Fault-tolerant and transactional stateful serverless workflows

Authors: Haoran Zhang, University of Pennsylvania; Adney Cardoza, Rutgers University–Camden; Peter Baile Chen, Sebastian Angel, and Vincent Liu, University of Pennsylvania

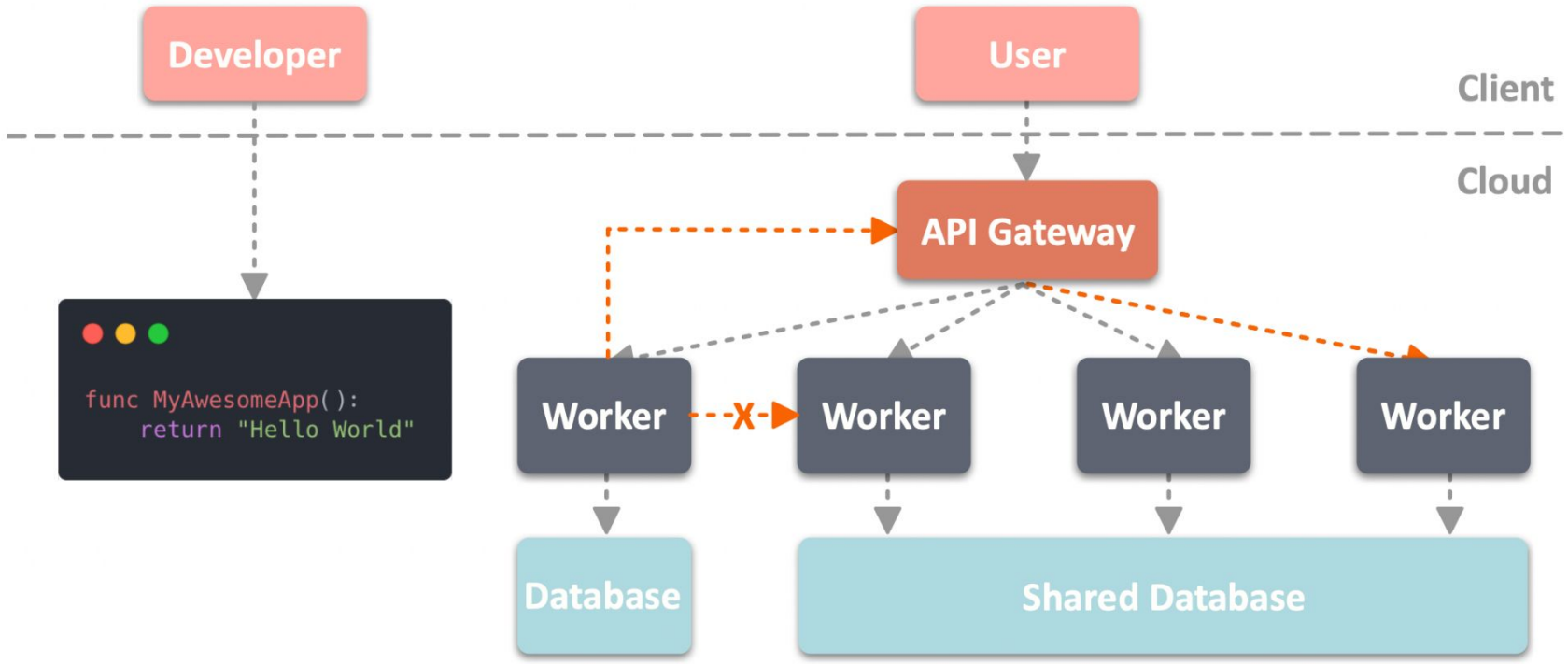
Presented By: Vaisaali Murali Krishnan

Date: April 19th, 2023

University at Buffalo The State University of New York

- What are workflows?
- What are serverless workflows?
- What does the paper cover?
- Why serverless computing? Examples of platforms.

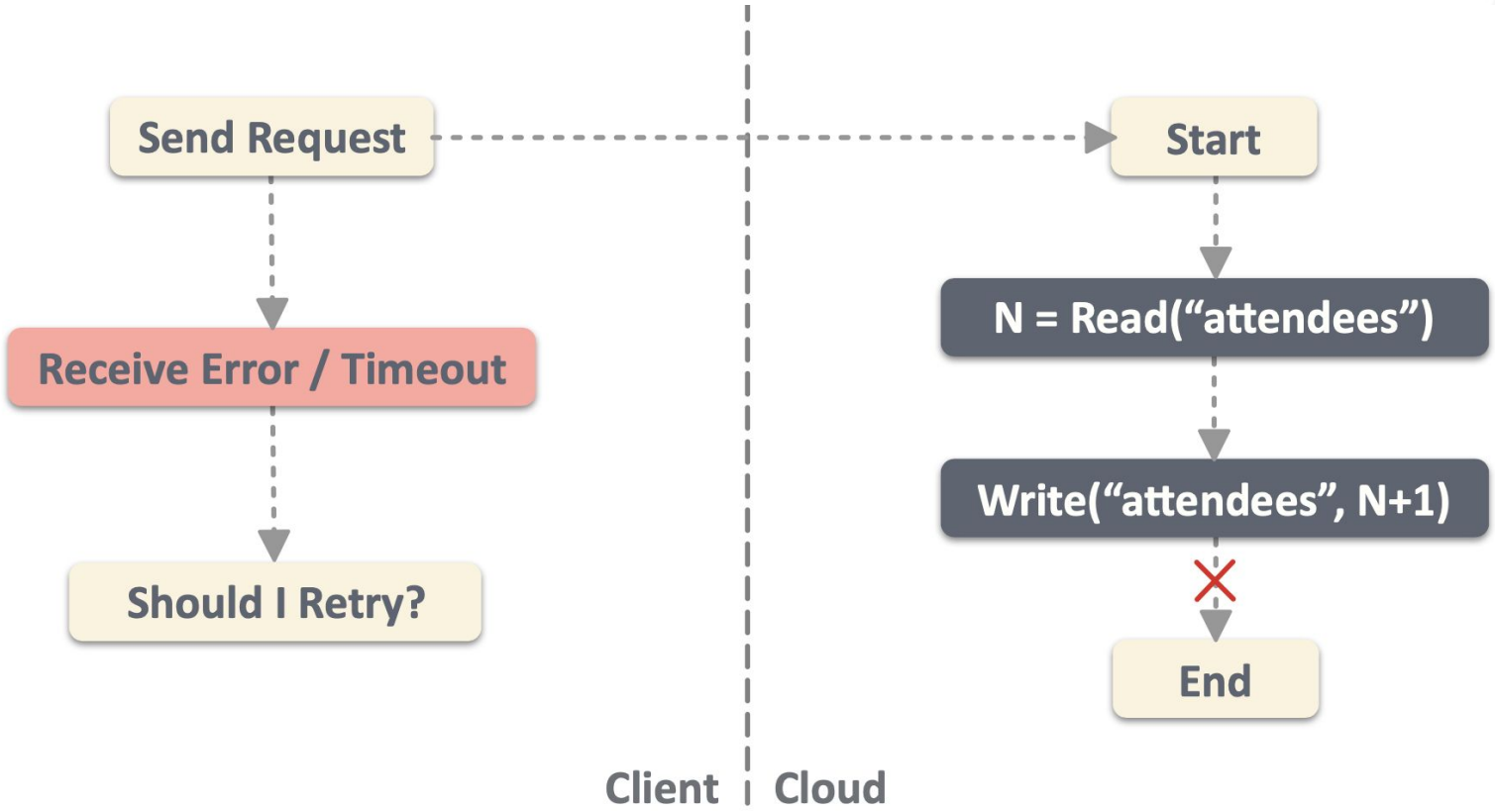
What is Serverless?



Basics of serverless computing (Function-as-a-Service)

- Eliminate the need to manage machines, runtimes, and resources.
- **Serverless functions** : Developers upload simple functions(Lambda) -> Clients invoke function -> cloud provider provisions the VMs or containers, deploying the user code, and scaling the allocated resources up and down based on current demand. These running functions are killed after a timeout which is configurable (helps in budgeting and limiting the effect of bugs).
- **workflows**: form directed graphs of functions. How to create ? -> AWS's step functions (takes care of all scheduling and data movement, and users get an identifier to invoke it) and driver functions(a single function specified by the developer that invokes other functions).
- Stateful serverless functions (**SSFs**): store state in fault-tolerant low-latency NoSQL databases. For example, AWS Lambdas can persist their state in DynamoDB, Google cloud functions can use Cloud Bigtable, and Azure functions can use Cosmos DB

How could serverless go wrong?



Drawback of existing approach in handling failures:

- If a function in a workflow crashes or its worker hangs, the provider will either (1) do nothing, leaving the workflow incomplete, or (2) restart the function on a different worker potentially corrupting database state and violating application semantics.
- How to solve? Write SSFs that are idempotent to ensure that re-execution is safe. Beldi simplifies this process so developers need only worry about their application logic and not the low-level details of how serverless providers respond to failures

Beldi

- A library and runtime system for writing and composing fault-tolerant and transactional stateful serverless functions.
- It is an extension of log-based fault-tolerant approach in Olive(OSDI 2016).

Beldi framework Goals:

- Exactly-once semantics
- SSF data sovereignty
- SSF reusability
- Workflow transactions
- Deployable today

Beldi Architecture

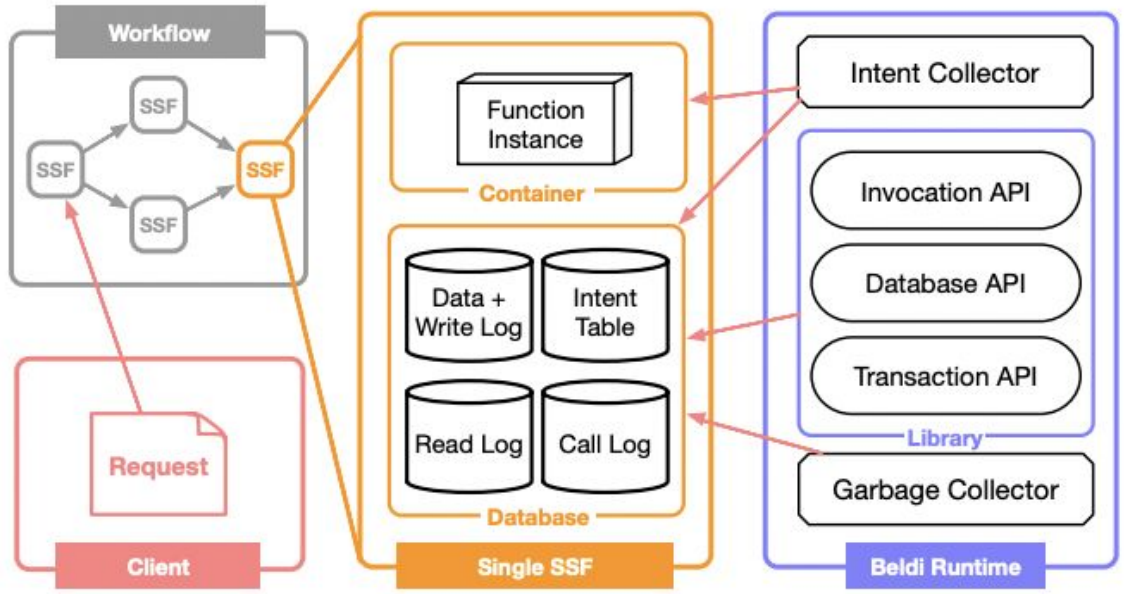


FIGURE 1—Beldi's architecture. Developers write SSFs as they do today, but use the Beldi API for transactions and externally visible operations. At runtime, operations for each SSF are logged to a database, which, when combined with a per-SSF intent and garbage collector, guarantees exactly-once semantics.

