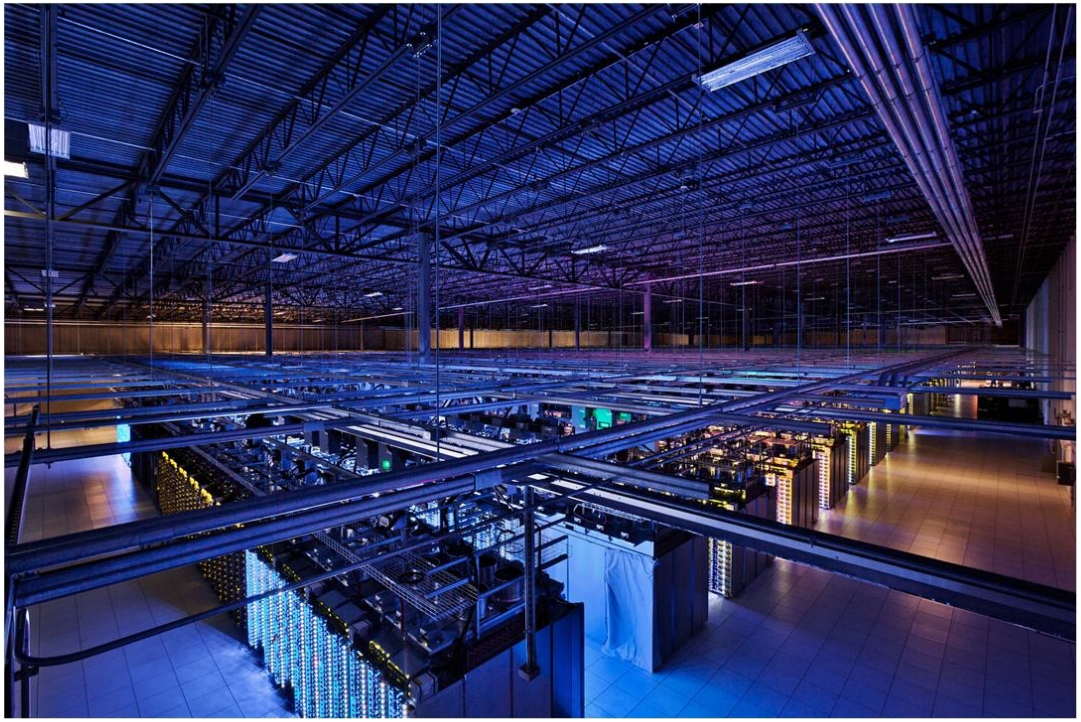
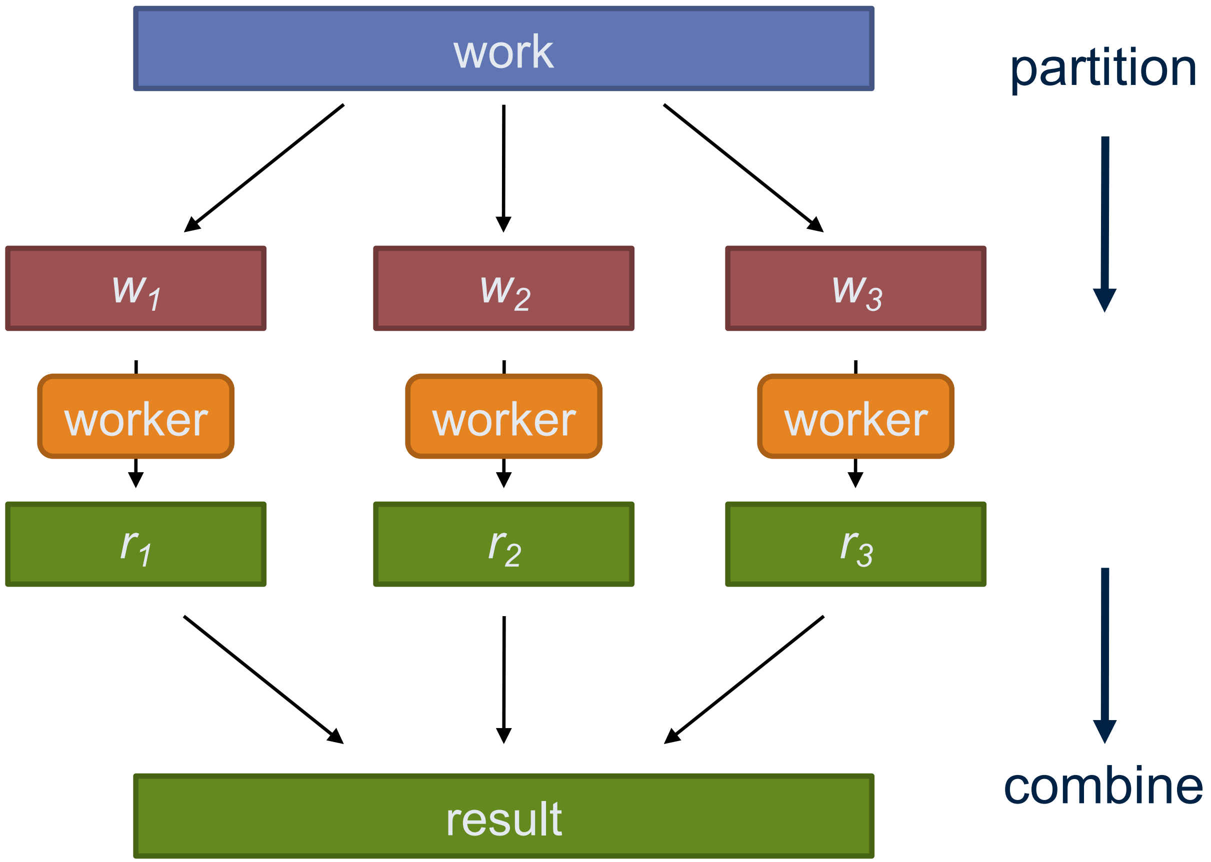
Large Scale Data Engineering

Big Data Frameworks: Hadoop & Spark



event.cwi.nl/lsde

Key premise: divide and conquer



Parallelisation challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we know all the workers have finished?
- What if workers die?
- What if data gets lost while transmitted over the network?

What's the common theme of all of these problems?

Common theme?

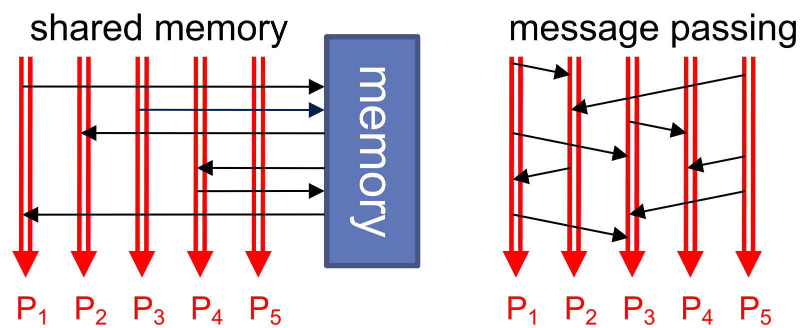
- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

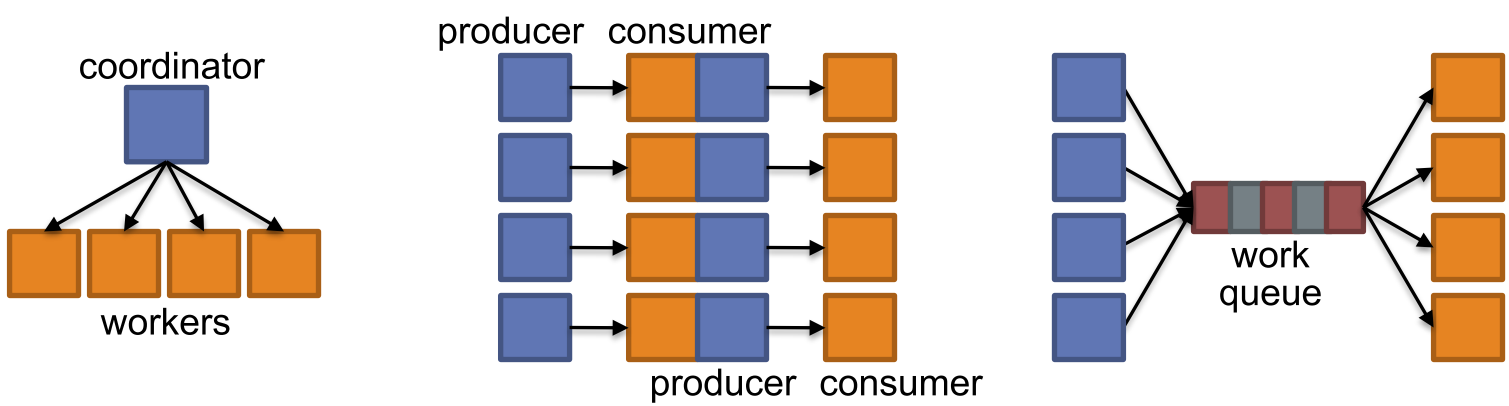
Managing multiple workers

- Difficult because
  - We don't know the order in which workers run
  - We don't know when workers interrupt each other
  - We don't know when workers need to communicate partial results
  - We don't know the order in which workers access shared data
- Thus, we need:
  - Semaphores (lock, unlock)
  - Conditional variables (wait, notify, broadcast)
  - Barriers
- Still, lots of problems:
  - Deadlock, livelock, race conditions...
  - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

Current tools

- Programming models
  - Shared memory (pthreads)
  - Message passing (MPI)
- Design patterns
  - Master-slaves
  - Producer-consumer flows
  - Shared work queues





Parallel programming: human bottleneck

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters and across datacenters
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

- The MapReduce Framework alleviates this
  - making this easy is what gave Google the advantage

What's the point?

- It's all about the right level of abstraction
  - Moving beyond the von Neumann architecture
  - We need better programming models
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the what from how
  - Developer specifies the computation that needs to be performed
  - Execution framework (aka runtime) handles actual execution

The data center *is* the computer!

Centrum Wiskunde & Informatica

MAPREDUCE AND HDFS

Typical Big Data Problem

Map

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Reduce

Key idea: provide a functional abstraction for these two operations

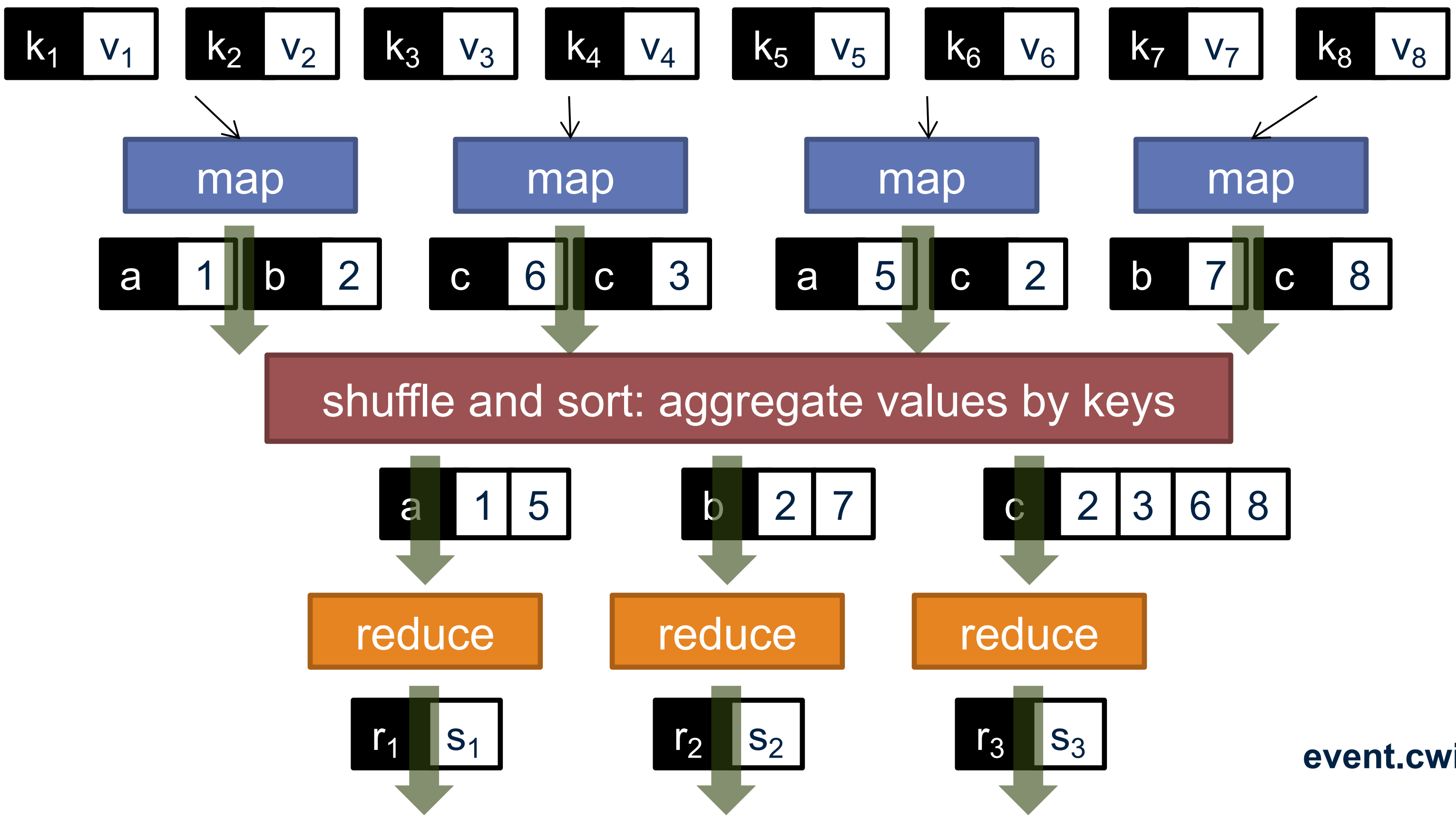
MapReduce

- Programmers specify two functions:

  map $(k_1, v_1) \rightarrow [\langle k_2, v_2 \rangle]$

  reduce $(k_2, [v_2]) \rightarrow [\langle k_3, v_3 \rangle]$

  – All values with the same key are sent to the same reducer



MapReduce runtime

- Orchestration of the distributed computation
- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles data distribution
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system (more information later)

