i^{2}MapReduce: Incremental MapReduce for Mining Evolving Big Data

Yanfeng Zhang, Shimin Chen, Qiang Wang, and Ge Yu, *Member, IEEE*

Abstract—As new data and updates are constantly arriving, the results of data mining applications become stale and obsolete over time. Incremental processing is a promising approach to refreshing mining results. It utilizes previously saved states to avoid the expense of re-computation from scratch. In this paper, we propose i^{2}MapReduce, a novel incremental processing extension to MapReduce, the most widely used framework for mining big data. Compared with the state-of-the-art work on Incoop, i^{2}MapReduce (i) performs key-value pair level incremental processing rather than task level re-computation, (ii) supports not only one-step computation but also more sophisticated iterative computation, which is widely used in data mining applications, and (iii) incorporates a set of novel techniques to reduce I/O overhead for accessing preserved fine-grain computation states. We evaluate i^{2}MapReduce using a one-step algorithm and four iterative algorithms with diverse computation characteristics. Experimental results on Amazon EC2 show significant performance improvements of i^{2}MapReduce compared to both plain and iterative MapReduce performing re-computation.

Index Terms—Incremental processing, MapReduce, iterative computation, big data

1 Introduction

Today huge amount of digital data is being accumulated in many important areas, including e-commerce, social network, finance, health care, education, and environment. It has become increasingly popular to mine such big data in order to gain insights to help business decisions or to provide better personalized, higher quality services. In recent years, a large number of computing frameworks [1], [2], [3], [4], [5], [6], [7], [8], [9], [10] have been developed for big data analysis. Among these frameworks, MapReduce [1] (with its open-source implementations, such as Hadoop) is the most widely used in production because of its simplicity, generality, and maturity. We focus on improving MapReduce in this paper.

Big data is constantly evolving. As new data and updates are being collected, the input data of a big data mining algorithm will gradually change, and the computed results will become stale and obsolete over time. In many situations, it is desirable to periodically refresh the mining computation in order to keep the mining results up-to-date. For example, the PageRank algorithm [11] computes ranking scores of web pages based on the web graph structure for supporting web search. However, the web graph structure is constantly evolving; Web pages and hyper-links are created, deleted, and updated. As the underlying web graph evolves, the PageRank ranking results gradually become stale, potentially lowering the quality of web search. Therefore, it is desirable to refresh the PageRank computation regularly.

Incremental processing is a promising approach to refreshing mining results. Given the size of the input big data, it is often very expensive to rerun the entire computation from scratch. Incremental processing exploits the fact that the input data of two subsequent computations A and B are similar. Only a very small fraction of the input data has changed. The idea is to save states in computation A, re-use A's states in computation B, and perform re-computation only for states that are affected by the changed input data. In this paper, we investigate the realization of this principle in the context of the MapReduce computing framework.

A number of previous studies (including Percolator [12], CBP [13], and Naiad [14]) have followed this principle and designed new programming models to support incremental processing. Unfortunately, the new programming models (BigTable observers in Percolator, stateful translate operators in CBP, and timely dataflow paradigm in Naiad) are drastically different from MapReduce, requiring programmers to completely re-implement their algorithms.

On the other hand, Incoop [15] extends MapReduce to support incremental processing. However, it has two main limitations. First, Incoop supports only *task-level* incremental processing. That is, it saves and reuses states at the granularity of individual Map and Reduce tasks. Each task typically processes a large number of key-value pairs (kv-pairs). If Incoop detects any data changes in the input of a task, it will rerun the entire task. While this approach easily leverages existing MapReduce features for state savings, it may incur a large amount of redundant computation if only a small fraction of kv-pairs have changed in a task. Second, Incoop supports only *one-step* computation, while important mining algorithms, such as PageRank, require iterative computation. Incoop would treat each iteration as a separate MapReduce job. However, a small number of input data

• *Y. Zhang is with the Computing Center, Northeastern University, Shenyang, China 110819. E-mail: zhangyf@cc.neu.edu.cn.*
• *S. Chen is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences. E-mail: chensm@ict.ac.cn.*
• *Q. Wang and G. Yu are with Northeastern University, Shenyang, China 110819. E-mail: 403268229@qq.com, yuge@mail.neu.edu.cn.*

Manuscript received 3 June 2014; revised 17 Oct. 2014; accepted 1 Jan. 2015. Date of publication 1 Feb. 2015; date of current version 1 June 2015.
Recommended for acceptance by S. Babu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2015.2397438

changes may gradually propagate to affect a large portion of intermediate states after a number of iterations, resulting in expensive global re-computation afterwards.

We propose i^{2}MapReduce, an extension to MapReduce that supports *fine-grain* incremental processing for both *one-step* and *iterative* computation. Compared to previous solutions, i^{2}MapReduce incorporates the following three novel features:

- *Fine-grain incremental processing using MRBG-Store*. Unlike Incoop, i^{2}MapReduce supports kv-pair level fine-grain incremental processing in order to minimize the amount of re-computation as much as possible. We model the kv-pair level data flow and data dependence in a MapReduce computation as a bipartite graph, called MRBGraph. A MRBG-Store is designed to preserve the fine-grain states in the MRBGraph and support efficient queries to retrieve fine-grain states for incremental processing. (cf. Section 3)
- *General-purpose iterative computation with modest extension to MapReduce API*. Our previous work proposed iMapReduce [10] to efficiently support iterative computation on the MapReduce platform. However, it targets types of iterative computation where there is a one-to-one/all-to-one correspondence from Reduce output to Map input. In comparison, our current proposal provides general-purpose support, including not only one-to-one, but also one-to-many, many-to-one, and many-to-many correspondence. We enhance the Map API to allow users to easily express loop-invariant structure data, and we propose a Project API function to express the correspondence from Reduce to Map. While users need to slightly modify their algorithms in order to take full advantage of i^{2}MapReduce, such modification is modest compared to the effort to re-implement algorithms on a completely different programming paradigm, such as in Percolator [12], CBP [13], and Naiad [14]. (cf. Section 4)
- *Incremental processing for iterative computation*. Incremental iterative processing is substantially more challenging than incremental one-step processing because even a small number of updates may propagate to affect a large portion of intermediate states after a number of iterations. To address this problem, we propose to reuse the converged state from the previous computation and employ a change propagation control (CPC) mechanism. We also enhance the MRBG-Store to better support the access patterns in incremental iterative processing. To our knowledge, i^{2}MapReduce is the *first MapReduce-based solution that efficiently supports incremental iterative computation*. (cf. Section 5)

We implemented i^{2}MapReduce by modifying Hadoop-1.0.3. We evaluate i^{2}MapReduce using a one-step algorithm (A-Priori) and four iterative algorithms (PageRank, SSSP, Kmeans, GIM-V) with diverse computation characteristics. Experimental results on Amazon EC2 show significant performance improvements of i^{2}MapReduce compared to both plain and iterative MapReduce performing re-computation. For example, for the iterative PageRank computation with 10 percent data changed, i^{2}MapReduce improves the run time of re-computation on plain MapReduce by an eight fold speedup. (cf. Section 6)

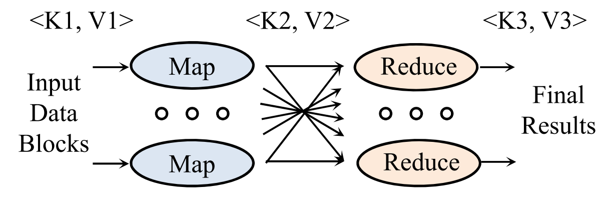


Fig. 1. MapReduce computation.

2 MAPREDUCE BACKGROUND

A MapReduce program is composed of a Map function and a Reduce function [1], as shown in Fig. 1. Their APIs are as follows:

$$\texttt{map}(K1, V1) \rightarrow [\langle K2, V2\rangle]$$

$$\texttt{reduce}(K2, \{V2\}) \rightarrow [\langle K3, V3\rangle].$$

The Map function takes a kv-pair $\langle K1, V1\rangle$ as input and computes zero or more intermediate kv-pairs $\langle K2, V2\rangle$s. Then all $\langle K2, V2\rangle$s are grouped by $K2$. The Reduce function takes a $K2$ and a list of $\{V2\}$ as input and computes the final output kv-pairs $\langle K3, V3\rangle$s.

A MapReduce system (e.g., Apache Hadoop) usually reads the input data of the MapReduce computation from and writes the final results to a distributed file system (e.g., HDFS), which divides a file into equal-sized (e.g., 64 MB) blocks and stores the blocks across a cluster of machines. For a MapReduce program, the MapReduce system runs a JobTracker process on a master node to monitor the job progress, and a set of TaskTracker processes on worker nodes to perform the actual Map and Reduce tasks.

The JobTracker starts a Map task per data block, and typically assigns it to the TaskTracker on the machine that holds the corresponding data block in order to minimize communication overhead. Each Map task calls the Map function for every input $\langle K1, V1\rangle$, and stores the intermediate kv-pairs $\langle K2, V2\rangle$s on local disks. Intermediate results are shuffled to Reduce tasks according to a partition function (e.g., a hash function) on $K2$. After a Reduce task obtains and merges intermediate results from all Map Tasks, it invokes the Reduce function on each $\langle K2, \{V2\}\rangle$ to generate the final output kv-pairs $\langle K3, V3\rangle$s.

