State & State Management

Apache Flink® Tuning & Troubleshooting Training

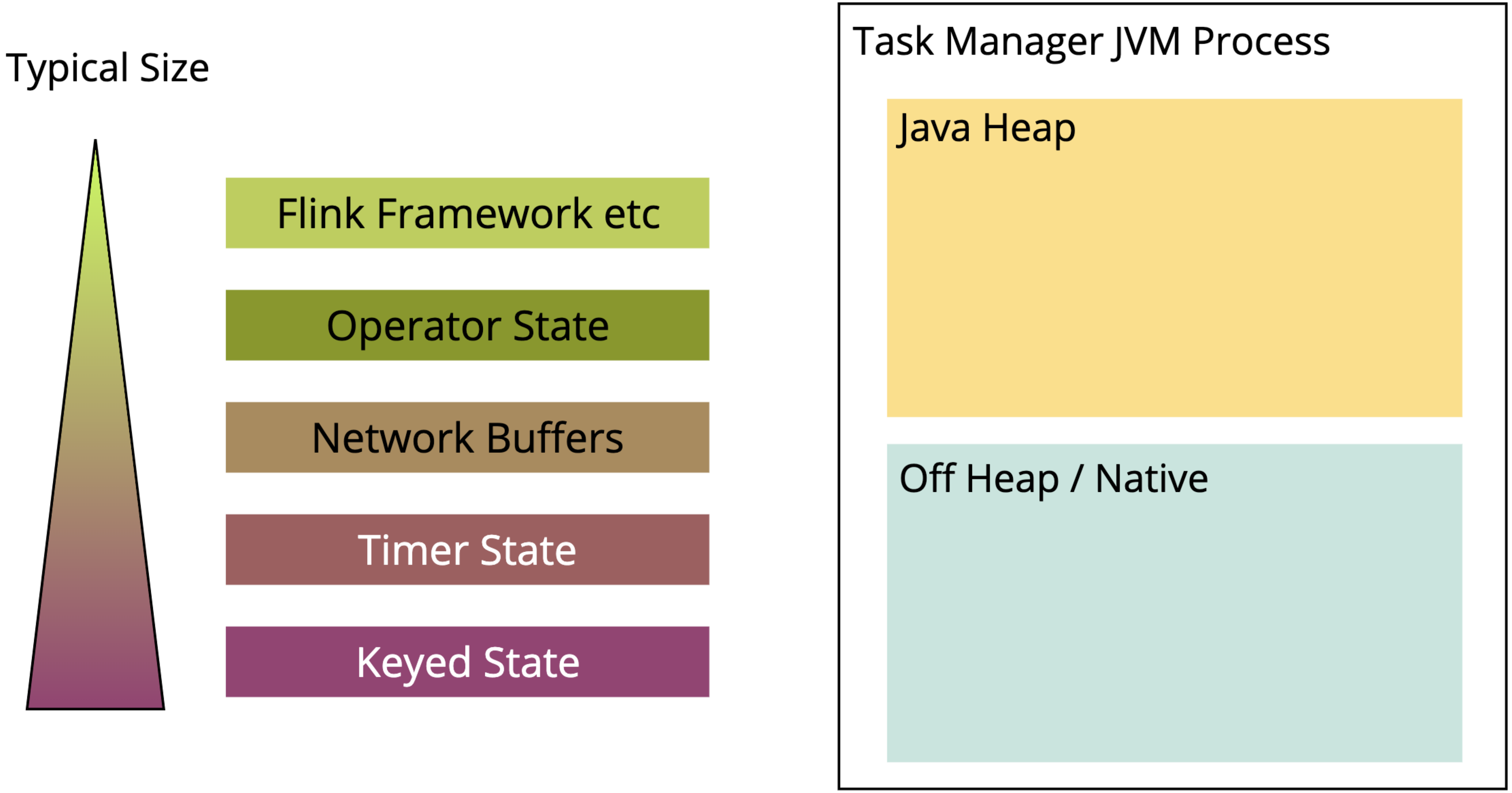
ververica

As you scale up

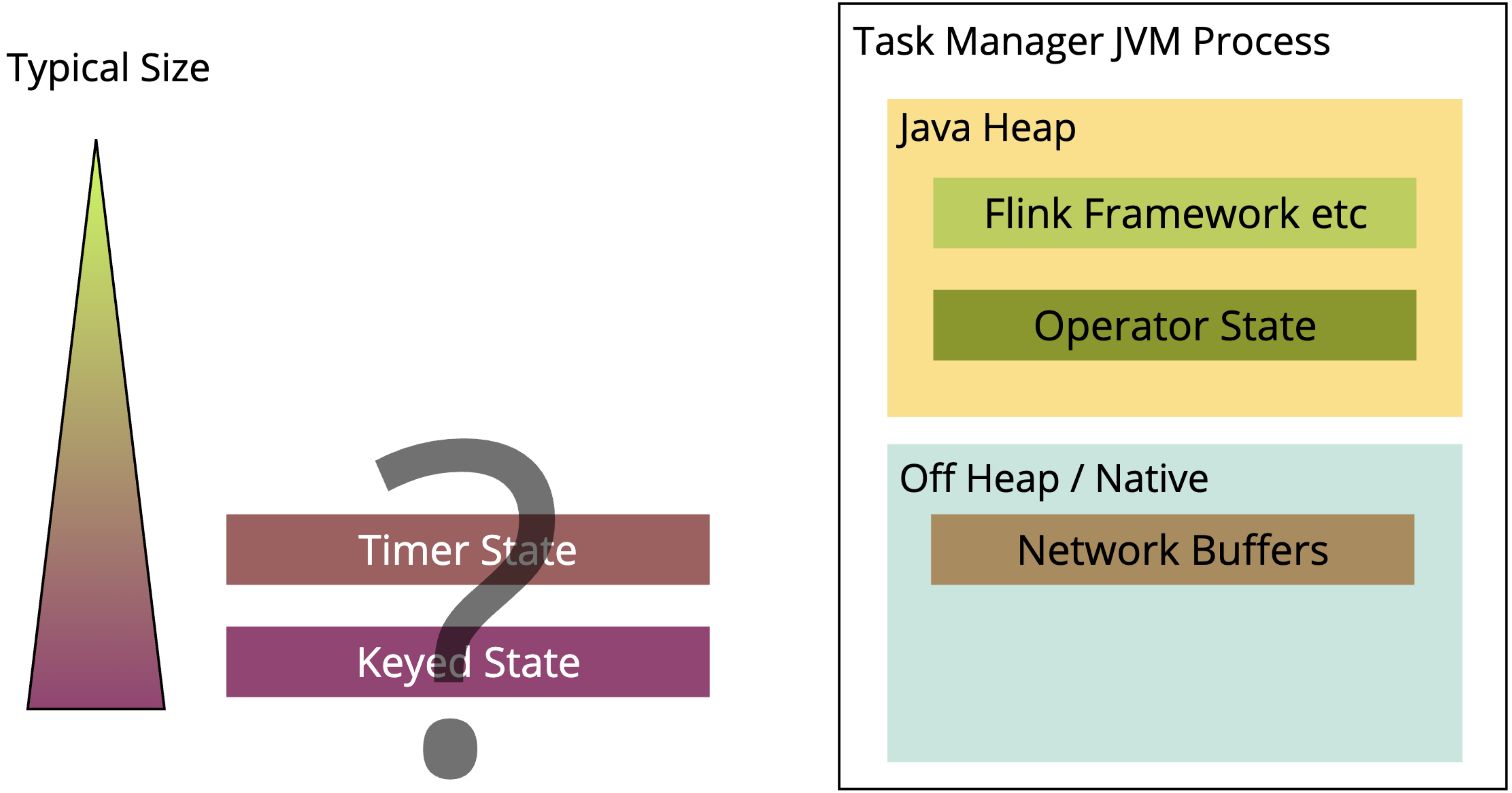
- You are more likely to need RocksDB, especially for incremental checkpointing
- Cleaning up stale state becomes more important
- Checkpointing becomes more likely to fail/timeout
- You are more likely to hit rate limits enforced by external services
- It becomes more worthwhile to be smart about recovery

