Future Generation Computer Systems 80 (2018) 102–111

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Efficient finer-grained incremental processing with MapReduce for big data

Liang Zhang [a], Yuanyuan Feng [a], Peiyi Shen [a], Guangming Zhu [a], Wei Wei [c,*], Juan Song [a], Syed Afaq Ali Shah [b], Mohammed Bennamoun [b]

[a] School of Software Engineering, The University of Xidian, China
[b] School of Computer Science and Software Engineering, The University of Western Australia, Australia
[c] School of Computer Science and Engineering, Xi'an University of Technology, Xi'an 710048, China

HIGHLIGHTS

- Illustrate the shortcoming of coarse grained result reusing for incremental processing.
- An algorithm to divide input datasets stably and quickly.
- Optimize the procedure of finding delta data and deliver an efficient and stable implementation.

ARTICLE INFO

Article history:
Received 25 February 2017
Received in revised form 31 May 2017
Accepted 30 September 2017
Available online 20 October 2017

Keywords:
Big data
Incremental processing
Finer grained reusing
Yarn

ABSTRACT

With the continuous development of the Internet and information technology, more and more mobile terminals, wear equipment etc. contribute to the tremendous data. Thanks to the distributed computing, we can analyze the big data with quite high speed. However, many kinds of big data have an obvious common character that the datasets grow incrementally overtime, which means the distributed computing should focus on incremental processing. A number of systems for incremental data processing are available, such as Google's Percolator and Yahoo's CBP. However, in order to utilize these mature framework, one needs to make a troublesome change for their program to adapt to the environment requirement.

In this paper, we introduce a MapReduce framework, named *HadInc,* for efficient incremental computations. HadInc is designed for offline scenes, in which real-time is needless and in-memory cluster computing is invalid. HadInc takes the advantages of finer-grained computing and Content-defined Chunking(*CDC*) to make sure that the system can still reuse the results which we have computed before, even if the split data has been changed seriously. Instead of re-computing the changed data entirely, *HadInc* can quickly find out the difference between the new split and the old one, and then merge the delta and old results into the latest result of the new datasets. Meanwhile, the dividing stability of the datasets is a key factor for reusing the results. In order to guarantee the stability of the dataset's division, we propose a series of novel algorithms based on *CDC*.

We implemented *HadInc* by extending the Hadoop framework, and evaluated it with many experiments including three specific cases and a practical case. From the comparing results it can be seen that the proposed *HadInc* is very efficient.

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

The thriving of big data is to benefit from the Internet and mobile communication technology. In our daily life, different kinds of devices collect information round the clock. The servers receive and analyze such information by using some intelligent methods like machine learning, and return the results to users [1]. To this end, we have to face some specialties of the big data, such as large scale [2], incremental or dynamic data changing, immediately user response, and so on.

As times passes, the datasets accumulate and continuously grow in size. If the fresh results are required periodically, in view of the enormous size of datasets, it is unadvisable to re-compute all the datasets at each time [3]. Re-computation will not only cost much time which will decay timelines of the results, but also affect the commercial value of the datasets. For example, when people surf websites, the most action they do is click [4] which

* Corresponding author.
E-mail address: wxw162730@utdallas.edu (W. Wei).

https://doi.org/10.1016/j.future.2017.09.079
0167-739X/© 2017 Elsevier B.V. All rights reserved.

is unique and immutable in the timeline. If we recalculate all the datasets at each time for the browsing information, the reduplicative and useless calculation is beyond 85%. Besides, because of the modification on old data which can make the computation more complex, the schema for processing incremental datasets should be enough flexible and must guarantee a stable efficiency in different incremental cases.

In this paper, we propose an incremental processing system named HadInc, which simultaneously takes the characters of big data and incremental processing into consideration. HadInc performs efficiently both in appending and modifying cases of the big datasets, which means HadInc can be applied to varying incremental processing scenes with impressive stability. We will introduce the HadInc around following properties:

Stability: Some of the existing incremental processing frameworks always divide the datasets into splits with CDC [5] just in Preprocessing, which means they can only reuse the pre-results in a coarse grain. The other frameworks like *HadUP* [6] divide the datasets by a fixed length chunking. *HadUP* can overcome the coarse-grained reusing problem, but it is not stable enough because of the dividing schematics. When the datasets changes heavily at some specific parts, the dividing result can be totally different from last division, which will increase the calculation load sharply. In contrast, *HadInc* differs from those traditional frameworks in that it takes advantage of *CDC* both in *Preprocessing* and MAP tasks, which results in a more stable chunking procedure.

Finer-grained: In order to cut down the calculation budget and get the delta dataset(change data between new and old inputs of the big data) as soon as possible, we should take a finer-grained schema to reuse as many of the previous results as possible. *HadInc* makes this happen in MAP task by associating the changed split in new datasets to its related split in the old datasets. Next, *HadInc* constructs those two splits into a finer-grained data structure called segment with *CDC*, and then divide each segment into a lot of chunks with fixed-length chunking. Finally, results of the delta datasets are utilized, which are figured out by analyzing the associated two splits, and the previous results of old split to produce results of the new datasets. By these steps, the framework does not have to re-compute all the split data, some of which are probably not changed, and we can get the delta data at runtime in demands.

Efficiency: Data chunking is a time-consuming process. Instead of chunking all the splits with finer grain, *HadInc* avoids doing this work in *Preprocessing*, because at that point we do not know which split is changed and which one not, so that we have to divide all the datasets finely. Obviously, this is inconsiderable. Because a lot of splits are "clean", we can use their previous results directly. Since *HadInc* could recognize the "dirty" splits which will be divided by *CDC*, therefore, we move the finer dividing job, which is responsible for cutting the changed split into several segments, to the MAP task In addition, we need to cut these segments into a lot of chunks for figuring out the delta datasets It is important to note that if the chunk size in each segment is too small, chunking will be quite time consuming. To avoid this, we employ the fixed-length chunking strategy in every segment. It is proved that the schema can make *HadInc* quite efficient.

This paper is organized as follow: Section 2 describes the background of incremental processing and related work; Section 3, illustrates the overview of *HadInc* design (Section 3.1) and its implementation (Sections 3.2 and 3.3); Section 4, evaluates *HadInc* with many cases and present the experiment results using different frameworks to demonstrate the benefits and advantages of the proposed *HadInc*

2. Background

Big data enables us to quickly make some decisions. We should dig out the valuable information as soon as possible, otherwise not only the time will be lost, but also the worth value hiding in the big data will be discarded.

Traditional solutions, which re-compute all the datasets, are most hardly applied in incremental processing, as the size of big data increases with an incredible speed. Recently, some novel solutions [7–18] based on incremental processing have appeared, and many of them work well in some specific cases, such as incremental appending. However, those solutions are either complex frameworks, which are short of compatibility to other systems, or not generally applicable enough.

In this section we focus on some existing approaches and frameworks that are designed to process incremental datasets.

2.1. Complex incremental processing frameworks

Nowadays, there are some mature frameworks used to process incremental big data, such as Google's percolator [19] and Yahoo's *CBP* [20].

