Realizing the fault-tolerance promise of cloud storage using locks with intent

Srinath Setty Chunzhi Su* Jacob R. Lorch Lidong Zhou
Hao Chen* Parveen Patel Jinglei Ren

Microsoft Research

Abstract

Cloud computing promises easy development and deployment of large-scale, fault tolerant, and highly available applications. Cloud storage services are a key enabler of this, because they provide reliability, availability, and fault tolerance via internal mechanisms that developers need not reason about. Despite this, challenges remain for distributed cloud applications developers. They still need to make their code robust against failures of the machines running the code, and to reason about concurrent access to cloud storage by multiple machines.

We address this problem with a new abstraction, called *locks with intent*, which we implement in a client library called *Olive*. Olive makes minimal assumptions about the underlying cloud storage, enabling it to operate on a variety of platforms including Amazon DynamoDB and Microsoft Azure Storage. Leveraging the underlying cloud storage, Olive's locks with intent offer strong *exactly-once* semantics for a snippet of code despite failures and concurrent duplicate executions.

To ensure exactly-once semantics, Olive incurs the unavoidable overhead of additional logging writes. However, by decoupling isolation from atomicity, it supports consistency levels ranging from eventual to transactional. This flexibility allows applications to avoid costly transactional mechanisms when weaker semantics suffice. We apply Olive's locks with intent to build several advanced storage functionalities, including snapshots, transactions via optimistic concurrency control, secondary indices, and live table re-partitioning. Our experience demonstrates that Olive eases the burden of creating correct, fault-tolerant distributed cloud applications.

1 Introduction

Cloud platforms such as Amazon AWS, Google Cloud, and Microsoft Azure are becoming popular choices for deploying applications because they permit elastic scaling, handle various operational aspects, and offer high reliability and availability. As a common practice, cloud platforms offer reliable storage services with simple APIs that hide the distributed nature of the underlying storage. Application developers are thereby freed from handling distributed-systems issues such as data partitioning, fault tolerance, and load balancing. Examples include Amazon's DynamoDB [1], Google's Cloud Storage [8], and Microsoft's Azure Storage [3]. This has led to a new paradigm for architecting applications where compute and storage components of an application are separated: applications store data on cloud storage, and perform computation on a set of client virtual machines (VMs).

This emerging architecture for applications poses an interesting new problem: Although cloud storage is made reliable by cloud service providers via fault-tolerance protocols [38, 49], it does not completely solve the problem of maintaining application-level consistency in face of failures. After all, clients running an application can fail, application processes on those clients can crash, and the network connecting those clients to the underlying storage can drop or reorder messages. Such issues can potentially leave the underlying storage in an inconsistent state, or block progress of application processes on other clients.

This problem is made even more challenging by the fact that cloud storage services tend to offer limited, low-level APIs. For example, the Azure Table storage service allows atomic batch update only on objects in the same partition [7, 21]. Cloud providers offer such APIs to allow efficient storage implementations, to offer applications the freedom to choose the right balance between performance and consistency, and to help themselves internally manage complexity and operational challenges. However, a limited API makes it hard for programmers of cloud applications to reason about correctness, given that clients can issue concurrent storage operations and can fail.

We address this problem with a new abstraction called *locks with intent*. The key insight behind this abstraction is that much of the complexity in handling failures and concurrency can be encapsulated in a simple *intent* concept that can be used in conjunction with locks. An intent is an arbitrary snippet of code that can contain both cloud storage operations and local computation, but with a key property that, when an intent execution completes, each step in the intent is guaranteed to have executed *exactly once*, despite failures, recovery, or concurrent executions.

A lock with intent lets a client lock an object in cloud storage as long as it first provides an intent describing what it plans to do while holding the lock. Once locked, the intent gains exclusive access to the object, just as

*Work done during an internship at Microsoft Research. Chunzhi Su is affiliated with The University of Texas at Austin, and Hao Chen is affiliated with Shanghai Jiao Tong University.

a traditional lock in a shared memory model. However, unlike a traditional lock, a locked object will *eventually* be unlocked even if the client holding the lock crashes, as long as the application is deadlock-free. Furthermore, before the lock is unlocked, each step in the associated intent is guaranteed to have been executed exactly once.

We implement this abstraction in a client library called Olive. Olive's design and implementation makes minimal assumptions about the underlying storage, which it encapsulates in the form of a common storage model. This model fits many existing cloud storage services as well as other large-scale distributed storage systems such as Apache Cassandra and MongoDB. Thus, Olive can work with any such storage service unchanged by using a shim that translates the service's API to the model's API.

To provide exactly-once execution semantics, Olive leverages the underlying storage's fault-tolerance properties. It stores each intent, with a unique identifier, in the underlying cloud storage system itself. Olive further introduces *distributed atomic affinity logging* (DAAL). DAAL colocates the log entry that corresponds to executing an intent step with the object changed by that step.

Olive also includes mechanisms to ensure progress. Because Olive provides exactly-once semantics even if multiple clients concurrently execute the same intent, any client can acquire any locked object by executing the associated intent. To ensure liveness for all intents, not just those associated with locks other clients wish to acquire, Olive introduces a special process called an *intent collector* that periodically completes unfinished intents.

Using Olive's locks with intent, we implement several libraries that provide advanced features on top of cloud storage. These include consistent snapshots, live table repartitioning, secondary indices, and ACID transactions. Our experience with these case studies suggests that Olive significantly reduces the burden on programmers tasked with making code robust to failures and concurrency. Furthermore, Olive's well-defined semantics make it easy to reason about correctness of application code despite failures and concurrency (§4, §5).

Our work makes the following contributions:

- We propose locks with intent, a new abstraction to simplify handling failures and concurrency in cloud applications built atop cloud storage services.
- We introduce a novel logging scheme called distributed atomic affinity logging (DAAL), and the idea of an intent collector. Together, they ensure exactly-once semantics despite failures and/or multiple clients executing the same intent.
- We demonstrate the feasibility of locks with intent by implementing them in Olive and making Olive compatible with a variety of cloud storage services.
- We demonstrate the generality and usability of locks with intent by using them to build several useful libraries and reason about their correctness.
- We experimentally evaluate Olive on Microsoft's Azure Storage to determine the performance cost of using locks with intent compared to baselines providing similar fault-tolerance guarantees.

2 Building cloud applications: challenges

Cloud applications typically run on multiple client VMs and store state on cloud storage: the client VMs are used only for computation and are effectively stateless. Such applications are fundamentally distributed and must cope with distributed-systems challenges such as asynchrony, concurrency, failure, and scaling.

The underlying reliable distributed cloud storage aims to alleviate the difficulty of building cloud applications. Its API thus generally hides the complexity of concurrency control, elasticity, and fault tolerance. Nevertheless, the developer of a cloud application still has to handle VM failures. She must also bridge the gap between rich application semantics and the cloud storage's simple API.

2.1 A common storage model

Different cloud storage services offer different, constantly-evolving APIs. But, we want Olive to operate on any cloud service without requiring significant reworking each time a provider decides to make changes. Thus, we introduce a *common storage model*, an API that has enough features to support Olive but is simple enough to be implemented by any cloud storage service. In particular, it is easily implemented by popular cloud storage systems such as Microsoft's Azure tables and Amazon's DynamoDB, and by large-scale distributed storage systems such as Apache Cassandra and MongoDB. By stripping away functionality unique to certain services and focusing only on basic operations, we enable broad applicability for Olive.

Our model is that of a storage system providing schemaless tables. Each table row, also called an *object*, consists of a key and a set of attribute/value pairs. A table may be divided into *partitions* to satisfy a system-imposed limit on maximum partition size.

API. The model's API includes operations to `Create`, `Read`, `Update`, and `Delete` rows (CRUD). It also includes `Scan`, `UpdateIfUnchanged`, and `AtomicBatchUpdate`, described in the next paragraphs.

`Scan` takes a table and a predicate as parameters, and returns a stream providing all rows in that table satisfying that predicate. For instance, the predicate might be "has a count attribute with value > 5." Every row that satisfies the predicate throughout the scan is guaranteed to be included. A row that only satisfies the condition some time during the scan (e.g., because it was created, updated,

or deleted during the scan) may or may not be included.

