Keep it going - How to reliably and efficiently operate Apache Flink

Robert Metzger
@rmetzger

dataArtisans

About this talk

- Part of the Apache Flink training offered by data Artisans
- Subjective collection of most frequent ops questions on the user mailing list.

Topics

- Capacity planning
- Deployment best practices
- Tuning
  - Work distribution
  - Memory configuration
  - Checkpointing
  - Serialization
- Lessons learned

Capacity Planning

First step: do the math!

- Think through the resource requirements of your problem
  - Number of keys, state per key
  - Number of records, record size
  - Number of state updates
  - What are your SLAs? (downtime, latency, max throughput)
- What resources do you have?
  - Network capacity (including Kafka, HDFS, etc.)
  - Disk bandwidth (RocksDB relies on the local disk)
  - Memory
  - CPUs

Establish a baseline

- Normal operation should be able to avoid back pressure
- Add a margin for recovery – these resources will be used to "catch up"
- Establish your baseline with checkpointing enabled

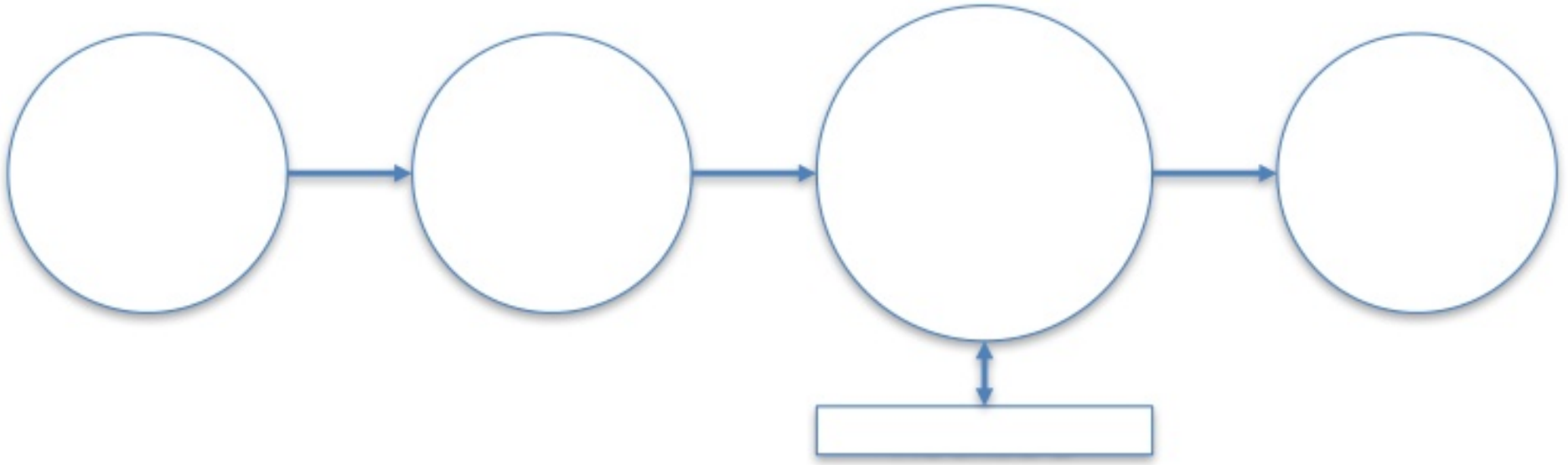
Consider spiky loads

- For example, operators downstream from a Window won't be continuously busy

- How much downstream parallelism is required depends on how quickly you expect to process these spikes

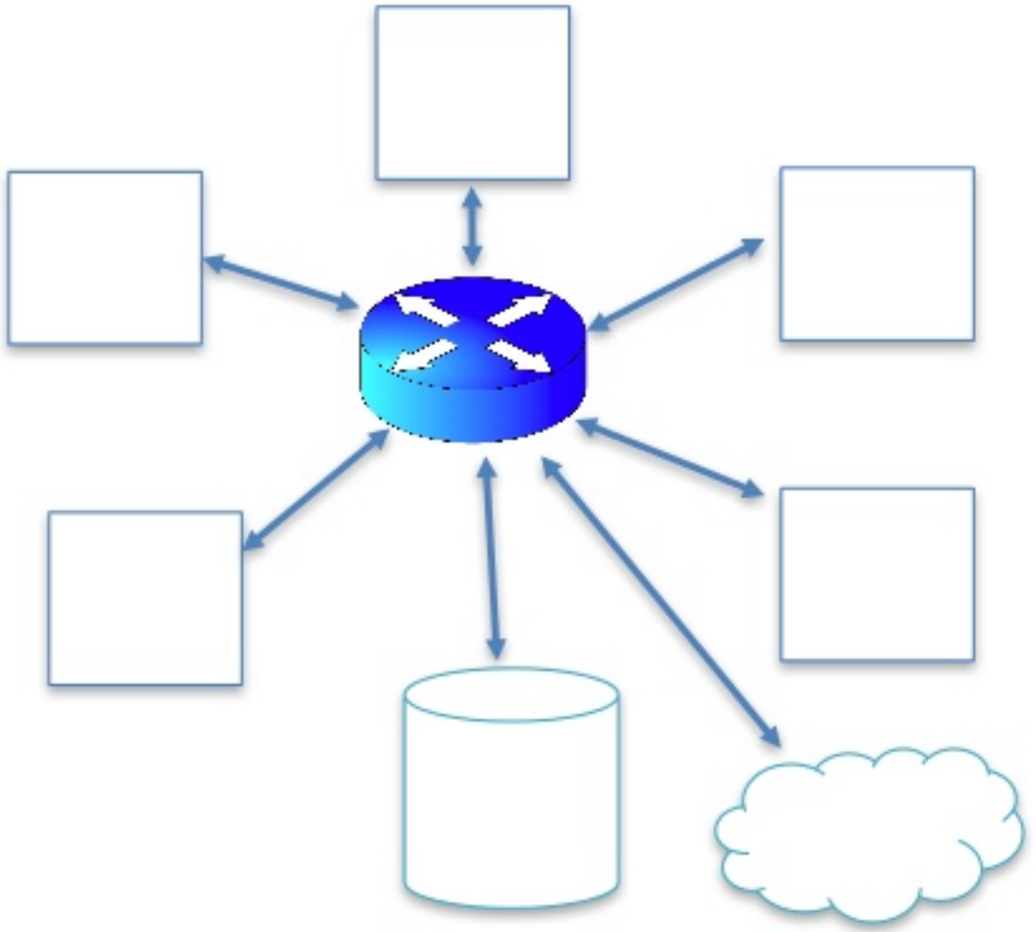
Example: The Setup

- Data:
  - Message size: 2 KB
  - Throughput: 1,000,000 msg/sec
  - Distinct keys: 500,000,000 (aggregation in window: 4 longs per key)
  - Checkpoint every minute



Example: The setup

- Hardware:
  - 5 machines
  - 10 gigabit Ethernet
  - Each machine running a Flink TaskManager
  - Disks are attached via the network
- Kafka is separate



Example: A machine's perspective

10 Gigabit Ethernet (Full Duplex)
In: 1250 MB/s

*2 KB * 1,000,000 = 2GB/s*
2GB/s / 5 machines = 400 MB/s

TaskManager *n*

Kafka Source

keyBy

window

Kafka Sink

10 Gigabit Ethernet (Full Duplex)
Out: 1250 MB/s

400MB/s / 5 receivers = 80MB/s
1 receiver is local, 4 remote:
*4 * 80 = 320 MB/s out*

Excursion 1: Window emit

How much data is the window emitting?

Recap: 500,000,000 unique users (4 longs per key)
Sliding window of 5 minutes, 1 minute slide

Assumption: For each user, we emit 2 ints (user_id, window_ts) and 4 longs from the aggregation = 2 * 4 bytes + 4 * 8 bytes = 40 bytes per key

100,000,000 (users) * 40 bytes = **4 GB every minute from each machine**

Example: A machine's perspective

10 Gigabit Ethernet (Full Duplex)
In: 1250 MB/s

*2 KB * 1,000,000 = 2GB/s*
2GB/s / 5 machines = 400 MB/*s*

TaskManager *n*

Kafka Source

keyBy

window

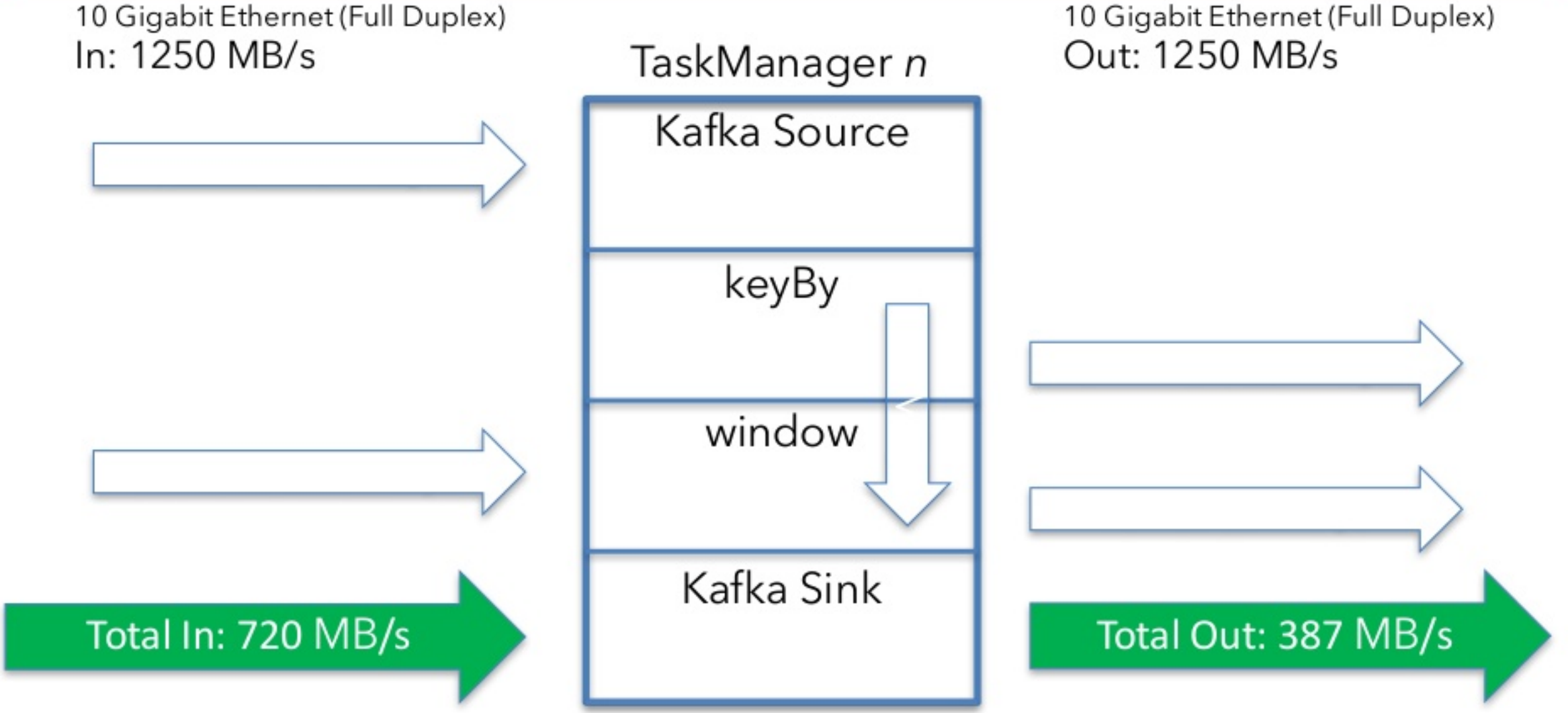
Kafka Sink

10 Gigabit Ethernet (Full Duplex)
Out: 1250 MB/s

*400*MB/*s / 5 receivers = 80*MB/*s*
*1 receiver is local, 4 remote: 4 * 80 = 320* MB/*s out*

