Re-Architecting Apache Spark for Performance Understandability

Kay Ousterhout

Joint work with Christopher Canel, Max Wolffe, Sylvia Ratnasamy, Scott Shenker

About Me

PhD candidate at UC Berkeley

Thesis work on performance of large-scale distributed systems

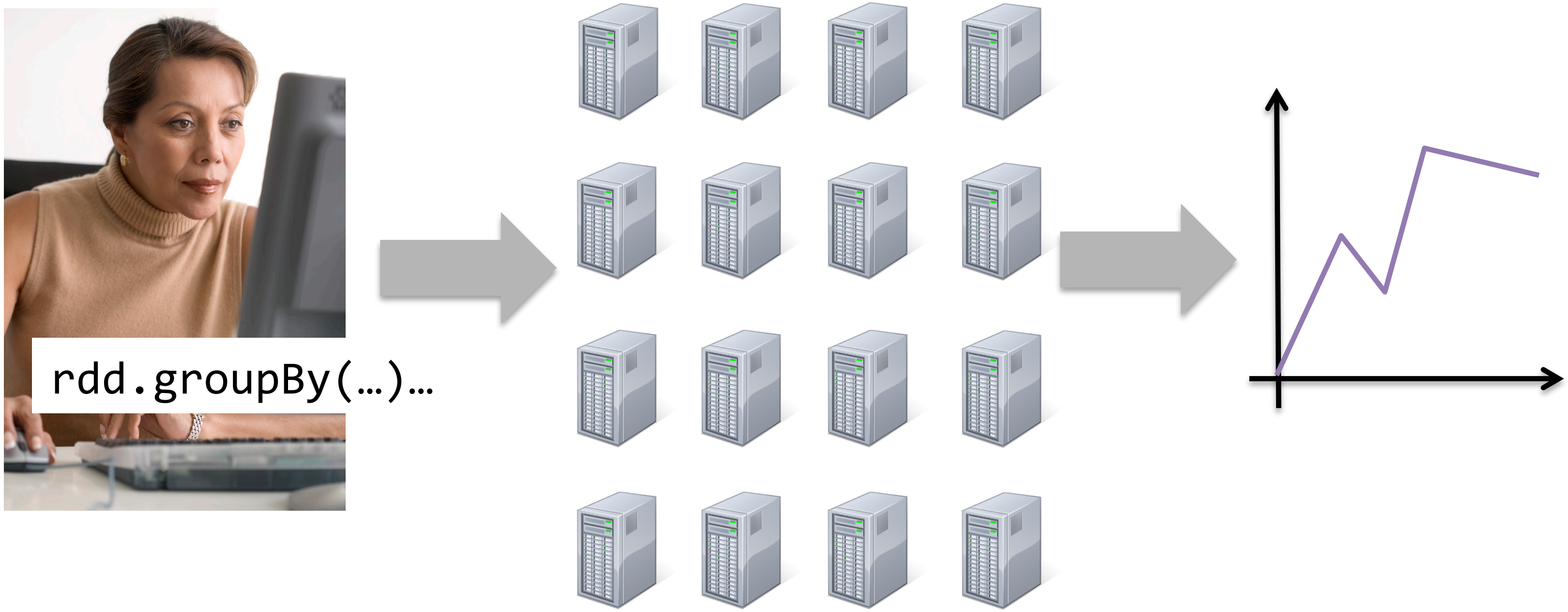
Apache Spark PMC member

About this talk

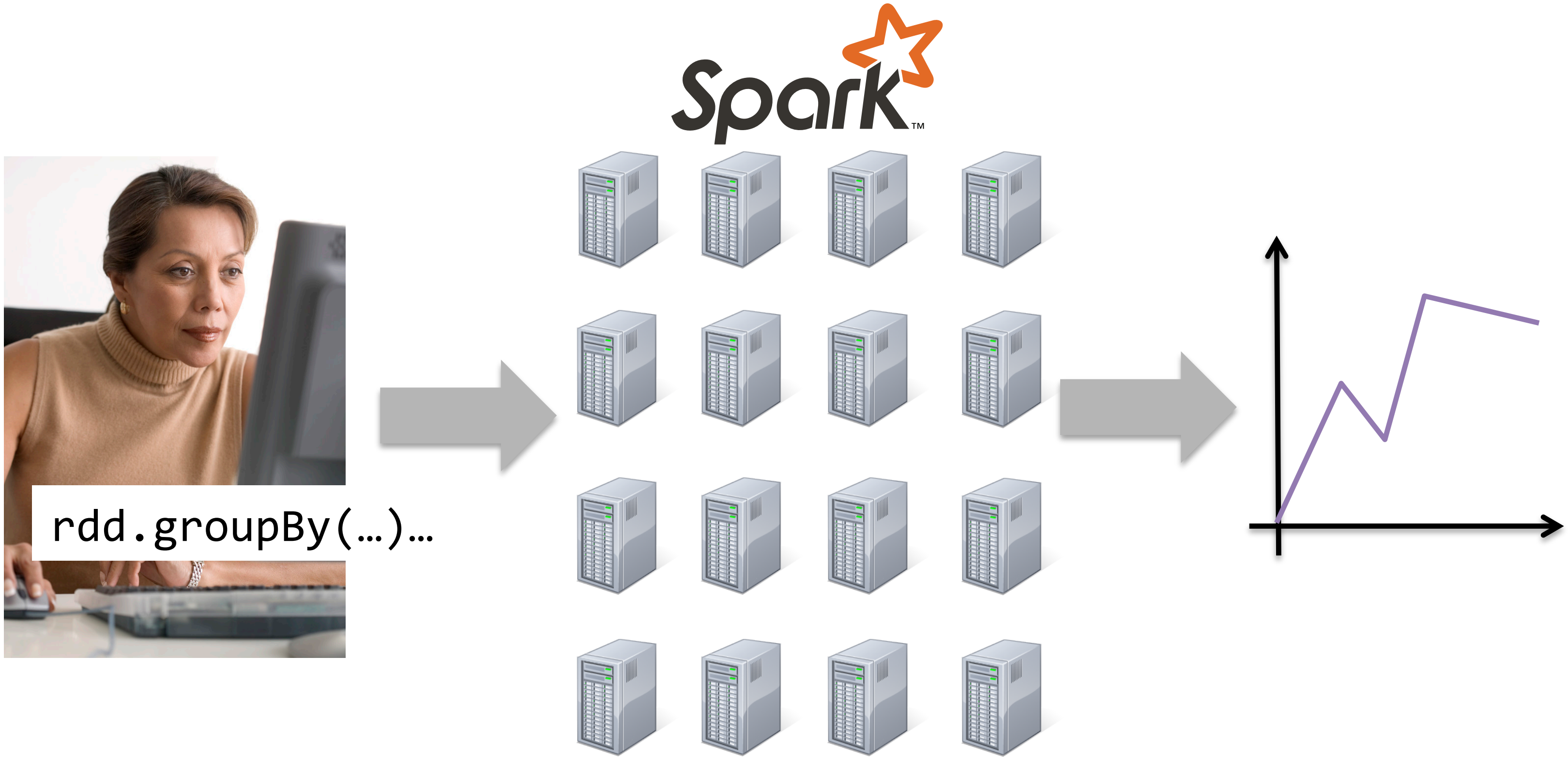
Future architecture for systems like Spark

Implementation is API-compatible with Spark

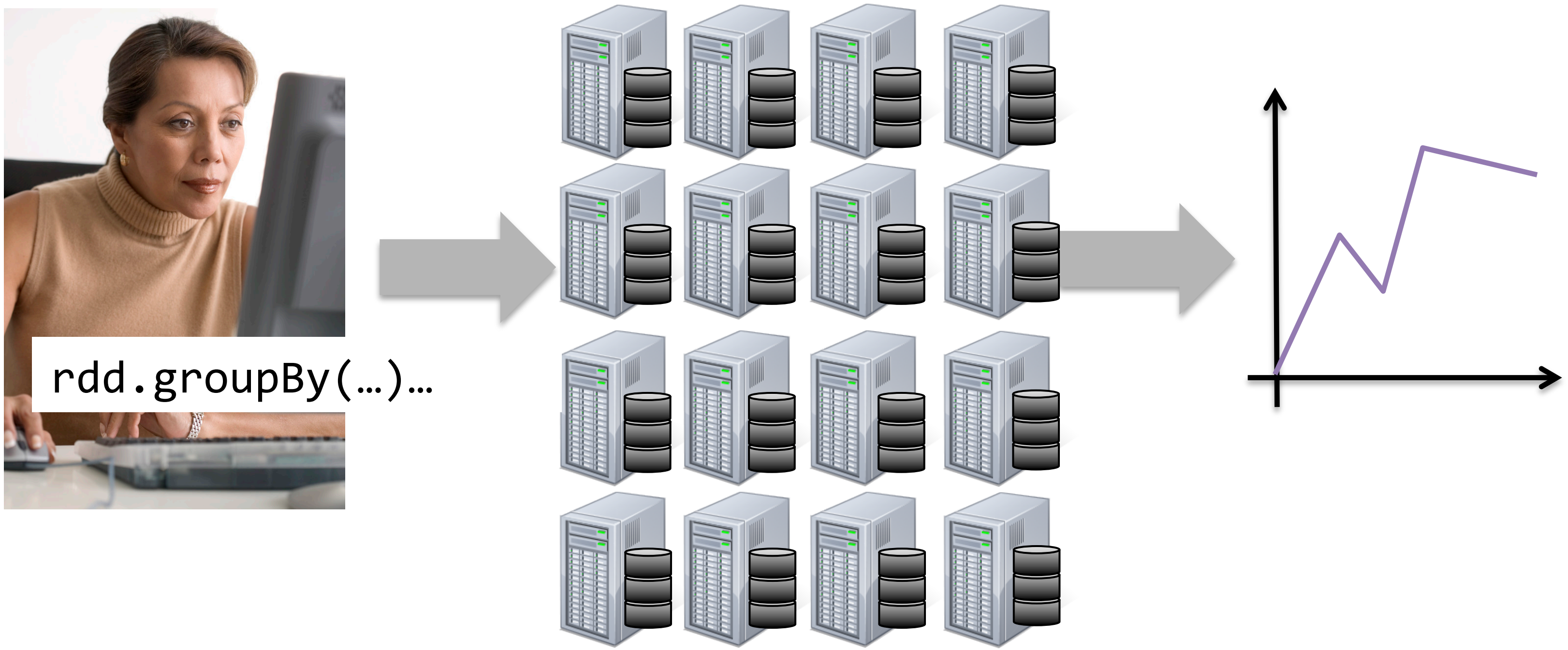
Major change to Spark’s internals (~20K lines of code)



