NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE PROGRAM IN INFORMATICS

MASTER'S THESIS

Optimizing the recovery of data consistency gossip algorithms on distributed object-store systems (CEPH)

Theofilos N. Mouratidis

***Supervisor*:** **Mema Roussopoulos,** Professor

ATHENS

DECEMBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση αλγορίθμου φλυαρίας για συνεπή δεδομένα σε κατανεμημένα συστήματα αποθήκευσης αντικειμένων (CEPH)

Θεόφιλος Ν. Μουρατίδης

Επιβλέπουσα: **Μέμα Ρουσσοπούλου,** Καθηγήτρια

ΑΘΗΝΑ

ΔΕΚΕΜΒΡΙΟΣ 2020

MASTER'S THESIS

Optimizing the recovery of data consistency gossip algorithms on distributed object-store systems (CEPH)

Theofilos N. Mouratidis

STUDENT'S ID: M1455

SUPERVISOR: **Mema Roussopoulos,** Professor

THESIS COMMITTEE:	**Mema Roussopoulos,** Professor
	Alexis Delis, Professor
	Dimitrios Gunopoulos, Professor

December 2020

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση αλγορίθμου φλυαρίας για συνεπή δεδομένα σε κατανεμημένα συστήματα αποθήκευσης αντικειμένων

Θεόφιλος Ν. Μουρατίδης
Α.Μ.: M1455

ΕΠΙΒΛΕΠΩΝ: **Μέμα Ρουσσοπούλου,** Καθηγήτρια

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:	**Μέμα Ρουσσοπούλου,** Καθηγήτρια
	Αλέξης Δελής, Καθηγητής
	Δημήτριος Γουνόπουλος, Καθηγητής

Δεκέμβριος 2020

ABSTRACT

The data growth on the internet is increasing rapidly and systems for storing and preserving the sheer volume of information are nowadays on the rise. Ceph is a distributed storage system for handling large amounts of data, it was initially developed by Sage Weil (Redhat) and it is gaining popularity over the years. Ceph is being used as a system for big data storage in large companies such as CISCO, CERN and Deutche Telekom. Although a popular system, as any other distributed system, its individual components fail over the course of time. In this case, the recovery mechanisms need to take place to resolve any issues. In this thesis, we introduce a new way to synchronise the data between the replicas to make the data consistent, by identifying and filtering unchanged objects. The current algorithm for recovery in Ceph is a durable yet simple implementation regarding disk access and memory consumption. As the technology evolves and faster storage solutions emerge (e.g. PCIe SSDs), practices such as Write-Ahead Logging for data consistency can also introduce new problems. Having thousands of write operations logged per second under a degraded cluster can rapidly increase memory consumption and fail a storage node (degraded is a cluster state in which a storage node is down for any reason). Although, Ceph now supports an upper limit on the number of entries in its WAL, this limit is often reached and it invalidates the log, because any new entries will be lost. Therefore, the system is left to check every object of the replicas so it can synchronize them, which is a very slow process. Hence, we introduce the Merkle trees as an alternative solution to Bloom filters so the recovery procedure can identify regions where objects were not modified and thus reduce the recovery time. The recovery process has an observable impact on the users' IO bandwidth, and the overall experience for them can be improved by reducing the cluster's recovery times. The benchmarks show a performance increase of 10% to 400% that varies with how many objects were affected during the downtime of a node.

SUBJECT AREA: Distributed Systems

KEYWORDS: optimization, gossip algorithms, object-store, Merkle trees, bloom filters

ΠΕΡΙΛΗΨΗ

Η αύξηση των δεδομένων στο Διαδίκτυο αυξάνεται ραγδαία και τα συστήματα αποθήκευσης και διατήρησης του τεράστιου όγκου πληροφοριών γίνονται όλο και ποιο δημοφιλή. Το Ceph είναι ένα κατανεμημένο σύστημα αποθήκευσης αντικειμένων για το χειρισμό μεγάλων ποσοτήτων δεδομένων. Το σύστημα αυτό αναπτύχθηκε αρχικά από τον Sage Weil (Redhat) και κερδίζει δημοτικότητα με την πάροδο του χρόνου. Το Ceph χρησιμοποιείται ως σύστημα αποθήκευσης μεγάλων δεδομένων σε μεγάλες εταιρείες όπως η CISCO, η CERN και η Deutche Telekom. Αν και είναι ένα δημοφιλές σύστημα, όπως και κάθε άλλο κατανεμημένο σύστημα, οι κόμβοι της συστάδας του αποτυγχάνουν με την πάροδο του χρόνου. Σε αυτήν την περίπτωση, θα πρέπει να πραγματοποιηθούν μηχανισμοί αποκατάστασης χαμένων δεδομένων για την επίλυση τυχόν προβλημάτων. Σε αυτή τη διατριβή, συστήνουμε έναν νέο τρόπο συγχρονισμού των δεδομένων μεταξύ των αντιγράφων για να κάνουμε τα δεδομένα συνεπή, εντοπίζοντας και φιλτράροντας τα αμετάβλητα αντικείμενα. Ο τρέχων αλγόριθμος για ανάκτηση χαμένων δεδομένων του Ceph είναι μια ανθεκτική αλλά απλοϊκή εφαρμογή σχετικά με την πρόσβαση στο δίσκο και την κατανάλωση μνήμης. Καθώς η τεχνολογία εξελίσσεται και γίνονται ταχύτερες λύσεις αποθήκευσης (π.χ. PCIe SSD, NVME), πρακτικές όπως το πρωτόκολλο προεγγραφής ημερολογίου (Write-Ahead Log) για τη συνέπεια των δεδομένων μπορούν επίσης να δημιουργήσουν νέα προβλήματα. Καταγράφοντας χιλιάδες εγγραφές ανά δευτερόλεπτο κάτω από μια υποβαθμισμένη συστάδα κόμβων μπορεί να αυξηθεί αρκετά γρήγορα η κατανάλωση μνήμης και να αποτύχει ένας κόμβος αποθήκευσης (η υποβαθμισμένη συστάδα είναι μια κατάσταση της συστάδας στην οποία ένας κόμβος αποθήκευσης είναι εκτός λειτουργίας για οποιονδήποτε λόγο). Παρόλο που το Ceph υποστηρίζει πλέον ένα ανώτατο όριο στον αριθμό των εγγραφών του WAL, αυτό το όριο επιτυγχάνεται συχνά και ακυρώνει το ημερολόγιο πλήρως, επειδή οι νέες εγγραφές θα χαθούν. Επομένως, το σύστημα στην τρέχουσα υλοποίηση χρειάζεται να ελέγχει κάθε αντικείμενο των κόμβων αντιγράφων, ώστε να μπορεί να τους συγχρονίσει, κάτι που προφανώς είναι μια πολύ αργή διαδικασία. Ως εκ τούτου, παρουσιάζουμε τα δέντρα Merkle ως μια εναλλακτική λύση στα φίλτρα Bloom, ώστε η διαδικασία ανάκτησης να μπορεί να εντοπίσει περιοχές όπου τα αντικείμενα δεν τροποποιήθηκαν και έτσι να μειώσει τον χρόνο ανάκτησης αυτών των δεδομένων. Η διαδικασία ανάκτησης έχει ένα εμφανές αντίκτυπο στην επίδοση των λειτουργιών των αντικειμένων (γράψιμο, ανάγνωση) των χρηστών και η συνολική εμπειρία για αυτούς μπορεί να βελτιωθεί με την μείωση των χρόνων ανάκτησης χαμένων δεδομένων της συστάδας. Σύμφωνα με τα πειράματα που πραγματοποιήσαμε, παρατηρούμε αύξηση απόδοσης της τάξης των 10% έως 400% που ποικίλλει ανάλογα με τον αριθμό των αντικειμένων που επηρεάστηκαν κατά τη διακοπή λειτουργίας ενός η περισσότερων κόμβων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κατανεμημένα Συστήματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: βελτιστοποίηση, διαδικασία φλυαρίας, αποθήκες αντικειμένων, δέντρα κατακερματισμού, bloom φίλτρα

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF ALGORITHMS

1. INTRODUCTION

1.1 Object Storage recovery

Distributed object store systems are starting to become the new trend for storage, and they are now used extensively to improve data operations storage-wise and performance-wise at scale. Nevertheless, system critical operations such as recovery and maintenance can still impact noticeably the clients' I/O performance. When hardware failures occur, or when update procedures are performed, or any other operation that can impact a part of the cluster, while the system is still available, it operates at a slower rate. Some systems may even require a full halt of the client operations in order to perform maintenance tasks. On the contrary, elastic systems, such as Ceph, can remain available at any possible recovery procedure, but with a noticeable impact on the client I/O [45]. This performance drop is caused by the number of storage nodes that have changed, and now the system has to re-balance its data across the new set of nodes to adapt with the changes. Ceph is the open-source system that it is used extensively as a distributed software-defined storage solution to support for example Openstack, an open source virtualization system. This will be the object storage system that will be discussed in depth in this thesis, and we will optimise its recovery procedure to improve its performance in various common use cases.

To begin with, when a storage daemon's binary is updated, it must be stopped first. Once the daemon is stopped, the object storage system can still serve the clients due to data redundancy. While the software update is performed, the stopped daemon, which has a "down" state, misses any updates that were done to its peers. Therefore, when the storage daemon comes up online again, it needs to recover the data changes that it has missed. Open-source software systems usually have shorter version life cycles compared to the commercial ones. Minor versions are updated frequently to tackle with the various bugs that system administrators have encountered. For example, the Ceph developers release a minor version every two months and hotfixes every two weeks at the time of writing. The administrators of large Ceph clusters tend to avoid updating the system frequently, because they can't afford the performance impact of the recovery phase that happens after the system update, thus leaving them susceptible to possible bugs that can impair the system. Even though a maintenance operation may not be prevalent, hardware or software failures occur far more often. When a hardware failure occurs, it will trigger the recovery procedure of the storage system. For example, an error can be a failure of a memory module, a degraded operating system disk, or any other problem that can occur without impacting the storage mediums that hold the object data and metadata. If the storage medium that holds the cluster's data fails and gets replaced, it must recover all of its data, therefore we can't do anything to optimise this use case. Once the hardware issues are resolved and the daemon is up and running again, it must synchronize with its peers to continue its regular work. Unfortunately, the synchronization procedure steals a noticeable part of the cluster's I/O bandwidth while it performs. Therefore, we will try to reduce the duration of the recovery procedure to minimize the clients' performance impact.

The current recovery implementation of Ceph called "backfill", performs a full scan of the daemon's objects, compares their versions with its peers and requests any changed object, and the objects may differ in data, metadata or both. The backfill operation happens when the recovery with a WAL (write-ahead log) has reached a certain memory limit and it has been discarded [25]. Now with the new storage mediums such as SSDs and NVMEs, the client operations are done fast enough to fill the allocated memory for the WAL in minutes. Ceph provides a duration parameter that triggers the backfill anyways if the storage node is in stale state for a long time, which has a default value of 15 minutes. Once the object storage system's nodes try to get synchronized, the full scan of the restarted daemon's data becomes a serious bottleneck.

Other systems, such as the distributed database Apache Cassandra, has a state-of-the-art recovery system, in which it builds a full binary tree from its tables' data hashes and propagates recursively the changes from the leaves to the root. Each node has unique identifier, a hash value, for the data it represents, any change to the underlying data changes that hash value and thus differences in data ranges can be detected. This idea is similar to the synchronization of directories in sharing applications, such as Dropbox[1]. For example, in order to synchronize two directories, the local and the remote one, the program will descend the folders recursively and check the modification timestamps of each file or folder. If they have an earlier timestamp, the program will try to update the files from the directory which has a latter modification timestamp. When this procedure finishes, both directories will have the files with the latest modification metadata and they will finally be synchronized. In a similar way, when a recovery happens in Apache Cassandra, the peers exchange their trees and they find changes by recursively descending their trees. Each leaf may represent any chunk of data in the database or a whole table compared to just a file in a sharing application. Once those leaf differences are found, the peers resolve the inconsistencies and eventually become synchronized. The trees generated in Apache Cassandra are full binary trees, compared to the directory structure of a filesystem which may not be balanced. The recursive approach in Cassandra greatly improves the recovery time over a full scan implementation, by following the divide-and-conquer paradigm, when finding nodes with same hash values, Cassandra can omit unnecessary checks on the children which can be costly at scale. Therefore, if during a failure of a database node only one table was updated, with the tree approach it is more likely for the node to try to update only the modified table, instead of scanning each table of the failed node for differences.

The current object storage systems don't have an optimised approach for recovery like what Apache Cassandra provides, as they usually have a metadata server (Gluster) that regularly scans and checks for differences between replicas which is still considered a periodic full scan. Other object storage systems, such as Amazon's S3 or Facebook's object storage, are closed-source commercial solutions which do not disclose any information on how they approach the procedure of recovery. In this thesis we will adapt Apache Cassandra's recovery system into an open-source object store system, such as Ceph, and introduce a few key improvements over the current state-of-the-art recovery solution. On the following pages of this thesis we will discuss in depth what object storage systems are, the state-of-the-art solution for recovery that Apache Cassandra provides, the adaptation of the key elements of the recovery mechanism into an object storage system instead, the performance improvements regarding building the trees and the results of the recovery experiments.

[1] https://www.dropbox.com/

1.2 Object Storage Systems

With the rapid increase of the amount of online data generated and stored all over the world, companies and researchers were led to develop the appropriate software that can tackle this growing problem. For example, the scientific computing teams that need to analyze huge volumes of data, they need the proper systems in order to perform their experiments as fast as possible. At first, supercomputers were in need and they worked well over conventional systems. High-Performance computing (HPC), the principle behind supercomputers is also in demand of fitting applications that can harness the parallel performance capabilities that those systems can offer. In addition, it was proven that there was an urgency for better filesystems over the classic ones (e.g. POSIX) to retrieve, analyze, and store data faster. Later, the object storage model was created by Garth Gibson on 1995 to deal with the filesystem usage on parallel/distributed systems [1]. By that time, the NFS (Network File System) and AFS (Andrew File System) were the popular distributed file systems of that time [2][3]. Thus, Garth Gibson proposed this alternative of the previously mentioned filesystems as an improvement in the areas of file operations, by decoupling I/O actions that happen often, for example read/write operations, and seldom such as namespace changes, as seen in the following model:

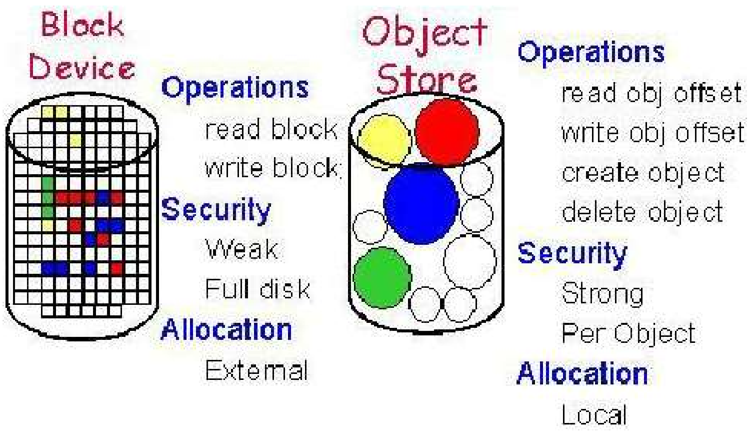

Figure 1: Differences between block and object storage I/O operations

In the object storage systems, the files or objects are in the form of a triplet, where the first field indicates the unique identifier of the object, the second field represents the actual data stored, and the last one describes the object's metadata. The metadata are dynamic, therefore someone can create applications on top of the metadata on various levels, for example on the disk level, the system level (filesystem) or the interface level (security). Having a simple architecture like this, the object storage model can be easily used by distributed systems, provided that the distribution of the objects and the data replication can be done with object-level granularity using the dynamic metadata.

The object store systems have been designed to hold an enormous amount of data and this is one of the reasons why companies nowadays invest in those systems. Currently, the popular cloud storage systems have an object store backend to cope with the volume of data stored. An example would be Amazon and its S3 object store that they have released for the broad audience, including both individuals and companies. The S3 storage is abbreviated from Amazon's Simple Storage Service, which is an in-house object store solution that they provide to their users on their cloud computing platform [19]. Additionally, in the industry, Openstack, an open source software for creating cloud solutions, is backed with object storage systems in order to provide images, volumes and file shares between Virtual Machines (VMs). Even companies like Dropbox and Facebook use object storage systems to save data in the form of multimedia, such as photos, music and video which consume most of their storage resources [20][21]. Therefore, object storage solutions are a popular choice when trying to build a scalable storage system.

1.2.1 Differences with key/value stores

Given the previous information about the object storage architecture, someone can claim that there is no clear difference between the object stores and the key/value stores. This is partially correct, for the fact that they both store key/value tuples on an abstract level. Still, from the user point of view both systems provide the same interface, but from the system point of view, they both measure and optimise their performance in different aspects. Therefore, this fuzzy difference between the two systems is defined in the details. For example, the key/value stores expect a small value to be in each key/value pair compared to the other system. Specifically, the in-memory key/value store Redis allows up to 512MB[2] of information stored in the value of a k/v pair. Although, the object stores try to optimise the storage of k/v pairs of arbitrary size. For example, a k/v pair can have a value that is equivalent to 4GBs of data. In this case, the k/v tuples are stored with a pointer instead of the actual value in the object store's index. The pointer can be in the form of two values, one pointing the offset on the data block and the other the size, to locate the value of the k/v pair in a disk.

1.2.2 Filesystems vs Object Stores

Filesystems are the basic system anyone can think of when the keyword "storage" is in discussion. Every operating system supports a variety of filesystems and it needs at least one to operate itself and the needs of the users. For instance, the Unix operating systems follow the POSIX protocol for the storage of the files. A POSIX compliant filesystem for example is XFS [5]. Compared to a strict POSIX filesystem, object stores follow a relaxed POSIX protocol than the one implemented in XFS. This has to do with various performance reasons due to the CAP theorem [6], because object stores are distributed systems compared to common filesystems, and they require some sacrifices to be able to perform well at scale. In addition, the distributed filesystems are popular amongst the HPC community, because they help with the efficient utilization of the supercomputer's resources to transfer, transform and analyze big data. One example of a distributed filesystem is Lustre. Lustre is found in many entries of the IO-500 list. This list contains the fastest filesystem setups and it is renewed each year. Most of the filesystems including Lustre in the list are based on an object storage backend. The flat structure of an object storage system, where there is an id for a file instead of a hierarchical path, can contribute a lot to the scalability of a distributed filesystem.

[2] https://redis.io/topics/data-types [17-02-2019]

2. CEPH OBJECT STORE

Ceph is a distributed software defined storage solution, that is based on objects. The system is autonomous, scalable to petabyte volume storage, has self-healing capabilities, provides automatic management of each node and has no single point of failure [10]. Software defined storage solutions provide efficient storing of data, without the need of specific hardware, such as RAID[3] cards that are used often in datacenters and database servers of web services. Ceph is the storage system that was created from the Ph.D. dissertation of Sage Weil, on the CRUSH algorithm in the University of California, Santa Cruz on 2006 [7]. On 2012 Sage Weil created the Inktank company to promote, sell and support the system for businesses. Two years later, Inktank was acquired by RedHat. The project is open source and written in C++ and python. The name CEPH comes from the group of marine animals called cephalopods, such as the octopus, the squid and the nautilus. Each major version is code-named after a cephalopod with a 12-month release cycle. CRUSH (Controlled, Scalable, Decentralized Placement of Replicated Data) is a pseudo-random hash algorithm for efficient data distribution, without redundant data transfers that offers the searching and finding of the objects within the store without a central directory. This system presents itself as a multi-purpose storage system that can scale to handle many petabytes with ease. This order of scaling is feasible, due to the flat model of the CRUSH algorithm where each object is identified and stored by its hash value without existing in any hierarchical structure. Additionally, CEPH differentiates the data and the metadata operations on the objects, exactly as Garth Gibson's model of object storage describes, therefore it gives more space for read and write operations and improves the overall performance of the underlying storage mediums.

Ceph is used primarily in the industry and in scientific organizations, such as Deutsche Telekom and CERN. The feature that makes it competitive in the market is that it provides three storage solutions in one system: object storage (Rados), block storage (RBD) and a POSIX filesystem (CephFS). Therefore, the businesses and the organizations that need at least two of the three use cases can cut costs by maintaining and licensing only one system. The block storage part and the filesystem part are implemented by simply using the object storage core and manipulating the dynamic metadata of the objects. The three storage types provided by CEPH are the following:

1. **Object Storage (RADOS[4]):** This is the basic functionality of CEPH, it is used as is to store and manipulate objects with the librados client library that has API bindings in frequently used languages, such as C/C++ and Python. Another way to store objects is through the HTTP protocols of Amazon's S3 and Openstack Swift that the Rados Gateway exposes to the users. Rados is the name for the distributed object store of CEPH [11]. Rados is comprised of object store daemons (OSDs) that are autonomous, self-healing and self-managing without any user intervention apart from the initialization of the system.
2. **Rados Block Storage (RBD):** This functionality is used for the creation of virtual block devices. These block devices can be used as network mounted devices on an operating system. Their size is arbitrary, and it is bounded by the underlying hardware capabilities of the storage cluster. This functionality can be used as the backend for Openstack to provide VMs, by using system images as RBD images.

[3] Redundant Array of Independent Disks

[4] Reliable Autonomous Distributed Object Store

Openstack also uses Manila shares, in which a share is a common space for different VMs that operate on that space. Those shares are also backed by RBD.

3. **CephFS:** It is based on the representation of a file an object. CephFS is a POSIX compliant filesystem and it can be used on the most popular Linux based operating systems, such as the RedHat OS family (RedHat, Fedora, Centos) and Ubuntu. This filesystem can be mounted with two different ways, one of them is called FUSE [8] and the other is the supported kernel mount client. This filesystem is designed to be used in HPC applications, as an alternative to Gluster[5] and Lustre. To harness the performance of CephFS, HPC users run CephFS nodes along the HPC nodes in a hyperconverged way, so they can benefit from the read/write and network locality.

In the following parts of section 2, the architecture of CEPH will be explained, by describing and analyzing the different daemons that are essential to the CEPH storage service. In the first part, the basis will be explained in detail. The second part will be about the system's Monitor and MGR (Manager daemon). The third part will describe briefly the optional daemons that support the Rados gateway, RBD and CephFS part of the system. In the last part, the object store daemon (OSD) will be explained in detail, which is the process that this thesis is based on.

2.1 The CEPH cluster

CEPH is a master/slave distributed object storage system that is comprised of multiple nodes with different roles. Each role interacts with the rest in a semi-autonomous way and has a specific task, such as handling I/O requests or checking whether a node is alive. CEPH is an AP system with eventual consistency using the vocabulary of the CAP theorem. For a basic setup, the most essential processes that are needed to be able to run a CEPH cluster are the Monitor, MGR and OSD daemons. The Monitors and the MGRs are the masters of the system and the OSDs are the slaves. The MGR daemons were a part of the monitor process until the Luminous version (v12.10.0) where they split as different processes, because the role of the Monitor was complex enough and it required a role disjunction. Now, the new MGR process has also implemented some cluster management through a handful of third-party modules that are written in a scripting language, such as Python or Lua. Besides the basic processes that will be explained further in detail, CEPH has a distinct daemon for RadosGW (Rados Gateway) to handle S3 and SWIFT requests through an HTTP interface and a daemon for supporting CephFS called MDS (Metadata Server Daemon).

