Understanding Scheduling and Checkpointing

Janani Ravi
CO-FOUNDER, LOONYCORN
www.loonycorn.com

Overview

FIFO and Fair scheduling

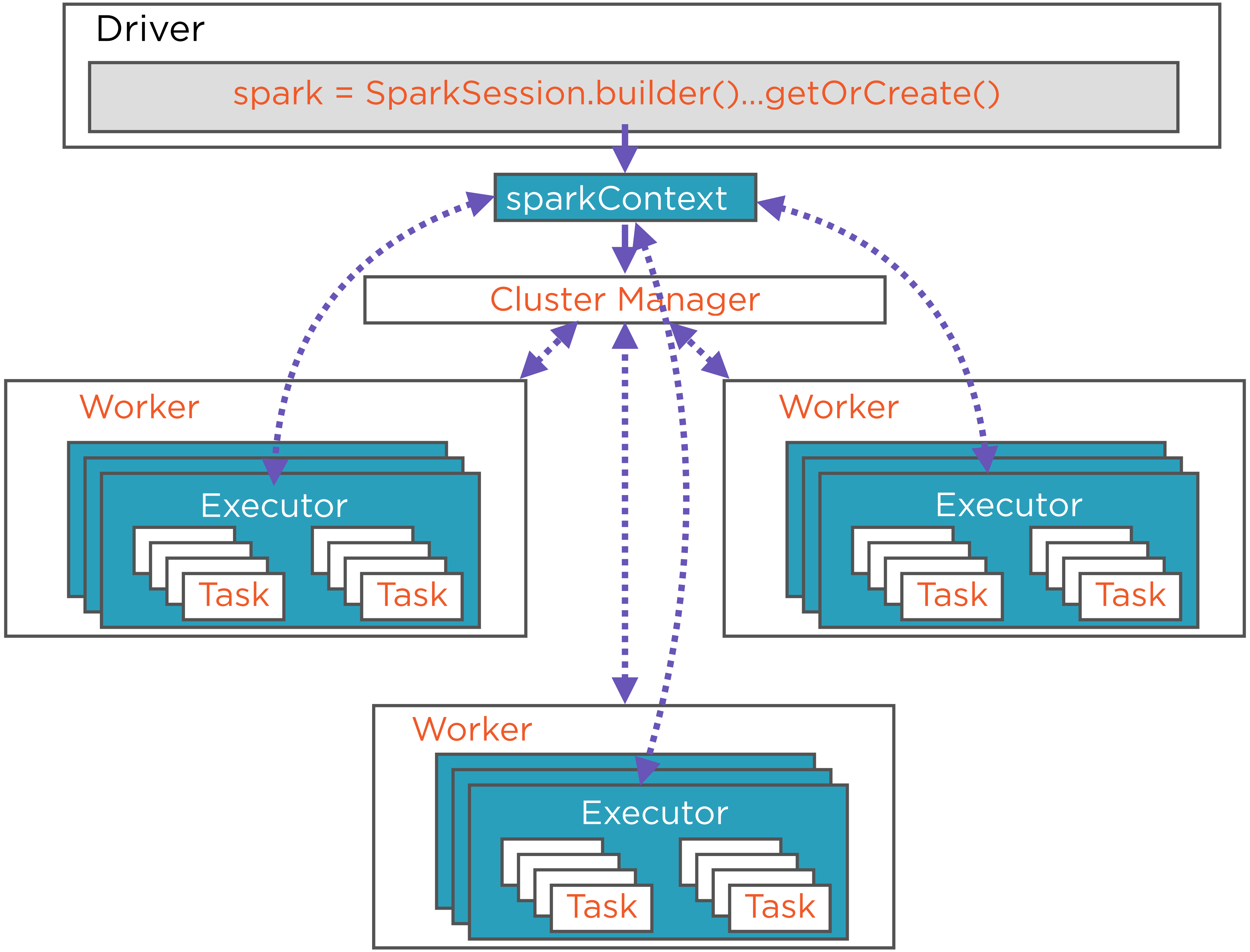
RDD lineage and recovery from node crashes

Fault tolerance semantics

Checkpointing and write-ahead logs

Scheduling in Spark

Spark 2.x Architecture



Spark Driver and Scheduling

Driver application runs separate process to execute commands

- creates SparkContext
- contains various schedulers

SparkContext in turn

- schedules job executions
- interacts with cluster manager

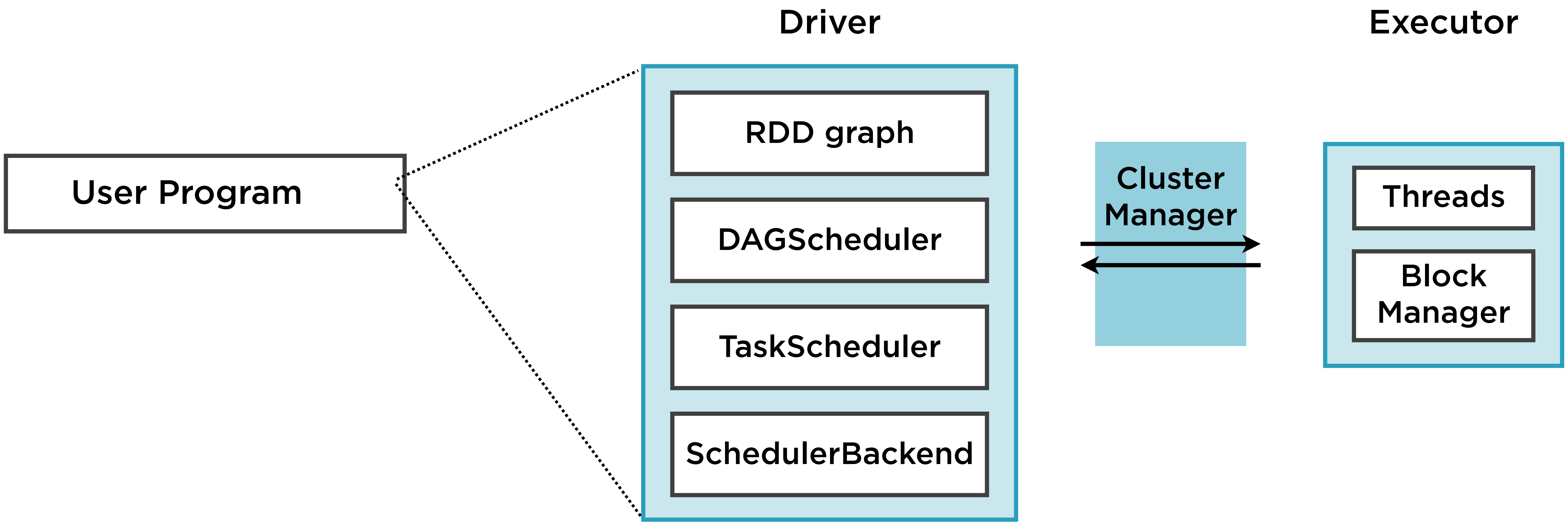
Spark Driver and Scheduling

Spark Driver contains

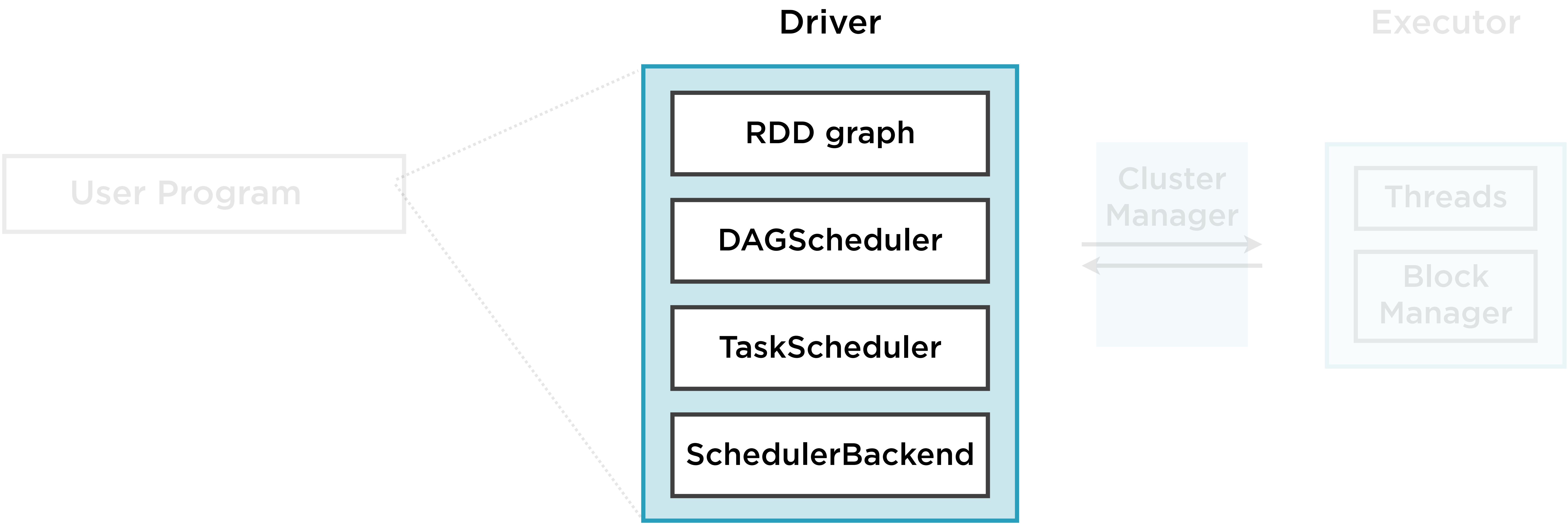
- DAG Scheduler
- Task Scheduler
- Scheduler Backend

