Information Retrieval Meets Scalable Text Analytics: Solr Integration with Spark

Ryan Clancy, Jaejun Lee, Zeynep Akkalyoncu Yilmaz, and Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

ABSTRACT

Despite the broad adoption of both Apache Spark and Apache Solr, there is little integration between these two platforms to support scalable, end-to-end text analytics. We believe this is a missed opportunity, as there is substantial synergy in building analytical pipelines where the results of potentially complex faceted queries feed downstream text processing components. This demonstration explores exactly such an integration: we evaluate performance under different analytical scenarios and present three simple case studies that illustrate the range of possible analyses enabled by seamlessly connecting Spark to Solr.

ACM Reference Format:
Ryan Clancy, Jaejun Lee, Zeynep Akkalyoncu Yilmaz, and Jimmy Lin. 2019. Information Retrieval Meets Scalable Text Analytics: Solr Integration with Spark. In *42nd Int'l ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '19), July 21–25, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3331184.3331395

1 INTRODUCTION

In the realm of data science, Apache Spark has emerged as the *de facto* platform for analytical processing, with broad adoption in both industry and academia. While not originally designed for scalable *text* analytics, it can nevertheless be applied to process large document collections in a scalable, distributed fashion. However, using Spark for text processing is hampered by the lack of integration with full-text indexes, particularly useful in applications where the data scientist wishes to analyze only a subset of the collection. By default, the only approach for selecting a collection subset is a brute-force scan over every document with a filter transformation to retain only the desired documents. For selective queries that only match a small number of documents, this is obviously inefficient.

In the realm of search, Apache Solr has emerged as the *de facto* platform for building production applications. Other than a handful of commercial web search engines that deploy custom infrastructure to achieve the necessary scale, most organizations today take advantage of Solr, including Best Buy, Bloomberg, Comcast, Disney, eHarmony, Netflix, Reddit, and Wikipedia. Although Solr is designed to be scalable via a distributed, partitioned architecture, the platform is primarily engineered around providing low-latency user-facing search. As such, it does not provide any analytics capabilities *per se.*

The current state of the broader ecosystem sees little overlap between Spark for general-purpose analytical processing on the one hand and Solr for production search applications on the other. This is a missed opportunity in creating tremendous synergies for text analytics, which combines elements of search as well as analytical processing. As a simple example, the output of a (potentially complex, faceted) query can serve as the input to an analytical pipeline for machine learning. Spark, by default, cannot take advantage of index structures to support efficient end-to-end execution of such pipelines. Lin et al. [9] previously described using Lucene indexes to support predicate pushdown optimizations in the Pig scripting language, but the approach never gained widespread adoption and Pig has generally given way to more modern analytics platforms such as Spark. Terrier-Spark [10] represents a recent effort to integrate Spark with an IR engine, but the project has a slightly different focus on building experimental IR pipelines as opposed to general-purpose text analytics. Nevertheless, we draw inspiration from Terrier-Spark, particularly its integration with Jupyter notebooks to support interactive explorations.

The contribution of this demonstration is an exploration of how Solr and Spark can be integrated to support scalable text analytics. We investigate the performance characteristics of using Solr as a document store, taking advantage of its powerful search functionality as a pushdown predicate to select documents for downstream processing. This is compared against an alternative where the raw documents are stored in the Hadoop Distributed File System (HDFS), requiring brute-force scans over the entire collection to select subsets of interest. As expected, our results confirm that significant performance improvements are realized when Spark leverages Solr's querying capabilities, especially for queries that match few documents. With the addition of simulated per-document processing workloads, Solr is unequivocally faster—even over large fractions of the entire collection—since processing costs mask I/O costs.

With Solr/Spark integration, we present three case studies that illustrate the range of interesting analyses enabled by end-to-end text processing pipelines. These examples include kernel density estimation to study the temporal distribution of tweets, named-entity recognition to visualize document content, and link analysis to explore hyperlink neighborhoods.

2 SOLR/SPARK INTEGRATION

To integrate Solr with Spark, we adopt the most obvious architecture where Solr serves as the document store. We assume that a document collection has already been ingested [2].

