J. Parallel Distrib. Comput. 83 (2015) 58–69

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

GOM-Hadoop: A distributed framework for efficient analytics on ordered datasets☆

Jiangtao Yin [a,*], Yong Liao [b], Mario Baldi [d,c], Lixin Gao [a], Antonio Nucci [c]

[a] University of Massachusetts Amherst, 151 Holdsworth Way, Amherst, MA 01003, USA
[b] China Academy of Electronics and Information Technology, China
[c] Cisco Systems, USA
[d] Politecnico di Torino, Italy

HIGHLIGHTS

- We generalize a class of big data analytics workload (Re-Org) on ordered datasets.
- We propose a novel distributed mechanism for efficiently executing Re-Org tasks.
- The proposed mechanism is implemented in a distributed framework by extending Hadoop.
- A model is presented to formally study the proposed framework.
- Experiments show that our framework is 6.3x faster than vanilla Hadoop.

ARTICLE INFO

Article history:
Received 28 July 2014
Received in revised form
25 April 2015
Accepted 21 May 2015
Available online 27 May 2015

Keywords:
GOM-Hadoop
Distributed framework
MapReduce
Ordered dataset

ABSTRACT

One of the most common datasets exploited by many corporations to conduct business intelligence analysis is event log files. Oftentimes, the records in event log files are temporally ordered, and need to be grouped by certain key with the temporal ordering preserved to facilitate further analysis. One such example is to group temporally ordered events by user ID in order to analyze user behavior. This kind of analytical workload, here referred to as RElative Order-pReserving based Grouping (Re-Org), is quite common in big data analytics, where the MapReduce programming paradigm (and its open-source implementation, Hadoop) is widely adopted for massive parallel processing. However, using MapReduce/Hadoop for executing Re-Org tasks on ordered datasets is not efficient due to its internal sort–merge mechanism when shuffling data from mappers to reducers. In this paper, we propose a distributed framework that adopts an efficient group-order–merge mechanism to speed up the execution of Re-Org tasks. We demonstrate the advantage of our framework by formally modeling its execution process and by comparing its performance with Hadoop through extensive experiments on real-world datasets. The evaluation results show that our framework can achieve up to 6.3x speedup over Hadoop in executing Re-Org tasks.

© 2015 Elsevier Inc. All rights reserved.

1. Introduction

Large corporations, such as Google, Amazon, and Facebook, routinely produce and collect terabytes of data on a daily basis, and continually improve their services and operations by analyzing the data. Completing the analysis of data at the scale of terabytes or even petabytes in a short time becomes a daunting task.

A large class of datasets used to gain business intelligence are often fundamentally temporal, such as webpage click streams, network traffic traces, and business transactions. Furthermore, a lot of analytical tasks over such temporal data require to group data points of a certain feature together and impose the temporal ordering on the data points in the same group. Such a processing is a vital step in many important analytical jobs, including:

- User sessionization [6,18]: widely used in recommendation systems, behavioral targeting advertisement display, and personalized web services.

☆ Part of this work has been published in Proceedings of HPDC'13: ACM Symposium on High-Performance Parallel and Distributed Computing (Yin et al. 2013, [36]), and Proceedings of UCC'13: IEEE/ACM International Conference on Utility and Cloud Computing (Yin et al. 2013, [35]).

* Corresponding author.
E-mail addresses: jyin@ecs.umass.edu (J. Yin), ly@ustc.edu (Y. Liao), baldi@polito.it (M. Baldi), lgao@ecs.umass.edu (L. Gao), anucci@cisco.com (A. Nucci).

http://dx.doi.org/10.1016/j.jpdc.2015.05.003
0743-7315/© 2015 Elsevier Inc. All rights reserved.

- Flow construction [28,5]: broadly utilized in traffic engineering and network security.
- Customer statement generation [14,8]: applied to billing, fraud detection, and risk analytics.

The input datasets for the above tasks can be generally seen as event log files, where each record is about an event. An event usually has some attributes, such as the event type, the event origin, and a timestamp of when the event happened. Such datasets often have an important property. That is, as a dataset is generated, the records in the dataset are already placed in certain order. For event log files, the ordering is often based on the timestamps of the events because earlier events are recorded before events occurring later.

In general, an input dataset in big data analytics with the same property as event log files can be represented as a list of *records*. Each record consists of a *primary key*, a *secondary key*, and a *value*. The records in the input dataset are already ordered by their secondary keys (e.g., timestamps). For such an input dataset, we define *RElative Order-pReserving based Grouping*, or Re-Org, as a processing that generates a set of output data points, each of which is a *group* of input records. Further, those groups of input records should satisfy the following requirements: (1) records are grouped based on their primary keys; (2) all the records in a group are ordered by their secondary keys.

Since a Re-Org task usually involves a huge quantity of data, parallelizing its execution is necessary (or at least desirable). MapReduce [12] has emerged as a popular scalable distributed framework for data intensive computation. It provides a simple programming model to allow a user to focus on the business logic in the analytics without worrying about the complexity of parallel computation. However, realizing Re-Org tasks using MapReduce is not efficient. MapReduce groups data by their keys via utilizing an internal *sort–merge* scheme, which cannot take advantage of the fact that the input records for Re-Org tasks are already placed in the order of the secondary key. To enable records with the same primary key to be sorted by their secondary keys, one has to rely on time-consuming sorting either in custom code or by instrumenting MapReduce to do so.

In this paper, we propose a *group-order–merge* mechanism for efficiently realizing Re-Org tasks in a distributed environment. Our proposed mechanism maximally utilizes the property of the input datasets to speed up the execution of Re-Org tasks. It efficiently utilizes hash techniques in grouping records on their primary keys and preserving the relative order of records with the same primary key. The hash table used in our mechanism is also designed to provide a lightweight way for imposing a global ordering of the grouped records across multiple worker nodes with limited sorting operations. The global ordering is utilized later when records are merged in parallel by different worker nodes to yield record groups, where each group has all records with the same primary key and the records are ordered by their secondary keys.

We have built a distributed framework for supporting the group-order–merge mechanism by extending Hadoop [4], the most popular open-source implementation of MapReduce. Our framework is referred to as *Group-Order–Merge Hadoop (GOM-Hadoop)*. To avoid memory overflow, the hash techniques utilized in the group-order–merge mechanism are implemented with bounded memory usage. Moreover, GOM-Hadoop retains all salient features of vanilla Hadoop, such as fault-tolerance, speculative execution, and data locality. We demonstrate the advantage of GOM-Hadoop (for Re-Org tasks) by formally modeling its execution process. We evaluate it by implementing different types of Re-Org tasks with real-world datasets on a local cluster of machines as well as on Amazon EC2 Cloud [3]. The evaluation results show that GOM-Hadoop can achieve up to 6.3x speedup over vanilla Hadoop.

```
1353637 - - [13/Jun/1998:22:00:01] "GET /r01.gif HTTP/1.0" 200 929
230887 - - [13/Jun/1998:22:00:01] "GET /venues.gif HTTP/1.0" 200 778
1353637 - - [13/Jun/1998:22:00:01] "GET /btm.gif HTTP/1.0" 200 283
1353638 - - [13/Jun/1998:22:00:02] "GET /bord_d.gif HTTP/1.1" 200 231
1353638 - - [13/Jun/1998:22:00:02] "GET /bord_g.gif HTTP/1.1" 200 231
```

Fig. 1. Sampled click stream data. IP addresses are mapped to integers.

The rest of the paper is organized as follows. Section 2 formally defines the problem targeted by this paper. Section 3 briefly surveys how the problem is solved with existing techniques. Section 4 presents our scheme for efficiently executing Re-Org tasks in a distributed environment. The framework for supporting the proposed scheme is presented in Section 5. An analytical model of the framework is provided in Section 6. Section 7 presents the evaluation results. Section 8 surveys related work. Section 9 concludes this paper.

2. Problem definition

In this section, we first describe a series of well-known applications that our framework targets. We then formulate them into one general problem.

2.1. Motivating applications

Click stream analysis. Many companies have web services and are interested in analyzing the click stream logs of their websites, which can provide tremendously valuable information. For instance, one can detect customer click patterns from the click stream data, and such click patterns are used for advertisement promotion, revenue prediction, and service personalization. One common step of analyzing the click stream data is to divide users' clicks into *sessions* [6,18]. Usually, a session consists of a user's temporally ordered clicks and is considered to be finished if the user has no clicks for some time duration (e.g., 5 min). Intelligence that can be gathered from sessionized clicks includes: the sequence of clicks in a session represents the fine-grained navigational behavior of a user; the session durations show how much time a user spends on the website each time; the last accessed pages of those sessions (i.e., "killer pages") give some hints on why a user leaves.