Executors in turn run tasks scheduled by driver

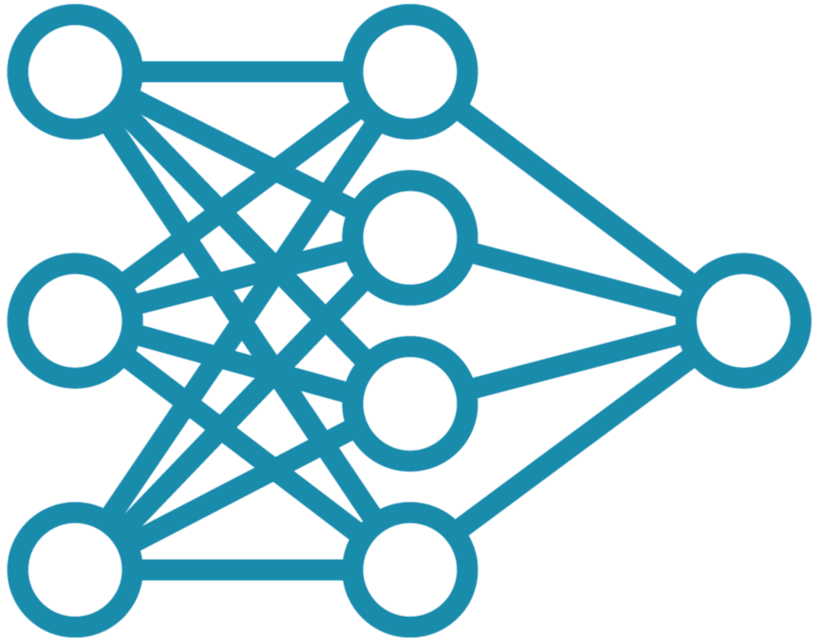
Spark Driver and Scheduling



Spark Driver and Scheduling



DAG Scheduler



Creates a Directed Acyclic Graph of stages for each job

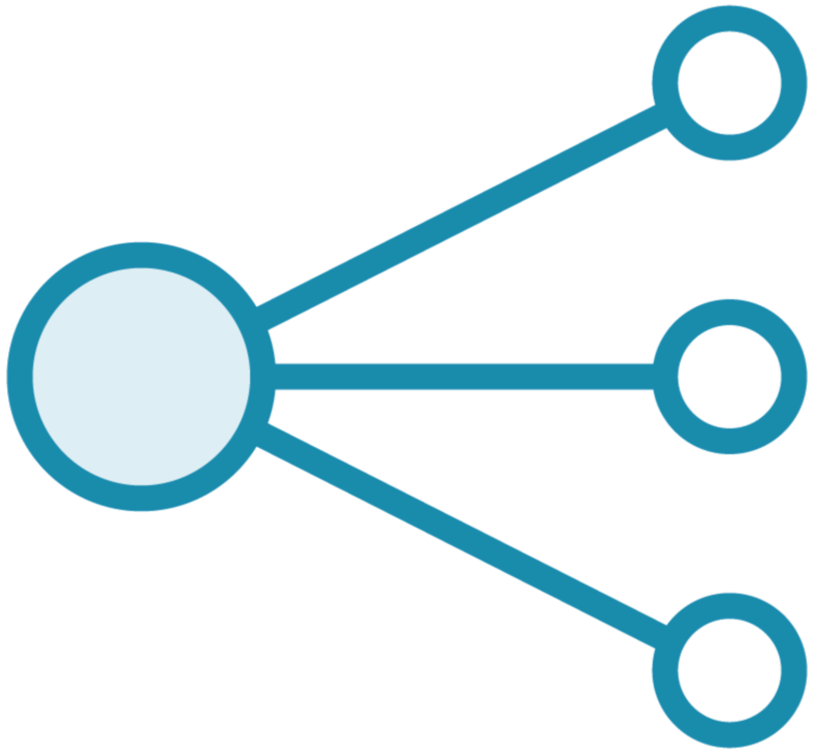
Submits to the TaskScheduler

Determines preferred locations for tasks

Based on cache status and file locations

Determines schedule to run jobs

Task Scheduler

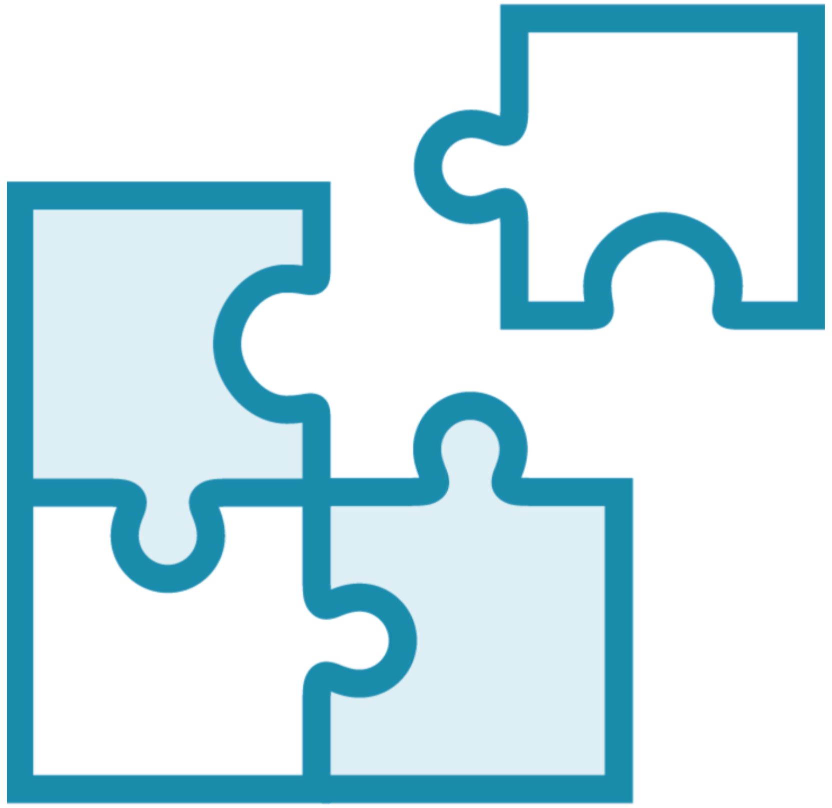


Executes tasks on the cluster

Retries tasks in case of failures

Handles slow-running tasks

Scheduler Backend



Backend interface for scheduling systems

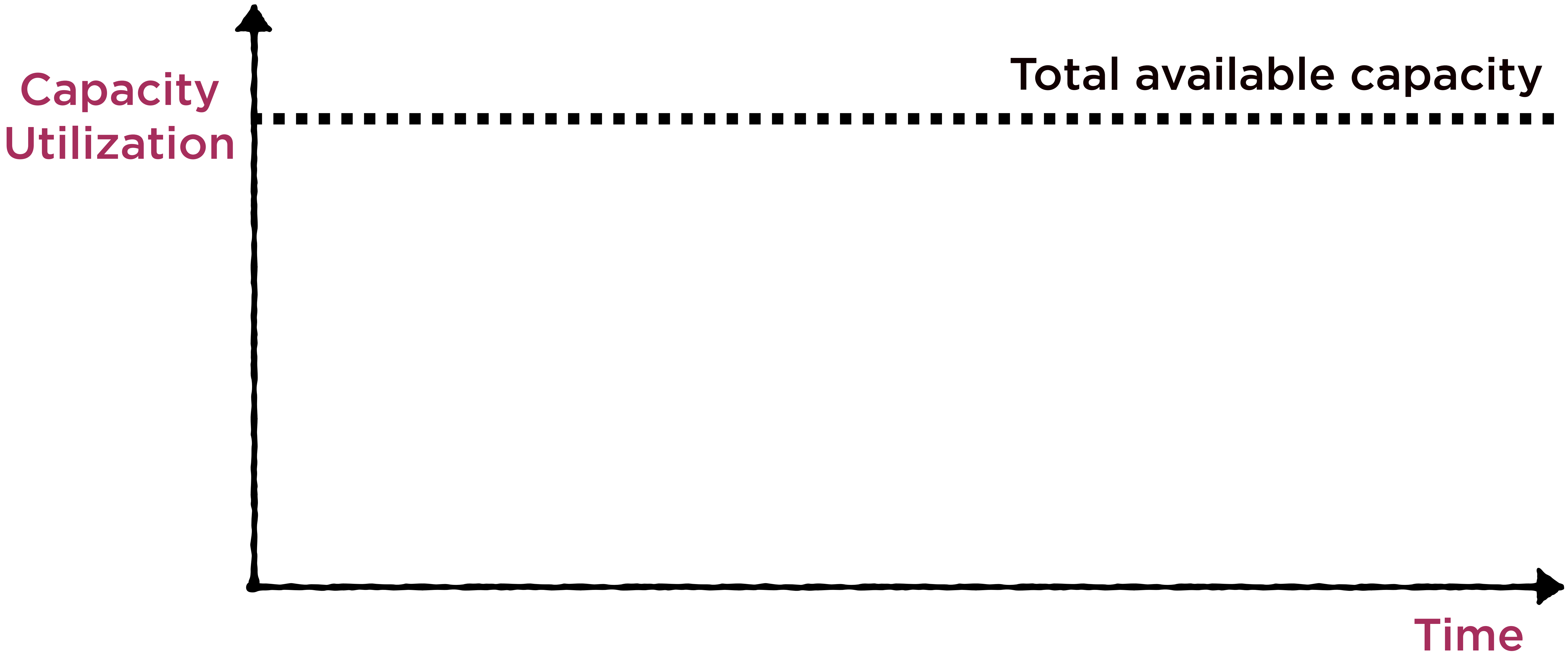
Allows plugging in different cluster managers

