Solr Integration in the Anserini Information Retrieval Toolkit

Ryan Clancy,[1] Toke Eskildsen,[2] Nick Ruest,[3] and Jimmy Lin[1]
[1] David R. Cheriton School of Computer Science, University of Waterloo
[2] Royal Danish Library [3] York University Libraries

ABSTRACT

Anserini is an open-source information retrieval toolkit built around Lucene to facilitate replicable research. In this demonstration, we examine different architectures for Solr integration in order to address two current limitations of the system: the lack of an interactive search interface and support for distributed retrieval. Two architectures are explored: In the first approach, Anserini is used as a frontend to index directly into a running Solr instance. In the second approach, Lucene indexes built directly with Anserini can be copied into a Solr installation and placed under its management. We discuss the tradeoffs associated with each architecture and report the results of a performance evaluation comparing indexing throughput. To illustrate the additional capabilities enabled by Anserini/Solr integration, we present a search interface built using the open-source Blacklight discovery interface.

ACM Reference Format:
Ryan Clancy, Toke Eskildsen, Nick Ruest, and Jimmy Lin. 2019. Solr Integration in the Anserini Information Retrieval Toolkit. 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/10.1145/3331184.3331401

1 INTRODUCTION

The academic information retrieval community has recently seen growing interest in using the open-source Lucene search library for research. Recent events to promote such efforts include the Lucene4IR [3] workshop held in Glasgow, Scotland in 2016 and the Lucene for Information Access and Retrieval Research (LIARR) Workshop at SIGIR 2017 [2]. In an evaluation of seven open-source search engines conducted in 2015 [4], Lucene fared well in comparisons of both effectiveness and efficiency. These results provide compelling evidence that IR researchers should seriously consider Lucene as the foundation of their work.

Advocates of using Lucene for IR research point to several advantages: building on a widely-deployed open-source platform facilitates replicability and brings academic research closer into alignment with "real-world" search applications. Lucene (via integration with Solr) powers search in production deployments at Bloomberg, Netflix, Comcast, Best Buy, Disney, Reddit, and many more sites.

Anserini [6, 7] is a recently-introduced IR toolkit built on Lucene specifically designed to support replicable IR research. It provides efficient multi-threaded indexing for scaling up to large web collections and strong baselines for a broad range of collections. The meta-analysis of Yang et al. [8] encompassing more than 100 papers using the test collection from the TREC 2004 Robust Track showed that the well-tuned implementation of RM3 query expansion in Anserini is more effective than most of the results reported in the literature (both from neural as well as non-neural approaches). A key feature of Anserini is its adoption of software engineering best practices and regression testing to ensure that retrieval results are replicable by other members of the community, in the sense defined by recent ACM guidelines.[1] That is, replicability is achieved when an independent group can obtain the same results using the authors' own artifacts (i.e., different team, same experimental setup).

This demonstration builds on Anserini and explores the question of how to best integrate it with Solr. We explore and evaluate different architectures, and highlight the new capabilities that Anserini/Solr integration brings.

2 SYSTEM ARCHITECTURE

The first obvious question worth addressing for the academic audience is: Why not just build on top of Solr in the first place? Why is Anserini, for example, built around Lucene instead of Solr? We begin by first articulating the distinction between Lucene and Solr.

2.1 Lucene vs. Solr

Lucene defines itself as a search library. Grant Ingersoll, a Lucene committer as well as the CTO and co-founder of Lucidworks, a company that provides commercial Lucene products and support, offers the analogy that Lucene is like "a kit of parts" [7]. It doesn't prescribe how one would assemble those parts (analysis pipelines, indexer, searcher, etc.) into an application that solves a real-world search problem. Solr fills this void, providing a "canonical assembly" in this analogy.

Solr is a complete end-to-end search platform that uses Lucene for its core indexing and retrieval functionalities. Designed as a web application, Solr is "self-contained" in the sense that all interactions occur via HTTP-based API endpoints. Although there are many client libraries that facilitate access to Solr instances in a variety of programming languages, this design has two main drawbacks from the perspective of IR research:

- Solr APIs were designed with developers of search applications in mind, and thus expose endpoints for indexing, search, administration, and other common operations. However, these APIs lack access to low-level Lucene internals needed by many researchers. While it is in principle possible to expose these functionalities as additional service endpoints for client access, this introduces friction for IR researchers.

