Experiences in running Apache Flink® at large scale

Stephan Ewen (@StephanEwen)

dataArtisans

Lessons learned from running Flink at large scale

Including various things we never expected to become a problem and evidently still did…

Also a preview to various fixes coming in Flink…

What is large scale?

Large Data Volume
(events / sec)

Large Application State
(GBs / TBs)

Complex Dataflow
Graphs
(many operators)

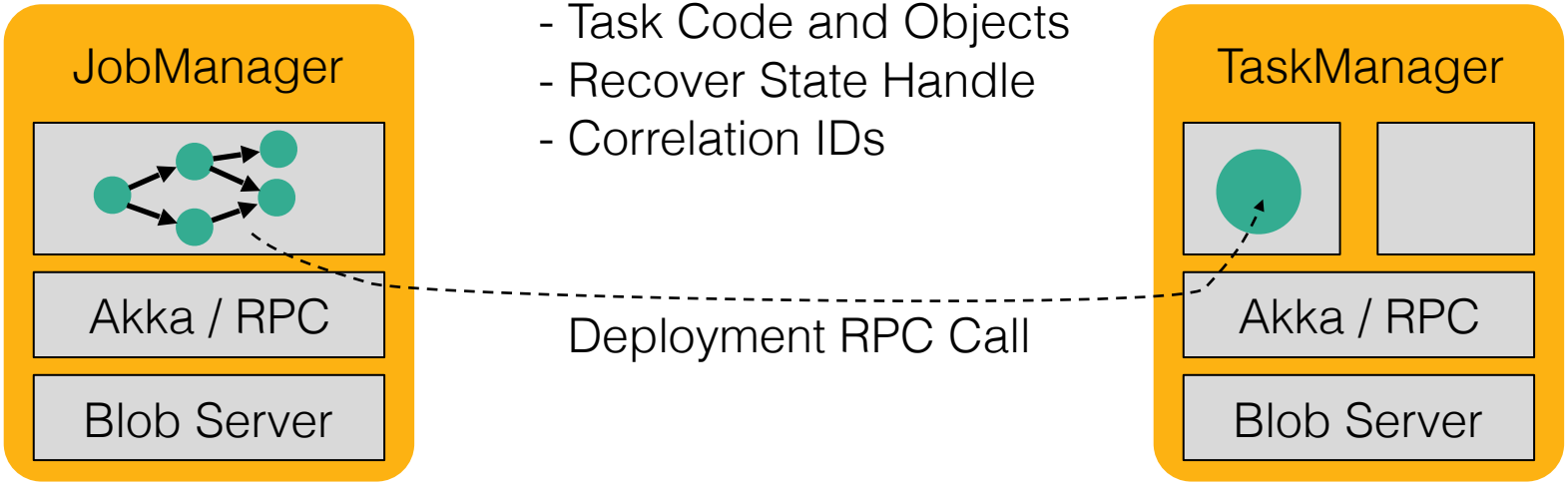
High Parallelism
(1000s of subtasks)

Distributed Coordination

Deploying Tasks

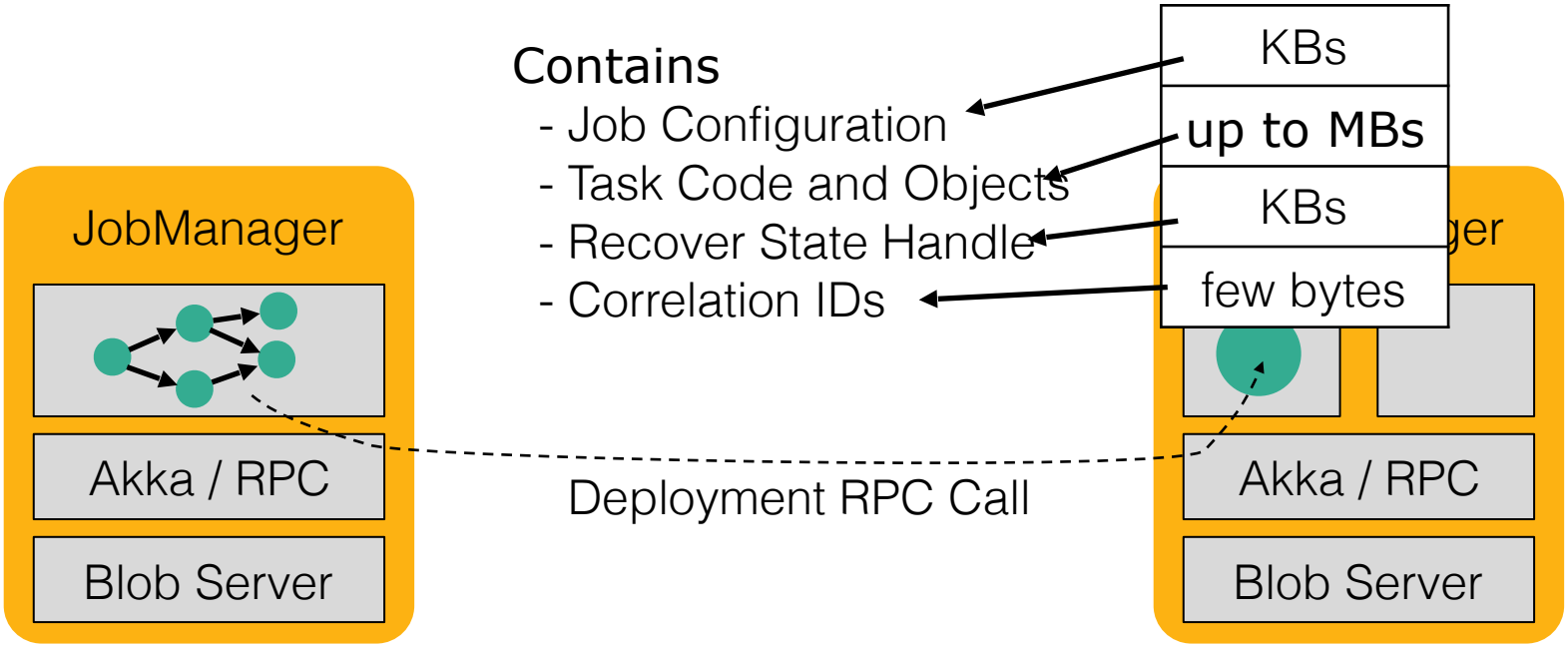
Happens during initial deployment and recovery