`UpdateIfUnchanged` is like `Update`, except it does nothing if the object to be updated has been updated or deleted since a certain previous operation on that object. That previous operation is identified by a *handle* passed to `UpdateIfUnchanged`. The application can obtain such handles because each `Create`, `Read`, and `Update` operation returns a handle representing that operation.

`AtomicBatchUpdate` lets the application perform multiple update and insert operations atomically. In other words, despite possible failures, either all or none of the operations will happen. However, this atomicity guarantee only works at a certain granularity: objects passed to `AtomicBatchUpdate` must be in the same atomicity scope, where the scope is a system-specific parameter.

Such a storage model is supported not only by cloud storage services, such as Amazon DynamoDB (with rows as the atomicity scope) and Microsoft Azure table storage (with partitions as the atomicity scope), but also by popular storage systems, such as MongoDB (with documents as the atomicity scope) and Cassandra (with partitions as the atomicity scope). Azure Table supports ETags, which can be considered as handles; DynamoDB supports conditional update. MongoDB supports *Update if Current* and Cassandra supports the IF keyword in INSERT, UPDATE and DELETE statements for conditional updates, which can be considered as generalizations of the conditional update primitive in our model. Such common capabilities are chosen by different storage services because they provide simple and flexible primitives for concurrency-control and fault-tolerance support, and because they can be supported at a manageable cost and complexity. The cost and complexity consideration leads to a somewhat limited API. For example, Cassandra chooses to support partition-level atomicity because "the underlying Paxos implementation works at the granularity of the partition" [5].

Invisible entries. In our model, it is always possible to put *invisible* entries in a scope. That is, a library interposing on the API between the application and the cloud storage can put entries in a scope, but hide them from the application by stripping them from returned results. Even if the only scope available is an object, this can be done by adding special attributes to it. If scopes are larger, such as partitions or tables, the library can use special rows.

Invisible entries should be used sparingly since they reduce performance and capacity. They reduce performance when an access to a real entry necessitates one or more accesses to invisible entries. They reduce capacity by using space that could otherwise be used for application data. In particular, a cloud storage system often places an upper bound on the size of a scope, e.g., a maximum row or partition size. By using invisible entries, the interposing library reduces the effective maximum size from the application's perspective. Indeed, when the application asks for the maximum allowable size of a scope, the library must provide a lower number than the underlying storage system to account for this.

```
1  def updateObject(key, newObj):
2    obj = curTable.Read(key)
3    lastSnapshot = curTable.Read(LAST_SNAP).value
4    curEpoch = lastSnapshot + 1
5
6    if (obj != None and obj.version <= lastSnapshot):
7      snapshotTables[lastSnapshot].Update(key, obj)
8
9    newObj.version = curEpoch
10   curTable.UpdateIfUnchanged(key, newObj)
```

FIGURE 1—Pseudocode for the object-update routine in a buggy snapshot design. It sometimes requires extra work because a snapshot table is being lazily populated.

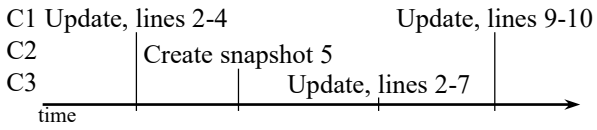

FIGURE 2—Execution trace exposing a bug in the Figure 1 code. Client C1, a slow updater, performs update lines 2–4, looking up a certain key and finding version #5. Since the last snapshot epoch is #4, the object is up to date and no copy-on-write is needed so it skips to line 9. Update lines 9–10, which happen later, update the current table with new contents but still version #5. Meanwhile, client C2 creates a new snapshot, and client C3 does an update involving a copy-on-write to snapshot table #5. At the end of this trace, the snapshot invariant is violated: snapshot table #5 contains contents that do not reflect the latest update, just performed by C1.

Lock. The primitives provided by the storage interface can be used to implement other useful client functionality. For example, we can implement an object lock by adding an invisible Boolean attribute called `locked` to the object. To acquire the lock, a client reads the object and gets a handle. If the `locked` bit is not set, the client issues a conditional update with the returned handle to set the `locked` bit. It gets the lock if and only if that update succeeds. To release the lock, the client resets the `locked` bit via another update.

2.2 A case study: supporting snapshots

As a case study, we present the example of supporting storage snapshots using the common storage model described earlier. A resulting *table-snapshotting* (or STable) service allows clients to create snapshots of a table without interrupting normal operations on the table, in addition to the standard CRUD operations. This is functionality we have actually designed and implemented for a production scenario.

To demonstrate how easy it is to accidentally introduce bugs when designing snapshot tables, we show one of

our earliest designs and the bug it contained. This design implements snapshotting tables directly on cloud storage, instead of using primitives like locks with intent.

Buggy STable design. In this design, each snapshot is implemented as an actual table. However, rather than fully populating this table when a snapshot is created, the table is populated lazily. This makes snapshot creation quick, which prevents snapshot creation from making the table unavailable for an extended period of time.

Snapshots are numbered in increasing order, with the first one being snapshot 1. Snapshots divide time into epochs, with epoch 1 preceding snapshot 1, epoch 2 coming between snapshots 1 and 2, etc. An invisible entry is put into each object to represent its *version*, defined as the last epoch it was updated in. An invisible entry is put into the table to represent the number of the last snapshot taken. The current epoch is one more than this number.

To lazily populate a snapshot, we use a snapshot-aware routine for updating objects, as shown in Figure 1. If it finds that the object version in the current table belongs in an earlier snapshot (i.e., smaller than the current snapshot number of the table), it copies the object to a snapshot table before overwriting it. This makes the current table essentially copy-on-write after a new snapshot is taken. A key *snapshot* invariant for STable is that, if snapshot table i has a row with key k, then that row contains the contents of the last update to key k made with version i.

Figure 2 illustrates an example execution demonstrating a bug in this design that violates the snapshot invariant. The subtle bug surfaces because a client holds on to an old snapshot number for the STable and completes its update only after a new snapshot is created and after another client performs a copy-on-write. This delayed update associates different object contents with the version copied to the snapshot, thereby violating the snapshot invariant. The use of conditional update does not help because copy-on-write is a multi-object operation.

One way to fix this bug is for the client to acquire a lock on the object for the duration of the code in Figure 1. This would prevent multiple updates from interleaving. While implementing a lock is feasible as shown earlier, one challenge is to ensure liveness when a lock holder fails. To relieve developers from worrying about these subtle issues and to help reason about correctness despite concurrency and failures, we introduce a new primitive called locks with intent, which the next section elaborates.

3 Locks with intent

As shown in the STable example of §2, the main challenge in developing cloud applications is to ensure correctness in the face of client failures and concurrent cross-scope client operations. Olive therefore introduces locks with intent, which ensures *exactly-once* execution (despite failures) and *mutual exclusion* (for concurrent operations).

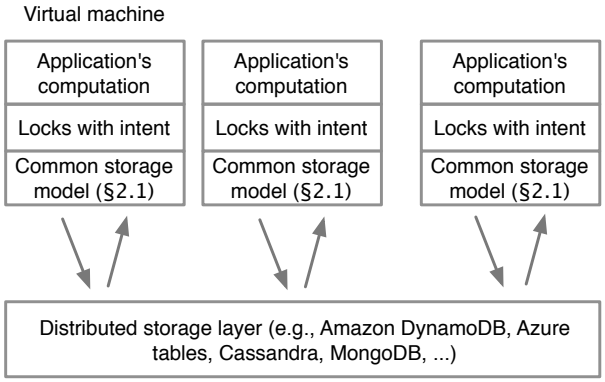

FIGURE 3—Olive's high level architecture. Olive exposes the abstraction of locks with intent (§3) to higher-level application in the form of a library. The abstraction provides eventual exactly-once execution semantics despite failures of nodes running the Olive library. Olive provides such strong semantics by leveraging the fault-tolerance properties of the underlying distributed storage layer.

We intend Olive to be used by both application and infrastructure developers. Since our approach is flexible enough to support a transaction library, as we will show in §4.4, users can always use that library to have the same simplicity offered by transactions. But, crucially, our design also allows sophisticated users to write more efficient implementations, by reducing complexity via automatic failure handling and simplification of concurrency.

