Stateful Streaming Data Pipelines with Apache Apex

Chandni Singh
PMC and Committer, Apache Apex
Founder, Simplifi.it

Timothy Farkas
Committer, Apache Apex
Founder, Simplifi.it

Agenda

- Introduction to Apache Apex
- Managed State
- Spillable Data-structures
- Questions

What is Apache Apex

- Distributed data processing engine
- Runs on Hadoop
- Real-time streaming
- Fault-tolerant

Anatomy of An Apex Application

- **Tuple:** Discrete unit of information sent from one **operator** to another.
- **Operator:** Java code that performs an operation on **tuples**. The code runs in a yarn container on a yarn cluster.
- **DAG:** Operators can be connected to form an application. **Tuple** transfer between **operators** is 1-way, so the application forms a Directed Acyclic Graph.
- **Window Marker:** An Id that is associated with **tuples** and **operators**, and is used for fault-tolerance.

Anatomy of An Apex Operator

```
public class MyOperator implements Operator {
    private Map<String, String> inMemState = new HashMap<>(); // checkpointed in memory state
    private int myProperty;

    public final transient DefaultInputPort<String> inputPort = new DefaultInputPort<String>() {
        public void process(String event) {
            // Custom event processing logic
        }
    }

    public void setup(Context context) { // One time setup tasks to be performed when the operator first starts
    }

    public void beginWindow(long windowId) { // Next window has started
    }

    public void endWindow() {
    }

    public void teardown() { // Operator is shutting down. Any cleanup needs to be done here.
    }

    public void setMyProperty(int myProperty) {
        this.myProperty = myProperty
    }

    public int getMyProperty() { return myProperty}
}
```

Fault tolerance in Apex

- Apex inserts window markers with IDs in the data stream, which operators are notified of.
- It provides fault-tolerance by checkpointing the state of every operator in the pipeline every N windows.
- If an operator crashes, it restores the operator with the state corresponding to a checkpointed window.
- **Committed window**: In the simple case, when all operators are checkpointed at the same frequency, committed window is the latest window which has been checkpointed by all the operators in the DAG.

What is the problem?

- Time to checkpoint $\propto$ size of operator state
- With increasing state, the operator will eventually crash.
- Even before the operator crashes, the platform may assume that the operator is unresponsive and instruct the Yarn to kill it.

Managed State - Introduction

A reusable component that can be added to any operator to manage its key/value state.

- Checkpoints key/value state incrementally.
- Allows to set a threshold on the size of data in memory. Data that has been persisted, is off-loaded from memory when the threshold is reached.
- Keys can be partitioned in user-defined buckets which helps with operator partitioning and efficient off-loading from memory.
- Key/values are persisted on hdfs in a state that is optimized for querying.
- Purges stale data from disk.

Managed State API

- Write to managed state

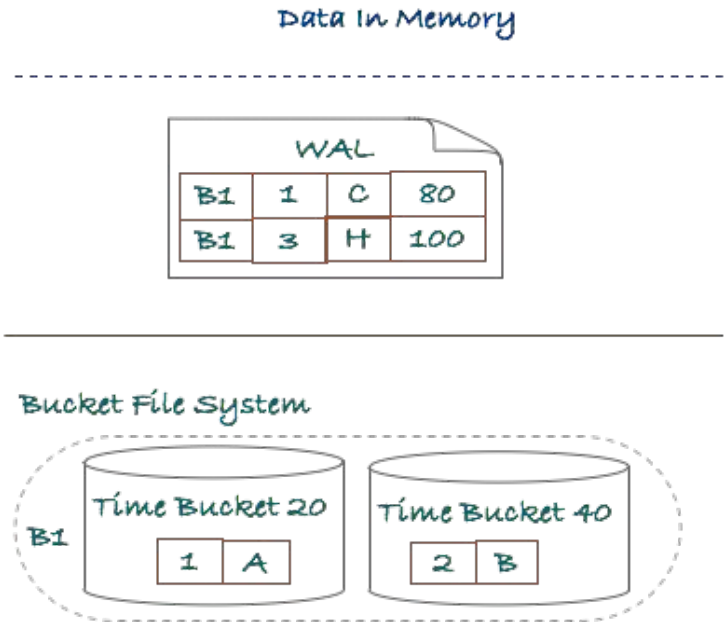
```
managedState.put(1L, key, value)
```