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

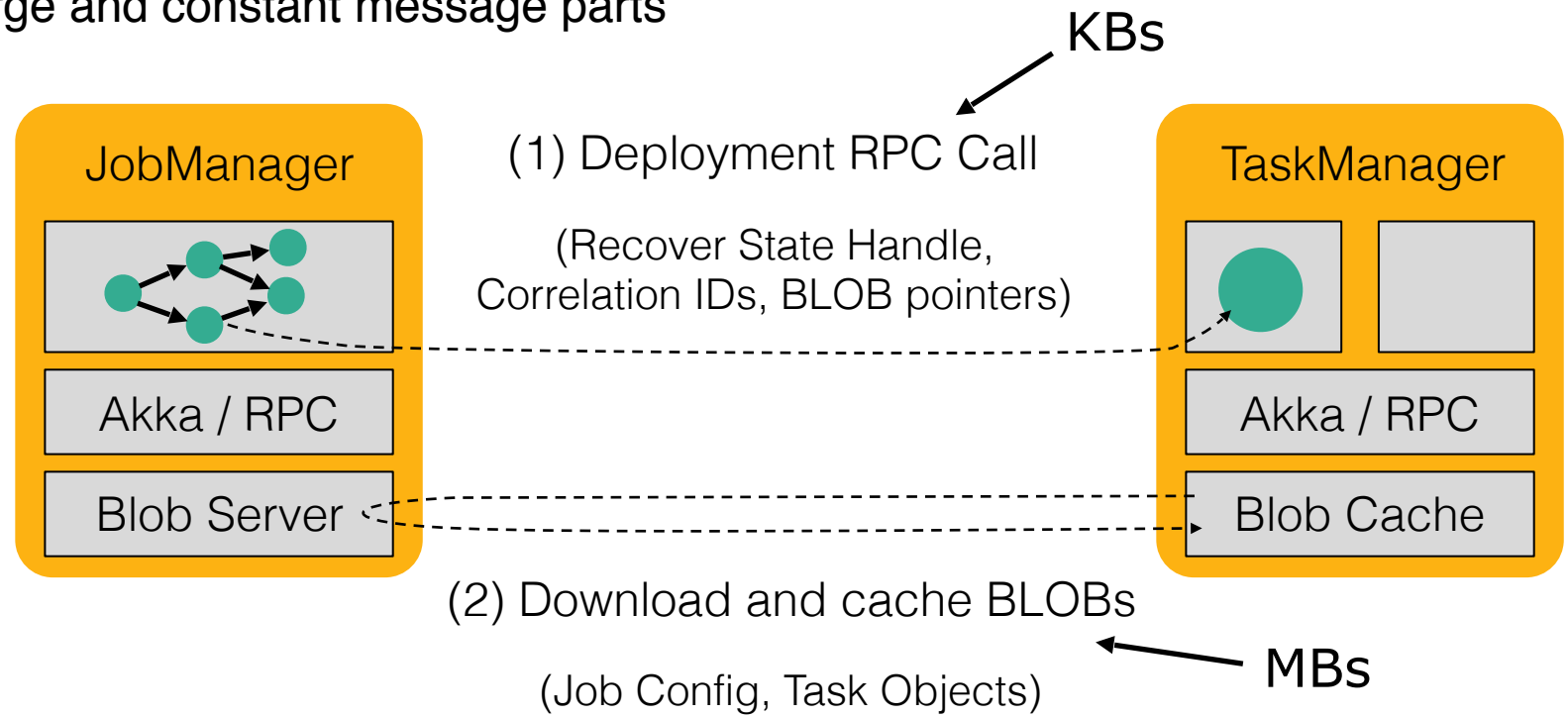
Future: Reduce RPC load (next slides)

