Future Generation Computer Systems 36 (2014) 66–79

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Large-scale incremental processing with MapReduce

Daewoo Lee [a,*], Jin-Soo Kim [b], Seungryoul Maeng [a]

[a] Computer Science Department, Korea Advanced Institute of Science and Technology, 335 Gwahak-ro, Yuseong-gu, Daejeon 305-701, Republic of Korea

[b] College of Information and Communication Engineering, Sungkyunkwan University, 300 Cheoncheon-dong, Jangan-gu, Suwon 440-746, Republic of Korea

HIGHLIGHTS

- Revealing the ineffectiveness of task-level memoization for incremental processing.
- An algorithm to detect changes in large datasets efficiently in Hadoop clusters.
- An efficient implementation to compute the updated result in Hadoop clusters.

ARTICLE INFO

Article history:
Received 20 December 2012
Received in revised form
14 August 2013
Accepted 5 September 2013
Available online 19 September 2013

Keywords:
Big data processing
Incremental processing
MapReduce
Hadoop
Data deduplication

ABSTRACT

An important property of today's big data processing is that the same computation is often repeated on datasets evolving over time, such as web and social network data. While repeating full computation of the entire datasets is feasible with distributed computing frameworks such as Hadoop, it is obviously inefficient and wastes resources. In this paper, we present HadUP (Hadoop with Update Processing), a modified Hadoop architecture tailored to large-scale incremental processing with conventional MapReduce algorithms. Several approaches have been proposed to achieve a similar goal using task-level memoization. However, task-level memoization detects the change of datasets at a coarse-grained level, which often makes such approaches ineffective. Instead, HadUP detects and computes the change of datasets at a fine-grained level using a *deduplication-based snapshot differential algorithm (D-SD)* and *update propagation*. As a result, it provides high performance, especially in an environment where task-level memoization has no benefit. HadUP requires only a small amount of extra programming cost because it can reuse the code for the map and reduce functions of Hadoop. Therefore, the development of HadUP applications is quite easy.

1. Introduction

As many industries and organizations handle increasing amounts of data, big data processing is being considered as a promising technology. Extracting meaningful and valuable information from huge datasets is important for providing attractive new services as well as improving the quality of existing services, such as web-data analysis, log processing, and click analysis. Analyzing large amounts of data collected from various sources poses great challenges to the fields of science, especially those involving massive-scale simulations and sensor networks.

Currently, distributed computing frameworks [1–9] are being widely used for big data processing. These systems allow the user to write applications using a set of high-level operations, and automatically handle the complex aspects of distributed computing, such as scheduling and fault tolerance. Among them, Hadoop [1], an open-source MapReduce implementation designed for large clusters, has emerged as a de facto standard for big data processing in both industry and academia.

An important property of today's big data processing is that the same computation is often repeated on datasets evolving over time, such as web and social network data. To keep a web index up-to-date, for example, a search engine repeatedly crawls a part of the web, merges it with the previously crawled pages, and performs several computations over all the crawled pages. Since the updated data between runs is typically much smaller than the entire datasets [10–13], we can improve the overall efficiency dramatically by performing the computation incrementally.

However, most of the current distributed computing frameworks lack support for incremental processing. While their highly scalable performance makes it feasible to repeat full computation of the entire datasets [12,13], doing so is obviously inefficient and wastes resources. Recently, a few systems have been designed for efficient incremental processing [12,13], but they require applications to be totally rewritten with new programming languages and

* Corresponding author. Tel.: +82 42 350 7719.
E-mail addresses: dwlee@calab.kaist.ac.kr (D. Lee), jinsookim@skku.edu (J.-S. Kim), maeng@kaist.ac.kr (S. Maeng).

0167-739X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.09.010

dynamic algorithms. Moreover, implementing dynamic algorithms is known to be very difficult, even for problems that are simple with static input data [10].

Our goal is to enable large-scale incremental processing with conventional MapReduce algorithms. Some approaches have been proposed to achieve a similar goal using *task-level memoization* [10,11,14,15]. They reuse the previous results of a task (or a set of tasks) when the same computation on the same task input is needed again. When most of the task inputs are unchanged, therefore, task-level memoization produces a large benefit. However, when many tasks repeat the same computation just due to a few changes on their input data, the computation delay and overhead can become considerably important. This can be illustrated by the following two observations: First, the unit of change is much smaller than a task input. Since a task is typically assigned to tens or hundreds of megabytes of data containing millions of records, many task inputs can be changed by a small change in the dataset. Second, a many-to-many communication pattern called shuffle is commonly used in big data processing; in the MapReduce context, it always appears in the data transfer between the map and reduce stages. Even with only a few changes in the task outputs of the preceding stage, shuffle can change almost all the task inputs of the following stage. As a result, task-level memoization can suffer from unnecessary computation and data transfer even when the application handles an incrementally augmented dataset such as log data.

In this paper, we present HadUP (Hadoop with Update Processing), a modified Hadoop architecture tailored to large-scale incremental processing. For efficient incremental processing, HadUP detects and computes the change of datasets at a fine-grained level. To reduce the granularity, we propose two techniques: a *deduplication-based snapshot differential algorithm (D-SD)* and *update propagation*.

When the application starts to run, HadUP detects the change of its input as the first step of incremental processing. Given the old and new snapshots of a dataset, we can figure out how the dataset changes by ignoring all duplicated data between them. Data deduplication, which is mainly used to eliminate duplicate copies of repeating data, is a viable technique to accomplish this efficiently. However, traditional data deduplication techniques are inefficient in handling big data because they are not designed for large-scale distributed systems. To improve the efficiency, we extend a technique called sparse indexing [16] to a large-scale Hadoop cluster. By exploiting the power of distributed computing, D-SD can perform well with big data.

The performance benefit of HadUP mainly comes from the next step, called update propagation, which is initiated with the result of D-SD. Update propagation is a style of computation that enables incremental processing with conventional algorithms for today's big data processing. Many applications for big data processing consist of data parallel operations, where an operation transforms one or more input datasets into one output dataset. For each operation, the same computation is concurrently applied to a single input record or a group of input records. The independence between these executions allows us to compute the records to be inserted into or deleted from the output dataset, if those records inserted into or deleted from the input datasets are explicitly given. In this way, HadUP computes the updated result without full recomputation. Furthermore, we implement HadUP to reuse the code for the map and reduce functions of Hadoop, targeting easy development of HadUP applications.

Evaluation shows that HadUP provides high performance, up to 2.5× speedup over Hadoop, while requiring only a small amount of extra programming cost. Since we assigned each task to a few hundred megabytes of data, we can suppose that task-level memoization performs similarly to (or worse than) Hadoop in our environment. HadUP incurs a penalty in the first run as it cannot take advantage of incremental processing, but it is just a one-time cost.

The rest of this paper is organized as follows. Section 2 reviews the MapReduce programming model and the Hadoop framework, and discusses the motivation of our work. Section 3 provides a brief description of HadUP's incremental processing. Sections 4 and 5 discuss how HadUP works in detail; D-SD in Section 4 and update propagation in Section 5. Section 6 evaluates HadUP and shows its benefit from incremental processing. We present related work in Section 7, and conclude in Section 8.

2. Background

2.1. MapReduce programming model

A MapReduce application consists of a workflow of jobs that process data structured in key/value pairs. Each job performs two user-specified functions: map and reduce. The *map* function is applied to each input record (i.e., key/value pair) and produces a list of intermediate records. The *reduce* function is applied to each group of intermediate records with the same key, and produces a list of output records. Inherently, the execution of the map and reduce functions can be parallelized. Within a job, however, the reduce execution is allowed to start after the map execution is finished. Optionally, the *combiner* function can be used to reduce the amount of intermediate data by aggregating a portion of the values associated with each intermediate key. It is effective in reducing network utilization between the map and reduce stages.

2.2. Hadoop

Hadoop is an open-source MapReduce implementation designed for large clusters. It consists of a single master node called the JobTracker and many slave nodes called the TaskTrackers. The JobTracker is responsible for parallelizing job execution across nodes and ensuring fault tolerance. The TaskTracker manages the execution of the tasks currently scheduled on its corresponding node and the inter-node communication between map and reduce tasks.

When a job is submitted, the JobTracker first schedules the job's map tasks. Each map task independently processes a *split*, a contiguous portion of the job's input data (64 MB by default). The task applies the map function to its split, and stores the intermediate data temporarily into a local disk. The JobTracker starts scheduling the job's reduce tasks after some of its map tasks are finished. To satisfy the semantics of the reduce function, all intermediate records with the same key should be processed by the same reduce task. This causes a many-to-many communication called *shuffle* between the map and reduce stages, which is likely to result in a large fraction of the job execution time. When storing intermediate records, a map task sorts them by key and partitions them using the user-specified partitioner function. Each partition is assigned to a reduce task. The task fetches its partition from all map outputs, and processes the data in increasing key order. Its output data is typically stored as a file in the Hadoop Distributed File System (HDFS).

2.3. Motivation

Task-level memoization [10,11,14,15] enables incremental processing by reusing the previous results of a task (or a set of tasks) when the same computation on the same task input is needed again. The underlying assumption of this approach is that, if the total amount of changed data is sufficiently small, most task inputs are unchanged. We argue that this situation is rare because of the following two reasons:

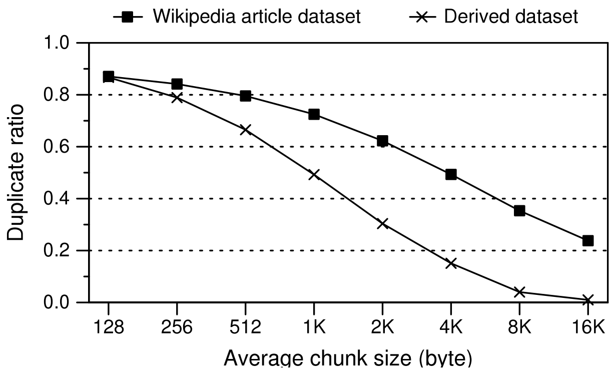


Fig. 1. Effectiveness of task-level memoization.

