Experiences Running Apache Flink at Very Large Scale

@StephanEwen
dataArtisans

Berlin Buzzwords, 2017

Some large scale use cases

@ NETFLIX

- Various use cases
  - Example: Stream ingestion, route events to Kafka, ES, Hive
  - Example: Model user interaction sessions

- Mix of stateless / moderate state / large state

- Stream Processing as a Service
  - Launching, monitoring, scaling, updating

@ NETFLIX

Keystone Prod Scale - only Kafka and ES Flink routers are deployed in prod, (Hive output Flink routers are in test, unaccounted below)

- 4000+ Kafka brokers, 50+ clusters
- 100's of Data Streams (Flink Jobs)
- 3700+ Docker containers running Flink
- 1400+ nodes with 22K+ cpu cores

@

- Blink based on Flink
- A core system in Alibaba Search
  - Machine learning, search, recommendations
  - A/B testing of search algorithms
  - Online feature updates to boost conversion rate
- Alibaba is a major contributor to Flink
- Contributing many changes back to open source

@ Alibaba Group

Alibaba Group 阿里巴巴集团

Blink in Alibaba Production

✓In production for almost one year

✓Run on thousands of nodes

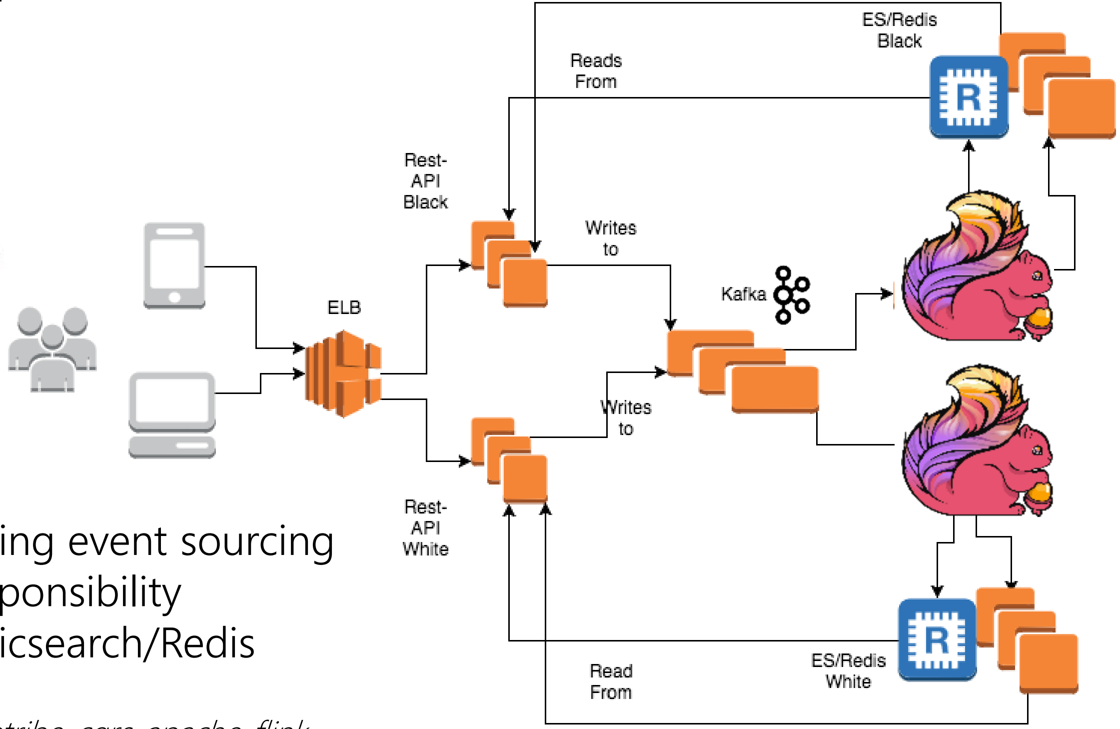
- hundreds of jobs
- The biggest cluster is more than 1000 nodes
- the biggest job has 10s TB states and thousands of subtasks

✓Supported key production services on last Nov 11th, China Single's Day

- China Single's Day is by far the biggest shopping holiday in China, similar to Black Friday in US
- Last year it recorded $17.8 billion worth of gross merchandise volumes in one day
- Blink is used to do real time machine learning and increased conversion by around 30%

@

THE SOCIAL NETWORK
FOR PETROLHEADS



Social network implemented using event sourcing and CQRS (Command Query Responsibility Segregation) on Kafka/Flink/Elasticsearch/Redis

More: https://data-artisans.com/blog/drivetribe-cqrs-apache-flink

How we learned to view Flink through its users

System for Event–driven Applications

Stateful, event-driven,
event-time-aware processing

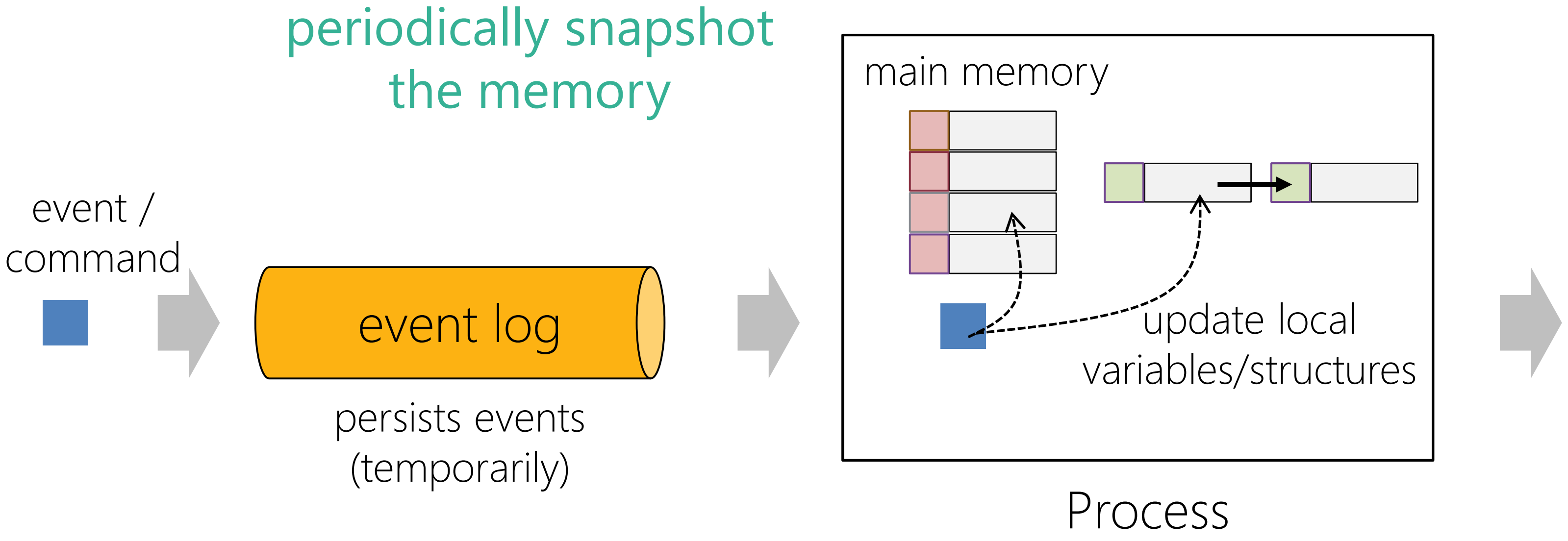
Event-driven
Applications

(event sourcing, CQRS, …)

Stream Processing
(streams, windows, …)

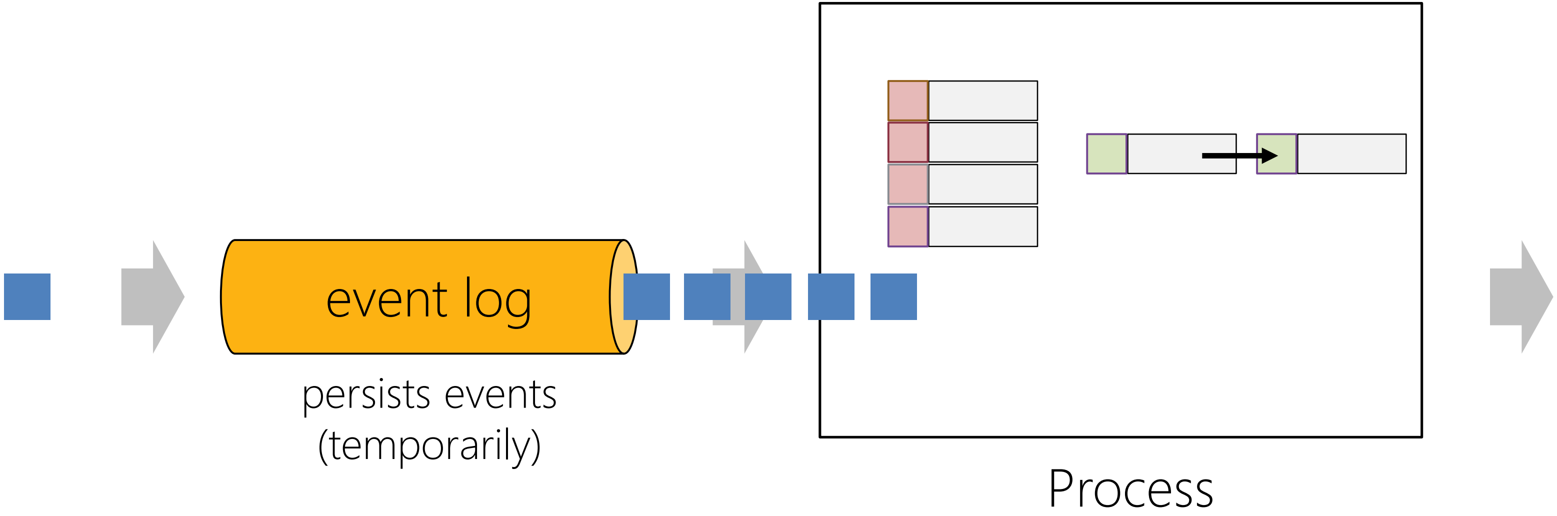
Batch Processing
(data sets)

Event Sourcing + Memory Image



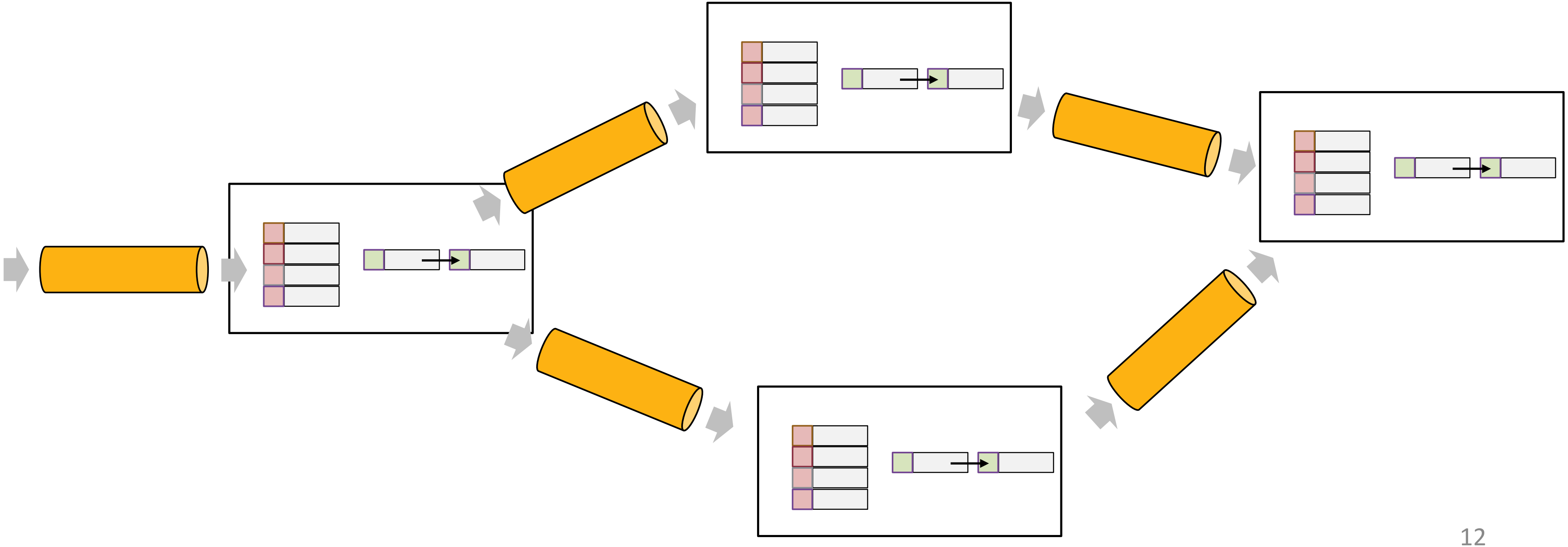
Event Sourcing + Memory Image

Recovery: Restore snapshot and replay events since snapshot



Distributed Memory Image

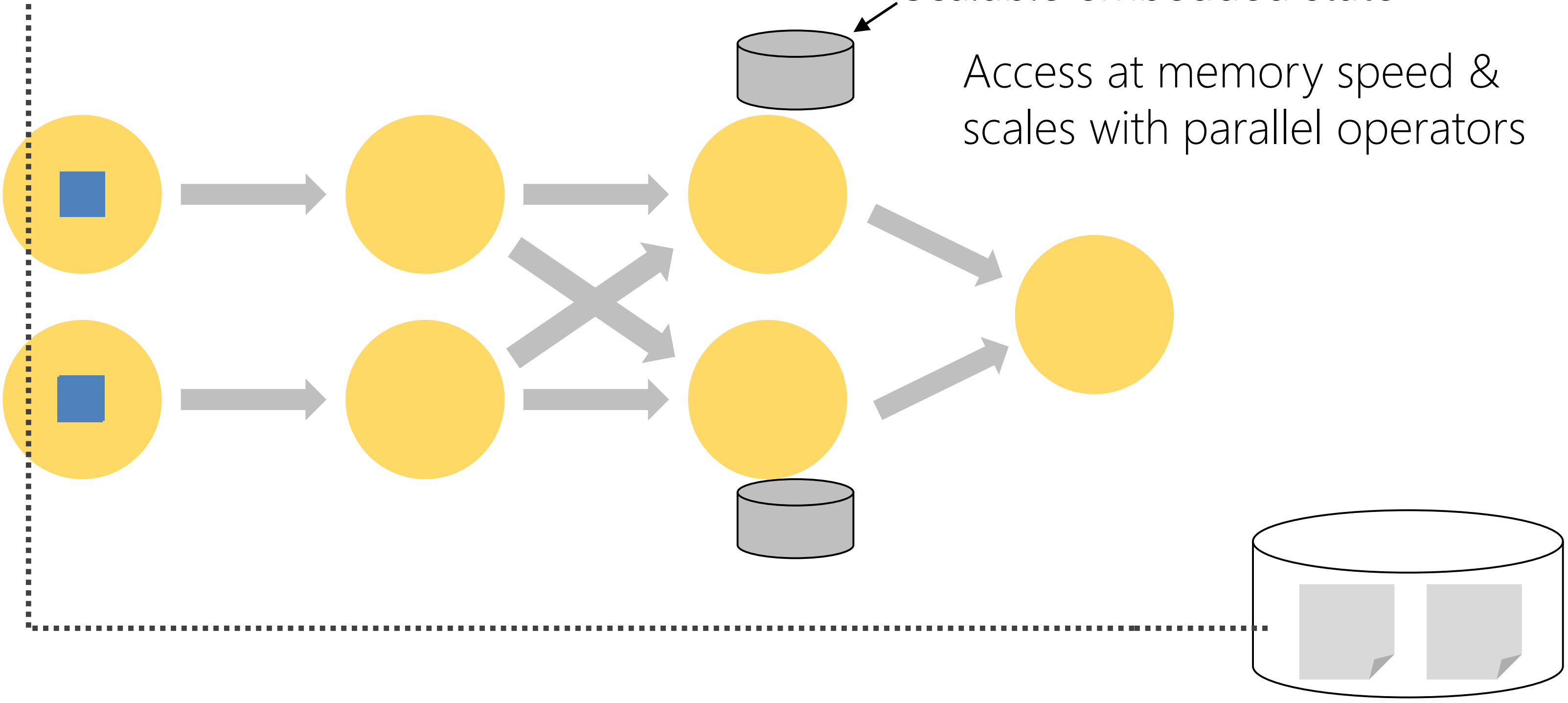
Distributed application, many memory images.
Snapshots are all consistent together.