- Beldi Library
- Database tables - store the SSF's state, as well as logs of reads, writes, and invocations
- Intent Collector
- Garbage Collectors

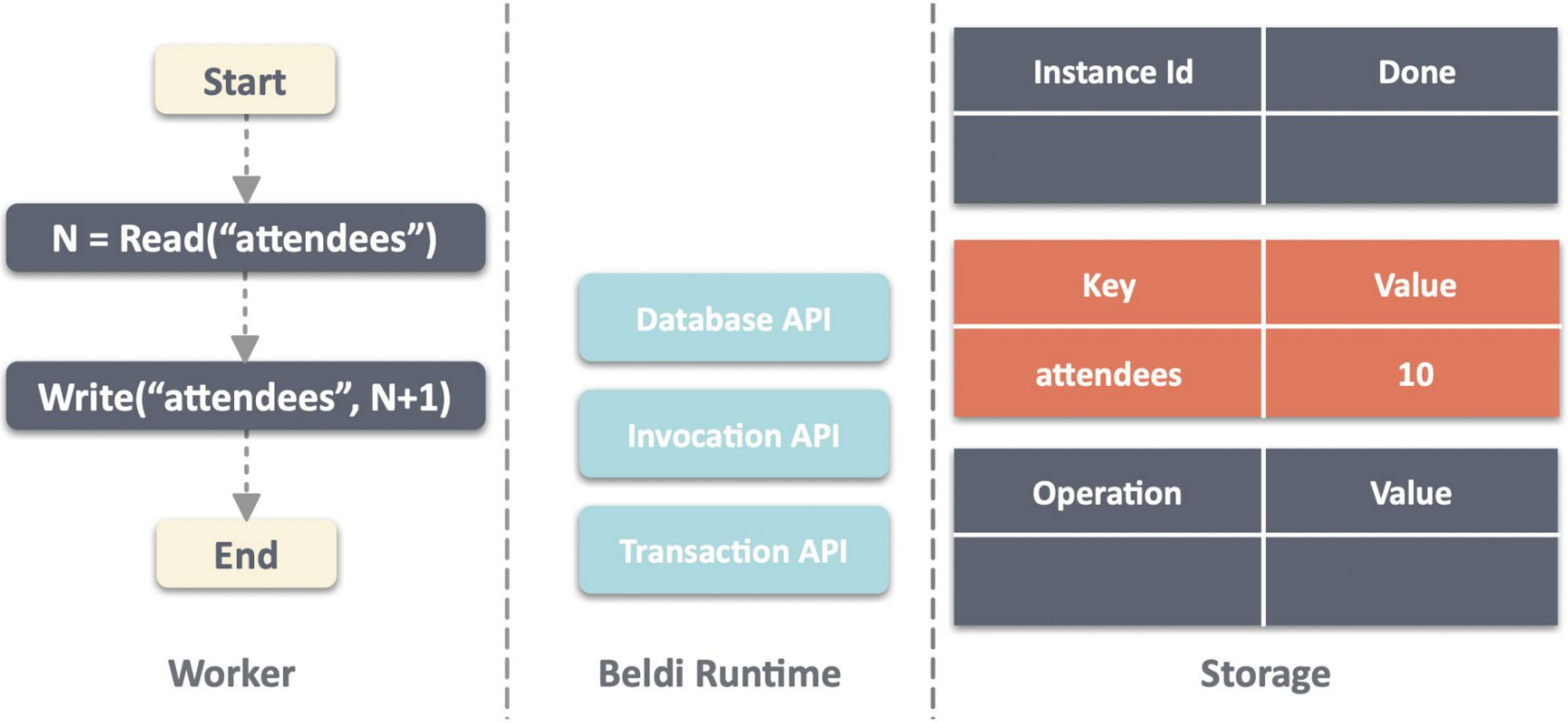
Beldi's runtime infrastructure

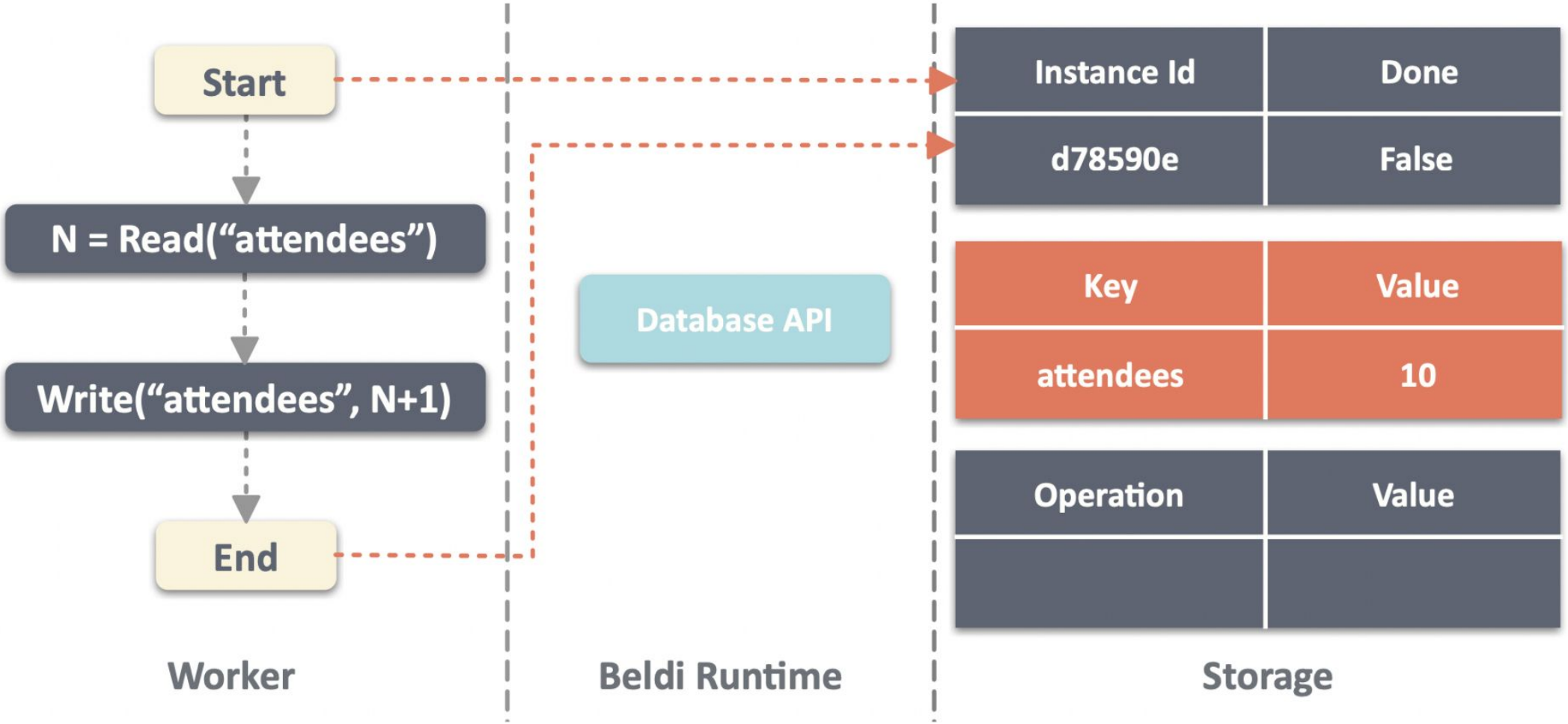
- Intent table - every SSF invocation associated with a distinct instance id even if the instances are of the same SSF and in the same workflow. Beldi ensures that the first operation in SSF is to check the intent table.
- Operation logs - Regular write - value is always true. Conditional write - whether condition has passed.
- Intent and Garbage Collectors - serverless functions that are triggered periodically by a timer.

