Building Histogram on Large Dataset in Apache Flink

Big Data Analytics Project - Issue 12

Dieu Tran Thi Hong	(396749)
Pandu Wicaksono	(393471)
Shibo Cheng	(396831)

March, 2018

Database Systems and Information Management Group
Technical University of Berlin

Table of Contents

1 Introduction

Histogram is one of the popular summarization method which represents information about data. Given a massive amount of data, creating a histogram from the data can provide an insightful information such as maximum element, minimum element, and the distribution of the data. Due to the benefit that it gives, histogram is mainly used for exploratory data analysis. One of the common choice for histogram is Haar wavelet histogram [1][2].

Haar Wavelet Histogram represents information about the data by making a wavelet tree recursively. Starting from the frequency of each element in the data, wavelet histogram builds the tree in bottom-up-fashion. Once the tree is created, each element of the tree represents the wavelet coefficients. The top-K coefficients from this tree gives an approximation of the distribution of the data. Using the top-K coefficients, we can reconstruct the frequency of each element, therefore Haar Wavelet Histogram can be used to compress and summarize information from a dataset. Due to popularity of distributed system to handle massive amount of data, researchers have begun implementing the histogram creation using distributed systems.

Jestes et al. (2011) [1][2] proposed a method of implementing the wavelet histogram in a distributed system. Using Hadoop MapReduce, they successfully implemented the parallel fashion of calculating histogram. The implemented algorithms include exact solutions and approximate solutions for calculating the histogram. The experimental studies were conducted using WorldCup dataset [3].

2 Goal of the Project

The goal of this project is to implement the proposed histogram algorithms in Flink and reproduce the experiments suggested by [1][2]. Mainly, this project aims to implement and test the following approaches for computing wavelet histogram:

- Exact Solutions
  - SendV
  - SendCoef
  - H-WTopK
- Approximate Solutions
  - Basic-S
  - Improved-S
  - TwoLevel-S

The evaluation of the method should be conducted using WorldCup dataset [3] as suggested by the paper.

3 Description of Wavelet Histogram Algorithms

This section describes about wavelet histogram algorithms. It is started by explaining the overall project workflow, then this section gives a brief explanation of calculating wavelet histogram. The third part of this section gives a brief introduction of the implemented algorithms for calculating the wavelet histogram.

3.1 Project Workflow

Our project workflow can be divided into three big parts: Data Preprocessing, Flink Execution, and Analysis.

In Data Preprocessing phase we download the WorldCup dataset [3] and preprocess the data. After pre-processing, the total number of record in the dataset is 1.35 billion record and the size is 11.46GB.

The second phase is Flink Execution in which we execute our implemented algorithms in the DIMA cluster. During this phase, we take a screenshot from Flink web interface to gather the communication cost and running time. In addition, our implemented algorithms produce a file that contains the wavelet coefficient.

The last phase is Analysis. In this phase, we reconstruct the frequency of the element in the dataset using the top-K coefficients. Then we calculate the SSE using formula SSE = ⍰(actual frequency - reconstructed frequency)^2. Using the Flink web interface and SSE calculation, we analyze the behaviour of the implemented algorithm based on our pre-defined metrics.

3.2 Histogram

Histogram is a summarization tool that represents data [1][2]. Given a dataset that has many attributes, we can calculate the histogram of an attribute. The domain of the histogram is a set of distinct elements in that attribute. The frequency of the histogram is the count of the distinct element in that attribute.

One of the common choice for histogram is Haar Wavelet Histogram which is used by the paper [1][2]. Haar Wavelet Histogram can be calculated using a recursive approach in bottom up fashion. Figure 1 illustrates the calculation of Haar Wavelet Histogram. In the lowest bottom row, we have the frequency of each element. We calculate the Haar Wavelet Histogram by calculating the average difference and the average of two elements. For example, average difference C_4 = (v(2) - v(1))/2 , and average a_4 = (v(2) + v(1))/2. We continue calculating the average difference and average in bottom up fashion until we arrive at the top element and we get a tree. Then, the wavelet coefficients w_i are [a_1, c_1, c_2, …, c_{u-1}] , which is highlighted as the value marked with the red circle. Then, we can take the top-K coefficients and keep the values as the compressed and summarized version of the dataset.