SIGIR '19, July 21–25, 2019, Paris, France

ACM ISBN 978-1-4503-6172-9/19/07…$15.00
https://doi.org/10.1145/3331184.3331401

[1] https://www.acm.org/publications/policies/artifact-review-badging

- Since Solr is a web application architecturally, it necessarily runs as a distinct process. Thus, conducting IR experiments involves first starting and configuring the Solr server (i.e., something for the client to connect to). This procedure makes conducting *ad hoc* retrieval experiments unnecessarily complicated.

In other words, the Solr "canonical assembly" was *not* designed with IR researchers in mind. This is where Anserini comes in: it builds directly on Lucene and was specifically designed to simplify the "inner loop" of IR research on document ranking models. The system allows researchers to conduct *ad hoc* experiments on a broad range of test collections right out of the box, with adaptors for standard TREC document collections and topic files as well as integration with standard evaluation tools such as `trec_eval`. A researcher issues one command for indexing and a second command for performing a retrieval run—and is able to replicate results for a range of ranking models, from baseline bag-of-words ranking to competitive approaches that exploit phrase queries as well as (pseudo-)relevance feedback. Anserini provides all these functionalities without the various overheads of Solr, e.g., managing a separate server, latencies associated with network traffic, etc.

2.2 Anserini Shortcomings

While Anserini already supports academic information retrieval research using standard test collections, there are two main missing capabilities.

The existing focus on supporting document ranking experiments means that the project has mostly neglected interactive search interfaces for humans. These are needed, for example, by researchers exploring interactive search and other human-in-the-loop retrieval techniques. Although Anserini has been integrated with other search frontends such as HiCAL [1], such efforts have been ad hoc and opportunistic. One obvious integration path is for Anserini to expose API endpoints for integration with different search interfaces. However, these are exactly the types of APIs that Solr already provides, and so such an approach seems like duplicate engineering effort with no clear-cut benefit.

As another shortcoming, Anserini does not currently support distributed retrieval over large document collections in a partitioned manner, which is the standard architecture for horizontal scale-out. Although previous experiments have shown Anserini's ability to scale to ClueWeb12, the largest IR research collection currently available (733 million webpages, 5.54TB compressed), using a single monolithic index, the observed query latencies are not suitable for interactive searching [6]. Building a distributed search engine is non-trivial, but this is a problem Solr has already solved—with numerous deployments in production demonstrating the robustness of its design. Once again, it makes little sense for Anserini to reinvent the wheel in building distributed search capabilities.

2.3 Anserini/Solr Integration

Given the two shortcomings discussed above, it makes sense to explore how Anserini can be more tightly integrated with Solr. Different possible architectures are shown in Figure 1. On the left, denoted (a), we show the current design of Anserini, where documents are ingested and inverted indexes are created directly using Lucene (and stored on local disk). In the middle, denoted (b), we show an architecture where Anserini is used as a frontend for document processing, but indexing itself is handled by Solr (which uses Lucene behind the scenes to construct the inverted index). In this design, an IR researcher uses the same exact Anserini indexing tool as before, with the exception of specifying configuration data pointing to a Solr instance. In principle, this Solr instance could be residing on the same machine that is running Anserini, as indicated in (b), or it could be on a different machine (not shown). Separating the frontend from the backend incurs network traffic, but allows distributing load across multiple machines.

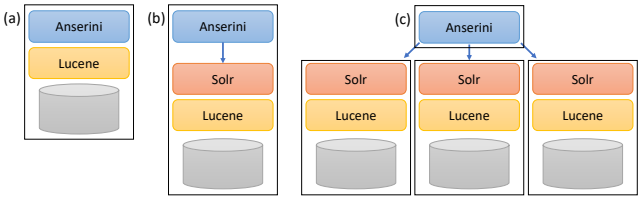

Figure 1: Different architectures for integrating Anserini with Solr. In order from left: (a) the current Anserini design; (b) Anserini indexing into a single-node SolrCloud instance (on the same machine); (c) Anserini indexing into a multi-node SolrCloud cluster.

Introducing this additional layer of indirection allows us to take advantage of Solr's existing capabilities. For example, we get SolrCloud, which is the ability to set up a cluster of Solr servers for distributed retrieval, "for free". This is shown in the rightmost diagram in Figure 1, denoted (c), where Anserini indexes into a SolrCloud cluster. The use of Solr also means that the backend can interoperate with any number of search interfaces and other frontends in the broader ecosystem (see Section 4). The downside, however, is that indexing occurs over an HTTP-based API endpoint, which is obviously less efficient than directly writing index structures to disk. Solr clients perform automatic batching to amortize the connection costs, but the performance penalty of such a setup is an empirical question to be examined.