State Backends

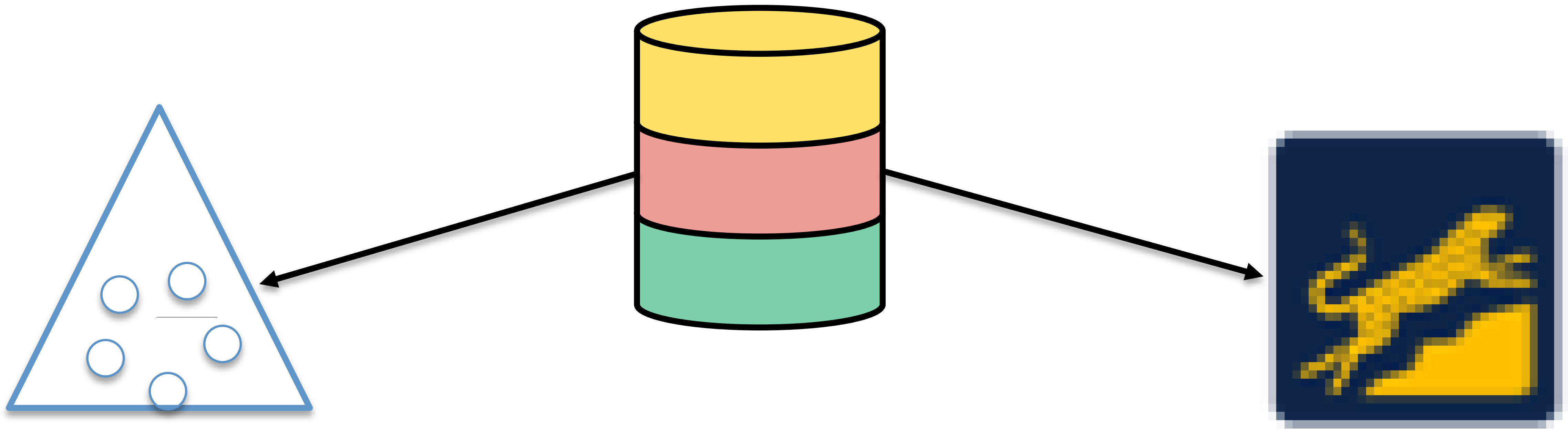
Task Manager memory layout



Task Manager memory layout



State backends



Based on Java Heap Objects

Based on RocksDB

Heap-based State Backend

Recap: Heap-based state backend

- State lives as Java objects on the heap
- Organized as chained hash table, key ↦ state
- One hash table per registered state descriptor
- Supports asynchronous state snapshots
- Data is de / serialized only during state snapshot and restore

