An Examination of the Parallelization of IR Metrics with Apache Spark A Use Case

Chase Greco
Virginia Commonwealth University
Richmond, Virginia, USA
grecocd@vcu.edu

***Abstract*—The abstract goes here.**

I. Introduction

Software developers often consult the Web when searching for solutions to development issues they encounter. The Stack Overflow Q & A forum, is one of the largest and most popular software development knowledgebases, with over 40 million monthly visitors, with an estimated 16.8 million professional developers and university students [].

With such a large knowledgebase available, the potential exists for recommendation systems to increase developer productivity by managing and extracting those nuggets of information necessary to their task at hand. Several recommendation tools aimed at Stack Overflow have been proposed with this goal by integrating relevant information from Stack Overflow directly into the IDE, such as Prompter [] and SeaHawk [] which automatically recommend posts from Stack Overflow based on the context of the source code within the IDE.

In this paper we examine StackInTheFlow, a tool we have previously proposed which intends to automate the task of discovering relevant Stack Overflow posts. StackInTheFlow is a personalized recommendation system which ties closely with developer behavior within the IDE, allowing them to remain in the flow of development. It does this through several mechanisms, however, for the purposes of this paper we will focus on one such mechanism the **Auto Query** feature and examine the benefits parallelization via Apache Spark can provide it.

The structure of the remainder of this paper is as follows:

- We present an overview of the Auto Query feature of StackInTheFlow, it's use case, and the procedure by which it generates queries
- We propose parallelization via Apache Spark as a method to speed up computation within the Auto Query generation process
- We examine the performance gains between our sequential and parallel implementations
- We offer some conclusions as to the viability of this method

II. Auto Query Generation

A. Use Case

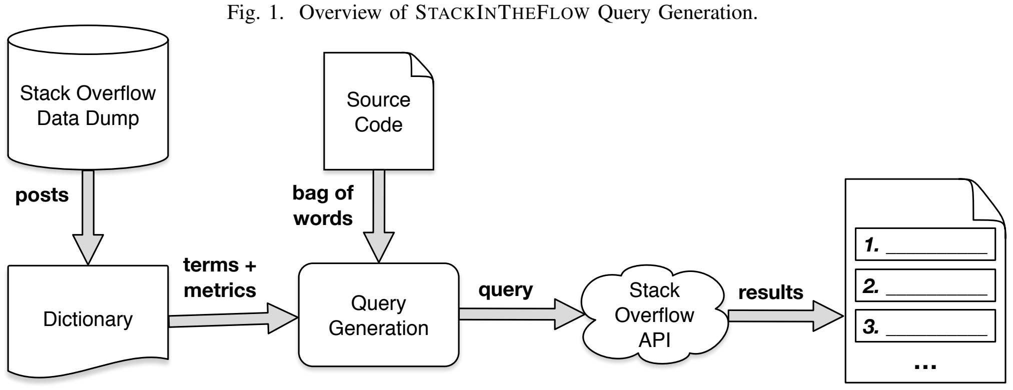
It may happen that when searching for the solution to a development problem the developer may not be able to form a query suitable to retrieve the information necessary to form a solution and may require assistance in query formulation. For example, the developer may be seeking information on the configuration options for the *SparkSession* object. In such a case she may highlight the section of code relevant to declaring or utilizing this object, right-click within the editor and select the *Auto Query* option. Utilizing a procedure detailed in Section II-B, StackInTheFlow will then generate a query from the selected snippet which it then sends to the Stack Overflow API to retrieve relevant results and display them within the IDE. From there she may browse the results, filter them by tags, or sort them by four different criteria: relevance, newest, last active, and number of votes.

B. Query Generation Procedure

StackInTheFlow utilizes a four-step procedure for generating queries illustrated by Figure 1. In the first step, a dictionary is constructed offline from the posts contained within the Stack Overflow Data Dump. The data dump contains all user posts within the Stack Overflow site and is published periodically by Stack Exchange. This dictionary is composed of terms extracted from the posts and for each term the *collection term frequency*, *document frequency*, *inverse collection term frequency*, and *inverse document frequency* are stored. Terms are selected from the body of posts based on the following criteria: They must begin with an upper or lowercase letter and they must be at least 2 characters in length. Upon storage in the dictionary, all terms are normalized to be lowercase.

In the second step, when a snippet of code is received it is converted into a bag of words. For each term within the bag of words, the corresponding metrics contained within the dictionary are examined. From these metrics several query pre-retrieval quality metrics such as *tf-idf* are calculated and linearly summed to form an overall score for the term. Terms not contained within the dictionary are discarded. The top four scoring terms are then selected to form the candidate query.

Fig. 1. Overview of STACKINTHEFLOW Query Generation.



In the third step, the candidate query is sent to the Stack Overflow API to retrieval an initial results listing.

In the fourth step the results listing is retrieved as sent for additional processing before displaying it to the user. If the candidate query returned no results and back-off technique is employed by incrementally removing lower scoring terms from the query.

III. Apache Spark Parallelization

The offline indexing of the Stack Overflow Data Dump represents a significant commitment of computational resources to achieve, taking several hours to complete. Though the computation is done offline and their is no direct impact to the user, it must be repeated each time a new data dump is released so that the model does not grow stale over time. Due to the continued investment of resources necessary re-index future data dumps, efforts to parallelize this approach may be quite useful. Once more, the problem is further compounded by the ever-growing amount of posts contained within Stack Overflow itself, meaning future data dumps will be even more resource-intensive to process. In addition, decreasing the indexing run-time may allow for more sophisticated metrics to be computed and thereby enable a more complex recommendation model.

The current implementation of the indexer relies on a sequential parser. The parser examines each post, one after the other, and keeps a running total of the *collection term frequency* and *document frequency* of the terms within the post. After processing all posts the *inverse collection term frequency* and *inverse document frequency* are calculated.

This approach, while effective, is a prime candidate for parallelization utilizing the *MapReduce* paradigm of Apache Spark, as each post can be processed separately in parallel with the resultant calculations reduced to form the final metric calculations. We implement this approach in the following steps:

1) For each post, map it's body into a bag of words, keeping track of the post from which they originate
2) Filter each bag of words, performing term selection via the criteria previously described
3) For each (post, term) paring, emit a one
4) Perform a reduction on the term and compute a sum of each corresponding one to calculate *Collection Term Frequency*
5) Repeat the previous step for distinct (post, term) parings to calculate *Document Frequency*
6) From these metrics, calculate *Inverse Collection Term Frequency* and *Inverse Document Frequency*

To examine this approach, we compare its runtime to that of the sequential parser for several sub-sets of the Stack Overflow Data Dump of varying size.

IV. Results

V. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...