The data storage itself is done inside a term called data pools. Those pools are a data container abstraction that has its own configuration regarding how the data is stored. These configurations can include geographical or hardware location, data compression, encryption and data availability through a replication or an erasure-coding schema [26] [27] [18]. For obvious reasons those pools can also be used to store various data to a different set of users for security reasons. Additionally, CEPH is advertised as a reliable storage that uses unreliable hardware. These data availability schemas can be applied in a variety of use cases, usually regarding the amount of data that need to be stored. For example, in a small-scale business that needs to store a small amount of data and the

[5] https://www.gluster.org/ [2019-10-03]

hardware failures are seldom because of light usage, a replication data availability schema can be used, such as the common one with three times replication. In a large-scale organization or business, the data replication model can be quite costly for hard disk procurements, therefore an erasure coding schema can be used to provide a safe storage with less data redundancy penalty. For example, with four data stripes and two parity ones, there will be 50% additional volume required, compared to the 200% additional volume requirement for a triple replication schema, which is half the cost for the same amount of data availability [17]. There is a balance between the two choices, one that has a little bit better I/O performance (with replication) or the better data volume cost (with erasure coding). Erasure coding is a bit slower for writing or reading because it requires to interact with more nodes in order to write the stripes and also in the cases of reading data, erasure coding will require to construct the object from the different stripes and send it back to the user who requested the whole file. Thus, erasure coding can cut the volume costs, but it imposes a heavier network traffic and a bit of slower I/O.

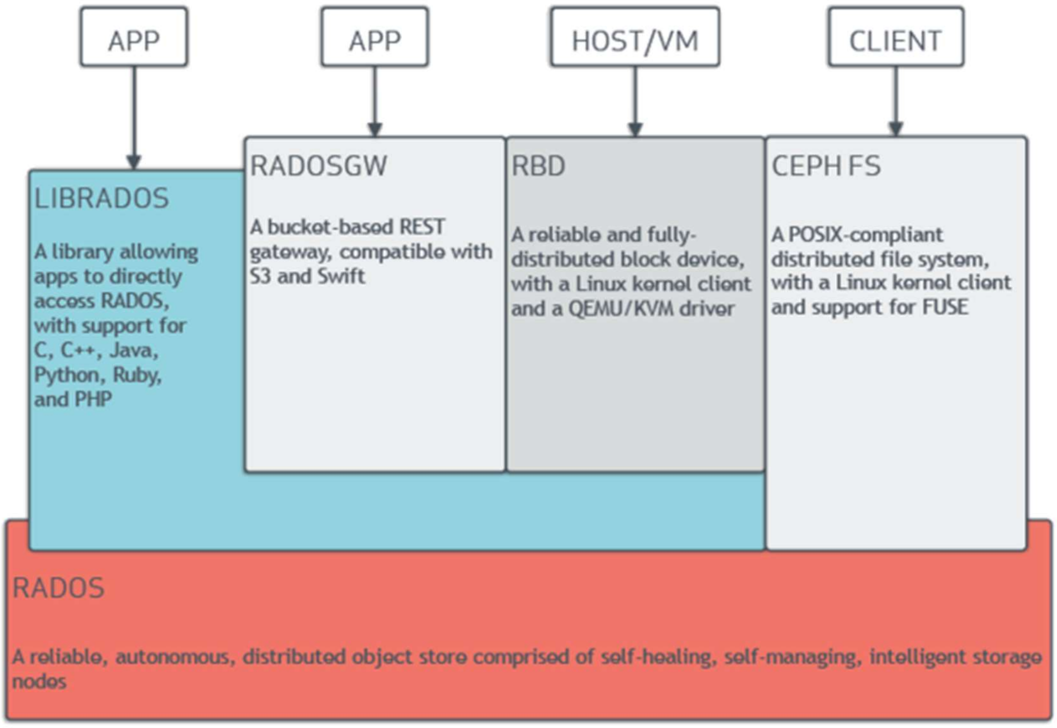
APP
APP
HOST/VM
CLIENT
LIBRADOS
A library allowing apps to directly access RADOS, with support for C, C++, Java, Python, Ruby, and PHP
RADOSGW
A bucket-based REST gateway, compatible with S3 and Swift
RBD
A reliable and fully-distributed block device, with a Linux kernel client and a QEMU/KVM driver
CEPH FS
A POSIX-compliant distributed file system, with a Linux kernel client and support for FUSE
RADOS
A reliable, autonomous, distributed object store comprised of self-healing, self-managing, intelligent storage nodes

Figure 2: The CEPH stack, it shows the connections between the parts that composes the system

As seen from the description, CEPH is a flexible system that provides a plethora of parameters that can be tuned to the user's needs. One more positive attribute it has is that the replicating and erasure coding data availability schemas can be applied to different topologies within the cluster. For example, the replication can be done between OSDs, or hosts or even server racks. Those characteristics of CEPH are what make it desirable by many businesses and organizations, because it can handle any type of hardware, network structure and scale, as it can be parametrized for any data requirements they may have.

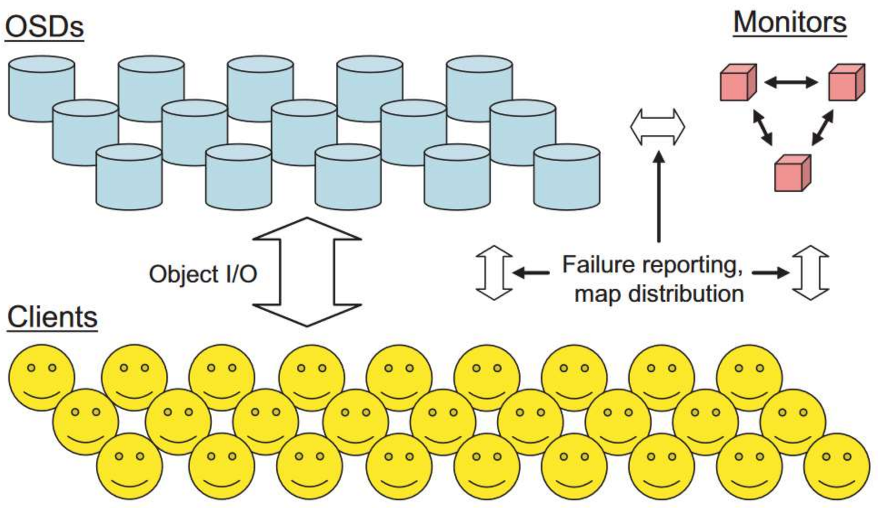

Figure 3: One example of a CEPH system. The users communicate with the monitors and the OSDs to submit I/O requests.

2.2 Monitor (MON) and Manager Daemon (MGR)

The process for the monitoring and the management of the nodes in the CEPH cluster is called a monitor. The monitor utilizes the PAXOS family of algorithms [9] for the distributed consensus between the monitor nodes, as seen on the above schema [11]. The role of the monitor is to detect changes in the CRUSH map and OSD map and act accordingly. The CRUSH map is the virtual structure of the OSDs in the cluster. This map has the form of a tree and its nodes besides the leaf ones are called "buckets" and the leaf nodes are the OSD ids. Below we can see an example for a CRUSH map:

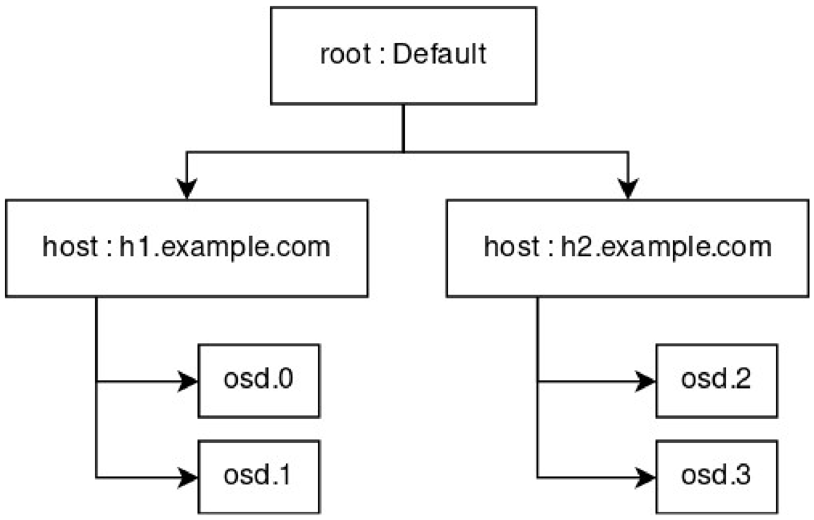

Figure 4: An example of a CRUSH map represented as a tree. The top node is of type "root", it has two children of type "host" and each host contains two OSDs.

In the above CRUSH map, the monitor can check the processes whether they are alive by sending heartbeat messages every couple of seconds. In addition, it is the role of the monitor to handle the requests for introducing new buckets and OSDs in the CRUSH map. If for example the OSDs 2 and 3 stop for any reason, the administrators of the system will get a health warning message about the second host “h2.example.com” being inactive in the cluster. Apart from the health checks, the primary reason of the CRUSH map is to be able to show how to place and replicate objects for the data pools. For example, if a data replication times three is needed in the above schema, it can be applied only on the OSD level, as on the host level there are only two hosts. To be able to replicate three times on the host level, we will need to have at least one more host added to the map that has at least one OSD. In this way, the businesses and organizations can create topological structures in order to partition the underlying hardware into different regions for controlled data safety and availability.

The way the data are distributed in the CRUSH map are called CRUSH rules. Those rules are defined in a simple descriptive language. In case an OSD daemon is killed or shut down manually for upgrades, the monitor picks up the change, follows the CRUSH rule that applies to the now inactive OSD, and finds another one to copy the replicated data in order to keep the data available. Thus, in the above structure if we have an OSD level replication and osd.2 is out for a long time, the monitors will replicate the data of osd.2 to osd.3, if the responsible group for the missing data were the first three OSDs. The OSD map has the list of all available OSDs and the amount of data they can handle in units of Terabytes (TBs). Based on some parameters the OSD map provides those rules pick the best candidate to support the replicated data.

When more monitors enter the cluster, a voting process starts to find the leader monitor. Usually monitors which were leaders before will stay as leaders to minimize the metadata exchange that costs network and time. Therefore, new monitors will always be peons and they may become temporary leaders if the leader node is failed. In any case a minimum of three monitors or a greater odd number of monitors is required to create a consensus. When the voting process happens, the cluster is in a frozen state and all I/O is paused. This is the way Ceph deals with failures, the I/O is paused indefinitely until a fix happens, to avoid generating errors that the clients would have to consider. Thus, the users from their side only see I/O halts and resumes instead of errors.

The manager daemon (MGR) is a separate process from the monitor, which takes care of external requests. In detail, these external requests can be either from clients or from 3rd party modules. These modules help extend the functionality of Ceph by providing the means to get the cluster’s metrics, support a sys-admin dashboard and further balance the data if it is required. Those previously mentioned tasks and more of them can be done by sending requests to the MGR daemon. Then, the daemon talks directly with the monitor to complete the request. Both the MGR and the MON daemons were once the same, but the MGR part got complicated enough to become its own process. They have to run together for the best performance. The MGR module is now the interface between third party programs and the cluster, as the whole CEPH group of CLI operations has moved from interacting directly with the monitors to communicate with the MGRs instead.

2.3 Object Storage Daemon (OSD)

The OSD is the process for handling the data and the underlying storage mediums. The OSD itself is considered as an unreliable storage system, this holds true as hardware and software errors can occur at any given point in time, therefore making the data on the failing OSD unavailable for some time. A cluster of OSDs that run along with the monitor nodes are considered as a reliable storage system because of the level of fault tolerance the system can support by introducing data redundancy. Each OSD process handles all the I/O queries for the part of the storage it is assigned to. The assignment is done by the monitor process through the CRASH algorithm. The OSDs handle a variety of I/O operations. Those operations can fall into two groups or categories and their combinations, read or write and data or metadata. An append operation would have a “write & data” flag and a get_omap (object map) operation will have the “read & metadata” flags. Omap is a key-value pair that shows the position of the object in the underlying medium. In a simple setup, the storage medium can be a disk, a disk with an SSD cache, just an SSD, or a logical volume with a complete custom layout, as an OSD could map for example to 2 or more hard disks or 1 large disk that can be split into multiple OSDs. The latter option would be the case where an SSD or NVME drive runs faster than what a single OSD can provide theoretically in a Ceph cluster.

Due to the strong consistency of the Ceph distributed system, the OSDs wait a lot of time in locks to ensure the data is written to all the replicas or stripes. Therefore, creating multiple OSDs that rely on the same storage media can improve performance by parallelizing the I/O on a single disk. It is advised to run a write benchmark first and check whether an OSD utilizes the full performance of its storage medium. Also, the OSDs rely a lot on working with their own metadata, as shown below in figure 5, and for better performance they should be stored on a fast storage medium, such as an SSD. To be able to retrieve the object metadata, there needs to be a directory within each OSD that maps the objects’ ids to the byte offset of the storage medium, this is the omap. Although, the omap is just a part of a larger key-value database that holds all the metadata of their respective OSDs. It is advised to run the metadata database of the OSD on a fast storage medium, and we can see that from getting the OSD’s performance counters, we can observe that the system refers the two entities as db and slow, where db is the metadata database and slow is the block device where the actual data are stored. For example, if an OSD resides on an SSD for the metadata and an HDD for the actual data, small writes will never have to go through the slow disk and they will stay in the metadata database, therefore the performance can be improved significantly. Once those key-value pairs of object id and object data entries start to occupy enough space, the OSD will transfer the data back to the HDD and replace the entry with an omap pair. Practically, the database size should not be less than 4% of the actual OSD size[6].

The way the data is stored in the OSD is an implementation that involves a simple filesystem and an embedded key-value metadata database on the disk. Ceph has two types of disk formats, FileStore and BlueStore. The Filestore format has Level DB as the metadata database, and XFS as the filesystem for the objects that are stored. This format is still used but it is proved to be a bottleneck in the I/O performance of an OSD. The new format is called Bluestore, where the metadata database is implemented with RocksDB and the objects themselves are stored with a format called BlueFS. The previous filesystem, BlueFS, is essentially a stripped-down version of a classic filesystem that supports the most basic operations an OSD needs. The error handling is done with CRC32c checksums [24]. Bluestore manages to be faster than Filestore by avoiding the double writes of journaling filesystems [16], it handles better the metadata with RocksDB which is an improvement to LevelDB and it uses copy-on-write for creating faster snapshots. The performance difference is about double [15] and we will implement our recovery optimization only on this format. We chose Bluestore, because we leverage its flat architecture, compared to Filestore’s hierarchical one, and we can filter the necessary objects for our recovery implementation effectively with a logarithmic cost.

[6] https://docs.ceph.com/docs/master/rados/configuration/bluestore-config-ref/ block WAL size

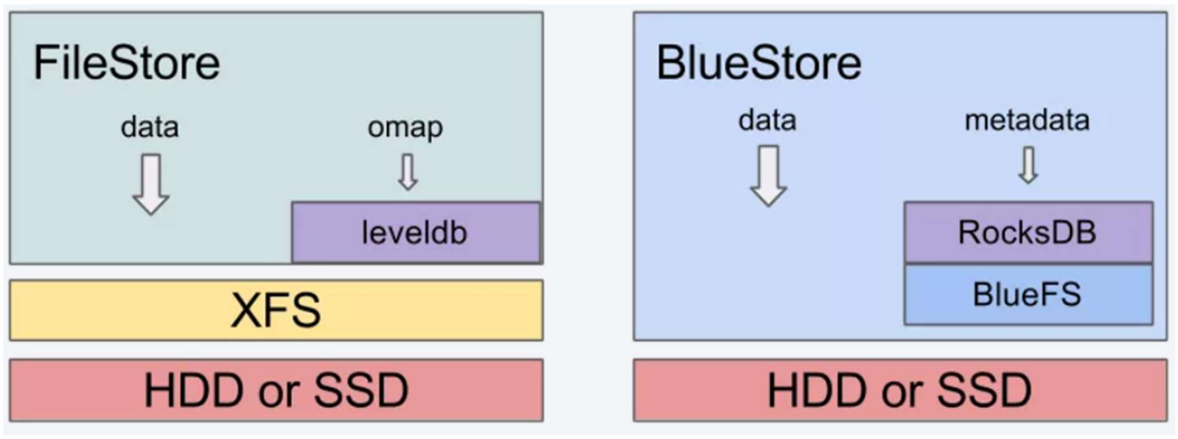

Figure 5: The differences between the structures of Filestore and Bluestore, the data backends for CEPH's OSDs.

In master/slave distributed systems the slave node usually has a passive role regarding its activities. They receive the master node's queries as they serve the elected node [23]. In the Ceph distributed system the slaves, which are the OSDs, they are autonomous compared to the classic model. The OSDs can trigger tasks within the system, such as recovery, and they can communicate directly with their neighbors without the constant supervision of the monitor, as seen in figure 6. The clients, upon interacting with the monitor, they receive a map of the OSDs. Finally, the clients can directly query the OSDs without having the monitor as the middleman.

The software model of the OSD is the classic multithreaded model that implements a finite-state automaton. Each state has three stages, the first one is the state transition in which the automaton enters a state, the second stage is when the machine is inside a state and the last stage is when the OSD exits a certain state. Each stage will trigger events, and those events will interact with different OSD nodes and they will create, modify or finish various tasks. Those tasks can refer to data recovery after a failure or a series of object checksum checks called scrubbing. Object scrubbing can have two forms, one that just tries to validate the stored checksums between replicas, or the one that is recalculating the checksums and is checking the results with the already stored ones. Scrubbing can help to detect any filesystem or hardware errors that may occur that alter the data in a non-trivial way. The OSDs which are up and running and are on par on their data with their replicas, they have the active+clean state. OSDs can have more than one state and each group of states has its own purpose. An OSD that is not responsible for any kind of data, maybe because it belongs to an unused root in the CRUSH tree, will have the inactive state. People who administrate Ceph clusters can easily understand the status of each OSD based on the states that it belongs to. For example, an active+scrubbing+clean state model can show us that the OSD is currently performing the light scrubbing tasks and it has no stale data. This way, the tasks and the actions that an OSD will take are more structured and from the developer side, the debugging becomes easier for this complex subsystem that Ceph has.

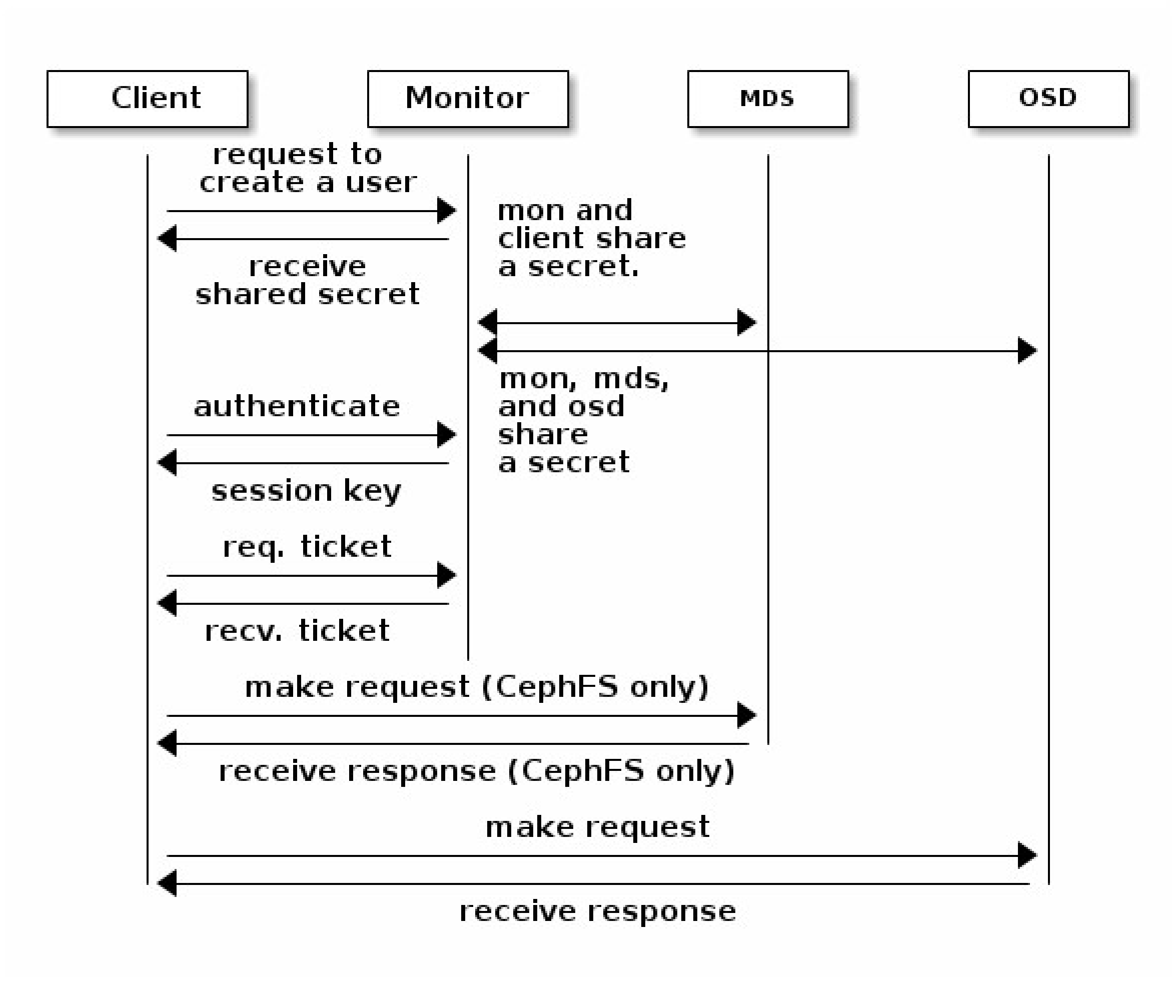

Figure 6: The direct communication of a client with an OSD after their authentication.

For storing and manipulating data, the Ceph system relies on Rados, the base layer of Ceph which consists of all the OSDs. Rados is a semi-autonomous system that can both serve and manage tasks. Figuring out the extent in which a role, such as an OSD, can operate and how it can interact optimally with the others, is a key feature for scaling well distributed systems. For example, in CERN, one cluster which serves CephFS can operate with 5 thousand peak IOPS using 6 OSD hosts of 24 OSDs each, 3 Monitors and 3 MDSes. After explaining the system in detail, we can now discuss the main subject of this thesis, the backfill recovery and how it can be improved.

3. MECHANISMS FOR DATA CONSISTENCY

3.1 Introduction

The Ceph storage system has two mechanisms for synchronizing the data between replicas and keeping the data consistent. These two procedures are called recovery and backfilling. The recovery mechanism uses an in-memory write-ahead log (WAL) to determine the recovery process [25]. In addition, the recovery in Ceph should be a quick process, therefore only a configurable amount of entries is kept. Usually, the number of WAL entries an OSD has a threshold of 10 thousand. Those entries are about 200KB each[7], and as they are growing, they can fill up the memory, with the previous upper limit a log can consume around 2GB of memory in a short amount of time. The limit imposed to the number of WAL entries was a soft limit, therefore a process that quickly fills the entries before they get trimmed can die from a shortage of available memory. In normal circumstances, when the WAL entries exceed the upper limit, the entries are discarded for the sake of memory consumption and the backfilling process starts. The backfilling process is the slower synchronization process of the two and it checks the data and metadata object by object of each replica. Then, it determines which entry should be replicated, so the degraded data gets overwritten and the missing data gets filled, soon after the replicas will eventually become synchronized. Finally, to be able to describe the backfilling process, we will need to explain the CRUSH algorithm first, then the level of granularity for data replication and finally the peering process, the task that triggers when an OSD enters the cluster.

