vCorfu: A Cloud-Scale Object Store on a Shared Log

Michael Wei[★†], Amy Tai[◇†], Christopher J. Rossbach[■†], Ittai Abraham[†], Maithem Munshed[‡],
Medhavi Dhawan[‡], Jim Stabile[‡], Udi Wieder[†], Scott Fritchie[†],
Steven Swanson[★], Michael J. Freedman[◇], Dahlia Malkhi[†]

[†]VMware Research Group, [‡]VMware,
[★]University of California, San Diego, [◇]Princeton University, [■]UT Austin

Abstract

This paper presents vCorfu, a strongly consistent cloud-scale object store built over a shared log. vCorfu augments the traditional replication scheme of a shared log to provide fast reads and leverages a new technique, *composable state machine replication*, to compose large state machines from smaller ones, enabling the use of state machine replication to be used to efficiently in huge data stores. We show that vCorfu outperforms Cassandra, a popular state-of-the art NoSQL stores while providing strong consistency (*opacity*, *read-own-writes*), efficient transactions, and global snapshots at cloud scale.

1 Introduction

Most data stores make a trade-off between *scalability*, or the ability of a system to be resized to meet the demands of a workload and *consistency*, which requires that operations on a system return predictable results. The proliferation of cloud services has led developers to insist on scalable data stores. To meet that demand, a new class of data stores known as NoSQL emerged which partition data, favoring scalability over strong consistency guarantees. While partitioning enables NoSQL stores operate at cloud-scale, it makes operations that are simple in traditional data stores (e.g. modifying multiple items atomically) difficult if not impossible in NoSQL stores.

Systems based on distributed shared logs [9, 10, 11, 40, 41] can address the scalability–consistency tradeoff. Instead of partitioning based on data contents as NoSQL stores do, these systems employ state machine replication (SMR) [27] and achieve scale-out by partitioning based on the order of updates. Since the log provides a single source of ground truth for ordering, shared logs offer a number of attractive properties such as strong consistency and global snapshots.

Shared logs, however, are not without drawbacks. In contrast to NoSQL, clients cannot simply "read" the latest data: the log only stores updates. Instead, clients *play* the log, reading and processing the log sequentially to update their own in-memory views. Playback can easily become a bottleneck: a client may process many updates which are irrelevant to the servicing of a request, dramatically increasing latencies when the system is under load. Figure 1 shows an example in which a client interested in reading Alice's last login time must read updates to other users to find and retrieve the most recent login. As a result, many shared log systems either target metadata services [10] with minimal state and client load, or delegate playback to an intermediate server [9, 11], further increasing latency.

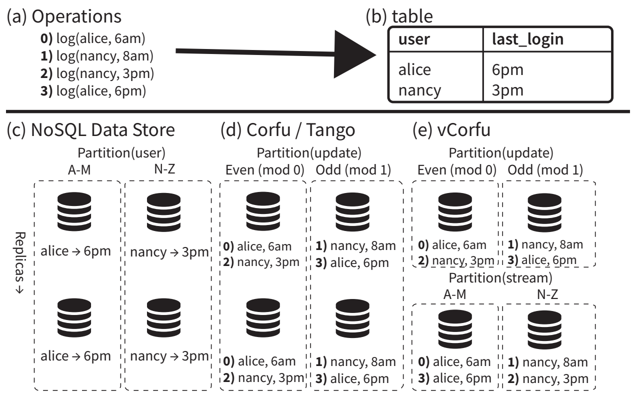

Figure 1: Physical layout of (a) operations on a (b) table stored in a (c) NoSQL data store (d) Shared log systems [9, 10] and (e) vCorfu.

