Dependency-Aware Data Locality for MapReduce

Xiaoyi Fan, Xiaoqiang Ma, Jiangchuan Liu
School of Computing Science
Simon Fraser University, Burnaby, Canada
{*xiaoyif, xma10, jcliu*}*@cs.sfu.ca*

Dan Li
Department of Computer Science and Technology
Tsinghua University, Beijing, China
tolidan@tsinghua.edu.cn

***Abstract*—Recent years have witnessed the prevalence of MapReduce-based systems, e.g., the Apache Hadoop, in large-scale distributed data processing. Fetching data from remote servers across multiple network switches is known to be costly. Hence, it is highly desirable to co-locate computation with data. State-of-the-art popularity-based replication achieves *data locality* through replicating popular files and spreading the replicas over multiple servers. While working well for independent files, they can store highly dependent files in different servers, resulting in excessive remote data accesses/exchanges and consequently prolonging the job completion time.**

In this paper, we develop DALM (Dependency-Aware Locality for MapReduce), a novel replication strategy for general real-world input data that can be highly skewed and dependent. DALM accommodates data-dependency in a data-locality framework that comprehensively weights such key factors as popularity and storage budget. We extensively evaluate DALM through both simulations and real-world implementations, and have compared with state-of-the-art solutions, including the Hadoop system and the popularity-based Scarlett. The results show that DALM can significantly improve data locality for different inputs. For a popular iterative graph processing application on Hadoop, our prototype implementation of DALM reduces the remote data access and job completion time by 34.3% and 9.4%, respectively.

I. Introduction

The emergence of MapReduce as a convenient computation tool for data-intensive applications has greatly changed the landscape of large-scale distributed data processing [1]. Harnessing the power of large clusters of tens of thousands of servers with high fault-tolerance, such practical MapReduce implementations as the open-source Apache Hadoop system[1] have become the preferred choice in both academia and industry in this era of big data computation at terabyte- and even petabyte-scales.

A typical MapReduce workflow transforms an input pair to a list of intermediate key-value pairs (*the mapping stage*) and the intermediate values for the distinct keys are computed and then merged to form the final result (*the reduce stage*) [1]. This effectively distributes the computation workload to a cluster of servers; yet the data are to be dispatched too and the intermediate results are to be collected and aggregated. Given the massive data volume and the relatively scarce bandwidth resources (especially for the clusters with high over-provisioning ratio), fetching data from remote servers across multiple network switches can be costly. As such, it is highly desirable to co-locate computation with data, making them as close as possible. Real-world systems, e.g., Google's MapReduce and Hadoop, have attempted to achieve better *data locality* through replicating each file block on three servers, so that two of them are within the same rack and the remaining one is in a different rack. This simple uniform replication reduces cross-server traffic as well as job completion time for inputs of uniform popularity, but is known to be ineffective with skewed input [2], [3]. It has been observed that in certain real-word inputs, the number of accesses to popular files can be ten or even one hundred times more than that to less popular files, and the highly popular files thus still experience severe contention with massive concurrent accesses.

There have been a series of works on alleviating *hot spots* through popularity-based replication. Representatives include Scarlett [2], DARE [4] and PACMan [5], all of which seek to smartly place more replicas for popular blocks/files. Using the spare disc or memory for these extra replicas, the overall completion time of data-intensive jobs can be reduced. The inherent relations among the input data, however, have yet to be considered. In particular, it is known that many of the real-world data inherently exhibit strong *dependency*. For example, Facebook relies on the Hadoop distributed file system (HDFS) to store its user data, which, in a volume over 15 petabytes, preserves diverse social relations [6]. A sample social network graph with four communities is shown in Figure 1. Here files A and B have very strong dependency (as compared to files A and C as well as B and C), since their users are in the same community, thus being friends to each other with higher probability. As such, accesses to these two files are highly correlated. The dependency can be further aggravated during mapping and reducing: many jobs involve iterative steps, and in each iteration, specific tasks need to exchange the output of the current step with each other before going to the next iteration; in other words, the outputs become highly dependent.

