ABSTRACT

WANG, ZHUOWEI. Design and Implementation of a Distributed Snapshot File System. (Under the direction of Vincent W. Freeh.)

File system is an important component of a operating system. It defines the way data is stored and retrieved. Its performance and efficiency affect overall operating system.

In this study, we design, implement, and validate a new file system built-on FUSE. The file system is a distributed file system with snapshot capability,that use MongoDB as storage backend. We introduce a patch-based new strategy to store snapshots. Further, we apply the rsync algorithm to enhance the efficiency of classic copy-on-write snapshot as well as the network performance. In addition, we studied the performance and the effect of some factors that effect the efficiency of rsync enhanced copy-on-write snapshot.

Test results indicate that the file system we proposed has performance comparable to the popular network file system NFS. Furthermore, it shows that the rsync enhanced copy-on-write snapshot system can boost the space efficiency of the snapshot system compared to classic copy-on-write snapshot system.

Design and Implementation
of
a Distributed Snapshot File System

by
Zhuowei Wang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2014

APPROVED BY:

Rada Y. Chirkova	Emerson Murphy-Hill

Vincent W. Freeh
Chair of Advisory Committee

DEDICATION

...

BIOGRAPHY

...

ACKNOWLEDGEMENTS

I would like to thank my advisor for his help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Mention "file system" and what usually comes to mind is a disk file system like FAT32 [34] or Ext4 [19], because the disk is the most common persistent storage. File systems can be used to access data from remote machine like NFS [24], can be used to access non-traditional form of data and storage space like GmailFS [15, 5], can provide useful and transparent functionalities like encryption, compression or soft-RAID [3, 18].

As the network bandwidth increases, we no longer have to keep our data on local disk. Nowadays more and more data are stored online. Individuals are uploading their personal data to web services, commercial companies are using systems like Amazon S3 to store their data remotely. There are a large number of reasons to store data in the cloud. It makes sharing and collaboration easier and ensures that the data can be accessed by its owner anywhere and anytime. Though there are lots of services and applications that help us to access data online, a network file system is able to make this process totally transparent to the user and user programs. A distributed file system can also maps different machines into a logically unified volume while providing features like redundancy and load balancing. Popular distributed file systems adopted by industry and academic institutes include NFS and AFS [11].

The NoSQL database [29] is used for store and retrieve data which is modeled differently from the tabular relations. It provides an alternative to relational databases, brings innovation and new ideas to the community, industry and academia. They have become alternatives for classic relational database. Commercial companies like Google and Amazon have been using them in production environments for many years. Compared to relational database, they have advantages in performance, scalability and flexibility whereas low functionality, e.g. not normalized data and poor transaction support [20], is their disadvantage.

We believe a document oriented NoSQL database [12] can serve as a backend of the file system to save and manage the actual data. Because a file system needs to ensure consistency and query single entity (file, directory) but not much demand of join queries and join updates.

A document oriented NoSQL database that focuses on performance, scalability and flexibility rather than the relational algebra may be a better fit to this task. Furthermore, the high scalability of NoSQL database makes it easy to distribute the storage on to different machine [28].

Furthermore, when the file system is released from the burden of on disk resource management by using a database as backend, we can put more efforts towards other features of the file system such as snapshot and deduplication. A snapshot subsystem is useful. It can be used to test installations, keep track of modifications, provide a rollback mechanism, and make the backup process more efficient. It is becoming more and more popular in modern file system to have the capability to take snapshots. For instance, from FAT to NTFS [32] and from Ext2 [33] to Ext4, more and more file systems have this feature built-in. Ground breaking file system ZFS [36] and experimental file system BTRFS [23] all come with snapshot capability. Deduplication is another popular feature, where the file system try to identify duplicated data and eliminate them in order to save storage space.

1.1 Contribution

We design a new distributed snapshot file system and study the way to improve its performance. We use MongoDB as a backend of the file system to improve the network performance and reliability. We introduce rsync algorithm into the file system and use it to improve the network performance and snapshot efficiency, by identifying the duplicated data between remote and local and between snapshots. In addition, we use a new design of snapshot system which is called patch-based snapshot.

We vaildate our design by implemenating it and testing it. The test results indicate that the file system we proposed has a comparable performance to the popular network file system NFS. Furthermore, it shows that the rsync enhanced copy-on-write snapshot system can boost the space efficiency of the snapshot system compared to classic copy-on-write snapshot system.

In this thesis, we propose a design of a new distributed file system with snapshot capability. We have the design implemented by Java, MongoDB, FUSE, and other techniques. We also propose several new ways to improve the efficiency of the file system. We tested and compared the performance of our implantation with other file systems, studied the snapshot system performance under different scenarios, compare and explain the outcome of tests.

Chapter 2

Related work

There are important prior work in customized file systems, NoSQL database, and snapshots. Several of them are related to this thesis and they are summarized in this chapter.

2.1 POSIX

POSIX (Portable Operating System Interface), is a family of IEEE standard for operating systems [35]. It influences the design of many modern operating system (e.g., Linux, Windows NT, Mac). The POSIX standard defines a standard environment for operating system (process, user, file, directory, etc.) along with a set of APIs for user program (like fork, exec, I/O functions, etc.) [9]. Some file format standard (e.g., tar) and some shell utilities (e.g. awk, vi) are also included in the standard. The purpose of this standard is to maintain compatibility between operating system such that a user program written for one POSIX operating system will work on any POSIX operating system theoretically.

The POSIX standard specifies a set of file system APIs that defines file and directory operations. Any file system that implements this set of APIs should be compatible with operating systems that follows POSIX standard.

2.2 FUSE

FUSE, the Filesystem in User Space, is a developer framework for file system. It was originally developed for AVFS (virtual file system that allows all users to look inside archived or compressed files, or access remote files) [25] but has become a separate project. It has now been adopted into the Linux kernel and has many ports on other Unix-like operating systems. FUSE provides a programming interface that is very similar to POSIX file operations. A file system with this interface can get file system call from kernel module VFS (the Virtual File

System) [7]. Such that a user programs is able to use standard file system call to access it as if the file system is supported by operating system natively. By running the file system in user space, FUSE isolates the file system from operating system. In such a way, developers of the file system do not need to understand complicated kernel code or to debug in the kernel [6].

FUSE routes the file system call from VFS in kernel space back to the user space and then lets the user program to process the file system call. It is a widely used component of Linux file system. A number of well-known projects are using FUSE, for example the OverlayFS, ntfs-3g, and vmware tools. Inspired by FUSE, there are some other user space file system like Dokan under windows.

There are different language bindings of FUSE. Our implementation is built on the Java bindings FUSE-JNA. The FUSE-JNA is a recent active project developed by Etienne Perot. It describes itself as "No-nonsense, actually-working Java bindings to FUSE using JNA" [21]. Other Java binding of FUSE include FUSE-J, jnetfs [37].

2.3 MongoDB

MongoDB is a document oriented NoSQL distributed database [14]. It focuses on scalability, performance and availability [14]. Document oriented storage that can store semi-structured data makes it flexible and makes it suitable for agile development [12]. In addition, MongoDB also provides features like load balancing and replication. These features make MongoDB an ideal backend for a distributed file system. Developers have already built a file storage system called GridFS using MongoDB which provides a mechanism to store and retrieve file of any size [22].

MongoDB represents document in JSON (JavaScript Object Notation) format. JSON is a open, human and machine-readable standard that facilitates data interchange. Behind the scenes, MongoDB uses BSON (Binary JSON) to encode and store the documents. Both JSON and BSON format supports embedding object and arrays within other objects and arrays. MongoDB can query and build index not only on top level keys but also nested objects. The developer believes this will grant users of MongoDB ease of use and flexibility together with the speed and richness of a lightweight binary format [13].

MongoDB is one of the most popular NoSQL database. It has a large and active developer community. A recent $150 million investment ensures the long term support and reflects the confidence of investors.

2.4 Snapshot

A snapshot is a state of a system at a particular point in timer [16]. They can be either read-only or writable. Writable snapshots are also called clones. They are supported by several advanced file system like ZFS and BTRFS. The Ext4 file system also has a development branch for writable snapshot. From the users point of view, a read-only snapshot is an immutable and exact copy of the file system at a specific time, whereas a writable snapshot is a fork of the file system at a particular time spot. A writable snapshot allows modification to snapshots but modifications to a snapshot are separate from the origin and other snapshots. (i.e., changes to the snapshot cannot be seen in other snapshots or the origin and vice versa) With the help of writable snapshot, one can easily create and switch between different file system branch and make changes to them without affecting other branch. Writable snapshots can also provide file system isolation for process, software or virtual machine.

2.5 Copy-on-Write

Copy-on-write is a strategy widely used in computer science. A program that uses copy-on-write strategy accesses data though a pointer or a reference. When a copy operation is requested, instead of making a copy of actual data, a program that uses copy-on-write strategy will simply return a new reference to the original data. Only when a modification to one of the "copies" is requested, the program will then make a actual copy of the original data and then apply the modification to that copy. At the end, the program will point corresponding pointer or reference to the updated copy. This design not only eliminates unnecessary overhead in data copy but also ensures consistency, integrity and an easy support of transactions. In addition to those benefits, a file system using copy-on-write strategy will be able to take snapshots with low overhead. Without copy-on-write strategy, snapshots either write in-place which is more expensive, or requires special architecture in storage system like the Split-Mirror architecture.

2.6 Existing Snapshot Storage Systems

Snapshot is an important features of a storage system. Existing works include LVM snapshot, SnapFS, OverlayFS, ZFS, BTRFS, and Ext4.

The LVM snapshot is a snapshot system included in the logical volume manager [30, 26]. It takes snapshot of blocks in the logical volume and gives snapshots capability to any file system build upon the LVM system. However, as an underlying service, LVM snapshot is not aware of file structures, making it hard to find duplicate data stream and thus is less efficient in terms of disk space.

SnapFS is a file system focusing on snapshots in Linux kernel [27]. It is strongly coupled with its underlying Ext file system. Therefore, SnapFS is able to apply copy-on-write strategy on block but restrict its underlying file system to Ext file system family.

OverlayFS [4], also called UnionFS, is a file system popular in embedded systems, hand held devices, PDAs and smartphones. Similar to SnapFS, OverlayFS is also built upon other file systems. But OverlayFS does not restrict the type of its underlying file system as SnapFS does. This gives OverlayFS a lot flexibility as it can adapt to any storage media that has a standard file system implemented. The OverlayFS applies copy-on-write on files and directory level.

BTRFS [23] is an experimental file system developed by Oracle. It is inspired by the well-known Solaris file system ZFS and shares a lot of similarity with ZFS. Both of them usese copy-on-write snapshots at block level granularity [16].

Ext4 file system was extends to include a snapshot subsystem that applies copy-on-write strategy on block level. A writable snapshot feature for Ext4 is currently under development by the same group of developer [8]. It is inspired by CTERA Network's NEXT3 file system which is a clone of ext3 file system with built-in support for snapshots.

2.7 The Rsync Algorithm

The rsync algorithm is used in the rsync utility in Unix-like system to synchronize files though network. This algorithm is invented by Andrew Tridgell and he described the algorithm as "for the efficient update of data over a high latency and low bandwidth link" [31]. The algorithm calculate and compare a weak checksum of blocks between remote file and local file to identify duplicated blocks. In order to achieve efficiency, the checksum algorithm takes only O(1) time complexity to calculate based on prior results, such that checking duplication at any offset in local file is possible. The algorithm performs well in delta encoding and reduces the data transferred between remote machines.

Chapter 3

The Kabi File System

This chapter discusses the design and implementation of the Kabi File System. It describes the system layout and its advantages. It will also describes the design choices and important implementation details of the file system entities and operations.

