Introduction to Spark Internals

Matei Zaharia

UC Berkeley

www.spark-project.org

Outline

Project goals

Components

Life of a job

Extending Spark

How to contribute

Project Goals

Generality: diverse workloads, operators, job sizes

Low latency: sub-second

Fault tolerance: faults shouldn't be special case

Simplicity: often comes from generality

Codebase Size

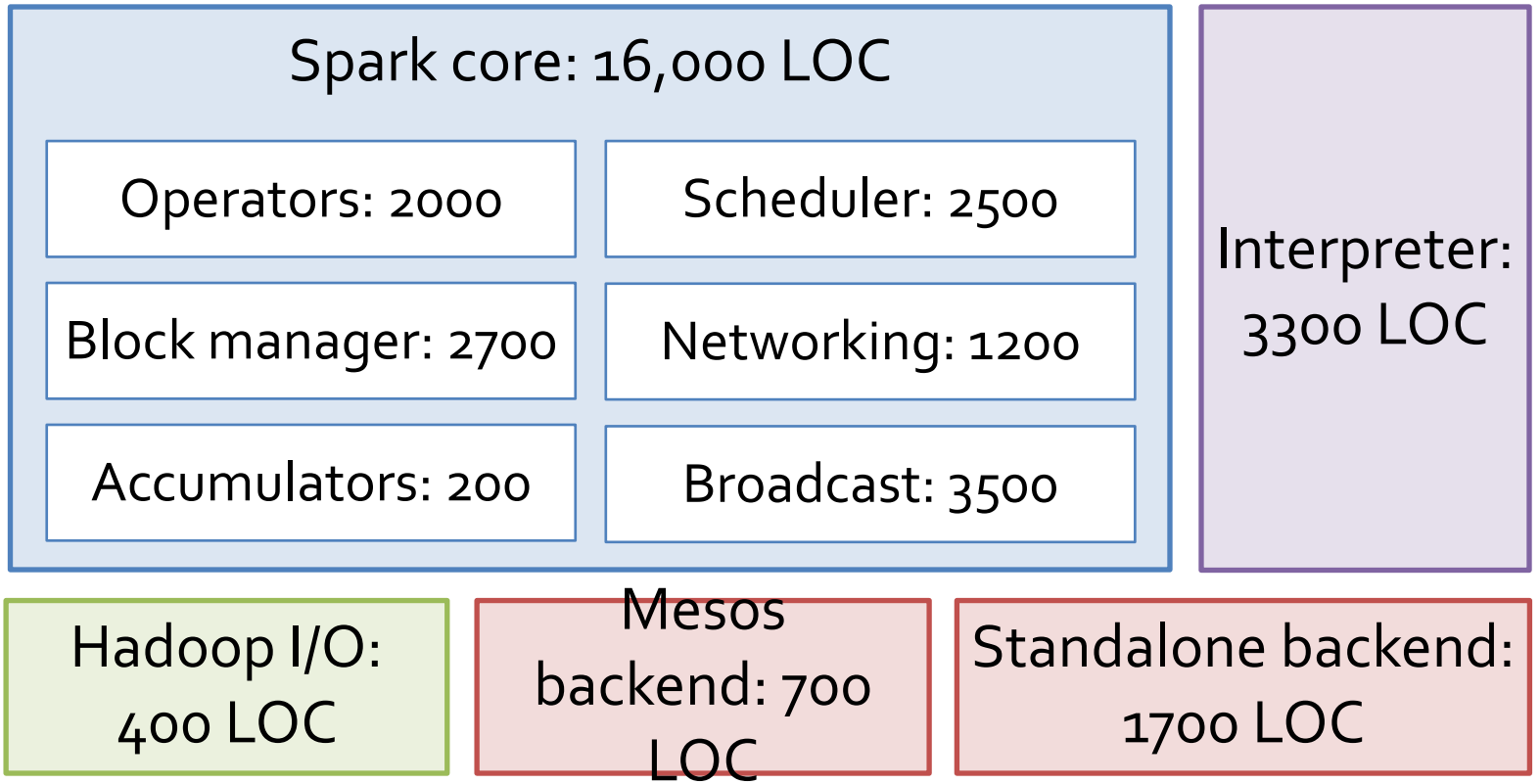
Spark: 20,000 LOC

Hadoop 1.0: 90,000 LOC

Hadoop 2.0: 220,000 LOC

(non-test, non-example sources)

