University of Denver

Digital Commons @ DU

University Libraries: Faculty Scholarship University Libraries

5-11-2020

Building a Library Search Infrastructure with Elasticsearch

Kim Pham

Fernando Reyes

Jeff Rynhart

Follow this and additional works at: https://digitalcommons.du.edu/libraries_facpub

Part of the Databases and Information Systems Commons, and the Library and Information Science Commons

This work is licensed under a Creative Commons Attribution 3.0 License.

Building a Library Search Infrastructure with Elasticsearch

Publication Statement
This article was originally published as:

Pham, K., Reyes, F., & Rynhart, J. (2020). Building a library search infrastructure with Elasticsearch. Code{4}lib Journal, 48. https://journal.code4lib.org/articles/15203

Copyright is held by the authors. User is responsible for all copyright compliance.

code{4}lib JOURNAL

Issue 48, 2020-05-11

Building a Library Search Infrastructure with Elasticsearch

This article discusses our implementation of an Elastic cluster to address our search, search administration and indexing needs, how it integrates in our technology infrastructure, and finally takes a close look at the way that we built a reusable, dynamic search engine that powers our digital repository search. We cover the lessons learned with our early implementations and how to address them to lay the groundwork for a scalable, networked search environment that can also be applied to alternative search engines such as Solr.

by Kim Pham, Fernando Reyes, and Jeff Rynhart

Introduction

The Elastic search platform [1], as its name implies, can consist of many different applications. Its distributed nature is used to solve a multitude of use cases, including application search, security analytics, metrics and logging.

Elastic's most popular application is Elasticsearch [2], known for its ability to provide extremely fast full-text search functionality for many of our applications. Its searching and analytics functionality relies on the open-source information retrieval software library Apache Lucene [3]. Elastic also supports other applications such as Kibana, which serves as the interface to the data stored in Elasticsearch. It allows us to add data to the cluster, build visualizations, and test new or existing queries. Logstash is a data processing pipeline that allows us to filter and manipulate data. It provides the ability to output the data to various products such as Elasticsearch, Redis, file systems and databases. Beats is one way to retrieve and ship data to Logstash and ultimately into the Elasticsearch cluster.

Our Elastic Stack

Elasticsearch is a distributed solution that is designed to be scaled horizontally. In other words, we can add Elasticsearch instances, called nodes, across multiple machines to handle our searching and analytics workloads [4]. A collection of nodes in the Elasticsearch ecosystem is called a cluster. The data in an index is divided and stored across a cluster in shards. A shard in each node contains only some of the data that makes up the overall index. Sharding, the process of dividing data among shards, is how Elasticsearch is able to distribute the data over a cluster. Every Elasticsearch index is configured by default to have five shards although the number of shards per node can be changed to meet your specific needs. JSON documents consisting of multiple fields are the basic units of information stored in Elasticsearch. This increases the speed and efficiency of searching, protects data loss and adds redundancy if a server becomes unavailable. [5]

For example, our Elastic cluster consists of 3 node servers (see Figure 1). We find it is best to use three shards for our system because we are working with a relatively small index (less than 1GB, our most complex mapping has about 100 fields). If we configure an index to have 3 shards and 2 replica shards, each node will contain 1 shard, and there will be 2 replica shards of each shard, and each replica shard will be placed on a different node than its shard. The resulting cluster contains 3 nodes, each containing one primary shard and 2 unrelated replica shards.

As mentioned, our cluster contains relatively small indices from multiple applications. We generally try to have a separate index for our production and for our test instance. For our repository, we have two production indices (one public and one for internal library use where you can view unpublished metadata) and multiple test indices for development.

