**MarkLogic**

## Solution Description

All work detailed here was conducted using MarkLogic 7.0-2.2, the latest official release at the time of the evaluation.

We based all development/sizing on the Elsevier Scopus Search Prototype Version 1.0 (March 17,2014) document. If these requirements change, this must be taken into consideration should the MarkLogic approach be adopted.

### Document Schema

Because MarkLogic's strength is its ability to maintain ACID properties across search indexes and XML at the same time, this implementation stored and indexed the Scopus XML in its original form. This contrasts with the ScienceDirect HotHouse where the original XML was not retained and only necessary fields for searching (and hit list retrieval) were created.

For certain fields, documents were enriched with extra elements. These extra elements serve much the same purpose as function-based indexes in an RDBMS. That is, they allow indexing based on content that is not in the original document at all, or is not in the desired format for indexing.

* Affiliation
  + affilctry
  + count
  + fastloaddate
  + empty parent (root)
* Author
  + autafnameidnav
  + count
  + fastloaddate
  + preffirstsort
  + active
* Core
  + authsort
  + authgrpid
  + datesort
  + dateloaded
  + numcitedby
  + fastloaddate
  + prefnameauid
  + refeid
  + refpubyr
  + subjabbr

These enrichment elements were generated at ingestion time, using XQuery code. They were stored in a document wrapper, leaving the original Scopus XML unchanged.

MarkLogic implements full ACID transactions, so both the original XML and the searchable field-based structure could be updated in atomic transactions.

### Database configuration

The index model in MarkLogic is fundamentally different from the approach taken by legacy search engines. Rather than requiring fields to be defined, MarkLogic automatically indexes all input XML. Because the goal of this project was to duplicate Scopus functionality, however, we used various MarkLogic features to customize its behavior.

By default a MarkLogic word query will match any word in any XML element. This corresponds reasonably well to the Scopus “all” field, but that field only needs to index a subset of the XML. So we configured MarkLogic's built-in word query field to exclude by default, and included only the elements we wanted. This probably reduced the on-disk size of the database somewhat. This also requires great care to ensure that all of the necessary elements defined in the CIP for the “all” field are included for indexing.

Next we mapped each Scopus field to a MarkLogic query type, and where necessary an index configuration. Some Scopus fields could be implemented using simple element-word queries or element-value queries. In some cases we scoped these queries by requiring a particular parent element (element-queries).

For facets and sort keys (and to further improve performance for some queries), we created MarkLogic range indexes (element, element/attribute, path). These are value indexes, and include frequency information for facets. The default MarkLogic indexes are all hash-based, and cannot be used for facets or sorting.

quality

ref-id

institution-profile/@affiliation-id

institution-profile/@parent

citation-type/@code

citation-language/@lang

source/@type

status/@type

/xocs:doc/xocs:institution-profile/institution-profile/address/city

/xocs:doc/xocs:institution-profile/institution-profile/certainty-scores/certainty-score/orig-id

/xocs:doc/xocs:institution-profile/institution-profile/certainty-scores/certainty-score/score

/xocs:doc/xocs:institution-profile/institution-profile/sort-name

/xocs:doc/xocs:author-profile/author-profile/affiliation-current/affiliation/ip-doc

[@type='parent']/address/city

/xocs:doc/xocs:author-profile/author-profile/affiliation-current/affiliation/ip-doc

[@type='parent']/address/city-group

/xocs:doc/xocs:author-profile/author-profile/affiliation-history/affiliation/ip-doc

[@type='parent']/address/city

/xocs:doc/xocs:author-profile/author-profile/affiliation-history/affiliation/ip-doc

[@type='parent']/address/city-group

/xocs:doc/xocs:author-profile/author-profile/affiliation-current/affiliation/ip-doc

[@type='parent']/address/country

/xocs:doc/xocs:author-profile/author-profile/affiliation-history/affiliation/ip-doc

[@type='parent']/address/country

/xocs:doc/xocs:author-profile/author-profile/journal-history/journal/sourcetitle

/xocs:doc/xocs:author-profile/author-profile/@id

/xocs:doc/xocs:author-profile/author-profile/preferred-name/given-name

/xocs:doc/xocs:author-profile/author-profile/preferred-name/initials

/xocs:doc/xocs:author-profile/author-profile/preferred-name/surname

/xocs:doc/xocs:author-profile/author-profile/classificationgroup/classifications

[@type='SUBJABBR']/search-classification/code

/xocs:doc/xocs:item/item/bibrecord/head/author-group/affiliation/@country

/xocs:doc/xocs:item/item/bibrecord/head/author-group/affiliation/@afid

/xocs:doc/xocs:item/item/bibrecord/head/author-group/author/@auid

/xocs:doc/xocs:item/item/bibrecord/item-info/itemidlist/itemid[@idtype='SCP']/@idtype

/xocs:doc/xocs:meta/xocs:eid

/xocs:doc/xocs:item/item/bibrecord/head/source/issn[@type='electronic']

/xocs:doc/xocs:item/item/bibrecord/head/citation-info/author-keywords/author-keyword

/xocs:doc/xocs:item/item/bibrecord/head/enhancement/descriptorgroup/descriptors

/descriptor/mainterm

/xocs:doc/xocs:item/item/bibrecord/head/enhancement/ tradenamegroup/tradenames

/trademanuitem/tradename

/xocs:doc/xocs:item/item/bibrecord/head/enhancement/chemicalgroup/chemicals

/chemical/chemical-name

/xocs:doc/xocs:item/item/bibrecord/head/source/sourcetitle

/xocs:doc/xocs:item/item/bibrecord/head/citation-title/titletext

/xocs:doc/xocs:item/item/bibrecord/item-info/itemidlist/itemid[@idtype='PUI']

/xocs:doc/xocs:item/item/ait:process-info/ait:date-sort/@year

/xocs:doc/xocs:item/item/bibrecord/tail/bibliography/reference/ref-info/refd-itemidlist/itemid

[@idtype='SGR']

For the Scopus allsmall field, we found it advantageous to create a MarkLogic field specific to its constituent elements. This was the only extra field that we created in MarkLogic, but other Scopus fields would likely benefit from this treatment. Other candidates we considered included allmed and ref. Treating more Scopus fields as MarkLogic fields would tend to increase ingestion time and the size of the database on disk, but might improve the performance of some queries.

The current MarkLogic solution has only been tested for search queries. Since the solution will need to service search queries as well as other Content Metadata Service queries (such as CTO, etc.) and retrieval queries, additional configuration changes will be required. With the current MarkLogic Scopus database at 4.8TB (spread across 32 forests and 4 nodes), the overall database size will increase.

### Ingestion (and Query Environment)

Because of the need to maintain affil.count, auth.count, and core.numcitedby, we chose to use the same environment for ingestion and query. This environment could have been duplicated in various ways to provide high availability (HA), disaster recovery (DR), and/or horizontal scalability.

The MarkLogic cluster consisted of four hi1.4xlarge PVM instances. Each of these instances has 16 cores, 60-GiB RAM, and 2x1024-GB ephemeral SSD storage. This instance type was chosen for its balance of relatively high CPU, RAM, and IOPS. Alternatives might include the c3.8xlarge with PIOPS EBS storage, or i2.8xlarge with a similar SSD configuration.

