Big Data

Hands on Spark

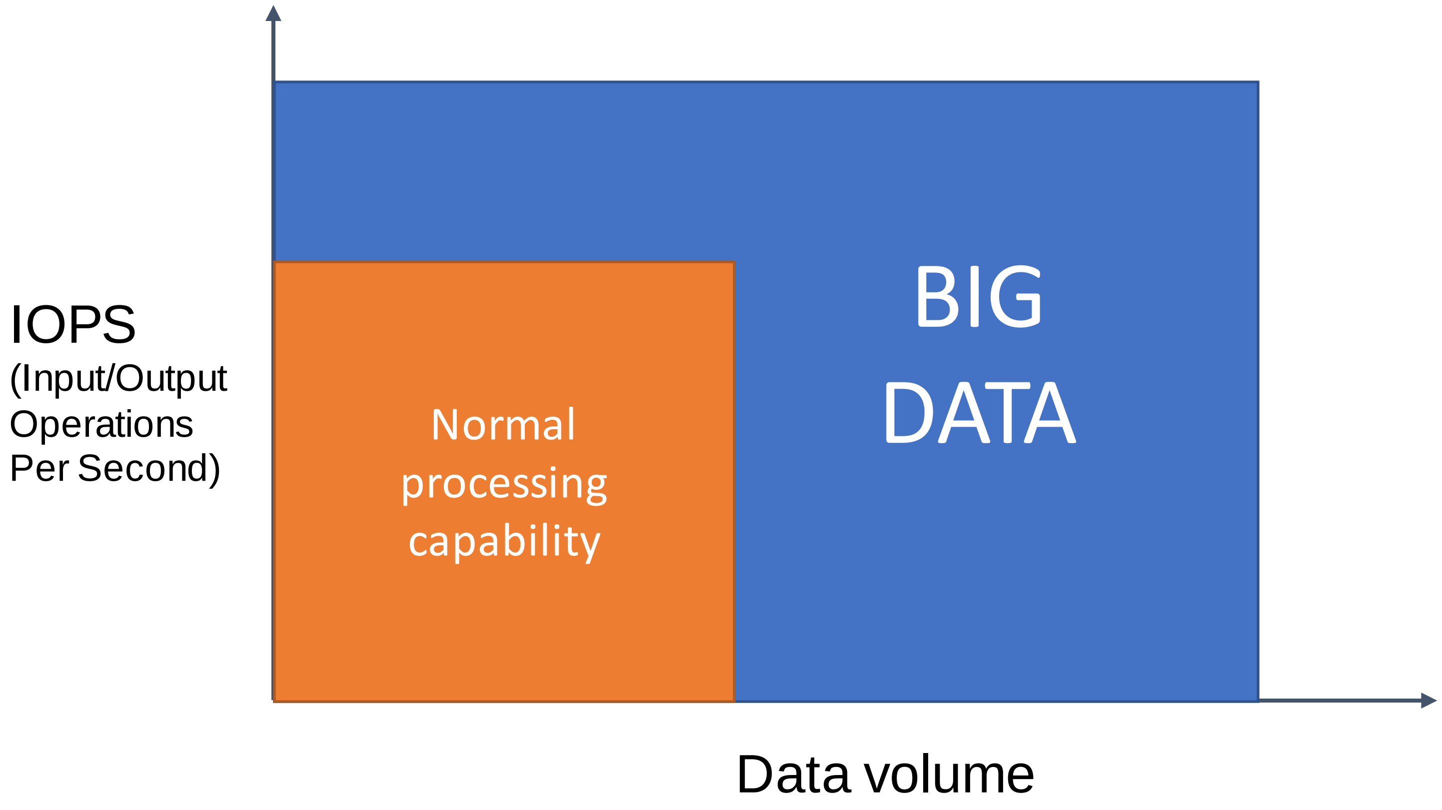
A definition for Big Data

"Big data **exceeds the reach of commonly used hardware** environments **and software** tools to capture, manage, and process it with in a tolerable elapsed time for its user population." - *Teradata Magazine article*, 2011

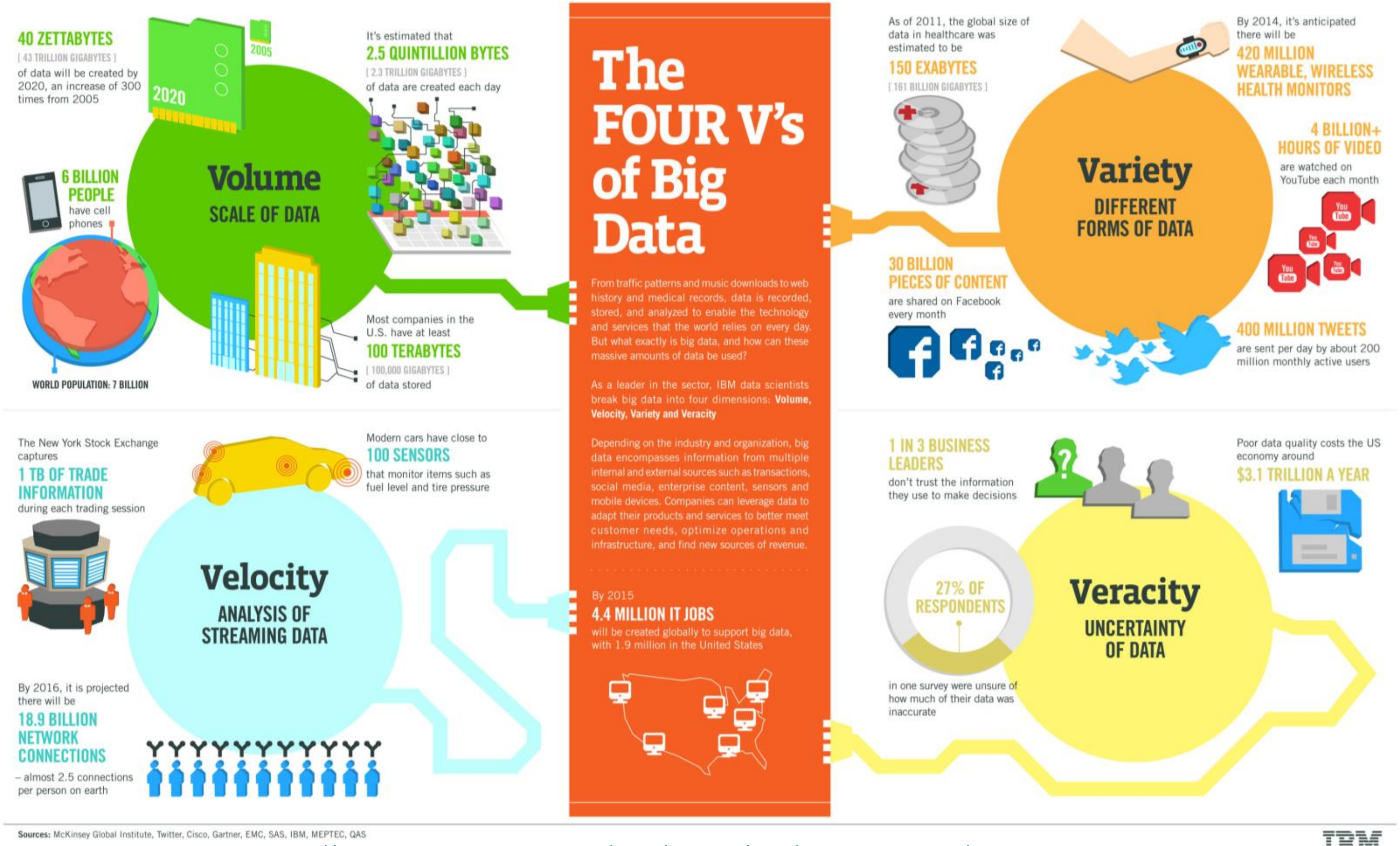
"Big data refers to data sets whose size is **beyond the ability of typical** database **software tools** to capture, store, manage and analyze." - *The McKinsey Global Institute*, 2012

"Big data is data sets that are so voluminous and complex that **traditional** data processing application **softwares are inadequate** to deal with them." - *Wikipedia*

When does data become "Big"?



The four "V's" of Big Data



http://www.ibmbigdatahub.com/sites/default/files/infographic_file/4-Vs-of-big-data.jpg

How do we process Big Data?

Main issues

- Where do we store the data?
- How do we process it?

Big Data greatly exceeds the size of the typical drives

- Even if a big drive existed, it would be too slow (at least for now)

Year: 1990
Size: 1.3 GB
Speed: 4,4 MB/s

5 minutes

Year: 2014
Size: 1 TB
Speed: 100 MB/s

3 hours

Year: 2015
Size: 1 TB
Speed: 600 MB/s

30 minutes

Scale up

Adding more processors and RAM, buying expensive and robust server

Pros

- Less power consumption than running multiple servers
- Cooling costs are less than scaling horizontally
- Generally less challenging to implement
- Less licensing costs
- Less networking equipment

Cons

- PRICE
- Greater risk of hardware failure causing bigger outages
- Generally severe vendor lock-in
- Not long-term: limited upgradeability in the future

Scale out

Adding more servers with less processors and RAM

Pros

- Much cheaper than scaling vertically
- New technologies simplify fault-tolerance and systems monitoring
- Easy to upgrade
- Usually cheaper
- Can literally scale infinitely

Cons

- More licensing fees
- Bigger footprint in the Data Center
- Higher utility cost (electricity and cooling)
- Possible need for more networking equipment (switches/routers)

Commodity hardware

You are not tied to expensive, proprietary offerings from a single vendor

You can choose standardized, commonly available hardware from a large range of vendors to build your cluster

Commodity ≠ Low-end!