- Read from managed state

```
managedState.getSync(1L, key)
```

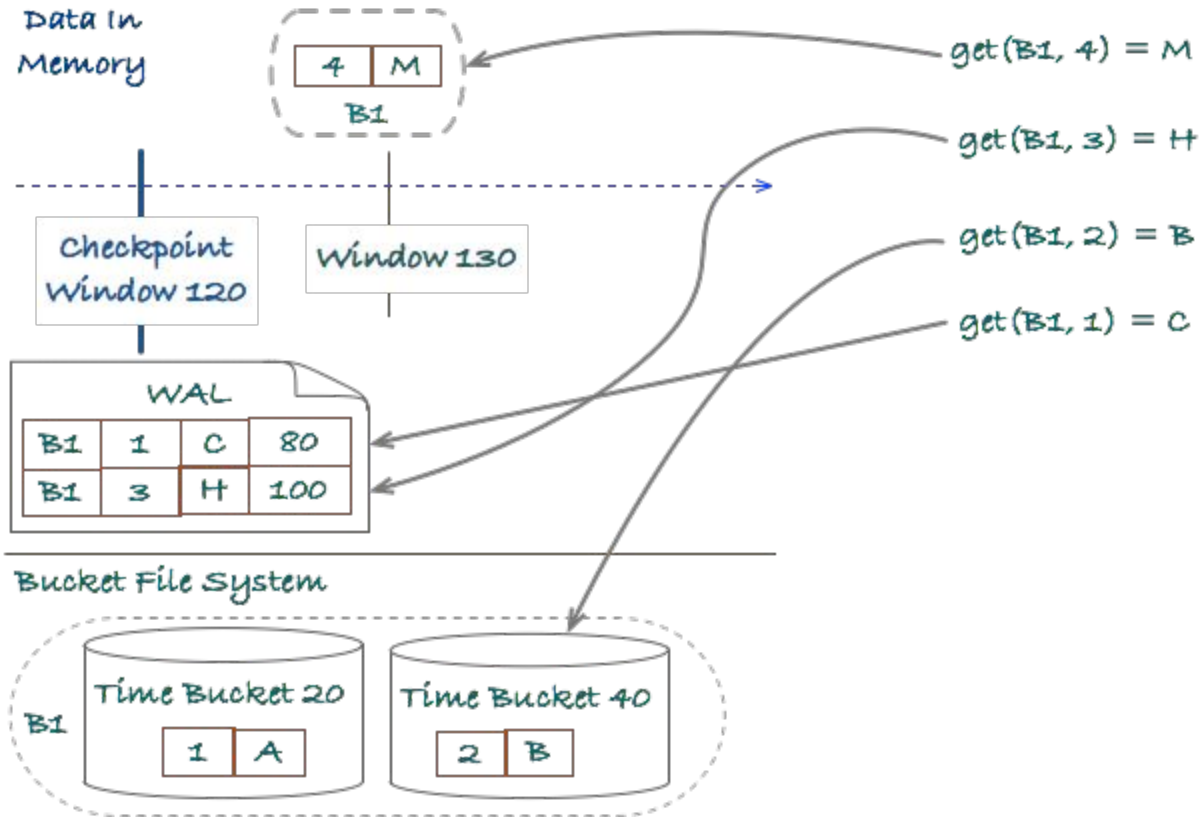
```
managedState.getAsync(1L, key)
```

Architecture



For simplicity, in the following examples we will use window Ids for time buckets because window Ids roughly correspond to processing time.

