Resilient Distributed Datasets

Knowledge objectives

1. Define RDD
2. Name the main Spark runtime components
3. Name the main Spark contributions and characteristics
4. Distinguish between transformations and actions
5. Explain available transformations
6. Explain available actions
7. Distinguish between narrow and wide dependencies
8. Name the two mechanisms to share variables
9. Enumerate current abstraction on top of Spark

Application Objectives

1. Provide the spark pseudo-code for a simple problem

BASICS

Addressed limitations

- Resource sharing
- Computation composition
- Work duplication
  - Raise abstraction level for
    - Work distribution
    - Fault tolerance
- Management and administration
  - Unify execution model
- Limited scope of programming model

Resilient Distributed Datasets

"Unified **abstraction** for cluster computing, consisting in a **read-only**, partitioned collection of records. Can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs."

(1) for the first RDD (import from text file) and then (2) for all rest

read-only means write-one

you cannot modify, but from one RDD you can create another one.

ETL example: RDDs are not replicating data, data is not stored on each step.

RDDs is something abstract, they represent the data, but they don't store data itself.

Matei Zaharia

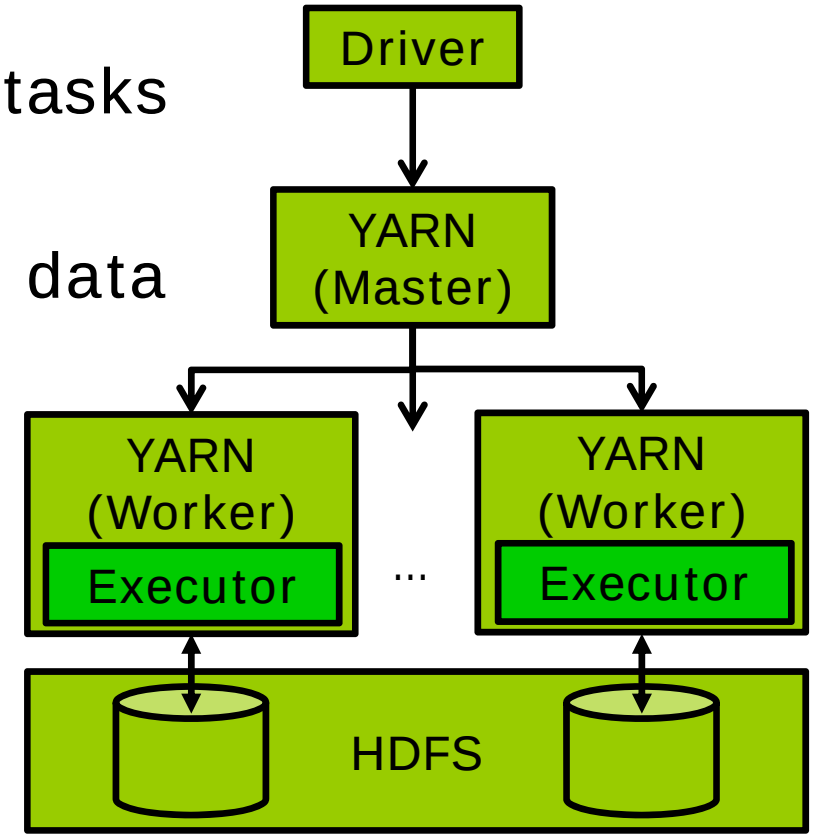
```
rdd := spark.textFile("hdfs://...")
```

ARCHITECTURE

Runtime architecture

- Driver
  - Creates the context
  - Decides on RDDs
  - Converts a program into tasks
  - Schedules tasks
  - Tracks location of cached data
- YARN (Master)
  - Resource manager
- Executors
  - Run tasks
  - Store data

some RDDs may be materialized (if it is reused many times) so t can get into memory or disk.



