DistributedSearchDesign

Distributed Search Design

Motivation and Goals

There is a need to support homogeneous indicies that are larger than will fit on a single machine and still provide acceptable latency. Aggregate throughput can also increase if a larger percent of the index can be held in memory.

Goals:

- split an index into multiple pieces and be able to search across those pieces as if it were a single index.
- retain high-availability for queries... the search service should be able to survive single server outages.
- automate index management so clients don't have to determine which index a document belongs to (esp important for overwrites or deletes)

Nice to haves:

- Retain ability to have complex multi-step query handler plugins
- Retain index view consistency in a handler request (i.e. same query executed twice in a single request is guaranteed same results)
- distributed global idf calculations (a component of scoring factoring in the rareness of a term)

Simple Distribution

The normal responses from the standard and dismax query handlers provide almost enough information in order to merge multiple responses from sub-searchers into a single client response.

Document Lists:

- need sort criteria added in order to merge. sort fields may not be stored though... may have to retrieve from FieldCache or by other means.

Highlighting info:

- trivial to merge
- CPU problem... if someone asks for documents 90-100 highlighted, the "merger" would have to request documents 0-100 highlighted (and that's a lot of CPU)

Faceted Browsing info:

- easy to merge faceting info by defined queries
- harder to merge facets defined by "top n terms of field f"
  - One can't simply retrieve the "top n" terms from subsearchers and merge them to get the correct result!
  - One would have to request *all* term counts to get 100% correct counts
  - Could increase the correctness by retrieving deeper in the list (i.e. if user asks for top 10, retrieve top 20 from each sub-searcher to calculate counts of top 10)

Debug info: currently not meant for machine parsing... we could simply include debug info from all sub-searchers

Advantages:

- reuse current handlers with a minimum of changes
- everything backward compatible
- should be least amount of effort

Disadvantages:

- index view consistency limited to a single request (the index might change inbetween two requests), hence no use of internal document ids is permitted
- global idf either impossible, or possibly inconsistent because it requires two RPCs
- can't support custom query handlers
- uses more network bandwidth on average... instead of just returning doc ids and sort criteria on the first phase, all requested fields must be returned for every document in the return range.
- potentially lower scalability (due to increased network bandwidth)
- potentially higher CPU load due to parsing sub-responses (depends on transfer syntax)
- higher CPU load when highlighting since all docs in potential range would need to be highlighted.

Merge XML responses w/o Schema

Create an external service to make HTTP requests to sub-searchers and simply combine the current XML responses. All operations operate on the XML alone, w/o reliance on the Solr schema.

Notes:

- returning sort criteria with the document would be slightly harder, but would more easily allow for streaming.

- slight problem: the strings that Solr uses to represent integers and floats in a sortable/rangeable representation are *not* text and XML isn't capable of representing all unicode code points. Solution: use the external form of the sort keys and don't do string sorts... sort based on the XML type <int> <float> <str> <bool>, etc

Advantages:

- Possible to stream (write response as sub-responses are streamed)
  - only important if requesting very large data sets
  - sort criteria would need to be bundled with each document to enable effective streaming

Disadvantages:

- Wouldn't support custom sorters
- Wouldn't support sort-missing-first, sort-missing-last
  - but perhaps that could be passed back as part of sort criteria
- Wouldn't support (easily) different output formats w/o a re-implementation of the other response writers (like JSON)

Merge responses with Schema awareness

Have a query handler that makes HTTP requests to sub-searchers and merges responses.

Advantages:

- can more easily support other output formats... sub-response is de-serialized and the correct writer is used
- more flexible w.r.t. sub-response formats... if JSON format is smaller and faster to parse, it may be used.

Disadvantages:

- streaming not really possible (or not w/o great effort)

Stateless request handlers

Optional: Have request handlers and APIs that don't use docids, and don't require query consistency. There may not be as much value in this option... If we need custom query handlers, complex distribution that allows index consistency and docids would probably be a better choice.

Complex Distribution

The big distinction here is index consistency... a single request handler would be guaranteed that the view of the index does not change during a single request (just like non-distributed Solr).

If the index doesn't change then:

- internal lucene docids may be used in APIs (returned to the query handler, etc)
- total consistency for multi-step requests
- total consistency for any global idf calculations (a multi-step process)
- possibly better on bandwidth (all document into need not be sent back from each segment)

Follow the basic Lucene design for MultiSearcher RemoteSearcher as a template.

- SolrSearchable as an interface that contains a subset of functionality common to all types of SolrSearchers
- SolrIndexSearcher would implement SolrSearchable
- SolrMultiSearcher would implement SolrSearchable via multiple SolrSearchers, implementing the logic of combining search results from multiple subsearchers. The implementation should be network friendly (no HitCollectors, avoid passing around DocSets BitSets, etc).

