BlobSeer: Next Generation Data Management for Large Scale Infrastructures

Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, Alexandra Carpen-Amarie

BlobSeer: Next Generation Data Management for Large Scale Infrastructures

Bogdan Nicolae[*], Gabriel Antoniu[†], Luc Bougé[‡],
Diana Moise[†], Alexandra Carpen-Amarie[†]

August 24, 2010

Abstract

As data volumes increase at a high speed in more and more application fields of science, engineering, information services, etc., the challenges posed by data-intensive computing gain an increasing importance. The emergence of highly scalable infrastructures, e.g. for cloud computing and for petascale computing and beyond introduces additional issues for which scalable data management becomes an immediate need. This paper brings several contributions. First, it proposes a set of principles for designing highly scalable distributed storage systems that are optimized for heavy data access concurrency. In particular, we highlight the potentially large benefits of using versioning in this context. Second, based on these principles, we propose a set of versioning algorithms, both for data and metadata, that enable a high throughput under concurrency. Finally, we implement and evaluate these algorithms in the BlobSeer prototype, that we integrate as a storage backend in the Hadoop MapReduce framework. We perform extensive microbenchmarks as well as experiments with real MapReduce applications: they demonstrate that applying the principles defended in our approach brings substantial benefits to data intensive applications.

Keywords: data intensive applications; data management; high throughput; versioning; concurrency; decentralized metadata management; BlobSeer; MapReduce

1 Introduction

More and more applications today generate and handle very large volumes of data on a regular basis. Such applications are called data-intensive. Governmental and commercial statistics, climate modeling, cosmology, genetics, bio-informatics, high-energy physics are just a few examples of fields where it becomes crucial to efficiently manipulate massive data, which are typically *shared* at large scale.

With the emergence of the recent infrastructures (cloud computing platforms, petascale architectures), achieving highly scalable data management is a critical challenge, as the overall application performance is highly dependent on the properties of the data management service.

[*]University of Rennes 1, IRISA.
[†]INRIA Rennes-Bretagne Atlantique, IRISA.
[‡]ENS Cachan Brittany Campus, IRISA.

For example, on *clouds* [36], computing resources are exploited on a per-need basis: instead of buying and managing hardware, users rent virtual machines and storage space. One important issue is thus the support for storing and processing data on externalized, virtual storage resources. Such needs require simultaneous investigation of important aspects related to performance, scalability, security and quality of service. Moreover, the impact of physical resource sharing also needs careful consideration.

In parallel with the emergence of cloud infrastructures, considerable efforts are now under way to build *petascale computing systems*, such as Blue Waters [40]. With processing power reaching one petaflop, such systems allow tackling much larger and more complex research challenges across a wide spectrum of science and engineering applications. On such infrastructures, data management is again a critical issue with a high impact on the application performance. Petascale supercomputers exhibit specific architectural features (e.g. a multi-level memory hierarchy scalable to tens to hundreds of thousands of cores) that are specifically designed to support a high degree of parallelism. In order to keep up with such advances, the storage service has to scale accordingly, which is clearly challenging.

Executing data-intensive applications at a very large scale raises the need to efficiently address several important aspects:

Scalable architecture. The data storage and management infrastructure needs to be able to efficiently leverage a large number of storage resources which are amassed and continuously added in huge data centers or petascale supercomputers. Orders of tens of thousands of nodes are common.

Massive unstructured data. Most of the data in circulation is unstructured: pictures, sound, movies, documents, experimental measurements. All these mix to build the input of applications and grow to huge sizes, with more than 1 TB of data gathered per week in a common scenario [24]. Such data is typically stored as huge objects and continuously updated by running applications. Traditional databases or file systems can hardly cope with objects that grow to huge sizes efficiently.

Transparency. When considering the major approaches to data management on large-scale distributed infrastructures today (e.g. on most computing grids), we can notice that most of them heavily rely on *explicit* data localization and on *explicit* transfers. Managing huge amounts of data in an explicit way at a very large scale makes the usage of the data management system complex and tedious. One issue to be addressed in the first place is therefore the *transparency* with respect to data localization and data movements. The data storage system should automatically handle these aspects and thus substantially simplify users access to data.

High throughput under heavy access concurrency. Several methods to process huge amounts of data have been established. Traditionally, message passing (with the most popular standard being MPI [9]) has been the choice of designing parallel and distributed applications. While this approach enables optimal exploitation of parallelism in the application, it requires the user to explicitly distribute work, perform data transfers and manage synchronization. Recent proposals such as MapReduce [6] and Dryad [15] try to address this by forcing the user to adhere to a specific paradigm. While this is not always possible, it has the advantage that once the application is cast in the framework, the details of scheduling, distribution and data transfers can be handled automatically. Regardless of the approach, in the context of data-intensive applications this translates to

massively parallel data access, that has to be handled efficiently by the underlying storage service. Since data-intensive applications spend considerable time to perform I/O, a high throughput in spite of heavy access concurrency is an important property that impacts on the total application execution time.

Support for highly parallel data workflows. Many data-intensive applications consist of multiple phases where data acquisition interleaves with data processing, generating highly parallel data workflows. Synchronizing access to the data under these circumstances is a difficult problem. Scalable ways of addressing this issue at the level of the storage service are thus highly desirable.

1.1 Contribution

This paper brings three main contributions. First, we propose and discuss a set of design principles for distributed storage systems: if combined together, these principles can help designers of distributed storage systems meet the aforementioned major requirements for highly scalable data management. In particular, we detail the potentially large benefits of using versioning to improve application data access performance under heavy concurrency. In this context we introduce a versioning-based data access interface that enables exploiting the inherent parallelism of data workflows efficiently.

Second, we propose a generalization for a set of versioning algorithms for data management we initially introduced in [21, 22]. We have introduced new data structures and redesigned several aspects to account for better decentralized management, asynchrony, fault tolerance and last but not least allow the user to explicitly control written data layout such that it is optimally distributed for reading.

Finally, we implement and evaluate our approach through the BlobSeer [20] prototype. We performed a series of synthetic benchmarks that push the system to its limits and show a high throughput under heavy access concurrency, even for small granularity and under faults. Moreover, we evaluate BlobSeer as a BLOB-based underlying storage layer for the Hadoop MapReduce framework. The work on integrating BlobSeer with Hadoop, presented in [23] has been used to evaluate the improvements of BlobSeer in the context of MapReduce data-intensive applications. We performed extensive experimentations that demonstrate substantial benefit of applying our principles in this context.

2 Related work

Data management for large-scale distributed architectures has been an active research area in the recent years, especially in the context of grid computing. Grids have typically been built by aggregating distributed resources from multiple administration domains, with the goal of providing support for applications with high demands in terms of computational resources and data storage. Most grid data management systems heavily rely on *explicit* data localization and on *explicit* transfers of large amounts of data across the distributed architecture: GridFTP [1], Raptor [17], Optor [17], are all representative of this approach. Managing huge amounts of data in such an explicit way at a very large scale makes the design of grid applications complex.

Therefore, *transparency* with respect to data localization and data movements appears as highly suitable, as it liberates the user from the burden of handling these aspects explicitly. *Distributed*

file systems [35] acknowledge the importance of a transparent data-access model. They provide a familiar, file-oriented API allowing to transparently access physically-distributed data through globally unique, logical file paths. A very large distributed storage space is thus made available to applications that usually use file storage, with no need for modifications. This approach has been taken by a few projects like GFarm [34], GridNFS [13], GPFS [29], XtreemFS [14], etc. Implementing transparent access at a global scale naturally leads however 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 accesses.

While recent work tries to alleviate this with the introduction of OSDs (*Object-based Storage Devices*) and distributed file systems relying on them [30, 37, 4], the needs of data-intensive applications have still not been fully addressed, as such systems still offer a POSIX-compliant access interface that in many cases limits the potential for concurrent access.

Therefore, specialized file systems have been introduced that specifically target the needs of data-intensive applications. The Google implementation of the MapReduce paradigm relies on GoogleFS [10] as the underlying storage layer. This inspired several other distributed file systems, such as HDFS [31], the standard storage layer used by Hadoop MapReduce. Such systems essentially keep the data immutable, while allowing concurrent appends. While this enables exploiting data workflow parallelism to a certain degree, our approach introduces several optimizations among which metadata decentralization and versioning-based concurrency control that enable writes at arbitrary offsets, effectively pushing the limits of exploiting parallelism at data-level even further.

With the emergence of cloud computing, storage solutions specifically designed to fit in this context appeared. Amazon S3 [39] is such a storage service that provides a simple web service interface to write, read, and delete an unlimited number of objects of modest sizes (at most 5 Gigabytes). Its simple design features high scalability and reliability. Recently, versioning support was also added. BlobSeer however introduces several advanced features such as huge objects and fine-grain concurrent reads and writes, as opposed to S3, which supports full object reads and writes only.

Closer to BlobSeer, versioning systems focus on explicit management of multiple versions of the same object. Several versioning file system proposals exist, such as Elephant [28] and ZFS [11]. While they offer a series of advanced versioning features, they are not natively distributed file systems and cannot provide concurrent access from multiple clients, therefore being restricted to act as local file systems. Amazon introduced Dynamo [7], a highly available key-value storage system that introduces some advanced features, among which explicit versioning support. Unlike Dynamo, BlobSeer treats concurrent writes to the same object as atomic, which enables the object to evolve in a linear fashion. This effectively avoids the scenario when an object can evolve in divergent directions, as is the case with Dynamo.

3 Distributed storage for data-intensive applications: main design principles

This section describes a series of key design principles that enable a distributed storage service for data-intensive applications to efficiently address the requirements mentioned in Section 1.

3.1 Why BLOBs?

We are considering applications that process huge amounts of data that are distributed at very large scale. To facilitate data management in such conditions, a suitable approach is to organize data as a set of *huge objects*. Such objects (called BLOBs hereafter, for *Binary Large OBjects*), consist of long sequences of bytes representing unstructured data and may serve as a basis for transparent data sharing at large-scale. A BLOB can typically reach sizes of up to 1 TB. Using a BLOB to represent data has two main advantages:

Scalability. Applications that deal with fast growing datasets that easily reach the order of TB and beyond can scale better, because maintaining a small set of huge BLOBs comprising billions of small, KB-sized application-level objects is much more feasible than managing billions of small KB-sized files directly. Even if there was a distributed file system that would support access to such small files transparently and efficiently, the simple mapping of application-level objects to file names can incur a prohibitively high overhead compared to the solution where the objects are stored in the same BLOB and only their offsets need to be maintained.

Transparency. A data-management system relying on globally shared BLOBs uniquely identified in the system through global ids facilitate easier application development by freeing the developer from the burden of managing data locations and data transfers explicitly in the application.

However, these advantages would be of little use unless the system provides support for *efficient fine-grain access* to the BLOBs. Actually, large-scale distributed applications typically aim at exploiting the inherent data-level parallelism and as such they usually consist of concurrent processes that access and process small parts of the same BLOB in parallel. Therefore, the storage service needs to provide an efficient and scalable support for a large number of concurrent processes that access (i.e., read or update) small parts of the BLOB, which in some cases can go as low as in the order of KB. If exploited efficiently, this feature introduces a very high degree of parallelism in the application.