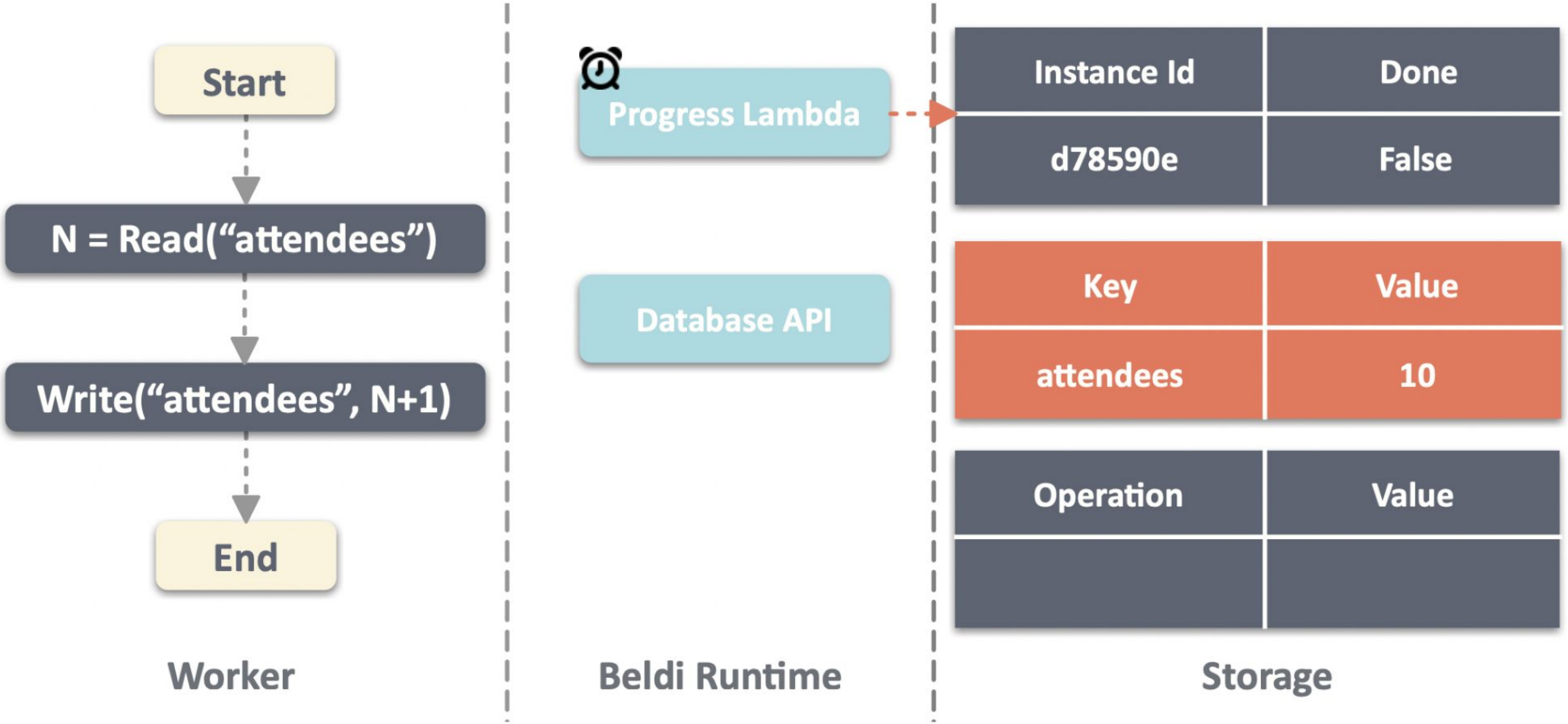
Log	Key	Value
`intent`	instance id	done, async, args, ret, ts
`read`	instance id, step number	value
`write`	instance id, step number	true / false
`invoke`	instance id, step number	instance id of callee, result

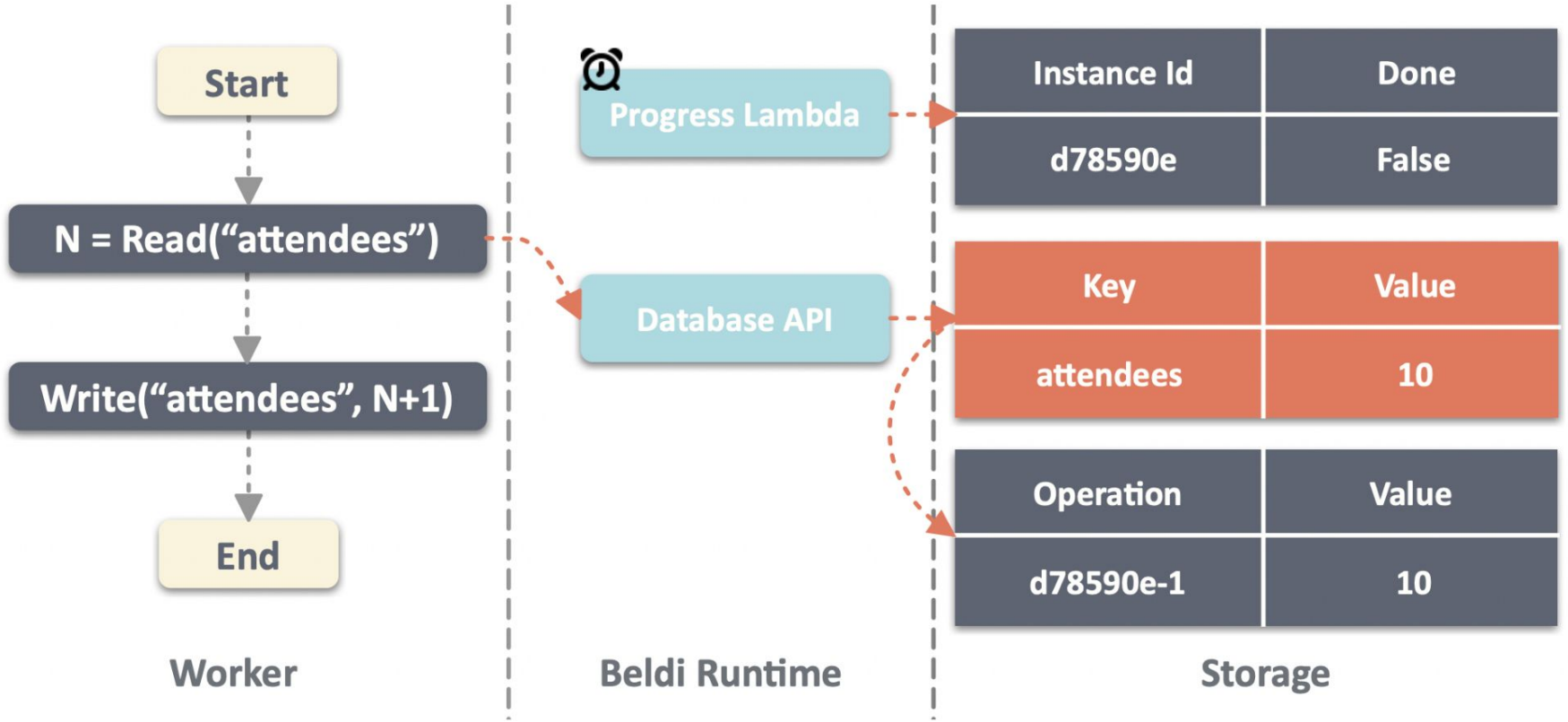
FIGURE 3—Beldi maintains four logs for each SSF. The intent table keeps track of an instance's completion status, arguments, return value, type of invocation, and timestamp assigned by its garbage collector (ts). The read log stores the value read. The write log stores true for writes, or the condition evaluation for a conditional write. The invoke log stores the instance id of the callee and its result.

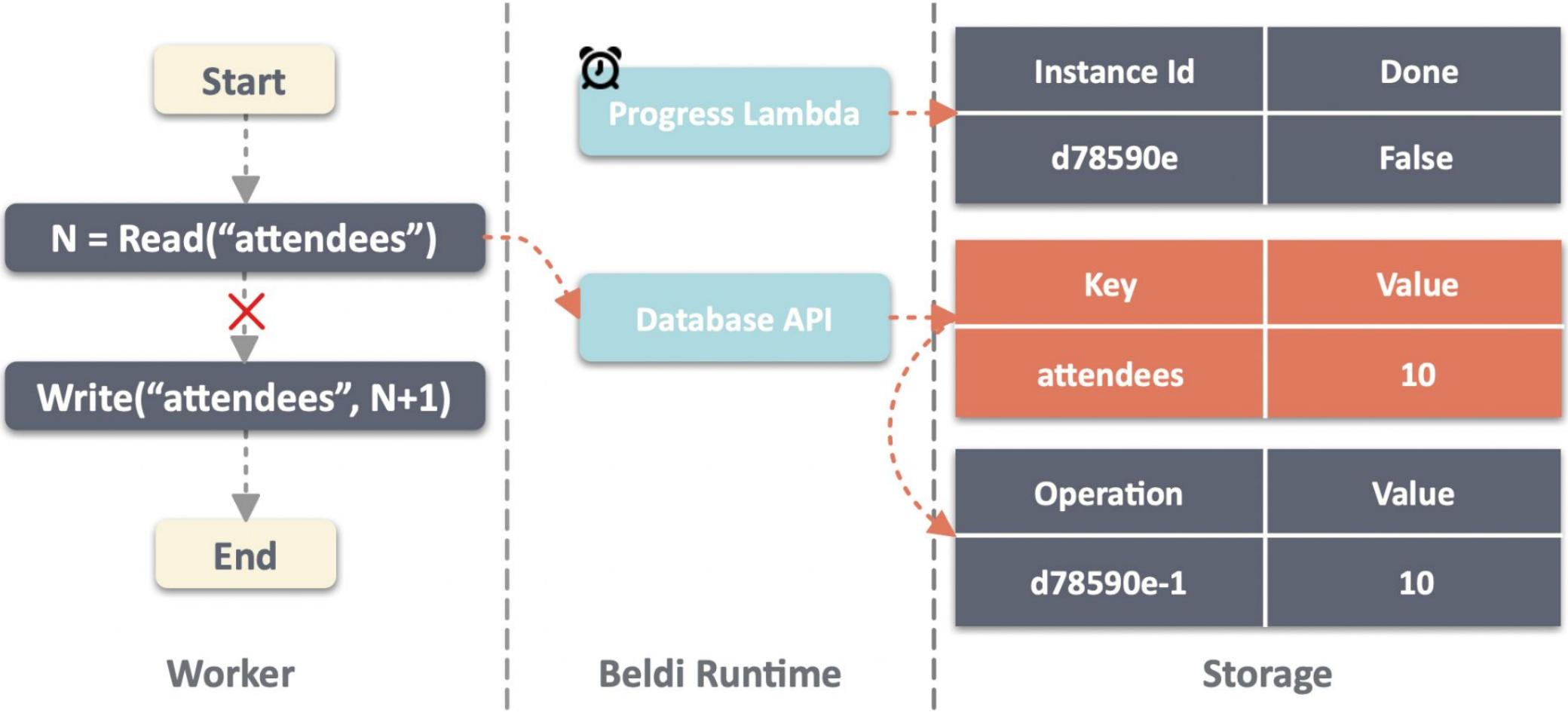
Example:

