Pravega

Storage Reimagined for a Streaming World

Srikanth Satya & Tom Kaitchuck

Dell EMC Unstructured Storage
srikanth.satya@emc.com
tom.kaitchuck@emc.com

Streaming is Disruptive

How do you **shrink to zero** the time it takes to turn

Stateful processors born for streaming, like **Apache Flink**, are **disrupting** how we think about **data computing** …

We think the world needs a complementary technology … to similarly **disrupt storage**.

- Ability to deliver **accurate results** processing continuously even with late arriving or out of order data

Introducing Pravega Streams

A new storage abstraction – a **stream** – for continuous and infinite data

- Named, durable, append-only, **infinite** sequence of bytes
- With low-latency appends to and reads from the tail of the sequence
- With high-throughput reads for older portions of the sequence

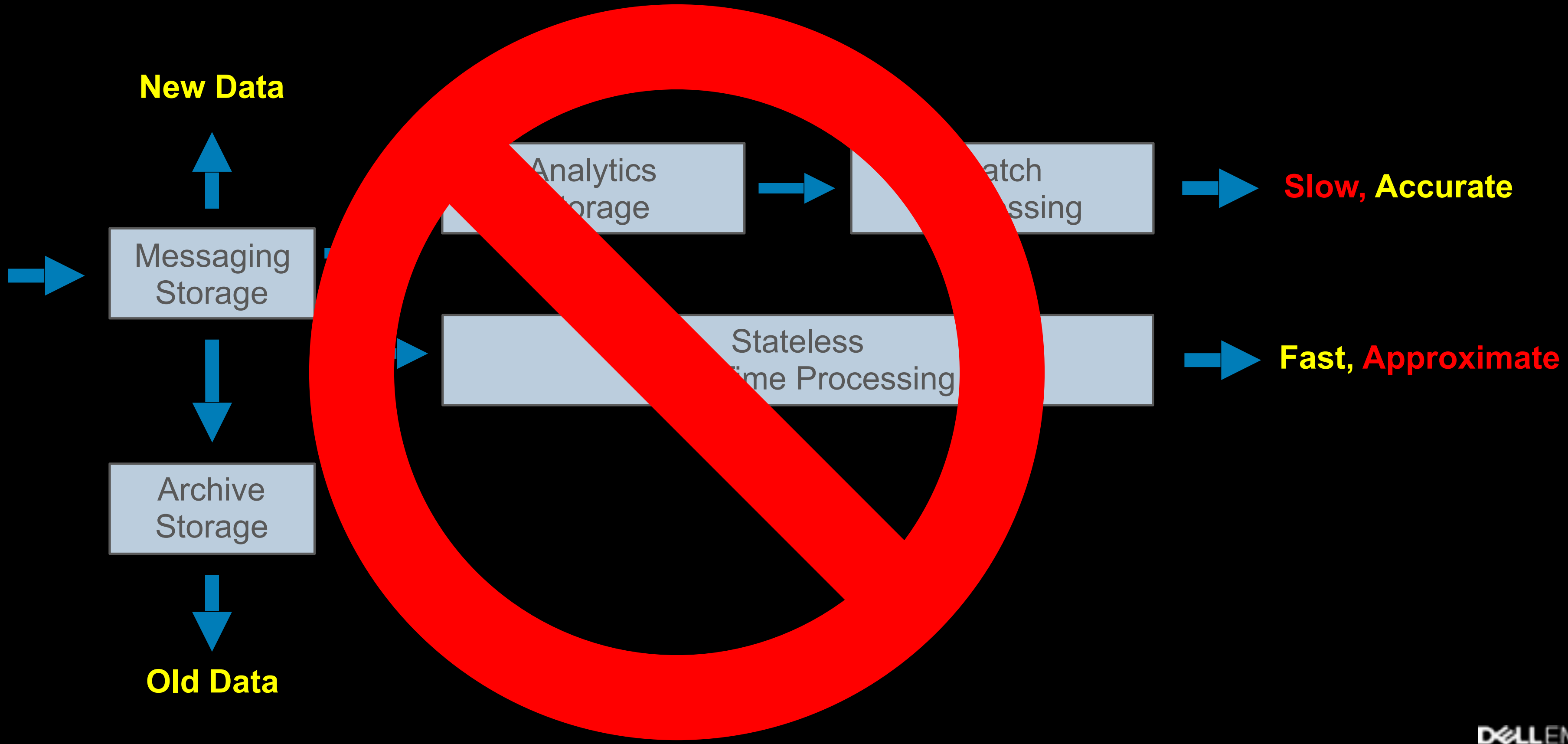
Coordinated scaling of stream storage and stream processing

- Stream writes partitioned by app key
- Stream reads independently and automatically partitioned by arrival rate SLO
- Scaling protocol to allow stream processors to scale in lockstep with storage

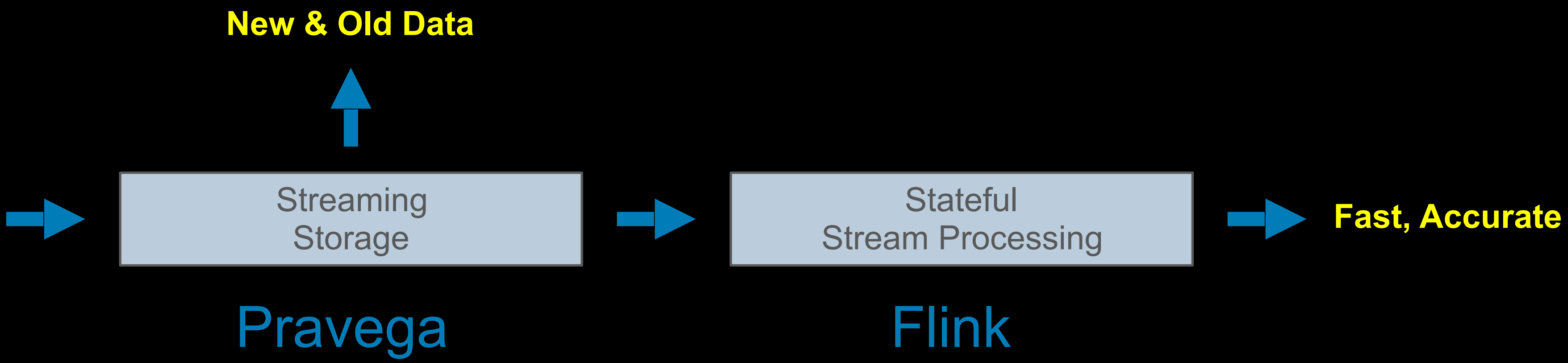
Enabling **system-wide exactly once** processing across multiple apps

- Streams are ordered and strongly consistent
- Chain independent streaming apps via streams
- Stream transactions integrate with checkpoint schemes such as the one used in Flink

In Place of All This …

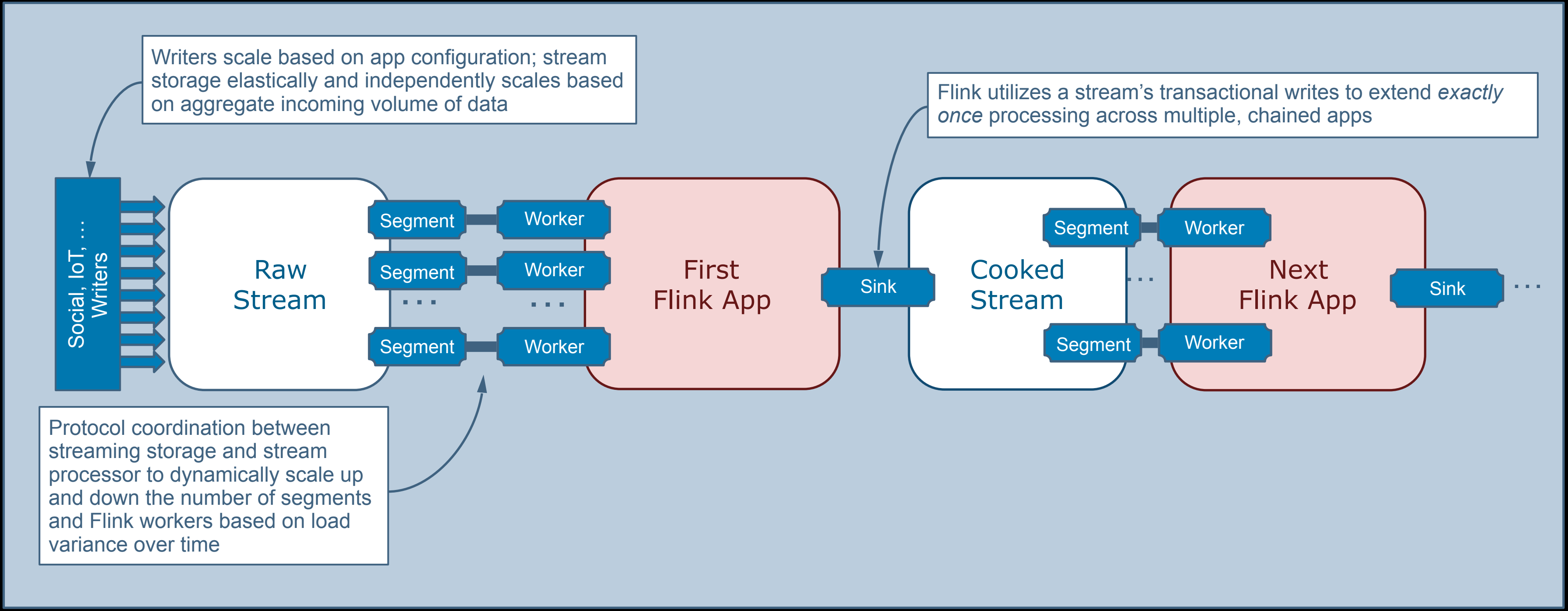


… Just Do This!



Each component in the combined system – writers, streams, readers, apps – is independently, elastically, and dynamically scalable in coordination with data volume arrival rate over time. Sweet!

Pravega Streams + Flink



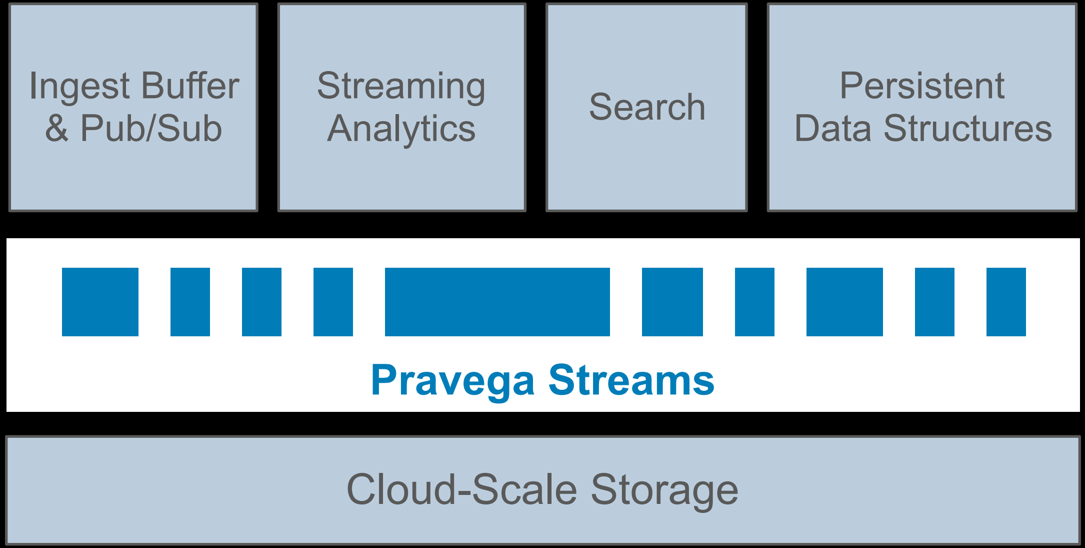
Writers scale based on app configuration; stream storage elastically and independently scales based on aggregate incoming volume of data

Flink utilizes a stream’s transactional writes to extend *exactly once* processing across multiple, chained apps

Protocol coordination between streaming storage and stream processor to dynamically scale up and down the number of segments and Flink workers based on load variance over time

And It’s Just the Beginning …

Enabling a new generation of distributed middleware reimagined as streaming infrastructure