3.1.1 The CRUSH algorithm

Controlled, scalable, decentralized placement of replicated data (CRUSH) is the algorithm which distributes the data amongst weighted storage devices for a uniform distribution. The storage devices are the leaves of a hierarchical cluster map, called CRUSH map, which represents the structure of the cluster, as someone can create various topologies to group storage devices together and create relations amongst them. In a real-case scenario, those logic groups can represent a host, a rack of hosts, a row of racks, a datacenter room and so on. Those groups are called buckets, not to be confused with hash buckets, and they are used in a way to set data redundancy schemas on various levels. Additionally, these buckets can define storage targets which share the same group of electrical or network switches, therefore a data replication across a group of storage nodes within different network switches would be advised to keep the data available. The weights also of the buckets are calculated bottom-up from the OSD level and each bucket has the sum of all the weights of its children. For example, a unit of weight in a Ceph cluster is set to be equivalent to the unit of one terabyte of data, thus an OSD with weight 3 has a capacity of 3TiB. Also, the data distribution is done using placement rules, which define the start and end points of the algorithm's traversal through the CRUSH map's tree. The output of the algorithm will be an ordered series of storage targets, as the output is determined from using an identifier as an input, the cluster map (CRUSH map) and the placement rules (CRUSH rules). The identifier can be an object's name or an identifier for a group of objects. For scalability reasons the replication is done in the placement group level rather than the object level. Placement groups are the hash buckets of RADOS, they hold a series of objects ordered by their hash values. The object's hash is a 32-bit integer that is calculated partially by the object's name, that is given by the object's creator, which can be a client or the system itself, as it can generate temporary

[7] Empirical value stated by the designer of Ceph, Sage Weil

objects or split CephFS files into multiple objects. A placement rule for data redundancy with replication can have the following form:

1. take root default
2. chooseleaf_firstn 1 critical type row
3. chooseleaf_firstn -1 regular type row

The above set of steps will place one copy in a row of racks called critical, maybe because the racks there could have a power backup for some additional protection, and then choose N-1 copies to be stored on the regular row of racks. For example, if we create a data pool with 32 placement groups and times three replication schema, one copy of the PG will go to the critical row by choosing pseudo-randomly a leaf (OSD) in that row. Also, the other two copies of the PG will be handled by two OSDs that are in the other two rows of the cluster. To be able to store data in the data pool that uses the above CRUSH rule, it is required that the cluster map has at least 3 buckets of type row that also contain OSDs. Those OSDs have a different ID notation so they can be distinguished from the buckets in the CRUSH map. For instance, the buckets use negative integer IDs and the OSDs use positive integers. This is a simple and efficient way to determine whether the given CRUSH component is a bucket or a leaf node.

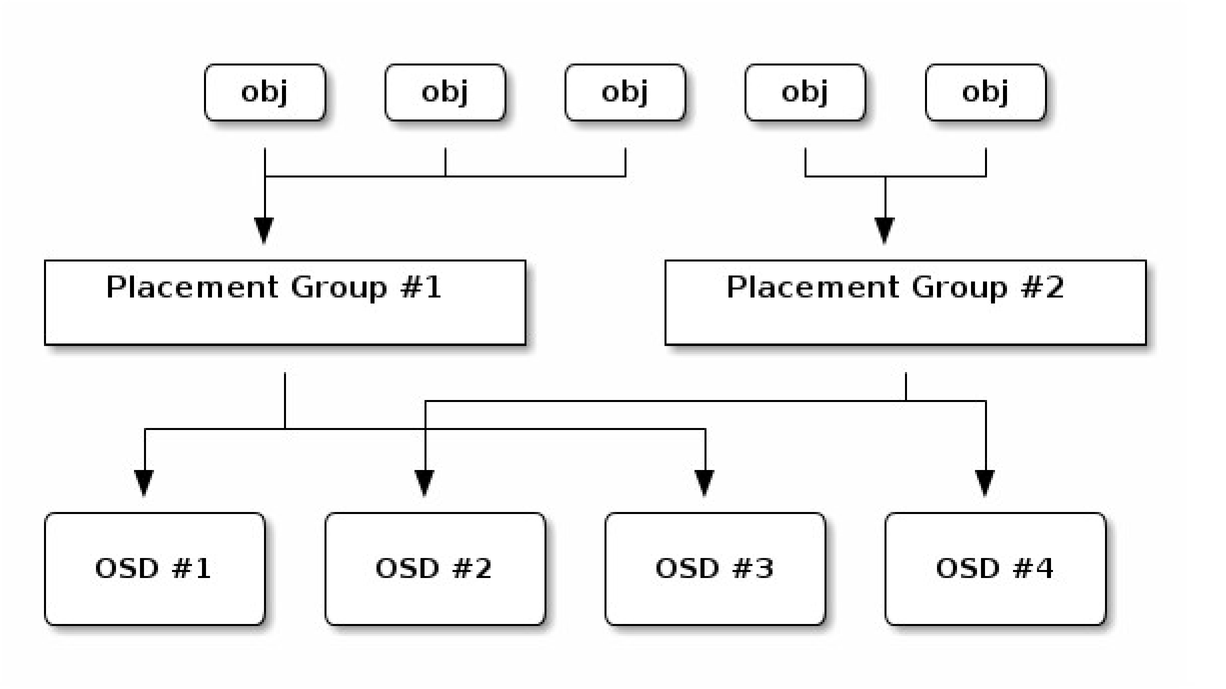

Figure 7: An example of objects being assigned to PGs, and PGs being assigned to OSDs through the CRUSH rules. In this example the PG #1 will be replicated to OSDs #1 and #3.

Typically, an OSD by default is set to handle 30 to 100 placement groups. This PG range is an empirical value that was obtained from a couple of experiments done from the developers of Ceph. If those numbers are exceeded, the OSD will be regarded as an underutilized or an overutilized one by the cluster. Up until version 13 of Ceph codenamed “mimic”, the administrators would have to calculate the number of PGs beforehand when creating data pools. From version 14 codenamed nautilus and onwards, the extension and merging of the number of placement groups in pools is possible, thus they can be dynamically changed based on their load. This way, the OSDs will be able to perform queries with more efficiency, as the OSDs would have on average an optimised workload.

Also, a balanced OSD is when its PGs are equally filled with the same number of objects. From a real-world experience, an equal number of PGs across all OSDs doesn't necessarily mean that the workload would be evenly distributed. There are a lot more factors that can determine an even workload. Additionally, the CRUSH algorithm places an object in a PG based on the hash ID of the object modulo the PG number of the OSD. The ceph-balancer module uses more complex heuristics in order to achieve a close to perfect data distribution across all the OSDs. This module creates mappings from a set of OSDs that the CRUSH algorithm picked to another, by prioritizing replacing overutilized to underutilized OSDs that the algorithm has left behind.

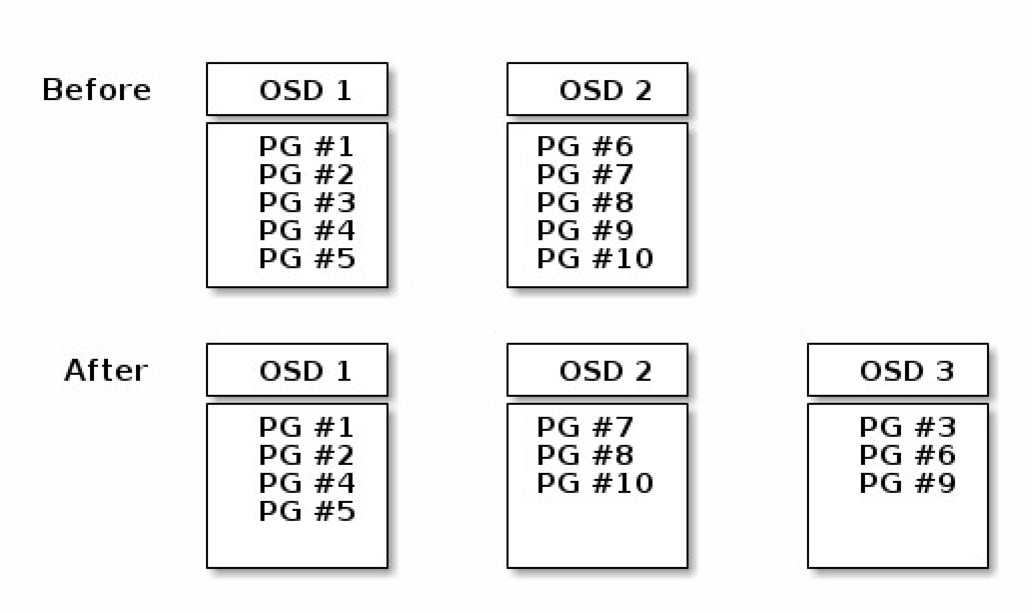

Figure 8: When a new OSD is added to the cluster, the CRUSH algorithm is run again to calculate the new PG locations.

The CRUSH buckets are divided into three types, the List buckets, the Tree buckets and the Straw buckets. The Straw buckets are the default choice because of their better average performance. The straw algorithm follows the idea of picking straws with variable length, and the item that picked the longest straw will be chosen to store that object for the cluster. The length of the straws is limited to a predefined range using a hash function of the object's hash, the replica number (e.g. 2nd copy of the 3) and the current bucket in the list. This hash output is then scaled by the weight of the bucket, so buckets that can hold more data are more likely to be selected to store the objects. Even though this process is at least two times slower than the one with list buckets, it is proved to be the algorithm that moves the data with the least cost when the cluster is changing size by either expanding or shrinking the number of OSDs [7].

CRUSH is a highly scalable algorithm for data placement on distributed systems. The data placement itself is one of the challenges of an efficient storage system. The CRUSH algorithm achieves a fine placement technique that is based on a pseudo-random algorithm on weighted sets of storage devices. Additionally, CRUSH can handle the overfilling of the devices by avoiding any new data placement on them

3.1.2 Placement Groups

The PGs are the hash buckets of CEPH, they accommodate many objects sorted by their hash value. The PGs are the units which get replicated or erasure coded across the cluster, and consequently the objects contained within a PG are replicated as well. This model scales better than a simpler model where the monitor daemons would have to keep track of all the objects individually and manage them directly. Additionally, the amount of memory required would increase with the number of objects stored. In large clusters, the monitor nodes would struggle to cope with those constant changes. Typically, in most real-world applications, the object storage of Ceph is not used directly with Rados, but the CephFS, the S3 and the RBD parts of Ceph are mostly in use. Therefore, the mapping of a “file” to an object is up to the application, as Ceph will internally split a file into multiple objects, typically into chunks that are multiple of the minimum allocation size of BlueFS. Another fact for the PGs is that they are the ones that implement the finite-state machine and not the OSD itself. Operations such as scrubbing, which is the procedure of checking the checksum of objects for various errors, are performed in the scope of PGs. Typically, BlueFS will provide a backend mechanism in which an OSD can query a series of objects to fetch from a PG. The operations are done by traversing the object’s hashes in ascending order. The placement groups exist to allow for batch operations on the cluster by the OSDs, so each procedure that would apply on an object granularity instead can now scale better.

When a client writes an object to the system, a primary OSD always must exist, which implies that it contains a primary PG and the data will be written there first. From this point of view, the primary PG will write the data to the other replicas and it will get the acknowledgement of the writes from the second and the rest of the PGs. When all other replica PGs agree that they have committed the object to their backend storage and sent a confirmation to the primary PG, the primary OSD will report back to the client that the object has been written. All the write I/O is done first on the primary PG, then it propagates the writes to the replicas and collects the commits as shown on the below figure 9. If a primary PG is down, which means that the OSD it is assigned to is down, CRUSH will select a temporary primary PG, and the replicas will try to synchronize with the new primary. There is no voting mechanism between the OSDs to determine the primary PG, the monitor will be responsible for selecting the next one. The PGs by design have a soft limit to the number of objects they can hold. When a certain threshold is exceeded, a procedure will begin called PG splitting. The PGs are split in order to perform tasks within them faster, so we can create better recovery queues. If a PG is loaded with many objects it is most likely to have a deteriorating performance. Hence, splitting the PG will distribute the workload and avert any heavy loaded PGs. This process is quite slow, as it needs to read all the data from the PG, to create a new one, to calculate the data movement using the CRUSH placement rules and finally to migrate the data aimed for the new PG. The data migration is done with the backfill procedure, we change the state of the PG to show that is missed some data, so backfill recovers them from the replicas.

The placement groups are a necessary tool in order to scale the object storage of CEPH effectively. The automatic management of the data by the PGs is what makes the system stand out from the other distributed storage systems. Therefore, this automation leads to a strong consistency model and a highly available peer to peer data backend, provided that there is always the bare minimum number of nodes required for any precalculated failure scenarios.

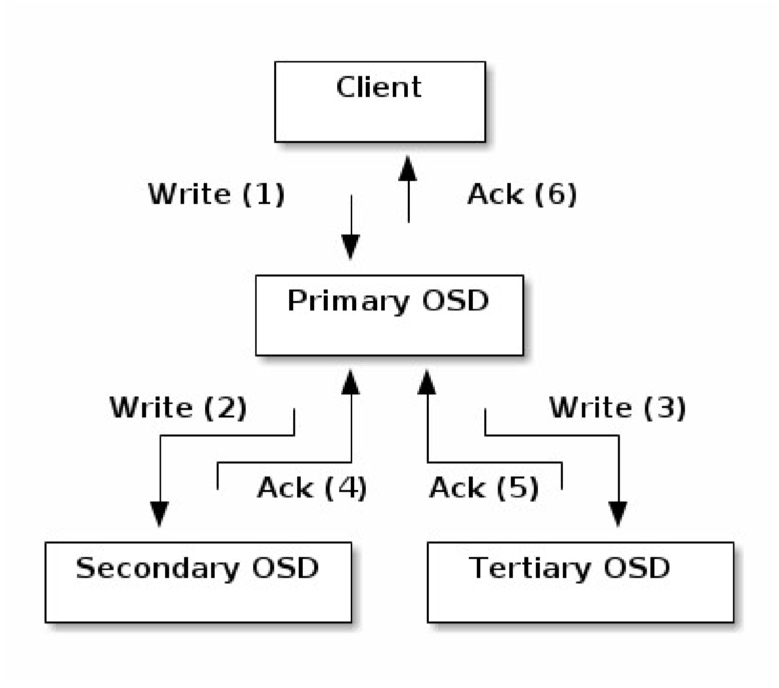

Figure 9: Example of a write operation being forwarded to the OSD replicas. The numbers indicate the order in which the messages are delivered.

3.1.3 OSD/PG peering and recovery

As constant changes happen to the cluster, such as newly added storage or an OSD startup after a failure, the affected PGs must perform a task called peering. Peering is the state in which a PG tries to synchronize with its peers, which are the other replicas, in order to find a consensus for the data held by each replica PG. In other words, the peering state is the state in which a PG tries to enter the cluster and synchronize with its peers. When an PG retrieves an OSD map, a change known as a new epoch, it needs to recalculate its data with the neighboring PGs. The PGs can be either primary, replicas or stray PGs. Firstly, stray PGs are the ones that were part of an active set, which is a set of PG replicas that handle a set of objects, and now after the new CRUSH placement calculations, they will not receive any data. For example, a set of stray PGs is the one that has been out of the cluster, by moving the OSD bucket out of the root bucket of any pool, and those PGs haven't been instructed to remove their previous data yet. Secondly, the primary OSDs are the ones that handle all the writes and propagate them to the replicas. Lastly, the replica PGs are the ones that replay what the primary does for data redundancy.

The peering process is always started by the primary OSD. All other OSDs which are not primary, gather a set of metadata and send a notify message to the primary one. This message contains the necessary information about the PG's state, such as the most recent update, the PG WAL log and the last known epoch in which a PG has successfully peered. Once the primary PG receives the messages, it creates a new active set of OSDs for that PG and a prior set in which the primary had known from a previous peering procedure for that particular PG. If both sets of OSDs are not the same, the primary will begin a backfill process to move the data. If P is the prior set and A the active set, the primary will begin a backfill process to move the data from each OSD in P-A to each OSD in A-P, the pairs will be chosen based on the replica number in ascending order. As the primary has now an overview of the PG state, it compares all the WAL logs that were sent to it, in order to determine the latest common point and read the PG logs from that epoch and onwards. If any differences occur, the primary will trigger a recovery to replay the logs on each PG to resolve the data discrepancy between the OSDs.

To minimize the impact of recovery on the performance of the cluster, a distributed system such as CEPH must implement a parallel failure recovery mechanism. As the PGs are spread all over the cluster, neighboring OSDs which share the same PGs with the failed device will recognize the failure, because they send frequently heartbeat messages. Once the failed OSD is recognized by the other OSDs, the recovery will start by comparing the in-memory logs to find the state of the cluster. Because of the distribution that was made with the crush rules, the OSDs which have previously failed will pull data and neighboring active OSDs will push data at any given time. Each new write will be added as a new WAL entry in the PG log, since each PG replays the log to synchronize with its peers, thus the recovery has no need for a write locking mechanism. Primarily, the recovery throughput is limited by the read I/O and not the write I/O.

The most important aspect of the recovery is to find the differences between the two replicas. There are two limitations for the recovery mechanism. The first one is if two different OSDs attempt a recovery procedure on the same OSD, there will be a waste of resources for seeking and reading on the block device for the objects. This waste is caused by the redundant operations that can be created by the two active OSDs, which can happen when the two individual OSDs do not choose the same objects to recover at the same time. The second limitation is that if an OSD is missing, the object update procedures based on [26] [27] replication schemes implemented in RADOS will become increasingly difficult and complex. Hence, the recovery mechanism is managed by the primary PG as it coordinates its replicas. Additionally, the actions on missing or degraded objects are delayed until the primary PG has a local copy. That can happen when the write was done to another replica PG and the primary must catch up to that update by pulling the change from the other PG. Then, the primary pushes that copy to its peers ensuring the read of a missing object is done once. Since each degraded/stale object will be read once, the recovery mechanism will run optimally.

The peering mechanism is the first action that runs on the OSD and it triggers the recovery mechanisms, WAL or backfill, if necessary. In our implementation we chose to create just a working example for demonstration purposes. Thus, we embedded the exchange of metadata inside the backfill procedure, where in principle this action should have been its own state that the PG would enter from the peering process.

3.2 Recovery with Backfilling

Backfilling is the procedure that this thesis will try to optimise. This procedure is triggered by the recovery mechanism when there is not enough information from the PG metadata, such as the PG log and last peering epoch, to determine how to keep the replica PGs in sync. Backfilling is a heavy batch operation that scans the data of each affected placement group and copies or recovers the correct objects. This procedure will always run on newly added storage, where in that case there is no optimization to be done, as every object in the affected PGs will need to be read and transferred to the new replicas. All the Ceph objects are represented by a class called hobject_t (hash object type) in the source code, which contains the name of the object which is the key, its hash value, the data pool it belongs to and if it is the last object of the PG hash-wise, determined by a Boolean value called max. As every max object is treated equally in each case, the hobject_t for the max object is acquired through a singleton class. The backfill is coordinated by the primary, since it will reset any previous backfill info and it will start a new process. The primary will pull object metadata from each other replica from objects with the minimum hash until the hash max is found. If a PG has returned a hash max object, this means that the backfill recovery is finished for that replica and all objects were checked.

The primary PG has a variable that we will call backfill “next” which is the counter of the hash value of an object that has been recovered already between the replicas. Considering the way of ordering the objects based on their hash value, if an object is being updated while the recovery happens and the next backfill object has a latter hash value, that update can be done without impacting the recovery procedure. Therefore, updates which are after the next backfill object will not be propagated to the replicas, they will have to be persisted by the primary only, and then they will be recovered once the next backfill object passes through that hash value in a future time. The profound limitation is that those writes will not be available if the primary OSD fails, and they will be applied only once it is back up again. In case of a storage medium failure, those updates will be lost. That is one disadvantage that the backfill process has, so it doesn’t block any writes while the PG is recovering using the backfill method.

Backfill is now a background process that has a substantially better performance compared to an earlier implementation that blocked the client writes for each PGs. Each PG starts two asynchronous reserver agents which pull and push data. The backfill starts when the primary has done its local backfill reservation. Those reservations are a set of data that will be backfilled in an instance, which is essentially a batch operation. When too many reservations are done and the OSD utilization calculations go above 95%, a backfill_full warning will arise and the OSD will block any I/O done, since it will not have any usable space left after the backfill reservation finishes. In this case it is advised to expand the cluster, so it can handle larger failures. Once the primary has done its local reservation (pull) on a new backfill procedure, it resets all the metadata it has from a previous backfill. Those reservations are in a priority queue, because some reservations are more critical than others. For example, if a backfill is done due to hitting the lower limit of replication for data consistency, those reservations will be carried out earlier than the others, as they will have higher priority in the backfill queue. The primary has a class called Backfill Interval which executes a reservation and keeps track of the backfill progress in that batch of objects. Also, the primary has an array of those types of classes for each peer, to keep track of their progress. Those classes hold data such as the starting hobject (hash object) and the last one in a list. If the max hobject is found in the end of this list, this means that this is the last backfill operation for a particular replica PG. Otherwise, if the end hobject is not found at the end of the list, the primary will ask the next batch of objects to be checked from that replica. In addition, the primary has the backfill next counter which is constantly increasing to the next object hash. When the primary requests all the objects for the next available hash range from its peers, it will check the versions of each object to see if they match with the latest one.

Each object has a version which describes the epoch of the PG when the modification was made. For each batch step, the primary will create 3 lists of objects, one that will keep the objects intact, one that will recover the object because they have mismatched versions with the peers and the last one which is called the missing set, that is the set of objects that have not been written yet to the replicas. In the first set for example, there are the objects that appear in primary’s next and the replicas’ next and have the same version. Then, in the second set there are always the objects that have been altered and they need to be updated to the new version (version mismatch). Lastly, the missing set can be easily distinguished as the primary will ask the new next from its peers, and some of them will have an object further in the hash range, so they will be missing the one primary asked for. Once those three groups are created, the primary will move those lists to the remote asynchronous reserver in order to push the changes to the replicas. When the batch operation is finished and the primary has cycled through every object of its peers' backfill intervals, it will update its local last backfill hash. This hash will determine the starting point of the next batch operation. If the primary has finished the batch operation but there are still objects to be backfilled, it will warn its peers with a Backfill Progress message, or when the primary has the last backfill hash as the max hobject, it will send a Backfill Finish message to notify the PG it tries to synchronize from that the backfill operation is finished for this placement group.

Backfilling is a massively parallel and heavy operation, since it requires each affected PG to be read for each OSD, thus the backfilling operation has a great impact on the I/O and network performance of the cluster. Nevertheless, by changing the Ceph's osd_max_backfills parameter, one can limit the amount of parallel backfills done per OSD to prevent any increase in the latency of the client's I/O requests while the cluster is performing backfilling. By configuring this variable, the backfilling process will slow down, to reduce the presence of the recovery in the cluster's I/O bandwidth. Even if the I/O performance for the clients can be improved by reducing the amount of backfill reservations, the recovery time will further be increased as the time passes by, because more objects will be updated on the recovering PGs. Therefore, by optimizing the performance of backfill we can further reduce its time to complete, as less writes will happen while the recovery happens, and the less chances those updates will affect the current recovery and extend it. Therefore, we will be susceptible to less errors while the recovery happens. The optimization of the backfill procedure will require to check less objects and not scan the whole PG. Finally, it has been observed that the backfilling procedure is run often, and its average performance could be improved.

3.3 Comments on Backfill

The recovery mechanisms in Ceph are highly consistent and stable enough to be used in the industry. Ceph mainly focuses on data integrity and availability, so it has to sacrifice a little bit on the I/O performance side. Therefore, we can try to introduce some performance improvements on the cluster's recovery without having to sacrifice the data availability. By improving the recovery process, we put less stress on the cluster and we avert any kind of bottleneck on the client I/O performance.

