UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE LICENȚĂ

Eliminarea datelor false stocate în BlobSeer

Coordonatori ştiinţifici:
Prof. Dr. Ing. Valentin Cristea
As. Drd. Ing. Cătălin Leordeanu

Absolvent:
Mihaela Badiu

BUCUREŞTI

2011

UNIVERSITY POLITECHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

BACHELOR THESIS

BlobSeer False Data Deletion

Thesis supervisors:
Prof. Dr. Ing. Valentin Cristea
As. Drd. Ing. Cătălin Leordeanu

Author:
Mihaela Badiu

BUCHAREST

2011

Table of Contents

Abstract

This project addresses the issue of garbage collection in the BlobSeer data management system. For the moment this distributed service offers efficient methods of storing and accessing very large binary data objects (blobs). It contains a versioning scheme and an original metadata scheme which allow a large number of clients to concurrently read, write and append data to huge blobs that are fragmented and distributed at a very large scale. However, malicious users can add false data or use inefficiently the available storage space. Taking this into consideration and the fact that data is always created and never overwritten in the system, the issue of being able to delete unwanted data becomes crucial. As a result, this thesis proposes a BlobSeer data deletion algorithm which can be used to eliminate false data that would otherwise pollute the system.

Keywords: *data management, data-intensive application, versioning, concurrency, useless data*

1. Introduction

Large scale data management is becoming increasingly important in the context of data-intensive, massively parallel applications. Examples of such applications with high demands in terms of computational resources and data storage could be found in fields like astronomy, bio-informatics, physics, databases or multimedia. As the efficiency of the data management system has a significant impact on the global application performance, a series of major requirements need to be considered such as *scalable architecture*, *massive unstructured data* processing, transparency with respect to *data localization* and *data transfers*, *high throughput under heavy access concurrency* and support for *highly parallel data workflows* [1]. Aspects like decentralized management, synchronization, consistency, fault tolerance, data fragmentation and replication are also important. Replication increases data availability, synchronization helps to maintain a consistent view of the system, while decentralized management and data distribution enhance the degree of parallelism.

BlobSeer is an example of a distributed storage infrastructure designed to target the above-mentioned needs of data-intensive applications. In this system, unstructured data is organized as a set of huge objects called BLOBs (binary large objects), consisting of long sequences of bytes. These objects are made up of fixed-size blocks, referred to as pages, and distributed among storage providers. Data access performance is achieved by means of a decentralized metadata management scheme and an original versioning-based approach which allow concurrent reads and writes to the same BLOB, as well as efficient fine-grain access to objects by associating a range for any generated version with the data providers that store it.

In the situation where we need to store huge amounts of data, BlobSeer offers performance and efficiency but there is a need to also delete some data. For example, a malicious user will create a Denial of Service (DoS) attack on the data-sharing system by writing large amounts of useless or corrupted data, thus overloading the whole system and increasing its response time. Once these hostile actions are detected, the inserted data must be immediately erased.

Moreover, a user may want to mark old versions of some BLOBs as not needed anymore. As each object evolves in a linear fashion, unintentional updates may prevent important overwritten pages from being part of the subsequently generated versions. Currently there is no way of eliminating useless or accidentally written data. As a consequence, this project proposes an efficient scheme for the deletion of such data from the system in order to free valuable disk space. We focus on adding the capability to delete a version of a given blob and then extend it to eliminate false data or data which is older than a specified version.

The structure of this thesis is as follows. Section 2 describes the main features of some data management systems for large scale distributed infrastructures, emphasizing and comparing the way deletion is handled in these systems. Section 3 offers an overview of BlobSeer distributed data storage service as a background to the proposed version deletion algorithm which is described in the next chapter together with its data structures and exposed user interface. Interactions between system entities as part of the deletion protocol are also reviewed. After the algorithm is thoroughly explained, Section 4 ends by addressing the implied concurrency issues. Section 5 focuses on implementation details of the delete primitive, including class diagrams and description of the used tools and libraries, while Section 6 shows how it can be applied to some test scenarios. Evaluation and experimental results of our approach are discussed in Section 7. Finally, Section 8 draws conclusions and outlines directions for future work.

2. Related work

Extensive research related to large scale data management has led to the development of many distributed storage systems, each with a specific architecture and functionality. Examples include Google File System (GFS), Hadoop Distributed File System (HDFS), Parallel Virtual File System (PVFS2), Amazon S3, Dynamo or Lustre. They provide a file-oriented application programming interface which complies with the transparent data-access model: remote files are accessed the same way as the local ones. Transparency is offered by *distributed file systems* [2] through a shared file namespace.

Next, we survey and compare three data management systems similar to BlobSeer, namely Hadoop Distributed File System (HDFS), Google File System (GFS) and Amazon S3. We focus on their key features, the implied advantages and disadvantages, and especially on the way data deletion is managed in these systems.

2.1 Hadoop Distributed File System

HDFS [3] is the underlying storage layer of Hadoop open-source MapReduce implementation. It stores very large data sets reliably across machines in a large cluster and accomplishes transfers of those data sets at high bandwidth. Like in any cluster-based distributed file system, metadata and data are decoupled. Metadata is stored on a dedicated server, called the NameNode, while DataNodes host application data. Besides maintaining metadata, the NameNode keeps track of each DataNode, or, more precisely, it records the total storage capacity, the fraction of storage in use and the number of data transfers currently in progress. This information is used to enable efficient space allocation and load balancing strategies. The primary role of the NameNode, however, is to serve client requests. The resulting disadvantages refer to scalability and data availability issues, as the number of files and concurrent requests grows, the NameNode will become unresponsive.

In contrast, BlobSeer divides the functionalities of the Hadoop centralized metadata server among a provider manager and multiple metadata providers. Featuring a *symmetric architecture* [2], this highly scalable storage system uses a distributed hash table (DHT) based metadata management scheme combined with a key-based lookup mechanism. This way, it avoids the bottleneck of querying the same node and the situation when the whole cluster becomes unavailable due to a single node failure.

In addition, HDFS implements a single writer, multiple readers model. When a client opens a file for writing, it is granted a lease and therefore no other client can write to this file. BlobSeer, on the other hand, efficiently supports concurrent writes to the same object thanks to versioning based concurrency control.

As for space reclamation, when a user issues a delete request, the file is not immediately removed from HDFS. Instead, it is renamed and moved in the /trash directory. A file remains in /trash for a configurable amount of time while it can be restored. After this period expires, the NameNode deletes the file from the namespace by delivering commands to all the DataNodes that store part of the file. In contrast, we intend to make the BlobSeer client responsible for eliminating data. It first gets the metadata that let him know the providers that host the pages which should be deleted and after that it contacts the metadata and data providers in parallel asking them to free the storage space.

2.2 Google File System

GFS [4] is another cluster-based distributed file system aimed to satisfy the needs of the MapReduce paradigm. Similar to HDFS, each cluster consists of a single master and multiple chunkservers. Huge files are divided into chunks of 64 Megabytes and distributed among chunkservers. The centralized master assigns unique chunk handlers each time a new chunk is created. It stores metadata by mapping files to chunk handlers and chunks to their locations. It also interacts with all chunkservers through regularly exchanged HeartBeat messages. Status information collected from those messages is used to take chunk placement and replication decisions.

GFS avoids the bottleneck of having a single metadata server by favoring a large chunk size which implies less stored metadata and fewer requests from clients. However, lazy space allocation leads to internal fragmentation. Moreover, small files will consist of a small number of chunks and, in consequence, the chunkservers storing them could be overloaded if many clients are accessing the same file. As opposed to that, in BlobSeer, the chunk size is specified at the time of blob creation and it usually matches or is a multiple of the size of the data the client is supposed to process in one step [5].

Optimized for applications that append new data rather than overwrite the existing data, GFS allows multiple clients to append sequences to the same file concurrently while guaranteeing the atomicity of each individual append. With the assumption that once written, files are seldom modified again, concurrent small writes to arbitrary positions in a file are poorly supported. More than that, concurrent writes to the same region are not serializable and so the region may end up containing data fragments from multiple clients. In contrast, BlobSeer efficiently handles concurrent writes at arbitrary offsets in the same blob.

Regarding storage reclamation, the GFS centralized master regularly scans the chunk namespace to identify unreachable or unreferenced chunks and to erase their associated metadata. During a HeartBeat message exchange, each chunkserver reports the set of the chunks it has and the master replies with the identifiers of all the chunks that are no longer present in its metadata so the chunkserver can delete them. When the client deletes a file, the master only logs the request and the file is renamed. Then, in a similar scan of the file system namespace, it removes such files if they have existed for a configurable period of time by erasing its metadata and implicitly the links to its chunks. As stated before, we aim instead to implement BlobSeer data deletion on the client side and to immediately and irrevocably remove one or more versions of a given blob once a delete request is initiated.

2.3 Amazon S3

Amazon S3 [6] is a distributed storage system that provides a simple web service interface to write, read and delete an unlimited number of objects of sizes up to 5 Terabytes. Its specialized design brings high scalability and data availability, reliability and fast access speed. Each object is stored in a bucket, the fundamental container in Amazon S3 for data storage, and retrieved via a unique key. The user is allowed to create a bucket, write a file by uploading it to a certain bucket, open and download an object, move it from one bucket to another and finally delete it. Only full object reads and writes are supported. Also, no object-locking mechanism is provided, but eventual consistency is always guaranteed. If two updates are simultaneously made to the same

key, the request with the latest time stamp wins. As opposed to S3, BlobSeer provides several advanced features such as data striping, efficient fine-grain access and concurrent writes to the same object.

Recently, versioning support was also added. Once versioning is enabled for a given bucket, unique version IDs are added to each object in that bucket. Therefore, two objects with the same key, but different version IDs can be stored in the same bucket. Data deletion is accomplished both at version and object level by specifying a key and optionally a version ID. Eventually, when all the objects in a bucket were removed, the client can also issue a bucket deletion request. However, with the aim of recovering from unintended overwrites or deletions, each new version involves writing an entire object and not just a difference from the previous one. BlobSeer new snapshot generation, on the other hand, entails storing only the different part of the blob with respect to the lower versions, this way saving storage space and maximizing concurrency.

