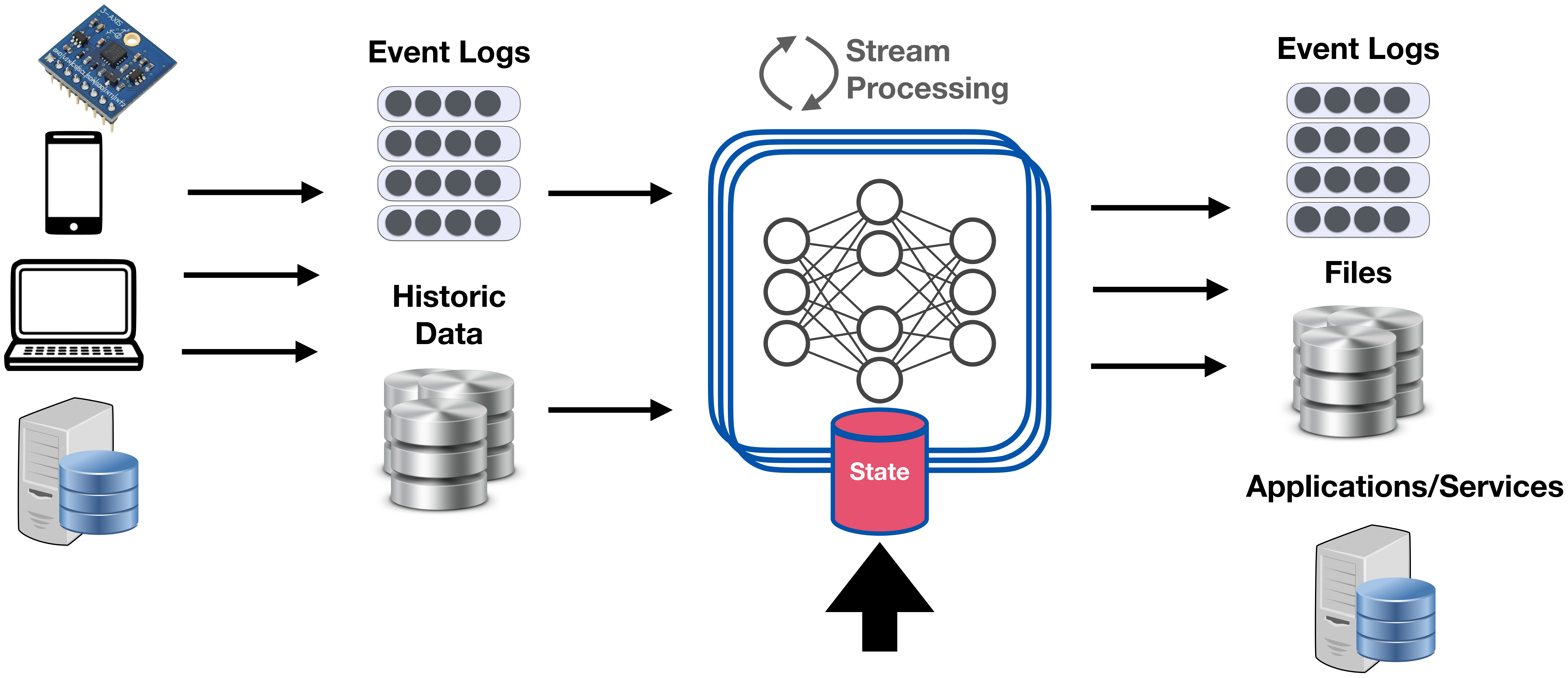
CS 591 K1:
Data Stream Processing and Analytics

Spring 2021

State Management

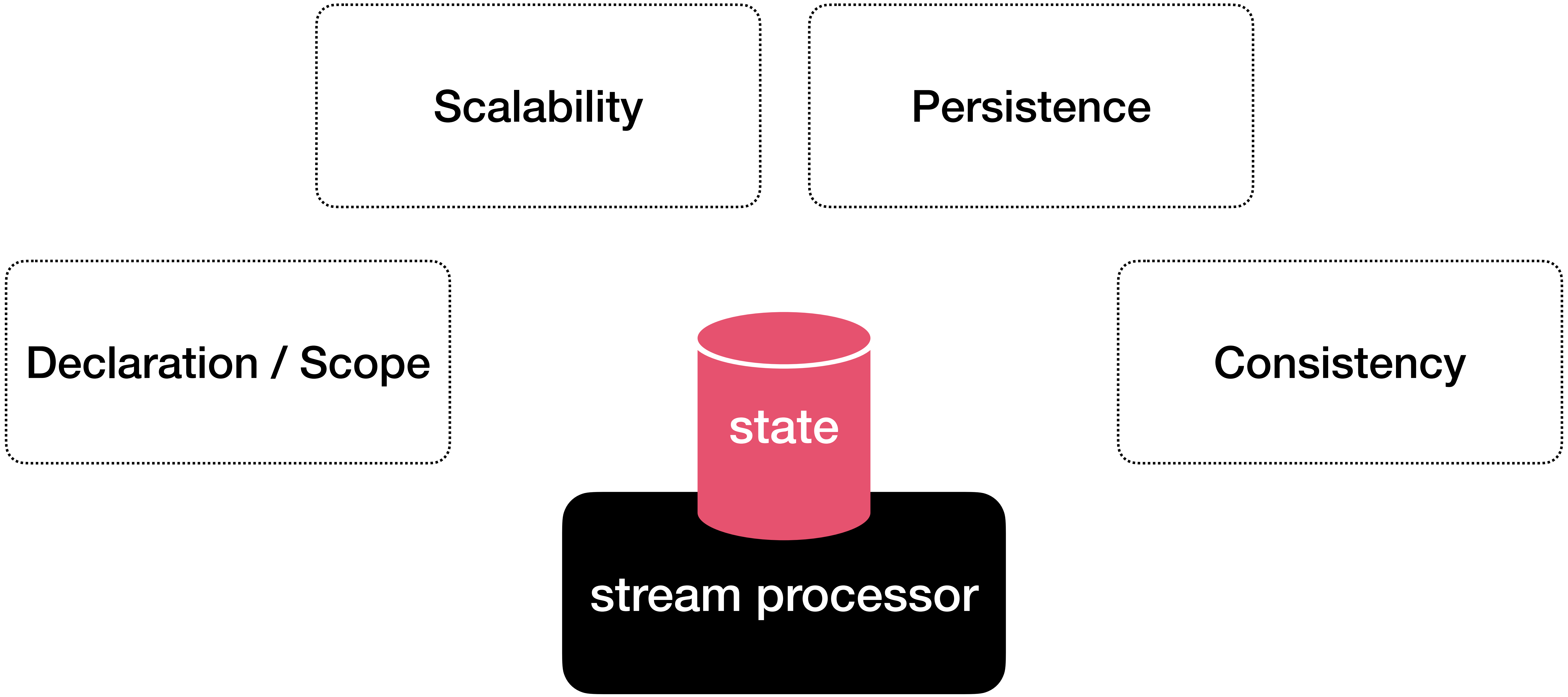
Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

What is State

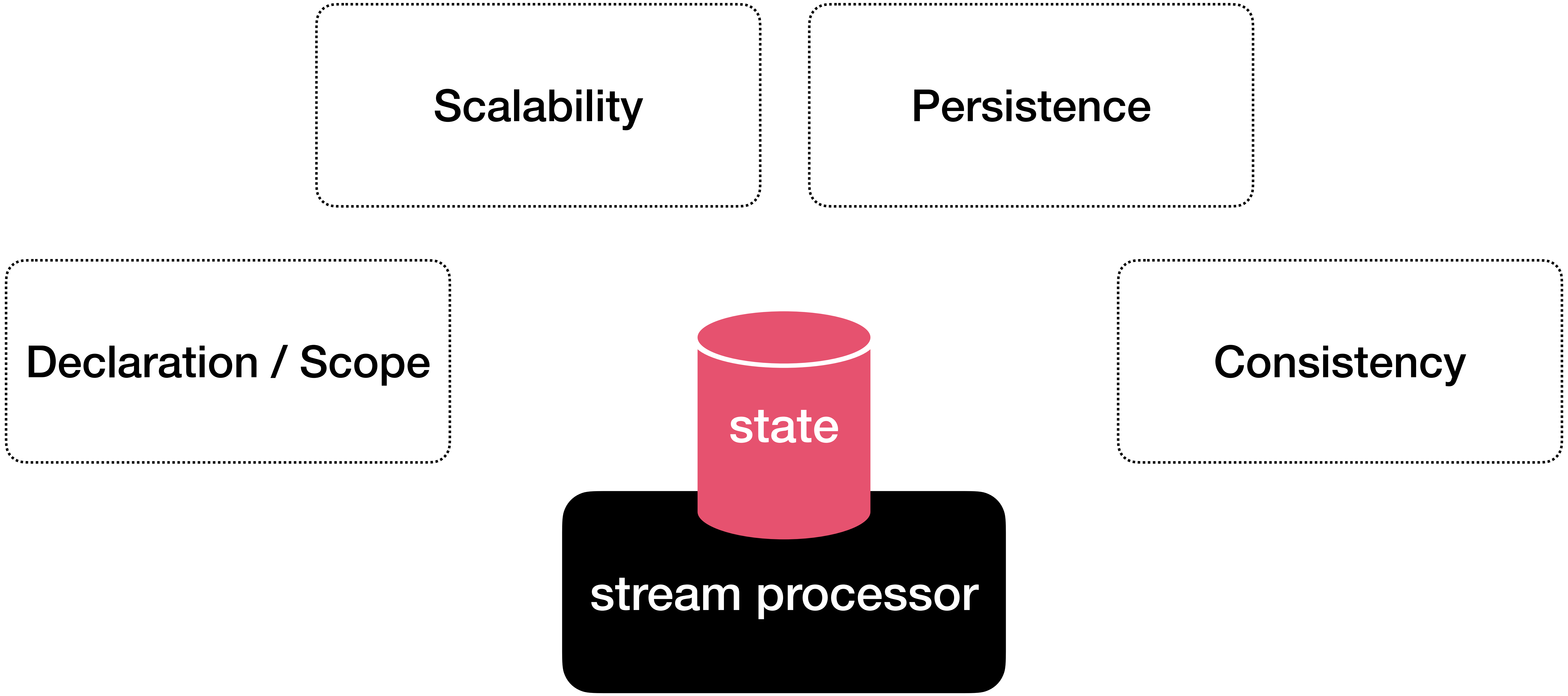


windows
aggregates
user-defined variables

What is Stream State Management

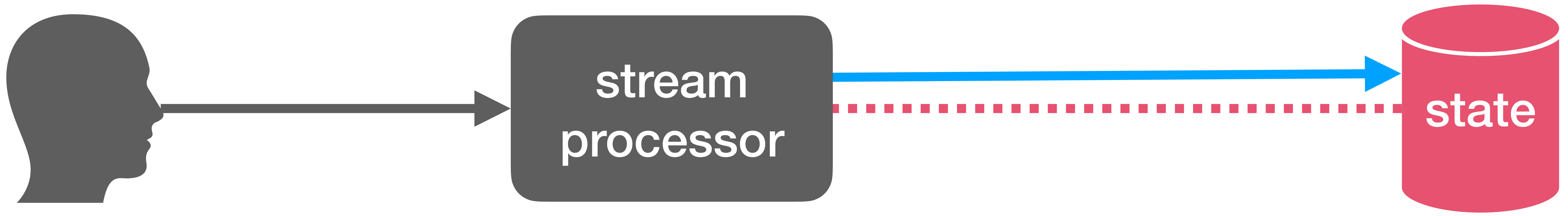


What is Stream State Management



Should the system or the user be responsible for (1) declaring and (2) managing streaming state?

State as a Synopsis



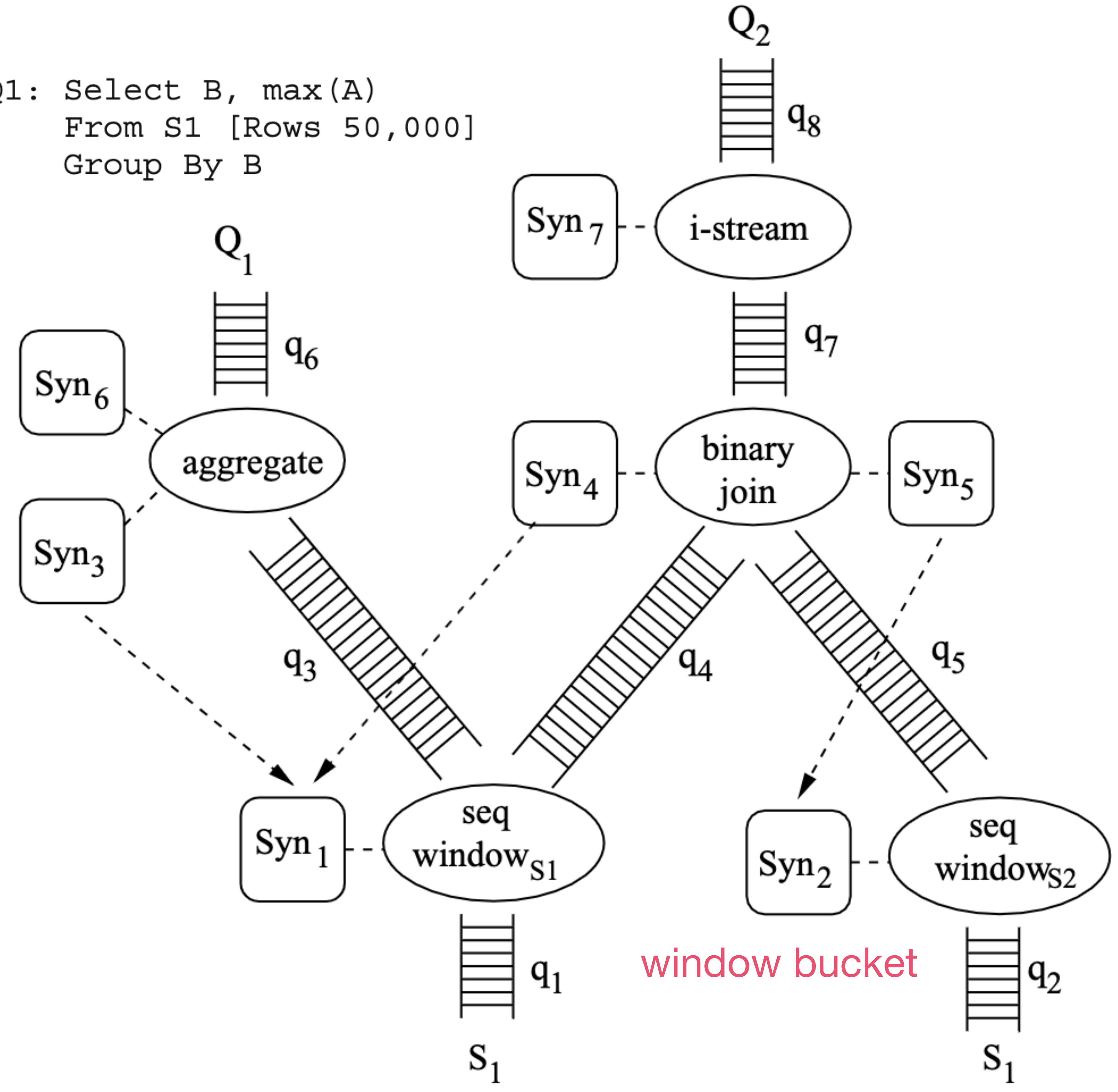
"synopsis" : short system-internal summarization of an input / computation
aka system-defined and system-managed state

- A *implicit* form of State that has been an integral component of DSMSs (2000s)
- It facilitates implementation of a set of **operators** (e.g., join, filter, aggregate, sort, window etc.)
- It is usually hidden from the user (System-Defined)
- Often Transient and Memory-bound - **Approximation**

Examples

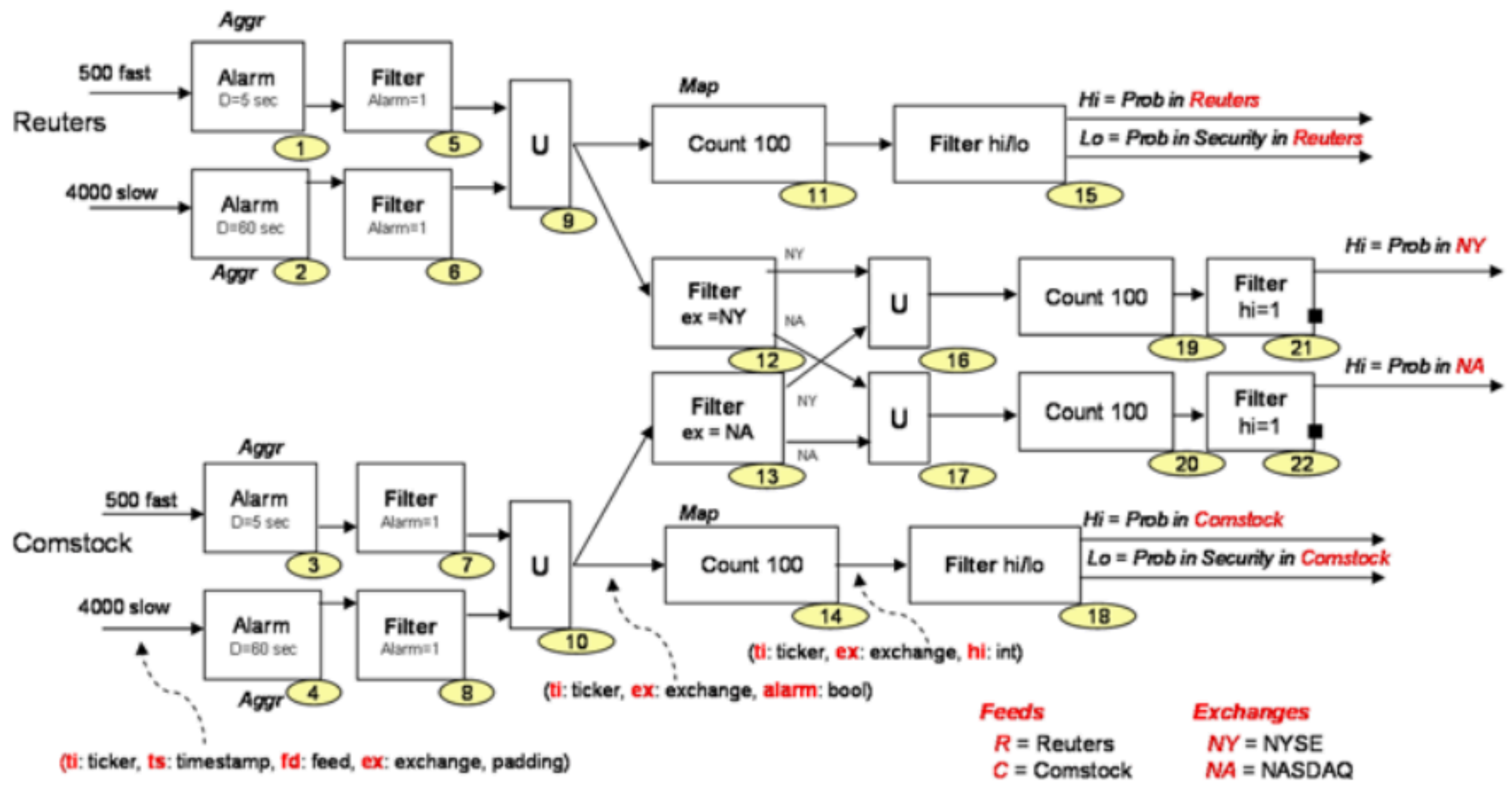
```
Q2: Select Istream(*)
    From   S1 [Rows 40,000],
           S2 [Range 600 Seconds]
    Where  S1.A = S2.A
```

