Optimisation of a Distributed Information Retrieval System

Spring 2016 - Matthieu Rudelle
DATA lab EPFL - Devsearch

Abstract

Devsearch came from the joint effort of 10 students during a *Big Data* course. The project aims to provide a fast access to Github's code database through a language agnostic code search service. One part of the project (the online part) handles the information retrieval system, the fast execution of this service is key for the usability of the search engine. Although, the current implementation yields a response in the order of 10 seconds. This optional semester project intends to reach a response-time below 1 second.

Introduction

Overview of devsearch

One can separate the project in four major parts:

1. Data Scraping
2. Feature Extraction
3. Data Formating
4. Online Lookup

The first part is solved on one side by Devmine project that provides us with huge tarballs of code fetched from github. And the other side by Github Archive that makes available tables in Google's BigQuery that stores event from Github's like stars and contributions.

The second part is performed by Hadoop Spark. On one side by extracting features from the extracted AST (Abstract Syntax Tree) of source code files. And on the other side by running a modified version of Google's pagerank algorithm in order to evaluate the popularity of repositories.

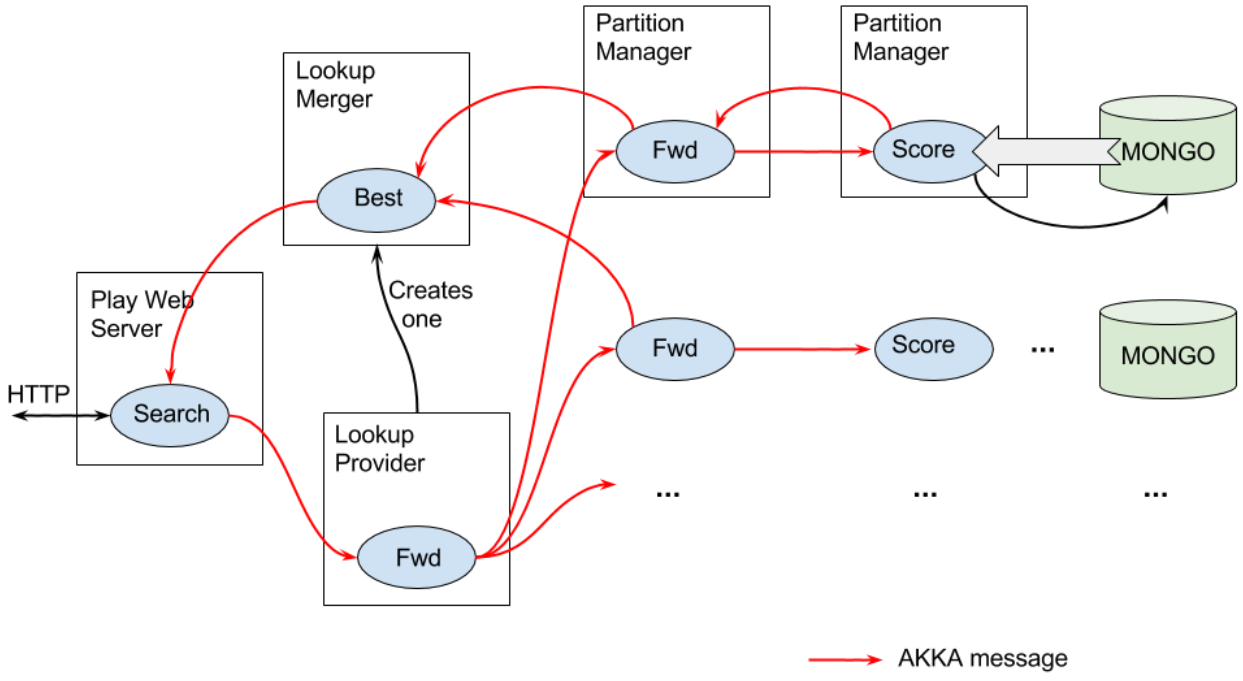
Data formating is also done with Hadoop Spark, data from the previous layer are aggregated to generates tables ready for importation by the online lookup service. Currently the exportation is done in json format and consist of several tables.

The last part is split between the web server (implemented in Play) and the lookup service that uses an inverted index to store and quickly aggregate data matching the query. This paper focuses on improving the rapidity of the latter. In following section we explain in greater length the current implementation of this layer.

