dstlr: Scalable Knowledge Graph Construction from Text Collections

by

Ryan Clancy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In recent years, the amount of data being generated for consumption by enterprises has increased exponentially. Enterprises typically work with structured data, but oftentimes the data being generated is semi-structured or unstructured in nature. In particular, there exists a wealth of unstructured text data (customer reviews, social media posts, news articles, etc.) containing information that could provide value to an organization. As data from different sources often reside in silos, a number of questions arise: How do we integrate the structured and unstructured data? How can we curate and refine the data? Can we do this at scale?

In this thesis, I present DSTLR – a platform for scalable knowledge graph construction from text collections. I show how assertions extracted from a collection of unstructured text documents can be used to form a knowledge graph, enabling integration of structured and unstructured data. Further, I show that linking to an existing knowledge graph enables rule-based data curation using the additional external information. I demonstrate this on a large collection of news articles, highlighting the horizontal scale-out of the system.

Acknowledgements

I would like to thank my supervisor, Professor Jimmy Lin, for his constant guidance throughout this journey for which I am very grateful.

Thank you to my thesis readers, Professor Ihab Ilyas and Professor Tamer Özsu, for taking the time to review my work.

Finally, I would like to thank all of the people I have collaborated with and the friends that I have made during my time at the University of Waterloo.

Dedication

This is dedicated to my family and loved ones for their constant support throughout my education.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The proliferation of the web in the past two decades has enabled the generation of large volumes of data by its users. With half of the Earth's population using the internet, it is estimated that 2.5 quintillion bytes (2.1 exabytes) of data is generated each day.[1] Social media networks and news outlets account for a large part of this content, for example, with around 473,000 tweets and 510,000 Facebook comments posted *every minute*.[2]

With all of this data available, it makes sense to explore how an enterprise could take advantage to provide value for the business. Enterprises typically work with structured data residing in relational databases. For example, a hotel chain may maintain records for customer transactions containing location, number of nights stayed, amount spent, and more. However, there exists large volumes of data that is "stranded" in unstructured text (e.g., in customer reviews on Google, TripAdvisor, etc.) that provide additional insights that company generated structured data cannot provide. With access to this unstructured knowledge, capabilities such as automatically identifying which competitors customers write about in negative reviews could be possible.

Fortunately, there are a number of natural language processing tools that facilitate the extraction of structured data from text. This data is typically represented as a knowledge graph, a structured collection of linked data. However, the creation of knowledge graphs, such as Wikidata, is not without problems. Knowledge graphs are generally handcrafted through community or crowd-sourcing efforts and are expensive (in terms of time, money, or both) to create. Further, maintaining provenance back to the source of information is

[1] https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read

[2] https://www.domo.com/learn/data-never-sleeps-6

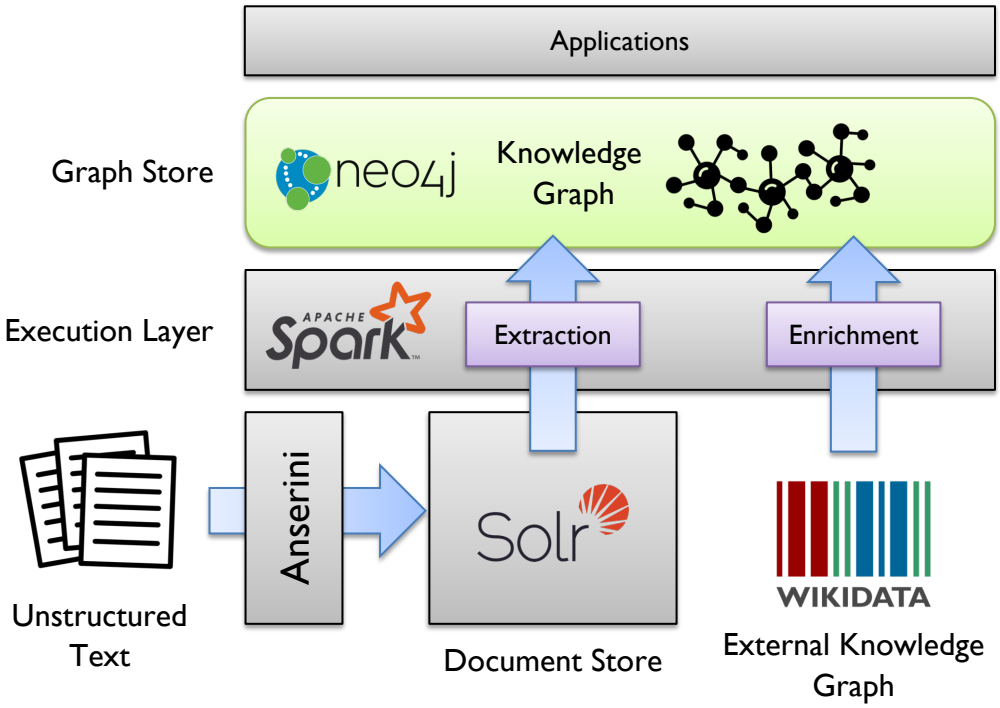

Figure 1.1: The DSTLR architecture

crucial for verification of information and is generally lost in the knowledge graph construction process. Although plenty of unstructured text for generating knowledge graphs is available, extracting structured knowledge from the text is non-trivial and computationally expensive. Additional problems such as the quality of extracted information and interacting with unstructured text from systems that typically work with structured data exist as well.

This work introduces a platform to address these problems through the automation of knowledge graph construction. The architecture for the platform, called DSTLR, is presented in Figure 1.1. DSTLR takes a collection of unstructured text documents and uses Anserini, our academic information retrieval (IR) toolkit, to index documents into the distributed Solr search engine for scalable indexing, searching, and as a data source for downstream processes [2, 10, 11]. The computations required to construct a knowledge graph from the documents are horizontally scaled out and an inference layer is exposed for data curation via graph queries enabled by storing information extracted from documents (called assertions) in a graph store. Provenance is maintained back to the specific spans of text in the source documents that information was extracted from to provide attribution.

Mentions of entities in assertions are linked to an external knowledge graph and ground-truth values for assertions (called facts) are stored in the graph store as well. The produced knowledge graph supports the development of applications to support a number of use cases such as finding supporting and inconsistent information between the text and existing data sources, providing missing information to existing data sources, generating distantly supervised training data for relational extractors to learn from, and more.

More concretely, given an input document collection D and a pre-existing, external knowledge graph G, the task is to produce a knowledge graph K comprised of (SOURCE, LINK, TARGET) tuples extracted from text documents in D. Stanford's CoreNLP toolkit [7] is scaled out via the Apache Spark distributed analytics engine [12] to perform information extraction on documents indexed into SolrCloud, a distributed Solr cluster, while maintaining provenance back to the source document. Mentions of entities (i.e., SOURCE nodes) were linked to Wikidata entities (the choice of external knowledge graph G) and the fixed LINK types from CoreNLP were manually aligned with Wikidata properties such that access was gained to existing information in the external knowledge graph, integrating the unstructured text with existing structured data. The assertions from the documents were augmented with facts from linked Wikidata entities and were loaded into the Neo4j graph database, enabling exploration and manipulation via Neo4j's native Cypher query language. This, in turn, allows the formulation of Cypher queries for data curation supporting the use cases described above. The platform is demonstrated on nearly 600K Washington Post news articles, a proxy due to the lack of enterprise data access for this thesis, and example applications are provided that demonstrate the use cases described above.

1.1 Contributions

The contribution of this thesis is the DSTLR architecture which is composed of:

- The integration of the Solr search engine and document store in Anserini to enable scalable, distributed document indexing, retrieval, and as a data source for downstream analytics
- The design of a horizontally scalable architecture integrating Solr, Spark, CoreNLP, Apache Jena, and Neo4j for knowledge graph construction from text documents, including linking entities to an external, pre-existing knowledge graph and augmenting extracted relations with external information

- A rule-based data curation approach using Cypher graph database queries enabling a number of applications such as: the discovery of supporting and inconsistent information between documents and the external knowledge graph (evidence for and against an assertion), missing information in the knowledge graph that is present in a document, and the generation of distantly supervised training data which could be used to further train relational extractors

1.2 Thesis Organization

The thesis is organized as follows:

- Chapter 2 discusses indexing text documents into Solr via Anserini, the benefits this provides, and a performance evaluation against Lucene. This corresponds to the bottom-left section of Figure 1.1.
- Chapter 3 describes the architecture and implementation of the system for knowledge graph construction, from querying text documents to loading data into the graph store. This corresponds to the middle and bottom-right sections of Figure 1.1.
- Chapter 4 highlights some data curation applications enabled by the platform. This corresponds to the top section of Figure 1.1.