MapReduce

- Programmers specify two functions:

  map (k, v) → <k’, v’>*

  reduce (k’, v’*) → <k’’, v’’>*

  – All values with the same key are reduced together
- The execution framework handles everything else
- This is the minimal set of information to provide
- Usually, programmers also specify:

  partition (k’, number of partitions) → partition for k’

  – Often a simple hash of the key, e.g., hash(k’) mod n

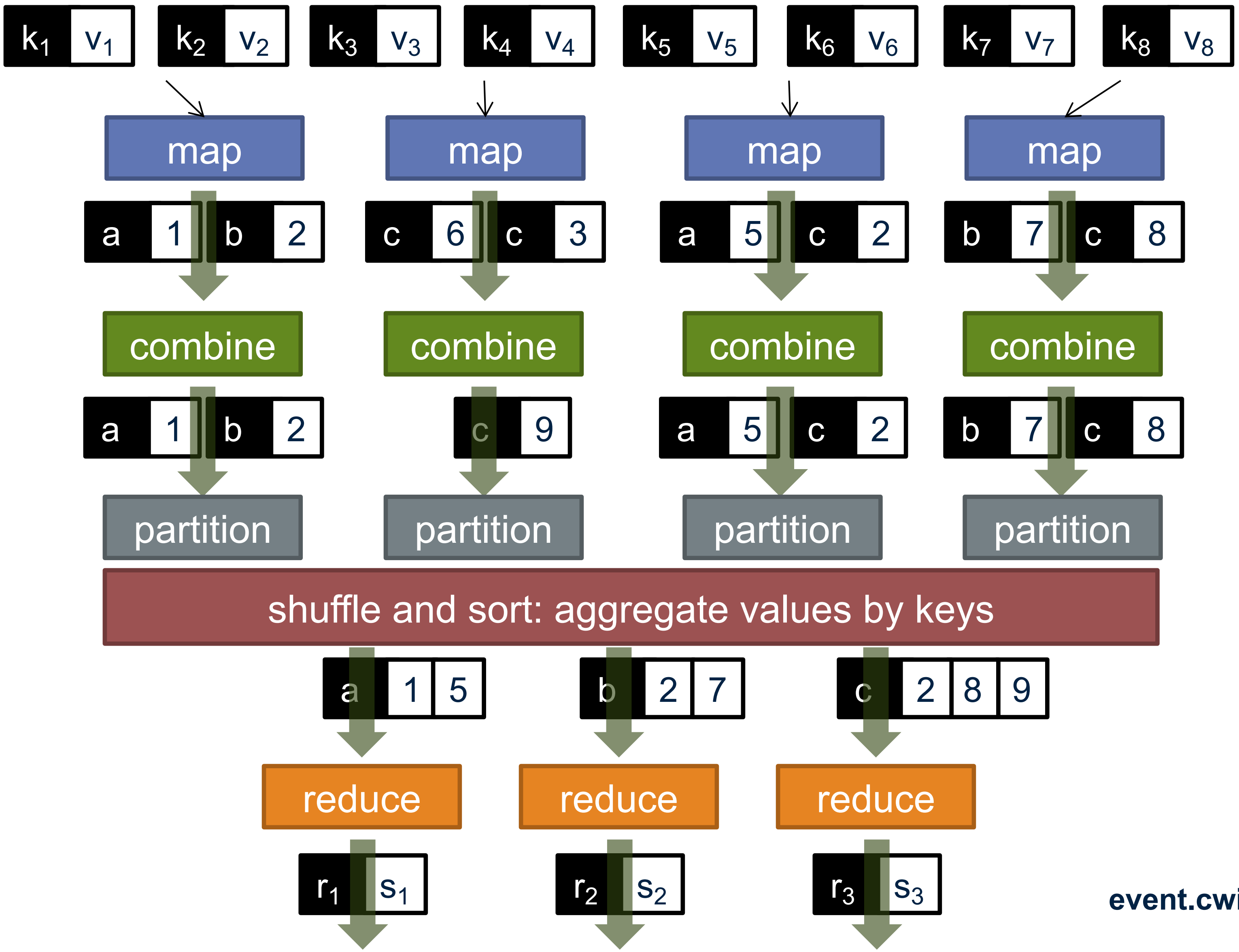
  – Divides up key space for parallel reduce operations

  combine (k’, v’*) → <k’, v’’*>*

  – Mini-reducers that run in memory after the map phase

  – Used as an optimization to reduce network traffic

Putting it all together



event.cwi.nl/lsde

“Hello World”: Word Count

```
Map(String docid, String text):
     for each word w in text:
          Emit(w, 1);

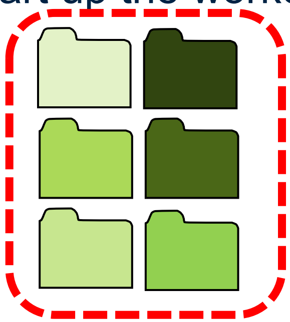
Reduce(String term, Iterator<Int> values):
     int sum = 0;
     for each v in values:
          sum += v;
     Emit(term, sum);
```

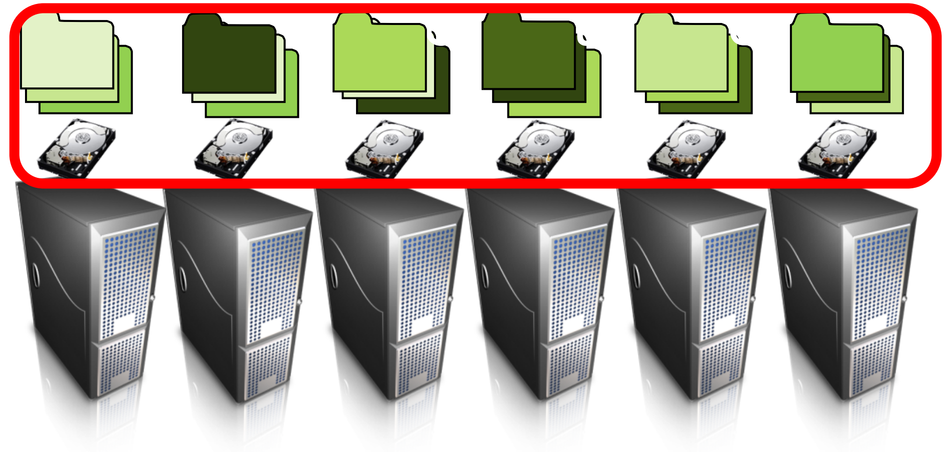
MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Facebook, Twitter, LinkedIn, Netflix, …
  - Popular on-premise big data processing platform, but..
    - Has been losing support to cloud-based platforms

Distributed file system

- Do not move data to workers, but move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local





- Why?
  - Avoid network traffic if possible
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

Note: all data is replicated for fault-tolerance (HDFS default:3x)

