Master Thesis: Implemetation of Centroid Decomposition Algorithm on Big Data Platforms—Apache Spark vs. Apache Flink

Qian Liu
Master of Science of Computer Science and Applied Mathematics
University of Bern
qian.liu@students.unibe.ch

Supervisors:
Dr. Mourad Khayati, Prof. Dr. Philippe Cudré-Mauroux
eXascale Infolab, Department of Informatics,
University of Fribourg

February 21, 2016

Abstract

The Centroid Decomposition (CD) algorithm is the approximation of the Singular Value Decomposition (SVD) algorithm, which is one of the most used matrix decomposition techniques to deal with real world data analysis tasks. CD algorithm is based on a greedy algorithm, termed the Scalable Sign Vector (SSV), that efficiently determines vectors that are consisted of 1s and -1s as elements, called sign vectors. CD algorithm is generally applied for data analysis tasks that involve long time series, i.e. where the number of rows (observations) is much larger than the number of columns (time series).

The goal of this thesis is to implement the CD algorithm on two Big Data platforms, i.e., Apache Spark and Apache Flink. The proposed implementation compares two different data structures for both platforms. The first data structure is the per-element data structure, which distributively transforms the matrix based on every single element. The second data structure, the per-vector data structure, executes every transformation on the basis of each row or column vector.

We empirically evaluate the efficiency of the non-streamed Spark and Flink CD implementations respectively. To simulate the streams of time series, we use Apache Kafka to periodically produce new matrix data to a broker and Spark Streaming and Flink Data Streaming to regularly fetch the data and run the CD algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Matrix decomposition techniques are widely applied for time series data in a number of real world applications, such as data prediction, recommender systems, image compression, recovery of missing values, stocks, etc.

The Centroid Decomposition (CD) algorithm was initially introduced as an approximation of the Singular Value Decomposition (SVD)[1]. It performs a decomposition of an input matrix $\mathbf{X}$ into the product of two matrices, i.e. $\mathbf{X} = \mathbf{L} \times \mathbf{R}^T$, where $\mathbf{L}$ is the Loading matrix and $\mathbf{R}$ is the Relevance matrix ($\mathbf{R}^T$ denotes the Transpose of $\mathbf{R}$). Every Loading and Relevance Vector is determined by a maximal Centroid Value, i.e., $\max \|\mathbf{X}^T \times Z\|$, which equals to the norm of the product between the transpose of the input matrix $\mathbf{X}$ and the sign vector Z consisting of 1s and -1s. Therefore, finding the maximal sign vector Z that maximizes the centroid value is the main part of CD algorithm.

Three approaches to find the maximal sign vector, Z, have been proposed in the literature. The first one enumerates all possible sign vectors and chooses the one which maximizes the centroid value[2]. This approach has linear space complexity since no data structures other than the input matrix are needed, but exponential runtime complexity. The second approach introduced by Chu and Funderlic[3] is more efficient than the first one, has quadratic runtime complexity, but has quadratic space complexity. The third one proposed by Khayati et al.[4], has also quadratic runtime cost (worst case) but linear space complexity. In this thesis we adopt Khayati's approach since it is the most space efficient and scalable one.

Based on the fact that the most efficient algorithm to compute the CD algorithm has quadratic runtime complexity and thus is hard to scale to large datasets, we propose in this thesis to distribute the computation of CD algorithm.

The remainder of this thesis is as follows. In Chapter 2, basic concepts including CD algorithm are introduced. In Chapter 3, the details about respectively the two strategies' implementations of Spark and Flink are described. Chapter 4 describes the empirical evaluation and the different experiments we ran. Chapter 5 summarizes the thesis and points out future work that could extend the current thesis.

Chapter 2

Background

This chapter describes the main concepts used throughout this thesis. The details of the two algorithms implemented in this work, i.e., CD algorithm and SSV algorithm, are also illustrated.

2.1 Hadoop Framework

The Apache Hadoop software library is a framework that allows to perform distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single server to thousands of machines, each offering local computation and storage[5].

Hadoop Distributed File System is designed to store large data across multiple distributed machines, typically inside a cluster system with large number of machines. The reliability is inherent for HDFS storage.

YARN is a framework for job scheduling and cluster resource management. Together with its counterpart Apache Mesos, they are both used to facilitate the management and coordination of distributed machines[6].

2.2 Spark

Apache Hadoop platform is not suitable to apply for algorithms that involve iterative tasks. In fact, as shown in Figure 2.1, though Apache Hadoop provides an abstraction for accessing computational resources, it lacks abstractions that allow access to the clusters' main memory[7]. The only way that a user can share results among multiple map reduce tasks is by writing them to HDFS. However, to preserve the fault tolerance, HDFS replicates the written files among nodes yielding an overhead in disk I/Os.

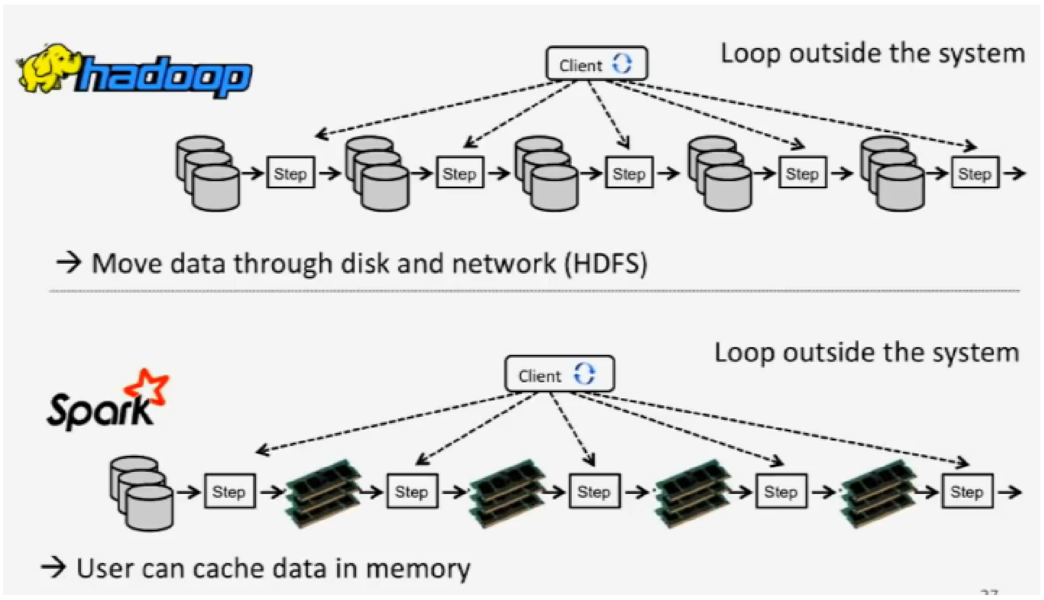


Figure 2.1: Hadoop vs. Spark Iterations[1]

Zaharia et al.[8] introduced a new distributed framework for iterative algorithms called Spark[9]. It provides a computational framework that gives an abstraction to access the distributed main memory, i.e., the Resilient Distributed Datasets (RDDs). RDDs are partitioned collections of objects which are distributed in the main memory of a cluster (as in Figure 2.1). They are resilient meaning that they are fault tolerant. RDDs handle the map and reduce operations as chainable coarsed-grained transformations which means that the data get read once from the HDFS and all subsequent transformations will take place in memory. Besides, transformations are lazy, i.e., they will be executed only when an action is requested.

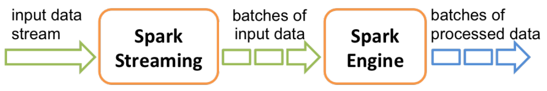
2.3 Flink

Apache Flink[10][11] (Stratosphere[12]) is a general-purpose data processing framework. It is a top level project of the Apache Software Foundation (ASF) and has a wide field of application for dozens of big data scenarios. The main difference between Flink and Spark is that the former takes a declarative approach that is quite similar to the optimization methods of typical Relational DBMS applies where the latter does not. In this declarative approach users

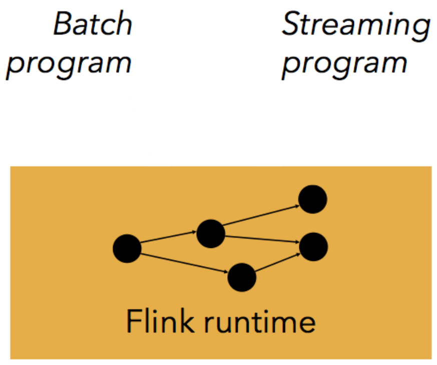
[1]Source:http://www.nextplatform.com/2015/02/22/flink-sparks-next-wave-of-distributed-data-processing/

don't need to write down painstaking details about how the data is to be processed, but rather to describe in a higher level what they want to compute. This is similar to the transformations from “Logical Execution Plans” to optimized “Physical Execution Plans” in RDBMS for SQL queries. Flink extends this approach to scalable data processing with many dimensions of optimization potentials, in particular, optimizations aiming to minimize the amount of data shuffling. For example, when users write down a data processing pipeline in Flink, the Flink Data Processing Runtime will further optimize the pipeline and turn it into a physical representation by using reordering the operations, and selecting the appropriate algorithms yielding the best performance.