3.2 Which interface to use for BLOB management?

At the basic level, the BLOB access interface must enable users to create a BLOB, to read/write a subsequence of *size* bytes from/to the BLOB starting at *offset* and to append a sequence of *size* bytes to the end of the BLOB.

However, considering the requirements with respect to data access concurrency, we claim that the BLOB access interface should provide *asynchronous* management operations. Also, it should enable to access past BLOB *versions*. Finally, it should guarantee *atomic generation* of these snapshots each time the BLOB gets updated. Each of these aspects is discussed below. Note that such an interface can either be directly used by applications to fully exploit the above features, or be leveraged in order to build higher-level data-management abstractions (e.g., a scalable BLOB-based, concurrency-optimized distributed file system).

Explicit versioning. Many data-intensive applications overlap data acquisition and processing, generating highly parallel data workflows. Versioning enables efficient management of such workflows: while data acquisition can run and lead to the generation of new BLOB snapshots,

data processing may continue undisturbed on its own snapshot which is immutable and thus never leads to potential inconsistencies. This can be achieved by exposing a versioning-based data access interface which enables the user to directly express such workflow patterns *without the need to manage synchronization explicitly.*

Atomic snapshot generation. A key property in this context is *atomicity.* Readers should not be able to access transiently inconsistent snapshots that are in the process of being generated. This greatly simplifies application development, as it reduces the need for complex synchronization schemes at application level.

Asynchronous operations. Input and output (I/O) operations can be extremely slow compared to the actual data processing. In the case of data-intensive applications where I/O operations are frequent and involve huge amounts of data, an asynchronous access interface to data is crucial, because it enables overlapping the computation with the I/O operations, thus avoiding the waste of processing power.

3.3 Versioning-based concurrency control

Providing the user with a versioning-oriented access interface for manipulating BLOBs favors application-level parallelism as an older version can be read while a newer version is generated. However, even if the versions are not *explicitly* manipulated by the applications, versioning may still be leveraged internally by the concurrency control to *implicitly* favor parallelism. For instance, it allows concurrent reads and writes to the same BLOB: if, as proposed above, updating data essentially consists in generating a new snapshot, reads will never conflict with concurrent writes, as they do not access the same BLOB version. In the snapshot-based approach that we defend, leveraging versioning is an efficient way to meet the major goal of maximizing concurrency. We therefore propose the following design principle: *data and metadata is always created and never overwritten.* This enables to parallelize concurrent accesses as much as possible, both at data and metadata level, for all possible combinations: concurrent reads, concurrent writes, and concurrent reads and writes.

The counterpart is of course that the required storage space needed to generate a new snapshot can easily explode to large sizes. However, although each write or append generates a new BLOB snapshot version, only the differential update with respect to the previous versions need to be physically stored. This eliminates unnecessary duplication of data and metadata and saves storage space. Moreover, as shown in section 4.2, it is possible to enable the new snapshot to share all unmodified *data* and *metadata* with the previous versions, while still creating the illusion of an independent, self-contained snapshot from the client's point of view.

3.4 Data striping

Data striping is a well-known technique to increase the performance of data access. Each BLOB is split into chunks that are distributed all over the machines that provide storage space. Thereby, data access requests can be served by multiple machines in parallel, which enables reaching high performance. Two factors need to be taken into consideration in order to maximize the benefits of accessing data in a distributed fashion.

Configurable chunk distribution strategy. The chunk distribution strategy specifies where to store the chunks in order to reach a predefined objective. Most of the time, load-balancing is

highly desired, because it enables a high throughput when different parts of the BLOB are simultaneously accessed. In a more complex scenario such as green computing, an additional objective for example would be to minimize energy consumption by reducing the number of storage space providers. Given this context, an important feature is to enable the application to configure the chunk distribution strategy such that it optimizes its own desired objectives.

Dynamically adjustable chunk sizes. The performance of distributed data processing is highly dependent on the way the computation is partitioned and scheduled in the system. There is a trade-off between splitting the computation into smaller work units in order to parallelize data processing, and paying for the cost of scheduling, initializing and synchronizing these work units in a distributed fashion. Since most of these work units consist in accessing data chunks, adapting the chunk size is crucial. If the chunk size is too small, then work units need to fetch data from multiple chunks, potentially making techniques such as scheduling the computation close to the data inefficient. On the other hand, selecting a chunk size that is too large may force multiple work units to access the same data chunks simultaneously, which limits the benefits of data distribution. Therefore, it is highly desirable to enable the application to fine-tune how data is split into chunks and distributed at large scale.

3.5 Distributed metadata management

Since each massive BLOB is striped over a large number of storage space providers, additional metadata is needed to map subsequences of the BLOB defined by *offset* and *size* to the corresponding chunks. The need for this comes from the fact that otherwise, the information of which chunk corresponds to which offset in the BLOB is lost. While this extra information seems negligible compared to the size of the data itself, at large scales it accounts for a large overhead. As pointed out in [32], traditional centralized metadata management approaches reach their limits. Therefore, we argue for a distributed metadata management scheme, which brings several advantages:

Scalability. A distributed metadata management scheme potentially scales better both with respect to the number of concurrent metadata accesses and to an increasing metadata size, as the I/O workload associated to metadata overhead can be distributed among the metadata providers, such that concurrent accesses to metadata on different providers can be served in parallel. This is important in order to support efficient fine-grain access to the BLOBs.

Data availability. A distributed metadata management scheme enhances data availability, as metadata can be replicated and distributed to multiple metadata providers. This avoids letting a centralized metadata server act as a single point of failure: the failure of a particular node storing metadata does not make the whole data unavailable.

4 Leveraging versioning as a key to support concurrency

4.1 A concurrency-oriented version-based interface

In Section 3.2 we have argued in favor of a BLOB access interface that needs to be *asynchronous*, *versioning-based* and which must guarantee *atomic generation* of new snapshots each time the BLOB gets updated.

To meet these properties, we propose the following series of primitives. To enable asynchrony, control is returned to the application immediately after the invocation of primitives, rather than waiting for the operations initiated by primitives to complete. When the operation completes, a callback function, supplied as parameter to the primitive, is called with the result of the operation as its parameters. It is in the callback function where the calling application takes the appropriate actions based on the result of the operation.

```
CREATE(callback(id))
```

This primitive creates a new empty BLOB of size 0. The BLOB will be identified by its id, which is guaranteed to be globally unique. The callback function receives this id as its only parameter.

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

The client may update the BLOB by invoking the corresponding WRITE or APPEND primitive. The initiated operation copies size bytes from a local buffer into the BLOB identified by id, either at the specified offset (in case of write), or at the end of the BLOB (in case of append). Each time the BLOB is updated by invoking write or append, a new snapshot reflecting the changes and labeled with an incremental version number is generated. The semantics of the write or append primitive is to submit the update to the system and let the system decide when to generate the new snapshot. The actual version assigned to the new snapshot is not known by the client immediately: it becomes available only at the moment when the operation completes. This completion results in the invocation of the callback function, which is supplied by the system with the assigned version v as a parameter.

The following guarantees are associated to the above primitives:

Liveness. For each successful write or append operation, the corresponding snapshot is eventually generated in a finite amount of time.

Total version ordering. If the write or append primitive is successful and returns version number v, then the snapshot labeled with v reflects the successive application of all updates numbered $1 \ldots v$ on the initially empty snapshot (conventionally labeled with version number 0), in this precise order.

Atomicity. Each snapshot appears to be generated instantaneously at some point between the invocation of the write or append primitive and the moment it is revealed to the readers by the system.

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

The READ primitive is invoked to read from a specified version of a BLOB. This primitive results in replacing the contents of the local buffer with size bytes from the snapshot version v of BLOB id, starting at offset, if v has already been generated. The callback function receives a single

parameter, result, a Boolean value that indicates whether the read succeeded or failed. If v has not been generated yet, the read fails and result is false. A read fails also if the total size of the snapshot v is smaller than offset + size.

Note that there must be a way to learn about both the available snapshots in the system and about their sizes, in order to be able to specify meaningful version values for the input parameters v, offset and size. This is the purpose of the following ancillary primitives.

```
GET_RECENT(id, callback(v, size))
GET_SIZE(id, v, callback(size))
```

The GET_RECENT primitive queries the system for a recent snapshot version of the blob id. The result of the query is the version number v which is passed to the *callback* function and the size of the associated snapshot. A positive value for v indicates success, while any negative value indicates failure. The system guarantees that: (1) $v \geq max(v_k)$, for all snapshot versions v_k that were successfully generated before the call is processed by the system; and (2) all snapshot versions with number lower or equal to v have successfully been generated as well and are available for reading. Note that this primitive is only intended to provide the caller with information about recent versions available: it does not involve strict synchronizations and does not block the concurrent creation of new snapshots.

The GET_SIZE primitive is used to find out the total size of the snapshot version v for BLOB id. This size is passed to the *callback* function once the operation has successfully completed.

Most of the time, the GET_RECENT primitive is sufficient to learn about new snapshot versions that are generated in the system. However, some scenarios require the application to react to updates as soon as possible after they happen. In order to avoid polling for new snapshot versions, two additional primitives are available to subscribe (and unsubscribe) to notifications for snapshot generation events.

```
SUBSCRIBE(id, callback(v, size))
UNSUBSCRIBE(id)
```

Invoking the SUBSCRIBE primitive registers the interest of a process to receive a notification each time a new snapshot of the BLOB id is generated. The notification is performed by calling the callback function with two parameters: the snapshot version v of the newly generated snapshot and its total size. The same guarantees are offered for the version as with the GET_RECENT primitive. Invoking the UNSUBSCRIBE terminates the notifications about new snapshot versions for a given BLOB id.

4.2 Optimized versioning-based concurrency control

As argued in the previous section, versioning can efficiently be leveraged to enhance concurrency. In this subsection we discuss this issue in more detail.

Obviously, concurrent readers never interfere with each other: they never modify any data or metadata and cannot generate inconsistent states. Therefore, concurrent reads are fully allowed. Also, a reader can only read a snapshot whose version number has been learned by the callback of a write primitive, or a notification. It is then guaranteed that the snapshot generation has

completed and that the corresponding data and metadata will never be modified again. Therefore, no synchronization is necessary and concurrent reads can freely proceed in the presence of writes.

The case of concurrent writers requires closer consideration. Updating a BLOB by means of a write or append involves two steps: first, writing the data, and second, writing the metadata. We specifically chose this order for two reasons: first, to favor parallelism; second, to reduce the risk for generating inconsistent metadata in case of failures. Writing the data in a first phase and metadata in a second phase enables full parallelism for the first phase. Concurrent writers submit their respective updates to the system and let the system decide about the update ordering. As far as this first phase is concerned, concurrent writers do not interfere with each other. They can write their chunks onto data storage providers in a fully parallel fashion. Besides, if a writer fails, no inconsistency may be created, since no metadata has been created yet.