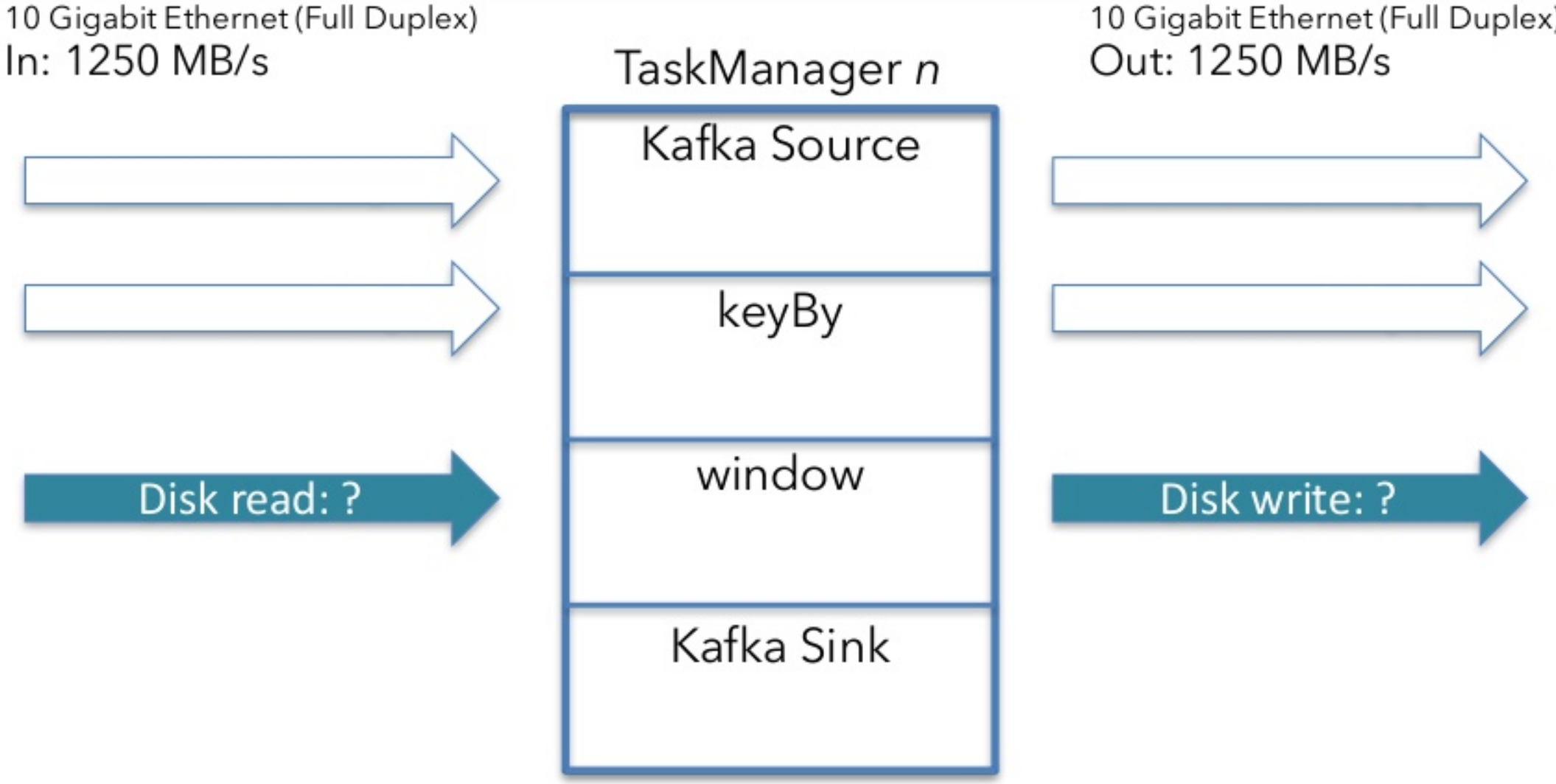
4 GB / minute => 67 MB/*second (on average)*

Example: Result



Example: Result

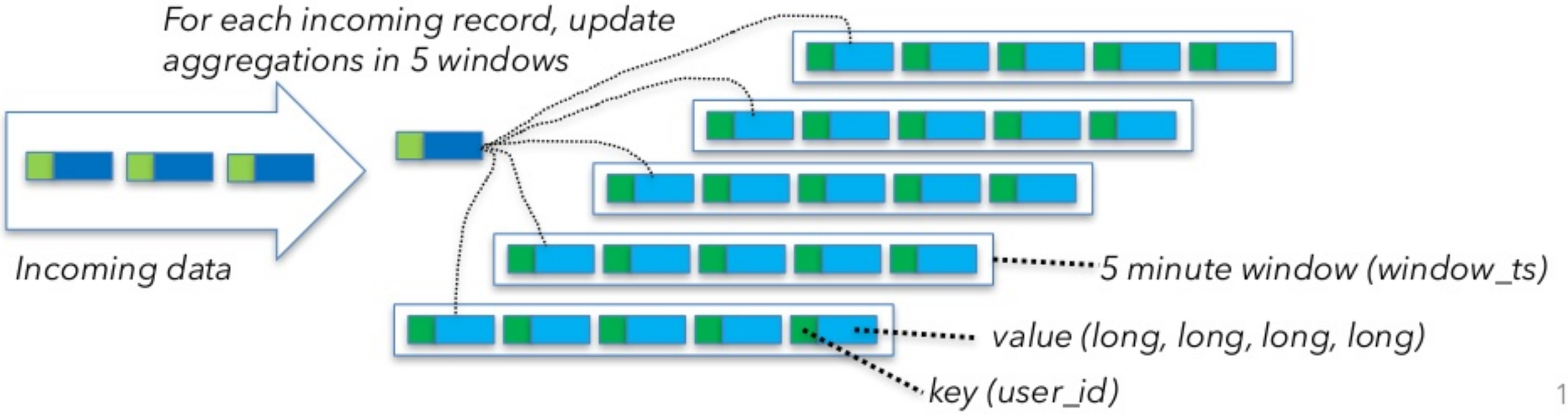
Example: Intermediate Result



Excursion 2: Window state access

How is the Window operator accessing state?

Recap: 1,000,000 msg/sec. Sliding window of 5 minutes, 1 minute slide
Assumption: For each user, we store 2 ints (user_id, window_ts) and 4 longs from the aggregation = 2 * 4 bytes + 4 * 8 bytes = **40 bytes per key**



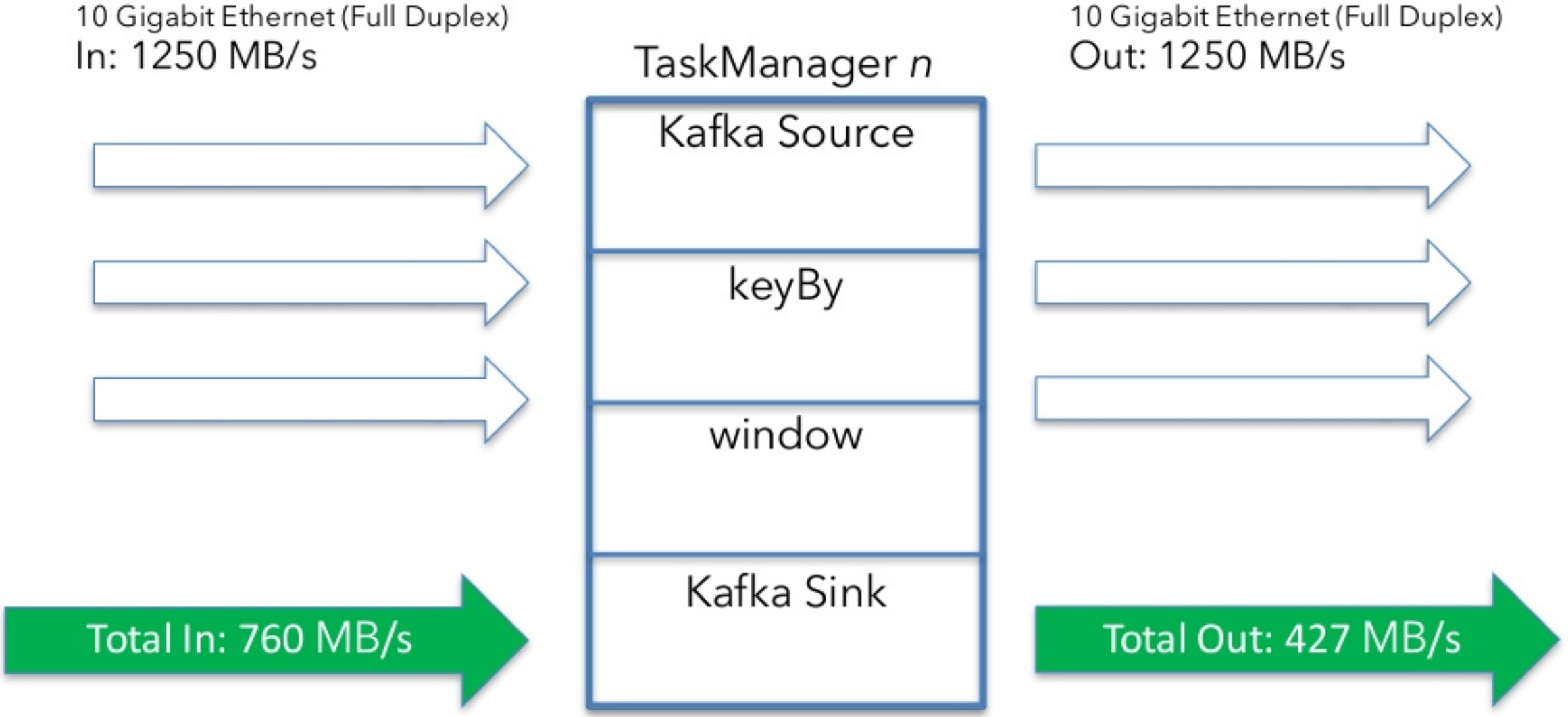
Excursion 2: Window state access

How is the Window operator accessing state?