The main points of the loading setup are:

* Ingestion is mapped across every instance (32) in the MarkLogic cluster
* Each instance's built-in Task Server manages a queue of jobs (documents to load)
* Each instance ingests only the keys in its shards
* Key files ingested and spread across the cluster instances
* Retry logic for S3 downloads and document ingestion
* Transformation logic based on JSON definition derived from CIP and implemented in XQuery

Loading the base 108M documents (50.8M cores, 49.7M authors, and 8.1M affiliations) without citation counts took 22 hours and 23 minutes. Calculating the citation counts took an additional 17 hours. We designed a strategy that should reduce this to less than 5 hours, but were unable to test it during the evaluation period

We had also originally planned on using the MarkLogic property axis to store the counts and numcitedby numbers (so the entire document would not need to be updated for a change in these values). However, when doing search load testing, we found that the underlying inherent join associated with the property axis significantly impacted performance. We were forced to abandon this approach and instead incorporate the counts into the actual document (requiring an update to the entire document when the count changes).

#### High Availability

Because the SSD storage is ephemeral, we considered how to handle instance failures in a production environment. One option would be to scale horizontally across N+1 clusters, where N clusters are capable of handling production load. A single instance failure would leave N clusters, so production response times would not suffer. After any failure a new instance would be brought online and its SSD populated with the current data.

We experimented with backing up the database to S3, and restoring it from S3. Backup took 3-4 hours, and restores took 7-8 hours. So in many cases a restore from S3, followed by any pending updates, could be faster than rebuilding from source XML.

Another approach would be to increase the cluster size to accommodate data replication within the cluster. As with a RAID 1 array, each host would mirror some data from other hosts. We calculate that a cluster with single redundancy would require six of these instances. If an instance failed, it could be replaced with a new instance, which would copy its data from the remaining mirrors. We did not prototype this strategy, but anticipate that full recovery would take at least an hour (if not more).

The choice of instance failure strategy would depend on the total query capacity needed, recovery time expectations, and the total costs involved.

#### Kernel tuning

It was necessary to do some tuning of the kernel to achieve the best results using MarkLogic. This is based on the underlying hardware configuration and the behavior seen during our testing, which is subject to modification upon any other changes to the setup. A list of specific changes can be found in Appendix C – Further Technical Details

The network related changes are aimed at improving throughput for the “whale” requests, those requesting 2000-4000 results in a single request. The remainder was largely to allow MarkLogic to work unhindered by OS overheads.

The dirty bytes tuning is specifically for tuning the behavior of the OS to prevent building up large backlogs of buffered writes, which is critical for loading/updates but irrelevant for search.

### Search Query Specifics

**Field Weightings –** As part of our mapping of the Scopus fields to MarkLogic, we encoded the relative field weights as specified in the PoC requirements. We then translated the relative field weightings into both our database configuration, as part of MarkLogic fields, and into XQuery used to construct the queries. Any filter terms were weighted at zero, removing any impact on relevance scoring.

**Related Articles –** The built-in MarkLogic cts:similar-query function was not needed. While there are ‘related article’ queries (based on refeid, authid, and keywords), the client has already constructed the list of query terms (authid, refied, kewyords) that should be used for the ‘related articles’ query. There is really nothing special required of the search engine (other than sorting the results by relevancy).

**Stemming –** We used the built in support for stemming terms set to the “basic” level. It is worth noting that a database wide default language can be set, however this is overridden by any xml:lang attributes. Indexing is always language-aware, whether stemming is enabled or not. If some searchable fields require different values for stemming (than defined in the default database configuration), then a separate MarkLogic field will need to be constructed for each of these fields in order to adjust the default behavior defined in the database configuration. This will impact the overall size of the MarkLogic database.

**Phrase Through** – We are heavily leveraging cts:element-word-query as part of the query execution. This requires database level configuration to identify children elements that should have their text descendants included in the query. If some searchable fields require a different setting than was configured in the database configuration, a separate MarkLogic field would need to be constructed for each of these fields. This will impact the overall size of the MarkLogic database.

**Wildcards –** We used the built in support for wildcarding (an option to cts:query constructors). Note that wildcarded words do not get matched against stems, even when the stemming option is provided.

**Query database –** to handle the set of load test queries we added a second database in the MarkLogic cluster placed onto one of the four machines only. This database held the MarkLogic transformed XML representation of the load test queries, which were loaded at the point of creating the servers through a similar load and transformation steps to the main content. Upon a call being made the XML representation of the query is retrieved from this database and then executed. This in itself may have a small impact on performance and is not something that would be carried forward in to production as it was used purely to support the load testing.

**Stop Words – S**top words were eliminated from query terms, unless the phrase options were specified. MarkLogic itself does not implement stop words as an indexing feature, so the indexes contained every word.

**Spelling suggestions –** This is typically an externally driven process rather than left to the engine to do by itself. No explicit support was coded for, however MarkLogic provides support for dictionaries to be loaded and used for spelling suggestions through the built-in spell functions. The only requirement is to provide a dictionary to work from; it will not work directly from the indexes produced. With a word lexicon enabled however it would be possible to extract a list of words from the index to populate a dictionary, with some threshold applied to the terms to be used. The suggestions are calculated using Levenshtein distance measure with an upper limit of 64 characters to be included. The number of suggestions returned and their distance from the input value can be controlled. This could be further combined with frequency information from word value lookup combined with queries to further rank any suggestions.

### Search Customizations

#### Possible Improvements

It was unclear on the expected behavior for handling wildcards and stemming. Currently MarkLogic does not apply stemming to wildcard expansions, although it is possible to create such behavior by pre-expanding the wildcard terms (using the cts:\*word-match functions) to produce a list of terms to replace the original, which when executed will have the normal stemming rules applied. This will come at some cost (possibly significant) because of the number of terms this may expanded to, particularly when the wildcard word is in a phrase as it will produce a set of phrases to be found.

## Architecture Diagram of POC



MarkLogic Cluster



Cores

Authors

Affiliations

Backups



SQS Messages populated by XFab for documents to index

S3

S3

S3

S3

EC2

EC2

EC2

EC2

Elastic Load

Balancer



User

## Performance Testing

Since MarkLogic was configured as a single database (containing all 3 content types) we must load test all 3 content types concurrently. This is different than SOLR where each of the content types was a separate index on different hardware. In addition to this concurrent testing, we also tested each of the content types in isolation. This isolated testing can help identify how each content type performs independently as well as the impact of the concurrent testing on each content type performance. It would be possible to separate the content into different databases (and hardware) but this would result in increased costs due to the need for additional MarkLogic licenses.

While the MarkLogic architecture assumes an update in place, the feasibility of this approach was never tested (nor the impacts to query performance analyzed). Because the existing hardware configuration could not meet the QPS and response time requirements (as detailed below), it was decided there was really little value in testing the incremental updates while concurrently performing a query load test. Furthermore, the MarkLogic architecture not only needs to support the search query load but also the Content Metadata Service query load and content retrieval load. Neither of these loads were tested concurrently (nor was it possible) while performing the search query load test. Should MarkLogic be considered as an option, additional testing along these lines would be strongly encouraged to provide proper estimates for the hardware sizing (and the associated cost). It should be noted that additional tuning/optimization could have been done on the database (and queries) to bring in the performance numbers, but we were constrained by a fixed budget for the MarkLogic PoC. We would also have been dependent on a couple bug fixes from MarkLogic (that we identified through testing) to ensure accurate testing.

The PoC required a ‘query’ endpoint written in XQuery and hosted on a MarkLogic HTTP Application Server on each cluster instance. This ‘query’ endpoint retrieved the ‘pre-built’ MarkLogic query from the MarkLogic query database, executed the query, and then returned the results. This in itself may have a small impact on performance and is not something that would be carried forward in to production as it was used purely to support the load testing.