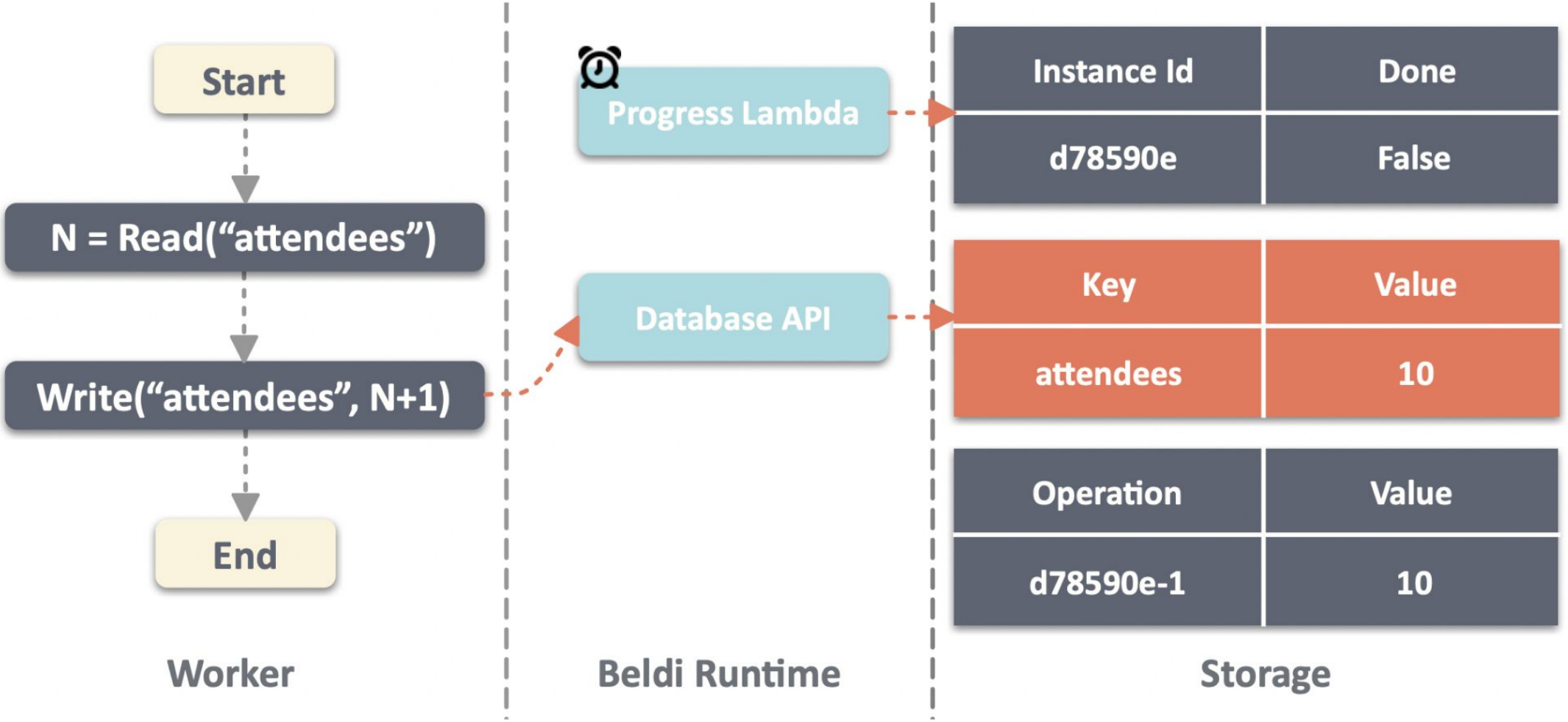

Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Database API
Invocation API
Transaction API
Beldi Runtime
Instance Id
Done
Key
Value
attendees
10
Operation
Value
Storage

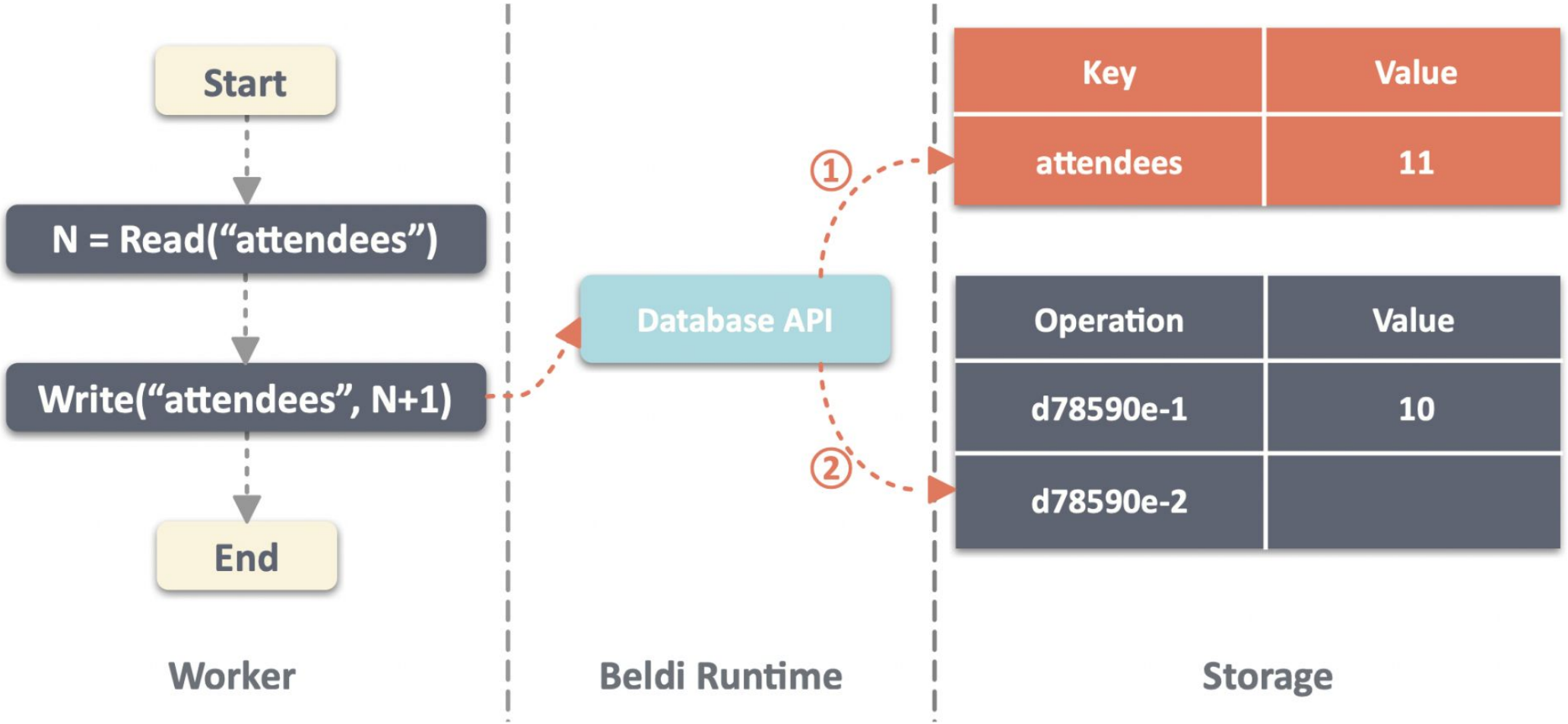

Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Database API
Beldi Runtime
Instance Id
Done
d78590e
False
Key
Value
attendees
10
Operation
Value
Storage

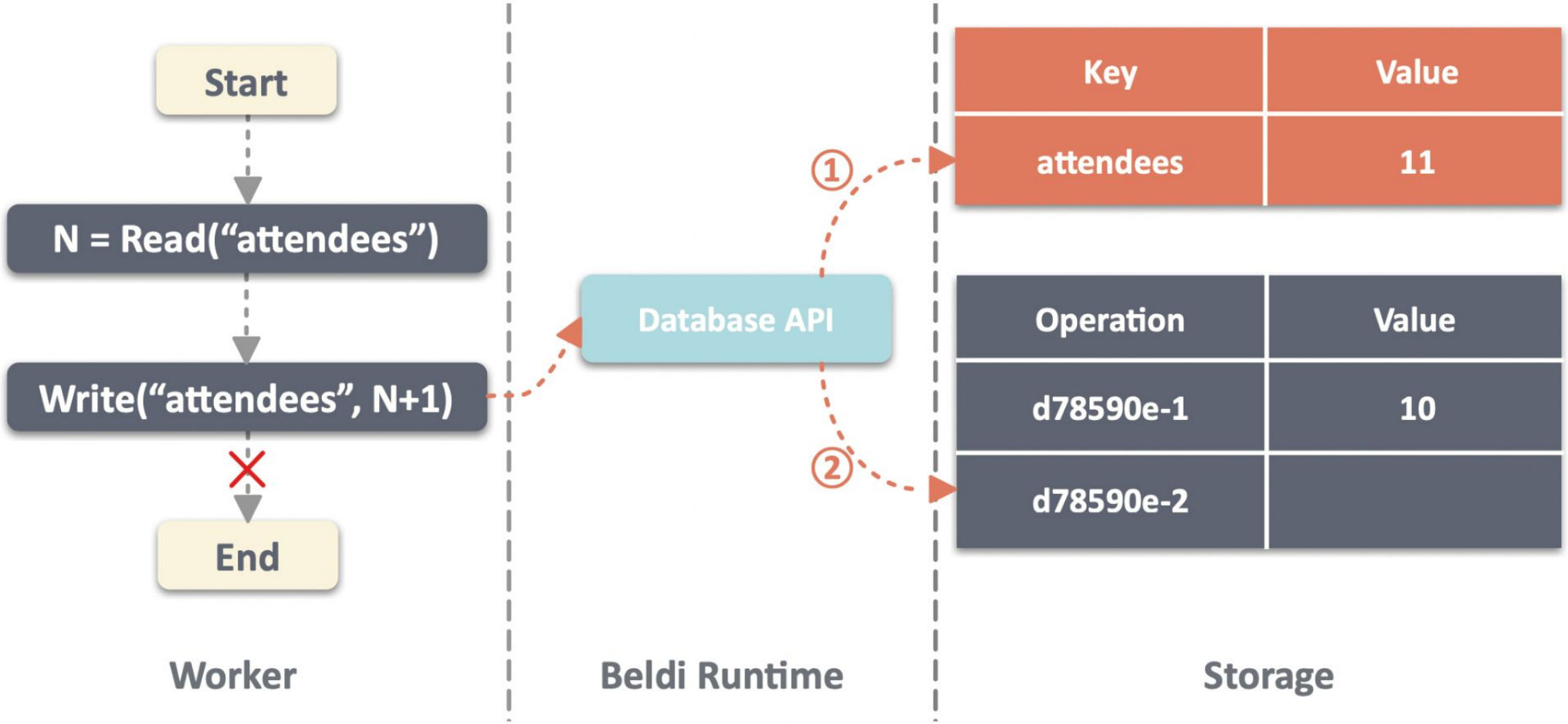

Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Progress Lambda
Database API
Beldi Runtime
Instance Id
Done
d78590e
False
Key
Value
attendees
10
Operation
Value
Storage

