SPARK SUMMIT 2017

CONTINUOUS APPLICATION WITH FAIR SCHEDULER

Robert Xue

Software Engineer, Groupon

About me

Software engineer, Behavioral Modeling team

Groupon Online Defense

- Malicious behavior detection
- A log stream processing engine

Tech stack

- Kafka
- Storm
- Spark on Yarn
- In-house hardware

Glossary

Application

- Created by spark-submit

Job

- A group of tasks
- Unit of work to be submitted

Task

- Unit of work to be scheduled

Scheduler

- Code name: SchedulableBuilder
- FIFO and FAIR.
- Update spark.scheduler.mode to switch

Job pool scheduling mode

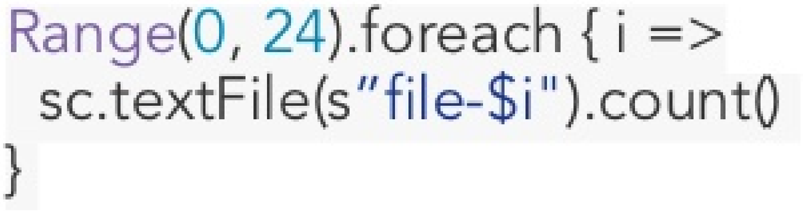
- Code name SchedulingAlgorithm
- FIFO and FAIR, applies to FAIR scheduler only
- Update fairscheduler.xml to decide

What does a scheduler do?

- Determine who can use the resources
- Tweak to optimize
  - Total execution time
  - Resource utilization
  - Total CPU seconds

When does your scheduler work?

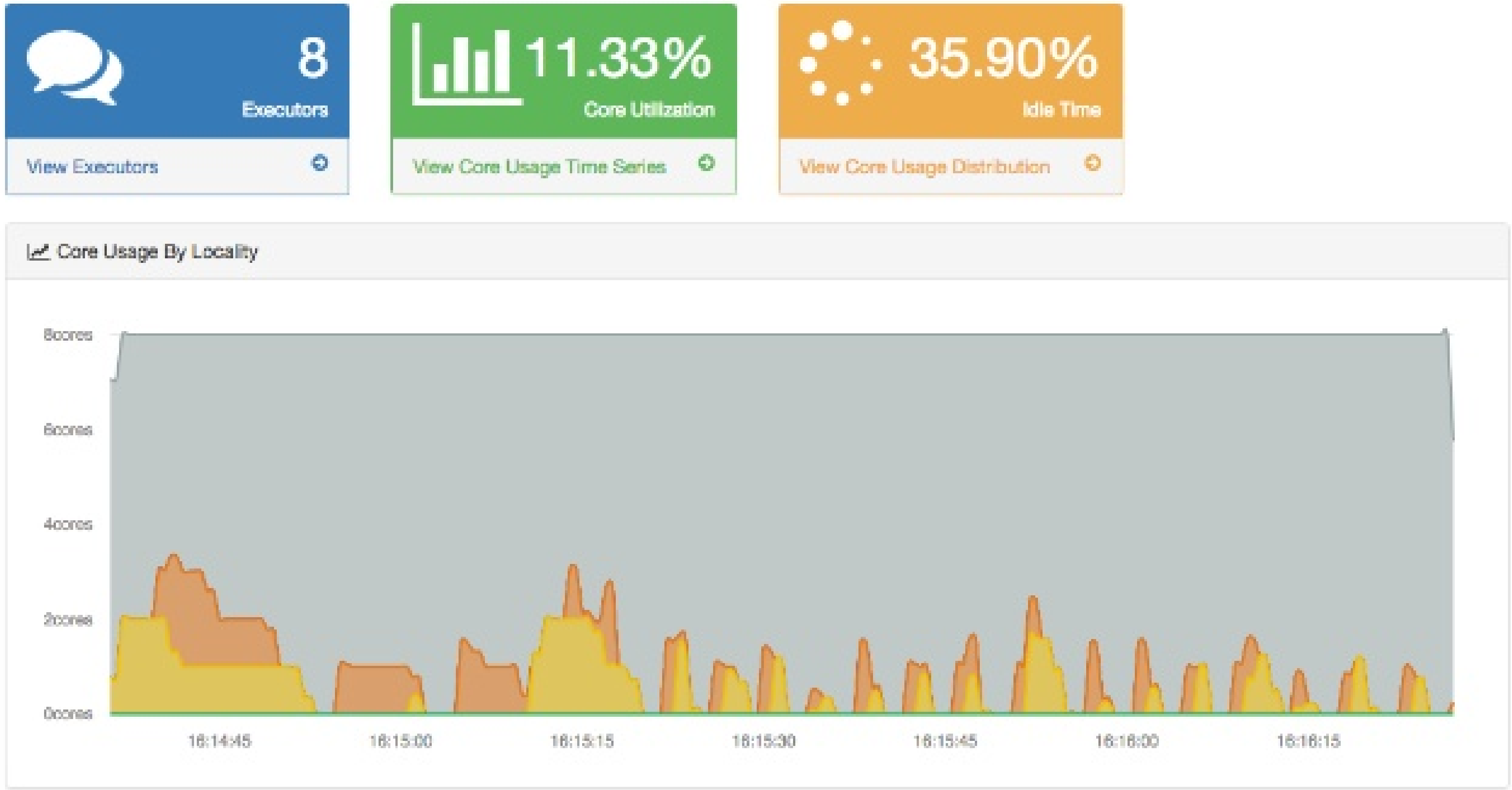
- Only when you have more than one job submitted to spark context
- Writing your rdd operations line by line != multiple jobs submitted at the same time



```
Range(0, 24).foreach { i =>
 sc.textFile(s”file-$i").count()
}
```

Run - 1
Sequentially submitting 24 file reading jobs

```
Range(0, 24).foreach { i =>
 sc.textFile(s"file-$i").count()
}
```

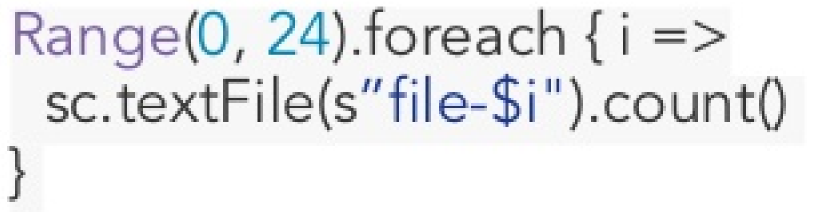


- 4 parts, ~500MB files * 24
- 8 cores
- 110 seconds
- 8 * 110 * 11.33% = 99.68 cpu seconds
- ~20 individual jobs clearly visible

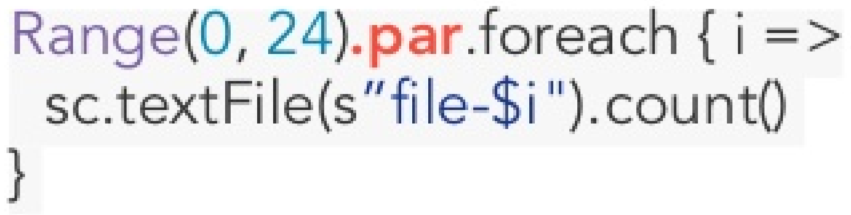
Node_local - Rack_local - Any - Unused

Submit your job in parallel

- Java Runnable
- Python threading
- Scala Future
  - Scalaz Task
  - Monix Task



```
Range(0, 24).foreach { i =>
 sc.textFile(s”file-$i").count()
}
```

