The Power of Snapshots

Stateful Stream Processing with Apache Flink

Stephan Ewen
dataArtisans

QCon San Francisco, 2017

Original creators of
Apache Flink®

dA Platform 2
Open Source Apache Flink +
dA Application Manager

Stream Processing

What changes faster? Data or Query?

Data changes slowly compared to fast changing queries

ad-hoc queries, data exploration, ML training and (hyper) parameter tuning

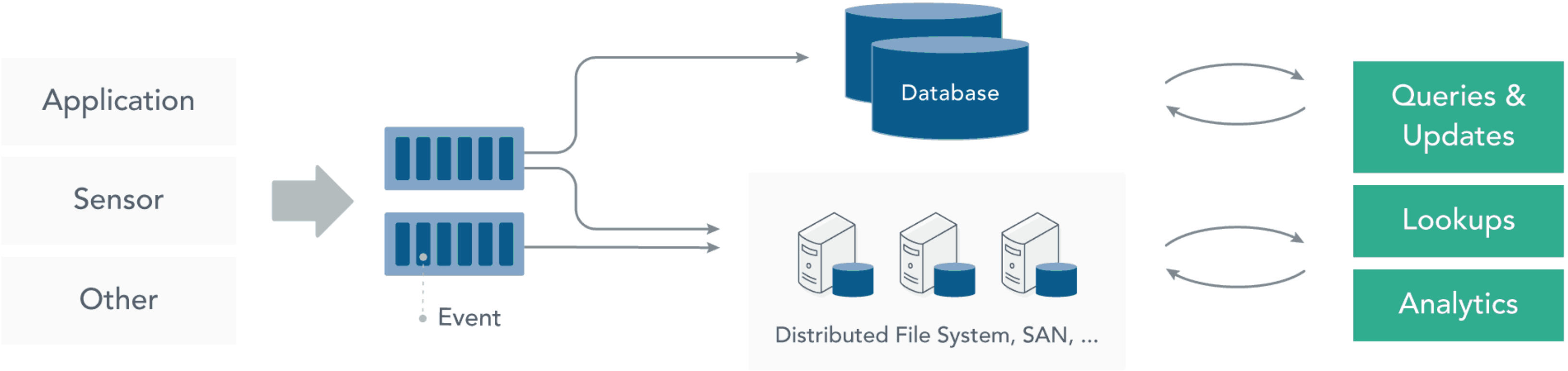
Batch Processing Use Case

Data changes fast application logic is long-lived

continuous applications, data pipelines, standing queries, anomaly detection, ML evaluation, …

Stream Processing Use Case

Batch Processing

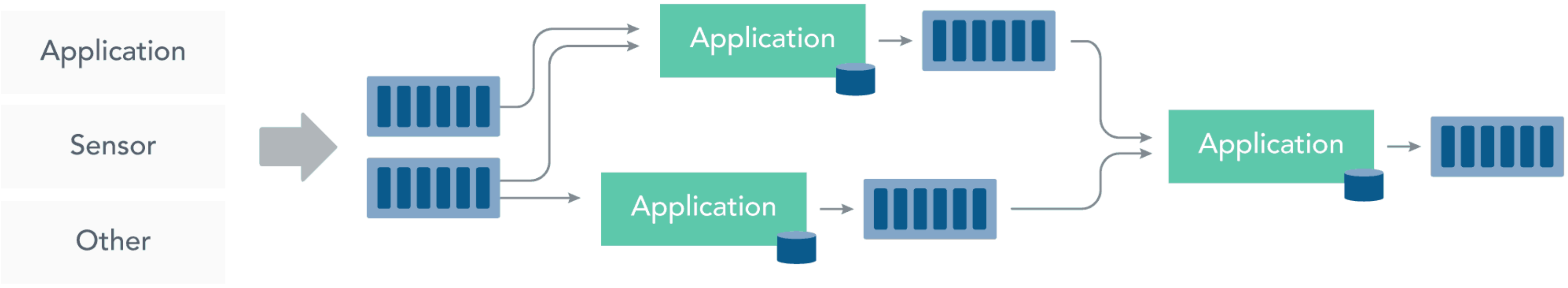


birth of the data
as streams of events

storing data
at rest

applications schedule
computation on the data

Stream Processing

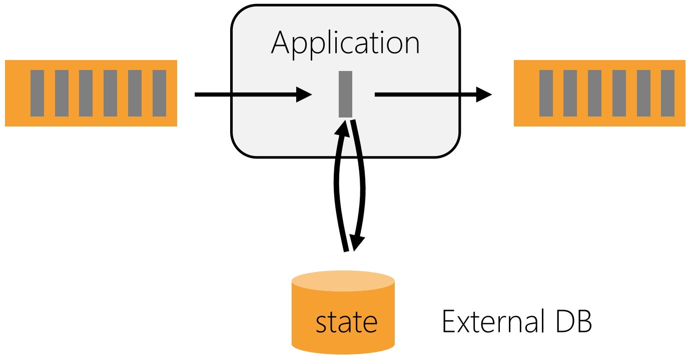


birth of the data
as streams of events

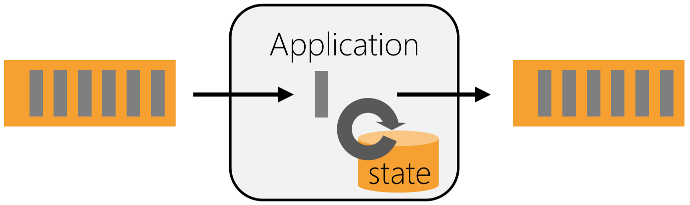
applications computing
over event data streams

Stateful
Stream Processing

Moving State into the Processors



Stateless
Stream Processor



Stateful
Stream Processor

Apache Flink

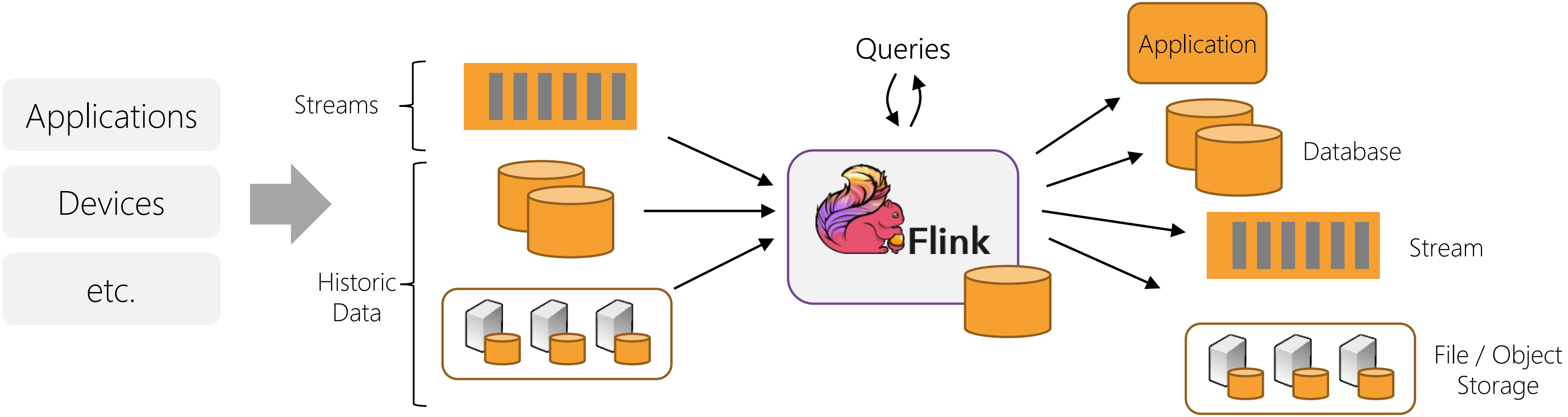
Apache Flink in a Nutshell

Stateful computations over streams

real-time and historic

fast, scalable, fault tolerant, in-memory,

event time, large state, exactly-once



The Core Building Blocks

Event Streams	State	(Event) Time	Snapshots
real-time and hindsight	complex business logic	consistency with out-of-order data and late data	forking / versioning / time-travel