3 FINE-GRAIN INCREMENTAL PROCESSING FOR ONE-STEP COMPUTATION

We begin by describing the basic idea of fine-grain incremental processing in Section 3.1. In Sections 3.2 and 3.3, we present the main design, including the MRBGraph abstraction and the incremental processing engine. Then in Sections 3.4 and 3.5, we delve into two aspects of the design, i.e., the mechanism that preserves the fine-grain states, and the handling of a special but popular case where the Reduce function performs accumulation operations.

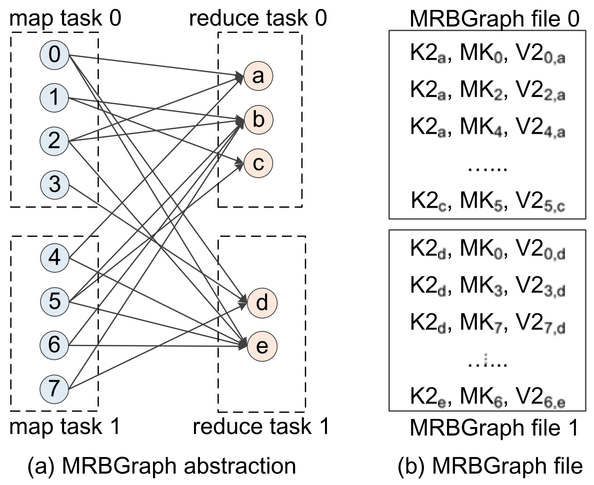


Fig. 2. MRBGraph.

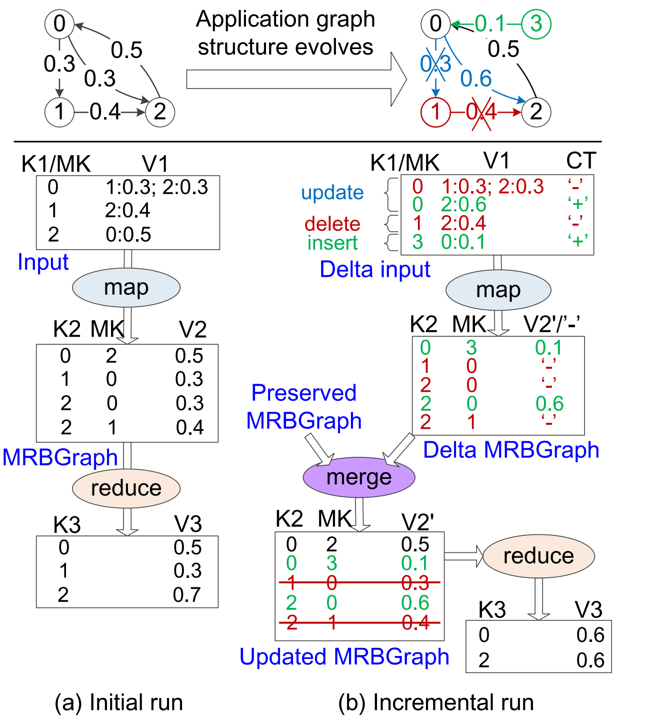


Fig. 3. Incremental processing for an application that computes the sum of in-edge weights for each vertex.

3.1 Basic Idea

Consider two MapReduce jobs A and A' performing the same computation on input data set D and D', respectively. $D' = D + \Delta D$, where ΔD consists of the inserted and deleted input $\langle K1, V1\rangle$s[1]. An update can be represented as a deletion followed by an insertion. Our goal is to re-compute only the Map and Reduce function call instances that are affected by ΔD.

Incremental computation for Map is straightforward. We simply invoke the Map function for the inserted or deleted $\langle K1, V1\rangle$s. Since the other input kv-pairs are not changed, their Map computation would remain the same. We now have computed the delta intermediate values, denoted ΔM, including inserted and deleted $\langle K2, V2\rangle$s.

To perform incremental Reduce computation, we need to save the fine-grain states of job A, denoted M, which includes $\langle K2, \{V2\}\rangle$s. We will re-compute the Reduce function for each $K2$ in ΔM. The other $K2$ in M does not see any changed intermediate values and therefore would generate the same final result. For a $K2$ in ΔM, typically only a subset of the list of $V2$ have changed. Here, we retrieve the saved $\langle K2, \{V2\}\rangle$ from M, and apply the inserted and/or deleted values from ΔM to obtain an updated Reduce input. We then re-compute the Reduce function on this input to generate the changed final results $\langle K3, V3\rangle$s.

It is easy to see that results generated from this incremental computation are logically the same as the results from completely re-computing A'.

3.2 MRBGraph Abstraction

We use a MRBGraph (Map Reduce Bipartite Graph) abstraction to model the data flow in MapReduce, as shown in Fig. 2a. Each vertex in the Map task represents an individual Map function call instance on a pair of $\langle K1, V1\rangle$. Each vertex in the Reduce task represents an individual Reduce function call instance on a group of $\langle K2, \{V2\}\rangle$. An edge from a Map instance to a Reduce instance means that the Map instance generates a $\langle K2, V2\rangle$ that is shuffled to become part of the input to the Reduce instance. For example, the input of Reduce instance a comes from Map instance 0, 2, and 4.

MRBGraph edges are the fine-grain states M that we would like to preserve for incremental processing. An edge contains three pieces of information: (i) the source Map instance, (ii) the destination Reduce instance (as identified by $K2$), and (iii) the edge value (i.e., $V2$). Since Map input key $K1$ may not be unique, i^{2}MapReduce generates a globally unique Map key MK for each Map instance. Therefore, i^{2}MapReduce will preserve ($K2$, MK, $V2$) for each MRBGraph edge.

3.3 Fine-Grain Incremental Processing Engine

Fig. 3 illustrates the fine-grain incremental processing engine with an example application, which computes the sum of in-edge weights for each vertex in a graph. As shown at the top of Fig. 3, the input data, i.e., the graph structure, evolves over time. In the following, we describe how the engine performs incremental processing to refresh the analysis results.

Initial run and MRBGraph preserving. The initial run performs a normal MapReduce job, as shown in Fig. 3a. The Map input is the adjacency matrix of the graph. Every record corresponds to a vertex in the graph. $K1$ is vertex id i, and $V1$ contains "j_1:w_{i,j_1}; j_2:w_{i,j_2}; ..." where j is a destination vertex and $w_{i,j}$ is the weight of the out-edge (i, j). Given such a record, the Map function outputs intermediate kv-pair $\langle j, w_{i,j}\rangle$ for every j. The shuffling phase groups the edge weights by the destination vertex. Then the Reduce function computes for a vertex j the sum of all its in-edge weights as $\sum_i w_{i,j}$.

For incremental processing, we preserve the fine-grain MRBGraph edge states. A question arises: shall the states be

1. We assume that new data or new updates are captured via incremental data acquisition or incremental crawling [16], [17]. Incremental data acquisition can significantly save the resources for data collection; it does not re-capture the whole data set but only capture the revisions since the last time that data was captured.

preserved at the Map side or at the Reduce side? We choose the latter because during incremental processing original intermediate values can be obtained at the Reduce side without any shuffling overhead. The engine transfers the globally unique MK along with $\langle K2, V2\rangle$ during the shuffle phase. Then it saves the states $(K2, MK, V2)$ in a MRBGraph file at every Reduce task, as shown in Fig. 2b.

Delta input. i^{2}MapReduce expects delta input data that contains the newly inserted, deleted, or modified kv-pairs as the input to incremental processing. Note that identifying the data changes is beyond the scope of this paper; Many incremental data acquisition or incremental crawling techniques have been developed to improve data collection performance [16], [17].

Fig. 3b shows the delta input for the updated application graph. A '+' symbol indicates a newly inserted kv-pair, while a '−' symbol indicates a deleted kv-pair. An update is represented as a deletion followed by an insertion. For example, the deletion of vertex 1 and its edge are reflected as $\langle 1, 2{:}0.4, `-'\rangle$. The insertion of vertex 3 and its edge leads to $\langle 3, 0{:}0.1, `+'\rangle$. The modification of the vertex 0's edges are reflected by a deletion of the old record $\langle 0, 1{:}0.3;2{:}0.3, `-'\rangle$ and an insertion of a new record $\langle 0, 2{:}0.6, `-'\rangle$.

Incremental map computation to obtain the delta MRBGraph. The engine invokes the Map function for every record in the delta input. For an insertion with '+', its intermediate results $\langle K2, MK, V2'\rangle$s represent newly inserted edges in the MRBGraph. For a deletion with '−', its intermediate results indicate that the corresponding edges have been removed from the MRBGraph. The engine replaces the $V2'$s of the deleted MRBGraph edges with '−'. During the MapReduce shuffle phase, the intermediate $\langle K2, MK, V2'\rangle$s and $\langle K2, MK, `-'\rangle$s with the same $K2$ will be grouped together. The delta MRBGraph will contain only the changes to the MRBGraph and sorted by the $K2$ order.