Network traffic analysis. ISPs and enterprise ITs often use tools such as Cisco NetFlow [11] to extract metadata about the traffic in their networks. Various network management and optimization tasks rely on analytics on the metadata. The metadata analytics often needs to group them based on certain criteria, such as grouping the metadata for flows from a common source. It is also often needed to sort those grouped metadata based on the timestamp because a lot of analytics tasks, such as malicious behavior detection, require correlating metadata at different time.

Customer statement generation. Banks and e-commerce companies usually need to divide their customer transactions into statements in their business operations. Transactions of each customer are grouped together and then sorted by their timestamps. The generated customer statements can be applied to billing, fraud detection, and risk analytics [14,8].

2.2. Formal problem setting

The input datasets for the above applications can be seen as event log files. In general, such an input dataset can be parsed as a list of *records*. Each record consists of a *primary key*, a *secondary key*, and a *value*. The records in the input dataset are already sorted by their secondary keys. Take the click stream data for example. As presented in Fig. 1, each click can be seen as a record, where the

source IP can be considered as the primary key, the timestamp as the secondary key, and other attributes of the click as the value. These clicks are sorted by their timestamps.

We define *RElative Order-pReserving based Grouping* mechanism, or Re-Org, as a processing that generates a set of output data points, each of which is a *group of records* from the input dataset. All the records with the same primary key are in and only in one output data point, and all the records in an output data point are sorted by their secondary keys. Take user sessionization for example. All the clicks belonging to one user can be seen as one output data point. Note that one may further apply a user-defined aggregation operation to output data points so as to produce final results. In user sessionization, we further split a user's clicks into sessions by traversing them in the order of timestamp (with a timeout threshold).

Now we give the formal definition of a Re-Org task. The input dataset of a Re-Org task can be parsed as a list of records:

$$R_{in} = [r_0, r_1, \ldots, r_n]. \tag{2.1}$$

Each record $r_i \in R_{in}$ consists of a primary key p_i, a secondary key s_i, and a value v_i, i.e., $r_i = \{p_i, s_i, v_i\}$. Any two records r_i and r_j in R_{in} are already ordered by their secondary keys. Let $\preceq$ represent the ordering. We have

$$s_i \preceq s_j \iff i < j, \quad \forall r_i, r_j \in R_{in}. \tag{2.2}$$

The output is a set of data points, which is represented as

$$R_{out} = \{\hat{g}_0, \hat{g}_1, \ldots, \hat{g}_m\}. \tag{2.3}$$

Each output data point $\hat{g}_i$ in R_{out} is a group of input records with the same primary key, i.e., $\hat{g}_u = [r_{u_0}, r_{u_1}, \ldots, r_{u_k}]$ where

$$p_{u_i} = p_{u_j} \iff r_{u_i} \in \hat{g}_u \quad \text{and} \quad r_{u_j} \in \hat{g}_u. \tag{2.4}$$

Furthermore, the records of an output data point are in the order defined by the secondary key:

$$s_{u_i} \preceq s_{u_j} \iff i < j, \quad \forall r_{u_i}, r_{u_j} \in G_u. \tag{2.5}$$

3. Existing MapReduce support for RE-ORG tasks

In this section, we discuss how to implement Re-Org tasks on Hadoop, an open-source framework for running MapReduce jobs. To ground our discussion, we begin with an overview of the MapReduce programming model and Hadoop. Then, we present two commonly adopted mechanisms to implement Re-Org tasks on MapReduce/Hadoop and point out their issues.

3.1. MapReduce/Hadoop

MapReduce, a popular distributed programming model for processing massive datasets in a cluster of commodity machines, has attracted a lot of attention over the past several years [13,19,38,39,37,1,16,2,7,27,15,25,26,9,17,20,24,21,33,23,10,22,29,34,30,40,35,36].

The essential functionality of the MapReduce programming model is to group data by key. The MapReduce programming model consists of two functions, the `map()` function and the `reduce()` function. Hadoop is the most popular open-source implementation of MapReduce. It leverages a sort–merge scheme to group data by key. Hadoop runs a MapReduce job by dividing it into two phases: the mapper phase and the reducer phase. When a mapper reads a trunk of data from HDFS (Hadoop Distributed File System), its `map()` function is called to produce a set of intermediate key–value pairs. Each intermediate key–value pair is assigned with a *partition number*, which is generated by applying a *partition function* to the key. Each partition number corresponds to one reducer.

Those key–value pairs are serialized into an in-memory buffer of the mapper. When the buffer is full, Hadoop performs sorting on the key–value pairs with partition numbers (using quicksort by default). The sorting orders those key–value pairs first by their partition numbers and then by their keys. A *key comparator* is used to determine which key is "larger" when comparing two keys. The sorted key–value pairs are written into a local disk as a *spill file*. Multiple spill files are merged together as the mapper output. A reducer merges the sorted outputs of different mappers it has fetched and groups key–value pairs with the same key through a *grouping comparator*. Then, the reducer passes the key of each group and the list of values within that group to its `reduce()` function.

3.2. Basic MapReduce approach

For input dataset $R_{in} = [r_0, r_1, \ldots, r_n]$, the basic implementation of Re-Org on Hadoop is as follows. A `map()` function transforms each input record (e.g., $r_i = \{p_i, s_i, v_i\}$) into one intermediate key–value pair, where the key is the primary key of the record (p_i), and the value consists of the secondary key of the record (s_i) and its value (v_i). In other words, the output of one `map()` function is a set of intermediate key–value pairs denoted as $[\{p_i, (s_i, v_i)\}]$. Hadoop ensures that all the key–value pairs with the same key are fed into the same `reduce()` function. In the `reduce()` function, one can have custom code to buffer values and to sort all values based on the secondary key. However, when a reducer receives a large number of values for a given key, it may run out of memory. As a result, this approach is not scalable.

3.3. Hadoop secondary sort

Hadoop has a built-in scheme for imposing order on the values. The scheme is usually referred to as *Hadoop secondary sort* [31]. In order to realize a Re-Org task using Hadoop secondary sort, a user needs to define the `map()` function to transform each input record (e.g., $r_i = \{p_i, s_i, v_i\}$) into one intermediate key–value pair as well. Within an intermediate key–value pair, the key consists of the record's primary key (p_i) and its secondary key (s_i), and the value is the record's value (v_i). Since the key produced by the `map()` function is a composition of the primary key and the secondary key, it is often called the *composite key*. Then, by customizing the key comparator, the user instruments Hadoop to sort key–value pairs based on the composite keys: first by the primary key and then by the secondary key. In order to enable that key–value pairs with the same primary key will be processed by the same reducer, the user needs to define a new partition function to assign a partition number according to the primary key only. In addition, the user needs to provide a customized grouping comparator to group key–value pairs via their primary keys only. As a result, all the values with the same primary key are sorted by the secondary key when fed into the `reduce()` function.

Although both of the aforementioned approaches can realize a Re-Org task, the sort–merge mechanism in the current MapReduce/Hadoop framework introduces unnecessary overhead. It does not utilize the fact that the input records are already placed in the order of the secondary key.

4. Our solution for RE-ORG tasks

In this section, we present our solution for realizing a Re-Org task in a distributed environment, maximally utilizing the property of the input dataset to speed up the process. We first describe the challenges of realizing a Re-Org task in a distributed environment and then illustrate how to efficiently solve those challenges.

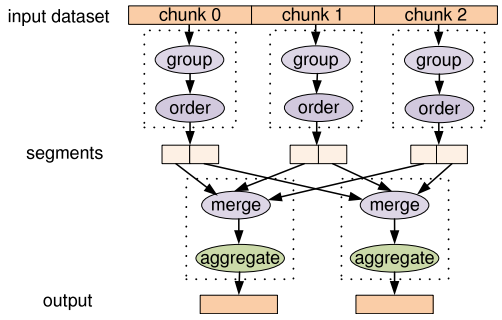


Fig. 2. Basic workflow of the proposed group-order–merge mechanism.

4.1. Challenges of distributed RE-ORG

A RE-ORG task needs to group records by their primary keys and to enable records in a group to be sorted by their secondary keys. As the input records are already sorted by the secondary key, a RE-ORG task can be easily done in the single machine scenario. For instance, the machine sequentially parses each record from the input dataset and puts them into a hash table by hashing on their primary keys. Then, each entry of the hash table has all records associated with one primary key. Since records are processed sequentially, each entry also preserves the ordering of the records from the input dataset (sorted by the secondary key).

However, it is challenging to realize a distributed RE-ORG task. To support parallel processing, the input dataset is often divided into multiple pieces and multiple workers process those pieces in parallel. Although each worker can group input records using a simple hashing technique as in the above single machine case, an entry in its hash table may not have all the records associated with a primary key. In other words, the records with the same primary key can scatter in different workers. Accordingly, we need to merge the hashing results from different workers so that all the records with the same primary key are in the same group. In addition, records in the merged group need to be sorted by their secondary keys. *Efficiently* merging the hashing results from multiple workers and restoring the order of records in each merged group are the challenges in realizing a distributed RE-ORG task.

