Dependency-aware Data Locality for MapReduce

by

Xiaoyi Fan

B.Eng., Beijing University of Posts and Telecommunications, 2013.

Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Xiaoyi Fan 2015
SIMON FRASER UNIVERSITY
Fall 2015

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name:	Xiaoyi Fan
Degree:	Master of Science (Computing Science)
Title:	*Dependency-aware Data Locality for MapReduce*
Examining Committee:	**Dr. Ping Tan** (chair) Assistant Professor

Dr. Jiangchuan Liu
Senior Supervisor
Professor

Dr. Qianping Gu
Supervisor
Professor

Dr. Nick Sumner
Examiner
Assistant Professor

Date Defended: 17 Nov 2015

Abstract

Recent years have witnessed the prevalence of MapReduce-based systems, e.g., Apache Hadoop, in large-scale distributed data processing. In this new generation of big data processing framework, *data locality* that seeks to co-locate computation with data, can effectively improve MapReduce performance since fetching data from remote servers across multiple network switches is known to be costly. There have been a significant studies on *data locality* that seeks to co-locate computation with data, so as to reduce cross-server traffic in MapReduce. They generally assume that the input data have little dependency with each other, which however is not necessarily true for that of many real-world applications, and we show strong evidence that the finishing time of MapReduce tasks can be greatly prolonged with such data dependency. State-of-the-art data replication and task scheduling strategies achieve data locality through replicating popular files and spreading the replicas over multiple servers. We take the graph data, a widely adopted data format in many big data application scenarios, as a representative case study. It illustrates that while working well for independent files, conventional data locality strategies can store highly dependent files on different servers, incurring excessive remote data accesses and consequently prolonging the job completion time.

In this thesis, we present DALM (Dependency-Aware Locality for MapReduce), a comprehensive and practical solution toward dependency-aware locality for processing the real-world input data that can be highly skewed and dependent. DALM accommodates data-dependency in a data-locality framework, organically synthesizing the key components from data reorganization, replication, placement. Beside algorithmic design within the framework, we have also closely examined the deployment challenges, particularly in public virtualized cloud environments. We extensively evaluate DALM through both simulations and real-world implementations in a typical virtualized environment, and compare it with state-of-the-art solutions, including the default Hadoop system, the partition-based Surfer, and the popularity-based Scarlett. The results show that DALM can significantly improve data locality for different inputs. For popular iterative graph processing applications on Hadoop, our prototype implementation of DALM significantly reduces the job finish time, as compared to the default Hadoop system, Scarlett, and Surfer.

Keywords: Load Balancing; MapReduce; Online; Datacenter Network

Dedication

To my family.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my senior supervisor Dr. Jiangchuan Liu, for his constant support and insightful directions throughout my studies. His significant enlightenment, guidance and encouragement on my research are invaluable for the success of my Master Degree in the past two years. He is an excellent example as a successful researcher who inspires me to continue pursuing a Ph.D. degree in academia. Without his encouragement and knowledge, this thesis could not have been written.

I also thank Dr. Qianping Gu and Dr. Nick Sumner for serving on the thesis examining committee. I thank them for their precious time reviewing my thesis and for advising me on improving this thesis. I would like to thank Dr. Ping Tan for chairing my Master thesis defence.

I thank a number of friends for their assistant in technical matters and for the enjoyable time I spent with them during my stay at Simon Fraser University. In Particular, I thank Dr. Feng Wang, Dr. Zhe Yang, and Dr. Xiaoqiang Ma, for helping me with the research as well as many other problems during my study. I thank my colleagues and friends at Simon Fraser University, as well as ex-colleagues whom I worked with. Although I am not listing your names here, I am deeply indebted to you.

I thank Dr. Dan Li, for his initial help and guidance since the year 2012. Without his help, I would not have started my research in Computer Science. He is always like a mentor to me, and I sincerely cannot imagine how my life would be without his help.

Last but not least, I thank my family for their love, care and unconditional support: my dearest, my parents and my grandparents. Although 14 years passed, I still remember the excitement of an 11-year-old boy, when he received a Pentium 4 Desktop from his parents as a birthday gift. I sincerely hope that I have made them proud of my achievements today. This thesis is dedicated to you all.

Table of Contents

List of Figures

Chapter 1

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 [15]. 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 project[2] have become a 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 results (*the reduce stage*) [15]. This effectively distributes the computation workload to a cluster of servers; yet the data is 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 clusters with high over-provisioning ratio), fetching data from remote servers across multiple network switches can be costly. 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 Apache Hadoop, have attempted to achieve better *data locality* through replicating each file block on multiple servers. This simple uniform replication approach reduces cross-server traffic as well as job completion time for inputs of uniform popularity, but is known to be ineffective with skewed input [5, 4]. It has been observed that in certain real-world inputs, the number of accesses to popular files can be ten or even one hundred times more than that of 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* with popularity-based approaches. Representatives include Scarlett [5], DARE [4] and PACMan [6], all of which seek to place more replicas for popular blocks/files. Using the spare disk 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. 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 [51], preserves diverse social relations. A sample social network graph with four communities is shown in

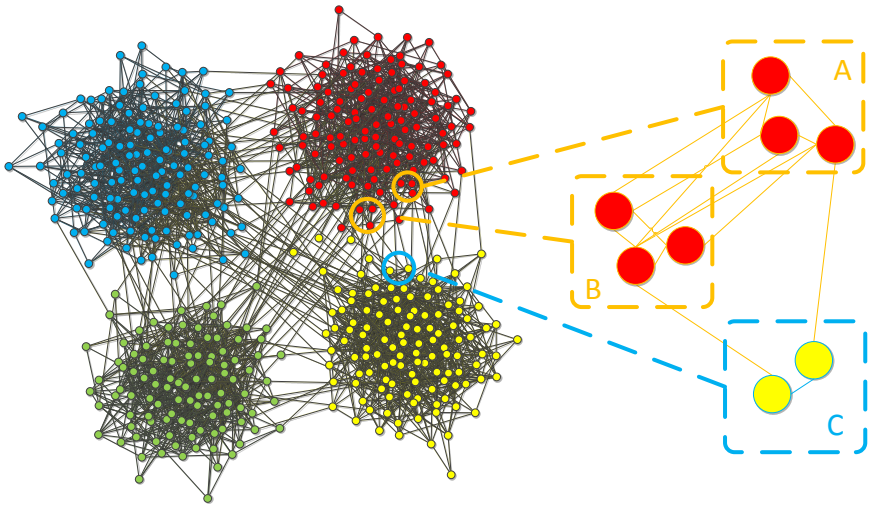


Figure 1.1: A social network graph with four communities

Figure 1.1. Here subsets A and B have very strong dependency (as compared to subsets 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 subsets 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.

Processing data with dependency, especially mining large graphs, generally involves iterative data exchange and state update. The original MapReduce framework, which was designed for parallelizable jobs, is not efficient for such jobs due to the high communication overhead [39]. To this end, a lot of efforts have been devoted to optimizing MapReduce for complex jobs involved with heavy communication. For example, Pregel [39], has been proposed for processing large graphs based on the Bulk Synchronous Parallel (BSP) model [52], followed by such implementations as Giraph[7], Surfer [13], and Mizan [36]. These systems successfully improve the job finish time, as compared to the original MapReduce. To deploy these systems in real world MapReduce clusters, such practical factors as skewed accesses, storage budget are yet to be considered. Further, many users deploy the MapReduce systems on such readily available public clouds as Amazon Web Services (AWS), instead of on dedicated servers, to enjoy the convenient and flexible *pay-as-you-go* billing option, as well as on-demand resource scaling. Such cloud services heavily rely on the *virtualization* technique to enable efficient resource sharing and scaling for massive users, which however introduces performance overhead as well as an additional topology layer and thus calls for re-examining data locality in this new context.

In this thesis, we develop a system prototype DALM (Dependency-Aware Locality for MapReduce) on the virtualized environment, for processing real-world input data that can be highly skewed and dependent. Unlike existing systems, DALM accommodates data-dependency in a data-locality framework, organically synthesizing such key components as topology-aware graph

partitioning, popularity- and dependency-based data replication, and virtualization-aware task scheduling. These efforts together lead to the development of DALM, a comprehensive and practical solution toward data locality in virtualized environments. We have implemented DALM on Hadoop 1.2.1, and have closely examined the design and deployment issues. We have extensively evaluated DALM through both simulations and real-world implementations, and have compared it with state-of-the-art solutions, including the default Hadoop system, the graph-partitioning based Surfer approach, and the popularity-based Scarlett approach. The experiment results demonstrate that the DALM replication strategy can significantly improve data locality for different inputs. For the popular iterative graph processing applications in Giraph, our prototype implementation of DALM significantly reduces the job finish time, as compared to the default Hadoop system, Scarlett and Surfer. For larger-scale multi-tenancy systems, more savings could be envisioned given that they are more susceptible to data dependency.

1.1 Background and Related Work

In this section, we offer the necessary background of our work, including a brief overview of data locality in MapReduce, as well as processing data with dependency.

1.1.1 MapReduce

MapReduce[15] is a programming model for distributed computations on the massive data and an execution framework for large-scale data processing, which has been written in different programming languages. It was originally designed by Google and built on well-known principles in parallel and distributed processing. Apache Hadoop[2] is one of the most popular free implementations, whose development was led by Yahoo (now an Apache project). The open-source Apache Hadoop system have become the preferred choice in both academia and industry in this era of big data computation at terabyte and even petabyte-scales.

As dataset sizes increase, the link between the compute nodes and the storage becomes a bottleneck. At that point, one could invest in higher performance but more expensive networks or special-purpose interconnects such as InfiniBand. In most cases, this is not a cost-effective solution, as the price of networking equipment increases non-linearly with performance. Existing datacenters generally adopt a multi-root tree topology to interconnect server nodes [29, 9]. In Fig. 1.2, we show a generic datacenter network topology. The 3 layers of the datacenter are the *edge* layer, which contains the Top-of-Rack switches that connect the servers to the datacenter's network; the *aggregation* layer, which is used to interconnect the ToR switches in the edge layer; and the *core* layer, which consists of devices that connect the datacenter to the WAN. In MapReduce, the workers are located on the leaf nodes of the tree, and the intermediate data will be moved from one leaf node to another node during the shuffle subphase.

MapReduce runs on a cluster of machines (the *nodes*) with a *master-worker* architecture, where a master node makes scheduling decision and multiple worker nodes run tasks dispatched from the master. The programmer defines a mapper and a reducer with the following two

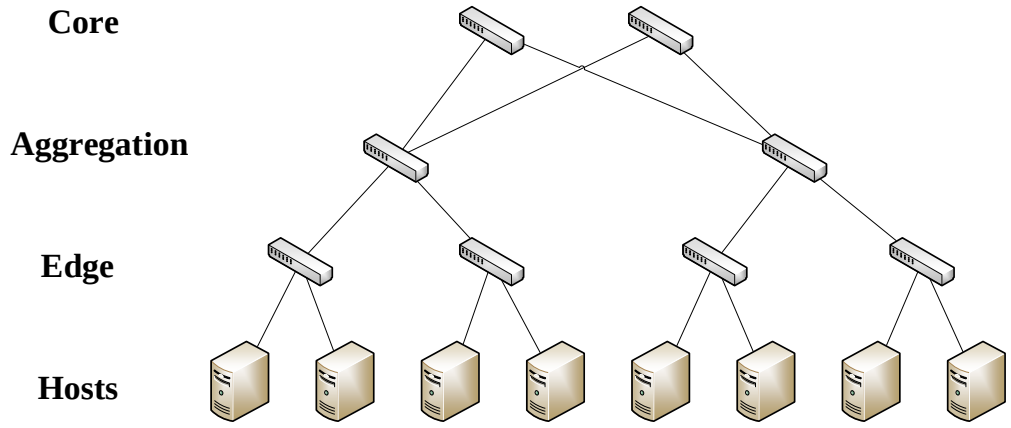


Figure 1.2: Typical 3-layer datacenter network topology.

stages. A typical MapReduce workflow consists of two major phases. In the *Map* Phase, the map processes (*mappers*) on slaves read the input data from the distributed file system, and transform the input data to a list of intermediate key-value pairs. The mapper is applied to every input key-value pair (split across an arbitrary number of files) to generate a large number of intermediate key-value pairs (key, value). The master node then uses a partition function to assign the intermediate data to different worker nodes to process in the *Reduce* Phase. In the *Reduce* Phase, the reduce processes (*reducers*) then merge the intermediate values for the distinct keys, which are stored in the local disks of slaves, to form the final results that are written back to the distributed file system. The shuffle and sort stage is between the map and reduce phases to deal with the intermediate keys. Intermediate data arrive at each reducer in order, merged by the key. Output key-value pairs from each reducer are written persistently back onto the distributed file system. The output ends up in r files on the distributed file system, where r is the number of reducers. For the most part, there is no need to consolidate reducer output, since the r files often serve as input to another MapReduce job. Fig. 1.3 illustrates the workflow of MapReduce with two-stage data processing structure.