This paper presents vCorfu, which makes distributed, shared log based systems applicable to a much broader array of settings by combining the consistency benefits of a shared log like Corfu [9] and Tango [10] with the locality advantages of scattered logs like Kafka [26] and Kinesis [30]. vCorfu is a cloud-scale, distributed object store. At the core of vCorfu is a scalable, virtualized shared log. The key innovation in vCorfu's log is *materialization*, which divides a single log into virtual logs called *materialized streams*. Unlike *streams* proposed in previous systems [10], which support only sequential reads, materialized streams support fast, fully random reads, because all updates for a stream can be accessed from a single partition. This design enables log replicas to use SMR to service requests directly, relieving the burden of playback from clients. Like other shared log systems, vCorfu supports strong consistency, linearizable reads and transactions, but the locality advantages of materialization enable vCorfu to scale to thousands of clients. vCorfu also leverages a sequencer to implement a fast, lightweight transaction manager and can execute read-only transactions without introducing conflicts. A novel technique called *composable state machine replication* (CSMR) enables vCorfu to store huge objects while still allowing client queries to be expressed using a familiar object-based model.

We make the following contributions:

- We present the design and architecture of vCorfu, a cloud-scale distributed object store built on a shared log. vCorfu's novel materialization technique enables reads without playback while maintaining strong consistency and high availability.
- We show that vCorfu's innovative design provides the same strong consistency guarantees as shared log designs while enabling scalability and performance that is competitive with, and often better than current NoSQL systems.
- We demonstrate that by conditionally issuing tokens, our sequencer performs lightweight transaction resolution, relieving clients of the burden of resolving transactions.
- We evaluate vCorfu against a popular NoSQL store, Cassandra, and show that vCorfu is just as fast for writes and much faster at reads, even while providing stronger consistency guarantees and advanced features such as transactions.
- We describe CSMR, a technique which enables efficient storage of huge objects by composition of a large state machine from smaller component state machines. vCorfu can store and support operations against 10GB YCSB! [16] database without sacrificing the strong consistency afforded by SMR.

2 Background

2.1 Data Stores

Modern web applications rely heavily on multi-tiered architecture to enable systems in which components may be scaled or upgraded independently. Traditional architectures consist of three layers: a front-end which communicates to users, an application tier with stateless logic, and a data tier, where state is held. This organization enabled early web applications to scale easily because stateless front-end and application tiers enable scaling *horizontally* in the application tier with the addition of more application servers or *vertically* in the data tier by upgrading to more powerful database servers.

As more and more applications move to cloud execution environments, system and application designers face increasingly daunting scalability requirements in the common case. At the same time, the end of Dennard scaling [21] leaves system builders unable to rely on performance improvements from the hardware: vertical scaling at the data tier is no longer feasible in most settings. As a consequence, modern cloud-scale systems generally trade off reduced functionality and programmability for scalability and performance at the data tier. A new class of NoSQL data stores [1, 4, 12, 14, 18, 26] has emerged, which achieve cloud-scale by relaxing consistency, eliding transaction support, and restricting query and programming models.

A severe consequence of this trend is an increased burden on programmers. In practice, programmers of modern cloud systems are forced to cobble together tools and components to restore missing functionality when it is needed. For example, a lock server such as ZooKeeper [23] is often used in conjunction with a NoSQL store to implement atomic operations. Programmers commonly implement auxiliary indexes to support queries, typically with relaxed consistency since the auxilliary index is not maintained by the data store.

2.2 Scalable Shared Logs

Shared logs have been used to provide highly fault-tolerant distributed data stores since the 1980s [36, 38]. Logs are an extremely powerful tool for building strongly consistent systems, since data is never overwritten, only appended, which yields a total order over concurrent modifications to the log. Early shared logs had limited scalability, as all appends must be serialized through a single server, quickly becoming an I/O bottleneck.

More recent shared log designs [9, 10, 11, 40, 41] address this scalability limitation to varying degrees. For example, the Corfu protocol [9] leverages a centralized sequencer which is not part of the I/O path, yielding a design in which append throughput is only limited by the speed in which a sequencer can issue log addresses.

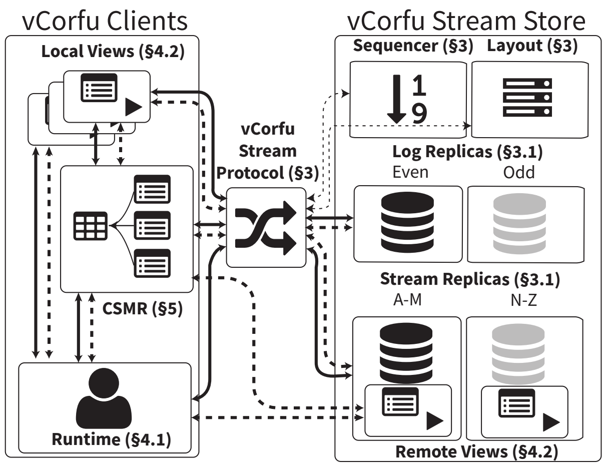

Figure 2: The architecture of vCorfu. Solid lines highlight the write path, while dotted lines highlight the read path. Thin lines indicate control operations outside of the I/O path.