Characteristics

- Statically typed
- Parallel data structures
  - Disk
  - Memory
- User controls ...
  - Data sharing
  - Partitioning (fixed number per RDD)
    - Repartition (shuffles data through disk)
    - Coalesce (reduces partitions in the same worker)

so it doesn't shuffle anything. From 3 partitions within the worker it creates 1

- Rich set of coarse-grained operators
  - Simple and efficient programming interface
- Fault tolerant

stores intermediate results into the disk (but not all of them)

- Baseline for more abstract applications

Contributions compared to MapReduce

- Immutable storage of arbitrary records across a cluster
- Low latency (in-memory storage)
- Palette of coarse-grained operators
- Control over data partitioning

Spark is not in-memory, it tries to keep things in memory, but sometimes it has to go to the disk (e.g. on shuffling)

PROGRAMMING MODEL

Suitable applications

- Batch applications
- Same operations to all elements
- No fine-grained updates
  - Read-only lookup is possible
- No asynchronous updates

upon modification you create a new RDD (there is no modification in same RDD possible)

Types of operations

a) Transformations

are not executed until you have actions. Triggered upon action

- Evaluation is grouped

execute batch transformation, not one-by-one

tuples if possible has to go through all tranformations until the end. That's in order to save memory.

  - Reduces passes

b) Actions

finding an action, spark pulls the data through all tranformations (RDDs).

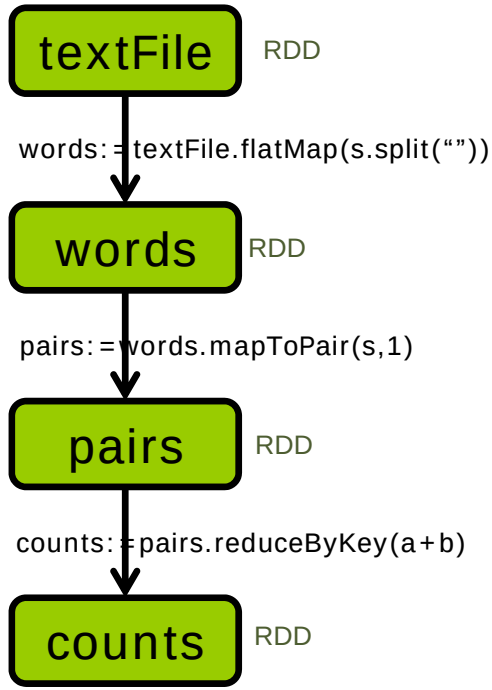
1. Force evaluation of transformations
   - From scratch (even loading data from disk)
2. Produce output
   a) Return a final value to the driver
   b) Write data to an external storage system

Example:Word count

```
trsf JavaRDD<String> textFile = sc.textFile("hdfs://...");
     JavaRDD<String> words = textFile.flatMap(
         new FlatMapFunction<String, String>() {
             public Iterable<String> call(String s) {
                 return Arrays.asList(s.split(" "));
                 }
             }
         );
trsf JavaPairRDD<String, Integer> pairs = words.mapToPair(
         new PairFunction<String, String, Integer>() {
             public Tuple2<String, Integer> call(String s) {
                 return new Tuple2<String, Integer>(s, 1);
                 }
             }
         );
trsf JavaPairRDD<String, Integer> counts = pairs.reduceByKey(
         new Function2<Integer, Integer, Integer>() {
             public Integer call(Integer a, Integer b) { return a + b; }
             }
         );
     counts.saveAsTextFile("hdfs://...");
```

needs to be commutative and associative for reduceByKey to work

action, that starts pulling data through transformations



Transformations on base RDDs

distinct(T): RDD[T]→RDD[T]

map(f:T→U): RDD[T]→RDD[U] one to one

flatMap(f:T→seq[U]): RDD[T]→RDD[U] one to many for example, for every line we can one call, and generates many words as output

filter(f:T→bool): RDD[T]→RDD[T] one to one-or-none based on a function

important because you can estimate the costs

sample(fraction: Float): RDD[T]→RDD[T] based on some randomization you know how much it reduces your records

union/intersection/substract(): (RDD[T],RDD[T])→RDD[T]

crossProduct(): (RDD[T],RDD[U])→RDD[(T,U)]

partitionBy(p:partitioner[T]): RDD[T]→RDD[T]

persist(): RDD[T]→RDD[T] persist is caching, persisting in-memory

why to do that while noone is asking for this

that's why it's not an action

mapToPair(f:T→(K,V)): RDD[T]→RDD[(K,V)]

from key to key,value

the idea of the action is to start pulling the data, but persist does not go to driver, so it's not an action.

until someone really requests to access data you don't need to execute anything.

the only point of having many operators is to explicit to the compiler to perform better, choosing the correct one in each case.