Ingest Buffer & Pub/Sub

Streaming Analytics

Search

Persistent Data Structures

Pravega Streams

Cloud-Scale Storage

How Pravega Works

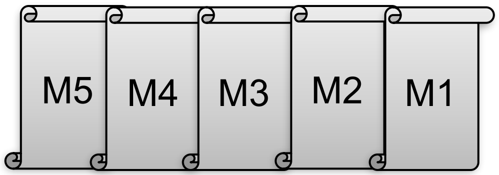
Architecture & System Design

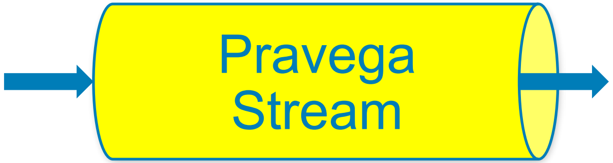
Pravega Architecture Goals

- All data is durable
  - Data is replicated and persisted to disk before being acknowledged
- Strict ordering guarantees and exactly once semantics
  - Across both tail and catch-up reads
  - Client tracks read offset, Producers use transactions
- Lightweight, elastic, infinite, high performance
  - Support tens of millions of streams
  - Low (<10ms) latency writes; throughput bounded by network bandwidth
  - Read pattern (e.g. many catch-up reads) doesn’t affect write performance
- Dynamic partitioning of streams based on load and throughput SLO
- Capacity is not bounded by the size of a single node

Streaming model

- Fundamental data structure is an ordered sequence of bytes
- Think of it as a durable socket or Unix pipe
- Bytes are not interpreted server side
- This implicitly guarantees order and non-duplication
- Higher layers impose further structure, e.g. message boundaries



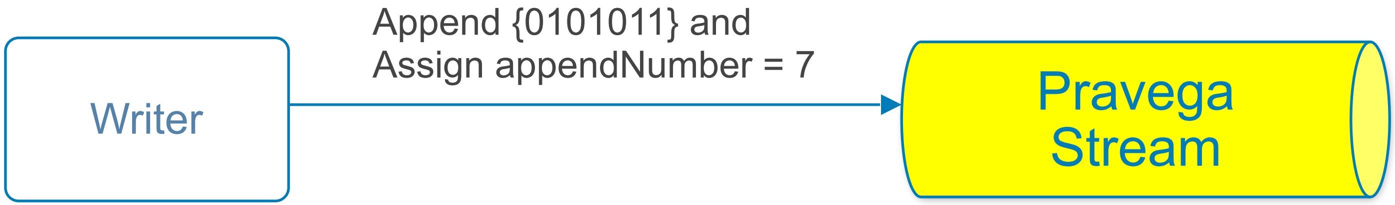


Cartoon API

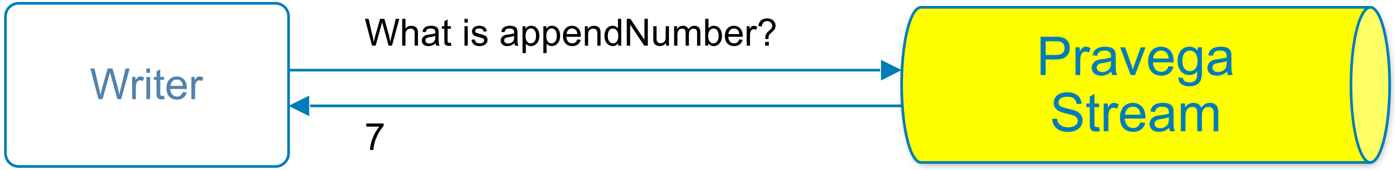
```
public interface SegmentWriter {

    /** Asynchronously and atomically write data
    */

    void write(ByteBuffer data);

    /** Asynchronously and atomically write the
    data if it can be written at the provided offset
    */

    void write(ByteBuffer data, long atOffset);

    /** Asynchronously and atomically write all of
    the data from the provided input stream */

    void write(InputStream in);

}
```

```
public interface SegmentReader {

    long fetchCurrentLength();

    /** Returns the current offset */

    long getOffset();

    /** Sets the next offset to read from */

    void setOffset(long offset);

    /** Read bytes from the current offset */

    ByteBuffer read(int length);

}
```

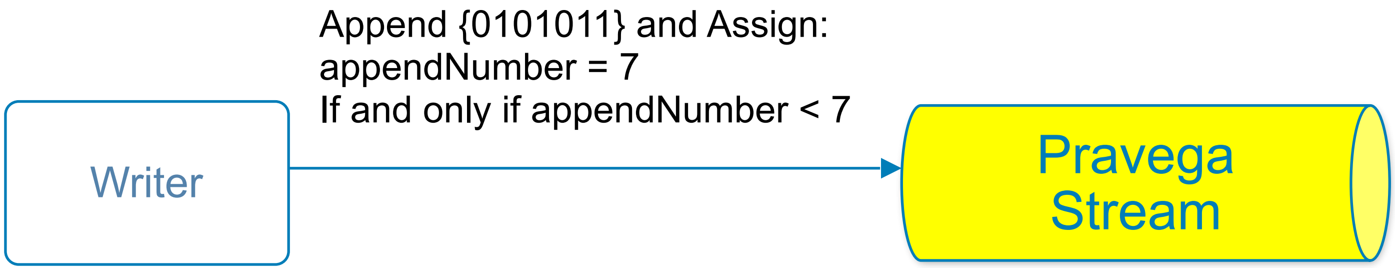
Idempotent Append



Idempotent Append



Idempotent Append

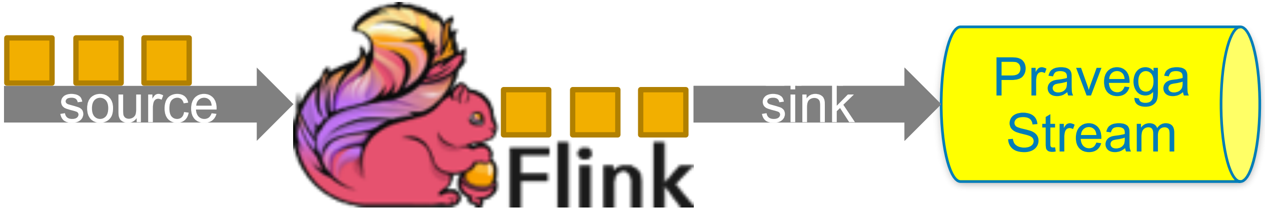


Writer

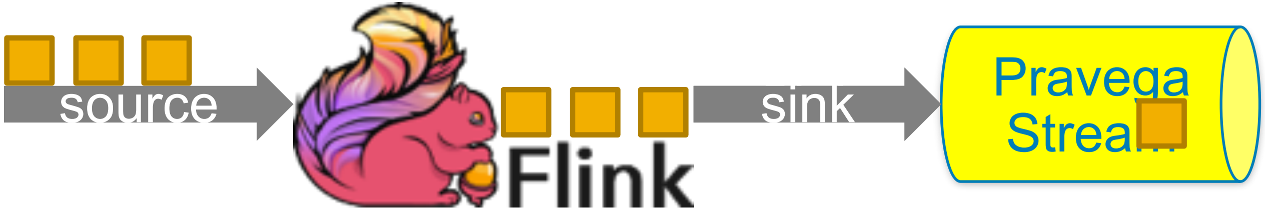
Append {0101011} and Assign:
appendNumber = 7
If and only if appendNumber < 7