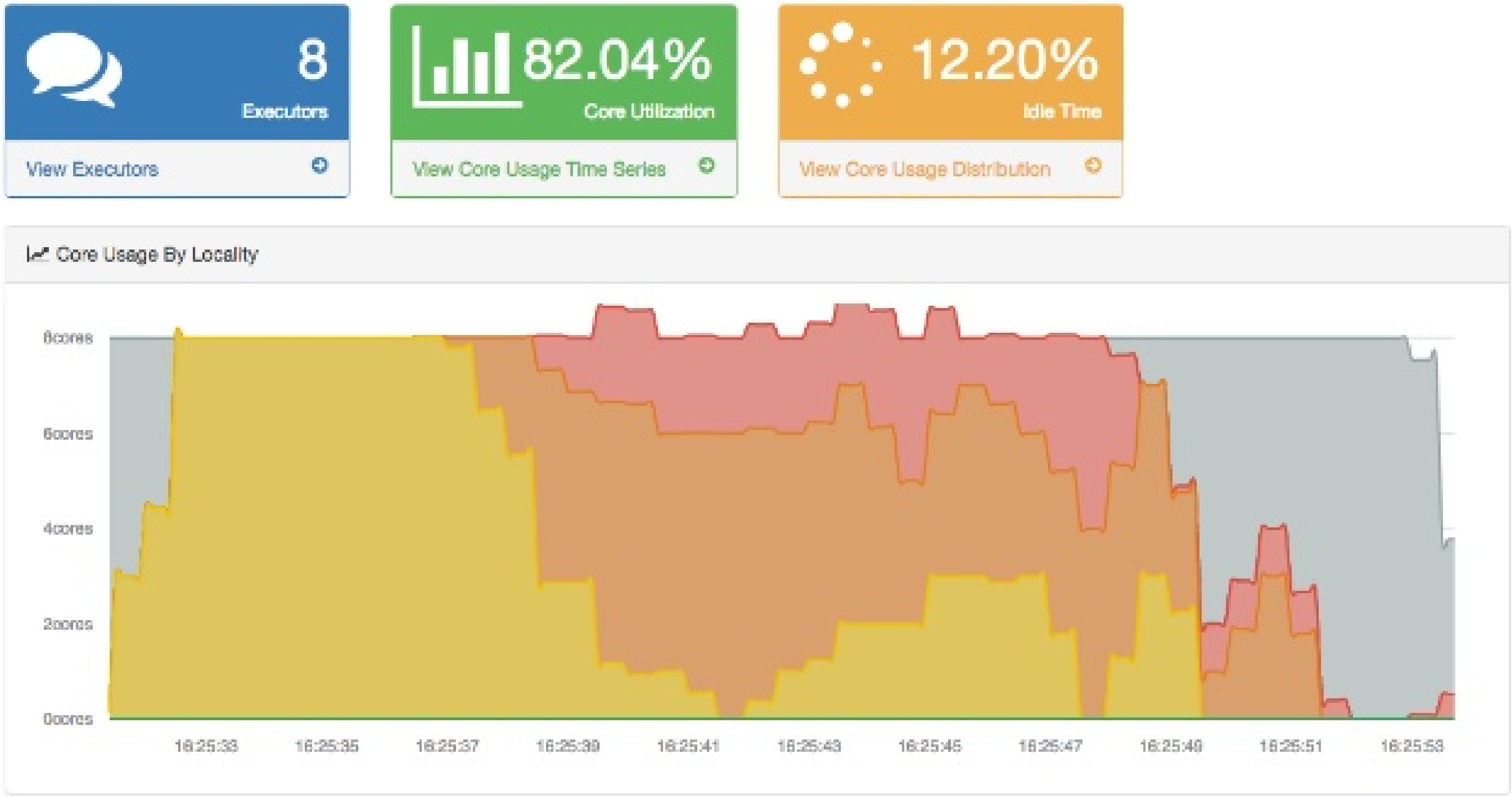


```
Range(0, 24).par.foreach { i =>
 sc.textFile(s”file-$i").count()
}
```

Run - 2
Parallel submit & FIFO(24)

```
Range(0, 24).par.foreach { i =>
 sc.textFile(s"file-$i").count()
}
```

- 8 cores
- ~~110~~ 22 seconds
- 8 * 22 * 82.04% = ~~99.68~~144.32 cpu seconds
- ~~0~~15 seconds 100% utilization period



8
Executors
View Executors

82.04%
Core Utilization
View Core Usage Time Series

12.20%
Idle Time
View Core Usage Distribution

Core Usage By Locality

Node_local - Rack_local - Any - Unused

Turn on FAIR scheduler

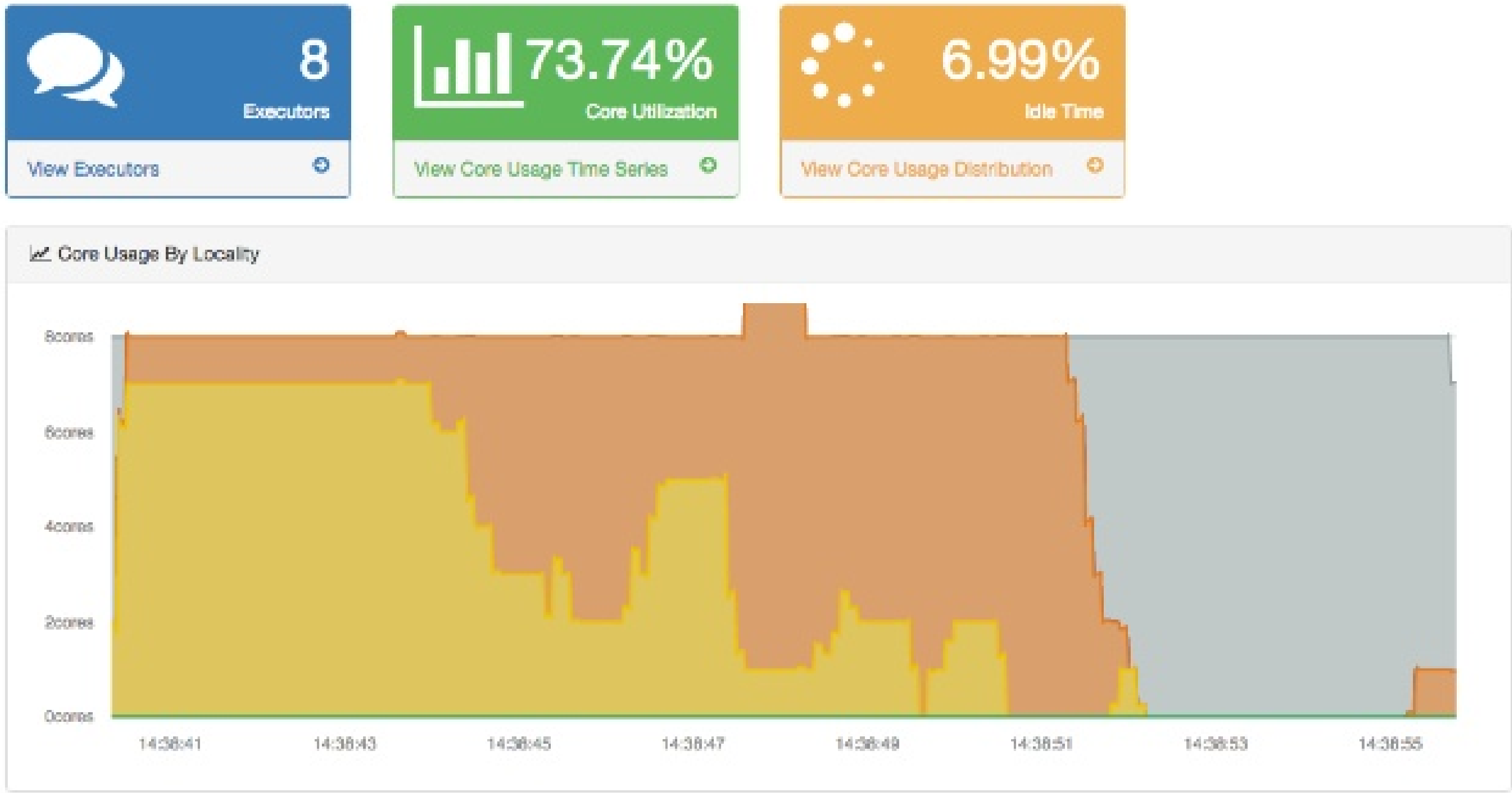
Provide --conf spark.scheduler.mode=FAIR in your spark submit…

… it is that simple

Run - 3
Parallel submit & FAIR(24)

```
Range(0, 24).par.foreach { i =>
  sc.textFile(s”file-$i”).count()
}
```

- 8 cores
- ~~22~~ 15 seconds
- 8 * 15 * 73.74% = ~~144.32~~88.48$^{\text{record}}$ cpu seconds
- ~~Bad~~Great locality



