Evolutionary Computation Access on Incremental Map Reduce for Mining Large Scale Data

M. Blessa Binolin Pepsi, S.Haseena, S.Saroja

***Abstract:** In recent era, data updates arrive constantly from different areas like social network, finance, healthcare, e-commerce etc… Hence the data becomes large and computation on it becomes difficult. A framework for mining data earlyand to refresh the computed result with the new data arrival is proposed. The framework includes an incremental mapreduce method on hadoop with evolutionary computation algorithm for reduction in time complexity and increased accuracy. Proposed approach is a key pair level incremental iterative processing to Mapreduce for mining big data and uses particle swarm optimization to avoid re-computation from scratch on the new data arrived. Thereby the I/O overhead gets reduced for accessing predefined states. Experimental results were tested on three iterative algorithms in hadoop showed good performance compared to traditional mapreduce with sequential computation access.*

***Keywords:** Mapreduce, incremental iterative computation, big data, evolutionary computation, bipartite graph.*

I. INTRODUCTION

As large amount of digital data are arriving through important area like social media, health care, education, weather, e-commerce, finance etc…, there is a need to provide quality decisions by data miners working on the data to attain proper insights. There exists a large variety of frameworks to handle big data analysis. Among that is the Hadoop, a open source environment with HDFS for storage and Mapreduce for computation. The main focus of this proposed research is to modify and improve the MapReduce using incremental iterative approach and evolutionary computation.

Big data is continuously growing. When the new data arrives, the input data gets changed and so the data mining algorithm results will change over time. So, there is a need to refresh the mining results periodically for every computation [1]. For example page rank algorithm updates the ranking scores of the crawled web pages constantly based on graph structure. If the ranking results are not updated regularly then the quality of web search becomes low. In these kinds of cases, if the given input data is also large, then incremental processing is required for regularly updating the mining results than to rerun the entire computation from the beginning even if very small data has been updated.

The basic idea of incremental processing is to save the computation state of data after process. When a new data arrives, it will be better to reuse old state and do recomputation only for the states with modified data. This concept is applied in the mapreduce computing technique [2].

In previous studies, techniques like Stateful bulk processing for incremental analytics [6], Percolator [7], A timely dataflow system named Naiad [8] follows incremental processing but when applied on a big data there is a need to reimplement the algorithms. Mapreduce framework with incremental processing is completely different from that for handling especially big data. A framework Incoop[3], also supports incremental processing with mapreduce but it proceeds as a task level incremental processing. So, if any change in data then it will rerun the entire task thereby creating a huge number of redundant computations. In addition Incoop supports one step and not iterative computation. Every iteration will be treated as an individual mapreduce job.

Hence to avoid and overcome all facts, an evolutionary algorithm with incremental iterative processing on mapreduce framework is proposed. The novel features of the proposed system are, incremental processing using bipartite graph, iterative computation using change propagation mechanism and particle swarm optimization for optimizing the iterative computation with good quality.

Incremental processing uses bipartite graph for minimizing the re-computation. The key value pair data flow and its dependence are mapped as a bipartite graph. This is used to conserve the states in the graph for supporting efficient queries with incremental processing. Iterative computation combined with incremental processing proceeds with less number of updates by reusing the converged state of previous computation. Bipartite graph is used as store to be an enhanced support for incremental iterative processing and also employs change propagation control mechanism. Evolutionary computation algorithm is used for optimization process to identify the old data when new data arrives thereby reduces the time complexity of incremental iterative processing on mapreduce framework.

The technique was implemented in Hadoop 2.6 on three

Revised Manuscript Received on July 10, 2019.

M. Blessa Binolin Pepsi, Senior Assistant Professor, Department of Information Technology, Mepco Schlenk Engineering College, Sivakasi, Tamilnadu. India.

S.Haseena, Senior Assistant Professor, Department of Information Technology, Mepco Schlenk Engineering College, Sivakasi, Tamilnadu. India.

S.Saroja, Senior Assistant Professor, Department of Information Technology, Mepco Schlenk Engineering College, Sivakasi, Tamilnadu. India.

iterative algorithms (PageRank, Kmeans, Apriori). Experimental results prove that iterative evolutionary incremental mapreduce performs better compared to traditional mapreduce framework.