Pravega Stream

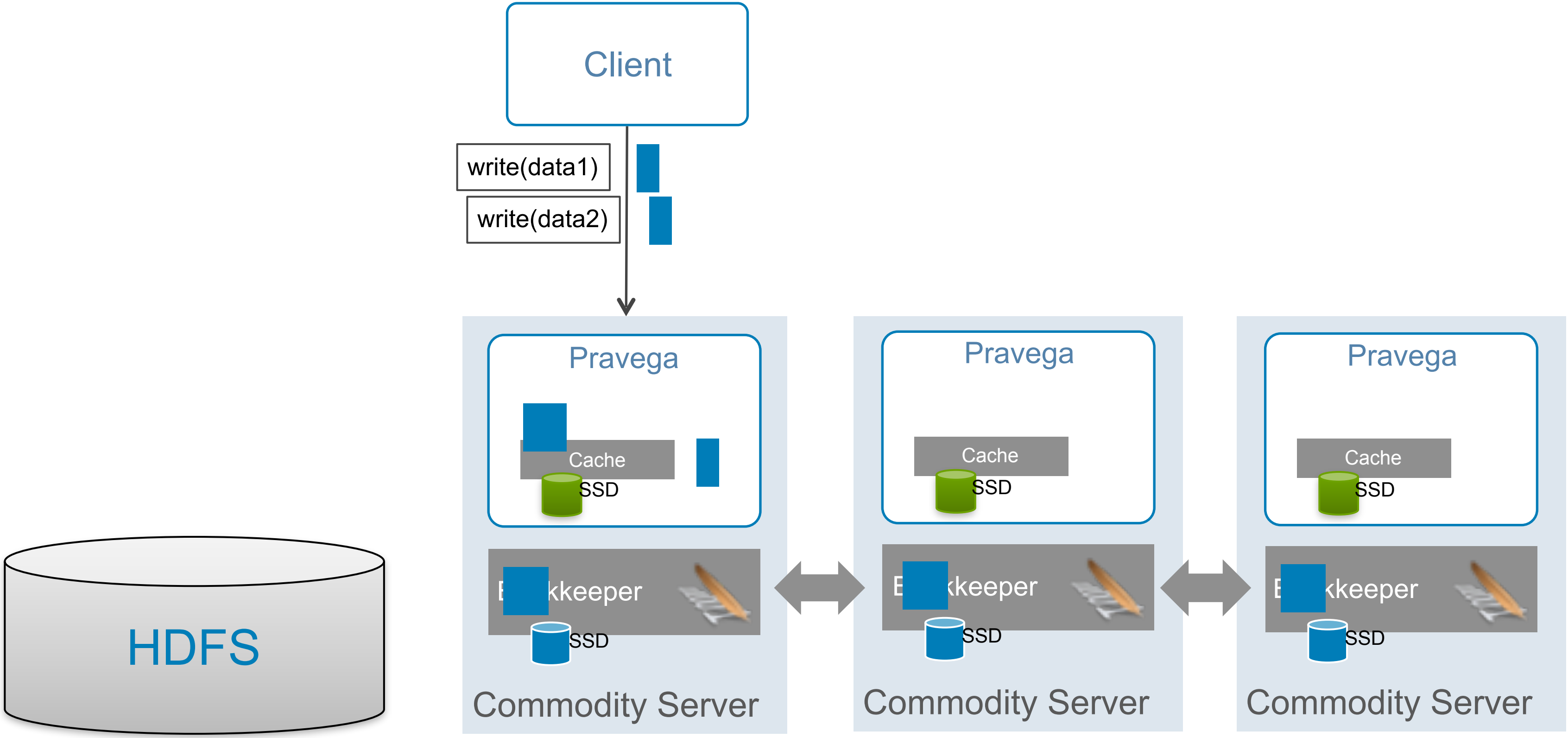
Idempotent output



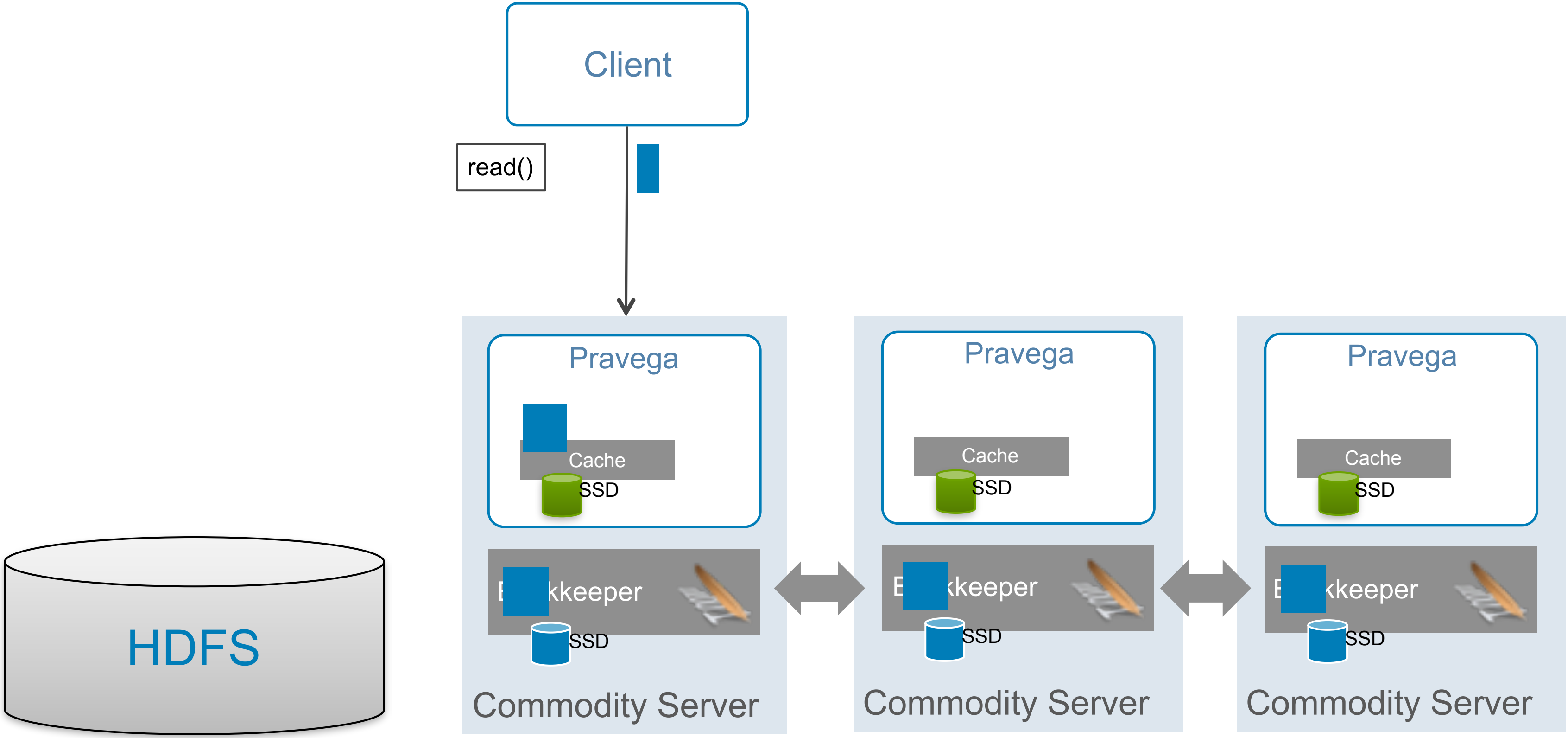
Idempotent output



Architecture overview - Write

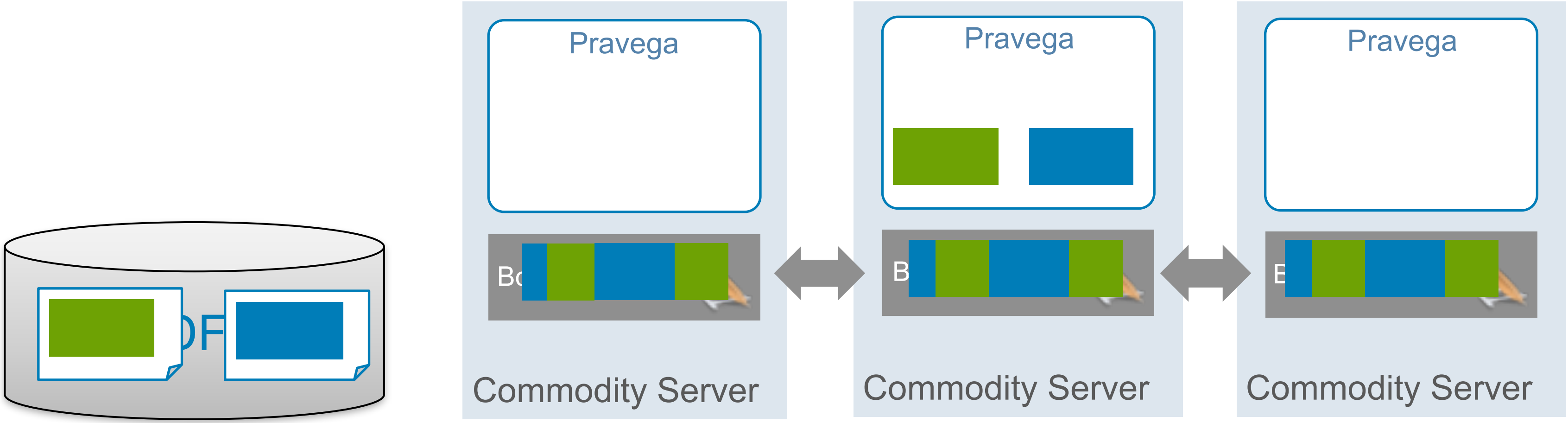


Architecture overview - Read

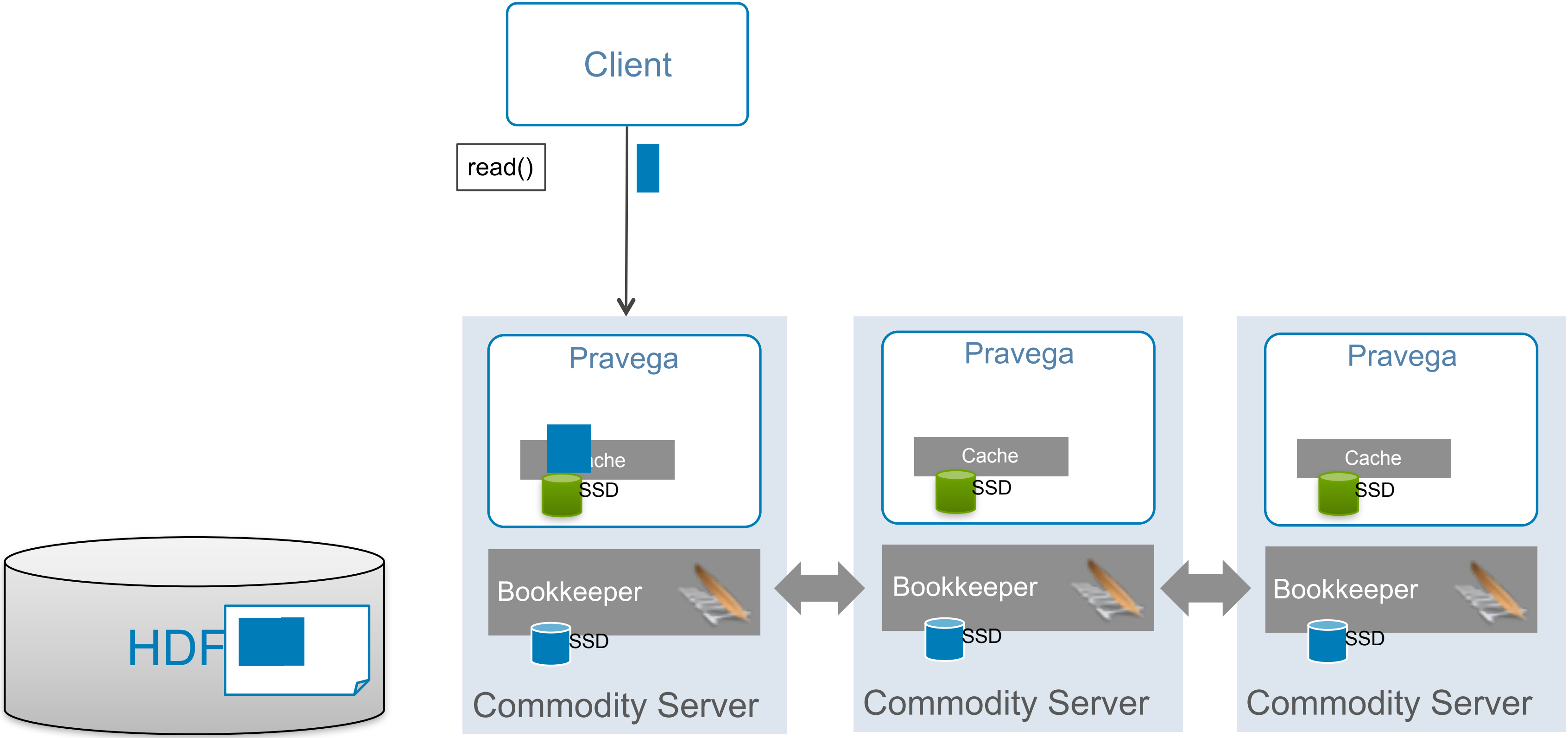


Architecture overview - Evict

- Files in HDFS are organized by Stream Segment
- Read-ahead cache optimizations are employed

