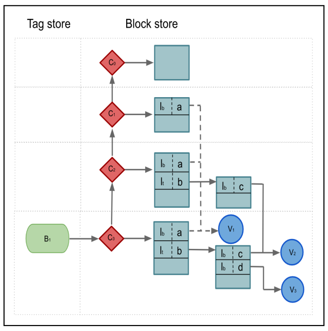
DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF
TECHNOLOGY
MADRAS
CHENNAI-600 036

Banyan: Coordination-free Distributed Transactions over Mergeable Types



A thesis

Submitted by

SHASHANK SHEKHAR DUBEY

For the award of the degree

Of

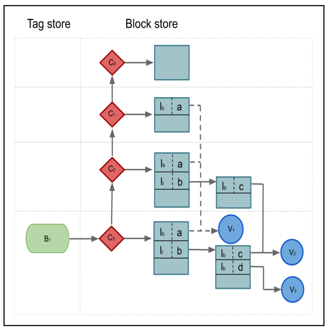
MASTER OF SCIENCE by

Research

September, 2021

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF
TECHNOLOGY
MADRAS
CHENNAI-600 036

Banyan: Coordination-free Distributed Transactions over Mergeable Types



A thesis

Submitted by

SHASHANK SHEKHAR DUBEY

For the award of the degree

Of

MASTER OF SCIENCE by

Research

September, 2021

To my parents, whose determination became my motivation

THESIS CERTIFICATE

This is to undertake that the Thesis titled, **Banyan: Coordination-free Distributed Transactions over Mergeable Types** submitted by me to the Indian Institute of Technology Madras, for the award of M.S. is a bonafide record of the research work done by me under the supervision of Dr. KC Sivaramakrishnan. The contents of this Thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Chennai - 600 036 **Shashank Shekhar Dubey**

Date: September 17, 2021

Dr. KC Sivaramakrishnan

LIST OF PUBLICATIONS

Shashank Shekhar Dubey, KC Sivaramakrishnan, Thomas Gazagnaire, and Anil Madhavapeddy. **"Banyan: Coordination-Free Distributed Transactions over Mergeable Types."** Programming Languages and Systems: 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30–December 2, 2020, Proceedings. Springer Nature.

ACKNOWLEDGEMENTS

This thesis is not only about the work that I have done in the MS program but also about the experience that I had in the journey, the experience that I will cherish all my life and pass it on. Every time I see this thesis, I would recall the ups and downs, challenges, support, love, and respect I had in my journey. I have met many remarkable individuals whose suggestions, comments, and feedback have consistently motivated me and helped me during my time at IITM. While I cannot mention all of those people, I want to take this opportunity to thank a few of them without whom this day would have never come for me to write this message.

First and foremost, I would like to express my sincere gratitude to my mentor and guide, Dr. KC Sivaramakrishnan. He agreed to take me in and support me at the time when I was most directionless in my journey. I can not thank him enough for this and would remember this all my life. When I first met him and asked what he expected from his students, he said, "I don't expect any prior subject knowledge, just a determination to learn, and yeah, enthusiasm for coding." Now that I am finishing my project with him, I can tell how he took me in as a clean slate and taught me concepts, reasoning, critical thinking, and made me a pragmatic programmer. The sense of fulfillment that I feel upon finishing this project is beyond comprehension. I am thankful to him for being so patient with me and guiding me, correcting me, and at the same time keeping me motivated towards my goal. I will always remember our discussions where, when stuck, he always asked me to take a step back and think again. This mantra will be my soulmate in my career ahead. I have not only learned problem-solving but also people skills from him. Being patient and at the same time keeping the mark of the progress of his students, I can only wish to learn his ways. No question was ever a wrong question with him, and our interesting research discussions, which could go for hours, well beyond our meeting time duration, have made me love my work. Dear sir, thank you for all this and many more experiences that I will never forget.

I want to thank Prof. D. Janakiram sir, for all the knowledge, support, and valuable life lessons that I will use all my life to sail through difficult situations. I can not forget the support of Prof. Jayalal, Prof. Rupesh, and Prof. Krishna Moorthy Sivalingam, who supported me during a very difficult period of my journey and helped me sail through it. Thank you for all your support.

Banyan is a programming model built over Irmin developed by OCaml labs, University of Cambridge and Tarides, France. While working on this project, there were numerous occasions when I got stuck and felt clueless. The open-source community of OCaml and Irmin has given me all the help I needed. I am thankful to them for taking up with all my doubts, regardless of being silly or reasonable, and helping me out. I thank them for creating the awesome library that Irmin is, without which materializing this idea would have been very difficult.

While my journey was no less than a roller-coaster, I was certainly not alone. I had the greatest support system one could wish for. I am lucky to have Nikhilesh, Pritha, Himanshu, Janani, and Nitesh as my dearest friends and companions. Nikhilesh made me realize that there is always more to life than what is seen in the present. Every time I talked with Pritha, I was amazed by her kindness and understanding. I learned from her that every time you make a big decision in life, remember that you are more important than your goal. There were times when my routine was taking over me, and that's not a good thing. That is when Himanshu reminded me that sometimes taking the sidewalk is better than the road. I have learned determination from Janani. The ultimate message that I took from her that, eventually, it does happen, so keep going and don't stop, has kept me motivated all along. And when I felt lost and clueless, Nitesh told me that a dream is as important as life itself and gave me the courage to restart. I am grateful to them for being there for me all the time in one way or another and wish them all the happiness in the world.

I want to thank my friends and batchmates in IIT, Sagar, Ajay, Rajendra, Nirav, Siddharth, and Himanshu, for providing me with their unique perspectives every time I approached them. I want to thank my friend and labmate Atul for all the fun times and short trips we had outside the campus. I can never forget his love for chai, his craving for samosa, and his knowledge of homemade remedies. Thanks for sharing all that with me. I feel delighted when I recall all the intellectual and knowledgeable discussions I had with Sumit and Shubham in the lab. I am thankful for the positive impact that it had on me. I want to thank Gaurav, Pradeep, Nishank, Akshay, and Anupreksha for being there for me. I thank my seniors Jyothi and Aman, who shared their invaluable experience and supported me when I needed it the most.

My journey at IITM will be incomplete if I don't mention MITR. My experience at MITR has shaped me into who I am and how I deal with people. I want to thank Vishal for being a great source of inspiration as a mentor and supporting me as a friend whenever I needed it.

I met these amazing people at IITM. But I can not forget those people without whose encouragement I would have never made it to IIT. I want to thank my friends Sourabh, Aayush, and Sanjay for their continuous motivation and encouragement not just during my preparation for GATE but also during every difficult decision of my life. I want to thank Ankit for giving me the occasional distractions that I needed the most. The talks on politics, tech, and life not only made me more knowledgeable and gave me perspective but also re-energized me for the work I was doing.

I can not thank enough to my parents for all their love and support. All the achievements of my life are the result of their relentless strive. I pray to God to help me become the person they have always wanted me to be. I want to thank my brother Mayank and sister Shreya for always being the last resort I need to go for anything I need. Their love, motivation, encouragement, and feedback have always made me improve myself and made me a better version of myself.

ABSTRACT

KEYWORDS: Distributed applications, Mergeable types, Three-way merge, Cassandra, Git

Programming loosely connected distributed applications is a challenging endeavour. Loosely connected distributed applications such as geo-distributed stores and intermittently reachable IoT devices cannot afford to coordinate among all of the replicas in order to ensure data consistency due to prohibitive latency costs and the impossibility of coordination if availability is to be ensured. Thus, the state of the replicas evolves independently, making it difficult to develop correct applications. Existing solutions to this problem limit the data types that can be used in these applications, which neither offer the ability to compose them to construct more complex data types nor offer transactions.

In this work, we describe Banyan, a programming model and a system for developing loosely connected distributed applications. Data types in Banyan are equipped with a *three-way merge* function à la Git to handle conflicts. Banyan provides isolated transactions for grouping together individual operations which do not require coordination among different replicas. We present a formal operational semantics for Banyan over a core Git-like store. To our knowledge, our semantics is the first formal description of the semantics of Git-like store equipped with three-way merges. We prove the correctness of a novel form of garbage collection using our semantics. Thanks to our semantics, we have also discovered bugs in the merge semantics of Irmin, a widely used distributed database built on the principles of Git. Banyan programming model can be instantiated over any weakly consistent distributed store. To illustrate this, we instantiate Banyan over Cassandra, an off-the-shelf industrial-strength distributed store. Several benchmarks, including a distributed build-cache, illustrates the effectiveness of the approach.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS i

ABSTRACT iii

LIST OF TABLES viii

LIST OF FIGURES xi

GLOSSARY xii

ABBREVIATIONS xiii

CHAPTER 1 INTRODUCTION 1

1.1 Consistency 1

1.2 Isolation levels 2

1.3 Trade-offs in distributed systems 3

1.4 Eventually consistent databases 5

1.4.1 Convergent Replicated Data Types (CRDT) 6

1.4.2 Mergeable Replicated Data Types (MRDTs) 7

1.4.3 Counter data type with MRDT 8

1.4.4 Limitations 10

1.5 Objective and scope of the thesis 12

1.6 Organization of the thesis 13

CHAPTER 2 MOTIVATION: A DISTRIBUTED BUILD CACHE 14

2.1 Mergeable types 14

2.2 Transactions 17

CHAPTER 3 PROGRAMMING MODEL 20

3.1 Banyan API 21

3.2 Transactions 23

CHAPTER 4 OPERATIONAL SEMANTICS 25

4.1 Syntax 25

4.1.1 Banyan objects 26

4.1.2 Banyan requests 27

4.2 Write Request 28

4.2.1 Banyan object creation with write request 30

4.3 Read Request 35

4.4 Lowest Common Ancestor 36

4.4.1 Finding the Lowest Common Ancestor 36

4.4.2 Operational Semantics of finding LCA 37

4.5 Merging two commits 39

4.6 Merging tree objects 42

4.7 Communication Operations 46

4.8 Connect and close operations 47

4.9 Garbage Collection 48

4.9.1 Banyan Garbage Collector 48

4.9.2 Operational Semantics of Banyan GC 50

4.9.3 Proof of GC correctness 53

CHAPTER 5 IMPLEMENTATION **60**

5.1 Irmin data model 61

5.2 Cassandra instantiation 62

5.3 Recursive merges 64

CHAPTER 6 EVALUATION **67**

6.1 Experimental setup 67

6.2 Baseline overheads 68

6.3 Mergeable Types 69

6.4 Distributed build cache 73

CHAPTER 7 RELATED WORK **75**

CHAPTER 8 LIMITATIONS AND FUTURE WORK **78**

CHAPTER 9 CONCLUSION **79**

REFERENCES **83**

LIST OF TABLES

Table	Title	Page
4.1	List of reachable commits from a given commit	38
4.2	List of descendant commits from a given commit	38
4.3	List of reachable objects from a given objects in figure 4.17	51

LIST OF FIGURES

1.1 CAP theorem: Under partition, a distributed system would have to trade-off between consistency and availability 4

1.2 PACELC theorem: In case of partition choice is between Availability and Consistency otherwise, Latency and Consistency 5

1.3 MRDT Counter: Initial values at `Replica1` and `Replica2`. `Replica2` is a clone of `Replica1` 8

1.4 MRDT Counter: Individual updates at `Replica1` and `Replica2` . . . 8

1.5 MRDT Counter: Merging `Replica1` into `Replica2` 9

1.6 MRDT Counter: Merging `Replica2` into `Replica1` after some individual updates 10

1.7 MRDT Counter: Convergence of `Replica1` and `Replica2` after merging them without any new updates. 10

1.8 Performance of Git pull: Each data point indicates the time to pull 10k byte-sized additional objects. It shows how the time taken to pull same number of objects increases as the number of objects in the local repository increases. 12

2.1 A slice of the build cache key-value store 15

2.2 Merging conflicting statistics updates. 17

2.3 Compiling `lwt_mutex.ml`. 18

3.1 Banyan system and programming model. 20

4.1 Syntax of Banyan store and requests 25

4.2 Semantics of Write 28

4.3 Initial state of the tag store and block store. Rounded rectangle is a tag, diamond is a commit, and rectangle is an empty tree object. 31

4.4 State of the tag store and block store after adding first key-value. Rounded rectangle is a tag, diamond is a commit, rectangle is a tree object, and circle is a blob. 31

4.5 State of the tag store and block store after adding second key-value. Rounded rectangle is a tag, diamond is commit, rectangle is a tree object, and circle is blob. 32

4.6 State of the tag store and block store after adding third key-value. Rounded rectangle is a tag, diamond is a commit, rectangle is a tree object, and circle is a blob. 34

4.7 Semantics of Read 35

4.8 Common Commits: C_{11} and C_{13} are common commits between two branches. C_{13} is the LCA. Arrow points to the parent(s) of the commit. 36

4.9 Reachable, Descendant and LCA commits 37

4.10 Two branches with head commit C_1 and C_2 are merged to form a new merged commit C_{12} 39

4.11 Commit C_1 and C_2 have conflict due to different values for the same key 40

4.12 Helper functions 42

4.13 Banyan Merge 43

4.14 Communication Operations 46

4.15 Session connect and close operations 47

4.16 Garbage collection. Here, the commits `p0-c0` and `s0-c0` may be deleted. 49

4.17 Reachable objects: Arrows show the objects that are connected with each other in the directional manner. Diamond is a commit, rectangle is a tree and circle is a blob object 50

4.18 Garbage Collection 52

5.1 A sample Irmin store. The rounded rectangles are tags, diamonds are commit objects, rectangles are tree object, and circles are blob objects. 61

5.2 Recursive merge. Rounded rectangles are the results of recursive merges. 64

6.1 Performance comparison between Banyan and Cassandra on LWW string value. 68

6.2 Performance of counter MRDT. 70

6.3 A snapshot of linked log storage. 72

6.4 Performance of mergeable logs. 72

6.5 Performance of complete reuse of build artefacts. 73

6.6 Performance of partial reuse of build artefacts. 74

GLOSSARY

The following are some of the commonly used terms in the thesis:

Distributed applications	An application which runs on more than one machine and communicate over the network
Sequential datatype	Datatypes such as lists, arrays, etc which can only safely be used under single threaded execution.
Write-skew	It is an anomaly in database that occurs with snapshot isolation. Two transactions cocurrently read a dataset, update them and commit them concurrently, without seeing other's updates. It creates a write-write conflict.
Throughput	Number of database operations performed per second
Latency	Time taken to perform one database operation
Continuous Integration System	Continuous Integration system is a regression testing framework that helps automate the integration of code changes from multiple contributors into a single software project.
Sticky availability	Ensuring availability by keeping the connection between a client and a particular logical node of the distributed server persistent for a given session.
Three-way merge	A merging technique in a version control system which computes the difference between the two objects, with respect to a common ancestor object. The merge is performed based on changes made in the two objects after the point of diversion.

ABBREVIATIONS

IoT	Internet of Things
DB	Database
LWW	Last-Write-Wins
MRDT	Mergeable Replicated Data Type
LCA	Lowest Common Ancestor
CRDT	Convergent Replicated Data Type

CHAPTER 1

INTRODUCTION

Modern web applications are supposed to be fast and always on. It is essential because they support time-critical operations like bank transactions, Internet gaming, collaborative document editing, etc. These applications typically host their data over multiple machines (replicas) for three reasons: improving throughput, lower user-perceived latency, and tolerating partial failures. Since there are multiple machines that can address the same requests, we can distribute the requests optimally among them to provide higher throughput. Replicas that are located near to the client provide faster accessibility, hence lower latency. Furthermore, since all replicas are equivalent in terms of data they provide, if one of the replicas goes down, others can handle the requests directed towards that replica, providing fault tolerance for partial failures. Such a design of systems is called as distributed system. The following section describes some of the definitions that would help understand the nature of applications running on distributed systems.

1.1 CONSISTENCY

Consistency in a distributed database refers to an expectation that any change in the database should be reflected in all the replicas in a certain specified manner. It is a safety property that deals with the correctness of data over all the replicas. There are several kinds of consistency levels designed to ensure the consistency of data among the replicas. These levels affect the time after which a particular write can be seen or read from the store over all the replicas. Typically, stronger the consistency level, the easier it is for the application developer to develop correct applications, but more expensive it is in terms of latency, throughput, and fault tolerance for the system to ensure the consistency level. Some of these consistency levels are described below:

Sequential Consistency: It implies that all the operations appear to take place in some total order and that order is consistent with the order of operation on each replica. This order may not be the same as the one in which the operations are executed, but it should be agreed upon by all the replicas involved.

Causal Consistency: Causal consistency requires causally related operations to be ordered in the same manner over all the replicas. Two operations could be considered as causally related if they follow a happens-before relationship (Lamport, 2019). Operations that are not causally related can be performed in any order. For example a write that uses the value of a previous read is said to be causally happen after that read.

Eventual Consistency: Eventual consistency does not guarantee any kind of ordering for the operations but it does ensure that the final value at all the replicas is eventually the same. This is the weakest form of consistency for distributed databases.

1.2 ISOLATION LEVELS

While consistency talks about the order in which individual operations must be made visible, isolation levels describe the expectations from a set of operations executed by the client (a transaction). It includes serializability, snapshot isolation, parallel snapshot isolation, Monotonic Atomic View, Read Committed and other relavent ones to what Banyan provides. A few of them are described below:

Serializability: Serializability ensures that in a transaction, the order of operations performed over all the replicas over any data item are the same as they were executed. If the set of concurrent operations performed are serializable, then it would mean that the final outcome would be the same as if the operations were performed in some serial order of transaction.

Snapshot isolation: Snapshot isolation ensures that a replica that needs to perform a transaction would read the last committed value in the database at the time it started the transaction. Concurrent updates made over the same committed value by different replicas which would result in a conflict won't be allowed.

1.3 TRADE-OFFS IN DISTRIBUTED SYSTEMS

Strong consistency and isolation levels ensure that before every new operation, the effect of the previous operation is reflected over all the replicas. This makes it easier to design correct applications. However, strong consistency and isolation levels adversely affects the performance of the system. The CAP theorem and the PACELC theorem described in this section draws the relationship between the consistency, availability, and latency in the distributed systems.

CAP: As shown in figure 1.2, CAP (Gilbert and Lynch, 2002) stands for Consistency-Availability-Partition tolerance. It states that if the distributed system suffers a partition, i.e., if the two replicas are unable to communicate with each other, then we can either choose the system to remain strongly consistent or available. Strong consistency requires immediate visibility of an update at every replica, which is not possible under a partition.

Hence, client operations will have to be blocked, resulting in the loss of availability. However, the choice is not a binary one since there are multiple levels of consistency. A system can be designed for which the availability depends upon the level of consistency it ensures. A strong consistency level would adversely affect the availability, whereas high availability all the time would mean using the weakest form of consistency.

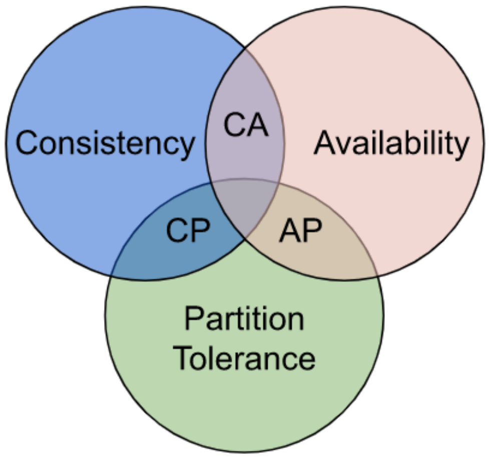


Figure 1.1: CAP theorem: Under partition, a distributed system would have to trade-off between consistency and availability

PACELC: PACELC theorem (Abadi, 2012) builds upon the CAP theorem. It states that in the presence of network **Partition**, we need to choose between system **Availability** and **Consistency**, **Else** if the system is not partitioned the choice is between **Latency** and **Consistency**.

