Cloud Computing

Chapter 4: Data-Intensive Applications on Cloud Architectures

Summer Term 2017

Complex and Distributed IT Systems
TU Berlin

Overview

- Motivation
- MapReduce
- Beyond MapReduce: Flink and Spark

Motivation

- Amount of digitally available data grows rapidly
  - WWW
    - User-generated content only is 10GB per day[1]
  - Scientific data
    - LHC will produce 60 TB per day[2]
  - Data warehousing
    - HP currently builds 4PB data store for Wal-Mart[2]
- Traditionally, domain of Internet search companies, but
  - „Big Data“ analysis is subject of vivid research
  - More and more new companies surface

More and more data is available to science and business!

video streams

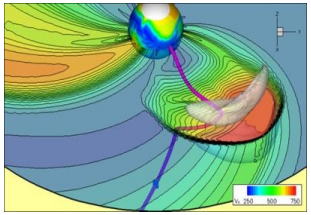
sensor data

web archives

Drivers:
Cloud Computing
Internet of Services
Internet of Things
Cyberphysical Systems

audio streams

RFID data



simulation data

Underlying Trends:
Connectivity
Collaboration
Computer generated data

Data and analyses are becoming increasingly complex!

volume (data size)
velocity (freshness, data rate, streams)
variability (format/media type)
veracity (uncertainty/quality)

Data

interactive (visual analytics, ad-hoc)
integrative (extraction, fusion)
iterative (learning, models)
incremental (mutable state, windows)

Analysis

Data-driven applications ...

lifecycle management

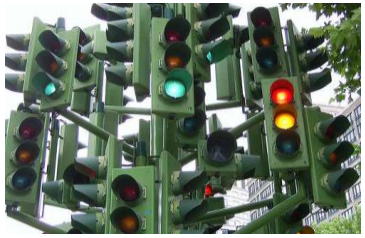
home automation

health

water management

market research

information marketplaces

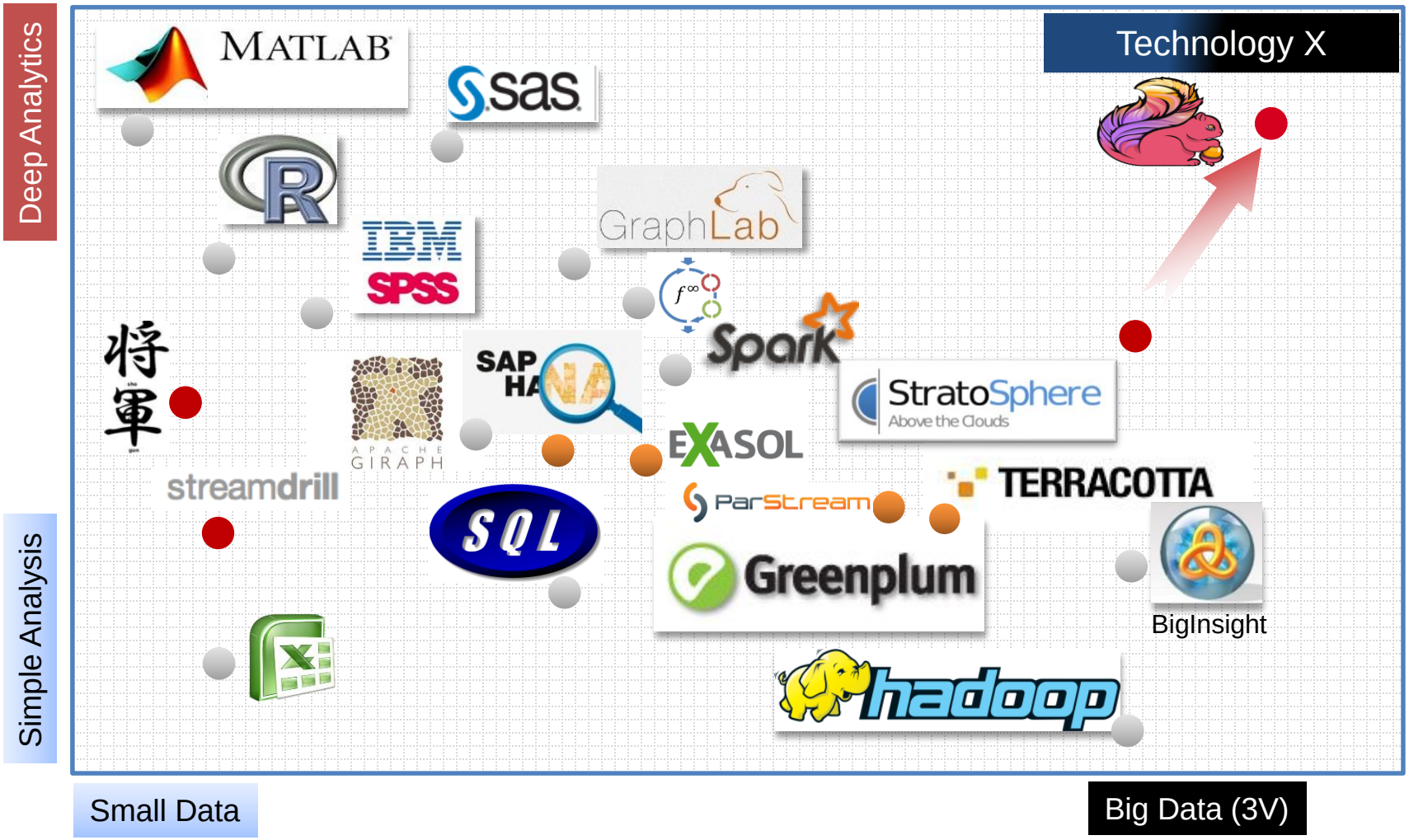


traffic management

energy management

- … will revolutionize decision making in business and the sciences!
- … have great economic potential!

Existing Systems Analyze Small Data or Only Allow for Simple Analysis!



Relation Data-Intensive Applications – Clouds (1/2)

- Requirements for data-intensive applications out-scaled several traditional solutions
  - Parallel databases are prohibitively expensive[4]
- Instead, large sets of commodity clusters are preferred
  - Data is stored on local hard disks
  - Allows to keep computation close to the data
- Individual commodity servers are less reliable
  - Reliability must be achieved on software level

→Data-intensive applications are typically elastic

- ♦ Can cope with changing number of nodes (Further details on this later)

Relation Data-Intensive Applications – Clouds (2/2)

- Cloud computing (in particular IaaS) often considered promising platform for data-intensive applications
  - Looks like a large pool of clusters to customer
  - Elastic platform meets elastic application
  - No large upfront capital expenses (good for customers with infrequent need for large-scale data analysis)
- However, there are also downsides
  - Virtualization overhead for I/O operations
  - No control over physical infrastructure

→ Interesting platform for start-ups, infrequent users

  - But data analysis companies buy their own clusters

Challenges of Large-Scale Data Processing

- Large clusters/clouds have 100s/1000s of servers
  - Extremely high performance/throughput possible
  - Problem: Highly parallel environment
    - Applications must be written to take advantage of that
- Writing efficient parallel applications at this scale is hard
  - Most developers are no experts in this domain
  - Don’t want to deal with concurrency issues, fault tolerance
  - They just care about extracting information from the data
- Needed: Suitable abstraction layer for developers

Abstraction Layer for Data-Intensive Applications

- Requirements for such an abstraction layer
  1. Developers don‘t have to think about parallelization
     - Can continue to write sequential code
     - Code is independent of degree of parallelism at runtime
  2. Developers don‘t have to think about fault tolerance
     - Abstraction layer takes care of failed nodes
     - Re-executes lost parts of computations if necessary
  3. Developers don‘t have to think about load balancing
     - Abstraction layer is in charge of distributing the work evenly across the available compute nodes

Overview

- Motivation
- **MapReduce**
- Beyond MapReduce: Flink and Spark

MapReduce

- System published by Google in 2004[4]
  - Introduces MapReduce programming model
  - Illustrates how model helps to meet previous requirements (sequential code, fault tolerance, …)

- Google uses MapReduce for various things
  - Process crawled documents, logs
  - Computing inverted indices
  - …
- Publication has spawned various research activities
  - Paper is among most cited research paper in recent years

