

CS162
Operating Systems and
Systems Programming
Lecture 24

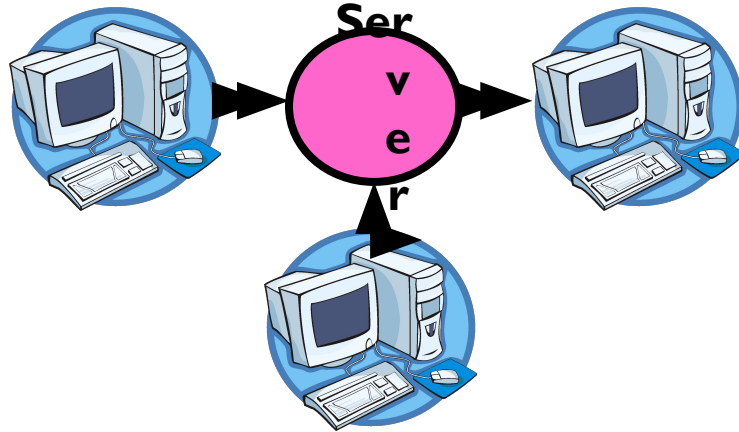
Distributed I: Transactions,
Distributed Decision Making, 2PC,
Apache Spark

November 26th, 2024

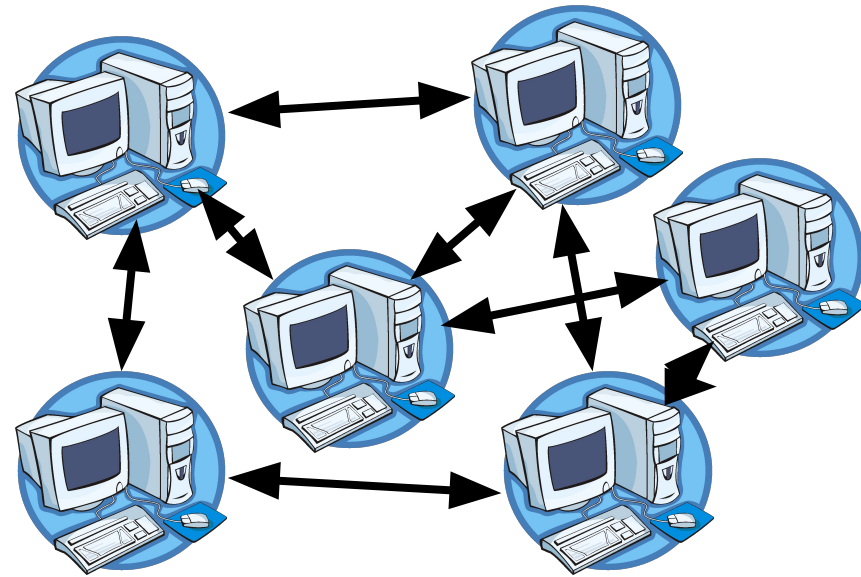
Prof. Ion Stoica

<http://cs162.eecs.Berkeley.edu>

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer Model

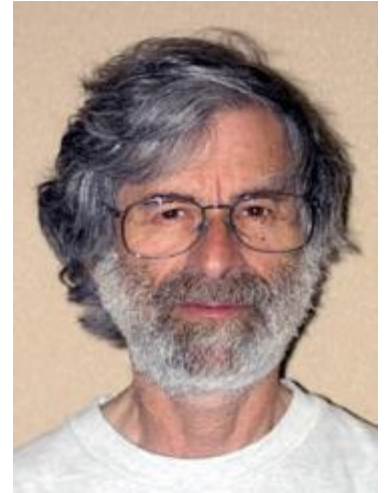
- **Centralized System:** major functions performed by a single physical computer
 - Originally, everything on single computer
 - Later: client/server model
- **Distributed System:** physically separate computers working together on task
 - Early model: multiple servers working together
 - » Probably in the same room or building
 - » Often called a “cluster”
 - Later models: peer-to-peer/wide-spread collaboration

Distributed Systems: Motivation/Issues/Promise

- Why do we want distributed systems?
 - Cheaper and easier to build lots of simple computers
 - More resources (such as cluster of GPUs for training)
 - Easier to add resources incrementally
 - Users can have complete control over some components
 - Collaboration: much easier for users to collaborate through network resources (such as network file systems)
- The *promise* of distributed systems:
 - *Higher availability*: one machine goes down, use another
 - *Better durability*: store data in multiple locations
 - *More security*: each piece easier to make secure

Distributed Systems: Reality

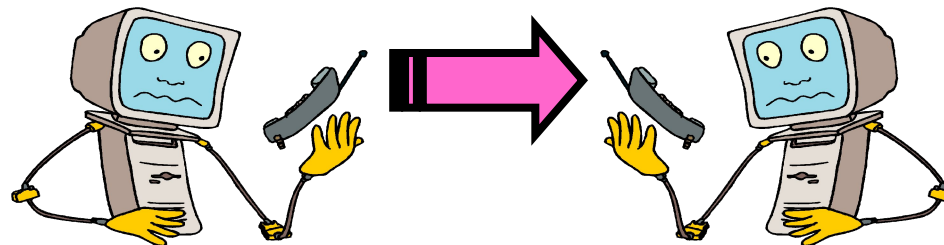
- Reality has been disappointing
 - *Worse availability*: depend on every machine being up
 - » Lamport: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”
 - *Worse reliability*: can lose data if any machine crashes
 - *Worse security*: anyone in world can break into system
- Coordination is more difficult
 - Must coordinate multiple copies of shared state information
 - What would be easy in a centralized system becomes a lot more difficult
- Trust/Security/Privacy/Denial of Service
 - Many new variants of problems arise as a result of distribution
 - Can you trust the other members of a distributed application enough to even perform a protocol correctly?
 - Corollary of Lamport’s quote: “A distributed system is one where you can’t do work because some computer you didn’t even know existed is successfully coordinating an attack on my system!”



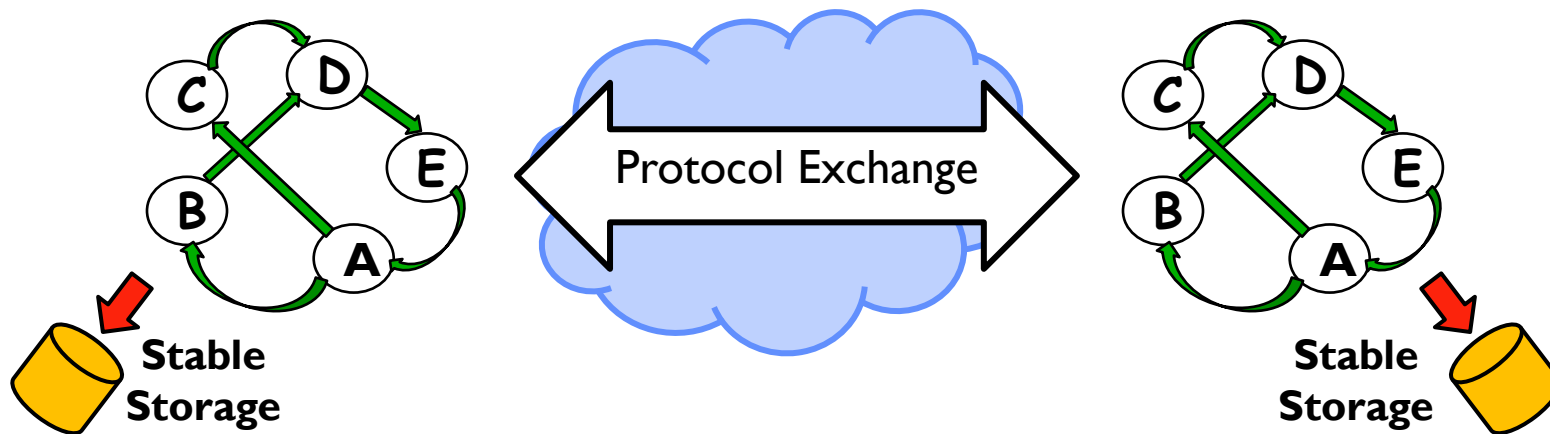
Leslie Lamport

Distributed Systems: Goals/Requirements

- **Abstraction:** the ability of the system to mask its complexity behind a simple interface
- Possible abstractions:
 - **Location:** Can't tell where resources are located
 - **Migration:** Resources may move without the user knowing
 - **Replication:** Can't tell how many copies of resource exist
 - **Concurrency:** Can't tell how many users there are
 - **Parallelism:** System may speed up large jobs by splitting them into smaller pieces
 - **Fault Tolerance:** System may hide various things that go wrong
- Transparency and collaboration require some way for different processors to communicate with one another



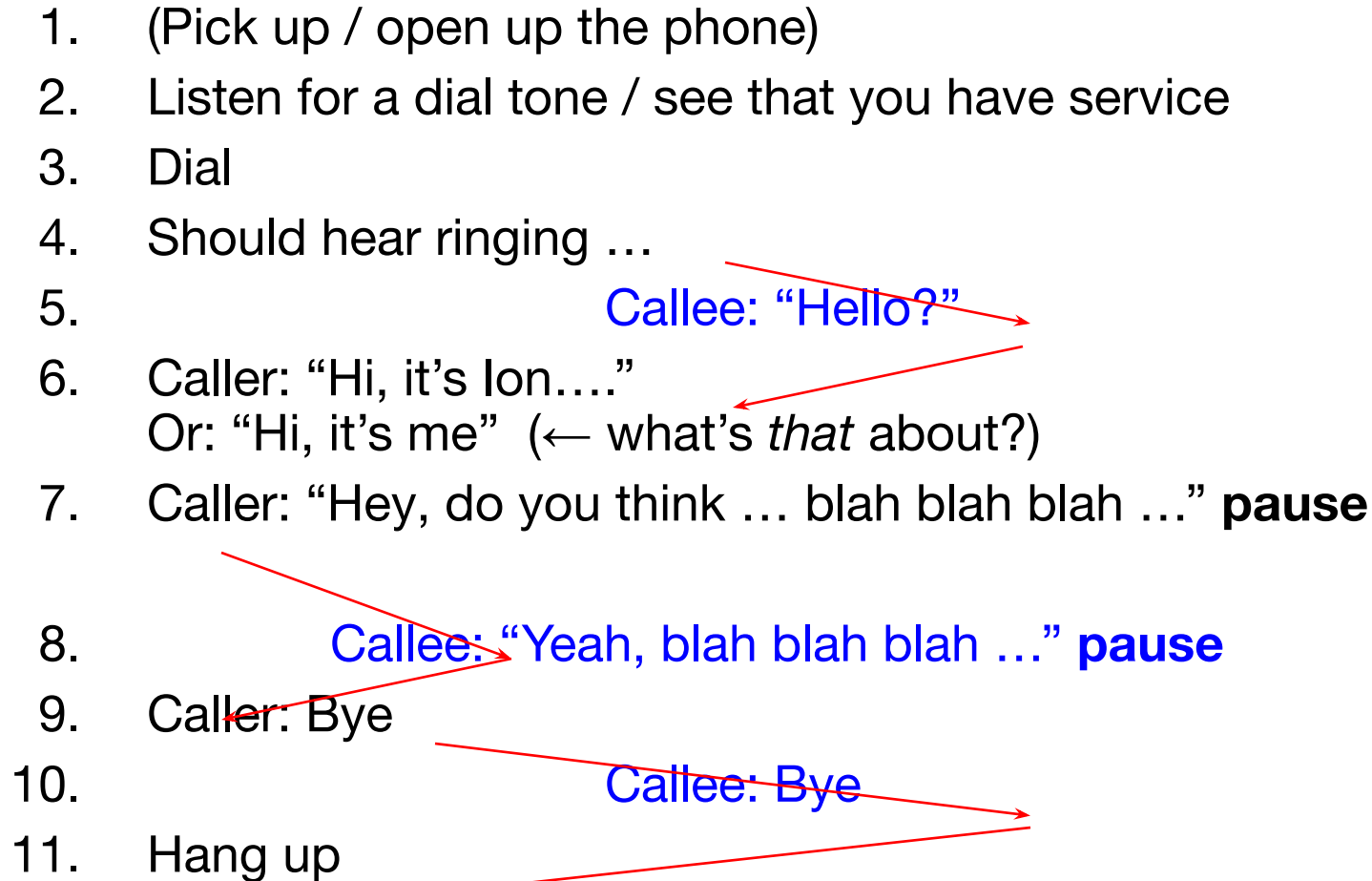
How do entities communicate? A Protocol!



- A protocol is **an agreement on how to communicate**, including:
 - **Syntax**: how a communication is specified & structured
 - » Format, order messages are sent and received
 - **Semantics**: what a communication means
 - » Actions taken when transmitting, receiving, or when a timer expires
- Described formally by a state machine
 - Often represented as a message transaction diagram
 - Can be a partitioned state machine: two parties synchronizing duplicate sub-state machines between them
 - Stability in the face of failures!