Added transformations on pair RDDs

mapValues(f:V→W): RDD[(K,V)]→RDD[(K,W)] ! one-to-one

keys are the same, so there is no need of shuffling, which produces no traffic, so its good. If you use map, it's going to use shuffling, because the framework doesn't know that you are not changing the key.

sort(c:comparator[K]): RDD[(K,V)]→RDD[(K,V)]

reduceByKey(f:(V,V)→V): RDD[(K,V)]→RDD[(K,V)]

groupByKey(): RDD[(K,V)]→RDD[(K,seq(V))]

join(): (RDD[(K,V)],RDD[(K,W)])→RDD[(K,(V,W))]

cogroup(): (RDD[(K,V)],RDD[(K,W)])→RDD[(K,(seq[V],seq[W])]

smaller than join because you dont replicate values

partitionBy(p:partitioner[K]): RDD[(K,V)]→RDD[(K,V)]

keys(): RDD[(K,V)] → RDD[K]

values(): RDD[(K,V)] → RDD[V]

Actions on base RDDs

save(path: String): store RDD

collect(): RDD[T]→seq[T]  dangerous call, because you say to bring all these data in the master, so it may run out of memory. Use if you know all data fit in memory

take(k): RDD[T]→seq[T]

first(): RDD[T]→T

count(): RDD[T]→Long

countByValue(): RDD[T]→seq[(T,Long)]

reduce(f:(T,T)→T): RDD[T]→T

foreach(f:T->U): RDD[T]→ - (executes in the workers)

Added actions on pair RDDs

countByKey(): RDD[(K,V)]→seq[(K,Long)]

lookup(k: K): RDD[(K,V)]→seq[V]

the result of an action is not an RDD. The results of transformations is an RDD.

So, we cannot chain actions.

Activity

- *Objective: Understand Spark operations*
- *Tasks:*
  1. *(10') Individually solve one exercise*
  2. *(20') Explain the solution to the others*
  3. *Hand in the three solutions*

- *Roles for the team-mates during task 2:*
  a) *Explains his/her material*
  b) *Asks for clarification of blur concepts*
  c) *Mediates and **controls time***

INTERNALS

Parallelism

- Degree is automatically inferred from partitions
- Too few parallelism wastes resources
- Too much parallelism may generate significant overheads

e.g. if you have only 4 partitions and 4 machines, maybe they will all go into 1 machine.

creating a partition can be expensive, network traffic.

Partitioning

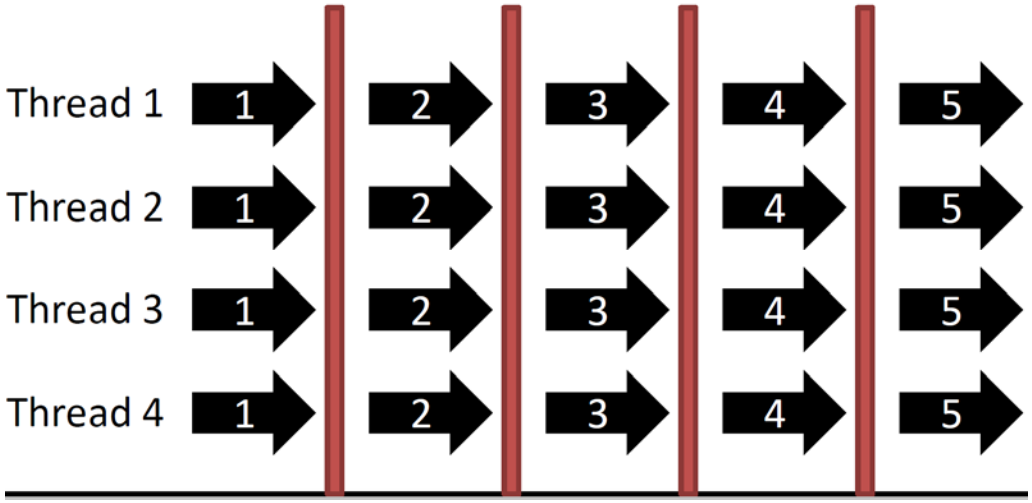
- Initially based on data locality
  - Useful based on keys
    - Hash
      - *partitionBy*
      - *groupByKey*
    - Range
      - *sortByKey*
- Partitions kept in workers' memory
- Different RDDs can use the same key
  - Similar to vertical partitioning
- Transformations lose partitioning information
  - *mapValues* retains partitioning information

