**Scopus FAST to Solr Migration**

**Hothouse PoC Report**

**May 8, 2014**

**Version 0.9**

**Elsevier Architecture**

**Curt Kohler**

**Darin McBeath**

1 Solution Description 3

1.1 Document Indexing Control System 4

1.2 Search Index and Document Schema 4

1.3 Solr cluster configuration(s) 5

1.4 Content Indexing program 6

1.5 AWS Kinesis/Redshift 6

1.6 Load Driver Program 7

1.7 Architecture Diagram of POC 7

2 Affiliation Testing Results 8

2.1 Affiliation Content Loading 8

2.2 Affiliation Query Load Testing 10

3 Author Testing Results 13

3.1 Author Content Loading 13

3.2 Query Load Testing 15

4 Core Testing Results 18

4.1 Core Content Loading 18

4.2 Core Query Load Testing 21

5 Potential Problematic/High Cost Features 24

6 Answer Features/Count Results 26

7 Scopus Specific Challenges Investigated in PoC 28

7.1 Introduction 28

7.2 Large, Accurate Navigators/Facets 29

7.3 High Update Volumes 30

7.4 Corpus Wide Statistics 31

7.5 Index Flipping 32

8 Operational Challenges 33

8.1 Integration with Existing XFAB Job Control System 33

8.2 Storage vs. Syndication format 33

8.3 Limited PoC Development with AWS Kinesis 34

8.4 Management of the Flip 34

8.5 Lack of “Rack-Awareness” in Solr Cloud 34

8.6 Resource Availability 34

8.7 CIP Mappings/Crispness of Product Functionality Requirements 35

9 Deployment Architecture/Cost 35

9.1 Proposed Architecture 35

9.2 Proposed Operating Costs 39

9.2.1 Scopus Affiliations 39

9.2.2 Scopus Authors 40

9.2.3 Scopus Core Abstracts 40

10 Pros/Cons 40

10.1 Pros of Solr 40

10.2 Cons of Solr 40

11 Known Limitations 41

12 Additional Thoughts 41

12.1 Minimize Document Size Sent to Solr for Indexing 41

12.2 Increasing Indexing Throughput 41

12.3 Investigate Virtual Private Network – Increase in network performance - Still need to write 41

12.4 Data cleanup 41

12.5 Resolve Facets in Parallel 42

12.6 XML Handling 42

12.7 Filter Cache 42

12.8 Relevancy 43

12.9 Directory Factory 43

12.10 Leverage Analyzers/Filters 43

12.11 Preload index into system cache 43

12.12 Garbage Collection 44

12.13 Highlighting 44

12.14 Scope Queries 44

13 Evaluation Matrix 45

# Solution Description

The Scopus PoC is an add-on effort to the ScienceDirect FAST migration PoC that was conducted in the second half of 2013. Specifically, the development teams determined that the Scopus product/datasets exhibited some distinct attributes that were not adequately investigated as part of the ScienceDirect PoC. The most significant of these attributes are: more distinct documents with a higher update volume, numerous high cardinality facets with exact counts, the need to incorporate accurate corpus-wide statistics, and more complex scoped queries. Since many of the Scopus requirements had been proved out in the ScienceDirect Hothouse, they were not repeated here and we focused on those key functional differentiators as well as ensuring performance would meet the defined targets. All development/sizing was based on the Elsevier Scopus Search Prototype Version 1.0 (March 17,2014) document. If these requirements change, this must be taken into consideration if the Solr approach is adopted.

One major design decision the team decided on early was to perform updates offline to the active index being searched. This decision was made to remove the impact of high update volumes adversely impacting the query performance as well as providing a solution to guarantee accurate corpus-wide statistics (e.g. publication and citedby counts). This solution dictates that updates to the index will only be released at predetermined intervals and implies that updates will not be immediately viewable on the system. Currently the team has proposed a daily promotion of the index (although it is possible it could happen more frequently depending on the amount of time needed to complete all processing work). We will discuss the proposed solution for this approach in more depth later in the document.

For the Solr PoC we leveraged release 4.6.0 of the software using Solr Cloud, Oracle’s Java version 1.7\_17, and ran all of our componentry in the Amazon AWS cloud. Specifically we leveraged Amazon SQS for workflow management, S3 as a content repository, DynamoDB as an inventory control database, Redshift to calculate corpus wide statistics present in Scopus, and Kinesis to populate data into Redshift as individual documents are updated.

## Document Indexing Control System

We leveraged Amazon SQS (Simple Queue Service) as a simple workflow management system. We add individual entries onto the content type specific queue for each document that needs to be processed in addition to the action to take (add, delete, update), and timestamp from the XFab system to ensure the order of changes to the index. The Document Indexing Program (detailed later in this section) reads these messages in order to process a document.

We also leveraged Amazon’s DynamoDB as an inventory control master for our solution. By using this repository, we can perform synchronization checks against the associated S3 content repository and XFab system that has provided to content to process in order to make sure we have processed all the changes to date.

## Search Index and Document Schema

For the PoC, we manually crafted an index schema based upon the SC13-3 V52 Scopus FAST CIP for each of the three content types. We also defined a number of analyzers (tokenizers plus filters) to represent the various permutations needed to support the stated requirements. Due to the variety of tokenization strategies needed to support things like punctuation sensitivity, stop word processing, faceting, etc. we were forced to define alternate “shadow” fields for many of the large textual fields in the CIP (similarly to what FAST needs to do today). These shadow fields have the side effect of drastically increasing the size of the index on disk (e.g. 2 shadow fields for different phrase matching for a given document field adds 2X to the required index storage for that field). Additionally, we had to define group fields to map to the concatenated fields generated by FAST (e.g. all, allmed, auth, etc.) during their indexing of content. As we were using a dump of the Scopus data in “storage format”, we did not have all of the merged data fields present in the documents (e.g. Affiliation fields in author profiles). In some cases, we were able to find the data from other fields in the document and in others we were unable to populate those fields. This implies that some of the queries captured from the Scopus logs will not find results when run. Finally, there were a number of fields that were generated by FAST as part of their indexing that were not present in the ingested content and we did not have the ability to reproduce. In those cases, we provided a standard default value when possible. When all was done, we had defined an index profile of approximately 160 fields (A number of fields were not present in our mappings as they are created during the FAST indexing pipeline and were not available in the XML).

## Solr cluster configuration(s)

For the PoC, we actually had three separate Solr Coud clusters, one for each content type. This was intentional, as it allows us to target each content type to the appropriate sized hardware based on index size as well as query throughput/response time requirements. In discussions with external Solr consultants that were engaged as part of the PoC, we determined that Solr is typically RAM throttled. This means that in order to get maximum performance out of a deployment, you want to have as much of the index resident in memory as possible. This was the guiding light as we sized and deployed our test clusters.

Both the affiliation and author indexes were small enough that we were able to run them on smaller, single machines with a single shard per content type. The cores index was significantly larger and required us to configure it for 5 shards and run it on five separate large memory footprint machines. All of the indexes store only the fields that are defined as being returnable in the Scopus CIP so any reloading of the content requires a re-fetch from the S3 buckets and a reprocessing of the entire document. The sharding of documents is done with a simple approach using the individual content types’ primary id.

In addition to sharding, Solr Cloud also supports the replication of shards. A replica is a complete copy of a shard. To increase query capacity, as well as provide for HA deployments, you can increase the replication factor for an index and add additional processing nodes into the cluster. Unlike sharding, the replication factor for an index can be adjusted in real time. The clusters we created for the PoC cluster had no replicas configured. This implies that losing a single shard in the cluster would cause all transactions against the cluster (updates or queries) to fail failing until the shard was restored. In the case of Scopus, a high availability (HA) solution can be provided by replicating the entire solution (indexing and search clusters) into an additional availability zone. SolrCloud transparently manages the various replicas and failover within the cluster.

In addition to the actual data nodes in the cluster, we also have a collection of three Zookeeper nodes that are responsible for cluster management. They take care of making sure that Solr data nodes are still alive, organize the migration and recovery of primary and replica shards as needed, as well as other housekeeping duties. For the PoC, our test driver processes were hardcoded with the IP addresses of the individual shards in a cluster and round robin-ed requests between them. For the true production deployment, we would recommend have our Solr clients configured to leverage the Zookeeper nodes when determining what Solr nodes are available to use to satisfy requests. The ZooKeeper cluster requires that a quorum of available nodes is available for the cluster to function properly. This implies the need for an odd number of ZooKeeper, deployed across a number of AWS Availability Zones such that losing one availability zone will not preclude having a quorum of ZooKeeper nodes available. In our case, we recommend 3 ZooKeeper Nodes spread across 3 availability zones in one AWS region.

## Content Indexing program

As part of the PoC, we developed a custom Java program for each content type to drive the indexing of documents in Solr. This program performs the following tasks for each record it processes:

* Read an entry from the SQS queue
* Retrieve the associated document from S3 and copy it to local disk (to avoid Amazon S3 I/O errors encountered while attempting to process the content as a stream from S3).
* Apply a large collection of XPath expressions to extract the textual content from our documents (as defined by the CIP for that content type) and convert it into a JSON representation that will be indexed. This includes massaging the content as needed to support specific functionality (e.g. scoped searching, group fields, etc.).
* Submit JSON document to Solr cluster for indexing
* Update DyanmoDB inventory control for that document with the new metadata information.
* For Core content only: extract out the associated author and affiliation ids as well as the ids for documents being cited by that paper. This information is then sent to AWS Redshift via AWS Kinesis for later batch processing to generate the corpus wide statistic counts to be merged back into all three content types’ indices.

For the PoC we developed three of these indexing programs (affiliation, author, cores). Additional content types such as patents, dummies, etc. would require additional development to create their custom implementation. As the overall task of processing documents for indexing can be highly parallelized, we utilize the same content type specific program to do ongoing updates as well as bulk loads of content. We merely start up the appropriate number of boxes, each running multiple copies of this program.

## AWS Kinesis/Redshift

As previously mentioned, when core records are indexed, the indexing program will extract key information from the record that will be fed into the calculations of author and affiliation publication counts as well as citedby statistics. As a single document can spawn changes in multiple other records in the system and multiple documents can be processed in parallel, it is not possible to perform real time updates to these calculated statistics and guarantee their accuracy at a given point in time. A “batch” calculation approach against a snapshot of all the data provides a solution to this problem. We are leveraging AWS Redshift to provide the “batch” calculation environment. Redshift is Amazon’s large-scale data warehouse offering. In our case, it essentially has a large datastore that contains all of the publication (author and affiliation) relationships as well as the cited-by relationships. As documents are processed, changes are posted to the datastore so it will eventually become consistent before the calculations are performed. After the calculations are performed, the results are exported out as flat files to AWS S3, and then promoted to the individual Solr services as external data files where the values can be used in queries, sorting, etc. without requiring an update to the underlying document.

As Redshift is not designed to accommodate the high update load from all of our generated changes, we are leveraging AWS Kinesis. Kinesis is a managed service for real-time processing of streaming data at massive scale. The core indexer posts the metadata into Kinesis where they are grouped together in a batch and updated into Redshift in a single update.

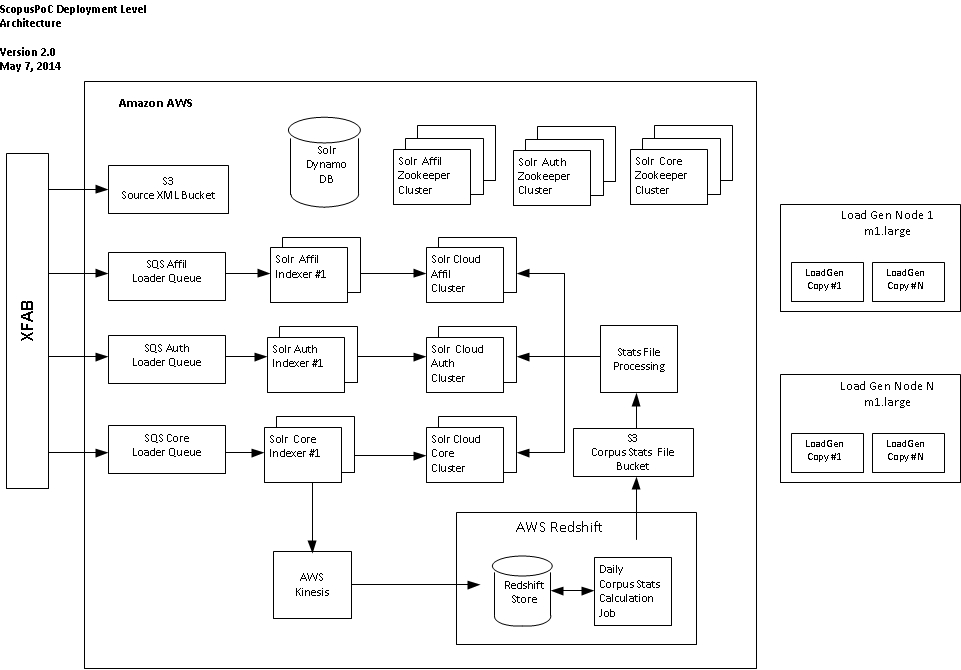
TODO: ONE VERSION OF TRUTH STUFF HERE??? Refer to CTO appendix

## Load Driver Program

We use a standard load driver script that was written as part of the hothouse effort to submit the proper mix of queries into the Solr Search cluster for a specific content type. Specifically, we take the load test query set and process them through a XSL stylesheet to transform them into Solr specific syntax. These Solr specific queries are stored in Dynamo DB with a key that relates them to the original query set. In order to generate a query mix to run, we create a list of all the DynamoDB keys and randomly shuffle the entries (the correct percentages of each type of query is maintained). The load test script then submits POST requests to a small Node.js application that retrieves the preformatted query and submits it to the Solr REST endpoint for processing. This node app captures performance information and logs it to a file where it is collected and transmitted to a Kibana installation where we can monitor the results and generate reports. The times collected are the time as measured from submitting the request to Solr and when the node program receives the complete response. For the PoC, the load driver programs run in the same availability zone as the Solr cluster.