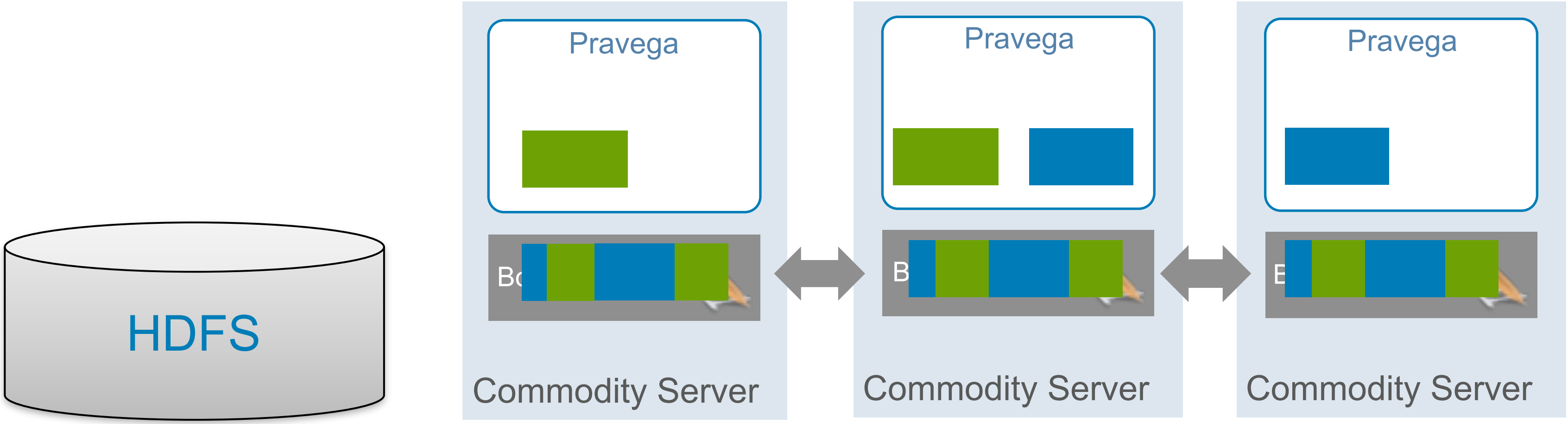


Architecture overview - Read



Architecture overview - Recover

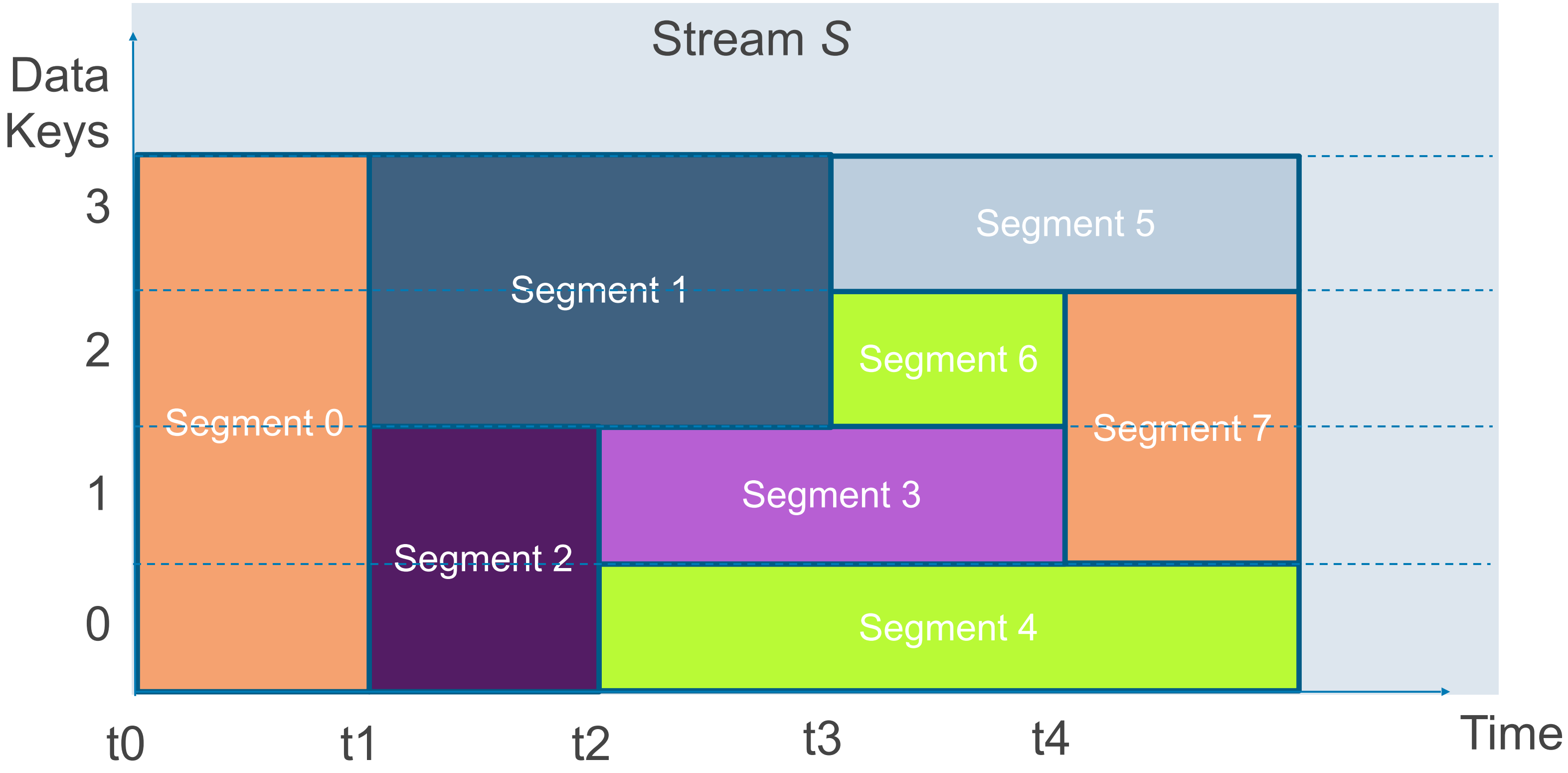
- Data is read from Bookkeeper only in the case of node failure
- Used to reconstitute the cache on the remaining hosts



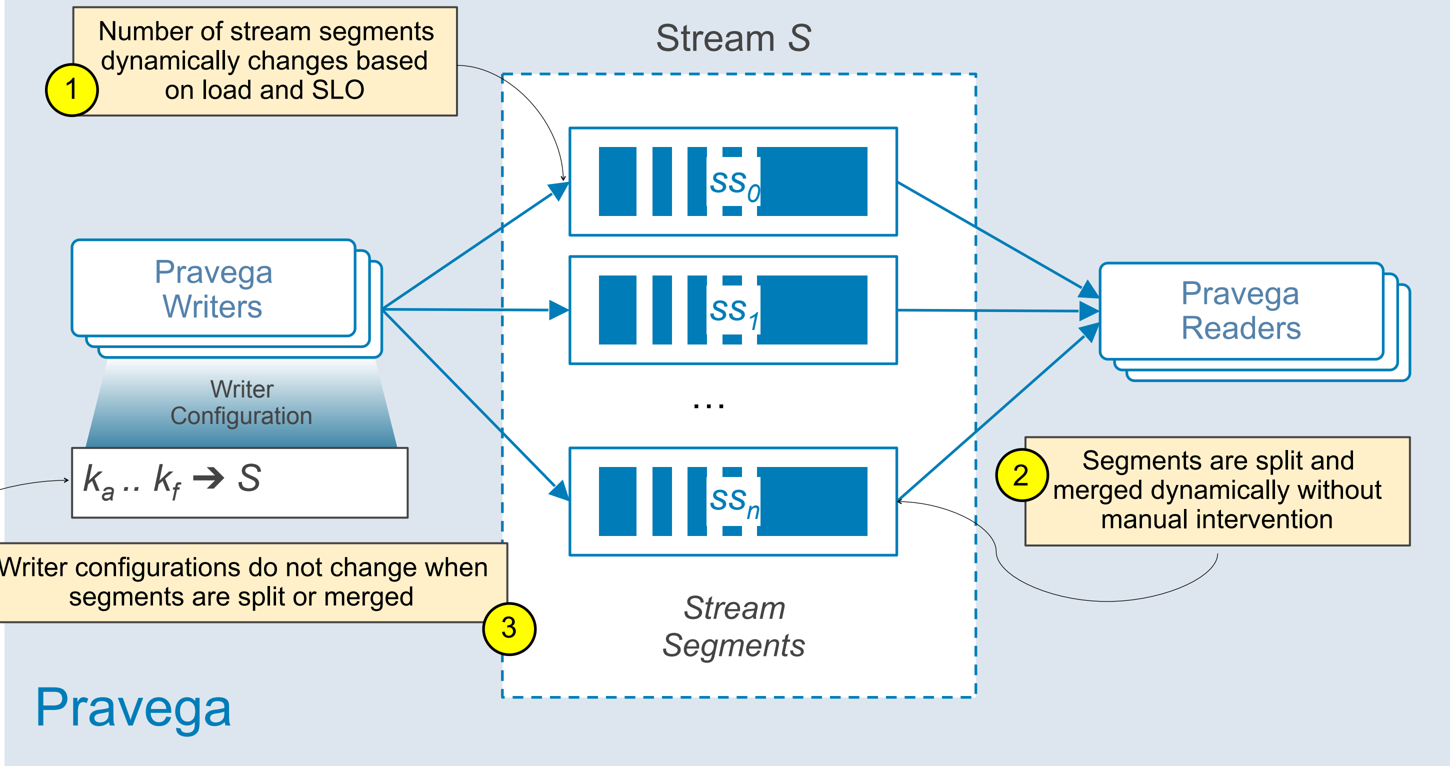
Performance Characteristics

- Fast appends to Bookkeeper
  - Data is persisted durably to disk 3x replicated consistently <10ms
- Big block writes to HDFS
  - Data is mostly cold so it can be erasure encoded and stored cheaply
  - If data is read, the job is likely a backfill so we can use a large read-ahead
- A stream’s capacity is not limited by the capacity of a single machine
- Throughput shouldn’t be either …

Scaling: Segment Splitting & Merging



Scaling: Write Parallelism



1 Number of stream segments dynamically changes based on load and SLO

2 Segments are split and merged dynamically without manual intervention

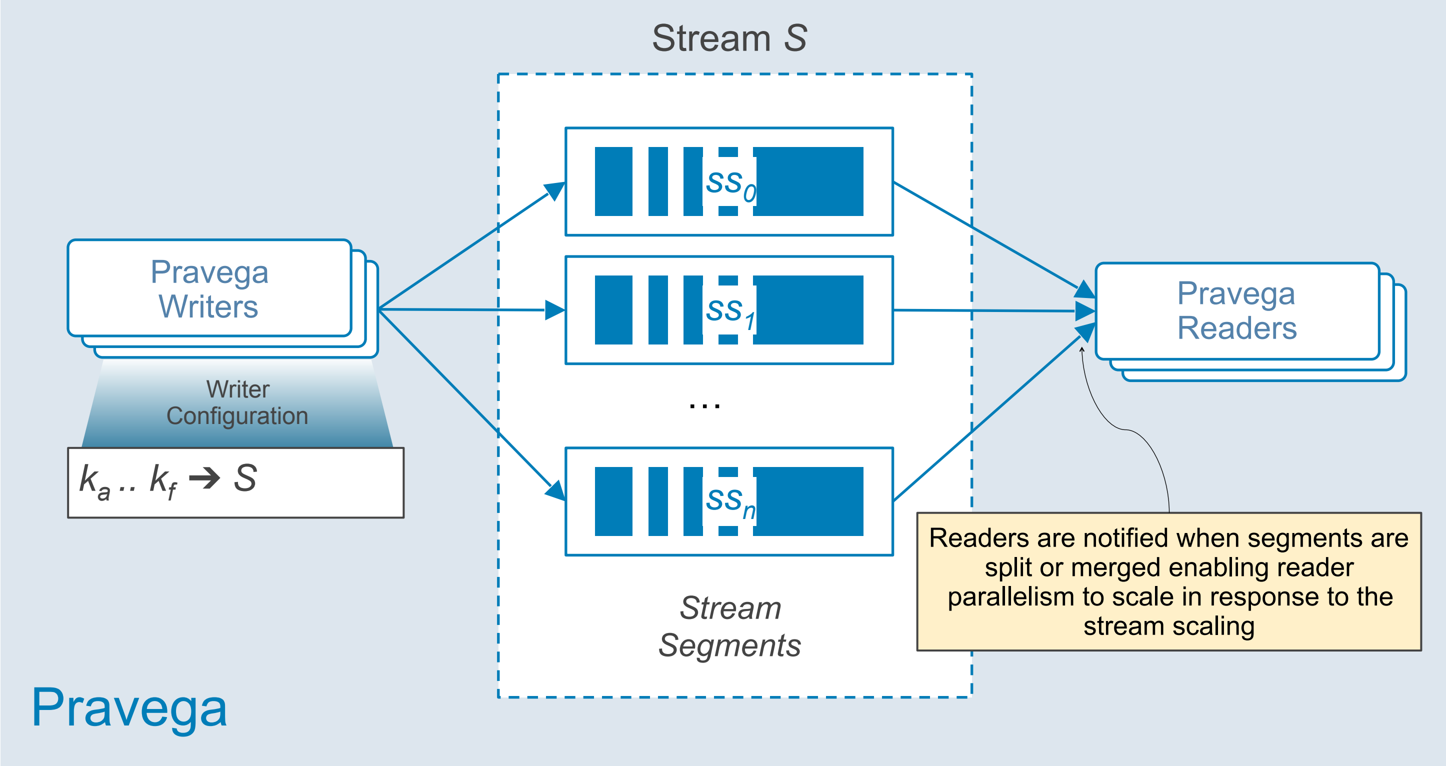
3 Writer configurations do not change when segments are split or merged

DELL EMC

EventWriter API

```
/** A writer can write events to a stream. */
public interface EventStreamWriter {

    /** Send an event to the stream. Event must appear in the stream exactly once */
    AckFuture writeEvent(String routingKey, Type event);

    /** Start a new transaction on this stream */
    Transaction<Type> beginTxn(long transactionTimeout);
}
```

Scaling: Read Parallelism

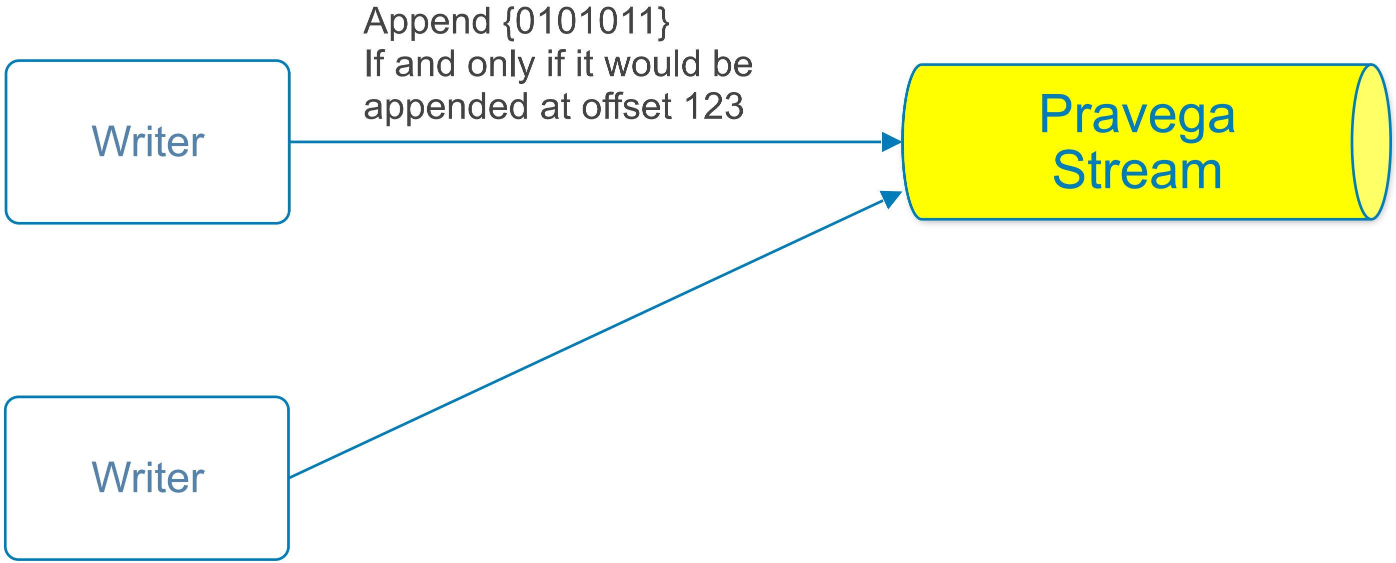


Readers are notified when segments are split or merged enabling reader parallelism to scale in response to the stream scaling

