Scalable data-management systems for Big Data

Viet-Trung Tran

THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN
Mention : Informatique
École doctorale MATISSE

présentée par

Viet-Trung TRAN

Préparée à l'Unité Mixte de Recherche n° 6074
Institut de recherche en informatique
et systèmes aléatoires

Scalable data management systems for Big Data

Thèse soutenue le 21 janvier 2013
devant le jury composé de :

Franck CAPPELLO
Directeur de recherche, INRIA, Saclay, France / *rapporteur*

Pierre SENS
Professeur, Université de Paris 6, France / *rapporteur*

Guillaume PIERRE
Professeur, Université de Rennes 1, France / *examinateur*

Patrick VALDURIEZ
Directeur de recherche, INRIA Sophia, France / *examinateur*

Dushyanth NARAYANAN
Chercheur, Microsoft Research Cambridge, Angleterre / *examinateur invité*

Luc BOUGÉ
Professeur, ENS Cachan Antenne de Bretagne, France / *directeur de thèse*

Gabriel ANTONIU
Directeur de recherche, INRIA, Rennes, France / *directeur de thèse*

Résumé

La problématique «Big Data» peut être caractérisée par trois «V»:
- «Big Volume» se rapporte à l'augmentation sans précédent du volume des données.
- «Big Velocity» se réfère à la croissance de la vitesse à laquelle ces données sont déplacées entre les systèmes qui les gèrent.
- «Big Variety» correspond à la diversification des formats de ces données.

Ces caractéristiques imposent des changements fondamentaux dans l'architecture des systèmes de gestion de données. Les systèmes de stockage doivent être adaptés à la croissance des données, et se doivent de passer à l'échelle tout en maintenant un accès à hautes performances. Cette thèse se concentre sur la construction des systèmes de gestion de grandes masses de données passant à l'échelle.

Les deux premières contributions ont pour objectif de fournir un support efficace des «Big Volumes» pour les applications data-intensives dans les environnements de calcul à hautes performances (HPC). Nous abordons en particulier les limitations des approches existantes dans leur gestion des opérations d'entrées/sorties (E/S) non-contiguës atomiques à large échelle. Un mécanisme basé sur les versions est alors proposé, et qui peut être utilisé pour l'isolation des E/S non-contiguës sans le fardeau de synchronisations coûteuses. Dans le contexte du traitement parallèle de tableaux multi-dimensionels en HPC, nous présentons Pyramid, un système de stockage large-échelle optimisé pour ce type de données. Pyramid revoit l'organisation physique des données dans les systèmes de stockage distribués en vue d'un passage à l'échelle des performances. Pyramid favorise un partitionnement multi-dimensionel de données correspondant le plus possible aux accès générés par les applications. Il se base également sur une gestion distribuée des métadonnées et un mécanisme de versioning pour la résolution des accès concurrents, ce afin d'éliminer tout besoin de synchronisation.

Notre troisième contribution aborde le problème «Big Volume» à l'échelle d'un environnement géographiquement distribué. Nous considérons BlobSeer, un service distribué de gestion de données orienté «versioning», et nous proposons BlobSeer-WAN, une extension de BlobSeer optimisée pour un tel environnement. BlobSeer-WAN prend en compte la hiérarchie de latence et favorise les accès aux méta-données locales. BlobSeer-WAN inclut la réplication asynchrone des méta-données et une résolution des collisions basée sur des «vector-clock».

Afin de traîter le caractère «Big Velocity» de la problématique «Big Data», notre dernière contribution consiste en DStore, un système de stockage en mémoire orienté «documents» qui passe à l'échelle verticalement en exploitant les capacités mémoires des machines multi-coeurs. Nous montrons l'efficacité de DStore dans le cadre du traitement de requêtes d'écritures atomiques complexes tout en maintenant un haut débit d'accès en lecture. DStore suit un modèle d'exécution mono-thread qui met à jour les transactions séquentiellement, tout en se basant sur une gestion de la concurrence basée sur le versioning afin de permettre un grand nombre d'accès simultanés en lecture.

Abstract

Big Data can be characterized by 3 V's.
• Big Volume refers to the unprecedented growth in the amount of data.
• Big Velocity refers to the growth in the speed of moving data in and out management systems.
• Big Variety refers to the growth in the number of different data formats.

Managing Big Data requires fundamental changes in the architecture of data management systems. Data storage should continue being innovated in order to adapt to the growth of data. They need to be scalable while maintaining high performance regarding data accesses. This thesis focuses on building scalable data management systems for Big Data.

Our first and second contributions address the challenge of providing efficient support for Big Volume of data in data-intensive high performance computing (HPC) environments. Particularly, we address the shortcoming of existing approaches to handle atomic, non-contiguous I/O operations in a scalable fashion. We propose and implement a versioning-based mechanism that can be leveraged to offer isolation for non-contiguous I/O without the need to perform expensive synchronizations. In the context of parallel array processing in HPC, we introduce Pyramid, a large-scale, array-oriented storage system. It revisits the physical organization of data in distributed storage systems for scalable performance. Pyramid favors multidimensional-aware data chunking, that closely matches the access patterns generated by applications. Pyramid also favors a distributed metadata management and a versioning concurrency control to eliminate synchronizations in concurrency.

Our third contribution addresses Big Volume at the scale of the geographically distributed environments. We consider BlobSeer, a distributed versioning-oriented data management service, and we propose BlobSeer-WAN, an extension of BlobSeer optimized for such geographically distributed environments. BlobSeer-WAN takes into account the latency hierarchy by favoring locally metadata accesses. BlobSeer-WAN features asynchronous metadata replication and a vector-clock implementation for collision resolution.

To cope with the Big Velocity characteristic of Big Data, our last contribution feautures DStore, an in-memory document-oriented store that scale vertically by leveraging large memory capability in multicore machines. DStore demonstrates fast and atomic complex transaction processing in data writing, while maintaining high throughput read access. DStore follows a single-threaded execution model to execute update transactions sequentially, while relying on a versioning concurrency control to enable a large number of simultaneous readers.

ENSC - 2013 n°.....

École normale supérieure de Cachan - Antenne de Bretagne

Campus de Ker Lann - Avenue Robert Schuman - 35170 BRUZ

Tél : +33(0)2 99 05 93 00 - Fax : +33(0)2 99 05 93 29

The completion of this thesis was made possible through the patience and guidance of my advisor, Gabriel and Luc. I am grateful for their support and encouragements not only in my research life but also in my social life. Thank you Luc and Gabriel for giving me the opportunity to pursue a Master and then a PhD in their research team.

I wish to express my gratitude to my thesis reviewers, Franck CAPPELLO, Pierre SENS and the members of my dissertation committee, Guillaume PIERRE, Patrick VALDURIEZ and Dushyanth NARAYANAN. Thank you for accepting to review my manuscript and for the insightful comments and careful consideration given to each aspect of my thesis. In particular, many thanks to Dushyanth for hosting me at Microsoft Research Cambridge, UK as an intern within the Systems and Networking Group for a duration of three months. During this stay, I had the change to broaden my knowledge in the area of in-memory Big Data analytics.

I have been fortunate enough to have the help and support of many friends and colleagues, especially all of my colleagues in KerData team. I will not be listing all of their particular support in here as my manuscript will end up in thousands pages. Thanks Bogdan Nicolae for co-working with me for the first two contributions of this thesis. Thanks Matthieu Dorier for translating the abstract of this manuscript into French. Thanks Alexandru Farcasanu for working with me in running some experiments of DStore in the Grid'5000 testbed.

I cannot find words to express my gratitude for the girls, Alexandra, Diana and Izabela. Those are the angels of mine in France and I wish they will alway be forever.

My deepest thanks go to my family, for their unconditional support and encouragements even from a very far distance.

Finally, many thanks to all other people that had a direct or indirect contribution to this work and were not explicitly mentioned above. Your help and support is very much appreciated.

To myself
Embrace the dreams and be happy in life.

Contents

Chapter 1

Introduction

Contents

1.1 Context: Big Data management

To date, more and more data are captured or generated than ever before. According to the 2011 Digital Universe study of International Data Corporation (IDC), the amount of information the world produced in 2011 surpassed 1.8 Zettabytes (ZB), which marks an exponential growth by a factor of nine in just five years. Interestingly, the study also shows that the amount of data generated by individual users such as documents, photos, digital musics, blogs is far less than the amount of data being produced by applications and Internet services about their activities on their generated data (documents, photos, blogs, etc.).

The sources of this data explosion can be easily identified. Nowadays, the world has approximately 5 billion mobile phones, millions of sensors to capture almost every aspect of life. As a particular example on Facebook, over 900 million active users share about 30 billion pieces of contents per month. Within the same time interval, over 20 billion Internet searches are performed. In the world of Data-intensive High Performance Computing (HPC), Large Hadron Collider (LHC) Grid[1] produces roughly 25 PB of data at the I/O rate of 300 GB/s annually.

In this context, "Big Data" is becoming a hot term used to characterize the recent explosion of data. Everybody ranging from Information technology companies to business

firms is buzzing about "Big Data". Indeed, Big Data describes the unprecedented growth of data generated and collected from all kinds of data sources that we mentioned above. This growth can be in the volume of data or in the speed of data moving in and out data-management systems. It can also be the growth in the number of different data formats in terms of structured or unstructured data.

Addressing the need of Big Data management highly requires fundamental changes in the architecture of data-management systems. Data storage should continue innovating in order to adapt to the growth of Big Data. They need to be scalable while maintaining high performance for data accesses. Thus, this thesis focuses on building scalable data management systems for Big Data.

1.2 Contributions

The main contributions of this thesis can be summarized as follows.

Building a scalable storage system to provide efficient support for MPI-I/O atomicity

The state of the art shows that current storage systems do not support atomic, non-contiguous I/O operations in a scalable fashion. ROMIO, the MPI-I/O implementation, is forced to implement this lacking feature through locking-based mechanisms: write operations simply lock the smallest contiguous regions of the file that cover all non-contiguous regions that need to be written. Under a high degree of concurrency, such an approach is not scalable and becomes a major source of bottleneck. We address this shortcoming of existing approaches by proposing to support atomic, non-contiguous I/O operations explicitly at the level of storage back-ends. We introduce a versioning-based mechanism that offers isolation for non-contiguous I/O operations and avoids the need to perform expensive synchronization. A prototype was built along this idea and was integrated with ROMIO in order to enable applications to use our prototype transparently without any modification. We conduct a series of experiments on Grid'5000 testbed and show that our prototype can support non-contiguous I/O operations in a scalable fashion.

Pyramid: a large-scale array-oriented storage system

In the context of data-intensive High Performance Computing (HPC), a large class of applications focuses on parallel array processing: small different subdomains of huge multi-dimensional arrays are concurrently accessed by a large number of clients, both for reading and writing. Because multi-dimensional data get serialized into a flat sequence of bytes at the level of the underlying storage system, a subdomain (despite seen by the application processes as a single chunk of memory) maps to a series of complex non-contiguous regions in the file, all of which have to be read/written at once by the same process. We propose to avoid such an expensive mapping that destroys the data locality by redesigning the way data is stored in distributed storage systems, so that it closely matches the access pattern generated by applications. We design and implement Pyramid, a large-scale array-oriented storage system that leverages an array-oriented data model and a versioning-based

concurrency control to support parallel array processing efficiently. Experimental evaluation demonstrates substantial scalability improvements brought by Pyramid with respect to state-of-art approaches, both in weak and strong scaling scenarios, with gains of 100 % to 150 %.

Towards a globally distributed file systems: adapting BlobSeer to WAN scale

To build a globally scalable distributed file system that spreads over a wide area network (WAN), we propose an integrated architecture for a storage system relying on a distributed metadata-management system and BlobSeer, a large-scale data-management service. Since BlobSeer was initially designed to run on cluster environments, it is necessary to extend BlobSeer in order to take into account the latency hierarchy on multi-geographically distributed environments. We propose an asynchronous metadata replication scheme to avoid high latency in accessing metadata over WAN interconnections. We extend the original BlobSeer with an implementation of multiple version managers and leverages vector clocks for detection and resolution of collision. Our prototype, denoted BlobSeer-WAN is evaluated on the Grid'5000 testbed and shows promising results.

DStore: an in-memory document-oriented store

As a result of continuous innovation in hardware technology, computers are made more and more powerful than their prior models. Modern servers nowadays can possess large main memory capability that can size up to 1 Terabytes (TB) and more. As memory accesses are at least 100 times faster than disk, keeping data in main memory becomes an interesting design principle to increase the performance of data management systems. We design DStore, a document-oriented store residing in main memory to fully exploit high-speed memory accesses for high performance. DStore is able to scale up by increasing memory capability and the number of CPU-cores rather than scaling horizontally as in distributed data-management systems. This design decision favors DStore in supporting fast and atomic complex transactions, while maintaining high throughput for analytical processing (read-only accesses). This goal is (to our best knowledge) not easy to achieve with high performance in distributed environments. DStore is built with several design principles: single threaded execution model, parallel index generations, delta-indexing and bulk updating, versioning concurrency control and trading freshness for performance of analytical processing. This work was carried out in collaboration with Dushyanth Narayanan at Microsoft Research Cambridge, as well as Gabriel Antoniu and Luc Bougé, INRIA Rennes, France.

1.3 Publications

Journal:

- *Towards scalable array-oriented active storage: the Pyramid approach.* Tran V.-T., Nicolae B., Antoniu G. In the ACM SIGOPS Operating Systems Review 46(1):19-25. 2012. (Extended version of the LADIS paper). http://hal.inria.fr/hal-00640900

International conferences and workshops:

- *Pyramid: A large-scale array-oriented active storage system.* Tran V.-T., Nicolae B., Antoniu G., Bougé L. In The 5th Workshop on Large Scale Distributed Systems and Middleware (LADIS 2011), Seattle, September 2011. http://hal.inria.fr/inria-00627665
- *Efficient support for MPI-IO atomicity based on versioning.* Tran V.-T., Nicolae B., Antoniu G., Bougé L. In Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011), 514 - 523, Newport Beach, May 2011. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5948642
- *Towards A Grid File System Based on a Large-Scale BLOB Management Service.* Tran V.-T., Antoniu G., Nicolae B., Bougé L. In Proceedings of the CoreGRID ERCIM Working Group Workshop on Grids, P2P and Service computing, Delft, August 2009. http://hal.inria.fr/inria-00425232

Posters:

- *Towards a Storage Backend Optimized for Atomic MPI-I/O for Parallel Scientific Applications.* Tran V.-T. In The 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011): PhD Forum (2011), 2057 - 2060, Anchorage, May 2011. http://hal.inria.fr/inria-00627667

Research reports:

- *Efficient support for MPI-IO atomicity based on versioning.* Tran V.-T., Nicolae B., Antoniu G., Bougé L. INRIA Research Report No. 7787, INRIA, Rennes, France, 2010. http://hal.inria.fr/inria-00546956
- *Un support efficace pour l'atomicité MPI basé sur le versionnage des données* Tran V.-T. INRIA Research Report. INRIA, Rennes, France, 2011. http://hal.inria.fr/hal-00690562

1.4 Organization of the manuscript

The rest of this thesis is organized in five parts, briefly described in the following.

Part I: Scalability in Big Data management systems

We discuss the context of our work by presenting the related research areas. This part consists of Chapter 2, 3 and 4. Chapter 2 introduces Big Data and the state of the art of current infrastructures for Big Data management. Particularly, we first focus on distributed infrastructures that are designed to aggregate resources to scale such as clusters, grids, and clouds. Secondly, we introduce a centralized infrastructure that attracted increasingly attention in Big Data processing: a single server with very large memory and multi-core multiprocessor. Chapter 3 narrows the focus on the data-management systems, and the way they are

designed to achieve scalability. We classify those systems in the catalogs presented in Chapter 2. In Chapter 4, we study BlobSeer in deep, a large-scale data-management service. We use BlobSeer as a reference system throughout this manuscript.

Part II: Scalable distributed storage systems for data-intensive HPC

This part consists of 3 chapters. In Chapter 5, we present some common I/O practices in data-intensive high performance computing (HPC). We argue that data-intensive HPC is one type of Big Data, and we highlight some challenges of scalable storage systems in such an environment. We continue in Chapter 6 by presenting our first contribution: design and implement a scalable storage system to provide efficient support for MPI-I/O atomicity. Finally, this part ends with Chapter 7, where we introduce our second contribution in the context of data-intensive HPC. This Chapter features the design, the architecture and then the evaluations of Pyramid: a large-scale array-oriented storage system that is optimized for parallel array processing.

Part III: Scalable geographically distributed storage systems

We present our contribution on building a scalable storage system in geographically distributed environments. Chapter 8 introduces our motivation to take BlobSeer as a building block for a global distributed file system. We discuss how we re-architect BlobSeer to adapt to WAN scale and then focus on the changes in the implementation of the new BlobSeer's branch: BlobSeer-WAN. The chapter closes with a set of experiments that evaluate the scalable performance of the system in comparison with that of the original BlobSeer.

Part IV: Vertical scaling in document-oriented stores

In this part, we discuss our fourth contribution on designing a scalable data-management system in centralized environments. Concretely, we present DStore: a document-oriented store that leverages large main memory, multi-core, multiprocessor architecture to scale vertically. We focus on giving out a clear motivation for the design and a clear description of the system architecture. Evaluation of the work is then presented at the end of the chapter.

Part V: Achievements and perspectives

This part consists of Chapter 10. We summarize the contributions of this thesis, discuss the limitations and a series of perspectives for future explorations.

Part I

Context: Scalability in Big Data management systems

Chapter 2

Current infrastructures for Big Data management

Contents

In this Chapter, we aims at presenting readers a clear definition of Big Data and the evolution of current infrastructures with regard to the need of Big Data management. First, we survey distributed infrastructures such as Clusters, Grids, Clouds that aggregate the resources of multiple computers. Second, we focus on centralized infrastructure that refers to a single server with a very large main memory shared by possibly multiple cores and/or processors. This centralized infrastructure is increasingly attractive for Big Data that requires high speed of data processing/analyzing.

2.1 Definition of Big Data

According to M. Stonebreaker, Big Data can be defined as "the 3V's of volume, velocity and variety" [2]. Big Data processing refers to applications that have at least one of the following characteristics.

BIG VOLUME. Big Data analysis often needs to process Terabytes (TBs) of data, and even more. On a daily basis, Twitter users generate approximately 7 TBs of data, Facebook users share one billion pieces of content that worth 10 TBs, etc. It is obvious that the volume of data is the most immediate challenge for conventional data-management and processing frameworks. This requires a fundamental change in the architecture of scalable storage systems. In fact, many companies have archived a large amount of data in form of logs, but they are not capable to process them due to a lack of appropriate hardware and software frameworks for Big Data.

Current state of the art highlights the increasing popularity of parallel processing frameworks (e.g., Hadoop [3] with MapReduce [4] model) as efficient tools for Big Data analytics. MapReduce is inspired by the Map and Reduce functions in functional programing but it is still a novel approach for processing large amounts of information using commodity machines. Thanks to MapReduce, the companies that have big volume of data can now quickly produce meaningful insights from their data to improve their services and to direct their products better.

BIG VELOCITY. A conventional meaning of velocity is how quickly data are generated and needs to be handled. To be able to deliver new insights from input data quickly, batch processing such as in MapReduce is no longer the only preferred solution. There is an increasing need to process Big Data "on-line" where data processing speed is needed to be close to that of data flows. One intuitive example is the following: if what we had was only a 10 minutes old snapshot of traffic flows, we would not dare to cross the road because the traffic changes so quickly. This is one of many cases where MapReduce and Hadoop can not fit to the "on-line" speed requirements. We simply cannot wait for a batch job on Hadoop to complete because the input data would change before we get any result from the processing process.

BIG VARIETY. Nowadays, data sources are diverse. With the explosion of Internet-capable gadgets, data such as texts, images are collected from anywhere and by nearly any device (e.g., sensors, mobile phones, etc.) in terms of raw data, structured data and semi-structured data. Data has become so complex that specialized storage systems are needed to deal with multiple data formats efficiently.

Recent storage trends have shown the raise of NoSQL approaches and particularly the emergence of graph databases for storing data of social networks. Although current relational database management systems (DBMS) can be used as "one-size-fit-all" for just every type of data formats, doing so results in poor performance as proved by a recent study [5]. By implementing the application-needed data models (e.g. Key/Value, Document-oriented, Graph-oriented), NoSQL storage devices are able to scale horizontally to adapt to the increasing workloads.

According to the above characteristics of Big Data, our contributions in this thesis can be classified as in the Table 2.1. We will present our arguments to support this classification

Contribution	Big Volume	Big Velocity	Big Variety
Building a scalable storage system to provide efficient support for MPI-I/O atomicity	√	—	—
Pyramid: a large-scale array-oriented storage system	√	—	√
Towards a globally distributed file systems: adapting BlobSeer to WAN scale	√	—	—
DStore: an in-memory document-oriented store	—	√	√

Table 2.1: Our contributions with regard to Big Data characteristics
(√ = Addressed, — = not addressed).

further in the next chapters.

2.2 Clusters

Cluster computing is considered as the first effort to build distributed infrastructures by interconnecting individual computers through fast local area networks. The goal is to gain performance and availability compared to the case when a high-end computer with comparable performance or availability is less cost-effective or unfeasible. The simplicity of installation and administration has made clusters become popular and important infrastructures.

There is no strict rules to build a cluster. An inexpensive cluster can be built from commodity computers, called nodes, connected through Ethernet networks, while a high-end cluster built on expensive high-end computers with high speed interconnections. The most basic cluster configuration is *Beowulf* [6, 7], originally referred to a specific cluster built by NASA in 1994. *Beowulf* cluster consists of normally identical cluster nodes in terms of both hardware and software. Usually, they are equipped with a standardized software stack: Unix-like operating system, Message Passing Interface (MPI [8]), or Parallel Virtual Machine (PVM) [9]. Nowadays, cluster size ranges from a couple of nodes up to tens of thousands.

In the context of Big Data, clusters can be considered as the most popular infrastructure for Big Data management. By federating resources, a cluster can potentially provide a decent amount of storage space for hosting Big Volume of data. In high-end clusters, data-management systems obviously can take advantages of its powerful processing capability of its cluster nodes as well as the high-speed interconnection among them to cope with the Big Velocity characteristic.

2.2.1 Commodity clusters

The main goal of commodity cluster computing (or commodity computing) is to deliver high computing power at low cost by federating already available computer equipments. The idea consists in using more low-performance, low-cost hardware that work in parallel rather than high-performance, high-cost hardware in smaller numbers. Therefore, commodity hardware components are mostly manufactured by multiple vendors and are incorporated based on open standards. Since the standardization process promotes lower costs and

identical products among vendors, building commodity cluster avoids the expenses related to licenses and proprietary technologies.

Low-cost hardware usually comes along with low reliability. In a system made out of a large number of poorly reliable components, failure is not an exception but the norm. For this reason, middleware running on commodity cluster needs to deal with fault-tolerance by design. A successful example of middleware for commodity cluster is MapReduce framework, introduced by Google in [4, 10].

To date, the largest cluster systems are built by industry giants such as Google, Microsoft and Yahoo! in order to cope with their massive data collections. Those clusters are made out of commodity machines, running a MapReduce or MapReduce-like framework. Google has not revealed the size of its infrastructures, but it is widely believed [11] that each cluster can have tens of thousands nodes interconnected with just standard Ethernet links.

2.2.2 High performance computing (HPC) clusters (a.k.a. Supercomputers

In contrast to commodity computing, HPC clusters are made by high-cost hardware (e.g., IBM Power7 RISC) that are tightly-coupled by high-speed interconnection networks (e.g., InfiniBand). Currently, the second fastest HPC cluster in the world is Japan's *K computer* [12]. *K computer*, produced by Fujitsu, consists of over 80,000 high-performance nodes (2.0 GHz, 8-core SPARC64 VIIIfx processors, 16 GB of memory), interconnected by a proprietary six-dimensional torus network.

HPC clusters are actively used in different research domains such as quantum physics, weather forecasting, climate research, molecular modeling and physical simulations. Those problems are highly compute-intensive tasks which need to be solved in a bound completion time. HPC cluster can offer an excellent infrastructure to dispatch a job over a huge number of processes and guarantee efficient interconnection between them for each computation step.

2.3 Grids

By definition in [13], the term "the Grid" refers to

> *a system that coordinates distributed resources using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service.*

In [14], Ian Foster differentiated "Grids" from other type of distributed systems by a three-point requirements checklist.

To coordinate distributed resources. A Grid must integrate and coordinate not only resources but also Grid users within different administrative domains. This challenge brings new issues of security, policy and membership administration which did not exist in locally managed system.

To use standard, open, general-purpose protocols and interfaces. A Grid should be built from standard, open protocols and interfaces to facilitate the integration of multiple organizations and to define a generic well-built framework for different kind of Grid

applications. A Grid covers a wide range of fundamental issues such as authentication, authorization, resource discovery, and resource access.

To deliver nontrivial quality of service. Grid resources should be used in a coordinated fashion to deliver various qualities of services such as response time, throughput, availability, and security. Grid allows co-allocation of various resource types to satisfy complex user demands, so that the utility of combined system is significantly better than just a sum of its components.

The term "Grid" originates from an analogy between this type of distributed infrastructure and the electrical power Grid: any user of the Grid can access the computational power at any moment in a standard fashion, as simple as plugging an electrical equipment into an outlet. In other words, a Grid user can submit a job without having to worry about its execution or even knowing the usage of resources in the computation. This definition has been used in many contexts where it is hard to understand what a Grid really is.

One of the main powerful features of Grid is the introduction of "virtual organization" (VO), which refers to a dynamic set of disparate groups of organizations and/or individuals agreed on sharing resources in a controlled fashion. VO clearly states under which conditions resources including data, software, and hardware can be shared to the participants. The reason behind is that the sharing resources will be seen transparently by applications to perform tasks despite their geographically disparate providers.

In the context of Big Data, a Grid infrastructure has the potential to address (but is not limited to) a higher scale of Big Volume than a cluster does. As a Grid can be built from many clusters of different administration domains, it can expose an aggregate storage space for storing a huge amount of data that cannot fit to any participant clusters in the Grid.

2.3.1 Grid architecture

The goal of Grid is to be an extensible, open architectural structure that can adapt to the dynamic, cross-organizational VO managements. Ian Foster argued on a generic design that identifies fundamental system components, the requirements for these components, and how these components interact with each other [13]. Architecturally, a Grid organize components into layers, as on Figure 2.1.

Grid Fabric provides the lowest access level to raw resources including both physical entities (clusters, individual computers, sensors, etc.) and logical entities (distributed file systems, computational powers, software libraries). It implements the local, resource-specific drivers and provides a unified interface for resources sharing operations at higher layers.

Connectivity guarantees easy and secure communication between fabric layer resources. The connectivity layer defines core communication protocols including transport, routing, and naming. Authentication protocols enforce security of communication to provide single sign-on, user-based trust relationships, delegation and integration with local security solutions. Those protocols must be based on existing standards whenever possible.

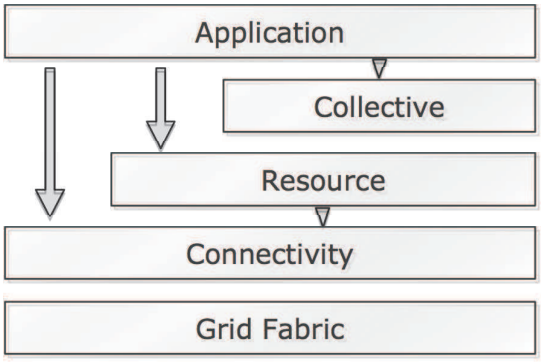


Figure 2.1: Generic Grid architecture.

Resource is built on top of connectivity layer using communication and authentication protocols provided to define protocols that expose individual resources to the Grid participants. The resource layer distinguishes two primary protocol classes: information protocols and management protocols. Information protocols are used to query the state of a resource such as its configuration, current load, and usage policy. Management protocols apply to negotiate access to a shared resource, and to specify resource requirements.

Collective does not associate with any single specific resource but instead coordinate individual resources in collections. It is responsible to handle resource discovery, scheduling, co-allocation, etc.

Application consists of user applications that make use of all other layers to provide functionalities within a VO environment.

2.3.2 Grid middleware

Grid middleware are general-purpose Grid software libraries that provide core components and interfaces for facilitating the creation and management of any specific Grid infrastructure. A lot of effort from industrial corporations, research groups, university consortiums has been dedicated to developing several Grid middleware.

Globus. The Globus Toolkit [15] has been developed by the *Globus Alliance* as a de-facto standard Grid middleware for the construction of any Grid infrastructure. Following a modular-oriented design, the Globus Toolkit consists of several core Grid components made up a common framework to address security, resource management, data movement, and resource discovery. The Globus Toolkit enables customization by allowing users to expose only desired functionalities when needed. For this reason, the Globus Toolkit has been widely adopted both in academia and industry such as IBM, Oracle, HP, etc.

UNICORE. *UNiform Interface to COmputing REsources (UNICORE)* [16] is a Grid middleware currently used in many European research projects, such as EUROGRID, GRIP,

VIOLA, Open MolGRID, etc. UNICORE has been developed with the support of the German Ministry for Education and Research (BMBF) to offer a ready-to-run Grid system including both client and server software. Architecturally, UNICORE consists of three tiers: user, UNICORE server and target system. The user tier provides a Graphical User Interface (GUI) that enables intuitive, seamless and secure access to manage jobs running on the UNICORE servers. The UNICORE servers implement services based on the concept of Abstract Job Object (AJO). Such an object contains platform and site-independent description of computational and data-related tasks, resource information and workflow specifications. The target tier provides resource-dependent components that expose interface with the underlying local resource management system.

gLite. The *Lightweight* middleware [17] was developed as part of the flagship European Grid infrastructure project (EGEE) [18], under the collaborative efforts of scientists in 12 different academic and industrial research centers. The gLite middleware has been used to build many large-scale scientific Grids, among them is the Worldwide LHC Computing Grid deployed at CERN. gLite was initially based on the Globus toolkit, but it eventually evolved into a completely different middleware specialized to improve usability in production environments. To this end, users are supplied with a rich interface to access to a variety of management tasks: submitting/canceling jobs, retrieving logs about job executions, uploading/deleting files on the Grid, etc.

2.4 Clouds

Cloud computing is an emerging computing paradigm that has attracted increasing attention in recent years, especially in both Information technology and Economy community. The media as well as computer scientists have a very positive attitude towards the opportunities that Cloud computing is offering. According to [19], Cloud computing is considered "no less influential than e-business", and it would be the fundamental approach towards Green IT (environmentally sustainable computing).

Several authors tried to find a clear definition of what Cloud computing is and how it is positioned with respect to Grid computing [20, 19]. Although there are many different definitions, they share common characteristics. Cloud computing refers to the capability to deliver both software and hardware resources as services in a scalable way. Architecturally, the core component of Cloud is the data center that contains computing and storage resources in term of raw hardware, together with software for lease in a pay-as-you-go fashion. In [21], Berkeley RAD lab defined Cloud computing as follows:

> *Cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). The datacenter hardware and software is what we will call a Cloud. When a Cloud is made available in a pay-as-you-go manner to the general public, we call it a Public Cloud; the service being sold is Utility computing. We use the term Private Cloud to refer to internal datacenters of a business or*

> *other organization, not made available to the general public. Thus, Cloud computing is the sum of SaaS and Utility computing, but does not include Private Clouds. People can be users or providers of SaaS, or users or providers of Utility computing [21].*

From a hardware point of view, Cloud computing distinguishes itself from other computing paradigms in three aspects.

- There is no need for Cloud computing users to plan the future development of IT infrastructure far ahead. Cloud offers the illusion of unlimited computing resources to users.
- Cloud users can start renting a small amount of Cloud resources and increase the volume only when their needs increase.
- Cloud providers offer the ability to pay-as-you-go on a short-term basis. For instance, processors and storage resources can be released if no longer useful in order to reduce costs.

Regarding to Grid computing, Cloud computing is different in many aspects as security, programming models, data models and applications, as argued by Foster et al. [22]. Cloud computing leverages virtualization to separate the logical layer from the physical one, that consequently maximizes resource utilization. Whereas Grid achieves high utilization through fair sharing of resources among organizations, Cloud potentially maximizes resource usages by allowing concurrent isolated tasks running on one server, thank to virtualization. Cloud gives users the impression that they are allocated dedicated resources scaling on demand, even though in a shared environment.

In the context of Big Data, Cloud computing is a perfect match for Big Data since it virtually provides unlimited resources on demand. Cloud computing opens the door to Big Data processing for any user that may not have the possibility to build it-own infrastructures such as Clusters, Grids, etc. By renting resources, Cloud users can potentially perform their Big Data processing in an economic way.

2.4.1 Cloud Architecture

Architecturally, Cloud computing may consist of three layers, as illustrated on Figure 2.2.

Infrastructure as a Service (IaaS). IaaS offers on-demand raw hardware resources such as computing power, network and storage in the form of virtualized resources. Users of IaaS Clouds typically rent customized virtual machines in which they have the possibility to select a desired Operating System (OS), and to deploy arbitrary software with specific purposes. Fees are charged with a pay-as-you-go model that reflects the actual amount of raw resources consumption: storage space per volume, CPU cycles per hour, etc. Examples of IaaS Cloud platforms include: Nimbus [23], Eucalyptus [24], OpenNebula [25], and Amazon Elastic Compute Cloud [26].

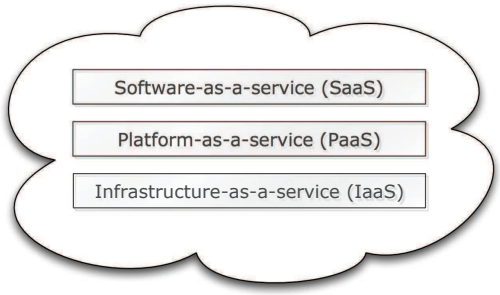


Figure 2.2: Cloud services and technologies.

Platform as a Service (PaaS). PaaS sits on top of IaaS to provide a high-level computing platform typically including Operating System, database, programming and execution environment. PaaS targets software developers. Using PaaS, they can design their applications using specific frameworks provided by the platform without the need of controlling the underlying hardware infrastructure (IaaS). Examples of PaaS are Google App Engine [27], Microsoft Azure [28], and Salesforce [29].

Software as a Service (SaaS). At the highest level, SaaS is the most visible layer for end-users. It delivers software as a service by allowing users to directly use applications deployed on Cloud infrastructure. Cloud users do not need to worry about installing software on their own computer and managing updates and patches. Typically, any Web browser can act as frontend of Cloud applications. Examples of SaaS Clouds are Google Docs [30] and Microsoft Office Live [31], etc.