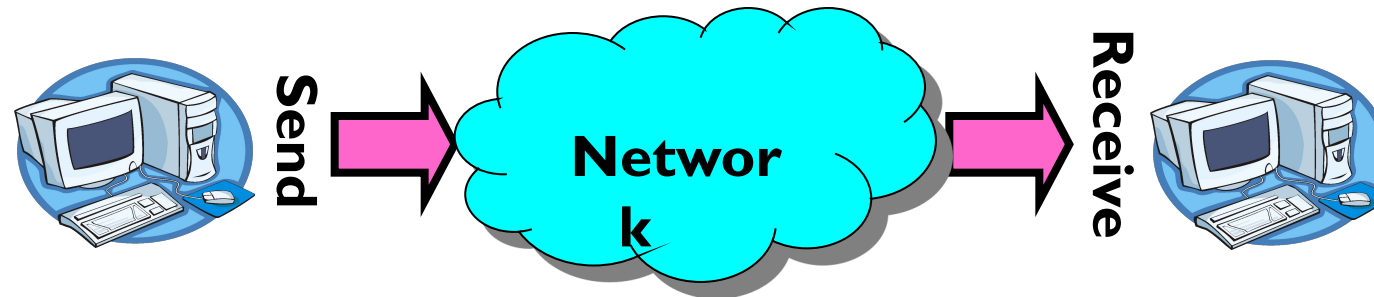
Examples of Protocols in Human Interactions

- Telephone

1. (Pick up / open up the phone)
 2. Listen for a dial tone / see that you have service
 3. Dial
 4. Should hear ringing ...
 5. Callee: "Hello?"
 6. Caller: "Hi, it's Ion...."
Or: "Hi, it's me" (← what's *that* about?)
 7. Caller: "Hey, do you think ... blah blah blah ..." **pause**
 8. Callee: "Yeah, blah blah blah ..." **pause**
 9. Caller: Bye
 10. Callee: Bye
 11. Hang up
- 

Distributed Applications

- How do you actually program a distributed application?
 - Need to synchronize multiple threads, running on different machines
 - » No shared memory, so cannot use test&set



- One Abstraction: send/receive messages
 - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
 - Mailbox (mbox): temporary holding area for messages
 - » Includes both **destination location** and **queue**
 - » Over Internet, **destination** specified by **IP address** and **Port** (Recall Web server example!)
 - Send(message,mbox)
 - » Send message to remote mailbox identified by mbox
 - Receive(buffer,mbox)
 - » Wait until mbox has message, copy into buffer, and return
 - » If threads sleeping on this mbox, wake up one of them

Using Messages: Send/Receive behavior

- When should `send(message, mbox)` return?
 - When receiver gets message? (i.e. ack received)
 - When message is safely buffered on destination?
 - Right away, if message is buffered on source node?
- Actually two questions here:
 - When can the sender be sure that receiver actually received the message?
 - When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from $T1 \rightarrow T2$
 - $T1 \rightarrow \text{buffer} \rightarrow T2$
 - Very similar to producer/consumer
 - » $\text{Send} = V, \text{Receive} = P$
 - » However, can't tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

Producer:

```
int msgl[1000];  
while(1) {  
    prepare message;  
    send(msgl,mbox);  
}
```



**Send
Message**

Consumer:

```
int buffer[1000];  
while(1) {  
    receive(buffer,mbox);  
    process message;  
}
```



**Receive
Message**

- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
 - This is one of the roles of the window in TCP: window is size of buffer on far end
 - Restricts sender to forward only what will fit in buffer

Messaging for Request/Response communication

- What about two-way communication?
 - Request/Response
 - » Read a file stored on a remote machine
 - » Request a web page from a remote web server
 - Also called: **client-server**
 - » Client \equiv requester, Server \equiv responder
 - » Server provides “service” (file storage) to the client
- Example: File service

Client: (requesting the file)
`char response[1000];`

`send("read rutabaga", server_mbox);`
`receive(response, client_mbox);`

Server: (responding with the file)
`char command[1000], answer[1000];`

`receive(command, server_mbox);`
`decode command;`
`read file into answer;`
`send(answer, client_mbox);`

**Request
File**

**Get
Response**

**Receive
Request**

**Send
Response**

Distributed Consensus Making

- Consensus problem
 - All nodes propose a value
 - Some nodes might crash and stop responding
 - Eventually, all remaining nodes decide on the same value from set of proposed values
- Distributed Decision Making
 - Choose between “true” and “false”
 - Or Choose between “commit” and “abort”
- Equally important (but often forgotten!): make it durable!
 - How do we make sure that decisions cannot be forgotten?
 - » This is the “D” of “ACID” in a regular database
 - In a global-scale system?
 - » What about erasure coding or massive replication?
 - » Like **BlockChain** applications!

General's Paradox

- General's paradox:

- Constraints of problem:

- » Two generals, on separate mountains
 - » Can only communicate via messengers
 - » Messengers can be captured

- Problem: need to coordinate attack

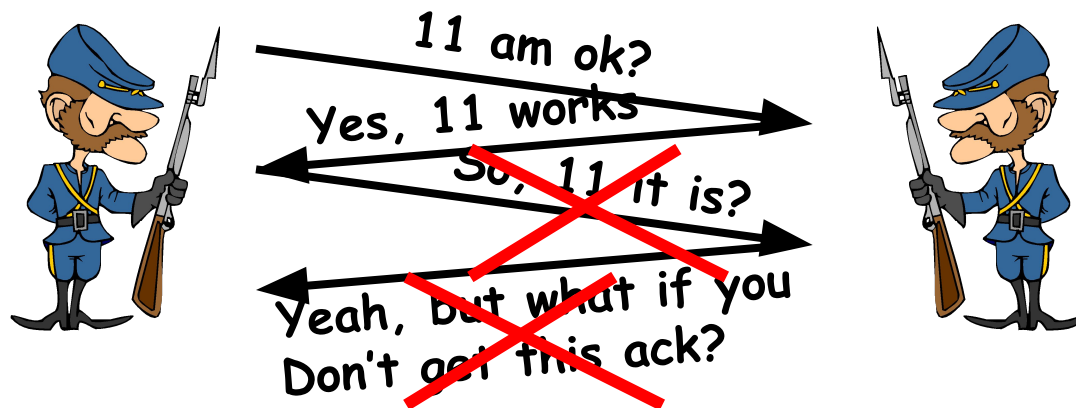
- » If they attack at different times, they all lose
 - » If they attack at same time, they win

- Named after Custer, who died at Little Big Horn because he arrived a couple of days too early



General's Paradox (con't)

- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
 - Remarkably, “no”, even if all messages get through



- No way to be sure last message gets through!
 - In real life, use radio for simultaneous (out of band) communication
- So, clearly, we need something other than simultaneity!

Two-Phase Commit

- Since we can't solve the General's Paradox (i.e. simultaneous action), let's solve a related problem
- **Distributed transaction**: Two or more machines agree to do something, or not do it, **atomically**
 - No constraints on time, just that it will eventually happen!
- **Two-Phase Commit protocol**: Developed by Turing award winner Jim Gray
 - (first Berkeley CS PhD, 1969)
 - Many important Database breakthroughs also from Jim Gray



Jim Gray

Two-Phase Commit Protocol

- **Persistent stable log on each machine:** keep track of whether commit has happened
 - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
- **Prepare Phase:**
 - The global coordinator requests that all participants will promise to commit or **rollback** the **transaction**
 - Participants record promise in log, then acknowledge
 - If anyone votes to abort, coordinator writes "Abort" in its log and tells everyone to abort; each records "Abort" in log
- **Commit Phase:**
 - After all participants respond that they are prepared, then the coordinator writes "Commit" to its log
 - Then asks all nodes to commit; they respond with ACK
 - After receive ACKs, coordinator writes "Got Commit" to log
- Log used to guarantee that all machines either commit or don't

2PC Algorithm

- One coordinator
- N workers (replicas)
- High level algorithm description:
 - Coordinator asks all workers if they can commit
 - If all workers reply “**VOTE-COMMIT**”, then coordinator broadcasts “**GLOBAL-COMMIT**”
Otherwise, coordinator broadcasts “**GLOBAL-ABORT**”
 - Workers obey the **GLOBAL** messages
- Use a persistent, stable log on each machine to keep track of what you are doing
 - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash

Two-Phase Commit: Setup

- One machine (*coordinator*) initiates the protocol
- It asks *every* machine to **vote** on transaction
- Two possible votes:
 - **Commit**
 - **Abort**
- Commit transaction only if unanimous approval

Two-Phase Commit: Preparing

Worker Agrees to Commit

- Machine has **guaranteed** that it will accept transaction
- Must be **recorded in log** so machine will remember this decision if it fails and restarts

Worker Agrees to Abort

- Machine has **guaranteed** that it will **never accept** this transaction
- Must be **recorded in log** so machine will remember this decision if it fails and restarts

Two-Phase Commit: Finishing

Commit Transaction

- Coordinator learns *all machines have agreed to commit*
- Record decision to commit in local log
- Apply transaction, inform voters

Abort Transaction

- Coordinator learns *at least one machine has voted to abort*
- Record decision to abort in local log
- Do not apply transaction, inform voters

Two-Phase Commit: Finishing

Commit Transaction

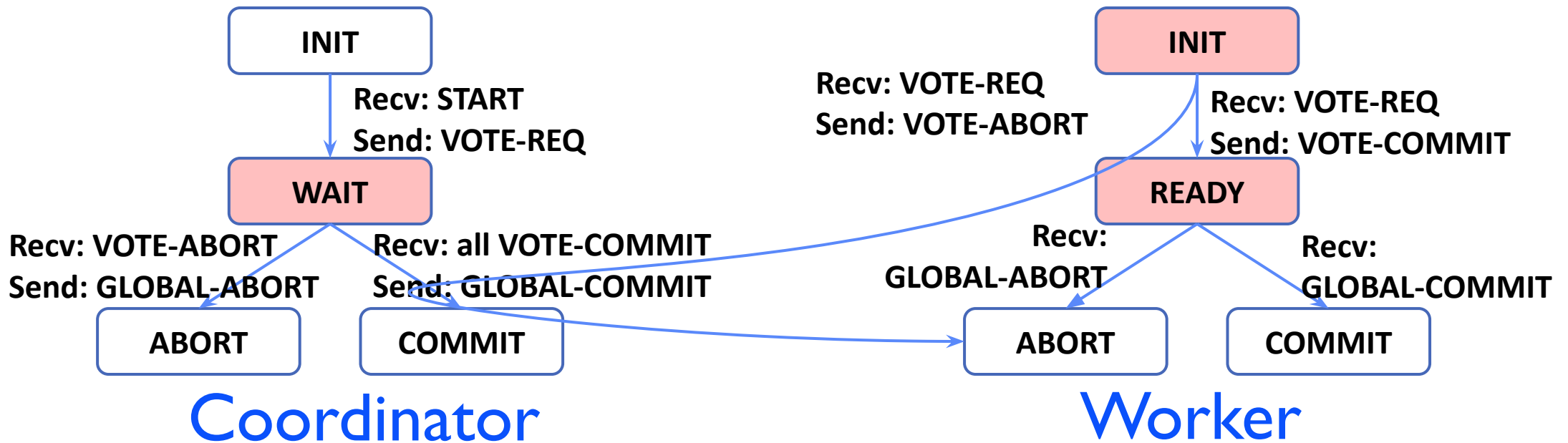
- Coordinator learns *all machines have agreed to commit*
- Record decision to commit in local log
- Apply transaction, inform voters

Abort Transaction

- Coordinator learns *at least one machine has voted to abort*
- Record decision to abort in local log
- Do not apply transaction, inform voters

Because no machine can take back its decision, exactly one of these will happen

State Machine Description of 2PC



- Two Phase Commit (2PC) can be described with interacting state machines
- Coordinator only waits for votes in “WAIT” state
 - In WAIT, if doesn't receive N votes, it times out and sends GLOBAL-ABORT
- Worker waits for VOTE-REQ in INIT
 - Worker can time out and abort (coordinator handles it)
- Worker waits for GLOBAL-* message in READY
 - Coordinator fails \Rightarrow workers BLOCK waiting for coordinator to recover and send GLOBAL_* message

Detailed Algorithm

Coordinator Algorithm

Coordinator sends **VOTE-REQ** to all workers

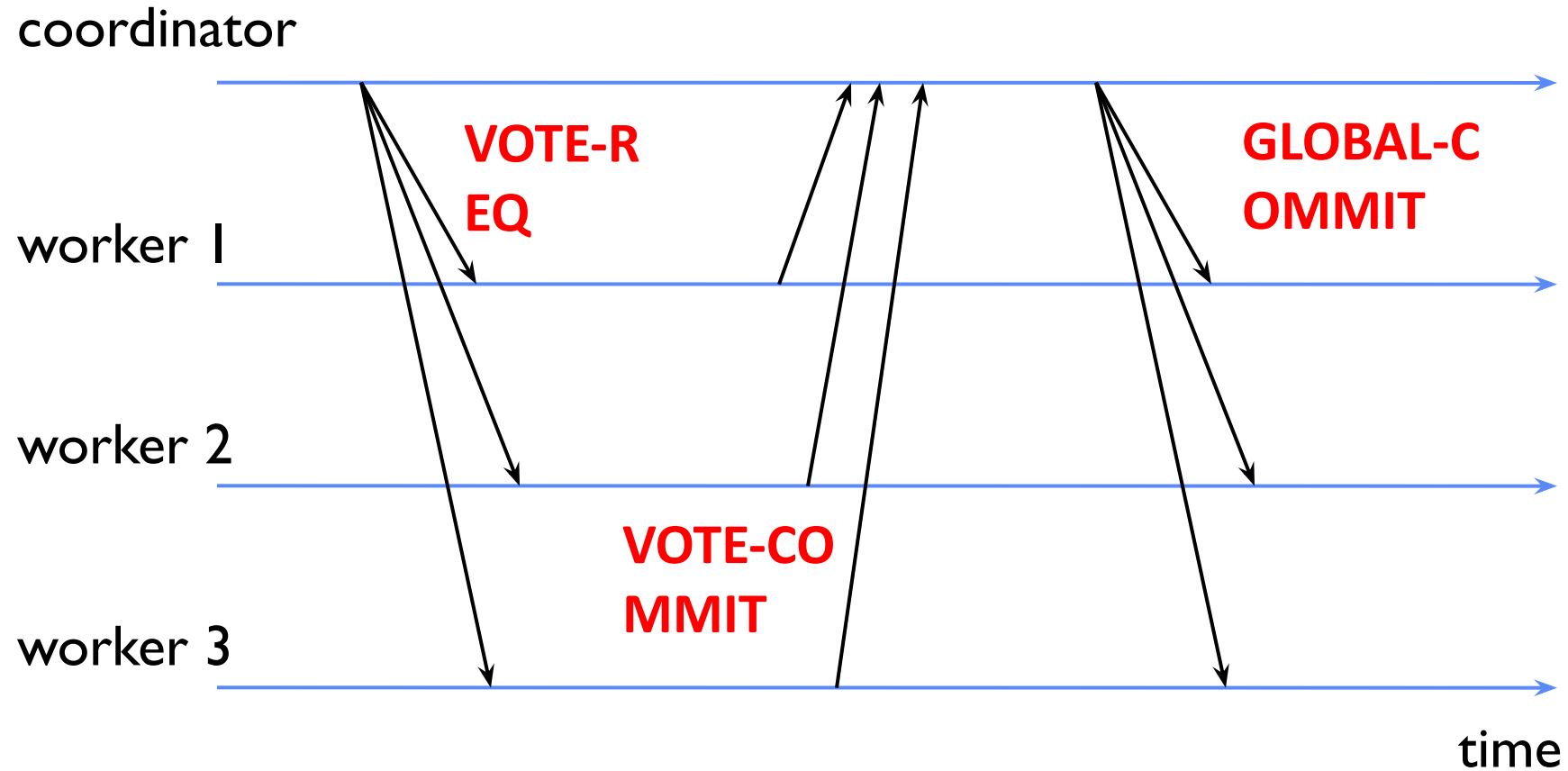
- If receive **VOTE-COMMIT** from all N workers, send **GLOBAL-COMMIT** to all workers
- If don't receive **VOTE-COMMIT** from all N workers, send **GLOBAL-ABORT** to all workers

