Benchmark Merkle Tree models on cloud environment for verifiable data storage

Dang Tran, Nakash Kumar, and Akefar Islam

Memorial University of Newfoundland, St. John's, NL, Canada

Abstract. Verifiable data storage is essential for maintaining integrity and security in cloud systems. Merkle Trees, a technique from the blockchain area, provides an efficient way to verify data integrity continuously in zero-trust networks, but different implementations can vary in performance. This study benchmarks some Merkle Tree models on cloud platforms, evaluating their efficiency in terms of parallel capability and construction time. The analysis considers factors such as task division direction, threshold for dividing, and cloud technologies. Results provide insights into the potential Merkle Tree models for application to scalable and secure cloud storage. The findings aid in selecting optimal Merkle Tree implementations and also discuss some ongoing developments of Merkle Tree for data-driven areas in cloud environments like smart IoT or healthcare.

Keywords: Merkle Tree · Parallel Computing Service · Cryptographic Hash · Dataflow.

1 Merkle Tree and Zero-knowledge Proof

This section aims to explain the concept of zero-knowledge proof and why it is so important to protect sensitive data's integrity. Merkle Tree is a zero-knowledge-based structure to verify data using hash algorithms.

1.1 Zero-knowledge proof

Zero-knowledge proof (ZKP) is a cryptographic procedure that enables a party to demonstrate to another that a claim is valid without revealing any information beyond the claim's validity [1]. ZKP has emerged as an approach for continuous verification in zero-trust networks. The most notable application of ZKP is in blockchain. Take Bitcoin as an example of when account A wants to make a transaction with account B. Obviously, B would need to check if A actually owned the money he transferred, but B cannot peek into A's account to see the available balance as it should be private. To solve this problem, the blockchain will provide proof when A creates a valid transaction. When B receives the transaction and the proof, he can verify whether the transaction belongs to a valid block or not using the proof provided. This helps B confirm the validity of the transaction without revealing any sensitive information from A. The proof must

have three properties: it must be convincing, sound, and must be zero-knowledge. ZKP is also applied in message signatures, authentication, and electronic voting.

ZKP is built using one or multiple cryptography hash functions, which are one-way functions. A one-way function is a function that can be computed in polynomial time but is hard or almost impossible to invert. This can help avoid tracing back the original data using the proof. Bitcoin uses the Elliptic Curve with the SHA-256 hash function and employs the Merkle Tree structure to support ZKF for transactions.

1.2 Merkle Tree

Merkle Tree was invented in 1979 by R. Merkle [2] but has only been commonly recognized since 2009 when Satoshi Nakamoto introduced Bitcoin. NIST defined Merkle Tree as "a tree in which each internal node has a hash of all the information in the leaf nodes under it. Specifically, each internal node has a hash of the information in its children. Each leaf has a hash of the block of information it represents. All leaf nodes are at the same depth.".
Figure 1 [3] displays the structure of a simple binary Merkle Tree. The lowest

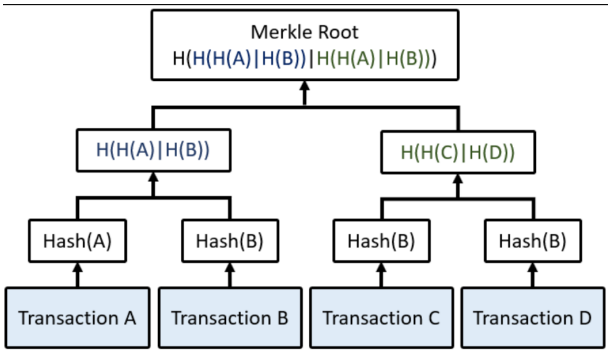

Fig. 1. Merkle Tree Structure

level includes raw data (transactions in the case of Bitcoin). From there, we calculate the hash of each leaf. Then, for each higher level, we recursively calculate an internal node by hashing the concatenation of its children nodes. The number of levels will be logarithmic to the number of leaves. After constructing the Merkle Tree, we only got one final hash called Merkle Root.