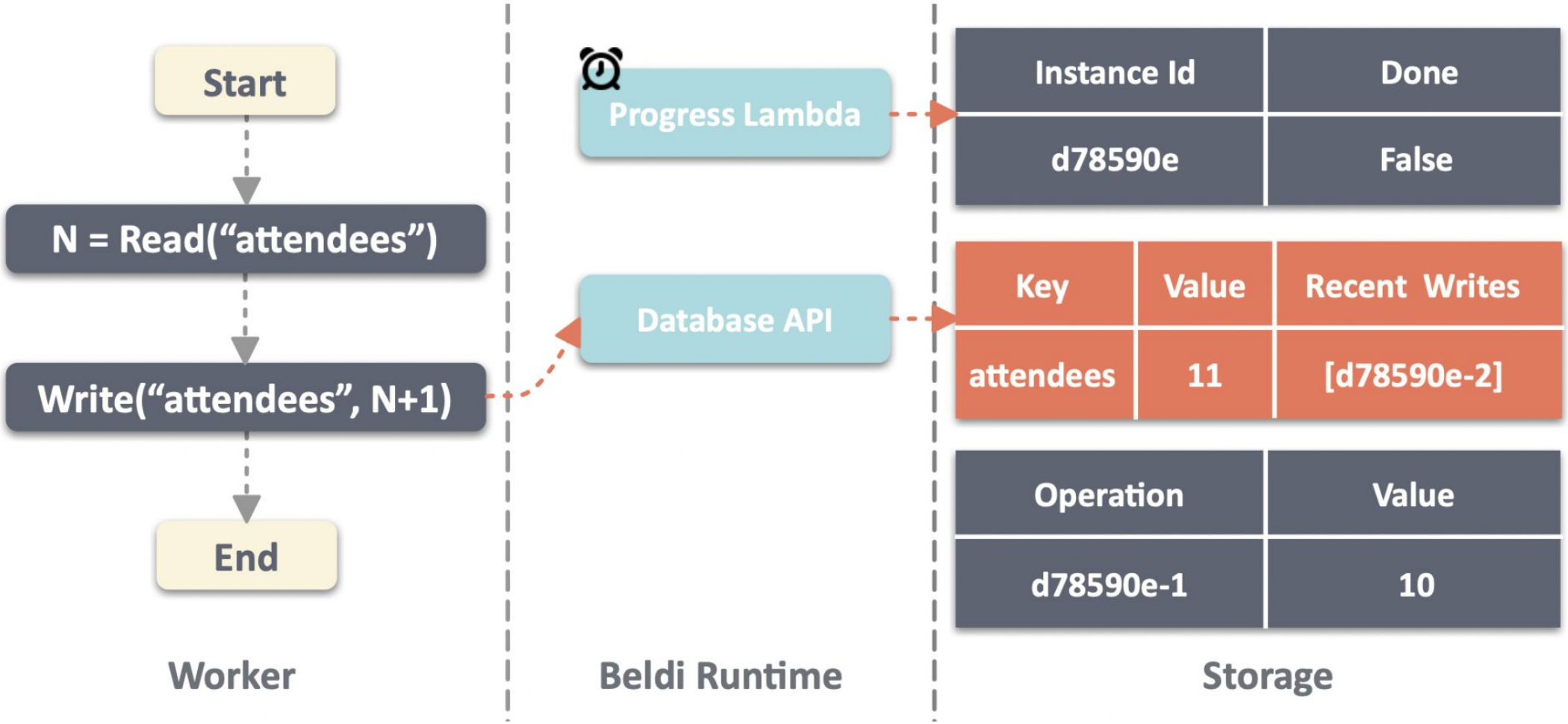

Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Progress Lambda
Database API
Beldi Runtime
Instance Id
Done
d78590e
False
Key
Value
attendees
10
Operation
Value
d78590e-1
10
Storage

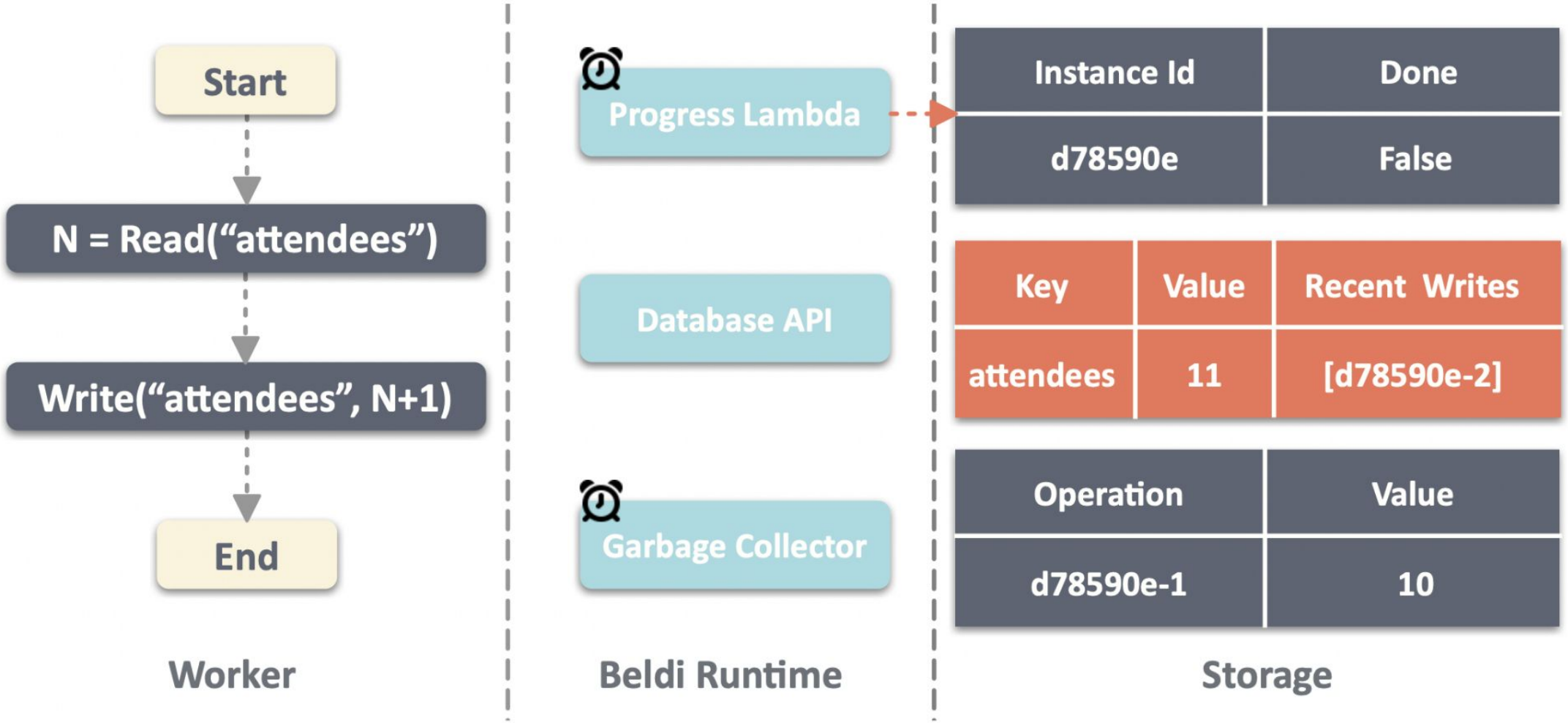

Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Progress Lambda
Database API
Beldi Runtime
Instance Id
Done
d78590e
False
Key
Value
attendees
10
Operation
Value
d78590e-1
10
Storage


Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Progress Lambda
Database API
Beldi Runtime
Instance Id
Done
d78590e
False
Key
Value
attendees
10
Operation
Value
d78590e-1
10
Storage


Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Database API
Beldi Runtime
1
2
Key
Value
attendees
11
Operation
Value
d78590e-1
10
d78590e-2
Storage


Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Database API
Beldi Runtime
①
②
Key
Value
attendees
11
Operation
Value
d78590e-1
10
d78590e-2
Storage


Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Progress Lambda
Database API
Beldi Runtime
Instance Id
Done
d78590e
False
Key
Value
Recent Writes
attendees
11
[d78590e-2]
Operation
Value
d78590e-1
10
Storage


Start
N = Read("attendees")
Write("attendees", N+1)
End
Worker
Progress Lambda
Database API
Garbage Collector
Beldi Runtime
Instance Id
Done
d78590e
False
Key
Value
Recent Writes
attendees
11
[d78590e-2]
Operation
Value
d78590e-1
10
Storage

Linked DAAL

DAAL introduced by Olive - requires an atomicity scope with high storage capacity, as otherwise few log entries can be added. Good fit for Cosmos DB (partition capacity is 20GB). But Dynamo and Big Table (400KB 256 MB). To support all common DBs:

Beldi introduces a new data structure called the linked DAAL that allows logs to exist on multiple rows (or atomicity scopes), with new rows being added as needed.

Why ?

1. linked DAALs continue to avoid the overheads of cross-table transactions and work on databases that do not support such transactions
2. allows multiple SSFs to access them concurrently
3. even with frequent accesses, our garbage collection protocol can ensure that the length of the list for each item is kept consistently small

Distributed Atomic Affinity Logging (DAAL) -

colocates log entries for an item in the same atomicity scope with the item's data