Chapter 2

Indexing Documents via Solr

Large volumes of unstructured text is available from which a knowledge graph can be constructed. These text documents could be collected and stored as text files in a file system, however this does not facilitate easy access from network enabled applications which may not reside on the same machine. Having documents available in a document store, on the other hand, makes it less cumbersome to write applications which need access to the documents. Solr is a distributed search engine which fills this need by providing APIs and user interfaces for access to documents. Solr leverages an inverted index to efficiently select document subsets of interest which is much faster than a brute-force scan and filter approach [3] when building a knowledge graph over subsets of documents. Using Solr as a document store provides a data source for downstream processes which construct the knowledge graph.

This chapter covers the integration of Solr, a Lucene-based distributed search engine, into our Anserini information retrieval toolkit. This is accompanied by a performance analysis comparing indexing with Solr to indexing with Lucene directly, the default implementation provided by Anserini.

2.1 Background

2.1.1 Lucene

Lucene is an open-source, high-performance text indexing and search library for the JVM maintained by the Apache Software Foundation.[1] Lucene provides a number of components (e.g., tokenizers, language-specific analyzers, indexers, searchers, etc.) that can be used in conjunction to construct a document retrieval pipeline within an application. It is important to note that Lucene itself is not a search engine, but merely a search *library* that provides components that could be used to create a search engine. Lucene is commonly used in industry to support both index- and search-heavy workloads. For example, it is used as the basis for serving billions of queries per day for tweets at Twitter.[2] Although Lucene itself can facilitate indexing and retrieval, it lacks the built-in machinery to conduct IR experiments.

2.1.2 Solr

Solr is an open-source, high-performance search platform based on Lucene also maintained by the Apache Software Foundation.[3] Although Lucene provides the individual components to support document retrieval, it itself is not a search engine but instead a search library containing a collection of components which *could* be used for the index and retrieval functionality required to build a search engine. Solr aims to operationalize Lucene into a fully-fledged search platform supporting easy document indexing and retrieval by various applications without needing to know the inner workings of retrieval as required with Lucene. Unlike Lucene, Solr provides: single-node and multi-node operational modes, distributed and fault-tolerant document indexing and retrieval capabilities, powerful management interfaces, and an easy-to-use REST API (which lowers the barrier of entry to Lucene while also exposing low-level Lucene internals for more advanced users).

2.1.3 Anserini

Anserini is an open-source information retrieval toolkit based on Lucene that enables researchers to perform replicable information retrieval experiments using technologies battle

[1] https://lucene.apache.org/
[2] https://lucidworks.com/post/how-twitter-uses-apache-lucene-for-real-time-search/
[3] https://lucene.apache.org/solr/

tested in industry [2, 10, 11]. Lucene itself is able to index arbitrary text but lacks the ability to natively index document collections used in IR research which are often in formats such as WARC, JSON, or HTML. Further, Lucene is not able to evaluate and compute metrics (e.g., MAP, P@30, etc.) for retrieval results using tools such as TREC_EVAL which are important for academic information retrieval.[4] Anserini aims to streamline this process for researchers by enabling an integrated environment to facilitate IR research. Anserini provides high-performance, multi-threaded document parsing and indexing using Lucene (scaling up to collections containing millions of documents), retrieval, and evaluation using TREC_EVAL.

2.2 Solr Integration in Anserini

Anserini is currently built around Lucene and while this proves to be a great base for IR research, there are two main drawbacks motivating the integration of Solr into Anserini.

First, Lucene indexes are monolithic in nature. When documents are indexed they are written to disk on a single machine. When document collections are large, this bottleneck means the indexing process takes a long time while also increasing search latencies. For example, indexing ClueWeb12 produces an index that is up to 12TB large.[5] Due to the large index size, operating on such a document collection requires a machine with:

1. many CPU cores for indexing (a process which takes multiple days)
2. large volumes of SSD disk space to hold the index and improve search latencies

Second, in using Lucene directly there is a lack of search interfaces available in the ecosystem to support IR research and interactive exploration of documents. With a Lucene index on disk, exploring documents is not very user friendly and requires command line tools or a Java program for searching through documents. Users of Anserini, who may not be information retrieval researchers, might not have the skills required to write such a program or use a more low-level index exploration tool such as luke.[6]

Motivated by these drawbacks, it makes sense to explore how Solr can be integrated into Anserini.

[4] https://trec.nist.gov/trec_eval/
[5] https://lemurproject.org/clueweb12/
[6] https://github.com/DmitryKey/luke

2.2.1 Integration

In terms of integration effort, as both Solr and Anserini are based on Lucene, the integration is fairly seamless and the existing Anserini machinery for document parsing and evaluation could be used. However, new indexing and retrieval components needed to be integrated into Anserini to support Solr. This work focuses on indexing and searching using the Solr REST APIs for interactive retrieval rather than retrieval for IR experiments (which is only briefly discussed as I did not implement such a feature). Whether a single-node or multi-node Solr is used, Solr runs in "cloud" mode which means that a ZooKeeper instance maintains state for each Solr node and holds the configuration for each Solr instance. Solr running in this mode is called SolrCloud.

In order to use Solr instead of Lucene as a document store, some Anserini specific configuration options for Solr with ZooKeeper to align metadata (e.g., field types) and a single new class for indexing was required.

First, Anserini specific configuration options needed to be provided to ZooKeeper such that when indexing a new collection in Solr, the schema that Anserini expects is present. In Solr, a schema defines the names of fields, their type (string, text, numeric, etc.), the analysis pipelines on a per-field basis, which similarity function to use for retrieval, and more. A "collection" in Solr consists of a name along with a schema where documents indexed to this collection should conform to the specified schema. At creation time, collections have other parameters such as the number of shards to split the index into and how many replicas of each shard to maintain across the cluster. In general, Anserini expects an `id` field (which is provided by default in the schema) along with either a `raw` or `contents` field depending on the indexing mode in Anserini. This differs for Twitter and WashingtonPost document collections as they have fields specific to their format. When `raw` documents are indexed, the document body is placed in the `raw` field without any additional parsing. For example, this could be the HTML source code of a webpage. On the other hand, indexing `transformed` documents does additional parsing of the content before indexing and is stored in the `contents` field. For example, this would be the HTML body text from a webpage. Solr uses BM25 similarity for documents by default, which is the default for Lucene, however the BM25 parameters needed to be adjusted to align with values used in Anserini.

Second, a new indexer was implemented which is able to take advantage of the existing document parsing capabilities of Anserini such that integration involved interfacing with the Solr REST APIs. The new indexer batches documents to offset the overhead of HTTP calls (see Section 2.3) and calls Solr REST APIs to index documents into a pre-created collection using the Anserini schema. Of course, Solr acts as a proxy and indexes documents

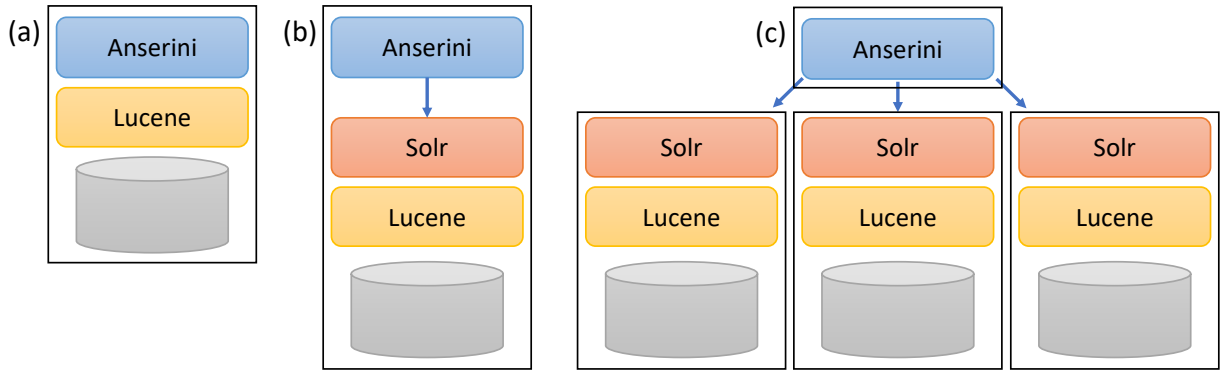

Figure 2.1: Architectures for Solr integration in Anserini. Left to Right: (a) the current Anserini design using Lucene; (b) Anserini indexing into a single-node SolrCloud instance (on the same machine); (c) Anserini indexing into a multi-node SolrCloud cluster.

to disk using Lucene.

In terms of retrieval, a searcher that interfaces with Solr REST APIs for document querying and a translation layer for creating Lucene document objects from Solr document objects was required. This translation layer allows the normal Anserini index–retrieval–evaluation pipeline to be run without changes. This, however, deals with information retrieval experiments rather than more traditional document search and as such is not covered in detail. Searching without the IR machinery for evaluation, however, is supported by the Solr REST APIs.