The problema of parallelism

Theory

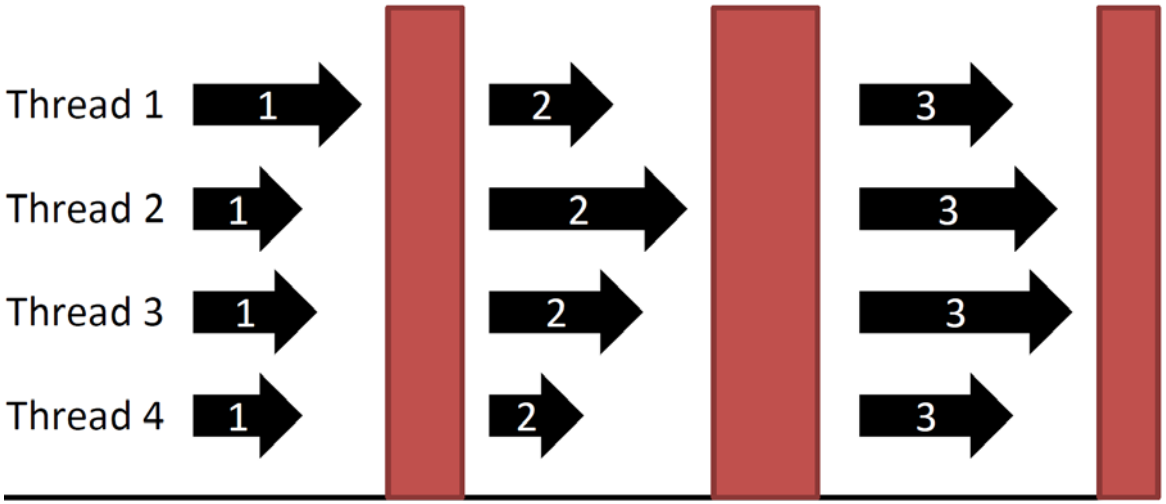
Samuel Yee

Bulk Synchronous Parallel Model



Ideal

Real



SAILING lab slides

Optimization

- Lineage graph is translated into a physical execution graph:
  - Truncate graph to use cached results
  - Pipeline or collapse several RDD into one stage
    - If no data movement needed
- Decompose one job into several stages
  - Stages are decomposed into tasks per partition
    - Each task has three phases:
      1. Fetch data
      2. Execute operation**s**
      3. Write result for shuffling/returning results to driver

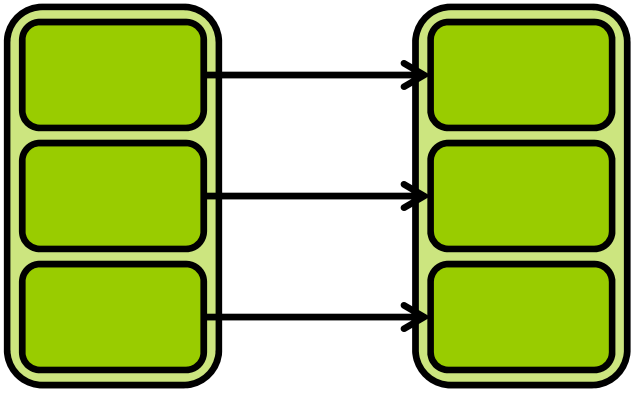
using the persisted RDDs

task= stage executed in a partition

within task happens in-memory processing and outputs to disk

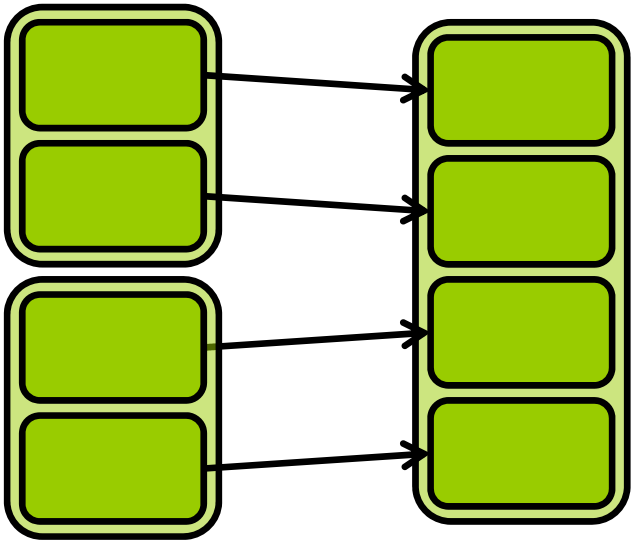
operations/transformations don't go to the disk (moving from one stage to another). Sync barrier is the disk. In between everything happens in memory.

