SparkCollector: A Distributed Data Collection Framework on top of Spark

Liu, Zhengyuan
zhengyuanliu@ucla.edu

Tang, Weijie
twjeric@ucla.edu

Wu, Wenjun
wenjunwu@ucla.edu

ABSTRACT

SparkCollector[1] is a distributed java framework on top of Apache Spark [2] for collecting data and generating analysis results. By extending the base class, users specify one input seed, such as a url, a point in a map, etc., and three functions:

- Get: how to grab the large scale of data by providing a path;
- Parse: how to parse the raw data to generate intermediate results and next stage's inputs;
- PostProcess: how to process the intermediate results to generate the analysis.

Lots of common data collection problems in reality can be easily expressed by this framework.

Programs written in this framework can be directly submitted to Apache Spark engine and automatically scaled according to the size of raw data. The framework itself would take care of the details of transforming the different data types according to Spark's requirements, passing the parsed results to fetching phase, dynamically collecting the data without blocking the analysis stage, partitioning the computing resources between different data collection jobs to optimize the whole system's performance. This makes it possible for programmers without any Spark's experience to develop distributed data collection by following the framework's routine.

Our implementation of the SparkCollector can run in MaxOS and is scalable for different data characteristics: we can generate real-time analysis for all kinds of data collection tasks. Programmers find this framework relatively easy to follow: we give the demonstration by developing two applications.

[1]`https://github.com/twjeric/UCLA-CS239-Big-Data-Systems`

CS239: Big Data Systems Project Report UCLA

Keywords

Data Collector, Spark, Spark Streaming

1. INTRODUCTION

Within recent decade, we have found lots of job descriptions online closely related to data-purpose collections and analysis, such as web crawler, IOT information gathering, etc., to compute different kinds of derived information, such as most popular websites within a region, best nearby places for dinner, most frequent queries in a given data, etc. Usually, these types of tasks are conceptually straightforward. However, the types of data are enormously different and it requires developers' very good understanding of the distributed programming model, such as Apache Spark [2]. The issues of how to transform the types of inputs according to the Spark's requirements, and handle scalability of the data collection system add too much complexity to the original simple program.

To solve this issue in reality, we design a novel java framework which allows us to express our simple data collection intentions without caring too much the details of how to exactly implement such tasks on top of Spark. And at the same time, the framework can provide very useful scalability handling. Our motivation for this framework is inspired by our recent experience of finding an intern. As students, when we are preparing an interview targeted for these type of jobs, we are thinking can we do something to ease the pain when people are doing related work so that it could make the companies easier to find qualified employees.

The major contributions of our project are a powerful but easy-to-use framework for data collection purpose which enables automatic transforming of different intermediate data types, combined with an initial implementation of such a framework to demonstrate the usages. Two very common usages of this type of framework has been shown very efficient for application developers.

Section 2 describes the basic background of our research and projects. Section 3 is focused on the new architecture of our framework based on Spark's java interface. Section 4 shows the details of our initial implementations, through which we can fully understand the framework's benefits and limitations. We have shown two common usages of our framework in Section 5. Section 6 describes our experiments and demos. Related work and how we get this idea is put in Section 7. Finally, we come to our conclusion in Section 8.

2. BACKGROUND

We chose Spark [6] as our computation engine for data collection. Spark is a cluster computing framework initially proposed by UC Berkeley, aiming to extend MapReduce in iterative algorithms like machine learning, and providing similar scalability and fault tolerance properties. The main abstraction in Spark is that of a resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition.

Spark Streaming [7] is the engine for streaming processing based on the Spark, which is initially developed by UC Berkeley for the implementation for discretized streams (D-Streams). The idea in D-Streams is to structure a streaming computation as a series of stateless, deterministic batch computations on small time intervals. By structuring computations this way, D-Streams make the state at each timestep fully deterministic given the input data, forgoing the need for synchronization protocols, and the dependencies between this state and older data visible at a fine granularity. We used Spark Streaming for processing collected streaming data.

Now Spark is open-sourced by Apache Software Foundation, and we can easily download Apache Spark [2] from its official website and then run Spark on our machines. There are several advantages of Apache Spark. First is speed. Apache Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine, which can run workloads up to 100 times faster than Hadoop in iterative machine algorithms. Second, ease of use. Spark offers over 80 high-level operators that make it easy to build parallel apps. It provides interface for Scala, Python, R, and SQL. Third, generality. Spark provides a general framework for batch processing, streaming processing, SQL query, machine learning and graph processing. Users can easily combine these libraries seamlessly in the same application. In our SparkCollector, we used the Java API of Apache Spark.