A basic popularity-based replication strategy would store files with strong dependency in different servers. Such placements can incur frequent remote data accesses, which

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

978-1-4799-5063-8/14 $31.00 © 2014 IEEE
DOI 10.1109/CLOUD.2014.62

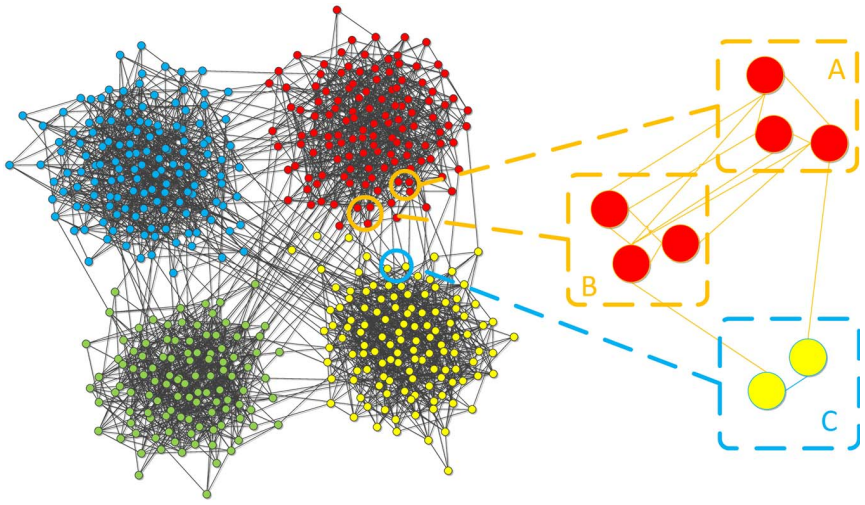

Figure 1. A social network graph with four communities

not only introduces excessive cross-server traffic but also prolong the overall job completion time. In this paper, we develop DALM (Dependency-Aware Locality for MapReduce), a novel replication strategy for general real-world input data that can be highly skewed and dependent. DALM accommodates data-dependency in a data-locality framework that comprehensively weights such key factors as popularity and storage budget. DALM consists of two major modules. The first is to calculate the replication factor of each file based on its popularity, and we demonstrate this task can be efficiently completed through a Sequential Minimal Optimization (SMO) technique [7]. Second, we develop a modified k-medoids algorithm for replica placement (where k is the number of servers) that minimizes the cross-server traffic in the presence of dependency. This algorithm decides which server to store each replica, such that the replicas of highly dependent files are stored as close as possible while not exceeding the storage budget and preventing potential hotspots. The end result is a reduced overall network traffic and consequently the job completion time.

We have extensively evaluated DALM through both simulations and real-world implementations, and have compared with state-of-the-art solutions, including the Hadoop system and the popularity-based Scarlett approach. The results have shown that the DALM's replication strategy can significantly improve data locality for different inputs. For a popular iterative graph processing system, Apache Giraph[2] on Hadoop, our prototype implementation of DALM reduces the remote data access and job completion time by 34.3% and 9.4%, respectively, as compared with Scarlett. For larger-scale multi-tenancy systems, more savings could be envisioned given that they are more susceptible to data dependency.

The rest of this paper is organized as follows. We present our motivation in Section 2, highlighting the impact of data locality. In Section 3 we formulate the problem and describe the design of our proposed replication strategy. In Section 4 we provide the evaluation results and in Section 5 we discuss practical issues on prototype implementation and deployment. Section 6 reviews the related work and Section 6 concludes this paper.

[2]Apache Giraph, http://giraph.apache.org.

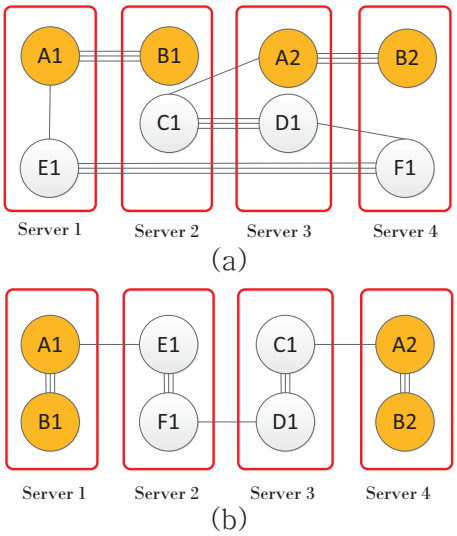

Figure 2. An example of different replication strategies based on (a) data popularity only; (b) both data popularity and dependency. The number of edges between nodes denotes the level of dependency.