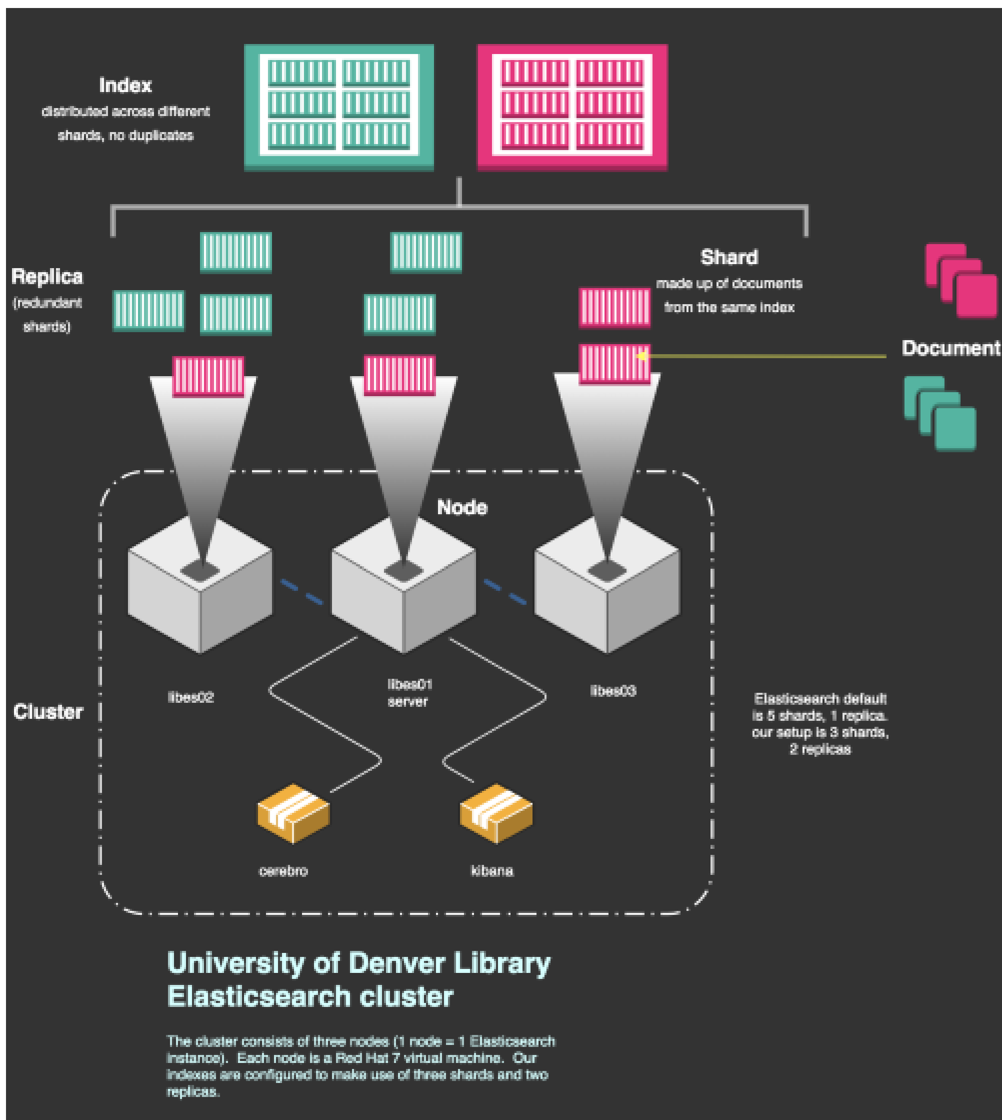

Figure 1. Elasticsearch cluster. Our cluster implementation consists of three nodes (1 node = 1 elasticsearch instance). Each node is a Red Hat 7 virtual machine. Our indexes are configured to make use of three shards and two replicas.

Logging

Centralized logging is another benefit of the Elastic stack. Our Elastic cluster is maintained by our IT department, and all of our applications that leverage Elastic utilize this cluster. This enables our IT department to continuously log the Elastic requests and responses from our applications, and also store logs that are produced by the applications themselves. Aside from this simple logging of requests and responses, our IT department can capture the various logs sent out from our applications on a live, automated system, configured to capture application logs on a set schedule. This system utilizes Filebeats to continuously check the output of application logging modules, such as log4js. We simply include log4js in our applications and the central Elastic stack can be easily configured to connect to each application's log output.

Central Digital Repository Index

Our Elasticsearch cluster serves as our central indexer for our digital collections ecosystem. The digital collections ecosystem contains four main systems (see Figure 2) – ArchivesSpace (archival description), ArchivesDirect (preservation microservices and storage), Kaltura (media management and streaming server), and our custom digital repository. Our digital repository (digitaldu) is comprised of a backend [6] to manage digital objects and a frontend [7] for public access and viewing.

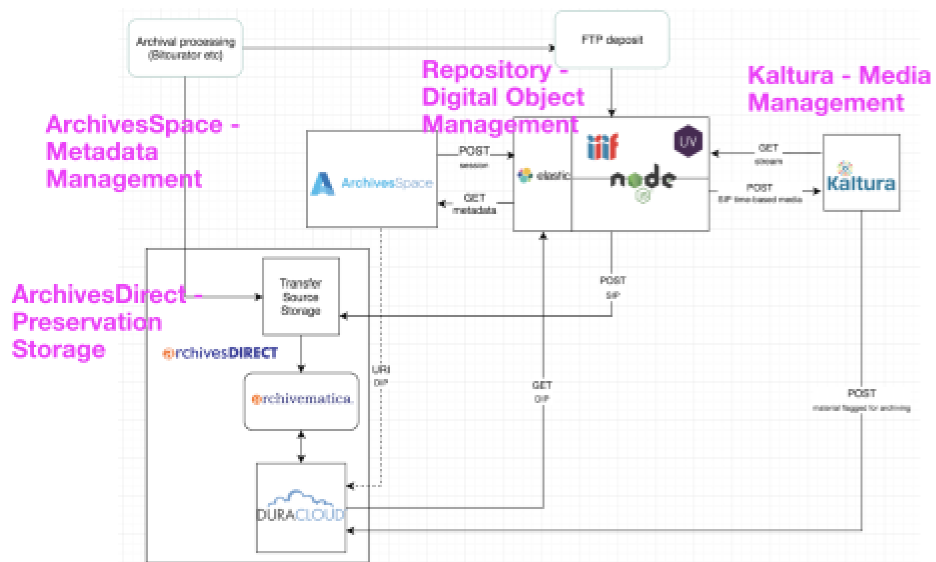

Figure 2. University of Denver Digital Collections Ecosystem, an integrated system architecture used to manage, access, and preserve Special Collections and Archives digital objects.

Data from these systems are mapped into Elasticsearch, where it can be accessed across the entire ecosystem in areas where it’s needed (see Figure 3).