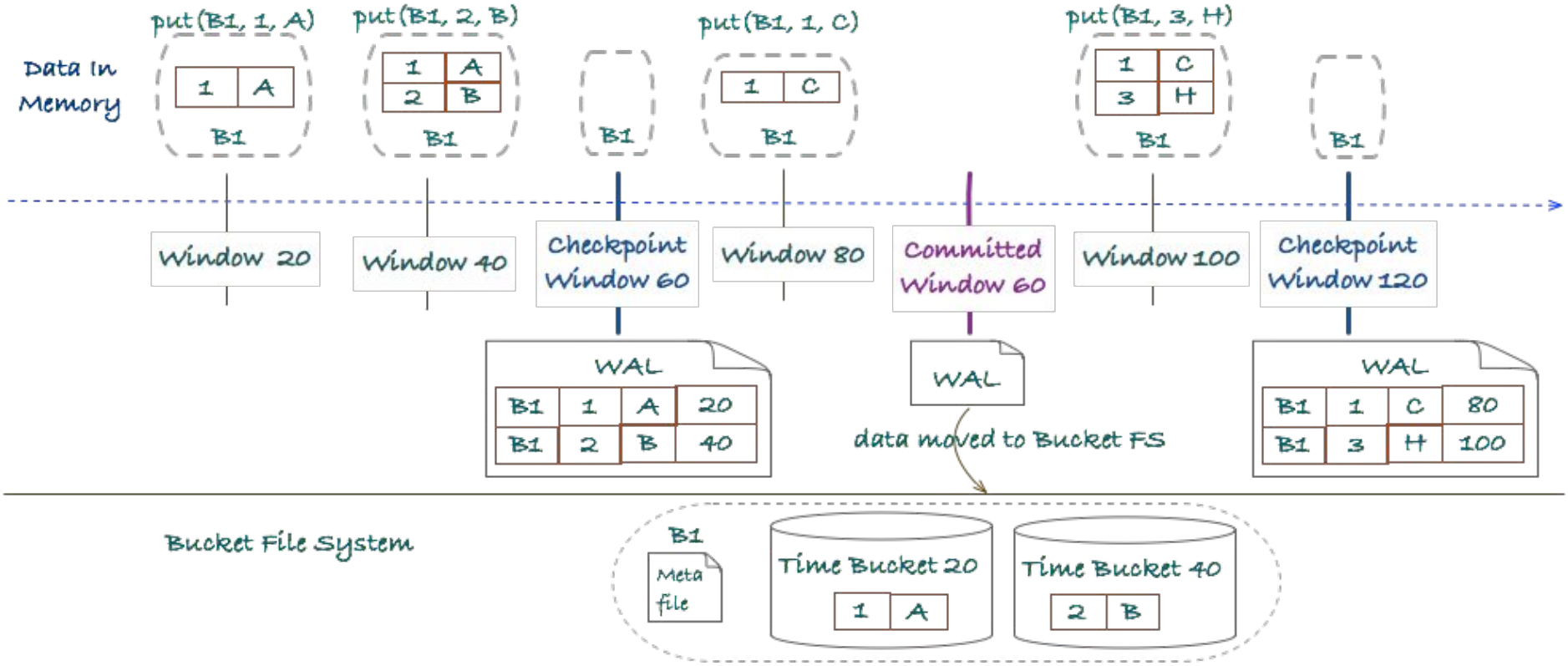
Read from Managed State



Writes to Managed State

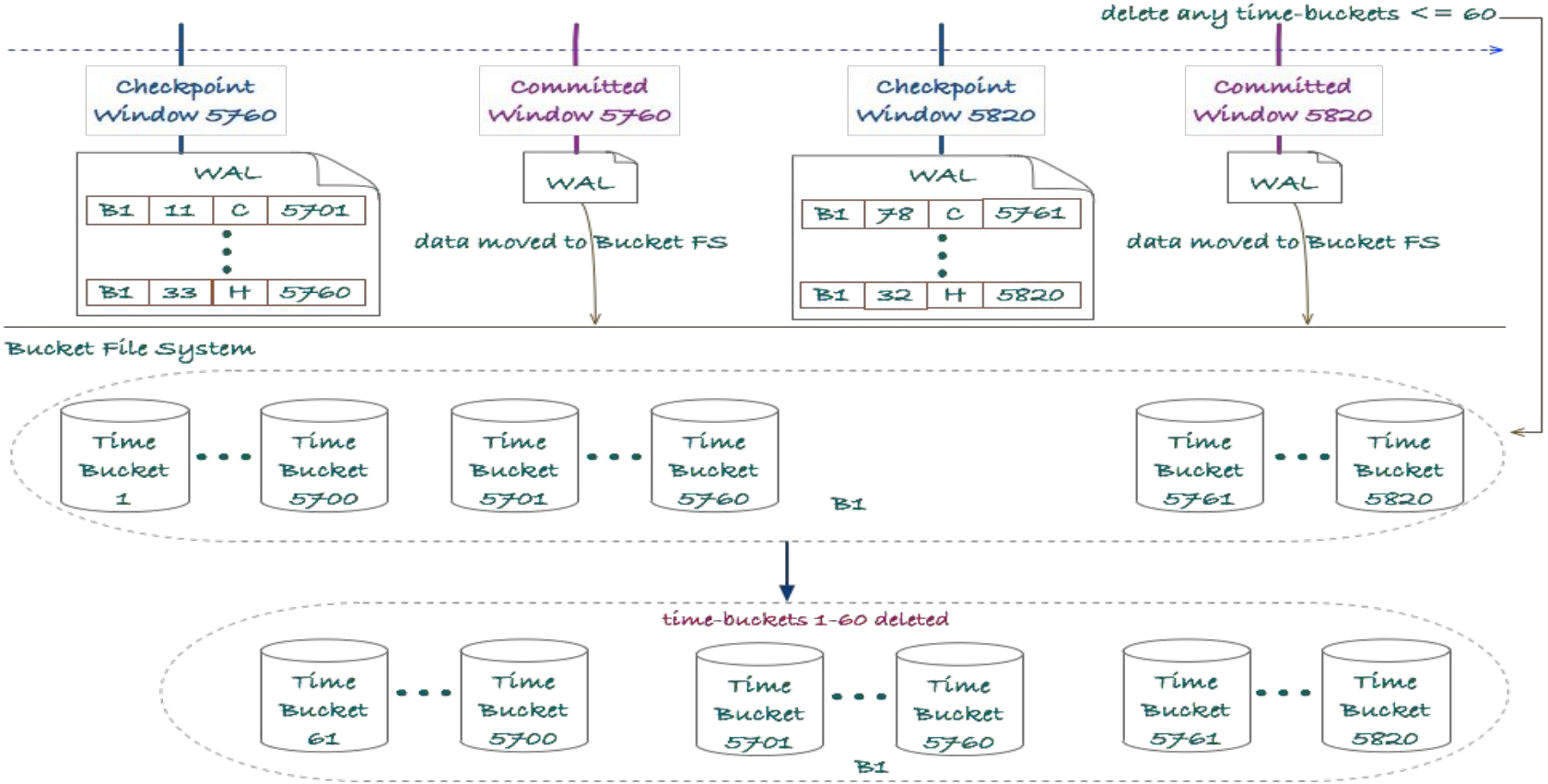
- Key/Values are put in the bucket cache.
- At checkpoints, data from the bucket cache is moved to checkpoint cache which is written to WAL.
- When a window is committed, data in the WAL till the current committed window is transferred to key/value store which is the Bucket File System.

Writes to Managed State - Continued



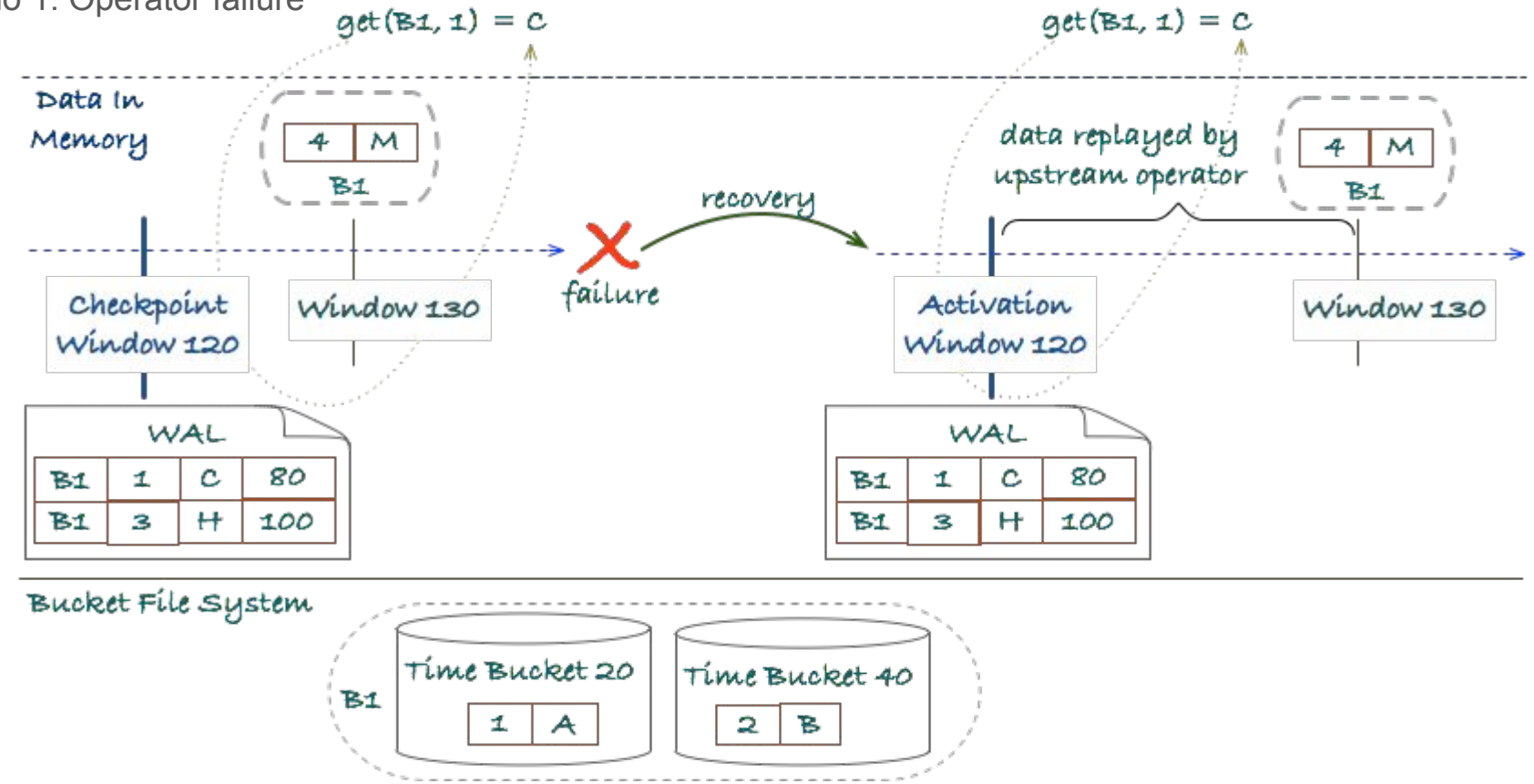
Purging of Data

Delete time-buckets older than 2 days. 2 days are approximately equivalent to 5760 windows.



