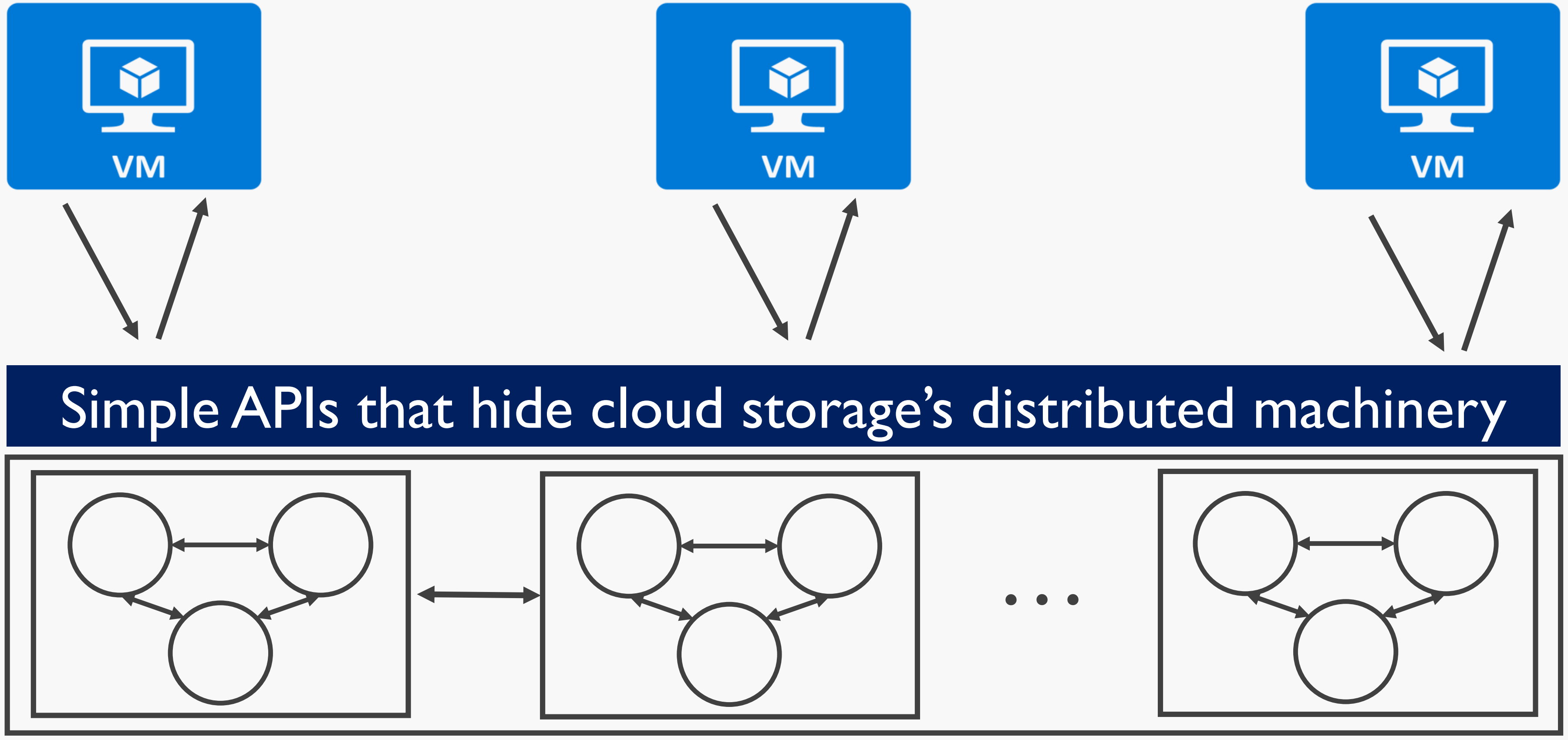
Realizing the fault-tolerance promise of cloud storage using locks with intent

Srinath Setty, Chunzhi Su,[*] Jacob R. Lorch, Lidong Zhou,
Hao Chen,[§] Parveen Patel, and Jinglei Ren

Microsoft Research
[*]The University of Texas at Austin
[§]Shanghai Jiao Tong University

Cloud application atop cloud storage is a recent model of distributed systems

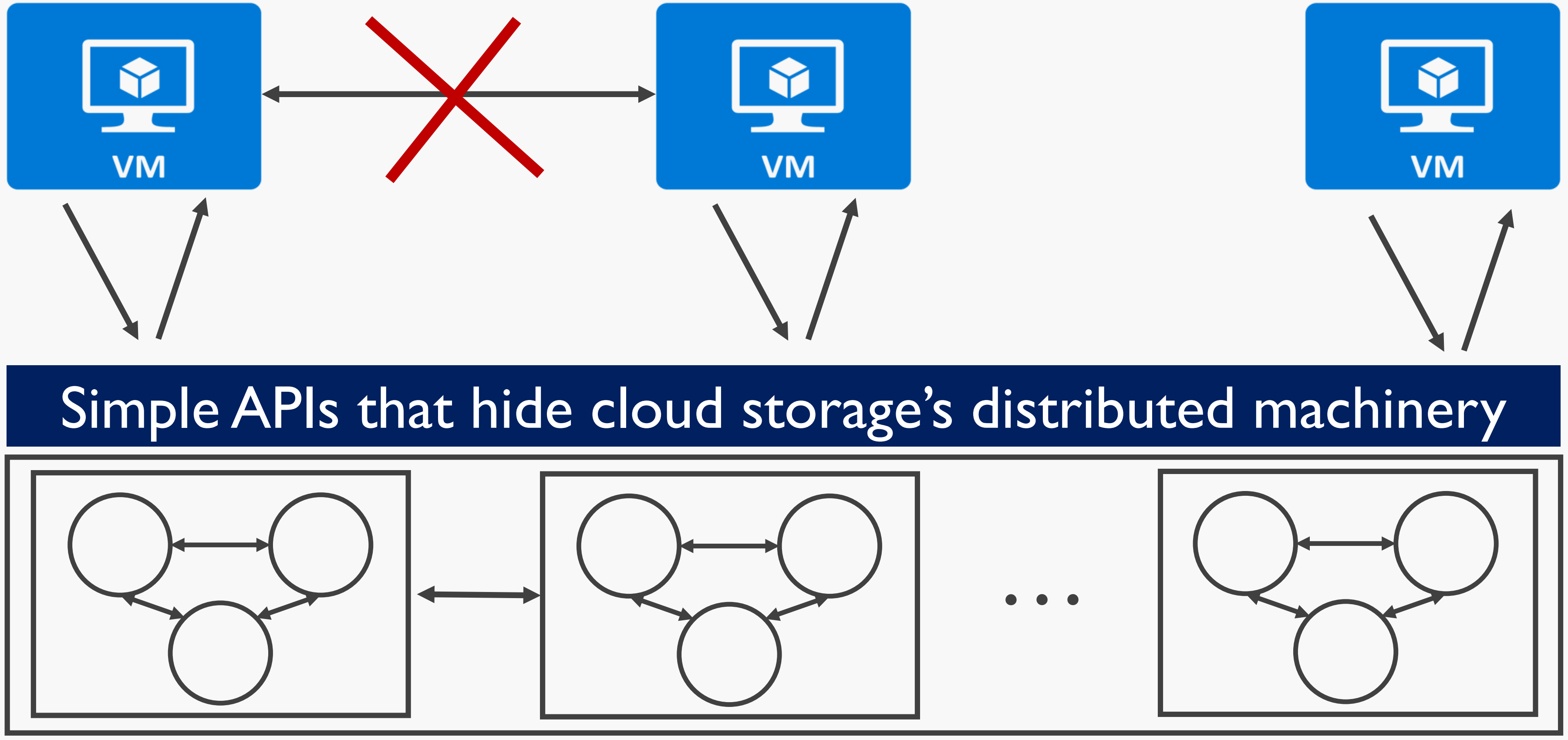


Application's computation is distributed

Application's state is in reliable cloud storage

Cloud application atop cloud storage is a recent model of distributed systems

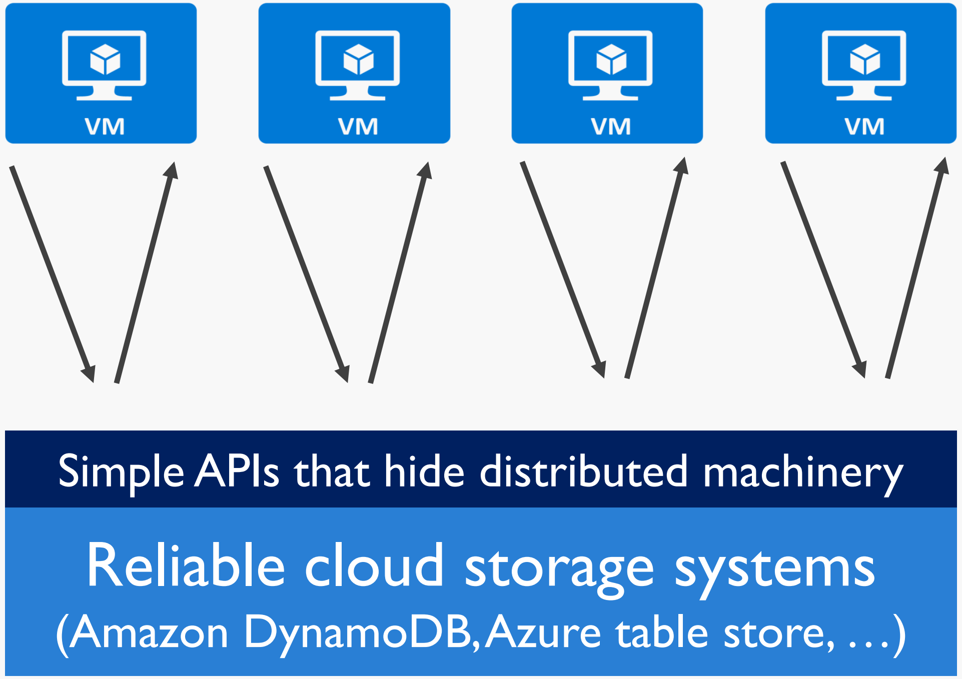
No distributed coordination among VMs



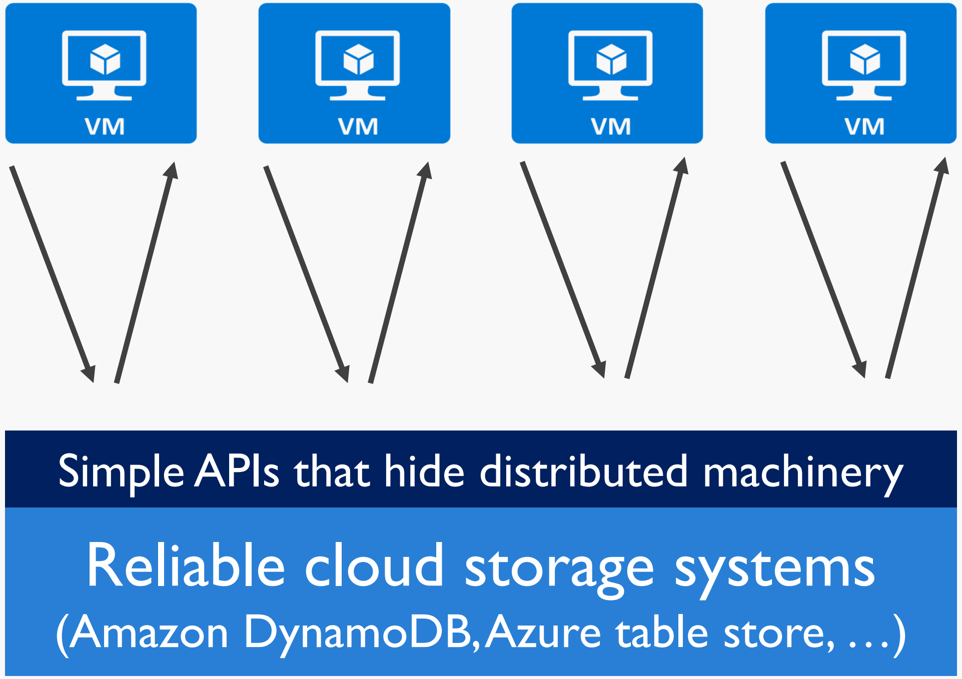
Application's computation is distributed