8 Executors

View Executors

73.74% Core Utilization

View Core Usage Time Series

6.99% Idle Time

View Core Usage Distribution

Core Usage By Locality

Node_local - Rack_local - Any - Unused

Tweak your locality config

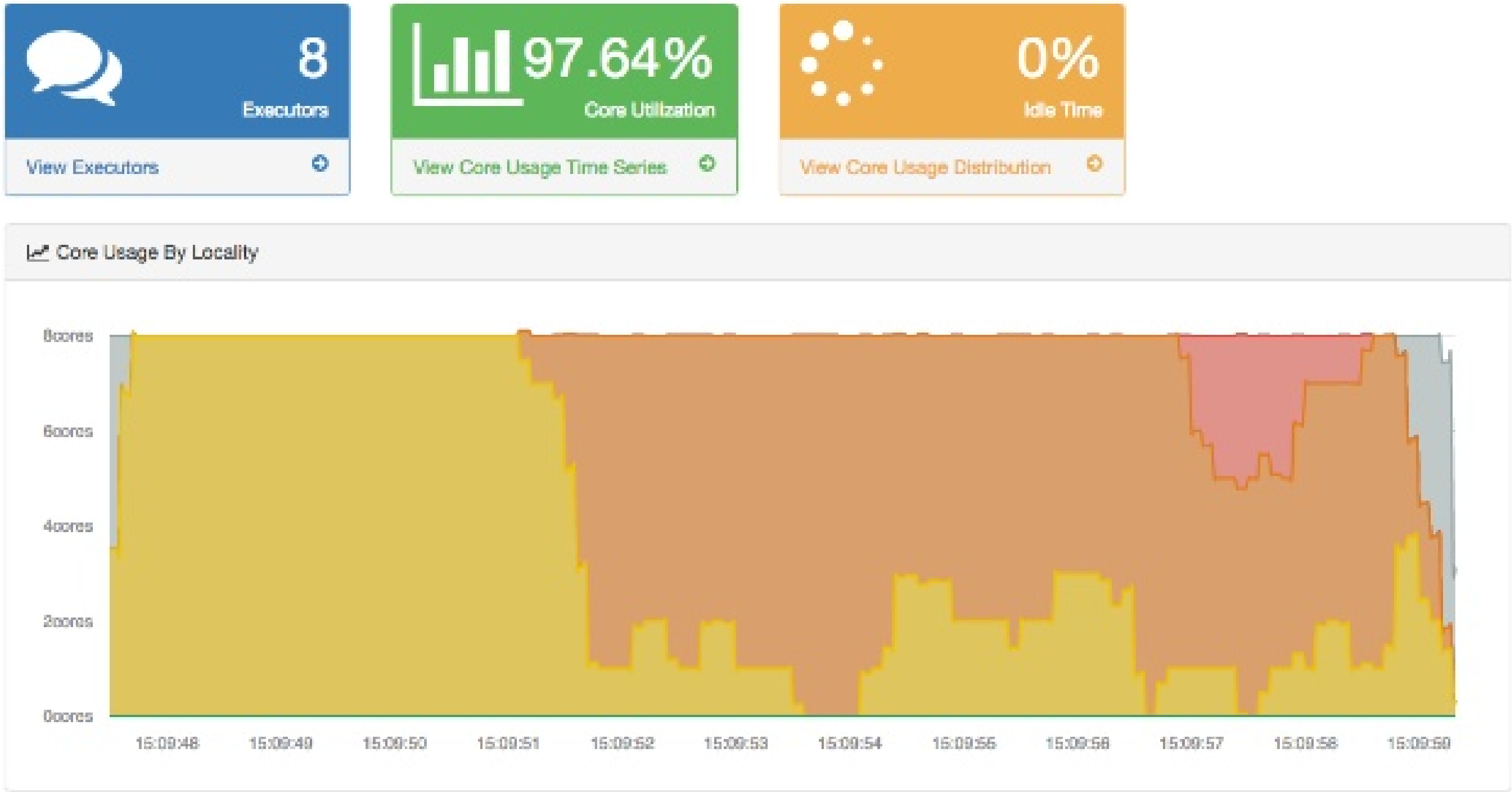
Provide --conf spark.locality.wait=1s in your spark submit…

- spark.locality.wait.process/node/rack available as well
- default is 3s

Run - 4
Parallel submit & FAIR(24) & locality.wait=1s

```
Range(0, 24).par.foreach { i =>
  sc.textFile(s"file-$i").count()
}
```

- 8 cores
- ~~15~~ 12$^{\text{record}}$ seconds
- 8 * 12 * 97.64% = ~~88.48~~93.68 cpu seconds



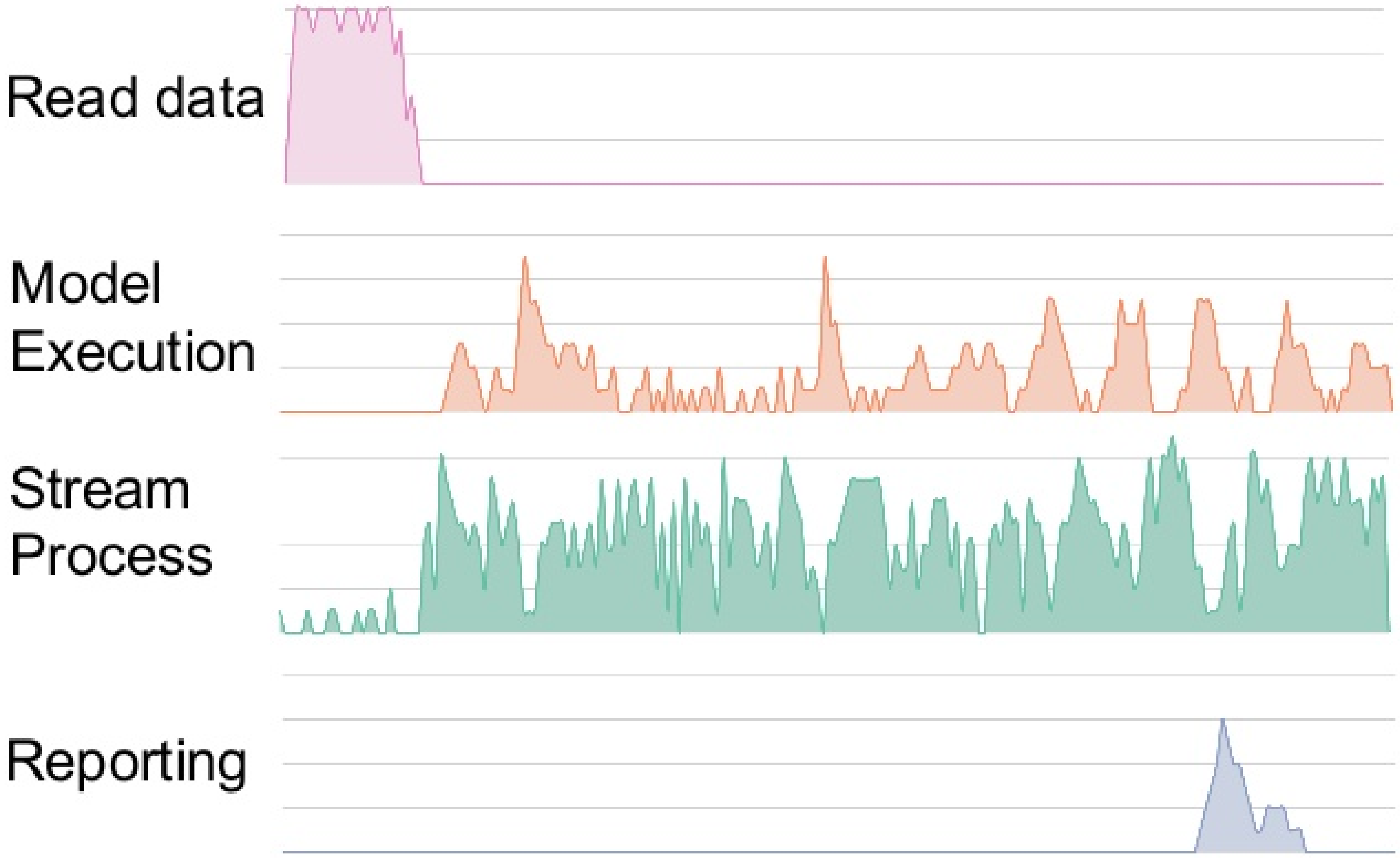
Node_local - Rack_local - Any - Unused

Summary

	Cluster utilization	Execution time	Locality tweaking	Miscellaneous
Sequential	Poor	Long	Hard	Default behavior in Spark-shell
Parallel {FIFO}	**Good**	**Short**	Hard	Default behavior in Notebooks
Parallel {FAIR}	**Good**	**Short**	**Easy**	

Standalone applications

Stream, Batch, Adhoc query, submitted from multiple apps



Continuous application

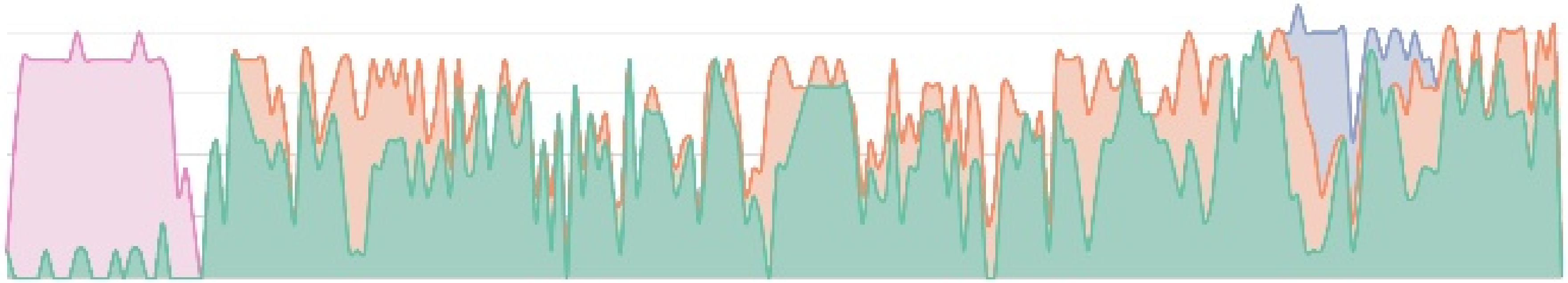
Stream, Batch, Adhoc query, parallel submitted in one app