Unlike CAP theorem, which assumes that a distributed system would always experience a network partition, PACELC points out that there is a trade-off that we need to consider even without partition. A distributed system that is designed to ensure the availability of the system in case of a device failure would replicate its data. This means that if there is no network partition, an update over one of the hosts has to be replicated to others.

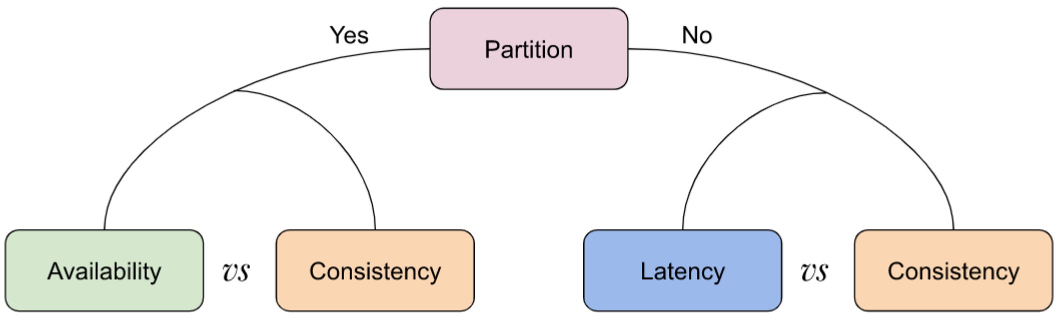


Figure 1.2: PACELC theorem: In case of partition choice is between Availability and Consistency otherwise, Latency and Consistency

Replication affects the latency depending upon the network bandwidth, congestion, etc., and until the replication is completely done, the system remains inconsistent. This trade-off between consistency and latency is important to make the system highly available even in normal conditions.

1.4 EVENTUALLY CONSISTENT DATABASES

Ideally, we would like to have a system that is highly available and exhibits low latency. However, like the CAP and PACELC theorems suggest, creating a system with strong consistency, high availability and low latency is not possible. This limitation has spurred the development of commercial weakly consistent distributed databases for wide-area applications such as DynamoDB (DynamoDB, 2021), Cassandra (Cassandra, 2021), CosmosDB (CosmosDB, 2021) and Riak (Riak, 2021). However, developing correct applications under weak consistency is challenging due to the fact that the operations may be reordered in complex ways even if issued by the same session (Burckhardt *et al.*, 2014). This makes it difficult to develop applications using eventually consistent databases, as it may lead to conflicts during convergence. Methods like using Convergent Replicated Data Types and Mergeable Replicated Data Types can help in resolving these conflicts.

1.4.1 Convergent Replicated Data Types (CRDT)

One of the major concerns of a distributed application is the consistency of data. CRDTs provide a set of data structures that ensures conflict-free convergence. It is because all the operations performed over these data structures are commutative. While CRDTs help in creating practical applications over weakly consistent databases, it provides only a limited set of data structures to work with. Data structures like Sets, Maps, and Graphs can be used as CRDTs as the operation performed over them are commutative. However, other important data structures such as a queue and stack can not be used as their operations like push and pop do not commute. There are several CRDT based built-in methods by which eventually consistent databases try to resolve these conflicts, such as:

Last-Writer-Wins: When multiple replicas converge, it is possible that more than one replica have updates for the same location. Last-Writer-Wins (LWW) chooses the value of the replica which made the last update. LWW typically uses the timestamp of the replicas at the time of write operation to compare the timings of the updates. LWW is used by the Cassandra distributed database to resolve conflicts.

Multi-valued data structures: These are the data structures that can store multiple values together. When there are conflicting updates, such data structures can keep the values from all the replicas, and give the user a chance to choose the final value based on application requirements. Dynamo DB uses this method to resolve the conflict.

These methods help in successfully converging the values over all the replicas, there is only a limited set of data types which offer such measures. Moreover, such built-in conflict resolution leads to anomalies such as write-skew (Berenson *et al.*, 1995) which makes it difficult (and often impossible) to develop complex applications with rich behaviors. Furthermore, CRDTs are not composable. Unlike ordinary data types which allow arbitrary contents through parametric polymorphism (generics), CRDTs do not permit such composition. Container CRDTs such as CRDT sets usually can only hold primitive elements. In particular, CRDT set with CRDT counter will not be a CRDT. Such a data structure will have to be implemented from scratch.

1.4.2 Mergeable Replicated Data Types (MRDTs)

Mergeable Replicated Data Types (MRDTs) (Kaki *et al.*, 2019) gives a way to automatically derive correct distributed variants of ordinary data types. The inductively defined data types are equipped with an invertible relational specification which is used to derive a three-way merge function à la Git (Git, 2021), a distributed version control system. MRDT tracks the point of diversion between two versions of data and uses this information to resolve the conflicts using the three-way merge method. Infact, a CRDT that stores the entire history acts exactly like an MRDT. That said, three-way merge certainly makes it easier to develop the applications which is the point of MRDTs. In the following sections, we describe the working of MRDTs with an example of a counter data type and the limitations of MRDT in creating the distributed applications.

1.4.3 Counter data type with MRDT

We take an example of an application built using the counter data type. Such a data type can have several use cases like calculating the number of accesses a certain web page has had. Since the web page could be hosted over several replicas of the server, a distributed counter would help us accumulate the results from all over the replicas and maintain the count without strong coordination.

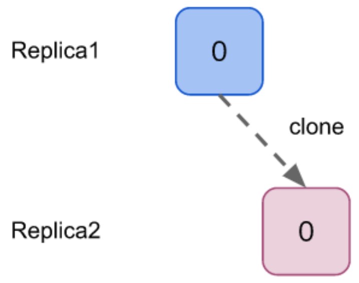


Figure 1.3: MRDT Counter: Initial values at `Replica1` and `Replica2`. `Replica2` is a clone of `Replica1`

In this example, we consider that there are two replicas **`Replica1`** and **`Replica2`**, shown in figure 1.3. The initial counter value at **`Replica1`** is **`0`**. **`Replica2`** is the clone of **`Replica1`**, meaning it is branched out of **`Replica1`**. Hence the values at both the replicas at this stage are the same.

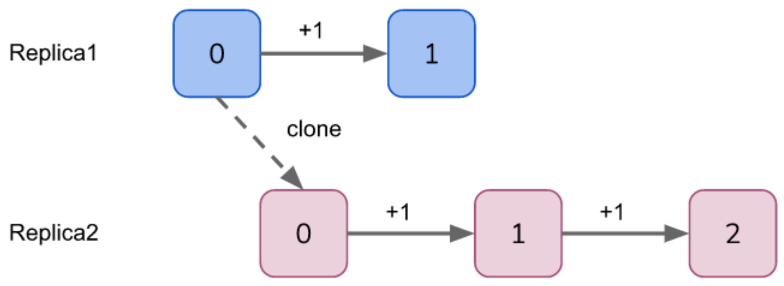


Figure 1.4: MRDT Counter: Individual updates at `Replica1` and `Replica2`

Now consider that there are several hits to our web page hosted at each replica. Separately updating the counter requires a simple increment to the existing values. Change in state of the two replicas is shown in figure 1.4.

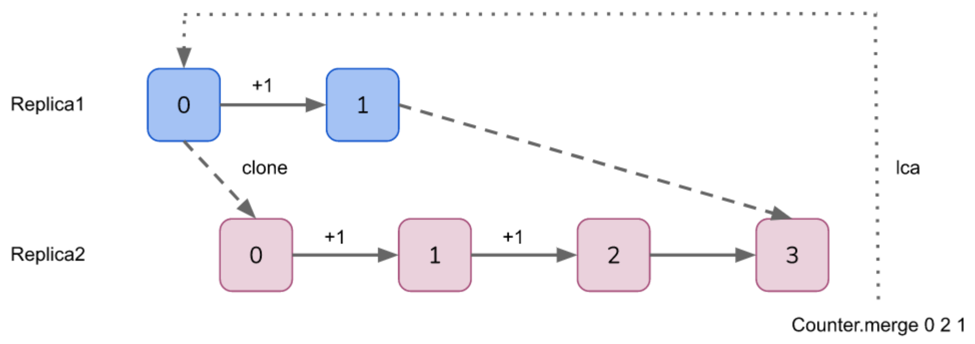


Figure 1.5: MRDT Counter: Merging `Replica1` into `Replica2`

At this point, if **`Replica2`** wants to update the value at its end with the total number of hits on the web page over all the replicas, then it will merge its counter value with that of **`Replica1`**. Figure 1.5 shows that the final value is the result of the merge operation performed using the latest value at the **`Replica1`** ($r1$), the latest value at the **`Replica2`** ($r2$), and the Lowest Common Ancestor (l) in the history of the two replicas.

Note that LCA is found from the history of the updates stored by the MRDTs. We will see in the chapter 5 how MRDTs represent this history. The merge for counter datatype is defined as: $v = (r1 - l) + (r2 - l) + l$. The total count is the sum of the counts at the lowest common ancestor and the counts in each of the replicas since.

`Replica2` gets the value 3, which is the total number of updates made over both the replicas. After few more updates at the local sites, **`Replica1`** wants to update its counter value by including the updates from **`Replica2`**. For this, **`Replica1`** needs to merge the values from **`Replica2`**. Figure 1.6 shows the newer updates and the merge result.

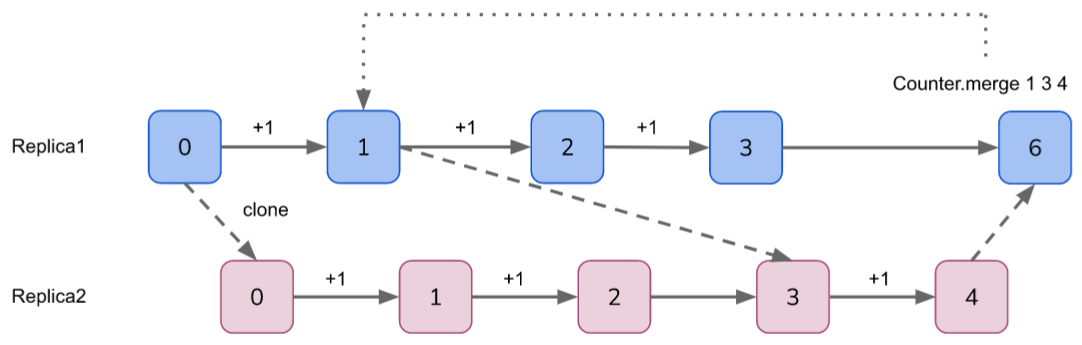


Figure 1.6: MRDT Counter: Merging `Replica2` into `Replica1` after some individual updates

MRDT implicitly computes the LCA of two replicas for every merge operation. **`Replica1`** gets the value as 6, which is the total number of updates made over all the replicas. If **`Replica2`** again merges with **`Replica1`**, then both the replicas would converge to the same value. Figure 1.7 shows the final convergence.

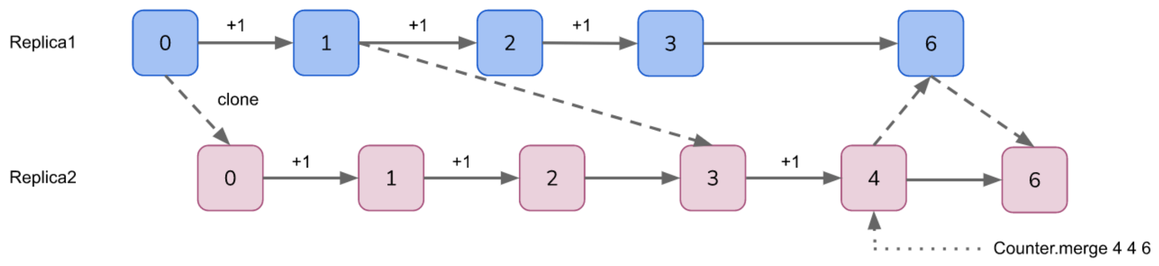


Figure 1.7: MRDT Counter: Convergence of `Replica1` and `Replica2` after merging them without any new updates.

1.4.4 Limitations

What does it take to make MRDTs practically viable for creating high-throughput, low-latency distributed applications such as the ones that would be implemented over industrial-strength distributed databases? There are several key challenges to getting there. While MRDTs define merge semantics for operations on individual objects, Kaki et al. do not describe the semantics of composition of operations on multiple objects, i.e., transactions. Transactions are indispensable for building complex applications. Strongly consistent distributed transactions suffer from unavailability (Abadi, 2012), whereas highly-available transactions (Bailis *et al.*, 2013*a*) combined with weakly consistent operations often lead to incomprehensible behaviours (Viotti and Vukolić, 2016). In addition, MRDTs impose a significant burden on the storage and network layer to be able to support three-way merges to reconcile conflicts. Kaki et al. implement MRDTs over Irmin (Irmin, 2021), a Git-like store for arbitrary objects, not just files. As with Git, in order to reconcile conflicts, three-way merges in MRDTs require the storage layer to record enough history to be able to retrieve the *lowest common ancestor* (LCA) state. For a distributed database, the performance of the network layer is quite important for throughput and latency. Industrial-strength distributed databases use gossip protocols (Kermarrec and van Steen, 2007) to quickly disseminate updates in order to ensure fast convergence between the replicas. Git comes equipped with a remote protocol for transferring objects between remote sites using *push* and *pull* mechanisms. However, the cost of transferring objects between two git remotes (replicas) using the Git remote protocol is proportional to the number of objects in the store. Figure 1.8 shows the time taken to pull 10k byte-sized objects successively from a remote repository into an initially empty local repository. The initial pull fetches 10k objects, and each subsequent pull fetches another 10k objects until the local repository has 100k objects. So, to pull 50k objects, five pull operations are performed subsequently pulling 10k objects each. All the clients perform the pull operation concurrently and the time taken is the total time taken from the start of the operation till the end of the last operation. Observe that even though the same number of objects are being fetched in each pull, the performance worsens as the number of objects in the local repository increases. The performance is also worse, with multiple clients pulling concurrently. Since the pull may update the state of multiple objects in order to be correct with respect to multi-object updates, the system will have to remain unavailable for multiple seconds when the pull is in progress. Clearly, the Git remote protocol is not suitable for high-throughput, low-latency distributed applications.

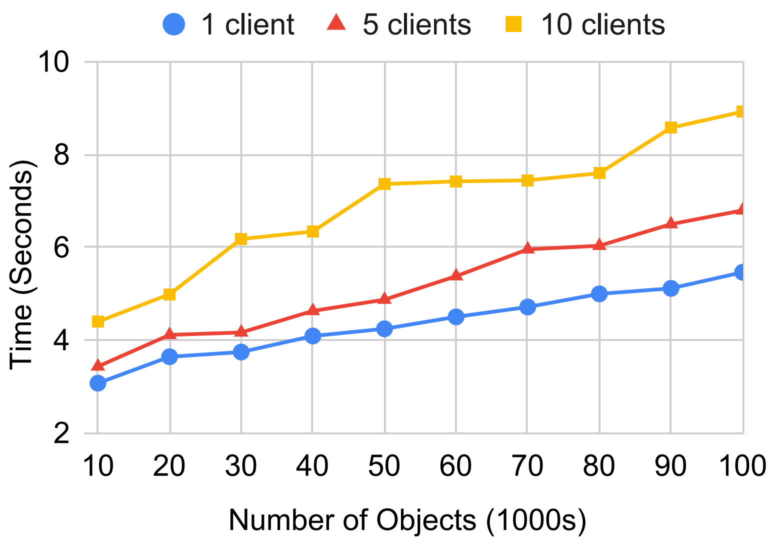


Figure 1.8: Performance of Git pull: Each data point indicates the time to pull 10k byte-sized additional objects. It shows how the time taken to pull same number of objects increases as the number of objects in the local repository increases.

1.5 OBJECTIVE AND SCOPE OF THE THESIS

In this work, we present Banyan, a programming model for building loosely connected distributed applications that provide coordination-free transactions over MRDTs. Banyan provides per-object causal consistency, and the transaction model is built on the principles of Git-like branches. Rather than relying on Git remote protocol for dissemination across replicas, we instantiate Banyan on top of Cassandra, an industrial-strength, off-the-shelf distributed store (Lakshman and Malik, 2010). Unlike Git, Banyan does not expose named branches explicitly and ensures eventual convergence. Importantly, Banyan only relies on eventual consistency, and it can be instantiated on any eventually consistent key-value store. We do not utilize the strong consistency features of Cassandra in our implementation. Extensive evaluation shows that Banyan makes it easy to build complex high-performance distributed applications.

1.6 ORGANIZATION OF THE THESIS

The rest of the thesis is organized as follows. We motivate the Banyan model by designing a distributed build cache in the next chapter. Chapter 3 describes the Banyan programming model. Chapter 4 describes the formal operational semantics of the banyan programming model and proves the correctness of a novel form of garbage collection that is essential for the practical viability of Banyan. Chapter 5 describes the instantiation of Banyan on Cassandra, an off-the-shelf distributed database. We evaluate the instantiation of Banyan on top of Cassandra in chapter 6. Chapter 7 and 9 present the related work and conclusions, respectively.

CHAPTER 2

MOTIVATION: A DISTRIBUTED BUILD CACHE

In this chapter we motivate Banyan programming model by building a distributed build cache. A distributed build cache enables a team of developers and/or a continuous integration (CI) system to reuse the build artefacts between several builds. Such a facility is provided by modern build tools such as Gradle (Gradle, 2021) and Bazel (Bazel, 2021), which can store and retrieve build artefacts from cloud storage services such as Amazon S3 or Google Cloud Storage. Consider the challenge of building a distributed build cache for OCaml packages. Let us assume that the builds are reproducible – that is, independent builds of the same source files yield the same artefact. In addition to storing the artefacts, it would be useful to gather statistics about the artefacts such as creation time, last accessed time and number of cache hits. Such information may be used in the cache eviction policy or replicating artefacts across several sites for increased availability. While an artefact itself is reproducible, care must be taken to ensure that the statistics are consistent. For the sake of exposition, we will assume that all the build hosts use the same operating system and compiler version.

2.1 MERGEABLE TYPES

Let us build this distributed cache using Banyan, implementing it in OCaml. At its heart, Banyan is a distributed key-value store. The keys in Banyan are *paths*, represented as list of strings. The values are algebraic data types equipped with a merge function that reconciles conflicting updates. In this example, we will use the following schema: `[<pkg_name>; <version>; <kind>; <filename>]` for the keys, where `<kind>` is either

lib indicating binary artefact or **stats** indicating statistics about the artefact. The value type is given below:

```
type timestamp = float
type value =
    | B of bigarray  (*binary artefact*)
    | S of timestamp (*created*)
        * timestamp (*last accessed*) * int (*hits*)
```

The value is either a binary artefact or a statistics triple where a timestamp is a Unix timestamp represented as seconds elapsed since the start of clock on January 1, 1970. Figure 2.1 shows the slice of the build cache key-value store. The cache stores the artefacts (**cmx** and **cmi** files) produced as a result of compiling the source file **lwt_mutex.ml** from the package **lwt** version **5.3.0**. The build cache also stores the statistics for every artefact. The example shows that the **lwt_mutex.cmx** was accessed 25 times.

Key	Value
/lwt/5.3.0/lib/ lwt_mutex.cmx	B(0x…)
/lwt/5.3.0/lib/ lwt_mutex.cmi	B(0x…)
/lwt/5.3.0/stats/ lwt_mutex.cmx	S(1593518762.20, 1593518822.36, 25)

Figure 2.1: A slice of the build cache key-value store

When several developers and/or CI pipelines are running concurrently on different hosts, they may attempt to add the same artefact to the store, or, if the artefact is already present, retrieve it from the cache and update the corresponding artefact statistics. It would be unwise to synchronize across all of the hosts for updating the store, and suffer the latency hit and potential unavailability. Hence, Banyan only writes an update to one of the replicas. The replicas asynchronously share the updates between each other, and resolve conflicting updates using user-defined three-way merge function. The merge function for the build cache is given below.