Stateful Event & Stream Processing

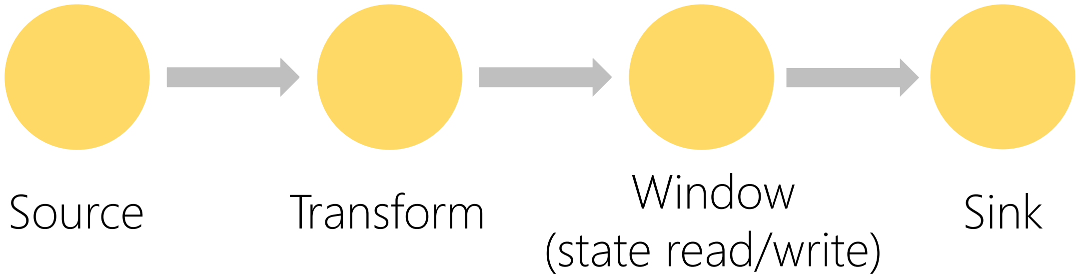
```
val lines: DataStream[String] = env.addSource(new FlinkKafkaConsumer09(…))

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

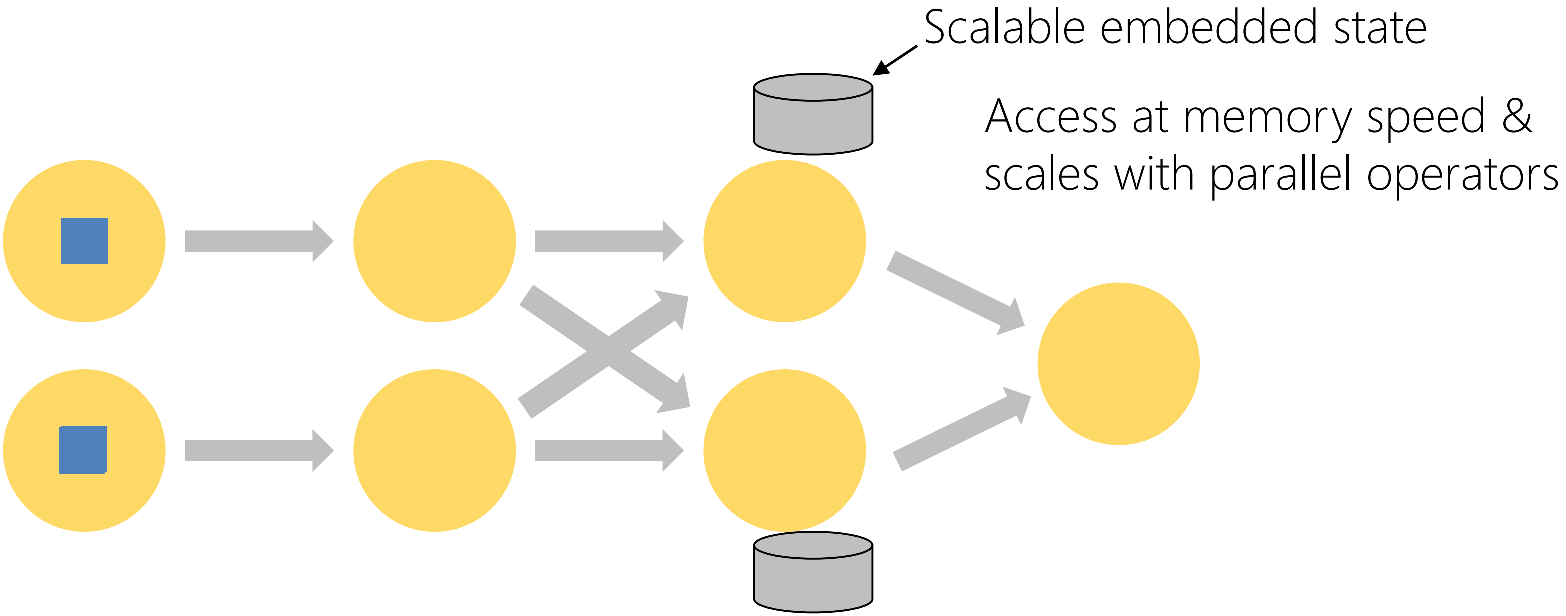
Transformation

Sink

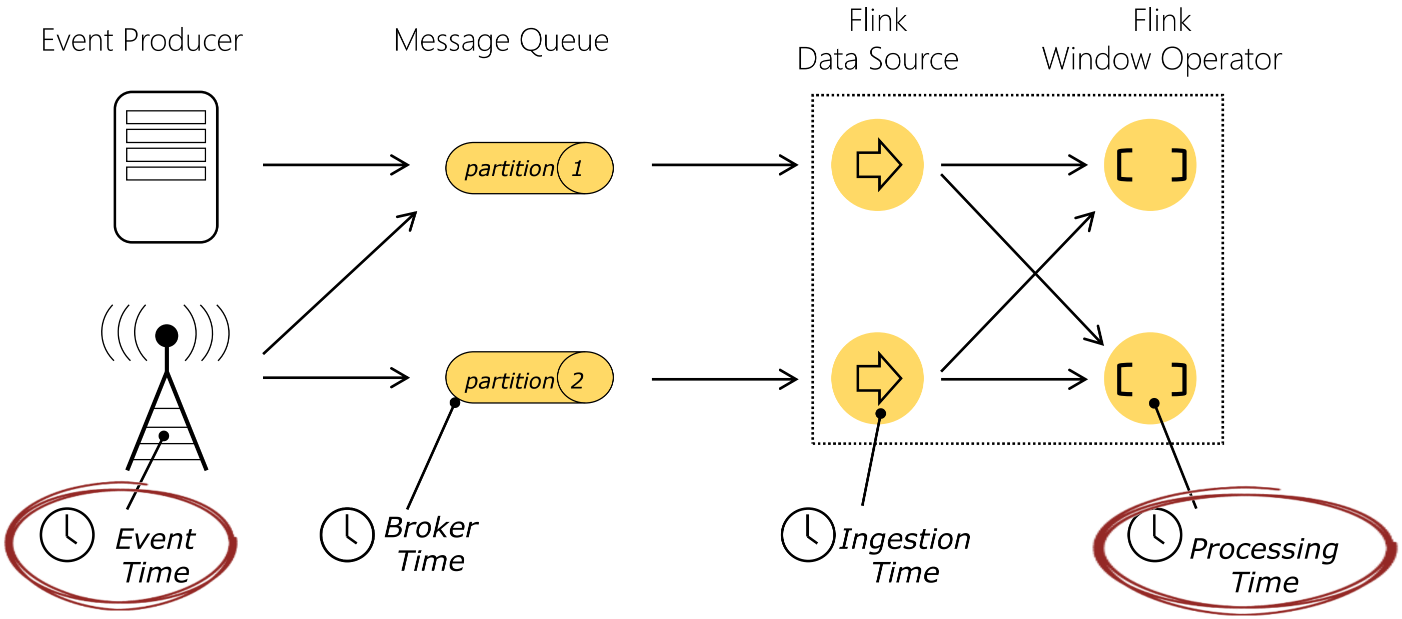


Streaming Dataflow

Stateful Event & Stream Processing



Event time and Processing Time



Event time, Watermarks, as in the Dataflow model

Powerful Abstractions

Layered abstractions to
navigate simple to complex use cases

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

High-level Analytics API — Stream SQL / Tables *(dynamic tables)*

Stream- & Batch Data Processing — DataStream API *(streams, windows)*

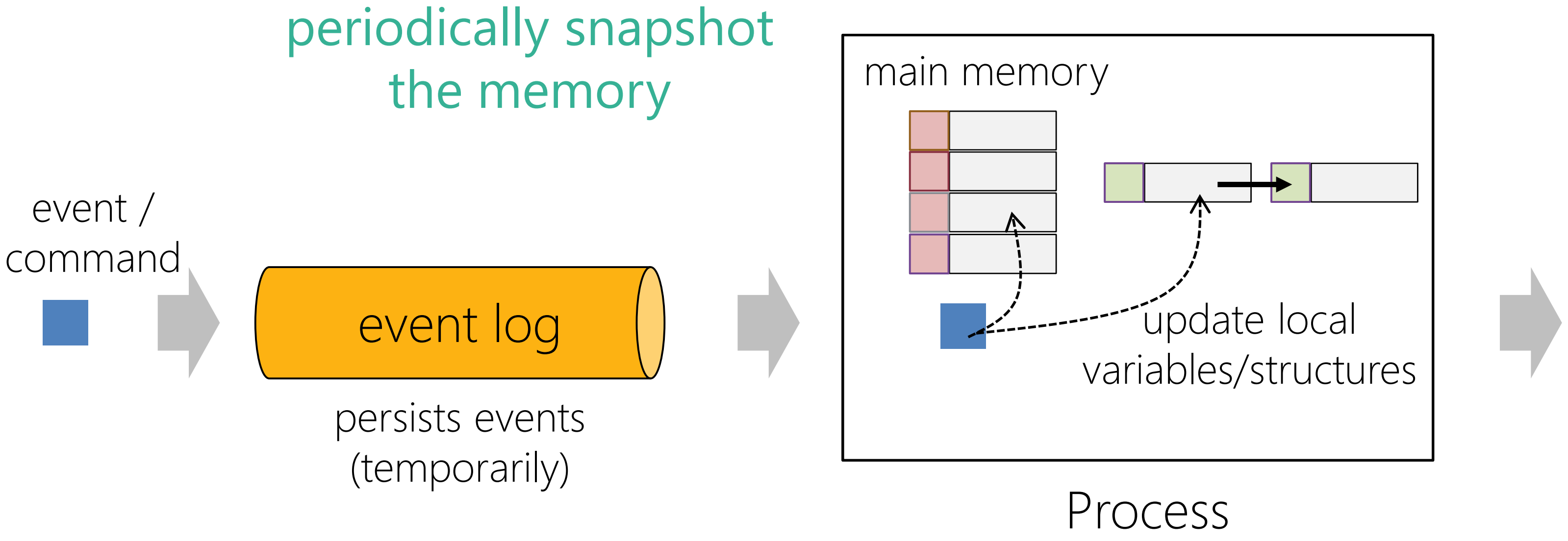
```
val stats = stream
   .keyBy("sensor")
   .timeWindow(Time.seconds(5))
   .sum((a, b) -> a.add(b))
```

Stateful Event-Driven Applications — Process Function *(events, state, time)*

