Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree

Janakirama Kalidhindi, Alex Kazorian, Aneesh Khera, Cibi Pari

Abstract— **Merkle trees allow for efficient and authenticated verification of data contents through hierarchical cryptographic hashes. Because the contents of parent nodes in the trees are the hashes of the children nodes, concurrency in merkle trees is a hard problem. The current standard for merkle tree updates is a naive locking of the entire tree as shown in Google's merkle tree implementation, Trillian. We present Angela, a concurrent and distributed sparse merkle tree implementation. Angela is distributed using Ray onto Amazon EC2 clusters and uses Amazon Aurora to store and retrieve state. Angela is motivated by Google Key Transparency and inspired by it's underlying merkle tree, Trillian. Like Trillian, Angela assumes that a large number of its 2^{256} leaves are empty and publishes a new root after some amount of time. We compare the performance of our concurrent algorithm against the approach used by Google Trillian in a local setting for which we see nearly a 2x speed up. We also show that Angela's distribution is scalable through our benchmarks.**

I. Introduction

Merkle trees are simple in their design. For some arbitrary node in the tree, we can derive its value by applying a hash function on the values of all of its children nodes [8]. One can think of a merkle tree as a set. By applying the definition of a node's value, inserting data into a merkle tree can be thought of as a commitment to the data. This commitment to a piece of data allows for verification of the integrity of data received over a network, giving them increased importance in recent years.

One such example is the Key Transparency project by Google, a public key infrastructure, that uses Google Trillian, a sparse merkle tree implementation with transparent storage. Key Transparency uses a merkle tree to pair a user identifier key in a leaf with a public key value [4]. When an update to a public key comes in, the entire tree is locked and updated serially, introducing massive bottlenecks which are exacerbated when the intended scale of Key Transparency is introduced due to the high surface area of contention. Google Key Transparency requires the scale of a billion leaves with high query throughput. Any concurrency overhead should not destroy the system's performance when the scale of the merkle tree is very large.

Outside of Key Transparency, this problem is generalizable to other applications that have the need for efficient, authenticated data structures, such as the Global Data Plane file system.

As a result of these needs, we have developed a novel algorithm for updating a batch of updates to a merkle tree concurrently. Run locally on a MacBook Pro with a 2.2GHz Intel Core i7, 16GB of RAM, and an Intel Iris Pro1536 MB, we achieved a 2x speedup over the naive algorithm mentioned earlier. We expanded this algorithm into a distributed merkle tree implementation, Angela, that is able to scale to handle demanding workloads while maintaining its performance. Angela is comprised of an orchestrator node that handles load balancing. This is written in Python and leverages Ray to handle distribution [9]. The orchestrator distributes the work to multiple worker nodes written in Golang for concurrent merkle tree updates.

The rest of the paper is structured as follows. Section 2 lays out our metrics of success and what we consider to be a strong and promising results. In Section 3, we discuss the scope of what Angela can do. In Section 4, we provide a fundamental assumption for our merkle tree implementation and a detailed description of the specifics on how the underlying merkle tree is implemented. Section 5 describes both the naive algorithm and why concurrency is not immediately obvious, following which we describe the concurrent insertion algorithm. Section 6 presents the system architecture for our complete distributed system under Ray. Section 7 describes some of the challenges we faced and reasons behind the language decisions that were made along the way. Section 8 shows benchmarking results of the system and discusses the evaluation of Angela overall. Section 9 presents related work, and Section 10 and 11 present future optimizations and work that we plan to act on before wrapping up with a conclusion and acknowledgements.

II. Metrics of Success

Our first goal was to develop a concurrent algorithm in the non-distributed case that cleanly beats the naive algorithm while maintaining the integrity of the merkle tree. The speedup we expect to see here should respect the number of cores available on the testing hardware with some latencies due to the percolation paths not being completely parallelizable. The second goal was to adjust the algorithm for effective distribution on Ray and launch the entire system on a cluster of Amazon EC2 instances while maintaining performance. Our final goal was to measure performance on this distributed system with respect to the number of compute nodes in a cluster. Ideally, we would see the throughput of writes scale with the number of nodes in the cluster.

III. Project Overview

In this section, we define the project scope of Angela through the client API. The specifics on how Angela handles these requests is detailed further in the Architecture section.

1) insert_leaf(index, data): A client is able to insert a new node into the merkle tree as well as update an existing node. *index* refers to the key used to insert into the merkle tree and *data* refers to the value that the key refers to.

2) get_signed_root(): A client is able to request the published root of the merkle tree. This is used in order to verify any read request from Angela.

3) generate_proof(index): A client is able to read a value from the merkle tree. This works for both items that have been inserted in the tree and for those that do not exist in the tree. In either case, we return a Proof object that the client can verify to confirm that a items actually does or doesn't exist. We use membership proofs for items that exist in the tree and and non-membership proofs for items that do not exist [10]. *index* refers to the key that would have been used to insert into the merkle tree.

4) verify_proof(proof, data, root): A client is able to verify that a proof generated by the server maintaining the merkle tree is valid. This verification happens locally on the client side and is not sent to the server. $proof$ is the value that is returned from $generate\_proof$. $data$ is the value that we are verifying to be correct. *root* is the digest that is retrieved form $get\_signed\_root()$.