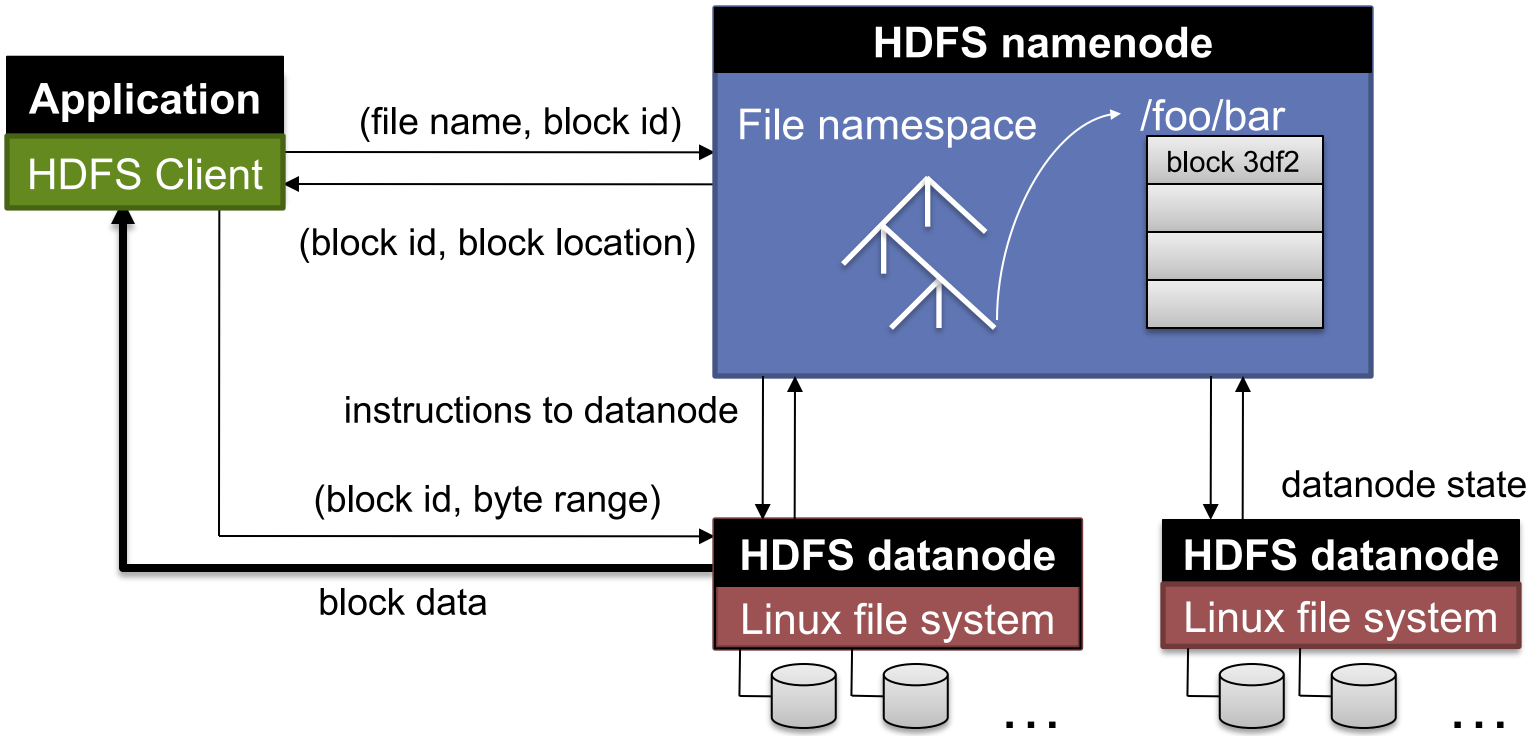
HDFS: Assumptions

- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

HDFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads

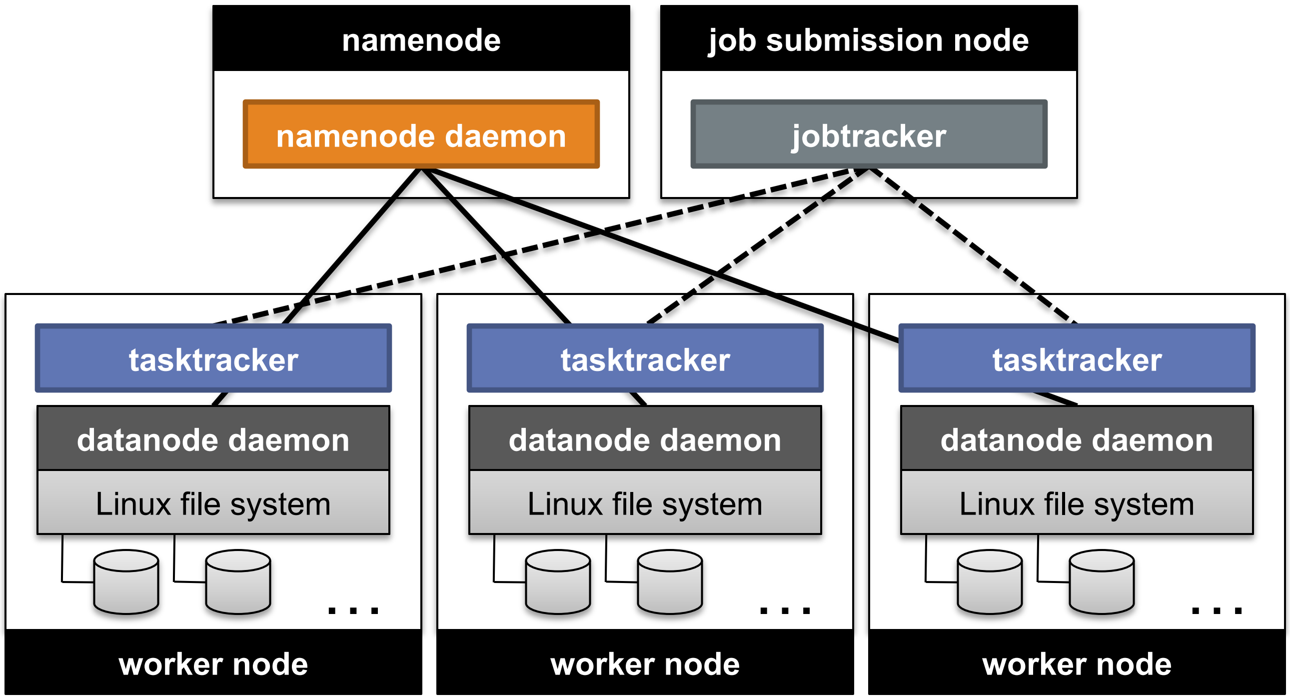
HDFS architecture



Namenode responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

Putting everything together



Basic cluster components

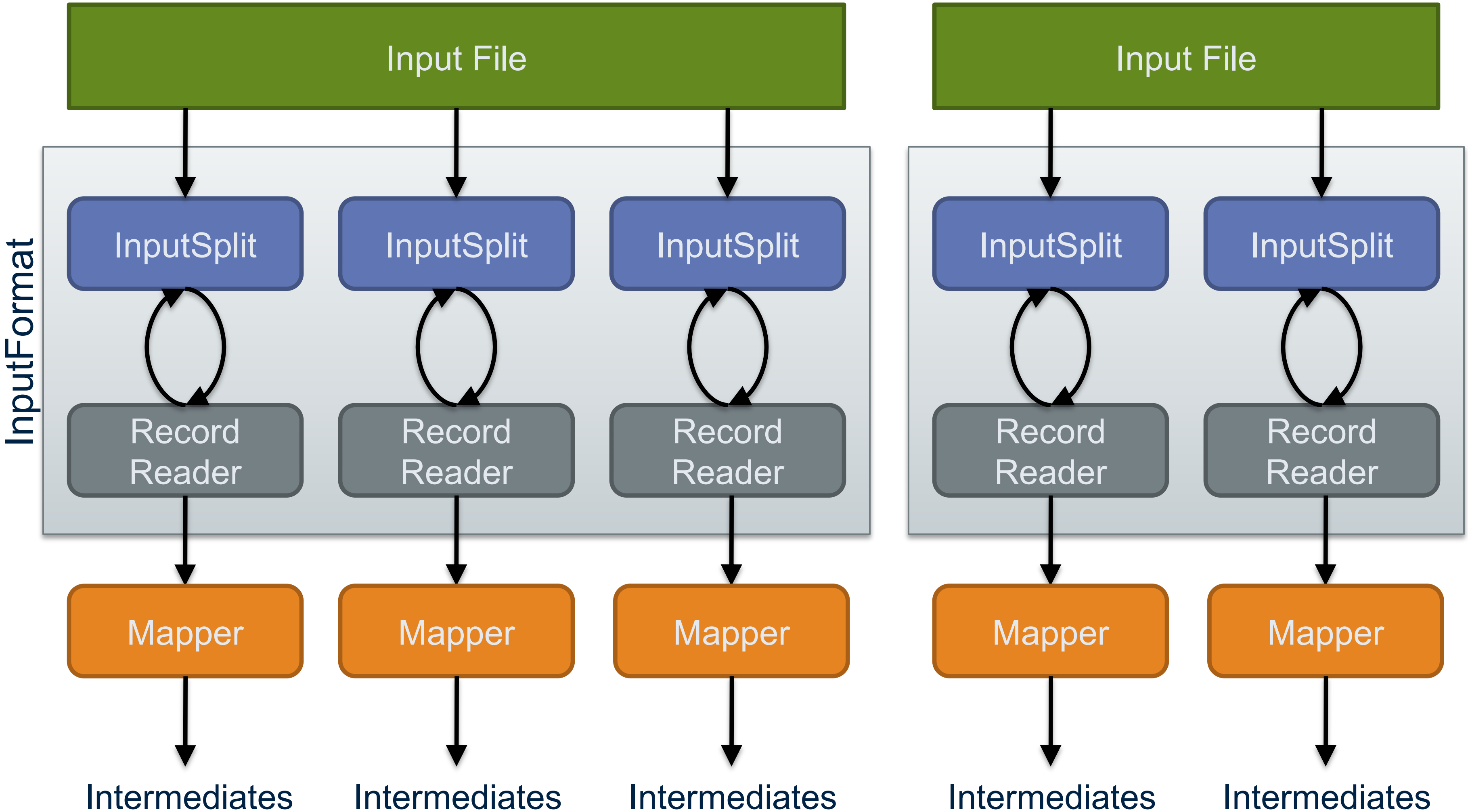
- One of each:
  - Namenode (NN): master node for HDFS
  - Jobtracker (JT): master node for job submission
- Set of each per worker machine:
  - Tasktracker (TT): contains multiple task slots
  - Datanode (DN): serves HDFS data blocks

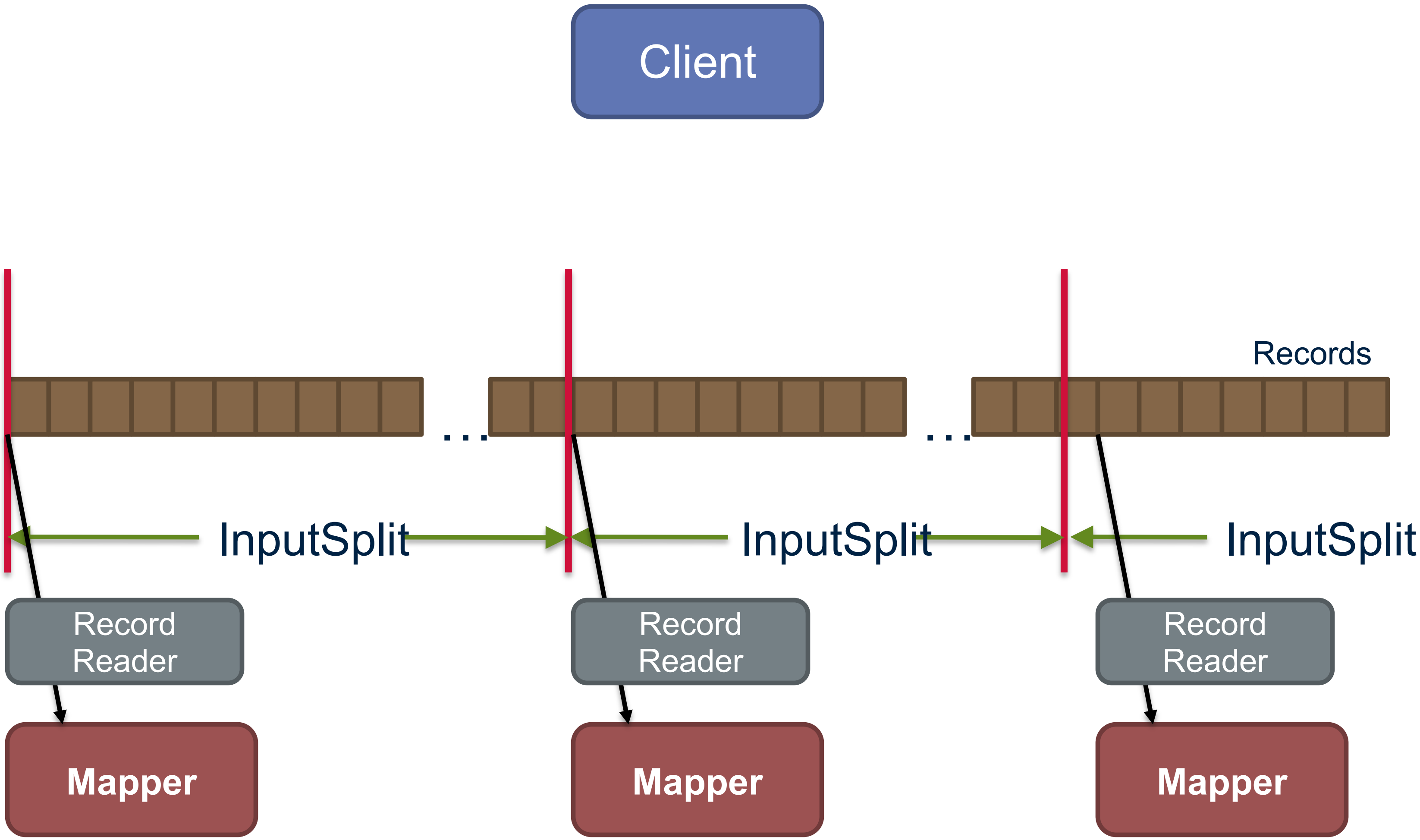
Anatomy of a job

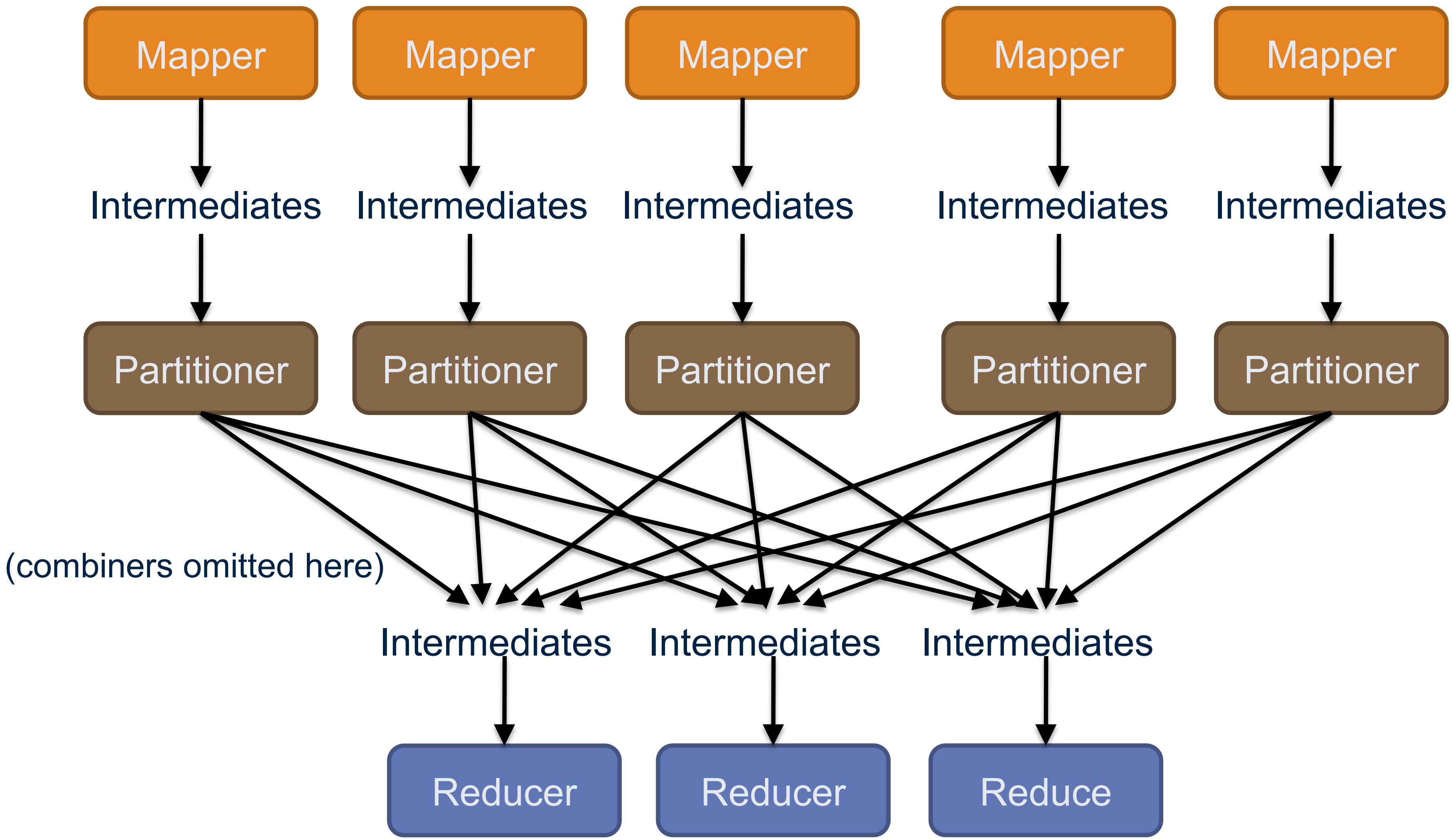
- MapReduce program in Hadoop = Hadoop job
  - Jobs are divided into map and reduce tasks
  - An instance of running a task is called a task attempt (occupies a slot)
  - Multiple jobs can be composed into a workflow
- Job submission:
  - Client (i.e., driver program) creates a job, configures it, and submits it to jobtracker
  - That's it! The Hadoop cluster takes over

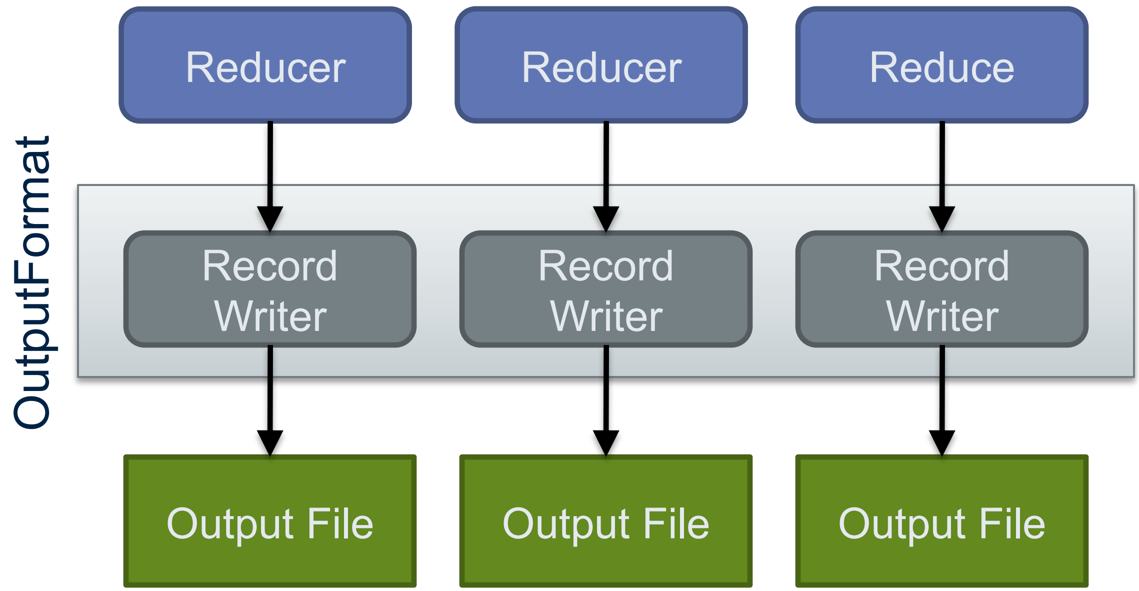
Anatomy of a job

- Behind the scenes:
  - Input splits are computed (on client end)
  - Job data (jar, configuration XML) are sent to JobTracker
  - JobTracker puts job data in shared location, enqueues tasks
  - TaskTrackers poll for tasks
  - Off to the races


Input File
Input File
InputFormat
InputSplit
InputSplit
InputSplit
InputSplit
InputSplit
Record Reader
Record Reader
Record Reader
Record Reader
Record Reader
Mapper
Mapper
Mapper
Mapper
Mapper
Intermediates
Intermediates
Intermediates
Intermediates
Intermediates


Client
Records
InputSplit
InputSplit
InputSplit
Record Reader
Record Reader
Record Reader
Mapper
Mapper
Mapper

