# DevSearch

**Amir Shaikhha (Supervisor)**
Christian Zommerfelds (Team Leader), Damien Engels, Nicolas Voirol, Matthieu Rudelle, Bastien Jacot-Guillarmod, Pascal Lau, Julien Graisse, Pierre Walch, Mateusz Golebiewski, Nicolas Hubacher

May 19, 2015

**Abstract**

This report presents DevSearch, a project conducted as part of the Big Data course given by Pr. Christoph Koch at EPFL. We present here the implementations and architecture choices made to achieve a functional product with high performance.

## 1 Introduction and Data Description

### 1.1 Introduction

**DevSearch** is a search engine dedicated to code presented as a user friendly web interface. The user inserts any code snippet and DevSearch will retrieve relevant pieces of code that are alike to provide a context and examples to API and language usage. As of now, **DevSearch** supports *Java*, *Scala*, *Go* and *JavaScript*. The user has the possibility to input his code snippet in any of these languages and filter the results returned at will. Indeed, the outputted results are not necessarily of the same language as the inputted code snippet. This allows a user to convert code from one programming language to another.

### 1.2 Common Representation

To enable language-agnostic comparison and indexing of source code, the first step was to elaborate a common and structured representation for the code.

#### 1.2.1 Abstract Syntax Tree

We used an abstract syntax trees (AST for short) in order to encode all supported languages. This common structure must be flexible enough to support multiple source languages yet precise enough to model inputs correctly without (too much) loss of information during the transformation. This leads to a flexibility-precision tradeoff that we believe remains quite reasonable for our use case. Note however that these trees are untyped as implementing a unified type system would be extremely complex and would not make sense in many cases as we also support untyped languages.

#### 1.2.2 Normal Form

A telling example of AST precision is that we were able to build an SSA normal form on top of it in order to perform semantic analysis in addition to syntactic ones. This normal form consists in a definition tree that follows the program static definition structure and annotates these with SSA control-flow graphs when the definitions are associated to behavior (typically functions). This normal form enables high-level reasoning about abstract code semantics and can serve to identify functional properties of complex structural syntax. However, it would be interesting to further extend this form with a fold-normal form in pure functional style in order to simplify common analysis patterns.

## 1.3 Data crawling

### 1.3.1 Repositories

DevSearch uses data from a previous project named DevMine. DevMine crawled code from GitHub and stored the repositories provided their data to DevSearch so there is no necessary crawling for this part of the project.

### 1.3.2 Data Preprocessing

The data provided by DevMine can not be efficiently processed by Spark. Each of the files would, when loaded into HDFS, occupy a whole data block and therefore waste a lot of space (since even when not used, a block has a reserved space that will not be usable for anything else). In order to avoid this, repository files were concatenated to Blobs of size 640MB. This allows the blocks to be fully used at their maximum capacity and thus, reduce wasted space. Only the last block outputted might have unused space. The Blobs where then transferred from DevMine's server to the Big Data cluster.

### 1.3.3 RepoRank

One of the coefficients used to sort search results is a ”trust” ranking among repositories. We implemented a Google PageRank like algorithm, except that it applies to repositories: RepoRank. For this purpose, RepoRank makes use of two relations; the repositories starred by a user and the users that contributed to a repository. In order to get this information, we use Google BigQuery to process an archive storing the huge amount of events streamed continuously by GitHub API. Storing, formatting and fetching the resulting tables is performed via Google Could Storage. The algorithm performs the following steps until convergence, starting with a uniform distribution of trust:

1. If a repository is popular, then the contributors have to be good coders. Thus the repositories share their trust to the contributors.

2. Good programmers use good tools, hence they will star important projects. The users then gives back the trust to the repositories they starred.

## 2 Implementation

DevSearch can cleanly be divided into two distinct parts: The offline part which consists of a training model while the online part makes use of an efficient search algorithm.