- Highest throughput

Performance considerations

- Choose TypeSerializers with efficient copy-methods
- Flag immutability of objects where possible to avoid needing copies
- GC choice / tuning
- Scale out using multiple task managers per node

RocksDB

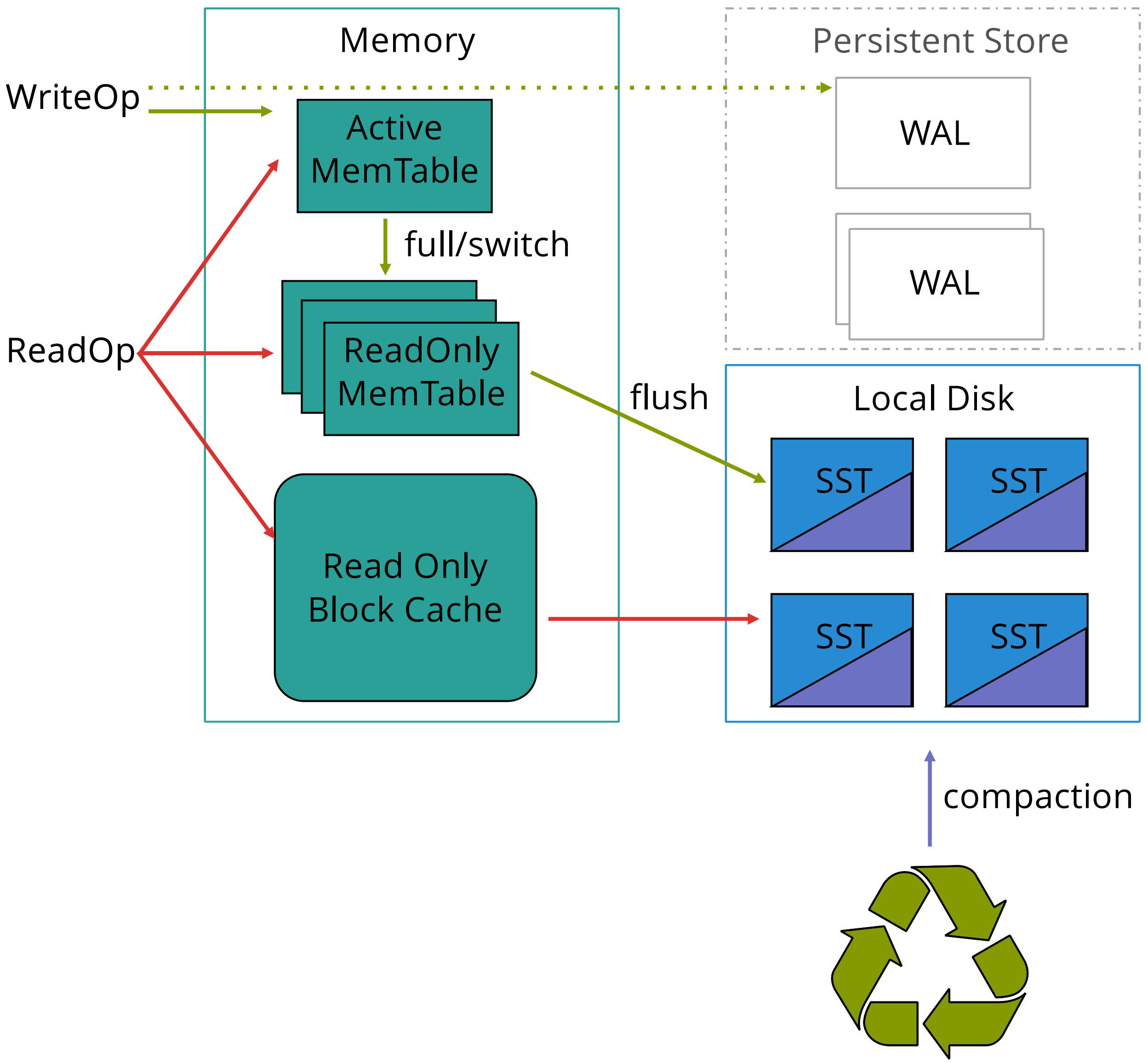
RocksDB keyed state backend

- State lives as serialized byte-strings in off-heap memory and on local disk
- Key / value store, organized as a log-structured merge tree (LSM tree)
- Key: serialized bytes of `<keygroup, key, namespace>`
- LSM naturally supports MVCC (multi-version concurrency control)
- Data is de / serialized on every read and update

High level architecture

- Keys and values: arbitrary byte streams, limited to at most 2^{31} bytes
- Organization
  - memtable: new writes are inserted into this in-memory structure
  - logfile: new writes are optionally also written to this sequentially written WAL file
    - not used by Flink
  - SST file: full memtables are flushed to an SST file (sorted by key for fast lookups)
    - SST = *static sorted table*
    - immmutable: once written, never modified

In Flink:
- WAL disabled
- persistence via checkpoints



RocksDB uses log-structured merge-trees

- Observations
  - Sequential disk access is faster than randomly accessing RAM
  - Sequential disk access is *much*, *much* faster than random disk access
  - Simple, append-only logs take advantage of this, but can't provide efficient key-based access
- SST files
  - Each contains a small, chronological, sorted subset of changes
  - Since they are immutable, duplicate entries are created as records are updated or removed
  - Reads start in the memtables, then the sstfiles are checked, in order from newest to oldest
  - sstfiles are compacted and merged
- Optimizations
  - Indexes and Bloom filters can be used to reduce the effort to locate keys within sstfiles