Codebase Details



Outline

Project goals

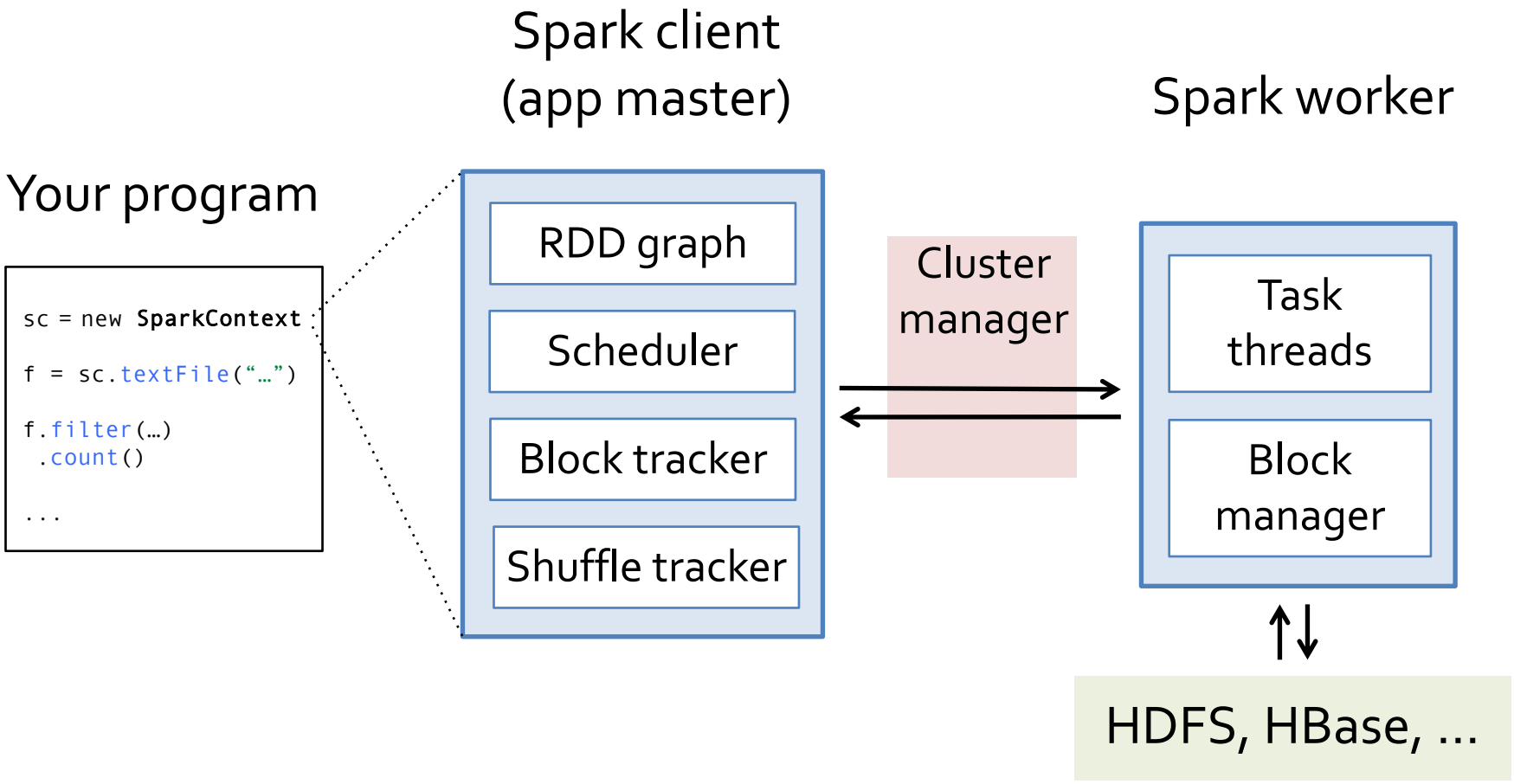
Components

Life of a job

Extending Spark

How to contribute

Components


Spark client
(app master)
Spark worker
Your program
sc = new SparkContext
f = sc.textFile("…")
f.filter(…)
.count()
...
RDD graph
Scheduler
Block tracker
Shuffle tracker
Cluster manager
Task threads
Block manager
HDFS, HBase, …