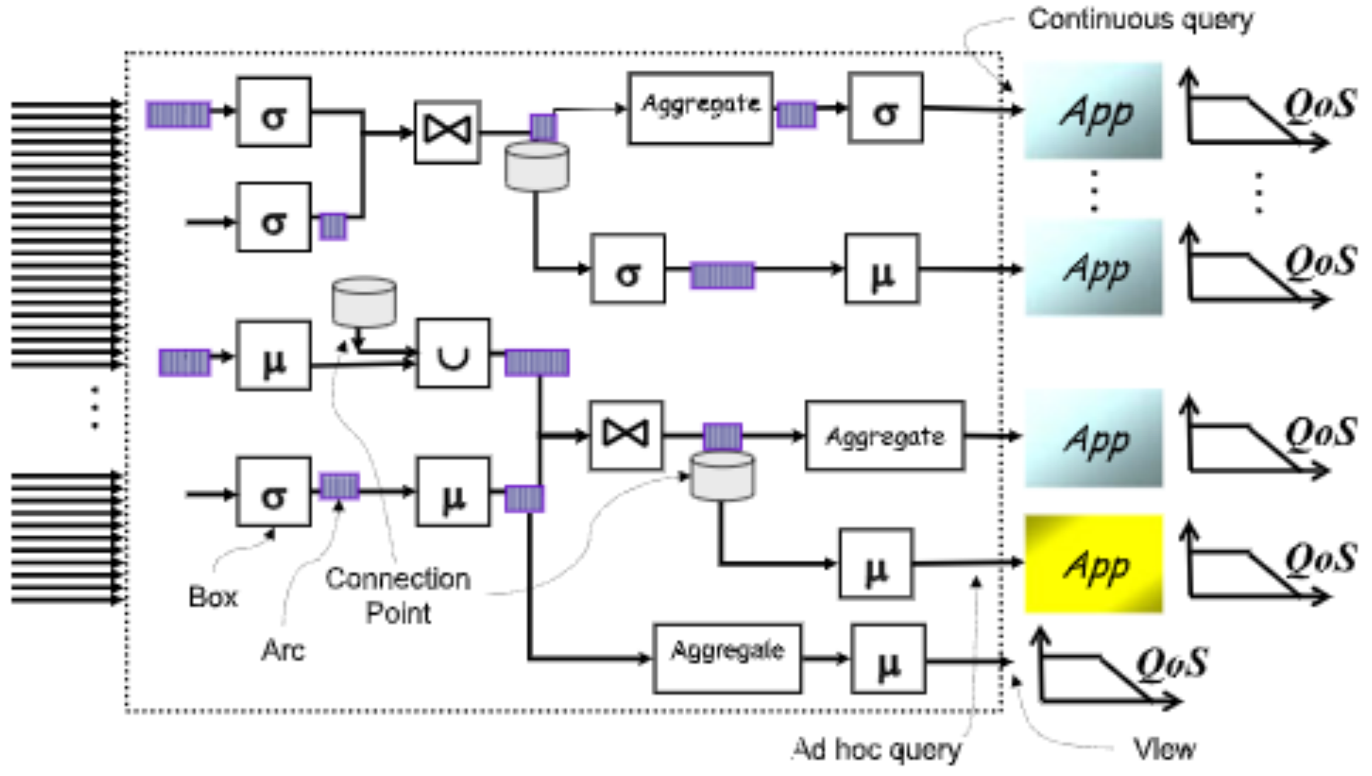
```
Q1: Select B, max(A)
    From S1 [Rows 50,000]
    Group By B
```



The CQL continuous query language: Semantic foundations and query execution
Arasu A, Babu S, Widom J in VLDB Journal (2006)

```
Aggregate(Group by ticker,
          Order on arrival,
          Window (Size = 2 tuples,
                  Step = 1 tuple,
                  Timeout = 5 sec))
```

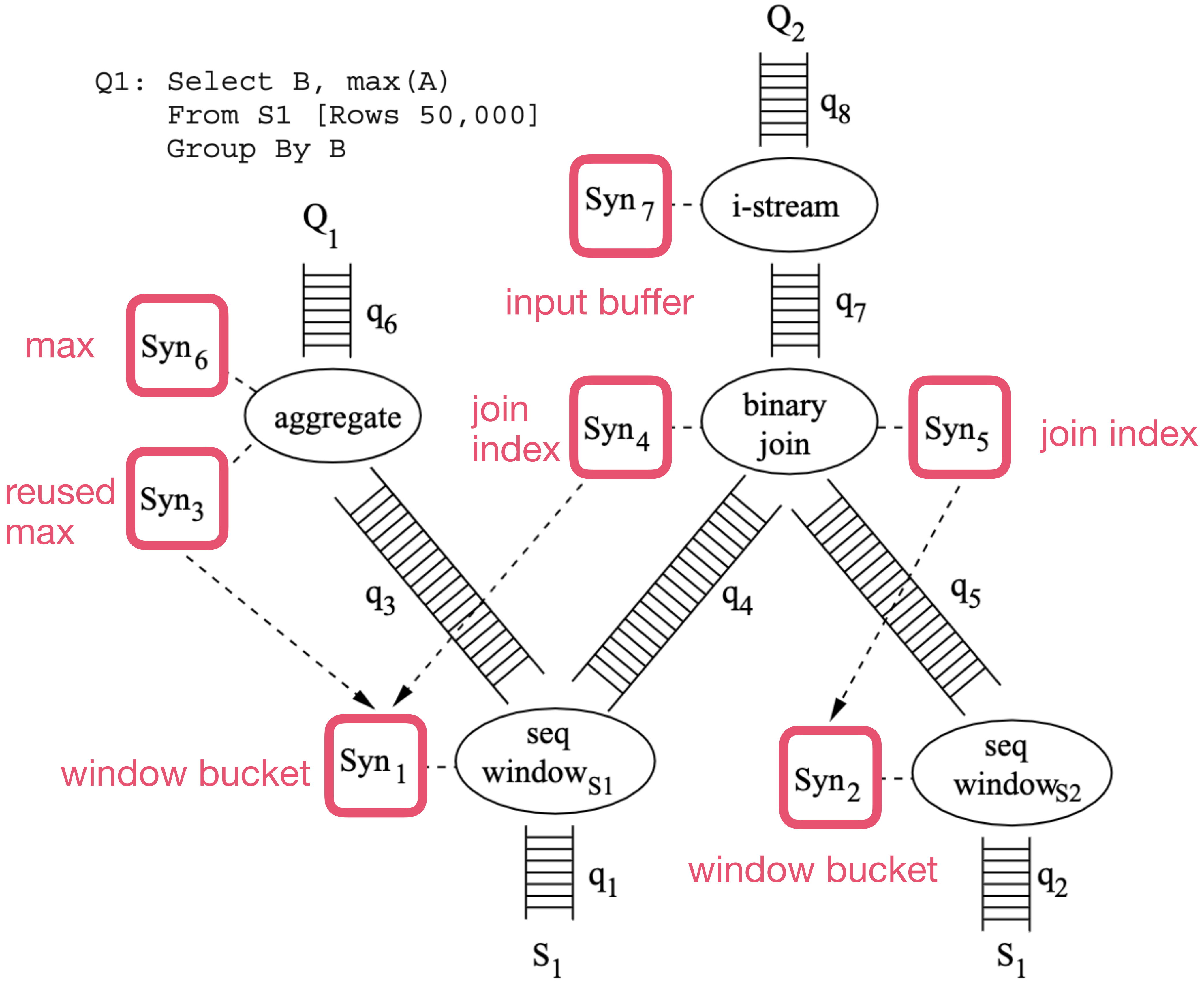




The Aurora and Borealis Stream Processing Engines
Çetintemel U, Abadi D, Ahmad Y, et. al. (2016)

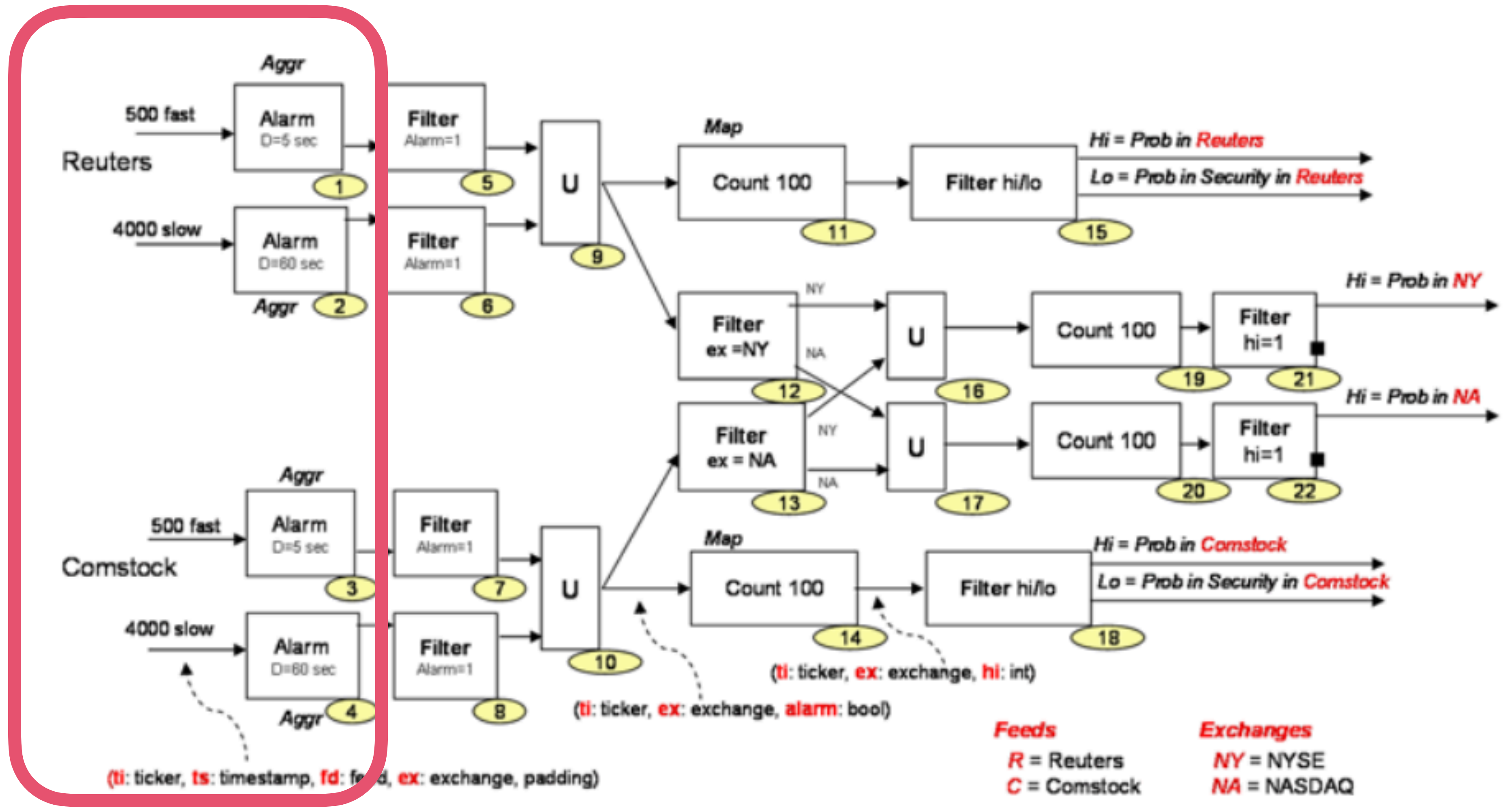
Examples

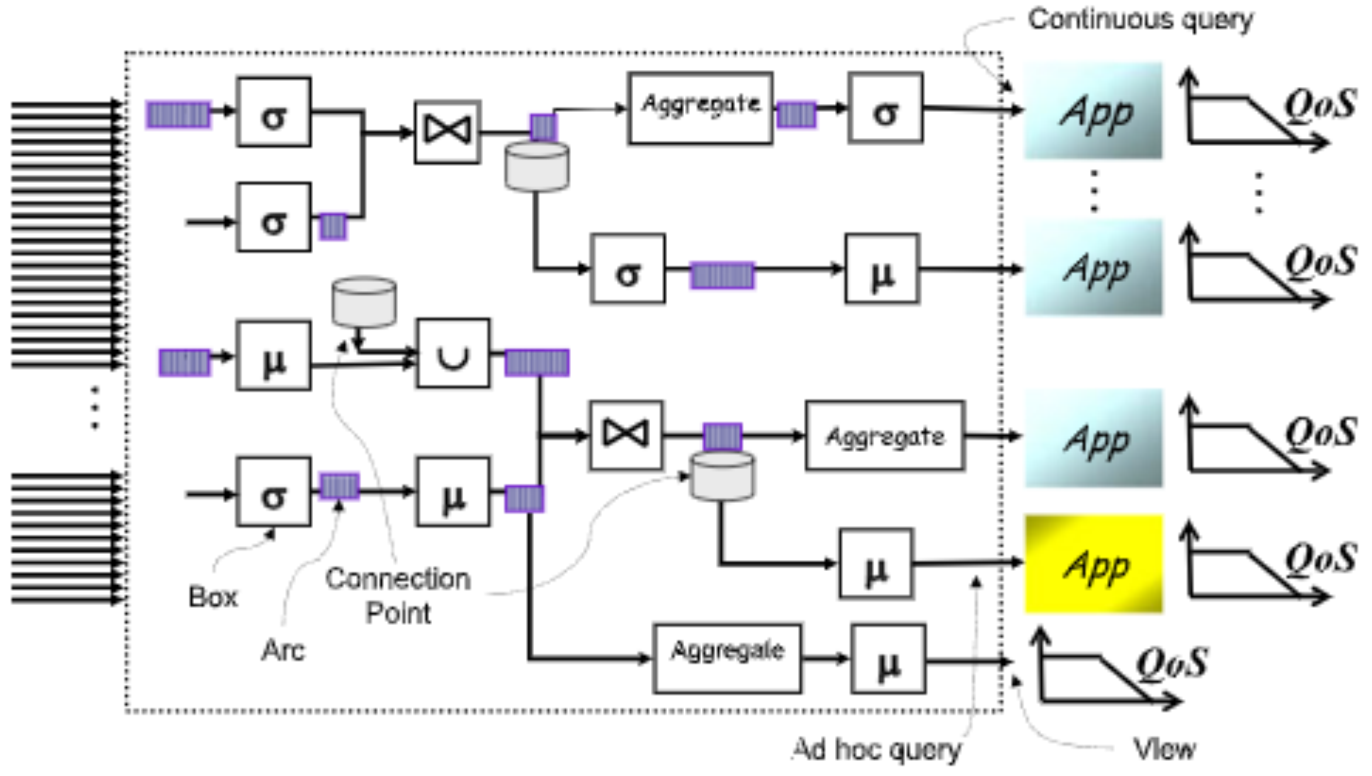
```
Q2: Select Istream(*)
    From   S1 [Rows 40,000],
           S2 [Range 600 Seconds]
    Where  S1.A = S2.A
```



The CQL continuous query language: Semantic foundations and query execution
Arasu A, Babu S, Widom J in VLDB Journal (2006)

```
Aggregate(Group by ticker,
          Order on arrival,
          Window (Size = 2 tuples,
                  Step = 1 tuple,
                  Timeout = 5 sec))
```





The Aurora and Borealis Stream Processing Engines
Çetintemel U, Abadi D, Ahmad Y, et. al. (2016)

Synopses overview

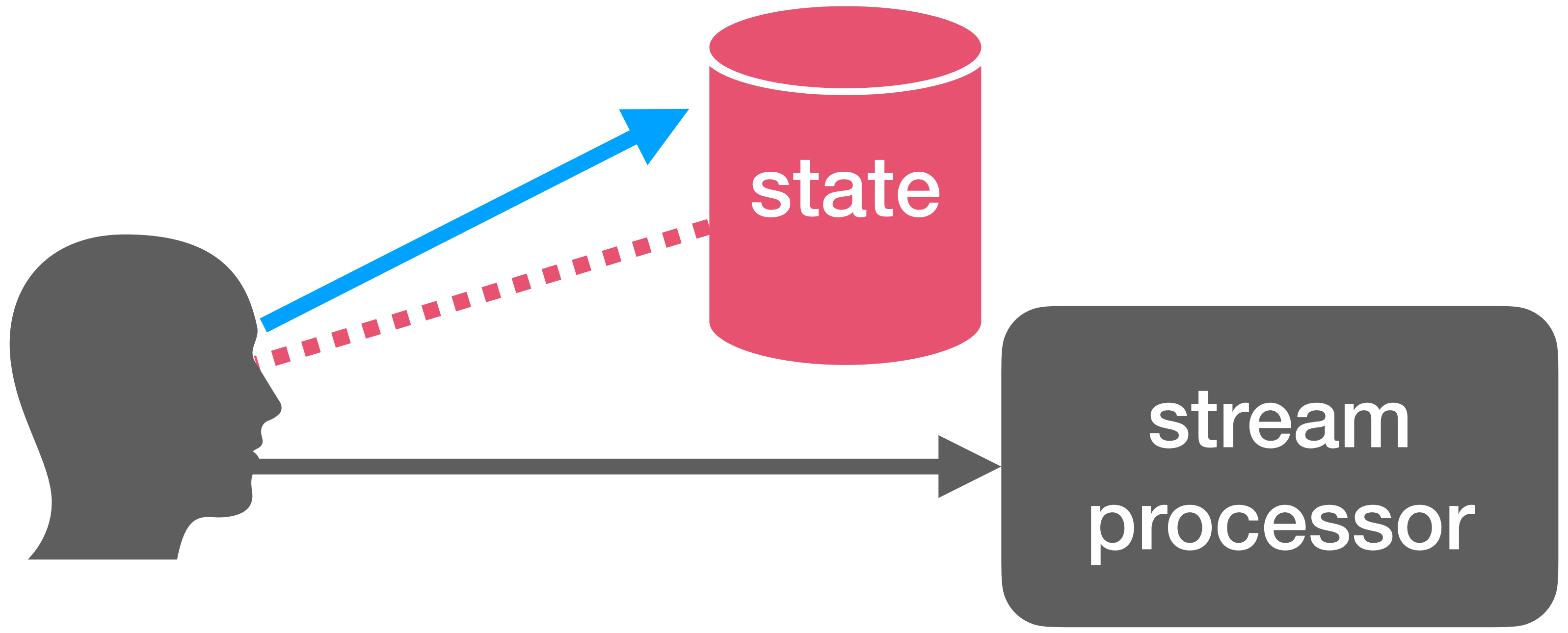
Pros

- System can manage synopses efficiently as internal data structures
- Synopses can be combined and reused across operators internally
- Powerful approach in shared-memory system implementations

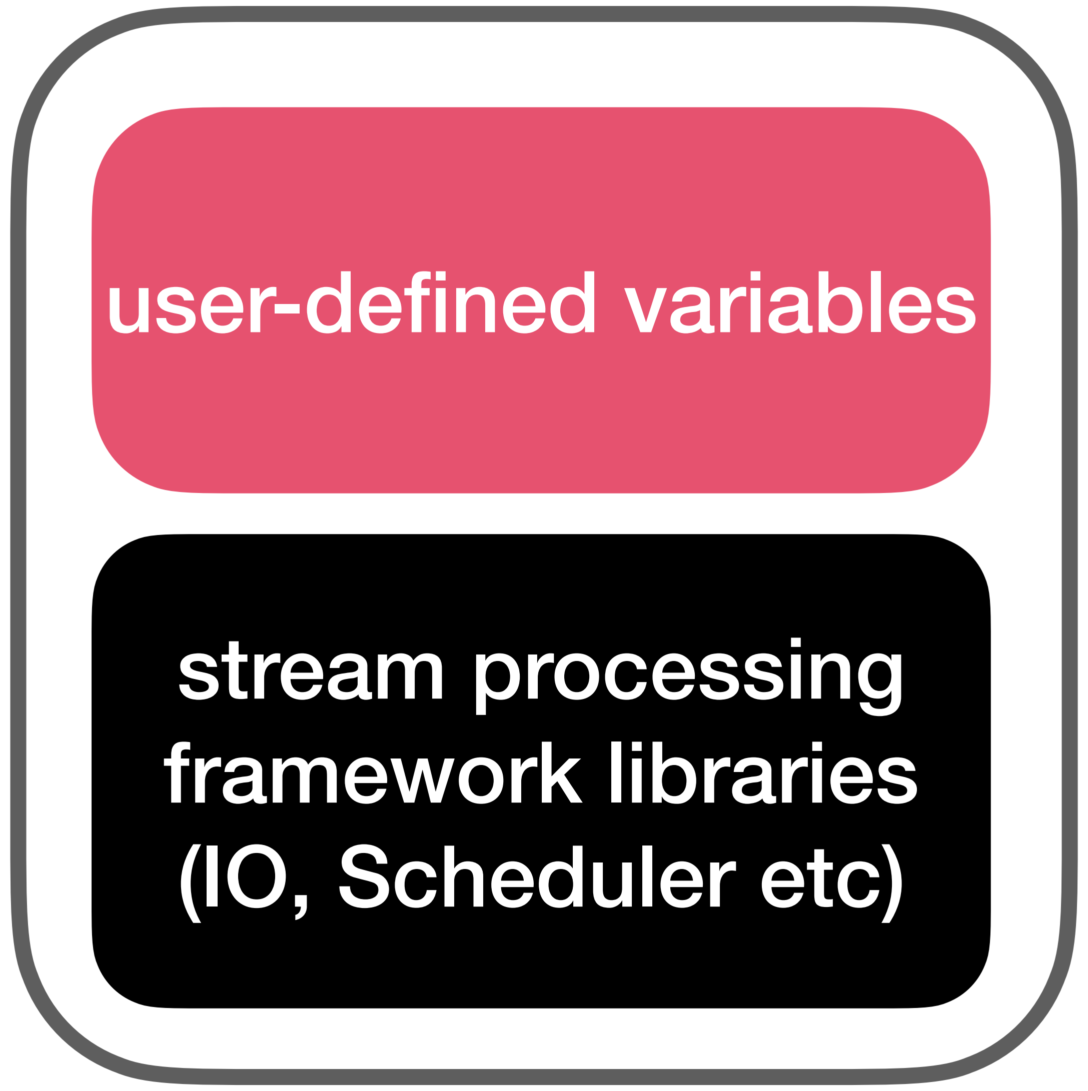
Cons

- Tight coupling of state to limited set of operator semantics (implementation)
- No general, declarative model for underlying user-defined state
- Over-specialization complicates auxiliary operations (fault tolerance, reconfiguration, elasticity etc.)