Stateful Event & Stream Processing

Scalable embedded state

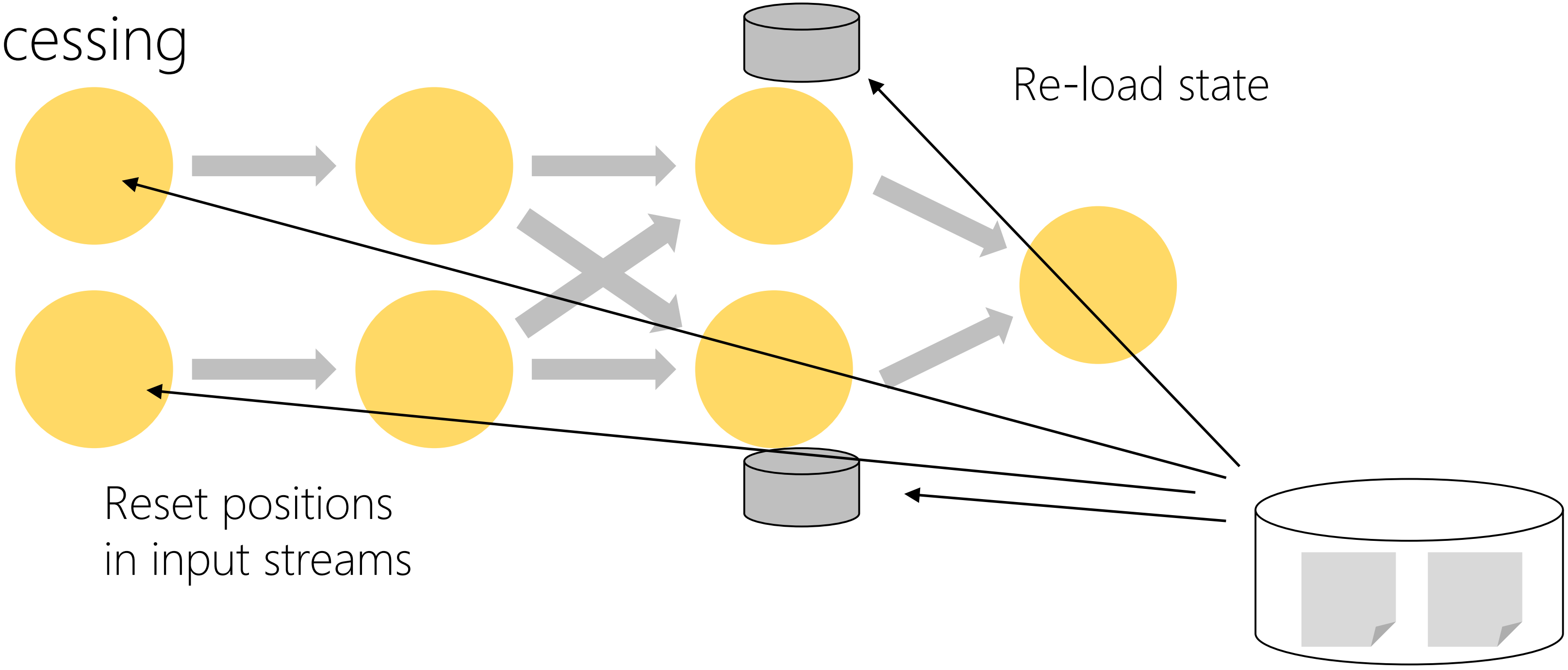
Access at memory speed &
scales with parallel operators



Stateful Event & Stream Processing

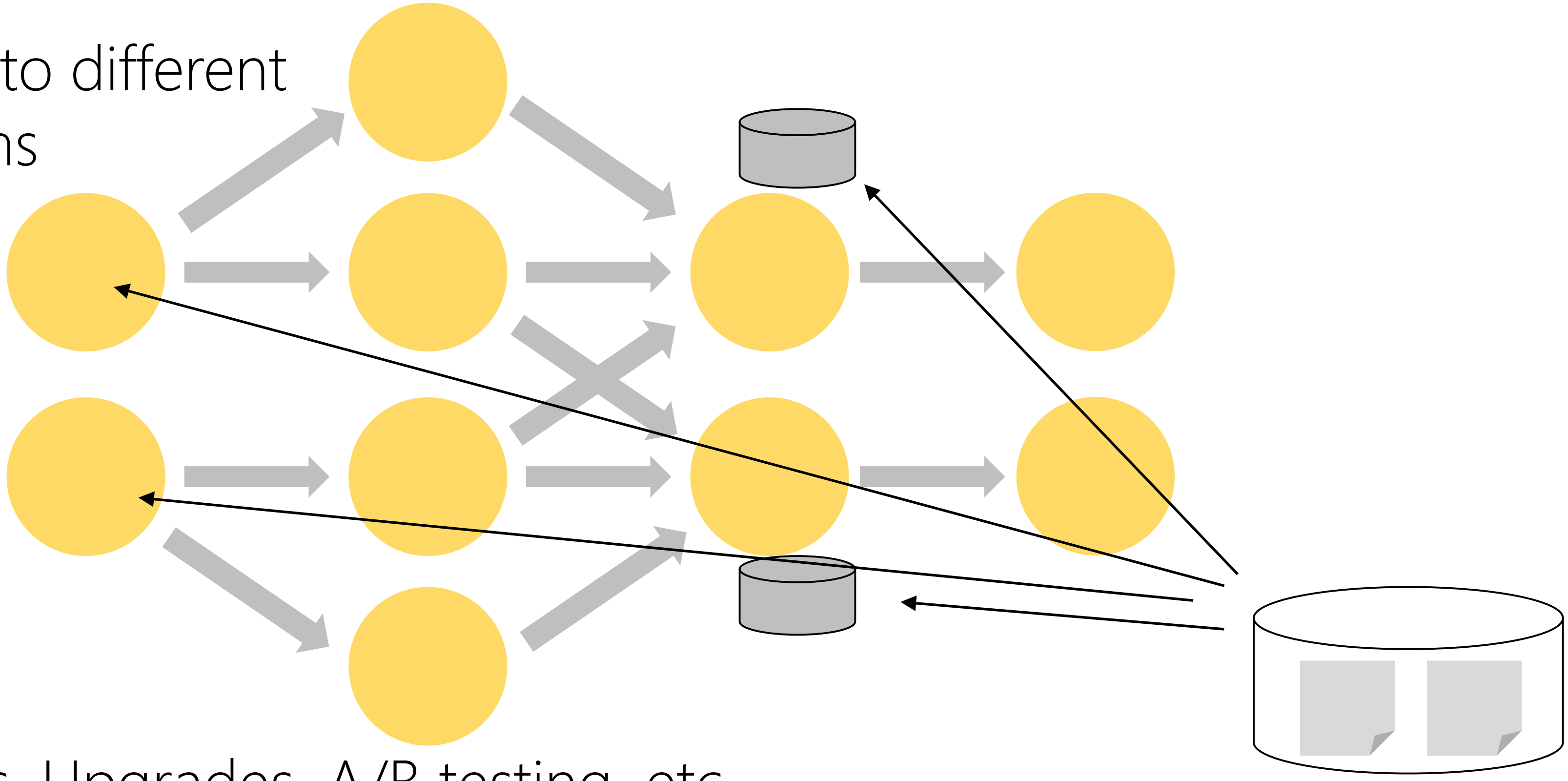
Rolling back computation

Re-processing



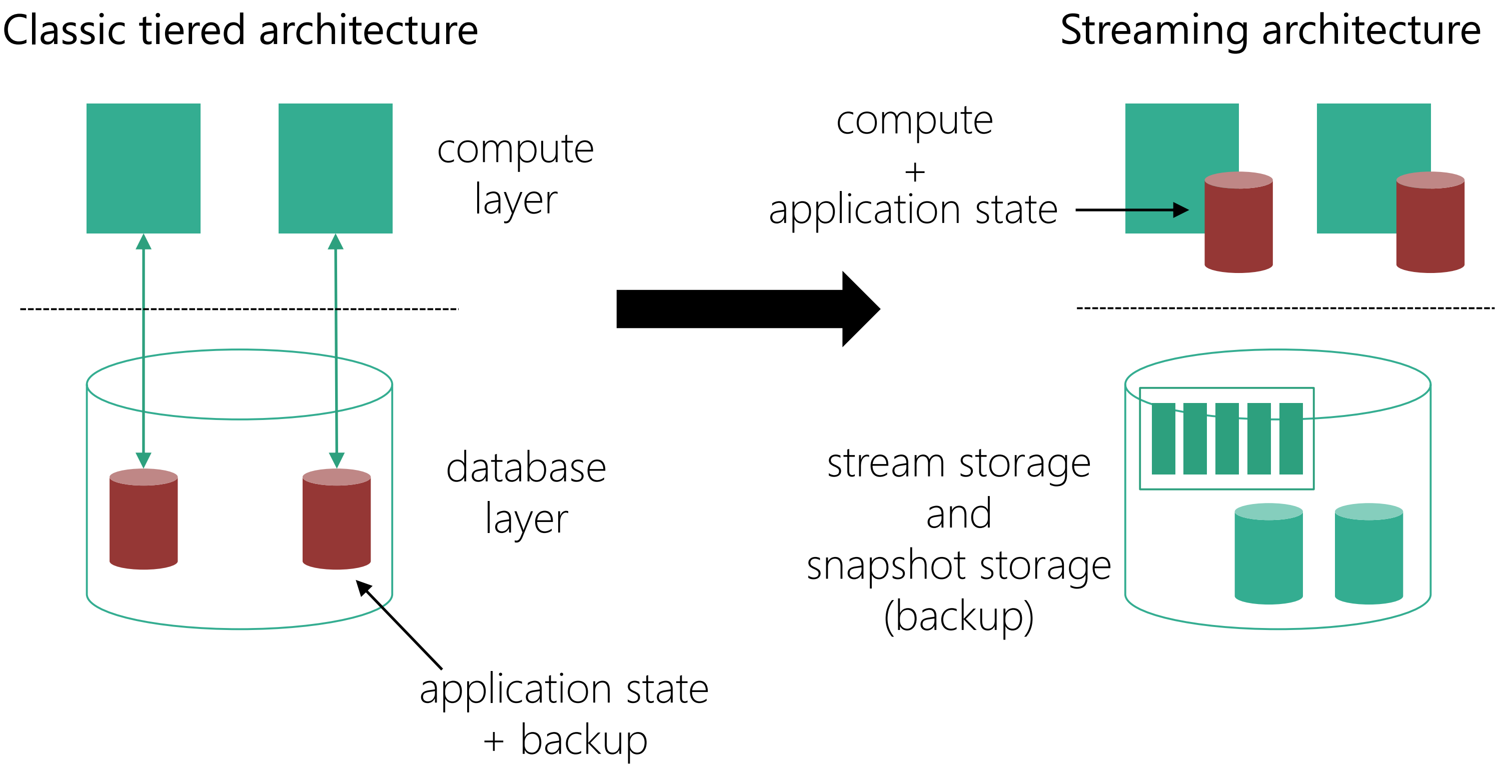
Stateful Event & Stream Processing

Restore to different programs



Bugfixes, Upgrades, A/B testing, etc

Compute, State, and Storage



System for Event–driven Applications

Stateful, event-driven,
event-time-aware processing

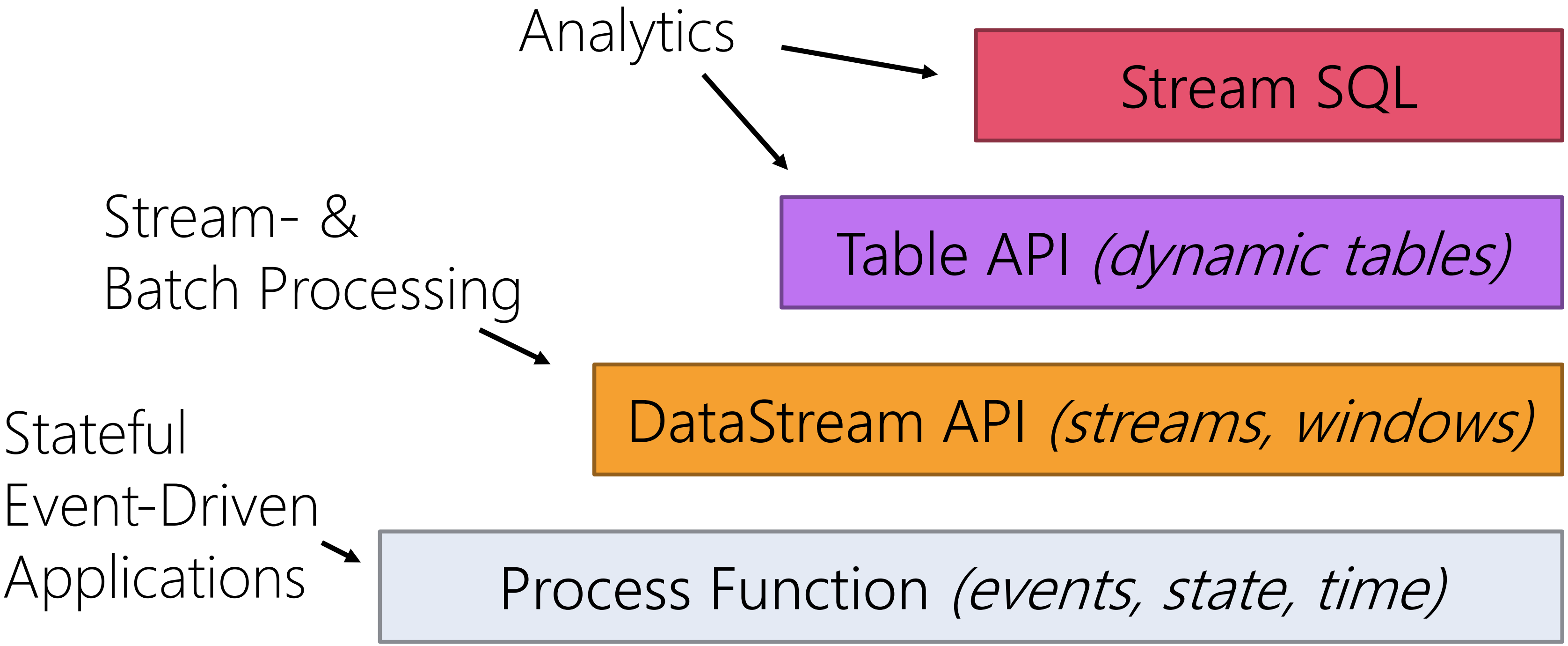
Event-driven
Applications

(event sourcing, CQRS, …)