The computation and storage are separated as distinct components in a cluster. The distributed file system (DFS) that underlies MapReduce adopts exactly this approach. The Google File System (GFS)[21] supports Google's proprietary implementation of MapReduce; meanwhile, Hadoop Distributed File System is an open-source implementation of GFS that supports Hadoop. The main idea is to divide user data into blocks and replicate those blocks across the local disks of nodes in the cluster. The master node divides the massive input data into small data blocks(default 64 MB) and spreads them to workers. Each data block has three copies stored in different machines. The distributed file system adopts a master-slave architecture in which the master maintains the file namespace, involving metadata, directory structure, file to block mapping, location of blocks, and access permissions, and the slaves manage the data blocks.

In HDFS, when one client wants to read a file (or a portion thereof), the client must first contact the namenode to determine where the actual data is stored. In response to the client

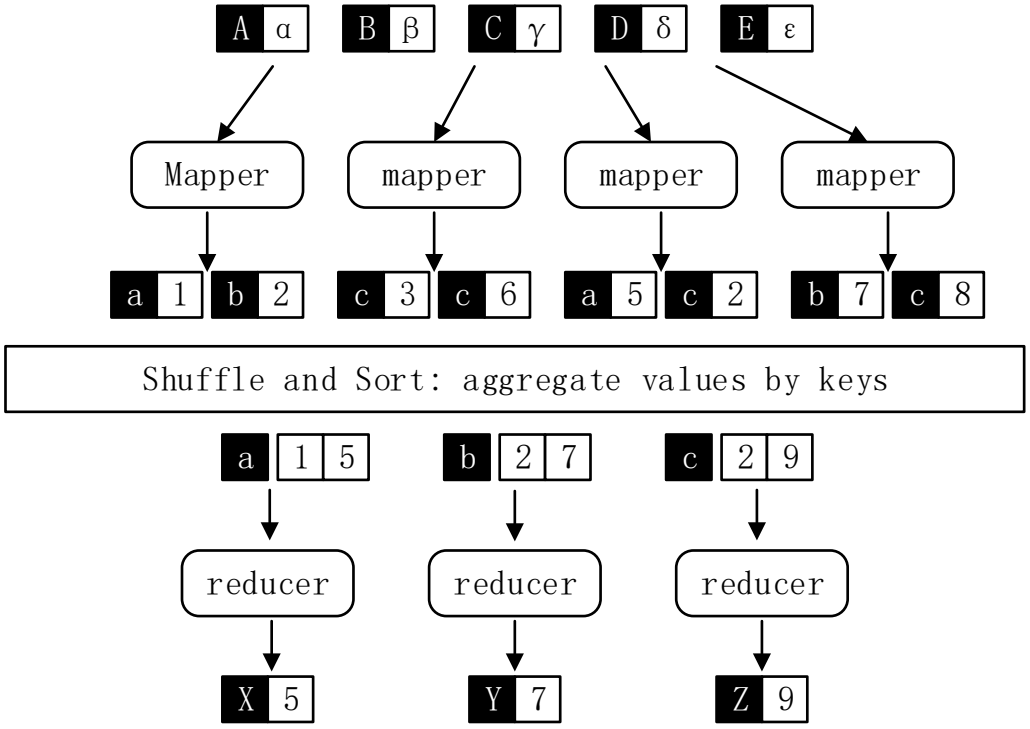


Figure 1.3: MapReduce Model.

request, the namenode returns the relevant block id and the specific location where the block is held. The client then contacts the datanode to fetch the data. An important feature of the design is that data is never moved through the namenode. Instead, all data transfer occurs directly between clients and datanodes; communications with the namenode only involves transfer of metadata. By default, HDFS divides data into fixed size blocks and stores three separate copies of each data block to ensure both reliability, availability, and performance. This mechanism is meant to be used for improved availability, response time, read throughput and load balancing. More recently, data replication in distributed file systems is widely used to improve data locality.[21] However, too many replicas may not significantly improve availability, but with unnecessary costs.

1.1.2 Data Locality in MapReduce System

Co-locating computation with data, namely, *data locality*, which largely avoids the costly massive data exchange crossing switches, can significantly improve the job finish time of most MapReduce applications [22]. By default, MapReduce achieves data locality via data replication, which divides data files into blocks with fixed size (e.g., 64 MB) and stores multiple replicas of each block on different servers. More advanced solutions have been proposed to further improve data locality, either through smart and dynamic data replication, e.g., [5, 4], or job scheduling, e.g., [57, 27, 55, 44].

Scarlett [5] captures the popularity of files and uses these to compute a replication factor for each file with the goal of alleviating the hotspots. It adopts a proactive replication design that

periodically replicates files based on predicted popularity. It captures the popularity of files, and computes a replication factor for each file to alleviate hotspots. Scarlett is an offline system, which uses a proactive replication design that periodically replicates files based on predicted popularity.

DARE [4] utilizes a reactive approach that dynamically detects popularity changes at smaller time scales. When a map task remotely accesses a block, the block is inserted into the file system at the node that fetched it. By doing this, the number of replications of the block is automatically increased by one, without generating explicit network traffic. DARE proposed a reactive approach how dynamically detects the popularity changes at smaller time scales. In DARE, when a map task remotely access to the block, the block is inserted into the file system at the node that fetched it. By doing this, the number of replications of the block is automatically increased by one, without generating explicit network traffic. The design philosophy of DARE uses a greedy approach, which assumes that any nonlocal data accessed remotely is worth replication.

Quincy [27] 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 launching new tasks, Quincy provides an optimal online scheduler according to a global cost model for scheduling concurrent distributed jobs with fine-grain resource sharing.

Parallel to the replications work on the storage system, other researchers did a large characterization in the job scheduling of MapReduce, which we cannot ignore these techniques if we want to have a deep insight in the data locality are.

Delay scheduling [55] lets 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, which improves the resource utilization, works pretty well for a variety of workloads.

Delay scheduling is proposed as a locality-aware scheduling policy to solve two locality problems in Fair Scheduler, which are head-of-line scheduling and sticky slots. Specifically the small head-of-line jobs cause a locality problem, since it is unlikely to have data locally on the node. The other related locality problem is caused by sticky slots, where there is a tendency for a job to be assigned the same slot repeatedly. Delay scheduling allocates equal shares to each of the users, and also tries to maximize data locality by delaying the scheduling of the task, when no local data is available. The key idea in delay scheduler is that: although the current slot have no available data for such a job, other tasks may finish soon so that some slots with data will be free in the coming few seconds. Specifically when a node requests a task, if the head-of-line job cannot launch a local task, it is skipped and looked at subsequent jobs. Moreover, if a job has been skipped long enough, non-local tasks are allowed to launch to avoid starvation.

Purlieus [44] considers the locality of intermediate data, and aims to minimize the cost of data shuffling between map tasks and reduce tasks. Purlieus focuses on both input and intermediate date, which differentiates from Quincy. The authors emphasize that the locality of any intermediate data during job execution is a key factor to scaling MapReduce in large data centers. Purlieus categorizes the characteristics of MapReduce workloads jobs into three classes:

$map-input\ heavy$, $map-and-reduce-input\ heavy$ and $reduce-input\ heavy$, and proposes a strategy for both data and computation placement. Purlieus scheduler depends on the prior workload information to make the decision, which aims to minimize the cost of data shuffling between map tasks and reduce tasks.

Our work complements prior research on data storage level by enhancing the popularity methods with file dependencies. We provide a better understanding of the correlation between popularity and file dependencies, which is a novel dimension at the storage level.

1.1.3 Processing Data with Dependency

Since the original MapReduce is not suitable for processing data with strong dependency, e.g., graphs, a lot of improvements have been proposed. For example, In Pregel [39], the inter-worker communication is implemented through message passing rather than remotely reading the entire new state in MapReduce, which significantly improves the performance on processing large graphs as compared to the original MapReduce. Based on Pregel, several systems have been developed, e.g., Giraph, Surfer [13], and Mizan [36]. Giraph is implemented on top of Hadoop and inherits the convenience of Hadoop such as distributed storage and fault tolerance, while Mizan and Surfer are developed with C++ in order to further improve the efficiency. Giraph simply uses the underlying HDFS for storing graph data and adopts an online hash- or range-based partitioning strategy to dispatch subgraphs to workers, which successfully balances computation and communication. This naive partitioning strategy largely neglects the structure of graphs; subgraphs with strong dependency can be assigned to different workers, which incurs excessive volume of message passing. Surfer considers the heterogeneous network bandwidth in the cloud environment and thus proposes a network performance aware graph partitioning framework. The partitions with a large number of cross-partition edges are stored in the machines with high network bandwidth. Mizan monitors the message traffic of each worker and balances the workload through migrating selected workers across workers. Surfer and Mizan outperform Giraph since they take a further step toward adapting to the structure of graphs, which should be incorporated into distributed large graph processing.

The need to extend MapReduce to support iterative jobs was also recognized by many researchers, which leads to the success of parallel execution frameworks for processing big data. These frameworks offer high-level languages for Map-Reduce-like environment and merely require the user to specify data-parallel computation, and automatically handle parallelization, locality, fault tolerance, and load distribution. For example, Dremel [40] and Trill[12] both propose fast execution frameworks for SQL-like queries to enable parallelism, yet these systems only parallelize a class of known SQL aggregators. Likewise, Pig[19] and Pig Latin[42], Spark[56], FlumeJava[11] all provide a lifting operation that pushes aggregations into maps, however, only for some limited built-in associative functions. Recently, SIMPLE[49] is provided to increase the expressivity of these frameworks by breaking dependencies through symbolic summaries.

Partitioning a graph into roughly equal subsets with the objective of minimizing the total cost of the edges cut is an NP-complete problem [35], which has been extensively studied by many

researchers. The multilevel approach has been widely used to find good solutions efficiently [24, 33], which consists of three phases, namely coarsening, partitioning and uncoarsening. Instead of directly partitioning the original graph, the multilevel approach first coarsens down the original graph to a much smaller graph, say hundreds of vertices. Then the partitions of this small graph will be computed. Since the size of the coarsened graph is quite small, classic graph partitioning algorithms such as Kernighan-Lin (KL) [35] or Fiduccia-Mattheyses (FM) [18] can obtain reasonably good results in polynomial time. Then the partitions are projected back toward the original graph, with refinements to further improve the quality of the partitioning results. Some optimized approaches [34, 14] have been further proposed in the specific phases of multilevel partitioning. Given the effectiveness and efficiency of the multilevel approach, several software libraries, such as METIS [32] and SCOTCH [47] have been developed based on it, which provide convenient tools for graph partitioning.