2.2.2 Multi-node Indexing and Retrieval

Solr provides operational modes for single-node and multi-node clusters. When running Solr, there are two modes: standalone mode or cloud mode. If a single-node Solr setup is desired, either standalone mode or cloud mode can be used. A multi-node setup, however, requires cloud mode.

In standalone mode, Solr runs exactly as you would expect as a self-contained process without any external dependencies. In cloud mode, Solr uses ZooKeeper for management of cluster state and for centralized configuration management. ZooKeeper can be run internally within one of the nodes or as an external process to facilitate high-availability with a cluster of ZooKeeper nodes. Different architectures of Anserini in conjunction with Lucene or Solr are shown in Figure 2.1. When using Solr, it is flexible where the Solr server

runs in relation to Anserini. Solr can run on the same machine or a different machine for standalone mode and on a cluster of machines for cloud mode.

For integration with Anserini, it is assumed that Solr is running in cloud mode as SolrCloud. Cloud mode provides a slightly richer management interface along with the ability to add nodes to the cluster in the future. It also requires a different SolrClient in the Java implementation for interfacing with the REST API for standalone vs. cloud mode Solr. Thus, for code simplicity, cloud mode was deemed a better option as it supports both single-node and multi-node setups.

The benefit of using cloud mode is that whether Solr is running a single-node cluster or a multi-node cluster with 100 nodes, it is no different for Anserini. Indexing and retrieval transparently works with either scenario. This addresses the first drawback of using Lucene, namely the performance/cost limitations of a single machine for indexing and retrieval.

2.2.3 User Interfaces and Management Capabilities

Within the Solr ecosystem, and built into Solr itself, there are a number of tools that facilitate interaction with documents. With a Lucene index sitting on disk there is a limited number of low-level tools available (which are targeted at technical users) in order to explore the index. Alternatively, the index can be interacted with via the Lucene API using Java. However, this is not ideal for all users who may not be familiar with the inner workings of Lucene (or retrieval in general) and simply want to index and search for documents. Luckily, there are numerous search interfaces available for use within the Solr ecosystem which are built around the more user friendly REST API that Solr provides.

Solr exposes an administration interface, shown in Figure 2.2, which affords users access to a number of tools for interacting with documents. Interfaces for browsing and querying documents are available with different parameters exposed for searching documents. Views that expose the schema of a given collection along with the field data types are also available, which is useful in cases where schema inference was used. When using Solr in distributed mode, cluster health and management interfaces are also available along with the ability to create collections (with options such as number of shards, number of replicas, etc.).

In terms of customization for building your own search interface, to support human-in-the-loop experiments for example, the open-source Ruby on Rails engine Project Blacklight can be used by more technical users.[7] Customization is fairly straightforward, even for non

[7]https://projectblacklight.org/

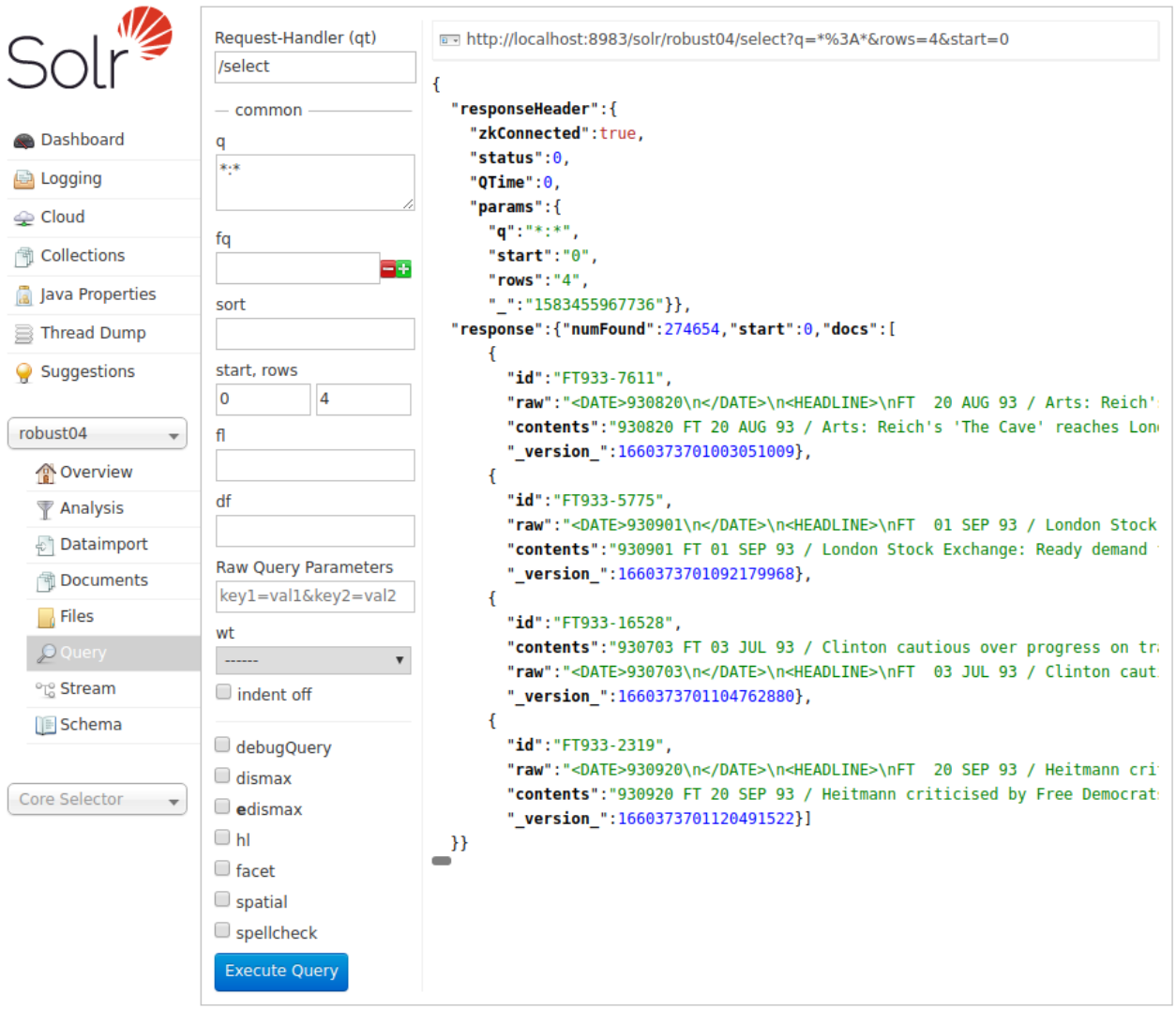

Figure 2.2: The Solr admin interface

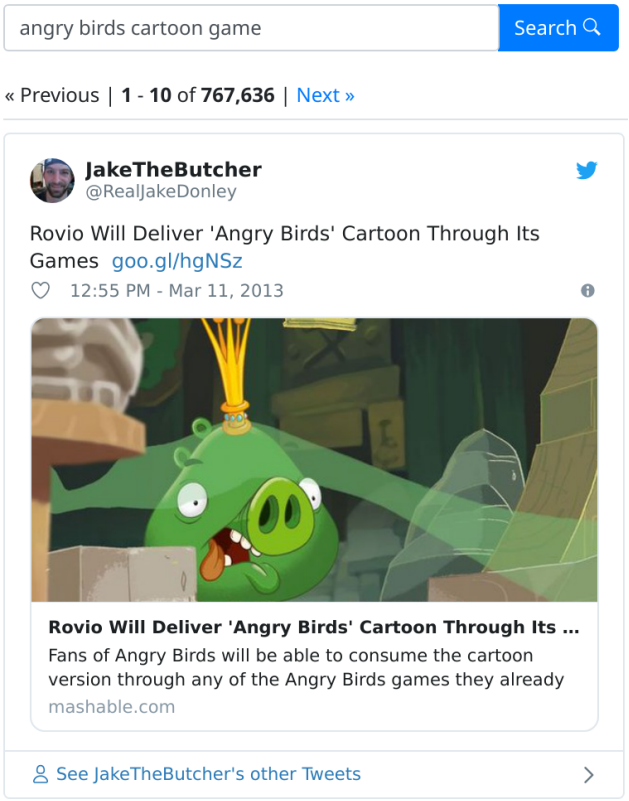

Figure 2.3: Gooselight search interface rendering a tweet from Tweets2013.

Ruby developers, to facilitate the easy creation of custom search interfaces working off of their base. Project Blacklight is often used in the library and archival space with wide adoption by academic institutions all over the world. Figure 2.3 shows the Gooselight search interface, built via Project Blacklight, rendering a rich tweet from the Tweets2013 corpus using the official Twitter API.

