Design and Implementation of Search for the Scratch Online Community

by David J. Wyrobnik


Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

February 2016

© Copyright 2016 Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author:

Department of Electrical Engineering and Computer Science January 28, 2016

Certified by:

Mitchel Resnick LEGO Papert Professor of Learning Research, MIT Media Lab Thesis Supervisor January 28, 2016

Accepted by:

Dr. Christopher Terman, Chairman, Masters of Engineering Thesis Committee
Abstract

The Scratch Online Community has been experiencing rapid growth and has around 9.5 million users and 12.5 million shared projects today. This exponential growth introduces a need for a new Search infrastructure that will enable users to find and discover more relevant projects, while allowing the Scratch team to control the ranking of search results. This thesis presents a complete Search system that will be deployed on the Scratch website and a novel visual tool named FIRST (Fast Iterative Relevance Search Tool) that aids in the process of defining a useful relevance function for Search.

Thesis Supervisor: Mitchel Resnick
Title: LEGO Papert Professor of Learning Research, MIT Media Lab
Acknowledgements

First of all, I would like to thank my supervisor, Mitchel Resnick, for the opportunity to join the Lifelong Kindergarten group. Thank you for your guidance, support, and helping me find a project that met my interests and was important to Scratch.

I would also like to thank Andrew Sliwinski for all your help, guidance, and feedback throughout the project and this thesis. You pointed me to useful resources and helped me learn a great deal throughout this project.

Next, I would like to thank the rest of the Scratch developer team. All of you were always very approachable and responsive when it came to questions about Software Design and questions related to the Scratch system and its components.

To the members of the Lifelong Kindergarten group, thank you, you are a very open and welcoming group.

Special thanks to my family and friends for always being there for me and helping me get to where I am today. To my parents, Shereen and Isy, thanks for supporting me in pursuing my interests from a young age. Even now, while you are an ocean away from me, I can always count on you to support me and what I do and to provide advice when I need it.
Contents

1 Introduction .......................................................... 12
   1.1 Introduction to Scratch ......................................... 12
   1.2 Motivation ......................................................... 13
   1.3 Thesis Overview .................................................. 15

2 The Need for Better Search .................................... 17
   2.1 Current Search and Explore .................................... 17
       2.1.1 Search ......................................................... 18
       2.1.2 Explore ......................................................... 20
   2.2 New Search Infrastructure ..................................... 22

3 System Design and Architecture ............................ 24
   3.1 Goals ............................................................. 24
   3.2 Components and Roles .......................................... 25
   3.3 Elasticsearch .................................................... 26
       3.3.1 Overview ..................................................... 27
       3.3.2 Index Configuration .......................................... 27
       3.3.3 Security Configuration ...................................... 29
   3.4 Crawler ........................................................... 29
       3.4.1 Projects Indexed State, Design Process .................. 30
       3.4.2 Projects Indexed State ....................................... 32
       3.4.3 Populating State ............................................. 33
3.5 Api ......................................................... 34

4 Relevance Function and Scoring 36
4.1 Default Scoring ........................................ 36
4.2 Relevance Function ................................... 38
4.3 FIRST Visual Tool .................................... 41
  4.3.1 Motivation ......................................... 41
  4.3.2 Features ........................................... 42
  4.3.3 Use Case: Generating “Explore” Search Function .......... 46

5 Conclusion 49
5.1 Future Work .......................................... 49
5.2 Conclusion ........................................... 50
List of Figures

1-1 Featured Projects strip on Scratch homepage .............................. 14
2-1 Scratch Search page screenshot ............................................. 18
2-2 Scratch Explore page screenshot .......................................... 21
3-1 Systems Architecture .......................................................... 26
3-2 A mapping of ‘project’ type in the configuration of the index ....... 28
4-1 function_score query structure .............................................. 39
4-2 FIRST screenshot of tools section ......................................... 43
4-3 FIRST screenshot of search results of “star wars” query .......... 44
Chapter 1

Introduction

1.1 Introduction to Scratch

Scratch, designed and developed by the Lifelong Kindergarten Group at the MIT Media Lab, is a visual programming language that allows users of all ages and backgrounds to easily create their own interactive stories, animations, games, greeting cards, and more, simply by snapping together puzzle-shaped blocks [4]. After a user has created a project, they can share their project on the Scratch website (http://scratch.mit.edu), where users can give each other feedback, remix each other’s projects, and form collaborations [5]. Scratch aims to engage young people with designing and creating interactive media. As children create and share programs with the Scratch community, they learn to express themselves creatively, solve problems systematically, and work together collaboratively, while also learning important computational concepts [4].

The creation of Scratch was deeply influenced by Seymour Papert’s constructionism theory and his work with the Logo programming language. Papert advocated the importance of teaching children how to program as a way to teach them about problem solving and computational concepts, such as debugging and abstraction [6]. According to Papert, a program language should be designed with a low floor, such that novices can pick up the language fairly quickly, and a high ceiling, so that advanced users can develop complex programs [4].
Scratch inherits many of these key ideas from Logo, but differs in various ways. First, in addition to the low floor and the high ceiling, Scratch has *wide walls* -- which allows users to create many different projects from interactive games to simulators. Furthermore, Scratch allows users to include their own images and sounds and thereby personalize their projects. Finally, users can share their projects with the online community, where they receive feedback, can view other projects, learn new techniques, find inspiration, and collaborate with each other [4].