Merkle Proof is the path to construct the Merkle Root from a leaf node. In each level of the tree, we only need a sibling node to calculate the hash for the

next level, as in Figure 2. So, the number of hashes needed to reconstruct a Merkle Tree is the same as the number of levels in a Merkle Tree.

This structure helps a user to verify a transaction without knowing any other

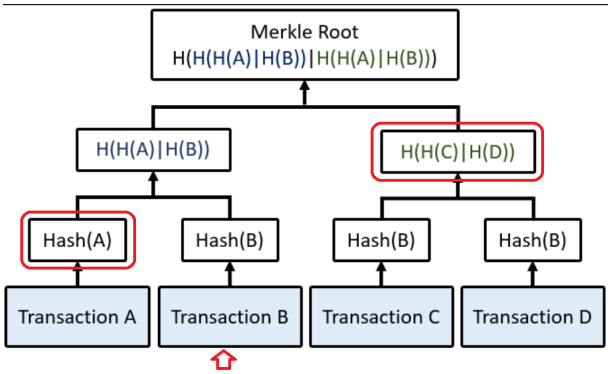

Fig. 2. Merkle Proof

transactions stored in the block by using the proof to reconstruct the Merkle Root and then compare the result with the Merkle Root stored in the valid block. The proof only consists of intermediate one-way hashed nodes, so the user cannot reverse any of the information in the proof into raw transaction data. Another advantage is that the proof size is relatively small to package and send between the servers and their clients.

2 Merkle Tree for Cloud computing

Merkle Tree supports zero-knowledge proof, which means that it allows us to verify whether a piece of information is in a collection of data without accessing the real data. For instance, email lists or voting systems employ Merkle Tree to check whether a vote is in a particular group without revealing the vote-sensitive data. The administration management of cloud services can also benefit from this model by applying Merkle Tree to check if a user has specific permission or not without really knowing what the permission is. The same can be applied to cloud databases to create a verifiable database while maintaining sufficient performance.

However, cloud databases can be really large and distributed, and cryptographic hash, such as SHA-256, is resource-sensitive and time-consuming. How do we construct a Merkle Tree in such an environment, and will it work efficiently? There are some approaches to parallelize the construction of Merkle

Tree, including the same-level parallel and recursive parallel approaches.

2.1 Same-level parallelism

This approach aims to calculate the hash of tuples of nodes within a level in multiple threads. At each level, for a k-ary Merkle Tree, k nodes are grouped into a tuple and hash the concatenation of all nodes inside a tuple. The index of the hash result in the next level can be calculated based on the indices of the nodes. Therefore, we can divide the tuples needed to process on a level into different threads to calculate the hashes in parallel.

Since the number of nodes in each level drops exponentially, this approach is

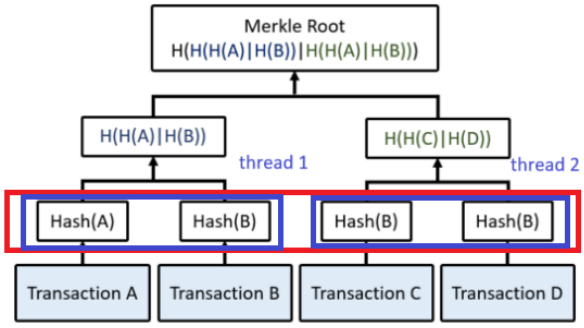

Fig. 3. Calculate hash within a level in parallel

good for lower levels of the tree, while we can calculate sequentially when the tree is high enough.

2.2 Recursive parallelism

Another approach is to divide the tree into sub-trees and calculate them parallelly. This approach divides the data for parallel tasks better, as each thread has already produced the sub-root for a tree, and we can save and reuse that result without recalculating the whole tree.