Another difference between the two systems lies on the way they deal with streaming. In fact, data streams are processed in Flink Streaming as true streams, i.e., data elements are immediately pipelined through a streaming programme as soon as they arrive (see Figure 2.2(b)). This allows to perform flexible window operations on streams[13].Besides, Flink uses one common runtime for data streaming applications and batch processing applications, which is called the Kappa Architecture[14][15]. Batch processing applications run efficiently as special cases of stream processing applications (As shown in Figure 2.2(b)).While in Spark, data streams are processed as micro batches (see Figure 2.2(a)). And batch processing applications and stream processing applications are separately processed, the Lambda Architecture[16].



(a) Spark Streaming.



(b) Flink Streaming.

Figure 2.2: Spark vs. Flink Streaming

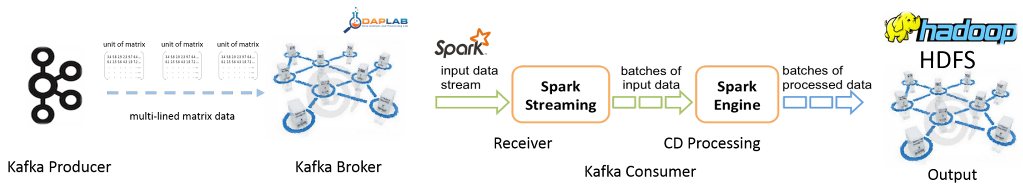
2.4 Apache Kafka

Apache Kafka[17] is a distributed, partitioned and replicated commit log service, which provides the functionality of a messaging system. Kafka maintains feeds of messages in categories called topics. The producers publish messages to a Kafka topic/kafka topics. The consumers subscribe to a topic/topics and process the feed of published messages. In this thesis, Kafka is used as a data feed to periodically generate multi-lined matrix data. Each of the Kafka messages is consisted of a unit of time slots from the long time series and is several columns of the sending matrix. The consumer is the Spark Streaming framework and the Flink Data Streaming runtime, which regularly receive the matrix data. After receiving the matrix data, the Spark streamingContext and the Flink StreamExecutionEnvironment respectively run the Centroid Decomposition algorithms on the input matrix data. Figure 2.3 gives an overview of the architecture for our Kafka-enabled long time series CD processing programme. In it, Kafka Producer (one of the Daplab cluster machine) periodically generates time unit (in our case, it represents 10 columns of the input matrix) matrix message. Afterwards, it sends the message to the Broker, i.e. the distributed message queue on the Daplab cluster. The Spark-based CD consumer (Figure 2.3(a)) runs the micro-batched CD algorithm every time when the time threshhold reaches, which is pointed when initializing the Spark DStream object. By contrast, the Flink-based CD consumer (Figure 2.3(b)) runs the CD algorithm whenever there is a new message generated on the Kafka Broker, which means it processes the matrix data as a continuous data flow without identification of micro-batches. The complete procedures for both CD consumers are as below:

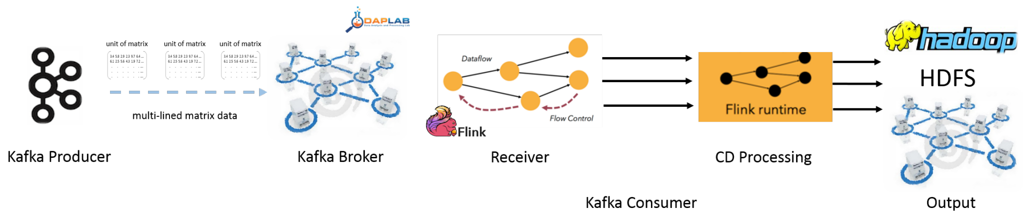
1) There is 'message receiver' in both of Spark-based and Flink-based consumer programmes. It is responsible for fetching Kafka messages from Kafka Broker.

2) After the receiver obtains new message, the CD method is invoked. During the period of CD computation on the incoming matrix data, the receivers in both the Spark-based and Flink-based consumer programmes can not fetch new matrix data since current CD computation has not been accomplished. Therefore, in Spark, when DStream object is initialized, it has to identify a time interval for the CD computation, whereas, in Flink although the time interval can not be controlled by the consumer programme, it should be identified by the Kafka producer programme when it sends the messages.

3) After one CD computation is completed, the resulted Loading and Relevance matrices are written into HDFS text files on Daplab cluster.



(a) Spark



(b) Flink.

Figure 2.3: CD processing of long time Series

2.5 Centroid Decomposition Algorithm

In this section, we describe in detail the CD algorithm and its embedded SSV procedure. The latter iteratively computes a sign vector Z and is the most computationally expensive step in CD algorithm.

2.5.1 CD Algorithm

Algorithm 1 describes the decomposition performed by CD technique. It iteratively computes the Loading matrix, **L** and the Relevance matrix, **R**, once per column. At each iteration i, the procedure $ScalabeSignVector(X, n, m)$ determines the sign vector Z that yields the maximal $\|\mathbf{X}^T \cdot Z\|$ (where $\|\mathbf{X}^T \cdot Z\|$ is the norm of $m \times 1$ product vector). Then, the centroid column vector C_{*i} is obtained. Finally, vectors L_{*i} and R_{*i} are respectively computed.

Algorithm 1: CD(**X**, n, m)

```
   Input: $n \times m$ matrix $\mathbf{X}$
   Output: $\mathbf{L}$, $\mathbf{R}$
1  $L = R = []$;
2  for $i = 1$ to $m$ do
3      $Z = ScalableSignVector(\mathbf{X}, n, m)$;
4      $C_{*i} = \mathbf{X}^T \cdot Z$;
5      $R_{*i} = \frac{C_{*i}}{\|C_{*i}\|}$;
6      $\mathbf{R} = Append(\mathbf{R}, R_{*i})$;
7      $L_{*i} = \mathbf{X} \cdot R_{*i}$;
8      $\mathbf{L} = Append(\mathbf{L}, L_{*i})$;
9      $\mathbf{X} := \mathbf{X} - L_{*i} \cdot R_{*i}^T$;
10 return $\mathbf{L}$, $\mathbf{R}$
```

Example 1 *Consider a 4×3 matrix $X = \{X_1, X_2, X_3\}$ which is consisted of three time series vectors, i.e., $X_1 = \{3, 2, 5, -2\}$, $X_2 = \{-2, 1, -3, 0\}$ and $X_3 = \{-1, -4, 1, -2\}$. CD algorithm decomposes* **X** *by finding the Loading and the Relevance vectors as shown in Figure 2.4.*

$$X = \begin{bmatrix} 3 & -2 & -1 \\ 2 & 1 & -4 \\ 5 & -3 & 1 \\ -2 & 0 & -2 \end{bmatrix}$$

$$CD(X) = \underbrace{\begin{bmatrix} 3.592 & 0.296 & 1.005 \\ 2.186 & 3.900 & -1.005 \\ 5.466 & -2.046 & 0.968 \\ -1.562 & 2.151 & 0.968 \end{bmatrix}}_{L}, \underbrace{\begin{bmatrix} 0.937 & -0.099 & -0.335 \\ -0.312 & 0.192 & -0.930 \\ -0.156 & -0.977 & -0.149 \end{bmatrix}}_{R}$$

such that

$$X = \begin{bmatrix} 3 & -2 & -1 \\ 2 & 1 & -4 \\ 5 & -3 & 1 \\ -2 & 0 & -2 \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} 3.592 & 0.296 & 1.005 \\ 2.186 & 3.900 & -1.005 \\ 5.466 & -2.046 & 0.968 \\ -1.562 & 2.151 & 0.968 \end{bmatrix}}_{\mathbf{L}} \times \underbrace{\begin{bmatrix} 0.937 & -0.312 & -0.156 \\ -0.099 & 0.192 & -0.976 \\ -0.335 & -0.930 & -0.149 \end{bmatrix}}_{\mathbf{R}^T}$$

Figure 2.4: Example 1 of Centroid Decomposition

2.5.2 SSV Algorithm

The Scalable Sign Vector (SSV) algorithm is described in Algorithm 2. On single machine, it has quadratic runtime complexity but requires only linear space. We expect to significantly reduce the quadratic runtime complexity of SSV algorithm when execute it in the distributed environment, however, both the SSV and CD algorithm are based on incremental iterations, therefore, in Spark, the improvement of time complexity for both SSV and CD algorithm is less than in Flink, since Flink has native closed-loop iteration operators to optimize the executions[18]. Later in Chapter 4, the reduction of time consumptions of SSV and CD algorithm in the distributed environment will be illustrated in detail.

```
Algorithm 2: SSV(X, n, m)
   Input: $n \times m$ matrix $\mathbf{X}$
   Output: maximizing sign vector $Z^T = [z_1, \ldots, z_n]$
 1 $pos = 0$;
 2 repeat
       // Change sign
 3     if $pos = 0$ then $Z^T = [1, \ldots, 1]$;
 4     else change the sign of $z_{pos}$;
       // Determine $S$ and $V$
 5     $S = \sum_{i=1}^{n}(z_i \times (X_{i*})^T)$;
 6     $V = []$;
 7     for $i = 1$ to $n$ do
 8         $v_i = z_i \times (z_i \times X_{i*} \cdot S - X_{i*} \cdot (X_{i*})^T)$;
 9         Insert $v_i$ in $V$;
       // Search next element
10     $val = 0, pos = 0$;
11     for $i = 1$ to $n$ do
12         if $(z_i \times v_i < 0)$ then
13             if $|v_i| > |val|$ then
14                 $val = v_i$;
15                 $pos = i$;
16 until $pos = 0$;
17 return $Z$;
```

The SSV algorithm calculates V from row vectors of **X**, one row per time: from the computation of intermediate vector S to computation of individual elements of V, searching for the index (pos) of the element $v_i \in V$ with the largest absolute value where v_i and $z_i \in Z$ have different signs, i.e. $z_i \times v_i < 0$. If such an element exists, the sign of z_i is changed. A new vector V is computed, which is different from the vector in the previous iteration due to the sign change of z_i. The iteration terminates when the signs of all corresponding elements in V and Z are the same. The vector Z in the final iteration is the maximizing sign vector that maximizes $Z^T \cdot V$. The SSV algorithm terminates with at most n iterations and with only $O(n)$ space complexity. In the worst case, the sign of each element of Z is changed.