Narrow Dependencies



MapValue/Filter

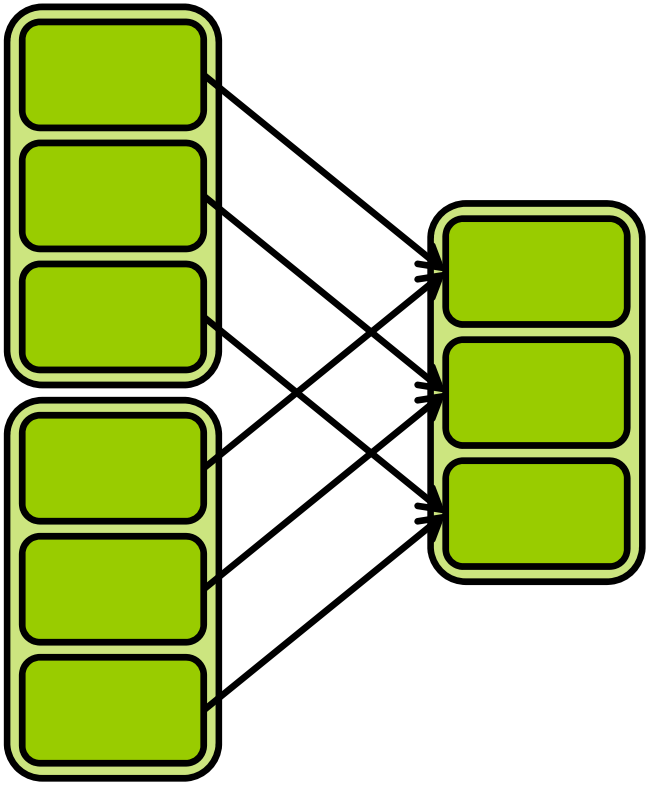
because they don't mofigy the key



Union

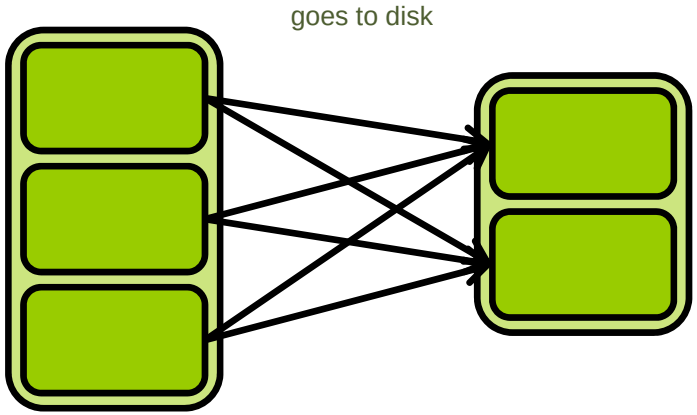
arrows don't cross

i know those 2 will go to the first and only them.