1. The unit of change (i.e., record) is much smaller than a task input;
2. Shuffle can change almost all of the task inputs in the following stage even though only a few task outputs are changed in the preceding stage.

To confirm our speculation, we estimated the effectiveness of task-level memoization using real datasets. We supposed that the input data is split using content-based chunking and each map task is assigned to a chunk.[1] By doing so with two snapshots of a dataset, we can measure how many task inputs are unchanged if the dataset is used as input. We measured the duplicate ratio, which is the ratio of the size of duplicate chunks to the size of the older snapshot, and considered this as a metric to evaluate the effectiveness of task-level memoization.

First of all, we performed the measurement with two snapshots of the Wikipedia English article dataset,[2] which were taken on Dec. 2011 and Jan. 2012, respectively; the newer snapshot is about 1% larger than the older one in size. Using our customized input parser for this dataset (Section 6.1), each record is about 100 byte in size. As shown in Fig. 1 (i.e., labeled "Wikipedia article dataset"), we find that although over 90% of data remain unchanged, most of the task inputs are changed even if a task input is only 16 KB in size. Since a task is typically assigned to tens or hundreds of megabytes of data containing millions of records, task-level memoization is obviously ineffective for such datasets.

Next, we investigated the impact of shuffle on task-level memoization. We first ran a MapReduce job on the Wikipedia English log dataset,[3] and performed the above measurement with the derived dataset (i.e., the job's result). We also used two snapshots taken on Dec. 2011 and Jan. 2012, respectively. Since the new data is always appended to this dataset, most of the map task inputs are unchanged. The result for the derived dataset, however, shows the same tendency as that of the first experiment, as shown in Fig. 1 (i.e., labeled "Derived dataset"). This means that, even with only few changes in the task outputs of the preceding stage, shuffle can change almost all the task inputs of the following stage. Many algorithms are known to be hard to implement with a single MapReduce job [17], so that multi-job applications are common in practice. Therefore, task-level memoization is inefficient for many real applications.

To conclude, we have found that task-level memoization is not the right direction toward efficient incremental processing. This motivates us to investigate a new approach that detects and computes the change of datasets at a fine-grained level.

3. HadUP overview

We develop a modified Hadoop architecture called HadUP (Hadoop with Update Processing). Fig. 2 illustrates the HadUP architecture. A map task runs similarly to the original map task of Hadoop, but performs a deduplication-based snapshot differential algorithm (D-SD) if it is assigned to the application input. A reduce task manages its partition of the intermediate dataset via caching and indexing for update propagation. During job execution, the JobTracker is responsible for task scheduling to guarantee the correctness of D-SD and update propagation (UP). After the application finishes, the user can get the updated result via the internal redirection mechanism in the HadUP client library.

3.1. Data representation

Throughout this paper, we consider that every dataset changes over time. If a dataset is produced by a computation, it is considered to change whenever the computation is applied to the changed input. Let D denote a dataset, then D_i denotes the *snapshot* of D processed (or produced) in the ith run. We view D_i as a *multiset* of records.

The *update* of a dataset represents the records required to transform its old snapshot into the new one. If the update of a dataset D can be used to transform D_{i-1} into D_i, we denote it by ΔD_i. ΔD_i is defined as a pair of multisets of records, $\langle \Delta D_i^+, \Delta D_i^- \rangle$, such that

$$(D_{i-1} \cup \Delta D_i^+) - \Delta D_i^- = D_i$$

where ΔD_i^+ and ΔD_i^- represent the records inserted into and removed from D_{i-1}, respectively. A record change is modeled as a deletion of the original record and an insertion of the new one. Note that ΔD_i is not unique for given snapshots D_{i-1} and D_i. It can be any pair of multisets of records as long as the above condition is satisfied.

For convenience, we assign a *type* to each record belonging to ΔD_i in order to represent its membership: $(+)$ to that in ΔD_i^+ and $(-)$ to that in ΔD_i^-. Simply, we represent a record in ΔD_i as 3-tuple, $\langle key, value, type \rangle$.

3.2. Incremental processing in HadUP

This section provides a brief description of update propagation, the core of HadUP's incremental processing. Consider a job that produces a list of web pages with their indegrees. For a link $source \rightarrow target$ in the input documents, the map function emits an inverted link (i.e., $\langle target, source \rangle$), and then the reduce function counts the number of distinct links to each target page (i.e., $\langle target, \#sources \rangle$). We denote the input, intermediate, and output dataset of this job as I, M, and R, respectively. Assume that I_1 contains two links to page a, namely $b \rightarrow a$ and $c \rightarrow a$, and one link to page b, namely $d \rightarrow b$. Assume that I_2 is identical to I_1 except that $d \rightarrow b$ is changed to $d \rightarrow a$. That is, $\Delta I_2 = \{\langle d, a, + \rangle, \langle d, b, - \rangle\}$.

Fig. 3 illustrates the computations by Hadoop and HadUP[4] where $\langle source, target \rangle$ in I_i denotes a link $source \rightarrow target$ in the input documents. In the second run, Hadoop recomputes I_2 although

[1] This input assignment scheme, which is actually used by Incoop [10], maximizes the effectiveness of task-level memoization. If the input data is split into fixed-sized chunks as done in Hadoop, insertion and deletion of a single record can change most of the split points.

[2] http://dumps.wikimedia.org/enwiki/.

[3] The Wikipedia English log dataset contains the log of actions performed on the Wikipedia English articles since Dec. 2004, and this job computes the statistics of the user actions for each month.

[4] The computation by HadUP in the first run is the same as that by Hadoop, thus we omit it. HadUP does not take advantage of incremental processing in the first run.

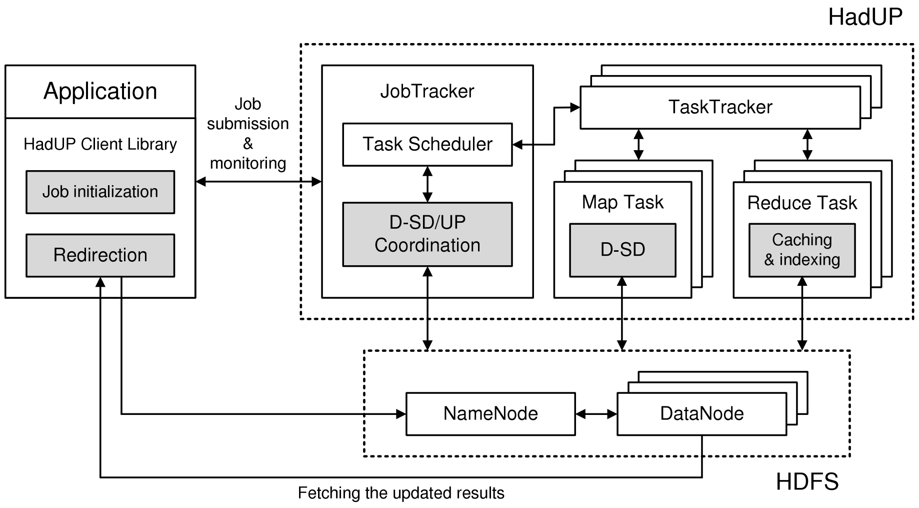


Fig. 2. HadUP architecture. The shaded area represents the modified part in each component.

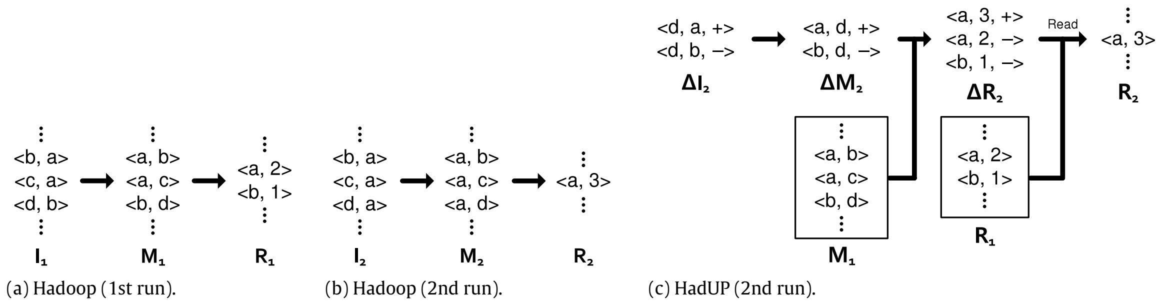


Fig. 3. A comparison between the computations by Hadoop and HadUP.

only one link has been changed. This recomputation is obviously inefficient and wastes resources. In contrast, HadUP performs update propagation, avoiding full recomputation. Given ΔI_2, HadUP derives ΔM_2 and ΔR_2 using the unmodified map and reduce function, and provides a view of R_2, the updated result, using R_1 and ΔR_2. Note that in Fig. 3(c), we assume that ΔI_2 is already computed; D-SD is used to compute ΔI_2 efficiently when I_1 and I_2 are given.

3.2.1. *Map*

Since the map function is applied to each input record, the insertion of $\langle d, a\rangle$ into I_1 does not affect any record in M_1. It causes the result of applying the map function to $\langle d, a\rangle$, namely $\langle a, d\rangle$, is inserted into M_1. In a similar way, we can find that the deletion of $\langle d, b\rangle$ from I_1 results in that of $\langle b, d\rangle$ from M_1. As a result, update propagation with the map function produces ΔM_2 $(= \{\langle a, d, +\rangle, \langle b, d, -\rangle\})$.

3.2.2. *Reduce*

Unlike the map function, the reduce function is applied to a group of intermediate records with the same key. To derive ΔR_2, therefore, we should translate ΔM_2 into the change of those record groups in M_1. The insertion of $\langle a, d\rangle$ into M_1 affects the group of records with key a in M_1, namely $\{\langle a, b\rangle, \langle a, c\rangle\}$. This group is changed into $\{\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle\}$, which is the group of records with key a in M_2. By applying the reduce function to these groups, we can find that $\langle a, 2\rangle$ in R_1 is changed into $\langle a, 3\rangle$ in R_2. This is represented as $\langle a, 3, +\rangle$ and $\langle a, 2, -\rangle$ in ΔR_2. In a similar way, the deletion of $\langle b, d\rangle$ results in that of $\langle b, 1\rangle$ from R_1; it causes no record insertion because there is no group with key b in M_2. As a result, update propagation with the reduce function produces ΔR_2 $(= \{\langle a, 3, +\rangle, \langle a, 2, -\rangle, \langle b, 1, -\rangle\})$.