The recovery process in the past had some memory issues which led to hosts being out of memory. This memory usage problem was observed when object writes or updates are appended to the PG log with a fast rate. That rate was enough to fill the memory before the faulty OSD gets fixed or replaced. Hence, all the up and running peer OSDs are filled with PG logs that can't get trimmed as the faulty OSD is still in down state. One solution would be to persist the PG log to the disk, to free the memory and reduce its usage, but this implementation would increasingly complicate the logic to handle and verify the integrity of the logs on the disk. In addition, the management of the entries would complicate a bit, so that they do not exceed a certain space limit. In conclusion, this implementation would just carry the current problems from the memory side to the disk one and create additional complexity. Therefore, the Ceph development team has decided to impose a hard limit to the PG log, so that the OSDs will be able to have an acceptable memory usage. Once this hard limit is reached, the PG log gets discarded and the recovery mechanism is delegated to the backfilling procedure. The PG log can be discarded either by reaching the hard limit or by being used long enough to be considered stale. Therefore, with the increasing I/O bandwidth of the current storage mediums, such as SSDs, and the slower rate of memory capacity, the PG log would be discarded at a faster rate and the backfilling process would take place more often. Thus, it is necessary to improve the recovery operation to be able to shorten its duration and the change in client I/O bandwidth will become less noticeable.

In order to shorten the backfill process duration, it will be necessary to identify which part of the backfilling process is present more in the recovery duration. As mentioned in the previous section, backfill requires reading the contents of all the objects which reside in the "slow" storage of the OSD. Thus, the procedure to read each object of each PG on each affected OSD is one of the heaviest tasks of backfill. Additionally, failing OSDs on average will share some identical data between their peers when they get up again, because the chances for the whole contents of the OSD to be updated during the failing period are minimal. Nevertheless, the more time an OSD stays down, the more data will be changed, provided that there is a write workflow from the clients during the failing period. Therefore, by identifying the parts which remain the same and skipping them from the backfilling checks, backfill will require much less reads from the "slow" storage and the overall performance of it will improve. Thus, if we identify which objects have not been altered, we can filter them out of the object scan the backfill performs, resulting is shorter recovery time by wasting less resources (I/O, network).

Because CRUSH has a good distribution amongst the objects in the OSDs, with every second that passes, the writes done to the OSDs will spread evenly across the PGs, making the identification of the changed parts increasingly hard. One solution would be to cut each PG into pieces which represent a group of hash ranges and check if they are changed. Knowing the average time of a failed PG and the average write throughput, one should define a number that cuts the PGs thin enough, so the even write distribution won't invalidate a large set of the PG. Let's say we have a data pool with p PGs that has a replication factor of R. Additionally, each PG has on average k objects. On each second that passes, the clients are updating or creating r random objects in total. The maximum number of objects changed in t seconds on each affected PG can be estimated by the following formula:

$$\boldsymbol{N_{stale} = min\left(\frac{tr}{pR}, k + dk_t\right)} \text{ (1)}$$

Each object is assumed to have an equal chance to be handled by one of the PGs, thus the number of OSDs or the number of down OSDs does not matter. We assume that this value is the same for each PG, and we count is as the maximum in case each change was done on a different object. The variable dk_t denotes the difference of the total number of objects in each PG in the course of t seconds. This equation just shows the estimation of what the maximum number of stale objects would be in each PG, given a steady write workflow. In addition, the duration of the recovery will also depend on the number of parallel PG backfills at a time. To give an exact number for $N_{updated}$ we will need to introduce more parameters that describe the write workflow with more accuracy over the time frame of the recovery.

For example, let's say that we have a pool of 128 PGs that have 50K objects each with a replication factor of 3 copies. Also, the client I/O is steady at 1000 write IOPS (input/output operations per second) and the failing OSD is down for an hour, therefore the maximum amount of objects updated on each PG will be about 9K out of the 50K, which affects almost 20 percent of the PG data. For the worst-case scenario each object will be approximately changed every 5th hash number in the PG range. Hence, if we cut the PG in 10K pieces, each piece will contain at least one changed object, which will not provide any performance improvement, as every range will be changed and it will have to be checked. If we cut the PG in double the pieces, we can already skip half the objects of each affected PG. However, the problem is that we can't predict the client I/O, as we can't predict which objects will land in which PG. As the straw bucket algorithm is used, we can't know beforehand the minimum and the maximum hash that each PG will have, as the objects will be placed randomly within the PGs. Even if the first approach, which is naive, can achieve a deterministic 50% cut in the recovery duration, the implementation will be quite inefficient, because we will have to adjust the number of ranges on each backfill, which will require many recalculations, and also by cutting equal hash ranges we cannot guarantee a constant X amount of objects in each range. To conclude, we will cut each PG using the full 32bit range, and we will have to choose a number big enough to create unchanged ranges that can be skipped and small enough that does not leave a large memory footprint.

3.4 Improving backfill

As discussed above, a good approach to improve the backfilling process is to be able to expose sets of objects that will remain the same even after the failing node comes online again. In order to accomplish this, there must be a way to view the PG as another kind of container, one that can detect and record changes within it. One way to achieve this would be to represent the PG info with a bitmap. The bitmap will have the value 1 as a change to a set of objects and 0 would mean that the subgroup has not changed at all. Each cell of this bitmap will map to a subset of the 32-bit hash range. This bitmap will be called objects' info (OI). Once a write is made, there is an easy way to find which cell needs to be changed in O(1). The below C-style formula can describe the change in the bitmap.

$$\mathrm{B}[\mathrm{hash}_{32}(\mathrm{obj}) \gg (32 - \log_2 N)] = 1, 0 \leq N \leq 32 \quad (\mathbf{2})$$

The variable B represents the bitmap, N represents its length. Once a write to an object is made, the subgroup on the bitmap will be changed to 1. If the bitmap divides the 32-bit range into 4 groups, we get 4 groups of 2^{30} length. The bitmap with length 4 can be represented by 2 bits. Therefore, if we shift the 32-bit hashes of objects into 2 bits we can map all of the objects' hashes into 4 groups. Therefore, the change of the bitmap can be done directly, without the need of O(N) traversal. It is obvious that the bitmap length cannot exceed the length of 2^{32} as there will not be any mapping left for the objects after the right bit shift (marked with >>). We will tend to avoid creating any bitmap with lengths close to the power 2^{32} in order to avoid redundant memory consumption. In addition, to make the calculations fast and precise, we will have to limit the lengths of the bitmap to be powers of two. Thus, these bitmaps can now be sent over the network to identify the regions that should be backfilled. Based on the first formula, the bitmap should not be a small value as it will not improve backfill at all. For example, if we cut the PG into four ranges, any four writes have a chance to change the whole bitmap and not improve the recovery at all. Thus, by choosing a larger number, we are creating smaller ranges, but at the cost of sending larger amounts of metadata over the network. On one hand, if we choose a bitmap of size 2^{20} and the PGs have an average size of 200k objects, there is a high probability that each object will be in a different range, and the bitmap will be about 19% utilized. Hence, the write changes can be detected per object. On the other hand, if we choose a small number, such as 4096, the objects would utilize most of the ranges.

Hence, a change in a range will require about 2% of the objects to be checked for backfill instead of the whole PG. Nonetheless, in a real case scenario, the first choice would be the worst one, as the objects go in a PG based on a modulo operation, therefore each PG can't have two objects with successive hash numbers. The modulo operation is unknown from the OSD side and thus we can't implement a trick to reduce the bitmap size further to ignore the elements that don't have that modulo remainder. Either way, the PG number per OSD can change in some other background process, therefore any information on the hash values of the expected objects cannot be obtained.

Even if the changes can be represented by ones and zeroes, the cluster may be in the same state after some writes have been made. For example, Ceph creates temporary objects that may stay alive or not until the replica has recovered, a client may open a file with an editor and do nothing, but the editor creates a temporary recovery file that gets deleted upon closing the program. Even if the temporary files may be in the same path as the edited file in CephFS, it can hash into another OSD that may be currently in a degraded state. Also, in certain periods of time some changes can cancel each other, and the contents of the cluster can remain the same. In addition, Ceph has a version system for all its objects, as each change on the object will increase a metadata counter by the PG version. The objects don't have their own versions, rather than they inherit the PG one. The PG whenever a change is made will increase its version by one, and the object that triggered the version change will get that value instead. Thus, we can't assume that the versions of the objects are continuous, an object for example can have the versions 0, 1, 5, 12, as a sequence and as we see we can't assume the previous version of 12 is 11. The objects can also revert to previous versions due to recovery from snapshots. Therefore, we can formulate the state of the objects in such a way that we can detect temporal changes that do not affect the data of the replicas.

In general, if two replica PGs have the same versions of objects, the system guarantees that their data is the same and therefore they are consistent. By using this knowledge, we could keep the object hashes and their versions in some hash table, but that would be very inefficient memory-wise. Furthermore, there will be the need to persist that hash table in a case of a failure, because it will be slower to get the versions of the objects by reading each one's metadata again. To improve the previous solution, a signature can be created instead by hashing the objects' ids and their versions. Additionally, to avoid possible collisions the result hash could be in a greater range such as a 64 or a 128-bit hash. Therefore, the replicas can detect the respective hashes, in order to check whether they are consistent or not. For example, to test for possible collisions, a test was created using a simulated PG that gets constant changes and recalculates the resulting 64bit hash signature from the objects' hash and version pairs. The system ran out of memory (16GB) before it encountered any colliding hash. The same test was conducted with a 32-bit signature hash, but a repeated hash was found in a matter of seconds. Therefore, it is not safe to use a PG hash that has the same size as each individual object's hash. Therefore, using a 32-bit signature hash for the PGs would damage the consistency of the cluster, until a periodic scrubbing process detects the actual differences between the replica PGs. Therefore, we will create 64bit hash signatures for the PGs instead of using 1 and 0 to detect changes between replicas with more detail and avoid scenarios in which changes were reverted and the PG is at the same state as before.

The process of hashing the objects and checking their resulting signature is not something new, it is a common task that version control systems like GIT do [28]. GIT hashes the files of a certain repository, which is the root directory of a project's source-code. Usually, programmers have a remote and a local repository that they work on. In order to have a coordinated project, the remote repository must be consistent with each user’s local repository. As a local repository may be behind or ahead from the remote one, the way to detect the changes is to check the resulting hash of each HEAD of the repositories. HEAD is the last commit applied in a repository and the last state of it. Finally, we can map those principles to Ceph, by hashing the objects and their versions and getting a HEAD hash for each PG to be able to tell whether they are consistent or not with their peers.

3.5 Merkle Trees

For the Git system to detect the differences in repositories which have not been synchronized, it implements a Merkle tree [29]. The Merkle trees were invented by Ralph Merkle in 1979. They are the trees in which their leaf nodes represent the hashed value of a data block and the non-leaf nodes represent the cryptographic hash of their children. Thus, the top hash has the combined hashes of all its children. This structure can enable a cryptographic verification of the data blocks, so someone can verify the integrity of the data based on the calculated hashes. Additionally, in secure peer-to-peer networks, the Merkle trees can be used to detect any malevolent intent by checking whether the data received is valid or not. For example, in Bitcoin and some other modern blockchain technologies, the Merkle Trees are used to avoid malicious people who try to deceive the system [38].

In our storage case scenario, we will not utilize the Merkle trees for any security reasons, but to validate the contents of the data blocks. In the following example, the data blocks can represent some parts of a placement group. If two PGs have the same data, the two top hashes that will occur by building their respective Merkle trees will be the same. Otherwise, if a small change happens to one of the two PGs, the top hash should change dramatically. Once equipped with this hash verification mechanism, we can apply the same principle to the children of the tree. In the following binary tree in figure 10, if the top hash of two replica PGs is not the same, the search can continue to the children. Then, if one of the children’s hash is equal to the peer’s hashes it means that the data blocks on their leaves are also the same. Thus, the search will continue to the child node with the different hash, in order to identify the parts that have been altered between the two desynchronized PGs. With this divide and conquer procedure, we can identify the different parts in O(logN) execution time where N is the number of data blocks, or in our case the number of the PG’s hash ranges. This approach is better than the O(N) approach with the bitmaps that were discussed earlier in this section.

The Merkle tree approach is also better when two PGs have the same top hash, we can just send the hashes first instead of sending the whole bitmap. Therefore, we can divide the PG into N regions and hash the regions which contain objects. The regions that do not have any objects can have a default value of 0. Now we can distinguish which regions are used and we can figure out how sparse the tree can be for further optimizations. These Merkle trees will basically have the bitmap that was discussed earlier as their leaf nodes and build a tree structure on top of it to detect changes in O(logN) time, where N is the leaf size.

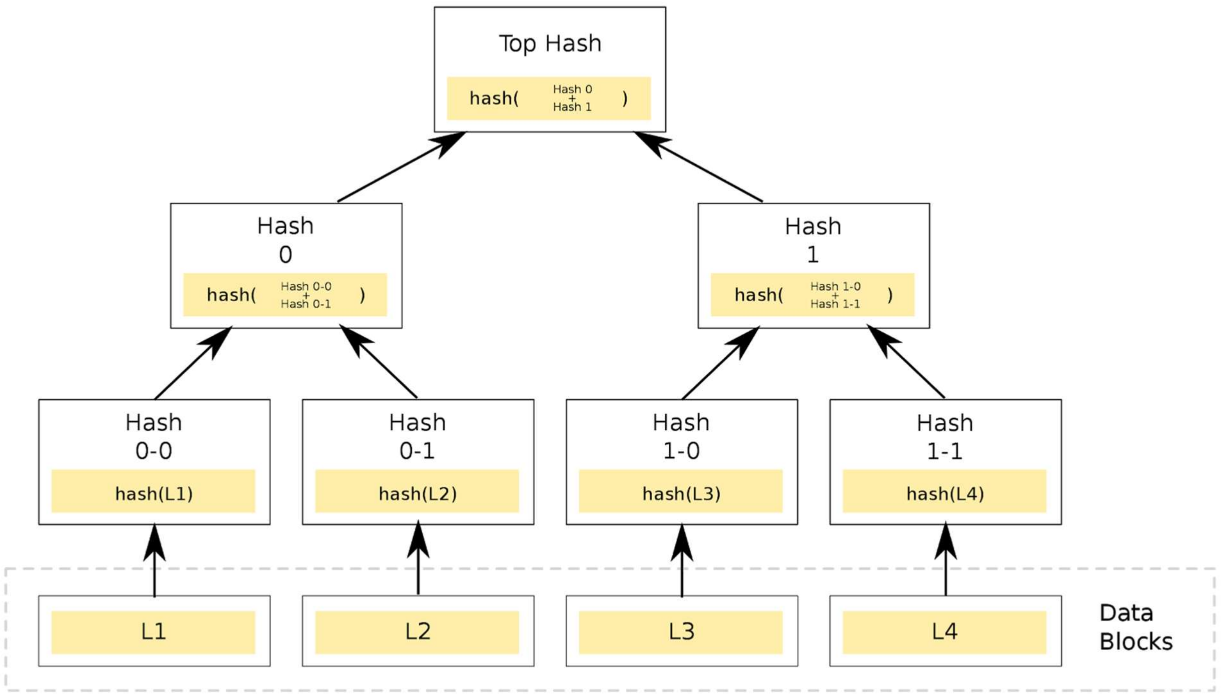

Figure 10: A full binary Merkle Tree. Each of the data blocks are hashed and stored to the respective leaf nodes. The non-leaf nodes combine the hashes of their two child nodes. Once the procedure is finished, the top hash is calculated.

When a PG starts up, it will either build the hash tree or load a previously persisted one. In the first case, it takes quite some time to build the Merkle tree as it requires to read the whole PG. This tree will be what the backfill process will use in our implementation to determine the differences between the replicas. Thus, if a Merkle tree is being built every time a backfill happens, the backfill time in the end will just increase by the amount of the PG scan, which is not optimal. Therefore, the building of the Merkle tree should be avoided at all costs, and it should be run only on the startup when the PG does not have any Merkle tree stored, or if the serialized copy of the Merkle tree on the disk gets corrupted for any reason. Hence, we implemented an update procedure on the trees to avoid the need of building the hash tree for every backfill procedure. First and foremost, there is no need for any kind of security in this case, because we are concerned only about the data validity, the use of cryptographic hashes is inefficient performance-wise. Therefore, we will use a fast non-cryptographic hash function, because the hash tree updates will be frequent. For example, Ceph has the non-cryptographic hash algorithm “xxHash” as an available option for checksums [30]. xxHash is advertised to be as fast as the memory copy operation and it passes the SMHasher[8] benchmark which checks the qualities of a hash function, such as collisions, randomness and dispersion. For this thesis, the XXH64 v3 variant will be used, which produces a hash value of 64 bits when given an arbitrary size binary input.

To be able to update the leaves of the hash tree frequently without the need of reading the respective data blocks every time, we will modify the structure of the Merkle tree and

[8] https://github.com/rurban/smhasher, written by Austin Appleby and tested by Reini Urban

introduce a 2-way function between the hash range and the Merkle tree's leaves. This 2-way function should work with 64-bits and it should not produce any number bigger than 64 bits, because there will be an integer overflow in which we will lose information that can't be reverted. Therefore, we can use the exclusive OR (XOR) to combine the hashes of each object to create a signature for the PG. The XOR operator has the following properties [37]:

- Identity element, $\mathrm{A\,XOR\,0 = A}$:
  Any XOR operation with the 0 element will yield the other operand as the result
- Commutative, $\mathrm{A\,XOR\,B = B\,XOR\,A}$:
  The order of the operands of the XOR operation does not matter
- Self-inverse, $A\,XOR\,A = 0$:
  A XOR operation with the same operands will always result to 0
- Associative, $A\,XOR\,(B\,XOR\,C) = (A\,XOR\,B)\,XOR\,C$:
  The XOR operations can be chained together because the order is insignificant.

With the above properties, we can create a formula so we can update the value of a hash range. The update formula will have the following form:

$$V^*_{hashrange} = V_{hashrange} \oplus V_{update} = (A_1 \oplus A_2 \oplus \dots \oplus A_n) \oplus (A_i \oplus A_i^*) =$$
$$(A_1 \oplus \dots \oplus A_{i-1} \oplus A_{i+1} \oplus \dots \oplus A_n) \oplus (A_i \oplus A_i) \oplus A_i^* =$$
$$(A_1 \oplus \dots \oplus A_{i-1} \oplus A_{i+1} \oplus \dots \oplus A_n) \oplus A_i^* \quad 1 \le i \le n \qquad \textbf{(3)}$$

Each A_i represents the object id/version pair hashed to 64 bits using xxHash of the i-th object in the hash range. As we scan all the objects of a PG to create the Merkle Tree, each object i will land on a leaf based on formula 2, and instead of turning the bit to 1, we XOR the leaf with the A_i of the object i and assign the new value. Once this procedure finishes, we have now $V_{hashrange}$ which represents an identity of all object within that range. For future writes that come after the tree is built, by using the associative and the commutative properties of the XOR function, we can cancel out the old id/version pair. Then, by using the self-inverse and the identity element we can conclude the last line of equation 3. By doing this XOR operation, we can guarantee that the old pairs will cancel each other and the new pair A_i* will take place. When we hash the id of the object and its version, we create a value that shows the state for a set of objects in the PG. As a non-cryptographic function is used and its seed remains the same, identical values will produce the same hash result. Because the xxHash has good hashing qualities based on some benchmarks, the XOR operator with each pair will be a good enough way to combine those values. We use the XOR operator because it is a working operator to merge hash values if those values are created from a good hash function, and it easy to revert from a value to another with minimal performance cost. By being able to revert to older states, we can now check whether a node undid some changes and has now the same top hash as the failed node, therefore we can skip the whole backfill process completely.

Even if there are better ways to combine two hashes that will lead to less possible collisions, those functions produce a result that can't be partially reverted and updated, since it will be necessary to combine the hashes of all the objects of that hash range again. Therefore, to combine these hashes again, we will need to read the metadata of the objects in the update hash range again, which is suboptimal. Hence, the use of the

XOR operator simplifies the hash combination of a PG region and it is a two-way function that can help with identifying changed that have been undone during the time frame of an OSD failure.

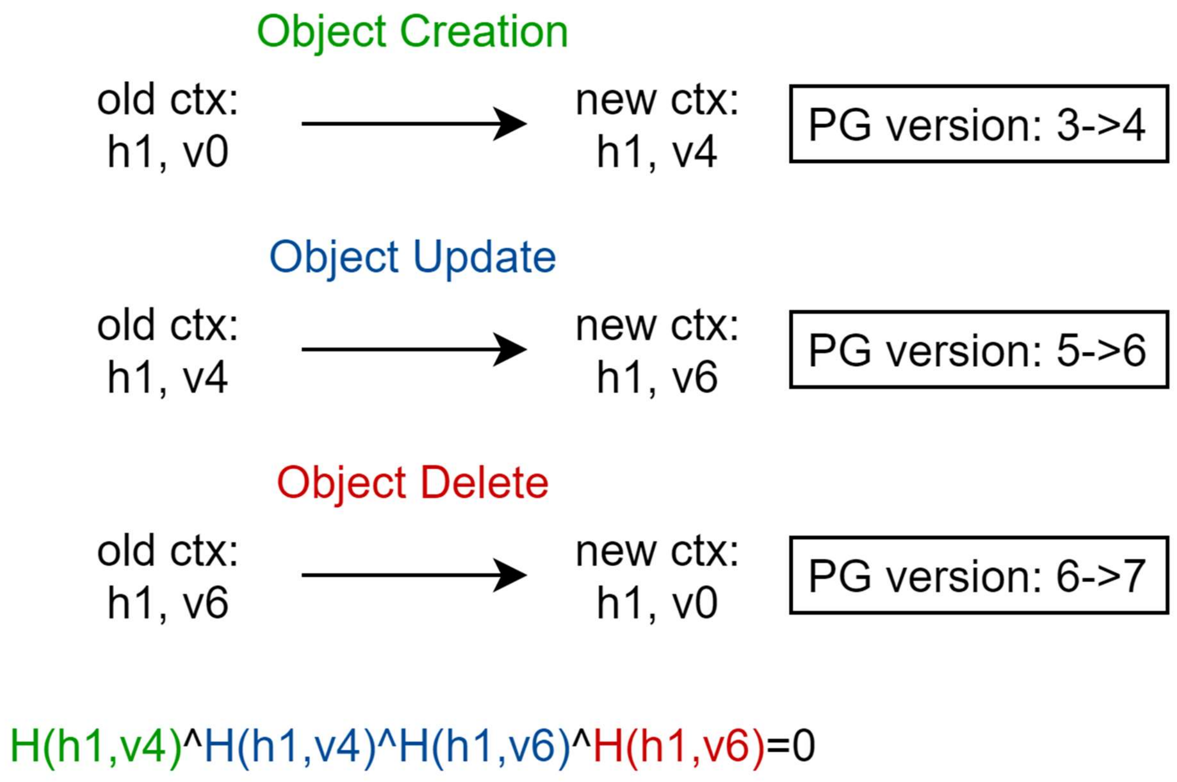

Figure 11: How an object's deletion cancels the Merkle tree's leaf data changes based on XOR operations on the object's versions. H refers to the xxHash64 function and "^" is the XOR operation.

Each I/O operation that is passed to the cluster is identified as a read or a write operation. Data or metadata write operations always alter the version of an object. To be able to keep the Merkle tree updated, we perform an update using equation 3 after the write has been committed the primary OSD. Then, we propagate the update to the replicas to perform the update. We don't need to do any checks if the write failed on a replica, this is an issue that the primary will resolve. Once the write has been done on the primary, the replicas will eventually get updated as well. Internally, the Ceph system on the write operations has two object contexts, one old and one new as seen on the above figure. With this metadata information, the next state of an object can be identified by those contexts. Upon an object creation, the old context is null, and the new context contains the next version of the PG. For example, when an object "i" is created the update value (V_{update}) will have 0 as A_i and the combines hash of the object and the version of the new context as A_i*. If the object already exists, there will be an old and a new context in which the A_i will contain hash(object.hash, old_context.version) and A* will be calculated the same way as A_i but with the version of the new context. A delete operation will have a null new object context and an old context as the result of the previous operation done to that object, so only the A_i or the old context will be applied, and it will cancel the existing A_i value from the $V_{hashrange}$ because of the way that XOR works. The update operation has a constant O(1) complexity, and it is equal to the height of the Merkle tree, because each parent of the leaf will have to get updated and the tree will never change in size. Finally, the update operation can be done on each object write because it is a quite fast operation, and it will keep the Merkle trees consistent at any point in time throughout the run of the Ceph cluster.