Similar to CONIKS and Trillian, we fix the values of empty nodes to something related to their depth [7] [5]. This means we can lazily calculate hashes of empty nodes during inserts or when generating proofs. The merkle tree will also be organized as a binary search tree, where most nodes are actually empty. A prefix is assigned to each node in the merkle tree with a 0 appended on a left child and a 1 appended on a right child, resulting in the leaf locations to be determined via their sorted key values [7]. Writes are batched together and processed in epochs. In our system, we wait for some number of update requests to be queued before processing them; however, it is also possible to use time elapsed as an indicator for publishing applying updates and publishing a new root.

IV. Design

A. Verifiable Random Function Assumption

CONIKS uses a verifiable random function to map user IDs to leaves within the tree [7]. We assume that IDs given to the service have already gone through a verifiable random function. This is a reasonable assumption since this can be added in trivially to all inputs, but to reduce the complexity of our implementation, we chose to not to. This gives us the interesting property that any query or workload will appear to be random and, therefore, properly load balanced.

B. Size

The merkle tree contains 2^{256} leaves. Again, we assume that a large number of nodes in the tree are empty. Moreover, to ensure that IDs are mapped to leaves properly, we assume that the aforementioned verifiable random function has an image of size 2^{256}.

C. Hash Functions

We use SHA256 as our hash function for Angela, but this is easily replaceable with any other cryptographic hash function.

D. Encoding

Our merkle tree design is similar to CONIKS in that we use a very basic prefix encoding. All nodes on level i have prefixes of length i. To determine the prefix of some node, one can just traverse the tree. Going down the left branch means we append a 0 to our prefix. Going down the right branch means we append a 1 to our prefix.

E. Sparse Merkle Trees

Literature on such sparse representations already exists under the name of Sparse merkle trees. Sparsity allows for lazy construction of and queries on our data. Thus, the tree itself will be more space efficient and

will theoretically result in easier coordination among the distributed clusters [3]. Figure 1 provides a visual representation of how a sparse tree would look using our encoding scheme and what state is actually stored. [7].

F. Sparse Representation

In order to represent the sparse tree, we assign a constant hash value to each empty node at level *i*. This constant value is determined by applying a hash function on the concatenation of the 2 empty children [3], [7]. The base case for an empty node at depth 256 is simply the hash of the empty string.

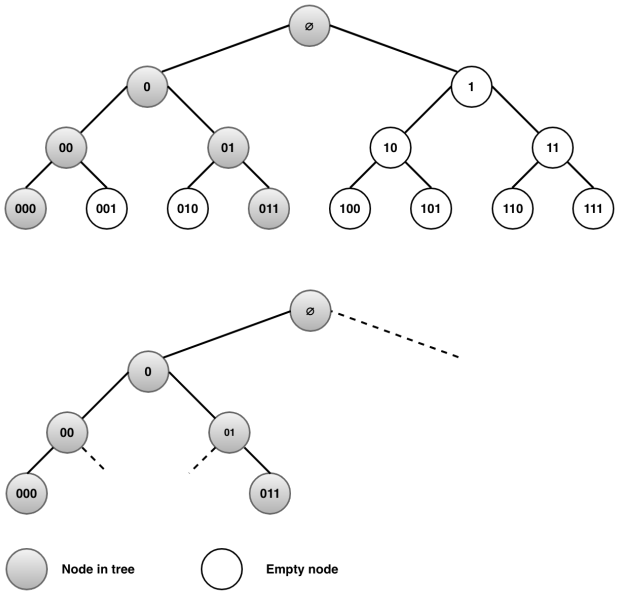

Figure 1

V. ALGORITHMS

For simplicity, we only discuss algorithms that will be applied to the non-distributed case. This allows us to abstract away the complexities of dealing with a distributed system and focuses on how concurrent updates are run on the merkle tree. Instead, we discuss challenges we faced while translating this to the distributed case as well as distributed implementation details in the section labeled "System Architecture."

A. Why Locking is Hard

Before we start describing the novel concurrent algorithm that is implemented for Angela, we describe the naive algorithm and why concurrency in this paradigm is difficult. Each parent node is dependent on each of its children when updated. Now, consider a scenario in which we are concurrently inserting or updating the value of two leaf nodes. These two nodes have a point of conflict and contention in some ancestor along their path to the root. It is at this ancestor we have a race condition. Therefore, the smallest example for which we have an obvious race condition is one with 2 leaves and a root. If we attempt to insert a value into or update the value of these two leaves concurrently, then we have reproduced the only race condition present in the merkle tree update procedure.

Applying a locking scheme on this minimal example is not simple. Below, we present an example where 2 transactions, A and B, fall into a situation of deadlock. Nodes labeled with A imply that that A currently has a lock on that node, likewise for B. Here, we see that A currently has a lock on the root node and is waiting for the its right child to be unlocked in order to update the root. The right child, with a B lock, is waiting for both its parent and sibling child lock to be released so that it can begin to percolate the changes upwards. Thus we have a situation of deadlock, because both transactions are in a state where they are waiting for a lock that the other transaction contains.