Application's state is in reliable cloud storage

This architecture poses an interesting problem

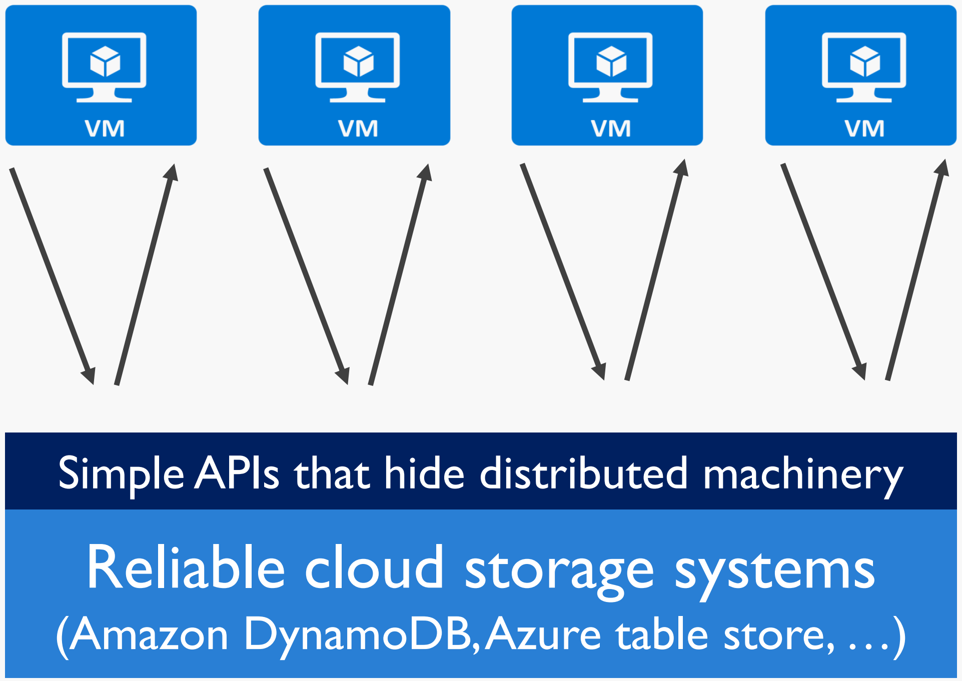


This architecture poses an interesting problem



Application processes or VMs can fail

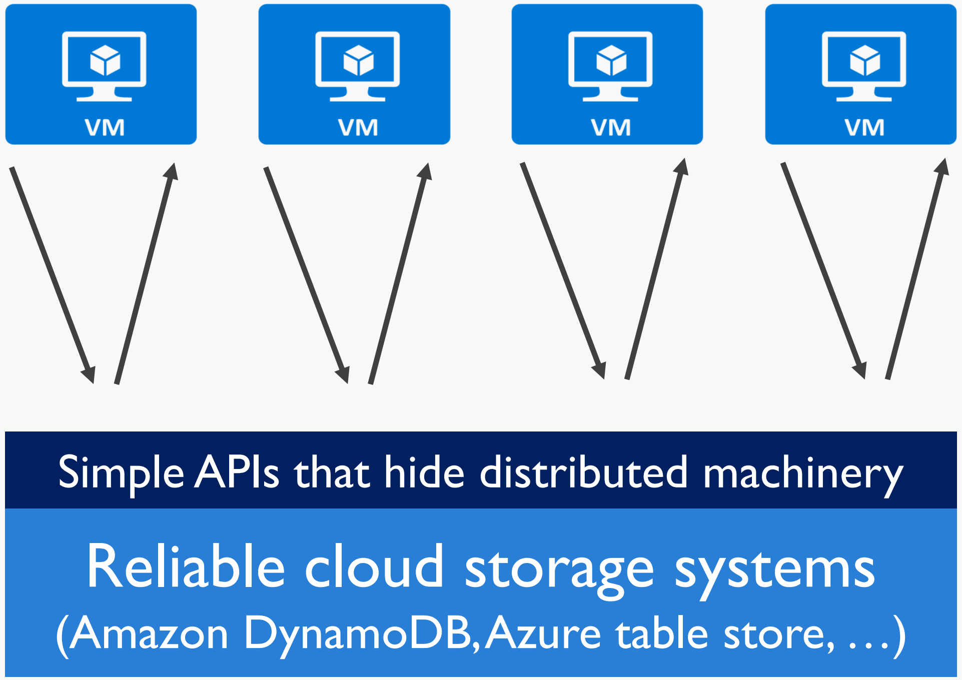
This architecture poses an interesting problem



Application processes or VMs can fail

Network can drop/reorder messages

This architecture poses an interesting problem



Application processes or VMs can fail

Network can drop/reorder messages

Such failures can introduce inconsistencies to application's state

Applications have to maintain invariants over their state

Example: Consistency between application's data and indexes

Applications have to maintain invariants over their state

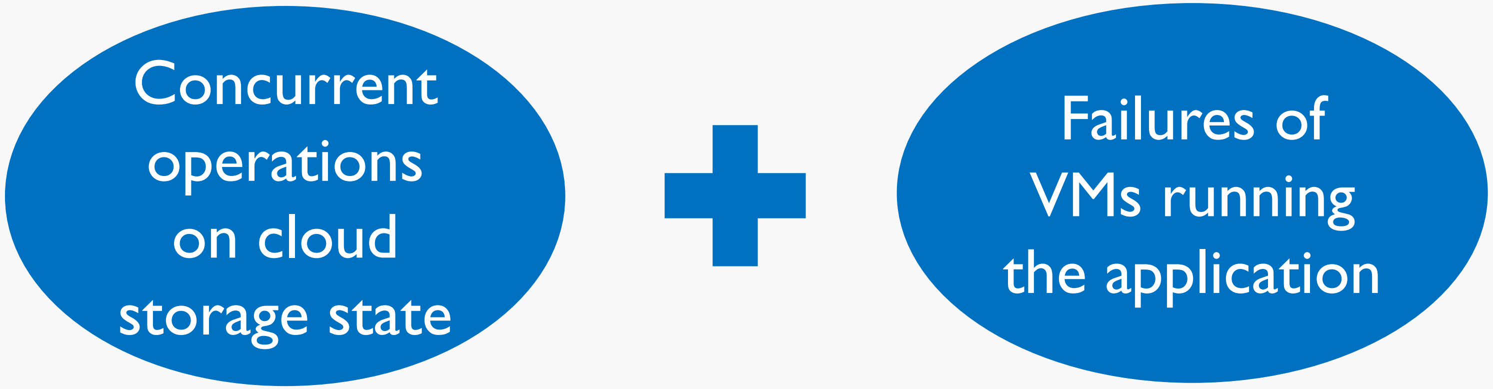
Example: Consistency between application's data and indexes

Invariants should hold even with:

Applications have to maintain invariants over their state

Example: Consistency between application's data and indexes

Invariants should hold even with:



Applications have to maintain invariants over their state

Example: Consistency between application's data and indexes

Invariants should hold even with:

Applications have to maintain invariants over their state

Example: Consistency between application's data and indexes

Invariants should hold even with:

Worse: APIs of cloud storage offer little support for this

Applications have to maintain invariants over their state

Example: Consistency between application's data and indexes

Invariants should hold even with:

Worse: APIs of cloud storage offer little support for this

Target systems: Azure table storage, Amazon DynamoDB, etc.

Applications have to maintain invariants over their state

Example: Consistency between application's data and indexes

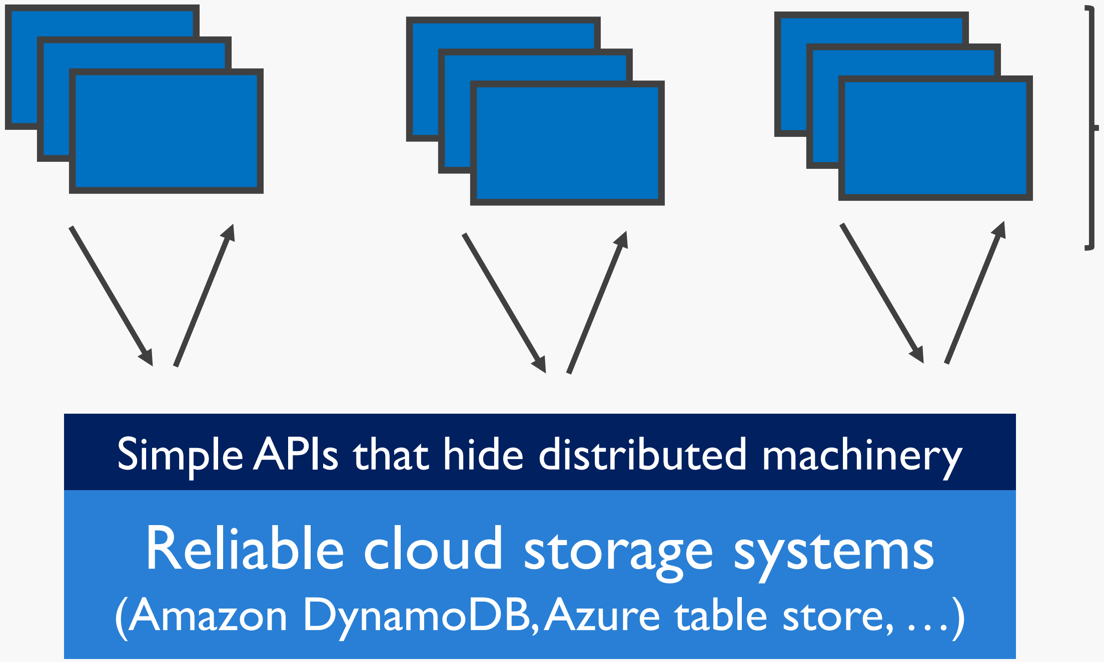
Invariants should hold even with:

Worse: APIs of cloud storage offer little support for this

Target systems: Azure table storage, Amazon DynamoDB, etc.

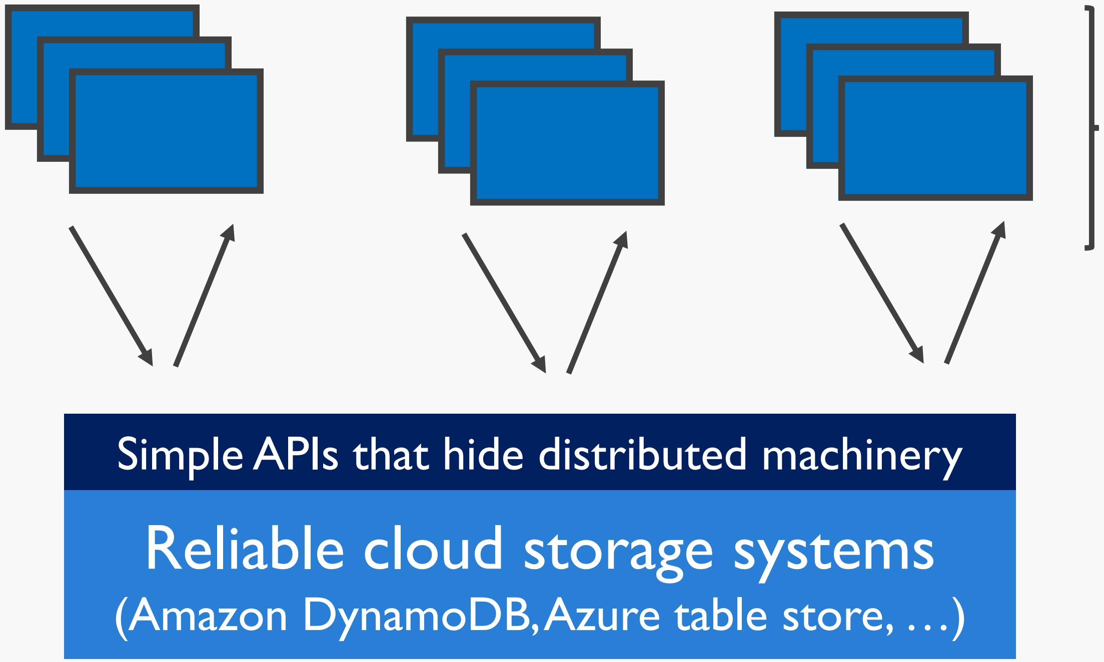
Note: Other cloud storage systems (e.g., Aurora, Azure SQL) offer support for failure handling, but they have different scaling, or monetary cost profiles

