CS 651 - Data-Intensive Distributed Computing

University of Waterloo

Department of Computer Science

Information Retrieval using Flink and Kafka

Authors:

Shruti Srivatsan (21000026)
Shakti Singh Rathore (21033525)
Krishna Jayakumar (21040057)

1 INTRODUCTION

In the rapidly evolving field of big data, Information Retrieval (IR) has become a critical aspect for organizations looking forward to harnessing the value of large amounts of information. It has emerged as a pivotal element for organizations aiming to extract value from extensive datasets. As the volume of information continues to grow exponentially, the need for efficient management, processing, and retrieval becomes paramount[12]

The rise of advanced technologies, coupled with sophisticated algorithms, has propelled IR these days to the forefront. With applications spanning search engines, recommendation systems, and data analytics among others, it plays a key role in enabling timely and precise access to information. Its newfound importance reflects a strategic shift toward harnessing the full potential of data, fostering informed decision-making and innovation across diverse industries[1].

Apache Flink[7], a distributed stream processing framework, performs well in real-time data analytics, offering organizations the capacity to process information with lesser latency and fault tolerance. Its flexibility in building complex data processing pipelines makes it a cornerstone for applications requiring rapid and dynamic data handling. Complementing Flink is Cassandra[17], a scalable NoSQL database, which gives a reliable solution for storage and retrieval of vast amounts of data across distributed environments. The highlight is its decentralized architecture, which aligns seamlessly with Flink's distributed processing model, forming a resilient foundation for managing diverse and evolving datasets.

Another pivotal component is Kafka[16], known for its distributed streaming platform which handles high-throughput data streams and facilitates seamless communication between various components of an IR system. Its message queuing and fault-tolerant design promotes efficient data ingestion and real-time processing, fixing the gap between data producers and consumers in the IR ecosystem.

Integration of sophisticated IR algorithms like BM25 boosts the capabilities of the system further. These algorithms, ranging from advanced text analysis to machine learning techniques, contribute to the precision and relevance of information retrieval[6]. They enable organizations to deduce actionable insights by understanding user intent, context, and the intricacies of natural language. Integrating all the above approaches offers real-time responsiveness, and promotes scalability, adaptability. Organizations can respond to user queries in real time, develop scalable systems to handle growing data volumes, and also adapt to evolving business requirements seamlessly.

2 FLINK

Apache Flink was introduced in 2010 by Professor Volker Markl at the Technical University of Berlin[2] for batch processing. However, due to Spark's dominance in this domain, the focus for Flink strategically shifted towards real-time stream processing. Despite its relatively recent emergence, Apache Flink has undergone vast development and has positioned itself in the evolving landscape of big data streaming computing.

Flink has been used widely in domains like healthcare[15] and energy consumption[4]. Organizations prefer using Flink since it seamlessly transitions between batch and stream processing, offering unparalleled flexibility in handling diverse data processing scenarios. Its fault-tolerant architecture ensures robustness in large-scale analytics, mitigating the impact of failures. Apache Flink is known to excel in event time handling, making it suitable for applications requiring precise temporal processing. Furthermore, its interoperability with popular big data technologies like Apache Hadoop and Apache Kafka enhances its versatility and integration capabilities, strengthning its position as a powerful and adaptive solution in the dynamic landscape of big data analytics [8].

The architecture [14] mainly comprises of JobManagers and TaskManagers. JobManagers coordinate the overall execution plan, while TaskManagers execute parallelized tasks across the data stream. A distribution coordination service like Apache Zookeeper is used for high availability and fault tolerance. Efficient resource allocation is done using adaptive scheduling and memory management [5].