RocksDB resource consumption

- One RocksDB instance per operator subtask
- One column family per registered state
- SST files, MemTable’s, block caches, compaction threads per column family

- `state.backend.rocksdb.block.cache-size`
  - amount of cache for data blocks (8MB)
- `state.backend.rocksdb.writebuffer.size`
  - max size of a MemTable (4MB)
- `state.backend.rocksdb.writebuffer.count`
  - maximum number of MemTable’s allowed in memory before flushing to SST file (2)

RocksDB resource consumption(2)

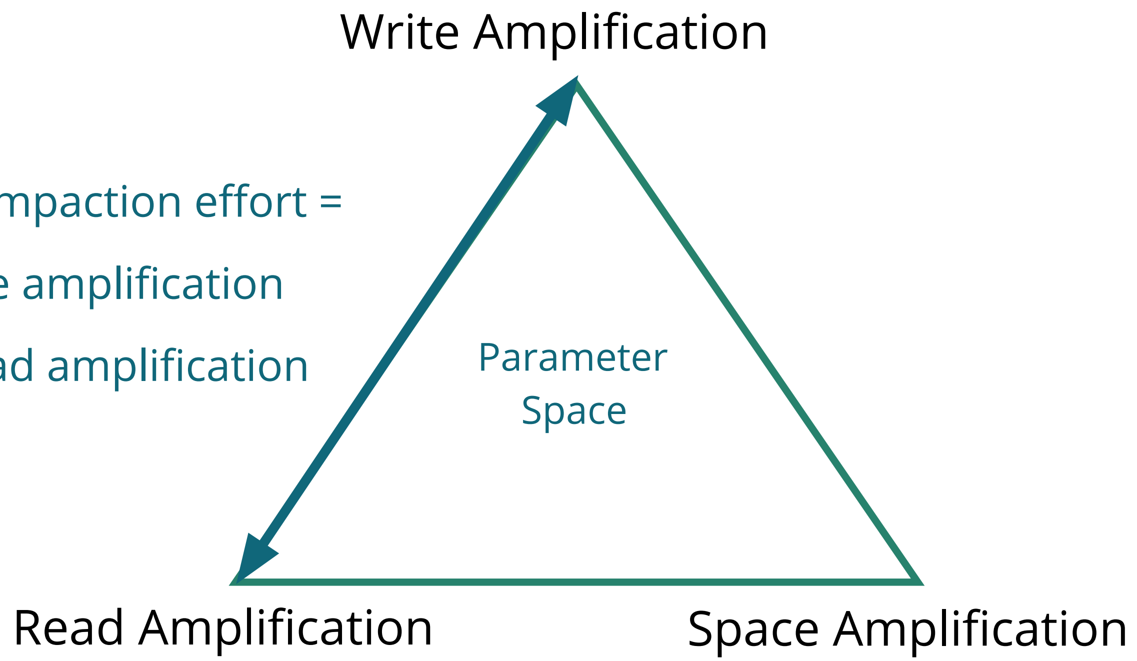
- One RocksDB instance per operator subtask
- One column family per registered state
- SST files, MemTable’s, block caches, compaction threads per column family

- Indexes and Bloom filters
  - optional, via `ConfigurableOptionsFactory`
  - see `PredefinedOptions#SPINNING_DISK_OPTIMIZED_HIGH_MEM` for an example
- Table Cache
  - caches open file descriptors to SST files
  - default: unlimited
- Shared compaction threads, via `ConfigurableOptionsFactory`
  - `DBOptions#setEnv(Env.getDefault())` - see FLINK-10198

RocksDB performance tuning

- Write amplification: bytes written to storage / bytes written to the DB
- Read amplification: disk reads / query
- Space amplification: size of database files / data size

Example: More compaction effort = increased write amplification and reduced read amplification



More details: https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

Predefined options

- These profiles are predefined collections of RocksDB settings
- Select one of these profiles via `state.backend.rocksdb.predefined-options`
  - `DEFAULT`
  - `SPINNING_DISK_OPTIMIZED`
  - `SPINNING_DISK_OPTIMIZED_HIGH_MEM`
  - `FLASH_SSD_OPTIMIZED`
- Look at the `PredefinedOptions` class to see what these do
- See the docs if you want to do your own fine-grained tuning

RocksDB metrics

- Some RocksDB native metrics can be forwarded to Flink's metric reporter(s)
- Can provide valuable insight for tuning
- Enabling native metrics can degrade performance, so proceed with caution
- See the docs for details

Timers and RocksDB

- By default, timers are stored on the heap
- If you don't have many timers, leave them on the heap
- For timers in RocksDB:
  - `state.backend.rocksdb.timer-service.factory: rocksdb`
- Note: If you use RocksDB for keyed state and have timers on the heap, the timers will be snapshotted during the synchronous portion of the snapshot.
  - see FLINK-10026

Incremental checkpointing

- Only the RocksDB state backend offers incremental checkpointing
- Flink observes the SST files created/deleted since the last checkpoint
- These changes are (carefully) mirrored into Flink's checkpoint store

