

Data Partitioning for Minimizing Transferred Data in MapReduce

Miguel Liroz-Gistau, Reza Akbarinia, Divyakant Agrawal, Esther Pacitti, Patrick Valduriez

▶ **To cite this version:**

Miguel Liroz-Gistau, Reza Akbarinia, Divyakant Agrawal, Esther Pacitti, Patrick Valduriez. Data Partitioning for Minimizing Transferred Data in MapReduce. Globe, Aug 2013, Prague, Czech Republic. pp.1-12, 10.1007/978-3-642-40053-7_1 . lirmm-00879527

HAL Id: lirmm-00879527

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00879527

Submitted on 4 Nov 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Data Partitioning for Minimizing Transferred Data in MapReduce

Miguel Liroz-Gistau[1], Reza Akbarinia[1], Divyakant Agrawal[2], Esther Pacitti[3], and Patrick Valduriez[1]

[1] INRIA & LIRMM, Montpellier, France
{Miguel.Liroz_Gistau, Reza.Akbarinia, Patrick.Valduriez}@inria.fr
[2] University of California, Santa Barbara
agrawal@cs.ucsb.edu
[3] University Montpellier 2, INRIA & LIRMM, Montpellier, France
Esther.Pacitti@lirmm.fr

Abstract. Reducing data transfer in MapReduce's shuffle phase is very important because it increases data locality of reduce tasks, and thus decreases the overhead of job executions. In the literature, several optimizations have been proposed to reduce data transfer between mappers and reducers. Nevertheless, all these approaches are limited by how intermediate key-value pairs are distributed over map outputs. In this paper, we address the problem of high data transfers in MapReduce, and propose a technique that repartitions tuples of the input datasets, and thereby optimizes the distribution of key-values over mappers, and increases the data locality in reduce tasks. Our approach captures the relationships between input tuples and intermediate keys by monitoring the execution of a set of MapReduce jobs which are representative of the workload. Then, based on those relationships, it assigns input tuples to the appropriate chunks. We evaluated our approach through experimentation in a Hadoop deployment on top of Grid5000 using standard benchmarks. The results show high reduction in data transfer during the shuffle phase compared to Native Hadoop.

1 Introduction

MapReduce [4] has established itself as one of the most popular alternatives for big data processing due to its programming model simplicity and automatic management of parallel execution in clusters of machines. Initially proposed by Google to be used for indexing the web, it has been applied to a wide range of problems having to process big quantities of data, favored by the popularity of Hadoop [2], an open-source implementation. MapReduce divides the computation in two main phases, namely map and reduce, which in turn are carried out by several tasks that process the data in parallel. Between them, there is a phase, called shuffle, where the data produced by the map phase is ordered, partitioned and transferred to the appropriate machines executing the reduce phase.

MapReduce applies the principle of "moving computation towards data" and thus tries to schedule map tasks in MapReduce executions close to the input data they process, in order to maximize data locality. Data locality is desirable because it reduces the amount of data transferred through the network, and this reduces energy consumption as well as network traffic in data centers.

Recently, several optimizations have been proposed to reduce data transfer between mappers and reducers. For example, [5] and [10] try to reduce the amount of data transferred in the shuffle phase by scheduling reduce tasks close to the map tasks that produce their input. Ibrahim et al. [7] go even further and dynamically partition intermediate keys in order to balance load among reduce tasks and decrease network transfers. Nevertheless, all these approaches are limited by how intermediate key-value pairs are distributed over map outputs. If the data associated to a given intermediate key is present in all map outputs, even if we assign it to a reducer executing in the same machine, the rest of the pairs still have to be transferred.