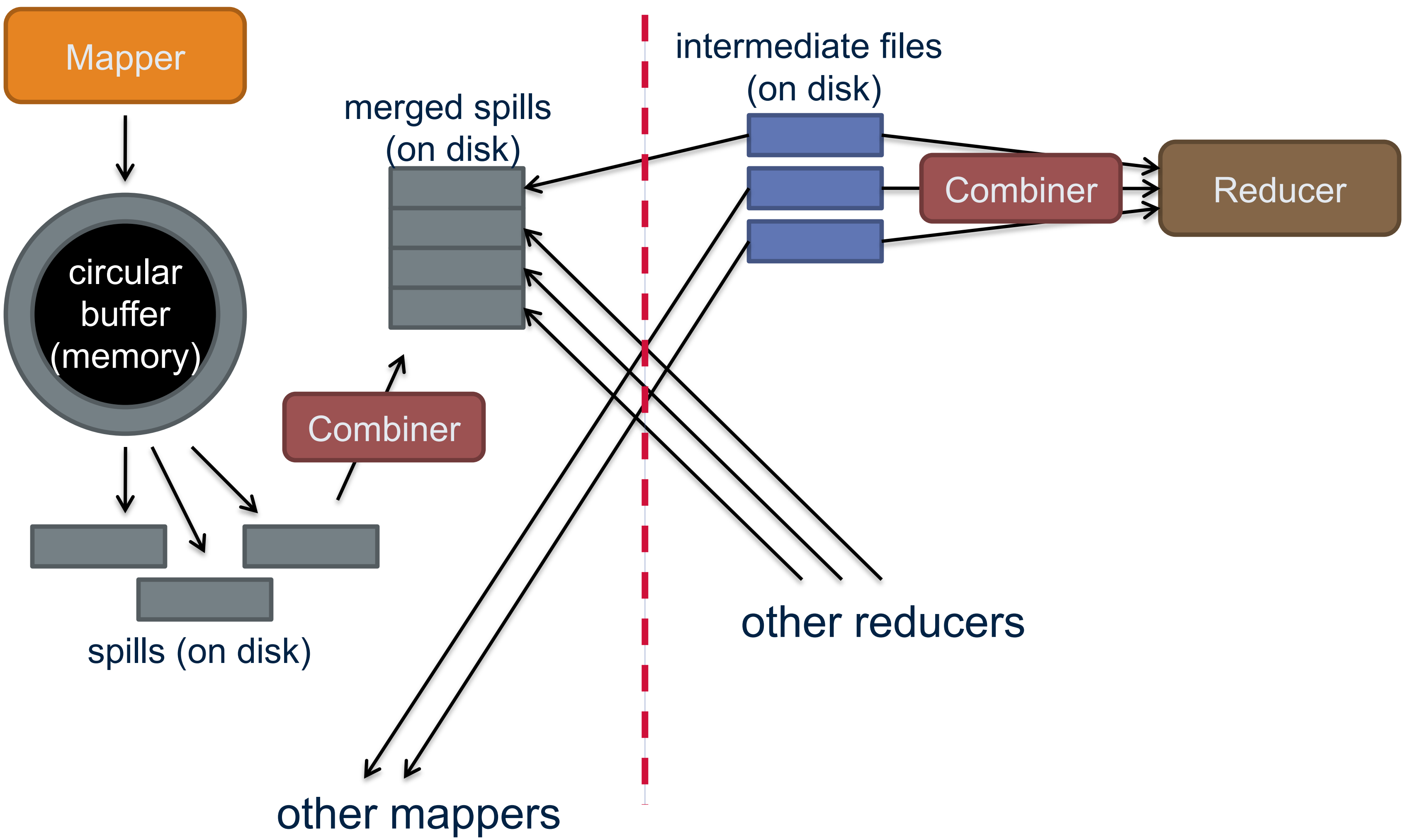

Mapper
Mapper
Mapper
Mapper
Mapper
Intermediates
Intermediates
Intermediates
Intermediates
Intermediates
Partitioner
Partitioner
Partitioner
Partitioner
Partitioner
(combiners omitted here)
Intermediates
Intermediates
Intermediates
Reducer
Reducer
Reduce


Reducer
Reducer
Reduce
OutputFormat
Record Writer
Record Writer
Record Writer
Output File
Output File
Output File

Shuffle and sort in Hadoop

- Probably the most complex aspect of MapReduce
- Map side
  - Map outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are spilled to disk
  - Spills merged in a single, partitioned file (sorted within each partition): combiner runs during the merges
- Reduce side
  - First, map outputs are copied over to reducer machine
  - Sort is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs during the merges
  - Final merge pass goes directly into reducer

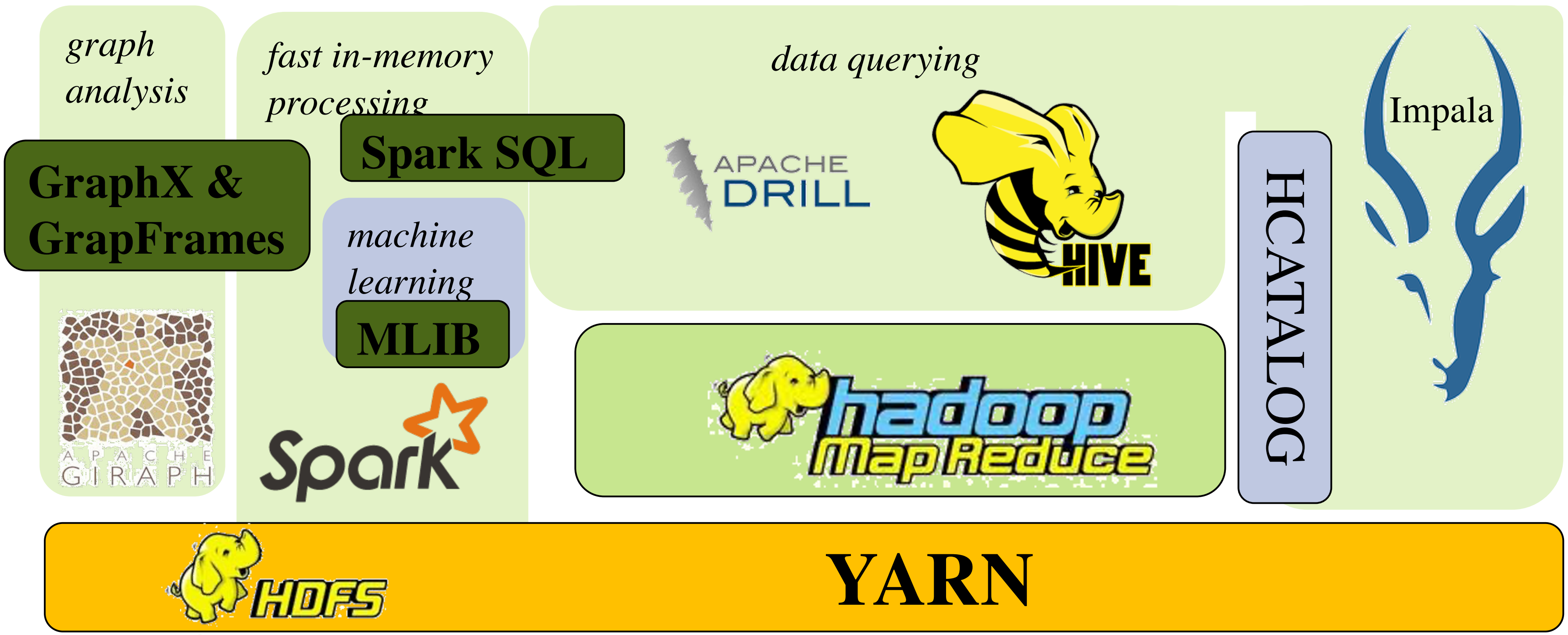
Shuffle and sort



YARN: Hadoop version 2.0

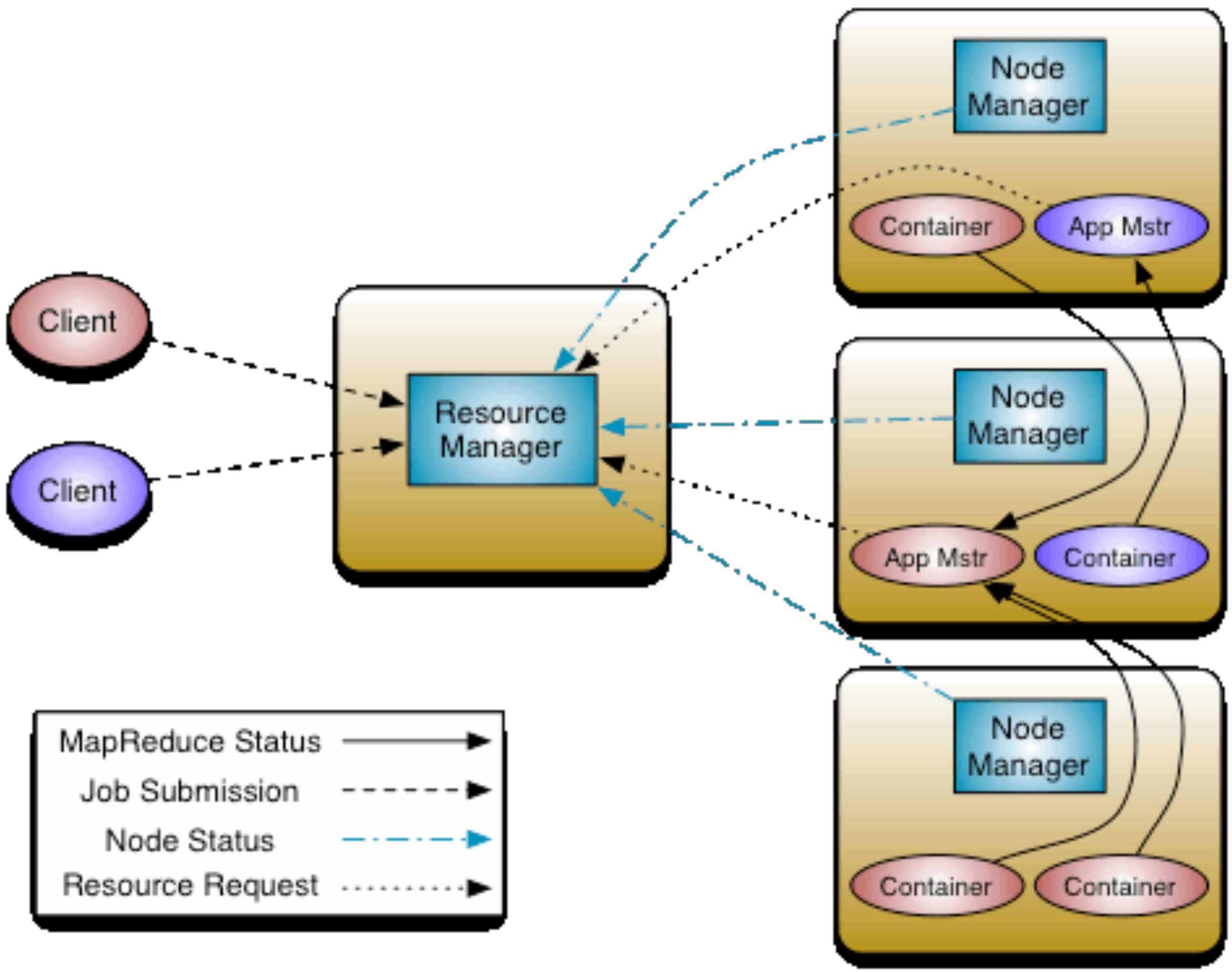
- Hadoop limitations:
  - Can only run MapReduce
  - What if we want to run other distributed frameworks?
- YARN = Yet-Another-Resource-Negotiator
  - Provides API to develop any generic distribution application
  - Handles scheduling and resource request
  - MapReduce (MR2) is one such application in YARN

The Hadoop Ecosystem

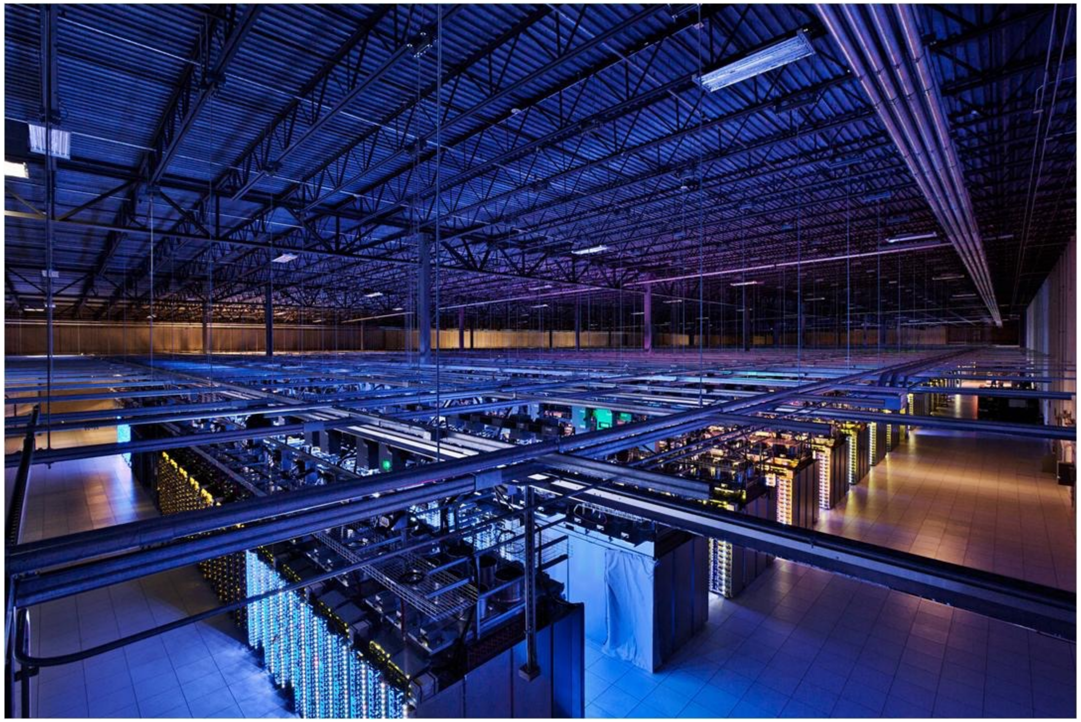


- "Data Lakes"
  - Large collections of raw data, stored cheaply in HDFS (or in the cloud)
  - A zoo of tools and pipelines to clean, transform & analyze this data
    - Drill, Hive and Impala are SQL systems that work in Hadoop
    - Hcatalog is the Hadoop meta-data repository (which tables exist?)