Figure 3 depicts Olive's high level architecture. Olive's locks with intent provide a new abstraction for cloud applications to handle failures and concurrency elegantly. This abstraction is built atop the common storage model described in §2.1, which can be mapped to different cloud storage or distributed storage systems. Olive does not modify the storage layer, so it preserves the performance and scalability characteristics of existing storage services. Furthermore, Olive does not require direct distributed coordination among clients running an application's computation: all interactions are through cloud storage, conforming to the existing cloud application model.

3.1 Intents: Exactly-once execution

An *intent* is a request for a certain code snippet to be executed exactly once. The snippet may contain loops or recursive calls, but must terminate in a bounded number of steps. An intent can involve both local computation and operations on cloud storage. The code snippet is arbitrary, but usually it is a critical section protected by a lock.

Determinism. Besides bounded run time, the main restriction on intent code is that it be deterministic. That is, it must produce the same result when executed with the same inputs and in the same state. Determinism makes it possible to replay an execution after a failure by pre-

cisely reproducing results up to the failure point and then continuing execution. Non-determinism is permitted only in Olive-provided routines, where Olive can track the sources of non-determinism and return the same result deterministically. For instance, Olive provides a routine for generating random numbers; the developer must use it instead of the system random number generator.

The code must be deterministic even if run by different clients. For instance, it should not depend on any special privileges possessed only by a subset of clients. §6 will discuss removing this restriction in future work.

Non-deterministic code in an intent constitutes a bug. Olive cannot detect this error; it simply does not guarantee exactly-once semantics in this case.

Tracking and executing intents. Exactly-once execution of an intent is challenging because (i) the initiating client may fail partway through executing it, and (ii) other clients attempting to recover from the initiator's failure may lead to multiple, possibly concurrent, clients executing the intent. After all, failure detection may be imperfect, so one client may incorrectly believe another has failed and attempt to recover from that apparent failure. Olive deals with these challenges as follows.

Olive assigns a unique `intentId` to each intent, and uses this as a key when storing the intent in the `intents` table. To ensure exactly-once execution semantics, Olive must log the steps any client executes as part of an intent. This way, if the client fails, another client will know where to continue from during recovery. Olive does this logging in a table named `executionLog`. For each step requiring logging, Olive adds a new row to `executionLog`, using a key combining `intentId` and the step number within the intent. For local non-deterministic operations, such as those done by Olive-provided random-number-generation routines, Olive stores any non-deterministic choices in `executionLog`. For cloud-storage operations that return results (e.g., reads), Olive stores those results in `executionLog`. This logging allows any future re-execution of an intent to return the same result.

Due to the limited storage model described in §2.1, Olive cannot *atomically* read an object from cloud storage and write it to `executionLog`. Fortunately, it does not have to, because read operations have no immediate externally visible effect. In fact, for better performance, Olive defers logging until right before it executes an externally visible operation. As a result, a client could crash immediately after issuing a read operation to the cloud storage, but before logging to `executionLog`. In that case, if a client resumes executing the intent, it will re-execute the read operation, potentially getting a different value from the cloud storage. This is safe because only the new execution leads to externally visible effects.

DAAL. We have so far treated `executionLog` as if it were a single standalone database table, but as we will now discuss it is only *logically* a single table. To achieve exactly-once semantics, when we update data we must also, in one atomic action, log that update to `executionLog`. But, as discussed in §2.1, most cloud storage services do not support atomic actions across tables. Thus, while it is possible to store logs of read operations in a single table, each log of a write operation must be in the same table as the object being written.

To solve this problem, Olive introduces a novel logging scheme called *distributed atomic affinity logging* (DAAL). With DAAL, `executionLog` consists of two parts. The first part, which stores the results of completed read operations, is a regular table. The second part, which stores log entries corresponding to writes, is a set of invisible entries distributed among the scopes in the system. To perform an `AtomicBatchUpdate` that both updates an object and inserts an `executionLog` entry, Olive chooses a scope for the entry that includes the updated object.

Olive deterministically derives each entry's identifier from the `intentId`, the current intent step number, and the key of the modified object. This ensures that, if another client later tries to perform the update a second time, it will fail because the invisible log entry already exists. If the log entry is a row, then the second insertion will fail because of a key conflict. If the log entry is an attribute, the second insertion will fail because the client will first do a read to ensure the absence of the attribute, then will perform the attribute insertion using `UpdateIfUnchanged`.

Olive also logs progress information in the central `intents` table—one column of each intent's row indicates how many update steps of that intent have been executed. The steps recorded in the intents table must actually have been performed although additional steps may have been executed that are not yet recorded. This is just an optimization to avoid clients wasting time attempting to re-perform already-executed steps. Not recording an already-executed step does not compromise correctness because DAAL ensures that no client will be able to successfully execute any update a second time.

Note that the first part of `executionLog`, the single table holding logs of read operations, is accessed by every client performing an intent. To prevent this table from becoming a throughput bottleneck, or from exceeding capacity limits, Olive partitions it on `intentId`. The degree of partitioning is configurable.

Liveness. Exactly-once semantics requires more than just never executing any intent more than once. It also requires executing each intent *at least* once. We ensure this liveness property as follows.

First, we put another requirement on intent code besides non-determinism and bounded run time. The developer

must ensure that, as long as the code is retried repeatedly, it eventually completes.

Given this requirement, all Olive must do to achieve liveness is to retry each intent repeatedly. To ensure such repeated retries, Olive uses an *intent collector*. This special background process periodically scans the `intents` table to identify incomplete intents and complete them.

Such an intent collector guarantees liveness as long as it never stops. Fortunately, cloud providers offer mechanisms to monitor core services and to restart them if they fail; such a mechanism should be used for the intent collector. Even if this causes multiple instances of the intent collector to coexist briefly, this is safe because of Olive's assurance of at-most-once semantics for each intent.

Indeed, it may be desirable to *always* run multiple instances of the intent collector, so that if one fails and the failure takes time to be detected and rectified, intents are still completed promptly. Multiple instances may, for efficiency, be designed to partition work among themselves, but we have not yet implemented such partitioning.

3.2 Mutual exclusion with exactly-once semantics

Intents can be combined with locks to ensure both mutual exclusion and exactly-once semantics, leading to a powerful new primitive called locks with intent. A "lock" in this context is like a typical lock in that it restricts access to an object or set of objects. However, the access restriction is not to a single client but to a single intent: only clients performing that intent are permitted access. That intent has a step that acquires the lock, then steps that access the locked objects, then a step that releases the lock.

From the developer's perspective, the lock is easy to use since it acts like a regular lock that restricts object access to only a single client. In reality, locked objects are accessible to multiple clients, but, because of the exactly-once semantics of intents, all those clients' object accesses are equivalent to accesses by a single client. Thus, semantically, acquiring one of our locks is equivalent to acquiring a lock that limits access only to a single client.

Even though our locks are semantically equivalent to normal locks, they are safer to use. An object locked with a normal lock can only be accessed by the client who locks it. This is dangerous since the client may fail, rendering the object forever unavailable. However, by allowing any client performing the intent to access the locked object, the developer no longer has to worry about this concern. Locks cannot be tied up indefinitely; they will eventually be released by the intent collector.

Despite this, a client that needs to access an object may still have to wait a long time for the collector to release an intent lock on it. Thus, as an additional optimization, we introduce the following mechanism. When code within an intent acquires a lock, we associate the intent's `intentId` with that lock using an invisible attribute. When the code later releases the lock, we remove the association with the intent's `intentId`. This way, if another client needs the lock but finds it unavailable, it can tell whether the lock is held by an intent. If so, the blocked client can take responsibility for immediately completing the intent, thereby allowing itself to make progress.

```
def updateObject_IntentCode(key, newObj):
  obj = curTable.Read(key)
  if obj == None:
    return NOT_FOUND

  table.Lock(obj.key)
  lastSnapshot = curTable.Read(LAST_SNAP).value
  curEpoch = lastSnapshot + 1

  if (obj.version <= lastSnapshot):
    snapshotTables[lastSnapshot].Update(key, obj)

  newObj.version = curEpoch
  curTable.UpdateIfUnchanged(key, newObj)
  table.Unlock(obj.key)
  return SUCCESS
```

FIGURE 4—Pseudocode for the intent code to update an object.