2.4.2 Cloud middleware

As the most attractive computing technology in recent years, Cloud middleware have been under heavy development in academia as well as in industry. Several Clouds offered by industry giants such as Amazon, Google, IBM, Microsoft already matured as commercial services under massive utilization. We survey some examples of Infrastructure as a Service (IaaS) Cloud.

Amazon EC2. EC2 [26] is an Infrastructure as a Service (IaaS) Cloud that has become the most widely-used commercial Cloud. It offers rich functionalities. Amazon EC2 provides resizable virtual computing environments by allowing user to rent a certain amount of compute and storage resources, and dynamically resize them on demand. Though a Web service interface, Amazon users can launch virtual machines with a range of selected operating systems (OS). Then, users are capable to load customized applications and run their own computations.

Amazon also delivers S3 [32], a Cloud storage service that offers a simple access interface for data transfers in and out of the Cloud. S3 not only stores users' data but also acts as a large pool of predefined *Amazon Machine Images (AMIs)*. Upon requested, an AMI will be load to run on allocated virtual machines, or be customized to form a new AMI.

Nimbus. Nimbus [23] is an open-source EC2/S3-compatible IaaS implementation that specifically targets scientific community, with the purpose of being an experimental testbed tailored for scientific users. Nimbus shares similar features with Amazon EC2 by exposing an EC2-like interface to enable the ease-of-use. The data-storage support of Nimbus, Cumulus [33] is also compatible with the Amazon S3. Cumulus can access various storage management systems and exposes S3-interface for storing users' data and customized images of virtual machines.

Eucalyptus. Eucalyptus [24] stands for *Elastic Utility Computing Architecture Linking Your Programs To Useful Systems*. Eucalyptus enables the creation of private IaaS Clouds with minimal effort without the need for introducing any specialized hardware or retooling existing IT infrastructure. By maintaining the de-factor Amazon Cloud API standards, Eucalyptus supports the construction of hybrid IaaS Clouds to bridge private to public Cloud infrastructures.

Additionally, Eucalyptus is one among the few Cloud middleware that feature *a virtual network overlay* implementation that provides network isolation. This design brings two major benefits. First, network traffic of different users is isolated to mitigate interference. Second, resources on different clusters are unified to give users the impression that they belong to the same Local Area Network (LAN).

2.5 Big memory, multi-core servers

As hardware technology is subject to continuous innovation, computer hardware are made more and more powerful than their prior models in the past. One obvious demonstration is the application of Moore's law. Gordon E. Moore, Intel co-founder, claimed in his 1965 paper [34] that the number of components in integrated circuits had doubled every two years and predicted the trend would continue in the near future.

The relation of price versus capacity has decreased exponentially over time in storage industry, not only in hard-drives but also in main-memory development. For example, 1 MB of main memory dropped at US $0.01 in 2010, which is an impressive decrease in comparison with the price at about US $100 in 1990. A similar observation can be also found in hard-drive industry. In 2012, a typical high-end server could be equipped with 100 GB of main memory and hundreds of TBs of storage space. The near future will see the existence of 1 TB main memory and more in a single server.

In the processor industry of the past, the speed of Central Processing Units (CPUs) had been doubled every 20 months on average. This brilliant achievement had been made possible thanks to two major factors. First, the creation of faster transistors results in increased clock speed. Second, the increased number of transistors per CPU not only made processor production more efficient but also decreased material consumption. More specifically, the number of transistors on a processor increased from 2300 transistors in 1971 to about 1.7 billion today at approximately the same price.

However, since 2002, the growth in clock speed stopped as it had done for almost 30 years. Moore's law on CPU speed has reached its limit due to power consumption, heat distribution and speed of light [35]. As a result, latest trends on CPU design highlighted the emergence of multi-core/many-core and multiprocessor architectures. The clock speed of

CPU is no longer on Moore's law, but the number of cores/processors on a single computer is.

2.6 Summary

This chapter presented the explosion of Big Data and a survey of current infrastructures in the context of Big Data management. First, we presented a clear definition of Big Data by "the 3V characteristics of Volume, Velocity and Variety". Second, we introduced some of current infrastructures that can be leveraged for Big Data management, including clusters, Grids, Clouds, and multi-core, multiprocessor servers with large memory.

However, to be able to deliver high scalability for Big Data management, we need to have a good understanding of how current storage systems have been designed and what are the main approaches for scalability. These aspects are addressed in the next chapter.

Chapter 3

Designing data-management systems: Dealing with scalability

Contents

SCALABILITY is defined as *the ability of a system, network, or process, to handle growing amount of work in a capable manner, or its ability to be enlarged to accommodate that growth* [36].

This definition of scalability can be applied to asset whether that a system is scalable or not on a specific system property. For example, scalable data throughput refers to the capability of a system to increase total throughput when resources are added in order to handle an increased workload.

A system that has poor scalability can result in poor performance. In many cases, adding more resources to an unscalable system is an inefficient investment that cannot lead to substantial improvements.

Obviously, scalability by design is needed, especially for Big Data management systems. There are two ways to scale.

Scale horizontally. This approach is usually referred to as "scale-out", meaning to add more nodes to a system, for instance new computing nodes to a cluster. As computer prices drop, a powerful computing cluster can be built by aggregating low-cost "commodity" computers connected via a local network. By following the "divide-to-conquer" model where each node is assigned only a subset of the global problem, the cluster can be easily scaled to a certain number of worker nodes, to adapt to each particular problem size.

Scale vertically. In this approach, resources are added to some nodes in a system, typically meaning to add more CPU and more memory to each node. This is usually referred to as "scale-up", which enables running services, both user and kernel levels, to have more resources to consume. For example, adding more CPUs allows more threads to be run simultaneously. Adding more memory can enlarge the cache pool to reduce accesses to secondary storage.

In this chapter, we present several existing approaches to scalable design of data-management systems, whose goal is to store and retrieve data in an efficient way. Thus, we will not go into details about other aspects such as fault-tolerant design of those systems. We classify the mechanisms in three levels: cluster (horizontal scaling), geographically distributed sites (horizontal scaling) and single server (vertical scaling).

3.1 Scaling in cluster environments

3.1.1 Centralized file servers

In cluster environments, the most basic form of shared data storage among cluster nodes is *Network-Attached Storage (NAS)*. NAS refers to a single dedicated machine that directly connects to block-based storage devices. It is responsible for all data accesses issued by other machines in the cluster. In this setting, this dedicated machine communicates to block-based storage devices through I/O bus protocols such as IDE, SATA, SCSI and Fiber channel, while exposing a file system interface to file system clients. In order to do so, it has to maintain a mapping between its file system's data structures (files and directories) and the corresponding blocks on the storage devices. This extra data for this mapping is commonly named metadata and the dedicated machine is usually called a *network file server*.

In order to access a *network file server* through the network, the *network file server* and file system clients agree on standardized network protocols, such as the *Network File System protocol (NFS)* for communication. By using such a protocol, the clients can access a remote file systems in the same way as accessing its local file systems.

The first type of block-based storage devices that can be attached to network file servers is *Direct-attached storage (DAS)* [37]. To be able to scale the system, either for storage capability

or/and for performance, DAS can be employed within a RAID [38] (Redundant Array of Independent Disk) setting.

RAID is capable to combine multiple block-based storage units into a single logical unit. RAID can organize a distribution scheme in one of several ways called "RAID levels", depending on each user's particular requirements in terms of capacity, redundancy and performance. For example, RAID 0 refers to a simple block-level striping over a number of storage devices without any fault-tolerance mechanism such as parity or mirroring. Consequently, RAID 0 offers the best performance but no fault-tolerance. Upon writing, data is fragmented into same-size blocks that are simultaneously written to their respective drives. Upon reading, data blocks are fed in parallel to achieve increasing data throughput.

Another technology to scale the storage capability in NAS configurations is to employ a centralized server on top of a storage-area network (SAN). A SAN features a high performance switched fabric in order to provide a fast, as well as scalable interconnect for a large number of storage devices. One observation is that in both cases where NAS is implemented on top of a SAN or DAS, the performance of the entire system is limited by the performance of the single file server.

SAN file systems [39], have been introduced to address the aforementioned issues. In a SAN file system setting, the clients are also connected to a SAN to directly access data through block-based protocols (e.g., iSCSI). Hence, the file server is only responsible for metadata management, reducing I/O overhead and increasing the overall performance. IBM SAN file system [40], EMC High-Road [41] are good examples of existing SAN file systems. Although the approach of SAN file systems seems to be scalable, it is practically hard and expensive to build a SAN at a large-scale, compare to other cross-platform approaches such as parallel file systems. In 2007, IBM discontinued selling the SAN file systems and replaced it by IBM General Parallel file system (GPFS) [42].

3.1.2 Parallel file systems

Parallel file systems are the most appropriate solution for data sharing in a high-performance computing (HPC) cluster, as they offer a higher level of scalability than centralized storage solutions. Parallel file systems also have the advantage of transparency, which allows any clients to access data using standardized network protocols. The clients do not have to possess a dedicated access to the underlying storage resources (e.g., Fiber channel).

To scale out, a parallel file system typically federates multiple nodes, each of which contributes its individual storage resources. Those nodes are called I/O nodes serving data to client/compute nodes on the cluster. Parallel file systems implement a well-known mechanism, called *data stripping*, to distribute large files across multiple I/O nodes, which greatly increases scalability in terms of both capacity and performance. First, it allows to store very large files far than the capacity of any individual I/O node in the cluster. Second, reading and writing can be served in parallel by multiple servers, which reduces the actual workload on each particular server.

Another scalable design aspect in parallel file systems refers to breaking I/O in two phases: metadata I/O and data I/O, so as to offload data I/O totally to storage nodes. Metadata refer to the information about file attributes and file content locations on the I/O nodes. Since this information is comparatively smaller than file data itself, metadata I/O workload

can be handled in a centralized way in one special node, named Metadata server (MDS).

Recently, it is claimed that a centralized MDS can be a bottleneck at large-scale [43, 44]. Many novel parallel file systems moved forward to a distributed design MDS [45, 44]. However, scalable MDS approaches face bigger challenges and are still under active research. A typical trade-off when designing a distributed MDS scheme is the need to choose between POSIX [46] file access interface and scalable performance. The highly standardized interface enables a high level of transparency, allowing clients to have the same I/O semantics as in local file systems. On the other hand, its strict semantics is hard to guarantee in a distributed environment, and limits the system scalability.

3.1.2.1 PVFS

Parallel virtual file system (PVFS) was first introduced in 2000 [45] as a parallel file system for Linux clusters. It is intended to provide high-performance I/O to file data, especially for parallel applications. Like other parallel file systems, PVFS employs multiple user-level I/O daemons, typically executed on separate cluster nodes which have local disks attached, called I/O nodes. Each PVFS file is fragmented into chunks and distributed across I/O nodes to provide scalable file access under concurrency. PVFS deploys one single manager daemon to handle metadata operations, such as directory operations, distribution of file data, file creation, open, and close. In this case, clients can perform file I/O without the intervention of the metadata manager.

Although metadata I/O seems to be less heavy than file I/O, one manager daemon still suffers from low performance under certain workloads, especially when dealing with massive metadata I/O on a huge number of small files. Therefore, a new version of PVFS was released in 2003, featuring distributed metadata management and object servers. To scale metadata on multiple servers, PVFS v2 does not implement a strict POSIX interface, it instead implements a simple hash function to map file paths to server IDs. This may lead to inconsistent states when concurrent modifications occur on the directory hierarchy.

3.1.2.2 Lustre

Lustre [47] was started in 1999 with the primary goal of addressing the bottlenecks traditionally found in NAS architectures, and of providing scalability, high performance, and fault tolerance in cluster environments. Lustre is one of the first file systems based on the object storage device (OSD) approach. File data is striped across multiple *object storage servers (OSSes)*, which have direct access to one or more storage devices (disks, RAID, etc.) called *object storage targets (OSTs)* that manage file data.

Lustre exposes a standard POSIX access interface and supports concurrent read and write operations to the same file using locking-based approach. Lustre is typically deployed with two metadata servers (one active, one standby), sharing the same metadata target (MDT) that hosts the entire metadata of the file system. In this setting, Lustre still manages metadata in a centralized fashion, and uses the standby server just in case of failures.

3.1.2.3 Ceph

Ceph [48], recently developed at University of California, Santa Cruz, is a distributed file system that leverages the "intelligent" and "self-managed" properties of OSDs to achieve scalability. Basically, Ceph delegates the responsibility for data migration, replication, failure detection and recovery to OSDs. File data is fragmented into objects of separate *placement groups (PGs)*, which are self-managed.

One novel approach in Ceph is that metadata management is decentralized by an approach based on dynamic subtree partitioning [49]. This distributed management in a metadata cluster potentially favors workload balancing, and eliminates the single point of failure. In contrast to existing object-based file systems [47, 50], Ceph clients do not need to access metadata servers for object placement since this information can be derived by using a configured flexible distribution function. This design eliminates the need to maintain object placement on metadata servers, and thus reduces metadata workload.

3.1.3 NoSQL data stores

Relational database management systems (RDBMS) have been considered a "one size fits all" model for storing and retrieving structured data along the last decades. RDBMS offer a powerful relational data model which can precisely define relationships between datasets. Under ACID (atomicity, consistency, isolation, durability) semantics, RDBMS guarantee database transactions are reliably processed. They release client applications from the complexity of consistency guarantees: all read operations will always be able to have data from the latest completed write operation.

There are several technologies to scale a DBMS across multiple machines. One of the most well-known mechanisms is "database sharding", which breaks a database into multiple "shared-nothing" "shards" and spread those smaller databases across a number of distributed servers. However, "database sharding" cannot provide high scalability at large scale due to the inherent complexity of the interface and ACID guarantees mechanisms.

In 2005 [51], scientists claimed the "one size fits all" model of DBMS had ended and raised the call for new design of alternative highly scalable data-management systems. Many NoSQL data stores have been introduced in recent years such as Amazon Dynamo [52], Cassandra [53], CouchDB [54], etc. They are found by industry to be good fits for Internet workloads.

To be able to scale horizontally, NoSQL architectures differ from RDBMS in many key design aspects as briefly presented below.

Simplified data model. NoSQL data stores typically do not organize data in a relational format. There are three main types of data access models: key/value, document-oriented, and column-based. Each the data model is appropriate to a particular workload so that the data model should be carefully selected by the system administrators.

Key/value stores implement the most simplified data model, which resembles the interface of a hash table. Given a key, key-value stores can provide fast access to its associate value through three main API methods: GET, PUT, and DELETE. Examples of key-value stores include: Amazon Dynamo [52] and Riak [55].

Document-oriented stores are designed for managing semistructured data organized as a collection of documents. As in key-value stores, each document is identified by its unique ID, giving access to its content. One of the main defining characteristics that differentiates document-oriented stores from key-value stores is that the interface is enriched to allow the retrieval of documents based on their data fields. Examples are CouchDB [54] and MongoDB [56].

In the column family approach, the data structure is described as "a sparse, distributed, persistent multidimensional sorted map" as in Google's Bigtable. Data is organized in rows as in RDBMS, but the rows do not need to have the same set of columns. Thus, the data table represents a sparse table with gaps of NULL values. Examples include: Google Bigtable [57] and Cassandra [53].

Complex queries such as JOIN are typically not supported in any of the three approaches.

Reducing unneeded complexity. Many Internet applications do not need rich interfaces and the ACID semantics provided by relational databases. These features, which aim at "one size fits all", are expensive to scale. RDBMS have to either scale vertically (e.g., buy more powerful hardware), or suffer from distributed locking mechanisms and high network latency while scaling horizontally (e.g., add more nodes to the clusters). NoSQL approaches, on the other hand, typically sacrifice ACID properties and complex query support for high scalability. NoSQL usually supports only the eventual consistency model, where read operations are allowed to return stale results but under the guarantee that: all the readers will eventually get the fresh, last written data.

Horizontal scaling on commodity hardware. NoSQL data stores are designed to scale horizontally on commodity hardware. They do not rely on highly reliable hardware. Storage nodes can join and leave the storage clusters without causing the entire system to stop functioning. In many NoSQL data stores, the key service to enable scalability is a distributed hash table (DHT).

3.2 Scaling in geographically distributed environments

Scalable data-management systems in geographically distributed environments are needed for many reasons. First, large datasets from various scientific disciplines have been growing exponentially and cannot fit in a centralized location. Second, the geographically distributed users may want to share their own datasets without storing in a central repository. In this section, we study the design of geographically distributed data-management systems, focusing on the design for scalability with regard to several key characteristics of the environments: interconnection latency, low bandwidth, heterogeneous resources, etc.

3.2.1 Data Grids

Data Grids were first introduced in [58] in a collaborative effort to design and implement an integrated architecture to manage large data that are distributed over geographically distant locations. Typically, this effort directly addressed the challenges of managing scientific

data in many disciplines, such as high-energy physics, computational genomics and climate research.

The architecture of a Data Grid consists of four layers. Each layer builds its functionality based on the services provided by lower layers and by the components of the same level. Those layers can be described in the following order from the lowest to the highest.

Data Fabric consists of the distributed storage resources that are owned by the grid participants, but are aggregated to form a global storage space. Those participant resources can be both software and hardware: file servers, storage area networks, distributed file systems, relational database management systems, etc.

Communication provides a number of protocols used to transfer data among resources of the fabric layer. These protocols are built on top of common communication protocols such as TCP/IP and use authentication mechanisms such as the public key infrastructure (PKI). Further, SSL (Secure socket layer) can be used to encrypt the communication to ensure security.

Data Grid Services consists of services enabling applications to discover, manage and transfer data within the Data Grid. More precisely, end users are equipped with replication services, data discovery services, and job submission services, which cover all aspects of efficient resource management while hiding the complexity of the Data Grid infrastructure.

Applications consists of domain-specific services that facilitate and boost up Data Grid experience. Those services are highly customized and standardized tools for Grid participants.

3.2.2 Scalability concerns and trade-offs

Scalability design in Data Grids faces many challenges due to high latency and low bandwidth of interconnection between geographically distributed participants. We briefly survey several design decisions, which address the final goal of building scalable systems.

Replication is a well-known mechanism to increase performance by reducing latency, especially in wide-area networks (WAN), and to serve as a fault-tolerance technique by creating multiple copies of the data. In Data Grids, replication is usually performed in a simple way and on a per-request basis. This is done with the purpose of reducing the expensive cost of synchronization via low-bandwidth, high-latency networks in geographically-distributed environments. To maintain the convergence of replicas, Data Grids select a single data source to act as a primary copy of each particular dataset. Upon request of grid participants, a copy of the dataset can be transferred to their own sites. Any update is done on the primary copy and then is propagated to Grid participants that subscribe to the updates on the primary copy.

Consistency defines the "freshness" of data seen by applications. Obviously, Data Grids do not provide "strong consistency" guarantees because of their high cost. Strong consistency explicitly requires locking on huge datasets and maintaining synchronous replication, which are not scalable in practice as discussed above.

Transaction support refers to the property that all of the processing operations of a transaction are either all succeed together or all failed together. This necessity requires some supports for checkpointing and rollback mechanisms similar to the ones in database management systems (DBMS). However, those mechanisms are not efficient at large scale. Therefore, Data Grids do not support transactions usually.

Moving computation close to data. When dealing with huge datasets, Data Grid scientists proposed to move computation close to the data rather than moving the data itself. One example of this interesting approach is the design and implementation of the Gfarm file system [59].

3.2.3 Examples

Grid Data Farm. The Grid Datafarm (Gfarm) [59] is a distributed file system designed for high-performance data access and reliable file sharing in large scale environments, including grids of clusters. To facilitate file sharing, Gfarm manages a global namespace which allows the applications to access files using the same path regardless of file location. It federates available storage space of Grid nodes to provide a single file system image. To enable high performance file I/O, files in Gfarm are fragmented into chunks that are distributed over storage nodes in the grid. Applications can configure the replication factor for each file individually to improve access locality, thus avoiding bottlenecks to popular files on remote sites. Furthermore, Gfarm enables scheduling computations close to data, as it explicitly exposes the location of chunks for each file through a special API at application level.

XtreemFS. XtreemFS [60] is an open-source object-based, distributed file system for wide-area deployments that enables file accesses over Internet. To mitigate security threats in public insecure networking infrastructures, XtreemFS transparently relies on SSL and X.509 to build secure communications channels between clients and XtreemFS servers. As it is designed for WAN environment, XtreemFS implements file replication to provide fault-tolerance and to reduce data movement across data centers. Additionally, it implements a metadata caching mechanism in order to improve performance over high-latency networks.

3.3 Scaling in big-memory, multi-core servers

3.3.1 Current trends in scalable architectures

Currently, there are two major trends in designing scalable data management systems running on high-end servers.

In-memory storage systems. Modern servers nowadays are often equipped with large main memory capability that can have a size up to 1 TB. Given this huge memory capability, one question arises: do we need to store data on secondary storage, say hard drives? According to a recent study [5], many data management systems are now able to fit entirely in main memory, so that disk-oriented storage becomes unnecessary (or just useful for backup purposes). The trend of employing an in-memory design emerged

as a result of the decreasing cost/size ratio of memory and the performance provided when compared to the disk, as memory accesses are at least 100 times faster.

As in-memory design is becoming an attractive key principle to vertically scale data-management systems, several commercial in-memory databases have recently developed. Systems such as Timesten [61], VoltDB (the commercial version of H-Store [62]), SAP HANA [63], are able to provide faster, higher-throughput online analytical processing (OLAP) and/or faster transaction processing (OLTP) than disk-based data-management systems. The reason behind is that the performance of disk-based systems is actually limited because of the I/O bottlenecks on disk accesses. Of course, disk-based data-management systems can leverage main memory as a big cache to improve I/O performance. However, their architectures optimized to use only disks have to implement complex mechanisms to keep data on cache and on disks consistent. This limits the system scalability by design.

Single-threaded execution model. Apart from the in-memory approach, the search for efficient utilization of compute resources in multi-core machines triggered a new architecture for multi-threading applications. Previously, application design was relying on multiple threads to perform tasks in parallel, in order to fully utilize CPU resources. In reality, most of the tasks are not independent of each other, as they either access the same part of data, or they need part of the results generated by the other tasks. This well-known problem (concurrency control) made it nearly impossible to have parallelization fully in a multi-threading environment.

Recent data-management systems are based on a single-threaded execution model [64] where there is no need to worry about concurrency control. As long as one single thread is performing data I/O, thread-safe data structures are not necessary. In other words, no locking mechanism is needed, which results in less execution overhead.

In the context of multi-core machines, the single-threaded execution model called for a shared-nothing architecture. CPU cores should be used in a way that pure parallelization is guaranteed. The cores should work on unshared data. Examples are H-Store [62], and HyPer [65], etc.

3.3.2 Examples

H-Store. H-Store [62] is an experimental row-based relational database management system (DBMS) born from a collaboration between MIT, Brown University, Yale University, and HP Labs. H-Store aims at being optimized for online transaction processing (OLTP) applications by leveraging main memory as the persistent storage for fast data accesses. To avoid the overhead of using multi-threaded data structures, H-Store follows the single threaded-execution model where each data structure belongs to one and only one thread.

To scale the system in a multi-core server, H-Store relies on a shared-nothing architecture. Each CPU core is delegated one *site* (a partition of the database) that is the basic atomic entity in the system; each *site* runs a single-threaded daemon performing transactions independently on its own unshared part of the database stored in the main memory. Additionally, H-Store introduces other optimizations, such as replica-

tion over *sites* and "pre-defined stored procedures", etc. Pre-defined stored procedures potentially allow H-Store reducing the cost of SQL query analyzing.

HyPer. HyPer [65] is a main-memory database management system built to handle both OLTP and OLAP simultaneously on a single multi-core server. HyPer follows the same single-threading approach first recommended in [64], where all OLTP transactions are sequentially handled by one single-threaded daemon. This architecture mitigates the need for concurrent data structures and expensive locking because only one thread owns the entire database.

To support OLAP and OLTP simultaneously, HyPer relies on a virtual memory snapshot mechanism that is assisted in hardware by the Operating System (OS). It maintains consistent snapshots for both OLAP and OLTP queries. Upon OLAP request, HyPer clones the entire database and forks a new process using kernel API methods of the OS. This new process is then able to work on that consistent snapshot without any interference with the main database. In multi-core machines, multi OLAP threads can be launched simultaneously as long as they only read on a private snapshot. By relying on hardware mechanisms, HyPer is demonstrated to be fast and high performance.

3.4 Summary

In this chapter, we studied the design of data-management systems, focusing on the scalability aspects of their architectures. In order to cope with the increasing workloads, data-management systems follow two approaches for scalability: scale horizontally (scale-out), and scale vertically (scale-up). While most of the studied systems are designed to scale-out in distributed environments, recent trends showed promising approaches based on big memory and multi-core to scale-up.

Despite there is a huge effort on designing scalable data-management systems, existing approaches face many limitations, especially when dealing with new challenges that arise in the context of Big Data management. For instance, existing storage systems do not efficiently support atomic non-contiguous I/O [1], which results in poor performance in processing of Big Volume of data. Furthermore, in the context of Big Variety, we observe a shortcoming of specialized storage systems optimized for array-oriented data model. This is also one of the challenges that we will address in this thesis.

In the next chapter, we will discuss in detail BlobSeer [66] as an interesting case study of a distributed data-management system. We select BlobSeer because it features several novel key design principles that we rely on this thesis. Next, we present our contributions on designing scalable data-management systems along the classification introduced in this chapter.

[1] Non-contiguous I/O refers to the access of non-contiguous regions from a file within a single I/O call

Chapter 4

Case study - BlobSeer: a versioning-based data storage service

Contents

BLOBSEER [66] is a versioning-oriented data sharing service. It is specifically designed to meet the requirements of data-intensive applications that are distributed at a large scale. (1) *A scalable aggregation of storage space* from a large number of participating machines with minimal overhead. (2) Support to store *huge data objects* while providing *efficient fine-grain access* to data subsets. (3) The ability to sustain a *high throughput under heavy access concurrency*. We selected BlobSeer as a case study for this chapter, as it is the building blocks for a part of our contributions in the context of scalable data storage.

4.1 Design overview

BlobSeer is the core project of KerData team, INRIA Rennes, Brittany, France. The main features of BlobSeer are *Data as BLOBs*, *data striping*, *distributed metadata management* and *versioning-based concurrency control* to distribute the I/O workload at a large scale, and avoid the need for access synchronization both at data and metadata level. As demonstrated by Nicolae et al. [66, 44, 67], these features are crucial in achieving a high aggregated throughput under concurrency.

Data as BLOBs. Data is abstracted in BlobSeer as long sequences of bytes called BLOBs (Binary Large OBject). These BLOBs are manipulated through a simple access interface that enables creating a BLOB, reading/writing a range of *size* bytes from/to a BLOB starting at a specified *offset* and appending a sequence of *size* bytes to the BLOB. BlobSeer addresses the management of huge BLOBs that may grow in size up to terabytes (TB) but also allows *fine-grained access* to small parts of each BLOB. Therefore, BlobSeer can be a generic storage engine for managing any kind of un-structured data.

To facilitate data sharing, BlobSeer maintains a flat namespace in which each BLOB is uniquely identified by a globally shared identifier (ID). Given a BLOB `ID`, users can access the BLOB's data in a *data-location transparent* manner, without being provided the location of that specific BLOB.

Data striping. As a BLOB can go beyond TB size, it is impossible and/or inefficient to store each BLOB in a centralized fashion. To improve the performance of data accesses, BlobSeer fragments each BLOB into equally-sized *chunks* and distributes them across multiple storage nodes. This mechanism is called *data striping* and is widely used in other distributed data-management systems. *Data striping* enables load-balancing by directing accesses to different *chunks* to different storage nodes.

Because the value of the chunk size has a big impact on the performance of both writing and reading operations, BlobSeer allows users to configure this parameter independently for each BLOB. The chunk size is specified during the creation of the BLOB and need to be fine-tuned according to the access patterns of the applications. If the chunk size is too large, there is high chance that multiple concurrent readers and writers will access the same *chunks*. This may create bottlenecks at the local level of the storage servers being accessed. In the inverse case, too small chunk sizes lead to the overhead of initiating many network connections and of transferring many small pieces of data.

Distributed metadata management. Since each BLOB is split into *chunks* stored on a large number of storage nodes, metadata is needed in order to know which chunks belong to which BLOB and where they are located. BlobSeer favors a distributed metadata-management scheme based on a Distributed Hash Table (DHT). Metadata pieces are distributed over a number of metadata providers, which alleviates pressure on each particular metadata provider, while a higher aggregated metadata storage capability can be achieved. Additionally, a standard replication mechanism can be built within a DHT to enable higher metadata availability.

Versioning-based concurrency control. To achieve high throughput under heavy concurrency, BlobSeer relies on a *versioning-based concurrency control*. Instead of keeping only the current state (latest version) of each BLOB, BlobSeer remembers any modification on any particular BLOB. Essentially, each modification made by a new `WRITE` operation results in a new BLOB version. BlobSeer only stores the incremental update that differentiates the current new version from the previous version. Thanks to versioning-enabled metadata, each version can be seen as the whole BLOB obtained after a successful `WRITE` operation.

By isolating updates in different versions, concurrent readers and writers can work independently as long as they access distinct BLOB versions. BlobSeer guarantees that

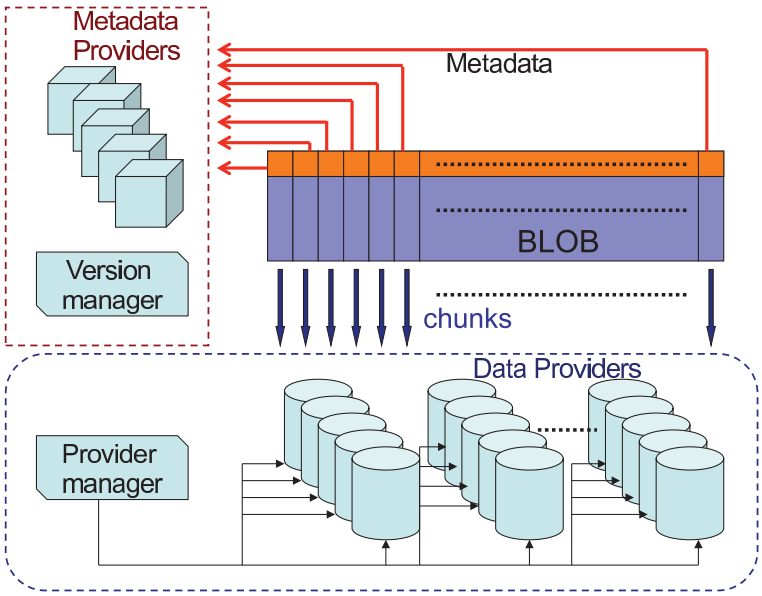


Figure 4.1: The architecture of BlobSeer.

a new version is unveiled to readers only when it is released by the writer. Concurrent writers are handled in a way that they can perform data I/O in parallel to create various versions, while metadata management is finally responsible for serializing versions in a consistent order.

4.2 Architecture

The system consists of distributed processes (Figure 4.1), that communicate through remote procedure calls (RPCs). A physical node can run one or more processes and, at the same time, may play multiple roles from the ones mentioned below.

Data providers. The *data providers* physically store the chunks. Each *data provider* is simply a local key-value store, which supports accesses to a particular *chunk* given a *chunk ID*. *Data providers* can be configured to use different persistent layers such as BerkeleyDB [68], an efficient embedded database, or just keep *chunks* in main memory. New data providers may dynamically join and leave the system.

Provider manager. The *provider manager* keeps information about the available storage space and schedules the placement of newly generated *chunks*. It employs a configurable chunk distribution strategy to maximize the data distribution benefits with respect to the needs of the application. The default strategy implemented in BlobSeer simply assigns new *chunks* to available *data providers* in a round-robin fashion.

Metadata providers. The *metadata providers* physically store the metadata that allow identifying the *chunks* that make up a snapshot version of a particular BLOB. BlobSeer employs a distributed metadata management organized as a Distributed Hash Table (DHT) to enhance concurrent access to metadata.

Version manager. The *version manager* is in charge of assigning new snapshot version numbers to writers and to unveil these new snapshots to readers. It is done so as to offer the illusion of instant snapshot generation, while guaranteeing total ordering and atomicity. The *version manager* is the key component of BlobSeer, the only serialization point, but is designed to not involve in actual metadata and data I/O. This approach keeps the *version manager* lightweight and minimizes synchronization.

Clients. BlobSeer exposes a client interface to make available its data-management service to high-level applications. When linked to BlobSeer's client library, application can perform the following operations: `CREATE` a BLOB, `READ`, `WRITE`, and `APPEND` contiguous ranges of bytes on a specific BLOB.

4.3 Versioning-based access interface

Following the design principles mentioned in Section 4.1, BlobSeer provides a versioning-based access interface to allow clients to manipulate BLOBs with respect to the versioning features, including the creations of a new version and the possibility to read data content of a specific BLOB version.

```
CREATE(id)
```

By invoking the `CREATE` primitive, a new BLOB with size 0 is created in the system with an identifier `id`. The `BLOB id` must be globally unique and is needed for further access operations.

```
WRITE(id, buffer, offset, size)
APPEND(id, buffer, size)
```

The `WRITE` or `APPEND` primitives modify a BLOB identified by the given `id`, by writing contents of a `buffer` of length `size` at a specified `offset` or the end of the BLOB. Each function call generates a new version of the BLOB that is assigned a version number, incrementally generated by the version manager. Remember that `WRITE` and `APPEND` only allow updating a contiguous range within a BLOB.

```
READ(id, v, buffer, offset, size)
```

A `READ` operation accesses a contiguous segment (specified by an `offset` and a `size`) from a BLOB (specified by its `id`) and copies it into a given `buffer`. The desired version of the BLOB from which the segment must be taken can be provided in `v`. In case `v` is missing, BlobSeer assumes by default that the latest version of the BLOB is accessed.

```
CLONE(id, v)
```

The `CLONE` primitive enables users to create a new BLOB based on an existing one. The new BLOB is a "shadow" copy of the version `v` of the BLOB identified by `id`. BlobSeer

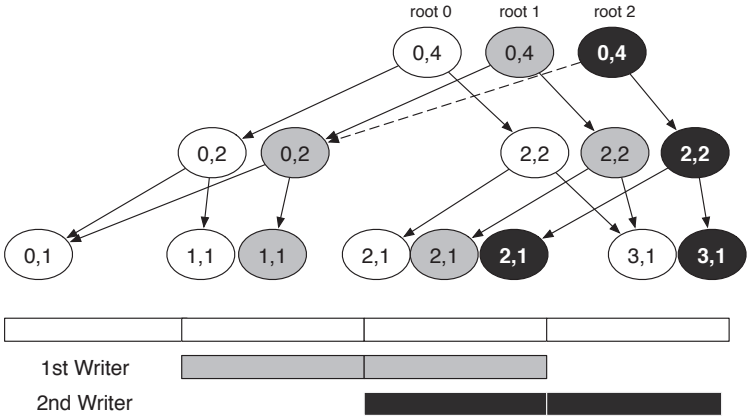


Figure 4.2: Metadata representation: whole subtrees are shared among snapshot versions.

implements the `CLONE` primitive in a smart way that does not require a full copy of data form the source BLOB to the destination BLOB. Instead, `CLONE` is done at metadata level by simply duplicating the metadata of version `v` of `BLOB id`.