3.1 System Modules and Layout

As shown in Figure 3.1, the Kabi File System has two internal modules and an internal interface in-between. The design is for replaceable modules. The internal interface defines some primary data structures and a set of standard operations on these data structures. The file system logic module decomposes any incoming file system call to a series of standard operations. While the storage abstraction module abstracts the underlying storage media to the primary data structures and exposes a set of method that implements the standard operations.

The user program communicates with the file system logic module through operating system's file system APIs. Therefore, we can have different implementations of file system logic module for different file system API (e.g., FUSE, Dokan, POSIX). The storage abstraction module operates the underlying storage media through its driver, protocol or API. We can also have different implementations of storage abstraction module for different storage media (e.g. MongoDB, Volume manager, Amazon S3 service). By using this design, we can easily migrate the file system to other operating system (by changing the file system logic module) or other underlying storage media (by replacing the storage abstraction module). In our proof-of-concept implementation, the file system logic module is built on FUSE and the storage abstraction module is written for MongoDB.

The system diagram in Figure 3.2 shows the relationship between the file system, MongoDB and operating system in this implementation. FUSE is used to connect the user program file system calls to the Kabi File System. A file system call from a user program will first be

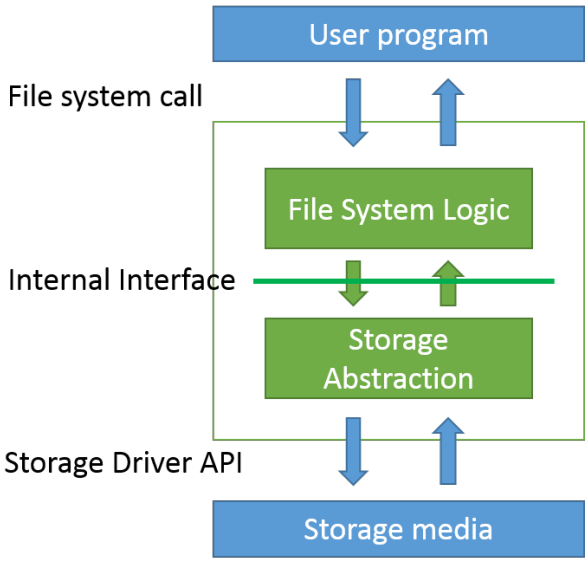

Figure 3.1: Modules of Kabi File System

Data structures	Remark
Block Node	Stores file data trunk
File Node	Stores the metadata of a file
Directory Node	Stores metadata of a directory
Snapshot Node	Stores metadata of a snapshot

Table 3.1: Primary Data Structures

captured by VFS module in the kernel and routed to FUSE. FUSE will then pass on the file system call to the file system and pop the return value back to VFS module. On the other side, the MongoDB Java driver is used by system abstraction module to communicate with the MongoDB daemon process.

3.2 File System Initialization

On file system initialization, parameters related to this instance of Kabi File system must be provided and be stored in a MongoDB collection. For example, the size of data block in bytes and the name of the MongoDB collection used by the file system. These parameters are immutable once the file system is established.

On file system mount operation, the mount program requires a JSON format configuration file. The configuration file contains parameters related to this mount operation. It has three

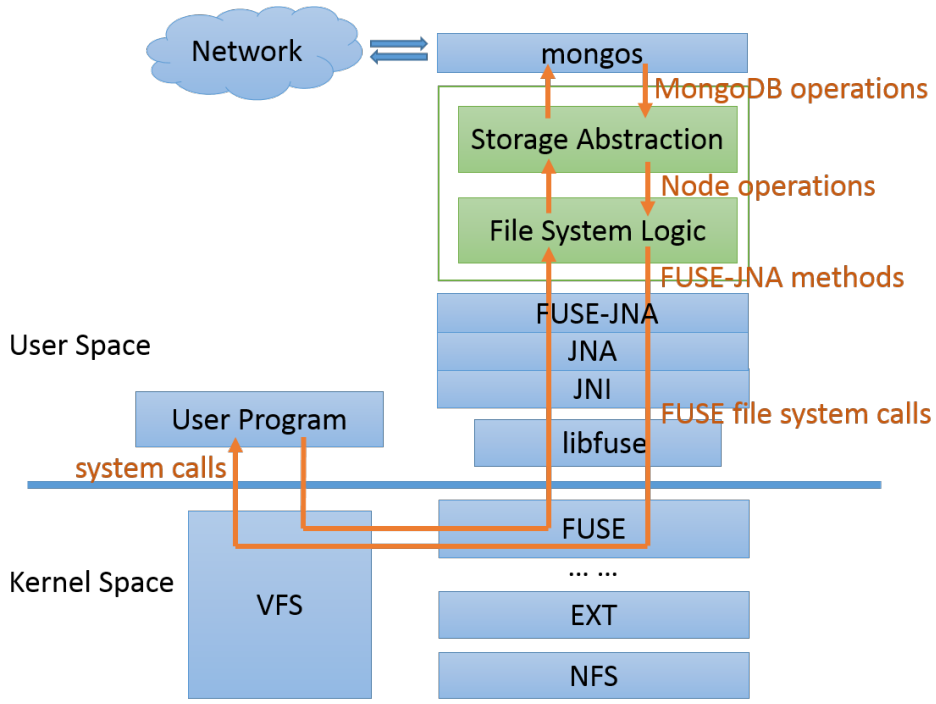

Figure 3.2: System Diagram of Kabi File System

sections, which specify the FUSE options, MongoDB options, and file system client options. These parameters are important to know about the data source (such as the IP address and server status of MongoDB server), the snapshot to be mounted and the name of the MongoDB collection where the initialization parameter is written. These parameters only affect this mount on the local machine and do not have influence on the remote server.

3.3 Nodes and Objects

Nodes are the primary data structures in the Kabi File System. A node is stored as a document in MongoDB and each type of node corresponds to a collection in MongoDB. Table 3.1 shows all four types of node used in the Kabi File System. A node can have a complex structure. A member of a node structure can be as simple as an integer but also can be as complex as an array of dictionaries. For the convenience of discussion, complex substructures (embedded objects [13]) will be discussed separately. These substructures are part of the node structure and will be called "object" instead of "node". The section object and the subnode object used by the Kabi File System will be discussed in this chapter and the patch object will be discussed in Chapter 4.

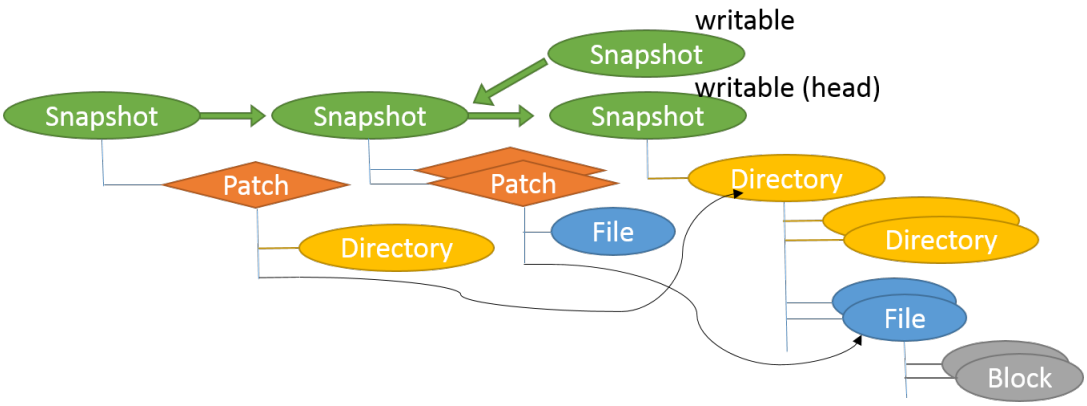

Figure 3.3: Basic Entities and Relations

field	remark
ID	the 128-bit SHA hash of block data
data	the block data

Table 3.2: Fields in Block Node

Figure 3.3 shows a basic view of the nodes and their relations in the Kabi File System. It consists of four nodes and an object: the block node, the directory node, the file node, the snapshot node, and the patch object. They refer to each other and form a directed acyclic graph.

3.3.1 Block Nodes

A block node is a representation of fixed-size data. That size of data represented by the node is called representation size. The rep-size is an immutable parameter which is determined on file system initialization. Different instance of Kabi file system can have different rep-size. Although rep-size is fixed, a block node does not have to actually store that amount. If the size of data stored in block node is smaller than rep-size, i.e. there's insufficient data, the file system logic module will get the data with '\0' padding from the storage abstraction module. The fixed rep-size is to ensure that each rolling hash correspond to exact one block node since the rolling hash is a function on fixed length data. While the variable actual data length is designed to make "truncated section" (discussed in Chapter 4) more efficient. In addition to the data field, the other filed of a block node is a 128-bit hash of the byte string. This field is for deduplication and will be discussed in later section. The structure of a typical block node is shown in Table 3.2.

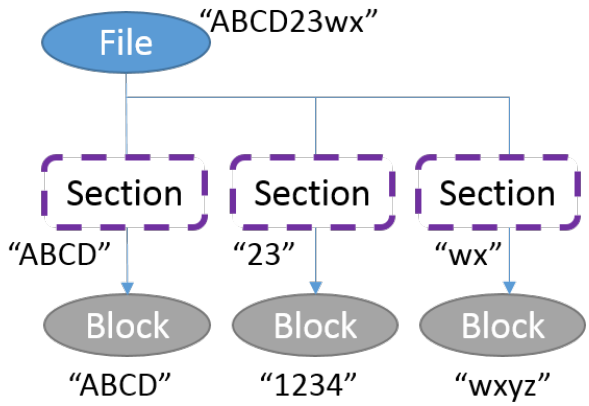

Figure 3.4: File Node, Section Object and Block Node

3.3.2 File Node and Section Object

Each file in the Kabi File System is associated with a file node that holds the metadata of the file. In the Kabi File System, the actual data of the file is stored in sections with variable length. Each section is represented through a “section object”. A file node consists of an array of section objects and other fields. Connecting the content of data section in order forms the content of the file. Each data section corresponds to exactly one block node and the block node may contribute only part of its binary data to the data section instead of its entire binary data. If only a part of the block node data is used by the data section, two integers will be specified to locate the start index and end index. This design is for reusing the block node in the snapshot system. By such design, if a block node is partly overwritten and the original block node is saved for snapshot system, we can use part of the original data to represent part of the content in file. Figure 3.4 shows a file node that consist of 3 data sections where the first data section contributes all 4 bytes of corresponding block node data, the second section gets 2 bytes from its corresponding block node and the last section receives the first 2 bytes from its corresponding block node. The detailed structure of a file node is shown in Table 3.3.

As mentioned above, a file node has an array of section object. Each section object is a substructure of file node and represents a section of file data. It consists of three fields as shown in Table 3.4: an object ID referring to the associate block node, an integer value specifies the number of omitted leading bytes, and another integer value specifying the index of last byte in block node. In the example shown in Figure 3.5, for the second block, the number of omitted leading bytes is 1, and the index of the last byte in block is 5.

The size field stores an integer value, representing the size of this file in bytes. When reading the file, ‘\0’ padding will be append to the end of files in read buffer if the value in the “size”

field	remark
mode	access mode of the directory
arc	a list of Section object
size	size of the file
owner	owner of the directory
gowner	group owner of the directory
modified	timestamp of last modification

Table 3.3: Fields in File Node

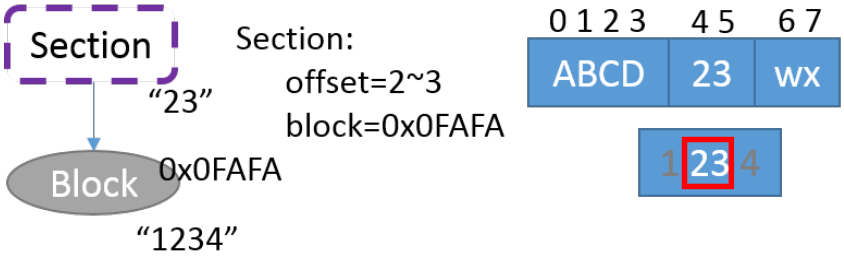

Figure 3.5: Section Object and Block Node

field exceed the total number of bytes in its data sections. On the contrary, if the total number of bytes in data sections exceeds the value of the "size" field, data will be truncated and the data exceeds the size will not be noticed by end user. This design allows faster creation and truncate operation to a file.