Stream Processing
(streams, windows, …)

Batch Processing
(data sets)

Apache Flink's Layered APIs



Lessons Learned from Running Flink

The event/stream pipeline generally just works

☺

Interacting with the environment

- Dependency conflicts are amongst the biggest problems
  - Next versions trying to radically reduce dependencies
  - Make Hadoop an optional dependency
  - Rework shading techniques

- The deployment ecosystem is crazy complex
  - Yarn, Mesos & DC/OS, Docker & K8s, standalone, …
  - Containers and overlay networks are tricky
  - Authorization and authentication ecosystem complex it itself
  - Continuous work to improve integration

External systems

- Dependency on any external system eventually causes downtime
  - Mainly: HDFS / S3 / NFS / … for checkpoints

- We plan to reduce dependency on those more and more in the next versions

Type Serialization

- Type serialization is a harder problem in streaming than in batch
  - The data structure updates require more serialization
  - Types are often more complex than in batch

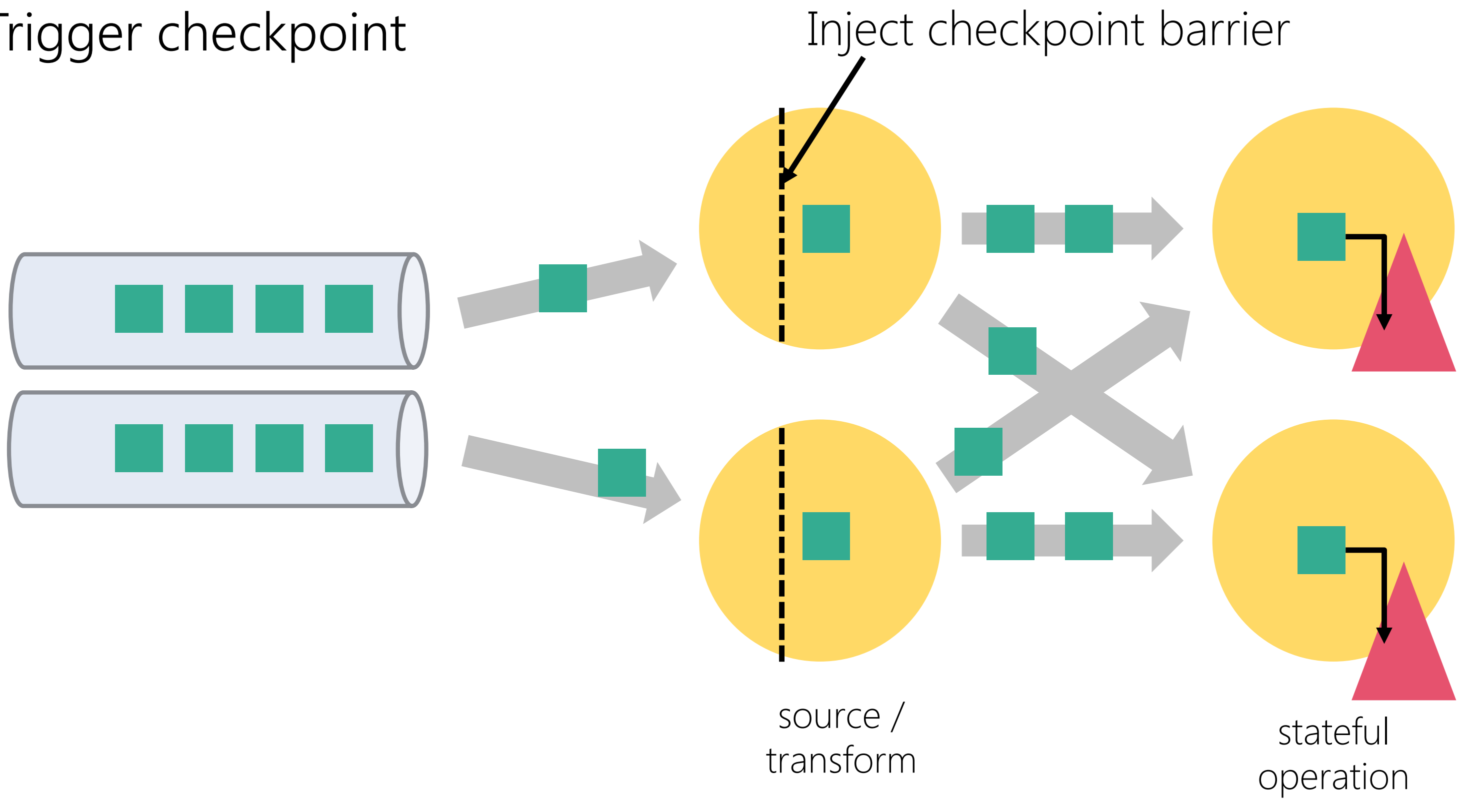
- State lives long and across jobs
  - Requires to "version" state and serializers
  - Requires a "schema evolution" path
  - Much enhanced support in Flink 1.3, more still to come

Robustly checkpointing…

…is the most important part of
running a large scale Flink application

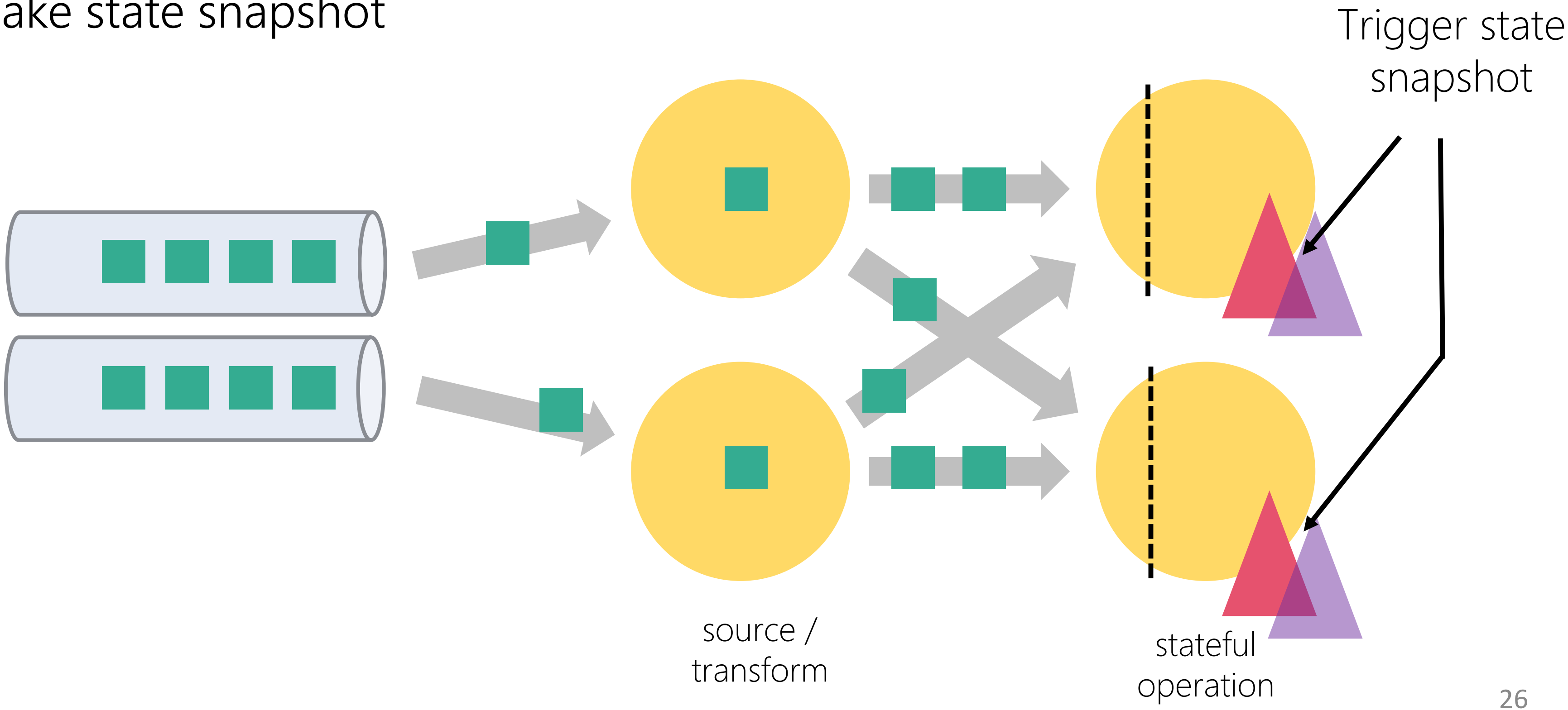
Review: Checkpoints

Trigger checkpoint

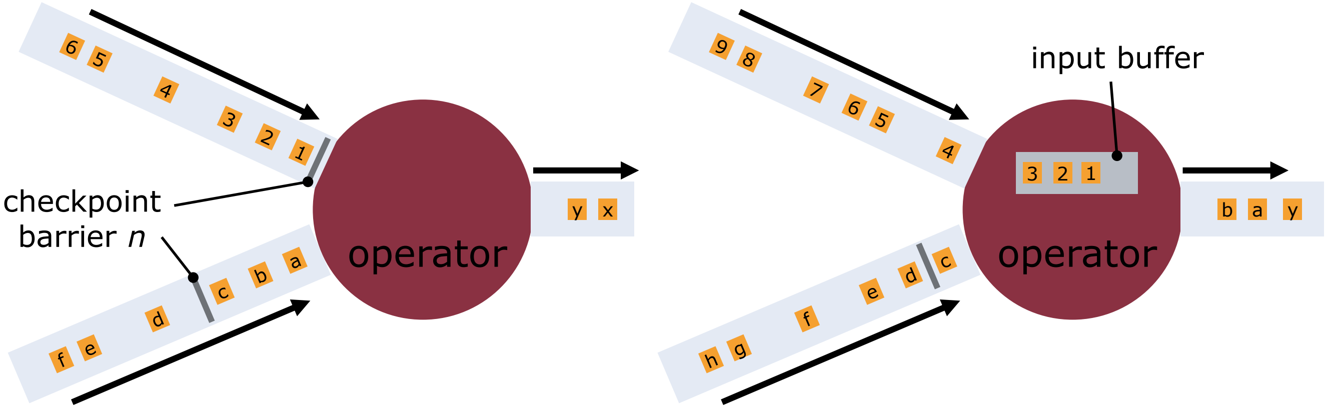


Review: Checkpoints

Take state snapshot



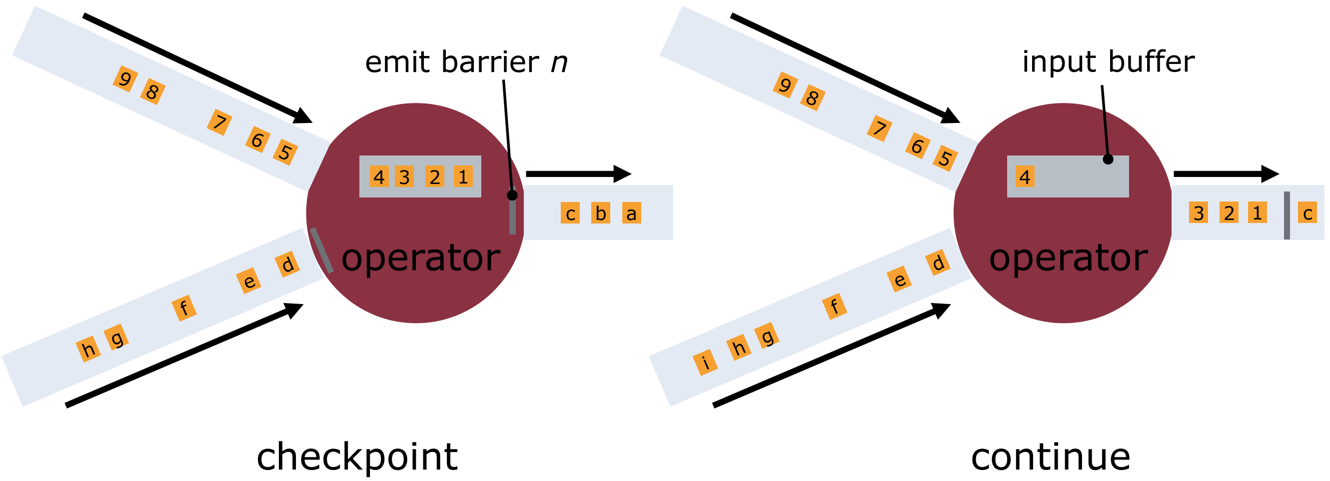
Review: Checkpoint Alignment



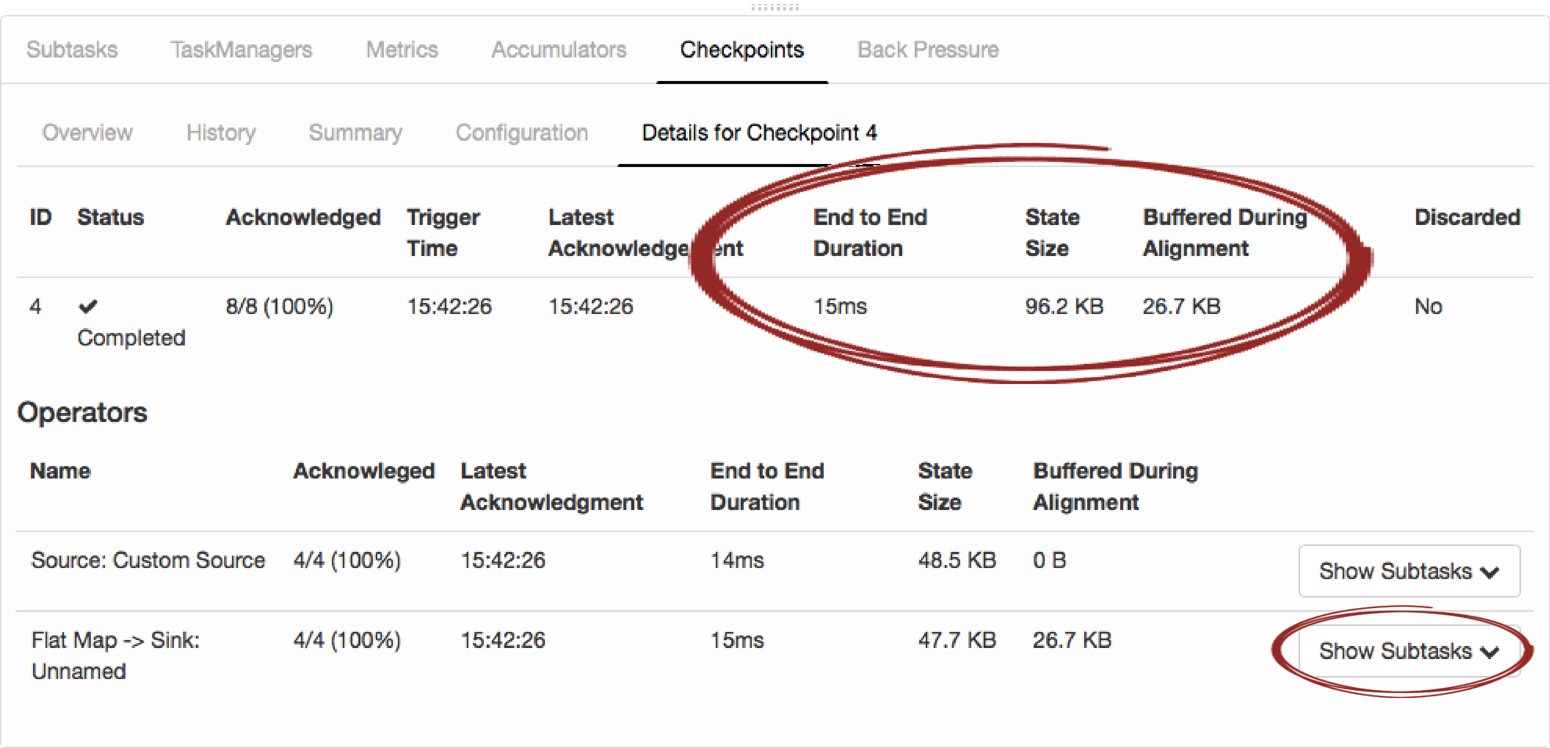
begin aligning

aligning

Review: Checkpoint Alignment



Understanding Checkpoints



Subtasks | TaskManagers | Metrics | Accumulators | Checkpoints | Back Pressure

Overview | History | Summary | Configuration | Details for Checkpoint 4

ID	Status	Acknowledged	Trigger Time	Latest Acknowledgement	End to End Duration	State Size	Buffered During Alignment	Discarded
4	✔ Completed	8/8 (100%)	15:42:26	15:42:26	15ms	96.2 KB	26.7 KB	No

Operators

Name	Acknowleged	Latest Acknowledgment	End to End Duration	State Size	Buffered During Alignment	
Source: Custom Source	4/4 (100%)	15:42:26	14ms	48.5 KB	0 B	Show Subtasks
Flat Map -> Sink: Unnamed	4/4 (100%)	15:42:26	15ms	47.7 KB	26.7 KB	Show Subtasks

Understanding Checkpoints

delay =
end_to_end – sync – async

How long do snapshots take?