In this paper, we propose a technique, called MR-Part, that aims at minimizing the transferred data between mappers and reducers in the shuffle phase of MapReduce. MR-Part captures the relationships between input tuples and intermediate keys by monitoring the execution of a set of MapReduce jobs which are representative of the workload. Then, based on the captured relationships, it partitions the input files, and assigns input tuples to the appropriate fragments in such a way that subsequent MapReduce jobs following the modeled workload will take full advantage of data locality in the reduce phase. In order to characterize the workload, we inject a monitoring component in the MapReduce framework that produces the required metadata. Then, another component, which is executed offline, combines the information captured for all the MapReduce jobs of the workload and partitions the input data accordingly. We have modeled the workload by means of an hypergraph, to which we apply a min-cut k-way graph partitioning algorithm to assign the tuples to the input fragments.

We implemented MR-Part in Hadoop, and evaluated it through experimentation on top of Grid5000 using standard benchmarks. The results show significant reduction in data transfer during the shuffle phase compared to Native Hadoop. They also exhibit a significant reduction in execution time when network bandwidth is limited.

This paper is organized as follows: In Section 2, we briefly describe MapReduce, and then define formally the problem we address. In Section 3, we propose MR-Part. In Section 4, we report the results of our experimental tests evaluating its efficiency. Section 5 presents the related work and Section 6 concludes.

2 Problem Definition

2.1 MapReduce Background

MapReduce is a programming model based on two primitives, $map : (K_1, V_1) \rightarrow \text{list}(K_2, V_2)$ and $reduce : (K_2, \text{list}(V_1)) \rightarrow \text{list}(K_3, V_3)$. The map function pro-

cesses key/value pairs and produces a set of intermediate/value pairs. Intermediate key/value pairs are merged and sorted based on the intermediate key k_2 and provided as input to the reduce function.

MapReduce jobs are executed over a distributed system composed of a master and a set of workers. The input is divided into several splits and assigned to map tasks. The master schedules map tasks in the workers by taking into account data locality (nodes holding the assigned input are preferred).

The output of the map tasks is divided into as many partitions as reducers are scheduled in the system. Entries with the same intermediate key k_2 should be assigned to the same partition to guarantee the correctness of the execution. All the intermediate key/value pairs of a given partition are sorted and sent to the worker where the corresponding reduce task is going to be executed. This phase is called shuffle. Default scheduling of reduce task does not take into consideration any data locality constraint. As a consequence, depending on how intermediate keys appear in the input splits and how the partitioning is done, the amount of data that has to be transferred through the network in the shuffle phase may be significant.

2.2 Problem Statement

We are given a set of MapReduce jobs which are representative of the system workload, and a set of input files. We assume that future MapReduce jobs follow similar patterns as those of the representative workload, at least in the generation of intermediate keys.

The goal of our system is to automatically partition the input files so that the amount of data that is transferred through the network in the shuffle phase is minimized in future executions. We make no assumptions about the scheduling of map and reduce tasks, and only consider intelligent partitioning of intermediate keys to reducers, e.g., as it is done in [7].

Let us formally state the problem which we address. Let the input data for a MapReduce job, job_α, be composed of a set of data items $D = \{d_1, ..., d_n\}$ and divided into a set of chunks $C = \{C_1, ..., C_p\}$. Function $loc : D \rightarrow C$ assigns data items to chunks. Let job_α be composed of $M_\alpha = \{m_1, ..., m_p\}$ map tasks and $R_\alpha = \{r_1, ..., r_q\}$ reduce tasks. We assume that each map task m_i processes chunk c_i. Let $N_\alpha = \{n_1, .., n_s\}$ be the set of machines used in the job execution; $node(t)$ represents the machine where task t is executed.

Let $I_\alpha = \{i_1, .., i_m\}$ be the set of intermediate key-value pairs produced by the map phase, such that $map(d_j) = \{i_{j_1}, ..., i_{j_t}\}$. $k(i_j)$ represents the key of intermediate pair i_j and $size(i_j)$ represents its total size in bytes. We define $output(m_i) \subseteq I_\alpha$ as the set of intermediate pairs produced by map task m_i, $output(m_i) = \bigcup_{d_j \in C_i} map(d_j)$. We also define $input(r_i) \subseteq I_\alpha$ as the set of intermediate pairs assigned to reduce task r_i. Function $part : k(I_\alpha) \rightarrow R$ assigns intermediate keys to reduce tasks.