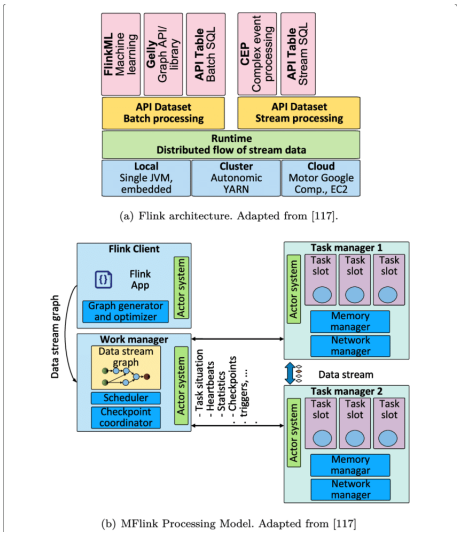

Fig. 1. Flink Architecture

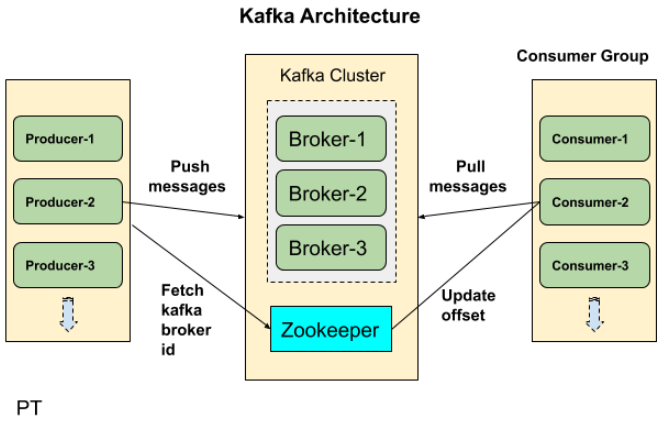

Fig. 2. Kafka Architecture

3 KAFKA

Apache Kafka is a scalable, highly available, distributed event processing platform used by many top tech companies today. It was developed at Linkedin by Jay Kreps, Jun Rao, and Neha Narkhede [9] in late 2000s to solve the problem of real-time processing of large amounts of event data. It was open sourced in 2011 and became top level Apache project in 2012. Kafka is mostly used to build real-time streaming pipelines that process continuous data stream from multiple sources, aggregate and move that data from one system to another.

Kafka architecture [13] mainly comprises of Kafka Cluster, Producer API and Consumer API. Cluster maintains multiple Kafka brokers which are set of servers that store messages. Single or a group of producers publish messages to a topic. Each topic is a stream of particular type of messages which is divided into multiple partitions and stored in different

broker depending on the replication config [9]. Consumers subscribe to particular topic and consume messages by maintaining an offset. This offset indicates to the kafka broker that consumer has read all the messages prior to this offset. Kafka brokers are managed by Zookeeper which handles addition and removal of any broker from the system.

4 INFORMATION RETRIEVAL

In information Retrieval, being able to retrieve the relevant document with low latency is important, especially with the vast amounts of data available. The data structure called an inverted index is what is used to perform efficient retrievals[11]. An inverted index is a collection of terms and a postings list associated with each term with each postings comprising of a document id and any information associated with that document. The associated information is could be simple information like the term frequency or something complex like the term position. In our project, the postings lists consisted of the document id, term frequency and document length.

The retrieval models we will be exploring in this project include include TF-IDF, Vector Space Modelling(VSM) and OKAPI BM25 models which are all ranked retrieval models. They are differentiated from Boolean retrieval models in that the queries use free text which rather than precise language and operators [12, p.14]. The scores in these models are calculated for each document and then then each document is ranked.

4.1 TF-IDF

TF-IDF[12, p.118] is a weighting scheme which is a combination of the TF(Term Frequency) which is the number of times a term occurred in a document and the IDF(Inverse Document Frequency), which is the inverse of the number of documents that contain a term. This model seeks to give high weight when a term occurs many times in a certain document and lower weight if that term occurs in many different documents. The formula for weighting the term and document is given below.

$$\text{TF-IDF}_{t,d} = \text{TF}_{t,d} \times \text{IDF}_t \tag{1}$$

$$\text{IDF}_t = \log\left(\frac{N}{\text{DF}_t}\right) \tag{2}$$

$$\text{Score}(Q,d) = \sum_{t \in Q} \text{TF-IDF}_{t,d} \tag{3}$$

where, $TF - IDF_{t,d}$, is term frequency of t in document d, IDF_t is inverse document frequency of term t in a document collection, N is the total number of documents in the collection and DF_t is the number of documents containing the term t. Score(q,d) is the total score for a given document d which is summed over all query terms q to give the final score to be used for ranking.

4.2 Vector Space Model

In the vector space model[12, p.123], we look at both documents and queries as vectors that have the length of the vocabulary size. This makes it easy to find the similarity between a document d and a query Q using cosine similarity to get a score. The cosine similarity essentially measures the angle between two vectors in the euclidean space (the size of vocabulary) which is what the numerator of $sim(Q, d_i)$ shown below measures. The denominator is the normalization factor which minimizes the effect of the length of the document. Instead of the proper normalization done in [10] we used the square root of the number of terms in d_i as the computational costs are lower while still lowering importance on document length.

$$\text{sim}(Q, d_i) = \frac{\sum_{j=1}^{V} w_{Q,j} \cdot w_{i,j}}{\sqrt{\text{number of terms in } d_i}} \tag{4}$$

where $w_{Q,j} \cdot w_{i,j}$ is equivalent to $tf_{Q,j} \cdot idf_j$.