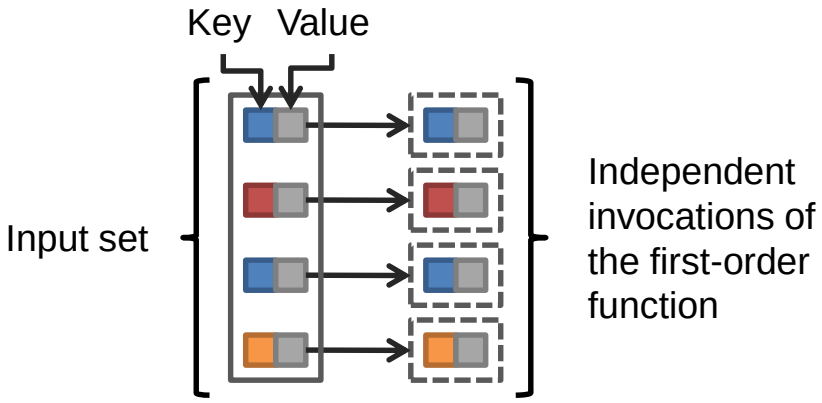
The MapReduce Programming Model

- Based on second-order functions *map* and *reduce*
  - Inspired by functional programming language Lisp
- Map and reduce take first-order functions as input
  - Specify signature of the first-order function
  - Specify how data is passed to first-order function
- MapReduce operates on a key-value (KV) model
  - Data is passed as KV pairs through the system
  - Developers can specify how to build KV pairs from input data

The Map Function

- Signature: $(k_1, v_1) \rightarrow list(k_2, v_2)$

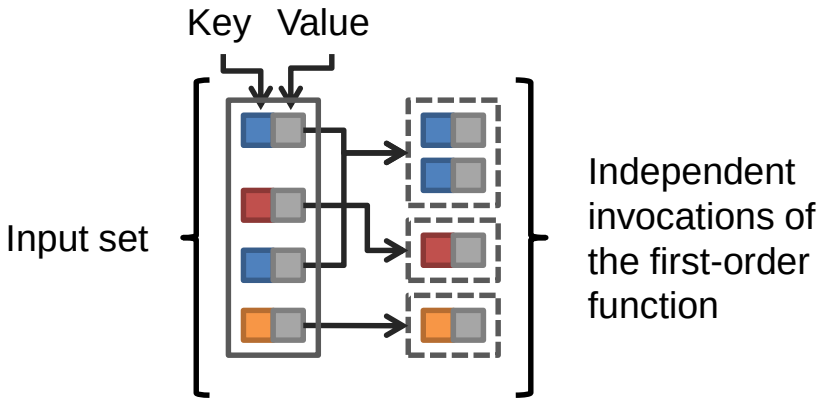
- Guarantees to the first-order function
  - First-order function is invoked once for each KV pair
  - Can produce a [0,*] KV pairs as output
- Useful for projections, selection, …



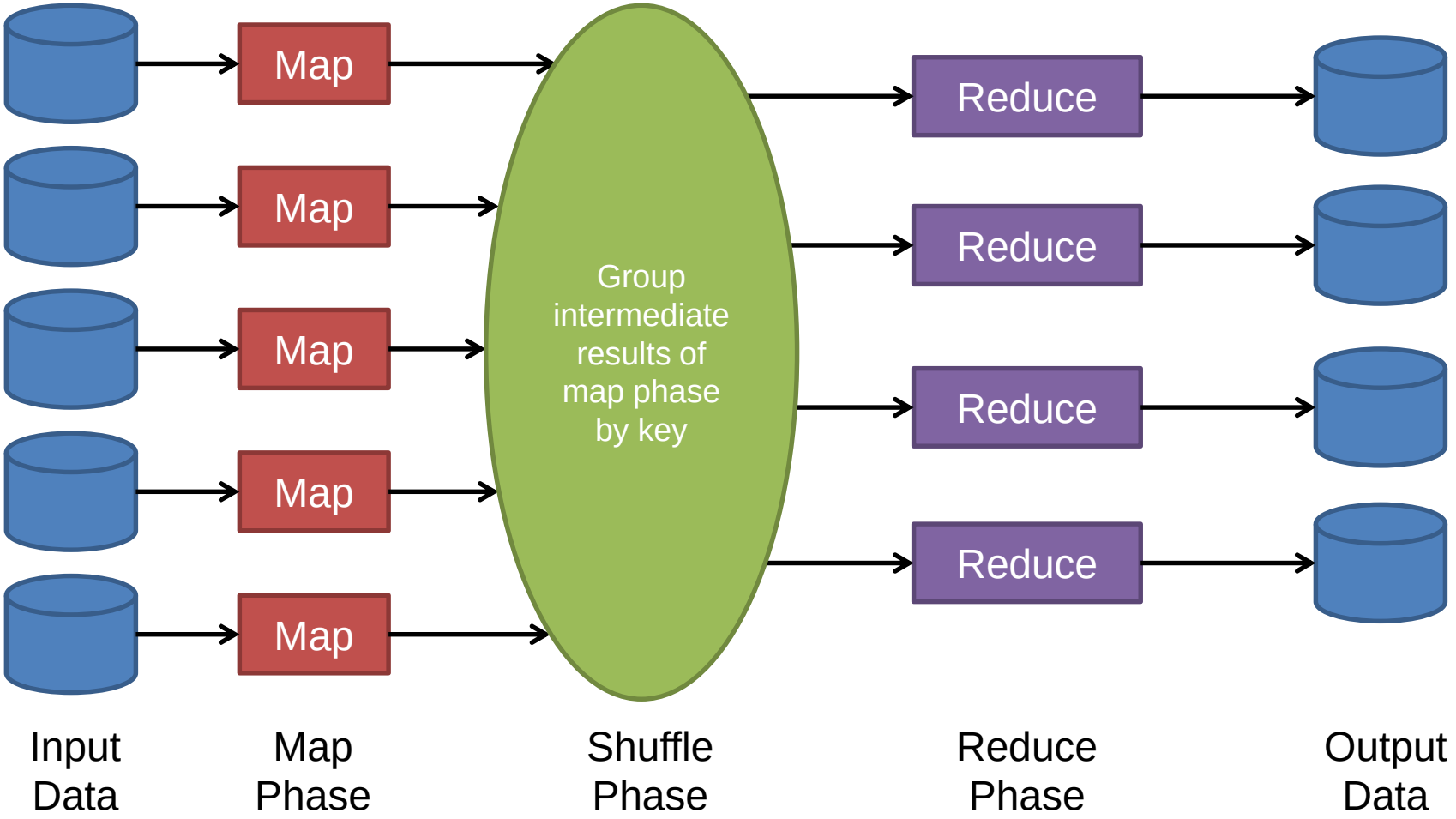
The Reduce Function

- Signature: $(k_2, list(v_2)) \rightarrow list(k_3, v_3)$

- Guarantees to the first-order function
  - All KV-pairs with the same key are presented to the same invocation of the first-order function
- Useful for aggregation, grouping, …



High Level View on a MapReduce Program



MapReduce Example: Word Count (1/4)

- Task: Count the occurrences of words in a text
- Example text:

```
1: One For The Money
2: Two For The Show
3: Three To Get Ready
4: Now Go Cat Go
```

- KV pairs for the map function: <Line number, line>
  - <1, One For The Money>, <2, Two For The Show>, …

→Four KV pairs

→Four invocations of the map function

MapReduce Example: Word Count (2/4)

- Code of Word Count map function:

```
Map(long lineNumber, string line) {
    for each word w in line {
        EmitIntermediate(w, 1);
    }
}
```

- Intermediate results produced by map functions
  1. <One, 1>, <For, 1>, <The, 1>, <Money, 1>
  2. <Two, 1>, <For, 1>, <The, 1>, <Show, 1>
  3. <Three, 1>, <To, 1>, <Get, 1>, <Ready, 1>
  4. <Now, 1>, <Go, 1>, <Cat, 1>, <Go, 1>

MapReduce Example: Word Count (3/4)

- In the shuffle phase the intermediate results from the map invocations are grouped by key

→Input for the reduce phase

1. <Cat, (1)>
2. <For, (1, 1)>
3. <Get, (1)>
4. <Go, (1, 1)>
5. <Money, (1)>
6. <Now, (1)>
7. <One, (1)>
8. <Ready, (1)>
9. <Show, (1)>
10. <The, (1, 1)>
11. <Three, (1)>
12. <To, (1)>
13. <Two, (1)>

MapReduce Example: Word Count (4/4)

- Code of Word Count reduce function:

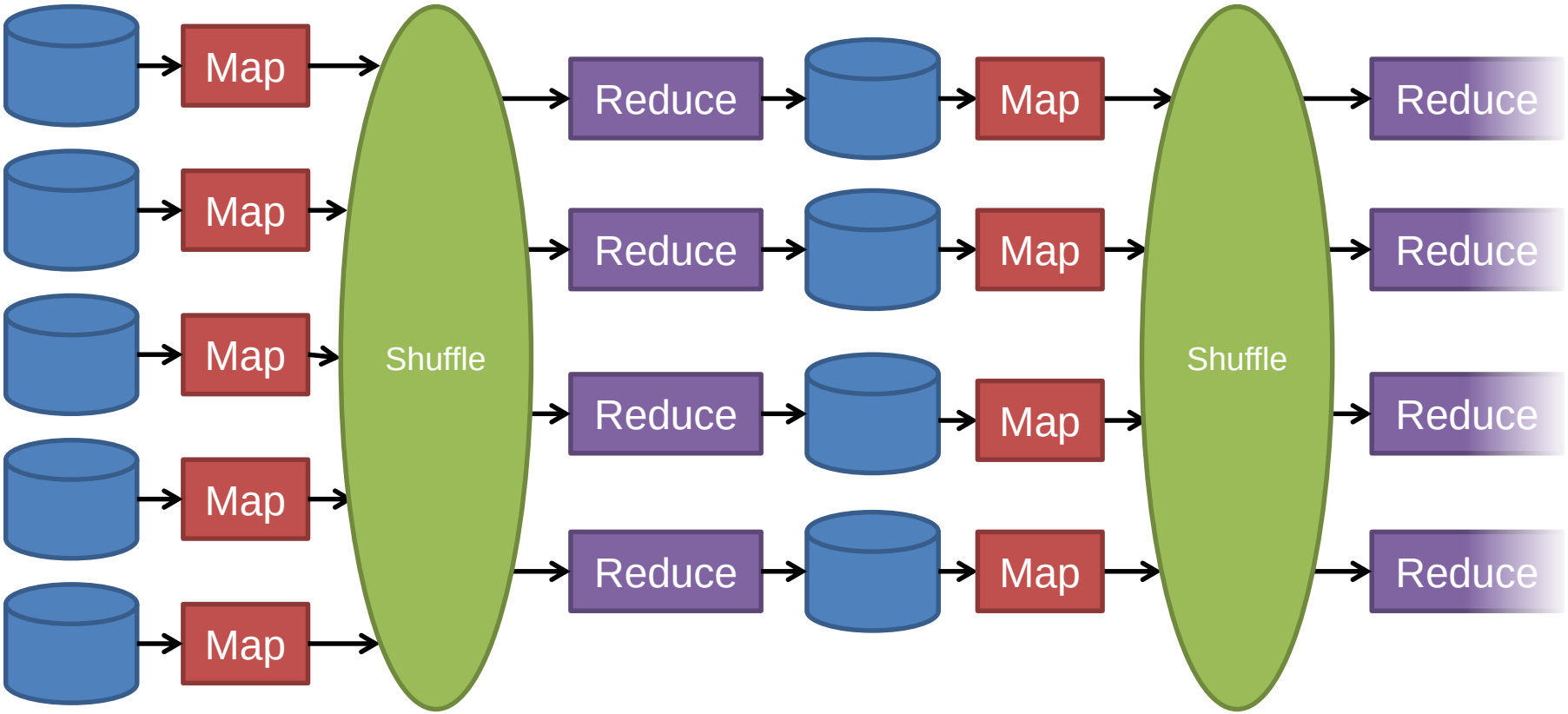
```
Reduce(String word, Iterator partialCounts) {
    int result = 0;
    for each v in partialCounts {
        result +=v;
    }
    Emit(word, result);
}
```

- Final results after the MapReduce job in output file

1. <Cat, 1>
2. <For, 2>
3. <Get, 1>
4. <Go, 2>
5. <Money, 1>
6. <Now, 1>
7. <One, 1>
8. <Ready, 1>
9. <Show, 1>
10. <The, 2>
11. <Three, 1>
12. <To, 1>
13. <Two, 1>

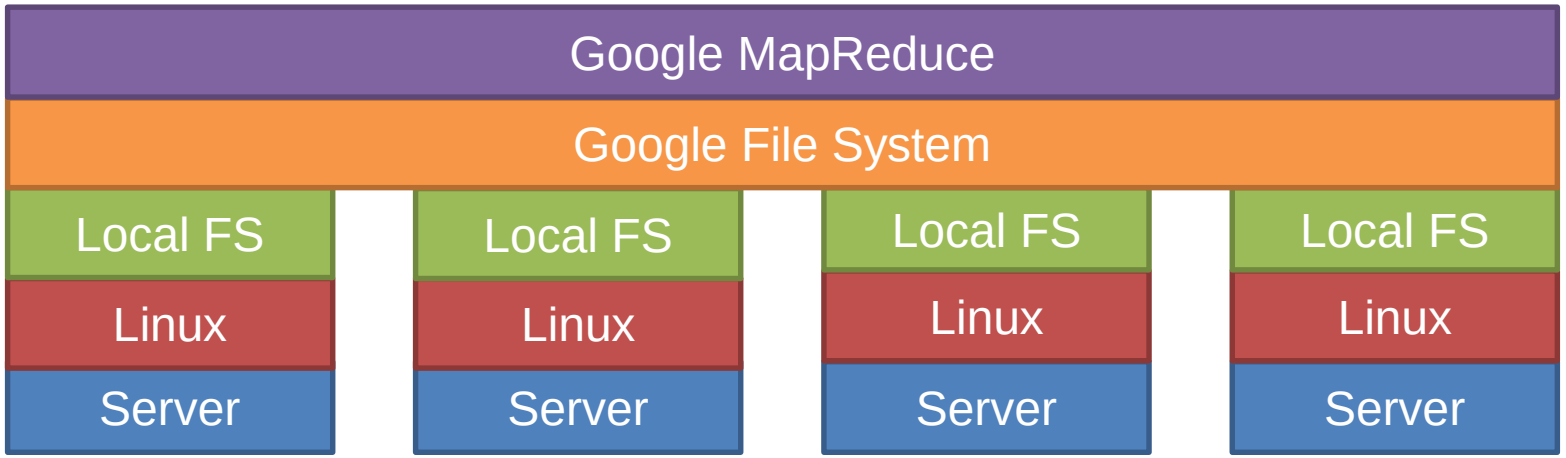
Combining Multiple MapRedue Programs

- More complex programs can be created by combining individual MapReduce jobs



MapReduce Implementation (1/2)

- Designed to run on large set of shared nothing servers
- Further assumptions
  - Expects distributed file system
    - Every node can potentially read every part of input
  - Individual nodes are likely to fail
  - Prefers local storage (computation is pushed to data)



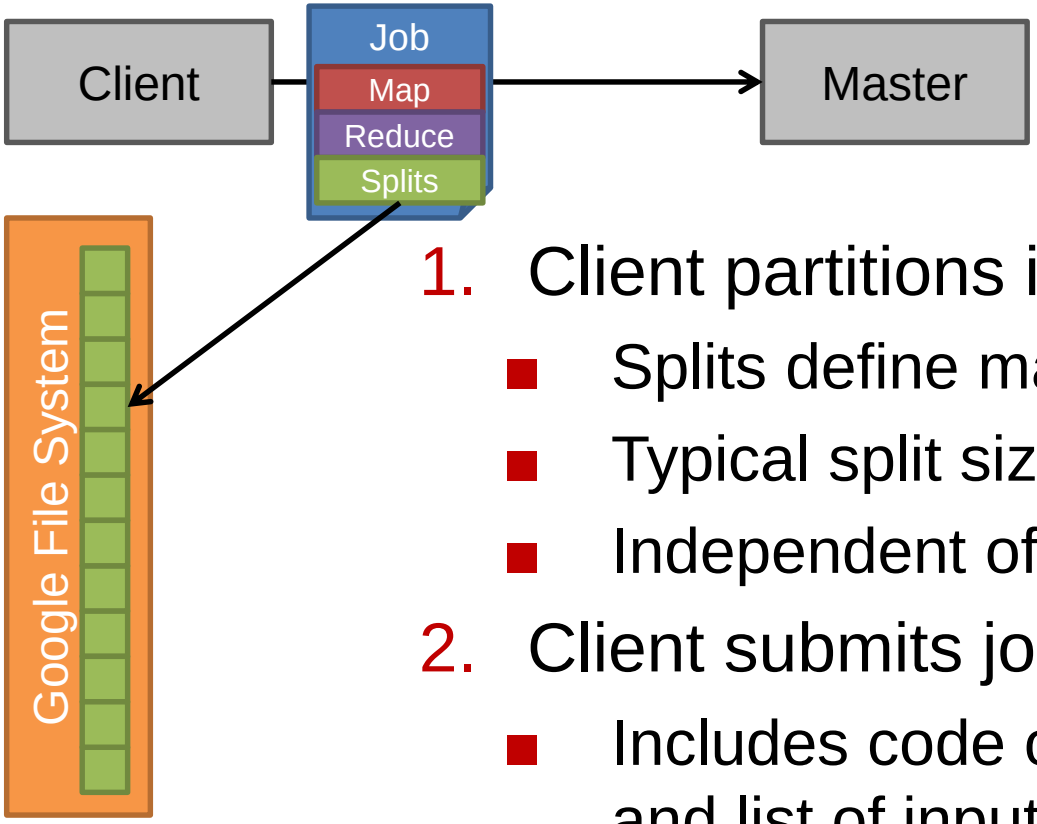
MapReduce Implementation (2/2)