1.2 Motivation

Since the release of Scratch in May 2007, the number of users and projects being shared has been growing rapidly. In January 2015 there were approximately 5 million users and 7.5 million shared projects. Today, one year later, there are 9.5 million users and 12.5 million shared projects. This is of course great news. Every day thousands of diverse projects are being shared with the community, including replicas of popular video games, interactive art, music videos, stories, holiday greetings cards, physics simulations, news reports, virtual instruments, petitions, and much more. But this growth also introduces a whole set of new problems. First, the Scratch website needs to be able to scale as the number of users and projects grow. Second, finding new interesting projects amongst the vast number of newly shared projects can be challenging. And third, when a user shares a project, they would like their project to be seen and discovered by other users. Established users that have many followers do not face this problem as much, but newer, less popular users have a harder time being discovered and getting feedback on their work.

The Scratch team has introduced various features to the Scratch website to promote project discovery. The homepage contains a *Featured Projects* strip (Figure 1-1) that contains curated projects selected by the Scratch team. These are usually projects that show an innovative use of
Scratch, are inspiring, or promote collaboration and remixing. Projects can get featured, if the creator has not recently had a featured project and if the project, at feature time, was not already very popular amongst the community. There are many other project strips to promote project discovery aside from the Featured Projects strip. These include the Projects Curated by [curator] strip (a Scratch user is picked as curator and picks five projects), Recently Shared Projects strip, Projects by Scratchers I’m Following (shows projects from Scratch users the logged in user is following), Projects Loved by Scratchers I’m Following, What the Community is Remixing and What the Community is Loving (most remixed and loved projects from the last ten days).

Additionally, a project’s page contains a remix strip that shows projects that were remixed from the currently viewed project.

When a user is trying to discover new projects or projects that are of interest to them, they can use the current Scratch Search and Explore pages. These are very popular and used extensively by the community, although currently these two pages have limited functionality and there is demand from both the community and the Scratch team for a better solution.
1.3 Thesis Overview

This thesis presents a new Search infrastructure for the Scratch website. Particularly, the thesis is organized in the following way. Chapter 2 gives some background on the current Search and Explore pages in Scratch, their downsides, and introduces the new Search backend. Chapter 3 presents the systems design and architecture. Chapter 4 describes how scoring and ranking of search results works and introduces a tool, FIRST, developed as part of this thesis that aids in the process of defining a relevance function. Finally, chapter 5 mentions some suggested future work and the conclusion of this thesis.
Chapter 2

The Need for Better Search

This chapter describes the need for a better search system for Scratch by describing the existing Search and Explore page, their issues, followed by a brief introduction to what the new search system can do, and how it solves many of the mentioned issues.

2.1 Current Search and Explore

Scratch provides two features to help Scratch users discover projects: the Search page and the Explore page. The Search page lets the user search the website and includes tabs for “All”, “Projects”, “Forums”, “Studios” and “Users”. The Explore page is focused on projects and studios and allows the user to discover projects or studios, categorized by tags, that were shared recently (past 7 or 30 days). Search is more general and lets the user discover projects using search queries.
2.1.1 Search

The Search page allows users to search the Scratch website (Figure 2-1). Currently the functionality is powered by Google Custom Search Engine (CSE). A query can search for
projects, forums, studios, users, or all of the above. Search results can be sorted by relevance or date. Google uses the PageRank [9] algorithm, amongst other algorithms, to rank search results. PageRank determines how important a page is by counting the number of quality links to the page. This means that a project on Scratch would show up higher in the search results, if there are many other pages that link to it. This in itself is not a good way to score projects on Scratch. In theory, a user could boost their project’s search score by posting many links on various websites to it. There have been cases of SEO spammers whose spam projects appear as top result. Instead, Scratch internal metadata, such as favorites, loves, and views, should be used to score search results, but CSE does not have access to the internal metadata. When asked, users and Scratch team members are often unsure why particular search results show up at the top.

Search currently does not provide any functionality to filter searches by language or project tags. This makes it hard for many users to find projects that are in their language. Users in general have a hard time finding a specific project and have requested more search features to allow them to sort and filter more appropriately.

Users have complained that their projects take a long time to show up in the search results. This happens because it takes a while until a new project is indexed and available in search. Users also complain that when they create, or a friend creates, a new Scratch account there is a long lag time before the user account shows up in search.

The Scratch team censors some projects, if the project contains inappropriate material, which effectively unshares the project. Some projects are marked as “not for everyone”, which means that they are still shared, but these projects do not show up on the homepage. Projects that have been censored, some with inappropriate language in the title, or marked as “not for everyone”, usually still show up in search results. Links to censored projects are dead. This is not the intended behavior and the Scratch team should be able to remove projects from Search the same way they can remove shared projects when needed.
In general, the Scratch team has little control over the current search site, since the functionality is implemented and deployed at a third party (i.e. Google).

2.1.2 Explore

The Explore page lets a user browse projects and studios with no particular project or studio in mind (Figure 2-2). The goal is to give the user an opportunity to browse and discover content. Explore allows the user to look at projects that have a specific tag, such as animations, art, games, music, and stories. Either projects from the last 7 or 30 days are displayed and the projects can be sorted by newest, most viewed, most loved, or most remixed. The Explore functionality is implemented by the Scratch team and is built internally, as opposed to the current Search page. Explore queries translate to database queries on the backend. Currently there is no search infrastructure used for Explore.