4 Applications and experience

Locks with intent make it easy to reason about desirable correctness and fault-tolerance properties of software. To demonstrate their general utility, this section will describe how we use them to build several components.

Note that these components are themselves generally useful. That is, each is a library that provides applications with a storage API richer than that of the underlying cloud storage system. In §4.1, we discuss our STable library, which augments the cloud storage API with a facility for snapshotting tables. In §4.2, we discuss a library that adds the ability to do live table re-partitioning. In §4.3, we show how to add support for secondary indices. Finally, in §4.4, we show how to add the ability to form ACID transactions out of arbitrary sequences of operations.

4.1 Snapshots

One component we build is the STable library, which provides applications with the ability to take snapshots of tables. In §2.2, we discussed how the complexity of table snapshotting can lead to subtle bugs. In particular, the code in Figure 1 can lead to a violation of our snapshot invariant. The fundamental reason the bug arises is the difficulty of reasoning about the many possible interleavings of concurrent clients.

Fortunately, intents provide a straightforward way to reduce the possible interleavings. That is, we can create an intent that locks `obj` while executing the code from Figure 1; the resulting intent code is shown in Figure 4. Because the intent locks `obj`, executions of intents on

the same obj are serialized; i.e., they do not overlap. Furthermore, we do not have to worry about liveness issues arising from introducing locks, because locks with intent automatically defend against failing lock holders.

Here is an argument that the snapshot invariant is maintained by this approach. Because the intent locks obj, all executions of the intent on the same obj are serialized, i.e., they do not overlap. Consider any run of the intent that copies the contents of obj to snapshot table i. Because this copy occurs in the middle of an intent, and all intents to obj are serialized, the copy must reflect all earlier executions of the intent. That is, it must reflect the last update performed so far, and the snapshot invariant holds. We must also demonstrate that the invariant continues to hold, i.e., that a later update will not violate it by writing to the current table with version i. To demonstrate this, we observe that any subsequent run of the intent for obj will be serialized afterward. Those runs will read a lastSnapshot $\geq i$, causing them to use a curEpoch $\geq i+1$. Thus, the snapshot invariant is maintained.

Note that the only object we lock is obj; we do not lock the special row with key LOCK_SNAP. Thus, we do not conflict with concurrent operations that update the current snapshot number. If we were to use transactions instead of locks with intent, we would have such a conflict.

Our STable implementation offers stronger properties than just the snapshot invariant. For instance, it ensures that any two reads of the same key from the same snapshot will return the same object contents. It also offers further functionality, like the ability to garbage-collect old snapshots and to roll back to earlier snapshots. These facilities also became easier to build with locks with intent.

4.2 Live table re-partitioning

Another component we build is a library that exports a facility for live re-partitioning of tables. This functionality is crucial if a table may grow to the point where it exceeds system-imposed size limits. It can also help relieve "hot spots" by dividing a frequently-accessed tables into multiple tables with consequently greater throughput. By building this library, we do for general cloud storage what Zephyr [28] did for transactional storage.

A straightforward approach would be to lock the table for the duration of re-partitioning. However, re-partitioning potentially involves an enormous amount of data movement, taking seconds or minutes. So, it is unreasonable to block clients during re-partitioning; we must allow concurrent operations during re-partitioning.

This concurrency requirement poses challenges for correct development. The developer must now reason about all the possible interleavings of client operations with steps of re-partitioning. Failure to do so can lead to bugs.

To illustrate this, Figure 5 depicts a buggy design aimed at enabling object updates during live re-partitioning of

```
1  def migratePartitionToNewTable(pKey, futTable):
2    curTable = metaTable.Read(pKey).value
3    metaTable.Update(pKey, [curTable, futTable])
4
5    objectsToMove =
6      Scan(curTable, partitionKey == pKey)
7    for (obj in ObjectsToMove):
8      futTable.Create(obj.key, obj)
9    metaTable.Update(pKey, [futTable])
10
11 def updateObject(key, newObj):
12   # get partition key associated with the key
13   pKey = getPartitionKey(key)
14   tablesList = metaTable.Read(pKey).value
15
16   # check if this table is being re-partitioned
17   if (tablesList.len == 1):
18     curTable = tablesList[0]
19     curTable.Update(key, newObj)
20   else: ...
```

FIGURE 5—Pseudocode for the object-update routine and a migration routine in a buggy live re-partitioning design.

tables. The bug arises in the following scenario. Suppose that, when a client starts executing updateObject for key k, there is no ongoing re-partitioning job, so the client reaches line 17. At this point, a re-partitioning job commences, and successfully migrates key k to the new partition. The client then continues from line 17, writing its update only to a table that will soon be obsolete. Eventually, the re-partitioning job reaches line 9 without realizing there is useful data it missed in the current table. So, when it updates the metaTable, it effectively and incorrectly rolls back the client's update.

Fortunately, such challenges and reasoning can be substantially mitigated due to locks with intent. Our general strategy is to break the job of re-partitioning into small tasks, each of which is short enough that it is acceptable to block clients for its duration. We then use a lock with intent for each such task, and a lock with intent for each client operation on the table.

In this way, we do not have to reason about arbitrary interleavings between clients and the re-partitioning job. We only have to reason about interleavings at the coarse scale of tasks. For instance, a client operation can overlap the re-partitioning job, but it cannot overlap a task that accesses the same object. More specifically, each task corresponds to migrating one object from one partition to another partition. This involves replacing the object in the old partition with a marker that redirects clients with outdated views to the new partition. Figure 6 depicts pseudocode for the migration routine as well the object-update procedure in the re-partitioning service that uses locks with intent.

With this migrator design, the object-update routine does not use locks. If an object is locked for migra-

```
def migrateIntent(curTable, futTable, obj):
  curTable.Lock(obj.key)
  futTable.Create(obj.key, obj)
  obj.migrated = True
  curTable.Update(obj.key, obj)
  curTable.Unlock(obj.key)

def migratePartitionToNewTable(pKey, futTable):
  curTable = metaTable.Read(pKey).value
  metaTable.Update(pKey, [curTable, futTable])
  objsToMove =
     Scan(curTable, partitionKey == pKey)
  for (obj in ObjsToMove):
    migrateIntent(curTable, futTable, obj)
  metaTable.Update(pKey, [futTable])

def updateObject(key, newObj):
  pKey = getPartitionKey(key)
  tablesList = metaTable.Read(pKey).value
  curTable = tablesList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tablesList[1]
    oldObj = curTable.Read(key)
    if (oldObj.migrated == True):
      futTable.UpdateIfUnchanged(key, newObj)
    elif (oldObj.locked == True):
      migrateIntent(curTable, futTable, oldObj)
      futTable.UpdateIfUnchanged(key, newObj)
    else:
      curTable.UpdateIfUnchanged(key, newObj)
```

FIGURE 6—Pseudocode for the migration routine and the object-update routine in the live table re-partitioning service based on Olive's locks with intent.

tion, it assists the migrator by executing the associated intent, before performing its update. Otherwise, it uses `UpdateIfUnchanged` to modify the object in the old partition (if the object is not migrated or if no migration in progress), or in the new partition (if the object is already migrated). If a client holds an outdated view (e.g., it incorrectly thinks no migration is in progress or an object is not migrated), the `UpdateIfUnchanged` fails, causing it to retry, which will update its view. Furthermore, unlike in the buggy design shown earlier, it is safe for the object-update routine to update an object in the old partition as long as it has not been migrated because the migrator will eventually move the updated object to the new partition.

Of course, locks with intent are not a panacea. There are still several tricky cases to consider, such as how to avoid conflict between a re-partitioning task and a client with an outdated view attempting to insert an object with the same key. However, we find that the number of cases to consider is much smaller thanks to the coarsening of operations enabled by locks with intent.

4.3 Secondary indices

Another component we build is a library that supports constructing, maintaining, and using secondary indices. A secondary index for a table `T` is a separate table `T'` designed to allow quick lookups into `T` using a non-key attribute `Attr`. Each row of `T'` consists of an `Attr` value and a `T` key. However, `T'` uses the `Attr` values as its keys.