3.3.3 Directory Node and Subnode Object

A directory node corresponds to a directory in the file system. Directory nodes share similar structures with the file node, it also consists of fields that store the metadata and an array of subnode objects. The subnode object is similar to the section object in file node. A subnode object corresponds to an "item" under the directory. A subnode object contains a reference to the "item" and a string represents the display name of the "item". Depending on the type of node that a subnode object is referencing, the "item" can be a file under the directory or a sub directory.

Other important data structures are snapshot node and patch object. These two structures are related to the snapshot system and will be discussed in Chapter 4.

field	remark
ID	ID of corresponding block
roll	the 32-bit rolling hash of block data
omit	specify how many leading bytes in block will be omitted
offset	the offset of last byte in this section

Table 3.4: Fields in Section Object

field	remark
mode	access mode of the directory
arc	a list of Subnode object
owner	owner of the directory
gowner	group owner of the directory
modified	timestamp for last modification

Table 3.5: Fields in Directory Node

3.4 File System Operations

The file system logic module decomposes the FUSE file system calls to standard node operations defined by the internal interface. Most of the FUSE file system calls have been implemented in the Kabi File System, some of those important calls include access(), getattr(), read(), write(), rmdir(), unlink(), mkdir(), truncate(), flush(), open() and release(). Related node operations are addDirNode2db(), addFileNode2db(), addDataNode2db() and patch(). The former three methods write a node object into the database as a new node. The patch() method replaces an existing node with its new version.

3.4.1 Read Operations

Three important read file system calls are access(), getattr(), and read(). The access() function tests the existence and permission settings of a file or directory in the file system. getattr() returns the meta information about a file or directory. The read() function reads and returns binary data of given length starting at a specified offset.

The first step of a read operation is finding the target file or directory node by its path. In order to do so, the file system parses the given path and finds all corresponding nodes on the path in order from root directory to the target. The file system will start with the root directory, continue traversing subnode list and find directories on the path one by one until the

field	remark
name	display name of the file or subdirectory
ID	ID of the file or subdirectory

Table 3.6: Fields in Subnode Object

algorithm hit the target or find it nowhere.

This strategy is generally not satisfactory. Because it may introduce a performance bottleneck into the file system read operation when the target lies deep in the directory tree. In such case, a simple access() call will be mapped to a sequence of database queries on directory nodes. To solve this issue, a cache that stores the path and corresponding node ID is introduced. It can be a cache in memory when the file system is mounted locally with FUSE's single thread option or a cache in MongoDB when multiple threads or client for consistency. With the help of the path cache, the file system no longer needs to traverse and find every directory node on the path.

The path cache is a move-to-front dictionary [10], which maps a path string to a integer which represents an ID of directory node or a file node. The cache has a fixed capacity and assumes temporal locality in the access pattern. It uses move-to-front algorithm to keep the ID of frequently accessed hot item in the cache and remove those cold items that have not been accessed for long. The cache also assumes spatial locality: when accessing a file or directory, not only the ID of corresponding file or directory itself is cached but all directories on the path, i.e., its parent and ancestor directories, also go into the cache. Thus, when touching contents under the same directory later, the file system can quickly find its parent node in cache.

Once the file node is retrieved, reading the file become straight forward. The file system will traverse the section object array to find the ID of block node associated with the read call. It will then calculate the begin and end index of the bytes of the block content that will be copied to read buffer. At last the file system will query MongoDB for block nodes and copy requested data to buffer.

3.4.2 Write Operations

The Kabi File System write files on file close or explicit flush. It uses write() call to write data into a file from some offset. Other write calls like chmod(), chown(), mkdir(), unlink() and rmdir() change the meta information of a directory or file.

Block nodes are designed to be immutable. File nodes and directory nodes use copy-on-write strategy when an overwrite is requested. So, when an overwrite to file node or directory node

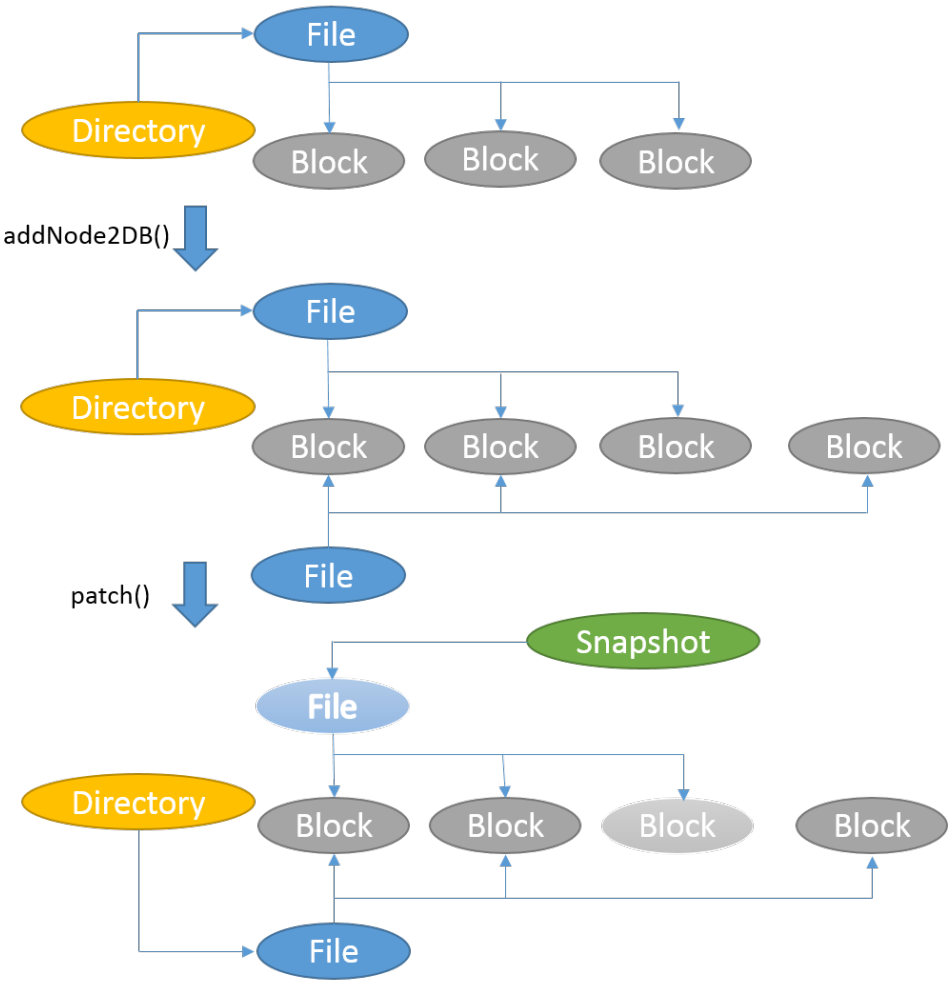

Figure 3.6: Copy-on-Write

is requested, the file system will make a local copy of the node and apply the modification to this local copy. The file system will then upload that copy to remote and attach it to its parent directory replacing the original node as shown in the Figure 3.6. In this way, the old version of the node is saved for snapshot and read operation will not accidentally read in a node that is in an incomplete state. As shown in Figure 3.6, during this process, we use addNode2DB() operation to add a new node and use the patch() method to replace the old version with this new node.

When writing a file, the write request will not immediately flush data into persistent storage. Instead, the data will be written into a local write buffer. Data in write buffer will be merged and subsequent write will overwrite previous buffered data when there is an overlap. When the file system receives a close() call or an explicit flush() request, data in buffer will be truncated

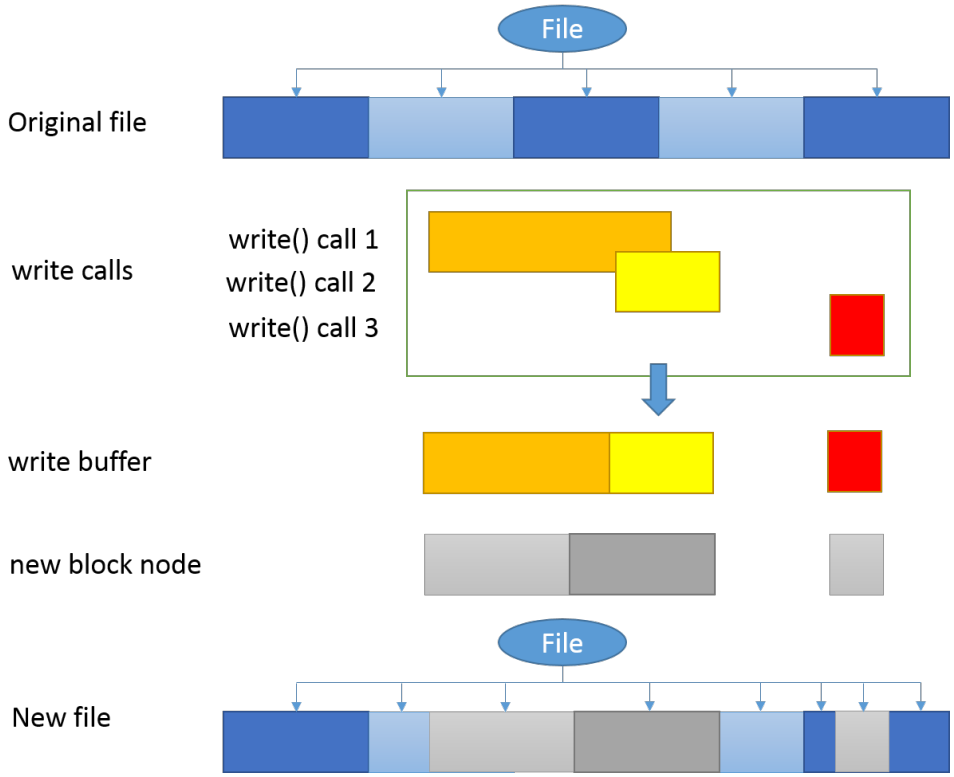

Figure 3.7: Write Buffer

into blocks and uploaded to remote servers. The content in local buffer may lost due to a system failure, but user can always use a fsync() or fdatasync() as suggested in POSIX standard to flush important data to remote. The flush process is atomic, guaranteed by MongoDB.

During a flush, a data section (section object) may be unaffected, partly overwritten or entirely overwritten. Block node will be detached from the file if its corresponding data section has been entirely overwritten. The data section object will be truncated if this section is partly overwritten. If a section is truncated, the start offset and end offset will be updated to reflect the change while the block node remain the same. The Figure 3.7 shows the routine to write a file.

Compared to the traditional copy-on-write snapshot file system which copies and overwrites the entire block whenever there is a byte change, this design can make use of the old block. Because the old block will be kept for snapshot, reusing it in current view may save some storage.