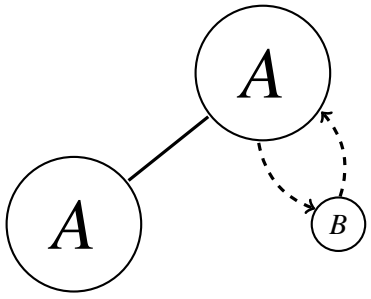

Figure 2

Another potential solution requires grabbing the sibling node before making an update. This approach, as shown below, may lead to deadlock as shown in the diagram below where Transaction A and Transaction B both require the other sibling in order to begin the update. Thus both transactions will not be able to continue.

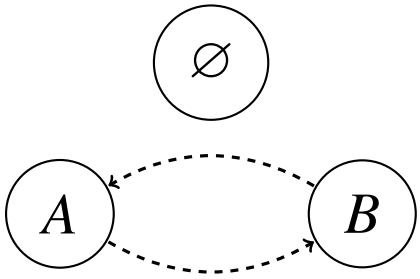

Figure 3

B. Naive Algorithm

Given this paradigm of locking, avoiding deadlock is difficult and continuously locking and resolving possible deadlock is costly. Because of this, the naive algorithm is a fairly simple serial solution that seems to be the current standard. Google Trillian, which is Google's implementation of a merkle tree that is used in the Google Key Transparency project, follows this naive algorithm [5] [1].

procedure $naive\_update_{\textbf{Tree}}$(Transaction T)
    Tree.$lockAcquire()$
    Process Update T
    Tree.$lockRelease()$
end procedure

The high surface area of locking presented in both the naive algorithm as well as the possible concurrency through locking schemes presented above cause these solutions to be sub-optimal. Abandoning this paradigm, we present our concurrent algorithm.

C. Finer Grain Conflict Locking

1) Conflicts: In the case that there are two leaves that need to be updated in the merkle tree, we define a conflict node to be the deepest common ancestor of two leaves. In other words, a common ancestor is the node that shares the longest common prefix among two leaf nodes. In the following figure, we present a diagram containing three nodes in the process of being updated and the points at which they conflict and there is contention.

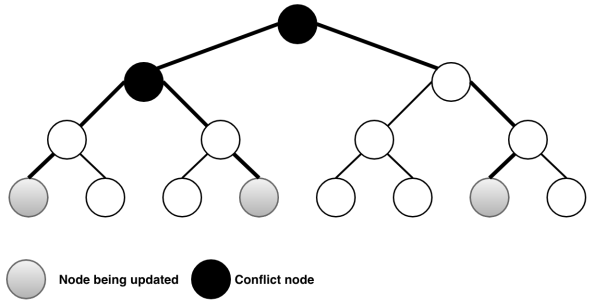

Figure 4

Because these are the points in which the update paths conflict, these are the nodes that require locking to ensure that a race condition does not occur. In fact, we can see that given a list of N leaf nodes, there are $N-1$ conflict nodes between. The proof for this is simple. Given two update nodes, we know that their paths must conflict at some point. In fact, since each node only has two children, at most two nodes can conflict at any given point. Even if there are more children being updated at a deeper level, those would have reached a conflict before. Thus, adding a new update node, adds a new conflict point. From here, the rest of the proof can be derived.

2) Finding Conflicts: Finding these conflict points is also simple because of the encoding used on our merkle tree. Given the ID of two update leaves, we can find their conflict node by finding the longest common prefix of the two IDs. For example, given node IDs, "000" and "011", the longest common prefix is "0". Looking at Figure 1 describing an encoded sparse merkle tree, we can see that the node encoded with "0" is the conflict point.

3) Sorting to Find Conflicts: However, comparing two nodes in a batch of updates does not guarantee that the conflict point will be unique and presents the possibility of conflict points being undiscovered. For example, given update node IDs "000", "101", and "001". Finding conflicts on "000" and "101" gives us the root. Finding conflicts on "101" and "001" also gives us the root. In reality, the conflict points should be the root and "00". This issue comes about because randomly checking for conflicts does not take locality of updates into account, and thus does not provide conflicts in those areas. We can fix this locality issue by running our conflict search on pairwise nodes when they are sorted based on our encoding. Looking back to our previous example, sorting "000", "101", and "001" gives us "000", "001", and "101". Running the conflict search on pairwise nodes then gives us "00" and the root.

For a better understanding as to why sorting works and provides us with all $N-1$ unique conflict nodes for N updates, we can look at the following reasoning. A conflict point is the nearest point that two update paths conflict. In order to find the conflict points that are lower in the tree, we need to minimize the distance between update nodes so that their conflict point is lower. Given the encoding on the leaves, we can sort the leaves and thus the neighbor on each side is the closest leaf on that side. Thus, searching for conflicts on these pairwise nodes results in finding the unique conflict point that is closest to those nodes.

4) Conflict Set: Given N updates, we will have $N-1$ conflict points. In fact, these $N-1$ points are the only places where synchronization is needed as the rest of the update path is not touched by other updates and safe to access. We have essentially reduced the surface area of locking contention from the tree to linear in the

number of updates. We can place locks around these nodes and maintain a set of these conflicts.