For example, when a new node is added to the tree, if the right-most branch is not full, we have to recalculate that branch. Otherwise, we create a new branch with only the new node from the bottom up to the root. In k-ary trees, we would save a lot of computation time by storing the result of sub-trees. This allows Merkle Tree to work with streaming data.

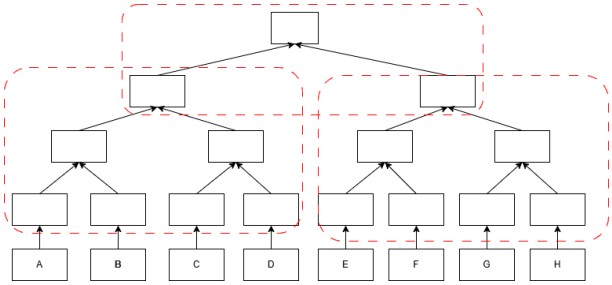

Fig. 4. Calculate sub-trees in parallel

2.3 Google Cloud Service

In this research, we will set up and run different Merkle Tree approaches in the cloud environment, mainly on Google Cloud. In the first model, we use Google Cloud Function, which is a service for deploying serverless applications. Although it can auto-scale based on the number of requests, which means that we can divide the input data into chunks, with each request handling a part of the files, it has a trade-off in terms of time to transfer data and to set up a mechanism to aggregate all the results for further processing. For the sake of simplicity, we only use an instance of the virtual machine in this setup.

For the multiple virtual machines environment, we employ Google Dataflow, which is a service that provides unified stream and batch data processing at scale. It is the next generation of Map Reduce, built on top of Apache Beam, which allows data to go through a predefined pipeline with join and aggregate runners processing them. Google Dataflow can autoscale by provisioning extra worker VMs or by shutting down some worker VMs if fewer are needed while hiding all the complexities.

3 Implementation & Benchmarking

To simulate a simple cloud database, we use Google Cloud Storage to store key-value files, each containing a million records. The Merkle Tree is deployed to calculate the root of each file whenever there is a new record or file added to the database. Additionally, we utilized AWS Lambda for its "warm container" feature to scale data processing across hundreds of workers.

3.1 Single VM with Cloud Function

In the first setup, we deploy 3 models of Merkle Tree (Standard, Layer-based Parallel, and Recursively Parallel) into 3 Cloud Functions. For the parallel models, we have to decide a threshold of how small we should divide the task. We

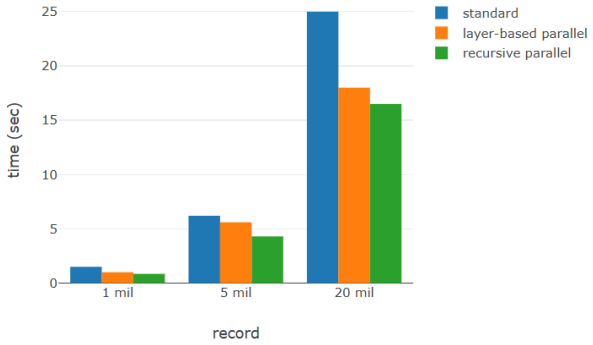

Fig. 5. Runtime of Merkle Tree on a single machine with 4 CPUs and 2Gb RAM

try with 3 threshold parameters of 256, 512, and 1024.

We conducted a run on a single virtual machine with only 4 CPUs and 2Gb of RAM. The result in Figure 5 shows that for verifying a database of 1 million records, the Standard and parallel versions perform almost the same, while for a large enough database of 20 million records, the Recursive parallel model can reduce runtime by 38%. The performance of multithreaded models will increase linearly with the number of CPUs. However, there is no major difference in runtime between the Layer-based Parallel model and the Recursive Parallel model.