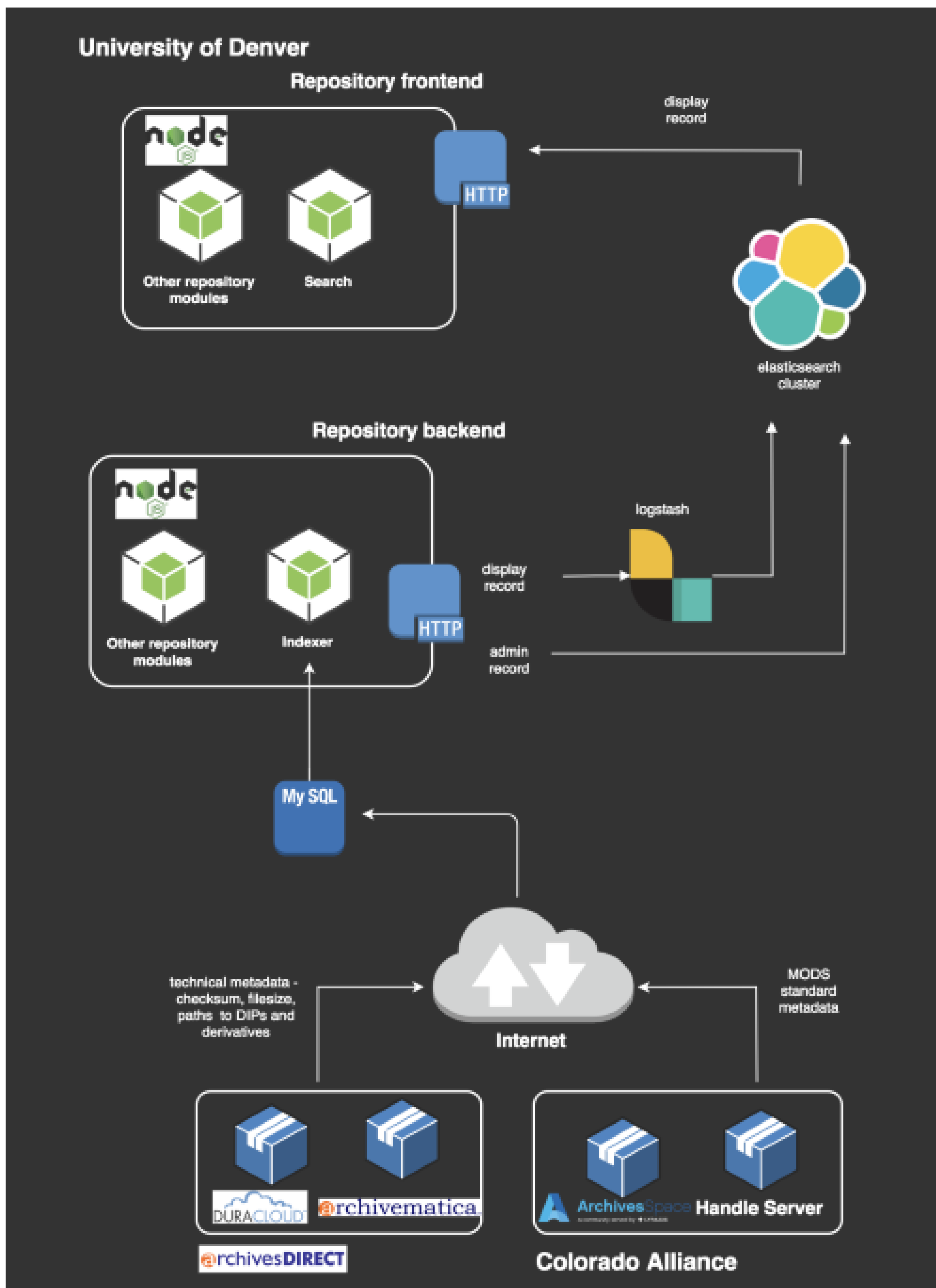

Figure 3. Centralized index for repository search. Data from various sources is sent to Elasticsearch – logstash processes and indexes on fields mapped to Elastic fields.

For instance, when an audio object is ingested into our digital repository through the digitaldu backend, as the object passes through the ingest workflow Elasticsearch is constructing its document record. From digitaldu it indexes viewing information such as its parent collection identifier and access controls, from ArchivesDirect it uses the Archivematica and Duracloud APIs to index the object's filepaths and technical information, from Kaltura it indexes the derivative identifier, and using a custom ArchivesSpace plugin it indexes archival descriptive metadata (see Figure 4). We view the data in Elasticsearch as an index to the persistent stores in each various system – it is not the authoritative data source. It is where data from multiple systems is consolidated that can be re-indexed by sending a request to the APIs and connectors that are associated with each system. This index powers our digital repository, both the digitaldu backend and digital frontend, including generating IIIF manifests for viewing content, and is especially important in developing the search module for both these applications.