Recently, the cluster or community detection in large-scale social graphs has attracted a lot of attention from researchers given the surge of social networks. Many approaches have been proposed to efficiently and accurately find communities in large graphs, e.g. [37, 31, 53]. Such approaches aim at identifying network communities in large graphs such that the nodes have more linked edges are grouped together. Such approaches are not suitable for our dependency-preserving data partitioning for the following reasons. First, the community detection approaches often divide graphs into communities consisting of up to hundreds of nodes [37]. Considering that the size of file blocks in HDFS is 64 MB by default, the granularity of communities is too small for data storage. Further, the size of communities can be highly heterogeneous across different communities, which also makes it difficult to use communities as the basic storage unit. Second, the communities detection approaches are non-parametric, which means that the number of communities is automatically determined once all the communities are found. On the other hand, the graph partitioning approach generally divides the graphs into k partitions of almost equal size, where k is the user-defined parameter which can adapt to the storage system and system scale. Third, graph partitioning approaches are often more computation efficient than community detection approaches, which are preferred in practical systems.

1.1.4 MapReduce in Virtualized Environment

As one of the most important technical foundations of virtualized clouds, virtualization techniques (e.g., Xen, KVM, and VMware) allow multiple *virtual machines* (VMs) running on a single *physical machine* (PM), which achieves highly efficient hardware resource multiplexing, and effectively reduces the operating costs of cloud providers. Yet, it has been identified that MapReduce jobs running on virtual machines can take significantly longer time to finish, as compared with directly running on physical counterparts [26]. The reasons are two-fold. First, the virtualization technology itself introduces nontrivial performance overheads to almost every aspect of computer systems (e.g., CPU, memory, hard disk, and network). Most applications, no matter computation or I/O intensive, will experience certain performance penalty when running on virtual machines. Second, the off-the-shelf MapReduce systems, e.g., Hadoop, are

originally designed for physical machine clusters; the unique characteristics of virtual machines, say resource sharing/contention and VM scheduling, are yet to be accommodated, which further exaggerate the negative impact of virtualization on MapReduce tasks.

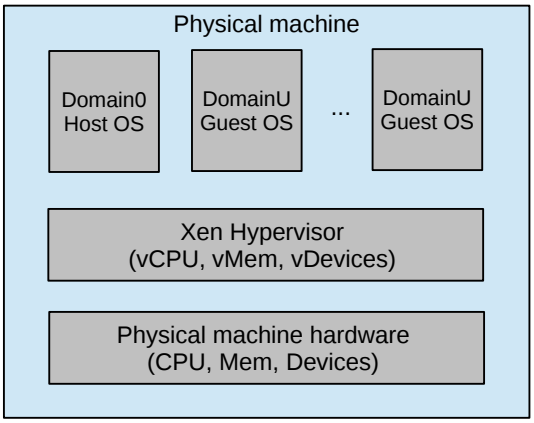


Figure 1.4: Xen architecture.

In state-of-the-art systems, virtualization is often achieved through the use of a software module known as a *Hypervisor*, which works as an arbiter between a virtual devices and the underlying physical devices [8]. Using Xen [8], an open source virtualization tool that has been widely used by public clouds, including Amazon AWS [1] and Rackspace [3], as a representative, Fig. 1.4 shows the overall architecture of a typical Xen-based virtualized system. A Xen *Hypervisor*, which is also called the *virtual machine monitor* (VMM), provides the resource mapping between the virtual hardware and the underlying real hardware of the physical machine. A privileged VM, namely, the *Domain0*, is created at boot time and is allowed to use the control interface. The Hypervisor works together with the *host OS* running on the Domain0 to provide system management utilities for users to manage the physical machine as well as VMs, e.g., the CPU scheduling policy and the resource allocation. The OS running on an unprivileged domain (*DomainU*) VM is called the *guest OS*, which can only access the resources that are allocated by the Hypervisor. It is worth noting that the host OS and the guest OSes can be different. The DomainU VMs cannot directly access the I/O devices; rather, the Domain0 VM handles all of the I/O processing. Xen uses the shared memory mechanism for data transfer between co-located VMs [8]. Intuitively, existing data locality mechanisms should also work well in a virtualized environment, which has yet to be validated (or invalidated).

Realizing the great need and potential of running MapReduce on virtualized public clouds, there have been pioneer studies toward improving the performance of MapReduce over virtual machine clusters, e.g., [25, 30, 46, 10]. One important aspect, *data locality* that seeks to co-locate computation with data, however has yet to be addressed in this new context. Since fetching data from remote servers across multiple network switches can be costly, data locality can effectively improve the MapReduce performance in a physical machine cluster or datacenter with high over-provisioning ratio [22]. Intuitively, it should also work well by placing data close to VMs, which unfortunately is not true in a virtualized cloud. Ibrahim *et al.* [25] identified that different disk

pair schedulers cause Hadoop performance variation and accordingly proposed an adaptive disk I/O scheduling algorithm. Fang *et al.* [17] discussed the I/O scheduling in Xen-based virtualized clouds, and suggested two improvements: enhancing Domain0's weight dynamically, and using physical machines as master nodes. Kang *etal.* [30] proposed the MapReduce Group Scheduler (MRG), targeting multiple MapReduce clusters running on the same physical machine cluster. Yuan *etal.* [54] developed an interference-aware algorithm that smartly allocates MapReduce tasks to different VMs. Given the multi-tenancy nature, the performance of virtual machines can be highly heterogeneous and variant, and the LATE scheduling algorithm [57] was proposed for job scheduling in this new context.

Data locality is critical to the performance of MapReduce tasks, and there have been significant studies toward improving data locality, e.g.,[22, 5, 4, 27, 55, 58, 23]. However, these works on data locality have yet to address the distinguished challenges from virtual machines in a public cloud. In the virtualized environment, a location-aware data allocation strategy was proposed in [20], which allocates file blocks across all physical machines evenly and the replicas are located in different physical machines. Purlieus [44] improves the data locality through locality-aware VM placement such that the data transfer overhead is minimized. DRR [46] enhances locality-aware task scheduling through dynamically increasing or decreasing the computation capability of each node. An interference and locality-aware (ILA) scheduling strategy has been developed for virtualized MapReduce clusters, using a task performance prediction model to mitigate inter-VM interference and preserve task data locality [10].

1.2 Motivation

In this section, we present an overview of MapReduce and datacenter network. We will also discuss the data locality issues and the cost in virtualized environment that motivates our study.

Hadoop[2], a popular open source implementation of the MapReduce framework, is used to process the large graph. They sequentially store the files, which are uniformly divided into blocks. Pregel[39] is made to balance the workload with different approaches. Giraph[7] as an existing implementations of Pregel system provides a hash-based graph partitioning schemes, which are taken efficiently as a preprocessing step. Surfer[13] with more sophisticated partitioning techniques, such as min-cuts. Hama[50] utilizes distributed data stores and graph indexing on vertices and edges. Yet graph partitioning alone is not sufficient for miniming the computation, Mizan[36] provides a range-based partitioning approach to adaptively balance workload in graph processing.

We take Figure 1.5 as an example to illustrate the influence of data replication strategies, where files A and B are popular, and files C, D, E, F are not. Now assume that there is strong data dependency between each of the file pairs (A,B), (C,D), and (E,F), weak dependency between (A,C), (A,E), and (D,F), and no dependency otherwise. We try to find a good storage strategy to minimize the traffic overhead, according to the dependency among files, as well as the file popularity.

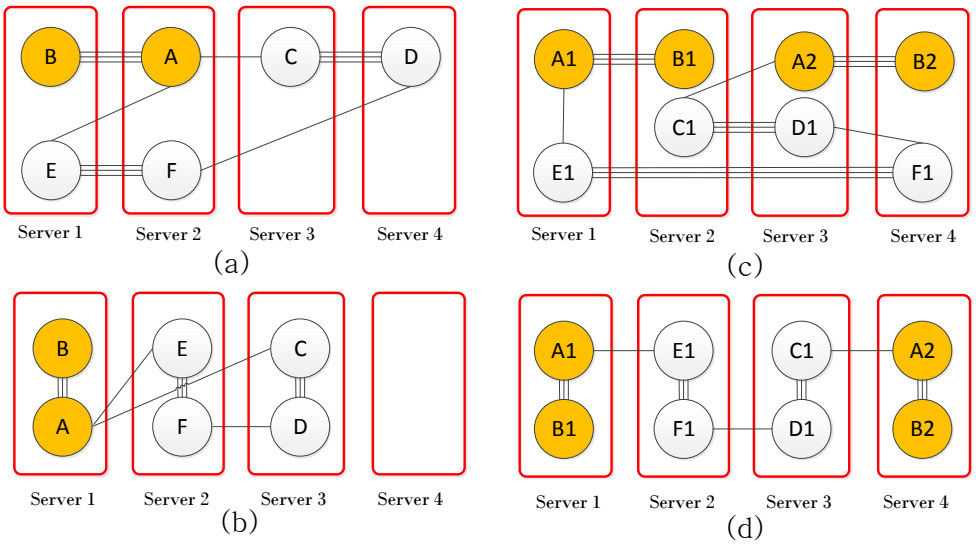


Figure 1.5: An example of different replication strategies based on (a) Hadoop only; (b)graph partition approach only; (c) data popularity only; (d) both data popularity and dependency. The number of edges between nodes denotes the level of dependency.

As Figure 1.5(a) shows, the default replication strategy of Hadoop uniformly spreads files over all the servers, which achieves good fault tolerance and scalability. Yet it does not consider file popularity and dependency, and thus would incur high cross-server traffic. Figure 1.5(b) demonstrates the efforts of more advanced solutions with min-cuts based partitioning techniques to balance computation and communication [13]. Not surprisingly, these approaches can reduce a large amount of cross-server traffic among workers since files with strong dependency are stored together. On the other hand, these works largely neglect file popularity, leading to imbalanced workload on individual servers. For example, server 1 would become the hotspot since it hosts more popular files, as compared to other servers. (Server 4 even has no jobs to schedule).

On the other hand, the popularity-based replication strategies, e.g., Scarlett [5], work well only 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. A basic popularity-based strategy would create 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 to distributed to four servers, each accompanying a replica of an unpopular file (see Figure 1.5(b)). This strategy, however, would incur frequent cross-server traffic when highly dependent files are stored separately. For example, since files A and B have strong mutual dependency, there will be frequent communications between servers 1 and 2, so will be servers 3 and 4.

Hence, a better replication strategy should consider both file popularity and dependency, as shown in Figure 1.5(d). We can replicate files based on file popularity and place the files with strong dependency in close vicinity. Not only the cross-server traffic can be largely reduced, but also the imbalanced computation workloads can be relieved.

Such data dependency exists 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 [16]. These tasks will keep waiting until all the necessary data exchanges are completed. Consider the page ranking algorithm [43]. 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 large social network will involve a number of files, since each single file generally stores a small portion of the whole graph data only.

It is worth noting that a dependency-aware strategy may result in less-balanced workload of individual servers, as shown in Figure 1.5(d). 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-based strategy should strive to localize data access as much as possible while avoiding potential hotspots through replicating popular files.

1.3 Thesis Organization

The organization of the rest of this thesis is as follows:

In chapter 2, we formulate the problem and describe the design of our proposed replication strategy. 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.

In chapter 3, through real-world experiments, we show strong evidence that the conventional notion of data locality designed for physical machines needs substantial revision to accurately reflect the data locality in virtualized environments. Modifying the storage architecture to improve data locality in the physical machine level however is nontrivial, as the current task scheduler module in Hadoop is unable to distinguish the difference: when scheduling tasks, co-located VMs have the same priority as the VMs on other physical machines in the same rack. To this end, we develop an enhanced task scheduling algorithm for DALM in the virtualized environment, namely, *vScheduling*, that assigns higher priority to co-located VMs.