Operation	Description
`read(laddresses)`	Get the data stored at *log address(es)*.
`read(stream, saddresses)`	Read from a *stream* at *stream address(es)*.
`append(stream, data)`	Append *data* to *stream*.
`check(stream)`	Get the last address issued to a *stream*.
`trim(stream, saddresses, prefix)`	Release all entries with $stream\ address < prefix$.
`fillhole(laddress)`	Invoke hole-filling for *log address*.

Table 1: Core operations supported by the vCorfu shared log.

2.3 State Machine Replication

Most shared log systems use state machine replication (SMR) [37] which relies on the log's total ordering of appends to implement an abstraction layer over the log. Data stored on the log is modeled as a state machine. Clients modify data by appending updates to the log and read data by traversing the log and applying those updates in order to an in-memory view. This approach enables strong consistency, and significantly simplifies support for transactions over multiple data items [10, 11].

The achilles' heel of shared log systems, however, is playback. To service *any* request, a client must read every single update and apply it to in-memory state, regardless of whether the request has any dependency on those updates. In practice, this has limited the applicability of shared log systems to settings characterized by few clients or small global state [9], such as metadata services [10, 40]. In contrast, data tiers in typical web applications manage state at a scale that may make traditional playback prohibitively expensive. Worse, in systems relying on stateless application tiers, naïve use of shared logs induces a playback requirement to reconstruct state for every request. The goal of vCorfu is to eliminate these limitations, enabling SMR with shared logs over large state and without client playback overheads.

3 vCorfu Stream Store

vCorfu implements a shared log abstraction that removes the overhead and limitations of shared logs, enabling playback that does not force a client to playback potentially irrelevant updates. vCorfu virtualizes the log using a novel technique called *stream materialization*. Unlike streams in Tango, which are merely tags within a shared log, *materialized streams* are a first class abstraction which supports random and bulk reads just like scattered logs like Kafka [26] and Kinesis [30], but with all the consistency benefits of a shared log like Corfu [9] and Tango [10].

The vCorfu stream store architecture is shown in Figure 2. In vCorfu, data are written to materialized streams, and data entries receive monotonically increasing tokens on both a global log and on individual streams from a *sequencer* server. The sequencer can issue tokens *conditionally* to enable fast optimistic transaction resolution, as described in Section 4. vCorfu writes data in the form of updates to both *log replicas* and *stream replicas*, each of which are indexed differently. This design replicates data for durability, but enables access to that data with different keys, similar to Replex [39]. The advantage is that clients can directly read the latest version of a stream simply by contacting the stream replica.

A *layout* service maintains the mapping from log and stream addresses to replicas. Log replicas and stream replicas in vCorfu contain different sets of updates, as shown in Figure 1. The log replicas store updates by their (global) log address, and stream replicas by their stream addresses. The replication protocol in vCorfu dynamically builds replication chains based on the global log offset, the streams which are written to, and the streams offsets. Subsequent sections consider the design and implementation of materialized streams in more detail.

vCorfu is elastic and scalable: replicas may be added or removed from the system at any time. The sequencer, because it merely issues tokens, does not become an I/O bottleneck. Reconfiguration is triggered simply by changing the active layout. Finally, vCorfu is *fault tolerant* - data which is stored in vCorfu can tolerate a limited number of failures based on the arrangement and number of replicas in the system, and recovery is handled similar to the mechanism in Replex [39]. Generally, vCorfu can tolerate the failures as long as a log replica and stream replica do not fail simultaneously. Stream replicas can be reconstructed from the aggregate of the log replicas, and log replicas can be reconstructed by scanning through all stream replicas.