3. ARCHITECTURE

This section introduces the architecture of the SparkCollector. SparkCollector is a layer on top of Spark which is specialized for data collection. The key parts in SparkCollector system consist a Data Source, a Data Fetcher, a Data Parser and a Post Processor.

Data Source is the source of the data. It can be any source of data, e.g. Internet, or a local dataset. Based on a collection seed initially provided by the user, the Data Fetcher fetches raw data from the Data Source, and passes the fetched raw data to the Data Parser. Data Parser parses the raw data and extract the data in the format that user wants from the raw data, and passes it to the post processor. In addition, Data Parser generates next batch of data seeds, and passes them as seed stream back to the Data Fetcher. As a result, the Data Fetcher can use these seeds as the next step for fetching data from Data Source. Post Processor processes the formatted data from Data Parser, and give the final aggregated results based on the post process function provided by the user. The operators used in Post Processor can be map, filter, count, sort, etc.

The input of SparkCollector includes a collection seed (or a set of collection seeds) and three functions. The collection seed defines the initial query of the Data Source and the entrance of the data collection process. For example, in a web crawler application, the collection seed could be an initial URL. Besides the collection seed, the user needs to specify three functions that need to use in the collection process: Get, Parse (or Next) and Post-Process. The Get function specifies the way to get the data from the Data Source. The Parse function specifies the way to parse raw fetched data and the way to generate next batch of seeds. The Post-Process function defines the post processing to the data.

4. IMPLEMENTATION

Different kinds of implementation are possible for this type of framework for collecting big data dynamically. The best framework totally depends on the user's preference, working environment, and task characteristics. For instance, due to the different code styles of developers, some implementations may be good for this group of people, yet another for researchers who just want to have a try.

This section targets the simplest implementation to show the possibility of such a powerful framework to save distributed data collection developers' lots of efforts. In our environment:

1. Machines are typically MacBook running MacOS.
2. Java 8 [4] will be used as the framework's language, and JavaScript as the demo language.
3. The version of Apache Spark we use is 2.4.0 [2], but our implementation does not require any fancy features of newer versions of Spark. So it's still available among previous versions.
4. We have implemented all the general processes inside an Interface, and users only need to extend a base class and override their own special functions (Get, Parse, and PostProcess) to their preference.
5. By means of Maven, after extending the only required class, users would get a jar package, which can be directly submitted to Spark.

4.1 Execution

After extending the base entrance class provided by the framework, users can start the program by calling the run function of the Interface. Later on, all the tasks will be handled by the framework itself. When users submit the jar package to the Spark, the following sequence of actions will occur:

1. A specialized server will be created to send the outputs of the system where users define. For example, an Http Server [3] will be set up if the users want to display the results on web browsers by requesting http

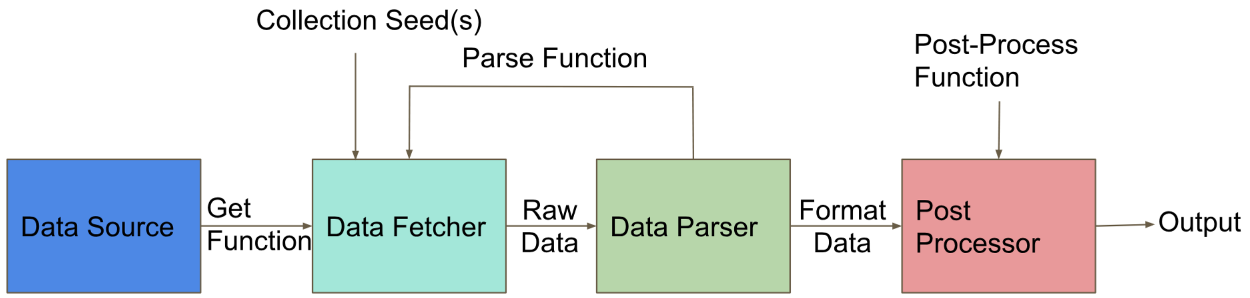


Figure 1: Architecture of SparkCollector

connection. In this case, the server would transform the streaming or batch outputs of the Spark system to user's preferred data type.

2. By calling the run method, spark will be configured by the user's configurations. For instance, whether or not the program will be running on distributed system.

3. A special Receiver will be created to transform the physical input type according to Spark's requirements. Inside the Receiver, users need to let the Spark know how it should retrieve the data from physical places.