3. BlobSeer

BlobSeer is a highly scalable data management service which meets the demands of the applications that generate and process very large volumes of data. This section describes the architecture of the system, the interactions between its entities and the main features that contribute to the enhancement of concurrency.

3.1 Architecture

BlobSeer consists of a series of distributed processes communicating through remote procedure calls. Figure 1 illustrates these processes, each fulfilling a specific function.

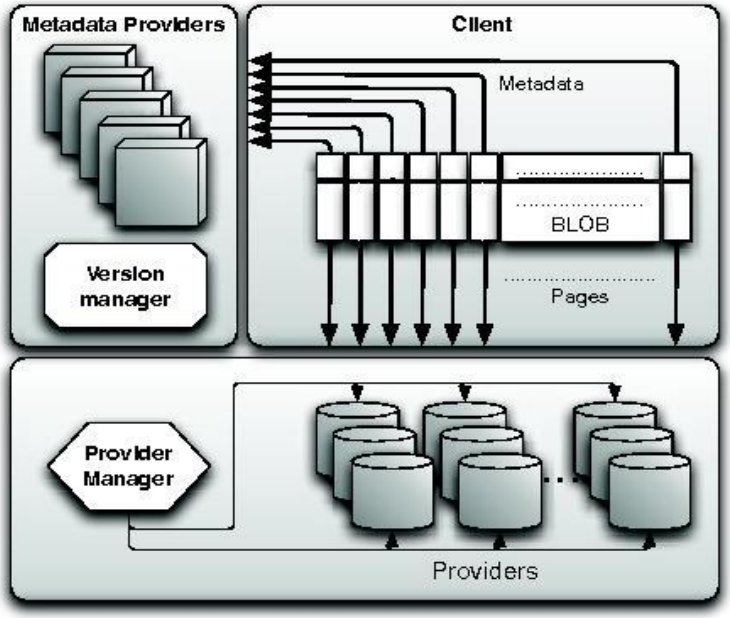


Figure 1: Architecture

Clients initiate CREATE, READ, WRITE and APPEND requests by means of a user interface. There may be multiple concurrent clients and more than one may access the same blob.

Data providers are responsible for storing and retrieving individual pages which make up a blob. New data providers may dynamically join or leave the system.

The provider manager monitors the registered data providers by keeping information about the available storage space. On each WRITE or APPEND request, it decides which providers should be used to store the newly generated pages based on a strategy that maximizes the data distribution benefits.

Metadata providers physically store the metadata that allows identifying the pages corresponding to a requested range and version of a blob. They are organized as a Distributed Hash Table which allows efficient concurrent access to metadata.

The version manager stores the number of the latest published version for every generated blob in order to provide readers with the latest available snapshot. It is also in charge of assigning new version numbers to writers and appenders and to reveal them to readers. Concurrent writes and appends to the same object are serialized as far as each request is atomically treated.

3.2 Interactions between system entities

In a typical scenario, each described process runs on a separate physical node, but one node may play multiple roles as it can be a client and a data or metadata provider at the same time. The interactions between entities as a result of the READ and WRITE requests are illustrated in Figure 2 [7].

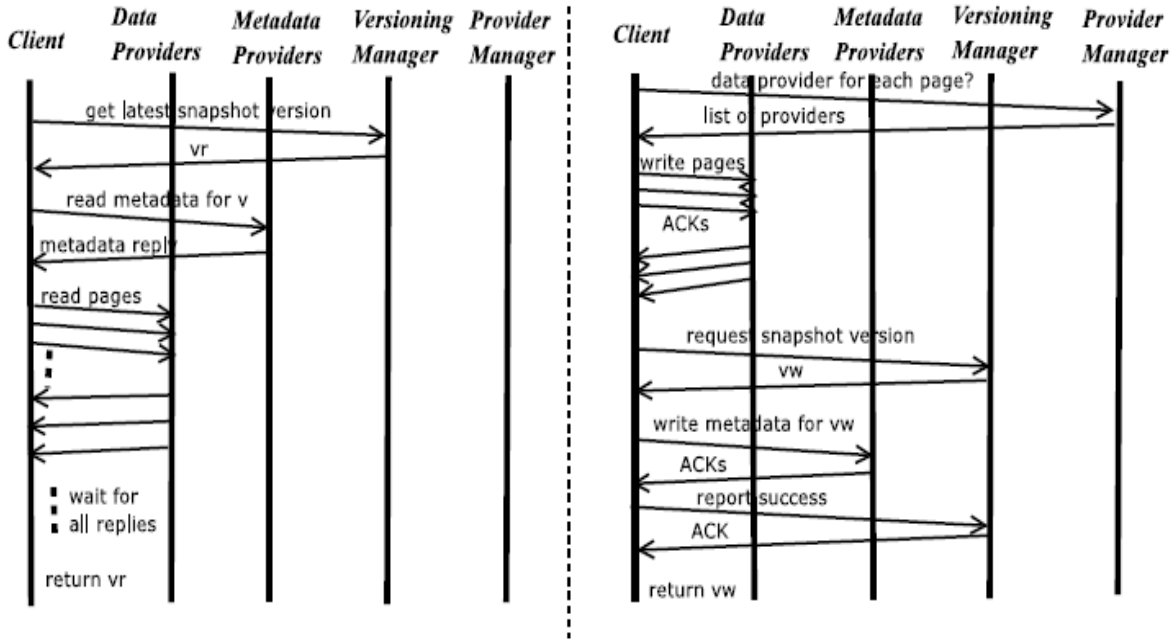


Figure 2: Read operation (left) and write operation (right)

Following a READ request, the client queries the version manager in order to get the latest published version of a specified blob. After getting the answer, it contacts the metadata providers and waits for the reply that comprises the information describing the pages that make up the requested range. Finally, data providers are queried in parallel to retrieve the physically stored pages. Readers can also supply a certain version number as a parameter to the primitive, in which case the version manager checks whether the version has been published and replies accordingly.

The client may also update one object by invoking a WRITE or APPEND primitive. These operations start with a query on the provider manager to get a list of data providers available to host the new pages that cover the supplied range. Then, the pages are sent in parallel to those providers. After required to store one page, each provider replies with an acknowledgement in case of successful operation. When all answers are received, the client requests a new version number from the version manager. This number is needed to generate new metadata and send them to the metadata providers. Only after metadata are written and completely integrated with those of older versions, the client reports success to the version manager and the primitive returns the attached version to the user. From now on, it is the responsibility of the version manager to publish the version, that is, to reveal it to readers in such a way as to maintain consistency.

3.3 Access interface

As mentioned above, the client is allowed to create a blob, to read a subsequence of *size* bytes starting from an *offset* and to write or append a range to the blob. Because blobs are uniquely identified in the system, the CREATE primitive will return the id associated to the newly-generated blob.

id = CREATE()

To read data, the client should pass the following parameters: a blob id, a certain version, a range specified by an offset and a size and a local buffer where the fetched data will be placed.

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

The WRITE and APPEND primitives receive instead a local buffer whose content should be copied into the specified blob and return the version number of the new snapshot that reflects the changes made. APPEND is a particular case of WRITE with the *offset* implicitly being the size of the previous snapshot.

vw = WRITE(id, buffer, offset, size)
va = APPEND(id, buffer, size)

The user may also want to learn about the last generated snapshot of a blob or the size of a specified version by issuing the GET_RECENT and GET_SIZE operations.

v = GET_RECENT(id)
size = GET_SIZE(id, v)

The simplicity of this access interface comes from the fact that the user does not have to be aware of data location or to manage transfers explicitly as these are handled transparently by the storage service. More than that, versioning is provided at the application level so that one client can read from any published snapshot of a blob while other generates new versions of the same blob without the need for synchronization. The corresponding algorithms of this series of primitives are presented in detail in [8].

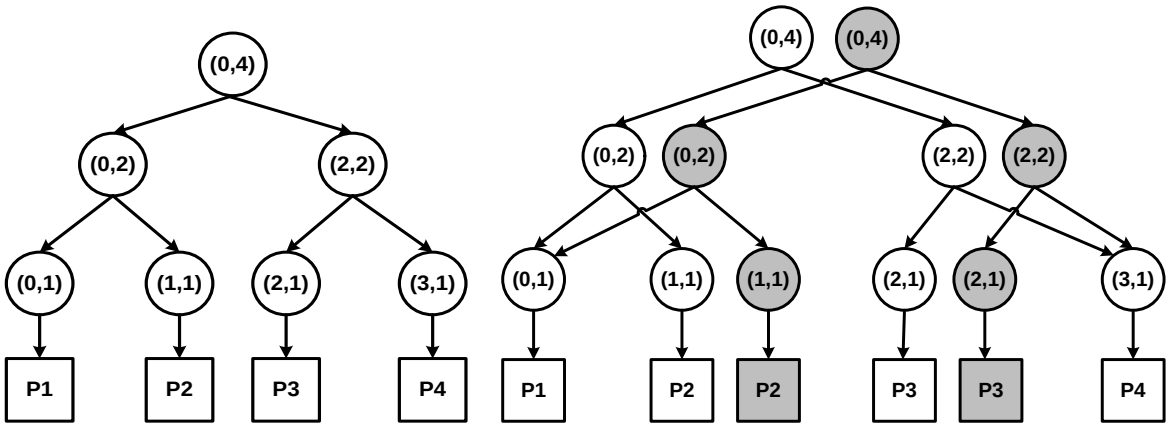
3.4 Distributed metadata management

Since each blob is fragmented and distributed among the storage space providers, there is a need to also maintain metadata that maps subsequences of bytes from any snapshot of each blob to the data providers that store them. This metadata is organized as a distributed segment tree and hosted by metadata providers which form a Distributed Hash Table (DHT). Each snapshot version is identified by such a segment tree which is in fact a binary tree with every node covering a range of the blob. For each inner node, its left child covers the first half of its associated range, while its right child covers the second half. Naturally, each leaf covers only one page. Details and algorithms used to read and build metadata can be found in [8].