Operationally, stream materialization divides a single

```
"sequencers": 10.0.0.1,
"segments": {
 "start" : 0,
 "log" : [[ 10.0.1.1 ], [ 10.0.1.2 ]],
 "stream" : [[ 10.0.2.1 ], [ 10.0.2.2 ]] ] }
```

Figure 3: An example layout. Updates are partitioned by their stream id and the log offset; a simple partitioning function mods these values with respect to the number of replicas. An update to stream 0 at log address 1 would be written to 10.0.1.2 and 10.0.2.1, while an update to stream 1 at log address 3 would be written to 10.0.1.2 and 10.0.2.2.

global log into materialized streams, which support logging operations: append, random and bulk reads, trim, check and fillhole; the full API is shown in Table 1. Each materialized stream maps to an *object* in vCorfu, and each stream stores an ordered history of modifications to that object, following the SMR [37] paradigm.

3.1 Fully Elastic Layout

In vCorfu, a mapping called a layout describes how offsets in the global log or in a given materialized stream map to replicas. A vCorfu client runtime must obtain a copy of the most current layout to determine which replica(s) to interact with. Each layout is stamped with an *epoch* number. Replicas will reject requests from clients with a stale epoch. A Paxos-based protocol [27] ensures that all replicas agree on the current layout. An example layout is shown in Figure 3. Layouts work like leases on the log: a client request with the wrong layout (and wrong epoch number) will be rejected by replicas. The layout enables clients to safely contact a stream replica directly for the latest update to a stream.

3.2 Appending to vCorfu materialized streams

A client appending to a materialized stream (or streams) first obtains the current layout and makes a request to the sequencer with a *stream id*. The sequencer returns both a *log token*, which is a pointer to the next address in the global log, and a *stream token*, which is a pointer to the next address in the stream. Using these tokens and the layout, the client determines the set of replicas to write to.

In contrast to traditional designs, replica sets in vCorfu are dynamically arranged during appends. For fault tolerance, each entry is replicated on two replica types: the first indexed by the address in the log (the *log replica*), and the second by the combination of the stream id and the stream address (the *stream replica*). To perform a write, the client writes to the log replica first, then to the stream replica. If a replica previously accepted a write to a given address, the write is rejected and the client must retry with a new log token. Once the client

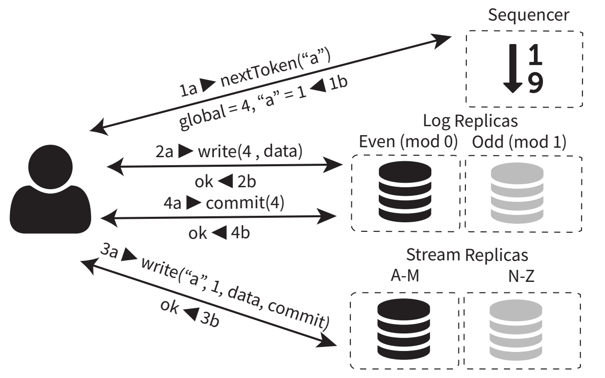

Figure 4: Normal write path of a vCorfu log write, which takes four roundtrips: one for token acquisition, two for writing to each replica (and committing at the stream replica), and one to send a commit message to the log replica.

writes to both replicas, it commits the write by broadcasting a commit message to each replica it accessed (except the final replica, since the final write is already committed). Replicas will only serve reads for committed data. This enables stream replicas to provide a dense materialized stream, without holes. The write path of a client, which takes four roundtrips in normal operation is shown in Figure 4. A server-driven variant where the log replica writes to the stream replica takes 6 messages; we leave implementation of this variant for future work.

3.3 Atomically appending to multiple streams

The primary benefit of materialized streams is that they provide an abstraction of independent logs while maintaining a total global order over all appends. This enables vCorfu to support atomic writes across streams, which form the basic building block for supporting transactions.

To append to multiple streams atomically, the client obtains a log token and stream tokens for each stream it wishes to append to. The client first writes to the log replica using the log token. Then, the client writes to the stream replica of each stream (multiple streams mapped to the same replica are written together so each replica is visited only once). The client then sends a commit message to each participating replica (the commit and write are combined for the last replica in the chain). The resulting write is ordered in the log by a single log token, but multiple stream tokens.