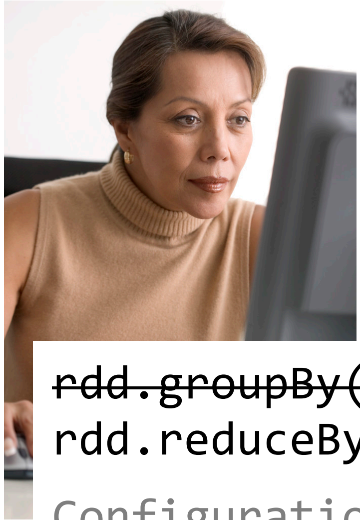
Spark cluster is a black box, runs the job fast



Idealistic view: Spark cluster is a black box, runs the job fast

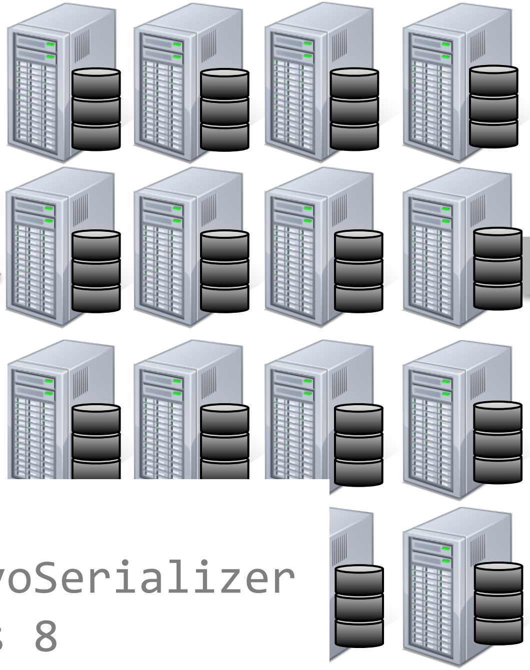


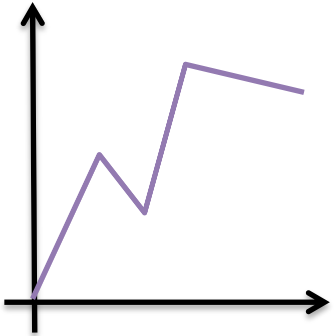
Idealistic view: Spark cluster is a black box, runs the job fast



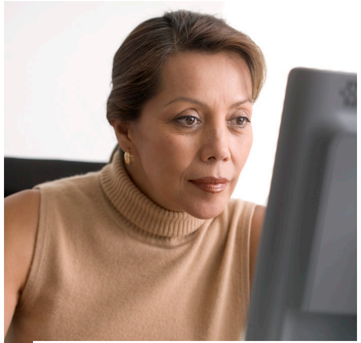
~~rdd.groupBy(…)…~~
rdd.reduceByKey(…)…

Configuration:
spark.serializer KryoSerializer
spark.executor.cores 8



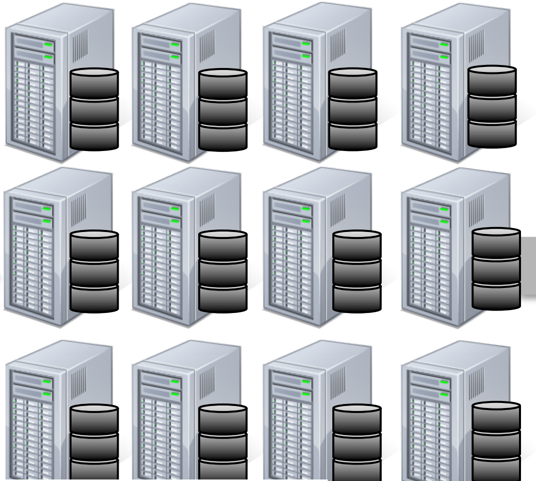


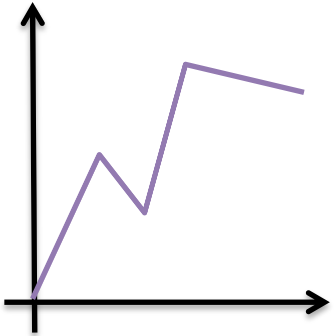
Idealistic view: Spark cluster is a black box, runs the job fast



```
~~rdd.groupBy(…)…~~
rdd.reduceByKey(…)…
```

```
Configuration:
spark.serializer KryoSerializer
spark.executor.cores 8
```



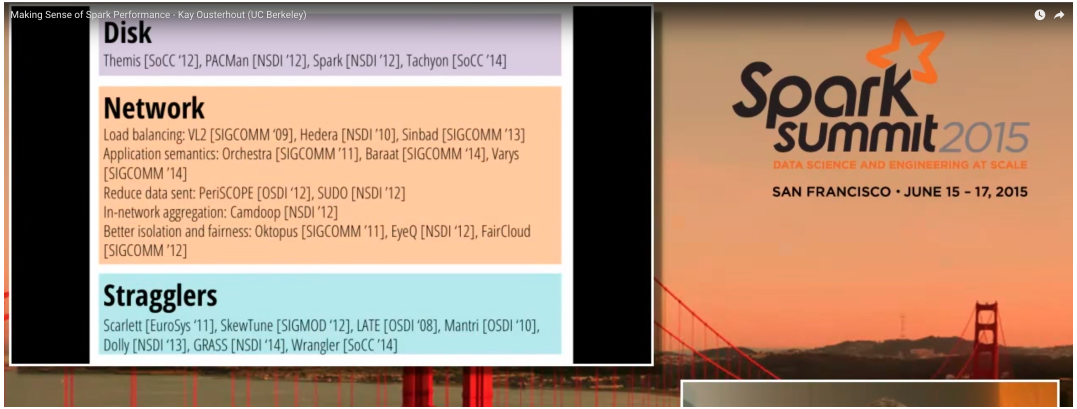


Realistic view: user uses performance characteristics to tune job, configuration, hardware, etc.

Users need to be able to reason about performance

Realistic view: user uses performance characteristics to tune job, configuration, hardware, etc.

Reasoning about Spark Performance

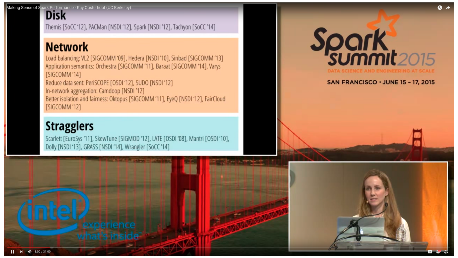


Widely accepted that network and disk I/O are bottlenecks

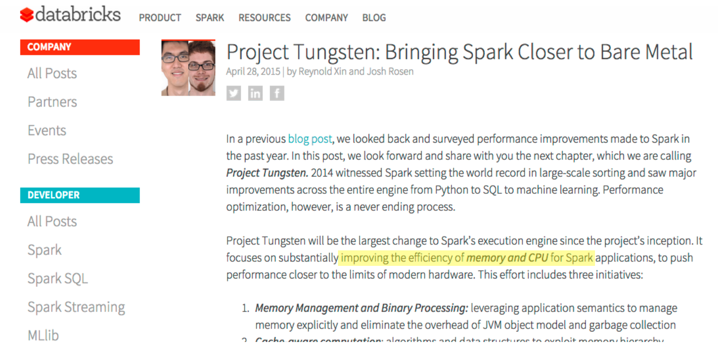
CPU (not I/O) typically the bottleneck

network optimizations can improve job completion time by at most 2%

Reasoning about Spark Performance



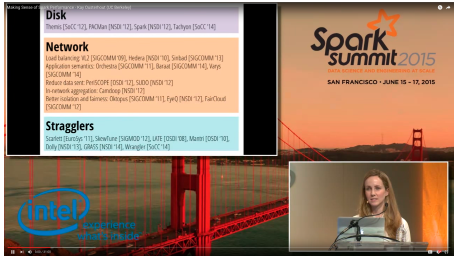
Spark Summit 2015:
CPU (not I/O) often
the bottleneck



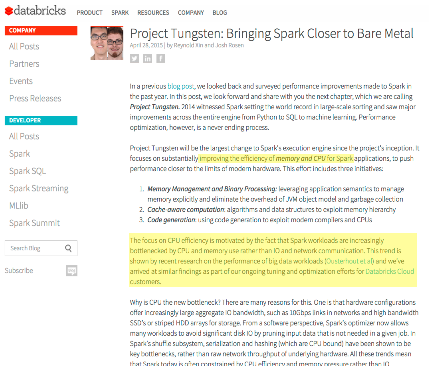
Project Tungsten:

initiative to optimize Spark's CPU use, driven in part by our measurements

Reasoning about Spark Performance



Spark Summit 2015:
CPU (not I/O) often the bottleneck



databricks PRODUCT SPARK RESOURCES COMPANY BLOG

COMPANY
All Posts
Partners
Events
Press Releases

DEVELOPER
All Posts
Spark
Spark SQL
Spark Streaming
MLlib
Spark Summit

Search Blog

Subscribe

Project Tungsten: Bringing Spark Closer to Bare Metal

April 28, 2015 | by Reynold Xin and Josh Rosen

In a previous blog post, we looked back and surveyed performance improvements made to Spark in the past year. In this post, we look forward and share with you the next chapter, which we are calling *Project Tungsten.* 2014 witnessed Spark setting the world record in large-scale sorting and saw major improvements across the entire engine from Python to SQL to machine learning. Performance optimization, however, is a never ending process.