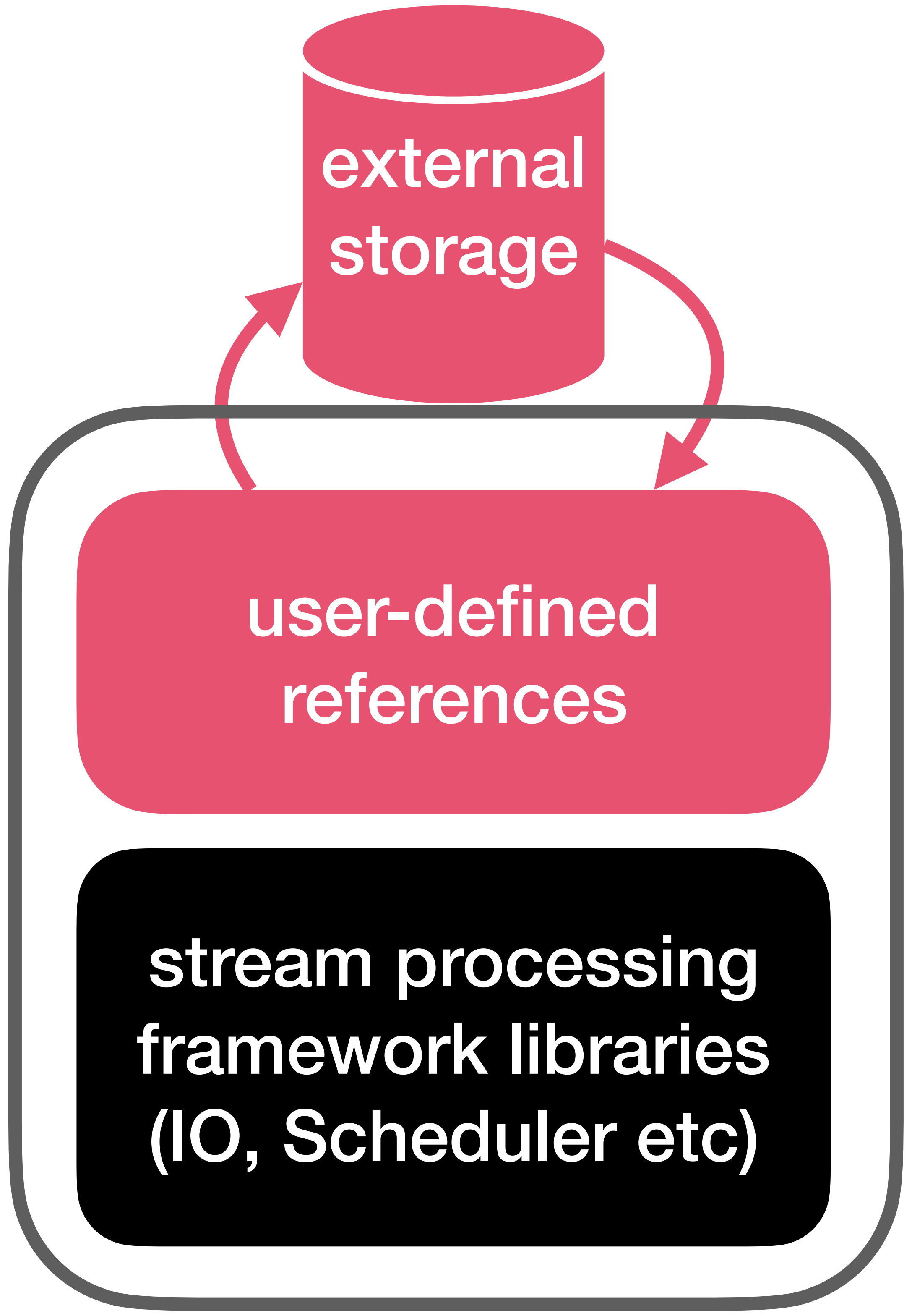
Application-Managed User-Defined State



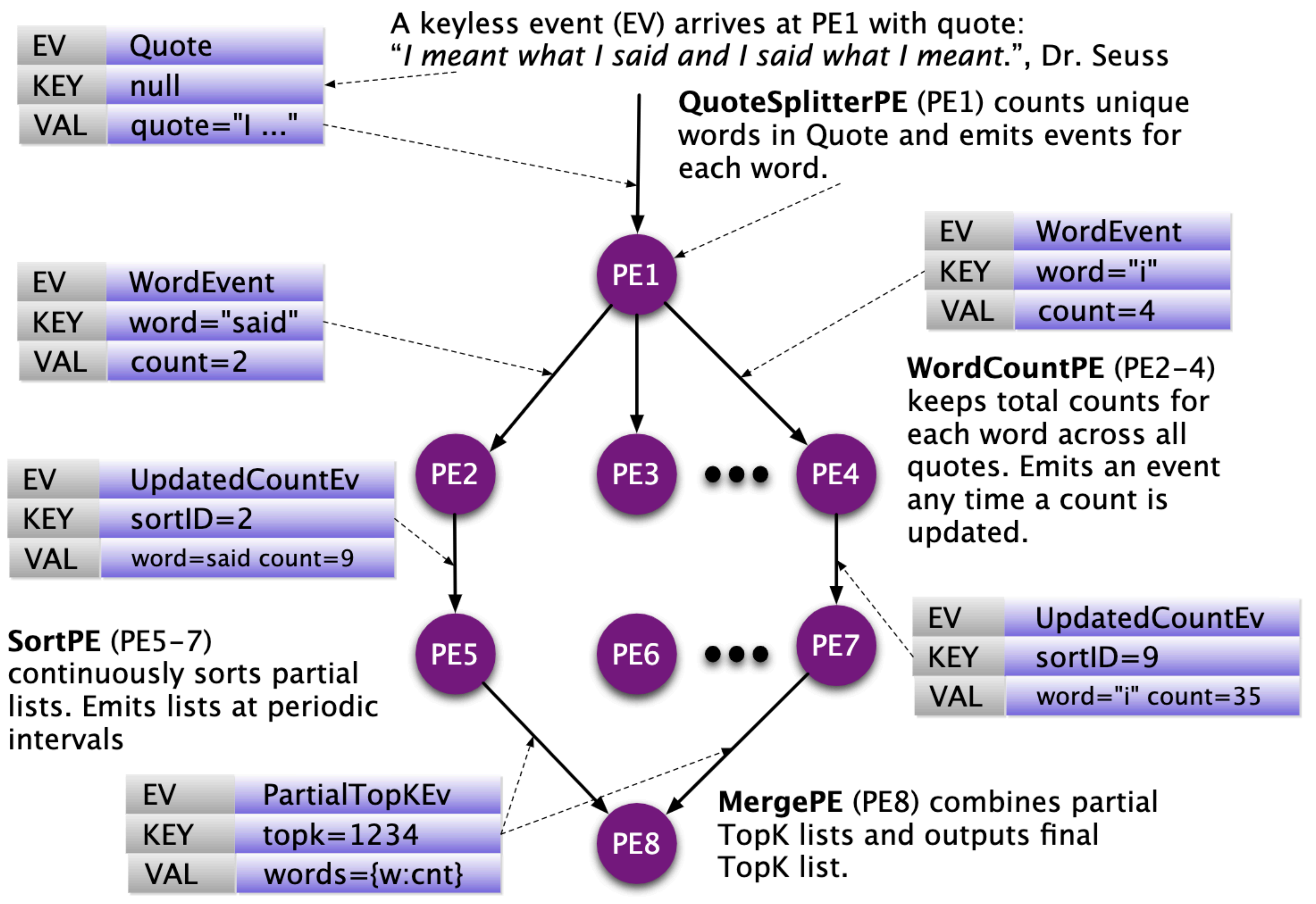
1. Process-Local State



2. External State



Examples

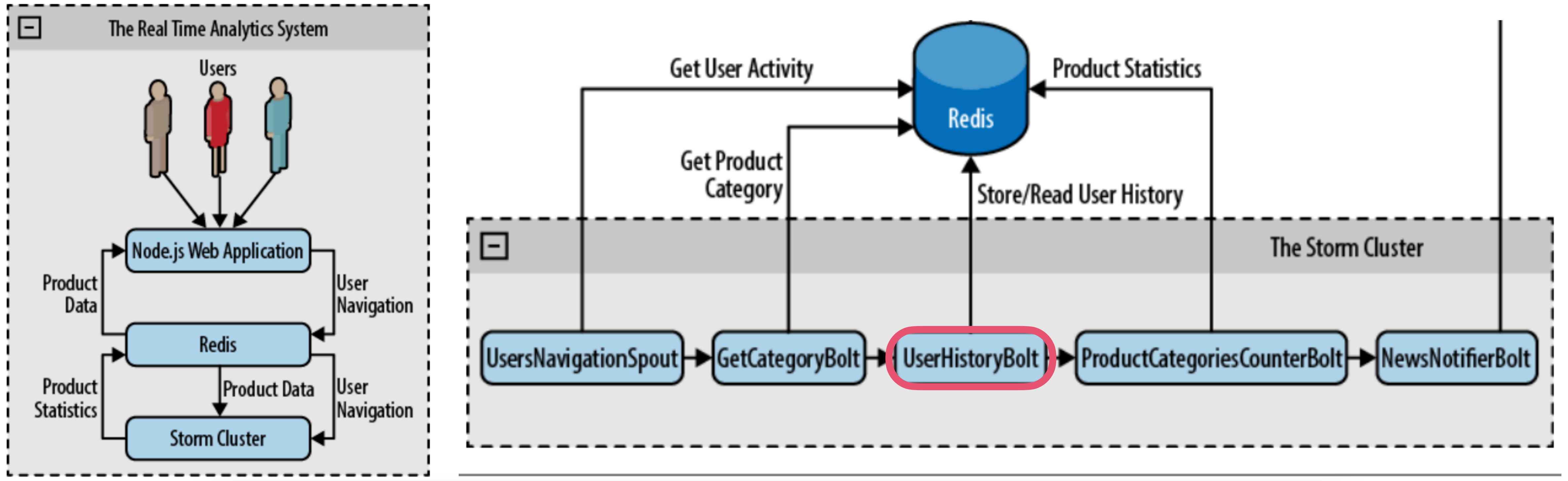



```
private queryCount = 0;

public void processEvent(Event event)
{
  queryCount ++;
}

public void output()
{
  String query = (String) this.getKeyValue().get(0);
  persister.set(query, queryCount);
}
```

S4: Distributed Stream Computing Platform
Neumeyer L, Robbins B, Nair A, et. al. (2010)




```
public class UserHistoryBolt extends BaseRichBolt{

  @Override
  public void execute(Tuple input) {
     String user = input.getString(0);
     String prodKey = input.getString(1)+":"+input.getString(2);
     Set<String> productsNavigated = getHistoryRedis(user);
     if(!productsNavigated.contains(prodKey)) {
        ...
        addHistoryRedis(user, prodKey);
     }
  }
  private Set<String> getHistoryRedis(String user) {
     Set<String> userHistory = redisNavItems.get(user);
     if(userHistory == null) {
        userHistory = jedis.smembers(buildKey(user));
        if(userHistory == null)
           userHistory = new HashSet<String>();
        redisNavItems.put(user, userHistory);
     }
     return userHistory;
  }
  private void addHistoryRedis(String user, String product) {
     Set<String> userHistory = getHistoryRedis(user);
     userHistory.add(product);
     jedis.sadd(buildKey(user), product);
  }
  ...
}
```

Getting Started with Storm
Leibiusky J, Eisbruch G, Simonassi D, et. al. (2012)

Application-managed overview

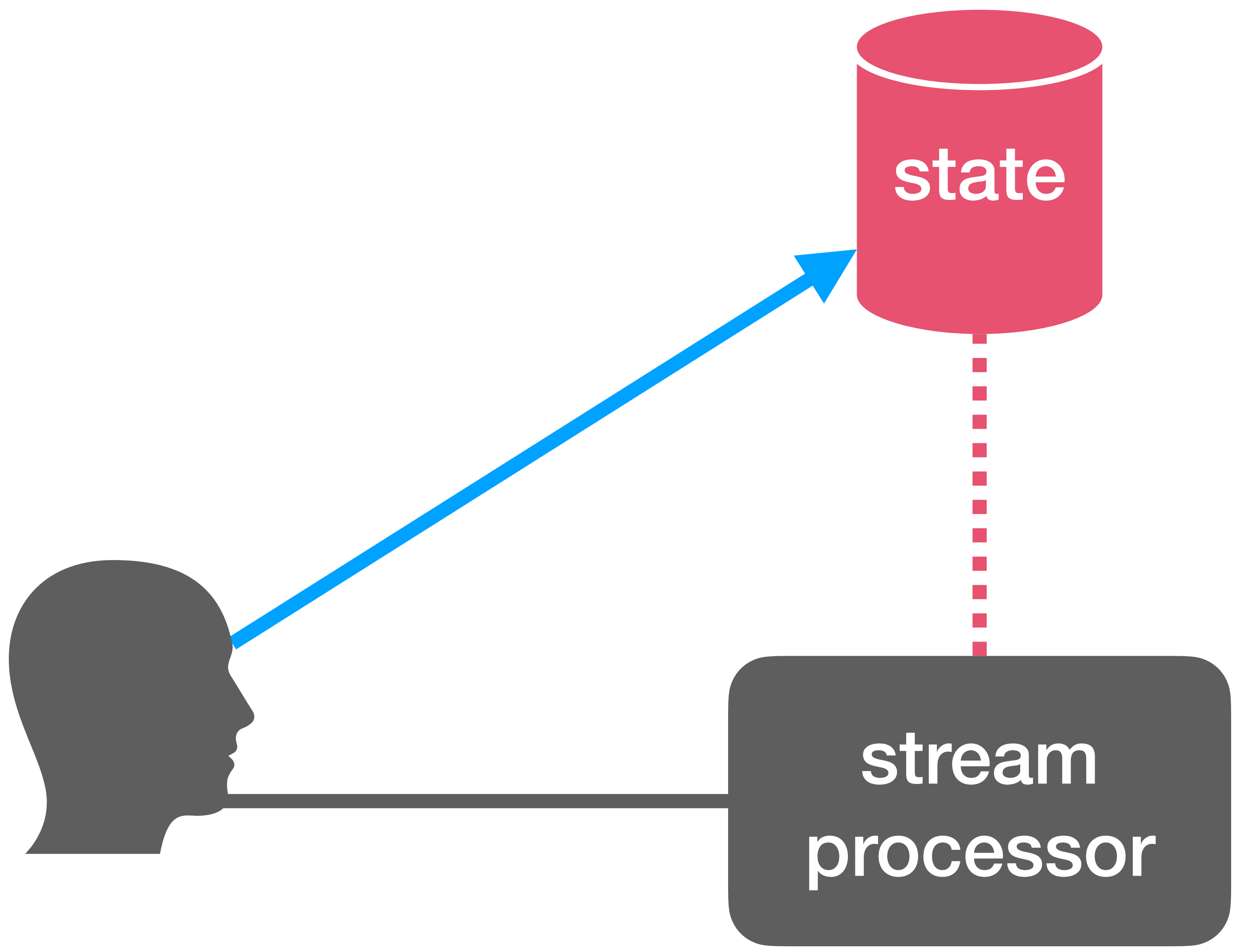
Pros

- Flexibility to use any complex state type (e.g., graphs, matrices, arrays etc.)
- Decoupling of state and computing in application-logic

Cons

- Manual application state management is necessary
- Third-party system dependencies are required for persistence and FT
- Requires deep user expertise on state management
- Missed opportunities for system-level state access optimization

System-Managed User-Defined State



- Provide users with a Stateful Processing API
- The system is aware of state declared using the API and can automatically manage it: checkpoint, repartition, restore, etc.

Examples

```
// the source data stream
val stream: DataStream[…] = ...
val result: DataStream[…] = stream
  .keyBy(0)
  .process(new MyCustomLogic())


class MyCustomLogic extends KeyedProcessFunction[…] {

  /** The state that is maintained by this process function */
  lazy val state: ValueState[…] = getRuntimeContext
    .getState(new ValueStateDescriptor[…](“myState”, classOf[…]))


  override def processElement(element: …)
    …
    state.update(…)
    …
    ctx.timerService.registerEventTimeTimer(…)
    …
  }

  override def onTimer( timestamp: Long,  StreamContext, TimerContext,
out: …): Unit = {

    state.value match {
      case foo => out.collect((key, count))
      case _ =>
    }
  }
}
```

https://flink.apache.org/

```
class IndexAssigningStatefulDoFn(DoFn):
  INDEX_STATE = CombiningStateSpec('index', sum)

  def process(self, element, index=DoFn.StateParam(INDEX_STATE)):
    unused_key, value = element
    current_index = index.read()
    yield (value, current_index)
    index.add(1)
```

```
# Events is a collection of (user, event) pairs.
events = (p | ReadFromEventSource() | beam.WindowInto(....))

user_models = beam.pvalue.AsDict(
                  events
                  | beam.core.CombinePerKey(ModelFromEventsFn()))

def event_prediction(user_event, models):
  user = user_event[0]
  event = user_event[1]

  # Retrieve the model calculated for this user
  model = models[user]

  return (user, model.prediction(event))

# Predictions is a collection of (user, prediction) pairs.
predictions = events | beam.Map(event_prediction, user_models)
```

https://beam.apache.org/blog/stateful-processing/

Examples

```
// the source data stream
val stream: DataStream[…] = ...
val result: DataStream[…] = stream
  .keyBy(0)
  .process(new MyCustomLogic())


class MyCustomLogic extends KeyedProcessFunction[…] {

 /** The state that is maintained by this process function */
 lazy val state: ValueState[…] = getRuntimeContext
   .getState(new ValueStateDescriptor[…](“myState”, classOf[…]))

 override def processElement(element: …)
   …
   state.update(…)
   …
   ctx.timerService.registerEventTimeTimer(…)
   …
 }

 override def onTimer( timestamp: Long,  StreamContext, TimerContext,
out: …): Unit = {

   state.value match {
     case foo => out.collect((key, count))
     case _ =>
   }
 }
}
```

https://flink.apache.org/

```
class IndexAssigningStatefulDoFn(DoFn):
  INDEX_STATE = CombiningStateSpec('index', sum)

  def process(self, element, index=DoFn.StateParam(INDEX_STATE)):
    unused_key, value = element
    current_index = index.read()
    yield (value, current_index)
    index.add(1)
```

```
# Events is a collection of (user, event) pairs.
events = (p | ReadFromEventSource() | beam.WindowInto(....))

user_models = beam.pvalue.AsDict(
                  events
                  | beam.core.CombinePerKey(ModelFromEventsFn()))

def event_prediction(user_event, models):
  user = user_event[0]
  event = user_event[1]

  # Retrieve the model calculated for this user
  model = models[user]

  return (user, model.prediction(event))

# Predictions is a collection of (user, prediction) pairs.
predictions = events | beam.Map(event_prediction, user_models)
```

https://beam.apache.org/blog/stateful-processing/

Examples

```
// the source data stream
val stream: DataStream[…] = ...
val result: DataStream[…] = stream
  .keyBy(0)
  .process(new MyCustomLogic())


class MyCustomLogic extends KeyedProcessFunction[…] {

  /** The state that is maintained by this process function */
  lazy val state: ValueState[…] = getRuntimeContext
    .getState(new ValueStateDescriptor[…](“myState”, classOf[…]))

  override def processElement(element: …)
    …
    state.update(…)
    …
    ctx.timerService.registerEventTimeTimer(…)
    …
  }

  override def onTimer( timestamp: Long,  StreamContext, TimerContext,
out: …): Unit = {

    state.value match {
      case foo => out.collect((key, count))
      case _ =>
    }
  }
}
```