Worker Algorithm

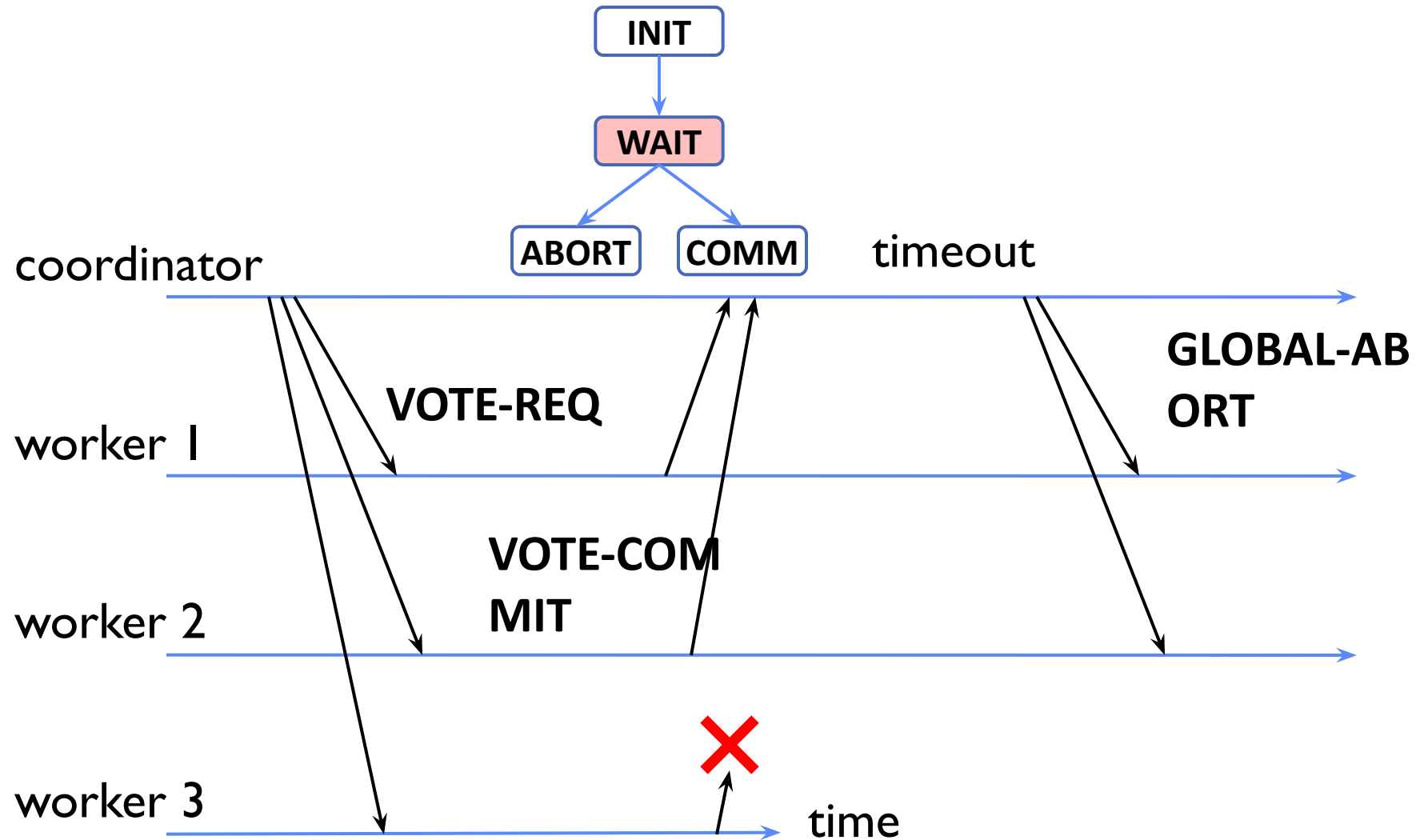
- Wait for **VOTE-REQ** from coordinator
- If ready, send **VOTE-COMMIT** to coordinator
- If not ready, send **VOTE-ABORT** to coordinator
 - And immediately abort

- If receive **GLOBAL-COMMIT** then commit
- If receive **GLOBAL-ABORT** then abort

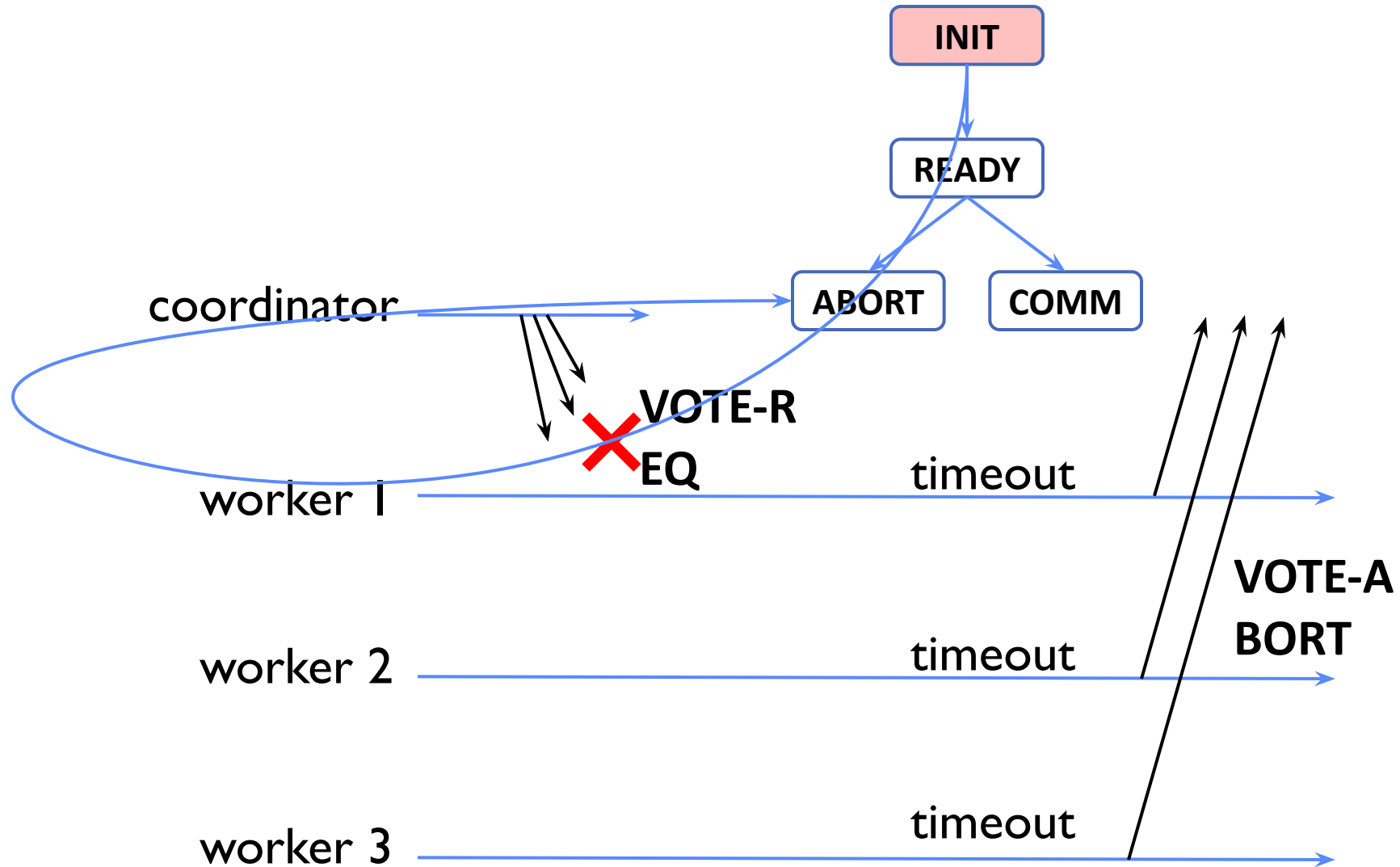
Failure Free Example Execution



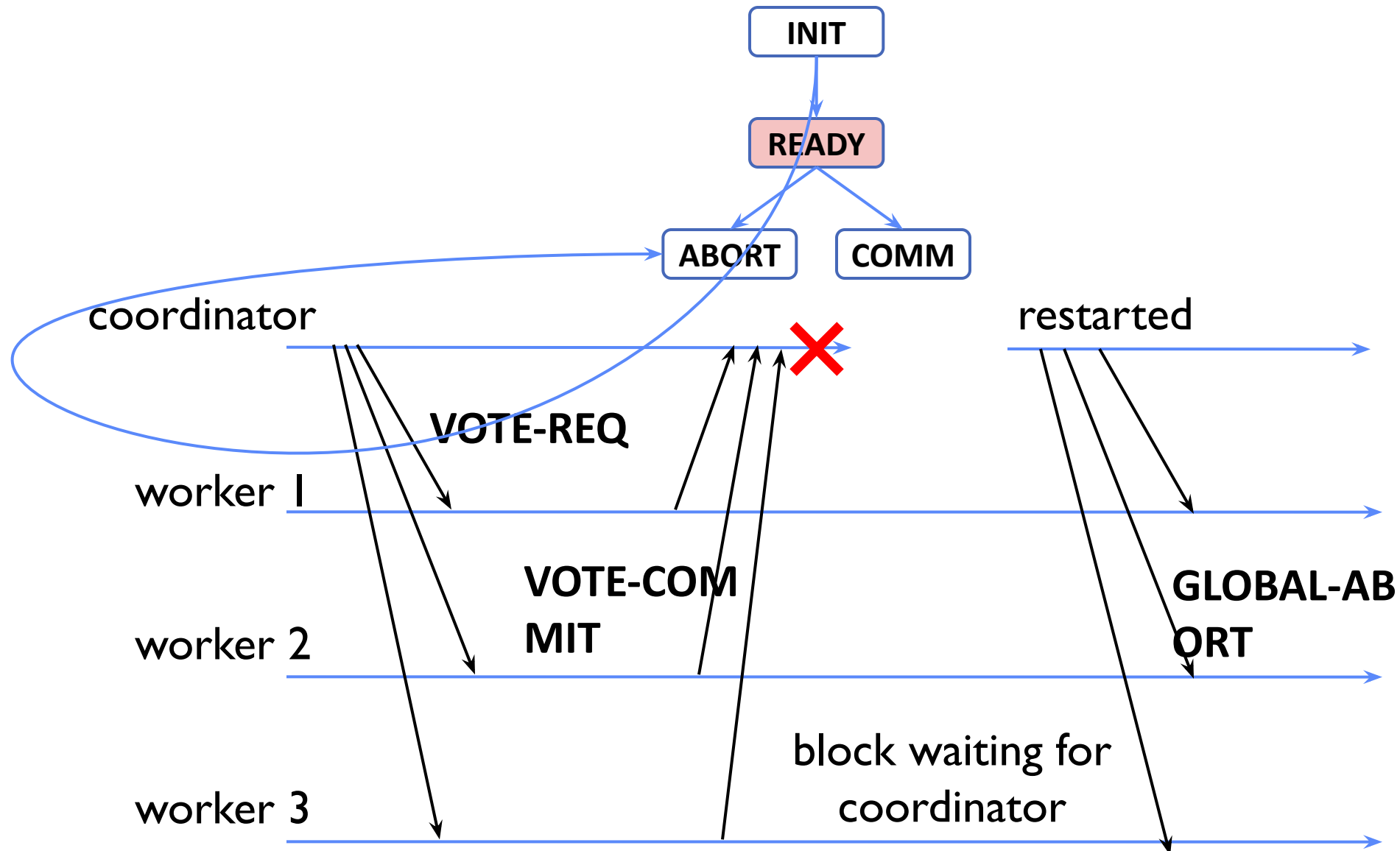
Example of Worker Failure



Example of Coordinator Failure #1



Example of Coordinator Failure #2



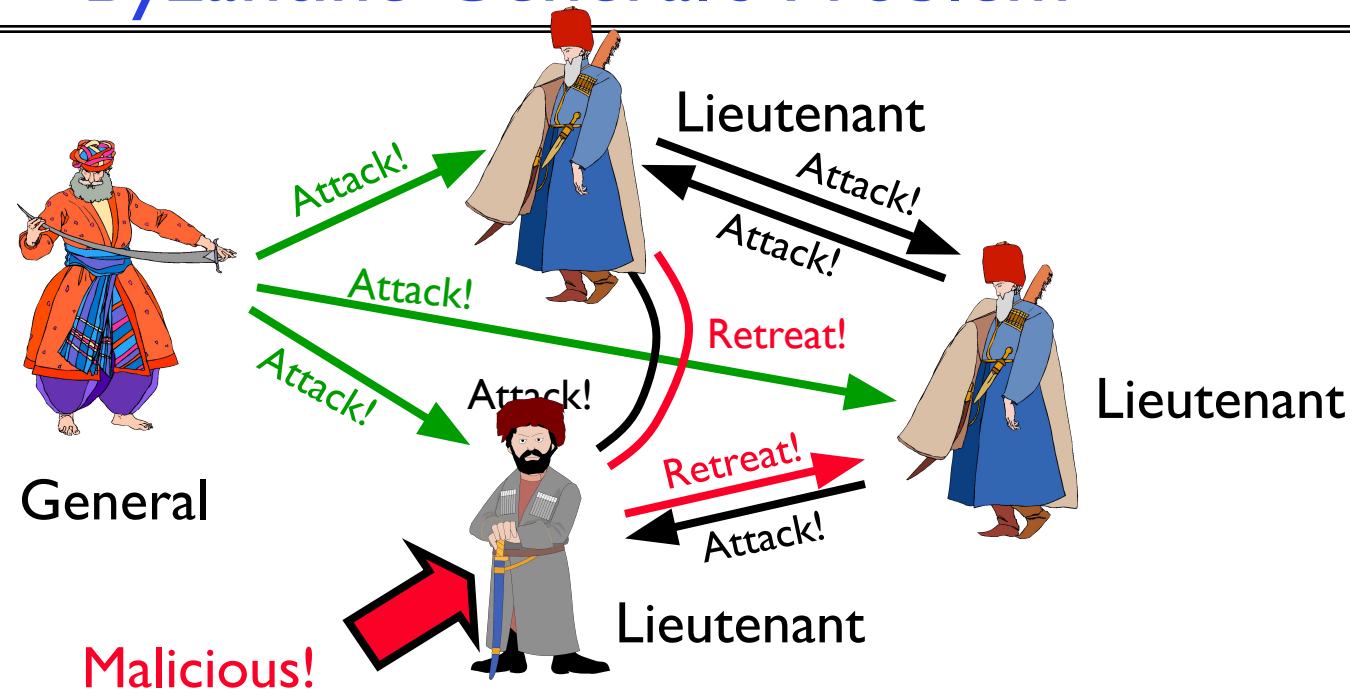
Durability

- All nodes use **stable storage** to store current state
 - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.
 - E.g.: SSD, NVRAM
- Upon recovery, nodes can restore state and resume:
 - Coordinator **aborts** in INIT, WAIT, or ABORT
 - Coordinator **commits** in COMMIT
 - Worker **aborts** in INIT, ABORT
 - Worker **commits** in COMMIT
 - Worker **“asks”** Coordinator in READY