Solr is best described as a REST-centric platform, whereas Spark programs define sequences of data-parallel transformations (e.g.,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGIR '19, July 21–25, 2019, Paris, France
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6172-9/19/07…$15.00
https://doi.org/10.1145/3331184.3331395

`filter`, `map`, etc.) over a data abstraction called Resilient Distributed Datasets (RDDs). The crux of the integration effort thus involves bridging Solr output (i.e., the result of search queries) with RDDs. To this end, we take advantage of an open-source library called `spark-solr`,[1] which constructs RDDs from Solr search results (not surprisingly, called SolrRDDs). This library leverages appropriate Solr indexes and exploits data locality by fetching documents from the Solr shard that is co-located with the same node as the Spark worker requesting the documents.

As part of initial explorations, we compared the performance of SolrRDD to a much simpler approach of mapping over an RDD of docids (that is, strings) and using the direct Solr APIs to fetch the documents to create the initial RDD for Spark processing. This seemed like a reasonable experiment since there is evidence that with modern networks and the trend toward storage disaggregation [5], data locality is no longer a serious concern [1]. Nevertheless, we found that SolrRDD was substantially faster than our "parallel document fetch" implementation, and thus we focused our efforts building on SolrRDD.

From a performance perspective, the baseline for comparing Solr/Spark integration is Spark processing over documents stored on HDFS. In this setup, all document manipulations require scanning the entire collection. Previous studies have found that such brute-force operations are not as inefficient as one might think at first glance; for example, researchers have explored document ranking using this architecture [4, 6]. Such designs lead to simple implementations that enjoy excellent data locality and can take advantage of high disk throughput.

Intuitively, we can characterize the performance tradeoffs between the two alternatives as follows: Suppose a data scientist wishes to process the entire collection. Fetching documents from Solr will likely be slower than simply scanning the entire collection on HDFS sequentially, since Solr index structures come with associated overheads. At the other end of the spectrum, suppose a data scientist is only interested in analyzing a single document. In this case, Solr would be obviously much faster than scanning the entire collection. Thus, the *selectivity* of the document subset, in addition to other considerations such as hardware configurations and the processing workload, will determine the relative performance of the two approaches. The balance of these various factors, however, is an empirical question.

3 PERFORMANCE EVALUATION

We set out to empirically characterize the performance tradeoffs between the designs discussed in the previous section, principally examining two characteristics: selectivity and workload.

3.1 Experimental Setup

Our experiments were conducted on a cluster with ten nodes. Each node has 2× Intel E5-2670 @ 2.60GHz (8 cores, 16 threads) CPUs, 256GB RAM, 6×600GB 10k RPM HDDs, 10GbE networking, running Ubuntu 14.04 with Java 1.8. One node is responsible for running the master services (YARN ResourceManager, HDFS NameNode) while the remaining nine nodes each hosts an HDFS DataNode, a Solr shard, and are available for Spark executor allocation (via YARN).

[1] https://github.com/lucidworks/spark-solr

Note that our processors are of the Sandy Bridge architecture, which was introduced in 2012 and discontinued in 2015, and thus we can characterize these computing resources as both "modest" and "dated". Similar amounts of compute power could be found on a single high-end server today.

We examined the following document collections:

- The New York Times Annotated Corpus, a collection of 1.8 million news article, used in the TREC 2017 Common Core Track.
- Tweets2013, a collection of 243 million tweets gathered over February and March of 2013, used in the TREC Microblog Tracks [8].
- ClueWeb09b, a web crawl comprising 50.2 million pages gathered by CMU in 2009, used in several TREC Web Tracks.

All collections were ingested into Solr using Anserini [2, 13, 14]. For comparison, all collections were also loaded into HDFS. In both cases, the same document processing (e.g., tokenization, stopword removal, etc.) was applied using Lucene Analyzers.

Our performance evaluation focused on two characteristics of large-scale text analytics: the number of documents to process (selectivity) and the per-document processing time (workload). In order to understand the first factor, we randomly selected terms according to document frequency, ranging from 10% to 60%. These terms were then issued as queries whose results were fed into Spark. For "processing", we simulated three different workloads by simply sleeping for 0ms, 3ms, or 30ms (per document).