This distributed metadata management scheme not only offers scalability in terms of concurrent metadata accesses, but also improves data availability. It avoids the case of a centralized server that stores and manages the whole metadata, being at the same time a single point of failure.

3.5 Versioning-based concurrency control

One key design principle that contributes substantially to the enhancement of performance rests on the idea that each write or append operation produces a new snapshot rather than overwriting the old ones. However, only the different pages with respect to the previous snapshots are actually created. In this way, the new snapshot physically shares all unmodified pages and part of the metadata with the older versions. The advantages resulted from this approach include saving storage space and maximizing concurrency as reads will never conflict with concurrent writes to the same blob.



(a) The metadata after a write of four pages (b) The metadata after overwriting two pages

Figure 3: Metadata representation

These principles are illustrated in Figure 3 above, where inner nodes labeled with the range they cover represent metadata and leaves identify the pages. In fact, each node is stored as a (key, value) pair, where key includes the blob id, the version number and the covered range

delimited by an offset and a size, while value identifies the left and right child. Given an initial blob made up of four pages, a subsequent write generates pages 2 and 3 and also new metadata nodes represented in gray. These new gray nodes are interleaved with the white nodes corresponding to the unmodified first page and forth page respectively.

Concurrent writes and appends are also allowed as they can send data to the storage space providers independently of each other. Synchronization takes place only at the version manager level where the updates are serialized and assigned version numbers in an incremental fashion. Then, support for *metadata forward reference* [1] enables generating metadata in parallel too with the assumption that all concurrent updates will be generated in the future.

4. Deleting data from BlobSeer

4.1 Useless data

Aimed at enabling high aggregated throughput and increasing the overall performance of data-intensive applications, the BlobSeer massive data-sharing system allows clients to directly access the storage nodes. This property makes the system vulnerable to intrusive actions such as Denial of Service (DoS) attacks that cannot be stopped through typical authentication mechanisms. The write operation is the most exposed one since it gives a malicious user the means to insert and publish corrupted data. Eventually, this may lead to an overload of the system and an increase in its response time.

Research work carried out in order to address these issues came up with a *generic security management framework* [9] designed to detect and stop malicious behaviors by enforcing a set of security policies. With the help of an *introspective layer* [10] built on top of BlobSeer, users' activity is monitored and the information collected from the data providers and the version manager is used to detect DoS attacks. Once identified, corrupted data must be irrevocably erased from the system.

Moreover, a user may want to remove old, unnecessary or unintentionally written versions. For this purpose, we propose in the next subsection two data deletion algorithms, both at version level. Of the two algorithms, we have chosen the most appropriate one to be implemented into BlobSeer and to evaluate its performance.

4.2 Proposed approaches

In order to provide an efficient data deletion strategy, new data structures and remote procedure calls need to be added.

One approach may introduce some modifications in the second phase of the write operation, that is, the metadata building step. As mentioned above, every node in the distributed segment tree is represented as a (key, value) pair, where key = (id, version, offset, size) and value = (left child's version, right child's version) for inner nodes, but (page id, provider) for leaves. This first approach proposes to modify the value field by adding the version number of the last snapshot which directly points to the node in question – in other words, the version of the last parent of the node. As a consequence, after receiving a version number from the version manager, the writer will not only generate new metadata but also update the nodes to which the resulting snapshot will be directly linked. This way, deleting version v implies one traverse of the associated metadata tree beginning with the root. If the last version parameter of the root is not greater than v, which means that this node is not shared with higher snapshot versions, its left and right child nodes will be processed in the same manner. Once a leaf is reached, the covered page can be deleted. While providing a simple way of deleting data at version level, this scheme has some disadvantages. First of all, it breaks the original design principle which states that data and metadata are never modified, but added to the system. Secondly, the modified write operation involves more queries sent to the metadata providers and requires additional synchronization, both leading to an increase of the metadata overhead and thus of the response time.

The second proposed approach instead associates each version with the range (offset, size) written by the corresponding update operation. These ranges are used to identify the pages and the metadata nodes that are shared or not between any two versions. If we consider the same example depicted in Figure 3, we can define two version descriptors: (1, (0, 4)) and (2, (1, 2)), where the first version is represented in white and the second in light gray. Assuming we want to delete the first version, all we have to do next is to traverse its distributed segment tree and intersect the segment covered by each processed node with the range associated to the second version. If the intersection is empty, the processing stops as the whole covered region is shared with the second snapshot. Otherwise, we can go deeper in the segment tree. Each reached leaf together with the nodes along the path to the root should be deleted.

Overcoming the major issues raised by the first method, we chose the latter algorithm to implement and evaluate. Next, we introduce the data structures needed to integrate the delete primitive with little impact on the rest of the operations.

4.3 Data structures

Version descriptor. Each write or append operation results in a new snapshot version v by copying the contents of a local buffer into the blob identified by id. The offset and size that define the integrated subsequence is associated with the version number v to form its descriptor (v, offset, size). The version manager stores a list of such descriptors for every blob. Each list contains descriptors of the snapshots completely generated as well as of those that are in the course of being generated. When the client wants to delete the version v, it has to first obtain the descriptor of the lowest version w higher than v which is still available and the descriptors of all the versions that lie between v and w. If any, those versions are already deleted. Thus, we are interested in retrieving a subsequence of entries like $VD_{id}[v...w]$, where VD_{id} is the list of version descriptors of the blob id. Note that on eliminating a version, if it is not the last one, its descriptor remains in the list as subsequent deletions of lower versions may depend on it.

Version map. Each version can be in one of the following states: pending write, that is, in the process of being generated, available or deleted. The version map is a list indexed by version numbers that holds the associated states. Such lists are also stored by the version manager, one for each blob. Before starting to delete version v, the client needs the first v-1 entries from this version map to eliminate all the metadata nodes and pages that make up the snapshot v and that are not shared with older available versions.

Page descriptor. Each leaf in any metadata tree holds the pid that uniquely identifies the page covered and the provider that stores it. These two parameters form the page descriptor (pid, provider).

4.4 User interface

To delete a version of a particular blob, a client must call the DELETE primitive:

DELETE(id, v)

A DELETE results in eliminating part of the pages and of the metadata nodes associated with the snapshot version v of the object identified by id. In order to preserve a consistent state of the

system, only those data and metadata which are not shared with other still available versions will be deleted.

Furthermore, a client may want to delete old data, which means all snapshot versions lower than a particular one labeled v. This can be achieved with the help of the following operation:

DELETE_EARLIER(id, v)

Important to notice is that the caller must be able to check if the supplied parameter v has a meaningful value. It needs to find out if one version has been published, but also if it has already been deleted.

status = GET_STATUS(id, v)

The additional primitive GET_STATUS returns a Boolean value that indicates whether the version v is available or not. It does not ensure that the version will also be available when subsequent read requests start to be processed by the system as far as it does not provide synchronization and does not block concurrent deletes.

4.5 Interactions between system entities

The process of deleting one snapshot of a blob starts with a notification sent to the version manager, which in turn registers that the client is willing to delete the supplied version and replies with the version map and a list of version descriptors needed for deletion proper. This first notification prevents concurrent readers from getting inconsistent data. If a read request comes right after this delete query, the version manager will reply from the beginning that this version is no longer available and the read operation will fail.

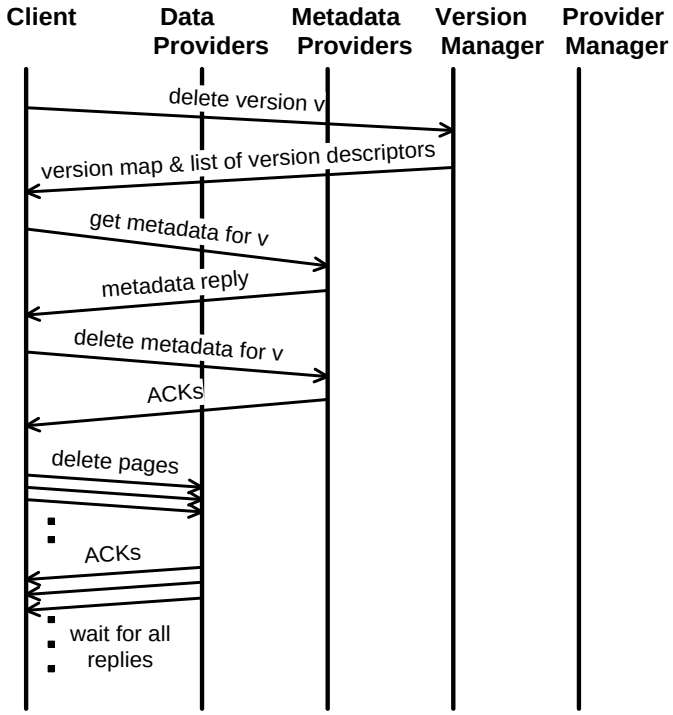


Figure 4: Delete protocol

Next, the client uses the information from the version manager's reply in order to fetch only those metadata nodes which lead to the leaves covering the pages that should be deleted. The metadata nodes traversed by walking the tree from the root to those pages should also be eliminated as long as they are not shared with other snapshots. Eventually, equipped with the necessary information, the client contacts the metadata providers and then the data providers in parallel asking them to free the storage space filled with the aforementioned metadata and pages respectively. All these steps are shown in Figure 4.

To sum up, the delete protocol involves three phases: identifying the metadata and data that are not part of any other still available snapshots, delete the metadata nodes and finally eliminate the pages. We specifically chose this order to reduce the risk of leaving the system with inconsistent metadata in case of failures.