A text book solution



Replicate each VM using Paxos

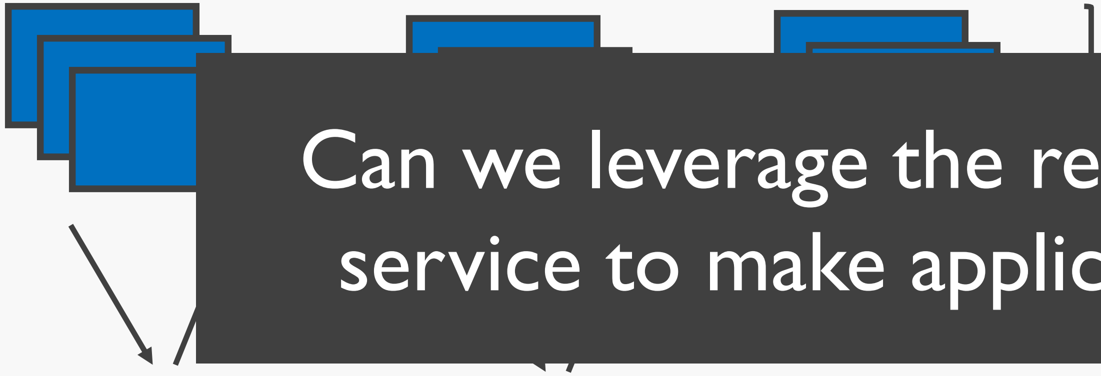
A text book solution



Replicate each VM using Paxos

Seems wasteful: storage uses replication for fault tolerance

A text book solution



Can we leverage the reliability from the storage service to make applications tolerate failures?

Simple APIs that hide distributed machinery

Reliable cloud storage systems
(Amazon DynamoDB, Azure table store, …)

Seems wasteful: storage uses replication for fault tolerance

Highlights of our system Olive

Powerful new primitives: intents and locks with intent

- Exactly-once execution semantics
- Mutual exclusion; locked objects associated with intents
- Eventual progress

Highlights of our system Olive

Powerful new primitives: intents and locks with intent

- Exactly-once execution semantics
- Mutual exclusion; locked objects associated with intents
- Eventual progress

Intent

Arbitrary snippet of code, with calls to cloud storage

Highlights of our system Olive

Powerful new primitives: intents and locks with intent

- Exactly-once execution semantics
- Mutual exclusion; locked objects associated with intents
- Eventual progress

Intent

Arbitrary snippet of code, with calls to cloud storage

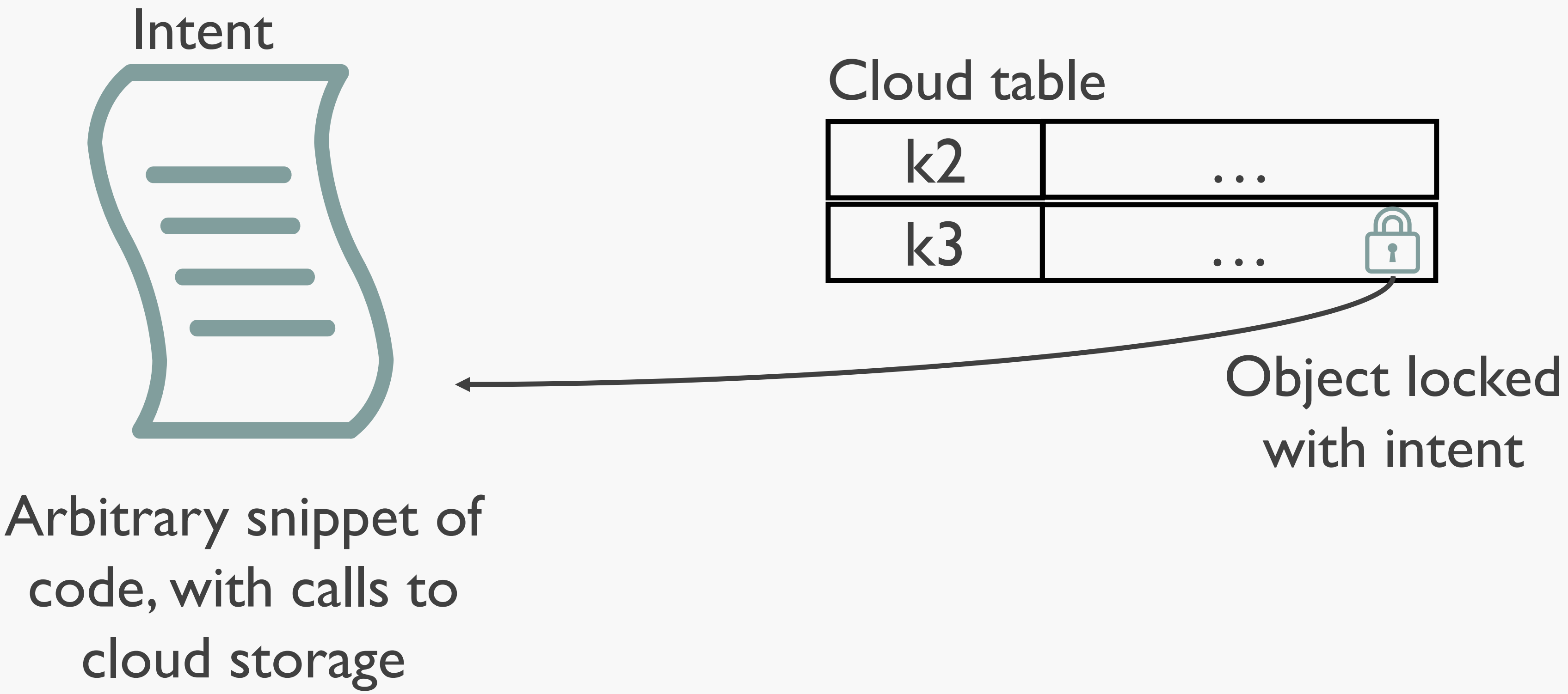
Cloud table

k2	…
k3	…

Highlights of our system Olive

Powerful new primitives: intents and locks with intent

- Exactly-once execution semantics
- Mutual exclusion; locked objects associated with intents
- Eventual progress



Highlights of our system Olive

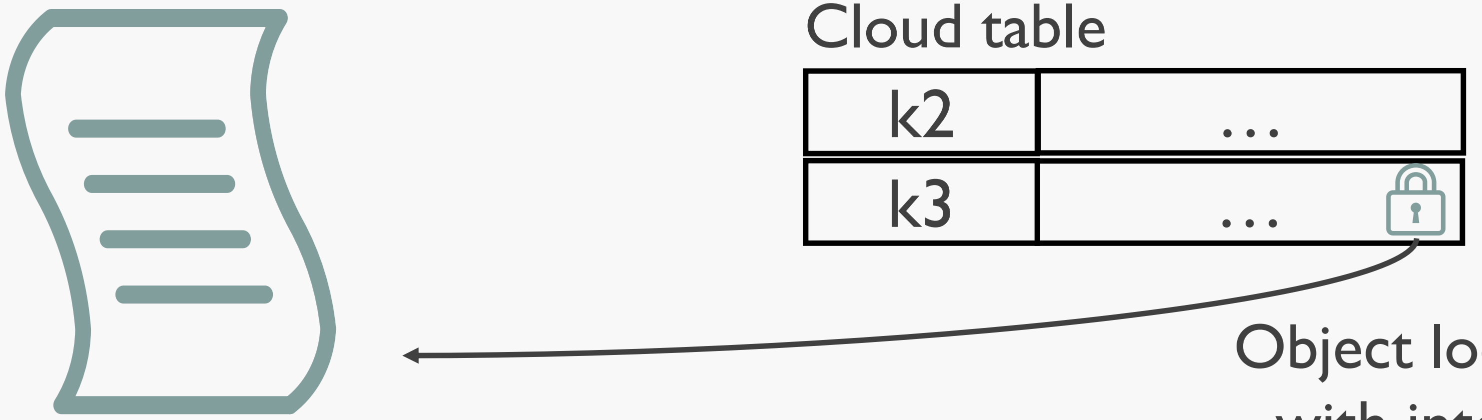
Powerful new primitives: intents and locks with intent

- Exactly-once execution semantics
- Mutual exclusion; locked objects associated with intents
- Eventual progress

Automatic failure handling and simplify concurrency

Intent

Cloud table



Object locked with intent

Arbitrary snippet of code, with calls to cloud storage

Highlights of our system Olive

Powerful new primitives: intents and locks with intent

- Exactly-once execution semantics
- Mutual exclusion; locked objects associated with intents
- Eventual progress

Automatic failure handling and simplify concurrency

New mechanisms to implement this abstraction

- Distributed atomic affinity logging (DAAL)
- Intent collector

Highlights of our system Olive

Powerful new primitives: intents and locks with intent

- Exactly-once execution semantics
- Mutual exclusion; locked objects associated with intents
- Eventual progress

Automatic failure handling and simplify concurrency

New mechanisms to implement this abstraction

- Distributed atomic affinity logging (DAAL)
- Intent collector

Require no modifications to storage; applies generally

Highlights of our system Olive

Powerful new primitives: intents and locks with intent

- Exactly-once execution semantics
- Mutual exclusion; locked objects associated with intents
- Eventual progress

Automatic failure handling and simplify concurrency

New mechanisms to implement this abstraction

- Distributed atomic affinity logging (DAAL)
- Intent collector

Require no modifications to storage; applies generally

Built several real-world, fault-tolerant cloud services

- Live re-partitioning of tables
- Snapshotting service
- ACID transactions
- …

Highlights of our system Olive

Powerful new primitives: intents and locks with intent

- Exactly-once execution semantics
- Mutual exclusion; locked objects associated with intents
- Eventual progress