Incremental reduce computation. The engine merges the delta MRBGraph and the preserved MRBGraph to obtain the updated MRBGraph using the algorithm in Section 3.4. For each $\langle K2, MK, `-'\rangle$, the engine deletes the corresponding saved edge state. For each $\langle K2, MK, V2'\rangle$, the engine first checks duplicates, and inserts the new edge if no duplicate exists, or else updates the old edge if duplicate exists. (Note that ($K2$, MK) uniquely identifies a MRBGraph edge.) Since an update in the Map input is represented as a deletion and an insertion, any modification to the intermediate edge state (e.g., $\langle 2, 0, *\rangle$ in the example) consists of a deletion (e.g., $\langle 2, 0, `-'\rangle$) followed by an insertion (e.g., $\langle 2, 0, 0.6\rangle$). For each affected $K2$, the merged list of $V2$ will be used as input to invoke the Reduce function to generate the updated final results.

3.4 MRBG-Store

The MRBG-Store supports the preservation and retrieval of fine-grain MRBGraph states for incremental processing. We see two main requirements on the MRBG-Store. First, the MRBG-Store must incrementally store the evolving MRBGraph. Consider a sequence of jobs that incrementally refresh the results of a big data mining algorithm. As input data evolves, the intermediate states in the MRBGraph will also evolve. It would be wasteful to store the entire MRBGraph of each subsequent job. Instead, we would like to obtain and store only the updated part of the MRBGraph. Second, the MRGB-Store must support efficient retrieval of preserved states of given Reduce instances. For incremental Reduce computation, i^{2}MapReduce re-computes the Reduce instance associated with each changed MRBGraph edge, as described in Section 3.3. For a changed edge, it queries the MRGB-Store to retrieve the preserved states of the in-edges of the associated $K2$, and merge the preserved states with the newly computed edge changes.

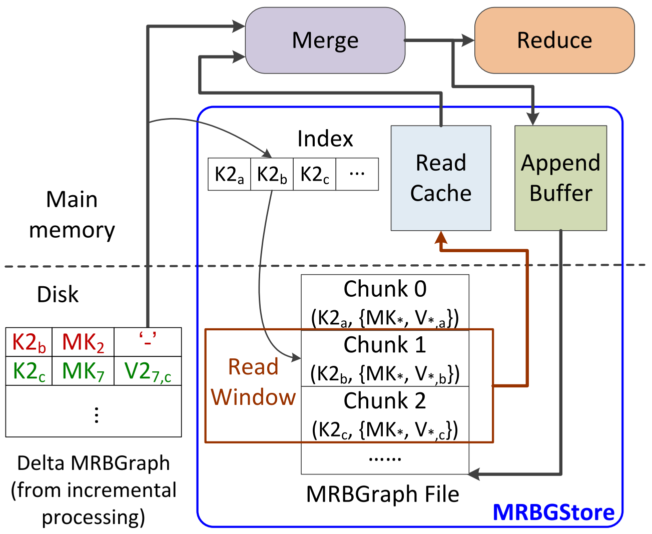


Fig. 4. Structure of MRBG-store.

Fig. 4 depicts the structure of the MRBG-Store. We describe how the components of the MRBG-Store work together to achieve the above two requirements.

Fine-grain state retrieval and merging. A MRBGraph file stores fine-grain intermediate states for a Reduce task, as illustrated previously in Fig. 2b. In Fig. 4, we see that the $\langle K2, MK, V2\rangle$s with the same $K2$ are stored contiguously as a *chunk*. Since a chunk corresponds to the input to a Reduce instance, our design treats chunk as the basic unit, and always reads, writes, and operates on entire chunks.

The contents of a delta MRBGraph file are shown on the bottom left of Fig. 4. Every record represents a change in the original (last preserved) MRBGraph. There are two kinds of records. An edge insertion record (in green color) contains a valid $V2$ value; an edge deletion record (in red color) contains a null value (as marked by '−').

The merging of the delta MRBGraph with the MRBGraph file in the MRBG-Store is essentially a join operation using $K2$ as the join key. Since the size of the delta MRBGraph is typically much smaller than the MRBGraph file, it is wasteful to read the entire MRBGraph file. Therefore, we construct an index for selective access to the MRBGraph file: Given a $K2$, the index returns the chunk position in the MRBGraph file. As only point lookup is required, we employ a hash-based implementation for the index. The index is stored in an index file and is preloaded into memory before Reduce computation. We apply the index nested loop join for the merging operation.

Can we further improve the join performance? We observe that the MapReduce shuffling phase will sort the intermediate keys. As seen in Section 3.3, the records in both the delta MRBGraph and the MRBGraph file are in the order generated by the shuffling phase. That is, the two files are sorted in $K2$ order. Therefore, we introduce a read cache and a dynamic read window technique for further optimization. Fig. 4 shows the idea. Given a sequence of $K2$s, there are two ways to read the corresponding chunks: (i) performing an individual I/O operation for each chunk; or (ii) performing a large I/O that covers all the required chunks. The former may lead to frequent disk seeks, while the latter may result in reading a lot of useless data. Fortunately, we know the list of sorted $K2$s to be queried. Using the index, we obtain their chunk positions. We can estimate the costs of using a large I/O versus a number of individual I/Os, and intelligently determine the read window size w based on the cost estimation.

Algorithm 1 shows the query algorithm to retrieve the the chunk k given a query key k and the list of queried keys $L = \{L_1, L_2, \ldots\}$. If the chunk k does not reside in the read cache (line 1), it will compute the read window size w by a heuristic, and read w bytes into the read cache. The loop (line 4–8) probes the gap between two consecutive queried chunks (chunk L_i and chunk L_{i+1}). The gap size indicates the wasted read effort. If the gap is less than a threshold T ($T = 100$ KB by default), we consider that the benefit of large I/O can compensate for the wasted read effort, and enlarge the window to cover chunk L_{i+1}. In this way, the algorithm finds the read window size w by balancing the cost of a large I/O versus a number of individual I/Os. It also ensures that the read window size does not exceed the read cache. Then the algorithm read the next w bytes into the read cache (line 9) and retrieves the requested chunk k from the read cache (line 11).

Algorithm 1. Query Algorithm in MRBG-Store

```
Input queried key: $k$; the list of queried keys: $L$
Output chunk $k$
 1: if ! $read\_cache$.contains($k$) then
 2:     $gap \leftarrow 0, w \leftarrow 0$
 3:     $i \leftarrow k$'s index in $L$          // That is, $L_i = k$
 4:     while $gap < T$ and $w + gap + length(L_i) < read\_cache.size$ do
 5:         $w \leftarrow w + gap + length(L_i)$
 6:         $gap \leftarrow pos(L_{i+1}) - pos(L_i) - length(L_i)$
 7:         $i \leftarrow i + 1$
 8:     end while
 9:     starting from $pos(k)$, read $w$ bytes into $read\_cache$
10: end if
11: return $read\_cache$.get_chunk($k$)
```

Incremental storage of MRBgraph changes. As shown in Fig. 4, the outputs of the merge operation, which are the up-to-date MRBGraph states (chunks), are used to invoke the Reduce function. In addition, the outputs are also buffered in an append buffer in memory. When the append buffer is full, the MRBG-Store performs sequential I/Os to append the contents of the buffer to the end of the MRBGraph file. When the merge operation completes, the MRBG-Store flushes the append buffer, and updates the index to reflect the new file positions for the updated chunks. Note that obsolete chunks are NOT immediately updated in the file (or removed from the file) for I/O efficiency. The MRBGraph file is reconstructed off-line when the worker is idle. In this way, the MRBG-Store efficiently supports incremental storage of MRBGraph Changes.

As a result of the incremental storage, the MRBGraph file may contain multiple segments of sorted chunks, each resulting from a merge operation. This situation frequently appears in iterative incremental computation, for which we enhance the above query algorithm with a multi-window technique to efficiently process the multiple segments. We defer the in-depth discussion to Section 5.

3.5 Optimization for Special Accumulator Reduce

We study a special case that appears frequently in applications and is amenable to further optimization. Specifically, the Reduce function is an accumulative operation '$\oplus$':

$$f(\{V2_0, V2_1, \ldots, V2_k\}) = V2_0 \oplus V2_1 \oplus \cdots \oplus V2_k,$$

which satisfies the distributive property:

$$f(D \cup \Delta D) = f(D) \oplus f(\Delta D),$$

and the incremental data set ΔD contains only insertions without deletions or updates. This property allows us to process the two data set D and ΔD separately and then to simply combine the results by the '$\oplus$' operation to obtain the full result. We call this kind of Reduce function *accumulator Reduce*. For this special case, it is not necessary to preserve the MRBGraph. The engine will optimize the special case by only preserving the Reduce output kv-pairs $\langle K3, V3 \rangle$. Then it simply invokes the accumulator Reduce to accumulate changes to the result kv-pairs.

Many MapReduce algorithms employ accumulator Reduce. A well-known example is WordCount. The Reduce function of WordCount computes the count of word appearances using an integer sum operation, which satisfies the above property. Other common operations that directly satisfy the distributive property include maximum and minimum. Moreover, some operations can be easily modified to satisfy the requirement of accumulator Reduce. For example, average is computed as dividing sum by count. While it is not possible to combine two averages into a single average, we can modify the implementation to allow/produce a partial sum and a partial count in the function input and the output. Then the implementation can accumulate partial sums and partial counts in order to compute the average of the full data set.