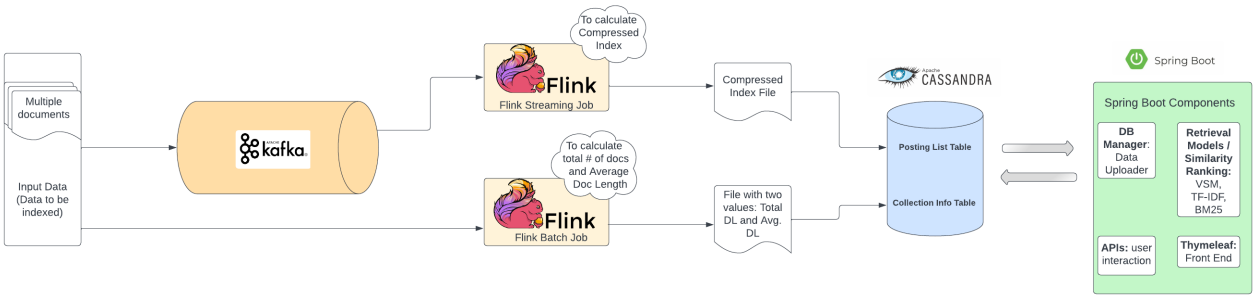

Fig. 3. Architecture Diagram

4.3 Okapi BM25

BM25 weighting scheme also called the Okapi weighting scheme is a probabilistic model [12, p.232] that gives importance to term frequency and document length. Here the IDF score is different to that of the previous two weightings.

$$\text{IDF}_t = \sum_{t \in q} \log\left(\frac{N - df_t + \frac{1}{2}}{df_t + \frac{1}{2}}\right) \tag{5}$$

$$\text{score}(D, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{tf_{t,d} \cdot (k_1 + 1)}{tf_{t,d} + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)} \tag{6}$$

Here, the $|D|$ is the document length, *avgdl* is the average document length while k_1 and b are tuning parameters. k_1 correlates with the weight we give to term frequency with $k_1 = 0$ meaning no weight is given to the term frequency. b controls the scaling to document length where if $b = 0$ then there is no normalization of the length and $b = 1$ means that means full scaling by length.

5 IMPLEMENTATION

5.1 Architecture Overview

The goal of this system is to fast retrieve matching information from a large dataset. This entire information retrieval system comprises of multiple top level Apache projects. The architecture is divided into two main parts: a data processing part comprising of Kafka and Flink to index the dataset, and a retrieval part comprising of Cassandra and Spring boot to interact with users and return relevant documents based on user queries. Sections below explain the entire proposed architecture as shown in fig. 2 in detail.

5.2 Data Transformation and Preparation

We describe all the steps in detail in the following sections. We first start the kafka stream by running our KafkaProducer. Section 6.1 contains information on how to start each service used in this system step by step. KafkaProducer reads each document from the input collection, appends an id at the start which is the offset of the document in the collection file. It also uses a modified Bespin tokenizer to clean the special character, numbers etc from the document. Every 5 microseconds, producer publishes one message to the "indexer-topic" topic in the format shown in Fig. 4. Downstream jobs subscribe themselves to this topic to further process the documents.

5.3 Processing

We implemented multiple Flink jobs to process the stream of messages and create inverted indexes.