The Explore page does a good job given its role. There is demand for more functionality, though. I have been told by some members of the Scratch team that they use the Explore page to find projects and that they have a hard time finding specific projects they are looking for. It seems to me that the Explore page is not only being used for browsing and discovering, but also for searching, which is not the role of the Explore page. My guess is that a better Search experience on the Search page will also affect the expectations of the Explore page. Nonetheless, Explore does have limited functionality. There is a lack of filtering options (such as filters for project language) and there are only a few simple sorting options. The Explore experience could be greatly enhanced, if projects were sorted or ranked better using a relevance function developed by the Scratch team.
Figure 2-2: Screenshot of Scratch Explore page.
2.2 New Search Infrastructure

The new Search system that is described in this thesis aims to replace the current Search page and potentially the Explore page. The system is built internally using open source software. The Scratch team has control over the system and the machines where the system runs. This new system is a complete overhaul of the existing Search page backend and addresses many of the issues and wishes mentioned in the previous section.

The new system has access to the Scratch internal databases and can index projects with metadata, such as love-its, favorites, and views. The relevance function is specified by the Scratch team and uses project metadata, as opposed to the current Search page that ranks search results by metrics that are mostly unrelated to projects.

Project metadata fields that are used for ranking of search results and fields that are useful for filtering, including tags and language, are indexed. With these fields in the search index, the new search has useful filtering options.

In response to the long lag time before project or users get indexed and appear in the current Search system, the new system runs the index job itself and some initial performance optimization has already been conducted. In future iterations, the time to index all projects will speed up. Additionally, since the team has access to the index, we can manually index or delete projects or users when needed. This means that the team can manually remove censored projects from the index.

The Explore page has not as much to gain from using the Search system on the backend as the Search page, but could still make use of it. The main advantage the Explore page would gain from replacing the backend with the new Search system, as opposed to database queries, is the ability to offer more sophisticated sorting or ranking of projects that get presented to the user.
Chapter 3

System Design and Architecture

This chapter describes the search system design from a high level and then describes each component in greater detail.

3.1 Goals

The search system was designed with the following three goals in mind. First, the infrastructure should mostly be standalone and impact the existing Scratch website infrastructure as little as possible. Second, the system should be easily extendable to other parts of the Scratch Online Community and, in general, the design should be usable for other websites or systems that want to add search capabilities. And third, the system is designed with scalability in mind. This goal is important, especially given the growth the Scratch community is currently experiencing.
3.2 Components and Roles

There are three main components implemented for search: the Api, the Crawler, and the Elasticsearch [1] cluster (Figure 3-1). At the heart of the search implementation is Elasticsearch. It provides all the capabilities needed to index and search documents. In 3.3, I will go into more detail about Elasticsearch and how it is used in the current implementation. The Api is the entry point to search and is the only component that is Search client facing. It formats and sends the search requests to Elasticsearch. The Api is described in 3.5. The Crawler provides Elasticsearch with documents to index and the Crawler maintains the state of these documents. It is called Crawler because it crawls the datatable for objects to index; the design process and implementation of the Crawler is described in 3.4.

Recall the three goals introduced in the previous section. In the following paragraph I describe how the overall system addresses these goals. First, the Elasticsearch cluster is completely contained within the search black box. The Api is the endpoint and the Crawler is the only component that touches the Scratch website’s database. Second, currently search is implemented for projects, but is easily extendable to other types of objects, such as users and forums. And third, Elasticsearch is scalable out of the box (it can be run replicated and sharded). The Api is easily scaled up. The Api is stateless and therefore could be run on distributed instances without complications. The Crawler is harder to scale and in the current implementation it runs on a single machine. Distributing the Crawler is part of future work. Although the crawler does not handle user requests, it should still be scalable for performance reason. The faster a single run of the Crawler is, the fresher the search index is. This means that if the Crawler takes 4 hours to complete a single crawl, then a newly shared project would take at most 4 hours until it has been indexed and appears in Search.
Figure 3-1: Elasticsearch at the heart providing the search capabilities. The Api forwards the formatted search queries and the Crawler tells Elasticsearch documents to index, reindex or delete. Crawler gets and writes state of documents (needs indexing or not).

### 3.3 Elasticsearch

This section describes the capabilities of Elasticsearch and how it is configured in the search infrastructure. This section may touch on how Elasticsearch works, but it is not the goal of this section, because for the most part in the implementation of the search system described by this document, Elasticsearch can be viewed as a black box that has certain endpoints and capabilities.
3.3.1 Overview

According to the Elasticsearch website [1]: “Elasticsearch is a highly scalable open-source full-text search and analytics engine. It allows you to store, search, and analyze big volumes of data quickly and in near real time. It is generally used as the underlying engine/technology that powers applications that have complex search features and requirements.”

In an Elasticsearch cluster you can have a single node or multiple nodes. A node is a single server that is part of the cluster. Elasticsearch handles replication and sharding based on the configuration specified.

To add/index a document to Elasticsearch, one needs to be familiar with three basic concepts. These are Index, Type, and Document. An index is a collection of documents that have similar characteristics. When conducting a search query, the query is executed against an index. Within an index, one or more types can be defined. A type is a partition of the index whose semantics can be arbitrarily defined. Documents of a certain type have similar fields. For example a type could be ‘project’ and documents of type ‘project’ could have fields ‘title’, ‘description’, and ‘view count’. A document is a basic unit of information that is indexed as a certain type, for example a specific Scratch project.

