducerFactory
+data_acceptor_: asio::ip::tcp::acceptor
+client_acceptor_: asio::ip::tcp::acceptor
+run()
+handle_data_accept()
+handle_client_accept()
+get_datanode_when_write(): nodeid_t
+get_datanode_when_read(): nodeid_t
+dispatch_to_all(op:CommandPtr ,reducer:Reducer*)
+dispatch_to_some(op:CommandPtr,reducer:Reducer*)
+dispatch_to_one(op:CommandPtr)
+updateRequestFlag(seqNo:seq_t ,datanodeID:nodeid_t)
HashingContainer
+getDataNodesStatus(): std::map<nodeid_t, DataNodeStatus>*
DataNode
+brokerSM_: BrokerSM*
+consistentHashing_: ConsistentHashing*
+nodeid_: nodeid_t
+run()
ConsistentHashing
+container_: HashingContainer*
+number_of_hosts_: int
+continuum_: vector<continuum_item_st>
+update_continuum()
+get_hosts_when_changed(key:int): pair<nodeid_t, nodeid_t>
+get_host(key:int): nodeid_t
RequestFlagMap
+map_: map<seq_t, pair<DataNodesFlag, ConditionVariablePtr> >
+mutex_: boost::mutex
+insert(seqNo:seq_t ,flag:DataNodesFlag, condPtr:ConditionVariablePtr)
+update(datanodeid:nodeid_t)
+remove(seqNo:seq_t)
Command
+seqno: seq_t
+nodeid: nodeid_t
+key: int64_t
+type: int
+status: string
+request(os:ostringstream)
+response(os:ostringstream)
+execute()
Reducer
+data: vector<Command*>
+status: string
+seqno: seq_t
+add(cmd:Command*)
+reduce()
+response(os:ostringstream)
ReducerFactory
+getReducer(): Reducer*
TestReducer
UpdateHashCmd
+map: std::map<nodeid_t, DataNodeStatus>*
HeartbeatCmd
HelloCmd
+location: DataNodeLocation
TestCommand

Figure 15: Map/Reduce Framework

locally. Consistent hashing is built up using the information of all the datanodes, including the datanode-id and the datanode status. If any of the DataNode's status has been changed, the consistent hashing is required to rebuild. That is to say, the Broker should notify all the available DataNodes when a new DataNode comes or some DataNode has been down.

We have defined a base class Command, to take charge of the messages between the Client ,the Broker, DataNodes. All possible messages should inherit from Command and we call these messages Commands. In order to identify different commands, a seqno is needed and is allocated by Broker to make sure it is unique globally. The command sent from Client should contain the field value of key with which the Broker could decide which DataNode the command will be forwarded using the consistent hashing. The base class Command has three pure virtual method: request(), response() and execute(). Request() is used to form a string stream to be sent when the command is going to be sent with the direction of Client to DataNode, while response() is used to form a string stream to be sent when the command is going to be sent with the direction of DataNode to Client. For example, suppose Client needs to send TestCmd, then it will call request() at first, after the Broker received data, it will parse the data in to an instance of TestCmd, retrieving the field value of key then before the Broker dispatches the TestCmd to DataNode A, it will also call request() to form the data to be send. In the opposite direction, DataNode calls response() to form the data before it is sent to Broker, and Broker also calls response() to return to Client. Execute() is only useful at the DataNode side, it is used to process the practical application logic when the DataNode receives a certain command. HelloCmd, HeartbeatCmd and UpdateHashCmd are the three inherent commands of the framework. When a DataNode has successfully connected with the Broker, it will send HelloCmd to the broker to tell its information, such as datanode id, ip address, etc. UpdateHashCmd is used to notify all the DataNodes to update their consistent hashing, as we have described it before, and the usage of HeartbeatCmd was also referred above.

If the command from Client is needed to be sent to more than one DataNode, then the Broker must combine the responses got from all the DataNodes, this procedure is called reducing, as shown in Figure 16.