Merge function for build cache

```
1 let merge (lca: value option) (v1: value) (v2: value)
                                                   : value =
2  match lca, v1, v2 with
3  | None, B a1, B a2 (* no lca *)
4  | Some (B _), B a1, B a2 -> assert (a1 = a2); B a1
5  | None, S(c1,la1,h1), S(c2,la2,h2) -> (* no lca *)
6      S(min c1 c2, max la1 la2, h1 + h2)
7  | Some(S(_,_,h0)), S(c1,la1,h1), S(c2,la2,h2)->
8      S(min c1 c2, max la1 la2, h1 + h2 - h0)
9  | _ -> failwith "impossible"
```

The key idea here is that Banyan tracks the *causal history* of the state updates such that it is always known what the *lowest common ancestor* (LCA) state is, if one exists. This idea is analogous to how Git tracks history with the notion of *branches*. The merge function is applied to the LCA and the two conflicting versions to determine the new state. In the case of build cache, since the builds are reproducible, the binary artefacts will be the same (line 4). The only interesting conflicts are in the statistics. The merge function picks the earliest creation timestamp, latest last accessed timestamp, and the sum of the new cache hits since the LCA in the two branches and the original value at the LCA, if present (lines 5–8).

Figure 2.2 shows how the merge function helps reconcile conflicts. The arrows capture the happens-before relationship between the states. Assume that replica `r2` starts off by cloning the branch corresponding to replica `r1`. Subsequently both `r1` and `r2` performed local updates. The remote updates are reconciled by calling the merge function on each of the conflicting values. The value **v5** is obtained with merging the values **v3** and **v4** with **v1** as LCA.

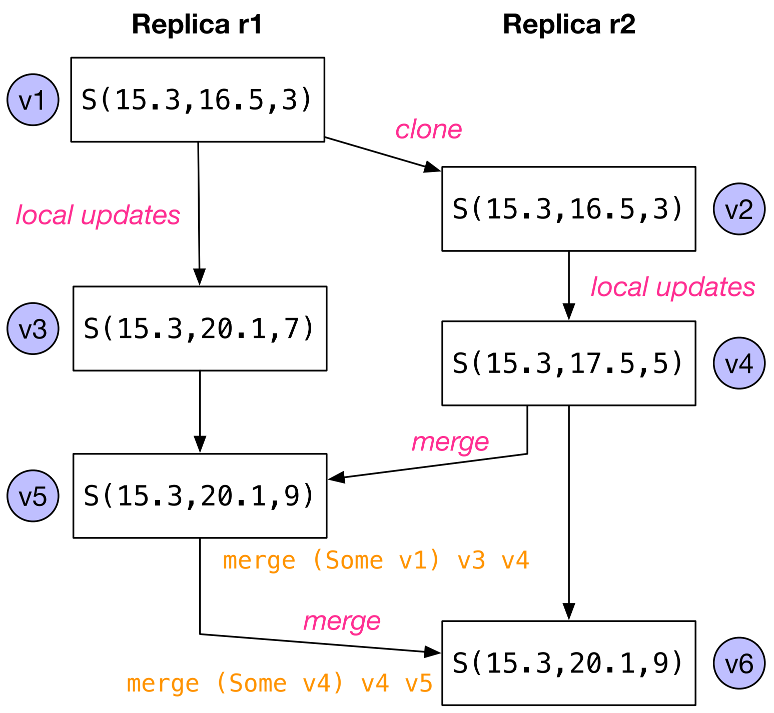


Figure 2.2: Merging conflicting statistics updates.

Importantly, observe that the cache hit count is **9** in **v5** which corresponds to the sum of **3** hits in the initial state, **4** additional hits in **r1** and **2** additional hits in **r2**. At this point, **r1** has all the changes from **r2**, but the vice-versa is not true. Subsequently, when **r1** is merged into **r2**, both the replicas have converged.

2.2 TRANSACTIONS

Now that we know the mergeable value type for the build cache, let us see how we can compile **lwt_mutex.ml** using Banyan. Figure 2.3 shows the code for compiling

lwt_mutex.ml.

```
let compile s (* session *) =
   let ts = Unix.gettimeofday () in
   let lib = ["lwt";"5.3.0";"lib"] in
   let stats = ["lwt";"5.3.0";"stats"] in
   refresh s >>= fun () ->
   read s (lib @ ["lwt_mutex.cmx"]) >>= fun v ->
   match v with
   | None ->
      let (cmx, cmi, o) = ocamlopt "lwt_mutex.ml" in
      write s (lib @ ["lwt_mtex.cmx"]) (B cmx) >>= fun _ ->
      write s (stats @ ["lwt_mutex.cmx"]) (S (ts,ts,0))
                                              >>= fun _ ->
      ... (* similarly for cmi and o files *)
      publish s >>= fun _ ->
      return (cmx, cmi, o)
   | Some cmx ->
      read s (stats @ ["lwt_mutex.cmx"])
                                 >>= fun (Some M(c,la,h)) ->
      write s (stats @ ["lwt_mutex.cmx"]) (S (c,ts,h+1))
                                              >>= fun _ ->
      read s (lib @ ["lwt_mutex.cmi"])
                                       >>= fun (Some cmi) ->
      read s (lib @ ["lwt_mutex.o"]) >>= fun (Some o) ->
      ... (* update stats for cmi and o file *)
      publish s >>= fun _ ->
      return (cmx, cmi, o)
```

Figure 2.3: Compiling `lwt_mutex.ml`.

In Banyan, the clients interact with the store in *isolated* sessions. Clients can perform a **read** operation to read a key from the session and **write** operation to write a key-value pair into the session. A session can fetch recent updates using the **refresh** primitive and make *all* the local updates visible to other sessions using the **publish** primitive. During **refresh**, any conflicting updates are resolved using the three-way merge function associated with the value type.

In order to compile **`lwt_mutex.ml`**, we first refresh the session to get any recent updates. Then, we check whether the **`lwt_mutex.ml`** file is in the build cache. If not, the source file is compiled, and the resultant artefacts (**`cmx`**, **`cmi`**, **`o`** files) and the corresponding entries for updated statistics are written to the store. Finally, all the local updates are published.

The all or nothing property of **`refresh`** and **`publish`** is critical for the correctness of this code. Observe that when the artefact is locally compiled, all the artefacts and their statistics are published atomically. This ensures that if a session sees the **`cmx`** file, then other artefacts and their statistics will also be visible. Thus, Banyan makes it easy to write highly-available, complex distributed applications in an idiomatic fashion.

CHAPTER 3

PROGRAMMING MODEL

In this chapter, we shall describe the system and programming model of Banyan from the developers point-of-view. The Banyan store consists of several replicas, which are fully or partially replicated (Crain and Shapiro, 2015). The replication factor is left to the underlying database. The replicas asynchronously distribute updates amongst themselves until they converge. The key property that enables Banyan to support mergeable types and isolated transactions is that Banyan tracks the history of the store in the same way that Git tracks the history of a repository. Figure 3.1 presents the schematic diagram of the system and programming model.

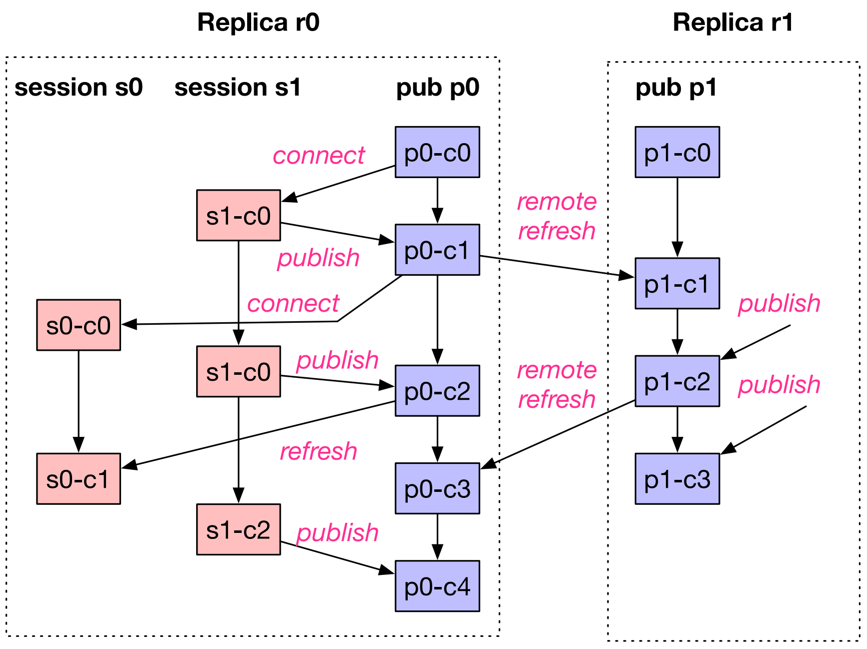


Figure 3.1: Banyan system and programming model.

Each replica has a distinguished public branch **pub**, which records the history of the changing state at that replica. Each node in this connected history graph represents a *commit*. Whenever a new client connection is established, a new branch is forked off the latest commit in the public branch. Any reads or writes in this session are only committed to this branch unless explicitly published. This ensures the isolation property of each session. The figure shows the creation of two sessions in the replica **r0**.

3.1 BANYAN API

The simplified Banyan API is given below:

```
type config  (* Store configuration *)
type session
type key = string list
type value   (* Type of mergeable values in the store *)

val connect : config -> session Lwt.t
val close   : session -> unit Lwt.t
val read    : session -> key -> value option Lwt.t
val write   : session -> key -> value -> unit Lwt.t
val publish : session -> unit Lwt.t
val refresh : session -> unit Lwt.t
```

When a client connects to a Banyan store, a new session is created, which is rooted to one of the replicas in the store. Every write creates a commit in the session performing the write. As previously explained, Banyan permits the sessions to atomically **publish** their updates and **refresh** to obtain latest updates. The **publish** operation squashes all the local commits since the previous **refresh** or **publish** to a single commit, and then *pushes* the changes to the public branch on the replica to which the session is rooted. The **refresh** operation *pulls* updates from the public branch into the current sessions branch. Both **publish** and **refresh** may invoke the merge function on the value type if there are conflicts. The objects that are written to each replica are asynchronously replicated to other replicas. Banyan offers causal consistency for operations on each key.

Periodically, the changes from other public branches are *pulled* into a replica's public branch (remote refresh). This operation happens *implicitly* and asynchronously, and does not block the client on that replica. When a session is closed, the outstanding writes are implicitly published. Similarly, when a session is connected, there is an implicit refresh operation.

Observe that both the local and the remote refresh operations are non-blocking – it is always safe for `refresh` to return with updates only from a subset of public branches. The only push operation is due to `publish`. When pushing to a branch, it is necessary to atomically update the target branch to avoid concurrency errors. The key observation is that only the session that belongs to a replica can push to the public branch *on that replica*. This can be achieved with replica-local concurrency control and does not require coordination among the replicas. Hence, Banyan transactions do not need inter-replica coordination, and hence, are always available.

When a particular replica goes down, the sessions that are rooted to that replica may not have enough history to be able to `refresh` and `publish` to other replicas. In particular, `refresh` and `publish` will need to discover the LCA in the case of conflicting updates. Since the objects are asynchronously replicated across the replicas, the recent writes to the replica that went down may not have been replicated to other replicas. Hence, Banyan requires sticky availability (Bailis *et al.*, 2013*a*) – the sessions need to reach the logical replica to which it originally connected. In practice, with partial replication, a logical replica may be represented by a set of physical servers. As long as one of these physical servers is reachable, the system remains available for that session.

3.2 TRANSACTIONS

Compared to traditional transactions usually executed at a particular isolation level, **`refresh`** and **`publish`** permits more fine-grained, explicit control of visibility. In Banyan, transactions are delimited by **`publish`** operations, begin and end of sessions. For example, the set of writes performed between consecutive **`publish`** operations are made visible atomically outside the session. The transaction may abort if the three-way merge function throws an exception. However, in practice, the useful MRDTs are designed in such a way that a merge is always possible, and the failure of the merge function represents a bug. This idea of merge always being possible ensures *strong eventual consistency*, espoused by convergent replicated data types (Shapiro *et al.*, 2011). Banyan adds transactional support over strong eventual consistency.

The **`publish`** and **`refresh`** can be used to achieve well-known isolation levels. For example, consider parallel snapshot isolation (PSI) (Sovran *et al.*, 2011), which is an extension of snapshot isolation (SI) (Berenson *et al.*, 1995) for geo-replicated systems. Like SI, the transactions in PSI operate on a snapshot of the state at a replica. While SI precludes write-write conflicts, PSI admits them on mergeable types. Since all the data types in Banyan are mergeable types, every write-write conflict can be resolved. We can achieve PSI by **`refresh`**ing at that beginning of the transaction and **`publish`**ing at the end of the transaction with no intervening **`refresh`**es.

Similarly, we get monotonic atomic view (MAV) (Bailis *et al.*, 2013*a*) isolation level if two consecutive **`publish`** operations are interspersed with **`refresh`**es. Since the **`refresh`**es may bring in new updates from committed transactions, the state of the transaction grows monotonically.

CHAPTER 4

OPERATIONAL SEMANTICS

In the previous chapter, we have seen the Banyan programming model from a developer's point of view. In this chapter, we formalise the semantics of the Banyan store using structural operational semantics. The operational semantics formalises a core language that captures the essence of a Git-like store, and the Banyan programming model on top. Our aim with the operational semantics is to succinctly but precisely present the details of the Banyan programming model and illustrate how Banyan may be implemented on a different eventually consistent data store. In addition, we prove the correctness of a novel kind of garbage collection (GC) algorithm in Banyan, which is essential to keep the storage requirements of the model in check.

4.1 SYNTAX

$$
\begin{array}{rccl@{\qquad}rccl}
\text{Label} & l & := & l_b \mid l_t & \text{Block Store} & \mathcal{B} & := & \phi \mid \mathcal{B}[h \mapsto v] \\
\text{Branch} & \beta & := & \text{String} & \text{Store} & \mathbb{S} & := & \langle \mathcal{B}, \mathcal{T} \rangle \\
\text{Hash} & h & & & \text{Requests} & r & := & \text{read}(\beta_L, k, b) \\
\text{Name} & n & := & \text{String} & & & \mid & \text{write}(\beta_L, k, b) \\
\text{Key} & k & := & \overline{n} & & & \mid & \text{publish}(\beta_L, \beta_P) \\
\text{Blob Object} & b & & & & & \mid & \text{refresh}(\beta_L, \beta_P) \\
\text{Tree Object} & t & := & \phi \mid t[n \mapsto (l, h)] & & & \mid & \text{connect}(\beta_L, \beta_P) \\
\text{Commit Object} & c & := & (\overline{h}, h) & & & \mid & \text{close}(\beta_L, \beta_P) \\
\text{Values} & v & := & b \mid t \mid c & & & \mid & \text{remote\_refresh}(\beta_{ps}, \beta_{pt}) \\
\text{Tag Store} & \mathcal{T} & := & \phi \mid \mathcal{T}[\beta \mapsto h] & & & \mid & gc
\end{array}
$$

Figure 4.1: Syntax of Banyan store and requests

Figure 4.1 shows the syntax of Banyan stores and requests that would be performed over them. The Banyan store $\mathbb{S}$ is composed of an *immutable*, *content-addressable* block store $\mathcal{B}$ which stores all the *objects* used by banyan store, and a *mutable* tag store $\mathcal{T}$ which records the association between the branch names and its associated state. Both of the stores are represented as maps in the semantics. Block store maps a hash (h) to its corresponding value (v). Since the block store is content-addressed, given a block store $\mathcal{B}[h \mapsto v]$, $h = hash(v)$. These hashes h correspond to objects that are in the block store $\mathcal{B}$. The tag store maps a branch name (β) to a hash h of a commit node. The empty map is represented by ϕ.

4.1.1 Banyan objects

A value (v) is one of the several kinds of Banyan objects. Since Banyan is a Git-like store, we use the same terminology to name the various objects in Banyan store (GitOjbects, 2021). The objects are blobs (b), trees (t) and commits (C).

Let us first look at the structure of the tree object. Recall from the previous chapter that Banyan is a key-value store where the key is a path in a file system with support for recursive directory structure. The path is represented by a list of strings $[n_1; n_2; \ldots; n_k]$, where each entry is a *name* n. The names n_1 to n_{k-1} are directory names and the entry n_k is the name of the file. In the rest of the section, where appropriate, we will alternatively use $[n_1; n_2; \ldots; n_k]$ and $/n_1/n_2/\ldots/n_k$ to represent the same path. A tree object is represented as a map that maps a name to a tuple of a label (l) and a hash (h). Since each *entry* in the tree object may refer to a sub-directory or a file, we utilise the label l to disambiguate whether the entry is a reference to a sub-directory (which would be another tree object) l_t or a file (represented as a blob) l_b.

A commit object c is represented by a pair, where the first component of the pair is a list of hashes corresponding to the commit objects of parent commits, and the second component is a hash of the tree object corresponding to this commit which represents the state of the store at this commit.

4.1.2 Banyan requests

Operations in Banyan are in the form of requests (r). The operational semantics rules are described as $\mathbb{S} \xrightarrow{r} \mathbb{S}$ where they transform the store from one state to the other based on the request. Hence, the requests include the arguments as well as the results. As per the Banyan programming model, `read` and `write` operations are performed on a branch specific to the session, called the local branch of that session. Consequently, read request takes local branch (β_L) and a key (k) as the argument and returns a blob (b) as the result. A `write` request takes a key (k) and a blob (b) value and writes them into the specified local branch, β_L. The operations `publish`, `refresh` and `remote_refresh` are merge related operations which take two branches as the arguments and merge them. The `connect` operation creates a new local branch (β_L) by connecting to replica whose public branch is β_P. The `close` operation publishes the contents of the local branch β_L to the public branch β_P and then deletes the local branch to close the session. The `gc` operation is used to garbage collect the Banyan objects that would no longer be used.

4.2 WRITE REQUEST

$$\textsc{WriteTree} \quad \frac{h = \text{hash}(b) \qquad t' = t[n \mapsto (l_b, h)] \qquad \mathcal{B}' = \mathcal{B}[h \mapsto b]}{\text{write\_tree}(\mathcal{B}, [n], b, t) = (\mathcal{B}', t')} \tag{1}$$

$$\frac{\begin{array}{c} n \in domain(t) \qquad (l_t, ht) = t(n) \\ (\mathcal{B}', t') = \text{write\_tree}(\mathcal{B}, ns, b, \mathcal{B}(ht)) \qquad h = \text{hash}(t') \end{array}}{\text{write\_tree}(\mathcal{B}, n :: ns, b, t) = (\mathcal{B}'[h \mapsto t'], t[n \mapsto (l_t, h)])} \tag{2}$$

$$\frac{n \notin domain(t) \qquad (\mathcal{B}', t') = \text{write\_tree}(\mathcal{B}, ns, b, \phi) \qquad h = \text{hash}(t')}{\text{write\_tree}(\mathcal{B}, n :: ns, b, t) = (\mathcal{B}'[h \mapsto t'], t[n \mapsto (l_t, h)])} \tag{3}$$

$$\textsc{Write} \quad \frac{\begin{array}{c} hc = \mathcal{T}(\beta_L) \qquad (hp, ht) = \mathcal{B}(hc) \qquad \text{write\_tree}(\mathcal{B}, k, b, \mathcal{B}(ht)) = (\mathcal{B}', t') \\ ht' = \text{hash}(t') \qquad c' = ([hc], ht') \qquad hc' = \text{hash}(c') \end{array}}{\langle \mathcal{B}, \mathcal{T} \rangle \xrightarrow{\text{write}(\beta_L, k, b)} \langle \mathcal{B}'[ht' \mapsto t'][hc' \mapsto c'], \mathcal{T}[\beta_L \mapsto hc'] \rangle}$$

Figure 4.2: Semantics of Write

The `write`(β_L, k, b) request writes a pair with key k and blob b into the local branch β_L. Banyan `write` updates both the stores of Banyan in a way that it retains all the previous updates on the same key as its history. If k is not already present, it inserts the new key-value pair into the store; otherwise, it updates the value to b. Rule WRITE in figure 4.2 describes the behaviour of a `write` request that writes to a session's local branch β_L at key k with value b. The bulk of the work is done by the helper function `write_tree`($\mathcal{B}$, k, b, t) that inserts the value b at key k in the tree t with the block store $\mathcal{B}$. The function `write_tree` returns a new block store $\mathcal{B}'$ and a new tree t' with the new key and value. Lets look at the function `write_tree` first, described in rule WRITETREE(1), (2) and (3). WRITETREE(1) describes the addition of a key-value to the store where the key is a singleton list. Recall that the only name n in the list refers to the file. In this case, the new tree has the same entries as the old tree except for a new entry that maps n

to a pair of (l_b, hash(b)). The new block store $\mathcal{B}'$ has the same entries as $\mathcal{B}$ except the mapping of hash(b) to b.