The online part

The entire lookup service is articulated around an AKKA messaging system on a local network. A query is initiated by the client in the Play framework, it consists of a set of features contained in the original code and is directed to a ***lookupProvider*** instance that forwards the query to many ***partitionManager*** instances (one per db shard) and creates a ***lookupMerger*** instance in charge of merging the results from each shard. Each ***partitionManager*** forwards the query to a ***partitionLookup*** instance that performs queries on the database and aggregate/formats the data to send back the N-best results on this shard.

Before the optimisation of this paper, the data was stored and served by a MongoDB instance. For each query from the user, we fetched all matching features and grouped them by file. This big amount of data was then retrieved in scala where each file was scored (the scoring system used is explained in greater length in its own section) and and the best results were return to the sender. The architecture is depicted on the following diagram.



This system was not fast enough for queries matching a big part of the data. So we were using the following approximation: Using features statistic we split the set of features in *common* and *rare* features. We were first retrieving the list of files containing one of the rare features then performed a second query matching this list and the entire list of features. This was greatly

reducing the amount of files returned and allowed to stay below Mongo's memory limit for the aggregation pipeline (100Mb).

The data consists of 5 '*buckets*' or shards each handling roughly a fifth of the data, separated per file. One bucket contains roughly:

- 27 million features
- Spread over 260K files
- For about 5.6Gb of json formated data

By sharding the entire data among 5 Azure *D11_V2 Standard* nodes (14Gb of memory, 100Gb ssd, 2 cores, 4*500 IOPS) this implementation was reaching 10 seconds response-time on average for big queries (queries matching with a big portion of the database).

Note: This part is separated in two github repositories, devsearch-play and devsearch-lookup.

Suspected Issues

In order to speed up the lookup process I identified several potential issues that could lower the response time if solved:

- Normalization of the data: the data is stored on Mongo using the entire string as key for files and features. This results roughly in respectively **94** and **40 character-long** keys each. We suspect this to be costly for index lookup and grouping by file.
- Slow Akka cluster: Akka could incur message overhead as it comes with many features that we do not need for this application. It is a cluster of nodes where messages are broadcast widely, we will try to measure the latency incurred by Akka.
- Huge data transfer from Mongo to Scala: Partition managers retrieve a big part of the database for each query, this could be done in place and save transfer time over the network (even if the database is local).
- Slow implementation of Mongo grouping: Several mongo bug reports mention slow grouping operations that do not make use of indexing. We will assess this aspect.

Finally, the current approximation over rare and common features is not desirable as it constraints the scoring system. Although this work do not aim to design or implement a good scoring system we avoid as much as possible to narrow down the available solutions.

Related Work

A lot of documentation is available on search engines, almost all of them concern text search which is conceptually different from what we are doing. In regular text search a user is interested to get a list of documents that are relevant to the query they enter. Here we aim to rather provide snippets and pieces of code that matches what the user enters.

However in technical terms we are performing similar operations (i.e. querying a huge inverted index split among multiple shards for fast lookup). Some reputed Information Retrieval systems stand out (like Google or Yahoo) for their commitment to small response time. These are usually around **a couple hundreds milliseconds** and reached by using purpose built software and hardware, along with important sharding of the data and computing.

Performance review of current system

AKKA

To monitor the latency added by Akka we measure the time in ms taken for one node to receive a message sent by another one. After some manual experimentation we found that a query message takes about 10ms to be transferred and a response message about 35ms (probably due to larger payload). While this is still high for messages on a private local network it is acceptable for our application and other improvements require more attention.

MongoDB

MongoDB is a nosql schemaless db that allows for nested objects and makes the data available in json format. It can be queried in javascript and implements some common database features like pipelining. While this has great advantages it comes with performance drawbacks. We will share some of my findings on this aspect.