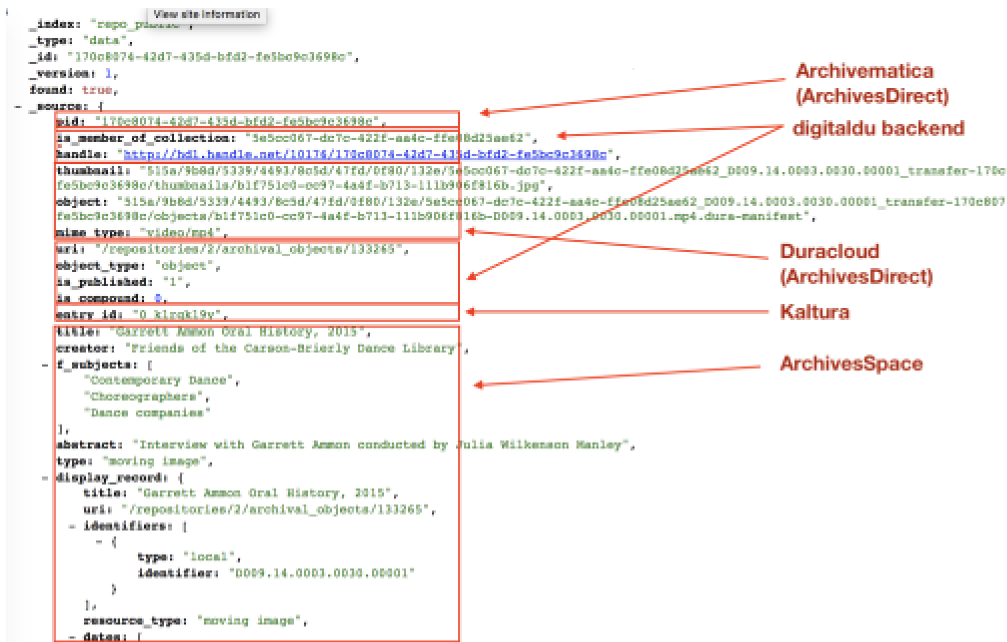

Figure 4. Digital Collections Ecosystem

Application Search

Elasticsearch is used to integrate search into our applications and has improved and simplified our design and implementation of application search functionality. Search is built as a reusable component where the same code can be used across our applications and only requires reconfiguring settings and integrating the search module. Elasticsearch has a powerful range of directives and options that enabled us to develop high quality search applications containing modular and scalable code.

Searches are performed by posting query data to the Elasticsearch REST API in a flexible and powerful format; this is one of the characteristics that makes Elastic search queries so easy to use and customize. This format consists of directives from the Elasticsearch Domain Specific Language (DSL [8]) in a JSON object structure, which is constructed dynamically using the user submitted search data along with application search configuration settings data. The “elasticity” of the structure of the Elasticsearch query object has simplified the development and integration of search functionality into our applications.

Building the Elasticsearch Query Object

When a user performs and submits a search in an application, the data from the search form is used to construct the Elasticsearch query object. Behind the scenes, a search query object can be dynamically created based on full-text queries [9], such as user-selected search parameters, control characters in the search string, and settings stored in the app’s configuration. These data are used to construct a query object for each search, which is posted to the Elasticsearch server to return the matching results (see Figure 5).