For each key-value access, we need to retrieve 40 bytes from disk, update the aggregates and put 40 bytes back

*per machine: 40 bytes * 5 windows * 200,000 msg/sec =* ***40 MB/s***

Example: Intermediate Result



Excursion 3: Checkpointing

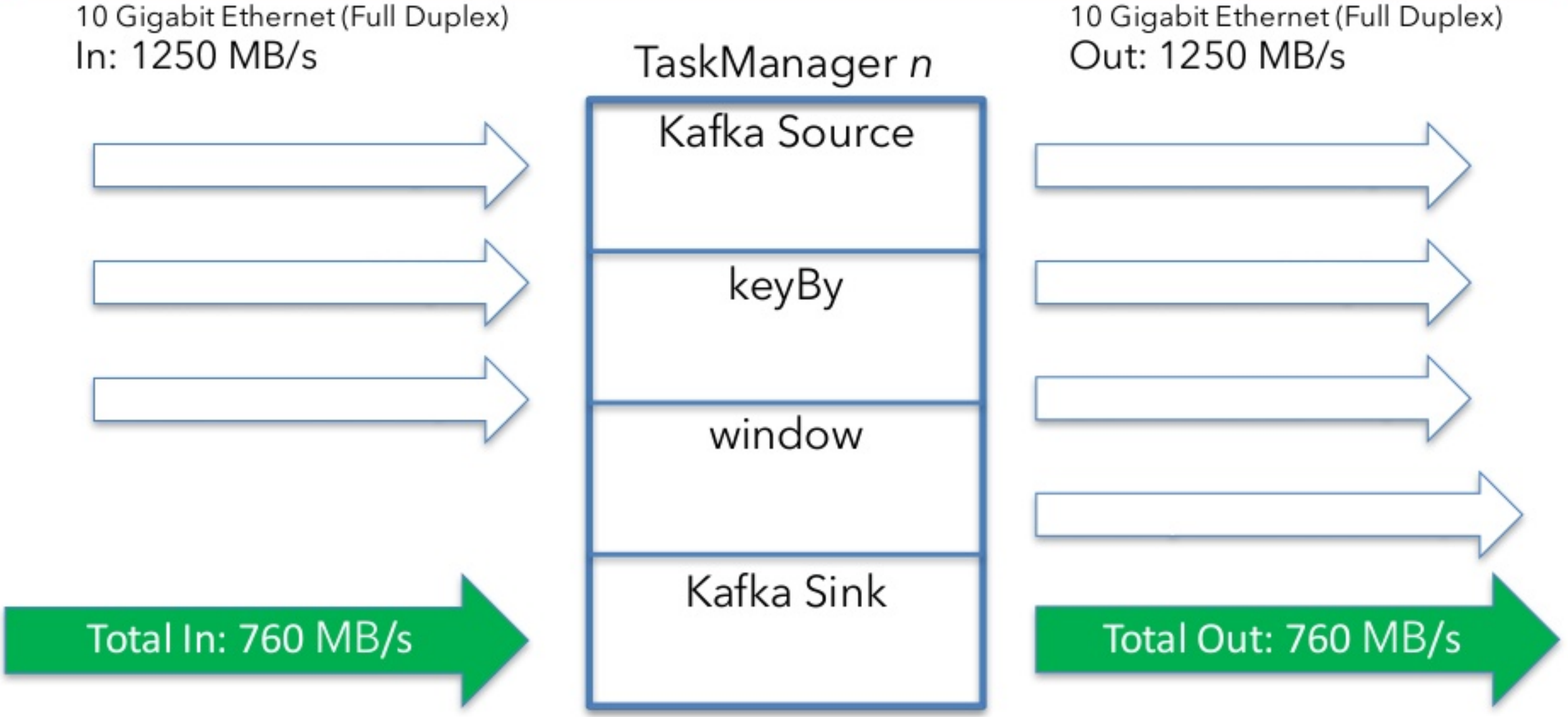
How much state are we checkpointing?

per machine: *40 bytes * 5 windows * 100,000,000 keys = 20 GB*

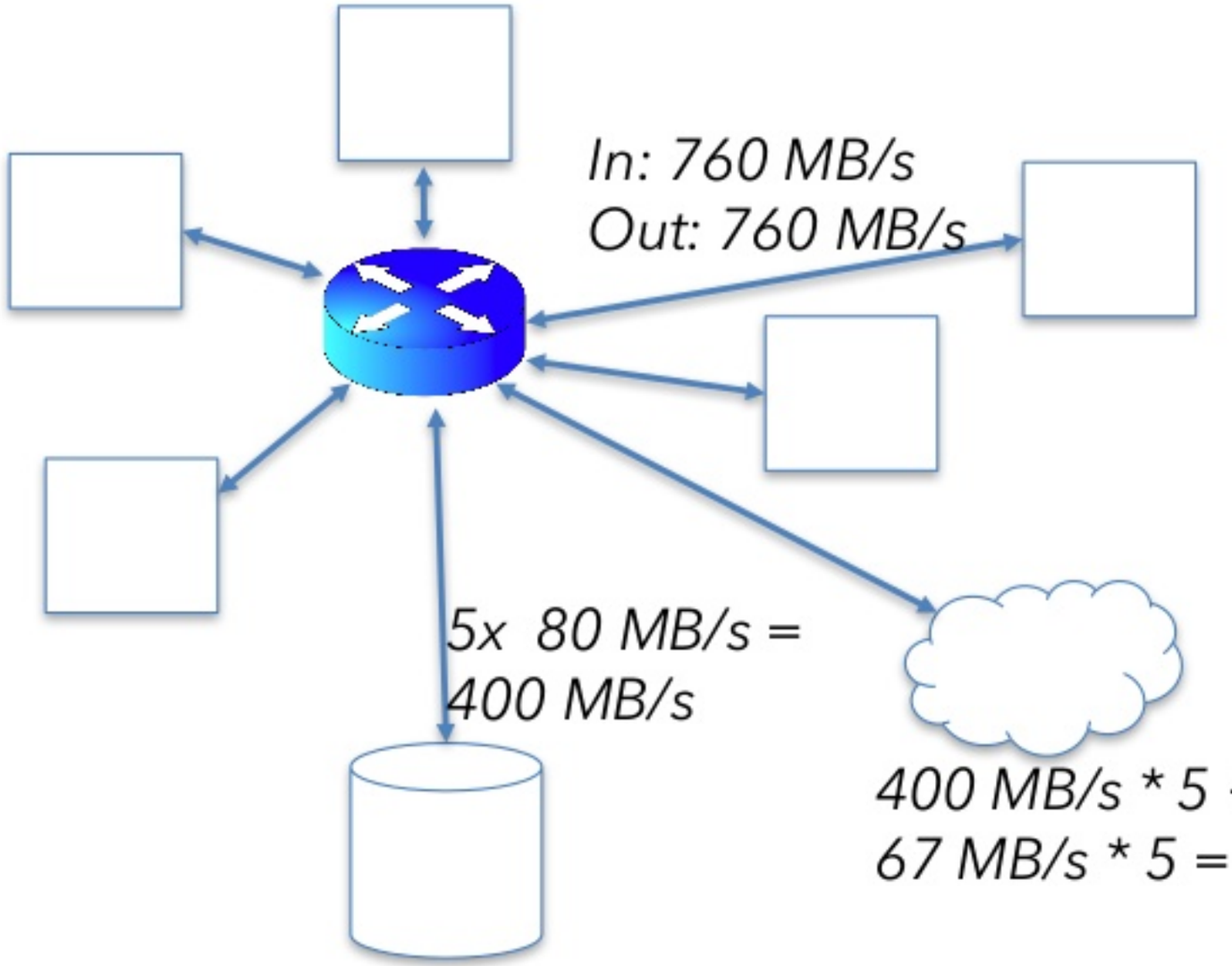
We checkpoint every minute, so

20 GB / 60 seconds = ***333 MB/s***

Example: Final Result



Example: Network requirements



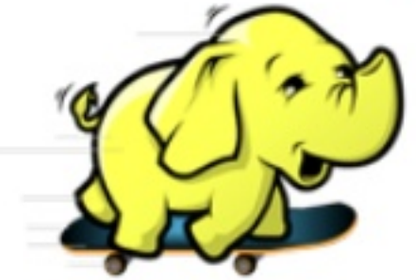
Overall network traffic:
2 * 760 * 5 + 400 + 2335
= 10335 *MB/s*
= **82,68 Gigabit/s**

Things you should consider before putting a job in production

Deployment Best Practices

Deployment Options

- Hadoop YARN integration
  - Cloudera, Hortonworks, MapR, …
  - Amazon Elastic MapReduce (EMR), Google Cloud dataproc