While running experiments, we used the master node as the driver while running Spark jobs in client mode. Each job used 9 executors with 16 cores and was allocated 48GB of RAM per executor. This allowed us to take full advantage of the available cluster resources and exploit data locality as Spark workers were co-located on the same nodes as HDFS DataNodes and Solr shards.

3.2 Experimental Results

Figure 1 summarizes the results of our experiments on ClueWeb09b, varying input selectivity and processing workload. We report averages over five runs (where each run is based on a different randomly-selected term at the specified selectivity) and include 95% confidence intervals. Results on the other collections yield similar findings and hence are omitted.

The left bar graph in Figure 1 simulates no per-document processing and captures the raw I/O capacity of both architectures. As expected, total execution time does not vary much with selectivity when brute-force scanning the collection on HDFS, since the entire document collection needs to be read regardless. Also as expected, the performance of using Solr as a pushdown predicate to select subsets of the collection depends on the size of the results set. For small subsets, Solr is more efficient since it exploits index structures to avoid needless scans of the collection. Execution time for Solr grows as more and more documents are requested, and beyond a certain point, Solr is actually slower than a scan over the entire collection due to the overhead of traversing index structures. This crossover point occurs at around half the collection—that is, if an analytical query yields more results, a brute-force scan over the entire collection will be faster.

The above results assume that no time is spent processing each document, which is obviously unrealistic. In the middle and right

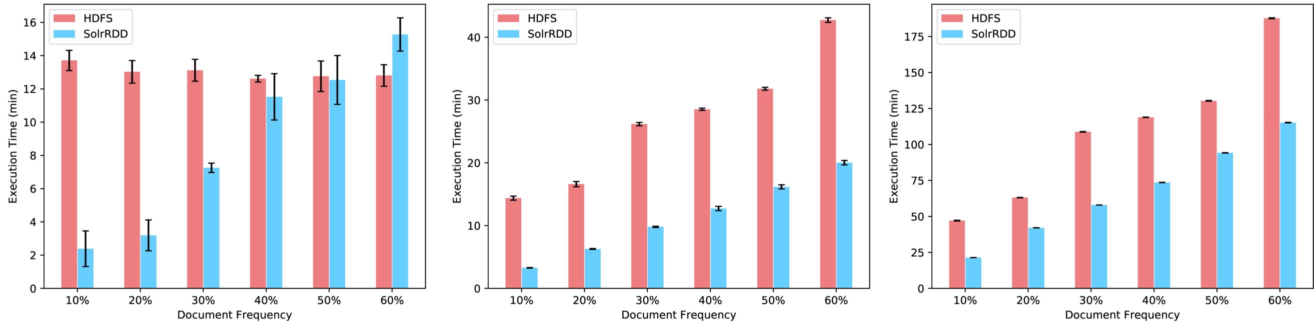

Figure 1: Average total execution time (over five trials) on ClueWeb09b with simulated per-document workloads of 0ms (left), 3ms (middle), and 30ms (right). Error bars denote 95% confidence intervals.

bar graphs in Figure 1, we simulate per-document processing latencies of 3ms and 30ms. The takeaway from these results is that Solr is *always* faster than a brute-force scan over the entire collection on HDFS. As the per-document workload increases, processing time occupies a growing fraction of the overall execution time and masks latencies associated with fetching a large number of documents from Solr. Thus, from these experiments we can conclude that, except in the most extreme case where text analytics is dominated by I/O, predicate pushdown via Solr is beneficial.

4 CASE STUDIES

We present three case studies that illustrate the range of analyses enabled by our Solr/Spark integration, taking advantage of existing open-source tools. While these analyses are certainly possible without our platform, they would require more steps: issuing queries to Solr, extracting the result documents from the collection, and importing them into downstream processing tools. In practice, this would likely be accomplished using one-off scripts with limited generality and reusability. In contrast, we demonstrate end-to-end text analytics with seamless integration of Spark and Solr, with a Jupyter notebook frontend.