5.3.1 Inverting Indexing Streaming Job. We implemented two methods of creating inverted indexes. In the first method shown in Fig 5. and Fig 6, we created the inverted index using two jobs. In the second method we used a single job

```
krishthek@Krishnas-MBA BigData-Retrieval-Project % java -cp target/assignments-1.0.jar ca.uwaterloo.cs451.kafkaproducer.KafkaProducer data/Shakespeare.txt
Start writing to kafka producer....
Message sent to Kafka: 0 THE SONNETSthesonnets
Message sent to Kafka: 18 by William Shakespearebywilliamshakespeare
Message sent to Kafka: 42   From fairest creatures we desire increase,fromfairestcreatureswedesireincrease
Message sent to Kafka: 113   That thereby beauty's rose might never die,
Message sent to Kafka: 159   But as the riper should by time decease,
Message sent to Kafka: 202   His tender heir might bear his memory:
Message sent to Kafka: 243   But thou contracted to thine own bright eyes,
Message sent to Kafka: 291   Feed'st thy light's flame with self-substantial fuel,
Message sent to Kafka: 347   Making a famine where abundance lies,
Message sent to Kafka: 387   Thy self thy foe, to thy sweet self too cruel:
Message sent to Kafka: 436   Thou that art now the world's fresh ornament,
Message sent to Kafka: 484   And only herald to the gaudy spring,
Message sent to Kafka: 523   Within thine own bud buriest thy content,
Message sent to Kafka: 567   And tender churl mak'st waste in niggarding:
Message sent to Kafka: 614     Pity the world, or else this glutton be,
Message sent to Kafka: 659     To eat the world's due, by the grave and thee.
Message sent to Kafka: 710   When forty winters shall besiege thy brow,whenfortywintersshallbesiegethybrow
Message sent to Kafka: 780   And dig deep trenches in thy beauty's field,
Message sent to Kafka: 827   Thy youth's proud livery so gazed on now,
Message sent to Kafka: 871   Will be a tattered weed of small worth held:
Message sent to Kafka: 918   Then being asked, where all thy beauty lies,
```

Fig. 4. Kafka Producer publishing stream of messages

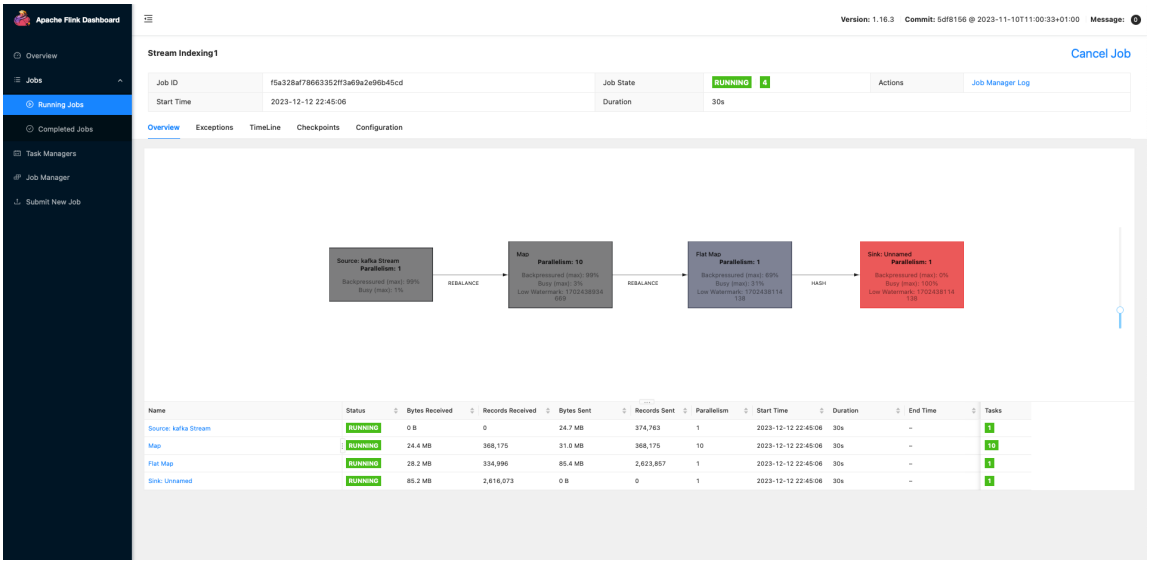

Fig. 5. Method 1 Job 1 for creating inverted indexes

which is shown in Fig 7.

In method one, the first job("**Indexer1.java**" file), we use Flink's Datastream API where the datastream originates from the Kafka source. Each line can be considered as a document and as they come in a flatMap operation splits the line into words, then the term frequency for every term in that line is calculated using a histogram, as well as the length of the document and we emit a tuple of the term, document id, term frequency and document length. We then store this output in a textfile. In the second job("**CollectPostingListsJob.java**"), a keyBy operation followed by a reduce operation is performed on the data based on the term, which does something similar to the the groupBy operaion in Spark. The two operations esentially groups the similar terms together so that the resulting ouput is a tuple of the term and a list of postings for each document related to that term. In the reduce function, we also compress the document IDs. The final inverted index is then sent to Cassandra to be stored for later retrieval.

