A Scalable Distributed Framework for Efficient Analytics on Ordered Datasets

Jiangtao Yin	Yong Liao	Mario Baldi	Lixin Gao	Antonio Nucci
UMASS Amherst	*Narus Inc.*	*Narus Inc.*	*UMASS Amherst*	*Narus Inc.*
jyin@ecs.umass.edu	*yliao@narus.com*	*mbaldi@narus.com*	*lgao@ecs.umass.edu*	*anucci@narus.com*

***Abstract*—One of the most common datasets used by many corporations to gain business intelligence is event log files. Oftentimes, the records in event log files are temporally ordered, and need to be grouped by user ID with the temporal ordering preserved to facilitate mining user behaviors. This kind of analytical workload, here referred to as RElative Order-pReserving based Grouping (RE-ORG), is quite common in big data analytics. Using MapReduce/Hadoop for executing RE-ORG tasks on ordered datasets is not efficient due to its internal sort-merge mechanism. In this paper, we propose a distributed framework that adopts an efficient group-order-merge mechanism to provide faster execution of RE-ORG tasks. We demonstrate the advantage of our framework 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.**

I. 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 perabytes 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 certain feature together and impose the temporal ordering on the data points in the same group. Such a processing is usually a vital step in many important analytical jobs, including:

- User sessionization [8], [14]: widely used in recommendation systems and personalized web services.
- Flow construction [7], [22]: utilized in traffic engineering and network security.
- Customer statement generation [9], [13]: 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, etc., and a timestamp representing 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 according to 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 sorted by their secondary keys. For such an input dataset, we define the *RElative Order-pReserving based Grouping*, or RE-ORG, as a processing that generates a set of output data points, where each one is generated from 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 sorted by their secondary keys.

Since a RE-ORG task usually involves a large quantity of data, parallelizing its execution is desirable. MapReduce [11] has emerged as a popular scalable framework for data intensive computation. It provides a simple programming model where a user can 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 his/her custom code or by instrumenting MapReduce to do that.

In this paper, we propose a *group-order-merge* mechanism[1] 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

[1]The high level idea of the group-order-merge mechanism is also illustrated in our poster paper [25].

records are merged in parallel by different worker nodes so as to yield record groups, where each group has all records with the same primary key and the records are ordered on their secondary keys.

We have built a distributed framework for supporting the group-order-merge mechanism by extending Hadoop [2], the most popular open-source implementation of MapReduce. Our framework is referred to as *Group-Order-Merge Hadoop (GOM Hadoop)*. 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 [1]. The evaluation results show that our GOM Hadoop can achieve up to 6.3x speedup over vanilla Hadoop.

The rest of the paper is organized as following. Section II defines the problem targeted by this paper formally. Section III briefly surveys how the problem is solved by using existing techniques. Section IV presents our scheme for efficiently executing RE-ORG tasks in a distributed environment. The framework for supporting the proposed scheme is presented in Section V. Section VI presents the evaluation results. Section VII surveys related work. Section VIII concludes this paper.

II. 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.

A. 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* [8], [14]. Usually, a session consists of a user's temporally ordered clicks and is considered to be finished if the user has no clicking for some time duration (e.g., 5 minutes). 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 [3] 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, such as malicious behavior detection, require to correlate 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 [9], [13].

B. 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 treated 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 Figure 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.

```
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
```

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

We define the *RElative Order-pReserving based Grouping* mechanism, or RE-ORG, as a processing that generates a set of output data points, where each one is generated from a *group of records* from the input dataset. For instance, in the RE-ORG task that splits users' clicks into sessions, each session is one output data point.