An alternative approach to integrating Anserini with Solr is to build indexes directly on local disk, and then copy those indexes into an already running Solr instance. This is possible because Solr itself builds on Lucene, and thus all we need to do is to properly synchronize Solr index metadata with the index structures directly built by Anserini. This works even with a SolrCloud cluster: we can build inverted indexes over individual partitions of a collection, and then copy the data structures over to the appropriate node. In such an approach, the Anserini indexing pipeline remains unchanged, but we need a number of auxiliary scripts to mediate between Solr and the pre-built index structures.

3 EXPERIMENTAL EVALUATION

3.1 Setup

Hardware. Our experiments were conducted on the following:

- A "large server" with 2× Intel E5-2699 v4 @ 2.20GHz (22 cores, 44 threads) processors, 1TB RAM, 26×6TB HDDs, running Ubuntu 16.04 with Java 1.8.
- A ten node cluster of "medium servers", where each node has 2× Intel E5-2670 @ 2.60GHz (8 cores, 16 threads) processors, 256GB

Collection	# docs	Large Server		Medium Server			Cluster
		Lucene	Solr (single-node)	Lucene Shard	Lucene	Solr (single-node)	Solr (multi-node)
NYTimes	1.8M	4m14s ± 6s	2m53s ± 11s	3m12s ± 9s	4m17s ± 6s	5m16s ± 50s	3m25s ± 15s
Gov2	25.2M	1h1m ± 3m	1h52m ± 3m	18m6s ± 29s	1h14m ± 1m	2h13m ± 6m	50m30s ± 35s
ClueWeb09b	50.2M	2h40m ± 2m	4h49m ± 2m	44m33s ± 1m	-	-	2h15m ± 9m
ClueWeb12-B13	52.3M	3h9m ± 2m	6h6m ± 9m	46m52s ± 1m	-	-	2h4m ± 4m
Tweets2013	243M	3h44m ± 2m	3h13m ± 10m	2h58m ± 3m	4h53m ± 2m	5h29m ± 4m	3h55m ± 4m

Table 1: Total indexing time (mean ± standard deviation) for various architectures on different collections.

RAM, 6×600GB 10k RPM HDDs, 10GbE networking, running Ubuntu 14.04 with Java 1.8. In the cluster setup, one node is used as the driver while the remaining nine nodes form a SolrCloud cluster. For comparison purposes, we also ran experiments on an individual server.

Note that processors in the medium servers date from 2012 (Sandy Bridge architecture) and the processors in the large server date from 2015 (Broadwell architecture), so there are significant differences in terms of compute power, both compared to each other and compared to current generation hardware.

Document Collections. We use a number of standard IR document collections in our evaluation:

- The New York Times Annotated Corpus, a collection of 1.8 million news article, used in the TREC 2017 Common Core Track.
- Gov2, a web crawl of 25.2 million .gov webpages from early 2004, used in the TREC Terabyte Tracks.
- ClueWeb09b, a web crawl comprising 50.2 million webpages gathered by Carnegie Mellon University in 2009, used in the TREC Web Tracks.
- ClueWeb12-B13, a web crawl comprising 52.3 million webpages gathered by Carnegie Mellon University in 2012 as the successor to ClueWeb09b, also used in the TREC Web Tracks.
- Tweets2013, a collection of 243 million tweets gathered over February and March of 2013, used in the TREC Microblog Tracks [5].

Architectures. We examined a few different architectures, as outlined in Figure 1. In all cases, we built full positional indexes and also store the raw document texts.