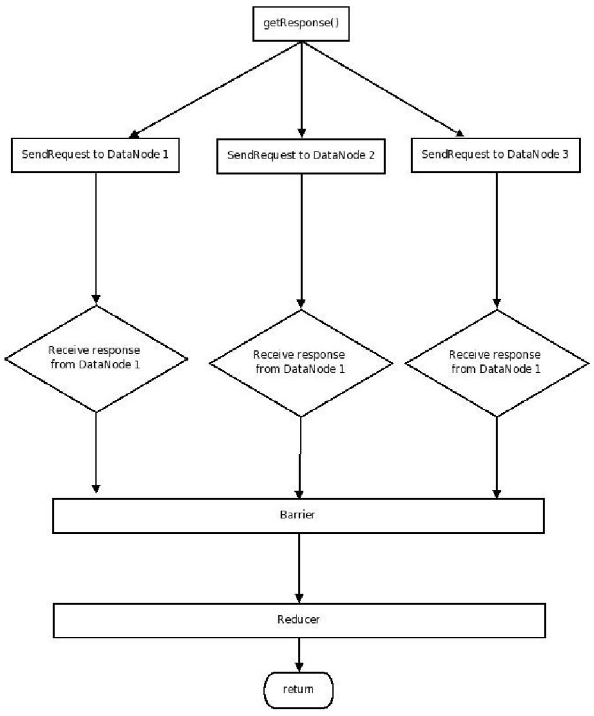

getResponse()
SendRequest to DataNode 1
SendRequest to DataNode 2
SendRequest to DataNode 3
Receive response from DataNode 1
Receive response from DataNode 1
Receive response from DataNode 1
Barrier
Reducer
return

Figure 16: Reducing

The Broker should support high concurrency, however the reducing operation will inevitably lead to wait because reducing could only be started when receiving from all the responses. When facing such an issue, we have used two means to resolve it: One is to let the Broker run with a thread pool; The other is the introduction of an internal class of RequestFlagMap.It has an internal data structure:

$$map < seq\_t, pair < map < nodeid\_t, bool >, boost :: condition\_variable* >>$$

This structure is a map, each record of which indicates the status of a command that the Broker has dispatched to some DataNodes. The key of that record is same as the seqno field of Command, which is unique globally and can identify that command. The value of that record is a pair, the first of which is also a map, indicating whether or not the Broker has received the response from a certain DataNode, and the second of that pair is a condition variable. When a command has been dispatched to the DataNodes, an instance of condition variable is created to ask the current thread to yield. When all the responses are received, that condition variable will notify that thread, then the reducing operation could be proceeded. In order to apply the map/reduce framework to a specific application, these works are needed: Define the commands that the application needs, all of which should inherit from class Command, and implement their methods of request(), response(), execute(); Define the reducers of these commands; Define the parsers of these commands, all Client, Broker, and DataNode need parsers to parse received data into certain commands.

4 Usage

In order to make IndexManager work, it should know two kinds of facts—the init parameters to start up, and the format of documents in all collections to be indexed.

4.1 Init parameters of IndexManager

The init parameters of IndexManager could be got through an XML file named *config.xml* or though the interface to ConfigurationManager—*setIndexManagerConfig* directly.Let us use the format of *config.xml* to illustrate the usage of the init parameters and how to config them.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--start of the config.xml -->
<config>
        <indexstrategy>
                <!--the index’s working directory -->
                <indexlocation type="string">./index</indexlocation>
                <!--the access mode of the indexes, a means append, w refers to create, a is ok for
                        all kinds of situation -->
                <accessmode type="string">a</accessmode>
                <!--the size of the memory cache, when the memory cache is full, the indexes in memory will
                        be flushed into one barrel and a new barrel will be generated -->
                <memory type="int64">2800000</memory>
                <!--the max indexed terms of a document, if the memory cache is nearly full,when indexing a
                        new document ,this size will be used to malloc an emergency memory pool-->
                <maxIndexTerms type="int32">10000</maxIndexTerms>
                <!--the cached document number of IndexWriter, when the cached document reaches this number,
                        these documents will be indexed-->
                <cachedocs type="int32">100</cachedocs>
        </indexstrategy>
        <mergestrategy>
                        <!--the merge method of index. It could be "OPT" or "DBT", OPT means all the postings exist
                in only one barrel, it  could provide higher search performance while much lower indexing
                        performance,"DBT" is default -->
                <strategy type="string">DBT</strategy>
                <!--the param of the merge method, as have been described in section 2.1  -->
                        <param type="string">m=3</param>
        </mergestrategy>
        <storestrategy>
        <!--whether the indexes is stored in file or memory -->
                <param type="string">file</param>
        </storestrategy>
        <distributestrategy>
                <!--the working mode of Indexer. It could be "local", which means it works locally;"indexbroker",
                        which means the indexer works distributively and the indexer serves as the client; "indexnode",
                        which means the indexer works distributively and the indexer servers as the index datanode. -->
                <distribute type="string">local</distribute>
                <!-- the partition strategy when working distributively It could be "document", which means the
                        document partition strategy or "term", which means the term partition.-->
                <partition type="string">document</partition>
                <!--if the indexer works as the broker("indexbroker" is set in the distributed type),the format of
                        this item should be: brokerIP:port1:port2:num_of_datanodes:num_of_threadpool, where brokerIP
                        means the ip address of the broker runs, port1 means the listen port of the broker that
                        process the requests from the client, port2 means the listen port of the broker that process
                        the requests from the datanodes. num_of_datanodes means the initial count of the working data
                        nodes. num_of_threadpool means the size of the thread pool of the broker . If the indexer
                        works as the datanode, the format is brokerIP:port:num_of_threads, where brokerIP means the ip
                        address of the broker, port means the listen port of the broker that process the data from
                                datanodes, num_of_threads means the size of the thread pool of datanodes. -->
                <param type="string">127.0.0.1:8888:2000:3:8</param>
        </distributestrategy>