3.3. *Writing HadUP applications*

It is easy to write HadUP applications because the code for map and reduce functions is compatible with Hadoop. For incremental processing, however, HadUP requires additional information that is unnecessary for Hadoop. For example, to perform the update propagation described in Section 3.2, HadUP requires the ID of the job executed in the first run. To describe such information, the user should add a new code to the driver program that initializes jobs and instructs Hadoop to execute them. In most applications, however, it is sufficient to add a few lines of code per job. Fig. 4 shows the driver program for running the example job of Section 3.2 on HadUP. It is almost the same as the driver program for Hadoop, except for some extra code added in lines 12–14.[5]

4. Deduplication-based snapshot differential algorithm (D-SD)

To begin update propagation in the ith run, HadUP needs ΔD_i where D is the application input. The goal of the snapshot differential algorithm is to compute ΔD_i with D_{i-1} and D_i. A naïve method is to run a job comparing the multiplicity of each record in D_{i-1} and D_i. Unfortunately, it requires gathering the same records into a task, which causes a large data transfer; almost all of D_{i-1} and D_i will be shuffled.

To reduce the amount of transferred data, data deduplication is a viable technique, which is mainly used to eliminate duplicate

[5] `setBaseJobID` specifies the ID of the job executed in the last run, and `setInputRecordTypeSelectorClass` specifies the type of input records that each map task processes.

```
Job j = new Job(new Configuration(), "Example");
j.setJarByClass(ExampleJob.class);
j.setMapperClass(ExampleMapper.class);
j.setReducerClass(ExampleReducer.class);
j.setOutputKeyClass(Text.class);
j.setOutputValueClass(IntWritable.class);
j.setInputFormatClass(TextInputFormat.class);
FileInputFormat.addInputPath(j, "/input");
FileOutputFormat.setOutputPath(j, "/output");

// New interfaces for HadUP
j.setBaseJobID("job_201305141650_0001");
j.setInputRecordTypeSelectorClass(
    FileNameSuffixBasedSelector.class);

j.waitForCompletion(true);
```

Fig. 4. An example of the driver program for HadUP.

copies of repeating data. It divides data into multiple chunks and ignores all chunks with the same content as the others processed earlier. To identify each chunk efficiently, the chunk fingerprint is used, which is typically a 20-byte SHA-1 hash of the chunk content. That is, two chunks with the same fingerprint are assumed to have the same content. Using data deduplication, we can ignore most of the unchanged data between D_{i-1} and D_i without record-by-record comparison. The remaining data can still be used as ΔD_i; ΔD_i^+ is from D_i, and ΔD_i^- is from D_{i-1}. Note that only the chunk information (i.e., the ID and the fingerprint of each chunk) of both snapshots is transferred, and there is a trade-off between effectiveness (i.e., the amount of unchanged data to be detected) and efficiency (i.e., the amount of transferred data).

Simply, we can design D-SD as a centralized algorithm. A centralized node such as the JobTracker can perform data deduplication after gathering all the chunk information of both snapshots. Although simple in design, this method is very inefficient in large-scale distributed systems such as Hadoop clusters. To detect a large amount of unchanged data, the chunk size should be sufficiently small (i.e., at most 1 KB), as discussed in Section 2.3. Then, the amount of chunk information is not small enough to be handled efficiently in a single node.

To overcome this problem, we extend a technique called *sparse indexing* [16] to large-scale Hadoop clusters. Sparse indexing is a technique that exploits the inherent locality within backup streams. According to [16], if the last time we encountered chunk *a*, it was surrounded by chunks *b*, *c*, and *d*, then the next time we encounter *a* (even in a different backup) it is likely that we will also encounter *b*, *c*, or *d* nearby. This tendency is called *chunk locality*. Based on this observation, [16] proposed a deduplication system that breaks up an incoming stream into relatively large segments, and deduplicates each segment against only a few of the most similar previous segments. The reasons for which we adopt the principle of sparse indexing in our work are as follows:

- Chunk locality between D_{i-1} and D_i is magnificent if ΔD_i is sufficiently small;
- Segmentation is well-suited to HDFS-like distributed file systems [18,19], where each file is stored as multiple fixed-size blocks distributed to nodes.

Like sparse indexing, D-SD divides D_{i-1} and D_i into *segments* (16 MB by default), and assigns a task to each segment to get a portion of ΔD_i from the segment. In HadUP, a map task takes charge of this task. Since a segment is typically smaller than a split (i.e., the input data of a map task), a map task often handles multiple segments. HadUP creates the map tasks on both D_{i-1} and D_i. These tasks find ΔD_i and apply the map function directly to ΔD_i for update propagation.

If a task is assigned to segment *s* of D_{i-1} (or D_i), it requires the chunk information of a few segments of D_i (or D_{i-1}). In this case, we call *s* the *base segment*, and the chosen segments the *reference segments* of *s*. In the rest of this section, we first describe the two main operations of D-SD: (1) choosing the reference segments of each base segment, and (2) getting a portion of ΔD_i from the base segment. After that, we describe the overall procedure of D-SD in HadUP.

Base	Reference
u_1	v_1, v_2
u_2	v_2, v_3
u_3	v_3
u_4	v_3

(a) Reference segments of D_{i-1}.

Base	Reference
v_1	u_1
v_2	u_1, u_2
v_3	u_2, u_3, u_4
v_4	-

(b) Reference segments of D_i.

Fig. 5. An example of choosing reference segments.

4.1. Choosing reference segments

To choose proper reference segments of each base segment, we should consider the following two properties.

- **Similarity:** *A base segment should share as many chunks as possible with its reference segments.* Otherwise, D-SD fails to detect a sufficient amount of unchanged data among segments.
- **Symmetry:** *For two segments s_1 and s_2, s_1 must be the reference segment of s_2 if and only if s_2 is the reference segment of s_1.* Assume that chunk *x* appears once in each snapshot, once in $s_1 \subset D_{i-1}$ and once in $s_2 \subset D_i$. If the task assigned to s_1 applies any operation on *x*, it is obvious that the task assigned to s_2 needs to apply the same operation on *x*. This can be guaranteed only by the symmetry property.

For the similarity property, D-SD uses the sampling-based method for identifying similar segments in [16]. It chooses a small portion of the chunks in each segment as *samples*,[6] and considers two segments to be similar if they share many samples. Definitely, this method also relies on chunk locality. Assume that all sample fingerprints of D_{i-1} and D_i have already been computed. For each base segment, D-SD chooses its reference segments one at a time until the maximum allowable number of its reference segments are chosen, or when no remaining segment shares at least 1% of samples with it. At each time, D-SD chooses the segment containing the most samples that the base segment contains and the previous reference segments (of the same base segment) do not contain.

This sampling-based method, however, does not guarantee that the symmetry property is satisfied. Therefore, D-SD uses it only for one snapshot, and inverts the partial result to determine the reference segments of the remaining base segments (of the other snapshot). Fig. 5 illustrates an example where reference segments are chosen, under the assumption that D_{i-1} is divided into four segments, $\{u_1, u_2, u_3, u_4\}$, and D_i is divided into $\{v_1, v_2, v_3, v_4\}$. We can derive the result in Fig. 5(b) by inverting that in Fig. 5(a), and vice versa.

Our approach for choosing reference segments is not costly because it needs only sample fingerprints. In our evaluation with about 30 GB of the Wikipedia article dataset, at most 4 MB of sample fingerprints are sufficient to choose proper reference segments.

4.2. Getting the update from base segments

In traditional data deduplication techniques (including [16]), a chunk is ignored if the same chunk has been already processed.

[6] A chunk is chosen as a sample if its fingerprint matches a certain pattern (e.g., first *n* bits are zero).

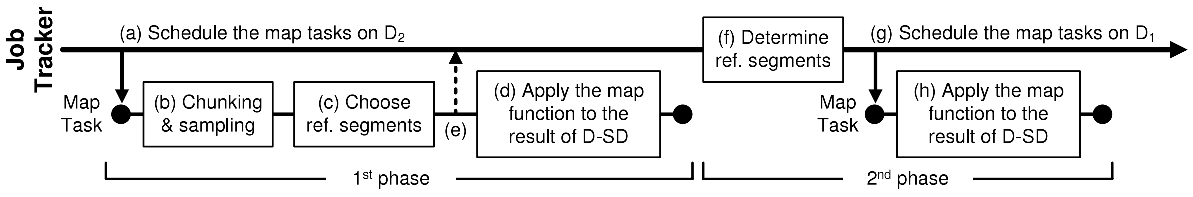

Job Tracker
(a) Schedule the map tasks on D2
Map Task
(b) Chunking & sampling
(c) Choose ref. segments
(e)
(d) Apply the map function to the result of D-SD
(f) Determine ref. segments
(g) Schedule the map tasks on D1
Map Task
(h) Apply the map function to the result of D-SD
1st phase
2nd phase

Fig. 6. Overall procedure of D-SD in the second run.