II. Motivation

The popularity-based replication strategy works well if the original data-intensive job can be evenly distributed and each computation node can work in parallel. In practice, ultra-high parallelization is often hindered by data dependency

among files. Consider Figure 2, where files A and B are popular, and C, D, E, F are not. A basic popularity-based strategy creates two replicas for the popular files, namely A1, A2 for file A, and B1, B2 for file B, and one for each of the remaining files. The replicas of the popular files can then be evenly distributed to four servers, each accompanying a replica of an unpopular file (see Fig. 2(a)). This leads to a balanced and highly parallelled workloads if the servers do not have frequent data exchanges. Now assume there is strong data dependency within each of the file pairs (A,B), (C,D) and (E,F), weak dependency within (A,C), (A,E) and (D,F), and no dependency otherwise. Cross-server traffic will then become inevitable. For example, since files A and B have very strong mutual dependency, there will be frequent communications between servers 1 and 2, so will be 3 and 4. On the other hand, if we place the data with strong dependency in close vicinity, as shown in Figure 2(b), the cross-server traffic can be largely reduced.

Such dependency occurs in many real-world applications. For example, many jobs involve iterative steps, and in each iteration, specific tasks need to exchange the output of the current step with each other before going to the next iteration [8]. These tasks will keep waiting until all the necessary data exchanges are completed. Consider the PageRank algorithm [9]. Millions of websites and the hyperlinks among them compose a huge graph, which is divided into a large number of subgraphs for MapReduce. The ranking-computing task is usually conducted on a cluster with hundreds and even thousands of servers. Since each server only deals with a subset of the websites, to compute the rank of all the websites, the servers storing dependent data need to exchange the intermediate results in each iteration. Another example is to count the hops between a user pair in an online social network—a frequently used function for such social network analysis as finding the closeness of two users in Facebook or Linkedin. The shortest path calculation in a big social network will involve a number of files since each single file generally stores a small portion of the topology information only.

It is worth noting that a dependency-aware strategy may result in less-balanced workload of individual servers, as shown in Figure 2(b). The side effect however is minimal: (1) the popular files still have more replicas, which remain to be stored on different servers, thereby mitigating the hotspot problem too; (2) for the servers storing popular files, e.g., servers 1 and 4, most of the data I/O requests can be serviced locally, resulting in higher I/O throughput and thus lower CPU idle time. In other words, a good dependency-aware strategy should strive to localize data access as much as possible while avoiding potential hotspots through replicating popular files.

Table I
SUMMARY OF NOTATIONS

p_i	Popularity of file i
s_i	Size of file i
d_{ij}	Dependency between files i and j
c_j	Storage capacity of server j
P_j	Aggregate popularity of server j
$\bar{P}$	Average of aggregated file popularity of all servers
η	Threshold of deviation of P_j from $\bar{P}$
r_i	Replication factor of file f_i
r_L	Lower bound of replication factor
r_U	Upper bound of replication factor
ρ_i^k	Placement of the k_{th} replica of file i
δ	Quota of storage capacity for extra replicas
S	Aggregate files size of servers
R	Aggregate replicas
D	File dependency matrix, where the element d_{ij} refers to the level of dependency between files i and j

III. DEPENDENCY-AWARE REPLICATION: SYSTEM DESIGN

In this section, we formulate the problem of dependency-aware data replication. We then propose our replication strategy. Table I summarizes the notations.

A. Problem Formulation

Consider that there are n different files, and that for file i, denote its the popularity by p_i and its size by s_i. The value of element d_{ij} of matrix D refers to the dependency between files i and j. In the first step, we need to decide the replication factor, or how many replicas that each file has, which is denoted by r_i. Then we need to place these replicas on m independent servers, where for each server j, denote the storage space by c_j. We use ρ_i^k to represent the k_{th} replica of file i. For a replica placement strategy, $\rho_i^k = j$ indicates that the k_{th} replica of file i is stored in server j.

In the following we introduce the constraints in the considered problem.

To ensure that the total file size on any server does not exceed the storage capacity of the server, we must have:

$$\sum_{i=1}^{n}\sum_{k=1}^{r_i} I_j(\rho_i^k)s_i \leq c_j \tag{1}$$

where I_j is the indicator function such that $I_j(\rho_i^k)$ is equal to 1 iff $\rho_i^k = j$ holds, and 0 otherwise.