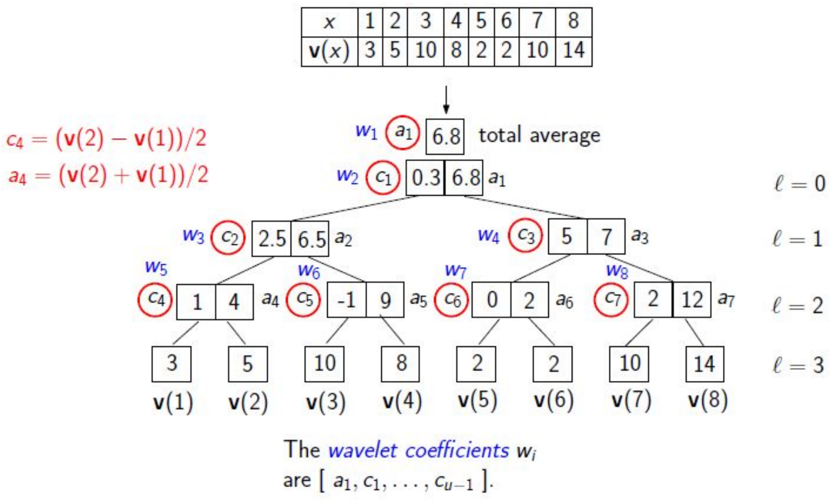

x 1 2 3 4 5 6 7 8
v(x) 3 5 10 8 2 2 10 14
$c_4 = (\mathbf{v}(2) - \mathbf{v}(1))/2$
$a_4 = (\mathbf{v}(2) + \mathbf{v}(1))/2$
w_1 a_1 6.8 total average
w_2 c_1 0.3 6.8 a_1
w_3 c_2 2.5 6.5 a_2
w_4 c_3 5 7 a_3
w_5 c_4 1 4 a_4
w_6 c_5 -1 9 a_5
w_7 c_6 0 2 a_6
w_8 c_7 2 12 a_7
3 5 10 8 2 2 10 14
v(1) v(2) v(3) v(4) v(5) v(6) v(7) v(8)
$\ell = 0$
$\ell = 1$
$\ell = 2$
$\ell = 3$
The wavelet coefficients w_i are $[a_1, c_1, \ldots, c_{u-1}]$.

Figure 1 Haar Wavelet Histogram [1]

3.3 Algorithms

3.3.1 SendV

SendV is the simplest algorithm as suggested in the paper [1][2]. SendV can be divided into three big steps: initialization phase, mapper phase, and reducer phase. During the initialization phase, the algorithm reads data from data source, then it splits the data and sends the data to each node in the cluster. During the mapper phase, the algorithm calculates the local frequency of each element. In the reducer phase, the reducer accepts the data and calculate the global frequency of each element. Then the reducer calculates the wavelet tree and takes the top-K coefficients. By the end of the execution, our implementation also writes the top-K coefficients into a file hence we can reconstruct the frequencies and calculate the SSE later.

3.3.2 SendCoef

SendCoef is the second algorithm introduced in the paper [1][2]. SendCoef is a slight modification of SendV. The difference is in the mapper phase, SendCoef not only calculates the local frequency of each element but also calculates the local wavelet tree. Then the mapper sends the value of the local wavelet tree to the reducer. In the reducer, SendCoef calculates the global wavelet tree and takes the top-K coefficients.

3.3.3 H-WTopK

H-WTopK is the last exact solutions introduced in the paper [1][2]. It starts by splitting the data into each node during the initialization phase. Then, the algorithm consists of three big phases.

In the first phase, the algorithm calculates the local frequency and local coefficient in each node. Then each node sends the k highest and k lowest coefficients to the reducer. The reducer calculates a lower bound $\tau(x)$ on its total score magnitude $|r(x)|$ for each item x through the following steps:

- Computes upper bound τ^+ and lower bound τ^- on its total score $r(x)$ by:
  - If a node sends out a score of element x then add it's exact score
  - Otherwise add the k-th highest score to τ^+ and add the k lowest score to τ^-
- Then, if τ^+ and τ^- have different signs assigns $\tau(x) = 0$
- Otherwise assigns $\tau(x) = min\{|\tau^+(x)|, |\tau^-(x)|\}$

After $\tau(x)$ has been calculated, pick the k-th largest $\tau(x)$ denoted by T_1. T_1 is a threshold for the magnitude of the top-k items.

In the second phase, node j emits all local items x if $|r_j(x)| > T_1/m$. Then the reducer will update the bounds using the same method as in the first phase. However, in this second phase if a node did not send the score of some x, the algorithm uses T_1/m for calculating $\tau^+(x)$ and $-T_1/m$ for calculating $\tau^-(x)$, respectively. This will produce a new tighter threshold T_2. Then the algorithm prunes items from R as follows: for any $x \varepsilon R$ computes the new threshold $\tau'(x) = max\{|\tau^+(x)|, |\tau^-(x)|\}$ based on the refined $\tau^+(x)$ and $\tau^-(x)$. If $\tau'(x) < T_2$, delete item x from R.

In the third phase, the algorithm asks each node for the scores of all items in R. From the values, the algorithm calculates the aggregated scores and picks the k-th largest items.

3.3.4 Basic-S

The main problem of the SendV algorithm is it sends too many data to the reducer [1][2]. Therefore, the author of the paper proposes to use sampling to solve this problem. Basic-S can be divided into three main phase: initialization, mapper, and reducer.

The difference with SendV can be seen in the mapper phase. In the mapper phase, the algorithm samples the data prior to the calculation of local frequency. The data is sampled using probability P1 = 1 / (ε^2 n), where ε is a parameter for sampling and n is the number of record in the entire dataset. ε needs to be fairly small to approximate well. After the data has been sampled, the algorithm calculates the local frequency and sends the value to the reducer.

In the reducer phase, first the algorithm calculates the global frequency of each element. Then the algorithm will approximate the global frequency of the entire dataset by dividing the calculated global frequency by the probability P1. After the algorithm has approximated the global frequency of the entire dataset, the algorithm calculates the wavelet tree. Finally, the algorithm takes the top-K coefficients and writes them into a file.

3.3.5 Improved-S

Improved-S tries to further reduce Basic-S' communication cost by ignoring elements with low frequency [1][2]. This is done by putting a condition in the mapper phase. After the data has been sampled and the local frequency has been calculated, if the local frequency of the element is more than ε * t_j, where t_j is the total number of sampled records in split *j*, the mapper will send the value to the reducer. Otherwise, the mapper will not send the value to the reducer. Therefore, the estimated frequency is biased because this method ignores all the items with small frequency. In Improved-S, the initialization and reducer phase works the same as Basic-S.