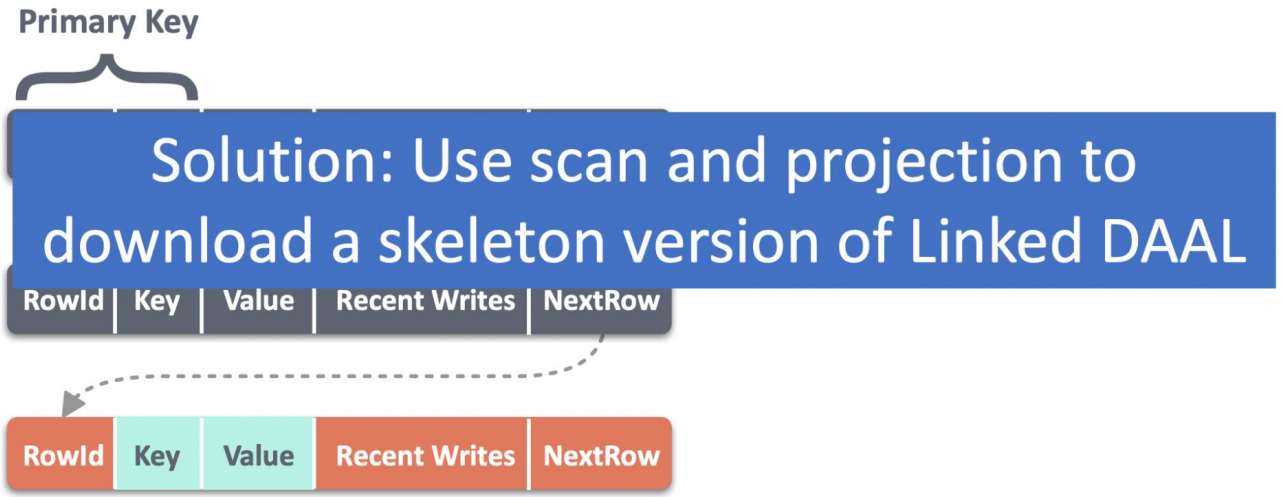
Key	Value	Recent Writes

Solution: spread the log for a given key across multiple rows

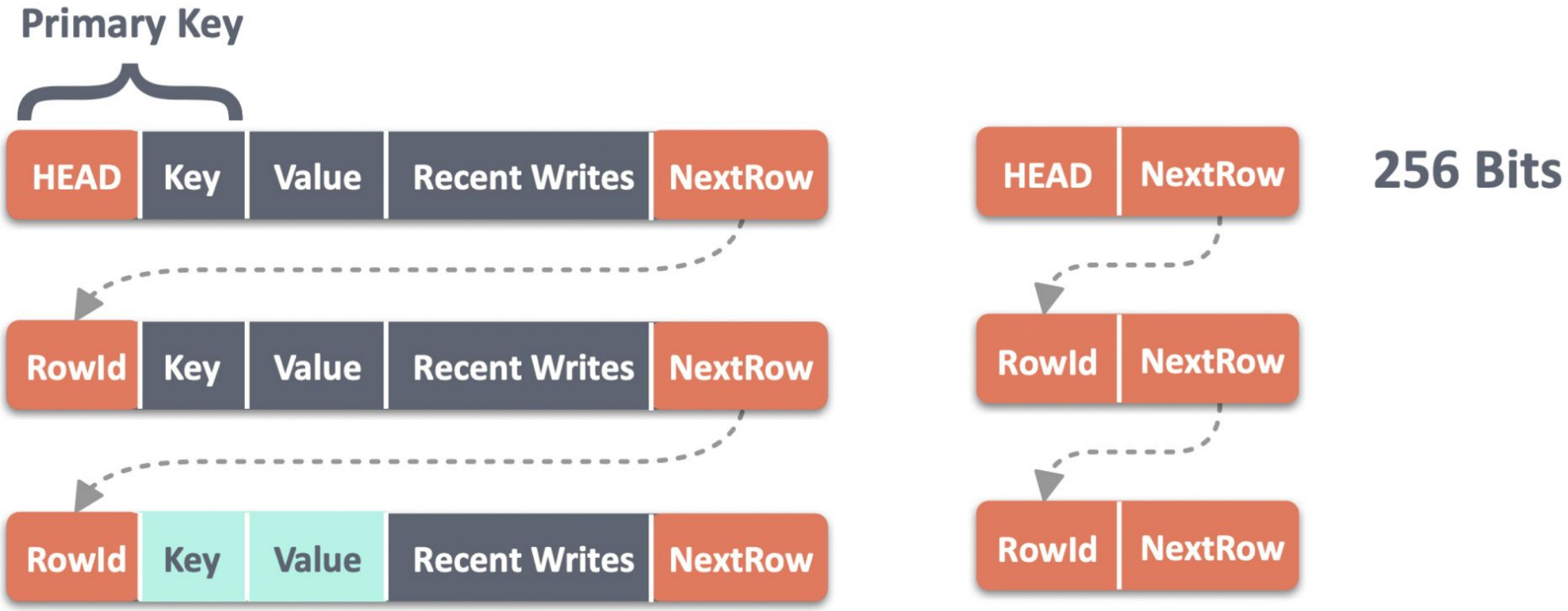
RowId	Key	Value	Recent Writes	NextRow
HEAD	attendees	10	[d78590e-1, d78590e-2, …, d78590e-1000]	f9cec2e

RowId	Key	Value	Recent Writes	NextRow
f9cec2e	attendees	11	[d78590e-1001]	

How do we traverse to the tail?



Solution: Use scan and projection to download a skeleton version of Linked DAAL


Primary Key
HEAD
Key
Value
Recent Writes
NextRow
RowId
Key
Value
Recent Writes
NextRow
RowId
Key
Value
Recent Writes
NextRow
HEAD
NextRow
RowId
NextRow
RowId
NextRow
256 Bits

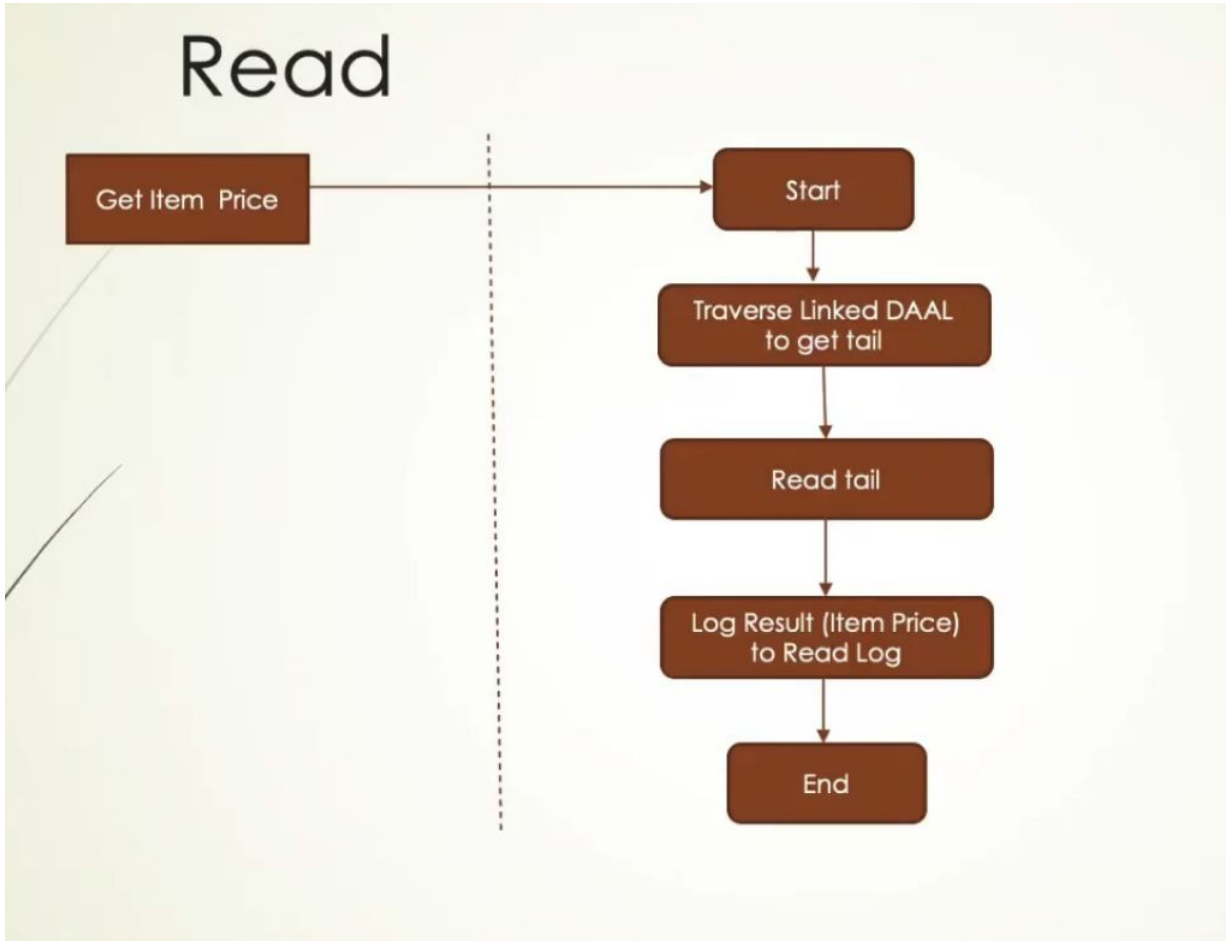
Beldi's API:

Beldi Library Function	Description
`read(k)` → v	Read operation
`write(k, v)`	Write operation
`condWrite(k, v, c)` → T/F	Write if c is true
`syncInvoke(s, params)` → v	Calls s and waits for answer
`asyncInvoke(s, params)`	Calls s without waiting
`lock()`	Acquire a lock
`unlock()`	Release a lock
`begin_tx()`	Begin a transaction
`end_tx()`	End a transaction

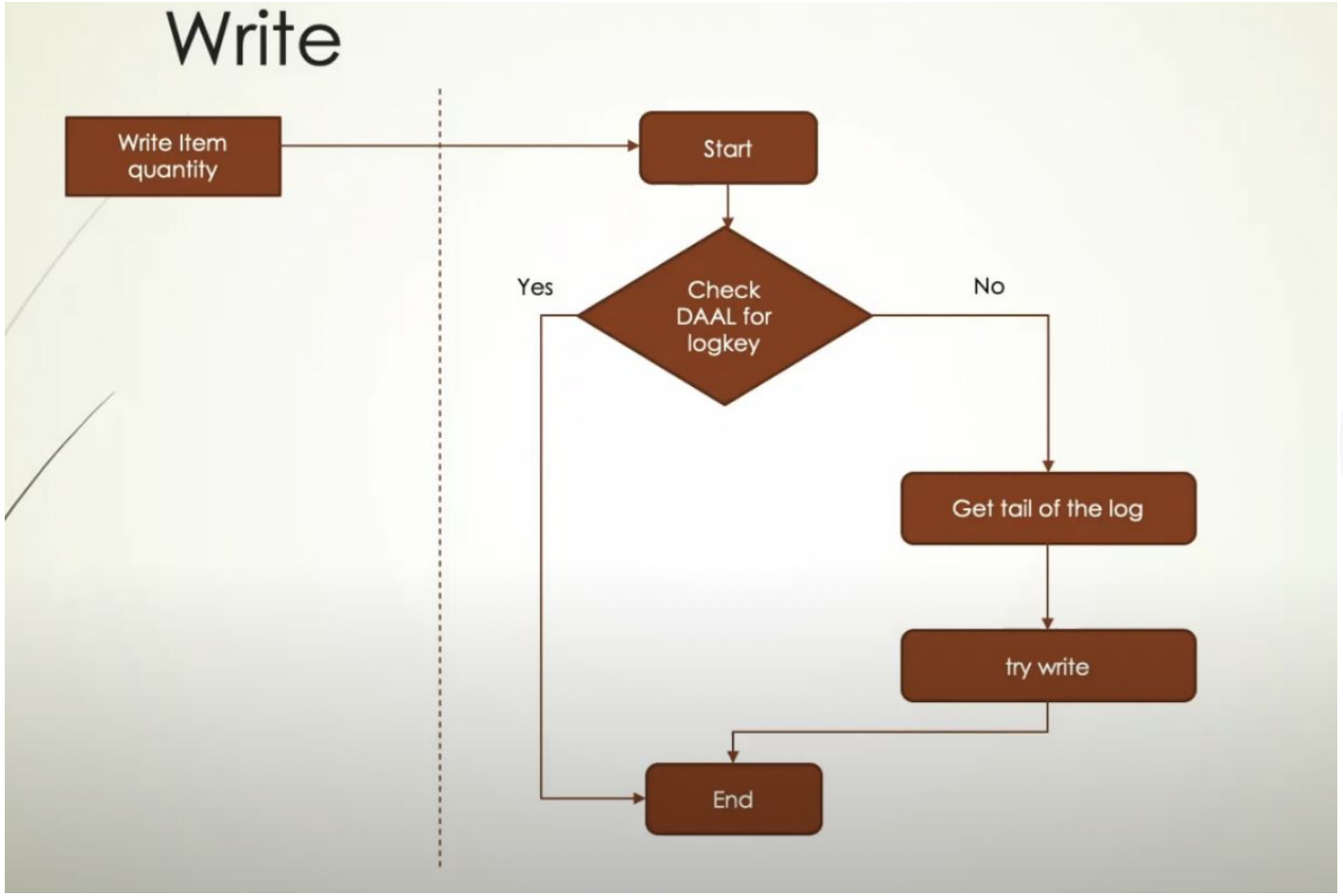
FIGURE 2—Beldi's API for SSFs, which includes all of Beldi's primitives and its transactional support (§6).