YARN, Mesos, Standalone, Kubernetes

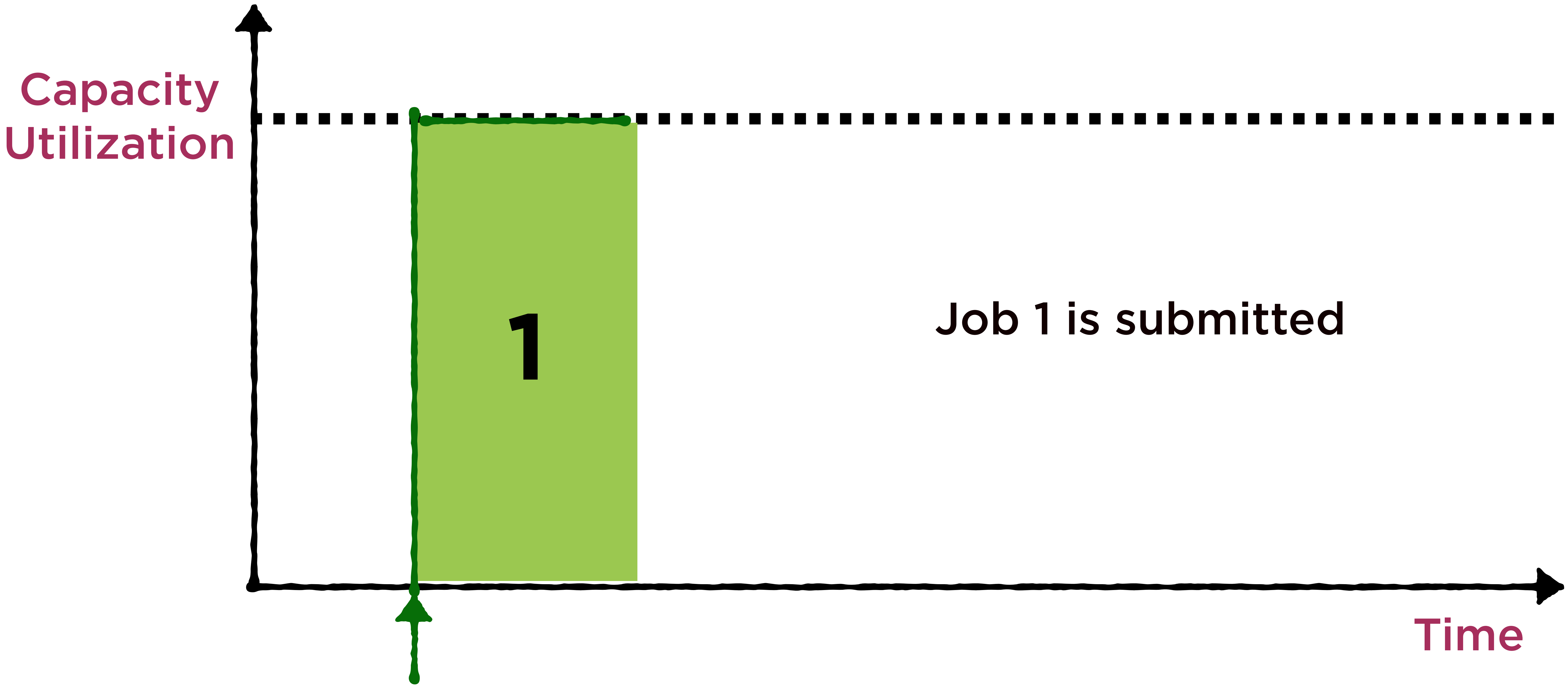
FIFO and Fair Scheduling

FIFO Scheduler

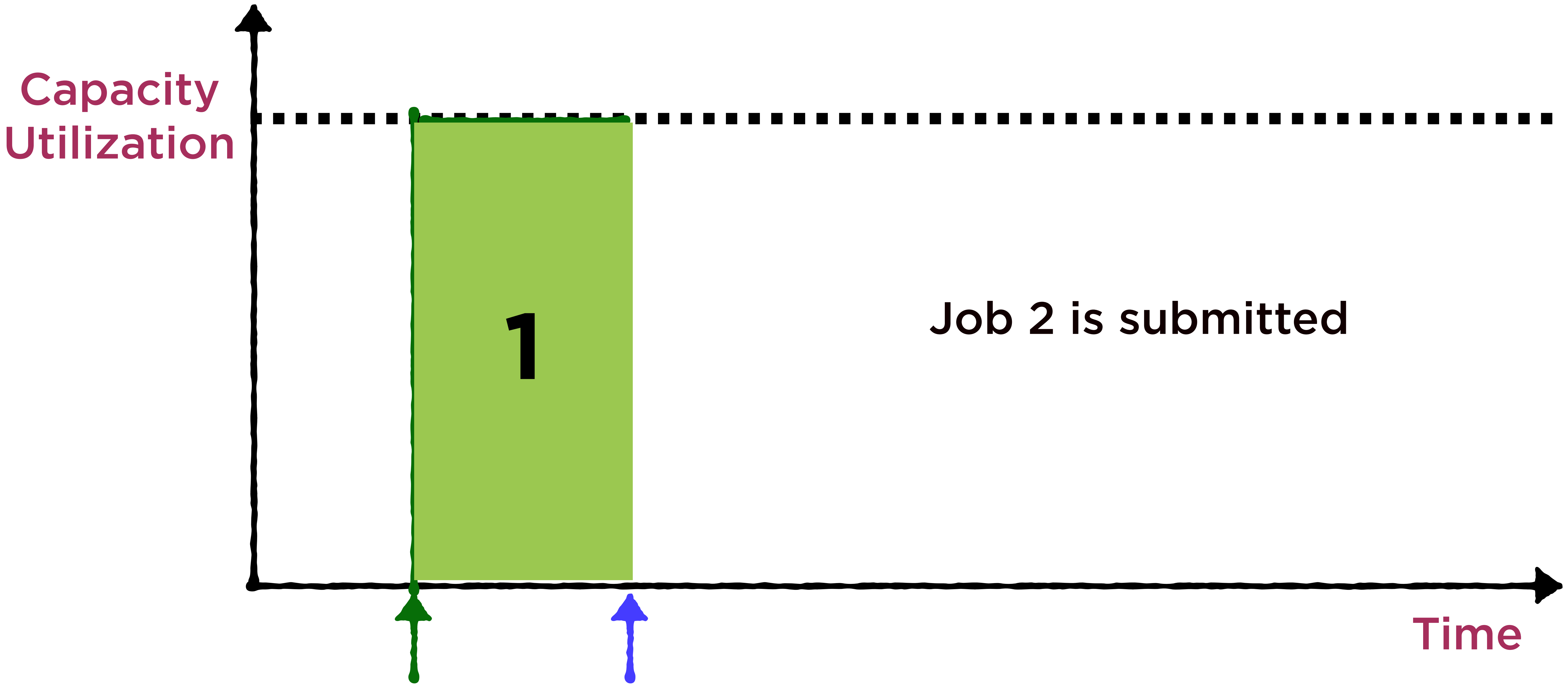
FIFO Scheduler



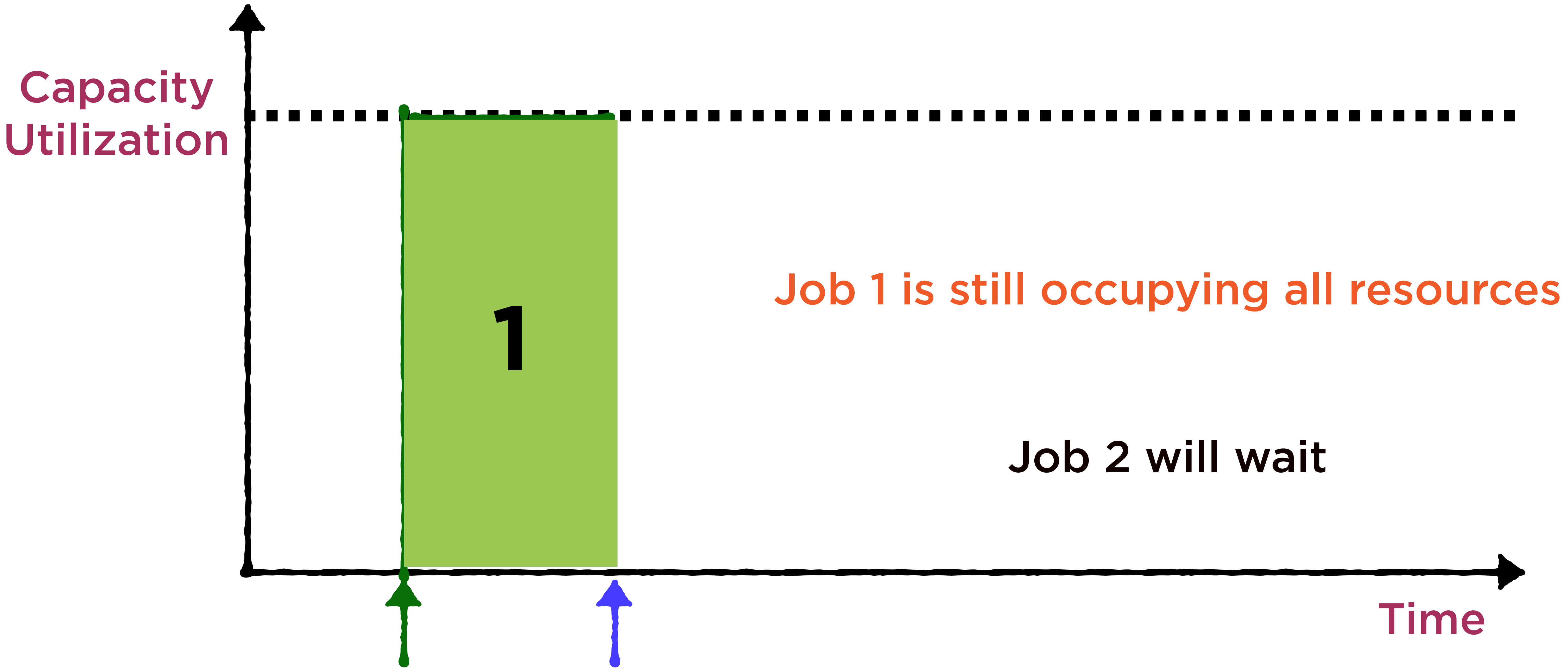
FIFO Scheduler



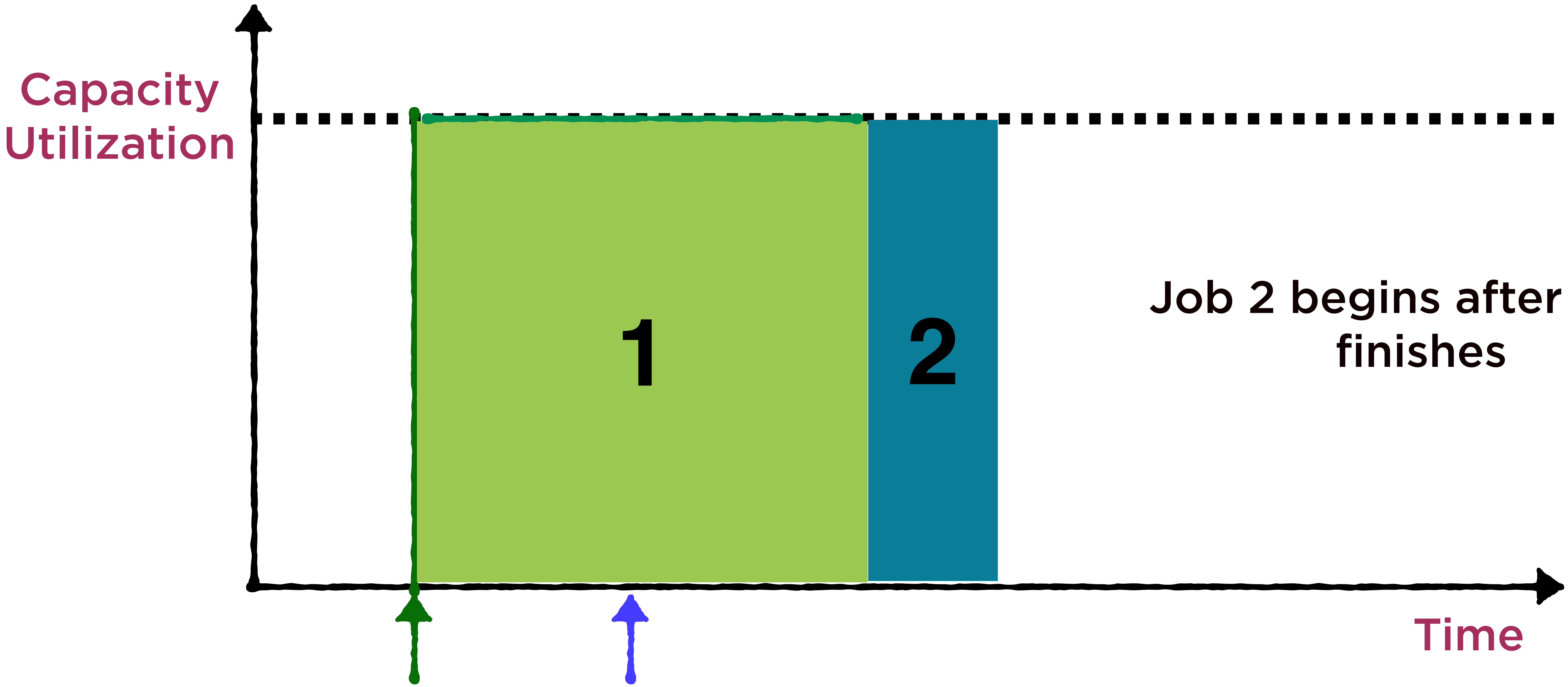
FIFO Scheduler



FIFO Scheduler

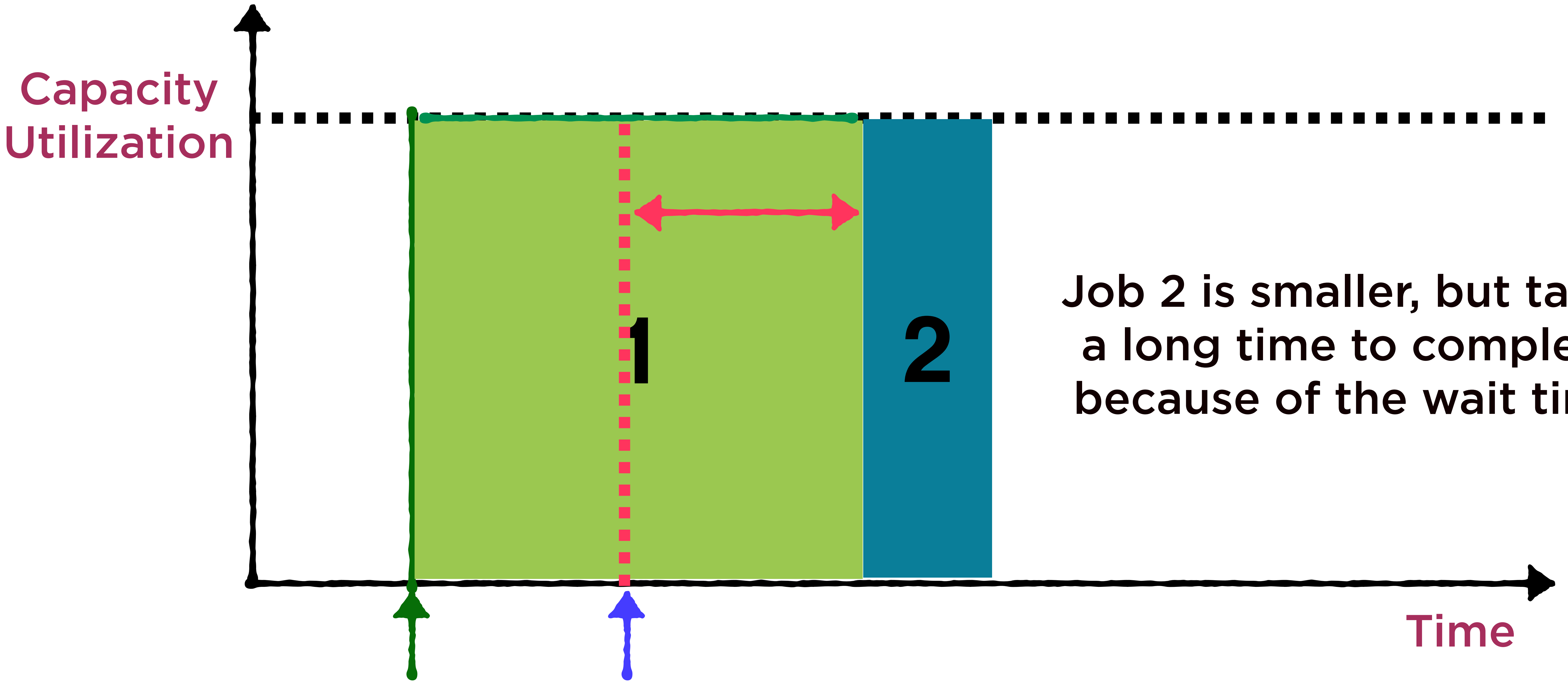