- Mesos & DC/OS integration

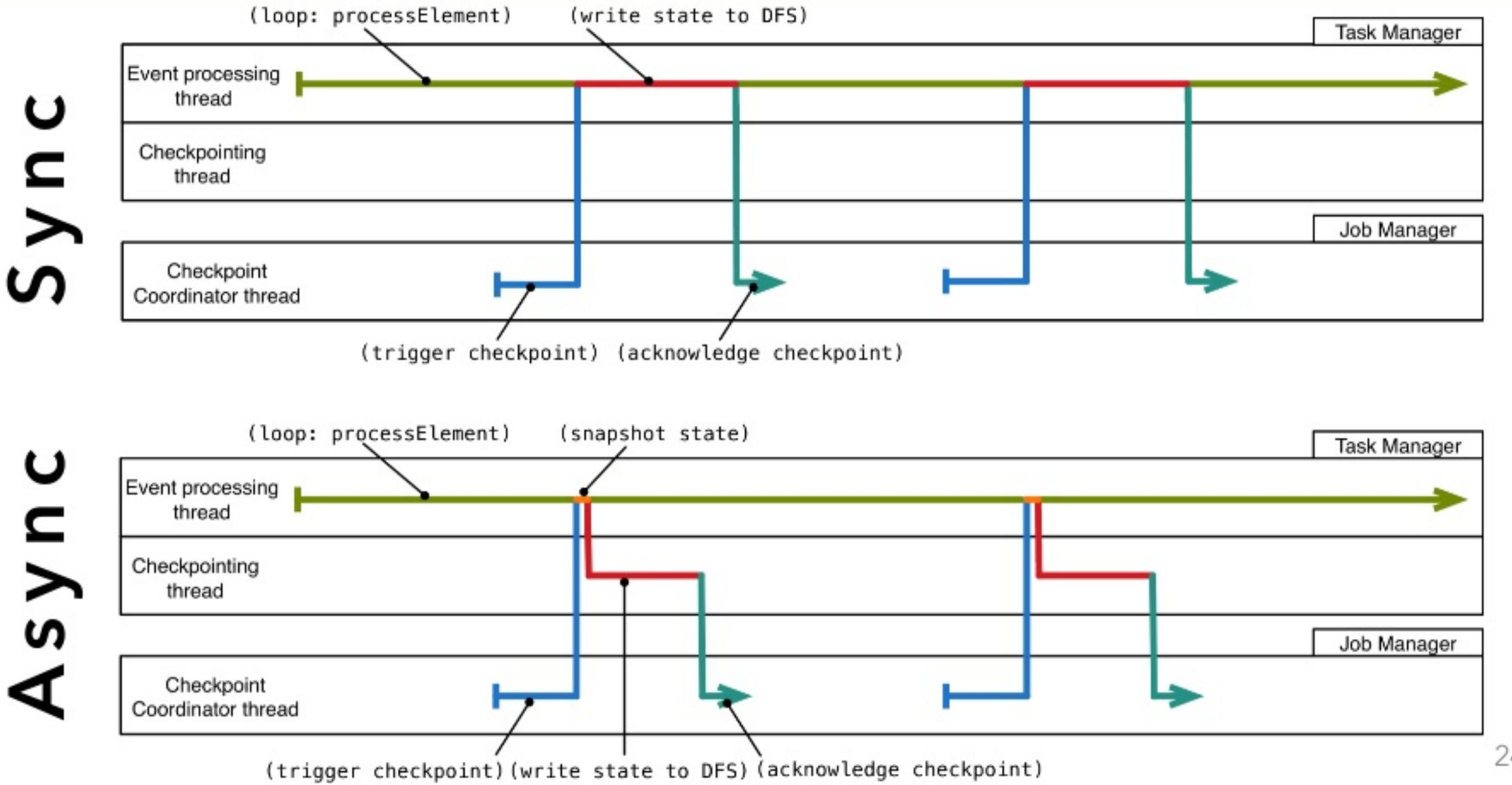
- Standalone Cluster (“native”)
  - provided bash scripts
  - provided Docker images

Flink in containerland
DAY 3 / 3:20 PM - 4:00 PM
MASCHINENHAUS

Choose your state backend

Name	Working state	State backup	Snapshotting
RocksDBStateBackend	Local disk (tmp directory)	Distributed file system	Asynchronously
• Good for state larger than available memory • Supports incremental checkpoints • Rule of thumb: 10x slower than memory-based backends			
FsStateBackend	JVM Heap	Distributed file system	Synchronous / Async
• Fast, requires large heap			
MemoryStateBackend	JVM Heap	JobManager JVM Heap	Synchronous / Async
• Good for testing and experimentation with small state (locally)			

Asynchronous Snapshotting



Check the production readiness list

- Explicitly **set the max parallelism** for rescaling
  - 0 < parallelism <= max parallelism <= 32768
  - Max parallelism > 128 has some impact on performance and state size
- **Set UUIDs for all operators** to allow changing the application between restores
  - By default Flink generates UUIDs
- Use the new *ListCheckpointed* and *CheckpointedFunction* interfaces

Production Readiness Checklist: https://ci.apache.org/projects/flink/flink-docs-release-1.3/ops/production_ready.html

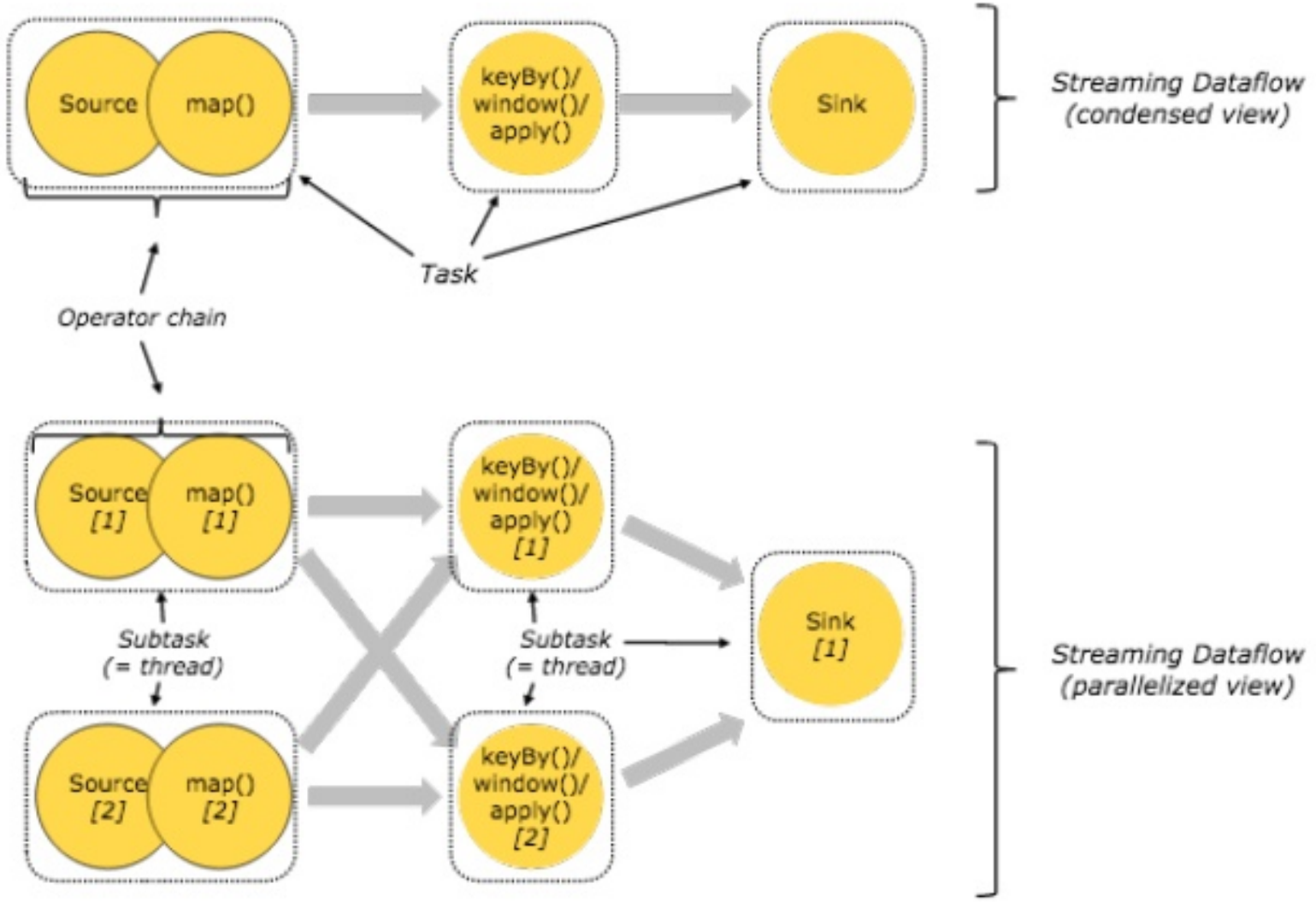
Tuning: CPU usage / work distribution

Configure parallelism / slots

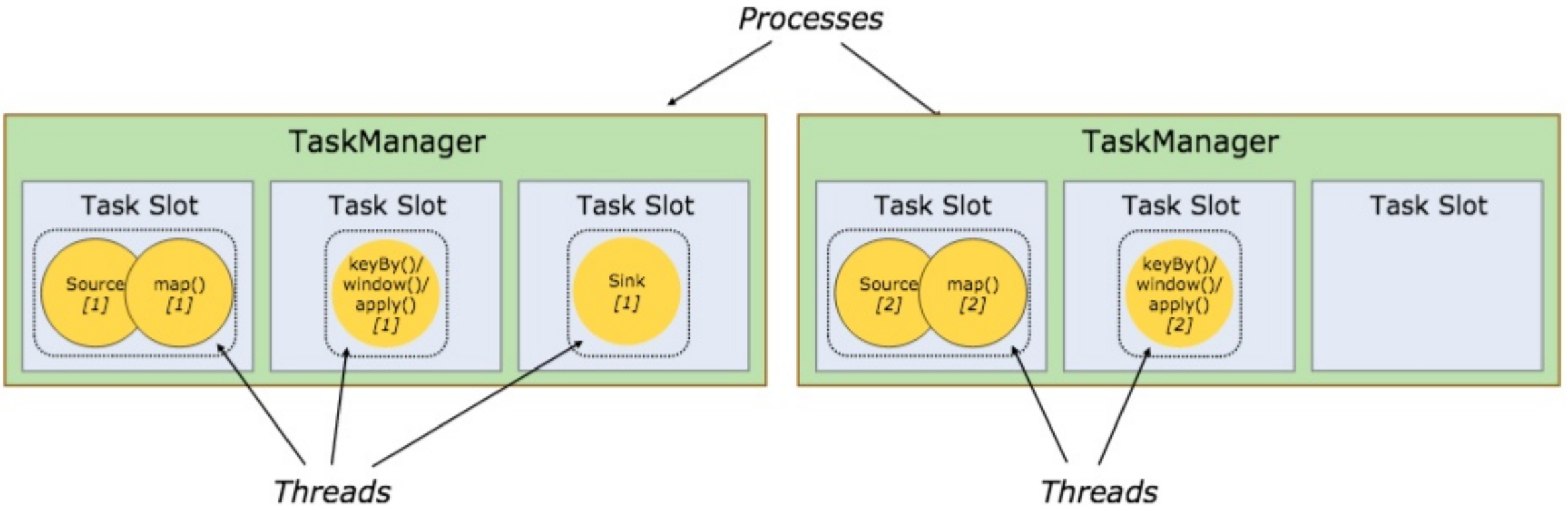
- These settings influence how the work is spread across the available CPUs
- 1 CPU per slot is common

- multiple CPUs per slot makes sense if one slot (i.e. one parallel instance of the job) performs many CPU intensive operations