- MapReduce follows master-worker pattern
- Master
  - Responsible for job scheduling
  - Monitoring worker nodes, detecting dead nodes
  - Load balancing
- Workers
  - Executing map and reduce functions
  - Storing input/output data (in traditional setup)
  - Periodically report availability to master node

Some MapReduce Terminology

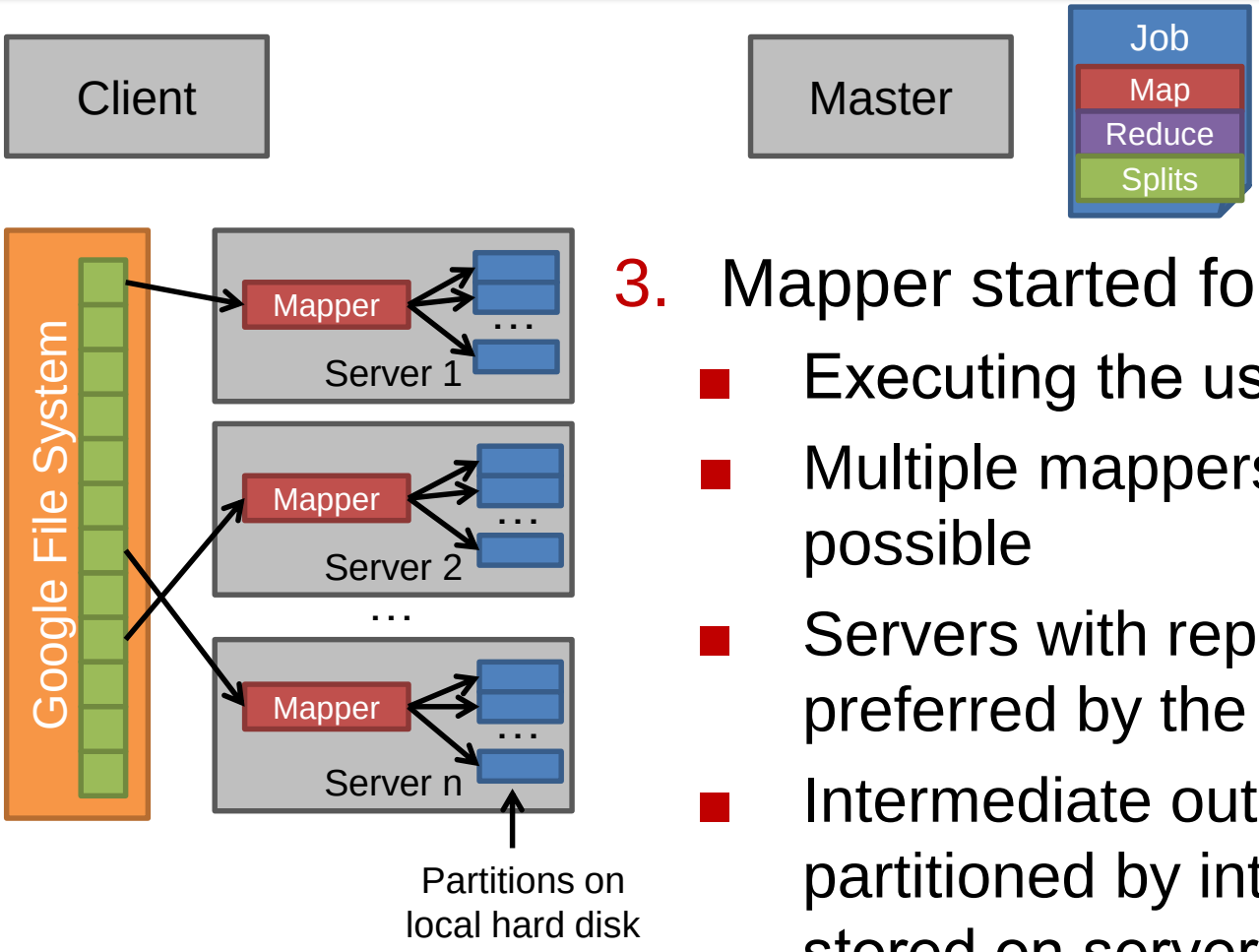
- Map function
  - First-order function provided by user
  - Specifies what happens to the data in job‘s map phase
- Mapper
  - A process running on a worker node
  - Invokes map function for each KV pair
- Reduce function
  - First-order function provided by user
  - Specifies what happens to the data in job‘s reduce phase
- Reducer
  - Process invoking reduce function on grouped data

Distributed Execution of MapReduce Program (1/3)



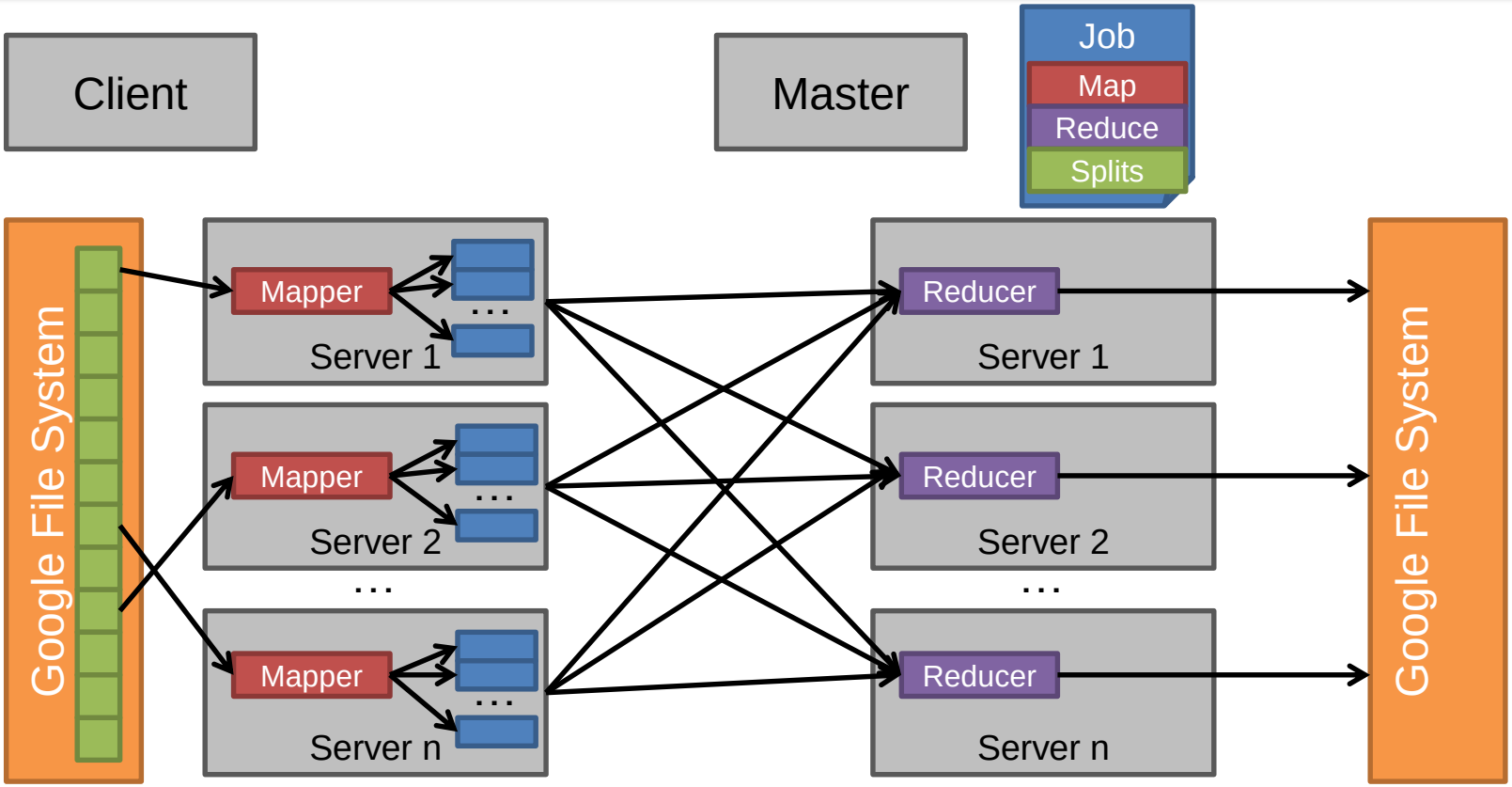
1. Client partitions input file into input splits
   - Splits define maximal scale-out
   - Typical split size is 16-64 MB
   - Independent of GFS block size
2. Client submits job to master
   - Includes code of map/reduce functions and list of input splits (file boundaries)
   - Master tries to find free resources to schedule mappers/reducers