It is at metadata level where the newly written chunks are integrated in a new snapshot. It is done by generating new metadata that reference both the new chunks and the metadata of the previous snapshot versions in such way as to offer the illusion of an independent, self-contained snapshot. At a first glance, writing the metadata might not seem parallelizable. First, once the data is written, a version number must be assigned to the writer. We make the assumption that version numbers are assigned in an incremental fashion: this very small step does require global synchronization and has to be serial. Second, since total ordering is now guaranteed, generating metadata for a snapshot that was assigned version number v relies on the metadata of snapshots with lower version numbers. It may seem that the metadata of snapshots versions $1 \dots v-1$ should be generated first before generating the metadata for snapshot version v. It is this waiting that we want to eliminate in order to further enhance the degree of concurrency in the process of writing data.

The key idea to do it is to structure metadata in such way as to support *metadata forward references*. More precisely, given a snapshot version k that was successfully generated and a set of concurrent writers that were assigned versions $k+1 \dots v$, the process of generating the metadata for snapshot version v must be able to precalculate all potential references to metadata belonging to versions $k+1 \dots v-1$ *even though the metadata of these versions have not been generated yet*, under the assumption that they will eventually be generated in the future. Considering this condition satisfied, metadata can be written in a parallel fashion as each writer can precalculate its metadata forward references individually if necessary.

However, consistency is guaranteed only if a new snapshot version v is considered as successfully generated after the metadata of all snapshots with a lower version are successfully written. This is so in order to ensure that all potential metadata forward references have been solved. Thus snapshot generation is an extremely cheap operation, as it simply involves delaying the revealing of snapshot version v to readers until the metadata of all lower snapshot versions has been written.

Avoiding synchronization between concurrent accesses both at data and metadata level unlocks the potential to access data in a highly parallel fashion. This approach is combined with data striping and metadata decentralization, so that concurrent accesses are physically distributed at large scale among nodes. This combination is crucial in achieving a high throughput under heavy access concurrency.

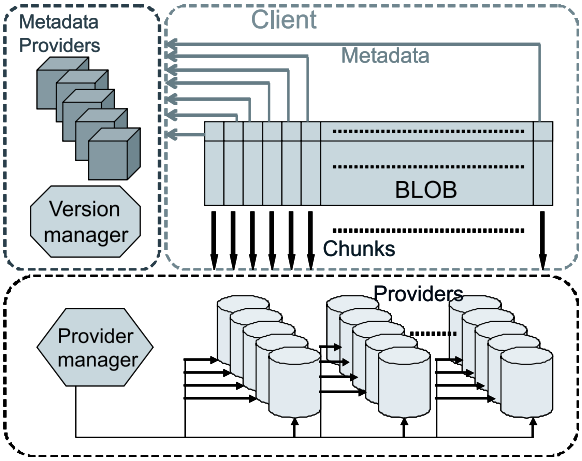


Figure 1: Architecture

5 BlobSeer

To illustrate and evaluate the design principles discussed in Section 3, we have implemented the BlobSeer BLOB management service. This section introduces its architecture, explains how the versioning-oriented interface has been implemented, then finally zooms on a few delicate aspects.

5.1 Architecture

BlobSeer consists of a series of distributed communicating processes. Figure 1 illustrates the processes and the interactions between them.

Clients create, read, write and append data from/to BLOBs. A large number of concurrent clients is expected, and they may all access the same BLOB.

Data (storage) providers physically store the chunks generated by appends and writes. New data providers may dynamically join and leave the system.

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.

Metadata (storage) providers physically store the metadata that allows identifying the chunks that make up a snapshot version. A distributed metadata management scheme is employed to enhance concurrent access to metadata.

The version manager is in charge of assigning new snapshot version numbers to writers and appenders and to reveal these new snapshots to readers. It is done so as to offer the illusion of instant snapshot generation and to satisfy the guarantees listed in Section 4.1.

5.2 Core algorithms

In this section we present a series of data structures and algorithms that enable efficient implementation of primitives presented in Section 4.1. In order to simplify the notation, we omit the BLOB

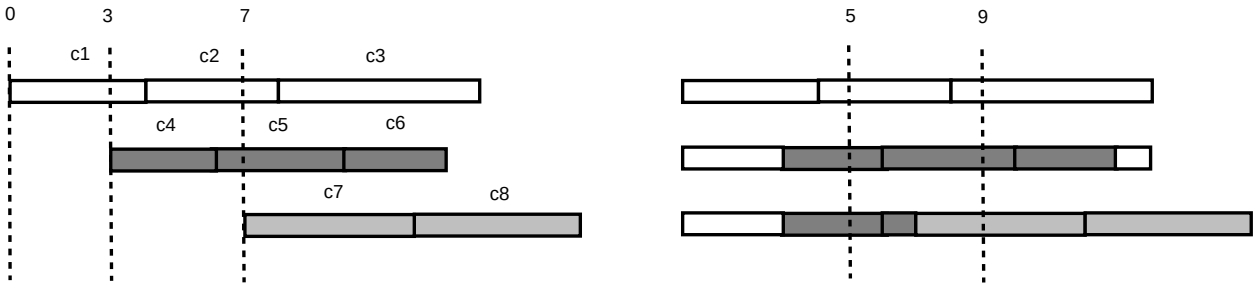


Figure 2: Evolution of the blob composition (right) after three successive writes (left)

id from all our algorithms. This does not restrict the general case in any way, as all data structures presented can be instantiated and maintained for each BLOB *id* independently.

5.2.1 Data structures

Each write or append updates the BLOB by writing a set of new chunks in the system, which are then used to generate a new snapshot version that reflects the changes. Thus, an arbitrary subsequence R of a snapshot version v, delimited by *offset* and *size* consists of the list of all parts of the chunks which overlap with R and which were written at the most recent version smaller or equal to v.

For example, Figure 2 depicts the snapshots that are generated (right side) by three consecutive updates (left side). The first update (white) is an append of 14 bytes in three chunks: $c1$, $c2$, $c3$. The second update (dark gray) is a write of 10 bytes in three chunks, $c4$, $c5$ and $c6$, starting at offset 3. Finally the last update (light gray) is a write of 10 bytes starting at offset 7 in two chunks: $c7$ and $c8$. Considering the snapshot that is generated after applying all three updates ($v = 3$), if we fix the subsequence R to $offset = 5$ and $size = 4$, the list of parts of chunks that make up R is the last byte of $c4$, the first byte of $c5$ and the first two bytes of $c7$.

Descriptor map. To R we associate the *descriptor map* D that describes the list of all parts of chunks that make up R. Each entry in D is a *chunk descriptor* of the form *(cid, coffset, csize, roffset)*, where *cid* is the id that uniquely identifies the chunk in the system; *coffset* and *csize* delimit the part of the chunk relative to the whole chunk; and *roffset* is the offset of the part of the chunk relative to R.

Considering the same example as mentioned above, the associated descriptor map contains the following chunk descriptors: $(c4, 2, 1, 0), (c5, 0, 1, 1), (c7, 0, 2, 2)$.

Such descriptor maps are shared among the processes. For this reason we assume a globally shared container D_{global} which enables storing and retrieving descriptor maps. Each descriptor map is identified by a globally unique id. Let i_D be the id of the descriptor map D. We denote the store operation by $D_{global} \leftarrow D_{global} \cup (i_D, D)$ and the retrieve operation by $D \leftarrow D_{global}[i_D]$.

Provider map. A similar container P_{global}, which holds entries of the form $(cid, address)$, is used to store the $address$ of the data provider which holds the chunk identified by *cid*.

History of writes. We define H_{global} to be the history of all writes in the system that were assigned a snapshot version, regardless whether their snapshot was generated or is in the course of being generated. Entries in this history are of the form $(v, (t, o, s, i))$, where v is the snapshot version assigned to the write, t, the total size of the snapshot, o, the offset of the update in the snapshot, s, the size of the update and i, the identifier of the descriptor map associated to the update. H_{global} enables global sharing of the entries, in a manner similar to D_{global}, but we are interested in retrieving a whole subsequence of entries $H_{global}[v_a \dots v_b]$ whose versions lie between v_a and v_b.

Again, we use the same example depicted in Figure 2. Assuming white was assigned version 1, dark gray version 2 and light gray version 3, the corresponding descriptor maps are $D_1 = \{(c1, 0, 4, 0), (c2, 0, 4, 4), (c3, 0, 6, 8)\}, D_2 = \{(c4, 0, 3, 3), (c5, 0, 4, 6), (c6, 0, 3, 10)\}$ and $D_3 = \{(c7, 0, 5, 7), (c8, 0, 5, 12)\}$. The associated globally unique identifiers of the descriptor maps are i_1, i_2 and i_3. Finally, the history of all writes contains the following entries: $(1, (14, 0, 14, i_1))$, $(2, (14, 3, 10, i_2))$, $(3, (17, 7, 10, i_3))$.

These globally shared containers can be organized as distributed data structures specifically designed to serve exact queries and range queries under heavy access concurrency. Extensive work in the literature [33, 25, 5] covers these aspects.

5.2.2 Reading

The version manager maintains the most recent snapshot version that was successfully generated. This version is denoted v_g and is the version obtained when calling the GET_RECENT primitive.

The SUBSCRIBE and UNSUBSCRIBE primitives, which register/unregister the client's interest to receive notifications about newly generated snapshots in the system, rely on publish-subscribe mechanisms. Such mechanisms are widely covered in the literature [8, 2]. In this case, the version manager acts as the publisher of v_g each time a new snapshot version is generated.

Using the primitives presented above, the client can obtain information about snapshots that are available in the system and read any subsequence of any snapshot, by calling the READ primitive. Algorithm 1 presents this primitive, which consists of the following steps:

1. The client first asks the version manager for the latest snapshot version that was successfully generated. If the requested version is higher than this version or if the requested offset and size overflow the total size of the snapshot ($offset + size > t$), then READ is aborted and the supplied *callback* is invoked immediately to signal failure.

2. Otherwise, the client needs to find out what parts of chunks fully cover the requested subsequence delimited by *offset* and *size* from version v and where they are stored. To this purpose, the primitive GET_DESCRIPTORS, detailed in Section 5.3.2, builds the descriptor map of the requested subsequence.

3. Having calculated the descriptor map of the subsequence, the client proceeds to fetch the necessary parts of the chunks in parallel from the data providers into the locally supplied buffer. If any part could not be retrieved successfully, READ is aborted and the supplied *callback* is invoked immediately to signal failure to the user.

Once all these steps have completed successfully, the READ primitive invokes the *callback* to notify the client of success.