In off-the-shelf MapReduce-based systems, e.g., Hadoop, all files have the same replication factor, which is three by default and can be tuned by users. Yet in our considered problem, the replication factor of each file can vary from a lower bound (r_L) to an upper bound (r_U).

Naturally, the extra replicas beyond the lower bound call for additional storage space. A user-set parameter δ controls the percentage of the total storage capacity reserved for extra replicas, which trades off the data locality and storage efficiency. Let $S = \sum_{1 \leq i} s_i r_L$ be the total storage capacity

of storing r_L replicas for each file. Then the replication factor should satisfy the following constraints:

$$\sum_{i=1}^{n}(r_i - r_L)s_i \leq \delta S \tag{2}$$

$$r_L \leq r_i \leq r_U, \textit{ for } i = 1, ..., n \tag{3}$$

The basic idea of the popularity-based replication approach is to adapt the replication factor of each file according to the file popularity in order to alleviate the hotspot problem and improve resource utilization. Let P_j be the aggregated popularity of server j, which is the sum of the popularity of all the replicas stored in this server, and $\bar{P}$ be the average of the aggregated file popularity of all servers. Further, the replica placement should prevent any single server from hosting too many popular files. Hence we have the following constraints,

$$P_j = \sum_{i=1}^{n}\sum_{k=1}^{r_i} I_j(\rho_i^k)\frac{p_i}{r_i} \leq (1+\eta)\bar{P}, \textit{ for } j = 1, ..., m \tag{4}$$

where η is the threshold of deviation of P_j from $\bar{P}$. We divide p_i by r_i, assuming that the accesses to file i evenly spread over all of its replicas.

In our considered problem, we aim at minimizing the remote access. In the current Hadoop system, the computation node will access the replicas as close as possible. Similar to the popularity-based strategy [2], [4], we consider *file* as the replica granularity and our dependency-aware replication strategy is then divided into two successive steps. The first is to calculate the replication factor of each file based on its popularity and second is to decide which server to store each replica, such that the replicas of highly dependent files are stored as close as possible while not exceeding the storage budget and preventing potential hotspots. The details are as follows.

B. Replication Strategy

We compute the replication factor r_i of file i based on the file popularity and size, as well as the storage budget δ by solving the following optimization problem.

$$\begin{aligned} \min \quad & \theta_p = ||\frac{\mathbf{r}}{R} - \mathbf{p}||^2 \\ s.t. \quad & \sum_{i=1}^{n}(r_i - r_L)s_i \leq \delta S \\ & r_L \leq r_i \leq r_U, \textit{ for } i = 1, .., n \end{aligned} \tag{5}$$

where $\mathbf{r} = (r_1, ..., r_n)$, $\mathbf{p} = (p1, ..., p_n)$, $R = \sum_{i=1}^{n} r_i$, .

Here, the main idea is to let the replica density of file i, namely $\frac{r_i}{R}$, to be as close to the file popularity p_i as possible.

To solve it, we start from the Lagrangian function:

$$\begin{aligned} f(\mathbf{r}) = ||\frac{\mathbf{r}}{R} - \mathbf{p}||^2 + \alpha[\sum_{i=1}^{n}(r_i - r_L)s_i - \delta S] \\ + \sum_{i=1}^{n}\beta_i(r_L - r_i) + \sum_{i=1}^{n}\gamma_i(r_i - r_U) \end{aligned} \tag{6}$$

where α, β and γ are the Lagrange multipliers. Now the Lagrangian dual becomes

$$f_{LD} = \max_{\substack{\alpha\geq 0,\\ \beta_i \geq 0,\\ \gamma_i \geq 0}} \min_{r_L \leq r_i \leq r_U} f(\mathbf{r})$$

Since the objective function (5) is convex and the constraints are linear, according to the Slater's condition [10], we have

$$\frac{\partial f(\mathbf{r})}{\partial r_i} = 2(\frac{r_i}{R} - p_i)\frac{1}{R} + \alpha s_i - \beta_i + \gamma_i = 0$$