3.3.6 TwoLevel-S

TwoLevel-S tries to further improve the sampling approach by performing the sampling with an unbiased estimator [1][2]. This is reflected in the mapper phase and in the reducer phase, the initialization phase is the same as Basic-S and Improved-S. The idea behind TwoLevel-S is to give low-frequency elements a chance to be considered in the approximation, therefore the algorithm performs two levels of sampling.

In the mapper phase, first the data is calculated with probability P1 = 1 / (ε^2 n). Then, the algorithm calculates the local frequency of each item x. If the local frequency is more or equal to 1 / (ε $\sqrt{m}$), where m is the number of mappers, emits (x, local frequency) to the reducer. Otherwise, emits (x, null) with probability P2 = (ε $\sqrt{m}$ * local frequency).

In the reducer phase, the algorithm calculates the global frequencies as follows:

- If (x, local frequency) is received then $\rho(x) = \rho(x) + local\,frequency$
- Else if (x, null) is received then $M(x) = M(x) + 1$
- After all x have been processed, $s(x) = \rho(x) + M(x) / \varepsilon\sqrt{m}$
- Calculate the global frequency $v(x) = s(x) / P1$, where $P1 = 1 / (\varepsilon^2 m)$

After the global frequency has been calculated, the algorithm will calculates the wavelet tree then it will take the top-K coefficients.

4 Experiment Design and Executions

4.1 Cluster Configuration

We utilized the available DIMA cluster in our project. We used Flink 1.3.2 with 6 nodes. The important configurations of the cluster are described in Table 1 and Table 2.

Task Managers	6
Task Slots	288

Table 1 General Information

akka.ask.timeout	6000 s
akka.framesize	10985760
akka.lookup.timeout	1000 s
akka.watch.heartbeat.interval	100 s
akka.watch.heartbeat.pause	600 s
env.java.home	/etc/alternatives/java_sdk_1.8.0
env.java.opts	-XX:+UseG1GC
jobmanager.heap.mb	15024
jobmanager.rpc.address	ibm-power-1.dima.tu-berlin.de
taskmanager.heap.mb	50152
taskmanager.memory.fraction	0.6
taskmanager.memory.off-heap	true
taskmanager.memory.preallocate	false
taskmanager.network.numberOfBuffers	300000
taskmanager.numberOfTaskSlots	48

Table 2 Detail Configuration

4.2 Flink Job Diagram

During the program execution, we took the Flink job diagram, therefore the reader can easily understand the Flink job that we implemented. We have put the Flink job diagram in the appendix section in this report.

4.3 Datasets

We used the WorldCup dataset in our experiment [3]. The dataset is a log of user accessing the WorldCup website between April 30th 1998 and July 26th 1998. The total record of the dataset is 1,352,804,107 records.

The dataset consists of 8 attributes: Timestamps, ClientID, ObjectID, Size, Method, Status, Type, and Server. We preprocessed the data by going sequentially through all the records, and converting the pair (ClientID, ObjectID) in each record into a positive integer. Each integer is a key that represent a unique (ClientID, ObjectID) pair and vice versa. The number of keys forms up the domain size.

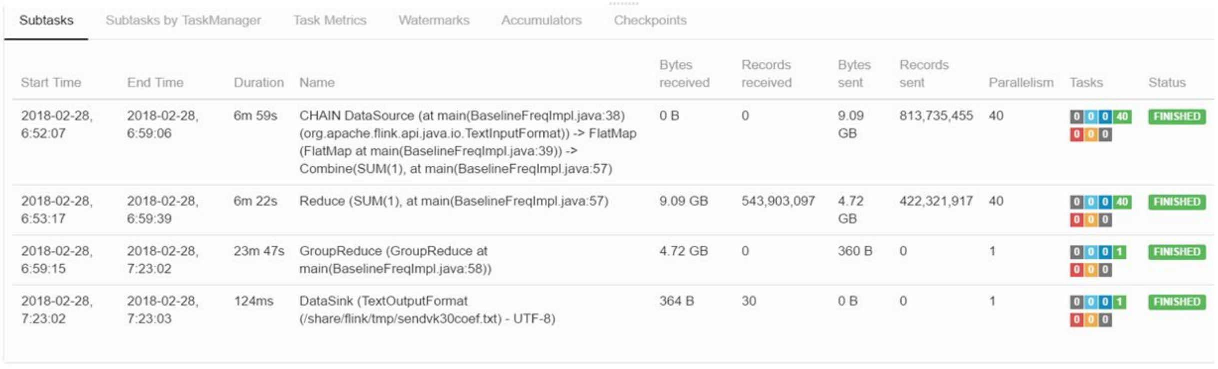
In summary, the dataset has n = 1.35 billion of records, domain key U = 2^{29} integers, and the size is 11.49GB after pre-processing. Due to the massive amount of the data, computing the whole dataset using our implemented algorithm cannot be done on one machine. Furthermore, the independent nature of each tuple in the dataset makes the dataset an ideal dataset to experiment in the cluster.

4.4 Experiment Metrics

4.4.1 Communication Cost

Communication cost is the number of bytes sent from the mapper to the reducer. The proposed algorithm in the paper tries to minimize the communication cost between machine. Therefore, we performed an evaluation of this metric. The data is gathered from Flink web interface during the execution and we manually process the data to analyze the result.

Figure 2 is an example of task execution plan in Flink web interface. We calculate the “Bytes sent” between the FlatMap operation and Reduce operation, which is 9.09GB in this example.