Read data 🟪

Model Execution 🟧

Stream Process 🟩

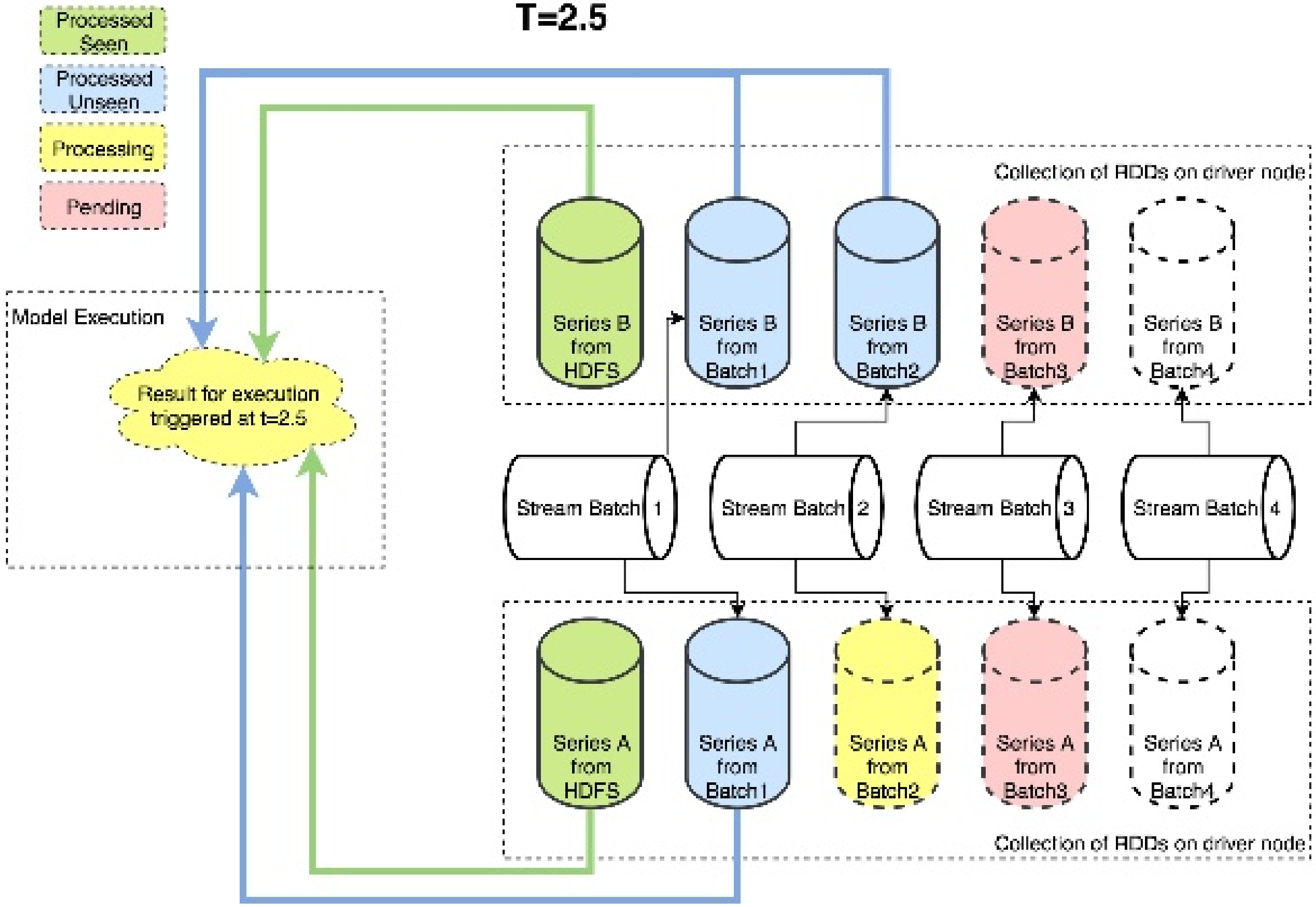
Reporting 🟦



Continuous Application

	Multiple standalone applications	Continuous application
Sharing context	Share file content using database, file system, network, etc	Simply pass RDD reference around
Resource allocation	Static; configure more cores than typical load to handle peak traffic	Dynamic; less important tasks yield CPU critical tasks
Job scheduling	Crontab, oozie, airflow… — all approaches leads to spark-submit, a typically 20s - 120s overhead	Spark-submit once and only once