Rule WRITETREE(2) and (3) describes the case when the key has more than one name in the list. Recall that the key represents the directory structure. Hence, each entry in the list is parsed starting from the top level directory until the last entry, i.e. the file name. The content of a directory or a file is hashed before storing into the block store to ensure that everything in the block store is content addressable. Rule WRITETREE(2) and (3) verifies if there already exists a directory structure same as the one given in the key list. To check that, the function `write_tree` verifies if the name n is present in the tree node t. In the semantics, the set of names n present in the tree t is represented as $domain(t)$. If the directory name n is present in the tree node, the function `write_tree` is called again to check if the sub-directories (ns) are also present in the same order as given in the key. To check the sub-directories the content of the current directory is parsed to get the hash of the sub-directory ($(l_t, ht) = t(n)$) and the sub-directory ($\mathcal{B}(ht)$) is passed as an argument to the `write_tree`. Alternatively, if the directory name n is not present in the tree node the function `write_tree` is called to walk through each directory name in the key with ϕ as the tree node. The function `write_tree` is recursively called as per Rule WRITETREE(2) or (3) until there is a single value in the key list, the case for which is described in rule WRITETREE(1). A new block store and a tree node is returned back from this function. The new block store $\mathcal{B}'$ has all the entries received from the `write_tree` function along with an added entry hash(t') mapping to t'. Similarly, the new tree node t has all the received entries in addition to a new entry n mapping to (l_t, hash(t')).

The `write` function triggered over the local branch β_L searches for an entry related to β_L in the tag store $\mathcal{T}$. If the entry is found, $\mathcal{T}$ returns the commit hash hc of the latest entry in the β_L. A lookup with this commit hash into the $\mathcal{B}$ returns a pair of values which also contains the corresponding tree hash. The tree hash along with the $\mathcal{B}$, k, and b is used to call the helper function `write_tree`, which returns a new block store $\mathcal{B}'$ and a new tree node t'. The `write` function creates a new commit node c' which contains the hash of the previous commit node hc as its parent and hash of the new tree node ht'. The `write` function creates a new block store $\mathcal{B}'$ which contains all the previous entries, entries made in the `write_tree` function, a mapping from hash(t') to t' for the new tree node, a mapping from hash(hc') to c' for the commit node and an entry into the tag store updating the handle of local branch β_L to point to the latest commit hash.

4.2.1 Banyan object creation with write request

Before describing about the semantics of other Banyan requests, lets see with an example how Banyan objects are created and manipulated with the `write` operation. For the purpose of explanation here we will write three key-values into the branch **`B1`** of the store and see how objects are created and stored in the block store and the tag store. The key-value pairs taken for this example are:

- Key: [a] Value: V1
- Key: [b; c] Value: V2
- Key: [b;d] Value: V3

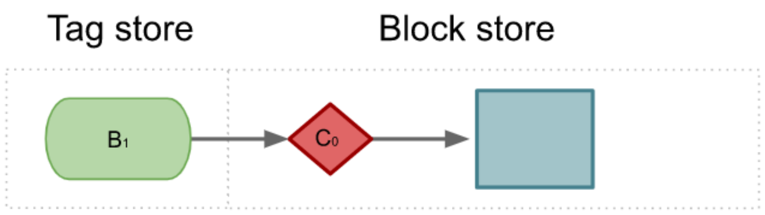


Figure 4.3: Initial state of the tag store and block store. Rounded rectangle is a tag, diamond is a commit, and rectangle is an empty tree object.

All the Banyan objects are stored in the block store. Tag store contains the branch names which point to the latest commit in that branch. Initially, branch **`B1`** points to a commit C_0, which in turn points towards an empty tree object. Figure 4.3 shows the initial state of the two stores. For the first write operation in Banyan, C_0 will act as a parent commit.

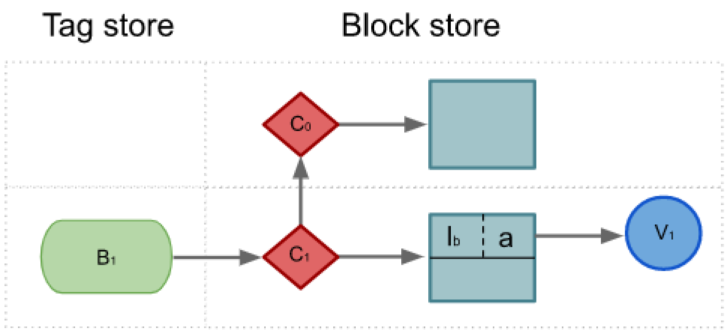


Figure 4.4: State of the tag store and block store after adding first key-value. Rounded rectangle is a tag, diamond is a commit, rectangle is a tree object, and circle is a blob.

Lets write the key **`[a]`** and value **`V1`** into the store now. Figure 4.4 demonstrates the objects created after this write operation. When the key **`[a]`** and the value **`V1`** is added into the Banyan store, a new blob for **`V1`** is created and stored in the block store. A new tree object is created with an entry for key **`[a]`** with label $\mathtt{l}_b$, pointing towards the blob **`V1`**. $\mathtt{l}_b$ denotes that the tree entry points to a blob. A new commit node C_1 is created which points to the latest tree node and the commit node C_0 denoting its parent. The entry in the tag store related to branch **B1** which was earlier pointing to commit node C_0 would now point to C_1 as C_1 is the latest commit for branch **B1**.

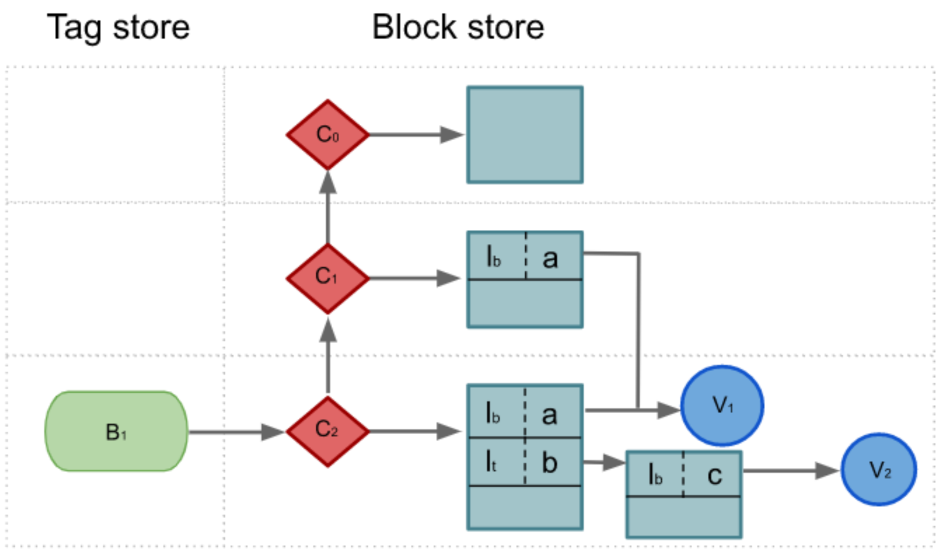


Figure 4.5: State of the tag store and block store after adding second key-value. Rounded rectangle is a tag, diamond is commit, rectangle is a tree object, and circle is blob.

The next write operation would write key `[b;c]` and value **V2** into the Banyan store. Recall that a key represents a directory structure. In this case **b** is the top-level directory and **c** is the nested directory inside **b**. The tree node pointed by the branch's latest commit always contains the entries related to the top-level directory of all the keys present in that branch. Each tree entry points to another tree object containing all its nested directories or the blob, as may the case be.

Figure 4.5 demonstrates the state of the store after writing key `[b;c]` and value **V2** into it. Since **V2** does not already exist in the block store, a new blob object is created for it.

A new tree object with entry for `c` and label $\mathtt{l}_b$ is created which points to the blob of `V2`. Further, a new top-level tree object is created, which contains all the entries from the previous top-level tree object, in addition to a new entry with `b` and label $\mathtt{l}_t$. Label $\mathtt{l}_t$ is due to the fact that the tree entry points to another tree object containing `c`. Notice that a new tree object is created which contains entries for both `a` and `b`, instead of just editing the previous tree object and adding `b` to it. The new objects are created because the block store is immutable, and we can not change the objects once created. As a result, we have two tree objects which contain `a` and both must point to a blob `V1`. Also, notice that there is a single blob object for `V1`, pointed by both the tree objects. Banyan avoids creating duplicate objects as it is content addressable. It helps in the re-usability of the objects. At last, a new commit node C_2 is created, which points to the latest tree object and the previous commit C_1, denoting C_1 as its parent. A pointer to the previous commit is stored because Banyan is a persistent store and keeps track of the history without removing any previous objects. Tag store is updated so that the handle for branch `B1` would point to the latest commit C_2.

Figure 4.6 demonstrates the state of the store when the key `[b;d]` and value `V3` is written into it. This example is just an extension of the previous write operation. Notice that the tree entry of `c` and `d` share the same tree object which is pointed by the tree entry `b`. This denotes that the top level directory b contains two entries `c` and `d` nested to it. All the three keys are accessible from the tag store entry of branch `B1`.

An important observation here is that there are several objects created for every write operation, and several others are retained in the store. This affects the storage requirements and performance of the Banyan. We discuss its impact and some optimization techniques in detail in Section 4.9 and Chapter 6.

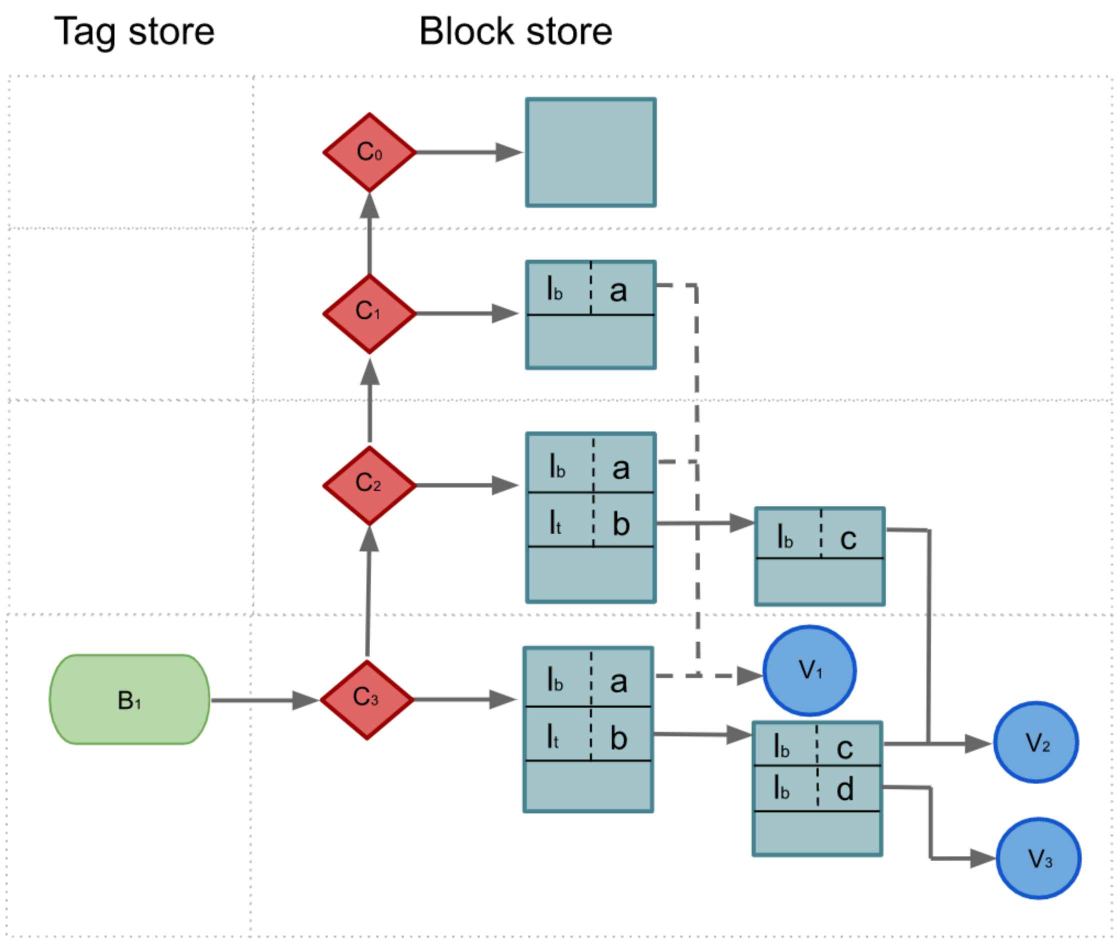


Figure 4.6: State of the tag store and block store after adding third key-value. Rounded rectangle is a tag, diamond is a commit, rectangle is a tree object, and circle is a blob.

4.3 READ REQUEST

$$\textsc{ReadTree}\quad \frac{(l_b, h) = t(n)}{\text{read\_tree}(\mathcal{B}, [n], t) = \mathcal{B}(h)} \quad (1)$$

$$\frac{(l_t, h) = t(n) \qquad \text{read\_tree}(B, ns, \mathcal{B}(h)) = b}{\text{read\_tree}(\mathcal{B}, n :: ns, t) = b} \quad (2)$$

$$\textsc{Read}\quad \frac{hc = \mathcal{T}(\beta_L) \qquad (hp, ht) = \mathcal{B}(hc) \qquad \text{read\_tree}(\mathcal{B}, k, \mathcal{B}(ht)) = b}{\langle \mathcal{B}, \mathcal{T} \rangle \xrightarrow{\text{read}(\beta_L, k, b)} \langle \mathcal{B}, \mathcal{T} \rangle}$$

Figure 4.7: Semantics of Read

Banyan `read`(β_L, k, b) reads the key k into the local branch β_L and returns the result as blob b. It does not change anything in any of the two stores. Figure 4.7 shows the semantics for Banyan read. Semantic rule READ suggests that the Banyan fetches the hash (hc) of the head commit of the local branch β_L from the tag store, and then the corresponding commit node is read from the block store. A commit node is a tuple that contains a list of hashes hp, denoting the parents of this commit and hash of a tree object. Further, with the helper function `read_tree`, the tree object is parsed with reference to the key to find if the key is present in the tree object. If the key is found, it returns the associated value; otherwise, it returns empty (ϕ).

If the key contains a single name n, tree object must have a mapping of n to a tuple with l_b and a hash h to its value. In this case the `read_tree` will read h into the block store ($\mathcal{B}(h)$) and return the value. Otherwise, it will return ϕ, considering the key provided is not available (rule READTREE (1)).

Semantic rule READTREE (2) describes the steps if the key contains more than one name.

In this case, the key is parsed from the top level directory name till the last. For each name n, `read_tree` finds out the tree entry containing the mapping of n. The hash h associated with that mapping is used to find the next tree object from the block store ($t = \mathcal{B}(h)$). For a successful search, the next name in the key should be available in t. The method is recursively applied until there is only a single name in the key. Rule READTREE (1) described above guides the steps when there is a single name in the key.

4.4 LOWEST COMMON ANCESTOR

In this section we will see what is a lowest common ancestor (LCA), how to identify one and the semantic rule to find the LCA.

4.4.1 Finding the Lowest Common Ancestor

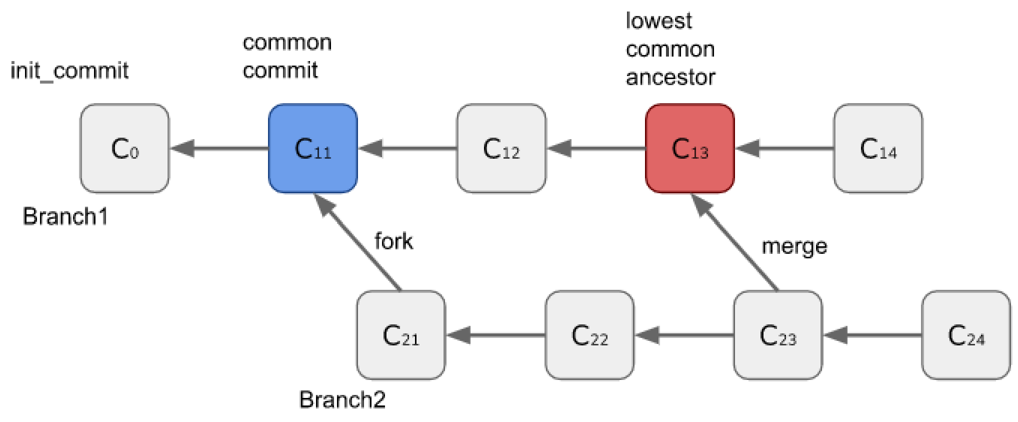


Figure 4.8: Common Commits: C_{11} and C_{13} are common commits between two branches. C_{13} is the LCA. Arrow points to the parent(s) of the commit.

Before understanding the rule to compute the LCA, lets understand what a LCA is and how it is created. A Lowest Common Ancestor (or LCA) is a special kind of commit node that is shared by two branches. To elaborate, a commit is a way of adding an update into the store. Every new commit follows its previous commit and records the previous one as its parent. A branch is created by a single or a series of commits formed in this manner. Banyan allows forking a new branch out of an existing one. In this process, the two branches share a common commit initially and then diverge based on further commits. Another way to create a common commit is to merge the two branches. The latest of common commits between two branches is called their Lowest Common Ancestor (LCA). Figure 4.8 shows how common commits are created and the LCA of the two commits.

4.4.2 Operational Semantics of finding LCA

$$\textsc{ReachableCommits} \qquad c \in \text{rea\_comm}(\mathcal{B}, c) \qquad (1)$$

$$\frac{(hp, ht) = c \qquad h \in \text{mem}(hp) \qquad c' \in \text{rea\_comm}(\mathcal{B}, \mathcal{B}(h))}{c' \in \text{rea\_comm}(\mathcal{B}, c)} \qquad (2)$$

$$\textsc{DescendentCommits} \qquad \text{des\_comm}(\mathcal{B}, c) = \{c' \mid c \in \text{rea\_comm}(\mathcal{B}, c')\} \backslash \{c\}$$

$$\textsc{LCA} \qquad \frac{\begin{array}{c} l \in \text{rea\_comm}(c_1) \qquad l \in \text{rea\_comm}(c_2) \\ \text{des\_comm}(\mathcal{B}, l) \cap \text{rea\_comm}(\mathcal{B}, c_1) \cap \text{rea\_comm}(\mathcal{B}, c_2) = \phi \end{array}}{l \in \text{lca}(\mathcal{B}, c_1, c_2)}$$

Figure 4.9: Reachable, Descendant and LCA commits

To understand the semantic rule to compute the LCA, first we need to understand the rule to find the reachable and descendent commits given in figure 4.9. Every commit is considered to be reachable to itself (rule ReachableCommits (1)). A commit c' is considered to be reachable from a commit c if c' is reachable from the parents of c

(rule ReachableCommits (2)). This definition forms a recursive method where every commit that is an ancestor of a commit c is a reachable commit for c. Table 4.1 shows the reachable commits for some of the commits in figure 4.9. If a commit c is reachable from

Table 4.1: List of reachable commits from a given commit

Commit	**Reachable Commits**
C_0	C_0
C_{11}	C_{11}, C_0
C_{12}	C_{12}, C_{11}, C_0
C_{14}	C_{12}, C_{11}, C_0
C_{21}	C_{11}, C_0
C_{23}	$C_{13}, C_{12}, C_{22}, C_{21}, C_{11}, C_0$
C_{24}	$C_{23}, C_{13}, C_{12}, C_{22}, C_{21}, C_{11}, C_0$

a commit c' then c' is a descendent of c, with an exception to itself. This means a commit is reachable to itself, but is not a descendent to itself (rule DescendentCommits). Table 4.2 show the list of descendants of the given commit in figure 4.8.