D. Percolating Changes Concurrently

1) Algorithm Overview: Given that we know the conflicts, the pseudo code for the batched update algorithm is as follows:

```
procedure batch_update_Tree(Transactions T)
    Tree.conflicts = find_conflicts(sort(T))
    for t in list(T) do
        Tree.update(t)
    end for
end procedure
```

Each individual update is described as follows:

```
procedure update_Tree(Transaction T)
    while parent_exists() do
        if parent in Tree.conflicts then
            Tree.conflicts(parent).acquire_lock()
            defer Tree.conflicts(parent).release_lock()
            if parent has not been visited then
                Mark Node as visited
                End Thread
            else
                Process Update
            end if
        else
            Process Update
        end if
    end while
end procedure
```

After finding the conflicts, we start running individual updates concurrently. In each of these concurrent updates, we follow a straightforward procedure to avoid inconsistent views of data at points of contention. Before updating a parent, we check if the parent is a conflict node. If it not a conflict node, we know that there will be no race conditions in accessing a sibling as there are no other updates on that path. If the parent is a conflict point, we grab the lock on this node. In previous sections, we explained how a locking scheme involving a parent and it children can lead to deadlock situations. Simply locking the parent circumvents this issue of deadlock as any node can only hold or wait for one lock at a time. After acquiring the lock on the parent node, we check to see if the parent node has been visited yet. If it has not been visited yet, we know that the sibling is not ready and another update is still happening. We do not want to wait for the sibling to finish as this is busy waiting and the resources can be used for another update. We mark the node as visited, release the lock, and the update procedure exits. When the sibling update finally comes, it will acquire the lock and see that the parent has been visited. Because the parent has been visited, we know it is safe to read the sibling and continue with our update path.

The pattern of returning earlier also means that any edge on the tree is only ever traversed by one update. Anytime two updates come to a point where they overlap, one of them returns and allows the other to continue. This also means that general contention on locks is also relatively small. There are other implementation details that speed up this process, but have been omitted for brevity.

VI. System Architecture

We utilized Ray as a distributed framework in order to manage the nodes of the distributed merkle tree. The computation for each of these nodes is performed on a cluster of Amazon EC2 instances. Ray's head node is utilized as an orchestrator, accepting incoming transactions and load balancing them to the different worker nodes of the tree. Amazon Aurora, a MySql compatible relational database, is being used as the storage layer.

A. Virtual Addressing and Tree Structure

The representation of the merkle tree is divided up into separate nodes, with each node containing a subtree of the full tree. In order to allow for independent distribution each subtree, we utilize a virtual addressing scheme, where a virtual address can be mapped to a physical address by prepending a prefix specific to that worker node. Each worker node holds a prefix that is generated from the worker's unique ID.

In Figure 5, we the physical addressing scheme and how it maps to the virtual addressing of each separate node. Each node contains only its view of the subtree, thus allowing for complete independence between the worker nodes.

In order to manage these subtrees, we utilize Amazon EC2 instances, spawning 1 instance per subtree. The tree is divided into a root node and subsequent worker nodes. The depth of the subtree in the root node is determined by $\log(num\_workers)$, and is thus is a much smaller than the rest of the worker node subtrees. We keep the root node subtree small in order to maintain a smaller number of total subtrees. Also, with a smaller

root subtree, there are fewer conflicts in the root node, whereas the as a percolation progresses higher in the tree, we would typically see more conflicts. Thus, by keeping the root node subtree small, we minimize the amount of conflict updates that need to happen serially.

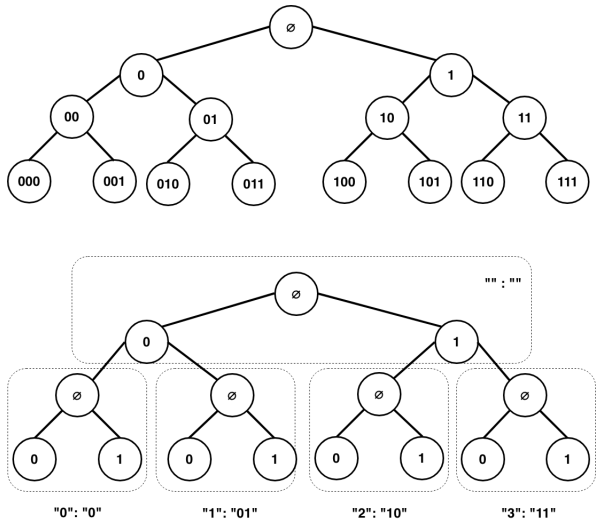

Figure 5

B. Ray

Ray is a flexible, high-performance distributed execution framework [9]. Ray is an ongoing project being developed in the UC Berkeley Rise Lab, built to support computation-intensive distributed, scalable systems in Python. Ray implements a unified interface, supported by a single dynamic execution engine, that allows for task-parallel and actor-based computations. Angela utilizes Ray as a workload management and distribution tool, specifically to control parallel computation.

Ray provides lineage-based fault tolerance for tasks and actors, and replication-based fault tolerance for the metadata store, which ensures the fault tolerance of the actor based portion in Angela [9]. If a node fails, Ray will restart the node in order to assure the computation completes. Ray handles the deployment of the cluster, spawning each node and managing its health.