Distributed Execution of MapReduce Program (2/3)



3. Mapper started for each input splits
   - Executing the user's map function
   - Multiple mappers per server possible
   - Servers with replicas of input data preferred by the master
   - Intermediate output of mappers is partitioned by intermediate key, stored on server's local hard disk

Distributed Execution of MapReduce Program (3/3)



3. Reducers pull data from mappers over network
   - Data is grouped by key, passed to user's reduce function

MapReduce Refinement: Combiners

- Can be used to locally aggregate intermediate results
  - Executed after the mappers
  - Useful to unburden network
  - Example: If intermediate KV pair <For, 1>, <For, 1> is created by the same mapper, the combiner aggregates it to <For, 2>, without combiner <For, (1,1)> is shipped
- Applicability of combiner depends on job
  - Sometimes local aggregation destroys correctness of results
    - Example: Reduce function shall compute a median

MapReduce Fault Tolerance

- Scenario 1: Mapper fails
  - Master detects failure through missing status report
  - Mapper is restarted on diff. node, re-reads data from GFS
- Scenario 2: Reducer fails
  - Again, detected through missing status report
  - Reducer is restarted on different node, pulls intermediate results for its partition from mappers again
- Scenario 3: Entire worker node fails
  - Master re-schedules lost mappers and reducers
  - Finished mappers may be restarted to recompute lost intermediate results

Apache Hadoop

- Open source implementation of Google's MapReduce
  - Written in Java
  - Many prominent users
    - Yahoo!, Facebook, Twitter, …

- Provides foundation for many data analytics projects
  - Higher level languages: Pig[5], Hive[6], …
  - Machine learning: Mahout, …

- Different commercial flavors of Hadoop available
  - Cloudera, Hortonworks, …

MapReduce Limitations

- MapReduce is powerful but has two major limitations
  1. Assumes finite input (files only)
  2. Data between MR jobs must go to Google File System

- Constraint to write to GFS especially detrimental for iterative algorithms
  - E.g. graph processing, machine learning, …

- Limitation of finite input prevents streaming processing
  - Useful to respond to events without large delays
  - E.g. click streams, IT health monitoring, …

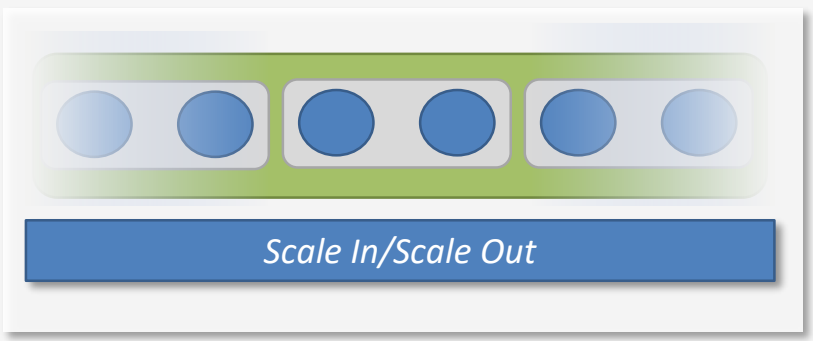
Map/Reduce with Stratosphere

"How to improve the efficiency of massively parallel data processing on Infrastructure as a Service (IaaS) platforms"

- IaaS cloud
  - Commercial platform to rent large pool of compute nodes (virtual machines)
  - Virtual machines are paid by the hour
  - On-demand resource allocation, heterogeneous machine types
- Opportunities: Elasticity
  - Scale-up/scale-down to respond to changes in the workload
  - Exploit resource heterogeneity to improve cost efficiency
- Challenges: Loss of control due to required virtualization
  - Shared infrastructure, loss of knowledge about I/O capacities
  - Network topology between machines is unknown