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

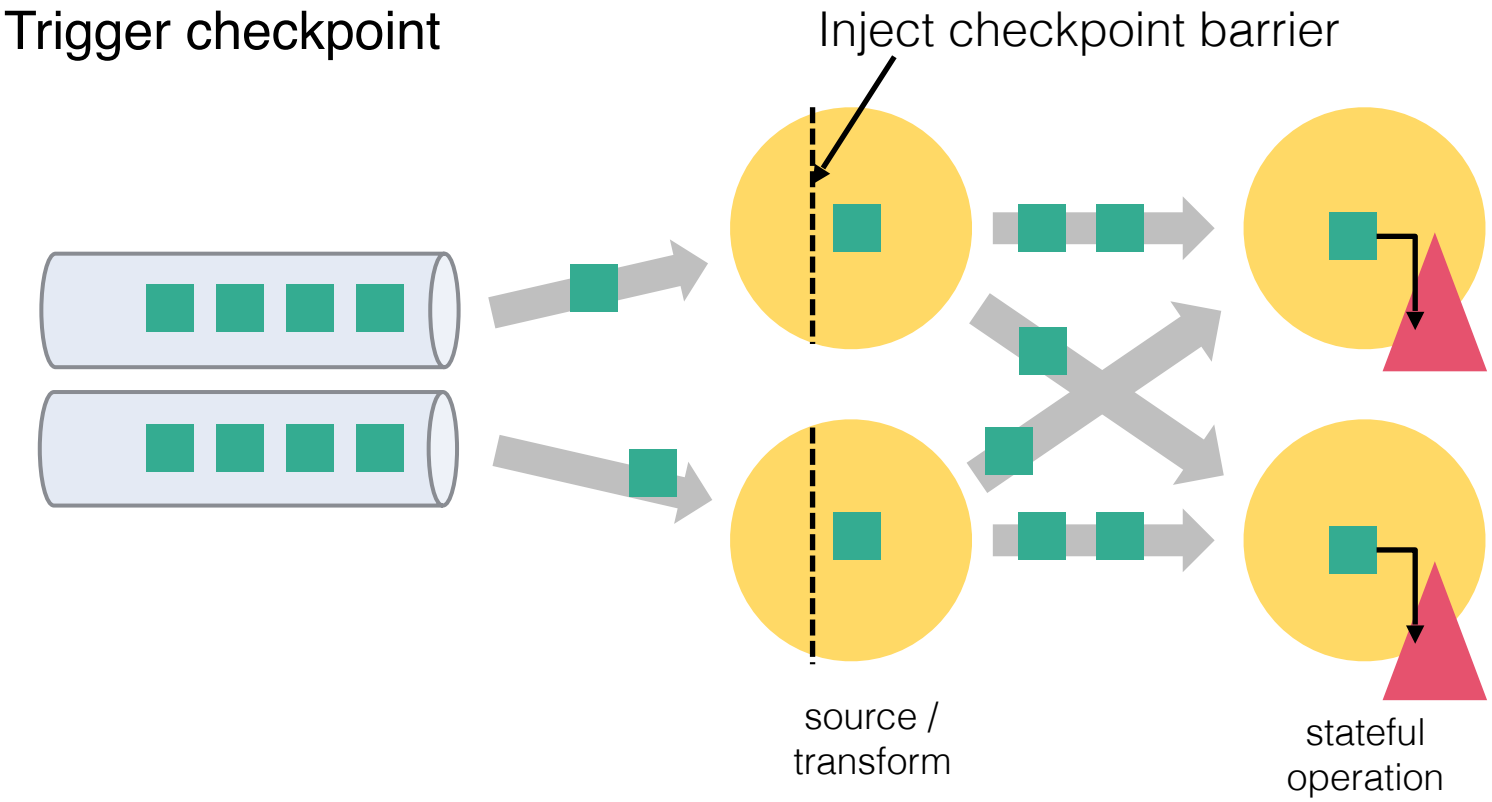


Checkpoints at scale

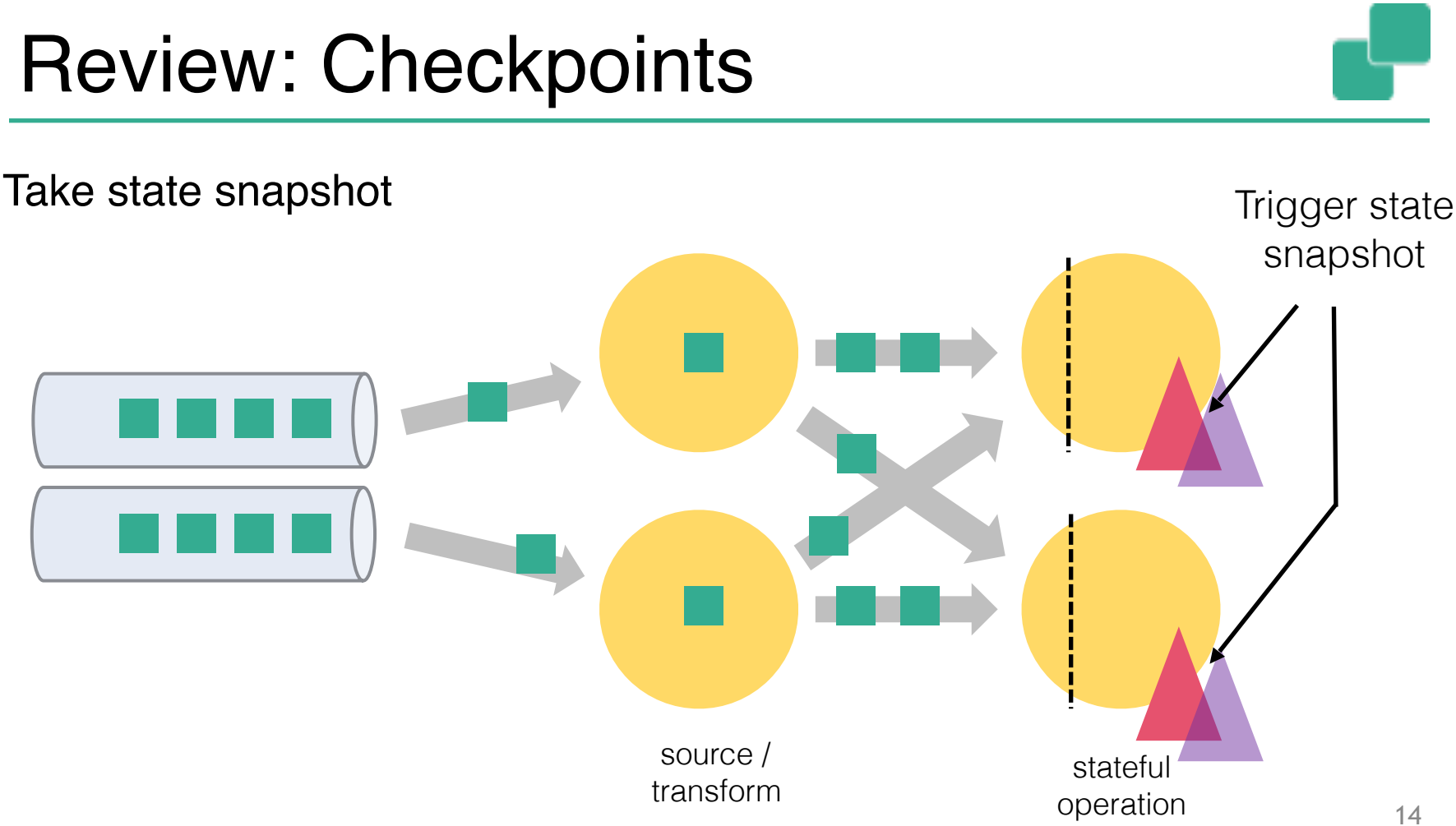
Robustly checkpointing…

…is the most important part of running a large Flink program

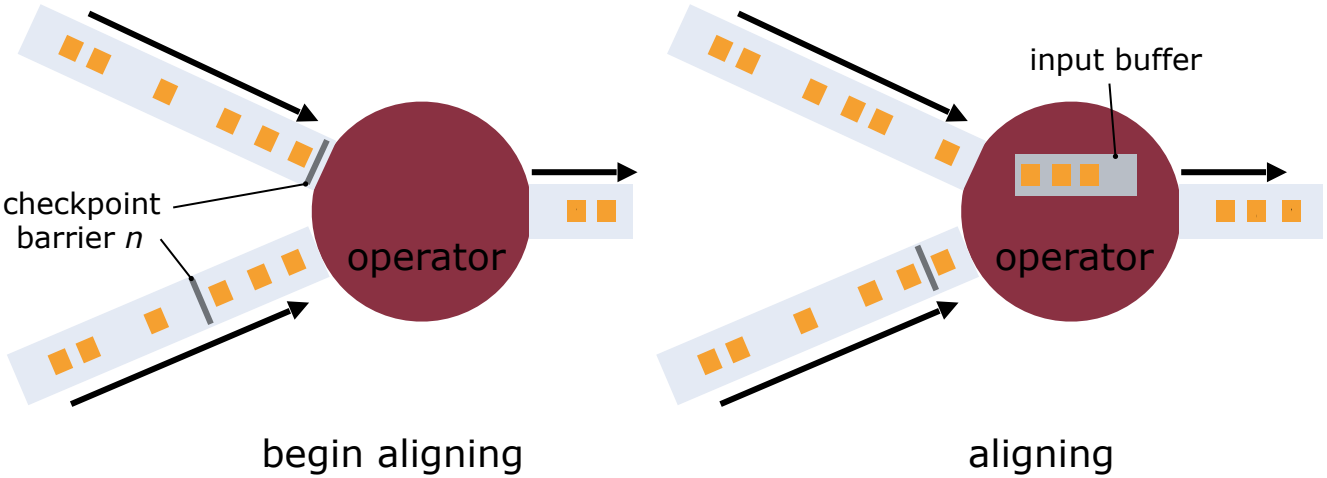
Review: Checkpoints



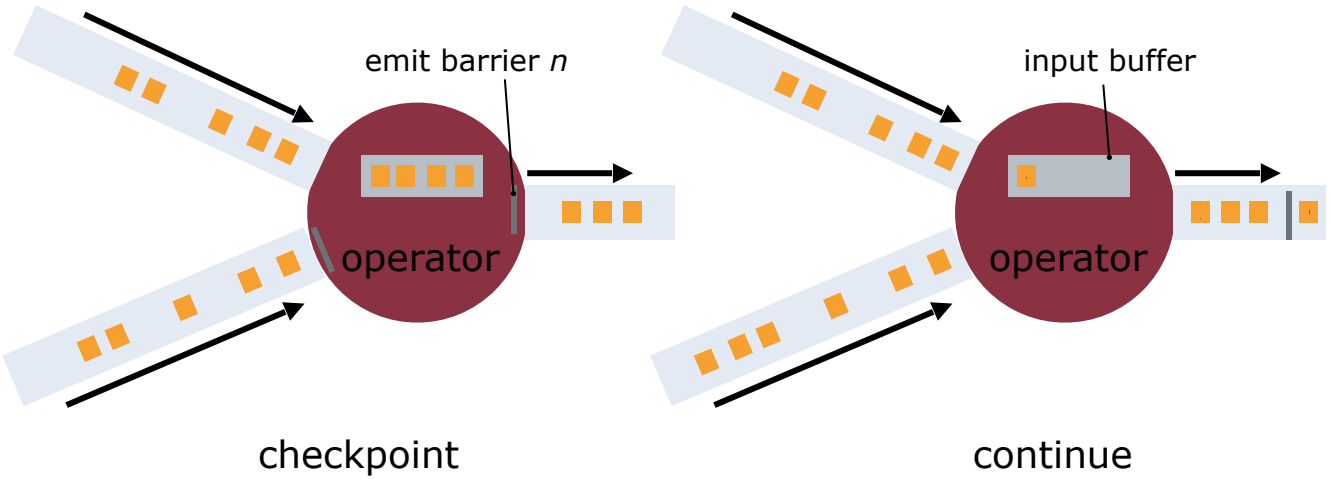
Review: Checkpoints



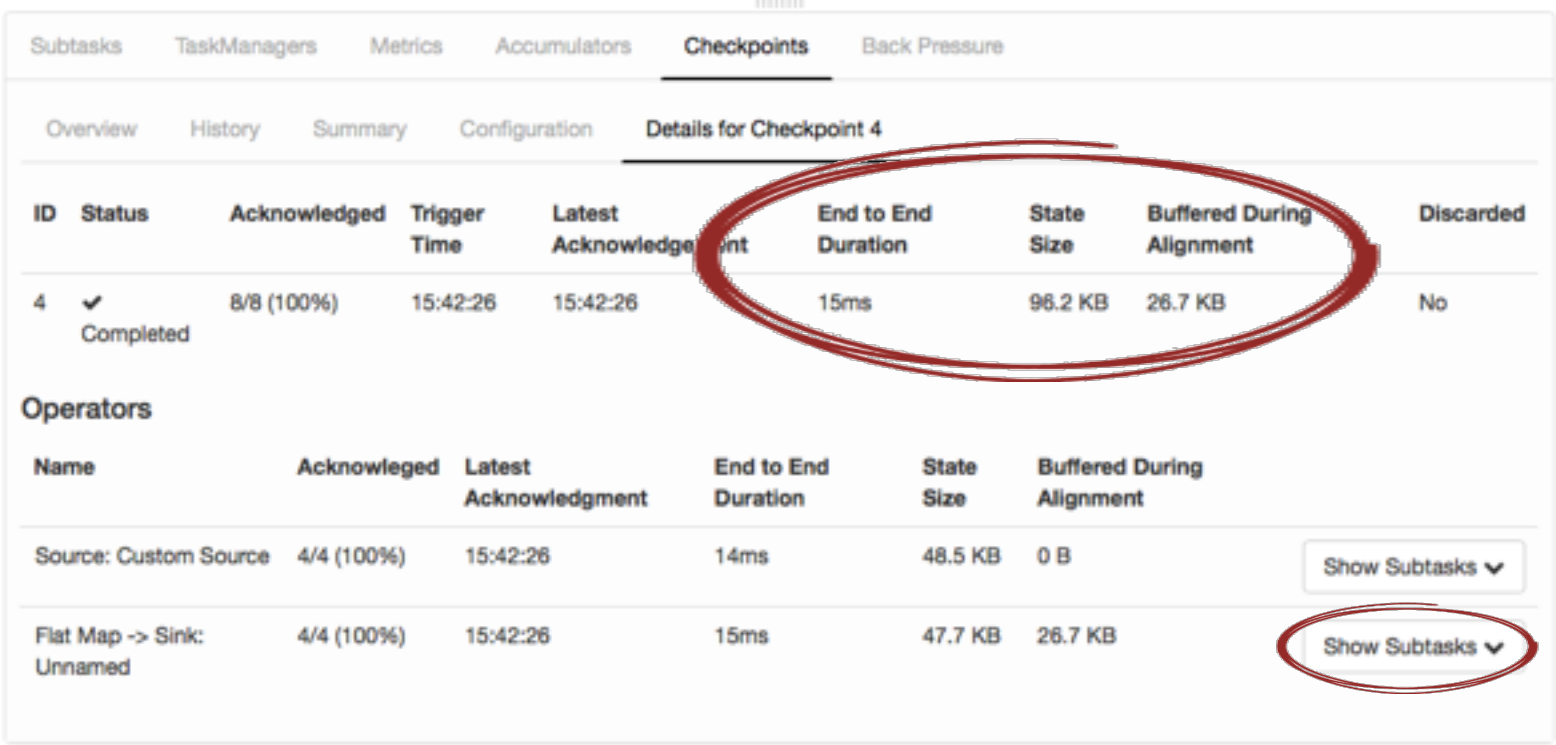
Review: Checkpoint Alignment



Review: Checkpoint Alignment



Understanding Checkpoints



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)

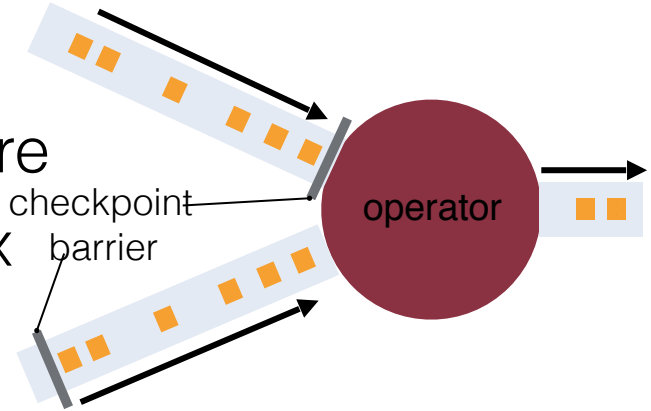
changes with incremental checkpoints

most important metric

Alignments: Limit in-flight data