3.3.2 Index Configuration

During initial development, one only needs to specify the name of the index, but one can let Elasticsearch infer the fields of a type. Later it makes sense, for performance and memory optimization, to define the type manually, because then one can tell Elasticsearch which fields need to be full text searchable (e.g. the title and description of a project), and which fields should only be stored, but not searchable (e.g. the image url of a project). To define the type manually one needs to configure the index before adding the first document to the index. Figure 3-2 shows
the mapping of the initial index where a single type ‘project’ is defined and implemented. Each field can specify a value for the key “index”. There are three values that can be set: “analyzed” (default), “not_analyzed”, and “no”. For fields of type string, Elasticsearch breaks the contents into terms such that a search of any of the terms in the string would return the document. For example, if the title of a project is “brown fox”, then a search for “fox” will still match the document. This is the default behavior. If a field does not need to broken into terms, but only searchable as is, then one should specify “not_analyzed” on the field. For example, the moderation_status of a project can be set to an enum value, and therefore a search on the moderation_status field can and should be for any of the enums. Lastly, if Elasticsearch should only store a field, but will never need to perform search on said field, then one should set “index”
to “no”. This tells Elasticsearch to only store a field in the raw source of the document and not make it searchable. This is more memory efficient, since Elasticsearch does not need to create and store another data structure that allows quick lookup based on the field. For example, the thumbnail url is also stored in Elasticsearch, because when searching for projects we want the search API to also provide the thumbnails of search results, but we never need to search for a thumbnail url.

### 3.3.3 Security Configuration

Elasticsearch does not perform authentication or authorization and in addition, depending on configuration, accepts and executes scripts. Therefore it is important to secure the cluster. Since the cluster only needs to accept requests the API and the Crawler, the Cluster should only accept connections from these two clients. All three components are deployed on Amazon EC2 [10] and EC2 has a feature called “security groups” that lets the developer specify that an instance should only accept connections from a specific security group.

### 3.4 Crawler

The Crawler’s main task is to tell Elasticsearch what documents to index, reindex, or delete -- and thereby the Crawler decides what should be available through search. The Crawler has access to the Scratch website’s database and crawls the relevant tables to get data, formats the data, and sends it to Elasticsearch for indexing. In this section, I will describe the design process and finally how the Crawler was implemented specifically for Scratch projects that are shared in the community.

The first subsection describes the design process for solving the problem of how the Crawler keeps track of the state of projects, whether a project needs to be updated in the search index.
The second subsection describes the design process for solving the problem of how to populate the state initially.

### 3.4.1 Projects Indexed State, Design Process

When the Crawler goes through projects in the database, it needs to recognize if a specific project needs to be sent to Elasticsearch for indexing or reindexing, or if a project needs to be deleted from the index, when a project is unshared. Therefore the Crawler needs to store some state to decide what projects need indexing. As a side note, this is the only stateful part of the system. This subsection will go through different designs and finishes with the final design that has been implemented.

The first goal listed at the beginning of this chapter is that this system should impact the existing website's infrastructure as little as possible. The projects table has a column named `datetime_modified` that stores the time a project was most recently modified. Which brings us to the first idea that I considered (but never implemented).

In this initial design idea, the system stores a timestamp that indicates: “projects that have not been modified since the timestamp are accurately reflected in the search index”. For example, if the timestamp is \( t_2 \), project A was modified at \( t_1 \), project B was modified at \( t_3 \), and \( t_1 < t_2 < t_3 \), then project A does not need to be indexed, but project B does, since its modification time is after \( t_2 \). Specifically when the Crawler starts a crawl/run through the projects table, it notes the current time as `start_timestamp`. The Crawler also has the `timestamp` from the last run. Then, for each project that has been modified since `timestamp`, index the project if it is shared, otherwise delete it from the index if it is not shared. The Crawler needs to do the delete for projects that are not shared, in case the project was unshared since the last timestamp, which means it needs to be removed from the index (since we don’t want it to show up in the search results). When the run completes, then replace `timestamp` with `start_timestamp`. This
last step acts as sort of the commit point. One needs to use the time of when the run began, because otherwise the Crawler could miss a project that was modified during this run in the next run. The benefit of this approach is that the Crawler can store the state itself and the state is small, i.e. only a timestamp. There are a few downsides to this approach. Projects that are private (unshared) and modified still get sent to Elasticsearch to delete from the index, although these projects are not in the index to begin with. Only around 1 in 8 projects is shared, so these types of projects would use up a lot of unnecessary bandwidth. Another issue is that the state gets updated only after a complete and successful run, which makes this a very coarse commit point (the point that the Crawler says that it has indexed all project that have been modified before some time). This means that if a run fails, then all the work from the run will get repeated.

The next iteration of the design was to keep track of each project individually, as opposed to keep track of all projects in a single state variable. The idea is to add a column to the projects table named something like **index_fresh**. The column should reflect if a project is fresh in the index or not, in which case it needs indexing. Given such a column, the Crawler can grab all projects that have **index_fresh** set to **false** and index the projects that are shared and delete the projects that are not shared and then set **index_fresh** to **true**. Using triggers, when a project is modified and it is or was shared, then the trigger updates **index_fresh** to **false**. This has the advantage that only projects that need to be sent to Elasticsearch for indexing/deleting are being sent and that the state of each project is stored. The downside of this approach is that one needs to add a column to a table that has over 90 million rows. The projects table schema needs to be changed, which would lock the table and would require downtime of the site. This approach goes against the goal of minimizing modifications to existing infrastructure of the website.