4.2. Logical workflow of GOM-Hadoop

We propose a novel group-order–merge mechanism to efficiently support RE-ORG tasks in a distributed environment. Our proposed mechanism consists of three phases: the *group phase*, the *order phase*, and the *merge phase*. A high-level illustration of our mechanism is presented in Fig. 2.

The input dataset is first split into *chunks*. Each data chunk is assigned to a logical worker to process. Note that our mechanism does not require the input dataset to be stored in one single file. As long as each chunk is a consecutive block of the input dataset, our mechanism will be applicable. In the group phase, a worker sequentially extracts the records from its input chunk and groups the records by applying a two-level hashing technique on their primary keys. The output of the group phase is a set of *segments*. The reason a worker produces multiple segments is that multiple segments can support parallel processing (e.g., parallel merging). Each segment contains a set of *lists*. Each list has all the records in that chunk with the same primary key, and those records preserve the ordering they have in the chunk (because the worker sequentially processes records).

The ordering is then applied on each segment so that the set of lists in the segment can have some global ordering based on their primary keys. This ordering is important for efficient merging of segments produced by different workers. Note that each list is treated as a whole in the order phase, which is much more efficient than treating each record individually. After the order phase is done, the worker sends all its segments to another set of workers. The segments are sent out in a manner that all lists whose records have the same primary key will be received by the same worker.

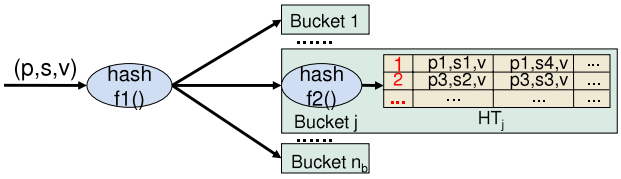


Fig. 3. Grouping records by their primary keys via hashing.

Workers in the merge phase process the segments produced in the order phase, merge the records with the same primary key into one final list, and ensure the records in the list preserving their relative ordering. Then the user-defined aggregation operation is applied to the final list to generate the final result. The aggregation operation is not part of the proposed mechanism but constitutes the implementation of the actual analytical task.

In the rest of this section, we present the detailed description on how our group-order–merge scheme works. Its implementation details will be presented in Section 5.

4.3. Grouping via hashing primary keys

The group phase groups records from one input chunk via two-level hashing on their primary keys. The first hash function, shown as $f1()$ in Fig. 3, disperses records into a fixed number (n_b) of hash buckets, where each bucket corresponds to one segment. The records hashed into bucket j by $f1()$ are stored in a logical hash table HT_j. The table has an array of entries. Each entry is associated with an integer index. We leverage another hash function $f2()$ (independent of $f1()$) to map the primary key of a record to an entry index in the table. For simplicity of exposition, we assume hash function $f2()$ can perform a 1-to-1 mapping so that an entry in the logical hash table has only those records with the same primary key. A new record is always appended to the tail of the corresponding entry in the hash table. Since records are processed sequentially and they are already ordered by the secondary key, the records in each entry preserve the ordering on their secondary keys.

Note that here we assume all records from one chunk can fit in a worker's memory for the sake of easy explanation. In Section 5.1, we will show that our mechanism is actually implemented with bounded memory usage.

4.4. Hash assisted ordering

The order phase streams out those n_b hash tables populated in the group phase to a local disk. Each hash table is streamed out as one segment. For hash table HT_j, the order phase imposes the ordering of its entries through ordering their indexes in the table when the entries are streamed out to a local disk. The order of record lists in each segment is important for efficient merging of segments. Without the ordering, our merge phase cannot simply use the linear time merge part of the merge–sort algorithm. Since the order of the record lists in a segment is based on the hashcodes (indexes of a hash table), we call it *hash-based order*.

Once a worker finishes generating the ordered segments, it sends them to another set of workers. The segments are sent out in a way that the jth segments of all workers running in the order phase will be sent to the same worker (i.e., worker j) running in the merge phase.

Note that in the ideal scenario without hash collision, the hash assisted ordering on record lists can completely avoid comparisons on primary keys when producing ordered segments. In reality, hash collision is inevitable so we need comparisons on primary keys in some cases. In Section 5.2, we will present our implementation which can yield ordered segments with minimal primary key comparisons.

4.5. Relative order based merge

The merge phase merges multiple segments into one final stream, where the records with the same primary key are consolidated into one final list. Records in the final list are ordered by their secondary keys.

We adopt the idea of merge–sort to efficiently merge multiple segments, since record lists in those segments are already in the hash-based order. Besides, because different segments are generated from different data chunks and the input dataset is ordered by the secondary key, all secondary keys in segment i should be either "larger" or "smaller" than all secondary keys in segment j. Therefore, one sorting can put the segments in an order based on the secondary keys of those records in them.

Once we have a sorting of those segments, we can start to merge them. During the merging, we always pick the lists at the heads of each segment and move the "smallest" one (the record with the "smallest" primary key or the record with the "smallest" secondary key when more than one record has the "smallest" primary key) into the stream.

5. Framework implementation

In this section, we present the implementation of our GOM-Hadoop framework for efficiently executing Re-Org tasks. Our GOM-Hadoop framework extends Hadoop to incorporate the group-order–merge scheme described in Section 4 as an alternative shuffle mechanism to its default sort–merge mechanism. Here, the shuffle mechanism means the whole process from the point where the `map()` function produces key–value pairs to the point where the `reduce()` function consumes these key–value pairs. Essentially, our GOM-Hadoop framework brings new implementation to MapReduce rather than improves its semantics. Our GOM-Hadoop framework can also execute ordinary Hadoop jobs with negligible overhead. The difference between GOM-Hadoop and vanilla Hadoop is that GOM-Hadoop uses a different way (i.e., the hash-based technique) to implement the shuffle mechanism. Furthermore, GOM-Hadoop retains all salient features of vanilla Hadoop, such as fault-tolerance, speculative execution, and data locality, even when executing a Re-Org task. As a result, GOM-Hadoop also inherits the scalability and reliability properties of vanilla Hadoop. The prototype implementation of our GOM-Hadoop framework is based on Hadoop version 1.0.3.

5.1. Mapper side grouping

When a Re-Org task is realized using GOM-Hadoop, the mapper is instrumented to use only the primary key of each input record as its output key, and to use the secondary key and the record's value as its output value. The key–value pairs produced by the `map()` function are serialized into a memory buffer. Each key–value pair in the buffer is assigned a *partition number* by applying a partition function to the key. Each key–value pair is also assigned an index for quick lookup in the buffer. The key–value pairs' partition numbers and indexes are stored in an auxiliary memory buffer. When either one of these two buffers reaches its maximum capacity, our implementation uses the hash-based technique presented in Section 4.3 to group the key–value pairs' indexes (by hashing key–value pairs but storing their indexes). It is important to note that the buffers will not accept new key–value pairs/indexes until their contents are spilled out to a local disk. Hence, there is no memory overflow problem. Additionally, since a hash table stores indexes instead of the actual key–value pairs and the size of an index is typically much smaller than that of a key–value pair, the memory overhead of the hash-based technique is small. In this group phase, each index is first put into a hash bucket by reusing its corresponding partition number as the hash key (i.e., the partition function is used as hash function $f1()$ in Fig. 3).

For indexes with the same partition number, we design a logical hash table to store them. The logical hash table has a fixed number (n_s) of slots, and each slot has a small hash table. The logical hash table first utilizes a hash function $h1()$ to disperse the indexes into slots via hashing on their corresponding keys. The indexes entering one slot are stored in one small hash table, which uses another independent hash function $h2()$ to map between the indexes' corresponding keys and its entries. Each entry stores indexes for one key, and the hash table uses separate chaining to resolve hash collision. An index is always appended to the tail of the corresponding entry.

5.2. Mapper side ordering

The order phase picks the indexes stored in hash tables in certain order and streams out their corresponding key–value pairs into a spill file. In this way, key–value pairs in the spill file are ordered (i.e., in the hash-based order), as illustrated in Section 4.4.

The indexes are picked in the following way. Indexes in bucket i (for partition i) are picked before the indexes in bucket j (for partition j), if $i < j$. In each bucket, the indexes are already ordered across slots, i.e., indexes in slot e have smaller hashcodes generated by hash function $h1()$ than indexes in slot f if $e < f$. Therefore, indexes in slot e are picked before the indexes in slot f, if $e < f$. However, to save space, the small hash table of each slot has a dynamic number of entries, and thus it does not support a fixed order of its entries over time (because it might be rehashed). In order to obtain a fixed order of a small hash table's entries, the order phase orders the entries by sorting the corresponding keys. For efficiency, it sorts their hashcodes given by hash function $h2()$ first and then sorts the keys themselves. After sorting, entries are picked by following their sorted order. The order phase uses each entry to generate a list of key–value pairs in the spill file via streaming out the key–value pairs indexed by the entry.