Alternatives to 2PC

- **Three-Phase Commit:** One more phase, allows nodes to fail or block and still make progress.
- **PAXOS:** An alternative used by Google and others that does not have 2PC blocking problem
 - Develop by Leslie Lamport (Turing Award Winner)
 - No fixed leader, can choose new leader on fly, deal with failure
 - Some think this is extremely complex!
- **RAFT:** PAXOS alternative from John Osterhout (Stanford)
 - Simpler to describe complete protocol
- What happens if one or more of the nodes is malicious?
 - **Malicious:** attempting to compromise the decision making
 - Use a more hardened decision-making process:
Byzantine Agreement and **Block Chains**

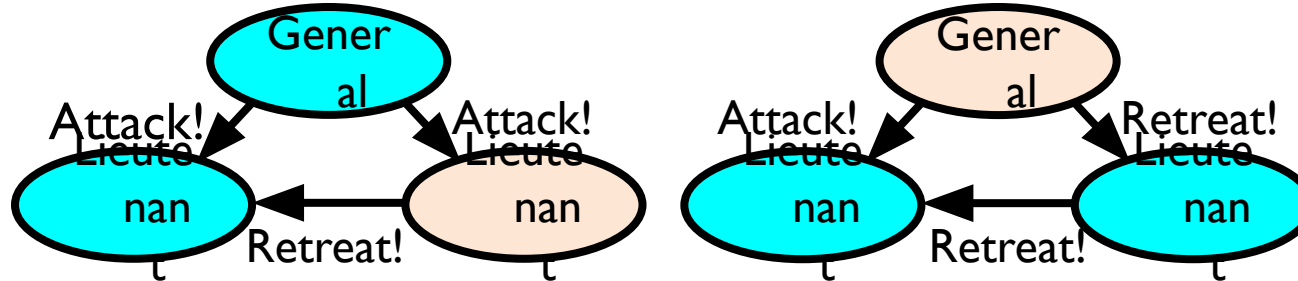
Byzantine General's Problem



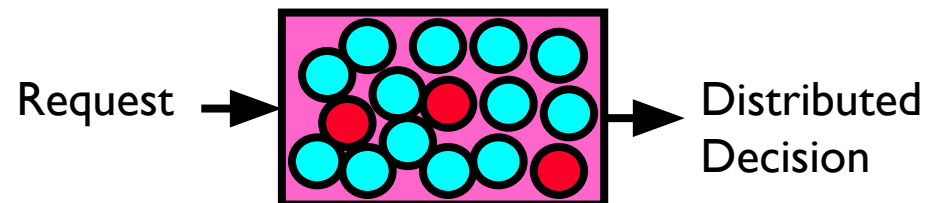
- Byzantine General's Problem (n players):
 - One General and $n-1$ Lieutenants
 - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his $n-1$ lieutenants such that the following Integrity Constraints apply:
 - IC1: All loyal lieutenants obey the same order
 - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

Byzantine General's Problem (con't)

- Impossibility Results:
 - Cannot solve Byzantine General's Problem with $n=3$ because one malicious player can mess up things



- With f faults, need $n > 3f$ to solve problem
- Various algorithms exist to solve problem
 - Original algorithm has #messages exponential in n
 - Subsequent algorithms have message complexity $O(n^2)$
 - » Example: Castro and Liskov, 1999
- Use of BFT (Byzantine Fault Tolerance) algorithm
 - Allow multiple machines to make a coordinated decision even if some subset of them ($< n/3$) are malicious



Summary

- A protocol is **an agreement on how to communicate**, including:
 - **Syntax**: how a communication is specified & structured
 - » Format, order messages are sent and received
 - **Semantics**: what a communication means
 - » Actions taken when transmitting, receiving, or when a timer expires
- Consensus problem
 - All nodes propose a value
 - Some nodes might crash and stop responding
 - Eventually, all remaining nodes decide on the same value from set of proposed values
- Two-phase commit: a form of distributed decision making
 - First, make sure everyone guarantees they will commit if asked (prepare)
 - Next, ask everyone to commit
- **Byzantine General's Problem**: distributed decision making with malicious failures
 - One general, $n-1$ lieutenants: some number of them may be malicious (often " f " of them)
 - All non-malicious lieutenants must come to same decision
 - If general not malicious, lieutenants must follow general
 - Only solvable if $n \geq 3f + 1$

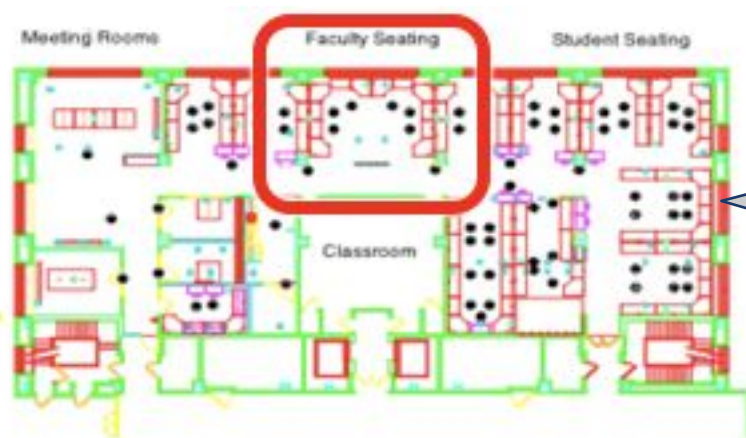
Announcements

- Homework 5: checkpoint due on Wednesday (11/27)
- Midterm 3: Thursday, 12/05
 - Everything fair game with focus on last 1/3 of class
 - Three *hand-written* cheat-sheets, double sided

2011-2017

Mission: Make sense of Big Data

- 8 faculty
- ~50 students and postdocs
- Bridge systems, ML, databases
- Collaborative environment
- Successor of RAD Lab (2005-2010)



ML, System, Database researchers,
working in the same open space

Berkeley
UNIVERSITY OF CALIFORNIA





Lester Mackey

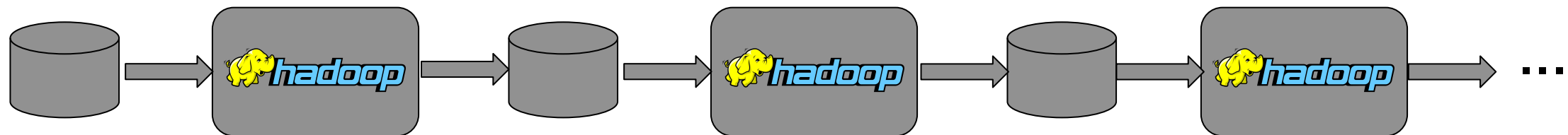


Prize

anonymized movie rating dataset

best recommendation algorithm

\$1m



Very slow: read/write to disk every iteration!

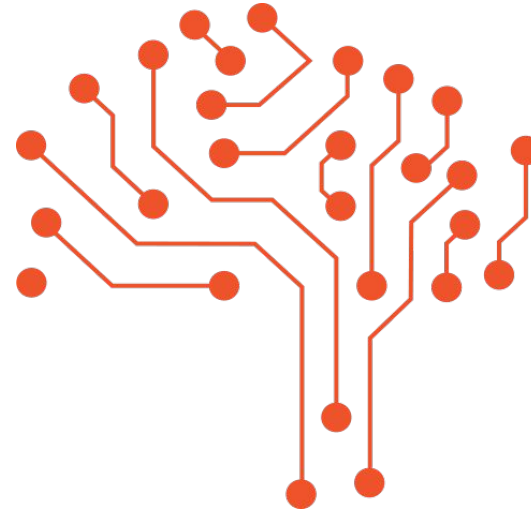
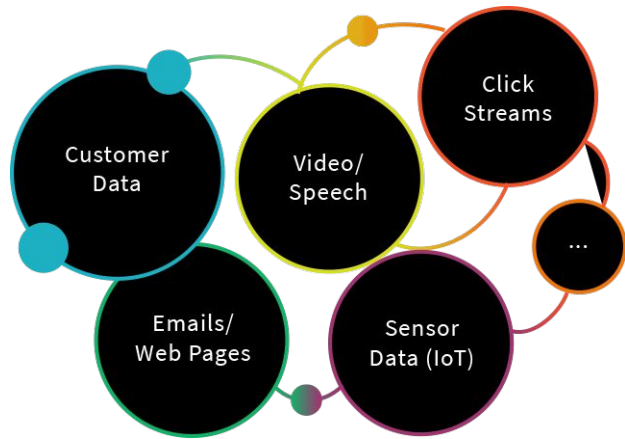


Lester Mackey



Matei Zaharia

The first unified analytics engine in 600 lines of code...



Big Data

Machine Learning



Much faster (store data in RAM) and flexible

Netflix Prize

COMPLETED

[Home](#) [Rules](#) [Leaderboard](#) [Update](#) [Download](#)

Leaderboard

Showing Test Score. [Click here to show quiz score](#)

Display top leaders.

tied for
best
score

20 mins
late

Rank	Team Name	Best Test Score	% Improvement	Best Submit Time
Grand Prize - RMSE = 0.8567 - Winning Team: BellKor's Pragmatic Chaos				
1	BellKor's Pragmatic Chaos	0.8567	10.06	2009-07-26 18:18:28
2	The Ensemble	0.8567	10.06	2009-07-26 18:38:22
3	Grand Prize Team	0.8582	9.90	2009-07-10 21:24:40
4	Opera Solutions and Vandelay United	0.8588	9.84	2009-07-10 01:12:31
5	Vandelay Industries !	0.8591	9.81	2009-07-10 00:32:20
6	PragmaticTheory	0.8594	9.77	2009-06-24 12:06:56
7	BellKor in BigChaos	0.8601	9.70	2009-05-13 08:14:09
8	Dace	0.8612	9.59	2009-07-24 17:18:43

Easy to Write Code

```
1 public class WordCount {
2     public static class TokenizerMapper
3         extends Mapper<Object, Text, Text, IntWritable> {
4
5         private final static IntWritable one = new IntWritable(1);
6         private Text word = new Text();
7
8         public void map(Object key, Text value, Context context
9             ) throws IOException, InterruptedException {
10             StringTokenizer itr = new StringTokenizer(value.toString());
11             while (itr.hasMoreTokens()) {
12                 word.set(itr.nextToken());
13                 context.write(word, one);
14             }
15         }
16     }
17
18     public static class IntSumReducer
19         extends Reducer<Text, IntWritable, Text, IntWritable> {
20         private IntWritable result = new IntWritable();
21
22         public void reduce(Text key, Iterable<IntWritable> values,
23             Context context
24             ) throws IOException, InterruptedException {
25             int sum = 0;
26             for (IntWritable val : values) {
27                 sum += val.get();
28             }
29             result.set(sum);
30             context.write(key, result);
31         }
32     }
33
34     public static void main(String[] args) throws Exception {
35         Configuration conf = new Configuration();
36         String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
37         if (otherArgs.length != 2) {
38             System.err.println("Usage: wordcount <input> <output>");
39             System.exit(2);
40         }
41         Job job = new Job(conf, "word count");
42         job.setJarByClass(WordCount.class);
43         job.setMapperClass(TokenizerMapper.class);
44         job.setCombinerClass(IntSumReducer.class);
45         job.setReducerClass(IntSumReducer.class);
46         job.setOutputKeyClass(Text.class);
47         job.setOutputValueClass(IntWritable.class);
48         for (int i = 0; i < otherArgs.length - 1; i++) {
49             FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
50         }
51         FileOutputFormat.setOutputPath(job,
52             new Path(otherArgs[otherArgs.length - 1]));
53         System.exit(job.waitForCompletion(true) ? 0 : 1);
54     }
55 }
```

```
1 val f = sc.textFile(inputPath)
2 val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
3 w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

WordCount in 50+ lines of Java MR

RDD: Resilient Distributed Datasets

- Collections of objects distr. across a cluster
 - Stored in RAM or on Disk
 - Automatically rebuilt on failure
- Operations
 - Transformations
 - Actions

Operations on RDDs

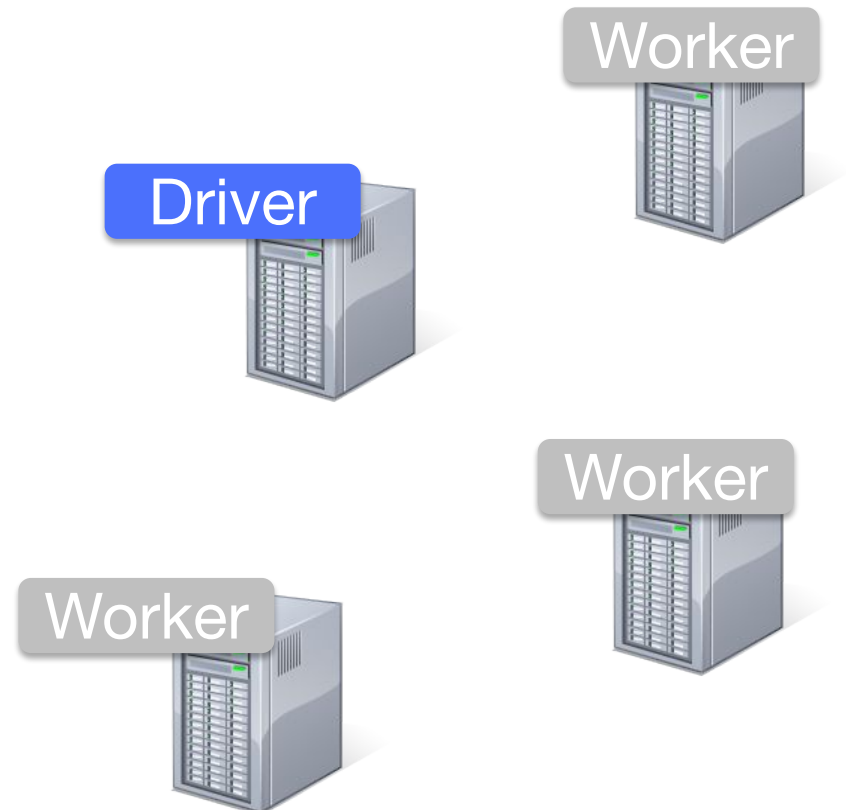
- Transformations $f(\text{RDD}) \Rightarrow \text{RDD}$
 - Lazy (not computed immediately)
 - E.g., “map”, “filter”, “groupBy”
- Actions:
 - Triggers computation
 - E.g. “count”, “collect”, “saveAsTextFile”

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

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



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Base RDD

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

Driver

Worker

Worker

Worker

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )  
errors = lines.filter(lambda s: s.startswith("ERROR"))
```