Project Tungsten will be the largest change to Spark's execution engine since the project's inception. It focuses on substantially improving the efficiency of ***memory and CPU*** for Spark applications, to push performance closer to the limits of modern hardware. This effort includes three initiatives:

1. ***Memory Management and Binary Processing:*** leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection
2. ***Cache-aware computation***: algorithms and data structures to exploit memory hierarchy
3. ***Code generation***: using code generation to exploit modern compilers and CPUs

The focus on CPU efficiency is motivated by the fact that Spark workloads are increasingly bottlenecked by CPU and memory use rather than IO and network communication. This trend is shown by recent research on the performance of big data workloads (Ousterhout et al) and we've arrived at similar findings as part of our ongoing tuning and optimization efforts for Databricks Cloud customers.

Why is CPU the new bottleneck? There are many reasons for this. One is that hardware configurations offer increasingly large aggregate IO bandwidth, such as 10Gbps links in networks and high bandwidth SSD's or striped HDD arrays for storage. From a software perspective, Spark's optimizer now allows many workloads to avoid significant disk IO by pruning input data that is not needed in a given job. In Spark's shuffle subsystem, serialization and hashing (which are CPU bound) have been shown to be key bottlenecks, rather than raw network throughput of underlying hardware. All these trends mean that Spark today is often constrained by CPU efficiency and memory pressure rather than IO

Project Tungsten:
initiative to optimize Spark's CPU use

Spark 2.0:
Some evidence that I/O is again the bottleneck [HotCloud '16]

Users need to understand performance to extract the best runtimes

Reasoning about performance is currently difficult

Software and hardware are constantly evolving, so performance is always in flux

Vision: jobs report high-level performance metrics

Spark Monotasks | Jobs | Stages | Storage | Environment | Executors

Details for Stage 17

Total task time across all tasks: 13 min
Shuffle read: 2.5 GB / 31589120

Performance Information

Bottleneck: Disk (if disk bandwidth were increased by 23% or more, network would become the bottleneck)

Vision: jobs report high-level performance metrics

Spark Monotasks | Jobs | Stages | Storage | Environment | Executors

Details for Stage 17

Total task time across all tasks: 13 min
Shuffle read: 2.5 GB / 31589120

Performance Information

Bottleneck: Disk (if disk bandwidth were increased by 23% or more, network would become the bottleneck)
Non-bottlenecks: Network (could reduce network bandwidth by up to 30% slower without impacting runtime), CPU (could increase CPU time by up to 2x without impacting runtime)

Vision: jobs report high-level performance metrics

Monotasks Jobs Stages Storage Environment Executors

Details for Stage 17

Total task time across all tasks: 13 min
Shuffle read: 2.5 GB / 31589120

Performance Information

Bottleneck: Disk (if disk bandwidth were increased by 23% or more, network would become the bottleneck)
Non-bottlenecks: Network (could reduce network bandwidth by up to 30% slower without impacting runtime), CPU (could increase CPU time by up to 2x without impacting runtime)
Benefit of caching: Storing input in-memory would reduce job completion time by 42%

Vision: jobs report high-level performance metrics

Spark Monotasks | Jobs | Stages | Storage | Environment | Executors

Details for Stage 17

Total task time across all tasks: 13 min
Shuffle read: 2.5 GB / 31589120

Performance Information

Bottleneck: Disk (if disk bandwidth were increased by 23% or more, network would become the bottleneck)
Non-bottlenecks: Network (could reduce network bandwidth by up to 30% slower without impacting runtime), CPU (could increase CPU time by up to 2x without impacting runtime)
Benefit of caching: Storing input in-memory would reduce job completion time by 42%
Job Runtime Predictor: (enter in properties of different cluster to estimate job's runtime)

Number of machines:
CPU cores per machine:
Network bandwidth per machine: Gbps
I/O bandwidth per machine: MB/s

Calculate new runtime

Predicted new job runtime:

How can we achieve this vision?

Spark overview

Reasoning about Spark's performance: why it's hard

New architecture: monotasks

Reasoning about monotasks performance: why it's easy

Monotasks in action (results)

Example Spark Job:

Read remote data
Filter records
Write result to disk

Task 1:
Read and filter **block 1**
write result to disk

Task 2:
Read and filter **block 2**
write result to disk

Task 3:
Read and filter **block 3**
write result to disk

Task 4:
Read and filter **block 4**
write result to disk

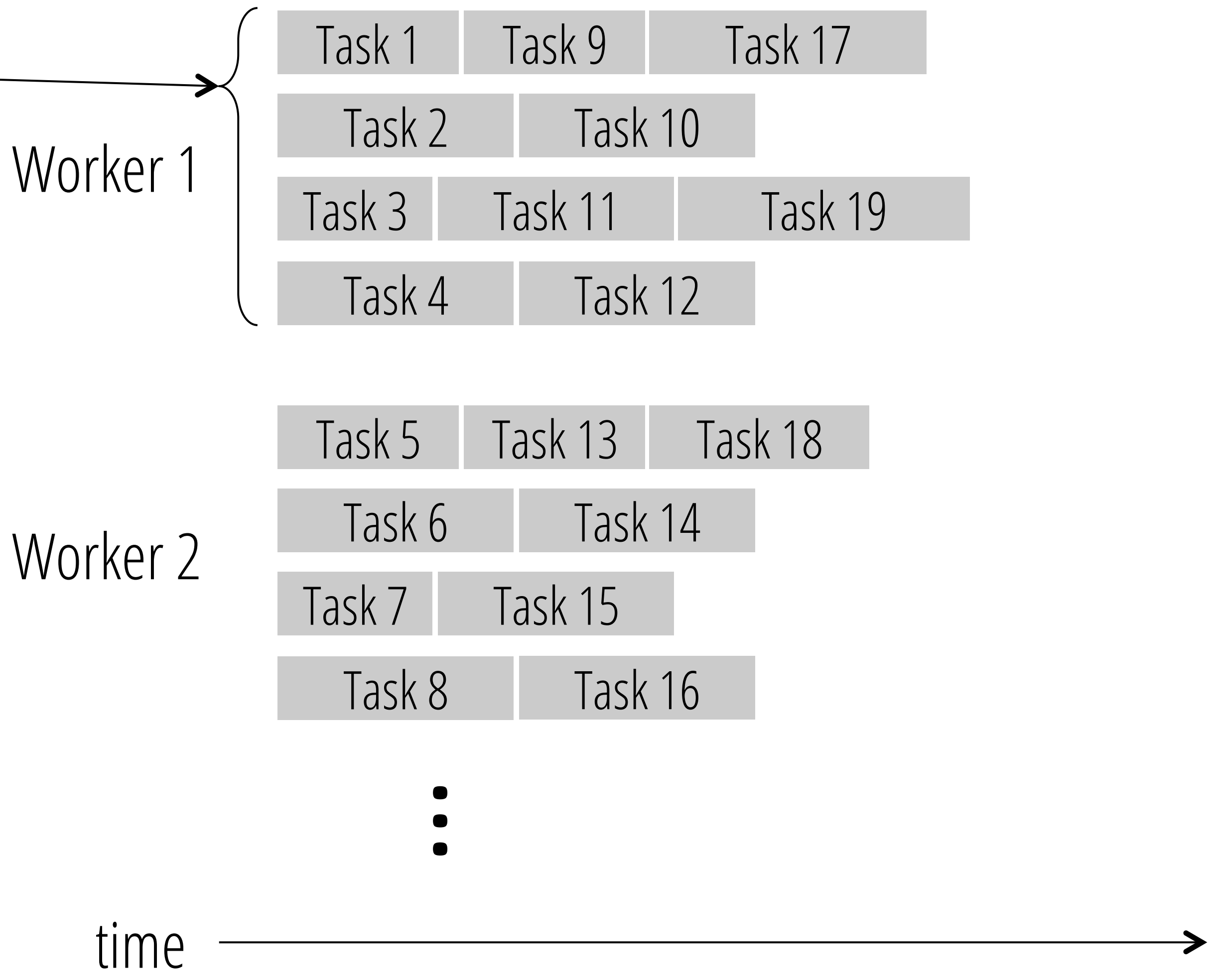
Task 5:
Read and filter **block 5**
write result to disk