Our final query is separated in multiple queries as follow:

- Q1: *Get list of occurrences on local db per language and per feature of the initial query.*
- Q2: *Get list of occurrences on global db per language and per feature of the initial query.*
- Q3: From Q3
    - Sort features per occurrence and extract a list of ‘rare features’.
    - *Get the list of files containing one of these features.*
- Q4: From Q4
    - *Fetch all the features matching the initial query for files in the list of Q4.*
    - *Group results by filename.*
    - Score files (uses Q1 for rarity) and return the N-best

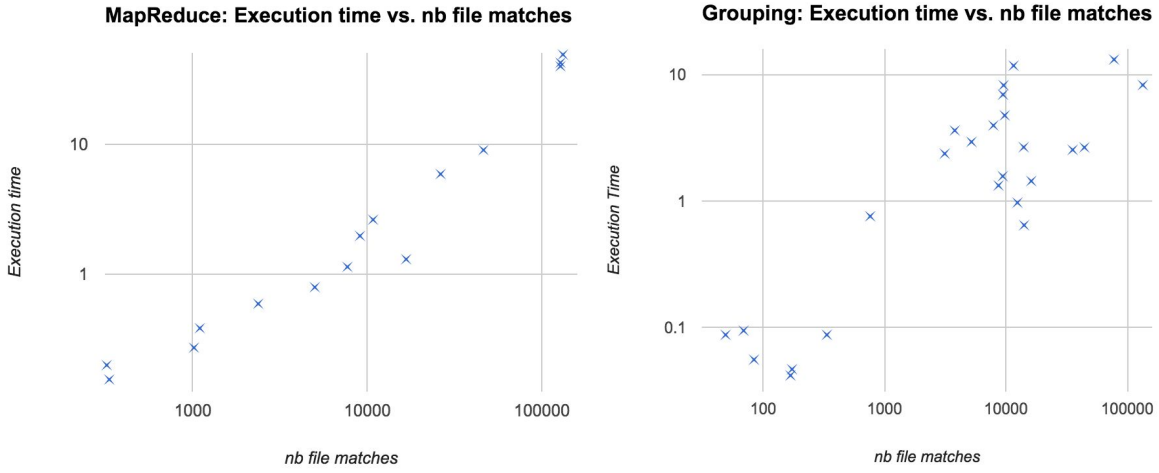
Note: *Italic* statements are performed in Mongo, the rest is done in Scala

Q1 and Q2 have negligible execution time (about 2ms). Q3 and Q4 are highly variable but Q4 is generally one order of magnitude slower than Q4. Hence we focus our measurements on Q4, we observed and reproduced this query manually.

Note: profiling of pipelined query plans in MongoDB is bad, it mainly focuses on counting lock operations and stating the chosen plan but fails to mention the time spent on each part of the pipeline.

First we find that Q3 yields more than 10K file entries (no “distinct” operation was performed on the list) and they do not fit on one query. So we remove this `$match` operator for the rest of the profiling as this can only accentuate the behaviour we try to observe.

The grouping operation of our aggregate pipeline is observed as the bottleneck, without this operator the pipeline is 3 order of magnitude faster (only the index scan is remaining). So we try to bypass the grouping operator by using the mapReduce feature of MongoDB (the implementation is available in annex A). This solutions solves problems of important data transfer, as it moves the computation of the score and sorting to the DB. However this has a very low impact on the query time as we stay around 40 seconds of query time (by taking only two features: “`controlStatement=for`” and “`controlStatement=for`” that jointly account for a 15th of the entire db and half of the files.The following two graphs show the comparison between the two system.



The graph for grouping is much less linear, mainly because these values were computed when the “rare feature” optimisation was still in use.

From those results, around 70K cpu cycles are spent per feature scanned on the DB, this is clearly too much and after having a look inside MongoDB implementation of group operation, they are actually interleaving normal execution in c++ with invocation of JavaScript code where the grouping is actually performed (using a JS map and the object as a key) and results are fetched from the JS scope using c++ again.

As a conclusion, MongoDB is not the right tool for our application (fast lookup and grouping of a lot of data). While we could simply further shard our data, the query time is still about 2 order of

magnitude too high and our experimentation proved that MongoDB is not efficient to process a lot of data.

We then take a look at other RDBMS, many choices seemed good (**VoltDB, CouchDB, PostgreSQL, Cassandra, ...**). We ended up choosing semi-arbitrarily PostgreSQL for its reputation and large community. Results of this experimentation are presented in a coming section.

At the same time we started to experiment building our own custom-made lookup database in C++. The idea being to have an optimize code that best fit our needs through a fine control over memory allocation and usage.

Current Scoring System

The scoring system in place before this paper is more sophisticated than common IR scoring algorithm. We are not scoring documents but parts of documents. From the list of features we obtain, the algorithm extracts clusters of features using a DBSCAN algorithm initialised with an epsilon of 5.