Example Job

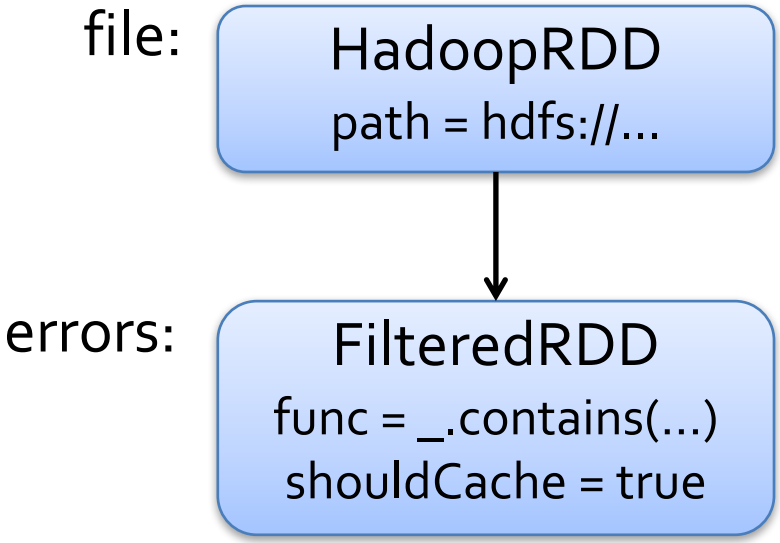
```
val sc = new SparkContext(
  “spark://...”, “MyJob”, home, jars)

val file = sc.textFile(“hdfs://...”)

val errors = file.filter(_.contains(“ERROR”))

errors.cache()

errors.count()
```

Resilient distributed datasets (RDDs)

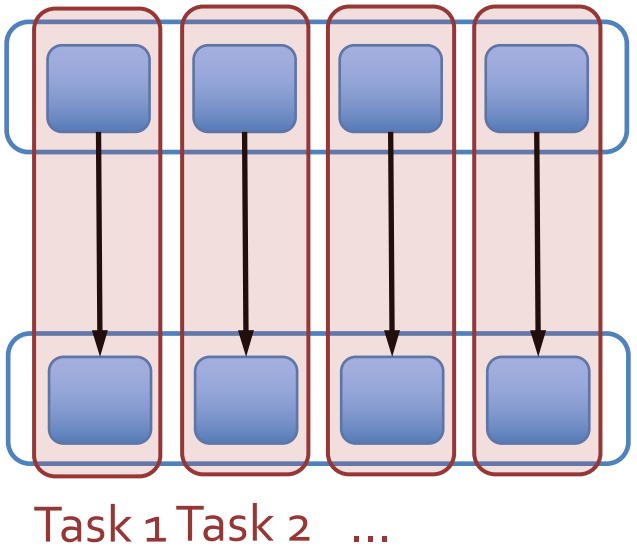
Action

RDD Graph

Dataset-level view:



Partition-level view:



Data Locality

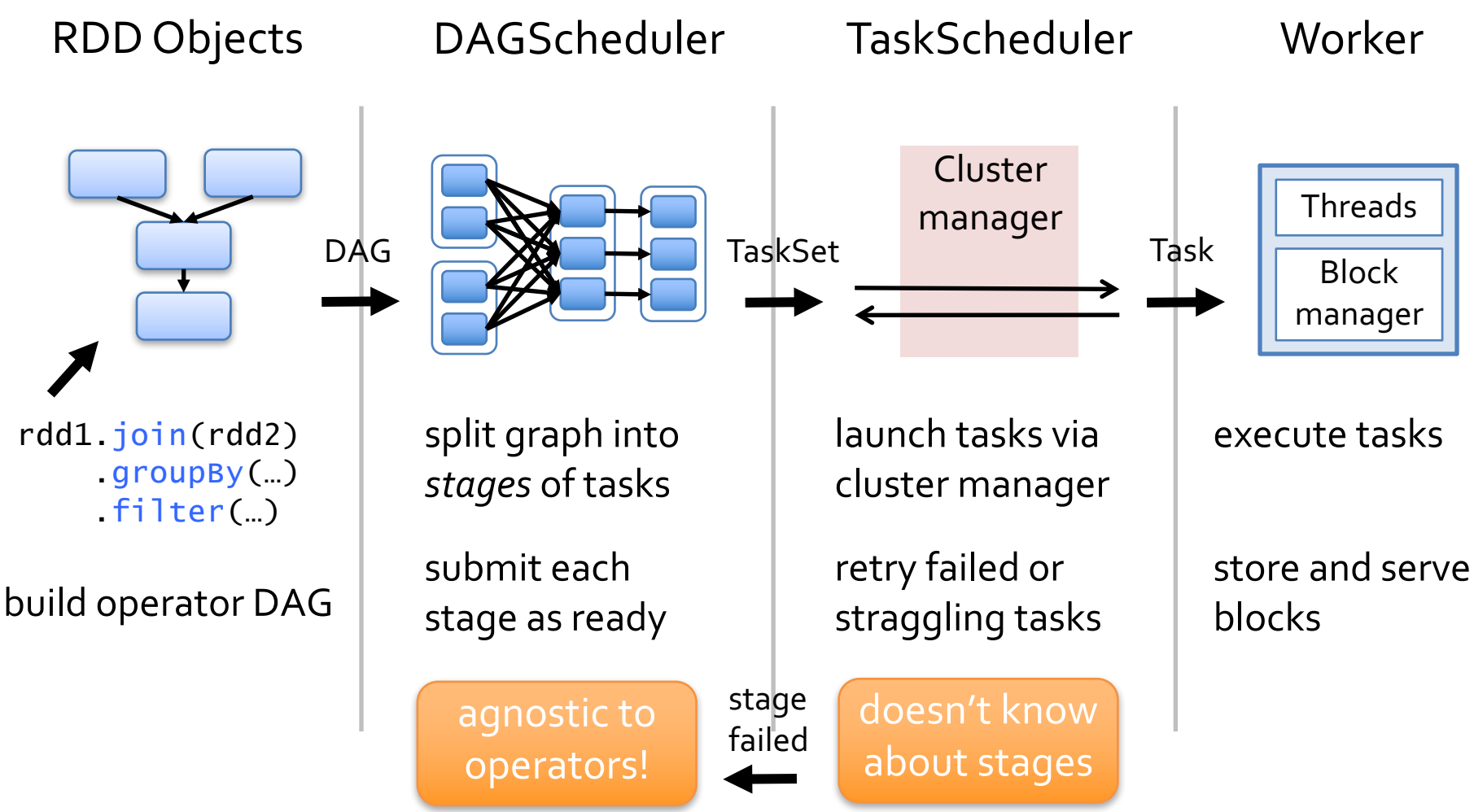
First run: data not in cache, so use HadoopRDD's locality prefs (from HDFS)

Second run: FilteredRDD is in cache, so use its locations

If something falls out of cache, go back to HDFS

In More Detail: Life of a Job

Scheduling Process



RDD Abstraction

Goal: wanted to support wide array of operators and let users compose them arbitrarily

Don’t want to modify scheduler for each one

How to capture dependencies generically?

RDD Interface

Set of *partitions* (“splits”)

List of *dependencies* on parent RDDs

Function to *compute* a partition given parents

Optional *preferred locations*

Optional *partitioning info* (Partitioner)

Captures all current Spark operations!