Let i_j be an intermediate key-value pair, such that $i_j \in output(m)$ and $i_j \in input(r)$. Let $P_{i_j} \in \{0, 1\}$ be a variable that is equal to 0 if intermediate

pair i_j is produced in the same machine where it is processed by the reduce task, and 1 otherwise, i.e., $P(i_j) = 0$ iff $node(m) = node(r)$.

Let $W = \{job_1, ..., job_w\}$ be the set of jobs in the workload. Our goal is to find *loc* and *part* functions in a way in which $\sum_{job_\alpha \in W} \sum_{i_j \in I_\alpha} size(i_j)P(i_j)$ is minimized.

3 MR-Part

In this section, we propose MR-Part, a technique that by automatic partitioning of MapReduce input files allows Hadoop to take full advantage of locality-aware scheduling for reduce tasks, and to reduce significantly the amount of data transferred between map and reduce nodes during the shuffle phase. MR-Part proceeds in three main phases, as shown in Fig. 1: 1) Workload characterization, in which information about the workload is obtained from the execution of MapReduce jobs, and then combined to create a model of the workload represented as a hypergraph; 2) Repartitioning, in which a graph partitioning algorithm is applied over the hypergraph produced in the first phase, and based on the results the input files are repartitioned; 3) Scheduling, that takes advantage of the input partitioning in further executions of MapReduce jobs, and by an intelligent assignment of reduce tasks to the workers reduces the amount of data transferred in the shuffle phase. Phases 1 and 2 are executed offline over the model of the workload, so their cost is amortized over future job executions.

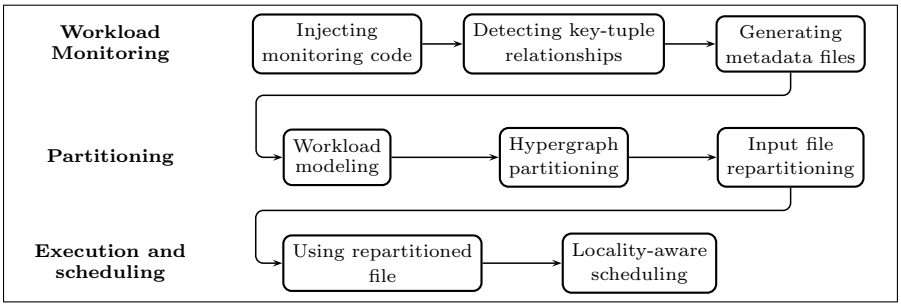


Fig. 1. MR-Part workflow scheme

3.1 Workload Characterization

In order to minimize the amount of data transferred through the network between map and reduce tasks, MR-Part tries to perform the following actions: 1) grouping all input tuples producing a given intermediate key in the same chunk and 2) assigning the key to a reduce task executing in the same node.

The first action needs to find the relationship between input tuples and intermediate keys. With that information, tuples producing the same intermediate key are co-located in the same chunk.

Monitoring We inject a monitoring component in the MapReduce framework that monitors the execution of map tasks and captures the relationship between input tuples and intermediate keys. This component is completely transparent to the user program.

The development of the monitoring component was not straightforward because the map tasks receive entries of the form (K_1, V_1), but with this information alone we are not able to uniquely identify the corresponding input tuples. However, if we always use the same `RecordReader`[4] to read the file, we can uniquely identify an input tuple by a combination of its input file name, its chunk starting offset and the position of `RecordReader` when producing the input pairs for the map task.