We utilize actors for our merkle tree in order encapsulate the mutable state of for each subtree. Angela utilizes m5.large EC2 instances in the cluster with spot instance types.

C. Stateless Worker Nodes

Worker nodes do not store any state between epochs. This is because the chance of hitting the same leaves from one epoch to another within a single subtree is very low, as we do not anticipate users to request multiple updates to the same leaf over successive epochs. Additionally, given the large size of the trees and randomization on the keys, the probability of hitting the same path is very low. Thus, it does not make sense to maintain a stateful cache within each worker node. Instead, only the copaths for each of the updating nodes are needed to perform an update - and this happens on each worker node without any necessity to maintain state. Because worker nodes are stateless, they can also easily be repurposed to be responsible for a different subtree, or to perform a different operation. This is particularly useful in the case in which we have more subtrees than available EC2 instances.

D. Storage Layer

Angela utilizes Amazon Aurora as the storage layer for the system. Amazon Aurora is a relational database built for high throughput data processing [12]. Aurora also supports parallel queries as it stripes data across hundreds of storage nodes distributed separate AWS availability zones [12]. For Aurora, we use a single replica, r4.xlarge, containing 4 virtual CPUs and 30.5 GiB RAM. Angela users have the option to change the database usage as they wish as long as they are able to supply a new Go driver. Currently, the implementation interacts with Aurora through a MySQL interface.

E. Database Schema

As a prototype database schema, we elected to store all node data from the merkle tree in one table called nodes. This table stores the nodeId (prefix that determines the node's location in the tree), the nodeDigest (actual hash value of the node), and the epochNumber (which version of the tree this node belongs to). Since we use bit strings as our representation for nodeId and nodeDigest, both are stored as VARBINARY(256) columns. We place a primary key on (nodeId, epochNumber) which subsequently causes MySQL to create an index on (nodeId) and (nodeId, epochNumber). Read queries for a set of nodeIds return the rows of the table belonging to the maximum epoch for those ids. With the index on both nodeId and epochNumber, it is expected to boost query performance. Write queries always write to the max epoch in the database +1 so the nodes table basically just holds each epoch's changelist. Versioning with epochs allows us to direct reads to the latest version of the data that exists in the database while writes are being accumulated for the next epoch. Another benefit

of versioning is that we have fallback values in case and update goes wrong during a particular epoch.

F. C Extensions

In order to connect Ray, which only supports a Python API, with our Go implementation of Angela we make use of Python's ctypes library and compilation of our Go code into a C shared object file that could be loaded in Python. We establish a thin API layer to reduce the number of types and operations we need to support on either end of the interface. The most complicated operation involved in our interface is that of a read request, which requires the server to return a proof including the copath of the key requested to the client. This led to an interesting requirement implementation-wise where we needed to pass an array of strings allocated through the cgo library to the ctypes library with separate API functionality for freeing the memory allocated through this method. Although this has limited us to data types supported by ctypes, sending transactions across this boundary has limited latencies in getting client input to the Go implementation.

G. Read Phase

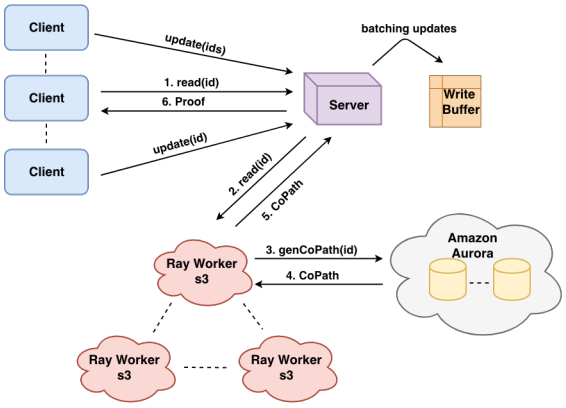

Figure 6

In Figure 6, we see the read phase of Angela. Clients initiate a read request to the Orchestrator node labeled Server. The Server then load balances and dispatches the read request to a designated Ray Worker running on an EC2 instance. Each Ray worker is responsible for a specific section of the merkle tree. In order to complete the read request, we need to provide the digest values of all the nodes in the co-path, so the client can verify the integrity of the data requested. The Ray Worker uses the ID that is requested to generate a list of the needed co-path nodes and makes a call to the database. The co-path is then returned to the server and finally returned to the client for client-side verification.

H. Write Phase

Figure 7 shows the flow of a write transaction. If there is a update request during the read phase, it is not processed immediately. Instead, it is placed into a write buffer in sorted order. Once the write buffer on the orchestrator reaches a designated epoch size a batch update is finally made to the merkle tree.

The orchestrator first sends updates to the worker node, determined by the worker nodes respective subtrees. The worker nodes query the database for copaths and follow the algorithm described above. They write their updates back to the database and push the root back to the server. The server then takes the new root nodes from the workers and sends them as a new batch update to the root node.