3.4 Properties of the vCorfu Stream Store

Materialized streams are a first class abstraction in vCorfu, unlike *streams* in Tango [10] which are merely tags within a shared log. Materialized streams strike a balance that combines the global consistency advantages of shared logs with the locality advantages of distributed data platforms. Specifically, these properties enable vCorfu to effectively support SMR at scale:

The global log is a single source of scalability, consistency, durability and history. One may wonder, why have log replicas at all, if all we care to read from are materialized streams? First, the global log provides a convenient, scalable mechanism to obtain a consistent snapshot of the entire system. This can be used to execute long running read-only transactions, a key part of many analytics workloads, or a backup utility could constantly scan the log and move it to cold storage. Second, the log provides us with a unique level of fault tolerance - even if all the stream replicas were to fail, vCorfu can fall back to using the log replicas only, continuing to service requests.

Materialized streams are true virtual logs, unlike streams. Tango streams enable clients to selectively consume a set of updates in a shared log. Clients read sequentially from streams using a `readNext()` call, which returns the next entry in the stream. Tango clients cannot randomly read from anywhere in stream because streams are implemented using a technique called *backpointers*: each entry in a stream points to the previous entry, inducing a requirement for sequential traversal. Materializing the stream removes this restriction: since clients have access to a replica which contains all the updates for a given stream, clients can perform all the functions they would call on a log, including a random read given a stream address, or a bulk read of an entire stream. This support is essential if clients randomly read from different streams, as backpointers would require reading each stream from the tail in order.

vCorfu avoids backpointers, which pose performance, concurrency and recovery issues. Backpointers can result in performance degradation when concurrent clients are writing to the log and a timeout occurs, causing a *hole filling protocol* to be invoked [9]. Since holes have no backpointers, timeouts force a linear scan of the log, with a cost proportional to the number of streams in the log. Tango mitigates this problem by keeping the number of streams low and storing multiple backpointers, which has significant overhead because the sequencer must maintain a queue for each stream. Furthermore, backpointers significantly complicate recovery: if the sequencer fails, the entire log must be read to determine the most recent writes to each stream. vCorfu instead relies on stream replicas, which contain a complete copy of updates for each stream, free of holes thanks to vCorfu's commit protocol, resorting to a single backpointer only when stream replicas fail. Sequencer recovery is fast, since stream replicas can be queried for the most recent update.

Stream replicas may handle playback and directly serve requests. In most shared log designs, clients must consume updates, which are distributed and sharded for performance. The log itself cannot directly serve requests because no single storage unit for the log contains all the updates necessary to service a request. Stream replicas in vCorfu, however, contain all the updates for a particular stream, so a stream replica can playback updates locally and directly service requests to clients, a departure from the traditional client-driven shared log paradigm. This removes the burden of playback from clients and avoids the playback bottleneck of previous shared log designs [10, 11].

Garbage collection is greatly simplified. In Tango, clients cannot trim (release entries for garbage collection) streams directly. Instead, they must read the stream to determine which log addresses should be released, and issue trim calls for each log address, which can be a costly operation if many entries are to be released. In vCorfu, clients issue trim commands to stream replicas, which release storage locally and issue trim commands to the global log. Clients may also delegate the task of garbage collection directly to a stream replica.

4 The vCorfu Architecture

vCorfu presents itself as an object store to applications. Developers interact with objects stored in vCorfu and a client library, which we refer to as the vCorfu runtime, provides consistency and durability by manipulating and appending to the vCorfu stream store. Today, the vCorfu runtime supports Java, but we envision supporting many other languages in the future.

The vCorfu runtime is inspired by the Tango [10] runtime, which provides a similar distributed object abstraction in C++. On top of the features provided by Tango, such as *linearizable reads* and transactions, vCorfu leverages Java language features which greatly simplify writing vCorfu objects. Developers may store arbitrary Java objects in vCorfu, we only require that the developer provide a serialization method and to annotate the object to indicate which methods read or mutate the object, as shown in Figure 5.