Start Time	End Time	Duration	Name	Bytes received	Records received	Bytes sent	Records sent	Parallelism	Tasks	Status
2018-02-28, 6:52:07	2018-02-28, 6:59:06	6m 59s	CHAIN DataSource (at main(BaselineFreqImpl.java:38) (org.apache.flink.api.java.io.TextInputFormat)) -> FlatMap (FlatMap at main(BaselineFreqImpl.java:39)) -> Combine(SUM(1), at main(BaselineFreqImpl.java:57)	0 B	0	9.09 GB	813,735,455	40	0 0 0 40 0 0 0	FINISHED
2018-02-28, 6:53:17	2018-02-28, 6:59:39	6m 22s	Reduce (SUM(1), at main(BaselineFreqImpl.java:57)	9.09 GB	543,903,097	4.72 GB	422,321,917	40	0 0 0 40 0 0 0	FINISHED
2018-02-28, 6:59:15	2018-02-28, 7:23:02	23m 47s	GroupReduce (GroupReduce at main(BaselineFreqImpl.java:58))	4.72 GB	0	360 B	0	1	0 0 0 1 0 0 0	FINISHED
2018-02-28, 7:23:02	2018-02-28, 7:23:03	124ms	DataSink (TextOutputFormat (/share/flink/tmp/sendvk30coef.txt) - UTF-8)	364 B	30	0 B	0	1	0 0 0 1 0 0 0	FINISHED

Figure 2 SendV Subtask Execution Summary

4.4.2 Execution Time

Execution time measures the time taken for the program to complete its entire execution. This data is gathered in the same fashion as communication cost, i.e., through Flink web interface. As shown in Figure 2, we calculate the time difference between the first operation’s start time and last operation’s end time, which is 1,855 seconds in this example of Figure 2.

4.4.3 Sum of the Squared Errors (SSE)

Sum of the squared errors represents the level of error that the algorithm gives when producing the wavelet histogram. It is calculated by the formula

$$SSE = \Sigma(actual\,frequency - reconstructed\,frequency)^2$$

The result of SSE gives an indication of the capabilities of the algorithm and its parameter to preserve the information in the data. The less SSE that it has, the better the result.

4.5 Experiment Parameter and Design

We utilized two parameters in our experiment: the number of top-K coefficients and ε which are parameters for the approximate methods to calculate the sample size = $1/\varepsilon^2$. Table 3 summarizes the design of our experiment.

Algorithm	**Number of Varying K**	**Number of Varying** ε	**Number of Run**
SendV	5	N/A	5
SendCoef	5	N/A	5
H-WTopK	5	N/A	5
Basic-S	5	4	9
Improved-S	5	4	9
TwoLevel-S	5	4	9
			42

Table 3 Experiment Design

We vary K = 10, 20, 30, 40, 50 and $\varepsilon = 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}$ as suggested by the paper. For the approximate method, we fixed K = 30 when iterating ε and we fixed $\varepsilon = 10^{-4}$ when iterating the K. For each run, we also run two jobs: reconstruct the frequency which is done in Java program and calculate the SSE which is done in Flink.

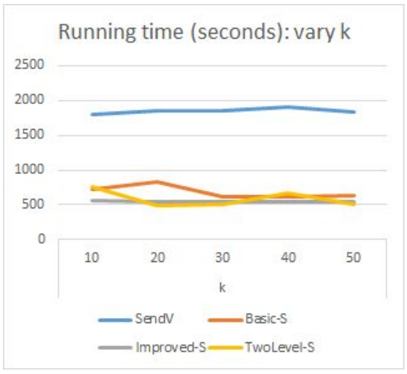
4.6 Executions

We compile all of the code into a jar file and we submit this jar to Flink. Then, we specify the main class and parameter in the arguments to run the algorithm or the SSE calculation program. We have put the detailed explanation of how to compile and run the jar in README file that we have provided.

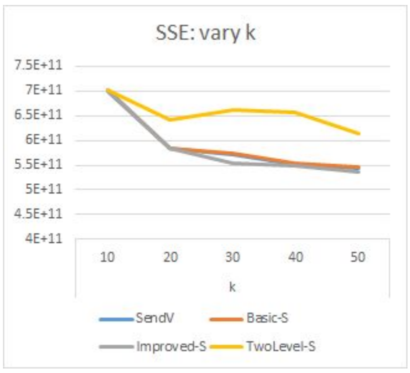
5 Experiment Result and Analysis

We successfully implemented 5 out of 6 algorithms: SendV, SendCoef, Basic-S, Improved-S and TwoLevel-S. For the H-WTopK, we implemented the algorithm using two approaches, however, the two approaches did not work as intended. The details of problem is explained in the section below. The result of the experiment shows that we obtained the same trend as our referenced paper with minor differences.

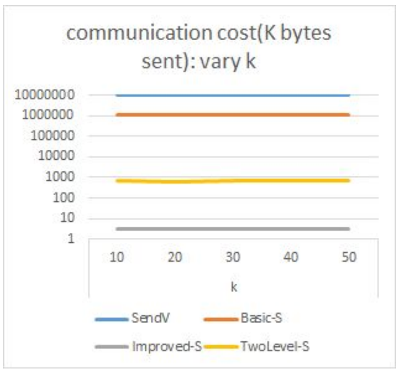
5.1 Varying K



(a) Running Time



(b) SSE



(c) Communication Cost

Figure 3: Results of Varying K

From Figure 3, we can see the three sampling methods' result compared to SendV.