These clusters are then treated as separate documents and from them are extracted the following scores:

- Reporank: that represents the popularity of the repository. The value from our algorithm is fixed between 0 and 1 with a log function. It is applied a coefficient **0.4**
- Density: how dense the cluster is in terms of matching features (topped to 5 features per line) brought to a scale between 0 and 1 and applied to a coefficient of **0.6**
- Ratio of matches: simple ratio of the number of distinct features from the original query that were found on the cluster. The ratio is applied a coefficient **0.1**
- Rarity: some rarity measure on the features that matched this cluster. With a 0 to 1 scale and coefficient **0.3**
- Size: the actual size of the cluster counted in number of matching features. (topped to 20 and brought back to a 0-1 scale with coefficient **0.4**)

The choice of this scoring system is totally arbitrary and needs a fair amount of attention in future improvements of the platform. My experimentations tried to get as close as possible to the original scoring system. Most importantly, the clustering algorithm does not work in a stream fashion and could be replaced in some way.

IMHO: Some tf-idf values other than rarity would be more meaningful.

Note: Reporank's scores should be formated to a 0-1 scale directly in the preprocessing phase.

Experimentation of new lookup systems

PostgreSQL

In this section we present the result of trying to replace MongoDB by PostgreSQL. At the same time we will try to fix problems of normalization, and consequent data transfer by moving the scoring function in the DB.

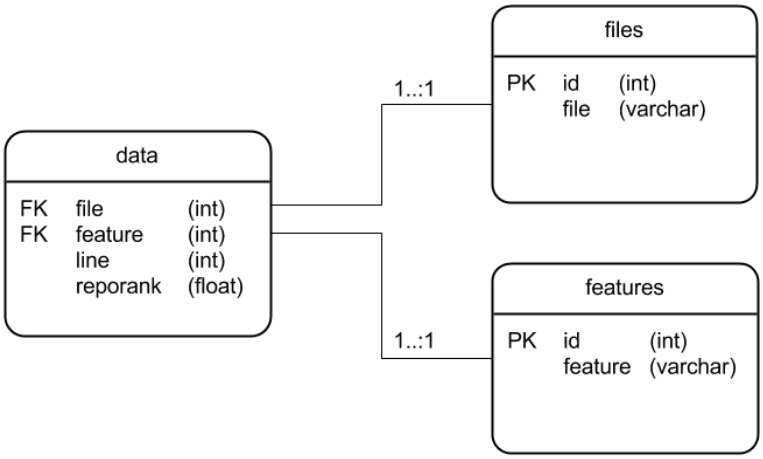
DB structure

Postgresql allows to easily import data from Json files, a first version of the schema was just using one table to store everything. The filePath and the feature as strings, the line as integer and reporank's score as a float (repoRank scores are denormalized at this stage). Then the query was simply a matching operator followed by a group, an ordering and returning the N-best results:

```
EXPLAIN ANALYZE ( SELECT file, size, reporank, (size*0.4)+(reporank*0.4) AS score
FROM (
    SELECT
        file,
        clamp(count(*), 0, 20)/20 as size,
        log(max(reporank)+1)/log(25567) as reporank,
        min(line) AS minLine,
        max(line) AS maxLine
    FROM devsearch_features
    WHERE feature IN ('controlStatement=if', 'controlStatement=for')
    GROUP BY file ) AS result
ORDER BY score DESC
LIMIT 10);
```

Even after indexing this was really slow, about 30 seconds (reference times are given here by querying those two common features as it represents a query with lots of matches). But the computation of the log is very inefficient so we prepared the data by precomputing the logs for each feature entry. This brought down the query time to about 4 seconds.

By analysing the query plan, we observe that a lot of time is spent doing a "heap scan". So in order to bring the query time further down we normalized the data. And obtain the following schema.



This makes the table *data* much faster to read from the heap and we reach about 1.7 seconds for a query which is the best we obtain by using Postgresql in this paper. It does not reach the goal of going under 1 second but improved the previous system with about one order of magnitude.

One should also note this new system is not relying on the rarity of features for optimizing the query time. But this solution limits the scoring capabilities, we are not extracting clusters anymore but taking whole files instead. And we are not computing the ratio of matches. These could be solved by implementing a custom made function in C to be used in the aggregation process. Also the current version does not return the total number of files that matches a particular query and this can be fixed by using views, which I did not have time to experiment.