## 2.1 Offline Part

### 2.1.1 Feature Mining

After having loaded the raw data onto the HDFS cluster as described in 1.2.3, we developped a spark job for extracting features from the code contained in the repositories. As a basis for the extraction, we have parsed the source code in order to produce an Abstract Syntax Tree (AST). For each feature we further assigned the exact location where it was extracted from. Prospective features contained in the list are class names, inheritances, class functions, function names, argument names, generic functions, abstract functions, overriding functions, exceptions, imports, map calls, flatMap calls, control statements, type references, typed variables and variables. Then, the features are stored in CSVs files that are located on the hHDFS.

### 2.1.2 Joining and Storing

Through a Spark job, the previously extracted features and the score given by RepoRank to each repository were joined and converted into JSON format. Then, the results are distributed into partitions and stored in different buckets located on the HDFS. Next, each of those buckets was transferred and stored on a remote NoSQL DBMS (MongoDB). The choice of using MongoDB primarily came from the efficiency of a NoSQL based DBMS for the management of high volume of data and its scalability. Two indices were created on the data in order to accelerate the future searches: There is an index for the ID field and a second one for the feature field. See Figure 1 for an illustration of the offline flow.

## 2.2 Online Part

The fundation of the DevSearch website's backend is an Akka network (see Figure 2). Its main components are one single Master node, the LookupMerger nodes and a fixed number of PartitionLookup nodes (slaves). Each of the PartitionLookup nodes is responsible for one MongoDB instance, where a partition of our data is saved.

DevSearch provides a nice and easy to use interface using the Play framework. The user is welcomed by a search box on the home page, where he is prompted for a code snippet. Similar to the offline part, the snippet is parsed for creating an AST and extracting features from it. The extracted features are then sent over to Akka's Master node which first creates and assigns a new LookupMerger node to the query. Then it propagates the query to each PartitionLookup nodes to perform the search on their smaller subset of the database. Each node is in charge of a partition of the files and finds clusters of feature for each file. The PartitionLookup nodes then send those results with their score to the LookupMerger node. The latter gathers the best results to be returned to the user via Play. See Figure 2 for the event flow.

## 3 Architecture

In this section, we present the architecture for the offline part of the project, respectively the online part.