4.6 Version deletion algorithm

The DELETE primitive is presented in Algorithm 1. The client first queries the version manager to check if the specified version v is available and fails if it is not the case. Otherwise, it gets part of the version map, named VM, and a list of version descriptors, noted VD, by invoking remotely on the version manager GET_VERSION_INFO. This information is needed to find out the pages that should be eliminated and on which providers they are stored. For this purpose, the client interacts with the metadata providers and fetches in turn only the nodes that lead to those pages. Detailed steps of this phase are given below where Algorithm 3 is discussed. The READ_METADATA procedure returns the list of page descriptors PD and the keys of the read metadata nodes in KEYS. Both sets are used to issue delete requests in parallel to the metadata providers and then to the data providers so as to finally signal success to the user.

Algorithm 1 DELETE

```
Require: The blob id
Require: The snapshot version v
 1: if GET_STATUS(id, v) == false then
 2:     fail
 3: end if
 4: (VM, VD) ← GET_VERSION_INFO(id, v)
 5: (PD, KEYS) ← READ_METADATA(id, v, VM, VD)
 6: for all key ∈ KEYS in parallel do
 7:     DELETE_NODE(key)
 8: end for
 9: for all (pid, provider) ∈ PD in parallel do
10:     delete pid from provider
11: end for
12: return success
```

Algorithm 2 is executed by the version manager process which given a certain blob id and a version v is responsible for updating the version map of the blob in such a way as to ensure that subsequent read requests for snapshot v will not be served. It is also supposed to build and return two sets: VM that comprises the first v-1 entries of the version map noted VM_{id} and VD that includes the descriptors of all the versions lying within the range [v+1 … w], where w is the first available snapshot higher than v. If no such version exists, then we are facing the particular case of removing the last published snapshot of some blob. In this situation, if new snapshots are in the process of being generated and consequently depend on v and most probably on its metadata, then w will be bound to the first one. Again, if there is no such snapshot, VD remains empty and the latest consecutive versions which have been marked as deleted are removed from VM_{id} as well as from VD_{id}, the version descriptor list of the blob id. In addition, the variable $v_{g,id}$ denoting the most recent version of the blob id and obtained when calling the GET_RECENT primitive is updated accordingly.

Algorithm 2 GET_VERSION_INFO

Require: The blob id
Require: The snapshot version v
Ensure: The version map VM
Ensure: The set of version descriptors VD

1: $VM_{id}[v] \leftarrow$ deleted
2: $VM \leftarrow VM_{id}[1 \dots v\text{-}1]$
3: **for** version $\in$ [v+1 … size(VM_{id})] **do**
4: **if** (VM_{id} [version] == available **or** VM_{id} [version] == pending_write) **then**
5: $VD \leftarrow VD_{id}$ [v+1 ... version]
6: **break**
7: **end if**
8: **end for**
9: **if** (VD == ϕ) **then**
10: **for** version $\in$ [size(VM_{id}) … 1] **do**
11: **if** (VM_{id} [version] == deleted) **then**
12: $VM_{id} \leftarrow VM_{id} \setminus VM_{id}$ [version]
13: $VD_{id} \leftarrow VD_{id} \setminus VD_{id}$ [version]
14: $v_{g,id} \leftarrow v_{g,id} - 1$
15: **else**
16: **break**
17: **end if**
18: **end for**
19: **end if**
20: **return** (VM, VD)

Algorithm 3 deals with the metadata tree processing. The steps required to read only those metadata nodes that lead to the pages appropriate for being deleted can be summarized as follows:

1. First, GET_ROOT(id, v) fetches and returns the root of the distributed segment tree corresponding to version v of the blob id. If a higher version w exists and if the region covered by the root of snapshot v intersects the segment extracted from the version descriptor of w, then the root is inserted in a queue. Otherwise, the processing stops as the whole segment tree is shared with other snapshots.
2. One by one each node is extracted from the queue and its type is checked. In the case of a leaf, the page descriptor found in its value field of the form (page id, provider) is added to the PD set. The case of an inner node is treated differently: for each of its two child nodes the first version descriptor that intersects it is sought. If such version exists, then the child node is not shared with higher available snapshots. However, it can be part of one or more older versions. This is the moment when the version map becomes useful. Note that the segment covered by each of the child nodes can be computed without the need to fetch them. Also, their versions are stored in the parent node.
3. Assuming the version number of one child node is w, if at least one snapshot within the range [w … v-1] is available, the node cannot be removed. Otherwise, GET_NODE(id, version, offset, size) fetches and returns the node identified by the supplied blob id, version, offset and size from the metadata provider. Only if all the above-mentioned conditions are fulfilled, the node is read and inserted in the queue. Step 2 is repeated until the queue becomes empty. Finally, the list of page descriptors and the KEYS set that holds the identifiers of the read metadata nodes are returned.

Algorithm 3 READ_METADATA

Require: The blob id
Require: The snapshot version v
Require: The version map VM
Require: The list of version descriptors VD
Ensure: The set of page descriptors PD
Ensure: The set of metadata nodes KEYS

PHASE 1

1: N = (key, value) ← {GET_ROOT(id, v)}
2: PD ← ϕ , KEYS ← ϕ
3: **for** (_, offset, size) ∈ VD **do**
4: **if** (key.offset, key.size) intersects (offset, size) **then**
5: Q ← {N}

PHASE 2

6: **while** Q ≠ ∅ **do**
7: N = (key, value) ← extract node from Q
8: KEYS ← KEYS ∪ {key}

```
 9:          if N is leaf then
10:              PD ← PD ∪ (value.pid, value.provider)
11:          else
```

PHASE 3 – PROCESSING THE LEFT CHILD NODE

```
12:              for (_, offset, size) ∈ VD do
13:                  if (key.offset, key.size/2) intersects (offset, size) then

14:                      for version ϵ [value.lv ... v-1] do
15:                          if (VM[version] == available) then
16:                              break
17:                          end if
18:                      end for

19:                      if (version == v) then
20:                          Q ← Q ∪ GET_NODE(id, value.lv, key.offset, key.size/2)
21:                          break
22:                      end if

23:                  end if
24:              end for
```

PHASE 3 – PROCESSING THE RIGHT CHILD NODE

```
25:              for (_, offset, size) ∈ VD do
26:                  if (key.offset + key.size/2, key.size/2) intersects (offset, size) then

27:                      for version ∈ [value.rv ... v-1] do
28:                          if (VM[version] == available) then
29:                              break
30:                          end if
31:                      end for

32:                      if (version == v) then
33:                          Q ← Q ∪ GET_NODE(id, value.rv, key.offset + key.size/2, key.size/2)
34:                          break
35:                      end if

36:                  end if
37:              end for


38:          end if
39:      end while
40:      break
41:  end if
42: end for

43: return (PD, KEYS)
```

4.7 Concurrency issues

Versioning is exploited by the concurrency control with the aim of increasing the number of operations processed in parallel. The algorithm described in the previous subsection also leverages versioning in an efficient way and liberates the user from the burden of managing synchronization explicitly. The only synchronization occurs at the version manager level where the status of each version is monitored.

Similar to other operations, deleting one snapshot should also guarantee atomicity. Each version removal should appear to take effect instantaneously at some point between its invocation and completion. Thereby, readers should not be able to access temporarily inconsistent snapshots that are in the process of being deleted. This requirement is fulfilled by notifying the version manager before starting the deletion proper so that subsequent reads for the same version will fail. Obviously, concurrent reads of other versions can freely proceed in the presence of this delete.

Regarding write-delete concurrency, as long as each new snapshot generation depends on the most recent published version, removing the lower ones does not interfere with writes. Deleting the highest version, however, requires closer consideration. *Total version ordering* guarantee was introduced in [1], which argued that if an update completes successfully and returns version number v, then snapshot v reflects the successive application of all updates labeled 1 … v on the initially empty blob. In order to preserve this total ordering, upon removing the most recent version and assuming there are no snapshots in the process of being generated that might depend on it, the version manager must eliminate the associated entries from all its data structures and also atomically decrease the global version counter for that blob.

The last case relates to concurrent deletions of several snapshots belonging to the same object. The version manager is in charge of serializing delete notifications by atomically updating the corresponding data structures and building adequate replies. Clients do not have to wait for other deletions to complete, so the next steps in the delete protocol, including reading and processing metadata, then removing metadata and data, can be performed in parallel. As a result, after all concurrent operations successfully complete, the final view of the blob will be the same, regardless of the order in which these invocations were first treated by the version manager.

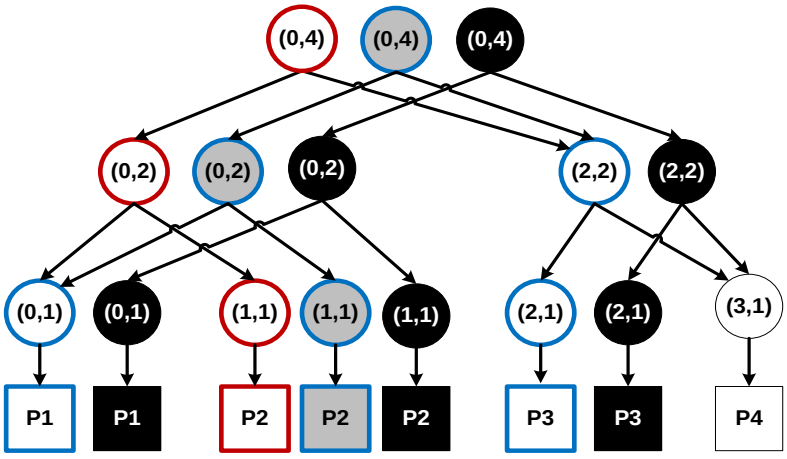


Figure 5 (a): White version deletion request first processed

In order to sustain the last statement, we consider a blob made up of four pages generated by three consecutive updates. Two concurrent deletion requests are issued: one for the white version and other for the gray snapshot. Depending on which of them is first received by the version manager, the lists of the pages and metadata nodes removed during one operation will be quite different. This way, if the white version deletion request is first processed, the data structures built by each client are: $KEYS_1$ = {(v1, 0, 4), (v1, 0, 2), (v1, 1, 1)}, PD_1 = {white P_2}, $KEYS_2$ = {(v2, 0, 4), (v2, 0, 2), (v1, 0, 1), (v2, 1, 1), (v1, 2, 2), (v1, 2, 1)} and PD_2 = {white P_1, gray P_2, white P_3}. The second client could delete white nodes and pages because he found out that a deletion demand for the first version had already been served. In Figure 5 (a), metadata nodes and pages eliminated by the first client are illustrated in red, while the other ones in blue.