Algorithm 1 Read a subsequence of snapshot version v into the local buffer

```
1: procedure READ(v, buffer, offset, size, callback)
2:     v_g ← invoke remotely on version manager RECENT
3:     if v > v_g then
4:         invoke callback(false)
5:     end if
6:     (t, _, _, _) ← H_global[v]                    ▷ t gets the total size of the snapshot v
7:     if offset + size > t then
8:         invoke callback(false)
9:     end if
10:    D ← GET_DESCRIPTORS(v, t, offset, size)
11:    for all (cid, co, csize, ro) ∈ D in parallel do
12:        buffer[ro .. ro + csize] ← get chunk cid[co .. co + csize] from P_global[cid]
13:        if get operation failed then
14:            abort other get operations
15:            invoke callback(false)
16:        end if
17:    end for
18:    invoke callback(true)
19: end procedure
```

5.2.3 Writing and appending

The WRITE and APPEND primitives are described in Algorithm 2. The main idea behind can be summarized in the following steps:

1. Split the local buffer into $|K|$ chunks by using the partitioning function SPLIT (which can be configured by the user).
2. Get a list of $|K|$ data providers, one for each chunk, from the provider manager.
3. Write all chunks in parallel to their respective data providers and construct the corresponding descriptor map D relative to the *offset* of the update. Add an entry for each chunk to P_{global}.
4. Add the descriptor map D to D_{global}.
5. Ask the version manager to be assigned a new version number v_a, write the corresponding metadata and notify the version manager that the operation succeeded and it can generate the snapshot v_a at its discretion.

The last step requires some closer consideration. It is the responsibility of the version manager to manage the assigned versions, which is done by atomically incrementing a global variable v_a each time a new snapshot version is requested and adding all necessary information about v_a (the total size of snapshot version v_a, *offset*, *size*, i_D) to the history of all writes H_{global}. This process is detailed in Algorithm 3.

In the case of append, *offset* is not specified explicitly, but is implicitly the total size stored by $H_{global}[v_a - 1]$. As it is the only difference between writes and appends, we will refer to both simply as writes from now.

Algorithm 2 Write a the content of a local buffer into the blob

1: **procedure** WRITE(*buffer, offset, size, callback*)
2: $K \leftarrow$ SPLIT($size$)
3: $P \leftarrow$ **get** $|K|$ providers **from** provider manager
4: $D \leftarrow \emptyset$
5: **for all** $0 \leq i < |K|$ **in parallel do**
6: $cid \leftarrow$ uniquely generated chunk id
7: $roffset \leftarrow \sum_{j=0}^{i-1} K[j]$
8: store $buffer[roffset$ .. $roffset + K[i]]$ as chunk cid on provider $P[i]$
9: **if** store operation failed **then**
10: abort other store operations
11: **invoke** $callback(-1)$
12: **end if**
13: $D \leftarrow D \cup \{(cid, 0, K[i], roffset)\}$
14: $P_{global} \leftarrow P_{global} \cup \{(cid, P[i])\}$
15: **end for**
16: $i_D \leftarrow$ uniquely generated id
17: $D_{global} \leftarrow D_{global} \cup (i_D, D)$
18: $(v_a, v_g) \leftarrow$ **invoke remotely** on version manager ASSIGN_WRITE(*offset, size,* i_D)
19: BUILD_METADATA(v_a, v_g, D)
20: **invoke remotely** on version manager COMPLETE(v_a)
21: **invoke** $callback(v_a)$
22: **end procedure**

Algorithm 3 Assign a snapshot version to a write

1: **function** ASSIGN_WRITE(*offset, size,* i_D)
2: $(t_a, \_, \_, \_) \leftarrow H_{global}[v_a]$
3: **if** $offset + size > t_a$ **then**
4: $t_a \leftarrow offset + size$
5: **end if**
6: $v_a \leftarrow v_a + 1$
7: $H_{global} \leftarrow H_{global} \cup \{(v_a, (t_a, offset, size, i_D)\}$
8: $Pending \leftarrow Pending \cup \{v_a\}$
9: **return** (v_a, v_g)
10: **end function**

Both v_g and v_a are reported back to the client. Note that since each writer publishes D, it is possible to establish the composition of any snapshot version by simply going backwards in the history of writes H_{global} and analyzing corresponding descriptor maps $D_{global}[i]$. However, such an approach is unfeasible as it degrades read performance the more writes occur for the BLOB. As a consequence, more elaborate metadata structures need to be maintained.

The BUILD_METADATA function is responsible to generate such new metadata that both references the descriptor map D and the metadata of previous snapshot versions, such as to provide the illusion of a fully independent snapshot version v_a, yet keep the performance levels of querying metadata high, regardless of how many writes occurred in the system, that is, regardless of how large v_a is. Moreover, it has to support metadata forward references introduced in Section 3.3, in order to avoid having to wait for other concurrent writers to write their metadata first, thus enabling a high-throughput under write-intensive scenarios. These metadata management aspects are discussed in more detail in Section 5.3.

After the client finished writing the metadata, it notifies the version manager of success by invoking COMPLETE remotely on the version manager and finally invokes the *callback* with the assigned version v_a as its parameter. At this point, the write operation completed from the client's point of view and it is the responsibility of the version manager to generate snapshot v_a.

5.2.4 Generating new snapshot versions

The version manager has to generate new snapshots under the guarantees mentioned in Section 4.1: liveness, total ordering and atomicity.

In order to achieve this, the version manager stores all the notifications from clients and increments v_g as long as a notification for it exists. This is done by maintaining two sets: *Pending* and *Completed*. The *Pending* set holds all versions that were assigned but for which a notification of success has not been received, while the *Completed* set holds all versions for which a notification has been received but the corresponding snapshot version has not been generated yet (i.e. v_g is lower than all versions in *Completed*).

Each time a client requests a new version, v_a is incremented and added to the *Pending* set (Line 8 of Algorithm 3). Once a notification of completion for v_a is received, COMPLETE, presented in Algorithm 4 is executed by the version manager. It simply moves v_a from *Pending* into *Completed* and tries to increment v_g as long as $v_g + 1$ exists in *Completed*, removing it from *Completed* if it is the case.

Algorithm 4 Complete the write

```
1: procedure COMPLETE(v_a)
2:     Pending ← Pending \ {v_a}
3:     Completed ← Completed ∪ {v_a}
4:     while v_g + 1 ∈ Completed do
5:         v_g ← v_g + 1
6:         Completed ← Completed \ {v_g}
7:     end while
8: end procedure
```

This way, the guarantees mentioned in Section 4.1 are satisfied:

Total ordering implies that snapshot v_a cannot be generated unless the metadata of all snapshots labeled with a lower version have been fully constructed, which translates into the following condition: all writers that were assigned a version number $v_i < v_a$ notified the version manager of success. Since v_g is incremented only as long as the corresponding client notified the version manager of success, this property is satisfied.

The liveness condition is satisfied assuming that writers do not take forever to notify the version manager of success, because eventually v_g will be incremented to reach v_a. If a writer fails to perform the notification in a predefined amount of time, the responsibility of writing the metadata can be delegated to any other process, since both the chunks and their descriptor maps have been successfully written by the client before failing (otherwise the client would not have requested a snapshot version for its update in the first place).

Atomicity is satisfied because the only way a new snapshot version is revealed to the clients is by incrementing v_g, which from the client's point of view corresponds to an instant appearance of a fully independent snapshot in the system, without any exposure to inconsistent transitory states.

5.3 Metadata management

In the previous section we have introduced a globally shared data structure denoted H_{global} that holds the history of all updates performed on a BLOB. While this history is enough to establish the composition of any subsequence of any snapshot version by simply walking the history backwards from the desired version, such trivial approach is not feasible, because performance degrades the more updates are performed on the BLOB (and thus the history grows).

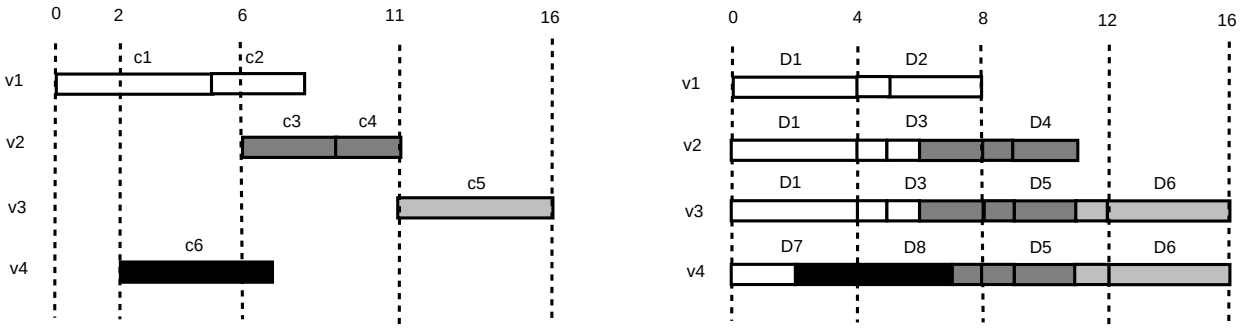
Balanced trees, in particular B-trees [27] are used by several file systems to maintain the composition of files and directories. B-trees are attractive because they offer guaranteed logarithmic-time key-search, insert, and remove. However, in order to maintain access performance, B-trees rely on rebalancing, which is incompatible with our requirement that metadata is *always added and never modified*. Recent work [26] aims at efficiently solving this issue, by addressing concurrency through lock-coupling [3]. For our distributed data structures, however, this approach implies distributed locking, which is a difficult problem that is best avoided. Moreover, locking does allow support for *metadata forward references*, which limits concurrent metadata access performance.

Thus, this section contributes with a set of metadata structures and algorithms oriented towards supporting *metadata forward references*, while still offering logarithmic access times.

5.3.1 Data structures

We organize metadata as a *distributed segment tree* [38], and associate one with each snapshot version of a given BLOB. A segment tree is a binary tree in which each node is associated with a subsequence of a given snapshot version v_i of the BLOB, delimited by *offset* and *size*. In order to simplify the notation, we denote *offset* by x_i and *offset* + *size* by y_i and refer to the subsequence delimited by *offset* and *size* as segment $[x_i, y_i]$. Thus, a node is uniquely identified by the pair $(v_i, [x_i, y_i])$. We refer to the association between the subsequence and the node simply as the node *covers* $(v_i, [x_i, y_i])$ (or even shorter *covers* $[x_i, y_i]$).

For each node that is not a leaf, the left child covers the left half of the parent's segment, and the right child covers the right half of the parent's segment. These two children are $(v_{li}, [x_i, (x_i + y_i)/2])$



(a) History of writes

(b) Descriptor maps of the leaf nodes

Figure 3: Four writes/appends (left) and the corresponding composition of the leaves when $lsize = 4$ (right)

and $(v_{ri}, [(x_i + y_i)/2, y_i])$. Note that the snapshot version of the left child v_{li} and right child v_{ri} must not necessarily be the same as v_i, enabling the segment trees for a given BLOB to share whole subtrees among each other. The root of the tree is associated with the minimal segment $[x_i, y_i]$ such that it covers the whole snapshot v_i. Considering t_i the total size of the snapshot v_i, the associated root covers $[0, r_i]$, where r_i is the smallest power of two greater than t_i.

A leaf node covers a segment whose length is $lsize$, a fixed size that is a power of two. Each leaf holds the descriptor map D_i, relative to x_i, that makes up the subsequence. In order to limit fragmentation, the descriptor map is restricted to hold not more that k chunk descriptors, which we call the *fragmentation threshold*.