Like Tango, vCorfu fully supports transactions over objects with stronger semantics than most distributed data stores, thanks to inexpensive global snapshots provided by the log. In addition, vCorfu also supports transactions involving objects not in the runtime's local memory (case D, §4.1 in [10]), *opacity* [22], which ensures that transactions never observe inconsistent state, and *read-own-writes* which greatly simplifies concurrent programming. Unlike Tango, the vCorfu runtime never

```
class User {
    String name; String password;
    DateTime lastLogin; DateTime lastLogout;

    @Accessor
    public String getName() {
        return name;}

    @MutatorAccessor
    public boolean login(String pass, DateTime time){
      if (password.equals(pass)) {
      lastLogin = time;
      return true;}
   return false;}

    @Mutator
    public void logout(DateTime time) {
      lastLogout = time;}}
```

Figure 5: A Java object stored in vCorfu. @Mutator indicates that the method modifies the object, @Accessor indicates the method reads the object, and @MutatorAccessor indicates the object reads and modifies the object.

needs to resolve whether transactional entries in the log have succeeded thanks to a lightweight transaction mechanism provided by the sequencer.

4.1 vCorfu Runtime

To interact with vCorfu as an object store, clients load the vCorfu runtime, a library which manages interactions with the vCorfu stream store. Developers never interact with the store directly, instead, the runtime manipulates the store whenever an object is accessed or modified. The runtime provides each client with a view of objects stored in vCorfu, and these views are synchronized through the vCorfu stream store.

The runtime provides three functions to clients: `open()`, which retrieves a in-memory view of an object stored in the log, `TXbegin()`, which starts a transaction, and `TXend()`, which commits a transaction.

4.2 vCorfu Objects

As we described earlier, vCorfu objects can be arbitrary Java objects such as the one shown in Figure 5. Objects map to a stream, which stores updates to that object.

Like many shared log systems, we use state machine replication (SMR) [27] to provide strongly consistent accesses to objects. When a method annotated with @Mutator or @MutatorAccessor is called, the runtime serializes the method call and appends it to the objects' stream first. When an @Accessor or @MutatorAccessor is called, the runtime reads all the updates to that stream, and applies those updates to the object's state before returning. In order for SMR to work, each mutator must be deterministic (a call to `random()` or `new Date()` is not supported). Many method calls can be easily refactored to take non-deterministic calls as a parameter, as shown in the `login` method in Figure 5.

The SMR technique extracts several important properties from the vCorfu stream store. First, the log acts as a source of *consistency*: every change to an object is totally ordered by the sequencer, and every access to an object reflects all updates which happen before it. Second, the log is a source of *durability*, since every object can be reconstructed simply by playing back all the updates in the log. Finally, the log is a source of *history*, as previous versions of the object can be obtained by limiting playback to the desired position.

Each object can be referred to by the id of the stream it is stored in. Stream ids are 128 bits, and we provide a standardized hash function so that objects can be stored using human-readable strings (i.e., "person-1").

vCorfu clients call `open()` with the stream id and an object type to obtain a view of that object. The client also specifies whether the view should be *local*, which means that the object state is stored in-memory locally, or *remote*, which means that the stream replica will store the state and apply updates remotely (this is enabled by the remote class loading feature of Java). Local views are similar to objects in Tango [10] and especially powerful when the client will read an object frequently throughout the lifespan of a view: if the object has not changed, the runtime only performs a quick `check()` call to verify no other client has modified the object, and if it has, the runtime applies the relevant updates. Remote views, on the other hand, are useful when accesses are infrequent, the state of the object is large, or when there are many remote updates to the object - instead of having to playback and store the state of the object in-memory, the runtime simply delegates to the stream replica, which services the request with the same consistency as a local view. To ensure that it can rapidly service requests, the stream replicas generate periodic checkpoints. Finally, the client can optionally specify a maximum position to open the view to, which enables the client to access the history, version or *snapshot* of an object. Clients may have multiple views of the same object: for example, a client may have a local view of the present state of the object with a remote view of a past version of the object, enabling the client to operate against a snapshot.