II. RELATED WORK

The basic mapreduce processing can work on multiple clusters. In 2004, Jeffrey Dean, Sanjay Ghemawat introduced a technique for data processing on large clusters [2]. The main contribution of the work is a simple and powerful interface that enables automatic parallelization and distribution of large scale computation combined with the implementation of this interface to achieve high performance on large clusters. This system is widely used and it is simple and mature.Usually, the input data is large and the computations have to be distributed across hundreds or thousands of machines in order to finish in reasonable amount of time. So,distribution, parallelization and failure handling leads to large amount of complex code. Hence in this system, a new abstraction is designed that allows expressing the simple computation and hides parallelization. The disadvantage of the system is that it does not have incremental approach, so it starts the re-computation from scratch

The next technique Incoop is a framework design for incremental mapreduce. In 2011, Pramod Bhatotia, Alexander Wieder, Umut A. Acer, Rafael Pasquini introduced a technique for incremental computations [3]. Incoop detects change to the inputs and enables the automatic update of the outputs by employing an efficient fine-grained result re-use mechanism. In Incoop, computations can respond automatically and efficiently to modifications to their input data by using intermediate results from previous computations and incrementally updating the output according to the changes in the input. To achieve this, it relies on memorization. Stable partitioning of input and reducing the granularity of tasks, maximize reusability. It employs affinity based scheduling techniques to increase the performance.

It consists of various design techniques that are incorporated in hadoop mapreduce framework. It includes incremental HDFS – provides mechanism to identity the similarities in the input data of consecutive job runs, contraction phase – it leverages combiner functions that is used to reduce the network traffic, memorization-aware scheduler – use affinity based scheduler that uses work stealing mechanism to minimize the amount of data movement across machines, use cases – two use cases are used: incremental log processing(to incrementally process logs) and incremental query processing(enable relation query processing on arriving data). The main disadvantage of the system is that it supports only task level incremental processing. So any changes in the result return the entire task and therefore the amount of computation increases.

In 2010, Jaliya Ekaenayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Quir, Geoffrey Fox proposed a runtime for iterative mapreduce computations known as Twister[4]. It uses publish/subscribe messaging infrastructure for communication and data transfer and support long running map reduce tasks. Most of the architecture focuses on performing single step map reduce. In these runtime, the repetitive application of mapreduce creates new map or reduce task in individual iteration. It leads to performance overhead for many iterative applications. But Twister extends programming model to support iterative mapreduce computation efficiently. It also provides programming extension to mapreduce with broadcast and scatter type data transfer. It allows twister to support iterative mapreduce computation compared to other mapreduce runtimes. It utilizes two types of data products - static and dynamic. The long running (cacheable) mapreduce task eliminates the necessity of reloading static data in each iteration. In this, to minimize the intermediate data values, it uses granularity of tasks. It uses combine operation to produce collective outputs from all the reduced outputs. Any changes in the input data lead to repeat the entire iterative algorithm again. The major disadvantage is that it does not support incremental computation.

Hence compared to existing work, the proposed system provides a framework to process in a incremental iterative approach including evolutionary computation for optimization thereby provides less time complexity.

III. OBJECTIVE

The goal of this work is to provide a MapReduce based framework for processing that reduces the runtime while refreshing the results of any algorithm. Other existing algorithm working on this platform requires re-computation on the entire data by this means increases the time complexity.

The method must satisfy these requirements:

- It enables an effective technique for incremental iterative computation.
- It provides an efficient optimization method for identifying the exact data to refresh using evolutionary computation approach.
- To reduce the time complexity on refreshing results of a mining algorithm

IV. PROPOSED SYSTEM DESIGN

The overall proposed work includes the following phases:

i) Incremental Processing
ii) Evolutionary Approach
iii) Iterative Processing

Generally the proposed system follows the preprocessing initially i.e. the special symbols present in the input data gets filtered. Then preprocessed data has to be given as input to Map phase. In the Map phase, the lines get tokenized. This is given as input to sort and shuffle phase where the tokenized words are arranged alphabetically and placed in store. In the Reduce phase, the process defined as per the problem.

Incremental Processing is proposed with the bipartite graph. Map reduce implementation illustrates that input to map function (K1,V1) where K1 is vertex id and V1 is represented as a1:w(i,a1),a2:w(i,a2)… where a is destination vertex and w(i,a) is weight value for the out edge (i,a). The shuffling phase combines the edge weights based on destination vertex. The reduce phase calculates the sum of all the in edges for vertex ‘a’ such that $\sum_a w(i,a)$. To preserve the edge of the states the input to the reduce task is represented by (K2,MK,V2). MK defines a globally unique value for every K2 since K2 is not unique by itself.

An example for bipartite graph with an initial and incremental processing is illustrated in Figure 1.