Table 4.2: List of descendant commits from a given commit

Commit	**Descendent Commits**
C_0	$C_{11}, C_{12}, C_{13}, C_{14}, C_{21}, C_{22}, C_{23}, C_{24},$
C_{11}	$C_{12}, C_{13}, C_{14}, C_{21}, C_{22}, C_{23}, C_{24},$
C_{13}	C_{14}, C_{23}, C_{24}
C_{14}	*no descendents*
C_{21}	C_{22}, C_{23}, C_{24}
C_{23}	C_{24}
C_{24}	*no descendents*

A lowest common ancestor of two commits c_1 and c_2 is a commit which is reachable from both c_1 and c_2, such that none of the descendents of this commit is reachable from

c_1 and c_2 (rule LCA). Basically, it is the most recent common commit between c_1 and c_2.

4.5 MERGING TWO COMMITS

In Banyan we often need to merge two branches, to share the updates made by individual sessions, through publish and refresh operations which is described later. Merging two branches means collecting all the key-value pairs written in the two individual sessions. Figure 4.10 shows two commits C_1 and C_2 with key-value as `[a;b] => v1` and `[c;d;e] => v2` respectively. Commit C_{12} shows the merged commit from which both the keys `[a;b]` and `[c;d;e]` are accessible.

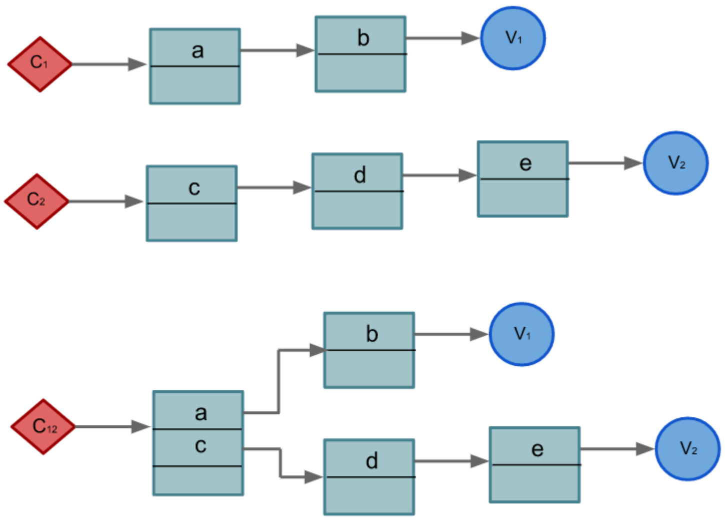


Figure 4.10: Two branches with head commit C_1 and C_2 are merged to form a new merged commit C_{12}

It is the tree objects of the latest commits that keep the latest set of key-value pairs that belong to that branch. Hence, to merge two commits, we need to merge their two tree objects. Merging the two commits may become complicated due to conflicts. There is said to be a conflict between two commits when there are different values for the same key in the two commits. Figure 4.11 shows a conflict between commits c_1 and c_2.

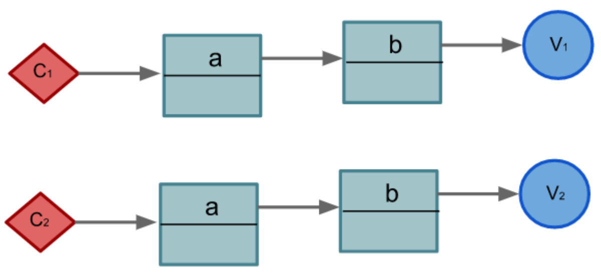


Figure 4.11: Commit C_1 and C_2 have conflict due to different values for the same key

The figure shows that there is a key [a;b] which is reachable from both commit c_1 and c_2 but have different values associated with them. In this case, it is unclear which value should be chosen as the value for the merged commit. Banyan uses a three-way merge mechanism to resolve such conflicts. For the key which is creating the conflict, it uses its values from the two commits and from the lowest common ancestor (LCA) commit. Finding LCA is described in the previous section. The three values are passed to a user-defined conflict resolution function, which provides an application specific solution to these conflicts.

Semantic rule Merge in figure 4.13 shows the semantics for merging the two branches β_s and β_t. It finds the latest commit objects of the two branches (cs and ct) and merges them to get the new commit c'. A merge operation ends in updating the tag store. For the branch which requested the merge, it updates its corresponding commit hash to that of the latest commit created after the merge.

As mentioned above, to merge two commits, we need to find their LCA. Rule LCA described in the previous section shows how the LCA is computed when two commits are given. A complex situation may occur when we are performing concurrent merge operations. Concurrent merge leads to a situation where there are two LCAs for a pair of commits, and none of them helps in computing the correct value. The solution to this problem is to recursively merge the two commits (found as LCAs) and keep doing so until we find a single LCA and then use it for resolving the original conflict we had. Section 5.3 describes how the concurrent merge occurs and how recursive merge solves the problem. Here, we will see how the LCA is computed by merging the list of commits.

Rule MERGECOMMITS (2) shows how to merge a list of commits. It takes two commit c_1 and c_2, merge them and then recursively merges other commits into the resultant commit c. Finally a single resultant commit c will be returned (rule MERGECOMMITS). To merge any two commits c_1 and c_2, `merge_commit` first computes its LCA. `merge_commit(`$\mathcal{B}$, `lca(`c_1, c_2`))` in the antecedent of the MERGECOMMITS (2) shows the recursive computation of LCA. The tree object of the resultant LCA commit c' and the other two commits c_1 and c_2 are then used to merge their keys. The resultant tree object t'' is used to create a new commit node which is then added to the block store and returned back for updating the tag store, as described earlier. The parent of the new commit object c' reflects that it is made by combining the contents of c_1 and c_2. In the figure 4.8, C_{23} is the result of merging C_{13} and C_{22}. Hence, the arrows point out that it has two parents.

In the next section, we will explore different situations which may occur when merging two tree objects.

4.6 MERGING TREE OBJECTS

A tree object contains the mapping of a key to its value. Merging two commits requires merging their two tree objects and create a new one with the final values. Rule MERGETREE in figure 4.13 shows the rules for merging the trees in different situations. Below we discuss each of those scenarios and how the final tree object is created.

The method `merge_tree` merges the tree objects t_1 and t_2 and uses tree object tl of the LCA to resolve the conflicts. It returns an updated block store and a new tree object.

If the tree objects which are needed to be merged are empty then MERGETREE returns the block store and an empty tree object (rule MERGETREE (1)).

Merging tree objects in Banyan is a commutative operation. Hence merging t_1 into t_2 or vice-versa, generates the same new tree object (rule MERGETREE (2)).

$$\textsc{Get} \qquad \text{get}(m,n) = \begin{cases} m(n) & \text{if } n \in \mathit{domain}(m) \\ \phi & \text{otherwise} \end{cases}$$

$$\textsc{Set} \qquad \text{set}(m,n,v) = \begin{cases} m' & \text{if } v = \phi \wedge m' = m[n \mapsto v'] \\ m[n \mapsto v] & \text{otherwise} \end{cases}$$

$$\textsc{Remove} \qquad \text{rem}(m,n) = \begin{cases} m' & \text{if } n \in \mathit{domain}(m) \wedge m' = m[n \mapsto v] \\ m & \text{otherwise} \end{cases}$$

Figure 4.12: Helper functions

When two tree objects have to be merged, we need to consider each of the names present in them. Merging trees require to compare the mapping for each name in all the three objects and take a decision to include them into the new tree object.

MERGETREE

$$\text{merge\_tree}(\mathcal{B}, \phi, \phi, \phi) = (\mathcal{B}, \phi) \tag{1}$$

$$\text{merge\_tree}(\mathcal{B}, tl, t_1, t_2) = \text{merge\_tree}(\mathcal{B}, tl, t_2, t_1) \tag{2}$$

$$\frac{\begin{array}{c}\text{get}(t_1, n) = \text{get}(t_2, n)\\ (\mathcal{B}', t') = \text{merge\_tree}(\mathcal{B}, \text{rem}(tl, n), \text{rem}(t_1, n), \text{rem}(t_2, n))\end{array}}{\text{merge\_tree}(\mathcal{B}, tl, t_1, t_2) = (\mathcal{B}', \text{set}(t', n, \text{get}(t_1, n)))} \tag{3}$$

$$\frac{\begin{array}{c}\text{get}(tl, n) = \text{get}(t_2, n) \qquad \text{get}(t_1, n) \neq \text{get}(t_2, n)\\ (\mathcal{B}', t') = \text{merge\_tree}(\mathcal{B}, \text{rem}(tl, n), \text{rem}(t_1, n), \text{rem}(t_2, n))\end{array}}{\text{merge\_tree}(\mathcal{B}, tl, t_1, t_2) = (\mathcal{B}', \text{set}(t', n, \text{get}(t_1, n)))} \tag{4}$$

$$\frac{\begin{array}{c}(\mathcal{B}', t') = \text{merge\_tree}(\mathcal{B}, \mathcal{B}(h_l), \mathcal{B}(h_1), \mathcal{B}(h_2)) \qquad h' = \text{hash}(t')\\ (\mathcal{B}'', t'') = \text{merge\_tree}(\mathcal{B}'[h' \mapsto t'], tl, t_1, t_2)\end{array}}{\begin{array}{c}\text{merge\_tree}(\mathcal{B}, tl[n \mapsto (l_t, h_l)], t_1[n \mapsto (l_t, h_1)], t_2[n \mapsto (l_t, h_2)])\\ = (\mathcal{B}'', t''[n \mapsto (l_t, h')])\end{array}} \tag{5}$$

$$\frac{\begin{array}{c}\text{get}(tl, n) = (l_b, h) \vee \text{get}(tl, n) = \phi \qquad (\mathcal{B}', t') = \text{merge\_tree}(\mathcal{B}, \phi, \mathcal{B}(h_1), \mathcal{B}(h_2))\\ h' = \text{hash}(t') \qquad (\mathcal{B}'', t'') = \text{merge\_tree}(\mathcal{B}'[h' \mapsto t'], tl, t_1, t_2)\end{array}}{\begin{array}{c}\text{merge\_tree}(\mathcal{B}, tl, t_1[n \mapsto (l_t, h_1)], t_2[n \mapsto (l_t, h_2)])\\ = (\mathcal{B}'', t''[n \mapsto (l_t, h')])\end{array}} \tag{6}$$

$$\frac{\begin{array}{c}m = \text{merge\_val}(\mathcal{B}(hl), \mathcal{B}(h_1), \mathcal{B}(h_2)) \qquad hm = \text{hash}(m)\\ (\mathcal{B}', t') = \text{merge\_tree}(\mathcal{B}[hm \mapsto m], tl, t_1, t_2)\end{array}}{\begin{array}{c}\text{merge\_tree}(\mathcal{B}, tl[n \mapsto (l_b, hl)], t_1[n \mapsto (l_b, h_1)],\\ t_2[n \mapsto (l_b, h_2)]) = (\mathcal{B}', t'[n \mapsto (l_b, hm)])\end{array}} \tag{7}$$

$$\frac{\begin{array}{c}(\mathcal{B}', t') = \text{merge\_tree}(\mathcal{B}, \mathcal{B}(hl), \phi, \mathcal{B}(h_2)) \qquad h' = \text{hash}(t')\\ (\mathcal{B}'', t'') = \text{merge\_tree}(\mathcal{B}'[h' \mapsto t'], tl, t_1, t_2)\end{array}}{\text{merge\_tree}(\mathcal{B}, tl[n \mapsto (l_t, hl)], t_1, t_2[n \mapsto (l_t, h_2)]) = (\mathcal{B}'', t''[n \mapsto (l_t, h')])} \tag{8}$$

$$\frac{(\mathcal{B}', t') = \text{merge\_tree}(\mathcal{B}, tl, t_1, t_2)}{\text{merge\_tree}(\mathcal{B}, tl[n \mapsto (l_b, hl)], t_1, t_2[n \mapsto (l_t, h_2)]) = (\mathcal{B}', set(t', n, get(t_2, n)))} \tag{9}$$

MERGECOMMITS

$$\text{merge\_commits}(\mathcal{B}, \{c\}) = (\mathcal{B}, c) \tag{1}$$

$$\frac{\begin{array}{c}\text{merge\_commits}(\mathcal{B}, lca(\mathcal{B}, c_1, c_2)) = (\mathcal{B}', c')\\ (hp_1, ht_1) = c_1 \qquad (hp_2, ht_2) = c_2 \qquad (hp', ht') = c'\\ (\mathcal{B}'', t'') = \text{merge\_tree}(\mathcal{B}'(ht'), \mathcal{B}'(ht_1), \mathcal{B}'(ht_2))\\ h'' = \text{hash}(t'') \qquad c = ([\text{hash}(c_1), \text{hash}(c_2)], h'')\end{array}}{\begin{array}{c}\text{merge\_commits}(\mathcal{B}, \{c_1, c_2\} \cup cs)\\ = \text{merge\_commits}(\mathcal{B}''[\text{hash}(c) \mapsto c], \{c\} \cup cs)\end{array}} \tag{2}$$

MERGE

$$\frac{\begin{array}{c}hcs = \mathcal{T}(\beta_s) \qquad hct = \mathcal{T}(\beta_t) \qquad cs = \beta(hcs)\\ ct = \mathcal{B}(hct) \qquad (\mathcal{B}', c') = \text{merge\_commits}(\mathcal{B}, \{cs, ct\})\end{array}}{\text{merge}(\mathcal{B}, \mathcal{T}, \beta_s, \beta_t) = (\mathcal{B}', \mathcal{T}[\beta_t \mapsto \text{hash}(c')])}$$

Figure 4.13: Banyan Merge

The helper function rem (rule REMOVE, figure 4.12) takes a map m and string n and removes the mapping of n from m. We use REMOVE in the semantics rule MERGETREE for removing the tree entries with mappings that are already considered in that tree object. Helper function get (rule GET) is used to fetch the mapping of name n in the map m. If the mapping is not present, it returns ϕ. Another function set (rule SET) writes a mapping of $(n \mapsto v)$ into m. If v is empty, it is considered to be request to delete the name from the map, and a map m' without that value is returned back.

If get(t_1, n) = get(t_2, n) then there is no conflict in the two tree objects regarding the name n. Hence, the new tree object will retain n and its mapping as it is, regardless of its value in the LCA. Rule MERGETREE(3) gives the semantics for this condition. Once n is resolved, it recursively checks the condition for other keys, until all the keys are covered.

If the value for a name n is different in tree object t_1 (belongs to branch$_1$) and tree object t_2 (belongs to branch$_2$) then there is a conflict. In this case the value in t_1 will be checked. Consider that the value of n in tl and t_2 are same, this means that the branch$_1$ has updated the value, where as branch$_2$ has retained the previous value. In this case the updated value will be retained in the merged tree (rule MERGETREE(4)). Since the Banyan merge is commutative (rule MERGETREE(2)), the semantics would still be same if tl and t_1 had the same value for n and t_2 had a different value.

When all the three tree objects being compared has a mapping to name n and all of them point to another tree object, but contain a different hash, then they might have a different name as their descendants or a different value for that key. To check this, we need to follow those hashes and explore their contents. Rule (MERGETREE(5)) shows this condition. When the recursive process returns to this level of the key, it will bring a modified tree t' as the return value if the conditions in the descendants follow one of the rules specified in MERGETREE.

Rule MERGETREE(6) shows the condition where a name n is found in the both tree objects t_1 and t_2 and both points to another tree object but with different hashes. Also, tl either points to a blob or does not contain the mapping for n at all. This means that both branch$_1$ and branch$_2$ has added a different key with name n, which was not already present in the LCA. n is added in a manner such that it is reachable from the head commits of both t_1 and t_2 following the same path. In this case, a new tree object will be formed which will contain all the names from top until n. n will have a mapping to the hash of a tree object which contains the descendants of n from both t_1 and t_2.

If all the three tree objects contain a name n, with label l_b but different hashes, then they contain a different value for the same key. This shows that there is a conflict among the branches and has to be resolved using the three-way merge mechanism. Hence the three values are passed to a user-defined function `merge_val`, which returns a final value based on application requirement. Rule (MERGETREE(7)) gives the semantics for this case. The returned value m is hashed, and their mapping is updated into block store. A tree entry with this mapping and the updated block store is returned.

If a name n is not present in the tree object t_1, but present in both tl and t_2 and has different hashes, then it means that t_1 has deleted a previous key in the store, and t_2 has changed it. Rule MERGETREE(8) describes this situation. In this case the a new tree object will be created by combining the effect of deletion at t_1 with modification at t_2. It can be done by recursively calling the `merge_tree` over the tree objects pointed by tl and t_2 and combining the resultant tree object with the result of other keys.

MergeTree(9) covers a situation where a name n is found in the tl and in t_2, but not in t_1. Moreover, in tl, n refers to a blob object and in t_2 it refers the tree object. This means that both t_1 and t_2 have modified the value after their previous common commit and t_1 has actually deleted that key. In this case, the value given in t_2 will be retained in the resultant tree object. It implicitly ensures that the effect of deletion at t_1 is also retained.

4.7 COMMUNICATION OPERATIONS

$$\textsc{Publish}\quad \frac{\text{merge}(\mathcal{B}, \mathcal{T}, \beta_L, \beta_P) = (\mathcal{B}', \mathcal{T}')}{\langle \mathcal{B}, \mathcal{T}\rangle \xrightarrow{\text{publish}(\beta_L, \beta_P)} \langle \mathcal{B}', \mathcal{T}'\rangle}$$

$$\textsc{Refresh}\quad \frac{\text{merge}(\mathcal{B}, \mathcal{T}, \beta_P, \beta_L) = (\mathcal{B}', \mathcal{T}')}{\langle \mathcal{B}, \mathcal{T}\rangle \xrightarrow{\text{refresh}(\beta_L, \beta_P)} \langle \mathcal{B}', \mathcal{T}'\rangle}$$

$$\textsc{RemoteRefresh}\quad \frac{\text{merge}(\mathcal{B}, \mathcal{T}, \beta_{ps}, \beta_{pt}) = (\mathcal{B}', \mathcal{T}')}{\langle \mathcal{B}, \mathcal{T}\rangle \xrightarrow{\text{remote\_refresh}(\beta_{ps}, \beta_{pt})} \langle \mathcal{B}', \mathcal{T}'\rangle}$$

Figure 4.14: Communication Operations

In order to share the updates among different sessions, Banyan provides some communication operations like `publish`, `refresh` and `remote_refresh`. Chapter 3 describes the working of these operations in detail. From semantics point of view, they are just a merge operation between two branches. We have two kinds of branches described in chapter 3, local and public branches. A publish operation merges the updates from the local branch β_L into the public branch β_P (rule Publish in figure 4.14). A refresh operation merges the updates in the public branch β_P into the local branch β_L (rule REFRESH). A remote refresh operation merges the updates between public branches β_{ps} and β_{pt} (rule REMOTEREFRESH).