Example: HadoopRDD

partitions = one per HDFS block

dependencies = none

compute*(partition)* = read corresponding block

preferredLocations*(part)* = HDFS block location

partitioner = none

Example: FilteredRDD

partitions = same as parent RDD

dependencies = "one-to-one" on parent

compute*(partition)* = compute parent and filter it

preferredLocations*(part)* = none (ask parent)

partitioner = none

Example: JoinedRDD

partitions = one per reduce task

dependencies = "shuffle" on each parent

compute*(partition)* = read and join shuffled data

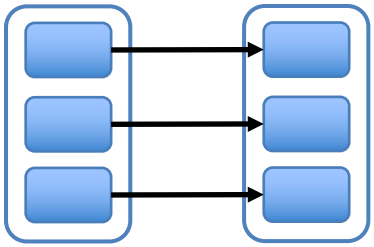
preferredLocations*(part)* = none

partitioner = HashPartitioner(numTasks)

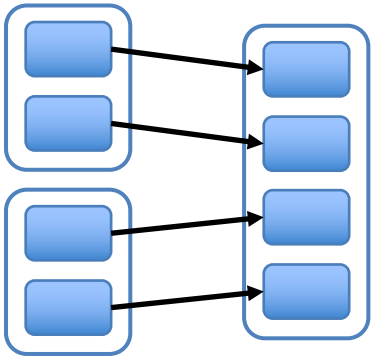
Spark will now know this data is hashed!

Dependency Types

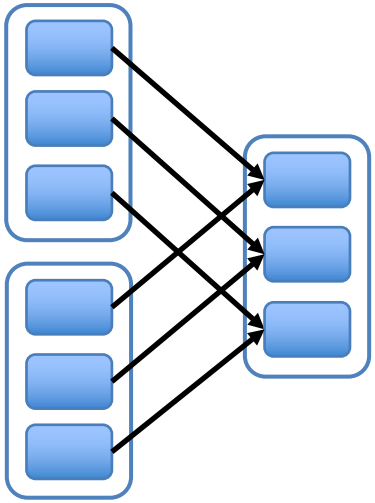
"Narrow" deps:



map, filter

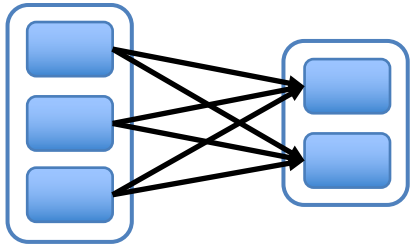


union

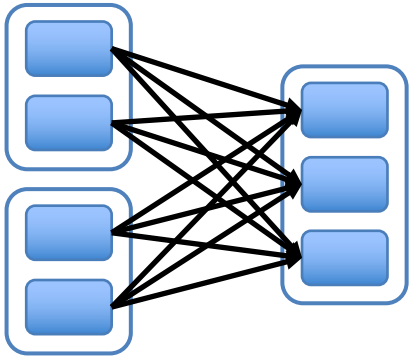


join with inputs co-partitioned

"Wide" (shuffle) deps:



groupByKey



join with inputs not co-partitioned

DAG Scheduler

Interface: receives a "target" RDD, a function to run on each partition, and a listener for results

Roles:

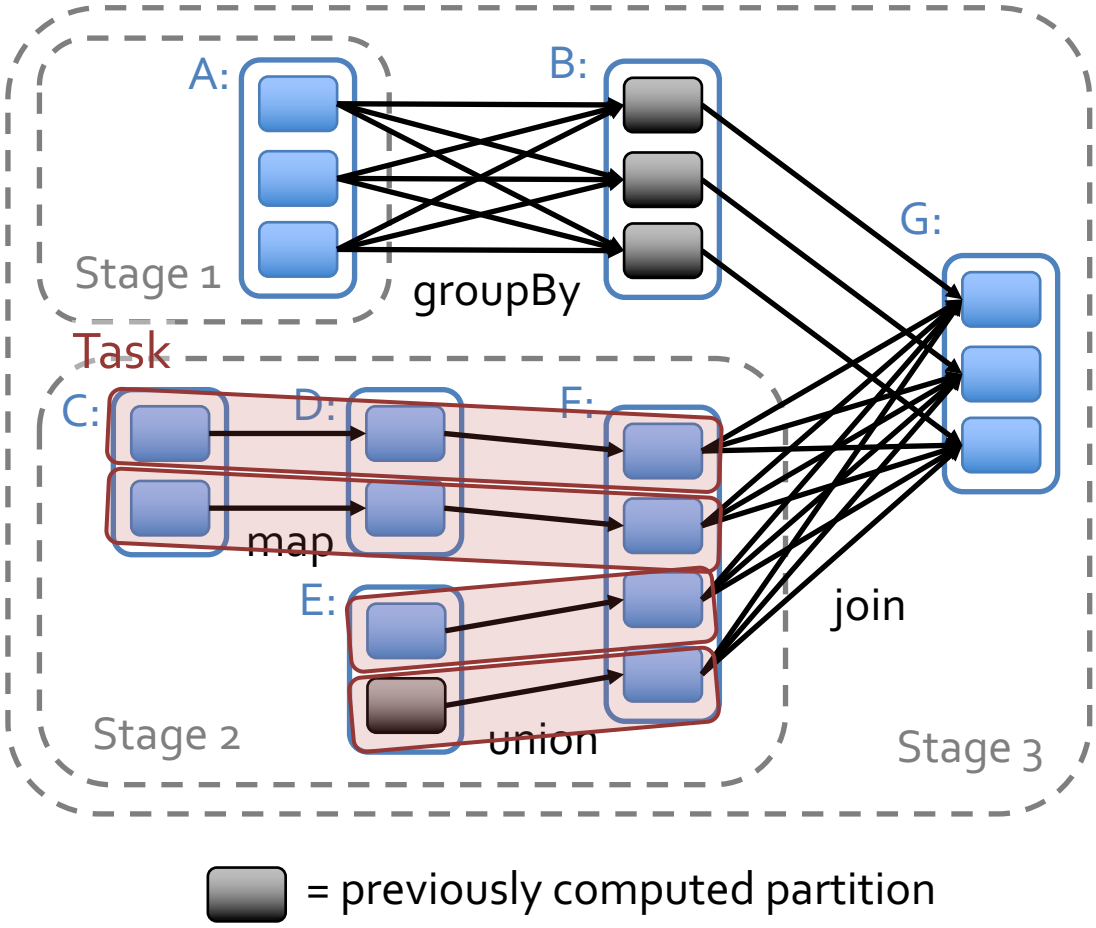
- Build stages of Task objects (code + preferred loc.)
- Submit them to TaskScheduler as ready
- Resubmit failed stages if outputs are lost

Scheduler Optimizations

Pipelines narrow ops. within a stage

Picks join algorithms based on partitioning (minimize shuffles)