Example 2 *Figure 2.5 illustrates the complete procedures of SSV Algorithm. It uses the input matrix* **X** *introduced in Example 1 as follows*

$$\mathbf{X} = \begin{bmatrix} 3 & -2 & -1 \\ 2 & 1 & -4 \\ 5 & -3 & 1 \\ -2 & 0 & -2 \end{bmatrix}$$

The Sign Vector Z is initialized to $Z = \{1, 1, 1, 1\}^T$, *and S and V are computed as shown in Figure 2.5:*

$$S = \begin{bmatrix} 3 \\ -2 \\ -1 \end{bmatrix} + \begin{bmatrix} 2 \\ 1 \\ -4 \end{bmatrix} + \begin{bmatrix} 5 \\ -3 \\ 1 \end{bmatrix} + \begin{bmatrix} -2 \\ 0 \\ -2 \end{bmatrix} = \begin{bmatrix} 8 \\ -4 \\ -6 \end{bmatrix}$$

$$v_1 = [3 \;\; -2 \;\; -1] \times \begin{bmatrix} 8 \\ -4 \\ -6 \end{bmatrix} - [3 \;\; -2 \;\; -1] \times \begin{bmatrix} 3 \\ -2 \\ -1 \end{bmatrix} = 24$$

$$v_2 = [2 \; 1 \;\; -4] \times \begin{bmatrix} 8 \\ -4 \\ -6 \end{bmatrix} - [2 \; 1 \;\; -4] \times \begin{bmatrix} 2 \\ 1 \\ -4 \end{bmatrix} = 15$$

$$v_3 = [5 \;\; -3 \; 1] \times \begin{bmatrix} 8 \\ -4 \\ -6 \end{bmatrix} - [5 \;\; -3 \; 1] \times \begin{bmatrix} 5 \\ -3 \\ 1 \end{bmatrix} = 11$$

$$v_4 = [-2 \; 0 \;\; -2] \times \begin{bmatrix} 8 \\ -4 \\ -6 \end{bmatrix} - [-2 \; 0 \;\; -2] \times \begin{bmatrix} -2 \\ 0 \\ -2 \end{bmatrix} = -12$$

i.e

$$Z^{(1)} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \;\; \textit{and } V^{(1)} = \begin{bmatrix} 24 \\ 15 \\ 11 \\ -12 \end{bmatrix}$$

Figure 2.5: Illustration of SSV Algorithm Break-Down

Only one element of $Z^{(1)}$ has a different sign from the corresponding element in $V^{(1)}$. Therefore, the index of the element v_i in $V^{(1)}$ with the largest absolute value is $pos = 4$. In the next iteration, the element z_4 in $Z^{(1)}$ is changed with different sign to -1, and the new Sign Vector $Z^{(2)}$ is used to compute $V^{(2)}$. Similar to the computation of previous iteration, we get

$$Z^{(2)} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} \textit{ and } V^{(2)} = \begin{bmatrix} 32 \\ 7 \\ 35 \\ -12 \end{bmatrix}$$

Since all corresponding elements in $Z^{(2)}$ and $V^{(2)}$ have the same sign, hence, the SSV Algorithm terminates and returns $Z^{(2)}$ as the maximizing Sign Vector which maximizes $Z^T \cdot V$.

Chapter 3

Implementation

This chapter describes the implementation part of this thesis. First, we define two types of in-memory data structures that we apply to manipulate matrix data. Afterwards, we describe in detail the implementation of the long time series Kafka and Spark/Flink Streaming CD processing solutions.

The proposed implementation takes full advantages of Spark and Flink API and splits CD algorithm into four separate packages:

1) world.clq.CD.spark/flink.core
2) world.clq.CD.spark/flink.matrixoperations
3) world.clq.CD.spark/flink.validation
4) world.clq.CD.spark/flink.streaming (used only for streaming processing)

The first package contains the implementation of CD and SSV algorithms, and includes the job submission codes on SparkContext/Flink ExecutionEnvironment. The second package is the basic matrix operations implemented by Spark/Flink API. The third package is a validation programme to check the correctness of the final results from CD algorithm, i.e. to check the equality $\mathbf{X} = \mathbf{L} \cdot \mathbf{R}^T$ with five fractional digits accuracy. The last package is used only for the Spark/Flink streaming processsing as the consumers of the Kafka messages.

3.1 Data Representation

In this thesis, we use two different in-memory data structures for matrix manipulations, which are respectively related to two different non-streaming implementation strategies. The two data structures are:

1) `JavaPairRDD<String, Double>/DataSet<Tuple2<String, Double>>`

2) `JavaPairRDD<String, Double[]>/DataSet<Tuple2<String, Double[]>>`

where the first data structure is matrix element based and the second data structure is matrix row vector based.

In Chapter 4, we will illustrate that the vector based in-memory data structure has better execution performance than the element based in-memory data structure. This scalability results from the significant reduction of the amount of join, grouping transformations yielding the reduction of the number of data shuffling. For example, let's consider the basic matrix multiplication operation between two matrices **A** and **B**. If the multiplication is implemented by the element based structure, then firstly we need to group matrix **A** by row and group matrix **B** by column. Afterwards, we need to collect matrix **B**, and transform matrix **A** to execute a vector dot product with every column vector of **B**. Whereas, if we implement the matrix multiplication operation with the vector based structure, then we only need to collect matrix B and a direct vector dot product operation is enough will be enough to get the correct result.

3.2 CD Processing

Using the above non-streaming implementation of CD algorithm, Spark Streaming framework and Flink Data Streaming runtime as Kafka Consumers, we further implement two long time series CD processing programmes, i.e. two separate CD processing programmes that can handle continuously generated input matrices.

1. The Spark streaming based CD processing programme and
2. The Flink data streaming based CD processing programme

Each of these streaming-based CD processing Programme is split into three modules, the Producer module, the Broker module and the Consumer module.

3.2.1 Producer and Broker

The Producer module for both of the two streaming-based CD Processing Programmes is same and is implemented by Apache Kafka Producer API, the KafkaProducer class. For the Broker module, we use the existing Daplab

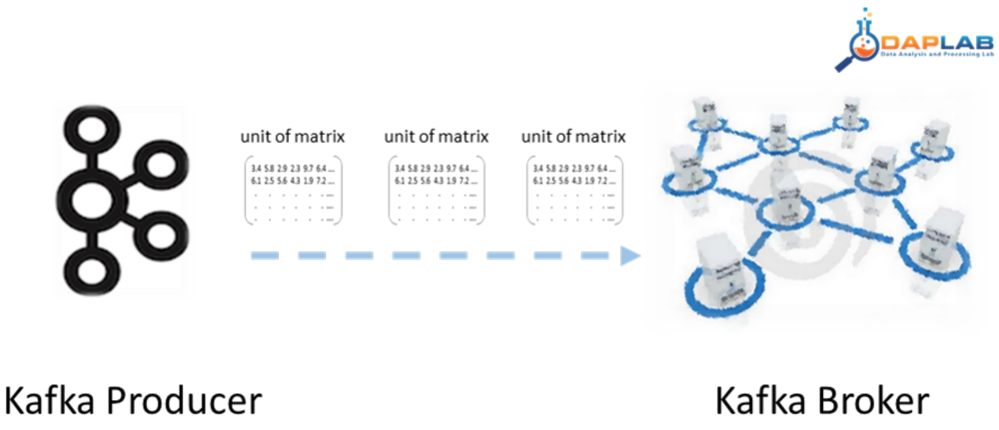


Figure 3.1: Producer and Broker of CD Processing Programmes

Hadoop & Kafka cluster for both of the two streaming-based CD processing Programmes [19].

Figure 3.1 illustrates that the producer periodically sends multi-lined matrix data to the Topics on the Broker. Each of the messages represents one unit of time slots in the long time series and several columns of the matrix (in the tests, we parameterize it as 10 columns). To consider that the Spark Streaming framework consumes all the new messages within every time slice, and Flink Data Streaming runtime continuously pulls the incoming matrix data whenever there is a new one on the Broker, the producer has to leave enough time (typical CD batch processing time on different scales of matrices as shown in Table 3.1) for the consumers to accomplish the CD processing for the received matrix data.