For each generated spill file, the key–value pairs are divided into partitions. In each partition, the pairs are ordered by the key in hash-based order, and those with the same key preserve their original relative ordering.

5.3. Mapper side merging

Similar to the behavior of vanilla Hadoop, merging of spill files happens at both the mapper side and the reducer side in GOM-Hadoop. At the mapper side, multiple spill files generated in the order phase are merged into one single output file. Depending on the merge parameter, multiple merging rounds might occur. Since each spill file already has the key–value pairs ordered in the order phase, the merging can be done by using the merge part of the merge–sort algorithm. That is, the mapper side merging can linearly scan each spill file in an interleaved way and pass the "smallest" one of all the currently encountered key–value pairs to the output file.

If two records have the same primary key, we need to use their secondary keys to determine which one is "smaller". Note that because a mapper sequentially processes records in an input

chunk, the key–value pairs in the $(i+1)$th spill file are guaranteed to have a "smaller" secondary key than the key–value pair in the ith spill file. To this end, each key–value pair is extended to a triple tuple, which includes a counter. The counter remembers the number of spill files the mapper already generated. The counter can be used to determine the order of tuples (key–value pairs) from different spill files. A smaller counter value means a "smaller" secondary key. Moreover, since it is an integer, the counter takes only a few bytes and thereby incurs negligible space overhead.

After a mapper merges all its spill files into one single output file, the partitions in that output file are sent to reducers. Similar to the behavior of vanilla Hadoop, GOM-Hadoop determines a partition to be sent to which reducer by its partition number (generated by the partition function).

5.4. Reducer side merging

Merging also occurs at the reducer side because a reducer needs to merge all the partitions it fetched from mappers into one single stream before feeding them into the `reduce()` function. Similar to the mapper side merging, the reducer side merging might occur in multiple rounds as well.

All key–value pairs in one partition come from one input chunk, which is a consecutive block of the input dataset. Hence, if one key–value pair of a partition has a "smaller" secondary key than that in another partition, all the key–value pairs in the former partition have "smaller" secondary keys than those in the latter partition. Similar to the counter in the spill file, we enable each mapper to use its numeric ID to extend key–value pairs when creating its output file. However, since we do not control a mapper to read which chunk, that one mapper has a smaller ID does not necessarily mean that the key–value pairs it produced have "smaller" secondary keys. To solve this problem, we pick one key–value pair from each partition that is under merging and then sort them by their secondary keys. Accordingly, the picked key–value pairs have an order, and so do the IDs in these pairs. Consequently, we know which ID denotes a "smaller"/"larger" secondary key.

6. Performance model

In this section, we build an analytical model for GOM-Hadoop to better understand its overall capability in executing Re-Org tasks. We leverage this model to compare the performance of our GOM-Hadoop with that of vanilla Hadoop so as to demonstrate its advantage.

6.1. Execution of RE-ORG tasks

Similar to vanilla Hadoop, the execution of a Re-Org task in GOM-Hadoop consists of the execution of mappers and that of reducers. We divide the execution of a mapper in GOM-Hadoop into three stages, as shown in Fig. 4. The operations in each stage are as follows:

1. Parse: reading the input chunk from HDFS and parsing key–value pairs from the data chunk.
2. Group-order: collecting key–values pairs in buffer and materializing these key–value pairs into a local disk as spill files after grouping and ordering them.
3. Merge: merging multiple spill files into one single mapper output file.

A reducer starts to fetch data from mappers when at least one mapper has completed its processing. The data is then merged and the aggregation operation is applied. We divide the reducer execution into three stages as well. As presented in Fig. 5, the three reducer stages are:

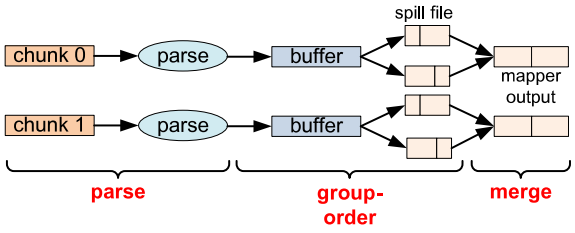


Fig. 4. Execution of a mapper.

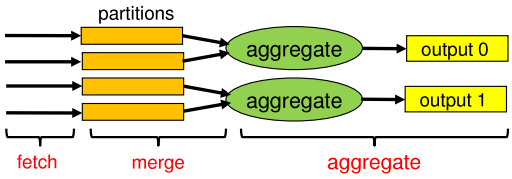


Fig. 5. Execution of a reducer.

Table 1
Notations used in the model.

Notation	Description
D	Input dataset size
C	Chunk size in HDFS
R	Number of reducers
I	Ratio of output size to input size for mapper
B	Output buffer size for mapper
A	io.sort.factor
L	Size of each record in input dataset
S	Size of each k–v pair in mapper output

1. Fetch: fetching its partitions of the mapper outputs from all mappers.
2. Merge: merging all key–value pairs with the same key (from different partitions) into one group.
3. Aggregate: applying the aggregation operation to each group of key–value pairs so as to produce the final output (to be stored in HDFS).

6.2. The model

We analyze the performance of our GOM-Hadoop framework in executing a Re-Org task by modeling the amount of time needed in each stage of the execution in mapper/reducer. Before presenting the model, we list in Table 1 the notations used in the modeling.

In the parse stage, a mapper sequentially processes each input record. The record parsing time is linear to the number of input records. There are $\frac{C}{L}$ input records in each chunk. Letting T_1 be the time needed to parse one input record, a mapper needs $\frac{C \cdot T_1}{L}$ time to finish the parse stage. In the group-order stage, our framework uses hashing to process key–value pairs, and each pair is processed once. The processing time in this stage is approximately linear to the number of key–value pairs, as demonstrated in Section 6.4. Therefore, the group-order stage takes $\frac{C \cdot T_2}{L}$ time, where T_2 represents the time needed to order one key–value pair. The number of rounds needed by a mapper to merge spill files depends on the merge parameter (i.e. parameter `io.sort.factor` in Hadoop configuration), which determines the maximum number of spill files that can be simultaneously merged. For simplicity, we assume the number of spill files ($\lceil \frac{C \cdot I}{B} \rceil$) is smaller than a number, i.e., $\lceil \frac{C \cdot I}{B} \rceil \leq A^2 - A + 1$ (in this case there are no more than 2 rounds of merging) [21]. Similar to the default setting of Hadoop, we set A to be 10 in GOM-Hadoop, then $A^2 - A + 1 = 91$. The input chunk size (C) is usually less than 1 GB, and the mapper buffer (B) is often larger than 100 MB. Accordingly, a mapper typically generates less

than 10 spill files. The times of a record to be merged under this setting, denoted as function $F_n()$, can be represented as:

$$F_n\left(\left\lceil\frac{C\cdot I}{B}\right\rceil, A, B, S\right) = n_f\cdot\beta\cdot I(cd_1) + \left[2\cdot\left\lfloor\frac{n_f}{A}\right\rfloor\cdot A + (n_f\%A)\right]\cdot\beta\cdot I(cd_2) + \left\{2\cdot\left[(\alpha+1) + \left\lfloor\frac{n_f-\alpha-1}{A}\right\rfloor\cdot A\right] + [(n_f-\alpha-1)\%A]\right\}\cdot\beta\cdot I(cd_3), \quad (6.1)$$

where $n_f = \lceil\frac{C\cdot I}{B}\rceil$, $\alpha = (n_f-1)\%(F-1)$, and $\beta = \frac{B}{S}$. Function $I(x)$ is an identity function,

$$I(x) = \begin{cases} 1, & \texttt{if } x \texttt{ is true} \\ 0, & \texttt{otherwise.} \end{cases}$$

Condition $n_f \le A$ is denoted as cd_1 in Eq. (6.1); cd_2 represents condition $A < n_f \le A^2 - A + 1$ and $\alpha == 0$; cd_3 represents condition $A < n_f \le A^2 - A + 1$ and $\alpha \ne 0$. Letting T_3 be the time needed to merge one record, the merge stage takes $F_n(\lceil\frac{C\cdot I}{B}\rceil, A, B, S)\cdot T_3$ time. Therefore, the overall running time of a mapper can be written as:

$$T_m = \frac{C}{L}\cdot T_1 + \frac{C}{L}\cdot T_2 + F_n\left(\left\lceil\frac{C\cdot I}{B}\right\rceil, A, B, S\right)\cdot T_3. \quad (6.2)$$