This addresses the second concern with using Lucene directly, namely the lack of search interfaces available and the higher barrier of entry.

2.3 Performance Evaluation

In order to compare the overhead of indexing with Solr relative to the Lucene baseline, the indexing throughput was tested under a number of different scenarios. Although bulk indexing is usually a one-time process, at least in this use case, investigating what sort of overhead the communication with Solr incurs is important to understand. Having a

small overhead for the HTTP communication would be acceptable while larger increases in end-to-end indexing time may not be.

The default indexing back-end for Anserini is Lucene. Obviously, writing an index to disk directly using Lucene should be more efficient than indexing to a local single-node Solr. Although Solr uses Lucene under the hood, it incurs the extra overhead of HTTP requests. However, when a multi-node Solr cluster is used, the intuition is that the indexing throughput should improve, despite the additional remote HTTP request overhead, as a number of Solr servers are indexing in parallel.

In order to test the performance of Solr against Lucene, and quantify the benefits a multi-node Solr cluster provides, 5 commonly used collections in the information retrieval space were indexed, namely:

- The New York Times Annotated Corpus, a collection of 1.8 million news article[8]
- GOV2, a web crawl of 25.2 million .gov web pages from early 2004[9]
- ClueWeb09b, a web crawl comprising 50.2 million webpages gathered by Carnegie Mellon University in 2009[10]
- ClueWeb12-B13, a web crawl comprising 52.3 million webpagesgathered by Carnegie Mellon University in 2012[11]
- Tweets2013, a collection of 243 million tweets gathered over February and March of 2013[12]

In terms of hardware, two different servers were used. The "large server" has 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. The "medium server" has 2× Intel E5-2670 @ 2.60GHz (8 cores, 16 threads) processors, 256GB RAM, 6×600GB 10k RPM HDDs, 10GbE networking, running Ubuntu 14.04 with Java 1.8.

A number of different scenarios were run when benchmarking Solr vs. Lucene performance. First, Lucene and single-node Solr indexing tests were run on both the large and medium servers. Second, the cluster of 10 medium servers was used to test the multi-node performance of a distributed Solr cluster. One node was used as the experimental driver

[8] https://catalog.ldc.upenn.edu/LDC2008T19
[9] http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm
[10] https://lemurproject.org/clueweb09.php/
[11] https://lemurproject.org/clueweb12/
[12] https://github.com/castorini/Tweets2013-IA

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 2.1: Total indexing time (mean ± std. dev.) for Lucene vs. Solr

(i.e., it ran Anserini) while the remaining 9 nodes each hosted a Solr shard. Finally, one additional scenario was run where 1/9th of the document collection was indexed on a medium server modeling the scenario where distinct partitions of the documents are indexed separately and could be used within the same collection in the Solr cluster. Comparing different combinations of scenarios on the same collection allowed us to see the performance gain or overhead of Solr relative to Lucene. For all tests, the buffer size for documents before flushing to disk (`ramBufferSizeMB`) for both Lucene and Solr was set to 2GB. The results of the performance comparison are presented in Table 2.1. Note that due to the limited disk space on the medium servers, the full ClueWeb collections were unable to be indexed under single-node scenarios.

When comparing Lucene to single-node Solr performance, where Solr is a direct replacement for Lucene on the same machine, we see that, in general, there is significant overhead. Solr pays a penalty of additional overhead for the communication via its HTTP REST API and for some of the larger collections is almost twice as slow as indexing to disk directly with Lucene.

When comparing Lucene to multi-node Solr performance on the medium servers, where Solr is distributed across a number of machines, we see that indexing time is strictly less than writing locally to disk with Lucene. The distributed indexing enabled by the Solr cluster outweighs the overhead from external HTTP calls. However, a linear speed-up is not achieved; in this case a 9× improvement would be expected in end-to-end runtime rather than the 30% improvement that was observed. This is largely due to the fact that document parsing within Anserini serves as a bottleneck in this case. Each document in the collection needs to be read from disk and parsed (e.g., from XML, HTML, etc.) on a single machine before being batch indexed into Solr.

Indexing 1/9th of the collection via Lucene does not achieve 1/9th of the end-to-end runtime of indexing the full collection. This is due to the current implementation within Anserini that requires reading of the entire document collection; documents that are in the current shard being generated are indexed while those that will not be indexed are

still read from disk and parsed as hashing occurs over document ids. In order to speed up the end-to-end runtime, the shards could be generated in parallel on each of the medium servers over distinct partitions of the documents for the sharded indexing method to be viable.

For all experiments, reasonable efforts were made to tune parameters (thread counts, buffer sizes, etc.) as a producer–consumer balance between Anserini threads and Solr indexing threads needed to be discovered. This configuration represents a reasonable configuration rather than leaving default parameters which would impact performance.

Chapter 3

Large-Scale Knowledge Graph Construction

Enterprises have long worked with structured data, which can be easily manipulated via traditional relational databases, however text data is unstructured in nature and requires different techniques to work with. For example, how do we extract structure from text in order to build applications that are able to take advantage of it? One such approach is to use natural language processing tools to perform information extraction, extracting the "facts" asserted in a document, which can be stored in a graph database due to its (SOURCE, LINK, TARGET) structure. This, in turn, enables applications to work with this data and take advantage of the wealth of text available on the internet and allows for integration with existing data sources.

This chapter presents the architecture, implementation, and evaluation of the knowledge graph construction portion of the platform from querying for text documents through to loading the data into a graph database for subsequent exploration.

3.1 Background

In order to construct a knowledge graph of relations present in unstructured text at scale, a number of techniques and tools from different domains (namely NLP, distributed computing, and databases) need to be used.

3.1.1 Natural Language Processing

In order to construct a knowledge graph from text documents, natural language processing (NLP) tools need to be used to extract structured information from the text. Information extraction, also called relation extraction, is the technique used for this, although there are a number of upstream NLP tasks that information extraction depend on.

Named Entity Recognition

Named entity recognition (NER) is the task of identifying mentions of entities (people, organizations, dates, etc.) within a given sentence. NER is an important pre-requisite for a number of downstream tasks, such as relation extraction, as relations naturally occur between a source and target entity. Figure 3.1 highlights mentions of entities within the sentence "Eric Clapton was born on March 30th, 1945 in England".[1]

Figure 3.1: Named Entity Recognition visualization.

Coreference Resolution

Coreference resolution is the task of identifying references to the same entity mention within a sentence. When a noun such as "John" is used in a sentence, it is often referred to later using pronouns such as "he" or "him". Downstream NLP tasks, such as relation extraction, can take this into account so that relations discovered using pronouns are resolved to their canonical mention. Figure 3.2 shows an example of coreference resolution.[2]

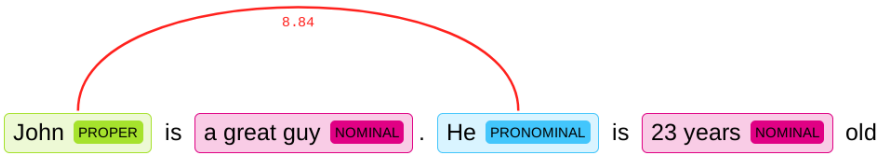

Figure 3.2: Coreference resolution visualization.

[1] https://explosion.ai/demos/displacy-ent
[2] https://huggingface.co/coref/

Entity Linking

Entity Linking is the task of resolving a mention of an entity in a sentence to a distinct real world referent. Intuitively, we know that "President Obama" and "Barack Obama" refer to the same person (entity). As spans of text such as these are often used to refer to the same entity, they need to be linked to the same entity in a knowledge graph. For example, in Wikidata Barack Obama has the entity id of Q76 which "President Obama" and "Barack Obama" should both be linked to. Figure 3.3 shows an example of entity linking.[3]

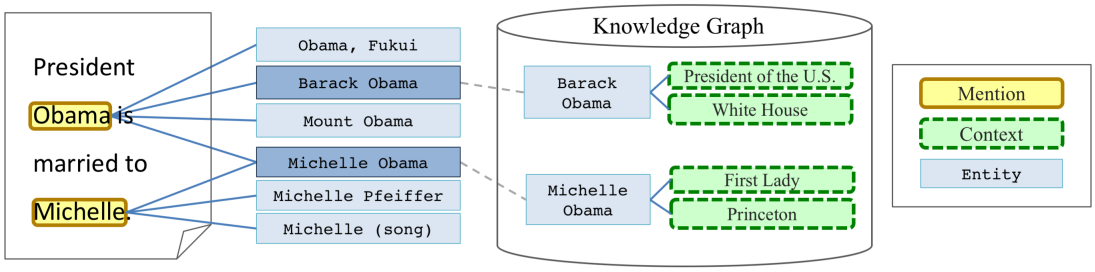

Figure 3.3: Entity linking visualization.