- Cheap components with high failure rate can be a false economy

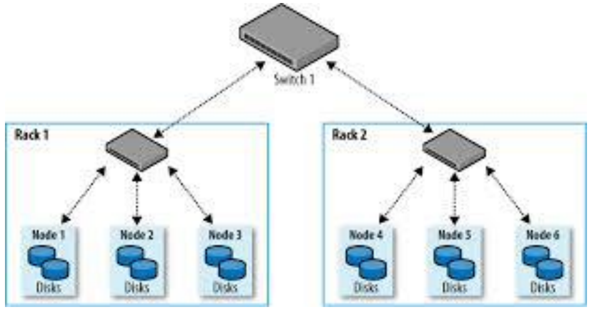
The answer: cluster computing

100 hard disks? 2 mins to read 1TB

Cluster computing

Compute nodes are stored on racks

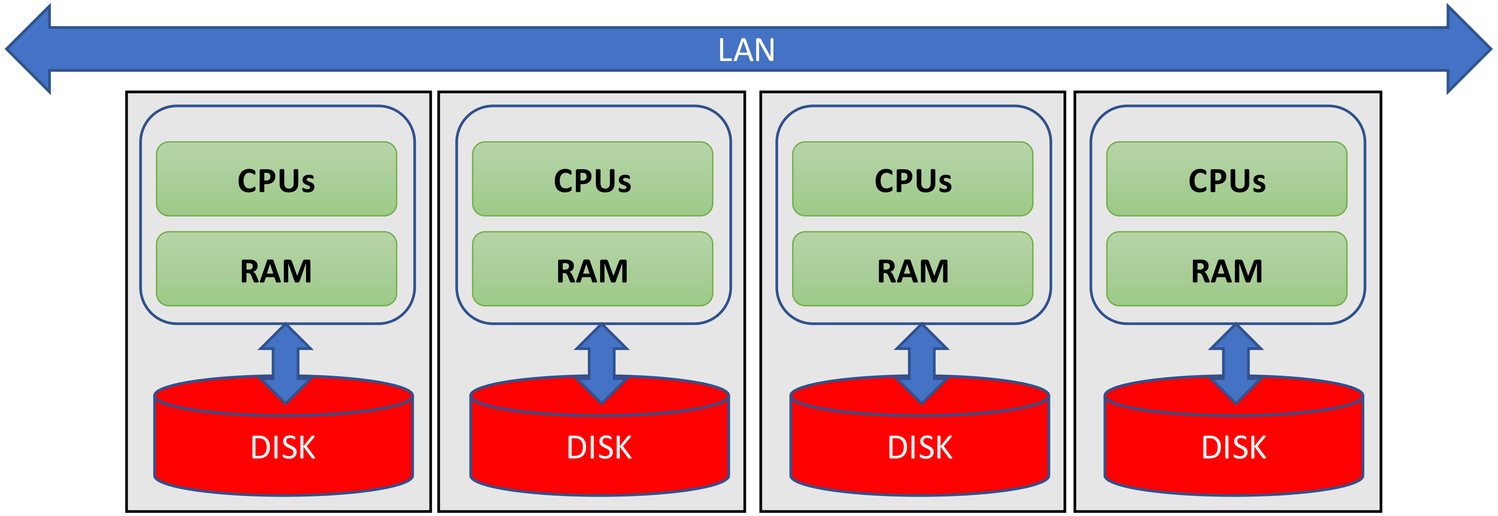
- 8–64 compute nodes on a rack
- There can be many racks of compute nodes
- The nodes on a single rack are connected by a network (typically gigabit Ethernet)
- Racks are connected by another level of network (or a switch)
    - The bandwidth of intra-rack communication is usually much greater than that of inter-rack communication



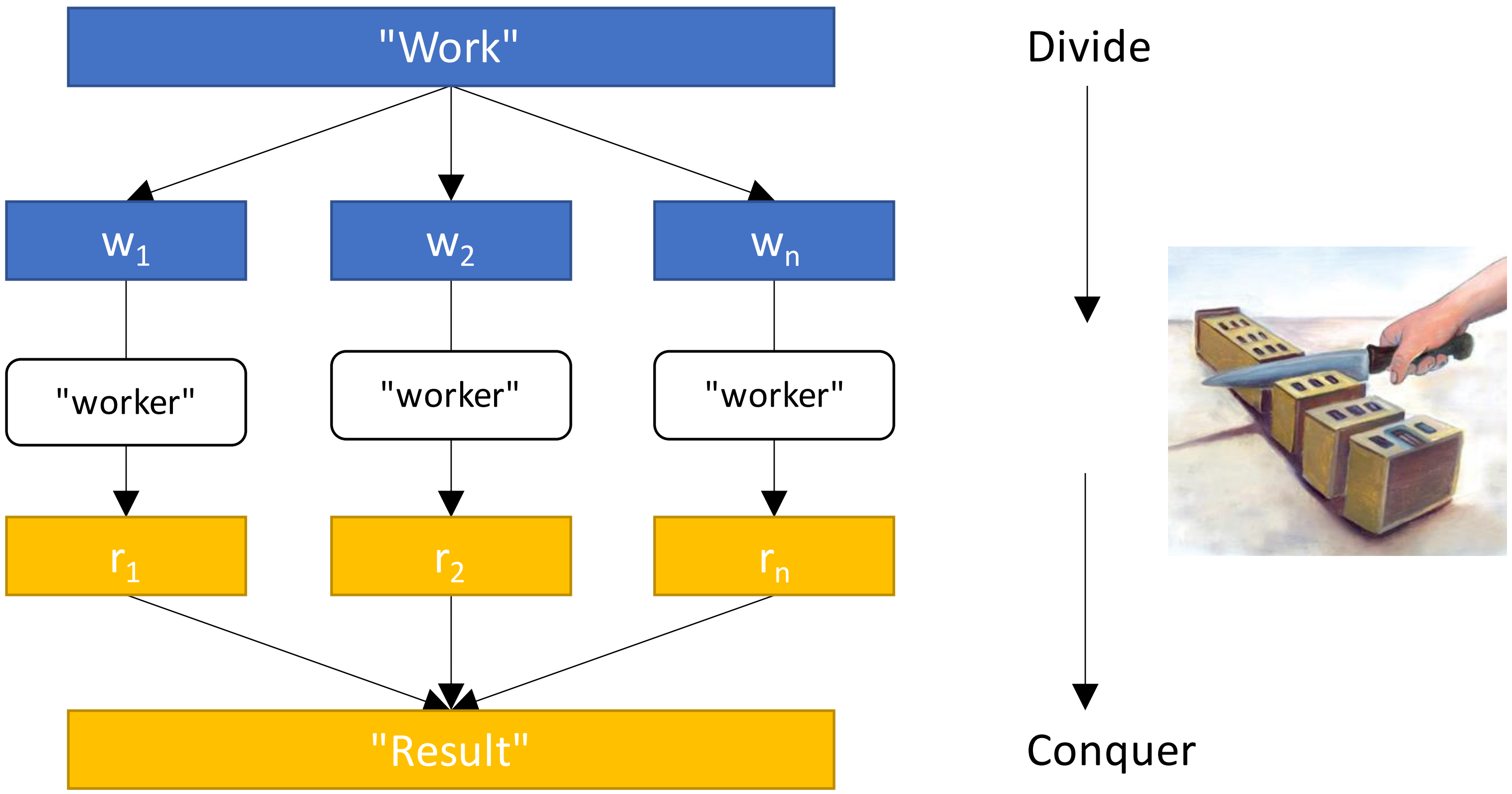
Cluster Computing Architecture

A computer cluster is a group of linked computers (nodes), working together closely so that in many respects they form a single computer

- Typically connected to each other through fast LAN
- Every node is a system on its own, capable of independent operations
  - Unlimited scalability, no vendor lock-in
- Number of nodes in the cluster >> Number of CPUs in a node



Distributed computing: an old idea



Parallelization 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 aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

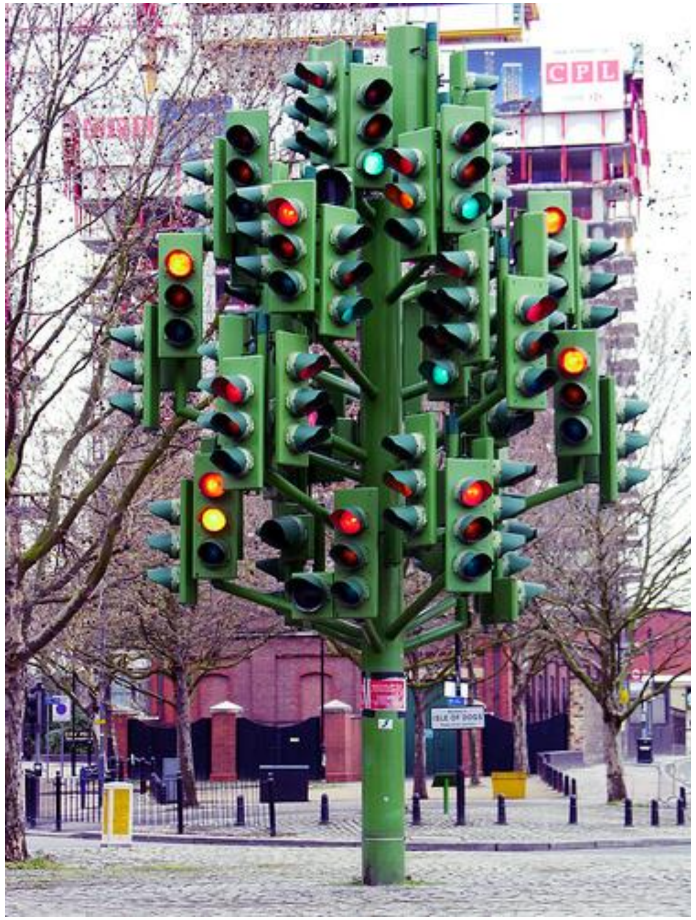
Risks?

Deadlock and starvation

Parallelization problems arise from:

- Communication between workers (e.g., to exchange state)
- Access to shared resources (e.g., data)

We need synchronization



But...

It is difficult to reason about parallelization

It is even more difficult to reason about concurrency

- At the scale of datacenters (even across datacenters)
- In the presence of failures
- In terms of multiple interacting services

Not to mention debugging…

The reality can be hard

- Lots of one-off solutions, custom code
- Write your own dedicated library, then program with it
- Burden on the programmer to explicitly manage everything

What is the solution?