However, there is a tradeoff between these savings and the overhead of such design. Each data section object requires 224 bits of storage to store its SHA hash, rolling hash and begin/end indexes, each block node requires 128 bits to store its SHA ID. There are many cases where

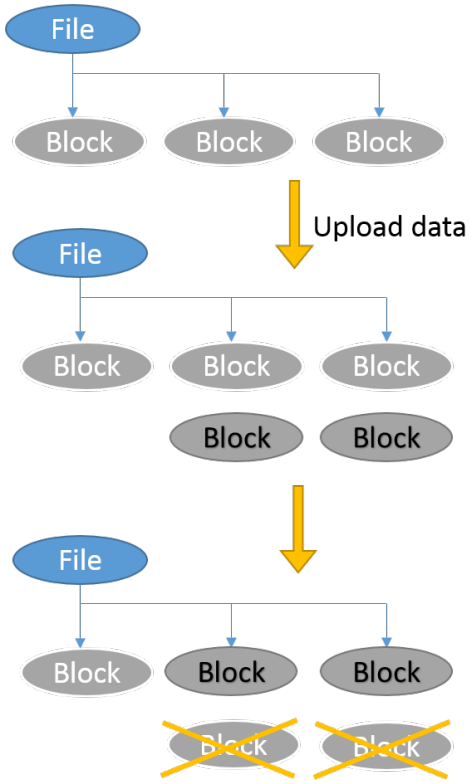

Figure 3.8: Consistency - within a snapshot

this overhead is worthy. In the extreme case, if we overwrite two bytes on the boundary of two blocks right after a snapshot is taken, two new blocks will be created in classic copy-on-write snapshot system but only a 2-byte block will be added in our design. With a block size of 2,048 bytes and a file size of 20,480 bytes, the saving in this case is 3,798 bytes.

3.4.3 Consistency

Maintaining consistency is an important task for file system. During a write call, the file system may enter an inconsistent state before the write call successfully returns. A file system should prevent a read call from reaching the inconsistent state to keep consistency. There are many ways to keep a system consistent. Some of the file systems use lock to prevent a read operation when the file system is processing a write call. In our implementation, we use atomic update to prevent the inconsistent state.

Figure 3.8 and 3.9 demonstrate how we prevent inconsistent state. The write process can be divided into two steps. The first step is upload, where we upload the new data node and new version of metadata node that is going to be affected by the write call. In this step, read

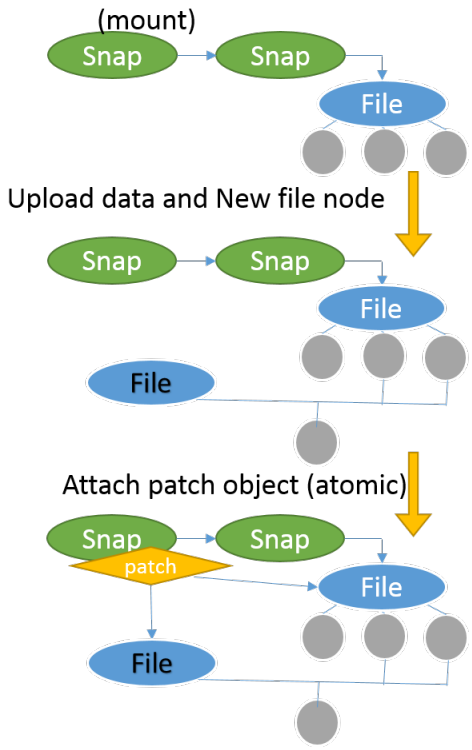

Figure 3.9: Consistency - cross snapshots

operations may come in at any time and observe the old content because all new nodes are not connected to the directed graph and all old nodes haven't been modified. The second step is atomic update. This step is always atomic, guaranteed by MongoDB. In this step, the file system will change affected reference from referring the old node to new node. When a write operation only affects one snapshot. The file system will directly replace the old node with new node by updating the referring reference as shown in Figure 3.8. When a write operation affects more than one snapshot like Figure 3.9, the file system will upload a patch object to replace the old node with new node and then attach the object to the snapshot node in the atomic update step.

3.4.4 Deduplication

Block node creation is an expensive operation as it requires storage space on remote and all data must be transferred to remote. But block node creation occurs when creating a file, when appending a file even when writing a file and copy-on-write applies.

We found that not all block node creations are necessary. Consider the following scenario, when some user program is trying to create a copy of a file, the program will read in all data

from the file to a buffer and then write the data in the buffer to another newly created file. As a result, the file system will experience a series of read() and write() function call, not aware that these function calls are related. File system has no way to know that the block node it is going to write already exists in the file system and the node can be reused.

By using SHA hash as the block node ID, it is possible to find and eliminate blocks that contain duplicate data. Before a block of binary data is uploaded, the file system will query the database with its SHA hash. If a instance with same ID is found, the file system will stop unloading the block node but directly return the ID of existing node with same hash value. In this way, the file system not only can perform a copy operation efficiently but also save space for duplicate blocks.

3.5 Conclusion

In this chapter, we presented a design of distributed file system with replaceable modules. We demonstrated the overview of the file system, basic entities and operations in the system, some detailed design choices like the cache and deduplicate feature.

Chapter 4

Snapshot

One of the major features in Kabi File System is the writable copy-on-write snapshot. A snapshot is a point-in-time copy of a defined collection of data [1]. A read-only snapshot is an immutable copy of the file system data at a time spot, while a writable snapshot can be considered as a writable fork of such copy. Snapshot nodes are designed as the basic components of Kabi File System. The "current" view of the file system is also treated as a writable snapshot (the latest snapshot in default branch). This snapshot system focuses on reducing the storage space occupied by a snapshot.

According to the Storage Networking Industry Association, three classic snapshot approachs include split-mirror, changed block, and concurrent [2] The split-mirror approach copies every byte from source to snapshot. The process is time consuming hence it usually requires planning in advance. The changed block approach applies copy-on-write on the snapshot. The concurrent approach redirects IO request to different storage spaces associated with snapshots. Instead of making a copy and overwrite the copy, write IO request will be redirected a separate storage space. While a read IO request will be redirected depends on whether the data has been changed since last snapshot.

Our snapshot system uses a strategy that is a mixture of copy-on-write and redirect IO. It uses enhanced copy-on-write strategy on actual data and redirect IO strategy on metadata. In our snapshot system, all but one snapshot log the changes since the prior snapshot. Such that the snapshot system can then recover the content of the a snapshot based on the previous snapshot and the log of changes, as shown in Figure 4.1. The only exception is a special snapshot called head snapshot which references contents of the entire file system. Other snapshots are called referencing snapshot. They contain a reference to another snapshot and an array of patch objects. Each patch object in the array reflects a series of changes on a single file or a single directory since the prior snapshot.

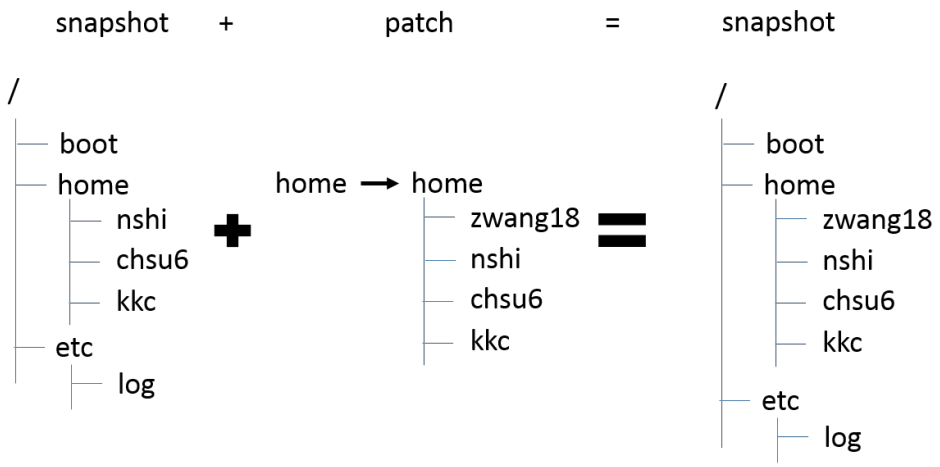

Figure 4.1: Snapshots and Patches

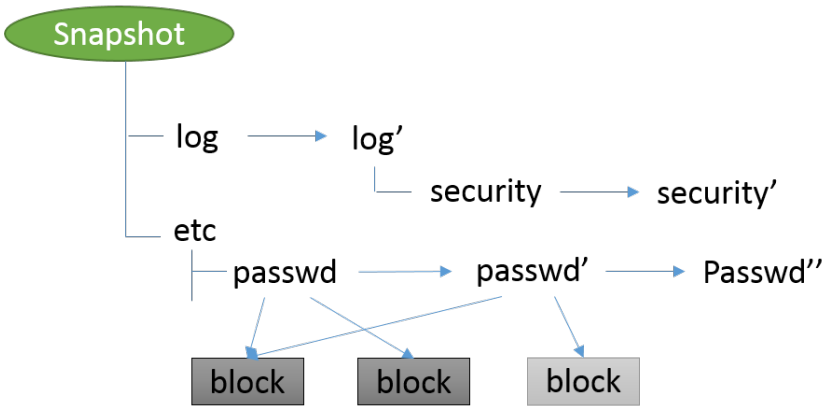

Figure 4.2: Snapshots in SnapFS

4.1 The Snapshot Tree

The snapshot system in Kabi File System uses an approach based on patches. A similar approach adopted by ext3cow uses a reserved field in inode to reference the previous version inode shown in Figure 4.2. One advantage of the patch based approach is that it supports tree structured snapshots and writable snapshots. With tree structured snapshots, one can fork the file system keep multiple "current" verison of the file system at the same time.

Snapshots in the Kabi File System are represented by snapshot nodes and form an up-tree. In an up-tree, each child points to its parent. Figure 4.3 shows an example of a snapshot tree. The right bottom node (node H) is the root of the tree. The root of the snapshot tree is always the head snapshot.

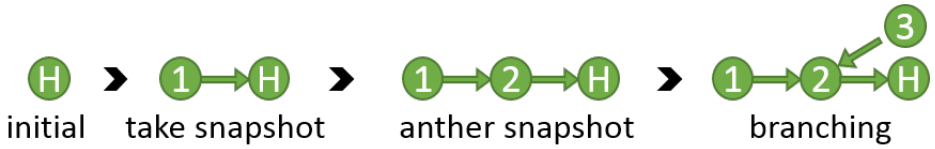

Figure 4.3: An example of snapshot tree

Initially, the file system contains a default writeable snapshot node (node H in Figure 4.3) representing the current view of the file system. This special snapshot called the head snapshot is the root of the snapshot tree and the only snapshot node that does not reference other snapshot. For initial state, any writes to the file system go directly into the head snapshot.

After a snapshot is taken, a new snapshot node (node 1 in Figure 4.3) is created referring to the head snapshot node. Subsequent write operations will not only write data into the head snapshot but also submit patches to all snapshot connected to head snapshot, so as to reflect the difference between snapshot node 1 and its referencing node H.

Branching a snapshot creates a writable copy of a existing snapshot. The branching operation is implemented by creating a new snapshot node referring the snapshot being forked. As shown in Figure 4.3, node 3 (the writable copy) is created and connected to snapshot node 2 (the existing snapshot) in order to fork the file system based on snapshot 2.

In the snapshot tree, each writable snapshot correspond to a branch. A branch consists of the writable snapshot (the current status of the branch) and its historical snapshots (the history of the branch). The main branch is the branch where the head snapshot node lies. Figure 4.4 shows the idea of branch. In the example, there are two branches in this 5-node snapshot tree. Node H is the root of the up-tree so corresponds to the head snapshot. Thus node 1, node 2, and node H form the main branch. In the figure, nodes on the left corresponds to an older snapshot and nodes on the right corresponds to a more recent snapshot. Therefore in main branch, node 1 is the oldest snapshot while node H represents the latest state. Node 1 and Node 2 form the history of main branch and node H is the current view of main branch. Node 4 is another writable snapshot and it belongs to a side branch. The side branch consists of node 1 (the oldest), node 2 (the second oldest), node 3 (the third oldest), and node 4 (most recent). Note the arrows between nodes reflect the referencing relations between snapshots, not the order of creation.