The approach is on a good path already and there is a fairly easy fix to avoid adding a new column to the existing projects table. Instead one can create a new **state table** that has columns **id**, **project_id**, and **index_fresh**. The table reflects the state of each project in the same
way described in the approach with the `index_fresh` column. Again use triggers to mark a project’s `index_fresh` state as `false` when it gets modified and is or was shared. Using this approach there is a row in the state table for each project, including projects that are not shared. The Crawler goes through the state table and for projects that need indexing it fetches the project’s data. A single crawl still needs to go through 90 million rows. It is more space and time efficient to only keep the state of projects that are shared in the table, which is around 12 million projects. The benefit is that a crawl is faster and therefore the index will be fresher, which means that the time it takes for a change to a project to be reflected in the index is smaller.

### 3.4.2 Projects Indexed State

This subsection explains the current implementation of the Crawler and the state table that keeps track of which projects need to be indexed.

In the database, we added a table that keeps track of the state of all shared projects for the Crawler. Each project has a flag named `index_fresh` that tells the Crawler if that project needs to be indexed. The Crawler crawls the state table, in batches, and for each project that has `index_fresh` not set, the Crawler gets the project data and if the project is shared, then it prepares the document for Elasticsearch to index. If the project is not shared, then the Crawler tells Elasticsearch to remove the project from the index. When the Crawler indexes or removes a project from the index, it sets the `index_fresh` flag for that project. When a project is modified, then, if the project is or was shared, a trigger sets the `index_fresh` flag to `false` using an upsert operation (update or insert operation) in the state table.
3.4.3 Populate State

The beginning of section 3.4.1 explained the need to keep state of whether a project needs to be indexed and the final design of the state table was described in section 3.4.2. In the stable case when the Crawler has been running for a while and the state table is populated with all shared projects, the current implementation works well. Let it be noted that a project that is newly created and shared is added to the state table by the upsert trigger. What is not accounted for merely by the state table and the trigger, are projects that were shared before the Crawler was started and therefore will never be added to the state table by the trigger. This section describes the designs for populating the state table with old shared projects.

The first attempt was to create the state table using a `CREATE TABLE ... SELECT mysql` command, which would have created the table with initial data from the projects table. The issue was that this would lock the projects table while performing the operation, which would cause downtime for estimated 30 minutes.

Instead the Crawler received the responsibility of populating the state table in batches in a separate process. Initially the trigger that set the `index_fresh` flag to `false` was only an update operation, not an upsert. Therefore the populate task would run repeatedly and would populate the state table with shared projects that were not represented in the state table yet. After the first run of the populate task, when all old shared projects were added to the state table, the populate task, which goes through the whole projects table, only finds newly shared projects very sparsely. A populate task run takes around 40 hours, which meant that a newly shared project can take over 40 hours until it appears in the search index (40 hours until it is in the state table and then another few hours until the Crawler indexes the project). The solution to remove this 40-hour delay is to run the populate task only once when the state table is newly created and to use an upsert trigger that inserts newly shared projects once the state table is created.
In the implementation, the Crawler has a command line tool to start the populate task manually. The populate task is run once, when the Crawler is first setup and the state table is empty. Old shared projects get added to the table using the populate task. Newly shared projects get added to the state table by the trigger.

3.5 Api

The Api exposes different types of search capabilities -- for example, searching for a project based on some term (e.g. spoof) with some set of filters (e.g. language) by newest. The Api has various functions that all execute a search on the Elasticsearch instance. Elasticsearch scores search results based on a relevance function (see chapter 4 for more details) and returns results in order of ranking. Each Api function accepts the following parameters: query terms, filters, limit, and offset. The query term is the search string (e.g. “spoof”). The filters are things such as language and project tags. The limit defines how many results should be returned. The offset defines which search result to return first. For example, if the limit is 10 and the offset is 0, then the first 10 results will be returned, 1 through 10. If the limit is 10 and the offset is 5, then results 6 through 15 will be returned.

The Api functions vary in the way the score search results. One implemented function is ‘explore’, which puts emphasis on returning recent projects that are popular. The goal is to let users search for recent projects that they might enjoy. The Api also has a ‘test’ function that is special. It is only used during development on a staging environment and accepts a relevance function that is used for scoring search results. This allows testing the different relevance functions, before hard coding them into the Api. Chapter 4 goes into detail about scoring and the relevance function.
Chapter 4

Relevance Function and Scoring

A functioning search machine needs to find documents that match a query and needs to rank the matching documents, such that better matching or more relevant documents are ranked higher. Elasticsearch can both find documents that match a query and ranks them based on a score. This chapter first describes how Elasticsearch scores documents by default in 4.1, then goes on to explain how the scoring can be customized using a relevance function in 4.2, and finally, in 4.3, introduces the visual tool, **FIRST**, I built to aid in the process of defining useful relevance functions.

4.1 Default Scoring

In the Elasticsearch documentation under “Theory Behind Relevance Scoring” [2] the scoring of search results is explained and can be referenced for more in depth explanation. When conducting a search query, Elasticsearch first finds a list of matching documents using the Boolean Model. For example, `scratch AND cat AND sprite AND (project OR studio)` would match documents that have the three terms `scratch`, `cat`, and `sprite` and either `project` or `studio`. Once the list of matching documents has been found, then the results get scored and ranked. Not all terms will exist in all results and some terms may be more
important than others. Elasticsearch computes scores for documents based on “Term Frequency/Inverse Document Frequency (TF/IDF)” and “Vector Model” for multi term queries.