Furthermore, BlobSeer's client library also exposes additional primitives for other management purposes such as: switching between BLOB versions, requesting the size of a BLOB, getting the latest available version of a BLOB, etc.

4.4 Distributed metadata management

BlobSeer organizes metadata as a *distributed segment tree* [69]: one such tree is associated to each snapshot of a given BLOB `id`. A segment tree is a binary tree in which each node is associated to a range of the BLOB, delimited by *offset* and *size*. We say that the node *covers* the range (*offset*, *size*). The root covers the whole BLOB snapshot, while the leaves cover single chunks (i.e., keep information about the data providers that store the chunk). Chunk size (in the order of KBs) is a configurable parameter per BLOB. For each node that is not a leaf, the left child covers the first half of the range, and the right child covers the second half. The segment tree itself is distributed at fine granularity among multiple metadata providers that form a DHT (Distributed Hash Table). This is done for scalability reasons, as a centralized solution becomes a bottleneck under concurrent accesses.

In order to avoid the overhead of rebuilding the whole segment tree for each new snapshot (which consumes both space and time), entire subtrees are shared among the snapshots, as shown on Figure 4.2. Root 0 represents the initial metadata tree for a BLOB consisting of 4 *chunks*. In our example, `chunk size` is set to 1 so that the root covers the range (0,4) (offset = 0, and size = 4 chunks). If we follow down the tree, each of the two inner nodes covers 2 *chunks* while each leaf covers exactly only one *chunk*. Regarding the metadata trees represented by root 1 and root 2, those trees belong to different snapshots but they shared leaves and inner nodes between them and with root 0.

To understand how BlobSeer's metadata management gets involved in I/O operations, especially in concurrency, we study two main cases: reading and writing.

WRITE. On writing, the client first contacts the provider manager to get a list of available data providers. It then splits the data range in *chunks* and distributes them over the given data providers in parallel. When *chunks* are written, the client contacts the version manager to get a version for the new write. Next, the metadata tree for the new snapshot is generated in a bottom-up fashion, starting from leaves to the new root.

Because of concurrency, multiple clients can simultaneously contact the version manager. For example in Figure 4.2, let us assume that there are two concurrent writers on the BLOB version identified by root 0 (version 0). BlobSeer relies on the version manager to decide upon the order of concurrent writers. Indeed, the version manager is the single serialization point of BlobSeer where concurrent writes are assigned new versions in a *first comes first served* order. In our example, the first writer gets version 1, the second writer gets version 2. Since the metadata nodes are shared across three versions 0, 1, and 2, we may think that concurrent writers cannot generate metadata trees in parallel.

To enable the parallelization in generating metadata trees, the version manager not only assigns a version for each writer, but also informs it about the given versions and the access ranges of the possible concurrent writers with lower version numbers. Based on this information, each writer can guess which the tree nodes will be generated by other writers. It can be done by calculating the identity of a tree node, which is just a triple of version, offset, size. Therefore, each writer can generate its metadata tree in an isolated fashion on the assumption that the shared tree nodes will be eventually generated by the other writers.

READ. On reading, the client first contacts the version manager to specify which version of which BLOB it wants to access. If that version exists, then the version manager sends back the root of the corresponding metadata tree to the client. Starting from the given root, the client can traverse down the metadata tree to the leaves that cover the desired data range. The client can then use the information in tree leaves to know which data providers it has to access for data.

Because metadata and data are never overwritten as explained in data writing, multiple readers can read simultaneously even in parallel with writers, as long as BlobSeer keeps writers and readers not accessing the same version. It is the role of the version manager to unveil new version to readers only when the writer have finished to generate it.

4.5 Summary

This chapter describes the design and the architecture of BlobSeer, a versioning-oriented large-scale data-management service. BlobSeer targets data-intensive distributed applications that need to manage massive unstructured data at large-scale. We focused on BlobSeer's distributed metadata management and its versioning-oriented interface as they are the main novel features in comparison with other distributed storage systems. By using BlobSeer as a building block for some of our systems, we were able to speed up the creation of several prototypes in order to validate our contributions. These are discussed in the next chapters.

Part II

Scalable distributed storage systems for data-intensive HPC

Chapter 5

Data-intensive HPC

Contents

In this chapter, we address the common practices in data-intensive HPC and highlight some challenges in building scalable storage systems in such an environment.

High performance computing (HPC) plays an important role in our modern life nowadays. We depend on HPC for a wide range of activities in both nature and social sciences. Researchers in institutes, universities and government labs use HPC systems and applications to study weather and climate, bioscience, chemistry, energy, etc. Engineers in industries rely on HPC to design almost every product we use. HPC exists in mechanical simulation, package design, automotive manufacturing, volcanic simulation, financial simulation, etc.

There is no clear definition of HPC. It refers to the use of parallel processing on high performance clusters for efficiently solving advanced problems that cannot be afforded by a single machine. Those advanced problems often set strict constrains on timing and precision so that a huge amount of resources is needed to address them.

Because of its nature to address big problems, HPC applications are increasingly becoming data-intensive. High-resolution simulations of natural phenomena (Cloud Model 1 - CM1 [70]), climate modeling (Weather Research and Forecasting Model - WRF [71]), large-scale image analysis, etc. generate and consume large volumes of data. Such applications currently manipulate data volumes in the Petabyte scale. With the growing trend of data sizes, we are rapidly advancing towards the Exabyte scale. In the context of Big Data, data-intensive HPC applications emphasize the *Big Volume* aspect.

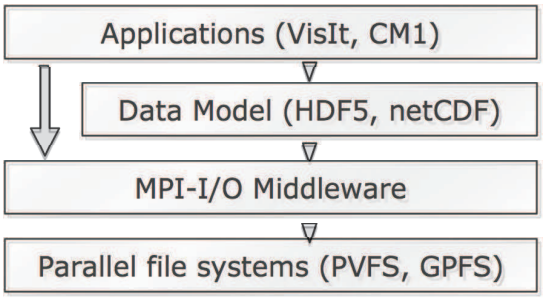


Figure 5.1: Typical parallel I/O stack.

5.1 Parallel I/O

Data-intensive HPC applications typically consist of a massive number of processes deployed on the compute nodes of HPC clusters. During each execution, those processes need to coordinate for reading input data and writing output data. A typical execution may also involve checkpointing where the intermediate data state is written to the underlying storage systems after a number of iteration steps. In all of those cases, the I/O access pattern exhibited by data-intensive HPC applications relies to parallel I/O.

5.1.1 Parallel I/O stack

A typical parallel I/O stack consists of 4 layers, as shown on Figure 5.1.

Parallel file systems. At the lowest level is the parallel file system that provides efficient access to data files. Its role is to federate storage resources and to coordinate accesses to files and directories in a consistent manner. The parallel file system exhibits a global file system namespace that is typically seen through a UNIX-like interface. Some parallel file systems such as PVFS allow users to access not only contiguous regions of files but also non-contiguous regions. This extension is provided to favor access patterns generated by parallel applications on upper layers.

I/O Middleware: MPI-I/O. On top of the parallel file system typically sits the MPI-I/O implementation, whose interface is part of the MPI-2 interface specification [72]. MPI-I/O middleware defines a standard API and implements several optimizations such as collective I/O and data caching. By providing a standard API, MPI-I/O can leverage file system-specific interfaces and optimizations transparently to upper layers. The role of MPI-I/O middleware is to translate accesses generated by applications or high-level I/O libraries to file system-specific accesses that can be performed efficiently by the underlying file systems.

High-level I/O library (data model). The MPI-I/O interface provides performance and portability, but the interface is relatively limited as it only allows accesses to unstructured data. However most of data-intensive scientific applications work with structured data, and MPI-I/O is simply not sufficient to represent those complex data models. For this reason, high-level I/O libraries installed on top of MPI-I/O middleware

are necessary (e.g., Parallel HDF5 [73] or netCDF [74]). They allow applications to describe easier complex data models.

Parallel applications. Parallel applications such as VisIt [75], Cloud Model 1 - (CM1) [70] rely on the I/O interface provided by the MPI-I/O middleware or by a high-level I/O library to access data in parallel file systems.

5.1.2 Zoom on MPI-I/O optimizations

MPI (Message Passing Interface) [8] is the dominant parallel programing model in HPC environments. Among MPI components, MPI-I/O [76] was developed to standardize the parallel I/O access interface with the goal to achieve portability and performance.

Non-contiguous I/O. MPI-I/O allows accessing non-contiguous regions from a file into non-contiguous memory locations within a single I/O call. Although most parallel file systems do not implement this I/O functionality, non-contiguous I/O accesses are common in many data-intensive HPC applications [77, 78]. Therefore, the ability to specify non-contiguous accesses in MPI-I/O helps bridging the gap between parallel file system interface and application needs.

In the case parallel file system does not support non-contiguous I/O, one simple approach to implement a non-contiguous I/O is to access each contiguous portion separately by using regular I/O function calls of the file system. However, such implementation does not feature any optimization. It often results in a large number of independent small I/O requests to the underlying file system, eventually degrading drastically I/O performance.

ROMIO, a MPI-I/O implementation, performs an optimization for non-contiguous I/O accesses by using a mechanism called *data sieving*. This mechanism tries to make a single contiguous I/O request to the file system that covers the first requested byte up to the last requested byte. On reading, the output data is then stored in temporary buffer in main memory, and only the requested non-contiguous portions are extracted and copied to user's buffer. On writing, a read-before-write is performed in order to make up a contiguous region to write to the file.

Collective I/O. Almost all parallel file systems only support independent I/O. As they provide only Unix-like interface, successive I/O requests are independently served in an isolated fashion. In context of large parallel computing where multiple distributed processes coordinate to solve a big problem, this form of I/O access does not capture the global picture of the access patterns, ignoring precious information for optimization.

One of the most important access optimizations done in the MPI-I/O specification is collective I/O. In contrast to independent I/O, collective I/O leverages the global information about parallel processes to merge, to aggregate accesses in order to favor the underlying storage system. Such optimizations can significantly improve I/O performance.

There are several advantages in performing I/O collectively. First, concurrent I/O requests that are overlapped and redundant can be filtered. Second, collective I/O can merge, aggregate many small and non-contiguous requests into smaller number

of large and contiguous I/O calls. Although each individual process may perform non-contiguous I/O, it is likely that the requests of multiple processes constitute one or many larger contiguous portions of a file. In this sense, collective I/O eliminate non-contiguous patterns on the underlying systems.

The default mechanism to implement collective I/O in ROMIO implementation refers to a two-phase strategy [79]. A communication phase must happen before the I/O phase. Its role is to aggregate I/O requests and to decide which processes will perform which parts of the aggregated I/O in the second phase. In other words, ROMIO tries to propose an optimized I/O access strategy based on the global information of concurrent accesses.

5.2 Storage challenges in data-intensive HPC

As I/O is the new bottleneck, storage system for HPC is becoming a critical factor of the overall performance of applications. Unless using an appropriate storage architecture, application performance will be disappointing regardless of how powerful the HPC cluster is. This section presents several storage challenges that are critical and relevant to our contribution.

Massive data size. Data-intensive HPC applications tend to generate and consume an amount of data that is dramatically increasing. Such applications currently manipulate data volumes in the Petabyte scale and with the growing trend of data sizes we are rapidly advancing towards the Exabyte scale. Facing this rapid pace, current I/O architectures and system designs are often overwhelmed by this immensity of data. Obviously, the poor I/O throughput of the underlying storage systems creates bad impacts on the overall performance of HPC environments. One question has been raised: "How do we efficiently manage this huge amount of data that our current storage architecture was not designed for?"

CHALLENGE: How to manage massive data size efficiently?

Massive parallelization. Together with the growth of the data volumes generated by applications, the number of parallel processes participating in the computation is increasing dramatically. Today, scientific simulation applications such as cloud modeling CM1 easily scale to tens of thousand processes. During each computation iteration, those processes perform I/O simultaneously, creating a massive number of concurrent I/O operations that is proportional to the increasing number of client processes. While the computational power of high-performance computing systems increases in Moore's law, innovation in I/O systems does not make progress at the same rate. This fact results in the new scalability challenges under massive parallelization within current storage technologies: I/O jitter, poor I/O throughput in concurrency, etc.

Several approaches have been proposed to address those new challenges. Nawad et al. [80], introduced a scalable I/O forwarding framework for HPC systems. Dorier et al., introduced Damaris [81] (Dedicated Adaptive Module for Application's Resources Inline Steering) that leverages dedicated cores for I/O services. The common point of

both those approaches was to reduce file system traffic under massive parallelization by aggregating, rescheduling, and caching I/O requests. Basically, the dedicated I/O nodes, or dedicated I/O cores take responsibility of performing I/O on behalf of the compute nodes, thus consequently reducing the number of concurrent I/O requests to the underlying storage systems.

CHALLENGE: How to deal with massive parallelization?

I/O atomicity. For concurrent I/O operations, atomicity semantics defines the outcome of overlapping regions in the file that are simultaneously read/written by multiple processes. Without atomic guarantees, the results state of the non-contiguous regions accessed by multiple processes may be inconsistent: the data of some contiguous parts may come from one process whereas the data of some other parts may come from another processes. When executing HPC scientific applications, guaranteeing atomicity of I/O operations is a crucial issue because those applications often perform a large number of overlapping I/O operations. Un-defined data at any intermediate iteration step, means inaccurate input for the sub-sequential steps, creating inaccurate final results. This consequence is undesired especially not only for the critical role of those applications but also for the expensive of rerunning the entire calculation process in the HPC environments.

The current approaches to implement I/O atomicity do not scale. Either storage systems compromise atomicity to get better performance, either they rely on locking mechanism to ensure atomicity while performing poorly at large-scale. One question should be addressed: "What is the novel approach to guarantee I/O atomicity at minimal cost?"

CHALLENGE: How to guarantee I/O atomicity at minimal cost?

Array-oriented data organization model. Many established storage solutions such as parallel file systems and database management systems strive to achieve high-performance at large scale. However, one major difficulty is to achieve performance scalability of data accesses under concurrency. One limitation comes from the fact that most existing solutions expose data access models (e.g., file systems, structured databases) that are too general and do not exactly match the natural requirements of the application. This forces the application developer to either adapt to the exposed data access model or to use an intermediate layer that performs a translation. In either case, this mismatch leads to suboptimal data management: as noted in [51], the one-storage-solution-fits-all-needs has reached its limits.

The situation described above highlights an increasing need to specialize the I/O stack to match the requirements of different types of applications. In scientific computing, of particular interest is a large class of applications that represent and manipulate data as huge multi-dimensional arrays [77]. Such applications typically consist of a large number of distributed workers that concurrently process subdomains of those arrays. In this context, an array-oriented data model will potentially help sustaining a high throughput for such parallel array processing.

CHALLENGE: How to design storage systems optimized for parallel array processing?

5.3 Summary

In this chapter, we have presented an overview of data-intensive HPC and argued that the data-management systems required in data-intensive HPC are one type of Big Data management. Indeed, those applications process "Big Volume" of data that are growing towards Exabyte scale. To be able to design scalable storage systems for data-intensive HPC, we studied the current parallel I/O frameworks and pointed out some challenges that need to be investigated. Our solutions to those challenges are then presented in the next chapters.

Chapter 6

Providing efficient support for MPI-I/O atomicity based on versioning

Contents

This chapter is mainly extracted from the paper: *Efficient support for MPI-IO atomicity based on versioning.* Tran V.-T., Nicolae B., Antoniu G., Bougé L. In Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011), 514 - 523, Newport Beach, May 2011.

We consider the challenge of building data-management systems that meet an important requirement of today's data-intensive HPC applications: to provide a high I/O throughput while supporting highly concurrent data accesses. In this context, many applications rely on MPI-I/O and require atomic, non-contiguous I/O operations that concurrently access shared data of the underlying storage systems.

Because many widely used storage back-ends for HPC systems such as Lustre [82] and GPFS [83] strictly enforce the POSIX [46] access model, they do not support atomic, non-contiguous I/O operations. As a result, ROMIO [76], the MPI-I/O implementation currently in circulation is forced to implement this lacking features at MPI-I/O level through locking-base schemes: writes simply lock the smallest contiguous region of the file that covers all non-contiguous regions that need to be written. Under a high degree of concurrency, such an approach is inefficient and becomes a major source of bottleneck.

In this chapter, we address this shortcoming of existing approaches by optimizing the storage back-end specifically for the access pattern mentioned above. We propose a novel versioning-based scheme that offers better isolation and avoids the need to perform expensive synchronization. The key idea is to use multiple snapshots of the same data. Those snapshots offer a consistent view of the globally shared file, thanks to an efficient ordering and resolution of overlapping at metadata level. This enables high throughput under concurrency, while guaranteeing atomicity.

The contributions in this chapter are summarized as follows.

- We introduce a set of generic design principles that leverage versioning techniques and data striping to build a storage back-end that explicitly optimizes for non-contiguous, non-conflicting, overlapped I/O accesses under MPI atomicity guarantees.
- We describe a prototype built along this idea, based on BlobSeer, a versioning-enabled, concurrency-optimized data management service that was integrated with ROMIO.
- We report on a series of experiments performed with custom as well as standard MPI-I/O benchmarks specifically written for the applications that we target and show improvements in aggregated throughput under concurrency between 3.5 to 10 times higher than what state-of-art locking-based approaches can deliver.

6.1 Problem description

In a large class of scientific applications, especially large-scale simulations, input and output data represents huge spatial domains made of billions of cells associated with a set of parameters (e.g., temperature, pressure, etc.). In order to process such vast amounts of data efficiently, the spatial domain is split into subdomains that are distributed and processed by a large number of compute elements, typically MPI processes.

The computations performed on the subdomains are not completely independent: typically, the value of a cell throughout the simulation depends on the state of the neighboring cells. Thus, cells at the border of subdomains (called "ghost cells") are shared by multiple MPI processes. In order to avoid repeated exchanges of border cells between MPI processes during the simulation, a large class of applications [77, 78, 84, 85] partition the spatial domain in such way that the resulting subdomains overlap at their borders. Figure 6.1(a) depicts an

example of a 2D space partitioned in 3×3 overlapped subdomains, each being handled by one of processes $P_1, \ldots, P_9$.

At each iteration, the MPI processes typically dump their subdomains in parallel into a globally shared file, which is then later used to interpret and/or visualize the results. Since the spatial domain is a multi-dimensional structure that is stored as a single, flat sequence of bytes, the data corresponding to each subdomain maps to a set of non-contiguous regions within the file. This in turn translates to a write-intensive access pattern where the MPI processes *concurrently write a set of non-contiguous, overlapping regions in the same file*.

Ensuring a consistent output is not a trivial issue. If all processes independently write the non-contiguous regions of their subdomains in the file, this may lead to a situation where the overlapped regions are the result of an inconsistent interleaving. Such a case is depicted on Figure 6.1(b), where two MPI processes, P_1 and P_2, concurrently write their respective subdomains. Only two consistent states are possible, where all the non-contiguous regions of P_1 and P_2 are *atomically* written into the file. They only differ by the order in which this happened, which for most applications is not important in practice: both results are acceptable under the assumption that P_1 and P_2 are both able to compute consistent cell values for the overlapped region, such that the difference between them does not affect the global outcome of the simulation.

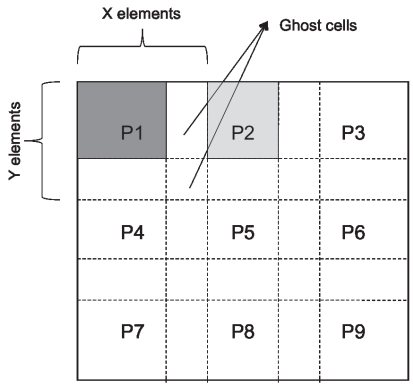
In an effort to standardize these access patterns, the MPI 2.0 standard [76] defines a specialized I/O interface, *MPI-I/O*, that enables read and write primitives to accept complex data types as parameters. These data types can represent a whole set of non-contiguous regions rather than a single contiguous region, as is the case of the POSIX read/write primitives. Thus, the problem of obtaining a consistent output is equivalent to guaranteeing atomicity for the MPI-I/O primitives, which is referred to as *MPI atomicity*. More precisely, MPI atomicity guarantees that in concurrent, overlapping MPI I/O operations, the results of the overlapped regions shall contain data from only one of the MPI processes that participates in the I/O operations.

Large-scale applications need to compute huge domains that are distributed among a large number of processes. Under these circumstances, guaranteeing MPI atomicity in a scalable fashion is difficult to achieve: there is a need to sustain a high data access throughput despite a growing number of concurrent processes. This chapter addresses precisely this problem, facilitating an efficient implementation of the MPI-I/O standard, which in turn benefits a large class of MPI applications.

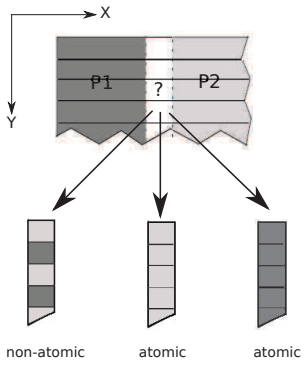
6.2 System architecture

6.2.1 Design principles

We propose a general approach to solve the issue of enabling a high throughput under concurrency for writes of non-contiguous, overlapped regions under MPI atomicity guarantees. This approach relies on three key design principles.



(a) 2D array partitioning with overlapping at the border.



(b) An example of two concurrent overlapping writes.

Figure 6.1: Problem description: partitioning of spatial domains into overlapped subdomains and the resulting I/O access patterns and consistency issues.

Dedicated API at the level of the storage back-end

Traditional approaches address the problem presented in Section 6.1 by implementing the MPI-I/O layer on top of the POSIX access interface which does not provide access to non-contiguous blocks. The rationale behind this is to be able to easily plug in various storage back-ends without the need to rewrite the MPI-I/O layer. However, this advantage comes at a high price: the MPI-I/O layer needs to guarantee MPI atomicity through the POSIX consistency model, which essentially implies the need to build complex locking schemes.

Such approaches greatly limit the potential to introduce optimizations in our context, because the POSIX access model was not originally designed for non-contiguous access patterns. For this reason, we propose to extend the storage back-end with a data access interface that closely matches the MPI-I/O read/write primitives. Using this approach circumvents the need to translate to a different consistency model and enables designing a better concurrency control scheme.

Data striping

The need to process huge spatial domain leads to an increasing trend in data sizes. This increasing trend can be observed not only on the total amount of data, but also on the data sizes that need to be individually handled by each process. As a general rule, the computation-to-I/O ratio is steadily decreasing, which means that the performance of the whole application depends more and more on the performance of the I/O.

In this context, storing the input and output files in a centralized fashion clearly does not scale. Data striping is a well-known technique to address this issue: the file where the spatial domain is stored can be split into *chunks* that are distributed among multiple storage elements. Using a load-balancing allocation strategy that redirects write operations to different storage elements in a round-robin fashion enables the distribution of the I/O workload among the storage elements, which ultimately increases the overall throughput that can be achieved.

Versioning as a key to enable atomic data accesses under concurrency

Most storage back-ends manipulate a single version of the file at a time under concurrency. For this reason, a locking-based mechanism is needed in order to synchronize accesses to the file. In our context, locking can become a major source of overhead, even if the storage back-end was designed to deliver high throughput under concurrency. The problem comes from the fact that locking enables only one single writer at a time to gain exclusive access to a region. Since there are many overlapped regions, this leads to a situation where many writers sit idle while waiting for their turn to lock. This in turn greatly limits the potential to achieve a globally high aggregated throughput.

In order to avoid this issue, we propose a versioning-based access scheme that isolates writers to their own snapshot of the file. This avoids the need to guarantee exclusive access to the same region and therefore the need for writers to wait for each other. Our approach is based on *shadowing* [86]: for each update to the file, the illusion is offered that a new standalone snapshot was created. Of course, only the differences are physically stored and metadata is manipulated in such way that the aforementioned illusion is upheld.

Concretely, we propose to enable concurrent MPI processes to write their non-contiguous regions in complete isolation, without having to care about overlapping and synchronization, which is made possible by keeping data immutable: the new modifications are separately registered without modifying the existing snapshot. It is at the metadata level where the ordering is done and the overlapping are resolved in such way as to expose a snapshot of the file that looks as if all modifications were applied in the sequential order of the version numbers. A concrete proposal of how to achieve this in practice is detailed in Section 6.3.1.

6.2.2 A non-contiguous, versioning-oriented access interface

We introduce a series of versioning-oriented primitives that facilitate non-contiguous manipulations of data file.

```
id = CREATE(size)
```

This primitive creates a new data file and associates to it a zero-filled snapshot whose version number is 0 and is `size` bytes long. The data file will be identified by its file handler `id` (the returned value). The `id` is guaranteed to be globally unique.

```
vw = NONCONT_WRITE(id, buffers[], offsets[], sizes[])
```

A `NONCONT_WRITE` initiates the process of generating a new snapshot of the data file (identified by `id`) starting from a list of memory buffers (pointed at in `buffers[]`) that will be written to the non-contiguous regions defined by the lists `offsets[]` and `sizes[]`.

The `NONCONT_WRITE` command does not "know" in advance which snapshot version it will generate, as the updates are totally ordered and internally managed by the storage system. However, after the primitive returns, the caller learns about its assigned snapshot version by consulting the returned value `vw`. Snapshot versions are totally ordered: if the assigned version is `vw`, then the final result is the same as if applying all `NONCONT_WRITE` calls in the

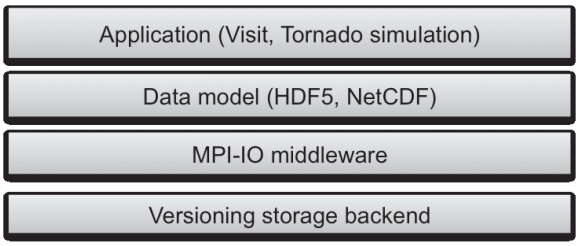


Figure 6.2: Parallel I/O Software Architecture within versioning-based storages.

order of their assigned versions. The snapshot is generated in the background and is said to be *published* when it becomes available to the readers. Note that the primitive may return before snapshot version `vw` is published. The publication of `vw` is eventually done, but it is guaranteed that the generated snapshot will obey MPI atomicity.

```
NONCONT_READ(id, v, buffers[], offsets[], sizes[])
```

A `NONCONT_READ` results in replacing the contents of the memory buffers specified in the list `buffers[]` with the contents of the non-contiguous regions defined by the lists `offsets[]` and `sizes[]` from snapshot version `v` of the data file `id`. If `v` has not yet been published, the read fails. If `v` is omitted, then it is automatically considered to be the latest published version at the time when the read primitive was invoked.

The `NONCONT_*` primitives can be directly mapped to MPI-I/O read/write primitives at higher level. Since MPI-I/O does not expose any versioning semantics, the versioning information is simply discarded. Figure 6.2 describes the integration of our versioning storage back-end to the standard parallel I/O software architecture. Together with MPI I/O middleware and high-level data model interface, a subset of functionality is provided to enable high-performance data access to parallel applications. Since we provide a versioning storage back-end that can understand MPI-I/O atomicity, atomicity of I/O operations can be brought to applications without any translation from/to different atomicity semantics. This is our advantage with regard to the other storage back-ends.

6.3 Implementation

In this section, we describe how to implement this architecture in practice, such that we achieve the design principles introduced in Section 6.2.1. Our approach is to extend a versioning-oriented data sharing service such that it fits our needs as a storage back-end. We have chosen to build our storage back-end on top of *BlobSeer*, a versioning-oriented data sharing service presented previously in Chapter 4.

The choice of building the storage back-end on top of BlobSeer was motivated by two factors. First, BlobSeer supports transparent striping of BLOBs into chunks and enables fine-grain access to them. This in turn enables the storage of each spatial domain directly as a BLOB, which avoids the need to perform data striping explicitly (chunk size is configurable in BlobSeer). Second, BlobSeer offers out-of-the-box support for shadowing by generating a new BLOB snapshot for each fine-grain update while physically storing only the differences.

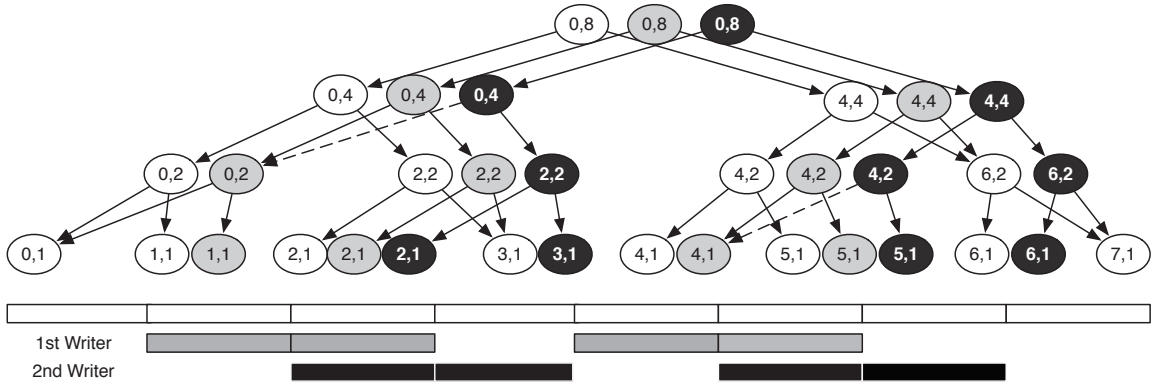


Figure 6.3: Metadata segment trees: whole subtrees are shared among snapshot versions.

This provides a solid foundation to introduce versioning as a key principle to support MPI atomicity.

However, the versioning-oriented access interface as exposed by BlobSeer could not be leveraged directly because it only supports atomic writes and appends of contiguous regions. It would thus imply the need for locking at the level of the MPI-I/O layer in order to support MPI atomicity. As discussed, we want to avoid locking and to introduce optimizations directly at the level of the storage back-end. Therefore, the first step is to extend the access interface of BlobSeer such that it can describe complex non-contiguous data access in a single call. We introduce `NONCONT_WRITE`, `NONCONT_READ` and the new `CREATE` primitives in BlobSeer as presented in Section 6.2.2

6.3.1 Adding support for MPI-atomicity

Our versioning-based approach to ensure MPI atomicity relies on the following idea that non-contiguous regions can be written in parallel as a series of differences that are ordered and consolidated into independent snapshots at the level of the metadata, such that the result is equivalent to the sequential application of all differences in an arbitrary order. The key difficulty in this context is to consolidate the differences at metadata level efficiently such that only consistent snapshots that obey MPI atomicity are published.

Since each massive BLOB snapshot is striped over a large number of storage space providers, metadata enable to remember the location of each chunk in the snapshot, such that it is possible to map non-contiguous regions of the snapshot to the corresponding chunks.

Non-contiguous writes. A `NONCONT_WRITE` operation first stores all chunks that make up the non-contiguous regions onto fresh memory areas on the data providers, after which it builds the metadata segment tree that is associated to the snapshot in a bottom-up manner: first the leaves that hold information about where the chunks were stored and then the inner nodes up towards the root. For example, a write of two non-contiguous regions delimited by (offset, size) = $(1, 2)$ and $(4, 2)$ on a BLOB whose total size is 8 leads to the gray segment tree depicted on Figure 6.3.

Non-contiguous reads. A `NONCONT_READ` operation first obtains the root of the segment tree that corresponds to the snapshot version from which it needs to read, after which it descends in the segment tree towards the leaves that hold information about the chunks that make up the non-contiguous regions. Once these chunks have been established, they are fetched from the data providers and stored in the memory buffers supplied as an argument.

6.3.1.1 Guaranteeing MPI atomicity efficiently under concurrency

Since an already published snapshot is never modified, readers never have to synchronize with writers. The case of concurrent writers is more delicate and needs closer consideration. Concurrent writers can independently write their non-contiguous regions without any need to synchronize, because the non-contiguous regions are ordered and consolidated into consistent snapshots at metadata level. In order to perform this efficiently, we propose three important optimizations.

Minimize ordering overhead. Since any arbitrary ordering that obeys MPI atomicity is valid, our goal is to minimize the time taken to establish the order in which to apply the overlapping. To this end, the writers request a snapshot version to be assigned only after they have successfully finished writing the chunks and need to build their corresponding segment trees. The assignment process, which is the responsibility of the version manager, is very simple and leads to a minimal overhead: versions are assigned on a first-come, first-served basis and practically involve only the atomic incrementation of an internal variable on the version manager. For example, in Figure 6.3, two concurrent writers, black and gray, wrote their non-contiguous regions. Assuming gray finished first, it was assigned version 1, while black was assigned version 2. If black finished first, the order of version assignment would have been the opposite.

Avoid synchronization for concurrent segment tree generation. Once the writer obtained a new snapshot version number from the version manager, it needs to build the metadata segment tree such that it is "weaved" in a consistent fashion with the segment trees of the previous versions (i.e., the correct links to the nodes of the segment trees of the previous versions are established). We call such nodes that are linked against by the segment trees of higher versions *border nodes*. For example, the border nodes of the first writer (gray) on Figure 6.3 are all belonging to the initial version (white). Under concurrency, it can happen that the border nodes of a snapshot version v belong to the segment tree of a lower version that is in the process of being generated itself by a concurrent writer and therefore do not exist yet. For example, the border nodes of the second writer (black) depend on gray nodes that have not necessarily been generated yet.