- Beldi supports the ability to begin and end transactions. Operations between these calls enjoy ACID semantics.
- Beldi's API hides from developers all of the complexity of logging, replaying, and concurrency control protocols
- An SSF using Beldi's API automatically determines the SSF's instance id, step number, and whether it is part of a transaction.
- Beldi takes actions before and after the main body of the SSF as well as around any Beldi API operations.

Read Operation

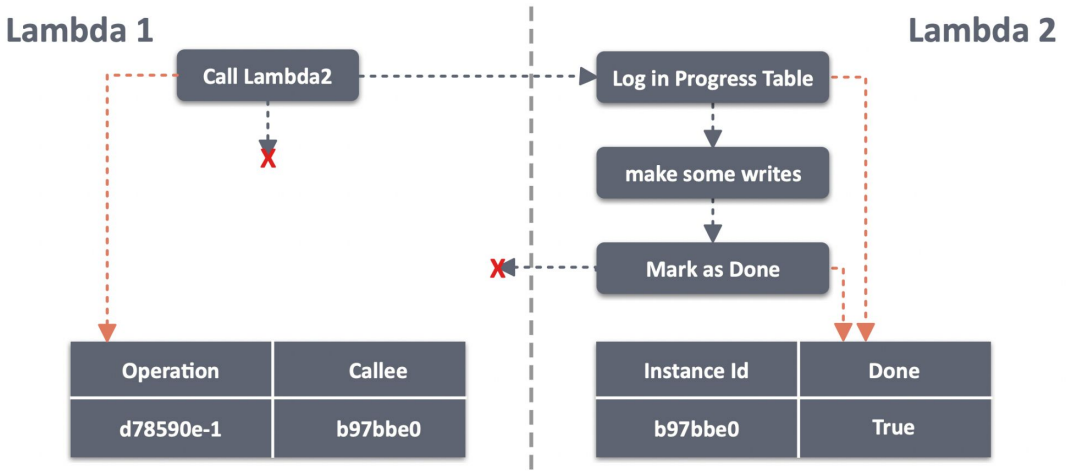


Write Operation

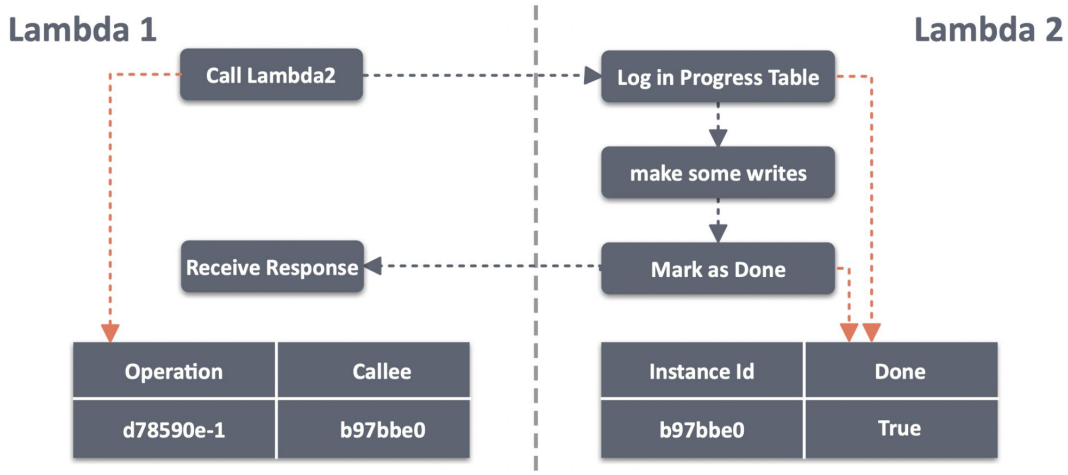


Invocation Operation

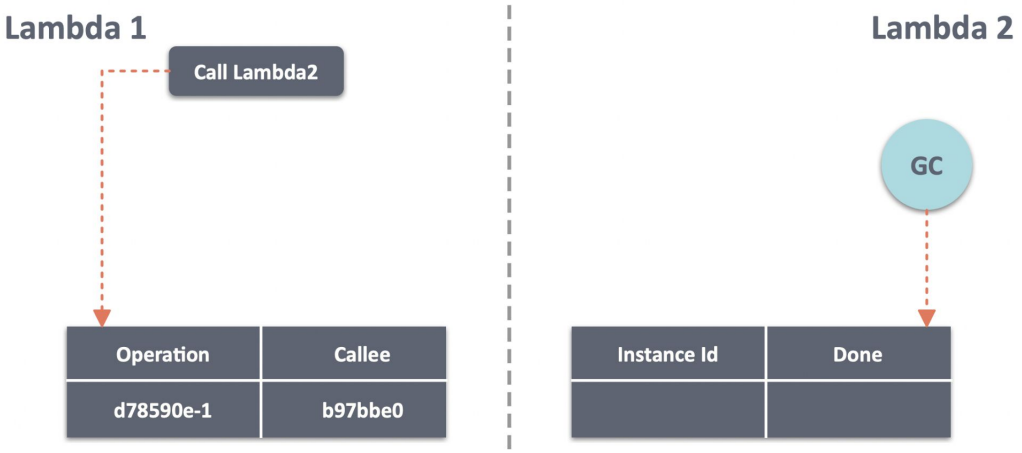
Invocation with exactly-once semantics



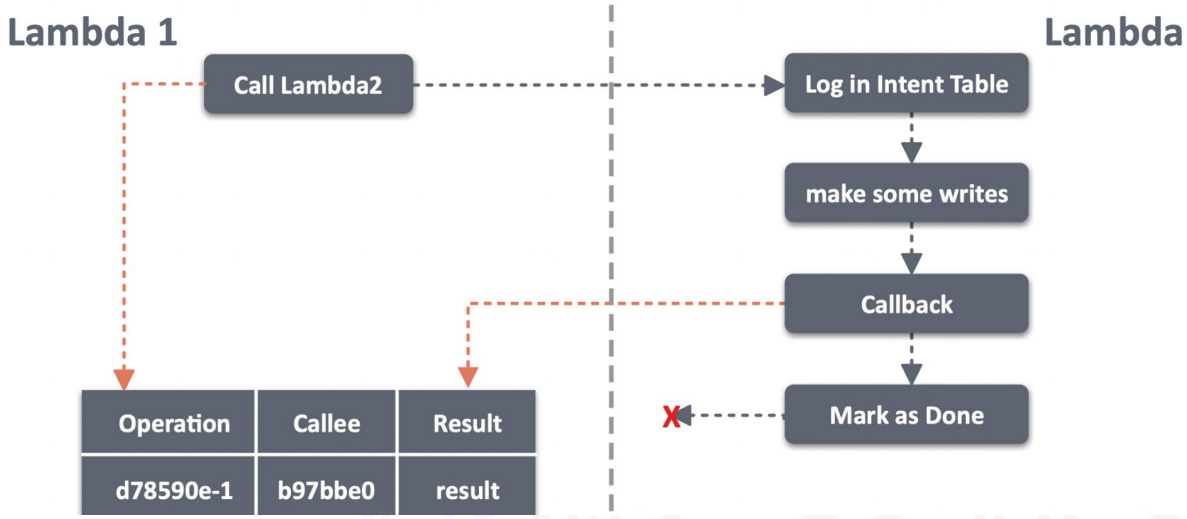
Invocation with exactly-once semantics



Invocation with exactly-once semantics



Invocation with exactly-once semantics



Beldi vs. Olive

- Beldi is inspired by Olive's high-level approach but makes key changes and introduces new data structures and tables, support for invocations so that SSFs can call each other (Olive only supports storage operations), and garbage collection mechanisms to keep overheads low.

Garbage Collection:

- Why? - Beldi's use of scans means that the linked DAAL's length is generally not the performance bottleneck. But unbounded growth of the linked DAAL and logs (intent table, read log, invoke log) can lead to significant overheads and storage costs. (GC) deletes old rows and log entries without blocking SSFs that are concurrently accessing the list.
- How? - GC is an SSF triggered by a timer. Garbage is collected in 6 parts:

1. Finds intents that have finished since the last time it ran, and assigns them the current time as a finish timestamp.
2. looks up all intents whose finish timestamp is 'old enough' (we expand on this next), and marks them as 'recyclable.'
3. Removes log entries that belong to recyclable intents.
4. For every item, disconnects the non-tail rows of their linked DAAL that have empty logs, marks these rows as 'dangling', and assigns them the current time as a dangling timestamp.
5. Removes all rows whose dangling timestamp is 'old enough.'
6. Removes the log entries from the intent table.

Implementation:

- Implemented a prototype of Beldi for Go applications that runs transparently on AWS Lambda and DynamoDB.
- Implement three case studies: a social media site, a travel reservation system, and a media streaming and review service

Experimental setup

- Experiments were run on AWS Lambda.
- Configured lambdas to use 1 GB of memory and set DynamoDB to use autoscaling in on-demand mode.
- Turn off automatic Lambda restarts and let Beldi's intent collectors take care of restarting failed Lambdas.
- Baseline - Running these applications on AWS Lambda without Beldi's library and runtime (will have no exactly once semantics or support transactions).

What are the costs of Beldi's primitives operations?