Knowledge Graphs

Knowledge graphs are a structured collection of information about any number of topics. For example, a knowledge graph may contain information ranging from the population of a country to the name of a person's spouse. This information is typically represented as <SUBJECT, PREDICATE, OBJECT> RDF triples, a W3C standard.[4] For example, a triple such as <BARACK OBAMA, DATE_OF_BIRTH, 1961-08-04> would represent Barack Obama being born on August 4th, 1961.

A number of large knowledge graphs are publicly available such as Wikidata and DBpedia. In this thesis Wikidata is used as an external data source, although any knowledge graph could be used in principle. Wikidata contains structured information on over 56 million "items" which makes it ideal for our use case due to the volume of data available. Figure 3.4 contains the Wikidata data model. Within the data model, there are a few important fields of note:

[3]https://qr.ae/TWGvfk
[4]https://www.w3.org/TR/rdf-concepts/

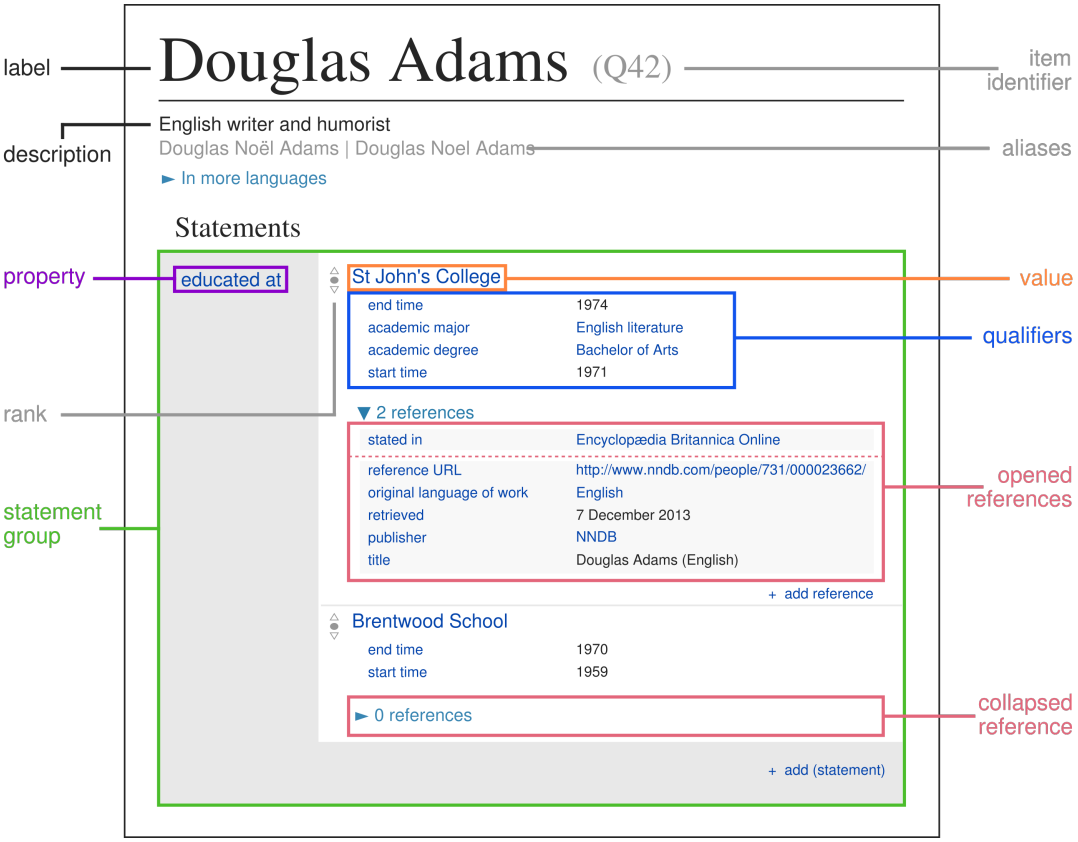

Figure 3.4: Wikidata data model

- Label - the name of the entity
- Item Identifier - the entity's unique ID
- Property - the key for a specific piece of information for an entity
- Value - the value for a piece of information for a given key (property)

Information Extraction

Information Extraction (IE) is the task of extracting relations between mentions of entities within a given sentence. Information Extraction techniques take a span of text as input and produces a list of <SUBJECT, PREDICATE, OBJECT> triples that are present in the sentence.

Method	Output
OpenIE	<Barack Obama, was born on, August 4th 1961>
KBP	<Barack Obama, DATE_OF_BIRTH, 1961-08-04>

Table 3.1: Information Extraction output

Information Extraction techniques fall into two categories: Open Information Extraction and Knowledge Base Population. Open Information Extraction (OpenIE) techniques extract open-class relations based on text present in the sentence using the syntactic dependencies (i.e., nouns linked by verbs or prepositions) between them [1]. Knowledge Base Population (KBP) extracts relations that correspond to a fixed vocabulary via slot-filling [9]. This approach generally normalizes relation names, whereas OpenIE approaches do not. As the task at hand requires matching extracted relations with those present in a knowledge graph, Knowledge Base Population is used over OpenIE. Approaches outside the scope of this thesis, such as clustering relation names produced by OpenIE to an ontology from a given knowledge graph, would also work. Table 3.1 shows an example of OpenIE and KBP results on the sentence "Barack Obama was born on August 4th, 1961".

3.1.2 Technologies

Stanford CoreNLP

The information extraction task described above, along with its prerequisite NLP tasks, need to be grounded in an implementation and wrapped into a library. Stanford CoreNLP is a popular NLP toolkit written in Java and distributed for use within JVM based languages (e.g., Java, Scala, Kotlin, etc.) [7]. CoreNLP is widely used in production in industry and offers good performance while also providing high-quality neural, statistical, and rule-based models. Through a number of annotators (e.g., tokenization, named entity recognition, coreference resolution, etc.) which are chained together into a pipeline, it provides support for most common natural language analysis tasks.

Apache Spark

In the past decade or so, distributed computing tools have been widely used in order to meet the computing requirements demanded by large datasets and the increasingly more demanding computing needs of applications. Traditional systems such as MapReduce [4] have now given way to newer, more specialized and easy-to-use systems such as Spark [12].

Apache Spark is a "unified analytics engine for large-scale data processing" that is commonly used in industry and academia.[5] Spark uses a lineage-based distributed Resilient Distributed Dataset (RDD) as a primitive for performing transformations in parallel on data across a cluster of machines. Spark transformations have roots in functional programming, exposing functions such as MAP and FILTER, and are executed lazily such that transformations are not computed until an action, like COUNT or REDUCE, is performed.

Apache Jena

RDF triples are usually stored in specialized databases, called RDF stores, rather than traditional relational databases or graph databases (which are closely related). RDF stores store <SUBJECT, PREDICATE, OBJECT> triples which are retrieved via the SPARQL semantic query language.

Apache Jena + Fuseki (Jena) is a popular platform for working with RDF data on the JVM.[6] Jena consists of a high performance storage engine, called TDB, for persisting data to disk and a SPARQL endpoint, called Fuseki, that enables querying stored triples. Previous work has shown that, for our use case, Jena is up to an order of magnitude faster than alternatives such as Blazegraph.[7] Jena is used to host a dump of Wikidata which can be queried via the SPARQL endpoint.

Neo4j

Graph databases are specialized databases which store data as nodes in a graph with semantic relationships to other nodes. For example, imagine a social network being stored in a graph database with FRIEND edges between PERSON nodes.

Neo4j is a widely used graph database, following the property graph model, which features the intuitive Cypher query language and a rich query visualization/exploration interface.[8] DSTLR uses Neo4j to store relations, along with auxiliary information such as entity type, which have been extracted from the source corpus.

[5] https://spark.apache.org/
[6] https://jena.apache.org/
[7] https://github.com/r-clancy/dstlr/issues/10#issuecomment-506975110
[8] https://neo4j.com/

3.2 Data

The TREC Washington Post corpus is used as the source corpus from which the knowledge graph is constructed.[9] The corpus contains 595,037 news articles and blog posts published between January 2012 to August 2017.

3.3 Architecture

The DSTLR architecture is comprised of three main components: the data layer, the execution layer, and the graph store. The platform was designed in a way such there are several intermediate persistence points in order to improve modularity. The persistence points are as follows:

- documents stored in the document store
- assertions extracted from documents
- facts from Wikidata
- assertions + facts loaded into graph store

With these persistence points, swapping one component for another is easy and the entire pipeline does not need to be re-run. For example, changing the graph store to a different database means that the processes run previous to loading assertions and facts into the graph database (i.e., indexing documents into Solr and then performing information extraction + enrichment) do not need to be re-run.