In what follows, we consider the example of an input matrix with relatively small size for the CD processing (since too big matrix size requires large quantity of time for each of the CD processing). Table 3.1 presents some of the time consumptions for the non-streaming CD Processing.

Based on Table 3.1, considering that using enough big input matrix as well as completing computation within an reasonable time period, we take the magnitude of 500×10 for all the tests. Hence, the producer should send 1 message to a topic every 50 minutes, considering CD processing time, streaming initialization time and other cluster coordination time. For each message, the content is the Comma Separated Values (CSV), which includes 500 rows, 10 columns matrix records. The values for each record is generated by a independent and identically distributed (i.i.d.) random double value

[1]All the processing times are computed on the cluster with 6 executors and 4 cores for each of the executors

Matrix Size	Processing Time(s)[1]	
	Spark	Flink
100×10	248	240
500×5	945	930
500×10	1927	1900
1000×10	7300	7200
10000×10	33800	32850
100000×10	261600	259980

Table 3.1: CD Batch Processing Time Consumption

generator. To improve the system throughput, the producer simultaneously sends 5 messages$(5*(500\times10)matrices)$ to five topics per time, which means for every 40 minutes the producer sends 5 messages (totally 2500 rows in five topics), one message per topic to the Broker. The total rows of the matrix is parameterized when the Producer program is started. If the given rows for sending are bigger than 2500, the Producer will firstly divide the given rows by 2500, and then distribute them to the 5 Topics in a Round Robin fashion and to several times. The number of columns (noc) of sending matrices is also parameterized when the Producer programme is started. It controls the time slots in the long time series. If the (noc) is more than ten, then we repeat the previous sending procedures by $(noc/10)$ rounds, e.g., we need to send a 5000 × 20 matrix. On the Broker we have five Topics from CD1 to CD5. The Producer program works as follows:

1) dividing 5000 by 2500 gives 2 times without remainder, dividing 20(noc) by 10 gets 2 rounds without remainder.

2) First round, sending Message<[1 to 500] [1 to 10] [double]> to CD1, Message<[501 to 1000] [1 to 10] [double]> to CD2 ...... Message<[2001 to 2500] [1 to 10] [double]> to CD5, after sending, wait for 40 minutes.

3) sending Message<[2501 to 3000] [1 to 10] [double]> to CD1, Message<[3001 to 3500] [1 to 10] [double]> to CD2 ...... Message<[4501 to 5000] [1 to 10] [double]> to CD5, after sending, wait for 40 minutes.

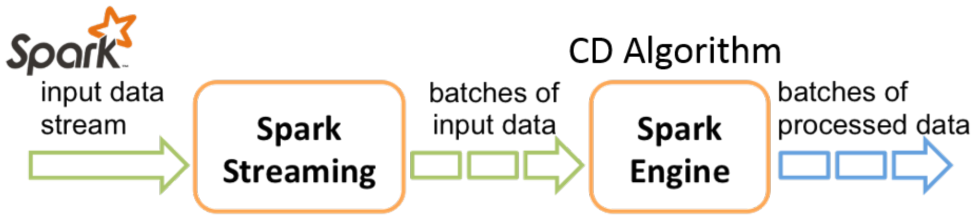
4) Second round, sending Message<[1 to 500] [11 to 20] [double]> to CD1, Message<[501 to 1000] [11 to 20] [double]> to CD2 ...... Message<[2001 to 2500] [11 to 20] [double]> to CD5, after sending, wait for 40 minutes.

5) sending Message<[2501 to 3000] [11 to 20] [double]> to CD1, Message<[3001 to 3500] [11 to 20] [double]> to CD2 ...... Message<[4501

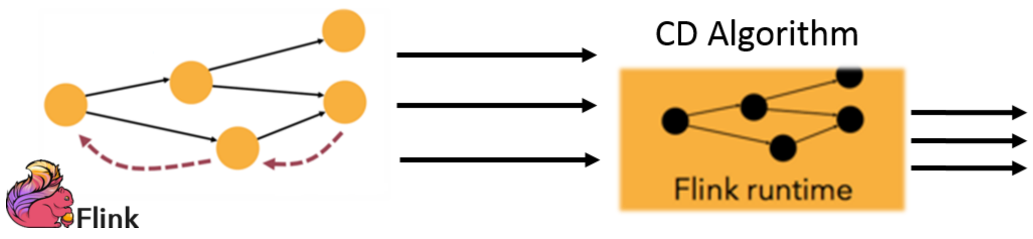
to 5000] [11 to 20] [double]> to CD5, the program terminates.

3.2.2 Consumers

This section illustrates the Consumer module for the two streaming-based CD Processing Programmes shown in Figure 3.2.



(a) Spark Consumers.



(b) Flink Consumers.

Figure 3.2: Consumers of CD processing Programs

The complete Consumer implementation code is furtherly separated into three sub-modules, the 'Message Receiver' module, the 'CD Processing' module and the 'HDFS Output' module. Seeing that the differences between Spark Streaming and Flink Data Streaming, there exist some small logical discrepancies between the two implementations of the 'Message Receiver'. In Spark Streaming framework, the DStream object is micro-batch based, therefore the time interval for each batch can be set. Each of the time interval is used for the 'CD processing' on the received 500×10 matrix. Whereas in Flink Data Streaming framework, taking account of the continuous streaming characteristic of DataStream object, it is not necessary to set the time interval on the Consumer side. Whenever the Producer sends data to the Broker, the Flink implemented 'Message Receiver' will start to receive. Therefore every CD execution on the 500×10 matrix is controlled by the Producer.

There are two main problems for the Spark micro-batch based stream processing framework. One is the back pressure problem, the other is the data out-of-order problem. The back pressure problem occurs when the volume of events coming across a stream is more than the stream processing engine can handle. The data out-of-order problem states that in a micro-batch based streaming processing framework it is more difficult to know if events arrived out of order or not. However, In Spark version 1.5, there have been changes that enable more dynamic ingestion rate capabilities and make back pressure less problematic. In addition, more work has been performed to enable user-defined time extraction functions. This enables developers to check event time against events already processed.

In our case, the incoming 500×10 matrices are independent from each other, which means no data out-of-order problem for our CD processing algorithm. And the back pressure problem is resolved by the time interval definition of DStream object in Spark. In order to increase the throughput of both Consumer programmes, a multi-threaded execution pool is adopted. This adoption enables five 'CD processing' tasks to be simultaneously executed on the Cluster, which means that both of the Consumer programmes can handle a 2500×10 input matrix during each of the 50 minutes time intervals.

3.3 Incremental Computation of V

In Algorithm 2, vector V is computed as follows:

```
**for** $i = 1$ **to** $n$ **do**
    $v_i = z_i \times (z_i \times X_{i*} \cdot S - X_{i*} \cdot (X_{i*})^T)$
```

For the sake of comparison, we implement another incremental method to compute V, as follows:

$$V^k = V^{k-1} - 2 \cdot \begin{bmatrix} X_{*1} \cdot X_{pos}^T \\ X_{*2} \cdot X_{pos}^T \\ X_{*3} \cdot X_{pos}^T \\ \vdots \\ X_{*n} \cdot X_{pos}^T \end{bmatrix}$$

Where X_{*i} is the i-th vector of the input matrix $\mathbf{X}$, *pos* is the position of the element in Z that has been changed at iteration k.

In the incremental method, except the first round computation (V^1), all the other iterations (V^k) for V computation are based on the result from preceding iteration (V^{k-1}). We expect that the incremental method of V computation has better performance than the method adopted in Algorithm 2, since every subsequent V computation of the alternative incremental method is based on the existing result from preceding iteration except the first round, which means less matrix operations than the original V computation method.

In Chapter 4, the performance difference between the two methods is evaluated.

Chapter 4

Empirical Evaluation

Chapter 4 presents the test results of the experiments and their interpretations. The experiments have been performed on the Daplab YARN cluster (Hortonworks Data Platform(HDP) 2.3.2)[19] composed of 26 nodes with per node from 32GB to 128GB available memory. For the software platforms, we adopted Java SE 1.7, Apache Spark 1.4.1, Apache Flink 0.9.1, and Apache Kafka 0.8.2.2. The test suite is split into 5 categories listed as follows:

1. Scalability with large matrices.
2. Scalability with hardware provisions.
3. Performance impact of value properties of matrix data.
4. Algorithms Break-down.
5. Difference between V-computation strategies.
6. Comparison between two long time series CD consumers.

4.1 Scalability with Large Matrices[1]

In order to evaluate the scalability of the CD algorithm with large order-of-magnitude of matrices, we compare different sizes of matrices as follows:

- varying # rows: 10×10, 100×10, $1'000 \times 10$, $10'000 \times 10$ and $100'000 \times 10$, $1'000'000 \times 10$.

[1]All the experiments in this section are executed on the YARN cluster with 6 nodes (4 cores, 20G memory for each node). Later in next section, we will see that this configuration has best performance for both Spark implemented and Flink implemented CD algorithms.