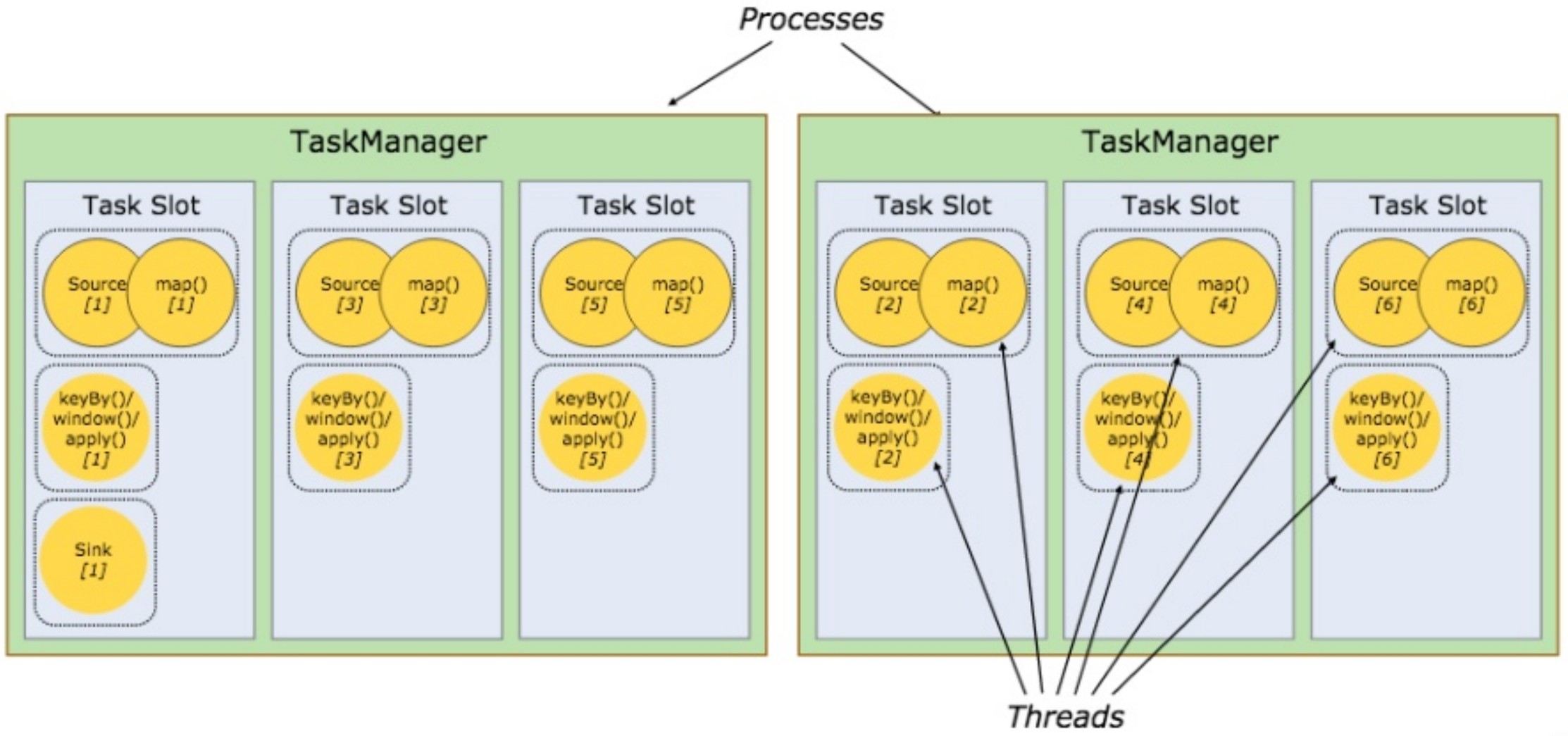
Operator chaining



Task slots


Processes
TaskManager
Task Slot
Source
[1]
map()
[1]
Task Slot
keyBy()/
window()/
apply()
[1]
Task Slot
Sink
[1]
TaskManager
Task Slot
Source
[2]
map()
[2]
Task Slot
keyBy()/
window()/
apply()
[2]
Task Slot
Threads
Threads

Slot sharing (parallelism now 6)



Benefits of slot sharing

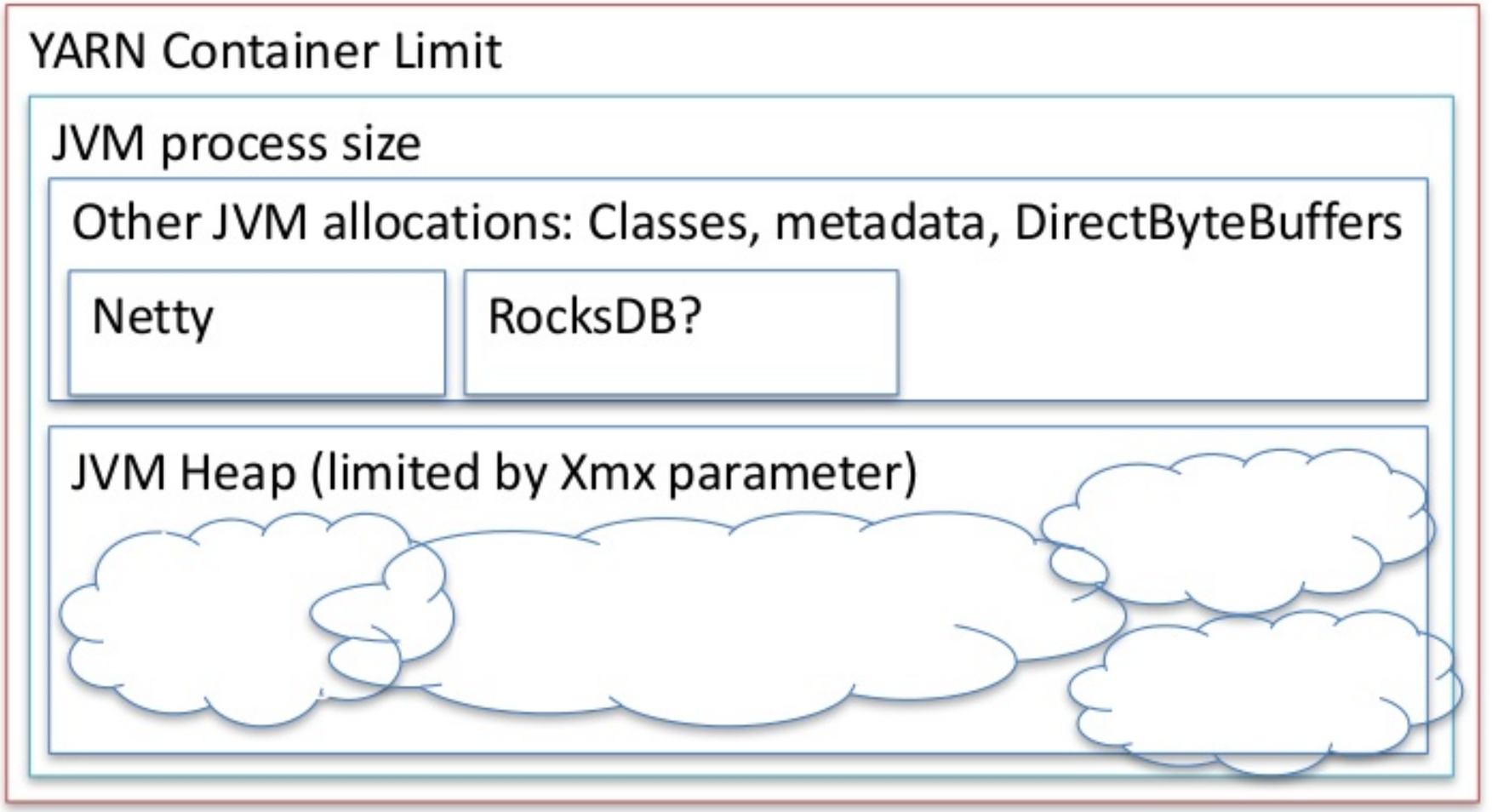
- A Flink cluster needs exactly as many task slots as the highest parallelism used in the job
- Better resource utilization
- Reduced network traffic

What can you do?

- Number of TaskManagers vs number of slots per TM
- Set slots per TaskManager
- Set parallelism per operator
- Control operator chaining behavior
- Set slot sharing groups to break operators into different slots

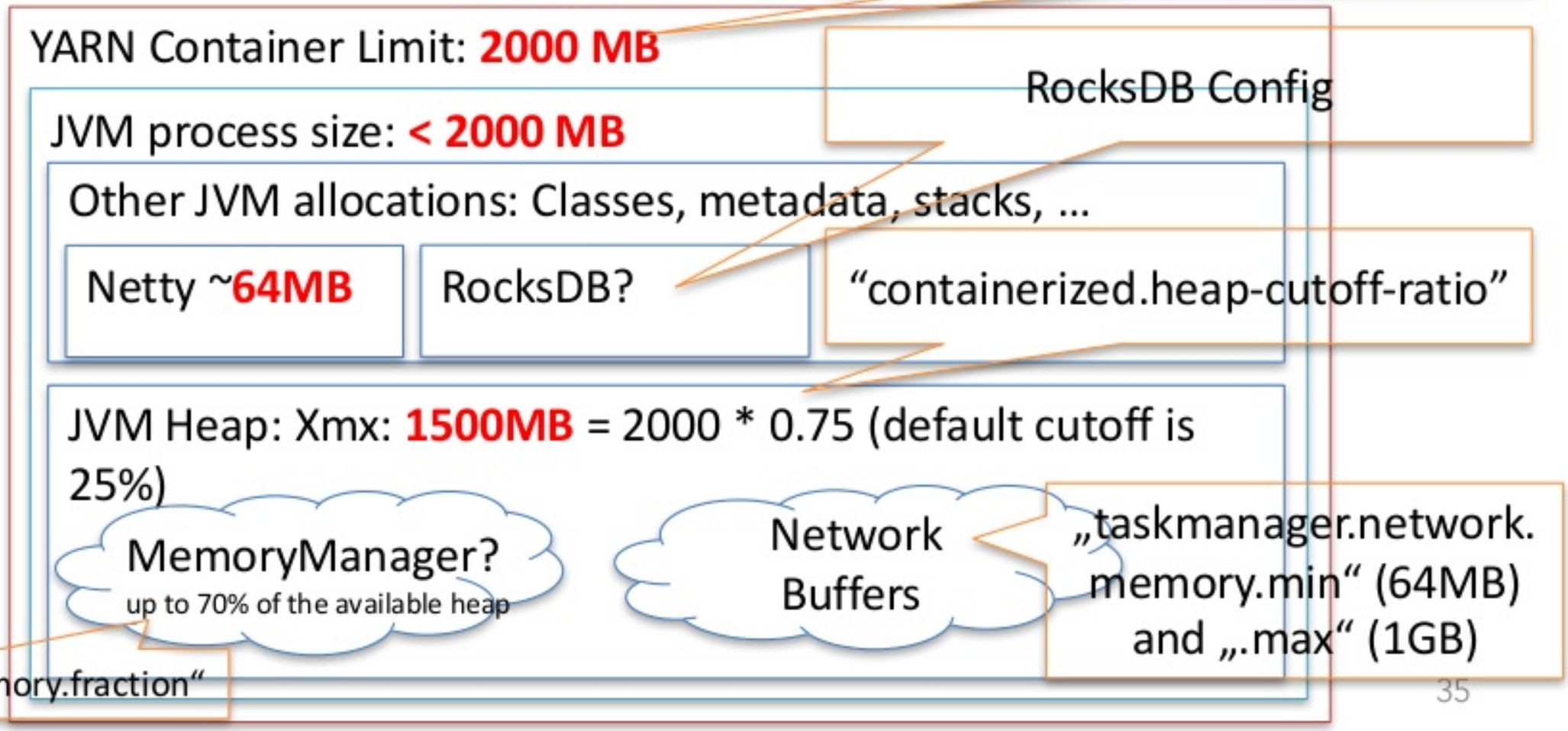
Tuning: Memory Configuration

Memory in Flink (on YARN)