We then model the running time of a reducer. During the fetch stage, a reducer fetches partitions from all mappers. Assuming fetching one unit of data takes T_4 time, a reducer needs $\frac{D\cdot I\cdot T_4}{R}$ time to finish the fetch stage (assuming mapper outputs are evenly distributed to all reducers). As the partitions from mappers accumulate on disk, the reducer launches a background thread to merge them into a single data stream. Similar to the merge stage in a mapper, the merge stage in a reducer might incur multiple rounds. There are $\lceil\frac{D}{C}\rceil$ partitions from all mappers (as there are $\lceil\frac{D}{C}\rceil$ mappers). Therefore, the merge stage takes $F_n(\lceil\frac{D}{C}\rceil, A, \frac{D\cdot I}{R}, S)\cdot T_3$ time. Letting T_5 be the time of processing one key–value pair using the aggregation operation and writing the corresponding output to HDFS, the aggregate stage takes $\frac{D\cdot I\cdot T_5}{R\cdot S}$ time. Therefore, the overall running time of a reducer can be represented as:

$$T_r = \frac{D\cdot I}{R}\cdot T_4 + F_n\left(\left\lceil\frac{D}{C}\right\rceil, A, \frac{D\cdot I}{R}, S\right)\cdot T_3 + \frac{D\cdot I}{R\cdot S}\cdot T_5. \quad (6.3)$$

Let T_{ov} be the constant overhead of initializing and cleaning up mappers and reducers in the cluster. On a cluster with W_m mapper slots and W_r reducer slots, the running time of a Re-Org task can be represented as:

$$T_{total} = \left\lceil\frac{D}{C\cdot W_m}\right\rceil\cdot(T_m + T_{ov}) + T_r. \quad (6.4)$$

Note that Eq. (6.4) assumes the cluster has enough reducer slots to finish all reducers in one pass, i.e., $R < W_r$.

6.3. Model verification

We have run a series of experiments to measure the values of $T_1 \sim T_5$ and T_{ov} when testing a Re-Org task in a 10-node cluster. By applying those values into Eq. (6.4), we can calculate the running time of the Re-Org task predicted by our model. The predicted running time is compared against the actual running time to verify the accuracy of the model. We show the comparison result in Fig. 6, where we vary the input chunk size and the mapper buffer size in our experiments.

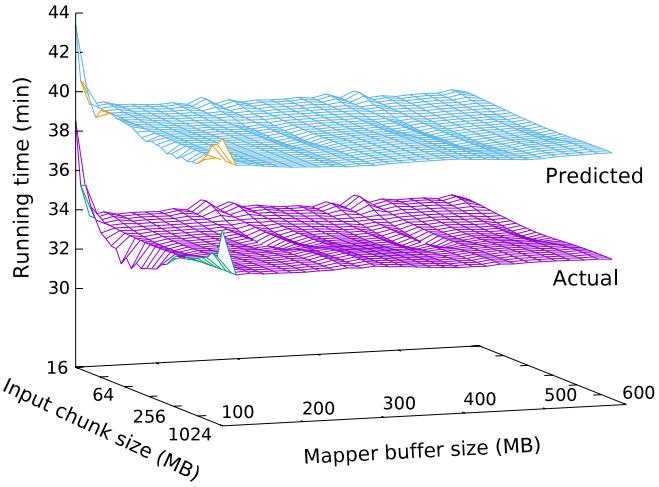


Fig. 6. Comparing the actual running time and the predicted running time.

As shown in Fig. 6, the measured running time and the predicted time exhibit very similar trends. The actual time is smaller than the predicted time because our model does not consider the overlap between mapper execution and reducer execution for the sake of simplicity. The overlap depends on many volatile factors, such as task scheduling and I/O request serving.

6.4. Time complexity comparisons

Now we can exploit the presented model to illustrate the advantage of our GOM-Hadoop framework. We here compare the time of running a Re-Org task on GOM-Hadoop with that of running it on vanilla Hadoop. The execution of a Re-Org task on Hadoop can be divided into six stages similar to those of GOM-Hadoop. Our analysis will be based on comparing the corresponding stages in two frameworks.

Parse and fetch stages: Our framework and vanilla Hadoop have no difference in the parse stage and the fetch stage. Both need to parse the same amount of key–value pairs from the input chunk in the parse stage. In the merge stage (in either mapper or reducer), our framework is more efficient than vanilla Hadoop. GOM-Hadoop utilizes the property of the input dataset to maximally avoid comparing secondary keys.

Merge stage: Assuming T_3' is the average time for vanilla Hadoop to merge one record, we must have $T_3' > T_3$. As a result, GOM-Hadoop saves $F_n(\lceil\frac{C\cdot I}{B}\rceil, A, B, S)\cdot(T_3' - T_3)$ time in the mapper's merge stage, and saves $F_n(\lceil\frac{D}{C}\rceil, A, \frac{D\cdot I}{R}, S)\cdot(T_3' - T_3)$ time in the reducer's merge stage.

Aggregate stage: Vanilla Hadoop has to call a customized *grouping comparator* to group key–value pairs. The grouping comparator needs to deserialize the composite key, which slows down the aggregate stage. Assuming T_5' is the time for vanilla Hadoop to process one key–value pair using the aggregation operation and to write the output, GOM-Hadoop saves $\frac{D\cdot I}{R\cdot S}\cdot(T_5' - T_5)$ time in the aggregate stage.

Group-order stage: The group-order stage is the key of GOM-Hadoop and therefore we present the analysis of this stage in detail. Table 2 shows the notations in our analysis. We treat sorting the key–value pairs to generate spill files as the group-order stage in vanilla Hadoop.

The time a mapper spent on sorting operations for generating spill files often dominates its group-order stage. Hence, we focus on analyzing the time spent on sorting in this stage. To simplify the analysis, we assume that each hash function generates evenly

Table 2
Notations in analyzing the time complexity.

Notation	Description
n	Number of key–value pairs in a mapper's buffer
r	Number of partitions (i.e., number of reducers)
k	Number of unique keys in a mapper's buffer
b	Number of slots
u	Number of unique keys that have the same hashcode in each slot
t_1	Time of one number (hashcode) comparison
t_2	Time of one primary key comparison
t_3	Time of one secondary key comparison

distributed hash values; key–value pairs have evenly distributed keys.

When a mapper generates a spill file, GOM-Hadoop compares items only when sorting keys of key–value pairs in each small hash table to obtain an order of its entries. We leverage quicksort to sort keys. We first compare their hashcodes (hc_2), which are integer numbers, and then compare the (primary) keys themselves only if their hashcodes are the same. There are $r \cdot b$ small hash tables in total, and each one has $\frac{n}{r \cdot b}$ keys. Sorting hashcodes of keys in one small hash table takes $t_1 \cdot \frac{n}{r \cdot b} \cdot \log \frac{n}{r \cdot b}$ time. Hence, it spends $t_1 \cdot n \cdot \log \frac{n}{r \cdot b}$ time to compare hashcodes. Similarly, GOM-Hadoop needs $t_2 \cdot n \cdot \log \frac{n}{r \cdot b \cdot u}$ time on comparing the keys. Let T_o be the sorting time when a mapper of GOM-Hadoop generates a spill file. We have

$$T_o = t_1 \cdot n \cdot \log \frac{n}{r \cdot b} + t_2 \cdot n \cdot \log \frac{n}{r \cdot b \cdot u}. \tag{6.5}$$

We then analyze the sorting time when a mapper of vanilla Hadoop generates a spill file. To sort all key–value pairs in a spill file, Hadoop uses quicksort to sort them in the following way. It first compares their partition numbers. If the numbers are the same, their primary keys are compared. If their primary keys are still the same, it compares their secondary keys. Therefore, their partition number must be compared. Consequently, the time spent on comparing partition number is $t_1 \cdot n \cdot \log n$. There are r partitions, and each one has $\frac{n}{r}$ key–value pairs. Primary keys in two key–value pairs are compared only when they have the same partition numbers. Hence, comparing primary keys within the same partition takes $t_2 \cdot \frac{n}{r} \cdot \log \frac{n}{r}$ time. Because there are r partitions, the total time spent on comparing primary keys is $t_2 \cdot n \cdot \log \frac{n}{r}$. Similarly, the time of comparing secondary keys is $t_3 \cdot n \cdot \log \frac{n}{r \cdot k}$. Therefore, when a mapper generates a spill file, the time needed by vanilla Hadoop in sorting, T_h, is

$$T_h = t_1 \cdot n \cdot \log n + t_2 \cdot n \cdot \log \frac{n}{r} + t_3 \cdot n \cdot \log \frac{n}{r \cdot k}. \tag{6.6}$$

Recalling the analysis in Section 6.2, we know that there are $\frac{B}{S}$ key–value pairs in a mapper's buffer ($n = \frac{B}{S}$), a mapper generates $\lceil \frac{C \cdot I}{B} \rceil$ spill files in total, and $R = r$. Therefore, compared to vanilla Hadoop, a mapper in our framework saves $\lceil \frac{C \cdot I}{B} \rceil \cdot (T_h - T_o)$ time in the group-order stage.