How well behaves the alignment?
(lower is better)

Source: Custom Source	4/4 (100%)	15:42:26	14ms	48.5 KB	0 B	Hide Subtasks

	End to End Duration	State Size	Checkpoint Duration (Sync)	Checkpoint Duration (Async)	Alignment Buffered	Alignment Duration
Minimum	8ms	11.9 KB	0ms	0ms	0 B	0ms
Average	10ms	12.1 KB	0ms	0ms	0 B	0ms
Maximum	14ms	12.3 KB	0ms	1ms	0 B	0ms

Understanding Checkpoints

delay =
end_to_end – sync – async

How long do
snapshots take?

How well behaves
the alignment?
(lower is better)

long delay = under backpressure

under constant backpressure
means the application is
under provisioned

too long means
→ too much state
per node
→ snapshot store cannot
keep up with load
(low bandwidth)

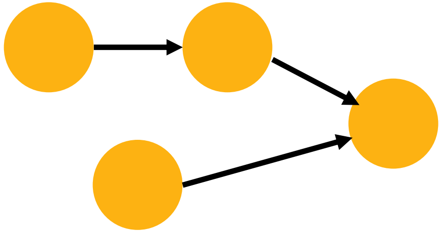
most important
robustness metric

vastly improved with
incremental checkpoints in Flink 1.3

Heavy alignments

- A heavy alignment typically happens at some point
  → Different load on different paths

- Skewed window emission
  (lots of data on one node)

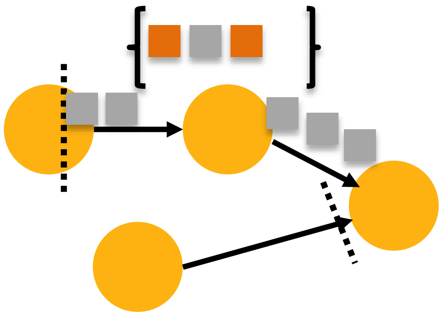


- Stall of one operator on the path

Heavy alignments

- A heavy alignment typically happens at some point
  → Different load on different paths

- Skewed window emission
  (lots of data on one node)



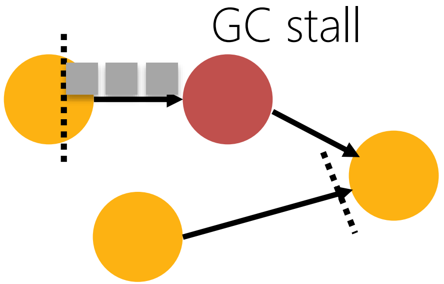
- Stall of one operator on the path

Heavy alignments

- A heavy alignment typically happens at some point
  → Different load on different paths

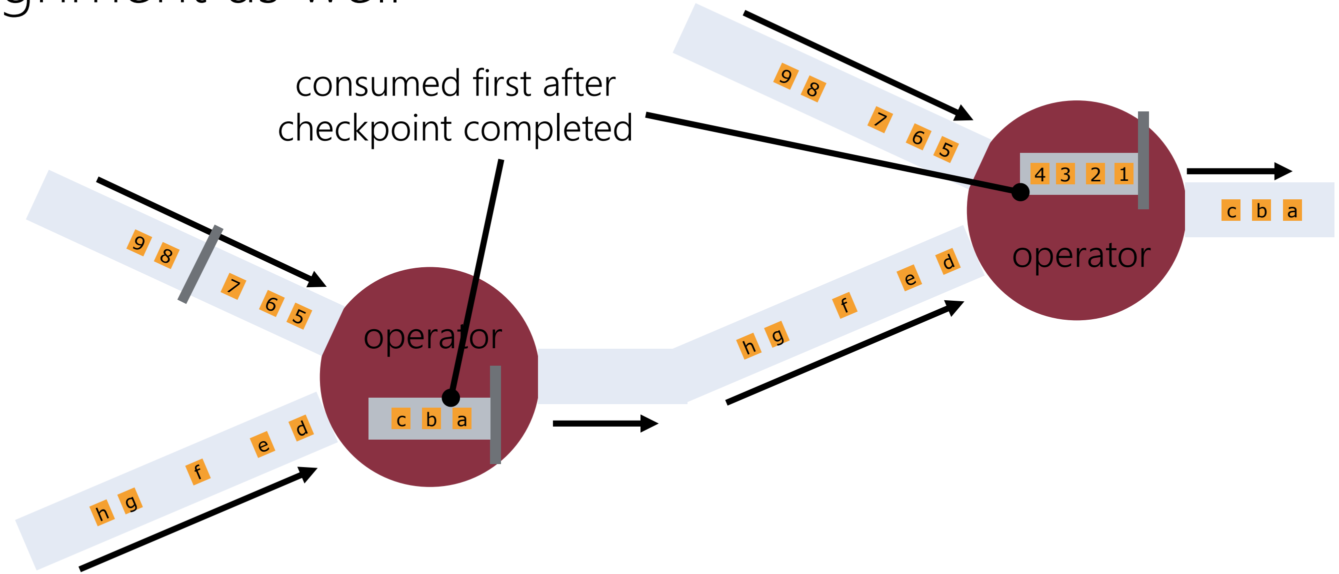
- Skewed window emission
  (lots of data on one node)

- Stall of one operator on the path



Catching up from heavy alignments

- Operators that did heavy alignment need to catch up again
- Otherwise, next checkpoint will have a heavy alignment as well



Catching up from heavy alignments

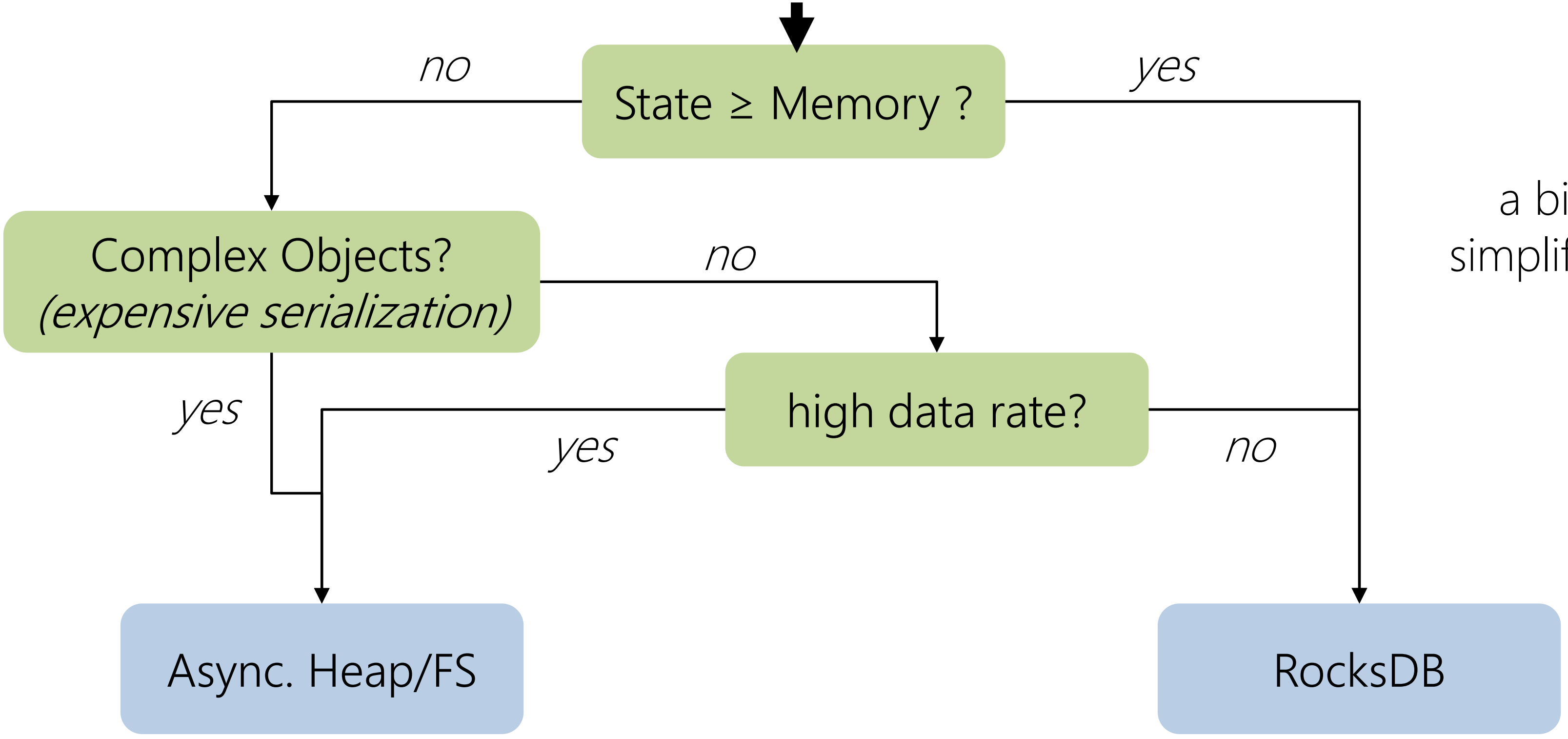
- Giving the computation time to catch up before starting the next checkpoint
  - Set the min-time-between-checkpoints
  - Ideas to change checkpoints to policy based (spend x% of capacity on checkpoints)

- Asynchronous checkpoints mitigate most of problem
  - Very short stalls in the pipelines means shorter alignment phase
  - Catch up already happens concurrently to state materialization

Asynchrony of different state types

State	Flink 1.2	Flink 1.3	Flink 1.4
Keyed state RocksDB	✓	✓	✓
Keyed State on heap	✗ (✓) (hidden in 1.2.1)	✓	✓
Timers	✗	✗	✓ (PR)
Operator State	✗	✓	✓

When to use which state backend?



a bit
simplified


FLINK FORWARD
Berlin
11-13 Sep 2017
Flink Forward, the premier conference on Apache Flink®, is coming back to Berlin
Call for Submissions is open

dataArtisans

We are hiring!