With the previous update mechanism, we can swap between states for objects without having to recalculate the hash range. On the above figure we can see that an object can be created, modified, deleted, and the leaf value will now return to the value that PG v3 had plus the update that changed the PG version from 4 to 5. If that update gets reverted as well, the update will get us back to PG version 3. Then for example, if the replica PG failed after version 3 and now it starts peering again, it will skip its backfill process because both object updates will be reverted and the value on all the replicas will be the same. Compared to Apache Cassandra's implementation, we now have a mechanism to keep the trees updated and we don't need to build them every time, which is an expensive task to do.

```cpp
void ReplicatedBackend::do_repop(OpRequestRef op)
{
...
if (m->new_temp_oid == hobject_t()) { //if object is not temporary
  eversion_t old_v = m->object_ctx_old_v;
  eversion_t new_v = m->object_ctx_new_v;

  uint64_t delta_hash = 0;
  if (new_v == eversion_t()) { //eversion_t() means version 0 or null
   delta_hash = hash_pair(m->poid, old_v);
  } else {
   delta_hash = hash_pair(m->poid, new_v) ^
    (old_v != eversion_t() ?
     hash_pair(m->poid, old_v) : 0);
  }

  get_parent()->update_object_info(m->poid, delta_hash);
 }
...
}

void MerkleTree::update_object(const hobject_t& hobj, const uint32_t& delta_hash)
{
 MerkleNode* node = root;

 int obj_pos = hobj.get_hash() >> reduce_bits; //hash range position

 std::vector<MerkleNode*> update_list;
 update_list.push_back(node);
 for (size_t bit = leaf_len >> 1; bit > 0; bit >>= 1) {
  node = obj_pos & bit ? node->right : node->left;
  update_list.push_back(node);
 }

 update_list[tree_depth]->value ^= delta_hash;
 for (auto it = ++update_list.rbegin(); it != update_list.rend(); ++it) {
  (*it)->value = join_hash((*it)->left->value, (*it)->right->value);
 }
}
```

Algorithm 1: The Merkle Tree update procedure on the replicas. Based on figure 11, we check if we have v0 on the old or new object context and we apply the appropriate update (create, update, delete). Then, we locate on the Merkle tree the leaf that has to be updated and we follow the path and update its parents up to the root.

The above algorithm describes the way we do updates on object writes. The issue_repop function is run by the replicas and the primary has a similar update procedure as well. We check if we have v0 on any of the new or old object context and we apply the correct update, whether is create, update or delete. We then call the Merkle tree function with the object hash and the delta to be applied, which is the V_{update} of equation 3. Because of equation two, we have now an easy way to identify which path we need to take to reach the object. We take the index of B in equation 2 and run its bits one by one, if the bit is 0, we traverse left, if the bit is 1, we go to the right child. Then at the end, we update the leaf and traverse the list from the end to the start by updating the path of nodes up to the root. A better implementation would be to have the Merkle tree with an array representation, since the tree doesn't change its size, and we can omit having to create a list of all the nodes to be updated from the leaf to the root. With an array representation, we have access to all the nodes without having to always traverse from the root, we can also compact the memory requirements because we don't need to include the pointers to the children anymore. Since we didn't see any noticeable I/O performance reduction on writing objects, we kept that implementation.

Now that we have defined the modified Merkle tree and its role in the Ceph cluster, we will have to alter the backfilling operation in such a way, so that the recovery algorithm's changes are minimal. From the start of the backfill run, up until the end there are some BackfillInterval classes which get filled from each PG and sent to the primary with their object metadata, so that the backfill operation can be done in batches. In our implementation, each batch will be tested against each range, if the batch object range intersects with a range to be skipped, then the unnecessary objects in that batch will be trimmed. Then, the next backfill batch will start after the skip range's end point. For the skip ranges to be formed, we will implement a new coordination algorithm withing the backfill procedure to gather all the Merkle trees to the primary OSD. The operation will be the similar as the one that fetches the object metadata for backfill, but it will exchange the trees instead. Because of the structure of the Merkle Tree, the primary can request at first the head or root node of each PG's Merkle tree. If the top hashes match, the primary will not request the full tree as both the primary and the replica are already consistent, and the backfill procedure for that replica will be cancelled. If the top hashes don't match, the primary will get the full tree from the replica, in order to identify which ranges need to be backfilled and which need to be skipped if there are any. Algorithm 1 describes the way that the trees are being compared with each other. The algorithm which produces the skip ranges by comparing the two Merkle trees is shown below

Merging those skip ranges would avoid unnecessary multiple checks and we have less memory footprint. The leaves of the tree are stored in an array for faster access compared to traversing the path from the tree's root, and from there we can derive their index to calculate the hash range they represent. To save a bit on the network load, the Merkle tree sharing procedure can be done in a streamlined fashion. Instead of the primary PG requesting the top node of each replica PG, it could send its top node to its neighbors, then the neighbors would either respond with an ok message if the top hashes match and if they don't, they will send a message to request the primary's tree. Once they have the primary's tree, they can detect the differences and send the primary the filtered objects to be checked. With this new way of metadata exchange, we can save some unnecessary messages and make the backfill implementation more efficient on the network side. Algorithm 2 describes the way the trees are exchanged and how we finish the exchange in less steps.

```cpp
void MerkleTree::compare(MerkleTree& other, HashRangeIndex& range_idx, pg_shard_t from)
{
  range_idx.clear();
  if(this->root->value == other.root->value) {
    return;
  }

  std::deque<std::pair<uint32_t, uint32_t> > skip_ranges;
  std::deque<std::pair<uint32_t, uint32_t> > range_queue;
  std::deque<MerkleNode*> lst1;
  std::deque<MerkleNode*> lst2;
  lst1.push_back(a1->left);
  lst1.push_back(a1->right);
  lst2.push_back(b1->left);
  lst2.push_back(b1->right);
  range_queue.emplace_back(0, (uint32_t)-1);

  bool isLeftChild = false;
  while (!lst1.empty()) {
    isLeftChild = !isLeftChild;
    auto curr1 = lst1.front(); lst1.pop_front();
    auto curr2 = lst2.front(); lst2.pop_front();

    auto top = range_queue.front();
    uint32_t middle = (top.first >> 1) + (top.second >> 1);
    if (curr1->value == curr2->value) {
      skip_ranges.emplace_back(
        isLeftChild ? top.first : middle + 1,
        isLeftChild ? middle : top.second
      );
    } else if (curr1->left) {
      lst1.push_back(curr1->left);
      lst1.push_back(curr1->right);
      lst2.push_back(curr2->left);
      lst2.push_back(curr2->right);

      range_queue.emplace_back(
        isLeftChild ? top.first : middle + 1,
        isLeftChild ? middle : top.second
      );
    }

    if (!isLeftChild) {
      range_queue.pop_front();
    }
  }

  if (skip_ranges.size() > 1) {
    auto it = skip_ranges.begin()+1;
    while (it != skip_ranges.end()) {
      auto prev = it-1;

      if(prev->second == it->first-1) {
        it->first = prev->first;
        skip_ranges.erase(prev);
      } else {
        ++it;
      }
    }
  }

  range_idx.fill(skip_ranges, from);
}
```

Algorithm 2: Compare a Merkle tree with another one in the object-oriented form of Tree1.compare(Tree2) in C++. Once the algorithm is finished it will produce a hash range index for the hash ranged to be skipped. The variable “pg_shard_t from” refers to the replica id.

The above algorithm compares a tree to another and fills a range index to query the ranges to be skipped in logarithmic time. This function is run both by the primary PG and the replicas so they can detect their differences. The replicas only compare their tree with the primary and not with the other replicas. In the first 5 lines, we compare the trees top hash to check whether they are the same. If the hashes match, we can omit the whole comparison process and avoid backfill for that replica. From lines 7 to 45 we create three lists, one for each Merkle Tree and one to store their matches (list1,list2,range_queue). The queues will be populated by the trees leaves in a breadth-first fashion and the range_queue will have the [0, UINT_32_MAX] as the initial range to skip. We then check whether the nodes in the lists contain the same hash value and if they do, we append the hash range that the node is accountable for into skip_ranges. Then, if the nodes are different, we create a subrange based on the first range of the range_queue and if the child is the left or the right one. If the child is the left one, we create a subrange of the left half of the top element of the range_queue, and if the child is the right one, we create the right half respectively. For example, if we have the range to skip which includes the current hash range [0,5] we would append the ranges [0,2] and [3,5] to the end of the queue if they have a different hash each and remove the [0,5] once we are done with it. When the loop finishes, we will be left with the queues empty and the skip_ranges to include the ranges to skip for every equal node. Since there is a high probability that we will have consecutive ranges to be skipped, we merge those ranges in lines 47 to 59. In the end, we populate the range index with the ranges we want to skip because they contain the same hash values.

```
void PrimaryLogPG::do_object_info(OpRequestRef op)
{
  const MOSDPGObjectInfo *m = static_cast<const MOSDPGObjectInfo*>(op->get_req());
  ceph_assert(m->get_type() == MSG_OSD_PG_OBJECT_INFO);
  dout(10) << "do_object_info " << *m << dendl;

  bool is_reply = false;
  op->mark_started();
  switch(m->op) {
    case MOSDPGObjectInfo::OP_GET_DIFF:
      uint64_t from_top_hash;
      auto p = m->get_data().cbegin();
      decode(from_top_hash, p);
      MOSDPGObjectInfo *reply;
      if (from_top_hash == backfill_tree.get_root()->value) {
        reply = new MOSDPGObjectInfo(MOSDPGObjectInfo::OP_HEAD_MATCH,
          pg_whoami, spg_t(info.pgid.pgid, get_primary().shard),
          get_osdmap_epoch(), m->query_epoch);
      } else {
        reply = new MOSDPGObjectInfo(MOSDPGObjectInfo::OP_FULL,
          pg_whoami, spg_t(info.pgid.pgid, get_primary().shard),
          get_osdmap_epoch(), m->query_epoch);
        backfill_tree.encode(reply->get_data());
      }
      osd->send_message_osd_cluster(reply, m->get_connection());
    break;

    case MOSDPGObjectInfo::OP_FULL:
      is_reply = is_primary();

      MerkleTree bt_tree;
      auto p = m->get_data().cbegin();
      bt_tree.decode(p);
      backfill_tree.compare(bt_tree, backfill_ranges_to_skip, m->from);
      if (is_primary()) {
        MOSDPGObjectInfo *reply = new MOSDPGObjectInfo(MOSDPGObjectInfo::OP_FULL,
          pg_whoami, spg_t(info.pgid.pgid, m->from.shard),
          get_osdmap_epoch(), m->query_epoch);
        backfill_tree.encode(reply->get_data());
        osd->send_message_osd_cluster(reply, m->get_connection());
      }
    break;

    case MOSDPGObjectInfo::OP_HEAD_MATCH:
      is_reply = true;
      backfill_ranges_to_skip.ignore(m->from);
    break;
  }

  if(is_reply) {
    if (waiting_on_backfill.erase(m->from)) {
      if (waiting_on_backfill.empty()) {
        ceph_assert(
                peer_backfill_info.size() ==
                get_backfill_targets().size());
        finish_recovery_op(hobject_t::get_max());
      }
    } else {
      dout(20) << __func__ << " canceled object info request" << dendl;
    }
  }
}
```

Algorithm 3: The tree exchange algorithm. This is run by both the primary and the replica shards. We have three types of messages, OP_GET_DIFF for requesting a tree, OP_HEAD_MATCH if the top hashes match, OP_FULL for exchanging the full trees.

The above algorithm describes the exchange of the trees at the start of backfill. We create a fake recovery procedure to pause backfill and exchange the trees. In a production scenario as mentioned before, we will need to implement the above algorithm in a set of states for the PG. The MOSDPGObjectInfo is a class that describes a message that is handled by the OSD which forwards it to a PG which implements a message type of ObjectInfo. We use the term ObjectInfo to describe the Merkle trees in a generic way. We will handle three cases of this type of message. The first case is the OP_GET_DIFF, in which the primary at the start of the backfill operation sends to the replicas. To skip a step, we encode the primary's top hash in that message so we can get an early reply from the replicas. The replicas will now handle this case and the top hash sent with theirs, if the hashes match, we will send an OP_HEAD_MATCH message to the primary to indicate that we don't need backfill on this shard/replica. In the latter case, we will send an OP_FULL message to the primary shard and we will encode our tree. Then the primary will receive the tree, compare the changes (Algorithm 1) and send its tree back to the replicas for themselves to compare as well. Now that the tree exchanges are finished, we remove the shards from the waiting list and we resume the backfill by finishing the fake recovery procedure.

When the backfill operation starts, it has a hash counter that goes from 0 to 32-bit integer max, this value increases to the hash of each object scanned. This counter is used by the primary to keep track of the backfill progress. If for any reason the backfill is postponed, it will resume from that point. Given the previous information, any operations done to objects behind that counter will continue normally, since the objects that have a hash lower than that counter have already been checked and they can't interfere with the current process. However, the I/O operations on objects that have a hash that is greater than this backfill counter will only be persisted to the primary PG. Then, as the backfill goes on, the counter will reach those objects and update them. Even if this approach provides a write lock-free PG, there can be updates that are within the ranges that they were calculated to be skipped. Thus, to be able to include those objects in our implementation, we will need to update the skip ranges on the fly as object operations happen.

To keep the implementation optimal, we will have to update the ranges in logarithmic time at least. We can’t afford to have an O(N) operation, where N is the number of hash ranges to be skipped. Therefore, to improve the search of the skip ranges, we will store them in a one-dimensional R-tree which will be our hash range index. The one-dimensional R-Tree will store the merged hash ranges, and this optimization saves a lot of space compared to having an array of ranges for each replica. Those arrays will have the form of the bloom filter as shown in equation 2, where 1 or 0 will correspond if that range should be skipped on not. The resulting “n” merged ranges are quite smaller and the O(logn) search on the R-tree will not impact the performance of the write I/O. Then, on each object write that is only persisted on the primary and it is not propagated to the replicas, we will query the hash of the object on the R-tree and receive a range in logarithmic time. We can also get 0 ranges, which means that this object is going to be checked anyways. The range received after a search on the R-Tree can be either deleted or split with the updated object’s hash excluded. On one hand, by using the first approach, we would probably slow down a lot the backfill process as the object update would end up in a large hash range that will get deleted, and therefore we will have to check all those objects unnecessarily because of one object update. On the other hand, many writes would create a lot of ranges, because they will exclude only certain objects and the adjacent skip ranges will have a tiny gap between them. Therefore, this approach defeats the purpose of having an R-Tree, because the ranges would start to increase a lot, as they will take more space than the bloom filter approach for the ranges discussed before. A better approach would be to split the skip ranges by removing a hash range partition instead. For example, if we have merged three continuous ranges together and the object updated is in the middle, we remove the whole range and we add the left and right parts that are left. Hence, if the skip range is not a merged result, it will be deleted, or if the skip range is a result of adjacent merged ranges, it will reduce in size by the length of a single hash range instead of being fully removed. Finally, the new updates may affect ranges which got deleted previously, thus the search on the R-Tree the second time and onwards will return zero results and no action will be taken.

The Merkle trees provide us all we need in order to identify unnecessary I/O performed by the current implementation of backfill. Also, their memory requirements are constant compared to an in-memory WAL log which constantly grows while writes happen [25]. As discussed before, the Merkle tree has a lot of advantages over a bitmap implementation which may resemble a variation of bloom filters [32]. Therefore, the Merkle trees can perform various tasks within the backfill context in logarithmic time compared to a bloom filter implementation.

Finally, after we have defined the exchange of the trees, how they get populated and updated, their comparison and an index with ranges to skip, we have now a way to filter the objects that won’t have to be checked. The backfill operation constantly requests from the primary and the replicas a batch of objects to be compared together to find any missing or stale objects. Therefore, all we need to do now is to alter this object listing function in the backend, Bluestore, and run the object cursor given by RocksDB against the hash range index. If the object is in a range to be skipped, its metadata won’t be added to the BakcfillInterval structure to be checked against the other objects, the object will be ignored. Thus, we have now a way to avoid scanning all the objects and we can do a filtered backfill which is faster. On the next section, we will discuss various aspects of our implementation against the basic implementation which is in the master branch of the upstream Ceph source code.

3.6 Similar work

This thesis was heavily inspired by the Anti-Entropy mechanism of the distributed wide-column store called “Apache Cassandra” [33]. In Cassandra they build Merkle Trees to check for differences in the replica tables of the database. The process goes in two stages, first they build the Merkle trees based on each row, even though the store is column based. As said before, this process happens every time the anti-entropy algorithm starts, in which they claim that the tree building operation is heavy and it should not be done frequently. In the current thesis we have established a mechanism for updating those trees in constant time without the need of rebuilding, except in cases of data corruption. Even if the updating procedure requires updating all the parents recursively of the respective leaf, the size of the tree is remaining constant. Thus, the number of cases in which the building of the trees is required is reduced significantly. The size of their Merkle trees is constant with a depth of 15 (2^{15} leaves), they use small size trees so they don’t have to use a lot of memory and they can be also transferred quickly between all replicas. On the second phase, there is an initiating node that receives all the trees and does the comparison. Finally, when all the differences are detected, the initiating node identifies the conflicting ranges and replaces them with the newest data. In the next section, we will check the recovery performance of different sizes of trees and we will compare to check whether the size used by Apache Cassandra is good enough for our use case.

Other object store systems apart from Ceph, such as Lustre, require metadata servers in order to identify and resolve inconsistencies between the replicas. Systems such as Farsite [39], Glacier [40] and Oceanstore [41] provide a distributed failure recovery that focuses on data safety, by using erasure codes or even encryption at the cost of performance. Filesystems such as Gluster [34] and HDFS [35] perform a block-based recovery. Each data block has a timestamp of the last checkpoint of the cluster. If that timestamp is old enough and the block is in a recovery state, which that state is maybe caused by either a replica node that got restarted after a failure in HDFS, or it may be caused by the self-heal daemon of Gluster which found some inconsistent metadata between the replicas. Both of those systems lock the underlying data block, cancel any writes on-fly for the clients and perform a full recovery. In our case, all the writes to the PGs are still allowed without impacting heavily the performance of the clients. A previous version of the recovery mechanism in Ceph was synchronous[9] and it locked the PG like HDFS and Gluster, though newer versions of the recovery mechanism in Ceph are more versatile and they do not impact the cluster’s I/O performance significantly.

In each store case for recovery, Ceph has introduced an improved way to do recovery on the cluster with the least performance impact. Now with our implementation, we can have a better recovery mechanism than what Apache Cassandra provides.

[9] https://docs.ceph.com/docs/master/dev/osd_internals/async_recovery/

4. EXPERIMENTS AND RESULTS

To be able to run the backfill operation in our test cases, we will have to set the minimum and maximum size of the PG WAL to 1 and 2 respectively. Once three writes happen while an OSD is down, the log will be discarded and the backfill process will immediately begin. Therefore, we can guarantee that the backfill process will always happen, so we will be able to compare the differences in various metrics. The metrics that will be measured during the experiment will be CPU load, memory used, network load, and finally the backfill duration. The source code developed that implements the backfill optimization is a fork of the Ceph master branch. At the time of writing, the Ceph version worked on was named Octopus. The two versions, our implementation versus the current code, will be tested using three parameters. The parameters will be the Merkle tree height, the number of objects per PG and the percent of the degraded data. For example, we will test a Merkle tree of height 10 (1024 partitions) with an object store of 10 thousand objects per PG. We will alter 1/5/10/20% of the data and we will start taking samples of the metrics to produce the results. The test environment will be a machine of 32 cores and 64 GB of RAM. We will create 3 OSDs and a pool with 2-64 PGs. On each test case we will drop down an OSD, alter some amount of data and start the OSD again once those writes finish. The replication factor will be three and the minimum size of replicas will be 2. We will not test the case in which we drop 2 or more OSDs, because we will go below the minimum number of replicas available in some PGs and we will lose consistency, because only 1 out of the 3 replicas will be up.

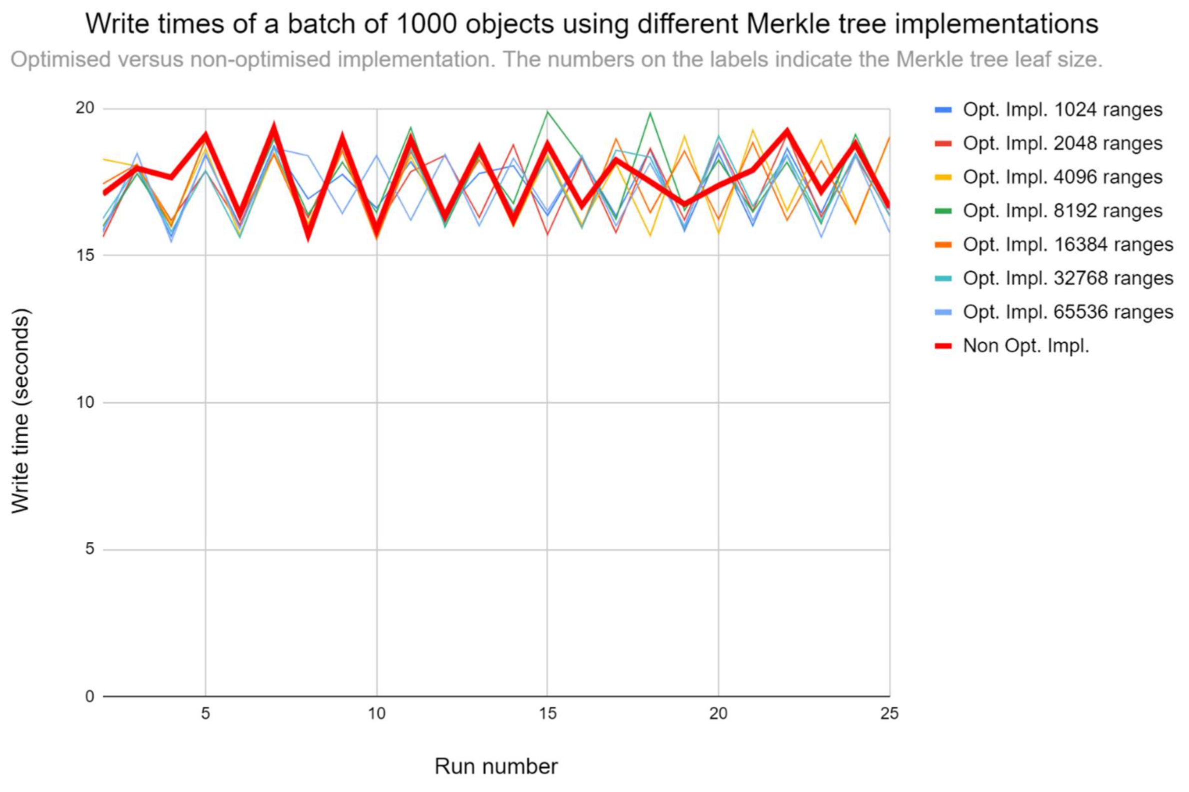

Figure 12: Time in seconds that takes for a run that writes 1000 objects in parallel. The numbers on the labels indicate the amount of ranges that the Merkle tree supports. A taller tree will require more updates between the root and the leaf on each object write.