Term Frequency/Inverse Document Frequency (TF/IDF) at index time assigns weights to each term in a document. The weight of a term is computed using 3 factors: term frequency, inverse document frequency, and field-length norm. The term frequency assigns a higher weight to a term that appears more often in a document, since the term probably plays an important role in the document, if it appears often. For example, if the word “cat” appears 10 times in a document, the document probably is about cats, but if it only appears once in a document, then cats is probably not an important word in the document. Inverse document frequency counts how often a term appears in all documents in the collection and assigns a higher weight to terms that appear less often in all documents. A term such as “the” gets a low score from inverse document frequency, since “the” is a common term that appears in most documents. The last factor field-length norm assigns higher weights to terms that appear in a short field. That means if a word appears in a short title, it receives a higher score, than if the same term appears in a long body field. This makes sense, since terms in a short title are probably more relevant to the document. The title of this document is “Design and Implementation of Search for the Scratch Online Community”. If someone were to search for the word “Search”, odds are this document is more relevant than a different document that contains the word “search” once in the sentence “I search for her” in the long body of the document.

TF/IDF assigns weights to each term of a document at index time. When a single term query is conducted, Elasticsearch can use these weights to determine the highest ranked document. When a multi-term query is made, Elasticsearch uses the Vector Space Model to compute the score of matching documents. The main idea here is that each term in the query gets a weight that is higher if a term is rare. The weights together form a vector. Each document in the search results receives a similar vector, consisting of the weight of each query term that appears in the document. The score of a document is computed using the angle between the query vector and the document vector. A smaller angle results in a higher score. This may be easier to understand
using an example. In this example, the query is “funny cat” and assuming “funny” is a more common term than “cat” and that their respective weights are 2 and 5. These weights form a two-dimensional vector $[2, 5]$. The index contains three documents and each gets a vector depending on which terms are found in the document:

1. This is funny. -- $[2, 0]$
2. My favorite pet, Scratch the cat. -- $[0, 5]$
3. Scratch is the name of a funny cat. -- $[2, 5]$

The angles between the query vector and the documents vectors are:

1. $\tan^{-1}(5/2) = 1.19$
2. $\tan^{-1}(2/5) = 0.38$
3. 0.00

Document 3 would be ranked highest, since the angle between the vectors is the smallest, followed by document 2 and 1.

This section briefly explained the methods Elasticsearch uses to score and rank search results by default, and the main takeaway is that Elasticsearch does a good job at ranking search results such that results whose text are more relevant to the search query get ranked higher. The next section 4.2 describes relevance functions and how Elasticsearch allows the default scoring to be tweaked or replaced with custom scoring functions such that, for example, popularity or recency of documents can be factored into the ranking of search results.

### 4.2 Relevance Function

When scoring search results, Elasticsearch does not take factors such as recency, popularity, or moderation status into account by default. Elasticsearch provides many ways to adjust and customize the scoring of search results by defining a relevance function that can be combined with the default scoring mechanism, which was described in 4.1. The `function_score` query can be used for the purpose of passing Elasticsearch a relevance function at query time.
The fact that the scoring can be adjusted at query time, as opposed to at index time, is very useful, as it makes defining a dynamic API possible and additionally testing different functions is easy, since the documents do not need to be reindexed whenever a change to the relevance function is made. To see the complete documentation of the `function_score` query, please see the documentation [3].

Figure 4-1 shows the structure of the `function_score` query. The general idea is to define a list of functions where each function on its own computes a value given a document. The functions are then combined based on `score_mode` and their respective weights to form the custom score, which in turn gets combined with the default score based on the `boost_mode`. In more mathematical terms, say there are three functions defined $f_1$, $f_2$, and $f_3$ with respective weights $a_1$, $a_2$, and $a_3$. Then the score of a document $d$ would be:

$$ _{score}(d) = \text{boost_mode}(\text{default_score}, \text{score_mode}(a_1 f_1(d), a_2 f_2(d), a_3 f_3(d))) $$

The rest of this section explains `boost_mode`, `score_mode`, and different types of functions that exist.

`boost_mode` is a function that accepts two values: the default score and the custom score. Options for `boost_mode` are `multiply`, `sum`, `avg`, `max`, `min`, or `replace`. `replace` ignores the default query score and only returns the custom score.
score_mode is a function that accepts multiple values, namely the weighted outputs of all the functions. score_mode can be multiply, sum, avg, max, min, or first.

functions is a list of weighted functions. There are a variety of functions that Elasticsearch offers, most notably field_value_factor, script_score, and decay functions (exp, gauss, linear). script_score allows the client to pass Elasticsearch a script to compute a custom score, but by default scripts are disabled, for security purposes. When possible, I would recommend using functions that Elasticsearch offers and avoiding the use of scripts.

field_value_factor can use certain fields of a document to influence the score. Often it is used to boost popular search results. Scratch projects, for instance, can be boosted using love counts. field_value_factor can be defined based on three parameters: field, factor, modifier. The field defines what field should be used, the factor defines a multiplier of the field value, which gets modified by the modifier. modifier can be none, various logarithmic functions, square, sqrt, or reciprocal. Using the Scratch projects popularity based on love count example again, one could define {field_value_factor:
  {field: "love_count", factor: 1.2, modifier: "sqrt"}}, which would translate into the following formula:

\[ f(d) = \text{sqrt}(1.2 \times d["love\_count"]) \]

“Decay functions score a document with a function that decays depending on the distance of a numeric field value of the document from a user given origin”[3]. For Scratch projects, this is useful in scoring recently shared projects, using the timestamp datetime_first_shared and defining the origin as “now”.
4.3 FIRST Visual Tool

In the previous section 4.2, the tools to customize the scoring of search results in Elasticsearch were introduced. Defining a relevance function can be challenging and often involves some guesswork. To aid in the process of finding and defining an appropriate relevance function for different type of searches, I developed the visual tool FIRST (Fast Iterative Relevance Search Tool) that allows the user to test different relevance function configurations and observe the resulting ranked search results. This section describes FIRST. First the motivation of the tool is discussed, followed by a description of the capabilities of the tool. The section concludes with a use case of FIRST.