For each map task, the monitoring component produces a metadata file as follows. When a new input chunk is loaded, the monitoring component creates a new metadata file and writes the chunk information (file name and starting offset). Then, it initiates a record counter (rc). Whenever an input pair is read, the counter is incremented by one. Moreover, if an intermediate key k is produced, it generates a pair (k, rc). When the processing of the input chunk is finished, the monitoring component groups all key-counter pairs by their key, and for each key it stores an entry of the form $\langle k, \{rc_1, ..., rc_n\}\rangle$ in the metadata file.

Combination While executing a monitored job, all metadata is stored locally. Whenever a repartitioning is launched by the user, the information from different metadata files have to be combined in order to generate a hypergraph for each input file. The hypergraph is used for partitioning the tuples of an input file, and is generated by using the matadata files created in the monitoring phase.

A hypergraph $H = (H_V, H_E)$ is a generalization of a graph in which each hyper edge $e \in H_E$ can connect more than two vertices. In fact, a hyper edge is a subset of vertices, $e \subseteq H_V$. In our model, vertices represent input tuples and hyper edges characterize tuples producing the same intermediate key in a job.

The pseudo-code for generating the hypergraph is shown in Algorithm 1. Initially the hypergraph is empty, and new vertices and edges are added to it as the metadata files are read. The metadata of each job is processed separately. For each job, our algorithm creates a data structure T, which stores for each generated intermediate key, the set of input tuples that produce the key. For every entry in the file, the algorithm generates the corresponding tuple ids and adds them to the entry in T corresponding to the generated key. For easy id generation, we store in each metadata file, the number of input tuples processed for the associated chunk, n_i. We use the function $generateTupleID(c_i, rc) = \sum_{j=1}^{i-1} n_i + rc$ to translate record numbers into ids. After processing all metadata of a job, for each read tuple, our algorithm adds a vertex in the hypergraph (if

[4] The `RecordReader` is the component of MapReduce that parses the input and produce input key-value pairs. Normally each file format is parsed by a single `RecordReader`; therefore, using the same `RecordReader` for the same file is a common practice

it is not there). Then, for each intermediate key, it adds a hyper edge containing the set of tuples that have produced the key.

Algorithm 1: Metadata combination

```
Data: F: Input file; W: Set of jobs composing the workload
Result: H = (H_V, H_E): Hypergraph modeling the workload
begin
    H_E ← ∅; H_V ← ∅
    foreach job ∈ |W| do
        T ← ∅; K ← ∅
        foreach m_i ∈ M_job do
            md_i ← getMetadata(m_i)
            if F = getFile(md_i) then
                foreach ⟨k, {rc_1, ..., rc_n}⟩ ∈ md_i do
                    {t_1.id, ..., t_n.id} ← generateTupleID(c_i, {rc_1, ..., rc_n})
                    T[k] ← T[k] ∪ {t_1.id, ..., t_n.id}; K ← K ∪ {k}
        foreach intermediate key k ∈ K do
            H_V ← H_V ∪ T[k]; H_E ← H_E ∪ {T[k]}
```

3.2 Repartitioning

Once we have modeled the workload of each input file through a hypergraph, we apply a min-cut k-way graph partitioning algorithm. The algorithm takes as input a value k and a hypergraph, and produces k disjoint subsets of vertices minimizing the sum of the weights of the edges between vertices of different subsets. Weights can be associated to vertices, for instance to represent different sizes. We set k as the number of chunks in the input file. By using the min-cut algorithm, the tuples that are used for generating the same intermediate key are usually assigned to the same partition.