Automatic failure handling and simplify concurrency

New mechanisms to implement this abstraction

- Distributed atomic affinity logging (DAAL)
- Intent collector

Require no modifications to storage; applies generally

Built several real-world, fault-tolerant cloud services

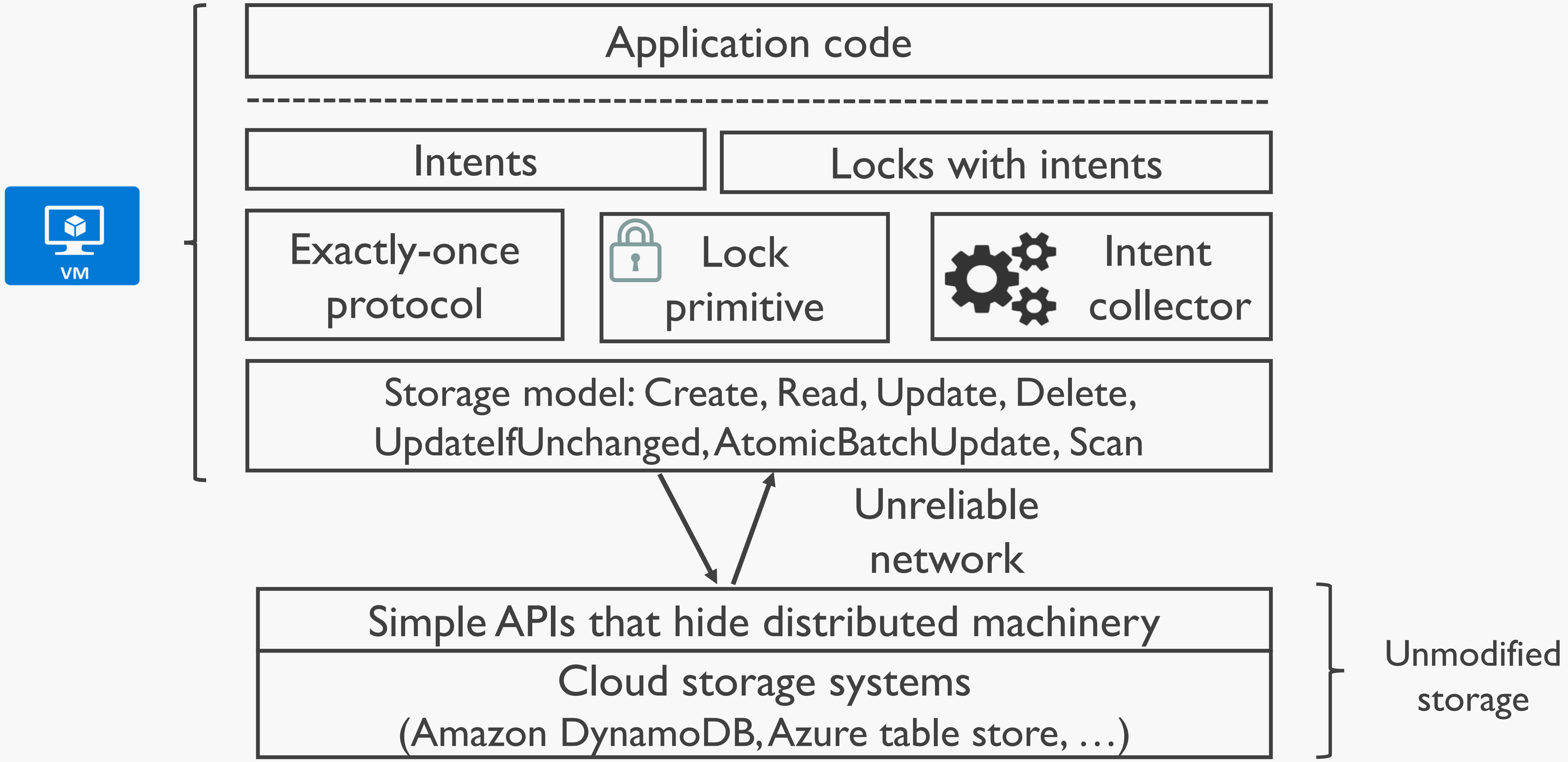
- Live re-partitioning of tables
- Snapshotting service
- ACID transactions
- …

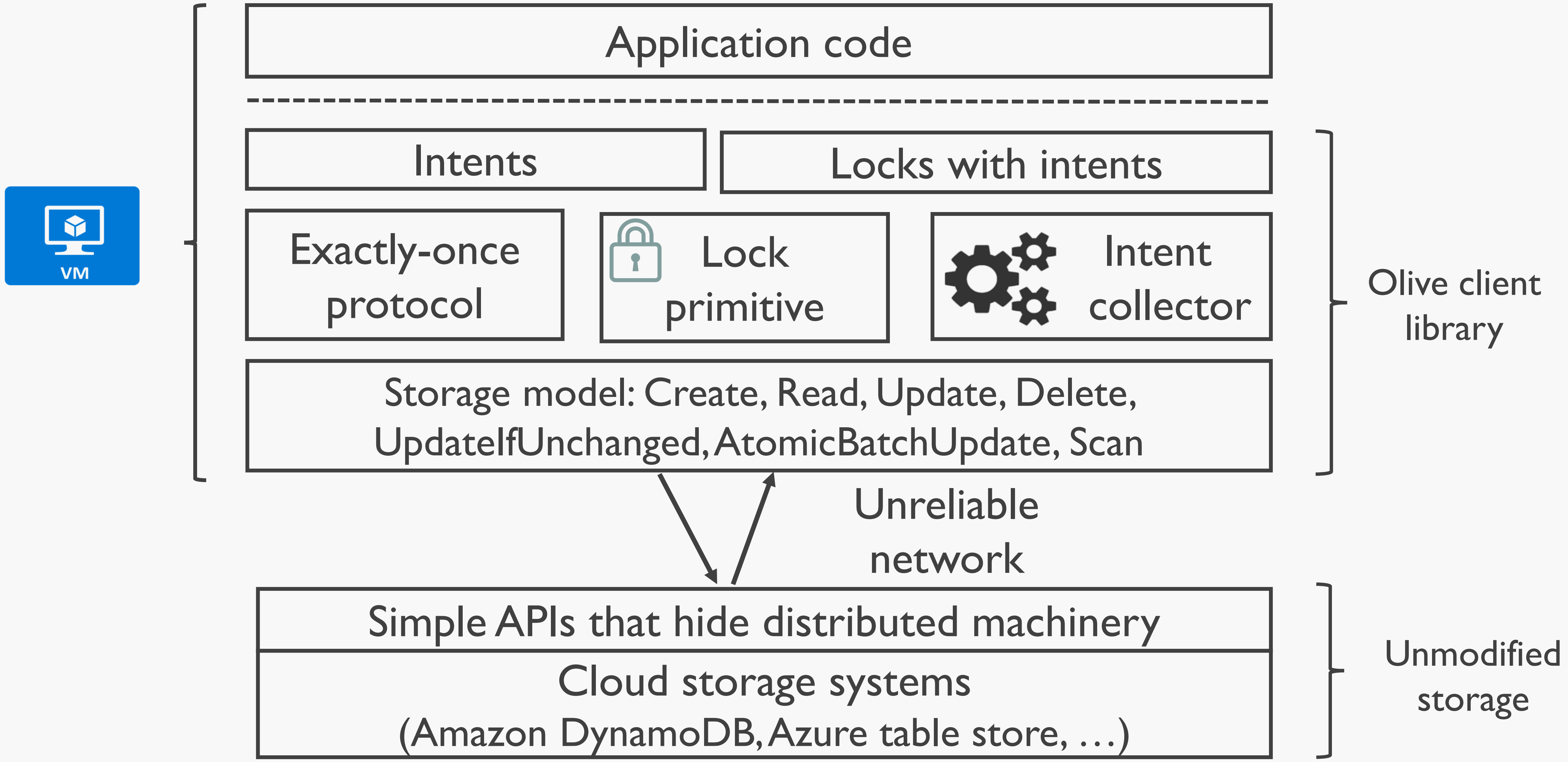
30-80% less code than building directly on cloud storage APIs

Rest of this talk

Olive's abstractions and mechanisms

Evaluation of Olive


Application code
Intents
Locks with intents
Exactly-once protocol
Lock primitive
Intent collector
Storage model: Create, Read, Update, Delete, UpdateIfUnchanged, AtomicBatchUpdate, Scan
Unreliable network
Simple APIs that hide distributed machinery
Cloud storage systems (Amazon DynamoDB, Azure table store, …)
Unmodified storage
VM


VM
Application code
Intents
Locks with intents
Exactly-once protocol
Lock primitive
Intent collector
Storage model: Create, Read, Update, Delete, UpdateIfUnchanged, AtomicBatchUpdate, Scan
Olive client library
Unreliable network
Simple APIs that hide distributed machinery
Cloud storage systems
(Amazon DynamoDB, Azure table store, …)
Unmodified storage

```
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

Intent = An arbitrary snippet of code:

- Cloud storage operations
- Local computation (loops, recursion, control flow, ...)

```
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

Intent = An arbitrary snippet of code:

- Cloud storage operations
- Local computation (loops, recursion, control flow, ...)