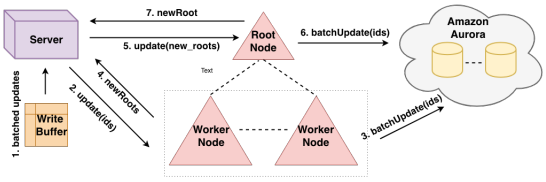

Figure 7

I. Recovery

One nice property of a merkle tree update is that redoing without undoing failed transactions is sufficient to recovery. The reason redoing an update to a node maintains a correct merkle tree is that this procedure sets the value of every ancestor of the node being updated to $H(l||r)$ where H is a cryptographically secure hash function and l, r are the left and right children respectively. Updates are idempotent and can be reapplied until the transaction it is associated with has finished.

As mentioned in a previous section, the fault tolerance properties of Ray also means that if anything within the worker node fails, Ray automatically creates a new worker to redo that batch of transactions. This does mean that the redo is as large as a subtree update. Finer grain redo transactions can be implemented by keeping track of which updates have gone through.

VII. Challenges

A. Python GIL

Our first approach was building Angela's backend in entirely Python. Python has several libraries that allow for easy bitwise operations and data structure manipulation. Additionally, Ray is a Python framework and thus interfaces well for this use case. However, when creating multithreaded applications, Python when run in a CPython environment is severely limited. This is due to the Global Interpreter Lock, a mutex inherently part of CPython. The GIL prevents multiple threads from executing Python bytecodes at once. This proved to be a serious issue in terms of performance and ultimately required us to look for other methods of multithreading.

B. Cython

In order to fix this multithreading issue, we began working with Cython in order to gain control of the GIL. Although Cython has far less documentation, Ray also supports interfacing through Cython. However, Cython proved very difficult to multithread as it only has OpenMP-like parallelism, which is not what we were looking for.

C. Golang

We decided to settle with a language that doesn't necessarily interface with Ray, but supports the high level of concurrency we needed for Angela. Golang, a programming language designed by Google, is fairly similar to C but with several additions to improve code simplicity and safety. Furthermore, Golang provides several concurrency primitives that make concurrency simple without the use of external libraries.

D. C Extensions

Figuring out the proper way to convert and maintain pointers across the Python - Go interface was tricky along with properly malloc-ing memory and freeing pointers in cgo proved to be quite challenging initially.

E. Ray

Ray is a new open source software, currently only on release 0.6.0. The documentation is still developing and we encountered many issues that were difficult to receive support on. Issues we faced included AWS permission configuration, nodes dying sporadically, and lack of full CPU utilization.

F. AWS Credits

Being students, we had limited access to resources and were able to gain access to $300 of AWS credits. While enough for some benchmarking, without the large clusters we were spawning through Ray, we definitely could have performed many tests at a much higher throughput given the ability to purchase better EC2 and Aurora instances.

VIII. Evaluation

In order to evaluate Angela, we consider raw throughput of our algorithm when compared to the naive algorithm as well as comparing latency measurements for the complete system, Angela. For each case, different sized workloads will be applied to Angela as well as varying distribution to assure the scalability of the system.

A. Algorithm Benchmarks

We ran our algorithm benchmarks on a MacBook Pro with a 2.2GHz Intel Core i7, 16GB of RAM, and an Intel Iris Pro1536 MB. On the MacBook Pro, we measured the algorithmic performance of our BatchInsert implementation against the naive insertion implementation in Golang. This implementation is not distributed and does not hit a separate storage layer or database. This means sorting, which was removed in the distributed case, is still done. The purpose of this benchmark was to see if the concurrency implemented would be able to outperform the naive. In our test, we found that BatchInsert achieves approximately 2x speedup over the naive implementation.

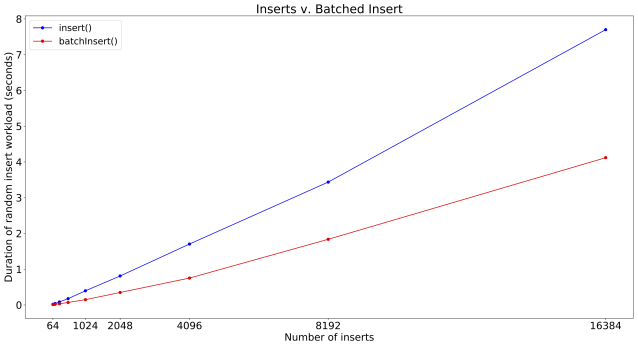

Figure 8

Currently, our benchmarks are limited to our own laptops, but given more cores, we expect to see BatchInsert to do even better. We expect the performance of the algorithm to be limited primarily by the number of available CPUs that can be run, when provided with an appropriate workload.

B. System Benchmarks

When benchmarking inserts with the database, we noticed that the values chosen for batchReadSize (how much of the copath we pull into the cache at a time), batchPercolateSize (determines how many goroutines we choose to run) and batchWriteSize (how many transactions are placed in the buffer before being written to the database) have a large impact on write performance. Although our MySQL interface limited us in the size of queries and the number of connections, we recognized that the hyper tuning these parameters is important for performance of the algorithm and ideally, these parameters would be determined dynamically based on the number of transactions that come in and the compute power of the worker node. The baseline values we have chosen for running our benchmarks are batchReadSize=50, batchPercolateSize=10, and batchWritesize=50. Since database calls are constant between the serial and the concurrent algorithm, and we have shown that the concurrent algorithm outperforms the naive, we can conclude that when implemented in a full system, the concurrent version will be faster. So, we can focus instead on how Angela scales when distributed on varying workload sizes.