## Architecture Diagram of POC

Figure 1: Solr PoC High Level Architecture



# Affiliation Testing Results

## Affiliation Content Loading

We were able to complete the initial bulk load of 8,163,349 records in approximately 2 hours and 30 minutes. We should note that due to current Scopus CIP restrictions, only a percentage of the records are actually searchable in the product (essentially only top-level affiliations). As a result we dropped those records that were not top-level affiliations during processing which resulted in only approximately 3.5 million records ending up in the index. The dropping of the records had a significant impact on our throughput levels compared to the other content types. We observed less than 50% CPU utilization during the initial load and update runs.

While loading, Solr was configured to use a maximum of 8 indexing threads as well as a modified ramBuffer from the default. Specifically, the ramBufferSize was set at 1536MB with a maxBufferedDocs of 100,000. These settings control how much processing Solr will do in memory before committing to disk in the transaction log and represents the amount of work that can be lost if a machine goes down. Finally, the autoCommit maxTime was set to a max time of 10 minutes. This controls how frequently transaction log records are merged into the Solr index segments. Reducing the log size reduces the recovery time needed to replay the log entries against the checkpointed index should a machine go down. Finally, at the end of indexing, we optimize the index to reduce the size and increase the search performance characteristics. We do this as we anticipate flipping indexes at which point the newly optimized index will become read only and only used for query resolution.

Machine/Solr details as run in the tests:

* AWS m1.xlarge
  + 4 vCPUs
  + 15GB RAM
  + 4 x 240GB Ephemeral Drives
  + High Network performance
    - Note: attempts to use a m1.large instance with a “Good” Network performance profile resulted in significant increase in measured time from the test drivers relative to the Solr reported query time. Upgrading to an AWS m1.xlarge removed that delay without changing the reported Solr response time.
* Solr Config
  + -XX:+UseParallelGC
  + –Xms4G
  + –Xmx4G
  + 2 x 240GB drives in raid 0 array.
* Index
  + Approx. 1GB

**Notes:**

* We did not actually compute the corpus wide statistics for the test dataset. Records were loaded with dummy values for the publicationCount field that encompass the corpus wide statistics for affiliations.
* In further research, we believe that we can allocate significantly less RAM to the Java heap than we did in the PoC. This was not an issue in the PoC, as there was much more RAM available than actually needed during the tests.

Table 1: Affiliation Bulk Load statistics

|  |  |
| --- | --- |
| Queue Size | 8,163,347 (2 records dropped during queue population) |
| Solr cluster | 1 x m1.xlarge instance |
| Number of indexer processes | 5 x cc1.4xlarge instances (20 indexer processes/instance) |
| Dynamo DB Read/Write rate | 1500/1500 |
| Processing Time | 2:26 |
| Index Rate | approx. 3,265,000/hr |
| Pre-optimize index size | 1.15 GB |
| Optimize time | 0:02 |
| Post optimize index size | 1.07GB |
| Total processing time | 2:28 |

Table 2: Affiliation 1x Update Load Statistics

|  |  |
| --- | --- |
| Queue Size | 250,000 |
| Solr cluster | 1 x m1.xlarge instance |
| Number of indexer processes | 5 x m1. xlarge instances (5 indexer processes/instance) |
| Dynamo DB Read/Write rate | 300/300 |
| Processing Time | 0:36 |
| Index Rate | approx. 500,000/hr (note indexers warming up during this run) |
| Optimize time | 0:02 |
| Total processing time | 0:38 |

Table 3: Affiliation 3x Update Load Statistics

|  |  |
| --- | --- |
| Queue Size | 750,000 |
| Solr cluster | 1 x m1.xlarge instance |
| Number of indexer processes | 5 x m1. xlarge instances (5 indexer processes/instance) |
| Dynamo DB Read/Write rate | 400/400 (increased from 1X run. Noticed some Dynamo DB capacity warning from previous run). |
| Processing Time | 1:05 |
| Index Rate | approx. 750,000/hr |
| Optimize time | 0:02 |
| Total processing time | 1:07 |

## Affiliation Query Load Testing

For the full query set, we had an overall target of an average rate of 4 qps with no average overall response time. In addition we had specific average response times for the two identified query types (navigators and no navigators). We ran one load test box with one copy of the load test script. The script ran for 15 minutes to warm the cluster before official measurements were taken. We were able to far exceed the target Scopus requirements, reaching 27 QPS on average. In fact, we exhausted our test query set in under the hour allocated for the test. In addition, Solr was able to surpass all of the existing specific query type measurements/requirements.

The single Solr server was running on one m1.xlarge instance. The Server was configured to use at most 4GB of the 15 available on the instance. This amount of RAM can easily contain the entire index in memory to get optimal performance from Solr. In addition to being memory sensitive, the Solr deployment is also sensitive to the Network I/O available. We had tried running on an m1.large instance type but were seeing a wide discrepancy between the request times measured at the load test box and those reported by Solr itself. We attributed this to the limited I/O bandwidth associated with the m1.large instance type and upgraded the hardware for the official run.

The instance itself showed minimal CPU usage during the tests that would seem to indicated significantly more capacity could be extracted from the hardware. One other item to note is that we had no replicas configured in the cluster to simulate an HA deployment. Adding a second instance would effectively double the query capacity to at least 54 QPS.

In general, this box is very overpowered for the cluster defined by the requirements. The only reason we are still recommending its use is the impact we observed of having a lower grade network connection. It is possible that a smaller box with less powerful network connectivity could be leveraged at the targeted query rate in the requirements. The PoC team did not have time to test that combination. Another possibility to explore would be combining the affiliation cluster onto the same hardware as the author cluster to save costs (assuming an instance type with adequate RAM for both indexes can be identified). Further research would be needed to identify an appropriately sized instance and to test to ensure there are no operational issues in running them together.

Table 4: Query Mix for Affiliation Test set – Navigator Based

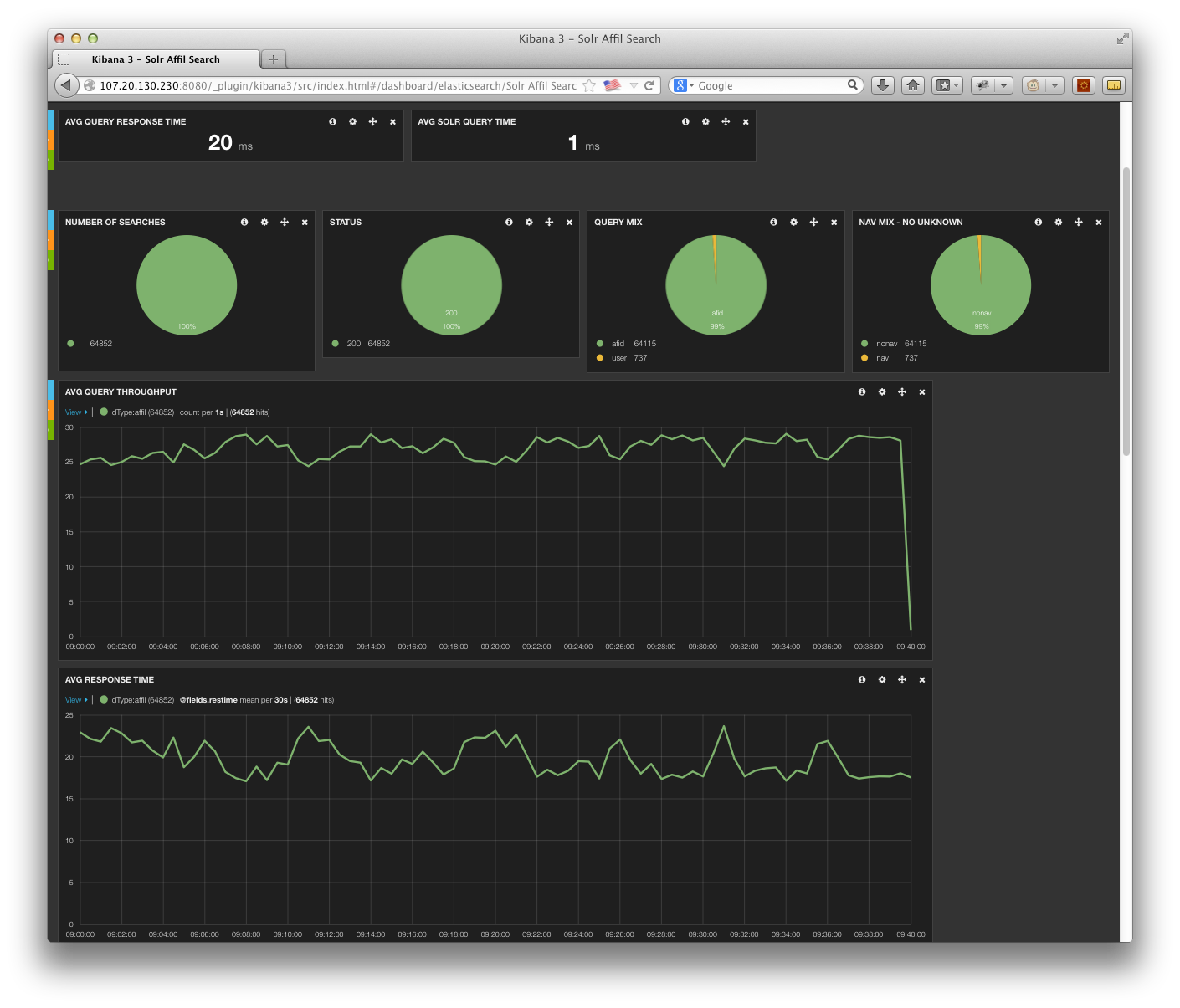
|  |  |  |
| --- | --- | --- |
| **Query Type** | **Number of queries** | **Percentage of mix** |
| Navigator | 737 | 1% |
| No Navigator | 64,115 | 99% |

Table 5: Affiliation full query mix 1 hour overall averages (Note ran out of queries after approx. 45 minutes)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **AVG Request Rate** | **Max**  **Request**  **Rate** | **Avg Response Time** | **Max Response Time** |
|  |  |  |  |  |
| Scopus Targets | 4 qps | 8 qps | N/A | 3,643 ms |
| Solr Observed | 27 qps | 29 qps | 20 ms | 1,024 ms |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **Mean** | **Median** | **P95th** | **Max** |
|  |  |  |  |  |
| Scopus Target - Navigator | 129 ms | 103 ms | 243 ms | 979 ms |
| Solr Observed - Navigator | 21 ms | 20 ms | 30 ms | 131 ms |
|  |  |  |  |  |
| Scopus Target - No Navigator | 95 ms | 70 ms | 180 ms | 3643 ms |
| Solr Observed - No Navigator | 20 ms | 18 ms | 28 ms | 1024 ms |

Figure 2: Affiliation Full Query Mix -1 Hour Snapshot (4/14/2014 9:00 EDT – 9:40 EDT)

****

****

# Author Testing Results

## Author Content Loading

We were able to complete the initial bulk load of 49,705,977 records in approximately 4 hours and 30 minutes. We observed less than 50% CPU utilization during the initial load and update runs. As we ramped up DynamoDB R/W capacities, load performance increased linearly, implying we are not CPU bound on the machine.

While loading, Solr was configured to use a maximum of 8 indexing threads as well as a modified ramBuffer from the default. Specifically, the ramBufferSize was set at 2048MB with a maxBufferedDocs of 100,000. These settings control how much processing Solr will do in memory before committing to disk in the transaction log and represents the amount of work that can be lost if a machine goes down. Finally, the autoCommit maxTime was set to a max time of 10 minutes. This controls how frequently transaction log records are merged into the Solr index segments. Reducing the log size reduces the recovery time needed to replay the log entries against the checkpointed index should a machine go down. Finally, at the end of indexing, we optimize the index to reduce the size and increase the search performance characteristics. We do this as we anticipate flipping indexes at which point the newly optimized index will become read only and only used for query resolution.

Machine/Solr details as run in the tests:

* AWS m3.2xlarge
  + 8 vCPUs
  + 30GB RAM
  + 2 x 80GB Ephemeral SSD Drives
  + High Network performance
  + Note: While we used an AWS m3.2xlarge for the PoC, we recommend using some of the newer Amazon r3 family instances in production. Amazon announced these instances, but they were not yet available during the PoC. They will be a much better match for this index in terms of RAM and disk space.
* Solr Config
  + -XX:+UseParallelGC
  + –Xms20G
  + –Xmx20G
  + 2 x 80GB drives in raid 0 array.
* Index
  + Approx. 22GB

**Notes:**

* We did not actually compute the corpus wide statistics for the test dataset for the official runs. Records were loaded with dummy values for the publicationCount field that encompass the corpus wide statistics for authors. Subsequent to this run, we generated corpus wide counts and ran a test leveraging Solr’s external file capability. Overall we saw essentially the same performance. We discuss the external file approach in a subsequent section of the document.
* In further research, we believe that we can allocate significantly less RAM to the Java heap than we did in the PoC. This was not an issue in the PoC, as there was much more RAM available than actually needed during the tests.