https://flink.apache.org/

```
class IndexAssigningStatefulDoFn(DoFn):
  INDEX_STATE = CombiningStateSpec('index', sum)

  def process(self, element, index=DoFn.StateParam(INDEX_STATE)):
    unused_key, value = element
    current_index = index.read()
    yield (value, current_index)
    index.add(1)
```

```
# Events is a collection of (user, event) pairs.
events = (p | ReadFromEventSource() | beam.WindowInto(....))

user_models = beam.pvalue.AsDict(
                  events
                  | beam.core.CombinePerKey(ModelFromEventsFn()))

def event_prediction(user_event, models):
  user = user_event[0]
  event = user_event[1]

  # Retrieve the model calculated for this user
  model = models[user]

  return (user, model.prediction(event))

# Predictions is a collection of (user, prediction) pairs.
predictions = events | beam.Map(event_prediction, user_models)
```

https://beam.apache.org/blog/stateful-processing/

System-managed User-defines overview

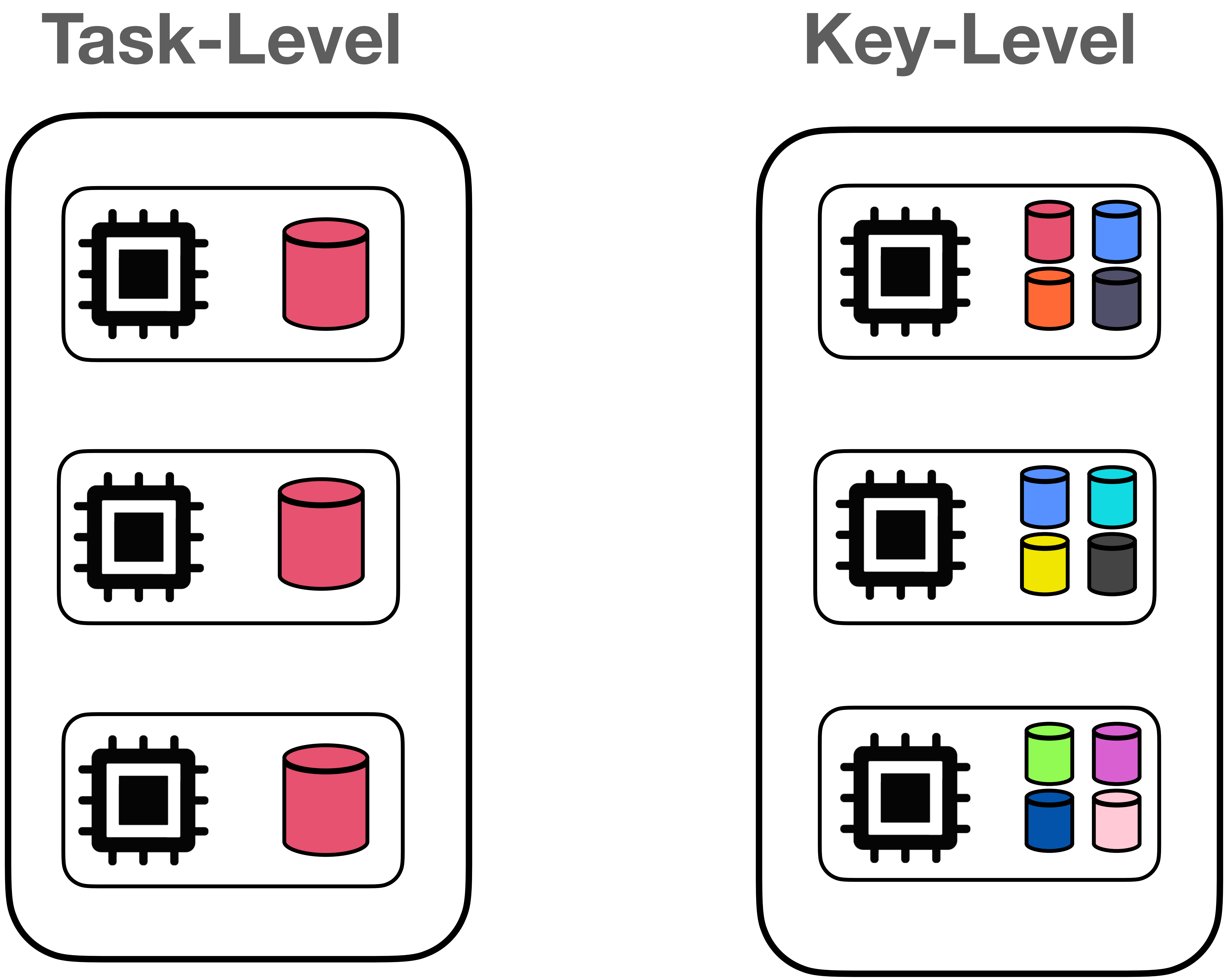
Pros

- Stream Processor can scale, persist, reconfigure and make state durable.
- No external data storage systems to be maintained by the user.

Cons

- Restrictions on supported state types (e.g., Flink's value, append-list, map).
- Limited set of user-level state optimizations.

Partitioned State



Task-Level State Examples

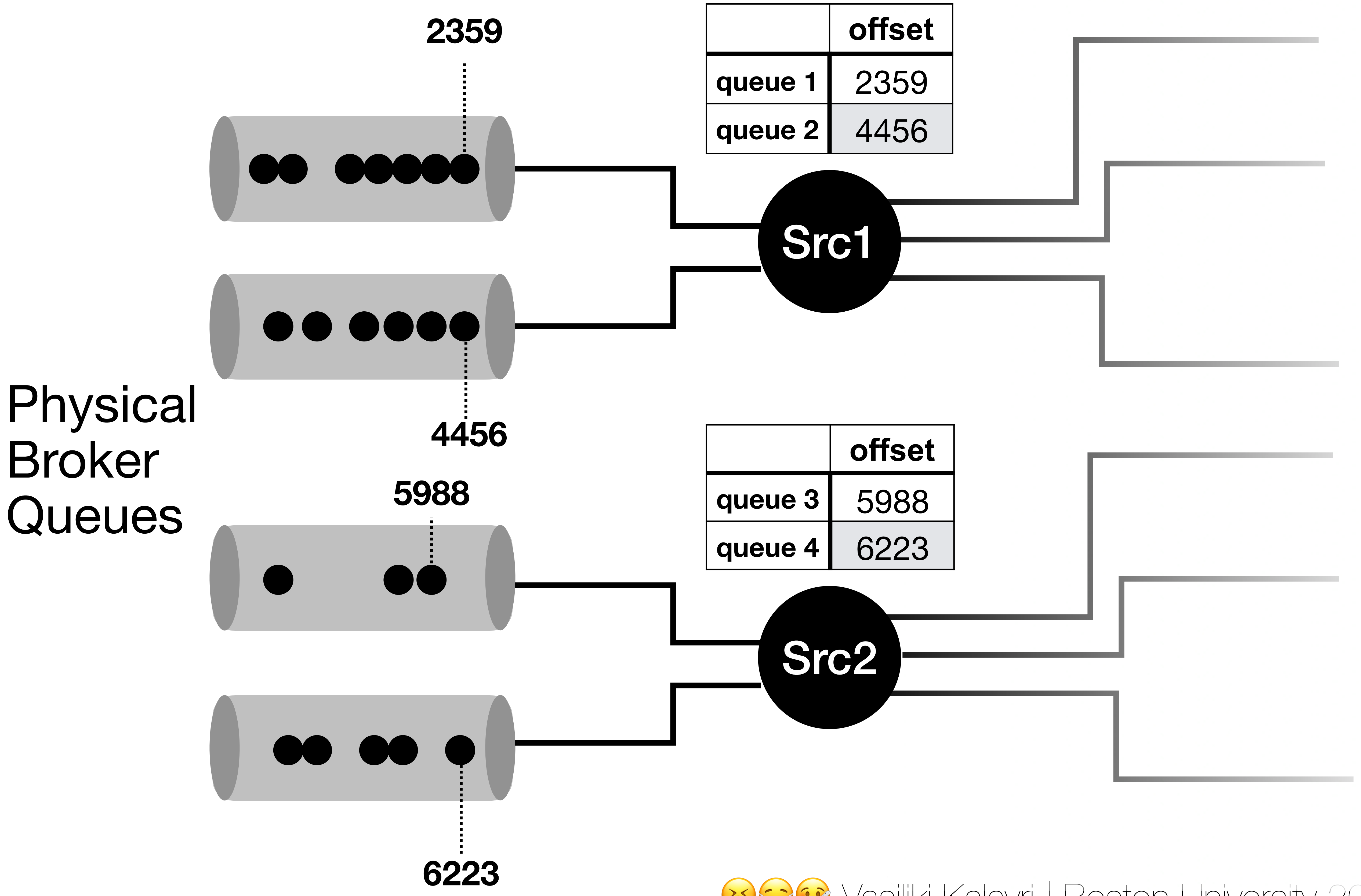
topK Window Counts in Samza

```
skillTags.merge(jobTags).map(MyCounter)
     .window(10, TopKFinder).sendto(topTags);

  class MyCounter implements Map<In, Out>{
  //state definition
   Store<String, int> counts = new Store();
   public Out apply (In msg){
      int cur = counts.get(msg.id) + 1;
      counts.put(msg.id, cur);
      return new Out(msg.id, cur)
   }
```

Samza: Stateful Scalable Stream Processing at LinkedIn
Noghabi S, Paramasivam K, Pan Y, et. al. in Proc. VLDB Endow. (2017)

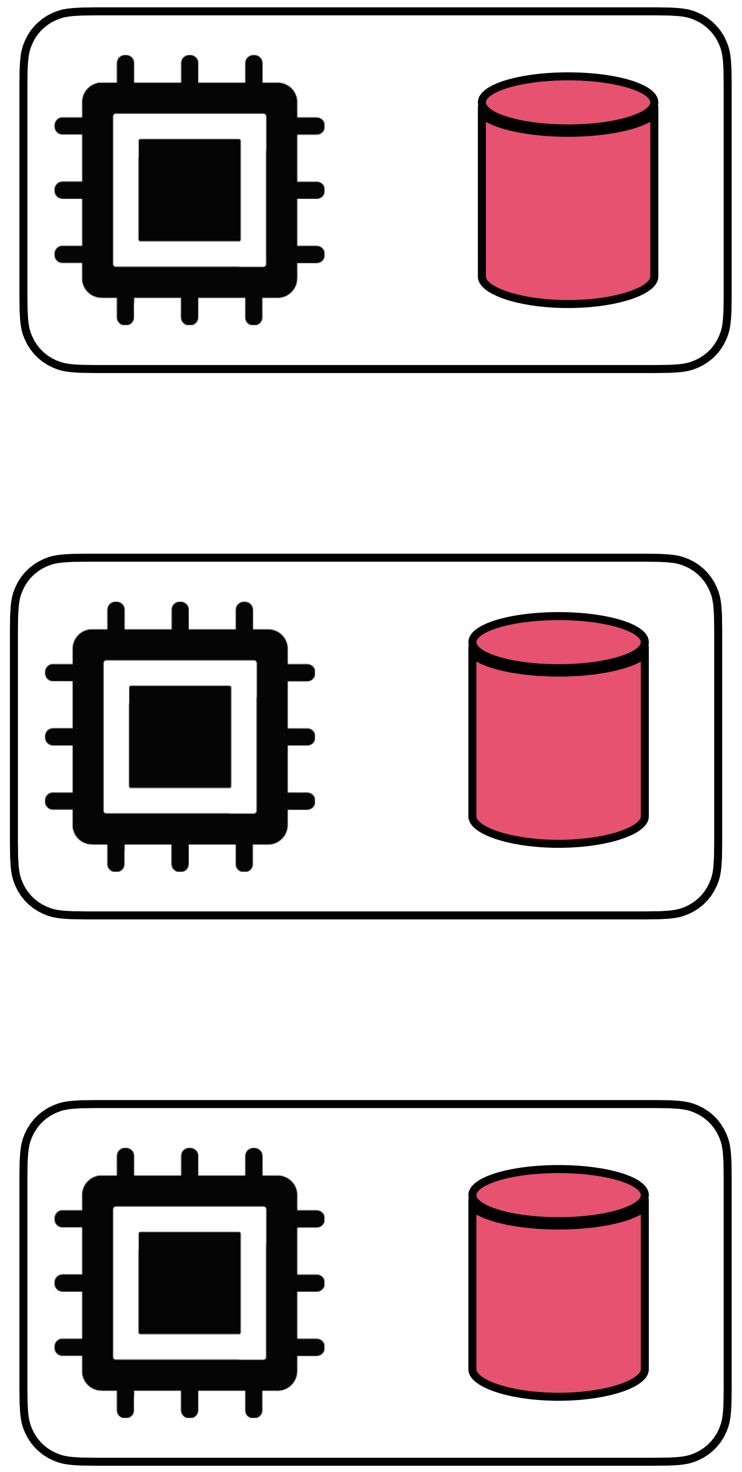
Kafka Source (Flink OperatorState)



	offset
queue 1	2359
queue 2	4456

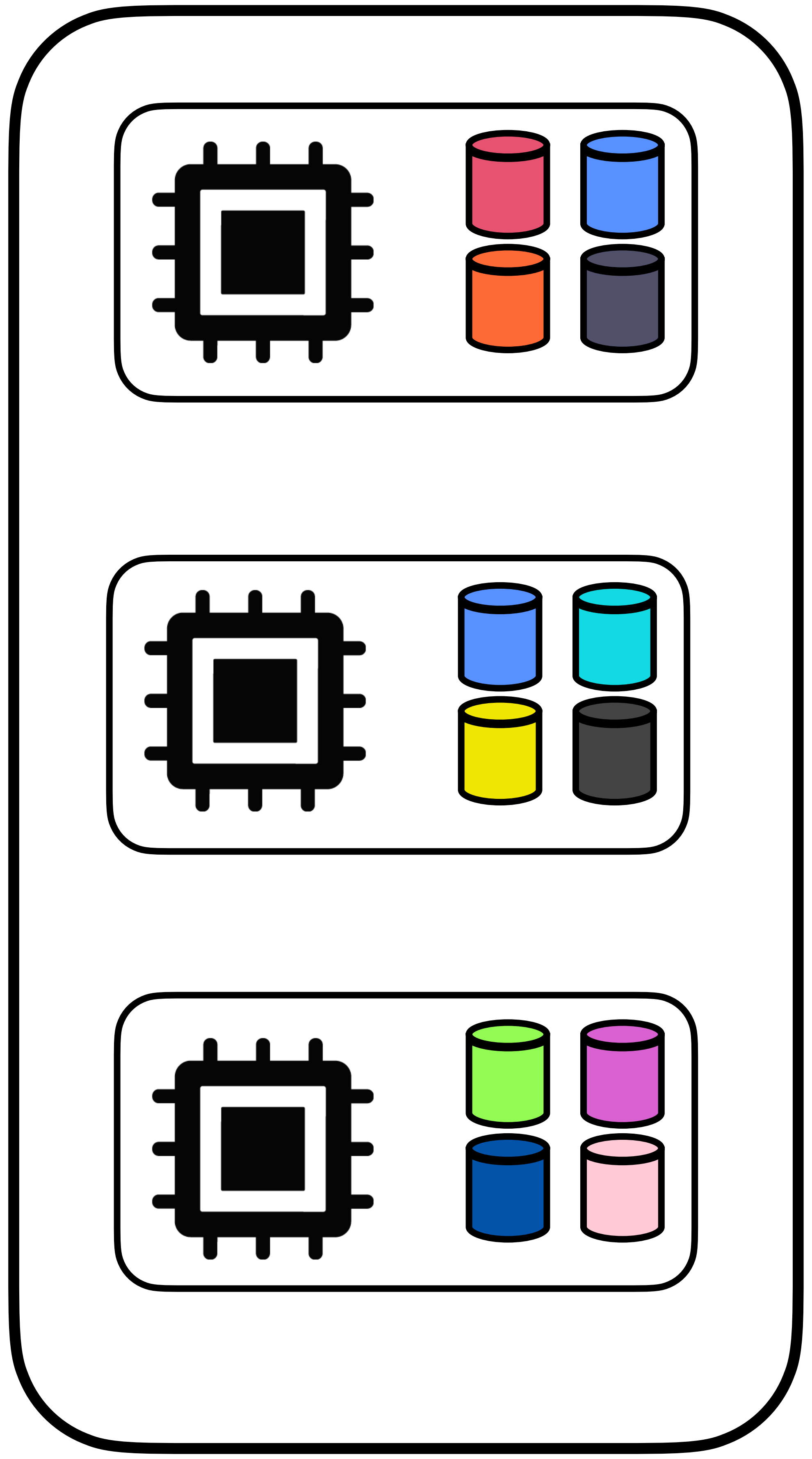
	offset
queue 3	5988
queue 4	6223

Task-level state



- Maps to physical partitioning of compute tasks.
- Relevant for continuous task-parallel computing (e.g., online ML)
- Typically *non-growing* state
- Common on message-broker native stream proc. frameworks (e.g., Samza)
- **Hard to re-partition**

Key-Level State

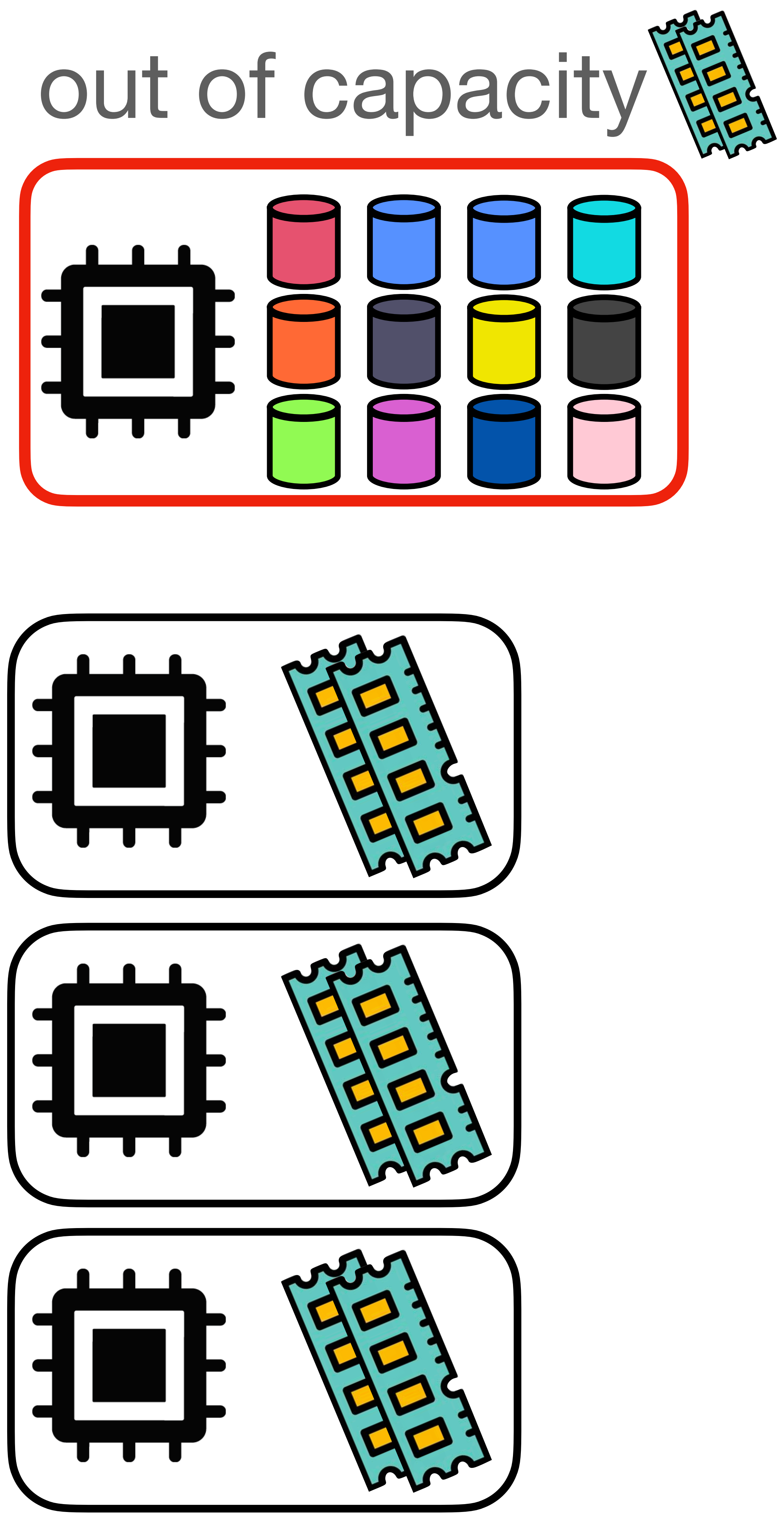


keyby(…)