Algorithm 1 Handling base segment s_B with its reference segments S_R

```
1: procedure DedupBasedSnapshotDiff($s_B$, $S_R$)
2:     if $s_B \subset D_i$ then
3:         for all chunk $x \in s_B$ do
4:             if $\rho(x, s_B) > \rho(x, S_R)$ then
5:                 $\rho(x, \Delta D_i^+) \leftarrow \rho(x, \Delta D_i^+) + [\,\rho(x, s_B) - \rho(x, S_R)\,]$
6:             else if $\rho(x, s_B) < \rho(x, S_R)$ then
7:                 $\rho(x, \Delta D_i^-) \leftarrow \rho(x, \Delta D_i^-) + [\,\rho(x, S_R) - \rho(x, s_B)\,]$
8:             end if
9:         end for
10:    else                                                   ▷ $s_B \subset D_{i-1}$
11:        for all chunk $x \in s_B$ do
12:            $n \leftarrow$ the number of the segments in $S_R$ containing $x$
13:            if $n = 0$ then
14:                $\rho(x, \Delta D_i^-) \leftarrow \rho(x, \Delta D_i^-) + \rho(x, s_B)$
15:            else if $n > 1$ then
16:                $\rho(x, \Delta D_i^+) \leftarrow \rho(x, \Delta D_i^+) + (n-1) \cdot \rho(x, s_B)$
17:            end if
18:        end for
19:    end if
20: end procedure
```

However, this ignorance leads D-SD to an incorrect result because a snapshot is modeled as a multiset of records. For a chunk x in D_{i-1} or D_i, D-SD should satisfy the following equation:

$$\rho(x, \Delta D_i^+) - \rho(x, \Delta D_i^-) = \rho(x, D_i) - \rho(x, D_{i-1}) \quad (1)$$

where $\rho(x, S)$ denotes the multiplicity of chunk x in a multiset of records S. If a task is assigned to a base segment containing x, D-SD allows it to increase either $\rho(x, \Delta D_i^+)$ or $\rho(x, \Delta D_i^-)$.[7] Since each task performs independently, however, satisfying (1) is challenging.

Algorithm 1 describes how D-SD satisfies (1) for all chunks. In Fig. 5, assume that chunk x appears in segments u_1, v_1, and v_2, and tasks t_1, t_2, and t_3 are assigned to those segments, respectively. In this case, $\rho(x, D_i) - \rho(x, D_{i-1}) = \rho(x, v_1) + \rho(x, v_2) - \rho(x, u_1)$. If a task is assigned to any base segment of D_i (i.e., t_2 and t_3 in this example), it increases either $\rho(x, \Delta D_i^+)$ or $\rho(x, \Delta D_i^-)$, assuming that it knows all segments containing x and no other task increases any of these values (lines 4–8). For example, t_2 increases $\rho(x, \Delta D_i^+)$ by $\rho(x, v_1) - \rho(x, u_1)$ if $\rho(x, v_1) > \rho(x, u_1)$, and otherwise, increases $\rho(x, \Delta D_i^-)$ by $\rho(x, u_1) - \rho(x, v_1)$. Definitely, the result from t_2 and t_3 does not satisfy (1) as follows:

$$\begin{aligned}\rho(x, \Delta D_i^+) - \rho(x, \Delta D_i^-) &= \rho(x, v_1) + \rho(x, v_2) - 2\rho(x, u_1)\\ &\neq \rho(x, D_i) - \rho(x, D_{i-1}).\end{aligned}$$

To correct this, D-SD relies on the symmetry property (lines 12–17). t_1, which is assigned to a base segment of D_{i-1}, can find that u_1 is one of the reference segments of two other base segments (i.e., v_1 and v_2), and $\rho(x, u_1)$ is counted twice, instead of once. Therefore, t_1 increases $\rho(x, \Delta D_i^+)$ by $\rho(x, u_1)$, and as a result, D-SD satisfies (1) for x.

4.3. Overall procedure in HadUP

Now, we describe the overall procedure of D-SD in HadUP. For convenience, we focus on D-SD performed in the first and second run of the application. In subsequent runs, D-SD works similarly to the second run.

In the first run, HadUP does not need to perform D-SD, but prepares some information for the next run. Each map task performs chunking and sampling of the base segments in its split while applying the map function, and stores the chunk information of each segment into a separate HDFS file. The JobTracker gathers the sample fingerprints of D_1 from all tasks, and stores them into a single HDFS file. It also stores the split information of D_1 in the HDFS to create the map tasks on D_1 in the second run.

In the second run, HadUP creates the map tasks on both D_1 and D_2 to detect ΔD_2. When the application starts to run, however, there is no sample fingerprint of D_2, therefore HadUP cannot determine the reference segments of any base segment. Instead of running an additional job to compute the sample fingerprints of D_2, HadUP divides D-SD into two phases and processes one snapshot in each phase, as shown in Fig. 6. As a result, each snapshot is read only once.

In the first phase, the JobTracker schedules the map tasks only on D_2 (Fig. 6(a)). Each task first performs chunking and sampling of the base segments in its split (Fig. 6(b)). Unlike in the first run, it does not apply the map function; all the input records are temporarily cached in memory and reused later. After processing the whole split, the task chooses the reference segments of each base segment, following the similarity property (Fig. 6(c)). This is possible because the sample fingerprints of D_1 have already been computed in the first run. Using the chunk information of the chosen reference segments, the task finds the records belonging to ΔD_2 using Algorithm 1, and applies the map function to them (Fig. 6(d)). Since all the input records are cached and reused, D_2 is not read again. The information required for the next run (i.e., the chunk information, sample fingerprints, and split information of D_2) is stored in the HDFS, as was done in the first run.

D-SD enters the second phase after the JobTracker finds the reference segments of all base segments of D_2. In the first phase, each task sends the IDs of those segments to the JobTracker through heartbeat messages (Fig. 6(e)). At the beginning of the second phase, therefore, the JobTracker can determine the reference segments of all base segments of D_1 based on the symmetry property (Fig. 6(f)). After that, it starts scheduling the map tasks on D_1 (Fig. 6(g)), which find the records belonging to ΔD_2 and apply the map function to them (Fig. 6(h)).

5. Update propagation

Since the success of Google's MapReduce, data parallel programming models have been widely used for big data processing

[7] Let $map(x)$ denote the result of applying the map function to each record in x. To increase $\rho(x, \Delta D_i^+)$ by one, the task emits $map(x)$ once as the $(+)$-typed records. To increase $\rho(x, \Delta D_i^-)$, it emits $map(x)$ as the $(-)$-typed records.

Table 1
Variables and notations used in this section.

Variable	Description
I	The input dataset of a job
M	The intermediate dataset of a job
R	The output dataset of a job
$map(\langle k, v\rangle)$	The result of applying the map function to record $\langle k, v\rangle$
$key(S)$	A set of keys that appears in a multiset of records S
$sub(k, S)$	A sub-multiset of S that contains all records with key k
$reduce(k, S)$	The result of applying the reduce function to $sub(k, S)$. It is an empty set if there is no record with key k in S.
$combine(k, S)$	The result of applying the combiner function to $sub(k, S)$. It is an empty set if there is no record with key k in S.

Algorithm 2 Update propagation with the map function

```
1: function MapUP($key$, $value$, $type$)
2:     $S \leftarrow$ Map($key$, $value$)          ▷ Compute $map(\langle key, value\rangle)$
3:     for all $\langle k, v\rangle \in S$ do
4:         Emit($k$, $v$, $type$)
5:     end for
6: end function
```

[1–3,5,8,9]. Using these models, an application is expressed as a workflow of *data parallel operations*, where an operation transforms one or more input datasets into one output dataset. For each operation, the same computation is applied to each input record (e.g., map and select) or each group of input records with the same key (e.g., reduce, join, and cogroup). Based on these characteristics, we design update propagation for the MapReduce model; the independence between computations allows each operation to compute the update of the output dataset if those of the corresponding input datasets are explicitly given. Note that our design can be easily applied to other programming models based on data parallelism.

In the rest of this section, we describe update propagation for the MapReduce model, and its efficient implementation for large-scale Hadoop clusters. Table 1 lists the variables and notations used in this section.

5.1. Update propagation with the map function

Let I and M denote the input and intermediate dataset of a job, respectively. Since the map function is applied to each input record, ΔM_i is dependent only on ΔI_i. Therefore, we can derive ΔM_i as follows:

$$\Delta M_i^+ = \bigcup_{\langle k,v\rangle \in \Delta I_i^+} map(\langle k, v\rangle)$$

$$\Delta M_i^- = \bigcup_{\langle k,v\rangle \in \Delta I_i^-} map(\langle k, v\rangle).$$

Algorithm 2 describes the implementation of this approach in HadUP. It is the same as the normal execution of the map function, except for handling the record types. If the job is involved in D-SD, the type of each input record is determined, as discussed in Section 4. For the other jobs, the user should override the function that determines the record types (i.e., `setInputRecordTypeSelectorClass` in Fig. 4). HadUP allows a file to contain records of only one type. This policy makes a map task handle one type so that overriding the above function is easy. It also enables the data load/store phases of existing Hadoop jobs to be reused in HadUP without change.

5.2. Update propagation with the reduce function

Let M and R denote the intermediate and output dataset of a job, respectively. Since the reduce function is applied to each group

Algorithm 3 Update propagation with the reduce function

```
1: function ReduceUP($key$, $values^T$)
2:     $values \leftarrow \emptyset$
3:     if $key$ exists $M_{i-1}$ then
4:         $\langle key, values\rangle \leftarrow$ Read($key$, $M_{i-1}$)  ▷ Read $sub(key, M_{i-1})$
5:         $S \leftarrow$ Reduce($key$, $values$)  ▷ Compute $reduce(key, M_{i-1})$
6:         for all $\langle k, v\rangle \in S$ do
7:             Emit($k$, $v$, $-$)
8:         end for
9:     end if
10:    $values \leftarrow$ Merge($values$, $values^T$)  ▷ Compute $sub(key, M_i)$
11:    if $values \neq \emptyset$ then
12:        $S \leftarrow$ Reduce($key$, $values$)      ▷ Compute $reduce(key, M_i)$
13:        for all $\langle k, v\rangle \in S$ do
14:            Emit($k$, $v$, $+$)
15:        end for
16:    end if
17:    Update($M_{i-1}$, $key$, $values$)           ▷ Preserve $sub(key, M_i)$
18: end function
```

of intermediate records with the same key, we can derive ΔR_i as follows:

$$\Delta R_i^+ = \bigcup_{k \in key(\Delta M_i)} reduce(k, M_i)$$

$$\Delta R_i^- = \bigcup_{k \in key(\Delta M_i)} reduce(k, M_{i-1}).$$

Algorithm 3 describes the implementation of this approach in HadUP, where *values* represents a list of values and $values^T$ represents a list of value/type pairs. To inform M_{i-1} of the reduce tasks, the user should specify the ID of the same job executed in the last run (i.e., `setBaseJobID` in Fig. 4). In HadUP, a reduce task produces two output files, one for each record type, because of the policy described in Section 5.1. In addition, the task buffers the output records temporarily in memory, and ignores all pairs of records with the same key/value, but different types. This implementation is effective with some reduce functions that, for key k, $reduce(k, M_{i-1})$ and $reduce(k, M_i)$ share many records (e.g., reduce-side join).