The output of the algorithm indicates the set of tuples that have to be assigned to each of the input file chunks. Then, the input file should be repartitioned using the produced assignments. However, the file repartitioning cannot be done in a straightforward manner, particularly because the chunks are created by HDFS automatically as new data is appended to a file. We create a set of temporary files, one for each partition. Then, we read the original file, and for each read tuple, the graph algorithm output indicates to which of the temporary files the tuple should be copied. Then, two strategies are possible: 1) create a set of files in one directory, one per partition, as it is done in the reduce phase of MapReduce executions and 2) write the generated files sequentially in the same file. In both cases, at the end of the process, we remove the old file and rename the new file/directory to its name. The first strategy is straightforward and instead of writing data in temporary files, it can be written directly in HDFS. The second one has the advantage of not having to deal with more files but has to deal with the following issues:

– *Unfitted partitions*: The size of partitions created by the partitioning algorithm may be different than the predefined chunk size, even if we set strict

imbalance constraints in the algorithm. To approximate the chunk limits to the end of the temporary files when written one after the other, we can modify the order in which temporary files are written. We used a greedy approach in which we select at each time the temporary file whose size, added to the total size written, approximates the most to the next chunk limit.

– *Inappropriate last chunk*: The last chunk of a file is a special case, as its size is less than the predefined chunk size. However, the graph partitioning algorithm tries to make all partitions balanced and does not support such a constraint. In order to force one of the partitions to be of the size of the last chunk, we insert a virtual tuple, $t_{virtual}$, with the weight equivalent to the empty space in the last chunk. After discarding this tuple, one of the partitions would have a size proportional to the size of the last chunk.

The repartitioning algorithm's pseudo-code is shown in Algorithm 2. In the algorithm we represent RR as the `RecordReader` used to parse the input data. We need to specify the associated `RecordWriter`, here represented as RW, that performs the inverse function as RR. The reordering of temporary files is represented by the function *reorder*().

Algorithm 2: Repartitioning

```
Data: F: Input file; H = (H_V, H_E): Hypergraph modeling the workload; k: Number of
      partitions
Result: F': The repartitioned file
begin
    H_V ← H_V ∪ t_virtual
    {P_1, ..., P_k} ← mincut(H, k)
    for i ∈ (1, ..., k) do
        create tempf_i
    foreach c_i ∈ F do
        initialize(RR, c_i); rc ← 0
        while t.data ← RR.next() do
            t.id ← generateTupleID(c_i, rc)
            p ← getPartition(t.id, {P_1, ..., P_k})
            RW.write(tempf_p, t.data)
            rc ← rc + 1
    (j_1, ..., j_k) ← reorder(tempf_1, ..., tempf_k)
    for j ∈ (j_1, ..., j_k) do
        write tempf_i in F'
```

The complexity of the algorithm is dominated by the min-cut algorithm execution. Min-cut graph partitioning is NP-Complete, however, several polynomial approximation algorithms have been developed for it. In this paper we use PaToH[5] to partition the hypergraph. In the rest of the algorithm, an inner loop is executed n times, where n is the number of tuples. *generateTupleID*() can be executed in $O(1)$ if we keep a table with n_i, the number of input tuples, for all input chunks. *getPartition*() can also be executed in $O(1)$ if we keep an array storing for each tuple the assigned partition. Thus, the rest of the algorithm is done in $O(n)$.

[5] `http://bmi.osu.edu/~umit/software.html`

3.3 Reduce Tasks Locality-Aware Scheduling

In order to take advantage of the repartitioning, we need to maximize data locality when scheduling reduce tasks. We have adapted the algorithm proposed in [7], in which each (key,node) pair is given a fairness-locality score representing the ratio between the imbalance in reducers input and data locality when key is assigned to a reducer. Each key is processed independently in a greedy algorithm. For each key, candidate nodes are sorted by their key frequency in descending order (nodes with higher key frequencies have better data locality). But instead of selecting the node with the maximum frequency, further nodes are considered if they have a better fairness-locality score. The aim of this strategy is to balance reduce inputs as much as possible. On the whole, we made the following modifications in the MapReduce framework:

- The partitioning function is changed to assign a unique partition for each intermediate key.
- Map tasks, when finished, send to the master a list with the generated intermediate keys and their frequencies. This information is included in the Heartbeat message that is sent at task completion.
- The master assigns intermediate keys to the reduce tasks relying on this information in order to maximize data locality and to achieve load balancing.

