Mining Modern Repositories with Elasticsearch

Oleksii Kononenko, Olga Baysal, Reid Holmes, and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, ON, Canada
{okononen, obaysal, rtholmes, migod}@uwaterloo.ca

ABSTRACT

Organizations are generating, processing, and retaining data at a rate that often exceeds their ability to analyze it effectively; at the same time, the insights derived from these large data sets are often key to the success of the organizations, allowing them to better understand how to solve hard problems and thus gain competitive advantage. Because this data is so fast-moving and voluminous, it is increasingly impractical to analyze using traditional offline, read-only relational databases.

Recently, new "big data" technologies and architectures, including Hadoop and NoSQL databases, have evolved to better support the needs of organizations analyzing such data. In particular, Elasticsearch — a distributed full-text search engine — explicitly addresses issues of scalability, big data search, and performance that relational databases were simply never designed to support. In this paper, we reflect upon our own experience with Elasticsearch and highlight its strengths and weaknesses for performing modern mining software repositories research.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Design

Keywords

Elasticsearch, agile data, scalability

1. INTRODUCTION

Although the MSR community is relatively young, its focus and scope of study have been evolving to address the emerging needs of a variety of stakeholders. Terms like "cloud computing" and "big data" have become less exotic as various search providers aggregate the data from all around the Web. Analysts now understand that they can access whatever information they need almost instantly; in turn, this drives them to search for new tools that can ease their everyday routines as the scale of the tasks at hand broaden and become more ambitious.

Most MSR techniques require significant preprocessing of the voluminous raw data: it must be cleansed, filtered, and organized into a usable format for querying. Often, this step is a one-time effort because the goal is to perform an "offline analysis", rather to provide ongoing support for interactive querying of "live" and growing online data. Consequently, researchers often opt to store the processed data in a well known RDBMS, such as MySQL or SQLite, which are simple to set up and easy to query.

While this kind of approach is likely the best choice for some tasks, it is not well suited to the needs of Big Data analysis, which requires supporting much larger volumes of data, accommodating the real-time nature of incoming data, and providing quick responses to queries. In a traditional RDBMS, large data requires the creation of many indices to reduce the execution time of queries; at the same time the presence of indices greatly slows the process of updating the data. Since neither of these choices is practical in the presence of Big Data, it is worthwhile to consider what other approaches might work well in this space.

In this paper we report on our experience with Elasticsearch, an open source search engine that provides near real-time search and full-text search capability, as well as a RESTful API. We have used Elasticsearch to build a tool called DASH, that would not have been possible with previous data management approaches. DASH provides aggregated information about patch review process for Mozilla developers. In the first prototype, we relied on the Bugzilla API to get the data; however, we found its performance to be unusably slow. With the help from Mozilla developers, we then changed DASH's backend to use Elasticsearch. The improvement in response time was remarkable, and it enabled developers to interact with our analysis in real time. This effectively turned our proof-of-concept research tool into a practical industrial-strength analytics tool.

2. ELASTICSEARCH

Elasticsearch is an open source full-text search engine written in Java that is designed to be distributive, scalable, and near real-time capable. The Elasticsearch server is easy to install, and the default configuration supplied with the server is sufficient for a standalone use without tweaking, although most users will eventually want to fine tune some of the parameters. A running instance of the Elasticsearch server is called a *node*, and two or more nodes can form the Elasticsearch cluster. To set up an Elasticsearch cluster, the only value that needs to be set in the configuration file is the

MSR '14, May 31 – June 1, 2014, Hyderabad, India

name of a cluster; Elasticsearch will take care of discovering nodes on the network and binding them into a cluster.

2.1 Background

While Elasticsearch and traditional RDBMSs differ in many ways, at the higher-level many of the core concepts of Elasticsearch have analogues in the RDBMS world (Table 1). All data in Elasticsearch is stored in *indices*. An index in Elasticsearch is like a *database* in a RDBMS: it can store different types of documents, update them, and search for them. Each *document* in Elasticsearch is a JSON object, analogous to a row in a table in a RDBMS. A document consists of zero or more *fields*, where each field is either a primitive type or a more complex structure. A document has a *Document type* associated with it; however, all documents in Elasticsearch are schema-free, which means that two documents of the same type can have different sets of fields. Document type here is similar to the RDBMS notion of a table: it defines the set of fields that can be specified for a particular document.