Example: Memory in Flink

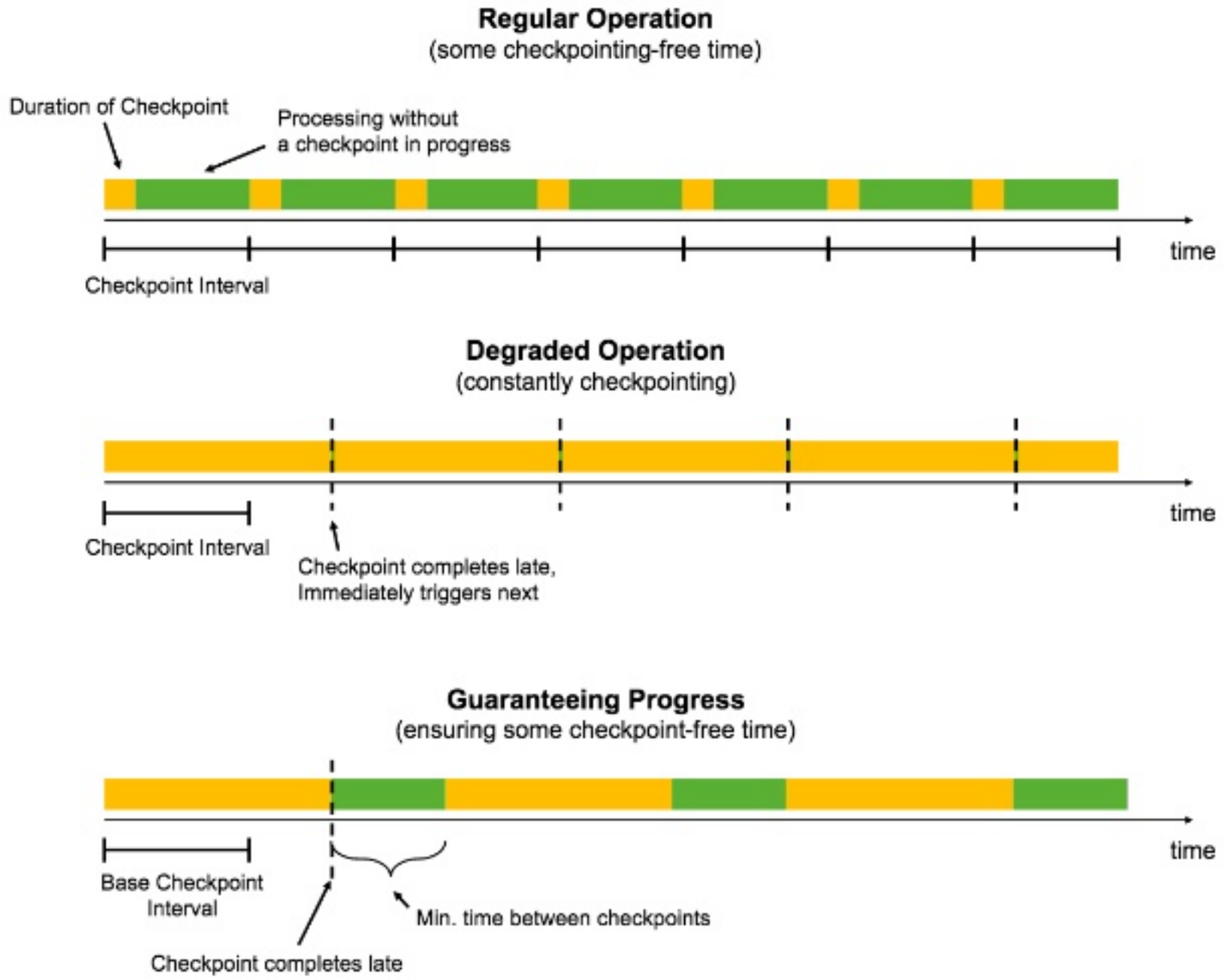
- TaskManager: 2000 MB on YARN



Tuning: Checkpointing

Checkpointing

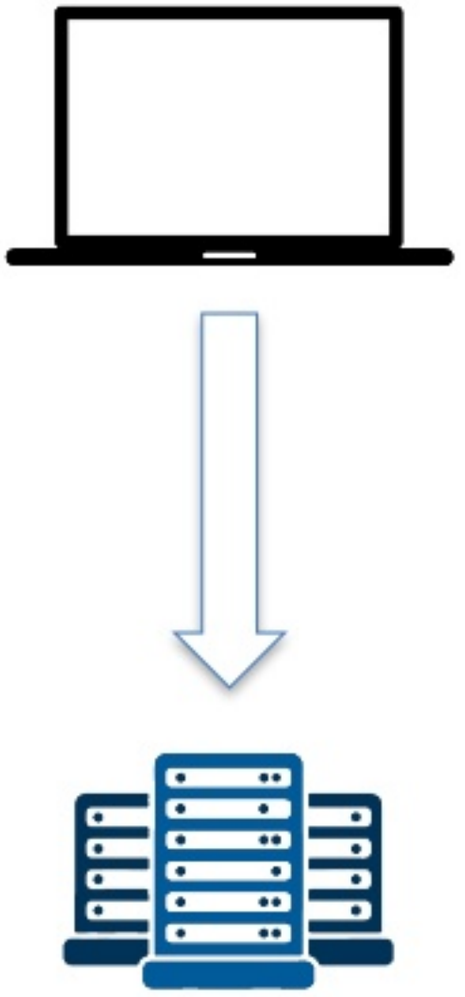
- Measure, analyze and try out!
- Configure a checkpointing interval
  - How much can you afford to reprocess on restore?
  - How many resources are consumed by the checkpointing? (cost in throughput and latency)
- Fine-tuning
  - "min pause between checkpoints"
  - "checkpoint timeout"
  - "concurrent checkpoints"
- Configure exactly once / at least once
  - exactly once does buffer alignment spilling (can affect latency)


Regular Operation
(some checkpointing-free time)
Duration of Checkpoint
Processing without
a checkpoint in progress
time
Checkpoint Interval
Degraded Operation
(constantly checkpointing)
time
Checkpoint Interval
Checkpoint completes late,
Immediately triggers next
Guaranteeing Progress
(ensuring some checkpoint-free time)
time
Base Checkpoint
Interval
Min. time between checkpoints
Checkpoint completes late

Conclusion

Tuning Approaches

- 1. Develop / optimize job locally
  - Use data generator / small sample dataset
  - Check the logs for warnings
  - Check the UI for backpressure, throughput, metrics
  - Debug / profile locally
- 2. Optimize on cluster
  - Checkpointing, parallelism, slots, RocksDB, network config, …

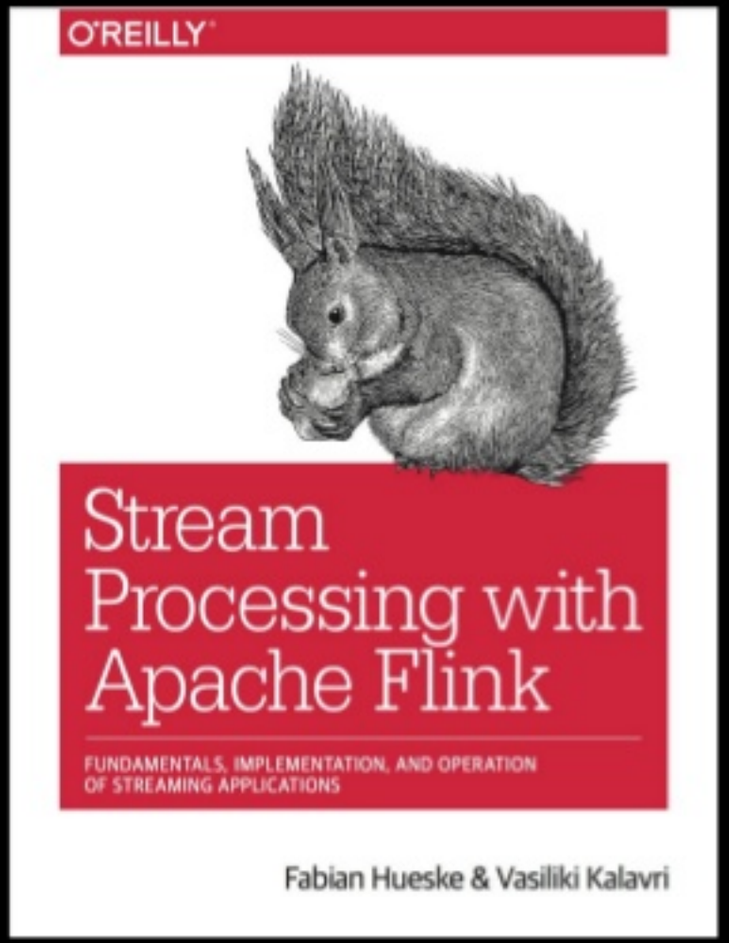


The usual suspects

- Inefficient serialization
- Inefficient dataflow graph
  - Too many repartitionings; blocking I/O
- Slow external systems
- Slow network, slow disks
- Checkpointing configuration

Let’s discuss …

Q & A



Thank you!