Task 6:
Read and filter **block 6**
write result to disk

⋮

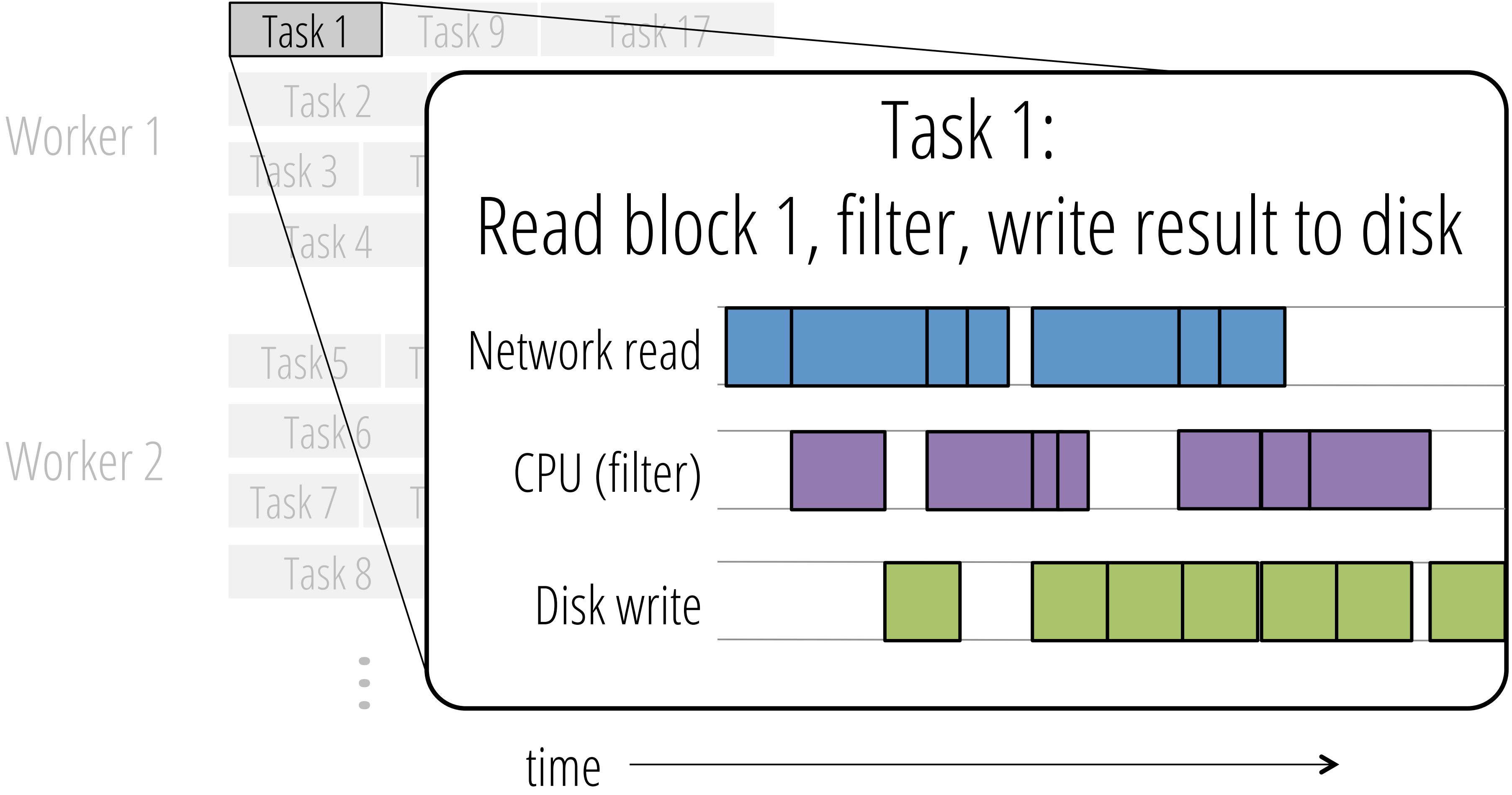
Task n:
Read and filter **block n**
write result to disk

Fixed number of “slots”



Example Spark Job:

Read remote data
Filter records
Write result to disk



Task 1
Task 9
Task 17
Worker 1
Task 2
Task 3
Task 4
Worker 2
Task 5
Task 6
Task 7
Task 8
Task 1:
Read block 1, filter, write result to disk
Network read
CPU (filter)
Disk write
time

How can we achieve this vision?

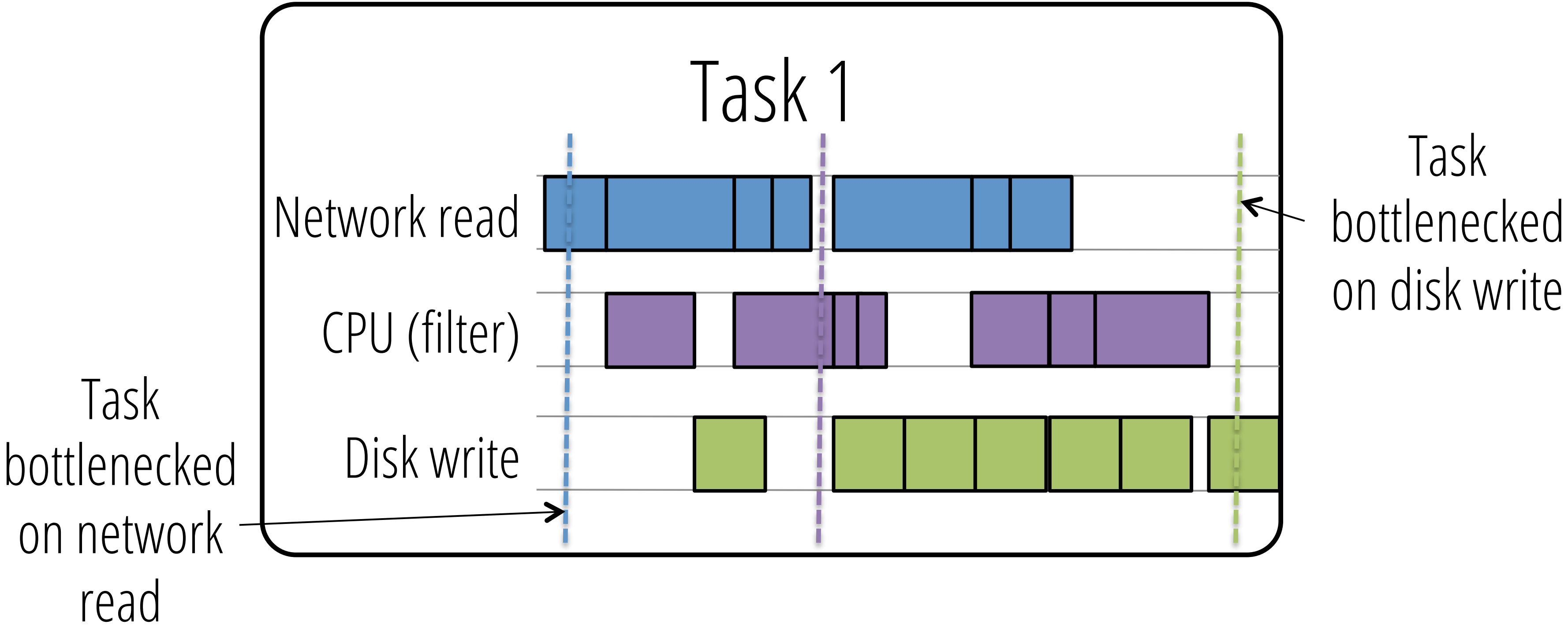
Spark overview

Reasoning about Spark’s performance: why it’s hard

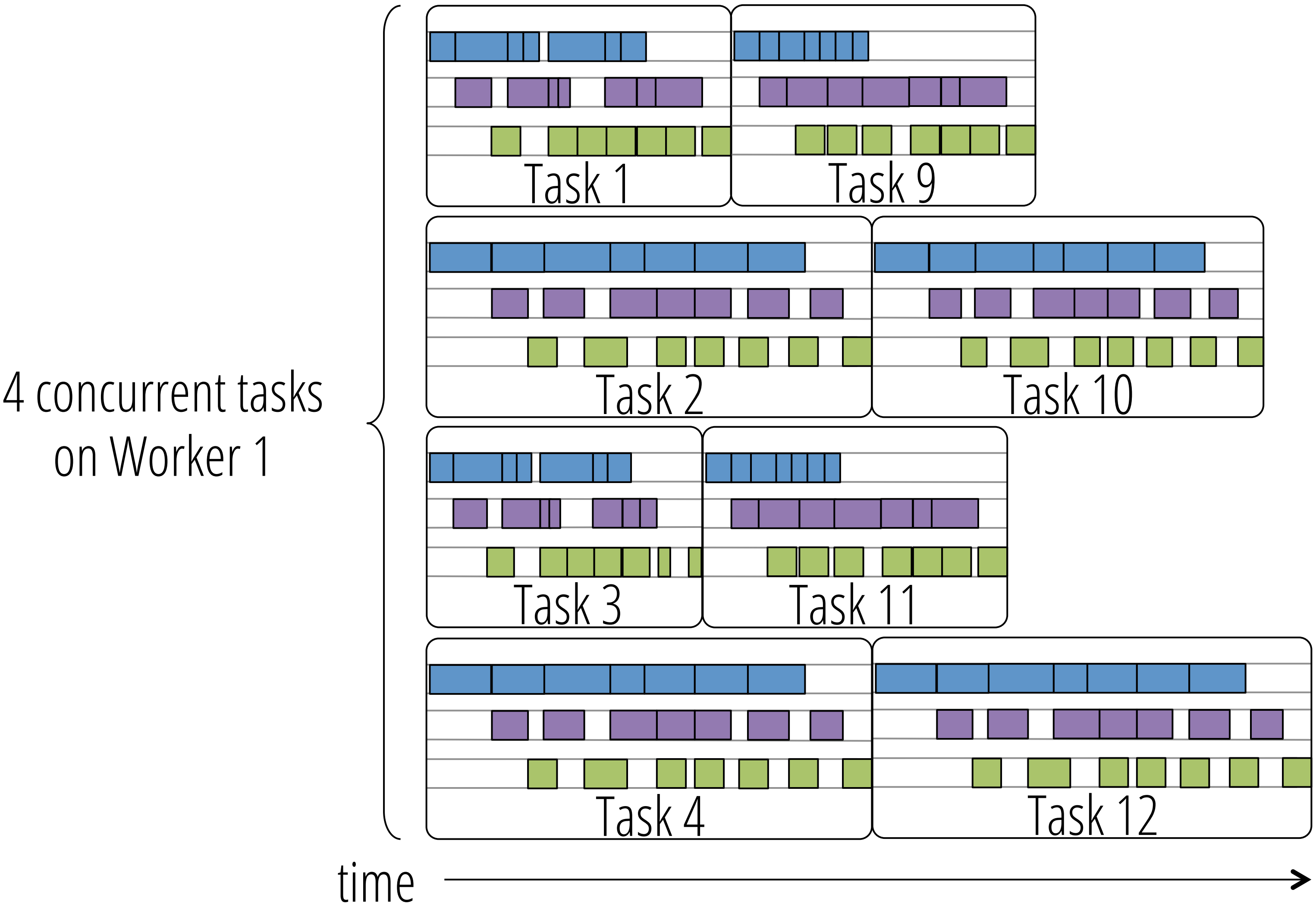
New architecture: monotasks

Reasoning about monotasks performance: why it’s easy

Monotasks in action (results)