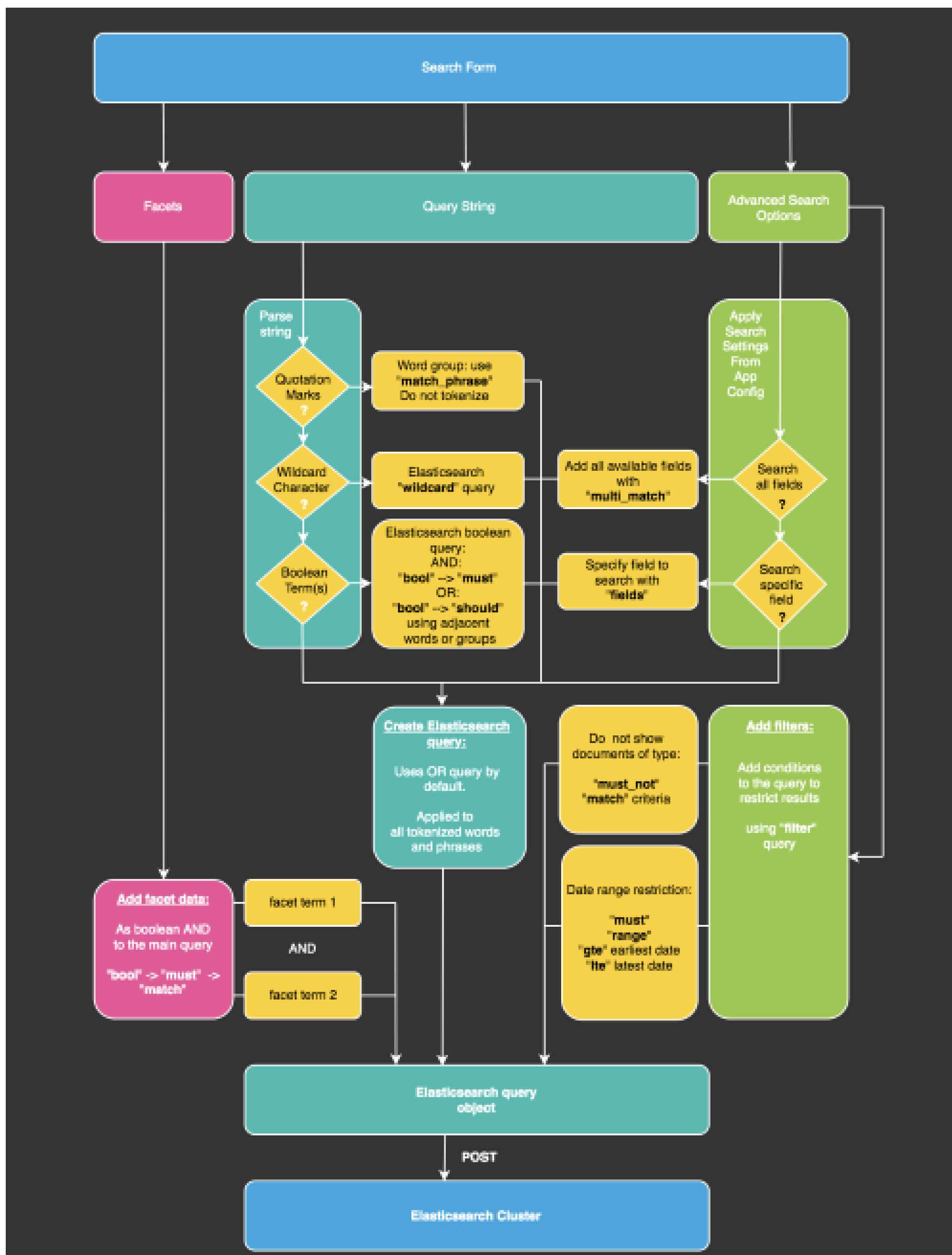

Figure 5. Flow to construct an Elasticsearch query object. The code checks the query string, also if any facets or advanced search parameters exist and their conditions. The query object is sent to the Elasticsearch cluster to get back a JSON response of search results.

Due to its simple structure, the query object can be constructed and posted to Elasticsearch very quickly, permitting a wide range of searches without requiring a specific function or query object for each type of search.

From Search Query to Query Object

A search request includes the field(s) in the index in which are to be searched, a query string containing search terms, search options (such as to target a specific index field or all available fields, or ignore a field), and facet filter data (such as a specific author). This data is inserted into the query object in the required format to define what types of documents will be considered a “match” [10] to the search query. The match query type is a standard, full-text query type that allows for fuzzy or proximity matching.

The query object may contain different “subqueries” that work together. For example, if a user opts to search in “All Fields,” a subquery will be created for each available search field (as set in the app configuration). They will be placed in the DSL “should” array in the query object. This means if one of the queries is a match on a document, the document will be a match for the search so a keyword match in just one field will be sufficient. If the user has selected a filter for the author “John Smith,” a query will be created to match “John Smith,” and will be placed in the DSL “filter” array; so then, one term match must occur in the specified fields, and the filter query must match as well.

Note: The query object can be dynamically created by a programming language, such as Javascript, Python, or Ruby. The following snippets are based off of Javascript code from our digital repository.

```
1  // A DSL subquery as it appears in the Elastic query object.
2  // The “query type” here is “match”
```

```
3  {
4      “match”: {
5          “Title”: “Great Expectations”
6      }
7  }
```

Typically, each query will use the DSL “match” directive. This will return results that match terms in the tokenized query string. This does not have to be an exact match on the string, but document fields that contain more matching terms from the querystring are given a higher score in the results. Facet queries will use the “match_phrase” directive [11], meaning they must be an exact match in the document.

Different directives may be used if the querystring contains certain control characters, to request a specific type of query for the terms.

Control Characters in the Querystring: Quotation Marks

Words grouped by quotation marks must appear in a document in the exact order to result in a search hit. Other combinations of the words will not result in a hit.

```
1  if(query.includes(‘“‘) {
2      queryType = "match_phrase";
3  }
```

For example, a query string may contain quotation marks that group some or all of the search terms together, or a wildcard character. Each query string is parsed, the control characters or terms are detected, and the appropriate directives are added to the query data object.

Control Characters in the Querystring: Wildcard Character

The presence of a wildcard character in the string will create an Elastic “wildcard” query, which will return results that contain any characters in place of the wildcard character. This dynamic creation of the Elastic query based on characters and terms present in the query string allows the utilization of common query based search techniques by the user.

```
1  if( searchTerms.indexOf('*') >= 0 ) {
2      queryType = "wildcard";
3  }
```

Additional Search Parameters: Array of Fields – Search All

Search parameters set within the search form are another way the user can customize a search. A user may specify a single field to search, or may select a group of fields, such as “Search All” (see Figure 6). Available groups of search fields are defined in the app configuration; so if a specific group is specified in the search, the fields are retrieved from the configuration and added to the Elastic query object, specified using the “multi_match” query [12]. Any field that contains the search terms will generate a hit. Individual fields can also be targeted in the search using the “fields” query, such as in “Search Title,” where a field called “title” will be examined but the others ignored. The multi_match or multiple field search is the same as a regular search except that you can search in multiple fields (the same as regular search except that you can combine queries).

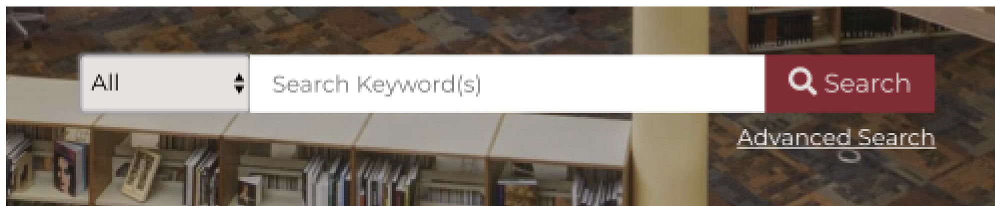

Figure 6. The digitaldu-frontend implementation of search all.