In conclusion, as compared to vanilla Hadoop, GOM-Hadoop needs T_{save} less time in executing a Re-Org task. We have

$$\begin{aligned} T_{save} = & \left\lceil \frac{D}{C \cdot W_m} \right\rceil \cdot \left[F_n \left(\left\lceil \frac{C \cdot I}{B} \right\rceil, A, B, S \right) \cdot (T_3' - T_3) \right. \\ & \left. + \left\lceil \frac{C \cdot I}{B} \right\rceil \cdot (T_h - T_o) \right] + F_n \left(\left\lceil \frac{D}{C} \right\rceil, A, \frac{D \cdot I}{R}, S \right) \\ & \cdot (T_3' - T_3) + \frac{D \cdot I}{R \cdot S} \cdot (T_5' - T_5). \end{aligned} \tag{6.7}$$

From Eq. (6.7) we can see that the time saved by GOM-Hadoop is approximately linear to the input dataset size (D). Besides, GOM-Hadoop shows its advantages over vanilla Hadoop more prominently when the group order and the merge stages dominate the running time of a job, because T_3 and T_5 are much smaller than T_3' and T_5', respectively. We believe that quite a lot of workloads have that characteristics. For instance, when the size of key–value pairs produced by a mapper is much smaller than the input records (i.e., I is small), only a small amount of data must be shuffled and output. In this case, the fetch phase and the aggregate phase take only a small portion of the entire running time. Also, because the parse phase is usually quite short, the group-order and the merge stages are the dominating parts in the job running time.

7. Evaluation

We have conducted extensive experiments to evaluate the performance of GOM-Hadoop. Our experiments show that the proposed framework constantly outperforms start-of-the-art vanilla Hadoop implementations in executing Re-Org tasks.

7.1. Experiment setup

We use both a local cluster and a large-scale cluster on Amazon EC2 [3] to evaluate our framework. The local cluster has 10 machines, each of which has 16 3.33 GHz Intel Xeon cores, 16 GB of RAM, and 1 TB of hard disk. Each machine is configured to have 12 slots for mappers and 4 slots for reducers. The Amazon cluster consists of 100 medium instances. Each instance has one core, 3.7 GB of RAM, and 400 GB of hard disk. Two mapper slots and one reducer slot are configured for each instance.

Two real-world datasets are used in our evaluation, denoted as *click stream data* and *network flow data*, respectively. The former one is the well-known 116 GB click stream data collected at the World Cup 1998 website [32]. The latter one is network flow metadata extracted from the traffic of a US national-wide mobile network. A commercial tool is used to sniff packets from the mobile network's backbone and extract semantic metadata of network flows. The metadata of a flow has multiple lines of records. A unique numeric ID is included in those lines to associate them with the same flow. The records of flows are outputted into text log files as they are generated, and each record has a timestamp to indicate when the record is generated. The records of different flows interleave with each other, and they are always ordered by their timestamps. The size of the network flow data is 1.35 TB.

We run two types of Re-Org tasks with distinct input–output characteristics on these two datasets. In the first type of task, the ratio between the input data size and the output data size is close to 1. In other words, this type of task does not produce summary of the input data. Rather, it just re-organizes the input data in a certain way. Examples of this type of task include user sessionization and flow construction. User sessionization groups clicks by user and then divides the click stream of each user into sessions by a timeout threshold (e.g., 5 min). Flow construction groups all records (metadata) of the same flow together and ensures the records are sorted on their timestamps. The other type of Re-Org task has low ratio of its output size to its input size, such as computing session duration of all sessions and figuring out the killer page of each session (i.e., last accessed page of a session). This type of task aggregates a lot of information so as to produce a summary of the input data.

We compare GOM-Hadoop with vanilla Hadoop's secondary sort implementation in realizing the aforementioned two types of Re-Org tasks. The basic MapReduce approach is not evaluated here since it is not scalable (as illustrated in Section 3.2).

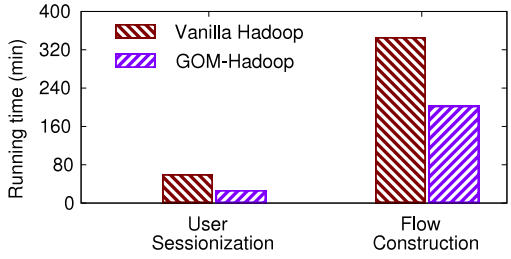
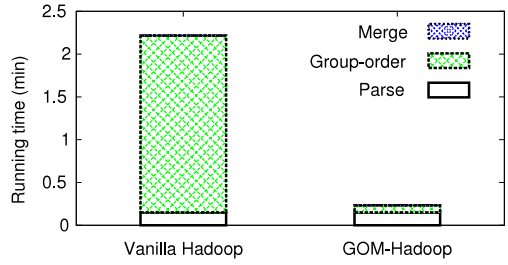
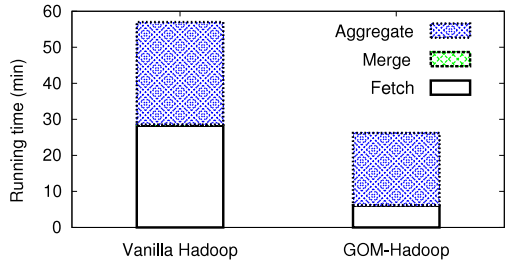


Fig. 7. Running times of high output ratio tasks.



(a) Map timings.



(b) Reduce timings.

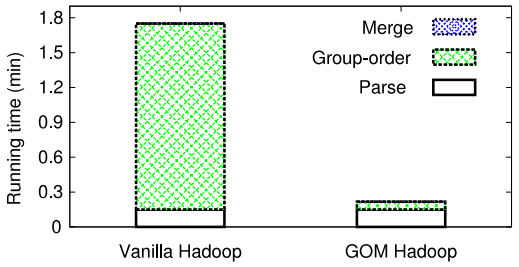
Fig. 8. Timings for user sessionization.

7.2. RE-ORG tasks with high output ratio

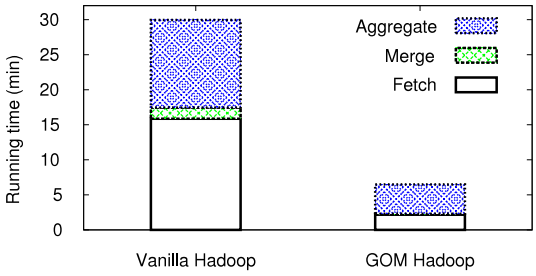
We first test running Re-Org tasks with high output ratio. Fig. 7 shows the end-to-end times of running user sessionization on the click stream data and running flow construction on the network flow data. We can see that GOM-Hadoop runs 2.2 and 1.7 times faster than vanilla Hadoop, respectively.

We also use the user sessionization task as an example to study the running time of each individual stage in a mapper and a reducer, to provide a microscopic view of GOM-Hadoop's performance benefits. Fig. 8 presents the measurement results.

As shown in Fig. 8(a), GOM-Hadoop is 25× faster than vanilla Hadoop in the group-order stage. In addition, by setting parameters (to allow each mapper to generate only one spill file), neither GOM-Hadoop nor vanilla Hadoop needs mapper side merging. As shown in Fig. 8(b), GOM-Hadoop is faster than vanilla Hadoop in all the three stages of the reducer execution. The performance benefit of the fetch stage is due to two reasons. First, the fetch stage completes after all mappers finish, and thus the slow mapper execution of vanilla Hadoop slows down its fetch stage. Secondly, part of the merge stage is counted into the fetch stage because they overlap with each other, and GOM-Hadoop has a more efficient merging scheme. The merge stage of GOM-Hadoop is faster because it leverages a more efficient merging scheme (relative order based merge). As shown in Section 3.3, vanilla Hadoop has to use a grouping comparator to group key–value pairs, which needs to deserialize the composite key and thus slows down its aggregate stage.



(a) Map timings.



(b) Reduce timings.

Fig. 9. Timings for session duration.

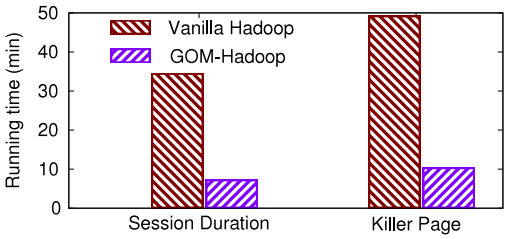


Fig. 10. Running times of low output ratio tasks.