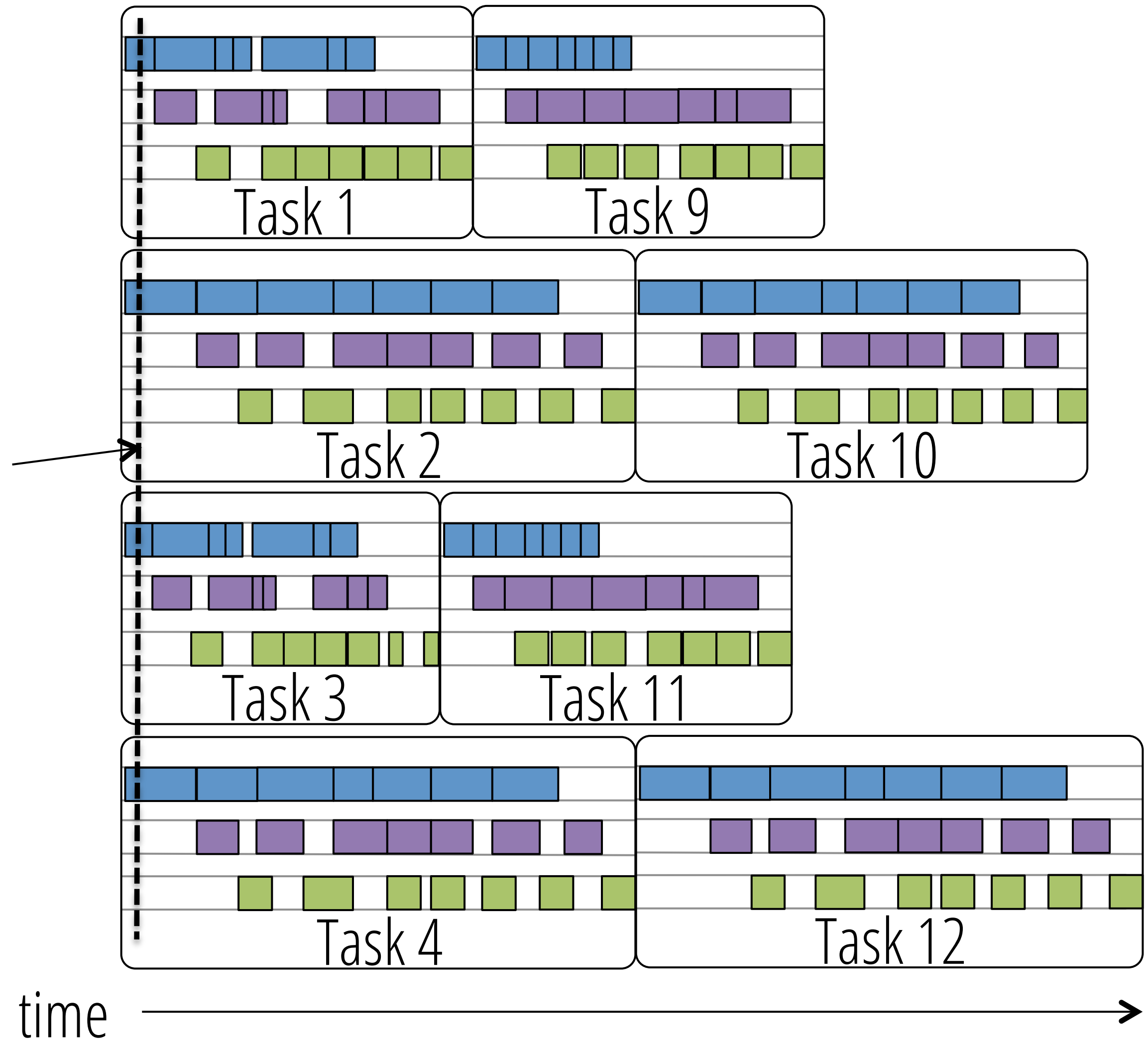


Challenge 1: Task pipelines multiple resources, bottlenecked on different resources at different times


4 concurrent tasks
on Worker 1
Task 1
Task 9
Task 2
Task 10
Task 3
Task 11
Task 4
Task 12
time

Challenge 2:

Concurrent tasks may contend for the same resource (e.g., network)

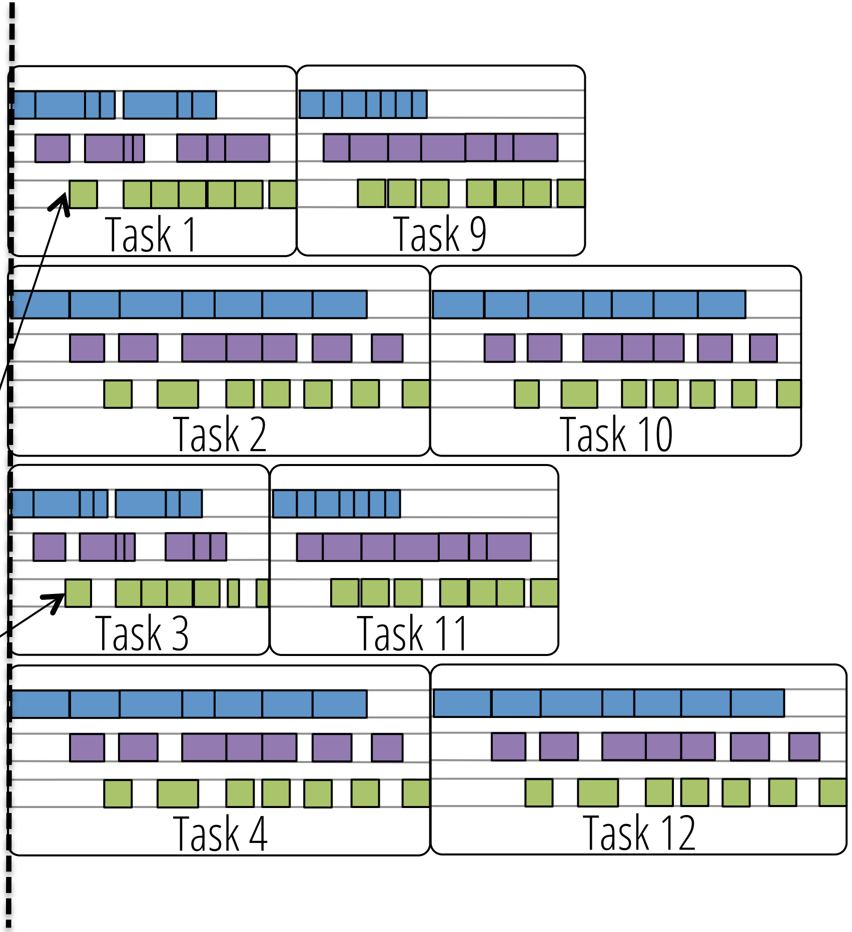


Spark Summit 2015:

Blocked time analysis: how quickly could a job have completed if a resource were infinitely fast? (upper bound)

Result of ~1 year of adding metrics to Spark!

How much faster would the job be with 2x disk throughput?



How would runtimes for these disk writes change?

How would that change timing of (and contention for) other resources?

Challenges to reasoning about performance

Tasks bottleneck on different resources at different times

Concurrent tasks on a machine may contend for resources

No model for performance

How can we achieve this vision?

Spark overview

Reasoning about Spark’s performance: why it’s hard

New architecture: monotasks

Reasoning about monotasks performance: why it’s easy

Monotasks in action (results)

Spark:

Tasks bottleneck on different resources

Concurrent tasks may contend

No model for performance

Spark:

Tasks bottleneck on different resources

Concurrent tasks may contend

No model for performance

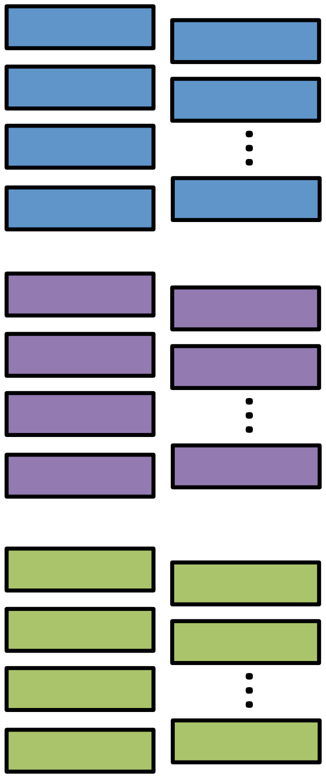
Monotasks: Each task uses one resource

Example Spark Job:

Read remote data

Filter records

Write result to disk



Network monotasks:
Each read one remote block

CPU monotasks:
Each filter one block, generate serialized output

Disk monotasks:
Each writes one block to disk

Spark:

Tasks bottleneck on different resources

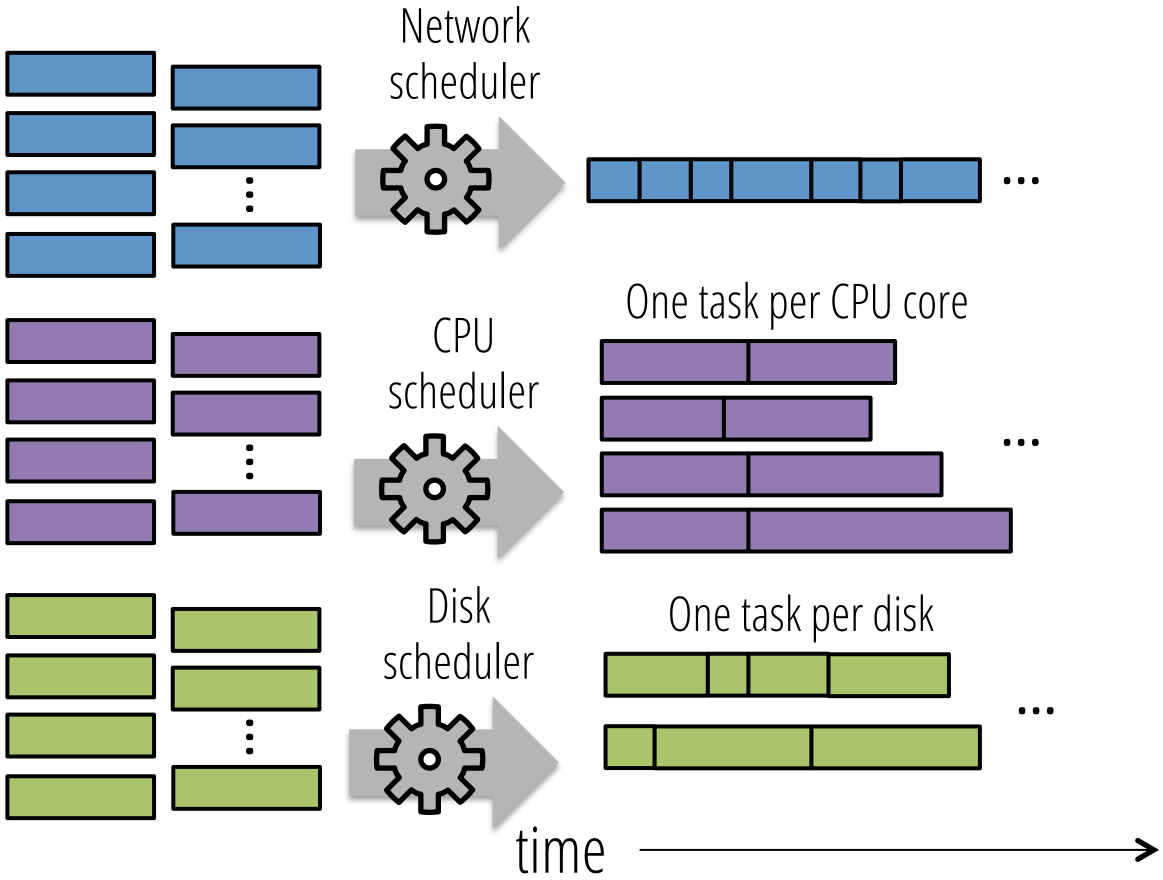
Concurrent tasks may contend

No model for performance

Monotasks:

Each task uses one resource

Dedicated schedulers control contention



Spark:

Tasks bottleneck on different resources

Concurrent tasks may contend

No model for performance

Monotasks:

Each task uses one resource

Dedicated schedulers control contention

Monotask times can be used to model performance

Ideal CPU time: total CPU monotask time / # CPU cores

Spark:

Tasks bottleneck on different resources

Concurrent tasks may contend

No model for performance

Monotasks:

Each task uses one resource

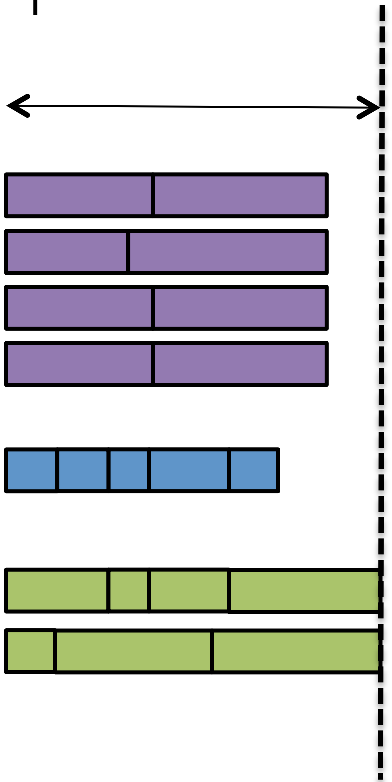
Dedicated schedulers control contention

Monotask times can be used to model performance

Ideal CPU time: total CPU monotask time / # CPU cores

Ideal network runtime

Ideal disk runtime



Job runtime: max of ideal times

Spark:

Tasks bottleneck on different resources

Concurrent tasks may contend

No model for performance

Monotasks:

Each task uses one resource

Dedicated schedulers control contention

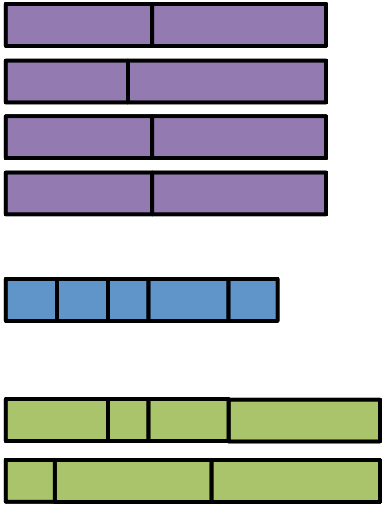
Monotask times can be used to model performance

How much faster would the job be with 2x disk throughput?

Ideal CPU time: total CPU monotask time / # CPU cores

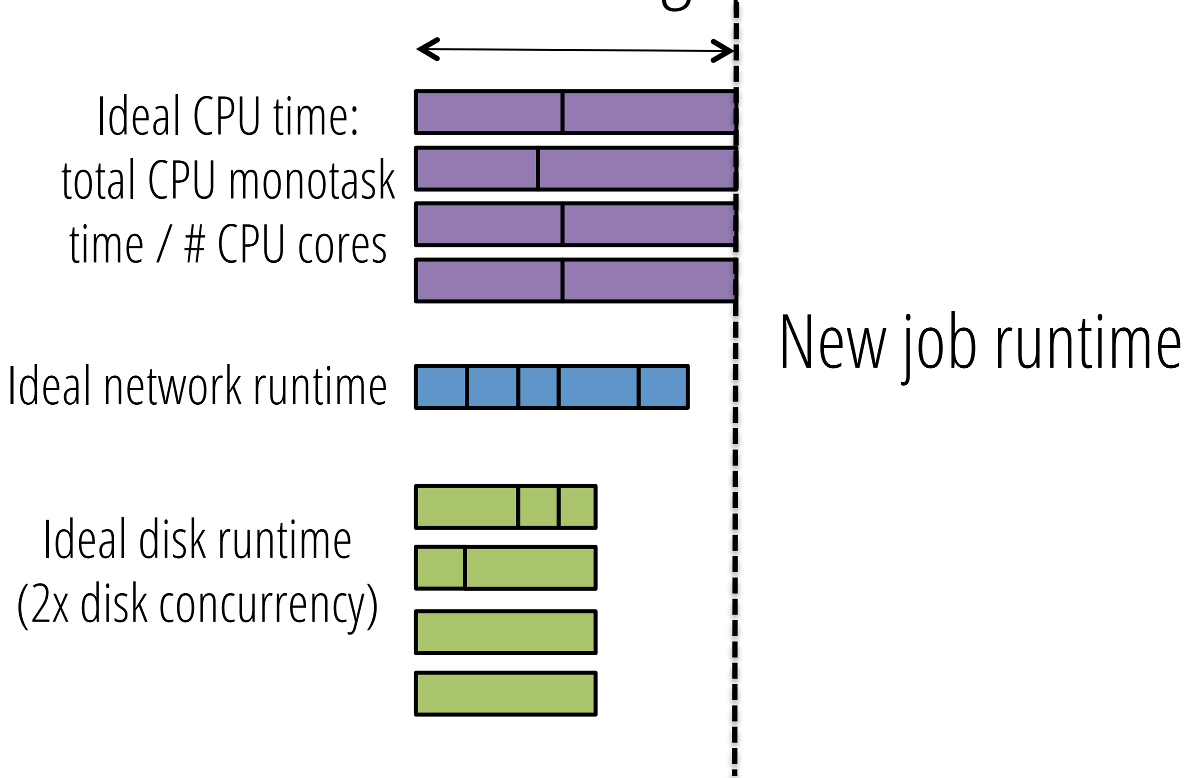
Ideal network runtime

Ideal disk runtime



Spark:	Monotasks:
Tasks bottleneck on different resources	Each task uses one resource
Concurrent tasks may contend	Dedicated schedulers control contention
No model for performance	Monotask times can be used to model performance

How much faster would the job be with 2x disk throughput?



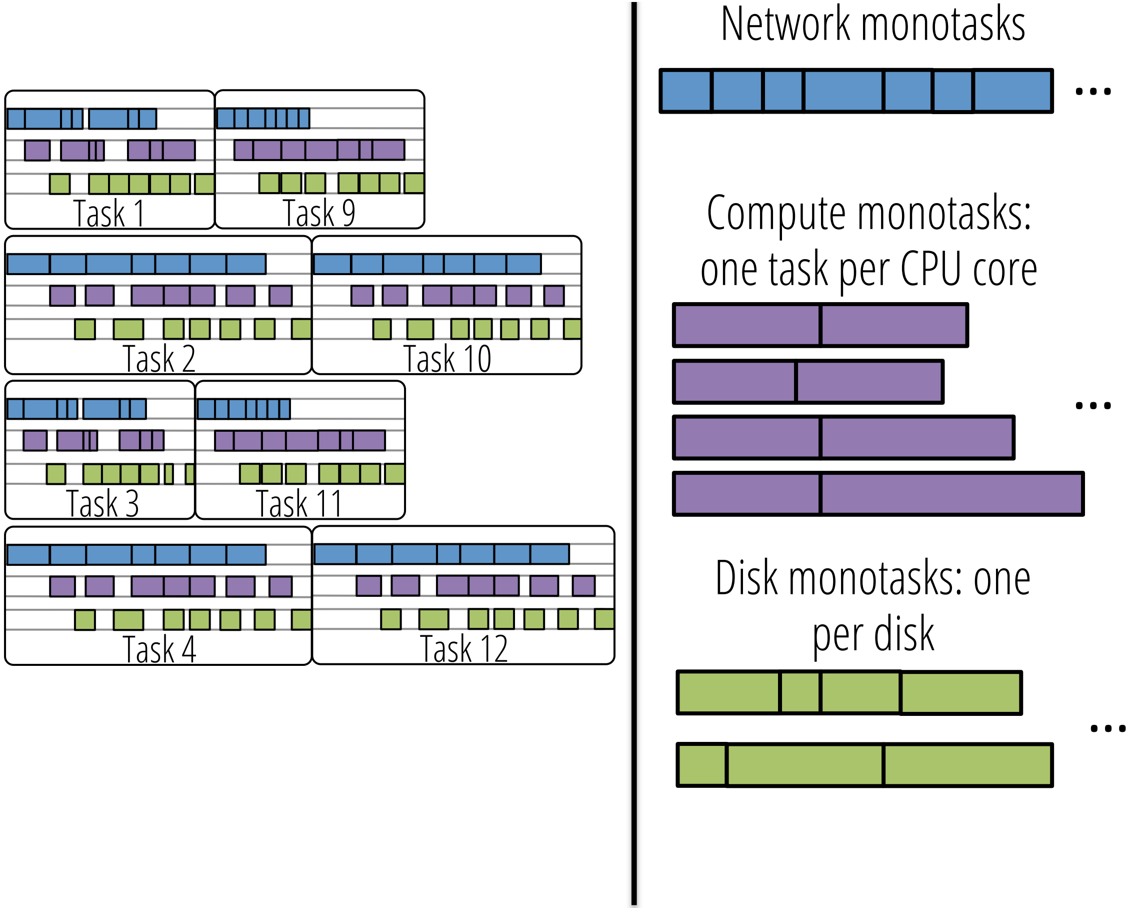
How does this decomposition work?