```
def processElement(event: MyEvent, ctx: Context, out: Collector[Result]) = {
  // 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)
}
```

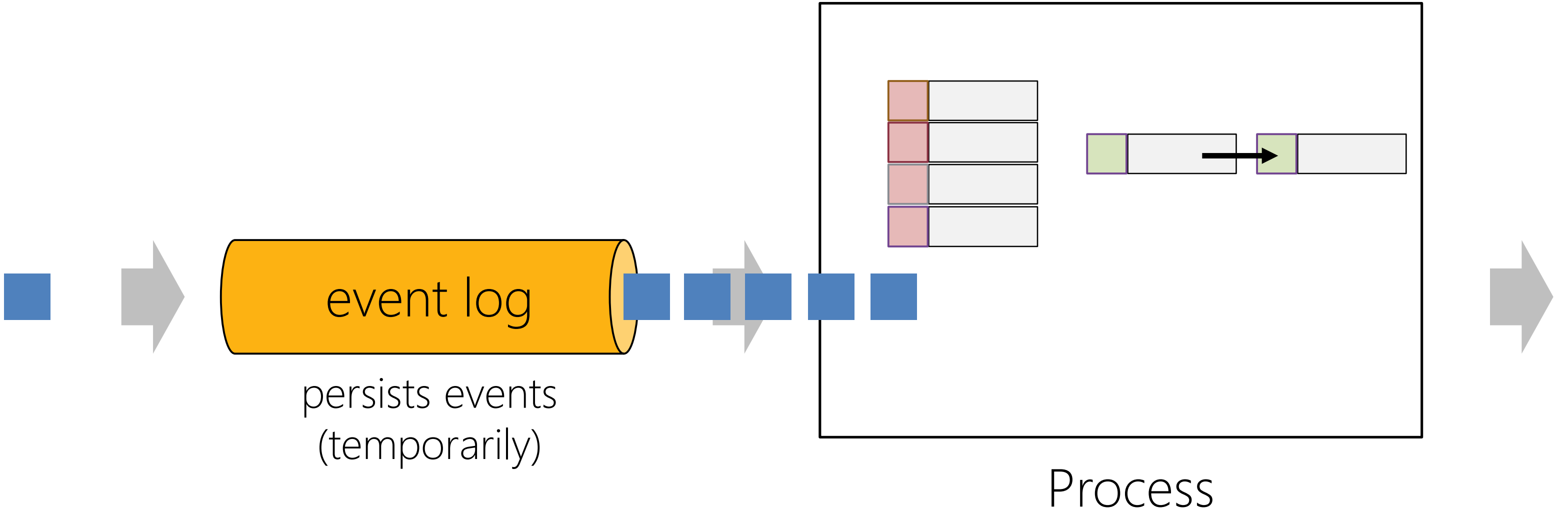
Distributed Snapshots

Event Sourcing + Memory Image

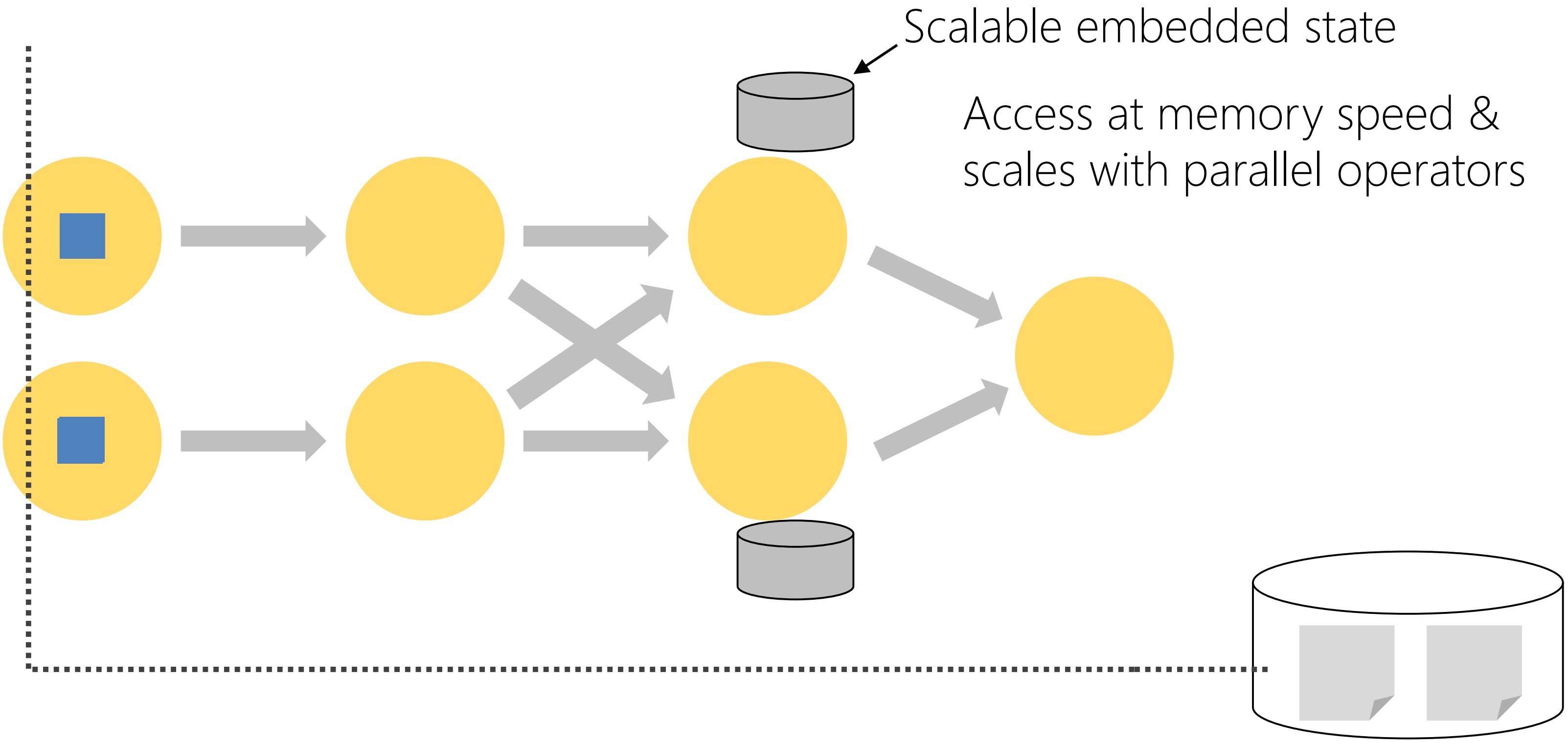


Event Sourcing + Memory Image

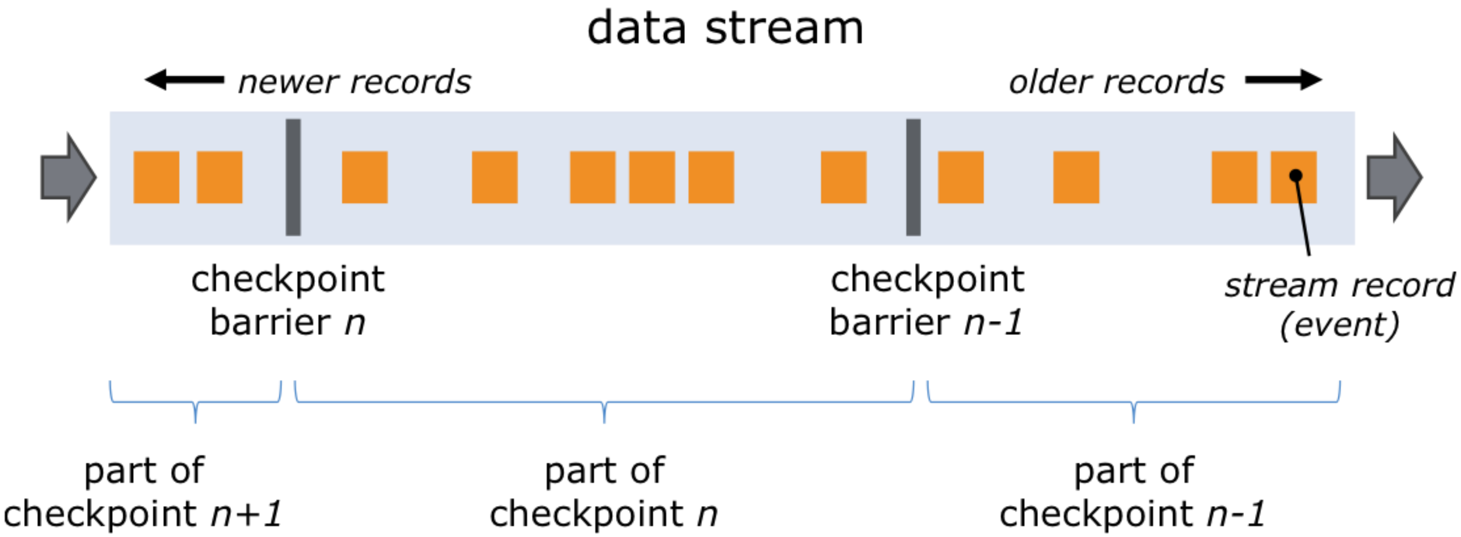
Recovery: Restore snapshot and replay events since snapshot



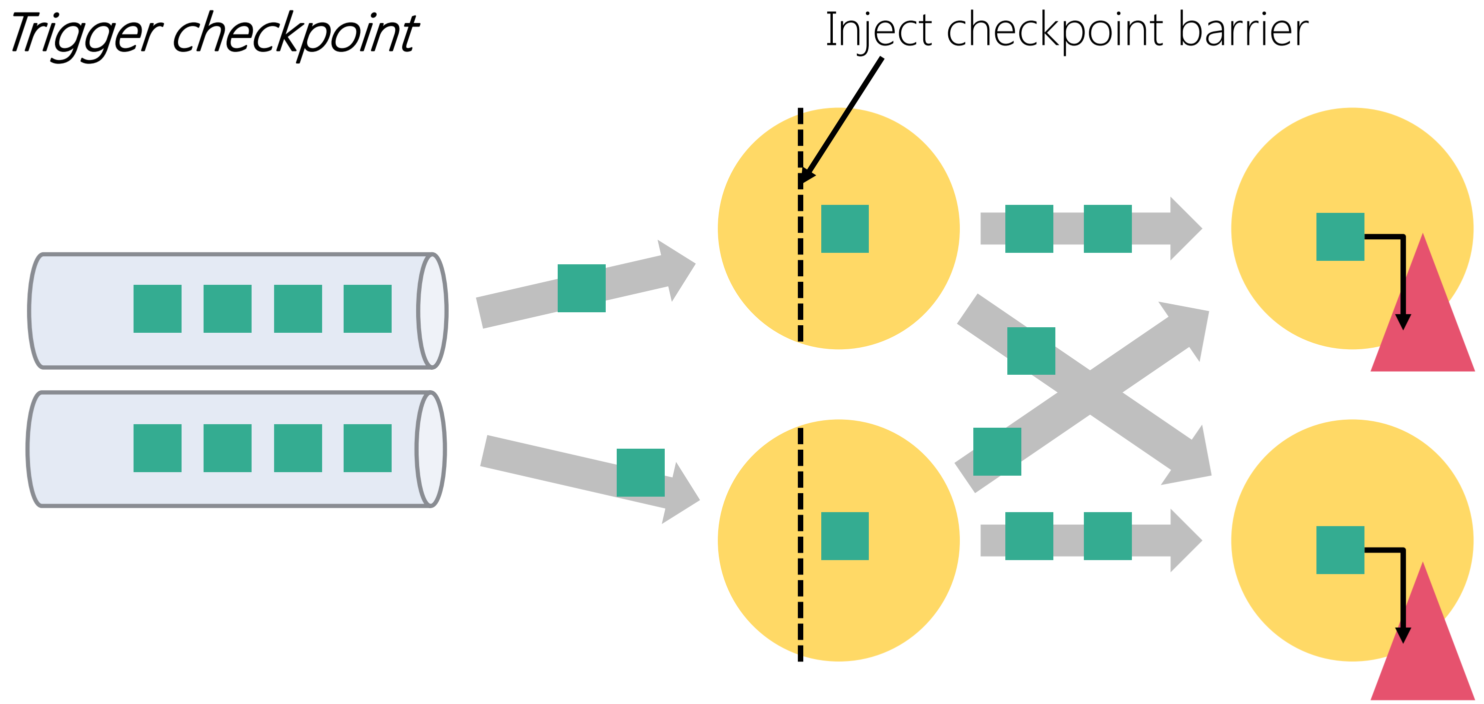
Consistent Distributed Snapshots



Checkpoint Barriers

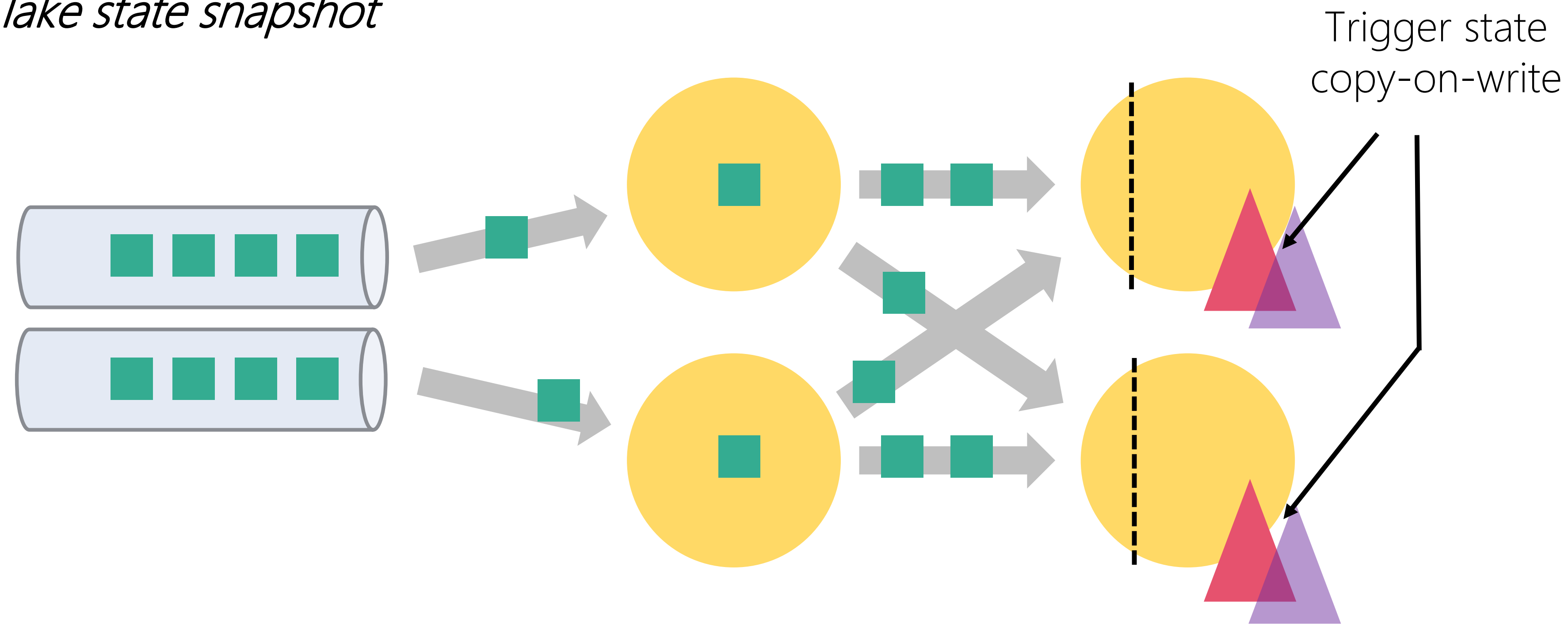


Consistent Distributed Snapshots



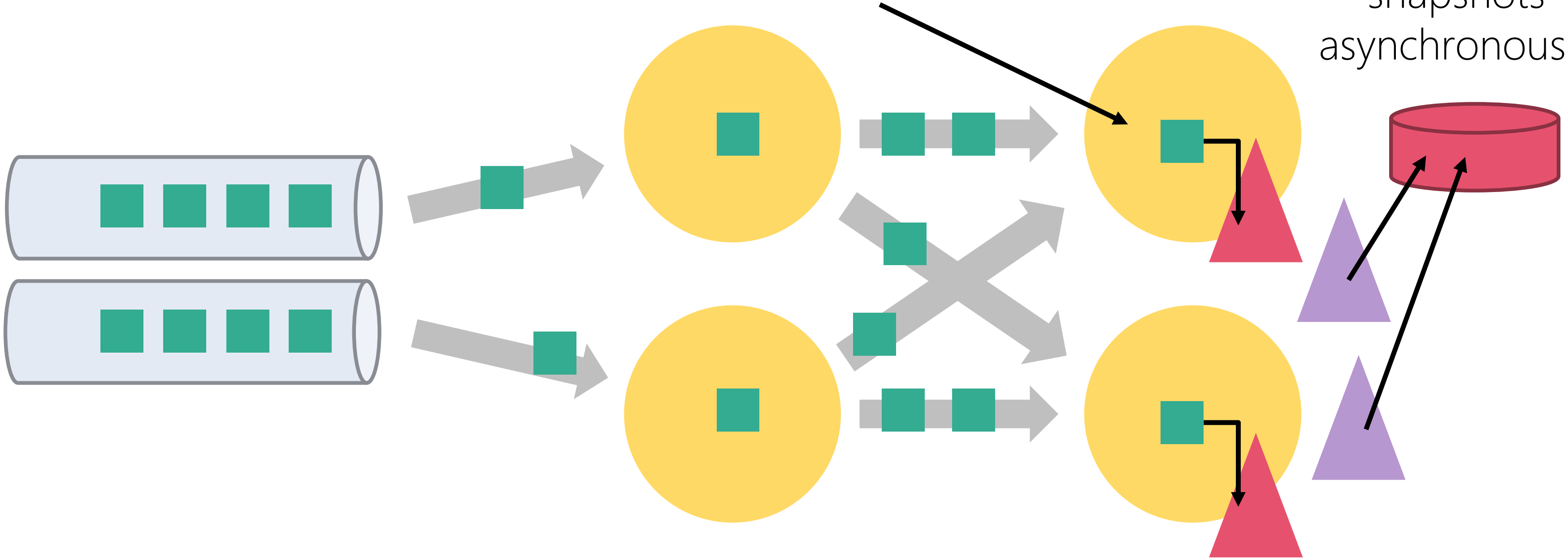
Consistent Distributed Snapshots

Take state snapshot



Consistent Distributed Snapshots

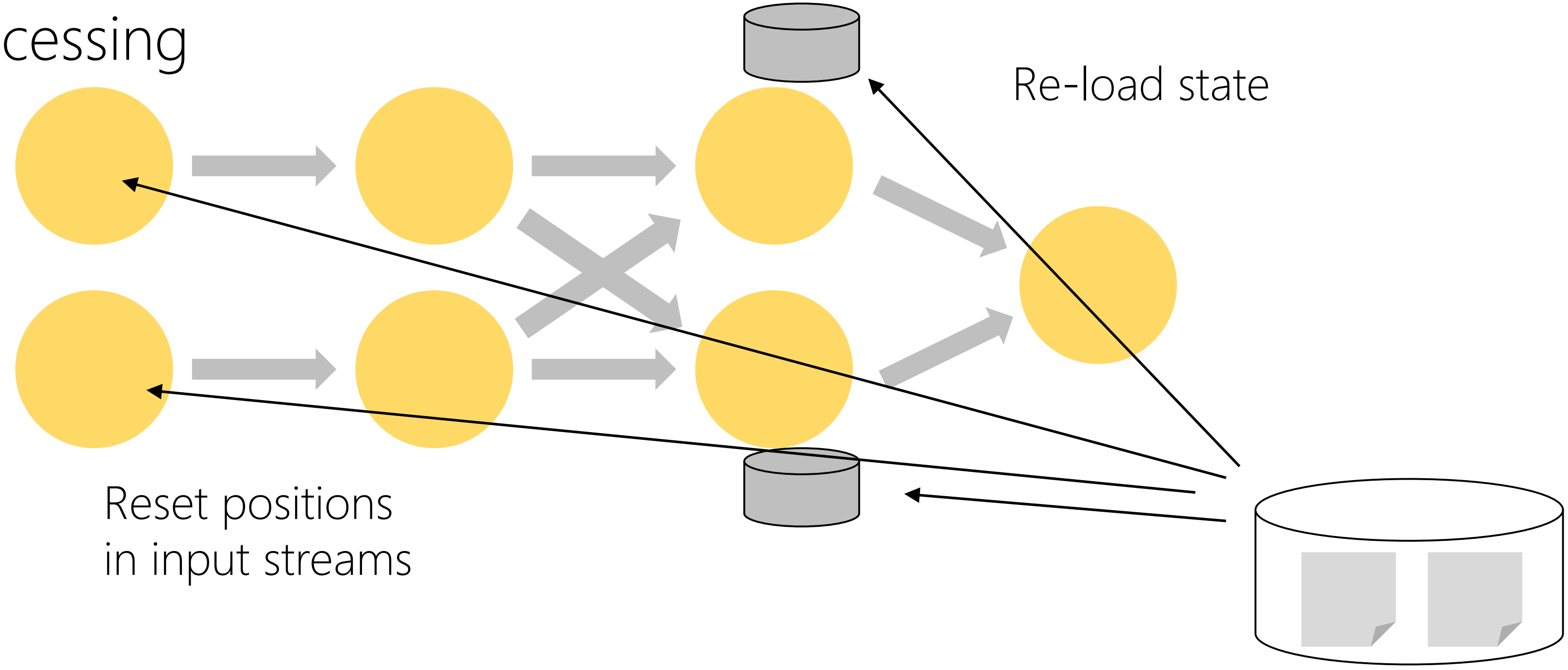
Persist state snapshots



Consistent Distributed Snapshots

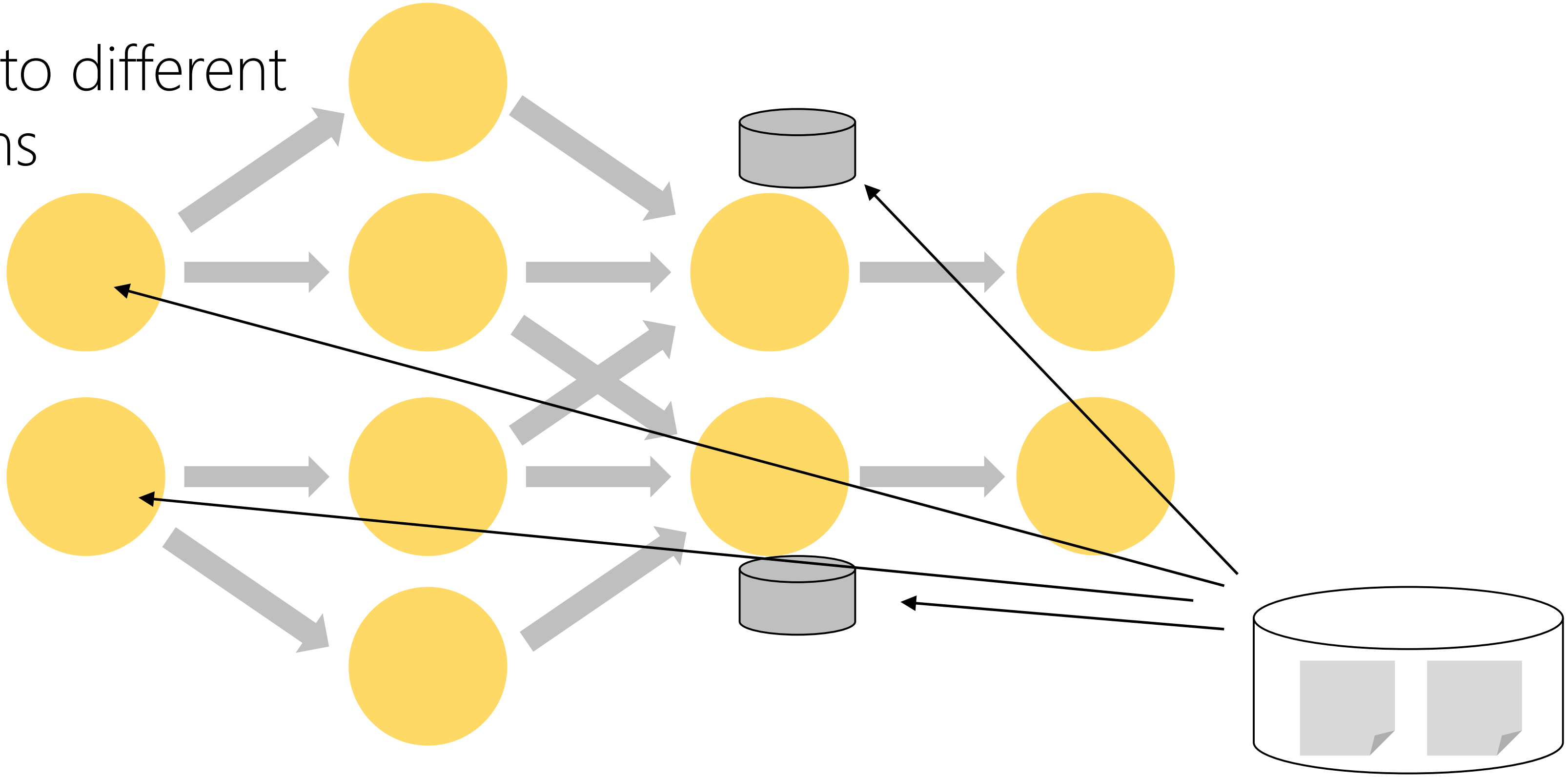
Rolling back computation

Re-processing



Consistent Distributed Snapshots