Table 6: Author Bulk Load statistics

|  |  |
| --- | --- |
| Queue Size | 49,705,977 |
| Solr cluster | 1 x m3.2xlarge instance |
| Number of indexer processes | 40 x cc1.4xlarge instances (20 indexer processes/instance) |
| Dynamo DB Read/Write rate | 4096/4096 |
| Processing Time | 4:33 |
| Index Rate | approx. 11,046,000/hr |
| Pre-optimize index size | 28.25 GB |
| Optimize time | 0:22 |
| Post optimize index size | 21.4 GB |
| Total processing time | 4:55 |

Table 7: Author 1x Update Load Statistics

|  |  |
| --- | --- |
| Queue Size | 250,000 |
| Solr cluster | 1 x m3.2xlarge instance |
| Number of indexer processes | 5 x m1. xlarge instances (5 indexer processes/instance) |
| Dynamo DB Read/Write rate | 500/500 |
| Processing Time | 0:37 |
| Index Rate | approx. 649,000/hr |
| Optimize time | 0:28 |
| Total processing time | 1:05 |

Table 8: Author 3x Update Load Statistics

|  |  |
| --- | --- |
| Queue Size | 750,000 |
| Solr cluster | 1 x m3.2xlarge instance |
| Number of indexer processes | 5 x m1. xlarge instances (5 indexer processes/instance) |
| Dynamo DB Read/Write rate | 500/500 |
| Processing Time | 1:52 |
| Index Rate | approx. 600,000/hr |
| Optimize time | 0:30 |
| Total processing time | 2:22 |

## Query Load Testing

For the full query set, we had an overall target of an average rate of 9 qps with no overall average response time. In addition we had specific average response times for the two identified query types (navigator and no navigator). We ran one load test box with one copy of the load test script. The script ran for 15 minutes to warm the cluster before official measurements were taken. We were able to far exceed the target Scopus requirements, reaching 20 QPS on average. In addition, Solr was able to surpass all of the existing specific query type measurements/requirements.

The single Solr server was running on one m3.2xlarge instance. The Server was configured to use at most 20GB of the 30GB available on the instance. The actual instance type we wanted to run on was not yet available from Amazon, so we were unable to leave enough RAM to run a separate update cluster on the same instance. This amount of RAM can easily contain the entire index in memory to get optimal performance from Solr.

The instance itself showed minimal CPU usage during the tests that would seem to indicated significantly more capacity could be extracted from the hardware. We did not push the hardware to its limit as we had already surpassed the targets One other item to note is that we had no replicas configured in the cluster to simulate an HA deployment. Adding a second instance would effectively double the query capacity to at least 40 QPS. It may be possible to investigate combining the author cluster onto the same hardware as the affiliation cluster to save costs (assuming an instance type with adequate RAM and disk drives for both indexes can be identified).

Table 9: Query Mix for Author Test set – Navigator Based

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Number of queries** | **Percentage of mix** |
| Navigator | 10,627 | 14% |
| No Navigator | 62,997 | 86% |

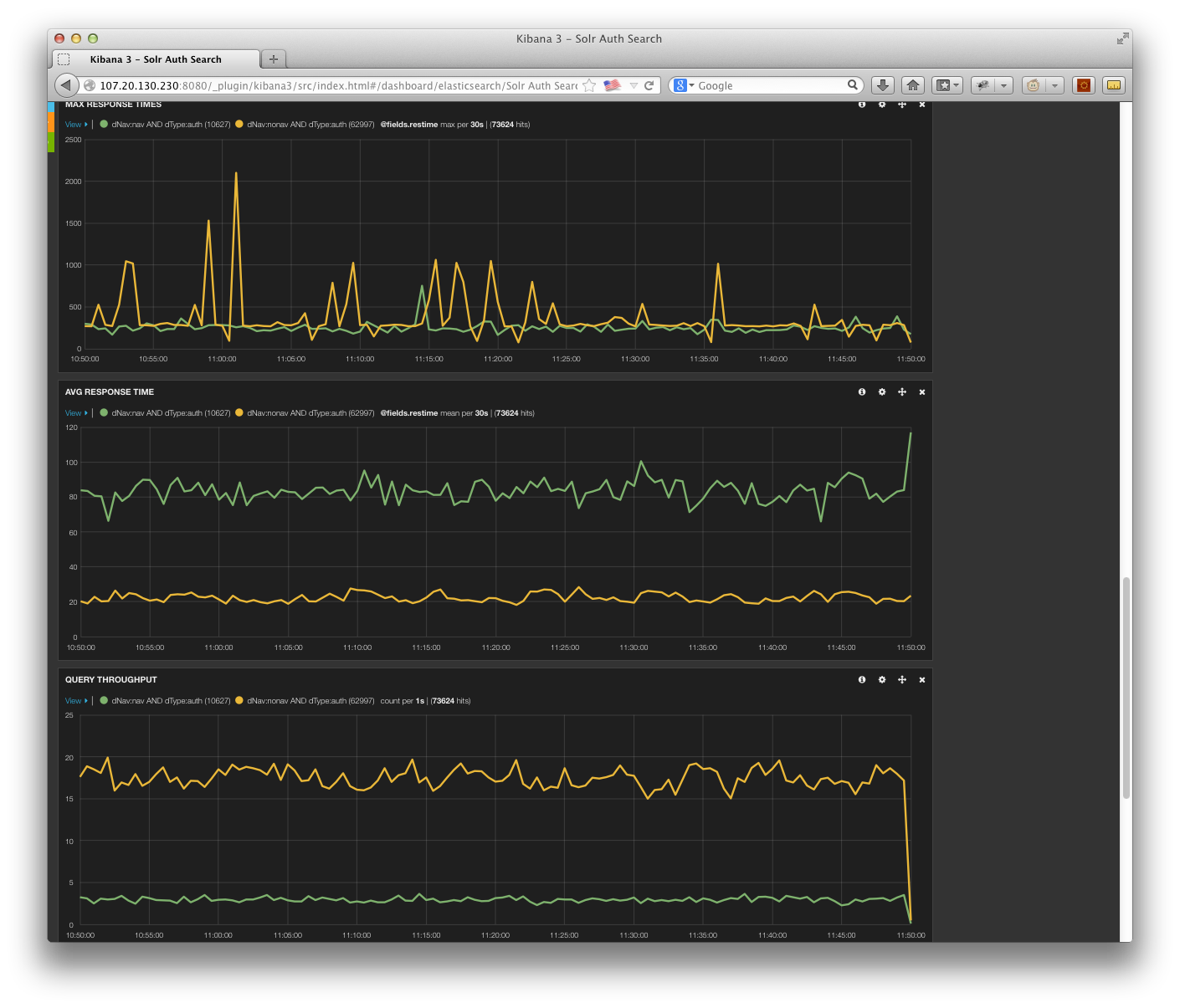
Table 10: Author full query mix 1 hour overall averages

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **AVG Request Rate** | **Max**  **Request**  **Rate** | **Avg Response Time** | **Max Response Time** |
|  |  |  |  |  |
| Scopus Targets | 9 qps | 15 qps | N/A | 29,091 ms |
| Solr Observed | 20 qps | 23 qps | 31 ms | 2,100 ms |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **Mean** | **Median** | **P95th** | **Max** |
|  |  |  |  |  |
| Scopus Target - Navigator | 242 ms | 101 ms | 1,013 ms | 29,091 ms |
| Solr Observed - Navigator | 83 ms | 73 ms | 172 ms | 752 ms |
|  |  |  |  |  |
| Scopus Target - No Navigator | 131 ms | 46 ms | 174 ms | 28,168 ms |
| Solr Observed - No Navigator | 22 ms | 19 ms | 35 ms | 2,100 ms |

Figure 3: Author Full Query Mix – 1 hour snapshot (4/14/14 10:50 EDT 11:50 EDT)





# Core Testing Results

## Core Content Loading

We were able to complete the initial bulk load of 50,773,230 records in approximately 17 hours and 26 minutes (including final optimization). We observed far less than 50% CPU utilization during the initial load and update runs (actual CPU usage during the update runs was 2% for the cluster). One will note that we were not running as many indexers as we had for the author records. As we ramped up DynamoDB R/W capacities, load performance increased linearly, implying we are not CPU bound on the machine and could significantly decrease the load times. In subsequent full content reloads, we were able to load the entire set of cores in 8 hours (including final optimization) leveraging 65 loader instances (20 loaders per instance) and DynamoDB set at 4096/4096. In this later load, the Solr instances showed approximately 30% CPU utilization. We could have run more instance to further reduce the time, but reached the instance type limit for our account.

While loading, Solr was configured to use a maximum of 16 indexing threads as well as a modified ramBuffer from the default. Specifically, the ramBufferSize was set at 4096MB with a maxBufferedDocs of 250,000. These settings control how much processing Solr will do in memory before committing to disk in the transaction log and represents the amount of work that can be lost if a machine goes down. Finally, the autoCommit maxTime was set to a max time of 10 minutes. This controls how frequently transaction log records are merged into the Solr index segments. Reducing the log size reduces the recovery time needed to replay the log entries against the check pointed index should a machine go down. Finally, at the end of indexing, we optimize the index to reduce the size and increase the search performance characteristics. We do this as we anticipate flipping indexes at which point the newly optimized index will become read only and only used for query resolution.

Machine/Solr details:

* AWS 5 x cr1.8xlarge
  + 32 vCPUs / each
  + 244GB RAM / each
  + 2 x 120GB Ephemeral SSD Drives / each
  + 10GBit Network performance
  + Note: recommend using some of the newer Amazon r3 family r3.8xlarge instances in production. This instance type will be a better match for Cores and initial reports list them as being less expensive than the instances we used. These new instances were announced but unavailable during the PoC
* Solr Config
  + -XX:+UseParallelGC
  + –Xms100G
  + –Xmx100G
  + 2 x 120GB drives in raid 0 array.
* Index
  + Approx. 500GB

Additonal notes:

* Flat optimize time based on index shard size. Optimize shards in parallel for fastest throughput.
* 15K errors reported during initial bulk load. Reran these records and they updated successfully. Error logs show they likely encountered timeouts during higher level Solr segment megers. Increasing the connection timeout accordingly in the indexers should enable us to avoid this situation.
* Could have run much harder, plenty of CPU available (decreasing the overall indexing time).
* We did not actually compute the corpus wide statistics for the test dataset. Records were loaded with dummy values for the citedByCount and citref fields that encompass the corpus wide statistics for cores.

Table 11: Core Bulk Load statistics

|  |  |
| --- | --- |
| Queue Size | 50,773,230 |
| Solr cluster | 5 x cr1.8xlarge instance |
| Number of indexer processes | 25 x cc1.4xlarge instances (20 indexer processes/instance) instances were ramped up over 30 minutes |
| Dynamo DB Read/Write rate | 1500/1500 |
| Processing Time | 16:08 |
| Index Rate | approx. 3,173,000/hr |
| Pre-optimize index size | 516.16 GB |
| Optimize time | 1:18 (run in parallel on each shard) |
| Post optimize index size | 465.95 GB |
| Total processing time | 17:26 |

Table 12: Core 1X Update Load Statistics

|  |  |
| --- | --- |
| Queue Size | 400,000 |
| Solr cluster | 5 x cr1.8xlarge instance |
| Number of indexer processes | 5 x m1. xlarge instances (5 indexer processes/instance) |
| Dynamo DB Read/Write rate | 500/500 |
| Processing Time | 1:59 |
| Index Rate | Approx. 200,000/hour |
| Optimize time | 1:20 (run in parallel on each shard) |
| Total processing time | 3:19 |

Table 13: Core 3X Update Load Statistics

|  |  |
| --- | --- |
| Queue Size | 1,200,000 |
| Solr cluster | 5 x cr1.8xlarge instance |
| Number of indexer processes | 5 x m1. xlarge instances (5 indexer processes/instance) |
| Dynamo DB Read/Write rate | 500/500 |
| Processing Time | 6:11 |
| Index Rate | Approx. 200,000/hour |
| Optimize time | 1:19 (run in parallel on each shard) |
| Total processing time | 7:30 |

## Core Query Load Testing

For the full query set, we had an overall target of an average rate of 65 qps with no overall average response time. In addition we had specific average response times for the two identified query types (navigator and no navigator). We ran 2 load test boxes with three copies of the load test script on each. The script ran for 15 minutes to warm the cluster before official measurements were taken. We were able to far exceed the target Scopus requirements, reaching 92 QPS on average. In addition, Solr was able to surpass all of the existing specific query type measurements/requirements.

The Solr cluster was running on 5 cr1.8xlarge instances. Each server was configured to use at most 100GB of the 244GB available on the instance (leaving enough to run a separate update cluster on the same instance without impacting performance). This amount of RAM can easily contain the entire index in memory to get optimal performance from Solr. This amount of RAM can easily contain the entire index in memory to get optimal performance from Solr.

The instance itself showed minimal CPU usage during the tests that would seem to indicate significantly more capacity could be extracted from the hardware. We did not push the hardware to its limit as we had already surpassed the targets. One other item to note is that we had no replicas configured in the cluster to simulate an HA deployment. Adding a second instance would effectively double the query capacity to at least average 184 QPS.

**Notes:**

The timings below reflect leveraging the dummy corpus wide statistic values that were initially inserted into the content when indexed. After the official statistics were gathered, we attempted to leverage the Solr external file capability as an alternative to having to individually update documents with their specific statistics after the Redshift job had run. Using this approach, we saw approximately the same general performance pattern leading us to believe we can leverage that capability to significantly reduce our ongoing update load with little performance impact. More information about our proposed use of the external file capability can be found in the “Corpus Wide Statistics” section later in this paper.