Now we give the formal definition of RE-ORG mechanism. The input dataset of a RE-ORG task is a list of records:

$$R_{in} = [r_0, r_1, ..., r_n]. \tag{1}$$

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

$$s_i \preceq s_j \iff i < j, \forall r_i, r_j \in R_{in}. \tag{2}$$

The output of a RE-ORG task is a set of data points, which is represented as

$$R_{out} = \{\hat{r}_0, \hat{r}_1, ..., \hat{r}_m\}. \tag{3}$$

Each output data point $\hat{r}_u$ in R_{out} is a generated (i.e, via an aggregation operation) from a group of input records associated with the same primary key:

$$\hat{r}_u = \mathrm{F}(G_u), \tag{4}$$

where $G_u = [r_{u_0}, r_{u_1}, ..., r_{u_k}]$ is a group of input records such that

$$p_{u_i} = p_{u_j} \iff r_{u_i} \in G_u \text{ and } r_{u_j} \in G_u. \tag{5}$$

To derive its result, operation $\mathrm{F}()$ has to parse the group of records in the order defined by the secondary key:

$$s_{u_i} \preceq s_{u_j} \iff i < j, \forall r_{u_i}, r_{u_j} \in G_u. \tag{6}$$

Take click stream analysis for example: supposing we are interested in the time a user spends on the website for each visit, which is measured as the duration of a session, the operation $\mathrm{F}()$ must traverse a user's clicks (G_u) in the order of timestamp (s_{u_i}) so as to figure out when a session ends.

III. Existing MapReduce Support in Realizing 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.

A. MapReduce/Hadoop

MapReduce, a popular distributed programming model for processing large-scale datasets in a cluster of commodity machines, has gained considerable attention over the past several years [5], [6], [12], [15], [18]–[21], [26]–[28].

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 key-value pairs. Each 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 a mapper. When the buffer is full, the Hadoop framework performs a sorting on key-value pairs with partition numbers (using quicksort by default). The sorting orders those key-value pairs first on their partition numbers and then on 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 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 associated 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.

B. Basic MapReduce Approach

For input dataset $R_{in} = [r_0, r_1, ..., r_n]$, 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 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 key-value pairs denoted as $[\{p_i, (s_i, v_i)\}, 0 \leq i \leq x]$. The Hadoop framework 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 it receives a large number of values for a given key, a reducer may run out of memory. Therefore, this approach is not scalable.

C. 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* [23]. 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 key-value pair as well. Within a 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 *composite key*. Then, by customizing the key comparator, the user instruments the Hadoop framework 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 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.

IV. 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.

A. 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 records have in the input dataset (sorted by the secondary key).

However, it is challenging to realize a RE-ORG task in a distributed environment. 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. Therefore, 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 challenges in realizing distributed RE-ORG.

B. Logic Workflow of Our Solution

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 Figure 2.

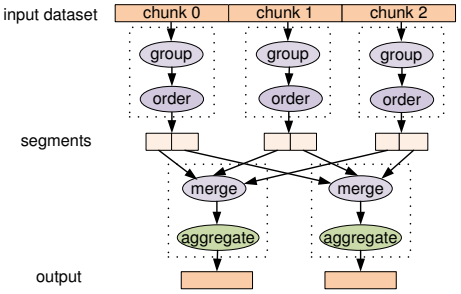


Figure 2. Basic workflow of the proposed group-order-merge mechanism.

The input dataset is first split into *chunks*. Each data chunk is assigned to a logic worker to process[2]. 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.

Workers in the merge phase process the segments produced in the order phase, merge the records (from different segments) 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 output.

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 V.

C. Grouping via Hashing Primary Key

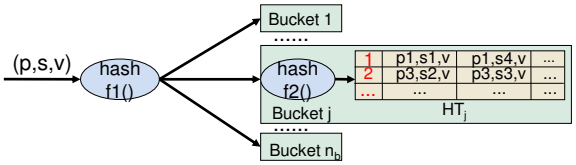


Figure 3. Grouping records by their primary keys via hashing.

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 Figure 3, disperses records into a fixed number of (n_b) 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, we assume hash function $f2()$ can perform 1-to-1 mapping so that an entry in the logic 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 input chunk can fit in a worker's memory for the sake of easy explanation. In Section V-A, we will show that our mechanism is actually implemented with bounded memory usage.

D. Hash Assisted Ordering

The order phase streams out those n_b hash tables populated in the group phase to local disk. Each hash table is

[2]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.

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 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 it finishes generating its ordered segments, the worker sends them out 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 V-B, we will present our implementation which can yield ordered segments with minimal primary key comparisons.