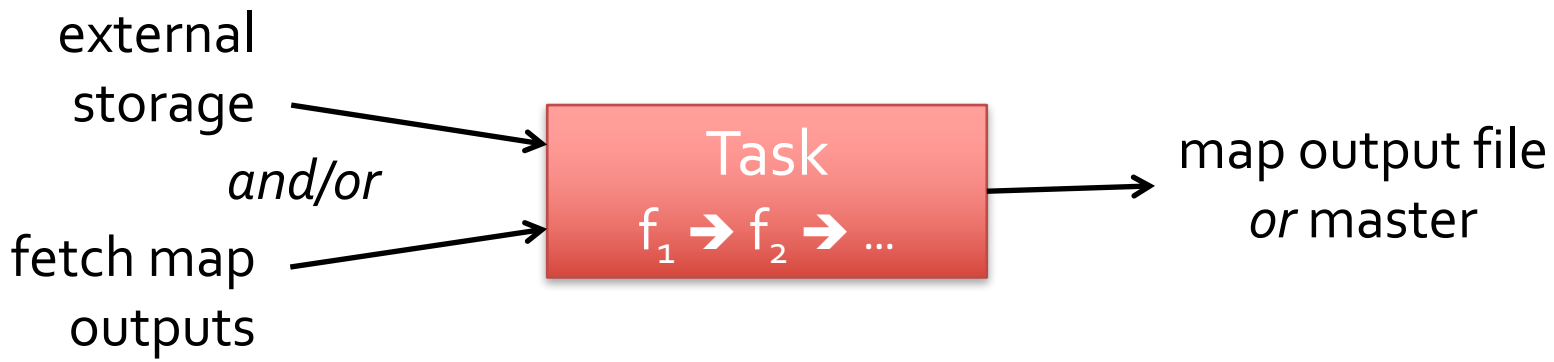
Reuses previously cached data



Task Details

Stage boundaries are only at input RDDs or "shuffle" operations

So, each task looks like this:



(Note: we write shuffle outputs to RAM/disk to allow retries)

Task Details

Each Task object is self-contained

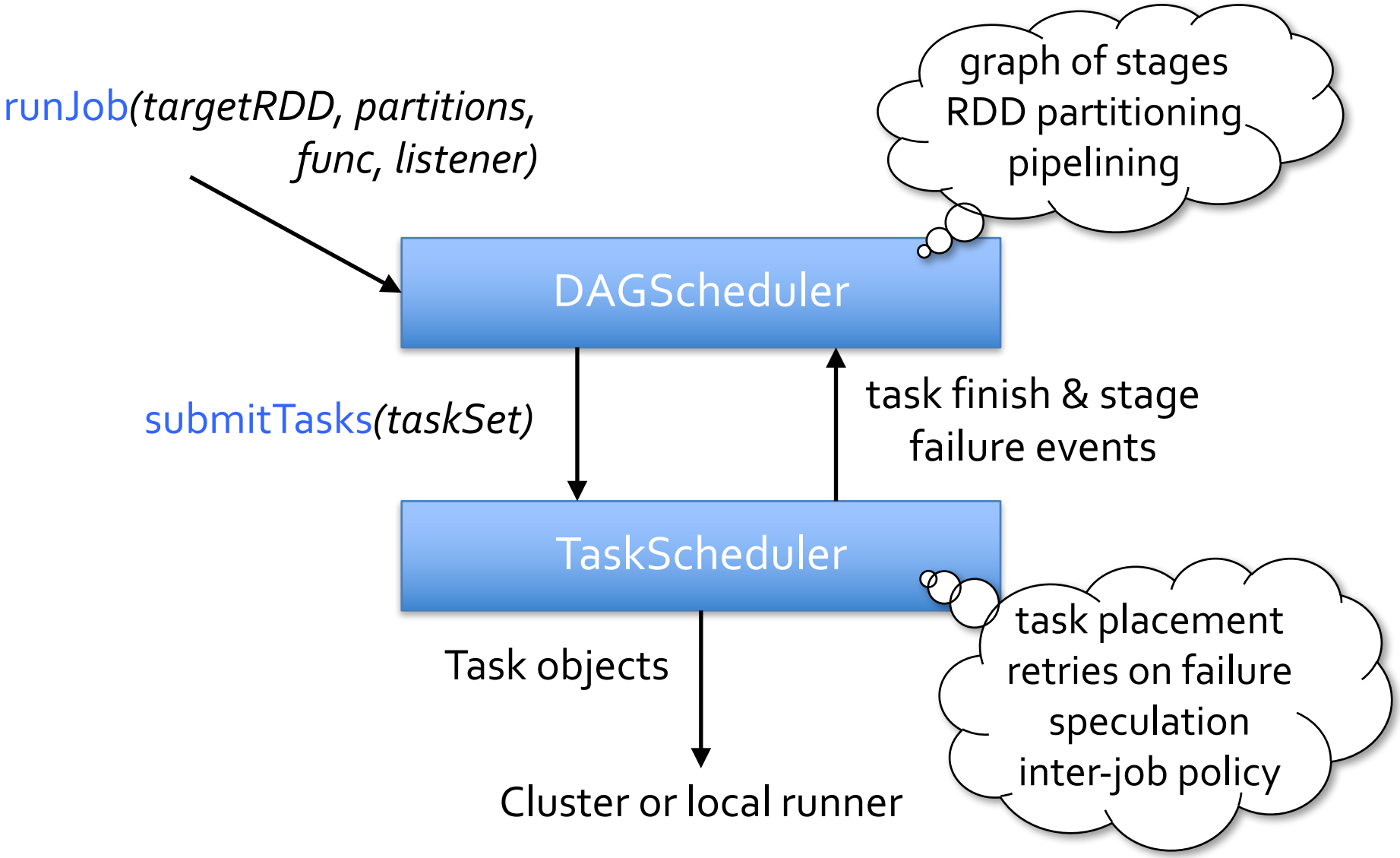
» Contains all transformation code up to input boundary (e.g. HadoopRDD => filter => map)