4.3 Transactions in vCorfu

Transactions enable developers to issue multiple operations which either succeed or fail atomically. Transactions are a pain point for partitioned data stores since a transaction may span across multiple partitions, requiring locking or schemes such as 2PL [32] or MVCC [35] to achieve consistency.

vCorfu leverages atomic multi-stream appends and global snapshots provided by the log, and exploits the sequencer as a lightweight transaction manager. Transaction execution is optimistic, similar to transactions in shared log systems [10, 11]. However, since our sequencer supports conditional token issuance, we avoid polluting the log with transactional aborts.

To execute a transaction, a client informs the runtime that it wishes to enter a transactional context by calling `TXBegin()`. The client obtains the most recently issued log token once from the sequencer and begins optimistic execution by modifying reads to read from a snapshot at that point. Writes are buffered into a write buffer. When the client ends the transaction by calling `TXEnd()`, the client checks if there are any writes in the write buffer. If there are not, then the client has successfully executed a read-only transaction and ends transactional execution. If there are writes in the write buffer, the client informs the sequencer of the log token it used and the streams which will be affected by the transaction. If the streams have not changed, the sequencer issues log and stream tokens to the client, which commits the transaction by writing the write buffer. Otherwise, the sequencer issues no token and the transaction is aborted by the client without writing an entry into the log. This important optimization ensures only committed entries are written, so that when a client encounters a transactional commit entry, it may treat it as any other update. In other shared log systems [10, 11, 40], each client must determine whether a commit record succeeds or aborts, either by running the transaction locally or looking for a decision record. In vCorfu, we have designed transactional support to be as general as possible and to minimize the amount of work that clients must perform to determine the result of a transaction. We treat each object as an opaque object, since fine-grained conflict resolution (for example, determining if two updates to different keys in a map conflict) would either require the client resolve conflicts or a much more heavyweight sequencer.

Opacity is ensured by always operating against the same global snapshot, leveraging the history provided by the log. Opacity [22] is a stronger guarantee than strict serializability as opacity prevents programmers from observing inconsistent state (e.g. a divide-by-zero error when system invariants prevent such a state from occuring). Since global snapshots are expensive in partitioned systems, these systems [1, 2, 3, 4] typically provide only a weaker guarantee, allowing programs to observe inconsistent state but guaranteeing that such transactions will be aborted. Read-own-writes is another property which vCorfu provides: transactional reads will also apply any writes in the write buffer. Many other systems [1, 4, 10] do not provide this property since it requires writes to be applied to data items. The SMR paradigm, however, enables vCorfu to generate the result of a write in-memory, simplifying transactional programming.

vCorfu fully supports *nested transactions*, where a transaction may begin and end within a transaction. Whenever transaction nesting occurs, vCorfu buffers each transaction's write set and the transaction takes the timestamp of the outermost transaction.

4.4 Querying Objects

vCorfu supports several mechanisms for finding and retrieving objects. First, a developer can use vCorfu like a traditional key-value store just by using the stream id for object as a key. We also support a much richer query model: a set of collections, which resemble the Java collections are provided for programmers to store and access objects in. These collections are objects just like any other vCorfu object, so developers are free to implement their own collection. Developers can take advantage of multiple views on the same collection: for instance a `List` can be viewed as a `Queue` or a `Stack` simultaneously. Some of the collections we provide include a `List`, `Queue`, `Stack`, `Map`, and `RangeMap`.

Collections, however, tend to be very large objects which are highly contended. In the next section, we discuss composable state machine replication, a technique which allows vCorfu to build a collection out of multiple objects.

5 Composable State Machine Replication

In vCorfu, objects may be composed of other objects, a technique which we refer to as composable state machine replication (CSMR). The simplest example of CSMR is a hash map composed of multiple hash maps, but much more sophisticated objects can be created.

Composing SMR objects has several important advantages. First, CSMR divides the state of a single object into several smaller objects, which reduces the amount of state stored at each stream. Second, smaller objects reduce contention and false sharing, providing for higher concurrency. Finally, CSMR resembles how data structures are constructed in memory - this allows us to apply standard data structure principles to vCorfu. For example, a B-tree constructed using CSMR would result in a structure with $O(\log n)$ time complexity for search, insert and delete operations. This opens a plethora of familiar data structures to developers.