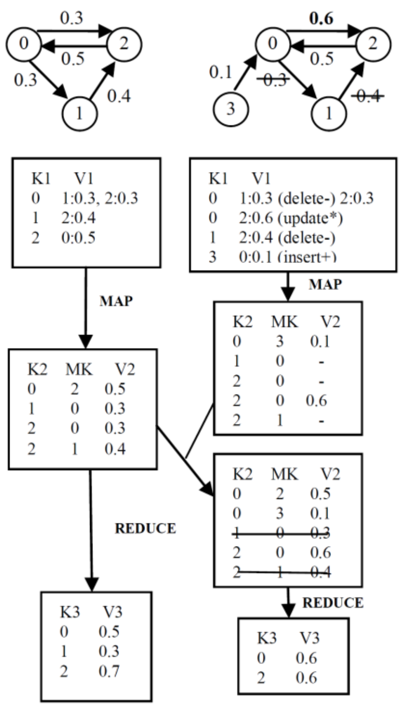


Fig. 1 Initial and Incremental run of Map reduce task

From the initial computation to incremental, the map input is based on the newly inserted, updated and deleted data. This can be represented along with V1 using any identifiable symbols like + for insert, * for update and – for delete. Hence the corresponding edges are computed in map phase as represented. The shuffle phase combines the intermediate values after updation along with initial run and places the result for reduce task thereby sorting. Finally reduce phase merges the result by checking for duplicates, inserts new edge, updates, deletes and does the task computation.

As shown in Fig.1 the symbol ‘+’ represents inserted key value pair, ‘-’ indicated deleted pair and ‘*’. In the given example, In the initial run, the bipartite graph has 3 unique vertices so includes 3 keys and their corresponding value pair is given by the representation source edge : weight [1:0.3]. As defined MK being defined as source edge i.e. from 2 to 0 the weight is 0.5. Similarly other values are listed. Finally in the reduce phase, the sum of all weights are mapped based on input key values V2.

The next step incremental run on a bipartite graph is explained as, the operations (delete, insert, update) to be performed on the graph are defined initially. Here there are 4 vertices since 3 is added as new vertex; hence initial map phase is built similar to previous initial run but after performing the operations. The main point of merging the operations i.e. incremental processing with the already preserved graph is done. The mapper gets updated with deleted, updated and inserted operations combining preserved and new graph. This doesn’t modify the entire content but rather preserves the original store thereby reducing the complexity of computation time. Finally the reduce phase does the process of calculating the sum of weights

This task computation of incremental processing maintains a store for saving and retrieving the states. This maintains an index file to get the chunk position rather to store the entire bipartite graph result. For the updated input data, the new store given as delta store is created. Here ‘-’ is given for deleted data and ‘+’ for newly added data. For the inserted key value in the delta store, the corresponding chunk position is obtained using index file and the word gets inserted in it.

If ‘-’ is present in value field, then traverse to the location provided by index file and delete it. Otherwise, insert the word to the store using index file. For deletion, the corresponding key value is obtained from the delta store, chunk position is obtained and the word gets deleted from the store. Hence the store includes merged state with newly computed edge values.

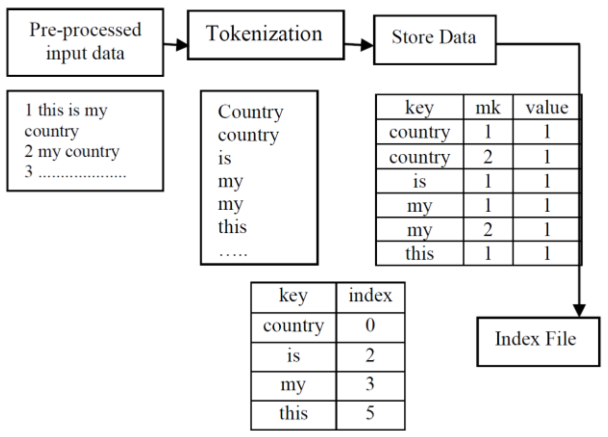


Fig. 2 Structure of the store and index file

The above given Fig.2 descript the steps for the creation of store and the corresponding index file for processing. The delta store further identifies the unique value and updates as delta store. ‘mk’ represents the mapped key.

For the delta store updation, ‘+’ symbol indicates new key value pair and ‘-‘symbol refers to deleted pair. The concept used is similar to bipartite graph updation [ ].