### 3.1 Offline Part

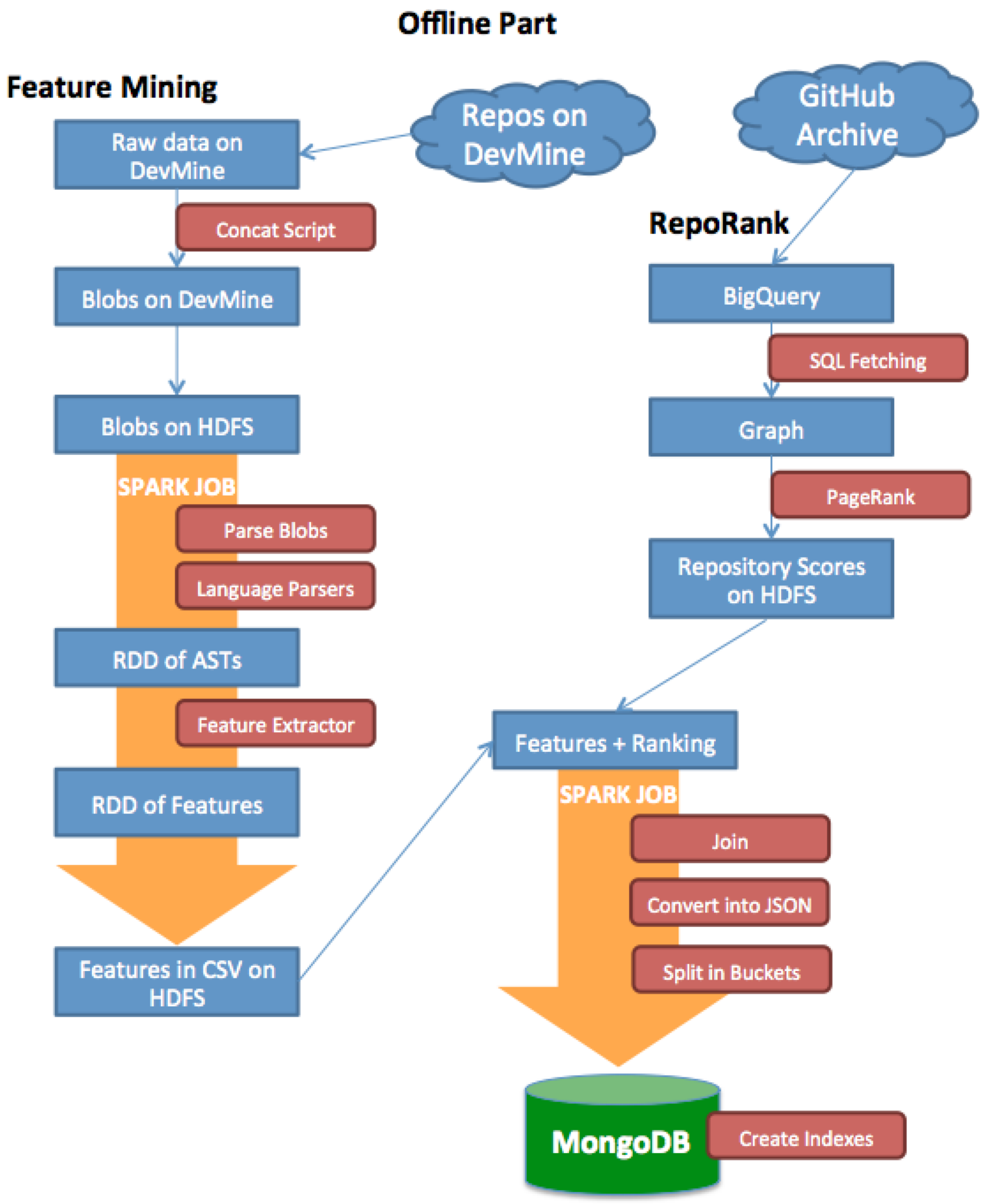

Figure 1: Offline Architecture

## 3.2 Online Part

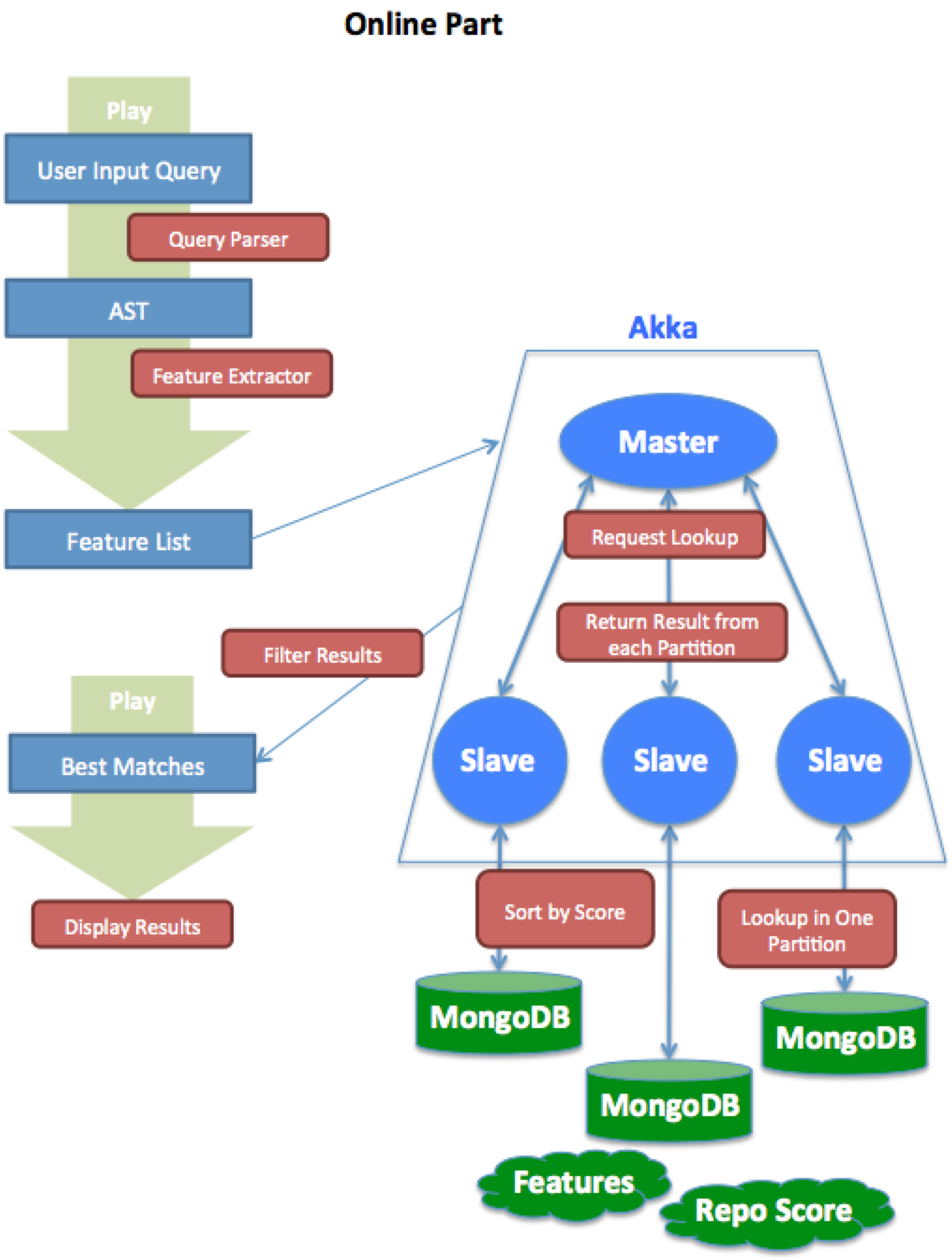

Figure 2: Online Architecture

# 4 Problems and Challenges Encountered

For our project we came to work with an exceptionally huge amount of data. This was a very interesting challenge to tackle as it raises a plethora of issues that you do not expect in the first place. In this section we will talk a little more in details about our journey through the world of Big Data!

## 4.1 Acquisition of the Data

One surprising issue we faced was the time taken to transfer data among various platforms. Indeed, we sometimes had to wait for hours/days for the data to be transferred and available between storage places. The same goes for processing time: Even using large amount of nodes and parallelisation of the computation, some steps had to be planed well in advance to be available for milestones. This is an interesting effect from the large nature of the data: everything takes more time. Here we list the problems and challenges we encountered during the data acquisition phase.

### 4.1.1 Feature Extraction

For feature extraction, the main problem was always to clean and filter the data. Running the Spark job and hoping for the result to come was not enough. At the very beginning, we noticed that some repositories contained more than 200MB of data points (for example a graph defined as vertices/edges). It was very surprising because Git is not intended to handle this type of big files. We decided to drop all files that are too big to be code (more than 2MB) in the pre-processing step (devsearch-concat). Then, we also filtered the files according to their extension to filter out non-code files that fall below the 2MB limit.After that, we still had very big files of at least 600MB.

We had two different formats for the code file data during the semester: one with a header preceding each code file in big text files, and one with tarballs containing the code files. We used the "header format" from the beginning, but we changed to the tarball format two weeks before the presentation. The TAR format enforces some constraints on the input, so we were able to detect some invalid tarballs that could not be processed at all.

These two formats implied the development of custom Hadoop InputFormats. For simplicity and because of uncertainty about the format of the input received from devsearch-concat, we used a non-splittable InputFormat for our two formats. To be able to measure progress and lose less data in case of failure, we splitted the input files into smaller parts of around 1 HDFS block.

After the job was finally running on some valid data, other problems arose. Sometimes, the parser for a language would get stuck in an infinite loop. This involved a lot of communication with the development of devsearch-ast to reproduce and solve the problem outside of Spark.

To avoid running the feature extractor on empty ASTs, we also filtered out the empty ones at this stage.