4.1 Temporal Analysis

Kernel density estimation (KDE), which has been applied to extract temporal signals for ranking tweets [3], can be used to explore the distribution of tweets over time. In this case study, we investigated the creation time of tweets that contain certain keywords from the Tweets2013 collection (243 million tweets) [8].

The top graph in Figure 2 shows results for four keywords, aggregated by hour of day (normalized to the local time zone): "coffee", "breakfast", "lunch", and "dinner". The bottom graph shows results for the following keywords, aggregated by day of week: "school", "party", and "church". In all cases, we began with a query to Solr, aggregated the creation time of the retrieved tweets, and then used Spark's MLlib[2] to compute the KDE.

The results show diurnal and weekly cycles of activity. Peaks for the three daily meals occur where we'd expect, although Twitter users appear to eat breakfast (or at least tweet about it) relatively late. Coffee is mentioned consistently throughout the waking hours of the day. In terms of weekly cycles, unsurprisingly, "church" peaks on Sunday, "party" peaks on Saturday, and mentions of school drop off on weekends. The core of this analysis is around 15 lines of code, highlighting the expressivity of Solr/Spark integration.

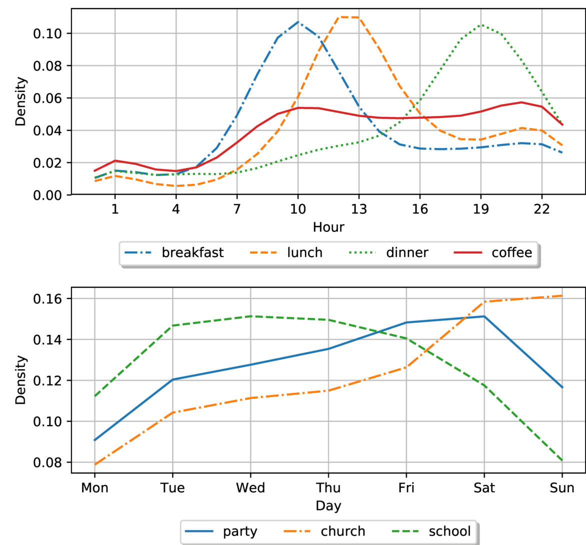

Figure 2: Results of Kernel Density Estimation on creation time of tweets to capture diurnal and weekly activity cycles.

4.2 Entity Analysis

We can take advantage of named-entity recognition (NER) to provide a broad overview of a corpus, corresponding to what digital humanities scholars call "distant reading" [12]. For example, what musical artists are prominently discussed in the New York Times? The answer is shown in Figure 3 as a word cloud, which was created by taking all documents that have the term "music" (154k documents), feeding the text into Stanford CoreNLP [11] to extract named entities, and then piping the results to an off-the-shelf tool.[3] We performed minor post-processing to remove entities that are single words, so common names such as "John" do not dominate.

[2]https://spark.apache.org/mllib/

[3]https://github.com/amueller/word_cloud

Figure 3: Word cloud for "music" from the New York Times

The people mentioned are, perhaps unsurprisingly, famous musicians such as Bob Dylan, Frank Sinatra, and Michael Jackson, but the results do reveal the musical tastes of New York Times writers. All of this can be accomplished in around 20 lines of code.

4.3 Webgraph Analysis

Network visualizations facilitate qualitative assessment by revealing relationships between entities. In this case study, we extracted links referenced by websites in the ClueWeb09b collection that contain the polysemous term "jaguar" (among many referents, a car manufacturer and an animal), which could reveal interesting clusters corresponding to different meanings. In order to produce a sensible visualization, we began by randomly sampling 1% of the documents that contain the term. Next, we extracted all outgoing links from these sources with the Jsoup HTML parser before aggregating by domain. Finally, we selected the top three most frequently-occurring links from each source node to reduce clutter. All of this can be accomplished in around 30 lines of code.

By feeding the edge list to Gephi,[4] we ended up with the network visualization in Figure 4 using a Multilevel Layout [7]. For better clarity in the visualization, we pruned nodes with small degrees. Unsurprisingly, the visualization features a large cluster centered around google.com, and multiple smaller clusters corresponding to websites associated with different meanings of the term.

