A Technical Discussion about Search Solutions

By Alejandro Perez, Software Architect, Flaptor, Inc.

In this paper we analyze several important aspects involved in defining search requirements, and discuss the approach we took with IndexTank in comparison with some other popular approaches.

Real-time Updates

The time it takes for an indexed document to show up in the search results can be an issue for certain projects. While some applications have very relaxed time requirements (e.g. a library index), and can work well with delays of several hours, other apps require very fast updates in order to be useful. If, for example, the search is used to help users avoid posting duplicate content, the freshness of search results is essential.

Most current search solutions are based on an index-push-search model in which the data is only available to be searched in discrete packets after being indexed. This model is a natural choice for several reasons: it allows an easy separation of tasks between different machines (indexer and searcher); it provides a neat point in time to synchronize (the push) and a data quantum to transmit (the data indexed between pushes); and it provides an opportunity to perform several optimizations given the fact that a group of documents can be processed in batch before they have to be searchable. However, it also creates a bottleneck while propagating data from one machine to the other, degrading index performance and freshness when frequent updates are involved.

The freshness vs. performance of index-push-search systems is a trade-off that needs to be addressed. Applications with small enough corpora and/or very few new documents being indexed can simulate quite well a real-time search system. It should be noted, however, that for most applications there needs to be a good control of push frequency in order to balance the freshness of the results and the incurred overhead created by frequent small pushes. This generally means that a custom push schedule has to be developed for the particular application.

When designing IndexTank, our years of experience with Lucene and other search engines based on the same paradigm taught us to steer away from this model and merge indexer and searcher in a single unit running on the same machine. The problem here is to maintain acceptable search performance regardless of the indexing load. In order to provide real-time search results without a significant overhead in performance/resources, we used two complementary techniques:

1. Our data model separates the document data into two parts: a rarely changed part (the fields) that is very compressed and slower to update, and a more dynamic part (the variables) to store the mutable data, that is slightly less space-efficient but very easy to update. This benefits applications where textual data is rarely changed (i.e. the description of an article), yet numeric data is changed much more frequently (i.e. price, number in stock, number of votes, etc.).

2. We store the new documents in an efficient in-ram index that is immediately searchable, while we buffer a large block of documents to create a large and highly compressible segment for a second index.

This model provides search results that expose changes within a few milliseconds, and yet performs as efficiently as the index-push-search approach. Implementing this strategy on top of existing search libraries such as Lucene turns out to be a very difficult task, because they are not designed to work this way.

The model also makes it cheaper for the customer to perform partial updates. With solutions like Lucene -where indexing the complete document is required to update a part of it- the process that sends updates to the indexer has to fetch the full document data whenever any part of it changes. This generates an additional load on the storage system. With IndexTank, if the update affects only the variables, there's no need to fetch the rest of the data.

Perceived relevance

Having a search that returns relevant results is always an elusive goal. The main problem is that different users have different ideas of which results are relevant for a particular domain, and this is exacerbated when they don't understand why the system returned a particular document for a given query. Users can adapt their search strategies to circumvent problems only if they understand what caused them.

Google saw this sooner than the rest, and one of the best features they incorporated to their web searcher was snippets. This single feature turned out to be so useful that now it is an expected standard feature for all search engines.

The problem with highlighting is that a search index is generally not a good structure to recall the original text. This prevents store-less solutions like Sphinx from providing the service without interfacing with an external data store, and in the case of Lucene/Solr, forces enabling the optional storage of the document content, greatly increasing the size of the index.

Since we knew from the beginning that we wanted to provide snippets for IndexTank, we focused on enabling a secondary storage structure, separated from the main index, optimized in size (compressed) and distributed among many servers. This dramatically improves the latency for all documents in a result set since, in practice, we are spreading the requests over many storage units.

Another feature that helps the user learn faster how to effectively use the search feature is a search box with suggestions (autocomplete). IndexTank provides the API for it, and currently it can use the corpus terms or the history of queries to provide the suggestion. Each data source is better suited for different use cases. For indexes with few queries, suggesting from the corpus works better, while in indexes with historic data and trendy topics, recent queries is preferable. Solr currently has this feature as an option provided by a plugin. However it requires considerable set up, and regular maintenance (like regeneration of the suggest index), which is not necessary with IndexTank.

Yet another way to guide the user, when the corpus is categorized, is to show how many results fall within each category. This is called faceting and has been made popular by retail sites. Solr implementation depends heavily on caches, and uses up memory in direct proportion to the number of faceted categories. Caches are also invalidated every time a new index gets pushed, making this approach quite expensive (and slow) for applications with frequent updates. In IndexTank we implemented the faceting algorithm in a way that can be run at the same time with the scoring system with very little overhead. Faceting data is stored along with the variables used for scoring, and the calculation of both is done at the same time. The net result is that faceting adds a negligible amount of time over a normal search.

Flexible scoring system

Defining the required relevance over a particular dataset is often (if not always) an iterative process. And even when we have a clear idea of how we want the documents to be ordered, the users may think otherwise, so it is a good idea to have some support for A/B testing.

Many search solutions allow limited tweaking of the scoring system, and some require regenerating the index from scratch to allow certain changes. For example, Lucene has some support for general functions, but it is not efficient; variables are stored in the index separately so multi-variable functions may hinder performance significantly.