Thus, nodes are fully represented by $(key, value)$ pairs, where $key = (v_i, [x_i, y_i])$ and $value = (v_{li}, v_{ri}, D_i)$ such that $D_i \neq \emptyset$ for leaves and $D_i = \emptyset$ for inner nodes.

Figure 3 depicts an example of four consecutive updates that generated four snapshots $v_1..v_4$, having version numbers from 1 to 4. The updates are represented in Figure 3(a): each written chunk is a rectangle, chunks from the same update have the same color, and the shift on the X-axis represents the absolute offset in the blob. v_1 and v_3 correspond to writes, while v_2 and v_4 correspond to appends.

This series of writes results in the snapshot composition presented in Figure 3(b). Each of these snapshots is associated with a segment tree, for which $lsize = 4$. The leaves of the segment trees thus cover the segments $[0, 4], [4, 8], [8, 12], [12, 16]$. There is a total of 8 distinct leaves, labeled $D_1..D_8$ after their associated descriptor maps, which are represented in the figure.

Finally, the full segment trees are depicted in Figure 4, where the inner nodes are labeled with the covered segment and their links to the left and right children are represented as arrows. Notice how entire subtrees are shared among segment trees: for example, the right child of the root of black $(v_4, [0, 16])$ is light gray $(v_3, [8, 16])$.

All tree nodes associated with the BLOB are stored in the globally shared container $Nodes_{global}$, which enables storing and retrieving $(key, value)$ pairs efficiently. We denote the store operation by $Nodes_{global} \leftarrow Nodes_{global} \cup (key, value)$ and the retrieve operation by $Nodes_{global} \leftarrow Nodes_{global}[key]$.

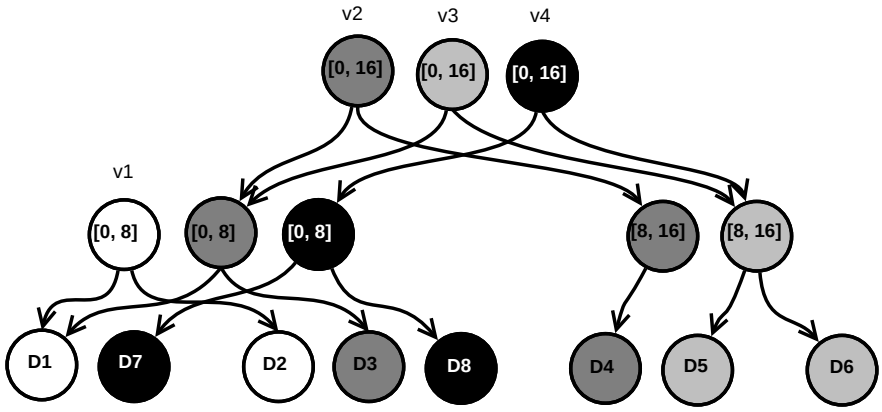


Figure 4: Segment tree: leaves are labeled with the descriptor maps, inner nodes with the segments they cover

5.3.2 Obtaining the descriptor map for a given subsequence

In order to read a subsequence R delimited by *offset* and *size* from the snapshot version v, the descriptor map D corresponding to R must first be determined, such that the corresponding data can be fetched from the data providers.

Algorithm 5 Get the descriptor map for a given subsequence

```
1:  function GET_DESCRIPTORS(v, offset, size)
2:      D ← ∅
3:      Q ←ROOT(v)
4:      while Q ≠ ∅ do
5:          ((v_i, [x_i, y_i]), (v_li, v_ri, D_i)) ← extract node from Q
6:          if D_i ≠ ∅ then
7:              D ← D∪ INTERSECT(D_i, x_i, [x_i, y_i] ∩ [offset, offset + size], offset)
8:          else
9:              if [offset, offset + size] ∩ [x_i, (x_i + y_i)/2] ≠ ∅ then
10:                 Q ← Q ∪ Nodes_global[(v_li, [x_i, (x_i + y_i)/2])]
11:             end if
12:             if [offset, offset + size] ∩ [(x_i + y_i)/2, y_i] ≠ ∅ then
13:                 Q ← Q ∪ Nodes_global[(v_ri, [(x_i + y_i)/2, y_i])]
14:             end if
15:         end if
16:     end while
17:     return D
18: end function
```

The GET_DESCRIPTORS function, presented in Algorithm 5, is responsible for constructing D. This is achieved by walking the segment tree of the snapshot in a top-down manner, starting from the root towards the leaves that intersect R. Once such a leaf is found, its chunk descriptors from D_i that are part of R are extracted and added to D.

The offsets in D_i, however, are relative to x_i, the left end of the segment covered by the leaf,

while the offsets in D need to be relative to *offset*. For this reason, the relative offsets of the extracted chunk descriptors need to shifted accordingly. This is performed by the INTERSECT function, presented in Algorithm 6.

Algorithm 6 Intersect a chunk map with a given segment

```
1: function INTERSECT(D_s, x_s, [x_i, y_i], x_d)
2:     D_d ← ∅
3:     ro_d ← max(x_s − x_d, 0)
4:     for all (cid_s, co_s, cs_s, ro_s) ∈ D_s such that [x_i, y_i] ∩ [x_s + ro_s, x_s + ro_s + cs_s] ≠ ∅ do
5:         co_d ← co_s + max(x_i − x_s − ro_s, 0)
6:         cs_d ← min(y_i − max(x_i, x_s + ro_s), cs_s)
7:         D_d ← D_d ∪ {(cid_s, co_d, so_d, ro_d)}
8:     end for
9:     return D_d
10: end function
```

INTERSECT has four arguments: the first two arguments determine the source descriptor map D_s where to extract the chunk descriptors from and the offset x_s to which its chunk descriptors are relative. The third argument is the segment $[x_i, y_i]$ that delimits R, and finally the fourth argument is the offset x_d to which the chunk descriptors of the resulting destination descriptor map D_d are relative.

In order to illustrate how INTERSECT works, let's take again the example presented in Figure 3. Assuming R is delimited by $x_i = 3$ and $y_i = 7$ and the source descriptor map is D_3, we have $x_s = 4$ and $x_d = 3$. All three chunk descriptors $(c_1, 4, 1, 0)$, $(c_2, 0, 1, 1)$ and $(c_3, 0, 2, 2)$ belonging to D_3 intersect R. $x_s - x_d = 1$, thus the relative offsets are incremented by 1. Moreover, only the first byte of c_3 is part of R, such that the chunk size of the new chunk descriptor gets adjusted accordingly: $\min(7 - \max(3, 4 + 2), 2) = 1$. Thus, in the end D_d contains the following chunk descriptors: $(c_1, 4, 1, 1)$, $(c_2, 0, 1, 2)$ and $(c_3, 0, 1, 3)$.

Once all chunk descriptors from all the leaves that intersect R have been extracted, adjusted to *offset* and added to D, GET_DESCRIPTORS returns D as the final result.

5.3.3 Building the metadata of new snapshots

Once the version manager assigned a new snapshot version v_a to the update, the segment tree construction can begin. The BUILD_METADATA procedure, described in Algorithm 7 is responsible for that, starting from the following parameters: v_a, the assigned snapshot version, v_g, the version of a recently generated snapshot and D, the descriptor map of the update corresponding to v_a.

The segment tree construction is performed in a bottom-up manner, starting from the leaves and working towards the root. It consists of two stages. In the first stage, the leaves corresponding to the update are build, while in the second stage the inner nodes and the root are build. Only the subtree whose inner nodes cover at least one of the new leaves is build. Any inner node which does not cover at least one of the new leaves is shared with the most recent segment tree assigned a lower version $v_i < v_a$, which has built that inner node. This way, whole subtrees are shared among versions without breaking total ordering.

Algorithm 7 Build the metadata for a given snapshot version

procedure BUILD_METADATA(v_a, v_g, D)
  $W \leftarrow H_{global}[v_g \dots v_a]$
  $(t_a, o_a, s_a, \_) \leftarrow W[v_a]$
  $x_i \leftarrow \lfloor o_a / lsize \rfloor \cdot lsize$ ▷ largest multiple of $lsize$ smaller or equal to o_a
  $Q \leftarrow \emptyset$
  while $x_i + lsize < o_a + s_a$ **do**
    $Q \leftarrow Q \cup \{((v_a, [x_i, x_i + lsize]), (0, 0, \mathsf{LEAF}(v_a, v_g, [x_i, x_i + lsize], D, W)))\}$
    $x_i \leftarrow x_i + lsize$
  end while
  $T \leftarrow \emptyset$
  while $Q \neq \emptyset$ **do**
    $((v_i, [x_i, y_i]), (v_{li}, v_{ri}, D_i)) \leftarrow$ extract node from Q
    $T \leftarrow T \cup ((v_i, [x_i, y_i]), (v_{li}, v_{ri}, D_i))$
    if $(v_i, [x_i, y_i]) \neq \mathsf{ROOT}(v_a)$ **then**
      if x_i **mod** $2 \cdot (y_i - x_i) = 0$ **then** ▷ $(v_i, [x_i, y_i])$ has a right sibling
        $(x_s, y_s) \leftarrow (y_i, 2 \cdot y_i - x_i)$
      else ▷ $(v_i, [x_i, y_i])$ has a left sibling
        $(x_s, y_s) \leftarrow (2 \cdot x_i - y_i, x_i)$
      end if
      if $\exists((v_i, [x_s, y_s]), (v_{lj}, v_{rj}, D_j)) \in Q$ **then** ▷ sibling is in Q, move it to T
        $v_s \leftarrow v_i$
        $Q \leftarrow Q \setminus \{((v_i, [x_s, y_s]), (v_{lj}, v_{rj}, D_j))\}$
        $T \leftarrow T \cup \{((v_i, [x_s, y_s]), (v_{lj}, v_{rj}, D_j))\}$
      else ▷ sibling is not in Q, it belongs to a lower version
        $(v_s, (t_s, o_s, s_s, \_)) \leftarrow (v_a - 1, W[v_a - 1])$
        while $v_s > v_g$ **and** $[o_s, o_s + s_s] \cap [x_i, y_i] = \emptyset$ **do**
          $v_s \leftarrow v_s - 1$
          $(t_s, o_s, s_s, \_) \leftarrow W[v_s]$
        end while
      end if
      if $x_i < x_s$ **then**
        $Q \leftarrow Q \cup \{((v_i, [x_i, y_s]), (v_i, v_s, \emptyset))\}$
      else
        $Q \leftarrow Q \cup \{((v_i, [x_s, y_i]), (v_s, v_i, \emptyset))\}$
      end if
    end if
  end while
  $Nodes_{global} \leftarrow Nodes_{global} \cup T$
end procedure

Building the leaves. First of all, the set of new leaves corresponding to the update of v_a must be built. A leaf corresponds to the update if its segment $[x_i, x_i + lsize]$ overlaps with the update, whose offset is o_a and size s_a. For each such leaf, the corresponding descriptor map D_i, relative to x_i must be calculated. This is performed by the LEAF function, presented in Algorithm 8.