Before the main backfill experiment, we will have to get some statistics about how our implementation impacts the performance of CEPH. We update the tree on the fly, after every write and we would like to know how that affects the write throughput of the test cluster. On a cluster with a monitor node, a manager node and three OSD nodes we tested the performance of small (1KiB) object writes. The test consists of 20 runs of a thousand object writes on 32 threads. The machine consists of a 32 core AMD Opteron 6276 processor with 2.3 GHz clock speed, 64 GB memory and 3 2TB Western digital HDDs with a rotation rate of 7200 RPM in Raid 1 configuration. The results are shown on the above figure. Both implementations ran with the same parameters except the basic one, which lacks the Merkle tree implementation. The builds are done with CMake using the option -D RelwithDebugInfo, which produces a release target that supports full debug info (coredump) in case of any crashes. All three configurations are run with full log level reports and therefore the Merkle Tree builds produce longer OSD logs than a real production Ceph cluster. We can notice that there is almost no difference in the performance of these runs. Each run was measured using the “time” command and the times were sampled from the “real” field. One explanation of the similarities on the write performance is that almost all the operations of the PG are done asynchronously, which include updating the PG log, forwarding the write to the Bluestore queue and sending a commit message to the primary PG. The primary will update its tree while it waits for the replicas to return a commit message and the replicas always update the Merkle tree while they write the object. Therefore, the Merkle tree update is overshadowed by the procedures that happen in each write between OSDs.

As seen from the above figure, there is no clear winning implementation, which is considered a positive result. Thus, the constant time of the Merkle tree update does not impact the write performance of the cluster. Also, those tests should be done on faster storage mediums, such as NVME drives, which may show some performance drawback due to the shorter times between the different tasks done by the primary OSD and the replicas. In addition, Table 1 shows the average time for the system to write 1 thousand new objects to the RADOS subsystem directly in detail. For the tests will use Rados and its basic get/put API to read and store objects. We could have simulated a more realistic workload, by using CephFS, but we would have to enable the full logs to parse the file to object mappings to be able to remove the right files that would affect a certain PG. Nevertheless, on the current tests we can see that there is little to no clear answer whether the update of the Merkle for each write impacts noticeably the I/O performance of the Ceph cluster.

Table 1: Average times and standard deviation of the 25 runs of figure 12

Implementation	Average time (seconds)	Standard Deviation
Base	17.665	1.137
1024 ranges	17.353	1.134
2048 ranges	17.347	1.261
4096 ranges	17.476	1.368

8196 ranges	17.59	1.339
16384 ranges	17.449	1.24
32768 ranges	17.35	1.272
65536 ranges	17.25	1.257

The below figure explains in general but not in exact detail, the series of function calls that happen in a certain order for an object to be written to the cluster. Also, the primary OSD will execute the object context which contains the operation that will be processed, and it could be an append, write or even a delete operation. Then, the primary PG will forward the state update of the object that affects its version to the replica PGs through “issue_repop”. The replicated PGs will then receive the message and process the operation forwarded to them. Once the object propagation to the replicas is done, each of those replicas will return a write commit message to the primary placement group. The object update which changes the Merkle tree’s leave values and then updates all the parent nodes will be executed just after the message sent. Therefore, we can hide the update between the message latencies from the primary PG and the replicas in both ways.

Even if the performance of the cluster is not impacted in an observable way, the implementation of the Merkle trees requires some additional memory usage. The tree’s size will remain constant throughout the cluster’s run. We can then measure the memory impact directly without running any kind of benchmark. Unlike the PG log, which grows as writes happen, the tree will remain constant in the sense that only its nodes’ values will be changed over time. The below figure 14 shows the additional memory required from the current base implementation of Ceph to support the Merkle trees for each placement group. While the number of PGs per OSD increase, so does the memory. Each PG has its own Merkle tree for their objects. The number 64 is a good enough number of PGs per OSD, when 30 and 100 are the minimum and maximum respectively. Even if the memory overhead seems small for the 65536 leaves tree implementation, which is shown in figure 14, the OSD hosts usually have a set of 20 to even 100 disks. Therefore, this overhead can be translated to roughly 3 GBs to 15GBs of total memory overhead for an OSD host, which now seems a lot.

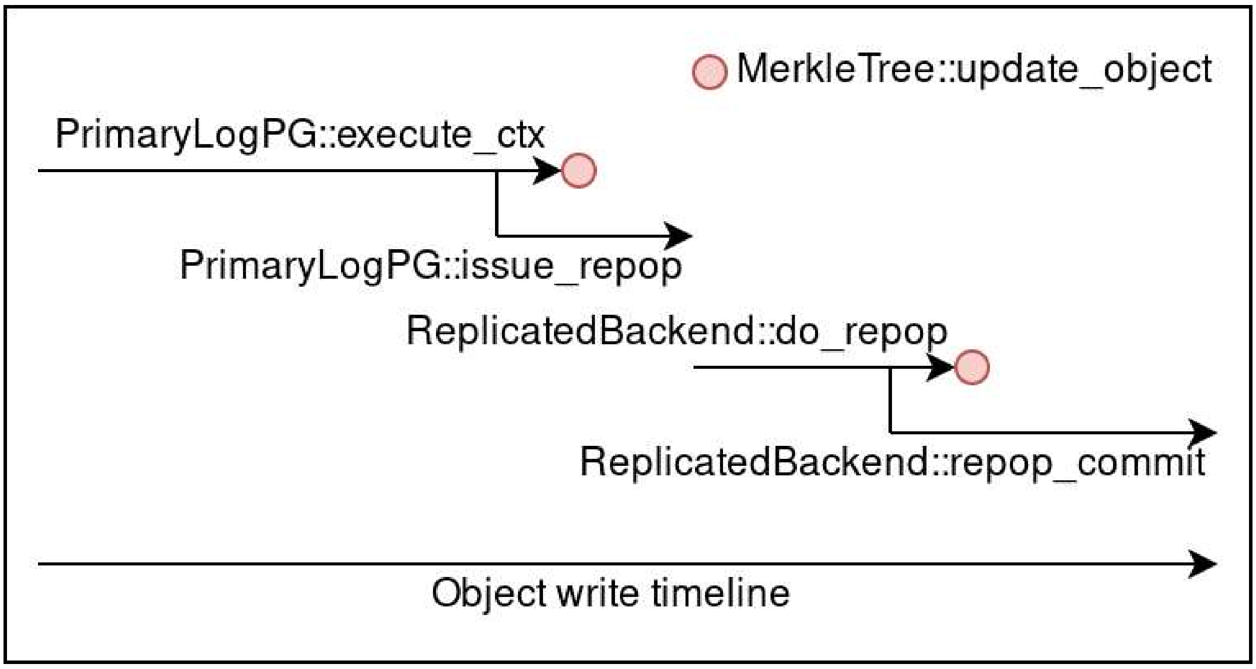
MerkleTree::update_object
PrimaryLogPG::execute_ctx
PrimaryLogPG::issue_repop
ReplicatedBackend::do_repop
ReplicatedBackend::repop_commit
Object write timeline

Figure 13: The timeline in which an object is written to the disk. The primary executes the object write and forwards the write to the replicas with PrimaryLogPG::issue_repop. The replicas execute the object write and send a commit back to the primary OSD.

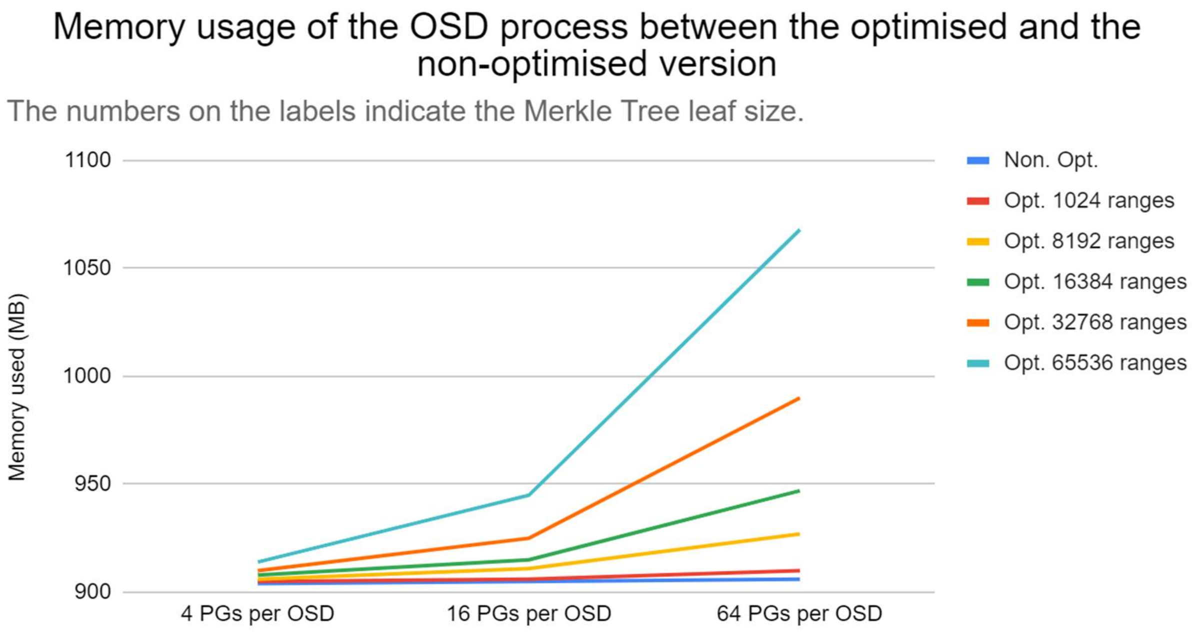

Figure 14: The additional memory used per each ceph-osd process. The base line is at the bottom and represents the memory usage on the original non-optimised implementation. We didn't fill the OSD with any objects, we just count the memory footprint after the first run of the ceph-osd executable to get how much memory the Merkle trees cost.

As the number of ranges increase and the number of PGs increase, the memory consumption of the Merkle trees may become an issue. We will run a benchmark to see how many ranges are utilized per given number of ranges for several objects per PG.

Each benchmark will write 50K objects in pools of 4, 8, 16, 32 and 64 PGs. Then we will sample the non-zero leaves per the number of objects written to the PG. As the PG gets more objects and their hashing is good enough, we should expect an almost linear increase of the ranges used per placement group up to a point. This point is where there is a minimal amount of unused ranges left and their chance to be selected against the used ones is a lot lower. This behavior is common to the coupons collector's problem [44]. Let's say we have a collector of cereal box coupons and the cereal box company has released m different coupons. How many cereal boxes does the collector have to buy in order to get all the distinct coupons? In our case, each object has at best an equal chance to be somewhere inside the full 32bit range (0 to 2^{32}-1). Each new object will map to a random range, which is the same as mapping a new cereal box to a coupon in the described problem. This way we can predict the utilization of the trees based on the number of objects each PG has. Provided that the hashing function will draw a range with an equal chance, it will require on average for a PG to have E(m) objects in order to fill m ranges, where E(m) is equal to:

$$E(m) = m\left(\frac{1}{m} + \frac{1}{m-1} + \cdots + \frac{1}{2} + \frac{1}{1}\right) = mH_m, m \geq 1$$

where H_m is the m-th harmonic number. The following formula has the following approximation using asymptotics:

$$E(m) = mH_m = m\,ln(m) + \gamma m + \frac{1}{2} + O\left(\frac{1}{m}\right)$$

The symbol "γ" above denotes the Euler–Mascheroni constant. In order to find how many ranges are filled per n objects, we will have to solve for the inverse function. We will first ignore the asymptotic big O by setting it to 0, then we will alter m to be able to compute the inverse function.

$$m = e^{ln(m)}, O\left(\frac{1}{m}\right), E(m) = n, n = ln(m)e^{ln(m)} + \gamma e^{ln(m)} + 0.5 \leftrightarrow$$

$$n - 0.5 = (ln(m) + \gamma)e^{ln(m)} \leftrightarrow$$

$$e^{\gamma}(n - 0.5) = e^{\gamma}(ln(m) + \gamma)e^{ln(m)}$$

$$W_0\big(e^{\gamma}(n - 0.5)\big) = W_0\left((ln(m) + \gamma)e^{ln(m)+\gamma}\right) \leftrightarrow$$

$$W_0\big(e^{\gamma}(n - 0.5)\big) = ln(m) + \gamma \leftrightarrow$$

$$e^{W_0\left(e^{y}(n-0.5)\right)} = e^{ln(m)+\gamma} \leftrightarrow$$

$$\frac{e^{\gamma}(n - 0.5)}{W_0\big(e^{\gamma}(n - 0.5)\big)} = me^{\gamma} \leftrightarrow$$

$$E^{-1}(n) = \frac{n - 0.5}{W_0\big(e^{\gamma}(n - 0.5)\big)} \quad \mathbf{(5)}$$

In the first step we set m to be equal to e to the power of a function f. If we apply the natural logarithm on both parts of the equation, we can get that $E^{-1}(m)$ is equal to ln(m).

In the first step we replace m with $e^{f(m)}$ and group the right part of the equation. Then we will multiply both parts with e^{γ} to bring the right part to the desirable form. The right part is the inverse of the Lambert W function [43], where this function is the inverse of $g(x)=xe^{x}$. Later, we apply the function on both sides and on the right part we get the form of $g^{-1}(g(x))=x$, where x in our case is f(m)+γ. Then we make both sides as the powers of e and solve for m. Because the Lambert W function provides many solutions, we keep the one with the real values (W_0). Therefore, the equation (5) shows us the size of the Merkle Tree that will be fully used by n different objects written in a PG. Thus, we can use the closest powers of 2 to be sure that our trees are well utilized. For example, if a PG has 250K objects, E^{-1}(250K) will give us about 23K ranges. if we split the full hash range into 16K ranges, we would be sure that our ranges are fully used and they contain on average about 16 objects. If we choose to use 32K ranges instead we would have about 8 objects if all those ranges are used, but because a subset of the 32K ranges will be actually used, the average number of objects that we would have to read for backfilling per write/update would increase, which is not optimal. Hence, using 16K ranges, one write operation on a degraded cluster would cost at most 16 additional object reads, instead of the whole 250K objects of the PG. Nevertheless, in a real case scenario there would be much more writes done during a degraded period. Thus, equation 5 provides us a heuristic to determine the number of ranges and therefore the size of the Merkle Tree, so it will be used optimally and we will not have any redundant memory footprint. The graph below shows the percent of ranges used per n objects.

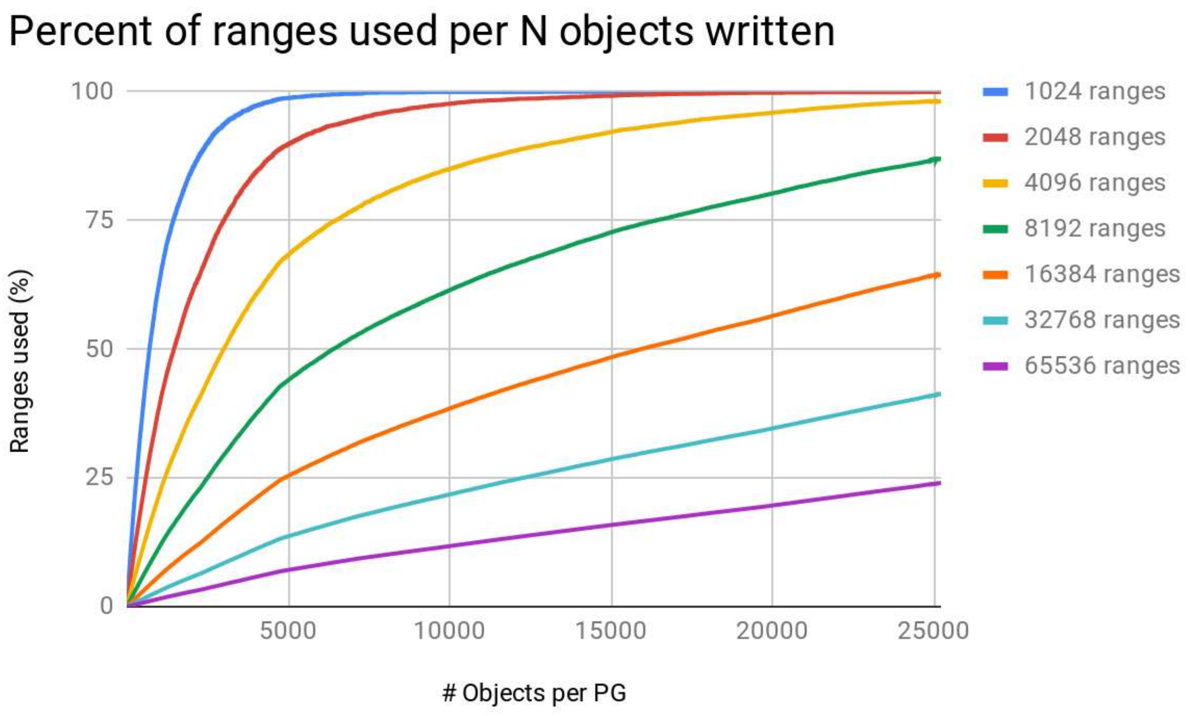

Figure 15: The average number of ranges used per PG per N objects written, where 0<N<25K. Each line represents the percentage of ranges used as we put more new objects in the PG. After some object insertions an object will be put in a used range, thus the used range won't increase, and we see a curve after a certain point.

The above graph confirms that the curves follow equation's (4) pattern. To detect the amount of non-zero leaves per object, we can try to predict what happens when an object o_i with i as the number of the write operation on the PG. We will also set as n_i the number of non-zero leaf values when we write o_i. A new object o_{i+1} will either do nothing if the leaf is a non-zero value, so we will have $n_{i+1}=n_i$ with probability n_i/N where N is the number of the trees' leaves, or $n_{i+1}=n_i+1$ with probability $1-n_i/N$. Therefore, for every n_i on average we will have:

$$E_N[n_0] = 0, E_N[n_{i+1}] = E_N[n_i]\left(1-\frac{1}{N}\right)+1$$

$$E_N[n_i] = \sum_{j=0}^{i-1}\left(1-\frac{1}{N}\right)^j = \frac{1-\left(1-\frac{1}{N}\right)^i}{1-\left(1-\frac{1}{N}\right)} = N\left(1-\left(1-\frac{1}{N}\right)^i\right) \quad \mathbf{(6)}$$

From figure 15 we can see that the line with the 16k ranges passes closely to 25% on 5k objects. Using equation (6) for i=5000 and N=16384 we get 4309 ranges over the ~4096 that we get from the above figure. This result is expected, because in equation 6 we assume a perfect distribution when the hashing functions used are not actually perfect. We can use the above equation 6 as the upper bound of ranges used per number of unique objects in a PG for our expectations.

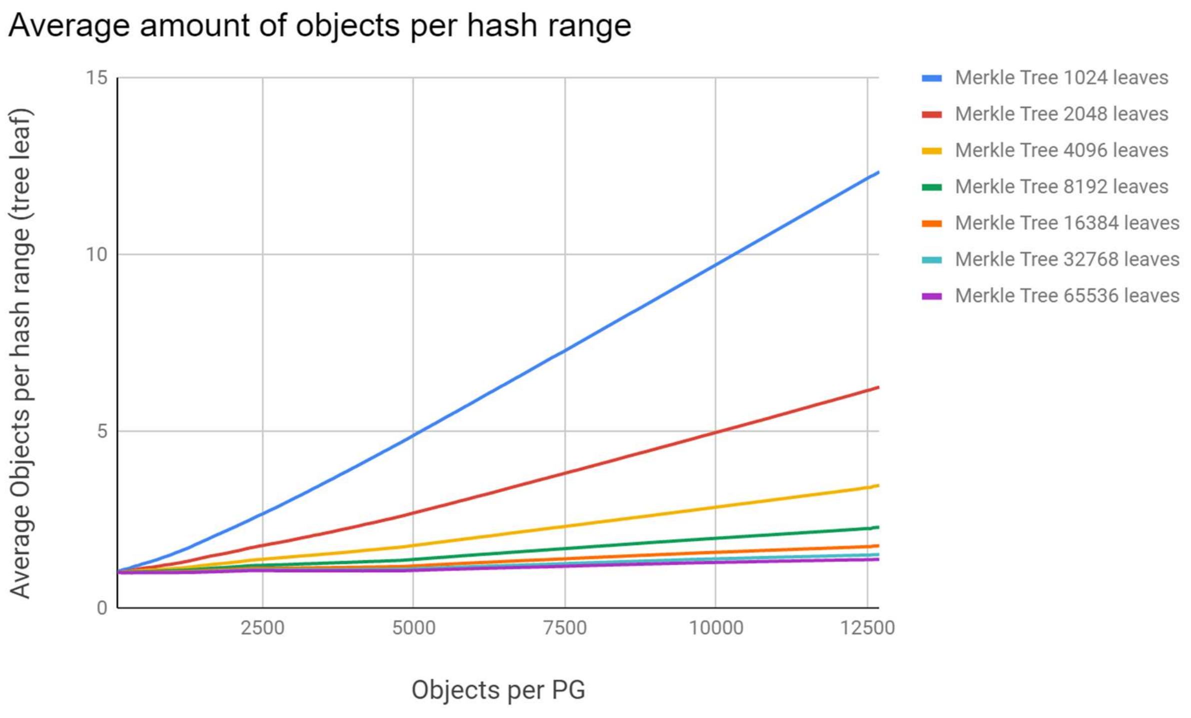

Figure 16: The average number of objects per hash range based on the number of objects per PG. The number of objects increases linearly after all hash ranges are filled with objects.

The above graph describes the number of objects per hash range in various configurations. Based on the graph's results, the PG number does not play any significant role in the way objects are distributed. The lines in figure 16 have a curve until all ranges are used and then the lines follow a straight path. The linear finish of that curve is because after a certain amount of object all hash ranges are used (figure 15), therefore each object from now on contributes the same to the average number of objects per hash range. Each data point is the average of the number of objects per PG divided by the number of non-zero ranges for that PG. Hence, we can see the maximum recovery penalty per write for different Merkle trees and a different number of objects.

The next benchmark indicates the time it takes to recover one thousand objects in different setups. Every benchmark has been done in a cluster with two PGs in replication times three and three OSDs. Therefore, we can see the differences in performance in a single placement group in clear. It should be noted that any pool can't function with a single PG and the in-memory recovery log (WAL) has been set to have min_size of 1 and max_size of 2 for the entries, so it has close to zero effect on the backfill time. The procedure goes as follows:

1. Several objects are being written into the cluster
2. A log file prints in which placement group the objects were written
3. A percentage of the objects is being sampled to be updated
4. The PG with the closer number of objects from the parameter given is selected
5. The OSD that has the previous PG as a replica (non-primary) is marked down
6. The sampled objects get updated while the replica OSD is down
7. Once the update finishes the OSD is restarted
8. The debug log is being monitored until the backfill finishes
9. The debug log is being saved into a different location for further analysis

From the debug log we can get the exact times of when the backfill starts and finishes and the amount of data sent/received for each OSD. Thus, we can setup a local cluster without the need of external monitoring of the network traffic. For the first example, we are going to show the time needed to recover 1 thousand objects in different setups. The setups include a PG with 5K objects and 20% of the data degraded (1K objects degraded), 10K and 10%, 20K and 5% and 100K and 1% respectively.