To use this feature, a programmer should declare the accumulative operation '$\oplus$' using a new interface `AccumulatorReducer` in the MapReduce driver program (see Table 2).

4 General-Purpose Support for Iterative Computation

We first analyze several representative iterative algorithms in Section 4.1. Based on this analysis, we propose a general-purpose MapReduce model for iterative computation in Section 4.2, and describe how to efficiently support this model in Section 4.3.

TABLE 1
Structure and State kv-Pairs in Representative Iterative Algorithms

Algorithm	Structure Key (SK)	Structure Value (SV)	State Key (DK)	State Value (DV)	SK $\leftrightarrow$ DK
PageRank	vertex id i	out-neighbor set N_i	vertex id i	rank score R_i	one-to-one
Kmeans	point id pid	point value $pval$	unique key 1	centroids $\{\langle cid, cval\rangle\}$	all-to-one
GIM-V	matrix block id (i, j)	matrix block $m_{i,j}$	vector block id j	vector block v_j	many-to-one

4.1 Analyzing Iterative Computation

PageRank. PageRank [11] is a well-known iterative graph algorithm for ranking web pages. It computes a ranking score for each vertex in a graph. After initializing all ranking scores, the computation performs a MapReduce job per iteration, as shown in Algorithm 2. i and j are vertex ids, N_i is the set of out-neighbor vertices of i, R_i is i's ranking score that is updated iteratively. '|' means concatenation. All R_i's are initialized to one.[2] The Map instance on vertex i sends value $R_{i,j} = R_i/|N_i|$ to all its out-neighbors j, where $|N_i|$ is the number of i's out-neighbors. The Reduce instance on vertex j updates R_j by summing the $R_{i,j}$ received from all its in-neighbors i, and applying a damping factor d.

Algorithm 2. PageRank in MapReduce

```
Map Phase input: $< i, N_i|R_i >$
1: output $< i, N_i >$
2: for all $j$ in $N_i$ do
3:     $R_{i,j} = \frac{R_i}{|N_i|}$
4:     output $< j, R_{i,j} >$
5: end for
Reduce Phase input: $< j, \{R_{i,j}, N_j\} >$
6: $R_j = d\sum_i R_{i,j} + (1-d)$
7: output $< j, N_j|R_j >$
```

Kmeans. Kmeans [18] is a commonly used clustering algorithm that partitions points into k clusters. We denote the ID of a point as pid, and its feature values $pval$. The computation starts with selecting k random points as cluster centroids set $\{cid, cval\}$. As shown in Algorithm 3, in each iteration, the Map instance on a point pid assigns the point to the nearest centroid. The Reduce instance on a centroid cid updates the centroid by averaging the values of all assigned points $\{pval\}$.

Algorithm 3. Kmeans in MapReduce

```
Map Phase input: $< pid, pval|\{cid, cval\} >$
1: $cid \leftarrow$ find the nearest centroid of $pval$ in $\{cid, cval\}$
2: output $< cid, pval >$
Reduce Phase input: $< cid, \{pval\} >$
3: $cval \leftarrow$ compute the average of $\{pval\}$
4: output $< cid, cval >$
```

GIM-V. Generalized Iterated Matrix-Vector multiplication (GIM-V) [19] is an abstraction of many iterative graph mining operations (e.g., PageRank, spectral clustering, diameter estimation, connected components). These graph mining algorithms can be generally represented by operating on an $n \times n$ matrix M and a vector v of size n. Suppose both the matrix and the vector are divided into sub-blocks. Let $m_{i,j}$ denote the (i, j)th block of M and v_j denote the jth block of v. The computation steps are similar to those of the matrix-vector multiplication and can be abstracted into three operations: (1) $mv_{i,j}$ = `combine2`$(m_{i,j}, v_j)$; (2) v'_i = `combineAll`$_i(\{mv_{i,j}\})$; and (3) v_i = `assign`(v_i, v'_i). We can compare `combine2` to the multiplication between $m_{i,j}$ and v_j, and compare `combineAll` to the sum of $mv_{i,j}$ for row i. Algorithm 4 shows the MapReduce implementation with two jobs for each iteration. The first job assigns vector block v_j to multiple matrix blocks $m_{i,j}$ $(\forall i)$ and performs `combine2`$(m_{i,j}, v_j)$ to obtain $mv_{i,j}$. The second job groups the $mv_{i,j}$ and v_i on the same i, performs the `combineAll` `(`$\{mv_{i,j}\}$`)` operation, and updates v_i using `assign`(v_i, v'_i).

Algorithm 4. GIM-V in MapReduce

```
Map Phase1 input: $< (i, j), m_{i,j} >$ or $< j, v_j >$
 1: if kv-pair is $< (i, j), m_{i,j} >$ then
 2:     output $< (i, j), m_{i,j} >$
 3: else if kv-pair is $< j, v_j >$ then
 4:     for all $i$ blocks in $j$'s row do
 5:         output $< (i, j), v_j >$
 6:     end for
 7: end if
Reduce Phase 1 input: $< (i, j), \{m_{i,j}, v_j\} >$
 8: $mv_{i,j}$ = combine2($m_{i,j}$, $v_j$)
 9: output $< i, mv_{i,j} >$, $< j, v_j >$
Map Phase 2: output all inputs
Reduce Phase 2 input: $< i, \{mv_{i,j}, v_i\} >$
10: $v'_i \leftarrow$ combineAll($\{mv_{i,j}\}$)
11: $v_i \leftarrow$ assign($v_i, v'_i$)
12: output $< i, v_i >$
```

Two kinds of data sets in iterative Algorithms. From the above examples, we see that iterative algorithms usually involve two kinds of data sets: (i) loop-invariant *structure data*, and (ii) loop-variant *state data*. Structure data often reflects the problem structure and is read-only during computation. In contrast, state data is the target results being updated in each iteration by the algorithm. Structure (state) data can be represented by a set of *structure (state) kv-pairs*. Table 1 displays the structure and state kv-pairs of the three example algorithms.

Dependency types between state and structure data. There are various types of dependencies between state and structure data, as listed in Table 1. PageRank sees one-to-one dependency: every vertex i is associated with both an out-neighbor set N_i and a ranking score R_i. In Kmeans, the Map

2. The computed PageRank scores will be $|N|$ times larger, where $|N|$ is the number of vertices in the graph.

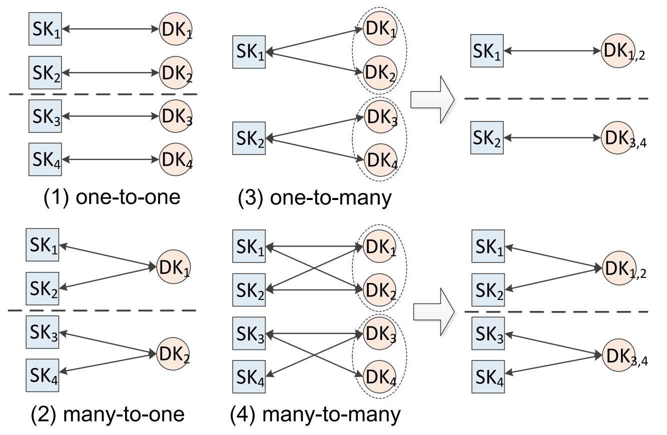


Fig. 5. Dependency types between structure and state kv-pairs. (3)/(4) can be converted into (1)/(2).

instance of every point requires the set of all centroids, showing an all-to-one dependency. In GIM-V, multiple matrix blocks $\forall j, m_{i,j}$ are combined to compute the ith vector block v_i, thus the dependency is many-to-one.

Generally speaking, there are four types of dependencies between structure kv-pairs and state kv-pairs as shown in Fig. 5: (1) one-to-one, (2) many-to-one, (3) one-to-many, (4) many-to-many. All-to-one (one-to-all) is a special case of many-to-one (one-to-many). PageRank is an example of (1). Kmeans and GIM-V are examples of (2). We have not encountered applications with (3) or (4) dependencies. (3) and (4) are listed only for completeness of discussion.

In fact, for (3) one-to-many case and (4) many-to-many case, it is possible to redefine the state key to convert them into (1) one-to-one and (2) many-to-one dependencies, respectively, as show in the right part of Fig. 5. The idea is to re-organize the MapReduce computation in an application or to define a custom partition function for shuffling so that the state kv-pairs (e.g, DK_1 and DK_2 in the figure) that Map to the same structure kv-pair (e.g., SK_1 in the figure) are always processed in the same task. Then we can assign a key (e.g., $DK_{1,2}$) to each group of state kv-pairs, and consider each group as a single state kv-pair. Given this transformation, we need to focus on only (1) one-to-one and (2) many-to-one cases. Consequently, each structure kv-pair is interdependent with **ONLY** a single state kv-pair. This is an important property that we leverage in our design of i^{2}MapReduce.