The main challenge in building secondary-index support is maintaining consistency between `T` and `T'`. Because cloud storage systems typically do not support multi-table atomic transactions, there are necessarily times when the two tables' contents are not consistent with each other. For example, a row may exist in `T` without a corresponding row in `T'`.

To see the challenge more concretely, consider the following naïve algorithm for updating an object in `T`:

1. Update the corresponding row in `T`.
2. Insert a row into `T'` mapping the new `Attr` value to the key of the updated row.
3. Delete the row from `T'` with the former `Attr` value.

Unfortunately, this logic is not robust to failures: if a process running the above procedure crashes after step 1 but before completing steps 2 and 3, it will leave the underlying storage in an inconsistent state.

We could address this with "cleanup" processes that run in the background to periodically find and fix inconsistencies between `T` and `T'`. However, in addition to complicating deployment and wasting resources by continuously scanning for inconsistencies, such cleanup processes can actually *introduce* inconsistency, as in the following scenario. First, a client updates an object to change `Attr` from OLD to NEW, but crashes before step 3. Next, a cleanup process notices this and decides to delete the row in `T'` with `Attr`=OLD. Next, another client decides to change `Attr` back to OLD, and completes steps 1 through 3. Finally, the cleanup process acts on its earlier decision and deletes the row in `T'` corresponding to OLD, not realizing that this is actually now a useful row. This leaves `T'` without any row corresponding to the object.

Olive provides a natural solution to eventually consistent secondary indices. We perform steps 1–3 described earlier in an intent that also locks the object from `T`. Because secondary indices are eventually consistent when the intents complete their executions, we do not have to worry about inconsistencies caused by intermediate failures, like the one discussed earlier. Additionally, the intent collector in this solution has to do less work than the cleaner process described earlier. After all, the cleaner process must scan *all* rows changed since the last time it ran, but the intent collector only has to scan the `intents` table for outstanding incomplete intents.

```
def atomicCommit(objectsRead, objectsModified):
  for (obj in objectsModified):
    table.Lock(obj.key)

  success = True
  for (obj in objectsRead): # verify read set
    retrievedObj = table.Read(obj.key)
    if (retrievedObj.version != obj.version):
      success = False
      break

  if (success): # commit and unlock
    for (obj in objectsModified):
      obj.locked = False
      table.Update(obj.key, obj)
  else: # abort and unlock
    for (obj in objectsModified):
      table.Unlock(obj.key)
```

FIGURE 7—Pseudocode for atomic commit in OCC-based transactions.

4.4 Transactions

The last component we build is a library that augments the cloud storage API with the ability to form transactions out of an arbitrary collection of operations. This is valuable because, as described in §2.1, most cloud storage systems do not support transactions across tables. We will see that Olive's locks with intent make it simple to build a client-side library that exports APIs to execute general-purpose ACID transactions [16, 33, 40, 55].

Our design of this transaction library is based on optimistic concurrency control (OCC), which has three phases: shadow execution, verification, and update in place [37]. To guarantee ACID semantics, an OCC protocol requires that the last two steps happen atomically. In particular, they must be isolated from updates of other transactions. Furthermore, all changes made by the transaction must be either committed or aborted in their entirety. Thus, a core piece of a distributed transaction protocol is the atomic-commit mechanism. To satisfy these requirements, distributed systems that implement transactions use the following techniques: a special *transaction coordinator* process uses a shadow write-ahead log to ensure atomicity, and uses locks during the verification step to ensure isolation from other transactions [27].

We observe that these techniques can be naturally implemented by using Olive's locks with intent. Figure 7 depicts pseudocode that can be wrapped in an intent to execute the atomic commit mechanism with the aforementioned properties. The mutual exclusion property of Olive's locks with intent provides the desirable isolation, and the intent's execution log acts as a write-ahead log. That is, it contains all the information needed to commit or abort a transaction. Most importantly, for liveness we require that transaction coordinators never fail; we ensure this by using intents as our transaction coordinators.

service	without Olive	with Olive
snapshots	987	665
OCC-transactions	2,201	408
live re-partitioning	2,116	474

FIGURE 8—Comparison of code line counts for services we built with and without Olive.

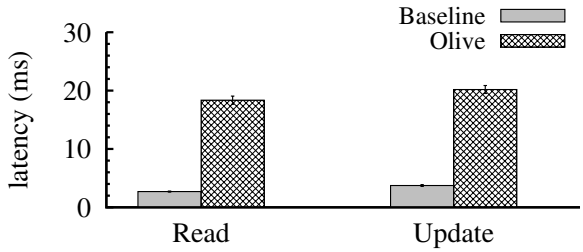

FIGURE 9—Latency of executing a storage operation, either inside an intent or directly on the raw storage interface. The logging required to ensure exactly-once execution semantics adds up to 6–7× the baseline latency. (See text for details.) Figure 10 depicts how these overheads are amortized when an intent contains more than one storage operation.

4.5 Evaluation: ease of development

Before we designed Olive's locks with intent, we created some of the cloud services described in this section by building directly atop the raw cloud-storage interface. It was both tedious and error-prone as we had to reason about many failure scenarios and consider many interleavings of code steps by different clients. To concretely demonstrate how Olive makes it easy to develop such cloud services, we now compare the complexity of developing such cloud services with and without Olive. We use lines of code as a proxy for code complexity.

Figure 8 depicts our results. These results demonstrate that Olive reduces lines of code written, and thus likely reduces complexity. This finding, in combination with our experience (§4.1–4.4), suggests that Olive makes it significantly easier to build these services. Note that one of the artifacts that does not use Olive (live re-partitioning) was built by a different team with a very different approach to making code robust to failures and concurrency. (We note this because the comparison and feature set may not be fully apples-to-apples.) For the case of OCC-based transactions, as discussed in §4.4, Olive makes it simple to express a transactional protocol.

5 Experimental evaluation

The previous section demonstrated that Olive's locks with intent make it easy to design, and to reason about the correctness of, new cloud services that are robust to failures and concurrency. This section experimentally evaluates Olive to understand its costs and benefits.

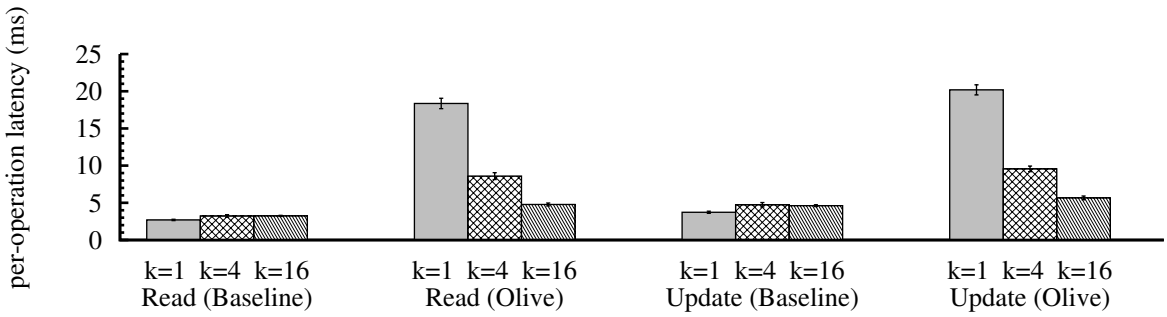

FIGURE 10—Per-operation latency when executing a sequence of k storage operations both normally and within an Olive intent. As the number of operations per intent increases, the per-intent costs (e.g., registering an intent, storing the final result, etc.) are amortized, and the per-operation latencies approach those of operations directly on the raw storage interface.

5.1 Implementation

We implement Olive as a client library in approximately 2,000 lines of C# code, including all features described in §3. As discussed there, although we have built an intent collector capable of coexisting with other instances of itself, our implementation does not support partitioning work among such instances for greater efficiency.