E. Relative Order based Merge

The merge phase merges multiple segments into one final stream, where the records (from different segments) 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 segments and move the "smallest" one (the record with the "smallest" primary key or the record with the "smallest" secondary key when more than one records have "smallest" primary keys) into the stream.

V. Framework Implementation

In this section, we present the implementation of our distributed framework (GOM Hadoop) for efficiently executing RE-ORG tasks. Our framework extends Hadoop to incorporate the group-order-merge scheme described in Section IV as an alternative shuffle mechanism[3] to its default sort-merge mechanism. In this way, our framework can also execute ordinary Hadoop jobs with negligible overhead. Hadoop is selected as the basis for the implementation because a RE-ORG task maps very well to the MapReduce programming model and Hadoop is the most popular open-source MapReduce implementation. The popularity of Hadoop stems from its good performance in handling failures and its capability of scaling to a large number of worker nodes. The prototype implementation of our GOM Hadoop is based on Hadoop version 1.0.3.

[3]In this paper, 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.

A. 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 the 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 IV-C to group the key-value pairs' indexes (by hashing key-value pairs but storing their indexes). Note that the buffers will not accept new key-value pairs/indexes until their contents are spilled out to local disk. Hence, there is no memory overflow problem. 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 Figure 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 of (n_s) 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 do mapping between the indexes' corresponding keys and its entries. Each entry stores indexes for the same key, and the hash table uses separate chaining to resolve hash collision. An index is always appended to the tail of the corresponding entry.

B. 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 IV-D. 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 keys corresponding to the indexes in them. 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 indexes in the entry.

In 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.

C. 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 (`io.sort.factor`), multiple merging rounds might occur. Since each spill file already has the key-value pair 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 phase 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 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 do this, 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 counters can be used to determine the order of tuples (key-value pairs) from different spill files. A smaller counter means a "smaller" secondary key. Moreover, since it is a small number, 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 the 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), i.e., partition i will be sent to reducer i.

D. 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 cannot control a mapper to read which chunk of the input dataset, 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.

VI. Evaluation

In this section, we present the performance evaluation of the proposed framework.

A. Experiment Setup

We build both a local cluster and a large-scale cluster on Amazon EC2 [1] to evaluate our framework. The local cluster consists of 10 machines, and each one has 16 3.33GHz Intel Xeon cores, 16GB of RAM, and 1TB 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, and each instance has one core, 3.7GB of RAM, and 400GB of hard disk. Each instance has 2 slots for mappers and 1 slot for reducers.

Two real-world datasets are used. One (*click stream data*) is the well known 116GB click stream data related to the World Cup 1998 [4]. The other one (*network flow data*) 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 is written into a text file as 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 the text file as they are generated, and each record has a timestamp to indicate when the record is generated. The records of different flows can interleave with each other, and they are always ordered by their timestamps. The size of the network flow log data is 1.35TB.

Two types of RE-ORG tasks are evaluated on these two datasets. One type of tasks has high ratio of its output size to its input data size. In other words, this type of tasks does not produce summary of the input data. Rather, this type of tasks just re-organizes the input data in a certain way. Examples

of this type of tasks 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 minutes). 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 tasks 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 (last accessed page of a session). This type of tasks aggregates a lot of information so as to produce a summary of the input data, and thus has low ratio output.

We compare our GOM Hadoop with vanilla Hadoop's secondary sort implementation when 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 III-B).

B. RE-ORG Tasks with High Output Ratio

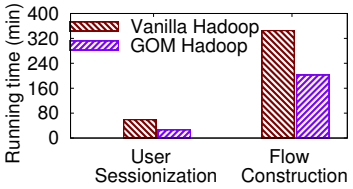


Figure 4. Running times of high output ratio tasks.

We first evaluate RE-ORG tasks with high output ratio: user sessionization on the click stream data, and flow construction on the network flow data. From Figure 4, we can see that comparing with vanilla Hadoop, GOM Hadoop achieves 2.2x speedup on user sessionization and 1.7x speedup on flow construction.