4.2 General-Purpose Iterative MapReduce Model

A number of recent efforts have been targeted at improving iterative processing on MapReduce, including Twister [9], HaLoop [8], and iMapReduce [10]. In general, the improvements focus on two aspects:

- *Reducing job startup costs.* In vanilla MapReduce, every algorithm iteration runs one or several MapReduce jobs. Note that Hadoop may take over 20 seconds to start a job with 10–100 tasks. If the computation of each iteration is relatively simple, job startup costs may consist of an overly large fraction of the run time. The solution is to modify MapReduce to reuse the same jobs across iterations, and kill them only when the computation completes.
- *Caching structure data.* Structure data is immutable during computation. It is also much larger than state data in many applications (e.g., PageRank, Kmeans, and GIM-V). Therefore, it is wasteful to transfer structure data over and over again in every iteration. An optimization is to cache structure data in local file systems to avoid the cost of network communication and reading from HDFS.

For the first aspect, we modify Hadoop to allow jobs to stay alive across multiple iterations.

For the second aspect, however, a design must separate structure data from state data, and consider how to match interdependent structure and state data in the computation. HaLoop [8] uses an extra MapReduce job to match structure and state data in each iteration. We would like to avoid such heavy-weight solution. iMapReduce [10] creates the same number of Map and Reduce tasks, and connects every Reduce task to a Map task with a local connection to transfer the state data output from a Reduce task to the corresponding Map task. However, this approach assumes one-to-one dependency for join operation. Thus, it cannot support Kmeans or GIM-V.

In the following, we propose a design that generalizes previous solutions to efficiently support various dependency types.

Separating structure and state data in map API. We enhance the Map function API to explicitly express structure vs. state kv-pairs in i^{2}MapReduce:

$$\texttt{map}(SK, SV, DK, DV) \rightarrow [\langle K2, V2\rangle].$$

The interdependent structure kv-pair $\langle SK, SV\rangle$ and state kv-pair $\langle DK, DV\rangle$ are conjointly used in the Map function. A Map function outputs intermediate kv-pairs $\langle K2, V2\rangle$s. The Reduce interface is kept the same as before. A Reduce function combines the intermediate kv-pairs $\langle K2, \{V2\}\rangle$s and outputs $\langle K3, V3\rangle$:

$$\texttt{reduce}(K2, \{V2\}) \rightarrow \langle K3, V3\rangle.$$

Specifying dependency with project. We propose a new API function, *Project*. It specifies the interdependent state key of a structure key:

$$\texttt{project}(SK) \rightarrow DK.$$

Note that each structure kv-pair is interdependent with a single state kv-pair. Therefore, Project returns a single value DK for each input SK.

Iterative model. Fig. 6 shows our iterative model. By analyzing the three representative applications, we find that the input of an iteration contains both structure and state data, while the output is only the state data. A large number of iterative algorithms (e.g., PageRank and Kmeans) employs a single MapReduce job in an iteration. Their computation can be illustrated using the simplified model as shown in Fig. 6b. In general, one or more MapReduce jobs may be used to update the state kv-pairs $\langle DK, DV\rangle$, as shown in Fig. 6a. Once the updated $\langle DK, DV\rangle$s are obtained, they are matched to the interdependent structure kv-pairs $\langle SK, SV\rangle$s with the Project function for next iteration. In this way, a kv-pair transformation loop is built. We

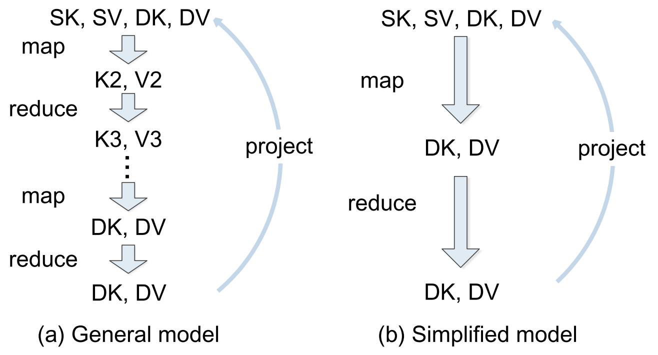


Fig. 6. Iterative model of i^{2}MapReduce.

call the first Map phase in an iteration the *prime Map* and the last Reduce phase in an iteration as the *prime Reduce*.

4.3 Supporting Diverse Dependencies Between Structure and State Data

Dependency-aware data partitioning. To support parallel processing in MapReduce, we need to partition the data. Note that both structure and state kv-pairs are required to invoke the Map function. Therefore, it is important to assign the interdependent structure kv-pair and state kv-pair to the same partition so as to avoid unnecessary network transfer overhead. Many existing systems such as Spark [2] and Stratosphere [7] have applied this optimization. In i^{2}MapReduce, we design the following partition function (1) for state and (2) for structure kv-pairs:

$$partition\_id = \texttt{hash}(DK, n) \tag{1}$$

$$partition\_id = \texttt{hash}(\texttt{project}(SK), n), \tag{2}$$

where n is the desired number of Map tasks. Both functions employ the same hash function. Since Project returns the interdependent DK for a given SK, the interdependent $\langle SK, SV\rangle$s and $\langle DK, DV\rangle$s will be assigned to the same partition. i^{2}MapReduce partitions the structure data and state data as the preprocessing step before an iterative job.

Invoking prime map. i^{2}MapReduce launches a prime Map task per data partition. The structure and state kv-pairs assigned to a partition are stored in two files: (i) a structure file containing $\langle SK, SV\rangle$s and (ii) a state file containing $\langle DK, DV\rangle$s. The two files are provided as the input to the prime Map task. The state file is sorted in the order of DK, while the structure file is sorted in the order of `project` (SK). That is, the interdependent SKs and DKs are sorted in the same order. Therefore, i^{2}MapReduce can sequentially read and match all the interdependent structure/state kv-pairs through *a single pass* of the two files, while invoking the Map function for each matching pair.

Task Scheduling: Co-locating interdependent prime reduce and prime map. As shown in Fig. 6, the prime Reduce computes the updated state kv-pairs. For the next iteration, i^{2}MapReduce must transfer the updated state kv-pairs to their corresponding prime Map task, which caches their dependent structure kv-pairs in its local file system.

The overhead of the backward transfer can be fully removed if the number of state kv-pairs in the application is greater than or equal to n, the number of Map tasks (e.g., PageRank and GIM-V). The idea is to create n Reduce tasks, assign Reduce task i to co-locate with Map task i on the same machine node, and make sure that Reduce task i produces and only produces the state kv-pairs in partition i. The latter can be achieved by employing the hash function of the partition functions (1) and (2) as the shuffle function immediately before the prime Reduce phase. The Reduce output can be stored into an updated state file without any network cost. Interestingly, the state file is automatically sorted in DK order thanks to MapReduce's shuffle implementation. In this way, i^{2}MapReduce will be able to process the prime Map task of the next iteration.

Supporting smaller number of state kv-pairs. In some applications, the number of state keys is smaller than n. Kmeans is an extreme case with only a single state kv-pair. In these applications, the total size of the state data is typically quite small. Therefore, the backward transfer overhead is low. Under such situation, i^{2}MapReduce does not apply the above partition functions. Instead, it partitions the structure kv-pairs using MapReduce's default approach, while replicating the state data to each partition.

5 Incremental Iterative Processing

In this section, we present incremental processing techniques for iterative computation. Note that it is not sufficient to simply combine the above solutions for incremental one-step processing (in Section 3) and iterative computation (in Section 4). In the following, we discuss three aspects that we address in order to achieve an effective design.

5.1 Running an Incremental Iterative Job

Consider a sequence of jobs $A_1, \ldots A_i, \ldots$ that *incrementally* refresh the results of an *iterative* algorithm. Incoming new data and updates change the problem structure (e.g., edge insertions or deletions in the web graph in PageRank, new points in Kmeans, updated matrix data in GIM-V). Therefore, structure data evolves across subsequent jobs. Inside a job, however, structure data stays constant, but state data is iteratively updated and converges to a fixed point. The two types of data must be handled differently when starting an incremental iterative job:

- *Delta structure data.* We partition the new data and updates based on Equation (2), and generate a delta structure input file per partition.
- *Previously converged state data.* Which state shall we use to start the computation? For job A_i, we choose to use the converged state data D_{i-1} from job A_{i-1}, rather than the random initial state D_0 (e.g., random centroids in Kmeans) for two reasons. First, D_{i-1} is likely to be very similar to the converged state D_i to be computed by A_i because there are often only slight changes in the input data. Hence, A_i may converge to D_i much faster from D_{i-1} than from D_0. Second, only the states in the last iteration of A_{i-1} need to be saved. If D_0 were used, the system would have to save the states of every iteration in A_{i-1} in order to incrementally process the corresponding iteration in A_i. Thus, our choice can significantly speed up convergence, and reduce the time and space overhead for saving states.