In order to avoid waiting for such border nodes to be generated, we maintain the list of all concurrent writers on the version manager and feed it as a hint to writers when they request a new snapshot version. Using this information, writers can predict what border nodes will eventually be written by the other concurrent writers and can build virtual links (which we call *metadata forward references*) to them without caring whether they exist or not yet, under the assumption that they will be eventually generated and the segment tree will then become complete. In our example, when black

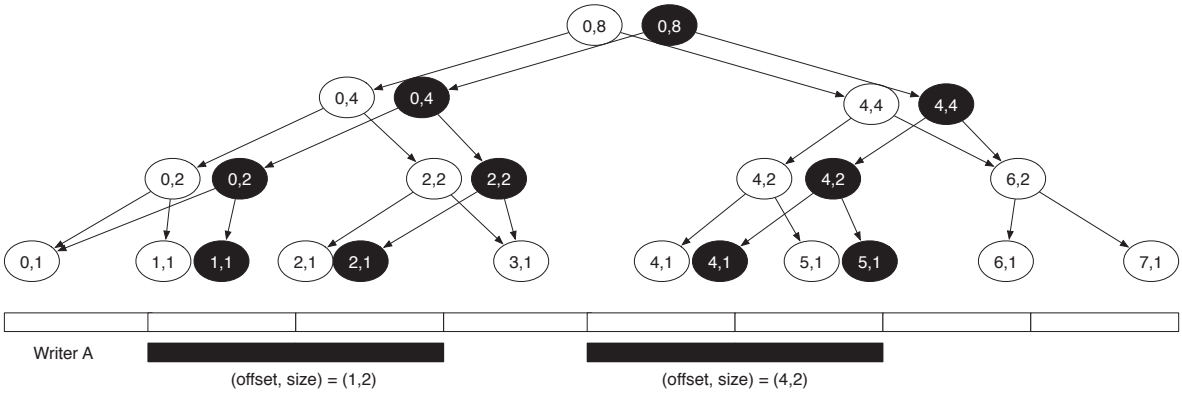


Figure 6.4: Lazy evaluation during border node calculation.

is assigned version number 2 it receives the hint about the existence of gray and the non-contiguous regions that gray intends to write. Using this information, black can establish the metadata forward references (dotted pattern) without waiting for gray to finish building the segment tree. When both gray and black finish, both segment trees are in a consistent state.

BlobSeer was previously designed to compute metadata nodes on the version manager and let the clients only need to send them over the DHT. In our prototype, the segment tree generation is entirely moved from the version manager to the client side. This optimization reduces the load on the version manager, especially for non-contiguous accesses when the number of metadata nodes that needs to be generated is usually bigger than that in the case of contiguous accesses. As the optimization is very significant, it was even integrated back to BlobSeer in its later versions.

Lazy evaluation during border node calculation. The scheme presented above greatly improves the scalability of concurrent segment tree generation, as it enables clients to independently calculate the border nodes in isolation, without synchronizing.

However, in addition to scalability, we aim at high performance, too. To this end, we optimized the border node calculation on the client-side by introducing a lazy evaluation scheme. More precisely, we avoid pre-calculating the border nodes for each contiguous region individually and rather delay their evaluation until the moment when the new tree nodes are generated themselves and the links to their children need to be established. This is particularly important in the context of non-contiguous accesses, because the union of all border nodes taken from each region individually is much larger that the set of border nodes that is effectively needed.

For example, on Figure 6.4 Writer A (black) writes two non-contiguous regions. If each region is taken individually, the white node that covers $(0,4)$ is a border node for the region delimited by (offset, size) = $(4,2)$. Similarly, the white node that covers $(4,4)$ is a border node for the region delimited by (offset, size) = $(1,2)$. However, neither of them are border nodes in the end result, because their gray counterparts will eventually become the children of the gray root $(0,8)$.

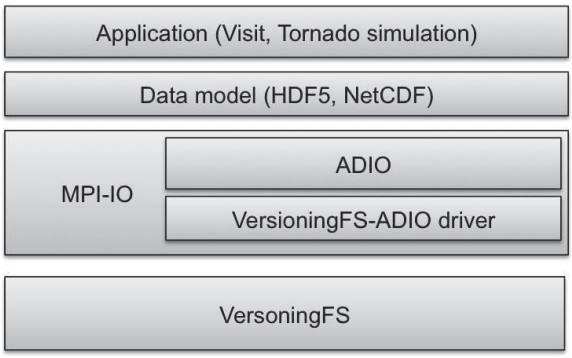


Figure 6.5: Integrating our versioning-oriented storage back-end to ROMIO.

6.3.2 Leveraging the versioning-oriented interface at the level of the MPI-I/O layer

Having obtained a storage back-end implementation that directly optimizes for MPI atomicity, the next step is to efficiently leverage this storage back-end at the level of the MPI-I/O layer. To this end, we used ROMIO [76], a library that is part of popular MPICH2 [8] implementation.

The motivation behind this choice is the fact that ROMIO is designed in a modular fashion, making it easy to plug-in new storage back-ends. Architecturally, ROMIO is divided into three layers: (1) a layer that implements the MPI I/O routines in terms of an abstract I/O device that exposes a generic set of I/O primitives, called *Abstract Device interface for parallel I/O (ADIO)*; (2) a layer that implements common MPI-I/O optimizations that are independent of the storage back-end (such as buffering and collective I/O); and finally (3) a layer that partially implements the ADIO device and needs to be extended for each storage back-end explicitly.

These layers provide a complete support for MPI-I/O at application level. The separation between ROMIO code that is dependent and independent to storage back-end enabled us to build a lightweight ADIO module that maps the interface required by the first layer almost directly on top of the versioning-oriented access interface we introduced. More precisely, an ADIO implementation needs to provide support for both contiguous and non-contiguous writes under MPI atomicity guarantees (Figure 6.5). Our interface proposal is generic enough to handle both scenarios efficiently (non-contiguous `READ` and `WRITE`) using a single primitive call.

6.4 Evaluation

We conducted three series of experiments.

- An evaluation of the scalability of our approach when the same amount of data needs to be read/written into an increasing number of non-contiguous regions by the same client.
- An evaluation of the scalability of our approach when increasing the number of clients

that concurrently write non-contiguous regions in the same file. In this scenario, we considered the extreme case where each of the clients writes a large set of non-contiguous regions that are intentionally selected in such way as to generate a large number of overlapping that need to obey MPI atomicity.

- An evaluation of the performance of our approach using a standard benchmark, *MPI-tile-IO*, that closely simulates the access patterns of real scientific applications that split the input data into overlapped subdomains that need to be concurrently written in the same file under MPI atomicity guarantees.

In the second and the third series of experiments, we compare our approach to the locking-based approach that leverages a POSIX-compatible file system at the level of the MPI-I/O layer, which is the traditional way of addressing MPI atomicity.

In order to perform this comparison, we used two standard building blocks that are available as open-source projects: (1) the Lustre parallel file system [47], version 1.6.4.1, in its role as a high-performance, POSIX-compliant file system and (2) the default ROMIO ADIO module, which is part of the standard MPICH2 release and was specifically written for POSIX-compliant file systems.

Data sieving is a mechanism to handle non-contiguous I/O accesses in ROMIO. This mechanism consists of accessing the smallest contiguous region of the file that covers the non-contiguous I/O request. We turned off data sieving in our experiments. According to the recommendations of Ching et al. [87], doing so greatly improves the performance of Lustre for MPI-I/O. Without data sieving enabled, the ADIO module is able to take advantage of standard POSIX byte-range file locking to lock the smallest contiguous region in the file that covers all non-contiguous regions that need to be read/written. Once this is done, the non-contiguous regions are read/written using a dedicated read/write call for each region individually, after which the lock is released.

6.4.1 Platform description

We performed our experiments on the Grid'5000 [88] testbed, a reconfigurable, controllable and monitor-able experimental Grid platform gathering 9 sites geographically distributed in France. For these experiments we used the nodes of the Rennes cluster, which are outfitted with x86_64 CPUs and 4 GB of RAM. All nodes are equipped with Gigabit Ethernet cards (measured throughput: 117.5 MB/s for TCP sockets with a MTU of 1500 B; latency: 0.1 ms). We invested a significant effort in preparing the experimental setup, by implementing an automated deployment process both for Lustre and BlobSeer.

Our experiments were performed on up to 80 nodes of the Rennes cluster in the following fashion: Lustre (respectively BlobSeer) is deployed on 44 nodes, while the remaining 36 nodes are reserved to deploy a MPI ring where the MPI processes are running (each on a dedicated node).

Lustre was deployed in the following configuration: one metadata server and 43 object storage servers, each on a dedicated machine. For BlobSeer we used the following deployment setup: one version manager and one provider manager deployed on dedicated machines, while the rest of 42 nodes was used to co-deploy the data and metadata providers in pairs, one pair on each node.

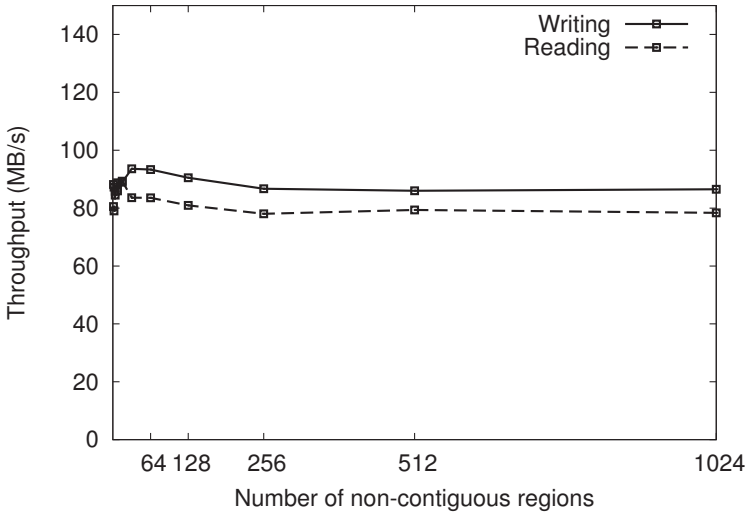


Figure 6.6: Scalability when the same amount of data needs to be written into an increasing number of non-contiguous regions: throughput is maintained almost constant.

6.4.2 Increasing number of non-contiguous regions

In the first series of experiments we evaluate the scalability of our approach when the same amount of data needs to be read/written from/into an increasing the number of non-contiguous regions by the same client.

To this end, we fix the amount of data that is read/written by the client (using `NONCONT_READ` and `NONCONT_WRITE` respectively) at 1 GB. At each step, we double the amount of non-contiguous regions into which this data is split and measure the throughput as observed on the client side. We start with a single contiguous region and end up with 1024 non-contiguous regions.

The results are shown on Figure 6.6. As can be observed, both reads and writes achieve a high throughput that reaches well over 80 MB/s. More importantly, the throughput does not significantly drop when increasing the number of non-contiguous regions, which demonstrates excellent scalability of our approach.

6.4.3 Scalability under concurrency: our approach vs. locking-based

In this scenario we aim at evaluating the scalability of our approach when increasing the number of clients that concurrently write non-contiguous regions in the same file, as compared to the locking-based approach used by Lustre. To this end, we setup a synthetic MPI benchmark that enables us to control the MPI-I/O access patterns generated by the application in such way that we generate a large number of overlapping.

More specifically, the synthetic benchmark corresponds to a special case of the overlapped subdomains depicted in Figure 6.1(a): namely when the subdomains (that need to be written under MPI atomicity guarantees) form a single row (sing column organization does not expose non-contiguous accesses). Each subdomain is a matrix of 1024×1024 elements, with each element 1024 bytes large. This amounts to a total of 1 GB worth of data written

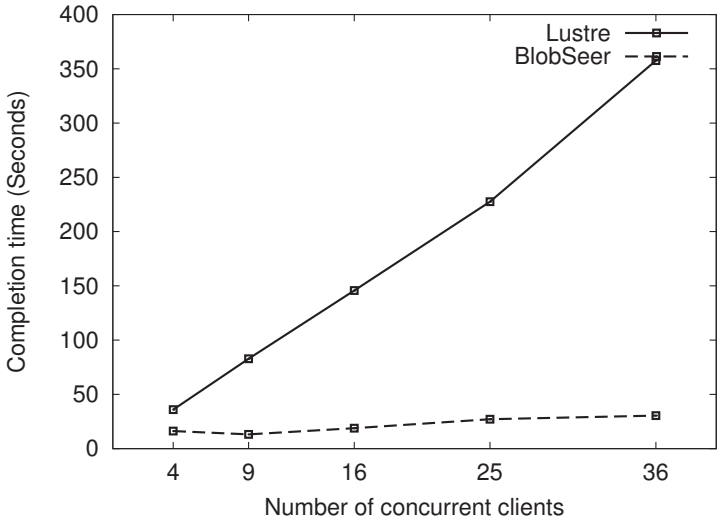


Figure 6.7: Our approach vs. locking-based: completion time for an increasing number of clients that concurrently write a large number of overlapping, non-contiguous regions that are far apart (lower is better).

by each process into 1024 non-contiguous regions, each of which is 1 MB large. Since the subdomains are arranged in a row, every MPI process (except the extremities) share a left and right overlapped subdomain with its left and respectively right neighbor. The size of the overlapping subdomain is fixed at 128×1024 elements, such that each region in the set of non-contiguous regions written by a process overlaps by 128 KB with two other regions belonging to the neighboring processes. This choice leads to a scenario that pushes both approaches to their limits: every single region generates at least one overlapping that needs to be handled in an MPI-atomic fashion.

We varied the number of MPI processes from 4 to 36 and ran the MPI benchmark both for our approach and the locking-based approach using on Lustre. Results are shown on Figure 6.7, where we measure the completion time to run the benchmark (i.e., the time taken by the slowest process to finish), as well as in Figure 6.8, where we measure the total aggregated throughput achieved by all processes (i.e., the total amount of data written by all processes divided by the completion time).

As can be observed, the completion time in the case of Lustre grows almost linearly. Since the processes are arranged in a row, the non-contiguous regions of each process are far apart, which leads to the case in which almost the whole file needs to be locked. Thus, in this extreme scenario, the accesses are practically serialized by the locking-based approach. This trend is confirmed by the total aggregated throughput as well: it remains constant at 114 MB, close to the maximal theoretical limit that can be achieved by a single client.

By contrast, the completion time in the case of our approach experiences a negligible growth, thanks to the fact that it completely avoids synchronization. This trend is confirmed by the total achieved aggregated throughput too: we can observe an increasing trend from about 300 MB/s to about 1500 MB/s for 36 concurrent clients, which more than 10 times higher that the throughput obtained by using Lustre and demonstrates excellent scalability. Obviously, the aggregated throughput does not linearly increase when increasing the

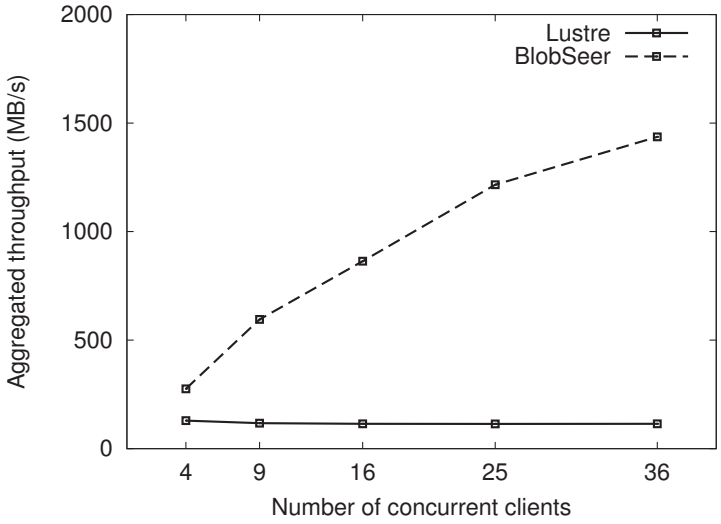


Figure 6.8: Our approach vs. locking-based: aggregated throughput achieved by an increasing number of clients that concurrently write a large number of overlapping, non-contiguous regions that are far apart (higher is better).

number of concurrent clients. This can be explained as the cost of border nodes calculation in each client also increases: more concurrent accesses, more time it takes to scan the list of concurrent accesses sequentially in order to find out which accesses are overlapped to a particular access.

6.4.4 MPI-tile-IO benchmark results

As a last series of experiments we evaluate the performance of our approach using a standard MPI-I/O benchmarking application: *MPI-tile-IO*. This application closely simulates the access patterns of real scientific applications described in Section 6.1: overlapped subdomains that need to be concurrently written in the same file under MPI atomicity guarantees.

MPI-tile-IO sees the underlying data file as a dense two-dimensional set of subdomains (referred to in the benchmark as tiles), each of which is assigned to an MPI process, as shown in Figure 6.1(a). The benchmark consists in measuring the total aggregated throughput (total data written by all processes divided by the total time to complete the benchmark) achieved by all processes when they concurrently write their subdomains to the globally shared file under MPI atomicity guarantees.

This benchmark is highly configurable, enabling fine tuning of the following parameters:

- `nr-tile-x`: number of subdomains in the X dimension;
- `nr-tiles-y`: number of subdomains in the Y dimension;
- `sz-tile-x`: number of elements in the X dimension of each subdomain;
- `sz-tile-y`: number of elements in the Y dimension of each subdomain;
- `sz-element`: size of an element in bytes;

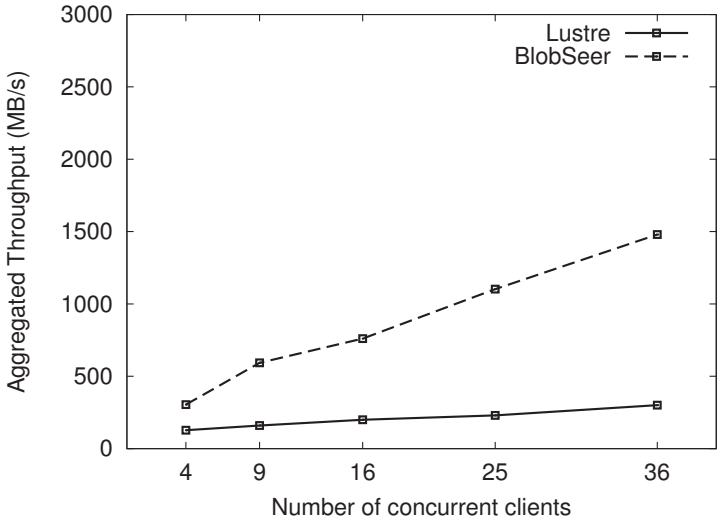


Figure 6.9: $1024 \times 1024 \times 1024$ tile size.

- `overlap-x`: number of elements shared between adjacent subdomains in the X dimension;
- `overlap-y`: number of elements shared between adjacent subdomains in the Y dimension;

We fixed the parameters of the MPI-tile-IO benchmark in such way as to resemble the layout of real applications as closely as possible. More specifically, each subdomain is a 1024×1024 matrix of elements that are 1024 bytes large (i.e. `sz-tile-x` = `sz-tile-y` = `sz-element` = 1024). Unlike the previous series of experiments, where the processes are arranged in a row to obtain an extreme case that pushes both approaches to their limits, this time we use a realistic layout where the processes are arranged in such way that they form a square (i.e. `nr-tile-x` = `nr-tile-y` = $2 \ldots 6$, which corresponds to $4 \ldots 36$ processes). The size of the overlapping was fixed at 128×128 (i.e. `overlap-x` = `overlap-y` = 128). Thus, each process writes 1 GB worth of data, out of which 128 MB is overlapping with each of its neighbors.

We ran the MPI-tile-IO benchmark for both our approach and the locking-based approach using Lustre. The experiment was repeated 5 times and the results were averaged. We observed a very low standard deviation. Figure 6.9 depicts the total aggregated throughput obtained for both approaches.

As can be observed, both approaches scale. Unlike the experiment presented in Section 6.4.3, the choice of arranging the subdomains in such way that they form a square leads to a situation where the non-contiguous regions are closer to each other, which in turn means that a more compact contiguous region needs to be locked by the Lustre-based approach. Under concurrency, this leads to a situation where different contiguous regions of the files can be locked simultaneously, thus enhancing the degree of parallelism for the locking-based approach. Nevertheless, even under such circumstances a significant number of accesses need to be serialized, which in turn enables our approach to outperform the

locking-based approach by almost 6 times for 36 concurrent processes. Again the performance of our system does not increase linearly due to the increment in the cost of border nodes calculation.

6.5 Positioning of the contribution with respect to related work

Previous work has shown that providing MPI atomicity efficiently enough is not a trivial task in practice, especially when dealing with concurrent, non-contiguous I/O, most of which rely on locking-based techniques. Several approaches have been proposed at various levels: at the level of the MPI-I/O layer, at the level of the file system and at the level of application.

A first series of approaches assumes no specific support at the level of the parallel file system This is typically the case of PVFS [89], a widely-used parallel file system which makes the choice of enabling high-performance data access for both contiguous and non-contiguous I/O operations without guaranteeing atomicity at all for I/O operations. For applications where the MPI atomicity requirement needs to be satisfied, a solution (e.g., illustrated in [90]) consists in guaranteeing MPI atomicity at the level of the MPI-I/O layer in a portable, generic way. The lack of guarantees on the semantics of I/O operations provided by the file system comes however at a high cost introduced by the use of coarse-grain locking at a higher level. Typically, the whole file is locked for each I/O request and thus concurrent accesses are serialized, which is an obvious source of overhead.

To avoid the bottleneck in the case of concurrent non-overlapping accesses to the same shared file, an alternative approach of Sehrish et al. [91] proposes to introduce a mechanism for automatic detection of non-overlapping I/O and thus avoid locking in this case. However, as acknowledged by the authors of this approach, the detection mechanism introduces an unnecessary overhead for non-overlapping concurrent I/O. Further optimizations are proposed in [87], where the authors propose a locking-based scheme for non-contiguous I/O which aims to strictly reduce the scope of the locked regions to the areas that are actually accessed. However, this approach cannot avoid serialization for applications which exhibit concurrent overlapping I/O such as the ones described in Section 6.1.

In [92], the authors propose to use process handshaking to avoid/reduce interprocess serialization. This approach enables processes to negotiate with each other who has the right to write to the overlapped regions. However, such an approach can only be applied when every process is aware of all other concurrent processes accessing the same file. This is not suitable for non-collective concurrent I/O operations, where such an assumption does not hold (concurrent I/O requests are typically not aware of each other in this case).

Another class of approaches addresses the case where the underlying parallel file system supports POSIX atomicity. Atomic contiguous I/O can then seamlessly be mapped to atomic read/write primitives provided by the POSIX interface. However, POSIX atomicity alone is not sufficient to provide the necessary atomicity guarantees for applications that exhibit concurrent, non-contiguous I/O operations.

Distributed file systems such as GPFS [83] and Lustre [82] provide POSIX atomicity semantics using a distributed locking approach: locks are stored and managed on the storage servers hosting the objects they control. Whereas POSIX atomicity can simply and directly be leveraged for contiguous I/O operations using *byte range locking*, enabling atomic *non-contiguous* I/O based on POSIX atomicity is not efficient. Consider a set of non-contiguous

byte ranges to be atomically accessed by an individual I/O request. In the default scheme, it is then necessary to lock the smallest contiguous byte range that covers all elements of the set of ranges to be accessed. This leads to unnecessary synchronization and thus to a potential bottleneck, since this contiguous byte range also covers un-accessed data that would not need to be locked.

Finally, given the limitations of the approaches described above, an ultimate solution is to design the parallel application in such a way that MPI atomicity is not required, e.g., by enabling each process of the parallel application to write to a separate file at every iteration. It can then manage this potentially large set of files with custom post-processing tools. The CM1 [70] tornado simulation illustrates this approach. In a typical execution where hundreds of thousands of cores iteratively run the simulation, each core generates a file at every iterations: the total number of files is therefore extremely large. This creates a substantial overhead for metadata management at the file system level and introduces too complex post-processing. Scalability is thus negatively impacted even if the need for MPI-I/O atomicity is avoided by application design.

To summarize, MPI-I/O atomicity has mainly been enabled using locking-based approaches. By contrast, we propose a novel versioning-based mechanism that allows to handle atomic *non-contiguous, overlapping* I/O more efficiently compared to such lock-based solutions.

6.6 Summary

We proposed an original versioning-based mechanism that can be leveraged to efficiently address the atomic I/O needs of data-intensive MPI applications. We are interested in applications involving data-partitioning schemes that exhibit overlapping non-contiguous I/O, where MPI-I/O atomicity needs to be guaranteed under concurrency.

Traditional approaches leverage POSIX-compliant parallel file systems as storage back-ends and employ locking schemes at the level of the MPI-I/O layer. In contrast, we propose to use versioning techniques as a key principle to achieve high throughput under concurrency, while guaranteeing MPI atomicity. We implemented this idea in practice by extending BlobSeer with a non-contiguous data access interface that we directly integrated with ROMIO.

We compared our BlobSeer-based implementation with a standard locking-based approach where we used Lustre as the underlying storage back-end. Our approach demonstrated excellent scalability under concurrency when compared to the Lustre-based approach. It achieved an aggregated throughput ranging from 3.5 times to 10 times higher in several experimental setups, including highly standardized MPI benchmarks specifically designed to measure the performance of MPI-I/O for non-contiguous overlapped writes that need to obey MPI-atomicity semantics.

Chapter 7

Pyramid: a large-scale array-oriented storage system

Contents

This chapter is mainly extracted from the paper: *Towards scalable array-oriented active storage: the Pyramid approach.* Tran V.-T., Nicolae B., Antoniu G. In the ACM SIGOPS Operating Systems Review 46(1):19-25. 2012.

CHAPTER 6 presented our contribution on proposing an original versioning-based mechanism to enable the atomicity of non-contiguous I/O without sacrificing the scalability under high concurrency. In this chapter, we investigate a further optimization to develop specialized storage systems that are capable to deal with specific access

patterns in a scalable fashion. The novel idea is to redesign the physical data organization inside distributed storage systems in such a way that it closely matches how data are accessed by applications.

In the context of data-intensive HPC, a large class of applications focuses on parallel array processing: small parts of huge multi-dimensional arrays are concurrently accessed by a large number of clients, both for reading and writing. A specialized storage system that deals with such an access pattern faces several challenges at the level of data/metadata management. We introduce Pyramid, an array-oriented storage system that addresses these challenges. Experimental evaluation demonstrates substantial scalability improvements brought by Pyramid with respect to state-of-art approaches both in weak and strong scaling scenarios, with gains of 100 % to 150 %.

In the context of Big Data, Pyramid addresses *Big Volume* and *Big Variety* because it features a novel array-oriented data model and provides support for massive data processing.

7.1 Motivation

Many established storage solutions such as parallel file systems and database management systems strive to achieve high-performance at large-scale. However one major difficulty is to achieve performance scalability of data accesses under concurrency. One limitation comes from the fact that most existing solutions expose data access models (e.g., file systems, structured databases) that are too general and do not exactly match the natural requirements of the application. This forces the application developer to either adapt to the exposed data access model, or to use an intermediate layer that performs a translation.

Traditionally, data-intensive HPC applications access the underlying storage systems through the parallel I/O stack. This stack consists of several layers as presented in Section 5.1.1. The lowest layer is the parallel file systems. It is responsible to aggregate the storage space of dedicated storage resources into a single common pool, which is exposed to higher layers using a file access model, typically POSIX [46]. This file access model is leveraged by the I/O middleware (such as MPI-I/O [72]), the layer directly on top of the parallel file system. It is specifically designed to to coordinate and optimize the parallel access patterns of HPC applications, acting as a bridge between these recurring patterns and the more generic file-oriented I/O access model. At this level, data is still in a raw form and has no structure attached to it. Since most HPC-oriented applications do not directly work with raw data but tend to associate a multi-dimensional structure to it, a third layer in form of an I/O library (such as netCDF [74], HDF5 [93], ADIOS [94]) provides the necessary mechanisms to attach structure to the data an to perform multi-dimensional queries.

A major problem with this traditional three-layered I/O stack is the existence of a mismatch between the access models expected at the lower layers. The I/O of the application is funneled from the highest layer down to the parallel file system, at each step going through a translation process that is necessary in order to adapt to the expected access model. These access models (i.e. POSIX and MPI-I/O) are designed to handle the worst-case scenarios for conflicts, synchronization, and coherence of data that is represented in a linear fashion, ignoring the original structure and purpose of the I/O. Thus, the translation process between the layers becomes very costly and incurs a major performance overhead.

One basic example is the presence of non-contiguous I/O at the level of the underlying

storage systems. Consider the the I/O requirements of scientific applications [77, 78, 84, 85] that partition multi-dimensional domains into subdomains that need to be processed in parallel and then stored in a globally shared file. Since the file is a flat sequence of bytes, a subdomain (despite seen by application processes as a single chunk of memory) maps to a series of non-contiguous regions in the file, all of which have to be read/written at once by the same process. Therefore, non-contiguous I/O destroys the simplicity in data access patterns generated by applications while exposing unpredictable and complex access patterns at the level of the storage systems. In this situation, the storage systems are not provided any information for doing any further optimization based on characteristics of applications. As in the previous chapter, non-contiguous I/O are usually expensive, especially when the atomicity of I/O accesses is required.

A stack-based design has its advantages as it is easier to design and to develop each layer independently. Recent work in [95, 51] pointed out that a stack-based design has limited potential for scalability at Exascale. The one-storage-solution-fits-all-needs has reached its limits.

The situation described above highlights an increasing need to specialize the I/O stack to match the requirements of the applications. We argue in favor of a specialized storage system that is designed from scratch to directly match the I/O needs of the application. Such a design effectively "shortcuts" through the I/O stack and gains direct control over the storage resources, setting it free from any unnecessary constraints and enabling it to take more informed decisions, which in turn provides better optimization opportunities.

We target the particular large class of applications that represent and manipulate data as huge multi-dimensional arrays [77]. Such applications typically consist of a large number of distributed workers that concurrently process subdomains of those arrays. We introduce *Pyramid*, a specialized array-oriented storage manager that features multi-dimensional-aware data chunking that splits the initial data in multi-dimensional chunks to optimize parallel array processing.

7.2 Related work

SciDB [96, 97] is an effort to design an open-source database system for scientific data analytics. SciDB departs from the relational database model, offering an array-oriented database model that specifically targets multi-dimensional data. Notably, SciDB proposes two features that are crucial in scientific data analytics: multi-dimensional-aware data striping and data versioning. However, a concrete solution to implement these features is still on-going work. Furthermore, with the upcoming Exascale age, scalable metadata management becomes a critical issue that is insufficiently addressed by SciDB. Our approach aims to address these limitations by proposing a distributed data-management scheme rather than a centralized approach.

Emad et al. introduced ArrayStore [98], a storage manager for complex, parallel array processing. Similarly to SciDB, ArrayStore partitions large arrays into chunks and stores them in a distributed fashion. ArrayStore organizes metadata as R-trees, which are maintained in a centralized fashion. At large scale, this inevitably leads a bottleneck with respect to the scalability of metadata access, because it places a limit on the number of metadata queries that can be concurrently served. Furthermore, ArrayStore is designed as a read-only

storage system. The authors acknowledge that ArrayStore was not optimized to handle updates to multi-dimensional data: this scenario can cause significant performance degradation due to design focus on read performance. Unlike ArrayStore, our Pyramid approach is designed to efficiently support workloads that consist of any mix of concurrent reads and writes.

7.3 General design

Pyramid is inspired by BlobSeer [44, 99], a versioning-oriented storage system specifically optimized to sustain a high throughput under concurrency. BlobSeer, as presented in chapter 4, focuses on unstructured Binary Large OBjects (BLOBs) that represent linear addressing spaces, whereas Pyramid generalizes the same principles for multi-dimensional data and introduces several new features.

7.3.1 Design principles

Our approach relies on a series of key design principles.

Multi-dimensional-aware data chunking

In the context of distributed data managements, the partitioning scheme plays a crucial role: under unfavorable conditions, read and write queries may generate "strided" access patterns (i.e., access small parts from a large number of chunks), which has a negative impact on performance. Indeed, this problem is well-known in current parallel file systems. PVFS, GPFS, and Lustre file systems allow system administrators to specify the *stripe size* that determines how much data is contained in a chunk. If *stripe size* is too small, accessing certain amount of data will involve transferring many small chunks from many different servers. This results in poor performance. If *stripe size* is too large, many data accesses will end up reading/writing small parts of chunks, leading to the need of synchronization within each chunk in order to preserve data consistency.

Furthermore, current parallel file systems have not taken into account the data structures in the partitioning schemes. As data files are considered unstructured and stored as sequences of bytes, each sub-domain of the initial multi-dimensional array data is serialized into multiple non-contiguous ranges in the corresponding data file. This leads to a constrain in parallel file system performance: the *stripe size* should be equal to the size of contiguous segments within the non-contiguous patterns. As a result, the *stripe size* is usually limited to small sizes.

To reduce this effect, we propose to take into account the nature of multi-dimensional data models by splitting the array into subdomains that are equally sized in each dimension. By leveraging this multi-dimensional-aware chunking, the famous non-contiguous accesses can be nearly avoidable at the level of the storage system. The neighbors of cells of the array data have a higher chance of residing in the same chunk irrespective of the query type, which greatly limits the number of chunks that need to be accessed. This also induces a side effect on the *stripe size*: the storage system now can use a bigger *stripe size* that can fit an entire sub-domain, thus increasing the performance.

Array versioning

Data versioning is a feature that is increasingly needed by users of scientific applications, because of several reasons.

First, it provides a convenient tool to explore the history of changes to a dataset and at the same time it avoids unnecessary data duplication by saving incremental differences only. This is a scenario frequently encountered during the simulation of scientific phenomena, where the result of each iteration usually corresponds to the changes in time observed during the simulation. In such scenarios, changes are often localized and affect only small parts of the data, making a full output of the state at each iteration both inefficient and difficult to track.

Second, versioning facilitates sharing of data sets among multiple users. For example, consider a globally shared data set that is used as input for a series of experiments, each of which is performed by a different scientist that is interested to manipulate the data in a different fashion. In such a scenario, users want their own view of the data set. A simple solution to this problem is to create a full copy of the data set for each user. However, such an approach is expensive both in terms of performance and storage utilization. Using versioning, the illusion of a dedicated copy can be easily created for each user, while internally optimizing performance and resource utilization.

Finally, versioning is a key tool that enables keeping track of data provenance [100]. This is becoming a critical issue in scientific communities, as external data sources are increasingly relied upon as an input to scientific applications. In this context, it is important to be able to track the whole I/O workflow that produced a data set through all transformations, analyzes, and interpretations, which enables better management, sharing and reuse of data in a reliable fashion.

Targeting versioning as a first-class feature, the core of our approach is built on the idea of representing data updates using immutable data and metadata. Whenever a multi-dimensional array needs to be updated, the affected cells are never overwritten, but rather a new snapshot of the whole array is created, into which the update is applied. In order to offer the illusion of fully-independent arrays with minimal overhead in both storage space utilization and performance, we rely on differential updates: only the modified cells are stored in each new snapshot. Any unmodified data or metadata is shared between the new snapshot and the previous ones. The metadata of the new snapshot is generated in such way that it seamlessly interleaves with the metadata of the previous snapshots to create incremental snapshots that look and act as independent arrays (at application level).