During the tests, we did not have any data loaded into the citref field (we did not run the Redshift jobs against the loaded set to generate that data). While we are able to leverage the external file capability for the counts, citref field needs to support true fulltext querying against it. This means we will need to support an update process after the Redshift job generating “citref” fields for impacted documents. The process will require an extension to the abstract transformation/indexer that will enable us to re-index the document while merging in the citref data. Based on our investigation, this implies an additional update volume of 500,000 records/day for cores. During load/update testing, we tested an update volume of up to 3x the maximum update load seen in our data from production. The additional load we will add would fall within that 3x test size and should still meet performance requirements.

Table 14: Query Mix for Core Test set – Navigator Based

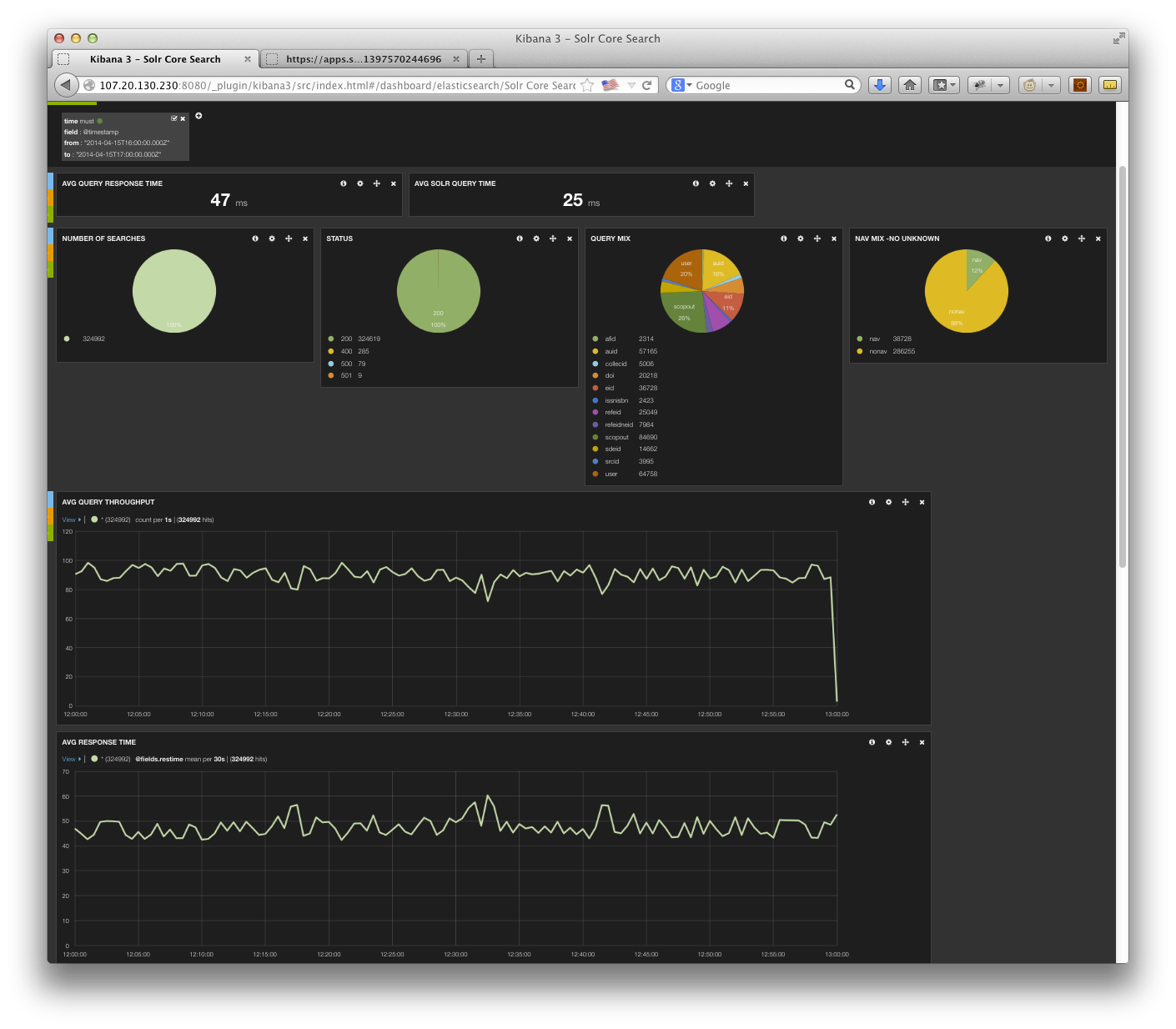
|  |  |  |
| --- | --- | --- |
| **Query Type** | **Number of queries** | **Percentage of mix** |
| Navigator | 38,728 | 12% |
| No Navigator | 286,255 | 88% |

Table 15: Full query mix 1 hour overall averages

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **AVG Request Rate** | **Max**  **Request**  **Rate** | **Avg Response Time** | **Max Response Time** |
|  |  |  |  |  |
| Scopus Targets | 65 qps | 91 qps | N/A | 179,189 ms |
| Solr Observed | 92 qps | 98 qps | 47 ms | 14,196 ms |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **Mean** | **Median** | **P95th** | **Max** |
|  |  |  |  |  |
| Scopus Target - Navigator | 452 ms | 259 ms | 1,022 ms | 65,058 ms |
| Solr Observed - Navigator | 117 ms | 88 ms | 326 ms | 12,707 ms |
|  |  |  |  |  |
| Scopus Target - No Navigator | 379 ms | 171 ms | 633 ms | 179,189 ms |
| Solr Observed - No Navigator | 38 ms | 25 ms | 81 ms | 14,196 ms |

Figure 4 Cores Full Query Mix – 1 hour snapshot (4/15/14 12:00 EDT – 13:00 EDT)

****

****

# Potential Problematic/High Cost Features

* **Dummies**: The Solr PoC did not include the Scopus Dummy abstract records. This was because the dummy data was not available when we received our dump of data from XFab. Dummies basically would follow the Core pattern in developing a solution. The main differences between the two are the number of records (dummies much greater) and record size (dummies much smaller than cores).

Based on our investigation it appears that Dummy searches occur on the order of a few thousand a day (a fairly small volume compared to the other content types). Should product decide that this is still a required feature in the product, we would recommend implementing it as yet another Solr Collection on separate hardware deployment. Given the data characteristics, it would likely need to be on a hardware footprint similar to cores and would incur similar development and ongoing hosting costs.

* **Patents**: Patent data was out of scope for the PoC as well. Assuming patents are a separate Solr collection, much as it is implemented today in Fast, it should be possible to implement without worrying about its impact on the other content types. We are aware of no “special” search capabilities required for patents beyond those already investigated so we believe Solr could support that data. Given there were no real requirements or sample records/layouts to look at, it is very difficult to provide a sizing estimate for the content type.
* **Tombstone queries**: This is not necessarily problematic but something that would require development (along with the associated cost). This functionality is currently provided by the Fast Unity layer and was out of scope for the PoC. Since Tombstone queries are a 2-step process, some of the scripting capabilities provided by Search Technologies QPL could potentially facilitate a faster and less expensive implementation. Another alternative would be to create a separate search endpoint that performs this 2-step query or implement the functionality in the new Solr SearchService layer.
* **Storing/Retrieving Many Fields**: Having pre-defined views that have blanket sets of fields to return results in lower performance as unused/unwanted data has to be retrieved, formatted, and transmitted to the client. All calls should explicitly list the fields that are actually needed.
* **Scoped Searching:** Scopus needs to ability to search within authors, affiliations, and references. The scope search is a bit more complex than the ScienceDirect scope requirements. Specifically, there are multiple levels of scoping (instead of the single level needed for ScienceDirect. It’s possible the custom development solution created for ScienceDirect could be used (or extended). Another option would be to leverage QPL as this feature is already supported.
* **Wildcard and stemming:** Solr will not analyze a term that contains wildcard characters. This has the net effect of not applying any stemmers during the query execution for that term. Many search engines have this limitation. If this is needed, custom development would be required. However, there could be a significant performance penalty.
* **Leading Wildcards:** We have elected to not enable the 'reversewildcard' index. Should the Scopus Product organization feel this feature must be carried forward to the new implementation, it will have the effect of nearly doubling the index size. This will have the side effect of having to double the hardware footprint needed to maintain the performance characteristics we observed in the PoC. This is specifically true as performance is tied to keeping as much of the index in memory as possible (and the index would grow in size significantly).
* **Limiting Wildcard expansion:** Currently FAST only allows a wildcard to expand to the first 2000 terms (presumably to prevent performance problems). If this feature is needed (and necessary for performance), custom Solr development will be required.
* **Facets:** While we have indicated what testing was done to verify values/counts across shards (when ordered by count descending), it should be noted that people have commented that 'range' type facets (think buckets) could have issues.  Since this was not a requirement for Scopus, this is not something we deeply investigated.  However, it is something that should be kept in mind going forward.  Facet values can't by default be ordered by value descending.  The workaround is to double-store the facet-value (regular and reversed). This was noticed as pubyr seems to be one of the few fields that are ordered descending by index value (instead of count).   It is unclear whether Solr could replicate the functionality provided by FAST today where values in a facet can be sorted by letter (and provide accurate results across shards). Further testing would be required.

# Answer Features/Count Results

This section contains the results of the manual verification of the feature/count queries. For this test, the same query was run in Scopus as well as the Solr search cluster. In general we expect the numbers from Solr to be slightly lower than Scopus. This is due to the changes in the production system document set since the snapshot for HotHouse was taken. Other minor variations may exist due to some errors/misunderstandings in the translation of the CIP into the Solr index, stemming variations between the two search engines, lack of merged fields in our dataset (e.g. XFab merged affiliation data into the other records at syndication), etc. For example, we noticed a CIP mapping issue with author first name where the explanation of the CIP syntax we received didn’t match with what the Scopus queries were returning. Additionally, some feature functionality required custom Scopus development or the purchase of 3rd party plug-ins that was out of scope for the PoC (e.g scoped searching).

In the table below, areas where we observed significant deviation between the systems are highlighted in red. In those cases, the Query/Notes column will have our best attempt of explaining why we saw the deviations.

Table 16 Feature/Count Testing Results

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Scopus Count** | **SOLR Count** | **Query/Notes** |
| CORE1 | 1 | 1 | doi(10.1109/lawp.2005.850798) |
| CORE2 | 2 | 2 | eid(2-s2.0-84861409323 OR 2-s2.0-0032977045 OR 2-s2.0-84871047622) |
| CORE3 | 1334 | 1315 | refeid(2-s2.0-0034824859) |
| CORE4 | 14 | 15 | au-id(55177469200) |
| CORE5 | 122599 | 121545 | af-id(60026851) |
| CORE6 | 2 | 0 | scopus-id(33749522827 OR 33750354968). CIP configuration problem. |
| CORE7 | 1927 | 1927 | issn(2090-4649 OR 0301-0538 OR 004-2675-X) |
| CORE8 |  | 0 | fulltext-eid(1-s2.0-S0169409X13002275) |
| CORE9 | 15 | 15 | srcid(24527) AND PUBYEAR = 2012 |
| CORE10 | 4920871 | 4779881 | all(blood) |
| CORE11 | 3524700 | 3499039 | title-abs-key-auth(blood) |
| CORE12 | 3523591 | 3498027 | title-abs-key(blood) |
| CORE13 | 1383826 | 1383796 | abs(blood) |
| CORE14 | 209717 | 207911 | affil(oxford). |
| CORE15 | 863 | 846 | chemname(rhamnose) AND indexterms (polysaccharide). |
| CORE16 | 1021 | 1002 | key(rhamnose AND polysaccharide). |
| CORE17 | 4145698 | 2388514 | ref(engineering). ML had similar results. Since we used the ‘storage’ format and not the ‘syndication’ formation for the XML, it’s possible that additional information was included in the ‘syndication’ format. |
| CORE18 | 60734 | 60580 | abs(blood) AND abs(cancer) |
| CORE19 | 664325 | 656006 | (abs(brain) AND NOT abs(cancer)) |
| CORE20 | 1611025 | 1597394 | (abs(brain) OR abs(cancer)) |
| CORE21 | 3308 | 2960 | abs(brain PRE/5 cancer). SOLR Surround Query Parser will not stem tokens. QPL could address this limitation. |
| CORE22 | 5045 | 4499 | abs(brain W/5 cancer). SOLR Surround Query Parser will not stem tokens. QPL could address this limitation. |
| CORE23 | 486 | 141 | title (practice W/5 "letter"). SOLR Surround Query Parser will not stem tokens. QPL could address this limitation. |
| CORE24 | 600 | Error | title(neuro\* W/3 letter). SOLR errored on too many terms. . Would require custom development or QPL. |
| CORE25 | 24767 | Error | title(neurological) AND NOT title(neurological W/5 letter). Could not construct query in SOLR (parse error). Would require custom development or QPL. |
| CORE26 | 754 | 141 | title((neurological OR letter) w/5 practice). SOLR Surround Query Parser will not stem tokens. . Would require custom development or QPL. |
| CORE27 | 3 | 3 | auth(Rumiá Arboix J). |
| CORE28 | 103987 | 103320 | title(blood) AND pubyear AFT 2003 |
| CORE29 | 23933106 | 23650409 | pubyear AFT 2000 AND pubyear BEF 2014 |
| CORE30 | 3727 | 3696 | abs("heart attack") |
| CORE31 | 21 | 13 | abs({heart-attack}) |
| CORE32 | 43462156 | 43082537 | all("the") |
| CORE33 | 16330 | 16268 | title ("the end" ) |
| CORE34 | 96730 | 95800 | abs(thio\*e) |
| CORE35 | 5325 | 3052 | abs(thio?e). Unclear why Solr has 2K less than Scopus. |
| CORE36 | 2924 | 2787 | title("\*snake venom") |
| CORE37 | 2924 | Not Implemented | title (\*snake pre/0 venom). Solr has a praser error. Would require custom development or QPL. |
| CORE38 | 65626 | Not Implemented | author-name(smith,j). Not implemented. Would require custom development or QPL. |
| CORE39 | 276 | Not Implemented | author-name(smith,john). Not implemented. Would require custom development or QPL. |
| CORE40 | 81374 | Not Implemented | affil(affilorg (university california) AND affilcity(san diego)). Not implemented. Would require custom development or QPL. |
| CORE41 | 22488 | Not Implemented | ref(sanger nicklen DNA). Not implemented. Would require custom development or QPL. |
| CORE42 | 6178 | Not Implemented | ref(refauth(darwin) AND refsrctitle(species) AND refpubyear IS 1859). Not implemented. Would require custom development or QPL. |