Once the task is defined in incremental run, the algorithm have to move through the key values sequentially in search of index thereby increasing the computation time when processed on a large scale data. So, to avoid a sequential approach evolutionary computation method i.e. particle swarm optimization is proposed. This is a method that optimizes the problem iteratively to define a best candidate solution.

In particle swarm optimization, the search space for the problem is to identify the set of key values from the large data space being big data to perform the task in reduce after merging the results from preserved and new graph. The key value is defined as ‘particle’ and the fitness function is fixed to be the similarity measure to identify the optimized solution. When the particle takes part in the large space as its topological neighbors, the best is found to be global best i.e. optimal value. The probability is fixed as random set with random particles as initial solution to search for optima. The velocity and positions are calculated for every iterated result until maximum data space is met.

The querying data from the store can be stated using the following algorithm with the evolutionary approach. Algorithm 1 defines the algorithm to query a chunk store of key values k from the given large space of list of keys L using PSO algorithm.

Initially k be the particles in the population, if the chunk k does not reside in the read cache i.e. similarity measure for fitness calculation, it will compute the read C2 byte into cache and read window size as C1. They are taken as learning factors for the PSO. The gap between a set of junks L_i to L_{i+1} is framed as total C2. These values are included to calculate velocity. The velocity is compared with Vmax. If less then iterates until it retrieve the set of best chunkstore as result. Hence the algorithm includes population as list of key chunk, fitness function as similarity measure to read chunk, fitness value to compare with threshold Vmax and finally output the optimized chunk store is returned at minimized cost.

Algorithm 1. // to store words in MRBG store and search optimized key using PSO
Input: queried key: k; the list of queried keys: L; Velcocity max V_{max}
Output: chunkstore k

```
K- particle
initialize random population with probability of iterations
for each particle
    if !read_cache.contains (k)//fitness function then
      c1←0, c2←0 //Learning factors
      i←k's index in L // $L_i$ = k
      while c1< T and c1+c2+length ($L_i$)
                                  <read_cache.size
        do
          //update the new fitness value
          C2←C1+C2+length ($L_i$)
          C1←pos ($L_{i+1}$)-pos ($L_i$)-length ($L_i$)
          //Calculate velocity
          v = v + C1* rand() * (bestK - presentK) + C2* rand() * (bestK - presentK)
          if V is not $V_{max}$ //iterate
            i←i+1
            end while
          starting from pos (k), read w bytes into read_cache
    end if
end for
    return read_cache. getchunkstore (k)
```

This incremental evolutionary approach is applied on iterative algorithms like PageRank, Kmeans and Apriori with the proposed iterative computation approach.

Algorithm 3: PageRank - // to find the rank of web pages.

i – Source id
j – Destination id
N_i – Number of neighboring nodes of i
N_j – Number of neighboring nodes of j
R_j – Rank of node j
d – Damping factor

```
Map Phase :
input: <i, Ni|$R_i$>
Output:<i,$N_i$>
 for all j in $N_i$
do
        $R_{i,j}=R_i/|N_i|$
        output <j, $R_{i,j}$>
 end for
Reduce Phase :
input: <j,{$R_{i,j}$,$N_j$}>
$R_j=d\sum_i R_{i,j}+(1-d)$
output : <j,$N_j$|$R_j$>
```

Algorithm 4: kmeans // to cluster the given data
cid – Cluster id
cval – Cluster value
pid – Point id
pval – Point value

```
Map Phase :
input: <pid,pval|{cid,cval}>
cid←find the nearest centroid of pval in{cid,cval}
output:<cid,pval>
Reduce Phase :
input: <cid, {pval}>
cval←compute the average of {pval}
output: <cid,cval>
```

Algorithm 5: Apriori // It is used to mine the frequent itemset.
Input: data
Output: Combinations greater than support count

```
Begin
        Get input file
  Tokenize each transaction
  Find word count [sup_count for each word= its word count]
        for each item
        {
                if  (sup_count>=given sup_count)
                    Add word to the resultset
        }
        for each item at resultset
        {
          Find two item combinations
          Get its sup_count from sup_count file generated
                if (sup_count>=given sup_count)
                        Add to the resultset
        }
        for each item at resultset
        {
                Find three item combinations
```

```
            Output the result
        }
End
```

V. RESULTS AND DISCUSSION

This prototype was implemented in Hadoop 2.6 version including Mapreduce API. The experiment compares the plain mapreduce execution with the incremental mapreduce on three different iterative algorithms (Pagerank, Kmeans and Apriori) to test its reduced computation.