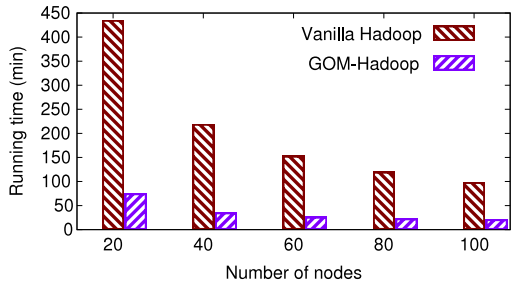
7.3. RE-ORG tasks with low output ratio

We then use the click stream data to evaluate two Re-Org tasks with low output ratio: (1) computing session duration, and (2) finding killer pages. This type of Re-Org task shuffles and outputs much less data. Hence, the sorting (of vanilla Hadoop) would contribute more to the running time of a task. Accordingly, the benefits of our framework would be more prominent. Fig. 9 plots the running time of each stage in mappers and reducers. Compared to the user sessionization task (with high output ratio), the computing session duration task running in vanilla Hadoop spends a little less time in the group-order phase and spends much less time in the fetch phase and the aggregate phase. As shown in Fig. 10, GOM-Hadoop achieves 5× speedup as compared to vanilla Hadoop on the session duration task and obtains similar performance on the killer page task.

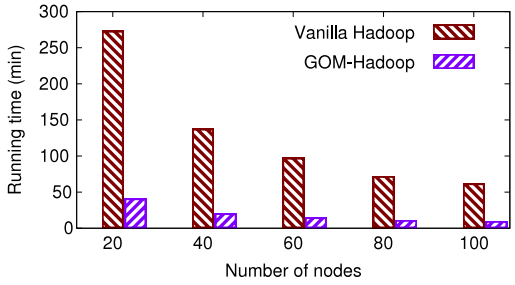
7.4. Scalability

We further evaluate GOM-Hadoop on the large-scale Amazon cluster to test its scalability. Both types of Re-Org tasks, the user sessionization task and the session duration task, are tested on the click stream data. The performance of vanilla Hadoop is evaluated as a reference point.

Fig. 11 presents the job running time as the number of nodes (instances) being used increases from 20 to 100. We can see that the running times of both tasks decrease smoothly as the number of nodes increases. GOM-Hadoop constantly outperforms vanilla Hadoop regardless of the cluster size. When the cluster

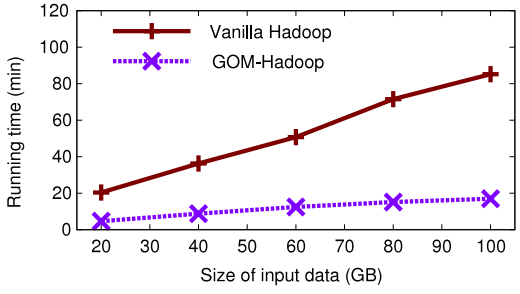


(a) User sessionization.

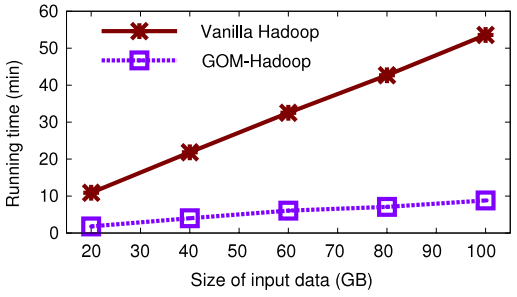


(b) Session duration.

Fig. 11. Varying number of nodes. Running times are on a log scale.



(a) User sessionization.



(b) Session duration.

Fig. 12. Varying size of the input dataset.

size is 100-node, GOM-Hadoop achieves 4.7× speedup on the user sessionization task and 6.3× speedup on the session duration task. More importantly, comparing with that on the local cluster, GOM-Hadoop on the Amazon cluster demonstrates even better speedup. This is because that the Amazon instances are less powerful (with slower CPUs) than the local machines and thus the deficiency of the CPU-intensive sorting in vanilla Hadoop is more serious.

We also test how GOM-Hadoop scales with increasing size of the input data. We choose data subsets of different sizes from the click stream data and perform both the user sessionization task and the session duration task. As presented in Fig. 12, when realized by GOM-Hadoop, the running time of both tasks increases linearly as the size of the input data increases. Moreover, running both tasks on GOM-Hadoop is always faster than that on vanilla Hadoop. We can conclude that GOM-Hadoop has good scaling performance in processing large scale datasets.

8. Related work

MapReduce [12] has attracted lots of attention over the past several years as a practical large-scale data processing framework. A series of studies have extended the basic idea of MapReduce and/or optimized its performance for various applications. Using hash techniques to improve the performance of MapReduce has been explored in [21,33]. However, running Re-Org workload directly on those frameworks incurs significant inevitable overhead, since neither of them utilizes the ordering property of the input datasets. The most relevant work to ours is the one-pass analytics platform proposed by Li et al. in [21]. They exploit multi-level hashing to group data by keys. Their platform does not preserve the original relative order of records in each group. Hence, implementing a Re-Org task on their platform requires the user to write customized code to sort records on their secondary keys, as in the basic MapReduce approach, and thus it is not a scalable solution. Map-Reduce-Merge [33] adopts hash techniques to implement efficient joining. Each reducer maintains a hash table so as to merge partitions from mappers. However, it does not preserve the original relative order of records in the merged group either.

Efficiently processing event log files has been studied in [23,10], which have different focuses with this paper. In-situ MapReduce [23] aims to process data on location without uploading it to a centralized place. TiMR [10] builds a time-oriented data processing system on top of MapReduce so as to support queries in behavioral targeting advertisement. In contrast, our work focuses on how to efficiently support relative order-preserving based grouping tasks by utilizing the ordering property of the input datasets.

9. Conclusion

We observed that a large class of analytical workload in big data analytics can be considered as relative order-preserving based grouping (Re-Org) tasks on ordered datasets. Additionally, we identified that the popular big data analytics tool, MapReduce/Hadoop, cannot efficiently realize such tasks because of its internal sort–merge mechanism. Therefore, this paper presents a scalable distributed framework, GOM-Hadoop, for efficiently executing Re-Org tasks. GOM-Hadoop adopts a novel group-order–merge mechanism to efficiently exploit the ordering property of the datasets. GOM-Hadoop is built by extending Hadoop to incorporate the proposed mechanism as an alternative shuffle mechanism. The performance of GOM-Hadoop was evaluated via both formal modeling and extensive experiments on real-world datasets. The experimental results show that GOM-Hadoop can be up to 6.3× faster than Hadoop in executing Re-Org tasks.

Acknowledgments

The authors would like to thank the editor, Dr. Per Stenström, and the anonymous reviewers for their comments and suggestions. This work is partially supported by National Science Foundation grants CNS-1217284 and CCF-1018114. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] M. AbdelBaky, H. Kim, I. Rodero, M. Parashar, Accelerating MapReduce analytics using cometcloud, in: CLOUD'12, 2012.

[2] F. Ahmad, S. Lee, M. Thottethodi, T.N. Vijaykumar, Mapreduce with communication overlap (MaRCO), J. Parallel Distrib. Comput. 73 (5) (2013) 608–620.

[3] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

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

[5] N. Basher, A. Mahanti, A. Mahanti, C. Williamson, M. Arlitt, A comparative analysis of web and peer-to-peer traffic, in: Proceedings of the 17th International Conference on World Wide Web, WWW'08, 2008, pp. 287–296.

[6] B. Berendt, B. Mobasher, M. Nakagawa, M. Spiliopoulou, The impact of site structure and user environment on session reconstruction in web usage analysis, in: WEBKDD 2002—Mining Web Data for Discovering Usage Patterns and Profiles, 2002, pp. 159–179.

[7] J. Berlińska, M. Drozdowski, Scheduling divisible MapReduce computations, J. Parallel Distrib. Comput. 71 (3) (2011) 450–459.

[8] M.J.A. Berry, G.S. Linoff, Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management, John Wiley & Sons, 2004.

[9] X. Bu, J. Rao, C.-z. Xu, Interference and locality-aware task scheduling for MapReduce applications in virtual clusters, in: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, 2013, pp. 227–238.

[10] B. Chandramouli, J. Goldstein, S. Duan, Temporal analytics on big data for web advertising, in: Proceedings of the 28th IEEE International Conference on Data Engineering, ICDE'12, 2012, pp. 90–101.

[11] Cisco, Introduction to Cisco IOS NetFlow—A Technical Overview. http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html.

[12] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in: The 6th Symposium on Operating Systems Design and Implementation, OSDI'04, USENIX Association, 2004, pp. 107–113.

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, G.C. Fox, Twister: a runtime for iterative MapReduce, in: The ACM International Symposium on High Performance Distributed Computing, HPDC, Chicago, 2010, pp. 810–818.

[14] T. Fawcett, F. Provost, Activity monitoring: Noticing interesting changes in behavior, in: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'99, 1999, pp. 53–62.

[15] I. Goiri, K. Le, T.D. Nguyen, J. Guitart, J. Torres, R. Bianchini, GreenHadoop: Leveraging green energy in data-processing frameworks, in: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys'12, 2012, pp. 57–70.