After all the parsing problems were solved, we encountered the same type of problems for the extraction of features from the ASTs. The normal form extractor would simply go into too many recursions and throw an OutOfMemoryError exception. To solve this, we simply put a limit on the number of recursions that the extractor can do, and ignore the data if the extractor runs over the limit.

Another important challenge of this task was communication with the rest of the group. Since this step was the connection between the offline and the online part, we had to communicate a lot to obtain and provide appropriate interfaces to each other. For example, the front-end developers were not aware of the code file utilities that were provided in devsearch-ast for a long time, which caused code duplication at some point in the project.

The biggest challenge in this part was the time needed to run each job. Before big data, we were not used to run code for more than one or two hours. For example, our slowest parser (JavaScript) took more than 3 days to process a tarball of 5GB. Because of the processing time, we had to be very careful when adding a new feature. We had to make sure the code was always running properly locally before wasting time on the cluster.

Since all files were concatenated together (either with header format or tarballs), there was no way to estimate in advance the processing time needed for a specific input. Indeed, there was a lot of variability in processing time for all the different extractors, and the JavaScript parser would often be a big part of the cost.

#### 4.1.2 Data Preparation

Compared to the splitting of the data into the buckets and converting it into JSON format, the biggest challenge in preparing the data for loading it into MongoDB was joining features and their according RepoRank score. The problem is that Spark RDD's shuffle operations (such as join) build a hash table within each task to perform the grouping. In our case, this hash table can grow extremely large since there is a huge amount of features per repository. The solution to this problem is to increase the level of parallelism to more than the number of cores in the cluster. Taking into account this advice, we ran into another problem: As the level of parallelism grows, the driver requires more memory to manage all the tasks. Therefore we needed to increase the memory allocated for the driver. Because the joining still did not work when executed with the maximum possible level of parallelism (ca. 30000 with maximal driver memory of 4GB) *and* with maximal executor memory (4GB), we finally implemented a custom "hash join": We first distributed the features to their buckets by hashing their file name and *only then* we joined them bucket-wise with the RepoRank scores. By doing so, the repositories could be split in several parts and therefore the resulting hash tables were of smaller size which avoided the outOfMemoryError exceptions.

## 4.2 Deployment issues

Going from small subsets of the data on laptop to the whole datasets running on Microsoft™ cloud racks powered by Student Pass was no easy feat. The code was not always as reliable as we thought, for instance, a lot of variables that should be configurable in `application.conf`. Some defaults configuration of the database were also not suited for our needs as well as the defaults virtual machine provided by Azure™. The usage of SSD disk was thus necessary. But the price of those machines are much higher, thus we expended the (small) wallet of the student pass rather quickly. Luckily, we were able to grab a few of those passes. We thus had to roll from one azure account to another, setting up the node each time. This led to a few sleepless nights and the average consumption of coffee was far beyond human body limitations.

## 4.3 RepoRank

GitHub provides an API to get a lot of information about the daily events taking place on its platform. So we first decided to crawl star and contribution events with an automated script but were quickly stopped by the quotas allocated to access tokens: 5000 API calls/day. Even by using the 10 tokens of our team and optimizing the script to reduce API calls, the job would have taken more than a hundred days to finish. Therefore we switched to the dataset proposed free of charge by Github Archive and available on BigQuery. This service is startlingly fast and allowed us to fetch every Watch and Push events quickly. Due to the huge size of the generated table, we had to store it on Google Cloud Storage as an intermediate step before being fetching it.

## 4.4 Lookup

In the first prototype of the search engine we had the inverted file index on HDFS and were launching a spark job that reads and sorts every matches directly. This was very "hacky" but worked for small datasets. It Obviously was not efficient and very slow on bigger datasets but it provided a first encouraging result.

As the dataset grew bigger we settled for an inverted index on MongoDB. This worked fine in the first place, but we shortly faced a problem of memory. MongoDB has a weird limitation of 100Mb for grouping results of a query. Also it does not allow Join operations so we had to denormalize information such as the RepoRank. We decided to distribute the workload over several nodes using Akka: each node handles a disjoint set of files and the N best matches are gathered by a unique node (LookupMerger). This was still not enough to avoid timeouts and memory limitations so we optimized the way queries are performed by splitting them according to their frequency and performing a smarter filtering.