FIFO Scheduler



Job 2 begins after Job 1 finishes

FIFO Scheduler



The FIFO scheduler can result in **very long** wait times

Fair Scheduler

Resources are always proportionally allocated to all jobs

Zero wait time for any job

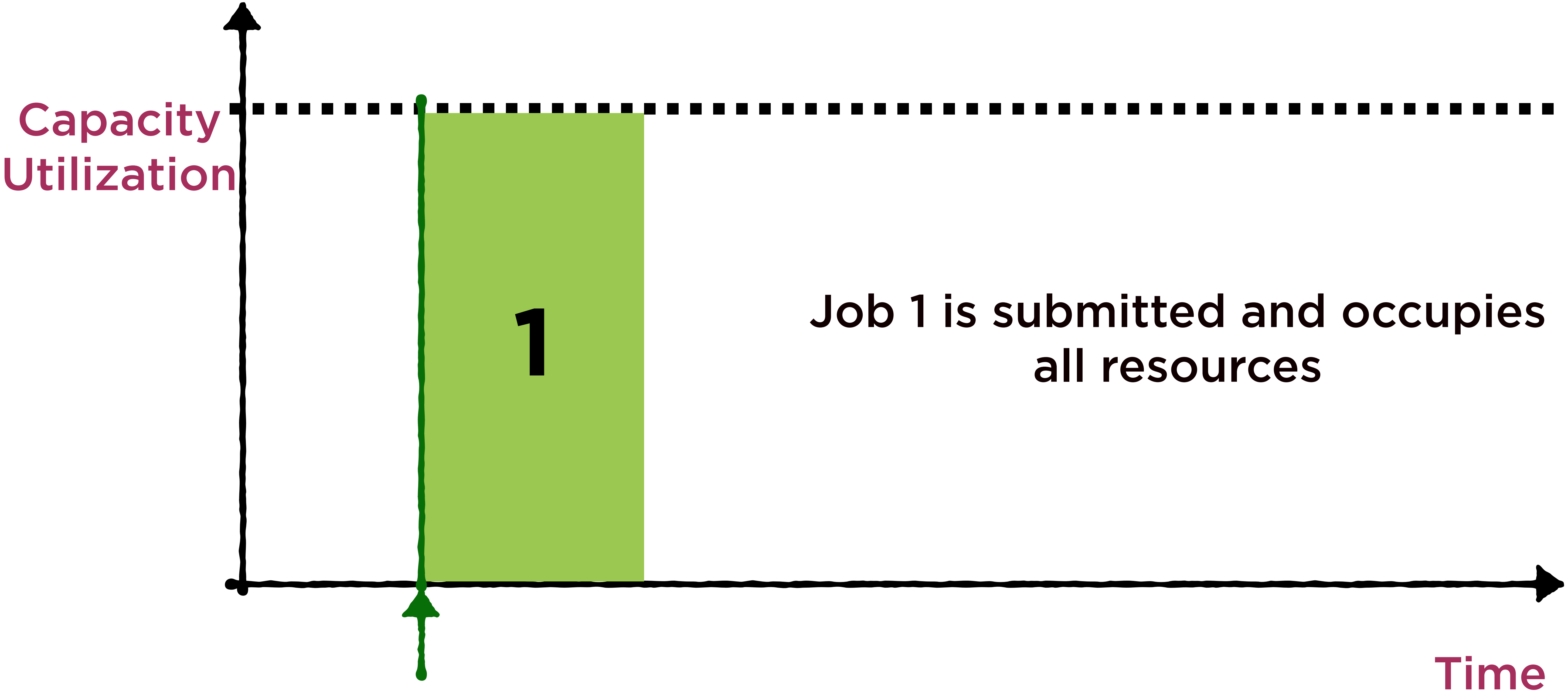
Can also specify job priorities