4.8 CONNECT AND CLOSE OPERATIONS

$$\textsc{Connect}\quad \frac{\beta_L \notin domain(\mathcal{T})}{\langle \mathcal{B}, \mathcal{T}\rangle \xrightarrow{\text{connect}(\beta_L,\beta_P)} \langle \mathcal{B}, \mathcal{T}[\beta_L \mapsto \mathcal{T}(\beta_P)]\rangle}$$

$$\textsc{Close}\quad \frac{\text{merge}(\mathcal{B}, \mathcal{T}, \beta_L, \beta_P) = (\mathcal{B}', \mathcal{T}'[\beta_L \mapsto h])}{\langle \mathcal{B}, \mathcal{T}\rangle \xrightarrow{\text{close}(\beta_L,\beta_P)} \langle \mathcal{B}', \mathcal{T}'\rangle}$$

Figure 4.15: Session connect and close operations

Connect operation (rule CONNECT, figure 4.15) is used by new clients to create a session for themselves in Banyan. It takes the names of local branch (β_L) and a public branch (β_P) as the arguments. It checks whether there is an entry related to β_L in the tag store $\mathcal{T}$. If there is no such entry in the tag store, it means that no local branch with name β_L exists. Connect operation creates a new branch β_L with a mapping to the hash of the latest commit of the public branch β_P.

Close operation closes a session after securing its updates to its associated public branch. It first calls the merge method, which is analogous to publish operation here, and then removes the entry for the local branch β_L from the tag store (rule CLOSE). Chapter 3 describes the application level details of Connect and Close functions.

4.9 GARBAGE COLLECTION

While traditional database systems only store the most recent version of the data, Banyan necessitates that previous versions of the data must also be kept around for three-way merges. While persistence of prior versions (Driscoll *et al.*, 1986; Farinier *et al.*, 2015) is a useful property for audit and tamper evidence, Banyan API presented here does not provide a way to access earlier versions. It is only needed to compute the LCA for merge operation. Hence, if we can identify the objects that would not be needed in any LCA computation, we can delete those objects to free up the memory. We do not implement a full garbage collector (GC) in this work, but present a sketch about how garbage collection can be performed in Banyan. In this section we will see how we can identify the objects which can be garbage collected in Banyan, present operational semantics rules for the same and inductively proof its correctness.

4.9.1 Banyan Garbage Collector

Banyan GC is an extension of Git GC. Git is equipped with a GC that considers that any object in the block store that is reachable from the tag store is alive. Unreachable objects are to be deleted. Our aim is to assist the Git-like GC by pruning the history graph of nodes which will no longer be used. The key idea is that if a commit node will not be used for LCA computation, then that commit node may be deleted. Deleting commit nodes will leave dangling references from its referees, but Irmin can be extended to ignore dangling references to commit nodes. For individual sessions, once the session is closed, the corresponding entry in the tag store, and all the commits by that session may be deleted. In the execution history in figure 4.16, the commit node **`s0-c0`** may be deleted.

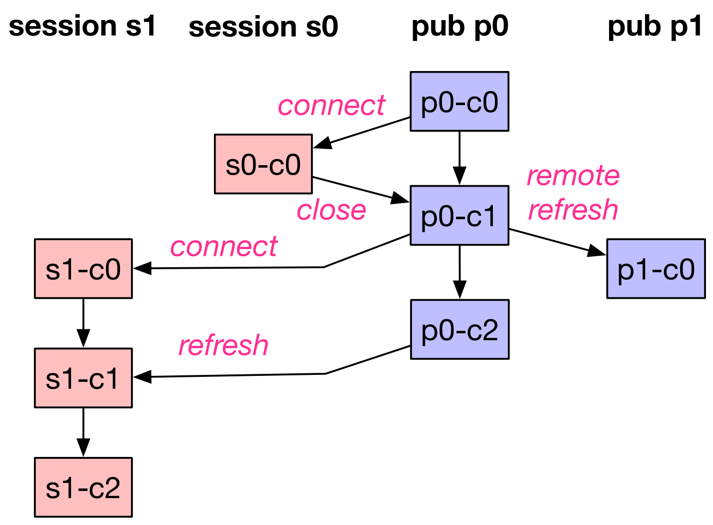


Figure 4.16: Garbage collection. Here, the commits `p0-c0` and `s0-c0` may be deleted.

The next question is when can commits on public branch be deleted. For each ongoing session in a replica, we maintain the latest commit in the public branch against which **`refresh`** was performed. The earliest of such commits in the public branch, and its descendants must be retained since they are necessary for three-way merge. For example, in figure 4.16, session **`s1`** refreshed against **`p0-c2`**, and **`s1`** is the only ongoing session. If **`s1`** publishes, then **`p0-c2`** will be the LCA commit. A similar reasoning is used for remote refreshes. When a commit in the public branch of a replica has been merged into the public branches of all the other replicas, then the ancestors of such commits will not be accessed and can be deleted. In figure 4.16, assume that we only have two replicas. Since **`p0-c1`** was merged by the public branch **`p1`**, **`p0-c1`** will be the LCA commit for subsequent remote refreshes by **`p1`**. Given that **`p0-c0`** is neither necessary for remote refreshes nor for ongoing sessions, **`p0-c0`** can be deleted.

We expect Banyan GC to save a significant amount of space compared to Banyan without GC. When there are no sessions in progress or when the updates to the database have stopped, the data maintained by Banyan would be comparable to any traditional databases, i.e. Banyan would only maintain the latest state of database. In particular, there would only be one latest commit, and associated tree and blob nodes. If there are ongoing sessions, then only the minimum information needed to perform refresh and publish operations in that session would be kept around after a GC. After the session is done, all the session local information would be GCed. This would significantly reduce the storage overhead of Banyan.

4.9.2 Operational Semantics of Banyan GC

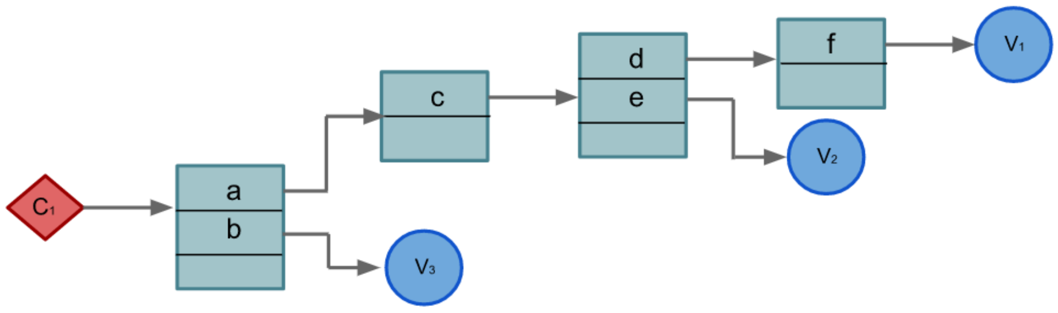


Figure 4.17: Reachable objects: Arrows show the objects that are connected with each other in the directional manner. Diamond is a commit, rectangle is a tree and circle is a blob object

Below are few rules described to help understand the GC semantics shown in figure 4.18:

ReachableObject: As we have discussed earlier, a tree object t is a collection of tree entries which has a mapping from a name n to a tuple which contains a hash to another object. Let us call this object a descendant of t. There can be more than one descendant for t. Similarly, a commit node also contains a hash for its associated tree object, which could be called as its descendant. An object ob is always reachable from itself (rule ReachableObject(1)). Also, another object ob' is reachable from ob, if it is reachable from one of the descendants of ob (rule ReachableObject(2)). This forms a recursive method which covers all the objects including blobs until the end. Figure 4.17 shows a commit and its associated objects. Table 4.3 shows the Banyan objects that are reachable from given objects in figure 4.17.

Table 4.3: List of reachable objects from a given objects in figure 4.17

Object	**Reachable Objects**
C_1	C_1, a, b, c, d, e, f, V_1, V_2, V_3
c	c, d, e, f, V_1, V_2
V_3	*No reachable objects*
b	b, V_3
d	d, f, V_1

RecursiveLca: It is used to find the LCA of the two commits. Section 4.5 describes the situation of concurrent merge where we need to recursively find the LCA of two commits until a single commit is found. Semantically, if we have a single commit object, then it is its own LCA (rule RecursiveLca(1)). LCA of two given commits can be computed as described in rule LCA. For a set of commits, we choose any two commits, compute their LCA (say C_l), and recursively compute the LCA of C_l and the next commit in the set (rule RecursiveLca(2)). It results in the lowest commit, which is common to all the commits combined.

Filter: `filter`($\mathcal{B}$, hs) filters the block store such that it returns a store which contains only hashes that are part of the list hs and their corresponding mappings. This means, if there is a hash h, such that $h \in hs$, then filter will retain h in the resultant store (rule FILTER(2)), otherwise h and its mapping will be removed from the resultant store (rule FILTER(3)). If the provided store is empty, then filter returns an empty store, regardless of hs (rule FILTER(1)).

REACHABLEOBJECT

$$ob = \text{rea\_obj}(\mathcal{B}, ob) \tag{1}$$

$$\frac{ob' \in \text{rea\_obj}(\mathcal{B}, \mathcal{B}(h))}{ob' \in \text{rea\_tree}(\mathcal{B}, t[n \mapsto (l, h)])} \tag{2}$$

RECURSIVELCA

$$c = \text{recursive\_lca}(\mathcal{B}, \{c\}) \tag{1}$$

$$\frac{c = \text{recursive\_lca}(\mathcal{B}, lca(\mathcal{B}, c_1, c_2) \cup cs)}{\text{recursive\_lca}(\mathcal{B}, \{c_1, c_2\} \cup cs) = c} \tag{2}$$

FILTER

$$\phi = \text{filter}(\phi, hs) \tag{1}$$

$$\frac{\mathcal{B}' = \text{filter}(\mathcal{B}, hs)}{\text{filter}(\mathcal{B}[h \mapsto v], \{h\} \cup hs) = \mathcal{B}'[h \mapsto v]} \tag{2}$$

$$\frac{h \notin hs}{\text{filter}(\mathcal{B}[h \mapsto v], hs) = \text{filter}(\mathcal{B}, hs)} \tag{3}$$

$$\text{GC}\ \frac{\begin{array}{c}\text{head\_commits} = \{c \mid h \in domain(\mathcal{T}) \wedge c = \mathcal{B}(h)\} \\ c = \text{recursive\_lca}(\mathcal{B}, \text{head\_commits}) \\ \text{live\_commits} = \{c\} \cup \text{des\_comm}(\mathcal{B}, c) \\ \text{live} = \text{live\_commits} \cup \{t \mid c \in \text{live\_commits} \wedge t \in \text{rea\_tree}(\mathcal{B}, c)\} \\ \text{live\_hashes} = \{h \mid l \in \text{live} \wedge h = \text{hash}(l)\} \\ \mathcal{B}' = \text{filter}(\mathcal{B}, \text{live\_hashes})\end{array}}{\langle \mathcal{B}, \mathcal{T} \rangle \xrightarrow{\text{gc}} \langle \mathcal{B}', \mathcal{T} \rangle}$$

Figure 4.18: Garbage Collection

Tag store contains the mapping of all the active branches to the hash of their latest commits or head commits. To perform any Banyan operation, we need this head commit. For example, a write operation would need the head commit to be noted as the parent of the new commit. A read operation would search for the key in the head commit. It is the merge related operations like publish, refresh, and remote refresh that will not only require the head commits but also the LCA of the two commits. To ensure that every merge operation is performed successfully even after GC, we need to ensure that the LCA of every pair of head commits is retained. Rule GC gives the semantics for garbage collection in Banyan. We identify all the head commits from the tag store and, using the `recursive_lca` method, identify the LCA c, of all the head commits. Every commit, which is a descendent of c, along with c itself, is considered to be a live commit and needs to be retained. Since GC is about deleting all the redundant objects and not just commit objects, we identify each object which is associated with the live commit. These are all the objects that are reachable from those commits. The set of such objects is called $live$ in the semantics. We find the hash of live objects ($live\_hashes$) and filter the block store such that it contains only the $live\_hashes$. GC operation returns back the modified block store ($\mathcal{B}'$). GC does not affect the tag store.

4.9.3 Proof of GC correctness

Since GC deletes some objects from the block store, we feel the need to prove that performing a GC would not affect the functionality of the Banyan. So, below we present an inductive proof of correctness for GC.

Definition 1 (Initial Store). *init_store = $\langle \mathcal{B}, \mathcal{T} \rangle$ such that $\forall \beta_P \in domain(\mathcal{T})$. $\mathcal{B}(\mathcal{T}(\beta_P)) = ([\,], h) \wedge \mathcal{B}(h) = \phi$. Intuitively, the initial store has a number of public branches β_P corresponding to the replicas, and all of the public branches point to the same initial commit $([\,], h)$ with no parent commits and whose tree is empty.*

Definition 2 (Well-formed Store). *A store $\langle \mathcal{B}, \mathcal{T} \rangle$ is said to be well-formed if there exists a sequence of requests r_1, r_2 ... r_n, such that, init_store $\xrightarrow{r_1} S_1 \xrightarrow{r_2} S_2$ ... $\xrightarrow{r_n} \langle \mathcal{B}, \mathcal{T} \rangle$.*

Definition 3 (Equivalent stores). *Given two stores $\langle \mathcal{B}_1, \mathcal{T}_1 \rangle$ and $\langle \mathcal{B}_2, \mathcal{T}_2 \rangle$ are said to be equivalent if $\forall \beta, k, b.\ \langle \mathcal{B}_1, \mathcal{T}_1 \rangle \xrightarrow{read(\beta,k,b)} \langle \mathcal{B}_1, \mathcal{T}_1 \rangle \Rightarrow \langle \mathcal{B}_2, \mathcal{T}_2 \rangle \xrightarrow{read(\beta,k,b)} \langle \mathcal{B}_2, \mathcal{T}_2 \rangle$. Intuitively, all the key-value pairs in every branch remains the same . We write $\langle \mathcal{B}_1, \mathcal{T}_1 \rangle \simeq \langle \mathcal{B}_2, \mathcal{T}_2 \rangle$.*

Lemma 1 (Unique Root). *Given a well-formed store $\langle \mathcal{B}, \mathcal{T} \rangle$, there exists a unique root commit c such that for all head_commits = $\{c \mid h \in domain(\mathcal{T}) \wedge c = \mathcal{B}(h)\}$, head_commits $\subseteq$ des_comm($\mathcal{B}$, c). Intuitively, all the head commits are descendants of the unique root commit.*

Proof. By induction on the definition of the well-formed store.

- **Base case:** The initial store by definition has a single commit, which is the unique root commit.
- **Inductive case:** Assume that $\langle \mathcal{B}, \mathcal{T} \rangle$ is well-formed. By induction hypothesis, it has a unique root. We need to show that for all r, $\langle \mathcal{B}, \mathcal{T} \rangle \xrightarrow{r} \langle \mathcal{B}', \mathcal{T}' \rangle$, $\langle \mathcal{B}', \mathcal{T}' \rangle$ has a unique root.
  - **Case Read:** does not change the store and hence, the unique root in $\langle \mathcal{B}', \mathcal{T}' \rangle$ is the same as $\langle \mathcal{B}, \mathcal{T} \rangle$.
  - **Case Write:** adds a new commit whose parent is one of the head commits. Hence, the unique node remains the same.
  - **Cases Publish, Refresh, Close, Remote Refresh:** all perform a merge. A merge operation adds a new commit whose parents are two head commits of the branch being merged. Hence, the root commit remains the same.

- **Case Connect:** does not change the block store, and hence, the root commit remains the same.
- **Case GC:** The GC procedure computes the recursive LCA of all of the head commits. There is a unique recursive LCA since $\langle \mathcal{B}, \mathcal{T} \rangle$ has a unique root. Let's call the unique recursive LCA as *l_gc*. GC only retains *l_gc* and its descendant commits. Hence, the unique root in $\langle \mathcal{B}', \mathcal{T}' \rangle$ will be *l_gc*.

$\square$

Lemma 2 (Merge GC)**.** *Given a well-formed store* $\langle \mathcal{B}, \mathcal{T} \rangle$*,* $\forall \beta_1, \beta_2 \in domain(\mathcal{T})$*, if merge(*$\mathcal{B}, \mathcal{T}, \beta_1, \beta_2$*) = (*$\mathcal{B}_1, \mathcal{T}_1$*), and* $\langle \mathcal{B}, \mathcal{T} \rangle \xrightarrow{gc} \langle \mathcal{B}_2, \mathcal{T}_2 \rangle$*,*
merge$(\mathcal{B}_2, \mathcal{T}_2, \beta_1, \beta_2) = (\mathcal{B}_3, \mathcal{T}_3)$ *then* $\langle \mathcal{B}_1, \mathcal{T}_1 \rangle \simeq \langle \mathcal{B}_3, \mathcal{T}_3 \rangle$

Proof. The merge function computes the LCA of the head commits pointed by the branches β_1 and β_2. If there are multiple LCAs then, a merge is computed recursively on the LCAs, which will in turn access the LCA of the previous LCAs. By Unique Root lemma, the commit history has a unique root. Hence, the recursive merge procedure will eventually find the unique recursive LCA *l_merge*. The merge will then access the tree and the blob nodes of any of the computed LCAs in this process.
The GC procedure computes the recursive LCA of all of the head commits of all the branches. This includes β_1 and β_2. Hence, the recursive LCA *l_merge* computed by merge on β_1 and β_2, will either be the recursive LCA computed by the GC *l_gc* or *l_merge* is a descendent of *l_gc*. The GC procedure retains *l_gc* and its descendent commits and the tree and the blob nodes of any of these commits. This set of nodes is a superset of the nodes accessed by merge.
Finally, the new nodes added by the merge will be the same in merge$(\mathcal{B}, \mathcal{T}, \beta_1, \beta_2)$ and merge$(\mathcal{B}_2, \mathcal{T}_2, \beta_1, \beta_2)$. Given that read will only access the head commits, and the corresponding tree and blob nodes, which will remain the same in $\langle \mathcal{B}_1, \mathcal{T}_1 \rangle$ and $\langle \mathcal{B}_3, \mathcal{T}_3 \rangle$, $\langle \mathcal{B}_1, \mathcal{T}_1 \rangle \simeq \langle \mathcal{B}_3, \mathcal{T}_3 \rangle$ □

Theorem 1 (GC Safety)**.** *Given a well-formed store* $\langle \mathcal{B}, \mathcal{T} \rangle$ *and a request r such that* $\langle \mathcal{B}, \mathcal{T} \rangle \xrightarrow{r} \langle \mathcal{B}_1, \mathcal{T}_1 \rangle$ *then* $\langle \mathcal{B}, \mathcal{T} \rangle \xrightarrow{gc} \langle \mathcal{B}', \mathcal{T}' \rangle \xrightarrow{r} \langle \mathcal{B}_2, \mathcal{T}_2 \rangle$*, then* $\langle \mathcal{B}_1, \mathcal{T}_1 \rangle \simeq \langle \mathcal{B}_2, \mathcal{T}_2 \rangle$*.*

Proof. By case analysis on the derivation of $S_1 \xrightarrow{r} S_2$.

Case 1 read

The read request only reads the latest commit and the tree associated with the commit. GC retains the latest commit, and its associated tree entry; the head commit is part of the live commits retained by the GC rule in figure 4.18. Hence, the two stores are equivalent.

Case 2 write

The write request only reads the latest commit and the tree associated with the commit. GC retains the latest commit, and its associated tree entry; the head commit is part of the live commits retained by the GC rule in figure 4.18.
A new commit is added in both the stores, whose parent is the latest commit and new tree entriesis added whose subtrees may be the subtrees from the latest commit. Hence, the two stores are equivalent.

Case 3 publish

The publish request reads the head commit and the tree associated with that commit both in the local branch β_L and the public branch β_P. It merges the tree entryof β_L into and the public branch β_P. It merges the tree node of β_L into the tree entryof β_P. By lemma Merge_gc, the two stores are equivalent.