In method2 we directly create the inverted index in a single job("**IndexWind.java**"). Since Flink data-streams work withing time windows, the approach we did was to treat the entire stream as a single window [9] by taking advantage

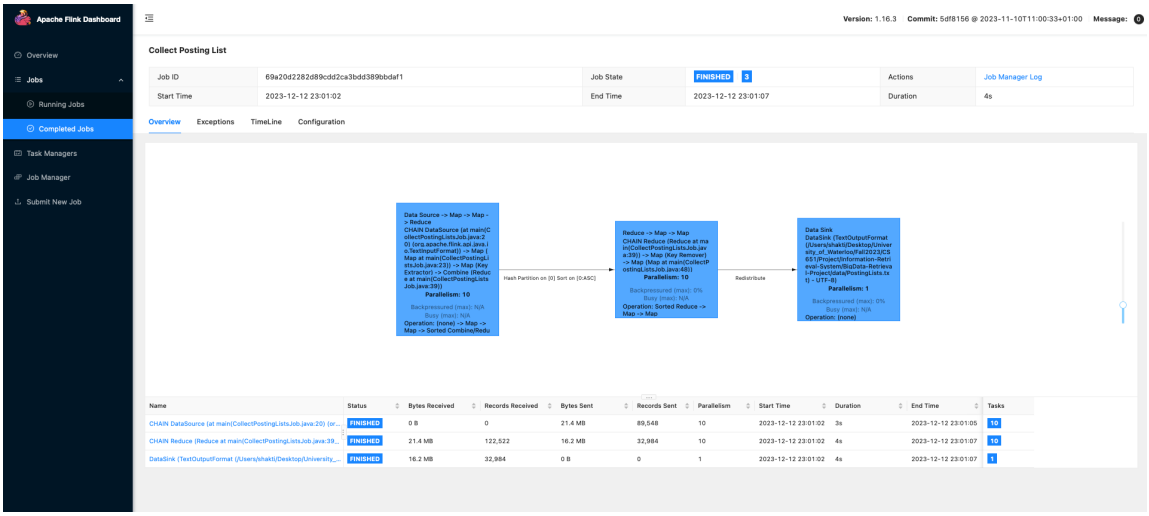

Fig. 6. Method 1 Job 2

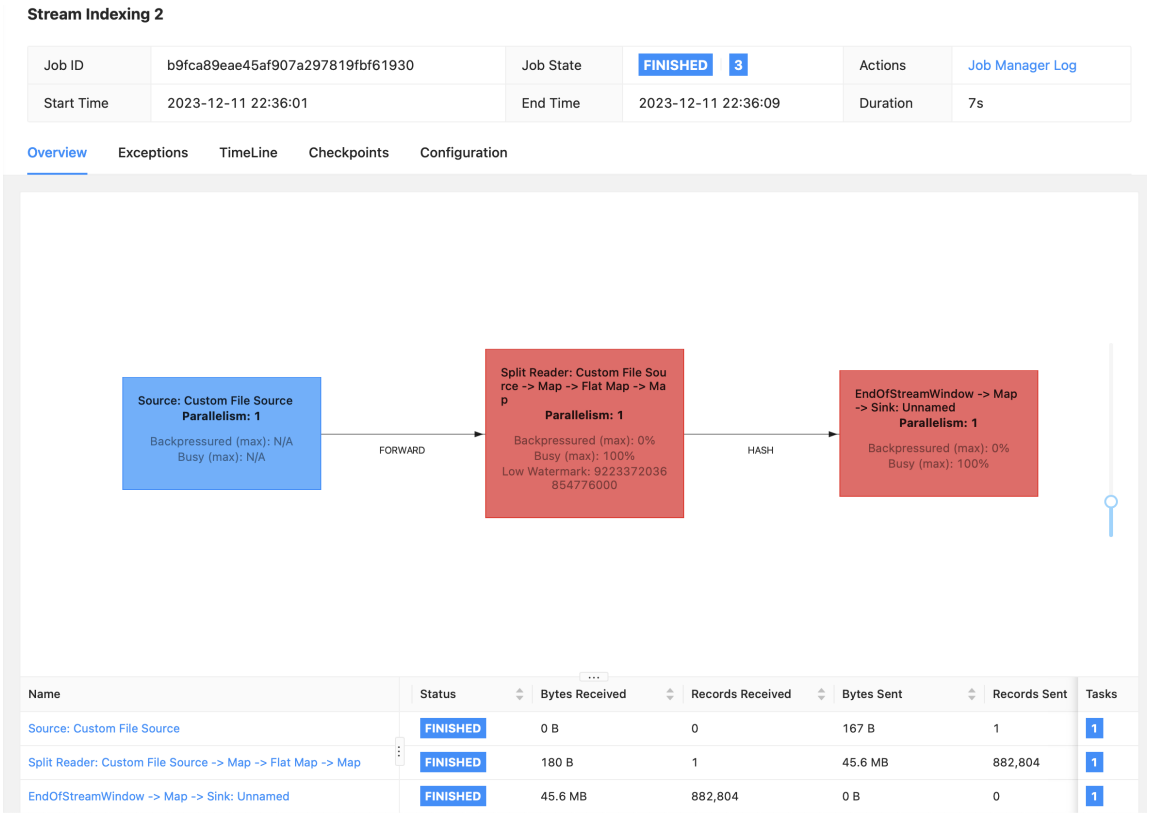

Fig. 7. Method 2 for creating inverted indexes

```
cqlsh> select * from indexer_space.posting_list_by_term4 where term='acquire';

 term    | postings
---------+-----------------------------------------------------------
 acquire | [[386098, 1, 7], [1080108, 1, 10], [315309, 1, 10]]

(1 rows)
```

Fig. 8. Posting_list_by_term Table sample result

```
cqlsh> select * from indexer_space.collection_info_by_name ;

 name        | avgdl | total_docs | uuid_time
-------------+-------+------------+--------------------------------------
 Shakespeare |     8 |     122458 | fd4f85ea-949f-11ee-bb5e-325096b39f47

(1 rows)
```

Fig. 9. collection_info_by_name Table sample result

of the **EndOfStreamWindow** class [3]. The class essentially looks at the window from start to end of the stream where the end of the stream is signified by +infinity time. This could be done because we assume that the stream is bounded. We were able to perform the processing when the source was a text file as there is a clear ending to a text file. However, when we used a Kafka source there were difficulties with getting posting lists for all terms as there was issues with the window processing.

5.3.2 Average Document Length Batch Job. This job is just to read the raw data from the input collection file and calculate the average document length for the entire collection. We need avg DL value in our BM25 retrieval model to rank the documents similarities. We used Flink DataSet