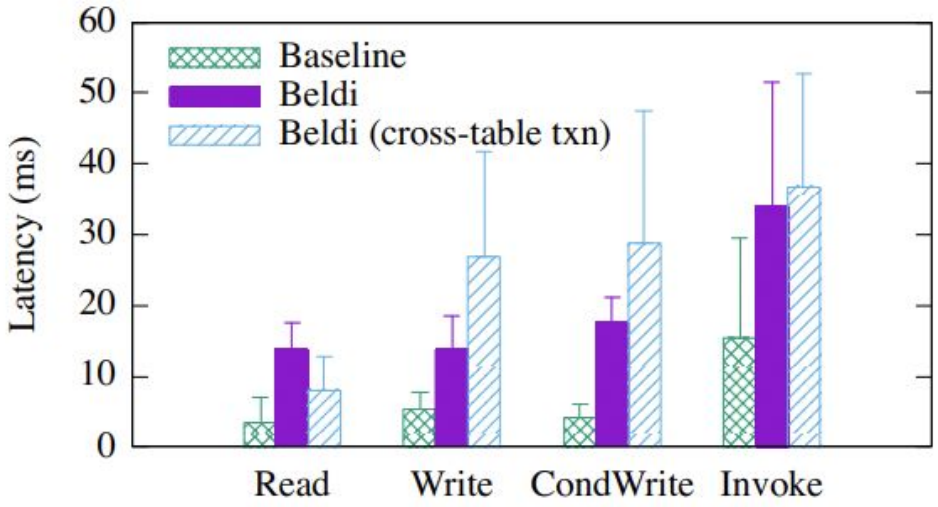


FIGURE 13—Median latency of Beldi's operations. Error bar represents the 99th percentile, and "cross-table tx" is an implementation of Beldi that uses cross-table transactions instead of the linked DAAL.

- There is overhead of Beldi's reads/writes compared to those of the baseline stem. Reason: scanning the linked DAAL (instead of reading a single row) and logging.
- .For invoke, the overheads come from the callback mechanism and logging to the invoke log.
- All of Beldi's operations are around 2–4× more expensive than the baseline.
- Cross-table transactions does not use a DAAL so reads avoid the scan (but not the logging), and writes perform an atomic transaction where the value is written to one table and the log entry is added to another. When SSF use transactions, because of read lock using condWrite, Beldi is cheaper for read Operation.

How does Beldi perform on our applications?

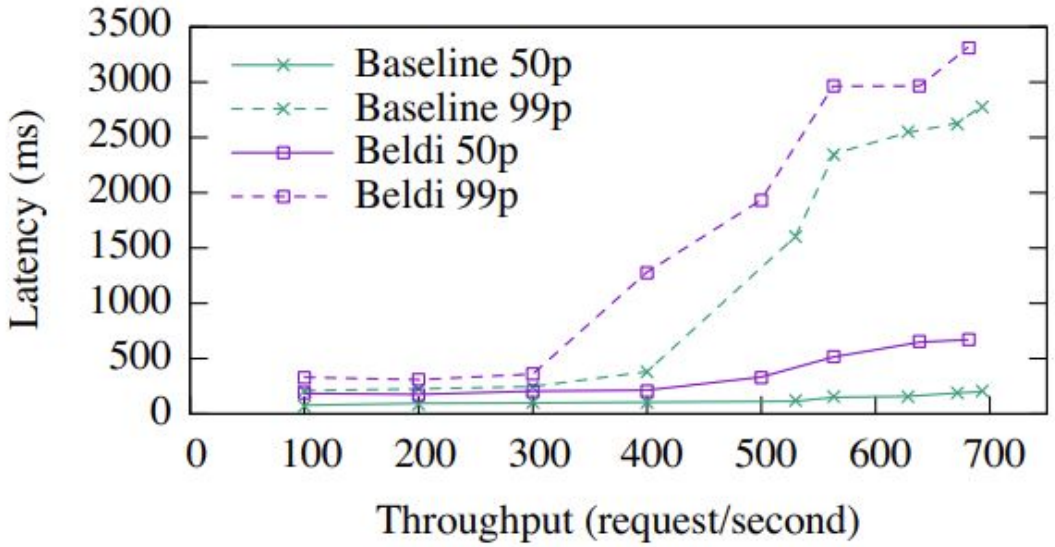


FIGURE 14—Median response time and throughput for our movie review service. Dashed lines represent 99th-percentile response time.

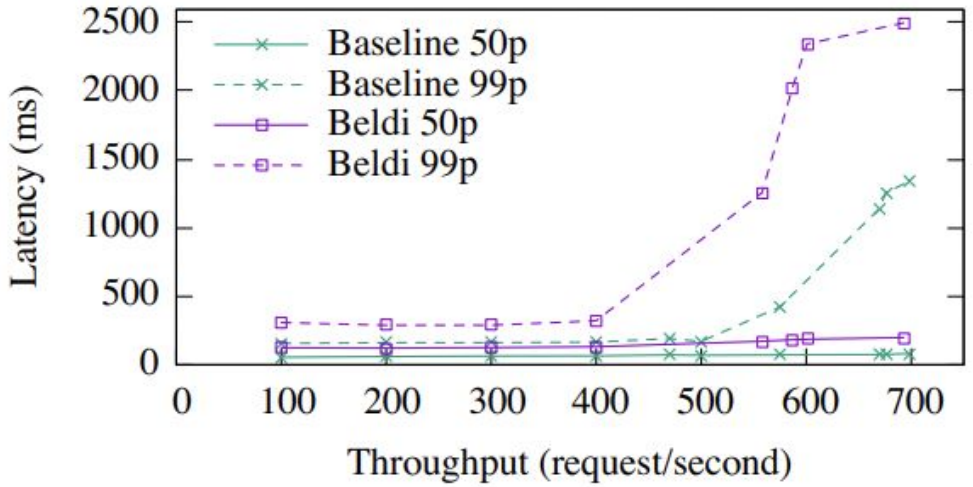


FIGURE 15—Median response time and throughput for travel reservation service. Dashed lines represent 99th-percentile response time. Beldi performs transactions over multiple SSFs to reserve a hotel room and a flight, while the baseline returns inconsistent results.

What is the effect of garbage collection?

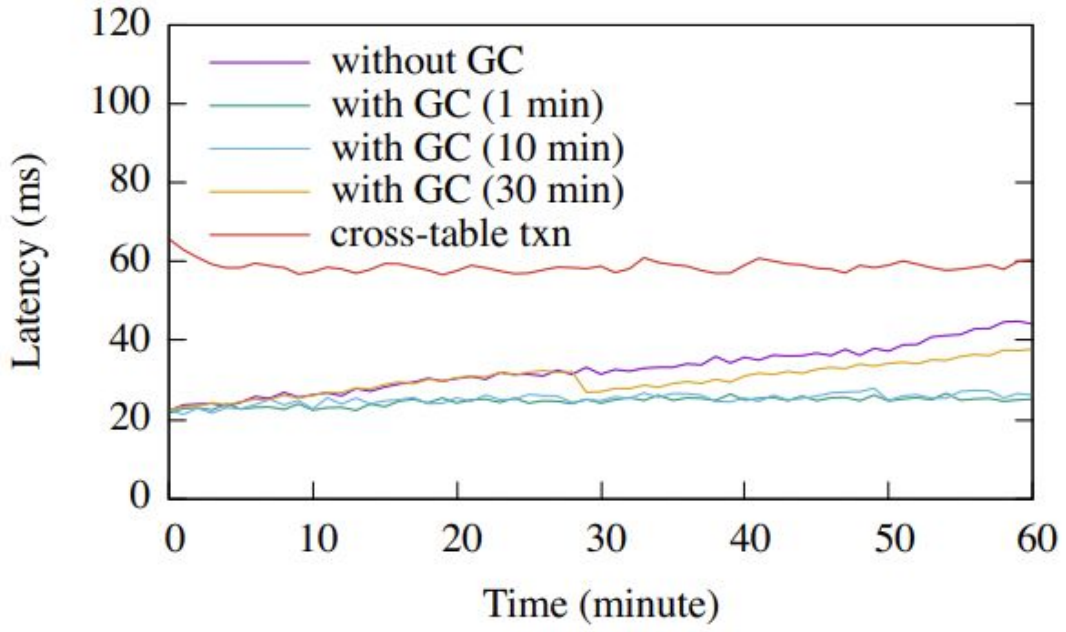


FIGURE 16—Median response time for an SSF that uses one `write` operation under different GC configurations. Without GC, the linked DAAL grows over time. As a baseline, we configure Beldi with cross-table transactions that do not use a linked DAAL.

- Here, we are evaluating the importance of garbage collector timeout on performance.
- The median response times for SSFs that access the linked DAAL are only lightly impacted by the choice of T.
- While T has a minor impact on performance, it does impact storage overhead and I/O, since read and write operations still fetch a projection of the linked DAAL which scales with the number of rows

Related Work

- Beldi is an extension Olive's elegant approach to fault tolerance and mutual exclusion by introducing new data structures, algorithms, and abstractions.
- Cloudburst proposes a new architecture for incorporating state into serverless functions. gg proposes workarounds to state-management issues that arise in desktop workloads that are outsourced to thousands of serverless functions. However, the general approach to fault tolerance in these works is to re-execute the entire workflow when there is a crash or timeout—violating exactly-once semantics if any SSF in the workflow is not idempotent.
- AFT is the closest proposal to Beldi and introduces a fault-tolerant shim layer for SSFs. However, AFT's deployment setting, guarantees, and mechanisms are very different.

Conclusion:

- Beldi makes it possible for developers to build transactional and fault-tolerant workflows of SSFs on existing serverless platforms.To do so, Beldi introduces novel refinements to an existing log-based approach to fault tolerance, including a new data structure and algorithms that operate on this data structure, support for invocations of other SSFs with a novel callback mechanism, and a collaborative distributed transaction protocol. With these refinements, Beldi extracts the fault tolerance already available in today's NoSQL databases, and extends it to workflows of SSFs at low cost with minimal effort from application developers.

My thoughts:

- Having a linked-DAAL, intent collector and a garbage collector are good ideas. However, data structure linked-DAAL with the use of Scan and projector is a little too complex. It can be simplified with a better single data structure without the need of additional mechanisms.
- The evaluation graphs are not very justifying to prove that this theory is beneficial. Only latency is being evaluated. Many other parameters such as the number of lines of code with and without Beldi library, impact of data corruption, etc could be used to make their claim strong.

References:

https://www.usenix.org/system/files/osdi20-zhang_haoran.pdf

https://www.usenix.org/sites/default/files/conference/protected-files/osdi20_slides_zhang-haoran.pdf

https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

https://www.youtube.com/watch?v=Qh8o5Q9UdQo