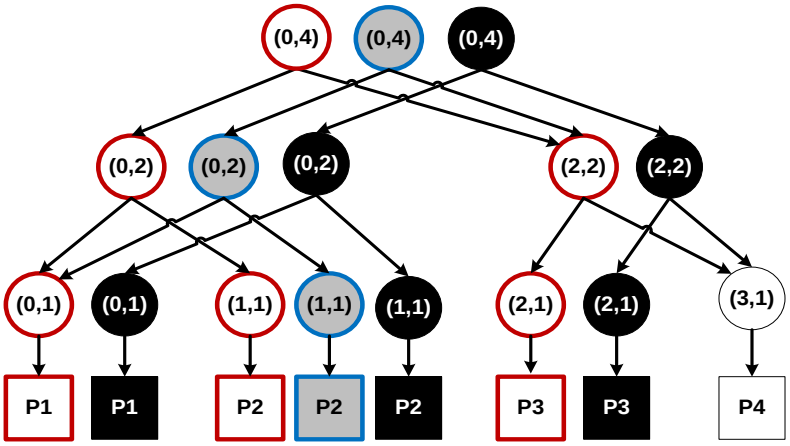


Figure 5 (b): Gray version deletion request first received

The second case depicted above assumes that the gray version deletion query is first served by the version manager. Again, the data structures built are: $KEYS_1$ = {(v1, 0, 4), (v1, 0, 2), (v1, 0, 1), (v1, 1, 1), (v1, 2, 2), (v1, 2, 1)}, PD_1 = { white P_1, white P_2, white P_3}, $KEYS_2$ = {(v2, 0, 4), (v2, 0, 2), (v2, 1, 1)} and PD_2 = { gray P_2}. This time, the first client is the one aware of the fact that a delete request for gray version has already been issued.

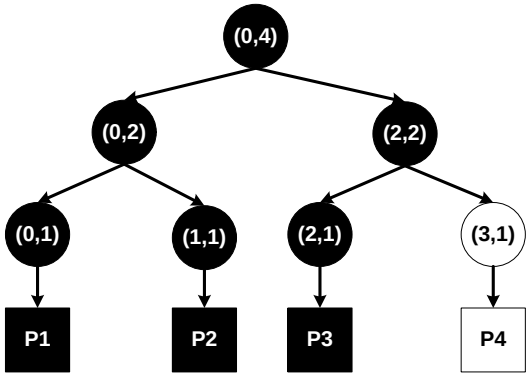


Figure 5 (c): The remaining segment tree

As mentioned above, synchronization is only needed at the beginning of the delete operation, but the overhead implied is insignificant due to the fact that the version manager is a centralized entity. Clients can afterwards traverse the distributed segment trees, delete metadata nodes and pages freely in the presence of other deletions. The reason for this relies on the fact that concurrent clients build disjoint sets, so they will never attempt to remove the same data. Back to our example, after deletion primitives successfully complete, the remaining segment tree looks like that in Figure 5 (c), regardless of which of the two queries is first processed by the version manager.

5. Implementation details

The difficulty of implementing the DELETE primitive came from the fact that on each write only the different pages with respect to the previous versions are actually added. The rest of the blob content and part of the metadata nodes are shared. As a consequence, a snapshot deletion is not assumed to free the whole associated segment tree and the pages covered. Only those metadata nodes and pages that are not shared with any available version should be eliminated.

This section introduces the data structures involved in our implementation with the help of two class diagrams. We omitted some attributes and methods of the updated classes and kept only those relevant to the chosen data deletion algorithm. Finally, used tools and libraries are discussed together with other significant implementation details.

5.1 Data structures

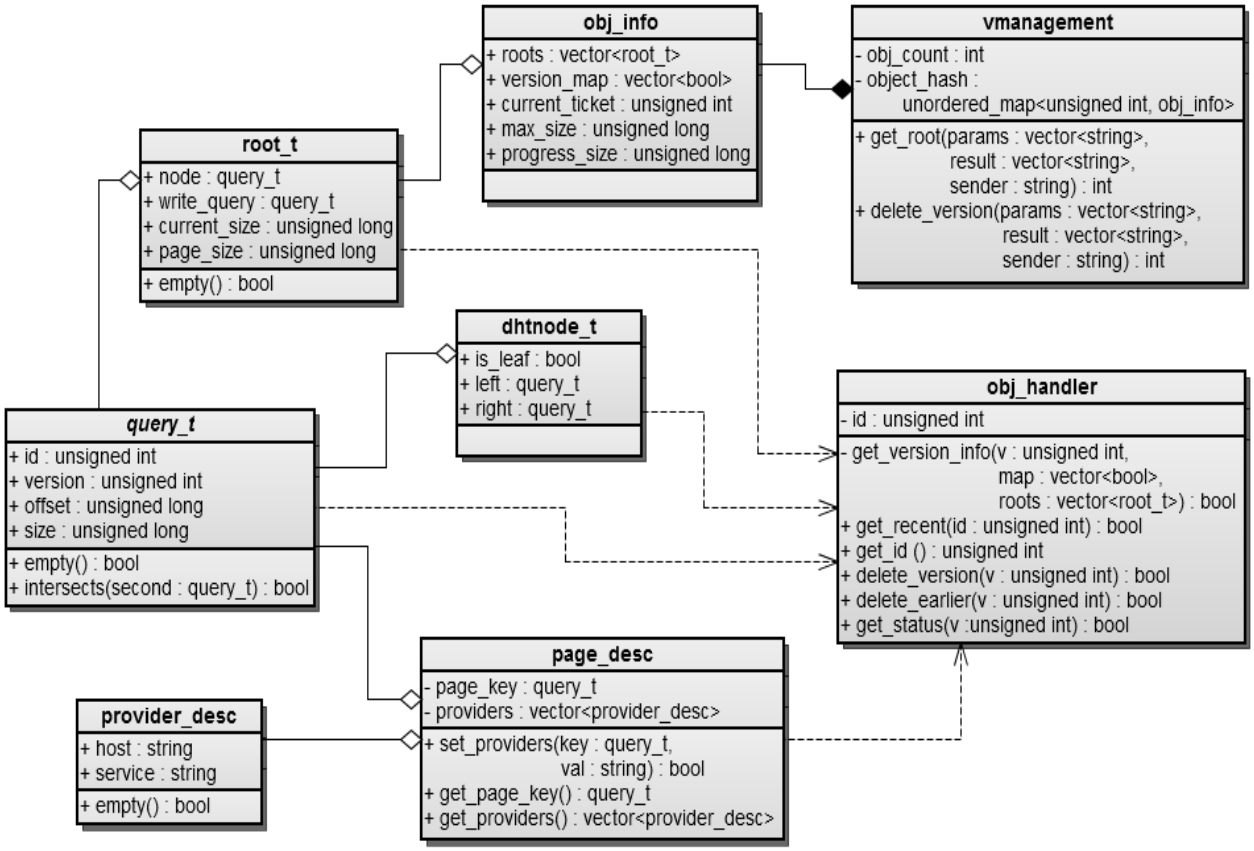


Figure 6 (a): Class diagram 1

Common data structures. The *query_t* class includes all the attributes needed to define a read or write request as well as the key that uniquely identifies a page or a metadata node: id of a given blob, version, offset and size. Thereby, an object of this type can have multiple meanings. From the *root_t* class point of view, it represents once the key of some root node and twice the write

query that generated the snapshot with that root. The *dhtnode_t* class denotes the value associated to a segment tree node consisting of the keys of its two children. Nevertheless, the *page_desc* class encapsulates a *query_t* object that identifies a certain page and the list of the providers that store it, each of them being represented as a *provider_desc* object.

Version manager data structures. The version manager keeps track of all the blobs that were created and of their versions. To this end, it stores a map of *obj_info* objects, each comprising attributes corresponding to a blob such as its size, the version counter named *current_ticket*, the vector of roots and the version map. The *delete_version* method included in the *vmanagement* class is the one executed when a delete request is received and it implements Algorithm 2 described in Section 4.6.

Client data structures. Each blob is manipulated with the help of an *obj_handler* object. It is used to create a new blob and bind to it or alternatively bind to the latest version of an existing blob by calling the *get_recent* method. Once successfully bound, the *obj_handler* can be used to read, write or append data and now to also delete a specified version. Two more methods were added: *delete_earlier* that removes all data older than a given version and *get_status* which returns true if the supplied version is available or false otherwise. All the classes mentioned so far and relationships between them are shown in the first class diagram in Figure 6 (a).