- Expected trade-off: faster* checkpoints, slower recovery
- Creates write amplification (upload compacted SST files to eventually prune checkpoint history)
  - Sum of all increments can be larger than the full state size
- No rebuild required: simply re-open the RocksDB backend from the SST files
- SST files are snappy-compressed by default

General Performance Considerations

- Use efficient `TypeSerializer`'s and serialization formats
- Decompose user code objects
  - `ValueState<List<Integer>>` ⟶ `ListState<Integer>`
  - `ValueState<Map<Integer, Integer>>` ⟶ `MapState<Integer, Integer>`
- Flatten POJOs / avoid deep objects
  - Reduces object overheads and avoids following references
- Use the correct configuration for your hardware setup
- Consider enabling RocksDB native metrics to profile your applications
- File Systems
  - Working directory on fast storage, ideally local SSD. Could even be memory.
  - EBS performance can be problematic. Shares network connection.

Expiring State

Be careful about indefinite state retention

It's easy to either implicitly or explicitly require Flink maintain unbounded state, e.g.:

```
DataStream<Tuple2<Long, Float>> maximums = input
      .keyBy(0)
      .maxBy(1);
```

To clean up expired state:

- Use a `ProcessFunction` to hold the state, and register a `Timer`
- Use State Time-To-Live (next slides!)
- If using Flink SQL, configure an idle state retention time

State TTL: cleanup

- Expired values are removed when they are read, e.g., by calling `ValueState.value()`
- You can avoid restoring expired state from full state snapshots via

```
StateTtlConfig ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .cleanupFullSnapshot()
    .build();
```

State TTL: generic cleanup in the background

```java
StateTtlConfig ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .cleanupInBackground()
    .build();
```

This will activate the default background cleanup (if any exists) for the current state backend.

State TTL: incremental cleanup

Heap-based state backend only

```
StateTtlConfig ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .cleanupIncrementally(10, true)
    .build();
```

- `10`: the number of state entries to check per state access (default: 5)
- `true`: whether to also trigger cleanup on a per-record basis (default: false)

- Cleanup depends on records being accessed or processed
- Incremental cleanup increases processing latency

State TTL: clean-up during RocksDB compaction

```
StateTtlConfig ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .cleanupInRocksdbCompactFilter(1000)
    .build();
```

- `1000`: RocksDB will query Flink for the current timestamp after checking each batch of 1000 entries
  - Smaller values will lead to more timely expirations and slower compactions
- This feature requires enabling RocksDB compaction filtering
  - `state.backend.rocksdb.ttl.compaction.filter.enabled`
- This feature slows down RocksDB compaction

State TTL: caveats

- Using state TTL increases the consumption of state storage
- Specifying TTL in event time is not supported
- Adding or removing a StateTtlConfig to an existing state descriptor is not supported
  - will cause a `StateMigrationException`

Working with Checkpoints

(Especially at Large Scale)

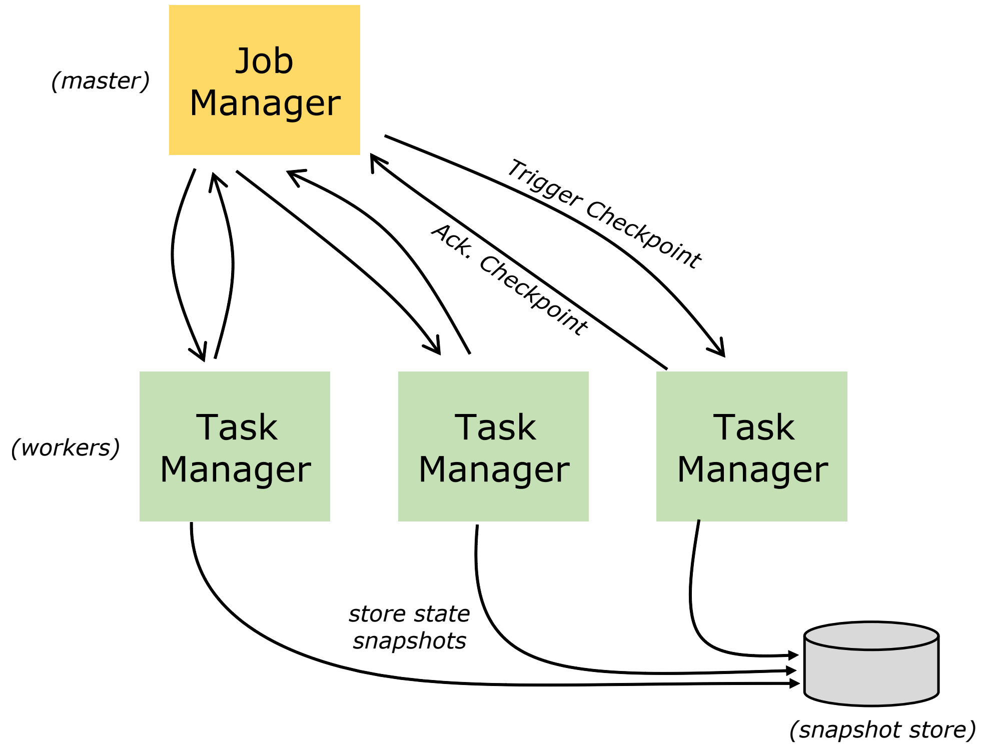
Snapshots, Checkpoints, and Savepoints

- Snapshot is the generic term; checkpoints and savepoints are snapshots
- Broadly speaking:
  - Checkpoints are managed by Flink, intended for
    - failure recovery
  - Savepoints are user-triggered and are retained indefinitely, intended for
    - rescaling
    - upgrades