Spark:

Tasks bottleneck on different resources

Concurrent tasks may contend

No model for performance



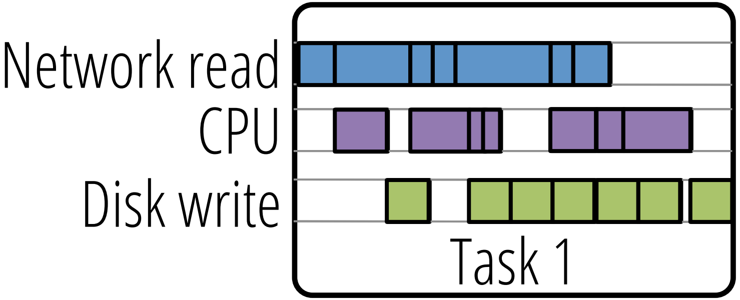
Monotasks:

Each task uses one resource

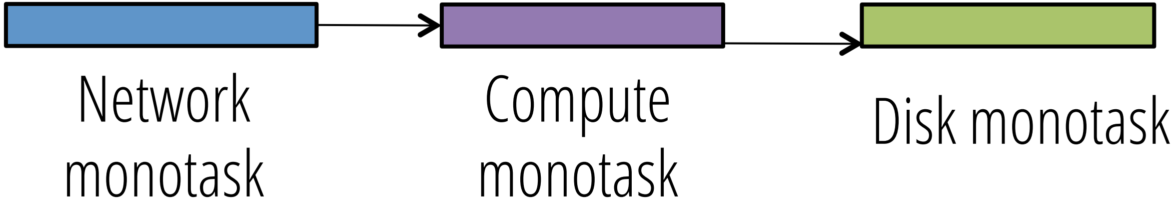
Dedicated schedulers control contention

Monotask times can be used to model performance

How does this decomposition work?



Monotasks



Implementation

API-compatible with Apache Spark

- Workloads can be run on monotasks without re-compiling
- Monotasks decomposition handled by Spark internals

Monotasks works at the application level

- No operating system changes

How can we achieve this vision?

Spark overview

Reasoning about Spark’s performance: why it’s hard

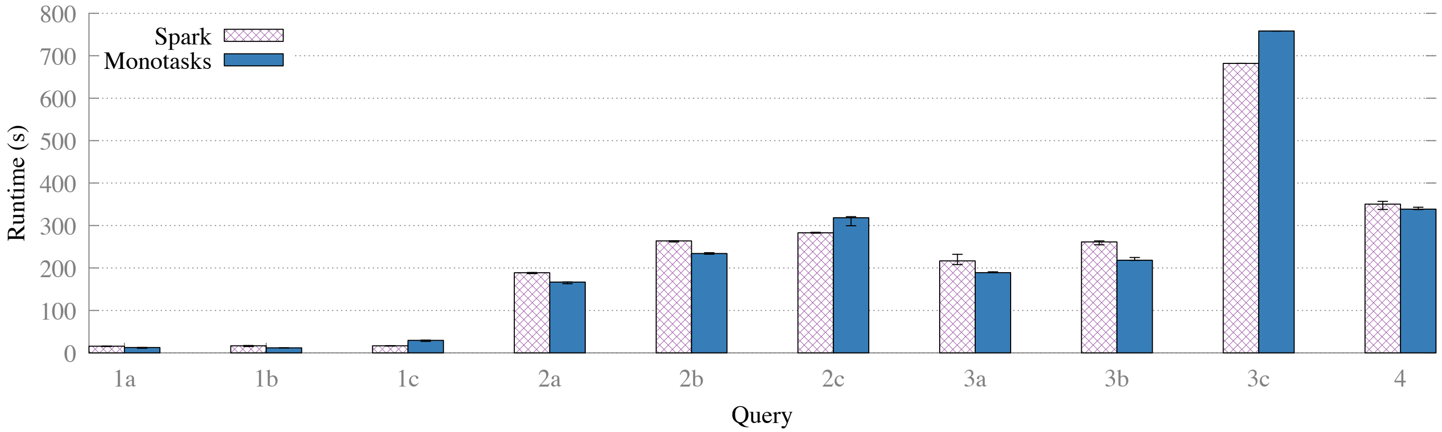
New architecture: monotasks

Reasoning about monotasks performance: why it’s easy

Monotasks in action (results)

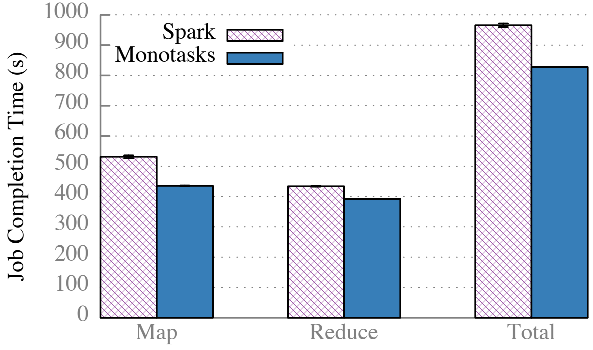
Performance on-par with Apache Spark

Big data benchmark
(SQL workload)

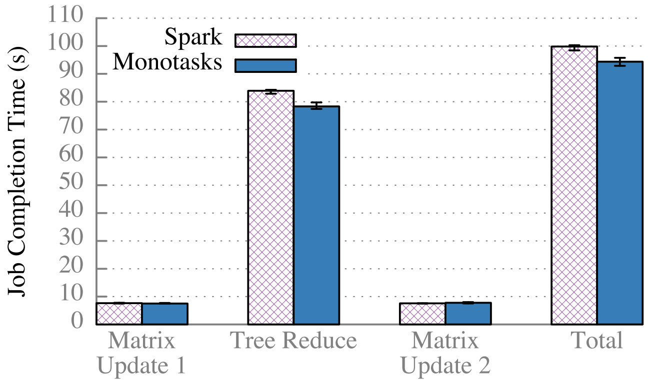


Performance on-par with Apache Spark

Sort
(600 GB, 20 machines)



Block coordinate descent
(Matrix workload used in ML applications)
16 machines

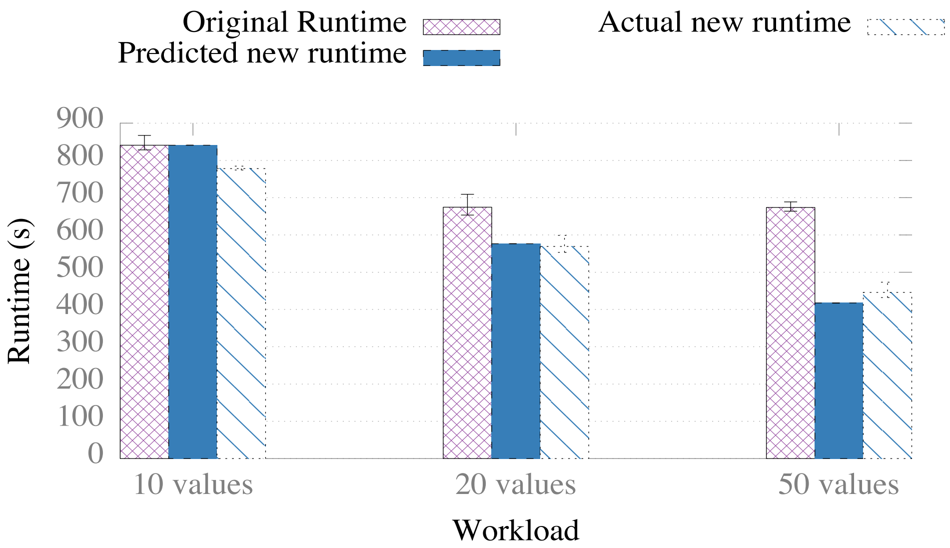


Monotasks in action

Modeling performance

Leveraging performance clarity to optimize performance

How much faster would jobs run if each machine had 2 disks instead of 1?