4.3.1 Motivation

Users of search machines tend to only look at one of the first few search results. Therefore it is important to show results that match what the user is looking for closer to the top. Defining a relevance function is not a straightforward process and usually involves guesswork. Even after deciding on a basic structure of the relevance function and what fields should affect the score, tweaking the parameters can be tricky. The goal is to score search results, such that search users will find what they are looking for quickly and close to the top of the search page. Two common behaviors when a user is not finding what they are looking for are: clicking on a link and returning back to the search page soon after; and trying many different search queries with slight variations in terms used.

It is important to keep tweaking and optimizing the relevance function, even after deployment. A common way of testing how a tweaked relevance function is performing is to track how users behave, how often they pick one of the first \( n \) (say 5) results, how often they attempt searching with a modified query, and so on. One can do A-B testing, where a group of users use an old relevance function \( a \) and the second group uses a new tweaked relevance function \( b \). Comparing
the two groups and how successful search is working for them can indicate which relevance function, old or new, works better for users.

Initially one has to guess a relevance function structure and parameters, and test it on a handful of example queries, before deploying. This can be done using the `curl` command line tool and then reading the raw json response of Elasticsearch. This can be a slow process, when playing around and having to reformat the raw search request’s `json`. When working with a team including people who are unfamiliar with reading raw `json`, but are very familiar with the documents, such as curators, it can be a tough process to keep having to modify the `json` request and then spending time reading and evaluating the Elasticsearch server’s response. The visual tool FIRST was developed with the goal of addressing many of these issues.

FIRST is especially useful for teams to work together to define a first relevance function for deployment, or for testing of newer/tweaked relevance functions. Elasticsearch offers many features for scoring documents that are often not needed in certain use cases. As a result, FIRST can simplify the process by limiting the features to ones that are potentially useful.

### 4.3.2 Features

FIRST is a convenient internal visual tool used during development and testing of the search machine, and especially when developing and tweaking a relevance function. FIRST can serve as a search interface prototype, but its main strength is that it allows you to specify your own relevance function. Currently FIRST is implemented specifically for projects. The rest of this subsection provides an overview of the capabilities of FIRST.

FIRST accesses search through the search Api, just as any search interface later on should. The Api exposes different types of search functions that differ in their relevance function. FIRST is a special use case and needs to be able to provide its own relevance function to the Api. Therefore
FIRST has a variety of features and capabilities. Overall the FIRST page can be broken up into three components: the tool, which includes search bar, boost and score mode, and number of results per page and page number; the relevance function definition area, which allows the user to specify how certain fields of a project should contribute to the score; and the search results area, which shows the search results including the scores. Figure 4-2 shows a screenshot of the tools section and relevance function definition area. Figure 4-3 is a screenshot of the search results area and contains the first four results to the query “star wars” with the relevance function defined in Figure 4-2. As a side note, the relevance function in 4-2 is an initial guess for the “explore” search, which is further discussed in 4.3.3. One can see that a lot of
Figure 4-3: FIRST screenshot of search results of “star wars” query. Each result has the following information: 1. _score; 2. Title; 3. Description; 4. Thumbnail; 5. _source field, which includes all data stored with a project in Elasticsearch.

emphasis is put on recent projects, since the weight of the function that uses datetime_first_shared is much greater than the weight of the other fields.

The user inputs the search query in the search bar (1 in figure 4-2). The number of search results per page and page number can be specified from a drop-down menu (3 in figure 4-2). If results per page is 10 and the page number is 2, then results 11 through 20 are shown, for example. Additionally the user can specify the boost_mode and score_mode through drop-down menus (2 in figure 4-2). Lastly there is the relevance function definition area (4 in 4-2), which allows the user to specify how certain fields should influence the score of search results. This is part of the true power of FIRST. I was able to decide which project fields may be useful and how they would be used for scoring (decay function, field value factor, or simple weight). By doing this work ahead of time, the problem of finding an appropriate relevance function gets
simplified, since the users can see the fields and the type of scoring functions that are available to them. Specifically the count fields (favorite_count, love_count, view_count) reflect popularity of a project and as such using these counts to boost a project’s score using the field_value_factor function made sense. datetime_first_shared reflects the first time a project was shared and with that how old a project is. Therefore this field can be used to give projects that are more recent a higher score and the FIRST user should use a decay function to score recent projects higher. It should be mentioned that datetime_shared, as opposed to datetime_first_shared, was deliberately not used to reflect how old a project is, because otherwise a Scratch user could theoretically unshare and reshare a project to fake the recency. moderation_status is an enum field and can have any of five values (notreviewed, safe, censored, delbyadmin, unsafe). The moderation status of a project could be used to award bonus points to projects with a positive moderation status. Using the normal weight function made sense here. Initially FIRST allowed the user to set a weight for all five moderation statuses, but was changed to only accept weights for notreviewed and safe. Projects that have a negative moderation status should not show up in search results and the Api was adjusted to filter out these projects.