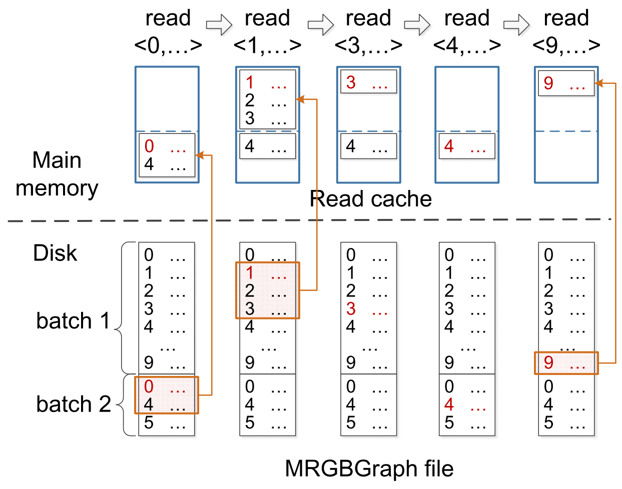


Fig. 7. An example of reading a sequence of chunks with key 0,1,3,4,9, . . . by using multi-dynamic-window.

To run an incremental iterative job A_i, i^{2}MapReduce treats each iteration as an incremental one-step job as shown previously in Fig. 3. In the first iteration, the delta input is the delta structure data. The preserved MRBGraph reflects the last iteration in job A_{i-1}. Only the Map and Reduce instances that are affected by the delta input are re-computed. The output of the prime Reduce is the delta state data. Apart from the computation, i^{2}MapReduce refreshes the MRBGraph with the newly computed intermediate states. We denote the resulting updated MRBGraph as MRBGraph$_1$.

In the jth iteration ($j \geq 2$), the structure data remains the same as in the $(j-1)$th iteration, but the loop-variant state data have been updated. Therefore, the delta input is now the delta state data. Using the preserved MRBGraph$_{j-1}$, i^{2}MapReduce re-computes only the Map and Reduce instances that are affected by the input change. It preserves the newly computed intermediate states in MRBGraph$_j$. It computes a new delta state data for the next iteration.

The job completes when the state data converges or certain predefined criteria are met. At this moment, i^{2}MapReduce saves the converged state data to prepare for the next job A_{i+1}.

5.2 Extending MRBG-Store for Multiple Iterations

As described previously in Section 3.4, MRBG-Store appends newly computed chunks to the end of the MRBGraph file and updates the chunk index to reflect the new positions. Obsolete chunks are removed offline when the worker machine is idle. In an incremental iterative job, every iteration will generate newly computed chunks, which are sorted due to the MapReduce shuffling phase. Consequently, the MRBGraph file will consist of multiple batches of sorted chunks, corresponding to a series of iterations. If a chunk exists in multiple batches, a retrieval request returns the latest version of the chunk (as pointed to by the chunk index). In the following, we extend the query algorithm (Algorithm 1) to handle multiple batches of sorted chunks.

We propose a *multi-dynamic-window* technique. Multiple dynamic windows correspond to multiple batches (iterations). Fig. 7 illustrates how the multi-dynamic-window technique works via an example. In this example, the MRBGraph file contains two batches of sorted chunks. It is queried to retrieve five chunks as shown from left to right in the figure. Note that the chunk retrieval requests are sorted because of MapReduce's shuffling operation. The algorithm creates two read windows, each in charge of reading chunks from the associated batch. Since the chunks are sorted, a read window will only slide downward in the figure. The first request is for chunk 0. It is a read cache miss. Although chunk 0 exists in both batches, the chunk index points to the latest version in batch 2. At this moment, we apply the analysis of Line 4-8 in Algorithm 1, which determines the size of the I/O read window. The only difference is that we skip chunks that do not reside in the current batch (batch 2). As shown in Fig. 7, we find that it is profitable to use a larger read window so that chunk 4 can also be retrieved into the read cache. The request for chunk 1 is processed similarly. Chunk 0 is evicted from the read cache because retrieval requests are always non-decreasing. The next two requests are for chunks 3 and 4. Fortunately, both of the chunks have been retrieved along with previous requests. The two requests hit in the read cache. Finally, the last request is satisfied by reading chunk 9 from batch 1. Since there are no further requests, we use the smallest possible read window in the I/O read.

Even though MRBG-Store is designed to optimize I/O performance, the MRBGraph maintenance could still result in significant I/O cost. The I/O cost might outweigh the savings of incremental processing. The framework is able to detect this situation and automatically turn off MRBGraph maintenance (see [20]).

5.3 Reducing Change Propagation

In incremental iterative computation, changes in the delta input may propagate to more and more kv-pairs as the computation iterates. For example, in PageRank, a change that affects a vertex in a web graph propagates to the neighbor vertices after an iteration, to the neighbors of the neighbors after two iterations, to the three-hop neighbors after three iterations, and so on. Due to this effect, incremental processing may become less effective after a number of iterations.

To address this problem, i^{2}MapReduce employs a *change propagation control* technique, which is similar to the *dynamic computation* in GraphLab [6]. It filters negligible changes of state kv-pairs that are below a given threshold. These filtered kv-pairs are supposed to be very close to convergence. Only the state values that see changes greater than the threshold are emitted for next iteration. The changes for a state kv-pair are accumulated. It is possible a filtered kv-pair may later be emitted if its accumulated change is big enough.

The observation behind this technique is that iterative computation often converges asymmetrically: Many state kv-pairs quickly converge in a few iterations, while the remaining state kv-pairs converge slowly over many iterations. Low et al. has shown that in PageRank computation the majority of vertices require only a single update while only about 3 percent of vertices take over 10 iterations to converge [6]. Our previous work [21] has also exploited this property to give preference to the slowly converged data items.

TABLE 2
API Changes to Hadoop MapReduce

Job Type	Functionality	Vanilla MapReduce (Hadoop)	i^{2}MapReduce
Incremental One-Step	input format	input: $\langle K1, V1' \rangle$	delta input: $\langle K1, V1, `+' / `-' \rangle$
Accumulator Reduce	Reducer class	`reduce(`$K2$`,{`$V2$`})` $\rightarrow \langle K3, V3 \rangle$	`accumulate(`$V2_{old}, V2_{new}$`)` $\rightarrow V2$
	input format	mixed input: $\langle K1, V1' \rangle$	structure input: $\langle SK, SV \rangle$
			state input: $\langle DK, DV \rangle$
Iterative	Projector class		`project(`SK`)` $\rightarrow DK$
			`setProjectType(ONE2ONE)`
	Mappper class	`map(`$K1$`, `$V1$`)` $\rightarrow [\langle K2, V2 \rangle]$	map(SK,SV,DK,DV)$\rightarrow [\langle K2, V2 \rangle]$
			`init(`DK`)` $\rightarrow DV$
	input format	input: $\langle K1, V1' \rangle$	delta structure input: $\langle SK, SV, `+' / `-' \rangle$
Incremental Iterative	change propagation		`job.setFilterThresh(thresh)`
	control		`difference(`DV_{curr}`,`DV_{prev}`)` $\rightarrow diff$

While this technique might impact result accuracy, the impact is often minor since all "influential" kv-pairs would be above the threshold and thus emitted. This is indeed confirmed in our experiments in Section 6.5. If an application has high accuracy requirement, the application programmer has the option to disable the change propagation control functionality.

5.4 Fault-Tolerance

Vanilla MapReduce reschedules the failed Map/Reduce task in case task failure is detected. However, the interdependency of prime Reduce tasks and prime Map tasks in i^{2}MapReduce requires more complicated fault-tolerance solution. i^{2}MapReduce checkpoints the prime Reduce task's output state data and MRBGraph file on HDFS in every iteration.

Upon detecting a failure, i^{2}MapReduce recovers by considering task dependencies in three cases. (i) In case a prime Map task fails, the master reschedules the Map task on the worker where its dependent Reduce task resides. The prime Map task reloads the its structure data and resumes computation from its dependent state data (checkpoint). (ii) In case a prime Reduce task fails, the master reschedules the Reduce task on the worker where its dependent Map task resides. The prime Reduce task reloads its MRBGraph file (checkpoint) and resumes computation by re-collecting Map outputs. (iii) In case a worker fails, the master reschedules the interdependent prime Map task and prime Reduce task to a healthy worker together. The prime Map task and Reduce task resume computation based on the checkpointed state data and MRBGraph file as introduced above. More implementation details and evaluation results of fault tolerance can be found in [20].

6 Experiments

We implement a prototype of i^{2}MapReduce by modifying Hadoop-1.0.3. In order to support incremental and iterative processing, a few MapReduce APIs are changed or added. We summarize these API changes in Table 2 (see [20] for more details). In this section, we perform real-machine experiments to evaluate i^{2}MapReduce.

6.1 Experiment Setup

Solutions to compare. Our experiments compare four solutions: (i) *PlainMR recomp*, re-computation on vanilla Hadoop; (ii) *iterMR recomp*, re-computation on Hadoop optimized for iterative computation (as described in Section 4); (iii) *HaLoop recomp*, re-computation on the iterative MapReduce framework HaLoop [8], which optimizes MapReduce by providing a structure data caching mechanism; (iv) *i^{2}MapReduce*, our proposed solution. To the best of our knowledge, the task-level coarse-grain incremental processing system, Incoop [15], is not publicly available. Therefore, we cannot compare i^{2}MapReduce with Incoop. Nevertheless, our statistics show that without careful data partition, almost all tasks see changes in the experiments, making task-level incremental processing less effective.

Experimental environment. All experiments run on Amazon EC2. We use 32 m1.medium instances. Each m1. medium instance is equipped with 2 ECUs, 3.7 GB memory, and 410 GB storage.