In chapter 4, 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[1] 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.

In chapter 5, we discuss some points and ideas that need to improve or deserve continuing working on in the future work and conclude this thesis finally.

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

Chapter 2

Dependency-aware Data Locality

In this chapter, we present the overview of our dependency-aware data replication in general MapReduce clusters. Figure 2.1 provides a high level description of the DALM workflow. The *preprocess* module operates on the raw data, which utilizes the ParMETIS algorithm [32] to identify the dependency in the data and reorganize the raw data accordingly. The *popularity* module in HDFS records the access counters for each data block, and appropriately modifies the replication factors of files using the replication strategy, which is described in Section 2.3. When one block has a higher replication factor than the actual number of the replicas, the *popularity* module will insert replication blocks in the HDFS triggered by remote-data accesses. Certain data structures are added to maintain and update such information as file dependency and popularity. In the following part, we discuss the design and optimization of DALM in detail.

2.1 Dependency-aware Data Reorganization

In the existing HDFS, the raw input data is sequentially inserted into the distributed file system, and thus the dependency among data will be hardly preserved. For example, a large connected component will be divided into a number of subgraphs and then stored on different servers. When specific tasks need to process this connected component, the states of all the servers storing data need to be synchronized, which may incur considerable communication overhead. In DALM, before loading directly the raw data files to the underlying distributed file system, the data dependency is identified via preprocessing through graph partitioning techniques, which helps to place dependent data as close as possible. Figure 2.1 shows how the preprocessing step is gracefully integrated into HDFS. DALM balances its workload by detecting the dependency in the graph, placing dependent data blocks together, and then determines the replication factor of data blocks based on the file popularity.

There have been extensive studies rapidly achieving relatively good partitioning quality for large scale graphs. Specifically, ParMETIS [32] is widely used to partition the graph based on dependency, which can be used to reorganize the input data in our proposed DALM. The partitioning algorithm is a very natural way of clustering points, which is based on the key observation that the points in the same cluster have a high degree of self-similarity among them.

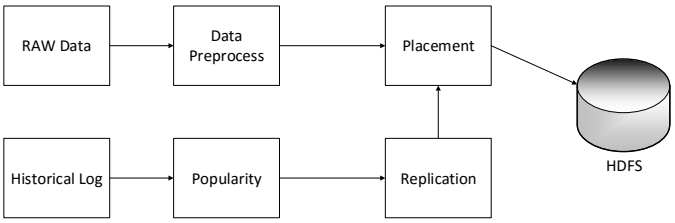


Figure 2.1: DALM workflow

Table 2.1: 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 file popularity of server j
$\bar{P}$	Average of aggregate 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 degree of dependency between files i and j

The algorithms implemented in ParMETIS are based on the parallel multilevel k-cut graph partitioning, adaptive repartitioning, and parallel multi-constrained partitioning.

2.2 Minimizing Cross-server Traffic: Problem Formulation

After reorganizing the raw data based on dependency, DALM aims at reducing the cross-server traffic by placing dependent data on nearby servers, as formulated in the following. Table 2.1 summarizes the notations. Consider that there are n different files, and that for file i, denoting its the popularity by p_i and its size s_i. The value of the element d_{ij} of the matrix D refers to the dependency degree between files i and j. In DALM, the dependency degree is obtained by normalizing the size of cut of the considered two files. In the first step, we need to determine 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, then 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 discuss the constraints in the considered data replication and placement problem.

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

$$\sum_{i=1}^{n}\sum_{k=1}^{r_i} I_j(\rho_i^k)s_i \leq c_j \tag{2.1}$$

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

In the 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. In our considered problem, the replication factor of each file can vary from the lower bound (r_L) to the upper bound (r_U) based on the file popularity.

Naturally, the extra replicas beyond the lower bound provide higher data availability, which also 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:

$$\sum_{i=1}^{n}(r_i - r_L)s_i \leq \delta S \tag{2.2}$$

$$r_L \leq r_i \leq r_U, \ \textit{for } i = 1, ..., n \tag{2.3}$$

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 aggregate file popularity of server j, which is the sum of the popularity of all the replicas stored in this server, and $\bar{P}$ the average of aggregate 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,

$$P_j = \sum_{i=1}^{n}\sum_{k=1}^{r_i} I_j(\rho_i^k)\frac{p_i}{r_i} \leq (1+\eta)\bar{P} \tag{2.4}$$

where η is the threshold of the deviation of P_j from $\bar{P}$. We divide p_i by r_i by assuming that the access to file i evenly spreads 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 [5, 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 the 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.

2.3 Replication Strategy

We compute the replication factor r_i of file f_i based on the file popularity and size, as well as the storage budget given by δ. p_i is the observed number of concurrent accesses. Files with large size have a strictly higher priority of getting their desired number of replications.

Following is the optimization problem to obtain the file replication factor:

$$\begin{aligned} & \min ||\frac{\mathbf{r}}{R} - \mathbf{p}||^2 \\ s.t. \quad & \sum_{i=1}^{n}(r_i - r_L)s_i \leq \delta S \\ & r_L \leq r_i \leq r_U, \ for\ i = 1, .., n \end{aligned} \tag{2.5}$$

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

To solve it, we start from the Lagrangian function:

$$\begin{aligned} f(\mathbf{r}) = ||\frac{\mathbf{r}}{R} - \mathbf{p}||^2 + \alpha[\sum_{i=1}^{n}(r_i - r_L)s_i - \delta S] \\ + \sum_{i=1}^{n}\beta_i(r_L - r_i) + \sum_{i=1}^{n}\gamma_i(r_i - r_U) \end{aligned} \tag{2.6}$$

where α, β_i and γ_i are the Lagrange multipliers. Now the Lagrangian Dual becomes

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

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

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

Namely,

$$r_i = Rp_i - \frac{R^2}{2}(\alpha s_i - \beta_i + \gamma_i) \tag{2.7}$$

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

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

This is a quadratic programming (QP) optimization problem with linear constraints. Rather than adopting the time-consuming numerical QP optimization, here we use the Sequential Minimal Optimization (SMO) technique [48] to solve the above problem to find r_i. The main idea of SMO is as follows. SMO breaks the original QP problem into a series of smallest sub-problems, which have only two multipliers and can be solved analytically. The original problem is solved when all the Lagrange multipliers satisfy the Karush-Kuhn-Tucker (KKT) conditions. In other words, SMO consists of two components: solving the sub-problems with two Lagrange multipliers through an analytic method, and choosing which pair of multipliers to optimize, through heuristics in general. The first multiplier is selected among the multipliers that violate the KKT conditions, and the second multiplier is selected such that the dual objective has a large increase. SMO is guaranteed to converge, and various heuristics have been developed to speed up the algorithm. Compared to numerical QP optimization methods, SMO scales better and runs much faster.

2.4 Placement Strategy

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

After obtaining the replication factor of each file, we now study the problem of placing replicas on a set of candidate servers, which is equivalent to finding a mapping from replicas to servers, so that the replicas with strong dependency can be assigned to the same server or nearby ones. This is similar to the clustering problem that the replicas are partitioned into m groups, where m is the numbers of servers. The k-means clustering approach [28] (including its variation k-median clustering) is perhaps the most widely used technique for data clustering. It is however not feasible for our problem, since the k-means algorithm requires the data points have coordinates in a feature space, which is not readily available in our scenario. The k-medoids algorithm [45] fits our problem well. Compared with the k-means algorithm, it is more reliable to outliers and only needs the distance information between data points, which corresponds to the data dependency in our scenario. 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.

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 the cross-server dependency while satisfying all the constraints. It is worth noting that this algorithm assumes that all the servers are in the same layer, e.g., interconnected by a single core switch. For a multilevel tree-topology cluster, this algorithm should be modified in a hierarchical way, which means that the data replicas are iteratively partitioned from the top level to the bottom level, such that dependent data is placed as close as possible.

Chapter 3

Data Locality in Virtualized Environment

In this chapter, we will discuss the implementation of the DALM prototype in the virtualized environment, and the deployment issues in practical systems. Before we dive into the prototype implementation details, we will examine the data locality issue in the virtualized environment, as many users deploy the MapReduce systems on the readily available public clouds, which have unique characteristics as compared to physical machine clusters. The virtualization techniques (e.g., Xen, KVM, and VMware) lie as the technical foundation of most public clouds, which allow multiple *virtual machines* (VMs) running on a single *physical machine* (PM). Virtualization achieves highly efficient hardware resource sharing and effectively reduces the operating costs of cloud providers. It is known that virtualization would introduce non-trivial performance overhead to most aspects of computer systems, and many applications, no matter computation or I/O intensive, will experience certain performance penalty when running on virtual machines. To this end, there have been a lot of studies toward examining and improving the performance MapReduce over virtual machine clusters, e.g., [30, 44, 46, 10]. The data dependency issue, however, has yet to be addressed in the virtualized environment.

3.1 Accommodating Machine Virtualization

In a physical machine cluster, each slave node as part of file system in MapReduce has a *DataNode* to store a portion of data, and a *TaskTracker* that accepts and schedules tasks. The *NameNode* on the master node keeps the directory tree of all files on DataNodes, and keeps track of the locations of files. Similarly, in a virtual machine cluster, each virtual machine works as a slave, which also contains a *DataNode* and a *TaskTracker* by default. This *balanced* architecture is very intuitive and is supposed to provide the optimal performance, so that each virtual machine can access the data locally, achieving the maximum *data locality*. To reduce the cross-server network traffic during the job execution, the MapReduce task scheduler on the master node usually places a task onto the slave, on which the required input data is available if possible. This is not always successful since the slave nodes having the input data may not have free

slots at that time. Recall that the default replication factor is three in the Hadoop systems, which means that each file block is stored on three servers–two of them are within the same rack and the remaining one is in a different rack. Hence, depending on the distance between DataNode and TaskTracker, the default Hadoop defines three levels of data locality, namely, *node-local*, *rack-local*, and *off-switch*. Node-local is the optimal case that the task is scheduled on the node having data. Rack-local represents a suboptimal case that the task is scheduled on a node different from the node having data; the two nodes are within the same rack. Rack-local will incur cross-server traffic. Off-switch is the worst case, in which both the node-local and rack-local nodes have no free slots, and thus the task needs to retrieve data from a node in a different rack, incurring cross-rack traffic. When scheduling a task, the task scheduler takes a priority-based strategy: node-local has the highest priority, whereas off-switch has the lowest.

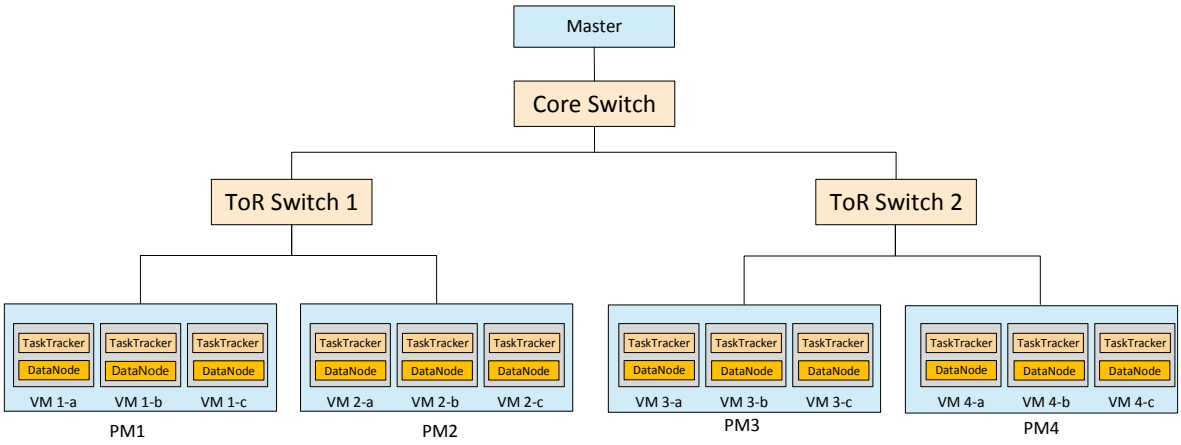


Figure 3.1: A typical virtual MapReduce cluster. A core switch is a high-capacity switch interconnecting top-of-rack (ToR) switches, which have relatively low capacity.