- In-flight data is data "between" operators
  - On the wire or in the network buffers
  - Amount depends mainly on network buffer memory

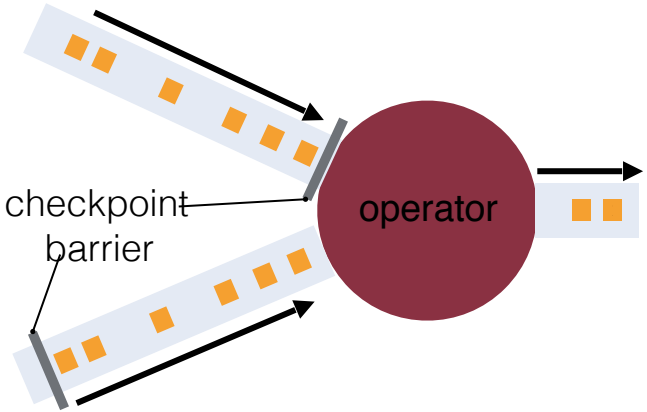
- Need some to buffer out network fluctuations / transient backpressure
- Max amount of in-flight data is max amount buffered during alignment



Alignments: Limit in-flight data

- Flink 1.2: Global pool that distributes across all tasks
  - Rule-of-thumb: set to 4 * num_shuffles * parallelism * num_slots

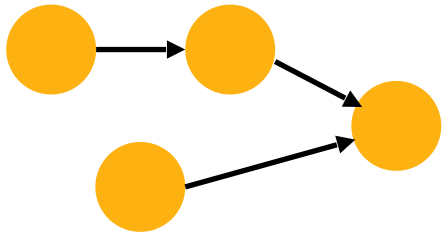
- Flink 1.3: Limits the max in-flight data automatically
  - Heuristic based on of channels and connections involved in a transfer step