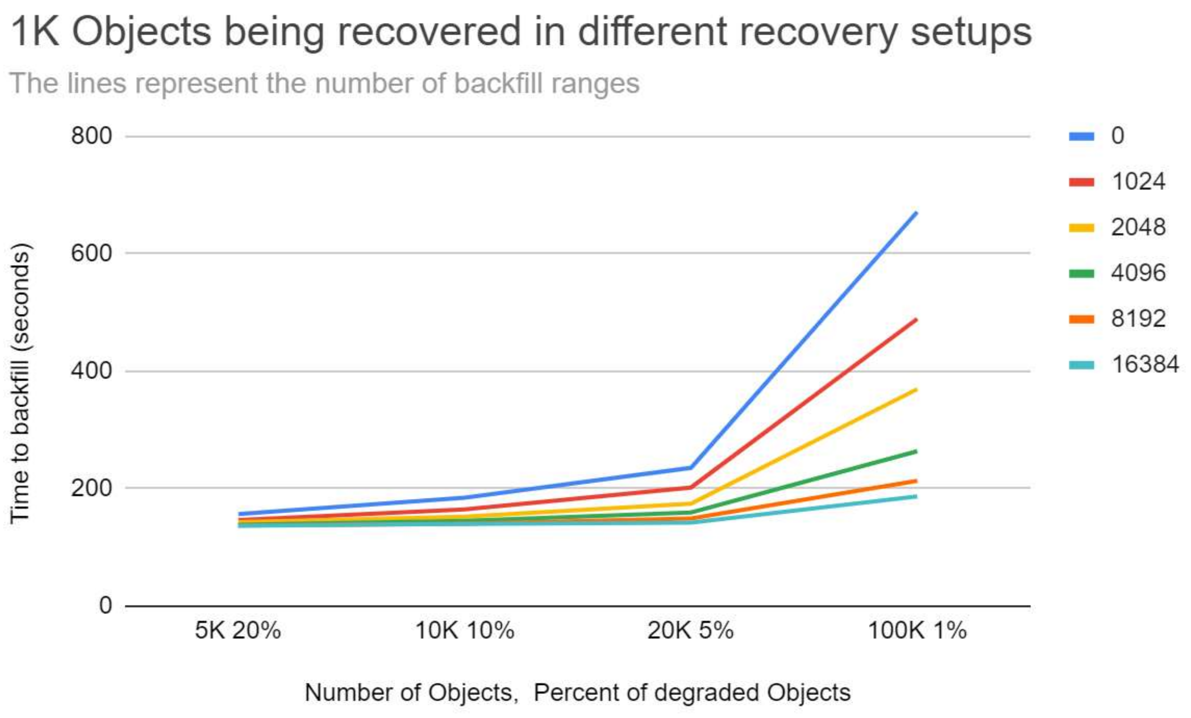

Figure 17: The time it takes to recover 1k objects in different recovery scenarios. The lines indicate the number of leaves of the Merkle Tree which represent the number of ranges that split the 32bit value range. As the number of objects increases the time to recover increases as well, along with the performance differences between the number of hash ranges used.

From the above graph we can see that the amount to recover 1k objects on the test VM is about 150 seconds. As the number of objects becomes greater, the number of objects that go into the ranges that will be backfilled increases. This effect results in more possible degraded objects that have to be checked and thus the performance worsens. On the above graph, the line with the label 0 shows the performance of a non-optimised backfill. We can observe that the time to recover the objects themselves takes about 150 to 200 seconds, and the time to check every object in the 100K takes more than double the time to recover the objects. The VM has disks with about 80 random write IOPS and writing to the cluster takes ~16 seconds of the optimal 12.5 which is about 60 write IOPS. The recovery time however shows that about 6 to 7 objects per second are being recovered, which can be confirmed from the live status message of the cluster during the backfilling benchmark. Ceph is making sure about the integrity of the data, valuing fault tolerance and consistency over performance. In addition, the build produced are debug builds will full log reports, so we can locate and sample the objects to be updated. The below graph describes the time it takes to recover 1 thousand objects in a 100 thousand objects PG, and it compares the time it takes based on equation 6, which is considered optimal. We removed the time it takes to recover 1K objects from the real line and the original backfill implementation, because it is a constant duration and we care more about the discovery time.

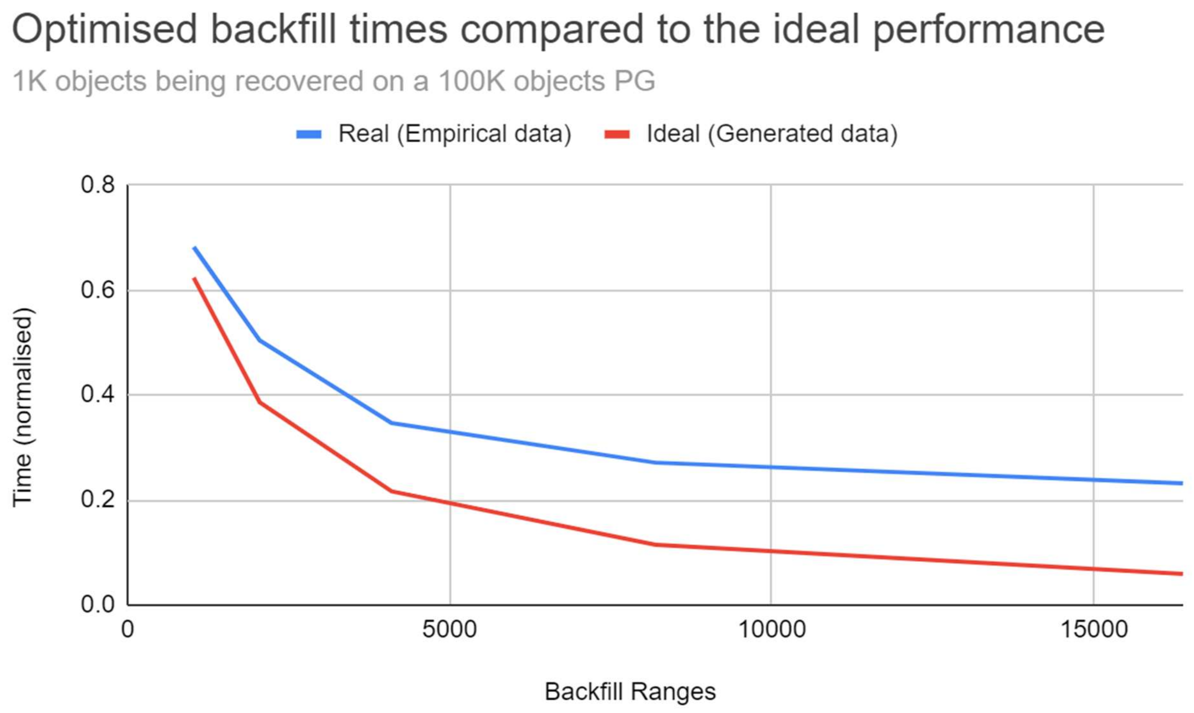

Figure 18: The time it takes for a thousand objects to backfill in a 100 thousand objects PG. The real data points are taken from the "100K 1%" of the above graph. The ideal points are taken from equation 6, using N as the number of ranges and i as 1000 and dividing the result by N.

Based on the calculations done previously about the likelihood of an object to fall on a range and how many of those ranges contain objects, we can observe that the real curve has a similar form to the ideal one. Many factors play a role to the backfill performance, therefore the real performance is slower than the ideal one. Those factors include for example the network load, the disk's random write/read performance and various other tasks performed by the cluster during the backfill. The below four graphs show that when the number of ranges are increased, the ranges are thinner and the objects that need to be backfilled are less.

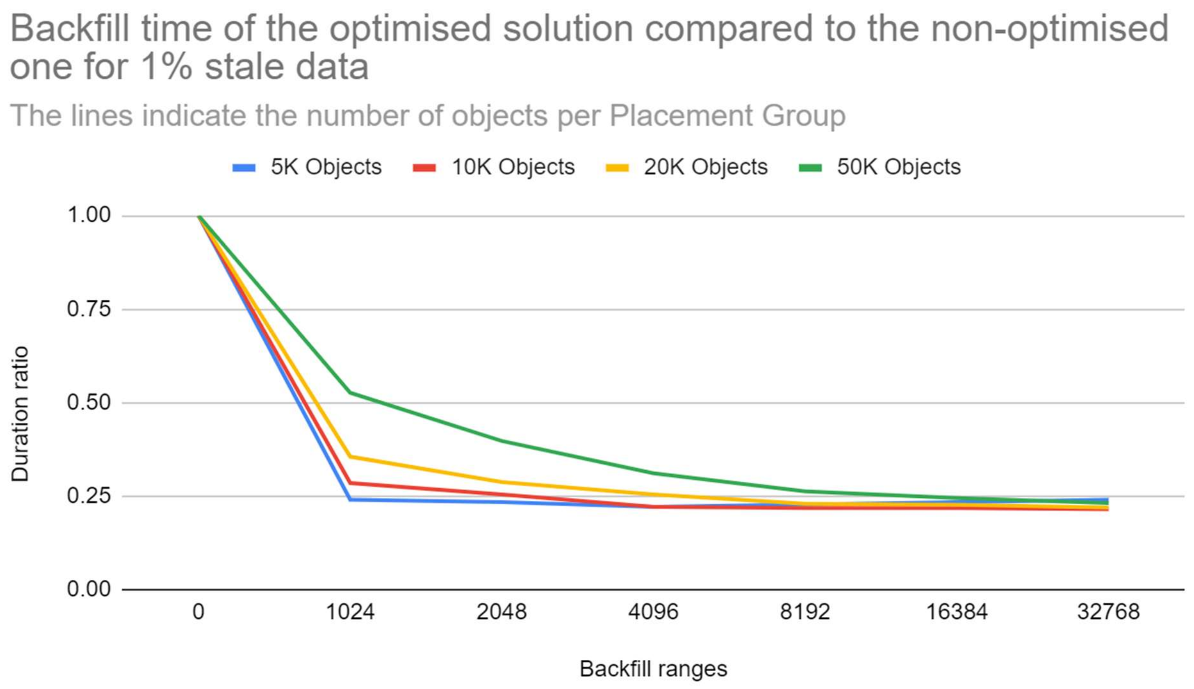

Figure 19: Backfill times normalized after changing the number of backfill ranges for a number of objects per PG and 1% of that data being stale. For example, the green line shows how much time it takes to backfill 1% of 50 thousand objects in a PG. As the number of ranges increase, the detection of the degraded objects becomes better. Therefore, less false objects are checked and the duration of backfill is 25% of the non-optimised implementation.

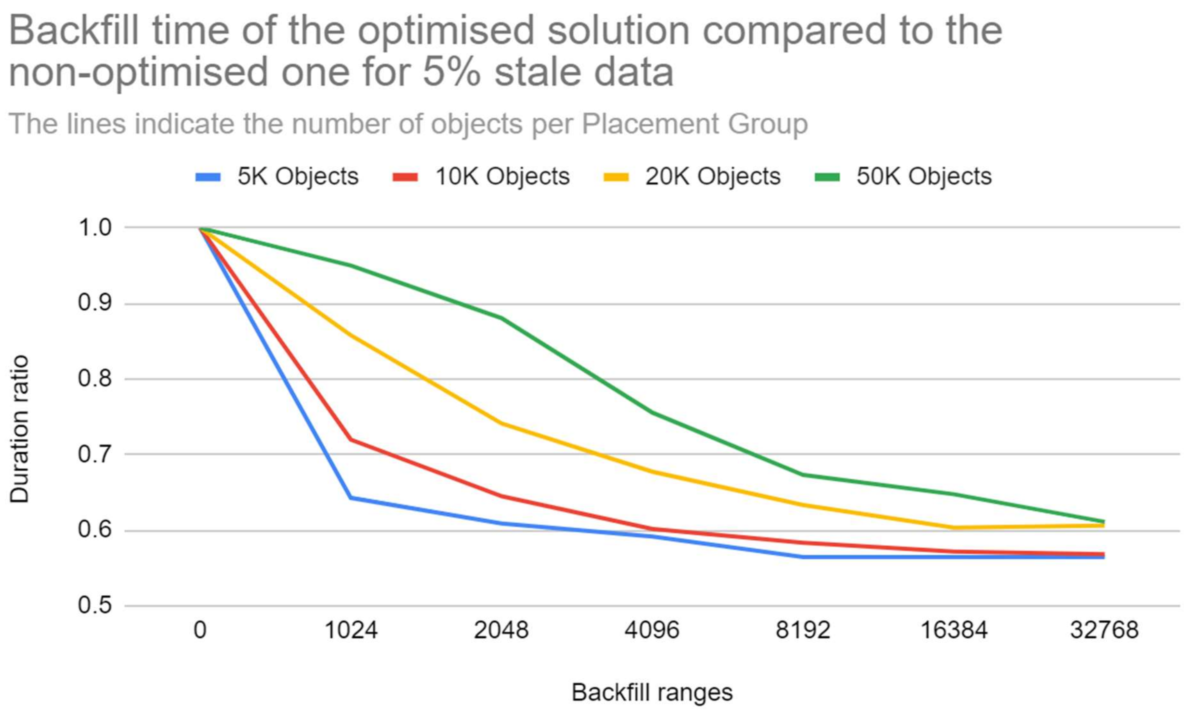

Figure 20: Backfill times normalized after changing the number of backfill ranges for a different number of objects per PG in thousands. The number of degraded objects has increased 5 times from the previous graph, the best that be achieved now is a bit less than half the time.

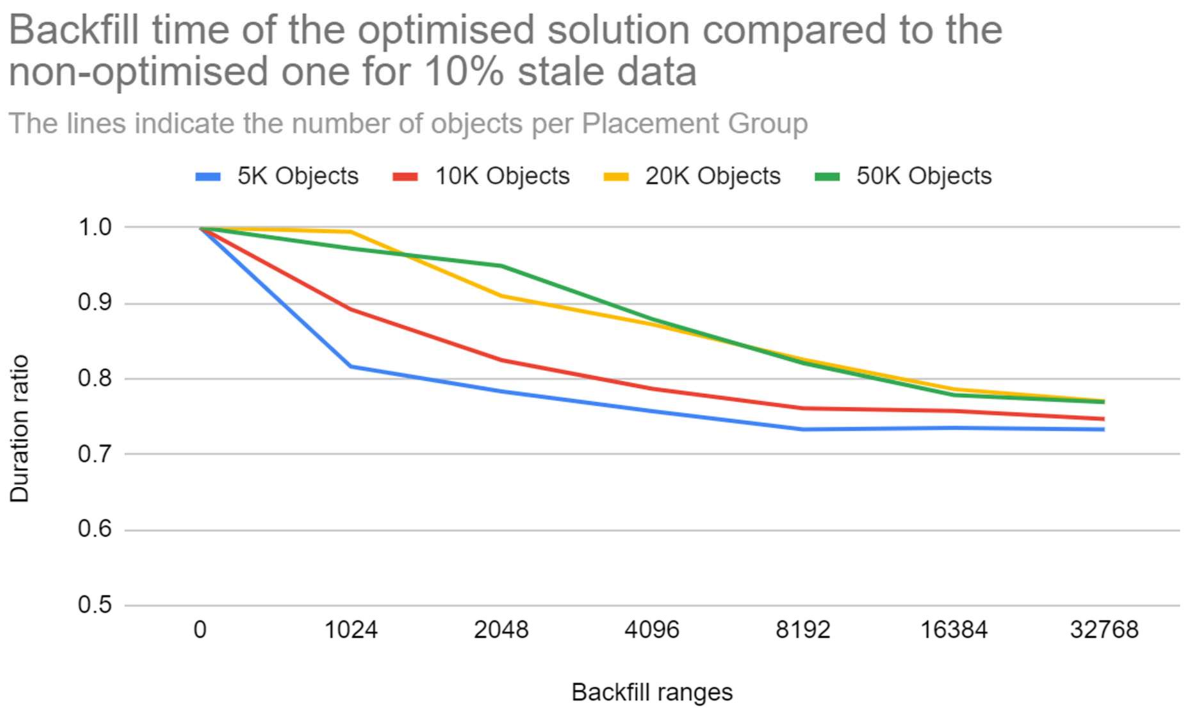

Figure 21: Backfill times normalized after changing the number of backfill ranges for a different number of objects per PG in thousands. By changing 10% of the PG data while one is down, the backfill times become at best ~35% faster as the number of ranges increase

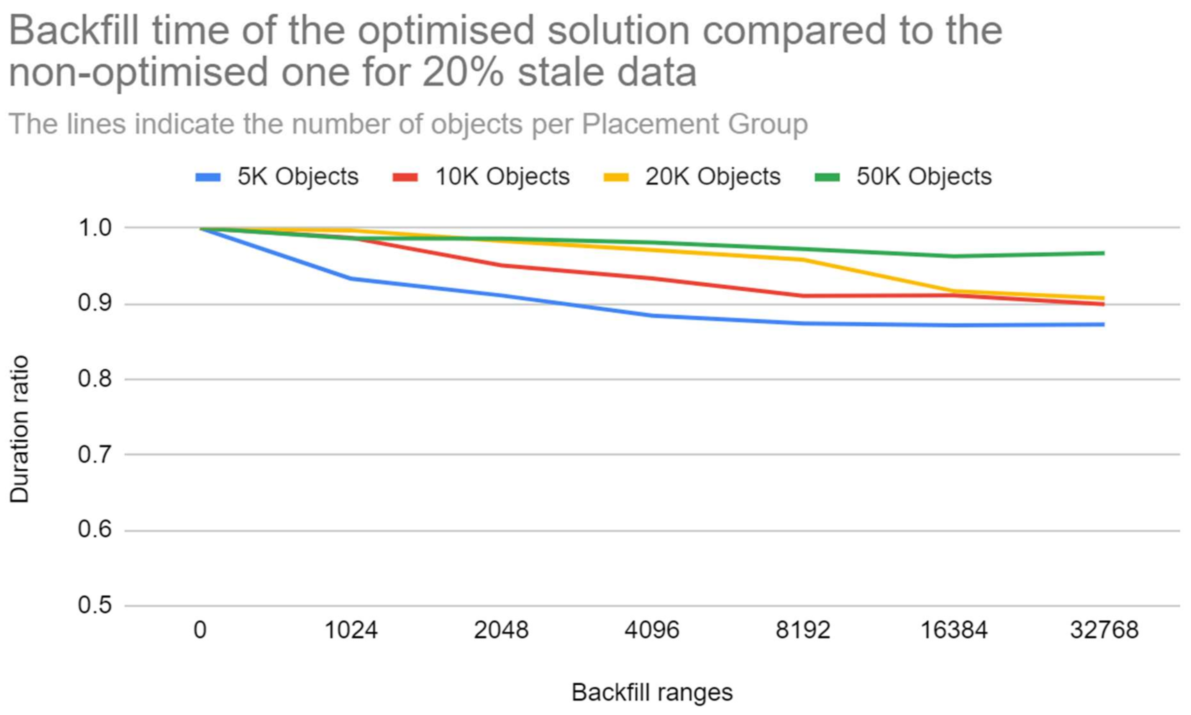

Figure 22: Backfill times normalized after changing the number of backfill ranges for a different number of objects per PG in thousands. After making 20% of the PG data stale, the backfill times become from 5% to 15% better. The 20% of the data has an even spread throughout the ranges due to hashing, therefore many objects are being checked.

On most of the graphs it seems that the 32k ranges seem to perform a bit worse than the 16k ranges. One reason that can cause this is that after 16K ranges on these benchmarks each range contains about the same number of objects, and when the backfill iterates the objects, the search function that identifies whether an object is within a range performs worse. Each object is queried on an R-tree to check whether it is within the affected ranges. Therefore, when there are more ranges to be checked, the search function performs worse because the R-tree height grows larger. If there can be a way to be able to get the objects of the PG with an order (e.g. ascending), this object filtering can perform better. Nevertheless, the ability to iterate the objects in ascending hash order does not invalidate the use of R-trees. The R-trees are still needed for the ranges, so they can be updated in logarithmic time while the backfill happens, as with the case of an object update during the recovery. The procedure to check if an object is in the hash range to skip in the case of ordered iteration, it would be to cache the first range, check each object if it belongs to that range, proceed accordingly and if an object has a hash value greater than the right side of the cached range, get the next range and check it against the new objects. Even without this additional optimization, the Merkle tree and the object filtering accomplish an observable difference in the backfilling performance.

The below graphs show the performance differences between the master branch implementation compared to the optimization on a steady set of objects and backfill ranges and an increasing number of degraded objects.

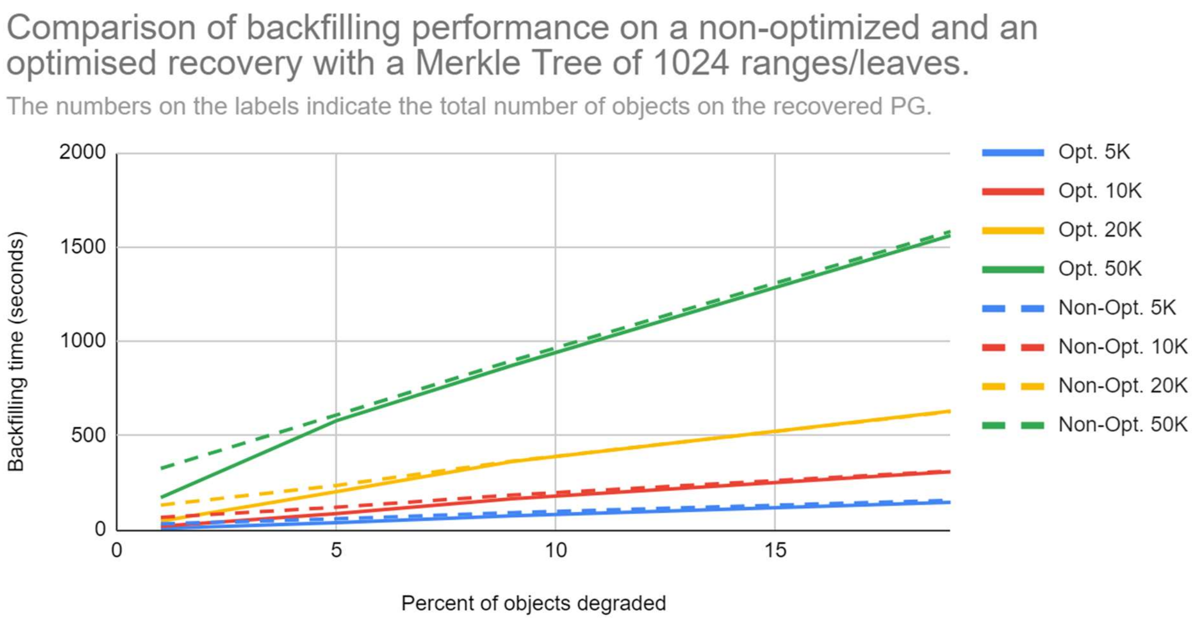

Figure 23: Backfill performance compared to the non-optimised version on a merkle tree with 1024 leaves. The lines represent the number of objects in a PG in thousands. The color coding and the line shapes try to show the differences between equivalent experiments on the optimised and the non-optimised implementation.

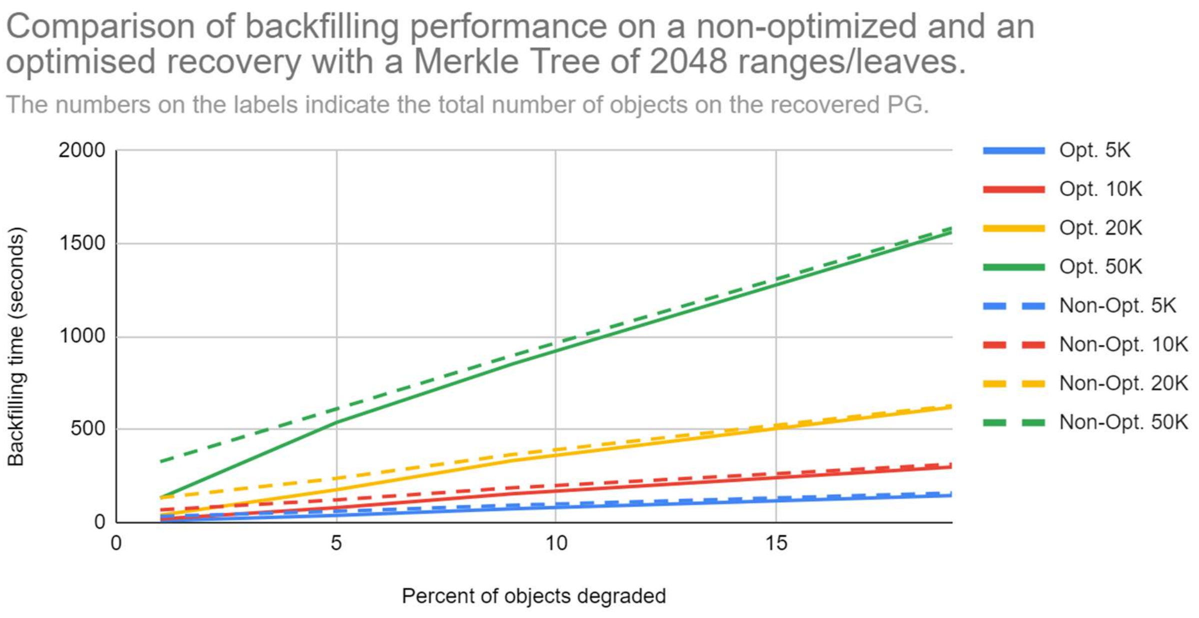

Figure 24: Backfill performance compared to the non-optimised version on a Merkle tree with 2048 leaves. The dotted lines represent the basic implementation and the straight line our improvement. Compared to the previous graph the performance difference starts to show a bit more.