Hide system-level details from the developers

- No more race conditions, lock contention, etc.
- No need to become hardcore techies

Separate the *what* from the *how*

- Developer specifies the computation that needs to be performed
- Execution framework ("runtime") handles the actual execution

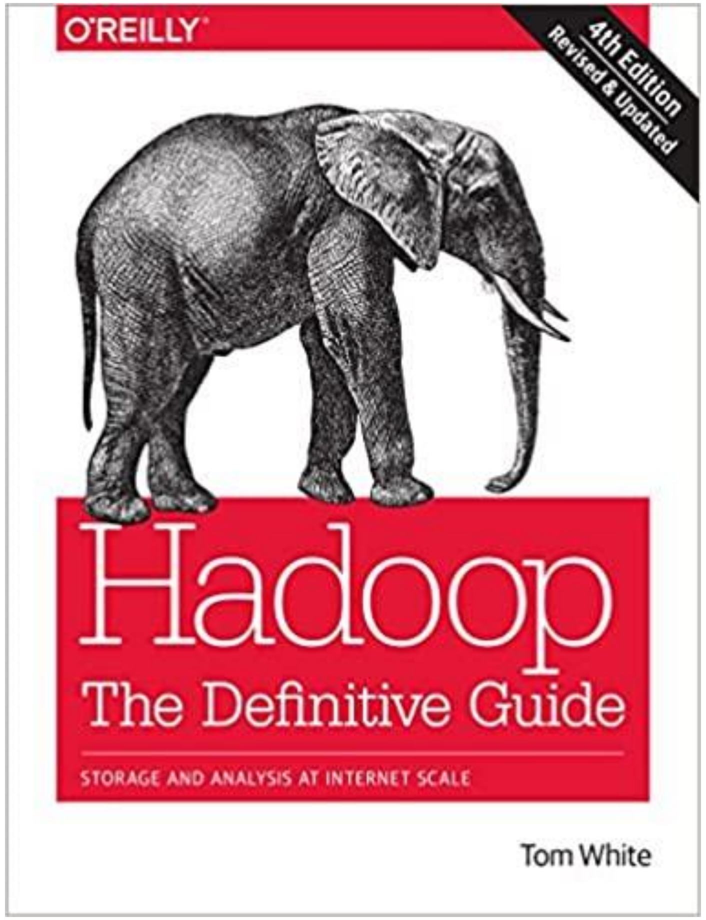
The datacenter IS the computer!

Disambiguation of MapReduce

"MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key."

-- Dean J., Ghemawat S. (Google)

Hadoop MapReduce is an open-source implementation of the MapReduce programming model



Limitations of Map Reduce

Designed for batch processing

- Not suitable for iterative algorithms or interactive data mining

Strict paradigm

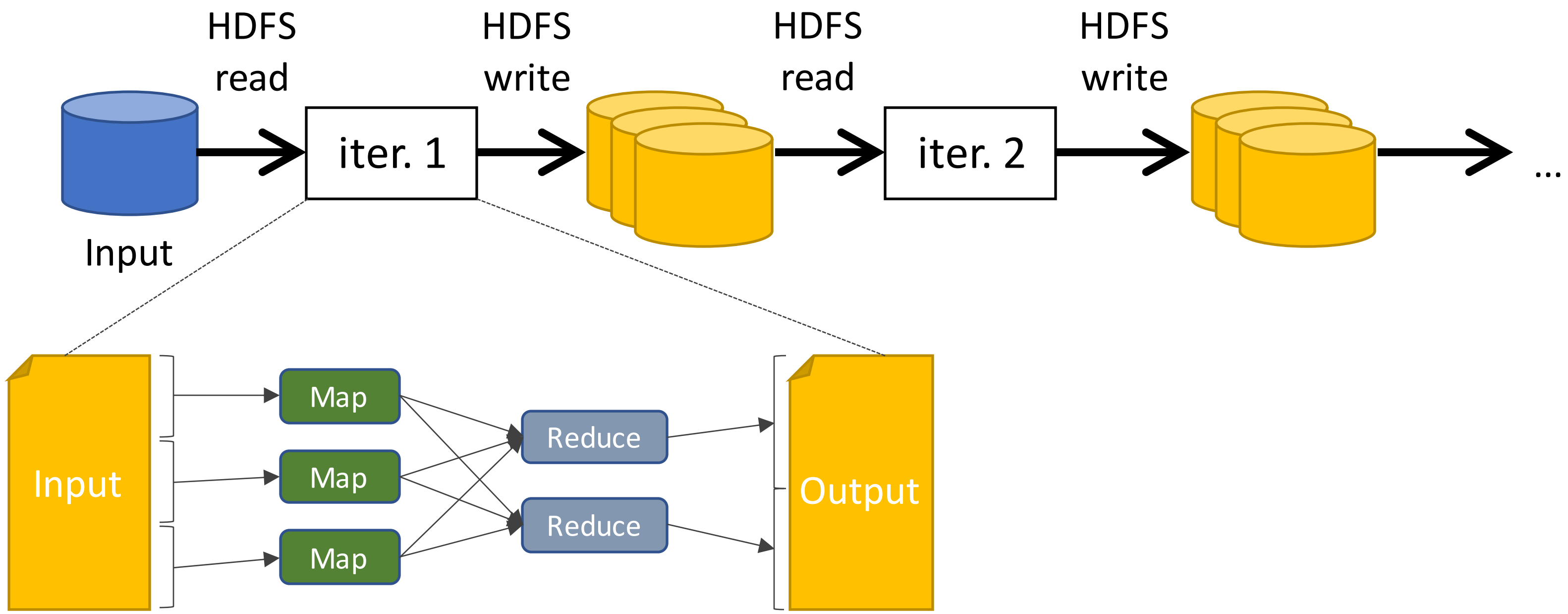
- Everything has to fit into Map and Reduce
- Complex algorithms will take multiple jobs and passes on hard disk

New hardware capabilities are not exploited

- Too much pressure on disk; RAM and multicore not adequately exploited

Too much complex

Limitations of Map Reduce



Spark

It is a fast and general-purpose execution engine

- In-memory data storage for very fast iterative queries
- Easy interactive data analysis
- Combines different processing models (machine learning, SQL, streaming, graph computation)
- Provides (not only) a MapReduce-like engine...
- ... but it's up to 100x faster than Hadoop MapReduce

Compatible with Hadoop's storage APIs

- Can run on top of a Hadoop cluster
- Can read/write to any database and any Hadoop-supported system, including HDFS, HBase, Parquet, etc.

What does Spark offer?

In-memory data caching

- HDD is scanned once, then data is written to/read from RAM

Lazy computations

- The job is optimized before its execution

Efficient pipelining

- Writing to HDD is avoided as much as possible

Spark pillars

Two main abstractions of Spark

RDD – Resilient Distributed Dataset

- An RDD is a collection of data items
- It is split into partitions
- It is stored in memory on the worker nodes of the cluster

DAG – Direct Acyclic Graph

- A DAG is a sequence of computations performed on data
- Each node is an RDD
- Each edge is a transformation of one RDD into another

RDD

RDDs are immutable distributed collection of objects

- **Resilient**: automatically rebuild on failure
- **Distributed**: the objects belonging to a given collection are split into *partitions* and spread across the nodes
  - RDDs can contain any type of Python, Java, or Scala objects
  - Distribution allows for scalability and locality-aware scheduling
  - Partitioning allows to control parallel processing

Fundamental characteristics (mostly from *pure functional programming*)

- **Immutable**: once created, it can’t be modified
- **Lazily evaluated**: optimization before execution
- **Cacheable**: can persist in memory, spill to disk if necessary
- **Type inference**: data types are not declared but inferred (≠ dynamic typing)

RDD operations

RDDs offer two types of operations: *transformations* and *actions*

Transformations construct a new RDD from a previous one

- E.g.: map, flatMap, reduceByKey, filtering, etc.
- https://spark.apache.org/docs/latest/programming-guide.html#transformations

Actions compute a result that is either returned to the driver program or saved to an external storage system (e.g., HDFS)

- E.g.: saveAsTextFile, count, collect, etc.
- https://spark.apache.org/docs/latest/programming-guide.html#actions

RDD operations

RDDs are **lazily evaluated**, i.e., they are computed when they are used in an action

- Until no action is fired, the data to be processed is not even accessed

Example (in Python)

```
sc = new SparkContext
rddLines = sc.textFile("myFile.txt")
rddLines2 = rddLines.filter (lambda line: "some text" in line)
rddLines2.first()
```

Transformations

Action

There is no need to compute and store everything

- In the example, Spark simply scans the file until it finds the first matching line

DAG

Based on the user application and on the lineage graphs,
Spark computes a logical execution plan in the form of a DAG

- Which is later transformed into a physical execution plan

The DAG (Directed Acyclic Graph) is a sequence of computations performed on data

- Nodes are **RDDs**
- Edges are operations on RDDs
- The graph is Directed: transformations from a partition A to a partition B
- The graph is Acyclic: transformations cannot return an old partition

Application decomposition

Application

- Single instance of SparkContext that stores data processing logic and schedules series of jobs, sequentially or in parallel

Job

- Complete set of transformations on RDD that finishes with action or data saving, triggered by the driver application