4. Start method of SparkContext will be called to start the whole task. During the process, Get, Parse, and PostProcess functions will be called when its event is triggered. After calling Get and Parse function, the framework would pass the intermediate results to the spark system by calling store API of Spark. After transforming the spark's results, PostProcess function will be applied to them.

4.2 Scalability

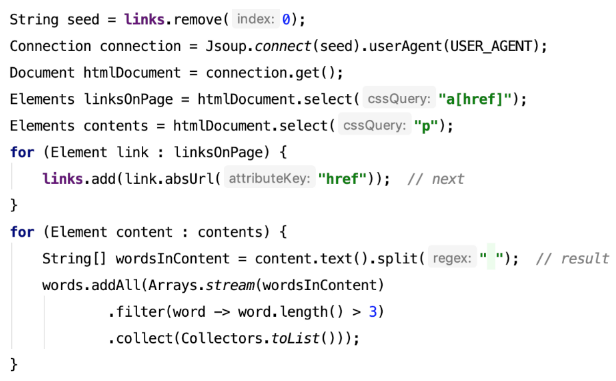
By specifying how many processes, the users can improve the system's throughput however they want. The bottleneck of this type of framework is caused by the master process, because it needs to handle the job's initialization and making the results available for other objectives, for example, building an Http Server. After redesign the order of tasks execution at the master process, we can greatly improve the framework's throughput without blocking any part of system. That is, we build the server first before we actually start the Spark job.

5. APPLICATIONS

We applied our SparkCollector on two real-world applications: web crawler and Place of Interest (POI) searching, which are in different data formats and different collecting ways, but can be unified into one computation model based on the architecture of our SparkCollector.

5.1 Web Crawler

Nowadays web crawlers are widely used by developers to collect various data from Internet, or analyze the contents from a website and the relationship between different websites. A Web crawler starts with a list of URLs to visit. As the crawler visits these URLs, it identifies all the hyperlinks in the page and adds them to the list of URLs to visit, and recursively visits these URLs. In the visiting process, the web crawler can parse and analyze the information on the web pages these URLs refer to. Based on this paradigm of web crawlers, we can easily use SparkCollector to implement a high-efficiency web crawler. We implemented three kinds of web crawlers: get most frequent URLs (URL SparkCollector), get most frequent websites (Website SparkCollector) and get most frequent words (Word SparkCollector).



```
String seed = links.remove( index: 0);
Connection connection = Jsoup.connect(seed).userAgent(USER_AGENT);
Document htmlDocument = connection.get();
Elements linksOnPage = htmlDocument.select( cssQuery: "a[href]");
Elements contents = htmlDocument.select( cssQuery: "p");
for (Element link : linksOnPage) {
    links.add(link.absUrl( attributeKey: "href"));  // next
}
for (Element content : contents) {
    String[] wordsInContent = content.text().split( regex: " ");  // result
    words.addAll(Arrays.stream(wordsInContent)
            .filter(word -> word.length() > 3)
            .collect(Collectors.toList()));
}
```

Figure 2: Code Segment in Word SparkCollector's Get and Parse Function

In the web crawler application, the collection seed is a URL or a set of URLs specified by the user. The Get function is to establish an HTTP connection requested by the crawler, and receive the HTML document from the server. The Parse/Next function is to parse the raw HTML document and extract the data we want and next batch of URLs. The get and parse function of the Word SparkCollector is shown in Figure 2. The Post-Process can be any aggregating operations written by the user. In our demo we returned the top 5 most frequent URLs, top 10 most frequent websites and top 20 most frequent words.

5.2 Place of Interest (POI) Searching

POI (Place of Interest, or Point of Interest), is a set of points with spatial location information and other data that is interested to users, or valuable by developers and researchers. POI usually contains basic information such as name, category, latitude and longitude coordinates, or other detailed information. POI is of great importance in the research and application of Geographic Information System (GIS) and other real-world location-based services (LBS)

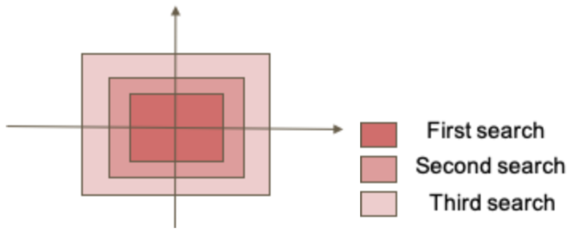
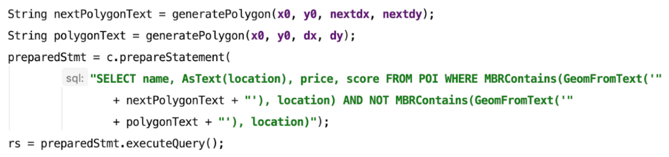