The following sections first detail the results for the load testing of each content type in isolation. The last section details the results for the concurrent load testing (mix) of all 3 content types together.

### Affiliation Performance Testing

For the affiliation query set, we had a target average rate of 4 qps (with a maximum of 8 qps). We had specific average response times for the identified query types (navigators and without navigators). We ran one load test box (m1.xlarge) with one copy of the load test script. The load test box was started in the same region/zone as the MarkLogic cluster (us-east-1c). We did not use an ELB but instead used a simple round-robin routing to distribute requests across the 4 machines in the MarkLogic cluster. The script ran for 15 minutes to warm the cluster before official measurements were taken. We were able to far exceed most of the target Scopus requirements, reaching 32 QPS on average. In fact, we exhausted our test query set in under the hour allocated for the test. The one area of deviation is the MarkLogic solution exhibited a larger max response time (for navigators) when compared to the Scopus target.

Table : Affiliation Query Types

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Number of queries** | **Percentage of mix** |
| Navigator | 1,031 | 1% |
| No Navigator | 90,495 | 99% |

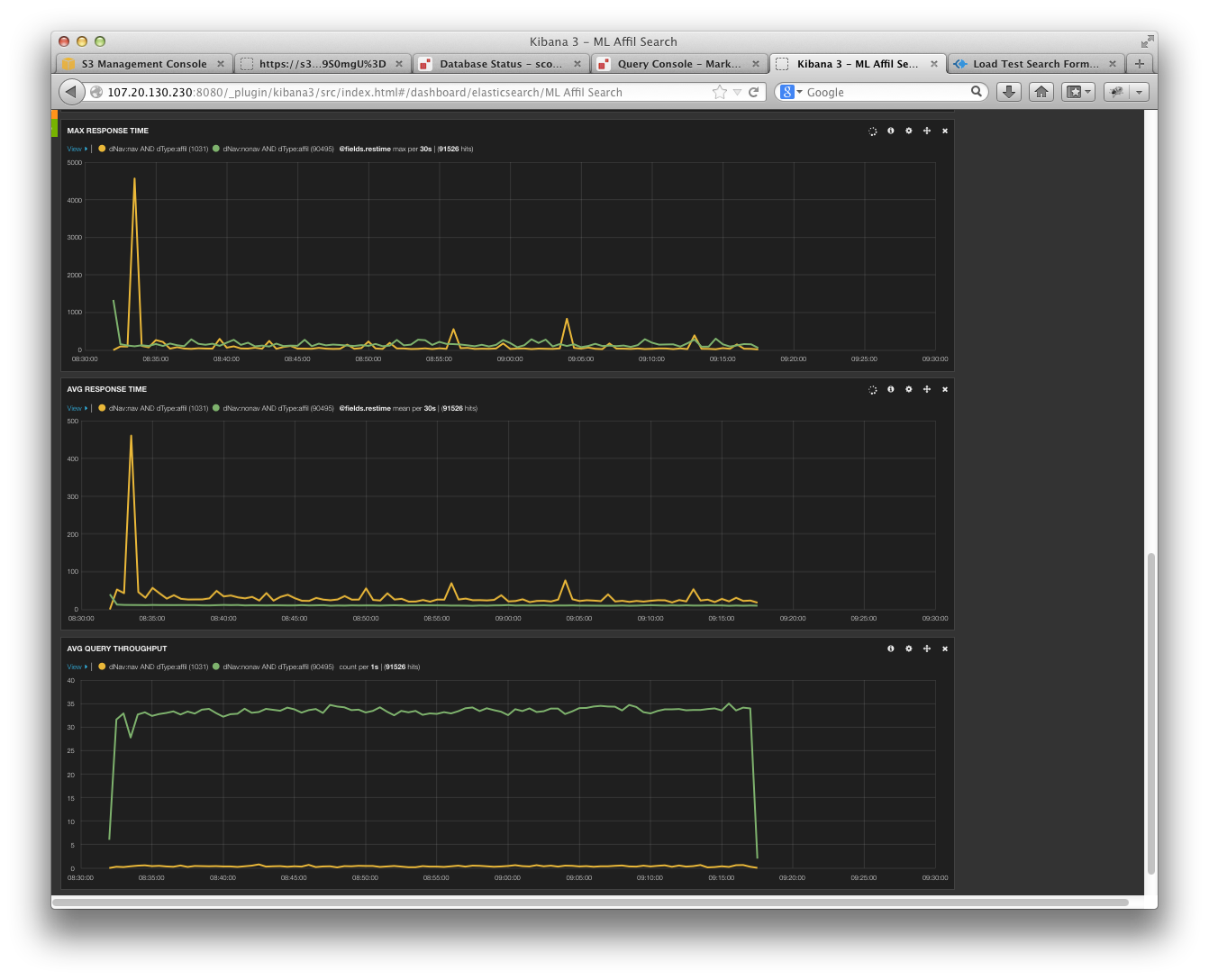
Table: Affiliation Query Types (overall averages and overall maximum)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |
| ML Observed | 32 qps | 35 qps | 11 ms | 4,564 ms |

Table: Scopus Targets and Observations for Affiliation Query Types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **Mean** | **Median** | **P95th** | **Max** |
|  |  |  |  |  |
| Scopus Target - Navigator | 129 ms | 103 ms | 243 ms | 979 ms |
| ML Observed - Navigator | 18 ms | 10 ms | 43 ms | 4,564 ms |
|  |  |  |  |  |
| Scopus Target - No Navigator | 95 ms | 70 ms | 180 ms | 3,643 ms |
| ML Observed - No Navigator | 11 ms | 8 ms | 24 ms | 1,335 ms |

Figure : Affiliation Query – 1 hour Snapshot



### Author Performance Testing

For the author query set, we had a target average rate of 9 qps (with a maximum of 15 qps). We had specific average response times for the identified query types (navigators and without navigators). We ran one load test box (m1.xlarge) with one copy of the load test script. The load test box was started in the same region/zone as the MarkLogic cluster (us-east-1c). We did not use an ELB but instead used a simple round-robin routing to distribute requests across the 4 machines in the MarkLogic cluster. The script ran for 15 minutes to warm the cluster before official measurements were taken. We were able to exceed the target Scopus requirements, reaching 32 QPS on average.

Table : Author Query Types

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Number of queries** | **Percentage of mix** |
| Navigator | 10,107 | 15% |
| No Navigator | 59,302 | 85% |

Table: Author Query Types (overall averages and overall maximum)