Every algorithm includes different dataset for its computation. For the three iterative algorithms, the delta input is generated by randomly changing 10 percent of the input data.

A. Apriori :

Initially the Apriori algorithm is implemented on a grocery store data of size 100GB. In the preprocessing step, the unique line number is added to each line which acts as an mk value. From the data each transaction can have one or more items that are purchased. Different combinations of item are given for each transaction.Hence n item in a transaction includes 2^n-1 combinations.

For example, the transaction - Wholemilk, butter, yogurt includes 2^3-1=7 combinations.

- wholemilk
- butter
- yogurt
- wholemilk butter
- wholemilk yogurt
- butter yogurt
- wholemilk butter yogurt

The items in each combination are shuffled to be arranged in an alphabetical order. With this data, store is created by pre-processed lines are tokenized. And each word is added to the Store provided with its key (word), mk (line number) and value (its count), As an example,

Butter, 1, 1
Butter, 3, 1

The index file is created for the Store. This is to find the position of a word for its insertion or deletion in the file. Now when a query is given into Store, the new lines are added and the existing lines are deleted from the input file.

Creation of delta store from the input file is taken from the data updated as input. If a line gets deleted, find all combinations of the transaction and add to delta store with its value provided as '-'. If a new line is added, the line is pre-processed to get transaction id (mk). Different combinations of the transaction are added to delta store with its value as '+'.

From the delta store created, searching from the position marked by start to the length in file based on the user query. Searching includes particle swarm optimization to identify the data at reduced cost of time. Then the data combinations identified from store are shuffled and reduced.

If the combinations of items with the support count referred at the file created above remains greater than or equal to the given support count, the combinations are given as output

B. Kmeans :

Population data of size 14.5GB is taken with the attributes as country and its population for the past four years. The data is pre-processed by removing ',' and by calculating average population of the country for the past four years. The goal is to determine the most and least populated countries using kmeans on a large data.

Each iterative algorithm includes a state kvpair and a structure kv pair. In state kv pair, the value for state key gets changed after each iteration. Structure kv pair remains same for all iterations. Kmeans and PageRank are both incremental and iterative algorithms.

For Kmeans,

- The structure kv pair includes key as a country name and population as its value.
- The state kv pair includes centroid unique id and value

Structure pair does not change but a state key value pair changes for every iteration based on the centroid values.

The concatenated state and structure file are given as input to the map phase. For each pval (population) in the state file,

- For each centroid in structure file, calculate |cval-pval|
- Find the nearest centroid to the pval
- Output the nearest centroid id and the population to the reduce phase.

In the iterative reduce phase, For each cid, compute the average for its population grouped to it.

- Update it to the state file.
- With this updated state file and an unchanged structure file, the iteration runs for the desired number of times.
- Updated state data

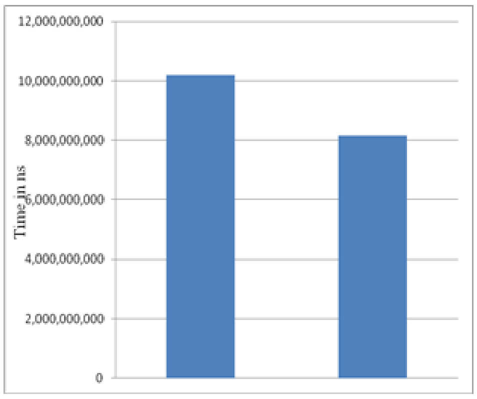
The store is created at the map phase of incremental approach with the data of cluster id (key), country name(mk), population(value). The index for is also created to find the position of a word for its insertion or deletion.

From the input file, the data updated is taken as input. If a country gets deleted, the cluster to which the country belongs is computed and inserted to delta storewith its key as cluster id, mk as country name and value as '-'.

If a new country is added, the line is pre-processed and the average population is computed. The cluster, to which the country belongs, is computed.

It is inserted to delta store with its key as cluster id, mk as country name and value as population. Finally searching from the position marked by start to the length in the index file to retrieve the country based on given query is done with reduced cost.

The experimental analysis of the kmeans is given by,



Incremental MR

C. PageRank : Plain MR

The wikitalk data of size 16 GB is taken as input with the features source and destination ID. Create state and structure input file. State file includes node and rank as key value pair. Structure file has node and number of neighbouring nodes as key value pair.

The concatenated state and structure file are given as input to the map phase. For every source node, the number of its neighbours from the structure data and rank from the state data are given as input. For each destination of the given source i.e, j in N_i compute $rank_{i,j}=R_i/(N_i)$.