Priorities used as weights to allocate cluster resources

Fair Scheduler

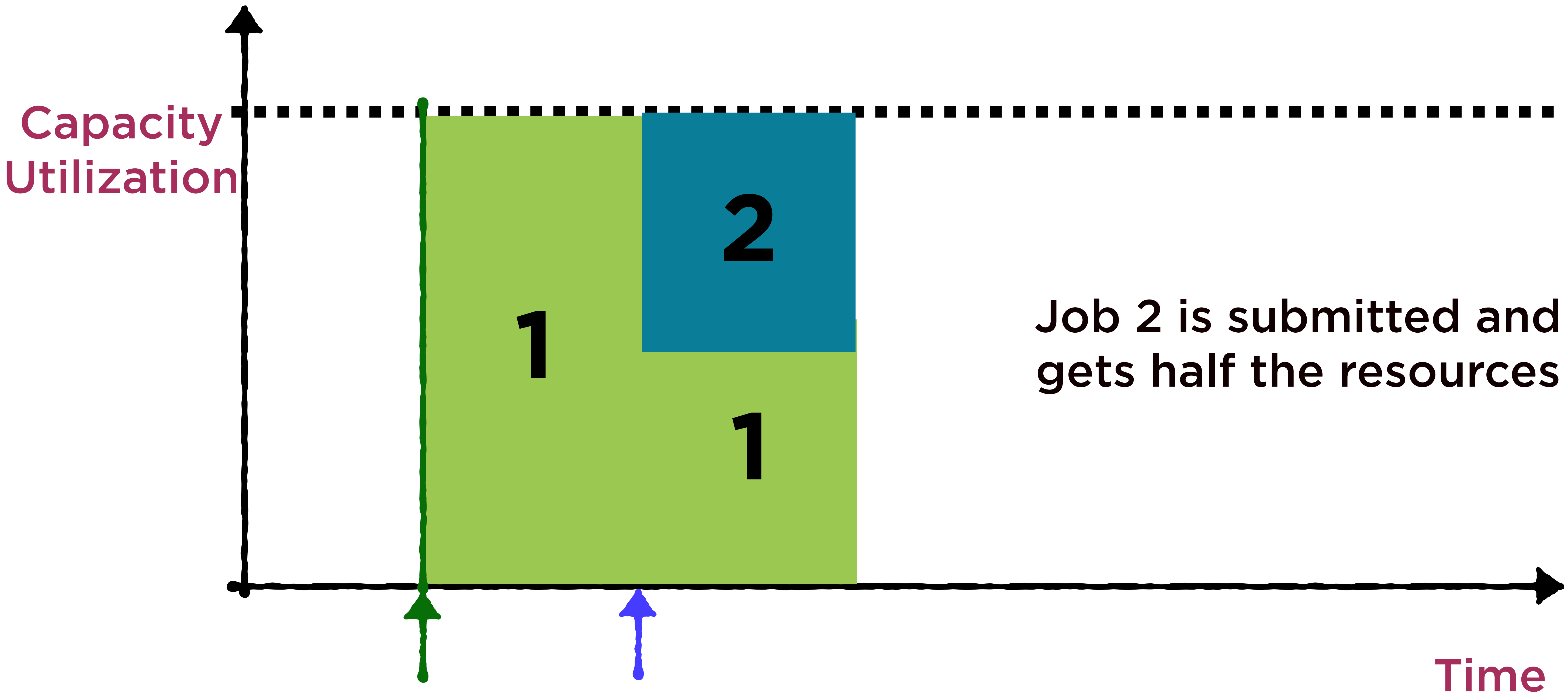
- Organizes jobs into pools
- Divides resources fairly between pools
- Separate pool for each user
- Allocates minimum shares to pools

Fair Scheduler



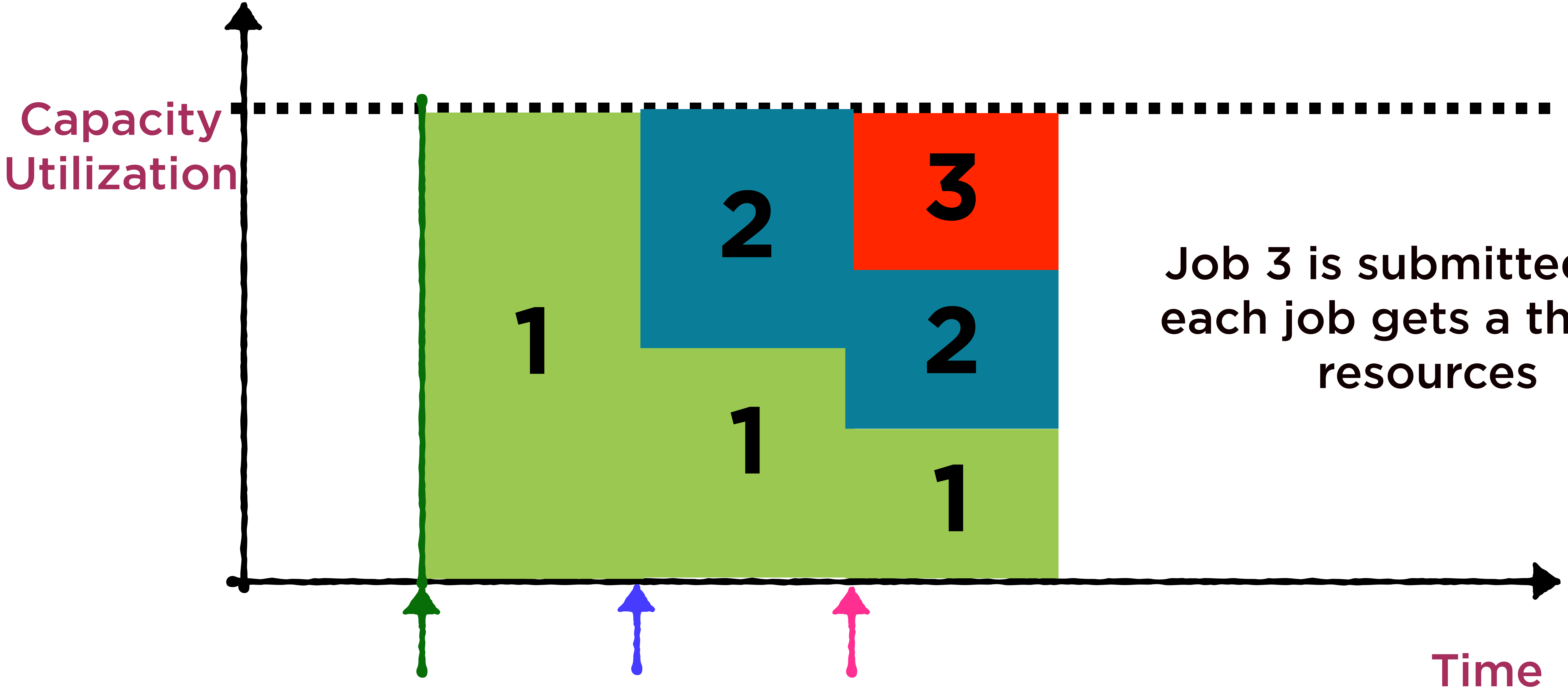
Job 1 is submitted and occupies all resources