Lock-free, distributed chunk indexing

Data is striped and stored in a distributed fashion, which implies that additional metadata is necessary to describe the composition of arrays in terms of chunks and where these chunks can be found. Since our system is designed to manipulate a large number of arrays and each array can reach huge size, scalable metadata management becomes an critical challenge.

The problem of building spatial indexes has been studied extensively, with several specialized data structures proposed: R-trees, xd-trees, quad-trees, etc. Most of those structures were originally designed and later optimized for centralized management.

However, a centralized metadata-management scheme limits scalability as in distributed file systems. Even in the situation when enough storage is available to store metadata, the metadata server can quickly become a bottleneck when attempting to serve a large number of clients simultaneously. As discussed in Chapter 3, a distributed metadata-management scheme offers the possibility to scale better than a centralized approach. Metadata I/O overhead can be distributed among metadata servers, that results in a reduced pressure on each metadata server. With respect to metadata availability, a distributed metadata management scheme also implies better fault-tolerance: the failure of any metadata server impacts negatively only on a partial part of the whole metadata-management system.

Regarding the aforementioned arguments, it is important to implement a distributed metadata-management scheme. To this end, we propose a distributed quad-tree like structure that is used to index the chunk layout and is specifically optimized for concurrent updates. Our scheme takes advantage of the fact that data and metadata remains immutable in order to handle concurrent metadata updates efficiently without locking.

7.3.2 System architecture

Based on the design principles presented in Section 7.3.1, we designed *Pyramid*, a complete storage solution for multi-dimensional data. It is a distributed system consisting of the following components:

Version managers. Version managers are the core of Pyramid. They coordinate the process of assigning new snapshot versions for concurrent writes such that total ordering is guaranteed. At the same time, they wait for the moment when snapshots are consistent and expose them to the readers in an atomic fashion. Pyramid can be configured to use multiple version managers that collaborate to achieve the aforementioned objectives in a distributed fashion. This design favors scalability and fault tolerance over centralized approaches. Version managers are organized in a distributed hash table (DHT) to balance the workload.

Metadata managers. They implement the distributed quad-trees introduced in the previous section. They are responsible for instructing the clients what chunks to fetch from what location for any given subdomain. Metadata managers are organized in a dedicated distributed hash table (DHT) for load-balancing.

Storage servers. These service components physically store chunks generated by creating new arrays or updating existing arrays.

A storage manager. That is in charge of monitoring all available storage servers and scheduling the placement of newly created chunks, based on the monitoring information.

Figure 7.1 displays the architecture of Pyramid. As we can observe, Pyramid shares the same architecture with BlobSeer. The main difference comes from the way that Pyramid considers data as multi-dimensional structures. In contrast with BlobSeer, it reorganizes the metadata-management scheme to host quad-tree like data structures rather than segment tree ones.

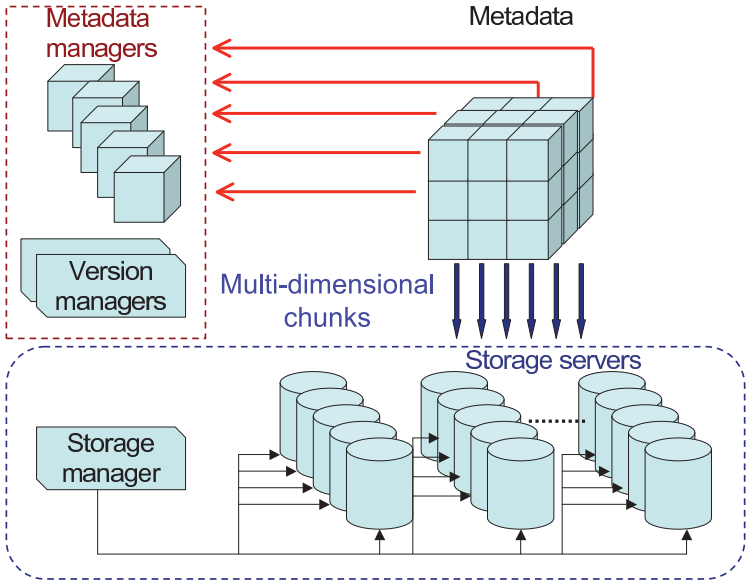


Figure 7.1: The architecture of Pyramid.

7.4 Implementation

7.4.1 Versioning array-oriented access interface

We propose an interface to handle multi-dimensional data that is specifically designed to enable fine-grained versioning access to subdomains, while offering the aforementioned features.

```
id = CREATE(n, sizes[], defval)
```

This function creates a n-dimensional array identified by `id` and spanning `sizes[i]` cells in each dimension $0 \leq i < n$. By convention, the initial snapshot associated to the array has version number 0 and all cells are filled with the default initial value `defval`. This is a lazy initialization: no data and metadata is added until some cells of the array are actually updated.

```
READ(id, v, offsets[], sizes[], buffer)
```

This primitive reads a subdomain from snapshot v of the array id. The subdomain is delimited by `offsets[i]` and spans `sizes[i]` cells in each dimension $0 \leq i < n$. The contents of the subdomain is stored in the local memory region `buffer`.

```
w = WRITE(id, offsets[], sizes[], buffer)
```

This primitive writes the content of the local memory region `buffer` to the cells of the subdomain delimited by `offsets[i]` and `sizes[i]` in each dimension $0 \leq i < n$ of the array id. The result is a new snapshot whose version number is w.

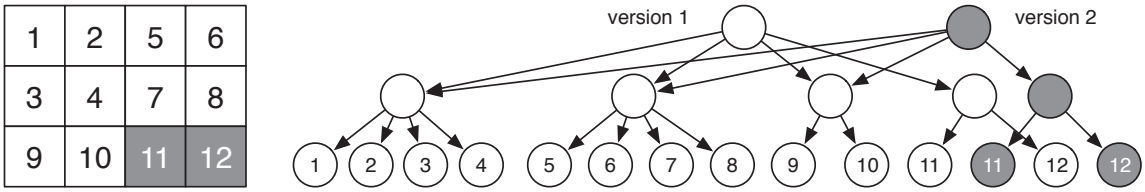


Figure 7.2: Metadata quad-trees by example: two chunks (dark color) of an initial array (partitioned according to the left figure) are updated, leading to the additional tree nodes and their links represented in the right figure.

7.4.2 Zoom on chunk indexing

In Section 7.3.1 we argued in favor of a distributed chunk indexing scheme that leverages versioning to avoid potentially expensive locking under concurrency. In this section we briefly describe how to achieve this objective by introducing a distributed tree structure that is specifically designed to take advantage of the fact that data and metadata remains immutable.

Our solution generalizes the metadata management proposed in BlobSeer [44], which relies on the same principle to achieve high metadata scalability under concurrency. For simplicity, we illustrate our approach for a two-dimensional array in the rest of this section, a case that corresponds to a quad-tree (i.e., each inner node has four children). The same approach can be easily generalized for an arbitrary number of dimensions.

Structure of the distributed quad-tree. We make use of a partitioning scheme that recursively splits the initial two-dimensional array into four subdomains, each corresponding to one of the four quadrants: Upper-Left (UL), Upper-Right (UR), Bottom-Left (BL), Bottom-Right (BR). This process continues until a subdomain size is reached that is small enough to justify storing the entire subdomain as a single chunk. To each subdomain obtained in this fashion, we associate a tree node (said to "cover" the subdomain) as follows: the leaves cover single chunks, i.e., they hold information about what storage servers store the chunks; the inner nodes have four children and cover the subdomain formed by the quadrants UL, UR, BL, BR; the root covers the whole array.

All tree nodes are labeled with a version number (initially 0) that corresponds to the snapshot to which they belong. Updates to the array generate new snapshots with increasing version numbers. Inner nodes may have as children nodes that are labeled with a smaller version number, which effectively enables sharing of unmodified data and their whole corresponding sub-trees between snapshots. Figure 7.2 illustrates this for an initial version of the array to which an update is applied.

Since the tree nodes are immutable, they are uniquely identified by their version number and the subdomain they cover. Based on this fact, we store the resulting tree nodes persistently in a distributed fashion, using a Distributed Hash Table (DHT) maintained by the metadata managers. For each tree node, a corresponding key-value pair is generated and added to the DHT. Thanks to the DHT, accesses to the quad-tree are distributed under concurrency, which eliminates metadata bottlenecks present in centralized approaches.

Reading. A read query descends into the quad tree in a top-down fashion: starting from the root, it recursively visits all quadrants that cover the requested subdomain of the read query until all involved chunks are discovered. Once this step has completed, the chunks are fetched from the corresponding storage servers and brought locally. Note that the tree nodes remain immutable, which enables reads to proceed in parallel with writes, without the need to synchronize quad tree accesses.

Writing. A write query first sends the chunks to the corresponding storage servers, and then builds the quad-tree associated to the new snapshot of the array. This is a bottom-up process: first the new leaves are added in the DHT, followed by the inner nodes up to the root. For each inner node, the four children are needed: some may belong to earlier snapshots. Under a concurrent write access pattern, this scheme apparently implies a synchronization of the quad-tree generation, because of inter-dependencies between tree nodes. However, we avoid such a synchronization by feeding additional information about the other concurrent writers during the quad-tree generation. This enables each writer to "predict" what tree nodes will be generated by the other writers and use those tree nodes as children if necessary, under the assumption that the missing children will be eventually added to the DHT by the other writers. This mechanism is similar to the one in BlobSeer but for n-dimensional case.

7.4.3 Consistency semantics

Since readers always access snapshots explicitly specified by the snapshot version, they are completely separated from the writers. Writers do not access any explicitly specified snapshot but can be thought of accessing an implicit virtual snapshot, which intuitively represents the most recent view of the multi-dimensional domain. Concurrent writes are guaranteed to be atomic and totally ordered from the user point of view. This guarantee is enforced by the version managers, responsible to delay the publication of the new snapshots until the moment when all metadata is consistent. Then readers can safely access the new snapshots.

An example of how this works is depicted in Figure 7.3. Client 2 finishes writing data faster than client 1 and thus generates a snapshot that precedes the snapshot of client 1. However, client 2 is slower at writing the metadata. Thus, the metadata of client 1 has dependencies on client 2 and neither of them can be safely accessed before client 2 finishes writing the metadata. When this happens, the version managers publish both the snapshot of client 1 and the snapshot of client 2 in an atomic fashion, effectively enabling the readers to access both snapshots.

7.5 Evaluation on Grid5000

We evaluated Pyramid through a set of experiments on the Grid'5000 [88] testbed. We aimed to evaluate both the performance and the scalability of our approach under concurrent accesses. To this end, we simulated a common access pattern exhibited by scientific applications: 2D array dicing. This access pattern involves a large number of processes that read and write distinct parts of the same large array in parallel.

We focus on two settings: a weak scalability setting and a strong scalability setting. In the weak scalability setting, we keep the size of the subdomain that each client process accesses

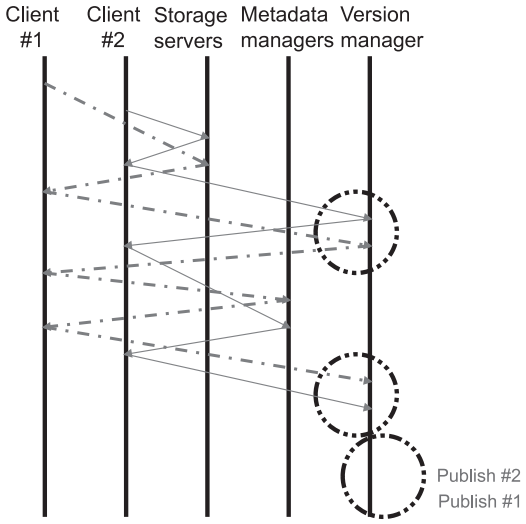


Figure 7.3: Total ordering of two concurrent updates: snapshots are published in the order in which the data was written in order to guarantee metadata consistency.

constant, while increasing the number of concurrent clients. In the strong scalability setting, we keep the total size of array constant while increasing the number of concurrent processes that access increasingly smaller parts of the array.

Each setting uses at most 140 nodes of the Graphene cluster of the Grid'5000. We dedicate 64 nodes to deploy our client processes while the rest of the nodes are used to deploy Pyramid in the following configuration: 1 version manager, 1 storage manager. We co-deployed on the 74 remaining nodes a metadata manager together with a storage server. We then compare Pyramid to the case when a standard parallel file system is used to store the whole array as a single sequence of bytes in a file. To this end, we deployed an instance of PVFS2 [45] on the same 76 nodes used to evaluate Pyramid.

Weak Scalability. In this setting, each process reads and writes a 1 GB large subdomain that consists of 32×32 chunks (i.e. 1024×1024 bytes for each chunk). We start with an array that holds a single such subdomain (i.e. it corresponds to a single process) and gradually increase its size to 2×2, 3×3, ... 7×7 subdomains (which corresponds to 4, 9, ... 49 parallel processes).

Results are shown in Figure 7.4. As can be observed, the throughput reaches 80 MB/s for one single client, demonstrating high performance even for fine granularity decompositions. Furthermore, on increasing number of concurrent clients, the aggregated throughput steadily grows up to 2.1 GB/s, which amounts to an increase of about 100% over PVFS. This demonstrates a much better scalability for our approach, which is a consequence of both the multi-dimensional-aware data striping and the distributed metadata management. On the other hand, the scalability of PVFS2 suffers because the array is represented as a single sequence of bytes, which leads to fine-grain, strided access patterns.

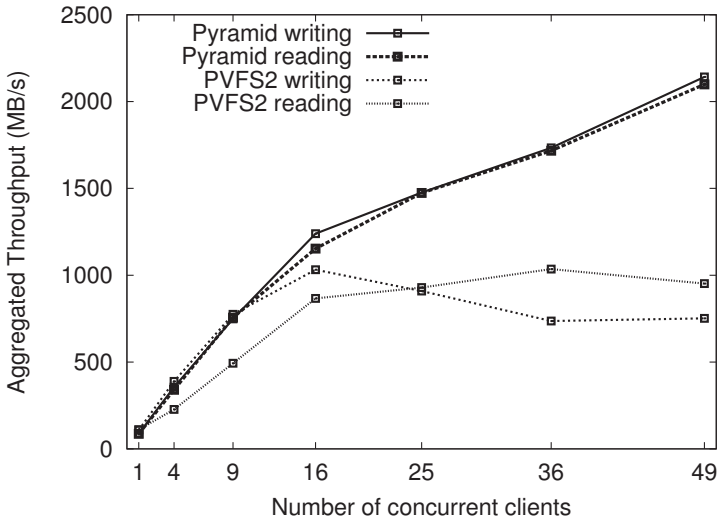


Figure 7.4: Weak scalability: fixed subdomain size, increasing number of client processes.

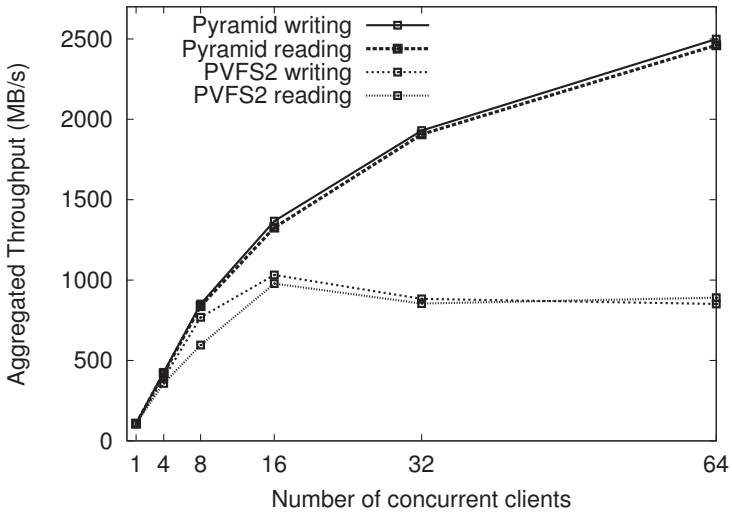


Figure 7.5: Strong scalability: fixed total domain size, increasing number of client processes.

Strong scalability In this setting, the domain size is fixed at 64 GB (1024×1024 chunks). We start with one single process and gradually increase the number of processes up to 64 concurrent processes in a layout of 2×2, 2×4, 4×4, 4×8, 8×8. In this case, the size of the subdomain for each client depends on the number of concurrent processes.

As can be observed in Figure 7.5, our approach demonstrates much higher strong scalability than PVFS for both read and write workloads. More specifically, both Pyramid and PVFS sustain a similar throughput for 1 process, both for read and write workloads. However, when increasing the number of concurrent processes, unlike for our approach, the curves corresponding to PVFS rapidly flatten. In the extreme case of 64 concurrent processes, Pyramid is able to sustain a total aggregated throughput of more than 2.5 GB/s, which represents an increase of over 150 % over PVFS.

7.6 Summary

We introduced Pyramid, an array-oriented storage system that offers support for *versioning*. Through striping techniques specifically optimized for multi-dimensional arrays combined with a distributed metadata-management scheme, our approach addresses the I/O requirements of scalable I/O parallel processing and avoids the I/O bottlenecks observed with centralized approaches.

Our evaluation shows promising results: our prototype demonstrates good performance and scalability under concurrency, both for read and write workloads. In terms of weak scalability of aggregated throughput, Pyramid outperforms PVFS by 100 %. As regards strong scalability, the gain over PVFS reaches 150 % for 64 concurrent processes, for a total aggregated throughput of more than 2.5 GB/s. As in BlobSeer and its extensions, Pyramid currently cannot deliver linear scalability due to the way it calculate border nodes for each particular access. We will investigate in finding an appropriate solution to speed up the calculation in the near future.

In the next chapter, we will present our work on designing scalable storage systems in geographically distributed environments. Our work addresses a crucial requirement to extend BlobSeer to take into account the latency hierarchy in an environment where multiple geographically distributed sites are interconnected through high-latency WAN networks.

Part III

Scalable geographically distributed storage systems

Chapter 8

Adapting BlobSeer to WAN scale: a case for a distributed metadata system

Contents

This chapter presents a part of our on-going collaboration with a research team lead by Osamu Tatebe at University of Tsukuba, in the context of the ANR-JST FP3C project [101].

OUR contributions presented in Chapter 6 and Chapter 7 feature scalable storage systems for data-intensive HPC applications. First, we proposed a novel versioning-based scheme to address the need for atomic, non-contiguous I/O support at file-system level. Second, we proposed Pyramid, a large-scale, array-oriented storage system optimized for parallel array processing. Pyramid revisits the physical organization of data

in distributed storage systems and proposes to leverage multidimension-aware data chunking in order to achieve scalable performance.

Both our first and second contributions share the focus on HPC environments. In this chapter, we address the challenge of building scalable storage systems, which target *Big Volume* in geographically distributed environments. Unlike HPC environments, geographically distributed environments usually exhibit high-latency along the connections between geographically distributed sites. We consider BlobSeer, a distributed, versioning-oriented data-management service, and propose BlobSeer-WAN, an extension of BlobSeer optimized for such geographically distributed environments. BlobSeer-WAN takes into account the high-latency feature of WAN and strives at reducing metadata accesses across geographically remote sites. It features asynchronous metadata replication and a vector-clock implementation for collision resolution.

8.1 Motivation

More and more applications in many areas (nuclear physics, health, cosmology, etc.) generate larger and larger volumes of data that are geographically distributed. Appropriate mechanisms for storing and accessing data at a global scale thus become increasingly necessary. Grid file systems (such as LegionFS [102], Gfarm [59], etc.) prove their utility in this context, as they provide a means to federate a large number of large-scale distributed storage resources. Thereby, they achieve a large storage capacity and good persistence through file-based storage.

Beyond these properties, grid file systems have the important advantage of offering a transparent access to data through the abstraction of a shared file namespace, in contrast to explicit data transfer schemes (e.g., GridFTP-based [103], IBP [104]) currently used on some production grids. Transparent access greatly simplifies data management by applications, which no longer need explicitly to locate and transfer data across various sites, as data can be accessed the same way from anywhere, using globally shared identifiers. However, implementing transparent access at a global scale naturally leads to a number of challenges related to scalability and performance, as the file system is put under pressure by a very large number of concurrent, largely distributed accesses.

Recent research [39] emphasizes a clear move currently in progress in storage architectures from a block-based interface to a object-based interface. The goal is to enable high-scalable, self-managed storage networks by moving low-level functionalities such as space management to storage servers, accessed through a standard object interface. This move has a direct impact on the design of today's distributed file systems: object-based file system would then store data as objects rather than as unstructured data blocks. According to Factor et al. [105], this move may eliminate nearly 90 % of management workload which was the major obstacle limiting file-system scalability and performance at a large scale.

Most object-based file systems exhibit a decoupled architecture that generally consists of two layers: a low-level object management service, and a high-level file-system metadata management. Therefore, we decided to explore how this two-layer approach could be used in order to build a scalable, object-based grid file system for applications that need to manipulate huge data, distributed and concurrently accessed in geographically distributed environment, especially over a wide area network (WAN). We investigated this approach

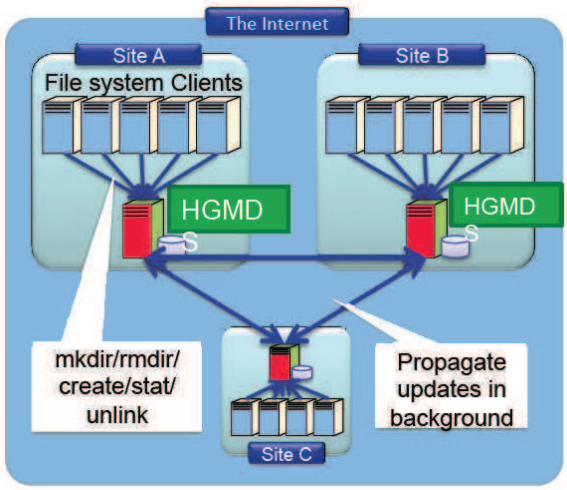


Figure 8.1: HGMDS architecture.

by designing an integrated architecture for a storage system relying on HGMDS [106], a distributed metadata-management system for global file systems developed at the University of Tsukuba, and BlobSeer [44], a large-scale BLOB management service currently being developed at KerData, INRIA Rennes. Thus, the joint architecture delegates file metadata management to HGMDS and data management to BlobSeer.

The HGMDS/BlobSeer integration targets at a large-scale distributed environment where multiple sites in different administrative domains interconnect with each other to form a global network. Since BlobSeer was initially designed to run on cluster environments, the integration with HGMDS requires extending BlobSeer for geographically distributed environments. The main idea is to favor I/O operations locally within each site to mitigate the high latency of WAN interconnections. We thus introduce an asynchronous metadata replication scheme across sites, and use vector clocks to detect and resolve collisions. This extended version, called BlobSeer-WAN, shares the same *eventual consistency model* as HGMDS, which allows them to be easily coupled.

8.2 State of the art: HGMDS and BlobSeer

8.2.1 HGMDS: a distributed metadata-management system for global file systems

HGMDS [106] is a recent project developed at the university of Tsukuba as a metadata-management system for the next version of the Gfarm file system [59]. HGMDS relies on multiple metadata servers (MDSs) to manage the file system namespace. Figure 8.1 illustrates a typical deployment of HGMDS over multi-geographical sites interconnected by a WAN: one MDS is deploy on each Grid site, handling metadata operations issued by file system clients located on the site. Therefore, HGMDS reduces remote metadata operations over WAN interconnections, minimizing the effect of the long network latency. To guarantee the global namespace consistency, MDSs are synchronized asynchronously.

Architecturally, HGMDS organizes metadata that are inode data structures as a key-value store. Each unique inode number (key) is mapped to an inode structure (value) which contains all the information about a file system object such as file, directory, etc. Since some metadata operations such as "mkdir" and "rm" affect multiple inodes, HGMDS ensures those multi-key transactions to be performed in an atomic fashion. It implements metadata versioning and uses vector clocks for collision detection.

8.2.2 BlobSeer in WAN context

As presented in detail in Chapter 4, BlobSeer addresses the problem of storing and efficiently accessing very large, unstructured data objects. It focuses on heavy access concurrency where data is huge, mutable and potentially accessed by a very large number of concurrent processes. BlobSeer introduces a versioning scheme which not only allows clients to roll back data changes when desired, but also enables access to multiple versions of the same BLOB within the same computation.

Although BlobSeer is proved to be a scalable data-management service under massive concurrency, it is more suitable for a cluster environment whose machines are interconnected by a low-latency network. As for read/write operations, a metadata I/O phase is needed that involves transferring metadata from the client to metadata servers. In WAN environments, the completion time of these operations are costly, as many sockets are opened just to transfer small pieces of metadata.

In order to deploy BlobSeer efficiently on geographically distributed environments, we need to redesign it to take into account the latency hierarchy. Indeed, intra-cluster latency may be hundreds to thousands times lower than inter-cluster latency. We developed BlobSeer-WAN as an extension of BlobSeer for geographically distributed environment. BlobSeer-WAN features distributed version managers and a synchronization scheme to maintain BLOB consistency while keeping as good performance as the original BlobSeer.

8.3 BlobSeer-WAN architecture

BlobSeer-WAN is designed for large-scale distributed environments, where multiple geographically distributed sites interconnect with each other to form a global computing environment. Therefore, it is vital that our design is consistent with this setting. As shown on Figure 8.2, BlobSeer-WAN differs from BlobSeer in its deployment setting. BlobSeer-WAN components are organized in site-oriented groups.

Multiple version managers. BlobSeer-WAN has multiple *version managers* rather than only one as in the original BlobSeer. Each *version manager* is deployed on one site and is dedicated to register update requests (APPEND and WRITE) issued by the clients of this site. At the global level, the *version managers* coordinate together asynchronously.

Metadata providers. On each site, a number of *metadata providers* is deployed to store metadata, which allows for finding all the chunks of a specific BLOB. Clients of the site only access the *metadata providers* on its site to avoid high latency connections. To replicate metadata across sites, *metadata providers* perform synchronization asynchronously.

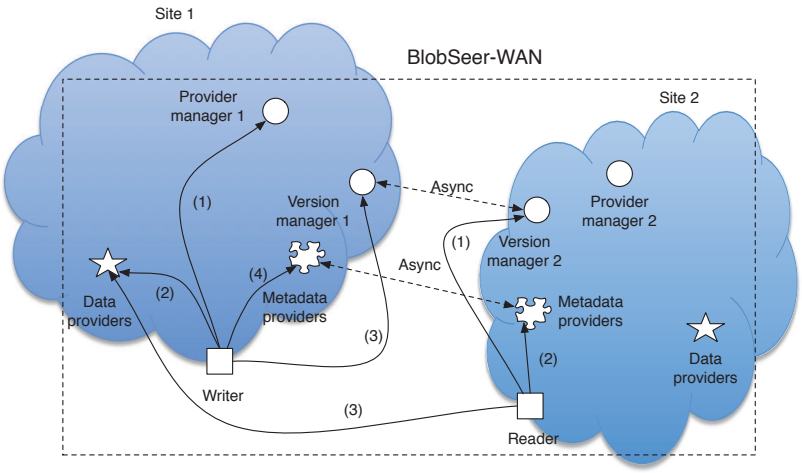


Figure 8.2: BlobSeer-WAN architecture.

Multiple provider managers. Each *provider manager* keeps information about the available *data providers* deployed on its local site. They are responsible for scheduling the placement of newly created chunks on *data providers* under their control.

Data providers. BlobSeer-WAN organizes *data providers* in different groups, each group is under the management of the *provider manager* dedicated to the local site.

The design goal of BlobSeer-WAN is to localize data and metadata operations on each site as much as possible. By implementing multiple version managers and organizing the system components in site-oriented groups, clients can perform most metadata I/O and data I/O operations locally within their own site, avoiding high latency in sites interconnections. From the point of view of a particular client, BlobSeer-WAN components on its site are called as *local version manger*, *local metadata providers*, *local provider manager*, and *local data providers*. Other BlobSeer-WAN components on the other sites are considered as *remote version manager*, *remote metadata servers*, *remote provider manager*, and *remote data providers* respectively.

Writing

For a `WRITE` request (Figure 8.2), the client contacts the *local provider manager* to obtain a list of *data providers*, one for each chunk of the BLOB that needs to be written. Since the *local provider manager* only manages providers on its site, the client always receives a list of *local data providers*. Therefore, that data I/O will be carried out in parallel within the local site. When the client finishes the data I/O phase, it contacts the *local version manager* to request a new version for the BLOB. This version is then used by the client to generate the corresponding new metadata, then to send them to *local metadata servers*.

From the perspective of a client, there is virtually no difference between a `WRITE` operation in BlobSeer-WAN and in BlobSeer. The client sees BlobSeer-WAN components on its site as if they made an isolated BlobSeer. It is the *local version manager* that propagates updates to *version managers* on other sites. Metadata is also replicated to other sites asynchronously by *local metadata servers* when a new update arrives from a client.

Reading

To READ data (Figure 8.2), the client first contacts its *local version manager*: it needs to provide a BLOB id, a specific version of that BLOB, and a range, specified by an offset and a size. If the specified version exists, then the client will traverse downward the metadata tree that was built for that version. It does so by sending requests to its *local metadata providers*. If any *metadata provider* cannot find a requested tree node in its local storage, it means that metadata piece has not been replicated to the local site, yet. In this situation, the *metadata provider* will have to pull that metadata piece from other sites of the BlobSeer-WAN deployment. Fortunately, this process is transparent to the client. Finally, after reaching the appropriate metadata leaves, the clients will know which *data providers* store the corresponding chunks. Note that BlobSeer-WAN does not replicate chunks, so that accessing *data providers* may involve contacting remote sites.

8.4 Implementation

Following the proposed design, we cloned BlobSeer to create BlobSeer-WAN: a variant of BlobSeer specialized for WAN environment. BlobSeer-WAN features an optimistic replication scheme, distributed multi-version managers, and an asynchronous synchronization scheme based on vector clocks.

8.4.1 Optimistic metadata replication

One important design principle of BlobSeer is to keep data and metadata "immutable". Upon each update, new metadata are generated and "weaved" with the metadata of previous versions in order to reflect updates in consistent snapshots. Obviously, this design principle facilitates our design regarding metadata replication over multiple distributed sites. There is no need to care about collision detection and resolution, because no update ever occur on metadata.

BlobSeer-WAN implements an optimistic metadata replication scheme among *metadata providers* of different sites. The idea of the optimistic replication is to not wait till the whole replication process is done. It aims at being transparent from the clients point of view. Whenever a *local metadata provider* receives new metadata, it forwards them asynchronously to the appropriate *metadata providers* on other sites. Thanks to asynchrony, replication is kept transparent to clients, thus adding very little overhead to the overall performance of metadata I/O in WRITE operations. Nevertheless, as replication is done in an asynchronous fashion, we can not guarantee that the *metadata providers* on other sites will immediately receive replicated metadata. In the situation where a *metadata provider* does not own a metadata piece upon receiving the reading request from a reader, that *metadata provider* will have to query the other sites to locate the missing metadata piece, and to pull it proactively.

Each site organizes its *metadata providers* in an independent, dedicated DHT ring. Thus, the replication operations are mere DHT PUT requests. To avoid re-replicating metadata in an infinite loop among *metadata providers*, there should be a mechanism to differentiate metadata sent by clients from metadata replicated by *metadata providers*. To this end, sender information in each DHT PUT request will be examined. If the metadata came from a client

process, it is forwarded. Otherwise, the *metadata provider* only need to save that metadata piece into its storage space.

8.4.2 Multiple version managers using vector clocks

BlobSeer-WAN features multiple distributed *version managers* to avoid clients contacting a *version manager* over a high-latency network. Each *version manager* is dedicated to handling all version-related requests coming from clients on its local site. Beside the functionalities of the *version manager* in BlobSeer, the *version managers* in BlobSeer-WAN need to synchronize with others to keep each BLOB in a consistent state (a BLOB is consistent if each version of the BLOB contains the same data contents across the whole system).

Assume that a particular BLOB is concurrently updated by multiple clients on different sites. Those clients will contact different *version managers* in order to get new version identities for their updates. When a *version manager* receives a request from a client, it issues a new version identity for the requested BLOB and then asynchronously inform the other *version managers*. In this situation, each *version manager* will soon come to the point that it receives multiple new version identities for a specific BLOB and it has to decide which is the order of these new versions. The common order must be agreed by all the *version mangers* in order to keep the BLOB in a consistent state across the system

To be able to order different updates in such a distributed environment, BlobSeer-WAN uses a variant of vector clocks [107] implementation. In BlobSeer-WAN, a vector clock is a list of local counters in which each counter represents the local counter of the corresponding version manager. Thus, each version of a BLOB is associated with a vector clock rather than an integer in the original BlobSeer.

To determine whether two versions of the BLOB are in conflict or have a causal ordering, the *version manager* has to compare the two vector clocks which are associated with the two version identities. A vector clock is an ancestor of another vector clock if all the local counters of the former vector clock are less than the corresponding counters of the latter one, and at least one of the local counters is strictly smaller. Otherwise, the two vector clocks are considered to be concurrent and in conflict. In the first case where the order of the two vector clocks are determined, BlobSeer-WAN knows which version happened before the other one. In the other case, the two are in conflict.

For each update to a specific BLOB, a client has to request its local *version manager* for a new version identity. Upon receiving the request, the *version manager* creates a new version identity by incrementing its own counter in the vector clock of that BLOB. The new version identity is then sent back to the client and is asynchronously propagated to the other *version managers*. Each time a *version manager* receives a new version identity of a BLOB, it updates all the counters in the vector clock for that BLOB by taking the maximum of the value in its own vector clock and in the new version identity it received.

Furthermore, the *version manager* also has to compare the latest version of the BLOB to the received version. If the two versions are not conflict and the received version is newer in the sense of vector clock comparison, then the *version manager* promotes it to be the latest version. If the two versions are in conflict, the received version is marked as a pending version, which means that the version is not visible to clients yet. The *version manager* needs to rely on subsequent updates in order to execute a reconciliation scheme that is able to

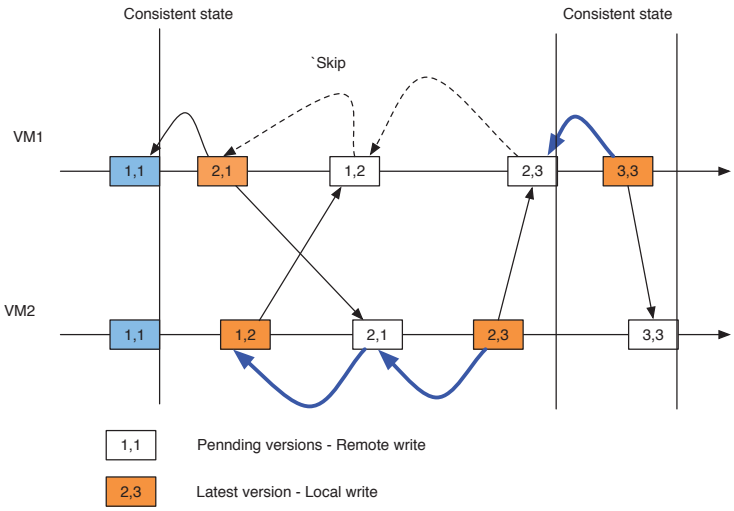


Figure 8.3: Vector clocks and optimistic metadata replication.

merge the update in the conflict version to the next versions of the BLOB.