- varying # columns: 10×10, 10×100, $10 \times 1'000$ and $10 \times 10'000$, $10 \times 100'000$, $10 \times 1'000'000$.
- varying # rows & columns: 10×10, 100×100, $1'000 \times 1'000$, $10'000 \times 10'000$, $100'000 \times 100'000$, $1'000'000 \times 1'000'000$

Table 4.1 shows the time costs for the matrices with different rows and Table 4.2 shows the time costs for the column increased matrices. Both of the two tables are based on the vector-based CD algorithm. For the element-based CD algorithm, the time costs are illustrated in Table 4.3 and Table 4.4, followed by Table 4.5 and Table 4.6, increase of both rows and columns.

Matrix Size	Processing Time(s)	
	Spark	Flink
10×10	71	67
100×10	248	240
$1'000 \times 10$	7'300	7'200
$10'000 \times 10$	33'800	33'200
$100'000 \times 10$	261'600	260'380
$1'000'000 \times 10$	2'559'600	2'557'480

Table 4.1: Time Costs for Row Increase - Vector Based CD Algorithm

Matrix Size	Processing Time(s)	
	Spark	Flink
10×10	71	67
10×100	188	181
$10 \times 1'000$	7'300	7'202
$10 \times 10'000$	33'800	33'203
$10 \times 100'000$	261'000	260'000
$10 \times 1'000'000$	2'559'000	2'557'000

Table 4.2: Time Costs for Column Increase - Vector Based CD Algorithm

Matrix Size	Processing Time(s)	
	Spark	Flink
10×10	144	139
100×10	498	490
$1'000 \times 10$	14’700	14’600
$10'000 \times 10$	67’700	67’100
$100'000 \times 10$	523’300	522’090
$1'000'000 \times 10$	5’119’000	5’117’300

Table 4.3: Time Costs for Row Increase - Record Based CD Algorithm

Matrix Size	Processing Time(s)	
	Spark	Flink
10×10	144	139
10×100	378	371
$10 \times 1'000$	14’500	14’403
$10 \times 10'000$	67’500	66’990
$10 \times 100'000$	523’000	521’900
$10 \times 1'000'000$	5’118’990	5’117’000

Table 4.4: Time Costs for Column Increase - Record Based CD Algorithm

Matrix Size	Processing Time(s)	
	Spark	Flink
10×10	71	67
100×100	440	425
$1'000 \times 1'000$	14’600	14’200
$10'000 \times 10'000$	67’600	66’200
$100'000 \times 100'000$	522’600	520’090
$1'000'000 \times 1'000'000$	5’115’000	5’111’300

Table 4.5: Time Costs for Row & Column Increase - Vector Based CD Algorithm

Matrix Size	Processing Time(s)	
	Spark	Flink
10×10	144	139
100×100	870	850
$1'000 \times 1'000$	29'200	29'000
$10'000 \times 10'000$	135'200	134'600
$100'000 \times 100'000$	1'046'300	1'041'600
$1'000'000 \times 1'000'000$	10'237'990	10'229'895

Table 4.6: Time Costs for Row & Column Increase - Record Based CD Algorithm

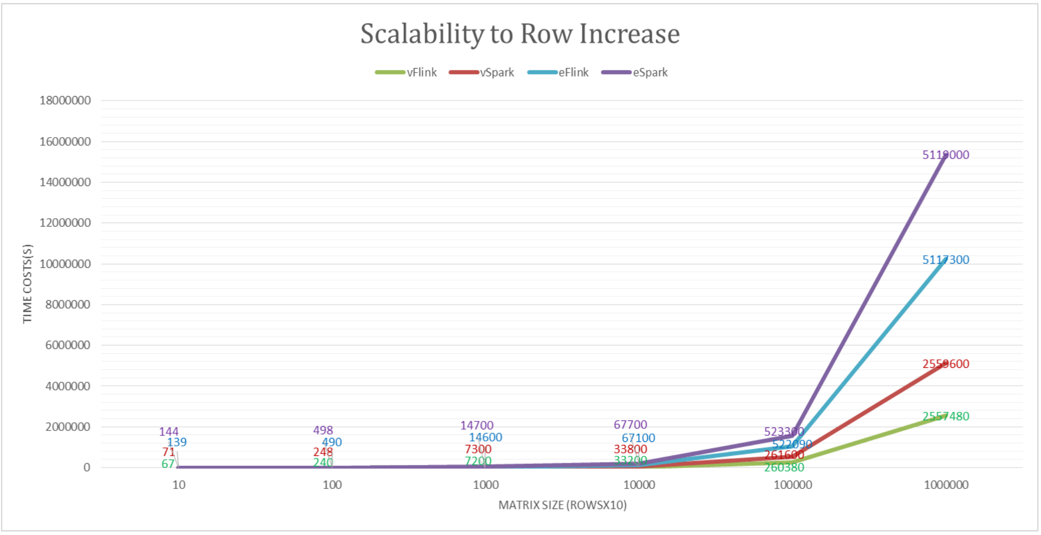


Figure 4.1: Scalability to Matrix Size-Row Increase

To better compare the time costs of element-based and vector-based CD algorithms implemented by Apache Spark and Apache Flink for different sized matrices, Figure 4.1 sumarizes all the above tables in one diagram, in which 'vSpark' represents the vector-based Spark implemented CD algorithm, 'vFlink' represents the vector-based Flink implemented CD algorithm, by contrast, 'rSpark' means the element-based Spark implemented CD algorithm and 'rFlink' means the element-based Flink implemented CD algorithm.In Figure 4.1, we can see that the vector-based CD algorithm has distinctly better performance than the element-based CD algorithm because of less shuffling requirements. Besides, we can also conclude that the Flink implemented CD algorithm performs better than the Spark implemented

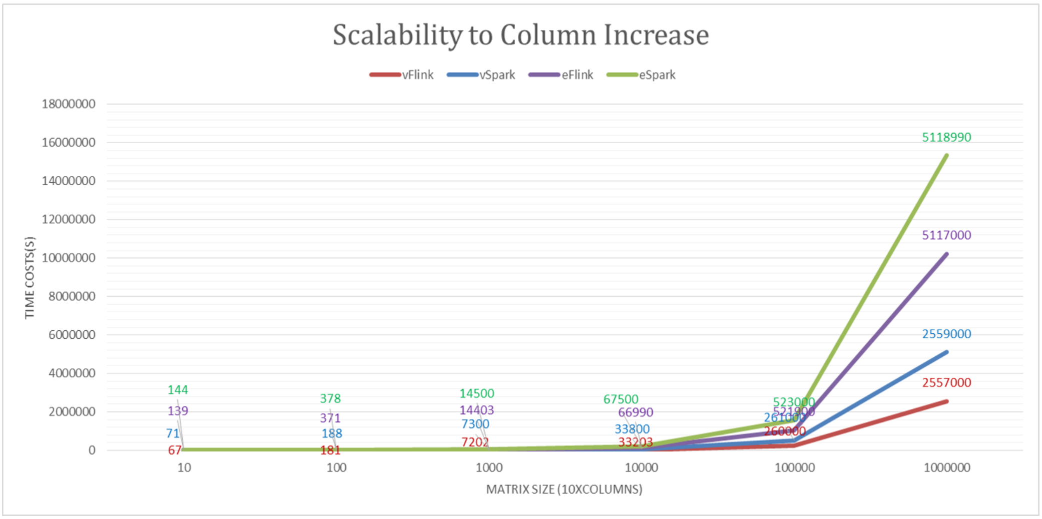


Figure 4.2: Scalability to Matrix Size-Column Increase

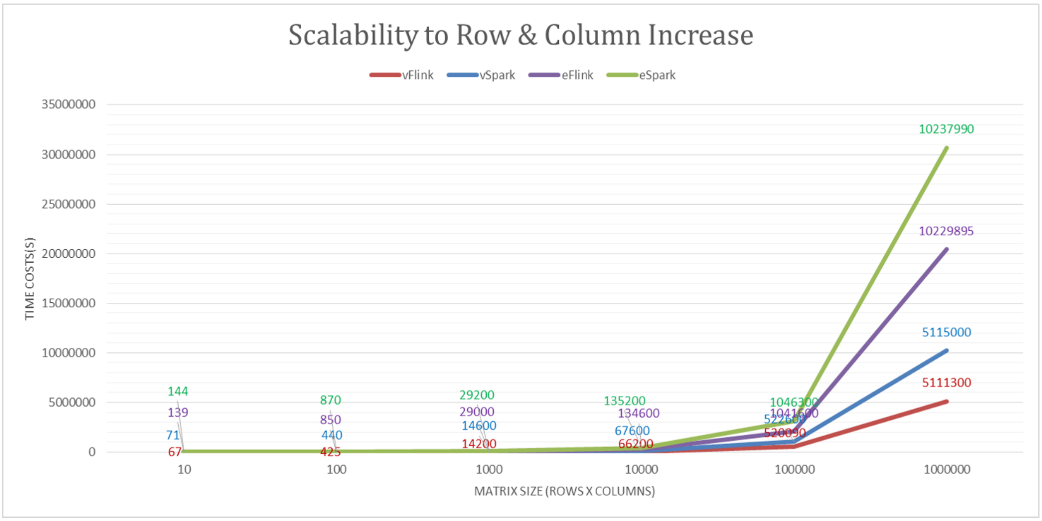


Figure 4.3: Scalability to Matrix Size-Row & Column Increase

CD algorithm since the optimizations for the data pipeline processing as we mentioned in Chapter 2.