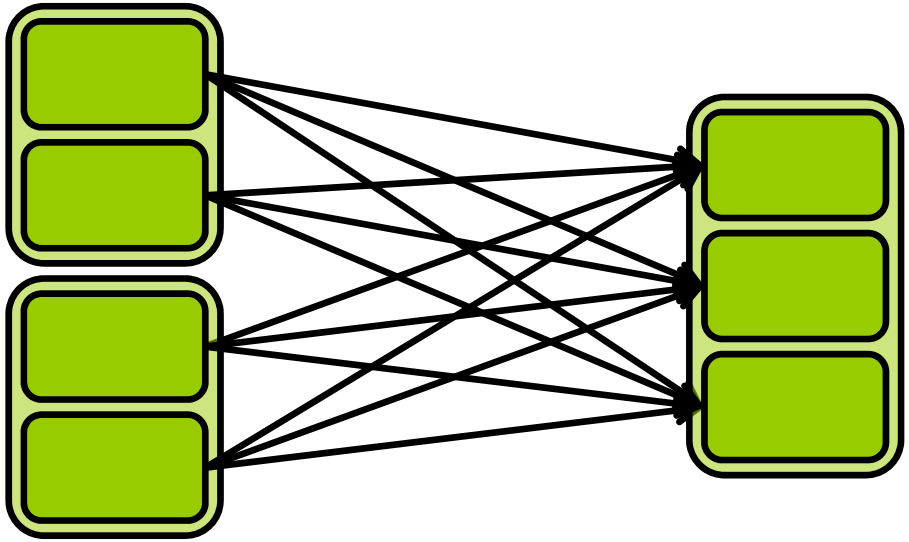


Join with inputs co-partitioned

Wide Dependencies



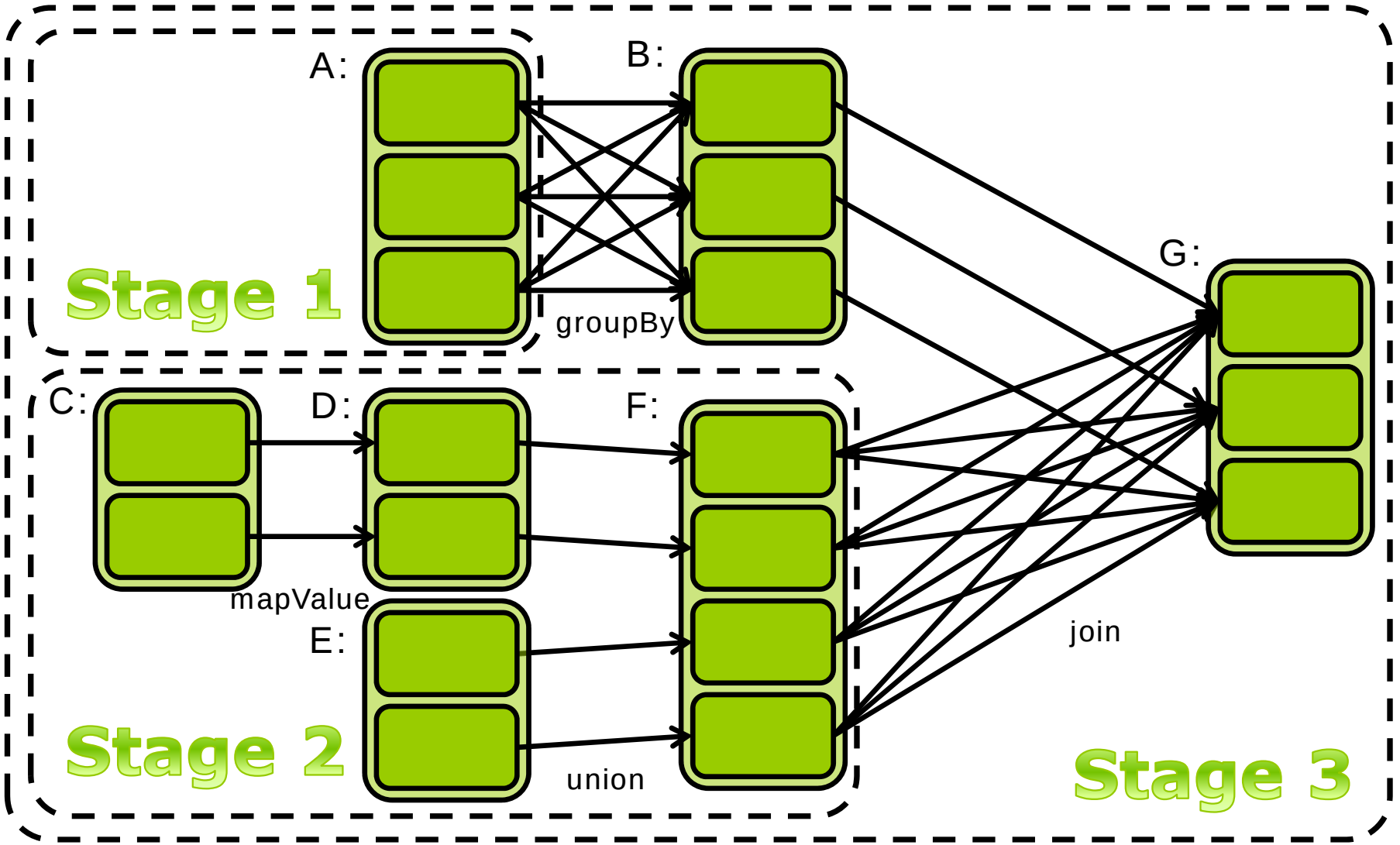
groupByKey



Join with inputs not co-partitioned

Scheduling

stage: transformations that can happen inside the machine (no disk access)
so, every new wide dependency need a new stage.