Applications. We have implemented four iterative mining algorithms, including PageRank (one-to-one correlation), Single Source Shortest Path (SSSP, one-to-one correlation), Kmeans (all-to-one correlation), and GIM-V (many-to-one correlation). For GIM-V, we implement iterative matrix-vector multiplication as the concrete application using GIM-V model.

We also implemented a one-step mining algorithm, APriori [22], for mining frequent item sets (see [20] for more details). Note that Apriori satisfies the requirements in Section 3.5. Hence, we employ the accumulator Reduce optimization in incremental processing.

Data sets, delta input, and converged states. Table 3 describes the data sets for the five applications (see [20] for more details). For incremental processing, we generate a delta input from each data set. For APriori, the Twitter data set is collected over a period of two months. We choose the last week's messages as the delta input, which is 7.9 percent of the input. For the four iterative algorithms, the delta input is generated by randomly changing 10 percent of the input data unless otherwise noted. To make the comparison as fair as possible, we start incremental iterative processing *from the previously converged states for all the four solutions*.

TABLE 3
Data Sets

algorithm	data set	size	description
APriori	Twitter	122 GB	52,233,372 tweets
PageRank	ClueWeb	36.4 GB	20,000,000 pages 365,684,186 links
SSSP	ClueWeb2	70.2 GB	20,000,000 pages 365,684,186 links
Kmeans	BigCross	14.4 GB	46,481,200 points each with 57 dimensions
GIM-V	WikiTalk	5.4 GB	100,000 rows 1,349,584 non-zero entries

6.2 Overall Performance

Incremental one-step processing. We use a priori to understand the benefit of incremental one-step processing in i^{2}MapReduce. MapReduce re-computation takes 1,608 seconds. In contrast, i^{2}MapReduce takes only 131 seconds. Fine-grain incremental processing leads to a 12x speedup.

Incremental iterative processing. Fig. 8 shows the normalized runtime of the four iterative algorithms while 10 percent of input data has been changed. “1” corresponds to the runtime of PlainMR recomp.

For PageRank, iterMR reduces the runtime of PlainMR recomp by 56 percent. The main saving comes from the caching of structure data and the saving of the MapReduce startup costs. i^{2}MapReduce improves the performance further with fine-grain incremental processing and change propagation control, achieving a speedup of eight folds (i2MR w/ CPC). We also show that without change propagation control the changes it will return the exact updated result but at the same time prolong the runtime (i2MR w/o CPC). The change propagation control technique is critical to guarantee the performance (see [20] for experimental results). Section 6.5 will discuss the effect of CPC in more details. On the other hand, it is surprising to see that HaLoop performs worse than plain MapReduce. This is because HaLoop employs an extra MapReduce job in each iteration to join the structure and state data [8]. The profit of caching cannot compensate for the extra cost when the structure data is not big enough. Note that the iterative model in i^{2}MapReduce avoids this overhead by exploiting the Project function to co-partition structure and state data. The detail comparison with HaLoop is provided in [20].

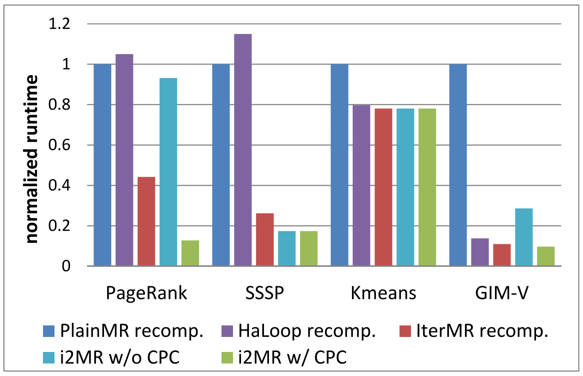


Fig. 8. Normalized runtime.

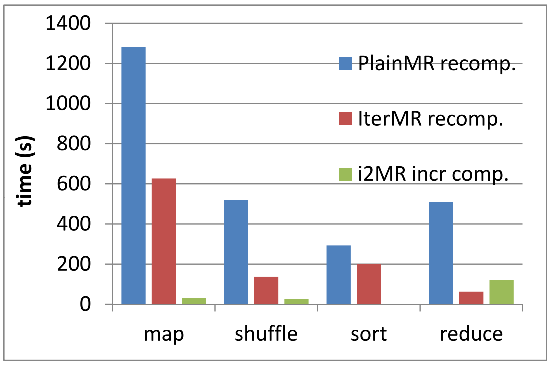


Fig. 9. Run time of individual stages in PageRank.

For SSSP, the performance gain of i^{2}MapReduce is similar to that for PageRank. We set the filter threshold to 0 in the change propagation control. That is, nodes without any changes will be filtered out. Therefore, unlike PageRank, the SSSP results with CPC are precise.

For Kmeans, small portion of changes in input will lead to global re-computation. Therefore, we turn off the MRBGraph functionality. As a result, i^{2}MapReduce falls back to iterMR recomp. We see that HaLoop and iterMR exhibit similar performance. They both outperform plainMR because of similar optimizations, such as caching structure data.

For GIM-V, both plainMR and HaLoop run two MapReduce jobs in each iteration, one of which joins the structure data (i.e., matrix) and the state data (i.e., vector). In contrast, our general-purpose iterative support removes the need for this extra job. iterMR and i^{2}MapReduce see dramatic performance improvements. i^{2}MapReduce achieves a 10.3x speedup over plainMR, and a 1.4x speedup over HaLoop.

6.3 Time Breakdown Into MapReduce Stages

To better understand the overall performance, we report the time[3] of the individual MapReduce stages (across all iterations) for PageRank in Fig. 9.

For the Map stage, IterMR improves the run time by 51 percent because it separates the structure and state data, and avoids reading and parsing the structure data in every iteration. i^{2}MapReduce further improves the performance with fine-grain incremental processing, reducing the plainMR time by 98 percent. Moreover, we find that the change propagation control mechanism plays a significant role. It filters the kv-pairs with tiny changes at the prime Reduce, greatly decreasing the number of Map instances in the next iteration. (see Section 6.5)

For the shuffle stage, iterMR reduces the run time of PlainMR by 74 percent. Most savings result from avoiding shuffling structure data from Map tasks to Reduce tasks. Moreover, compared to iterMR, i^{2}MapReduce shuffles only the intermediate kv-pairs from the Map instances that are affected by input changes, thereby further improving the shuffle time, achieving 95 percent reduction of PlainMR time.

For the sort stage, i^{2}MapReduce sorts only the small number of kv-pairs from the changed Map instances, thus removing almost all sorting cost of PlainMR.

3. The resulted time does not include the structure data partition time, while both the iterMR time and i2MR time in Fig. 8 include the time of structure data partition job for fairness.

TABLE 4
Performance Optimizations in MRBG-Store

technique	# reads	read size(GB)	time (s)
index-only	5,519,910	34.2	718
single-fix-window	1,263,680	1,0512.6	1,361
multi-fix-window	1,188,420	337.8	513
multi-dynamic-window	2,418,809	153.6	467

For the Reduce stage, iterMR cuts the run time of PlainMR by 88 percent because it does not need to join the updated state data and the structure data. Interestingly, i^{2}MapReduce takes longer than iterMR. This is because i^{2}MapReduce pays additional cost for accessing and updating the MRBGraph file in the MRBG-Store. We study the performance of MRBG-Store in the next section.

6.4 Performance Optimizations in MRBG-Store

As shown in Table 4, we enable the optimization techniques in MRBG-Store one by one for PageRank, and report three columns of results: (i) total number of I/O reads in Algorithm 1 (which likely incur disk seeks), (ii) total number of bytes read in Algorithm 1, and (iii) total elapsed time of the merge operation. (i) and (ii) are across all the workers and iterations, and (iii) is across all the iterations. Note that the MRBGraph file maintains the intermediate data distributively, the total size of which is 572.4 GB in the experiment.

First, only the chunk index is enabled. For a given key, MRBG-Store looks it up in the index to obtain the exact position of its chunk, and then issues an I/O request to read the chunk. This approach reads only the necessary bytes but issues a read for each chunk. As shown in Table 4, index-only has the smallest read size, but incurs the largest number of I/O reads.

Second, with a single fix-sized read window, a single I/O read may cover multiple chunks that need to be merged, thus significantly saving disk seeks. However, since PageRank is an iterative algorithm and multiple sorted batches of chunks exist in the MRBGraph file (see Section 5.2), the next to-be-accessed chunk might not reside in the same batch. Consequently, this approach often wastes time reading a lot of obsolete chunks. Its elapsed time gets worse.

Third, we use multiple fix-sized windows for iterative computation. This approach addresses the weakness of the single fix-sized window. As shown in Table 4, it dramatically reduces the number of I/O reads and the bytes read from disks, achieving an 1.4x improvement over the index-only case.

Finally, our solution in i^{2}MapReduce optimizes further by considering the positions of the next chunks to be accessed and making intelligent decisions on the read window sizes. As a result, multi-dynamic-window reads smaller amount of data. It achieves a 1.6x speedup over the index-only case.