In the complete implementation of Angela that we benchmarked, we used m5.large EC2 instances, containing 2 virtual CPUs and 16 GiB RAM. For Aurora, we use a single replica, r4.xlarge, containing 4 virtual CPUs and 30.5 GiB RAM.

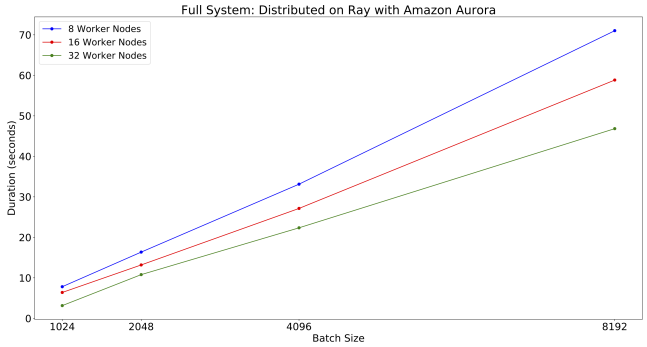

Figure 9

As we increase the workload, we are able to see that in general, the latencies double as we double the workload. We can also see that increasing the number of worker nodes reduces the latencies.

In the perfect scaling situation, we would see that if we doubled the worker nodes, we would be able to handle double the workload with minimal impact to the run time. If that was the case, we would be able to connect those points and see a horizontal line. Initially, when we drew this line, we saw that the system was not able to scale perfectly. Issues here might be attributed to imperfect distribution over the nodes and other network latencies across the database. Instead, when we quadruple the worker nodes, and double the workload, we are much closer to seeing the horizontal scaling line, which would be optimal performance. This is shown below.

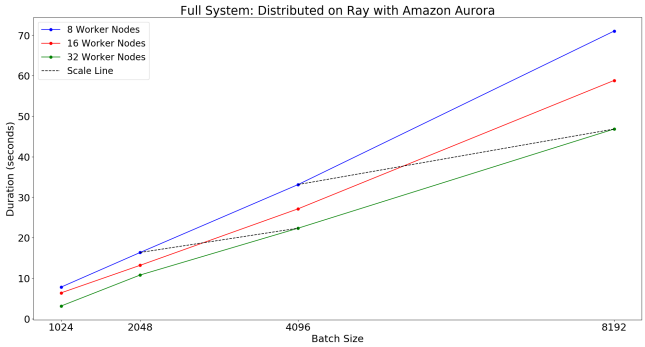

Figure 10

IX. Related Work

Our work is one possible solution for the problem of distributed, authenticated key-value stores. There has been other work in this area of research; however, few focus on distributed merkle trees. One such effort though is that of Tamassia and Triandopoulos whereby they develop a scheme for distributing a merkle over a peer-to-peer network [11]. While novel, this work does not fall in line with merkle tree designed to solve the same class of problems as CONIKS. In this problem setting, we want merkle trees to be append-only data structures that guarantee protection against key revocation as well as the expected property of authentication that comes with a merkle tree. The merkle tree design presented by Tamassia and Triandopoulos is too flexible. Because they allow for removal of keys, their design does not immediately protect against a malicious server attempting to revoke a user's access whereas ours does. We don't implement the entire CONIKS protocol. Instead, we implement the smallest subset of functionality that would allow another party to implement CONIKS at scale.

Now, another solution to the problem our work aims to solve is Google's implementation of a merkle tree, Trillian. This work serves a similar purpose as our merkle tree. Trillian implements a merkle tree with a restricted API to allow Google to implement their own version of CONIKS at scale known simply as KeyTransparency [1]. Apart from obvious differences with our work, like the lack of Ray and concurrency,

Trillian differs from our work in how they organize there data storage layer. While our data storage layer is meant to be flexible and "plug-and-play", Trillian's data storage layer was designed specifically for its usage. In short, Google does not store an entire tree in their database. Instead, they break up their tree into subtrees. Note, each of these subtrees is of some depth that is a power of two. Moreover, the nodes in each of these subtrees are not explicitly stored in the database. The novelty in this data storage layer is that the only nodes that are explicitly stored in the underlying database are the leaves of the subtree. Therefore, subtrees are recalculated after loading the leaves of a subtree into main memory from the database. The idea behind this is simple: the cost of recomputing nodes in a subtree is less than the cost of I/O [5].

