The search experience challenge

Scaling the search experience in our customer's environment is one of the key areas where our efforts are focussed for going beyond 100 million documents. As a primary system of record on insurance-related documents, the use case of search and retrieval is critical, and we aim to achieve a goal of less than 3 seconds response time for any user query. Besides the increasing scale of data, this goal is being affected by the specific complexities of queries imposed by the user interface and security requirements:

- queries are broad in nature, with potentially extensive large result sets refined only via facets or subsequent detail queries
- facet values are always loaded, including dynamic, time-specific facet queries
- Search results are ordered on commonly used metadata properties, e.g. cm:name
- access rules governing what documents a user can view based on metadata state are enforced in part by enriching queries with extensive sets of filter queries

The combination of broad queries and result ordering on commonly used metadata properties may start to introduce a noticeable overhead once the size of the index no longer permits the terms of the relevant index fields to be reliably held in memory or disk cache. The more extensive the metadata model of the specific scenario, and the more the various queries access and use other metadata properties, the less likely it becomes that these terms are evicted from caches and sorting has to perform more and more costly, fragmented IO reads.

In a similar way, the performance of both facet and filter queries is heavily dependent on caching. All of these queries are by default executed sequentially within the context of the overall user request. Any delay in one of the queries directly contributes to the overall duration, and any frequently used, but potentially expensive query which cannot be found in high-level query caches must be (re-)executed with all of its associated cost. Frequent index updates as part of normal change tracking invalidate large parts of these high-level caches, making successful query reuse significantly less likely while growing amounts of data make each query become more costly on its own over time.

Looking at sharding

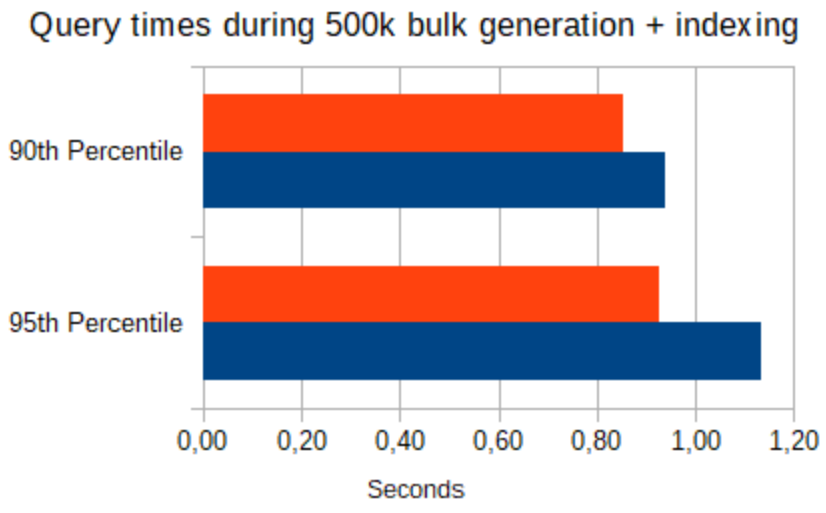
The general recommendation to scale Alfresco Search Services for large amounts of documents is to use sharding to split the index into smaller segments. On its own, sharding does nothing to eliminate or reduce the overall cost of queries – it only splits one large request process into many, potentially concurrent processes on the individual shards. Provided that processing and storage

resources are appropriately scaled with the number of used shards, or shards located on different server machines altogether, index sharding will significantly improve the perceived search performance using this divide-and-conquer approach.

Considering the ever-growing number of documents stored in our customer's system and how frequent index updates invalidate large parts of caches, we have been focusing part of our scaling tests on the sharding method DB_ID_RANGE. This sharding method allows new shards to be added when necessary without requiring a full re-index of all the existing shards and also offers the possibility of explicitly sizing the individual shards based on the configured ranges of node database IDs. In our tests, we were able to demonstrate the feasibility of using extremely small shards (~10 million nodes) for the most recently added and thus most frequently updated documents, in order to limit the worst-case performance and overheads of cache misses, while significantly larger shards (~25 or more million nodes) handle „old" documents, with extremely infrequent (if ever) cache invalidations due to index updates.