Algorithm 8 Build the descriptor map for a given leaf

```
1: function LEAF(v_a, v_g, [x_i, y_i], D_a, W)
2:     (_, o_a, s_a, _) ← W[v_a]
3:     (t_g, o_g, _, _) ← W[v_g]
4:     D_i ← INTERSECT(D_a, o_a, [o_a, o_a + s_a] ∩ [x_i, y_i], x_i)
5:     v_j ← v_a − 1
6:     Reminder ← [x_i, y_i] \ [o_a, o_a + s_a]
7:     while v_j > v_g and Reminder ≠ ∅ do
8:         (t_j, o_j, s_j, i_j) ← W[v_j]
9:         for all [x_j, y_j] ∈ Reminder ∩ [o_j, o_j + s_j] do
10:            D_i ← D_i ∪ INTERSECT(D_global[i_j], o_j, [x_j, y_j], x_i)
11:            Reminder ← Reminder \ [x_j, y_j]
12:        end for
13:        v_j ← v_j − 1
14:    end while
15:    if x_i < t_g and Reminder ≠ ∅ then
16:        ((v_j, [x_j, y_j]), (v_lj, v_rj, D_j)) ← ROOT(v_g)
17:        while x_j ≠ x_i and y_j ≠ y_i do
18:            if x_j < x_i then
19:                x_j ← x_j + lsize
20:                (v_lj, v_rj, D_j) ← Nodes_global[(v_lj, [x_j, y_j])]
21:            else
22:                y_j ← y_j − lsize
23:                (v_lj, v_rj, D_j) ← Nodes_global[(v_rj, [x_j, y_j])]
24:            end if
25:        end while
26:        D_i ← D_i ∪ INTERSECT(D_j, x_i, Reminder ∩ [x_i, y_i], x_i)
27:    end if
28:    if |D_i| > k then
29:        D_i ← DEFRAGMENT(D_i)
30:    end if
31:    return D_i
32: end function
```

In order to obtain D_i, the LEAF function must extract the chunk descriptors of D that overlap with the segment of the leaf, adjusting their relative offsets from the original offset o_a of D to x_i, the offset of the leaf.

Since the leaf may not be fully covered by $[o_a, o_a + s_a]$, the remaining part of the leaf, denoted *Reminder*, may cover chunks belonging to snapshot versions lower than v_a. If the snapshot $v_a - 1$ has already been generated, that is, $v_g = v_a - 1$, extracting the chunk descriptors that fill *Reminder* can be performed directly from the descriptor map of the leaf that covers $[x_i, y_i]$ in the segment

tree of snapshot $v_a - 1$.

The case when $v_g < v_a - 1$ implies that the status of the metadata for all concurrent writes, which were assigned version v_j and for which $v_g < v_j < v_a$, is uncertain. Therefore, the segment tree of snapshot $v_a - 1$ cannot be relied upon to fill *Reminder*, as it may not have been generated yet.

However, since a write which was assigned a snapshot version v_j requested a version *after* it added to D_{global} its full descriptor map D_j, the *Reminder* of the leaf can be gradually filled by working backwards in the history of writes starting from $v_a - 1$ and extracting the chunk descriptors that overlap with the leaf, while adjusting the relative offset to x_i. This step might seem costly, but we need to observe that D_{global} has to be queried to obtain D_j *only if Reminder* intersects with the update of v_j, which in practice is rarely the case. Obviously, once v_g has been reached, the process stops, because the leaf of v_g can be consulted directly.

At this point, D_i has been successfully generated. It is however possible that overlapping updates have fragmented the segment $[x_i, y_i]$ heavily, such that D_i contains a lot of chunk descriptors. This is turn reduces access performance, because many small parts of chunks have to be fetched. For this reason, if the number of chunk descriptors goes beyond the fragmentation threshold, that is, $|D_i| > k$, a healing mechanism for the leaf is employed. More precisely, the DEFRAGMENT primitive, which is responsible for this task, reads the whole range $[x_i, y_i]$ and then applies the SPLIT primitive to obtain less then k chunks. These chunks are written on the storage space providers and D_i is reset to contain their corresponding chunk descriptors. This effectively reorganizes D_i to hold less than k chunk descriptors.

Building the inner nodes. After having generated the set of leaves for the snapshot v_a, the inner nodes of the segment tree are built in a bottom-up fashion, up towards the root. This is an iterative process: starting from the set of leaves Q, any two siblings for which at least one of them belongs to Q are combined to build the parent, which in turn is eventually combined, up to the point when the root itself is obtained and Q becomes empty.

In order to find two siblings that can be combined, an arbitrary tree node $((v_a, [x_i, y_i]), (v_{li}, v_{ri}, D_i))$, is extracted from Q. This tree node is either the left child of its parent and thus it has a right sibling or it is the right child of its parent and thus has a left sibling. In either case, the segment $[x_s, y_s]$ covered by the sibling can be easily calculated. Thus, the only missing information in order to completely determine the sibling is its version v_s. This version, however, needs closer attention. If the sibling itself belongs to Q, then $v_s = v_a$. Otherwise, v_s is the version assigned to the most recent writer for which $v_s < v_a$ and the corresponding update of v_s intersects with $[x_s, y_s]$, that is, the sibling was generated or is in the process of being generated by that writer. Once v_s is established, both versions of the children are available and and thus the parent can be generated.

Since the sibling may correspond to a lower snapshot version whose metadata is generated concurrently, it is unknown whether the sibling was indeed generated yet. Thus the reference of the parent to it may be a potential *metadata forward reference*. Since snapshots are revealed to the readers only after the metadata of all lower version snapshots was written, the segment tree associated to v_a will be consistent at the time snapshot v_a is revealed to the readers.

In any of the two cases, both the node (which belongs to Q) and its sibling (if it belongs to Q) are moved from Q to T, which holds the set of all tree nodes belonging to v_a that have been successfully combined. Once the root of the segment tree has been generated, T holds the whole subtree corresponding to v_a and is committed to $Nodes_{global}$. At this point the metadata building

process has successfully completed and BUILD_METADATA returns.

6 Discussions

6.1 Fault tolerance

In large-scale distributed systems an important issue is fault tolerance: because of the large number of machines involved, faults invariably occur frequently enough to be encountered in regular use rather than exceptional situations. A desirable property to achieve is then fault transparency: based on a self-healing mechanism automatically invoked when faults occur, applications can continue their execution without interruption. Additionally, it is also desirable for the system to withstand faults while providing a level of performance close to the case when no faults occur.

In our approach, we address fault transparency at several levels. First, any write operation writes data first, and, once this phase is finished, a snapshot version is assigned to the write, then finally the corresponding metadata is generated and stored in a second phase. Thus, if a writer fails during the first phase (i.e. before requesting a snapshot version), the fault is tolerated transparently. The system remains in a consistent state, because nobody else in the system (but the writer) is aware of the new data that were written. These extra data do no harm and can be garbage-collected in an offline manner. Let us now assume that a writer completes the first phase, is assigned a snapshot version, then fails during the second phase. The system is in an inconsistent state because metadata has not fully been built yet. This blocks the generation of the corresponding snapshot and also that of the snapshots corresponding to subsequent write operations. This situation is handled as follows. If metadata takes too long to be generated, a timeout occurs on the version manager, which then delegates metadata generation for that snapshot to a randomly selected metadata provider.

To cope with failures of data or metadata providers, we use a simple replication scheme. Each data chunk is replicated on several data providers, whose identity is kept within the metadata. Similarly, metadata is also replicated on multiple metadata providers which can be determined using a simple hashing scheme. Finally, to avoid for a possible failure of the version manager to compromise the whole system, a group of version managers can be employed that run a consensus protocol like [18, 19], such that other members of the group can take over for a failing member. This feature has not been implemented yet in BlobSeer.

Note also that the use of versioning and the application of our principle consisting in only adding data and metadata rather than rewriting metadata greatly simplifies replica management. By isolating readers from writers, replication can efficiently be realized asynchronously. There is no need for any complex and costly mechanisms for maintaining replica consistency, as both data chunks and metadata are read-only. As illustrated in section 7.3, the cost of our replication scheme is negligible. In order to maintain the replication factor constant, we have chosen an offline approach that relies on monitoring active replicas. As soon as it is detected that a provider goes down, all replicas stored by that provider are marked as unavailable. For each unavailable replica, a new provider is instructed to fetch a copy of the replica from one of the providers that holds an active copy. Once this is successfully performed, the replica is marked as active again.

6.2 Consistency semantics

With respect to consistency semantics, our approach guarantees *linearizability* [12], which provides the illusion that each operation applied by concurrent processes appears to take effect instanta-

neously at some moment between the invocation and the completion of the operation. This provides strong consistency guarantees that enable easy reasoning on concurrency at application level.

In our case, linearizability applies to snapshot generation. Readers access snapshots explicitly specified by the snapshot version. Writers do not access any explicitly specified snapshot, so we associate them to an implicit virtual snapshot, which intuitively represents the most recent view of the BLOB. This virtual snapshot will actually be materialized when the write operation completes. Total ordering of all updates is guaranteed. For read operations, we define the completion of the primitive to be the invocation of the associated callback (see Section 4.1). For writes, the completion is the moment when the assigned snapshot is successfully generated. Using this abstraction, readers and writers never access the same object and any interleaving of reads and writes is linearizable.

Note that our choice to define write completion as the moment when the corresponding snapshot version is successfully generated, has an interesting consequence. The callback associated to the asynchronous write operation may be invoked on the writer *before* the actual completion and the corresponding snapshot may be exposed by the system to readers at a later time. This can potentially lead to a situation where a snapshot cannot be read immediately even by the same writer. This does not lead to inconsistent situation, as any read operation always takes the version as an input parameter. Our approach has even an advantage: a writer can produce multiple updates asynchronously and does not have to wait in turn for the corresponding snapshots to be generated. This feature enhances parallelism without sacrificing the ease of use provided by strong consistency guarantees and still avoids the need for complex synchronization mechanisms that are typically necessary in weaker models.

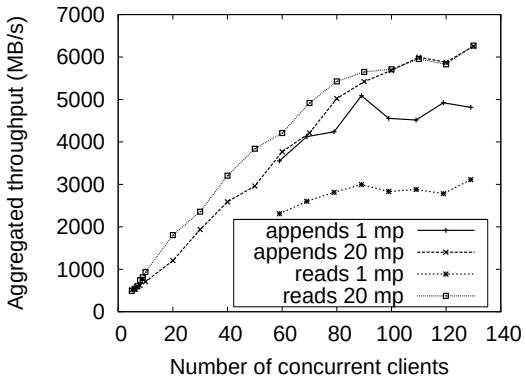
7 Evaluation

7.1 Experimental environment

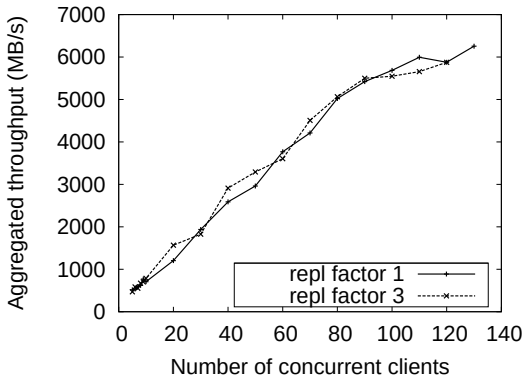
The experiments presented in this section have been performed on the Grid'5000 [16] experimental grid testbed distributed over 9 different sites in France. We have used the nodes of the clusters located in Rennes and Orsay. The nodes are outfitted with x86_64 CPUs and 4 GB of RAM for the Rennes cluster and 2GB of RAM for the Orsay cluster. Intracluster bandwidth is 1 Gbit/s (measured: 117.5 MB/s for TCP sockets with MTU = 1500 B) for Rennes and 10 Gbit/s (measured: 527 MB/s for TCP sockets with MTU = 1500 B) for Orsay.