Heavy alignments

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

- Big window emission concurrent to a checkpoint

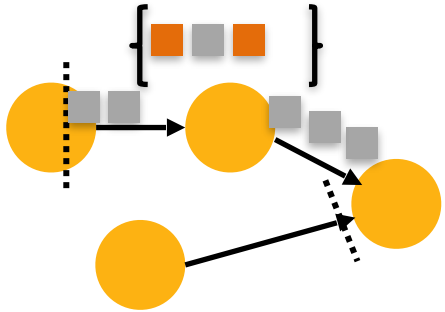


- Stall of one operator on the path

Heavy alignments

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

- Big window emission concurrent to a checkpoint

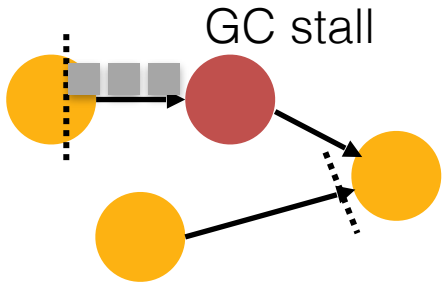


- Stall of one operator on the path

Heavy alignments

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

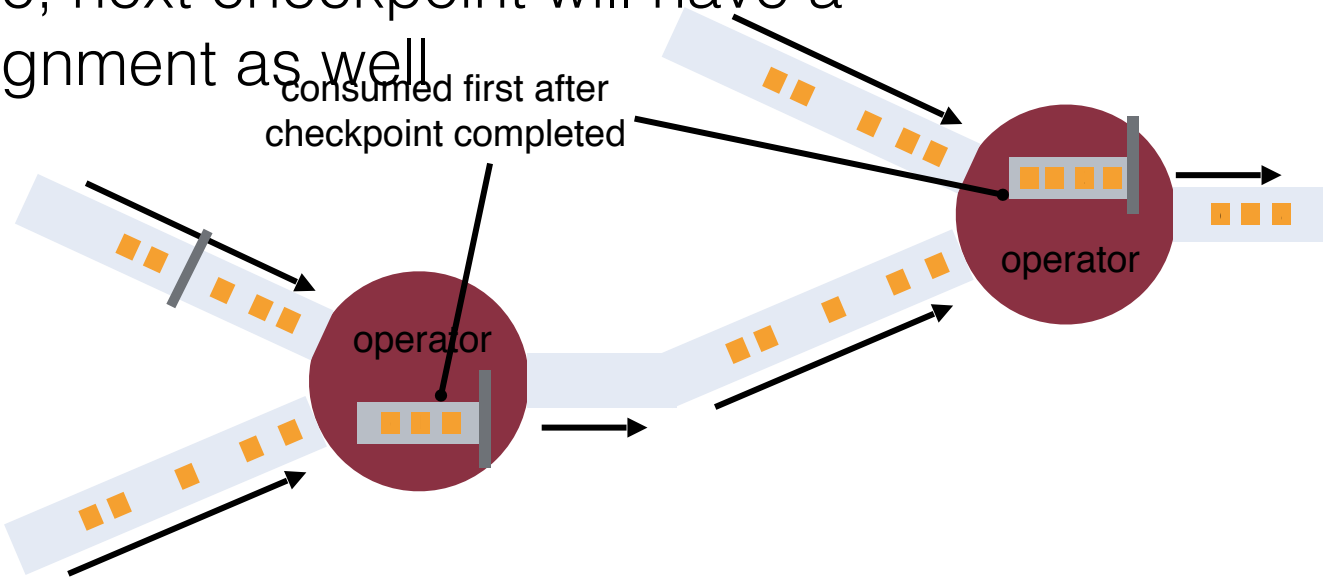
- Big window emission concurrent to a checkpoint



- 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

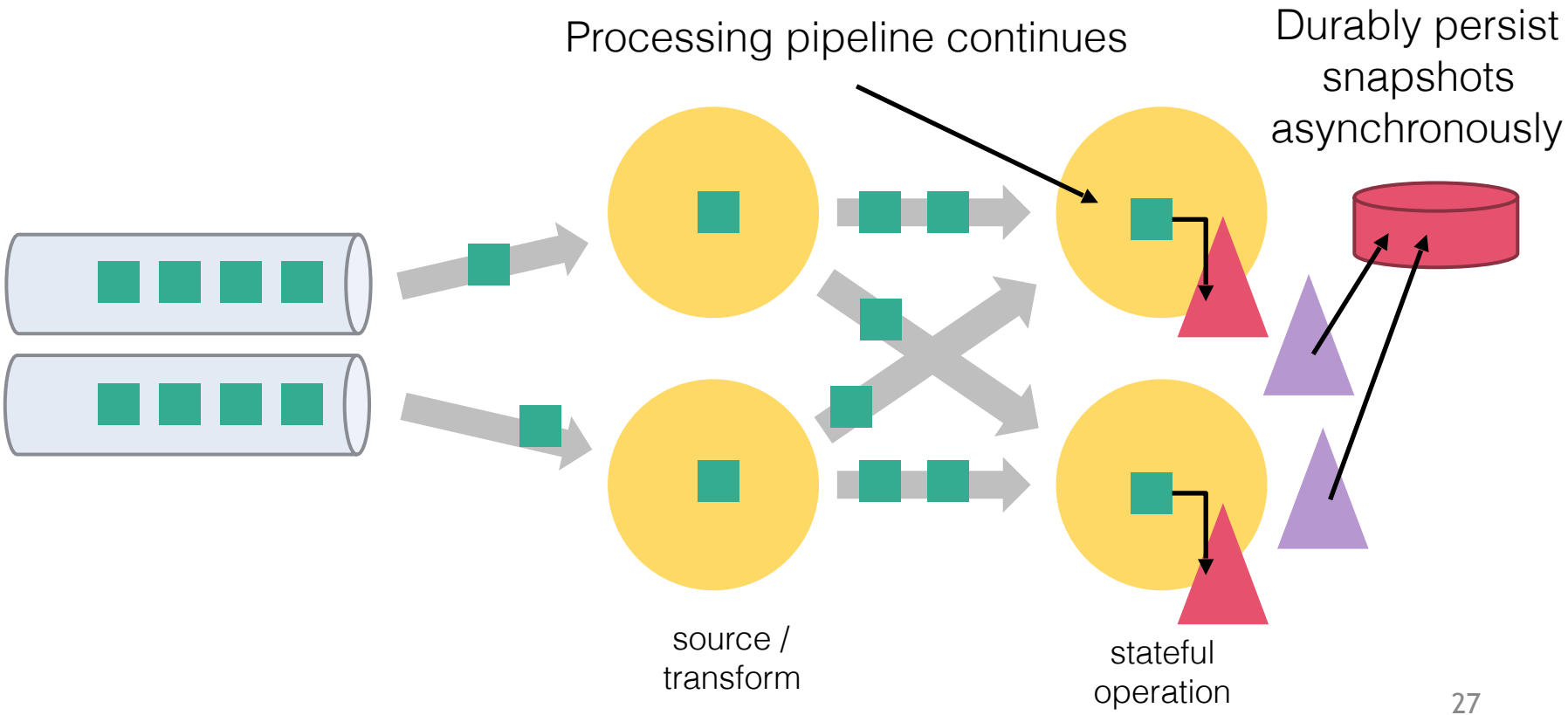


Catching up from heavy alignments

- Giving the computation time to catch up before starting the next checkpoint
  - Useful: Set the min-time-between-checkpoints

- Asynchronous checkpoints help a lot!
  - Shorter stalls in the pipelines means less build-up of in-flight data
  - Catch up already happens concurrently to state materialization