Namely,

$$r_i = Rp_i - \frac{R^2}{2}(\alpha s_i - \beta_i + \gamma_i) \tag{7}$$

Combining Equations (6) and (7), the Lagrangian Dual problem becomes:

$$\begin{aligned} \max \quad & g(\alpha, \boldsymbol{\beta}, \boldsymbol{\gamma}) = \sum_{i=1}^{n}(\frac{R(\alpha s_i - \beta_i + \gamma_i)}{2})^2 + \\ & \alpha[\sum_{i=1}^{n}(Rp_i + \frac{R^2}{2}(\alpha s_i - \beta_i + \gamma_i) - r_L)s_i - \delta S] + \\ & \sum_{i=1}^{n}\beta_i(r_L - Rp_i + \frac{R^2}{2}(\alpha s_i - \beta_i + \gamma_i)) + \\ & \sum_{i=1}^{n}\gamma_i(Rp_i - \frac{R^2}{2}(\alpha s_i - \beta_i + \gamma_i) - r_U) \\ s.t. \quad & \alpha \geq 0, \beta_i \geq 0, \gamma_i \geq 0, \textit{ for } i = 1, ..., n \end{aligned}$$

We apply a customized Sequential Minimal Optimization (SMO) technique [7] to solve Equation (10) to find r_i.

After obtaining the replication factor for each file, we now study the problem of placing replicas on a set of candidate servers, which is equivalent to finding a mapping of replicas to servers, so that the replicas with strong mutual dependency can be assigned to the same or close servers. This is similar to the clustering problem such that the replicas are partitioned into m groups, where m is the numbers of servers.

The k-medoids algorithm [11] fits our problem well. Compared with the widely used k-means algorithm, it is more reliable to outliers and only needs the distance information between data points, while classic k-means algorithm needs the coordinates of data points, which is not readily available in our scenario. Yet the original k-medoids can not be directly applied to our problem, since we need to further incorporate the following constraints:

- Only one replica of the same file can be placed in the same server.
- The aggregated popularity of each server cannot exceed $(1+\eta)\bar{P}$.
- The total file/replica size on any server cannot exceed the storage capacity of the server.

Algorithm 1 The modified k-medois algorithm for replica placement

1: Randomly select m replicas be the initial cluster centers, namely medoids.
2: For each of the remaining replicas i, assign it to the cluster with the most dependent files**if** the constraints are satisfied.
3: **for** each medoid replica i **do**
4: **for** each non-medoid replica j **do**
5: Swap i and j **if** the constraints are satisfied.
6: Memorize the current partition if it preserves more data dependency
7: **end for**
8: **end for**
9: Iterate between steps 3 and 8 until convergence.
10: **return** m clusters

The pseudo-code of the modified k-medoids algorithm is shown in Algorithm 1. The algorithm takes the dependency matrix D as the input and returns a partition of replicas that minimizes cross-server dependency while satisfying all the constraints.

IV. Performance Evaluation

A. Simulation

We first conduct simulations to examine the effectiveness of the proposed dependency-based replication strategy. As in previous studies [12], we assume that the file popularity follows the Pareto distribution [13], yet our approach can easily adapt any popularity distribution once it is known or can be estimated. The probability density function (PDF) of a random variable X with a Pareto distribution is

$$f_X(x) = \begin{cases} \frac{\alpha x_m^{\alpha}}{x^{\alpha+1}} & x \geq x_m, \\ 0 & x < x_m. \end{cases}$$

where x_m is the minimum possible value of x, and α is a positive parameter that controls the skewness of the distribution. Assuming that there are 50 files in total in the file system, and that the file popularity follows the Pareto distribution, we plot the relative file popularity, which is the percentage of accesses of each file in the total accesses of all the files, as a function of the rank of the file in Figure 3. We can see that with larger α, the most popular files will have more accesses.

We generate data dependency based on the file popularity. We randomly pick up a dependency ratio for each pair of files between zero and the popularity of the less popular file such that popular files will have strong mutual dependency with higher probability.

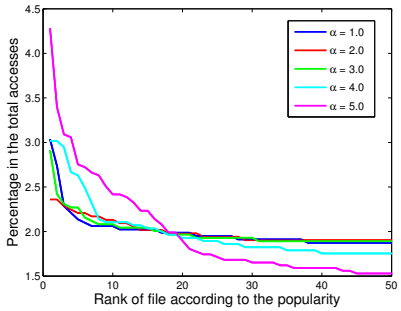

Figure 3. File rank ordered by relative popularity

We compare our dependency-based replication strategy (DALM) with the popularity-based Scarlett [2]. The default Hadoop system serves as the baseline. In the simulation, there are 50 different files with equal size and each file has at least 3 replicas. We examine the sensitivity of such system parameters as the extra storage budget ratio, the upper bound of replication factor, the number of servers, and the Pareto index α, with the default value of 20%, 5, 10, and 4.0, respectively.