YARN: architecture



Spark



credits:
Matei Zaharia & Xiangrui Meng

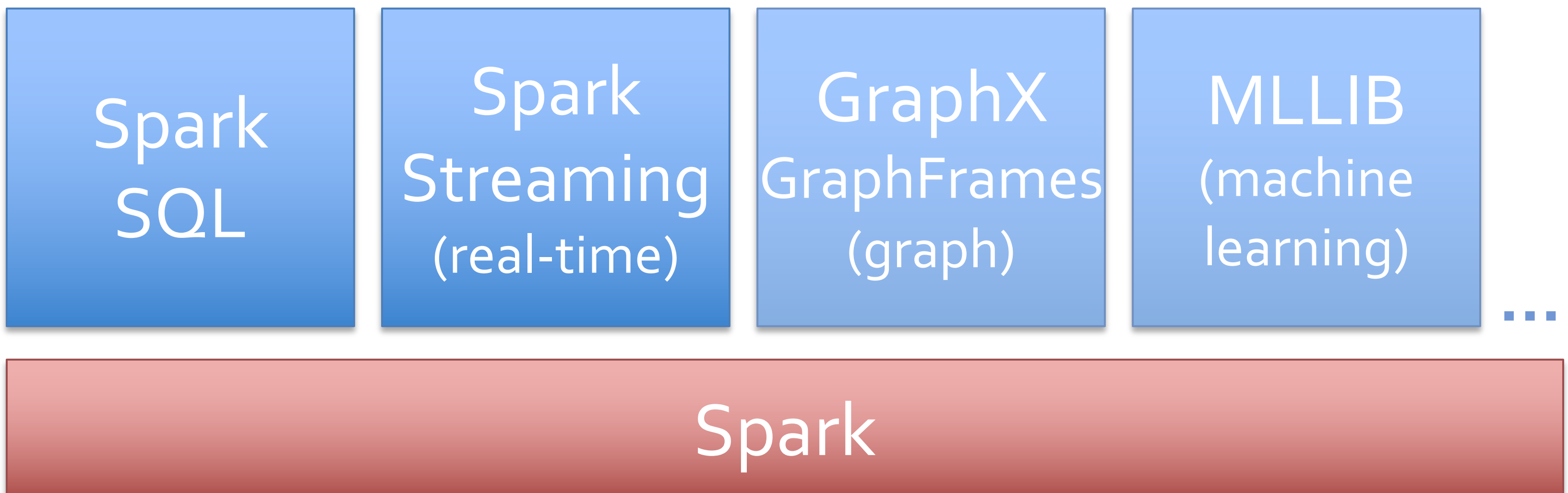
What is Spark?

- Fast and expressive cluster computing system interoperable with Apache Hadoop
- Improves efficiency through: → Up to 100 × faster (2-10 × on disk)
  - In-memory computing primitives
  - General computation graphs
- Improves usability through: → Often 5 × less code
  - Rich APIs in Scala, Java, Python
  - Interactive shell

credits:
Matei Zaharia & Xiangrui Meng

The Spark Stack

- Spark is the basis of a wide set of projects in the Berkeley Data Analytics Stack (BDAS)



More details: amplab.berkeley.edu

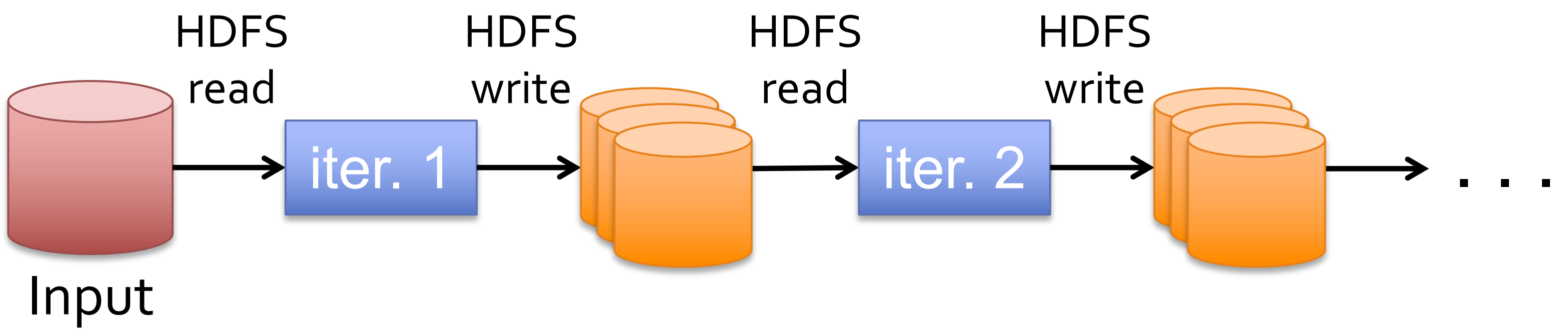
credits:
Matei Zaharia & Xiangrui Meng

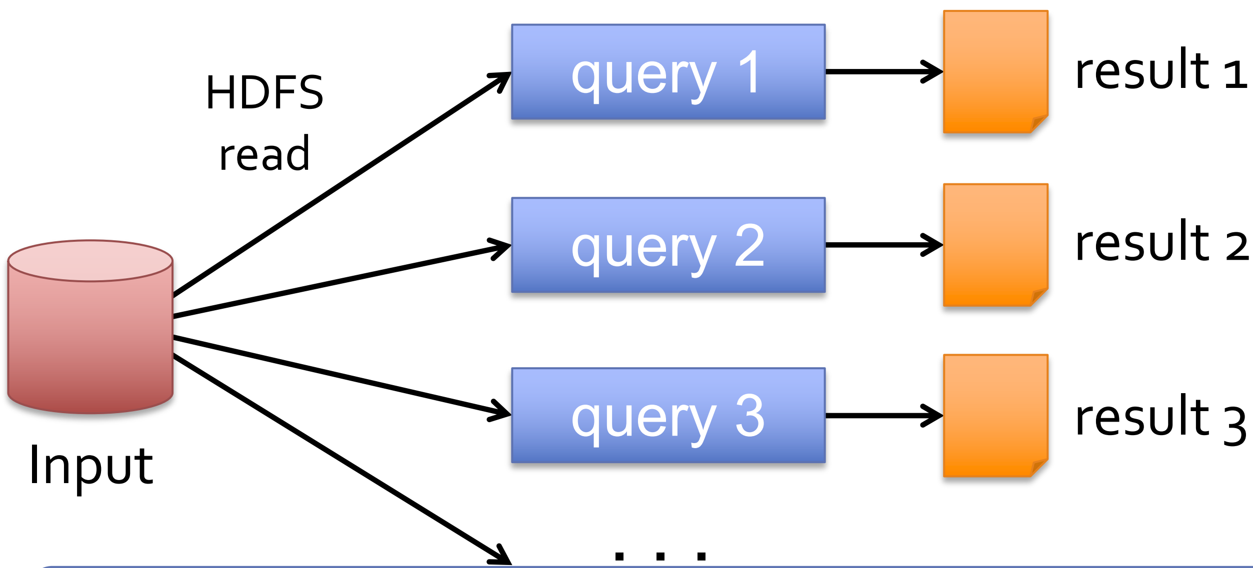
Why a New Programming Model?

- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
  - More **complex**, multi-pass analytics (e.g. ML, graph)
  - More **interactive** ad-hoc queries
  - More **real-time** stream processing
- All 3 need faster **data sharing** across parallel jobs

credits:
Matei Zaharia & Xiangrui Meng

Data Sharing in MapReduce

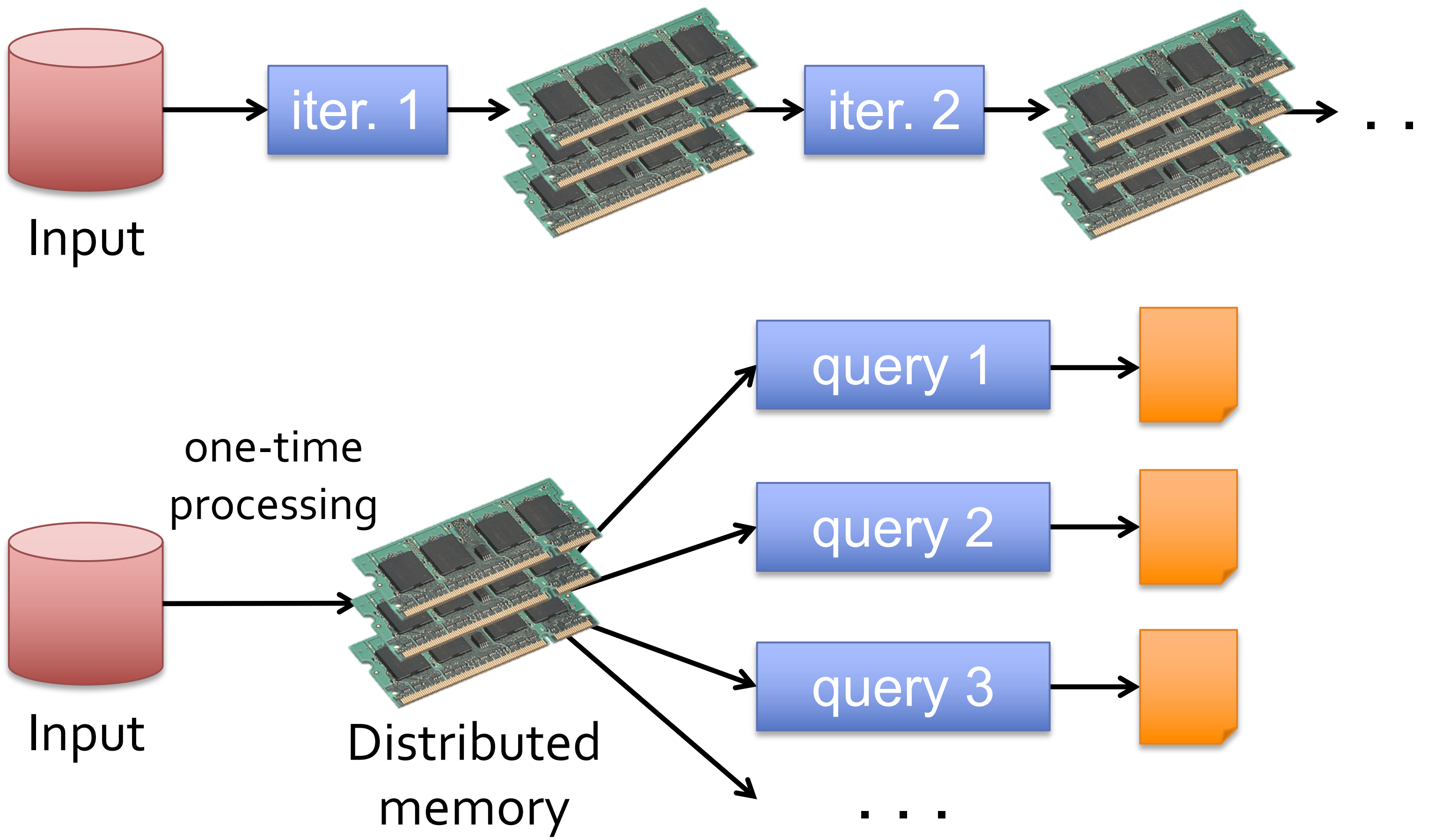




Slow due to replication, serialization, and disk IO

credits:
Matei Zaharia & Xiangrui Meng

Data Sharing in Spark



~10× faster than network and disk

credits:
Matei Zaharia & Xiangrui Meng

Spark Programming Model

- Key idea: *resilient distributed datasets (RDDs)*
  - Distributed collections of objects that can be cached in memory across the cluster
  - Manipulated through parallel operators
  - Automatically *recomputed* on failure
- Programming interface
  - Functional APIs in Scala, Java, Python
  - Interactive use from Scala shell

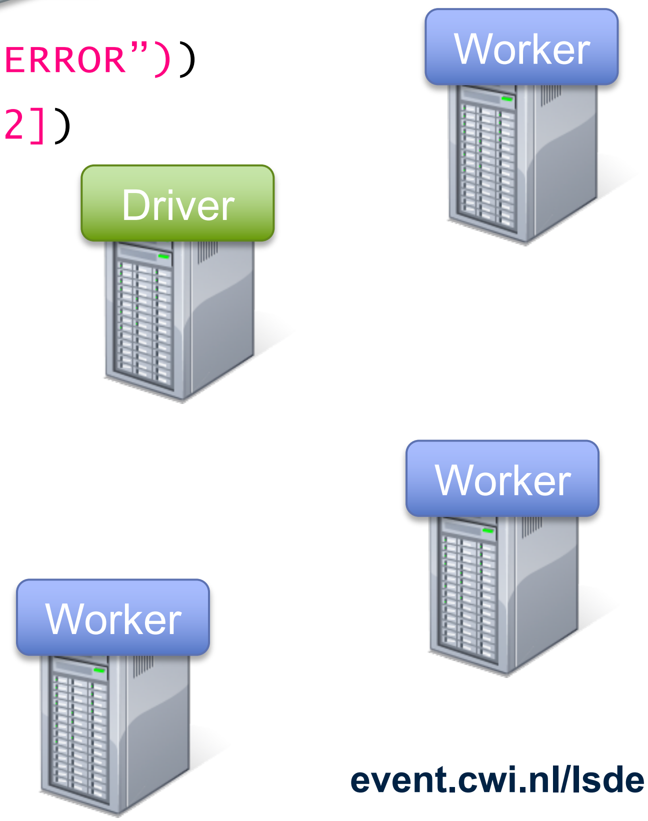
credits:
Matei Zaharia & Xiangrui Meng

Example: Log Mining

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

Bas
Transformed RDD

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda x: x.startswith("ERROR"))
messages = errors.map(lambda x: x.split('\t')[2])
messages.cache()
```



credits:
Matei Zaharia & Xiangrui Meng

Lambda Functions