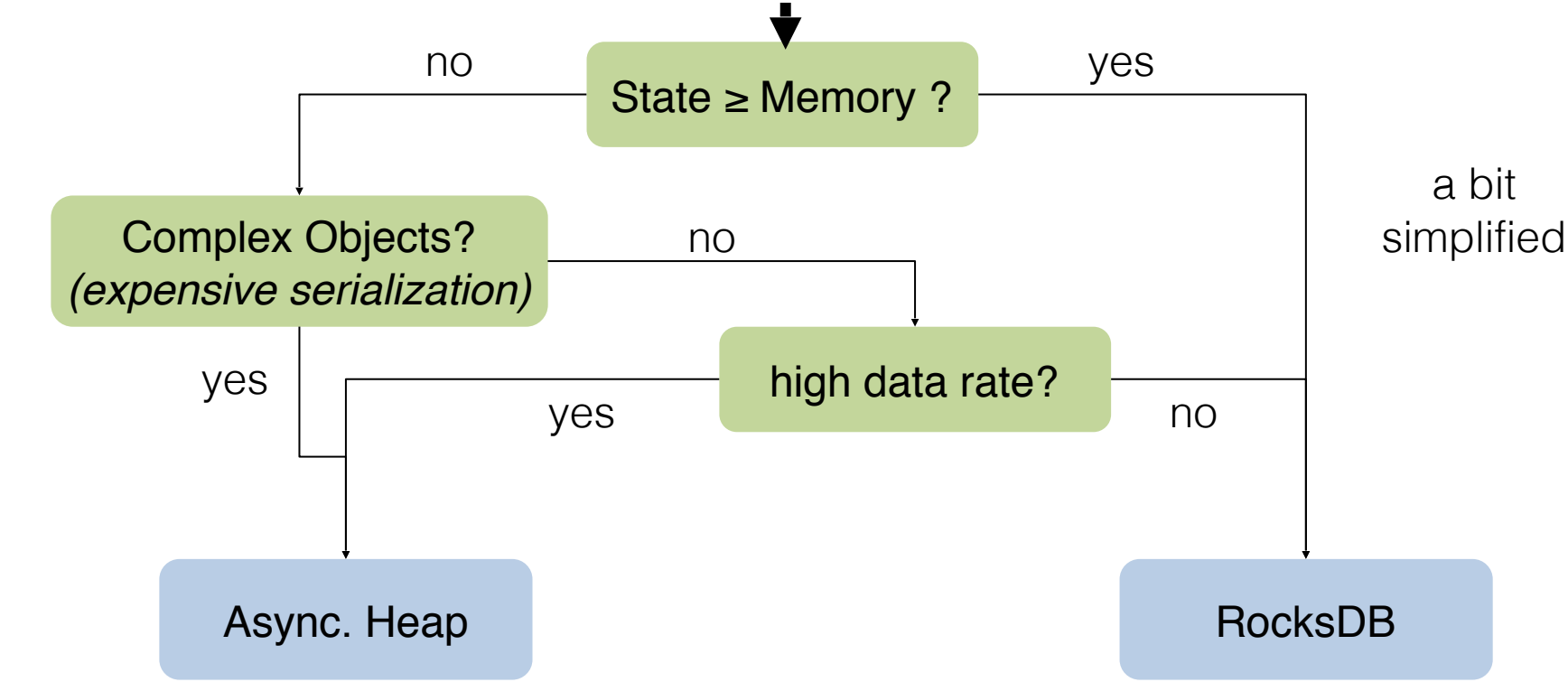
Asynchronous Checkpoints



Asynchrony of different state types

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

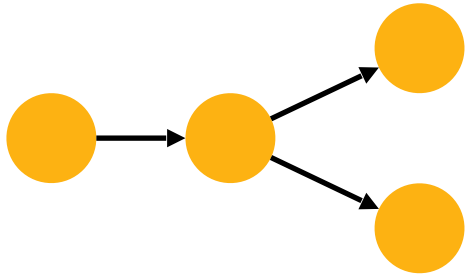
When to use which state backend?



File Systems, Object Stores, and Checkpointed State

Exceeding FS request capacity

- Job size: 4 operators
- Parallelism: 100s to 1000

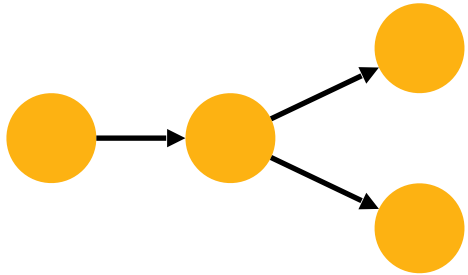


- State Backend: FsStateBackend
- State size: few KBs per operator, 100s to 1000 of files
- Checkpoint interval: few secs

- Symptom: S3 blocked off connections after exceeding 1000s HEAD requests / sec

Exceeding FS request capacity

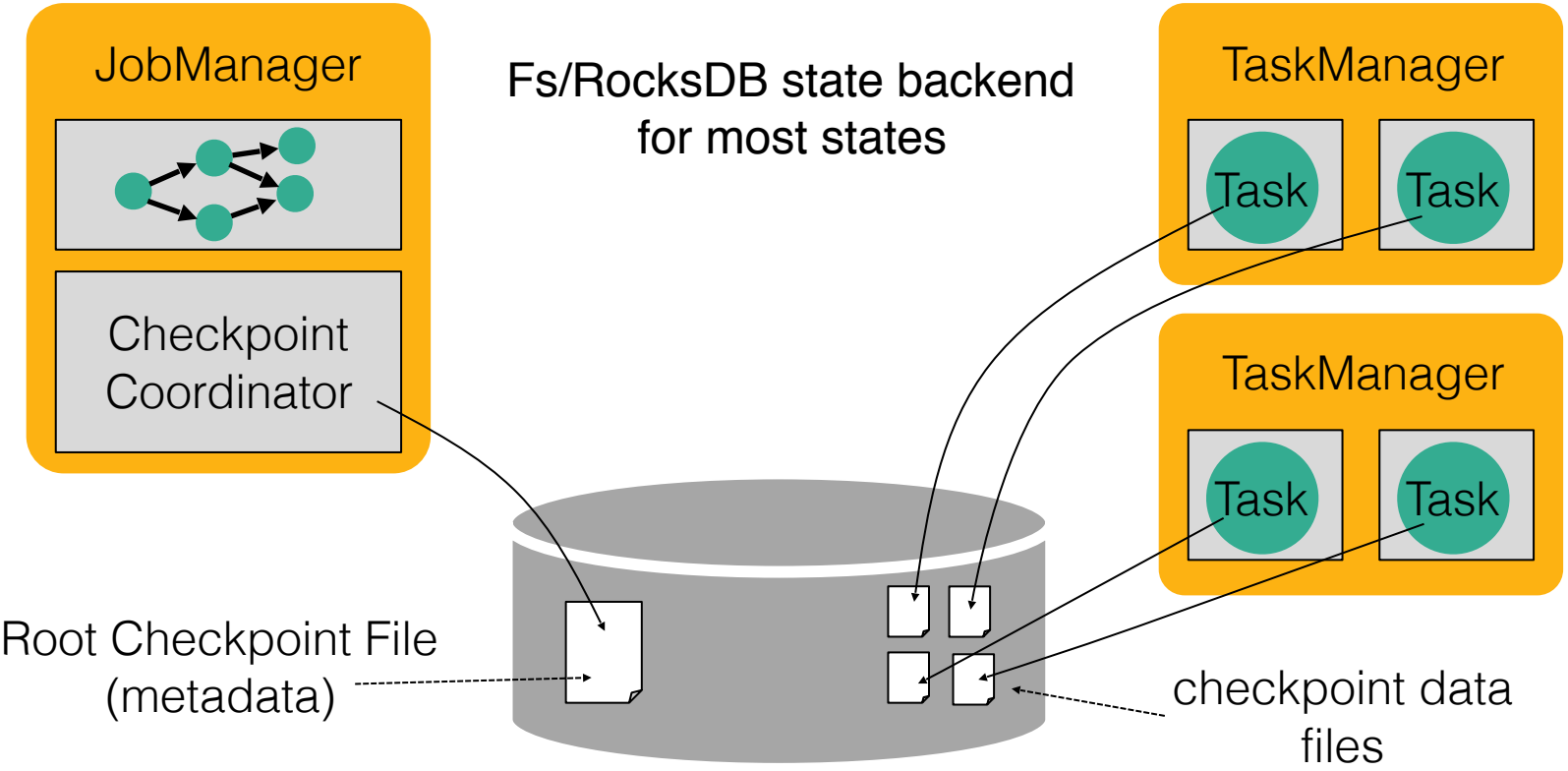
What happened?