Driver

Worker

Worker

Worker

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Transformed RDD

```
lines = spark.textFile( "hdfs://..." )  
errors = lines.filter( lambda s: s.startswith("ERROR") )
```

Driver

Worker

Worker

Worker

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )
errors = lines.filter(lambda s: s.startswith("ERROR") )
messages = errors.map(lambda s: s.split("\t")[2] )
messages.cache()

messages.filter(lambda s: "mysql" in s ).count()
```

Driver

Worker

Worker

Worker

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )
errors = lines.filter(lambda s: s.startswith("ERROR") )
messages = errors.map(lambda s: s.split("\t")[2] )
messages.cache()

messages.filter(lambda s: "mysql" in s ).count()
```

Driver

Action

Worker

Worker

Worker

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )
errors = lines.filter(lambda s: s.startswith("ERROR") )
messages = errors.map(lambda s: s.split("\t")[2] )
messages.cache()

messages.filter(lambda s: "mysql" in s ).count()
```

Driver

Worker

Block 1

Worker

Block 2

Worker

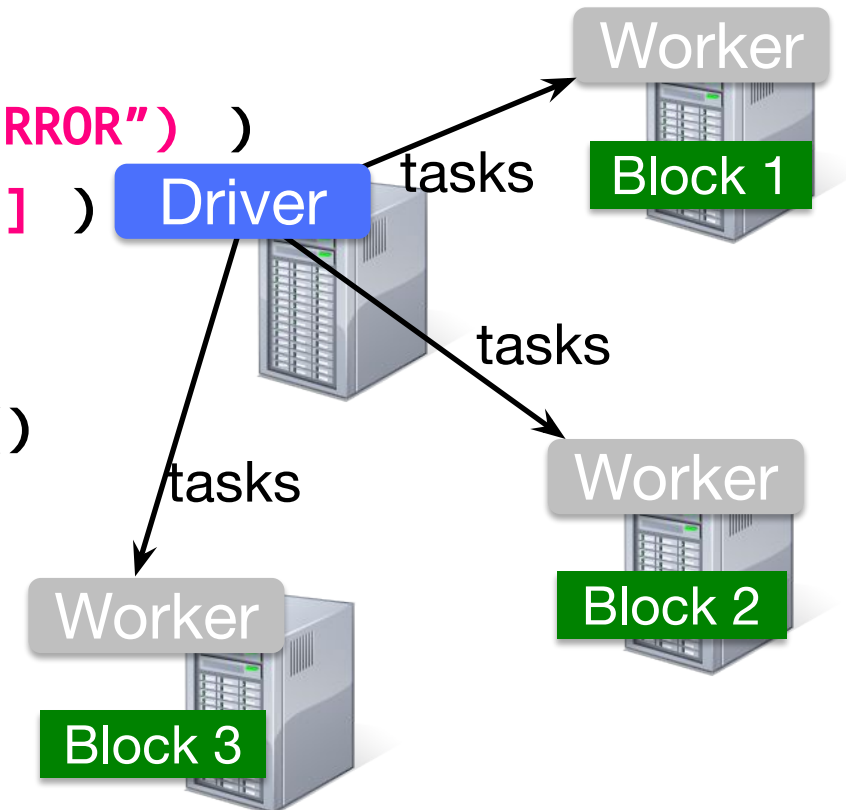
Block 3

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )
errors = lines.filter(lambda s: s.startswith("ERROR") )
messages = errors.map(lambda s: s.split("\t")[2] )
messages.cache()

messages.filter(lambda s: "mysql" in s ).count()
```



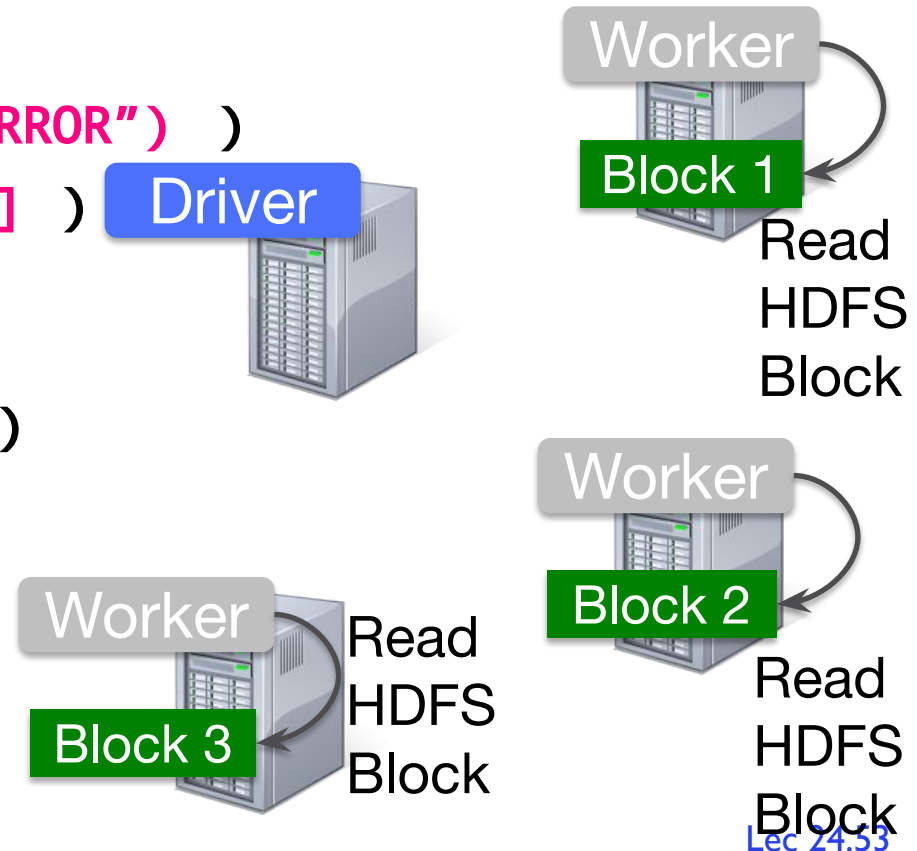
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )
errors = lines.filter(lambda s: s.startswith("ERROR") )
messages = errors.map(lambda s: s.split("\t")[2] )
messages.cache()

messages.filter(lambda s: "mysql" in s ).count()
```

Driver

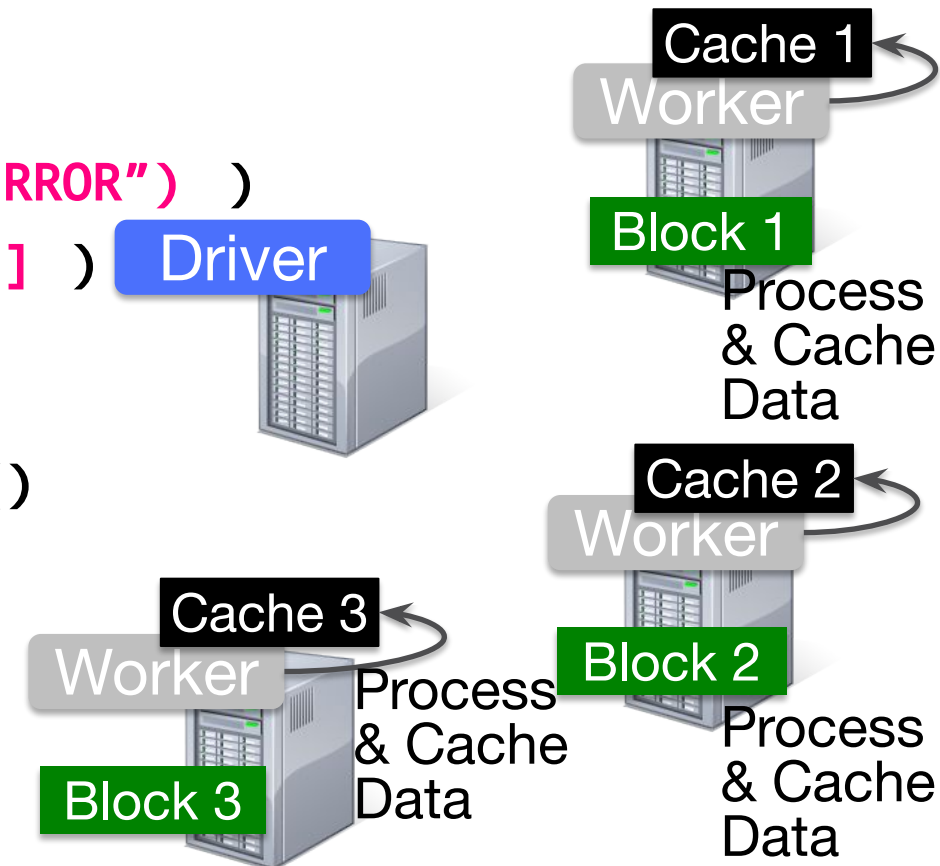


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )
errors = lines.filter(lambda s: s.startswith("ERROR") )
messages = errors.map(lambda s: s.split("\t")[2] )
messages.cache()

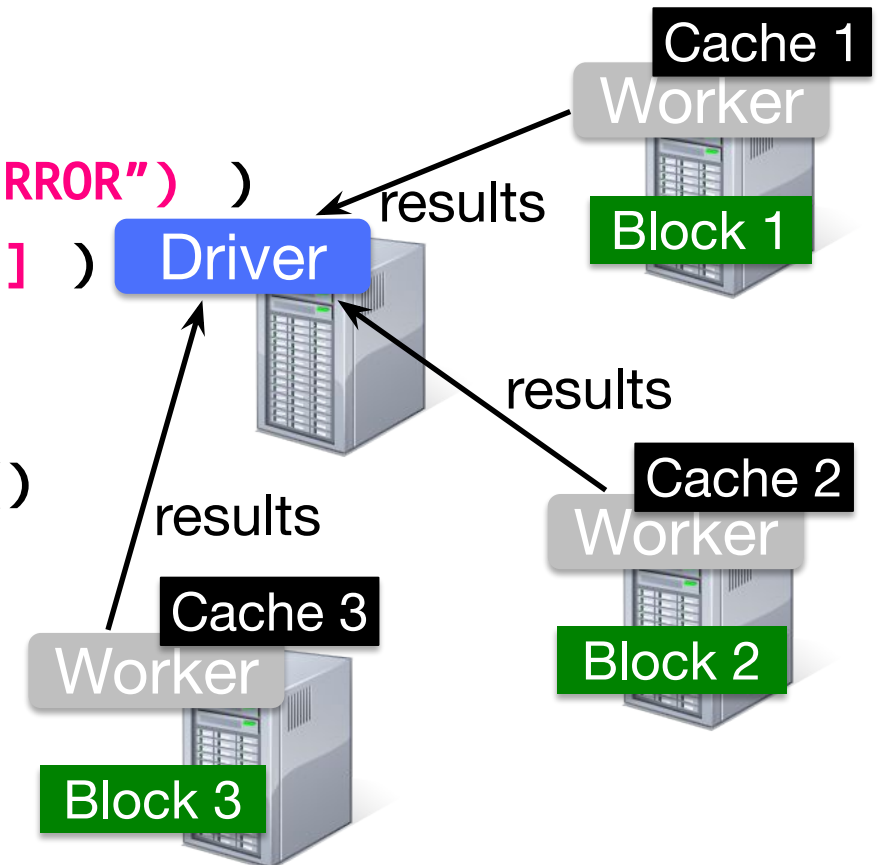
messages.filter(lambda s: "mysql" in s ).count()
```



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )  
errors = lines.filter(lambda s: s.startswith("ERROR") )  
messages = errors.map(lambda s: s.split("\t")[2] )  
messages.cache()  
  
messages.filter(lambda s: "mysql" in s ).count()
```

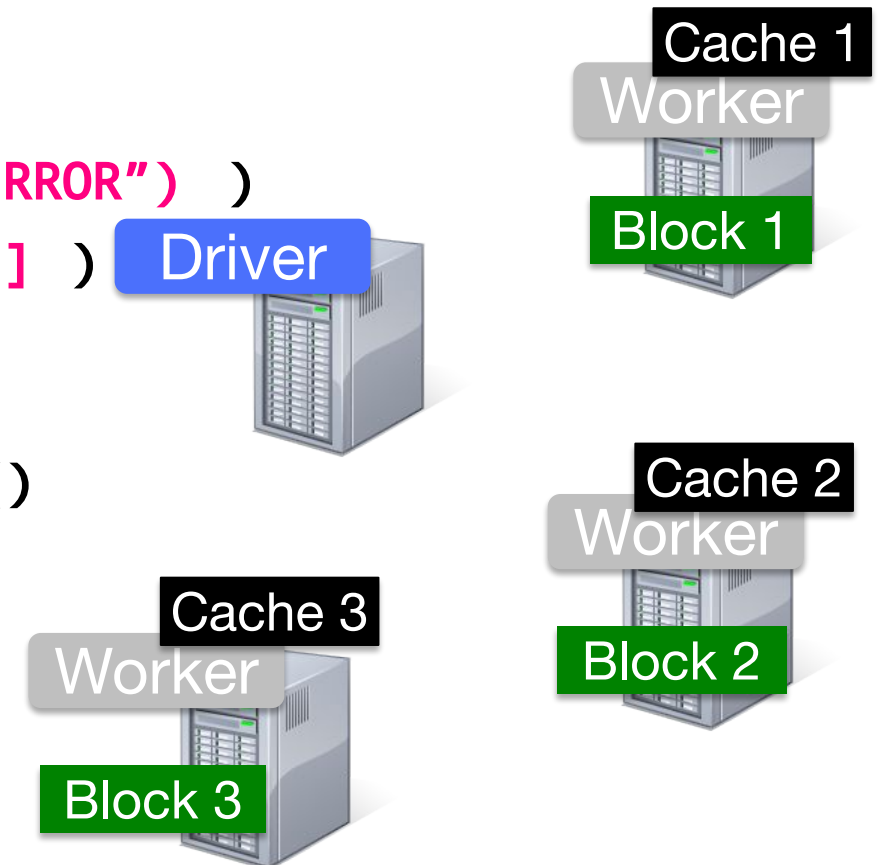


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )  
errors = lines.filter(lambda s: s.startswith("ERROR") )  
messages = errors.map(lambda s: s.split("\t")[2] )  
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s ).count()  
messages.filter(lambda s: "php" in s ).count()
```