Figure 3: POI Searching Steps



```
String nextPolygonText = generatePolygon(x0, y0, nextdx, nextdy);
String polygonText = generatePolygon(x0, y0, dx, dy);
preparedStmt = c.prepareStatement(
        sql: "SELECT name, AsText(location), price, score FROM POI WHERE MBRContains(GeomFromText('"
                + nextPolygonText + "'), location) AND NOT MBRContains(GeomFromText('"
                + polygonText + "'), location)");
rs = preparedStmt.executeQuery();
```

Figure 4: Code Segment in POI Searching's Get Function

and applications.

The most basic user case related to POI is, given a coordinate (which is typically user's current location), finding the nearby POI in a giving order (by distance, by rating, or by price, etc.). However, because the number of POI in the database is typically huge, it is not practically to return all the results to the user in one time. Instead, we want to return the POI to the user gradually from the near to the distant, and dynamically update the results. Therefore, it naturally can be solved by SparkCollector.

We mimicked this user case by using an simplified offline version of Baidu Map POI database. We used Baidu Map Web API to acquire POI data. POIs' attribute information together with location information are stored in local MySQL databases. Spatial query supporting by MySQL spatial database module is used to search the nearby POIs giving a boundary box. The searching steps is shown in Figure 3. First we search the initial boundary box whose center is the target coordinate, and get the nearest POI information; next we enlarge the width and height of the boundary box and search the points in the current box but not in the previous, and get the next batch of POIs which are farther than the first batch. Keep this step, we can get POI information gradually from the near to the distant.

In POI searching application, the collection seed is a coordinate, which is typically user's current location. The Get function is spatial query in SQL, as shown in Figure 4. The Parse/Next function is to parse raw data from the database and get next scope the boundary box for spatial query based on the number of POI found in the current step. The Post-Process can be any aggregating operations. In our demo we returned the Top 10 Rating POIs.

6. EXPERIMENTS

In this section we describes our experiments and demos. We used a MacBook Pro 2016 as the experiments machine. The operating system is macOS Mojave 10.14.1. The CPU is 2.7 GHz Intel Core i7 and memory is 16GB. The persistent storage is 500GB SSD. Our program is running in the JVM provided by IntelliJ IDEA.

Time (ms)	# of HTML Parsed	# of URLs Parsed
2895	11	1358
5537	21	2013
8546	33	3637
11611	42	4722
14438	49	7708
17434	59	20774
20485	62	21477
23549	67	22035
26455	71	22640
29413	74	23071

Table 1: Performance of URL SparkCollector

Time (ms)	# of SQL Queries	# of POIs Parsed
563	3	74
649	13	553
763	21	975
845	28	1211
1305	58	3398
1455	68	3613
1530	74	3694
1671	81	3834
1891	99	4175

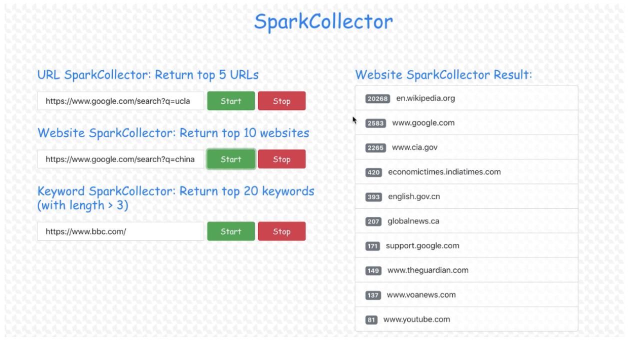
Table 2: Performance of POI SparkCollector

6.1 Performance

We evaluated the performance of the URL SparkCollector and POI Collector in our experiment settings. The results for URL SparkCollector and POI Collector are shown in Table 1 and Table 2 respectively. The time in both tables are elapsing time minus the time spend on the Spark initialization. The initialization time in URL SparkCollector is 3020ms while the initialization time in POI Collector is 2796ms. The results showed that every second URL SparkCollector can parse more than 700 URLs and 2.5 HTML documents. For POI Collector, every second it can emit more than 50 SQL queries and parse more than 2200 POIs.