- **Lucene.** The default implementation for Anserini, and our baseline, has a single, shared Lucene `IndexWriter` for all threads indexing to disk. We set the thread count to be equal to the number of physical CPU cores and use a write buffer of 2GB. This corresponds to Figure 1(a).
- **Solr.** Anserini is used as a frontend for indexing into a single-node SolrCloud instance, corresponding to Figure 1(b), as well as a nine node SolrCloud cluster, corresponding to Figure 1(c). In the single-node case, the Anserini frontend and the SolrCloud instance both reside on the same server. In the SolrCloud cluster, the Anserini frontend runs on one of the medium servers while the remaining nine servers each host a single Solr shard. Although strictly not necessary in the single-node case, we nevertheless use SolrCloud to simplify implementation. In both cases we use a dedicated `CloudSolrClient` for each indexing thread (with one thread per physical CPU core); batch size is set to 500 for ClueWeb09b and 1000 for the other collections (the smaller batch size for ClueWeb09b is necessary to avoid out-of-memory errors). We set Solr's `ramBufferSizeMB` to 2GB, matching the Lucene condition, and define a schema to map Anserini fields to the appropriate types in Solr. The performance difference between Lucene and the single-node SolrCloud instance characterizes Solr overhead, and performance comparisons between single-node and multi-node SolrCloud quantifies the speedup achievable with distributed indexing.
- **Lucene Shard.** In this configuration, Anserini builds indexes over 1/9th of each collection. This models the scenario where we separately build indexes over document partitions "locally" and then copy each index to the corresponding server in SolrCloud. We use the same settings as the Lucene configuration above. Comparisons between this condition and a multi-node SolrCloud cluster characterizes the overhead of distributed indexing.

3.2 Results

Table 1 shows indexing performance for the various architectures described above. We report means with standard deviations over three trials for each condition. Note that due to the smaller disks on the medium servers, we were not able to index ClueWeb09b and ClueWeb12-B13 under the single-node condition.

These experiments show that Anserini indexing into Solr has substantial performance costs. In our results table, the "Lucene" vs. "Solr (single-node)" columns quantify the overhead of Solr's application framework—the extra layer of indirection that comes with REST APIs. For small collections, the overhead is modest (and for NYTimes on the large server, Solr is actually faster), but the overhead can be quite substantial for the large collections. For example, on ClueWeb12-B13, indexing into Solr takes almost twice as long as directly writing local indexes.

We can compare "Solr (single-node)" vs. "Solr (multi-node)" to understand the performance characteristics of distributed SolrCloud. Information is limited since we only have a cluster of medium servers, and limited drive capacity prevented comparisons on the ClueWeb collections. Nevertheless, it is clear that we do *not* achieve linear speedup: perfect scalability implies that we would be able to index the entire collection in 1/9th of the time on a cluster of nine nodes. However, we are pleased with the ease of multi-cluster setups in SolrCloud, since a distributed architecture would be necessary to support query latencies for interactive search on even larger collections (e.g., the full ClueWeb collections).

From these experiments, we discovered that tuning of various configurations (e.g., thread counts, batch sizes, and buffer sizes) has a large impact on performance. For example, on a single node,

we essentially run into a producer–consumer queuing problem: Anserini threads are "producing" documents that are "consumed" by Solr threads. It is difficult to perfectly balance throughput, and thus one side is often blocking, waiting for the other. In our experiments, we have taken reasonable effort to optimize various parameter settings across all our experimental conditions, but have not specifically tuned parameters for individual collections—and thus it is likely that more fine-grained collection-specific tuning can further increase performance. Nevertheless, we believe that our results reasonably reflect the true capabilities of Lucene and Solr in the various configurations, as opposed to performance that has been hobbled due to poor parameter settings.

Comparing "Lucene Shard" vs. "Lucene" on the medium server, we see that indexing 1/9th of the collection does not take 1/9th of the time. This is an artifact of our current implementation, where we still scan the entire collection (i.e., parsing every document) in order to determine which shard a document belongs in. Thus, we encounter substantial document processing overhead in this naïve implementation of Lucene sharding, i.e., nine passes over the collection, building a shard index in each pass. The overall indexing time could be reduced by running the indexing in parallel on each of the servers hosting the Solr shards. Nevertheless, we believe that with an improved implementation, our alternative integration strategy—building indexes for each shard locally and then copying them to the appropriate SolrCloud server—can be viable.

Finally, we note that all our experiments were conducted on magnetic spinning disks. SSDs have very different characteristics, and it would be desirable to replicate these experiments with more modern server configurations.

4 INTERACTIVE SEARCH

An important capability enabled by Anserini/Solr integration is entrée into the rich ecosystem of Solr frontends. In particular, this allows IR researchers to leverage efforts that have been invested in creating Solr-based search interfaces. This would specifically benefit researchers working on interactive IR, who often have the need to create custom search interfaces, to, for example, support user studies. Users have come to expect much from such interfaces, and instead of trying to implement these features from scratch, researchers can reuse existing components.