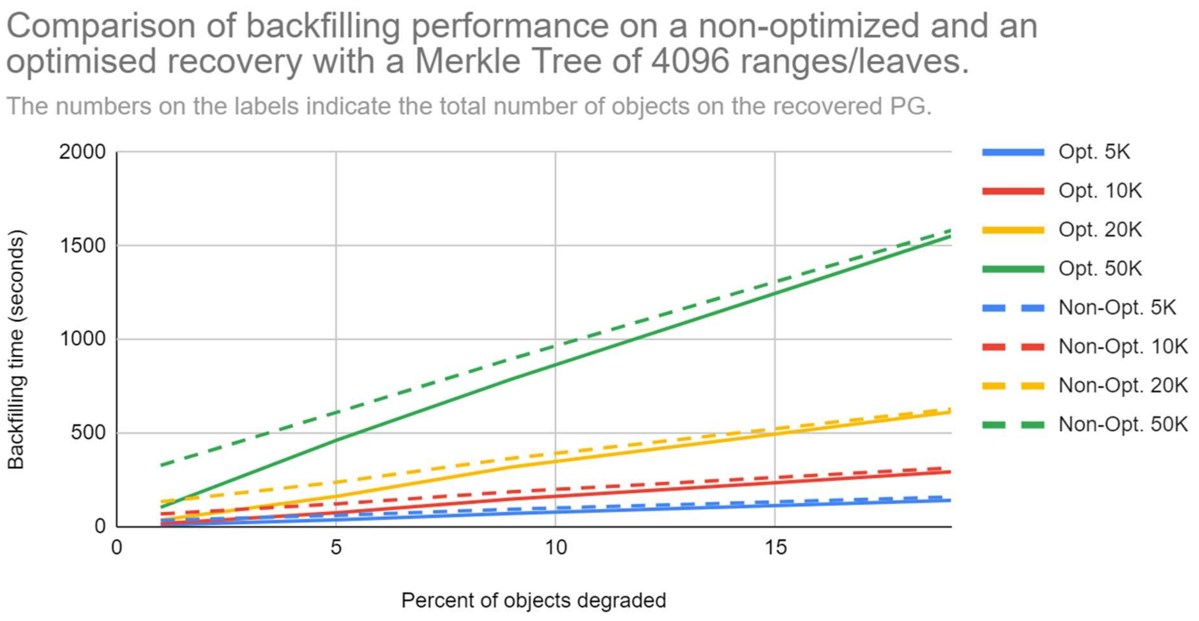

Figure 25: Backfill performance compared to the non-optimised version on a Merkle tree with 4096 leaves. The full lines of the optimised version to the dash's lines of the non-optimised one start to show that the performance gains are greater, when the number of degraded objects is lower.

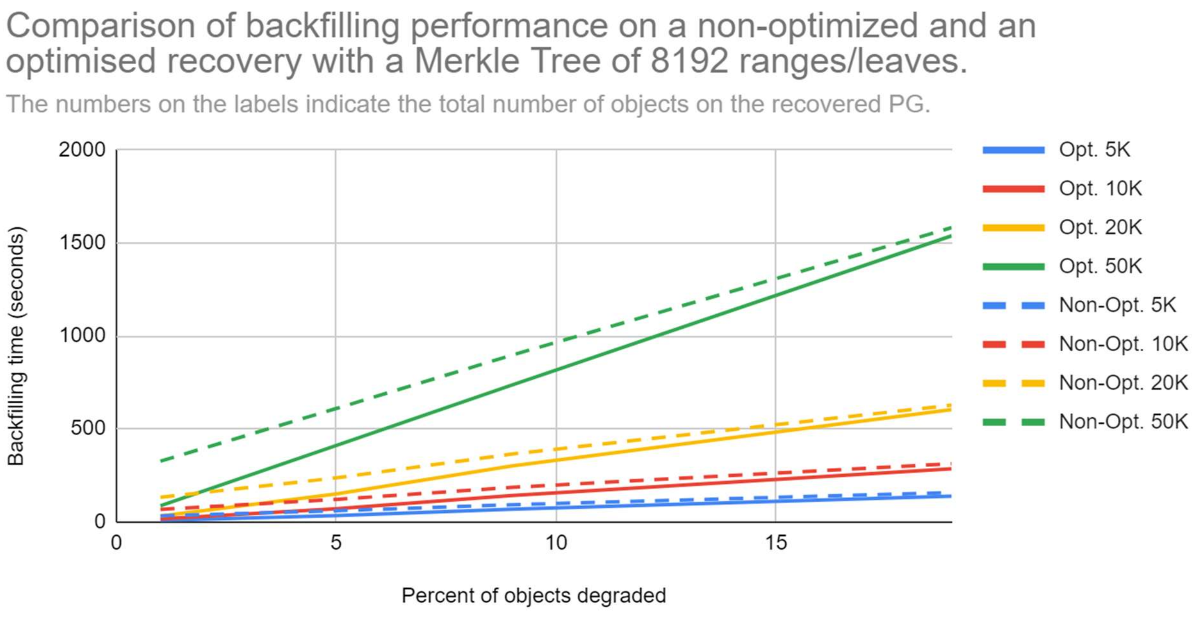

Figure 26: Backfill performance compared to the non-optimised version on a Merkle tree with 8196 leaves. The non-optimised version shows a linear increase that can be validated from the data.

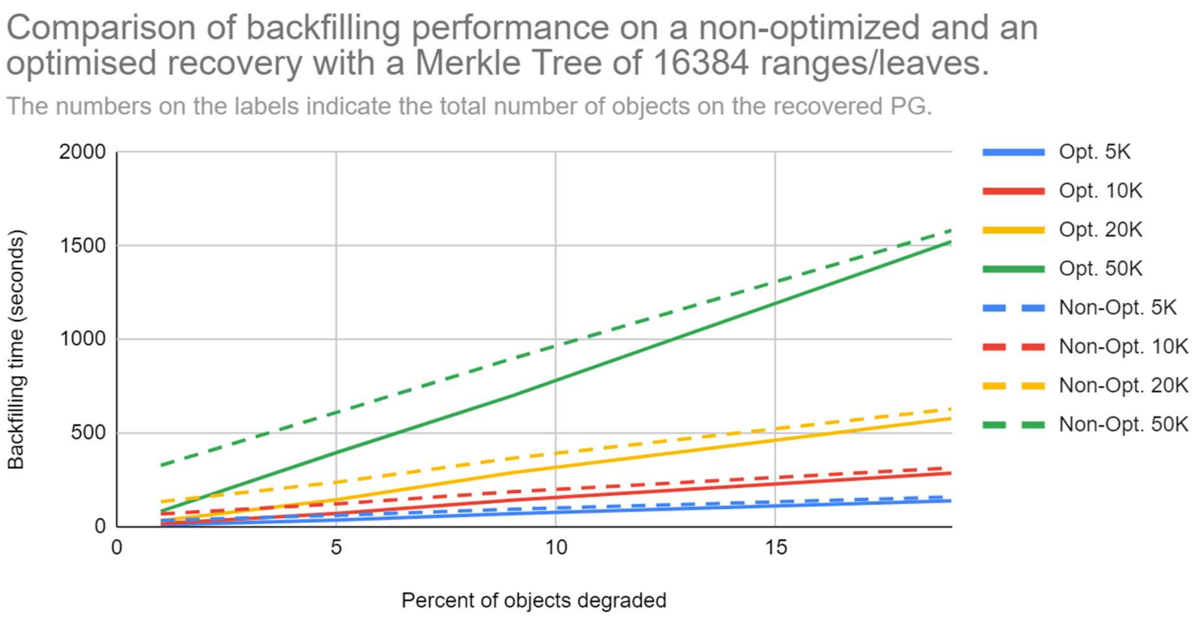

Figure 27: Backfill performance compared to the non-optimised version on a Merkle tree of 16384 ranges. It has the best performance on all the benchmarks, but it comes with a memory cost.

Based on the above graphs we can observe that the backfilling time for the base version is linear to the number of degraded objects. The time to scan the objects is the same, and thus the basic backfill scales with the amount of time to recover the objects. The table 2 below shows the real and estimated linear times of the non-optimised backfill. We see that the Merkle tree implementation follows a lightly curved line that seems to be asymptotic compared to the basic backfill. In the case that every range is marked as degraded, the backfill time will be the same or even a little bit slower than the basic backfill, because every object will have to be checked if it belongs to the degraded range, which is always true. In these cases where the degraded hash range is [0, UINT32_MAX] we can disable the R-tree that holds the ranges to skip and the filtering so that we can have the same performance with the basic backfill. The previous workaround should happen only when the checking of all objects in the PG is inevitable. Additionally, when the benchmarks were done, the replica PG that was on par with the primary one did not backfill at all. The reason is that with the current optimization we send a BACKFILL_FINISH message to the primary to indicate that there is no need to go through the recovery process for that particular PG. As expected, the more objects we have to recover, the closer we get to the basic backfill performance. Therefore, it has to be known that the experiments were done on a particular PG. If the writes did not take into account the object placement, the objects would be distributed across the PGs of the down OSD and the OSD would recover in parallel less objects per PG. In the experiments we see that for double the degraded objects, we get a bit more than double time to recover. Thus, if an OSD recovers in parallel two PGs with half the objects, the recovery theoretically should take a bit less than half the time. Nevertheless, the performance is noticeably improved per PG basis. In a real-case scenario the chance for all the PGs of an OSD to have the same number of objects being degraded is substantially low. Generally, only a handful of PGs will be affected while others may not even need recovery, because the CRUSH data placement algorithms take care of that. Therefore, it is proven that the implementation with the Merkle trees has improved the performance of the recovery procedure of Ceph, by filtering object groups which showed that they were not affected by any writes during the downtime, based on a unified hash value per object group.

Table 2: Backfill times of the dashed lines shown in figures 23 to 27 versus a simple linear estimation based on the first two rows of this table. Note that in the 50K experiment the degraded objects were closer to 9% and 19%.

	50K real	50K linear	20K real	20K linear	10K real	10K linear	5K real	5K linear
1%	325.6	325.6	131.4	131.4	66	66	7.2	7.2
5%	609	609	235.6	235.6	120	120	33.2	33.2
10%	896.6	892.4	364	365.85	184.8	187.5	67	65.7
20%	1582.4	1600.9	627.2	626.35	312.2	322.5	137.2	130.7

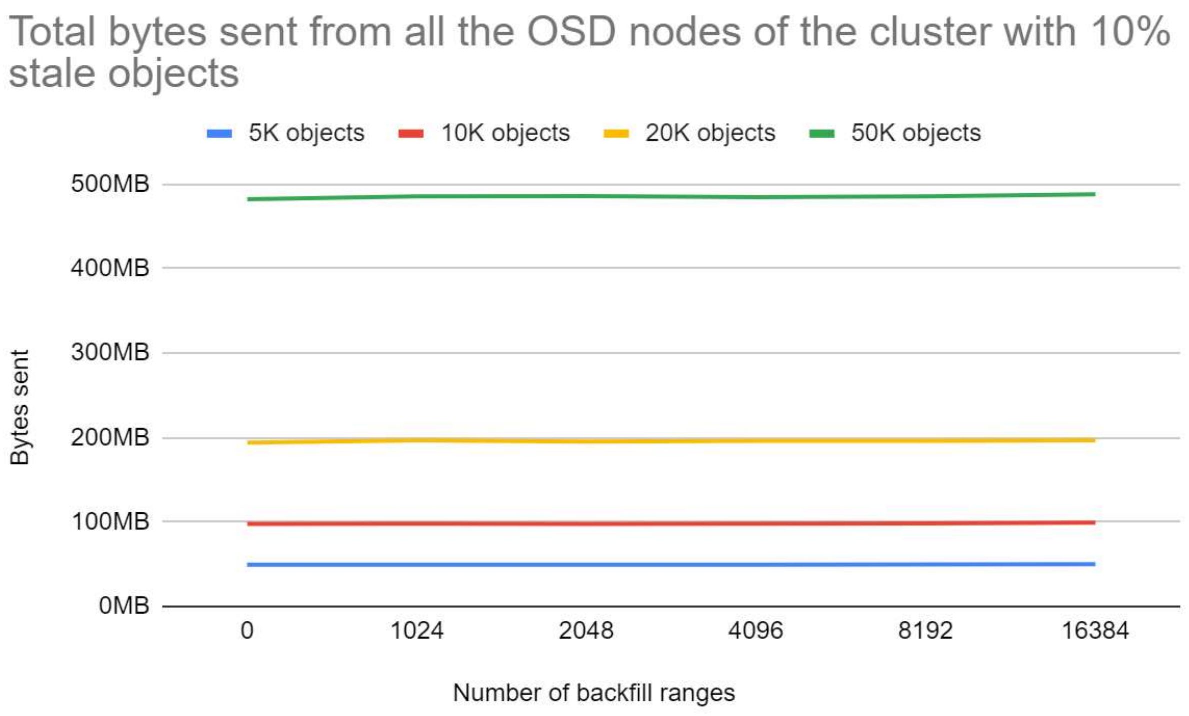

Figure 28: Total bytes sent and received from the start of the cluster to the point of the backfill finish. The numbers show a proportionate size of byte traffic respective to the number of objects recovered.

On the network side however, there is a close to zero difference between the different benchmarks. The below graph shows the total bytes sent by each OSD during the backfill times. Each type of benchmark has an almost flat line from the basic implementation of the backfill to the optimization with 16 thousand ranges. This observation is showing that the network exchange is heavily dependent on the size of the objects recovered, and any additional information message about the recovery process has a steady footprint based on the total size of the objects. The size of bytes sent includes everything from the start of the benchmark to the end of the backfill. For example, for the green line of the below graph, the accumulated bytes include the write of 50K objects, the update of the 10% of the objects and backfilling those to the down OSD, plus any other type of message sent, including heartbeats, reports to the monitor and such until the primary OSD receives the BACKFILL_FINISH message from the recovered PG.

Even if with the Merkle tree optimization we achieve better backfill times, the network traffic seems substantially unaffected compared to the usual traffic of the cluster on the non-optimised recovery. This is due to the type of messages sent. The backfill messages contain only a map of the hash of the objects and their versions, which are small in size compared to the actual size of the objects. If there is a mismatch, the filesystem backend, in this case Bluestore, will recover the object with the old version. Also, even if less metadata messages are sent, the exchange of the Merkle trees can cover any possible network traffic margin. Therefore, the object metadata messages sent to the primary can’t contribute much to the basic cluster’s traffic during the recovery.

5. CONCLUSIONS

The backfilling process of Ceph is the algorithm that synchronizes the objects between replicated placement groups when the in-memory WAL gets invalidated. For the backfill to bring stale replicas up to date, it has to list all the objects and their versions and give them to the primary replica. All writes pass through the primary replica, thus based on the way that writes are done, the primary replica should always have the latest data. The primary replica then asks for pairs of the objects' hash and version from the secondary replicas and synchronizes the objects according to the latest object version. The process is slow, because it must check every replicated object. By creating object groups in the replica, we can identify which groups of objects got altered during the downtime of an OSD. Then, there is a chance that some groups are not modified and by avoiding the version checks of those becomes a noticeable performance improvement. By grouping the objects based on their hash value and creating a change discovery mechanism based on Merkle Trees, we achieve time cuts for the backfill process ranging from 75% to 10%, based on the amount of the stale objects compared to the total of their placement group. On the best performing Merkle tree of depth 14 (16K leaves/ranges), we only observe an increase of ~100MB memory usage per OSD process, which is stable, because the Merkle tree structure remains intact as a full tree and it doesn't change its size throughout the whole backfill process. Compared to Apache Cassandra's 32K ranges, in our benchmarks half of those ranges were enough to greatly improve the backfill performance. On the network side, the use of Merkle trees does not introduce any noticeable load, which is a good sign. Regarding the CPU usage, during the backfill process there weren't any observable differences to be able to make a coherent conclusion. The CPU graphs didn't have a consistent pattern, and thus they were excluded from this thesis. One can revise the methods done to create the graphs and maybe find a more detailed CPU and network monitoring to show any possible differences in the Merkle tree implementation and the basic backfill.

The current optimised backfill implementation is a working example and it is thus far from perfect to be able to be used in production. First and foremost, Ceph is an elastic system, since it can grow and shrink in size on demand. As the PG number fluctuates, there should be a way to merge and split PGs without having to recalculate every hash to build the trees again. In the case of merging, the procedure is simple, we can XOR both leaves and recalculate the parent nodes of the tree, but in the case of splitting, if the hashes don't follow a pattern, there is no clear way to move the calculated hashes and building the tree will require to list every moved object and recalculate the hash values. Furthermore, there should be a schedule so that the tree is persisted to the disk periodically instead only on a clean shutdown. In cases where the OSD hits an error, the tree is lost and the restart of an OSD will be slower, because it will have to list every object and build the tree again before it enters the cluster. Moreover, the Merkle tree should be able to get updated also from the in-memory WAL recovery, because the writes talk directly with the backend (Bluestore) bypassing the normal client writes. Nonetheless, this proof of concept implements a lot of features that fully support the benchmarks done and they were able to show an improvement over the current implementation of Ceph's backfill recovery.

REFERENCES

[1] Gibson, Garth A., et al. "File server scaling with network-attached secure disks." Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems. 1997.

[2] Sandberg, Russel, et al. "Design and implementation of the Sun network filesystem." Proceedings of the Summer USENIX conference. 1985.

[3] Howard, John H. An overview of the andrew file system. Vol. 17. Carnegie Mellon University, Information Technology Center, 1988.

[4] Factor, Michael, et al. "Object storage: The future building block for storage systems." 2005 IEEE International Symposium on Mass Storage Systems and Technology. IEEE, 2005.

[5] XFS Filesystem, Silicon Graphics Inc. http://www.xfs.org/

[6] Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." Acm Sigact News 33.2 (2002): 51-59.

[7] Weil, Sage A., et al. "CRUSH: Controlled, scalable, decentralized placement of replicated data." SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. IEEE, 2006.

[8] M. Szeredi, File System in User Space, https://sourceforge.net/projects/fuse, (visited 2019).

[9] Lamport, Leslie. "The part-time parliament." Concurrency: the Works of Leslie Lamport. 2019. 277-317.

[10] Weil, Sage A., et al. "Ceph: A scalable, high-performance distributed file system." Proceedings of the 7th symposium on Operating systems design and implementation. 2006.

[11] Weil, Sage A., et al. "Rados: a scalable, reliable storage service for petabyte-scale storage clusters." Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07. 2007.

[12] Fielding, Roy T., and Richard N. Taylor. Architectural styles and the design of network-based software architectures. Vol. 7. Irvine: University of California, Irvine, 2000.

[13] RocksDB, Facebook Open Source https://rocksdb.org/

[14] O'Neil, Patrick, et al. "The log-structured merge-tree (LSM-tree)." Acta Informatica 33.4 (1996): 351-385.

[15] Ceph Bluestore, https://ceph.com/community/new-luminous-bluestore/ (visited 2019)

[16] Prabhakaran, Vijayan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Analysis and Evolution of Journaling File Systems." USENIX Annual Technical Conference, General Track. Vol. 194. 2005.

[17] Weatherspoon, Hakim, and John D. Kubiatowicz. "Erasure coding vs. replication: A quantitative comparison." International Workshop on Peer-to-Peer Systems. Springer, Berlin, Heidelberg, 2002.

[18] Wicker, Stephen B., and Vijay K. Bhargava, eds. Reed-Solomon codes and their applications. John Wiley & Sons, 1999.

[19] Amazon S3, https://aws.amazon.com/s3/

[20] Dropbox Magic Pocket, https://blogs.dropbox.com/tech/2016/05/inside-the-magic-pocket/

[21] Beaver, Doug, et al. "Finding a Needle in Haystack: Facebook's Photo Storage." OSDI. Vol. 10. No. 2010. 2010.

[22] LevelDB by Google https://github.com/google/leveldb

[23] Enslow, Philip Harrison. "What is a" distributed" data processing system?." Computer 11.1 (1978): 13-21.

[24] Stewart, Randall, et al. "Stream control transmission protocol." (2007).

[25] Mohan, Chandrasekaran, et al. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging." ACM Transactions on Database Systems (TODS) 17.1 (1992): 94-162.

[26] Van Renesse, Robbert, and Fred B. Schneider. "Chain Replication for Supporting High Throughput and Availability." OSDI. Vol. 4. No. 91–104. 2004.

[27] Alsberg, Peter A., and John D. Day. "A principle for resilient sharing of distributed resources." Proceedings of the 2nd international conference on Software engineering. 1976.

[28] GIT versioning system, https://git-scm.com/

[29] Merkle, Ralph C. "A digital signature based on a conventional encryption function." Conference on the theory and application of cryptographic techniques. Springer, Berlin, Heidelberg, 1987.

[30] xxHash fast non-cryptographic function, https://github.com/Cyan4973/xxHash

[31] SMHasher hash functions test suite, https://github.com/aappleby/smhasher

[32] Rousskov, Alex, and Duane Wessels. "Cache digests." Computer Networks and ISDN Systems 30.22-23 (1998): 2155-2168.

[33] Apache Cassandra Anti-entropy repair mechanism, https://docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbArch/archAntiEntropyRepair.html

[34] Gluster Documentation, https://docs.gluster.org/en/latest/

[35] HDFS consistency mechanism, https://issues.apache.org/jira/secure/attachment/12445209/appendDesign3.pdf

[36] Aghayev, Abutalib, et al. "File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution." Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019.

[37] All about XOR, https://accu.org/index.php/journals/1915

[38] Nakamoto, Satoshi. Bitcoin: A peer-to-peer electronic cash system. Manubot, 2019.

[39] Adya, Atul, et al. "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment." ACM SIGOPS Operating Systems Review 36.SI (2002): 1-14.

[40] Haeberlen, Andreas, Alan Mislove, and Peter Druschel. "Glacier: Highly durable, decentralized storage despite massive correlated failures." Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2. 2005.

[41] Kubiatowicz, John, et al. "Oceanstore: An architecture for global-scale persistent storage." ACM SIGOPS Operating Systems Review 34.5 (2000): 190-201.

[42] Braam, Peter. "The Lustre storage architecture." arXiv preprint arXiv:1903.01955 (2019).

[43] Corless, R.M., Gonnet, G.H., Hare, D.E., Jeffrey, D.J. and Knuth, D.E. On the LambertW function. *Advances in Computational mathematics,* 1996.

[44] Boneh, A. and Hofri, M., 1997. The coupon-collector problem revisited—a survey of engineering problems and computational methods. *Stochastic Models*.

[45] Buyya, R., Broberg, J. and Goscinski, A.M. eds., 2010. *Cloud computing: Principles and paradigms* (Vol. 87), John Wiley & Sons.