@rmetzger | @theplucas
@dataArtisans

We are hiring!

data-artisans.com/careers

Set UUIDs for all (stateful) operators

- Operator UUIDs are needed to restore state from a savepoint
- Flink will auto–generate UUIDs, but this results in fragile snapshots.
- Setting UUIDs in the API:

```
DataStream<String> stream = env
  .addSource(new StatefulSource())
  .uid("source-id") // ID for the source operator
  .map(new StatefulMapper())
  .uid("mapper-id") // ID for the mapper
  .print();
```

Use the savepoint tool for deletions

- Savepoint files contain only metadata and depend on the checkpoint files
  - `bin/flink savepoint -d :savepointPath`
- There is work in progress to make savepoints self-contained
  → deletion / relocation will be much easier

Avoid the deprecated state APIs

- Using the *Checkpointed* interface will prevent you from rescaling your job
- Use *ListCheckpointed (*like *Checkpointed,* but redistributeble*)* or *CheckpointedFunction* (full flexibility) instead.

Production Readiness Checklist:
https://ci.apache.org/projects/flink/flink-docs-release-1.3/ops/production_ready.html

Explicitly set max parallelism

- Changing this parameter is painful
  - requires a complete restart and loss of all checkpointed/savepointed state
- 0 < parallelism <= max parallelism <= 32768
- Max parallelism > 128 has some impact on performance and state size

RocksDB

- If you have plenty of memory, be generous with RocksDB (note: RocksDB does not allocate its memory from the JVM's heap!). When allocating more memory for RocksDB on YARN, increase the memory cutoff (= smaller heap)

- RocksDB has many tuning parameters.

- Flink offers predefined collections of options:
  - `SPINNING_DISK_OPTIMIZED_HIGH_MEM`
  - `FLASH_SSD_OPTIMIZED`

Tuning: Serialization

(de)serialization is expensive

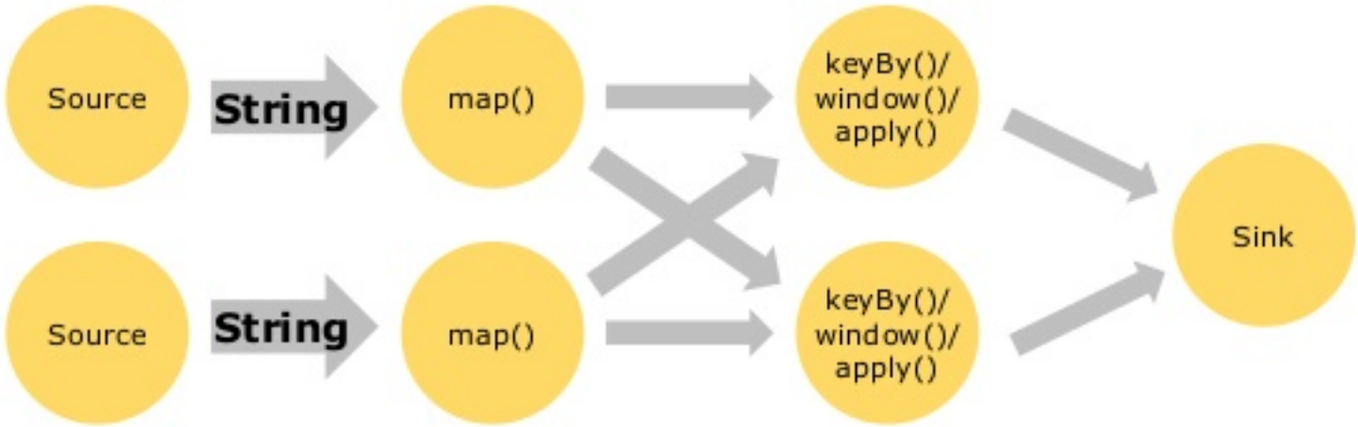
- Getting this wrong can have a huge impact
- But don’t overthink it

Serialization in Flink

- Flink has its own serialization framework, which is used for
  - Basic types (Java primitives and their boxed form)
  - Primitive arrays and Object arrays
  - Tuples
  - Scala case classes
  - POJOs
- Otherwise Flink falls back to Kryo

A note on custom serializers / parsers

- Avoid obvious anti-patterns, e.g. creating a new JSON parser for every record



- Many sources (e.g. Kafka) can parse JSON directly
- Avoid, if possible, to ship the schema with every record

What else?

- You should register types with Kryo, e.g.,
  - env.registerTypeWithKryoSerializer(DateTime.class, JodaDateTimeSerializer.class)
- You should register any subtypes; this can increase performance a lot
- You can use serializers from other systems, like Protobuf or Thrift with Kyro by registering the types (and serializers)
- Avoid expensive types, e.g. Collections, large records
- Do not change serializers or type registrations if you are restoring from a savepoint

(High Availability) Deployment

Deployment Options

- Hadoop YARN integration
  - Cloudera, Hortonworks, MapR, …
  - Amazon Elastic MapReduce (EMR), Google Cloud dataproc

- Mesos & DC/OS integration

- Standalone Cluster (“native”)
  - provided bash scripts
  - provided Docker images

Flink in containerland
DAY 3 / 3:20 PM - 4:00 PM
MASCHINENHAUS

Deployment Options

- Docs and best-practices coming soon for
  - Kubernetes
  - Docker Swarm

→ Check Flink Documentation details!

Flink in containerland
DAY 3 / 3:20 PM - 4:00 PM
MASCHINENHAUS

High Availability Deployments

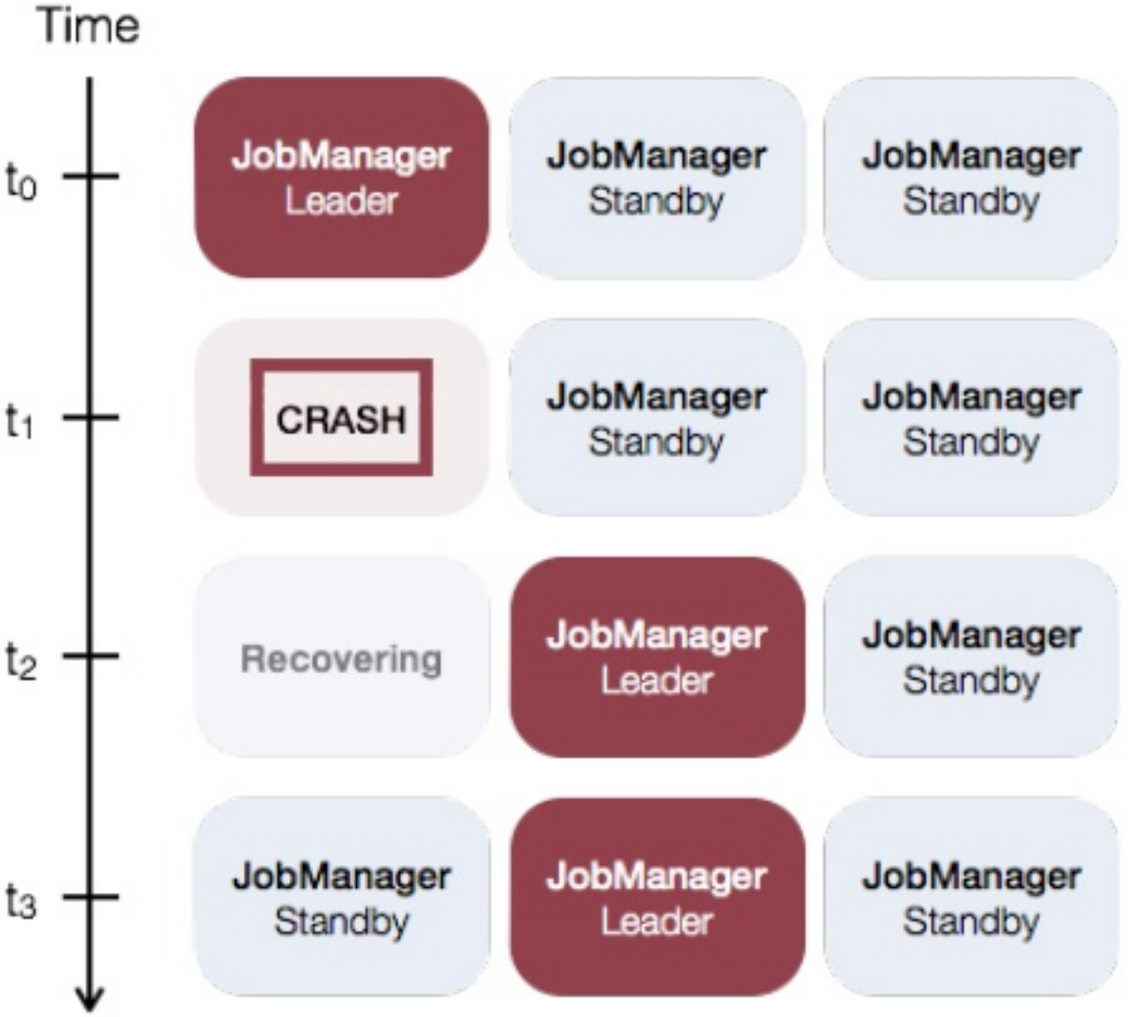
YARN / Mesos HA

- Run only one JobManager
- Restarts managed by the cluster framework
- For HA on YARN, we recommend using at least Hadoop 2.5.0 (due to a critical bug in 2.4)
- Zookeeper is always required

Standalone cluster HA

- Run standby JobManagers
- Zookeeper manages JobManager failover and restarts
- TaskManager failures are resolved by the JobManager