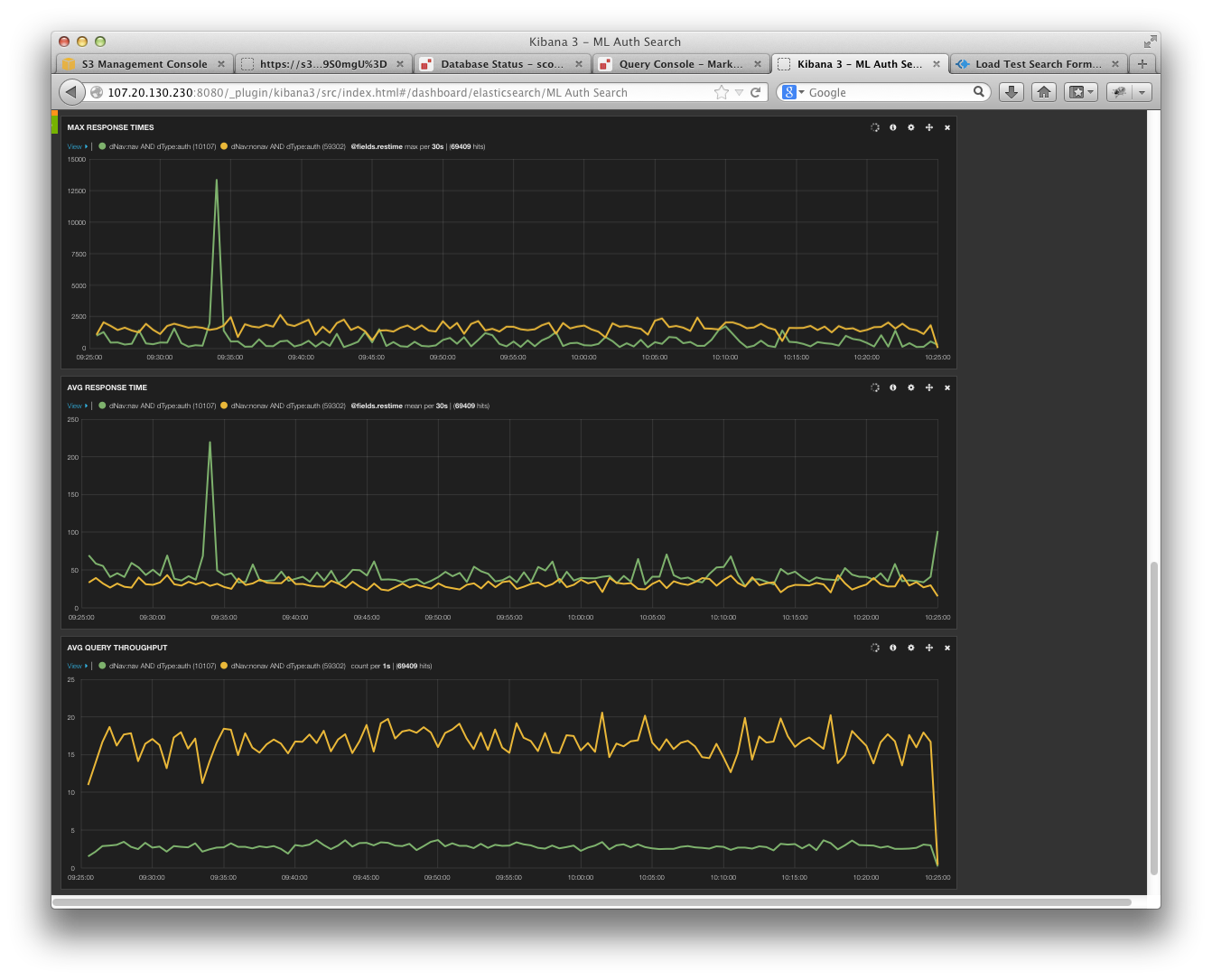
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **AVG Request Rate** | **Max**  **Request**  **Rate** | **Avg Response Time** | **Max Response Time** |
|  |  |  |  |  |
| Scopus Targets | 9 qps | 15 qps | N/A | 29,091 ms |
| ML Observed | 18 qps | 24 qps | 33 ms | 13,341 ms |

Table: Scopus Targets and Observations for Author Query Types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **Mean** | **Median** | **P95th** | **Max** |
|  |  |  |  |  |
| Scopus Target - Navigator | 242 ms | 101 ms | 1,013 ms | 29,091 ms |
| ML Observed - Navigator | 36 ms | 14 ms | 68 ms | 13,341 ms |
|  |  |  |  |  |
| Scopus Target - No Navigator | 131 ms | 46 ms | 174 ms | 28,168 ms |
| ML Observed - No Navigator | 32 ms | 13 ms | 58 ms | 2,640 ms |

Figure : Author Query – 1 hour Snapshot





### Core Performance Testing

For the core query set, we had a target average rate of 65 qps (with a maximum of 91 qps). We had specific average response times for the identified query types (navigators and without navigators). We ran six load test boxes (m1.xlarge) with five copies of the load test script on each box. The load test boxes were started in the same region/zone as the MarkLogic cluster (us-east-1c). We did not use an ELB but instead used a simple round-robin routing to distribute requests across the 4 machines in the MarkLogic cluster. The script ran for 15 minutes to warm the cluster before official measurements were taken. We were unable to meet the target Scopus requirements for either QPS (avg or max) or response time (mean and 95th percentile). We also noticed that there were approximately 460 requests that failed with a 500 RC from MarkLogic. This is an order of magnitude higher than what we typically see during a SOLR load test run. In addition, the query mix seems to be slightly off (only 5% navigator requests instead of the 10% expected).

Table : Core Query Types

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Number of queries** | **Percentage of mix** |
| Navigator | 9,425 | 5% |
| No Navigator | 168,213 | 95% |

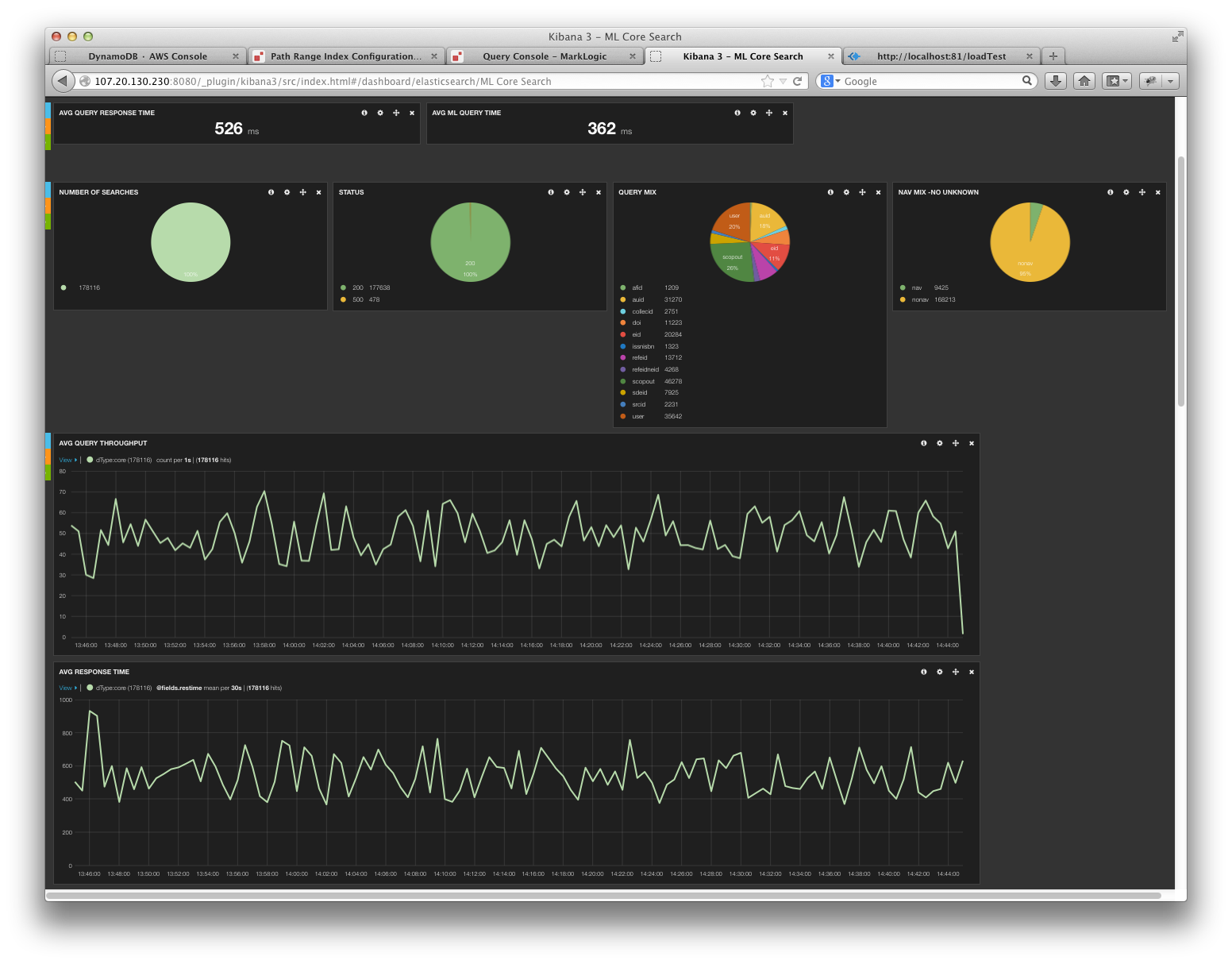
Table: Core Query Types (overall averages and overall maximum)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |
| ML Observed | 50 qps | 70 qps | 526 ms | 59,496 ms |