```
errors = lines.filter(lambda x: x.startswith(“ERROR”))
messages = errors.map(lambda x: x.split(‘\t’)[2])
```

Lambda function 🡐 functional programming!

= implicit function definition

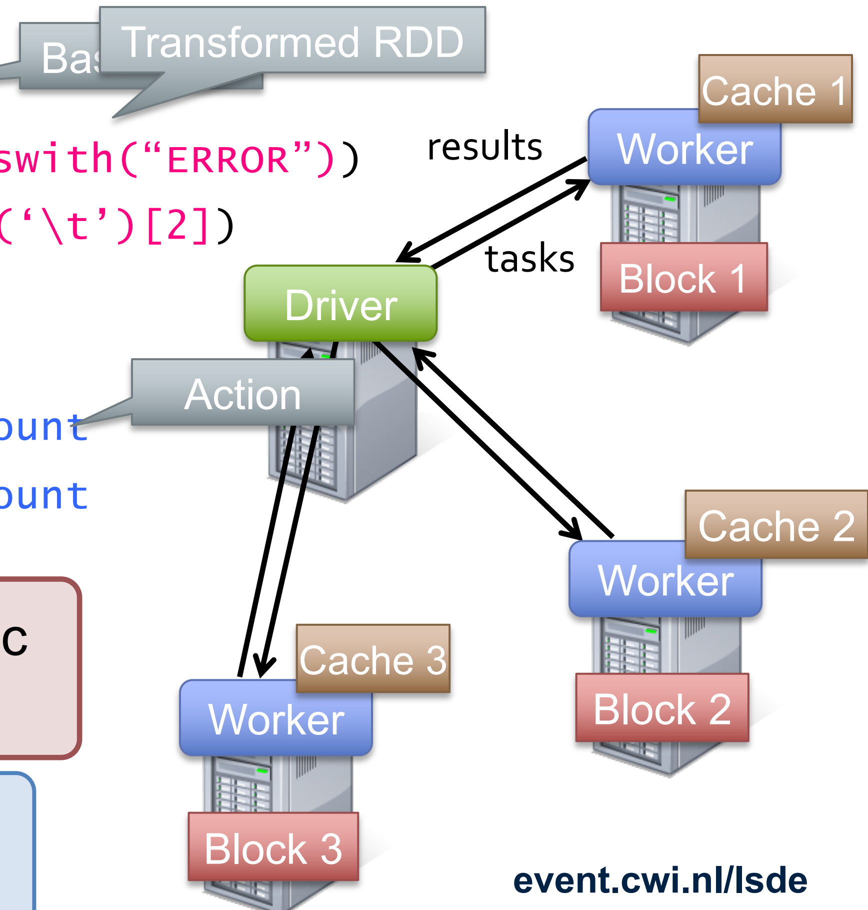
```
bool detect_error(string x) {
    return x.startswith(“ERROR”);
}
```

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 x: x.startswith("ERROR"))
messages = errors.map(lambda x: x.split('\t')[2])
messages.cache()

messages.filter(lambda x: "foo" in x).count
messages.filter(lambda x: "bar" in x).count
. . .
```



Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

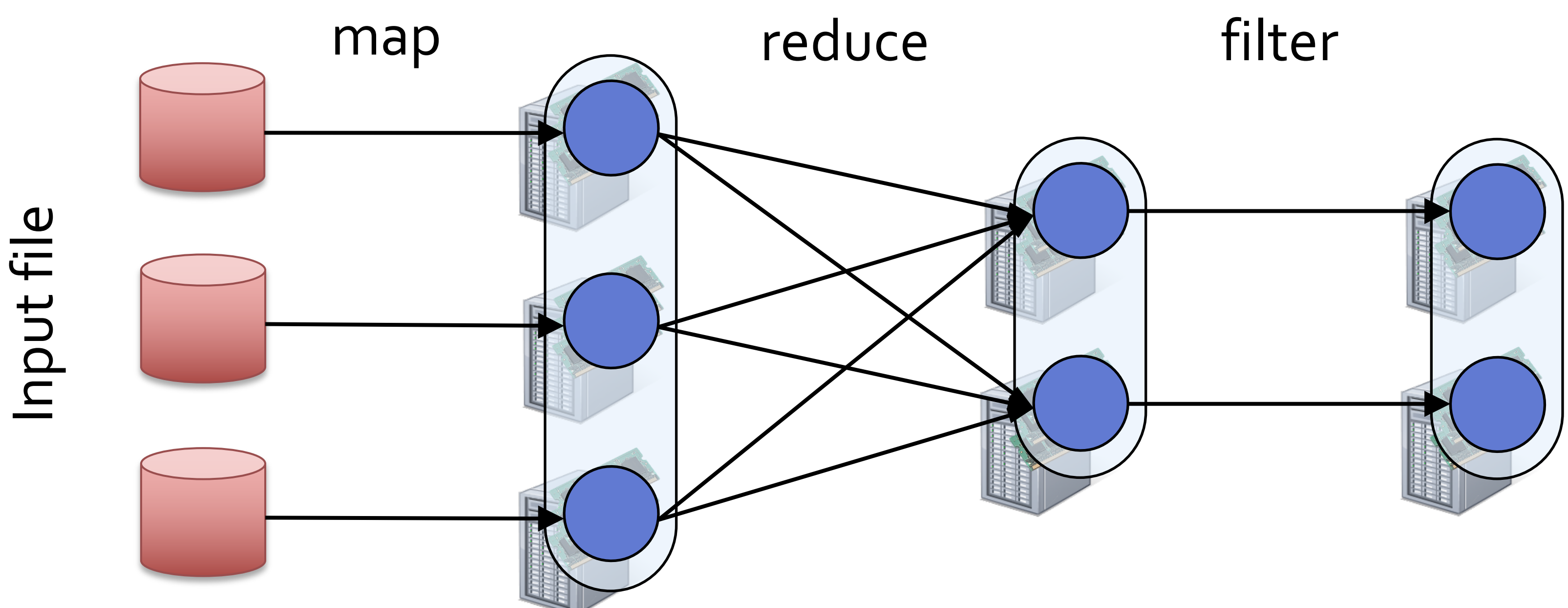
Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

credi
Mate

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

- ```
  file.map(lambda rec: (rec.type, 1))
      .reduceByKey(lambda x, y: x + y)
      .filter(lambda (type, count): count > 10)
  ```

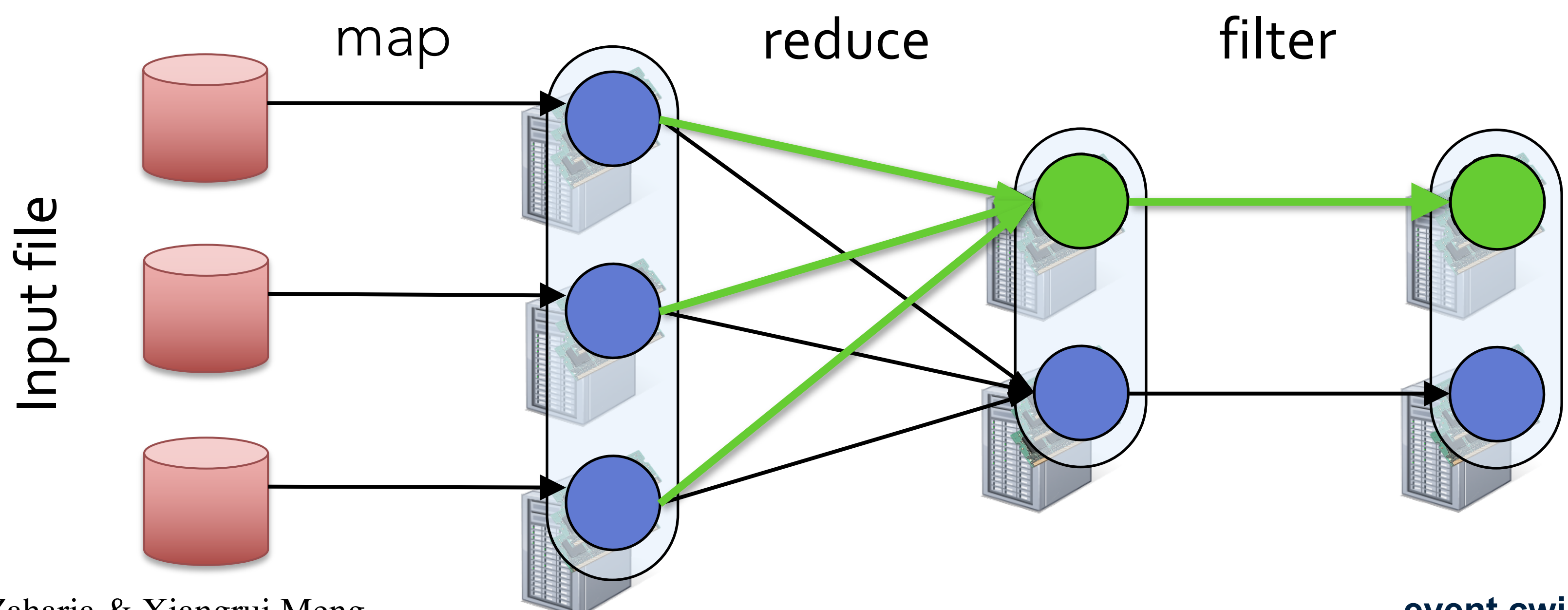


credits:
Matei Zaharia & Xiangrui Meng
  ```

Fault Tolerance

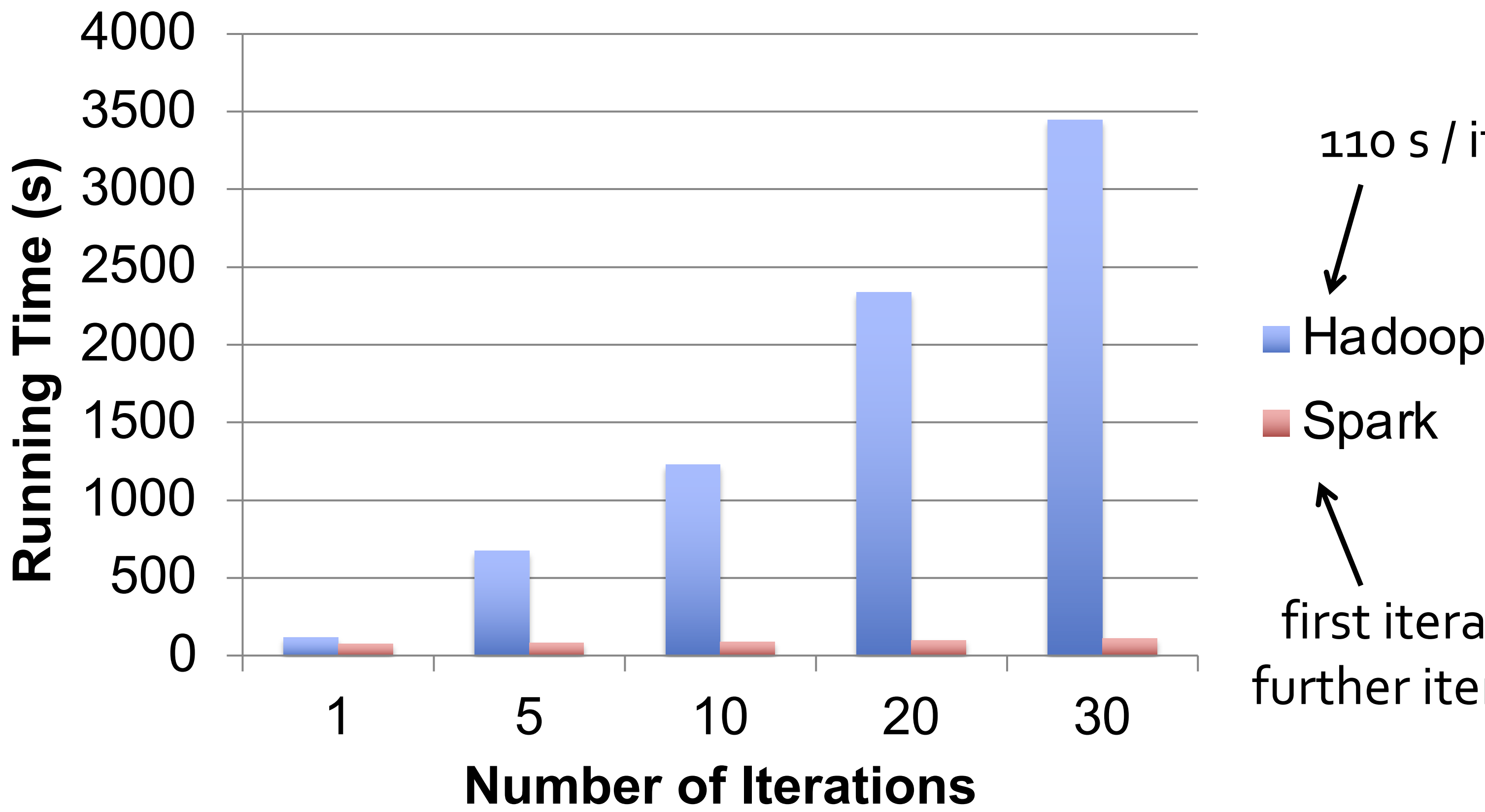
RDDs track *lineage* info to rebuild lost data

- ```
  file.map(lambda rec: (rec.type, 1))
      .reduceByKey(lambda x, y: x + y)
      .filter(lambda (type, count): count > 10)
  ```



credits:
Matei Zaharia & Xiangrui Meng
  ```

Example: Logistic Regression

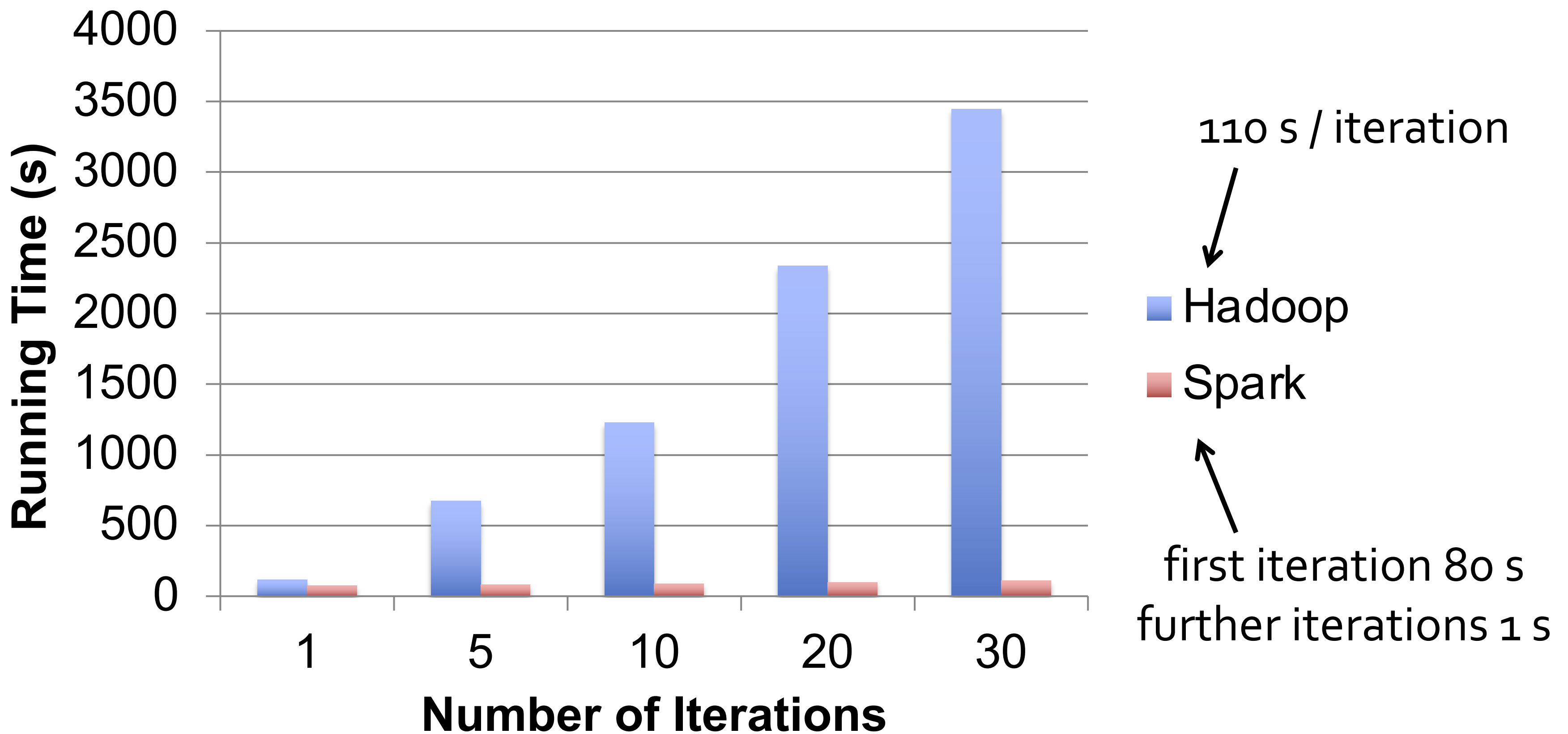


110 s / iteration

first iteration 80 s
further iterations 1 s

credits:
Matei Zaharia & Xiangrui Meng

Example: Logistic Regression



credits:
Matei Zaharia & Xiangrui Meng

Spark in Scala and Java

```
// 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();
```