C. RE-ORG Tasks with Low Output Ratio

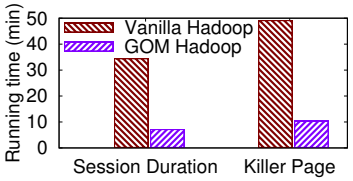


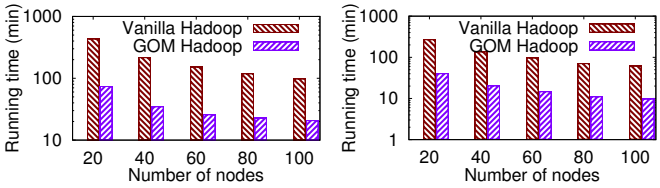
Figure 5. Running times of low output ratio tasks.

We then evaluate RE-ORG tasks with low output ratio: computing session duration and figuring out killer pages on the click stream data. Compared to the above type, this type of RE-ORG task shuffles and outputs much less data. Hence, the sorting (of vanilla Hadoop) would take more portion of the running time of a task. Accordingly, the benefits of our framework would be more obvious. As shown in Figure 5, comparing with vanilla Hadoop, GOM Hadoop achieves 5x speedup on the session duration task, and obtains similar performance on the killer page task.

D. Scalability

To validate the scalability of our GOM Hadoop, we evaluate it on the large-scale Amazon cluster. 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 also evaluated to be a reference point.

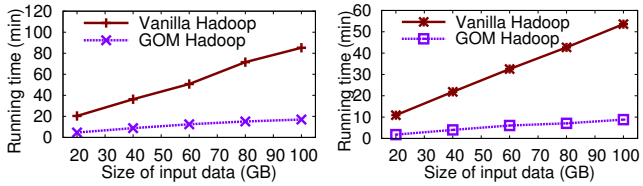
Figure 6a and Figure 6b plot the performance as the number of nodes (instances) being used in the cluster increases from 20 to 100. From the figures, we can see that the running times of both tasks decrease smoothly as the number of nodes increases. In addition, GOM Hadoop outperforms vanilla Hadoop with any number of nodes in the cluster. For example, when the number of nodes is 100, GOM Hadoop achieves 4.7x speedup on the user sessionization task and 6.3x speedup on the session duration task. Comparing with that on the local cluster, GOM Hadoop on the Amazon cluster obtains a even better speedup. This is because that the Amazon instance is less powerful (with a slower CPU) than the local machine and thus the deficiency of the CPU-intensive sorting in vanilla Hadoop is more serious.



(a) User sessionization (b) Session duration

Figure 6. Varying number of nodes. Running times are on a log scale.

We also measure how GOM Hadoop scales with increasing size of the input data. We choose subset data of different sizes from the click stream data, and perform both the user sessionization task and the session duration task. As presented in Figure 7a and Figure 7b, when realized by GOM Hadoop, the running times of both tasks increase linearly as the size of the input data increases. Moreover, the running time of either task on GOM Hadoop is always shorter than that on vanilla Hadoop. Therefore, our GOM Hadoop demonstrates a good scaling performance.



(a) User sessionization (b) Session duration

Figure 7. Varying size of the input dataset.

VII. Related Work

MapReduce [11] has gained considerable attention over the past several years as a large-scale data processing framework. A series of studies have extended the basic idea of MapReduce and optimized its performance in various applications. For example, several studies focus on improving MapReduce for iterative computations [12], [26]–[28].

Using hash techniques to improve the performance of MapReduce has been explored in [16], [24]. However,

running RE-ORG directly on those frameworks incurs significant overhead that could be avoided. The most relevant work to ours is the one-pass analytics platform proposed by Li et al. [16]. They utilize multi-level hashing to group data by key. Unfortunately, their platform does not preserve the original relative order of records in each group. Implementing a RE-ORG task on their platform requires the user to write code for sorting records based on their secondary keys, as in the basic MapReduce approach, and thus it is not a scalable solution. Map-Reduce-Merge [24] adopts hash techniques to realize hash join. 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, and has to maintain an on-disk (slow) hash table when the data size exceeds memory capacity.