Table: Scopus Targets and Observations for Core Query Types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **Mean** | **Median** | **P95th** | **Max** |
|  |  |  |  |  |
| Scopus Target - Navigator | 452 ms | 259 ms | 1,022 ms | 65,058 ms |
| ML Observed - Navigator | 604 ms | 179 ms | 2,499 ms | 23,554 ms |
|  |  |  |  |  |
| Scopus Target - No Navigator | 397 ms | 171 ms | 633 ms | 179,189 ms |
| ML Observed - No Navigator | 521 ms | 157 ms | 2,081 ms | 59,495 ms |

Figure : Core Query– 1 hour Snapshot





### Mix Performance Testing

As mentioned previously, since the MarkLogic architecture defined a single database for all 3 content types (on the same hardware) it is imperative that we concurrently load test all of the content types. While the MarkLogic architecture was designed to support both updates/query within the same database/hardware, the feasibility of this approach was never tested. The mix performance testing only included query requests. Since it was not possible to achieve the performance goals (qps or response times) with just query requests, we saw little value in adding the content loading requests to the mix.

**Affiliation Mix Performance Testing**

The performance requirements and load test startup were the same as previously discussed in the earlier Affiliation Performance Testing section. While the numbers across the board (avg response time, median response time, 95th percentile response time were 2x to 5x higher than the isolation run for affiliations, there were still mostly within the requirements. The one area of deviation is the MarkLogic solution exhibited a larger max response time (for navigators) when compared to the Scopus target.

Table : Affiliation Query Types

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Number of queries** | **Percentage of mix** |
| Navigator | 535 | 1% |
| No Navigator | 48,824 | 99% |

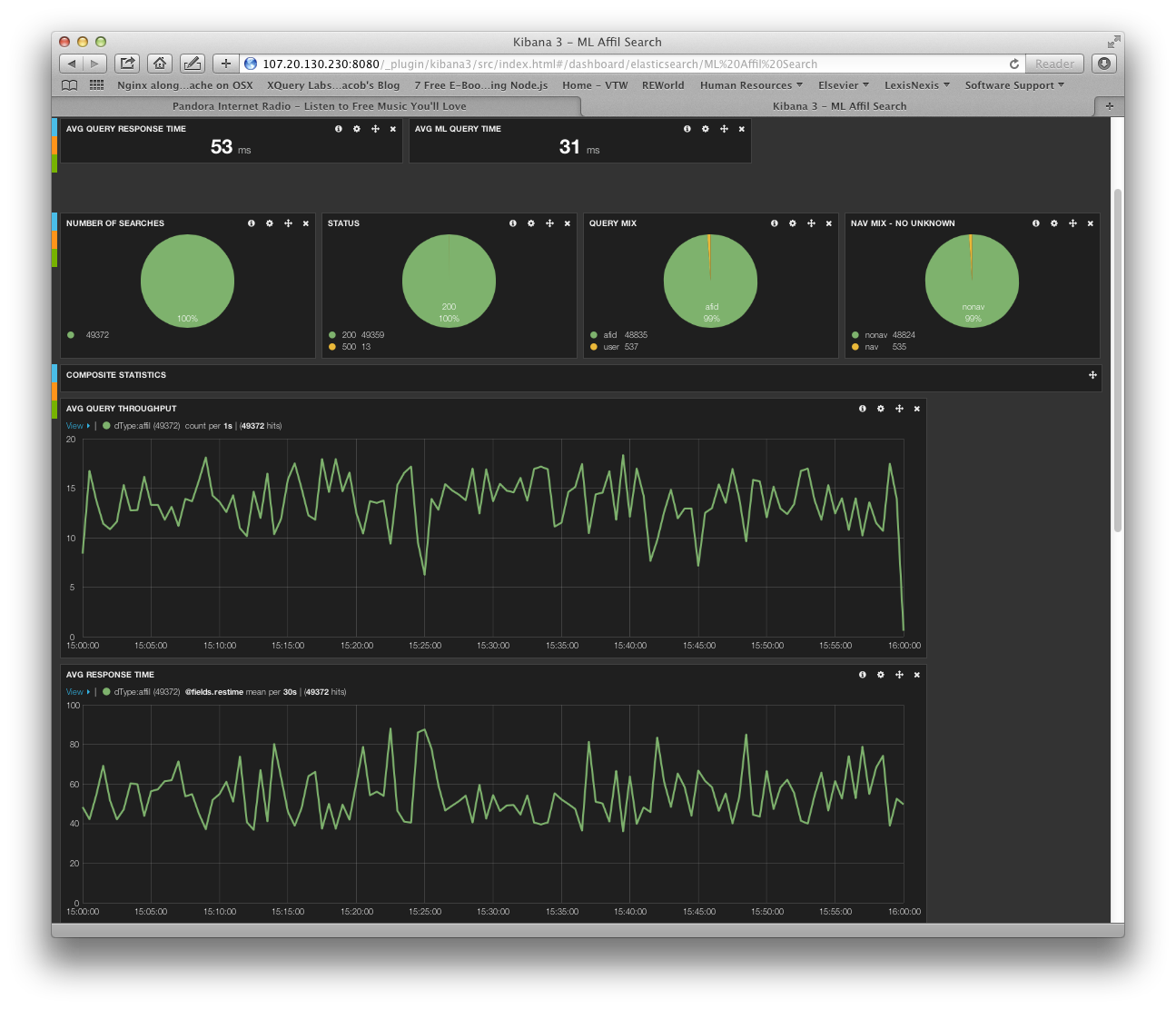
Table: Affiliation Query Types (overall averages and overall maximum)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |
| ML Observed | 13 qps | 18 qps | 53 ms | 13,372 ms |

Table: Scopus Targets and Observations for Affiliation Query Types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **Mean** | **Median** | **P95th** | **Max** |
|  |  |  |  |  |
| Scopus Target - Navigator | 129 ms | 103 ms | 243 ms | 979 ms |
| ML Observed - Navigator | 56 ms | 24 ms | 199 ms | 13,372 ms |
|  |  |  |  |  |
| Scopus Target - No Navigator | 95 ms | 70 ms | 180 ms | 3,643 ms |
| ML Observed - No Navigator | 52 ms | 23 ms | 109 ms | 2,932 ms |

Figure : Affiliation Query (Mix)– 1 hour Snapshot



****

**Author Mix Performance Testing**

The performance requirements and load test startup were the same as previously discussed in the earlier Author Performance Testing section. While this time we started one load test box, we needed 3 scripts (instead of 1) to reach the required QPS. While the numbers across the board (avg response time, median response time, 95th percentile response time) were 2x to 4x higher than the isolation run for affiliations, there were still mostly within the requirements. The one area of deviation is the MarkLogic solution exhibited a larger mean and 95th percentile response time (for no navigators) when compared to the Scopus target.