Google is the most popular and successful searching engine all over the world. While people enjoy the abundant searching results from the increasing datasets, the servers of searching engine are faced with huge computation load. Before Percolator shows up, when the engine updates the index of websites, it should batch compute all the datasets including old and new data uploaded from everywhere on world wide web. This usually takes several days to complete the update. The tremendous overhead decreases the timeliness, therefore, people cannot get the fresh data uploaded a few hours ago. Percolator is designed to solve this problem. When the system applies Percolator in generating websites searching index, by replacing the batch computing, the overhead is decreased by 50%.

Yahoo uses *CBP* system to process daily photos and videos uploading, logs analyzing, websites index updating etc. *CBP* contributes through decreasing data migration and bandwidth load to process incremental big data. *CBP* makes migration status of data as input of the system, and utilizes parallel computing to unify the status of program, so that the lower-class system can migrate data as few as possible. That is the base theory of *CBP* to cut down the overhead of incremental data processing. Researchers evaluated *CBP* with some actual datasets, and the result shows that *CBP* can decrease 46% data migration and 50% time cost in PageRank test.

The above frameworks look like a perfect solution to process incremental datasets, however, it is not an easy job. Taking Percolator for example, it is based on Google's three big frameworks named Bigtable [21], GFS [22], MapReduce [23]. To take advantage of Percolator, the compatibility between the old nonincremental system and Percolator must be implemented. The only way is to modify your system, which is difficult to accept in most cases.

2.2. Simplex incremental processing models

Besides those complex distributed frameworks for incremental datasets processing, there are some simplex models, such as *Incoop* [24], *IncMR* [25], *Hourglass* [26], *MapReuse* [27], *HadUP*, *I^2MapReduce* [28], etc.

Incoop, which can analyze the relationship between old and new datasets, is based on *Hadoop* 1.X [29] and is designed to accelerate the procedure of incremental datasets processing. Before *MapReduce*, *Incoop* divides the datasets into a lot of splits with several threads working together. At initial round, *Incoop* calculates all the datasets and uploads the information with key value pairs, in which key is md5 code of the task input and value is the output of each task. At subsequent round, every task will query the Memorization Server where the information is uploaded. If the split has been computed before, which means we can find the result at Memorization Server, *Incoop* takes the result from HDFS directly. Otherwise, *Incoop* will compute the "dirty" split and upload the new information to Memorization Server.

Hourglass is another open source library for incremental calculation. It has been applied to some popular websites like LinkedIn [30], and it benefits from machine learning. *Hourglass* is an outstanding model for analyzing and processing sequential datasets. All the input data is classified by date. At initial round, *Hourglass* computes the total datasets accumulated day by day. At subsequent round, the system calculates the new sequential data iteratively every day or week. Then, *Hourglass* combines the new and old results in reduce tasks to figure out the output of the current datasets

Different from the above mentioned models, *HadUP* is a finer-grained incremental processing model. *HadUP* can find out the delta data between old and new splits by a novel algorithm named D-SD which is based on the sparse indexing technology [31], so that the system can avoid calculating all the data of the changed split. In subsequent round, in order to collect the delta data, primarily, *HadUP* analyzes the changed split and its relevant split in old datasets. Then *HadUP* submits the delta data to the context of *Hadoop* job. Finally, it combines the results of delta data and previous results into current new input.

In conclusion, in order to utilize the existing frameworks to fulfill our demands, we either configure a complex environment to take advantage of Percolator or *CBP*, or choose a kind of simplex model for one specific case. In fact, *Incoop* and *IncMR* are good at processing appending datasets. In the mean time Hourglass are skillful in processing sequential datasets, and *HadUP* is a good choice when the delta data is not too big. In a word, currently, we need a universal model to process all kinds of cases in incremental datasets with a stable efficiency.

3. HadInc: A universal finer-grained incremental processing programming model

In the previous section, the requirements of a framework that would support the needs of incremental datasets processing have been highlighted. In this section, we introduce *HadInc*, a programming model designed to address these requirements, and then discuss its implementation.

3.1. Overview

Fig. 1 shows the structure of *HadInc* system which consists of four parts:

- *Preprocessor*: To prepare for incremental processing, first, *Preprocessor* gets the basic information of datasets from HDFS. It then starts many threads based on the amount of the machine cores, and divide the datasets into splits by *CDC* in different ranges respectively. Finally, it generates split information file named *RangeInfo* and uploads that into HDFS.
- *ResultRecycler*: It is designed to recognize unchanged splits and reuse the previous results, instead of computing repeatedly. First, we start several map tasks as the number of splits. The input of each task is a piece of range information by which we can download the split data from HDFS. At initial round, *HadInc* calculates all the datasets and records the result of each task, then writes the finger print of splits and the directory of the result into a structure named *ResultMemory*. At subsequent round, map task checks whether the input split has been calculated in *ResultMemory*, if so, it takes the result from HDFS directly. Otherwise, first we call *FinerGrainedDivider* to divide the split and its relevant split in old datasets into a smaller data structure named segments. Then, *DeltaFinder* will find out the delta data between the new and old splits, and merge the result of delta data and previous associated split's computation result. Finally, we update the *ResultMemory* and the new result into HDFS.
- *Reduce*: The new task results will be shuffled from map tasks to reduce tasks. Then, reduce tasks collect these data and output the final results of the current datasets
- *Postprocessor*: After a round of subsequent calculation, there are some useless data, for example, the information in *ResultMemory* which we did not read last time and will not be read anymore, such information should be deleted along with the task results it points to.

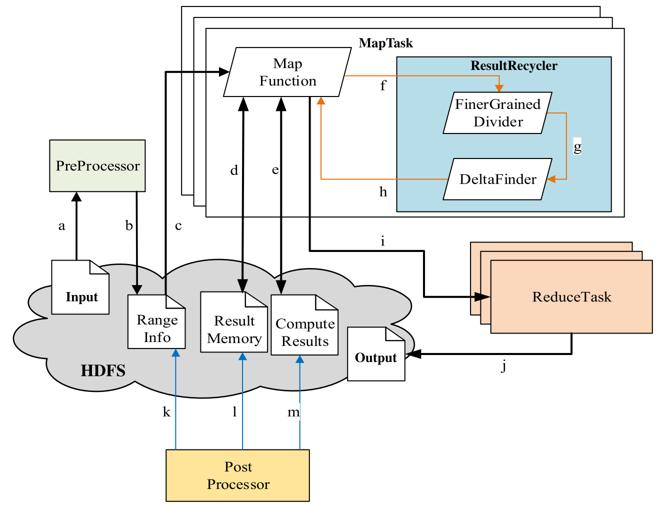


Fig. 1. The structure of HadInc system. In initial round, we compute all the splits and save the task results. In subsequent rounds, reusing previous results is the key to cut down time cost.

In initial round, (a) *Preprocessor* gets the basic information of the input from HDFS. (b) *Preprocessor* uploads the *RangeInfo* file to HDFS which records the dividing information of the datasets. (c) Map tasks read a piece of RangeInfo and then download the split data. (d) Map tasks figure out the md5 code of the split data and search *ResultMemory* for the md5 code. If it is existing, it will take the result according to the information, otherwise it computes the split data and uploads the result. (i) Map tasks shuffle the task result to reduce tasks. (j) After collecting the data, *HadInc* outputs the final result to HDFS. (k) Finally, in order to serve the next round, *Postprocessor* transfers *RangeInfo* into the *LastRangeInfo*. (l) *Postprocessing* deletes the useless information in *ResultMemory*. (m) *Postprocessing* deletes the task result files which were not accessed last time.