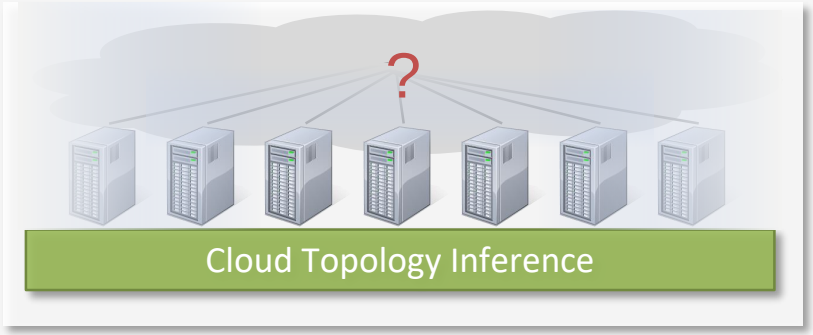
Stratosphere features

Exploiting the cloud's elasticity



Detecting parallelization constraints

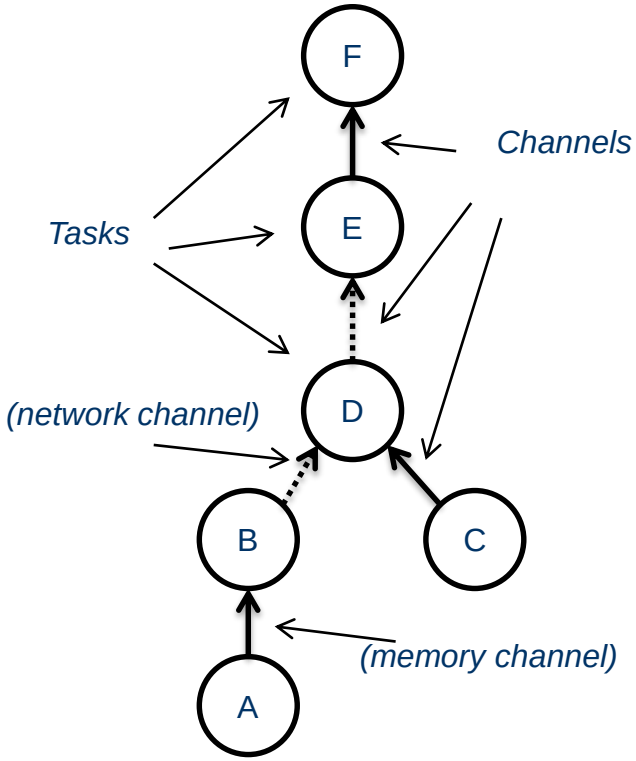
Mitigating I/O bottlenecks



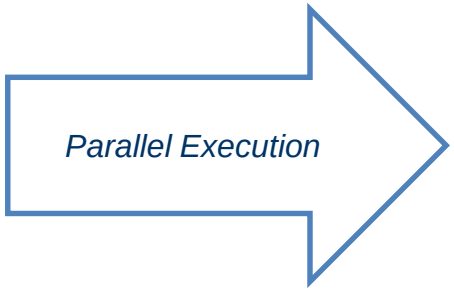
Inferring physical network topologies

Stratosphere Job Graphs

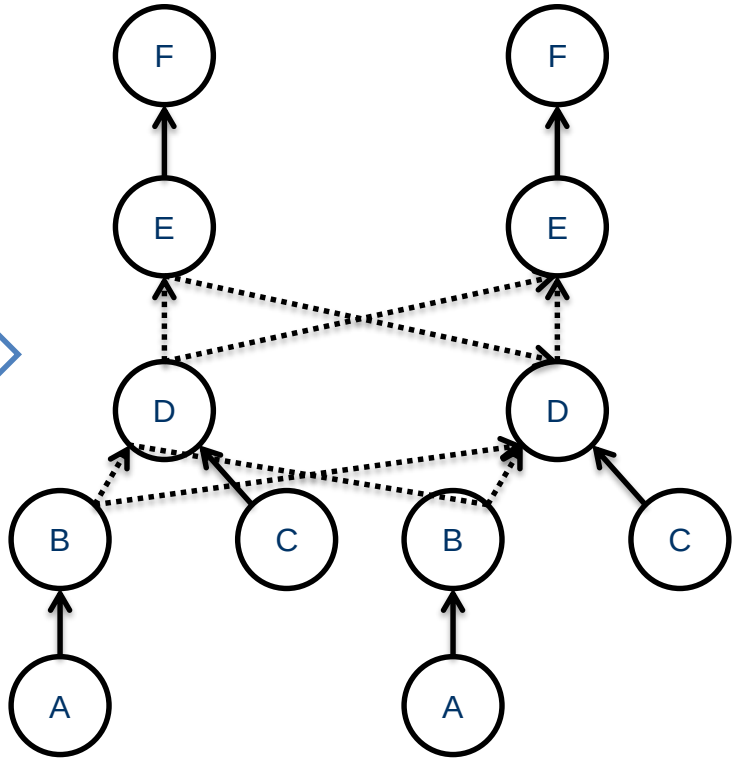
Job Graph



Tasks consume data streams
and produce data streams



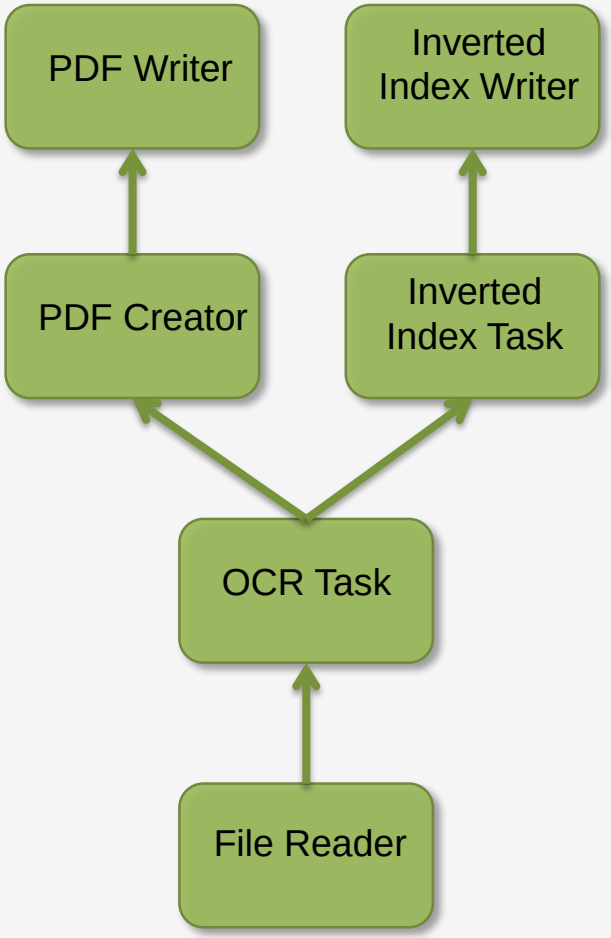
Execution Graph



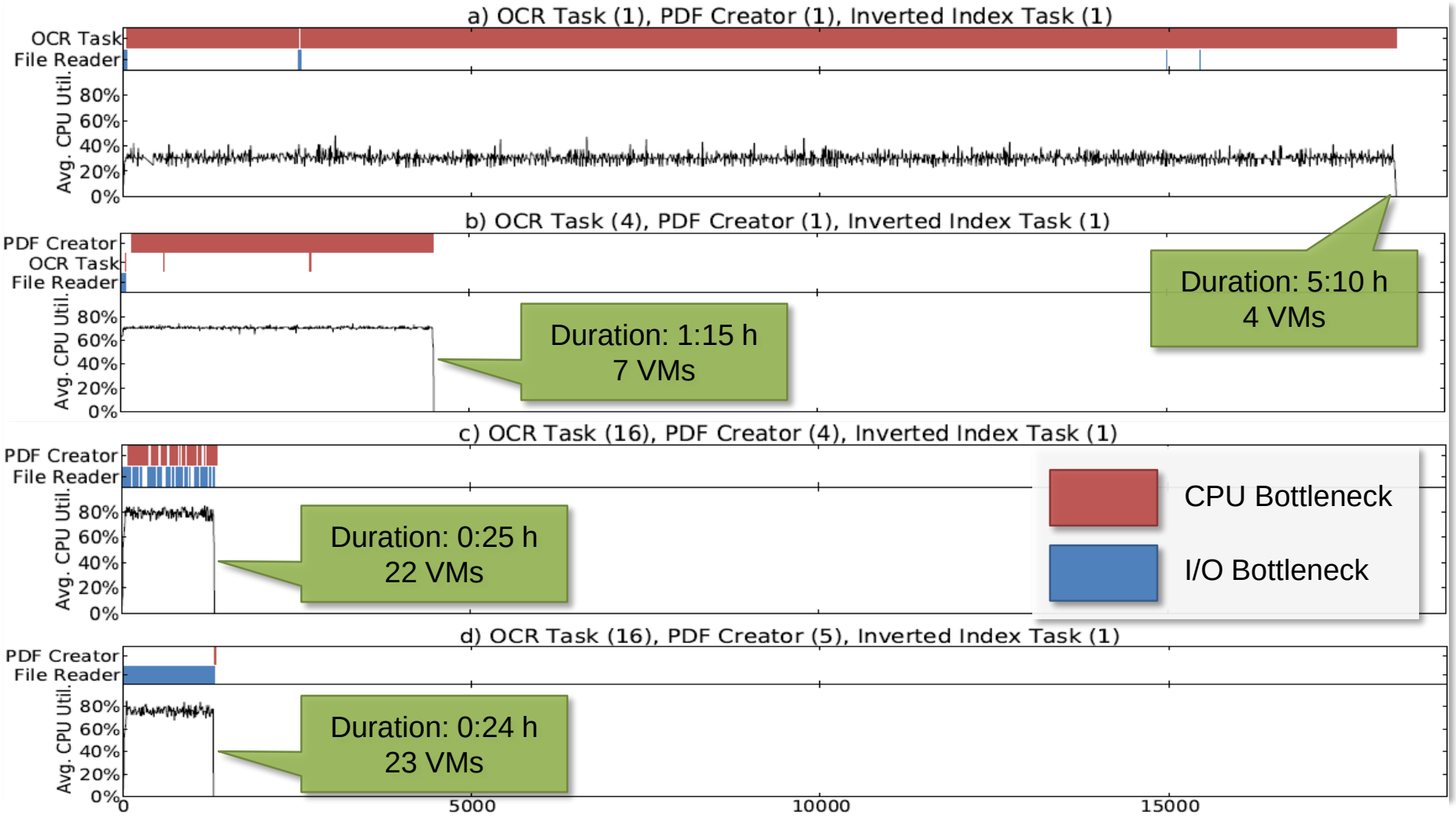
Channels are spanned according to a
"distribution pattern"

Evaluation (1/2)

- Evaluation job
  - Conversion of article DB
  - 40 GB of bitmap images to PDF
- Properties of job
  - Different computational complexities of tasks
  - Each parallel instance runs on separate VM (with 1 CPU core)
  - Input data reside on external storage
- Goal of evaluation
  - Find ideal degree of parallelization for each task



Evaluation (2/2)