Goal of exactly-once execution

Code should run as if it is executed by a single, failure-free client

```
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

Intent = An arbitrary snippet of code:

- Cloud storage operations
- Local computation (loops, recursion, control flow, ...)

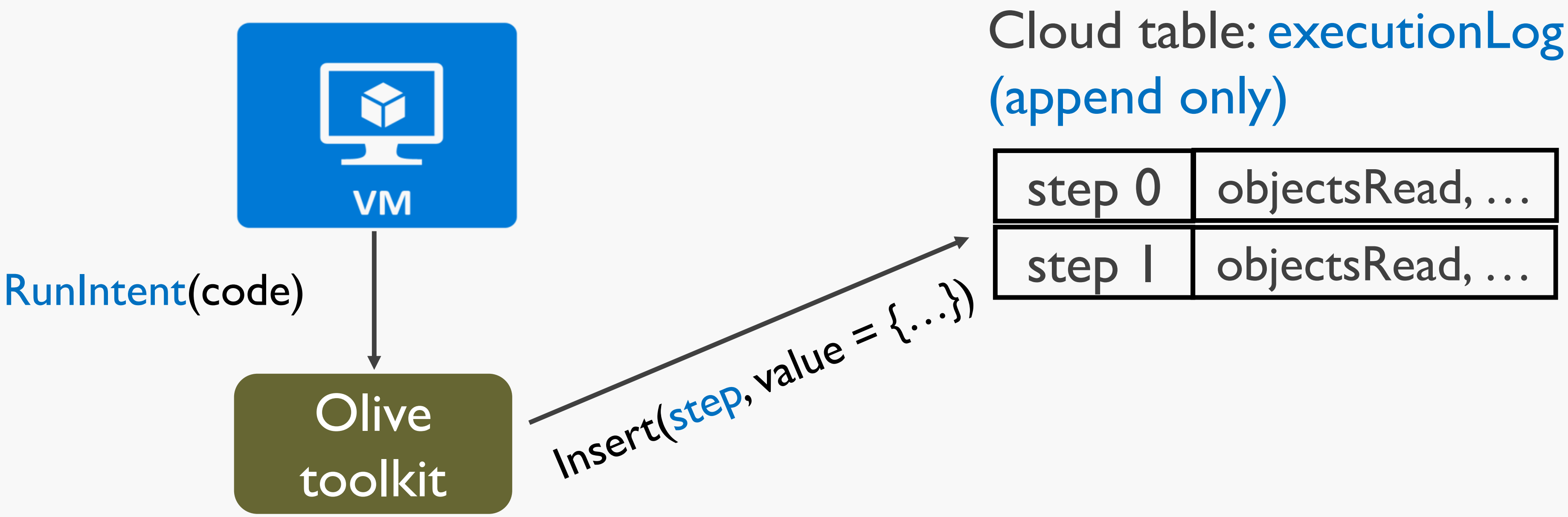
Goal of exactly-once execution

Code should run as if it is executed by a single, failure-free client

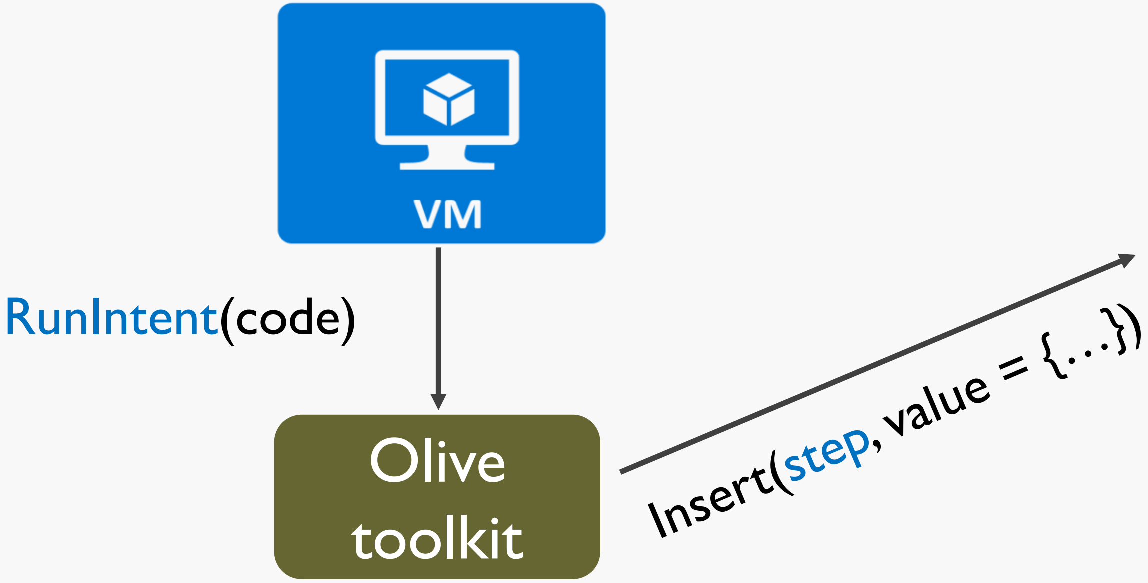
Challenges for exactly-once execution

- Clients can fail partway
- Imperfect failure detection → multiple, concurrent intent executions

Olive records in reliable cloud storage whenever a step of an intent is executed



Olive records in reliable cloud storage whenever a step of an intent is executed

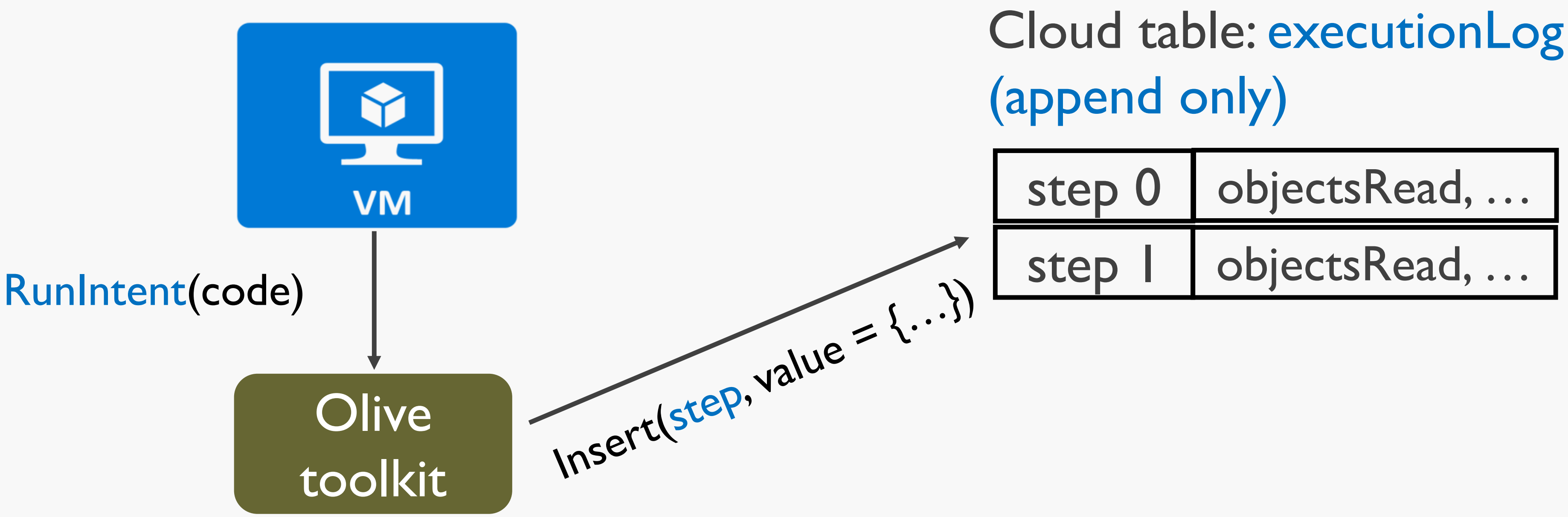


Cloud table: executionLog (append only)

step 0	objectsRead, …
step 1	objectsRead, …

To execute read:

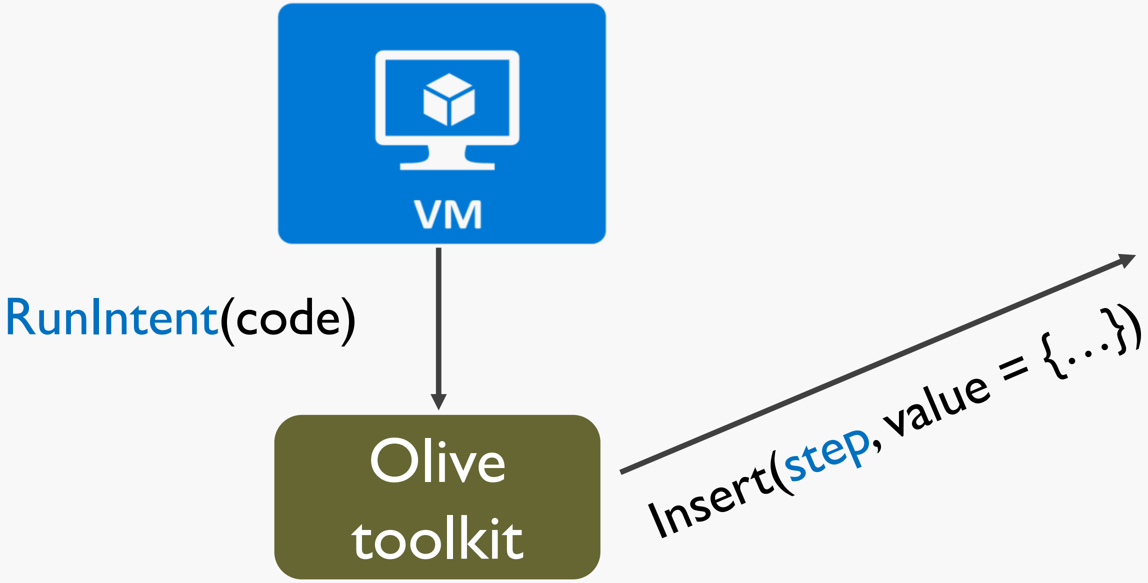
Olive records in reliable cloud storage whenever a step of an intent is executed



To execute read:

1. Execute the read normally

Olive records in reliable cloud storage whenever a step of an intent is executed



Cloud table: executionLog (append only)

step 0	objectsRead, …
step 1	objectsRead, …

To execute read:

1. Execute the read normally
2. Append an entry to executionLog

Olive records in reliable cloud storage whenever a step of an intent is executed

RunIntent(code)

Cloud table: executionLog (append only)

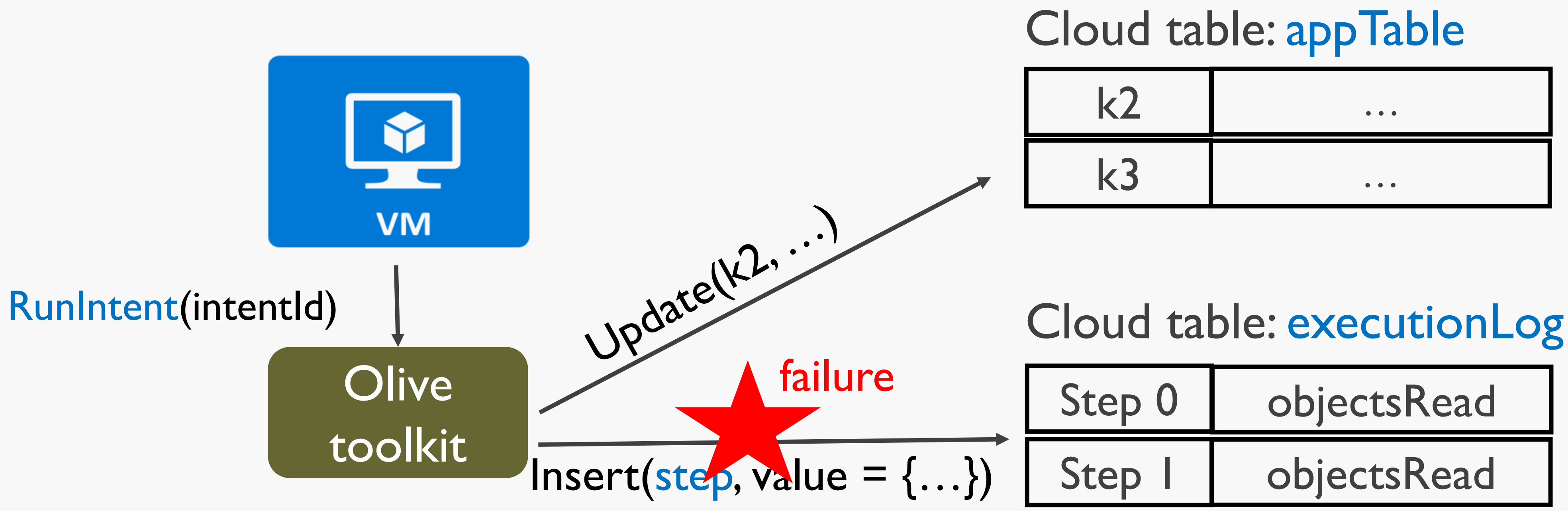
step 0	objectsRead, …
step 1	objectsRead, …

This will not work for executing an update inside an intent

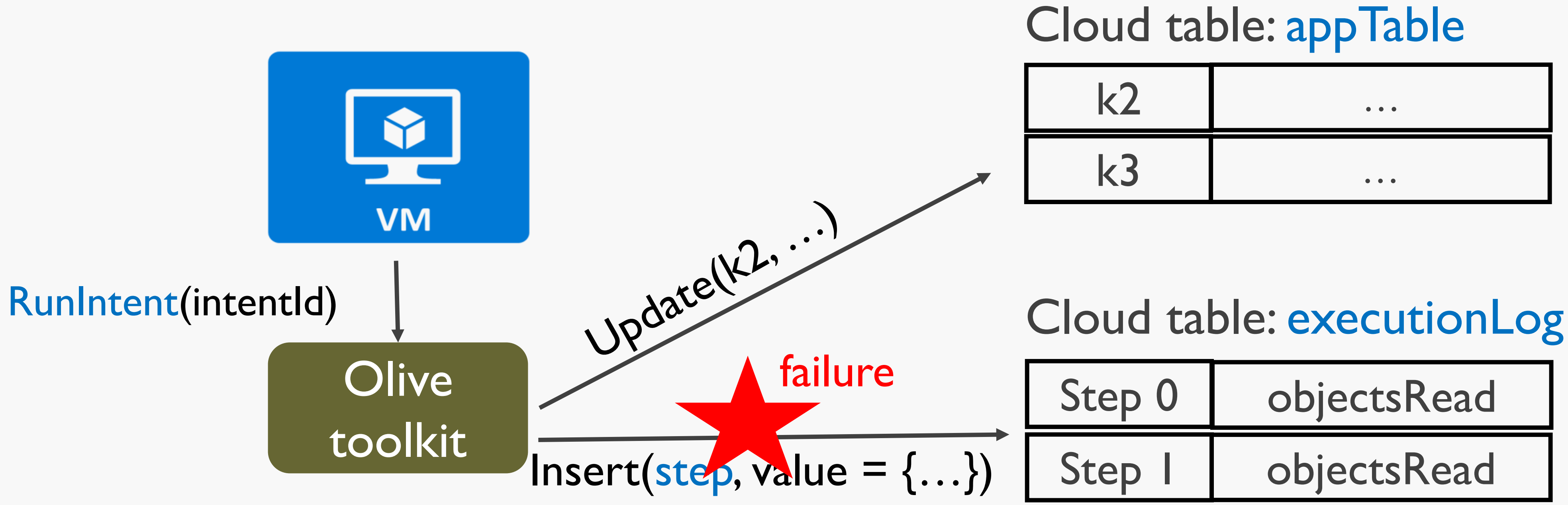
To execute read:

1. Execute the read normally
2. Append an entry to executionLog

Executing an update and recording it in executionLog must be atomic

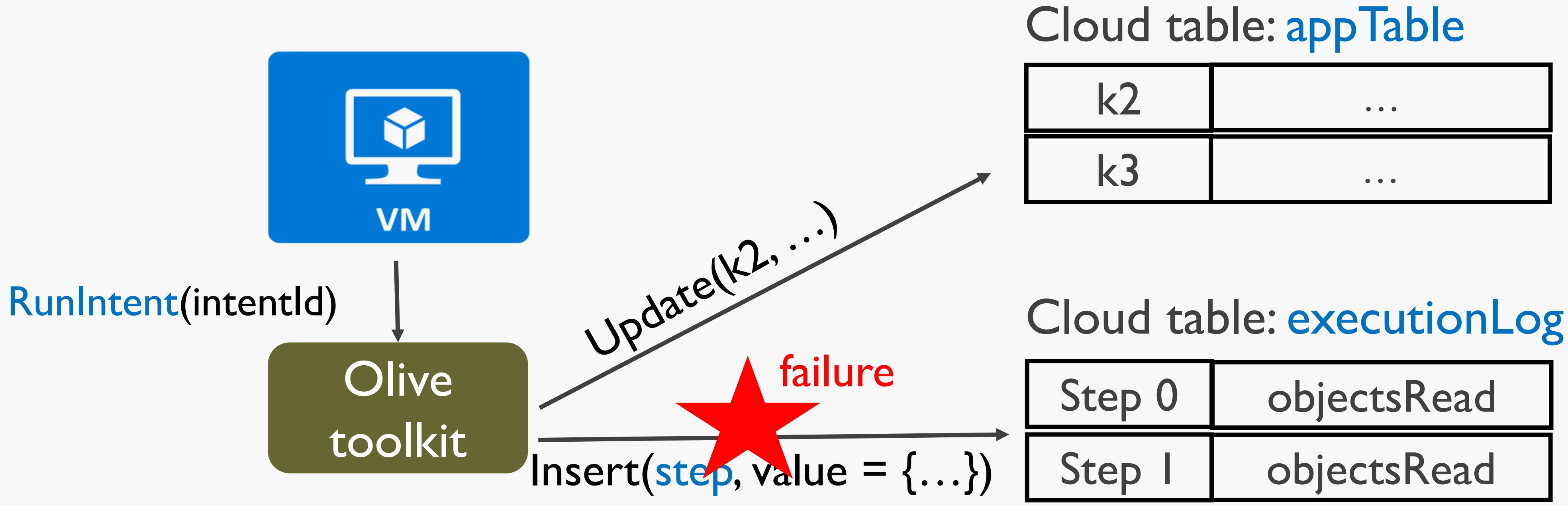


Executing an update and recording it in executionLog must be atomic



Failure to record after executing → violation of exactly-once

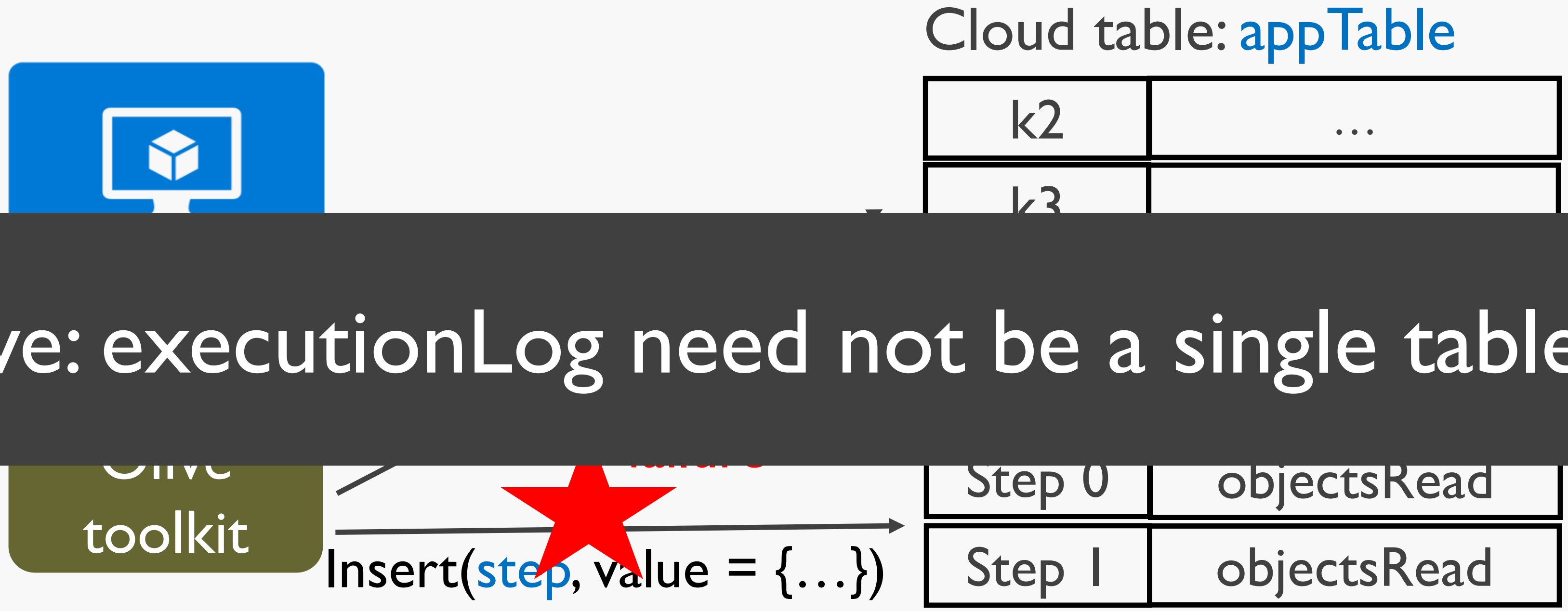
Executing an update and recording it in executionLog must be atomic



Failure to record after executing → violation of exactly-once

Storage systems we target do not support cross-table atomic updates

Executing an update and recording it in executionLog must be atomic



Observe: executionLog need not be a single table

Failure to record after executing → violation of exactly-once

Storage systems we target do not support cross-table atomic updates

Olive introduces: Distributed atomic affinity logging (DAAL)

Olive introduces: Distributed atomic affinity logging (DAAL)

Leverage AtomicBatchUpdate for objects in the same shard or partition.

Olive introduces: Distributed atomic affinity logging (DAAL)

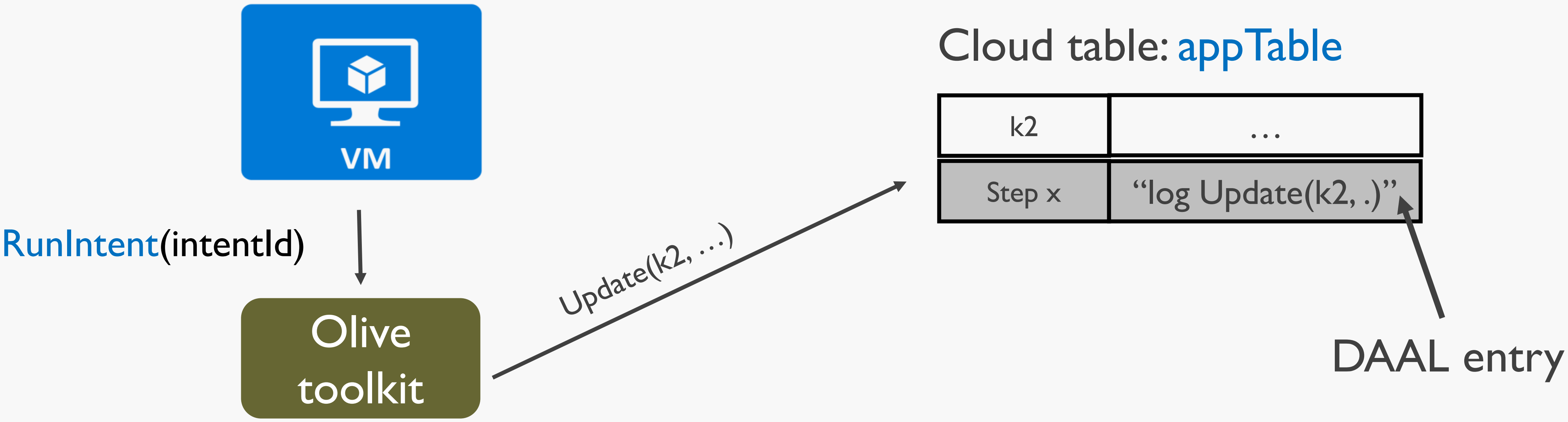
Leverage AtomicBatchUpdate for objects in the same shard or partition.

- Azure table storage, Amazon DynamoDB, MongoDB, Cassandra, etc.

Olive introduces: Distributed atomic affinity logging (DAAL)

Leverage AtomicBatchUpdate for objects in the same shard or partition.

- Azure table storage, Amazon DynamoDB, MongoDB, Cassandra, etc.



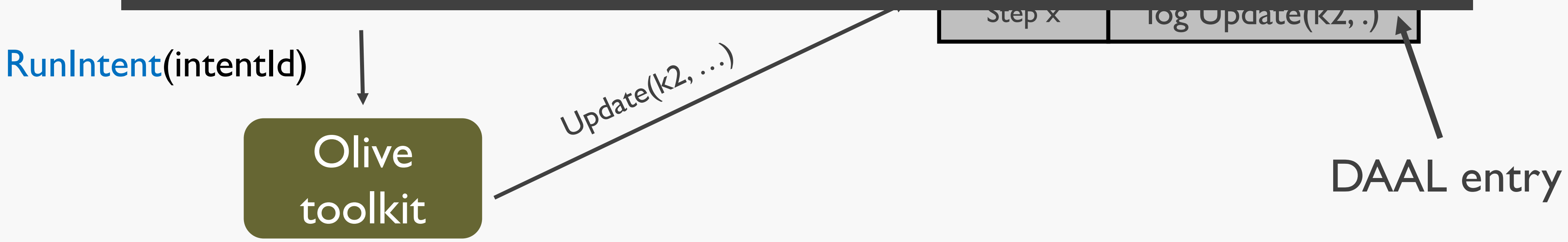
Olive introduces: Distributed atomic affinity logging (DAAL)

Leverage AtomicBatchUpdate for objects in the same shard or partition.

- Azure table storage, Amazon DynamoDB, MongoDB, Cassandra, etc.

executionLog is not a single, global table:

- A global cloud table for recording read operations, and
- DAAL entries spread throughout



Benefits of Olive