In subsequent round, most of steps are same as the initial, except for the procedure of finer-grained results reusing. (f) If the md5 code is not found in *ResultMemory*, *HadInc* divides current split and the relevant split in old datasets into segments by *CDC*. (g) *DeltaFinder* analyzes these two groups of segments, and then finds out the delta data. (h) Map tasks is responsible for merging the results of delta data and previous results into the new task result, and transfers the result to HDFS and reduce task in step (e) and (i).

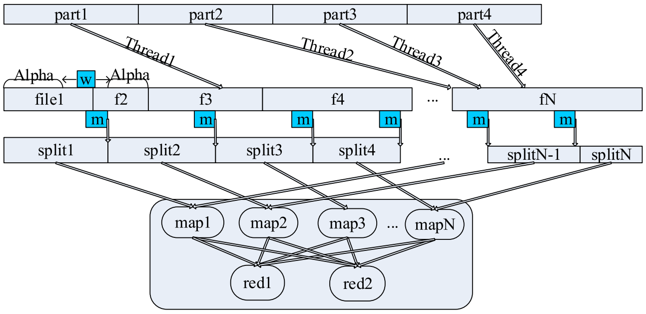


Fig. 2. Procedure of Preprocessing. Based on content-defined Chunking, HadInc divides input datasets into many splits. Each split is a input for map task.

3.2. *Preprocessor*

The key to recognize the "clean" data, which can reuse previous results, is that we should find a quick and stable way to divide input data into smaller bulks. The smaller bulk can ensure that more bulks are clean. However, too many bulks will increase the overhead heavily, therefore, the *Preprocessor* must make a compromise between stability and speed.

There are two popular methods to divide data, one is fixed-length chunking and another is *CDC*. Fixed-length chunking can efficiently process the scenario where the most data change is appending. However, this method is not stable enough when a lot of insertion or modification takes place, as they affect the previous dividing results seriously. *CDC* is a complex dividing method which consumes more memory and time, however, it offers an impressive stability for tolerating insertion and modification actions. *HadInc* proposes a novel method, called sliding detection matching algorithm, which divides the input data for reusing the previous results based on *CDC*. Meanwhile, this algorithm can guarantee that the sizes of those divided splits are very close to each other, which can avoid decreasing the load balance of *MapReduce*.

Fig. 2 shows the procedure of *Preprocessing*. First, according to the core number of the servers, *HadInc* starts several threads to find the split positions in different ranges. We set the size of split as *SS*, for example 16 MB. The minimum splits number of each thread to process is *BA*, and the rest splits number is *EA*. For example, the size of input datasets is *S*, and there are *N* threads alive, then $BA = (S/SS)/N$, $EA = (S/SS)\%N$. The last split of input is definitely smaller than *SS*, therefore the last *EA* threads additionally process those rest splits. Next, *HadInc* download $(1-alpha)*SS$ data at the end of each split forward and backward respectively (*alpha* should be smaller than 1), and the length of this data *MS* is equal to twice of $(1-alpha)*SS$, which is the range of the sliding window.

After that, *HadInc* moves the sliding window (the size of sliding window can be decided by user, it is smaller than *MS*, typical 10 K bytes) from the end of a split. If the Rabin code of this window matches the mode we set before, this window is recognized as a mark that is used to divide the datasets. Otherwise, *HadInc* moves the window reversely, until we find a valid dividing position or run out the *MS* data. We describe this algorithm in Table 1 and the time complexity O(*SplitNum***MoveCount*).

In order to avoid dividing an atomic data into two parts, the dividing positions should be adjusted. At each position, a separator is found backward as a final dividing position, and the information is written into *RangeInfo*.

3.3. *ReslultResycler*

There are two kinds of result reusing methods. One is coarse-grained method, which is suitable to the conditions that most of the

Table 1
Sliding detection matching algorithm.

Input：
Input Path p_{in}, Range start s_r, Range length l_r
Output：
split list = $\{\mathbf{S}_{sp}\}$
$\mathbf{D}_{ran} \leftarrow F_{getData}(p_{in}, s_r, l_r)$
Split $\mathbf{D}_{ran}$ into pieces $\{\mathbf{S}_{init}\}$ with fixed length *SS*
for all split∈$\{\mathbf{S}_{init}\}$ **do**
  moveCount = *SS* *(1-alpha)–*MS*+1;
  Get head data $\mathbf{D}_{head}$, the tail of last split
  Get tail data $\mathbf{D}_{tail}$, the head of next split
  Current split has no splitor : flag← false
  for k=0 : moveCount **do**
    Put *MS* data from $\mathbf{D}_{head}$ into window
    fingerPrint←F_{Rabin}(window)
    if F_{match}(fingerPrint, mode) **then**
      $\mathbf{S}_{sp} \leftarrow \mathbf{S}_{sp}$ U window
      flag ← true
      break;
    end if
    Put *MS* data from $\mathbf{D}_{tail}$ into window
    fingerPrint←F_{Rabin} (window)
    if F_{match}(fingerPrint, mode)) **then**
      $\mathbf{S}_{sp} \leftarrow \mathbf{S}_{sp}$ U window
      flag ← true
      break;
    end if
  end for
  if flag == false **then**
    $\mathbf{S}_{sp} \leftarrow \mathbf{S}_{sp}$ U window ▹this window is the end of the split
  end if
end for
end function

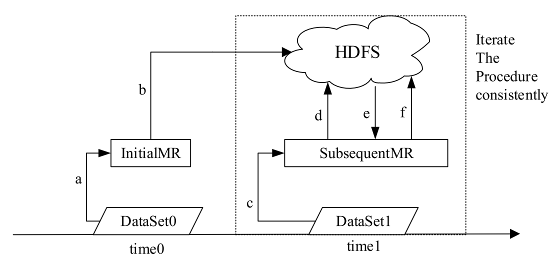
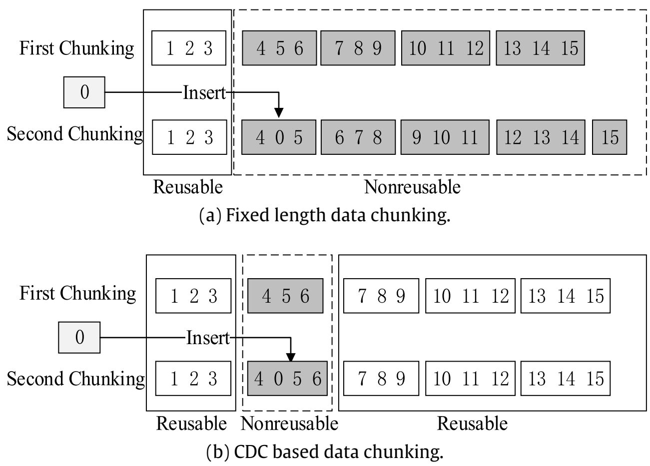


Fig. 3. Subsequent rounds utilize the results produced by previous calculation to decrease the overhead.

task inputs are not changed. Another is finer-grained presented in *HadUP*, which can recognize the delta data between old and new splits, and then merge the previous result with delta data. However, the mentioned methods are either inefficient or unstable. In Section 3.3.1, an implementation of a more stable finer-grained dividing strategy will be described. In Section 3.3.2, a method to find the delta data between old and new splits will be discussed.