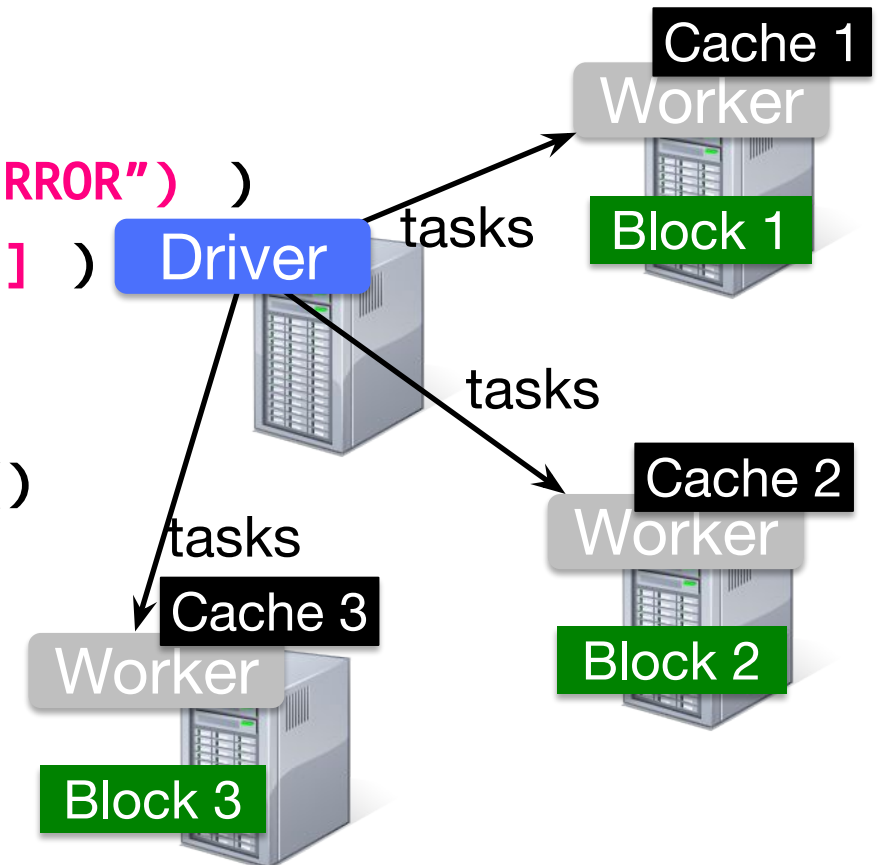


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )  
errors = lines.filter(lambda s: s.startswith("ERROR") )  
messages = errors.map(lambda s: s.split("\t")[2] )  
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s ).count()  
messages.filter(lambda s: "php" in s ).count()
```

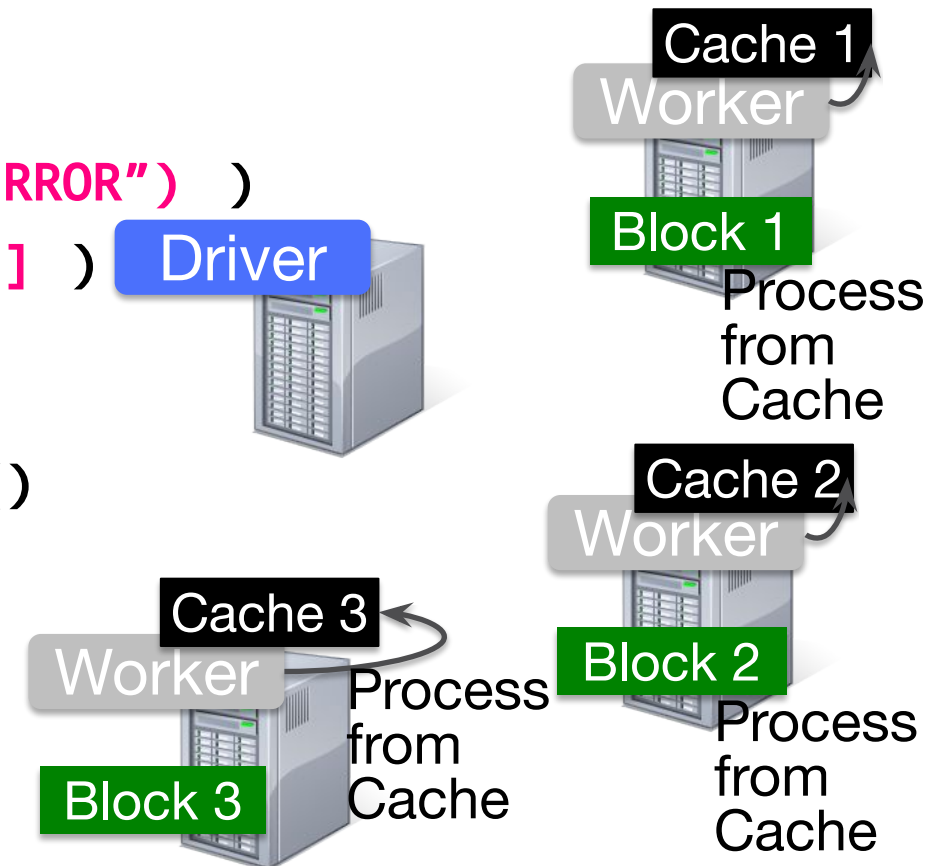


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )  
errors = lines.filter(lambda s: s.startswith("ERROR") )  
messages = errors.map(lambda s: s.split("\t")[2] )  
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s ).count()  
messages.filter(lambda s: "php" in s ).count()
```

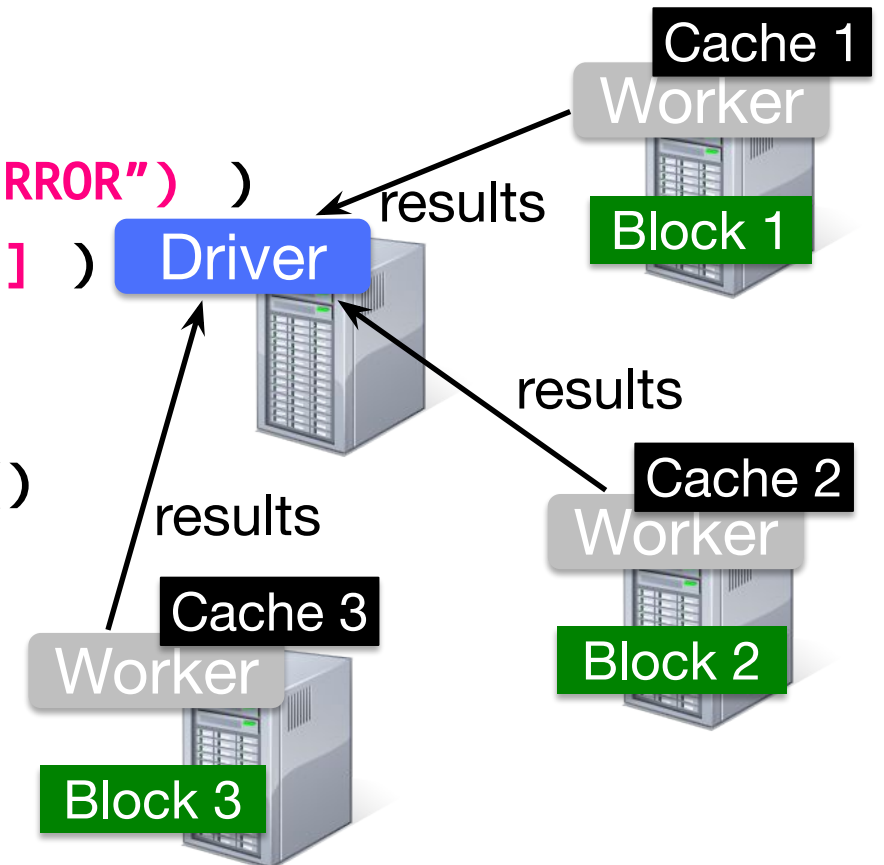


Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )  
errors = lines.filter(lambda s: s.startswith("ERROR") )  
messages = errors.map(lambda s: s.split("\t")[2] )  
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s ).count()  
messages.filter(lambda s: "php" in s ).count()
```



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile( "hdfs://..." )  
errors = lines.filter( lambda s: s.startswith("ERROR") )  
messages = errors.map( lambda s: s.split("\t")[2] )  
messages.cache()
```

```
messages.filter( lambda s: "mysql" in s ).count()  
messages.filter( lambda s: "php" in s ).count()
```

Cache your data ☐ **Faster Results**

Full-text search of Wikipedia

- 60GB on 20 EC2 machines
- 0.5 sec from mem vs 20s for on-disk

Driver

Cache 1
Worker

Block 1

Cache 2
Worker

Block 2

Cache 3
Worker

Block 3

Language Support

Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x =>
  x.contains("ERROR") ).count()
```

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>()
{
    Boolean call(String s) {
        return s.contains( "error" );
    }
}).count();
```

Standalone Programs

- Python, Scala, & Java

Interactive Shells

- Python & Scala

Performance

- Java & Scala are faster due to static typing
- ...but Python is often fine

Expressive API

- map

reduce

Expressive API

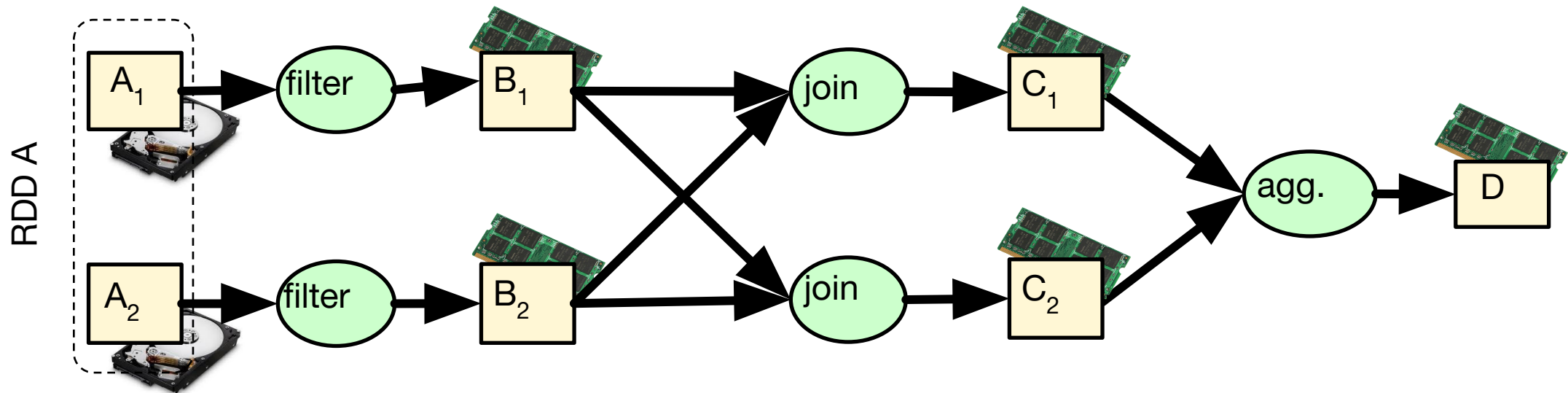
• map	reduce	sample
• filter	count	take
• groupBy	fold	first
• sort	reduceByKey	partitionBy
• union	groupByKey	mapWith
• join	cogroup	pipe
• leftOuterJoin	cross	save ...
• rightOuterJoin	zip	

Fault Recovery: Design Alternatives

- Replication:
 - Slow: need to write data over network
 - Memory inefficient
- Backup on persistent storage
 - Persistent storage still (much) slower than memory
 - Still need to go over network to protect against machine failures
- Spark choice:
 - Lineage: track sequence of operations to efficiently reconstruct lost RRD partitions

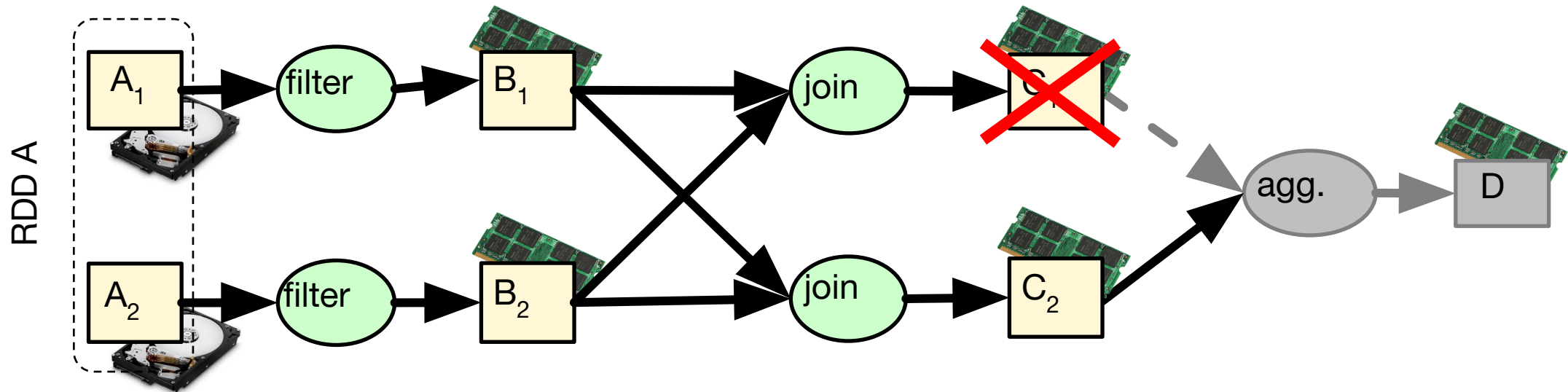
Fault Recovery Example

- Two-partition RDD A = {A₁, A₂} stored on disk
 - 1) filter and cache \square RDD B
 - 2) join \square RDD C
 - 3) aggregate \square RDD D