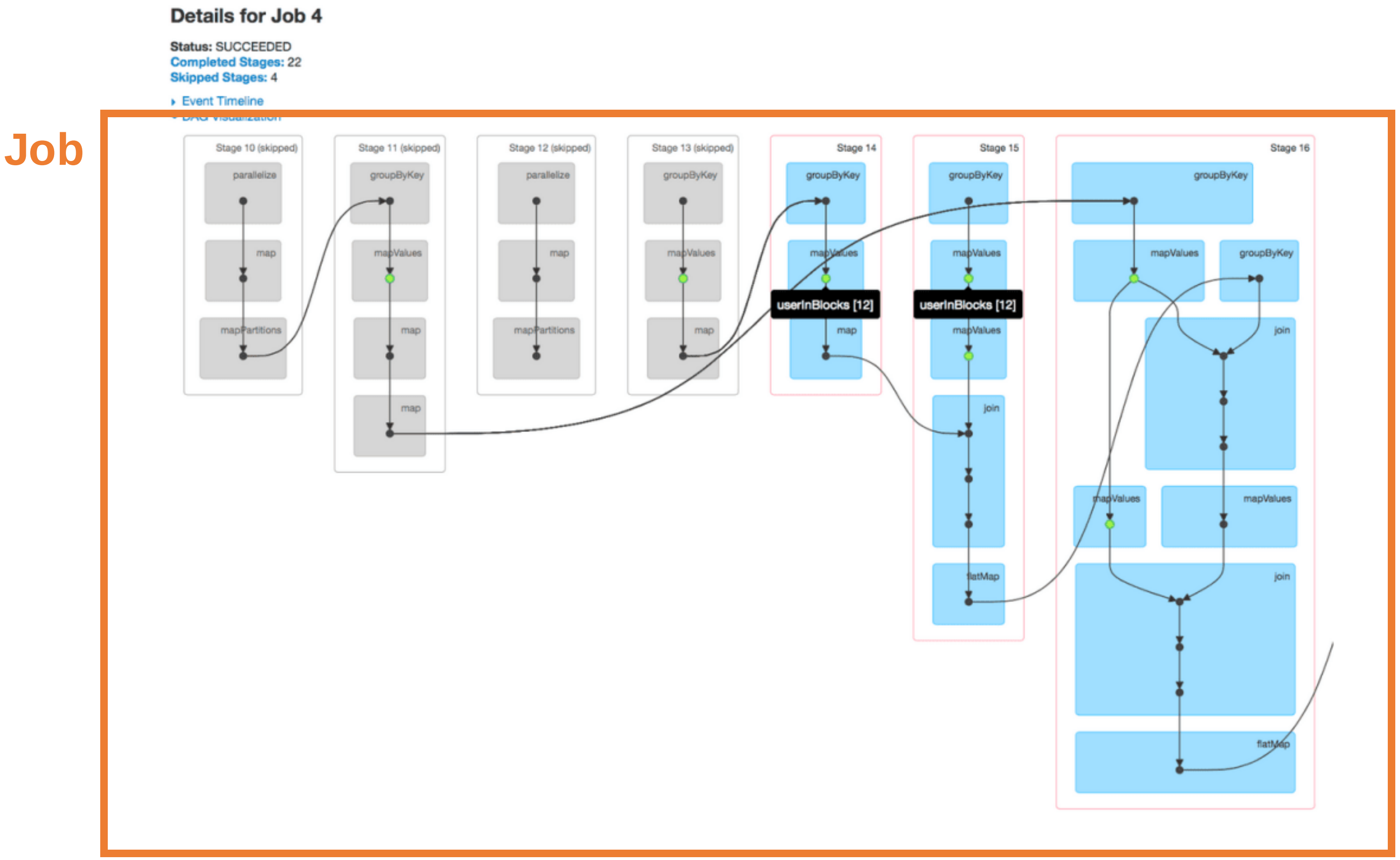
Stage

- Set of transformations that can be pipelined and executed by a single independent worker

Task

- Basic unit of scheduling: executes the stage on a single data partition

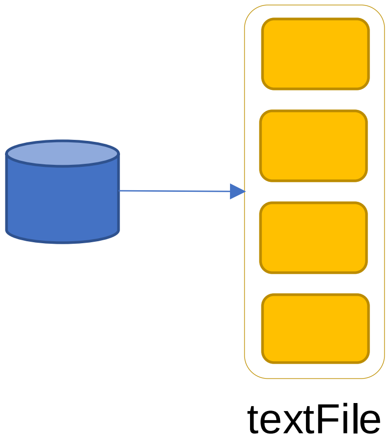
Application decomposition



DAG example

Word count in Scala

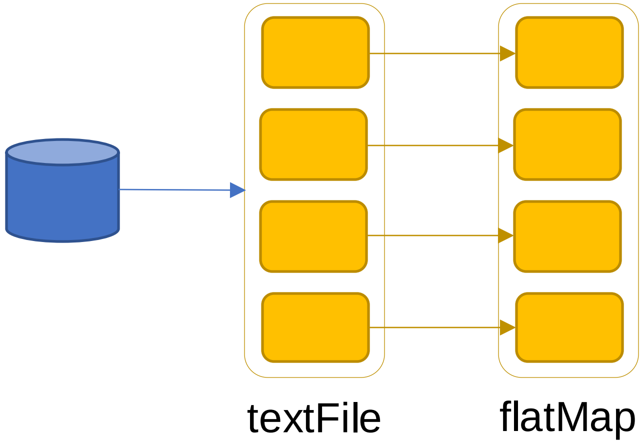
textFile = sc.textFile("hdfs://...")



DAG example

Word count in Scala

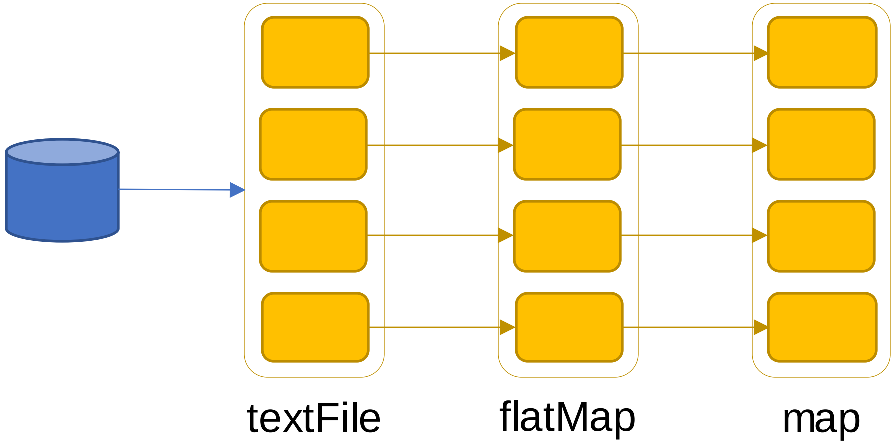
```
textFile = sc.textFile("hdfs://...")
counts = textFile
    .flatMap(line => line.split(" "))
```



DAG example

Word count in Scala

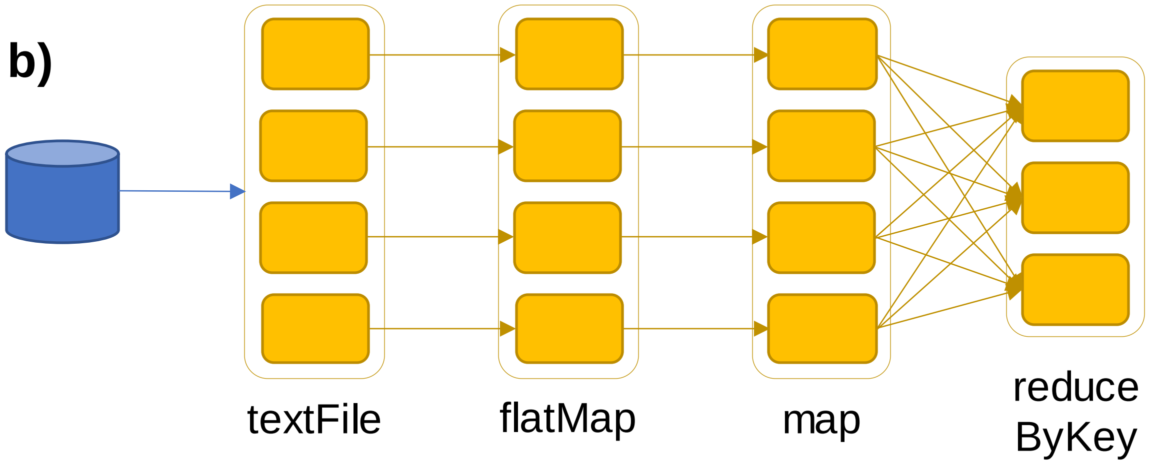
```
textFile = sc.textFile("hdfs://...")
counts = textFile
    .flatMap(line => line.split(" "))
    .map(lambda word: (word, 1))
```



DAG example

Word count in Scala

```
textFile = sc.textFile("hdfs://...")
counts = textFile
    .flatMap(line => line.split(" "))
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
```