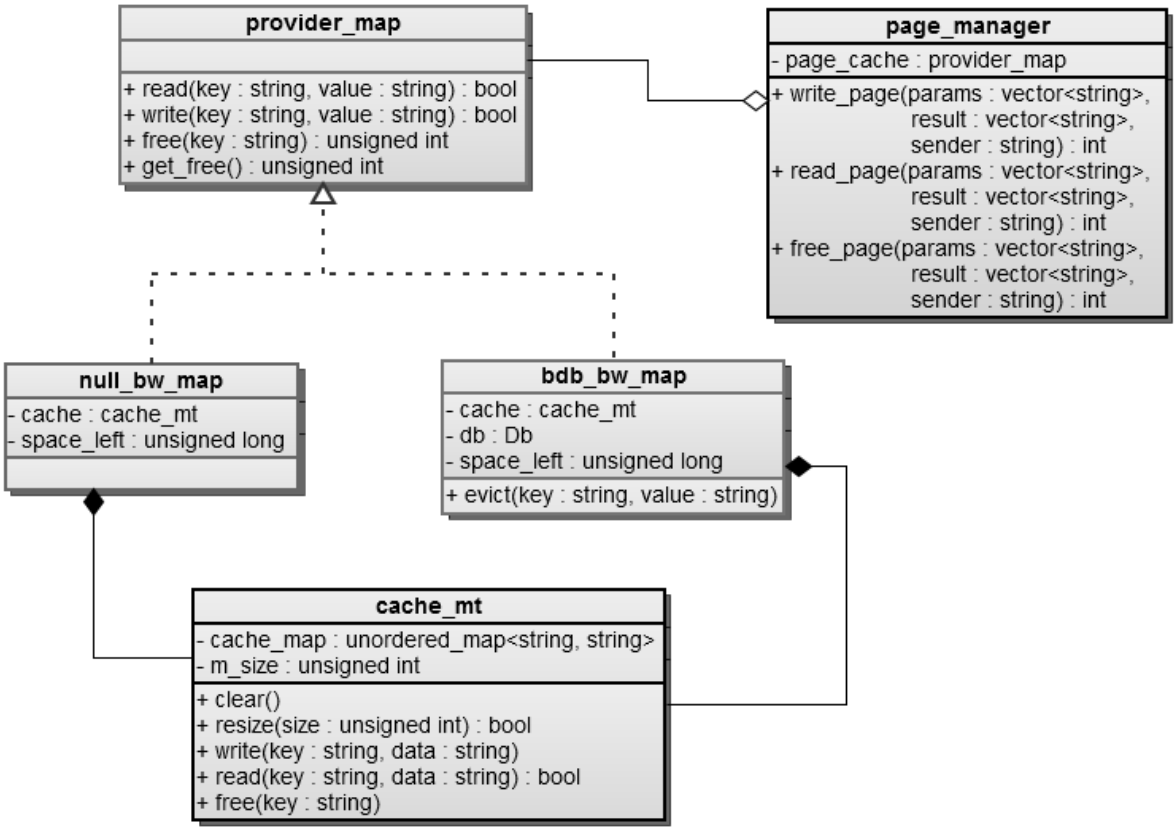


Figure 6 (b): Class diagram 2

Provider data structures. Providers were initially designed to store pages and metadata in their local RAM. For this purpose, the *null_bw_map* class wraps a cache, which is essentially a key-value data store described by the *cache_mt* class, and the common methods needed to read, write or free a cache entry. Then, a persistent layer built on top of BerkeleyDB [11] was added while keeping the initial RAM-based storage scheme as an underlying caching mechanism. The *bdb_bw_map* class reflects this change and apart from the ordinary operations, it also provides an eviction callback function that inserts a recently erased cache entry in the database. Each provider exposes a remote read/write data access interface. In order to delete pages and metadata nodes, we included the *free_page* method. All these classes instantiated by each provider process are illustrated in the second class diagram in Figure 6 (b).

5.2 Tools and libraries

In order to comply with the existing implementation, the additional primitives were written in C++ using the Boost [12] collection of meta-template libraries. We especially used the Boost Bind library favoring its uniform syntax for creating function objects and its powerful functional composition. Bind was applied on free functions as well as member methods to pass callback functions as parameters to some asynchronous primitives. Once a primitive completes, the associated callback function is called with the result as its arguments so that appropriate actions can be taken.

We also took advantage of the RPC layer responsible for mediating communication between different components of BlobSeer to add new asynchronous remote procedure calls. This way, the client initially invokes *delete_version* remotely on the version manager to notify it about the intention of removing the supplied version and to get information necessary to proceed. In addition, providers exposed a new method, *free_page*, that enables clients to issue delete requests for metadata nodes and pages.

As mentioned above, persistent key-value data store was provided based on Berkeley DB, a high-performance embedded database. Apart from being easy to use, as the library can directly be linked into the application, Berkeley DB includes important features such as: keys of any data type or structure, values of arbitrary sizes, three access methods and default functions to operate on keys. The Hash access method was chosen for BlobSeer persistent layer as insertion, deletion and look-up of records by exact match was only needed. As far as we were interested in concurrently deleting records from the database given their keys, we used the *del* method of the *Db* main class provided by the C++ interface of Berkeley DB.

5.3 Other details

We dropped the cache used to store roots of some distributed segment trees from the client side. For reasons of consistency, the client is not allowed to keep metadata of removed versions. Thus, at the beginning of each version deletion, it has to acquire the snapshot root from the version manager if the version really exists and if it is available.

We also updated the client side DHT access interface which enables clients to issue get and put queries to metadata providers. Results from these queries are cached on the client so that they can be reused later. By adding the remove function, we provided the capability to also issue asynchronous requests for metadata nodes deletion. These requests are aggregated and finally

sent to the metadata providers. At the same time, the corresponding entries in the local cache are erased for the same reason mentioned above.

On the data provider side, once a free_page request is received, the entry associated with the supplied key is eliminated from the cache and optionally, if persistent storage is supported, from the database. Then, information about the left space is updated and reported to the provider manager.

6. Test scenarios

In the first test scenario, we consider a blob with four pages and four published versions illustrated in Figure 5 below. Versions are color-coded: the initial snapshot is white, the second one is light-gray, snapshot 3 is dark-gray and the last version is black. The associated data structures stored by the version manager, the list of version descriptors and the version map, are as follows: VD_{id} = {(v1, 0, 4), (v2, 1, 2), (v3, 0, 1), (v4, 2, 2)}, VM_{id} [i] = available, i = v1 … v4. By convention and in order to mark the difference, the covered region each metadata node is labeled with is represented in round brackets, while the segments from version descriptors we added above each root are enclosed by square brackets.

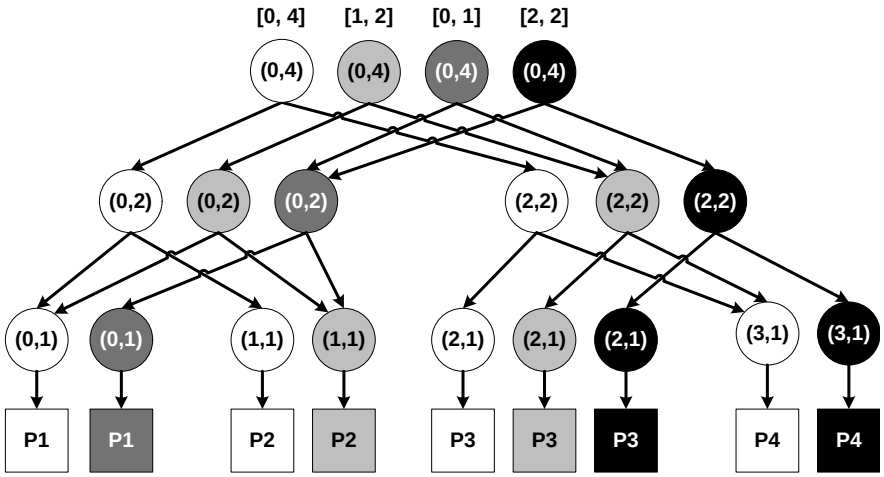


Figure 7: Data deletion scenario

On issuing a delete request for the first version, only pages 2 and 3 will be eliminated. The leaves covering the rest of the pages are directly referenced by two metadata nodes of the light gray-colored snapshot. The deletion steps can be summarized in the following way:

1. The version manager is notified of the deletion and adequately updates the version map of the blob: VM_{id} [v1] = deleted. Then it builds and returns VD = {(v2, 1, 2)} and VM = ϕ.
2. The client performs some segment intersections beginning with the root – the first read node. Because (0, 4) $\cap$ [1, 2] $\neq \phi$, the child nodes are checked: (0, 2) $\cap$ [1,2] $\neq \phi$ and also (2, 2) $\cap$ [1, 2] $\neq \phi$. Accordingly, both of them are obtained from the metadata providers and their children in turn are evaluated: (0, 1) $\cap$ [1, 2] = ϕ and (3, 1) $\cap$ [1, 2] = ϕ, which means they are referenced by inner nodes of the second version which is available and thus the regions covered, pages 1 and 4, cannot be deleted. For the remaining leaves, the intersection is not empty and therefore they will be read.
3. Finally, reaching the leaves, the stored page descriptors are added to a list and returned along with the set of read metadata nodes, which is KEYS = {(v1, 0, 4), (v1, 0, 2), (v1, 1, 1), (v1, 2, 2), (v1, 2, 1)}. All of them are outlined in Figure 6 in red. After removing them, the metadata segment trees of the blob will look like those in Figure 7.