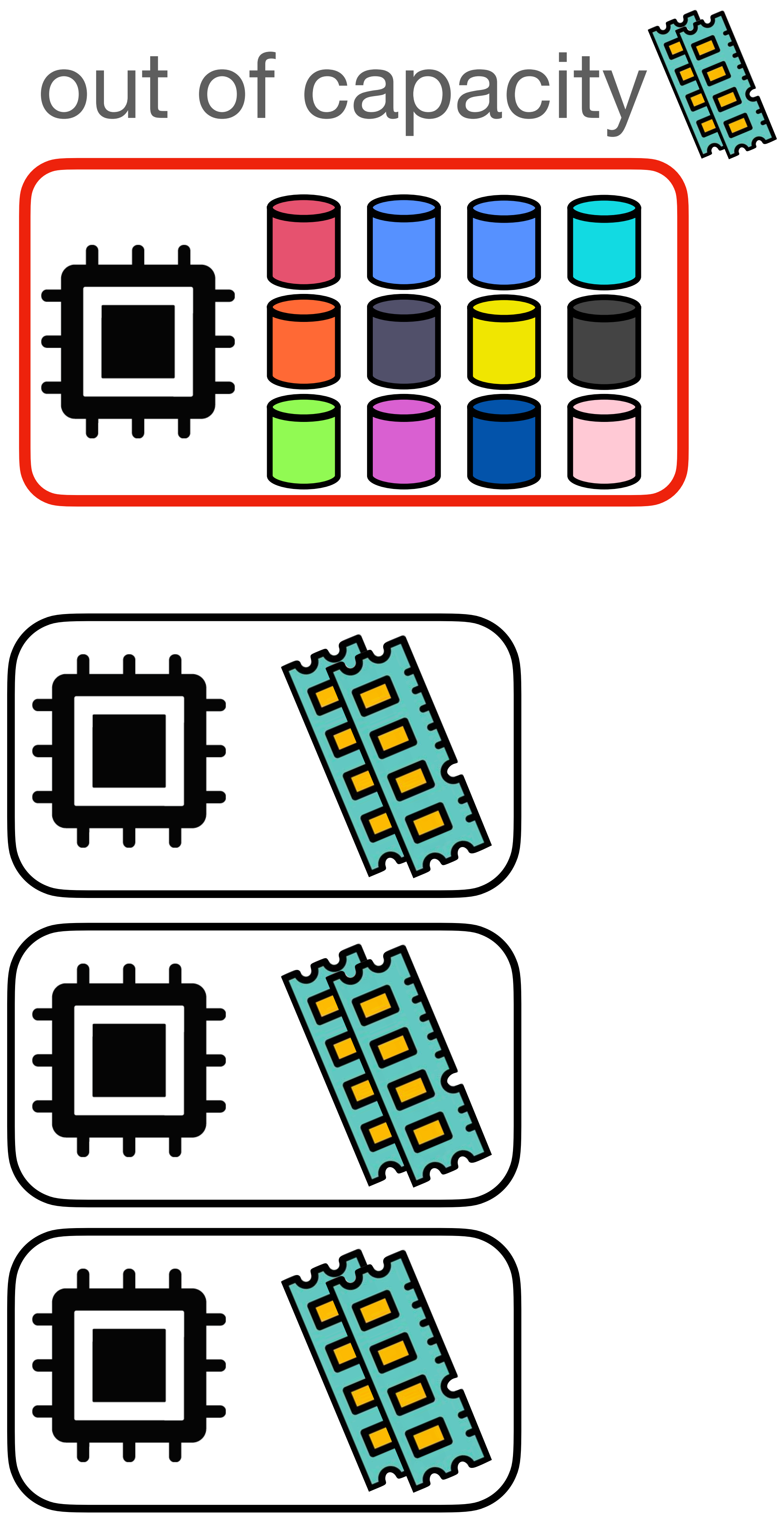
groupBy(..)

- Maps to logical partitioning of compute tasks (multiple keys handled by each task).
- Typically *growing* state
- Common for data parallel stream computation
- **Easy to partition**

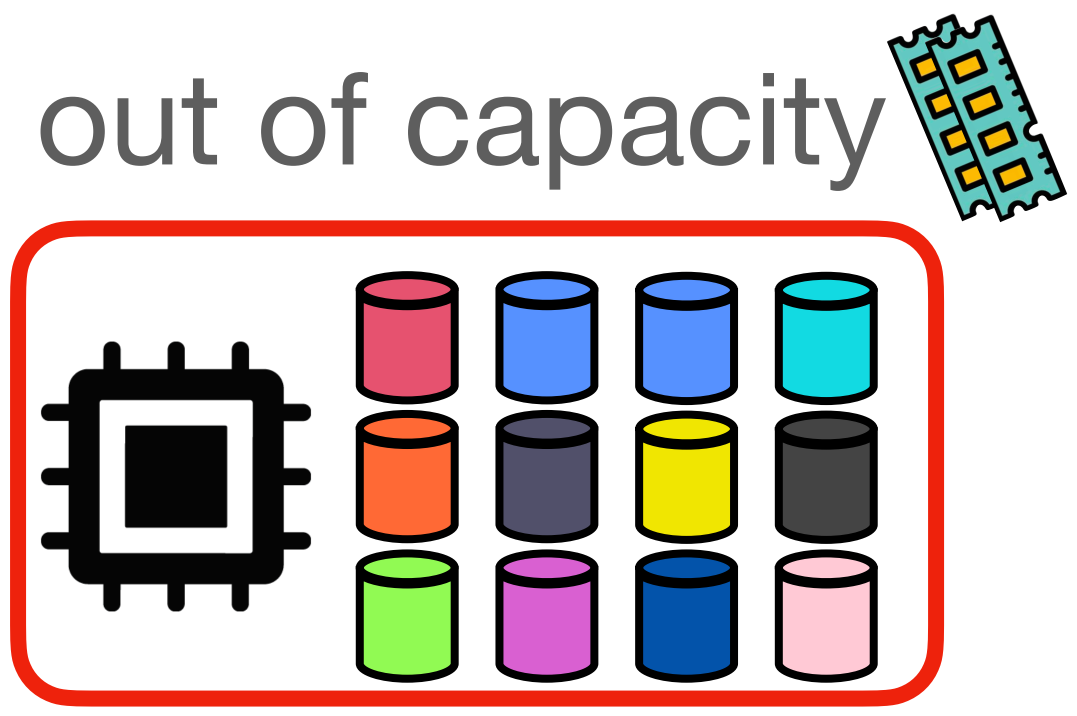
Scalable (Out-of-Core) State Architectures