Restore to different programs



Checkpoints and Savepoints in Apache Flink

Speed or Operability?

What to optimize for?

Fast snapshots	Flexible Operations on Snapshots
Checkpoint	Savepoint

Savepoints: Opt. for Operability

- **Self contained:** No references to other checkpoints
- **Canonical format:** Switch between state structures
- **Efficiently re-scalable:** Indexed by key group

- Future: More self-describing serialization format for to archiving / versioning (like Avro, Thrift, etc.)

Checkpoints: Opt. for Efficiency

- **Often incremental:**
  - Snapshot only diff from last snapshot
  - Reference older snapshots, compaction over time
- **Format specific to state backend:**
  - No extra copied or re-encoding
  - Not possible to switch to another state backend between checkpoints
- **Compact serialization:** Optimized for speed/space, not long term archival and evolution
- **Key goups not indexed:** Re-distribution may be more expensive

What else are snapshots / checkpoints good for?

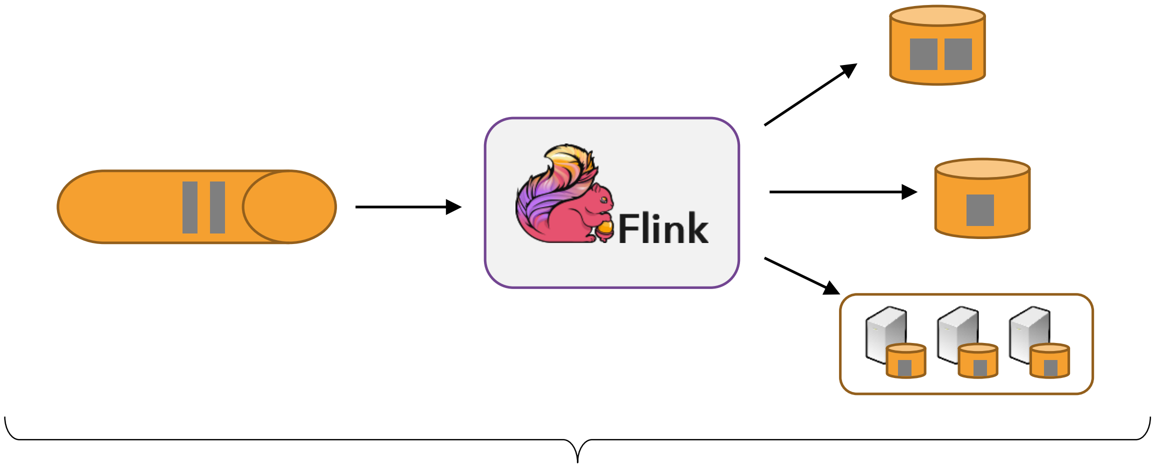
What users built on checkpoints

- Upgrades and Rollbacks
- Cross Datacenter Failover
- State Archiving
- State Bootstrapping
- Application Migration
- Spot Instance Region Arbitrage
- A/B testing
- …

Distributed Snapshots and side effects

Transaction coordination for side fx

Snapshots may include side effects



One snapshot can transactionally move
data between different systems

Transaction coordination for side fx

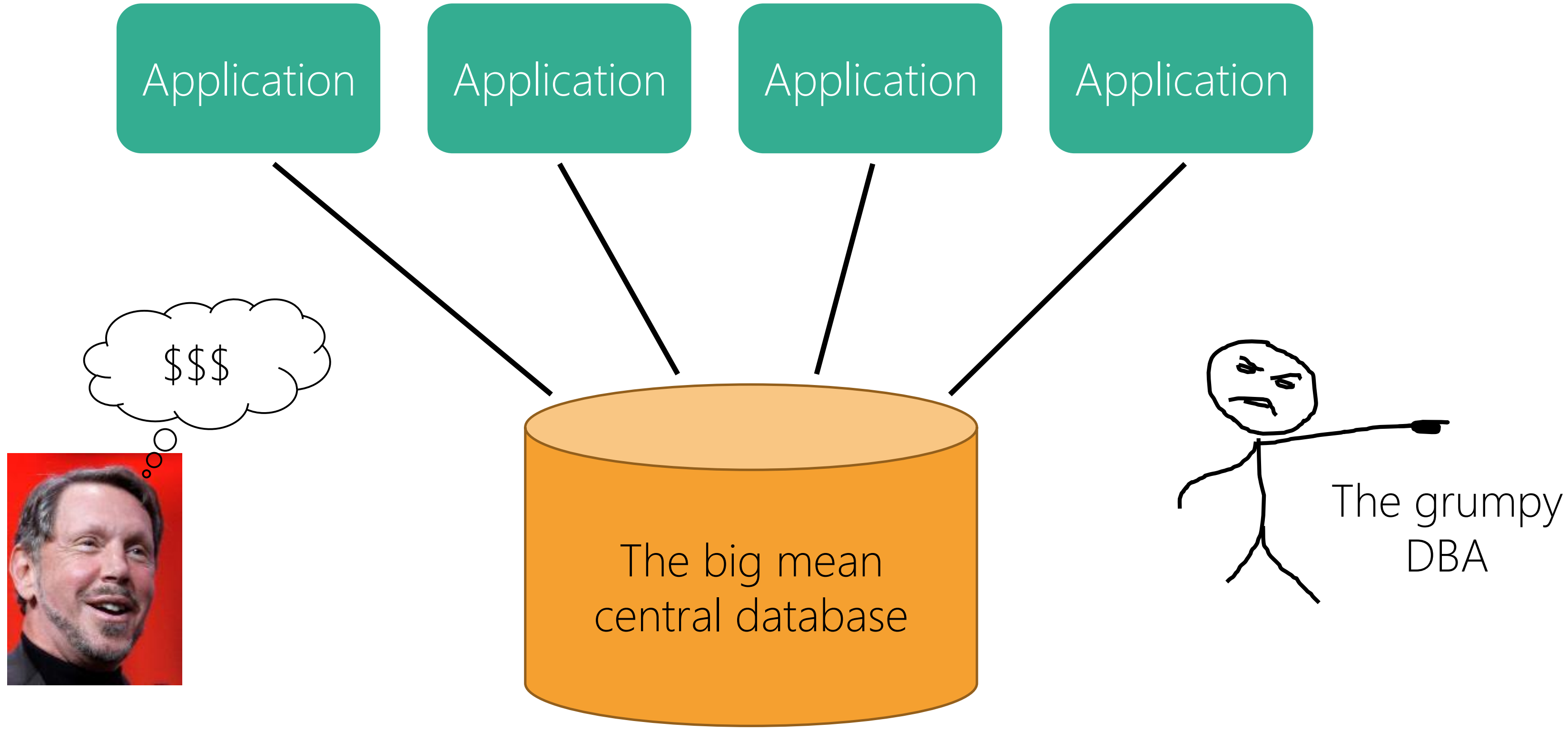
- Similar to a distributed 2-phase commit
- Coordinated by asynchronous checkpoints, no voting delays

- Basic algorithm:
  - Between checkpoints: Produce into transaction or Write Ahead Log
  - On operator snapshot: Flush local transaction *(vote-to-commit)*
  - On checkpoint complete: Commit transactions *(commit)*
  - On recovery: check and commit any pending transactions