DALM can be easily incorporated into this conventional three-level design with a physical machine cluster. Such machine virtualization tools as Xen, KVM, and VMware allow multiple *virtual machines* (VMs) running on a single *physical machine* (PM), offering highly efficient hardware resource sharing and effectively reducing the operating costs of cloud providers. They have not only become a foundation for today's modern cloud platforms, but also affect the notion of data locality [30, 44, 46, 10]. For illustration, consider the overall architecture of a typical Xen-based virtualized system in Figure 3.1. For a file block stored on VM 1-a, scheduling the task on VM 1-b or VM 2-b are considered identical by the task scheduler, for both are rack-local. The data exchanges between co-located VMs (e.g., VM 1-a and VM 1-b), however, are much faster than those between remote VMs (e.g., VM 1-b and VM 2-b), and hence the two cases should have different scheduling priorities.

To understand the impact to locality with highly-dependent data, we set up a MapReduce cluster consisting of three physical machines, as Fig 3.2 shows: PM3 acts as the master node, and the other two PM1 and PM2 runs three VMs. As such, our cluster has six slave nodes in total. For the file block placement strategy, we consider the following three representatives: 1.

first divide the whole graph into two partitions, and then further divide each partition into three smaller partitions, which are placed on the co-located three VMs, respectively; 2. divide the graph into six partitions, and randomly place each partition on each VM; 3. uniformly place the graph data on all VMs.

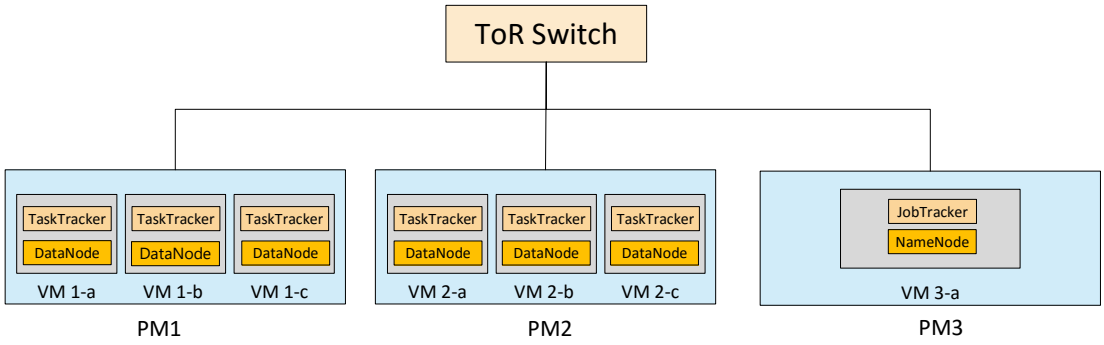


Figure 3.2: The Xen-based testbed examines the different storage strategies.

We have investigated the impact of data dependency when the data is highly dependent on our testbed virtualized platform. Specifically, we extract 3.5 GB wikipedia data from the wikimedia database[1], and then run the *Shortest Path* application in the widely used Giraph system as the microbenchmark. We run the shortest path application five times with each storage strategy, report the average job finish time in Figure 3.3. The first storage strategy achieves the best performance since it matches well with the underlying server topology. As compared with the second strategy, it can significantly reduce the cross-server traffic by placing dependent data on the same PM, which on the other hand, is offloaded to the much more efficient communication among co-located VMs.

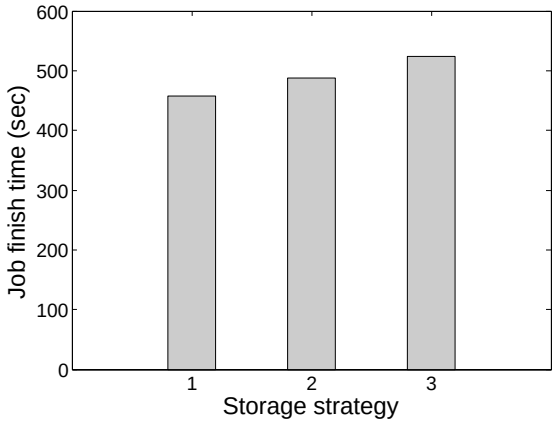


Figure 3.3: Job finish time with different storage strategies.

Since fetching data from remote servers across multiple network switches can be costly, data locality can effectively improve the MapReduce performance in a physical machine cluster or datacenter with high over-provisioning ratio [22]. When the basic popularity-based replication strategy works on virtualized environment, it would spread files with strong dependency over

[1]Available at `http://dumps.wikimedia.org/`

different virtual machines. Such a default MapReduce scheduling with a basic replication strategy fairly distribute data replicates onto these VMs, which can incur frequent remote data accesses. The basic strategy and scheduling not only introduces excessive cross-server traffic but also prolong the overall job completion time.

Based on above observation, we further provide a two-level storage strategy of DALM to fit in the virtualized environment. Similar to the first representative approach in the above experiment, we will first partition, replicate and place the data in physical machine level, then further divide the data block clusters in the virtual machine level. Intuitively, we can understand that most traffic are transferred inside the physical machine, which can considerately reduce the network cost.

3.2 System Design

Given the observations above, it is necessary to re-visit the notation of data locality for MapReduce in virtualized clouds. This inspires our design and development of DALM, which seeks to improve the effectiveness of locality in the virtualized environment, yet with minimized modifications to the existing MapReduce implementations. In this section, we first illustrate the high-level design of DALM and then, using Hadoop as a representative, discuss a series of practical issues toward real-world implementation that offers a convenient interface configure parameters based on specific virtual environments.

Fig. 3.1 illustrates the architecture of DALM. To reduce the cross-server network traffic during the job execution, the task scheduler on the master node usually places a task onto the slave, on which the required input data is available if possible. Yet this is not always successful since the slave nodes having the input data may not have free slots at that time. Recall that the default replication factor is three in the Hadoop systems, which means that each file block is stored on three servers—two of them are within the same rack and the remaining one is in a different rack. Hence, depending on the distance between DataNode and TaskTracker, the default Hadoop defines three levels of data locality, namely, *node-local*, *rack-local*, and *off-switch*. Node-local is the optimal case that the task is scheduled on the node having data. Rack-local represents a suboptimal case that the task is scheduled on a node different from the node having data; yet the two nodes are within the same rack. Rack-local will incur cross-server traffic. Off-switch is the worst case, which both the node-local and rack-local nodes have no free slots, and thus the task needs to retrieve data from a node in a different rack, incurring both cross-server and cross-rack traffic. When scheduling a task, the task scheduler takes a priority-based strategy: node-local has the highest priority, whereas off-switch has the lowest.

In virtualized MapReduce clusters, however, the three-level design is not enough to accurately reflect the data locality. For example, in Fig. 3.1, for a file block stored on VM 1-a, scheduling the task on VM 1-b or VM 2-b are considered identical by the task scheduler, for both are rack-local. Yet data exchanges between co-located VMs (e.g., VM 1-a and VM 1-b) are much faster than those between remote VMs (e.g., VM 1-b and VM 2-b), and hence the two cases should have different scheduling priorities. As such, we modify the original three-level priority

scheduling strategy by splitting node-local into two priority levels, namely, *VM-local* and *PM-local*, in virtualized MapReduce clusters, defined as follows:

Definition 1. ***VM-local****. A task and its required data are on the same virtual machine;*

Definition 2. ***PM-local****. A task and its required data are on two co-located virtual machines, respectively.*

At the beginning of running a MapReduce job, DALM launches a topology configuration process that identifies the structure of the underlying virtualized clusters from a configuration file. The points in the same or nearest communities in the graph are able to be placed on the same data block, so that they will be dealt with in the same computation node. A VM is associated with a unique ID, in the format of *rack-PM-VM*, such that the degree of locality can be easily obtained. The scheduling algorithm then schedules tasks in a virtualized MapReduce cluster based on the newly designed priority-levels. The key change of this scheduler in Algorithm 2 is that, when a VM has free slots, it requests new tasks from the master node through heartbeat, and the task scheduler on the master node assigns tasks according to the following priority order: VM-local, PM-local, rack-local, and off-switch.

Algorithm 2 vScheduling: Task scheduling in DALM MapReduce systems

```
1: when the task scheduler receives a heartbeat from node v:
2:   if v has a free slot then
3:     sort the pending tasks according to the priority policy and obtain a list T
4:     schedule the first task to node v
5:   end if
```

3.3 Implementation Details

The above high-level description provides a sketch of the DALM's workflow. Implementing it in real-world MapReduce systems however is nontrivial. In particular, we need to modify the original priority level as well as the corresponding scheduling policy, calling for careful examination of the whole Hadoop package to identify the related classes and methods. We also need to ensure that our modifications are compatible with other modules.

To demonstrate the practicability of DALM and investigate its performance in real-world MapReduce clusters, we have implemented the prototype of DALM based on Hadoop 1.2.1 and Apache Giraph 1.0.0 by extending the Hadoop Distributed File System (HDFS) with the proposed adaptive replication scheme. We have also modified the existing job scheduling strategy to be virtualization-aware. As we have re-defined the locality levels, and revised the task scheduling policy accordingly, we carefully examine the whole Hadoop package to identify the related source codes, and make necessary modifications. We ensure that our modifications are compatible with the remaining modules. We now highlight the key modifications in the practical implementation in the following.

```
Public synchronized List <Task>
assignTasks (TaskTracker taskTracker) {
List<Task> assignedTasks = new ArrayList<Task>();
for (int i=0; i < availableMapSlots; ++i) {
    synchronized (jobQueue) {
        for (JobInProgress job : jobQueue) {
            Task t = null;
            t = job.obtainNewLocalMapTask();
            if (t != null) {
               assignedTasks.add(t);
               break;
            }
            t = job.obtainNewNonLocalMapTask();
            if (t != null) {
               assignedTasks.add(t);
               break;
               break scheduleMaps;
            }
        }
    }
}
```

(a)

```
Public synchronized List <Task>
assignTasks (TaskTracker taskTracker) {
List<Task> assignedTasks = new ArrayList<Task>();
for (int i=0; i < availableMapSlots; ++i) {
    …
            t = job.obtainNewVmOrPmLocalMapTask();
            if (t != null) {
               assignedTasks.add(t);
               break;
            }
            t = job.obtainNewRackLocalMapTask();
            if (t != null) {
               assignedTasks.add(t);
               break;
            }
            t = job.obtainNewNonLocalMapTask();
            if (t != null) {
               assignedTasks.add(t);
               break;
               break scheduleMaps;
            }
    …
}
```

(b)

Figure 3.4: Task scheduler in Standard Hadoop (a) and DALM (b)

First, we modify the original `NODE-LOCAL` to `PM-LOCAL` and `VM-LOCAL` as follows. In the source code `Locality.java`, `NODE_LOCAL` is replaced by `VM_LOCAL` or `PM_LOCAL` depending on whether a task is launched on the local VM or on a co-located VM. The `NetworkTopology` class in `NetworkTopology.java`, which defines the hierarchical tree topology of MapReduce clusters, is extended to `VirtualNetworkTopology` by overriding such methods as `getDistance()` and `pseudoSortByDistance()`. During the initialization of NameNode, `VirtualNetworkTopology` returns `clusterMap`, which is the network topology, to the DataNodeManager. In DALM, the distance calculation is updated to VM-local (0), PM-local (1), rack-local (2), and off-switch (4).