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Scopus Count** | **SOLR Count** | **Query/Notes** |
| AUTH1 | 1 | 1 | au-id(10038760900) |
| AUTH2 | 11686 | 11479 | auth-last-name(smith) AND auth-first(j\*). |
| AUTH3 | 7021 | 4599 | auth-last-name (EQUALS(smith)) AND auth-first (STARTS-WITH(j)). CIP mis-mapping issue in the Solr index. |

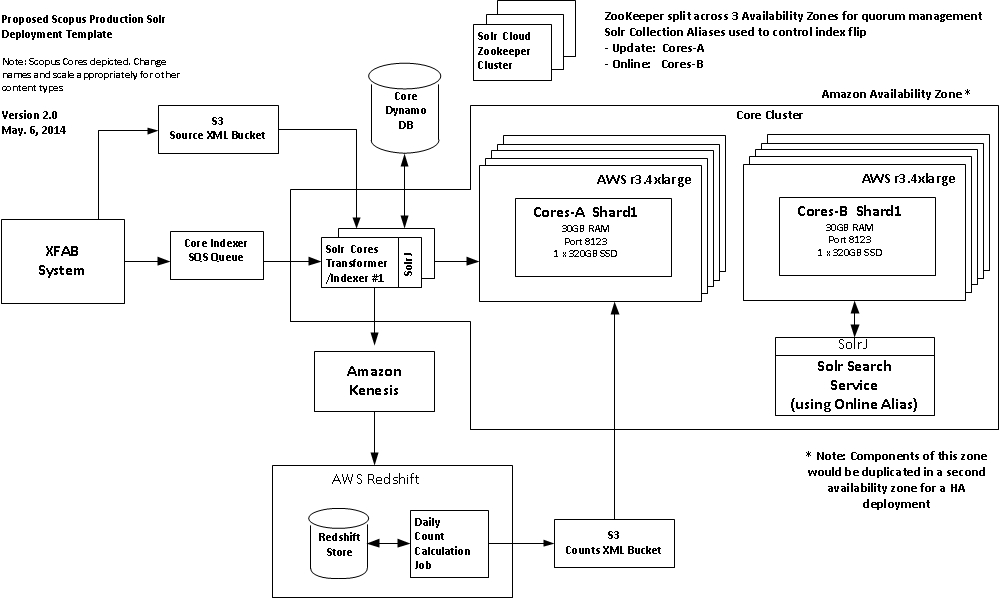
|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Scopus Count** | **SOLR Count** | **Query/Notes** |
| AFFIL1 | 1 | 1 | af-id(60030162) |
| AFFIL2 | 348 | 17835 | affilcity(london). Likely attributed to differences in publication counts (indexed records had hardcoded values) as well as identification of root nodes. MarkLogic and SOLR had same count (25,847) when all filtering was removed. |
| AFFIL3 | 1 | 4 | affil("university of york"). Likely attributed to differences in publication counts as well as identification of root nodes. MarkLogic and SOLR had similar counts (9/6) without filters with the difference being stemming of ‘universities’ in MarkLogic. |
| AFFIL4 | 86 | 666 | affil (university of york). Likely attributed to differences in publication counts as well as identification of root nodes. MarkLogic and SOLR had similar counts (1087/1075) without the filter with the difference being stemming of ‘universities’ in MarkLogic. |

# Scopus Specific Challenges Investigated in PoC

## Introduction

The following diagram presents a more detailed representation of the proposed deployment architecture as it relates to managing the update of content and management of the various shards and Solr Collections. It will be used as a reference in the discussions of a number of the Scopus specific challenges discussed in this section. The diagram specifically depicts the deployment in support of the Cores datatype, as it is the most complex of the three that were investigated. All three Solr deployments will be running at the same time to provide a complete search solution for Scopus. The Authors and Affiliations diagram appear similar to Cores (lacking the AWS Kinesis/Redshift Components and encompassing only 1 AWS instance each).

Figure 5: Detailed Scopus Cores Deployment proposal



## Large, Accurate Navigators/Facets

As previously mentioned in this paper, Scopus presents the user with a number of navigators that can have millions of distinct values in them. This was one of the areas of concern that was noted in the ScienceDirect Hothouse, as performance appeared to degrade linearly as the number of items to count for the navigator increased. After discussions with consultants from Search Technologies, we investigated the use of the Solr “Doc Values” feature that was added in Solr 4.2 as a potential solution for this problem. “Doc Values” leverages a different way of building the index for those fields where it is applied. Instead of inverting those field’s data into the index to make them quickly searchable, it builds a column-oriented field with a doc:value mapping that greatly increases speed for things like faceting, sorting, grouping, etc. When applying this approach to the Scopus content, we saw a drastic increase in search performance for those queries with large answer sets and high cardinality navigators (based on our previous experience during the ScienceDirect hothouse).

There was also the concern for whether the facet values/counts are consistent (specifically in a Solr sharded environment). We developed a Node.js testing tool that allowed us to request all of the core content’s facet values and their associated counts for each Solr query specified in a file. The tool was run four times requesting the top 3, 10, 20, and 200 facet values/counts for the 50 queries specified in the file. In between each run, the Solr cluster was restarted to flush the cache. We then compared the results for each facet and verified that the top 3 values matched the first 3 values in the top 10, top 20, and top 200. We did the same for the top 10 and top 20. In the end, we saw no difference in the top values/counts across all of the facets. Our findings with this test align with email threads on the Solr usergroups. One such thread that provides a very good explanation of how Solr calculates facets can be found in the document’s appendices

We then wanted to verify that the counts for the facet values were accurate. We developed another Node.js testing tool (similar to the first one) that executed the queries from a file. However, this time in addition to requesting the top n values for each of the facets for the query, we then submitted a search request (appending the facet value to the original search request) and compared this result with the count returned for the facet. We ran this test for the same 50 queries specified in the file (requesting the top 20 values for each facet both sorted by count as well as value) and saw no differences between the count returned for the facet value and the count returned from the value appended to the original search query.

## High Update Volumes

Based on the statistics we collected over a 3 month period, the Scopus product exhibits a much higher update (and erratic) rate than ScienceDirect. In addition, there is a significant potential search index update load related to the corpus-wide statistics generation. The team had serious concerns about the ability to guarantee search performance using the “update in place” on the live index strategy proposed by the ScienceDirect Hothouse. As a result, we are proposing a dual copy solution, where there is an online copy of index that is actively queried by the product and an offline copy that processes updates in real time. Periodically, these two indexes are flipped to make the latest set of changes visible in the product and the old “online” index starts getting used for updates.

This approach and how we have chosen to implement it has some impacts on our hardware configuration as well as management of each Solr Collection’s shards. In order to get the best query performance from Solr, you want to keep as much of the index resident in RAM as possible and for indexing you want as much RAM as possible as well as fast disk drives (SSD if possible) to decrease the latency of writing Solr segments and merges. Furthermore, you must also take the latency needed to sync up the “new” version of the index to the outdated version after the index “flip” occurs.

Taking these three goals into account, we decided it would be best to run the two indexes as two distinct Solr deployments. This implies that optimally each machine will need sufficient RAM to hold 1x shard size + Solr Heap (variable dependent on content type) + 5 GB (for OS). Additionally, you need 2-3x index size SSD drive space (to provide space for the index as well as work space for Solr optimization). AWS instance RAM size and SSD capacity on instance types will likely drive the sharding strategy for the various indexes.

At one point, we had considered running the two collections in parallel on larger AWS instances. In order to avoid possible cross Solr installation interference, we decided to move them onto discrete, smaller instances. This decision added the need to transfer snapshots of the index between machines as part of the “Flip” (discussed later in this section). As part of our testing, we measured the amount of time needed to backup our largest content type shard (cores) to S3 and then to recover it onto a separate machine. A cores shard is approximately 95GB in size. It took approximately 2 hours to back up the shard and another 30 minutes to recover it to the second Solr instance. The backup time will need to be taken into account as part of the “Flip” elapsed time calculations (as the backup should be completed before flipping indexes to avoid network/disk contention).

## Corpus Wide Statistics

One of the key product features of Scopus is the availability of corpus wide statistics that assist the researcher in validating the merit of a paper, author, or institution. This is presented via a “count” of the number of times a paper is cited, or number of papers an author or institution has published. The data that drives these “counts” is derived from Scopus Core records that have been augmented with ids for the various entities in the Scopus corpus. A new core record or changes to an existing record can have rippling effects across the entire set of Scopus content and drive an exponential number of updates. As the product has requirements for providing “accurate” statistics at all times, it poses a problem in that individual updates to impacted documents do not occur within a single transaction and the counts can “drift” for a period of time until all the associated updates complete. This “drift” has been an issue with the existing Fast search implementation from the beginning.

In order to avoid this drift, the Solr solution has approached the problem with a “snapshot” based solution. Specifically, The Solr index will only be updated and made visible to the end user at specific intervals. On an ongoing basis, XFAB puts Core changes onto the appropriate SQS queue for processing. As each core is merged into the Solr index, the key information that drive the counts is extracted and sent to our AWS Redshift cluster via AWS Kinesis queuing (Kinesis is being leveraged as Redshift is unable to handle the direct update load without significantly degrading indexing speed). At the pre-defined index promotion intervals, ongoing updates from the XFAB system into the Solr SQS queues will be paused and the queues allowed to drain of records. When the queues have drained, XFAB will kick off a set of Redshift jobs (one for each corpus-wide statistic) against the data stored there to generate the record level statistics for each content type. At the end of the Redshift job, the resulting statistics will be flattened into a record id:value file and exported to S3. Once the export has completed, a copy of the appropriate file(s) will be posted to each shard of the update Solr Collection in preparation for the promotion of the new index as part of the “Flip”.

Within Solr, we propose leveraging the “external file” feature that allows you to store information for a record outside of the index without needing to update it into the index itself. In the Scopus use case, these statistics values’ are typically used for sorting, returned in search results, or used as part of range queries. The Solr “external file” feature supports these use cases and allows us to avoid having to update all of the impacted documents with new count values, significantly reducing the update load on the index. The one drawback to this approach is the lack of direct querying against fields stored in these external files (not currently a Scopus requirement for the counts). Should this become necessary, it would be possible to migrate back to updating the values into the index itself at the drawback of longer cycle times to update and promote an index to the end user. This capability could be extended to other types of statistical data that the product wanted to sort on as well. For instance, Scopus could integrate usage “altmetrics” from the product by capturing abstract views data, rolling it up into an external file field, and having a sort by popularity option based on those counts.

DISCUSS/Mention Single Source of Truth results, Refer to CTO index

## Index Flipping

The key to the proposed Solr solution rests in our ability to manage our updates (content and corpus wide statistics) as snapshots and to be able to easily promote new versions of the index into production easily. We are referring to this promotion as the “Flip”. In this section we will lay out how we envision the Flip actually occurring in production.

To implement the Flip we are relying on Solr’s alias capability. Aliases allow you to point a logical name to a physical set of nodes/shards that encompass a set of content. In our case, we would have two aliases per content type: “online” and “update”. Each of them would point at one of the two copies of our index for that content type. In the diagram depiction above for Scopus Cores, we have A and B copies for each shard, one pair per box. In the case of Affiliation and Authors, there is only one shard and one box in play.

SolrCloud leverages Zookeeper to keep track on shard copies for a Solr Collection within a cluster. It also manages the definition of aliases based on the contents of a collection. Zookeeper keeps track of the current state of the cluster as machine join/depart the cluster and makes that information visible to all of the participants. In this way, it behaves as a smart load balancer for those processes that leverage it to manage the list of active nodes.

Clients would leverage the SolrJ CloudSolrServer class (or equivalent class in other languages) to manage their connections to the Solr cluster. In our Scopus deployment the “online” alias would point to either the A or B index, and update to the other. The Transform/Indexers would use the “update” alias while the SolrSearch Service and other search clients would use the “online” alias. At the predefined promotion times, the flow of updates from XFab would be paused, the Corpus Wide Statistic Updates performed, and the index is prepared for the flip. Once the Statistic Updates are copied to each shard in the cluster, a Solr optimize is kicked off on each shard to compress it down as much as possible. At the completion of the compress, a snapshot and backup is made and stored off to S3. Finally, the alias update is initiated. The “online” alias is changed to point to the copy of the index that was just optimized. A side effect of this command will have the side effect of updating all the clients’ local routing tables with the new endpoints and search traffic will migrate to the new index. Once traffic has migrated, the previous index will be restored from the snapshot of the new index that is on the same AWS instance. Once complete, we will refresh the ‘update’ alias to point at the newly restored index and update flow from XFab will be resumed. We have tested the updating of the collection alias that would happen in the “Flip”, and the rerouting of the traffic seems appears instantaneous.

# Operational Challenges

Describe predicted challenges that will/may be experienced if this solution were chosen to go forward.