Fig. 3 illustrates that when users submit a job, *HadInc* checks the history data. If the history is null, then this round is declared as initial otherwise it is considered the subsequent round.

In initial round, (a) Map tasks receive a split information which contains the file path, offset and length of the split. Map tasks download the split from HDFS and figure out the result. (b) After computing, *HadInc* restores the result in HDFS, and calculates the



(a) Fixed length data chunking.

(b) CDC based data chunking.

Fig. 4. Two algorithms for data chunking.

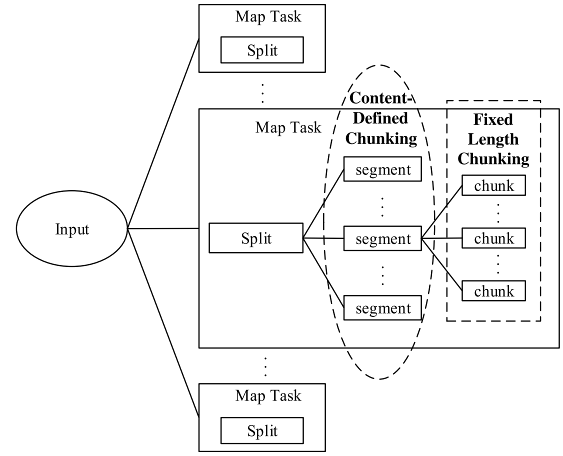


Fig. 5. In order to find delta data, HadInc divides the dirty splits with finer grain.

md5 code of this split. Then it puts the md5 code and result path into *ResultMemory* with a key-value form. Finally, the context of *MapReduce* job collects the task results and shuffles them, then the reduce tasks output the job result into HDFS.

In subsequent round, (c) every map task downloads *ResultMemory* and input data with the split information. (d) In view of incremental processing, a lot of splits are not changed, *HadInc* therefore can find their results in *ResultMemory* easily and reuse it directly. (e) Context collects those results and transmits them to reduce tasks. (f) If the split is changed, *HadInc* finds out the delta data between the old split and new split. Then the results of delta data and previous results will be combined and uploaded to HDFS.

3.3.1. *FinerGrainedDivider*

FinerGrainedDivider is a dividing algorithm for splits based on *CDC* with good stability. When the split is changed, there are no results to reuse. *FinerGrainedDivider* will divide this split into segments, and further divide each segment into chunks.

Fixed length chunking is used by the existing finer grained incremental processing system. It is reliable at processing appending datasets, however when some modifications appear in the middle of the datasets, the dividing result are significantly affected

In Fig. 4(a), we assume a fixed length chunking schema, in which each split contains 3 numbers. We then divide 15 numbers into 5 splits. Now, if we insert a 0 into 4 and 5, and divide the datasets again, the dividing result will be changed, which means only few of them can be reused.

In *CDC* based data chunking, we assume that the end number of each split is multiple of 3. The first chunking result is showed in Fig. 4(b). We then insert new data 0 into 4 and 5, and the dividing result is second chunking. Obviously, there is only one dirty split, and any other splits can reuse its previous results. In a word, although *CDC* will complicate the dividing algorithm, however, it can make the dividing very stable to ensure that much more results can be reused.

To make a compromise between stability and efficiency of the dividing, in consideration of that the finer grained dividing will cost more memory and time, and the cost of re-computing a segment is not big because of its small size, *FinerGrainedDivider* uses *CDC* only for dividing splits into segments, and utilize fixed length chunking for segments, as shown in Fig. 5.

This algorithm can avoid finer-grained dividing for all splits and take advantage of stability of *CDC*. It means that with this algorithm, we can get both stability and efficiency with low overhead.

3.3.2. *DeltaFinder*

In subsequent round, if the input split of a map task is new, *HadInc* cannot find its finger print in *ResultMemory*. *HadInc* then calls *DeltaFinder* to find out the delta data between old and new splits, and merges the delta data and old task into the new task result.

First, we obtain the index x of current split according to *RangeInfo* file. Then, download the old split through the xth record in *LastRangeInfo* file. If the xth record is null, it indicates that this split is appending and we should compute it. Finally, *HadInc* will start *DeltaFinder* to find the difference between those two splits.

DeltaFinder takes advantage of D-SD algorithm described in *HadUP*. It finds out the different data between old and new splits through analyzing. To decrease the invalid computation, the reference segments of each base segment [32] should be found out. The smaller difference between these two segments, the stronger correlation they have [33].

After the previous introduction, we probably understand the principle of *DeltaFinder* In the following, we describe different steps of *DeltaFinder* in detail.

(1) Finding reference segments for every new segment: If a new segment shares the most chunks with an old segment compared to others, then this old segment is the most relevant segment of the new segment. However, there are more than thousands chunks in a segment. If we check those chunks one by one, obviously, it decreases the efficiency of the system seriously. Therefore, we sample some of chunks to find the reference segments. The detailed procedure is showed in Table 2, and the time complexity is shown below:

$$T = O(SegNum1 * (ChkNum1 + rsnum_{max} * SegNum2 * ChkNum2)) \quad (1)$$

SegNum1 and *SegNum2* are very close, so do *ChkNum1* and *ChkNum2*, and $rsnum_{max}$ is a constant, so the complexity can be simplified as below:

$$T = O(SegNum^2 * ChkNum) \quad (2)$$

(2) Finding reference segments for every old segment: Based on symmetry between the basic and reference segments, we can find reference segments of old segments easily through reversing the result that we get in step (1).

(3) Calculating delta datasets: We optimize the D-SD algorithm in this step to accelerate the procedure of finding the delta data, as showed in Table 3, and time complexity is O(*SegNum* **ChkNum*).

Table 2
Reference segments searching algorithm.

```
Input:
    Old split S_old, New split S_new, Minimum share rate sr_min, Maxmum
reference segment number rsnum_max
Output:
    Reference segment map M_rs&lt;segment, refSegSet&gt;
for all newSeg ∈ S_new do
    for all chunk ∈ newSeg do
        if F_match(chunk, mode)
            sampleMap ← sampleMap ∪ <chunk.fp,+1>
        end if
    end for
    tmpNS ← newSeg
    while shareRate < sr_min && refSegNum < rsnum_max
        for all oldSeg ∈ S_old
            tmpSamMap ← sampleMap
            for all chunk ∈ oldSeg
                if chunk∈ tmpSamMap then
                    F_increase(shareRate) ▷ newSeg and oldSeg share
 this chunk
                    tmpSamMap ←tmpSamMap – chunk
                end if
            end for
        end for
        index = arg max(tmpNS ∩ seg_i) i=1,2,3…n ▷ select the
segment with the highest share rate to be a reference segment
        refSegSet ← refSegSet ∪ seg_index
        sampleMap ← sampleMap - seg_index
        tmpNS ← tmpNS – (tmpNS ∩ seg_index)
        F_increase(refSegNum)
    end while
    M_rs ← M_rs ∪ <newSeg, refSegSet>
end for
```