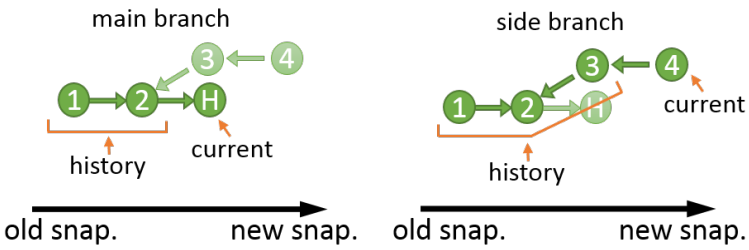

Figure 4.4: Branches

4.2 Snapshot Nodes and Patch Object

As mentioned in the previous section, there are two types of snapshot nodes in this file system, namely the head snapshot node and the referencing snapshot node. The head snapshot is special in the file system. It is the root of the up-tree and does not reference any other snapshots. Instead it stores the entire content of current view of main branch by referencing the directory node of the root directory. In this way, read access to the head snapshot node is straight forward and faster than any other snapshot. Because of this property, the head snapshot is recommended to represent the most frequently accessed snapshot or the current state of the default branch.

On the other hand, a referencing snapshot does not reference its own root directory but it keeps a reference to another snapshot node and an array of patch objects. The array of patch objects represent the difference in content between this referencing snapshot and the referenced snapshot. Reading a referencing snapshot will first read the file system content in referenced snapshot and then lookup the patch list to find out if the content is changed in this snapshot. To write data, the referencing snapshot uploads the changed nodes into the database, build a patch object with both IDs, upload and attach the patch to snapshot node in remote. This upload and attaching process is atomic and ensures the file system will not be left in a incomplete state. Once a referencing snapshot is referenced by some other snapshot nodes, it becomes a read only snapshot. Figure 4.5 shows the structure of a head snapshot node (right) and the structure of a referencing snapshot (left).

A patch object reflects a single or a series of modifications to a single node. The node can be either file node or directory node. The patch object references a pair of file nodes or directory nodes. The two nodes referred are the target node (original version) and its replacement (changed version).

Figure 4.6 demonstrates the way the patch system works. Snapshot 1 and 2 demonstrates how a directory changes between snapshots. The directory that is represented by node d_2 in snapshot 2 is now replaced by directory node d_1 in snapshot 1. In snapshot 2, the directory has a new subdirectory but the file under that directory remains unchanged. Hence the target

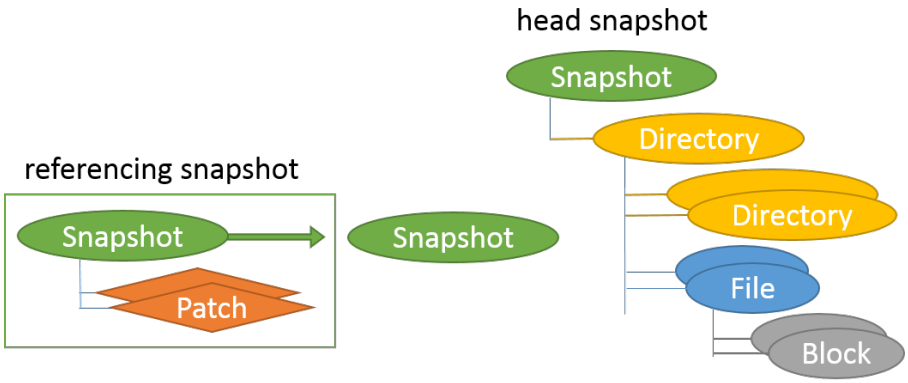

Figure 4.5: Root Snapshot and Non-root Snapshot

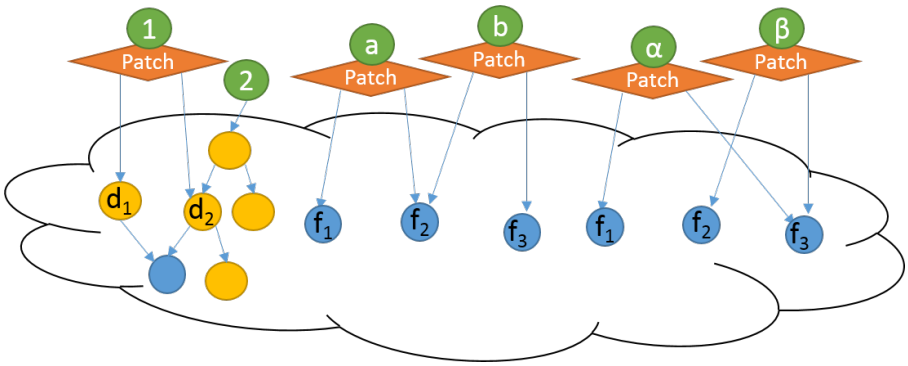

Figure 4.6: The Principle of Patches

node (node d_2) and replacement node (node d_1) have a reference to the same file node but the target node d_2 has one more reference to a directory node. In the example of snapshot (a) and snapshot (b), a file changes in both snapshot, its original version is f_1, the intermediate version in snapshot (b) is f_2 and final version is f_3 in snapshot (a). In snapshot α and snapshot β, file f_3 is replaced by f_2 in snapshot (β) and replaced by f_1 in snapshot (α).

4.3 Snapshot Related Operations

Operations in snapshot system read the content in a snapshot and make changes to a writeable snapshot. Making changes to the current view of the file system also involves snapshot write operation as the current view of the file system is also treated as a snapshot in Kabi file system.

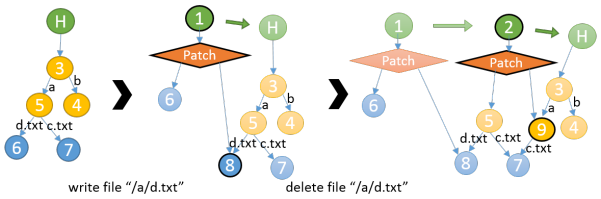

Figure 4.7: Read a snapshot

4.3.1 Read Operations

When reading a file or a directory in a referencing snapshot, the file system must to do a large number of lookup in patch lists to ensure that the file system is referring to the correct version of the node. For instance, in Figure 4.7, to read the file "/a/d.txt" in snapshot 1 shown in figure below, the file system will first read in "/" directory in head snapshot which is node 3. Then it transverses all involved snapshots (1 and 2) to see if there's a patch whose target node is node 3. Since no such patch is available, this means node 3 is the "/" directory node of all snapshot node (1, 2 and 3). Then the file system will look for a directory under node 3 and corresponding patches. In this example, there is a directory (node 9) with display name a under node 3 but there is also a patch to node 9 in snapshot 2. That means node 5 is the "/a" directory node of snapshot H while node 6 is the directory node for snapshot 1 and snapshot 2. Follow the same procedure, we will find node 8 is the "/a/d.txt" node in snapshot 2 but for snapshot 1 "/a/d.txt" represented by node 6.

As one can see, read operations rely on the patch lookup operation. To avoid query and traverse all involved snapshot and patch objects on every read operation, a local patch list is built and stored as hash table in memory when a referencing snapshot node is mounted. The local patch list combines all patches that may be used for a node lookup. In the example shown in Figure 4.8, when mounting snapshot 3, a local patch list that contains all effective patches in snapshot 3 and 2 is created.

Not all patches in patch lists will be combined into a local patch list. Because patches come from different patch lists, when they combine together some patches are mergeable and some are unnecessary. For instance, in Figure 4.9 snapshot (a) and (b) the replacement node f_2 of a patch is also the target node of another patch, these two patch object can be merged into one local patch. Another example is snapshot (α) and (β), the two patch shown in the figure have the same target node f_3. When snapshot (α) is mounted, the patch in snapshot (β) will become ineffective and will not be read in.

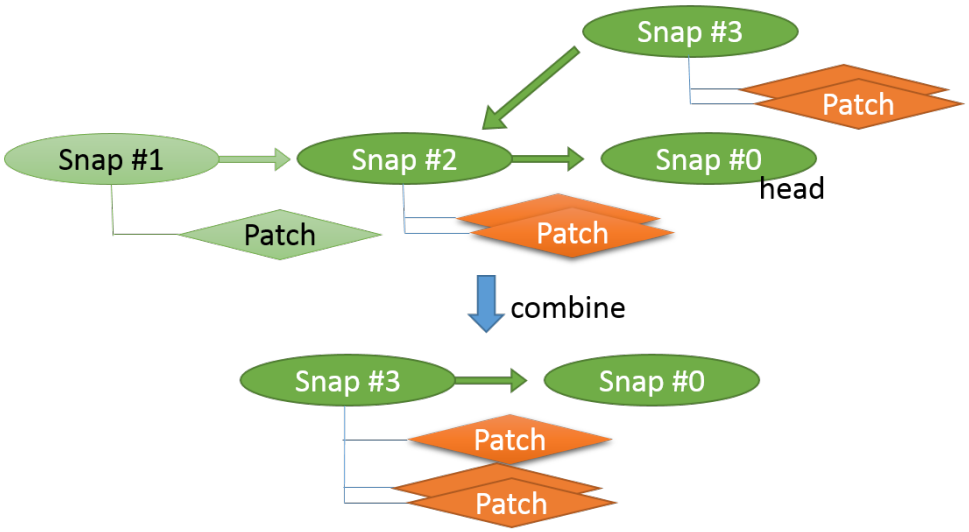

Figure 4.8: Combine Patch Lists

4.3.2 Write Operations

In both copy-on-write snapshot system or redirect IO snapshot system, a file system entity (block, file, directory) may be referenced one or more times. The referencer could be snapshots or the "current" view of the file system. If an entity is referenced only once, write operation to that entity will be straight forward. This is a write operation within a snapshot. On the other hand, when an entity is referenced more than once, the snapshot system usually need special treatment to the write operation. Otherwise a direct in-place write will affect all referencers. In the snapshot system, we focus on the later case. If not otherwise specified, "write operation" in this section refers to those write calls whose target entities has been referenced more than once.

Figure 4.10 briefly demonstrates how patch objects and snapshots work with write operations. This example has three snapshots. Node H is the head snapshot node representing the current status of the main branch. Snapshot node 2 represents the initial status and node 1 is the intermediate status of main branch. Initially, the file system contains three directory "/", "/a", "/b" and two files "/a/c.txt", "/a/d.txt". In between the initial snapshot and the next snapshot, file "/a/d.txt" was overwritten so its representing node is changed by a patch. After snapshot 2 is taken, the file "/a/d.txt" was deleted. A new patch is attached to snapshot 2 to reflect this change.

Figure 4.11 and 4.12 shows how to take snapshots on the main branch and side branches. In order to take a new snapshot on the main branch, the file system will create a new snapshot node in between the head snapshot and its adjacent snapshot node. This newly created snapshots will

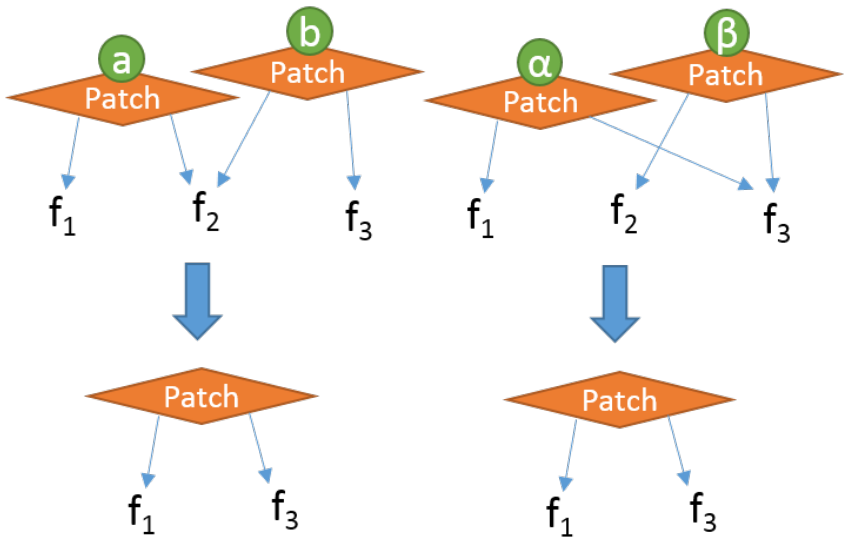

Figure 4.9: Merge Patches (Local)

have an empty patch list referencing the head snapshot. This new snapshot node will represent the status of this branch at the time it is created. The current state of the main branch is still the head snapshot.