Table : Author Query Types

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Number of queries** | **Percentage of mix** |
| Navigator | 8,439 | 14% |
| No Navigator | 49,879 | 86% |

Table: Author Query Types (overall averages and overall maximum)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |
| ML Observed | 16 qps | 27 qps | 148 ms | 16,327 ms |

Table: Scopus Targets and Observations for Author Query Types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **Mean** | **Median** | **P95th** | **Max** |
|  |  |  |  |  |
| Scopus Target - Navigator | 242 ms | 101 ms | 1,013 ms | 29,091 ms |
| ML Observed - Navigator | 150 ms | 43 ms | 527 ms | 16,327 ms |
|  |  |  |  |  |
| Scopus Target - No Navigator | 131 ms | 46 ms | 174 ms | 28,168 ms |
| ML Observed - No Navigator | 144 ms | 42 ms | 533 ms | 8,366 ms |

Figure : Author Query (Mix)– 1 hour Snapshot





**Core Mix Performance Testing**

The performance requirements and load test startup were the same as previously discussed in the earlier Core Performance Testing section. We decided not to add more boxes(scripts) to increase the load since it appeared MarkLogic was already thrashing and injecting more load would likely deteriorate performance. We were unable to meet the target Scopus requirements for either QPS (avg or max) or response time (mean and 95th percentile). We also noticed that there were approximately 531 requests that failed with a 500 RC from MarkLogic. This is an order of magnitude higher than what we typically see during a SOLR load test run. In addition, the query mix seems to be slightly off (only 5% navigator requests instead of the 10% expected).

Table : Core Query Types

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Number of queries** | **Percentage of mix** |
| Navigator | 8,891 | 5% |
| No Navigator | 160,669 | 95% |

Table: Core Query Types (overall averages and overall maximum)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |
| ML Observed | 50 qps | 67 qps | 547 ms | 52,296 ms |

Table: Scopus Targets and Observations for Core Query Types

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Query Type** | **Mean** | **Median** | **P95th** | **Max** |
|  |  |  |  |  |
| Scopus Target - Navigator | 452 ms | 259 ms | 1,022 ms | 65,058 ms |
| ML Observed - Navigator | 546 ms | 169 ms | 2,211 ms | 22,913 ms |
|  |  |  |  |  |
| Scopus Target - No Navigator | 397 ms | 171 ms | 633 ms | 179,189 ms |
| ML Observed - No Navigator | 545 ms | 164 ms | 2,173 ms | 52,296 ms |

Figure : Core Query (Mix)– 1 hour Snapshot





## Features/Counts Testing

In general the numbers from MarkLogic will be slightly lower due to the changes in the document set since the snapshot for HotHouse was taken. Other minor variations may exist due to the transformation of content not being exactly right, stemming variations, or configuration issues within MarkLogic. For example, we did notice an incorrect namespace was used when constructing some of the ‘phrase-through’ elements. There was also a bug(s) filed with MarkLogic after we identified some errors testing the proximity and scope queries.

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Scopus Count** | **ML 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 | 121530 | af-id(60026851) |
| CORE6 | 2 | 0 | scopus-id(33749522827 OR 33750354968) |
| 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 | 4814880 | all(blood) |
| CORE11 | 3524700 | 3497535 | title-abs-key-auth(blood) |
| CORE12 | 3523591 | 3497524 | title-abs-key(blood) |
| CORE13 | 1383826 | 1383712 | abs(blood) |
| CORE14 | 209717 | 217827 | affil(oxford). Some difference be attributed to the fact that MarkLogic stemmed this field (it should not have been stemmed) but that only results in 76 more results. This looks to be a bug in MarkLogic. |
| CORE15 | 863 | 660 | chemname(rhamnose) AND indexterms (polysaccharide). Element-query version returns the same number of results. Unclear why MarkLogic returns fewer results. Solr returns 846. Does not appear to be a CIP mapping issue. |
| CORE16 | 1021 | 796 | key(rhamnose AND polysaccharide). Unclear why MarkLogic returns fewer results. Solr returns 1002. Does not appear to be a CIP mapping issue. |
| CORE17 | 4145698 | 2721642 | ref(engineering). Solr 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 | 60577 | abs(blood) AND abs(cancer) |
| CORE19 | 664325 | 660871 | (abs(brain) AND NOT abs(cancer)) |
| CORE20 | 1611025 | 1602043 | (abs(brain) OR abs(cancer)) |
| CORE21 | 3308 | 18596 | abs(brain PRE/5 cancer). Appears to be a bug in MarkLogic. Element-query version returns 2952. |
| CORE22 | 5045 | 18596 | abs(brain W/5 cancer). Appears to be a bug in MarkLogic. Element-query version returns 4484. |
| CORE23 | 486 | 751 | title (practice W/5 "letter"). Appears to be a bug in MarkLogic. Element-query version returns 431. |
| CORE24 | 600 | 1304 | title(neuro\* W/3 letter). Appears to be a bug in MarkLogic. Element-query version returns 457. |
| CORE25 | 24767 | 24640 | title(neurological) AND NOT title(neurological W/5 letter). Element-query version returns 24580. Unclear why this did not show the same bug that appeared in other proximity queries. |
| CORE26 | 754 | 1046 | title((neurological OR letter) w/5 practice). Appears to be a bug in MarkLogic. Element-query version returns 692. |
| CORE27 | 3 | 0 | auth(Rumiá Arboix J). Incorrect namespace for ‘ce’ was specified for a phrase-through element. Fixing this would correct the problem. |
| CORE28 | 103987 | 103273 | title(blood) AND pubyear AFT 2003 |
| CORE29 | 23933106 | 23647208 | pubyear AFT 2000 AND pubyear BEF 2014 |
| CORE30 | 3727 | 3697 | abs("heart attack") |
| CORE31 | 21 | 3697 | abs({heart-attack}). It’s not possible for MarkLogic to perform a punctuation-sensitive search in an unfiltered mode. |
| CORE32 | 43462156 | 46629734 | all("the"). Odd that MarkLogic is 3M higher. Could be that one or more elements were not excluded from all and because ‘the’ is a common term. Other searches against the ‘all’ field such as CORE10 seem to match up. |
| CORE33 | 16330 | 16318 | title ("the end" ) |
| CORE34 | 96730 | 235628 | abs(thio\*e). Initially thought this might be the wildcard expansion cutoff in FAST, but Solr returns similar results to Scopus. Unclear why MarkLogic is returning so many results. |
| CORE35 | 5325 | 5290 | abs(thio?e) |
| CORE36 | 2924 | 3628 | title("\*snake venom"). Bug in MarkLogic. Similar to other proximity queries that had problems. |
| CORE37 | 2924 | 3628 | title (\*snake pre/0 venom). Changed to element-query and returned 2908 results. Bug in MarkLogic. |
| CORE38 | 65626 | 4915698 | author-name(smith,j). MarkLogic does not appear to be scoping the query to a specific author. Appears to be bug in MarkLogic. |
| CORE39 | 276 | 1506 | author-name(smith,john). MarkLogic does not appear to be scoping the query to a specific author. Appears to be bug in MarkLogic. |
| CORE40 | 81374 | 79569 | affil(affilorg (university california) AND affilcity(san diego)) |
| CORE41 | 22488 | 16435 | ref(sanger nicklen DNA). Seems odd that MarkLogic is missing 6K answers. Could be a bug in MarkLogic. |
| CORE42 | 6178 | 6052 | ref(refauth(darwin) AND refsrctitle(species) AND refpubyear IS 1859) |