```
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

Benefits of Olive

```
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

An Intent executes in entirety

Benefits of Olive

```
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

An Intent executes in entirety

Without intents: "Does failing at line i violate any invariant?"

Benefits of Olive

```
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

An Intent executes in entirety

Without intents: "Does failing at line i violate any invariant?"

```
def migrateIntent(curTable, futTable, obj):
  curTable.Lock(obj.key)
```

Benefits of Olive

```python
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

An Intent executes in entirety

Without intents: "Does failing at line i violate any invariant?"

```python
def migrateIntent(curTable, futTable, obj):
  curTable.Lock(obj.key)
```

```python
def migrateIntent(curTable, futTable, obj):
  curTable.Lock(obj.key)
  futTable.Create(obj.key, obj)
```

Benefits of Olive

```
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

An Intent executes in entirety

Without intents: "Does failing at line i violate any invariant?"

```
def migrateIntent(curTable, futTable, obj):
  curTable.Lock(obj.key)
```

```
def migrateIntent(curTable, futTable, obj):
  curTable.Lock(obj.key)
  futTable.Create(obj.key, obj)
```

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

Benefits of Olive

```
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 = tableList[0]
  if (tablesList.len == 1):
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

An Intent executes in entirety

Without intents: "Does failing at line i violate any invariant?"

```
def migrateIntent(curTable, futTable, obj):
  curTable.Lock(obj.key)
```

```
def migrateIntent(curTable, futTable, obj):
  curTable.Lock(obj.key)
  futTable.Create(obj.key, obj)
```

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

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

Benefits of Olive

```
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,
  for (obj in ObjsTo
    migrateIntent(cu
  metaTable.Update(

def updateObject(key
  pKey = getPartiti
  tablesList = metaT
  curTable = tableLi
  if (tablesList.len
    curTable.UpdateIfUnchanged(key, newObj)
  elif (tablesList.len == 2):
    futTable = tableList[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)
```

An Intent executes in entirety

Without intents: "Does failing at line i violate any invariant?"