## Integration with Existing XFAB Job Control System

The Scopus Solr workflow is much more complex than the one proposed last year in the ScienceDirect Hothouse. Beyond posting all changes to the appropriate SQS queue, we need to be able to pause that posting, monitor queue depth to make sure queues have drained, initiate Kinesis and Redshift jobs, push the results to multiple boxes, and manage the index “Flip”. Scopus will obviously need some sort of job control management layer to orchestrate all of these tasks in the proper sequence. At one point we considered looking at AWS Simple Work Flow, but decided that it probably makes more sense to try and drive all of this processing by extending the existing XFAB job control system to provide end-to-end management of the process. Since all of the Solr and AWS components expose administration APIs via HTTP, we assume this integration should be possible with some additional internal development on the XFAB system.

## Storage vs. Syndication format

For the PoC, we leveraged a static dump of the storage format of the Scopus content. This format is actually different from the version of the records sent to FAST, referred to as the “syndication format”. The majority of the differences revolve around the inclusion of information out of the affiliation records into the author records, although there are other minor enhancements made as well. While we were able to re-implement some of these modifications for the PoC, we weren’t able to replicate all of them. This prevented us from being able to execute some functional and load test queries during the PoC, specifically in the author cluster. In the long term XFAB may want to revisit having 2 different formats of the records in the system.

## Limited PoC Development with AWS Kinesis

Time limitations as well as Java version restrictions prevented the team from doing any significant development prototyping of the Kinesis data stream population or the periodic feed to Redshift.

NOTE: We are still going to try and do something on this if time permits.

## Management of the Flip

While the proposed solution for the Flip appears that it should work on paper, we have not actually implemented or attempted it as part of the PoC. It is possible that there would be unforeseen issues that would require a change in approach.

## Lack of “Rack-Awareness” in Solr Cloud

Unlike other search engines like the Lucene-based ElasticSearch, Solr Cloud does not currently provide ability for nodes in the cluster to be location aware. This implies that out of the box, it will be more difficult to ensure the distribution of shard replicas that ensures high availability of components (e.g. replicas for shards in 2 different availability zones). Current posting on the Solr user mailing lists indicate this is an outstanding request, but a low priority one. We believe we may be able get around some of these limitations by programmatically managing the ZooKeeper clusterstate.json during server provisioning to ensure the proper distribution of shard replicas to nodes.

Since individual replicas of a shard are all treated with the same priority, it is also probably that a given query may be routed to shard-replicas in a different availability zones than the client. During the PoC, we actually ran follow on load tests where we manually split the collection across availability zones. Results show an increase in response time of approximately 10ms as compared to the official tests where all components were in the same availability zone. More details of this subsequent test are located in the appendix of this document.

## Resource Availability

There is currently a limited pool of development resources with Solr experience within Elsevier. They are currently engaged in implementing the ScienceDirect migration to Solr. It is possible that there will be availability conflicts depending on the timing of the Scopus project initiation. In addition, the Solr PoC team is recommending using SolrCloud. We do not believe that the current Solr development team has extensive experience with SolrCloud. It is unclear how big an impediment that will be to the project.

## CIP Mappings/Crispness of Product Functionality Requirements

After working with the Scopus data in the PoC, it became obvious that there is more than a bit of ambiguity as well as a lack of clear understanding of how the current implementation is working. While we implemented the PoC to match our best understanding of the requirements, it became obvious that we did not always succeed in implementing the features as they currently exist. It is not always clear if people don’t understand how the product works, or if the current implementation is not working as expected. In addition, the CIP documentation is much more obtuse than the ScienceDirect counterpart. Part of this is due to the presentation while other parts is due to the inherent complexity it is trying to represent. A good example is the use of “AND” and “OR” as connectors of the XPath definitions for fields. After a long discussion of the semantics of theses operators, during testing it became obvious that the presented explanation was generating indexes that would not properly match some of the test queries. In a couple of cases we were forced to note that we got the mappings wrong and that a certain percentage of searches would not find the anticipated results.

# Deployment Architecture/Cost

The following proposed architecture is based on one of the possible deployment configurations that is possible with Solr, and the one most similar to what was implemented in the PoC (leveraging SolrCloud). Specifically, it assumes each content type has its own SolrCloud cluster on appropriately sized hardware. There are also a number of “shared” components that are leveraged across all of the content types. Typically, those components that are content-type specific have the content type identified in the label in the diagram. As mentioned elsewhere in the document, it may be possible to combine some of the content types onto a single hardware deployment at a reduce cost. For the sake of explicitness, we have them broken out separately.

## Proposed Architecture

Below are the proposed deployment architectures for SolrCloud for Scopus, one for each content type. They basically differ in the number and type of SolrCloud nodes needed to achieve the desired performance requirements for Scopus. In addition, the Cores architecture reflects the Kinesis feed needed to populate data into the Redshift store that is used to generate the Scopus corpus-wide statistics that are periodically migrated back into the various Solr clusters. It is only present in the Cores’ architecture, as the core data contains all the data that drives the statistics that feed into all the content types.

The architecture is actually fairly straightforward. The STEPS XFab fabrication system populates the appropriate S3 buckets with appropriate changes to the Scopus records and adds corresponding new/updated/deleted records into the appropriate processing queues for Solr indexing. The SolrCloud deployment for each collection is spread across both availability zones. We have already discussed the difference between the A vs. B version of the shards in our discussion of our “Flip” approach to offline updating, so we will focus more on the replication of shards to achieve HA for the product.

For each content type collection, here is a 3-node Zookeeper cluster that manages the Solr nodes for that collection. We iitaqlly proposed leverage a single shared ZooKeeper cluster, but changed our design based on discussions with RBI. In order to manage potential quorum issues with a loss of a ZooKeeper node or complete AZ, each Zookeep node runs in a distinct availability zone (implying the ZK cluster will span 3 AZs). These Zookeeper instances run behind an AWS Elastic Load Balancer) ELB that is used to shield the actual IP addresses from clients of the cluster. Each availability zone is running a copy of the appropriate conversion/indexer program to provide redundancy for that component.

All conversion/indexer nodes actively process messages from the same queue and submit the converted XML document to the Solr cluster for indexing. The conversion/indexers leverage the SolrJ CloudSolrServer class to manage a local copy of the official ZooKeeper list of Solr nodes that are available in the cluster. This class periodically communicates with ZooKeeper to confirm its local node list is accurate. The actual calls to index the documents are made directly to the Solr nodes bypassing the load balancer. As SolrCloud leaders can migrate within the cluster between replicas, there is no guarantee that the leader for the shard the document will end up on will be in the same AZ as the indexer. This implies there will be a certain level of cross AZ “chatter” processing update requests. As is the case with SolrCloud, a non-leader node that receives an update will forward the request to the appropriate shard leader (where ever it is located). From there, it will be sent to all the replicas for processing. Once all the shard replicas confirm processing the request, the leader will send back a confirmation to the indexer (potentially through the initial Solr node it contacted).

The query resolution approach has many similarities with indexing. It too assumes that the SearchService components being deployed leverage the ZooKeepers via SolrJ CloudSolrServer. This implies that individual searches bypass the load balancers as well when submitting queries to Solr. They also have the same potential issue for cross AZ “chatter” on query requests. Since the Zookeeper has information for all nodes across the AZs, it is possible that requests will cross AZs for some or all of the shard replicas in the cluster. Unfortunately, SolrCloud does not provide “rack awareness” which would allow us to modify the priority of “local” shards when distributing queries across the cluster. This will lead to increases in query response time. We ran a follow up load test to quantify this impact, which was actually minor. More details are available in the appendix at the end of this document.

Should the cross-AZ challenges become significant, it would be possible to modify the deployment to have two completely distinct clusters, each of which gets an identical feed of processing entries on their respective SQS queues. While this approach would allow us to ensure all traffic remained within an AZ, it would add complexity to synchronizing both clusters as well as the “Flip” of the offline updating index to online as well as additional hardware costs.

Figure 6: Cores Architecture

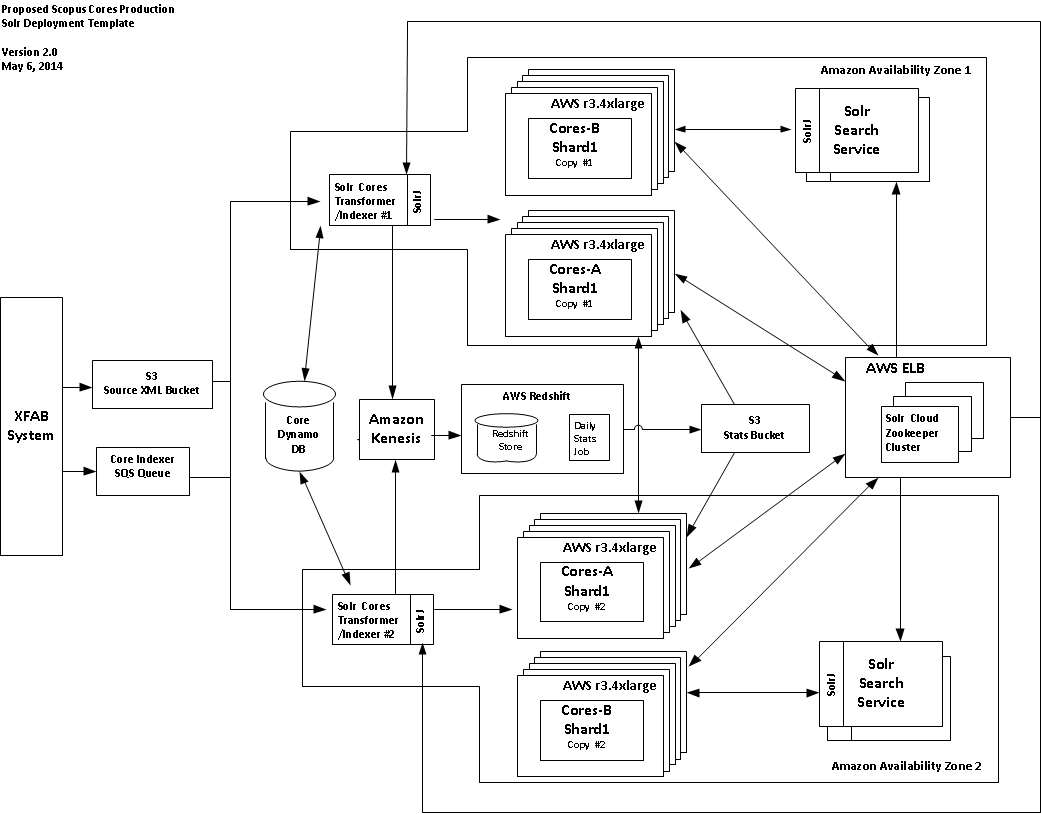


Figure 7: Author Architecture

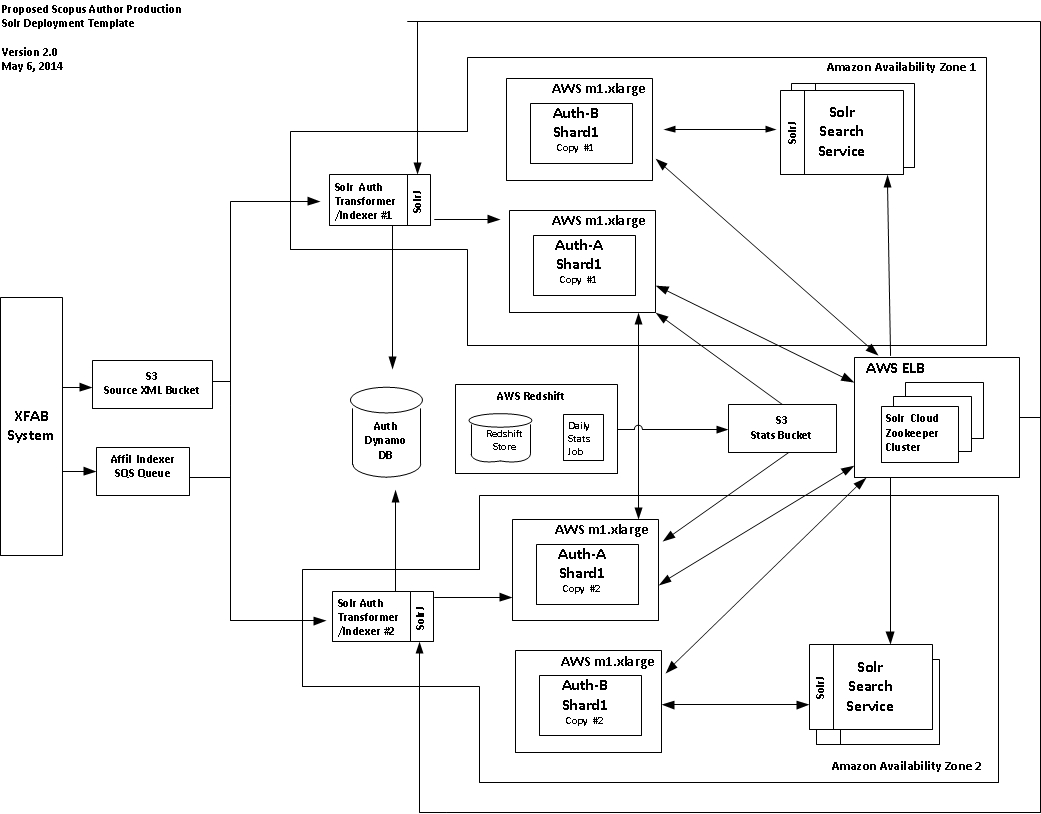
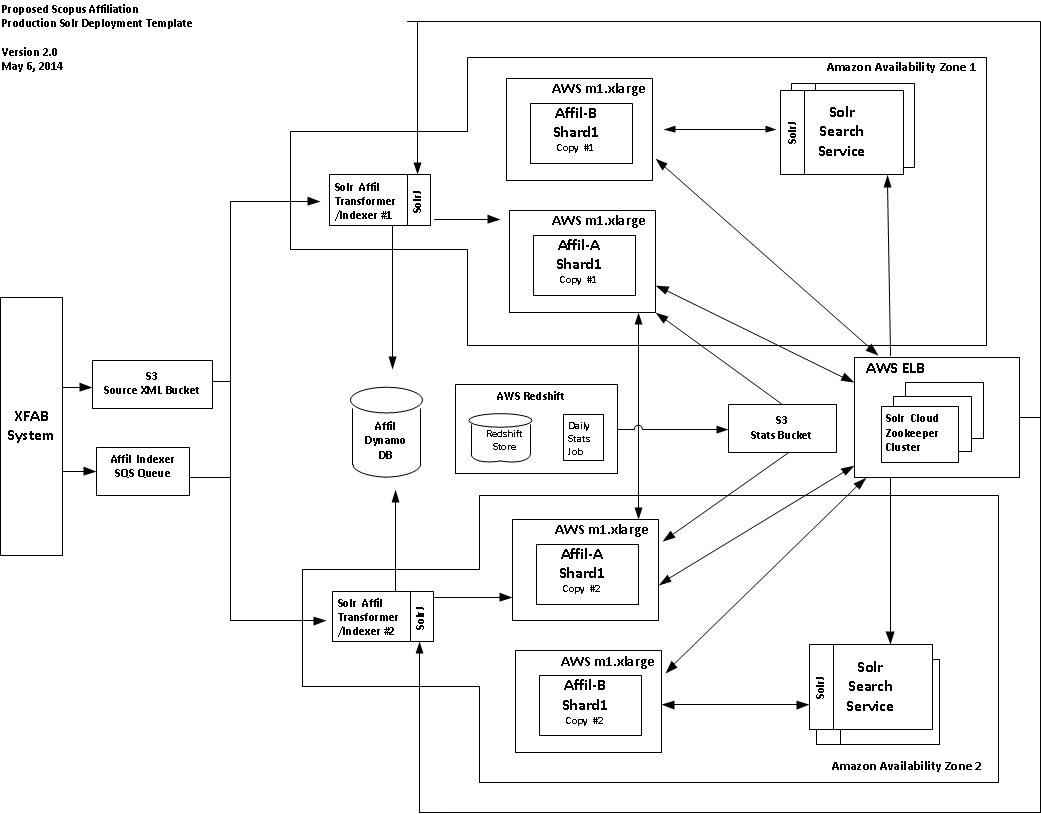
****