In the search results section, the user can view a list of search results depending on page number and results per page, by default the top 10 results. Each project has a collection of information, such as the thumbnail (4 in figure 4-3), title (2 in figure 4-3), and description (3 in figure 4-3). The raw_source of a project (5 in figure 4-3) contains all the fields stored under a project. Additionally the _score of each document/project is displayed (1 in figure 4-3), which is helpful when deciding on relevance function parameters.

Elasticsearch queries can “explain” the scores of search results. As a suggested extension, FIRST could conduct queries with “explain” and display the explained scoring of projects.
4.3.3 Use Case: Generating “Explore” Search Function

This subsection describes the process some of the Scratch team members went through when using FIRST to come up with an initial relevance function for explore search. The main idea behind explore, is to allow Scratch users to discover new projects that are of interest to them. Therefore the recency of a project is most important, but also the favorite, love, and view counts are an indicator if a project is, or is becoming, popular amongst the community.

Before starting the process, I gave team members an overview of how Elasticsearch scores search results from a high level. This is important so that the team members are aware of what Elasticsearch does by default (such TF/IDF and field-length norm). After, we looked up common search queries in existing Scratch search and picked one query to use to develop the relevance function against. We used the query “spoof”. At the end of the process the relevance function was tested against other search queries to tweak and validate.

At first we set the boost_mode to replace in order to ignore Elasticsearch default scoring and focus on the custom scoring. This way we could decide on relative weights of different fields, regardless how good of a match a project was to the search query, as long as the project matched the search query. We set the score_mode to sum.

We focused on the three count fields (favorite, love, and view) and decided on their relative weights. We decided to assign relative weights to the favorite, love, and view counts of 0.1, 0.5, and 0.4 respectively. Next we focused on the recency of a project, using datetime_first_shared field, and decided to use the exponential decay function with around 15 day scale and 0.8 decay. We tuned the weight of this field, until we were predominantly getting recent projects. With this setting we could observe that top result were usually no more than a week old, and some popular projects (high counts) were up to two weeks old. When this step was done, the scores of the first few search results were composed of around
1/9 popularity and 8/9 recency. This 1 to 8 ratio of the popularity and recency was what we were targeting.

In the last step, we combined the relevance function with the default scoring of Elasticsearch. Initially we set the boost_mode to sum and then scaled the weights of all the fields until the overall score was mostly computed by the custom scoring function. The overall function worked well for the search query “spoof”, but did not work well for a query as “happy hippopotamus” or other multi-term queries that included one common term and one less common term. These two term queries often would yield projects that only contained the common term, although the user presumably is more interested in projects that contain the uncommon term. Hence, we changed the boost_mode to multiply, which works better for a variety of search queries.

The resulting relevance function probably will undergo a few more iterations of tweaks, but even this initial guess is ranking search results in such a way that new projects that are receiving good feedback from the community, in terms of views and loves, are ranking higher.
Chapter 5

Conclusion

5.1 Future Work

The Search infrastructure for Scratch outlined in this thesis is mostly ready to be integrated into the Scratch website, though there is space for more work and improvements. The system is designed and implemented, such that most of the following suggested future work should be fairly straightforward to implement and the existing system should not need much modification.

Search is currently only implemented for projects. In the future, other Scratch entities will be searchable using the new search infrastructure, such as users and forums. To index, say, Scratch users, first one needs to create a index state table and add triggers and procedures to mark user profiles that need to be indexed. The new user type needs to be saved in Elasticsearch. The Crawler’s program structure will be mostly identical and only the database and Elasticsearch queries will need to be modified to account for the different database tables and fields.

In general adding more fields and metadata to projects in the index can open the possibility of new search functionality that was not possible with the old system. In the future, adding metadata about program blocks or techniques that are being used in a project will allow Scratch users that are looking to learn about a block or technique to find example projects. Another idea
would be to rate project complexity, such that novice users could discover simpler projects and advanced users could discover more complex projects.

The current design and implementation of Search focused on the backend and functionality, but search machines need to have a user interface. The user interface needs to be designed and implemented. The website’s search interface will access the search functionality through the Api. In 2012 another Masters of Engineering student did work on a search user interface for Scratch, which might be useful during this process [7].

The search infrastructure was built with scalability in mind to account for the growing number of users and shared projects. As mentioned in chapter 3, the Elasticsearch cluster and Api are scalable. The Crawler will be scaled in the future, to allow quicker indexing of documents. The Crawler was designed in such a way that dividing up the work it does into chunks is possible. Specifically, the Crawler crawls batches of documents defined by a range of ids. There are some initial ideas of how one could scale the Crawler. Each chunk of work is independent and therefore a system with a distributed task queue, such as Celery [8], will probably work.

5.2 Conclusion

This thesis presents the design and implementation of the new Search infrastructure for the Scratch website. It aims to replace the current Scratch Search page and potentially the Scratch Explore page’s backend. The system is designed in such a way that it can easily be extended to other use-cases within Scratch and even outside Scratch. My hope is that a reader looking to implement their own Search system for their website will find inspiration in the design presented in this paper. Finally, this thesis also presents FIRST, a visual tool to aid teams and developers in the process of defining a suitable relevance function for ranking search results.
Bibliography