Efficiently processing event log files has been studied in [10], [17], which have different focuses with this paper. In-situ MapReduce [17] 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 the behavioral targeting advertisement. In contrast, our work focuses on how to efficiently support relative order-preserving based grouping tasks.

VIII. 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. Aslo, 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 for efficiently executing RE-ORG tasks. Our framework adopts a novel group-order-merge mechanism to efficiently utilize the ordering property of the ordered datasets. We evaluated its performance via extensive experiments. The evaluation results show that our framework can be up to 6.3 times faster than the current Hadoop implementation in executing RE-ORG tasks on real-world datasets.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments and suggestions. This work is partially supported by NSF grants CNS-1217284 and CCF-1018114.

REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.
[2] Apache Hadoop. http://hadoop.apache.org/.
[3] Introduction to Cisco IOS NetFlow - A Technical Overview. http://goo.gl/7VzDI.
[4] World Cup 1998 Dataset. http://goo.gl/2UqlS.
[5] M. AbdelBaky, H. Kim, I. Rodero, and M. Parashar. Accelerating MapReduce analytics using cometcloud. In *CLOUD '12*, 2012.
[6] A. S. Balkir, I. Foster, and A. Rzhetsky. A distributed look-up architecture for text mining applications using MapReduce. In *SC '11*, 2011.
[7] N. Basher, A. Mahanti, A. Mahanti, C. Williamson, and M. Arlitt. A comparative analysis of web and peer-to-peer traffic. In *WWW '08*, 2008.
[8] B. Berendt, B. Mobasher, M. Nakagawa, and M. Spiliopoulou. The impact of site structure and user environment on session reconstruction in web usage analysis. In *WEBKDD '02*, 2002.
[9] M. J. A. Berry and G. S. Linoff. *Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management*. John Wiley & Sons, 2004.
[10] B. Chandramouli, J. Goldstein, and S. Duan. Temporal analytics on big data for web advertising. In *ICDE '12*, 2012.
[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, 2004.
[12] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *HPDC '10*, 2010.
[13] T. Fawcett and F. Provost. Activity monitoring: noticing interesting changes in behavior. In *KDD '99*, 1999.
[14] J. Hu, H.-J. Zeng, H. Li, C. Niu, and Z. Chen. Demographic prediction based on user's browsing behavior. In *WWW '07*, 2007.
[15] C. Jin and R. Buyya. MapReduce programming model for .NET-based cloud computing. In *Euro-Par '09*, 2009.
[16] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using MapReduce. In *SIGMOD '11*, 2011.
[17] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ MapReduce for log processing. In *USENIXATC '11*, 2011.
[18] P. Lu, Y. C. Lee, C. Wang, B. B. Zhou, J. Chen, and A. Zomaya. Workload characteristic oriented scheduler for MapReduce. In *ICPADS '12*, 2012.
[19] C. Mi, Q. Chen, and T. Liu. An efficient cross-match implementation based on directed join algorithm in MapReduce. In *UCC '11*, 2011.
[20] P. Nguyen, T. Simon, M. Halem, D. Chapman, and Q. Le. A hybrid scheduling algorithm for data intensive workloads in a MapReduce environment. In *UCC '12*, 2012.
[21] B. Sharma, T. Wood, and C. R. Das. HybridMR: A hierarchical MapReduce scheduler for hybrid data centers. In *ICDCS '13*, 2013.
[22] A. Soule, K. Salamatia, N. Taft, R. Emilion, and K. Papagiannaki. Flow classification by histograms: or how to go on safari in the Internet. In *SIGMETRICS/Performance '04*, 2004.
[23] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.
[24] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *SIGMOD '07*, 2007.
[25] J. Yin, Y. Liao, M. Baldi, L. Gao, and A. Nucci. Efficient analytics on ordered datasets using MapReduce. In *HPDC '13*, 2013.
[26] J. Yin, Y. Zhang, and L. Gao. Accelerating expectation-maximization algorithms with frequent updates. In *CLUSTER '12*, 2012.
[27] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: A distributed computing framework for iterative computation. In *IPDPSW '11*, 2011.
[28] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. In *SOCC '11*, 2011.