EventReader API

```
public interface EventStreamReader<T> extends AutoCloseable {

        /** Read the next event from the stream, blocking for up to timeout */
        EventRead<T> readNextEvent(long timeout);

        /**
         * Close the reader. The segments owned by this reader will automatically be
         * redistributed to the other readers in the group.
         */
        void close()
}
```

Conditional Append

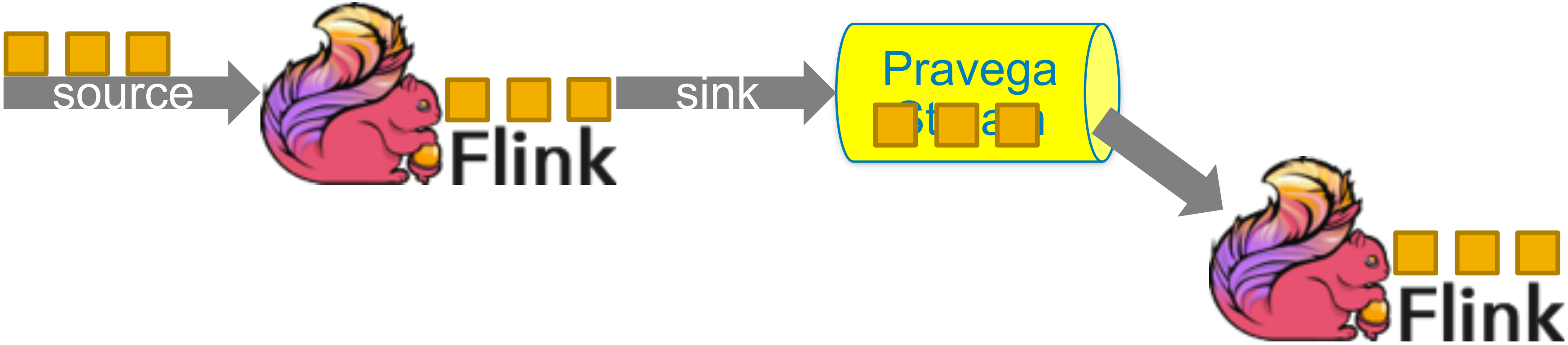


Append {0101011}
If and only if it would be appended at offset 123

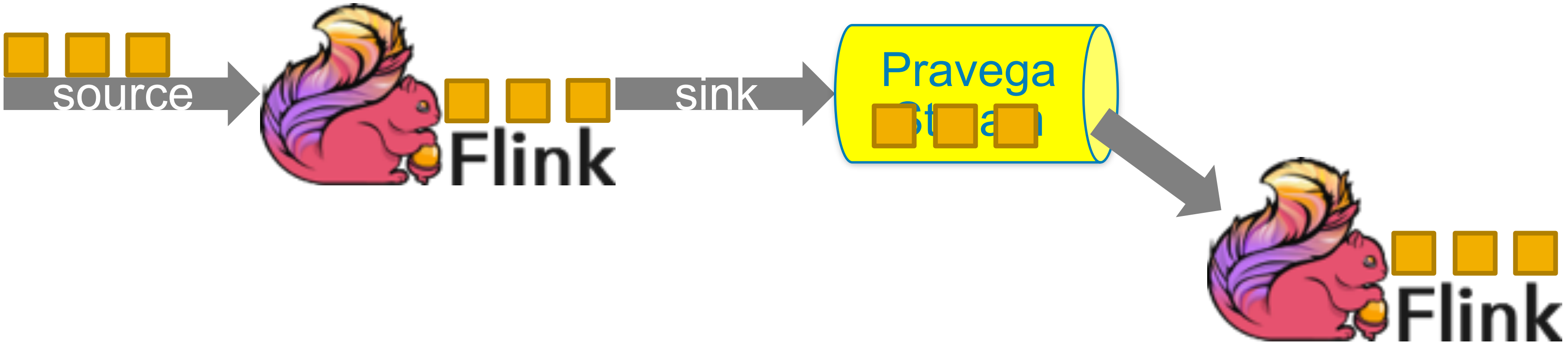
Synchronizer API

```
/** A means to synchronize state between many processes */
public interface StateSynchronizer<StateT> {

    /** Gets the state object currently held in memory */
    StateT getState();

    /** Fetch and apply all updates to bring the local state object up to date */
    void fetchUpdates();

    /** Creates a new update for the latest state object and applies it atomically */
    void updateState(Function<StateT, Update<StateT>> updateGenerator);

}
```

Transactional output



Transactional output

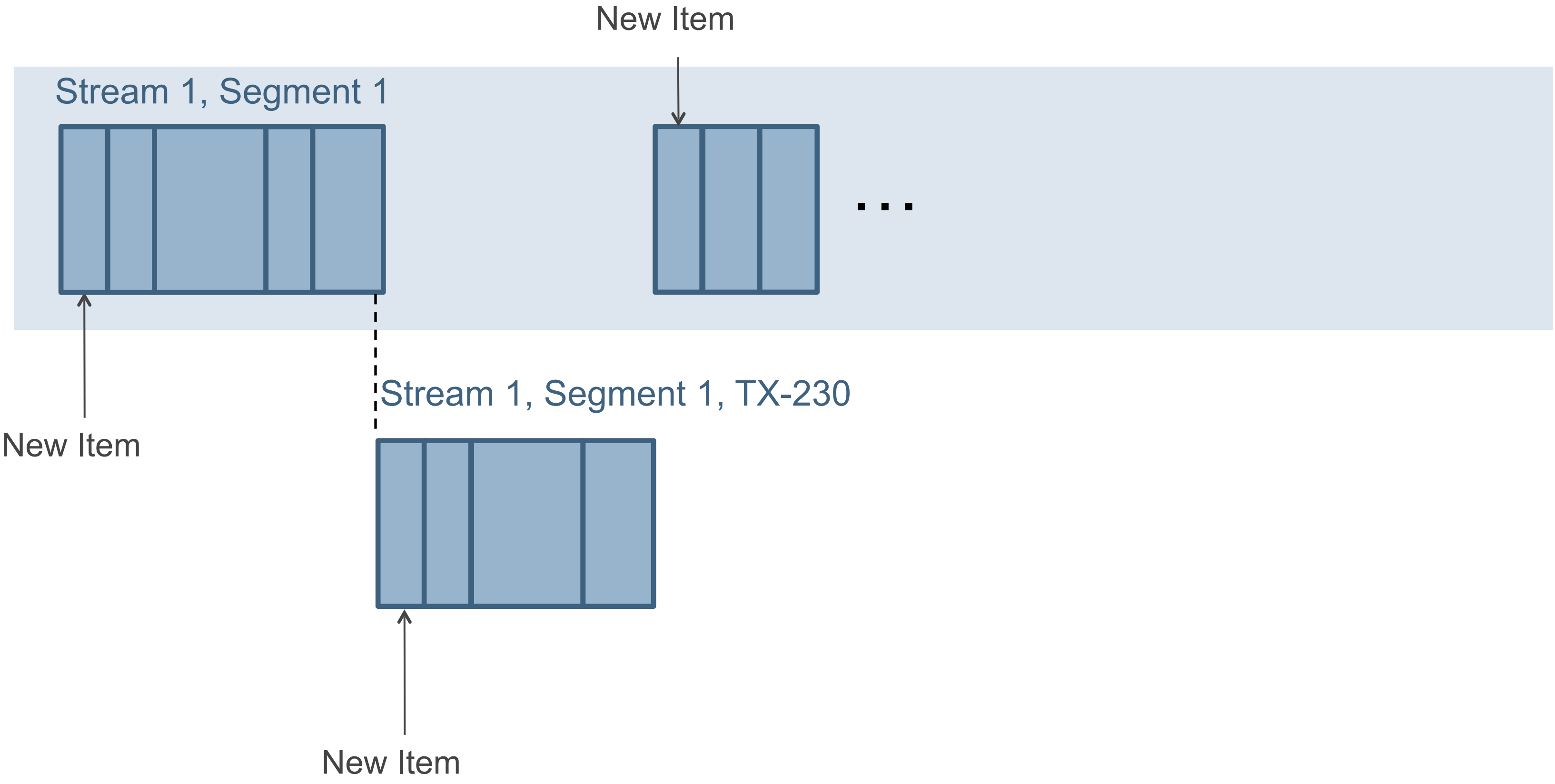


EventWriter and Transaction API

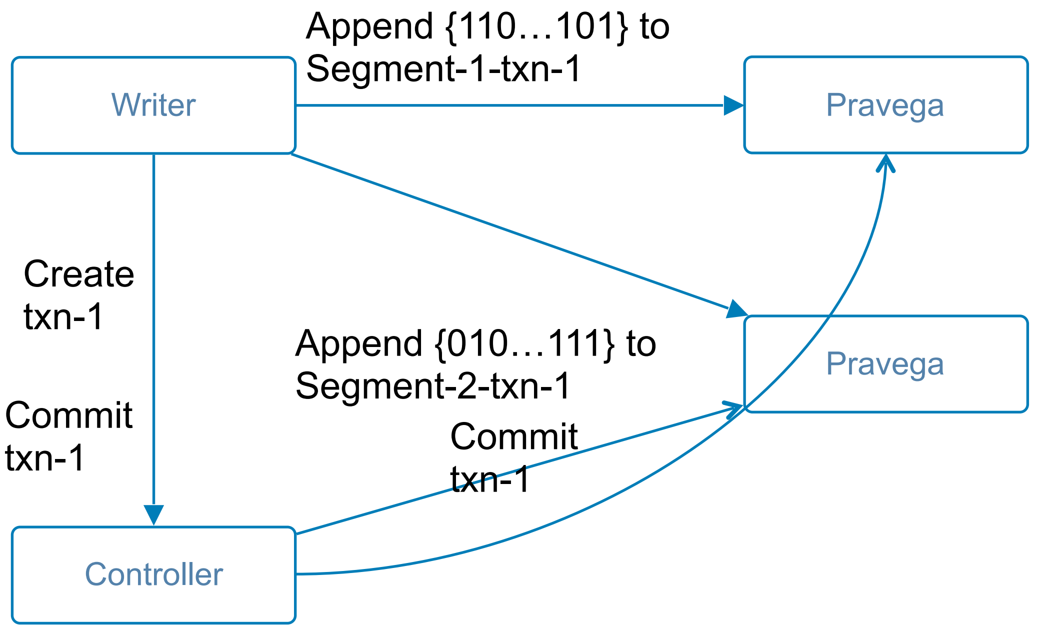
```
/** A writer can write events to a stream. */
public interface EventStreamWriter {

    /** Send an event to the stream. Event must appear in the stream exactly once */
    AckFuture writeEvent(String routingKey, Type event);

    /** Start a new transaction on this stream */
    Transac
}
```

```
public interface Transaction<Type> {
    void writeEvent(String routingKey, Type event) throws TxnFailedException;
    void commit() throws TxnFailedException;
    void abort();
}
```