X. Future Optimizations

There are some optimizations related to the storage layer that we find necessary in the future to boost performance. The first idea we have is to encode 256-bit hash digests in hexadecimal, which should reduce the digest size to 16 bytes, giving a 16x reduction in size from the current storage format as 256 bytes. Another performance hit we take currently is in the preloading of the cache from the database before performing our percolation algorithm. This cache is cleared everytime a batch is sent to the worker node but ideally, the cache's elements, or at least some of the higher level elements in the cache would be maintained for the next run so that we can pull in fewer nodes from the database. Since all the worker subtrees are completely isolated, we can be sure that the latest write is what should be pulled into the cache. Some more advanced database optimizations could be made on our queries such that we could perform reads and writes for larger numbers of nodes. Specifically, MySQL has a limit on the number of placeholders that may be placed in a prepared statement and also has a limit on the size of packets sent to the database so being free of those limitations would make hypertuning for optimal batch read and batch write sizes easier. The database should also be sharded and partitioned for more efficient querying. Ideally, since when writing, we always have a new epoch number, it would be ideal to have read-optimized database instances for data that is behind the current epoch and a small efficient write storage for the current epoch that gets pushed to the read instance efficiently. Becuase epochs grow without bound, it would be helpful to clean up old epochs if storage is an issue.

XI. Future Work

Currently, the primary queries we support are single-leaf writes that arrive in batches and single-leaf reads. We plan to extend our functionality to support range queries and multi-leaf updates and reads [6]. Both range and multi-leaf reads will be straightforward extensions of the single-read function since one worker node is always in charge of completing the entire read request. Multi-leaf updates are currently supported atomically within an epoch. That is, Angela currently guarantees that either all of the transactions, or none of the transactions, are persisted within an epoch. We plan to add finer grained logging of transactions in our orchestrator node and as a result, have finer grained atomicity guarantees on multi-leaf write transactions. The Go implementation of our batch inserts would indicate to the Ray actor which specific batches and those batches could be redone. Additionally, having multiple orchestrator nodes to load balance incoming transactions to worker nodes will remove this bottleneck. Our final planned addition to the server layer is to support smaller batching within an epoch to get streaming-like behavior on our transactions [2]. Such behavior could also be used to dynamically determine an optimal number of worker nodes to spawn to perform the writes.

XII. Conclusion

We have presented Angela, a sparse distributed merkle tree that is optimized for highly concurrent reads and writes. Angela is implemented with a concurrency algorithm that achieves a 2 times speedup from the naive algorithm. Angela's infrastructure is easily portable, and allows easy exchange of the database and worker nodes to match the user's needs. In addition, Angela is highly scalable and distributes easily while maintaining its multithreading advantages.

XIII. Acknowledgements

We would like to thank Professor John Kubiatowicz for general advising on the project, Richard Liaw from RISELab for his invaluable help with Ray, and Professor Raluca Ada Popa for discussing our approaches to the project with us.

References

[1] Yahoo! Inc. Antonio Marcedone Google Inc. *KeyTransparency*. URL: `https://github.com/google/keytransparency/`.

[2] Arvind Arasu et al. “Concerto: A High Concurrency Key-Value Store with Integrity”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD ’17. Chicago, Illinois, USA: ACM, 2017, pp. 251–266. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3064030. URL: http://doi.acm.org/10.1145/3035918.3064030.

[3] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. *Efficient Sparse Merkle Trees: Caching Strategies and Secure (Non-)Membership Proofs*. Cryptology ePrint Archive, Report 2016/683. https://eprint.iacr.org/2016/683. 2016.

[4] Saba Eskandarian et al. “Certificate Transparency with Privacy”. In: *CoRR* abs/1703.02209 (2017). arXiv: 1703.02209. URL: http://arxiv.org/abs/1703.02209.

[5] Antonio Marcedone et al. *Trillian*. https://eprint.iacr.org/2014/1004.

[6] Charles Martel et al. “A General Model for Authenticated Data Structures”. In: *Algorithmica* 39.1 (Jan. 2004), pp. 21–41. ISSN: 0178-4617. DOI: 10.1007/s00453-003-1076-8. URL: http://dx.doi.org/10.1007/s00453-003-1076-8.

[7] Marcela S. Melara et al. *CONIKS: Bringing Key Transparency to End Users*. Cryptology ePrint Archive, Report 2014/1004. https://eprint.iacr.org/2014/1004. 2014.

[8] Ralph C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology — CRYPTO ’87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3.

[9] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI Applications”. In: *CoRR* abs/1712.05889 (2017). arXiv: 1712.05889. URL: http://arxiv.org/abs/1712.05889.

[10] Maithili Narasimha and Gene Tsudik. “Authentication of Outsourced Databases Using Signature Aggregation and Chaining”. In: *Proceedings of the 11th International Conference on Database Systems for Advanced Applications*. DASFAA’06. Singapore: Springer-Verlag, 2006, pp. 420–436. ISBN: 3-540-33337-1, 978-3-540-33337-1. DOI: 10.1007/11733836_30. URL: http://dx.doi.org/10.1007/11733836_30.

[11] Roberto Tamassia and Nikos Triandopoulos. “Efficient Content Authentication in Peer-to-Peer Networks”. In: *Proceedings of the 5th International Conference on Applied Cryptography and Network Security*. ACNS ’07. Zhuhai, China: Springer-Verlag, 2007, pp. 354–372. ISBN: 978-3-540-72737-8. DOI: 10.1007/978-3-540-72738-5_23. URL: http://dx.doi.org/10.1007/978-3-540-72738-5_23.

[12] Alexandre Verbitski et al. “Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD ’17. Chicago, Illinois, USA: ACM, 2017, pp. 1041–1052. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3056101. URL: http://doi.acm.org/10.1145/3035918.3056101.