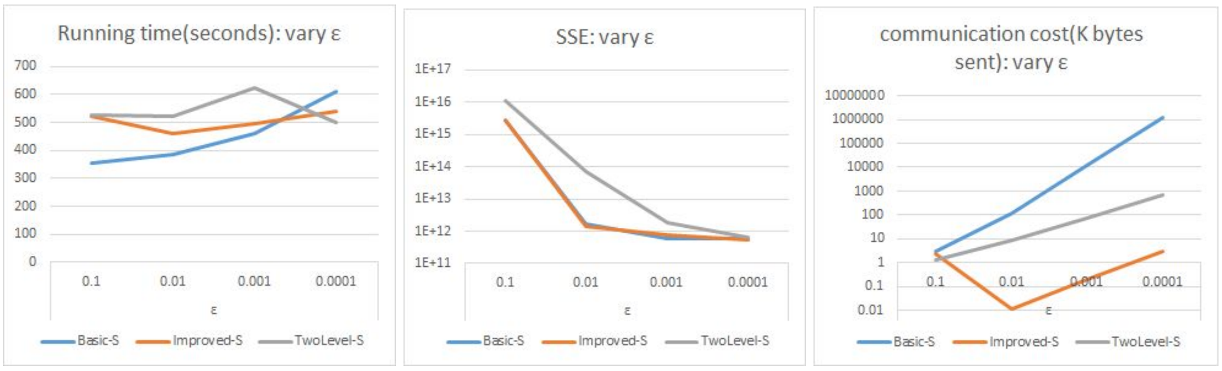
For running time in Figure 3(a), the three sampling methods take 500-1,000 seconds, while SendV needs around 1,800 seconds. We can see the effect after applying sampling, the data needed for computation and sent between operations are decreased, so the running time is decreased. The communication cost are also reduced, SendV's communication cost is around 9GB, Basic-S is 1GB. And Improved-S and Twolevel-S manage to decrease the communication cost down to less than 1MB. These two metrics do not vary much with different K. This is because we put all local coefficients to a priority queue, this queue output top-K coefficients which takes little cost difference between different K.

As for SSE in Figure 3(b), the graph does not show the expected result. We expected all the four methods should have similar SSE and TwoLevel-S should be better than Improved-S. In theory, Twolevel-S does not ignore items with small frequency like Improved-S and generate unbiased data frequency. However, in our experiment result, TwoLevel-S has the worst SSE. After outputting the intermediate data after the two level sampling, we found out that the problem lies in our data distribution and threshold.

In our dataset, each record represents a user accessing a URL of WorldCup website. Usually, people do not access the same URL evenly in a period of time. For example, a match news will lose people's attention after a certain time. Some data only exists in a certain period of time which might exist in only one split. This makes our dataset non-uniform and causes big variance and instability for TwoLevel-S. For example, an item has a frequency of 80 and the threshold is 100. If the data is distributed uniformly, it will be estimated around 80 based on the algorithm, but if it is not distributed uniformly and only appears in one split, it would be estimated to 100 with 80% chance, or 0 with 20% chance. Both have an obvious gap with 80. This would cause a higher SSE.

In our experiment, threshold also plays an important role in TwoLevel-S. Improved-S has a threshold around 250, which generate a good SSE that is only 5% difference to SendV for all K. However, based on the algorithm, TwoLevel-S' threshold is 1,581 high, which results in only 22 items pass the threshold and 64,018 items are survived in second level sampling. Of all the small-frequency items, 63,200 of them only appears in one split, which would be estimated as the example in previous paragraph, either 1581 or 0. This causes a large variance and instability in TwoLevel-S, which is shown by SSE.

5.2 Varying ε



(a) Running Time (b) SSE (c) Communication Cost

Figure 4 Results of Varying ε

The sample size equals to $1/\varepsilon^2$. The smaller the ε means the larger sample size. From Figure 4(a), we can see that with larger sample size, the data need to be computed and communicated will increase which in turn increase the running time slightly for the three algorithms. And a larger sample helps the algorithms to generate a more accurate wavelet tree, therefore, the SSE is smaller. As for communication cost, the cost of TwoLevel-S and Basic-S increase while Improved-S does not increase largely. This is because Improved-S has a threshold in sampling, only data larger than the threshold can be sent from mapper to reducer, and the number is quite small because the dataset similarly follows a Zipfian distribution.

5.3 Issues with H-WTopK

There are 3 requirements for this HTopK algorithm:

- No recomputation.
- Only sending minimal data from mapper to reducer (i.e., send only top-K): so as to improve performance by reducing communication cost.
- Phase 2 can only start after phase 1 has finished (same for phase 3 only starts after phase 2).

In our implementations, we implement phase 1 of the algorithm in two ways:
The first approach is to calculate local coefficients and top-K coefficients in one operator. This mapper will output top-K coefficients as the result of the dataset, meanwhile writing local coefficients to disk for computation in second phase and third phase. This method satisfies the first and second requirement. However, there is no guarantee of a synchronization control that phase 2 starts after phase 1 finishing writing file. In our experiment, the result is not correct because phase 1 has not finished writing when phase 2 starts.

The second approach is to calculate local coefficients and top-K coefficients in two separate operators. One dataset will store local coefficients, and another mapper will take this dataset as input and calculate top-K coefficients. This method can produce correct result and satisfy the first and third requirement. However, it violates the second by sending all local coefficients from the first mapper to the second, which makes the execution time extends to 8 hours.