6.5 Effect of Change Propagation Control

We run PageRank on i^{2}MapReduce with 10 percent changed data while varying the change propagation filter threshold from 0.1, 0.5, to 1. (Note that, in all previous experiments, the filter threshold is set to 1.) Fig. 10a shows the run time, while Fig. 10b shows the mean error of the kv-pairs, which is the average relative difference from the correct value (computed offline).

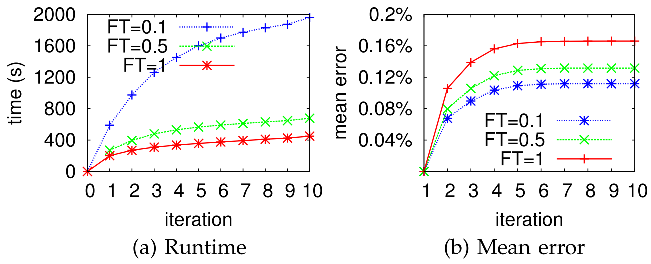


Fig. 10. Effect of change propagation control.

The change propagation control technique filters out the kv-pairs whose changes are less than a given threshold. These filtered kv-pairs are considered very close to convergence. As expected, the larger the threshold, the more kv-pairs will be filtered, and the better the run time (The experiment result about filtered kv-pairs in each iteration is presented in [20]). On the other hand, larger threshold impacts the result accuracy with a larger mean error. Note that "influential" kv-pairs that see significant changes will hardly be filtered, and therefore result accuracy is somewhat guaranteed. In the experiments, all mean errors are less than 0.2 percent, which is small and acceptable. For applications that have high accuracy requirement, users have the option to turn off change propagation control.

7 Related Work

Iterative processing. A number of distributed frameworks have recently emerged for big data processing [3], [5], [6], [7], [21], [23]. We discuss the frameworks that improve MapReduce. HaLoop [8], a modified version of Hadoop, improves the efficiency of iterative computation by making the task scheduler loop-aware and by employing caching mechanisms. Twister [9] employs a lightweight iterative MapReduce runtime system by logically constructing a Reduce-to-Map loop. iMapReduce [10] supports iterative processing by directly passing the Reduce outputs to Map and by distinguishing variant state data from the static data. i^{2}MapReduce improves upon these previous proposals by supporting an efficient general-purpose iterative model.

Unlike the above MapReduce-based systems, Spark [2] uses a new programming model that is optimized for *memory-resident* read-only objects. Spark will produce a large amount of intermediate data in memory during iterative computation. When input is small, Spark exhibits much better performance than Hadoop because of in-memory processing. However, its performance suffers when input and intermediate data cannot fit into memory. We experimentally compare Spark and i^{2}MapReduce in [20], and see that i^{2}MapReduce achieves better performance when input data is large.

Pregel [4] follows the Bulk Synchronous Processing (BSP) model. The computation is broken down into a sequence of supersteps. In each superstep, a Compute function is invoked on each vertex. It communicates with other vertices by sending and receiving messages and performs computation for the current vertex. This model can efficiently support a large number of iterative graph algorithms. Open source implementations of Pregel include Giraph [24], Hama [25], and Pregelix [26]. Compared to i^{2}MapReduce, the BSP model in Pregel is quite different from the MapReduce programming paradigm. It would be interesting future work to exploit similar ideas in this paper to support incremental processing in Pregel-like systems.

Incremental processing for one-step application. Besides Incoop [15], several recent studies aim at supporting incremental processing for one-step applications. Stateful Bulk Processing [13] addresses the need for stateful dataflow programs. It provides a groupwise processing operator Translate that takes state as an explicit input to support incremental analysis. But it adopts a new programming model that is very different from MapReduce. In addition, several research studies [27], [28] support incremental processing by task-level re-computation, but they require users to manipulate the states on their own. In contrast, i^{2}MapReduce exploits a fine-grain kv-pair level re-computation that are more advantageous.

Incremental processing for iterative application. Naiad [14] proposes a timely dataflow paradigm that allows stateful computation and arbitrary nested iterations. To support incremental iterative computation, programmers have to completely rewrite their MapReduce programs for Naiad. In comparison, we extend the widely used MapReduce model for incremental iterative computation. Existing MapReduce programs can be slightly changed to run on i^{2}MapReduce for incremental processing.

8 Conclusion

We have described i^{2}MapReduce, a MapReduce-based framework for incremental big data processing. i^{2}MapReduce combines a fine-grain incremental engine, a general-purpose iterative model, and a set of effective techniques for incremental iterative computation. Real-machine experiments show that i^{2}MapReduce can significantly reduce the run time for refreshing big data mining results compared to re-computation on both plain and iterative MapReduce.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (61300023, 61433008, 61272179), Fundamental Research Funds for the Central Universities (N141605001, N120816001), China Mobile Fund (MCM20125021), and MOE-Intel Special Fund of Information Technology (MOE-INTEL-2012-06). The second author is partially supported by the CAS Hundred Talents program and by NSFC Innovation Research Group No. 61221062. S. Chen is the corresponding author.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Opear. Syst. Des. Implementation*, 2004, p. 10.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for, in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, p. 2.

[3] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation*, 2010, pp. 1–14.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[5] S. R. Mihaylov, Z. G. Ives, and S. Guha, "Rex: Recursive, delta-based data-centric computation," in *Proc. VLDB Endowment*, 2012, vol. 5, no. 11, pp. 1280–1291.

[6] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," in *Proc. VLDB Endowment*, 2012, vol. 5, no. 8, pp. 716–727.

[7] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," in *Proc. VLDB Endowment*, 2012, vol. 5, no. 11, pp. 1268–1279.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," in *Proc. VLDB Endowment*, 2010, vol. 3, no. 1–2, pp. 285–296.

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Proc. 19th ACM Symp. High Performance Distributed Comput.*, 2010, pp. 810–818.

[10] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," *J. Grid Comput.*, vol. 10, no. 1, pp. 47–68, 2012.

[11] S. Brin, and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1–7, pp. 107–117, Apr. 1998.

[12] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation*, 2010, pp. 1–15.

[13] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 51–62.

[14] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. 24th ACM Symp. Oper. Syst. Principles*, 2013, pp. 439–455.

[15] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 7:1–7:14.

[16] J. Cho and H. Garcia-Molina, "The evolution of the web and implications for an incremental crawler," in *Proc. 26th Int. Conf. Very Large Data Bases*, 2000, pp. 200–209.

[17] C. Olston and M. Najork, "Web crawling," *Found. Trends Inform. Retrieval*, vol. 4, no. 3, pp. 175–246, 2010.

[18] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inform. Theory.*, vol. 28, no. 2, pp. 129–137, Mar. 1982.

[19] U. Kang, C. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proc. IEEE Int. Conf. Data Mining*, 2009, pp. 229–238.

[20] Y. Zhang, S. Chen, Q. Wang, and G. Yu, "i^{2}mapreduce: Incremental mapreduce for mining evolving big data," *CoRR*, vol. abs/1501.04854, 2015.

[21] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritized iterative computations," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 13:1–13:14.

[22] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. 20th Int. Conf. Very Large Data Bases*, 1994, pp. 487–499.

[23] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Accelerate large-scale iterative computation through asynchronous accumulative updates," in *Proc. 3rd Workshop Sci. Cloud Comput. Date*, 2012, pp. 13–22.

[24] Apache giraph. [Online]. Available: http://giraph.apache.org/, 2015.

[25] Apache hama. [Online]. Available: https://hama.apache.org/, 2015.

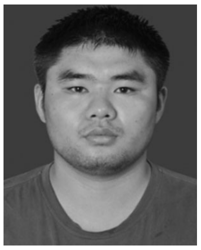
[26] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregelix: Big(ger) graph analytics on a dataflow engine," in *Proc. VLDB Endowmen*, 2015, vol. 8, no. 2, pp. 161–172.

[27] C. Yan, X. Yang, Z. Yu, M. Li, and X. Li, "IncMR: Incremental data processing based on mapreduce," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp.pp. 534–541.

[28] T. Jörg, R. Parvizi, H. Yong, and S. Dessloch, "Incremental recomputations in mapreduce," in *Proc. 3rd Int. Workshop Cloud Data Manage.*, 2011, pp. 7–14.

Yanfeng Zhang received the PhD degree in computer science from Northeastern University, China, in 2012. He is currently an associate professor at Northeastern University, China. His research consists of distributed systems and big data processing. He has published many papers in the above areas. His paper in ACM Cloud Computing 2011 was honored with "Paper of Distinction".

Shimin Chen received the PhD degree from Carnegie Mellon University in 2005, and worked as a researcher, senior researcher, and research manager at Intel Labs, Carnegie Mellon University, and HP Labs before joining ICT. He is a professor at ICT CAS. His research interests include data management systems, computer architecture, and big data processing.



Qiang Wang received the BE degree in 2012 from the Hebei University of Economics and Business and the ME degree in 2014 from Northeastern University. His current research interests include cloud computing and big data processing.

Ge Yu received the PhD degree in computer science from the Kyushu University of Japan in 1996. He is now a professor at the Northeastern University, China. His current research interests include distributed and parallel database, OLAP and data warehousing, data integration, graph data management, etc. He has published more than 200 papers in refereed journals and conferences. He is a member of the IEEE, ACM, and CCF.