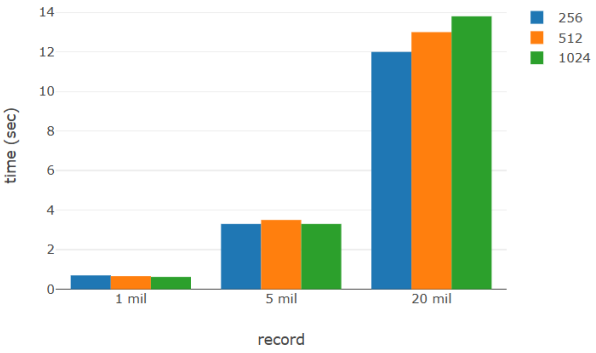

Fig. 6. Runtime of Merkle Tree on a single machine with 8 CPUs and 4Gb RAM

Next, we tested different threshold values for dividing sub-tasks using the Recursive Parallel model. In Figure 6, for a small number of records, a threshold of 1024 nodes per sub-tree has the best performance. For 20 million records, a

smaller threshold of 256 performs better since we can optimize the number of CPUs run parallelly.

As we scale vertically (increasing the number of CPUs), parallel models of Merkle Tree perform better. However, the power of a machine is limited; therefore, we need to scale it horizontally, i.e., calculating the Merkle Tree using multiple VMs.

3.2 Multiple VMs with Dataflow

We created a Merkle Tree construction pipeline with Dataflow, which was based on Apache Beam. The input files are identical to the first setup, and for the threshold, we decided to use a value of 512. We set the scale factor to 6 virtual machines, each with 8 CPUs and 4Gb of RAM.

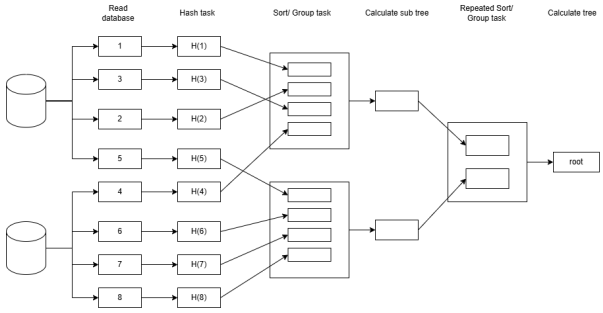

Fig. 7. Merkle Tree construction pipeline with Dataflow

First, the data are fetched from database files; then, they go through a hash task, which uses the SHA-256 hash function. As described in Figure 7, the hashes collection will then be grouped into ordered tuples based on the record sequence number. After that, each tuple will be passed into a calculation task to get the root of the sub-tree. The calculation task can use one of the three models of Merkle Tree, and the result is a collection of sub-roots. Then, it repeats by grouping the sub-roots into tuples again, using the newly calculated sequence number of the root. The pipeline ends when there is only one node left, which is also the Merkle Root.

Although there was time overhead for grouping and transferring nodes, the experiment was conducted under the condition that the running VMs and the data were in the same region, and all the VMs had 8 CPUs and 4 GB of RAM. The Dataflow was slow at the beginning, with 1 million records, but it outperformed single VM options in large data volumes. It should be noted that the

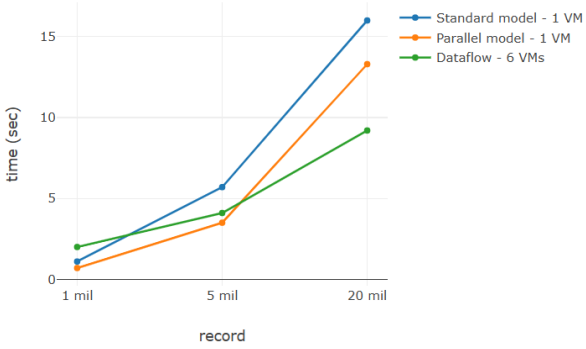

Fig. 8. Merkle Tree in single VM versus Dataflow

performance of virtual machines is unpredictable. Therefore, we could not conclude that using the Dataflow setup for Merkle Tree construction is always a good choice.