To take a snapshot on a side branch, the file system will create a new snapshot node attached to the end of the side branch. The newly created snapshot node will reference the latest snapshot on this branch and have an empty patch list. After attaching to the branch, this snapshot will become the current status of this branch.

When writing a snapshot other than the head snapshot, the Kabi File System will submit new patches to that referencing snapshot to reflect the change. In the first example shown in Figure 4.13, the file system is writing the head snapshot in main branch, replacing node X with its newer version Y. Node Y will replace node X directly in the head snapshot. In order to keep its previous version X in snapshot 2, a "reverse" patch object (Y-to-X) will be submitted to revert node Y back to its original version X. In this way, the head snapshot will have the new version Y while all other snapshots keep the old version X. In the second example, the file system is trying to replace node X with its new version Y in snapshot 3. Compared to the first example, the file system now can simply submit a X-to-Y patch to snapshot 3. In the third example, we demonstrate a walk around to writing an internal snapshot node by creating a branch based on that internal node and write to that branch.

To create a branch based on the head snapshot, it is recommended to create a dummy snapshot first and then fork from that dummy node rather than branching the head snapshot

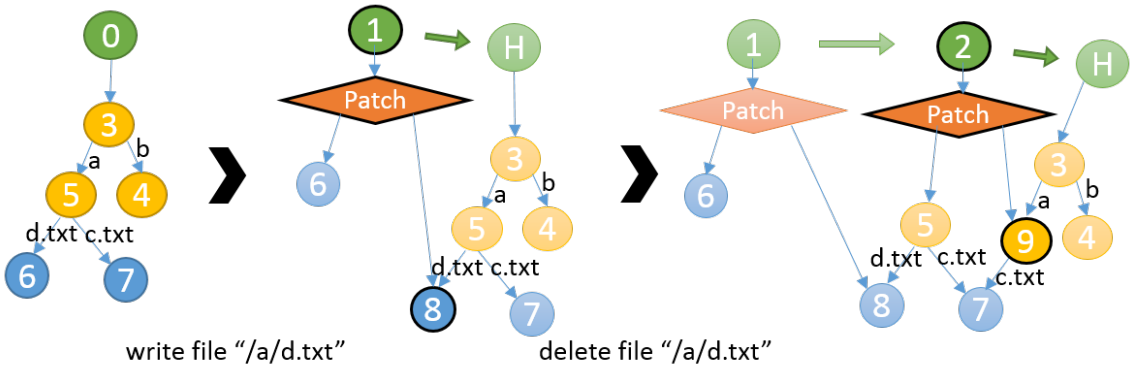

Figure 4.10: Example: Create Snapshot after Write

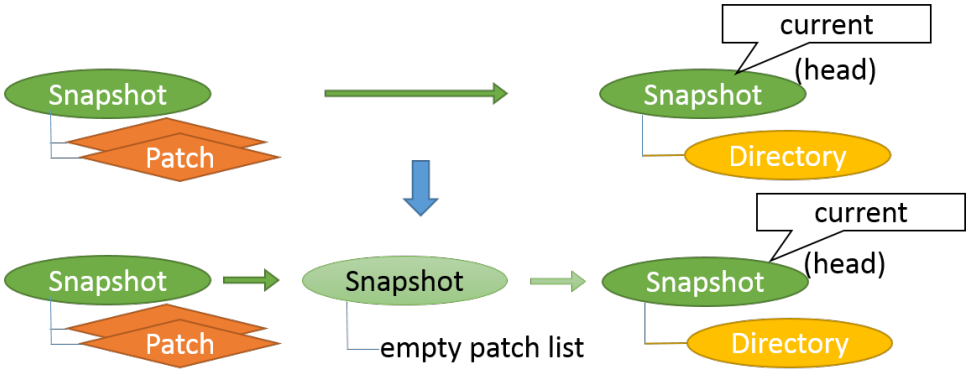

Figure 4.11: Take Snapshots on Main Branch

directly. This is because a write operation to head snapshot will submit patches to all snapshot nodes connected to head snapshot node. Hence we wish to limit the number of snapshots connected to the head snapshot. If we fork the head snapshot directly then there will be multiple snapshot nodes connected on to the head snapshot node. Figure 4.14 demonstrates this issue. In the upper case, with dummy snapshot node (node 2), a write operation to head snapshot (node 0) only submits a patch to the dummy node. But in the lower case, when there is no dummy node, a write operation to the head node has to submit two identical patches to node 1 and node 3.

Generally, within a snapshot, each node (file or directory) should have no more than one patch associated with it. For example, a patch replacing node 0 with node 1 and a patch replacing node 1 with node 2 can be merged into one patch. This happens when a node is modified multiple times. For time and space efficiency, it is better to merge them into one patch as Kabi does.

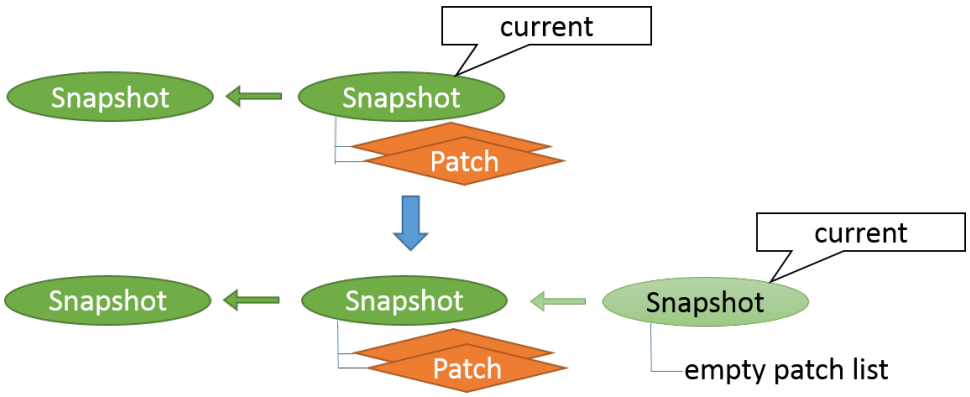

Figure 4.12: Take Snapshots on Side Branch

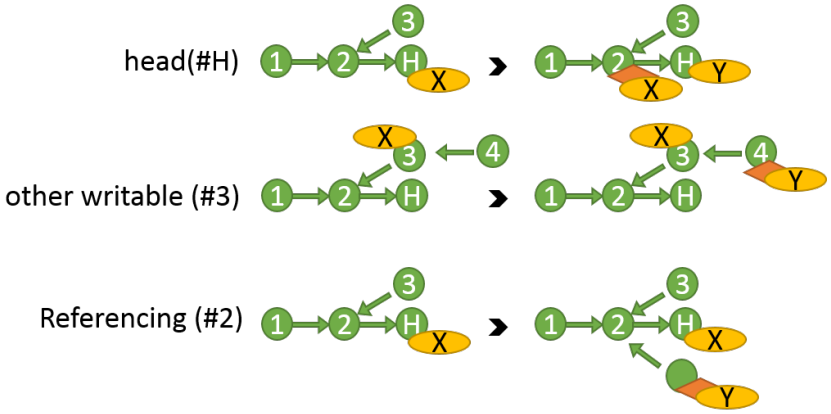

Figure 4.13: Write a Snapshot Node

4.4 Enhancing Copy-on-Write and Deduplication

The copy-on-write strategy and file system deduplication improve space efficiency of the file system. File system deduplication finds and eliminates duplicated blocks and files while the classic copy-on-write strategy eliminates unnecessary copy of an unchanged block to snapshots.

4.4.1 Motivation

A classical copy-on-write snapshot system applies copy-on-write at block level. As shown in Figure 4.15, unchanged blocks will not be copied to the snapshot.

However, this is not always ideal. In many use cases, like insertion or deletion, a write operation only affects a few bytes instead of a whole block. But a classic file system will rewrite all successor blocks in these scenarios. A classic snapshot system will make a copy of all successor

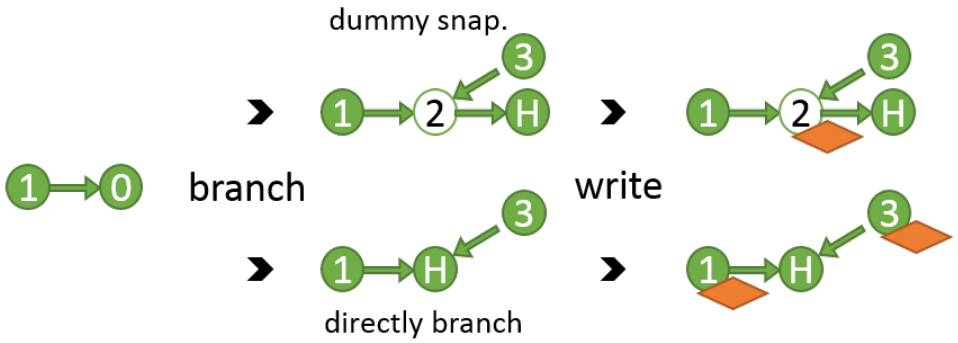

Figure 4.14: Branching Root Snapshot Node Directly

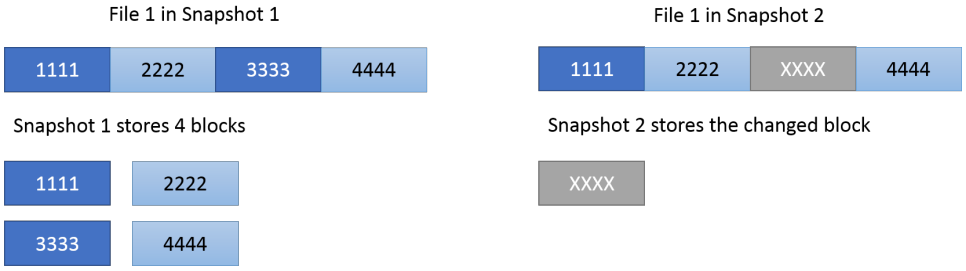

Figure 4.15: Classic Copy-on-Write

blocks despite the fact that only very few bytes is changed. The following Figure 4.16 addresses this issue. In this example, a byte is inserted into the file at offset 8. In classical copy-on-write snapshot system, two successor blocks are treated as changed blocks and will be copied. However, the data in those blocks did not change. They only moved one byte forward. In the figure we also demonstrates a potentially better approach.

To solve this problem, we have to let the file system be aware of the true intention of the end user. This is not straightforward because the POSIX standard uses only one file system call to handle all kinds of modification to a file. The only function of this file system call is to rewrite a part of a file.