We also tried Flink iteration for this algorithm, however it did not work. This is because in iteration, we still need to decide whether to put the calculation of local coefficients and top-K coefficients in one operation or two operations. More specifically, we tried with DeltaIteration, where maxIterations is set to 3. In DeltaIteration, we have workset and solution set and our idea is to have workset contains all local coefficients (i.e., it plays the role of mapper in phase 2 and 3), and solution set to keep track of top-K coefficients (i.e., it plays the role of reducer in all phases). However, the challenge in setting up initial workset and initial solution set would be the same as before. More precisely, we still need to have two dataset, one contains all local coefficients (initial workset) and one contains only top-K coefficients in each partition (initial solution set, which is also used to compute the bound). To compute these dataset separately, we would need to send all the data (we tried with initial workset contains tuple with (key, coefficient) then compute top-K with mapPartition using PriorityQueue, and also tried with initial workset that contains tuple with (key, coefficient, isTopK, mapperID) information, then filter/groupBy by isTopK. In both cases, the data sent is huge). In short, to the best of our knowledge, implementation using Flink iteration did not work without violating requirement 2 of H-WTopK.

6 Discussions and Future Work

Based on the experiment and algorithms, we can see the overall running time decreased a lot from the MapReduce implementation in the paper, and all the three sampling algorithms' running time does not differ much. We can see Flink's efficiency in computing in memory compared to MapReduce writing the intermediate result to disk. Therefore, if the required response time is around 30 minutes, then SendV is the option because it calculates the exact wavelet coefficient. If the required response time is around 10 minutes, then Basic-S is the best option. Because it produces a similar SSE to SendV, and Improved-S generate biased data in theory and TwoLevel-S is affected by data distribution and threshold.

In our opinion, the most difficult part in our project is understanding the algorithm and coding it in Flink. Reading the paper and understanding the algorithm proves to be the most difficult part, due to the complexity of the algorithm and mathematical notation presented in the paper. Fortunately, the author of the paper provides slides that explains the algorithm in a more clear and understandable way. Therefore, we suggest that researchers who would like to continue this project should read both the paper and the slides.

Performing the experiment in the cluster is a long process. We spent around three days in performing the experiment as well as performing SSE calculation in the cluster. Therefore, we suggest that researchers who would like to continue this project should plan the ahead the time required to perform the experiment.

7 Conclusions

To sum up, in this project we have successfully implemented the exact and approximate solution as suggested by the paper, except for H-WTopK. H-WTopK is implemented in two ways. The first is by sacrificing synchronization barrier which in result throws an execution error. The second is by sacrificing the communication cost which produces correct result but too much communication.

We have successfully performed the experiment on the cluster using the WorldCup dataset. The experiment is performed using 5 values of varying K and 4 values of varying ε. During the experiment, we are running the algorithms as a Flink job, in addition we also run the SSE calculation in Flink. The result of the experiment shows that we got the similar trend as with our referenced paper.

The result of our experiment shows that varying the number of top-K coefficient does not affect the running time, this is because the calculation of top-K coefficient is performed in the reducer. Furthermore, the sampling method reduces the data sent from the mapper to the reducer. Therefore, the sampling method decreases the running time and approximates the underlying data distribution. In addition, we found out that the data distribution is impacting the result of the approximate solution. This is especially shown when non-uniform data distribution affects the performance of TwoLevel-S.

References

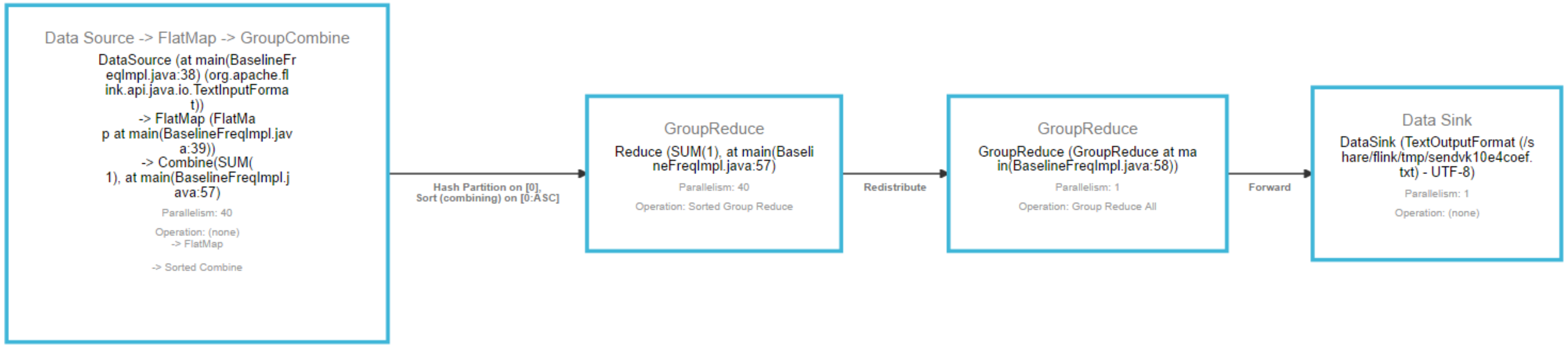
[1] Jestes, Jeffrey, Ke Yi, and Feifei Li. Building wavelet histograms on large data in MapReduce. Presentation Slides from https://www.cs.utah.edu/~lifeifei/papers/histogramSlides.pdf (Accessed 18 January 2018).

[2] Jestes, Jeffrey, Ke Yi, and Feifei Li. Building wavelet histograms on large data in MapReduce. Proceedings of the VLDB Endowment 5.2 (2011): 109-120.