Table 3
Delta data searching algorithm.

```
Input:
    Old split S_old, New split S_new, Reference segment map M_rs<newSeg,
refSegSet>, Reversed reference map M_rvs, Old split result R_old
Output:
    New split result R_new
1:  Init delta data map:M_delta<chunk,number>
2:  for all <base,refset>∈M_rs do ▷ base∈newData, ref∈oldData
3:      for all chk ∈base do
4:          chkAmt ← F_Count(chk,M_delta) + F_Count (chk,baseSeg) –F_Count
(chk, refset)
5:          F_Update(M_delta,<chk, chkAmt>)
6:      end for
7:  end for
8:  for all <base,refset>∈M_rvs do ▷ base∈oldData, ref∈newData
9:      for all chk ∈base do
10:         X segments contain this chunk
11:         if X = 0 then
12:             chkAmt ← F_Count(chk,refset) –F_Count(chk,base)
13:         else if X > 1 then
14:             chkAmt ← F_Count(chk,refset) + (X - 1) * F_Count(chk,base)
15:         end if
16:         F_Update (M_delta, <chk, chkAmt>)
17:     end for
18: end for
19: R_new← F_Merge(R_old, M_delta)
```

Different from using fixed length chunking in D-SD, HadInc takes advantages of CDC to divide the changed splits into segments, therefore, we can obtain a quite stable dividing results even when the split has been changed a lot. In addition, D-SD uses two data structures to save the modified records and deleted records respectively. Instead, we use one data structure to save these two kinds of data, and we merge these data when the structure collects them immediately.

Line 2–7 in Table 3 for the delta data searching algorithm describe that firstly we make segments in new split as basic segments and make segments in old split as reference segments, and figure out the first part of delta data saved in structure *deltaDataMap*. Hence different basic segments probably share the same reference segments, some data will be calculated many times. Line 8–18 show how we correct this mistake. Line 19 describes that *HadInc* merges the old and delta data results to generate the current datasets results.

4. Evaluation

In our experiments, Hadoop 2.3.0 platform is used to test *HadInc* and other methods. The hardware details are as follows: (1) CPU mode:64-bit; (2) CPU(s): 4; (3) Thread per core: 1; (4) CPU MHz:2128; (5) Memory: 8 GB.

In this section, we will evaluate *HadInc* through two types of experiments.

(1) In Section 4.1, we design three cases to simulate that the dataset is changed in different degrees.

(a) Few splits are changed slightly for simulating appending case.

(b) Lots of splits are changed slightly for simulating complex appending case.

(c) Few splits are changed seriously and a lot of splits are changed slightly for simulating complex modifying case.

(2) In Section 4.2, in order to compare the efficiency between *HadInc* and other classical incremental processing algorithm we use two actual datasets, which are downloaded from Wikipedia by web crawler at different times The datasets we download contains two parts, one is hot words on Internet which will be changed probably, and another is common vocabulary which usually does not change. We therefore can obtain a general dataset to test our system.

Before we start the experiments, let us analyze each step of incremental processing.

An incremental processing job can be separated into five steps, and they are *submit*, *map*, *shuffle*, *reduce* and *output*. In *submit* and *output* steps, we will do some works to support incremental processing job which could increase the time cost. To avoid destroying transparency and compatibility, *shuffle* and *reduce* are hardly to utilized to increase the performance of incremental processing. If the time we saved in map tasks is much more than the time consumption in other steps, the method is useful.

We define the total increased efficiency as E_t:

$$E_t = 1 - \frac{T_{mr} + T_1 - T_2 + T_3}{T_{mr}} \quad (3)$$

$$E_t = \frac{T_2 - T_1 - T_3}{T_{mr}} \quad (4)$$

where:

T_{mr}—The time that a common *MapReduce* job costs.

T_1—The time that *Preprocessing* step costs.

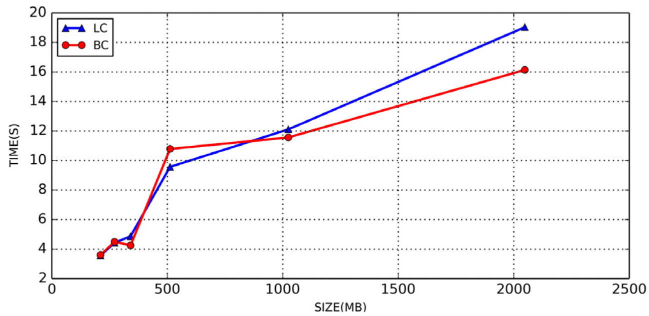


Fig. 6. With algorithm HadInc, T_1 trend is linear with dataset size increasing both in *LC* and *BC*.

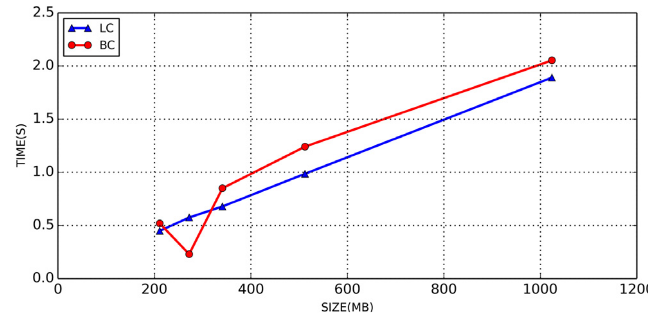


Fig. 7. With algorithm HadInc, T_3 trend is linear with datasets size increasing both in *LC* and *BC*.

T_2—The time which is saved in map by specific methods.

T_3—The time cost of *Postprocessing* step, in which system will deletes useless results and do some work for the next calculation.

T_1 mainly consists of the time that we search datasets for markers matching. It is defined as follow:

$$T_1 = N_s * N_m * P * T_{md5} \tag{5}$$

$$N_m = S_s * (1 - \alpha) * 2 \tag{6}$$

Where:

N_s—The split number of all input datasets

S_s—The standard split size we set before.

N_m—The range that the window slides.

P—The expectation ratio of the move range and the distance that window slides.

T_{md5}—It is a constant time cost to calculate the md5 code of a window.

As it can be seen, there is a linear positive correlation between T_1 and the size of input datasets

Fig. 6 describes how T_1 changes when the size of input datasets increases step by step. We change few splits of old datasets to generate a datasets named *LC*, and change lots of splits to generate datasets *BC*. When a datasets increases, there are more modifying actions to change the records in it. We will discuss this in detail in the next subsection.

Besides the first few points which are not clear to predict the trend, it is easy to find that the slope of T_1 and datasets size is quite low, which means T_1 increases very slowly, even if the data size increases almost by seven times.

In *map* step, there are two kinds of results reusing in *HadInc*. One is task level reusing which is called coarse grained reusing. Another is chunk level reusing which is called fine grained reusing. T_2 is defined as the time cost of the *MapReduce* step in increasing processing job. In *HadInc*, T_2 not only contains task level reusing, but also includes chunk level reusing. So we define T_2 as follow:

$$T_2 = P * T_{2t} + (1 - P) * T_{2c} \tag{7}$$

$$T_{2t} = T_{chk} + T_{down} \tag{8}$$

$$T_{2c} = T_{chk} + T_{cdc} + N_{ns} * (T_{ref} + T_{del}) \tag{9}$$

where:

P—The expectation task number of task level reusing. (1 -P) is the expectation task number of chunk level reusing.