Fair Scheduler



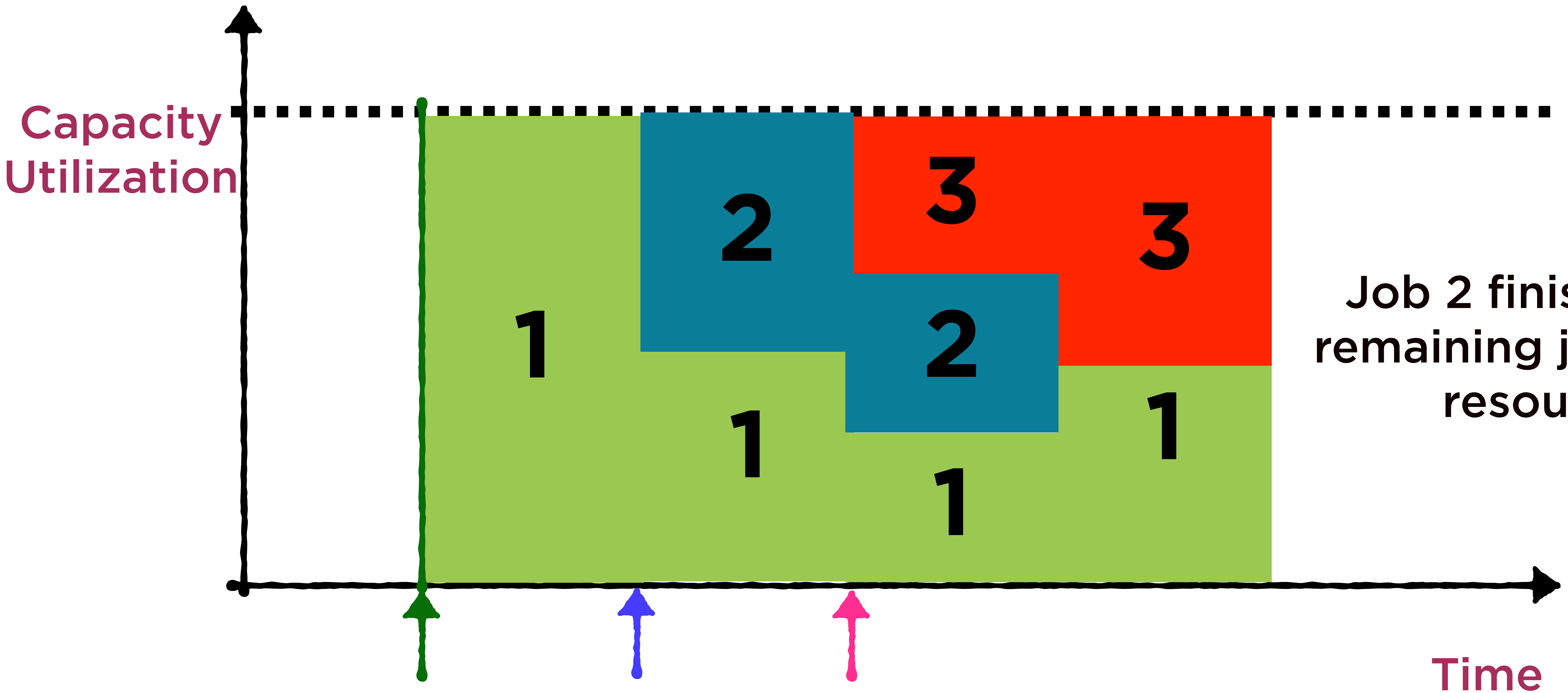
Job 2 is submitted and gets half the resources

Fair Scheduler



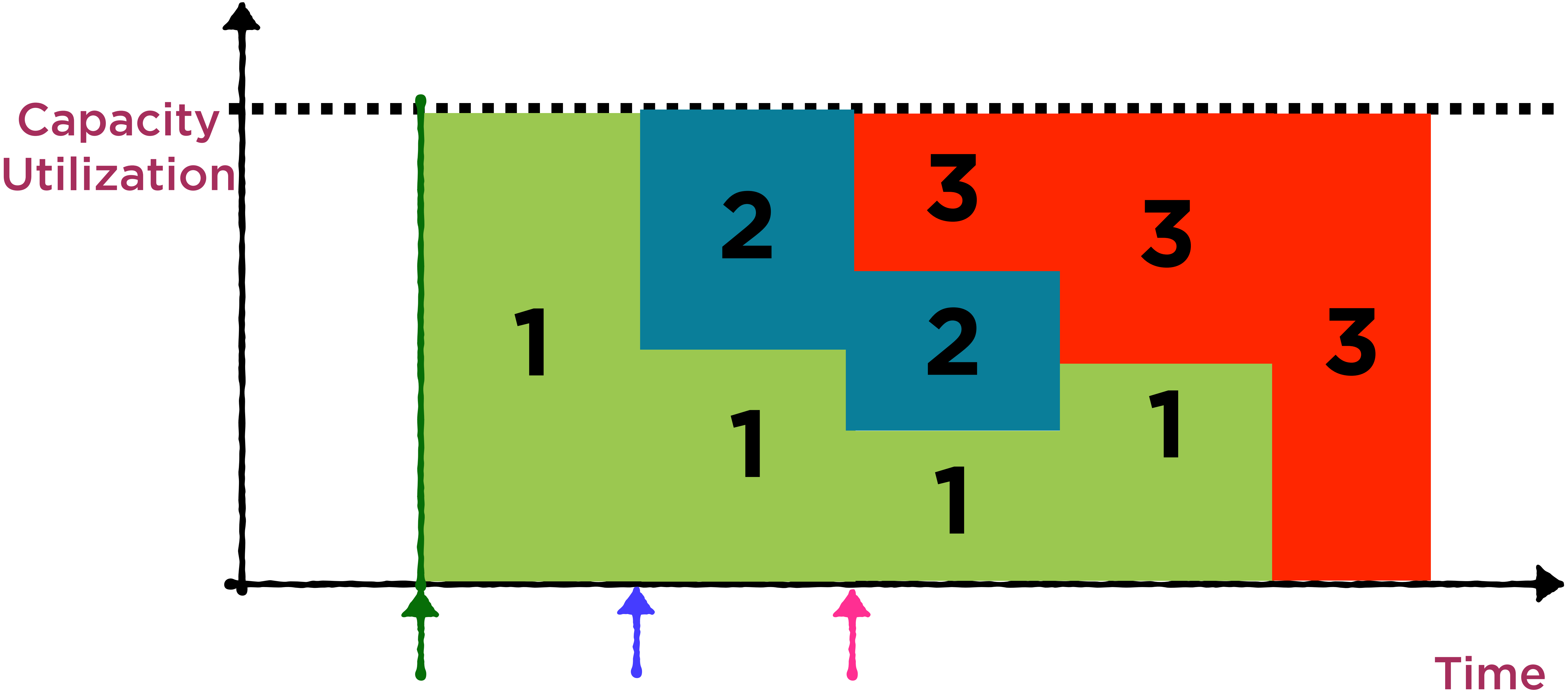
Job 3 is submitted and each job gets a third of resources

Fair Scheduler



Job 2 finishes and remaining jobs share resources

Fair Scheduler



RDDs and Fault Tolerance

Mechanism for Fault Tolerance in Spark 2.x is essentially unchanged from Spark 1.x