If the user program intends to insert a byte right in the middle of the file, the file system will receive a set of write calls to rewrite all later blocks in order to move original data 1 byte forward. The same behavior can also be observed when user program trying to rewrite the later half of the file. It is difficult for the file system to distinguish these two scenario. Access patterns can be used to guess the intention of an operation (i.e., an insertion usually results in a truncate call followed by a series of write() calls), but it is not an ideal solution as it depend on how the user program will behave. Therefore, our major challenge is to identify the true intention of an write operation, whether should be an insertion or a complete rewrite. Next chapter, we will

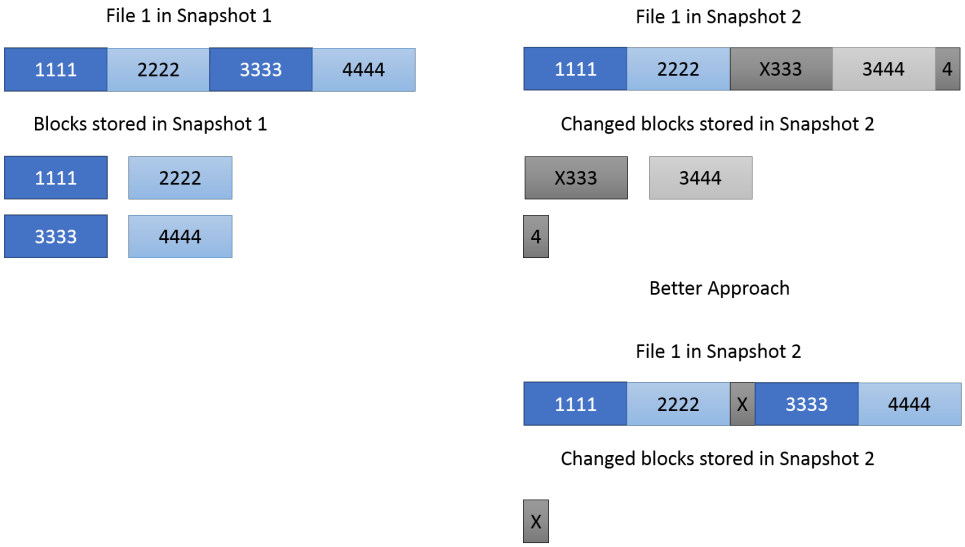

Figure 4.16: Issue in Classic Copy-on-Write

show how to accomplish this goal by using the rsync algorithm [31].

4.4.2 The rsync algorithm

As discussed in the previous section, in order to identify duplication in a better way, we need to have a mechanism to compare the data to be written into the file and the original data. The rsync algorithm is originally designed for the efficient update of data over a high latency and low bandwidth link. Compared to brute force search and string search algorithms, rsync algorithm is much faster in practice and requires less data exchange between the remote server and local machine. These features make it suitable for a distributed file system. Because both time consuming file system and a high bandwidth consumption file system will become a bottleneck in the operating system.

The basic flow of the rsync algorithm is to split the remote file into blocks of length S, calculate their rolling checksum and then send them to the local machine. The local machine will search through local file to find all blocks of length S bytes (at any offset, not just multiples of S) that matches the received rolling checksum. This can be done in a single pass very quickly since the rolling checksum only requires $O(1)$ time to compute checksum at offset k given the checksum at offset $k-1$.

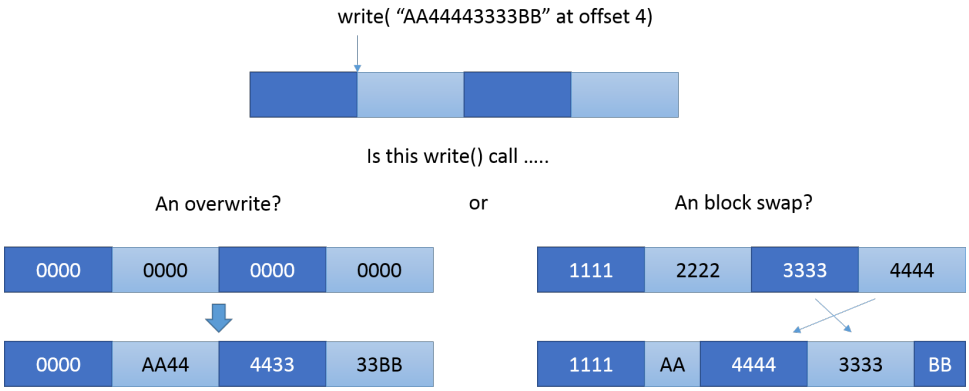

Figure 4.17: Identify the intention of write operations

4.4.3 Enhancing the Space Efficiency

The Kabi File System uses the rsync algorithm to enhance the space efficiency of snapshots. It assumes that in most cases two different versions of the same file will share part of their data.

In the Kabi File System, a section object in a file node contains not only the reference to the corresponding block, but also contains the rolling checksum of the block data. Before flushing the write buffer, the local machine will calculate the rolling checksum of data block at all possible offsets. The file system will compare these rolling checksums with those fetched from remote. If a match is found, the file system will then double check their SHA-1 hash to confirm that it is indeed a duplication.

Once all data blocks have been examined, the local machine will send the ID of duplicated blocks and all remaining data back to the remote server.

During this process, the computational overhead is only the calculation and matching of rolling checksums. An important property of rolling checksum algorithm is that successive values can be computed in $O(1)$ time, thus ensuring that all rolling checksums can be calculated in $O(n)$ time. In contrast, the benefits are a significant decrease in network flow and remote storage when there is duplicate data in buffer.

4.5 Conclusion

In this chapter, we present the basic idea of this snapshot system and discuss some implementation details of the snapshot subsystem in Kabi File System. We demonstrated our efforts to make the snapshot system efficient in space usage. We also introduced the rsync algorithm to improve the snapshot system and the network performance.

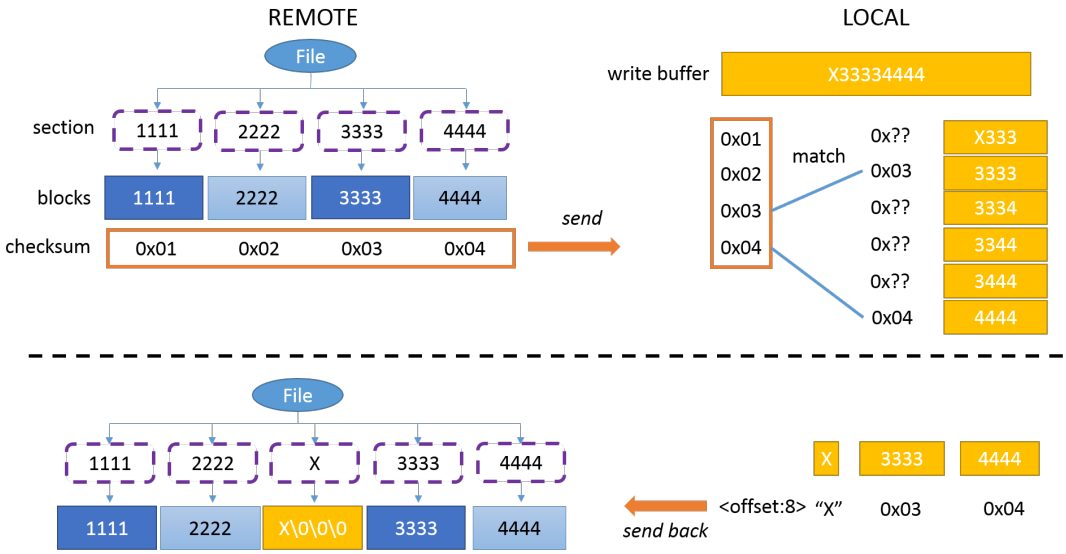

Figure 4.18: Using rsync to find unchanged blocks

Chapter 5

Performance

As an important resource management component in the operating system, the performance of a file system has a large influence on the operating system. In this chapter, we will evaluate the performance of the design. We will be focusing on the time efficiency of the file system and the space efficiency of the snapshot system.

5.1 File System Benchmark

In this section, the performance of the Kabi File System is evaluated using the file system benchmark tool "postmark"[17]. All tests are using the default benchmark settings of postmark which include 500 stand-alone file creations, 264 file creations mixed with transactions, 243 file reads, 257 file appends and 764 file deletions.

We compare our proof-of-concept implementation with other popular file systems. In our implementation, we use the FUSE-JNA[21] as the Java language binding for FUSE. FUSE-JNA is designed for fast development of concept file system but not for the performance. FUSE-JNA creates a Java thread for every file system call, uses JNA to communicate with the fuse library, and switch between kernel space and user space very frequently. Hence the overhead of using FUSE-JNA is significant. In order to eliminate such overhead in comparison, other file system will be wrapped with FUSE-JNA.

Two sets of tests are performed: the local test set and the remote test set. In the local test set, Kabi File System client and the backend MongoDB are deployed on the same machine. The performance of Kabi File System is compared to a Ext4 file system. The testing environment is 64-bit Ubuntu 12.04LTS with one 2.4GHz 6MB cache Intel i7-2760QM CPU, in total 24GB DDR3 1333MHz RAM, and a 7200 rpm SATA hard disk. In the remote test set, Kabi File System client and the backend MongoDB are deployed on different machines. The remote test uses Amazon AWS service to build the testing environment. Both machines use Amazon EC2

Scenario		Tests[1]				
File System	Test Set	Creation[4]	Creation[5]	Read	Append	Delete
Kabi	local	83	33	30	32	47
Ext4[2]	local	96	49	53	46	88
Kabi	remote	55	29	27	28	36
NFS [2,3]	remote	45	26	24	25	36

[1] Test results are in "operation per second", the larger the better.
[2] The file system is wrapped with FUSE-JNA.
[3] The client uses NFS to mount a directory on remote Ext4 partition.
[4] Stand-alone creation: this test does nothing but create files
[5] mixed with transactions: these creations are performed between other operations (Read, Append, Delete)

Table 5.1: File System Performance Test

t2.micro instance with one 8GB EBS volume and connect to a 10 Gbps local area network. Their operating systems are standard 64-bit Ubuntu image provided by Amazon AWS. The remote test set compares Kabi File System with a FUSE-JNA wrapped NFS.

The file system test results are shown in Table 5.1. The result shows that although as a local file system the Kabi File System is not as good as Ext4, but as a disitributed file system it is comparable to NFS.

5.2 Efficiency of Snapshot and Deduplication

This section will focus on the space efficiency of the snapshot system. We are going to measure the space efficiency of the snapshot system by estimating the average space cost of a snapshot of a single file. We will study the influence of two factors against the space efficiency. The first factor is the ratio of file size against block size. The second factor is the proportion of "truncated section" in the file. A "truncated section" referes to those section that are not referring to a whole block. For example, the third section in Figure 4.18 or the second and third section in Figure 3.4 are all truncated sections.

We follow the steps below to estimate the storage space occupied by a snapshot (all random numbers follow a uniform distribution, size of space calculated by adding up size of all fields in all newly created node):

1. Initialize the file system with block size B.
2. Generate a file with size F.

3. Fill the file with sections and let a certain proportion (P) of the sections be truncated.

4. Take a snaphot of the file and make two side branches.

5. Switch to side branch 1, insert random number of bytes into the file at random offset.

6. Take a snapshot on side branch 1 and calculate the total space occupied by this snapshot.

7. Switch to side branch 2, overwrite random number of bytes from random offset.

8. Take a snapshot on side branch 2 and calculate the total space occupied by this snapshot.

9. Repeat above steps for 10,000 times to get the average value.

Table 5.2 shows two sample results of such experiment. The first three columns in the table are the variables of the experiments. The next four columns are the data gathered from the experiments. For example, the column labeled "overwrite, classic" means corresponding write operation is an overwrite request and the algorithm used by snapshot system is classic copy-on-write.

The first row in Table 5.2 represents an experiment on a Kabi File System initialized with 128-byte block size. The target file is a 12,608-byte file where 3% of all sections are truncated. The second row is for a file 10 times larger. For the first experiment, the result shows that on average it takes a classic copy-on-write snapshot system 3,288 bytes to take a snapshot after an overwrite operation. It takes 103 bytes more for a Kabi File System to take a snapshot under the same condition. When it comes to insertion, on average it only takes the Kabi File System 3,256 bytes to take a snapshot after an insertion while it costs almost 2 times more space for a classic copy-on-write snapshot system to do so. In order to compare data between first and second experiment, we will use normalized data instead of absolute value in all tables here after. The normalization wil be based on the file size.