Second, we modify the task scheduling algorithm to be virtualization aware, which is mainly implemented in `JobInProgress.java` and `JobQueueTaskScheduler.java`. In the standard Hadoop and DALM systems, the task scheduler works as shown in Figure 3.4(a) and Figure 3.4(b), respectively. We modify the original `JobQueueTaskScheduler()` method to respond to TaskTracker heartbeat requests. The `getMachineLevelForNodes()` method in the source code `JobInProgess.java` is the key component of the task scheduler. In our implementation of DALM, the task scheduler in DALM works as shown in Figure 3.4(b): it first assigns VM-local and PM-local tasks through `obtainVmOrPmLocalMapTask()`, then rack-local tasks through

`obtainRackLocalMapTask()`, and finally other tasks through `obtainNonLocalMapTask()`. All these three methods are encapsulated in a general method, namely, `obtainNewMapTask()` in `JobInProgress.class`. We keep the original interfaces of `obtainNewMapTask()` to ensure the compatibility.

After making the above modifications, we re-compile the whole package to get the executable `.jar` file, which is then deployed on the MapReduce cluster.

To reduce the cross-server network traffic during the job execution, the task scheduler on the master node usually places a task onto the slave, on which the required input data is available if possible. This is not always successful since the slave nodes having the input data may not have free slots at that time. Recall that the default replication factor is three in the Hadoop systems, which means that each file block is stored on three servers–two of them are within the same rack and the remaining one is in a different rack. Hence, depending on the distance between DataNode and TaksTracker, the default Hadoop defines three levels of data locality, namely, *node-local*, *rack-local*, and *off-switch*. Node-local is the optimal case that the task is scheduled on the node having data. Rack-local represents a suboptimal case that the task is scheduled on a node different from the node having data; the two nodes are within the same rack. Rack-local will incur cross-server traffic. Off-switch is the worst case, which both the node-local and rack-local nodes have no free slots, and thus the task needs to retrieve data from a node in a different rack, incurring both cross-server and cross-rack traffic. When scheduling a task, the task scheduler takes a priority-based strategy: node-local has the highest priority, whereas off-switch has the lowest.

At the beginning of running a MapReduce job, DALM launches a topology configuration process that identifies the structure of the underlying virtualized clusters from a configuration file. The points in the same or nearest communities in the graph are able to be placed on the same data block, so that they will be dealt with in the same computation node. Then a VM is associated with a unique ID, in the format of *rack-PM-VM*, such that the degree of locality can be easily obtained. A customized scheduling algorithm then schedules tasks in a virtualized MapReduce cluster based on the newly designed priority-levels. The main idea of the scheduling algorithm is as follows. When a VM has free slots, it requests new tasks from the master node through heartbeat, and the task scheduler on the master node assigns tasks according to the following priority order: VM-local, PM-local, rack-local, and off-switch. To demonstrate the practicability of DALM and investigate its performance in real world MapReduce clusters, we have implemented the prototype of DALM by extending the Hadoop Distributed File System (HDFS) with the proposed adaptive replication scheme, and modifying the existing job scheduling strategy to be virtual-aware, based on Hadoop 1.2.1 and Apache Giraph 1.0.0. As we have re-defined the locality levels, and revised the task scheduling policy accordingly, we carefully examine the whole Hadoop package to identify the related relevant source codes, and make necessary modifications. We ensure that our modifications are compatible with the remaining modules. We now highlight the key modifications in the practical implementation in the following.

First, we modify the original `NODE-LOCAL` to `PM-LOCAL` and `VM-LOCAL` as follows.

In `Locality.java`, `NODE_LOCAL` is replaced by `VM_LOCAL` or `PM_LOCAL` depending on whether a task is launched on the local VM or on a co-located VM. The `NetworkTopology` class in `NetworkTopology.java`, which defines the hierarchical tree topology of MapReduce clusters, is extended to `VirtualNetworkTopology` by overriding such methods as `getDistance()` and `pseudoSortByDistance()`.

During the initialization of NameNode, `VirtualNetworkTopology` returns `clusterMap`, which is the network topology, to the DataNodeManager. In DALM, the distance calculation is updated to VM-local (0), PM-local (1), rack-local (2), and off-switch (4).

Second, we modify the task scheduling algorithm to be virtualization aware, which is mainly implemented in `JobInProgress.java` and `JobQueueTaskScheduler.java`. In the standard Hadoop systems, the task scheduler works as shown in Figure 3.4(a). We modify the original `JobQueueTaskScheduler()` method to respond to TaskTracker heartbeat requests. The `getMachineLevelForNodes()` method in `JobInProgess.java` is the key component of the task scheduler. In our implementation of DALM, the task scheduler in DALM works as shown in Figure 3.4(a): it first assigns VM-local and PM-local tasks through `obtainVmOrPmLocalMapTask()`, then rack-local tasks through `obtainRackLocalMapTask()`, and finally other tasks through `obtainNonLocalMapTask()`. All these three methods are encapsulated in a general method, namely, `obtainNewMapTask()` in `JobInProgress.class`. We keep the original interfaces of `obtainNewMapTask()` to ensure the compatibility.

After making the above modifications, we re-compile the whole package to get the executable `jar` file, which is then deployed on the MapReduce clusters.

So far we have outlined the design of DALM and laid its algorithmic foundations. We now discuss the practical challenges toward deploying DALM in real-world cloud environment, and also present our prototype implementation.

Chapter 4

Performance Evaluation

In this chapter, the performance evaluation includes two sets of algorithms: online load balancing algorithms and datacenter network-aware load balancing algorithms. We will show the performance evaluation results of comparing our online load balancing algorithms with the state-of-the-art algorithms in section 4.1 and the results of our datacenter network-aware load balancing algorithms in section 4.2.

4.1 Dependency-aware Data Locality

In this section, we provide the performance evaluation of DALM, through extensive simulations and testbed experiments. We compare DALM and several state-of-the-art data locality solutions

4.1.1 Simulations

We first conduct a series of simulations to examine the effectiveness of the proposed dependency-based replication strategy in general distributed MapReduce systems. As in previous studies [38], we assume that the file popularity follows the Pareto distribution [41]. Our approach can easily adapt to 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. We plot the relative file popularity (the percentage of accesses of each file in the total accesses of all the files) as a function of the rank of the file for a sample set of 50 files in Figure 4.1. 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.

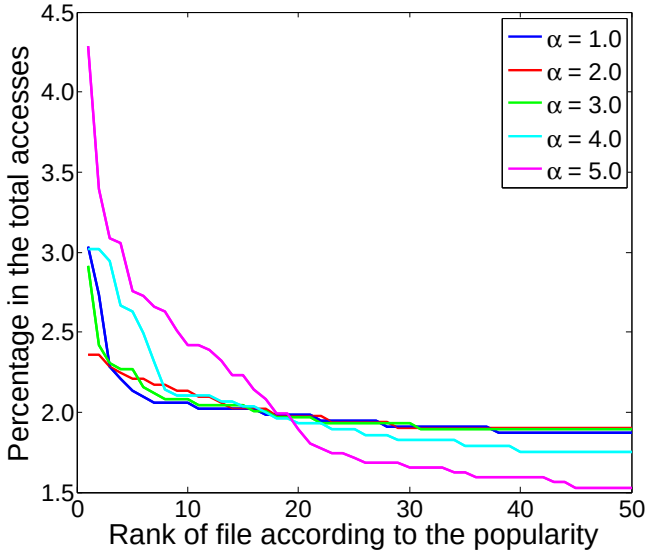


Figure 4.1: File rank ordered by relative popularity

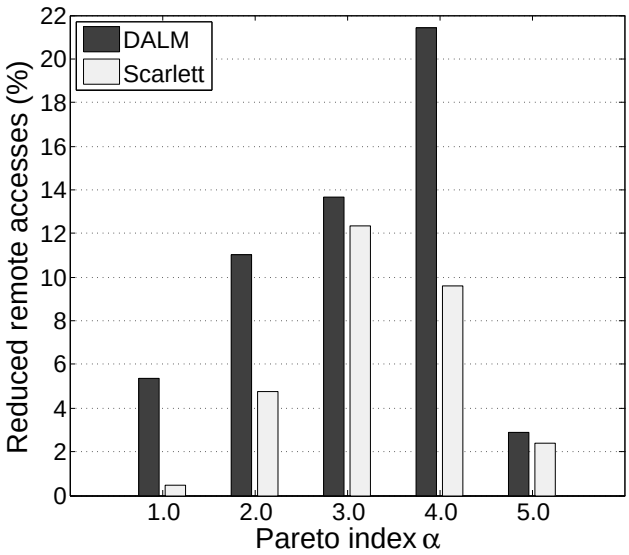


Figure 4.2: Impact of the skewness of file popularity

We compare our dependency-based replication strategy (DALM) with the popularity-based Scarlett [5]. 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 4.2-4.5, for each of the above system parameters, respectively. In general, we find that DALM outperforms the other two approaches with significantly reduced remote accesses. In the following we present our detailed analysis with respect to each system parameter.

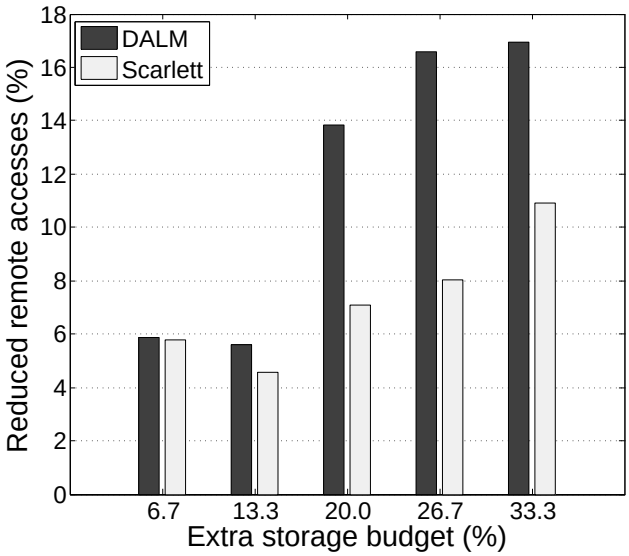


Figure 4.3: Impact of the extra storage budget ratio

Figure 4.2 shows the impact of the skewness of file popularity on the reduction of remote access, with the 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 $\alpha = 5.0$, 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, with less improvement over the baseline.

Figure 4.3 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. Therefore, 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%, which reduces the remote access substantially by 22%, compared with 7% of Scallett. It is noting that the gap between DALM and Scallett will slightly decrease when the budget ratio keeps growing. The reason is that with more extra storage, Scallett will have more replicas, and thus dependent files would be stored together with higher probability.