To allow Olive to work on multiple underlying storage systems, we implement it atop an abstract C# interface that exposes the storage model described in §2.1. We then build concrete C# classes that call cloud storage systems' APIs to implement the abstract interface. We implement two such concrete mappings. The first maps to Azure Table storage; it is only 38 lines of code because our abstract storage interface maps one-to-one to its API. The second maps to Amazon DynamoDB; it is 107 lines of code. DynamoDB provides atomicity at the granularity of individual objects [4], so our concrete class only allows `AtomicBatchUpdate` for object scopes.

5.2 Setup and method

We experiment on Olive with Microsoft Azure Table service as the cloud storage. For computation, we use a G3 VM instance (8-core Intel Xeon E5 v3 family with 112 GB RAM) running Windows Server 2012 R2 in the same availability zone as the storage service.

The principal goal of our evaluation is to understand the costs of robustness due to Olive's mechanisms relative to alternative mechanisms. To do so, we compare the performance of Olive-based artifacts with baselines providing similar fault-tolerance guarantees. For these comparisons, our performance metric is the latency of storage operations. In each experiment, we report the mean of at least 1,000 measurements along with the 95% confidence interval for that mean. For these end-to-end experiments, we use YCSB [25] to generate workloads.

5.3 Cost of Olive's exactly-once semantics

To understand the costs of Olive's logging for ensuring exactly-once semantics, we experiment with a series of microbenchmarks. We write two intents, one of which issues a single `Read` on an object and the other of which issues a single `Update`. Each object consists of a random 64-byte key and a random 1-KB value. Our baseline for this is a snippet of code that issues the same operations but without using Olive's intent-execution machinery. We run these intents and the associated baselines 1,000 times, and measure the latency of the aforementioned operations. Figure 9 depicts our results.

As expected, Olive pays significant latency overhead compared to a baseline that does not ensure exactly-once semantics. The reason is that Olive has to register its intent by writing to the `intents` table, then insert DAAL entries. Furthermore, our implementation writes an entry to another `results` table when an intent execution is complete. The last operation is not crucial to Olive, but stores a succinct summary of the intent execution including the final return value of the intent. Our implementation does this so that other clients can quickly learn the final return value of an intent by simply doing a lookup on this table.

Amortizing setup costs. Much of this overhead (registering an intent, saving the final results, etc.) is per-intent cost. Thus, to understand the costs of Olive's intent execution in a comprehensive manner, we run another set of experiments in which we vary the number of operations k per intent, setting k=1, 4, and 16. Figure 10 depicts our results. As expected, the aforementioned per-intent costs amortize over multiple operations, and the per-operation cost of a storage operation in an intent is comparable to that of directly executing the operation without Olive.

Varying object sizes. We experiment with Olive and the baseline under varying value sizes (16 bytes, 128 bytes, and 1 KB) and with varying k. We find that neither Olive's costs nor the baseline's costs grow with value size, so we do not depict these results.

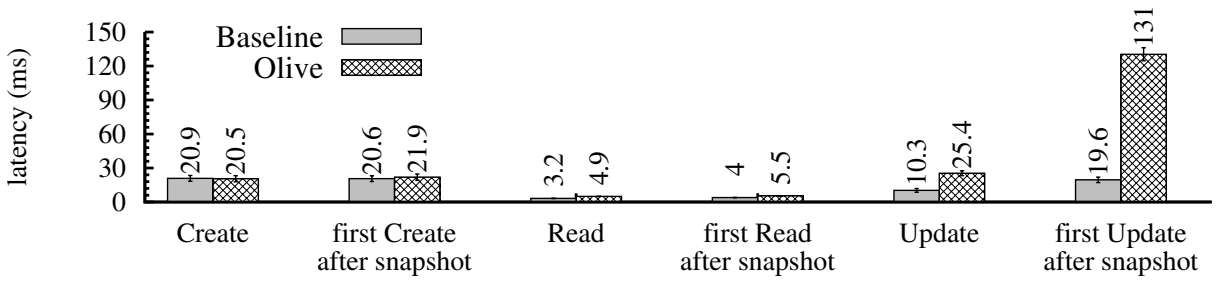

FIGURE 11—Latency of `Create`, `Read`, and `Update` operations using two snapshotting services: (i) a baseline that uses a cloud database's native support for creating snapshots, and (ii) the Olive-based artifact. Under Olive, the latency of an `Update` immediately after taking a snapshot takes 5× as long as a normal `Update`; the baseline incurs only 2× higher latency due to native support for snapshots. The Olive-based artifact is competitive with the baseline for non-`Update` operations.

5.4 End-to-end performance: snapshots

To understand the performance of an Olive-based artifact, we experiment with the snapshotting service we built. Two reasonable alternative approaches for building this artifact in a cloud environment are: (i) create a snapshotting service atop a cloud storage system similar to the Olive-based artifact, but, instead of using DAAL, put all the application data as well as snapshots of that data in the same atomicity scope; and (ii) employ a cloud storage system that natively supports creating snapshots, e.g., Azure SQL or Amazon Aurora.

Alternative (i) can exploit `AtomicBatchUpdate` to create snapshots of application data, without having to incur difficulties we discussed earlier (§2.2). Unfortunately, for most cloud applications, this is truly not an option. It severely limits throughput and scalability, because each atomicity scope supports only a few thousand requests per second. It also limits capacity, because each atomicity scope supports only a certain amount of data. Furthermore, in some cloud storage systems (e.g., Amazon DynamoDB), the atomicity scope is a single object, thereby rendering this option infeasible. We thus use alternative (ii) for our baseline; specifically, we use Azure SQL. Of course, this baseline supports far more features than our artifact, but we find that it is the closest alternative with similar functionality in a cloud environment.

In our measurements, a client process first preloads a table created by YCSB's benchmarking tool; the table contains 1,000 objects, each having ten attributes with 100-byte values. The client process then runs a series of experiments, in which it uses YCSB's core workloads a–d to generate a stream of 1,000 requests with varying mixtures of `Read`, `Create`, and `Update` operations. We also run another set of experiments, in which the client process creates a snapshot between preloading the table with data and running the workloads. This causes each `Update` operation in the experiments to perform copy-on-write. Figure 11 summarizes our results.

For operations other than `Update`, the Olive-based artifact's performance is competitive with the baseline. The largest difference is that, under Olive, the first `Update` immediately after taking a snapshot incurs 5× higher latency than a normal `Update`; for the baseline, it is only 2× higher. The primary reason is that, while the baseline requires only a single round trip to the database server, Olive incurs additional round trips and extra logging to ensure the exactly-once semantics. Furthermore, the database service likely uses complex machinery to implement the snapshotting feature efficiently.

Finally, because the Olive-based snapshotting service uses a NoSQL cloud storage system instead of the baseline's SQL database service, it likely incurs much lower monetary cost. Unfortunately, the pricing models of these cloud services are complex and hard to compare, so we now provide only a rough comparison.

Azure's table service charges on two axes: amount of data stored and number of operations performed. For example, in the West-US data center, it costs $0.045–0.12 per GB of data per month (depending on amount of data stored and the desired geo-replication level), and $0.036 per million cloud storage operations [2].

On the other hand, Azure SQL charges for desired throughput, measured in database transaction units (DTUs) [6], and amount of data stored. A DTU is much more complex than the number of cloud storage operations because it accounts for the number of disk operations and the amount of processing consumed by a SQL query. As an example, for 5 DTUs and 2 GB of data, the charge is $5/month. This increases to $465/month with 125 DTUs and 500 GB of data.

A rough comparison using these figures suggests that the cost of a SQL database is at least an order of magnitude higher than Azure's table store. As a result, the Olive-based artifact reaps lower monetary costs from its underlying store while providing a snapshotting feature with comparable performance.

5.5 End-to-end performance: live re-partitioning

To understand the benefits of the flexible isolation properties of Olive's locks with intents, we evaluate two Olive-based implementations of the live re-partitioning service (§4.2): one using the Olive-based transaction library (§4.4) and the other using intents directly for fine-granularity isolation. Such a comparison will also help demonstrate the flexibility of locks with intent: developers can build their service atop our transaction library first for simplicity and then later optimize it by writing that service with intents directly for performance.