This result is not difficult to explain. Because there is not much duplicated data in the overwrite scenario, SHA hashes and rolling checksums become overheads. This makes the performance of Kabi File System slightly less than the classic approach. But in the insertion scenario, lots of duplicated data can be found. The rsync algorithm is able to find the duplications to improve the efficiency. On the contrary, the classic approach cannot detect and make use of these duplicated data. So the Kabi File System have a better performance when it comes to insertion. The Kabi File System is optimized for efficient insertation. The test shows that it does performs better than classic copy-and-write strategy.

experiment variables			overwrite results		insert results	
block size	file size	truncated section	classic	Kabi	classic	Kabi
128	12608	0.03	3288	3391	9530	3256
	126080		31722	32205	94867	32557

Table 5.2: Sample result of the experiment

experiment variables			overwrite results		insert results	
block size	file size	truncated section	classic	Kabi	classic	Kabi
128	704	0.17	0.4531	0.5127	0.8338	0.2983
	1408		0.3514	0.3934	0.7933	0.2962
	2112		0.3129	0.3532	0.7789	0.2964
	3520		0.2884	0.3213	0.7702	0.2972
	7040		0.2681	0.2949	0.7589	0.2955
	14008		0.2593	0.2828	0.7545	0.2957

Table 5.3: File Size to Block Size ratio

5.2.1 Block Size and File Size

The block size and the file size also influence the efficiency of a snapshot system. A high file-size-to-block-size ratio usually means fine-grained blocks. A classic copy-on-write snapshot will get better efficiency with a fine-grained block. Consider the extreme case where the file-size-to-block-size ratio is 1. Then the file contains exactly one block. Then any change means a complete rewrite to the file by classic copy-on-write strategy. This is shown on Table 5.3 column 4 and column 6 where the overhead of classic strategy decreases as file-size-to-block-size increases.

However, column 7 shows that there is no obvious relationship between file-size-to-block-size ratio and efficiency when doing an insertion in Kabi File System. One reason for this could be the Kabi File System can truncate block into smaller sections freely as shown in Figure 3.7. Therefore the Kabi File System can have fine-grained sections even though the file system uses a large block size.

The data in column 5 reflects the fact that an increase in file-size-to-block-size ratio will result in an improvement in efficiency. The reason is that the proportion of *extra* metadata will reduces as the file size increases. For example, an overwrite operation can result in at most two additional sections added to the file node. This is an extra 56-byte metadata overhead. Compared to a 704-byte file, such overhead is large (almost 8%). On the other hand, if the file size is 14,008 bytes, this overhead (metadata of 2 extra section) can be omitted (about 0.4%).

experiment variables			overwrite results		insert results	
block size	file size	truncated section	classic	Kabi	classic	Kabi
128	12800	0.00	0.2596	0.2645	0.7547	0.2495
		0.10	0.2600	0.2752	0.7558	0.2750
		0.18	0.2599	0.2856	0.7557	0.3002
		0.33	0.2606	0.3071	0.7569	0.3513
		0.46	0.2616	0.3290	0.7579	0.4026
		0.57	0.2616	0.3500	0.7583	0.4532
		0.71	0.2596	0.3792	0.7530	0.5252
		0.82	0.2579	0.4089	0.7505	0.5980
		0.91	0.2593	0.4417	0.7536	0.6758
		1.00	0.2601	0.4737	0.7561	0.7536

Table 5.4: Truncate Ratio

5.2.2 Truncate Ratio

The rsync algorithm identifies duplication in local buffer and remote sections. It uses the rolling checksum of sections to find duplications. But a truncated section does not have a valid rolling checksum thus it cannot be benefited from the rsync algorithm. Hence we can infer that the more truncated sections we have, the less efficient snapshot system and file system deduplication will be.

Table 5.4 supports this inference. It shows the relationship between the efficiency and truncated sections. Because a classic copy-on-write snapshot system does not have "truncated section", they are not influenced by this experiment variable. On the other hand, truncated sections significantly influence the efficiency of Kabi File System. It makes rsync algorithm less powerful. Because rsync algorithm depends the rolling checksum which is a checksum of fixed-length data . But truncated section is "shorter" than normal section thus it checksum will not be calculated. Large number of truncated sections also reduces the chance to find two duplicated block using the deduplicate function offered by the file system. When all sections are truncated, the rsync algorithm no longer provide any extra efficiency in insert operation. Hence the insert performance of classic copy-on-write and Kabi File System is almost the same in the last row.

In this experiment, we explicitly set the size of truncated section to be half of the block size. That means for a Kabi File System with a 128-byte block size, the overhead (28 bytes) in metadata is almost half of the data contains in truncated section. In other words, for every 2 bytes stored in truncated section, there is 1 extra byte cost in metadata. Since the increase in truncated-to-untruncated ratio implies more truncated section which is less efficient, this explains the significant decrease in efficiency in column 5 of Table 5.4.

5.3 Conclusion

In this Chapter, we showed the effects of three important factors that affect the snapshot efficiency. The three factors are the block size, the file-size-to-block-size ratio and the proportion of truncated sections.

Generally, in Kabi snapshot system, an increase in block size or an increase in file-size-to-block-size ratio will make the overheads of metadata more efficient. The proportion of truncated sections is the most important factor among the three factors. An increase in this proportion will significantly decrease the efficiency of the snapshot system in Kabi File System.

Chapter 6

Conclusion

In this thesis, we presented the design of a new distributed file system built on document oriented NoSQL database with snapshot capability. We also built a proof of concept implementation of such design using MongoDB, FUSE and other techniques. We designed and implemented a new patch based snapshot system and try to improve its efficiency in many ways. We evaluated the snapshot system by testing variables that effect the efficiency of the snapshot and file system deduplication. We also came up with some explanation of how variables effect the performance.

We believe the major contribution of this thesis are the design of a distributed file system based on document oriented database, the design of the patch based snapshot subsystem, and the evaluation of such design.

6.1 Future work

The Kabi File System is a prototype file system and our implementation is not perfect. There are possible fixes or enhancements to the Kabi File System. Some of them are discussed below.

Reducing the number of truncated sections. From Chapter 5, we demonstrates a strong negative relation between the proportion of truncated sections versus the space efficiency. Therefore, if we can reduce the number of truncated sections, we may have increased performance in return. One of the possible ways may be merging adjacent truncated sections into less but larger sections. Furthermore, since the number of truncated sections never reduce over time in our current design, it is possible that after long run a file node may filled with truncated sections. So it may be worthy the rebuild the whole file node such that the new file node contains the same data but less truncated section.

Finding optimal value for block size. Smaller block size will lead to fine-grained sections. As shown in Chapter 5, fine-grained sections increases the performance of snapshot after file insertion operation. However, larger block size means less sections. Thus it reduces the metadata

occupied by section objects and block nodes. We may find an optimal value for block size to maximize the performance of snapshot system and deduplication.

Using native FUSE instead of FUSE-JNA. As discussed in Chapter 5, the file system uses FUSE-JNA to connect the file system logic with fuse library. FUSE-JNA is a good language binding for Java development, but it is slow in performance [21]. A FUSE-JNA wrapped ext4 is almost 1,000 times slower than the native ext4 according to our tests. Thus we believe by replacing FUSE-JNA with original fuse C library, we will observe improvement in performance.

Promoting a writable snapshot to head snapshot. In our design of snapshot system, head snapshot is special and access to head snapshot is optimized. We assume the working branch of a Kabi File System user is always the main branch. So we make sure the writable snapshot on the main branch is always the head snapshot. However, a user may make some mistake on main branch and abandon the main branch. If he want to start working on a side branch, he will suffer the overhead of patch processing. In such case, that user may want to promote the writable snapshot on that side branch to head snapshot, so as to avoid the overhead. This can be done by simply revert all patches on the snapshot chain from previous head snapshot to new head snapshot.

REFERENCES

[1] Storage Networking Industry Association. SNIA dictionary. `http://www.snia.org/education/dictionary/s`.

[2] Alain Azagury, MF Factor, Julian Satran, and William Micka. Point-in-time copy: Yesterday, today and tomorrow. In *NASA CONFERENCE PUBLICATION*, pages 259–270. NASA; 1998, 2002.

[3] Matt Blaze. A cryptographic file system for unix. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16. ACM, 1993.

[4] Neil Brown. Overlay filesystem documentation. Kernel Git Respository - pub/scm/linux/kernel/git/mszeredi/vfs.

[5] Hsiang-Ching Chao, Tzong-Jye Liu, Kuong-Ho Chen, and Chyi-Ren Dow. A seamless and reliable distributed network file system utilizing webspace. In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*, pages 65–68. IEEE, 2008.

[6] Paul R Eggert and Douglas Stott Parker Jr. File Systems in User Space. In *USENIX Winter*, pages 229–240, 1993.

[7] Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu I Siminiceanu. Model-checking the linux virtual file system. In *Verification, Model Checking, and Abstract Interpretation*, pages 74–88. Springer, 2009.

[8] Amir Goldstein, Aditya Dani, Shardul Mangade, Piyush Nimbalkar, Harshad Shirwadkar, and Yongqiang Yang. ext4-snapshot project repository. `https://github.com/YANGYongqiang/ext4-snapshots`.

[9] The Open Group. *IEEE Std 1003.1*. IEEE Standards Association, 2013 edition.

[10] R Nigel Horspool and Gordon V Cormack. Constructing word-based text compression algorithms. In *Data Compression Conference*, pages 62–71, 1992.

[11] John H Howard et al. *An overview of the andrew file system*. Carnegie Mellon University, Information Technology Center, 1988.

[12] MongoDB, Inc. Document Databases. `http://www.mongodb.com/document-databases`.

[13] MongoDB, Inc. JSON and BSON. `http://www.mongodb.com/json-and-bson`.

[14] MongoDB, Inc. MongoDB Overview. `http://www.mongodb.com/mongodb-overview`.

[15] Richard Jones. Gmal filesystem over FUSE. `http://sr71.net/projects/gmailfs/`.

[16] Jan Kára. Ext4, btrfs, and the others. In *Proceeding of Linux-Kongress and OpenSolaris Developer Conference*, pages 99–111, 2009.

[17] Jeffrey Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html, 1997.

[18] Peter W Madany, Michael N Nelson, and Thomas K Wong. File system level compression using holes, Jun 1998. US Patent 5,774,715.

[19] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33. Citeseer, 2007.

[20] ABM Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *International Journal of Database Theory and Application*, 4, 2013.

[21] Etienne Perot. FUSE-JNA project page. `http://fusejna.net/`.

[22] Eelco Plugge, Peter Membrey, and Tim Hawkins. *GridFS*. Springer, 2010.

[23] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[24] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.

[25] Fareha Shafique. Progress report: Secure isolated copy-on-write file system. 2006.

[26] Bhavana Shah. Disk performance of copy-on-write snapshot logical volumes. Master's thesis, The University Of British Columbia, 2006.

[27] Seungjun Shim, Woojoong Lee, and Chanik Park. An efficient snapshot technique for ext3 file system in linux 2.6. *Realtime Linux Foundation (RTLW)*, 2005.

[28] Michael Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010.

[29] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 2011.

[30] David Teigland and Heinz Mauelshagen. Volume managers in linux. In *USENIX Annual Technical Conference, FREENIX Track*, pages 185–197, 2001.

[31] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm, 1996.

[32] Werner Vogels. File system usage in windows nt 4.0. *ACM SIGOPS Operating Systems Review*, 33(5):93–109, 1999.

[33] Wikipedia. EXT2 page on wikipedia. `http://en.wikipedia.org/wiki/ext2`.

[34] Wikipedia. POSIX page on wikipedia. `http://en.wikipedia.org/wiki/File_Allocation_Table`.

[35] Wikipedia. POSIX page on wikipedia. http://en.wikipedia.org/wiki/POSIX.

[36] Wikipedia. ZFS page on wikipedia. http://en.wikipedia.org/wiki/ZFS.

[37] Jacky Wu. FUSE (FileSystem in UserSpace) for JAVA. https://launchpad.net/jnetfs.