Amazon Elastic MapReduce

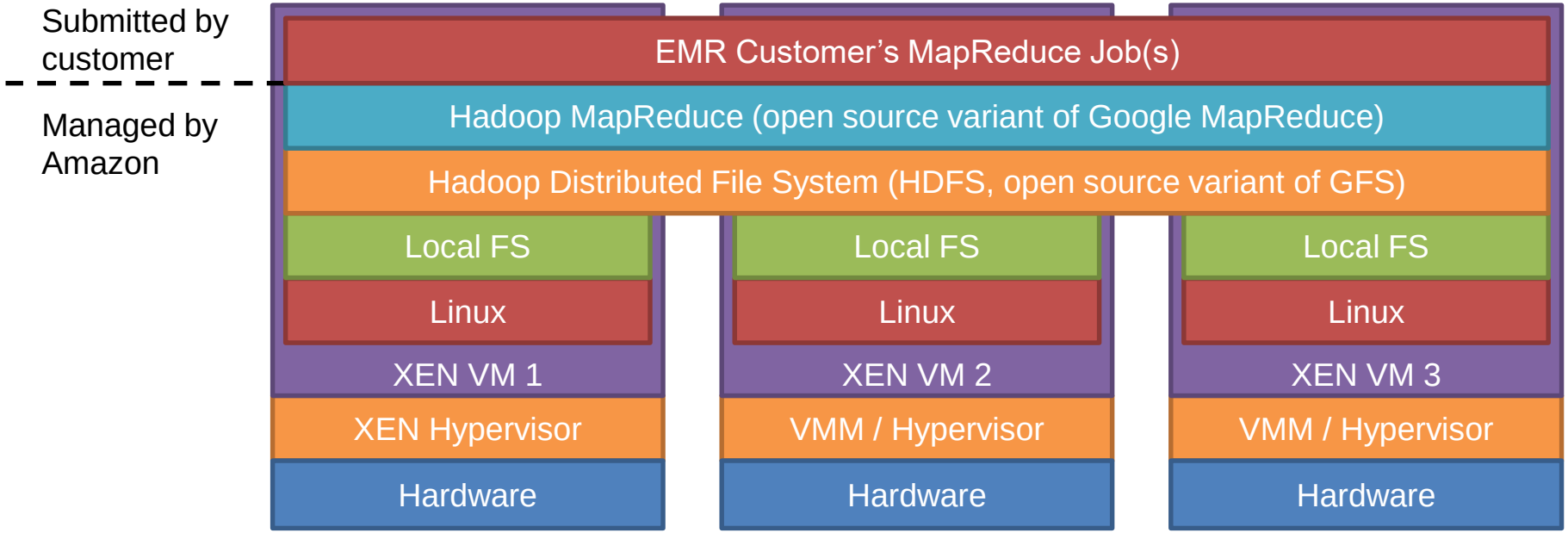
- Cloud service for data-intensive applications
  - Introduced in 2009 as part of AWS

- Bundles Apache Hadoop with EC2 infrastructure
  - No setup/configuration effort for EC2 customer

- Can be categorized as Platform-as-a-Service
  - Also follows Amazon‘s pay-by-the-hour model
  - Interesting for users with store data in S3 and new to process it infrequently

Software Stack of Amazon EMR

- Amazon runs preconfigured Hadoop on EC2 instances
  - User submits/monitors jobs through web interface
  - Dedicated VMs per user / no VM sharing
  - No direct access to VMs (like in IaaS case)



Job Processing Cycle on Amazon EMR (1/2)

1. Customer submits job through web interface
   - Job specification contains
     - Location of input data on Amazon S3
     - MapReduce code, user libraries, parameters, …
     - Number virtual machines to run the job on
     - Type of virtual machines to run the job on
     - Designated output location on Amazon S3
2. Requested virtual machines are started on EC2
   - Regular EC2 pricing model starts to apply
   - Booted AMIs to preconfigured to start HDFS/Hadoop
   - Hadoop worker nodes automatically contact master

Job Processing Cycle on Amazon EMR (2/2)

3. MapReduce job is processed on rented VMs
   - Customer can monitor execution progress through special web interface
4. After job completion, EC2 instances are automatically shut down
   - Output previously saved to Amazon S3
   - Job completed after all map/reduce tasks have finished
   - All VMs must remain allocated until that point in time
     - Intermediate data might be required for potential recovery
   - Shutdown also marks end of billing period

Executing Sequences of Jobs

- Deployment model requires to store initial input/final output on Amazon S3
  - Violates principle to keep data local to computation
  - Increased effort to move data to/from virtual machines

- EMR also allows to execute sequence of jobs
  - Initial input/final output still stored on Amazon S3
  - Intermediate results between MR jobs stored in HDFS
    - Data kept at least local between two MR jobs

EMR and Elasticity (1/2)

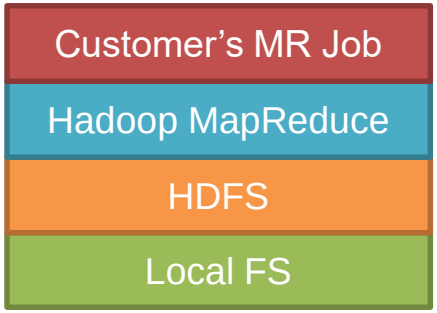
- EMR allows customers to adjust number of virtual machines while job is running
  - Customers can monitor job and respond to unexpected changes in workload

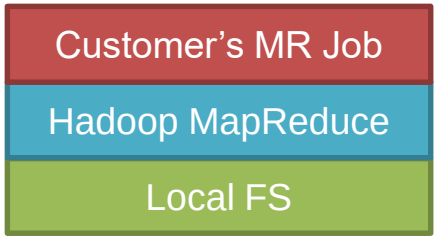
- Problem: VMs to be removed might store intermediate results in HDFS
  - HDFS expects node loss as result of hardware failure
    - ♦ If too many nodes are removed at the same time, replication mechanism cannot catch up
    - →Risk that job execution fails, initial input must be re-read from Amazon S3 (expensive!)

EMR and Elasticity (2/2)

- Solution: EMR separates VMs in three distinct groups
  1. Master group: Contains only VM running MR master
  2. Core group: VMs run Hadoop worker and HDFS node
     - ♦ Size of core group can only be increased, not decreased
  3. Task group: VMs only Hadoop worker
     - ♦ VMs store no local data between two MR jobs
     - ♦ Intermediate data is transferred to VMs of core group
     - ♦ Size of task group can be increased/decreased

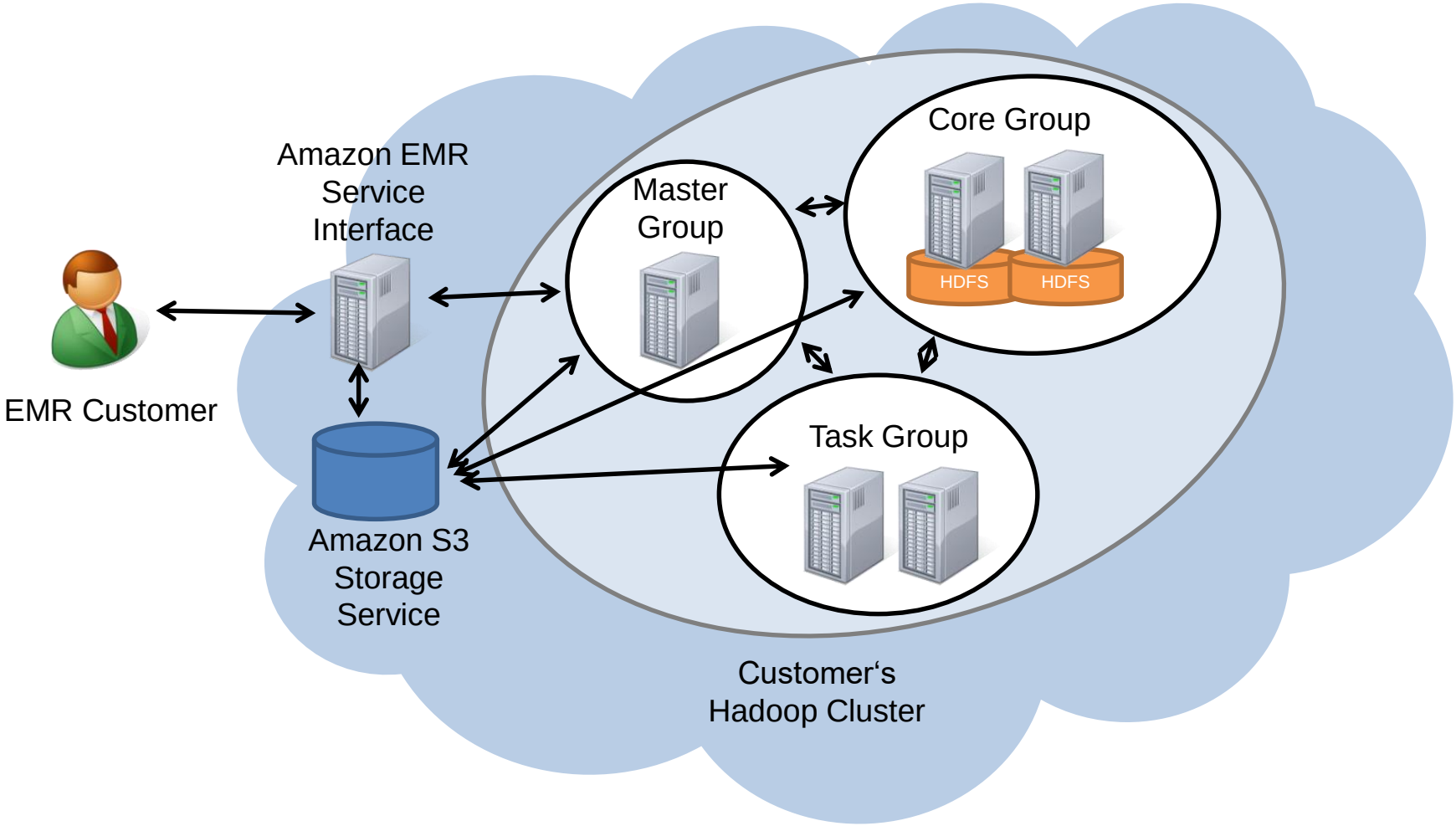
Software stack excerpt of a core group VM