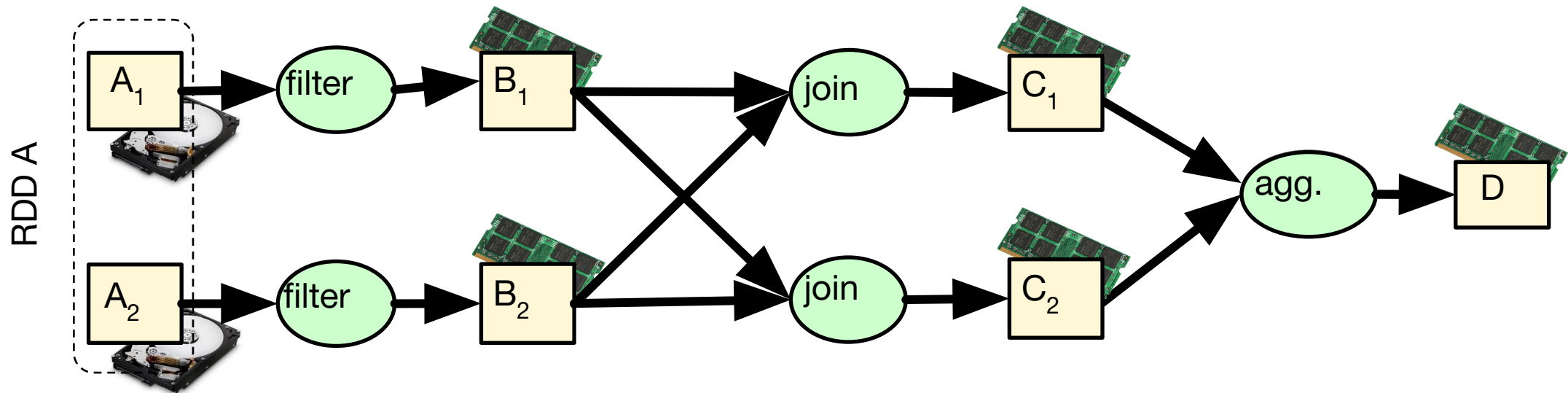
Fault Recovery Example

- C_1 lost due to node failure before reduce finishes

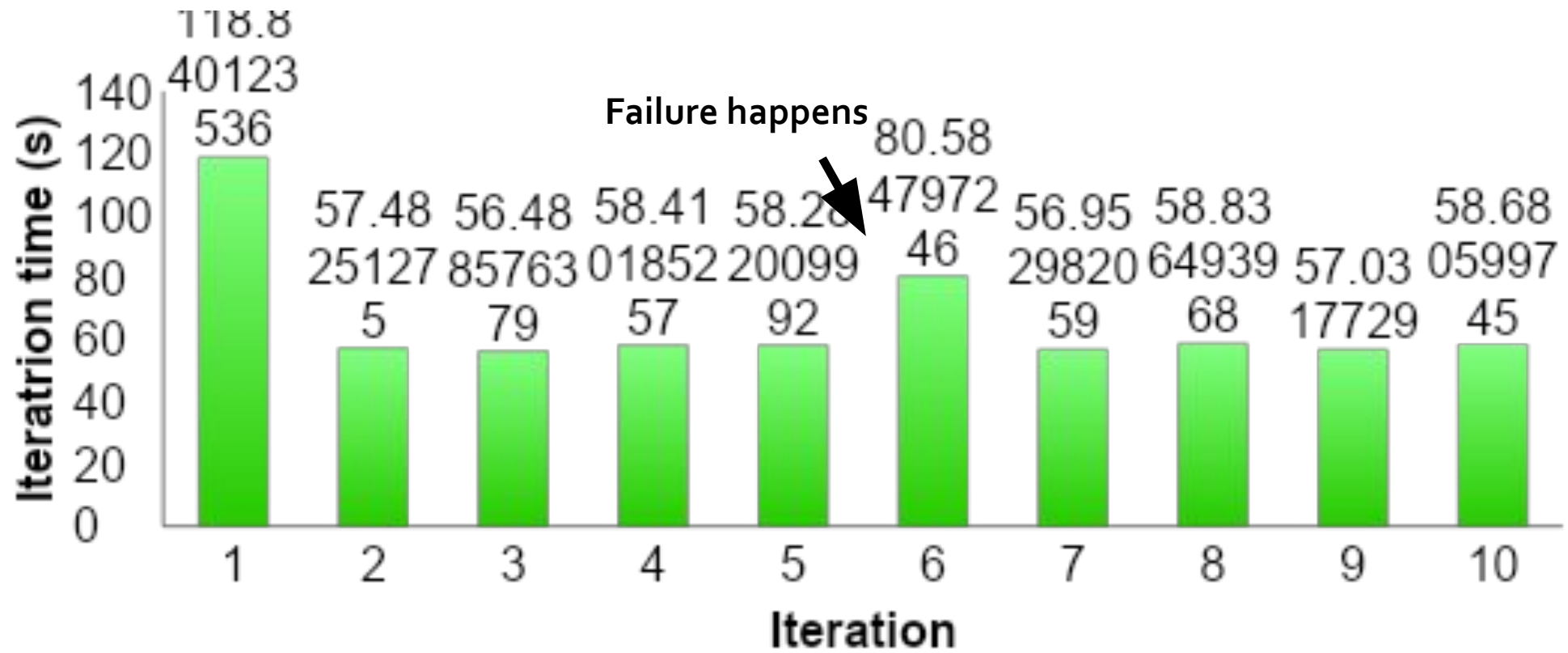


Fault Recovery Example

- C_1 lost due to node failure before reduce finishes
- Reconstruct C_1 , eventually, on different node



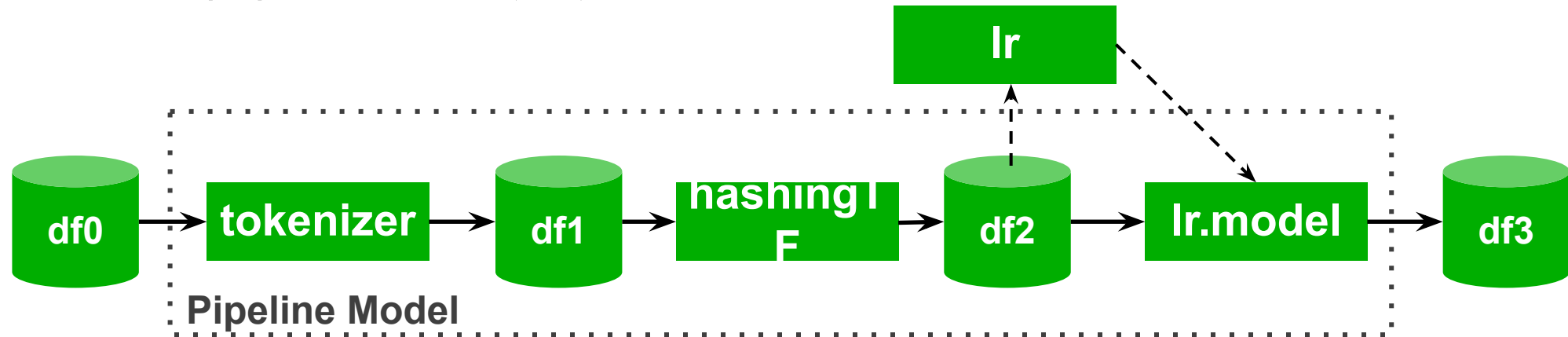
Fault Recovery Results



Machine Learning Pipelines

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol="words", outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

```
df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
```



Beyond Hadoop Users

**Spark early
adopters**



Users

**Understands
MapReduce
& functional
APIs**



**Data
Engineers
Data
Scientists
Statisticians
R users
PyData ...**

```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
    .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
    .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
    .collect()
```

```
data.groupBy("dept").avg("age")
```

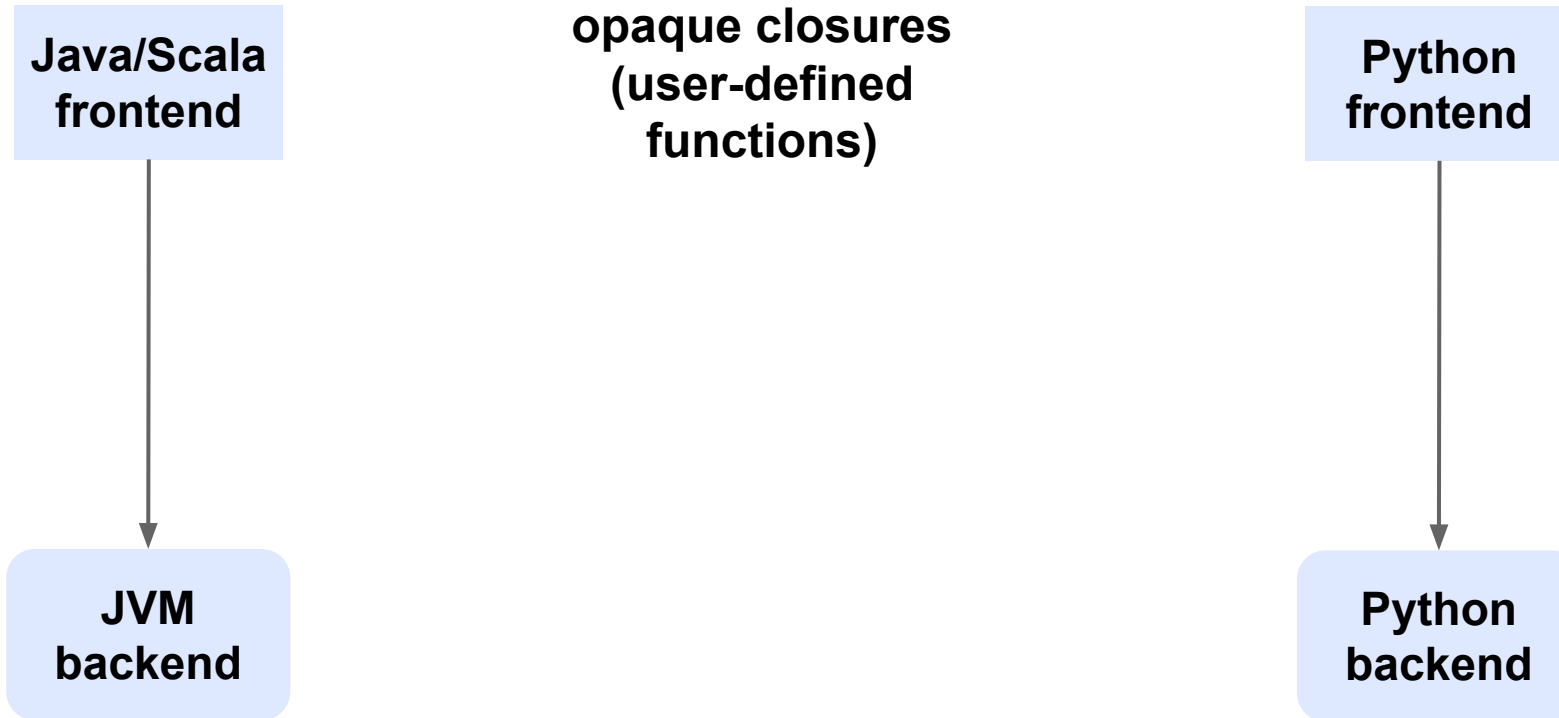
DataFrames in Spark

- Distributed collection of data grouped into named columns (i.e. RDD with schema)
- Domain-specific functions designed for common tasks
 - Metadata
 - Sampling
 - Project, filter, aggregation, join, ...
 - UDFs
- Available in Python, Scala, Java, and R

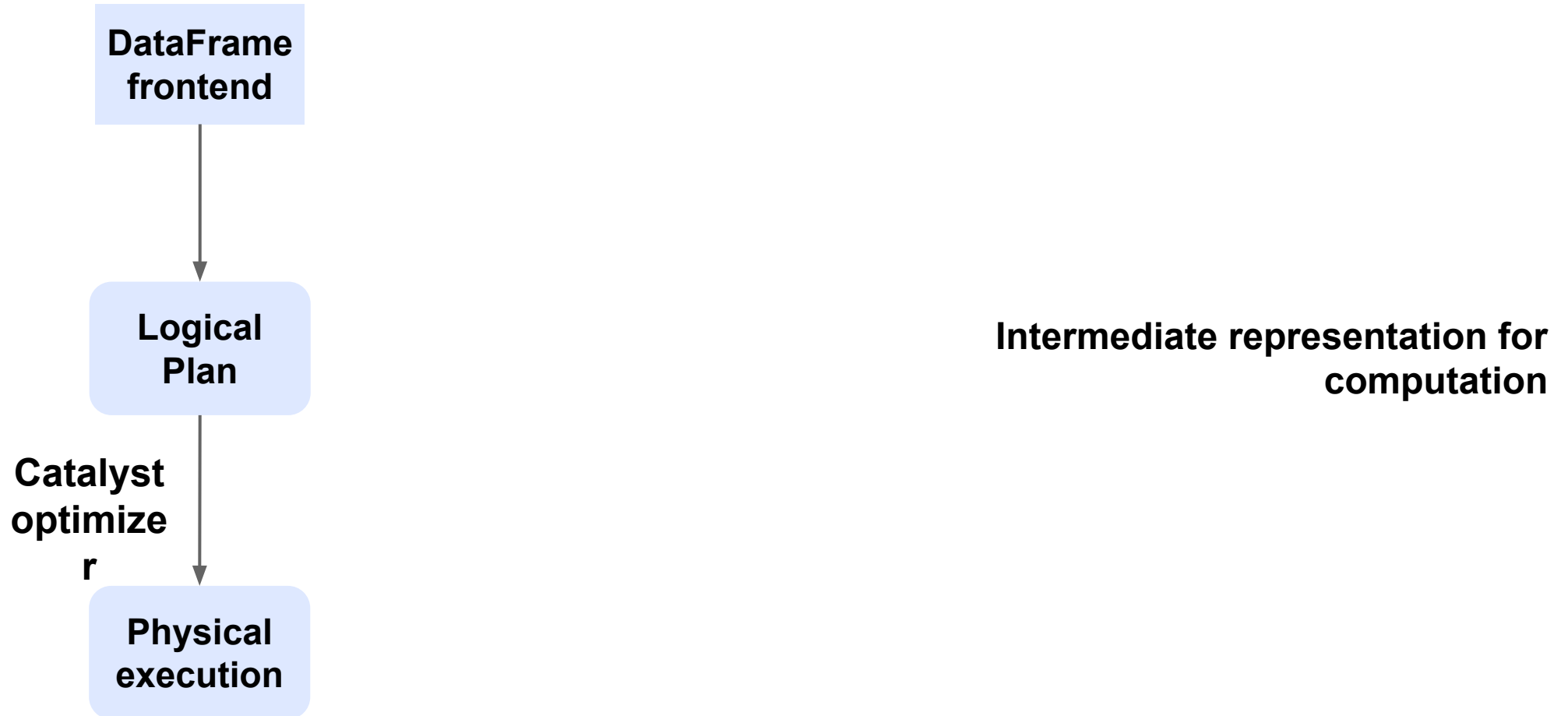
Similar APIs as single-node tools (e.g., Pandas), i.e. easy to learn

```
• > head(filter(df, df$waiting < 50)) # an example in R
• ## eruptions waiting
• ##1      1.750      47
• ##2      1.750      47
• ##3      1.867      48
```