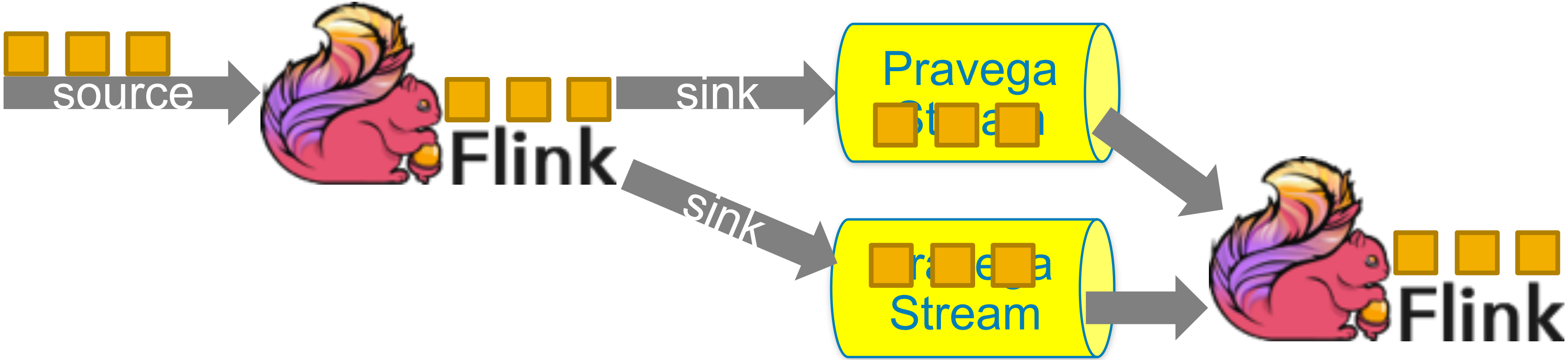
Transactions



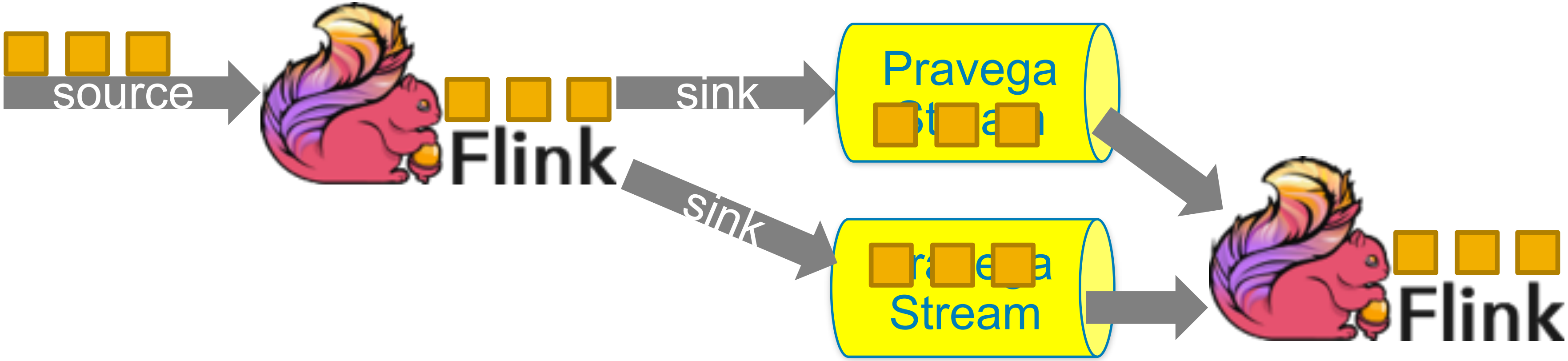
Transactions



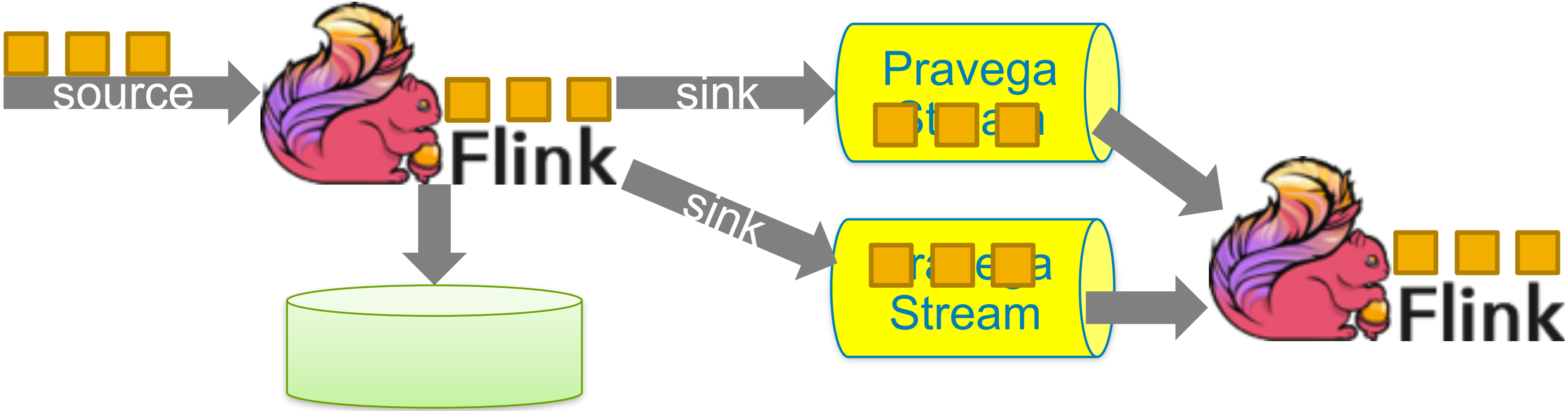
Transactional output



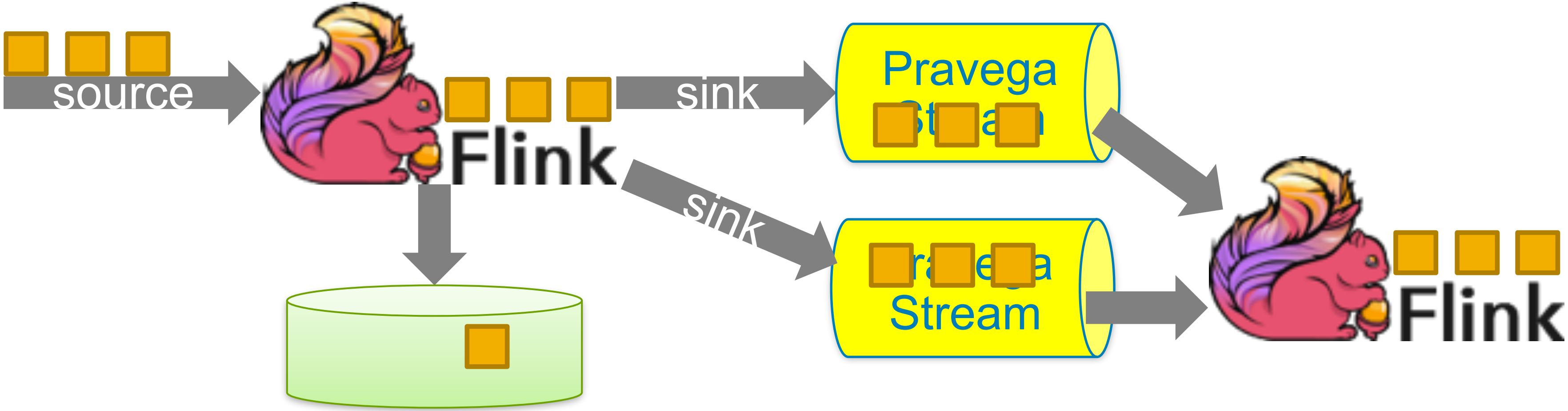
Transactional output



Transactional output



Transactional output



Pravega: Streaming Storage for All

- Pravega: an open source project with an open community
  - To be launched @ Dell EMC World this May 10th
  - Includes infinite byte stream primitive
  - Plus an Ingest Buffer with Pub/Sub built on top of streams
  - And Flink integration!
- Visit the Dell EMC booth here @ Flink Forward to learn more
- Contact us at pravega@emc.com for even more information!

Pravega

BB-8 Drawing

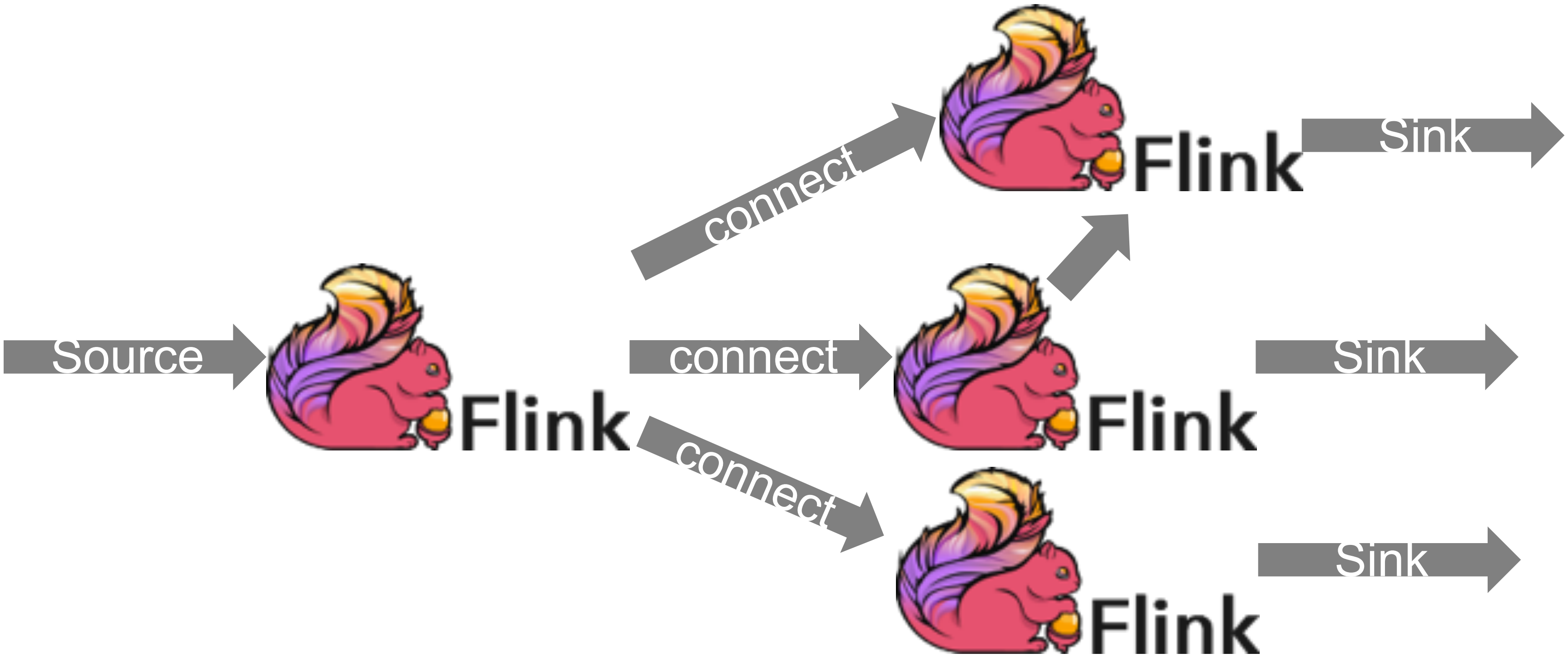
- ➢ Stop by the Dell EMC booth and enter to win
- ➢ Winner will be chosen after the closing Keynote
    - ➢ Must be present to win

Email Pravega@emc.com for the latest news and information on Pravega!

DELL EMC

DELL EMC

Why a new storage system?



Why a new storage system?