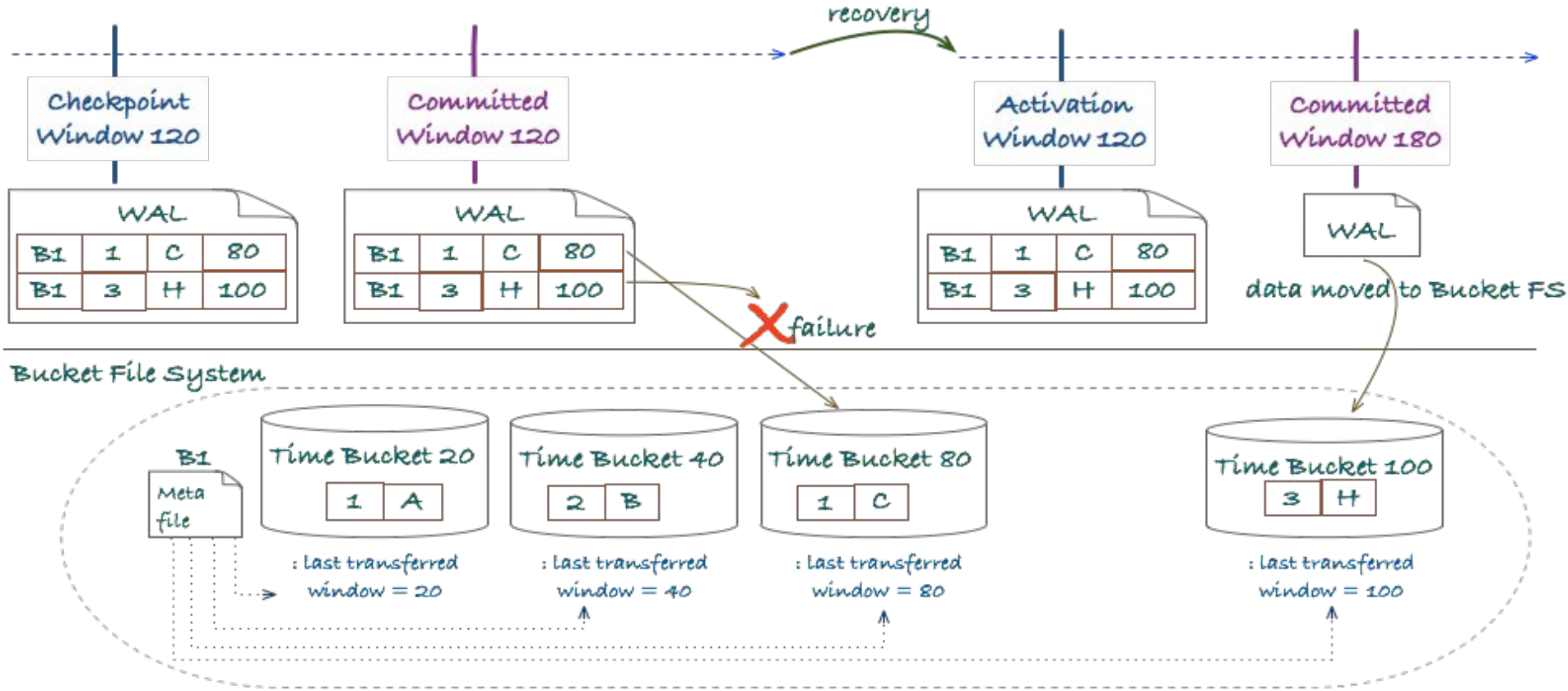
Fault-tolerance in Managed State

Scenario 1: Operator failure



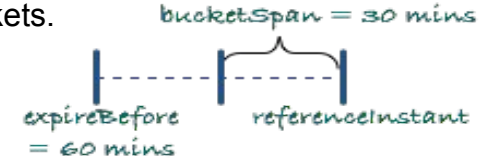
Fault-tolerance in Managed State

Scenario 2: Transferring data from WAL to Bucket File System



Implementations of Managed State

	ManagedStateImpl	ManagedTimeStateImpl	ManagedTimeUnifiedStateImpl

Buckets	Users specify buckets	Users specify buckets	Users specify time properties which are used to create buckets.
			 
			Example:
			`bucketSpan = 30 minutes`
			`expireBefore = 60 minutes`
			`referenceInstant = now, then`
			`Number of buckets = 60/30 = 2`

Data on Disk	A bucket data is partitioned into time-buckets. Time-buckets are derived using **processing time**.	A bucket data is partitioned into time-buckets. Time-buckets are derived using **event time**.	In this implementation a bucket is already a time-bucket so it is not partitioned further on disk.
Operator Partitioning	A bucket belongs to a single partition. Multiple partitions cannot write to the same bucket.	Same as ManagedStateImpl	Multiple partitions can write to the same time-bucket. On the disk each partition's data is segregated by the operator id.

Spillable Data Structures

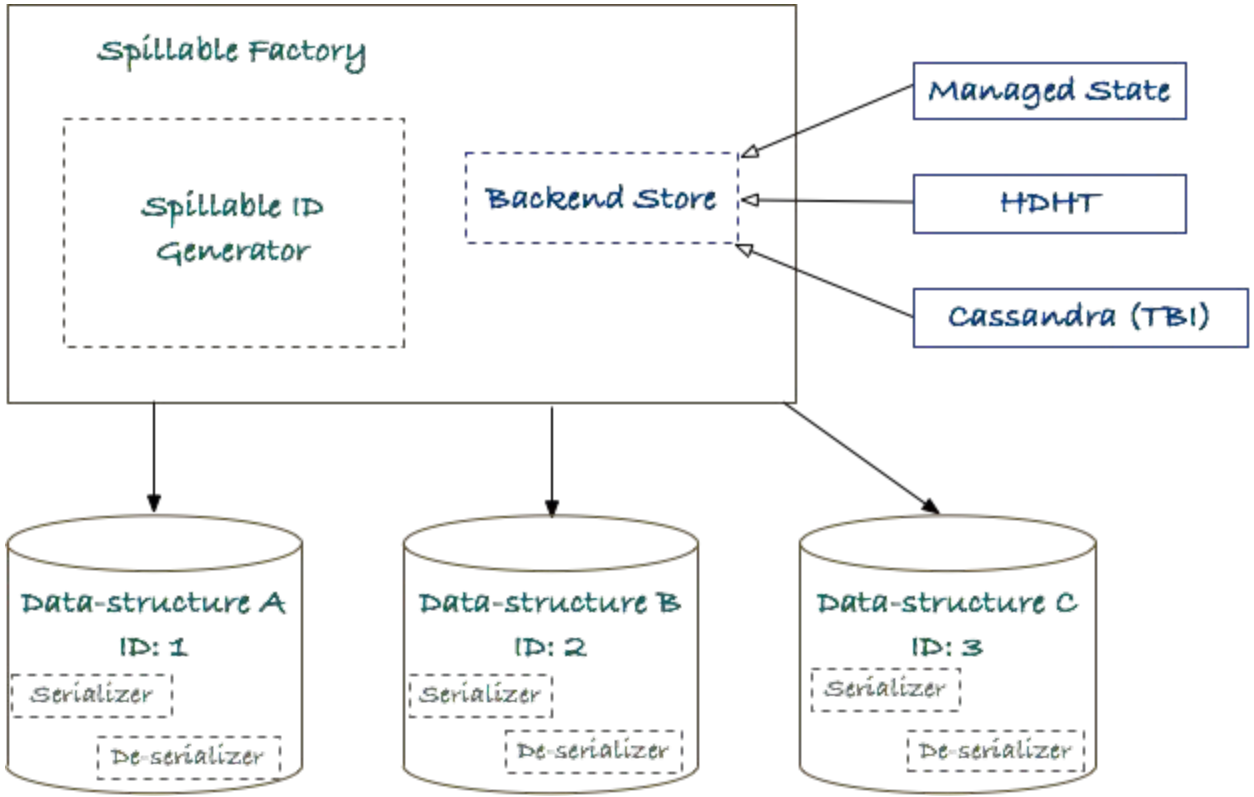
Why Spillable Data Structures?

```
store.put(0L, new Slice(keyBytes), new Slice(valueBytes));

valueSlice = store.getSync(0L, new Slice(keyBytes));
```