T=2.5

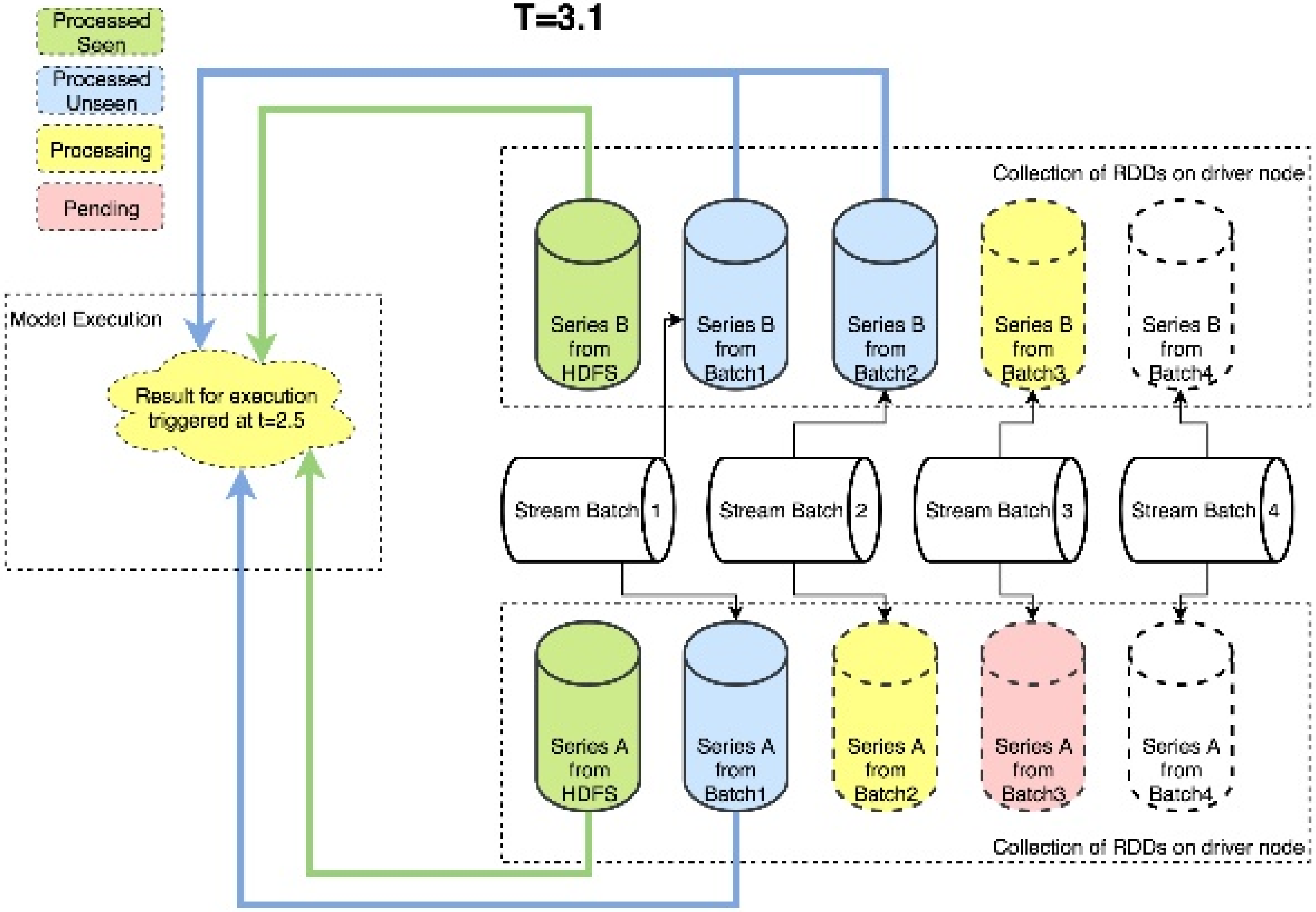


At t=2.5, SeriesA and B finished processing stream input at t=1.0

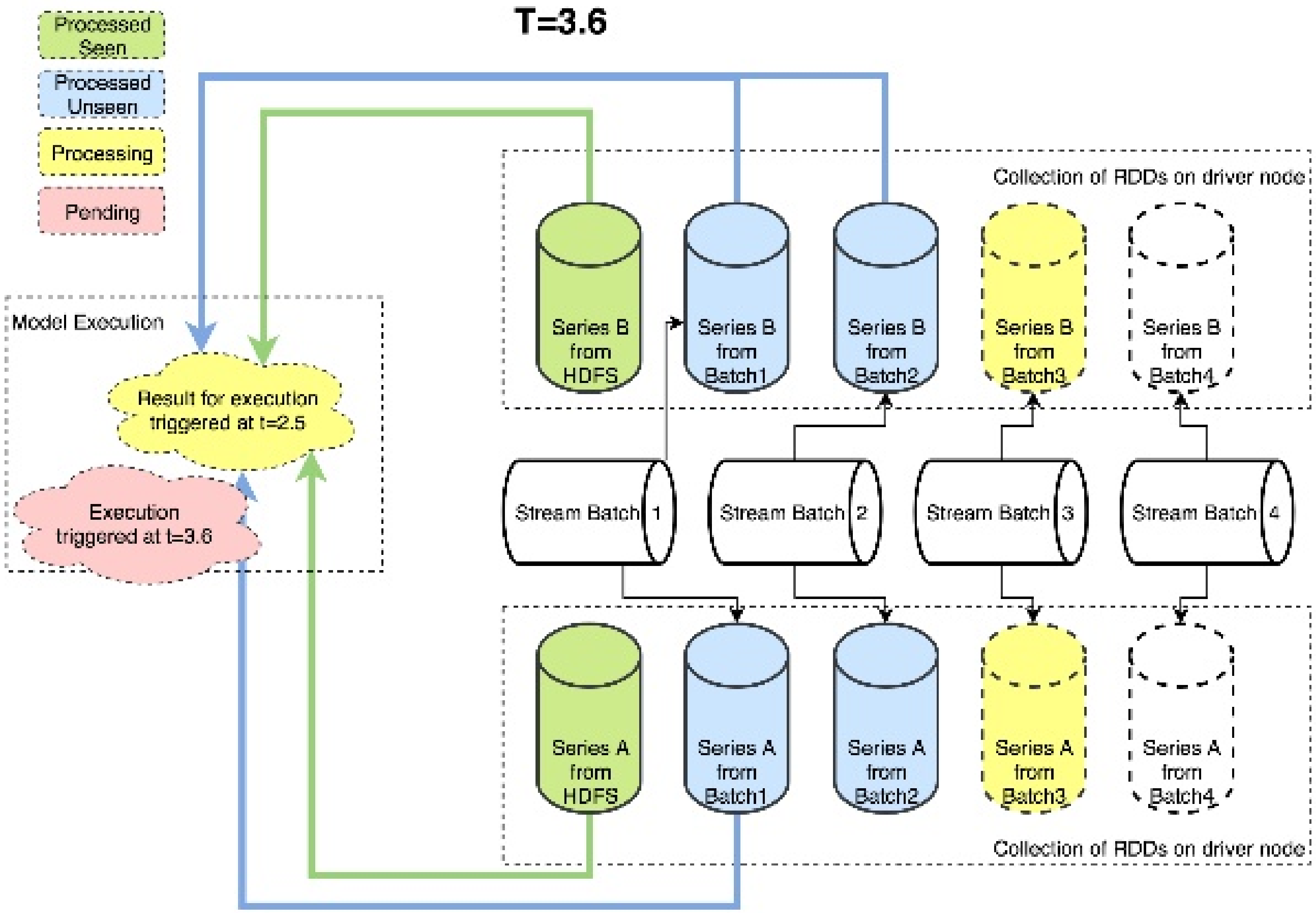
A round of model execution had been triggered at t=2.5

Model was triggered using all available data at that moment

T=3.1



At t=3.1, app received stream input at t=3.0
SeriesA was still processing input at t=2.0
SeriesB started processing the new input.