To illustrate how vector clocks work in BlobSeer-WAN, let us consider a small example as shown in Figure 8.3. We assume that BlobSeer-WAN is deployed on two sites: VM1 and VM2 are the *version manager* on the first site, and on the second site, respectively. At the beginning, the latest version of a specific BLOB is associated to version (1,1) by both VM1 and VM2. Thus, the BLOB is in a consistent state. Now, two clients on the respective sites simultaneously perform updates to the BLOB. On the first site, VM1 increases its local counter for the BLOB and assigns version (2,1) to the new update. On the second site, VM2 follows the same procedure and assigned version (1,2) to reflect the other new update. As VM1 and VM2 are synchronized asynchronously, they are eventually aware of both two versions (1,2) and (2,1). Because version (1,2) and version (2,1) are in conflict, neither VM1 nor VM2 can immediately decide upon the order of the updates. In other words, no consistent state which merges the two updates to the base version (1,1) is determined.

In this conflict situation, BlobSeer-WAN relies on the subsequent updates to decide upon the causal order. In our example, VM2 gets a subsequent update request to the BLOB, creating a new version (2,3). Since (2,3) should contain the updates made by both (1,2) and (2,1), VM2 arbitrarily decides the latest state of the BLOB is a result of applying sequentially updates reflected in version (1,2), version (2,1), and version (2,3) to the based version (1,1), in that order. Upon synchronization, VM1 will be eventually informed of the new version (2,3). Because that version is greater than any version of the BLOB seen at the moment, VM1 promotes version (2,3) to be the BLOB latest version and also publishes version (1,2). As this state, the BLOB is in the consistent state and is the same on both two sites. With respect to the consistent state in version (1,1), the versions (1,2) and (2,1) are in conflict and they do not contain updates made by each other. With respect to the consistent state in version (2,3), the ordering of (1,2) and (2,1) is determined.

It may happen the case that a new version (3,2) is created by VM1 before the VM1 gets to know the version (2,3) created by VM2. In this case, VM1 also has to decide the order of two versions (1,2) and (2,1) with respect to the version (3,2). Therefore, the latest state of

the BLOB seen by VM1 is a result of applying sequentially updates reflected in version (2,1), version (1,2), and version (3,2) to the based version (1,1), in that order. As (3,2) and (2,3) are still in conflict, both VM1 and VM2 will have to wait again for the subsequent updates in order to reach to a consistent state of the BLOB.

8.5 Evaluation on Grid'5000

To evaluate BlobSeer-WAN, we conducted several preliminary experiments on the Grid'5000 testbed. Our experiments were carried on at most 60 nodes of the Nancy site and 40 nodes of the Grenoble site. Regarding the network connections, Intra-cluster measured bandwidth is 117.5 MB/s for TCP sockets with MTU set at 1500 B. Nancy and Grenoble are linked through the Grid'5000 backbone, featuring a 10 Gb/s fiber channel.

BlobSeer-WAN vs. BlobSeer in local site accesses

In the first series of experiments, we aim at assessing the overhead of the multiple version managers implementation and the metadata replication scheme in BlobSeer-WAN. To this end, we compared the performance of BlobSeer-WAN to the original BlobSeer in the setting where BlobSeer is deployed on a cluster environment. Specifically, we used 40 nodes of Nancy cluster to deploy BlobSeer: 1 version manager, 1 provider manager, 10 metadata providers and 28 data providers. On the same nodes, we deployed BlobSeer-WAN daemons that share similar functionalities with those of BlobSeer. We used another 40 nodes of Grenoble cluster to deploy the rest of BlobSeer-WAN daemons for Grenoble site. The number of each type of daemons is the same as for Nancy cluster, and thus the same as for the original BlobSeer instance.

We reserved 20 nodes of Nancy cluster to deploy clients, each of them appends 512 MB to the same BLOB with 1 MB configured chunk size. As clients reside in Nancy cluster, they are on the same site as the BlobSeer instance. In this setting, clients only perform I/O over Nancy local networks in our experiments with BlobSeer.

We measured the aggregated throughput achieved when we progressively increase the number of concurrent clients. As shown on Figure 8.4, both BlobSeer-WAN and BlobSeer demonstrated excellent scalability. The achieved throughput on BlobSeer-WAN was similar to that of the original BlobSeer, proving that the additional overhead due to multiple version managers implementation and a replication scheme is minimal. This can be explained as version vectors and metadata are asynchronously replicated at the server side to Grenoble, thus these replication processes are transparent to the clients.

BlobSeer-WAN vs. BlobSeer in remote site accesses

In this series of experiments, we demonstrate the advantage of BlobSeer-WAN over the original BlobSeer in geographically distributed environments. We deployed BlobSeer-WAN on 80 nodes of 2 sites Nancy and Grenoble as in the first experiment. Regarding BlobSeer, we deployed one BlobSeer instance on Grenoble cluster. The BlobSeer instance consists of 40 nodes in which there are 1 version manager, 1 provider manager, 10 metadata providers and

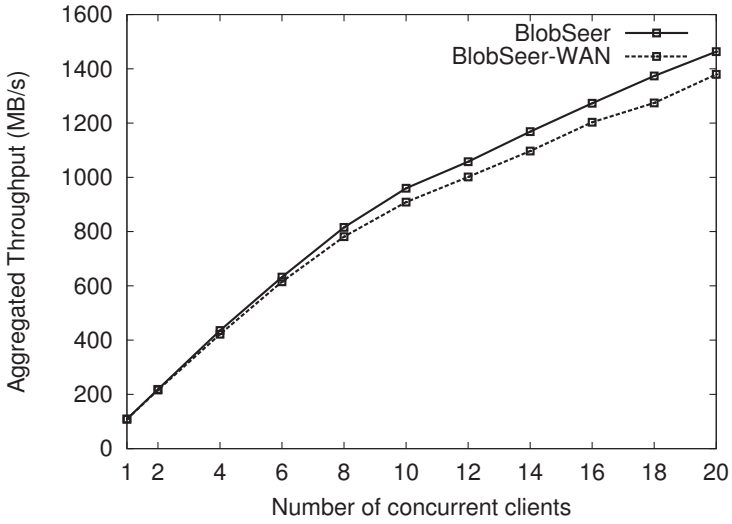


Figure 8.4: BlobSeer-WAN vs. BlobSeer in local-site accesses scenario.

28 data providers. As can be observed, we have the BlobSeer instance used the same amount of resources as a part of BlobSeer-WAN on each cluster.

To force BlobSeer clients to use the interconnection between Nancy and Grenoble, we reserved 20 nodes only in Nancy for client processes. Thus, our experiment will demonstrate the advantages of BlobSeer-WAN over the original BlobSeer in its extreme deployment where BlobSeer clients have to access BlobSeer over inter-sites connections. The results are shown on Figure 8.5. We progressively increase the number of concurrent appending clients, in this scenario, BlobSeer-WAN provides a better aggregated throughput with a gain of 15 % for 20 clients. Interestingly, remote site accesses in BlobSeer still achieve good aggregated throughput, which is made possible by the excellent 10 Gb/s interconnection of between Nancy and Grenoble.

8.6 Summary

In this chapter, we have presented BlobSeer-WAN, an extension of BlobSeer optimized for geographically-distributed environments. BlobSeer-WAN is designed to take into account the latency hierarchy exhibited in such distributed environments. It features an asynchronous metadata replication scheme and implementation of multiple version managers to hide high latency on WAN interconnections. In the BlobSeer-WAN setting, clients only contact their site-local BlobSeer-WAN servers to perform I/O and it is the role of the BlobSeer-WAN servers on different sites to synchronize asynchronously among them. To resolve collisions, we introduced an implementation of vector clocks that allows BlobSeer-WAN to determine the order of concurrent updates. After ordering, each BLOB is in a consistent state even if it is globally shared on multiple distributed sites.

Several experiments were performed on the Grid'5000 testbed demonstrating that BlobSeer-WAN can offer scalable aggregated throughput under heavy concurrency. We also compared BlobSeer-WAN to the original BlobSeer to observe the overhead of the multiple

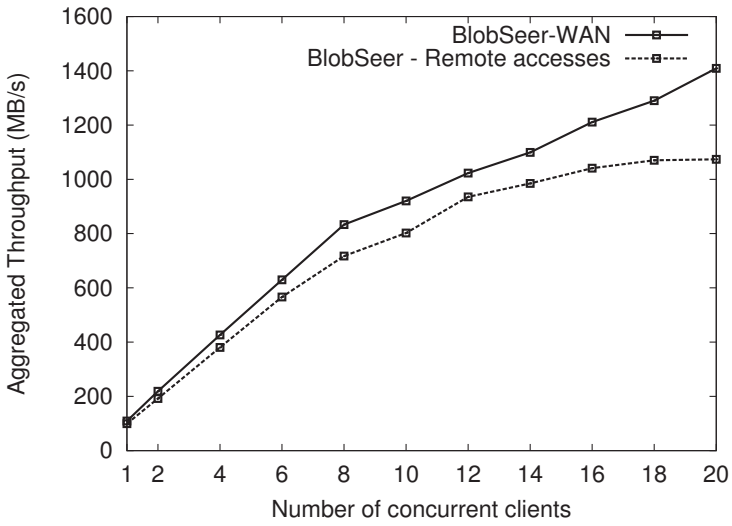


Figure 8.5: BlobSeer-WAN vs. BlobSeer in remote-site accesses scenario.

version managers implementation and the metadata replication scheme in BlobSeer-WAN. Our results shown that BlobSeer-WAN can provide a comparable throughput as of the original BlobSeer.

Part IV

Vertical scaling in document-oriented stores

Chapter 9

DStore: An in-memory document-oriented store

Contents

This work was carried out in the context of a 3-month internship under the guidance of Dushyanth Narayanan at Microsoft Research Cambridge.

In the previous chapters, we have presented our work on building scalable distributed storage systems that can *scale out (horizontally)* by adding more storage servers to the systems. In this chapter, we explore an alternative approach to build a scalable system for multi-core machines with large main memory. The proposed solution, DStore, is

designed to *scale up (vertically)* by adding more CPU cores and more memory, rather than adding more servers like in distributed systems.

DStore targets the *Big Velocity* characteristic of Big Data, where data flows in and out at high speed. DStore aims at providing fast, atomic complex transaction processing that refers to the update rate in data writing, while it also delivers high throughput read accesses for analytical purposes. In the context of *Big Variety*, DStore addresses a document-oriented data model which is optimized for popular Internet applications.

9.1 Introduction

NoSQL movement

Over the past few years, NoSQL data stores have been widely adopted by major Internet companies, such as Amazon, Facebook, and Google. NoSQL rose in response to the observation that current relational database management systems (RDBMS) are not a good fit for storing and retrieving Internet-specific data. First, RDBMS have been built along the relational model that is considered to poorly meet the requirements of the current data models found in popular Internet applications. In some cases, the relational model is too powerful for data that have no associative relationship among them. For example when what is needed is simply to access a piece of data given a key (key-value model). In other cases, data from social network services rely on graph structures withs nodes, edges and properties to represent their complex relationships: the relational model becomes insufficient in this situation. Second, most RDBMS sacrifices scalability for ACID semantics, whereas popular Internet applications don't often need ACID for their data. For instance, Facebook status updates or tweets can be stale and there is no penalty to propagate those updates later under the guarantee that the propagation is done eventually.

NoSQL data stores trade the complexity of having ACID semantics and a relational data model for higher performance and higher scalability. Among different types of NoSQL data stores, *document-oriented stores* offer an interesting compromise between the system performance and the rich functionality on a document-oriented data model. While key-value stores operate on independent key-value pairs and RDBMS are optimized for structured data having associative relationships, document-oriented stores fall in the middle: data is packed in documents, which are similar to data rows in RDBMS but they are very flexible and self-organized. Each document consists of multiple data fields, but it does not need to follow a fixed schema that defines the structure of the document. Therefore, a document-oriented store can be considered as a semi-structured database.

Compared to key-value stores, document-oriented stores can be seen as the next logical step from storing simple key-value pairs to more complex and meaningful data structures that are encapsulated in documents. Similarly to key-value stores, each document can be accessed given a globally shared document ID. Moreover, document-oriented stores provide the capacity to query a document not only based on the document ID but also based on the document content itself.

Trending towards vertically scaling by leveraging large memory

As hardware technology is subject to continuous innovation, computer hardware has become more and more powerful than the prior model. Modern servers are often equipped with large main memory capabilities, which can have sizes up to 1 TB. Given this huge memory capability, data-management systems are now able to store their data in main memory, so that hard disks become unnecessary [5]. While memory accesses are at least 100 times faster than disk, keeping data in main memory becomes an obvious trend to increase the performance of data-management systems.

In a recent study conducted by Microsoft Research [108], memory is at an inflection point where 192 GB of memory cost only $2640. Further, the decreasing cost/size ratio of memory still benefits from the Moore's law, this resulting in the possibility to double the memory size with the same cost every 18 months. In this context, vertically scaling data management systems by using bigger memory is cost-effective.

Current disk-based data-management systems only leverage main memory as a caching mechanism. One way to scale those systems vertically (scale-up) is to increase the cache size so that more data can be cached in main memory for faster accesses. However, this optimization is suboptimal, as it cannot fully exploit the potential of large memory. The scalability is limited as disk-based data-management systems have to implement complex mechanisms to keep data on cache and on disks consistent.

9.2 Goals

In the following section, we explore the potential benefits of the "scale-up" approach to the design of a scalable system. DStore aims to be able to scale up by adding more memory rather than scaling horizontally by adding more servers as in distributed data management systems. We design and implement DStore, a document-oriented store with the following goals.

Fast and atomic complex transaction support. Various document-oriented stores such as CouchDB [54] and MongoDB [56] offer only support for atomic access at the granularity of a single document. Complex transactions that access multiple documents are not guaranteed to be atomic. Because documents are distributed across different storage servers, support for atomicity of multiple document access may require distributed locks, which results in poor system performance.

Atomic complex transactions are required in various scenarios in many current applications. Consider a bank company where each user account is represented by a document in a document-oriented store. In order to complete a bank transfer, the following operations in multiple documents must be done in a single atomic query: (1) verifying if there are sufficient funds in the source account in the first document, (2) decreasing the balance of the source account, and (3) increasing the balance of the destination account in the second document. Without atomic guarantees, the database will fall in an inconsistent state. Similarly, multiple operations in a single document also need to be atomic such as conditional update, read before write, etc.

The main design goal of DStore is to support atomicity for complex transactions and to do so with low additional overheads.

High throughput for analytical processing. Analytical processing refers to read queries that may access multiple documents or the entire data store in order to get a summary report for analytic purposes. One typical example of analytical processing in DBMS is the scan query whose goal is to select any data records that satisfy a particular condition. In the situation where concurrent update and read queries may access the same pieces of data, synchronization for data consistency is unavoidable, which leads to a slow update rate and a low-throughput analytical processing.

To deliver high throughput for analytical processing without interfering with update queries, DStore must be able to isolate both types of workloads with low overhead.

Achieving these two goals, we thus claim that DStore can provide fast, atomic complex transaction processing that refers to the update rate in data writing, while it also delivers high throughput read accesses for analytical purposes. This claim will be validated through the design of DStore and some preliminary synthetic benchmarks at the end of this chapter. In the future, we plan to reenforce our claim by performing more benchmarks on real-life workloads.

9.3 System architecture

9.3.1 Design principles

In order to achieve the aforementioned goals, we design DStore with the following principles:

Single threaded execution model

To date, the search for efficient utilization of compute resource in multi-core machines triggered a new architecture for multi-threading applications. Previous application designs rely on multiple threads to perform tasks in parallel in order to fully utilize CPU resources. In reality, most of the tasks are not independent of each other, as either they access the same part of data, or some tasks need part of the results from another task. This well-known problem (concurrency control) made it nearly impossible to have full parallelization in multi-threading environments.

DStore targets complex transactions where each transaction consists of several update/read operations on multiple different documents. Let us consider the following example: Transaction T1 updates 3 documents A, B and C. Transaction T2 updates 3 documents B, C and D. Another transaction T3 has to read documents B and C and update document D. Since T1, T2 and T3 are not mutually independent, it is impossible to perform these transactions concurrently using multiple threads without synchronization with exclusive locks. Obviously, in the case of more complex transactions, there is a higher possibility that those transactions are not independent of each other.

As a result, DStore relies on a single-threaded execution model to implement only one thread (called the *master thread*) for executing all update transactions sequentially. This approach is first advocated in [64] as a result of trending in in-memory design. As long as one single thread is performing data I/O, thread-safe data structures are not necessary. In other words, the code for locking mechanisms in concurrency control can be completely removed without loosing correctness, which results in less execution overhead. A recent study in [5] showed that locking and latching mechanisms in the multi-threading model create nearly 30% overhead in data-management systems.

Parallel index generations

With its rich document-oriented data model, a document-oriented store can offer an interface that can be close to that of RDBMS. One of the nice features is the possibility to query a document not only based on the document ID but also based on the document contents itself. To enhance the query performance, both RDBMS and document-oriented stores such as CouchDB [54] and MongoDB [56] rely on indexes. Maintaining indexes allows fast lookup to desired documents, but usually creates a certain amount of overhead for updates and deletes. Particularly, if indexes are built by the same thread for update transactions (the *master thread*) in a synchronous fashion, the system performance will be reduced twice when doubling the number of indexes.

To speed up index generation and to reduce overhead on the *master thread* for update transactions, DStore assigns index generation task to dedicated background threads, called *slave threads*. Each *slave thread* manipulates one index by maintaining a data structure such as the *B+tree* data structure that we will present further in Section 9.4 . *B+tree* allows searches in $O(\log(n))$ logarithmic time. Because DStore resides in main memory, all indexes keep only pointers to the actual documents in order to minimize memory consumption. In this setting, an *in-place data modification* when updating a particular document is very expensive. Locking and synchronization among *slave threads* and with the *master thread* are needed to keep all indexes in consistent states.

To avoid such a synchronization and to keep all the indexes independent from each other (even when they share the documents), *in-place data modifications* have to be avoided. DStore keeps each document immutable. Update transactions will not modify document contents, but rather create new documents and rely on indexes to commit the changes back to DStore. This mechanism is referred to as *copy-on-write* in the data-management community.

Delta-indexing and bulk updating

Since indexes are maintained by *slave threads*, the *master thread* that executes update transactions needs only to write new documents (in case of an update or an insert) to the memory space (part of the main memory reserved for holding DStore data), and to push an index request in the waiting queues of each index. This mechanism is referred to as *differential files* or *delta-indexing* [109] (Figure 9.1). DStore names the waiting queues as *the delta buffers*. Each queued element is typically a key-value pair, where the key is the indexed document attribute and the value is the pointer to the created document.

This *delta-indexing* mechanism allows DStore to potentially sustain high update rates under the expectation that pushing an index request to the *delta buffer* is faster than updating

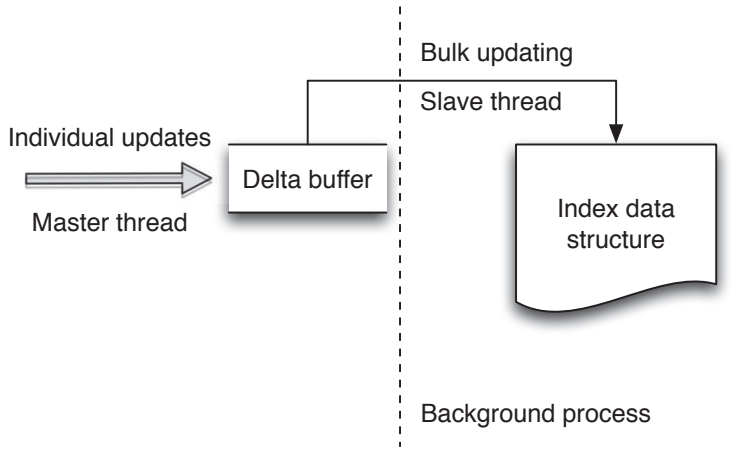


Figure 9.1: Delta indexing mechanism.

the data structure for the index itself. This expectation is obviously made possible as following: The *delta buffer* can be a simple queue that has the complexity of $O(1)$ for push operations whereas data structures for indexing such as *B+tree* need $O(\log(n))$ for every insert of lockup. Even when the *delta buffer* is a *B+tree*, the time to insert in the *delta buffer* is still lower due to its small size.

Moreover, one novel design choice we make is to leverage *bulk updating* to maintain indexes in background. This technique allows us to achieve three goals in DStore:

- First, bulk inputs in *delta buffers* are sorted before inserting into the indexes in order to better leverage cache behavior. As data is sorted, there is a high chance that inserting a new element in the *B+tree* will follow the same path from the *B+tree* root to the leaf or a partial path that was already cached in previous accesses.
- Second, merging the *delta buffer* to the index data structure in bulk avoids readers to read partial updates of a multi-key complex transaction. Thus, it guarantees transaction atomicity. DStore implements a versioning mechanism to control the moment when the new versions of indexes are revealed to readers. In our design, this happens after all updates in the *delta buffer* are merged to the previous index.
- Third, the *delta buffer* can be compacted before being processed. As a sequence of inserts and deletes on the same documents may exist in the *delta buffer*, DStore can remove obsolete updates and keep only the latest update for each particular document. Therefore, DStore potentially minimizes the number of modifications to the persistent index data structure.

Versioning for concurrency control

Even if DStore has one *master thread* to execute update transactions sequentially, it allows multiple reader threads to run analytical processing concurrently. DStore relies on versioning for concurrency control to be able to deliver high-throughput for reading while minimizing interference with the *master thread* and with the *slave threads* building indexes.

DStore uses *B+tree* as the data structure for indexes and leverages shadowing (copy-on-write) [86] to guarantee that a set of updates are isolated in a new snapshot. Before a *slave thread* updates its index, it clones the index to create a newer snapshot and applies the updates only to that snapshot. When this process is completed, the new snapshot will be revealed to readers. Since the *slave threads* and readers access different snapshots of the indexes, they can work concurrently without locking.

Moreover, the novel idea in DStore is to merge updates to each index in bulk where only one new snapshot represents all the updates in the *delta buffer*. This approach has the advantage of reducing the number of intermediate snapshots in order to reduce memory consumption. This is clearly the difference between our approach and a pure copy-on-write implementation in *B+tree* [86], in which each update requires cloning an entire path from the *B+tree* root to the corresponding leaf to create a new snapshot.

Stale read for performance

Analytical processing can accept a certain level of staleness [110]. Results from a read query can be slightly out-of-date if they are used for analytic purposes. For example, social network websites such as Facebook allow users to write messages on their wall and get updates from other users. It is acceptable for such a query for all recent updates to return stale data which contain updates performed seconds to minutes ago.

To take advantage of the above property, DStore gives users the choice to decide the freshness level of an analytic query on a per-query basis. Instead of returning up-to-date data, a *stale read* only accesses the latest snapshot of the indexes. This choice leaves *stale reads* to be executed independently in an isolated fashion, at the cost of not being able to query data in the *delta buffers*. For a *fresh read*, locking is needed before scanning each *delta buffer* to avoid threading exceptions. Of course, this will impact negatively on the update rate of the *master thread* and also on the *slave threads*.

9.3.2 A document-oriented data model

Documents are the main data abstraction in DStore. Each document consists of multiple data fields but it does not need to follow a fixed schema that defines the structure of the document. Thus, it is very flexible and self-organized. For example, here is a document holding an employee information:

```
ID: int
Name: char[30]
Address: char[60]
```

Because a document is flexible, it can contain more data fields for another employee as shown below:

```
ID: int
Name: char[30]
Address: char[60]
Passport: int
```

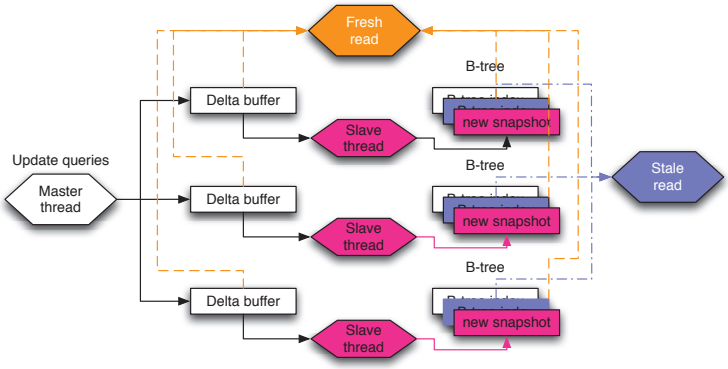


Figure 9.2: DStore service model.

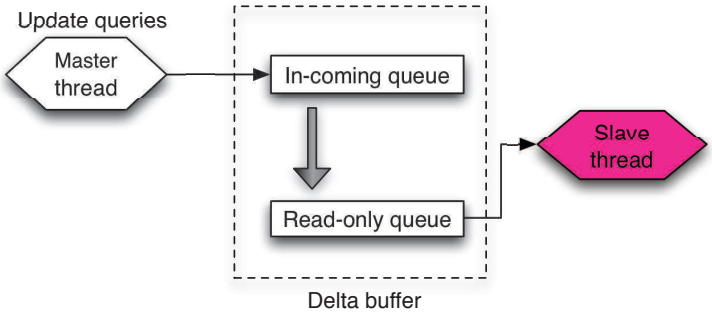


Figure 9.3: Zoom on the *delta buffer* implementation.

DStore allows users to access a document not only by the document ID, but also through several indexes. Assuming that `Name`, `Address` are globally unique, DStore can build three indexes for `ID`, `Name`, and `Address`.

9.3.3 Service model

As discussed in Section 9.3.1, DStore is built on the idea of having only one *master thread* to process update transactions sequentially but to allow concurrent analytical processing. Figure 9.2 represents the service model of DStore.

DStore executes *slave threads* in background for maintaining the *B+tree* indexes, where each *slave thread* is responsible for one index. Once the *slave thread* starts, it checks if the *delta buffer* is not empty, then it pushes all elements in the *delta buffer* to a new snapshot of its *B+tree* index. Only when finished, this snapshot is revealed for reading and the *slave thread* can then repeat the whole process for subsequent data updates.

Actually, each *delta buffer* consists of two parts: an *in-coming queue* for input coming from the *master thread* and a *readonly queue* holding updates that are under processing by the *slave thread* (Figure 9.3). When the *slave thread* starts building a new snapshot, the *in-coming queue* is flagged to be the *readonly queue* and a new *in-coming queue* is created. Basically, this mechanism is done to favor bulk processing and to minimize locking overheads on the *delta buffer*.

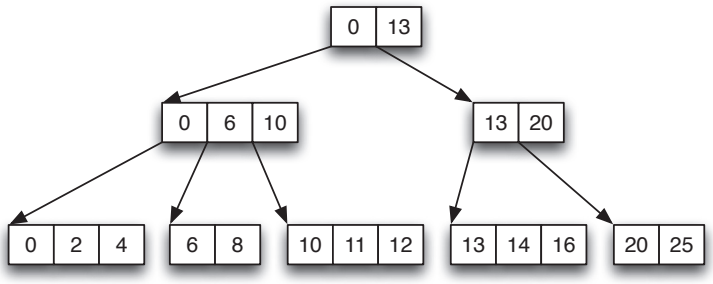


Figure 9.4: An example of a *B+tree*.

DStore allows users to configure which data structures will be used for the *delta buffers*. For instance, the *delta buffer* can be a *B+tree* that provides $O(\log(n))$ for both lookup and insertion. By default, DStore uses the vector datatype, as it allows faster insertion of $O(1)$ at the cost of slower lookup $O(n)$. This design decision favors the implementation of one *master thread* for updates in DStore.

9.4 DStore detailed design

9.4.1 Shadowing *B+tree*

DStore uses *B+tree* data structures for the indexes. The *B+tree* is a tree data structure for keeping data sorted for fast retrieval in logarithmic time. Each *B+tree* inner node contains entries that are mappings between keys and their children, while a *B+tree* leaf node contains key-value pairs. Keys in a *B+tree* follow a minimum-key rule: If a node N2 is a child of node N1, then any key in the child node N2 is bigger or equal to the key in N1 pointing to N2. Figure 9.4 represents a concrete example of a *B+tree* where each node has maximum 3 keys.

To favor concurrent accesses without using a locking mechanism, DStore *B+tree* implementation is inspired by the work presented in [86]. Unlike the original work, our *B+tree* is designed for one writer and many readers in order to eliminate locking overheads entirely in case of concurrent writers. DStore ensures that only one *slave thread* modifies the *B+tree*, so that there is no need to lock *B+tree* nodes during tree traversals and during the cloning process, as discussed in their paper.

B+tree in DStore is configured to have between `b` and `2b + 1` entries per node and uses a proactive approach for rebalancing. During a tree traversal from root to leaves in an `INSERT` or a `DELETE` operation, a node with `2b + 1` entries is split before examining its child. When a node with `b` entries is encountered, either it has to be merged with its siblings or keys have to be moved from siblings into it. This proactive approach simplifies tree modifications as nodes will not be modified after visited. When shadowing, nodes are cloned in the order that the downward traversal is done from root to the leaves.

To enable shadowing in which different snapshots share *B+tree* nodes, each *B+tree* node has an internal reference counter that records how many parent nodes currently point to it. Figure 9.5 represents the example of the aforementioned *B+tree* with a reference counter in each node.

B+tree operations

Create. To create a new *B+tree*, a root node is initialized with zero entries and its reference counter is set to 1. The root node can contain less than `b` entries while all other nodes have to consist of `b` to `2b + 1` entries.

Clone. To clone a *B+tree* having the root node `V1`, the contents of `V1` will be copied into a new root `V2`. This operation leads to the fact that any child of `V1` is also referenced by the new root `V2`. Therefore, the reference counter in each child node of `V1` has to be increased.

An example of this cloning process is presented in Figure 9.6. The reference counters of two nodes `[0, 6, 10]` and `[13, 20]` are set to 2.

Select. To lookup for a key in a *B+tree*, a downward tree traversal is needed. The algorithm starts examining the *B+tree* root of the desired snapshot and follows the appropriate inner node that covers the input key. When it reaches the leaf, the associate value of the selected key is returned if the key exists. Furthermore, during this downward traversal, no locking is required because tree nodes are immutable, except those of the snapshot under modification. DStore guarantees only read-only snapshots are visible for `Select` operations.

Insert. An `Insert` operation requires a lookup for the corresponding leaf while it has to clone all the nodes in the downward tree traversal. When a node `N` is encountered, the following procedure is executed:

- The reference counter is examined. If it is 1, meaning it only belongs to the current snapshot, there is no need to clone the node. Otherwise, the reference counter is greater than 1 and that node has to be cloned. This is done in order to avoid modifying nodes that also belong to other snapshots. The contents of this node are copied to a new node `N'` with the reference counter set to 1. The reference counter of the node `N` is decremented because `N` no longer belongs to the current snapshot. In addition, the reference counters in children of `N` are incremented to reflect the change that they have another new parent `N'`. The pointer in the parent node of `N` is now pointing to `N'`.
- `N'` is then examined under the proactive split policy. If it is full with `2b + 1` entries, it has to be split. If it has only `b` entries, it has to be merged with its siblings or some keys have to be moved from its siblings into it. As discussed, the proactive split policy prevents modifications from propagating up to parent nodes.

For example, Figure 9.6 shows two snapshots `V1` and `V2` share tree nodes. When inserting a key 15 to the snapshot `V2`, the node `[13, 20]` is cloned first as shown in Figure 9.7. At the end, the leaf `[13, 14, 16]` is cloned and key 15 is added to the new leaf (Figure 9.8).

Delete. To delete a key, the same procedure as in `Insert` is executed. Nodes in the downward tree traversal are cloned and examined under a proactive merge policy. If a node that has the minimum number of `b` entries is encountered, the algorithm merges it with its sibling or moves entries into it. This policy guarantees that a node modification due to a `Delete` affects only its immediate parent.

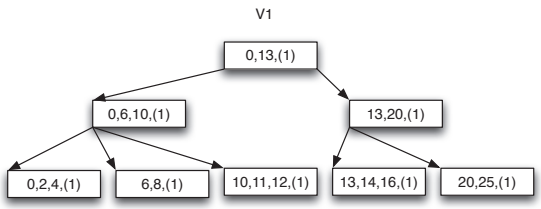


Figure 9.5: Initial *B+tree* V1.

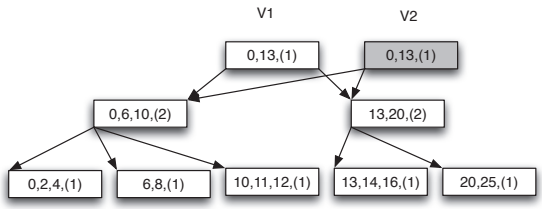


Figure 9.6: Creating a clone V2 of *B+tree* V1.

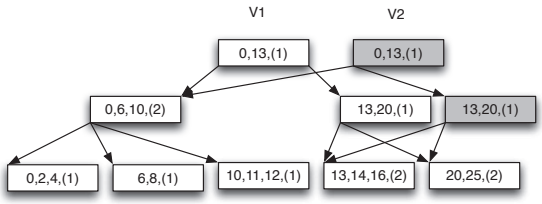


Figure 9.7: Node [13, 20] is cloned.

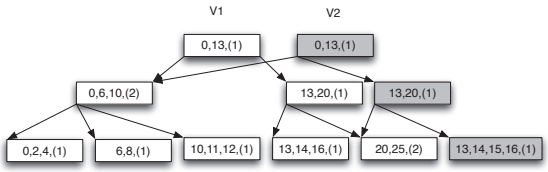


Figure 9.8: Leaf [13, 14, 16] is cloned. The new leaf is modified to host a new key 15.

9.4.2 Bulk merging

Using a *B+tree* implementation with a shadowing mechanism as presented in the previous section, it is easy to implement *bulk merging* in DStore. For each index, a *slave thread* creates a new clone of the latest snapshot of the *B+tree* index by using the *B+tree* `Clone` functionality. This operation is fast, as only a new root is created as a copy of the source *B+tree* root. Then, the *slave thread* starts examining all elements in the *delta buffer* to run the corresponding `Insert` or `Delete` operations on the new clone. When this process is done, the new clone is read-only and is revealed to readers. The *slave thread* then creates a new snapshot based on the new clone and the whole procedure loops again.