If the user selects “search all fields,” an array of fields is passed in and all of these fields are added to the query as a “multi_match” query:

```
1  for(var field of selectedSearchFields) {
2      let query = {}, temp = {};
3      Query[ field ] = query;
4
5      Temp[ “multi_match” ] = query;
6
7      // Will be added to the booleanQuery
8      matchFields.push( temp );
9  }
```

Additional Search Parameters: Single Field – Search Single Value

If the user wants to search in a particular field, it will not use “multi_match” and it will just search in the specified field. In our system, we provide the user the option to search in specific fields, such as “Title,” “Creator,” or “Description” (see Figure 7):

```
1  let query = {}, tempObj = {};
2  Query[ singleSearchField ] = query;
3
4  temp[ “match” ] = query;
5
```

```
6  // Will be added to the booleanQuery
7  matchFields.push( temp );
```

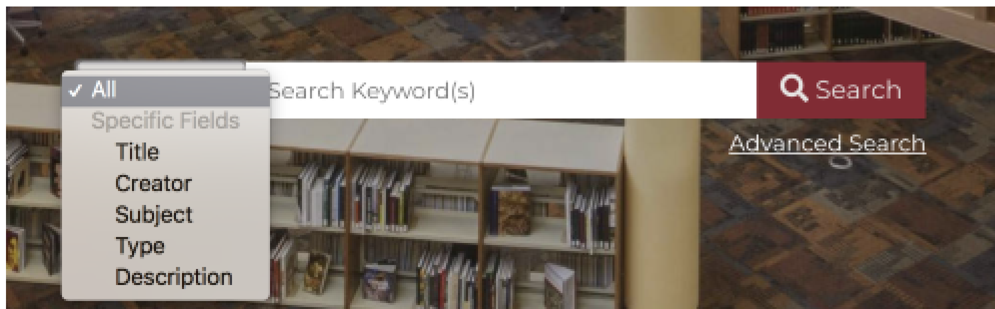

Figure 7. The digitaldu-frontend implementation of search by a single field.

Add Restrictions to the Query Data

Elastic's "must not" [13] query is used to restrict results. For example we don't want our collection type objects to be returned:

```
1  // Add the restricted query
2  restrictions.push({
3      "match": {
4          "object_type": "collection"
5      }
6  });
```

Facets Filter the Results

If facet fields are included in the request, they are combined with the query string search with an Elastic "must" query, so the results must contain the facet data and match the facet query exactly. Facet queries can be executed with or without an accompanying query string. The Elastic query object is built to either apply the facet fields to the main query to limit its results (query string search limited by the facet query), or to search the index using the facet data with no search query string (facet term query not limited by a query string). Facet searches always use the "match_phrase" query in a boolean AND relation to the queries already present in the current search. Only an exact match on the facet string, combined with any existing queries, will result in a hit. This allows faceting to limit the search results.

Add Facets to the Query Data

If facets are selected, loop through the facets by name and retrieve the requested facet value to add to the search. For example, the facet name is "Creator" and the value to limit the results to is "John Smith."

The facet object is built by looking up the facet field in the configuration by -facet name. An example facet name might be "Creator" and an example facet field in the index might be "names.namePart." A facet query is added for each facet field included in the search request.

```
1   searchFacets = {
2       "Creator": "John Smith"
3   }
4   configuration.facets = {
5       "Creator": "name.namePart"
6   }
7
8   // Create facet query
9   if( searchFacets ) {
10      for(var facetName of searchFacets) {
11          facetValue = searchFacets[ facetName ]
12          facetField = configuration.facets[ facetName ];
13
14  // query => {"names.namePart" : "John Smith"}
15          query[ facetField ] = facetValue;
16
17          facetQuery.push({
18                  "match_phrase": query
19              });
20      }
21  }
```

Query Object Is Nested in the Boolean Query

To return search results, documents are brought back from the index that matches a query. A query is typically broken down into sub-queries that contain a set of built-in conditions. The conditions contained within the query object wrapper are called the boolean query object [14] (see Figure 8). This object consists of a set of arrays, each containing a set of individual queries. Each individual array imposes a specific condition on the queries within to determine if they will match a document:

"should" array: If one query in this array matches a document, the array will be interpreted as "true" by the bool query.
"must" array: If all of the queries in this array match a document, the array will be interpreted as "true" by the bool query. Otherwise, it is false.
"must_not" array: If none of the queries in this array match a document, the array will be interpreted as "true" by the bool query.

"filter" array: All queries must match the document. The difference between this array and the "must" array is that the documents that match in that array are given weights to assign priority within the search result list. Filter does not weight the results; it simply blacklists any results that do not match all of the queries within the array. This is useful for limiting results with facet queries.

If all of the above arrays are interpreted as true for a document, the document is a match and is returned as a search result.

```
1  queryObj = {
2     "bool": {
3         "should": booleanQuery,
4         "filter": facetQuery,
5         "must_not": restrictions
6     }
7  }
```

Three subquery arrays are available to be added to the main query object. Each array can contain multiple data objects to control the main query.

1. **Should:** The *booleanQuery* array contains the field/keyword queries. If a search field has been specified, this will contain one field. For a search of all available fields, the array will contain one data object for each field. Any object with at least one matching field will result in a hit.
2. **Filter:** The *facetQuery* array contains the currently selected facet search fields. An object must match all facet queries here to result in a hit.
3. **Must Not:** The *restrictions* array contains data to omit from search results, such as objects that are part of a compound object. Any object that matches any query in this array will not result in a hit.