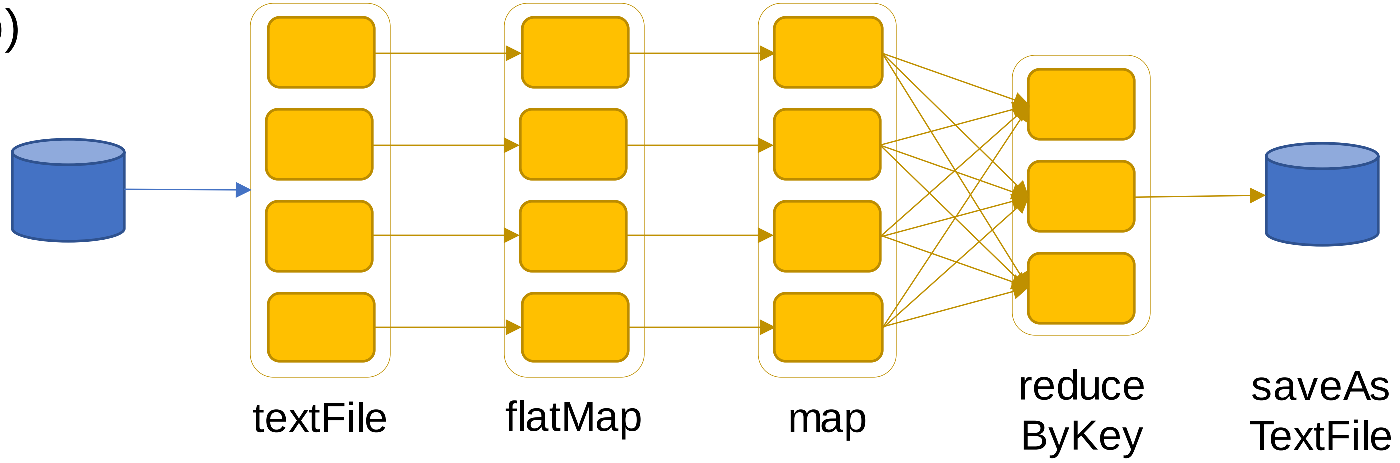


DAG example

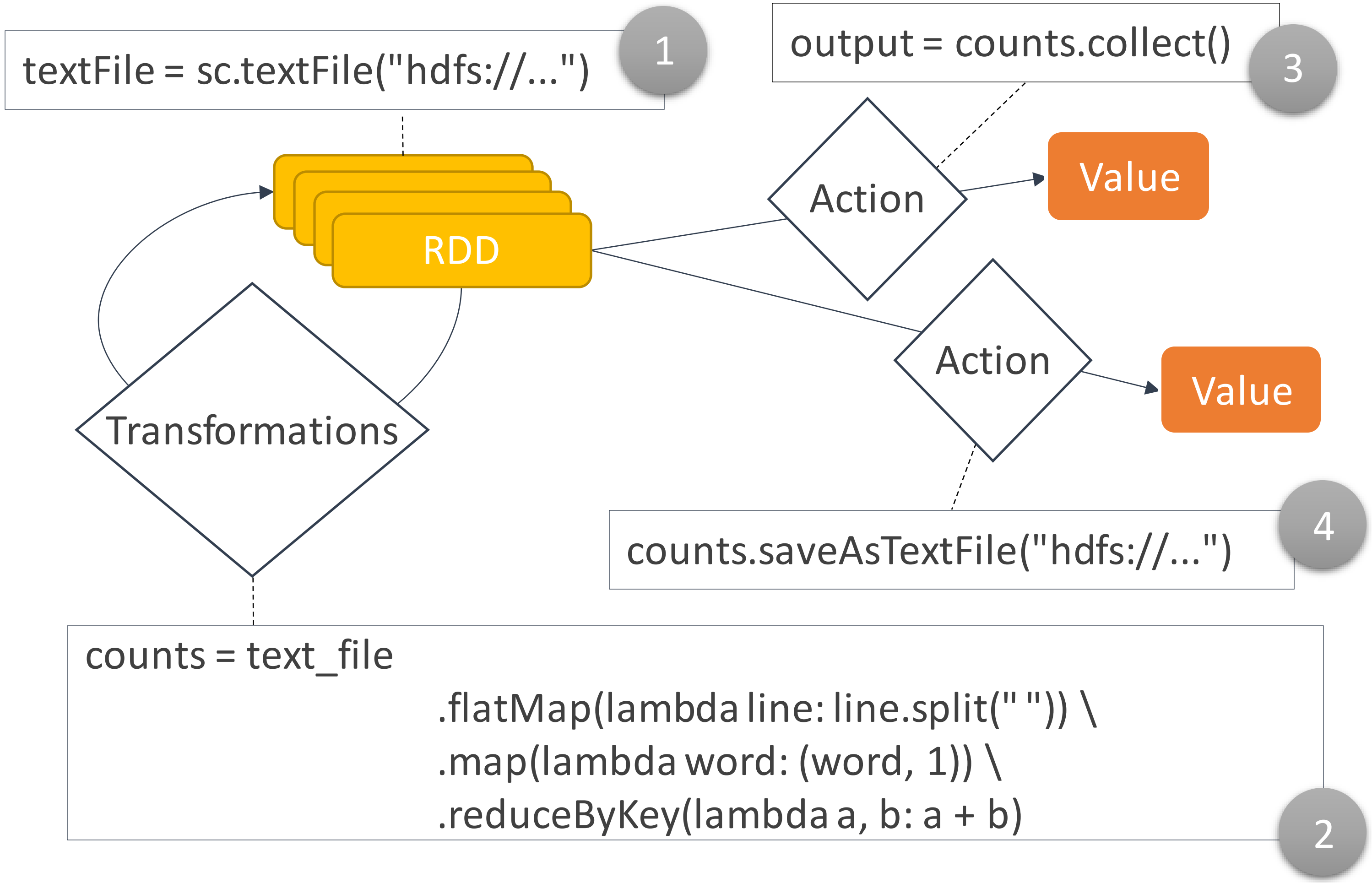
Word count in Scala

```
textFile = sc.textFile("hdfs://...")
counts = textFile
    .flatMap(line => line.split(" "))
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a + b)
```

counts.saveAsTextFile("hdfs://...")



Conceptual representation



DataFrame and DataSet

RDDs are immutable distributed collection of objects

DataFrames and DataSets are immutable distributed collection of records organized into named columns (i.e., a table)

- **Simply put, RDDs with a schema attached**
- Support both relational and procedural processing (e.g., SQL, Scala)
- Support complex data types (struct, array, etc.) and user defined types
- Cached using columnar storage

Can be built from many different sources

- DBMSs, files, other tools (e.g., Hive), RDDs

Type conformity is checked

- At *compile time* for DataSets; at *runtime* for DataFrames

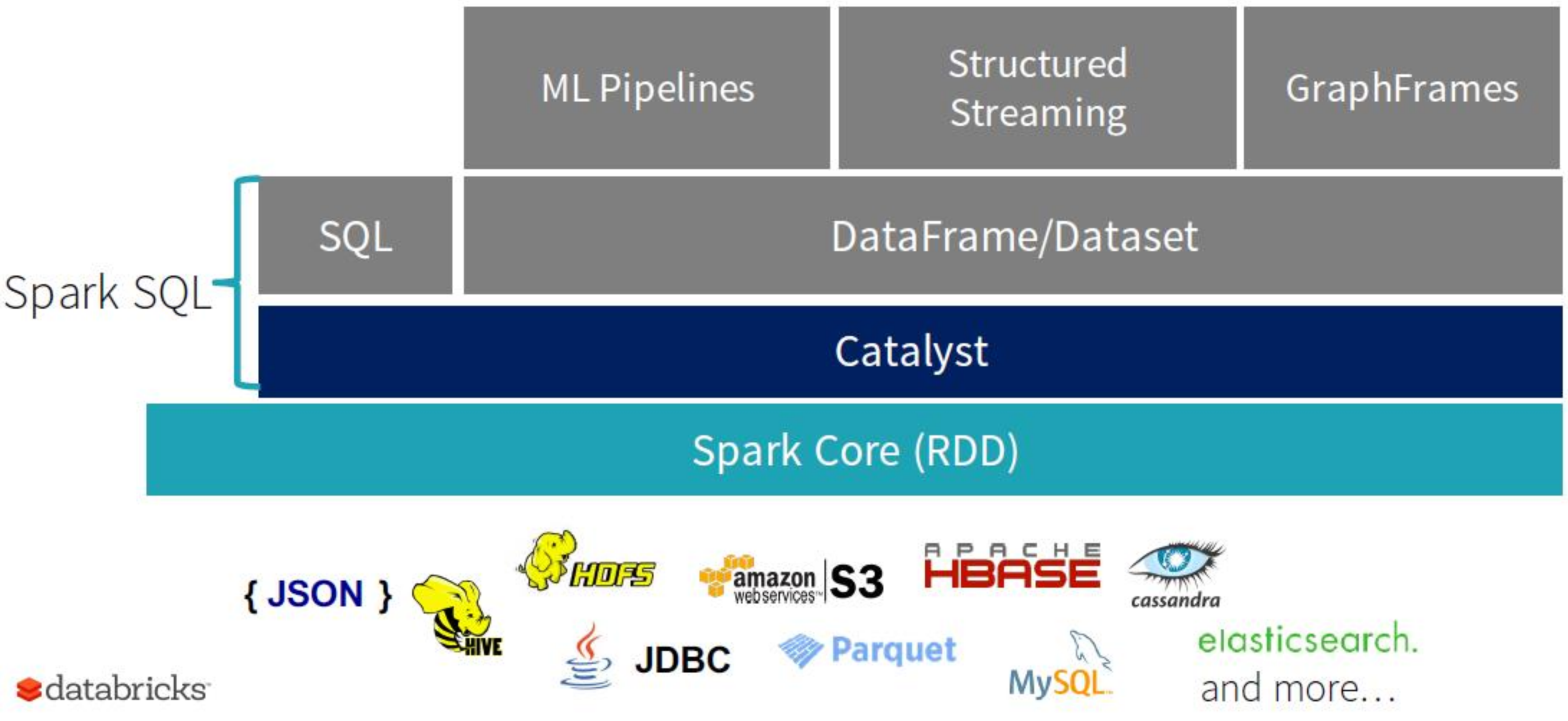
DataFrame and DataSet

Still lazily evaluated…

…but supports under-the-hood optimizations and code generation

- **Catalyst optimizer creates optimized execution plans**
  - IO optimizations such as skipping blocks in parquet files
  - Logic push-down of selection predicates
- JVM code generation for all supported languages
  - Even non-native JVM languages; e.g., Python

Spark structured



Why structure?