Figure 4.2 presents the time costs of CD algorithms on column increased matrices. It shows the same tendency as Figure 4.1 that the vector-based CD algorithm performs better than the element-based CD algorithm and Flink implemented CD algorithm performs better than the Spark implemented CD algorithm.

Figure 4.3 exhibits time consumptions when rows and columns of the matrices increase simultaneously. In it, we can see that the vector-based CD algorithm still performs better than the element-based CD algorithm and Flink implemented CD algorithm still performs better than the Spark implemented CD algorithm, but with higher time costs.

4.2 Scalability to Hardware Provisions[2]

Parallel processing is a key feature of big data infrastructures. In order to evaluate the scalability to different hardware provisions, in this section, we conduct experiments with both machine scaling out and machine scaling up. For scaling out, we change the number of task executors from 1 to 8. Whereas, for scaling up, we add the number of execution cores for each of the executors. The test results are based on the combinations of different Scale-Out and Scale-Up configurations.

As shown in Figure 4.4, the vector-based CD algorithm still has better performance than the element-based CD algorithm, besides the Flink implemented CD algorithm gains advantages over Spark implemented CD algorithm. There is a summit in this Figure, which means the worst performance of the CD algorithm when it executes with only 1 machine using 1 core. Conversely, when the CD algorithm runs on 6 machines with 4 cores for each, it performs best. There is a tendency in this figure that the performance enhancement does not achieve linearly along with the scaling up/out of hardware provisions. The best performance is obtained in between the minimum and maximum hardware provisions, since the increase of machines means more network communication costs, to certain number, it impairs the distributed computation gains.

[2]All the experiments in this section are executed on 500×10 sized matrix.

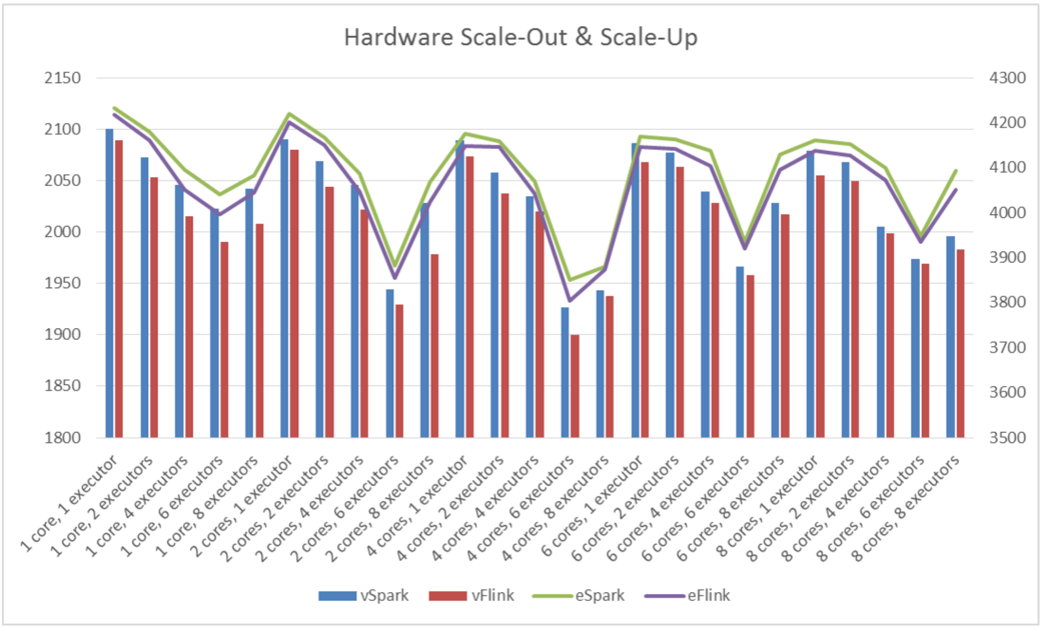


Figure 4.4: Scalability to Changes of Hardware Provision

4.3 Performance impact of value properties of matrix data[3]

All the previous experiments run on matrix data with independent and identically distributed (i.i.d.) random double values. However, based on the fact that the characteristics of the matrix data can have influences on the execution time of the CD algorithm, in this section, we test CD algorithm on five different characteristics of matrix data as listed below to see the real impacts:

1) Matrix Data with Complete Positive values
2) Matrix Data with Complete Negative values
3) Matrix Data with Mostly Positive values (only 10 negative values)
4) Matrix Data with Mostly Negative values (only 10 positive values)
5) Matrix Data with i.i.d Random values

According to the characteristic of SSV algorithm, when all values in the input matrices have same sign, either positive or negative, the number of iterations in SSV algorithm is least, i.e. all elements of the weight vector computed in the first iteration of the SSV algorithm, $V^{(1)}$, are positive. The

[3]All the experiments described in this section are run with 6 nodes (4 cores, 20G memory for each node), 500×10 sized matrix.

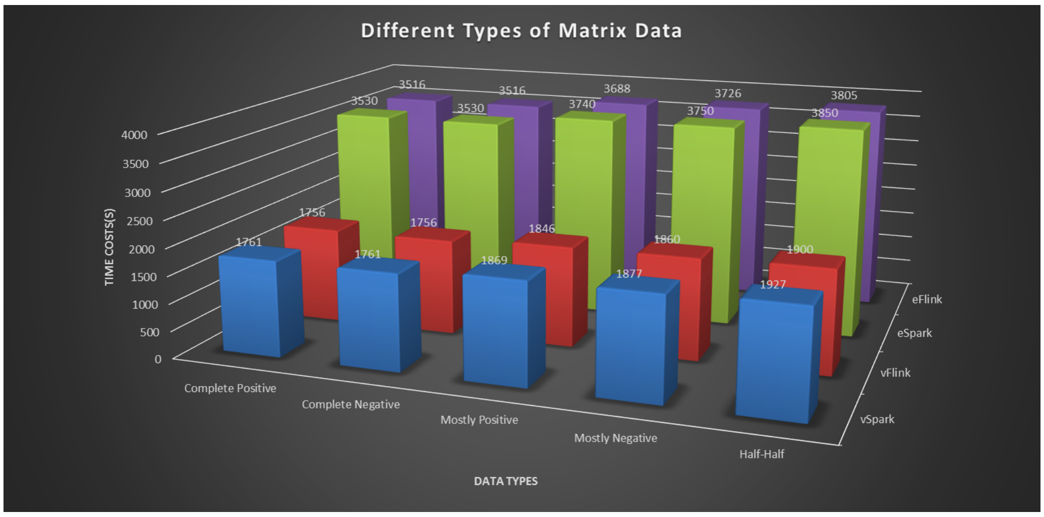


Figure 4.5: Influences from Different Types of Matrix Data

sign vector, Z, that contains only 1s is the maximizing vector[4]. As dinstinctly illustrated in Figure 4.5 that the matrix data with 'Complete Negative' and 'Complete Positive' values have the lowest time consumptions. By contrast, matrix data with 'Half-Half' random values have the highest time costs. Followed by matrix data with 'Mostly Positive' and 'Mostly Negative' values. Besides, as previous experiments, CD algorithm based on vectors has better time performance than the element based CD algorithm, and the Flink implemented CD algorithm is faster than the Spark implemented CD algorithm.

4.4 Algorithms Break-Down[4]

Preceding tests treat the CD algorithm as a whole to obtain the evaluation results. To better understand the execution mechanism of the CD algorithm and SSV algorithm, this section breaks down both of these algorithms to see the distribution of time expenses during their executions.

Figure 4.6 and Figure 4.7 exhibit the time consumptions of each step in CD algorithm. From them, we can conclude that the FindSignVector function spends most of the time (69%) during executions in both of the vector based and element based Spark/Flink CD algorithms, since input matrices with random values have highest time costs as shown in Section 4.3,

[4]All the experiments in this section are executed on YARN cluster with 6 nodes (4 cores, 20G memory for each node), 500×10 sized matrix.