API to process the documents. Flink's DataSet API is to perform batch operations on the data. "**AvgDocumentLengthJob**" filename in our project folder is the job that calculates avg document length and total number of documents. It writes the result in txt file in the format - (DocInfo,122458,8) where "122458" is the total documents and 8 is the average document length.

5.4 Data Storage

Inverted indexes are uploaded to Apache Cassandra's table - posting_list_by_term using Datastax Java Driver and Google's GSON library. This is done periodically whenever one stream of messages are completely processed. Currently this table contains inverted indexes for Shakespeare.txt collection but any collection can be streamed, processed, indexed, and stored in Cassandra for long term using the processes defined above in this section. Average Document Length Batch Job result is also saved in Cassandra's collection_info_by_name. Fig 8. and Fig 9. show cqlsh window with sample queries.

5.5 Retrieval

We created Java Spring-boot web application to interact with the users. User submit its query and choice of ranking model (VSM, TF-IDF, BM25) via the form available at **/search** API. In the back-end, similarity module ranks the documents based on model selected by the user by fetching inverted indexes from Cassandra, creating subset of common documents, and giving similarity score to each document. Top k documents with max similarity scores are return to the user as a response to the /search API call.

This spring-boot application handles entire communication with Cassandra. It takes care of batch uploading inverted indexes created by Flink streaming job. "**CassandraService**" file has **dataUploader** and **executeCustomQuery** methods to handle data upload and user query respectively.