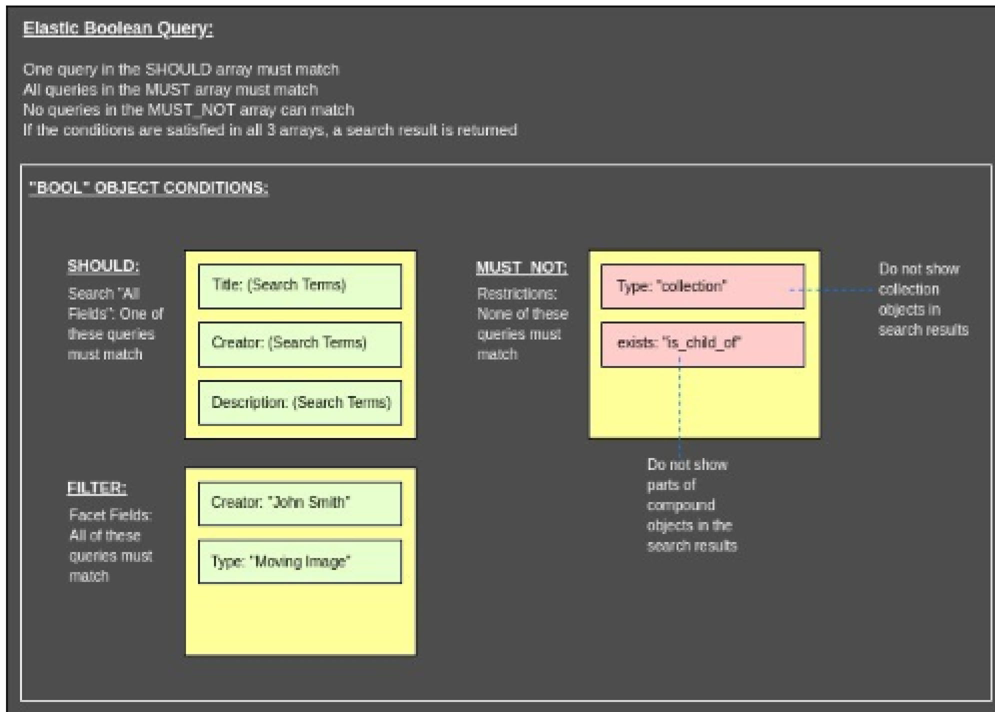

Figure 8. The Elastic Boolean Query serves as the wrapper for the main query object.

Advanced Search

In our advanced search (see Figure 9), the user can add multiple queries combined by boolean logic (AND, OR, NOT) and can specify contains or not contains for each search field (such as Title) or group of fields (see Figure 10). This utilizes Elastic's native boolean query must/must not. In addition to this, the user can target a wider range of search fields to fine tune the search. It is possible to add additional search parameters as well, such as CONTAINS, DOES NOT CONTAIN.

Advanced Search

All Fields | Contains | scrapbook
AND | Subject | Contains | Denver
OR | All Fields | Contains | Boulder
NOT | All Fields | Contains | Seattle

Add another query

Submit

Figure 9. The digitaldu-frontend advanced search interface.

To construct the advanced search query, another boolean query object is assigned to the main query object “should” field, replacing the array of queries that is used in a simple search. The “must_not” and “filter” arrays of the main query object contain the same data as in a simple search.

If there are multiple queries present, these queries are added to the new bool object per the user’s selection of boolean logic for their combination. The bool object has 3 arrays or “buckets” that hold the individual queries and define their boolean relationships: “should” (OR combination), “must” (AND combination), and “must_not” (NOT combination). The first query in an advanced search is always assigned to the “must” array. For subsequent queries, any query inserted as an OR will be assigned to the “should” array, queries inserted as AND will be assigned to the “must” array, and queries inserted as NOT will be added to the “must_not” array.

Therefore, an advanced search query object will contain the fields “must_not” (the restrictions), and “filter” (the facets), and “should”, which contains another nested bool object with the “should,” “must,” and “must_not” buckets containing the queries included in the advanced search.