Programmers manipulate CSMR objects just as they would any other vCorfu object. A CSMR object starts

```
class CSMRMap<K,V> implements Map<K,V> {
    final int numBuckets;

    int getChildNumber(Object k) {
      int hashCode = lubyRackoff(k.hashCode());
      return Math.abs(hashCode % numBuckets);}

    SMRMap<K,V> getChild(int partition) {
      return open(getStreamID() + partition);}

    V get(K key) {
    return getChild(getChildNumber(key)).get(key);}

    @TransactionalMethod(readOnly = true)
    int size() {
      int total = 0;
      for (int i = 0; i < numBuckets; i++) {
      total += getChild(i).size();}
   return total;}

    @TransactionalMethod
    void clear() {
      for (int i = 0; i < numBuckets; i++) {
      total += getChild(i).clear();}}}
```

Figure 6: A CSMR Java Map in vCorfu. @TransactionalMethod indicates that the method must be executed transactionally.

with a *base object*, which defines the interface that a developer will use to access the object. An example of a CSMR hash map is shown in Figure 6. The base object manipulates *child objects*, which store the actual data. Child objects may reuse standard vCorfu objects, like a hash map, or they may be custom-tailored for the CSMR object, like a B-tree node.

In the example CSMR map shown in Figure 6, the object shown is the base object and the child objects are standard SMR maps (backed by a hash map). The number of buckets is set at creation in the `numBuckets` variable. Two functions, `getChildNumber()` and `getChild()` help the base object locate child objects deterministically. In our CSMR map, we use the Luby-Rakoff [28] algorithm to obtain an improved key distribution over the standard Java `hashCode()` function. Most operations such as `get` and `put` operate as before, and the base object needs to only select the correct child to operate on. However, some operations such as `size()` and `clear()` touch all child objects. These methods are annotated with @TransactionalObject so that under the hood, the vCorfu runtime uses transactions to make sure objects are modified atomically and read from a consistent snapshot. The vCorfu log provides fast access to snapshots of arbitrary objects, and the ability to open remote views, which avoids the cost of playback, enables clients to quickly traverse CSMR objects without reading many updates or storing large local state.

In a more complex CSMR object, such as our CSMR B-tree, the base object and the child object may have completely different interfaces. In the case of the B-tree, the base object presents a map-like interface, while the child objects are nodes which contain either keys or references to other child objects. Unlike a traditional B-tree, every node in the CSMR B-tree is versioned like any other object in vCorfu. CSMR takes advantage of this versioning when storing a reference to a child object: instead of storing a static pointer to particular versions of node, as in a traditional B-tree, references in vCorfu are dynamic. Normally, references point to the latest version of an object, but they may point to any version during a snapshotted read, allowing the client to read a consistent version of even the most sophisticated CSMR objects. With dynamic pointers, all pointers are implicitly updated when an object is updated, avoiding a problem in traditional trees, where an update to a single child node can cause an update cascade requiring all pointers up to the root to be explicitly updated, known as the recursive update problem [42].

6 Evaluation

Our test system consists of sixteen 12 core machines running Linux (v4.4.0-38) with 96GB RAM and 10G NICs on each node with a single switch. The average latency measured by `ping` (56 data bytes) between two hosts is 0.18±0.01 ms when the system is idle. All benchmarks are done in-memory, with persistence disabled. Due to the performance limitations and overheads from Java and serialization, our system was CPU-bound and none of our tests were able to saturate the NIC (the maximum bandwidth we achieved from a single node was 1Gb/s, with 4KB writes).

Our evaluation is driven by the following questions:

- What advantages to we obtain by materializing streams? (§ 6.1)
- Do remote views offer NoSQL-like performance with the global consistency of a shared log? (§ 6.2)
- How does the sequencer act as a lightweight, lock-free transaction manager and offer inexpensive read-only transactions? (§ 6.3)
- How does CSMR keep state machines small, while reducing contention and false conflicts? (§ 6.4)

6.1 vCorfu Stream Store

The design of vCorfu relies on performant materialization. To show that materializing streams is efficient, we implement streams using backpointers in vCorfu with chain replication, similar to the implementation described in Tango [10].