Predictions for different hardware within 10% of the actual runtime

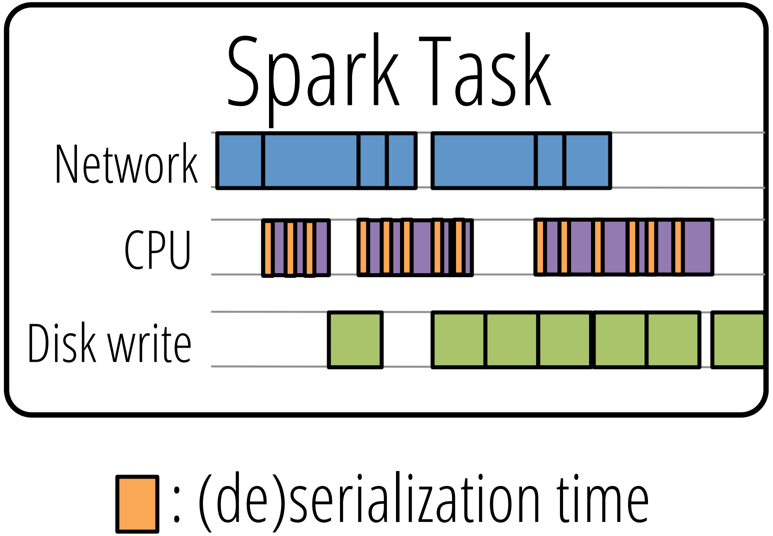
How much faster would job run if data were de-serialized and in memory?

Eliminates disk time to read input data

Eliminates CPU time to de-serialize data

How much faster would job run if data were de-serialized and in memory?

Measuring (de) serialization time with Spark



(De) serialization pipelined with other processing *for each record*

Application-level measurement incurs high overhead

How much faster would job run if data were de-serialized and in memory?

Measuring (de) serialization time with Monotasks

Original compute monotask

Un-rolled monotask

: (de)serialization time

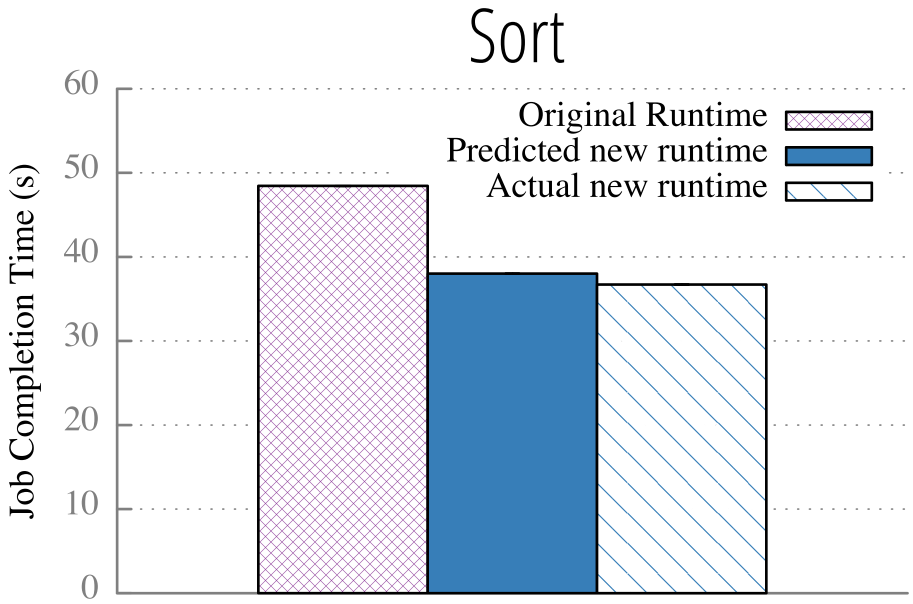
Eliminating fine-grained pipelining enables measurement!

How much faster would job run if data were de-serialized and in memory?

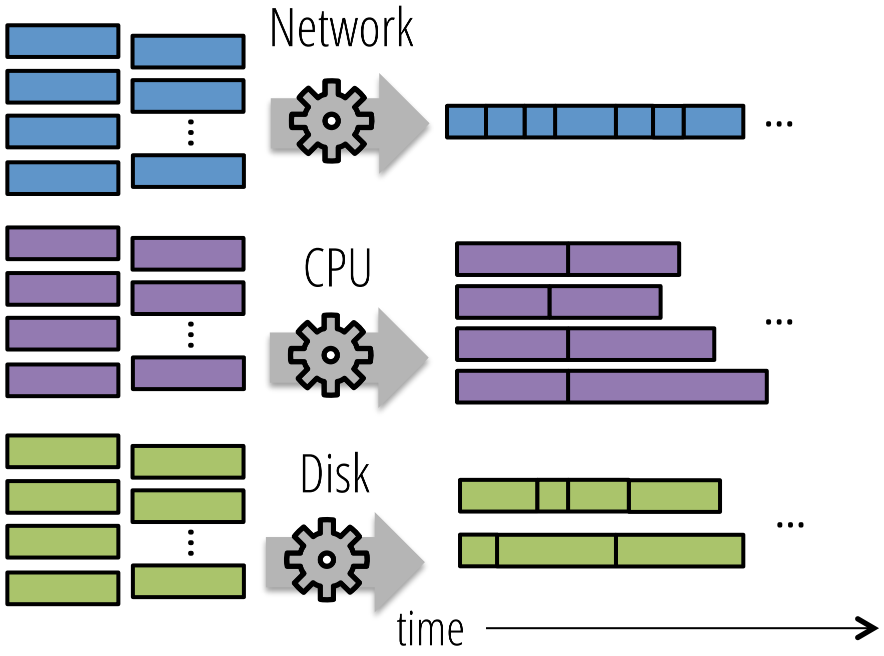
Eliminate disk monotask time

Eliminate CPU monotask time spent (de)serialiazing

Re-run model



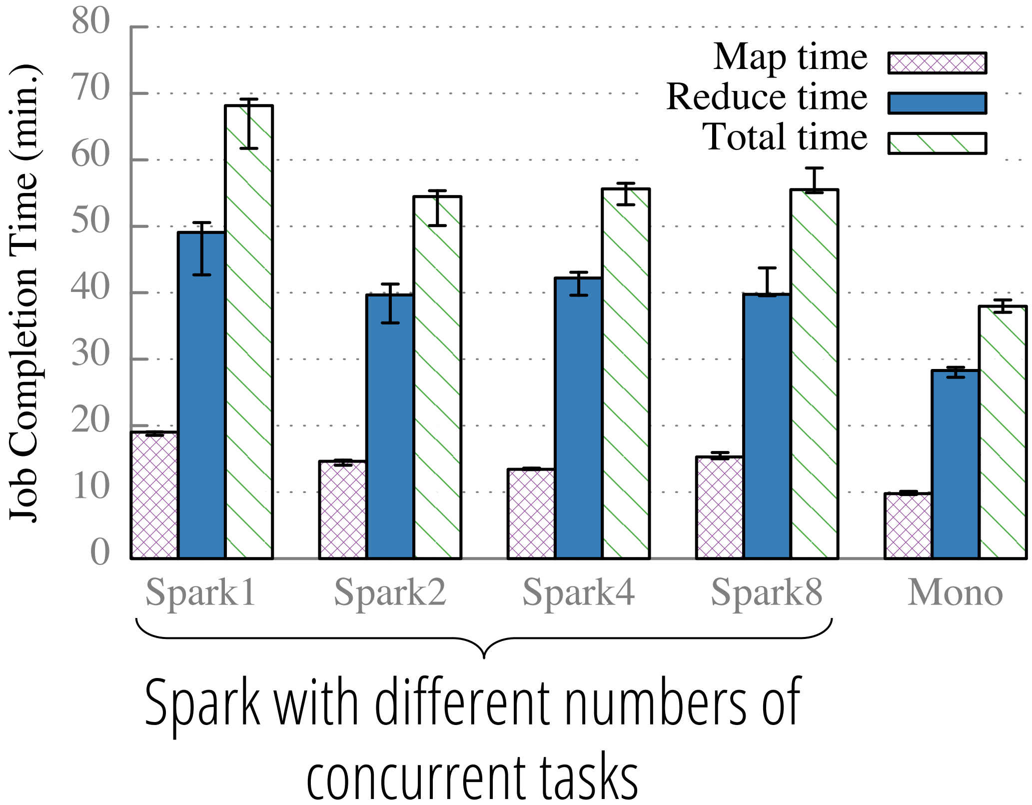
Leveraging Performance Clarity to Improve Performance



Schedulers have complete visibility over resource use

Framework can configure for best performance

Configuring the number of concurrent tasks



Monotasks better than any configuration:
per-resource schedulers automatically schedule with the ideal concurrency

Future Work

Leveraging performance clarity to improve performance

- Use resource queue lengths to dynamically adjust job

Automatically configuring for multi-tenancy

- Don’t need jobs to specify resource requirements
- Can achieve higher utilization: no multi-resource bin packing

Spark Monotasks | Jobs | Stages | Storage | Environment | Executors

Details for Stage 17

Total task time across all tasks: 13 min
Shuffle read: 2.5 GB / 31589120

Performance Information

Bottleneck: Disk (if disk bandwidth were increased by 23% or more, network would become the bottleneck)
Non-bottlenecks: Network (could reduce network bandwidth by up to 30% slower without impacting runtime
increase CPU time by up to 2x without impacting runtime)
Benefit of caching: Storing input in-memory would reduce job completion time by 42%
Job Runtime Predictor: (enter in properties of different cluster to estimate job's runtime)
Number of machines:
CPU cores per machine:
Network bandwidth per machine: Gbps

Vision:

Spark always reports bottleneck information

Challenging with existing architecture

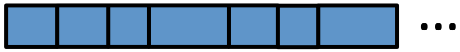
Monotasks:

Each task uses one resource

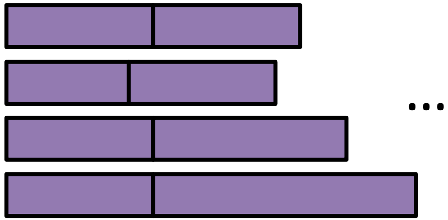
Dedicated schedulers control contention

Monotask times can be used to model performance

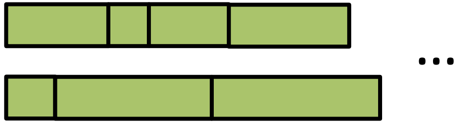
Network monotasks



Compute monotasks: one task per CPU core



Disk monotasks: one per disk



Interested? Have a job whose performance you can't figure out? Email me: keo@cs.berkeley.edu