Our *bulk merging* approach reduces the number of intermediate snapshots and is potentially faster than a copy-on-write approach. In fact, our approach generates one single snapshot for multiple updates while a copy-on-write approach needs to clone the entire tree traversal from root to leaf for each `Insert` or `Delete`.

9.4.3 Query processing

We now discuss how DStore handles different kinds of queries.

Insert and Delete. `Insert` and `Delete` queries are executed sequentially by the *master thread*. Each `Insert` or `Delete` operation is translated into a series of messages that are fed into the *delta buffers*. Each message consists of a `MESSAGE_TYPE` that defines if it is an `Insert` or a `Delete` to the index, the value of the index key and a pointer to the new Inserted document or `NULL` in case of a `Delete`.

By default, DStore uses a vector datatype for *delta buffer*. When the *delta buffers* are not full, the cost for the *master thread* to finish one `Insert` or `Delete` is $m \times O(1)$ where m is the number of indexes. This is expected to be faster than $m * O(\log(n))$ in case the *master thread* has to manipulate the indexes himself (*B+tree* has $O(\log(n))$ complexity).

Read. DStore supports the following two types of `Read` queries.

Stale Read. DStore favors `Stale Read` operations in order to achieve high performance for analytic processing. A `Stale Read` accesses only the latest snapshots of the *B+tree* indexes that are generated by the *slave threads*. Thanks to the shadowing mechanism that ensures snapshot isolation, each `Stale Read` can be performed independently without any interference with the *slave threads*. Obviously, the *slave threads* are working on the newer snapshots of the indexes and will reveal them to readers only when finished.

`Stale Reads` achieve high performance at the cost of not accessing unindexed updates in the *delta buffers*. Thus, the staleness of results depends on the *delta buffers sizes* and the disparity between the update rate of the *master thread* and the processing rates of *slave threads*.

Obviously, `Stale Reads` on different indexes are independent and are executed in parallel.

Fresh Read. To guarantee the results of `Fresh Read` are up-to-date, both the *delta buffer* and the latest snapshot of the appropriate *B+tree* index needs to be accessed. Compared to `Stale Read`, a `Fresh Read` requires a lock on the *delta buffer* for scanning unindexed updates, thus it negatively impacts on the update rate of the *master thread*. However, we expect the cost for locking is minimal as the *delta buffer* is small size, the *readonly queue* is immutable, and only the *in-coming queue* needs to be locked.

Update. To perform this kind of query in DStore, a `Fresh read` is needed to select the desired document. This operation includes two steps: scanning the *delta buffer* and lookup the *B+tree* structure of the appropriate index. When it is done, the *master thread* can transform the original update query into a series of `Delete` and `Insert` pairs, one for each index. Its purpose is to delete the old index entry and insert a new one to reflect the update in the new snapshots.

Because only the *master thread* executes queries sequentially and any index is updated in bulk, update query is guaranteed to be atomic. For each index, a `Delete` and `Insert` pair is put atomically to the *delta buffer* so that any concurrent `Fresh read` will be aware of the atomic update. Additionally, since an `Update` is translated to a `Delete, Insert` pair, DStore avoids *in-place data modification* as `Delete` and `Insert` only affect an index. Therefore, DStore can achieve parallel index generation in which indexes are built independently by a number of *slave threads*.

One particular case is that some indexes may be updated faster than the others and thus `Read` query on those indexes may return more up-to-date results. However, this is not a problem because it does not break out the atomicity guarantee in DStore.

Complex queries. Complex query is a query involving more than one operation (`Read`, `Insert` and `Delete`, etc.) on more than one document. NoSQL stores often do not support this kind of queries in an atomic fashion [111]. DStore, on the other side, is capable of atomically handing any complex query, which is a combination of aforementioned `Read`, `Insert` and `Delete` operations. This guarantee is simply achieved through two mechanisms: (1) Insert multiple operations to *delta buffers* is atomic. (2) Every *B+tree* index structure is updated in bulk thanks to shadowing. Therefore, a `Stale read` will

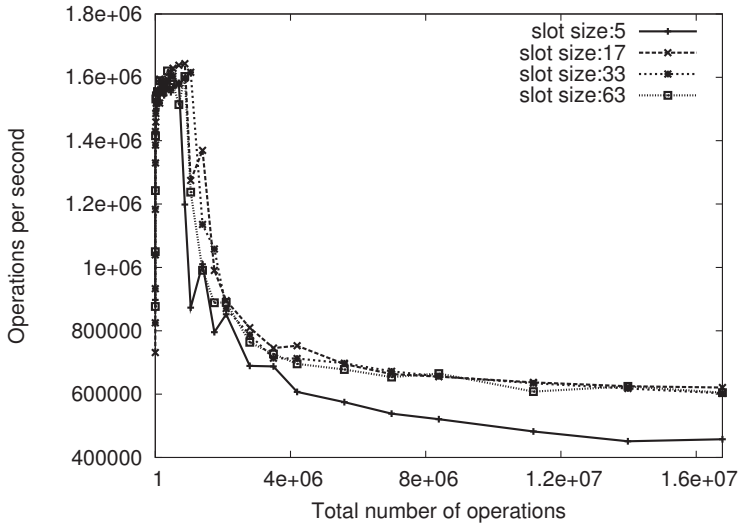


Figure 9.9: Impacts of *B+tree* slot size on DStore performance.

not see partial updates as long as it accesses a read-only snapshot. A `Fresh read` will not see partial updates either as long as the *master thread* can put updates of a complex query to each *delta buffer* atomically.

9.5 Evaluation

DStore is implemented from scratch in C++, using the Boost library [112] for threading and memory management.

We evaluate DStore through a series of synthetic benchmarks, focusing on its design principles presented in Section 9.3.1. Our experiments were carried out on the Grid'5000 testbed, on one node of the Parapluie cluster located in Rennes. The node is outfitted with AMD 1.7 Ghz (2 CPUs, 12 cores per CPU) and 48 GB of main memory.

Impact of the *B+tree* slot size on performance

In DStore, the *B+tree* `slot size` refers to the number of entries configured per *B+tree* node. Changing this value has an impact on the *B+tree* height, which defines the number of steps for a downward traversal from the root to a leaf (the *B+tree* height is equal to $\log_m n$ where `m` is the `slot size` and `n` is the total number of keys). If the `slot size` is too small, examining a tree node to find the pointer to the appropriate child is fast (binary lookup) but more nodes will be accessed before reaching the appropriate leaf. If the `slot size` is too big, tree traversal from root to leaf is fast as the tree height reduced but it will increase the time to access a *B+tree* node. Especially for `Insert` or `Delete` operations, modifications on tree nodes that require merging or shifting keys are slower for bigger `slot size`.

Furthermore, the performance of the *B+tree* implementation with a shadowing mechanism in DStore is heavily influenced by the `slot size`. A bigger `slot size` means bigger

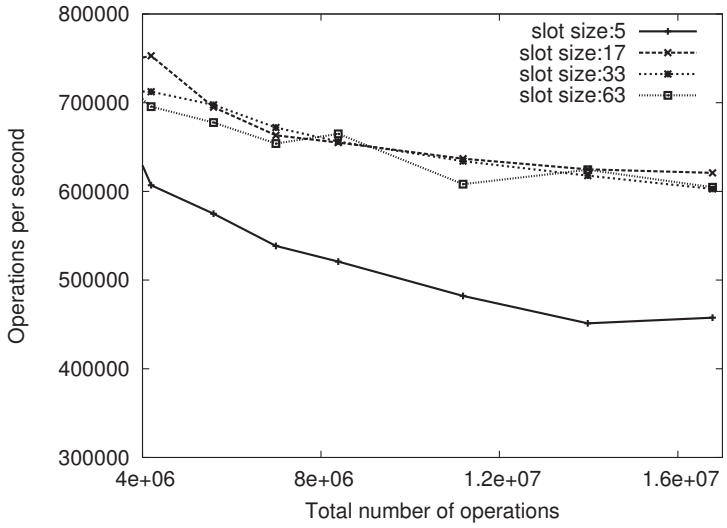


Figure 9.10: A Zoom in the impacts of *B+tree* slot size for large number of Insert operations.

B+tree nodes need to be cloned during shadowing. Thus, it increases the memory consumption as well as the time to copy contents from one node to another. As the `slot size` impacts the *B+tree* performance, it impacts directly the performance of DStore as well.

In the first experiment, we aim to evaluate the impact of the `slot size` on the performance of DStore. To this end, we configure DStore to build only one index. We start by inserting from 1 to 2^{24} distinct random integer key-value pairs to DStore and measure the completion time. We take the total number of `Insert` operations and divide it by the measured completion time to get the insert rate in terms of operations per second. The experiment was done for different `slot sizes` which are 5, 17, 33 and 63 entries. Each test has been executed 3 times and the average was taken into consideration.

The results are shown in Figure 9.9. As observed, DStore with `slot size` 17 achieves the best performance. Figure 9.10 is a zoom in for a clearer view of the impact. When `slot size` is bigger than 17, the cost for shadowing, merging *B+tree* nodes, shifting keys is getting to be higher than what is gained from reducing the tree traversal path. Thus, the performance of DStore did not increase when increasing the `slot size`.

Impact of sorted delta buffers

In this experiment, we evaluate an optimization we introduced in DStore when merging updates in *delta buffers* to the corresponding indexes. As discussed, the *delta buffer* is sorted in order to better leverage caching effects. With a sorted *delta buffer*, there is a higher chance that inserting or deleting an element in a *B+tree* will follow the same traversal path from the *B+tree* root to a leaf or a partial path that was already cached in previous accesses.

We conduct the same tests as in the previous experiment. We keep the `slot size` to be 17 and a maximum *delta buffer size* of 524288 elements. We measure the operations per second when we insert 1 to 2^{24} randomly distinct integer key-value pairs to an empty DStore with only one index. Two cases are examined: *sorted delta buffer* vs *non-sorted delta buffer*.

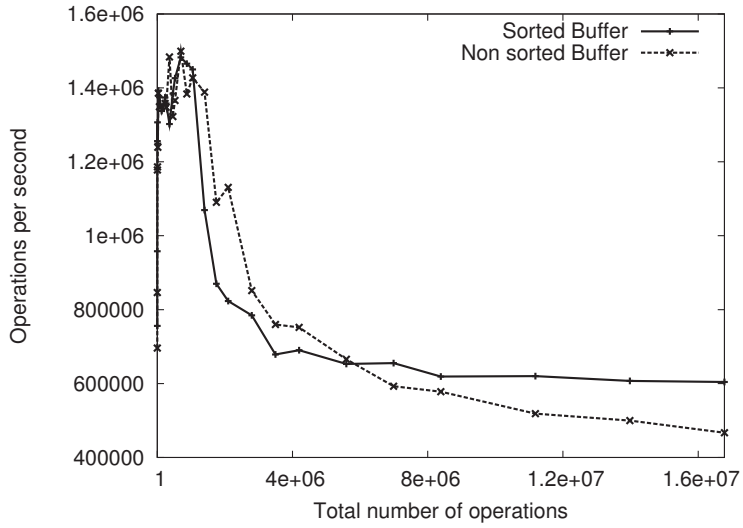


Figure 9.11: Performance of DStore in two cases: Sorted vs non-sorted delta buffers.

As represented in Figure 9.11, DStore with *sorted delta buffer* performs less fast in the beginning, but outperforms the case of *non-sorted delta buffer* when *B+tree* size increases. This can be explained by the cost to sort the *delta buffer*. When the *B+tree* index is small, the caching effect is not significant, such that sorting the input ends up making DStore slower. When the *B+tree* is big, the benefits of caching effects are more significant than the initial effort to sort the *delta buffer*.

Comparing DStore to a pure *B+tree* implementation

We now aim to get a hint of how DStore performs compared to a pure *B+tree* implementation. Our third experiment measures the insert rate in terms of operations per second when we insert from 1 to 2^{24} randomly distinct integer key-value pairs to DStore and to a *B+tree* structure. Again, each test was run 3 times and the average of the results is taken into consideration (the standard deviation was low). Both the pure *B+tree* structure and *B+tree* implementation in DStore were configured to use a `slot size` of 17.

We can observe on Figure 9.12 that DStore performs slightly better. In the beginning, DStore outperformed the *B+tree* approach and this result is due to the *delta buffer*. Internally, each index of DStore is implemented in a producer-consumer model where the *master thread* keeps putting new operations in the *delta buffer* and the *slave thread* in turn takes all operations from the *delta buffer* to update its *B+tree* structure. Consequently, when the *delta buffer* is not full, the *master thread* can finish one insert operation in $O(1)$ time which is far better than $O(\log(n))$ in case of a pure *B+tree* implementation.

Further, as the *slave thread* cannot keep up with the *master thread*, the *delta buffer* gets full in a long-term run. This situation is shown in the right part of the Figure 9.12 where DStore performs only slightly better than the pure *B+tree* structure. Obviously, the *B+tree* implementation with shadowing mechanism in DStore should be slower than a pure *B+tree* due to the cost for shadowing. However, the obtained results can be explained by two reasons. First, DStore sorts the *delta buffer* before merging to the *B+tree* so that it can leverage better the

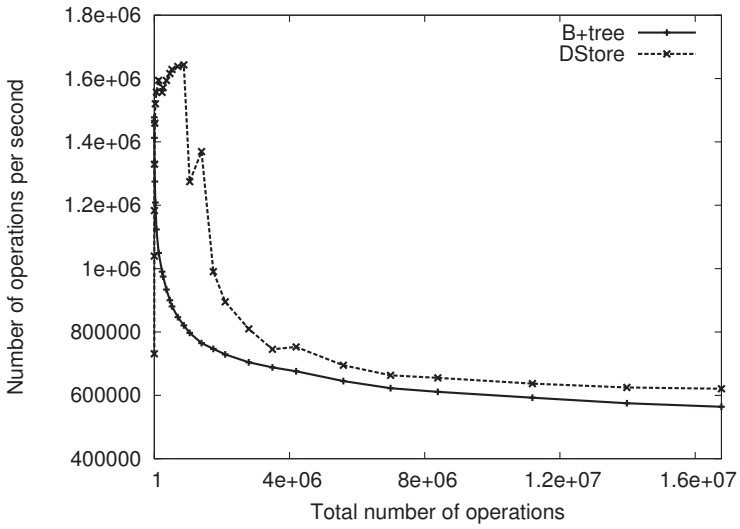


Figure 9.12: B+tree vs DStore: Slot size = 17.

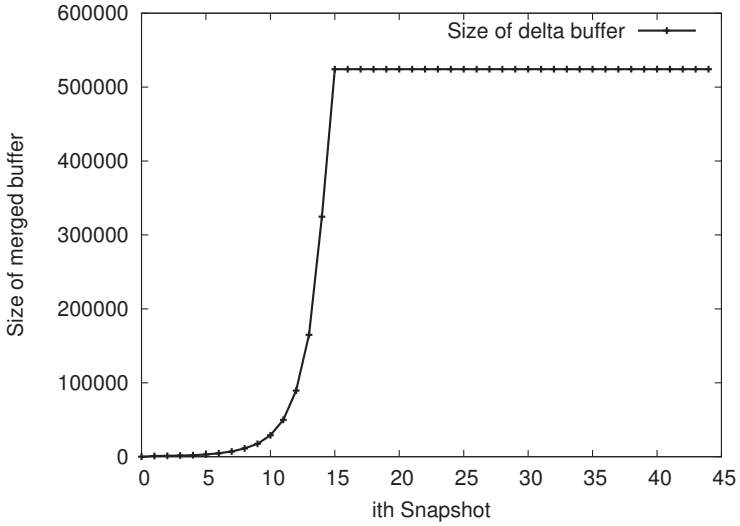


Figure 9.13: Evolution of the delta buffer.

caching effects, as demonstrated in the previous experiment. Second, DStore performs the merging in bulk that creates only one new snapshot to reflect all updates in the *delta buffer*. This minimizes the number of *B+tree* nodes to be cloned and thus increases the shadowing performance.

Figure 9.13 gives more details with regard to the above arguments. When inserting 2^{24} randomly distinct key-value pairs to DStore, only 44 snapshots were created. The curve also shows the evolution of the *delta buffer size*. The *delta buffer* increased to the maximal configured value, when the speed of the *master thread* cannot be faster than that of the *slave thread*.

DStore performance under concurrency

One of the design goal of DStore is to provide high performance for both update queries in transactional processing and read queries in analytical processing. DStore supports one *master thread* for updates, but allows multiple read queries to be processed concurrently with the *master thread* as well. The idea is to leverage a shadowing mechanism to isolate read queries and update queries (`Insert` and `Delete`) in different snapshots, so that they can be processed independently in a lock-free fashion.

To evaluate DStore performance under concurrency, we design one experiment that starts by a warm-up phrase: 2^{24} distinct key-value pairs are inserting to DStore with one index. Then, we launch concurrent readers (up to 14), each of them performing `Stale read` operations that request 2^{24} keys from DStore. In the meantime, the *master thread* keeps inserting and deleting random keys with the purpose of constantly having 2^{24} records in DStore. We measure the number of operations per second for both the *master thread* and the readers.

As expected, Figure 9.14 demonstrates that DStore achieves a good scalability when increasing the number of concurrent readers. Moreover, there is very little overhead on the *master thread* as its performance does not decrease, but remains constant at about 600,000 operations per second.

Impact of building multiple indexes

DStore supports multiple indexes and it provides a mechanism to build those indexes in parallel. In this experiment, we measure the impact of building many indexes on the insert rate in terms of operations per second. For each test, we fix the number of indexes to be built in DStore and start inserting randomly-distinct 2^{24} key-value pairs to DStore. The completion time is measured and the insert rate in operations per second is calculated.

Figure 9.15 shows the insert rate decreases when increasing the number of indexes. The result was anticipated, as the more indexes have to be built, the more work the *master thread* has to do to finish one insert operation. In fact, it has to transform each operation into a series of corresponding operations for each *delta buffer*.

However, the performance of DStore in building multiple indexes is good as compared to the case where the indexes are built sequentially. In that setting, the performance must drop by a factor of 2 when doubling the number of indexes. DStore performance, on the other hand, decreased less than 50 % thanks to the delta indexing mechanism.

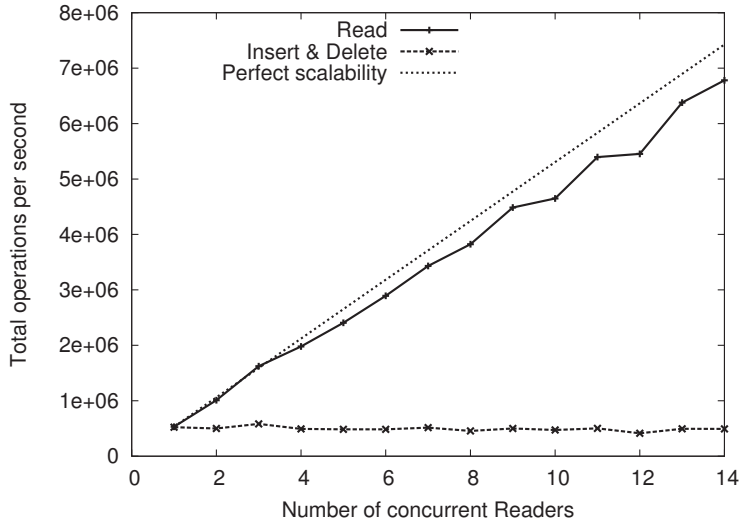


Figure 9.14: DStore performance in concurrency: Multiple readers and one master thread for inserts and deletes.

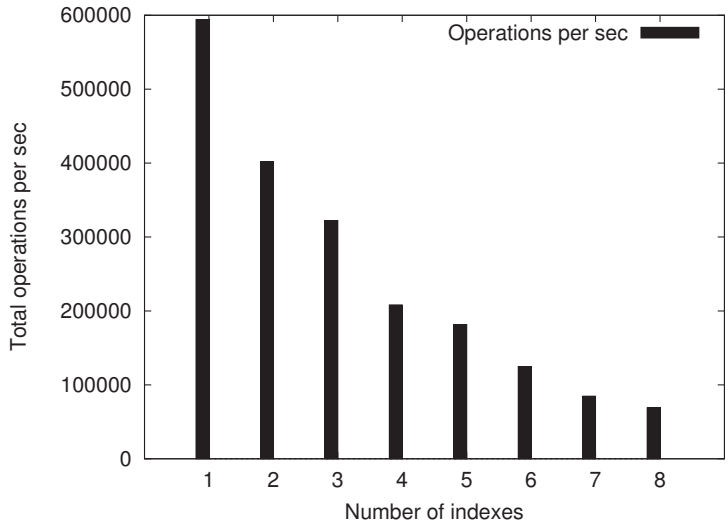


Figure 9.15: DStore performance when building multiple indexes.

Features	DStore	H-Store	HyPer	CouchDB
Document-oriented	Yes	—	—	Yes
In-memory	Yes	Yes	Yes	—
Versioning	Yes	—	—	—
Atomic complex query	Yes	Yes	Yes	—
Concurrent readers	Yes	—	Yes	Yes
Fresh Read	Yes	Yes	Unknown	No/Eventual Consistency
Stale Read	Yes	—	Yes	Yes
Bulk merging	Yes	—	Unknown	—

Table 9.1: A comparison between DStore, H-Store, HyPer and CouchDB.

9.6 Related work

There are several related work as below.

H-Store. H-Store [62] is an experimental row-based relational database management system (DBMS) born in a collaboration between MIT, Brown University, Yale University, and HP Labs. H-Store supports fast transaction processing by leveraging main memory for storing data. To avoid the overhead of locking in multi-threading environments, H-Store follows single-threaded execution model where only one thread is used to execute transactions sequentially.

Compared to our approach, H-Store does not support high-throughput analytical processing. It is only optimized for online transaction processing (OLTP) and cannot execute read queries in parallel. Our approach serializes transaction processing on one single *master thread*, but allows multiple readers to access DStore concurrently without any interference with the *master thread*. Therefore, DStore has the potential to handle efficiently both fast update transactions and high-throughput analytic read queries.

Hyper. HyPer [65] is a main-memory database management system built with the purpose of being able to handle both online transaction processing (OLTP) and online analytical processing (OLAP) simultaneously. HyPer relies on a virtual memory snapshot mechanism that is assisted in hardware by the Operating System (OS) to maintain consistent snapshots for both OLAP and OLTP queries. Upon an OLAP request, HyPer clones the entire database and forks a new process using kernel APIs in the OS. This new process is then able to work on that consistent snapshot without any interference with the main database. In multi-core machines, multiple OLAP threads can be launched simultaneously as long as they only read a private snapshot.

HyPer shares many similarities with our system, but DStore differentiates from HyPer in many aspects. First, DStore is a document-oriented data store rather than a RDBMS. Its simplified document-oriented interface, which is close to the actual data models found in popular Internet applications, allows the system to achieve higher performance under that particular Internet workload [113]. Second, both DStore and HyPer leverage shadowing to separate transactional processing from analytical processing, but DStore does not clone the entire database. DStore implements a *B+tree* shadowing mechanism to clone only indexes and does so in a way that index cloning operations are done in parallel. Thus, our scheme minimizes memory consumption and is potentially faster than that of HyPer.

Moreover, DStore fully supports versioning as the direct result of its shadowing mechanism. DStore maintains all generated snapshots of each index and allows selecting any snapshot for reading purposes. Regarding HyPer, it does not provide a versioning functionality due to an expensive cost of cloning the entire database.

CouchDB. CouchDB [54] is a document-oriented store under the Apache License. Unlike our system, CouchDB does not leverage main memory. It was designed to *scale out* in distributed environments, not to *scale up*. CouchDB supports ACID semantics with eventual consistency, but only for single document access. Complex transactions that update multiple documents are not guaranteed to be atomic.

Moreover, CouchDB leverages Multi-version Concurrency Control (MVCC) to avoid locking on writes. This mechanism is known as copy-on-write that clones the entire traversal path from root to an appropriate leaf of the *B+tree* for each update. As discussed, our cloning scheme is expected to be faster and better in memory consumption.

A comparison between DStore, H-Store, HyPer and CouchDB is summarized in Table 9.1. Because DStore is in an early prototype state rather than a fully implemented system, we cannot perform any performance comparison between DStore and the presented systems. In the near future, we will finalize our comparison by performing more experiments on real-life workloads.

9.7 Summary

In this chapter, we have introduced DStore, a document-oriented store. It is designed to scale up (vertically) in single server by adding more CPU resources and increasing the memory capacity. DStore targets the *Big Velocity* characteristic of Big Data that refers to the high speed of data accessing in storage system. DStore demonstrates fast and atomic transaction processing reflected in the update rate in data writing, while it also delivers high-throughput read accesses for analytical purposes. DStore adds support for atomicity for complex queries and does so with low overheads, property that has not been possible in document-oriented stores designed to scale-out.

In order to achieve its goals, DStore relies entirely on main memory for storing data. It leverages several key design principles, such as: single threaded-execution model, parallel index generation to leverage multi-core architectures, shadowing for concurrency control, and `Stale Read` support for high performance.

Our preliminary synthetic benchmarks demonstrate that DStore achieves high performance even under concurrency, where Read queries, Insert queries and Delete queries are performed in parallel. The experiments show low overheads for both reading and writing when increasing the number of concurrent readers. The measured processing rate was about 600,000 operations per second for each process. Moreover, DStore demonstrates good support for parallel index generations. Indeed, the processing rate does not drop down by a factor of 2 when doubling the number of indexes.

Part V

Conclusions: Achievements and perspectives

Chapter 10

Conclusions

Contents

BIG Data calls for fundamental changes in the architecture of data-management systems. Big Data characterizes the unprecedented growth of all kinds of data that are too "Big" to be managed by traditional data-management systems. These "Big" challenges are *Big Volume*, *Big Velocity* and *Big Variety*. While *Big Volume* describes the huge amount of data that are generated or collected by users or applications, *Big Velocity* refers to the frequency at which the data are generated and need to be handled. As a result of the explosion in data sources, *Big Variety* refers to the growth in the number of data formats that exist more than ever before.

This thesis was carried out in the context of Big Data, focusing on building scalable data-management systems. First, we targeted data-intensive HPC in the sense of *Big Volume*. Such data-intensive HPC applications must manage huge volumes of data with high performance at the level of storage systems. Second, we targeted geographically distributed environments where the storage systems can aggregate storage resources from everywhere

in order to provide a single storage space, whose capability cannot be achieved in smaller scales. Finally, we targeted the *Big Velocity*, *Big Variety* characteristics of Big Data by designing DStore, an in-memory document-oriented store for fast and atomic data update and high throughput data read accesses.

10.1 Achievements

We can describe our achievements in three main research directions as follows.

10.1.1 Scalable distributed storage systems for data-intensive HPC

In the context of data-intensive HPC, we studied the current parallel I/O frameworks. We pointed out the challenges to designing scalable storage systems, that can support efficiently *massive data volume*, *massive parallelization*, *atomicity of non-contiguous I/O*, and *parallel array processing*.

Providing efficient support for MPI-I/O atomicity based on versioning

Typically, the support for atomic, non-contiguous I/O operations is only provided at the level of MPI-I/O implementation. We argued that atomic, non-contiguous I/O operations should be directly supported at the parallel file-system level. To this end, we proposed a versioning-based mechanism that can be leveraged to address the needs of atomic I/O efficiently. This mechanism offers atomicity for non-contiguous I/O without the need to perform expensive synchronizations. We implemented this idea in practice by extending BlobSeer with a non-contiguous data access interface that we directly integrated with ROMIO. Our implementation optimizes the generation of metadata trees so that non-contiguous regions are consolidated into a new independent snapshot, which results in the atomicity of the non-contiguous I/O. In order to perform this efficiently, we proposed two main optimizations. First, we introduced a lazy evaluation scheme to reduce the completion time when building a metadata tree in a bottom-up fashion. Second, we reduced the load on the version manager by moving the metadata node's computation to the clients. This optimization is significant so that it was integrated back to the original BlobSeer.

We conducted 3 series of experiments that compared our BlobSeer-based implementation with a standard locking-based approach where we used Lustre as the underlying storage backend. Our approach demonstrated excellent scalability under concurrency when compared to the Lustre file system. We achieved an aggregated throughput ranging from 3.5 times to 10 times higher in several experimental setups, including highly standardized MPI benchmarks specifically designed to measure the performance of MPI-I/O for non-contiguous overlapped writes that need to obey MPI-atomicity semantics.

Pyramid: a large-scale array-oriented storage system

We have proposed Pyramid, a large-scale, array-oriented storage system optimized for parallel array processing. Our proposal leverages the idea of redesigning the physical data organization inside distributed storage systems in such a way that it closely matches the access

patterns generated by applications. In the context of parallel array processing, we proposed the *multidimension-aware data chunking* model that takes into account the nature of multi-dimensional data models by splitting the array into subdomains. The multi-dimensional chunks are then distributed among the storage servers, which results in a distribution of the I/O workload. Using this approach, related data cells of the initial array have a higher change of residing in the same chunk irrespective of the query type, which greatly favors accessing entire chunks to increase the I/O throughput. Together with the *multidimension-aware data chunking* model, we proposed a distributed metadata-management scheme that avoids potential I/O bottlenecks observed with centralized approaches. Concretely, we introduced a distributed quad-tree like structure that is used to index the chunk layout. Quad-tree nodes are distributed over a DHT implementation for better load balancing. Finally, we proposed to leverage array versioning as a key design principle for efficient concurrency control and for manipulating critical data.

We evaluated Pyramid for highly concurrent data access patterns. Our prototype demonstrates good performance and scalability, both for read and write workloads. It outperforms PVFS in both weak scalability and strong scalability scenarios, by a factor of 100 % to 150 %.

10.1.2 Scalable geographically distributed storage systems

We proposed BlobSeer-WAN, an extended branch of BlobSeer optimized for geographically distributed environments. BlobSeer-WAN was built as part of an integrated architecture comprising a distributed file metadata-management system and a large-scale data-management service. Our work addressed a crucial requirement to extend BlobSeer to take into account the latency hierarchy in an environment where geographically distributed sites are interconnected through WAN networks. First, in order to keep metadata I/O local to each site as much as possible, we proposed an asynchronous metadata replication scheme at the level of metadata providers. As metadata replication is asynchronous, we guarantee a minimal impact on the writing clients that generate metadata. Second, we introduced a distributed version management in BlobSeer-WAN by leveraging an implementation of multiple version managers and using vector clocks for detection and resolution of collision. This extension to BlobSeer keeps BLOBs consistent while they are globally shared among distributed sites under high concurrency.

Several experiments were performed on the Grid'5000 testbed demonstrated that BlobSeer-WAN can offer scalable aggregated throughput when concurrent clients append to one BLOB. The aggregated throughput reached to 1400 MB/s for 20 concurrent clients. Due to the high-speed 10 Gb/s between Nancy and Grenoble, BlobSeer-WAN only achieved the aggregated throughput 15 % higher than the BlobSeer with remote site accesses. Further, we compared BlobSeer-WAN and the original BlobSeer in local site accesses. The experiments shown that the overhead of the multiple version managers implementation and the metadata replication scheme in BlobSeer-WAN is minimal, thanks to our asynchronous replication scheme.

10.1.3 Scalable storage systems in big memory, multi-core machines

To cope with the *Big Velocity* characteristic of Big Data, we have designed and implemented DStore, an in-memory, document-oriented store that scales vertically by leveraging large

memory capability in multi-core machines. DStore demonstrated fast and atomic complex transaction processing in data writing while maintaining high throughput read accesses for analytical purposes. DStore exposes a document-oriented data model that is favored by Internet-specific applications. The data model offers an interesting compromise between the potential system performance and the rich functionality that its interface can deliver. While key-value stores only allow accesses to independent key-value pairs and relational databases are optimized for structured data having associative relationships, document-oriented stores fall in the middle: they organized data in documents that are similar to data rows in RDBMS but they are very flexible and are self-organized.

To achieve its goals, DStore resides in main memory to leverage fast memory accesses. It is built with several design principles. DStore follows a single threaded execution model to execute update transactions sequentially by one *master thread* while relying on a versioning concurrency control to enable multiple *reader threads* running simultaneously. DStore builds indexes for fast document lookups. Those indexes are built using the *delta-indexing* and *bulk updating* mechanisms for faster indexes maintenance and for atomicity guarantees of complex queries. Moreover, DStore is designed to favor stale reads that only need to access isolated snapshots of the indexes. Thus, it can eliminate interference between transactional processing and analytical processing.

We conducted multiple synthetic benchmarks on the Grid'5000 to evaluate the DStore prototype. Our preliminary results demonstrated that DStore achieved high performance even in scenarios where *Read*, *Insert* and *Delete* queries were performed simultaneously. In fact, the processing rate measured was about 600,000 operations per second for each concurrent process.

10.2 Perspectives

This manuscript presented our contributions in several directions towards building scalable data-management systems for Big Data. Each direction addressed some particular challenges in the global context of Big Data management. In this section, we describe several new research directions brought forth by our work.

10.2.1 Exposing versioning at the level of MPI-I/O

Our first contribution is demonstrated in a prototype that explicitly exposes a non-contiguous, versioning-oriented access interface to efficiently address the need of atomic I/O operations. However, this versioning-oriented interface remained hidden to applications when we integrated our prototype with ROMIO, a MPI-I/O implementation.

As an interesting future work direction, we aim at exposing the versioning interface directly at application level by extending the MPI-I/O implementation. The ability to make use of versioning at application level brings several potential benefits. To take an example, consider the case of producer-consumer workloads where the output of simulation is concurrently used as the input of visualization. Using versioning at application level could avoid expensive synchronization schemes, which is an acknowledged problem of current approaches.

10.2.2 Pyramid with active storage support

Many datacenters nowadays consist of machines equipped with commodity hardware that often act as both storage elements and compute elements, called active storage servers. In this context, it is highly desirable to be able to move the computation to the data rather than the other way around, for two reasons: (1) it saves bandwidth, which is especially important when data transfers are expensive (e.g., because of cost concerns or because of high latency/ low bandwidth); (2) it enables better workload parallelization, as part of the work can be delegated to the storage elements (e.g., post-processing filters, compression, etc.).

Pyramid favors a multi-dimensional aware data chunking. Pyramid splits each array into subdomains that are equally sized in each dimension and distributes them across storage servers. This scheme brings an important advantage for active storage support. Data is distributed among multiple storage elements in a way that the neighbors of cells have a higher chance of residing in the same chunk. Thus any computation based only on cell data and its neighbors can be delegated to the chunks, leading to an efficient implicit parallelization. This is a promising research direction for the future versions of Pyramid.