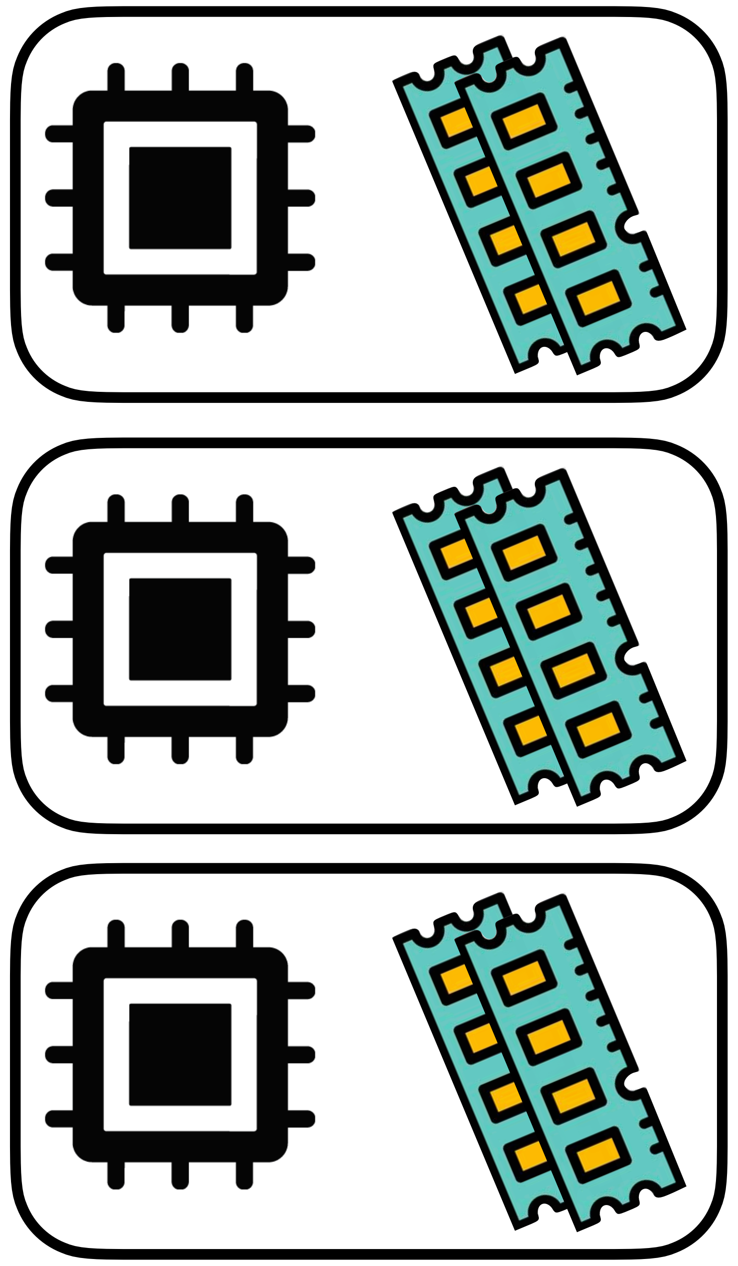


Scalable (Out-of-Core) State Architectures

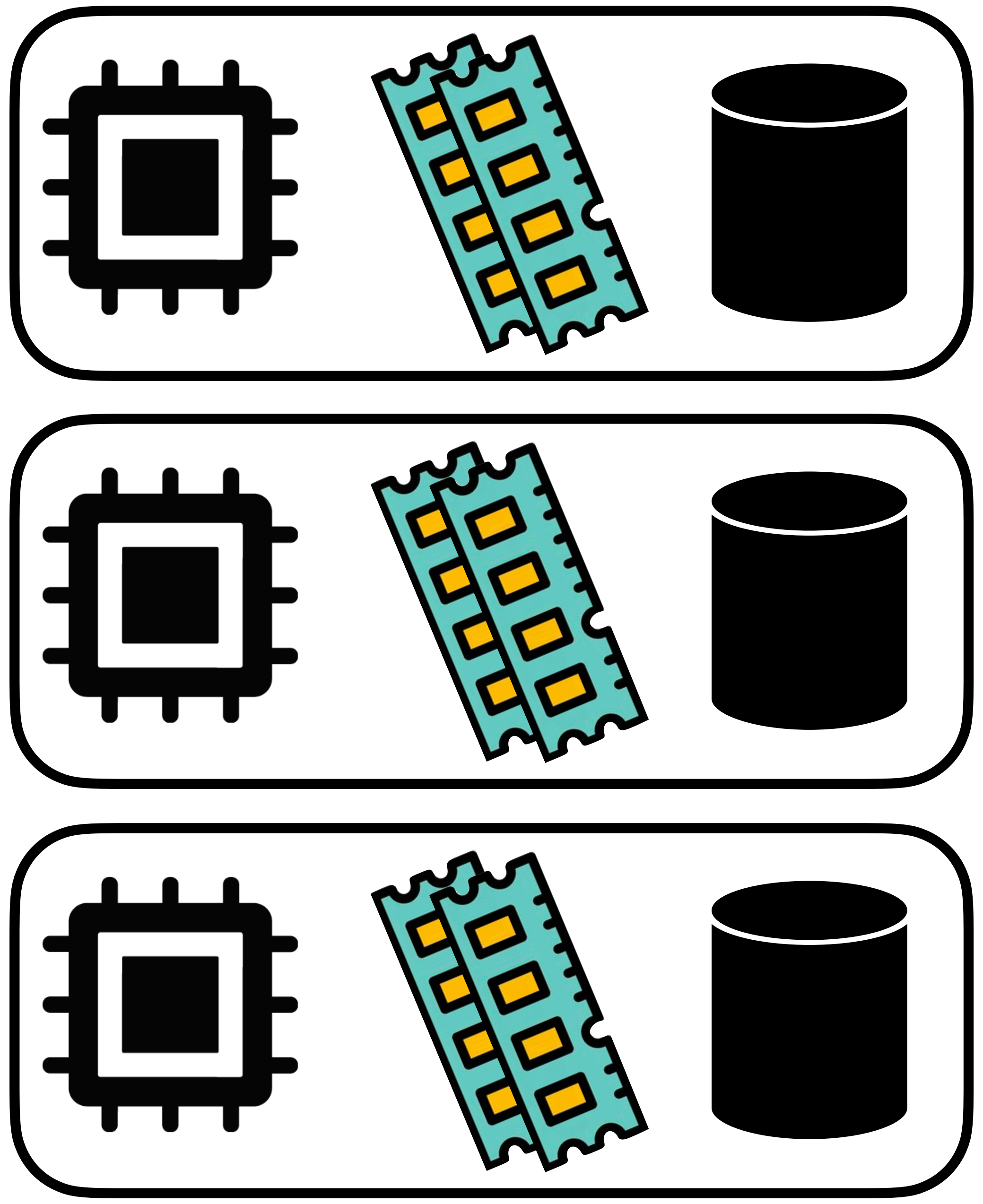


Scalable (Out-of-Core) State Architectures





I. Embedded State



Flink

Seep

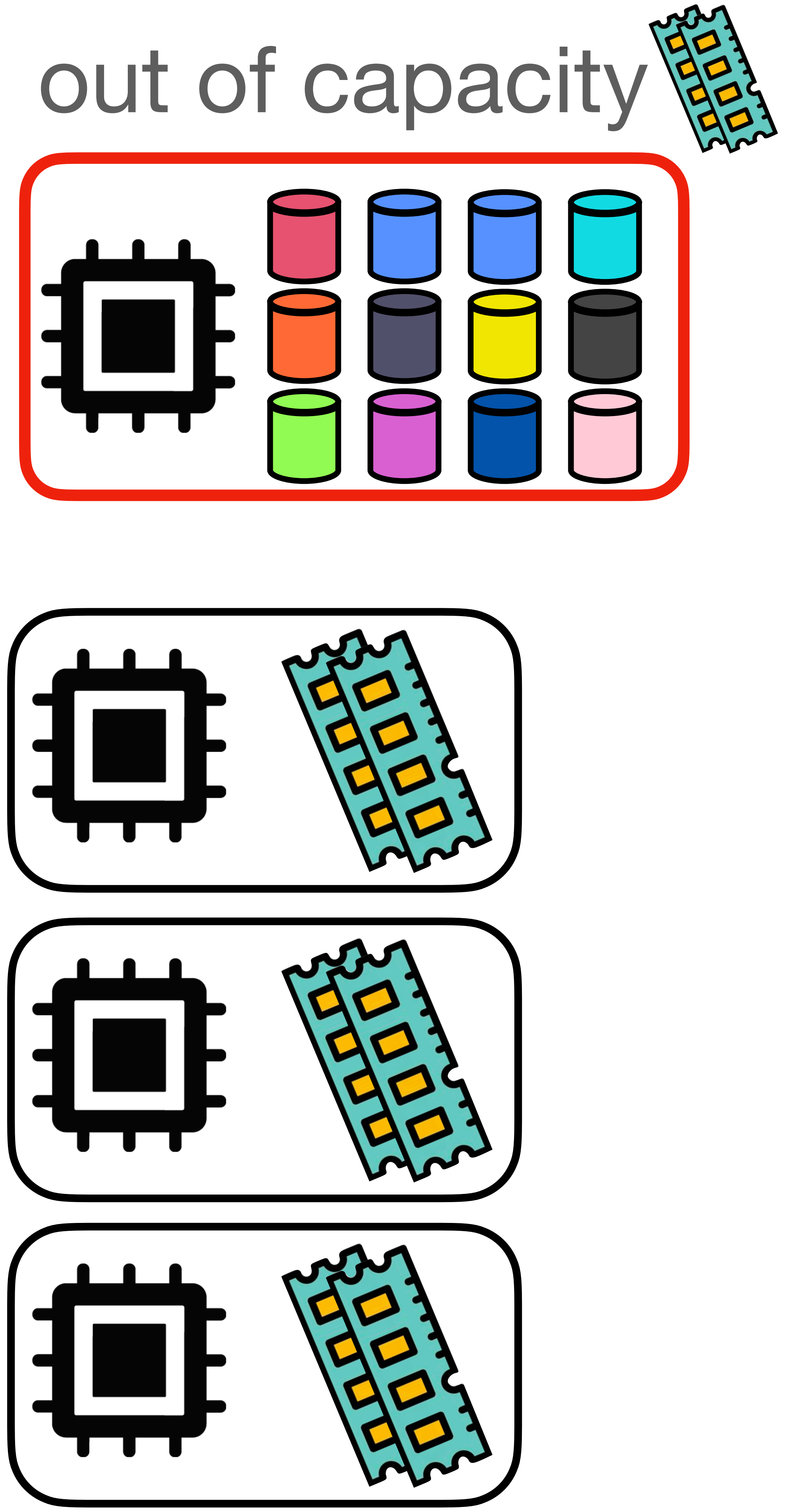
IBM-Streams

Apex

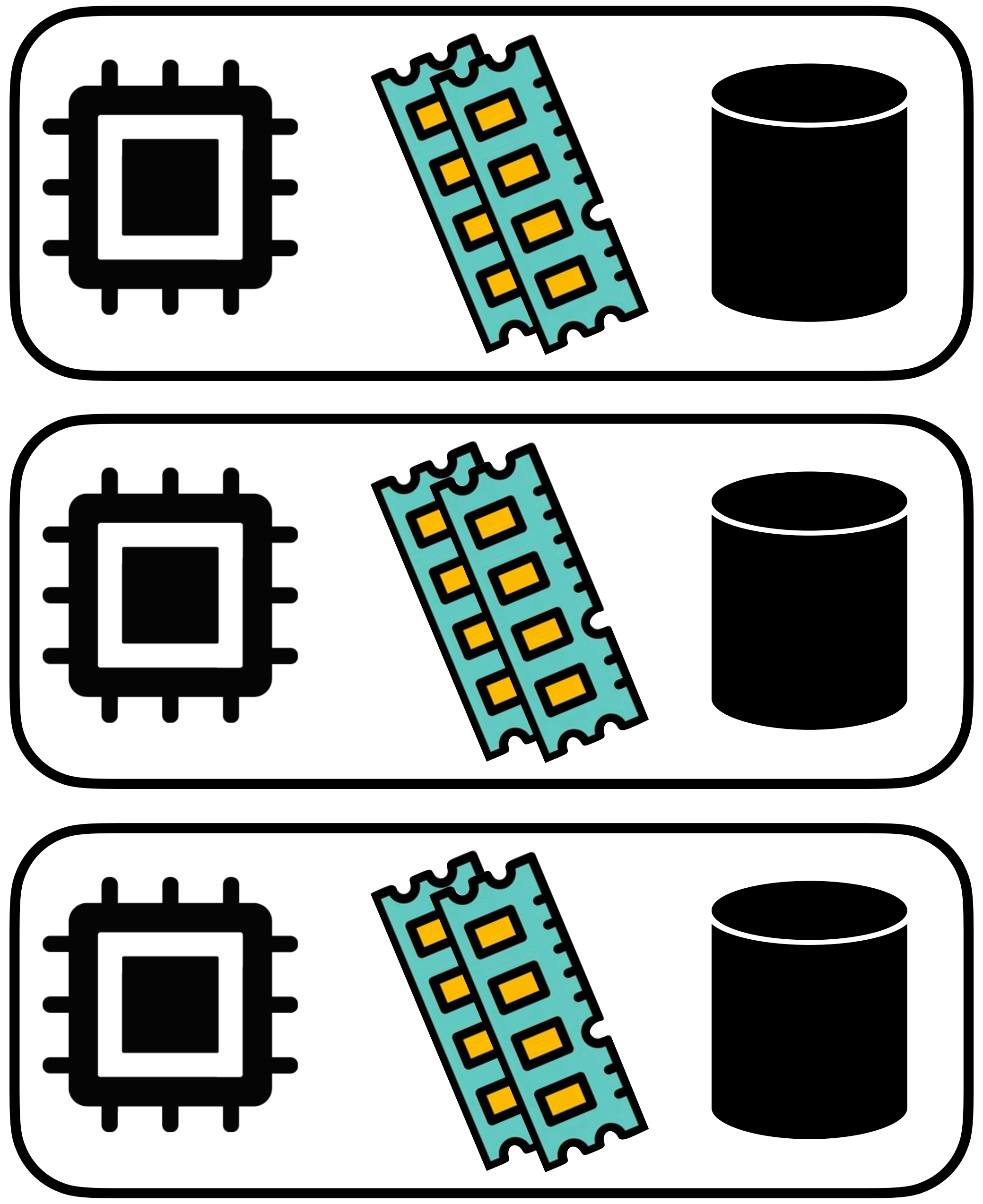
Samza

Spark Structured Streaming

Scalable (Out-of-Core) State Architectures

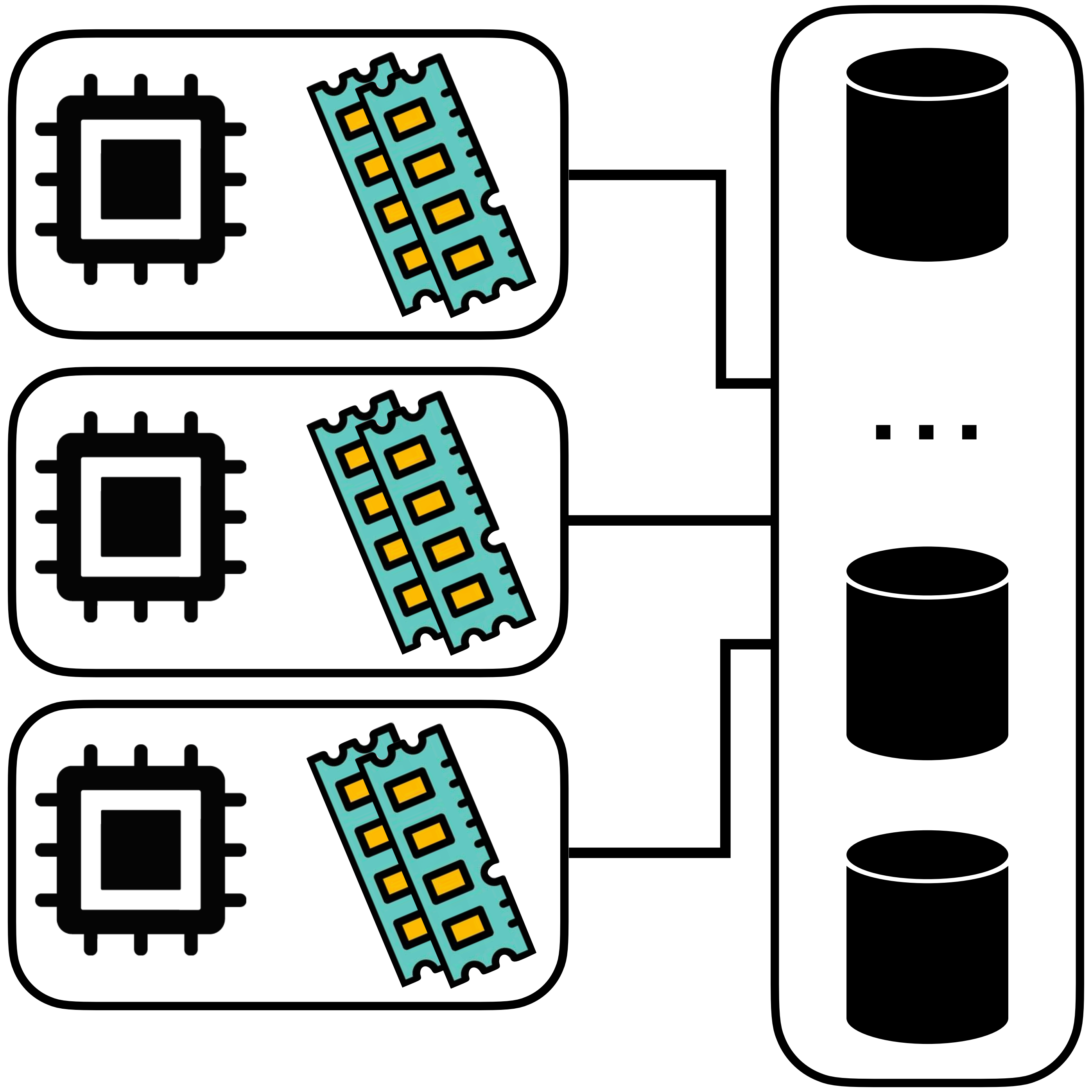


I. Embedded State



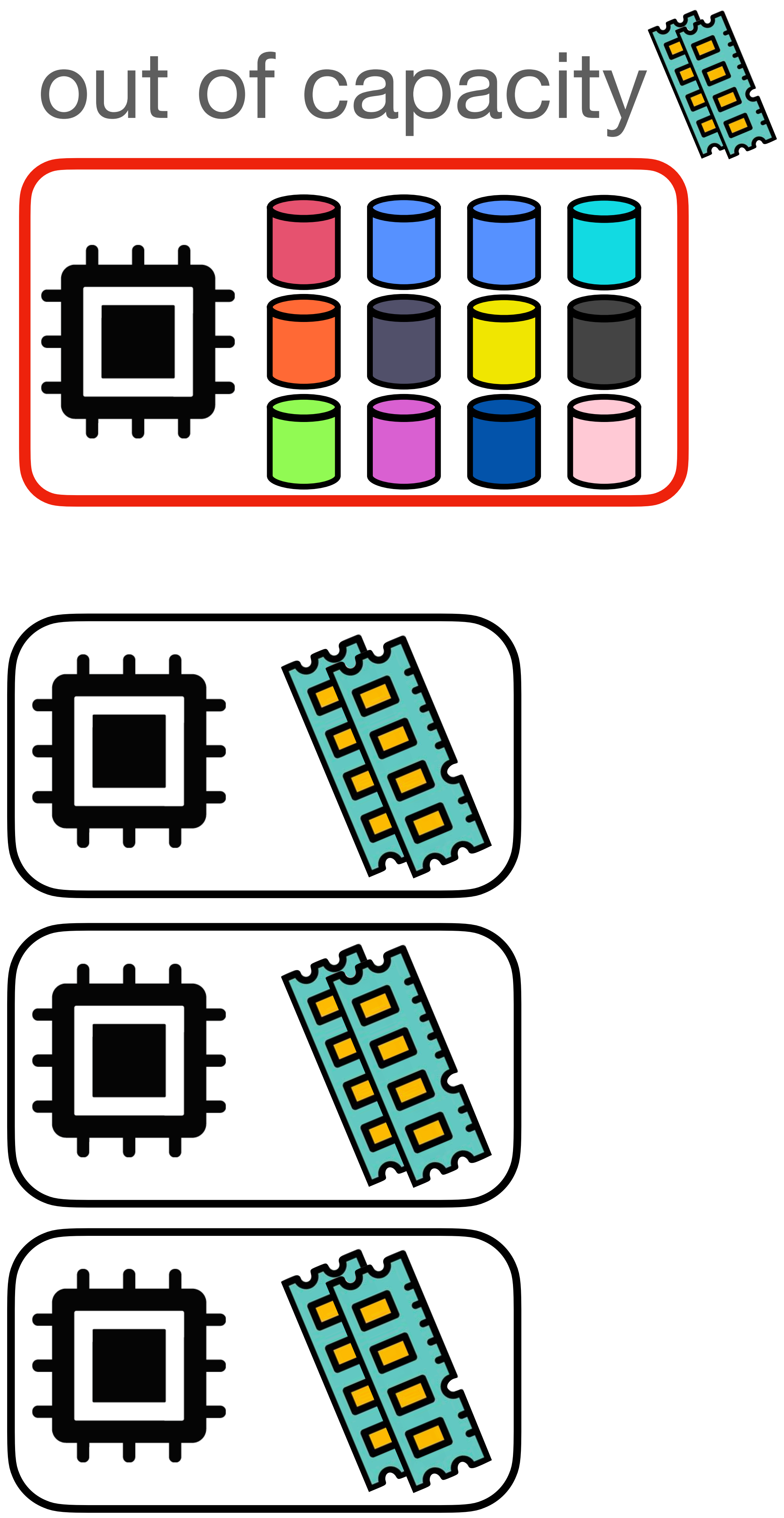
Flink | Apex
Seep | Samza
IBM-Streams | Spark Structured Streaming

II. External State

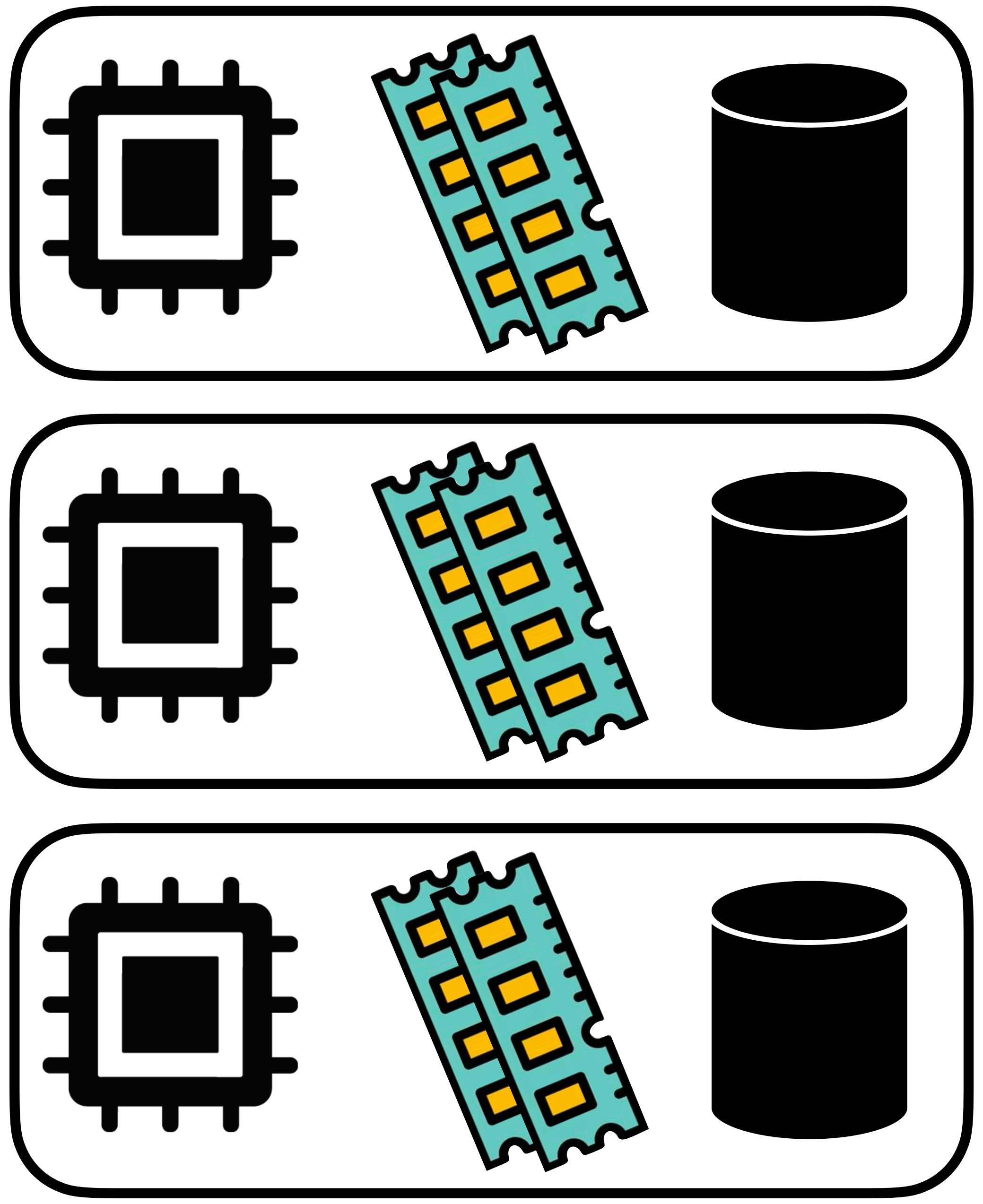


Millwheel (Google Dataflow)

Scalable (Out-of-Core) State Architectures



I. Embedded State



Flink

Seep

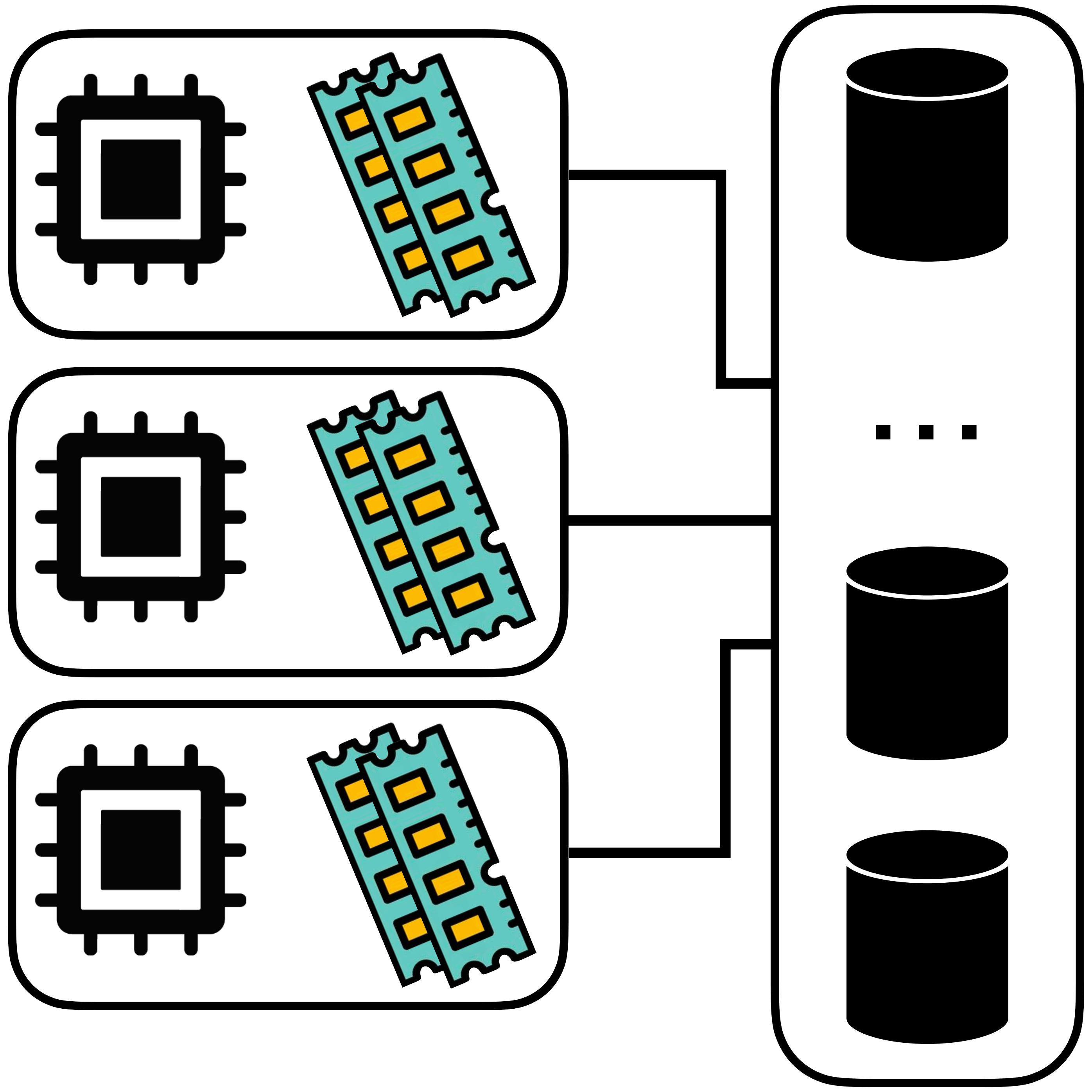
IBM-Streams

Apex

Samza

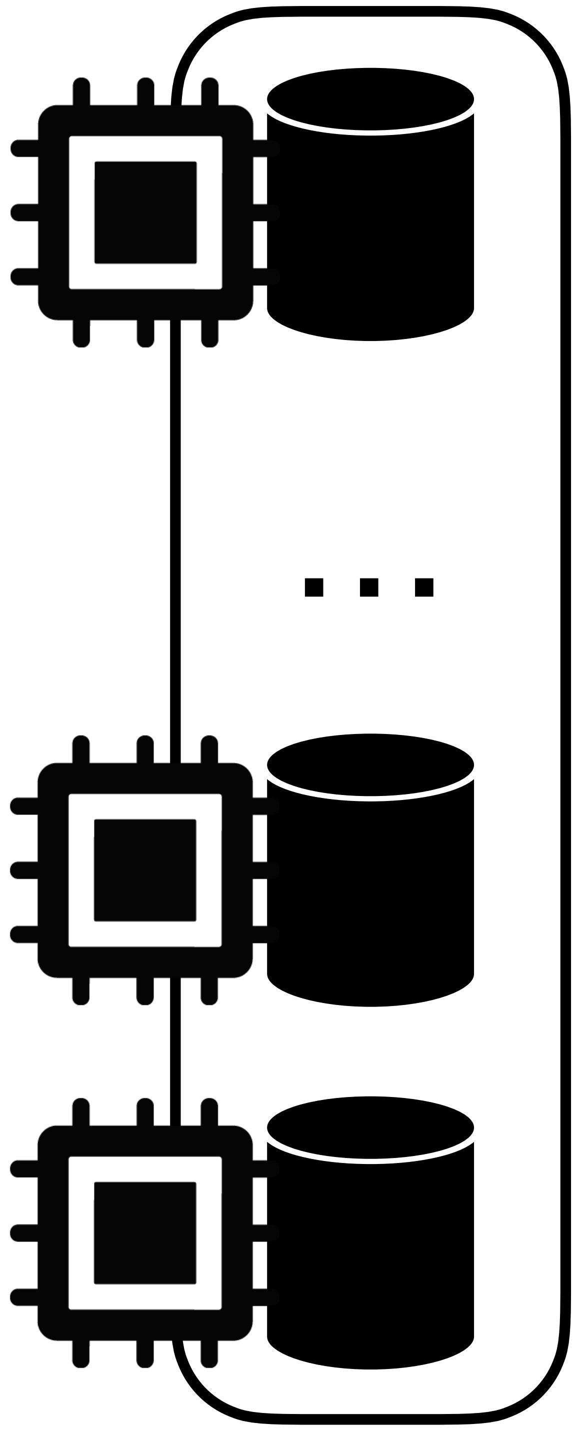
Spark Structured Streaming

II. External State



Millwheel (Google Dataflow)

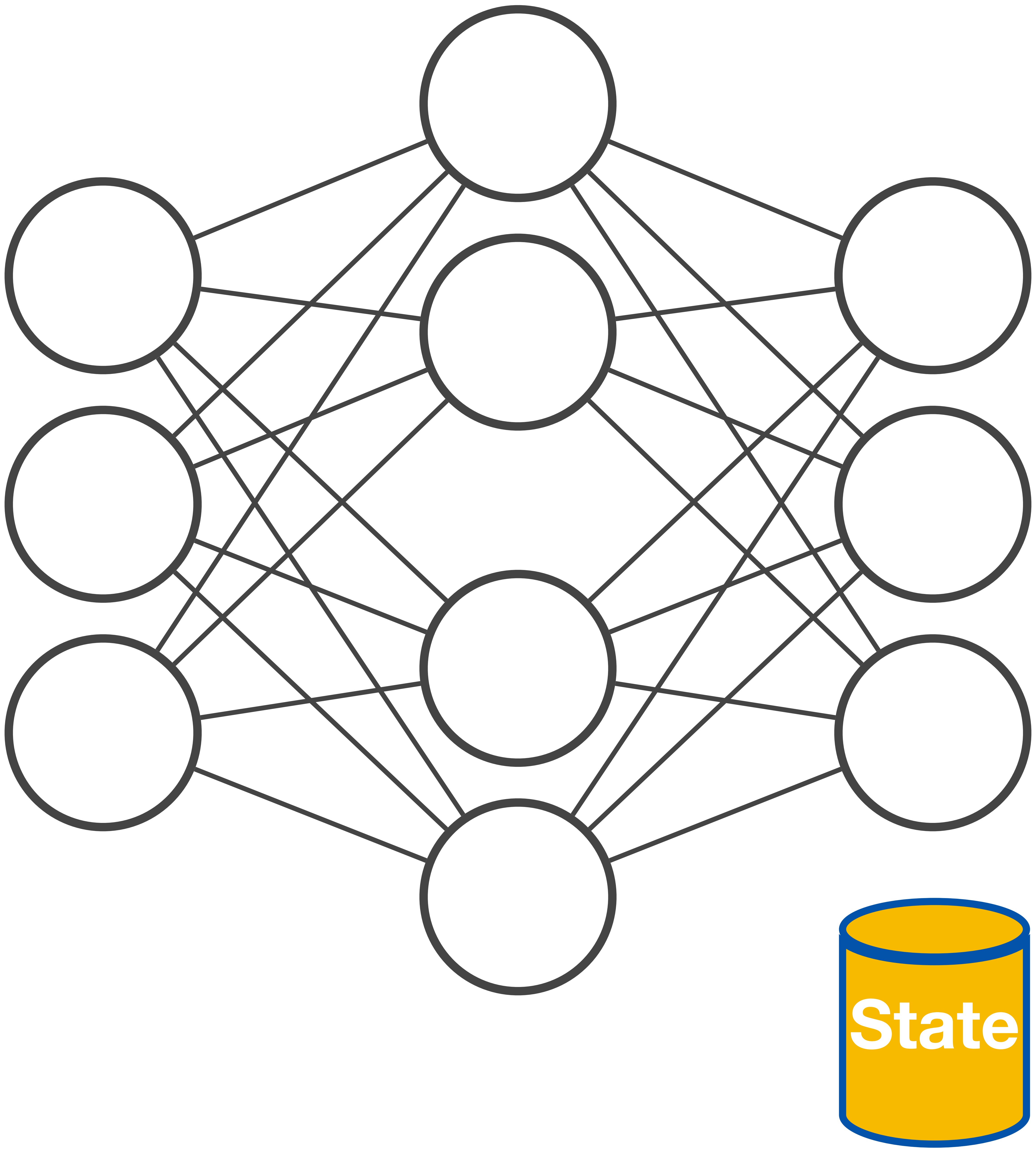
III. Embedded Compute



S-Store (on H-Store)