Allows Tasks on cached data to even if they fall out of cache

Design goal: any Task can run on any node

Only way a Task can fail is lost map output files

Event Flow


runJob(targetRDD, partitions, func, listener)
graph of stages
RDD partitioning
pipelining
DAGScheduler
submitTasks(taskSet)
task finish & stage failure events
TaskScheduler
Task objects
task placement
retries on failure
speculation
inter-job policy
Cluster or local runner

TaskScheduler

Interface:

- Given a TaskSet (set of Tasks), run it and report results
- Report "fetch failed" errors when shuffle output lost

Two main implementations:

- LocalScheduler (runs locally)
- ClusterScheduler (connects to a cluster manager using a pluggable "SchedulerBackend" API)

TaskScheduler Details

Can run multiple concurrent TaskSets, but currently does so in FIFO order

- » Would be really easy to plug in other policies!
- » If someone wants to suggest a plugin API, please do

Maintains one TaskSetManager per TaskSet that tracks its locality and failure info

Polls these for tasks in order (FIFO)

Worker

Implemented by the Executor class

Receives self-contained Task objects and calls run() on them in a thread pool

Reports results or exceptions to master

- » Special case: FetchFailedException for shuffle

Pluggable ExecutorBackend for cluster

Other Components

BlockManager

- » "Write-once" key-value store on each worker
- » Serves shuffle data as well as cached RDDs
- » Tracks a StorageLevel for each block (e.g. disk, RAM)
- » Can drop data to disk if running low on RAM
- » Can replicate data across nodes

Other Components

CommunicationManager

- » Asynchronous IO based networking library
- » Allows fetching blocks from BlockManagers
- » Allows prioritization / chunking across connections (would be nice to make this pluggable!)
- » Fetch logic tries to optimize for block sizes

Other Components

MapOutputTracker

- » Tracks where each "map" task in a shuffle ran
- » Tells reduce tasks the map locations
- » Each worker caches the locations to avoid refetching
- » A "generation ID" passed with each Task allows invalidating the cache when map outputs are lost

Outline

Project goals

Components

Life of a job

Extending Spark

How to contribute

Extension Points

Spark provides several places to customize functionality:

Extending RDD: add new input sources or transformations

SchedulerBackend: add new cluster managers

spark.serializer: customize object storage

What People Have Done

New RDD transformations (`sample, glom, mapPartitions, leftOuterJoin, rightOuterJoin`)

New input sources (DynamoDB)

Custom serialization for memory and bandwidth efficiency

New language bindings (Java, Python)

Possible Future Extensions

Pluggable inter-job scheduler

Pluggable cache eviction policy (ideally with priority flags on StorageLevel)

Pluggable instrumentation / event listeners

Let us know if you want to contribute these!

As an Exercise

Try writing your own input RDD from the local filesystem (say one partition per file)

Try writing your own transformation RDD (pick a Scala collection method not in Spark)

Try writing your own action (e.g. product())

Outline

Project goals

Components

Life of a job

Extending Spark

How to contribute

Development Process

Issue tracking: spark-project.atlassian.net

Development discussion: spark-developers

Main work: "master" branch on GitHub

» Submit patches through GitHub pull requests

Be sure to follow code style and add tests!

Build Tools

SBT and Maven currently both work (but switching to only Maven)

IDEA is the most common IDEA; Eclipse may be made to work

Thanks!

Stay tuned for future developer meetups.