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Scopus Count** | **ML Count** | **Query/Notes** |
| AUTH1 | 1 | 1 | au-id(10038760900) |
| AUTH2 | 11686 | 58335 | auth-last-name(smith) AND auth-first(j\*). SOLR gets 11479 so this sounds like an issue (bug/configuration) in MarkLogic (and not a publication count issue). If the wildcard is removed from auth-first, then MarkLogic returns 11486. Mike believes he has an optimization to address this, but it has not been tested. |
| AUTH3 | 7021 | 52235 | auth-last-name (EQUALS(smith)) AND auth-first (STARTS-WITH(j)). Had to modify original MarkLogic query as it was constructed incorrectly. Modified to use element-value-query. Appears to be a MarkLogic issue (bug/configuration). Mike believes he has an optimization to address this, but it has not been tested. |

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Scopus Count** | **ML Count** | **Query/Notes** |
| AFFIL1 | 1 | 1 | af-id(60030162) |
| AFFIL2 | 348 | 10154 | affilcity(london). Likely attributed to differences in publication counts as well as identification of root nodes. MarkLogic and SOLR had same count (25,847) when all filtering was removed. |
| AFFIL3 | 1 | 0 | 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) with the difference being stemming of ‘universities’. MarkLogic should not have applied stemming to ‘affil’. |
| AFFIL4 | 86 | 332 | 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) with the difference being stemming of ‘universities’. MarkLogic should not have applied stemming to ‘affil’. |

The features/counts testing identified a couple of issues/bugs and areas for further investigation. The query set also highlights the importance of closely aligning the MarkLogic configuration (and index settings) to match the Scopus CIP.

The following items would need to be addressed or the limitations accepted:

**Proximity Queries**

Several queries identified a bug in MarkLogic with handling word positions. It’s a bit surprising that with MarkLogic being a 10 year old search engine, that there are issues with proximity based search. This bug would need to be fixed.

**Punctuation sensitivity**

This can’t be done using a unfiltered strategy. This can only be accomplished with a filtered strategy yielding much slower response times (potentially).

**Scoped Queries**

Several queries identified a bug in MarkLogic with handling an element-query with a nested element-word-query. This bug would need to be fixed.

**Starts-with and Equals**

Could be possible with an element value queries (or optimization suggested by Mike). Further testing would be required.

**Wildcards and Stemming**

Like most search engines, MarkLogic will not process stems when a term contains wildcards. There is a workaround (using cts:match to expand the wildcard so it can be stemmed) but this would introduce potentially significant performance penalities.

## Operational Challenges

### Operating in the cloud

Deploying a cluster in AWS EC2 comes with its own limitations and restrictions. For example in a private datacenter we might specify each cluster instance with more cores or more RAM than the hi1.4xlarge offers. We might also be able to specify more cost-effective storage than the SSD used, without sacrificing IOPS. There is also an inherent performance cost to running under any virtualization technology, including the Xen hypervisor.

### Database and System Configuration

There is a fair amount of effort required to configure the database correctly to achieve all the correct functions on all the fields without affecting each other. We experienced this with configuring the default ‘all’ and the database field ‘allmed’.

Tied into this will be managing the overall system (OS + MarkLogic) and careful management of memory allocation. This can have an impact on performance but also if not correctly setup can lead to errors with caches becoming full during query evaluations. This may be one of the more challenging ongoing issues with respect to setup and monitoring.

### Functional Requirements

#### Result Facets

Facets with exhaustive exact counts will increase response times, particularly with queries hitting even more than 10k documents and worsening as the hit size increase. One option maybe to have two modes, with an exhaustive (exact) count when the number of hits is below a given threshold, and a sampled (estimated) count above the threshold. This of course would need further testing but could yield a good compromise between usefulness and impact on the system.

#### Returnable Fields

The data to be returned can have an impact on responsiveness. The main impact is whether or not results are returned at all, and whether or not results are highlighted. For highlighting we used the internal MarkLogic function cts:highlight, via the high-level search API.

#### Stopword Removal

The current PoC had stopwords removed in the stored queries. We processed these with additional XQuery code at ingestion time. This again is something which ought to exist outside of the search solution and tailored to each product’s needs. MarkLogic does not provide built in support for this.

#### Spelling Suggestions

As previously noted spelling suggestions can be managed within MarkLogic but that is provided a dictionary has been created and loaded. This is currently not an automated process but work can be done to manage this, utilizing the database to extract term lists. Of course this will need to be updated regularly to keep in line with changing terminology.

#### Synonym Expansion

As mentioned previously this really is something that can be done to the queries prior to sending to the search engine and tailored to each product.

## Deployment Architecture

### Based on current solution

The final solution based on this evaluation can’t really be based upon the work already done as part of this PoC. The final solution assumes both a search setup and metadata request setup along with the document repository being leveraged in unison. However, the current configuration was only tested with search queries and was not tested with metadata queries or retrieval queries. Furthermore, the configuration was also not tested with concurrent updates while load testing with queries. While some tuning/optimization could have been done on the current configuration, it must be understood that the current configuration was unable to meet the qps and response time targets. All of this must be taken into consideration when developing a final solution as this could significantly impact cost. We highlight the various approaches below not to get an exact cost, but to offer a few different approaches for a deployment architecture.

The main element missing from the deployment today is the redundancy aspect. In the current form this can be achieved by three methods (some can be used together):

* Cluster duplication - replicating the four-instance cluster as many times as appropriate within AWS region/zone
* Forest replication – redundancy within cluster
* Database replication with MarkLogic server support
  + with or without XA support

Points to consider are:

* affinity between applications and MarkLogic clusters to avoid inconsistent results whilst updates are in progress
* complexity of managing updates
* managing forests merges from time to time (typically a nightly task during the periods of lowest demand)
* database/forest backup management

#### Simple Cluster Setup

The existing PoC has shown it is unable to handle the required level of performance (for search query only). How much additional capacity would be needed is not clear from a performance perspective although we did aim to utilize the existing hardware to the max. This would suggest additional instances in the cluster would be required to meet the performance targets. Currently 32 forests were used, distributed 8 per instance and 4 per SSD. Each forest is approximately 145GB in size, or 580GB per SSD.

Keeping things simple then we currently have:

* 4 instances (hi1.xlarge with 2X1TB SSD)
* 8 forests per instance
* 4 per disk @ 145GB each
* 400GB free space per disk to accommodate merging and updates

In this setup replication would occur by cloning the cluster a number of times, the advantage being you can have each cluster in different zones and/or regions to provide data center failover. This however means any failure of a single instance in the cluster leaves the entire cluster unusable till such time as the instance is recovered.

Updating content across all clusters in parallel could incur some additional complexity, but again for simplicity we can opt to update each cluster independently with an as of yet undetermined latency between full synchronicity across all clusters. Alternatively updates could be applied to one cluster whilst offline. An alternative to applying the updates multiple times is to complete one and then clone the data over to the others. The impact on overall performance would need to be assessed based on the final architecture to determine the method with least impact on the overall live system. Also to aid in recovery online backups can be taken periodically, although with multiple clusters, a clone from one to the other is also possible as a means of recovery.