Kafka-Streams (on Kafka)

Embedded State



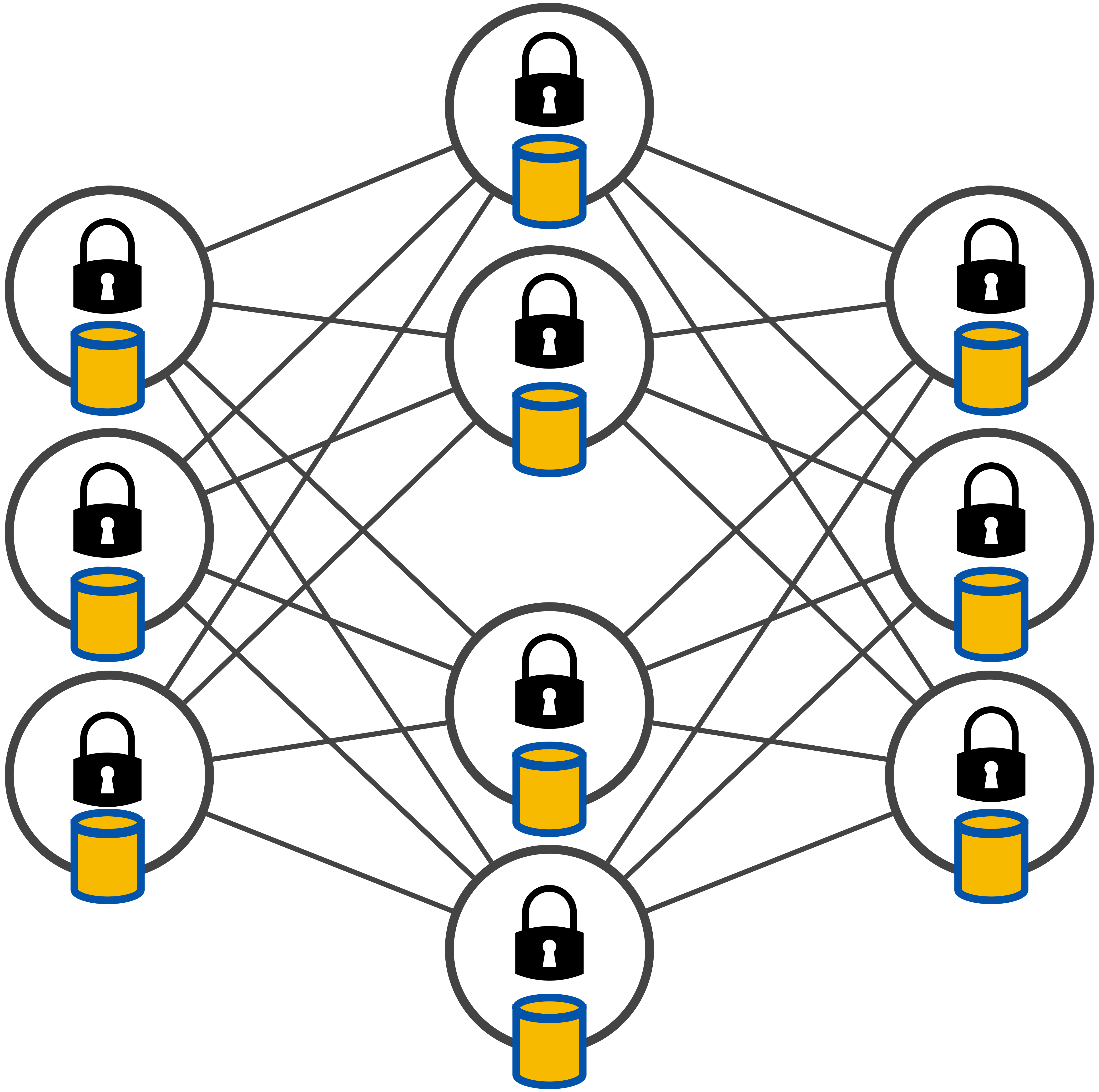
Pros

- Direct, fast state access
- No external calls
- Flexibility on local data structures

Cons

- Strong Coupling on Scalability
- Challenging Transactional Processing
- Complex/Slow Reconfiguration

Embedded State



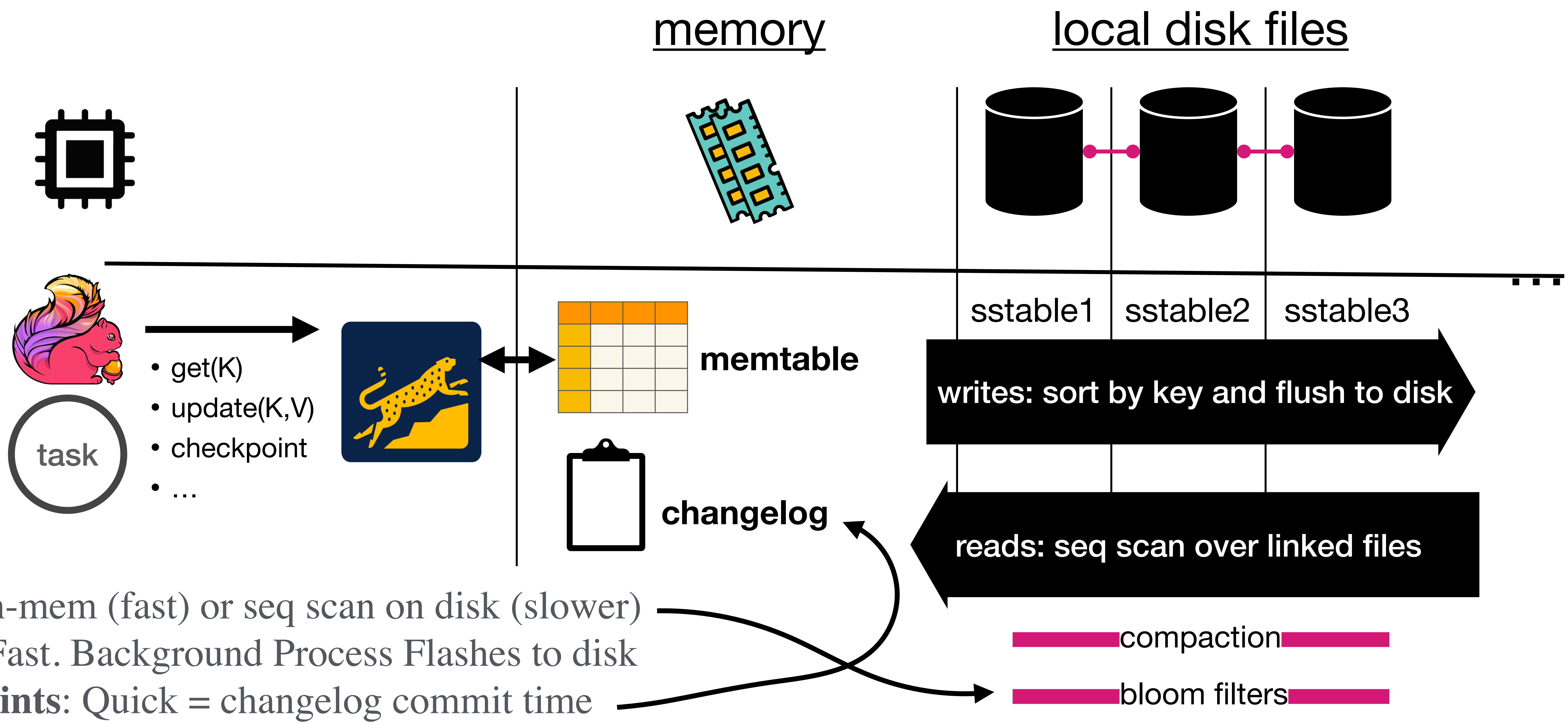
Pros

- Direct, fast state access
- No external calls
- Flexibility on local data structures

Cons

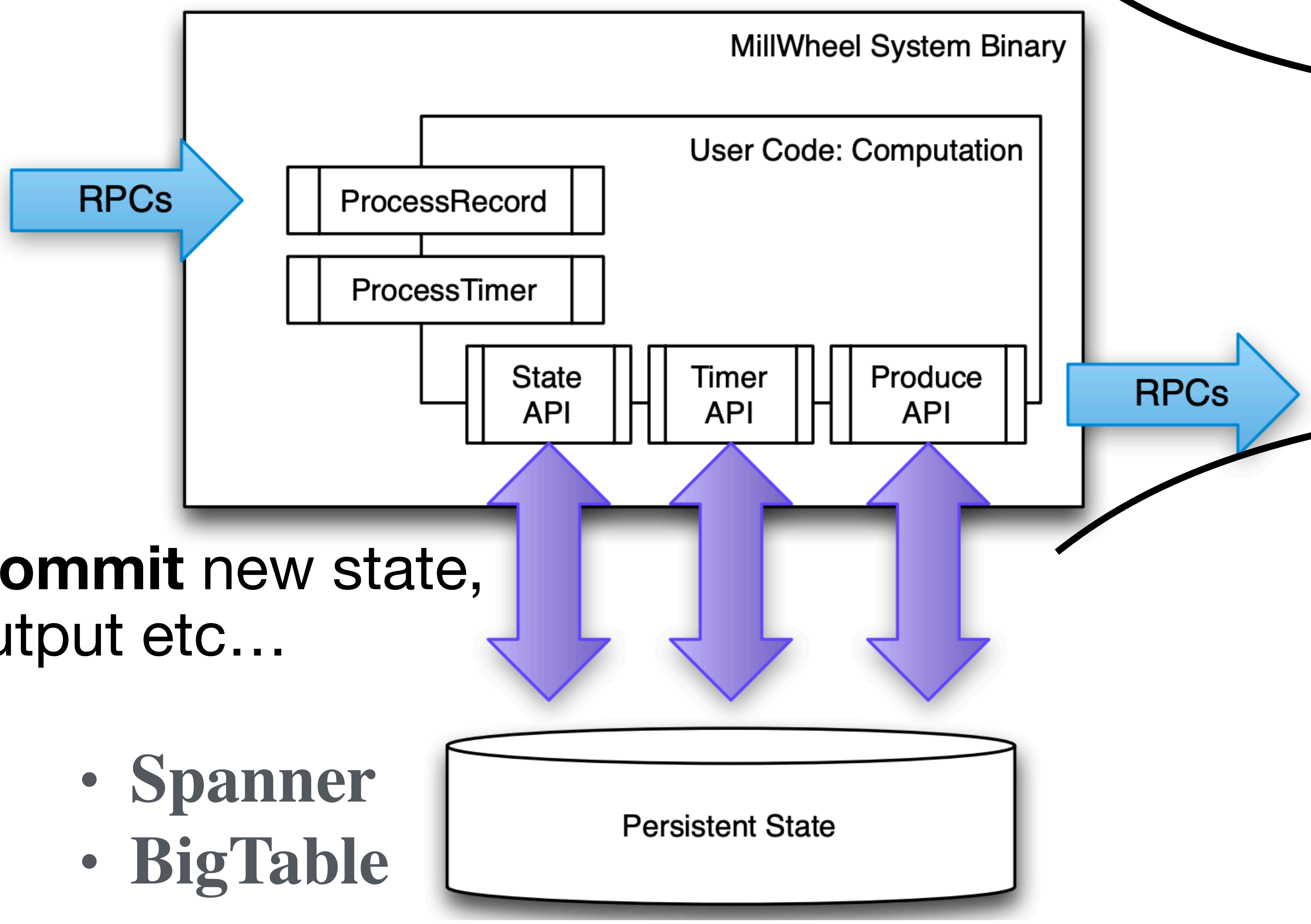
- Strong Coupling on Scalability
- Challenging Transactional Processing
- Complex/Slow Reconfiguration

Embedded State Example



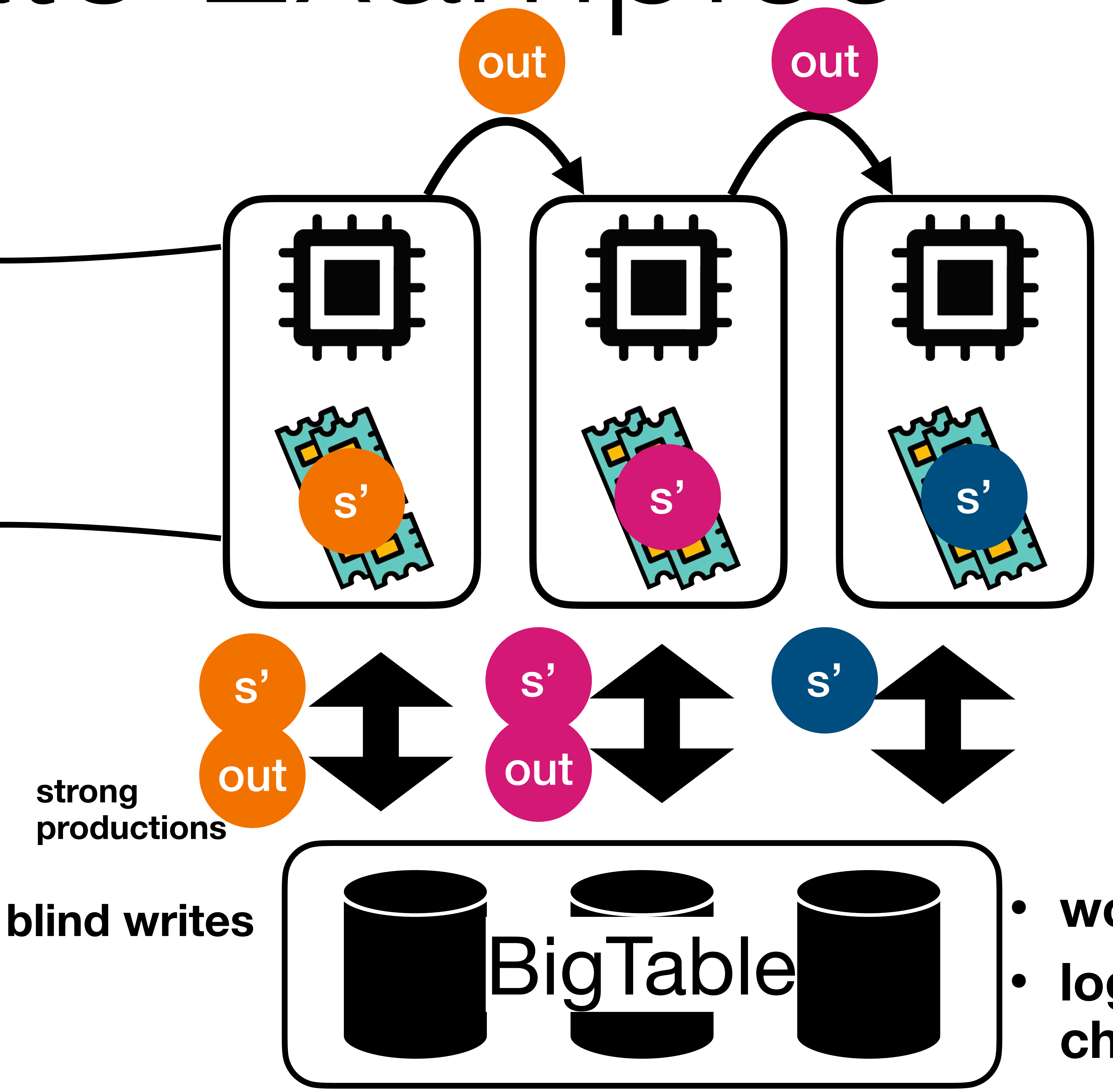
- **Reads**: in-mem (fast) or seq scan on disk (slower)
- **Writes**: Fast. Background Process Flashes to disk
- **Checkpoints**: Quick = changelog commit time

External State Examples



commit new state, output etc…

- Spanner
- BigTable



- working state
- logs/ checkpoints

Millwheel: Fault-tolerant stream processing at internet scale
Akidau T, Balikov A, Bekirŏglu K, et. al. in Proceedings of the VLDB Endowment (2013)

State Management in Apache Flink

State management in Apache Flink

All data maintained by a task and used to compute results: a local or instance variable that is accessed by a task's business logic

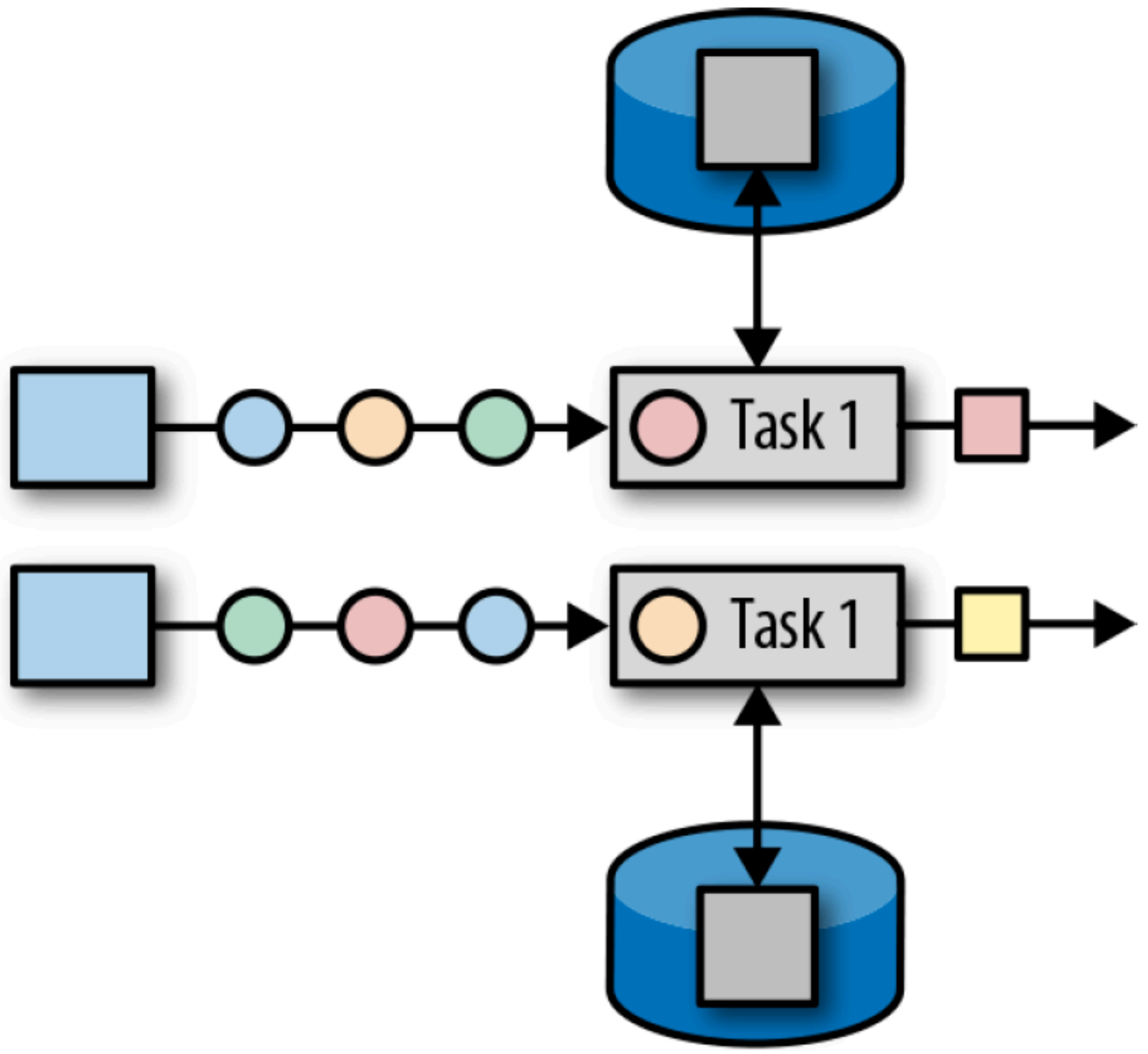
Operator state is scoped to an operator task, i.e. records processed by the same parallel task have access to the same state

- It cannot be accessed by other parallel tasks of the same or different operators