Connector	Real Time	Exactly once	Durability	Storage Capacity	Notes
HDFS	No	Yes	Yes	Years	
Kafka	Yes	Source only	Yes* (Flushed but not synced)	Days	Writes are replicated but may not persisted to durable media. (flush.messages=1 bounds this but is not recommended)
RabbitMQ	Yes	Source only	Yes* (slowly)	Days	Durability can be added with a performance hit
Cassandra	No	Yes* (If updates are idempotent)	Yes	Years	App developers need to write custom logic to handle duplicate writes.
Sockets	Yes	No	No	None	

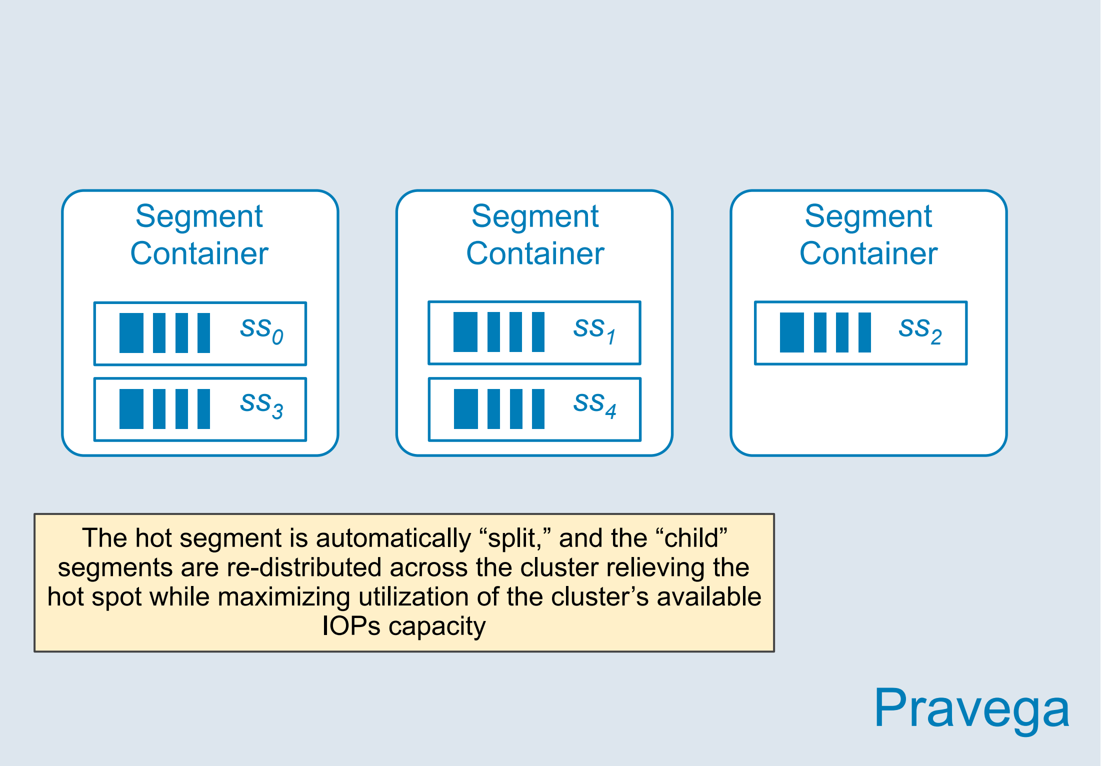
Flink storage needs

	Flink	Implications for storage
Guarantee	Exactly once	Exactly once, consistency
Latency	Very Low	Low latency writes (<10ms)
Throughput	High	High throughput
Computation model	Streaming	Streaming model
Overhead of fault tolerance mechanism	Low	Fast recovery Long retention
Flow control	Natural	Data can backlog Capacity not bounded by single host
Separation of application logic from fault tolerance	Yes	Re-reading data provides consistent results
License	Apache 2.0	Open Source and linkable

Shared config

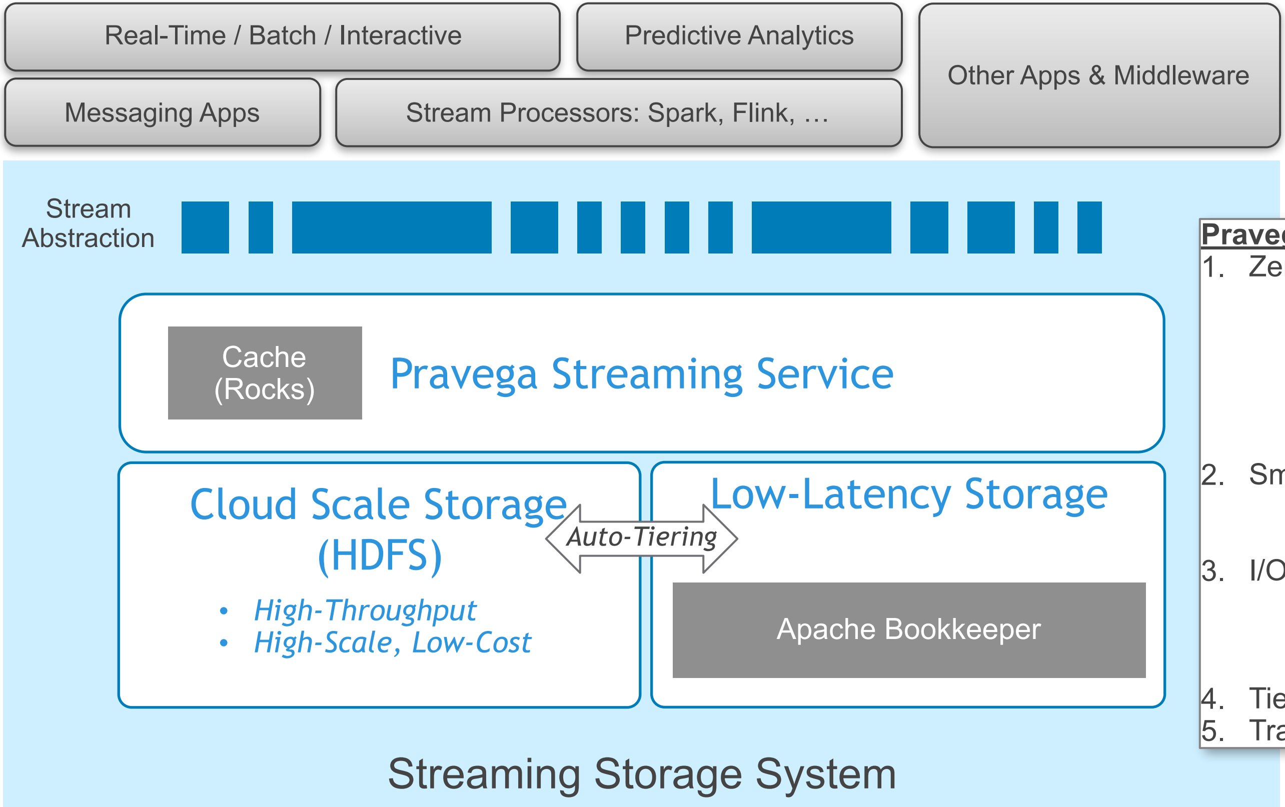
```
public class SharedConfig<K extends Serializable, V extends Serializable> {

    public V getProperty(K key);

    public V putPropertyIfAbsent(K key, V value);

    public boolean removeProperty(K key, V oldValue);

    public boolean replaceProperty(K key, V oldValue, V newValue);

}
```

Smart Workload Distribution



The hot segment is automatically “split,” and the “child” segments are re-distributed across the cluster relieving the hot spot while maximizing utilization of the cluster’s available IOPs capacity

Architecture



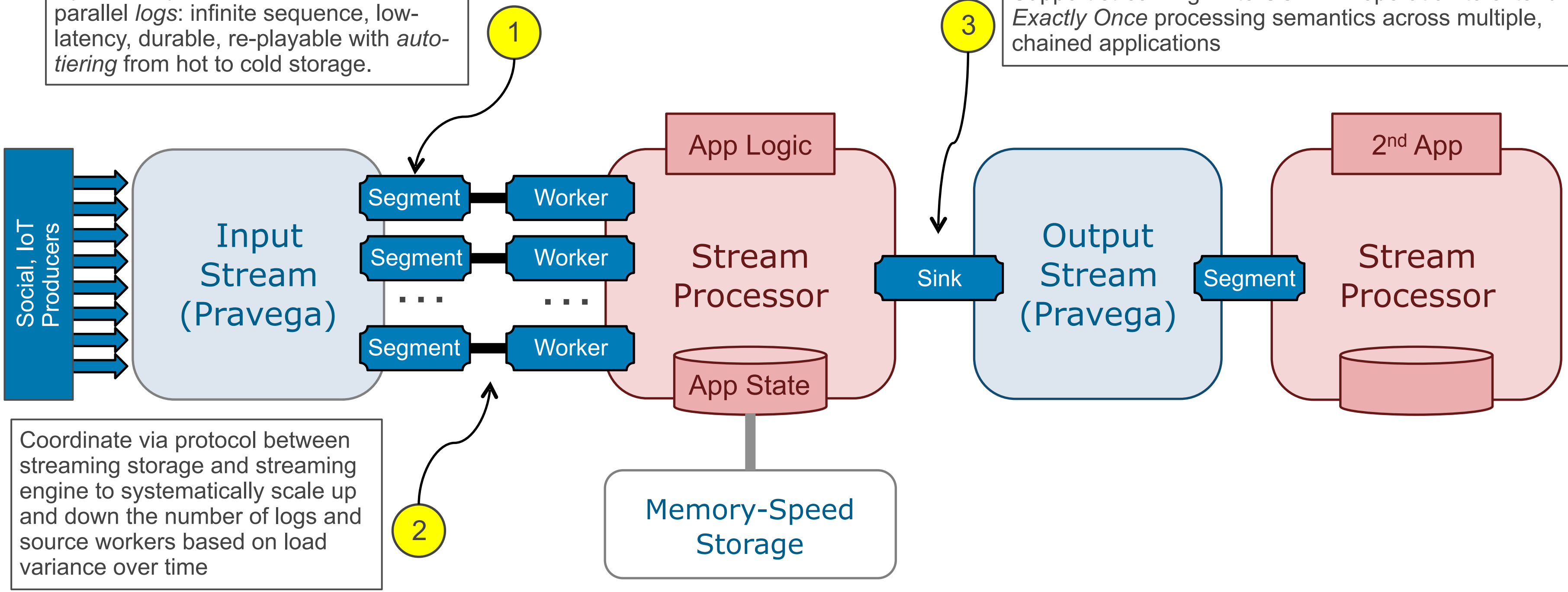
Pravega Design Innovations

1. Zero-Touch Dynamic Scaling
   - Automatically scale read/write parallelism based on load and SLO
   - No service interruptions
   - No manual reconfiguration of clients
   - No manual reconfiguration of service resources
2. Smart Workload Distribution
   - No need to over-provision servers for peak load
3. I/O Path Isolation
   - For tail writes
   - For tail reads
   - For catch-up reads
4. Tiering for “Infinite Streams”
5. Transactions For “Exactly Once”

Pravega Optimizations for Stream Processors

Dynamically split input stream into parallel *logs*: infinite sequence, low-latency, durable, re-playable with *auto-tiering* from hot to cold storage.

Support streaming write COMMIT operation to extend *Exactly Once* processing semantics across multiple, chained applications



Coordinate via protocol between streaming storage and streaming engine to systematically scale up and down the number of logs and source workers based on load variance over time

Comparing Pravega and Kafka Design Points

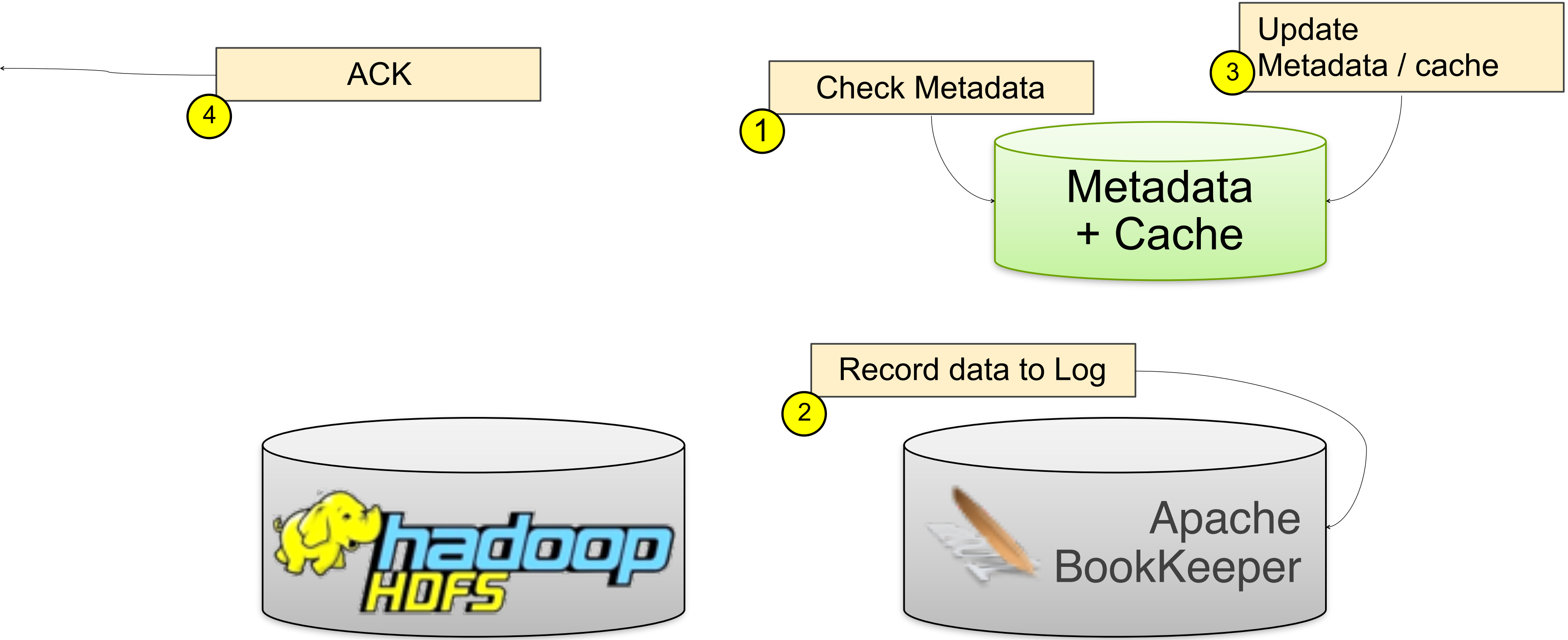
Unlike Kafka, Pravega is designed to be a durable and permanent storage system