Areas that will need change:

- Solr's caches don't contain enough info to merge search results from subsearchers
  - could subclass DocList and add sort info, and cache that
  - could dynamically add the sort info if requested via the FieldCache... this would make Solr's result cache smaller.
  - might want to re-use FieldDocSortedHitQueue, which means returning TopFieldDocs, or creating them on the fly from a DocList w/ field info
- SolrQueryRequest currently returns a Searcher... it would beed to be a MultiSearcher

Optional:

- Support for plugins at the sub-searcher level? Normally, custom handlers would only be invoked at the super-searcher level... but if there are any operations that require low level IndexReader access to perform efficiently (and we don't provide an API), it would be nice to allow a super-searcher handler to invoke a sub-searcher handler. User code would be invoked to merge results from all the sub-searchers.

Transport Syntax:

If custom query handlers are to be allowed with Distributed Search, there is a problem of how to represent Queries. The Lucene MultiSearcher distributes Weights to sub-searchers... this was done for global-idf reasons... the weights are calculated in the super-searcher and can thus use doc counts from all sub-searchers.

- Use query strings (Lucene QueryParser)
    - Query strings need to be reparsed by every sub-searcher, increasing CPU
    - Custom query handlers would be limited to using query strings to remote APIs as the process isn't reversible (can't go from Query to QueryString)
    - Attempt to provide schema aware re-parsable Query.toString() support for all query classes we can think of (but there could always be others we don't know about)
    - many query types aren't representable in QueryParser syntax
- Use serialized Query objects (binary)
    - re-use built-in support already there for serializing objects
    - Solr's caches are Query based, so it would be better to use Query rather than Weight to pass to subsearchers
    - Global IDF can still be done w/o passing Weights... global term doc counts can be passed with the request and the subsearchers should be able to weight accordingly.
    - serialized Query objects are binary... this means the transport syntax would be RMI, or at least HTTP with a binary body.
- Use serialized Query objects (text)
    - Use some other human readable representation of queries... good for debugging, bad for support of unknown queries.
    - could perhaps wrap binary serialization (base64) for query types we don't know about.

Network Transports:

- RMI
    - If distributed garbage collection is needed for maintaining a consistent view, RMI already does this
    - Are connections too sticky? Will the same client always go to the same server if we are going through a load balancer?
        - for the most control, implement load balancing ourselves
- XML/HTTP
    - easier to debug, easier to load-balance?
    - requires a lot of marshalling code

Need new style APIs geared toward faceted browsing for a distributed solution (avoid instantiating DocSets... pass around symbolic sets and set operations)

Consistency via Retry

Maintaining a consistent view of the index can be difficult. A different method would return an index version with any request to a sub-index. If that version ever changed during the course of a request, the entire request could be restarted.

Advantages:

- no distributed garbage collection required... always use the latest one.

Disadvantages:

- scalability not as good... if different index parts are committing frequently at different times, the retry rate goes up as the number of sub indicies increases.
- no custom query handlers

Consistency via Specifying Index Version

Every request returns an index version that created it. On any request, one may specify the version of the index they want to serve the request. An older IndexSearcher may be made available for some time after a new IndexSearcher is in place to service requests that started with it. Every request to a particular version of a searcher could extend the "lease" for another 5 seconds (configurable, or even per-request).

- if load balanced, a different server offering the same sub-index may be hit and go backward in time (it may not have switched to a new version yet).
    - implementing our own load-balancing would solve this... stick to the same server for the duration of a request.
        - how would a server be taken out of service??? need a mechanism that is visible from the real load-balancers and our software load-balancer.
    - making all servers switch to a new index snapshot at the same time would also solve this
        - complicated logic (what if one server is unreachable, then becomes reachable again, etc)
        - one slow server could hold up all others
- if streaming the document list (retrieving the stored fields for each doc as it is sent), it would be possible (though very unlikely) to hit an error in the middle of a response. If it did happen it would most likely be due to:
    - a long gc... longer than the lease period between retrieving documents
    - an upstream client not reading as we write, causing us to block

Approach:

- no custom query handlers
- make changes to normal request handlers to allow returning internal ids only
- make changes to normal request handlers to return index version
- make changes to normal request handlers to be able to operate on a list of internal ids (do highlighting, retrieve stored fields, only for listed ids).
- make either binary response writer, or RMI
- make a super request handler that knows about sub-searchers, and can dispatch requests, combine responses, make further requests for highlighting & stored fields, combine everything into another SolrQueryResponse for writing.

Q: how would request handler get a particular version of a SolrIndexSearcher? One is already bound to a SolrQueryRequest (the newest), but if an older one is requested should that logic be built into SolrCore? Handling it in SolrCore would be both cleaner for request handlers, but it would complicate SolrCo re too.