Table 1: Elasticsearch vs. SQL

Elasticsearch element	SQL element
Index	Database
Mapping	Schema
Document type	Table
Document	Row

Elasticsearch is based on Apache Lucene; each Elasticsearch index consists of one or more Lucene indices, called *shards*. The number of shards that each index has is a fixed value that is defined before the index can be created. When a document is added to an index, the Elasticsearch server defines the shard that will be responsible for storing and indexing that document. By doing this, Elasticsearch balances the loads between available shards and also improves overall performance, since all shards can be used simultaneously. While such automatic sharding is only one key part of the distributed nature of Elasticsearch, the other part of it is automatic distribution of shards among the nodes in a cluster. For example, imagine we have an index that consists of six shards and that our cluster has only one node. In this case, all six shards will be on the same node; however, if we add one more node to the cluster, Elasticsearch will automatically move half of the shards to this new node, and we will then have two nodes with three shards each. Regardless of the number of shards in an index or the number of nodes they occupy, an index is always seen to a client as a single entity.

2.1.1 Communication with the Server

Elasticsearch is a RESTful server, so the main way of communication with it is through its REST API. Communication between the Elasticsearch server and a client is straightforward. In the majority of cases, a client opens a connection and submits a request, which is a JSON object, and receives a response, which is also a JSON object. The simplicity of this mode of communication places no restrictions on programing language used to implement clients or the platforms that they operate on; if a client can send HTTP requests, it can communicate with the Elasticsearch server. Moreover, there are libraries for different languages (e.g., PyES for Python) that take care of some mechanics, and can provide better integration with the language.

2.1.2 Mapping

Mapping is similar to a schema definition in SQL databases. A mapping is a crucial part of every index in Elasticsearch: it defines all document types within the index and how each document and its fields are stored, analyzed, and indexed.

Elasticsearch can work with either implicit or explicit mapping. If the Elasticsearch server has not been provided with a mapping before a document is inserted, the server will try to infer the type of the document — based on the values in the fields of the document — and add this type to the mapping.

While implicit mapping might be an adequate solution in some cases, the use of explicit mapping provides an opportunity to create complex document types and to control how the Elasticsearch server analyzes each field. Explicit mapping allows the disabling of indexing of some fields in a document (by default the Elasticsearch server indexes all fields), which reduces the amount of the disk space needed and increases the speed of adding new documents. This also provides a way to store the data that must not be searched but must be quickly accessible through indexed fields. For example, if we have a set of commits in a version control repository, we might want to index fields like author, date, commit message, etc., but remove the actual changesets from the index. While changesets remain to be instantly accessible through other fields, they neither take additional disk space nor increase the time required to index a document.

2.1.3 Near Real-Time Search

The search in Elasticsearch is near real-time. It means that although documents are indexed immediately after they are successfully added to an index, they will not appear in the search results until the index is refreshed. The Elasticsearch server does not refresh indices after each update, instead it uses a specified fixed time interval (the default value is 1 second) to perform this operation. Since refreshing is costly in terms of disk I/O, it might affect the speed of adding new documents [2]. Therefore, if you need to perform a large number of updates at once, you might want to temporally increase the default indexing interval value (or even disable auto-refresh) and then manually refresh indices after updates are completed.

2.2 Performing a Search

Elasticsearch provides its own query language based on JSON called *Query DSL*. A given search can be performed in Elasticsearch in two ways: in a form of a query or in a form of a filter. The main difference between them is that a query calculates and assigns each returned document with the relevance score, while a filter does not. For this reason, searching via filters is faster than via queries. The official documentation recommends using queries only in two situations: for full text searches or when the relevance of each result in the search is important. For simplicity, we will use term `query` to describe both queries and filters; however, our experience with Elasticsearch is limited to working only with filters, thus we do not report about use of queries.

To execute a search, a client sends a search request to one of the following addresses:

```
http://<server>/_search
http://<server>/<index>/_search
http://<server>/<index>/<documentType>/_search
```

```
{ "query":{"filtered":{
    "query":{"match_all":{}},
    "filter":{"and":[
      {"range":
        {"modified_ts":
          {"gte":0,"lt":1400000000000}}},
      {"term":
        {"reported_by":"johndoe@mozilla.com"}},
      {"terms":{
        "bug_status":["new", "reopened"]}},
      {"not":{"term":{"priority":"p1"}}}
    ]}
  }},
  "from":0,
  "size":100,
  "fields":["bug_id"] }
```

Figure 1: A sample search query using filters.

The first URL represents a search on all indices on the server, while the last one searches only the documents of a particular type withing a particular index.