7.2 Impact of metadata decentralization

In a first series of experiments, we analyze the impact of metadata decentralization on the total aggregated throughput achieved by concurrent readers and appenders respectively, when small amounts of data are written in a fine-grain manner using small chunk sizes, such that metadata overhead becomes significant. Each of the curves measures the aggregated throughput achieved when N concurrent clients append 512 MB in chunks of 256 KB (respectively read 512 MB from disjoint parts of the resulting blob). For each scenario we use the nodes of the Rennes cluster. 130 data providers are deployed on different nodes, while each of the N clients is co-deployed with a data provider on the same node. We consider the case when a single metadata provider is deployed versus 20 metadata providers deployed on separate nodes, different from the ones where data providers are deployed.



(a) Impact of distributed metadata management: read and append performance for 1 vs 20 deployed metadata providers.



(b) Replication overhead: aggregated throughput for appends with variable metadata replication factor

Figure 5: Metadata efficiency under unfavorable conditions: small data sizes (512MB) manipulated by the clients in tiny chunk sizes (256KB)

The results are represented in Figure 5(a). As expected, in the case of a small chunk size, having 20 metadata providers instead of one substantially improves access performance, both for readers and appenders. The best improvement occurs is the case of reads, where the total aggregated throughput has more than doubled.

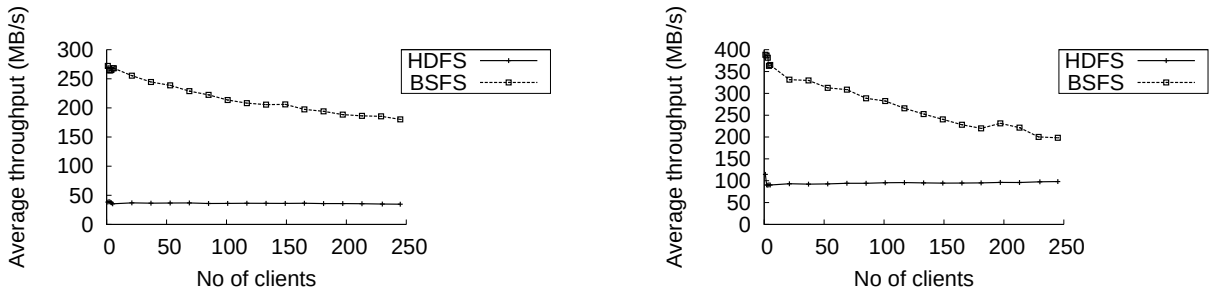
7.3 Impact of metadata replication overhead on throughput

In a second series of experiments we analyze the impact of metadata replication on append throughput. We use the same deployment setup as in the previous section, fixing the number of metadata providers to 20. We execute the appends in the same conditions (512MB per client, 256KB chunk size) and measure the aggregated throughput using metadata replication factors of 1 (no replication) and 3 (2 replicas) respectively.

Results are represented in Figure 5(b). As can be observed, the cost of metadata replication is negligible, even when concurrent appenders write data in small chunks (thus generating a lot of metadata).

7.4 BlobSeer as a file system for Hadoop

To evaluate the benefits of using BlobSeer as a concurrency-optimized storage layer for MapReduce applications, we integrated BlobSeer into the Hadoop MapReduce framework. The Hadoop framework accesses its storage layer through a Java API, that exposes a specific file system interface. We implemented this interface on top of BlobSeer, by adding an new layer that we called the *BlobSeer File System - BSFS*. Then we substituted the original data storage layer of Hadoop, *Hadoop Distributed File System - HDFS* with BSFS. Using this approach (preliminarily described in our previous work [23]), we have now extensively evaluated the impact of our BSFS by performing new experiments both with synthetic microbenchmarks and with real MapReduce applications. The microbenchmarks consist of processes that access the storage layer directly using the file system



(a) Write performance

(b) Read performance

Figure 6: Performance of HDFS and BSFS when concurrent clients access different files

interface, whereas MapReduce applications access the storage layer through the MapReduce framework. Due to space constraints we only included the results obtained with the *sort* application.

7.4.1 Environmental setup

For this series of experiments we used the nodes of the Orsay cluster. Both the microbenchmarks and the MapReduce applications were performed using 270 nodes, on which we deployed both BSFS and HDFS. For HDFS we deployed the metadata server (called namenode) on a dedicated machine and the storage nodes (called datanodes) on the remaining nodes (one entity per machine). For BSFS, we deployed one version manager, one provider manager, one node for the namespace manager and 20 metadata providers. The remaining nodes were used as data providers. As HDFS handles data in 64 MB chunks, we set the page size at the level of BlobSeer, to 64 MB.

7.4.2 Microbenchmarks

The goal of the microbenchmarks is to evaluate the throughput achieved by BSFS and HDFS when multiple, concurrent clients access the file systems, under several test scenarios. The scenarios we chose are common access patterns in MapReduce applications. For each microbenchmark we measure the average throughput achieved when multiple concurrent clients perform the same set of operations on the file system. The clients are launched simultaneously on the same machines as the datanodes (data providers, respectively). The number of concurrent clients ranges from 1 to 246. Each test is executed 5 times, for each set of clients.

Concurrent writers, each writing to a different file In this test scenario, we start N clients that write to HDFS/BSFS concurrently. Each client writes a 1 GB file sequentially in blocks of 64 MB. This microbenchmark reproduces a pattern corresponding to a typical "reduce" phase of a MapReduce application, when all the reduce tasks generate and write different output files. Figure 6(a) shows the write performance of both HDFS and BSFS. As expected, when a client writes on a machine where a datanode was started, HDFS's policy of writing the first copy (and the only one, in this case) locally, leads to a constant average throughput. BSFS achieves a significantly

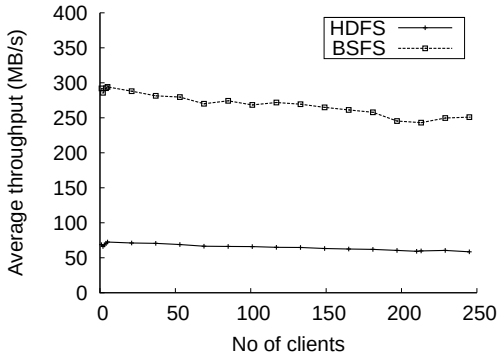


Figure 7: Performance of HDFS and BSFS when concurrent clients read different parts from the same file

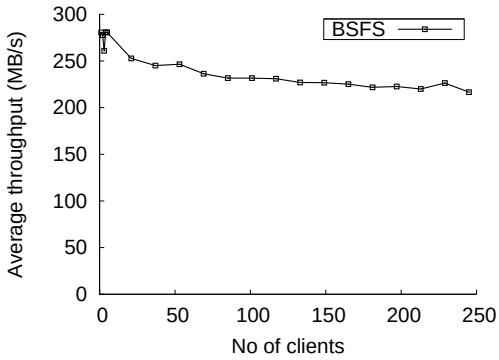


Figure 8: Performance of BSFS when concurrent clients append data to the same file

higher throughput than HDFS, which is a result of the balanced, round-robin block distribution strategy used by BlobSeer. A high throughput is sustained by BSFS even when the number of concurrent clients increases.

Concurrent readers, each reading from different files In this experiment, N concurrent clients read each a different 1 GB file, sequentially in chunks of 64 MB. This test scenario with multiple concurrent readers, each processing a large file, corresponds to the "map" phase of a MapReduce application, when the mappers read the input files in order to parse the $(key, value)$ pairs. As shown by the previous test, writing a file on a datanode is performed locally. Thus, reading the file on the same datanode is also performed *locally*. As far as BSFS is concerned, reading a file is performed *remotely*, because the file is spread over several providers. In order to achieve a proper comparison, we configured HDFS so that the clients read files *remotely*. This is done by letting the files be stored by datanodes not co-deployed with them. The average throughputs delivered by HDFS and BSFS in this test case are shown in Figure 6(b). Although HDFS reads remotely, the chunks read by a client are all stored by the same datanode. Since the reading is done sequentially, the datanode will serve the read requests one at a time. For this reason, HDFS is able to maintain a constant throughput even when dealing with a large number of clients. In contrast, in BSFS, a provider has to serve read requests arriving concurrently from multiple clients. Although BSFS performs significantly better than HDFS, there is a decrease in the average throughput when the number of clients increases.

Concurrent readers, each reading from the same file This microbenchmark tests the performance of the file systems when concurrent clients read different (non-overlapping) parts from the same file. Each client reads a 64 MB chunk, starting from a unique offset in the shared file. For a configuration of N clients, the input shared file is $N \times 64$ MB of size. The file is created so that the N chunks are distributed among the datanodes/providers for both HDFS and BSFS. The obtained results are displayed on Figure 7. In BlobSeer the shared file is uniformly striped among the providers, using a round robin pattern. The average throughput delivered by BSFS is thus high. In contrast, HDFS uses a random data layout policy which leads to load imbalance for datanodes when processing read requests: some of the datanodes get saturated with client requests.

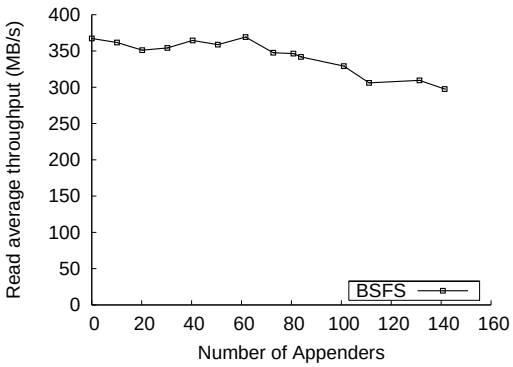


Figure 9: Impact of concurrent appends on concurrent reads from the same file

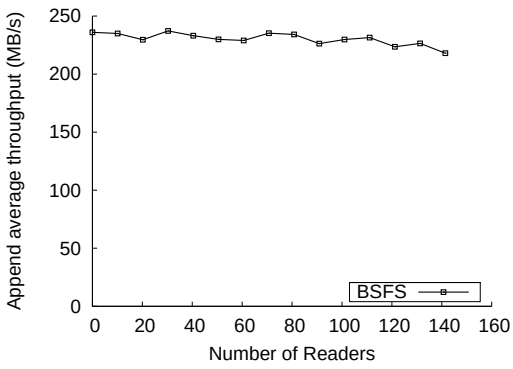


Figure 10: Impact of concurrent reads on concurrent appends to the same file

Concurrent appends to the same file This scenario is not originally supported by Hadoop, since HDFS does not implement appends. However, this functionality could be useful in the MapReduce context. It could for instance enable all the reducers to write their outputs to the same file, instead of creating many output files as it is currently done in Hadoop. In this test case, N concurrent clients append each a 64 MB chunk to the same file. As HDFS does not implement appends, this experiment was performed only for BSFS. The results are displayed on Figure 8. They show that BSFS maintains a good throughput as the number of appenders increases.