In IndexTank, we allow the user to define multiple scoring functions, and specify which one to use in each query. This allows a developer to try new scoring functions while the users see results scored with the old and tested function. It also makes it very easy to implement an A/B test.

The scoring attributes are stored in memory, and all attributes for a single document are stored together. We also pre-process the scoring functions entered by the user, generating optimized bytecode. All this makes score evaluation very fast and quite insensitive to the number of attributes used in the formula.

User-specific Relevance and Geolocation Support

Scoring formulas can refer to document variables (provided at indexing time or updated thereafter) or query variables (provided at search time), enabling, for example, a scoring formula to provide user-specific relevance. This opens the door to many applications.

One such application is geographical relevance. This can be accomplished simply by adding coordinates to the documents and sending the user's location with the query. Predefined functions that calculate the distance between two geographic points can be incorporated into any scoring formula. This can be used, for instance, to increase the relevance of documents that refer to locations near the user, or to filter out documents that are too far away.

Most other solutions do not provide geolocation support. Lucene requires a third-party plugin that reaches down into the core of the search engine and performs several optimizations to make up for the inefficient way in which Lucene accesses that kind of data, and it comes with several limitations. For example, a document that falls outside of the specified radius but is otherwise a perfect match cannot be selected.

The ability to define arbitrary and very efficient scoring functions is one of IndexTank's strengths. Strictly speaking we did not need to provide distance functions to support geolocation, as it could also be done by using the standard sqrt and pow functions. This highlights the flexibility of our approach (although our distance functions are more precise than the simple planar calculation).

Redundancy

All hardware fails eventually. A service that requires 24/7 availability cannot rely on a single machine for any critical part of it. Unfortunately reliability is expensive for small scale applications as it often requires doubling the hardware, which ends up being mostly idle.

IndexTank's solution to this problem is simple: all the indexed data gets backed up in the cloud, using a RAID-like service. Accessing this data sequentially to regenerate a full index takes only minutes. Also, index structures are checkpointed frequently and synced to a redundant storage system (currently Amazon EBS) in order to accelerate recovery. The recovery process only needs to reindex data received after the last good checkpoint.

Scalability

In the typical life-cycle of a search service the index starts small and grows continuously thereafter. As the index volume grows, hardware requirements do too. Either the server has to be oversized to allow for future demands, or it will eventually need to be replaced by a larger server. Redeploying to a different machine is a complex and delicate task, especially if availability needs to be maintained during the process.

IndexTank needs to perform this feat on several scenarios, not only to move a growing index to a larger server, but to balance the load among servers, to decommission a server, and even to update the IndexTank software. While the problem seems simple, the solution is quite complicated when it has to be accomplished without downtime while keeping index coherence and without loosing a single update.

Indexing and search requests need to be handled by a distributor that routes the requests to the appropriate machines. With this in place, an index is moved as follows:

- A checkpoint of the old index is made so the new index starts as fresh as possible.
- A new index is created on another machine using that checkpoint.
- The distributor starts sending all updates/adds to *both* indexes, while search request are routed only to the old one.
- The new index missed all changes from the checkpoint to the time it started running, so all data from the last checkpoint is be recovered from the storage system.
- After the recovery is complete, both indexes are up to date. The distributor starts routing all requests only to the new index.
- The old index is shut down.

Robustness

The web is a harsh place. We learned the hard way that systems may pass all tests and yet fail within minutes when exposed to the wilderness. Worse yet, it may fail after days or weeks, when developers feel confident that everything is stable and stop paying close attention.

Here at Flaptor we have devoted hundreds of hours to develop different synthetic load tests, in order to assert IndexTank's performance and survivability under different conditions. Even so, we are always learning new ways to bring a system down. We know however that for any search system to survive in the wild there are three main issues that have to be addressed:

- Concurrency has to be controlled. Too little and the throughput is disappointing; too much and latency sky-rockets. But the relation between concurrency and performance (be it throughput or latency) is not simple and shows some hysteresis: as the load gets higher and the system passes its throughput peak, latency increases, the request queue grows and the load increases even further. This sometimes makes recovering from a sudden burst impossible. With some testing we determined IndexTank's sweet spot of concurrency to be only slightly above the number of available processors. This result is consistent with the whole process being cpu-bound (actually the process is memory-bus limited, but bus contentions do not impose such a big overhead as software context switches).

- Scheduling has to adapt to the system's load. When things go wrong scheduling the queued documents becomes important. In the context of a search service, when there is a human on the other side that has limited patience, there is no point in starting execution of a forgotten (old) query. And since users usually hit reload on an unresponsive system, we can increase performance by detecting this situation and avoiding executing the same query several times in close succession, by using the results of the first one to answer all of them.

- Simple queries have to be partially isolated from heavy ones. A few complicated queries can make the simplest queries take too long. To avoid this we need a way to limit the resources used by each query, in order to provide some fairness. We implemented a double trigger that cuts a query short when it exceeds a number of matches and a certain execution time. This allows us to complete complex queries when the server is under light load, while providing full service to simple queries under abnormal service conditions. It's worth to note that when a query execution is cut short, the partial results are returned, providing a reasonable result most of the time.