We run experiments in which a client process first preloads a table with 1,000 objects and then issues a stream of `Create`, `Read`, and `Update` operations generated via YCSB's core workloads a–d. Figure 12 depicts the performance of the intent-based artifact and compares it with a transaction-based implementation. We find that in all cases the intent-based artifact performs better than, or as well as, the transaction-based one.

A notable scenario where Olive's locks with intent enable us to optimize the re-partitioning service is in the implementation of its `Update` operation. In the intent-based artifact, this operation does not need to lock any objects, but the transaction library cannot avoid locking. In particular, the intent-based artifact implements this by exploiting the `UpdateIfUnchanged` API supported by the underlying cloud storage system. As a result, the latency goes down by roughly 6×. Similarly, for `Create` and the data-migration routine, the intent-based artifact locks fewer objects, enabling it to achieve better performance than the transaction-based one.

Finally, because the migration routine in the re-partitioning service locks and migrates one object at a time, if a normal table operation (e.g., `Read`) observes that an object is locked, it has to block for the duration of data migration. (Figure 12 does not depict this case.)

5.6 Storage overheads

Up to this point, we have measured the latency overhead Olive incurs to ensure exactly-once semantics. We now evaluate Olive's storage overhead. We do this by running a microbenchmark with our re-partitioning service. In particular, we use the data-migration routine depicted in Figure 6. For the experiment, we preload data into a table and use YCSB's core workload a, which inserts 1,000 objects each having ten attributes with 100-byte values. We then run Olive's migration routine to move those objects to a new table. We measure the total size of all tables before and after the migration.

The application data inserted by the workload is roughly 1 MB. Before the migration runs, we find that the total size of the tables is 2.6 MB. The 2.6× overhead comes from the internal use of an intent in the table re-partitioning service's `Create` operation. This intent

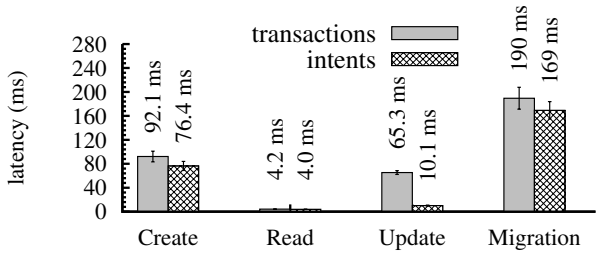

FIGURE 12—Comparison of the performance of table operations in two implementations of the re-partitioning service (§4.2), one using Olive's transaction library and another using Olive's intents directly. The intent-based artifact achieves better performance because we carefully optimize the set of objects that get locked, whereas the implementation based on Olive's transaction library (naively) locks all the objects affected by a table operation. (See text for details.)

serializes the object to be inserted and stores it in the `intents` table (as the initial state of the intent). As a result, it doubles the size of data in tables. The remaining overhead is due to DAAL entries and invisible attributes (e.g., `locked`). After the migration procedure completes, the total size of the data in the tables increases to 5.5 MB, which again is due to the use of intents. Fortunately, most of this overhead is ephemeral: after garbage collection and deletion of objects in the old table, the storage overhead is less than 8% of the application-data size.

5.7 Summary

While Olive incurs unavoidable latency and storage overheads to ensure exactly-once semantics (§5.3), our experience suggests that Olive's strong semantics make it easy to quickly build cloud services that are correct and fault-tolerant (§4). Furthermore, our end-to-end experiments show that Olive-based artifacts have performance comparable to baselines that provide similar fault-tolerance guarantees. Finally, even when a feature is offered by some cloud storage system, building the feature with Olive can save significant money by enabling the use of a less expensive cloud storage system.

6 Discussion

Comparison to transactions. Transactions are arguably simpler to use than intents, but offer less flexibility. Intents can support both strong consistency and eventual consistency, can avoid full isolation when not required, and can support exactly-once semantics which are often not provided by transactions. Indeed, as shown in §4.4, intents are general enough to support transactions, so developers who want a simple experience can always use the transaction library Olive provides.

Liveness of intents. The liveness of Olive's locks with intent hinges on the liveness of intents. Just as with code,

developers must ensure that an intent does not contain bugs leading to infinite loops, crashes, or deadlocks. A more subtle concern is that Olive may amplify such bugs. For example, testing often misses bugs occurring in rare scenarios. So, Olive may trigger latent bugs by exercising the rare scenario in which an intent is executed not by its owner but by another client or the intent collector.

It is possible to automatically recover from deadlock bugs in intents. After all, Olive's locks with intent can be used to encode deadlock recovery logic inside an intent, e.g. by undoing the effects of the intent to release a lock. Olive's semantics ensure that such recovery logic inside an intent is executed exactly once. As an example, our OCC-based transaction library (§4.4) includes such logic to abort a transaction if it detects read-write conflicts during the verification step.

Garbage collection of intent logs. Entries in the `intents` table, as well as DAAL entries, need to be garbage-collected. But, this must be done with care, because a recovering client uses the existence of an entry to decide whether to execute a step in an intent. A slow client that attempts to execute a step in an intent might mistakenly re-execute it because the corresponding entries have been garbage collected. One solution is to introduce the notion of epochs for intents, with a mandate that an intent created in epoch n must be completed by the end of epoch $n+1$ and is considered *outdated* in epoch $n+2$ and beyond. It this case, is safe to garbage-collect entries for outdated intents. The exact duration of an epoch can be application-specific (e.g., a day).

Security and privacy. Olive assumes that all clients sharing an `intents` table belong to the same application, and thus any client can complete any other client's intent. However, for some applications this is not possible due to security restrictions. For instance, a client serving user Alice may downgrade its capabilities, to ensure it cannot accidentally leak information to Alice about other users. So, that client may be unable to complete an intent running on behalf of another user Bob.

Differences in clients' permissions can also lead to privacy violations. For instance, a client running on Alice's behalf may write Alice's private data into the `intents` table. Then, another client running on Bob's behalf may read her data from that table and leak it to Bob.

For these reasons, Olive is suitable only for applications with clients in equivalent security domains. In future work, we plan to address this limitation, e.g., by propagating clients' security restrictions to the `intents` table.

Cloud support. We have designed Olive under the assumption that cloud providers are unwilling or unable to change their APIs. However, a cloud provider could choose to add locks with intent to its external API. After all, unlike richer primitives like transactions, locks with intent would not require significant changes to cloud-storage internals. By adding locks with intent natively, and having more efficient execution paths for performing and completing intents, the cloud provider might achieve better performance than Olive.

7 Related work

Olive is related to a host of techniques that provide a substrate for building fault-tolerant services. It is also related to work that makes reasoning about concurrency easier for programmers.

State machine replication [38, 49] is a classic technique for building fault-tolerant services. A cloud application can use replication directly for fault tolerance. Olive instead takes a different approach by leveraging the underlying cloud storage, which is already made fault tolerant by using replication internally. Olive's approach avoids consensus at the application layer and maintains reliable persistent states only in cloud storage.

There is also a long line of work on recovering computation from failures [22–24, 42, 43]. Compared with this work, nodes in Olive maintain and share state via cloud storage, which makes recovering from failed computation harder. Even with microreboots [22, 23], maintaining persistent state consistently despite failures is left to applications, which Olive addresses.

Write-ahead logging [46] is a well-known technique, widely used in database systems [9, 14, 18, 47], to provide atomicity and durability in the presence of failures. Olive's intent `executionLog` is logically a write-ahead redo log, but Olive uses DAAL to achieve exactly-once semantics and to cope with concurrent executions of the same intent. A related technique, journaling, is also widely used in file system implementations [44, 45, 50] to ensure data consistency despite inopportune machine crashes.

Transactions, another popular primitive, provide strong ACID properties. Transactions simplify concurrency control by providing strong isolation [11, 17, 40, 41] among concurrently executing transactions. Recognizing the performance overheads imposed by general-purpose transactions in a distributed system, Sinfonia [13] proposes a restricted form of transactions called minitransactions. Sagas [31], on the other hand, proposes to split long-lived transactions into smaller pieces to enhance concurrency. It relies on user-defined compensating transactions to recover the database to a consistent state if individual transactions fail. Like Sagas, Salt [53] allows developers to improve performance by gradually weakening the semantics of performance-critical transactions. In the last few years, several works [12, 16, 26, 27, 55] have built distributed storage systems with general-purpose transactional features, sometimes exploiting modern hardware such as RDMA [27], a cluster of flash devices [15, 16], and TrueTime [26]. Olive takes a different approach since

distributed transactions on cloud storage would be expensive. The lock with intent in Olive has the benefits of a transactional primitive (automated failure handling, robustness to concurrency), but exposes a simple concurrency primitive that can be implemented without the full machinery and expense of transactions.