5.2.1. Partition files

For efficient update propagation with the reduce function, it is important for HadUP to read records quickly from M_{i-1} (line 4) and to transform M_{i-1} into M_i (line 17), which is needed for subsequent runs. To achieve this, HadUP maintains M using *partition files*, which are similar to Hadoop's MapFile. Each partition file contains a portion of ΔM_i and the indices for each key. Fig. 7 illustrates the layout of the partition file. A Group field contains all records with the same key in ΔM_i. The last property is used for the indirect files (which will be described in Section 5.2.2).

Each reduce task often creates one partition file with the shuffled data,[8] and reads M_{i-1} from some of the previously created partition files. HadUP does not allow the user to change the partitioner function used in the first run so that no task shares the partition files. To retrieve $sub(k, M_{i-1})$ for key k, a reduce task reads the records in all the Group fields associated with k while ignoring all pairs of records with the same value but different types. Since the intermediate data is always processed in increasing key order,

[8] A reduce task creates more than one partition file when the shuffled data is large. In Hadoop, the shuffled data is stored in the task's memory, but spilled to a local disk if it does not fit in memory. HadUP replaces this spill file with the partition file.

the partition file is always read sequentially, and all indices in it are created without additional sorting.

To improve the performance of reading the partition files, HadUP schedules each reduce task on the node where the task assigned to the same partition was executed in the last run. Since the HDFS always stores one replica of a block in the node where the writer runs, this policy prevents reduce tasks from reading the partition files remotely. If the target node is failed or overloaded, HadUP schedules the task on the node storing the most blocks of the required partition files.

5.2.2. Indirect files

If R is the application result, each reduce task creates an *indirect file* instead of the output file with the $(-)$-typed records. For key k in M_i, the indirect file contains the location of $reduce(k, M_i)$ (i.e., file name, offset, and length); the last property of the Group field is used to store this information for subsequent runs. When the user reads this file, the HadUP client library redirects the read requests to the appropriate output files created previously. Therefore, the user can get the updated result (i.e., R_i) as if they were using Hadoop.

5.2.3. Compaction

As the application is executed repeatedly, a reduce task might read a number of partition files if there is no additional management for them. To avoid this, HadUP bounds the number of partition files by executing *compaction*. Whenever the number of partition files for a partition reaches a threshold, HadUP merges them into a large one in the background. Compaction also reduces space overhead because the compacted partition file does not need to contain $(-)$-typed records and record types.

If a job produces the indirect files, HadUP also executes compaction of the job's output files for better performance retrieving the updated result. After compaction, therefore, the indirect files become unnecessary, as they were at the end of the first run.

5.3. Update propagation with the combiner function

The combiner function is effective in reducing data transfer between the map and reduce tasks, but complicates update propagation. Recall that, unlike the reduce function, the combiner function aggregates a portion of the values associated with each intermediate key. Because of this property, the strategy described in the previous section is not suitable for the combiner function; the $(+)$-typed records can be handled, but the $(-)$-typed ones cannot. If the combiner function is applied only to the $(+)$-typed records, update propagation with the following reduce function becomes impossible.[9] One option is to disable the combiner function, but it causes a large data transfer between the map and reduce tasks, which can result in a significantly poor performance.

Currently, HadUP supports a popular class of the combiner functions, which is distributive with respect to both insertion and deletion. These functions can compute the new output with the old output and the input change. Input deletion is processed using their *inverse functions*. For example, "sum" is one of the most widely used combiner function belonging to this class, and its inverse function is "additive inverse".

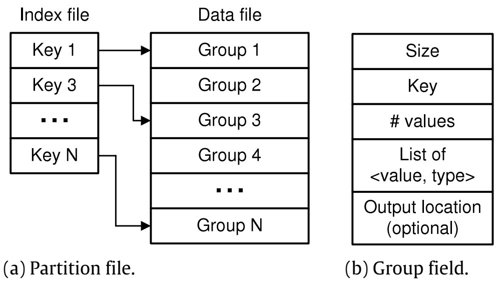


Fig. 7. Layout of the partition file.

Based on the definition of the combiner functions, we can derive ΔR_i as follows:

$$\Delta R_i^+ = \bigcup_{k \in key(\Delta M_i)} reduce(combine(k, M_i))$$

$$\Delta R_i^- = \bigcup_{k \in key(\Delta M_i)} reduce(combine(k, M_{i-1})).$$

If the given combiner function is distributive with respect to both insertion and deletion, $sub(k, M_i)$ is not essential to compute $combine(k, M_i)$. Let C be the combined intermediate dataset, where $sub(k, C_i)$ is defined as $combine(k, M_i)$. Then, $sub(k, C_i)$ can be derived as follows:

$$\begin{aligned} sub(k, C_i) = {} & combine(sub(k, C_{i-1}) \cup sub(k, \Delta M_i^+) \\ & \cup inv(k, \Delta M_i^-)) \end{aligned} \tag{2}$$

where $inv(k, \Delta M_i^-)$ denotes the result of applying the inverse function to $sub(k, \Delta M_i^-)$. That is,

$$inv(k, \Delta M_i^-) = \{\langle k, v^{-1}\rangle | \langle k, v\rangle \in sub(k, \Delta M_i^-)\}$$

where v^{-1} is the inverse of v. Since $reduce(combine(k, M_i))$ is equal to $reduce(sub(k, C_i))$, we can derive ΔR_i without M_i but with C_i.

Consider the well-known word count job, whose combiner function is "sum". Assume that word w appeared twice in the input documents in the first run. Then, $sub(w, M_1) = \{\langle w, 1\rangle, \langle w, 1\rangle\}$ and $sub(w, C_1) = \{\langle w, 2\rangle\}$. If one of the w's has been removed before the second run (i.e., $sub(w, \Delta M_2) = \{\langle w, 1, -\rangle\}$), $inv(w, \Delta M_2^-)$ is derived as $\{\langle w, -1\rangle\}$. In this case, therefore, $sub(w, C_2)$ is derived from (2) as follows:

$$sub(w, C_2) = combine(\{\langle w, 2\rangle, \langle w, -1\rangle\}) = \{\langle w, 1\rangle\}.$$

This is equal to the result without the combiner function.

Unfortunately, (2) is not always valid. For any key k, $sub(k, C_i)$ is always derived as a non-empty set from (2) if $sub(k, C_{i-1})$ is a non-empty set. Assume that, in the above word count example, all of the w's have been removed before the second run. In this case, $sub(w, C_2)$ must be an empty set, but it is derived from (2) as follows:

$$\begin{aligned} sub(w, C_2) &= combine(\{\langle w, 2\rangle, \langle w, -1\rangle, \langle w, -1\rangle\}) \\ &= \{\langle w, 0\rangle\} \neq \emptyset. \end{aligned}$$

To avoid this case, there are two possible solutions. The first is to check whether the result of (2) is valid. In Hadoop, the combiner function of the word count job always produces non-zero values for any word. Therefore, we can regard $sub(w, C_i)$ as an empty set if $\{\langle w, 0\rangle\}$ is derived from (2). Though simple, this correction is impossible for algorithms without such property.[10] The second

9 Assume that a reduce task encounters $\langle k, v, -\rangle$ in ΔM_i. If the combiner function is not used, HadUP guarantees that the task gets $\langle k, v, +\rangle$ from the partition files (i.e., M_{i-1}) or from the shuffled data (i.e., ΔM_i^+). With the combiner function, however, this is not guaranteed because $\langle k, v\rangle$ may have been combined with other records.

10 For example, consider a job whose combiner function is also "sum", but whose map function emits any number as a value.

Algorithm 4 Update propagation with the combiner function that is distributive with respect to insertion and deletion

```
1: function MapUP($key$, $value$, $type$)
2:     $S \leftarrow$ Map($key$, $value$)                         ▷ Compute $map(\langle key, value \rangle)$
3:     if $type = +$ then
4:         for all $\langle k, v \rangle \in S$ do
5:             Emit($k$, $v$, 1)
6:         end for
7:     else
8:         for all $\langle k, v \rangle \in S$ do
9:             $v^{-1} \leftarrow$ Inverse($v$)                   ▷ Compute $inv(\langle k, v \rangle)$
10:            Emit($k$, $v^{-1}$, $-1$)
11:        end for
12:    end if
13: end function
14: function CombinerUP($key$, $values^C$)
15:     $values \leftarrow$ Values($values^C$)
16:     $count \leftarrow$ Sum($values^C$)
17:     $\langle k, v \rangle \leftarrow$ Combiner($key$, $values$)
18:     Emit($k$, $v$, $count$)
19: end function
20: function ReduceUP($key$, $values^C$)
21:     if $key$ exists $C_{i-1}$ then
22:         $\langle key, value, count \rangle \leftarrow$ Read($key$, $C_{i-1}$)   ▷ Read $sub(key, C_{i-1})$ where $count = |sub(key, M_{i-1})|$
23:         $S \leftarrow$ Reduce($key$, $\{value\}$)            ▷ Compute $reduce(key, C_{i-1})$
24:         for all $\langle k, v \rangle \in S$ do
25:             Emit($k$, $v$, $-$)
26:         end for
27:         $values^C \leftarrow$ Merge($values^C$, $\{\langle value, count \rangle\}$)
28:     end if
29:     $\langle key, value, count \rangle \leftarrow$ CombinerUP($key$, $values^C$)   ▷ Compute $sub(key, C_i)$ where $count = |sub(key, M_i)|$
30:     if $count > 0$ then
31:         $S \leftarrow$ Reduce($key$, $\{value\}$)            ▷ Compute $reduce(key, C_i)$
32:         for all $\langle k, v \rangle \in S$ do
33:             Emit($k$, $v$, $+$)
34:         end for
35:     end if
36:     Update($C_{i-1}$, $key$, $value$, $count$)              ▷ Preserve $sub(key, C_i)$
37: end function
```

solution, which is more general, is to track $|sub(k, M_i)|$, the number of records in $sub(w, M_i)$. If it is zero, M_i contains no record with key w, and therefore, $sub(w, C_i)$ must be an empty set, regardless of the result of (2).