With a storage budget of 20%, the impact of the upper bound of the replication factor is shown in Figure 4.4. 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

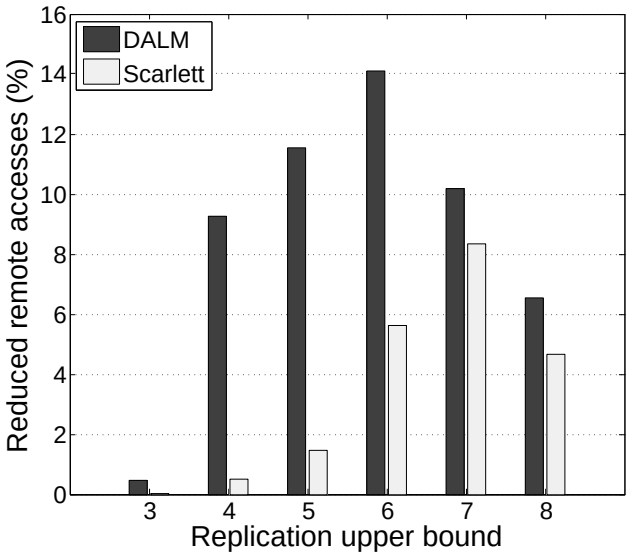


Figure 4.4: Impact of the upper bound of the replication factor

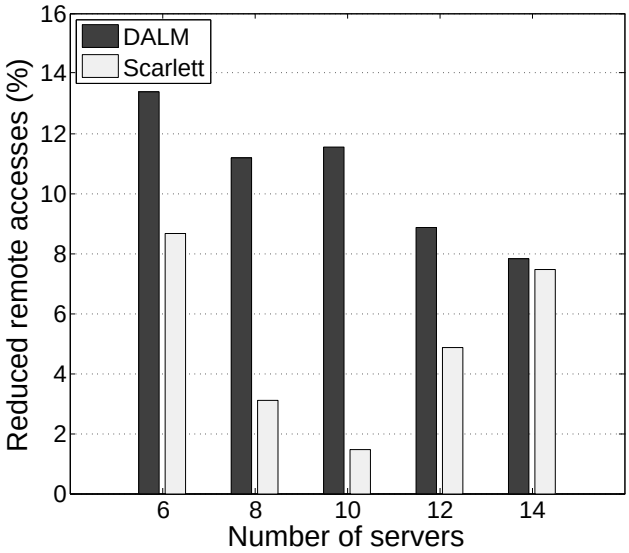


Figure 4.5: Impact of system scale

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 4.5 shows the impact of the number of servers. Interestingly, the improvement of DALM becomes less with more servers, though DALM still noticeably 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 expect 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.

4.1.2 Experiments

Experiment Setup

We next present the real-world experiment results in a representative virtualized environment. Our testbed consists of 4 Dell servers (OPTIPLEX 7010), each equipped with an Intel Core i7-3770 3.4 GHz quad core CPU, 16 GB 1333 MHz DDR3 RAM, a 1 TB 7200 RPM hard drive, and a 1 Gbps Network Interface Card (NIC). Hyper-Threading is enabled for the CPUs so that each CPU core can support two threads. All physical machines are inter-connected by a NETGEAR 8-port gigabit switch. This fully controllable testbed system allows the machines to be interconnected with the maximum speed, and enables us to closely examine the impact of data dependency and data locality. We use the open source monitoring system Ganglia[1] 3.5.12 to monitor the system running conditions.

We compare DALM with Scarlett and Surfer, as well as the default Hadoop system, in terms of job completion time, and cross-server traffic. We use a public social network data set [53], which contains 8,164,015 vertices and 153,112,820 edges, and the volume is 2.10 GB. There are roughly five communities in the graph, and we select one of them as the popular one. 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 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. We run two representative applications to evaluate the performance of these systems, namely, *shortest path* and *PageRank*. The `ShortestPath` application finds the shortest path for all pairs of vertices; the `PageRank` application ranks the vertices according to the PageRank algorithm.

Experiment Results

A. Job Finish Time

We first compare the job finish time of different systems. For each system, we run each benchmark application five times, and plot the average improvement of job finish time in Figure 4.6. The default Hadoop system serves as the baseline.

It can be observed that DALM significantly improves the system performance in the job finish time over the other systems. Compared with the default Hadoop, Scarlett, Surfer, our proposed DALM on average improves the job finish time by 10.74%, 12.40%, and 28.09% for *shortest path*, respectively. For *PageRank*, the improvements are 6.74%, 16.37%, and 17.80%, respectively. These results clearly demonstrates all the three systems can significantly reduce the job finish time over the default Hadoop system. Scarlett improves data locality by smartly replicating popular blocks, while Surfer can reduce cross-server traffic with dependency-aware placement. DALM achieves the shortest job finish time by integrating the dependency-aware placement strategy of Surfer, and the popularity-based replication strategy of Scarlett. Further,

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

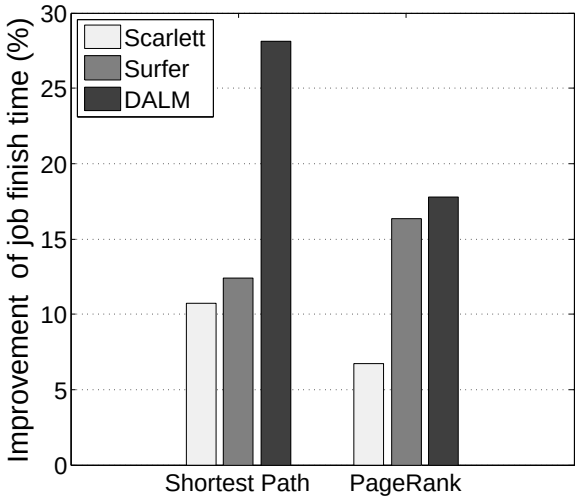


Figure 4.6: Average Job Completion Time

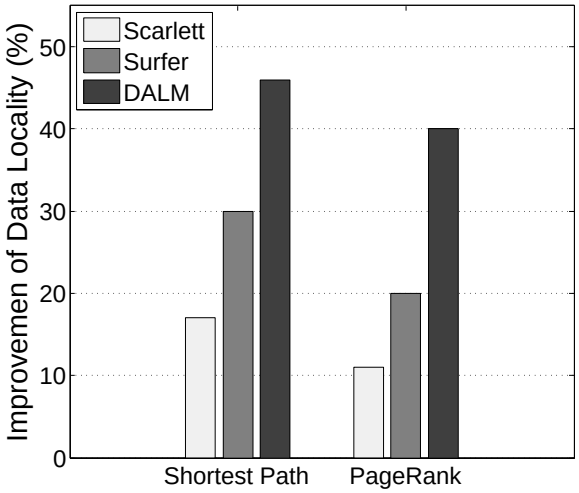


Figure 4.7: Data Locality of Jobs

DALM adapts the task scheduling algorithm to be virtual aware, which has the potential to further improve job finish time in larger scale systems.

B. Data Locality and Cross-server Traffic

We now take a closer look at how DALM achieves the best performance. Data locality is an important metric in MapReduce systems, since it largely reflects the data traffic pressure on the network fabric. Data locality is highly desirable for MapReduce applications, especially in commercial data centers where the network fabrics are highly oversubscribed, and the communications across racks are generally costly. Figure 4.7 shows that DALM achieves remarkably improvement in data locality, as compared to Scarlett and Surfer for both benchmark applications. Since Scarlett has more replicas for popular data blocks than Surfer, the data locality of Scarlett is higher. With both popularity and dependency-based schemes, DALM is able to provide 100% data locality.

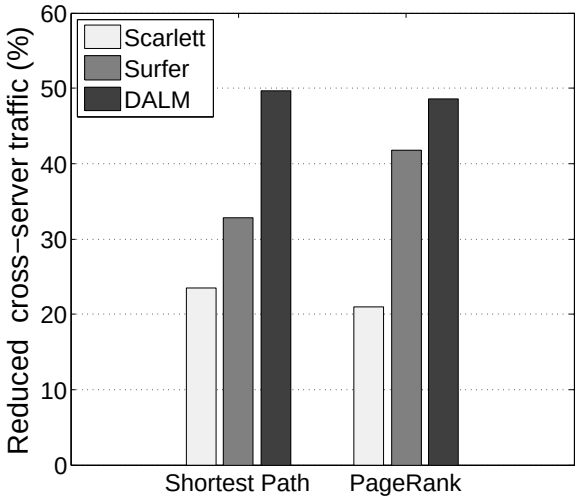


Figure 4.8: Cross-server Traffic Size (MB)

Figure 4.8 explains the advantages of DALM over Hadoop, Scarlett, and Surfer, from the perspective of cross-server traffic. In the default Hadoop, only half of the tasks are localized, which means there is considerable overhead caused by remote task executions and communications. Scarlett and Surfer successfully reduce the cross-server traffic with more replications and dependency-aware data placement, respectively. Yet Scarlett can evict strong dependent tasks on remote VMs, leading to increased task finish time. Surfer, on the other hand, successfully reduces a large amount of remote communications by placing dependent tasks on nearby VMs, but incurs the hotspot problem for popular data blocks. In DALM, most tasks are assigned to the appropriate VMs storing the necessary data blocks, minimizing the out-of-rack tasks. The hot spot problems are also largely avoided through data replication.

4.2 DALM Performance in Virtualized Environment

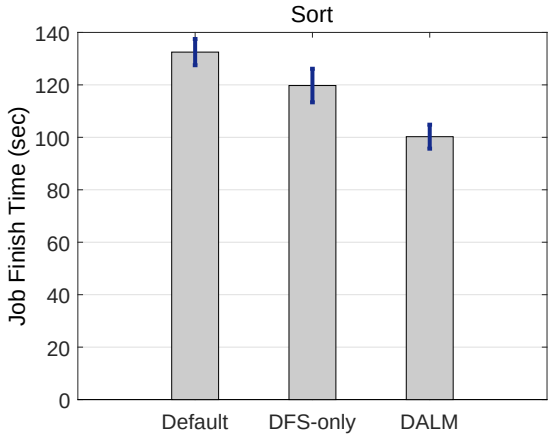
In this section, we evaluate the performance of DALM based on our testbed virtualized cloud platform. Our platform consists of eight physical machines, which are inter-connected through a gigabit switch. machines. We use a separate physical machine as the master node, which, as the central controller, involves heavy I/O operations and network communications to maintain the state of the whole cluster and schedule tasks. Using a dedicated physical machine, rather than a virtual machine, ensures fast response time with minimized resource contention, and accordingly enables a fair comparison with fully non-virtualized systems. Other physical machines that host virtual slave nodes run the latest Xen Hypervisor (version 4.1.3). On each physical machine, besides the Domain0 VM, we configure three identical DomainU VMs, which act as slave nodes. For the operating systems running on VMs (both Domain0 and DomainU), we use the popular Ubuntu 12.04 LTS 64 bit (kernel version 3.11.0-12). All the VMs are allocated two virtual CPUs and 4 GB memory. We use the *logical volume management* (LVM) system, which is convenient for on demand resizing, to allocate each DomainU VM a 100 GB disk space in default.

Hence, our virtualized MapReduce cluster has twenty-two nodes in total, namely, one physical master node and twenty-one virtual slave nodes. We compare DALM integrating to the vScheduler with two other systems, *Default* and *DFS-only* (DALM File System only). Default runs the standard Hadoop 1.2.1 system with one DataNode on each virtual slave node, which serves as the baseline; DFS-only only adopts the storage strategy of DALM. We select three widely used Hadoop benchmark applications `Sort`, `ShortestPath`, and `PageRank`. We use the 3.5 GB wikipedia data as the input for `Sort`, and process the data to the input format (directed graph) of `ShortestPath` and `PageRank`.[2] The `PageRank` application ranks the vertices according to the PageRank algorithm, and the `ShortestPath` application finds the shortest path for all pairs of vertices in the input file.

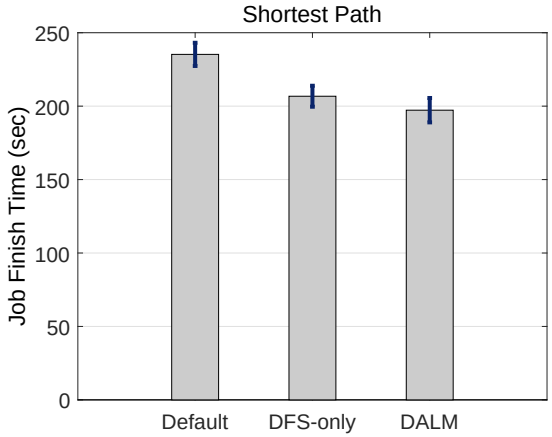
We compare the job finish time of different systems. For each system, we run the experiments for each benchmark application five times, and plot the average job finish time, as well as the standard deviation in Fig. 4.9. It is observed that DALM significantly improves the performance of all the selected MapReduce applications over the other systems. Compared with Default, DFS-only improves the job finish time by 9.6% on average for `Sort`, which again verifies the effectiveness of revising the DataNote placement strategy. DALM, by adapting the task scheduling algorithm to be virtual aware, can further improve the job finish time by 16.3% on average, as compared with DFS-only (24.3% as directly compared with Default). The improvements on `ShortestPath` and `PageRank` are less significant (16.2% and 12.3% as compared with Default, respectively), since the number of reducers in these two applications is less than that in `Sort`, and thus the reduce phase, in which data locality has less impact, accounts for most of the running time. Further, these two applications involve iterative steps, which incur a lot of communication overhead.