The architecture for DSTLR is shown in Figure 1.1. The data layer consists of: the raw documents in their native format, the Anserini toolkit for indexing documents, the document store, and the external knowledge graph. In this thesis, the documents come from the WashingtonPost corpus, Solr was used as the document store, and Wikidata was used as the external knowledge graph. Anserini reads raw documents from disk and indexes them into a distributed SolrCloud cluster to support efficient document retrieval in down-stream processes. The external knowledge graph is available for consumption by the enrichment process.

The execution layer consists of the Spark distributed analytics engine which the core logic is implemented within. This enables transparent horizontal scalability, if required,

[9] https://trec.nist.gov/data/wapost/

for the three Spark jobs form the base of our platform: extracting, enriching, and loading tuples. For efficient execution, extraction should be parallelized with as many processes as the available resources allow while enrichment and loading should have very few workers in order to balance query latency and insertion rate, respectively.

The graph store contains the final knowledge graph which is available for consumption by applications. Neo4j was chosen as a graph store as it is open-source and widely used.

The application layer consists of applications which can consume the knowledge graph. Examples of such applications are provided in Section 4.

3.4 Implementation

Each component of the platform is implemented as a Spark program written in Scala. Intermediate results are saved in Parquet files, a columnar file format, which Spark can natively read from and write to. Implementing the platform as a series of Spark jobs ensures that, for components which are able to be easily parallelized, the platform can scale horizontally based on available hardware.

3.4.1 Extract

The first stage of the knowledge graph construction involves running the `extract` Spark job which requires the following parameters:

- `solr.uri` - the URI of the Solr server containing the documents
- `solr.index` - the name of the index on the Solr server to read from

A number of auxiliary options are available such as the query to execute (default to all documents), the number of partitions for the Spark RDD to generate, and more.

The job first starts by defining a CoreNLP pipeline with annotators enabled based on the desired task. In this thesis, as information extraction and entity linking are being performed, the `kbp` and `entitylink` annotators were enabled. These, however, have prerequisites as outlined in Section 3.1.1. As such, the annotators that `kbp` and `entitylink` depend on needed to be enabled as well; these are outlined in Table 3.2.

Next, the job read batches of 10,000 documents at a time from Solr using the configured query and a cursor to keep state. These documents were stored in a RDD such that

Annotator	Description
tokenize	tokenize the input document
ssplit	split the input document into sentences
pos	identify the part-of-speech of each token
lemma	reduce tokens to their lemmas
parse	constituency parsing on tokens
ner	named entity recognition on tokens
coref	co-reference resolution on tokens
kbp	knowledge base population (information extract) on named entities
entitylink	link mentions of named entities to their canonical Wikidata entity

Table 3.2: CoreNLP annotators for information extraction and entity linking

subsequent transformations to the documents (i.e., CoreNLP annotators) can be applied in parallel across the cluster. Empty documents and documents with sentences longer than 256 characters were filtered out. These documents characterize web pages that contain HTML tables where table cells were concatenated into very long "sentences" by the document parser. These abnormally long sentences lead to greatly increased processing time by CoreNLP, due to algorithms with super-linear complexity, which do not characterize typical conditions.

For each document from Solr, a CoreNLP document was constructed which was run through the annotation pipeline. The pipeline takes the document, applies each annotator to it based on the topological sort of dependencies, and adds auxiliary information from each annotator to fields in the original document. From this annotated document, two different fields were populated by the annotator: the `entityMentions` and `relations` fields.

The `entityMentions` field holds information about the entities extracted from the document. First, for each entity mention a tuple of the form `([DOC], MENTIONS, [UUID])` was produced where `DOC` is the document ID, `MENTIONS` is a new link type defined for indicating a mention, and `UUID` is a generated UUID for the mention. If the mention text had not previously occurred in the document, a random UUID was generated and a mapping from mention text to UUID was kept track of. If the mention text had previously occurred (i.e., it's in the UUID mapping for this document), the existing UUID was used. For each mention, the class (`person`, `organization`, etc.), the mention text, the character offsets of the mention text, and the normalized date (if the mention is a date) was added to the tuple as metadata. Second, a tuple was produced for each entity mention containing

its entity link to Wikidata. If the mention could be linked, a tuple of the form `([UUID], LINKS_TO, [URI])` was produced where `UUID` is the mention's UUID, `LINKS_TO` is the name given this link, and `URI` is the URI of the Wikidata entity. If the mention could not be linked, the same tuple with a `NULL` value for `URI` was produced.

The `relations` field holds information about the relations between mentions of entities extracted from the document. For each relation, a tuple of the form `([SUB_UUID], [REL], [OBJ_UUID])` was produced where `SUB_UUID` is the source mention's UUID, `REL` is the name of the link, and `OBJ_UUID` is the target mention's UUID. Each tuple was also augmented with metadata containing the confidence level of the relation.

After all documents had been processed, the results were written to disk such that they could be further processed by subsequent Spark jobs.

3.4.2 Enrich

The `Enrich` Spark job takes the extracted assertions and produces new tuples containing facts (ground-truth values) from Wikidata. In order to do this, SPARQL queries were issued against a Jena instance storing a dump of Wikidata. This required a mapping from CoreNLP relations to Wikidata property types which was produced with minimal manual effort due to the limited number of relation classes in CoreNLP.

First, the mapping from CoreNLP relations to Wikidata properties was read from disk and stored in a Spark broadcast variable which was used for quick look-ups by Spark workers. Next, all `LINKS_TO` links were inspected from the previous stage and the distinct, non-null entities that were mentioned in the documents were kept; ground-truth values only needed to be fetched once for each entity.

Finally, ground-truth values were fetched by issuing a number of queries against Jena. A SPARQL query was issued using the Wikipedia URL of the entity as the subject with a predicate of `<http://schema.org/about>` where the returned object is the Wikidata ID; this is required as CoreNLP returns Wikipedia URIs rather than Wikidata URIs. Next, a list of all properties that are available for the entity in Wikidata was fetched and compared with the relations extracted from documents. For each of these properties, the property's value was fetched from Wikidata and a fact tuple of the form `([URI], [REL], [VALUE])` was produced where `URI` is the entity's URI, `REL` is the name of the relation, and `VALUE` is the ground-truth value. This final step corresponds to Figure 3.5.

After all entities were processed and ground-truth values had been fetched, the results were written to disk such that they can be further processed to subsequent Spark jobs.

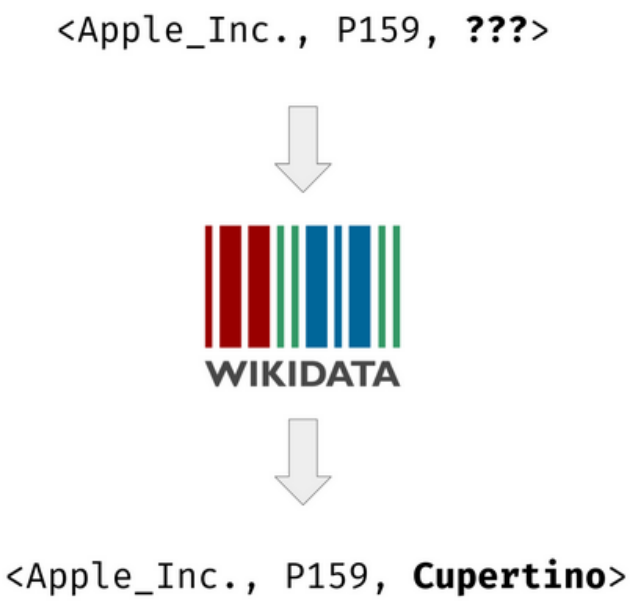

Figure 3.5: Fetching ground-truth values from Jena

3.4.3 Load

The `Load` Spark job takes tuples that had been written to disk and inserts them into Neo4j. The job is passed a directly containing tuples that had been saved to disk. Tuples are batched and inserted into Neo4j using the `MERGE` directive in order to ensure that new nodes were merged with existing nodes rather than creating new nodes each time. This is important as the majority of the tuples contain mentions to existing nodes (e.g., documents or entities).

3.5 Performance Evaluation

This section presents the results of the performance analysis of the Extract and Enrich stages of the platform where the Load stage has been folded into each section.

In order to test the horizontal scale-out of the platform, the tests were run on a cluster of 9 nodes. Each node has 2× Intel E5-2670 @ 2.60GHz (16 cores, 32 threads) CPUs, 256GB RAM, 6×600GB 10K RPM HDDs, 10GbE networking, and runs Ubuntu 14.04 with Java 9.0.4. One node hosts master services (YARN ResourceManager and HDFS

NameNode) while the remaining nodes each host a HDFS DataNode, a Solr shard, and are available for Spark worker allocation via YARN.