T_{2t}—The time cost of task level reusing.

T_{2c}—The time cost of chunk level reusing.

T_{chk}—At the beginning of map tasks, *HadInc* will check whether *ResultMemory* contains the result of current input split.

T_{down}—If *ResultMemory* contains the result of current split, *HadInc* will download the result from HDFS.

Table 4
Datasets information.

Size(MB)	Total num	LC num	BC num
211	13	2	6
272	17	2	9
374	22	3	11
512	34	3	17
1024	65	7	32
2048	131	14	65

T_{cdc}—If the split is new, *HadInc* needs to divide the split into segments and chunks based on CDC.

N_{ns}—New splits number.

T_{ref}—The time cost of searching reference segments.

T_{del}—The delta data between new and old split.

T_{ref} is irrelevant to the content of input splits. Usually it is constant as long as the standard split size is not changed. The more records modified, the more chunk should be re-computed. However, the time consumption of re-computing a chunk is constant. When modifying actions happen randomly, we can consider that T_{del} is linear positive correlative to the number of modifying actions.

As we discussed before, in *Postprocessing*, *HadInc* will delete useless materials, for example, the records in *ResultMemory* that is not accessed at this time will never be accessed. It is easy to understand that T_3 is linearly correlated to modifying actions. Because the more splits changed, the results become more invalid. Fig. 7 illustrates this trend

4.1. *Targeted Testing*

To simulate the specific cases, we modified the source datasets in different degrees. *LC* and *BC* respectively represent the new datasets which come from the source datasets with 10% and 50% modified splits.

In case1 and case2, we respectively use *LC* and *BC* as input for subsequent calculation. In detail, Table 4 describes the basic information of source datasets, *LC* and *BC Size* represents the size of datasets. *Total num* represents the split number of datasets. *LC num* and *BC num* represent the changed split number in *LC* and *BC* respectively. For example, in the first experiment, the size of three datasets is 211 MB, and they all have 13 splits. There are 2 changed splits in *LC* and 6 changed splits in *BC*. We use *LC* and *BC* to test the methods in case1 and case2 respectively.

Table 5 shows the time consumption of four different algorithms to finish the incremental processing job after few splits changed slightly. *PlainMR* represents the common MapReduce algorithm without results reusing schema. *TaskReuse* denotes a kind of coarse-grained results reusing algorithm, which means if the task input is changed, the task should re-compute the split totally. *ChkReuse* gives a kind of fine-grained results reusing algorithm

Table 5
Case 1—Few splits are changed slightly.

Algorithm	211	272	341	512	1024	2048
PlainMR	91.2	111.8	132.3	181.0	312.7	583.7
TaskReuse	83.6	103.4	119.9	166.9	283.5	520.0
ChkReuse	83.3	102.3	120.6	168.7	278.4	**519.9**
HadInc	**83.2**	**102.6**	**119.6**	**162.8**	**277.6**	520.0

Table 6
Case 2—Lots of splits are changed slightly.

Algorithm	211	272	341	512	1024	2048
PlainMR	92.3	113.0	132.3	181.7	311.7	584.3
TaskReuse	84.3	104.5	122.6	169.9	286.5	538.0
ChkReuse	84.2	103.4	**121.4**	**167.7**	283.5	530.8
HadInc	**84.2**	**103.0**	122.6	166.9	**280.5**	**528.1**

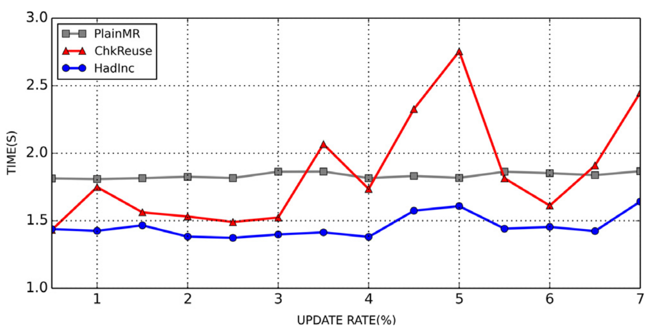


Fig. 8. Case 3—Time cost of a task finished by different methods.

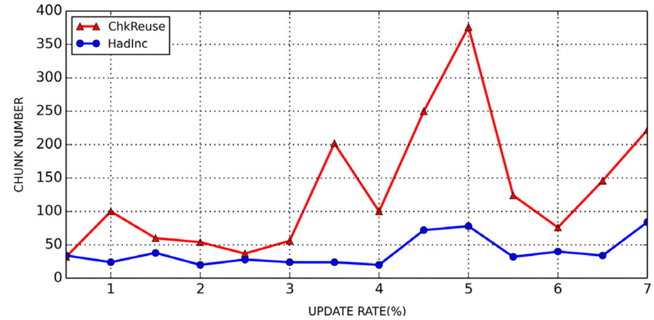


Fig. 9. Case 3—The changed chunks number grows with update ratio growth.

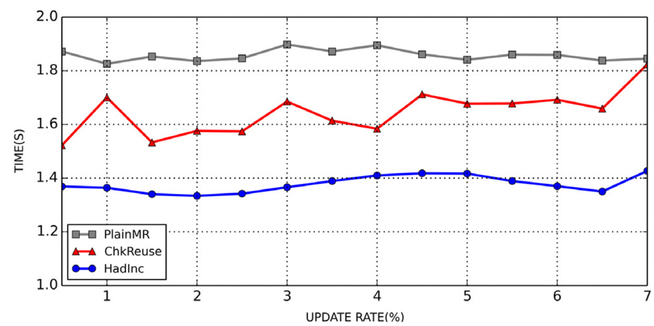


Fig. 10. Case 3—Average time cost of a task finished by different methods.

like *HadUP*. If the task input is changed, *ChkReuse* will find the delta dataset between new and old inputs based on fixed length chunking.

In case1, the time costs of *TaskReuse*, *ChkReuse* and HadInc are very close, and all of them are quicker than *PlainMR*. The reason is that only few splits are changed, most of splits results can be reused. Table 5 shows that *HadInc* is as good as other methods.

Table 6 shows the consumption efficiency when lots of splits are changed. Generally, case2 costs more time than case1, because in this condition, the algorithms need to process more changed splits. Two kinds of chunk level reusing methods perform better than *TaskReuse*. This is because of the reason that 50% splits are changed, the *TaskReuse* method needs to re-compute much more splits than in case1.

Overall, *HadInc* leads other methods. Hence the time cost to set up and run a *MapReduce* job is much more than the time that a task takes, thus, the advantage is not big enough. To make sure that our *HadInc* is efficient, we design case 3, in which we focus on a task to compare the stability and efficiency between *HadInc* and other methods. At first, we modify the datasets randomly with an update ratio increasing from 0.5% to 7%. Then we mark down the time consumption of each task, and figure out the average of every method in different update ratio, which is described in Fig. 8.