Still, the developer must reason about concurrent executions of intents

```
utTable, obj):

j)
```

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

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

Locks are well-studied concurrency control primitive

CloudTable.Lock(k)

…

CloudTable.Update(k, …)

CloudTable.Unlock(k)

Client 1

Client 2

CloudTable.Lock(k)

…

CloudTable.Update(k, …)

CloudTable.Unlock(k)

Locks are well-studied concurrency control primitive

CloudTable.Lock(k)

…

CloudTable.Update(k, …)

CloudTable.Unlock(k)

Client 1

Client 2

CloudTable.Lock(k)

…

CloudTable.Update(k, …)

CloudTable.Unlock(k)

Executions will be serialized, no interleavings

Locks are well-studied concurrency control primitive

Locks are dangerous, since clients can fail after acquiring a lock

CloudTable.Lock(k)
…
CloudTable.Update(k, …)
CloudTable.Unlock(k)

Client 1

Client 2

CloudTable.Lock(k)
…
CloudTable.Update(k, …)
CloudTable.Unlock(k)

Executions will be serialized, no interleavings

Olive composes locks with intents

Olive composes locks with intents

Locks are owned by intents, not client VMs → any client can unlock an object by executing the associated intent

Olive composes locks with intents

Locks are owned by intents, not client VMs → any client can unlock an object by executing the associated intent

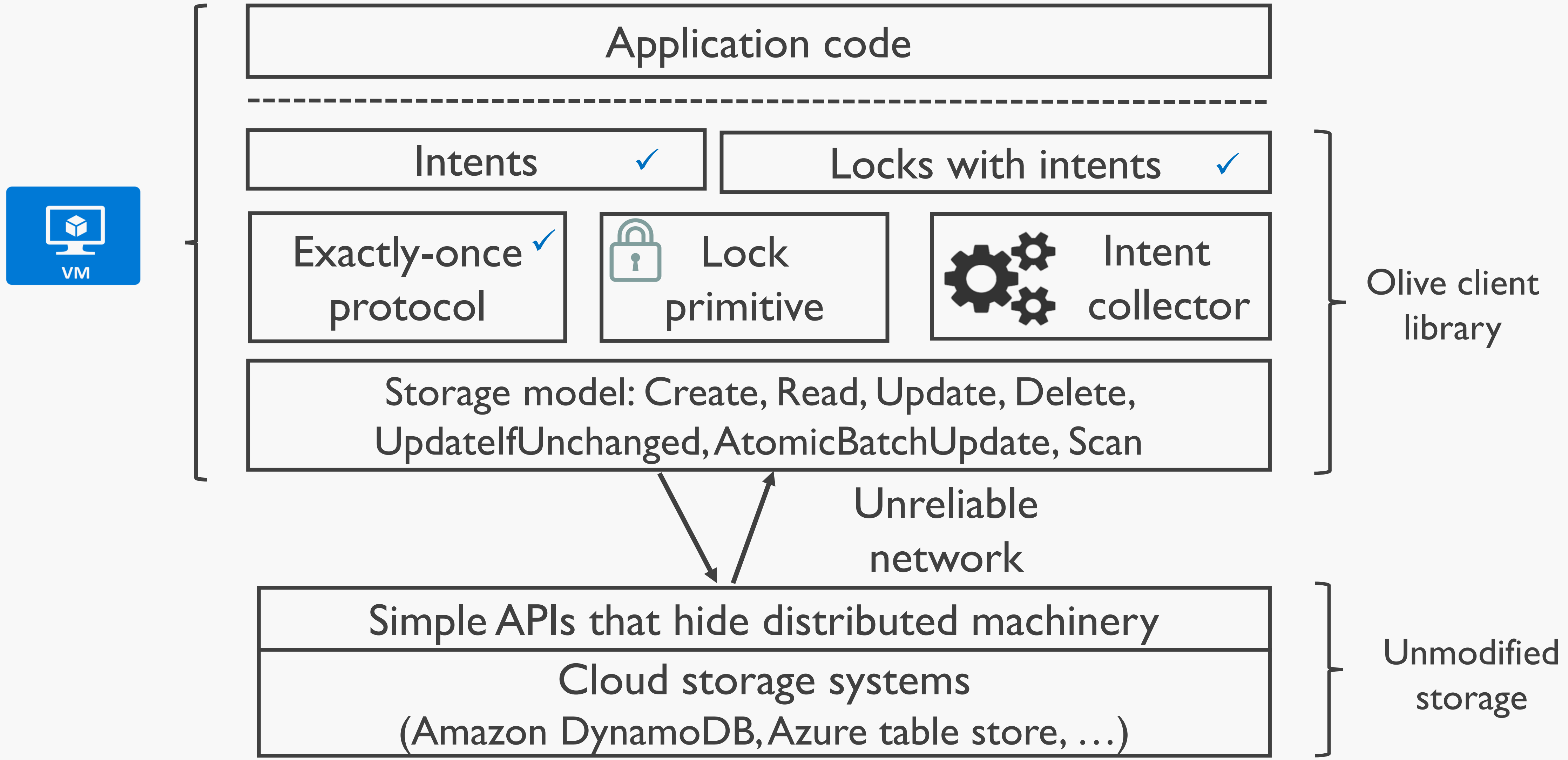
	Traditional lock	Locks with intent
Mutual exclusion	Yes	Yes
Survives client failures	No	Yes

Olive composes locks with intents

Locks are owned by intents, not client VMs → any client can unlock an object by executing the associated intent

	Traditional lock	Locks with intent
Mutual exclusion	Yes	Yes
Survives client failures	No	Yes

Overall benefit: simplifies reasoning about concurrency in the presence of failures (see our paper)


VM
Application code
Intents ✓
Locks with intents ✓
Exactly-once protocol ✓
Lock primitive
Intent collector
Storage model: Create, Read, Update, Delete,
UpdateIfUnchanged, AtomicBatchUpdate, Scan
Olive client library
Unreliable network
Simple APIs that hide distributed machinery
Cloud storage systems
(Amazon DynamoDB, Azure table store, …)
Unmodified storage

Implementation of Olive

Implemented 2,000 lines of C#

Abstracts the underlying storage system with a C# interface

- We write code to map that interface to different storage systems: 38 lines of code for Azure table store, 107 lines of code for Amazon DynamoDB

Can be extended easily to Cassandra, MongoDB, Azure DocumentDB, other cloud storage services, etc.

✓ Olive's abstractions and mechanisms

Evaluation of Olive

Evaluation questions

- Do Olive's abstractions simplify building fault-tolerant applications?

- How do Olive-based artifacts perform relative to alternatives?

Does Olive's locks with intent simplify building fault-tolerant applications?

Metric: lines of code, with and without Olive

Does Olive's locks with intent simplify building fault-tolerant applications?

Metric: lines of code, with and without Olive

Service	Without Olive	With Olive
Snapshots	987	665
OCC-transactions	2,201	408
Live table re-partitioning	2,116	474

Does Olive's locks with intent simplify building fault-tolerant applications?

Metric: lines of code, with and without Olive

Service	Without Olive	With Olive
Snapshots	987	665
OCC-transactions	2,201	408
Live table re-partitioning	2,116	474

Note: Olive's library is 2,000 lines of code

Does Olive's locks with intent simplify building fault-tolerant applications?

Metric: lines of code, with and without Olive

Service	Without Olive	With Olive
Snapshots	987	665
OCC-transactions	2,201	408
Live table re-partitioning	2,116	474

Note: Olive's library is 2,000 lines of code

Key takeaway: Olive reduces lines of code by 30–80%

Does Olive's locks with intent simplify building fault-tolerant applications?

Metric: lines of code, with and without Olive

Service	Without Olive	With Olive
Snapshots	987	665
OCC-transactions	2,201	408
Live table re-partitioning	2,116	474

Note: Olive's library is 2,000 lines of code

Key takeaway: Olive reduces lines of code by 30–80%

Our paper discusses how Olive simplifies reasoning about correctness

How do Olive-based artifacts perform relative to alternatives?

How do Olive-based artifacts perform relative to alternatives?

- Consider snapshotting service

How do Olive-based artifacts perform relative to alternatives?

- Consider snapshotting service

- Baseline: database service in the cloud (Azure SQL)

How do Olive-based artifacts perform relative to alternatives?

- Consider snapshotting service
- Baseline: database service in the cloud (Azure SQL)
- Metric: latency of cloud storage operations (Create, Read, Update)

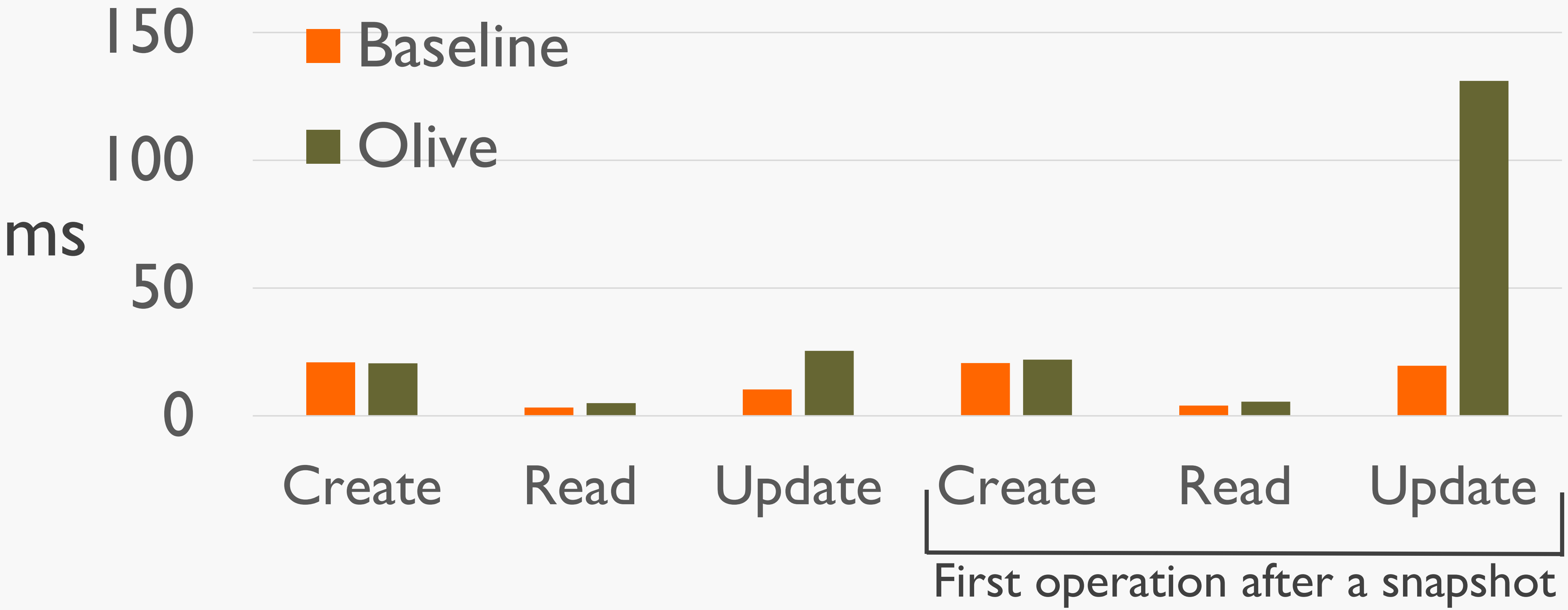
How do Olive-based artifacts perform relative to alternatives?