credits:
Matei Zaharia & Xiangrui Meng

Supported Operators

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin

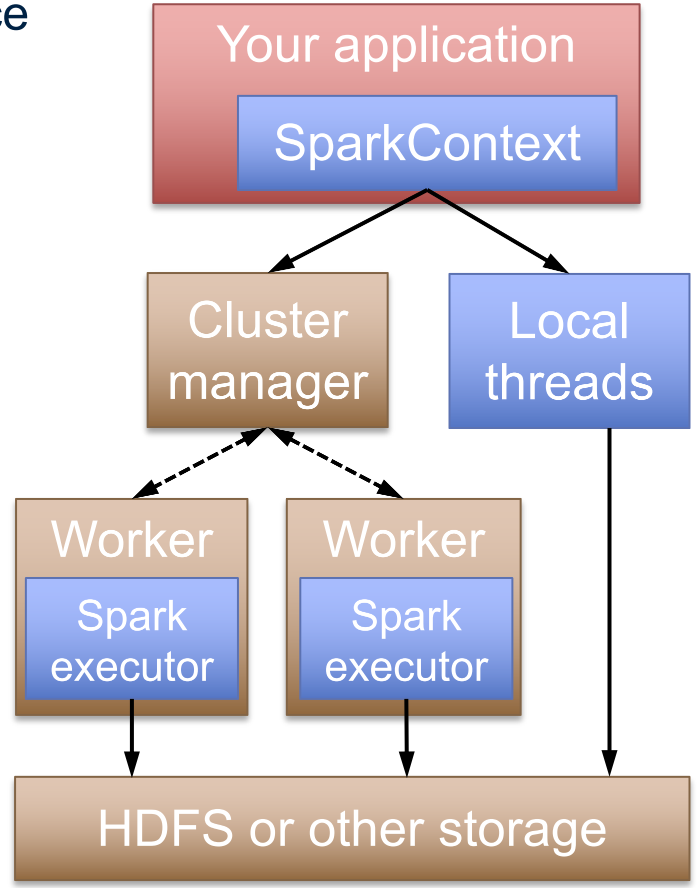
- reduce
- count
- fold
- reduceByKey
- groupByKey
- cogroup
- cross
- zip

sample
take
first
partitionBy
mapWith
pipe
save
...

credits:
Matei Zaharia & Xiangrui Meng

Software Components

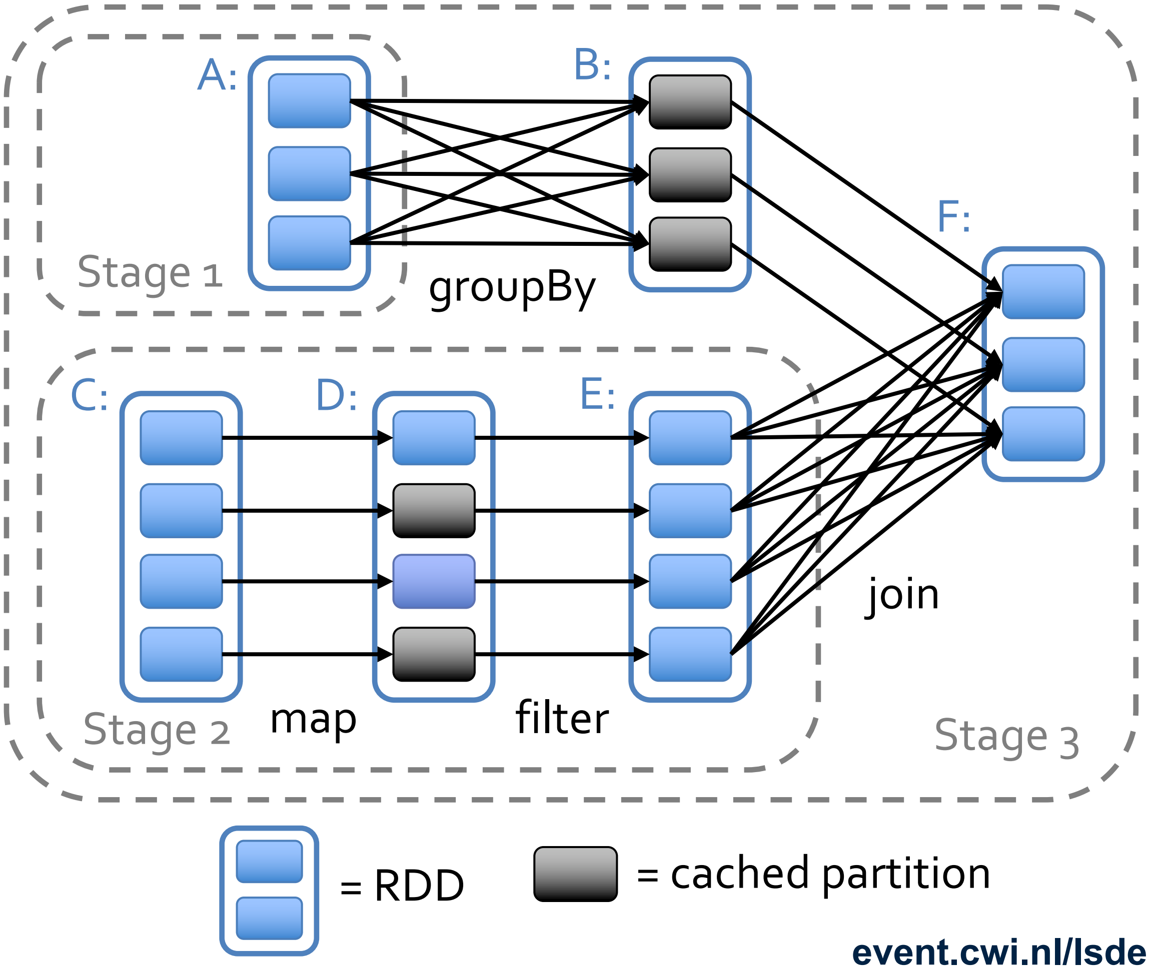
- Spark client is library in user program (1 instance per app)
- Runs tasks locally or on cluster
  – Mesos, YARN, standalone mode
- Accesses storage systems via Hadoop InputFormat API
  – Can use HBase, HDFS, S3, …



credits:
Matei Zaharia & Xiangrui Meng

Task Scheduler

- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles

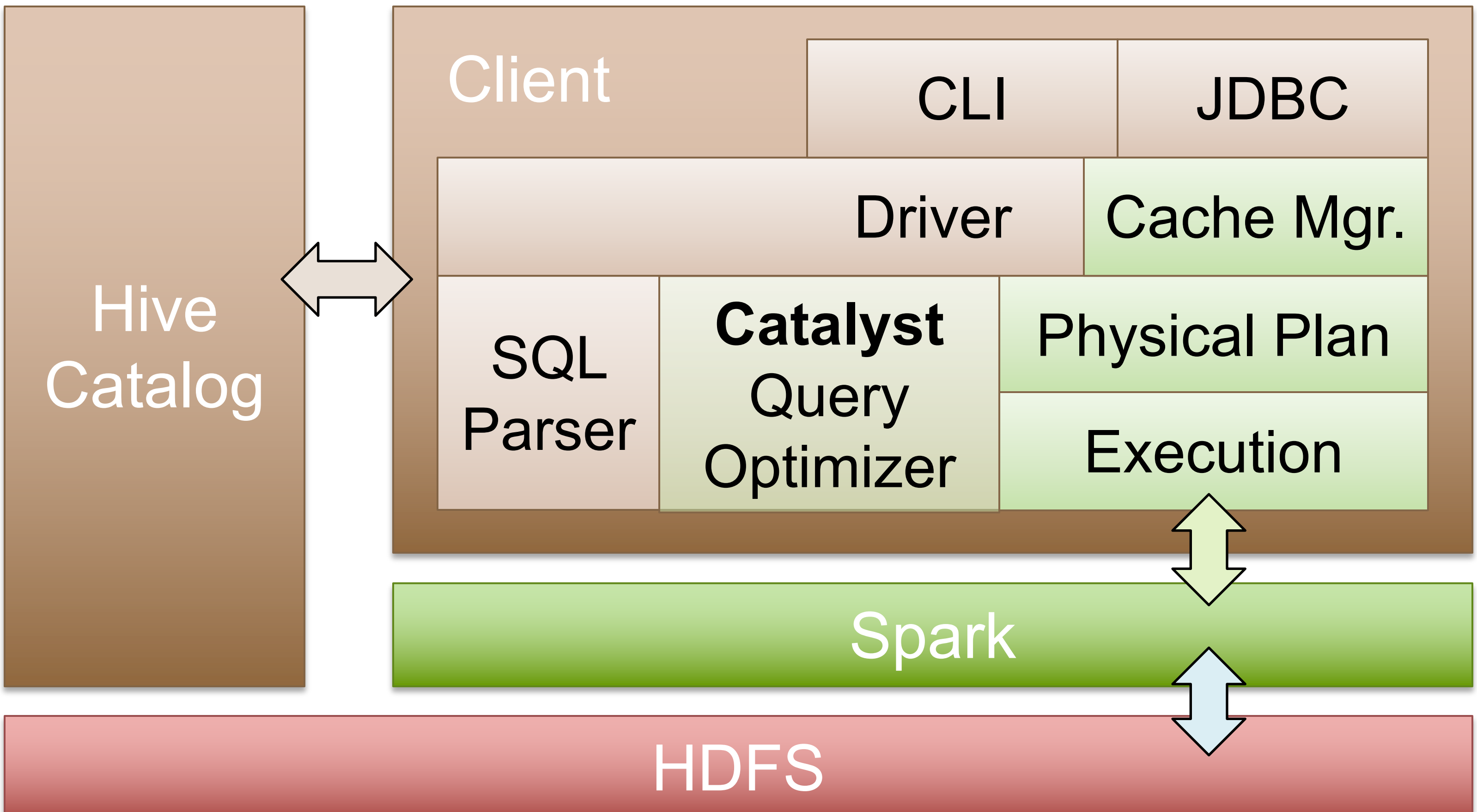


credits:
Matei Zaharia & Xiangrui Meng

Spark SQL

- Columnar SQL analytics engine for Spark
  - Support both SQL and complex analytics
  - Columnar storage, JIT-compiled execution, Java/Scala/Python UDFs
  - Catalyst query optimizer (also for DataFrame scripts)

Spark SQL Architecture



[Engle et al, SIGMOD 2012]

credits:
Matei Zaharia & Xiangrui Meng

From RDD to DataFrame

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```

- A distributed collection of rows with the same schema (RDDs suffer from type erasure)
- Can be constructed from external data sources or RDDs into essentially an RDD of Row objects (SchemaRDDs as of Spark < 1.3)
- Supports relational operators (e.g. *where*, *groupby*) as well as Spark operations.
- Evaluated lazily → non-materialized *logical* plan

credits:
Matei Zaharia & Reynold Xin

DataFrame: Data Model

- Nested data model
- Supports both primitive SQL types (boolean, integer, double, decimal, string, data, timestamp) and complex types (structs, arrays, maps, and unions); also user defined types.
- First class support for complex data types

credits:
Matei Zaharia & Reynold Xin

DataFrame Operations

- Relational operations (select, where, join, groupBy) via a DSL
- Operators take *expression* objects
- Operators build up an abstract syntax tree (AST), which is then optimized by *Catalyst*.