        <advance>
                <!--Memory Management Strategy -->
                <mms type="string">exp:32:2</mms>
        <!--when the memory cache of building up index is exhausted, if the documents have not yet been
                finished indexing, then it will enter the state of "UPTIGHT MEMORY ALLOCATE",then it resent
                the request to allocate memory with the size configured here. -->
        <memsize type="int32">40000</memsize>
        <!-- chunk size of posting. -->
        <chunksize type="int32">8</chunksize>
    </advance>


    <log>
    <level type="string">default_level</level>
    </log>
</config>
```

4.2 Formats of Documents in All Collections

Different collection have the according DocumentSchema's definition, which can let the IndexManager know the format of the documents in a certain collection, including the property name, property type(which has the same definition as DocumentManager's PropertyDataType), whether this property is needed to be indexed,property id, and its according collection id.

If it is required to work distributively, then three kinds of programs are going to be started: Client, which needs the distribute field in config.xml is set with indexbroker Broker, which runs independently as a server, and it also need config.xml to initialize itself, the relevant field to be used has been illustrated in the former example of config.xml. In fact, the Client and Broker always lie together, therefore the config.xml can be shared; DataNode, which needs the distribute field in config.xml is set with indexnode Each datanode will have its own id which should be sole in the whole system. The id could be assigned manually through the parameter of the parameter, or else, if the datanode program is started without any parameter, then the IP address of the machine that runs the datanode will be assigned as the id.

5 Experimentation and Performance

5.1 Index Construction

It has limitations to build up a test environment because of such reasons: DocumentManager is a memory based mock implementation, therefore the number of document objects is limited; In addition, the content of a document object is not flexible to change.The performance of index construction depends on the following factors:

- The size of memory pool, which could be customerized through configuration file. Since the index is constructed first in the memory, therefore the larger size of the memory pool has, the faster speed of the index is constructed.
- Number of tokens of each document object has.
- The size of term vocabulary. Larger size of vocabulary will lead to more posting lists, which will take more time to build index.

	Test 1	**Test 2**	**Test 3**	**Test 4**
Document Number	100,000	100,000	1,000,000	1,000,000
Memory Pool Size	2,800K	28,000K	2,800K	28,000K
Vocabulary Size	100	100	10000	10000
Number of Tokens per Doc	300	300	300	300
Fields Indexed	2	2	2	2
Collection Size	156M	156M	1.8G	1.8G
Index Size	30M	30M	793M	793M
Index Construction Time	4s	2s	501s	365s
Index Merge Time	$< 1s$	$< 1s$	51s	51s

As can be seen from the experiments, If we changed the memory pool size from 2,800K bytes to 28,000K bytes, the index construction would be speeded up from 501s to 365s. The Index Merge Time in the above table means the time that we merge the multi-barrel index into a monad manually, which can improve the query performance. In summary, the IndexManager can construct the index very fast and would not be the bottle neck of SF-1. The index size is relative large because of the position posting lists: our mock DocumentManager generate the document objects whose positions are added at random, therefore the d-gap encoding and variable-length would not take into effect on the *.pop files, which are the position postings. In the real world, however, under the similiar situations, the index size would add up to about 630M or so.

5.2 Index Query

According to our experiments, the index query performance mainly depends on the API's encapsulation.

	Test 1	**Test 2**	**Test 3**	**Test 4**
Document Number	100,000	100,000	1,000,000	1,000,000
Vocabulary Size	100	100	10000	10000
Number of Tokens per Doc	300	300	300	300
Fields Indexed	2	2	2	2
Optimized Index	NO	YES	NO	YES
Term Queried	100	100	100	100
Query Time	10s	8s	4s	2s