As the input split of each task is changed, the time cost of *TaskReuse* is the same as *PlainMR* which needs to re-compute the split *TaskReuse* method is therefore discarded at this time. Since *ChkReuse* is based on fixed length chunking, when datasets updates quickly, the previous dividing result is destroyed heavily, the time cost of this method fluctuates seriously. When update ratio is less than 4%, *HadInc* keeps the time cost in a low level steadily. As the more update ratio increases, the less chunks can be reused. When the ratio is more than 4%, the stability drastically decreases

Fig. 9 shows the trend of changed chunks number, which goes up when more and more modifying actions appear. The changed chunks number with method *ChkReuse* is always much bigger than the number of HadInc. Different from *ChkReuse*, using *HadInc* can keep changed chunks number at a very low level steadily. In addition, the test data is randomly modified from source data. In Figs. 8 and 9, because the modifications are so dispersive that the number of changed chunks grows quickly, when update ratio grows up to 5%, it makes task time consumption jumps to a high level.

To avoid the impact of random modifications on our experiments, we test case 3 fifty times, and the average result is showed in Fig. 10. All the curves grow up smoothly. When the ratio grows up to 7%, *ChkReuse* even performs worse than *PlainMR*, however, HadInc still remains at a low time consumption level.

In Fig. 11 we compare the average changed chunks number among *PlainMR*, *ChkReuse*, *HadInc*. With the ratio growing, there are much more changed chunks with method *ChkReuse* than *HadInc*. The reason is that data modifications will make the dividing results to collapse easily when fixed length chunking method is used.

From the experiments we can see that *HadInc* performs as good as other methods in case1 and case2, and performs much better than others in case 3 both in computation efficiency and stability.

4.2. Practical testing

In order to test the performance of *HadInc* on practical datasets, we download two datasets from Wikipedia on January 10 and 19 respectively. The key words contain hot words used in 2016 which probably changed during a short time, and common things, such as famous people, cities, movies and events, etc.

In detail, we divide the keywords into 10 categories, for example, people, place, news, sports, movies, etc., and the total data size is up to 1.11 GB. In case1 and case2, when the size jumps to 1024 MB, compared to *PlainMR* the computation time for all other methods is reduced by approximately thirty seconds. In case 4, as

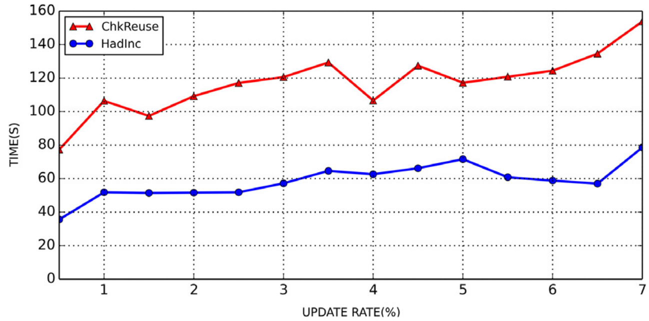


Fig. 11. Case 3—Average changed chunks number.

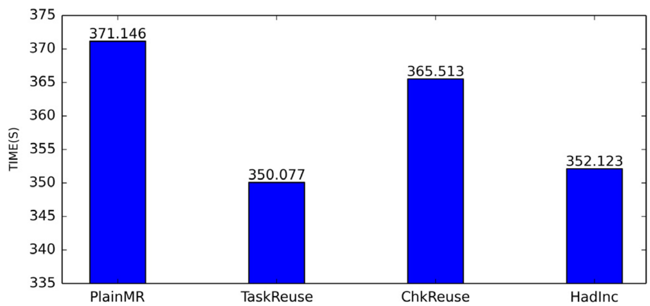


Fig. 12. Case 4—Practical testing with Wikipedia datasets.

showed in Fig. 12, it is only twenty seconds, and *ChkReuse* performs much worse than *TaskReuse* and *HadInc*

We also find that the update ratio of hot words is more than 20% in the datasets, including some invalid changes like the editors modify description of a word from uppercase to lowercase which causes re-computing of this chunk. These hot words are also very dispersive in the datasets. In addition, case 4 is also a big challenge for incremental processing, however, *HadInc* performs very good in this test.

5. Conclusions

In this paper, we propose an incremental processing model called *HadInc* which is based on CDC and finer-grained results reusing. *HadInc* uses sliding detection matching algorithm to divide datasets into splits for a stable result and high efficiency. Different from other finer-grained results reusing methods, *HadInc* divides splits into finer data structure based on CDC only when the split is changed. We evaluate *HadInc* on four different cases. Our experimental results show that *HadInc* achieves higher efficiency compared to other methods even when the update ratio is 7%. Even when the update ratio of a part of datasets increases to 20%, *HadInc* is steady and achieves good efficiency. To conclude, *HadInc* is a stable, finer-grained and efficient method to process incremental big data.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This work is partially supported by the China Postdoctoral Science Foundation (Grant No. 2016M592763, No. 2013M542370), the Fundamental Research Funds for the Central Universities (Grant No. JB161006, JB161001), the National Natural Science Foundation of China (Grant No. 061401324, 61305109), Shaanxi Provincial Education Department (Program No. 2013JK1139), the Specialized Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20136118120010). And this project is also supported by NSFC Grant (Program No. 11226173 & 11301414 & 61272283). This work is also supported by Shaanxi Province Hundred Talents Program, Natural Science Foundation of China under grants 61272509, Research Fund for the Doctoral Program of Higher Education 20136118110002, Natural Science Foundation of Shaanxi Province 2016JM6058.