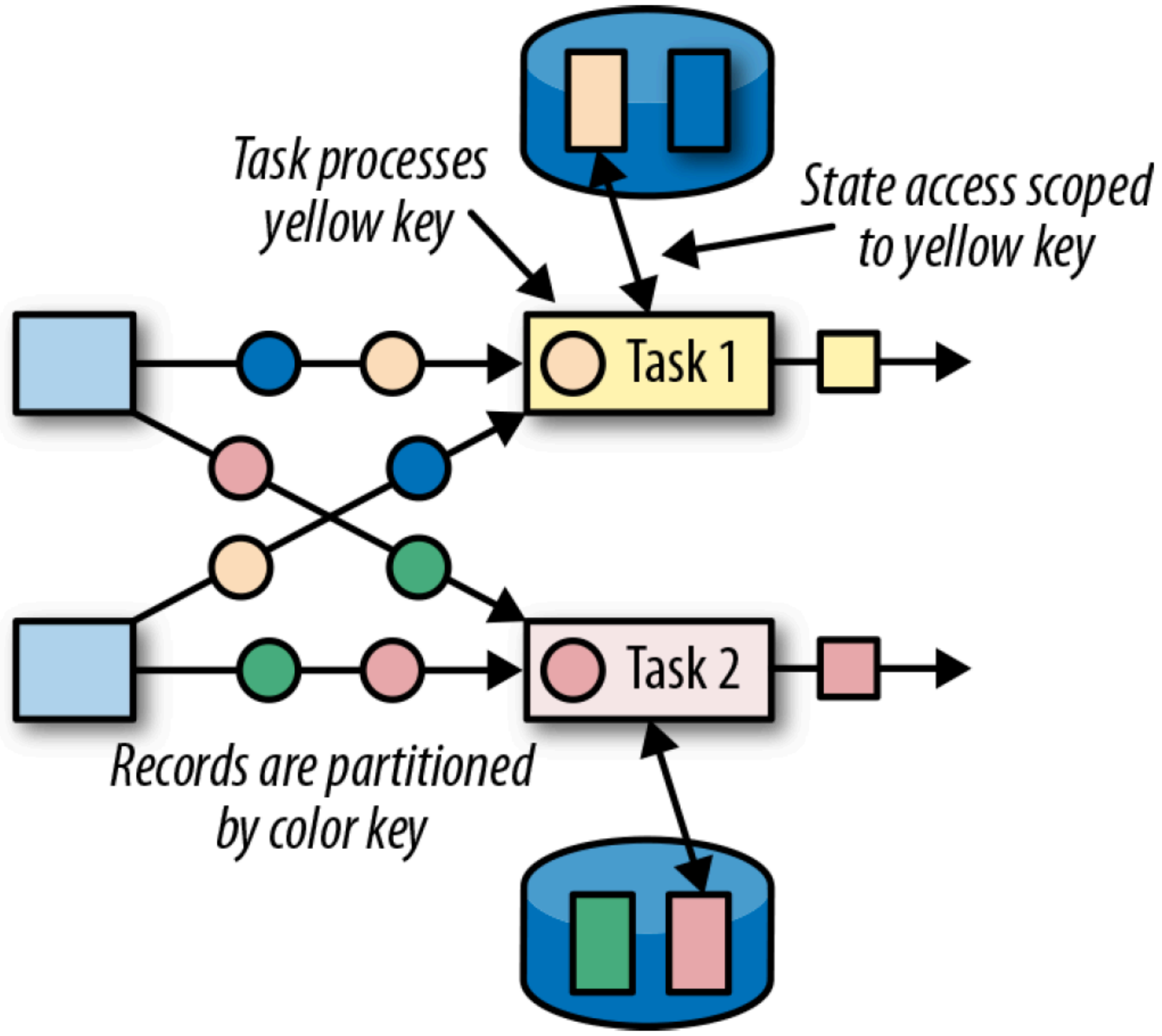
Keyed state is scoped to a key defined in the operator's input records

- Flink maintains one state instance per key value and partitions all records with the same key to the operator task that maintains the state for this key
- State access is automatically scoped to the key of the current record so that all records with the same key access the same state

State types



Operator state



Keyed state

State backends

A pluggable component that determines how state is stored, accessed, and maintained.

State backends are responsible for:

- local state management
- checkpointing state to remote and persistent storage, e.g. a distributed filesystem or a database system

- Available state backends in Flink:
  - In-memory
  - File system
  - RocksDB

Which backend to choose?

MemoryStateBackend

- Stores state as regular objects on TaskManager’s heap
- Low read/write latencies
- OutOfMemoryError if large grows too large, GC pauses
- Checkpoints sent to JobManager's heap memory, i.e. the state is lost in case of failure
- Use only for development and debugging purposes!

FsStateBackend

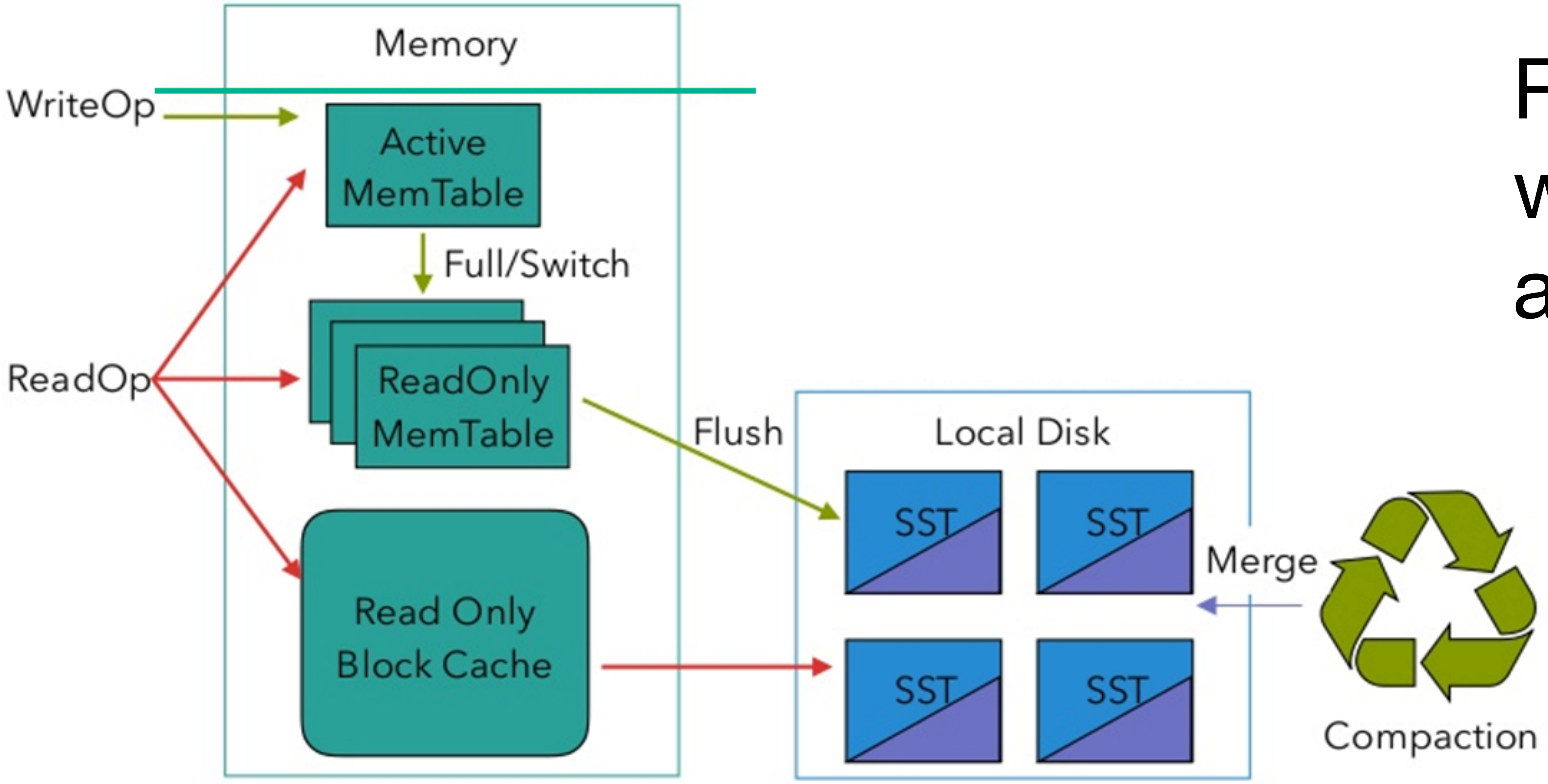
- Stores state on TaskManager’s heap but checkpoints it to a remote file system
- In-memory speed for local accesses and fault tolerance
- Limited to TaskManager’s memory and might suffer from GC pauses

Which backend to choose?

RocksDBStateBackend

- Stores all state into embedded RocksDB instances
- Accesses require de/serialization
- Checkpoints state to a remote file system and supports incremental checkpoints
- Use for applications with very large state

RocksDB



RocksDB is an LSM-tree storage engine with key/value interface, where keys and values are arbitrary byte streams.

https://rocksdb.org/

https://www.ververica.com/blog/manage-rocksdb-memory-size-apache-flink

RocksDB

- RocksDB is a *persistent* key value store: data lives on disk, state can grow larger than available memory and will not be lost upon failure.
- Keys and values are arbitrary byte arrays: serialization and deserialization is required to access the state via a Flink program.
- The keys are *ordered* according to a user-specified comparator function.

Basic operations

- **Get(key)**: fetch a single key-value from the DB
- **Put(key, val)**: insert a single key-value into the DB
- **Iterator/RangeScan**: seek to a specified key and then scan one key at a time from that point (keys are sorted)
- **Merge**: a lazy read-modify-write

Configuring the state backend

In `conf/flink.conf.yaml`:

```
# Supported backends are 'jobmanager', 'filesystem', ‘rocksdb'
#
# state.backend: rocksdb
#
# Directory for checkpoints filesystem
#
# state.checkpoints.dir: path/to/checkpoint/folder/
```

In your Flink program:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val checkpointPath: String = ???

// configure path for checkpoints on the remote filesystem
val backend = new RocksDBStateBackend(checkpointPath)

// configure the state backend
env.setStateBackend(backend)
```

Flink's state primitives

- **`ValueState[T]:`** a single value of type T
  - `ValueState.value()`
  - `ValueState.update(value: T)`

- **`ListState[T]:`** a list of elements of type T
  - `ListState.add(value: T)`
  - `ListState.addAll(values: java.util.List[T]).`
  - `List State.get(): Iterable[T]`
  - `ListState.update(values: java.util.List[T])`

Flink's state primitives

- **`MapState[K, V]:`** a map of keys and values
  - `get(key: K), put(key: K, value: V), contains(key: K), remove(key: K)`
  - iterators over the contained entries, keys, and values

- **`ReducingState[T]:`** aggregates values using a `ReduceFunction`
  - `ReducingState.add(value: T)`
  - `ReducingState.get()`

- **`AggregatingState[I, O]:`** aggregates values using an `AggregateFunction`

Using state in Flink

```
val sensorData: DataStream[Reading] = ???

// partition and key the stream on the sensor ID
val keyedData: KeyedStream[Reading, String] =
                  sensorData
                  .keyBy(_.id)

// apply a stateful FlatMapFunction on the keyed stream
val alerts: DataStream[(String, Double, Double)] =
                  keyedData
                  .flatMap(new TemperatureAlertFunction(1.7))
```

Using state in Flink

```
val sensorData: DataStream[Reading] = ???

// partition and key the stream on the sensor ID
val keyedData: KeyedStream[Reading, String] =
                 sensorData
                 .keyBy(_.id)

// apply a stateful FlatMapFunction on the keyed stream
val alerts: DataStream[(String, Double, Double)] =
                 keyedData
                 .flatMap(new TemperatureAlertFunction(1.7))
```

KeyedStream

State access inside the flatMap will be scoped to the key being processed

Registering state

- To create a state object, we have to register a `StateDescriptor` with Flink's runtime via the `RuntimeContext`, which is exposed by **`RichFunctions`** (`RichFlatMapFunction`, `RichMapFunction`, (`Co`)`ProcessFunction`).

- The `StateDescriptor` is specific to the state primitive and includes the **name** of the state and the **data types** of the state:
  - The state name is scoped to the operator so that a function can have more than one state object by registering multiple state descriptors.
  - The data types handled by the state are specified as `Class` or `TypeInformation` objects.

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
  extends RichFlatMapFunction[Reading, (String, Double, Double)] {

 // the state handle object
 private var lastTempState: ValueState[Double] = _

 override def open(parameters: Configuration): Unit = {
  // create state descriptor
  val lastTempDescriptor =
   new ValueStateDescriptor[Double]("lastTemp", classOf[Double])
  // obtain the state handle
  lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
 }
 …

}
```

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
 extends RichFlatMapFunction[Reading, (String, Double, Double)] {

1. // the state handle object                     declare state handle
   private var lastTempState: ValueState[Double] = _

 override def open(parameters: Configuration): Unit = {
  // create state descriptor
  val lastTempDescriptor =
   new ValueStateDescriptor[Double]("lastTemp", classOf[Double])
  // obtain the state handle
  lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
 }
 …

}
```

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
  extends RichFlatMapFunction[Reading, (String, Double, Double)] {

1. // the state handle object                    declare state handle
   private var lastTempState: ValueState[Double] = _

   override def open(parameters: Configuration): Unit = {
    // create state descriptor      assign name and get the state handle
2.  val lastTempDescriptor =
     new ValueStateDescriptor[Double]("lastTemp", classOf[Double])
    // obtain the state handle
    lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
   }
   …

}
```

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
  extends RichFlatMapFunction[Reading, (String, Double, Double)] {

1. // the state handle object                              declare state handle
   private var lastTempState: ValueState[Double] = _

  override def open(parameters: Configuration): Unit = {
   // create state descriptor      assign name and get the state handle
2. val lastTempDescriptor =
    new ValueStateDescriptor[Double]("lastTemp", classOf[Double])
   // obtain the state handle
   lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
  }
  …

}
```

In the operator (FlatMap) class

In the open() method

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
  extends RichFlatMapFunction[SensorReading, (String, Double, Double)] {
 …

 override def flatMap(
   reading: SensorReading,
   out: Collector[(String, Double, Double)]): Unit = {

  // fetch the last temperature from state
  val lastTemp = lastTempState.value()
  // check if we need to emit an alert
  val tempDiff = (reading.temperature - lastTemp).abs
  if (tempDiff > threshold) {
   // temperature changed by more than the threshold
   out.collect((reading.id, reading.temperature, tempDiff))
  }
  // update lastTemp state
  this.lastTempState.update(reading.temperature)
 }
}
```

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
  extends RichFlatMapFunction[SensorReading, (String, Double, Double)] {
 …

 override def flatMap(
   reading: SensorReading,
   out: Collector[(String, Double, Double)]): Unit = {

  // fetch the last temperature from state
3. val lastTemp = lastTempState.value() get state value
  // check if we need to emit an alert
  val tempDiff = (reading.temperature - lastTemp).abs
  if (tempDiff > threshold) {
   // temperature changed by more than the threshold
   out.collect((reading.id, reading.temperature, tempDiff))
  }
  // update lastTemp state
  this.lastTempState.update(reading.temperature)
 }
}
```

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
  extends RichFlatMapFunction[SensorReading, (String, Double, Double)] {
 …

 override def flatMap(
   reading: SensorReading,
   out: Collector[(String, Double, Double)]): Unit = {

  // fetch the last temperature from state
3.val lastTemp = lastTempState.value() get state value
  // check if we need to emit an alert
  val tempDiff = (reading.temperature - lastTemp).abs
  if (tempDiff > threshold) {
   // temperature changed by more than the threshold
   out.collect((reading.id, reading.temperature, tempDiff))
  }
  // update lastTemp state     update state
4.this.lastTempState.update(reading.temperature)
 }
}
```

Using state in Flink

```
class TemperatureAlertFunction(val threshold: Double)
  extends RichFlatMapFunction[SensorReading, (String, Double, Double)] {
 …

 override def flatMap(
   reading: SensorReading,
   out: Collector[(String, Double, Double)]): Unit = {

  // fetch the last temperature from state
3. val lastTemp = lastTempState.value() get state value
  // check if we need to emit an alert
  val tempDiff = (reading.temperature - lastTemp).abs
  if (tempDiff > threshold) {
   // temperature changed by more than the threshold
   out.collect((reading.id, reading.temperature, tempDiff))
  }
4. // update lastTemp state    update state
  this.lastTempState.update(reading.temperature)
 }
}
```

This is the state of the current key (sensor id)

Keyed state scope

Use keyed state to store and access state in the context of a key attribute:

- For each distinct value of the key attribute, Flink maintains one state instance.
- The keyed state instances of a function are distributed across all parallel tasks of the function's operator.

Keyed state can only be used by functions that are applied on a `KeyedStream`:

- When the processing method of a function with keyed input is called, Flink's runtime automatically puts all keyed state objects of the function into the context of the key of the record that is passed by the function call.
- A function **can only access the state that belongs to the record it currently processes**.

Java example

```java
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

// taxi ride events (start, end)
DataStream<TaxiRide> rides = env.addSource(…)).keyBy("rideId");

// taxi fare events (payment, tip)
DataStream<TaxiFare> fares = env.addSource(…).keyBy("rideId");

// match ride and fare events
DataStream<Tuple2<TaxiRide, TaxiFare>> connectedRides = rides
    .connect(fares)
    .flatMap(new MatchFunction());
```

Java example (cont.)

```
public static class EnrichmentFunction extends RichCoFlatMapFunction<TaxiRide, TaxiFare, Tuple2<TaxiRide, TaxiFare>> {
   // define the state primitives here
   private ValueState<TaxiRide> rideState;
   private ValueState<TaxiFare> fareState;

   @Override
   public void open(Configuration config) {
      // initialize the state descriptors here
      rideState = getRuntimeContext().getState(new ValueStateDescriptor<>("saved ride", TaxiRide.class));
      fareState = getRuntimeContext().getState(new ValueStateDescriptor<>("saved fare", TaxiFare.class));
   }

   @Override
   public void flatMap1(TaxiRide ride, Collector<Tuple2<TaxiRide, TaxiFare>> out) throws Exception {
      TaxiFare fare = fareState.value();
      if (fare != null) { // a matching fare exists
         fareState.clear(); // always clear the state you don't need anymore!
         out.collect(new Tuple2(ride, fare));
      } else {
         rideState.update(ride); // no matching fare -> store the ride
      }
   }

   @Override
   public void flatMap2(TaxiFare fare, Collector<Tuple2<TaxiRide, TaxiFare>> out) throws Exception {
     // similar logic for processing fare events
      }
   }
}
```

Operator state

- A function can work with operator list state by implementing the `ListCheckpointed` interface
- `snapshotState()` is invoked when Flink triggers a checkpoint of the stateful function.
- `restoreState()` is always invoked when the job is started or in the case of a failure.

```
List<T> snapshotState(long checkpointId, long timestamp)
void restoreState(List<T> state)
```

A stateful source

```
public static class CounterSource extends RichParallelSourceFunction<Long> implements ListCheckpointed<Long> {

    /**  current offset for exactly once semantics */
    private Long offset = 0L;
    private volatile boolean isRunning = true;

    @Override
    public void run(SourceContext<Long> ctx) {
        final Object lock = ctx.getCheckpointLock();

        while (isRunning) {
            // output and state update are atomic
            synchronized (lock) {
                ctx.collect(offset);
                offset += 1;
            }
        }
    }

    @Override
    public List<Long> snapshotState(long checkpointId, long checkpointTimestamp) {
        return Collections.singletonList(offset);
    }

    @Override
    public void restoreState(List<Long> state) {
        for (Long s : state)
            offset = s;
    }
}
```

A stateful source

```
public static class CounterSource extends RichParallelSourceFunction<Long> implements ListCheckpointed<Long> {

    /**  current offset for exactly once semantics */
    private Long offset = 0L;
    private volatile boolean isRunning = true;

    @Override
    public void run(SourceContext<Long> ctx) {
        final Object lock = ctx.getCheckpointLock();

        while (isRunning) {
            // output and state update are atomic
            synchronized (lock) {
                ctx.collect(offset);
                offset += 1;
            }
        }
    }

    @Override
    public List<Long> snapshotState(long checkpointId, long checkpointTimestamp) {
        return Collections.singletonList(offset);
    }

    @Override
    public void restoreState(List<Long> state) {
        for (Long s : state)
            offset = s;
    }
}
```

get a lock to make output and state update atomic

Further resources

- Working with State: https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/stream/state/state.html

- Managing State in Apache Flink - Tzu-Li (Gordon) Tai: https://www.youtube.com/watch?v=euFMWFDThiE

- Webinar: Deep Dive on Apache Flink State - Seth Wiesman: https://www.youtube.com/watch?v=9GF8Hwqzwnk