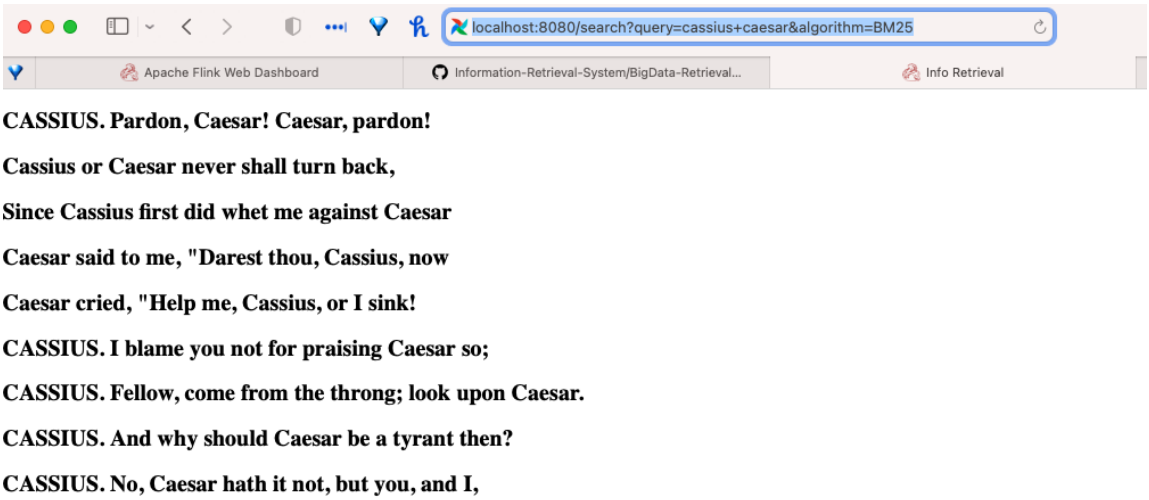

Fig. 10. BM25 Result

5.6 Evaluation

We will consider one same query and compare results for all the three models with same query. Lets consider "cassius caesar" as our sample query. The below table tabulates top 5 documents with maximum scores by all three models. Figures 10, 11, and 12 show the documents set results of each query.

Model	MaxScore 1	MaxScore 2	MaxScore 3	MaxScore 4	MaxScore 5
VSM	7.7017648	6.2905733	5.1362316	4.4626013	4.4626013
TF-IDF	17.2216696	12.5811467	12.5811467	11.7561215	11.7561215
BM25	17.4154728	13.5872518	12.9153547	12.9153547	12.3067778

Table 1. Results table for the query "cassius caesar"

6 CONCLUSION AND FUTURE WORKS

In conclusion we have successfully implemented an Information Retrieval system with the help of Apache Kafka and Flink. During the course of this project we got to dig deeper into creating and working with data streams, learned new frameworks such as Flink and Kafka, and also learned new retrieval algorithms. Although Flink is more powerful as a stream processing framework compared to Spark Streaming, the steep learning curve and limited documentation and support does make Flink challenging to use.

For future work, we would like to directly use Cassandra as a sink and be able to handle multiple sets of data streams sent at different time intervals. To achieve this we would be appending the UUID along with the postings list to Cassandra. We would also like to successfully be able to handle creating the complete postings list with all documents in a single job with a Kafka source.

ACKNOWLEDGMENTS

We would like to thank Prof. Dan Holtby for his constant support throughout the term.