The simulation results are shown in Figures 4-7, for each of the above system parameters, respectively. In general, we find that DALM outperforms the other two approaches with significantly reduced remote access. In the following we present our detailed analysis with respect to each system parameter.

Figure 4 shows the impact of the skewness of file popularity on the reduction of remote access, with Pareto index α varying from 1.0 to 5.0. We can see that with increased skewness, the data exchanges between the popular files will be more frequent, and thus DALM can reduce more remote accesses by putting highly dependent files together. However, when α = 5.0, the popularity of the most popular files is extremely high, such that the replicas of the highly dependent files cannot be stored together if both of them are very popular, due to the constraint of aggregated popularity. Scallett also experiences the similar trend, yet with less improvement over the baseline.

Figure 5 illustrates the impact of the extra storage budget ratio. We can see that when the budget ratio is low, say 6.6% and 13.3%, the reduction of remote access is limited. The reason is that with limited extra storage, only a small number of the most popular files have a lot of replicas while the other popular files do not have any replicas, and thus a lot of data access/exchange requests need to seek to remote servers. The improvement of DALM becomes remarkable when the budget ratio reaches 20%. A budget of 20% reduces the remote access substantially by 22% (need

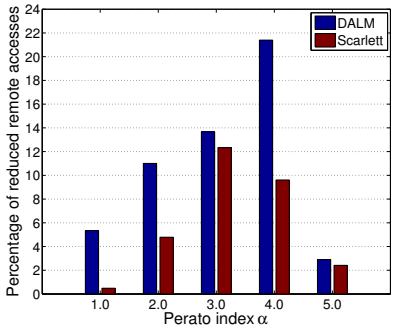

Figure 4. Impact of the skewness of file popularity

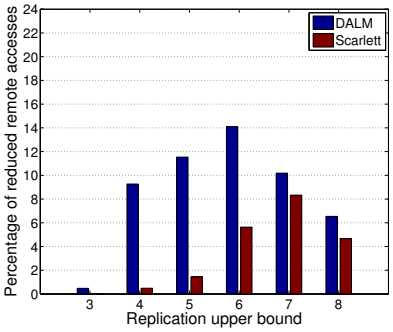

Figure 6. Impact of the upper bound of the replication factor

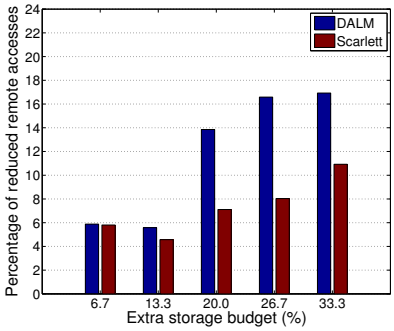

Figure 5. Impact of the extra storage budget ratio

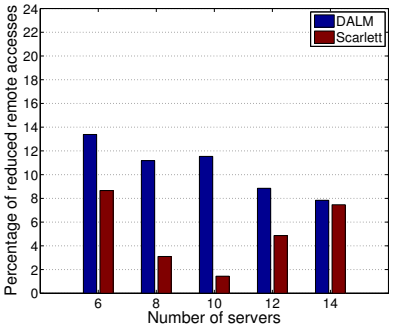

Figure 7. Impact of system scale

to rerun the simulation to get this number), compared with 7% of Scallett. It is noting that the gap between DALM and Scallett will slightly decrease when the budget ratio keeps growing, since with more extra storage, Scallett will have more replicas and thus dependent files would be stored together with higher probability.

With a fixed storage budget of 20%, the impact of the upper bound of the replication factor is shown in Figure 6. The improvement of DALM becomes more significant when the replication factor grows from 3 to 6, since a higher upper bound can give more opportunities to alleviate frequent access of the popular files; on the other hand, when the upper bound is too high, the improvement goes down since the most popular files will have excessive replicas while few of the other files have more than three replicas. Intuitively, this imbalance of the number of replicas maximizes the locality of the most popular files while sacrificing the locality of other files that occupy a large portion of the whole accesses.