Quality	Pravega Goal	Kafka Design Point
Data Durability	Replicated and persisted to disk before ACK	Replicated but not persisted to disk before ACK ✗
Strict Ordering	Consistent ordering on tail and catch-up reads	Messages may get reordered ✗
Exactly Once	Producers can use transactions for atomicity	Messages may get duplicated ✗
Scale	Tens of millions of streams per cluster	Thousands of topics per cluster ✗
Elastic	Dynamic partitioning of streams based on load and SLO	Statically configured partitions ✗
Size	Log size is not bounded by the capacity of any single node	Partition size is bounded by capacity of filesystem on its hosting node ✗
	Transparently migrate/retrieve data from Tier 2 storage for older parts of the log	External ETL required to move data to Tier 2 storage; no access to data via Kafka once moved ✗
Performance	Low (<10ms) latency durable writes; throughput bounded by network bandwidth	Low-latency achieved only by reducing replication/ reliability parameters ✗
	Read pattern (e.g. many catch-up readers) does not affect write performance	Read patterns adversely affects write performance due to reliance on OS filesystem cache ✗

Attributes

Connector	Streaming	Exactly once	Durability	Storage Capacity
HDFS	No	Yes	Yes	Years
Kafka	Yes	Source only	Yes* (Flushed but not synced)	Days
Pravega	Yes: Byte oriented and event oriented	Yes. With either idempotent producers, or transactions	Yes. Always flushed and synced, with low latency.	As much as you can fit in your HDFS cluster.

Architecture overview - Write



4 ACK

1 Check Metadata

3 Update Metadata / cache

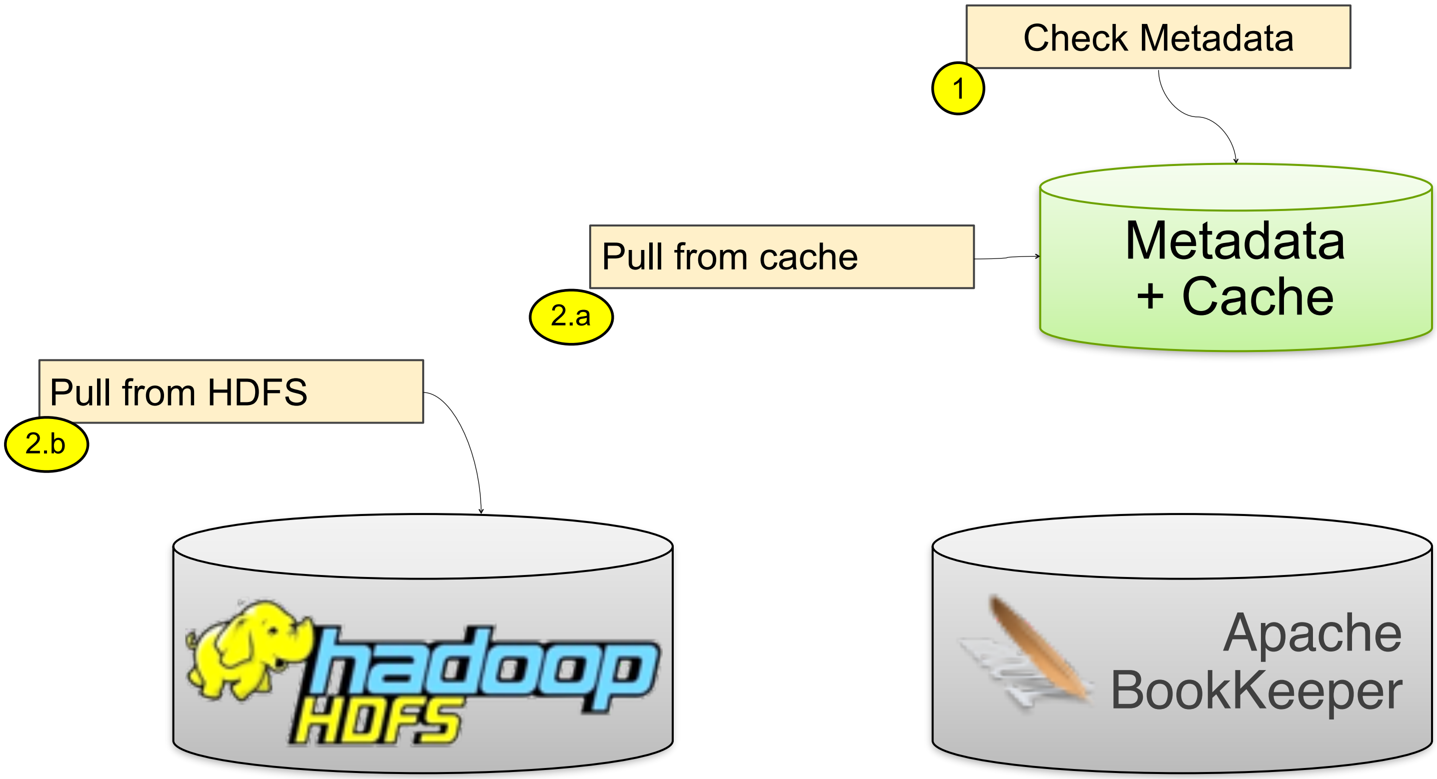
Metadata + Cache

2 Record data to Log

hadoop HDFS

Apache BookKeeper

Architecture overview - **Read**



Check Metadata

1

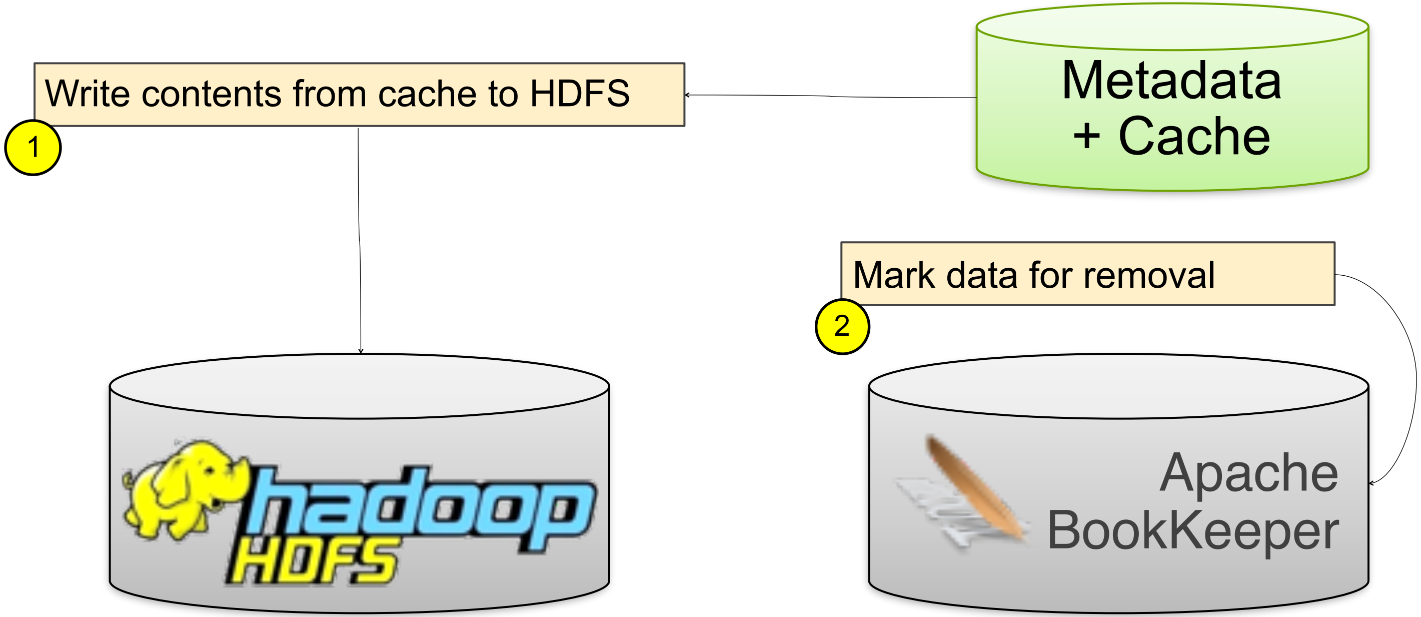
Pull from cache

2.a

Pull from HDFS

2.b

Architecture overview - **Evict**



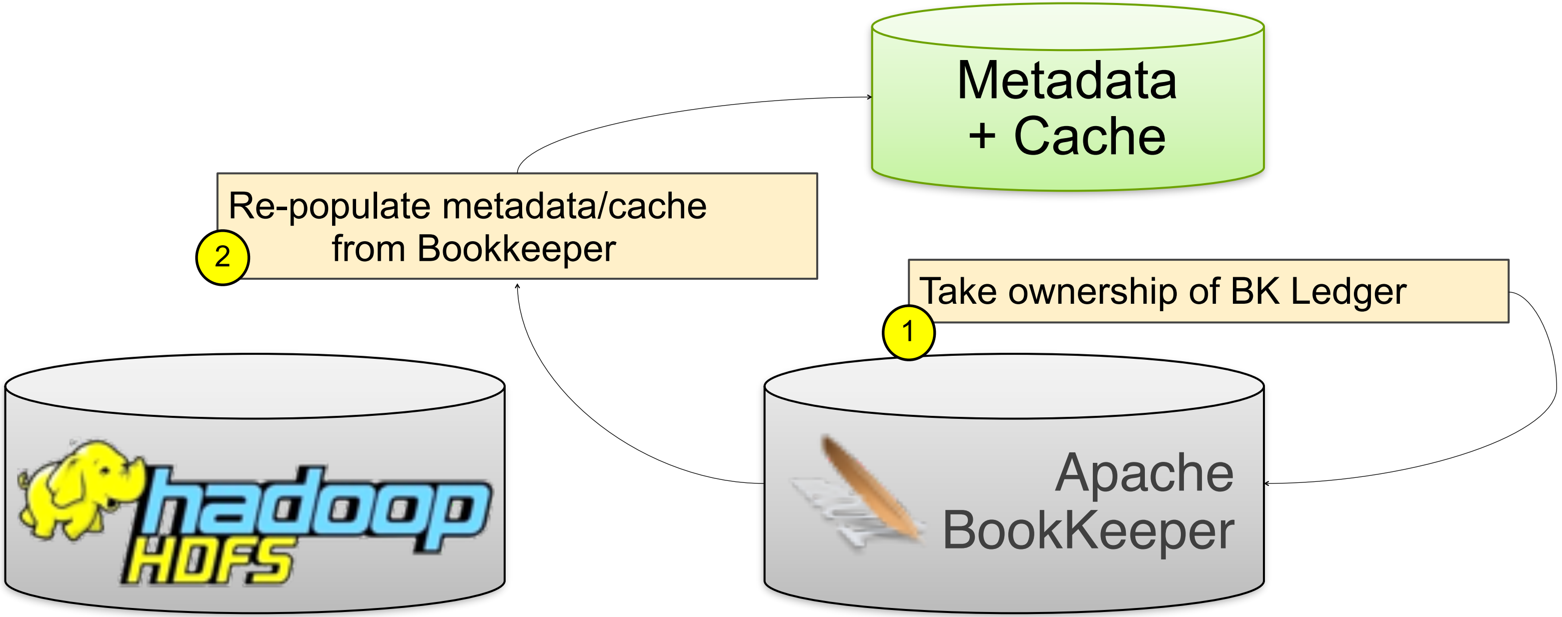
1 Write contents from cache to HDFS

Metadata + Cache

2 Mark data for removal

Apache BookKeeper

Architecture overview - **Recover**



Metadata + Cache
2
Re-populate metadata/cache from Bookkeeper
1
Take ownership of BK Ledger
hadoop HDFS
Apache BookKeeper