3.3 Distributed Data Processing using AWS Lambda

The system is designed with a central coordinator that distributes the work among several serverless functions. The coordinator takes a large file and splits the processing into manageable pieces by assigning each serverless function a set of parameters: the file name, the size of each chunk, a starting point (skip), and the number of chunks to process (take). This allows the heavy processing to be broken down into smaller tasks, which are then handled concurrently by different functions.

Each serverless function, upon receiving its task, downloads the entire file from cloud storage into its local storage even though it only processes a specific segment determined by the provided parameters. This means that if the function's container is already warm (i.e., it has recently processed a task, and the file is still present), it skips the download step, which helps reduce overhead and network usage. Once the file is available locally, the function processes its assigned segment and computes a hash tree for that part of the file.

At the end of its computation, each serverless function returns the hash for the root of the sub-tree it has computed, formatted in hexadecimal. The central coordinator then takes these hexadecimal hash values and treats them as the leaves of a final Merkle tree. Combining these sub-root hashes computes the global root hash, which represents the entire dataset. This layered approach to hashing not only enables efficient parallel processing but also provides a reliable way to verify the integrity of the processed data.

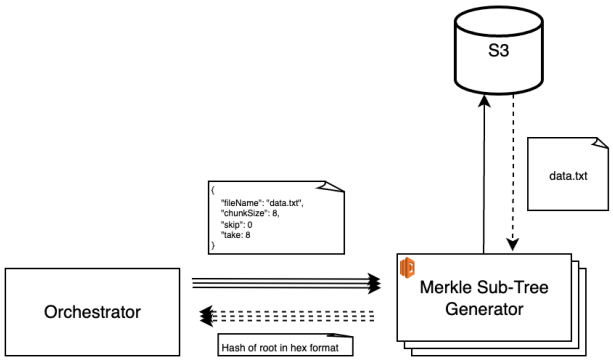

Fig. 9. Architecture for orchestrator-worker serverless pattern

Results The chart below compares how long it takes to process files of different sizes when varying the chunk size assigned to each worker. On the horizontal axis, you have the chunk size in powers of two, and on the vertical axis, the total time taken in milliseconds. Each line represents a different input file size: one for 1 million lines, another for 5 million lines, and another for 20 million lines.

When the chunk size is relatively small (towards the left side of the chart), the number of workers required can be extremely high based on the input file size. The orchestrator can only manage a certain number of outgoing HTTP requests at once; it quickly hits a bottleneck of 300 maximum concurrent HTTP requests: it simply can't dispatch thousands of requests in parallel. This leads to longer total processing times because many requests end up waiting in a queue. One way to address this is by introducing an intermediate layer of client servers to distribute these requests more efficiently rather than relying on a single orchestrator.

As you move to larger chunk sizes, the number of workers needed drops, which means the system no longer saturates its concurrency limit. Once you go below around 300 workers, where each request can be sent at the same time, parallelism is maximized for all file sizes. At that point, the main factor becomes the processing work each worker does rather than how many requests the orchestrator can send out.

Interestingly, by the time you reach a chunk size of about 2 ^ 17, the lines on the chart converge. At this chunk size, the performance differences between the various file sizes are negligible. Essentially, whether you have 1 million, 5 million, or 20 million lines, the total time flattens out because the system is neither overwhelmed by too many workers nor slowed down by each worker handling an enormous amount of data. This "sweet spot" shows that once the concurrency limit is no longer a concern and the chunk size is balanced with the workload, the total processing time becomes roughly the same for all input sizes.