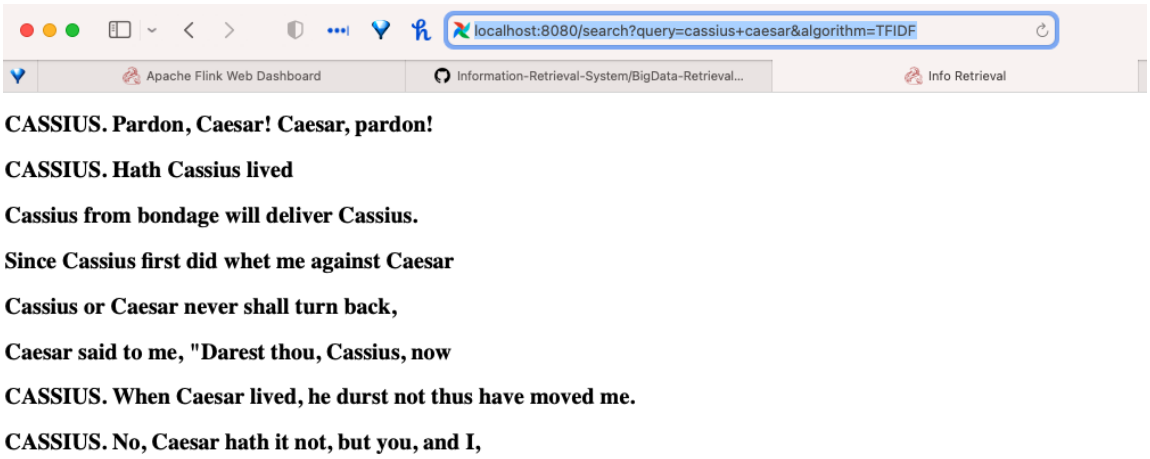

Fig. 11. TF-IDF Result

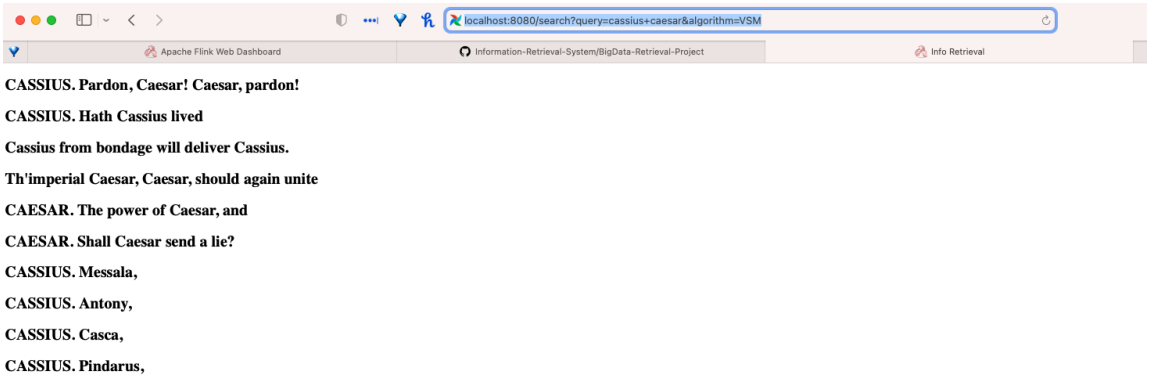

Fig. 12. VSM Result

REFERENCES

[1] Hiteshwar Kumar Azad and Akshay Deepak. 2019. Query expansion techniques for information retrieval: a survey. *Information Processing & Management* 56, 5 (2019), 1698–1735.

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[3] Etienne Chauchot. [n. d.]. tpcds-benchmark-flink/src/main/java/org/example/tpcds/flink/Query3ViaFlinkRowDatastream.java at 9589c7c74e7152badee8400d775b4af7a998e487 · echauchot/tpcds-benchmark-flink. https://github.com/echauchot/tpcds-benchmark-flink/blob/9589c7c74e7152badee8400d775b4af7a998e487/src/main/java/org/example/tpcds/flink/Query3ViaFlinkRowDatastream.java#L258

[4] Manuja DeSilva and Michael Hendrick. 2020. Using Streaming Data and Apache Flink to Infer Energy Consumption. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems.*

[5] Ellen Friedman and Kostas Tzoumas. 2016. *Introduction to Apache Flink: Stream Processing for Real Time and Beyond.* "O'Reilly Media, Inc.".

[6] Qin He. 1999. *A review of clustering algorithms as applied in IR.* Technical Report 6. Graduate School of Library and Information Science, University of Illinois at Urbana-Champaign. 1–33 pages.

[7] Fabian Hueske and Vasiliki Kalavri. 2019. *Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications.* O'Reilly Media.
[8] Asterios Katsifodimos and Sebastian Schelter. 2016. Apache Flink: Stream Analytics at Scale. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. IEEE.
[9] Jay Kreps. 2011. Kafka : a Distributed Messaging System for Log Processing. https://api.semanticscholar.org/CorpusID:18534081
[10] D.L. Lee, Huei Chuang, and K. Seamons. 1997. Document ranking and the vector-space model. *IEEE Software* 14, 2 (1997), 67–75. https://doi.org/10.1109/52.582976
[11] Jimmy Lin and Chris Dyer. 2010. *Data-intensive text processing with MapReduce.* Morgan Claypool.
[12] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval.* Cambridge University Press.
[13] maverick. 2019. Apache Kafka Architecture and Components. https://programmertoday.com/apache-kafka-architecture-and-components/
[14] Dianne Medeiros, Helio Cunha Neto, Martin Andreoni, Claudio Luiz, Magalhães, Natalia Fernandes, Alex Borges, Edelberto Silva, and Diogo Menezes. 2020. A Survey on Data Analysis on Large-Scale Wireless Networks: Online Stream Processing, Trends, and Challenges. *Journal of Internet Services and Applications* (2020). https://doi.org/10.1186/s13174-020-00127-2
[15] Elham Nazari, Mohammad Hasan Shahriari, and Hamed Tabesh. 2019. BigData Analysis in Healthcare: Apache Hadoop, Apache Spark and Apache Flink. *Frontiers in Health Informatics* 8, 1 (2019), 14.
[16] Theofanis P. Raptis and Andrea Passarella. 2023. A Survey on Networked Data Streaming with Apache Kafka. *IEEE Access* (2023).
[17] Abdul Wahid and Kanupriya Kashyap. 2019. Cassandra—A distributed database system: An overview. In *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2018, Volume 1*. 519–526.