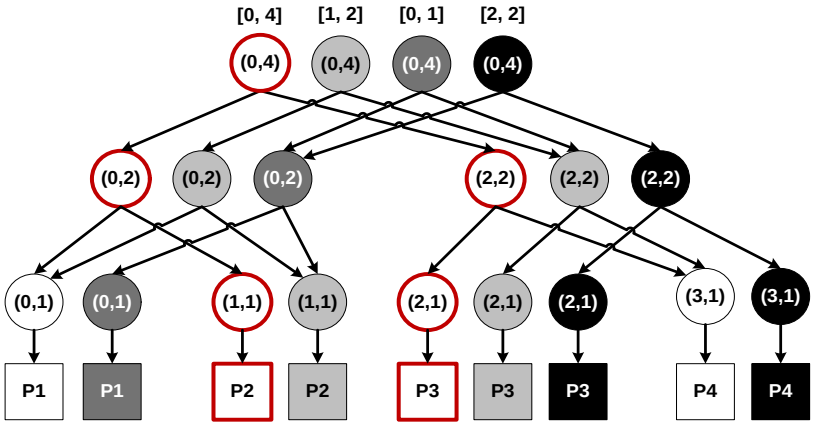


Figure 8: Deleting the first version (white)

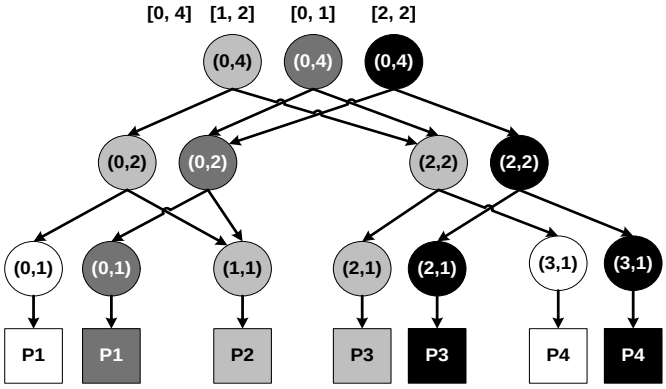


Figure 9: Metadata after deleting the first version

Next, we would like to remove the second version. Following the same algorithm as above, where VD = {(v3, 0, 1)} and VM = {(v1, deleted)}, only the first page that was also part of the white snapshot, now already deleted, will be erased. The whole right subtree of the root, however, is shared with the dark-gray snapshot. Again, we represent below the data that will be deleted and the remaining segment trees.

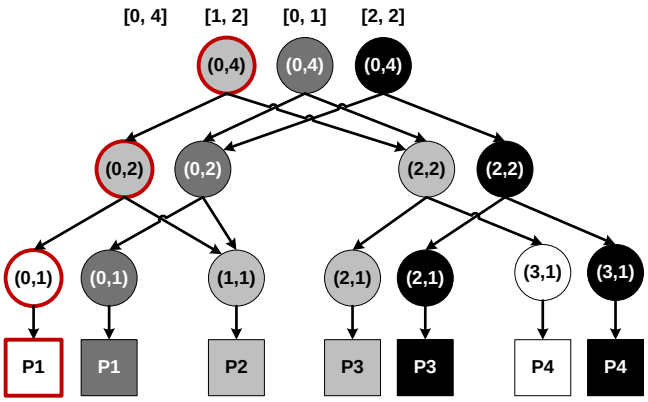


Figure 10: Deleting the second version (light-gray)

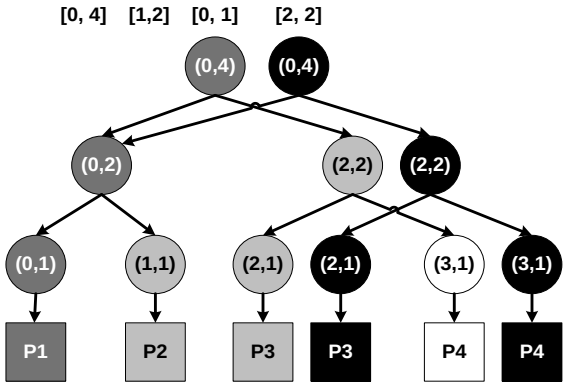


Figure 11: After light-gray version deletion

In the second scenario, we consider a more complex example and a random order of version deletions. The blob depicted in Figure 10 is also made up of four pages and four versions, but the first one consists only of two pages. Similar, data structures can be defined as VD_{id} = {(v1, 0, 2), (v2, 2, 2), (v3, 1, 2), (v4, 3, 1)} and VM_{id} [i] = available, i = v1 … v4.

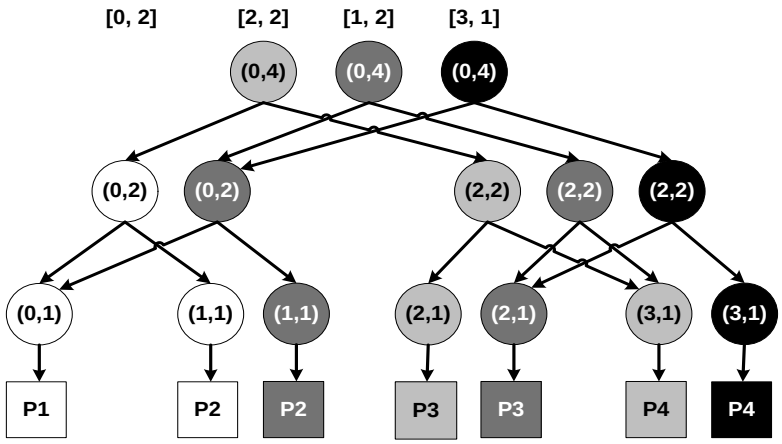


Figure 12: Version deletion scenario

First, we intend to eliminate the third version represented in dark-gray. Hence, the steps followed are:

1. The client gets the lists VD = {(v4, 3, 1)} and VM = {(v1, available), (v2, available)} from the version manager.
2. Then, it traverses the segment tree in a top-down manner to find out the nodes that intersect the region extracted from the received version descriptor. This way, no page will be reached and only the root and its right child will be deleted. The leaf identified by (v2, 3, 1) also intersects [3, 1], but its version, v2, is still available.
3. Finally, KEYS = {(v3, 0, 4), (v3, 2, 2)} is returned and the metadata providers are asked to delete them. The remaining segment trees are shown in Figure 11.

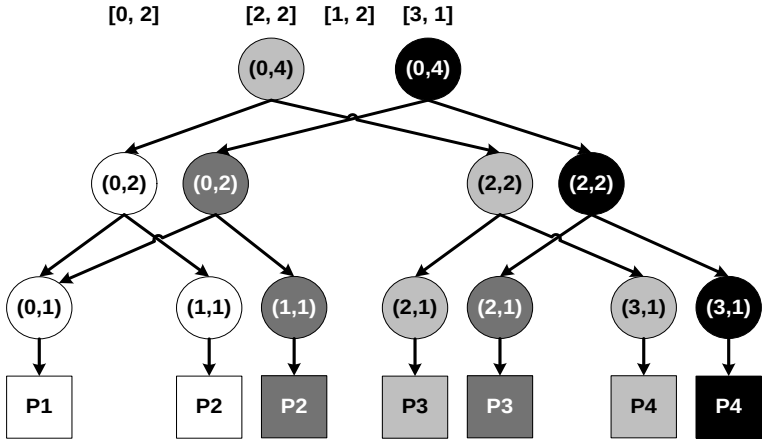


Figure 13: Metadata after first deleting the dark-gray version

Next, we proceed to erase the second version. This time, VD = {(v3, 1, 2), (v4, 3, 1)} and VM = {(v1, available)}. Processing a node involves searching through the list of version descriptors until the first that intersects the node is found. If no such descriptor exists, then the whole subtree is shared with higher snapshots. As the first descriptor intersects the root, its children are checked: (0, 2) $\cap$ [1, 2] $\neq \phi$, but version v1 is available, so the processing of the left subtree stops. Only the right node (v2, 2, 2) is read and so we go deeper in the tree. Its left half segment defined by (2, 1) intersects the first descriptor while the right one (3, 1) the second descriptor. Therefore, both pages, P3 and P4, will be deleted. Eventually, KEYS = {(v2, 0, 4), (v2, 2, 2), (v2, 2, 1), (v2, 3, 1)} and the rest of the nodes and pages are represented below.

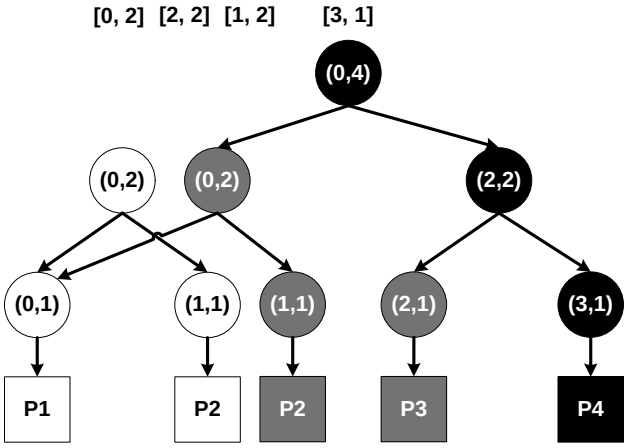


Figure 14: Metadata after light-gray version deletion

Finally, we apply the same algorithm for the first version, in which case VD = {(v2, 2, 2), (v3, 1, 2), (v4, 3, 1)} and VM = ϕ. (v3, 1, 2) is the first descriptor that intersects both the root and its right child so the page it covers, P2, will be deleted. The left node, on the other hand, is part of the last snapshot. The remaining segment tree is shown in Figure 13 below.

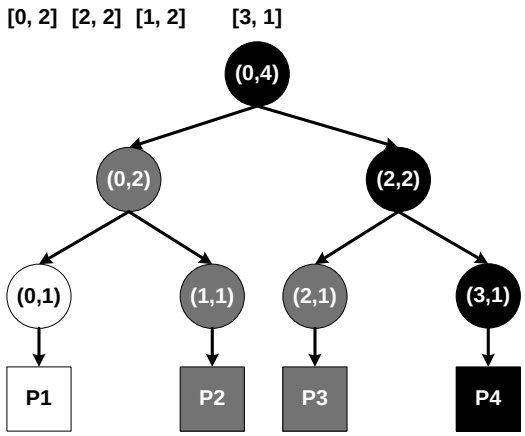


Figure 15: Metadata after deleting the first three versions

7. Experimental results

We tested the proposed algorithm on a local machine outfitted with dual-core processor and 3 GB of RAM by running all BlobSeer entities on localhost. The experiments carried out aimed at evaluating to what degree metadata trees' density influences one or multiple versions deletion. Among the observed parameters we mention deletion time and number of metadata nodes and pages removed. We ran a single instance of each BlobSeer component as we were more interested in applying our algorithm on different types of objects.