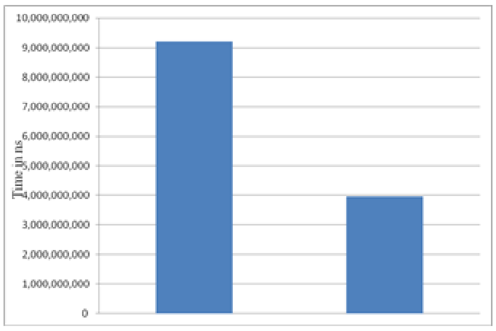
For example, if source node 2 has 3 neighbouring nodes, compute its rank $R_{2,j}=1/3$(the initial rank for all the nodes are 1).

Output the destination node j and the computed rank R_{ij} to the iterative reduce phase.

For each input j, R_{ij} values compute $R_j=d\sum_i R_{i,j}+(1-d)$

d is the damping factor usually given by the value 0.85. Update the computed result j, R_j to the state file. This is continued for required number of iterations.

The store is created at the map phase of incremental approach with the data destination as a key value, source as the mk and rank as the value. Similarly, delta store is created on updating nodes and are updated to the file using index file. The rank values are stored at state file used for updation. The basic steps for incremental approach (insert and delete) are used for updation. The experimental results of the pagerank is given by,



Plain MR Incremental MR

VI. CONCLUSION:

Hence, MapReduce-based framework for incremental big data processing is proposed. This proposed work builds an incremental engine, an iterative model to reduce the runtime while refreshing the big data mining results compared to re-computation with plain mapreduce. The incremental approach used here reduces the time for re-computation of stored results using bipartite graph. The iterative approach makes use of state and structure file. In many cases, the convergence of the state file remains same reducing the time for computation. The above two approaches along with evolutionary optimization method was applied to the PageRank, Kmeans and Apriori algorithm that shows better performance than existing plain mapreduce algorithm. Particle swarm optimization evolves an optimized chunk of queried keys as a support for incremental iterative approach. The future work can be extended for real time large space streams since the time complexity of analysis has high impact.

REFERENCES:

1. S. Brin, and L. Page, “The anatomy of a large-scale hypertextual web search engine,” Comput. Netw. ISDN Syst., vol. 30, no. 1–7, pp. 107–117, Apr. 1998.
2. J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in Proc. 6th Conf. Symp. Opear. Syst. Des.Implementation, 2004, p. 10.
3. P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, “Incoop: Mapreduce for incremental computations,” in Proc. 2nd ACM Symp. Cloud Comput., 2011, pp. 7:1–7:14.
4. J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, “Twister: A runtime for iterative mapreduce,” in Proc. 19th ACM Symp. High Performance Distributed Comput., 2010, pp. 810–818.
5. U. Kang, C. Tsourakakis, and C. Faloutsos, “Pegasus: A peta-scale graph mining system implementation and observations,” in Proc. IEEE Int. Conf. Data Mining, 2009, pp. 229–238.
6. D. Peng and F. Dabek, “Large-scale incremental processing using distributed transactions and notifications,” in Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation, 2010, pp.1-15.
7. D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum,“Stateful bulk processing for incremental analytics,” in Proc. 1st ACM Symp. Cloud Comput., 2010, pp. 51–62.
8. D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, “Naiad: A timely dataflow system,” in Proc. 24th ACM Symp. Oper. Syst. Principles, 2013, pp. 439–455.
9. Y. Zhang, S. Chen, Q. Wang, and G. Yu, “i2mapreduce: Incremental mapreduce for mining evolving big data,” CoRR, vol. abs/1501.04854, 2015.
10. Y. Zhang, Q. Gao, L. Gao, and C. Wang, “Priter: A distributed framework for prioritized iterative computations,” in Proc. 2nd ACM Symp. Cloud Comput., 2011, pp. 13:1–13:14.
11. R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases,” in Proc. 20th Int. Conf. Very Large Data Bases, 1994, pp. 487–499.
12. J. Cho and H. Garcia-Molina, “The evolution of the web and implications for an incremental crawler,” in Proc. 26th Int. Conf.Very Large Data Bases, 2000, pp. 200–209.
13. C. Olston and M. Najork, “Web crawling,” Found. Trends Inform. Retrieval, vol. 4, no. 3, pp. 175–246, 2010.
14. S. Lloyd, “Least squares quantization in PCM,” IEEE Trans.Inform. Theory, vol. 28, no. 2, pp. 129–137, Mar. 1982.