3.4 Improving Scalability

Two strategies can be taken into account to improve the scalability of the presented algorithms: 1) the number of intermediate keys; 2) the size of the generated graph.

In order to deal with a high number of intermediate keys we have created the concept of virtual reducers, VR. Instead of using intermediate keys both in the metadata and the modified partitioning function we use $k \mod VR$. Actually, this is similar to the way in which keys are assigned to reduce tasks in the original MapReduce, but in this case we set VR to a much greater number than the actual number of reducers. This decreases the amount of metadata that should be transferred to the master and the time to process the key frequencies and also the amount of edges that are generated in the hypergraph.

To reduce the number of vertices that should be processed in the graph partitioning algorithm, we perform a preparing step in which we coalesce tuples that always appear together in the edges, as they should be co-located together. The weights of the coalesced tuples would be the sum of the weights of the tuples that have been merged. This step can be performed as part of the combination algorithm that was described in Section 3.1.

4 Experimental Evaluation

In this section, we report the results of our experiments done for evaluating the performance of MR-Part. We first describe the experimental setup, and then present the results.

4.1 Set-Up

We have implemented MR-Part in Hadoop-1.0.4 and evaluated it on Grid5000 [1], a large scale infrastructure composed of different sites with several clusters of computers. In our experiments we have employed PowerEdge 1950 servers with 8 cores and 16 GB of memory.We installed Debian GNU/Linux 6.0 (squeeze) 64-bit in all nodes, and used the default parameters for Hadoop configuration.

We tested the proposed algorithm with queries from TPC-H, an ad-hoc decision support benchmark. Queries have been written in Pig [9][6], a dataflow system on top of Hadoop that translates queries into MapReduce jobs. Scale factor (which accounts for the total size of the dataset in GBs) and employed queries are specified on each specific test. After data population and data repartitioning the cluster is rebalanced in order to minimize the effects of remote transfers in the map phase.

As input data, we used `lineitem`, which is the biggest table in TPC-H dataset. In our tests, we used queries for which the shuffle phase has a significant impact in the total execution time. Particularly, we used the following queries: Q5 and Q9 that are examples of hash joins on different columns, Q7 that executes a replicated join and Q17 that executes a co-group. Note that, for any query data locality will be at least that of native Hadoop.

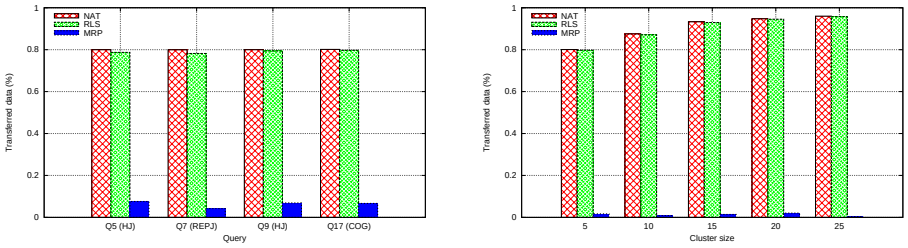
We compared the performance of MR-Part with that of native Hadoop (NAT) and reduce locality-aware scheduling (RLS) [7], which corresponds to changes explained in Section 3.3 but over the non-repartitioned dataset. We measured the percentage of transferred data in the shuffle phase for different queries and cluster sizes. We also measured the response time and shuffle time of MapReduce jobs under varying network bandwidth configurations.

4.2 Results

Transferred Data for Different Query Types. We repartitioned the dataset by using the metadata information collected from monitoring query executions. Then, we measured the amount of transferred data in the shuffled phase for our queries in the repartitioned dataset. Fig 2(a) depicts the percentage of data transferred for each of the queries on a 5 nodes cluster and scale factor of 5. As we can see, transferred data is around 80% in non repartitioned data sets (actually the data locality is always around 1 divided by the number of nodes for the original datasets), while MR-Part obtains values for transferred data below 10% for all the queries. Notice that, even with reduce locality-aware scheduling, no gain is obtained in data locality as keys are distributed in all input chunks.