Memory (default) usage

- 60% RDD storage
- 20% Shuffle and aggregation buffers
- 20% User code and variables

RDD Abstraction Representation

where does the 60% go

- A set of dependencies on parent RDDs
- A function for computing the dataset
- Partitioning schema/metadata
  - Hash
  - Range
- A set of partitions
- Data placement
  - Partitions per node

Persistence

- Does not force evaluation (is not an action)
- Options:
  - In-memory
    - Deserialized (more performance-more memory)
      - I.e., caching
    - Serialized (less performance-less memory)
  - On-disk serialized (free memory)
- Set persistence priority in the RDD
  - Partition based
  - LRU RDD eviction policy
    - In case of only one RDD, MRU partition is removed
- Support for checkpointing
  - For long lineage graphs with wide dependencies
- Per Spark instance
  - Not multi-tenant unified memory management

Recovery

- An RDD has enough information to be reconstructed after a failure
  - Lineage graph (logging not needed)
- Data can be cached/persisted in two nodes
  - Orthogonal to persistency options

spark: share nothing architecture

Shared variables

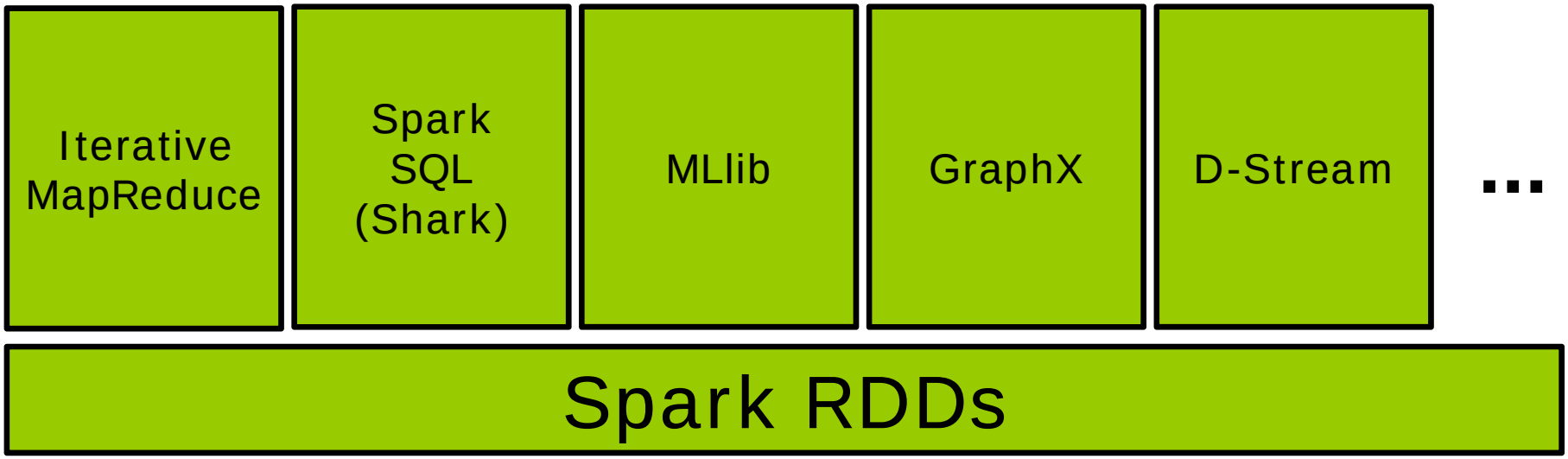
two options to share data

variable communication is always between driver & workers.

- Broadcast variables — from driver -> worker
  - Usage
    - Passed as a serializable object to the context
    - Accessed by workers (read-only)
  - Guarantees
    - The value is sent only once to each worker
- Accumulators — from worker -> driver
  - Usage
    - Initialized by the driver
    - Incremented by workers (write-only)
    - Value accessed by driver
  - Guarantees
    - Consistent inside actions
    - Unpredictable result inside transformations
      - In case of rexecution

if you put accumulator inside of transformation, its value may be higher than reality, bcoz when transformation fails its re-executed. So the accumulator would be executed every time (more times that it should).

Abstractions



CLOSING

Summary

- Resilient Distributed Datasets
  - Architecture
  - Operations
    - Transformations
    - Actions
  - Persisting
  - Dependencies
  - Scheduling
  - Partitioning
- Abstractions

Bibliography

- H. Karau et al.: *Learning Spark*. O'Really, 2015
- M. Zaharia: *An Architecture for Fast and General Data Processing on Large Clusters*. ACM Books, 2016