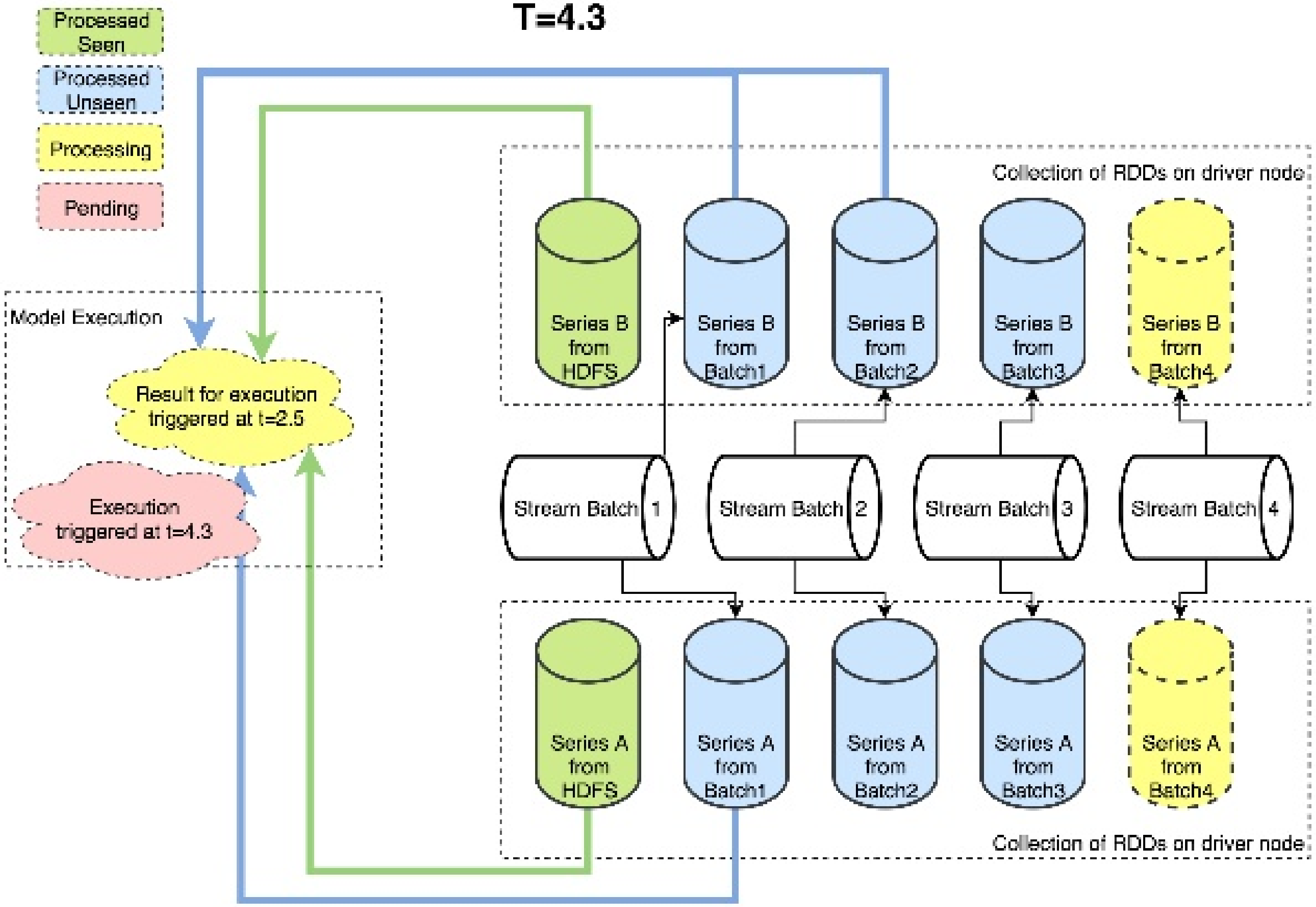


At t=3.6, SeriesA finished processing stream input at t=2.0

SeriesA started processing stream input at t=3.0

Model had a pending execution triggered at t=3.6

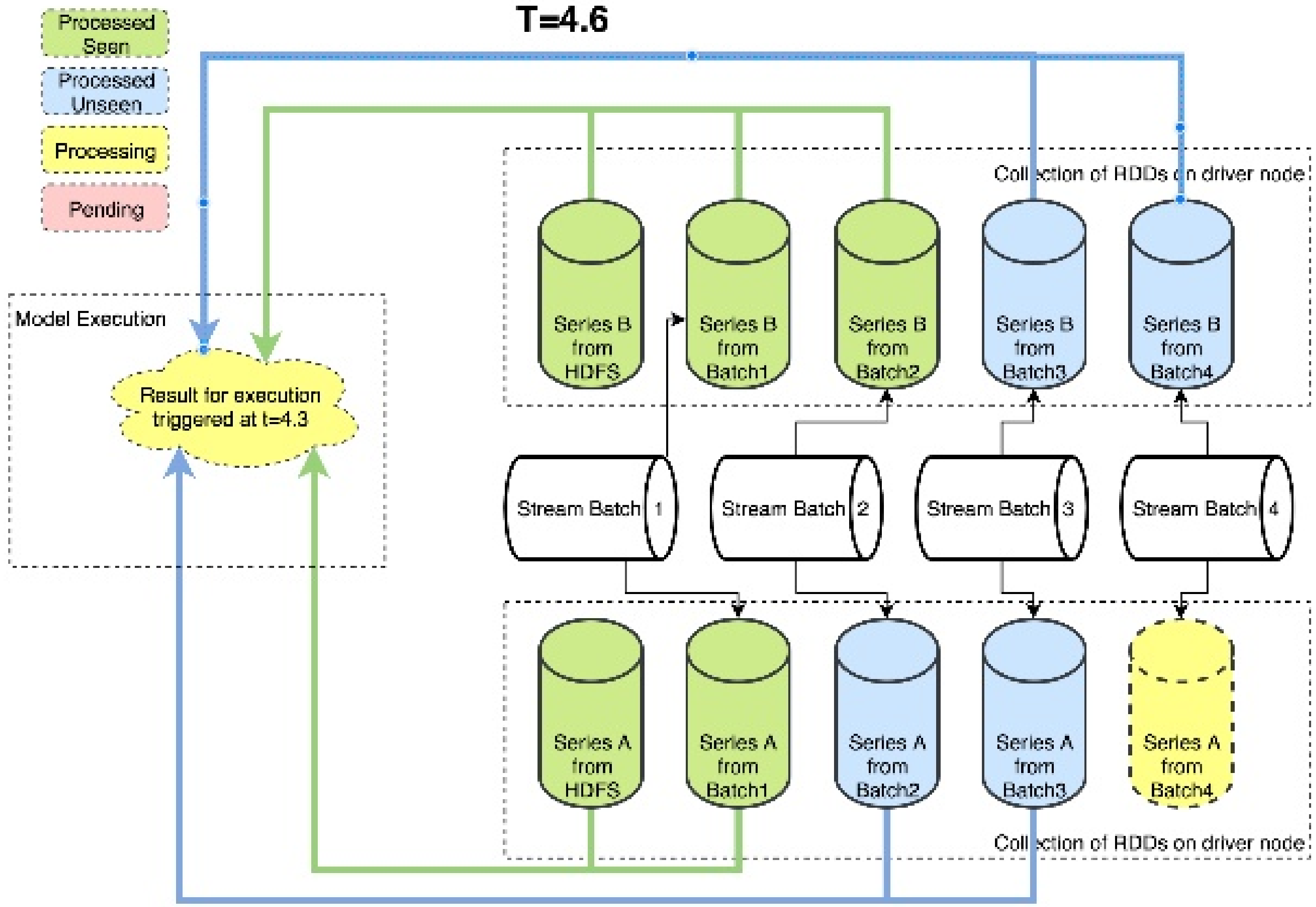
T=4.3



At t=4.3, Series A and B finished processing stream input at t=3.0

Model had a pending execution triggered at t=4.3

Model’s execution at t=3.6 was cancelled



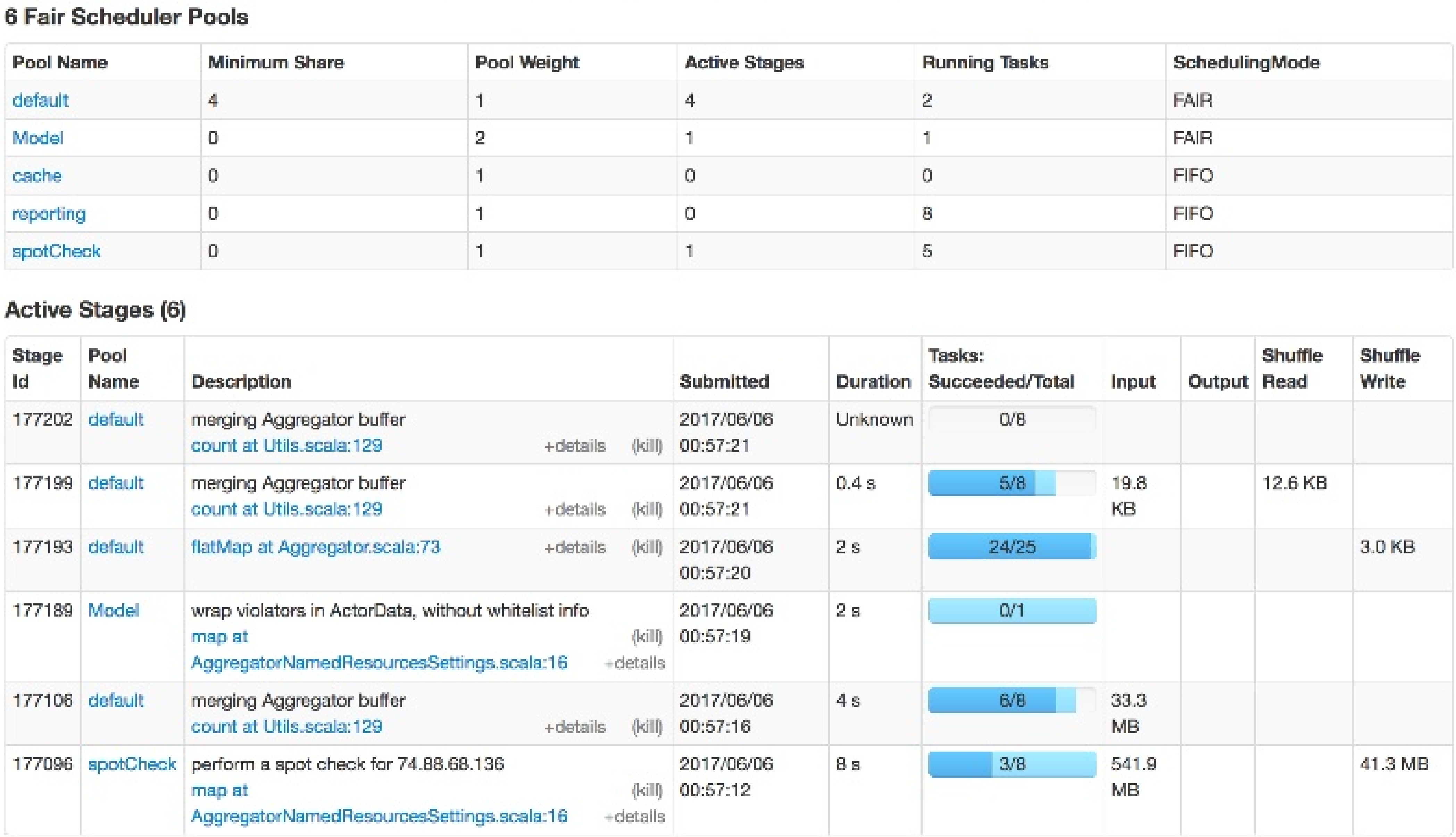
At t=4.6, model finished it's execution at t=2.5

Series B finished processing stream input at t=4.0

Model started execution that was triggered at t=4.3

Model used data that was available at t=4.6 as input

Example:
4 models executing in parallel from a 16 cores app



6 Fair Scheduler Pools

Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode
default	4	1	4	2	FAIR
Model	0	2	1	1	FAIR
cache	0	1	0	0	FIFO
reporting	0	1	0	8	FIFO
spotCheck	0	1	1	5	FIFO

Active Stages (6)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
177202	default	merging Aggregator buffer count at Utils.scala:129 +details (kill)	2017/06/06 00:57:21	Unknown	0/8				
177199	default	merging Aggregator buffer count at Utils.scala:129 +details (kill)	2017/06/06 00:57:21	0.4 s	5/8	19.8 KB		12.6 KB	
177193	default	flatMap at Aggregator.scala:73 +details (kill)	2017/06/06 00:57:20	2 s	24/25				3.0 KB
177189	Model	wrap violators in ActorData, without whitelist info map at (kill) AggregatorNamedResourcesSettings.scala:16 +details	2017/06/06 00:57:19	2 s	0/1				
177106	default	merging Aggregator buffer count at Utils.scala:129 +details (kill)	2017/06/06 00:57:16	4 s	6/8	33.3 MB			
177096	spotCheck	perform a spot check for 74.88.68.136 map at (kill) AggregatorNamedResourcesSettings.scala:16 +details	2017/06/06 00:57:12	8 s	3/8	541.9 MB			41.3 MB

Practices for Continuous Application

Decouple stream processing from batch processing

- Batch interval is merely your minimum resolution

Emphasis tuning for streaming part

- Assign to a scheduler pool with minimum core guarantee

Execute only the latest batch invocation

- Assign to a scheduler pool with high weight

Reporting and query onsite

- Assign to a scheduler pool with low weight

Summary

Coding

- Share data by passing RDD/DStream/DF/DS
- Always launch jobs in a separate thread
- No complex logic in the streaming operation
- Push batch job invocation into a queue
- Execute only the latest batch job

App submission

- Turn on FAIR scheduler mode
- Provide fairscheduler.xml
- Turn off stream back pressure for Streaming app
- Turn off dynamic allocation for Streaming app
- Turn on dynamic allocation for long-lived batch app

Job bootstrapping

- sc.setJobGroup(“group”, “description”)
- sc.setLocalProperty("spark.scheduler.pool", “pool”)
- rdd.setName(“my data at t=0”).persist()

Packaging

- Multiple logic, one repo, one jar
- Batch app can be long-lived as well
- Replace crontab with continuous app + REST

Tuning

- Understand your job’s opportunity cost
- Tweak spark.locality.wait parameters
- Config cores that yields acceptable SLA with good resource utilization

Thank You.