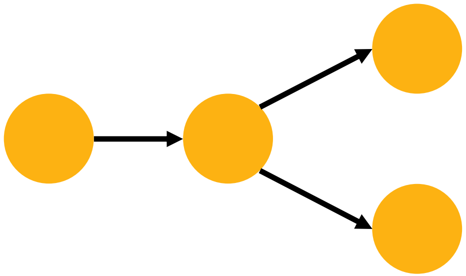
data-artisans.com/careers

Backup Slides

Avoiding DDOSing other systems

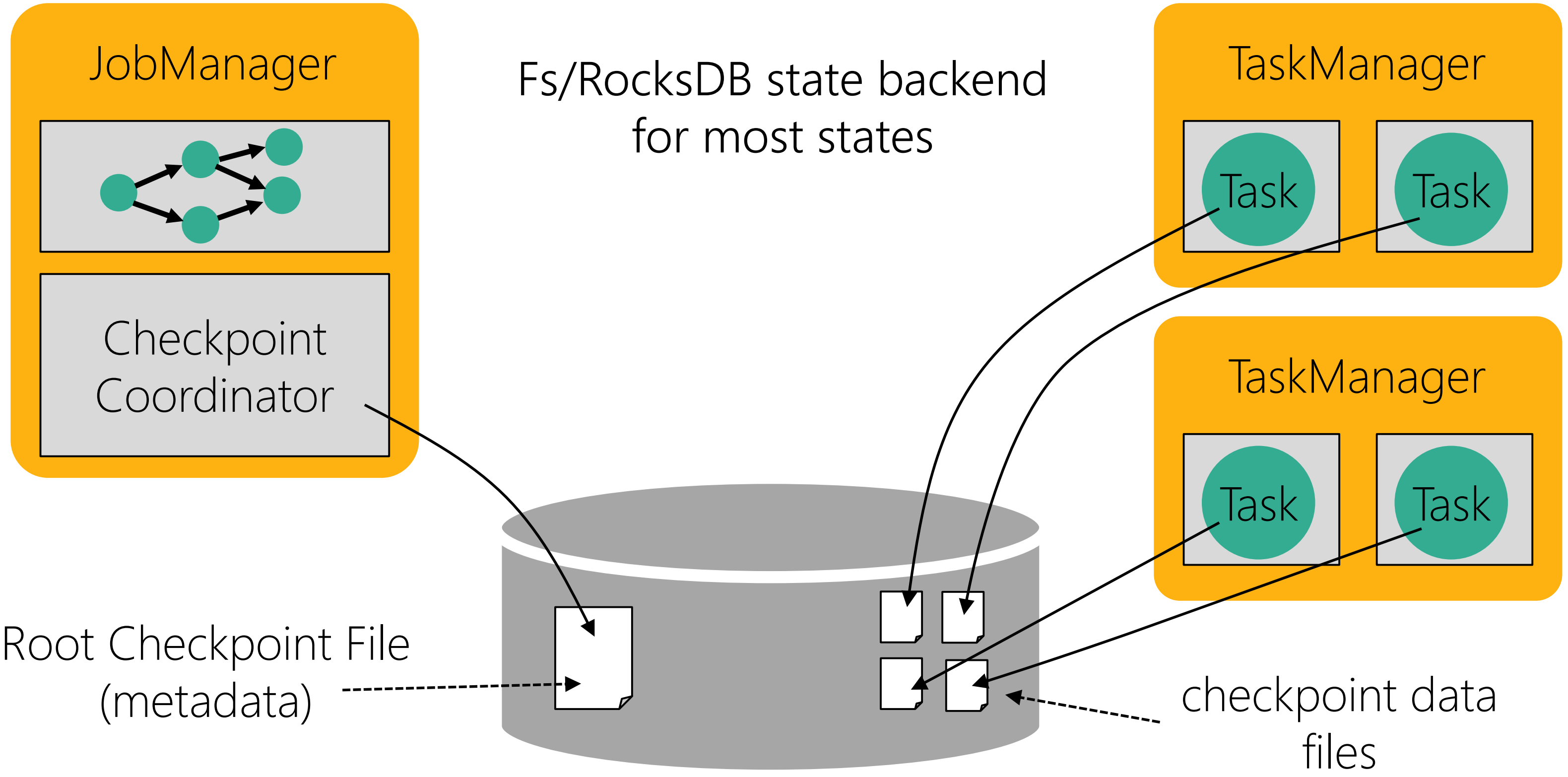
Exceeding FS request capacity

- Job size: multiple 1000 operators
- Checkpoint interval: few secs

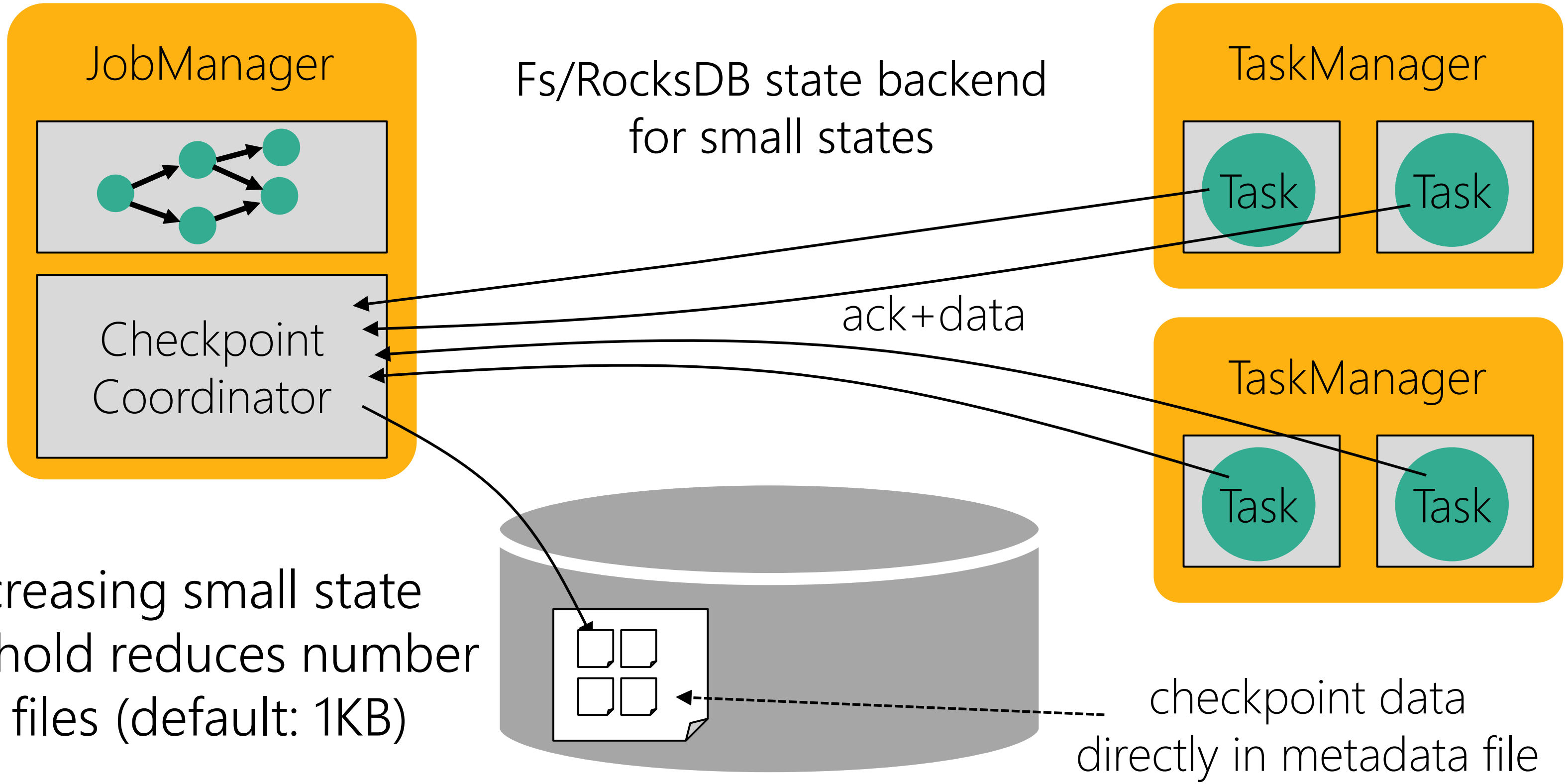


- State size: KBs per operator, 1000 of state chunks
- Via the S3 FS (from Hadoop), writes ensure "directory" exists, 2 HEAD requests
- Symptom: S3 blocked off connections after exceeding 1000s HEAD requests / sec

Reducing FS stress for small state



Reducing FS stress for small state



Increasing small state threshold reduces number of files (default: 1KB)

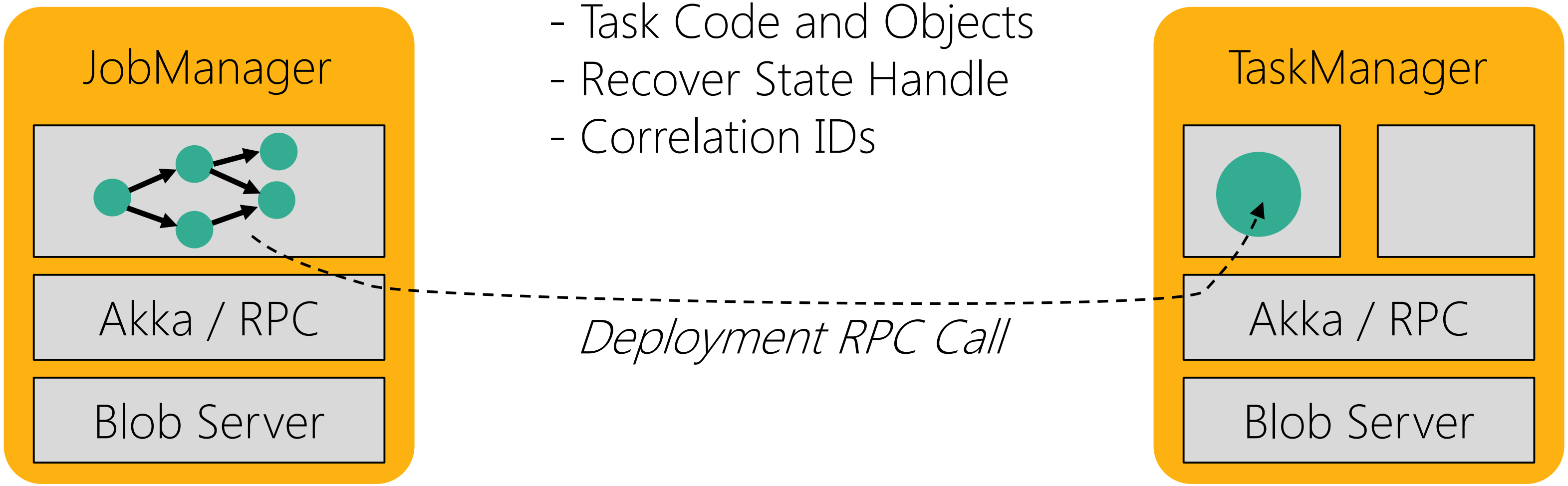
Distributed Coordination

Deploying Tasks

Happens during initial deployment and recovery

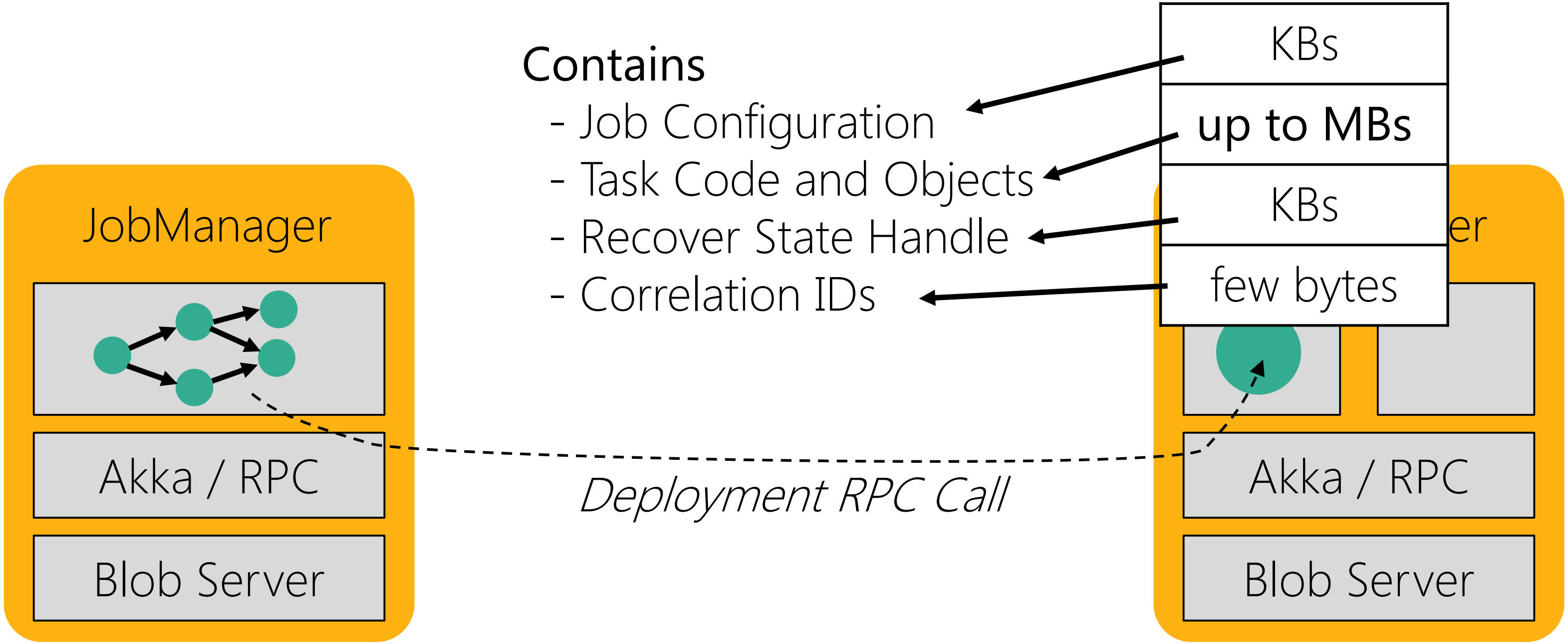
Contains

- Job Configuration
- Task Code and Objects
- Recover State Handle
- Correlation IDs



Deploying Tasks

Happens during initial deployment and recovery



RPC volume during deployment

(back of the napkin calculation)

number of tasks	x	parallelism	x	size of task objects	=	RPC volume
10	x	1000	x	2 MB	=	20 GB

~20 seconds on full 10 GBits/s net

> 1 min with avg. of 3 GBits/s net

> 3 min with avg. of 1GBs net

Timeouts and Failure detection

~20 seconds on full 10 GBits/s net

> 1 min with avg. of 3 GBits/s net

> 3 min with avg. of 1GBs net

Default RPC timeout: **10 secs**

default settings lead to **failed deployments with RPC timeouts**

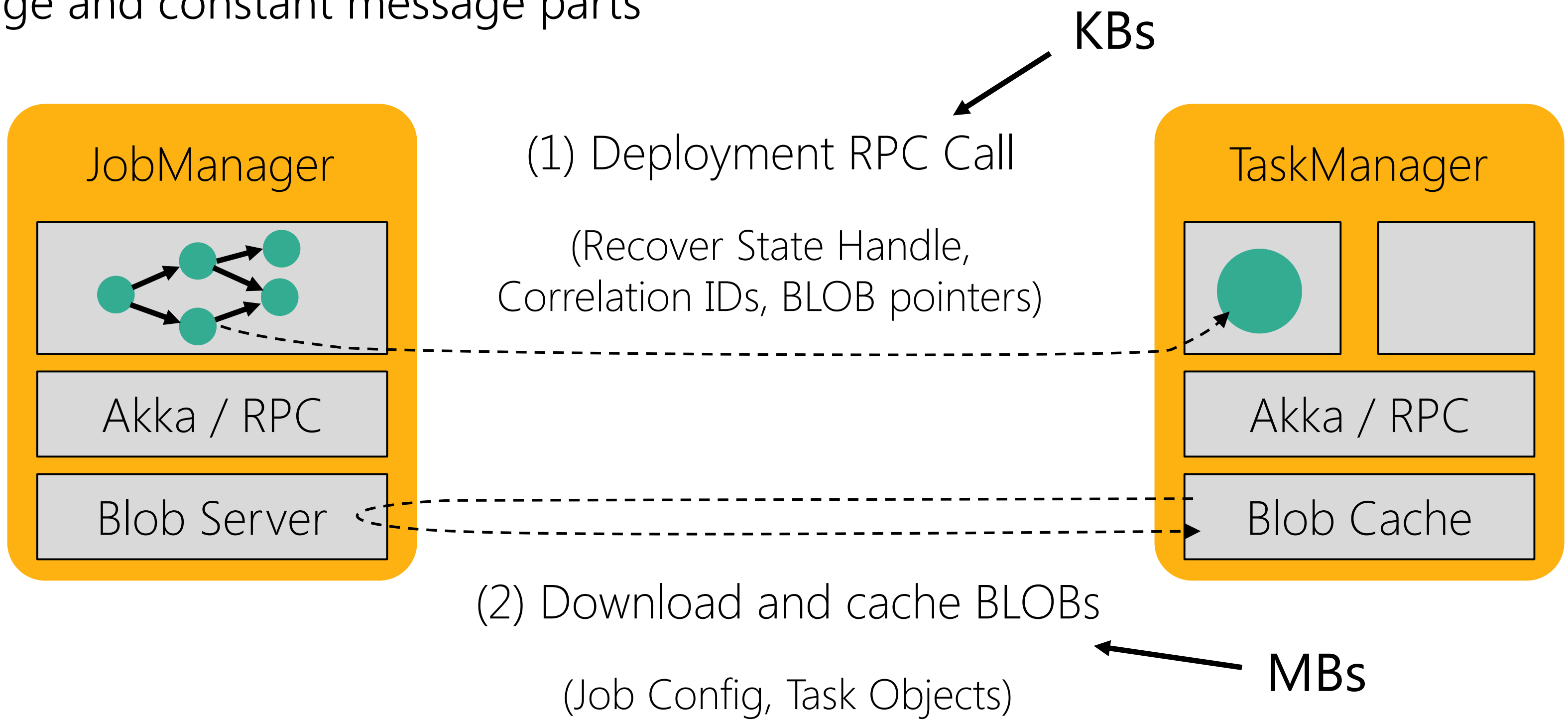
Solution: Increase RPC timeout
Caveat: Increasing the timeout makes failure detection slower
Future: Reduce RPC load (next slides)

Dissecting the RPC messages

Message part	Size	Variance across subtasks and redeploys
Job Configuration	KBs	constant
Task Code and Objects	**up to MBs**	constant
Recover State Handle	KBs	variable
Correlation IDs	few bytes	variable