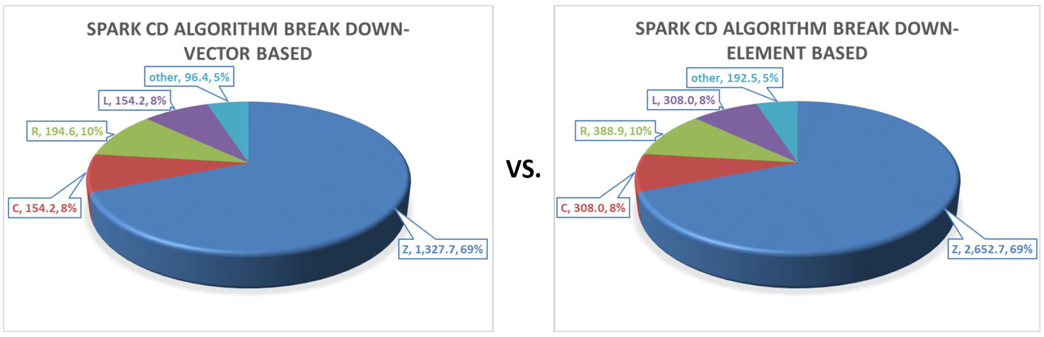


Figure 4.6: Spark CD algorithm Break-Down

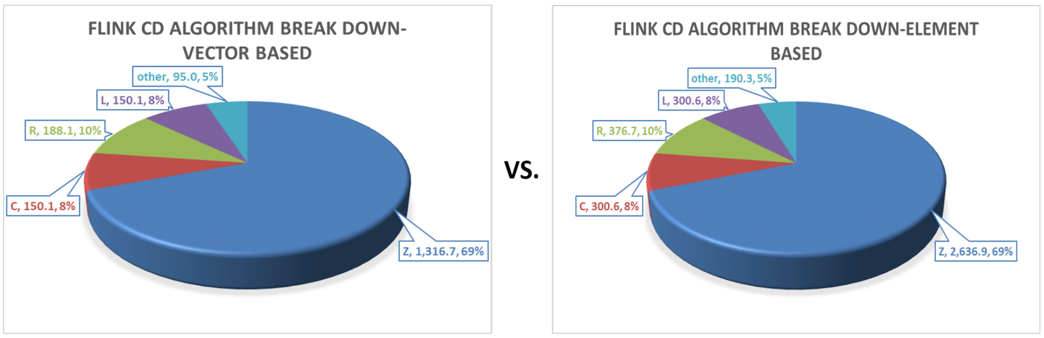


Figure 4.7: Flink CD algorithm Break-Down

it executes with many iterations. The computation of R takes 10% of the total computation time, followed by the computation of L and C which have almost the same time costs, 8% and 8% in both figures.

Figure 4.8 and Figure 4.9 expose the detailed time expense of SSV algorithms. We can clearly see that the computation for V grips the most part of the time expenses(45%), since it has most computation complexities as shown in Algorithm 2, followed by the V_i maximization operation 30% and the calculation of S 25%.

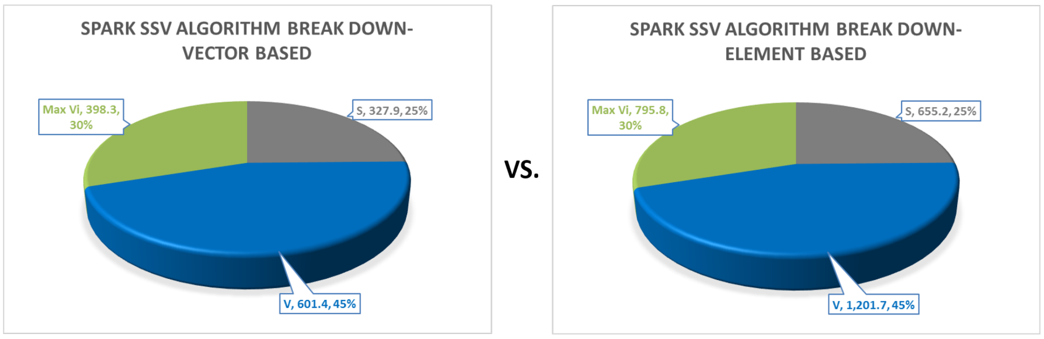


Figure 4.8: Spark SSV algorithm Break-Down

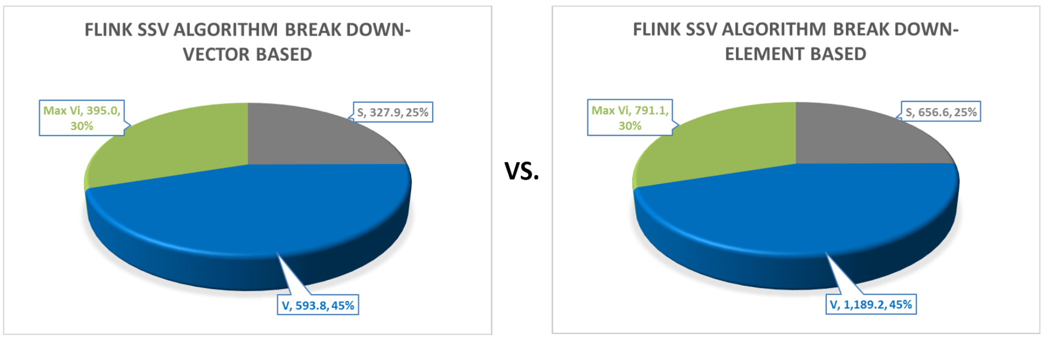


Figure 4.9: Flink SSV algorithm Break-Down

4.5 Difference Between Two V-Computation Strategies[5]

As we mentioned in Section 3.3, in the SSV algorithm, we adopt an alternative methodology to compute V. In this section, we can see the performance difference between these two strategies.

We can see in Table 4.7 and Table 4.8 that the detailed execution time consumptions of algorithm CD and SSV. In them, we can clearly know the performance discrepancy between the two different V computation strategies. The original strategy has obviously better performance than the alternative strategy, since the original one computes V vectror for each iteration separately, which means independent iterations and higher possibility of parallelism, whereas, the alternative one computes V vector in every iteration (except the first iteration) depending on result from previous iteration, which means for each iteration (except the first iteration), the computation has to wait for the accomplishment of previous iteration, therefore, lower possibility of parallelism.

Matrix Size	vSpark (S)	vFlink (S)	eSpark (S)	eFlink (S)	Percentage for Spark	Percentage for Flink
500x10	1927.0	1900.0	3850.0	3805.0		
CD Algorithm						
Z	1327.7	1316.7	2652.7	2636.9	68.90%	69.30%
C	154.2	150.1	308.0	300.6	8%	7.9%
R	194.6	188.1	388.9	376.7	10.10%	9.9%
L	154.2	150.1	308.0	300.6	8%	7.9%
other	96.4	95.0	192.5	190.3	5%	5%
Z	1327.7	1316.7	2652.7	2636.9	68.90%	69.30%
SSV Algorithm						
S	327.9	327.9	655.2	656.6	24.70%	24.90%
V(Original V)	**601.4**	**593.8**	**1201.7**	**1189.2**	**45.30%**	**45.10%**
Max V_i	398.3	395.0	795.8	791.1	30%	30%

Table 4.7: Original Strategy of Computing V

[5]All the experiments in this section are executed on YARN cluster with 6 nodes (4 cores, 20G memory for each node), 500×10 sized matrix.

Matrix Size	vSpark (S)	vFlink (S)	eSpark (S)	eFlink (S)	Percentage for Spark	Percentage for Flink
500x10	1987.8	1957.0	3911.0	3862.0		
CD Algorithm						
Z	1388.5	1373.7	2713.6	2693.9	69.54%	69.90%
C	154.2	150.1	308.0	300.6	8%	7.75%
R	194.6	188.1	388.9	376.7	9.89%	9.71%
L	154.2	150.1	308.0	300.6	8%	7.75%
other	96.4	95.0	192.5	190.3	5%	5%
Z	1388.5	1373.7	2713.6	2693.9	69.54%	69.90%
SSV Algorithm						
S	327.9	327.9	655.2	656.6	23.97%	24.20%
V(Alternative V)	**662.3**	**650.8**	**1262.6**	**1246.2**	**46.92%**	**46.64%**
Max V_i	398.3	395.0	795.8	791.1	29.11%	29.16%

Table 4.8: Alternative Strategy of Computing V

Table 4.8 also exhibits us that along with the increase of time consumptions of V computation, its proportion in total time consumptions also increases.

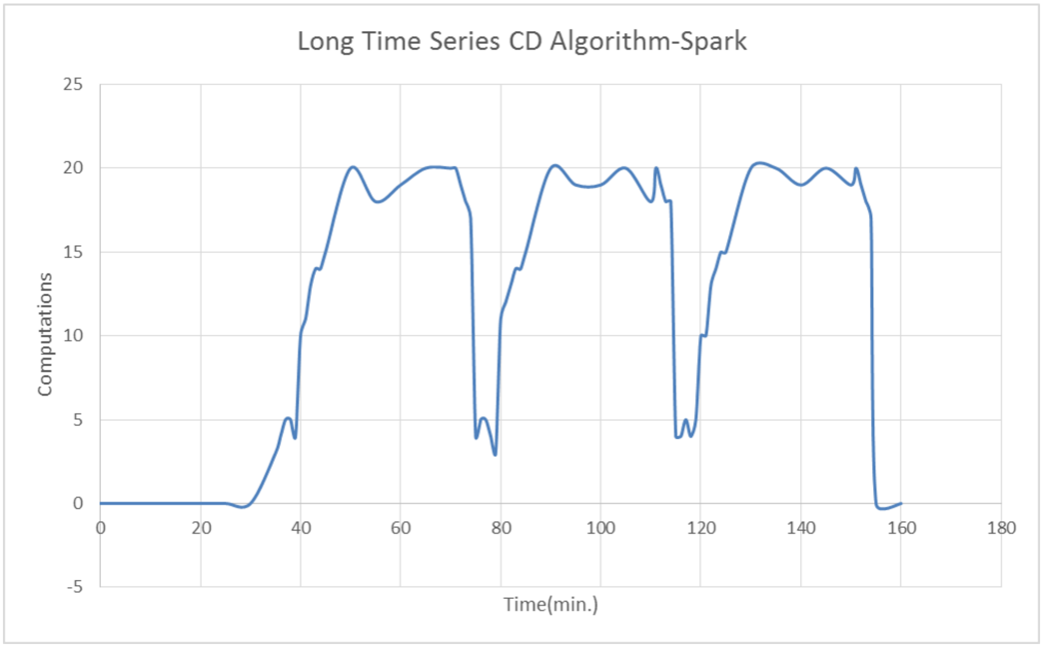
4.6 Comparison between two long time series CD consumers[6]

All the above sections evaluate the non-streaming CD algorithm, this section compares the two streaming versioned long time series CD algorithm.