7.1 Deleting one version

We first performed a set of experiments to emphasize how version deletion time depends on the density of the distributed segment trees associated to a blob. To be more precise, the more metadata is added and interleaved due to many overwritten pages, the more it would take to eliminate one version. Three different types of blobs, depending on their metadata density, were evaluated: one featuring high density, which is equivalent to subtrees of small height being shared, the second type of medium density and the third one including more references to tall subtrees and even whole trees. All these blobs were created by applying successive updates, each time with a double interval of written data, but at different offsets, depending on the desired density. This way, we began with an offset increase equal to the size of one page, which is 1 KB, and then progressively increased it in such a way as to overwrite fewer and fewer pages, thus obtaining lower density. Since the blobs reached a size between 4 and 8 MB, we chose to delete the version that was covering approximately 2 MB. For each type of blob concerning their density, time spent in each of the three phases of the delete protocol was measured and recorded in Figure 16. Results show that removing a version from a compact blob requires more time than erasing a version of roughly the same size from a loose object.

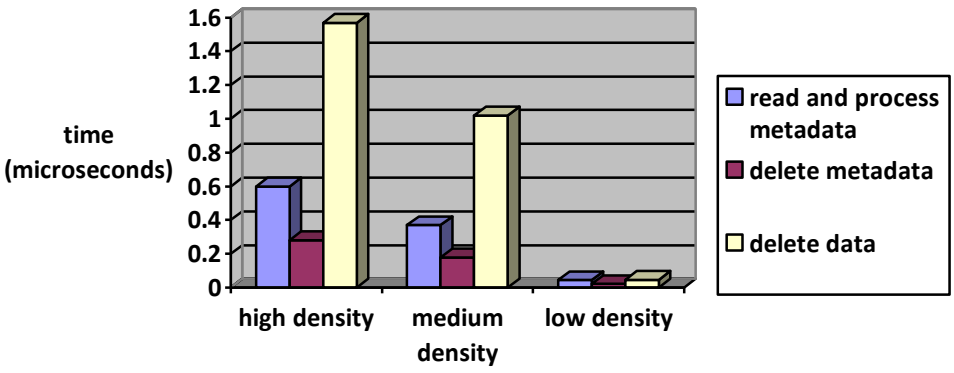


Figure 16: Deleting one version from different blobs

In order to sustain the results illustrated above, we collected information about the total number of written pages, the effective size of each blob, the number of deleted metadata nodes and deleted pages respectively. Hence, all three blobs were made up of 8188 pages, each of 1 KB,

but depending on how many pages were overwritten, their effective sizes resulted different with the most compact object having the smallest size. However, deleting one snapshot implies a decrease in the number of deleted pages and metadata nodes in inverse proportion to the effective size of the blob. The reason for this relies on the fact that high density is associated with few shared, but many overwritten pages, while low density features the opposite. Information gathered upon removing the version covering a range of 2 MB from each blob is shown in the table below.

	Total number of pages	Effective size of the blob	Number of deleted pages	Number of deleted metadata nodes
High density	8188	4106 KB	2047	6151
Medium density	8188	5632 KB	1280	3842
Low density	8188	8064 KB	32	100

Figure 17: Density influence on arbitrary version deletion

7.2 Deleting multiple versions

In the second experiment, we tested the DELETE_EARLIER primitive by removing data which is older than a specified version. The previous setting was used and the same three types of blobs were evaluated in order to see to what degree data density influences the number of deleted metadata nodes and pages while removing more and more versions. We recall that with each update a double data range was inserted. This is why in the case of the compact blob, the number of deleted pages represented in Figure 18 below approximately doubles with each version additionally removed.

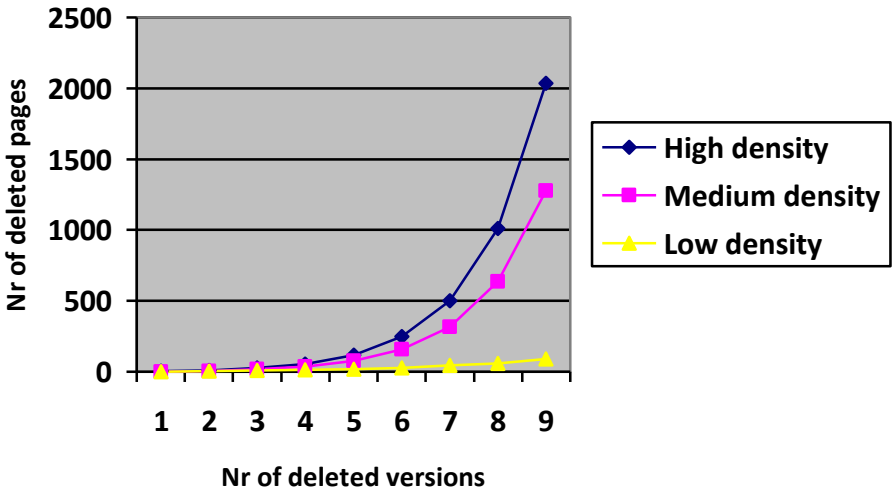


Figure 18: Number of deleted pages / Number of deleted versions

Results reflect the same idea explained in the previous section: deleting versions from a dense blob leads to eliminating more pages than in the case of a less compact object. In a similar way, we recorded the number of removed metadata nodes. Although their number is by far greater, the graph exposes the same shape.

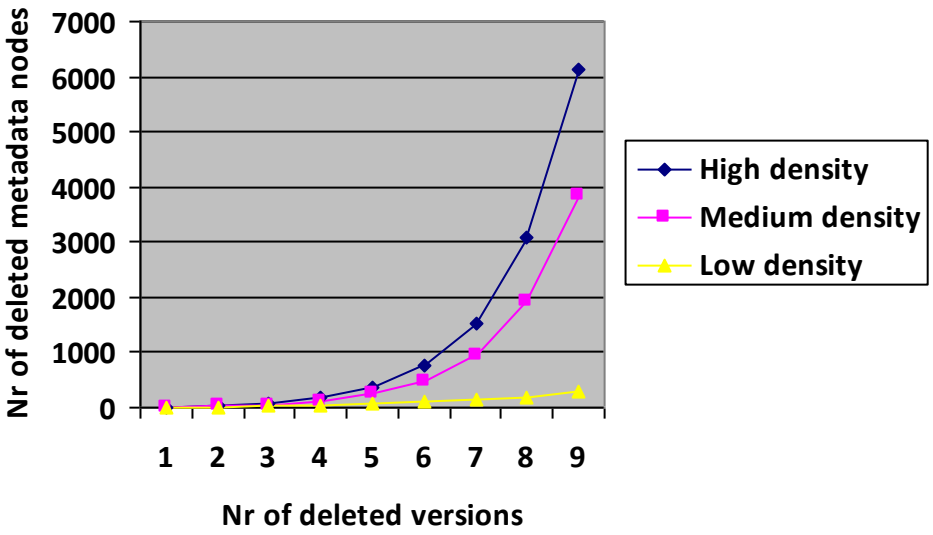


Figure 19: Number of deleted metadata nodes / Number of deleted versions

8. Conclusions and future work

8.1 Contributions

This thesis proposed an efficient scheme for BlobSeer data deletion in order to free valuable disk space. We added the capability to delete a version of a given blob and data which is older than a specified version. This feature can be used to eliminate false, unnecessary or accidentally written data.

In designing the version deletion algorithm, we aimed at guaranteeing total ordering and atomicity with little impact on the rest of the primitives. Versioning was leveraged to avoid synchronization as much as possible, both at data and metadata level, so that deletions can freely proceed in parallel with other operations.

8.2 Future work

Although the current delete implementation efficiently addresses concurrency issues, some improvements related to fault tolerance should be considered. If a delete operation fails in one of the last two phases - metadata or data deletion – no inconsistency may be created since the version manager has already marked the version as removed and subsequent reads will not be served. However, the remaining unreferenced metadata and pages should be reported and then eliminated. One further step would entail providing the support for identifying such data.

References

[1] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *J. Parallel Distrib. Comput.*, vol. 71, pp. 169-184, 2011.
[2] 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.
[3] The Hadoop Distributed File System. http://hadoop.apache.org/core/docs/current/hdfs_design.html
[4] Ghemawat, S., Gobioff, H., Leung, S.T., "The Google file system", ACM SIGOPS Operating Systems Review, Volume 37 , Issue 5, pp. 29-43, December, 2003.
[5] B. Nicolae, G. Antoniu, and L. Bougé, "BlobSeer: Efficient Data Management for Data-Intensive Applications Distributed at Large Scale," in *Proc. IPDPS '10: 24th IEEE International Symposium on Parallel and Distributed Processing: Workshops and Phd Forum*, Atlanta, USA, 2010, pp. 1-4.
[6] http://aws.amazon.com/s3/
[7] B. Nicolae, G. Antoniu, and L. Bougé, "Enabling lock-free concurrent fine-grain access to massive distributed data: Application to supernovae detection," in *Proc. Cluster '08: 10th IEEE International Conference on Cluster Computing*, Tsukuba, Japan, 2008, pp. 310-315.
[8] B. Nicolae, G. Antoniu, and L. Bougé, "BlobSeer: How to enable efficient versioning for large object storage under heavy access concurrency," in *Proc. EDBT/ICDT '09 Workshops*, Saint-Petersburg, Russia, 2009, pp. 18-25.
[9] C. Băsescu, C. Leordeanu, A. Costan, A. Carpen-Amarie, and G. Antoniu, “Managing Data Access on Clouds: A Generic Framework for Enforcing Security Policies,” in *International Conference on Advanced Information Networking and Applications (AINA)*, 2011.
[10] A. Carpen-Amarie, J. Cai, A. Costan, G. Antoniu, and L. Bougé, “Bringing introspection into the BlobSeer data-management system using the MonALISA distributed monitoring framework,” in *International Workshop on Autonomic Distributed Systems*, Krakow, Poland, 2009.
[11] Michael A. Olson, Keith Bostic, and Margo Seltzer, “Berkeley DB,” in *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference,* Monterey, California, USA, June 6–11, 1999.
[12] http://www.boost.org/doc