Robxue Xue, Software Engineer, Groupon

rxue@groupon.com | @roboxue

Tool used in this deck: groupon/sparklint

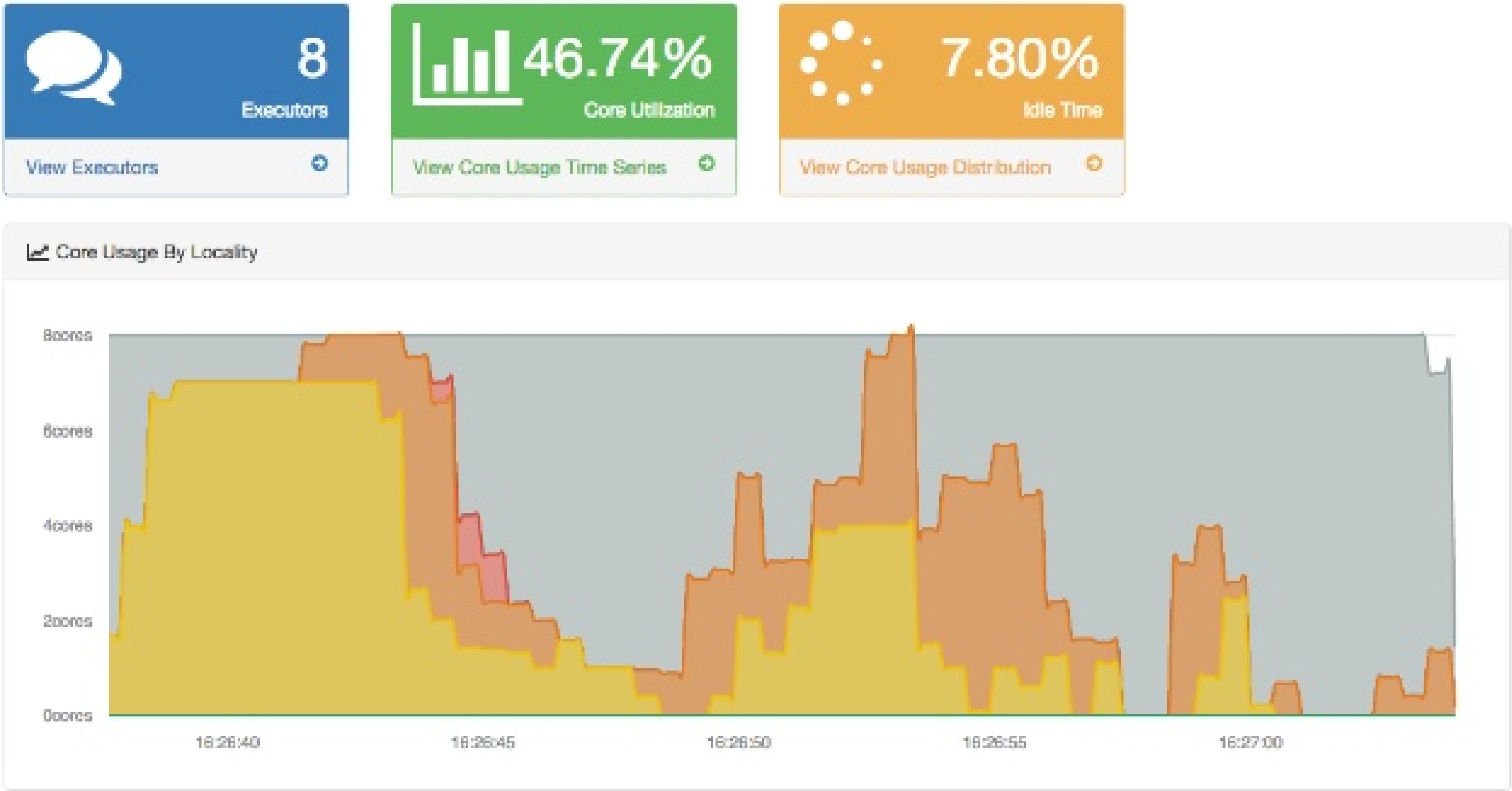
Open sourced on github since Spark Summit EU 2016

Q & A

Why do you need a scheduler - 3
Parallelly submitting 24 file reading jobs - FIFO(4)

- 8 cores
- ~~22~~ 27 seconds
- 8 * 27 * 46.74% = ~~99.68~~100.88 cpu seconds
- ~~Bad~~Improved locality

```
implicit val ec =
 ExecutionContext.fromExecutor(
  new ForkJoinPool(4))
Range(0, 24).foreach(i=> Future{
 sc.textFile(s”file-$i”).count()
})
```



8 Executors — View Executors

46.74% Core Utilization — View Core Usage Time Series

7.80% Idle Time — View Core Usage Distribution

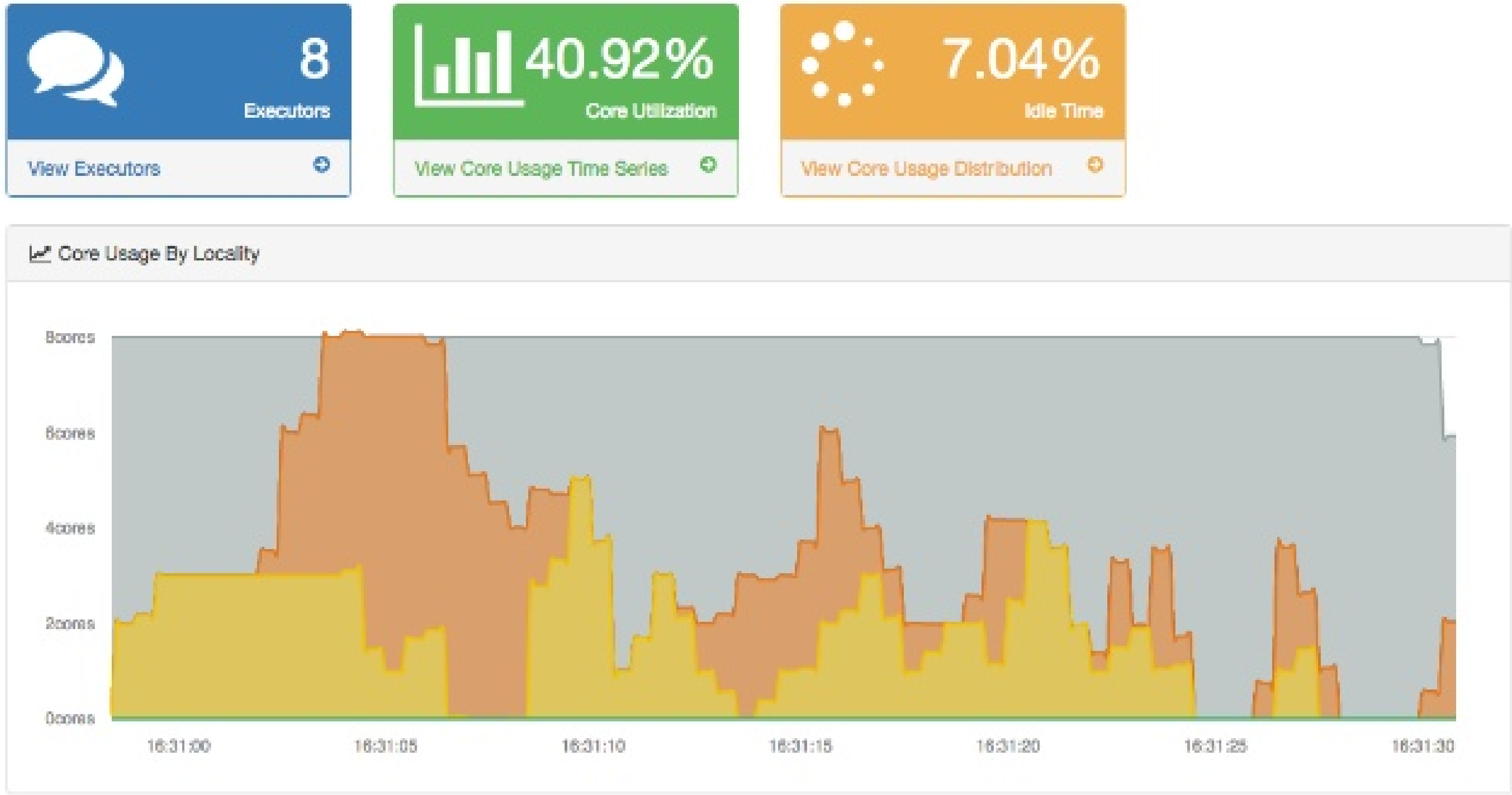
Core Usage By Locality

Node_local - Rack_local - Any - Unused

Why do you need a scheduler - 4
Parallelly submitting 24 file reading jobs - FAIR(4)

- 8 cores
- ~~22~~ 32 seconds
- 8 * 32 * 40.92% = ~~99.68~~104.72 cpu seconds
- ~~Bad~~Great locality