Integration to lookup framework

Connecting to the database was an easy work. Postgresql provides its own driver to be used with Java Database Connectivity (JDBC) through the url:

```
val db_url = "jdbc:postgresql://" + db_host + ":" + db_port + "/" + db_name
```

The query needs to be written as string with insertion of the needed values and keywords. For this matter I did not have time to make sure that features string were correctly escaped. Also I did not have time to integrate the language selection. This could be easily done by adding another column on *data* for the language of the feature, or by adding a join operation on the list of files but I am afraid this would be costly in time.

Postgresql has a lot of options for memory tuning and we obtained better result by increasing "*shared_buffer*" to have most of the DB in memory, as well as "*work_mem*" to allow for in-memory sorting and pipelining.

Custom C implementation

In this section we try reducing the query time by manually managing memory and optimizing the lookup process. We were able to solve the normalization and huge data transfer problem by moving the computation close to the data and using string dictionaries.

Algorithm

In order to take advantage of data locality we designed an algorithm where the data is a huge contiguous array of struct in memory. This array is sorted by features then files so that lookup will just be a succession of contiguous reads on this array, jumping from one feature to the other.

In order to get rid of variable length strings we use a trie to implement a string dictionary during data importation. A first implementation was using a fixed length pointer array at each node but this was highly memory consuming and switching to a black-red tree map allowed us to keep memory consumption low and lookup very fast.

By using an index on features, we first retrieve the range of memory to scan for each feature of the initial query. Then we greedily “consume” data from each of the pointers one file at a time until all of them are exhausted. To do so we use a priority queue (implemented with a heap) that will always give the pointer with the slowest *fileId* (and that is not exhausted) first so that we are sure to consume all data from one file before switching to another one. As we read data from our big array, we use an aggregation object that accumulate some values for one file in a stream fashion and computes the final score and other information when “*finalized*” is called.

The N files with best scores are kept in a min heap as we traverse the database. This is implemented using c++ priority queue with inversed priority on the score. Then these scores and other statistics are returned to the sender.

This scan system is extremely fast, the only limiting factor is the memory allocation performed during the accumulation process and some optimization could bring it down. When running against an entire shard of the DB, we reach a query time of 1 second using the same two features as before. And it is important to note that this time we are giving a ratio of matches score along with density as well. Plus this system allows to easily add a cluster algorithm during the finalize phase.

Next step

From the previously mentioned performance this lookup system is very promising. It is currently as a state of draft (proof of concept) and should be further improved. The following steps are necessary before having a usable system:

- Have a server daemon running to accept and perform queries.
- Be able to store the state of the db on disk
- Limit memory allocations during aggregation (Eg. via reusing accumulators memory space)
- Implement a set of actions (Eg. import, drop, query) to manage the DB's content.
- Speed up imports, they are very slow at the moment (1h for 26Mio json lines).

The first and second items can greatly take advantage of Google's protobuf library. I looked into it but did not have time to implement a solution.

The repo for this new lookup system is on github: devesearch-db

Conclusion

As a conclusion to all this work, even without reaching a response-time lower than 1s we found the major bottlenecks of Devsearch's lookup system and proposed a solution for each of them.

The final system will require a better insight into the targeted scoring system to use and this is out of the scope of this paper. However, be it with PostgreSQL or a custom made database lookup, the new solution will overcome the current system in place by one order of magnitude more importantly without using approximating optimisations.

Getting a first hand on the system and the data was not easy, mostly due to lacunar documentation. We hope this paper and the improved documentation on the online part will convey more developers to gain interest and contribute in this promising project.

Annex

A. Mongo Map Reduce Code (JavaScript)

```
var map = function() {
    emit(this.file, {line: this.line, feature: this.feature, rank: this.repoRank});
}

var reduce = function(key, values) {

    var ret = {rank:12, list: []};

    values.forEach(function(val) {
        ret.rank = val.rank;

        if (val.list) {
```

```
            ret.list = ret.list.concat(val.list);
        } else {
            ret.list.push({line: val.line, feature: val.feature})
        }
    })

    return ret;
}

var finalize = function(key, reduced) {

    list = reduced.list || [{line: reduced.line, feature: reduced.feature}];
    len = list.length;
    poses = list.map(function(e){return e.line});

    center = Array.avg(poses);
    std = Array.stdDev(poses);


    return {
        file: key,
        score: reduced.rank * len,
        center: center,
        std: std,
        list: list
    }
}
```