In terms of a graph database, the Community Edition of Neo4j was used. A single-node Neo4j instance was run on the same node as the master services and a single Spark worker, co-located on the same node, was used for loading tuples in the database. Neo4j also supports a distributed setting via the Enterprise Edition, although a commercial license is required.

3.5.1 Extract

In order to test the performance of extraction, the end-to-end runtime which includes fetching documents from Solr, performing information extraction using CoreNLP, and writing the tuples to HDFS was measured. A Spark job was configured to run with 32 executors (4 per machine) with each executor allocated 8 CPU cores. Each executor was also allocated 48GB of memory to have sufficient overhead to store batches of documents from Solr and the CoreNLP models in memory. Using this configuration, the resource usage across the cluster was maximized.

After fetching documents from Solr, those with length greater than 10,000 tokens and sentences longer than 256 tokens were discarded as mentioned previously. After filtering, there remained approximately 580K documents.

97M tuples were extracted in 10.4 hours which amounts to a processing rate of 13K tokens per second. The breakdown of the tuples is as follows:

- 46.1M - mentions of entities
- 46.1M - mention-to-entity links
- 4.8M - relations between mentions

Although there were 46.1M mention-to-entity links, 30.7M link to an actual Wikidata entity while the remaining links were explicitly set to NULL which could signify the opportunity to add new entities to the external knowledge graph. The 30.7M mention-to-entity links represent 324K distinct entities. Loading the 97M tuples into Neo4j took 7.8 hours which amounts to an ingestion rate of 2.9K tuples/sec.

3.5.2 Enrich

In order to test the performance of the enrichment phase, the time taken to query and fetch ground-truth values from Jena was measured. A Spark job was configured to use a single executor using a single CPU core and 8GB of memory as the enrichment process is not compute or memory heavy but instead is bounded by the performance of Jena.

For each of the 324K distinct Wikidata entities extracted from the documents, ground-truth values were fetched based on the CoreNLP relation to Wikidata property mapping. This means that for each distinct entity, a query was performed for each of the available relation types until a match is found as Wikidata entities of a certain type are not guaranteed to have certain properties. For example, for the CITY_OF_HEADQUARTERS relation, one particular query would check if Wikidata contains property P159[10] for a particular entity and if so, fetch and store this fact in Neo4j.

Querying was done against a local installation of Jena co-located on the same node as the Spark worker; 11.7K ground-truth values for the CITY_OF_HEADQUARTERS relation were fetched in about 14 mins. These values were inserted into Neo4j in a matter of seconds.

[10] https://www.wikidata.org/wiki/Property:P159

Chapter 4

Rule-based Data Curation

Using DSTLR to create a knowledge graph supports the development of a number of applications for data curation which previously would have required the end-user to develop the text document to distilled knowledge graph pipeline themselves. With the knowledge graph available in Neo4j, the user is able to focus on the development of applications to solve a given task instead of focusing on the infrastructure and tools for creating the knowledge graph in the first place. One of the main selling points of Neo4j is the intuitive and powerful Cypher query language and this, in turn, was one of the driving factors behind using Neo4j as a graph database for storing the knowledge graph.

4.1 Cypher Query Language

Cypher is a declarative query language originally built for the Neo4j graph database, but has since been open-sourced as openCypher[1]. Currently a number of different graph database systems, including Graphflow [6], utilize Cypher while it has also been voted for inclusion into Apache Spark 3.0 as the graph query language. Cypher follows the property graph model where the primitives are nodes and relationships which may optionally have properties (key–value pairs) associated with them [8].

[1] https://www.opencypher.org/

4.2 Data Curation Scenarios

In this section, I highlight a number of data curation scenarios focused around fact-checking information extracted from text documents. Specifically, discussion on supporting information, inconsistent information, and missing information is provided which can have a number of different applications.

4.2.1 Supporting Information

Supporting information is defined as an agreement between values extracted from text documents and the corresponding ground-truth values in the external knowledge graph. More concretely, for an assertion of the form (SOURCE, LINK, TARGET) where the SOURCE mention has a linked Wikidata entity with a ground-truth value available that is aligned with relation type in LINK, the value of the Wikidata property should match TARGET.

```
MATCH (d:Document)-->(s:Mention)
MATCH (s)-->(r:Relation {type:  "CITY_OF_HEADQUARTERS"})-->(t:Mention)
MATCH (s)-->(e:Entity)-->(f:Fact {relation:  r.type})
WHERE t.span = f.value
RETURN d, s, r, t, e, f
```

Figure 4.1: Cypher query to find supporting facts.

In order to discover sub-graphs that match this pattern, the Cypher query in Figure 4.1 is issued against Neo4j. This query matches DOCUMENT nodes that contains a source MENTION with a CITY_OF_HEADQUARTERS relation to a target MENTION node. The source node should have a corresponding Wikidata ENTITY that links to a FACT of the same type as the relation. The WHERE clause ensures that the value from the document (the span of text from the target) matches the value of the fact in Wikidata.

Figure 4.2 shows a concrete example of one such sub-graph. A document mentions entities Good Technology and Sunnyvale and states that former is headquartered in the latter. The platform links Good Technology to its Wikidata entity, Good_Technology[2], which contains a fact corresponding to CoreNLP's CITY_OF_HEADQUARTERS relation. Note that the right-most (value from the document) and left-most (value from Wikidata) node values are in agreement.

[2]https://en.wikipedia.org/wiki/Good_Technology

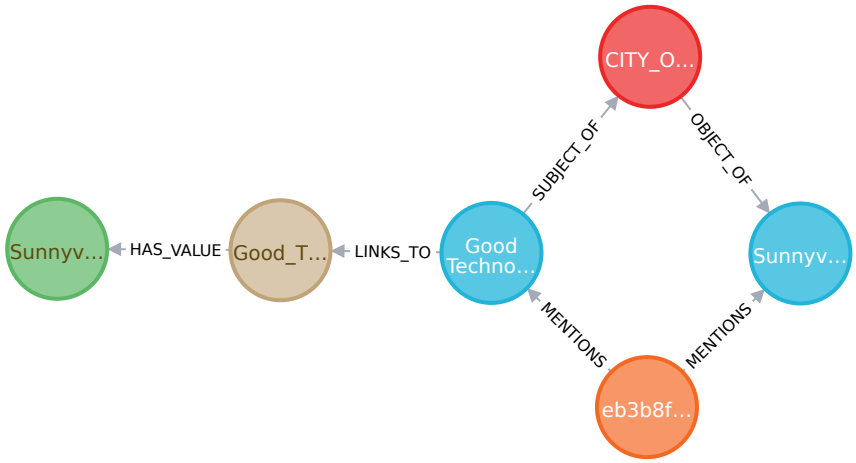

Figure 4.2: Neo4j sub-graph for an example of a supporting fact.

This type of query provides bi-directional supporting evidence for information verification for both the document and Wikidata. On one hand, it is known that this particular piece of information present in the document is factual (at least, in accordance to our source of ground-truth) as it matches the ground-truth value in the external knowledge graph. On the other hand, this document provides supporting evidence for the fact in Wikidata being accurate as the creation process of a knowledge graph often loses provenance to its source. That is, the question of "Where did we get this fact from?" cannot be directly answered from information within the knowledge graph. Using our platform, large volumes of text can be processed in order to provide supporting evidence.

4.2.2 Inconsistent Information

Inconsistent information is defined as a disagreement between values extracted from text documents and the ground-truth values in the external knowledge graph. For an extracted tuple (SOURCE, LINK, TARGET) where the SOURCE has a linked Wikidata entity with a ground-truth value available that is aligned with the relation type in LINK, the value of the Wikidata property would be different than TARGET.

The Cypher query in Figure 4.3 can be run against Neo4j to find sub-graph instances that match the definition of inconsistent information. The MATCH directives in this query is the same as for supporting information (Figure 4.1) however, the WHERE clause is negated.

```
MATCH (d:Document)-->(s:Mention)
MATCH (s)-->(r:Relation {type:  "CITY_OF_HEADQUARTERS"})-->(t:Mention)
MATCH (s)-->(e:Entity)-->(f:Fact {relation:  r.type})
WHERE NOT(t.span = f.value)
RETURN d, s, r, t, e, f
```

Figure 4.3: Cypher query to find missing facts.

Figure 4.4 shows an instance of a sub-graph showing inconsistent information. A document mentions the company Isetan and the city of Paris and the extractor believes there is a CITY_OF_HEADQUARTERS relations between them. Iseten is linked to its Wikidata entity[3] which contains the ground-truth value of "Tokyo" for the relation. Note that the right-most (value from the document) and left-most (value from Wikidata) node values are different.