Algorithm 4 describes our implementation to support the combiner functions in HadUP, where $values^C$ represents a list of value/count pairs. In the map stage, HadUP emits the inverse values of the ($-$)-typed map output records (line 9). Note that the user should write the inverse function. Then, HadUP assigns a counter to each intermediate record instead of type (line 5, 10), which will be used to track $|sub(k, M_i)|$ for each intermediate key k. For the ($+$)-typed records, this counter is initialized to 1, and otherwise, -1.

In the reduce stage, HadUP computes both $sub(k, C_i)$ and $|sub(k, M_i)|$ before invoking the reduce function for key k (line 29). The counters of intermediate records are added up during the combiner execution (line 16), thus $|sub(k, M_i)|$ can be computed. If it becomes zero, the reduce function is bypassed. HadUP maintains $sub(k, C_i)$ and $|sub(k, M_i)|$ using the partition files. If several partition files contain the information for the same key, HadUP takes only the information from the newest one and ignores the rest.

6. Evaluation

We evaluated HadUP on a 16-node cluster from the Amazon Elastic Compute Cloud (EC2). Each node is the High-CPU medium instance (c1.medium) that contains two virtual cores (with 2.5 EC2 Compute Units each) and 1.7 GB of memory, and provides moderate I/O performance. We configured each TaskTracker to run two map tasks and one reduce task concurrently, and limited the heap memory of each task to 512 MB. We also assigned a sufficiently large input to each task. To each map task, we assigned 128 MB of data by setting the HDFS block size to 128 MB. To each reduce task, we assigned a few hundred megabytes of data by carefully choosing the number of reduce tasks of each job. All other configurations were set to the Hadoop's default values. Note that, in this environment, we can suppose that task-level memoization based approaches perform similarly to (or worse than) Hadoop, as discussed in Section 2.3.

Table 2
Hadoop applications used in our evaluation.

Application	Domain	# Jobs
Word count (WC)	Text analytics	1
Word co-occurrence (WO)	Natural lang. processing	1
Clustering coefficient (CC)	Link analysis	7
PageRank (PR)	Graph algorithm	5

Table 2 lists the Hadoop applications used in our evaluation,[11] which are representative for different domains. Only a single job was used to implement WC and WO. Though both are similar, WO shuffles much more data than WC (about 26×). CC and PR consist of several jobs. In CC, two jobs are dominant, which are executed in series to count triangles in the input graph. The reason is that the former job produces a significant amount of output data, almost 20 times that of its input data. In PR, we configured two jobs to run 10 times repeatedly for the PageRank computation (i.e., 10 iterations). In all applications, we used the same code for each job on both Hadoop and HadUP.

We used two performance metrics: *time* and *work*. Time refers to the completion time of each application, while work refers to the sum of the running time of all tasks. The tendency of both time and work is similar, but work is less affected by resource heterogeneity in EC2 and load imbalance. The difference between time and work, therefore, tends to be larger, as the application consists of more jobs.

In the rest of this section, we evaluate HadUP performance for incremental processing, and then analyze the performance impact of D-SD and update propagation on the overall HadUP performance.

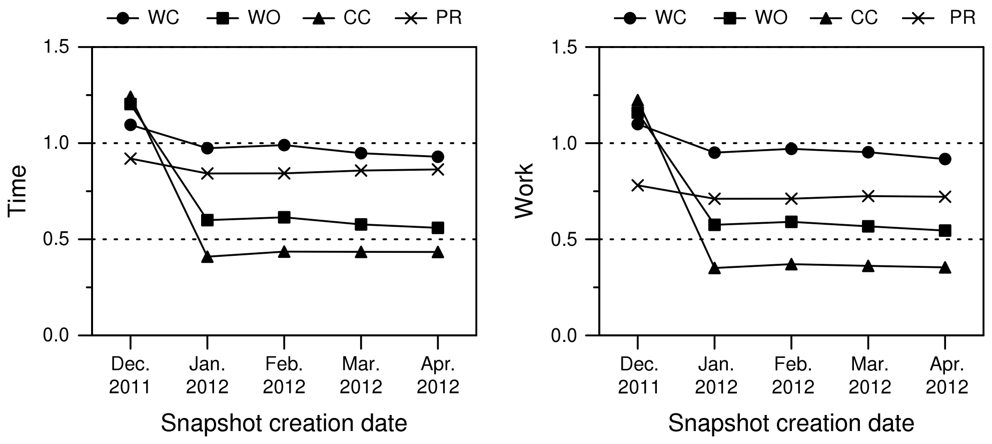
6.1. Incremental processing with HadUP

To evaluate HadUP performance for incremental processing, we ran the applications with five snapshots of the Wikipedia English article dataset, created monthly from Dec. 2011 to Apr. 2012. The snapshot size increases almost linearly; the oldest snapshot is 32.8 GB and the newest one is 34.9 GB in size. We set the average chunk size of D-SD to 256 bytes, which shows the best performance. To reduce the average chunk size to such a small size, we implemented a customized input parser for the Wikipedia English article dataset, which produces about 100-byte records on average.[12]

Fig. 8 shows HadUP's incremental performance, normalized to Hadoop's non-incremental performance. Except in the first run, HadUP outperforms Hadoop in all subsequent runs. The

[11] We implement all these applications on our own. We refer to [20] and [21] to implement CC and PR, respectively.

[12] When processing text data with Hadoop, `LineRecordReader` is often used to parse the data, which returns a line as a record. Using this parser, however, we cannot sufficiently reduce the average chunk size, because each paragraph in the Wikipedia article dataset is expressed as a single line whose average is over 1 KB. Our customized input parser approximately breaks each paragraph into multiple sentences. It returns the article name as key and each sentence in the article as value. For other information such as wiki metadata, it returns a line as value.



(a) Time. (b) Work.

Fig. 8. HadUP performance for incremental processing on the Wikipedia English article dataset.

performance benefit of HadUP varies depending on application characteristics. Among the four applications, WC is the most computationally inexpensive and involves the smallest amount of data transfer. The benefit from update propagation with WC is not sufficiently higher than the cost of D-SD, therefore HadUP performs similarly to Hadoop with WC. In contrast, WO and CC are computationally expensive and involve a large amount of data transfer. Consequently, HadUP performs much better than Hadoop with WO and CC, up to 1.8× and 2.5× better respectively. With PR, HadUP outperforms Hadoop, but the performance benefit is relatively low. The reason is that all PageRank computations should always be re-executed because the total number of pages affects the PageRank value of each page. Therefore, the benefit by avoiding unnecessary computation mostly comes from incrementally building the initial PageRank table and the inter-page link table, which occupies only 11% of the total computation in our environment.

In the first run, HadUP cannot take any advantage of update propagation, but must do additional work in preparation for the second run: the input data chunking (for D-SD) and the creation of partition files (for update propagation). This is a cause of the degraded performance of HadUP in the first run. The chunking cost is affected only by the input size, but HadUP can perform worse if the application consists of jobs processing larger intermediate data. With WO and CC, HadUP performance is degraded by up to 25% compared with that of Hadoop. However, we argue that this one-time cost is acceptable because it is amortized and transformed into performance benefits in subsequent runs.

Interestingly, HadUP outperforms Hadoop with PR even in the first run. In many iterative algorithms, a significant fraction of the processed data remains invariant across iterations, thus it is effective to avoid reprocessing the invariant data on each iteration [21]. In PR, a number of inter-page links are invariant but processed at each iteration. If we regard an iteration of PR as a *sub-application*, update propagation can avoid reprocessing them after the first iteration. This is the main factor affecting the performance improvement of HadUP with PR.

6.2. Impact of D-SD

Next, we explore the impact of D-SD on HadUP performance. We used the snapshot of the Wikipedia English article dataset that was taken on Dec. 2011 as input in the first run, and that taken on either Jan. 2012 or Apr. 2012 as input in the second run.

Fig. 9 shows the update size and the amount of chunk information while varying the average chunk size of D-SD. The results are normalized to the size of the older snapshot, and NF in Fig. 9(a) denotes the update size computed by comparing both snapshots record-by-record. The results reveal the tradeoff between the effectiveness (Fig. 9(a)) and the efficiency (Fig. 9(b)) of D-SD. With smaller chunks, D-SD finds more unchanged data, but also handles more chunk information. In our evaluation, a map task reads up to 30 MB of chunk information when the average chunk size is set to 128 bytes.

Fig. 10 shows HadUP performance in the second run versus the average chunk size. The results are normalized to Hadoop's non-incremental performance. The trends of both time and work are the same, so we omit the measurement of work. In our environment, using 256-byte chunks achieves the best performance because the effectiveness and efficiency of D-SD are the most balanced. However, multi-job applications such as CC and PR are less sensitive to the chunk size than single-job applications such as WC and WO. D-SD affects only the jobs processing the application input. In CC and PR, such jobs take only 9% and 8% of the total runtime in our environment, respectively. As a result, the impact of D-SD on HadUP performance is not severe. Since multi-job applications are common in practice, most applications are insensitive to the average chunk size of D-SD.