[16] R. Gu, X. Yang, J. Yan, Y. Sun, B. Wang, C. Yuan, Y. Huang, SHadoop: Improving MapReduce performance by optimizing job execution mechanism in Hadoop clusters, J. Parallel Distrib. Comput. 74 (3) (2014) 2166–2179.

[17] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, Z. Xu, RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems, in: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE'11, 2011, pp. 1199–1208.

[18] J. Hu, H.-J. Zeng, H. Li, C. Niu, Z. Chen, Demographic prediction based on user's browsing behavior, in: Proceedings of the 16th International Conference on World Wide Web, WWW'07, 2007, pp. 151–160.

[19] K. Kambatla, N. Rapolu, S. Jagannathan, A. Grama, Asynchronous algorithms in MapReduce, in: Proceedings of CLUSTER'10, 2010, pp. 245–254.

[20] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, X. Zhang, YSmart: Yet another SQL-to-MapReduce translator, in: Proceedings of the 2011 31st International Conference on Distributed Computing Systems, ICDCS'11, 2011, pp. 25–36.

[21] B. Li, E. Mazur, Y. Diao, A. McGregor, P. Shenoy, A platform for scalable one-pass analytics using MapReduce, in: Proceedings of ACM SIGMOD International Conference on Management of Data, SIGMOD'11, 2011, pp. 985–996.

[22] L. Liu, J. Yin, L. Gao, Efficient social network data query processing on MapReduce, in: Proceedings of the 5th ACM Workshop on HotPlanet, HotPlanet'13, 2013, pp. 27–32.

[23] D. Logothetis, C. Trezzo, K.C. Webb, K. Yocum, In-situ MapReduce for log processing, in: Proceedings of 2011 USENIX Annual Technical Conference, ATC'11, 2011, pp. 9:1–9:15.

[24] P. Lu, Y.C. Lee, C. Wang, B.B. Zhou, J. Chen, A.Y. Zomaya, Workload characteristic oriented scheduler for MapReduce, in: Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems, ICPADS'12, 2012, pp. 156–163.

[25] B. Palanisamy, A. Singh, L. Liu, B. Jain, Purlieus: Locality-aware resource allocation for MapReduce in a cloud, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11, 2011, pp. 58:1–58:11.

[26] A. Rau-Chaplin, B. Varghese, D. Wilson, Z. Yao, N. Zeh, QuPARA: Query-driven large-scale portfolio aggregate risk analysis on MapReduce, in: 2013 IEEE International Conference on Big Data, 2013, pp. 703–709.

[27] A. Sarje, S. Aluru, A MapReduce style framework for computations on trees, in: Proceedings of the 2010 39th International Conference on Parallel Processing, ICPP'10, 2010, pp. 343–352.

[28] A. Soule, K. Salamatia, N. Taft, R. Emilion, K. Papagiannaki, Flow classification by histograms: Or how to go on safari in the Internet, in: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS'04/Performance'04, 2004, pp. 49–60.

[29] S. Su, X. Cheng, L. Gao, J. Yin, Co-ClusterD: A distributed framework for data co-clustering with sequential updates, in: 2013 IEEE 13th International Conference on Data Mining, ICDM, 2013, pp. 1193–1198.

[30] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, D. Chen, G-Hadoop: MapReduce across distributed data centers for data-intensive computing, Future Gener. Comput. Syst. 29 (3) (2013) 739–750.

[31] T. White, Hadoop: The Definitive Guide, first ed., O'Reilly Media, Inc., 2009.

[32] World Cup 1998 Dataset. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[33] H.-c. Yang, A. Dasdan, R.-L. Hsiao, D.S. Parker, Map-Reduce-Merge: simplified relational data processing on large clusters, in: Proceedings of ACM SIGMOD International Conference on Management of Data, SIGMOD'07, 2007, pp. 1029–1040.

[34] J. Yin, L. Gao, Z.M. Zhang, Scalable nonnegative matrix factorization with block-wise updates, in: ECML/PKDD'14, 2014, pp. 337–352.

[35] J. Yin, Y. Liao, M. Baldi, L. Gao, A. Nucci, A scalable distributed framework for efficient analytics on ordered datasets, in: IEEE/ACM International Conference on Utility and Cloud Computing, UCC'13, 2013.

[36] J. Yin, Y. Liao, M. Baldi, L. Gao, A. Nucci, Efficient analytics on ordered datasets using MapReduce, in: The ACM International Symposium on High Performance Distributed Computing, HPDC, HPDC'13, 2013, pp. 125–126.

[37] J. Yin, Y. Zhang, L. Gao, Accelerating expectation–maximization algorithms with frequent updates, in: Proceedings of IEEE Cluster Computing, CLUSTER'12, 2012, pp. 275–283.

[38] Y. Zhang, Q. Gao, L. Gao, C. Wang, iMapReduce: A distributed computing framework for iterative computation, in: Proceedings of IPDPSW'11: The First International Workshop on Data Intensive Computing in the Clouds, 2011, pp. 1112–1121.

[39] Y. Zhang, Q. Gao, L. Gao, C. Wang, PrIter: A distributed framework for prioritized iterative computations, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC'11, 2011, pp. 13:1–13:14.

[40] J. Zhao, L. Wang, J. Tao, J. Chen, W. Sun, R. Ranjan, J. Kolodziej, A. Streit, D. Georgakopoulos, A security framework in G-Hadoop for big data computing across distributed cloud data centres, J. Comput. System Sci. 80 (5) (2014) 994–1007.

Jiangtao Yin received his B.E. degree from Beijing Institute of Technology, China, in 2006, and his M.E. degree from Beijing University of Posts and Telecommunications, China, in 2009. Since fall 2009, he has been working as a research assistant in University of Massachusetts at Amherst, where he is currently pursuing the Ph.D. degree in the department of electrical and computer engineering. His current research interests include distributed systems, large-scale data mining, and cloud computing.

Yong Liao received his B.S. degree in 2001 from University of Science and Technology of China, his M.S. from Chinese Academy of Sciences in 2004, and Ph.D. from University of Massachusetts at Amherst in 2010. After graduating from UMass Amherst, he was Software Engineer at Microsoft Redmond, Washington; Senior Member of Technical Staff in the CTO group of Narus Inc. (a Boeing company), Sunnyvale, California; and Senior Principal Data Scientist at Symantec Corp., Mountain View, California. Currently he is a Principal Research Scientist at China Academy of Electronics and Information Technology, Beijing, China. His research interests include big data framework and analytics, network forensics, security and privacy.

Mario Baldi is Associate Professor at Politecnico di Torino and Principal Architect at Cisco Systems. He was Data Scientist Director at Symantec Corp., Mountain View, CA, Principal Member of Technical Staff with the CTO Office at Narus, Inc., Sunnyvale, Principal Architect at Embrane, Inc., Santa Clara; Vice Dean of the PoliTong Sino-Italian Campus at Tongji University, Shanghai; Vice President for Protocol Architecture at Synchrodyne Networks, Inc., New York. Through his research, teaching and professional activities, Mario Baldi has built considerable knowledge and expertise in big data analytics, next generation network data analysis, internetworking, high performance switching, optical networking, quality of service, multimedia networking, trust in distributed software execution, and computer networks in general.

Lixin Gao is a professor of Electrical and Computer Engineering at the University of Massachusetts at Amherst. She received her Ph.D. degree in computer science from the University of Massachusetts at Amherst. Her research interests include social networks, Internet routing, network virtualization, and cloud computing. Between May 1999 and January 2000, she was a visiting researcher at AT&T Research Labs and DIMACS. She was an Alfred P. Sloan Fellow between 2003-2005 and received an NSF CAREER Award in 1999. She won the best paper award from IEEE INFOCOM 2010, and the test-of-time award in ACM SIGMETRICS 2010. Her paper in ACM Cloud Computing 2011 was honored with ''Paper of Distinction''. She received the Chancellor's Award for Outstanding Accomplishment in Research and Creative Activity in 2010, and is a fellow of IEEE and ACM.

Antonio Nucci is the Chief Technology Officer of Smart Services, CSTG at Cisco Systems leading the design of the next generation Cisco Service Delivery Platform and driving disruptive technology innovation in the areas of Mobility, IOE, Security and High-performance distributed analytics. Before joining Cisco, Antonio was Vice President and Chief Data Scientist at Symantec Corporation, executive at the Boeing Company and Chief Technology Officer of Narus Inc. In his career Antonio has published over 100 ACM/IEEE papers, has been awarded 46 U.S. patents, and has co-authored a definitive textbook titled, ''Design, Measurement and Management of Large-Scale IP Networks: Bridging the gap between Theory and Practice'' published by Cambridge University Press. He serves as a technical advisor of several venture capital firms in Silicon Valley advising on emerging technologies and trends. Antonio obtained his Ph.D. and Master's Degrees in electrical engineering from Politecnico di Torino, Italy.