An example of the search query using filters is shown in Figure 1. An Elasticsearch query is broadly similar to a `SELECT` query in SQL. The `filter` field specifies the conditions that must be met to return a document, similar to the `WHERE` clause in a SQL query. Unlike `SELECT` in SQL, where one must specify the list of tables to be joined before filter conditions are applied, in Elasticsearch the scope of the search is restricted by the URL the query is sent to, and queries are likely to differ only in their filter conditions. Thus, such a query can serve as a boilerplate for a variety of further queries: one need only put the required conditions into the `filter` clause, and a new query is ready to be executed. The fields `from` and `size` are used for pagination (similar to `LIMIT` clause in MySQL) — the former sets the first document to be returned, while the latter sets the maximum number of documents in the returned set. The field `fields` allows the selection of specific fields of interest in the document to be returned. If no fields are listed, the query performs similar to `SELECT * FROM` and results in all fields of the document being returned. As the data is sent from the server over HTTP in a text form, reducing the number of fields will likely improve overall performance.

2.2.1 Writing Filter Conditions

Similar to SQL, Query DSL has Boolean `and`, `or`, and `not` operators, which may be applied to a list of filters. Other common Elasticsearch operators include `term`, `terms`, and `range`. The `term` operator works slightly differently for single value fields (i.e., primitive data types) and multi-value fields (e.g., arrays): for single value fields, it checks if a field value is equal to the value specified in the clause, while for multi-value fields it checks if a field contains the specified value. The `terms` operator works like the `IN` operator in SQL, although it has richer functionality for multi-value fields. Finally, the `range` operator is used to check for inequalities in Elasticsearch; the field value is checked as to whether it falls within the specified interval or not. If both ends of the interval are defined it performs similar to the `BETWEEN` operator in SQL; if only one end is defined it performs like a SQL inequality operator (i.e., `<,<=,>,>=`).

2.3 Strengths

Scalability. Before researchers can do anything with Elasticsearch, they need to decide where they want their data to be stored. A relational database can be used for this; however, all the data is typically stored in a single database. As a result, the more data we store, the more powerful server we need (vertical scaling). The server can be pushed to its limits very quickly, and we would need to start sharding our database and putting each shard on different servers (horizontal scaling). Unfortunately, it is not a trivial operation and to the best of our knowledge none of the current relational database provides this functionality out-of-box. Elasticsearch automatically distributes shards of an index across the nodes of a cluster and controls that they are loaded equally. So if you expect to add more data in future and want to accommodate the growth in data, Elasticsearch is your choice as it scales horizontally.

Agility. Data can be agile in terms of the number of updates or new records, in terms of constantly changing structure of a logical piece of information or document, or a combination of both of them. Relational databases are good at changing/adding data as long as the amount of data in a database is not too large. The time needed to perform a database maintenance (mainly recalculating indices) increases with its size. Elasticsearch can better handle agile data because of a) each of the shards is being indexed/refreshed independently, and b) indices are constantly refreshed with fixed time interval, which means that it is unlikely that a shard has accumulated a lot of unrefreshed data.

In the RDBMS world, a database schema is fixed and known before the first record arrives. If any updates might take place, the schema must be changed and this must be propagated to the records that are already in the database. If the database stores big data, this process can be very slow. Additionally, if the database is used in a domain where documents can have a lot of optional fields, the database can end up having large sparse tables that waste disk space for storing NULLs. Elasticsearch does not impose schema on the documents in indices. If a new document is added to an index and there is a new field in this document, Elasticsearch will automatically update the mapping. There is no need to change already stored documents since they do not have such a field. In addition, Elasticsearch can automatically alter the data type of a field if a value in a new document requires a "wider" type (e.g., changing integer to long).

Performance. Best practises of the relational databases world dictate that each relational database must go through the normalization process during the design phase. By converting a database to a particular normal form we decouple bulk data into several tables and minimize the amount of redundant information. While the normalization benefits the create, update, and remove operations, it is likely to complicate the read operations. Most `SELECT` statements hit more than one table and they must be joined before the filtering conditions are applied. Although every RDBMS handles this operation as efficiently as it can, it is a time-consuming process if the query involves complex schemas. But since Elasticsearch is document-oriented it does not need to spent time on this preliminary step (i.e., gathering the data). Moreover, all shards within an index are searching for the documents satisfying some filter criteria concurrently, and after that results from all them are combined and returned.

While scalability and schema-free documents are common for NoSQL systems, the combination of all three (scalability, agility, and performance) in one system is what makes Elasticsearch stand out from other systems.

2.4 Weaknesses

Security. While offering many great features, Elasticsearch has a major flaw — it lacks security features such as authentication and access control. If an adversary knows the URL of the server, he can easily delete all the indices and shut down the cluster. Because of that there is a need of some proxy and/or firewall that can protect the data. For example, Mozilla is currently working on setting up a public Elasticsearch server that will contain real-time data from their Bugzilla issue tracking. Since there are sensitive (aka security) bugs in Bugzilla that can be exposed to everybody else in the community, they are creating a special proxy that prevents such sensitive data to be visible to the wider community and allowing public users to execute "search only" queries on their Elasticsearch cluster.