- More cognitive load to worry about the details of storing data.
- We are used to working with Maps, Lists, and Sets.
- But we can’t work with simple in memory data structures.
- We need to decouple data from how we serialize and deserialize it.

Spillable Data Structures Architecture

- Spillable Data Structures are created by a factory
- Backend store is pluggable
- The factory has an Id Generator, which generates a unique Id (key prefix) for each Spillable Data Structure
- Serializer and deserializer a configured for each data structure individually



Spillable Data Structures Usage

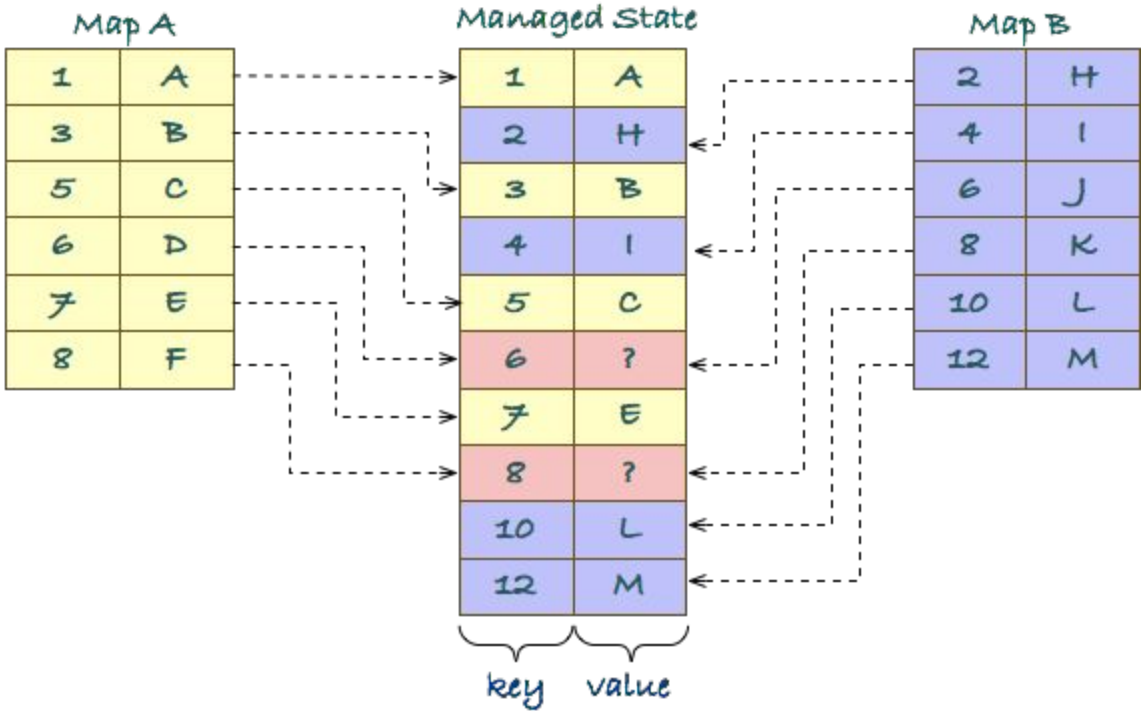
```
public class MyOperator implements Operator {
   private SpillableStateStore store;
      private SpillableComplexComponent spillableComplexComponent;
      private Spillable.SpillableByteMap<String, String> mapString = null;

    public final transient DefaultInputPort<String> inputPort = new DefaultInputPort<String>() {
        public void process(String event) { /* Custom event processing logic */ }
    }
    public void setup(Context context) {
            if (spillableComplexComponent == null) {
                spillableComplexComponent = new SpillableComplexComponentImpl(store);
                mapString = spillableComplexComponent.newSpillableByteMap(0, new StringSerde(), new StringSerde());
      }
      spillableComplexComponent.setup(context);
    }

    public void beginWindow(long windowId) { spillableComplexComponent.beginWindow(windowId); }
    public void endWindow() { spillableComplexComponent.endWindow(); }
    public void teardown() { spillableComplexComponent.teardown(); }

    // Some other checkpointed callbacks need to be overridden and called on spillableComplexComponent, but are omitted
for shortness.
    public void setStore(SpillableStateStore store) { this.store = Preconditions.checkNotNull(store); }
    public SpillableStateStore getStore() { return store; }}
```

Building a Map on top Of Managed State

```
// Psuedo code
public static class SpillableMap<K, V> implements Map<K, V> {
     private ManagedState store;
     private Serde<K> serdeKey;
     private Serde<V> serdeValue;

     public SpillableMap(ManagedState store, Serde<K> serdeKey, Serde<V> serdeValue) {
          this.store = store;
          this.serdeKey = serdeKey;
          this.serdeValue = serdeValue;
     }

     public V get(K key) {
          byte[] keyBytes = serdeKey.serialize(key)
          byte[] valueBytes = store.getSync(0L, new Slice(keyBytes)).toByteArray()
          return serdeValue.deserialize(valueBytes);
     }

     public void put(K key, V value) { /* code similar to above */ }
}
```