Upcoming: Deploying Tasks

Out-of-band transfer and caching of large and constant message parts

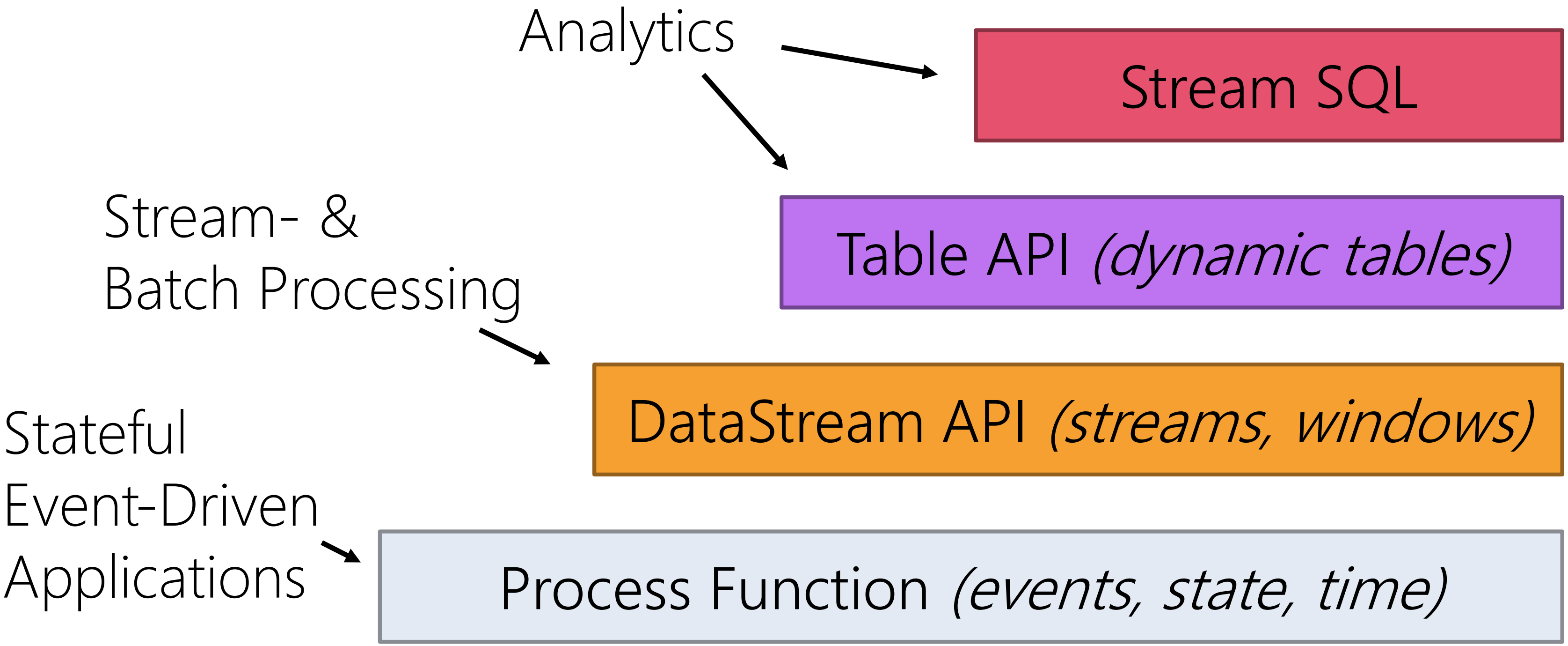


Ogres have layers

So do squirrels

Layers of abstraction

Apache Flink's Layered APIs



Process Function

```
class MyFunction extends ProcessFunction[MyEvent, Result] {

    // declare state to use in the program
    lazy val state: ValueState[CountWithTimestamp] = getRuntimeContext().getState(…)

    def processElement(event: MyEvent, ctx: Context, out: Collector[Result]): Unit = {
        // work with event and state
        (event, state.value) match { … }

        out.collect(…) // emit events
        state.update(…) // modify state

        // schedule a timer callback
        ctx.timerService.registerEventTimeTimer(event.timestamp + 500)
    }

    def onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[Result]): Unit = {
        // handle callback when event-/processing- time instant is reached
    }
}
```

Data Stream API

```
val lines: DataStream[String] = env.addSource(
                                      new FlinkKafkaConsumer09<>(…))

val events: DataStream[Event] = lines.map((line) => parse(line))

val stats: DataStream[Statistic] = stream
    .keyBy("sensor")
    .timeWindow(Time.seconds(5))
    .sum(new MyAggregationFunction())

stats.addSink(new RollingSink(path))
```

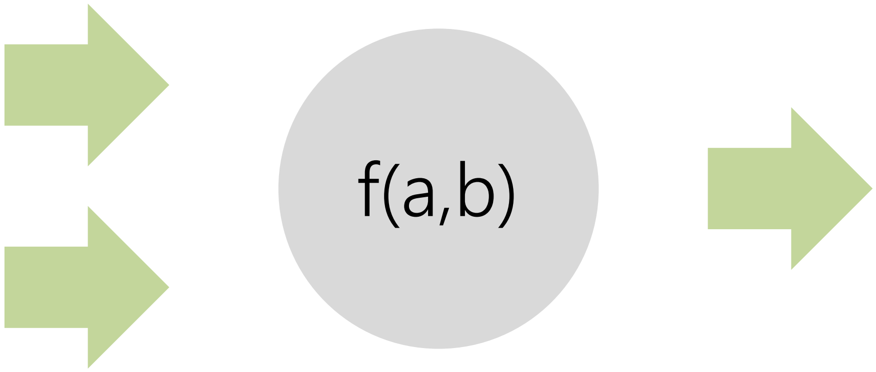
Table API & Stream SQL

```scala
// Table API
val tapiResult: Table = tEnv.scan("sensors")   // scan sensors table
 .window(Tumble over 1.hour on 'rowtime as 'w) // define 1-hour window
 .groupBy('w, 'room)                           // group by window and room
 .select('room, 'w.end, 'temp.avg as 'avgTemp) // compute average temperature
```

```sql
|SELECT room, TUMBLE_END(rowtime, INTERVAL '1' HOUR), AVG(temp) AS avgTemp
|FROM sensors
|GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR), room
```

Events, State, Time, and Snapshots

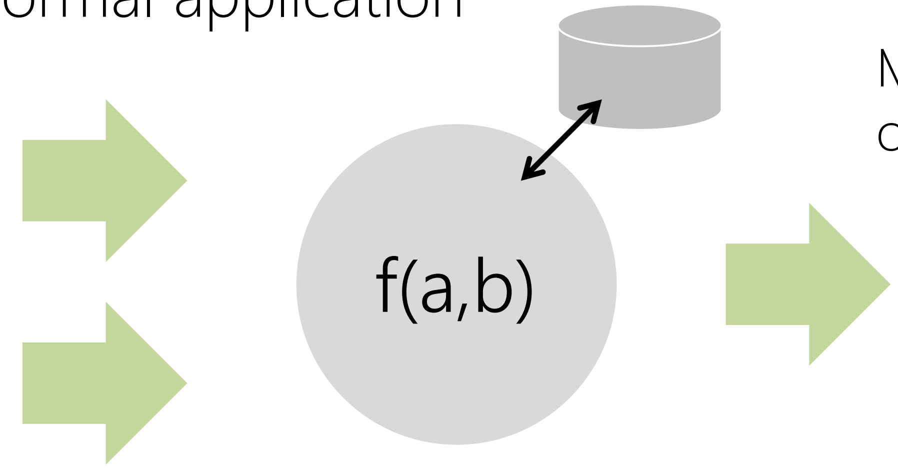
Events, State, Time, and Snapshots



Event-driven function
executed distributedly

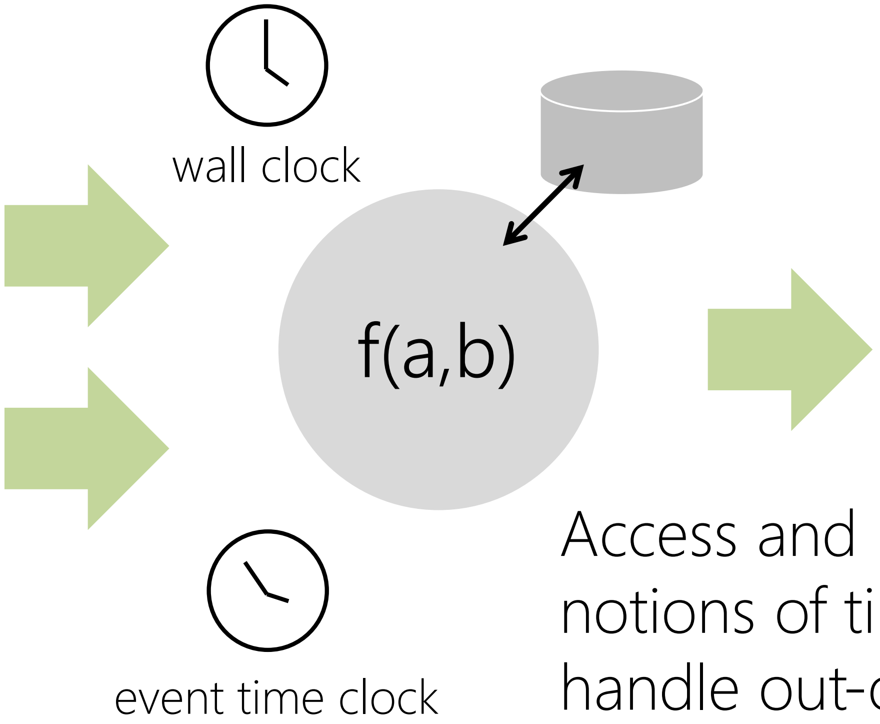
Events, State, Time, and Snapshots

Maintain fault tolerant local state similar to any normal application



Main memory +
out of core (for maps)

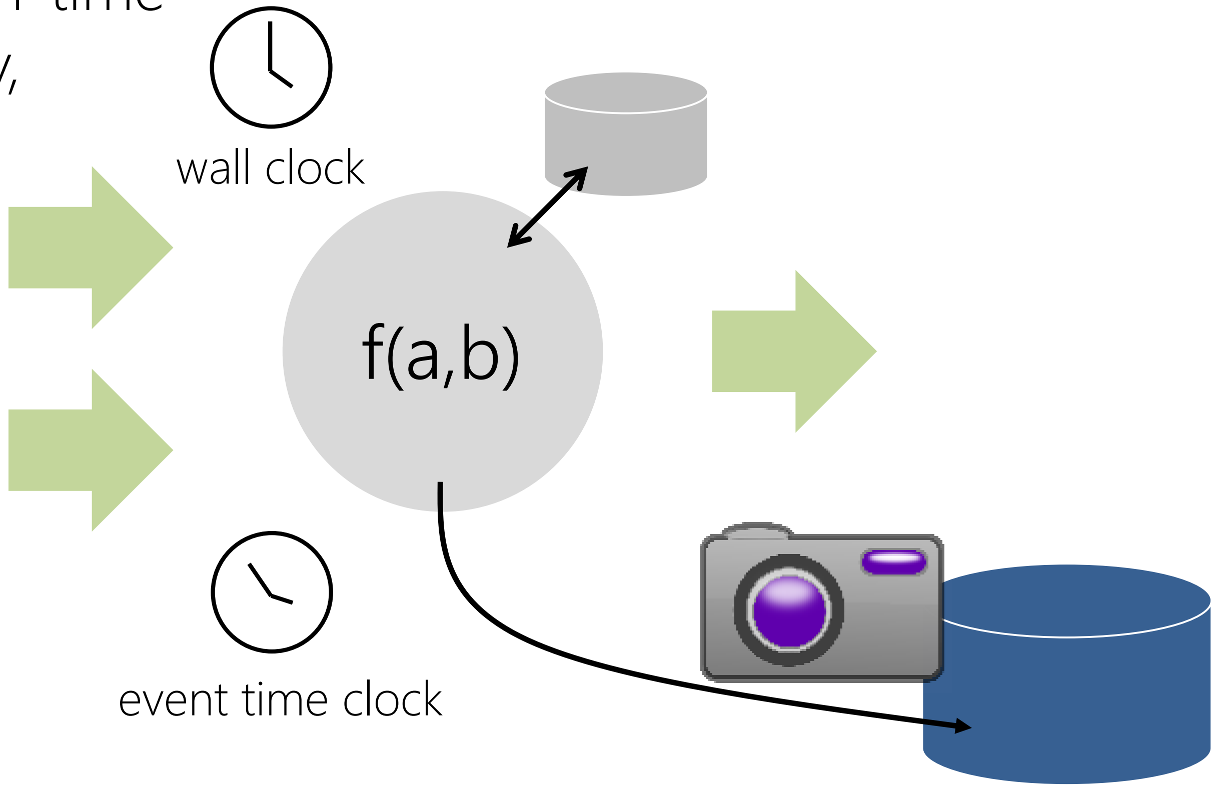
Events, State, Time, and Snapshots



Access and react to notions of time and progress, handle out-of-order events

Events, State, Time, and Snapshots

Snapshot point-in-time view for recovery, rollback, cloning, versioning, etc.



Stateful Event & Stream Processing

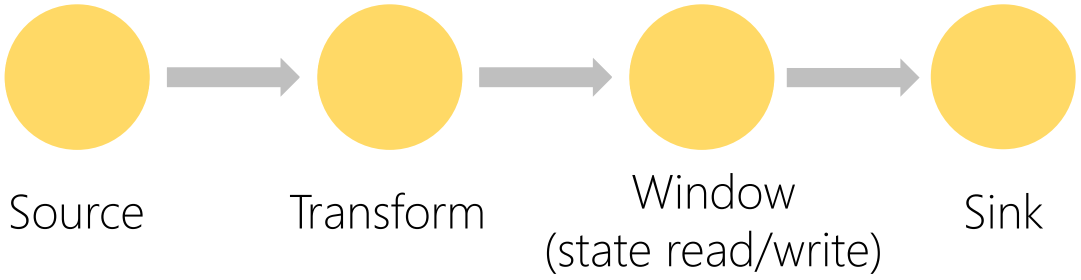
```
val lines: DataStream[String] = env.addSource(new FlinkKafkaConsumer09(…))

val events: DataStream[Event] = lines.map((line) => parse(line))

val stats: DataStream[Statistic] = stream
    .keyBy("sensor")
    .timeWindow(Time.seconds(5))
    .sum(new MyAggregationFunction())

stats.addSink(new RollingSink(path))
```