10.2.3 Leveraging Pyramid as a storage backend for higher-level systems

Another future direction concerning Pyramid is to explore the possibility of using Pyramid as a storage backend for SciDB [96] and HDF5 [93]. This direction has a high potential to improve I/O throughput of those systems while keeping compatibility with standardized data access interfaces. Further, Pyramid can be used for scientific applications that process arrays at different resolutions: It is often the case that large subdomains can be easily described implicitly, for instance many times whole subdomains (e.g. zero-filled regions) can be characterized by simple summary information. In this context, our distributed metadata scheme can be enriched to hold such summary information about the subdomains in the tree nodes, which can be relied upon to avoid deeper fine-grain accesses when possible.

10.2.4 Using BlobSeer-WAN to build a global distributed file system

BlobSeer-WAN is a part of an integrated architecture comprising a distributed file metadata-management system and a large-scale data management service. As we finished implementing BlobSeer-WAN, we plan to integrate it to HGMDS in the next step. BlobSeer-WAN will be a low-level object management service, while HGMDS will provide a high-level file system metadata management. The resulting global file system is expected to exhibit scalable file access performance in scenarios where huge files are globally shared among geographically distributed sites.

Furthermore, we plan to run extensive experiments for the integrated system on the infrastructure that features high-latency WAN networks. To this end, we will consider a multiple-site deployment that involves both Grid'5000 and the grid at University of Tsukuba. This work will allow us to providing enough convincing arguments in the performance of the proposed system.

10.2.5 Evaluating DStore with real-life applications and standard benchmarks

We have evaluated DStore with synthetic benchmarks on the Grid'5000 testbed. To provide more convincing results, we plan to evaluate DStore with real-life applications and standard benchmarks. This work will probably require a significant effort either to develop a fully-functional document-oriented interface for DStore or to hard-code the benchmarks with current DStore primitives. We estimate both directions would take 3 to 6 person-months and that is the reason why we could not implement them in the scope of this thesis.

Bibliography

[1] "Large Hadron Collider Grid." wlcg.web.cern.ch.

[2] M. Stonebraker. cacm.acm.org/blogs/blog-cacm/155468-what-does-big-data-mean.

[3] "Hadoop." hadoop.apache.org.

[4] R. Lämmel, "Google's MapReduce programming model — revisited," *Science of Computer Programming*, vol. 68, no. 3, pp. 208–237, 2007.

[5] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era (it's time for a complete rewrite)," in *VLDB '07: Proceedings of the 33rd international conference on very large data bases*, pp. 1150–1160, VLDB Endowment, 2007.

[6] M. Meredith, T. Carrigan, J. Brockman, T. Cloninger, J. Privoznik, and J. Williams, "Exploring Beowulf clusters," *Journal of Computing Sciences in Colleges*, vol. 18, no. 4, pp. 268–284, 2003.

[7] R. D. Meeker, "Comparative system performance for a Beowulf cluster," *J. Comput. Small Coll.*, vol. 21, no. 1, pp. 114–119, 2005.

[8] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[9] V. S. Sunderam, "PVM: a framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315–339, Nov. 1990.

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

[11] J. Markoff and S. Hansell, "Hiding in plain sight, Google seeks more power," *The New York Times*, June 2006.

[12] "K computer." www.riken.jp.

[13] I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a new computing infrastructure*. Morgan Kaufmann, 1999.

[14] I. Foster, "What is the Grid? a three point checklist," *GRIDtoday*, vol. 1, July 2002.

[15] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," in *NPC'05: Proceedings of th International Conference on Network and Parallel Computing* (Springer-Verlag, ed.), pp. 2–13, 2005.

[16] M. Romberg, "The UNICORE Grid infrastructure," *Scientific Programming*, vol. 10, no. 2, pp. 149–157, 2002.

[17] C. Marco, C. Fabio, D. Alvise, G. Antonia, G. Francesco, M. Alessandro, M. Moreno, M. Salvatore, P. Fabrizio, P. Luca, and P. Francesco, "The gLite workload management system," in *GPC '09: Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing*, pp. 256–268, Springer-Verlag, 2009.

[18] E. Laure and B. Jones, "Enabling Grids for e-Science: The EGEE project," Tech. Rep. EGEE-PUB-2009-001, The EGEE Project, September 2008.

[19] K. Stanoevska-Slabeva and T. Wozniak, "Cloud basics – an introduction to Cloud computing," in *Grid and Cloud Computing*, pp. 47–61, Springer-Verlag, 2010.

[20] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the Clouds: Towards a Cloud definition," *ACM SIGCOMM: Computer Communication Review*, vol. 39, pp. 50–55, January 2009.

[21] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley view of Cloud computing," Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009.

[22] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and Grid computing 360-degree compared," in *GCE '08: Grid Computing Environments Workshop*, pp. 1–10, November 2008.

[23] K. Keahey and T. Freeman, "Science Clouds: Early experiences in Cloud computing for scientific applications," in *CCA '08: Cloud Computing and Its Applications*, 2008.

[24] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus open-source Cloud computing system," in *CCGRID'09: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 124–131, IEEE Computer Society, 2009.

[25] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Elastic management of cluster-based services in the Cloud," in *ACDC'09: Proceedings of the 1st workshop on automated control for datacenters and Clouds*, pp. 19–24, ACM, 2009.

[26] D. Robinson, *Amazon Web Services Made Simple: Learn how Amazon EC2, S3, SimpleDB and SQS Web Services enables you to reach business goals faster*. Emereo Publishing, 2008.

[27] D. Sanderson, *Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure*. O'Reilly, 2009.

[28] T. Redkar, *Windows Azure Platform*. Apress, 2010.

[29] "Salesforce." www.salesforce.com/platform.

[30] "Google Docs." docs.google.com.

[31] "Microsoft Office Live." www.officelive.com.

[32] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon S3 for science Grids: A viable solution?," in *DADC '08: Proceedings of the 2008 international workshop on data-aware distributed computing*, pp. 55–64, ACM, 2008.

[33] J. Bresnahan, K. Keahey, D. LaBissoniere, and T. Freeman, "Cumulus: An open-source storage Cloud for science," in *ScienceCloud '11: Proceedings of the 2nd international workshop on Scientific cloud computing*, pp. 25–32, ACM, 2011.

[34] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82 –85, January 1998.

[35] S. Naffziger, J. Warnock, and H. Knapp, "SE2 when processors hit the power wall (or "when the CPU hits the fan")," in *ISSCC '05: IEEE International Solid-State Circuits Conference*, pp. 16 –17, February 2005.

[36] M. D. Hill, "What is scalability?," *ACM SIGARCH Computer Architecture News*, vol. 18, pp. 18–21, Dec. 1990.

[37] K. Magoutis, *Exploiting direct-access networking in network-attached storage systems*. PhD thesis, Harvard University, 2003. Advisor: Seltzer Margo.

[38] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on management of data*, pp. 109–116, ACM, 1988.

[39] M. Mesnier, G. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 41, pp. 84–90, August 2003.

[40] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM Storage Tank: A heterogeneous scalable SAN file system," *IBM Systems Journal*, vol. 42, no. 2, pp. 250 –267, 2003.

[41] "EMC." www.tech.proact.co.uk/emc/emc_celerra_highroad.htm.

[42] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, USENIX Association, 2002.

[43] B. Nicolae, G. Antoniu, and L. Bougé, "Enabling high data throughput in desktop Grids through decentralized data and metadata management: The BlobSeer approach," *Euro-Par 2009: Parallel Processing*, pp. 404–416, 2009.

[44] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, pp. 169–184, February 2011.

[45] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, pp. 317–327, USENIX Association, 2000.

[46] "Information technology - portable operating system interface (POSIX)." ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5393893&isnumber=5393892, 2009.

[47] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the Linux Symposium*, pp. 380–386, 2003.

[48] S. A. Weil, S. A. Brandt, E. L. Miller, D. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 307–320, USENIX Association, 2006.

[49] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for Petabyte-scale file systems," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pp. 4–16, IEEE Computer Society, 2004.

[50] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 29–43, ACM Press, 2003.

[51] M. Stonebraker and U. Cetintemel, "One size fits all: An idea whose time has come and gone," in *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pp. 2–11, 2005.

[52] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[53] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[54] "CouchDB." couchdb.apache.org.

[55] "Riak." wiki.basho.com.

[56] "MongoDB." www.mongodb.org.

[57] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "BigTable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[58] S. Venugopal, R. Buyya, and K. Ramamohanarao, "A taxonomy of Data Grids for distributed data sharing, management, and processing," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 3, 2006.

[59] O. Tatebe and S. Sekiguchi, "Gfarm v2: A Grid file system that supports high-perfomance distributed and parallel data computing," in *Proceedings of the 2004 Computing in High Energy and Nuclear Physics*, 2004.

[60] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "The XtreemFS architecture - a case for object-based file systems in Grids," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 17, pp. 2049–2060, 2008.

[61] "Oracle Timesten." www.oracle.com/us/corporate/acquisitions/timesten.

[62] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[63] "SAP HANA." www.sap.com/hana/.

[64] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP through the looking glass, and what we found there," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on management of data*, pp. 981–992, ACM, 2008.

[65] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots," in *ICDE '11: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, pp. 195 –206, April 2011.

[66] B. Nicolae, *BlobSeer: Towards efficient data storage management for large-scale, distributed systems*. PhD thesis, University of Rennes 1, November 2010. Advisors: Gabriel Antoniu and Luc Bougé.

[67] B. Nicolae, D. Moise, G. Antoniu, L. Bougé, and M. Dorier, "BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map/Reduce applications," in *IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, pp. 1–12, 2010.

[68] "Berkeley DB." www.oracle.com/technetwork/products/berkeleydb.

[69] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed segment tree: Support of range query and cover query over DHT," in *IPTPS '06: Proceedings of the 5th International Workshop on Peer-to-Peer Systems*, 2006.

[70] G. Bryan, "CM1 homepage." www.mmm.ucar.edu/people/bryan/cm1.

[71] "WRF." www.wrf-model.org.

[72] R. L. Graham, "The MPI 2.2 standard and the emerging MPI 3 standard," in *EuroMPI '09: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 2–2, 2009.

[73] "Parallel HDF5." www.hdfgroup.org/HDF5/PHDF5.

[74] J. Li, W.-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on supercomputing*, pp. 39–47, 2003.

[75] "VisIt." wci.llnl.gov/codes/visit.

[76] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-I/O portably and with high performance," in *IOPADS '99: Proceedings of the 6th Workshop on I/O in parallel and distributed systems*, pp. 23–32, ACM, 1999.

[77] E. Smirni, R. Aydt, A. A. Chien, and D. A. Reed, "I/O requirements of scientific applications: An evolutionary view," in *HPDC '02: Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*, pp. 49–59, IEEE, 2002.

[78] E. Smirni and D. A. Reed, "Lessons from characterizing the I/O behavior of parallel scientific applications," *Performance Evaluation*, vol. 33, no. 1, pp. 27–44, 1998.

[79] A. Ching, A. Choudhary, K. Coloma, W.-K. Liao, R. Ross, and W. Gropp, "Noncontiguous I/O accesses through MPI-I/O," in *CCGRID '03: Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 104–111, May 2003.

[80] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems," in *CLUSTER '09: Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, pp. 1–10, September 2009.

[81] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O," in *CLUSTER '12: Proceedings of the IEEE International Conference on Cluster Computing*, IEEE, Sept. 2012.

[82] P. M. Dickens and J. Logan, "A high-performance implementation of MPI-I/O for a Lustre file system environment," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 1433–1449, August 2010.

[83] J. Garcia Blas, F. Isaila, J. Carretero, D. Singh, and F. Garcia-Carballeira, "Implementation and evaluation of file write-back and prefetching for MPI-I/O over GPFS," *International Journal of High Performance Computing Applications*, vol. 24, pp. 78–92, February 2010.

[84] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed, "Input/Output characteristics of scalable parallel applications," in *SC '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, ACM, 1995.

[85] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best, "File-access characteristics of parallel scientific workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 1075–1089, October 1996.

[86] O. Rodeh, "B-trees, shadowing, and clones," *ACM Transactions on Storage*, vol. 3, no. 4, pp. 1–27, 2008.

[87] A. Ching, W.-K. Liao, A. Choudhary, R. Ross, and L. Ward, "Noncontiguous locking techniques for parallel file systems," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–12, November 2007.

[88] Y. Jégou, S. Lantéri, J. Leduc, M. Noredine, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Iréa, "Grid'5000: a large scale and highly reconfigurable experimental Grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, pp. 481–494, November 2006.

[89] I. F. Haddad, "PVFS: A parallel virtual file system for Linux clusters," *Linux Journal*, November 2000.

[90] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen, "Implementing MPI-I/O atomic mode without file system support," in *CCGRID '05: Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid*, pp. 1135–1142, IEEE Computer Society, 2005.

[91] S. Sehrish, J. Wang, and R. Thakur, "Conflict detection algorithm to minimize locking for MPI-I/O atomicity," in *Euro PVM/MPI '09: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 143–153, Springer-Verlag, 2009.

[92] W.-K. Liao, A. Choudhary, K. Coloma, G. Thiruvathukal, L. Ward, E. Russell, and N. Pundit, "Scalable implementations of MPI atomicity for concurrent overlapping I/O," in *ICPP '03: Proceedings of International Conference on Parallel Processing*, pp. 239–246, October 2003.

[93] "HDF5." www.hdfgroup.org/about/hdf_technologies.html.

[94] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible I/O and integration for scientific codes through the adaptable I/O system (ADIOS)," in *CLADE '08: Proceedings of the 6th international workshop on challenges of large applications in distributed environments*, pp. 15–24, 2008.

[95] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, ACM, 2009.

[96] P. Brown, "Overview of SciDB: large scale array storage, processing and analysis," in *SIGMOD '10: Proceedings of the 2010 International conference on Management of data*, pp. 963–968, ACM, 2010.

[97] M. Stonebraker, J. Becla, D. Dewitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik, "Requirements for science databases and SciDB," in *CIDR '09: Proceedings of the 4th Conference on Innovative Data Systems Research*, 2009.

[98] E. Soroush, M. Balazinska, and D. Wang, "ArrayStore: a storage manager for complex parallel array processing," in *SIGMOD '11: Proceedings of the 2011 International conference on management of data*, pp. 253–264, ACM, 2011.

[99] V.-T. Tran, B. Nicolae, G. Antoniu, and L. Bougé, "Efficient support for MPI-I/O atomicity based on versioning," in *CCGRID '11: Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 514–523, 2011.

[100] J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in databases: Why, how, and where," *Foundations and trends in databases*, vol. 1, pp. 379–474, April 2009.

[101] "ANR-JST FP3C project." jfli.nii.ac.jp/medias/wordpress/?page_id=327.

[102] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw, "LegionFS: a secure and scalable file system supporting cross-domain high-performance applications," in *SC '01: Proceedings of the 2001 ACM/IEEE conference on supercomputing*, pp. 59–59, ACM Press, 2001.

[103] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data management and transfer in high-performance computational Grid environments," *Parallel Computing*, vol. 28, pp. 749–771, May 2002.

[104] A. Bassi, M. Beck, G. Fagg, T. Moore, J. S. Plank, M. Swany, and R. Wolski, "The Internet Backplane Protocol: A study in resource sharing," in *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, p. 194, IEEE Computer Society, 2002.

[105] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: the future building block for storage systems," in *Local to Global Data Interoperability - Challenges and Technologies*, pp. 119–123, June 2005.

[106] K. Hiraga. www.hpcs.cs.tsukuba.ac.jp/~tatebe, Mar. 2011.

[107] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Information Processing Letters*, vol. 43, no. 1, pp. 47–52, 1992.

[108] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas, "Nobody ever got fired for using Hadoop on a cluster," in *HotCDP '12: Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, pp. 2:1–2:5, ACM, 2012.

[109] D. G. Severance and G. M. Lohman, "Differential files: their application to the maintenance of large databases," *ACM Transactions on Database Systems (TODS)*, vol. 1, pp. 256–267, Sept. 1976.

[110] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, C. A. Soules, and A. Veitch, "Lazybase: trading freshness for performance in a scalable database," in *EuroSys '12: Proceedings of the 7th ACM european conference on Computer Systems*, pp. 169–182, ACM, 2012.

[111] S. Das, D. Agrawal, and A. E. Abbadi, "G-Store: a scalable data store for transactional multi key access in the Cloud," in *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pp. 163–174, ACM, 2010.

[112] "Boost." www.boost.org.

[113] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

Part VI

Appendix

Chapter 11

Résumé en Francais

Contents

11.1 Context: Big Data management

À ce jour, les données sont capturées and générées plus que jamais auparavant. Selon l'étude de l'International Data Corporation (IDC), la quantité d'informations créées et reproduites mondialement en 2011 a dépassé 1,8 Zettabytes (ZB), ce qui marque une croissance exponentielle par un facteur de neuf en seulement cinq ans. En plus, l'étude montre également que la quantité de données générées directement par les utilisateurs individuels tels que des documents, des photos, numériques musiques, blogs est beaucoup moins que la quantité de données reproduites par les applications et les services d'Internet au sujet de leurs activités en ligne.

Les sources de cette explosion des données peuvent être facilement identifiés. Aujourd'hui, le monde compte environ 5 milliards de téléphones mobiles, des millions de capteurs pour capturer quasiment tous les aspects de la vie. Comme un exemple particulier de Facebook, plus de 900 millions d'utilisateurs actifs partage environ 30 milliards de morceaux

de contenu par mois. Dans la même durée d'intervalle, plus de 20 milliards de recherches sont effectuées sur Internet.

Dans ce contexte, le terme "Big Data" est utilisé pour caractériser l'explosion récente des données. Dans ce contexte, le terme "Big Data" est utilisé pour caractériser l'explosion récente des données. Selon M. Stonebreaker, Big Data peut être défini comme " l'a 3V du volume, de la vitesse et de la variété" [2]. Le traitement des données de Big Data se réfère à des applications qui ont au moins une des caractéristiques suivantes.

Big Volume. L'analyse de Big Data doit souvent traiter des Téraoctets (TBs) des données, et même plus. Quotidiennement, les utilisateurs de Twitter produisent approximativement de 7 TBs des données, celles de Facebook partagent un milliard d'éléments de contenue qui valent 10 TBs. It est évident que le volume de données est le défi le plus immédiat pour les systèmes conventionnelles de gestion de données. Cela nécessite un changement fondamental dans l'architecture des systèmes de stockage évolutif. En fait, de nombreuses entreprises ont archivé une grande quantité de données sous forme de logs, mais ils ne sont pas capables de les traiter en raison d'un manque de cadres appropriés de matériel et de logiciel pour Big Data.

L'état de l'art souligne la popularité des cadres de traitement en parallèle comme des outils efficaces pour analyser Big Data (par exemple, le cadre Hadoop avec la modèle de programmation MapReduce qui est inspiré par le Map et la reduction de la programmation fonctionnelle). Grâce à MapReduce, les sociétés qui ont un grand volume de données peuvent maintenant produire rapidement des analyses significatifs à partir de leurs données afin d'améliorer leurs services et de diriger leurs produits mieux.

Big Velocité. A signification conventionnelle de la vélocité est la rapidité avec laquelle les données sont générées et doivent être traitées. A fin de fournir rapidement des nouvelles connaissances à partir des données d'entrée, le traitement par lots comme dans MapReduce n'est plus la seule solution préférée. Il est de plus en plus nécessaire de traiter les données "en-ligne", où la vitesse du traitement doit être proche de celle des flux de données d'entrée. Un exemple intuitive est la suivante: si ce que nous avions était seulement un instantané datant de 10 minutes des flux de trafic, nous n'aurions pas osé traverser la route parce que l'état du trafic change si rapidement. C'est un des nombreux cas où la MapReduce et l'Hadoop ne peuvent pas tenir la vitesse exigée du traitement des données. On ne peut pas attendre pour un travail par lots sur l'Hadoop se terminé si les données d'entrée changerait avant que nous obtenions le résultat des traitements par lots précédents.

Big Variété. De nos jours, les source de données sont diverses. Avec l'exposition des gadgets Internet-capables, des données tell que les textes, les images sont collectés à partir de n'importe où par n'importe quel dispositif (par exemple, les capteurs, les téléphones mobiles, etc.). Elles sont en termes de données non-structurées, structurées et semi-structurées. Donc, les données sont devenues si complexes que les systèmes de stockage spécialisés sont nécessaires pour faire face aux multiples formats de données efficacement.

La tendance récente de stockage a montré l'augmentation des approaches NoSQL. Bien que les bases de données relationnelles peuvent être utilisés dans n'importe quelle circonstance, pour tous les formats de données, elles ne sont que performantes pour

les données relationnelles []. En mettant en œuvre seulement des modèles de données qui sont indispensables pour les applications (par exemple, la modèle orientée clé-valeur ,la modèle orientée document, la modèle orientée graphie), NoSQL est capables d'évoluer horizontalement pour s'adapter à la augmentation des charges de travail.

Pour répondre aux besoins de la gestion de Big Data, il faut des changements fondamentaux dans l'architecture des systèmes de stockage des données. Les stockages de données devraient continuer d'innover afin de s'adapter à la croissance de Big Data. Ils ont besoin d'être évolutive, tout en maintenant de hautes performances pour l'accès aux données. Donc, cette thèse se concentre sur le renforcement des systèmes évolutives de gestion des données pour Big Data.

11.2 Contributions

Nous pouvons décrire nos contributions dans trois principaux axes de recherche dans ce qui suit.

11.2.1 Systèmes de stockage scalable, distribués pour la calcul haute performance (HPC) de données à forte intensité

Dans le context de HPC de données à forte intensité, nous étudions les cadres actuels des E/S (entrées-sorties) parallèles. Nous soulignons les difficultés à concevoir des systèmes de stockage évolutives, qui peuvent soutenir efficacement *le volume massive de données*, *la parallélisation massive*, *l'atomicité d'accès d'E/S non-contiguë* et *le traitement en parallèle des matrices de données*.

Fournir un support efficace pour l'atomicité MPI basé sur le versionnage des données

Typiquement, le support pour les opérations d'E/S atomiques, non-contiguës n'est fourni qu'au niveau de MPI I/O. Nous soutenons que l'atomicité des opérations d'E/S non-contiguës doivent être directement pris en charge sur le niveau des systèmes de gestion de fichiers parallèles. Donc, nous proposons un méchanisme basé sur le versionnage, qui peut être utilisé pour satisfaire les besoins des opérations atomiques de maniere efficace. Ce mécanisme assure l'atomicité pour les opérations d'E/S non-contiguës sans la nécessité d'effectuer des synchronisations qui peuvent être très coûteux. Il s'appuie sur l'idée que les modifications d'un fichier par une écriture non-contiguë engendre un nouveau snapshot indépendant. Nous basons sur les méta-données pour mémoriser les données dans chaque snapshot, même si les données viennent d'un écriture non-contiguë.

Nous mettons en œuvre cette idée en pratique en étendant BlobSeer avec une interface d'accès non contiguë. En plus, nous intégrons notre prototype avec ROMIO, une implementation de MPI I/O. Nous proposons deux principales optimisations dans notre prototype basé sur BlobSeer. Tout d'abord, nous avons mis en place un schéma d'évaluation paresseuse pour réduire le temps d'exécution lors de la construction d'un arbre de méta-données pour un snapshot particulier. Deuxièmement, nous réduisons la charge sur le serveur gestionnaire de versions en déplaçant le calcul des nœuds des méta-données aux clients. Cette optimisation est significatif de telle sorte qu'il est intégré au retour à la BlobSeer originale.

Nous avons effectué 3 séries d'expériences pour comparer notre prototype basée sur BlobSeer et une approche de verrouillage où nous avons utilisé Lustre. Notre prototype a démontré une scalabilité excellente sous les accès concurrentiels par rapport au système de fichiers Lustre. Nous avons atteint un débit agrégé allant de 3,5 fois à 10 fois plus dans plusieurs configurations expérimentales, y compris des repères hautement standardisés spécialement conçus pour mesurer la performance de MPI-I/O pour les opérations non-contiguës.

Pyramid: un système de stockage à grande échelle, orienté matrice

Dans le contexte du calcul haute performance (HPC) de données à forte intensité, d'une large classe d'applications nécessite un traitement des matrices en parallèle : les petits sous-domaines différents d'un seul tableaux multi-dimensionnels sont accédé concurremment par un grand nombre de clients, tant pour la lecture et l'écriture. Parce que les données multidimensionnelles sont sérialisés en une séquence d'octets au niveau du système de stockage, un sous-domaine (malgré vue par les processus applicatifs comme un seul bloc de mémoire) correspond à une série de complexes régions non-contiguës du fichier de données, qui doivent être lues/écrites ensemble par le même processus. Nous proposons d'éviter une telle sérialisation coûteuse qui détruit la localité des données en redéfinissant la façon dont les données sont stockées dans les systèmes de stockage distribué, pour qu'elle corresponde au modèle d'accès généré par les applications. Nous concevons et réalisons Pyramid, un système de stockage à grande échelle, orienté matrice, qui exploite un modèle d'accès orienté matrice et un contrôle de concurrence basé sur le versionnage pour pouvoir soutenir le traitement des matrices en parallèle de manière efficace. L'évaluation expérimentale démontre des améliorations importantes sur la scalabilité apportées par Pyramide par rapport aux approches de l'état de l'art, avec des gains de 100 % à 150 %.

11.2.2 Systèmes de stockage scalables, distribués géographiquement

Nous proposons BlobSeer-WAN, une extension de BlobSeer optimisé pour les environnements répartis géographiquement. BlobSeer-WAN est construit dans le cadre d'une architecture intégrée comprenant un système de gestion de métadonnées répartis et un service de gestion de données à grande échelle. Notre travail répond à une exigence cruciale pour étendre BlobSeer de prendre en compte la hiérarchie de latence dans un environnement où des sites distribués géographiquement sont interconnectés par des réseaux WAN. Tout d'abord, afin de conserver les opérations sur les méta-données locales dans chaque site, autant que possible, nous proposons un schéma de réplication de méta-données asynchrone au niveau des serveurs de gestion de métadonnées. Comme la réplication des métadonnées est asynchrone, nous garantissons un impact minimal sur les opérations d'écriture chez les clients. Deuxièmement, nous introduisons un système de gestion de version distribué dans BlobSeer-WAN et utilisons des horloges vectorielles pour la détection et la résolution de collision. Cette extension de BlobSeer maintient BLOBs cohérentes tandis qu'ils sont globalement partagées entre les sites distribués, et sous une forte concurrence.

Plusieurs évaluations expérimentales réalisées sur le testbed Grid'5000 ont démontré que BlobSeer-WAN peut offrir un débit agrégé évolutive lorsque les clients écrivent simultanément sur un BLOB particulier. Le débit agrégé a atteint à 1400 Mo/s pour 20 clients concur-

rentes. En raison d'un haut-débit de 10 Gb/s entre Nancy et Grenoble, le débit agrégé de BlobSeer-WAN n'a atteint que 15 % plus que ceci de BlobSeer avec les accès à site distance. De plus, nous avons comparé BlobSeer-WAN et le BlobSee dans le mode d'accès locals. Les expériences montrent que le surcoût de la gestion des versions distribuée et du système de réplication des métadonnées dans BlobSeer-WAN est minimal, grâce à notre schéma de réplication asynchrone.

11.2.3 Systèmes de stockage scalables pour les machines multi-cœur avec de grande mémoire

À la suite de l'innovation continue dans la technologie du matériel, les ordinateurs sont de plus en plus puissants que leurs modèles précédents. Les serveurs modernes de nos jours possèdent une grande capacité de mémoire vide dont la taille est jusqu'à 1 téra-octets (To) et plus. Puisque les accès en mémoire vide sont au moins 100 fois plus rapide que sur le disque dur, conserver les données dans la mémoire vide devient un principe de conception intéressant pour augmenter la performance des systèmes de gestion des données. Nous désignons DStore, un stockage orienté documents résidant en mémoire vide pour tirer parti de la haute vitess des accès en mémoire vide. DStore peut se mettre à l'échelle en augmentant la capacité de mémoire vide et le nombre de cœurs de CPU au lieu de faire mettre à l'échelle horizontalement comme dans les systèmes de gestion de données répartis. Cette décision de conception favorise DStore à soutenir des transactions complexes en assurant la rapidité et l'atomicité, tout en conservant un haut débit élevé pour le traitement analytique (seulement des lectures). Cet objectif est (à notre connaissance) pas facile à atteindre dans les environnements distribués.

DStore est construit avec plusieurs principes de conception. DStore suit un modèle d'exécution utilisée un seul thread (*master thread*) qui exécute des transactions de mise à jour de manière séquentielle. It s'appuie également sur un contrôle de concurrence basé sur le versionnage afin de permettre multiples *reader threads* à lire simultanément. DStore construit des indices pour accélérer la recherche des documents. Ces indices sont construits en utilisant des mécanismes de *delta indexing* et *bulk updating* pour maintenir efficacement et bien pour garantir l'atomicité des requêtes complexes. De plus, DStore est conçu pour favoriser des lectures stales qui ont seulement besoin d'accéder à des clichés isolés des indices. It peut donc éliminer l'interférence entre le traitement transactionnel et le traitement analytique.

Nous avons effectué plusieurs benchmarks synthétiques sur le Grid'5000 afin d'évaluer le prototype DStore. Nos résultats préliminaires ont démontré que DStore a obtenu de bons résultats même dans des scénarios où les opérations *lire*, *Insérer* et *Supprimer* ont été exécutées simultanément. En fait, la cadence de traitement est mesurée d'environ 600.000 opérations par seconde pour chaque processus concurrent.

11.3 Publications

Journal:

- *Towards scalable array-oriented active storage: the Pyramid approach.* Tran V.-T., Nicolae B., Antoniu G. In the ACM SIGOPS Operating Systems Review 46(1):19-25. 2012. (Ex-

tended version of the LADIS paper). http://hal.inria.fr/hal-00640900

Conférences et ateliers internationaux:

- *Pyramid: A large-scale array-oriented active storage system.* Tran V.-T., Nicolae B., Antoniu G., Bougé L. In The 5th Workshop on Large Scale Distributed Systems and Middleware (LADIS 2011), Seattle, September 2011. http://hal.inria.fr/inria-00627665
- *Efficient support for MPI-IO atomicity based on versioning.* Tran V.-T., Nicolae B., Antoniu G., Bougé L. In Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2011), 514 - 523, Newport Beach, May 2011. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5948642
- *Towards A Grid File System Based on a Large-Scale BLOB Management Service.* Tran V.-T., Antoniu G., Nicolae B., Bougé L. In Proceedings of the CoreGRID ERCIM Working Group Workshop on Grids, P2P and Service computing, Delft, August 2009. http://hal.inria.fr/inria-00425232

Posters:

- *Towards a Storage Backend Optimized for Atomic MPI-I/O for Parallel Scientific Applications.* Tran V.-T. In The 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011): PhD Forum (2011), 2057 - 2060, Anchorage, May 2011. http://hal.inria.fr/inria-00627667

Rapports de recherche:

- *Efficient support for MPI-IO atomicity based on versioning.* Tran V.-T., Nicolae B., Antoniu G., Bougé L. INRIA Research Report No. 7787, INRIA, Rennes, France, 2010. http://hal.inria.fr/inria-00546956
- *Un support efficace pour l'atomicité MPI basé sur le versionnage des données* Tran V.-T. INRIA Research Report. INRIA, Rennes, France, 2011. http://hal.inria.fr/hal-00690562

11.4 Organisation du manuscrit

Le reste de cette thèse est organisé en cinq parties, brièvement décrites ci-après.

Partie I: La scalabilité des des systèmes de gestion de Big Data

Nous discutons le contexte de notre travail en présentant les domaines de recherche relis. Cette partie est constituée du chapitre 2, 3 et 4. Chapitre 2 introduit Big Data et l'état de l'art des infrastructures de gestion de Big Data actuelles. En particulier, nous nous concentrons tout d'abord sur des infrastructures distribuées qui sont conçus pour agréger les ressources tels que les clusters, les grilles, et les clouds. Deuxièmement, nous introduisons une infrastructure centralisée qui a attiré de plus en plus l'attention dans le monde du traitement

de Big Data: un seul serveur avec un processeur multi-cœur et une très grande mémoire vide. Chapitre 3 restreint la focalisation sur les systèmes de gestion des données actuels, et la façon dont ils sont conçus pour accomplir la scalabilité. Nous classons ces systèmes dans les catalogues présentés dans le chapitre 2. Dans le chapitre 4, nous étudions en profondeur BlobSeer, une service de gestion de données à grande échelle puisque nous utilisons BlobSeer comme système de référence tout au long de ce manuscrit.

Partie II: Systèmes de stockage scalables, repartis pour HPC de données à forte intensité

Cette partie se compose de 3 chapitres. Dans le chapitre 5, nous présentons quelques pratiques communes pour les opérations d'E/S dans l'HPC de données à forte intensité. Nous affirmons que l'HPC de données à forte intensité possède Big Data, et nous soulignons certains défis dans la conception des systèmes de stockage scalables dans un tel environnement. Nous continuons dans le chapitre 6 en présentant notre première contribution: la conception et la mise en œuvre d'un système de stockage scalable pour fournir un support efficace pour l'atomicité MPI-I/O. Cette partie se termine par le chapitre 7, où nous présentons notre contribution seconde dans le contexte de HPC de données à forte intensité. Ce chapitre consiste de la conception, l'architecture et les évaluations de Pyramid: un système de stockage à grande échelle, orienté matrice qui est optimisé pour le traitement des matrices en parallèle.

Partie III: Systèmes de stockage scalables, distribués géographiquement

Nous présentons notre contribution à la création d'un système de stockage scalable dans les environnements répartis géographiquement. Chapitre 8 introduit notre motivation de prendre BlobSeer comme un bloc de construction pour la construction d'un système de fichiers distribué globalement. Nous discutons la façon dont nous re-architecturons BlobSeer pour s'adapter à l'échelle WAN, et ensuite nous se concentrons sur les changements introduits dans la mise en œuvre d'une extension de BlobSeer: BlobSeer-WAN. Le chapitre se termine par une série d'expériences qui évaluent la performance du système par rapport à celle de la BlobSeer.

Partie IV: Un système de stockage orienté documents

Dans cette partie, nous discutons de notre quatrième contribution à la conception d'un système de gestion des données scalable dans les environnements centralisés. Concrètement, nous présentons DStore: un stockage orienté documents qui tire parti de la grande mémoire vide, multi-cœur pour se mettre à l'échelle verticalement. Nous donnons une motivation claire pour la conception et une description de l'architecture du système. l'évaluation du travail est ensuite présenté à la fin du chapitre.

Partie V: Conclusions et perspectives

Cette partie est constituée du chapitre 10. Nous résumons les contributions de cette thèse, discutons des limites et une série de perspectives pour des futures explorations.