RDDs are still the **fundamental building blocks** of Spark

Characteristics of RDDs

Partitioned

RDDs are split across nodes in a cluster

Immutable

RDDs, once created, cannot be changed

Resilient

Can be reconstructed on node crashes

Characteristics of RDDs

Partitioned

RDDs are split across nodes in a cluster

Immutable

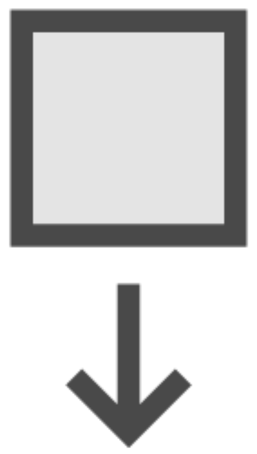
RDDs, once created, cannot be changed

Resilient

Can be reconstructed on node crashes

RDDs Are Resilient

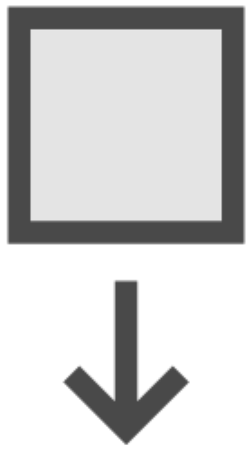
RDDs can be created in 2 ways



Reading a file

Transforming another RDD

RDDs Are Resilient

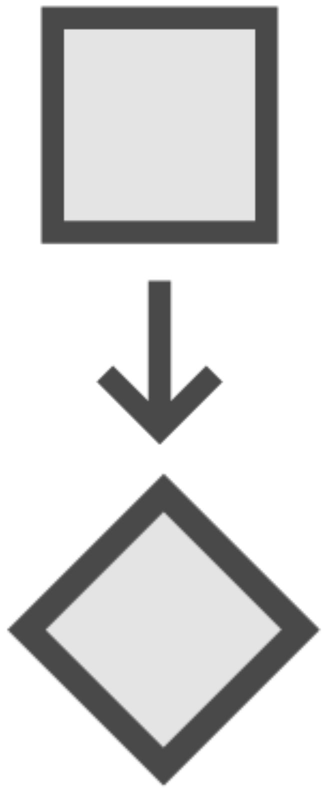


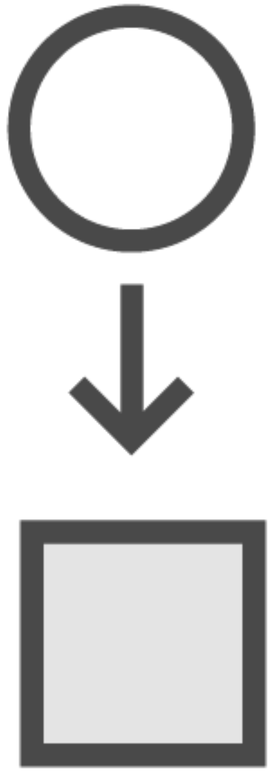
Reading a file

Transforming another RDD

Every RDD keeps track of *where* it came from

RDDs Are Resilient



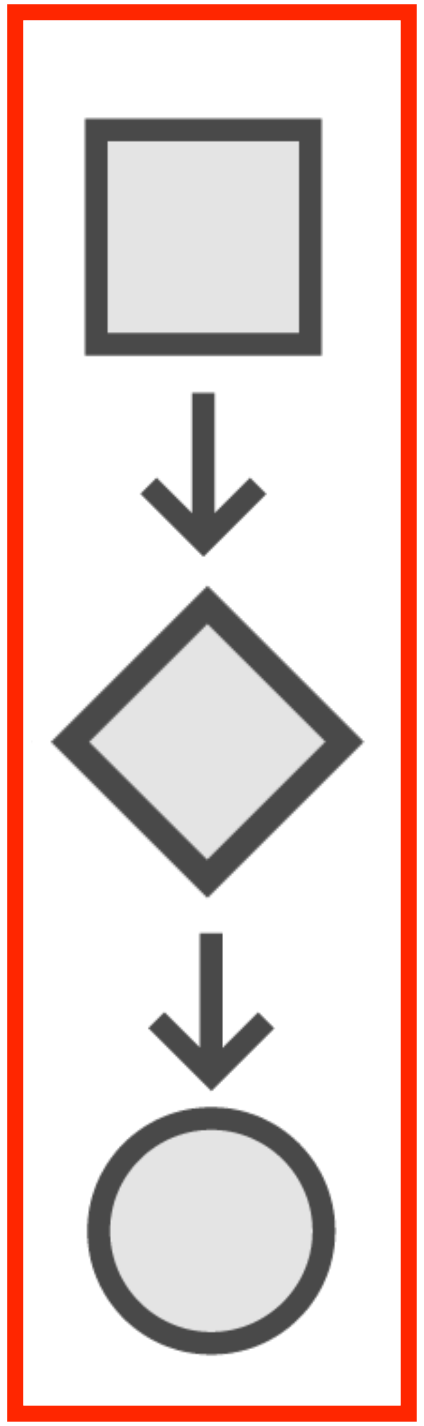


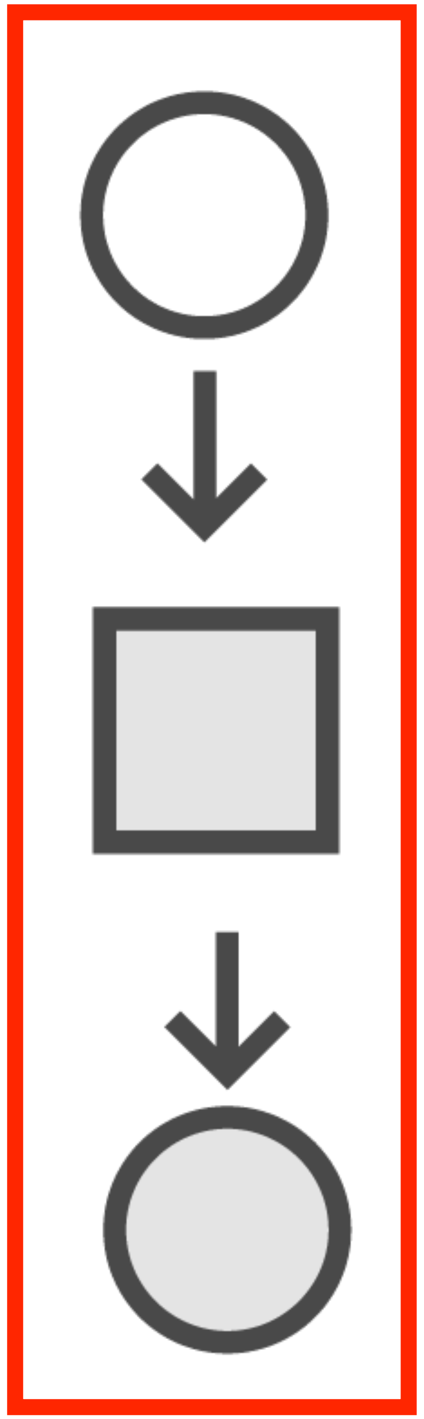
It tracks every transformation which led to the current RDD