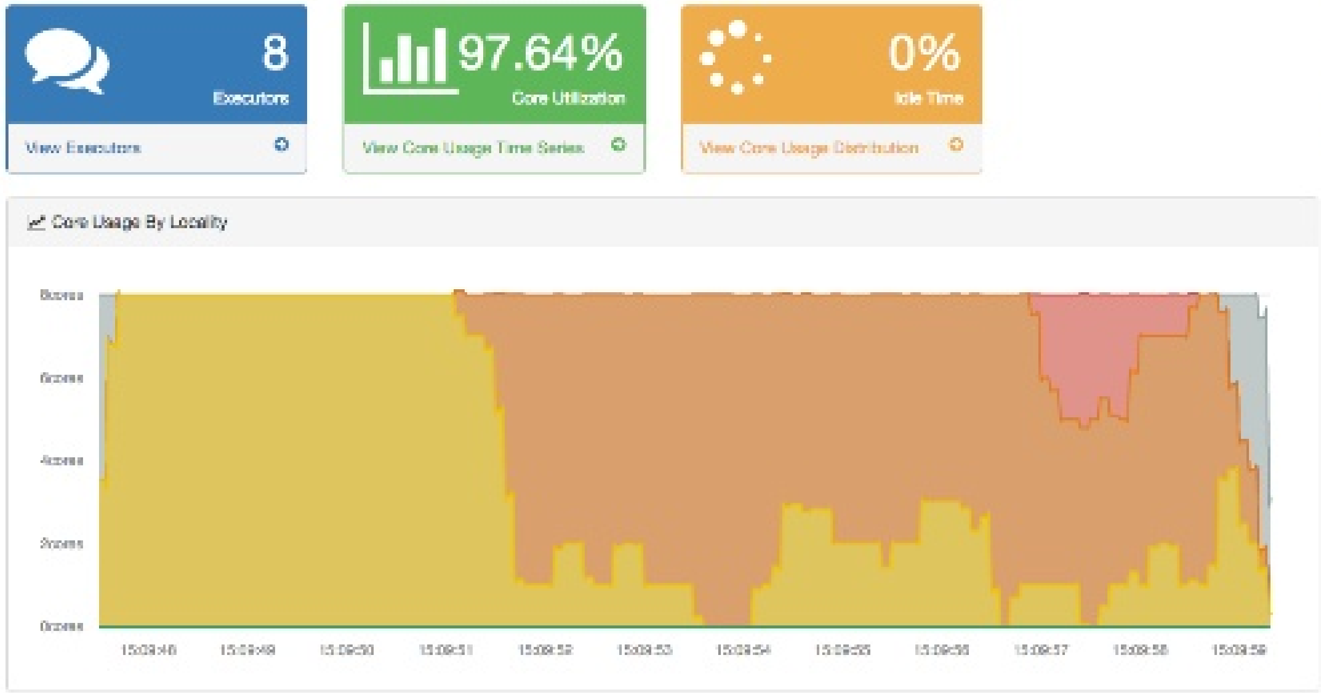
```
// --conf spark.scheduler.mode=FAIR
implicit val ec =
 ExecutionContext.fromExecutor(
   new ForkJoinPool(4))
Range(0, 24).foreach(i=> Future{
 sc.textFile(s"file-$i").count()
})
```



Node_local - Rack_local - Any - Unused

Back up: locality settings matters

spark.scheduler.mode=FAIR
spark.locality.wait=500ms



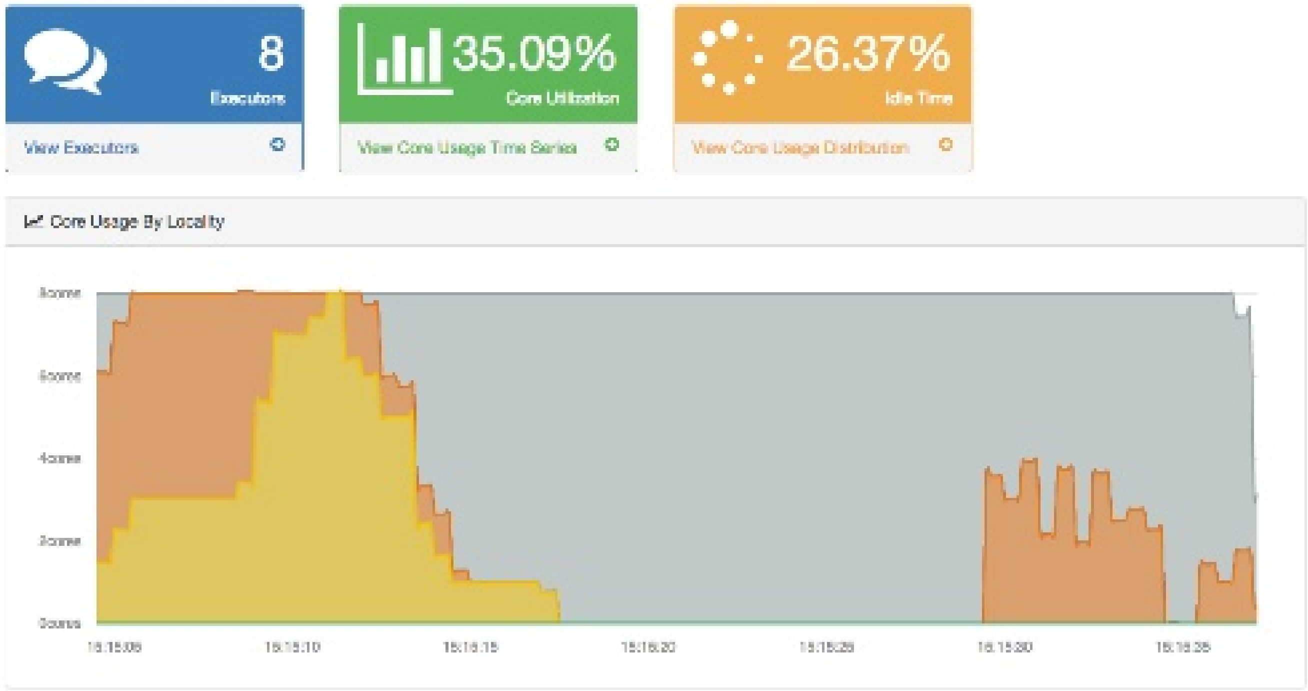
8 Executors — View Executors

97.64% Core Utilization — View Core Usage Time Series

0% Idle Time — View Core Usage Distribution

Core Usage By Locality

spark.scheduler.mode=FAIR
spark.locality.wait=20s



8 Executors — View Executors

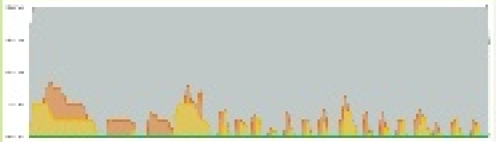
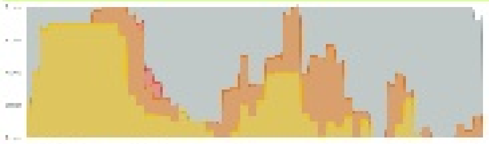
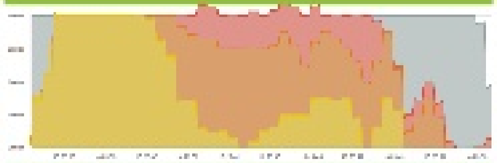
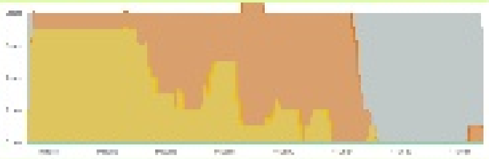
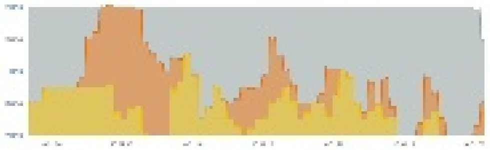
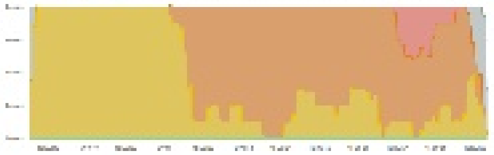
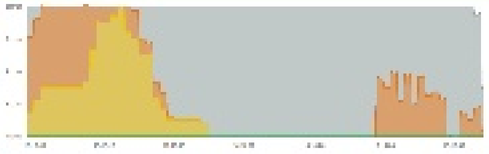
35.09% Core Utilization — View Core Usage Time Series

26.37% Idle Time — View Core Usage Distribution

Core Usage By Locality

Node_local - Rack_local - Any - Unused

Backup:
Scheduler summary

	Sequential job submit	Parallel job submit Under Parallelized	Parallel job submit Perfect Parallelized	Parallel job submit Over Parallelized / Poor locality settings
				
FIFO scheduler	Under-utilized cluster Good locality Low core usage Long execution time	Under-utilized cluster Poor locality High core usage Short execution time	Well-utilized cluster Poor locality High core usage Short execution time	Well-utilized cluster Poor locality High core usage Long execution time
FAIR scheduler	N/A	Under-utilized cluster Good locality Low core usage Short execution time	Well-utilized cluster Good locality Low core usage Short execution time	
				

Node_local - Rack_local - Any - Unused

Backup:
What’s wrong with Dynamic Allocation in streaming

… if you are analyzing time series data / “tensor” …

- Streaming app, always up. Workloads comes periodically and sequentially.
- Core usage graph has a saw-tooth pattern
  - Less likely to return executors, if your batch interval < executorIdleTimeout
- Dynamic allocation is off if executor has cache on it
  - “by default executors containing cached data are never removed”
- Dynamic allocation == “Resource blackhole” ++ “poor utilization”