## 5 Performance and Results

### 5.1 Search Delay

One of the issues we faced is that we had to make a tradeoff between the speed (or delay) at which results are returned and the quality of these results. A user obviously wants to quickly get a result for any input, but he also wants the results to be relevant for his search. Despite using Akka for concurrent lookups on the database, the large amount of data leads to long query time.

### 5.2 Relevance of Results (Scoring Function)

The scoring of a result is computed through 5 metrics: RepoRank, cluster density, cluster size, ratio of feature matches and feature rarity. The latest is defined as the occurrence at which a feature appears within all the repositories. According to figure 2, the feature occurrences is given by a logarithmic curve: There are for example more than 10'000'000 features that were extracted only once (see the top left of the graph). In contrast, only a few features occur extremely often. The control statement 'if' was extracted 21'016'033 times (see bottom right of the graph).
The score is then obtained from the balance of 5 coefficients attached to each metric assessing the quality of the matches. We also developed a visualization and statistical tool for accurate calibration of those parameters.

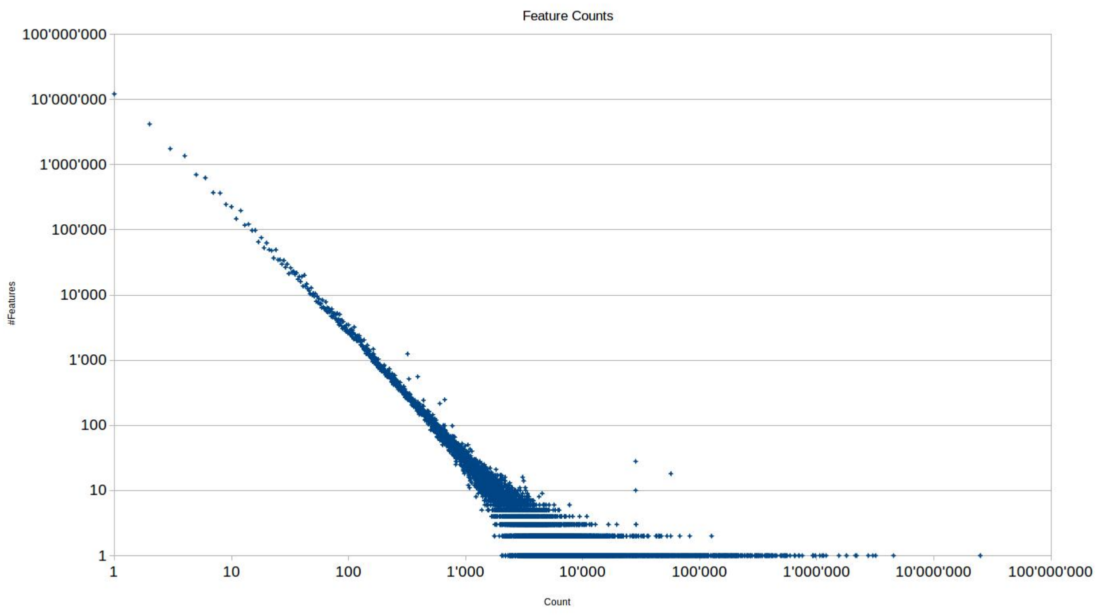

Figure 3: Graph of feature occurrences

## 6 Conclusion

This project was a great learning opportunity. At the beginning, we had very little knowledge on the tools and challenges involved by Big Data and especially search engines. On the path to this final version of the project, we got the chance (and curse) to face issues we were unaware of. Solving them was challenging and helped us improve a lot. The outcome of this project is encouraging and shall be seen as the basis of a huge project. It shall capture the attention of future developers and motivate improvements and extensions of DevSearch.

## 7 Annexes

GitHub repository: https://github.com/devsearch-epfl
Wiki page: http://wiki.epfl.ch/devmine2015