Learning curve. The JSON origin of the Elasticsearch query language makes it really easy to start writing simple queries. However, query writing becomes more complicated if it involves nested objects. There is a special type of queries, `nested queries`, in Elasticsearch that must be used when one of the filter conditions is a condition on a field from a nested object. The use of such queries requires the understanding of how particular documents are stored and analyzed by Elasticsearch (i.e., the mapping that is currently in use).

Finally, Elasticsearch inherits some weaknesses of being a NoSQL system — lack of transactions, lack of `JOIN` operation, possible inconsistencies in data, etc.

3. APPLICATIONS

In this section we provide a concrete example of the system that uses Elasticsearch. We also speculate about the boundaries of a domain where Elasticsearch should be chosen over other systems.

3.1 Software Analytics

Elasticsearch is best suited for the applications that are built to handle real time data that needs to be processed and analyzed in a rapid manner. Such applications include software analytics. As an example of software analytics applications, we describe our qualitative developer dashboard tool, called DASH [1].

DASH[1] aims to help developers maintain their awareness of their working environment and individual tasks within issue tracking systems such as Bugzilla. It provides each developer with a set of personalized views of Bugzilla such as bugs assigned to or reported by a developer, bugs that a developer is following up, and outstanding and completed code reviews.

Initially, DASH was implemented using Bugzilla's REST API and relied on the Bugzilla's API servers. We aimed to develop a "live" dashboard displaying dynamic updates on the items and developer tasks, which was challenging. By constantly querying Bugzilla we caused unacceptably large load on Bugzilla's RDBMS back-end. Talking to the Mozilla developers we have learnt about Elasticsearch server which was used by the company's Metrics team to populate chart-like quantitative dashboards. We decided to employ this technology to overcome Bugzilla performance issues.

We created two indices on our own server: one for bugs and one for comments. Each index contained documents of one type only (i.e., a bug and a comment documents respectively). We loaded the data into our Elasticsearch cluster using Bugzilla ETL[2]. DASH itself takes no responsibility for maintaining the data in Elasticsearch in sync with the real data in Bugzilla.

Moving to the Elasticsearch server and being able to execute real time search on the issue repository improved the performance of our tool being very crucial for DASH to be useful to the developers. We noticed a big improvement in the execution time, the average response time of querying (a developer-specific query for one-week data on a full Bugzilla data) and displaying the information on the dashboard was reduced from 30 seconds (using Bugzilla API servers) to 3.5 seconds (on Elasticsearch servers). The response time of a larger query (a one month request) is now 4 seconds compared to the previous 1.5 minutes.

3.2 Other Applications

Elasticsearch is a good choice if you expect to perform any real-time analysis of the data such as as social media analysis, or if you need to provide real-time search that scales to terabytes of information. Thousands organizations worldwide including Netflix, Facebook, GitHub and Stack Overflow, have adopted Elasticsearch to help them overcome limitations of their old approaches in handling new demands of agile data processing and storage. For example, for McGraw-Hill Education Labs, Elasticsearch has provided scalability and high performance to their personalized student learning system. For Xing, Europe's largest professional social network, Elasticsearch brings the ability to handle one million updates per day in real-time and search performance to support 6 million queries per day.

With Elasticsearch we can answer a wide range of questions, from the aggregated information to a fine grained data pieces. We believe that in time current MSR like techniques, in particular, creating traditional offline databases, loading data and use database querying tools, will be replaced by setting up Elasticsearch clusters, writing and performing ETL (execute, transform and load) jobs and performing real-time large data analysis using Elasticsearch search functionality. While Elasticsearch is currently adopted by industrial organizations (mainly due to high costs of hardware and experts), research community is also evolving and exploring solutions such as Elasticsearch or Hadoop to be able to mine modern data repositories.

4. CONCLUSION

Elasticsearch is an easily scalable, full-text search engine that is capable of handling large amounts of online and schema-less data. We built a tool called DASH that allows Mozilla developers see the "live" picture of the patch review process. By switching to Elasticsearch, we remarkably improved DASH's performance which rendered the tool suitable for the real-time use.

5. REFERENCES

[1] O. Baysal, R. Holmes, and M. W. Godfrey. Developer dashboards: The need for qualitative analytics. *IEEE Software*, 30(4):46–52, 2013.

[2] Sematext. Elasticsearch refresh interval vs indexing performance. `http://bit.ly/1iZoPGc`, July 2013.

[1] `http://claw.cs.uwaterloo.ca/~okononen`

[2] `https://github.com/klahnakoski/Bugzilla-ETL`