In Test 1 and Test 2, the index contains 100,000 documents and postings of each term will also include about 100,000 elements. If there are multi index barrels which means the index has not been optimized manually, then it would take 10s to query a certain term for 100 times. As can be seen from Figure 17, scanning the index, which includes searching the index files to locate the posting, retrieving the posting list and decompress it, occupies only about 22% of the total time elapsed.

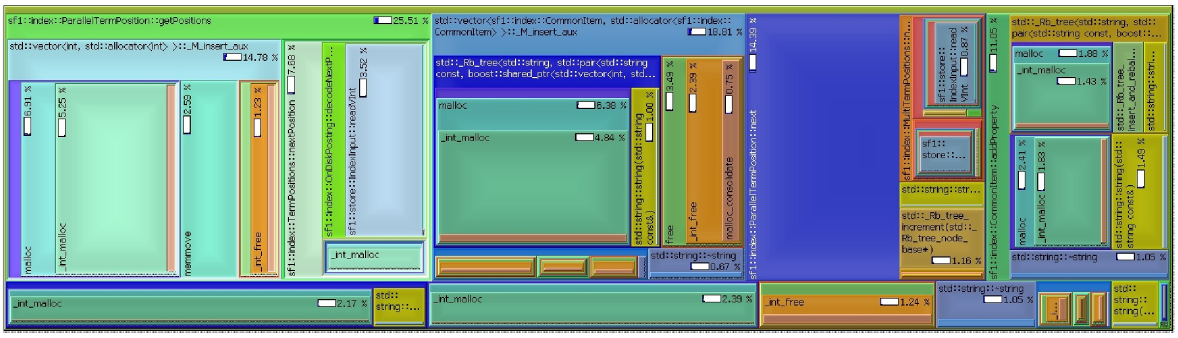


Figure 17: Function execution time

In Figure 17, Function TermPositions::nextPosition and ParallelTermPosition::next are responsible for scanning the index, they have occupied 7.68% + 14.39% = 22.07% of the total execution time. Putting the outputs into the vector that the API requires have spent 14.78% + 18.81% + 11.05% = 44.64% of all the execution time. Other time elapsed are taken by the destructor of the encapsulated outputs objects. If we optimize the index into a single monad and repeat the query process, then the time elapsed will decrease from 10s to 8s. Therefore we can draw a conclusion that, if the postings in the index contain lots of elements, then the bottleneck of the query performance does not lie in the index itself, but in the API's encapsulation. If the query performance is required to be improved, then the most important measture to be taken is to redesign the API.

If we built up the index which contains 1,000,000 documents but the vocabulary size is still 100, which means the posting list of a certain term would include about 1,000,000 elements, when we repeat the query experiment, the time elapsed would increase to more than 50 seconds. However, if the we change the vocabulary size from 100 to 10000, just as Test 3 and Test 4 indicates, then the posting list of each term would contain less elements, therefore the query response would be much more quick: Test 3 has taken only 4 seconds and if we optimized the index manually, only 2 seconds is needed, just as Test 4 indicates.

6 Future Improvements

Possible improvements could come from these aspects:

- Add SkipList to postings to improve AND query performance.
- Add index pruning to IndexManager. As we have shown in part 6, the query performance will be decreased severely if the postings contain lots of elements. Therefore it is reasonable to return top-k results while not total. Through index pruning the top-k results can be got much faster[4, 6]. Index pruning can be divided into term pruning and document pruning. As to the large postings, the latter scheme is more useful because only top-k document ids are kept in the pruned postings. This solution will require term weighting values such as BM-25, which is provided by interfaces of RankManager. Both of these two improvement items will affect the whole system design.
- Substitute the index compression scheme of variable-length with the PForDelta approach[8]. According to Jiangong's research[7], among all the available index compressing methods, PForDelta can provide the best trade off between index size and encoding/decoding speed and therefore is the best approach for index compression. However, when dealing with short postings which include less than 128 elements, PForDelta does not perform well than variable-length solution. Therefore, the possible solution is to adopt both variable-length and PForDelta, with the former used for short postings and the latter for the large postings.
- Currently, the elements stored in postings are ordered by document-id. It performs well on merging postings when there is a multi-keyword query. However, if a posting of a certain term is very long, then all elements in the posting would be processed during a query, it would lead to a bad performance. Many suggestions have indicated that elements should be ordered according to a query-irrelevant ranking, which provide ability to permit inverted files to be pruned and can therefore searching performance. However, this method will lead to a slower merge during a multi-keyword query, and also lower the index construction performance during index merging. Therefore, there should be a trade off A possible improvement is: elements of a posting is divided into blocks, within each of which the elements are still ordered by document-id, while blocks are ranked according to those suggestions above.