5 CONCLUSIONS

In this work we have demonstrated the integration of Solr and Spark to support end-to-end text analytics in a seamless and efficient manner. Our three usage scenarios only scratch the surface of what's possible, since we now have access to the rich ecosystem that has sprung up around Spark. With PySpark, which provides

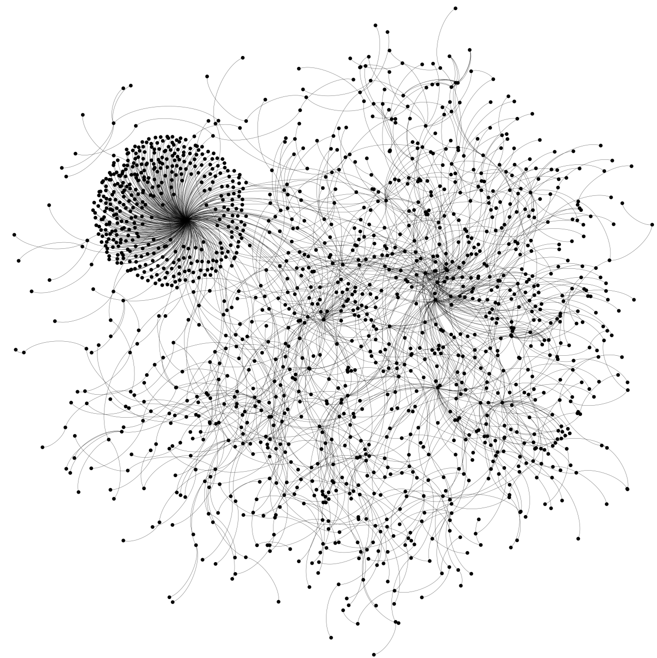

Figure 4: Network visualization for "jaguar".

Python bindings for Spark, we gain further integration opportunities with PyTorch, TensorFlow, and other deep learning frameworks, enabling access to state-of-the-art models for many text processing tasks, all in a single unified platform.

Acknowledgments. This work was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada, the Canada Foundation for Innovation Leaders Fund, and the Ontario Research Fund.

REFERENCES

[1] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. 2011. Disk-Locality in Datacenter Computing Considered Irrelevant. In *HotOS*.

[2] R. Clancy, T. Eskildsen, N. Ruest, and J. Lin. 2019. Solr Integration in the Anserini Information Retrieval Toolkit. In *SIGIR*.

[3] M. Efron, J. Lin, J. He, and A. de Vries. 2014. Temporal Feedback for Tweet Search with Non-Parametric Density Estimation. In *SIGIR*. 33–42.

[4] T. Elsayed, F. Ture, and J. Lin. 2010. *Brute-Force Approaches to Batch Retrieval: Scalable Indexing with MapReduce, or Why Bother?* Technical Report HCIL-2010-23. University of Maryland, College Park, Maryland.

[5] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *HotCloud*.

[6] D. Hiemstra and C. Hauff. 2010. MapReduce for Information Retrieval Evaluation: "Let's Quickly Test This on 12 TB of Data". In *CLEF*. 64–69.

[7] Y. Hu. 2005. Efficient, High-Quality Force-Directed Graph Drawing. *Mathematica Journal* 10, 1 (2005), 37–71.

[8] J. Lin and M. Efron. 2013. Overview of the TREC-2013 Microblog Track. In *TREC*.

[9] J. Lin, D. Ryaboy, and K. Weil. 2011. Full-Text Indexing for Optimizing Selection Operations in Large-Scale Data Analytics. In *MAPREDUCE*. 59–66.

[10] C. Macdonald. 2018. Combining Terrier with Apache Spark to Create Agile Experimental Information Retrieval Pipelines. In *SIGIR*. 1309–1312.

[11] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL Demos*. 55–60.

[12] F. Moretti. 2013. Distant Reading.

[13] P. Yang, H. Fang, and J. Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *SIGIR*. 1253–1256.

[14] P. Yang, H. Fang, and J. Lin. 2018. Anserini: Reproducible Ranking Baselines Using Lucene. *JDIQ* 10, 4 (2018), Article 16.

[4]https://gephi.org