Current differences:

	user-controlled	incremental	rescaling	unified format
checkpoints	Maybe	Yes	Maybe	No
savepoints	Yes	No	Yes	Yes

Checkpoints require global coordination



- At scale, the JM eventually becomes a bottleneck
- May need to increase checkpoint timeout and/or increase slots/TM and reduce # of TMs

Checkpoint configuration, basics

- Exactly-once vs at-least-once (i.e., barrier alignment on/off)
- Incremental vs full (incremental only w/ RocksDB)
- Checkpoint interval
  - Has a big impact on recovery time
  - Also has a big impact on latency for committing transactions

Checkpoint configuration, continued

- Number of retained checkpoints (default: 1)
- `env.getCheckpointConfig().setCheckpointTimeout(n)` (default: 10 minutes)
- Compression (default: off)
  - RocksDB SST files are already compressed (used in incremental snapshots)
- Fail/continue on checkpoint errors (default: fail)
- Number of concurrent checkpoints (default 1)
- Minimum pause between checkpoints

```
StreamExecutionEnvironment.getCheckpointConfig().setMinPauseBetweenCheckpoints(milliseconds)
```

External (retained) checkpoints

```
CheckpointConfig config = env.getCheckpointConfig();
config.enableExternalizedCheckpoints(ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
```

- By default, checkpoints are deleted when a program is cancelled.
- Example above will retain them instead
  - Cheap / incremental "savepoint" but in internal state backend format
- Location is specified in
  - config — `state.checkpoints.dir: hdfs:///checkpoints/`
  - code — `env.setStateBackend(new RocksDBStateBackend("hdfs:///checkpoints-data/"))`
- You can resume from a retained checkpoint (just as with a savepoint)

```
$ bin/flink run -s :checkpointMetaDataPath [:runArgs]
```

Why checkpoints might fail (timeout)

- Timeout is simply too short
  - very large cluster
  - very large state (non-incremental checkpoints)
  - high rate of state change (incremental checkpoints)
  - checkpoint storage and/or network are slow or unavailable
- Backpressure is preventing checkpoint barriers from advancing fast enough
  - many possible causes

Scenario 1

- Computing an event time join on multiple sources
- One source is lagging far behind the other(s)
- Leading to excessive buffering in Flink state, and large checkpoints

Scenario 2

- Many timers firing simultaneously
- Operator isn't reading from its inputs while triggering onTimer calls
- Checkpoint barriers delayed

Scenario 3

- An operator makes synchronous calls to an external service
  - e.g., a flatMap that calls a REST API
- Everything is fine under normal conditions
- But when the external service is under heavy load, it responds more slowly
- Leading to intermittent backpressure and checkpoint failures

Checkpoint Counts	Triggered: 71	In Progress: 1	Completed: 70	Failed: 0	Restored: 0

Latest Completed Checkpoint	ID: 70	Completion Time: 16:59:12	End to End Duration: 11ms	State Size: 9.55 KB

Checkpoint Detail: **Path:** <checkpoint-not-externally-addressable> | **Discarded:** true

Operators:

	Name	Acknowledged	Latest Acknowledgment	End to End Duration	State Size	Buffered During Alignment
+	Source: test data	1/1 (100%)	16:59:12	8ms	261 B	0 B
+	sawTooth	4/4 (100%)	16:59:12	9ms	1.02 KB	0 B
+	assignKey(temp)	4/4 (100%)	16:59:12	9ms	0 B	0 B
+	sineWave	4/4 (100%)	16:59:12	9ms	0 B	0 B
+	assignKey(pressure	4/4 (100%)	16:59:12	10ms	0 B	0 B
+	squareWave	4/4 (100%)	16:59:12	9ms	0 B	0 B
+	assignKey(door)	4/4 (100%)	16:59:12	10ms	0 B	0 B
+	Sink: sensors-sink	4/4 (100%)	16:59:12	10ms	0 B	0 B
	window	4/4 (100%)	16:59:12	11ms	8.28 KB	0 B

-	window	4/4 (100%)	16:59:12	11ms	8.28 KB	0 B

SubTasks:

	End to End Duration	State Size	Checkpoint Duration (Sync)	Checkpoint Duration (Async)	Alignment Buffered	Alignment Duration
Minimum	10ms	2.01 KB	0ms	0ms	0 B	2ms
Average	10ms	2.07 KB	0ms	0ms	0 B	3ms
Maximum	11ms	2.17 KB	0ms	0ms	0 B	4ms

ID	Acknowledgement Time	E2E Duration	State Size	Checkpoint Duration (Sync)	Checkpoint Duration (Async)	Align Buffered	Align Duration
1	16:59:12	10ms	2.17 KB	0ms	0ms	0 B	2ms
2	16:59:12	10ms	2.01 KB	0ms	0ms	0 B	3ms
3	16:59:12	11ms	2.01 KB	0ms	0ms	0 B	4ms
4	16:59:12	10ms	2.09 KB	0ms	0ms	0 B	3ms

Latest Failed Checkpoint **ID:** 9 | **Failure Time:** 15:36:08 | **Cause:** The job has failed.

Checkpoint Detail: **Path:** - | **Discarded:** - | **Failure Message:** The job has failed.