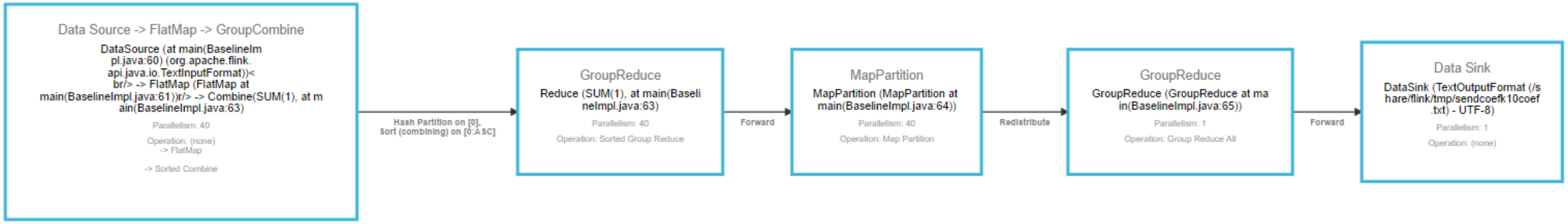
[3] http://ita.ee.lbl.gov/html/contrib/WorldCup.html (Accessed 18 January 2018)

Appendix

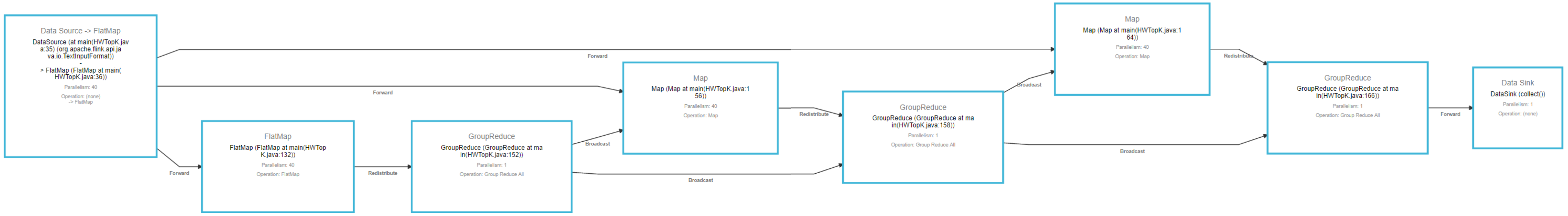
SendV Flink Job Diagram


Data Source -> FlatMap -> GroupCombine
DataSource (at main(BaselineFreqImpl.java:38) (org.apache.flink.api.java.io.TextInputFormat))
-> FlatMap (FlatMap at main(BaselineFreqImpl.java:39))
-> Combine(SUM(1), at main(BaselineFreqImpl.java:57)
Parallelism: 40
Operation: (none)
-> FlatMap
-> Sorted Combine
Hash Partition on [0], Sort (combining) on [0:ASC]
GroupReduce
Reduce (SUM(1), at main(BaselineFreqImpl.java:57))
Parallelism: 40
Operation: Sorted Group Reduce
Redistribute
GroupReduce
GroupReduce (GroupReduce at main(BaselineFreqImpl.java:58))
Parallelism: 1
Operation: Group Reduce All
Forward
Data Sink
DataSink (TextOutputFormat (/share/flink/tmp/sendvk10e4coef.txt) - UTF-8)
Parallelism: 1
Operation: (none)

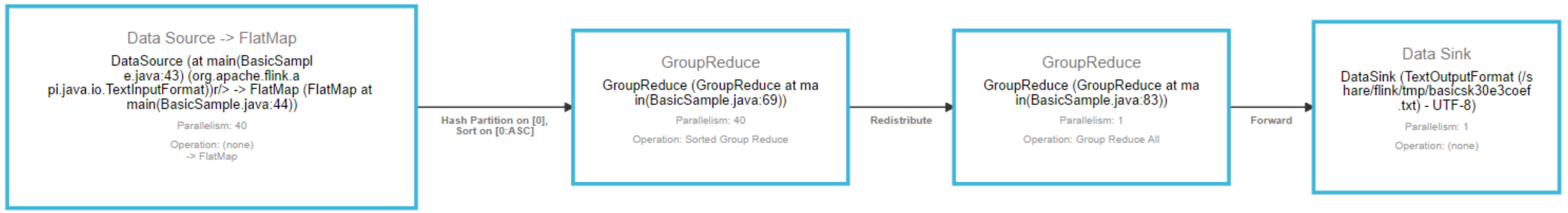
SendCoef Flink Job Diagram


Data Source -> FlatMap -> GroupCombine
DataSource (at main(BaselineImpl.java:60) (org.apache.flink.api.java.io.TextInputFormat))<br/> -> FlatMap (FlatMap at main(BaselineImpl.java:61))r/> -> Combine(SUM(1), at main(BaselineImpl.java:63)
Parallelism: 40
Operation: (none)
-> FlatMap
-> Sorted Combine
Hash Partition on [0], Sort (combining) on [0:ASC]
GroupReduce
Reduce (SUM(1), at main(BaselineImpl.java:63)
Parallelism: 40
Operation: Sorted Group Reduce
Forward
MapPartition
MapPartition (MapPartition at main(BaselineImpl.java:64))
Parallelism: 40
Operation: Map Partition
Redistribute
GroupReduce
GroupReduce (GroupReduce at main(BaselineImpl.java:65))
Parallelism: 1
Operation: Group Reduce All
Forward
Data Sink
DataSink (TextOutputFormat (/share/flink/tmp/sendcoefk10coef.txt) - UTF-8)
Parallelism: 1
Operation: (none)