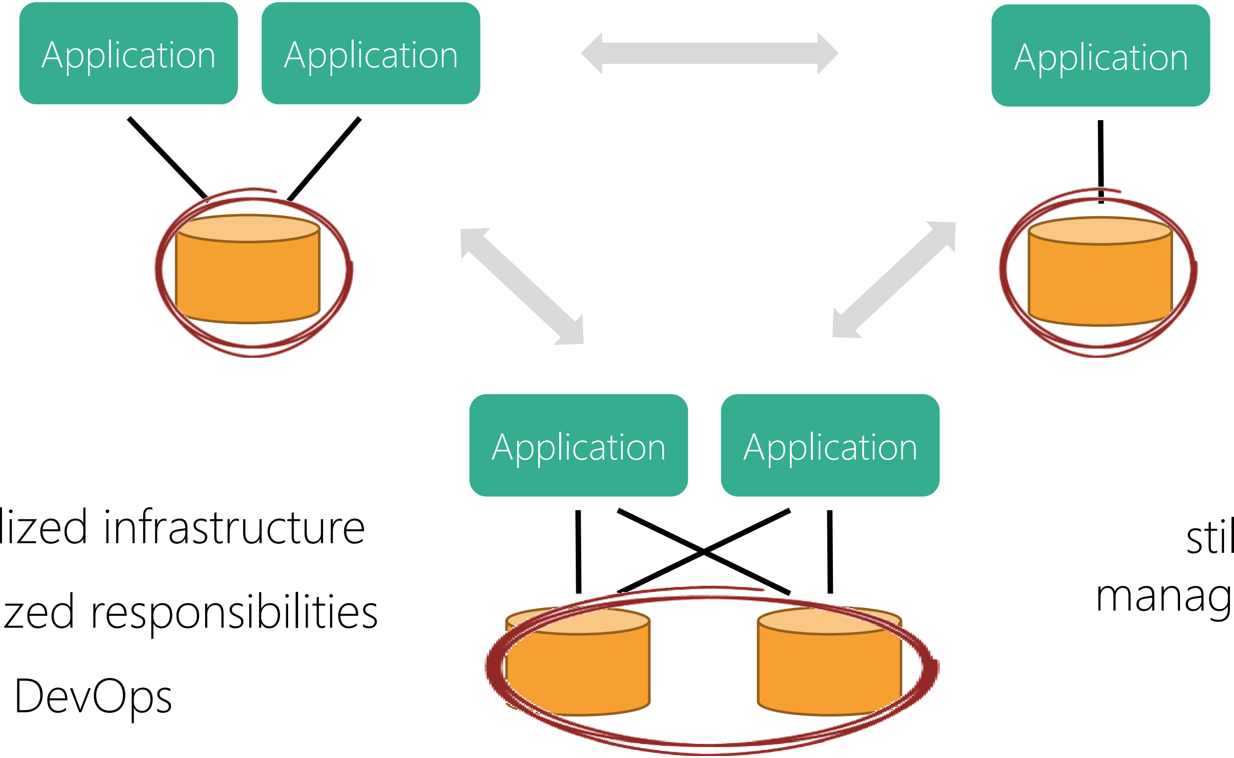
Distributed Snapshots and Application Architectures

(A Philosophical Monologue)

Good old centralized architecture



Stateful Stream Proc. & Applications



decentralized infrastructure

decentralized responsibilities

DevOps

still involves
managing databases

Stateless Application Containers

State management
is nasty, let's pretend we don't
have to do it

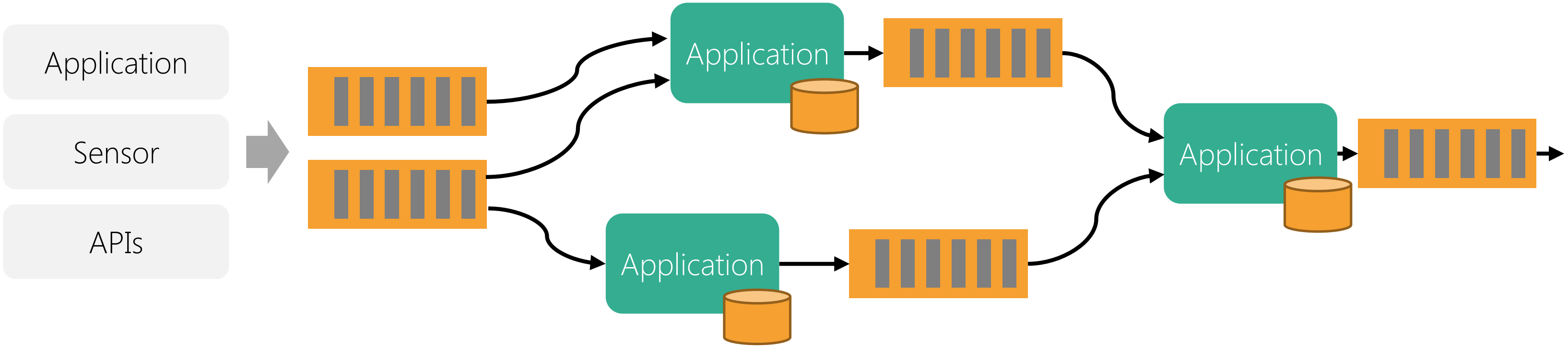
Stateless Application Containers

Broccoli ~~(state management)~~ is nasty, let's pretend we don't have to eat ~~do~~ it

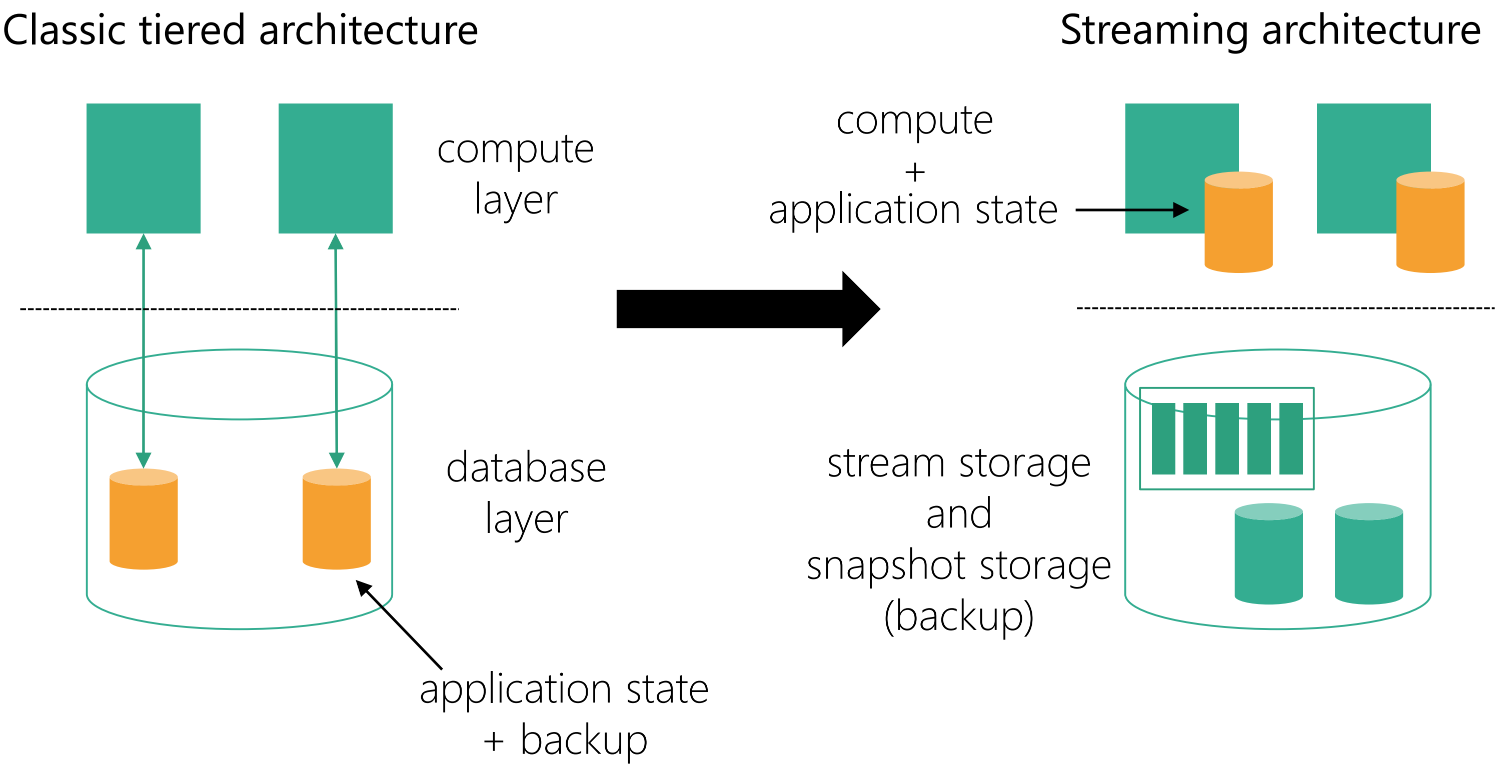
Kudos to Kiki Carter for the Broccoli Metaphor

Stateful Stream Proc. to the rescue

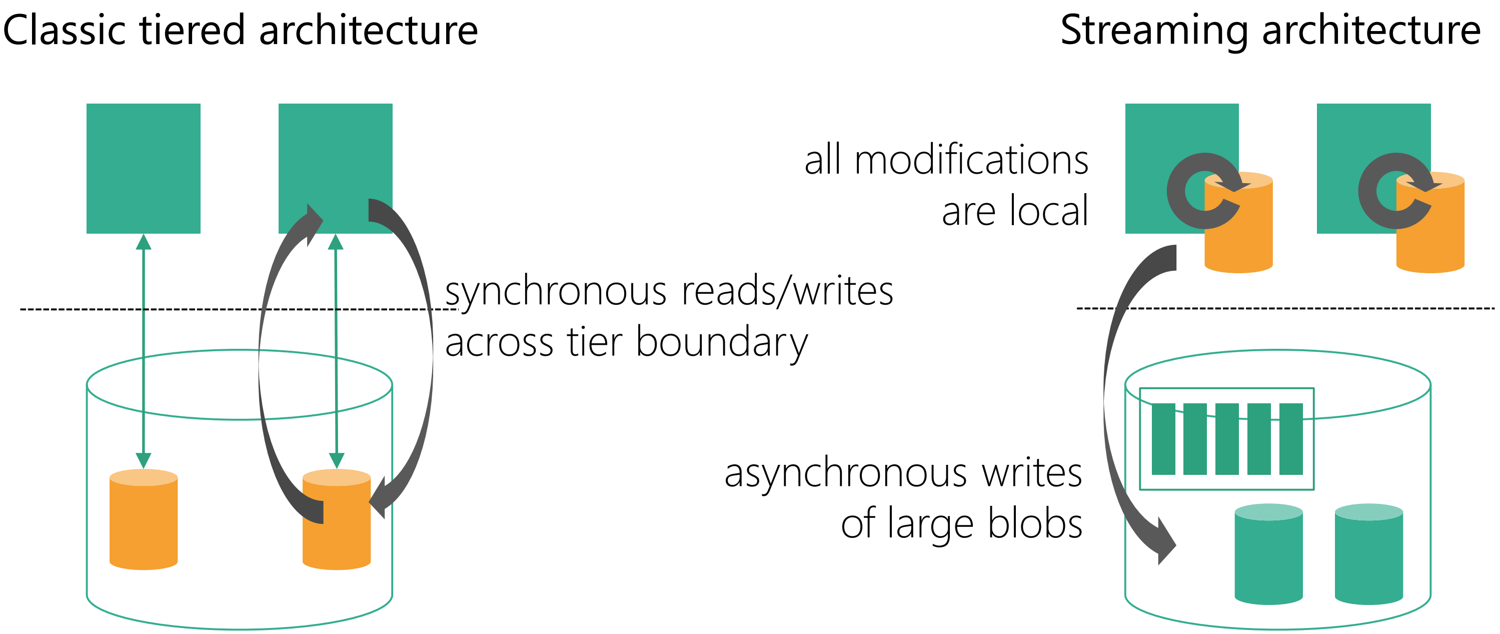
very simple: state is just part of the application



Compute, State, and Storage

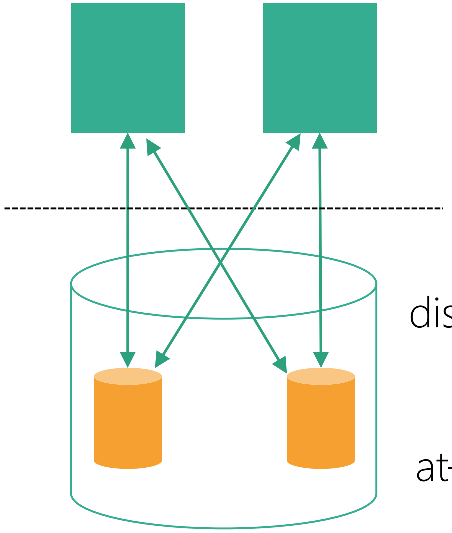


Performance



Consistency

Classic tiered architecture