Figure 7 shows the impact of the number of servers. Interestingly, the improvement of DALM becomes less with the more servers, yet still significantly outperforms Scarlett in most cases. The reason is that, with more servers, the replicas have to spread over all the servers such that the deviation of popularity of each server does not exceed the pre-defined value, namely η, in which case the data dependency cannot be well preserved. We will consider to how to smartly adapt our strategy to the number of servers. In particular, when the number of available servers is abundant, the imbalance of the servers' aggregated popularity is allowed and thus more popular files with strong dependency can be placed on the same server without causing tangible hardware resource contention.

B. Real-world experiment

We next present the real-world experiment results. We have implemented DALM on Apache Hadoop 0.23.0, and deployed it on our cluster, which consists of 4 physical machines. All the machines have a 3.40 GHz Intel Core i7-3770 quad-core CPU, 8 GB RAM, 1 TB hard disc, and Gigabit Ethernet NICs, and run the Ubuntu Linux 12.04 LTS operating system. We use the open source monitoring system Ganglia[3] 3.5.12 to monitor the system running conditions.

We compare DALM with Scarlett as well as the default Hadoop system, in terms of job completion time and cross-server traffic. We utilize a public social network data set [14], which contains 8,164,015 vertices and 153,112,820 edges and the volume is 2.10 GB. The default system settings are as follows: the block size is 32 MB and the replication factor is 2 in the default Hadoop system; the extra storage budget

[3]Ganglia Monitoring System, http://ganglia.sourceforge.net

	Average	Min	Max	Standard Deviation
DALM	115	105	123	3.4034
Scarlett	127	121	131	1.4142
Default-Hadoop	139	127	140	5.7782

Table II
JOB COMPLETION TIME (SEC)

	Average	Min	Max	Standard Deviation
DALM	493	442	524	36.1790
Scarlett	750	711	806	42.7279
Default-Hadoop	858	803	896	49.3997

Table III
CROSS-SERVER TRAFFIC (MB)

ratio is 20%, with the replication lower bound of 2 and the upper bound of 4 for both DALM and Scarlett. We allocate 32 workers in total, with 8 map workers on each slave server, and 7 map workers and 1 reduce workers on the master server. There are roughly five communities in the graph, and one of them is the popular one. We randomly a set of source nodes from the popular cluster, and a set of the destination nodes from the graph, then calculate the shortest path between each node pair. We run each strategy five times and present the detailed statistics on job completion time and cross-server traffic in Table II and Table III, respectively.

As Table II shows, DALM reduces the job completion time by 9.4% and 17.3% on average, as compared with Scarlett and the default Hadoop system, respectively. This improvement is rather encouraging since DALM has no extra overhead against Scarlett.

Table III shows that DALM reduces the cross-server network traffic by 34.3% and 42.5% on average, as compared with Scarlett and the default Hadoop system, respectively. We can see that replica placement has a significant impact on the system performance of MapReduce, particularly the iterative jobs.

V. DISCUSSION

DALM is basically composed of two modules, one computing the replication factor of each files, and the other determining the replica placement.

To compute the replication factor, we need the information of file popularity. We implemented a monitor daemon on the NameNode to automatically collect the statistics of file accesses and data dependency. We then compute the replication factor using the SMO solver. HDFS already provides a flexible API to change the default replication factor for each file. Yet it does not provide any policy to specify this value automatically and dynamically. Hence, we write a program to set the replication factor for each file, and then determine where to store each replica using our modified k-medoids algorithm, which outputs to the replica placement module on the NameNode.

In order to enforce our replica placement strategy, we modify the file placement module in the default Hadoop system. As a result, the replica placement module will transfer the replicas to the corresponding servers determined by the modified k-medoids algorithm.

DALM is highly compatible with state-of-the-art data center infrastructure and can be extended to other distributed computation paradigms if strong data dependency has been observed.

VI. RELATED WORK

The works on improving data locality in MapReduce, can be roughly divided into two categories, namely data replication, and job scheduling.