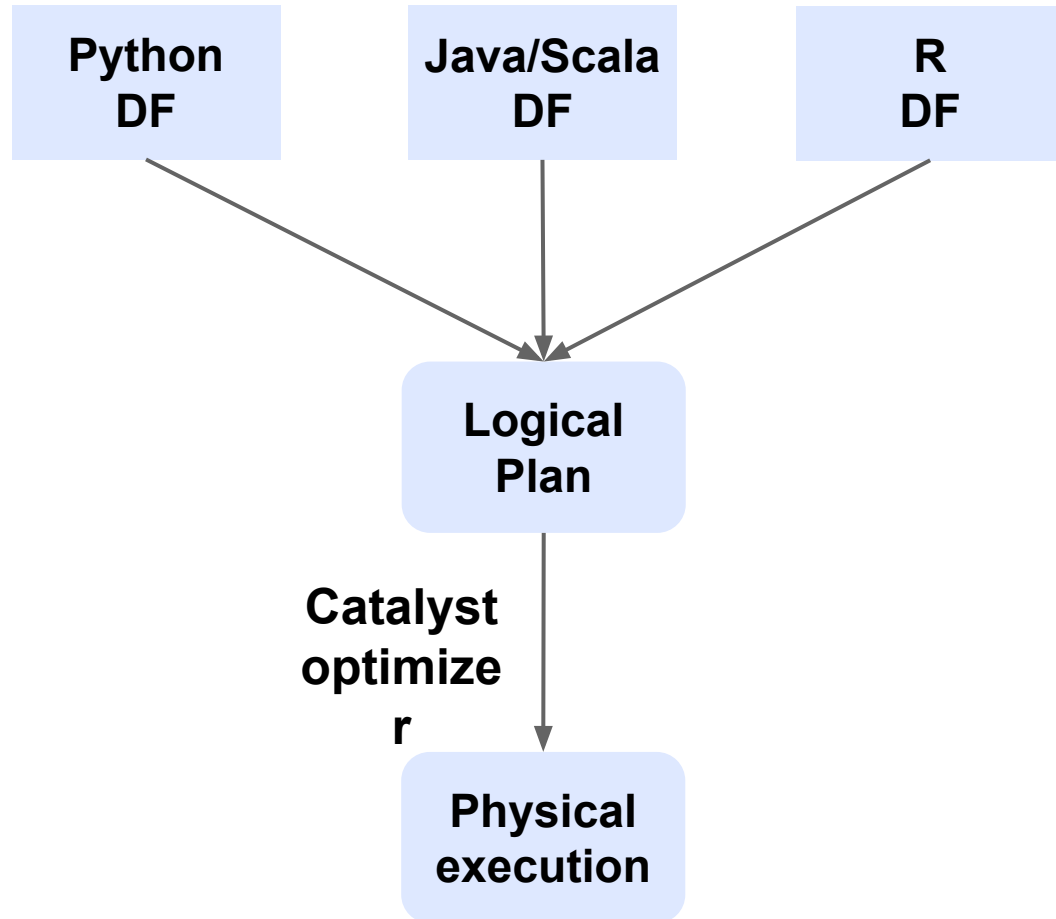
Spark RDD Execution



Spark DataFrame Execution



Spark DataFrame Execution



Simple wrappers to create logical plan

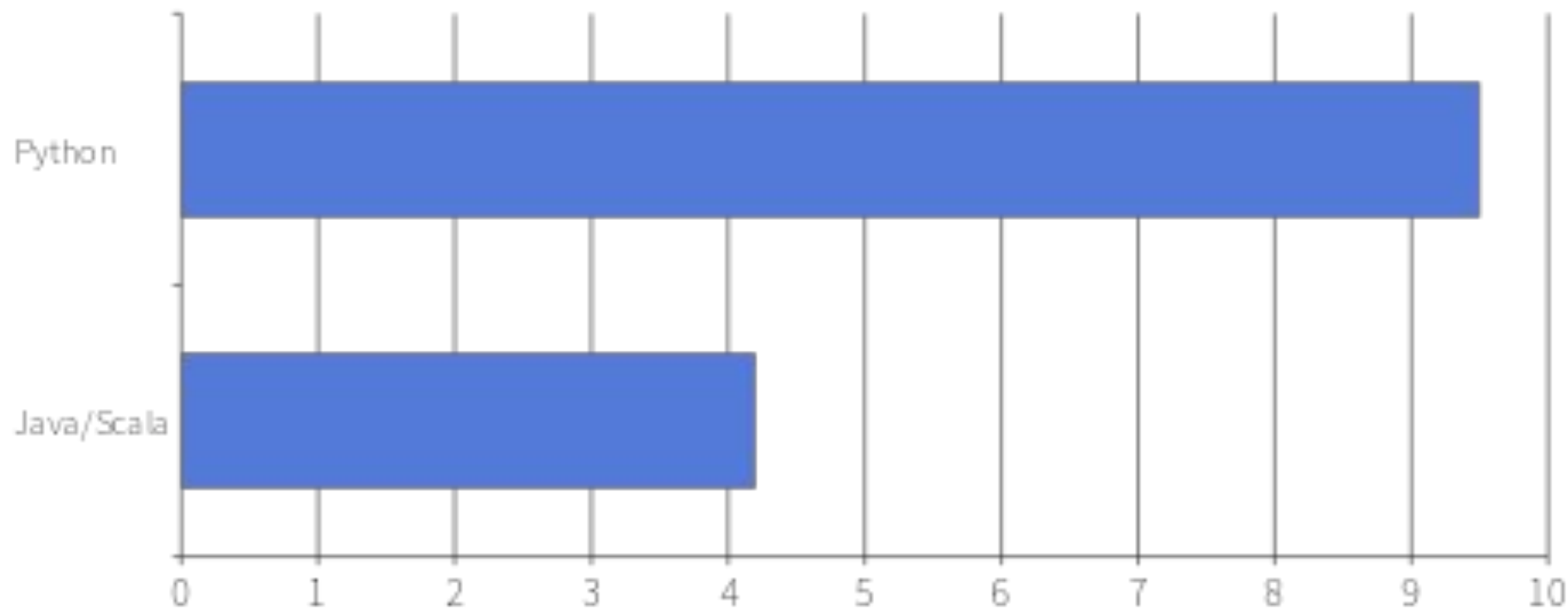
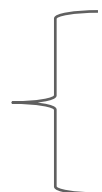
Intermediate representation for computation

Benefit of Logical Plan: Simpler Frontend

- Python : ~2000 line of code (built over a weekend)
- R : ~1000 line of code
- i.e. much easier to add new language bindings (Julia, Clojure, ...)

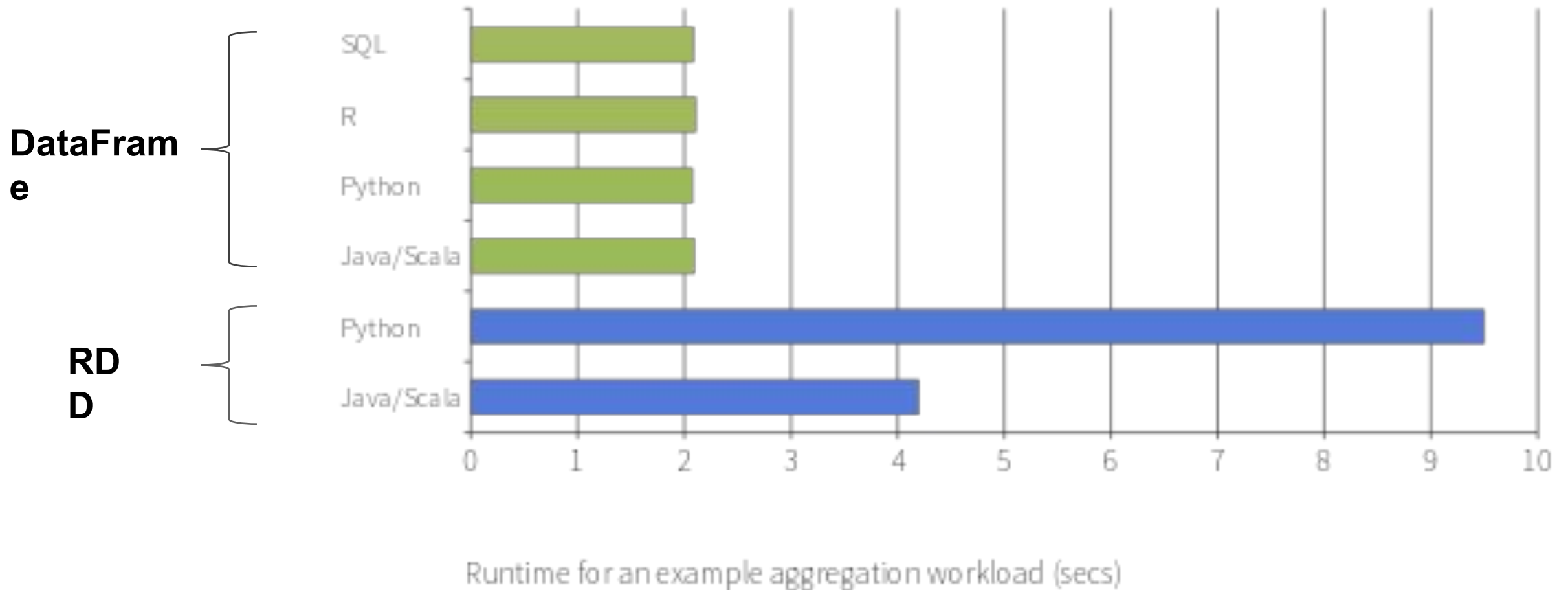
Performance

**RD
D**

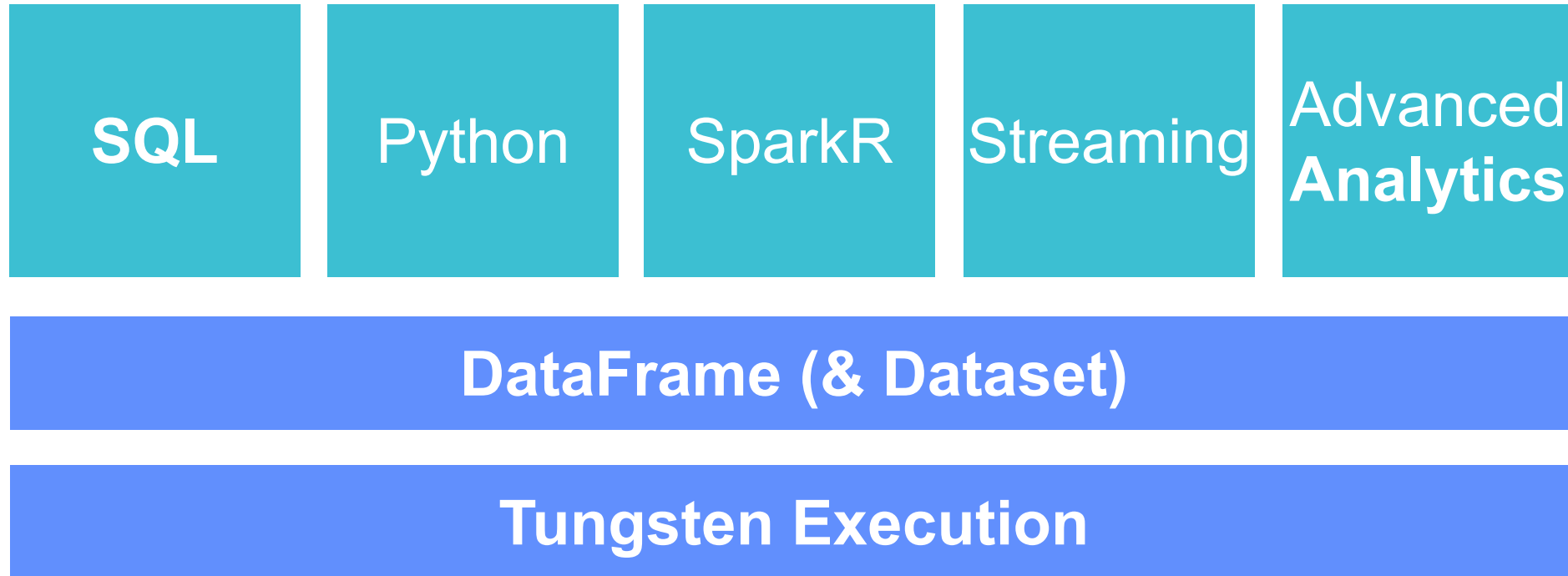


Runtime for an example aggregation workload

Benefit of Logical Plan: Performance Parity Across Languages



Refactoring Spark Core



Summary

- General engine with libraries for many data analysis tasks
- Access to diverse data sources
- Simple, unified API
- Easy of use (DataFrames)