6.2 Generality and Demos

We build two demos for web crawler application and POI



SparkCollector

URL SparkCollector: Return top 5 URLs

https://www.google.com/search?q=ucla

Website SparkCollector: Return top 10 websites

https://www.google.com/search?q=china

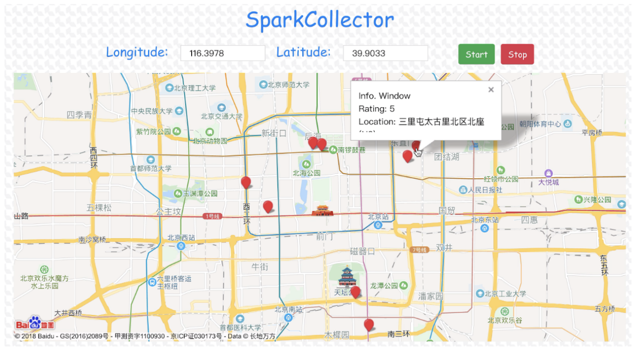
Keyword SparkCollector: Return top 20 keywords (with length > 3)

https://www.bbc.com/

Website SparkCollector Result:

20268	en.wikipedia.org
2583	www.google.com
2265	www.cia.gov
420	economictimes.indiatimes.com
393	english.gov.cn
207	globalnews.ca
171	support.google.com
149	www.theguardian.com
137	www.voanews.com
81	www.youtube.com

Figure 5: Web Crawler Demo



Longitude: 116.3978 Latitude: 39.9033

Info. Window
Rating: 5
Location: 三里屯太古里北区北座

Figure 6: POI Searching Demo

searching application. In Figure 5 we show three types of web crawler application using the SparkCollector. Users can do the tasks of data collection by giving a seed. And after clicking the start button, the results along with their counts will keep updating in the right panel until the stop button is clicked. Similarly, in Figure 6 users can search for POIs by giving the longitude and the latitude of a center point. At first, the top 10 results within a small neighborhood will be displayed. Then the results will keep updating as the search region becomes larger and larger. By clicking the markers, users can check the detailed POI score and location information of each result.

Although web crawler and POI searching are very different data collection problems with different data formats and collecting methods, they can be unified into one computation paradigm based on SparkCollector. The demos show the generality of our SparkCollector.

7. RELATED WORK

SparkCollector is designed as a general-purpose data collector tool. Among the various potential use cases, the web crawler is the most popular one. In general, users start from a list of URLs called seeds, and get all the hyperlinks on those web pages. Then they recursively crawl web pages according to a set of policies. This is typically done for web indexing.

Apache Nutch [1] is a well matured, production ready web crawler. Started from the year of 2002, it became one of Apache Top-Level projects in 2010. Since then, more and more researchers use it for web crawling tasks. It can be run either on a single machine or on Apache Hadoop. Nutch provides extensible interfaces and pluggable indexing for Apache Solr, which is an open source full text search framework.

Recently, the IRDS group at USC developed evolved Apache Nutch on Spark, called Sparkler [5]. Like Apache Nutch, Sparkler is an extensible, highly scalable, and high-performance web crawler. What is different is that it runs on Apache Spark Cluster, thus it takes advantage of the caching and fault tolerance capability of Apache Spark. It also supports real-time analysis and instant visualizations.

8. CONCLUSION

Our SparkCollector framework has been successfully shown by WebCrawler and POI application demos. We contribute the framework's success to several reasons. First, the framework is quite easy to use without any knowledge of how Spark works, because it hides most of the critical parts of Spark's operations. The person who develops the two applications hasn't used Spark before. Second, most of data collection related problems can be easily expressed using our framework. For instance, by just providing Parse and Post-Process functions we can directly tell the framework to operate the data however we want. Third, we have taken into account the scalability issue in our framework, which reduces the users concerns of scaling in the future.

We also gained lots of experiences from this research and developing process. First, modularization can help us develop the framework gradually, by adding more and more advanced features step by step instead of stuffing the whole system at the beginning. Second, abstracting the data collection related tasks is hard, because each application's usage is very different, but by restricting the framework's features we can generalize the common parts to some extent. Third, we can gain lots of inspirations and the critical features of the framework by implementing the demos at the same.

9. REFERENCES

[1] Apache nutch. http://nutch.apache.org/.
[2] Apache spark. https://spark.apache.org/.
[3] Http server. https://docs.oracle.com/javase/8/docs/jre/api/net/httpserver/spec/com/sun/net/httpserver/HttpServer.html.
[4] Java 8. https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.
[5] Sparkler. http://irds.usc.edu/sparkler/.
[6] M. Zaharia. Spark: Cluster computing with working sets. *HotCloud*, 10.10-10 (2010): 95.
[7] M. Zaharia. Discretized streams: Fault-tolerant streaming computation at scale. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ACM, 2013.