Figure 8: Affiliation Architecture

****

## Proposed Operating Costs

Please see the associated costing estimate spread sheets for specific cost estimates for our proposed deployment.

### Scopus Affiliations

Include the various AWS costs (both ongoing and complete reload)

* DynamoDB
* SQS
* SNS
* EC2
* S3 and/or EBS
* Elastic LB
* Static IP Addresses

### Scopus Authors

Include the various AWS costs (both ongoing and complete reload)

* DynamoDB
* SQS
* SNS
* EC2
* S3 and/or EBS
* Elastic LB
* Static IP Addresses

### Scopus Core Abstracts

Include the various AWS costs (both ongoing and complete reload)

* DynamoDB
* SQS
* SNS
* EC2
* S3 and/or EBS
* Elastic LB
* Static IP Addresses
* Kinesis
* RedShift

# Pros/Cons

## Pros of Solr

What would be the advantages?

* Mature offering
* Solr experience within Elsevier
* Open Source (cost savings?)
* Extensible/Flexible
  + Add handlers, parsers, etc.
  + Target specific content types to appropriately sized AWS EC2 instance types
  + Availability of 3rd party plugins (e.g. QPL)
* Scalable
* Cloud friendly
* What else?

## Cons of Solr

What would be the disadvantages?

* No internal experience with SolrCloud
* Record length limitations with Kinesis
* Stop the world GC.
* Lack of “Rack-Awareness” or control of shard-replica placement in cluster.
* What else?

# Known Limitations

* None currently identified

# Additional Thoughts

## Minimize Document Size Sent to Solr for Indexing

Currently, when we create the documents for indexing to Solr, we concatenate the values for many of the group fields (such as all, allsmall, allmed, etc) on the client side and then send this larger document to Solr for indexing.  A more efficient approach would be to use the copyfield concept in the schema.xml file.  This feature allows many fields to be copied (concatenated) into the same field achieving the same goal we were trying to achieve but without the overhead of sending the larger document to Solr for indexing.  The net result should be better throughput for indexing.

## Increasing Indexing Throughput

I may be possible to increase indexing throughput by tweaking some of the Solr configuration settings in the solrconfig.xml. Since the update instances are now running on distinct AWS images, it may be possible to increase the number of indexing threads, modify the merge policy or RAM buffer settings. Additional testing would be necessary to determine the optimal settings for each content type.

## Investigate Virtual Private Network – Increase in network performance - Still need to write

## Data cleanup

Looking at the data that has been indexed in Solr as well as the distinct facet values returned, it is obvious that there are opportunities to improve the quality of the data that is indexed. There are fields/facets like dates that have non-numeric or values that are obviously out of range. Other fields have obviously incorrect values as well such as journal names that consist of single commas. It could be possible to add a set on “no brainer”, conversion rules into the transform/indexer code to address the most blatant offenders that give the impression of poor data quality.

## Resolve Facets in Parallel

The default behavior in Solr is to serially resolve multiple facets specified in a request.    To improve overall query performance, the facets can be resolved in parallel by specifying facets.thread=-1 as an int parameter in the correct handler defined in solrconfig.xml.  We did some basic testing with this setting and noticed a significant increase in performance for queries with navigators as well as the overall query response time.

## XML Handling

One of the requirements for some Scopus fields is to retain the XML markup.  Since the XML markup does not need to be searched (only stored/retrieved), it should be possible to leverage a combination of the charFilters for PatternReplaceCharFilterFactory and HTMLStripCharFilterFactory in the analyzer for those fields in combination with a tokenizer and the necessary filters.  We did some basic testing and this solution seems to be sufficient.

## Filter Cache

The LFU algorithm is probably more appropriate for the filter cache (instead of the LRU).  This should allow a smaller cache specification for the filter cache while maintaining the benefit.

Currently, queries were rewritten during the load test to move any date query (pubyr or fastloaddate) to the 'filter query'.  This allows the date type query to be cached as a filter and potentially re-used for subsequent queries.  We did notice improvement after implementing this change.  Currently, the date was appended to an existing 'fq' as an AND clause.  However, it might make sense to keep each data as a separate 'fq' clause so each atomic date could be cached as a separate filter (or even not cached).

## Relevancy

Keep in mind that IDF/TF is only maintained at the shard level (and not at the collection level).  This is important to remember.  Assuming the content is relatively evenly distributed across the shards, the theory is the TF/IDF will roughly be equivalent across the shards.  When using edismax, consideration should be giving to using pf and/or pf2/pf3 (think bi-gram/tri-gram) to boost the relevance.  Also consideration should be given to specifying tie=1.0 to sum the score for the edismax.  The default behavior is to only use the score from the highest scored field.  May also want to consider leveraging the 'mm' parameter in edismax.  During the prototype, we also 'rewrote' the query to limit the number of edismax fields to 4 (instead of the 12 or so for all, 8 for allmedium, and 5 for allsmall).  This did not impact the results returned for a query, only the scoring. As a general rule of thumb, no more than 4 fields should be used in an edismax.  A similar approach that was used with FAST (rank profiles) should be leveraged.  Consideration should also be given to using 'bq' with some date math (obviously based on a date) to provide a freshness boost (if that is needed for Scopus).

## Directory Factory

Currently, the default Directory Factory is the NRTCachingDirectory.  Since we don't plan on updating the live index in place, there is no need for NRT operations.  Subsequently, consideration should be given to using the MMapDirectoryFactory instead.

## Leverage Analyzers/Filters

Currently, Unity does some work that could likely be handled through an analyzer (and filter).  For example, the charFilter (PatternReplaceCharFilterFactory) can be easily used to remove punctuation from an ISSN/ISBN search term (something which the Unity layer does today.  This is necessary because the ISSN/ISBN is stored with the punctuation removed.  Another charFilter was mentioned above to remove XML from a field.    There are likely other things that could be leveraged in this area to prevent you from needlessly complicating the implementation logic.

## Preload index into system cache

All of the official tests run as part of the PoC leveraged a lazy loading of the index into system cache as it was required by Solr to resolve queries. A potential alternative to this approach would be to force the system to pre-load the index into OS cache prior to starting Solr on each node. The following command could be incorporated into the provisioning scripts to implement the pre-load. Here is the command to do this:

find $SOLR\_INSTALL/example/solr/\*/data/ -type f –exec cat {} \; > /dev/null

This command could be used to alleviate some of the searcher warm up delay when getting a new index ready for the flip. When we tried it on the core cluster, we did not see a noticeable impact on ongoing performance.

## Garbage Collection

Close attention should be given to garbage collection times and their overall impact on Solr. During the tests we observed a few “stop the world” garbage collection cycles. While we did some investigation during the PoC, this is an ongoing area of research/investigation. We are currently using the ParallelGC based on recommendations by Search Technologies.

## Highlighting

The default simple highlighter was used for the PoC to highlight the abstract field (the only field required for highlighting). If the default simple highlighter does not meet the needs for highlighting other options are available (even the option for developing a custom highlighting solution).  The FastVectorHighlighter should not be used as it will significantly increase the size of the index.  Consideration could be given to the new Solr PostingsHighlighter.  While it will increase the size of the index, the impact will not be as significant as the FastVectorHighlighter.

## Scope Queries

I will provide some text for this section tomorrow.

# Evaluation Matrix

**Total Possible Score Categories:**

**3 = for items of low importance**

**5 = for items of medium importance**

**7 = for items of medium-high importance**

**10 = for items of high importance**

| **Criteria** | **Total Possible Score** | **SOLr** |
| --- | --- | --- |
| Meets or Exceeds Performance Requirements | **10** |  |
| Meets or Exceeds Functional Requirements | **10** |  |
| Meets or Exceeds Answer Count Requirements | **7** |  |
| Provides Quality Relevance Results | **7** |  |
| Costs (of final solution) | **10** |  |
| Time to Market | **7** |  |
| Operational Challenges | **5** |  |
| Total Score | **56** |  |

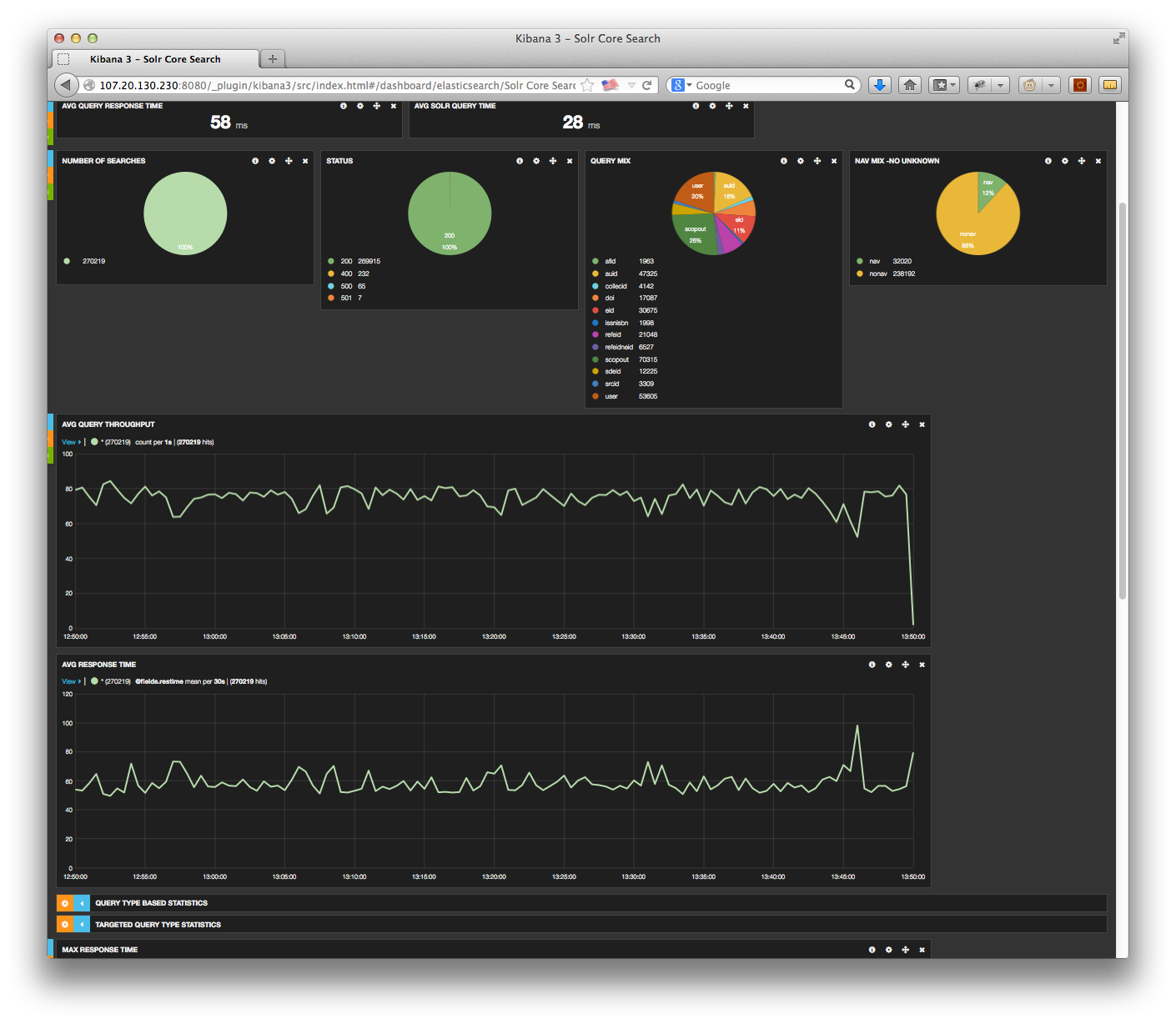
1. **Running across different Availability Zones**

As noted in the document, all of our official PoC load tests were run with the Solr cluster and load generator running within the same Amazon availability zone. While this made for a well contained PoC, it is not representative of the proposed production deployment architecture where HA is achieved with replicating shards across AZs. Since SolrCloud views every shard replica as equivalent for querying, and currently lacks “rack awareness”, there is no way to ensure that Solr will be smart and leverage “local” replicas to resolve the search query. As a result, there is the likelihood of increased cross AZ communication between Solr nodes. This adds additional overhead onto query resolution as well as adds a small AWS cost based on the amount of traffic being sent between zones. In an attempt to simulate this cross AZ traffic, we instantiated a cluster with shards split between 2 availability zones and ran our standard core load test against it to quantify the performance impact.