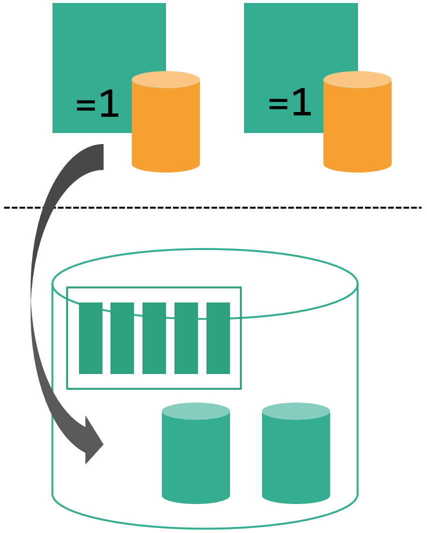


distributed transactions

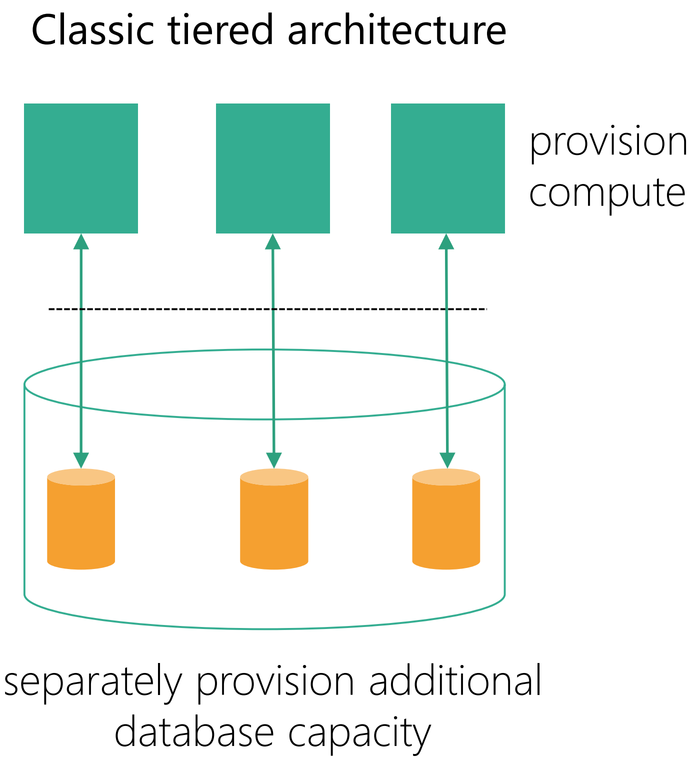
at scale typically
at-most / at-least once

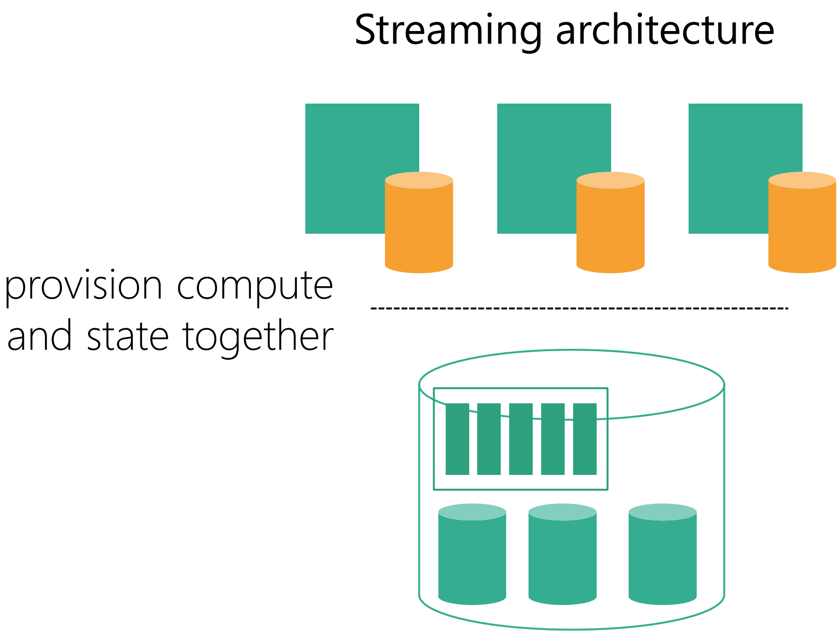
Streaming architecture

exactly once
per state



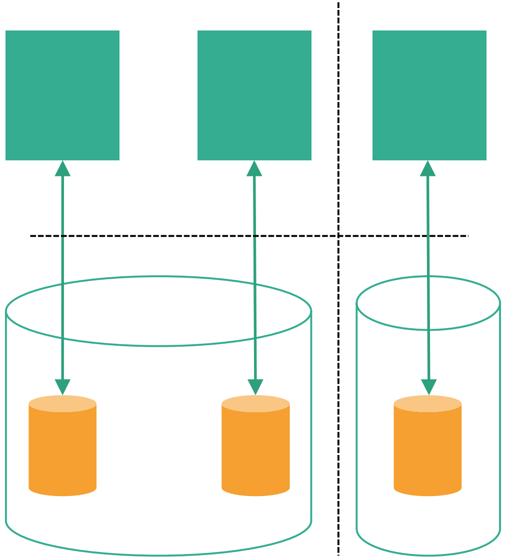
Scaling a Service





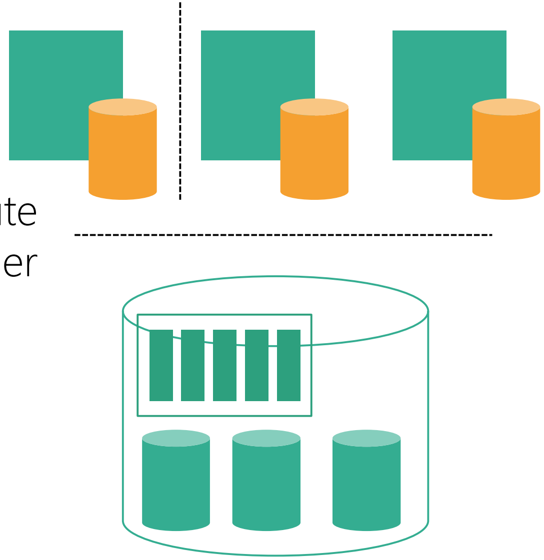
Rolling out a new Service

Classic tiered architecture



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

Streaming architecture

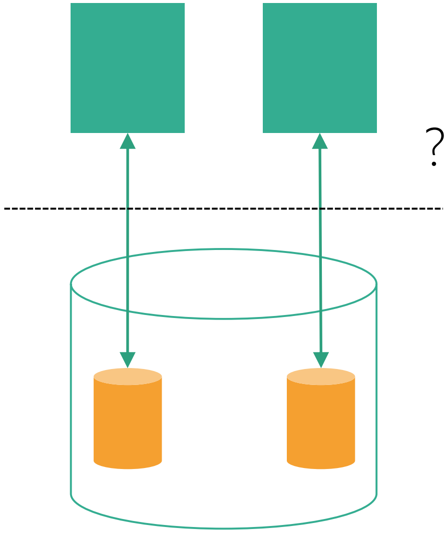


provision compute
and state together

simply occupies some
additional backup space

Time, Completeness, Out-of-order

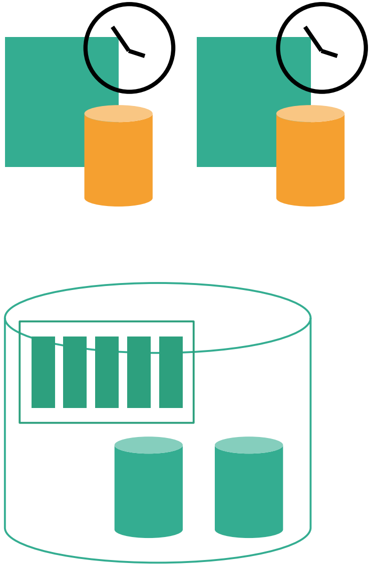
Classic tiered architecture



Streaming architecture

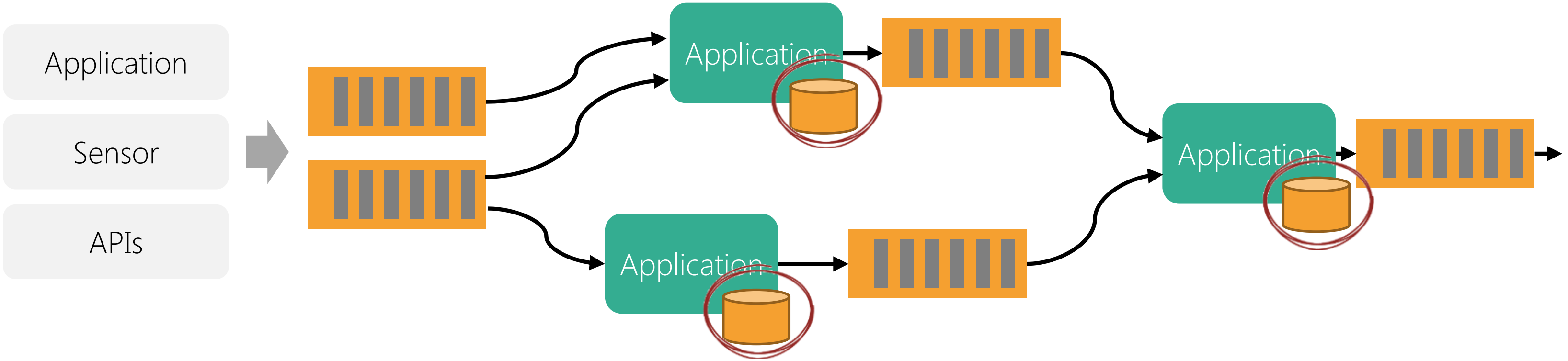
event time clocks
define data completeness

event time timers
handle actions for
out-of-order data



Stateful Stream Processing

very simple: state is just part