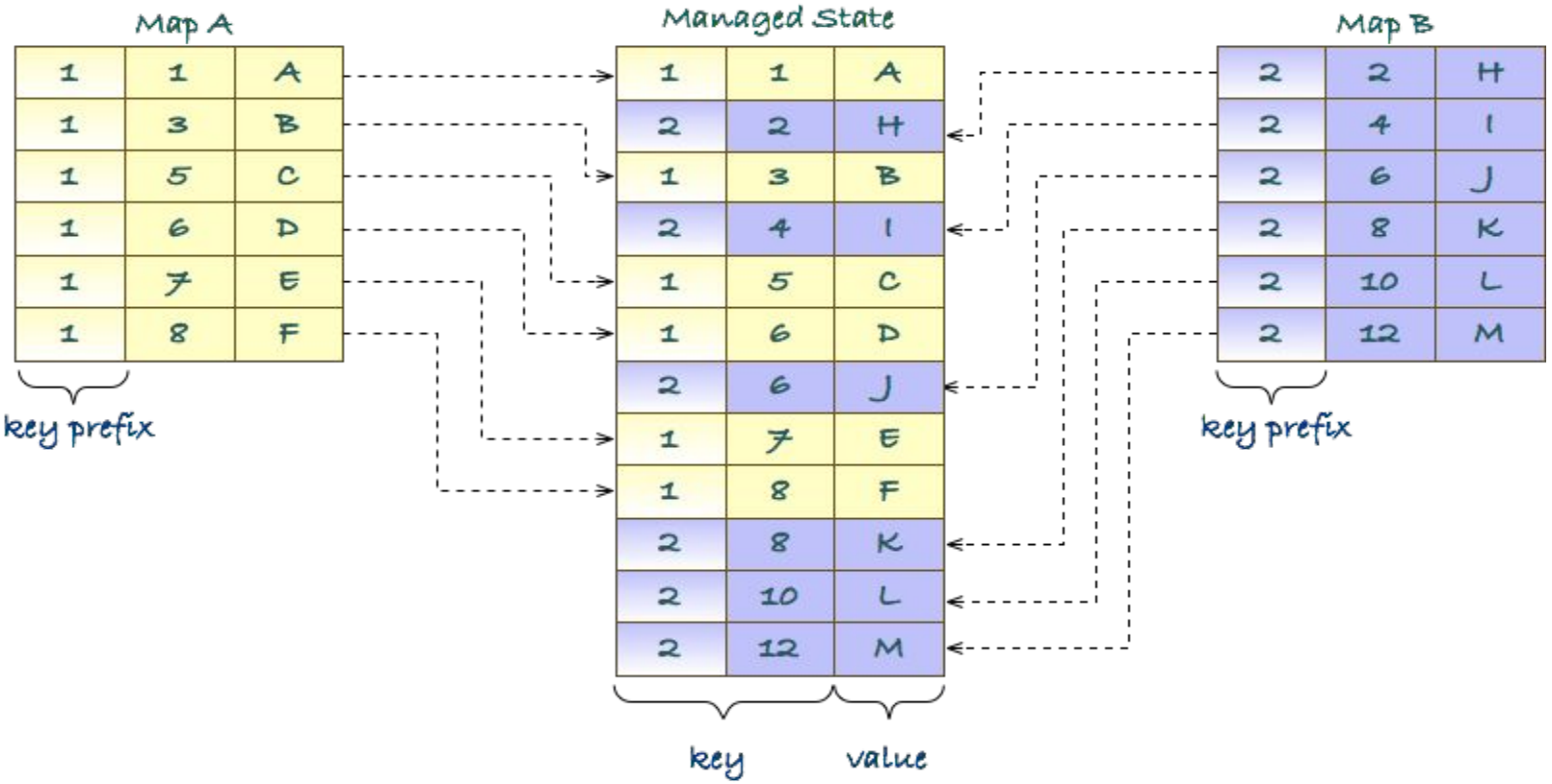
What If I Wanted To Store Multiple Maps?

Key collisions for multiple maps



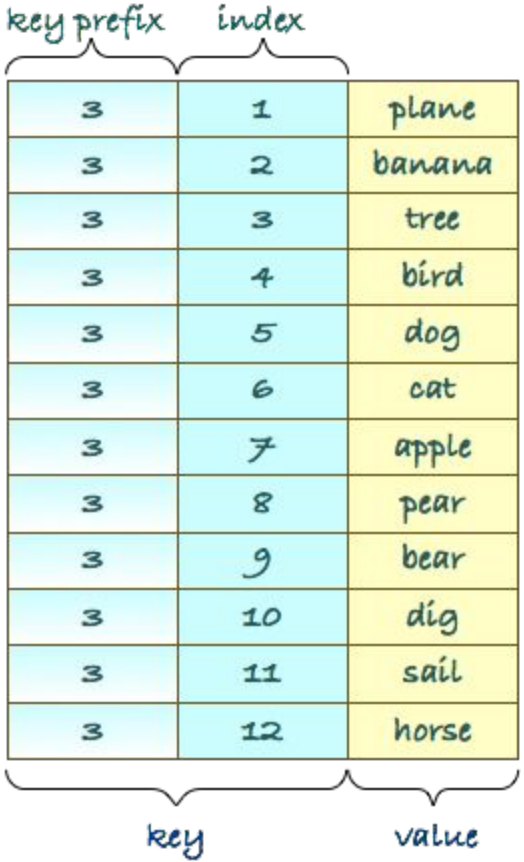
Handling Multiple Maps (And Data-structures)