Since data locality is mainly related to the map phase, we plot of the empirical CDF of the map task finish time in Fig. 4.10, which gives us a much more clearer picture of the impressive improvements of DALM over Default. We can see that the system design has a remarkable impact on the finish time of individual map tasks. Taking `Sort` as an example, in the Default system, all the map tasks take more than 20 seconds, and nearly 30% tasks need more than 40 seconds. On the other hand, in both DFS-only and DALM systems, more than 70% tasks can be finished in less than 18 seconds, indicating that the reduced number of DataNodes can effectively mitigate the interference and speed up task execution. The remaining tasks, accounting for about 20% of the total, have divergent finish time distributions in DFS-only and DALM: all the remaining tasks can be finished within 40 seconds in DALM, while most of them need more than 40 seconds in DFS-only, implying that vScheduling significantly reduces rack-local (cross-PM) tasks.

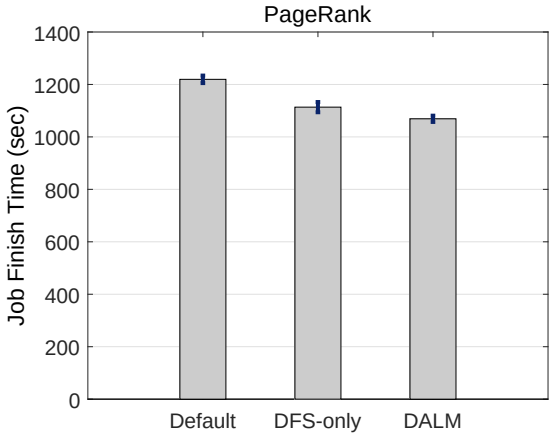
[2]Each URL in the file is represented by a vertex, and each hyperlink is represented by a directed edge.



(a)



(b)



(c)

Figure 4.9: Job finish time of different systems

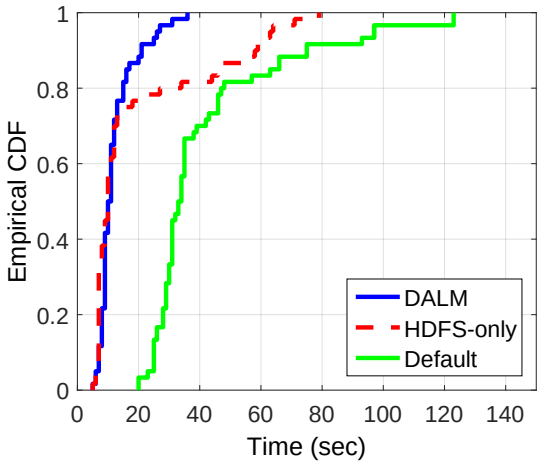


Figure 4.10: Empirical CDF of map task finish time of different systems

Chapter 5

Discussion and Conclusion

5.1 Discussion

As an important practice to improve MapReduce performance, data locality has been extensively investigated in physical machine clusters. In this thesis, we showed strong evidence that the conventional efforts on improving data locality can cause negative impact on typical MapReduce applications in virtualized environments. By examining the inter-VM contention for the shared resources, we suggested to adapt the existing storage architecture to be virtual-machine-aware, which offloads a large portion of congested disk I/O to the highly efficient network I/O with memory sharing. This new storage design demands revision on the traditional three-level data locality, so does the task scheduling. To this end, we developed DALM, a systematic solution to improve data locality in virtualized MapReduce clusters. Through real-world implementation and deployment of DALM, we demonstrated the superiority of DALM against state-of-the-art Hadoop systems. There is one potential limit of DALM that it requires the topology information of the underlying cluster, which is not readily available in current public clouds. Yet, we believe that this information will be provided by cloud providers in the near future to facilitate cloud users, to further improve the performance through topology-aware resource provisioning.

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.

Recently, cluster or community detection in large-scale social graphs has also attracted remarkable attention [37, 31, 53]. The detection algorithms group nodes having more linked edges together, and could be used in DALM's preprocessing module. Unfortunately, they generally divide graphs into communities consisting of up to hundreds of nodes [37]. Considering that the size of file blocks in HDFS is 64 MB by default, the granularity of communities can be too small for data storage. The size of communities can be highly heterogeneous across different communities, too, making it difficult to use a community as the basic storage unit. We expect more efficient and flexible community detection algorithms to be developed and possibly incorporated into DALM in the future.

For the future work, we will explore the following directions. First, we plan to further optimize the two basic components of DALM: for the storage design, we can incorporate the file popularity issue to balance the workloads of individual DataNodes; for the task scheduling, we may incorporate other advanced strategies, e.g., the Delay Scheduling [55] to the virtualized environment. Second, our preliminary DALM design in this thesis focuses on a single user scenario. In the multi-user scenario, the fairness among users should be addressed too for storage design and task scheduling. Third, we will investigate the possibility of more flexible resource allocation in DALM. Given that not all VMs have DataNodes, the workloads of different types of VMs can be different, leading to a heterogeneous environment. We plan to conduct a more detailed profiling analysis to identify the potential performance bottleneck of the DALM design, and compare both offline solutions that allocate different amounts of resources (e.g., CPU and memory) based on workloads, and online solutions that dynamically adjusts the capabilities of VMs through the Xen control interfaces.

5.2 Conclusion

In this thesis, we have showed strong evidence that data dependency, which exists in many real-world applications, has a significant impact on the performance of typical MapReduce applications. 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. Taking graph data as a representative case study, we have illustrated that traditional replication strategies can store highly dependent data on different servers, leading to massive remote data accesses. To this end, we have developed DALM, a comprehensive and practical solution toward dependency-aware locality for MapReduce in virtualized environments. DALM first leverages the graph partition technique to identify the dependency of the input data, and reorganize the data storage accordingly. It then replicates data blocks according to the historical data popularity, and smartly places data blocks across VMs subject to such practical constraints as storage budget, replication upper bound, and the underlying network topology. Through both

simulations and real-world deployment of the DALM prototype, we illustrated the superiority of DALM against state-of-the-art solutions.

It is worth noting that cross-server traffic still exists in DALM. For example, if the VMs on one PM are all busy and have no free slots, while some VMs on other PMs have free slots, the task scheduler has to allocate the pending tasks. This could be improved through incorporating delay scheduling [55] into DALM. DALM also needs the topology information of the underlying clusters for data placement. If such information is not available, a topology inference algorithm might be needed, e.g., through bandwidth/latency measurement between VM pairs.

Bibliography

[1] Amazon web services. `http://aws.amazon.com/`.

[2] Apache hadoop. `http://hadoop.apache.org/`.

[3] Rackspace. `http://www.rackspace.com/`.

[4] Cristina L Abad, Yi Lu, and Roy H Campbell. Dare: Adaptive data replication for efficient cluster scheduling. In *Proceedings of IEEE CLUSTER*, pages 159–168, 2011.

[5] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proceedings of EuroSys*, pages 287–300, 2011.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of USENIX NSDI*, 2012.

[7] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[9] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM IMC*, Melbourne, Australia, 2010.

[10] Xiangping Bu, Jia Rao, and Cheng-zhong Xu. Interference and locality-aware task scheduling for mapreduce applications in virtual clusters. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 227–238. ACM, 2013.

[11] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[12] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. Trill: a high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[13] Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 3. ACM, 2012.

[14] Cédric Chevalier and Ilya Safro. Comparison of coarsening schemes for multilevel graph partitioning. In *Learning and Intelligent Optimization*, pages 191–205. Springer, 2009.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of ACM HDPC*, 2010.

[17] Jun Fang, Shoubao Yang, Wenyu Zhou, and Hu Song. Evaluating i/o scheduler in virtual machines for mapreduce application. In *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, pages 64–69. IEEE, 2010.

[18] Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation, 1982. 19th Conference on*, pages 175–181. IEEE, 1982.

[19] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[20] Yifeng Geng, Shimin Chen, YongWei Wu, Ryan Wu, Guangwen Yang, and Weimin Zheng. Location-aware mapreduce in virtual cloud. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 275–284. IEEE, 2011.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[22] Zhenhua Guo, Geoffrey Fox, and Mo Zhou. Investigation of data locality in mapreduce. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 419–426. IEEE Computer Society, 2012.

[23] Mohammad Hammoud and Majd F Sakr. Locality-aware reduce task scheduling for mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 570–576. IEEE, 2011.

[24] Bruce Hendrickson and Robert Leland. A multi-level algorithm for partitioning graphs. 1995.

[25] Shadi Ibrahim, Hai Jin, Lu Lu, Bingsheng He, and Song Wu. Adaptive disk i/o scheduling for mapreduce in virtualized environment. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 335–344. IEEE, 2011.

[26] Shadi Ibrahim, Hai Jin, Lu Lu, Li Qi, Song Wu, and Xuanhua Shi. Evaluating mapreduce on virtual machines: The hadoop case. In *Cloud Computing*, pages 519–528. Springer, 2009.

[27] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of ACM SOSP*, SOSP '09, pages 261–276, 2009.

[28] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[29] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 9th ACM IMC*, Chicago, Illinois, USA, 2009.

[30] Hui Kang, Yao Chen, Jennifer L Wong, Radu Sion, and Jason Wu. Enhancement of xen's scheduler for mapreduce workloads. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 251–262. ACM, 2011.

[31] Brian Karrer and Mark EJ Newman. Stochastic blockmodels and community structure in networks. *Physical Review E*, 83(1):016107, 2011.

[32] G Karypis and V Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs, department of computer science tech. rep. 96-036. *University of Minnesota, Minneapolis, MN*, 1996.

[33] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 29. ACM, 1995.

[34] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[35] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[36] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[37] Jure Leskovec, Kevin J Lang, and Michael Mahoney. Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on World wide web*, pages 631–640. ACM, 2010.

[38] Jimmy 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.

[39] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of ACM SIGMOD*, pages 135–146, 2010.

[40] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[41] MEJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46(5):323–351, 2005.

[42] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[43] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[44] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan 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.

[45] Hae-Sang Park and Chi-Hyuck Jun. A simple and fast algorithm for k-medoids clustering. *Expert Systems with Applications*, 36(2, Part 2):3336–3341, 2009.

[46] Jongse Park, Daewoo Lee, Bokyeong Kim, Jaehyuk Huh, and Seungryoul Maeng. Locality-aware dynamic vm reconfiguration on mapreduce clouds. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 27–36. ACM, 2012.

[47] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

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

[49] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2015.

[50] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726. IEEE, 2010.

[51] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 1013–1020. ACM, 2010.

[52] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[53] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[54] Yi Yuan, Haiyang Wang, Dan Wang, and Jiangchuan Liu. On interference-aware provisioning for cloud-based big data processing. In *Quality of Service (IWQoS), 2013 IEEE/ACM 21st International Symposium on*, pages 1–6. IEEE, 2013.

[55] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys*, pages 265–278, 2010.

[56] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[57] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

[58] Xiaohong Zhang, Zhiyong Zhong, Shengzhong Feng, Bibo Tu, and Jianping Fan. Improving data locality of mapreduce by scheduling in homogeneous computing environments. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 120–126. IEEE, 2011.