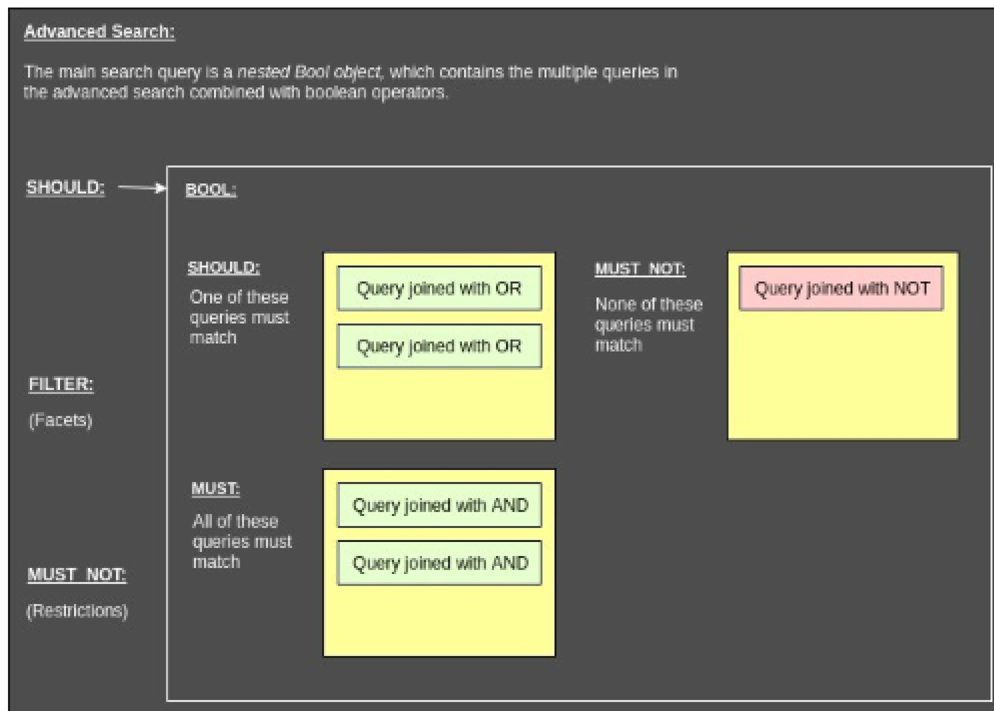

Figure 10. Advanced search, still wrapped inside the Boolean query object but itself is made up of boolean query objects.

To construct the advanced search, multiple queries are created and added to the SHOULD, MUST NOT, and MUST array. Where the user selects ‘AND’ as the operator for search fields, the field gets pushed to the MUST array.

```
1  if(bool == "and") {
2      booleanQuery.bool.must.push(boolObj);
3  }
```

Where the user selects NOT as the operator for search fields, the field gets pushed to the MUST NOT array.

```
1  else if(bool == "not") {
```

```
2        booleanQuery.bool.must_not.push(boolObj);
3    }
```

Where the user selects 'SHOULD' as the operator for search fields, the field gets pushed to the MUST array.

```
1    else {
2        booleanQuery.bool.should.push(boolObj);
3    }
```

Adding the Query Object to the Elastic Data Object

The data object contains index and type settings, the query object, fields in the index to aggregate in the search, search results size and pagination data. This object is converted to JSON and POSTed to the Elastic server.

```
1   /* Build the data object to POST to the Elastic server:
2   Get parameters from the configuration and add the query object
3   Add facet aggregations to the search, such as title, creator, etc. */
4       var data = {
5         index: config.elasticsearchIndex,
6         type: config.searchIndexName,
7         body: {
8           from : (pageNum - 1) * config.maxResultsPerPage,
9           size : config.maxResultsPerPage,
10          query: queryObj,
11          aggregations: facetAggregations
12        }
13      }
```

These are some examples of how the Elastic query object's versatility permits a relatively simple search module to be created to leverage the power of Elastic; as controlled by application search settings, selected search parameters, and control characters within the user's query.

Future Directions

As a tool for logging, as a core component to integrate the various systems in our digital collections ecosystem, and as a powerful search engine, we are able to use Elasticsearch in multiple ways that proves the platform's flexibility. Most recently we developed a repository API using Elasticsearch which is currently in beta, but provides programmatic access to data based on REST queries. Internally we are discussing other potential projects which include improving our central indexer to collect and update data at a more frequent timecycle, application performance monitoring, which includes generating periodic reports and notifications about applications using log data, consolidating our Elasticsearch instances into a single cluster, and improving our application search by incorporating some of Elasticsearch's built-in capabilities like autosuggestion, predictive text, highlighting and fuzzy searching.

References

[1] Elastic Stack Overview, Introduction https://www.elastic.co/guide/en/elastic-stack-overview/current/introduction.html

[2] Elasticsearch Reference, Elasticsearch Introduction https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html

[3] Apache Lucene Core, https://lucene.apache.org/core/index.html

[4] Pranav, Shukla, Sharath Kumar. Learning Elastic Stack 6.0, Seminar presented at the Elasticsearch Engineer I Training, December 2017

[5] Comparing Elasticsearch with Solr – https://thishosting.rocks/comparing-elasticsearch-with-solr/

[6] Digitaldu-backend Github repository. https://github.com/dulibrarytech/digitaldu-backend

[7] Digitaldu-frontend Github repository. https://github.com/dulibrarytech/digitaldu-frontend

[8] Elasticsearch Reference, Query DSL. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html

[9] Elasticsearch Reference, Full text queries. https://www.elastic.co/guide/en/elasticsearch/reference/current/full-text-queries.html

[10] Elasticsearch Reference, Match query. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-match-query.html

[11] Elasticsearch Reference, Match phrase query. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-match-query-phrase.html

[12] Elasticsearch Reference, Multi-match query. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-multi-match-query.html

[13] [14] Elasticsearch Reference, Boolean query. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-bool-query.html

About the Authors

Jeff Rynhart is a software developer at the University of Denver, specializing in full-stack Node.js applications. He is the lead developer and architect of the frontend discovery application for the library's new digital repository. He is also a part-time web developer.

Kim Pham is the Information Technologies Librarian at the University of Denver. She is the coordinator for the Library Technology Services department and oversees the development of the digital repository ecosystem. On Twitter she's @tolloid.

Fernando Reyes is a full-stack software developer at the University of Denver. He is the lead developer and architect of the backend application for the digital repository and the archival system integrations for the digital collections ecosystem.

Subscribe to comments: For this article | For all articles

This work is licensed under a Creative Commons Attribution 3.0 United States License.