Keys have a fixed bit-width prefix

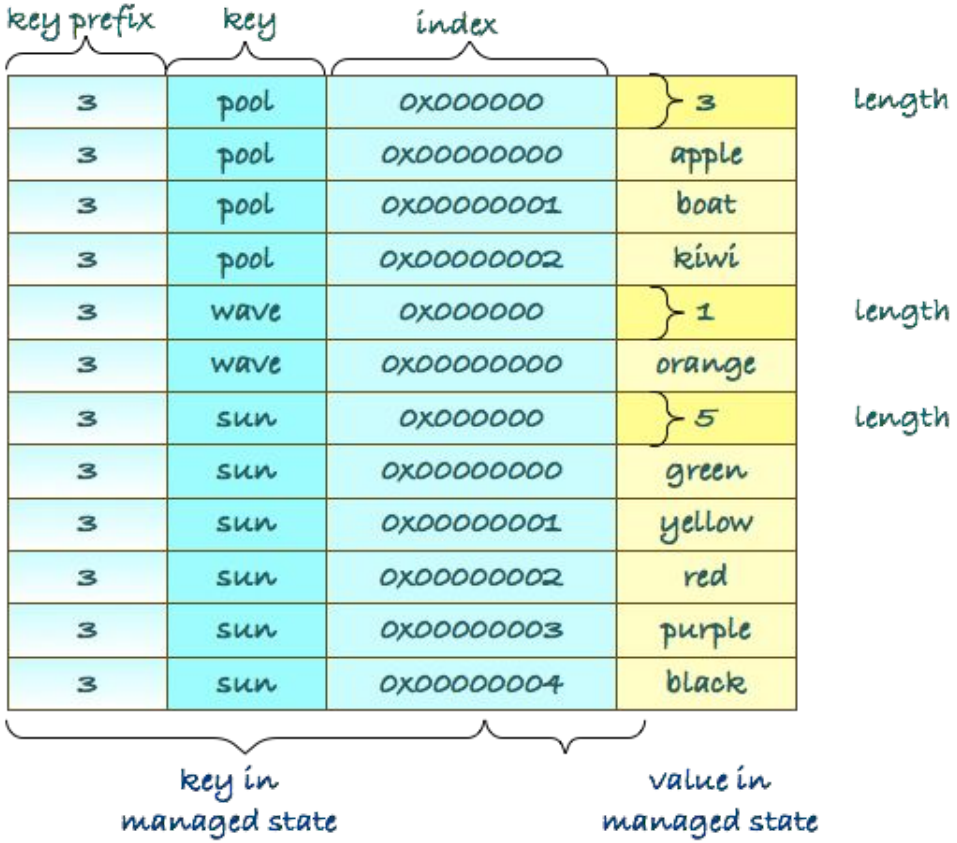


Implementing ArrayLists

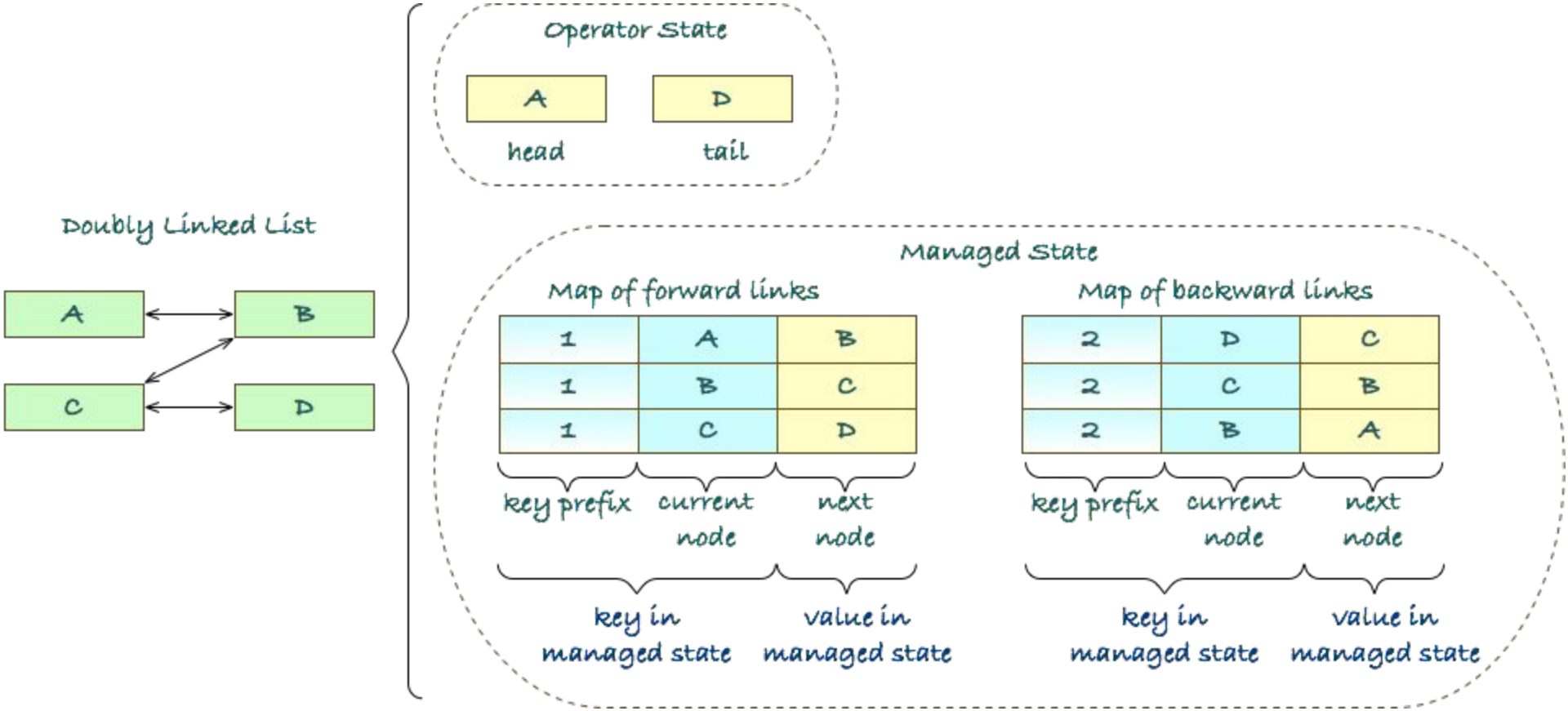
Index keys are 4 bytes wide



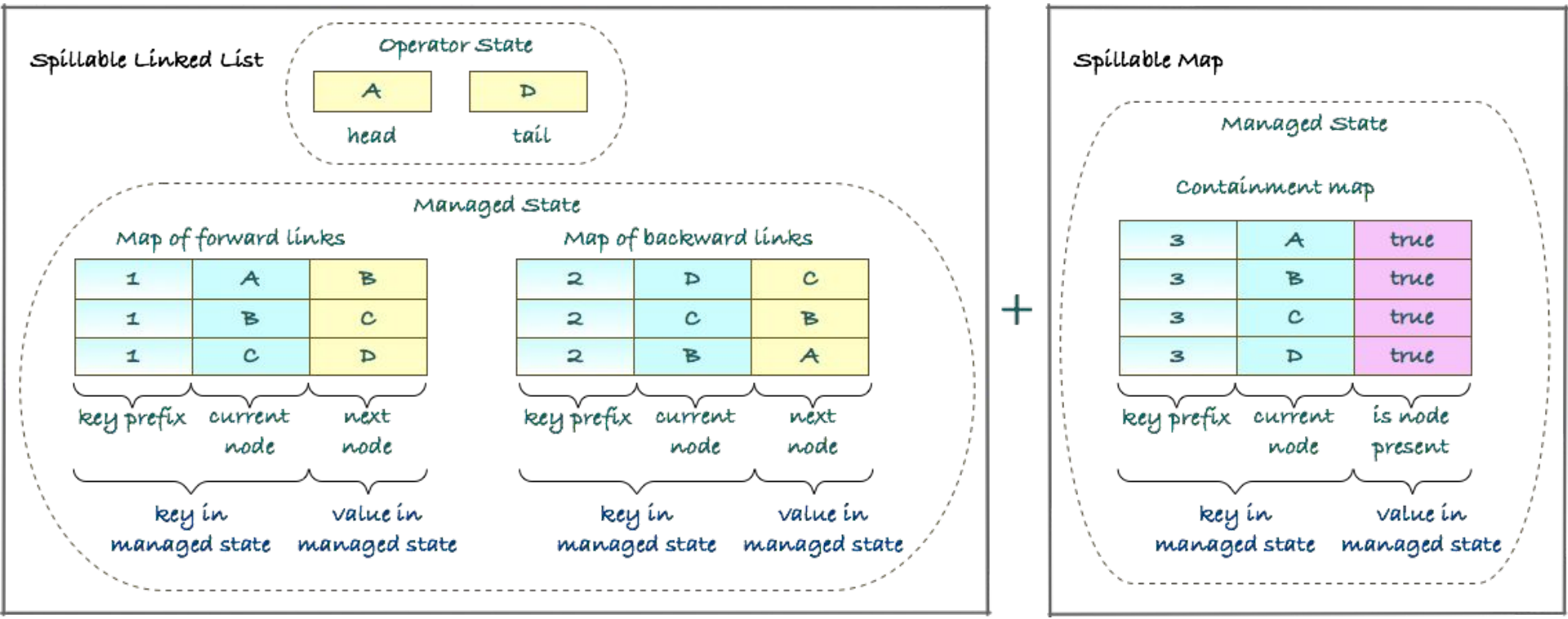
Implementing an ArrayListMultimap



Implementing a Linked List

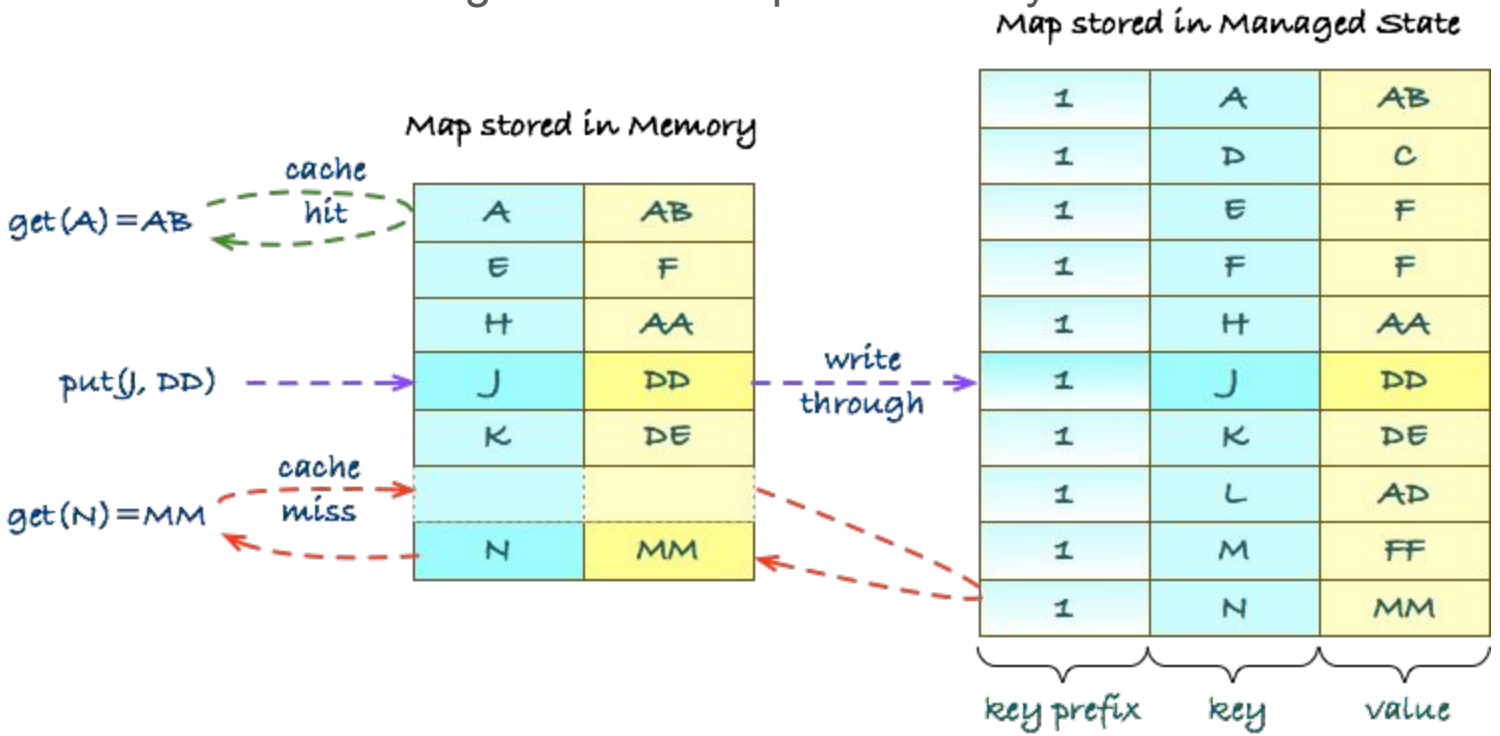


Implementing An Iterable Set



Caching Strategy

Simple write and read through cache is kept in memory.



Implementations For Apache Apex

- **SpillableMap:**https://github.com/apache/apex-malhar/blob/master/library/src/main/java/org/apache/apex/malhar/lib/state/spillable/SpillableMapImpl.java
- **SpillablArrayList:**https://github.com/apache/apex-malhar/blob/master/library/src/main/java/org/apache/apex/malhar/lib/state/spillable/SpillableArrayListImpl.java
- **SpillableArrayListMultimap:**https://github.com/apache/apex-malhar/blob/master/library/src/main/java/org/apache/apex/malhar/lib/state/spillable/SpillableArrayListMultimapImpl.java
- **SpillableSetImpl:**https://github.com/apache/apex-malhar/blob/master/library/src/main/java/org/apache/apex/malhar/lib/state/spillable/SpillableSetImpl.java
- **SpillableFactory:**https://github.com/apache/apex-malhar/blob/master/library/src/main/java/org/apache/apex/malhar/lib/state/spillable/SpillableComplexComponentImpl.java

Spillable Data Structures In Action

We use them at Simplifi.it to run a Data Aggregation Pipeline built on Apache Apex.

Simplifi.it

Questions?