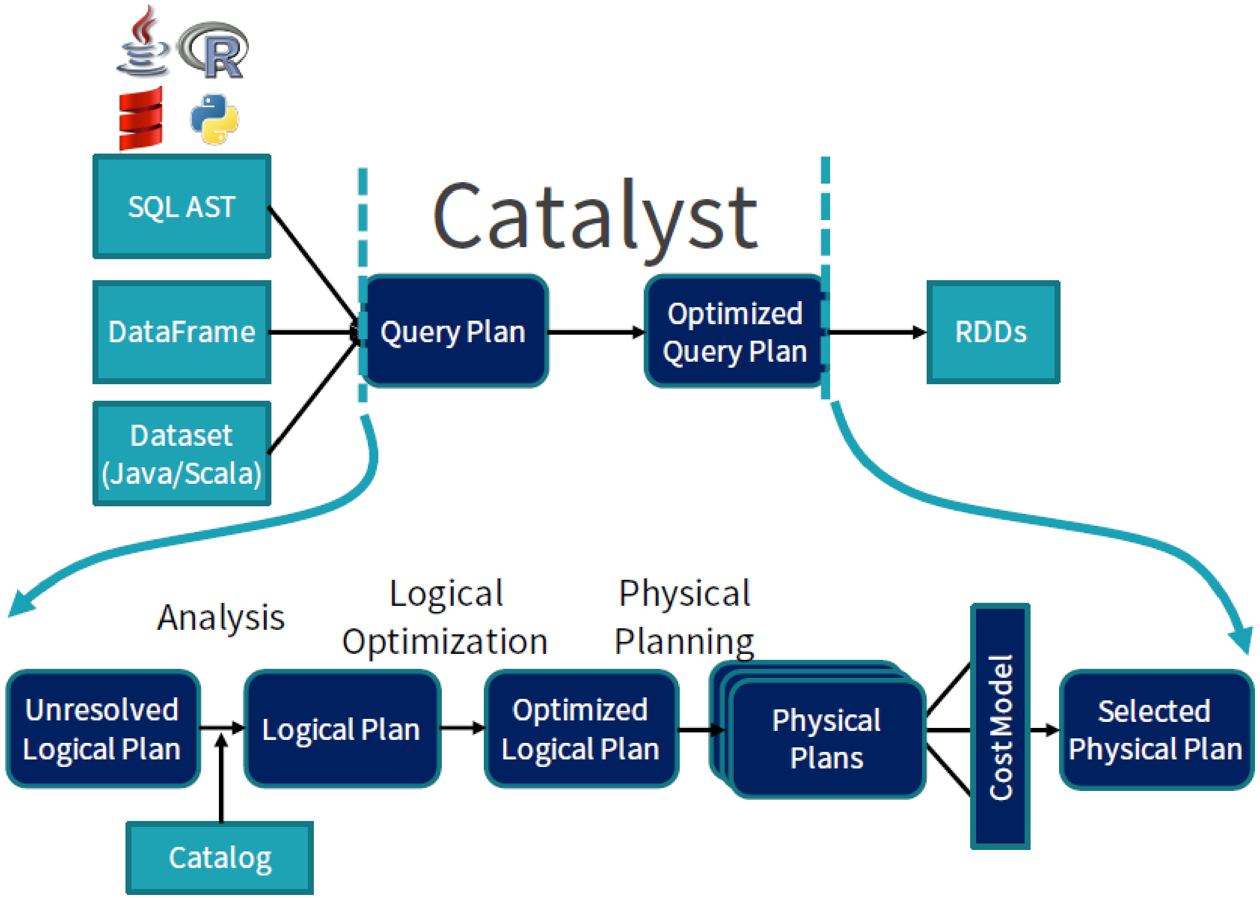
Cons

- **Structure imposes some limits**
  - RDDs enable any computation through user defined functions

Pros

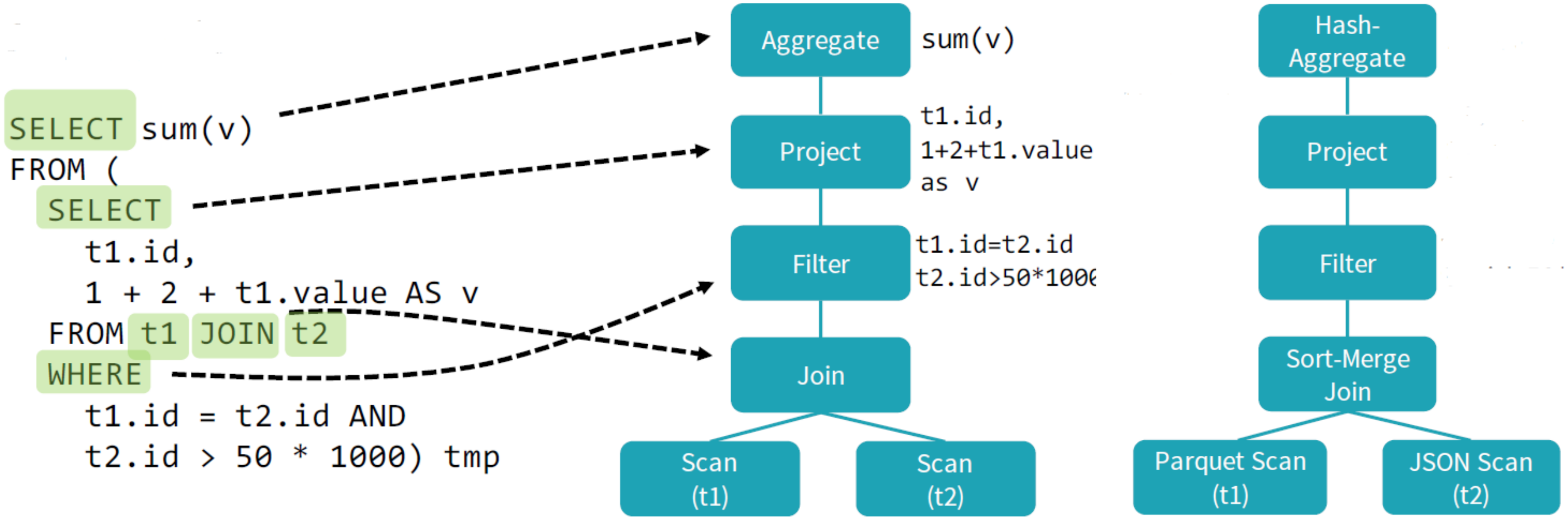
- The most common computations are supported
- Language simplicity
- **Opens the room to optimizations**
  - Hard to optimize a user defined function

Catalyst



SQL AST
DataFrame
Dataset (Java/Scala)
Catalyst
Query Plan
Optimized Query Plan
RDDs
Analysis
Logical Optimization
Physical Planning
Unresolved Logical Plan
Logical Plan
Optimized Logical Plan
Physical Plans
Cost Model
Selected Physical Plan
Catalog

Logical and Physical Plan



Logical Plan

Describes **what** computation must be done

Physical Plan

Describes **what** computation must be done and **how** to conduct it (i.e., which algorithms are used)