Data Replication: Scarlett [2] is a proactive replication design that periodically replicates files based on predicted popularity. The main objective of Scarlett is to alleviate hotspots caused by massive concurrent accesses to the popular files. DARE [4] dynamically detects the popularity changes at smaller time scales and adopts a caching approach: when a map task remotely accesses to some file block, this block is added to the local file system of the server, thereby automatically increasing the number of replicas of this block by one. When the workers on this server need to read this block in the future, the requests can be serviced locally.

Job Scheduling: The studies on jobs scheduling keep the default data replication approach, while striving to improve data locality by scheduling computation close to data. Quincy [15] considers the scheduling problem as an assignment problem and different assignments have different costs based on locality and fairness. Different from killing the running tasks to free the resources and launches new tasks, Delay scheduling [16] let a job wait for a small amount of time if it cannot launch tasks locally, allowing other jobs to launch tasks instead. This simple strategy works pretty well in a variety of workloads. Purlieus [17] further considers the locality of intermediate data and aims to minimize the cost of data shuffling between map tasks and reduce tasks.

Our proposed DALM takes a similar approach with the popularity-based replication strategy when deciding the replication factor of file. It differs in the replica placement approach by storing highly dependent files on the same server. DALM can also incorporate advanced job scheduling techniques, say Delay Scheduling [16], to further reduce the job completion time.

VII. CONCLUSION

As an efficient approach to improve data locality, replication has been widely adopted in many MapReduce. The existing popularity-based replication strategy mainly strive

to alleviates hotspots, at the expense of excessive cross-server data accesses/exchanges. Unveiling the root cause of this phenomenon that the files with strong data dependency are placed on different servers, we developed DALM, a dependency-aware replication strategy preserving the data dependency in replica placement to minimize cross-server traffic. Through both extensive simulations and real-world deployment of the DALM prototype, we illustrated the superiority of DALM against state-of-the-art solutions, including the Hadoop system and the popularity-based Scarlett strategy.

In the future work, we first plan to conduct a larger-scale experiment with more servers and larger datasets using the Amazong EC2 platform, to examine the gain of DALM under this multi-tenancy environment with virtualized systems. We envision that more savings can be achieved since DALM can effectively mitigate the burden on network, and thus incur less interference from the jobs of other tenants running on the same physical machine. Further, we will explore how to smartly adapt the constraint of the servers' aggregated popularity to the number of servers. That is, when the number of available servers is abundant, the imbalance of the servers' aggregated popularity is allowed and thus more popular files with strong dependency can be placed on the same server without causing tangible hardware resource contention.

ACKNOWLEDGEMENTS

This work is supported by a Canada NSERC Discovery Grant, an NSERC Strategic Project Grant, and a China NSFC Major Program of International Cooperation Grant 61120106008.

Dan Li's work is partially supported by the National Key Basic Research Program of China (973 program) under Grant 2014CB347800.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with skewed content popularity in mapreduce clusters," in *Proceedings of EuroSys*, pp. 287–300, 2011.

[3] C. L. Abad, N. Roberts, Y. Lu, and R. H. Campbell, "A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns," in *Proceedings of IEEE Symposium on Workload Characterization (IISWC)*, pp. 100–109, 2012.

[4] C. L. Abad, Y. Lu, and R. H. Campbell, "Dare: Adaptive data replication for efficient cluster scheduling," in *Proceedings of IEEE CLUSTER*, pp. 159–168, 2011.

[5] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated memory caching for parallel jobs," in *Proceedings of USENIX NSDI*, 2012.

[6] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 1013–1020, ACM, 2010.

[7] J. C. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," *Technical Report MSR-TR-98-14, Microsoft Research*, 1998.

[8] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Proceedings of ACM HDPC*, 2010.

[9] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[10] S. Boyd and L. Vandenberghe. Cambridge University Press, 2004.

[11] H.-S. Park and C.-H. Jun, "A simple and fast algorithm for k-medoids clustering," *Expert Systems with Applications*, vol. 36, no. 2, Part 2, pp. 3336–3341, 2009.

[12] J. Lin, "The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce," in *Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009.

[13] M. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary Physics*, vol. 46, no. 5, pp. 323–351, 2005.

[14] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of ICDM*, pp. 745–754, 2012.

[15] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of ACM SOSP*, SOSP '09, pp. 261–276, 2009.

[16] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of EuroSys*, pp. 265–278, 2010.

[17] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for mapreduce in a cloud," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.