→ Use custom tool to ensure a certain number of Job- and TaskManagers



Bonus Slides

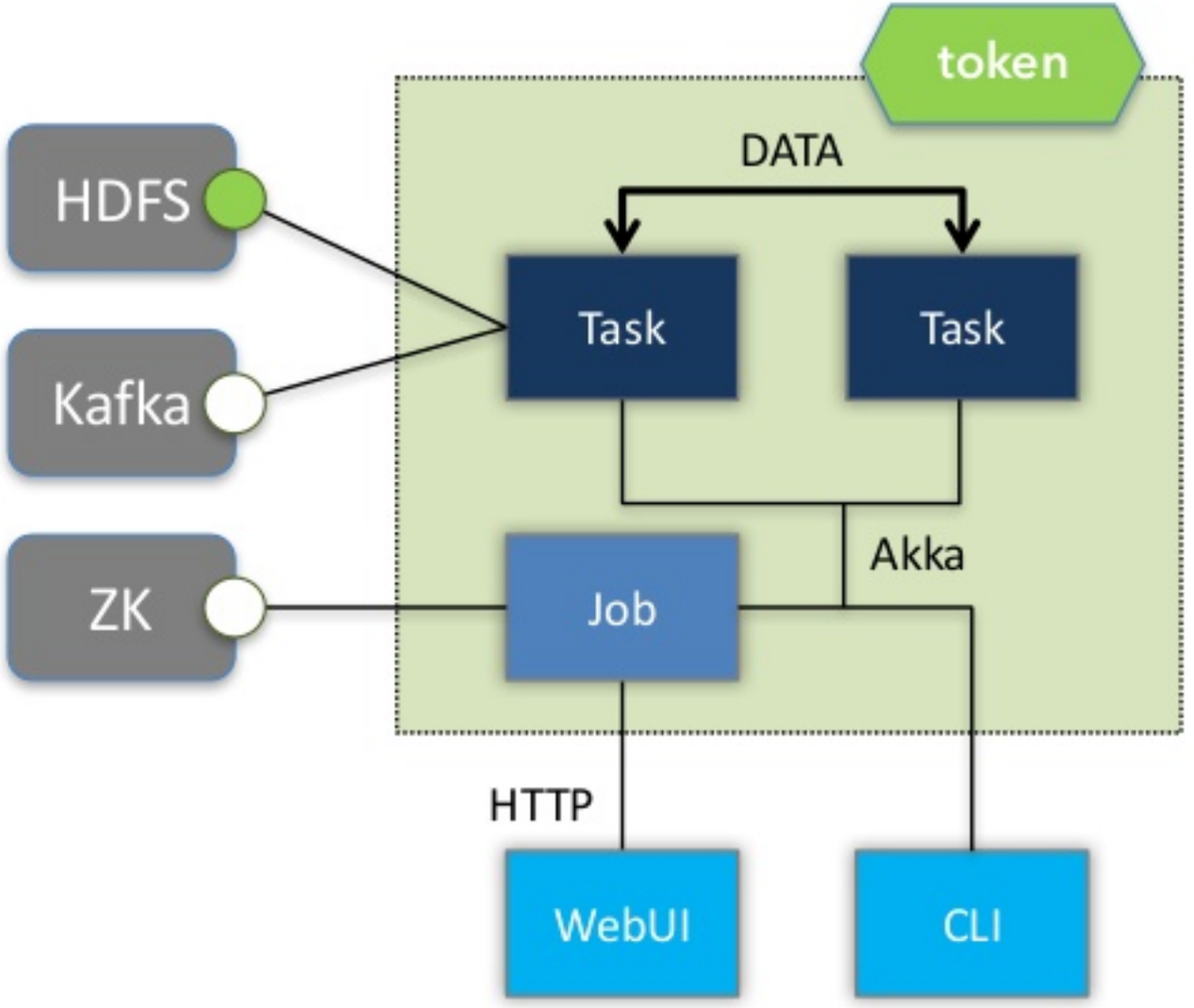
Deployment: Security

Outline

1. Hadoop delegation tokens
2. Kerberos authentication
3. SSL

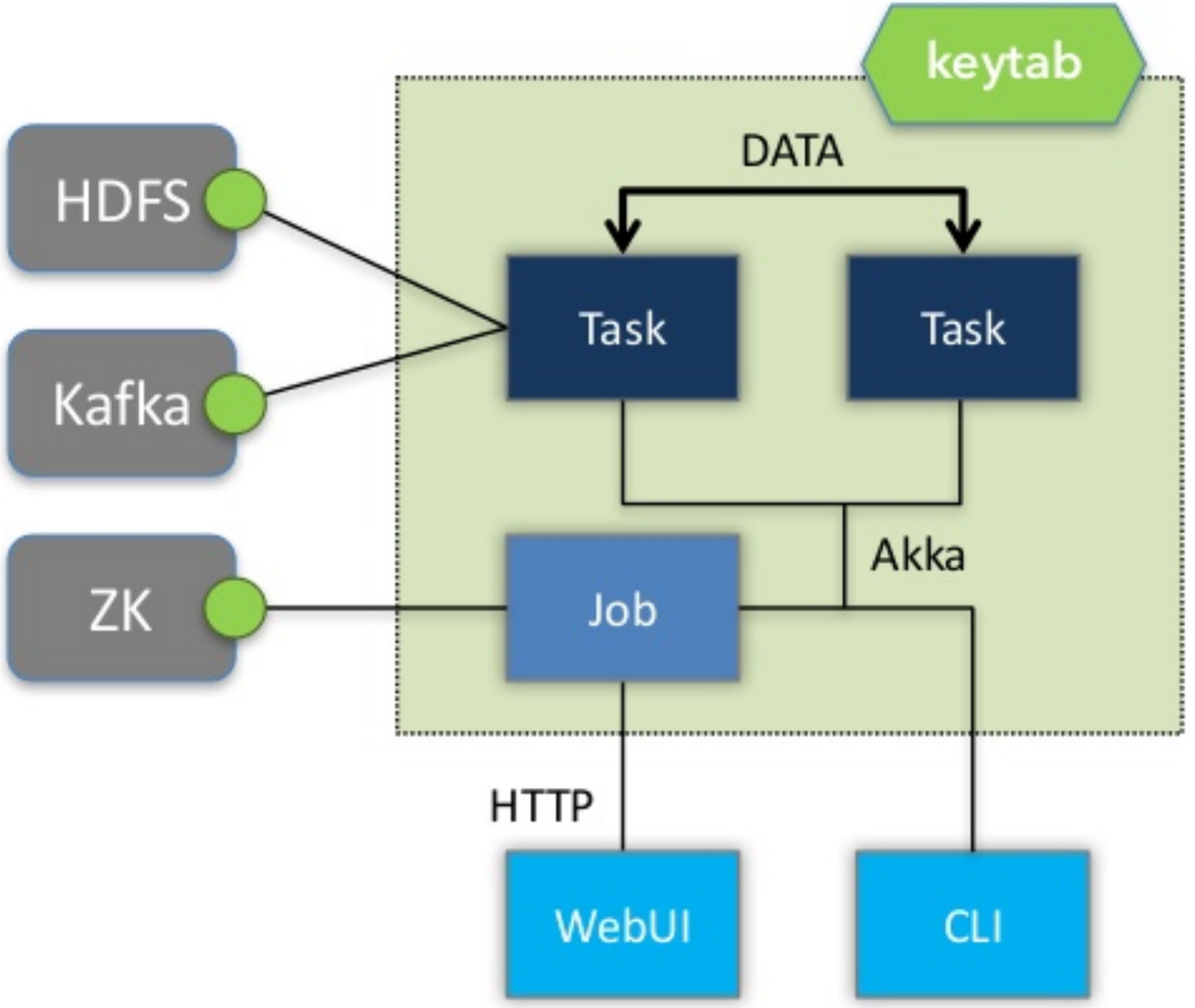
Hadoop delegation tokens

- Quite limited
  - YARN only
  - Hadoop services only
  - Tokens expire



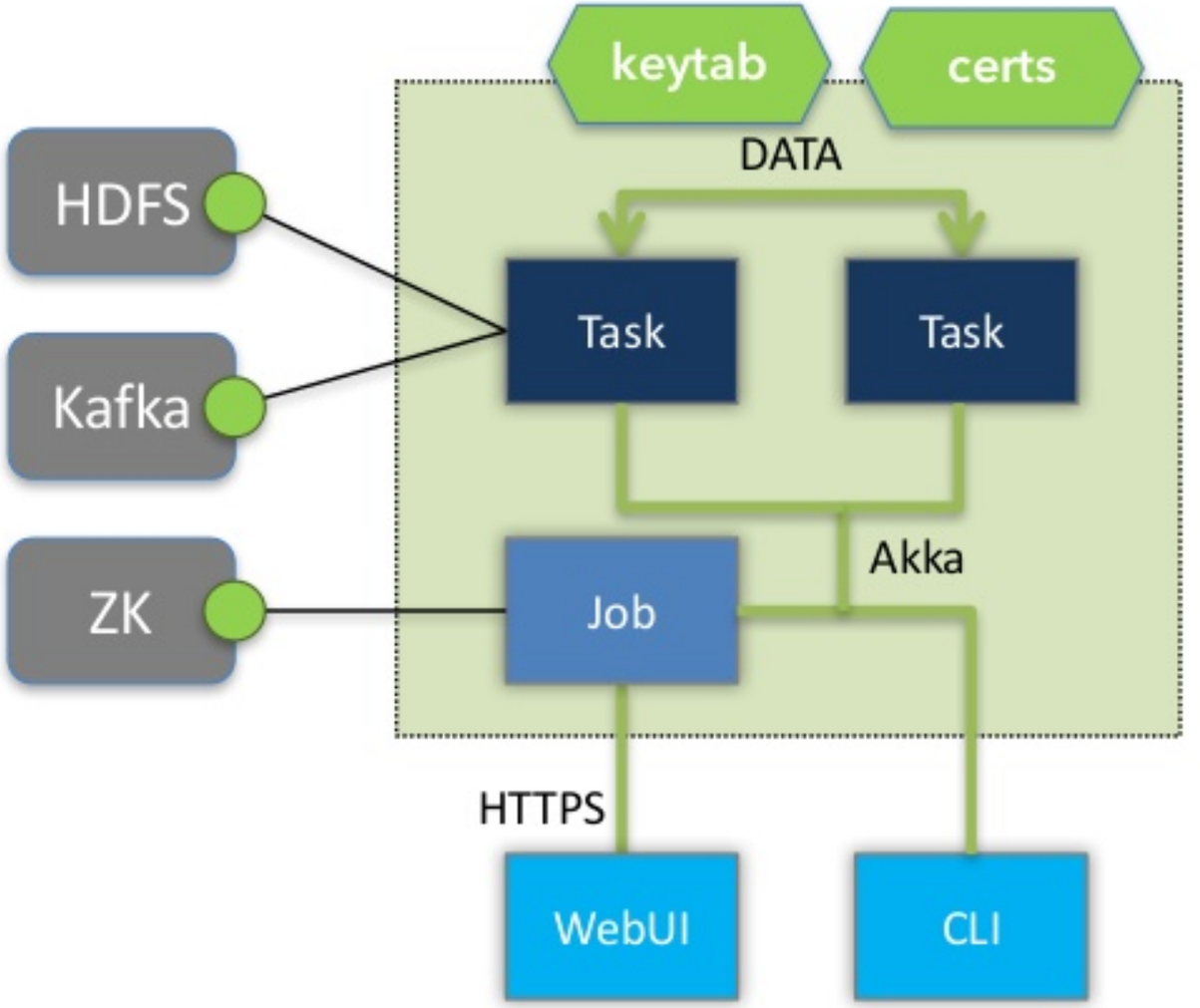
Kerberos authentication

- Keytab-based identity
- Standalone, YARN, Mesos
- Shared by all jobs



SSL

- **taskmanager.data.ssl.enabled**: communication between task managers
- **blob.service.ssl.enabled**: client/server blob service
- **akka.ssl.enabled**: akka-based control connection between the flink client, jobmanager and taskmanager
- **jobmanager.web.ssl.enabled**: https for WebUI



Limitations

- The clients are not authenticated to the cluster
- All the secrets known to a Flink job are exposed to everyone who can connect to the cluster's endpoint
- Exploring SSL mutual authentication