As a demonstration of Solr's capabilities, we have adapted Blacklight[2] as a search interface to Anserini. Blacklight is an open-source Ruby on Rails engine that provides a discovery interface for Solr. It offers a wealth of features, including faceted search and browsing, keyword highlighting, and stable document URLs. The entire system can be customized via standard Rails templating mechanisms. Blacklight has a vibrant developer community and has gained broad adoption in the library and archives space, being deployed at dozens of university libraries and cultural heritage institutions.

Figure 2 shows a screenshot from our custom Blacklight instance, dubbed "Gooselight", searching over the collection of 243 million tweets from the TREC 2013 Microblog Track [5] indexed via our Anserini/Solr integration. Here, we highlight the flexibility of Blacklight by rendering results using Twitter's official API, which shows

[2]http://projectblacklight.org

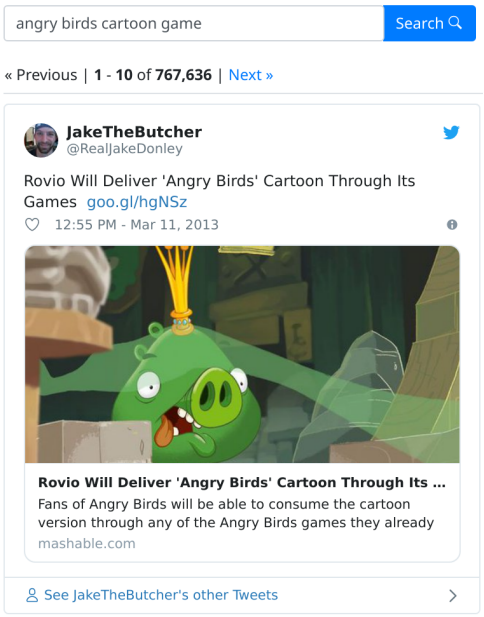

Figure 2: Screenshot of Gooselight, a search interface using Blacklight that connects directly to Anserini/Solr.

media previews and threaded discussions (if available), and provides direct links to the original source and other Twitter "actions". Use of this rendering API also allows our search interface to respect Twitter's terms of service regarding private and deleted tweets.

5 CONCLUSIONS

With Anserini/Solr integration, we argue that it is possible to "have your cake and eat it too". Anserini continues to support the tight "inner loop" of IR research (i.e., model refinement), but now additionally offers the broader capabilities described here.

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] M. Abualsaud, N. Ghelani, H. Zhang, M. Smucker, G. Cormack, and M. Grossman. 2018. A System for Efficient High-Recall Retrieval. In *SIGIR*. 1317–1320.
[2] L. Azzopardi, M. Crane, H. Fang, G. Ingersoll, J. Lin, Y. Moshfeghi, H. Scells, P. Yang, and G. Zuccon. 2017. The Lucene for Information Access and Retrieval Research (LIARR) Workshop at SIGIR 2017. In *SIGIR*. 1429–1430.
[3] L. Azzopardi, Y. Moshfeghi, M. Halvey, R. Alkhawaldeh, K. Balog, E. Di Buccio, D. Ceccarelli, J. Fernández-Luna, C. Hull, J. Mannix, and S. Palchowdhury. 2017. Lucene4IR: Developing Information Retrieval Evaluation Resources Using Lucene. *SIGIR Forum* 50, 2 (2017), 58–75.
[4] J. Lin, M. Crane, A. Trotman, J. Callan, I. Chattopadhyaya, J. Foley, G. Ingersoll, C. Macdonald, and S. Vigna. 2016. Toward Reproducible Baselines: The Open-Source IR Reproducibility Challenge. In *ECIR*. 408–420.
[5] J. Lin and M. Efron. 2013. Overview of the TREC-2013 Microblog Track. In *TREC*.
[6] P. Yang, H. Fang, and J. Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *SIGIR*. 1253–1256.
[7] P. Yang, H. Fang, and J. Lin. 2018. Anserini: Reproducible Ranking Baselines Using Lucene. *JDIQ* 10, 4 (2018), Article 16.
[8] W. Yang, K. Lu, P. Yang, and J. Lin. 2019. Critically Examining the "Neural Hype": Weak Baselines and the Additivity of Effectiveness Gains from Neural Ranking Models. In *SIGIR*.