Concurrent reads and appends to the same file The test shown in Figure 9 assesses the performance of concurrent read operations from a shared file, when they are executed simultaneously with multiple appends to the same file. The test consists in deploying 100 readers and measuring the average throughput of the read operations for a number of concurrent appenders that ranges between 0 (only readers) and 140. Each reader processes 10 chunks of 64 MB and each appender writes 16 such chunks to the shared file. Each client processes disjoint regions of the file. The obtained results show that the average throughput of BSFS reads is sustained even when the same file is accessed by multiple concurrent appenders. As a consequence of the versioning-based concurrency control in BlobSeer, the appenders work on their own version of the file, and thus do not interfere with the older versions accessed by read operations.

Concurrent appenders maintain their throughput as well, when the number of concurrent readers from a shared file increases, as can be seen on Figure 10. In this experiment, we fixed the number of appenders to 100 and varied the number of readers accessing the same file from 0 to 140. Both readers and appenders access 10 chunks of 64 MB.

7.4.3 Experiments with MapReduce applications

We compared the performance of HDFS and BSFS when being used by the Hadoop framework to execute several MapReduce applications: *sort*, *distributed grep* and *random text generator*. We chose to discuss here only the results for the *sort* application: it is the most complex, it involves both reads and writes and is representative of workloads commonly encountered in the data-intensive community.

Sort is a standard MapReduce application that sorts key-value pairs. The key is represented

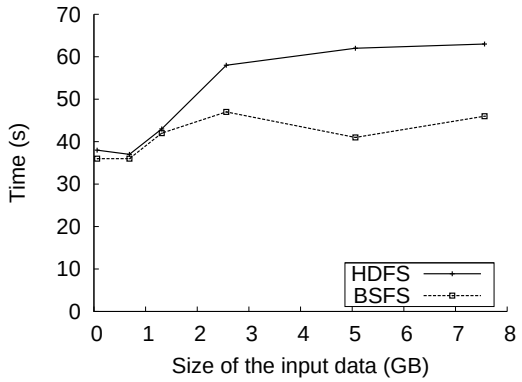


Figure 11: Sort - Job completion time

by the first 10 bytes from each record, while the value is the remaining 100 bytes. This application is read-intensive in the *map* phase and it generates a write-intensive workload in the *reduce* phase. The access patterns exhibited by this application are thus *concurrent reads from the same file* and *concurrent writes to different files*.

In addition to the deployment of HDFS and BSFS, the environmental setup in which this application was run also includes the entities belonging to the Hadoop framework: the jobtracker, deployed on a dedicated node, and the tasktrackers, co-deployed with the datanodes/providers. The input file processed by the application is stored in 64 MB chunks spread across the datanodes/providers. The Hadoop jobtracker starts a mapper to process each chunk from the input file. The input data was generated so as to vary the number of mappers from 1 to 121. This corresponds to an input file whose size varies from 64 MB to 8 GB. For each of these input files, we measured the job completion time when HDFS and BSFS are respectively used as storage layers.

Figure 11 displays the time needed by the application to complete, when increasing the size of the input file. When using BSFS as a storage layer, the Hadoop framework manages to finish the job faster than when using HDFS. These results are consistent with the ones delivered by the microbenchmarks. However, the impact of the average throughput when accessing a file in the file system is less visible in these results, as the job completion time includes not only file access time, but also the computation time and the I/O transfer time.

8 Conclusion

Data intensive computing brings forward many challenges in exploiting the parallelism of current and upcoming computer architectures. In this paper, we addressed several major requirements related to these challenges. One such requirement is the need to efficiently cope with massive unstructured data (organized as huge sequences of bytes - BLOBs that can grow to TB) in very large-scale distributed systems while maintaining a very high data throughput for highly concurrent, fine-grain data accesses.

To address these requirements, we propose a set of principles optimized for heavily concurrent data accesses. In particular, we introduce *versioning* at the very core of the design and we show how it can be efficiently leveraged to optimize data access of data-intensive applications under heavy access concurrency. To this end, we propose a concurrency-optimized, versioning-based

BLOB management interface that guarantees atomic snapshot generation for BLOB updates. We also show how to implement such interface efficiently by introducing a set of algorithms for data management, which are based on the above mentioned principles. They enable fully decoupled read and write operations, thanks to versioning. This in turn improves parallel access: reads never block writes and writes never block reads. Moreover, our approach enables writes to execute mostly in parallel as well, with minimal synchronization. These algorithms are demonstrated in the BlobSeer prototype and evaluated at a large scale on the Grid'5000 testbed.

The experiments show that it is definitely possible to sustain a high data throughput thanks to our efficient versioning scheme for massive objects that are accessed at fine granularity. We demonstrated scalability of concurrent access to the same BLOB in the order of hundreds of clients, and predict the same level of scalability in the order of thousands. By using BlobSeer as a storage layer for Hadoop, substantial gains are obtained for many access patterns which exhibit concurrency: concurrent reads to the same file, concurrent writes to the same file, concurrent reads (or writes) to different files. Moreover, we show how these theoretical benefits apply in practice, by showing substantial speed-ups for real MapReduce data-intensive applications.

We believe that BlobSeer opens the way towards a new approach to massive data management at a very large scale under heavy access concurrency, especially thanks to *versioning*, which proves to be a very competitive alternative to traditional lock-based approaches.

Many aspects have not been studied in depth yet. The precise semantics of the version-based management of shared data can be investigated closer, and compared with the traditional consistency notions which have been proposed. Finally, we are investigating more possible applications of this approach. In this paper, we presented a BlobSeer-based file system, BSFS which outperforms HDFS for MapReduce applications. Our approach could certainly be beneficial to other classes of data-intensive applications, as its underlying principles are generic enough: in this direction, there is certainly a large field for investigation ahead.

Acknowledgments

The experiments presented in this paper were carried out using the Grid'5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/ for details).

References

[1] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing Journal*, 28(5):749–771, 2002.

[2] Roberto Baldoni, Mariangela Contenti, and Antonino Virgillito. The evolution of publish/subscribe communication systems. In *Future Directions in Distributed Computing*, pages 137–141, 2003.

[3] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. pages 129–139, 1988.

[4] Philip H. Carns, Walter B. Ligon, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[5] Wang Dan and Li Maozeng. A range query model based on DHT in P2P system. In *NSWCTC '09: Proceedings of the 2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, pages 670–674, Washington, DC, USA, 2009. IEEE Computer Society.

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

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[8] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[9] Ian Foster and Nicholas T. Karonis. A grid-enabled MPI: message passing in heterogeneous distributed computing systems. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–11, Washington, DC, USA, 1998. IEEE Computer Society.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS - Operating Systems Review*, 37(5):29–43, 2003.

[11] Val Henson, Matt Ahrens, and Jeff Bonwick. Automatic performance tuning in the Zettabyte file system, 2003.

[12] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[13] Peter Honeyman, Wandros A. Adamson, and Shawn McKee. GridNFS: global storage for global collaborations. In *Proc. IEEE Intl. Symp. Global Data Interoperability - Challenges and Technologies*, pages 111–115, Sardinia, Italy, June 2005. IEEE Computer Society.

[14] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The XtreemFS architecture – a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, 2008.

[15] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[16] Yvon Jégou, Stephane Lantéri, Julien Leduc, Melab Noredine, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touche Iréa. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[17] Peter Z. Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. File-based replica management. *Future Generation Computing Systems*, 21(1):115–123, 2005.

[18] Leslie Lamport. Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, pages 32–4, 2001.

[19] Leslie Lamport and Keith Marzullo. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[20] Bogdan Nicolae. BlobSeer: Efficient data management for data-intensive applications distributed at large-scale. In *24th IEEE International Symposium on Parallel and Distributed Processing: Workshops and Phd Forum (IPDPS '10)*, pages 1–4, Atlanta, USA, 2010.

[21] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Blobseer: how to enable efficient versioning for large object storage under heavy access concurrency. In *EDBT/ICDT '09 Workshops*, pages 18–25, Saint-Petersburg, Russia, 2009. ACM.

[22] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling high data throughput in desktop grids through decentralized data and metadata management: The BlobSeer approach. In *Proc. 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*, volume 5704 of *Lect. Notes in Comp. Science*, pages 404–416, Delft, The Netherlands, 2009. Springer-Verlag.

[23] Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, and Matthieu Dorier. BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map/Reduce applications. In *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS '10)*, pages 1–11, Atlanta, USA, 2010.

[24] Aravindan Raghuveer, Meera Jindal, Mohamed F. Mokbel, Biplob Debnath, and David Du. Towards efficient search on unstructured data: an intelligent-storage approach. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 951–954, New York, NY, USA, 2007. ACM.

[25] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: a public DHT service and its uses. In *Proc. 2005 Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '05)*, pages 73–84, New York, NY, USA, 2005. ACM.

[26] Ohad Rodeh. B-trees, shadowing, and clones. *Transactions on Storage*, 3(4):1–27, 2008.

[27] Bayer Rudolf and McCreight Edward. Organization and maintenance of large ordered indexes. pages 245–262, 2002.

[28] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 110–123, New York, NY, USA, 1999. ACM.

[29] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 231–244. USENIX Association, 2002.

[30] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium*, 2003.

[31] K. Shvachko, H. Huang, S. Radia, and R. Chansler. The hadoop distributed file system. In *26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies*, May 2010.

[32] Konstantin Shvachko. Hdfs scalability: The limits to growth. *;login: - The USENIX Magazine*, 35(2), 2010.

[33] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, Frans F. Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

[34] Osamu Tatebe, Youhei Morita, Satoshi Matsuoka, Noriyuki Soda, and Satoshi Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *Proc. 2nd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (Cluster 2002)*, page 102, Washington DC, USA, 2002. IEEE Computer Society.

[35] Tran Doan Thanh, Subaji Mohan, Eunmi Choi, SangBum Kim, and Pilsung Kim. A taxonomy and survey on distributed file systems. In *NCM '08*, pages 144–149, Washington, DC, USA, 2008. IEEE Computer Society.

[36] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[37] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[38] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed Segment Tree: Support range query and cover query over DHT. In *Proceedings of the Fifth International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, California, 2006.

[39] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[40] The Blue Waters project. http://www.ncsa.illinois.edu/BlueWaters/.