Transferred Data for Different Cluster Sizes. In the next scenario, we have chosen query Q5, and measured the transferred data in the shuffle phase by varying the cluster size and input data size. Input data size has been scaled

[6] We have used the implementation provided in `http://www.cs.duke.edu/starfish/mr-apps.html`



(a) (b)

Fig. 2. Percentage of transferred data for a) different type of queries b) varying cluster and data size

depending on the cluster size, so that each node is assigned 2GB of data. Fig 2(b) shows the percentage of transferred data for the three approaches, while increasing the number of cluster nodes. As shown, with increasing the number of nodes, our approach maintains a steady data locality, but it decreases for the other approaches. Since there is no skew in key frequencies, both native Hadoop and RLS obtain data localities near 1 divided by the number of nodes. Our experiments with different data sizes for the same cluster size show no modification in the percentage of transferred data for MR-Part (the results are not shown in the paper due to space restrictions).

Response Time As shown in previous subsection, MR-Part can significantly reduce the amount of transferred data in the shuffle phase. However, its impact on response time strongly depends on the network bandwidth. In this section, we measure the effect of MR-Part on MapReduce response time by varying network bandwidth. We control point-to-point bandwidth by using Linux `tc` command line utility. We execute query Q5 on a cluster of 20 nodes with scale factor of 40 (40GB of dataset total size).

The results are shown in Fig 3. As we can see in Fig 3 (a), the slower is the network, the biggest is the impact of data locality on execution time. To show where the improvement is produced, in Fig 3 (b) we report the time spent in data shuffling. Measuring shuffle time is not straightforward since in native Hadoop it starts once 5% of map tasks have finished and proceeds in parallel while they are completed. Because of that, we represent two lines: NAT-ms that represents the time spent since the first shuffle byte is sent until this phase is completed, and NAT-os that represents the period of time where the system is only dedicated to shuffling (after last map finishes). For MR-Part only the second line has to be represented as the system has to wait for all map tasks to complete in order to schedule reduce tasks. We can observe that, while shuffle time is almost constant for MR-Part, regardless of the network conditions, it increases significantly as the network bandwidth decreases for the other alternatives. As a consequence, the response time for MR-Part is less sensitive to the network bandwidth than that of native Hadoop. For instance, for 10mbps, MR-Part executes in around 30% less time than native Hadoop.

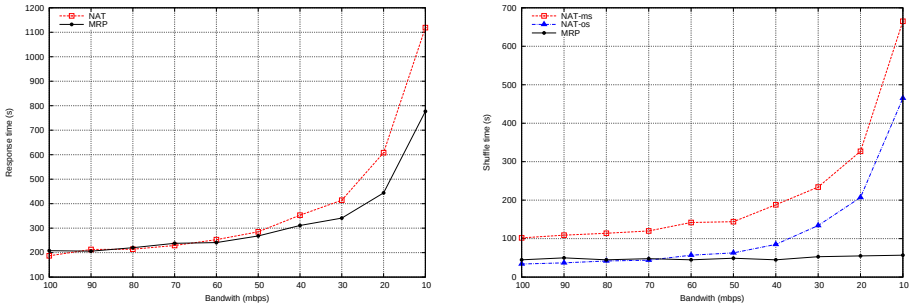


Fig. 3. Results for varying network bandwidth: a) total response time b) shuffle time