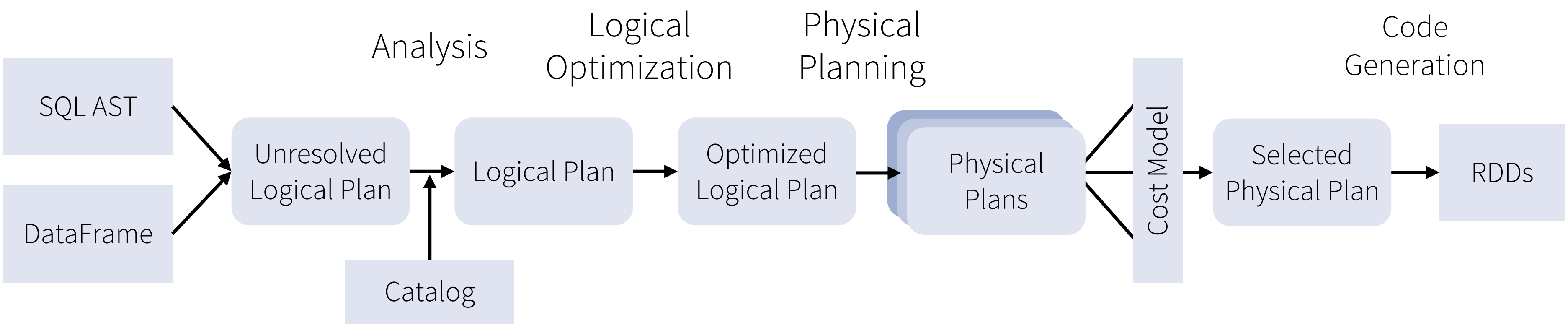
```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

- Alternatively, register as temp SQL table and perform traditional SQL query strings

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

credits:
Matei Zaharia & Reynold Xin

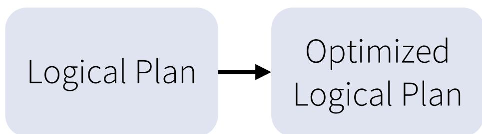
Catalyst: Plan Optimization & Execution



credits:
Matei Zaharia & Reynold Xin

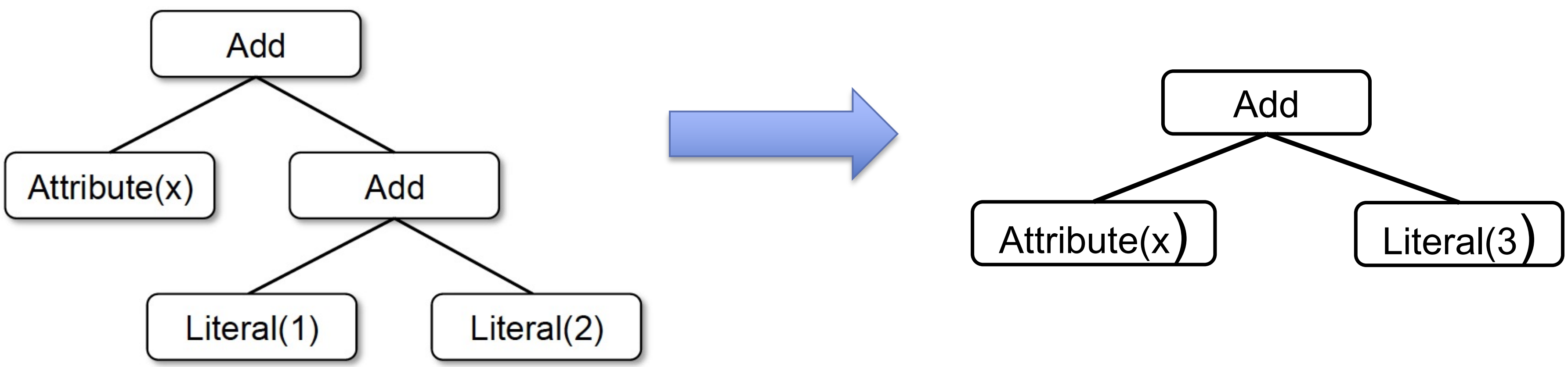
Catalyst Optimization Rules

Logical Optimization



- Applies standard rule-based optimization (constant folding, predicate-pushdown, projection pruning, null propagation, boolean expression simplification, etc)

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```

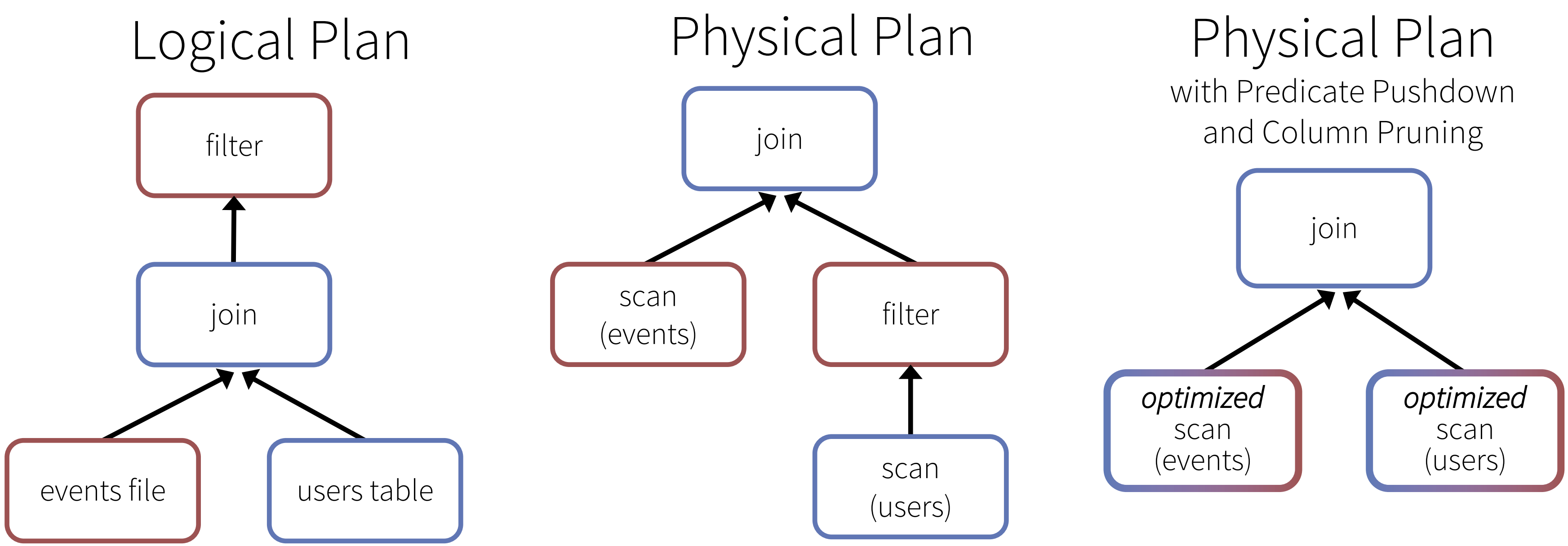


$x + (1 + 2)$

$x + 3$

credits:
Matei Zaharia & Reynold Xin

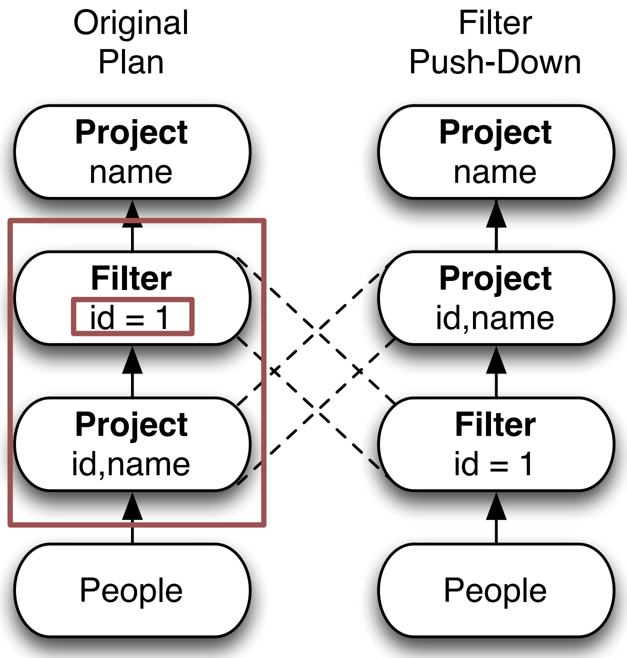
```
def add_demographics(events):
   u = sqlCtx.table("users")                     # Load partitioned Hive table
   events \
     .join(u, events.user_id == u.user_id) \     # Join on user_id
     .withColumn("city", zipToCity(df.zip))      # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "Melbourne").select(events.timestamp).collect()
```



credits:
Matei Zaharia & Reynold Xin

An Example Catalyst Transformation

1. Find filters on top of projections.
2. Check that the filter can be evaluated without the result of the project.
3. If so, switch the operators.



credits:
Matei Zaharia & Reynold Xin

Other Spark Stack Projects

We will revisit **Spark SQL** in the **SQL on Big Data** lecture

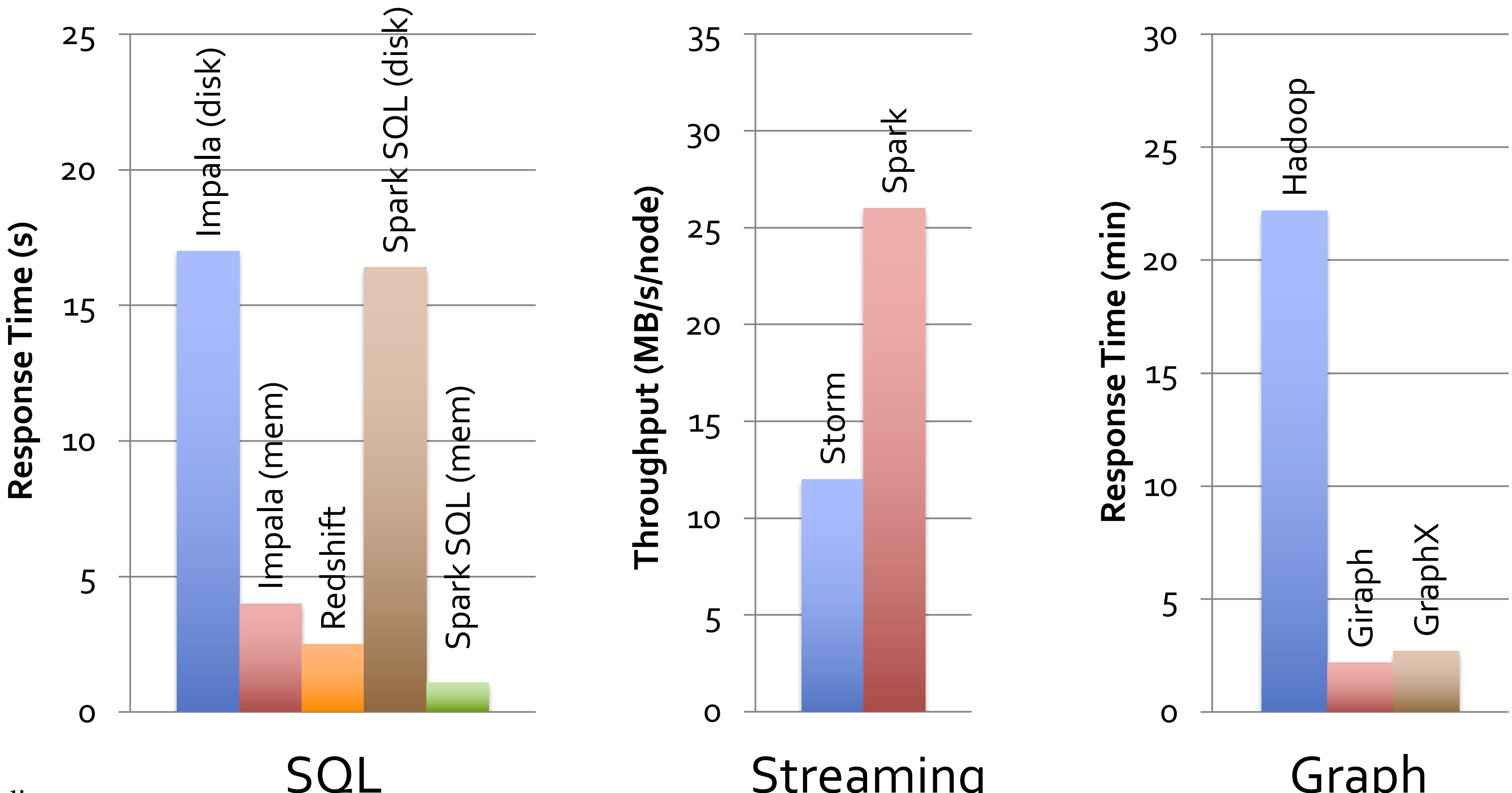
- **Structured Streaming:** stateful, fault-tolerant stream
  - ```
    sc.twitterStream(...)
       .flatMap(_.getText.split(" "))
       .map(word => (word, 1))
       .reduceByWindow("5s", _ + _)
    ```
  - we will revisit **structured streaming** in the Data Streaming lecture

this lecture, still:

- **GraphX & GraphFrames:** graph-processing framework
- **MLlib:** Library of high-quality machine learning algorithms

credits:
Matei Zaharia & Xiangrui Meng
    ```

Performance

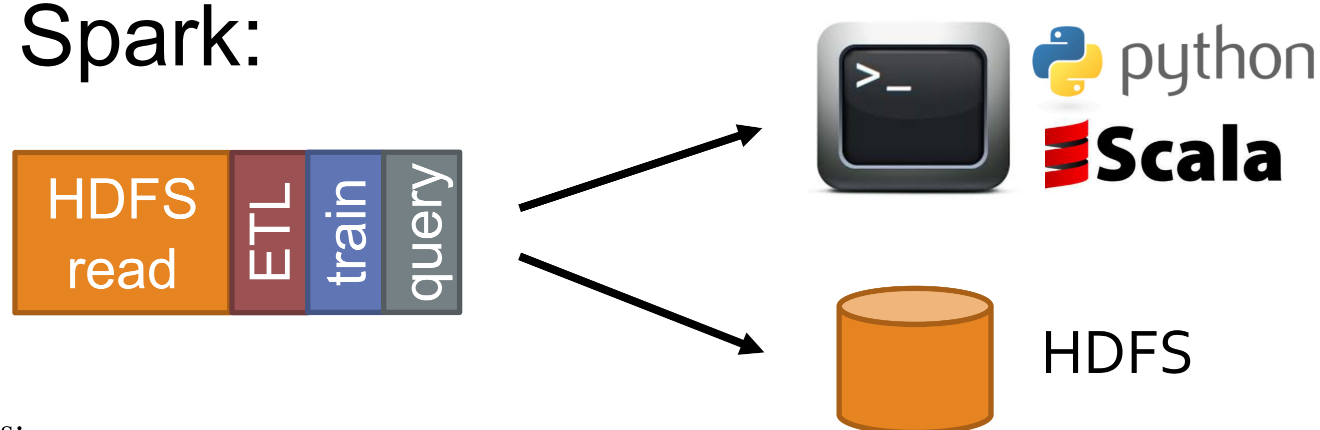


credits:
Matei Zaharia & Xiangrui Meng

What it Means for Users

- Separate frameworks:

Spark:



credits:
Matei Zaharia & Xiangrui Meng

Summary

- Hadoop: The MapReduce Framework
  - The first to simplify parallel processing on big data
    - You write two functions (Map, Reduce), runtime does the rest
    - Tight coupling with HDFS (distributed file system), for locality
  - First generic Big Data platform
    - 2.0 split functionality into HDFS, YARN and MapReduce
    - Still popular on-premise, HDFS/YARN often combined with other tools
- The Spark Framework
  - Generalize Map(),Reduce() to a much larger set of operations
    - Join, filter, group-by, …➔ closer to database queries
  - Tight coupling with Streaming, ML and Graph APIs
  - High(er) performance (than MapReduce)
    - In-memory caching, catalyst query optimizer, JIT compilation, ..
    - More schema knowledge: RDDs ➔ DataFrames