Case 4 refresh

The refresh request reads the head commit and the tree associated with that commit both in the public branch β_P and the local branch β_L. It merges the tree entryof β_P into the tree entryof β_L. By lemma Merge_gc, the two stores are equivalent.

Case 5 remote_refresh

The remote refresh request reads the head commit and the tree associated with that commit in the two public branch β_{P1} and β_{P2}. It merges the tree node of β_{P1} into the tree entryof β_{P2}. By lemma Merge_gc, the two stores are equivalent.

Case 6 connect

The connect request takes a local branch name (β_L) and a public branch (β_P). It adds an entry into the tag store in which β_L maps to the hash of the latest commit of β_P. It does not change anything in the block store. GC retains the latest commit, and its associated tree entry; the head commit is part of the live commits retained by the GC rule in figure 4.18. Hence, the two stores are equivalent.

Case 7 close

The close request performs a merge and then removes an entry from the tag store. For merge, it reads the head commit and the tree associated with that commit both in the local branch β_L and the public branch β_P. It merges the tree node of β_L into the tree entryof β_P. By lemma Merge_gc, the two stores are equivalent. Later, in both the stores, it removes an entry related to β_L from the tag store, hence retaining the equivalence of the two stores.

Case 8 gc

A gc request identifies the head commit of all the branches and secures the set of all the live commits associated with those head commits as shown in figure 4.18. Since the gc request itself does not change the head commit for any branch, another run for the gc request would result in the same set of live commits. Hence, a store with single gc request and one with two consecutive gc requests would result into equivalent stores.

□

CHAPTER 5

IMPLEMENTATION

In this chapter, we describe the instantiation of Banyan on Cassandra (Cassandra, 2021), a popular, industrial-strength, column-oriented, distributed database. Cassandra offers eventual consistency with the last-write-wins (LWW) conflict resolution policy. Cassandra also offers complex data types such as list, set and map, with baked-in conflict resolution policies. Given the richness of replicated data types, the available complex data types are quite limiting. In addition to storing the values in different formats like complex data types, replicated data types offers in-built conflict resolution techniques. Cassandra also offers lightweight transactions (distributed compare-and-update) implemented using the Paxos consensus protocol (Lamport, 2001). Lightweight transactions are also limited to operate on only one object. Banyan does not use lightweight transactions since their cost is prohibitively high due to consensus in a geo-distributed setting. As mentioned previously Banyan only requires sticky availability and so uses a replica-local lock for ensuring mutual exclusion when multiple sessions try to update the public branch on a replica concurrently.

By instantiating Banyan on Cassandra, we offload the concerns of replication, fault tolerance, availability, and convergence to the backing store. On top of Cassandra, Banyan uses Irmin (Irmin, 2021), an OCaml library for persistent stores with built-in branching, merging, and reverting facilities. Irmin can be configured to use different storage backends, and in our case, the storage is Cassandra. Importantly, Cassandra being a distributed database, serves the purpose of the networking layer in addition to persistent storage. While Irmin permits arbitrary branching and merging, Banyan is a specific workflow on top of Irmin which retains high availability.

5.1 IRMIN DATA MODEL

The expressivity of Irmin imposes a significant burden on the underlying storage. For efficiently storing different versions of the state as the store evolves, Irmin uses the Git object model. In chapter 4, we had seen the different kinds of objects that Banyan (and also Irmin and Git) creates and how these objects are used to implement a version controlled store. Figure 5.1 shows a snapshot of the state of the Irmin store.

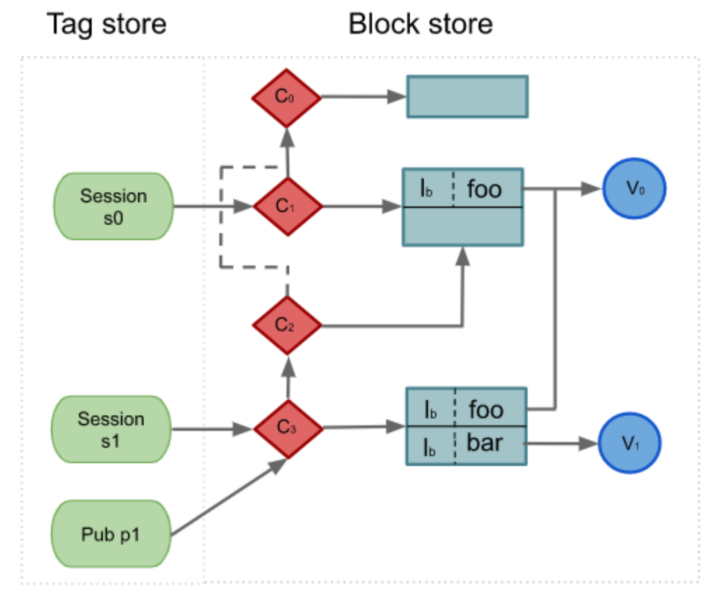


Figure 5.1: A sample Irmin store. The rounded rectangles are tags, diamonds are commit objects, rectangles are tree object, and circles are blob objects.

As described earlier, the tag store records the branch and the commit that corresponds to that branch. In this example, we have three branches, `Session s0`, `Session s1` and `Pub p1`. The content-addressable block store contains the commit, tree, and blob objects reachable from those branches. A commit object representing a commit may have a single reference to a tree object. A commit node would have one or more parent commits, except for the initial commit which has no parents. Here, the commit `c0` is the initial commit which does not have any parent commit. It points to an empty tree node. The commit `c1` is associated with `Session s0` which contains a single key `["foo"]` and value `V0`. The commit `c0` is the parent of commit `c1` showing that `Session s0` is forked out of the initial commit. The commit `c3` is the latest commit of `Session s1` with `c2` as its parent. Like commit `c1`, `c2` also has a single key `["foo"]` and value `V0`. Both the commits share the same objects as the block store is content addressable. `Session s1` adds a new key `["bar"]` and value `V1` which results in the creation of commit `C3`. A new tree object is created for commit `C3` as the block store is immutable and hence we can not edit the existing tree object. `Session s1` publishes its content to another branch `Pub p1`. As a result, `Pub p1` at this point would have the exact same content as `Session s1`. Hence, it shares the commit node with `Session s1`, instead of creating new objects for itself. As we have seen previously in chapter 4, a single Banyan write translates to multiple writes to the underlying store. This is necessary to maintain the historical state of the store which is needed for the three-way merge.

5.2 CASSANDRA INSTANTIATION

For instantiating Banyan on Cassandra, we use two tables, one for the tag store and another for the block store. In the tag store a tag is mapped to a commit hash. It uses `Cassandra String typed value` for tag and a `Cassandra blob typed value` for the commit hash. In the block store, a hash is mapped with the different Banyan objects representing the contents of the store. It uses **`Cassandra blob typed value`** for both hash and the corresponding content in the store. Irmin handles the logic necessary to serialize and deserialize the various Git objects into binary **`blob`**s and back.

Cassandra replicates the writes to the tag and block tables asynchronously amongst the replicas. Each replica periodically merges the public branches of other replicas into its public branch to fetch remote updates. Due to eventual consistency of Cassandra, it may be the case that not all the objects from a remote replica are available locally. For example, the merge function may find a new commit from a remote replica, but the tree object referenced by a commit object may not available locally. In this situation, Banyan simply skips merging this branch in this round. Cassandra ensures that eventually the remote tree object will arrive at this replica and will be merged in a subsequent remote refresh operation. Thus, fetching remote updates is a non-blocking operation.

In Irmin, the tag store is updated with a compare-and-swap to ensure that concurrent updates to the same tag should be disallowed. This is necessary to ensure that Banyan publish operation which writes to the public branch on a replica does not lose updates. Naively implementing this in Cassandra would necessitate the use of lightweight transactions and suffer prohibitive costs. By restricting the Banyan programming model (Chapter 3) such that entries in the tag store, in particular, the tag corresponding to the public branch of the replica is only updated on that replica, we remove the necessity for lightweight transactions. Thus, we don't depend on any special features of Cassandra to realise the Banyan model, and Banyan can be instantiated on *any* eventually consistent key-value store.

5.3 RECURSIVE MERGES

A particular challenge in making Banyan scalable is the problem of recursive merges. Consider a simple mergeable counter MRDT, whose implementation is:

```
let merge lca v1 v2 =
  let old = match lca with
  | None -> 0
  | Some v -> v
  in
  v1 + v2 - old
```

Consider the execution history presented in Figure 5.2 which shows the evolution of a single counter. The history only shows the interaction between two replicas, and does not show any sessions. Each node in the history is a commit. Since we want to focus on a single counter, for simplicity, we ignore the tree nodes and the node labels show the counter value. Initially the counters are **`0`**, and each replica concurrently increments

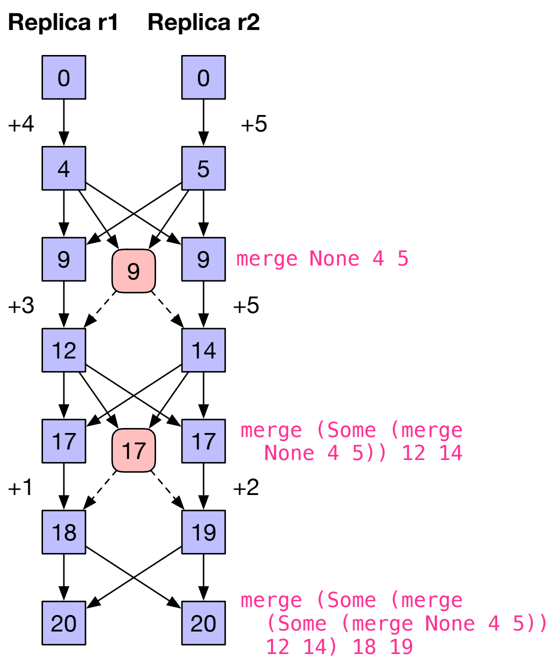

Replica r1
Replica r2
0
0
+4
+5
4
5
9
9
9
merge None 4 5
+3
+5
12
14
17
17
17
merge (Some (merge None 4 5)) 12 14
+1
+2
18
19
20
20
merge (Some (merge (Some (merge None 4 5)) 12 14) 18 19

Figure 5.2: Recursive merge. Rounded rectangles are the results of recursive merges.

the counter by `4` and `5`. When the replicas perform remote refreshes, they invoke `merge None 4 5` to resolve the conflict updates yielding `9`. The LCA is `None` since there is no common ancestor.

Subsequently, the replicas increment the counters by `3` and `5`. Now, consider that the replicas merge each other's branches. When merging `12` and `14`, there are two equally valid LCAs `4` and `5`. Picking either one of them leads to incorrect result. At this point, Irmin merges the two LCAs using `merge None 4 5` to yield `9`, which is used as the LCA for merging `12` and `14`. This yields the value `17`. The result of merging the LCAs is represented as a rounded rectangle. Importantly, the result of the recursive merge `9` is not a parent commit of `12` and `14` (distinguished by the use of dotted arrows). This is because the commit nodes are stored in the content-addressed store, and adding a new parent to the commit node would create a distinct node, whose hash is different from the original node. Any other nodes that referenced the original commit node will continue to reference the old node. As a result, the recursive merges will need to be performed again for subsequent requests!

Consider that the replicas further evolve by incrementing `1` and `2`, yielding `18` and `19`. When these commits are merged on remote refresh, there are two LCAs `12` and `14`, which need to be merged. This in turn has two LCAs `4` and `5`, which need to be merged. Thus, every subsequent recursive merge, which is very likely since the replicas merge each other's branches, requires repeating all the previous recursive merges. This does not scale.

We solve this problem by having a separate table in Cassandra that acts as a cache, recording the result of LCA merges. Whenever Banyan encounters a recursive merge, the cache is first consulted before performing the merge. In this example, when **18** and **19** are being merged, Banyan first checks whether the two LCAs **12** and **14** are in the cache. They would not be. This triggers a recursive merge of LCAs **4** and **5**, whose result is in the cache, and is reused. The cache is also updated with an entry that records that the merge of the LCAs **12** and **14** is the commit corresponding to **17**.

CHAPTER 6
EVALUATION

In this section, we evaluate the performance of Banyan instantiation on Cassandra. Our goal is to assess the suitability of Banyan for programming loosely connected distributed applications. To this end, we first quantify the overheads of implementing Banyan over Cassandra. Subsequently, we assess the performance of MRDTs implemented using Banyan. And finally, we study the performance of distributed build cache (Chapter 2).

6.1 EXPERIMENTAL SETUP

For the experiments, we use a Cassandra cluster with 4 nodes within the same data center. Each Cassandra node runs on a baremetal Intel®Xeon®E3-1240 CPU, with 4 physical cores, and 2 hardware threads per core. Each core runs at 3.70GHz and has 128KB of L1 data cache, 128KB of L1 instruction cache, 1MB L2 cache and 8MB of L3 cache. Each machine has 32GB of main memory. The machines are unloaded except for the Cassandra node. The ping latency between the machines is 0.5ms on average. The clients are run on a machine with the same configuration in the same data center.

For the experiments, Cassandra cluster is configured with a replication factor of 1, read and write consistency levels of `ONE`. Hence, the cluster maintains a single copy of each data item, and only waits for one of the servers to respond to return the result of read and write to the client. These choices lead to eventual consistency where the reads may not return the latest write. The cluster may be configured with larger replication factor for better fault tolerance. However, stronger consistency levels are not useful since Banyan enforces per-key causal consistency over the underlying eventual consistency offered by Cassandra. In fact, choosing strong consistency for reads and writes in Cassandra does not offer strong consistency in Banyan since the visibility of updates in Banyan is explicitly controlled with the use of **`refresh`** and **`publish`**.

6.2 BASELINE OVERHEADS

Given that Banyan has to persist every version of the store, what is the impact of Banyan when compared to using Cassandra in a scenario where Cassandra would be sufficient? We measure the throughput of performing 32k operations, with 80% reads and 20% writes with different numbers of clients. The keys and values are 8 and 128 byte strings, respectively. For Banyan, we use last-writer-wins resolution policy, which is the policy used by Cassandra. The results are presented in Figure 6.1.

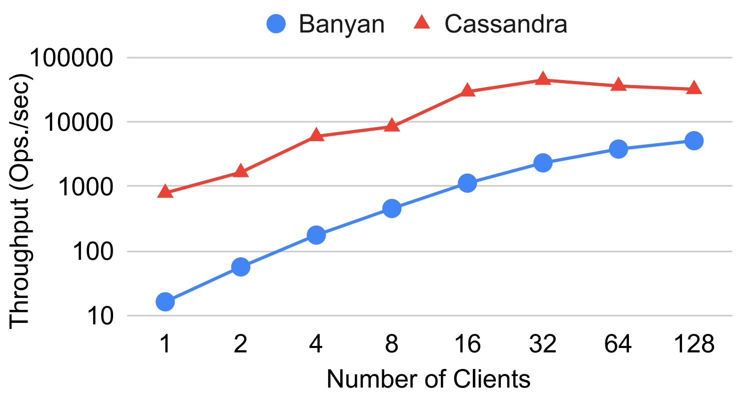


Figure 6.1: Performance comparison between Banyan and Cassandra on LWW string value.

With 1 client, Banyan performs 16 operations per second, while Cassandra performs 795 operations per second. Cassandra offers 50× more throughput than Banyan with 1 client. This is due to the fact that every read (write) performs 4 reads (3 reads and 4 writes) to the underlying store to create and access the tag, commit and tree nodes. Banyan additionally includes marshalling and hashing overheads for accessing the content-addressed block store. Cassandra does not include any of these overheads. Luckily, Banyan overheads are local to a client, and hence, can be easily parallelized. With 1 client, the cluster is severely under utilized, and the client overheads dominate. With increasing number of clients, the cluster is better utilized. At 128 clients, Cassandra performs 31274 operations per second where as Banyan performs 5131 operations per second, which is a slowdown of 6.2×. We believe that these are reasonable overheads given the stronger consistency and isolation guarantees, and better programming model offered by Banyan.

At the end of 32k operations, Cassandra uses 4.9MB of disk space, while Banyan uses 1.8GB of disk space. With the use of GC whose sketch and operational semantics was presented in Chapter 4 this space usage will come down significantly.

6.3 MERGEABLE TYPES

Counter We begin with the counter data type discussed in Section 5.3. How does Banyan counter perform when concurrently updated by multiple clients? For the experiment, the value type is a counter that supports increment, decrement and read operations. The clients perform 32k increment or decrement operations on a key randomly selected from a small key space. Each client refreshes and publishes after every 100 operations. By choosing a small key space, we aim to study the scalability of the system with large number of conflicts.

Figure 6.2 shows the performance result for two key spaces of size 1024 and 4096 keys. With 1 client, there are no conflicts. The conflicts increases with increasing number of clients. We get a peak throughput of 1814 (2027) operations per second with a key space of 1024 (4096) keys. Observe that the number of conflicts is considerably lower with 4096 keys when compared to 1024 keys. As a result, the throughput is higher with 4096 keys. The result shows that the throughput of the system is proportional to the number of conflicting operations.

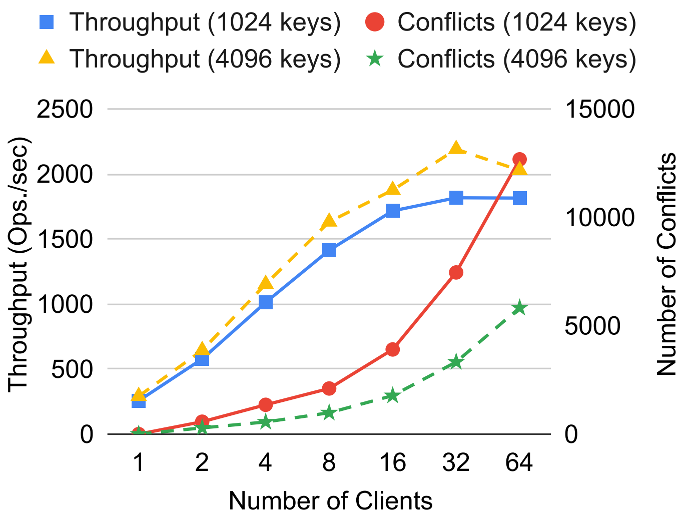


Figure 6.2: Performance of counter MRDT.

Blob log Another useful class of MRDTs are *mergeable logs*, where each log message is a string. Such a distributed log is useful for collecting logs in a distributed system, and examining the logs in their global time order. To this end, each log entry is a pair of timestamp and message, and the log itself is a list of such entries in reverse chronological order. The merge function for the mergeable log extracts the newer log entries from both the versions, sorts the newer entries in reverse chronological order and returns the list obtained by appending the sorted newer entries to the front of the log at the LCA.

While this implementation is simple, it does not scale well. In particular, each commit stores the entire log as a single serialized blob. This does not take advantage of the fact that every commit can share the tail of the log with its predecessor. Moreover, every append to the log needs to deserialize the entire log, append the new entry and serialize the log again. Hence, append is an $O(n)$ operation, where n is the size of the log. Merges are also worst case $O(n)$. This is undesirable. We call this implementation a *blob log*.

Linked log We can implement a efficient logs by taking advantage of the fact that every commit shares the tail of the log with its predecessor. The value type in this log is:

```
type value =
  | L of float (* timestamp *) * string (* message *)
       * blob (* hash of prev value *)
  | M of blob list (* hashes of the values being merged *)
```

The value is either a log entry **`L(t,m,h)`** with timestamp **`t`**, message **`m`** and a hash of the previous value **`h`**. Appending to the log only needs to add a new object that refers to the previous log value. Hence, append is $O(1)$. Figure 6.3 shows a snapshot of the log assuming a single key **`x`**. The log at **`x`** in the public branch **`p0`** (session **`s0`**) is **`[a;b;c]`** (**`[a;b;d]`**). The merge operation simply adds a new value **`M [h1;h2]`**, which refers to the hashes of the two log values being merged. This operation is also $O(1)$. The read function for the log does the heavy-lifting of reading the log in reverse chronological order.