References

[1] R.I. Jony, R.I. Rony, M. Rahman, et al., Big data characteristics, value chain and challenges, in: International Conference on Advanced Information and Communication Technology, 2016.
[2] The past and present of big data. http://www.leiphone.com/news/201410/NgTsZw3yDjEbk9on.html.
[3] S. Kim, H. Han, H. Jung, et al., Harnessing input redundancy in a MapReduce framework, in: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, New York, NY, USA, 2010, pp. 362–366.
[4] B.J. Jansen, A. Spink, T. Saracevic, Real life, real users, and real needs: A study and analysis of user queries on the web, Inf. Process. Manage. 36 (2) (2000) 207–227.
[5] A. Muthitacharoen, B. Chen, D. Mazières, A low-bandwidth network file system, in: SOSP, 2001.
[6] D. Lee, J.S. Kim, S. Maeng, Large-scale incremental processing with MapReduce, Future Gener. Comput. Syst. 36 (7) (2014) 66–79.
[7] J. Dean, S. Ghemawat, MapReduce:simplified data processing on large clusters, in: Proc. USENIX Symp. Operating Systems Design and Implementation, OSDI, Dec. 2004.
[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed dataparallel programs from sequential building blocks, in: Proc. ACM European Conf. Computer Systems, EuroSys, Mar. 2007.
[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig Latin: a not-soforeign language for data processing, in: Proc. ACM Int'l Conf. Management of Data, SIGMOD, Jun. 2008.
[10] W. Wei, X.L. Yang, B. Zhou, et al., Combined energy minimization for image reconstruction from few views, Math. Probl. Eng. 16 (7) (2012) 2213–2223.
[11] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P.K. Gunda, J. Currey, DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language, in: Proc. USENIX Symp. Operating Systems Design and Implementation, OSDI, Dec. 2008.
[12] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proc. ACM Int'l Conf. Management of Data, SIGMOD, Jun. 2010.
[13] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, R. Murthy, Hive—a petabyte scale data warehouse using Hadoop, in: Proc. IEEE Int'l Conf. Data Engineering, ICDE, Mar. 2010.
[14] D.G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, S. Hand, CIEL: a universal execution engine for distributed data-flow computing, in: Proc. USENIXSymp. Networked Systems Designand Implementation, NSDI, Mar. 2011.
[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: Proc. USENIX Symp. Networked Systems Design and Implementation, NSDI, Apr. 2012.
[16] W. Wei, X.L. Yang, P.Y. Shen, et al., Holes detection in anisotropic sensornets: Topological methods, Int. J. Distrib. Sens. Netw. 21 (9) (2012) 3216–3229.
[17] Y. Low, D. Bickson, J. Gonzalez, et al., Distributed GraphLab: a framework for machine learning and data mining in the cloud, Proc. VLDB Endowment 5 (8) (2012) 716–727.
[18] K.H. Lee, Y.J. Lee, H. Choi, et al., Parallel data Processing with MapReduce: a survey, ACM SIGMOD Record 40 (4) (2012) 1–20.
[19] D. Peng, F. Dabek, Large-scale incremental processing using distributed transactions and notifications, in: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10), USENIX Association, Berkeley, CA, USA, 2010, pp. 251–264.
[20] D. Logothetis, C. Olston, B. Reed, et al., Stateful bulk processing for incremental analytics, in: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, New York, NY, USA, 2010, pp. 51–62.
[21] F. Chang, J. Dean, S. Ghemawat, et al., Bigtable: a distributed storage system for structured data, in: Symposium on Operating Systems Design and Implementation. USENIX Association, 2006, pp. 205–218.
[22] S. Ghemawat, H. Gobioff, S.T. Leung, The Google file system, Acm Sigops Oper. Syst. Rev. 37 (5) (2003) 29–43.
[23] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters. in: Conference on Symposium on Opearting Systems Design & Implementation, DBLP, 2004, pp. 137–150.

[24] A. Bhatotia P'Wieder, R. Rodrigues, et al., Incoop: MapReduce for incremental computations, in: Proceedings Ofthe 2nd ACM Symposium on Cloud Computing, ACM, New York. NY-USA, 2011, pp. 1–14.
[25] C. Yan, X. Yang, Z. Yu, et al., Incmr: Incremental data Processing based on mapreduce, in: Proceedings of the 2012 IEEE 5th International Conference on Cloud Computing(CLOUD), IEEE, Honolulu, HI, 2012, pp. 534–541.
[26] M. Hayes, S. Shah, Hourglass: A library for incremental processing on hadoop, in: Proceedings of the 2013 IEEE International Conference on Big Data, IEEE, Silicon Valley, CA, USA, 2013, pp. 742–752.
[27] D. Tiwari, Y. Solihin, Mapreuse: reusing computation in an in-memory mapreduce system, in: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS'14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 61–71.
[28] Y. Zhang, S. Chen, Q. Wang, et al., i2MapReduce: Incremental MapReduce For mining evolving big data, IEEE Trans. Knowl. Data Eng. 27 (7) (2015) 1906–1919.
[29] Apache Hadoop. http://hadoop.apache.org/.
[30] LinkedIn. http://www.linkedin.com/.
[31] M. Lillibridge, K. Eshghi, D. Bhagwat, et al., Sparse Indexing: Large scale, inline deduplication using sampling and locality, in: Proccedings of the 7th Conference on File and Storage Technologies, FAST'09, USENIX Association, Berkeley, CA, USA, 2009, pp. 111–123.
[32] D. Fetterly, M. Haridasan, M. Isard, S. Sundararaman, TidyFS: a simple and small distributed file system, in: Proc. USENIX Annual Tech. Conf. USENIX, Jun. 2011.
[33] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, P. Camble, Sparse indexing: largescale, in line deduplication using sampling and locality, in: Proc. USENIX Conf. File and Storage Technologies, FAST, Feb. 2009.

Liang Zhang received the Ph.D degree in instrument science and technology from Zhejiang University, in September 2009. In September 2009, he joined the School of Software, Xidian University, where he is currently an associate Professor and the Director of the Embedded Technology and Vision Processing Research Center. He has published more than 40 academic papers in peer-reviewed international journals and conferences. His research interests lie in the areas of big data processing, multicore embedded systems, computer vision, deep learning, simultaneous localization and mapping (SLAM), human robot interaction, image processing.

Yuanyuan Feng is a master student in Xidian University, Xi'an, China, his research interests contain big data processing architecture, distributed computing, image processing.

Peiyi Shen is a Professor in National School of Software at Xidian University. He is also a member of IEEE. He was a research officer and a Ph.D. student in the MTRC, Computer Science at the Univ. of Bath, and a research fellow in CVSSP at the Univ. of Surrey, studying and working under Prof. Philip Willis and Prof. Adrian Hilton. He was also with Agilent Technologies in the USA, UK, Malaysia and Singapore from 2000 to 2003. He was also a postdoctoral research fellow in the School of Computing at the National Univ. of Singapore in 2000 after he took his first Ph.D. in Xidian University in 1999. His research interests are in big data processing, Computer Vision, Volume Visualization and ITS applications.

Guangming Zhu received the Ph.D. degree in instrument science and technology from Zhejiang University, China, in March 2015. He is currently a Post-Doctoral researcher at School of Software, Xidian University. His major research fields are information fusion, human action/gesture recognition, scene recognition, and deep learning.

Wei Wei received his Ph.D. and M.S. degrees from Xian Jiaotong University in 2011 and 2005, respectively. Currently he is an assistant Professor at Xian University of Technology. His research interests include Wireless Networks and Wireless Sensor Net-works Application, Image Processing, Mobile Computing, Distributed Computing, and Pervasive Computing.

Juan Song received her B.S. degree in school of communication engineering from Hohai University, Nanjing, China in 2006, and received her Ph.D. degree in communication and information system from Xidian University, Xi'an, China in 2012. She is now an associated professor in national school of software at Xidian University. Her research interests include image processing and pattern recognition. She has published more than 20 academic papers in peer-reviewed international journals and conferences.

Dr. Syed Afaq Ali Shah is a Research Associate and Lecturer in the School of Computer Science and Software Engineering (CSSE), the University of Western Australia (UWA), Perth. He obtained his Ph.D. in 3D computer vision and machine learning from UWA. Afaq has developed machine learning systems and various feature extraction/matching algorithms for 3D object recognition and reconstruction. He has published research papers in high impact factor journals and reputable conferences.

Mohammed Bennamoun is currently a W/Professor at the School of Computer Science and Software Engineering at The University of Western Australia. He lectured in robotics at Queen's, and then joined QUT in 1993 as an associate lecturer. He then became a lecturer in 1996 and a senior lecturer in 1998 at QUT. In January 2003, he joined The University of Western Australia as an associate professor. He was also the director of a research center from 1998–2002. He is the coauthor of the book Object Recognition: Fundamentals and Case Studies (Springer-Verlag, 2001). He has published close to 100 journals and 250 conference publications. His areas of interest include control theory, robotics, obstacle avoidance, object recognition, artificial neural networks, signal/image processing, and computer vision.