Source

Transformation

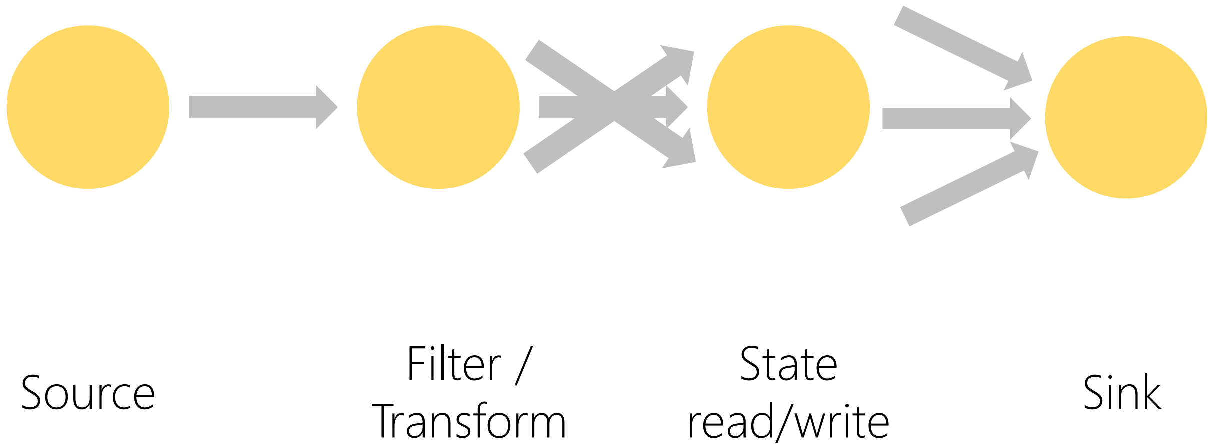
Transformation

Sink

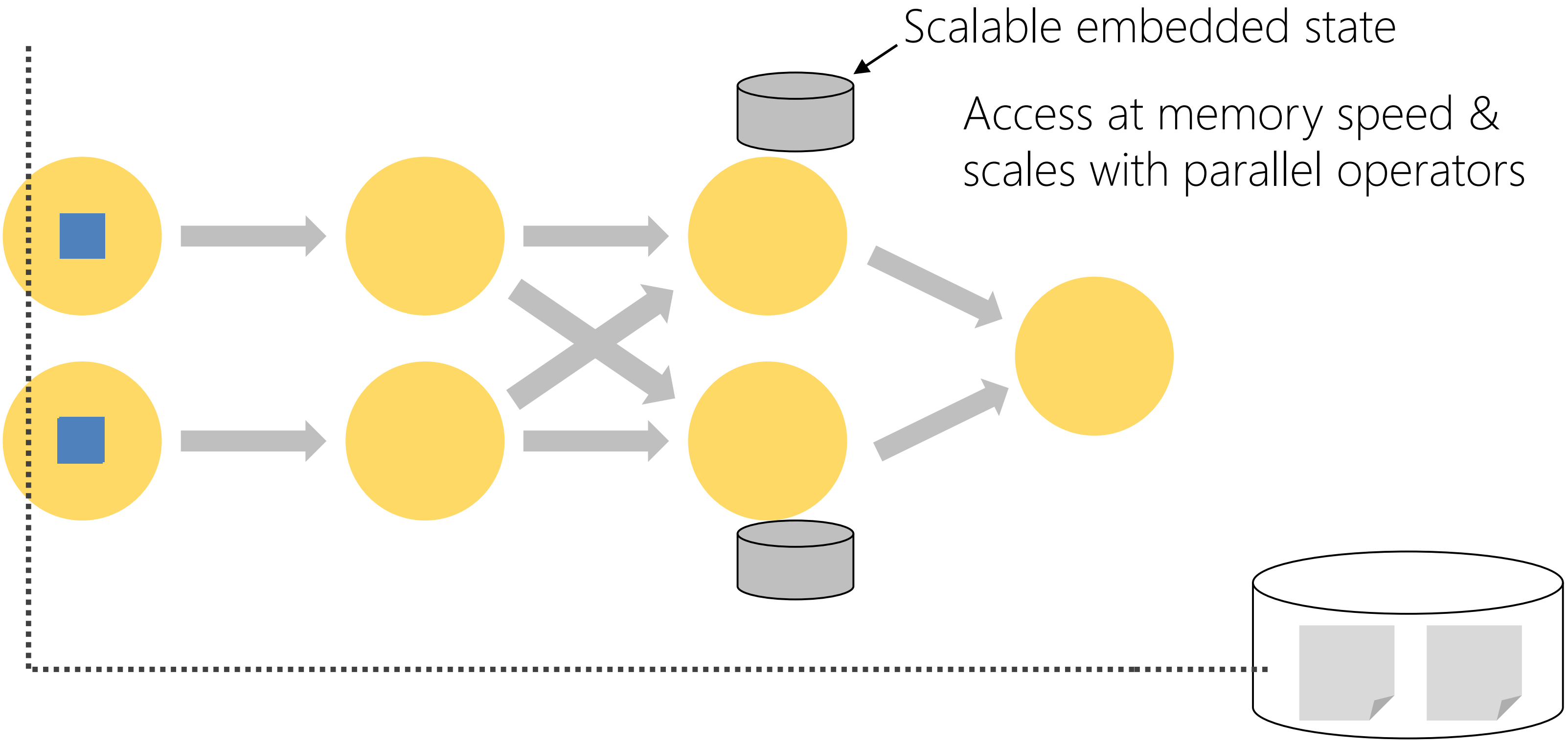


Streaming Dataflow

Stateful Event & Stream Processing



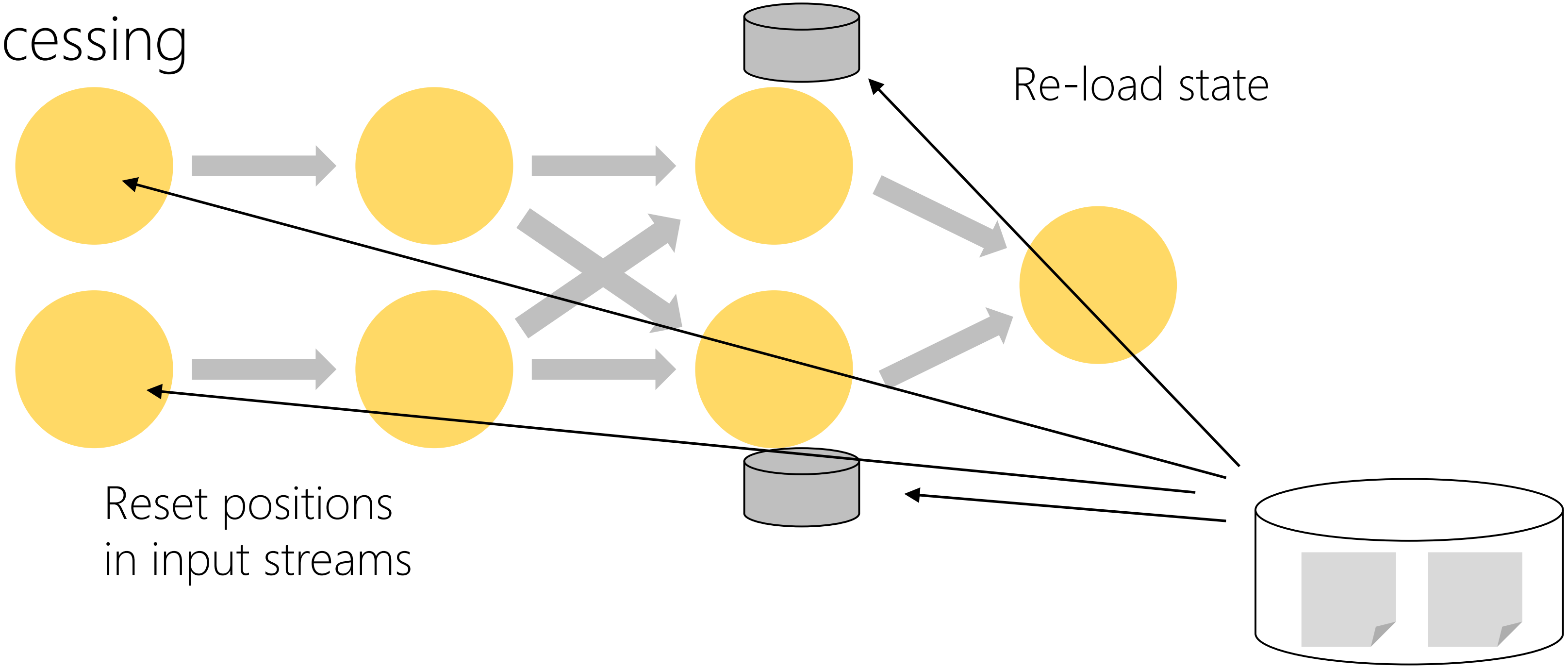
Stateful Event & Stream Processing



Stateful Event & Stream Processing

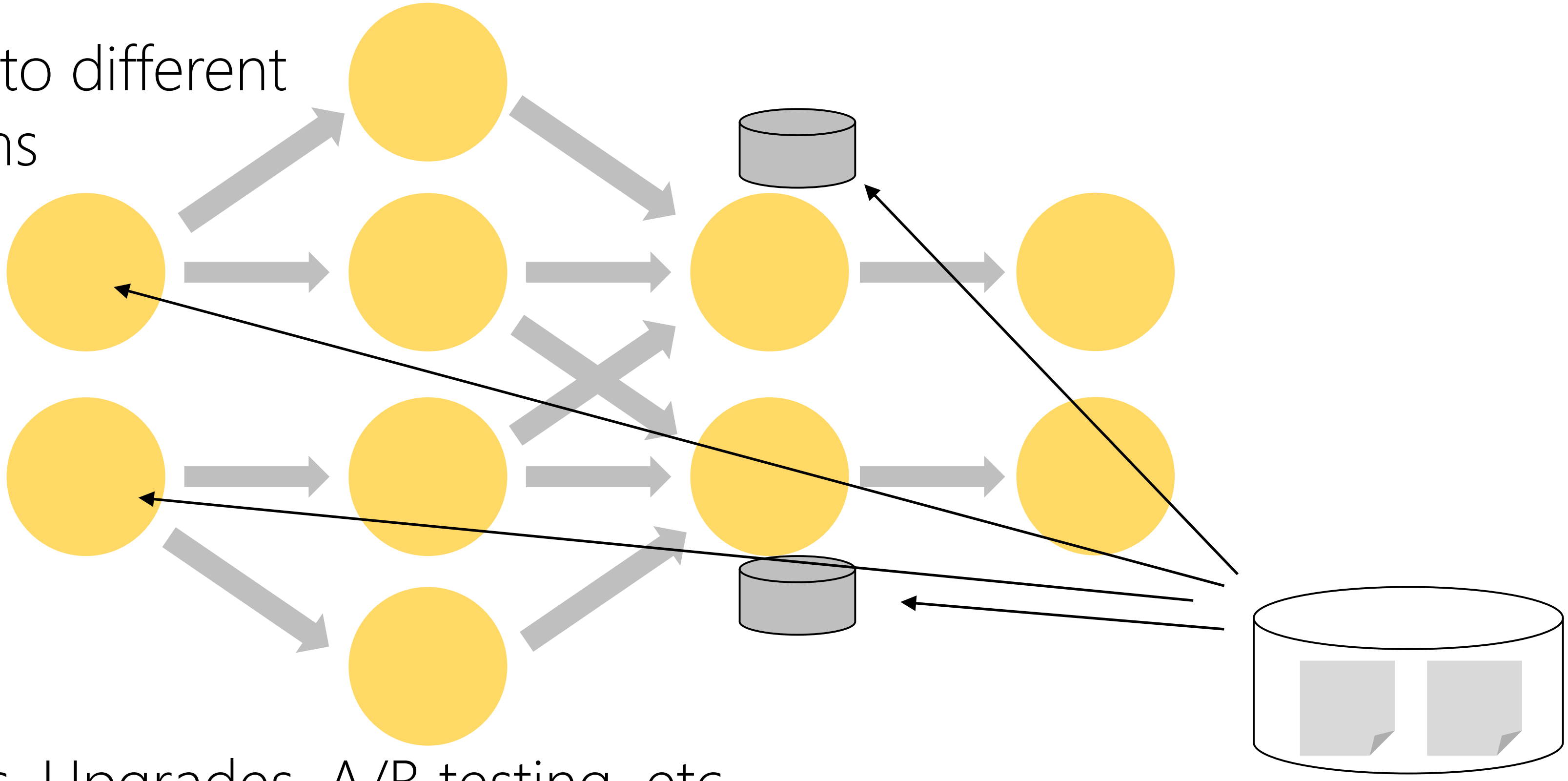
Rolling back computation

Re-processing



Stateful Event & Stream Processing

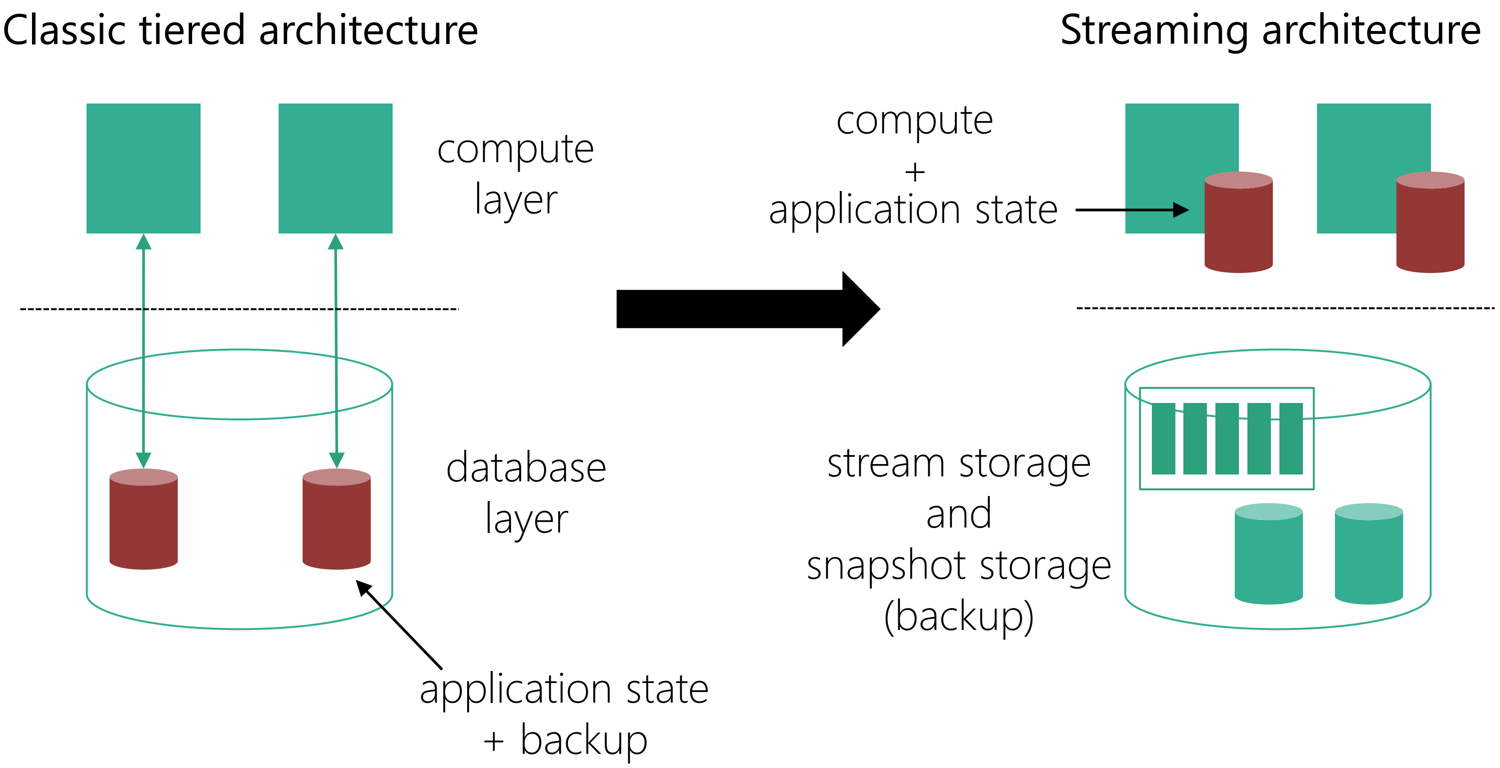
Restore to different programs



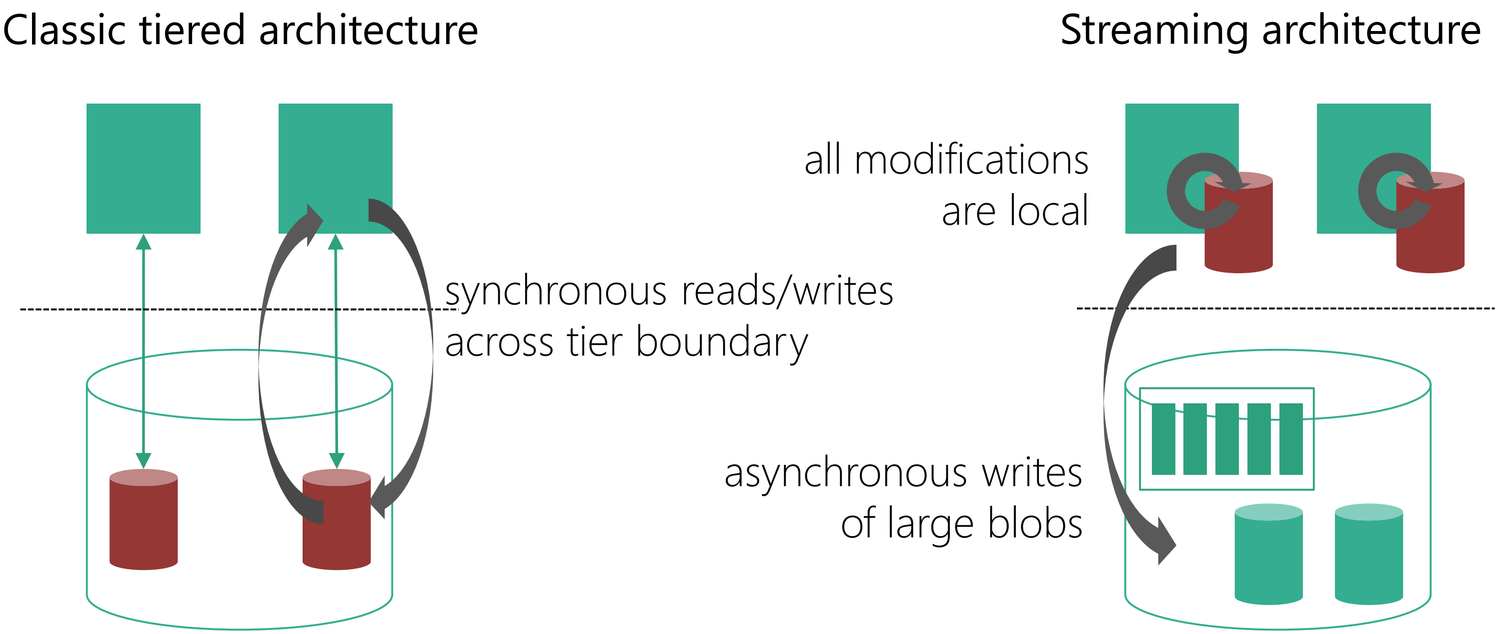
Bugfixes, Upgrades, A/B testing, etc

"Classical" versus Streaming Architecture

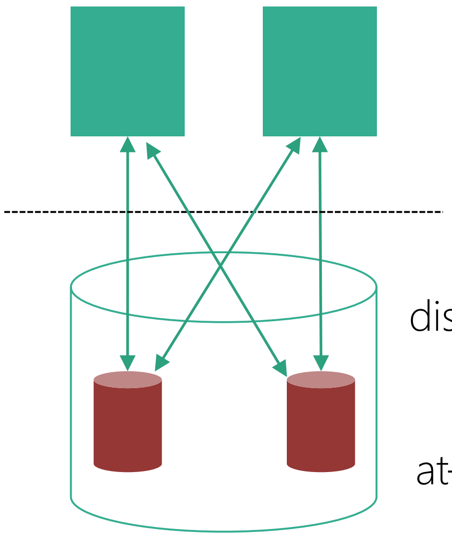
Compute, State, and Storage

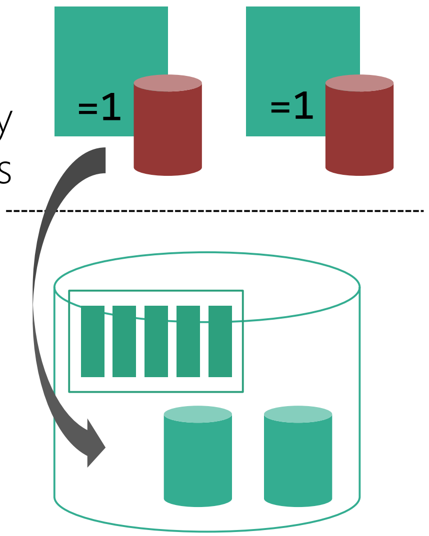