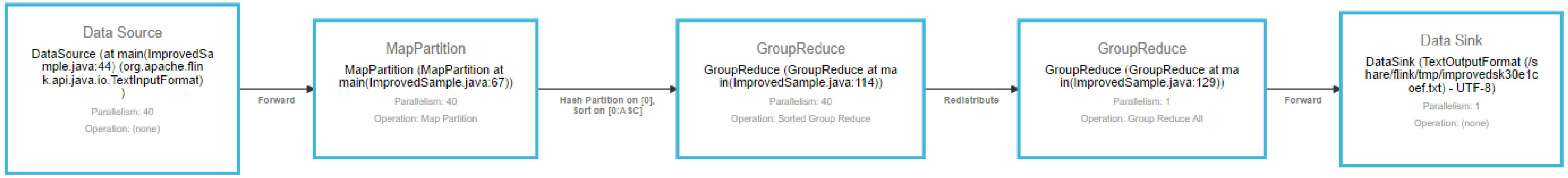
H-WTopK Flink Job Diagram


Data Source -> FlatMap
DataSource (at main(HWTopK.java:35) (org.apache.flink.api.java.io.TextInputFormat)) -> FlatMap (FlatMap at main(HWTopK.java:36))
Parallelism: 40
Operation: (none)
-> FlatMap
Forward
Forward
Forward
FlatMap
FlatMap (FlatMap at main(HWTopK.java:132))
Parallelism: 40
Operation: FlatMap
Redistribute
GroupReduce
GroupReduce (GroupReduce at main(HWTopK.java:152))
Parallelism: 1
Operation: Group Reduce All
Broadcast
Broadcast
Map
Map (Map at main(HWTopK.java:156))
Parallelism: 40
Operation: Map
Redistribute
GroupReduce
GroupReduce (GroupReduce at main(HWTopK.java:158))
Parallelism: 1
Operation: Group Reduce All
Broadcast
Broadcast
Map
Map (Map at main(HWTopK.java:164))
Parallelism: 40
Operation: Map
Redistribute
GroupReduce
GroupReduce (GroupReduce at main(HWTopK.java:166))
Parallelism: 1
Operation: Group Reduce All
Forward
Data Sink
DataSink (collect())
Parallelism: 1
Operation: (none)

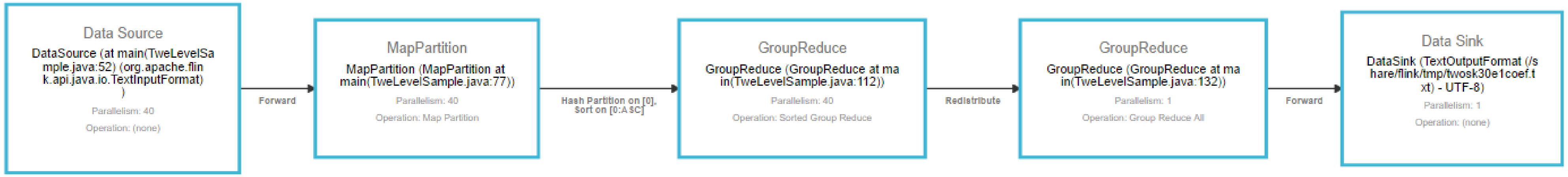
Basic-S Flink Job Diagram


Data Source -> FlatMap
DataSource (at main(BasicSample.java:43) (org.apache.flink.api.java.io.TextInputFormat))r/> -> FlatMap (FlatMap at main(BasicSample.java:44))
Parallelism: 40
Operation: (none) -> FlatMap
Hash Partition on [0], Sort on [0:ASC]
GroupReduce
GroupReduce (GroupReduce at main(BasicSample.java:69))
Parallelism: 40
Operation: Sorted Group Reduce
Redistribute
GroupReduce
GroupReduce (GroupReduce at main(BasicSample.java:83))
Parallelism: 1
Operation: Group Reduce All
Forward
Data Sink
DataSink (TextOutputFormat (/share/flink/tmp/basicsk30e3coef.txt) - UTF-8)
Parallelism: 1
Operation: (none)

Improved-S Flink Job Diagram


Data Source
DataSource (at main(ImprovedSample.java:44) (org.apache.flink.api.java.io.TextInputFormat))
Parallelism: 40
Operation: (none)
Forward
MapPartition
MapPartition (MapPartition at main(ImprovedSample.java:67))
Parallelism: 40
Operation: Map Partition
Hash Partition on [0], Sort on [0:ASC]
GroupReduce
GroupReduce (GroupReduce at main(ImprovedSample.java:114))
Parallelism: 40
Operation: Sorted Group Reduce
Redistribute
GroupReduce
GroupReduce (GroupReduce at main(ImprovedSample.java:129))
Parallelism: 1
Operation: Group Reduce All
Forward
Data Sink
DataSink (TextOutputFormat (/share/flink/tmp/improvedsk30e1coef.txt) - UTF-8)
Parallelism: 1
Operation: (none)

TwoLevel-S Flink Job Diagram


Data Source
DataSource (at main(TweLevelSample.java:52) (org.apache.flink.api.java.io.TextInputFormat))
Parallelism: 40
Operation: (none)
Forward
MapPartition
MapPartition (MapPartition at main(TweLevelSample.java:77))
Parallelism: 40
Operation: Map Partition
Hash Partition on [0], Sort on [0:ASC]
GroupReduce
GroupReduce (GroupReduce at main(TweLevelSample.java:112))
Parallelism: 40
Operation: Sorted Group Reduce
Redistribute
GroupReduce
GroupReduce (GroupReduce at main(TweLevelSample.java:132))
Parallelism: 1
Operation: Group Reduce All
Forward
Data Sink
DataSink (TextOutputFormat (/share/flink/tmp/twosk30e1coef.txt) - UTF-8)
Parallelism: 1
Operation: (none)