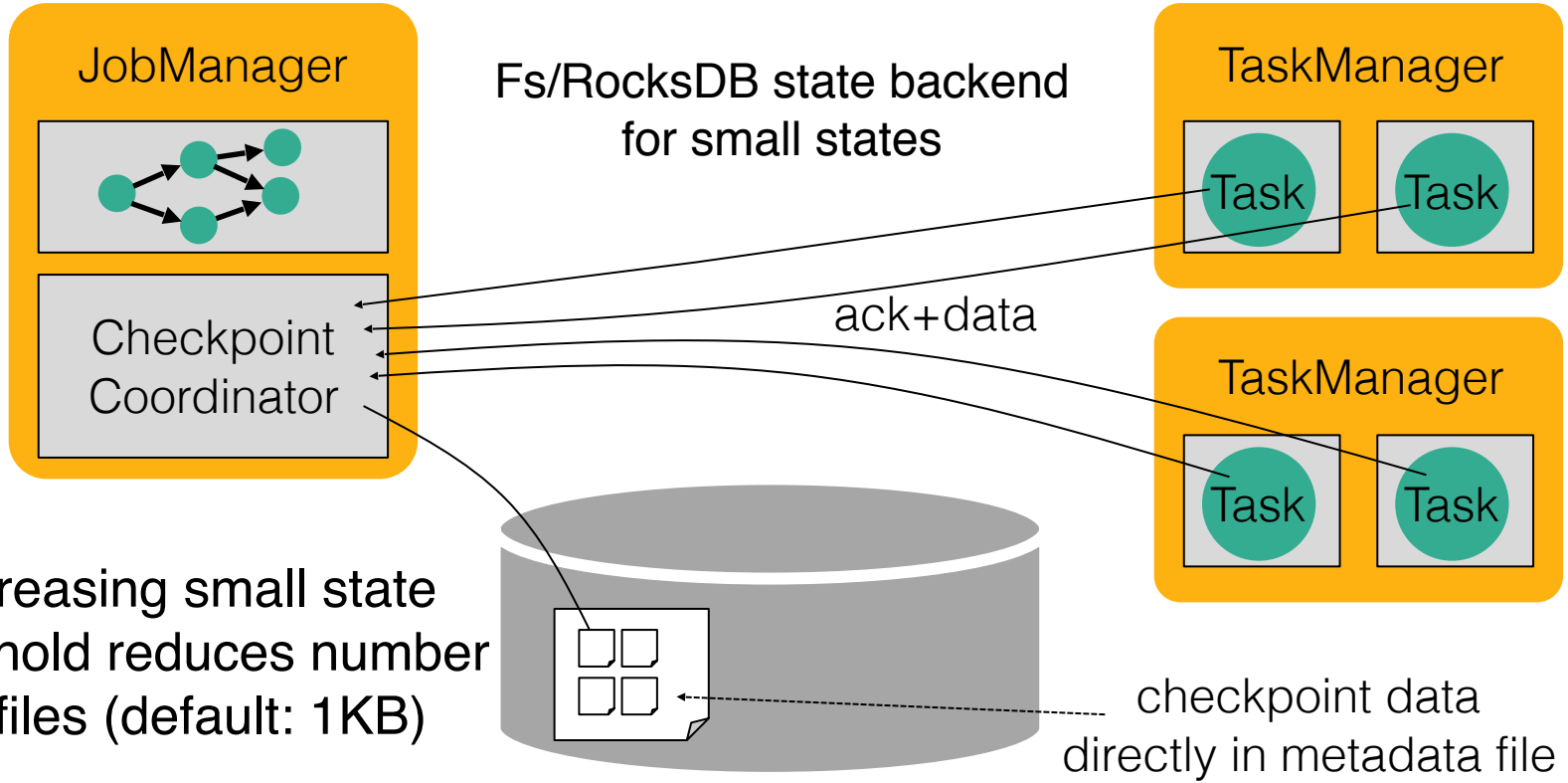
- Operators prepare state writes, ensure parent directory exists
- Via the S3 FS (from Hadoop), each **mkdirs** causes 2 HEAD requests

- **Flink 1.2:** Lazily initialize checkpoint preconditions (dirs.)
- **Flink 1.3:** Core state backends reduce assumption of directories (PUT/GET/DEL), rich file systems support them as fast paths