- Change the B-Tree Indexer into a B+-tree Indexer. Currently, when proceeding a range search, the B-Tree Indexer will get value one by one, in a B+-tree, it does not need to do so, therefore it is necessary to implement a B+-tree in future. More jacobinically, there should exist a search tree that has taken the access speed difference among CPU, memory and disk into consideration, to fully utilize the different level of caches, which is a cache oblivious solution.

- Current distributed index could be either document-partition or term-partition. Either of them have its drawback. There have already been some new approaches, such as quey-log based partition, within which the query log will be analyzed using a clustering algorithm; or hybrid partition, which combine document-partition and term-partition together. The distributed strategy is also a possible improvement.
- Change the reduce process from Broker to Client, then the Broker would not be the bottleneck and can perform better when support multi-Clients.

References

[1] R. Guo, X. Cheng, H. Xu, and B. Wang. Efficient on-line index maintenance for dynamic text collections by using dynamic balancing tree. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 751–760. ACM New York, NY, USA, 2007.

[2] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[3] N. Lester, J. Zobel, and H. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *Proceedings of the 27th Australasian conference on Computer science-Volume 26*, pages 15–23. Australian Computer Society, Inc. Darlinghurst, Australia, Australia, 2004.

[4] A. Ntoulas and J. Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 191–198. ACM Press New York, NY, USA, 2007.

[5] I. Pyarali, T. Harrison, and D. Jordan. An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events.

[6] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates. ResIn: a combination of results caching and index pruning for high-performance web search engines. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 131–138. ACM New York, NY, USA, 2008.

[7] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. 2008.

[8] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the International Conference of Data Engineering (IEEE ICDE), Atlanta, GA, USA*, 2006.

7 Appendix

Table 1: Implement schedule

	Miles-tone	Start	Finish	In Charge	Description
1	Research on inverted files	2008-04-07	2008-04-18	Yingfeng	Choosing index construction algorithm, make research on how to implement distributed index.
2	Implement IndexManager according to basic understanding	2008-04-11	2008-05-16	Yingfeng	Implementing basic framework of IndexManager that could work at local machine.
3	Adjust implementation to satisfy new requirements	2008-05-16	2008-05-30	Yingfeng	New requirements containing the support of indexing multi-collections , numeric range search, has been added, and the interfaces of IndexManager has been defined. The implementation should be adjusted to suit for the new requirements.
4	Research on numeric search	2008-06-03	2008-06-27	Yingfeng	Choosing scheme to store numeric value and support range query. This research process has been misleaded, because a multi-collection's support has lead to a multi-dimensional problem, although adopting composite key is proven to be a better solution later.
5	Design the network library	2008-06-30	2008-07-22	Yingfeng	Implementing a basic network library that support map/reduce with high availibility and scalability.
6	Implement the general file based B-tree	2008-06-30	2008-07-25	Peisheng	General version of B-tree is used for Sequential DB
7	Implement B-tree that satisfy IndexManager's requirement	2008-08-01	2008-08-07	Peisheng	As to IndexManager, all the operation on B-tree is essentially an updating. Make encapsulation of B-tree API to suit for the IndexManager's interface
8	Finish the implementation of IndexManager	2008-07-24	2008-08-15	Yingfeng	Finish the interfaces that have not been done; Integrate B-tree to IndexManager; Integrate network library to IndexManager to implement the distributed version of IndexManager
9	Debugging of IndexManager	2008-08-18	2008-08-29	Yingfeng	Debug the IndexManager, make it work both locally and distributedly
10	IndexManager's documentation	2008-09-01	2008-09-04	Yingfeng	Add comments to the source codes. Write design manual
11	IndexManager's maintainence	2008-09-05	2008-11-30	Yingfeng	Maintain the IndexManager, fix bugs if they appeared; Improve the performance if necessary; Help engineers of Wisenut to integrate IndexManager into SF1