In all, there was a slight increase in average response time and decrease in query throughput as a result of the cross zone communication. Average query time increased from 47ms to 58 ms. Average query throughput decreased from 92 qps to 76 qps. Both of these new numbers are still within the specifications laid out in the Scopus requirements document.

For reference, the full test results are located in Kibana. The load test run occurred on 5/7/2014 from 12:50 EDT to 13:50 EDT.

Figure 9 Cross Availability Zone Load Test -1 Hour Snapshot (5/7/2014 12:50 EDT – 13:50 EDT)

****

****

1. **Redshift processing for global statistics.**

All of the testing was done on the least expensive instance available in RedShift (dw2.large) and on a single node.  This translates into a RedShift cost of $.25/hr or $6 a day.

|  |  |
| --- | --- |
| **Statistic generated** | **Elapsed Time** |
|  |  |
| Affiliation Pub count | Data not available. Anticipated similar to authors as amount of records much smaller. |
| Author Pub Count | 1 minute |
| Core Cited By Count | 6 minutes |
| CitRef mappings for Cores | 34 minutes |

**Generating Publication Counts for Authors - Elapsed processing time of 1 minute to generate results**

166,246,342 records in corestoauthors

27,865,734 records in authorpubcnt

15,571,914 records in authorpubcnt have a pubcnt of 1

// Table to hold the mappings of published documents to authors

CREATE TABLE corestoauthors(

eid VARCHAR(20),

auid VARCHAR(20) SORTKEY,

PRIMARY KEY(eid,auid));

// Table to hold the calculated publication count for each author

CREATE TABLE authorpubcnt(

auid VARCHAR(20) SORTKEY,

pubcnt INTEGER,

PRIMARY KEY(auid));

// Calculate the publication count for an author

INSERT INTO authorpubcnt(auid,pubcnt)

SELECT auid, COUNT(\*)

FROM corestoauthors

GROUP BY auid

ORDER BY auid;

Takes 48.83s

// Copy the file to S3 so the search engine can pick it up

UNLOAD ('

SELECT \* FROM authorpubcnt

ORDER BY auid') to  's3://els-ats/scopuscnts/pc/author/darin/auth-' CREDENTIALS 'aws\_access\_key\_id=;aws\_secret\_access\_key=' delimiter '=' GZIP;

Creates (2) zip files each about 51MB in size. Single compressed file of 84MB.   Uncompressed 382MB.

Takes less than 1s.

**Generating Cited By Counts for Cores - Elapsed processing time of 6 minutes to generate results**

 759,190,044 records in corestoreferences

 141,752,199 records in corecbcnt

   99,575,432 records in corecbcnt have a cbcnt=1

// Table to hold the mappings of published documents to citing documents

CREATE TABLE corestoreferences(

eid VARCHAR(20),

refid VARCHAR(20) SORTKEY,

PRIMARY KEY(eid,refid));

// Table to hold the cited by count count for each document

CREATE TABLE corecbcnt(

refid VARCHAR(20) SORTKEY,

cbcnt INTEGER,

PRIMARY KEY(refid));

// Calculate the cited by count for a document (core or dummy)

INSERT INTO corecbcnt(refid,cbcnt)

SELECT refid, COUNT(\*)

FROM corestoreferences

GROUP BY refid

ORDER BY refid;

Takes  4m 54s

// Copy the file to S3 so the search engine can pick it up

unload ('

SELECT \* FROM corecbcnt

ORDER BY refid') to  's3://els-ats/scopuscnts/cbc/core/darin/core-' CREDENTIALS 'aws\_access\_key\_id=;aws\_secret\_access\_key= ' delimiter '=' GZIP;

Creates (2) zip files each about 231MB in size.  Single compressed file of 404MB.  Uncompressed 1.9GB.

Takes  57s

**Generating CitRef for Cores: Elapsed processing time of 34 minutes to generate results**

**In contrast to calculating publication and cited by counts (where the entire file is calculated from scratch), this approach**

**assumes tables will be maintained for the previous day (n-1) and the current day (n).  A delta will then be calculated**

**between the 2 tables to identify those records that would need to be updated with the new citref information.**

// Create the table

to hold the mappings of published documents to citing documents

**// Strip the prefix form the eid so it can be stored as a decimal field**

**CREATE TABLE citref(**

**neid DECIMAL,**

**refid VARCHAR(20) SORTKEY,**

**PRIMARY KEY(neid,refid));**

**// Insert the records (make a numeric version of the eid)**

**INSERT INTO citref(**

**SELECT CAST(SUBSTRING(eid,8) AS DECIMAL) AS neid, refid**

**FROM corestoreferences);**

// Table to hold the sum (think of this as a hash)

**CREATE TABLE citrefsum(**

**refid VARCHAR(20) SORTKEY,**

**tot DECIMAL(38,0),**

**PRIMARY KEY(refid));**

// Group and Sum

**INSERT INTO citrefsum(**

**SELECT refid, SUM(neid) AS tot**

**FROM citref**

**GROUP BY refid**

**ORDER BY refid);**

**Takes 5m 39s**

**Need to do the above twice (we will have an old and new version of these tables)**

// Table to hold the delta

**CREATE TABLE citrefdelta(**

**refid VARCHAR(20) SORTKEY,**

**tot DECIMAL(38,0),**

**PRIMARY KEY(refid));**

// Calculate the delta (to identify the records that need to have their citref records updated)

**INSERT INTO citrefdelta(**

**SELECT \* FROM citrefnew**

**MINUS**

**SELECT \* FROM citrefold);**

**Takes 2m 51s**

**// Table to hold the results**

**CREATE TABLE citrefresults(**

**refid VARCHAR(20) SORTKEY,**

**eid VARCHAR(20),**

**PRIMARY KEY(refid,eid));**

// Go go get the records that have changed

**INSERT INTO citrefresults(**

**SELECT citrefdelta.refid, corestoreferences.eid  FROM citrefdelta, corestoreferences**

**WHERE citrefdelta.refid= corestoreferences.refid**

**ORDER BY citrefdelta.refid);**

**Takes 23m 28s**

Note, this time is not a real example as both tables were the same when I ran the test (I did not have a citrefnew and citrefold).

So, essentially all the records were considered new.

1. **Accurate Facet Counts**

A very good discussion of how facets are calculated in Solr.

|  |  |  |
| --- | --- | --- |
| **Subject:** | [At a high level how does faceting in SolrCloud work?](http://markmail.org/message/pjqkbyfygc7fykvt) | [ermalink](http://markmail.org/message/pjqkbyfygc7fykvt) |
| **From:** | Jamie Johnson (jej2...@gmail.com) | |
| **Date:** | Sep 24, 2012 9:04:17 am | |
| **List:** | org.apache.lucene.solr-user | |

I'd like to wrap my head around how faceting in SolrCloud works, does Solr ask each shard for their maximum value and then use that to determine what else should be asked for from other shards, or does it ask for all values and do the aggregation on the requesting server?

|  |  |  |
| --- | --- | --- |
| **Subject:** | [Re: At a high level how does faceting in SolrCloud work?](http://markmail.org/message/dyokvy3ixuquez5g) | [ermalink](http://markmail.org/message/dyokvy3ixuquez5g) |
| **From:** | Chris Hostetter (hoss...@fucit.org) | |
| **Date:** | Sep 26, 2012 3:21:20 pm | |
| **List:** | org.apache.lucene.solr-user | |

: I'd like to wrap my head around how faceting in SolrCloud works, does : Solr ask each shard for their maximum value and then use that to : determine what else should be asked for from other shards, or does it : ask for all values and do the aggregation on the requesting server?

For things like facet.range and facet.query, it's really simple: each shard is given the facet params, each shard computes an identical list of constraints (they are deterministic) and returns them to the coordinator, who sums the counts.

For facet.field things are a little more interesting...

The overall approach taken is designed to ensure that the constraint counts you get back are always correct (w/o requiring each shard to return the fill list of terms it knows about) even if the document distribution is really skewed and the constraints returned aren't the constraints with the highest count across the entire index.

how it works (from memory) is basically..

1) each shard is asked for it's top constraints using an artificitlaly inlated "facet.limit" (i think it's something like 20 + 1.5 the original limit)

2) the coordinator node sums up the counts for any constraint returned by multiple nodes, and then picks the top (facet.limit) constraints based n the counts it knows about.

3) the coordinator node then asks each shard to compute it's exact count for the selected constraints (since some of those constraints may not have been in the original lists returned by some shards), and it then computes the final sum. This ensures that the constraint count will match the numFound if filter on that constraint (but i believe this is second query is optimized to only ask a shard about a constraint if it didn't already get the count in the first request)

So imagine you have 3 shards, and querying them individually with facet.field=cat&facet.limit=3 you get...

shardA: cars(8), books(7), computers(6) shardB: toys(8), books(7), garden(5) shardC: garden(4), books(3), computers(3)

If you made a solr cloud query (or an explicit distributed query of those three shards), the first request the coordinator would send to each shard would specify a higher facet.limit, and might get back something like...

shardA: cars(8), books(7), computers(6), cleaning(4), ... shardB: toys(8), books(7), garden(5), cleaning(4), ... shardC: garden(4), books(3), computers(3), plants(3), ...

...in which case "cleaning" pops up as a contender for being in the top constraints. The coordinator sums up the counts for the constraints it knows about, and might decide that these are the top 3...

books(17), computers(9), cleaning(8)

...at which point it sends a second query to shardB asking for an explicit count of constraint "computers" and to shardC for a count of the constraint "cleaning". so that the final results can be exact. when the responses come back, those counts are added in to the totals the coordinator already knows about. So in our example, if the results of the second query are...

shardB: computers(0) shardC: cleaning(2)

..then the final top 3 constriants for the cat field might be...

books(17), cleaning(10), computers(9)

(Note that if we had just done a really naive sum of the original counts, "cleaning" wouldn't have even made the list)

-Hoss

|  |  |  |
| --- | --- | --- |
| **Subject:** | [Re: At a high level how does faceting in SolrCloud work?](http://markmail.org/message/lkl7ffi77w7hpv6n) | [ermalink](http://markmail.org/message/lkl7ffi77w7hpv6n) |
| **From:** | Yonik Seeley (yon...@lucidworks.com) | |
| **Date:** | Sep 26, 2012 9:14:54 pm | |
| **List:** | org.apache.lucene.solr-user | |

On Wed, Sep 26, 2012 at 6:21 PM, Chris Hostetter <hoss...@fucit.org> wrote:

2) the coordinator node sums up the counts for any constraint returned by multiple nodes, and then picks the top (facet.limit) constraints based n the counts it knows about.

It's actually more sophisticated than that - we don't limit to the top facet.limit constraints at the first phase. For \*all\* constraints we see from the first phase, we calculate if it could possibly be in the top facet.limit constraints (based on shards we haven't heard from). If so, we request exact counts from those shards we haven't heard from.

(but i believe this is second query is optimized to only ask a shard about a constraint if it didn't already get the count in the first request)

Correct.

So imagine you have 3 shards, and querying them individually with facet.field=cat&facet.limit=3 you get...

shardA: cars(8), books(7), computers(6) shardB: toys(8), books(7), garden(5) shardC: garden(4), books(3), computers(3)

If you made a solr cloud query (or an explicit distributed query of those three shards), the first request the coordinator would send to each shard would specify a higher facet.limit, and might get back something like...

shardA: cars(8), books(7), computers(6), cleaning(4), ... shardB: toys(8), books(7), garden(5), cleaning(4), ... shardC: garden(4), books(3), computers(3), plants(3), ...

...in which case "cleaning" pops up as a contender for being in the top constraints. The coordinator sums up the counts for the constraints it knows about, and might decide that these are the top 3...

books(17), computers(9), cleaning(8)

To extend your example, Solr notices that "plants" has a count of 3 on one shard, and was missing from the other two shards. The maximum possible count it \*could\* have is 11 (3+4+4), which could possibly put it in the top 3, hence it will also ask shardA and shardB about "plants".

-Yonik <http://lucidworks.com>

1. **Scoped Queries**

**Copy email exchange here**

1. **Leveraging Solr index for CTO query resolution.**

**Still to write**