5 Related Work

Reducing data transfer in the shuffle phase is important because it may impose a significant overhead in job execution. In [12] a simulation is carried out in order to study the performance of MapReduce in different scenarios. The results show that data shuffling may take an important part of the job execution, particularly when network links are shared among different nodes belonging to a rack or a network topology. In [11], a pre-shuffling scheme is proposed to reduce data transfers in the shuffle phase. It looks over the input splits before the map phase begins and predicts the reducer the key-value pairs are partitioned into. Then, the data is assigned to a map task near the expected future reducer. Similarly, in [5], reduce tasks are assigned to the nodes that reduce the network transfers among nodes and racks. However, in this case, the decision is taken at reduce scheduling time. In [10] a set of data and VM placement techniques are proposed to improve data locality in shared cloud environments. They classify MapReduce jobs into three classes and use different placement techniques to reduce network transfers. All the mentioned jobs are limited by how the MapReduce partitioning function assigns intermediate keys to reduce tasks. In [7] this problem is addressed by assigning intermediate keys to reducers at scheduling time. However, data locality is limited by how intermediate keys are spread over all the map outputs. MR-part employs this technique as part of the reduce scheduling, but improves its efficiency by partitioning intelligently input data.

Graph and hypergraph partitioning have been used to guide data partitioning in databases and in general in parallel computing [6]. They allow to capture data relationships when no other information, e.g., the schema, is given. The work in [3, 8] uses this approach to generate a database partitioning. [3] is similar to our approach in the sense that it tries to co-locate frequently accessed data items, although it is used to avoid distributed transactions in an OLTP system.

6 Conclusions and Future Work

In this paper we proposed MR-Part, a new technique for reducing the transferred data in the MapReduce shuffle phase. MR-Part monitors a set of MapReduce

jobs constituting a workload sample and creates a workload model by means of a hypergraph. Then, using the workload model, MR-Part repartitions the input files with the objective of maximizing the data locality in the reduce phase. We have built the prototype of MR-Part in Hadoop, and tested it in Grid5000 experimental platform. Results show a significant reduction in transferred data in the shuffle phase and important improvements in response time when network bandwidth is limited.

As a possible future work we envision to perform the repartitioning in parallel. The approach used in this paper has worked flawlessly for the employed datasets, but a parallel version would be able to scale to very big inputs. This version would need to use parallel graph partitioning libraries, such as Zoltan.

Acknowledgments. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several universities as well as other funding bodies (see https://www.grid5000.fr).

References

1. Grid 5000 project. `https://www.grid5000.fr/mediawiki/index.php`
2. Hadoop. `http://hadoop.apache.org`
3. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment 3(1), 48–57 (2010)
4. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI. pp. 137–150. USENIX Association (2004)
5. Hammoud, M., Rehman, M.S., Sakr, M.F.: Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In: IEEE CLOUD. pp. 49–58. IEEE (2012)
6. Hendrickson, B., Kolda, T.G.: Graph partitioning models for parallel computing. Parallel Computing 26(12), 1519–1534 (2000)
7. Ibrahim, S., Jin, H., Lu, L., Wu, S., He, B., Qi, L.: LEEN: Locality/fairness-aware key partitioning for mapreduce in the cloud. In: Cloud Computing, Second International Conference, CloudCom 2010, November 30 - December 3, 2010, Indianapolis, Indiana, USA, Proceedings. pp. 17–24 (2010)
8. Liu, D.R., Shekhar, S.: Partitioning similarity graphs: a framework for declustering problems. Information Systems 21(6), 475–496 (1996)
9. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD Conference. pp. 1099–1110. ACM (2008)
10. Palanisamy, B., Singh, A., Liu, L., Jain, B.: Purlieus: locality-aware resource allocation for mapreduce in a cloud. In: Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011. p. 58 (2011)
11. Seo, S., Jang, I., Woo, K., Kim, I., Kim, J.S., Maeng, S.: HPMR: Prefetching and pre-shuffling in shared mapreduce computation environment. In: CLUSTER. pp. 1–8. IEEE (2009)
12. Wang, G., Butt, A.R., Pandey, P., Gupta, K.: A simulation approach to evaluating design decisions in mapreduce setups. In: MASCOTS. pp. 1–11. IEEE (2009)