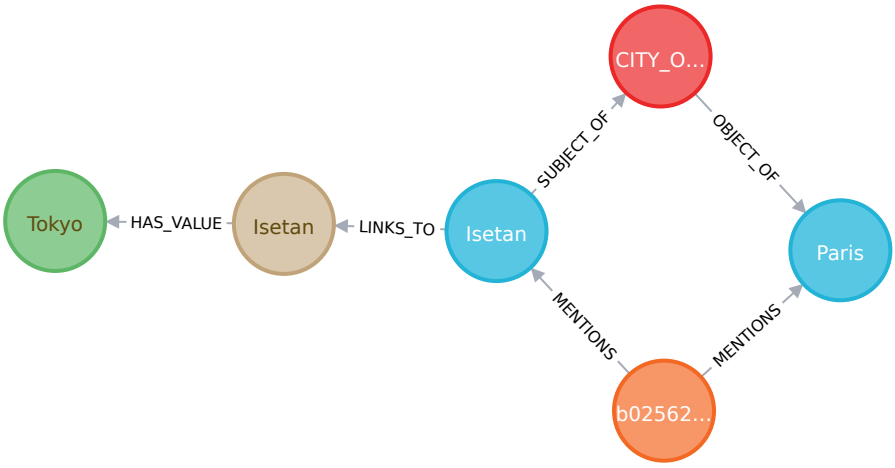

Figure 4.4: Neo4j sub-graph for an example of an inconsistent fact.

This type of query has two different uses: it enables manual repairs and can automatically generate labelled training data for training relational extractors. In terms of repair, it identifies information in text documents that differ from the ground-truth values. As provenance is retained back to the span of text in the source document, this enables the manual repair of documents. In order to help alleviate the impact of inaccurate extractors,

[3]https://en.wikipedia.org/wiki/Isetan

signals from multiple tuples that express the same relation (via having the same SOURCE and relation in LINK) could be combined and potentially used for automatic repairs.

This use case also serves as a method of data augmentation – it can generate labelled positive and negative examples that rid of the need for manual annotations by humans. From manual inspection, the inconsistent fact from the query above was due to an error in the extractor. Due to the provenance retained for the SOURCE and the known ground-truth value, each inconsistent fact can produce a negative example (the tuple itself) and a positive example (by substituting in the ground-truth value). Likewise, each supporting fact from the previous section could be used as a positive example as well.

4.2.3 Missing Information

If no ground-truth value is present for a given relation, it cannot be determined whether a tuple provides supporting or inconsistent information. It does, however, provide an opportunity to add information to the external knowledge graph. Care has to be taken to ensure that the information is accurate since extractors can be imperfect. Just like with inconsistent information, signals from multiple tuples could be combined in order to determine if a relation is worth adding to the knowledge graph.

The Cypher query in Figure 4.5 returns sub-graphs where there is no ground-truth value. Following the same MATCH semantics as before, this queries differs in that it does an OPTIONAL MATCH on the FACT (since it may not exist) and then only returns the instances where the FACT is actually null.

```
MATCH (d:Document)-->(s:Mention)
MATCH (s)-->(r:Relation {type:  "CITY_OF_HEADQUARTERS"})-->(t:Mention)
MATCH (s)-->(e:Entity)
OPTIONAL MATCH (e)-->(f:Fact {relation:  r.type})
WHERE f IS NULL
RETURN d, s, r, t, e, f
```

Figure 4.5: Cypher query to find inconsistent facts.

A concrete example of a sub-graph returned by this query is shown in Figure 4.6. Note that there is no outgoing HAS_VALUE edges from the left-most node as in the previous examples. This indicates a missing fact in Wikidata.

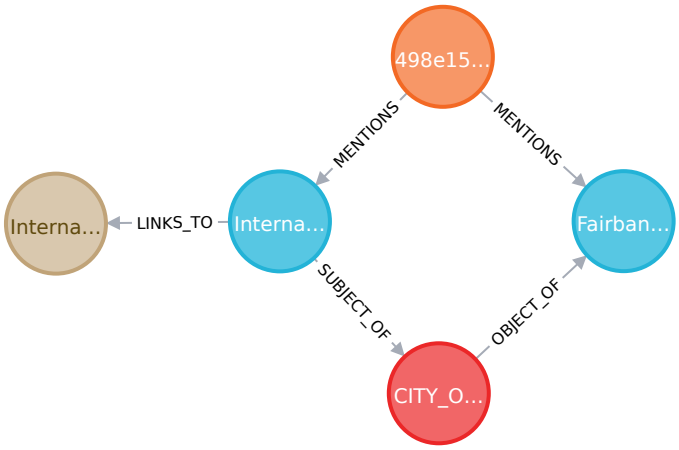

Figure 4.6: Neo4j sub-graph for an example of a missing fact.

Although missing ground-truth values have been found, it cannot determined if there is enough information in the document collection to say, with reasonable certainty, whether the information is accurate. The query in Figure 4.7 finds the missing values as before and groups by the ENTITY and RELATION then returns a list of values (along with the documents they appear in) to the user. The distribution could be analyzed and used to determine how accurate the available information is before providing the new information to the external knowledge graph. For example, for a given entity and relation a histogram of possible values could be built and only the top element could be deemed as accurate. Further rules, such as the values must come from a certain number of different documents, could be in place as well.

```
MATCH (d:Document)-->(s:Mention)-->(r:Relation)-->(t:Mention)
MATCH (s)-->(e:Entity)
OPTIONAL MATCH (e)-->(f:Fact {relation:  r.type})
WHERE f IS NULL
WITH e as ent, r.type as rel, collect(d.id) as docs, collect(t.span) as vals
RETURN ent, rel, docs, vals
```

Figure 4.7: Cypher query to find inconsistent facts.

4.3 Other Use Cases

Outside of the examples outlined above, it is not hard to imagine other use cases that the platform could support. For example, DSTLR could form the basis of a question answering service. Using a large language model such as BERT [5], the question could be mapped to the closest relation available in the knowledge graph while further matching could reduce the set of SOURCE and TARGET nodes that are applicable to the question text.

Chapter 5

Conclusion

In this thesis, I presented DSTLR – a platform for constructing knowledge graphs from large collections of text documents. First, I showed how text documents on disk can be indexed into the distributed Solr search engine through Anserini. Second, I showed how natural language processing tools can be horizontally scaled out via Apache Spark in order to perform information extraction at scale using a cluster of machines. Third, I showed how linking entities to an external knowledge graph and aligning relations from CoreNLP to properties available in Wikidata can enrich the information extracted from the documents. Finally, I demonstrated a number of data curation use cases on top of the generated knowledge graph which motivated the creation of the platform.

Moving forward, there are several improvements to the platform that would improve its capabilities. First, moving away from Knowledge Base Population methods of information extraction towards Open Information Extraction approaches would improve the coverage of relations that are able to be extracted. However, this leads to the creation of superfluous relations while also introducing the problem that relations from the text can no longer be aligned manually with those available in an external knowledge graph. Clustering of relations using embedding models and matching to an external knowledge graph predicate ontology may be a viable approach. Second, making the platform more modular within the natural language processing tools would be beneficial. Currently, the platform is tightly bound to CoreNLP as the inherit dependencies between NLP tasks is handled by the library. Some form of intermediate persistence within the information extraction process itself such that custom named entity extractors or entity linkers could be used would improve the flexibility of the platform and make it a platform for continued NLP research.

References

[1] G. Angeli, M. Premkumar, and C. Manning. Leveraging linguistic structure for open domain information extraction. In *ACL — IJCNLP 2015*, volume 1, pages 344–354, 2015.

[2] R. Clancy, T. Eskildsen, N. Ruest, and J. Lin. Solr integration in the Anserini information retrieval toolkit. In *SIGIR*, 2019.

[3] R. Clancy, J. Lee, Z. Akkalyoncu Yilmaz, and J. Lin. Information retrieval meets scalable text analytics: Solr integration with spark. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1313–1316, 2019.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[5] J. Devlin, M. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[6] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698. ACM, 2017.

[7] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *ACL 2014*, pages 55–60, 2014.

[8] J. Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, volume 2324, 2013.

[9] M. Surdeanu, J. Tibshirani, R. Nallapati, and C. Manning. Multi-instance multi-label learning for relation extraction. In *EMNLP 2012*, pages 455–465. Association for Computational Linguistics, 2012.

[10] P. Yang, H. Fang, and J. Lin. Anserini: Enabling the use of Lucene for information retrieval research. In *SIGIR*, pages 1253–1256, 2017.

[11] P. Yang, H. Fang, and J. Lin. Anserini: Reproducible ranking baselines using Lucene. *JDIQ*, 10(4):Article 16, 2018.

[12] M. Zaharia, R. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.