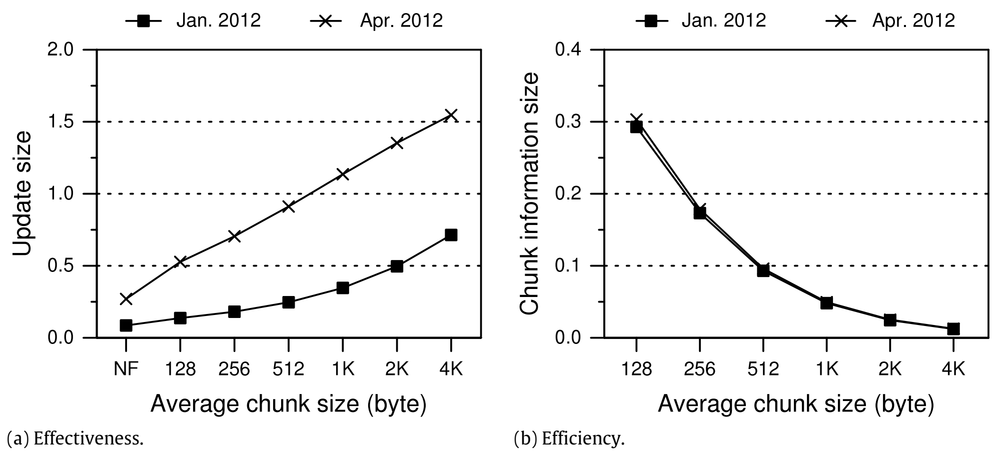
6.3. Impact of update propagation

Now, we evaluate update propagation without D-SD. In this section, we use the term *update ratio* to represent how much the application input changes. We say that the update ratio is r% when r% of the input data in the last run is changed into new data. In our experiments, we used multiple, same-sized files as input in the first run, and changed r% of these files to new ones in the second run to set the update ratio to r%. Therefore, HadUP handles only $2r$% of the input files used by Hadoop; the newly added files are treated as the (+)-typed data and the removed ones as the (−)-typed data.

For WC and WO, we used the Wikipedia English article dataset snapshotted on Dec. 2011, but used the LiveJournal dataset[13] for CC and PR. The LiveJournal dataset is a social network graph dataset that contains the friendship network of an on-line community. We enlarged the dataset without changing the network structure as was done in [21], and the extended dataset is about 18 GB in size.

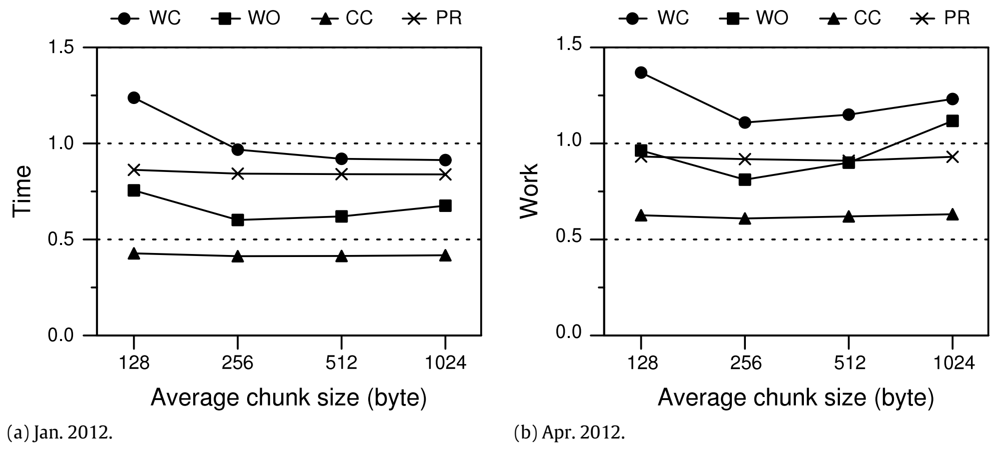
Fig. 11 shows HadUP performance in the second run while varying the update ratio. The results are normalized to Hadoop's non-incremental performance. For all applications, HadUP performance

[13] http://snap.stanford.edu/data/.



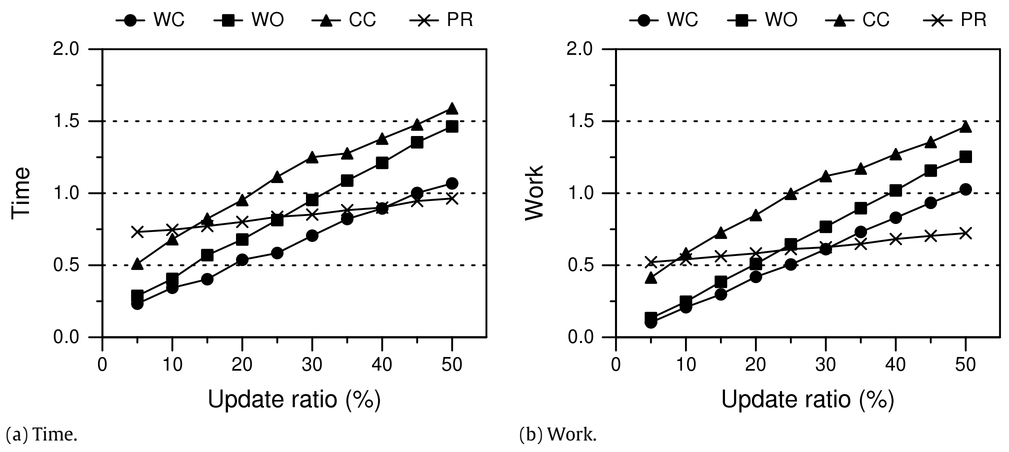
(a) Effectiveness. (b) Efficiency.

Fig. 9. Effect of the average chunk size on D-SD.



(a) Jan. 2012. (b) Apr. 2012.

Fig. 10. Effect of the average chunk size on HadUP performance.

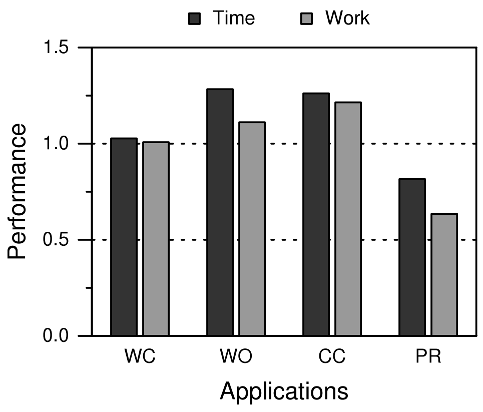


(a) Time. (b) Work.

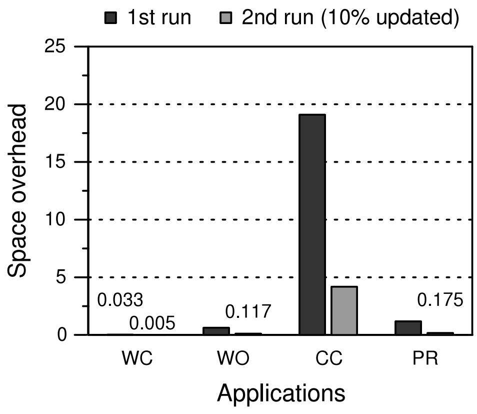
Fig. 11. Update propagation performance versus the update ratio.

degrades almost linearly as the update ratio increases. Though it performs even worse than Hadoop with a high update ratio, HadUP still considerably outperforms Hadoop with a small update ratio as claimed. The benefit from update propagation depends on how much computation the given update involves. WC shows the best performance because a single input record is dependent on only a few output records on average. WO shows similar performance in the presence of a small update, but requires more computation as the update ratio increases. Both CC and PR require relatively more computation with a small update. In particular, the result of PR is close to one because all PageRank computations are always re-executed. However, PR is least affected by the update ratio, because the performance benefit for PR comes from avoiding reprocessing the inter-page link table in each iteration, as discussed in Section 6.1.

Update propagation incurs space overhead to maintain the partition files. Fig. 12 shows HadUP performance in the first run, normalized to that of Hadoop, and the associated space overhead



(a) Time/work.



(b) Space overhead.

Fig. 12. HadUP performance in the first run.

Table 3
Programming effort (lines of code).

Application	Jobs	Driver (Hadoop)	Driver (HadUP)
WC	52	17	21
WO	77	17	21
CC	969	73	89
PR	512	72	117

normalized to the application input size. Without D-SD, performance overhead in the first run mainly comes from creating the partition files. For all applications, this overhead is not significantly high, as shown in Fig. 12(a). In contrast, space overhead varies substantially depending on application characteristics; it can be as high as 20 times the input data size (with CC), as shown in Fig. 12(b). Inevitably, HadUP consumes more space as the application produces more intermediate data. Note that such space overhead does not cause a significantly long runtime. The growth rate of space usage is reduced in subsequent runs, especially with small updates. Furthermore, compaction can prevent space usage from growing without bounds.

6.4. Programming effort

We measure the programming effort by lines of code, and the result shows that HadUP requires only a small amount of extra programming cost, compared with Hadoop. Table 3 shows the lines of code for each application without our customized input parser. Most applications require only a few more lines of code for their drivers, while PR requires more in order to avoid reprocessing the inter-page link table in each iteration.

7. Related work

Large-scale incremental processing. There are several approaches for large-scale incremental processing. They are divided into two categories: transparent and non-transparent approaches. Our approach represents a middle ground between the two categories. HadUP provides high performance and a sufficient level of transparency.

Most transparent approaches are based on task-level memoization. Comet [14] leverages the previous intermediate results of the same query in an overlapping window of data. Nectar [11] provides a general, datacenter-wide solution by managing the cached results of the sub-computations and by automatically rewriting applications to reuse them. Both Incoop [10] and IncMR [15] exploit task-level memoization in the context of MapReduce. They preserve intermediate data and reuse them whenever the reduce stage of the future job needs them. Incoop, furthermore, tries to reduce the number of tasks to be re-executed by applying content-based chunking to the HDFS. While providing full transparency, none of them can save a sufficient amount of computation in most cases because of the inherent disadvantage of task-level memoization. In contrast, HadUP detects and computes the change of datasets at a fine-grained level using D-SD and update propagation. As a result, it provides higher performance, especially in an environment where task-level memoization has no benefit.