- Consider snapshotting service
- Baseline: database service in the cloud (Azure SQL)
- Metric: latency of cloud storage operations (Create, Read, Update)
- Olive’s artifact: uses lazy copy-on-write technique

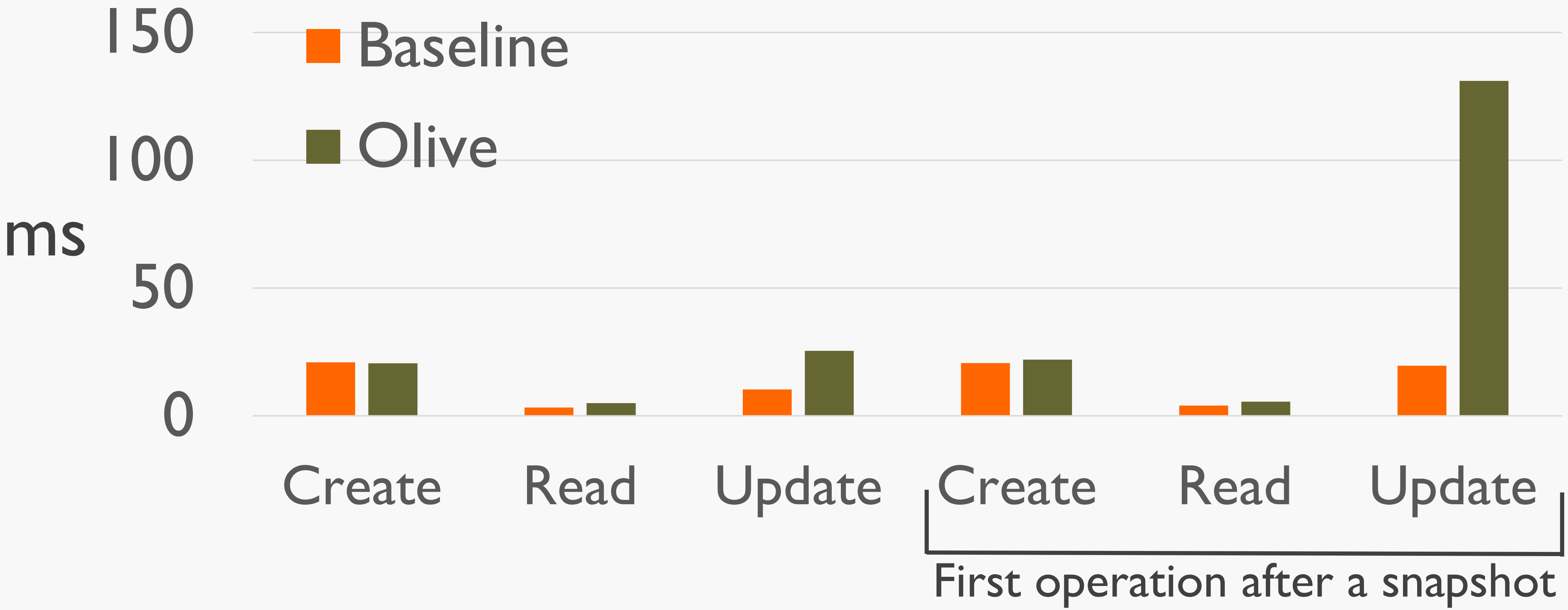
How do Olive-based artifacts perform relative to alternatives?

- Consider snapshotting service
- Baseline: database service in the cloud (Azure SQL)
- Metric: latency of cloud storage operations (Create, Read, Update)
- Olive's artifact: uses lazy copy-on-write technique
- Olive's underlying storage service: Azure table store (US-West)

Performance of Olive-based snapshotting service

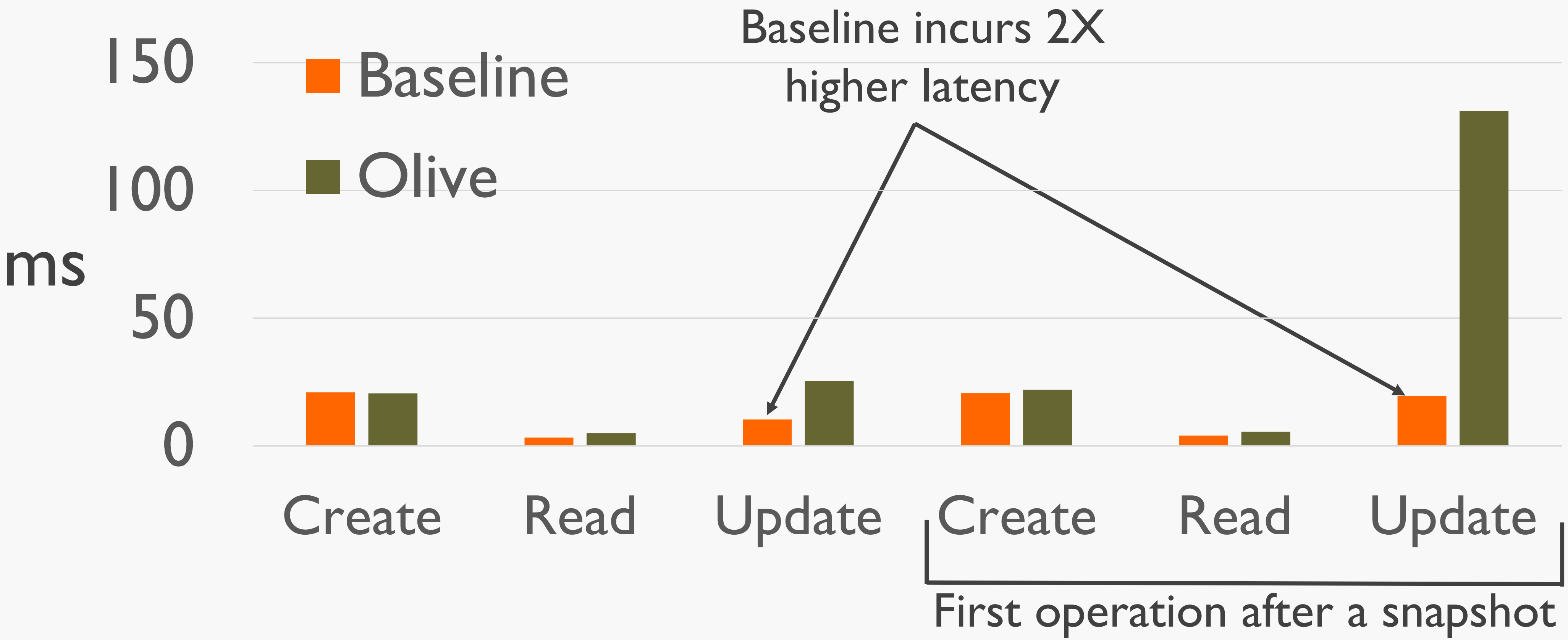


Performance of Olive-based snapshotting service



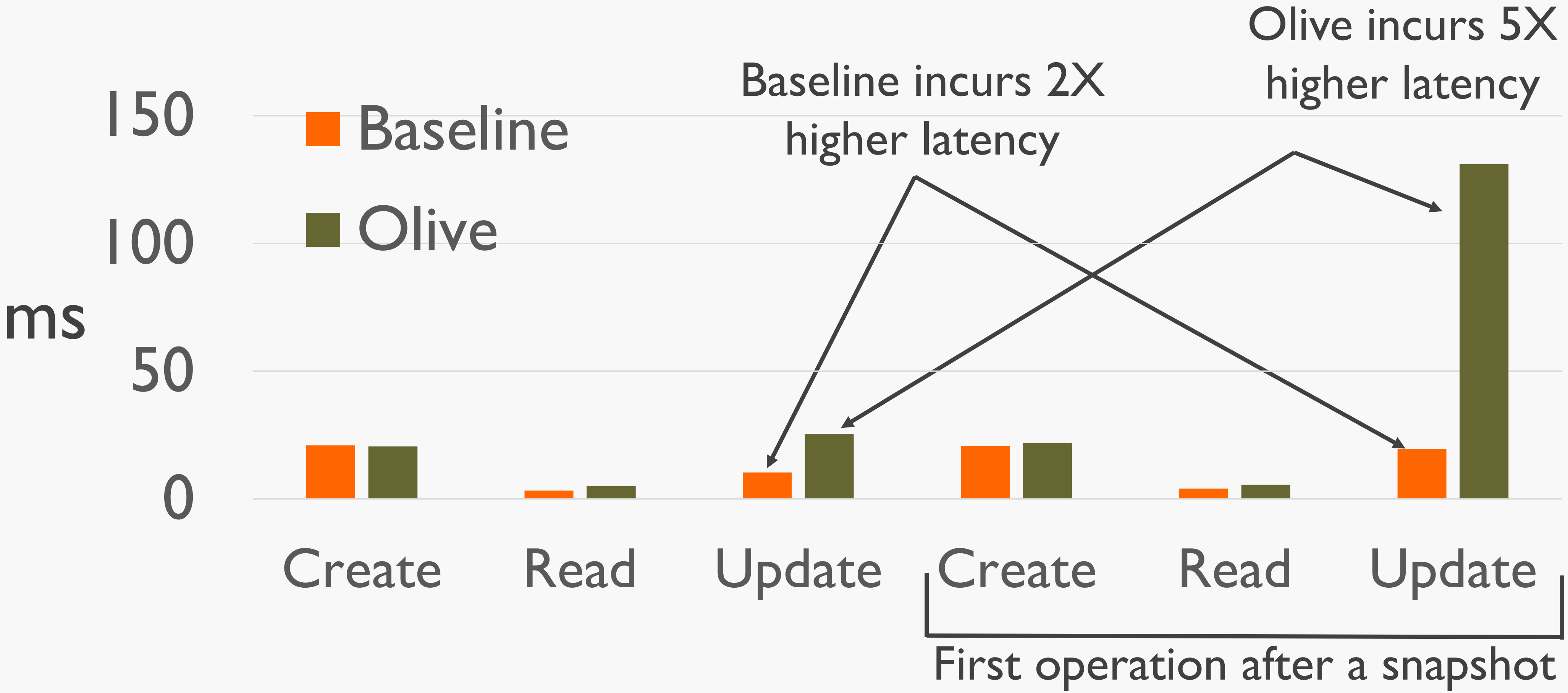
Olive is competitive with the baseline for most operations

Performance of Olive-based snapshotting service



Olive is competitive with the baseline for most operations

Performance of Olive-based snapshotting service



Olive is competitive with the baseline for most operations

Olive relates to many works

State machine replication [Schneider CSUR'90, Lamport TOCS'98, ….]

Failure recovery [Chandy & Ramamoorthy IEEE'72, Lowell et al. OSDI'00], Microreboot [Candea et al. OSDI'04]

Leases [Gray SOSP'89], distributed locks with lease-like expiration [Burrows OSDI'06], revocable locks [Harris & Fraser PPoPP'05]

Write-ahead logging [Astrahan TODS'76, Mohan et al. TODS'92, Olson et al. ATC'99, ...]

Database and distributed transactions [Liskov CACM' 88, Adya et al. ICDE'00, Balakrishnan SOSP'13, Aguilera et al. SOSP'15, …]

Systems that provide exactly-once semantics [Frolund PODC'00, Huang & Garcia ICDE'01, Helland CACM'12, Ramalingam & Vaswani POPL'13, Lee et al. SOSP'15]

Distributed ACID transactions vs. locks with intent

Transactions are simpler to program with, but offer less flexibility

Distributed ACID transactions vs. locks with intent

Transactions are simpler to program with, but offer less flexibility

By decoupling atomicity from isolation, locks with intent:

- Enable consistency levels from weak eventual to strong transactional

Distributed ACID transactions vs. locks with intent

Transactions are simpler to program with, but offer less flexibility

By decoupling atomicity from isolation, locks with intent:

- Enable consistency levels from weak eventual to strong transactional
- Avoid full isolation when not needed

Distributed ACID transactions vs. locks with intent

Transactions are simpler to program with, but offer less flexibility

By decoupling atomicity from isolation, locks with intent:

- Enable consistency levels from weak eventual to strong transactional
- Avoid full isolation when not needed

We provide an intents-based transactional library if they prefer the simplicity of transactions (see our paper for examples)

Distributed ACID transactions vs. locks with intent

Transactions are simpler to program with, but offer less flexibility

By decoupling atomicity from isolation, locks with intent:

- Enable consistency levels from weak eventual to strong transactional
- Avoid full isolation when not needed

We provide an intents-based transactional library if they prefer the simplicity of transactions (see our paper for examples)

If the cloud storage service provided a general transactional interface, locks with intent can leverage it for exactly-once semantics, liveness, etc.

Olive's key takeaways

Cloud applications atop cloud storage pose a new problem: what is the right primitive for making such applications fault tolerant?

Olive's key takeaways

Cloud applications atop cloud storage pose a new problem: what is the right primitive for making such applications fault tolerant?

We propose two new primitives: Intents and locks with intent, which guarantee exactly-once semantics, mutual exclusion, and eventual progress

Olive's key takeaways

Cloud applications atop cloud storage pose a new problem: what is the right primitive for making such applications fault tolerant?

We propose two new primitives: Intents and locks with intent, which guarantee exactly-once semantics, mutual exclusion, and eventual progress

We propose new mechanisms: DAAL and an intent collector

Olive's key takeaways

Cloud applications atop cloud storage pose a new problem: what is the right primitive for making such applications fault tolerant?

We propose two new primitives: Intents and locks with intent, which guarantee exactly-once semantics, mutual exclusion, and eventual progress

We propose new mechanisms: DAAL and an intent collector

We apply these primitives to build practical, fault-tolerant services

- Snapshots, live table re-partitioning, ACID transactions, …