Reducing FS stress for small state

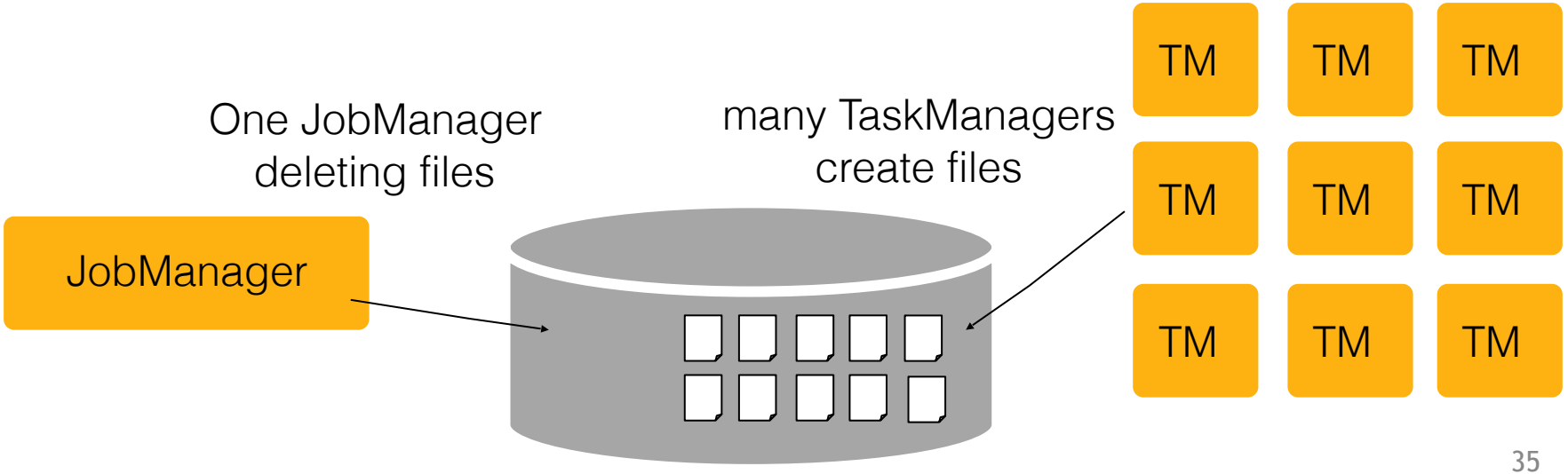


Reducing FS stress for small state



Lagging state cleanup

Symptom: Checkpoints get cleaned up too slow
State accumulates over time



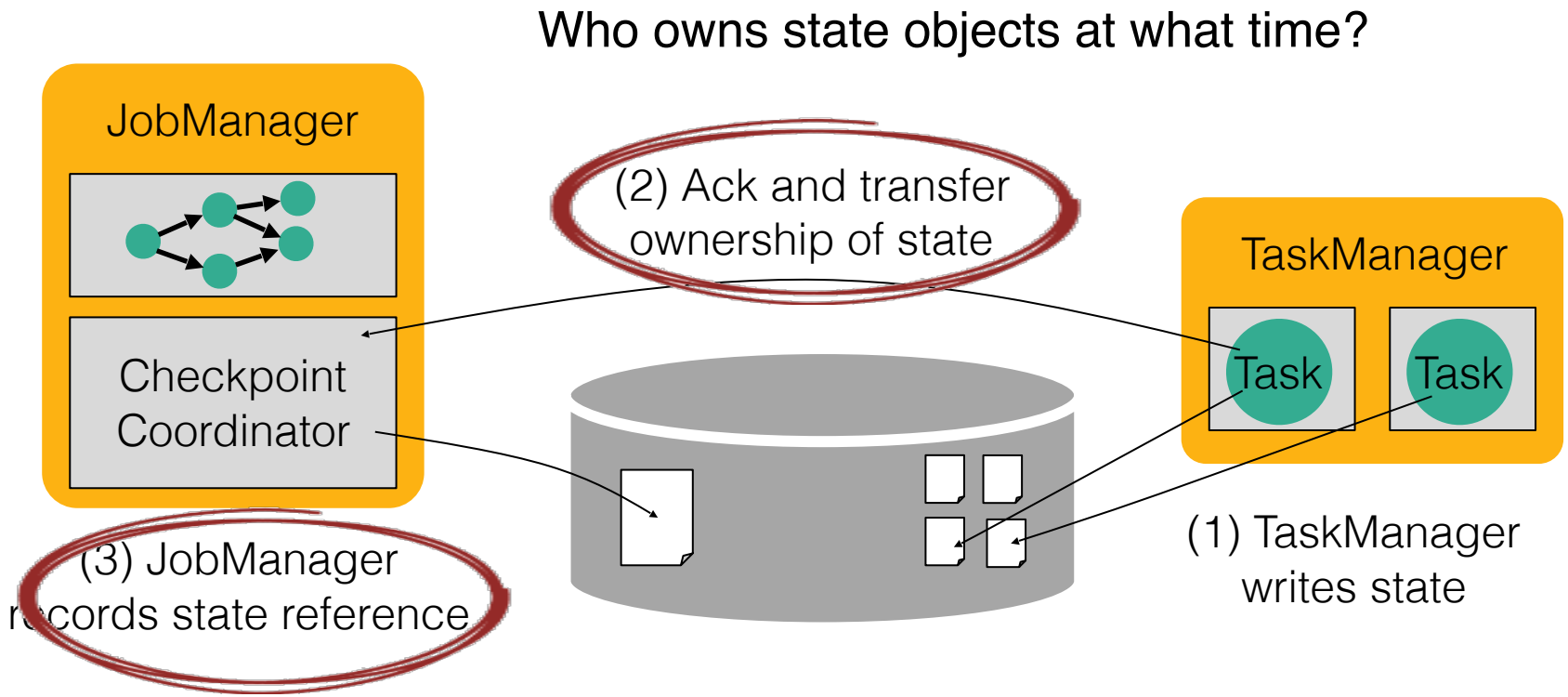
Lagging state cleanup

- **Problem:** FileSystems and Object Stores offer only synchronous requests to delete state object
  Time to delete a checkpoint may accumulates to minutes.

- **Flink 1.2:** Concurrent checkpoint deletes on the JobManager

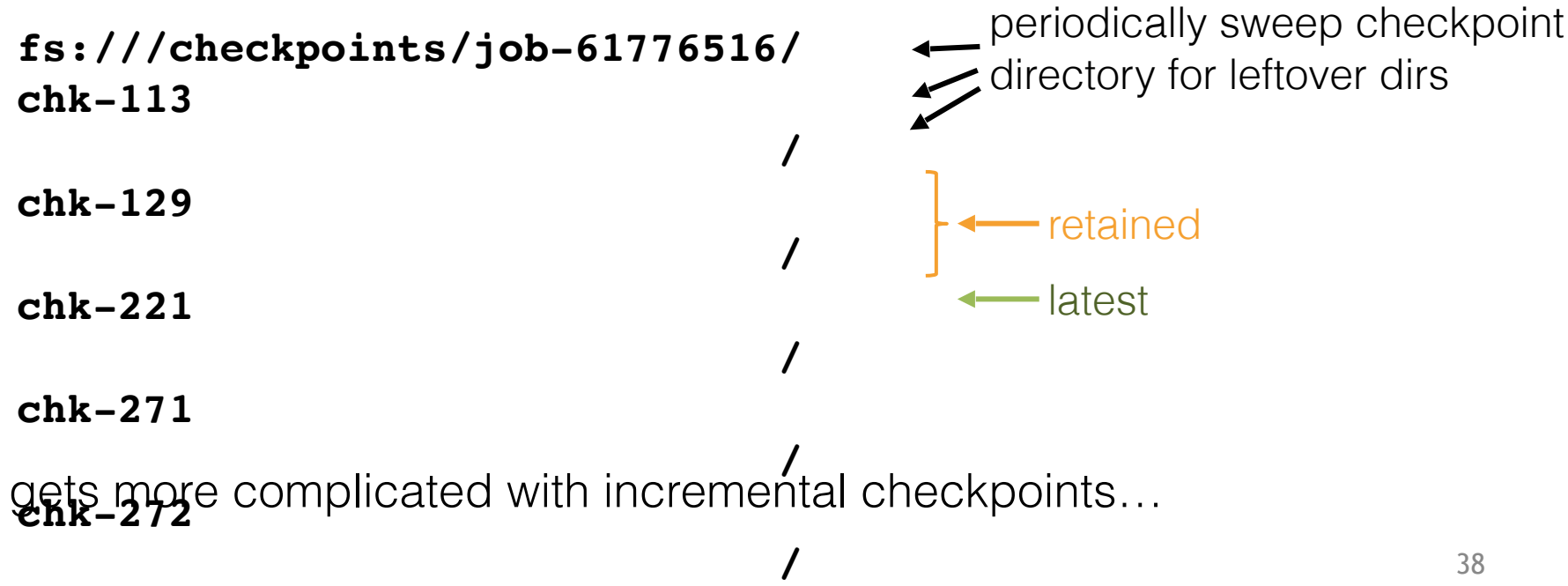
- **Flink 1.3:** For FileSystems with actual directory structure, use recursive directory deletes (one request per directory)

Orphaned Checkpoint State



Orphaned Checkpoint State

Upcoming: Searching for orphaned state



It gets more complicated with incremental checkpoints…

Conclusion &
General Recommendations

The closer you application is to saturating either
network, CPU, memory, FS throughput, etc.
the sooner an extraordinary situation causes a regression

Enough headroom in provisioned capacity means
fast catchup after temporary regressions

Be aware that certain operations are spiky
(like aligned windows)

Production test always with checkpoints ;-)

Recommendations *(part 1)*

Be aware of the inherent scalability of primitives

- Broadcasting state is useful, for example for updating rules / configs, dynamic code loading, etc.
- Broadcasting does not scale, i.e., adding more nodes does not. Don't use it for high volume joins

- Putting very large objects into a `ValueState` may mean big serialization effort on access / checkpoint
- If the state can be mappified, use `MapState` – it performs much better

Recommendations *(part 2)*

If you are about recovery time

- Having spare TaskManagers helps bridge the time until backup TaskManagers come online

- Having a spare JobManager can be useful
  - Future: JobManager failures are non disruptive

Recommendations *(part 3)*

If you care about CPU efficiency, watch your serializers

- JSON is a flexible, but awfully inefficient data type
- Kryo does okay - make sure you register the types

- Flink's directly supported types have good performance basic types, arrays, tuples, …
- Nothing ever beats a custom serializer ;-)

Thank you!

Questions?

dataArtisans