Non-transparent approaches include Percolator [13] and Continuous Bulk Processing (CBP) [12]. Percolator is designed to process many small updates concurrently to a large dataset in Bigtable [22]. Whenever a user-specified column changes, an observer is invoked to perform computation with the change. It can trigger another observer by writing to the table. A Percolator application, therefore, consists of a series of observers. CBP integrates state into distributed computing using a groupwise operator that takes state as an explicit input. It also offers dataflow management primitives to accommodate continuous execution. While providing high efficiency, these approaches require that the applications be rewritten using their interfaces. The more fundamental disadvantage is that they require the user to devise dynamic algorithms, which are known to be difficult to develop and implement [10]. Compared with non-transparent approaches, HadUP provides higher transparency because existing Hadoop jobs can be reused.

Incremental view maintenance. Database systems maintain a view, a derived relation defined in terms of base relations, to improve query performance [23]. Since recomputing a view from scratch is often unaffordable, these systems incrementally update a view as the contents evolve.

Update propagation is inspired by incremental view maintenance. Abstractly, a MapReduce program can be seen as a view definition and the computed result as a materialized view [24]. Palpanas et al. [25] describe incremental maintenance for non-distributive aggregate functions. While their approach is similar to update propagation, it is tailored to RDBMS while ours is suited to big data processing. Jörg et al. [24] discuss incremental view maintenance in a Hadoop/HBase software stack. However, their approach is valid only if the reduce function is distributive with respect to both insertion and deletion. Compared with this approach, HadUP supports a broader class of algorithms.

Stream processing. Traditional stream processing systems [26,27] and distributed stream processing systems [28,29] focus on minimizing latency. Differently, to improve throughput, HadUP deals with data in a batch fashion.

MapReduce Online [30] attempts to unify stream processing and the MapReduce paradigm by pipelining between operations. Pipelining enables the production of approximate outputs for online aggregation and continuous queries, but cannot produce exact results without recomputation from scratch. In contrast, HadUP always produces exact results using the indirect files. Nova [31] is a workflow manager that supports batched incremental processing for continually arriving data. It requires some specialized functions to manage updates; for example, user-specified merge functions for applying updates to base data. HadUP is free from such constraints.

Data deduplication. Data deduplication is mainly used to improve storage utilization or to reduce the amount of network data transfers. As datasets grow larger, the main challenge in this context is how to manage chunk information efficiently using insufficient memory. Zhu et al. [32] and sparse indexing [16] exploit chunk locality in backup data streams. Extreme Binning [33] exploits file similarity for fine-grained, low-locality backup workloads consisting of files, and SiLo [34] adopts both locality and similarity. All of these techniques are proposed for data deduplication by a single computing node. [33] proposes a mechanism to perform Extreme Binning in a distributed system, but it is ineffective with large files. In this work, we extended the principle of sparse indexing to a distributed environment so that D-SD can effectively handle big data.

8. Conclusions

We have presented HadUP (Hadoop with Update Processing), a modified Hadoop architecture tailored to large-scale incremental processing with conventional MapReduce algorithms. Through an analysis of task-level memoization, we find that it is essential to detect and compute the change of datasets at a fine-grained level for efficient incremental processing. To reduce the granularity, HadUP adopts two techniques: a deduplication-based snapshot differential algorithm (D-SD) and update propagation. D-SD extends the principle of sparse indexing [16] in order to detect the change of the application input in large-scale Hadoop clusters, and update propagation exploits data parallelism in the MapReduce programming model to compute the updated result efficiently. As a result, HadUP shows up to 2.5× speedup over Hadoop in an environment where task-level memoization has no benefit. Though HadUP incurs a penalty in the first run (up to 25% with respect to Hadoop), it is just a one-time cost when no computation can be reused. Furthermore, HadUP requires only a small amount of extra programming cost so that the development of HadUP applications is quite easy.

In future work, we would like to improve the efficiency of HadUP for a broad class of algorithms. For example, we plan to extend HadUP to support all classes of the combiner functions, including those that are not distributive with respect to deletion (e.g., max and min). We also plan to port Hadoop-based NoSQL systems, such as Pig [6] and Hive [7], to run on HadUP. Since their queries are compiled into Hadoop jobs, we expect that HadUP can process unmodified (or little modified) queries in an incremental manner.

References

[1] Apache Hadoop. http://hadoop.apache.org/.

[2] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: Proc. USENIX Symp. Operating Systems Design and Implementation, OSDI, Dec. 2004.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: Proc. ACM European Conf. Computer Systems, EuroSys, Mar. 2007.

[4] 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.

[5] 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. USENIX Symp. Networked Systems Design and Implementation, NSDI, Mar. 2011.

[6] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig Latin: a not-so-foreign language for data processing, in: Proc. ACM Int'l Conf. Management of Data, SIGMOD, Jun. 2008.

[7] 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.

[8] 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.

[9] 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.

[10] P. Bhatotia, A. Wieder, R. Rodrigues, U.A. Acar, R. Pasquini, Incoop: MapReduce for incremental computations, in: Proc. ACM Symp. Cloud Computing, SoCC, Oct. 2011.

[11] P.K. Gunda, L. Ravindranath, C.A. Thekkath, Y. Yu, L. Zhuang, Nectar: automatic management of data and computation in data centers, in: Proc. USENIX Symp. Operating Systems Design and Implementation, OSDI, Oct. 2010.

[12] D. Logothetis, C. Olston, B. Reed, K.C. Webb, K. Yocum, Stateful bulk processing for incremental analytics, in: Proc. ACM Symp. Cloud Computing, SoCC, Jun. 2010.

[13] D. Peng, F. Dabek, Large-scale incremental processing using distributed transactions and notifications, in: Proc. USENIX Symp. Operating Systems Design and Implementation, OSDI, Oct. 2010.

[14] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, L. Zhou, Comet: batched stream processing for data intensive distributed computing, in: Proc. ACM Symp. Cloud Computing, SoCC, Jun. 2010.

[15] C. Yan, X. Yang, Z. Yu, M. Li, X. Li, IncMR: incremental data processing based on MapReduce, in: Proc. Int'l Conf. Cloud Computing, CLOUD, Jun. 2012.

[16] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, P. Camble, Sparse indexing: large scale, inline deduplication using sampling and locality, in: Proc. USENIX Conf. File and Storage Technologies, FAST, Feb. 2009.

[17] K.-H. Lee, Y.-J. Lee, H. Choi, Y.D. Chung, B. Moon, Parallel data processing with MapReduce: a survey, ACM SIGMOD Record 40 (4) (2011) 11–20.

[18] 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.

[19] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google file system, in: Proc. ACM Symp. Operating Systems Principles, SOSP, Oct. 2003.

[20] J. Cohen, Graph twiddling in a MapReduce world, Computational Science & Engineering (2009).

[21] Y. Bu, B. Howe, M. Balazinska, M. Ernst, HaLoop: efficient iterative data processing on large clusters, in: Proc. VLDB Endow., Sep. 2010.

[22] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, in: Symp. Operating Systems Design and Implementation, OSDI, Nov. 2006.

[23] A. Gupta, I.S. Mumick, Maintenance of materialized views: problems, techniques, and applications, IEEE Data Engineering Bulletin 18 (2) (1995) 3–18.

[24] T. Jörg, R. Parvizi, H. Yong, S. Dessloch, Incremental recomputations in MapReduce, in: Proc. Int'l Workshop on Cloud Data Management, CloudDB, Oct. 2011.

[25] T. Palpanas, R. Sidle, R. Cochrane, H. Pirahesh, Incremental maintenance for non-distributive aggregate functions, in: Proc. VLDB Endow., Aug. 2002.

[26] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, J. Widom, STREAM: the Stanford data stream management stream, in: Data Stream Management: Processing High-Speed Data Streams, Springer, 2009.

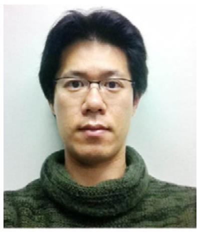
[27] J. Chen, D.J. Dewitt, F. Tian, Y. Wang, NiagaraCQ: a scalable continuous query system for internet databases, in: Proc. ACM Int'l Conf. Management of Data, SIGMOD, May 2000.

[28] Storm. https://github.com/nathanmarz/storm/.

[29] L. Neumeyer, B. Robbins, A. Nair, A. Kesari, S4: distributed stream computing platform, in: IEEE Int'l Conf. Data Mining Workshops, ICDMW, Dec. 2010.

[30] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, MapReduce online, in: Proc. USENIX Symp. Networked Systems Design and Implementation, NSDI, Apr. 2010.

[31] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V.B.N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, X. Wang, Nova: continuous Pig/Hadoop workflows, in: Proc. ACM Int'l Conf. Management of Data, SIGMOD, Jun. 2011.
[32] B. Zhu, K. Li, H. Patterson, Avoiding the disk bottleneck in the data domain deduplication file system, in: Proc. USENIX Conf. File and Stroage Technologies, FAST, Feb. 2008.
[33] D. Bhagwat, K. Eshghi, D. Long, M. Lillibridge, Extreme binning: scalable, parallel deduplication for chunk-based file backup, in: Proc. IEEE Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS, Sep. 2009.
[34] W. Xia, H. Jiang, D. Feng, Y. Hua, SiLo: a similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput, in: Proc. USENIX Ann. Tech. Conf., ATC, Jun. 2011.



Daewoo Lee received his B.S. degree in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST) in 2002, and M.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2005. Currently, he is a Ph.D. candidate in Computer Science at KAIST. His research interests include big data processing, cloud computing, and distributed systems.

Jin-Soo Kim received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University, Republic of Korea, in 1991, 1993, and 1999, respectively. He is currently an associate professor at Sungkyunkwan University. Before joining Sungkyunkwan University, he was an associate professor at Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of the research staff, and with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Seungryoul Maeng received his B.S. degree in Electronics Engineering from Seoul National University (SNU), Republic of Korea, in 1977, and M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), in 1979 and 1984, respectively. Since 1984, he has been a faculty member of the Computer Science Department at KAIST. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar. His research interests include micro-architecture, parallel processing, cluster computing, and embedded systems.