of the application



The Challenges with that:

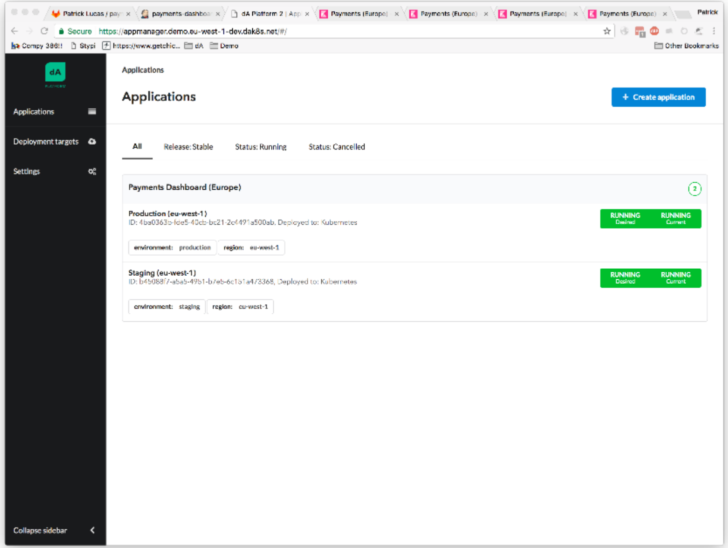
- Upgrades are stateful, need consistency
  - application evolution and bug fixes
- Migration of application state
  - cluster migration, A/B testing
- Re-processing and reinstatement
  - fix corrupt results, bootstrap new applications
- State evolution (schema evolution)

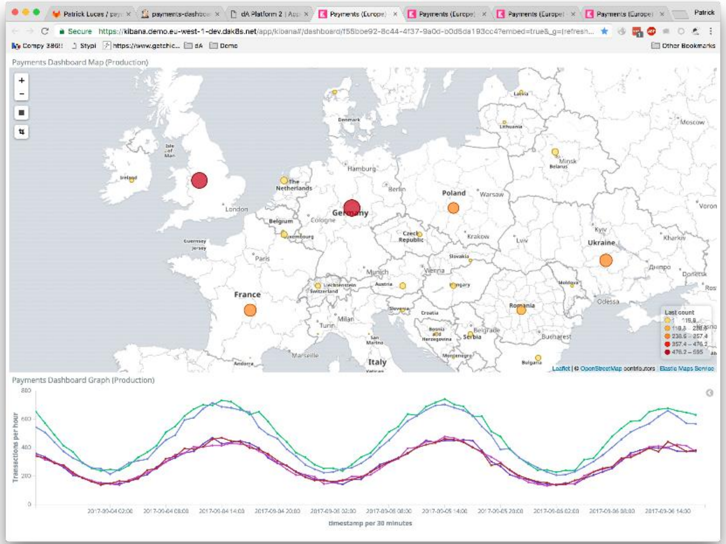
The answer

(my personal and obviously biased take)

Consistent Distributed Snapshots

Demo Time!





Payments Dashboard

Thank you very much ☺

(shameless plug)

dataArtisans

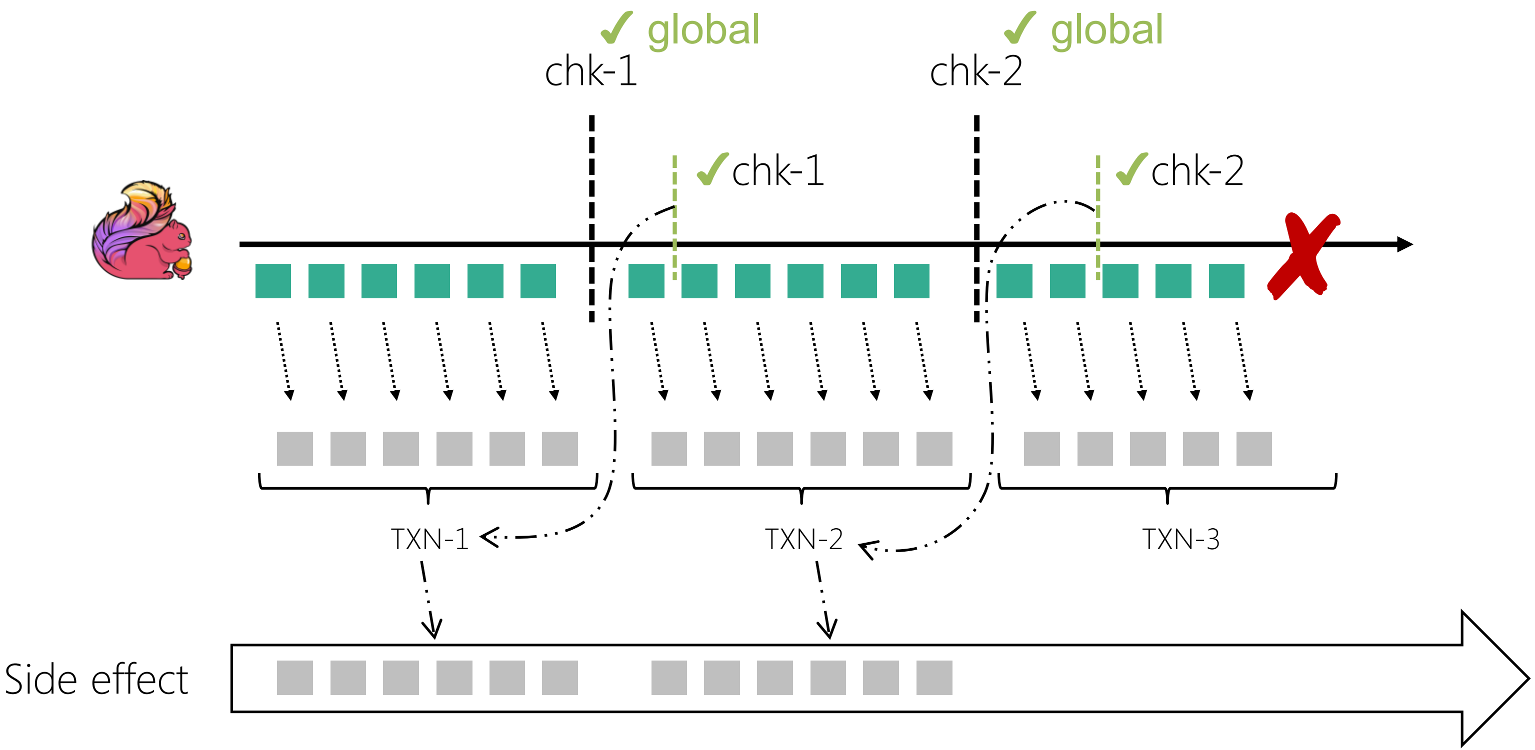
We are hiring!

data-artisans.com/careers

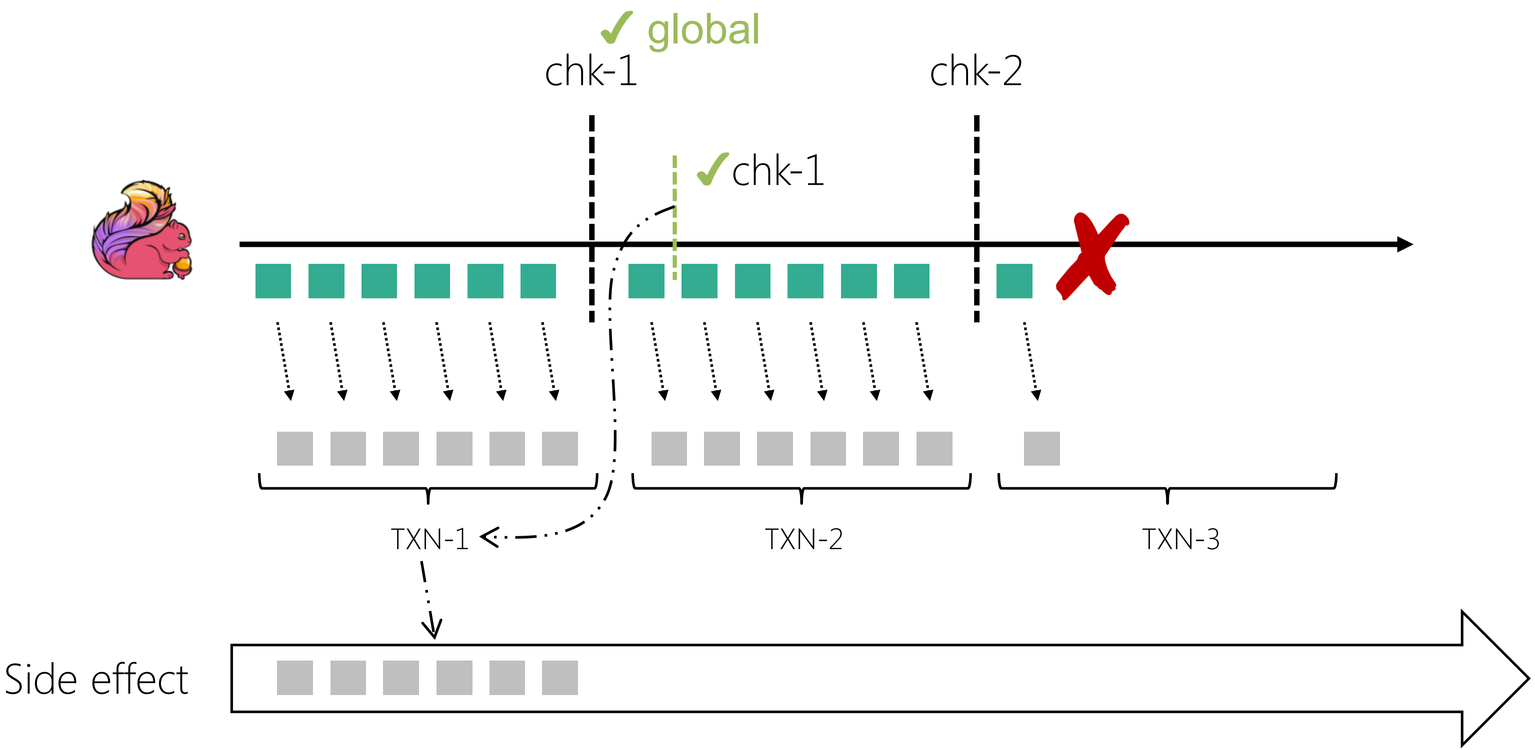
Appendix

Details about Snapshots and Transactional Side Effects

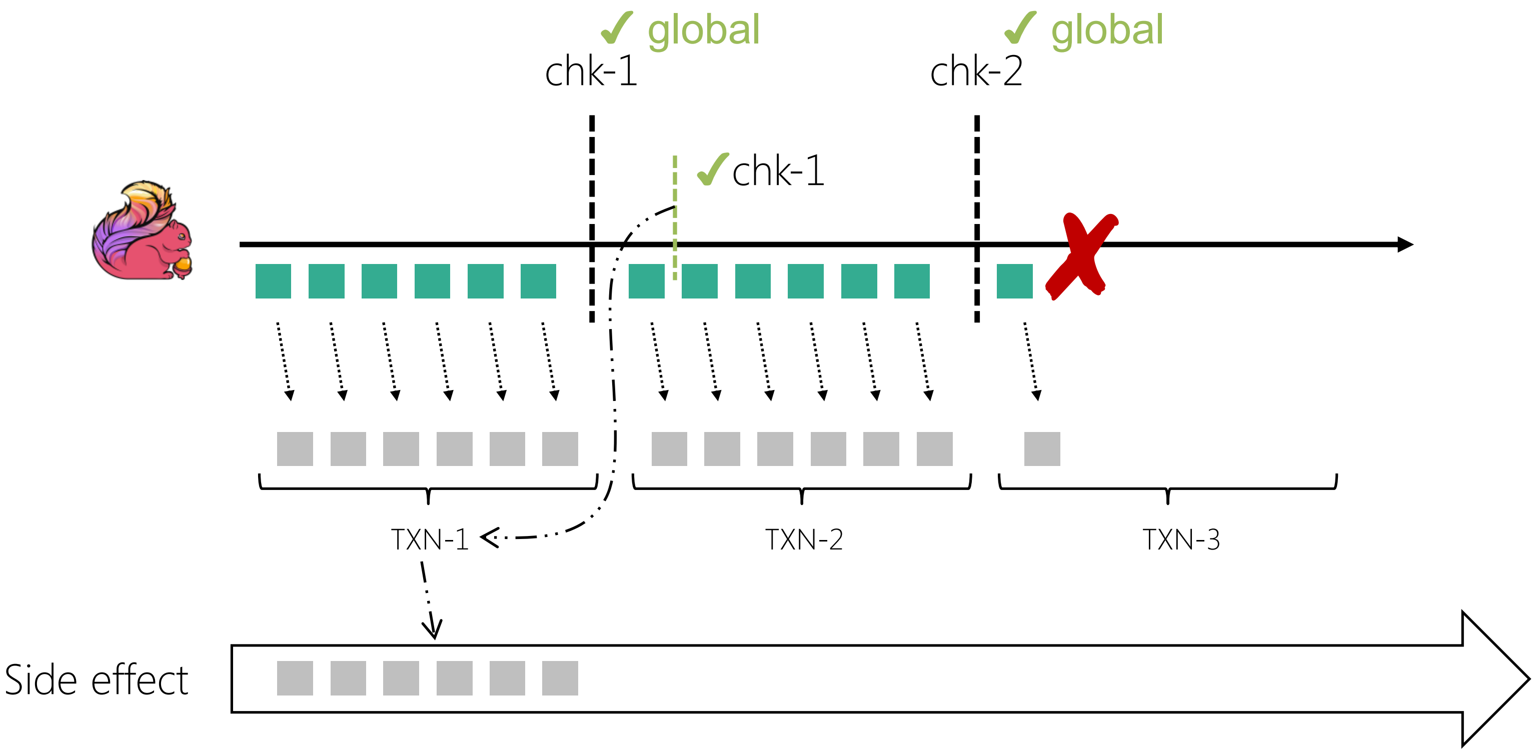
Exactly-once via Transactions



Transaction fails after local snapshot



Transaction fails before commit…



… commit on recovery