Using those small shards for the most recent documents also allowed us to gather valuable information about the process of setting these shards up, especially the cost and duration of the initial index build. This is not only relevant for the addition of new shards but also – at some point – the consolidation of existing smaller shards into large ones when the documents contained therein transition from the „currently used / relevant " to „primarily archive" lifecycle. As a result of these repeated shard builds, we also identified potential improvements in the indexing process, which we shared with Alfresco engineers on the Search Services team to help improve the overall product in the long run.

The following graph compares the query times of the DBID (blue) and DBID range (orange) shard methods as the most closely comparable methods of equally distributed vs. targeted index updates. These measurements were taken from identical system setups under load, specifically while generating and indexing 500.000 new random test documents in a concurrent process. The system with DBID range sharding uses the distribution model described in the previous paragraph. In general, we were able to observe a consistent 10-15% improvement in the 90th and 95th percentile ranges going from the equal index update distribution to the targeted index update model when testing with a random sample out of a set of ~2040 queries, encompassing the majority of currently known access rule combinations.



Cache pre-warming

While sharding allows us to deal with the cost overhead of broad queries and result ordering by splitting the index and thus enabling each shard to hold more or even the entirety of the sort-relevant terms in memory or disk cache, it does not by itself improve the challenge of the number of facet and filter queries, and their caching. Despite splitting the overall request into concurrent executions on each shard, each shard still sequentially processes each query and thus can introduce delay if some of the potentially expensive facet and filter queries need to be re-executed if not already cached. This is especially noticeable on the first execution of requests after a fresh server start as well as any situation where all relevant queries have been purged from the cache, e.g. having been displayed by other queries.

In the scenario of our customer, the access rules alone generate around 60 filter queries for each request and we have identified a total of around 160 common facet and filter queries. This means that in a worst-case scenario, a user may have to wait on ~60 sequentially processed queries until results are returned. Even if all queries only take 100 to 500 ms on average on a shard, this still amounts to an estimated 6 to 30 seconds of wait time on the Alfresco Search Services tier, not counting any additional processing or loading from the database on the part of ACS before returning the results to the client / UI.

While Alfresco Search Services by default supports carrying over a subset of cache entries between index updates, it provides no means to pre-load any specific cache entries or ensure that these entries are picked when carrying over after an update. We, therefore, developed our own little extension to pre-load the common / most often used facet and filter queries that we had identified, and thus ensure that users are highly likely to find the majority of their request pre-cached. By decoupling the cache loading from the regular request and cache entry transfer handling, we were also able to use concurrent processing to better utilize available processing power and speed up cache initialization.

Combined with our sharding test setups, including the one where only the smaller shards for the „current documents“ perform index updates, and tooling to bulk generate millions of additional document records, we were able to demonstrate that even under active load and frequent index updates, cache pre-warming adds the only minimal overhead of around 50 milliseconds. These tests also showed that worst-case performance with all pre-warming (query + sorting) enabled barely exceeded 2 seconds per query (Alfresco Search Service and ACS layers combined) and typically kept within 350 milliseconds (95th percentile) when testing with a random sample out of a set of ~2040 queries, encompassing the majority of currently known access rule combinations.

Testing for “first query performance” after a complete restart of the Alfresco Search Services layer in an otherwise idle system showed that – with all pre-warming enabled – the worst-case performance of a random query from the same set of ~2040 queries did not exceed 1 second and the first 20 sequentially executed queries generally kept within 0.55 seconds (95th percentile), compared to a worst-case of 70 seconds without any pre-warming and 24.6 seconds (95th percentile) for most of the first 20 queries. The relative improvement of the “first query

performance" with full pre-warming enabled comes at a cost of a 60 seconds delay in SOLR startup, which – when included in a running sum of the times for the first 20 queries – still leads to a favorable result of 66 to 119 seconds total query time of full pre-warming compared to no pre-warming.

The following graphs summarise the measurements mentioned above, taken from a scenario with 6 shard setup using DBID sharding, using no warmup (blue), only filter/facet query pre-warming (orange) and full pre-warming (yellow).