Observe that unlike the examples seen so far where the values do not refer to other values, this *linked log* implementation refers to other values as heap data structures would do. Figure 6.4 shows the time taken to add 100 additional messages to the log with 4 clients.

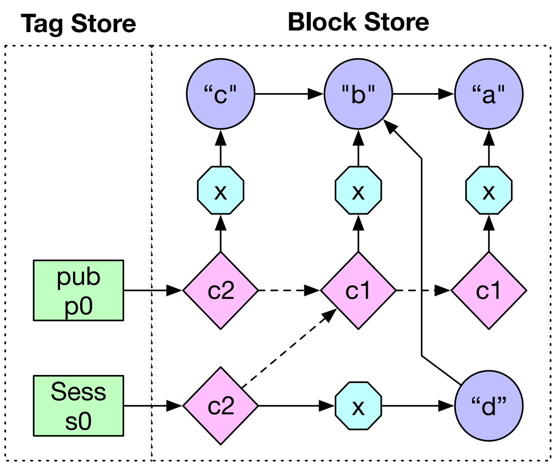


Figure 6.3: A snapshot of linked log storage.

Observe that the time stays constant with linked log but increases linearly with blob log. By being able to share objects across different commits (versions), Banyan leads to efficient implementations of useful data structures.

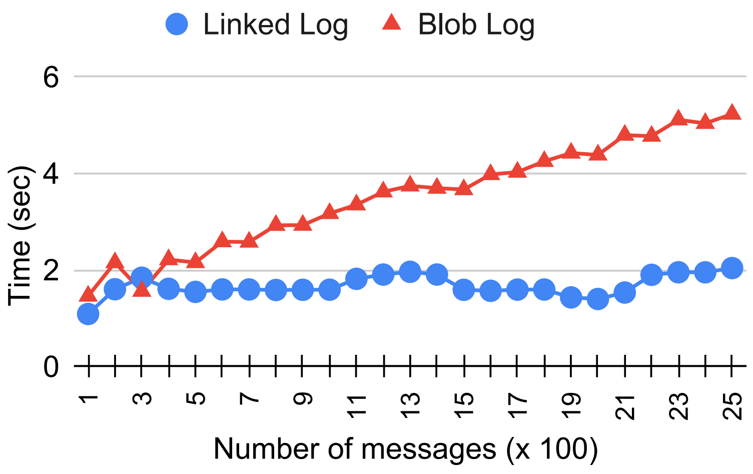


Figure 6.4: Performance of mergeable logs.

6.4 DISTRIBUTED BUILD CACHE

In this section, we evaluate the performance of distributed build cache described in Chapter 2. We have chosen three OCaml packages **`git`**, **`irmin`** and **`httpaf`** with common dependent packages. In the first experiment, we measure the benefit of building a package that has already been built in another workspace. Hence, the package artefacts will already be in the build cache.

For each library, we measure the baseline build time (1) without using the build cache, (2) using an empty build cache, and (3) building the same package on a machine with the same package having built earlier on a different machine. Figure 6.5 shows the results. We see that case using an empty build cache is slower than not using the cache since the artefacts are stored in the cache. We also see that building the same package on a different machine is faster due to the build cache when compared to the baseline.

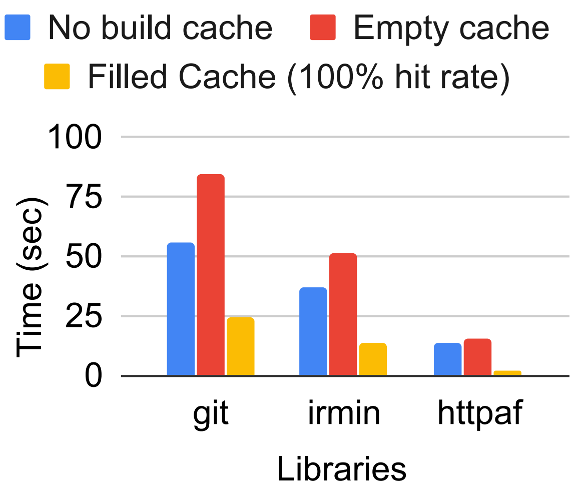


Figure 6.5: Performance of complete reuse of build artefacts.

A more realistic scenario is partial sharing of artefacts, where some of the dependencies are in the cache and other need to be build locally, and added to the cache. In this experiment, git package is first build on a machine with an empty cache. Subsequently,

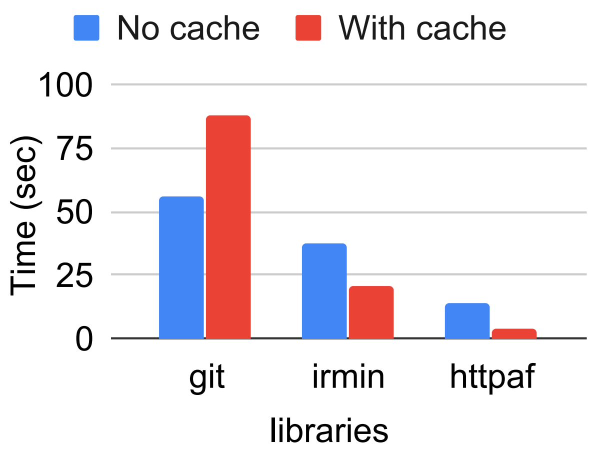


Figure 6.6: Performance of partial reuse of build artefacts.

`irmin` package is built on a second machine (which will now benefit from the common artefacts in the cache). And finally, building **`httpaf`** on a third machine, which benefits from both of the builds. Figure 6.6 shows the results. As expected, the **`git`** package build is slower with cache than without since the cache is empty and the artefacts need to be written to the cache additionally. Subsequent package builds benefit from partial sharing of build artefacts. The results illustrate that Banyan not only makes it easy to build complex applications like distributed build caches, but the implementation also performs well under realistic workloads.

CHAPTER 7

RELATED WORK

Several prior works have addressed the challenge of balancing the programmability and performance under eventual consistency. RedBlue consistency (Li *et al.*, 2012) offers causal consistency by default (blue), but operations that require strong consistency (red) are executed in single total order. Quelea (Sivaramakrishnan *et al.*, 2015) and MixT (Milano and Myers, 2018) offer automated analysis for classifying and executing operations at different consistency levels embedded in weakly isolated transactions, paying the cost of proportional to the consistency level. Indeed, mixing weaker consistency and transactions has been well-studied (Brutschy *et al.*, 2017; Kraska *et al.*, 2013; CosmosDB, 2021).

Banyan only supports causal consistency, but it is known to be the strongest consistency level that remains available (Lloyd *et al.*, 2011). While prior works attempt to reconcile traditional isolation levels with weak consistency, Banyan leaves the choice of reading and writing updates to and from other transactions to the client through the use of **`publish`** and **`refresh`**. We believe that traditional database isolation levels are already quite difficult to get right (Kaki *et al.*, 2017), and attempting to provide a fixed set of poorly understood isolation levels under weak consistency will lead to proliferation of bugs.

Banyan is distinguished by the equipping data types with the ability to handle conflicts (three-way merge functions). Banyan builds on top of Irmin (Irmin, 2021). Irmin allows arbitrary branching and merging between different branches at the cost of having to expose the branch name. Banyan refreshes and publishes implicitly to the public branch at a repository, which obviates the need for naming branches explicitly. Irmin does not include a distribution and convergence layer; Banyan uses Cassandra for this purpose. Banyan provides causal consistency and coordination free transactions over weakly consistent Cassandra. Several prior work have similarly obtained stronger guarantees on top of weaker stores (Sivaramakrishnan *et al.*, 2015; Bailis *et al.*, 2013*b*).

TARDiS (Crooks *et al.*, 2016) supports user-defined data types, and a transaction model similar to Banyan. TARDiS is however a machine model that exposes the details of explicit branches and merges to the developer, whereas Banyan is a programming model that can be instantiated on any eventually consistent key-value store. For instance, in TARDiS programmers need to invoke a separate merge transaction that does an n-way merge. Banyan transaction model is more flexible than TARDiS. For example, Banyan can support monotonic atomic view, which TARDiS cannot – TARDiS transactions do not have a way of allowing more recent updates since the transaction began. TARDiS does not discuss merges without LCAs or the issue with recursive merges. We found recursive merges to be a very common occurrence in practice.

Concurrent revisions (Burckhardt *et al.*, 2012) describe a programming model with branch and merge workflow with explicit branches and restrictions on the shape of history graphs. Banyan makes the choice of branches to publish and refresh implicit leading to a simpler model. Concurrent revisions does not include an implementation. Antidote SQL (Lopes *et al.*, 2019) is a database system for geo-distributed applications that provides the user the ability to relax SQL consistency when possible, but remain strict when necessary. Similar to Banyan, Antidote SQL transactions are executed over replicated data types. While Antidote SQL only permits parallel snapshot isolation level (Sovran *et al.*, 2011), by making **`refresh`** and **`publish`** explicit, Banyan permits weaker isolation levels such as monotonic atomic view (Bailis *et al.*, 2013*a*).

In this work, we also present a formal operational semnatics for Banyan over a core Git-like store. To our knowledge, our semantics is the first formal description of the semantics of a Git-like store equipped with three-way merges

CHAPTER 8

LIMITATIONS AND FUTURE WORK

Many eventually consistent databases such as CosmosDB (CosmosDB, 2021), DynamoDB (DynamoDB, 2021) and Cassandra provide tunable consistency levels for operations ranging from eventual consistency to strong consistency. Banyan only provides causal consistency, which is known to be the strongest available consistency level, but does not provide weaker or strong consistency levels. As such applications that require strong consistency, such as bank accounts with a minimum balance requirement, cannot be expressed in Banyan. We believe that we can extend Banyan with strongly consistent operations. However, operations with weaker consistency (and presumably better performance) cannot be incorporated in Banyan due to the underlying expectation about the *causal history* for each operation.

We have yet to implement the garbage collector for Banyan based on the design sketched in Section 4.9. In the absence of a garbage collector, the storage requirements are quite significant compared to traditional databases which only store the most recent version of the data (Section 6.2). We leave the implementation of the garbage collector for future work.

CHAPTER 9

CONCLUSION

Programming loosely connected distributed application is challenging. CAP and PACELC theorems suggest that there is always a trade-off between consistency and availability with distributed systems. If we need to design a highly available system, we must expect weak consistency among the replicas. Eventually consistent databases are used to design applications in such scenarios. Eventually consistent databases are designed assuming that there will always be a divergent view of data among different replicas. Hence it provides means to converge the replicas eventually. Current solutions based on Convergent Replicated data types such as Last-Writer-Wins, and multi-valued data structures provide very few data types to work with. Moreover, they support only the primitive types. That becomes a hindrance in developing complex applications over eventually consistent databases. In this work, we present Banyan, which is based on Mergeable Replicated Data types and uses the principles of Git to allow coordination-free distributed transactions. Like Git, the data type in Banyan is equipped with three-way merge capabilities. It records the entire history of updates and finds the lowest common ancestor between two replicas to track the diversion between them. It uses this information to merge these replicas based on rules defined by the application. Banyan is a programming model which is instantiated on top of an off-the-shelf eventually consistent distributed database. A distributed database provides not only storage capabilities but also a distribution network. This makes it easy to develop distributed applications without worrying about complex network protocols. Banyan provides a way to perform isolated transactions to each client. It provides methods like **`publish`** **`refresh`** and **`remote`** `refresh` to share the updates among the replicas and other clients. With extensive evaluation, we show that Banyan helps build complex distributed applications without compromising the performance. Banyan uses a large amount of space as it stores the entire history of updates. It uses this history to compute the lowest common ancestor (LCA) for merge-related operations. We propose a sketch for a garbage collector that identifies the objects that would not be needed by Banyan for the computation of LCA and could be deleted from the store. Garbage collection would help in significantly reducing the space usage of Banyan. We also present a formal operational semantics for Banyan over a core Git-like store. To our knowledge, our semantics is the first formal description of the semantics of a Git-like store equipped with three-way merges. We prove the correctness of the novel form of garbage collection using our semantics. Thanks to our semantics, we have also discovered bugs in the merge semantics of Irmin, a widely used distributed database built on the principles of Git.

REFERENCES

1. **Abadi, D.** (2012). Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, **45**(2), 37–42. ISSN 0018-9162.

2. **Bailis, P.**, **A. Davidson**, **A. Fekete**, **A. Ghodsi**, **J. M. Hellerstein**, and **I. Stoica** (2013*a*). Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.*, **7**(3), 181–192. ISSN 2150-8097.

3. **Bailis, P.**, **A. Ghodsi**, **J. M. Hellerstein**, and **I. Stoica**, Bolt-on Causal Consistency. *In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13. 2013*b*. ISBN 9781450320375.

4. **Bazel**, Bazel: A fast, scalable, multi-language build system. 2021. URL `https://bazel.build/`.

5. **Berenson, H.**, **P. Bernstein**, **J. Gray**, **J. Melton**, **E. O'Neil**, and **P. O'Neil** (1995). A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.*, **24**(2), 1–10. ISSN 0163-5808.

6. **Brutschy, L.**, **D. Dimitrov**, **P. Müller**, and **M. Vechev**, Serializability for Eventual Consistency: Criterion, Analysis, and Applications. *In Proceedings of the 44th ACM SIGPLAN Symposium on POPL*, POPL 2017. 2017. ISBN 9781450346603.

7. **Burckhardt, S.**, **A. Gotsman**, **H. Yang**, and **M. Zawirski** (2014). Replicated Data Types: Specification, Verification, Optimality. *SIGPLAN Not.*, **49**(1), 271–284. ISSN 0362-1340.

8. **Burckhardt, S.**, **D. Leijen**, **M. Fähndrich**, and **M. Sagiv**, Eventually Consistent Transactions. *In ESOP 2012*, ESOP'12. 2012. ISBN 9783642288685.

9. **Cassandra, A.**, Apache Cassandra: The right choice when you need scalability and high availability without compromising performance. 2021. URL `https://cassandra.apache.org/`.

10. **CosmosDB, A.**, Azure CosmosDB: Build or modernise scalable, high-performance apps. 2021. URL `https://azure.microsoft.com/en-in/services/cosmos-db/`.

11. **Crain, T.** and **M. Shapiro**, Designing a Causally Consistent Protocol for Geo-Distributed Partial Replication. *In Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15. 2015. ISBN 9781450335379.

12. **Crooks, N.**, **Y. Pu**, **N. Estrada**, **T. Gupta**, **L. Alvisi**, and **A. Clement**, TARDiS: A Branch-and-Merge Approach To Weak Consistency. *In Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16. 2016. ISBN 9781450335317.

13. **Driscoll, J. R.**, **N. Sarnak**, **D. D. Sleator**, and **R. E. Tarjan**, Making Data Structures Persistent. *In Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86. 1986. ISBN 0897911938.

14. **DynamoDB, A.**, Amazon DynamoDB: Fast and flexible NoSQL database service for any scale. 2021. URL `https://aws.amazon.com/dynamodb/`.

15. **Farinier, B.**, **T. Gazagnaire**, and **A. Madhavapeddy**, Mergeable Persistent Data Structures. *In Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*. 2015.

16. **Gilbert, S.** and **N. Lynch** (2002). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News*, **33**(2), 51–59. ISSN 0163-5700.

17. **Git**, Git: A distributed version control system. 2021. URL `https://git-scm.com/`.

18. **GitOjbects**, Git Objects. 2021. URL `https://git-scm.com/book/en/v2/Git-Internals-Git-Objects`.

19. **Gradle**, Gradle: An open-source build automation tool. 2021. URL `https://gradle.org/`.

20. **Irmin**, Irmin: A distributed database built on the principles of Git. 2021. URL `https://irmin.org/`.

21. **Kaki, G.**, **K. Nagar**, **M. Najafzadeh**, and **S. Jagannathan** (2017). Alone Together: Compositional Reasoning and Inference for Weak Isolation. *Proc. ACM Program. Lang.*, **2**(POPL).

22. **Kaki, G.**, **S. Priya**, **K. Sivaramakrishnan**, and **S. Jagannathan** (2019). Mergeable Replicated Data Types. *Proc. ACM Program. Lang.*, **3**(OOPSLA).

23. **Kermarrec, A.-M.** and **M. van Steen** (2007). Gossiping in Distributed Systems. *SIGOPS Oper. Syst. Rev.*, **41**(5), 2–7. ISSN 0163-5980.

24. **Kraska, T.**, **G. Pang**, **M. J. Franklin**, **S. Madden**, and **A. Fekete**, MDCC: Multi-Data Center Consistency. *In Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13. 2013. ISBN 9781450319942.

25. **Lakshman, A.** and **P. Malik** (2010). Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, **44**(2), 35–40. ISSN 0163-5980.

26. **Lamport, L.** (2001). Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, 51–58. URL `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

27. **Lamport, L.**, Time, clocks, and the ordering of events in a distributed system. *In Concurrency: the Works of Leslie Lamport*. 2019, 179–196.

28. **Li, C.**, **D. Porto**, **A. Clement**, **J. Gehrke**, **N. Preguiça**, and **R. Rodrigues**, Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. *In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12. 2012. ISBN 9781931971966.

29. **Lloyd, W.**, **M. J. Freedman**, **M. Kaminsky**, and **D. G. Andersen**, Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. *In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11. 2011. ISBN 9781450309776.

30. **Lopes, P.**, **J. a o Sousa**, **V. Balegas**, **C. Ferreira**, **S. 'e rgio Duarte**, **A. Bieniusa**, **R. Rodrigues**, and **N. M. P. c c a** (2019). Antidote SQL: Relaxed When Possible, Strict When Necessary. *CoRR*, **abs / 1902.03576**. URL `http://arxiv.org/abs/1902.03576`.

31. **Milano, M.** and **A. C. Myers**, MixT: A Language for Mixing Consistency in Geodistributed Transactions. *In Proceedings of the 39th ACM SIGPLAN Conference on PLDI*. 2018. ISBN 9781450356985.

32. **Riak**, Riak: Enterprise NoSQL Database. 2021. URL `https://riak.com/`.

33. **Shapiro, M.**, **N. Preguiça**, **C. Baquero**, and **M. Zawirski**, Conflict-free Replicated Data Types. *In Symposium on Self-Stabilizing Systems*. Springer, 2011.

34. **Sivaramakrishnan, K.**, **G. Kaki**, and **S. Jagannathan**, Declarative Programming over Eventually Consistent Data Stores. *In Proceedings of the 36th ACM SIGPLAN Conference on PLDI*. 2015. ISBN 9781450334686.

35. **Sovran, Y.**, **R. Power**, **M. K. Aguilera**, and **J. Li**, Transactional Storage for Geo-Replicated Systems. *In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11. 2011. ISBN 9781450309776.

36. **Viotti, P.** and **M. Vukolić** (2016). Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.*, **49**(1). ISSN 0360-0300.

CURRICULUM VITAE

1.	**NAME**	:	Shashank Shekhar Dubey
2.	**DATE OF BIRTH**	:	27 July 1993
3.	**EDUCATIONAL QUALIFICATIONS**		

Bachelor of Technology (B.Tech)

Year of Graduation	**:**	2014
Institution	**:**	KIIT University
Specialization	**:**	Information Technology

Master of Science (M.S.)

Year of Graduation	**:**	2021
Institution	**:**	IIT Madras
Specialization	**:**	Computer Science & Engineering

GTC Committee

CHAIRPERSON:	Prof. Madhu Mutyam Professor Computer Science & Engineering
GUIDE(S):	Dr. KC Sivaramakrishnan Assistant Professor Computer Science & Engineering
MEMBERS:	Dr. Yadu Vasudev Assistant Professor Computer Science & Engineering
	Prof. Vinita Vasudevan Professor Electrical Engineering