RDDs Are Resilient

However many transformations it takes

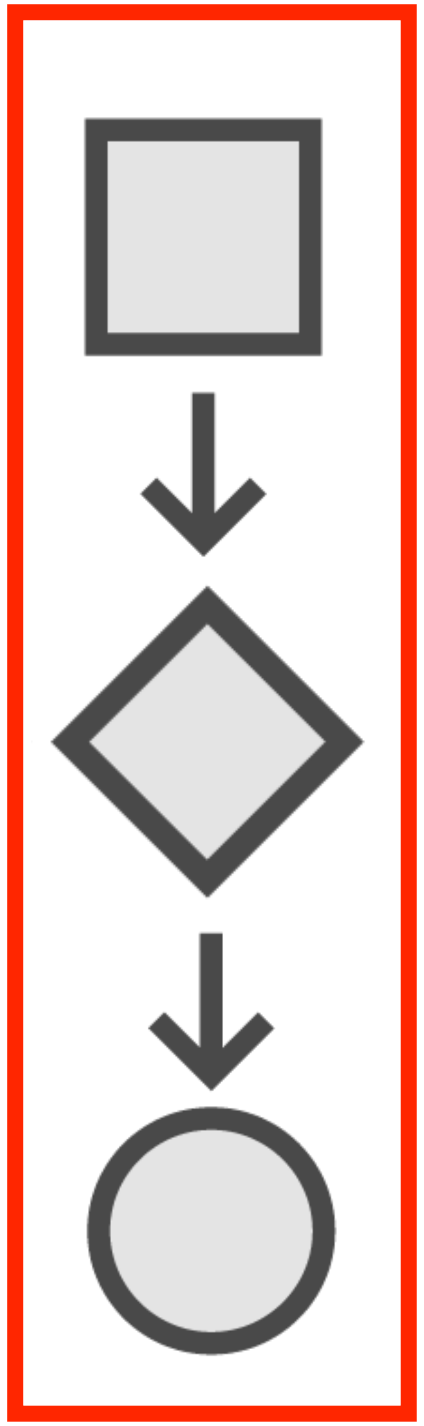
RDDs Are Resilient



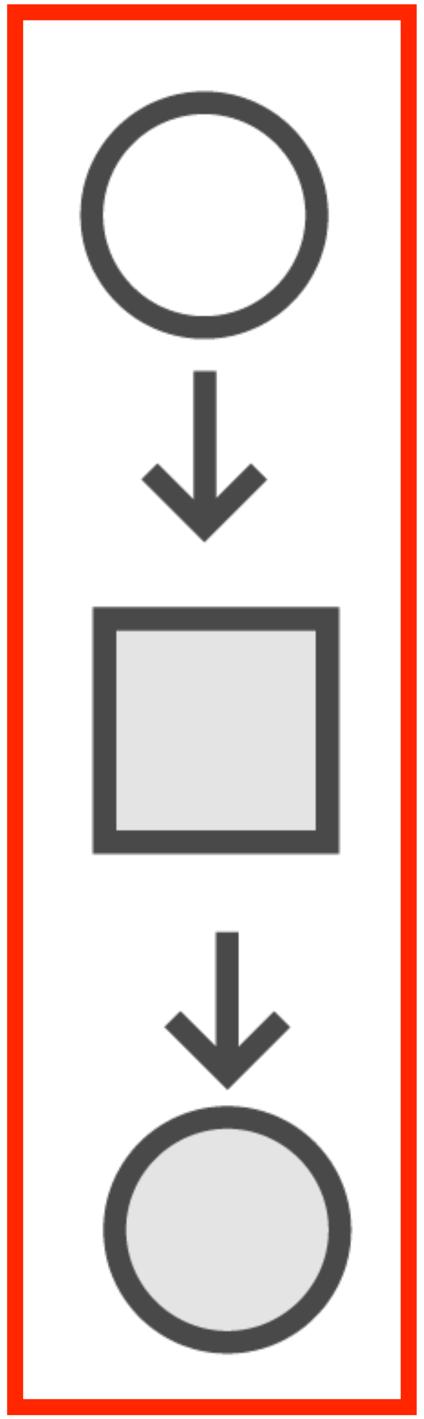


This is the RDD's lineage

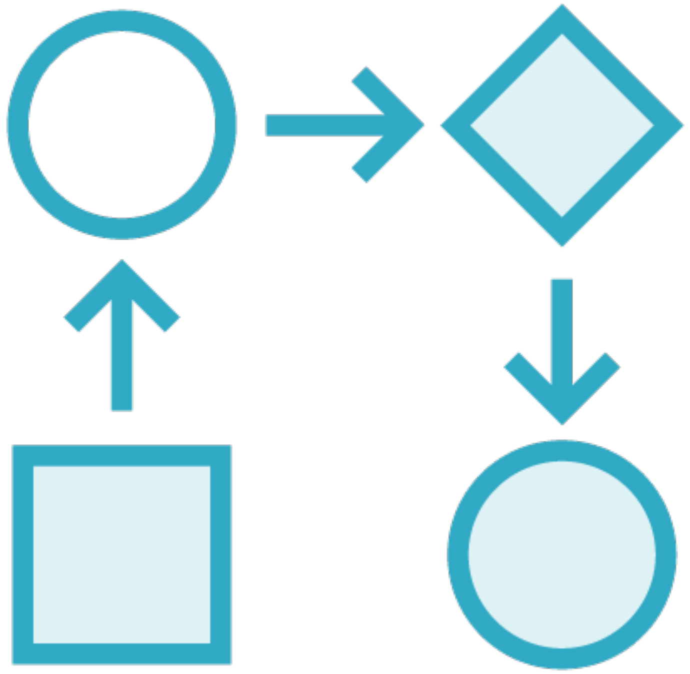
RDDs Are Resilient



None of the transformations are applied till we access the results



Lineage



Allows RDDs to be reconstructed when nodes crash

Allows RDDs to be lazily instantiated (materialized) when accessing the results

Receivers for Input Streams

Received data is replicated among multiple executors in the cluster

Data received and replicated:

- node crashes means the replicated copy of node can be used

Data received and not replicated:

- data needs to be replayed from the source

Fault Tolerance Semantics

Fault Tolerance

Types of guarantees

- At most once
- At least once
- Exactly once

For a streaming app to provide an exactly-once guarantee, each step must provide an exactly-once guarantee

Fault Tolerance

Steps in processing streaming data

- Receiving data
- Transforming data
- Pushing out data

Semantics of Receiving Data

Receivers can be reliable receivers

Acknowledge reliable sources only when data has been replicated

If receiver is restarted un-replicated data will be resent by the source

Semantics of Receiving Data

Receivers can be unreliable receivers

Do not send acknowledgements to the source

Data loss possible

To ensure recovery from job crashes input sources should be **replayable**

Semantics of Receiving Data

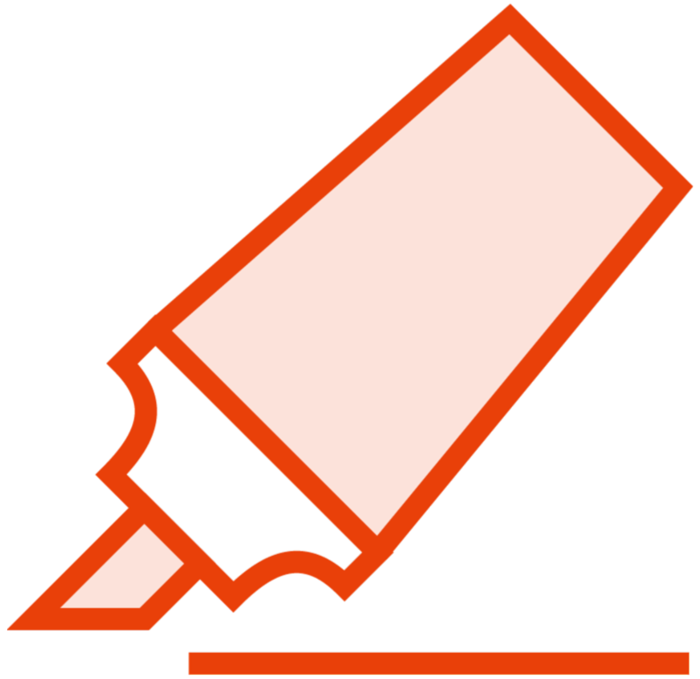
Semantics depend on receiver

Files, Kafka sources provide exactly-once guarantee

Unreliable receiver with Driver failure has undefined behavior

- Worst-case outcome

Write Ahead Logging

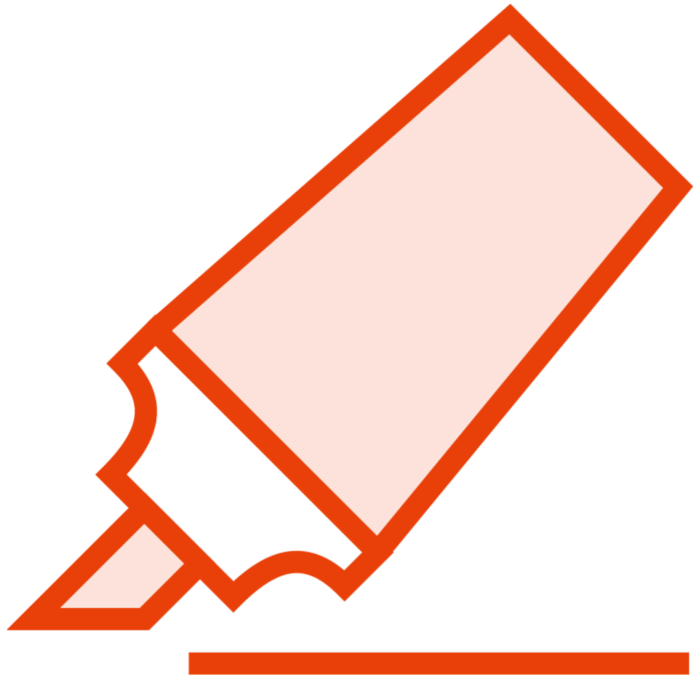


Important mechanism used to provide fault tolerance

- Introduced in Spark 1.2

Involves saving received data to reliable storage e.g. HDFS

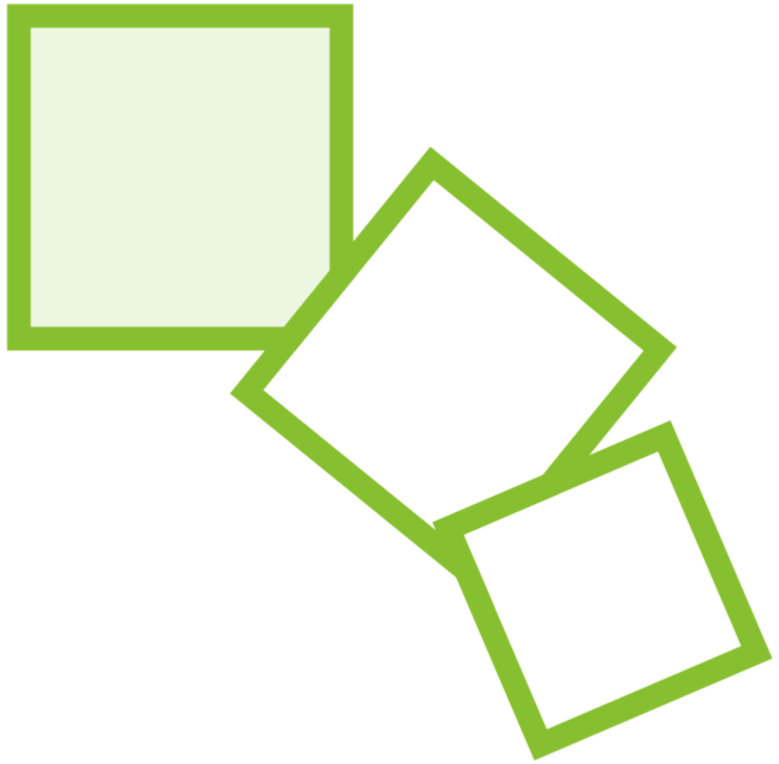
Write Ahead Logging



Avoids loss of past received data

- If enabled, provides at least once guarantee in almost all scenarios*
- *Except unreliable receiver with driver failure

Semantics of Transforming Data



Transformations are internal to Spark

RDDs provide exactly-once guarantee

Any data received exactly once will be processed exactly once

Semantics of Pushing Out Data

Output operations have at-least once semantics

Need additional processing to get exactly-once semantics

Two approaches possible

- Idempotent updates
- Transactional updates

Checkpointing

Checkpointing in Spark

Streaming applications need to be resilient to external failures

Spark Streaming uses checkpointing to maintain intermediate state

- Intermediate state must be saved to reliable storage e.g. HDFS

Helps recover from failures and ensure fault-tolerance

Checkpointing in Spark

Can configure query with checkpoint location on reliable storage

Recover previous progress and state of query, and resume

Thus, checkpointing and write ahead logging help recover from failures

Checkpointing in Spark

Metadata checkpointing

- Needed to recover from driver failures

Data checkpointing

- Needed whenever stateful transformations are used
- Stateful transformations combine data across batches

Incremental Checkpointing

Only key-value pairs that are changed are committed or aborted

(Other, unchanged key-value pairs are not touched)

Incremental Checkpointing is based on StateStore objects

StateStore

Abstraction for key-value pairs used in managing state in stateful stream processing; supports incremental checkpointing.

Demo

Recovering from failures using checkpoints

Summary

FIFO and Fair scheduling

RDD lineage and recovery from node crashes

Fault tolerance semantics

Checkpointing and write-ahead logs

Up Next:

Configuring Processing Models