Logical optimization

Based on rules

- **A rule is a function** that can be applied on a portion of the logical plan

Implemented as Scala functions

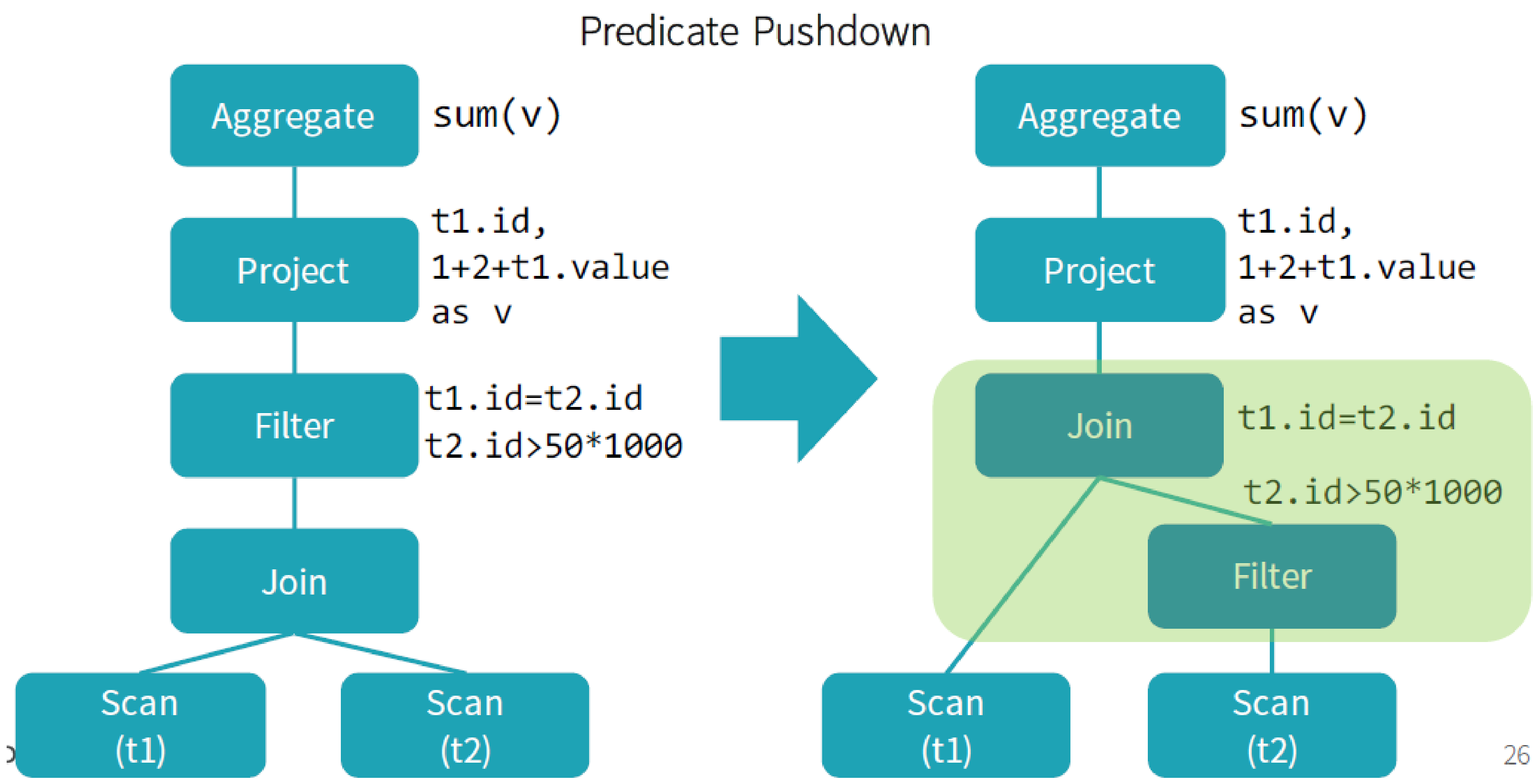
```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

Several types of rules

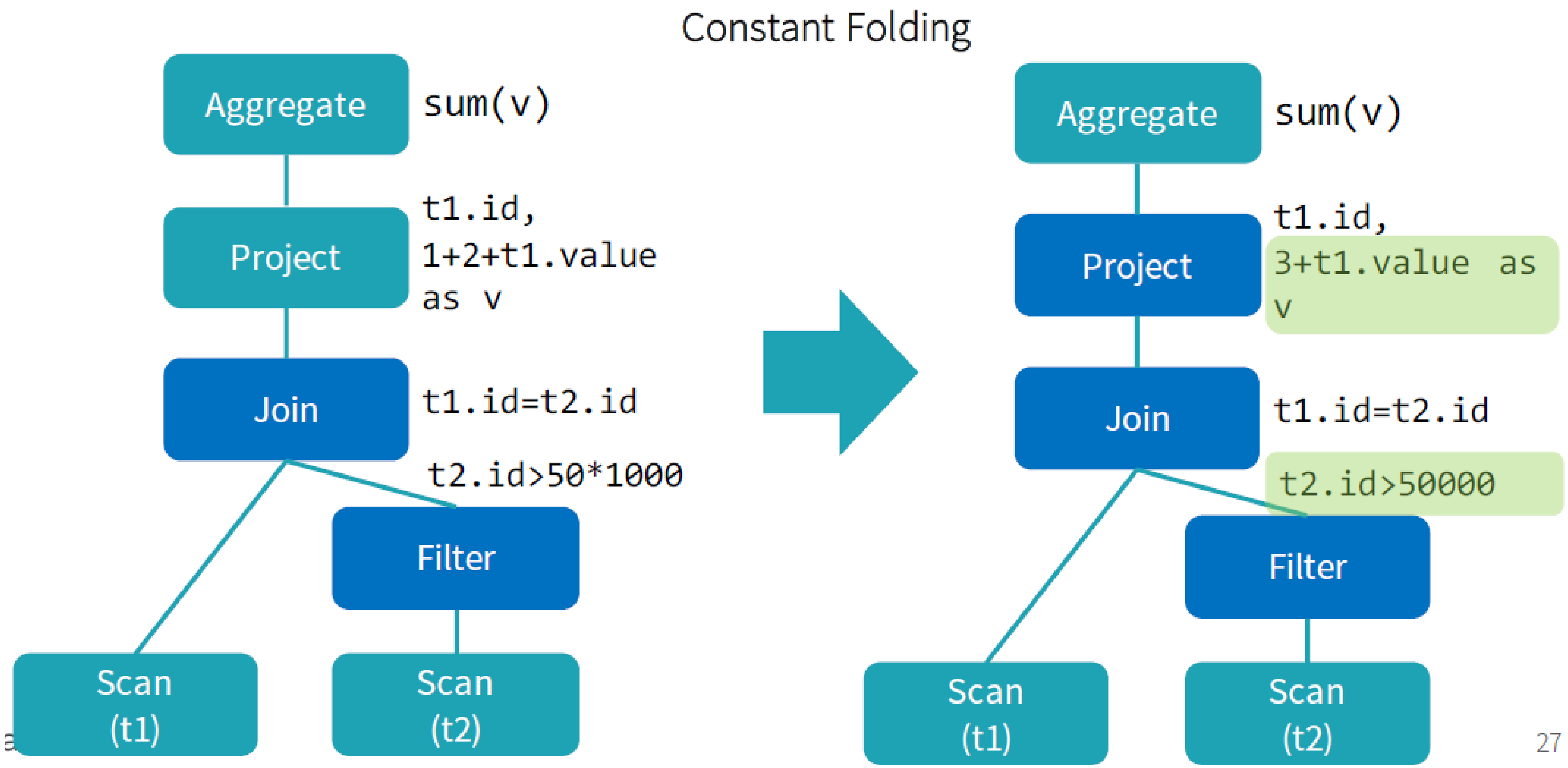
- **Constant folding**: resolve constant expressions at compile time
- **Predicate pushdown**: push selection predicates close to the sources
- **Column pruning**: project only the required column
- **Join reordering**: change the order of join operations

Applied recursively and iteratively until the plan reaches a *fixed point*

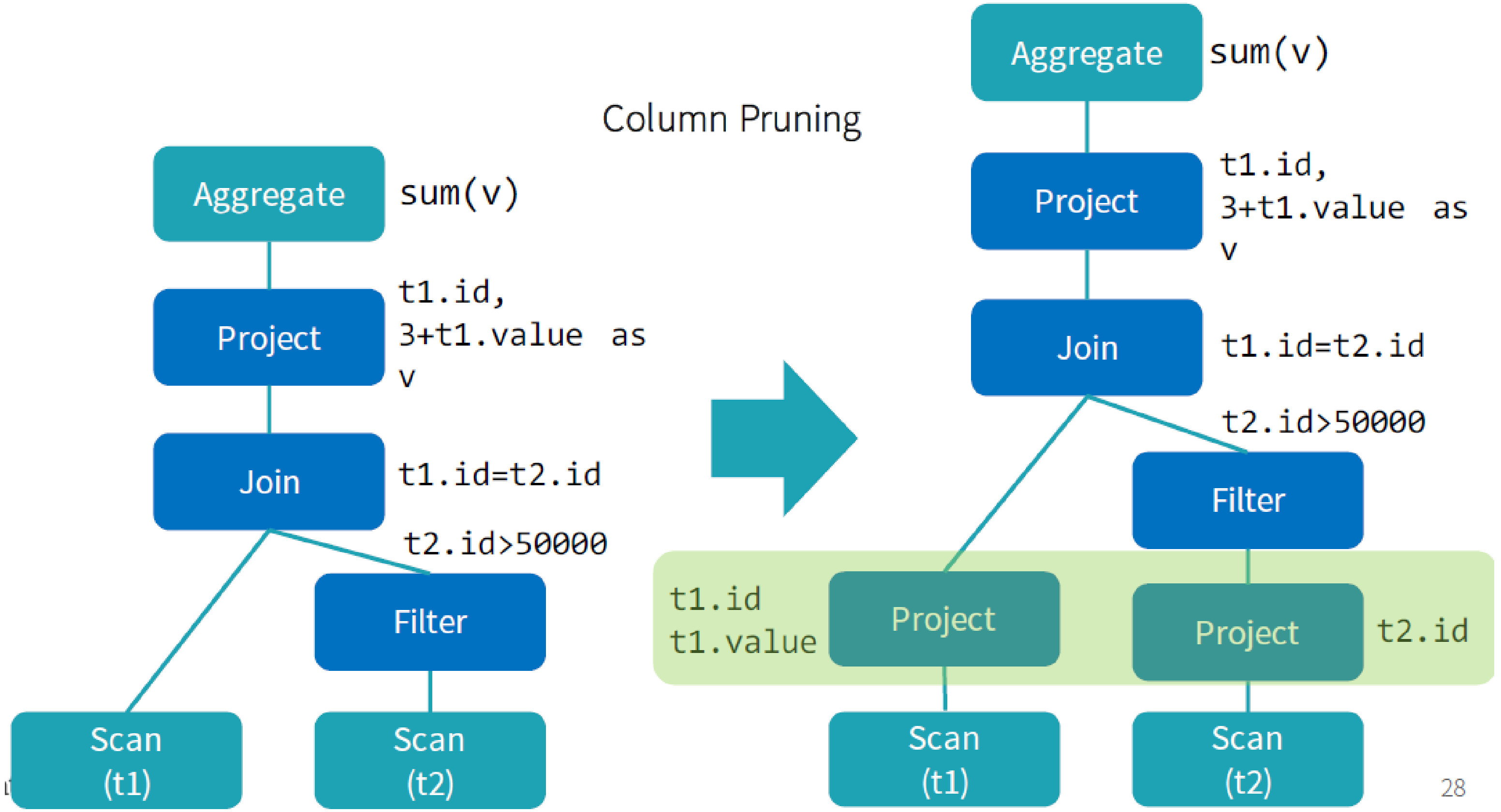
Logical optimization



Logical optimization



Logical optimization



What determines the cost?

Catalyst only considers the size and the cardinality of tables

Other important factors

- Network throughput
- Disk throughput
- Allocation of resources
  - Number of Executors
  - Number of Cores per Executor
- Allocation of tasks
  - Data locality probability

Research work: defining a more accurate, probabilistic cost model

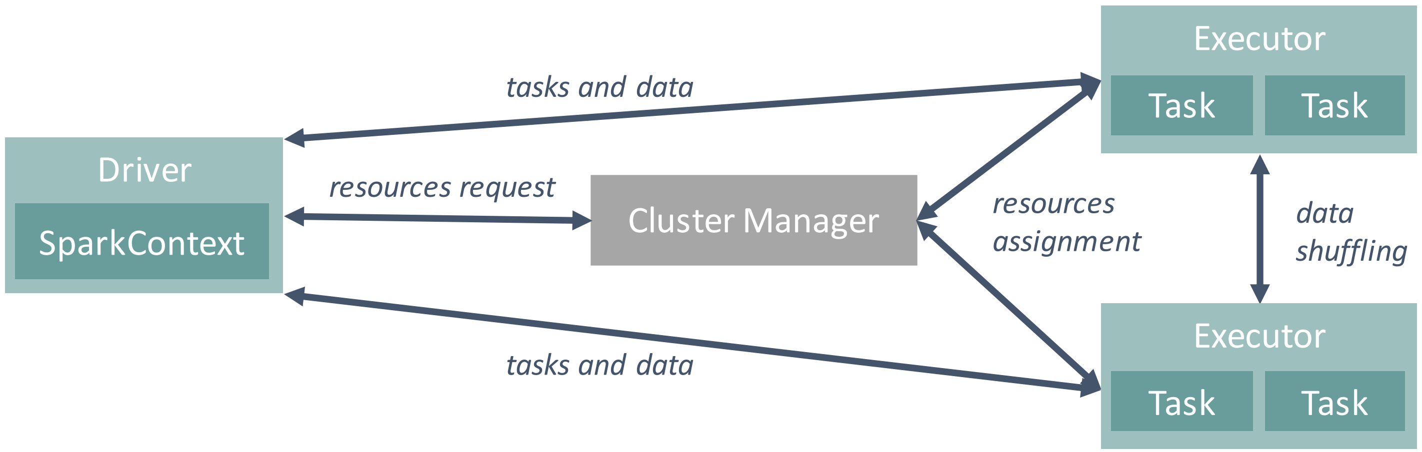
- Baldacci, L., & Golfarelli, M. (2018). A cost model for Spark SQL. IEEE Transactions on Knowledge and Data Engineering, 31(5), 819-832.
- Gallinucci, E., & Golfarelli, M. (2019). SparkTune: tuning Spark SQL through query cost modeling. EDBT 2019: 546-549.

Spark architecture

Spark uses a *master/slave architecture* with one central coordinator (*driver*) and many distributed workers (*executors*)

- The driver and each executor are independent Java processes
- Together they form a Spark *application*

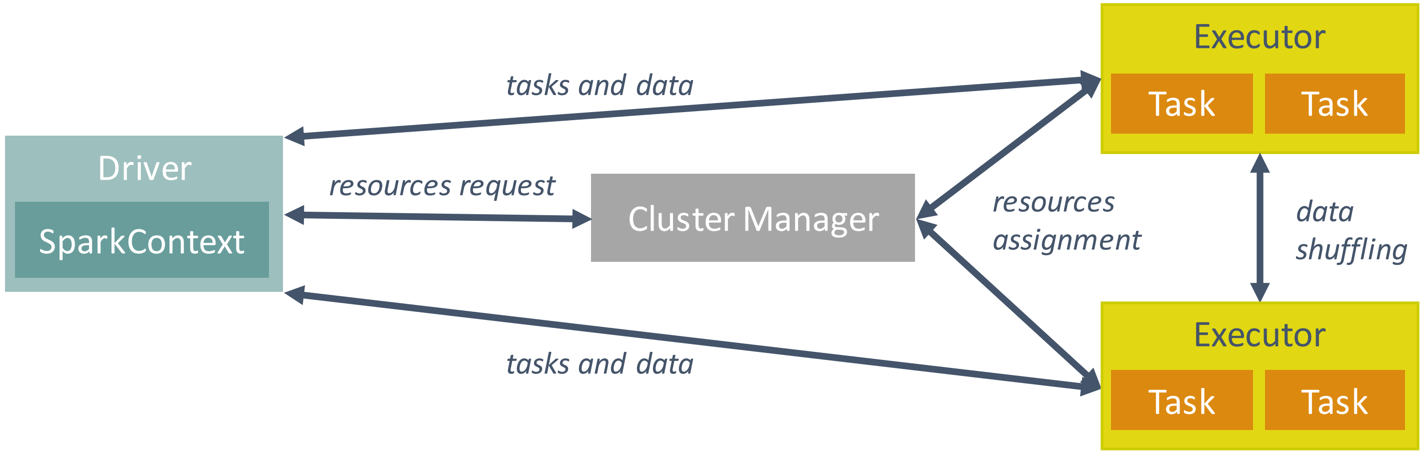
The architecture is independent of the cluster manager that Spark runs on



Spark architecture

Executor: a process responsible for executing the received tasks