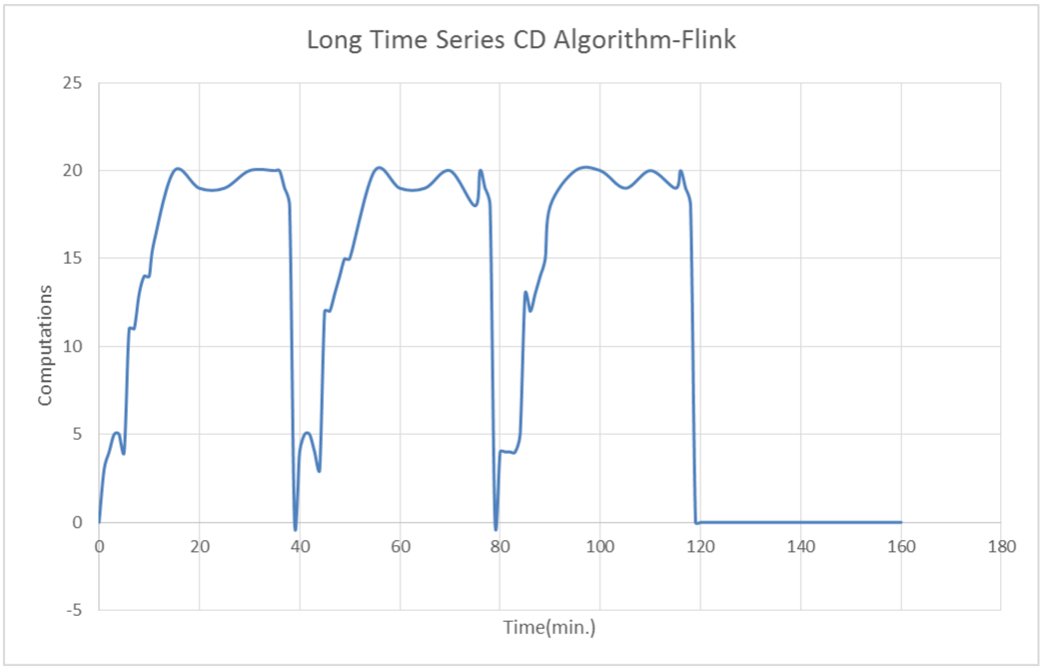
From Figure 4.10 we can clearly see that because Spark and Flink implement two different data pipeline architecture, Lambda architecture for Spark[16], Kappa architecture for Flink[14], the behaviours for the streaming process are also inconsistent. Spark regards streaming process as micro batches, between two micro batches, it has to wait for a parameterized time period, which implies that after Spark Cconsumer gets messages from Kafka Topics, if the next micro batch execution time does not arrive, the Spark

[6]All the experiments in this section are executed on YARN cluster with 6 nodes (4 cores, 20G memory for each node), evaluation lasts for 160 minutes, which means three rounds(column 0 to 30) of consumptions. For each round, Consumer simultaneously receives messages from 5 topics, 500 rows matrix per topic, i.e. totally 2500 rows matrices. The evaluation plot starts plotting after receivers finish registration on the Consumer side.

Consumer has to wait. Whereas, in Flink, processings are all based on real



(a) Spark



(b) Flink

Figure 4.10: Long Time Series CD Algorithm

streams. Therefore, whenever Flink Consumer obtains messages from Kafka Topics, the Flink runtime will start the CD processing immediately without Consumer side waiting.

Chapter 5

Conclusions

This thesis aims to improve the quadratic time complexity of single-machined Centroid Decomposition algorithm in distributed environment and to scale the CD algorithm to large matrices, as shown in Figure 5.1. It implements both non-streamed and streamed CD algorithm, with Apache Spark and Flink APIs and compares the two platforms. We firstly empirically evaluate the non-streamed CD processing algorithm under different circumstances. Afterwards, we compare the execution characteristics of the two long time series CD algorithm.

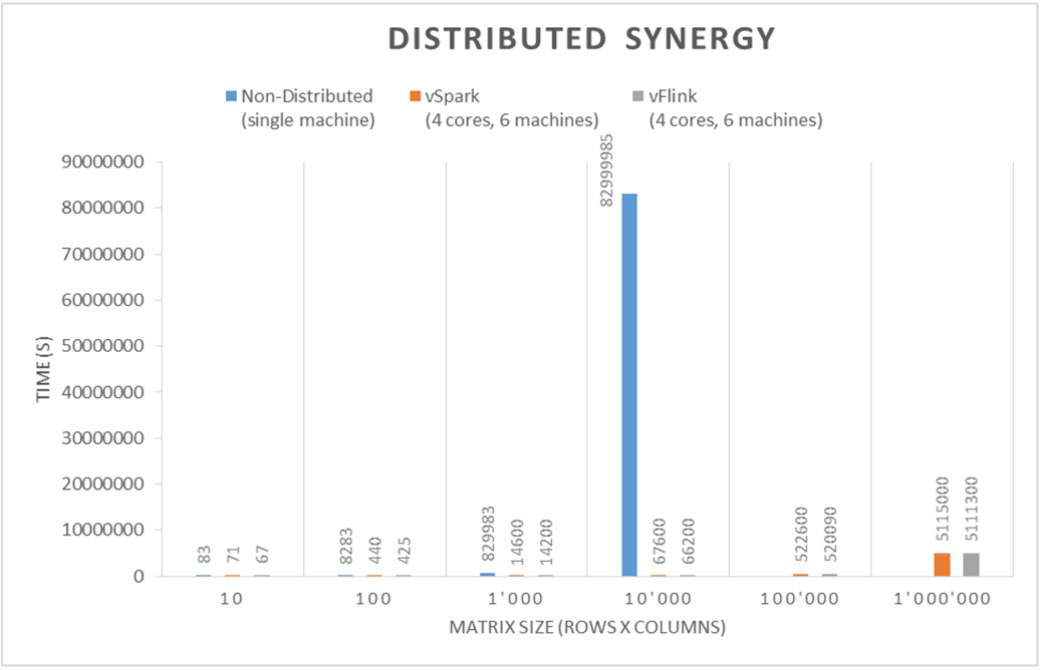


Figure 5.1: Single-Machined vs. Distributed CD Algorithm

The result of our experiments show that the vector based implementation has better performance than the element based implementation. Additionally, since Flink has inherent optimizations when executing the data processing pipelines, it has distinctly better performance than Spark, at least in our case. Besides, Flink has native closed-loop iteration (cyclic data flow) operators, it has more performance advantanges when executing incremental iterations over Spark[20].

Seeing that in Spark MLLib there is specific data structures, such as the 'DenseMatrix', 'DistributedMatrix', etc. The next step for us would be to implement our CD algorithm with these whole matrix data structure, since there have already been complete distributed version of basic matrix operations implemented in these classes for Spark MLLlib. However, although there is also same 'DenseMatrix' class in Flink, there haven't been implemented relating basic matrix operations in this class yet until Flink 0.10.0.

References

[1] "Singular Value Decomposition, http://mathworld.wolfram.com/singularvaluedecomposition.html."

[2] K. Karadimitriou and J. Tyler, "The centroid method for compressing sets of similar images," *Pattern Recognition Letters, vol. 19, no. 7, pp. 585-593*, 1998.

[3] R. Funderlic and M. Chu, "The centroid decomposition: Relationships between discrete variational decompositions and svds," *SIAM J. Matrix Anal. Appl., vol.23, no. 4, pp. 1025-1044*, 2001.

[4] M. Khayati, M. Boehlen, and J. Gamper, "Memory-efficient centroid decomposition for long time series," *ICDE*, 2014.

[5] "Apache Hadoop, https://hadoop.apache.org/."

[6] "Apache YARN, http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/yarn.html."

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *NSDI*, 2012.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," *HotCloud*, 2010.

[9] "Apache Spark, http://spark.apache.org/."

[10] "Apache Flink, https://flink.apache.org/."

[11] "Apache flink what how why who where, http://www.slideshare.net/sbaltagi/apacheflinkwhathowwhywhowherebyslimbaltagi-57825047."

[12] "Stratosphere, http://stratosphere.eu/."

[13] S. E. et al, "Spinning fast iterative data flows," *VLDB*, 2012.

[14] L. Foundation, "Kappa architecture: Our experience," *Linux Foundation Press Release*.

[15] "Lambda vs. Kappa architecture, http://www.ericsson.com/research-blog/data-knowledge/data-processing-architectures-lambda-and-kappa/."

[16] "Lambda architecture, http://lambda-architecture.net/."

[17] "Apache kafka, http://kafka.apache.org/documentation.html."

[18] "Iterations,https://ci.apache.org/projects/flink/flink-docs-release-0.10/apis/iterations.html."

[19] "Daplab, http://daplab.ch/."

[20] F. Hueske, "Apache flink fast and reliable large-scale data processing,"