Multi-phased approach, allowing for inconsistency

Do a mulit-phased approach (separate query phase from stored field retrievial and document highlighting), but communicate using the uniqueKey fields rather than internal docids.

This is simpler, but opens a window of inconsistency because the index could always change between phases. This level of inconsistency may be acceptable given that there is already inconsistency caused by clients paging through results.

Downsides of using uniqueKeys instead of lucene docids:

- doing internal_id->uniqueKey is either expensive, or requires a lot of memory (FieldCache entry)
  - could mitigate in the future with payloads, or up-and-coming fields-stored-separately
- paging deeper into results will require more network bandwidth since uniqueKeys could be large

High Availability

How can High Availability be obtained on the query side?

- sub-searchers could be identified by VIPs (top-level-searcher would go through a load-balancer to access sub-searchers).
- could do it in code via HASolrMultiSearcher that takes a list of sub-servers for each index slice.
  - would need to implement failover... including not retrying a failed server for a certain amount of time (after a certain number of failures) * offers advantages for complex distribution... one can stick to a particular server for better caching effects

Master

How should the collection be updated? It would be complex for the client to partition the data themselves, since they would have to ensure that a particular document always went to the same server. Although user partitioning should be possible, there should be an easier default.

Single Master

A single master could partition the data into multiple local indicies and subsearchers would only pull the local index they are configured to have.

- hash based on unique key field to get target index
- commit should be changed so nothing is done if a particular sub-index hasn't been changed

Directory structure for indicies:
Current: solr/data/index OptionA: solr/data/index0, solr/data/index1, solr/data/index2, OptionB: solr/data/index, solr/data2/index, solr/data3/index, OptionC: solr/data/index, another_solr_home/data/index, yet_another_solr_home/data/index
Option (C) mimics having multiple masters

Disadvantages:

- scalability limits for index distribution at some point... we are limited by the outbound network
  bandwidth of a single box.
  - for better bandwidth usage for incremental updates, we could consider alternate update methods...
    add all documents to a separate index and distribute separately... then merge into main index.
    This would also require changes to Lucene to not optimize at the start and end of a segment.

Multiple Masters

There could be a master for each slice of the index. An external module could provide the update interface and forward the request to the correct master based on the unique key field.

The directory structure for indicies would not have to change since each master would have it's own solr home and hence it's own index directory.

Advantages:

- better network scalability for index distribution

Disadvantages:

- complexity of managing multiple masters

Commits

How to synchronize commits across subsearchers and top-level-searchers? This is probably only needed if one is trying to present a SolrSearcher java interface with a consistent index view.

Misc

- Any realistic way to use Hadoop?
    - probably not... map-reduce is more for long running batch jobs (data mining, index building, log processing, etc)
- Multi-core seems to be a lasting trend.
    - x86: Dual cores are now standard, 4 cores/chip are right around the corner (2006 end)
    - parallelize certain request portions to lower latency...
        - faceted browsing set intersections
        - parallel multisearcher over multiple index segments
    -

Conclusions

What won't work

- load balanced RMI (that I can find)
- Simple Distributed Search: After analysis of search patterns (for a particular application needing distributed search), simple distributed search is not an option because of the depth (topN) of the searches. The solution would quickly become network-bound from the large responses from sub-searchers, and IO bound from reading all of the stored fields of the documents.

Current approach

"Multi-phased approach, allowing for inconsistency" is what is being first used for the query side of https://issues.apache.org/jira/browse/SOLR-303 . Distributing the indexing will be up to users via a Multiple Master approach. In the future, we may want to migrate to "Consistency via Specifying Index Version" and lucene internal docids.

The query is executed in phases. In each phase a request is sent to relevant shards in a separate thread. After all the responses are received for all requests the next phase is executed.

Phase 1: GET_TOP_IDS [& GET_FACETS]

Each shard is requested for the top matching document's unique keys and sort fields with facets for the given query. The number of keys requested in this phase is 'N' (start=0&rows=N) regardless of the start specified, so that the results can be correctly merged together.

The response gets the unique keys for each document and their scores. If GET_FACETS is requested it returns the top 'N' facets. n=facet.count. After the responses are obtained they are merged and sorted by the rank. From the sorted list the documents to be returned are identified on the basis of 'start' and 'rows' parameter.

Phase 2

Request are sent to fetch fields, highlighting and MoreLikeThis information only for the documents identified in Phase 1. The request contains the document unique keys and is sent to only the relevant shard which has the document.

Phase 3: REFINE_FACETS (only for faceted search)

The original returned facets may have insufficient information. So more requests are sent to shards for refining facets. Note that the approach applied here gives accurate counts but theoretically, it is possible to miss some facet terms.

After the document fields and facets are obtained the response is constructed and sent back to client.

It is possible that during the small window of time (from phase 1-3) the index may change. In that case the responses may have incorrect data. That is ignored for the time-being.