Performance



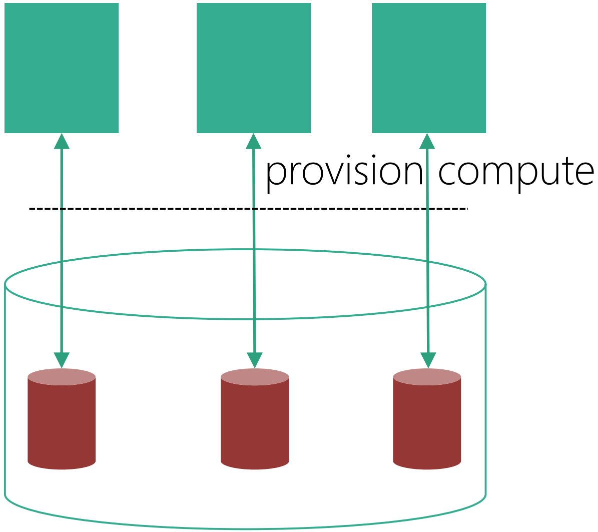
Consistency





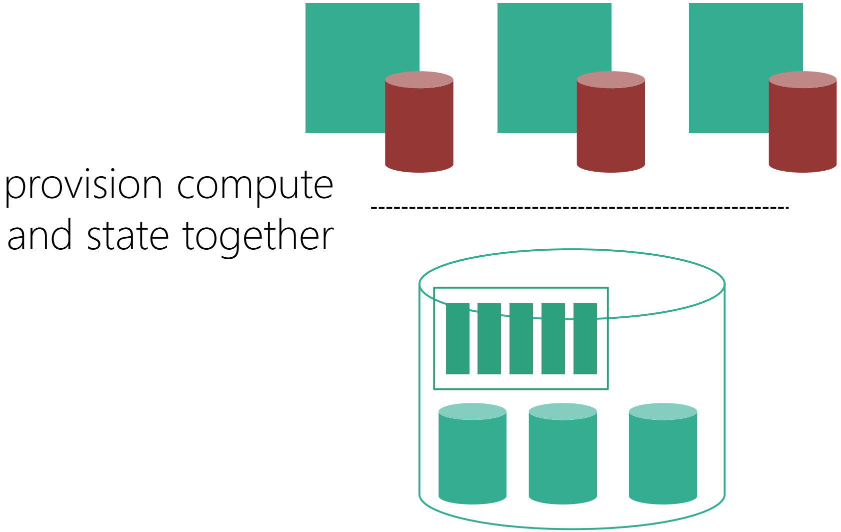
Scaling a Service

Classic tiered architecture



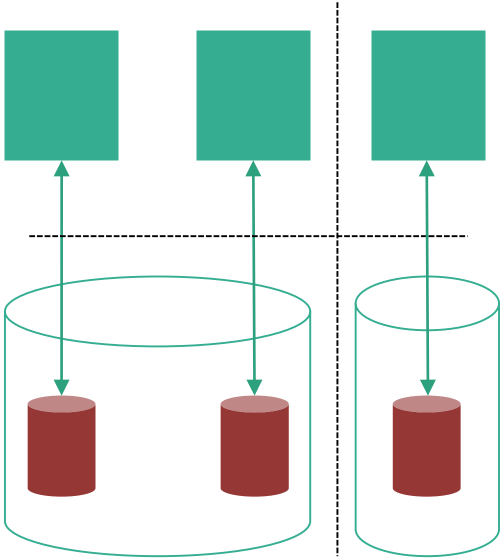
separately provision additional database capacity

Streaming architecture



Rolling out a new Service

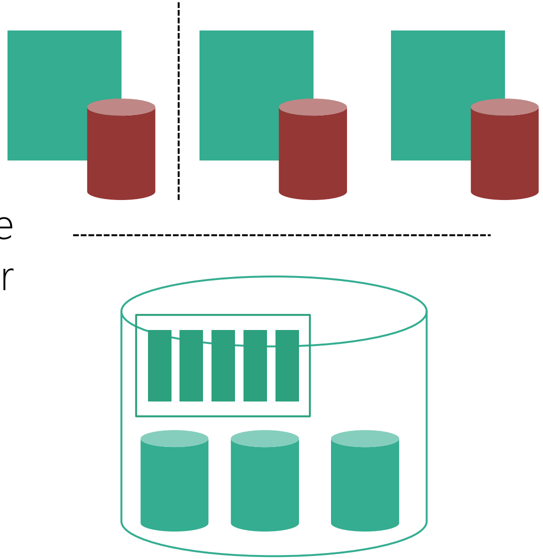
Classic tiered architecture



provision a new database
(or add capacity to an existing one)

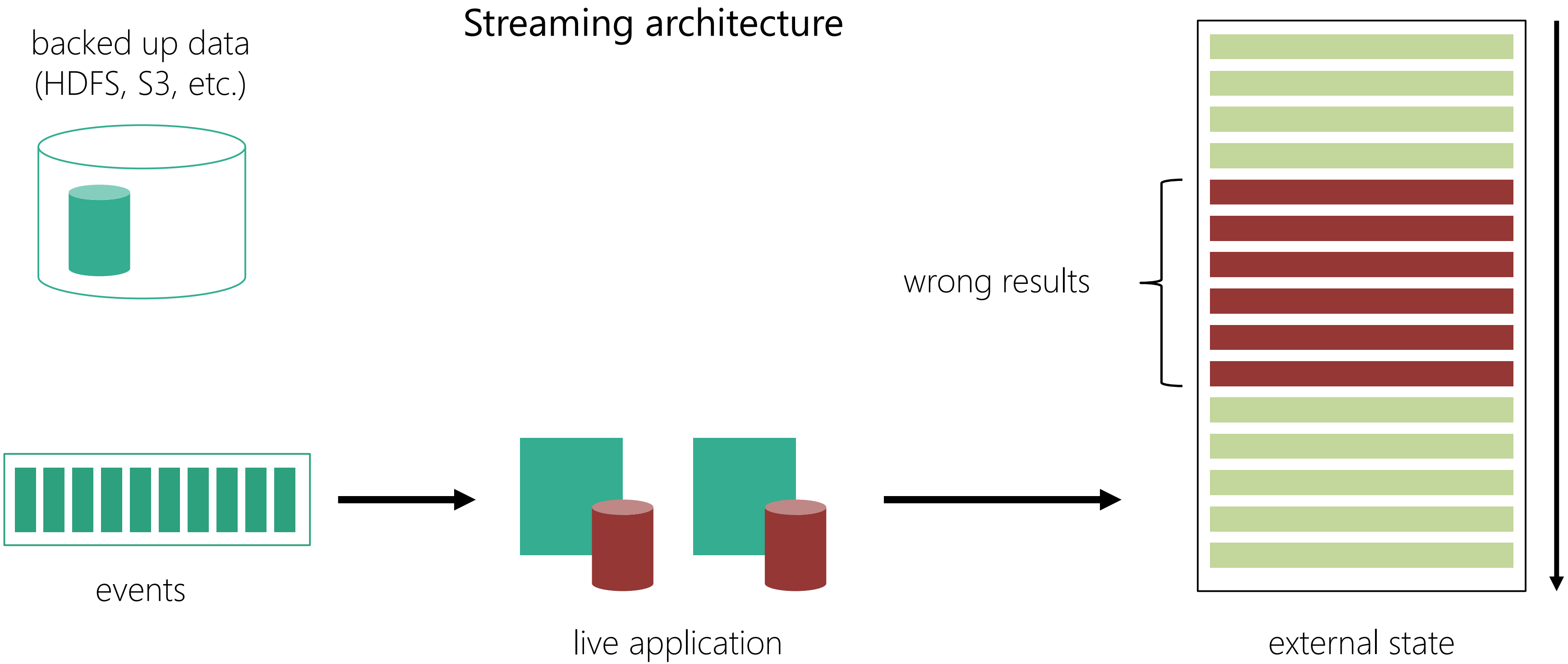
Streaming architecture

provision compute
and state together

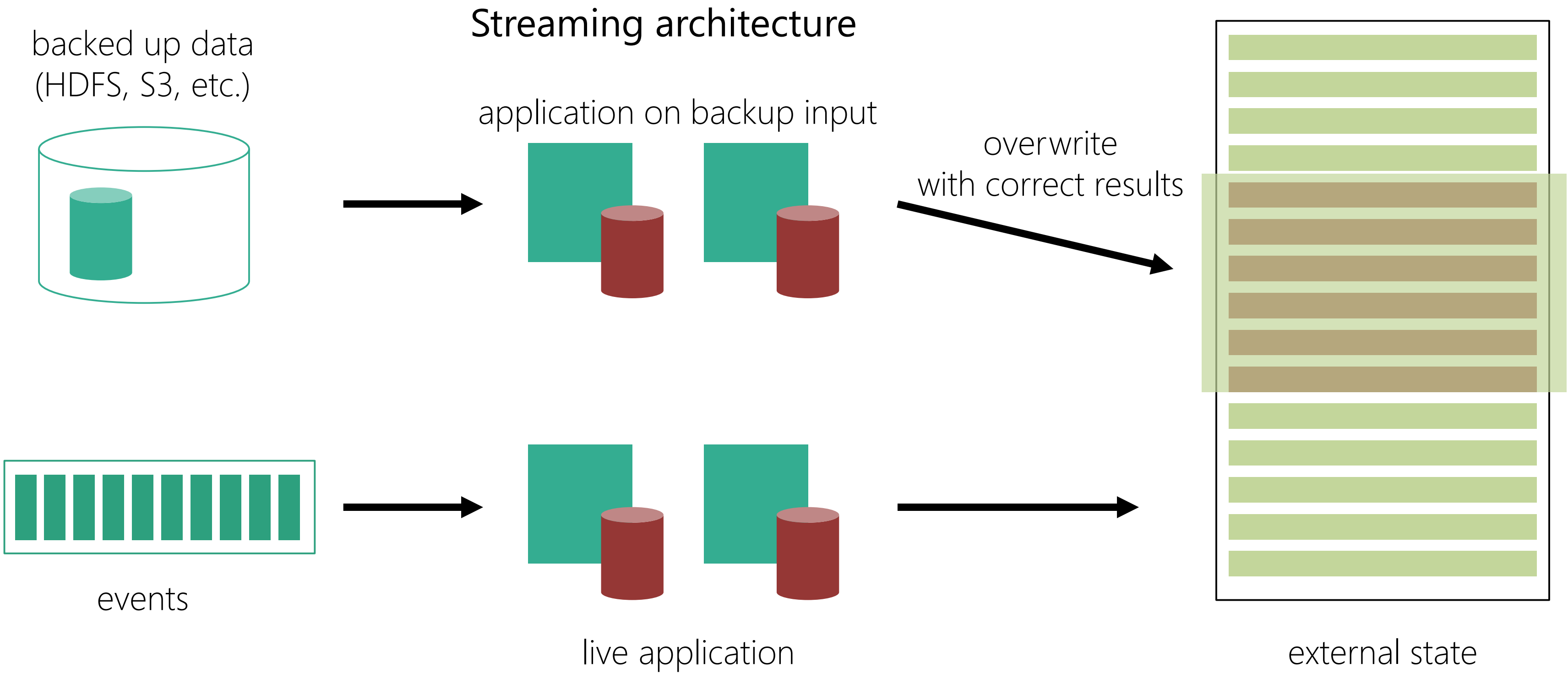


simply occupies some
additional backup space

Repair External State



Repair External State



Repair External State