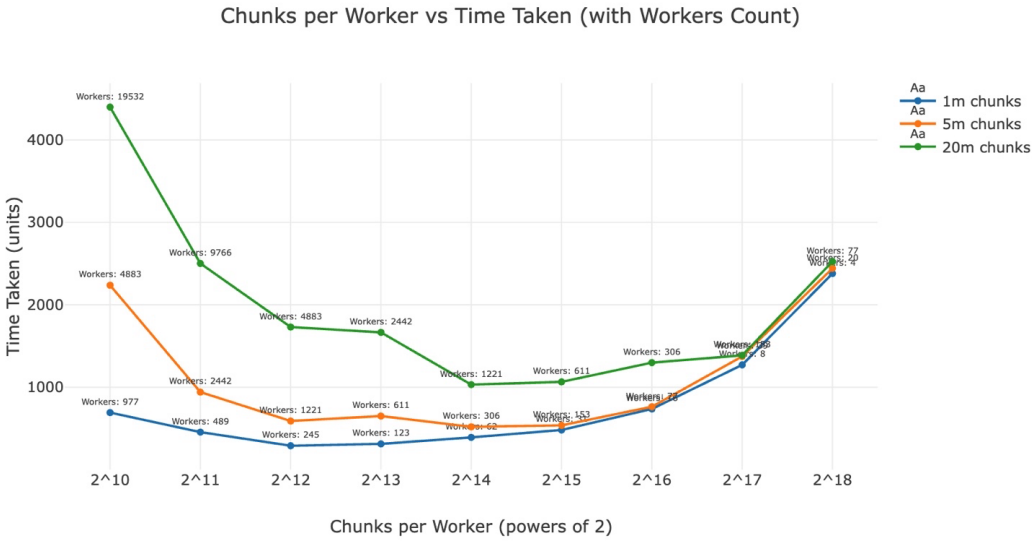

Fig. 10. Orchestrator-Worker approach results

4 Discussion

Cloud security and data integrity have always been major concerns in cloud computing. A zero-trust network is a paradigm that has been applied widely but is not yet common in the scope of data records. Since the introduction of Bitcoin and smart contracts, Merkle Tree variants have been commonly adopted to verify data with ZKP, with an average of 2000 transactions per block. By constructing a pipeline to build the Merkle Tree using virtual machines or cloud architectures, we can utilize its power to verify a larger set of data, such as millions of voting records or permission checks.

The Merkle Tree can also be modified to support streaming data, such as verifying data from IoT sensors, by saving untouched branches and only recalculating the branch containing the new data. C. Smith and A. Rusnak [4] provided a solution with the Dynamic Merkle B-Tree structure that allows adding or removing elements to and from a Merkle Tree. Applying this model with Dataflow pipeline and Pub-Sub service will enable seamless data verification from different sources of IoT devices.

Another area of application would be securing healthcare data, as more and more electronic medical records are created containing tons of sensitive data. To reduce the number of hash calculations, Hegui Zhu et al. [5] proposed a convolutional Merkle tree structure as in the CNN network and replaced the original hash function with a hash function for medical data. This improved the hash

computation time and is efficient for securely storing medical data in the cloud environment.

Cloud computing provides an efficient way to store and process data, but it is also a risky environment. Applying blockchain technology like Merkle Tree to cloud storage will improve the security of data stored on the cloud. However, specific implementations of this technique must be carefully designed for each type of data to work efficiently since it also requires a lot of computing resources and data storage.

References

1. Marinka Zitnik. 2013. Zero-knowledge Proofs. XRDS 20, 1 (Fall 2013), 65–67. https://doi.org/10.1145/2517258
2. Paul E. Black, "Merkle tree", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 19 February 2019. (accessed TODAY) Available from: https://www.nist.gov/dads/HTML/MerkleTree.html
3. An Image Authentication Scheme Using Merkle Tree Mechanisms - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Merkle-tree-structure_fig3_334291891
4. Smith, C. and Rusnak, A., “Dynamic Merkle B-tree with Efficient Proofs”, Art. no. arXiv:2006.01994, 2020. doi:10.48550/arXiv.2006.01994.
5. Hegui Zhu, Yujia Guo, and Libo Zhang. 2021. An improved convolution Merkle tree-based blockchain electronic medical record secure storage scheme. J. Inf. Secur. Appl. 61, C (Sep 2021). https://doi.org/10.1016/j.jisa.2021.102952