Operators:

	Name	Acknowledged	Latest Acknowledgment	End to End Duration	State Size	Buffered During Alignment
+	Source: test data	1/1 (100%)	15:36:08	3ms	261 B	0 B
+	sawTooth	1/1 (100%)	15:36:08	12ms	260 B	0 B
+	assignKey(temp)	1/1 (100%)	15:36:08	12ms	0 B	0 B
+	sineWave	1/1 (100%)	15:36:08	11ms	0 B	0 B
+	assignKey(pressure	1/1 (100%)	15:36:08	13ms	0 B	0 B
+	squareWave	1/1 (100%)	15:36:08	11ms	0 B	0 B
+	assignKey(door)	1/1 (100%)	15:36:08	15ms	0 B	0 B
+	Sink: sensors-sink	0/1 (0%)	n/a	n/a	0 B	0 B
+	window	1/1 (100%)	15:36:08	16ms	2.37 KB	0 B
+	Sink: summed-sensors-sink	0/1 (0%)	n/a	n/a	0 B	0 B

Diagnosing checkpoint timeouts

Look for clues in the logs and/or web UI:

- end_to_end_duration – synchronous_duration – asynchronous_duration
  - this is how long it look each subtask to begin checkpointing
  - high values typically indicate constant backpressure
  - constant backpressure → application is under provisioned
- amount of data buffered during alignment and alignment duration
  - ideally these are small; large values indicate trouble
- unbalanced / asymmetric behavior
  - do some subtasks have much more state or signs of backpressure than others?

Heavy alignments

- Heavy alignment means different load on different paths
- E.g., could be caused by
  - skewed window emission (one node with some hot keys)
  - GC stall of one subtask
- Might want to set min-time-between-checkpoints if problem is severe
- Unaligned checkpoints (planned for Flink 1.10) should resolve this issue
  - see FLIP-76

Reducing the likelihood of checkpoint failures

- Don't implement operators that can block (e.g., don't do synchronous i/o)
- Reduce latency
- Upgrade to Flink 1.10 when it arrives
  - FLIP-27 and FLINK-10886 will support event time alignment between sources
  - FLIP-76: unaligned checkpoints will allow checkpoint barriers to overtake in-flight data, and make the in-flight data part of the checkpoint
    - goal is to make checkpointing immune to backpressure

Avoid DDOSing other systems

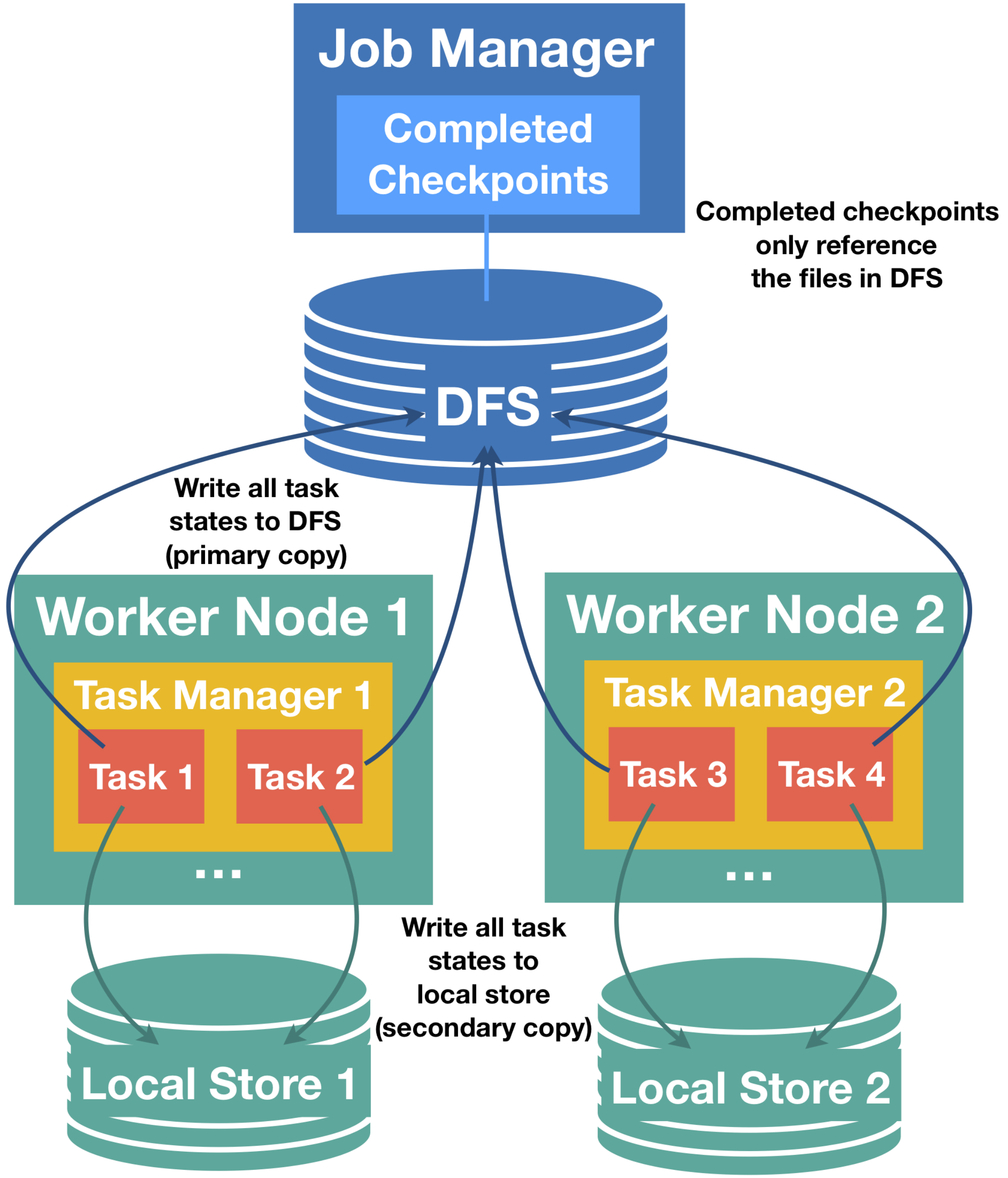
Scenario:

- Job with 5000 stateful subtasks
- Checkpoint interval: 1 second
- State size: KBs per subtask
- Problem: S3 will block off connections after exceeding 1000s of requests/sec
- Solution: reduce FS stress for small state
  - by increasing the value of `state.backend.fs.memory-threshold` (default: 1KB)
  - state chunks smaller than this value are stored inline in the checkpoint metadata file

Configuring Recovery

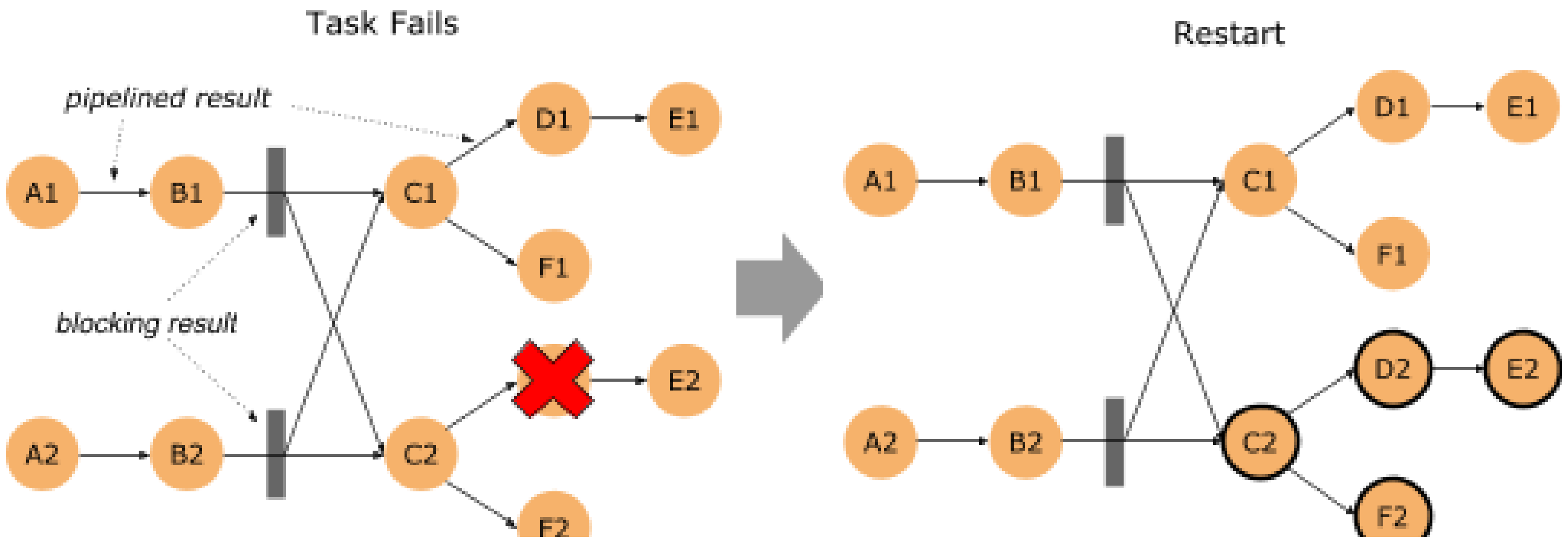
Local recovery: setup and cost

- How to configure:
  - `state.backend.local-recovery`
    - disabled by default
  - `taskmanager.state.local.root-dirs`
    - where the files used for local recovery are stored
- Impact:
  - ~~`MemoryStateBackend`~~: not supported
  - `FsStateBackend`: duplicate writes + storage
  - `RocksDBStateBackend`:
    - incremental checkpoints: no cost
    - full checkpoints: duplicate writes + storage



Fine-grained recovery (batch jobs)

- Rather than cancelling all tasks and restarting the whole job, recovery can be limited to only those tasks in the same *failover region*
- Regions are disjoint sets of tasks connected via pipelined data exchanges
- Set `jobmanager.execution.failover-strategy` to "region" (default in 1.9)
- Set `ExecutionMode` in `ExecutionConfig` to `BATCH`



Fine-grained recovery (streaming jobs)

- Set `jobmanager.execution.failover-strategy` to "region" (default in 1.9)
- Only applies to embarrassingly parallel jobs
  - no keyBy()
  - no rebalance

See FLIP-1: Fine Grained Recovery from Task Failures for details.

Wrap-up; we’ve looked at

- A high-level overview of how RocksDB is organized, and how to optimize/tune it
- Expiring state with state TTL
- How to approach checkpoint failures
- Tuning recovery

ververica