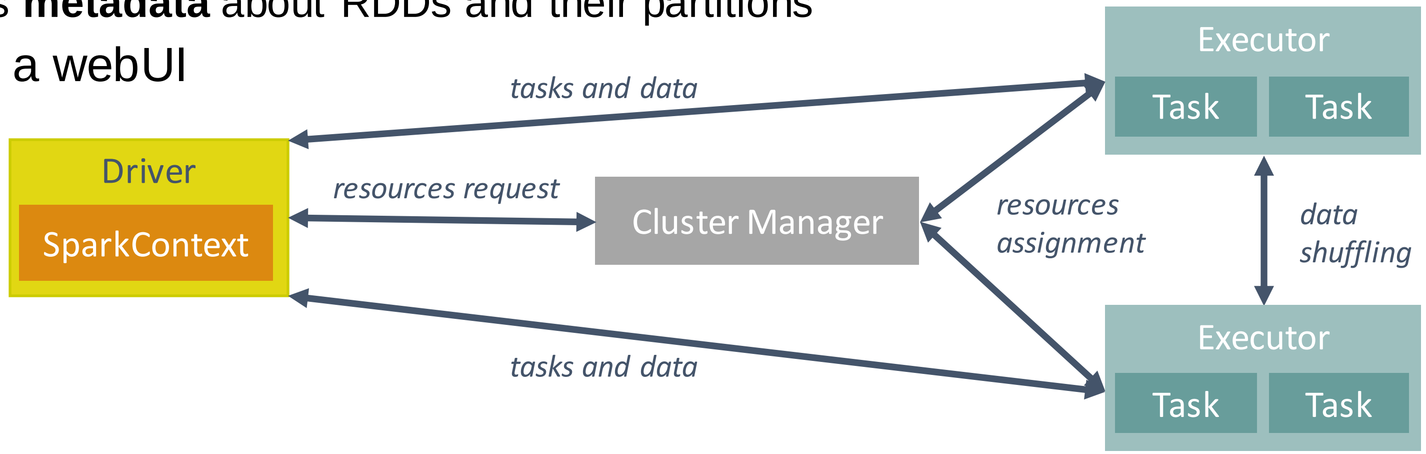
- Each spark application can have (and usually has) multiple executors, and each worker node can host many executors
- Typically runs for the entire duration of the application
- Stores (caches) RDD data in JVM heap
- Tasks are the smallest unit of work and are carried out by executors



Spark architecture

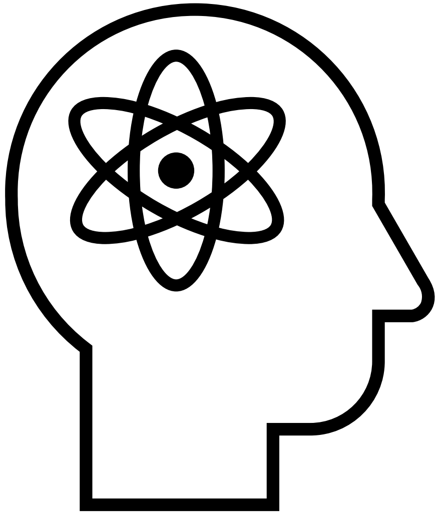
Driver Program (a.k.a. *Spark Driver*, or simply *Driver)*

- Each spark application can only have one driver (entry point of Spark Shell)
- Converts user program into tasks
  - Creates the **SparkContext**, i.e., the object that handles communications
  - Computes the logical **DAG** of operations and converts it into a physical **execution plan**
- Schedules tasks on executors
  - Has a **complete view** of the available executors and schedules tasks on them
  - Stores **metadata** about RDDs and their partitions
- Launches a webUI



In action!

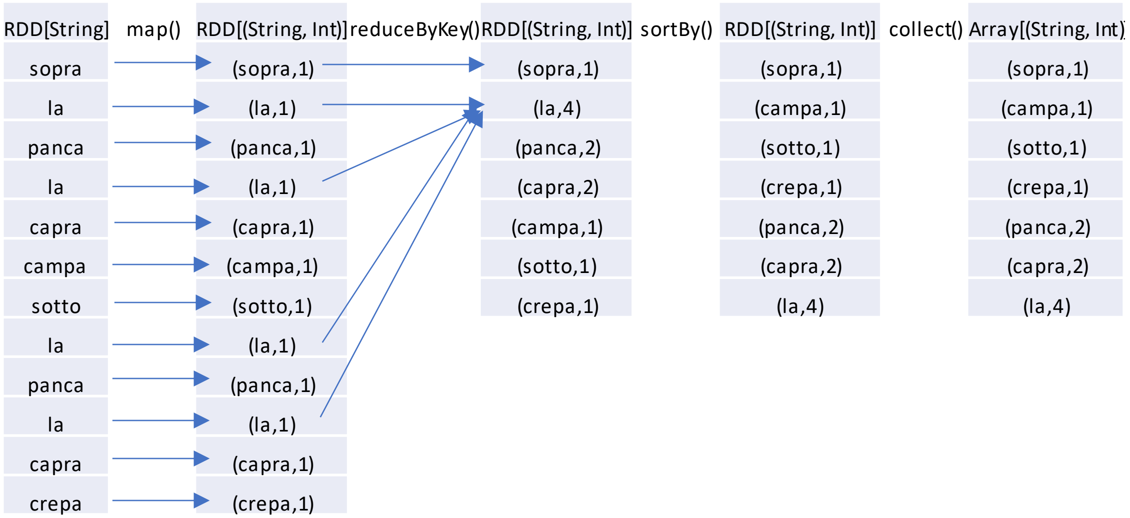
Enter the notebook `03-BigData`



https://github.com/w4bo/2022-bbs-dsaa/blob/master/materials/01-DataPreprocessing.ipynb

Word count

```
rdd
    .map(s => (s, 1))
    .reduceByKey((a,  b) => a + b)
    .sortBy(x => x._2) tuples
    .collect()
```



Foodmart

```
foodmartDataset
        .groupBy("category")
        .sum("unit sales")
        .orderBy("sum(unit sales)")
        .show()
```

foodmartDataset		
product	category	unit_sales
beer	drink	2
cola	drink	3
bread	food	4
pizza	food	5

groupBy(category)

foodmartDataset		
product	category	unit_sales
bread	**food**	4
pizza	**food**	5
beer	**drink**	2
cola	**drink**	3

sum(unit_sales)

foodmartDataset	
category	sum(unit_sales)
food	9
drink	5

orderBy(sum(unit_sales))

foodmartDataset	
category	sum(unit_sales)
drink	5
food	9

Spark

Suggested reading and resources