MarkLogic does provide database replication, which would be the same setup with machines and cluster but managed by MarkLogic itself. Updates are sent to the master cluster and replicated out, with some control over a lag limit to ensure replicas do not fall more than N seconds behind the master.

With this setup you can spread your front end applications across all clusters, and any necessary load balancer to provide automatic failover. There may be slight differences in results between clusters during update operations.

#### Forest Replication Cluster Setup

In the previous setup there was no redundancy within a cluster. This can be achieved with forest level failover (replication) to provide some redundancy within a cluster. This however comes at the cost of maintaining a copy of every forest across the cluster (assuming local disk only and no shared filesystem). This works like RAID-1 with each forest mirroring its contents including any updates to a backup on another host. To avoid complexity of managing forest merges to prevent running out of disk space a cluster with the following setup would work:

* 8 instances (hi1.xlarge with 2x1TB SSD)
* 64 forests
* 4 primary + 4 replica forests per instance
* 4 forests per disk @ 145GB
* 400GB free space per disk to accommodate merging and updates

Note that we have increased the number of forests. This setup should provide additional capacity beyond the current demand, which in a failover scenario should hopefully provide comparable performance. It may be that the doubling of the instances might help meet the qps and performance targets. If not, more instances would likely be needed.

Without further investigation it is not possible to determine whether alternative sized EC2 instances could cope with the forest failover setup adequately.

Note this setup is compatible with the previous database replication option to provide cross region/zone redundancy as well as redundancy within each cluster.

#### XA Support

Yet another approach would use XA and a transaction manager to distribute updates across multiple, and provides atomic updates across independent MarkLogic clusters. This is more complex to set up than database replication, and would require some investment in software development. But this approach has the advantage of eliminating lag between the various copies of the data, and is used by at least one MarkLogic customer.

## Comparison with ScienceDirect HotHouse

We were a bit surprised that the MarkLogic implementation of the Scopus HotHouse did not scale nearly as well (on similar hardware) when compared to the MarkLogic implementation of the ScienceDirect HotHouse. Some of the key differences between the two approaches are listed below.

* There was an order of magnitude difference in the amount of content loaded for Scopus (108M) vs ScienceDirect (12M). There will be some overhead with each document. If the additional documents were causing problems, this would have been further aggravated by the use of the property-axis (which we abandoned due to the inherent underlying join). With a property-axis, each property becomes an additional document so that 108M would have really been 216M.
* As mentioned previously, the Scopus implementation served the dual purpose of a content repository and a search engine. With the ScienceDirect implementation specifically optimized for only search, one would expect it to perform/scale much better.
* The size of the Scopus Database (approximately 5TB) is a bit larger than the ScienceDirect Database (3TB). This could impact the ability for caching (since both architectures specified similar hardware configurations).
* More time was spent on optimizing queries for the ScienceDirect implementation. We had the luxury of an additional internal resource that we paired up with the external consultant. For the Scopus implementation, this was did not happen nor did we think it would have been necessary as we thought many lessons learned during the ScienceDirect HotHouse could have been applied to the Scopus HotHouse.
* There were many more range indexes defined for the Scopus implementation. Since these require memory, it’s questionable whether the current configuration of 4 nodes (60GB ram each) was really sufficient.

# Appendix A - Code & Configuration

**Basic Setup**

The code written is primarily in XQuery, with some small external elements in Java. The CIP was converted to a JSON definition of fields with the subfields and/or XPath expressions along with various attributes (some hand edited from the requirements document) including:

* type
* relative weighting

A number of XQuery function were then written to turn this JSON definition into either parts of the database configuration, or further XQuery statements to be used directly in the end point code base.

The same JSON was used to automatically construct the bulk of the database configuration. Phrase-throughs and element-word-query-throughs were carried over from the previous phase of Science Direct evaluation.

**Deployment**

For deployment “ansible” has been used for remote calls into EC2 instances, and a number of packages/scripts have been written for various tasks.

**Content Loading**

XQuery modules exist for loading the primary content, building citation counts, building query sets, and executing queries. The ingestion-oriented modules all utilize the MarkLogic Task Server to create a queue of jobs to be completed. The main content loading tasks had built in retry logic to recover from S3 errors, with error reporting for replaying of failed content.

Calculating the citation counts used a second set of XQuery modules running on the Task Server. These often share library functions with the primary ingestion routines.

**Searching**

XQuery modules were written to handle conversion of the XML queries into MarkLogic search API requests. This code included stopword removal and exact phrase handling. After processing each query was stored in a special queries database. During load testing queries were fetched from this query database by id, then evaluated on the main database.

# Appendix B – Database Configuration Details

For wildcards a number of database options were chosen.

<word-searches>true</word-searches>

This supplements the stemmed-search index. It is used for unstemmed search, and wildcards are unstemmed.

<word-positions>true</word-positions>

<element-word-positions>true</element-word-positions>

<element-value-positions>true</element-value-positions>

<attribute-value-positions>false</attribute-value-positions>

<field-value-positions>false</field-value-positions>

<three-character-word-positions>true</three-character-word-positions>

<trailing-wildcard-word-positions>false</trailing-wildcard-word-positions>

Position data helps resolve some complex wildcards, as well as near-query.

<trailing-wildcard-searches>false</trailing-wildcard-searches>

<trailing-wildcard-word-positions>false</trailing-wildcard-word-positions>

<fast-element-trailing-wildcard-searches>false</fast-element-trailing-wildcard-searches>

<three-character-searches>true</three-character-searches>

<three-character-word-positions>true</three-character-word-positions>

<fast-element-character-searches>true</fast-element-character-searches>

<two-character-searches>false</two-character-searches>

<one-character-searches>false</one-character-searches>

We chose these options to balance wildcard functionality with database size.

<word-lexicon>http://marklogic.com/collation/codepoint</word-lexicon>

This word lexicon is a list of all words in the database. Internally the database resolves some wildcard terms using the three-character index, and others using the word lexicon. We can trace the evaluation of any query to discover which option was used for any given term.

# Appendix C – Further Technical Details

## Linux Kernel Settings

These are the significant changes made on our PV based cluster:

net.core.rmem\_default = 262144

net.core.wmem\_default = 262144

net.core.rmem\_max = 8388608

net.core.wmem\_max = 8388608

net.core.netdev\_max\_backlog = 10000

net.ipv4.tcp\_rmem = 8192 87380 8388608

net.ipv4.tcp\_wmem = 8192 87380 8388608

net.ipv4.udp\_rmem\_min = 16384

net.ipv4.udp\_wmem\_min = 16384

net.ipv4.tcp\_mem = 8388608 12582912 16777216

net.ipv4.udp\_mem = 8388608 12582912 16777216

vm.swappiness = 0

vm.hugetlb\_shm\_group = 2

vm.dirty\_background\_bytes = 1048576

vm.dirty\_bytes = 8388608

vm.vfs\_cache\_pressure = 1

## Forest Management

The usual recommendation in a runtime environment is 2 cores per forest, so with 4 x hi1 we had 64 cores and up 32 forests. Each forest was about 145GB and 3.3M documents, both of which are good sizes. If the forests grew beyond 8M documents or 200GB, we would recommend adding more instances and correspondingly more forests.

## EC2 AMI

**Amazon AMI**

**amzn-ami-pv-2014.03.0.x86\_64-ebs (ami-2f726546)**

|  |  |
| --- | --- |
| **Description:** | Amazon Linux AMI x86\_64 PV EBS |
| **Status:** | available |
| **Platform:** | Amazon Linux |
| **Image size:** | 8 GB |
| **Visibility:** | Public |
| **Owner:** | amazon (137112412989) |