Leases [32], another popular distributed-system primitive, ensure exclusive access to a data object for a configurable amount of time. Chubby [20] and ZooKeeper [10] each implements a reliable lock service with lease-like expiration, enabling nodes in a distributed system to coordinate, usually at a coarse granularity. Like Olive's locks with intent, the failure of a lease owner will not block the entire system forever, as leases eventually expire. However, when a lease owner crashes, lease expiration does not automatically restore cloud storage to a consistent state, a key problem that locks with intent address.

Revocable locks [34] provide the abstraction of non-blocking locks in a shared-memory model on a single machine. To get non-blocking semantics for locks, a thread can revoke a lock from its current owner and direct that lock owner's thread to execute a predefined recovery code block. Olive's locks with intent are similar in spirit. However, Olive does not require that the user write and reason about recovery logic. Also, Olive goes beyond a single-machine context to solve issues arising from machine failure and asynchrony in a distributed system. Such a design is crucial in Olive's context given the distributed-systems setting where accurate failure detection and synchronization among clients is hard.

Exactly-once semantics and idempotence have been recognized as critical properties in various systems for simplifying application development and achieving stronger semantics [35]. Exactly-one semantics has been used as a correctness criterion for replicated services [30], for building three-tier Web services [29], and for distributed message delivery systems [36]. It has also been incorporated into database systems via queued transaction processing [19]. Recognizing the power of such semantics, Ramalingam and Vaswani [48] design a programming language monad that uses idempotence and exactly-once semantics to tolerate process failures and message loss in a distributed system. However, they neither consider concurrency control primitives in the presence of failures, nor use automatic failure detection and retry mechanisms, leading to different design decisions. For example, in Olive's locks with intent, we find it crucial to track all intents associated with a locked object via cloud storage and to let any client execute any intent in the system. We also achieve a useful liveness property via an intent collector, which is not covered in their work.

In more recent work, RIFL [39] implements a reusable module to enhance the semantics of a key-value storage system's interface from at-least-once to exactly-once. An application that builds atop such a storage service can handle server failures more easily. Olive's locks with intent provide similar exactly-once semantics, but in a stronger sense: the improved semantics are useful to tolerate failures in the application layer, and they are guaranteed for arbitrary snippets of code rather than only for RPCs.

Besides fault tolerance and concurrency control, there are many works that enhance other properties of cloud services, such as the following. Tombolo [54] proposes the use of cloud gateways to reduce the latency of cloud data accesses. CosTLO [52] reduces the latency variance of cloud storage services. SPANStore [51] helps developers manage the use of multiple cloud storage services, to reduce service cost while still meeting the latency, data consistency, and fault tolerance requirements.

8 Conclusion

Cloud applications atop distributed reliable cloud storage services represent a new model of building fault-tolerant distributed systems, where all coordination at the application layer goes through cloud storage, without the need to re-implement consensus protocols.

Devising the right programming abstraction in this model involves the art of balancing a set of attributes, such as simplicity, programmability, expressiveness, efficiency, and generality. Olive's lock with intent strikes such a delicate balance: its exactly-once semantics and mutual exclusion property are simple to understand and to use when reasoning about correctness; it is easy for developers to program with because it reuses common constructs such as locks, with an intent just as an arbitrary code snippet; it can be used to implement both weak eventual consistency and strong transactional consistency, allowing an efficient design without excessive constraints; it is generally applicable to a set of cloud storage services and popular distributed storage systems with the use of a common storage API. The result is a powerful new primitive that allows us to develop a set of useful advanced functionalities easily, correctly, and efficiently.

Acknowledgments

We thank Mahesh Balakrishnan, Albert Greenberg, Rama Kotla, Ashwin Murthy, and Doug Terry for creating the Azure Replicated Table (RTable) project, which inspired us to investigate foundational primitives for building reliable cloud applications. We thank John Erickson and Matt McCutchen for introducing us to the live table repartitioning problem described in §4.2. We thank Sebastian Angel, Natacha Crooks, Thomas Moscibroda, Lenin Sivalingam, and the anonymous reviewers for their comments and discussions, which substantially improved this work. We are particularly grateful to our shepherd Peter Druschel for his insightful suggestions and guidance.

References

[1] Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[2] Azure Storage Pricing. https://azure.microsoft.com/en-us/pricing/details/storage/tables/.

[3] Azure Table storage. https://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-how-to-use-tables/.

[4] BatchWriteItem. http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchWriteItem.html.

[5] CQL for Cassandra/BATCH. https://docs.datastax.com/en/cql/3.3/cql/cql_reference/batch_r.html.

[6] Database Transaction Units (DTUs). https://azure.microsoft.com/en-us/documentation/articles/sql-database-what-is-a-dtu/.

[7] Entity group transactions. https://msdn.microsoft.com/en-us/library/azure/dd894038.aspx.

[8] Google cloud Bigtable. https://cloud.google.com/bigtable/docs/.

[9] PostgreSQL. http://www.postgresql.org/.

[10] Apache ZooKeeper. https://zookeeper.apache.org/, 2008.

[11] A. Adya, B. Liskov, and P. O. Neil. Generalized isolation level definitions. In *International Conference on Data Engineering (ICDE)*, pages 67–78, 2000.

[12] M. K. Aguilera, J. B. Leners, and M. Walfish. Yesquel: scalable SQL storage for web applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[13] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–174, 2007.

[14] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[15] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2012.

[16] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–340, 2013.

[17] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD*, pages 1–10, 1995.

[18] P. A. Bernstein and E. Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 2009.

[19] P. A. Bernstein and E. Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 2009.

[20] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[21] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

[22] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. *Perform. Eval.*, 56(1-4):213–248, Mar. 2004.

[23] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[24] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 100(6):546–556, 1972.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, 2010.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 251–264, 2012.

[27] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 54–70, 2015.

[28] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *ACM SIGMOD*, pages 301–312, 2011.

[29] S. Frølund and R. Guerraoui. Transactional exactly-once. Technical report, Hewlett-Packard Laboratories, 1999.

[30] S. Frølund and R. Guerraoui. X-ability: A theory of replication. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

[31] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ICMD*, pages 249–259, 1987.

[32] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1989.

[33] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *International Conference on Very Large Data Bases (VLDB)*, pages 144–154, 1981.

[34] T. Harris and K. Fraser. Revocable locks for non-blocking programming. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 72–82, 2005.

[35] P. Helland. Idempotence is not a medical condition. *Communications of the ACM*, 55(5):56–65, 2012.

[36] Y. Huang and H. Garcia-Molina. Exactly-once semantics in a replicated messaging system. In *International Conference on Data Engineering (ICDE)*, pages 3–12, 2001.

[37] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[38] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[39] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing linearizability at large scale and low latency. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–86, 2015.

[40] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, 1988.

[41] B. Liskov, D. Curtis, P. Johnson, and R. Scheifer. Implementation of Argus. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1987.

[42] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 20–20, 2000.

[43] D. E. Lowell and P. M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical report, Technical Report CSE-TR-410-99, University of Michigan, 1998.

[44] C. Mason. Journaling with ReisersFS. *Linux Journal*, 2001(82es):3, 2001.

[45] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[46] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[47] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 183–191, 1999.

[48] G. Ramalingam and K. Vaswani. Fault tolerance via idempotence. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 249–262, 2013.

[49] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, Dec. 1990.

[50] S. C. Tweedie. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.

[51] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[52] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[53] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 495–509, 2014.

[54] S. Yang, K. Srinivasan, K. Udayashankar, S. Krishnan, J. Feng, Y. Zhang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Tombolo: Performance enhancements for cloud storage gateways. In *IEEE Conference on Massive Data Storage (MSST)*, 2016.

[55] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–278, 2015.