Software stack excerpt of a task group VM

EMR Architectural Summary



EMR Pricing Model

- Amazon adds surcharge on regular EC2 usage fees

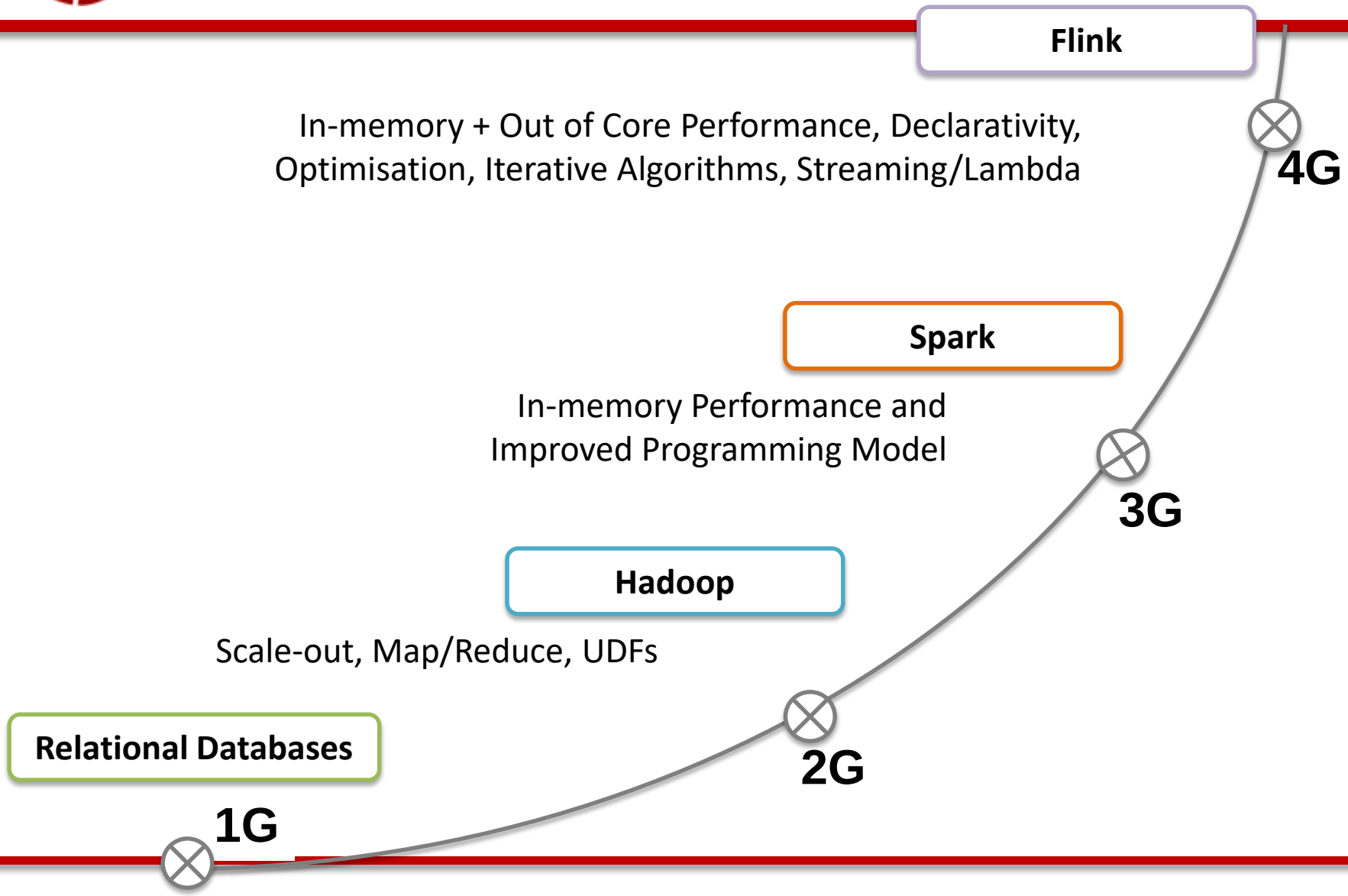
Instance Type	Amazon EC2 Price	Amazon EMR Price
Small	0.08 USD	0.015 USD
Medium	0.16 USD	0.03 USD
Large	0.32 USD	0.06 USD
Extra Large	0.64 USD	0.12 USD

- In addition, the regular AWS usage fees apply
  - S3 storage/access fee
  - Fee for Internet traffic
  - …

Overview

- Motivation
- MapReduce
- **Beyond MapReduce: Flink and Spark**

Evolution of Big Data Platforms



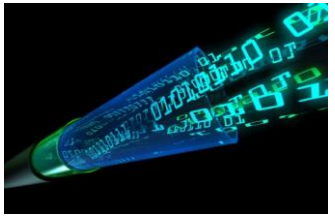
Technological Challenges for Deep Analysis on Big Data

Optimizing Access on Raw Data:
in-situ data analysis

Adaptive Seamless Deployment:
Scale from laptop to cluster

Evolving Datasets:
First results fast, stream mining

Advanced Data Analysis Programs:
Declarative specification and optimization of programs with iteration and state



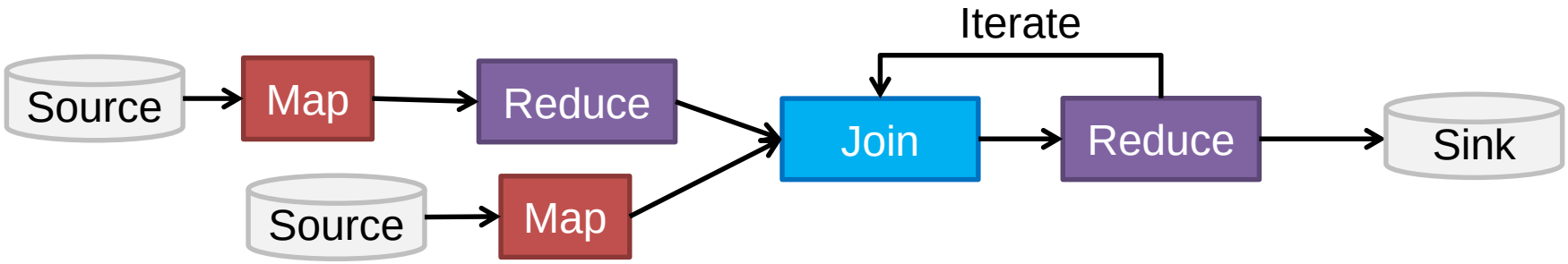
Low Latency Streaming:
Distributed, heterogeneous CPUs

Multi-tenancy:
Continuous, workload-aware optimizations

Engines: one size does not fit all - pluggable engines and libraries

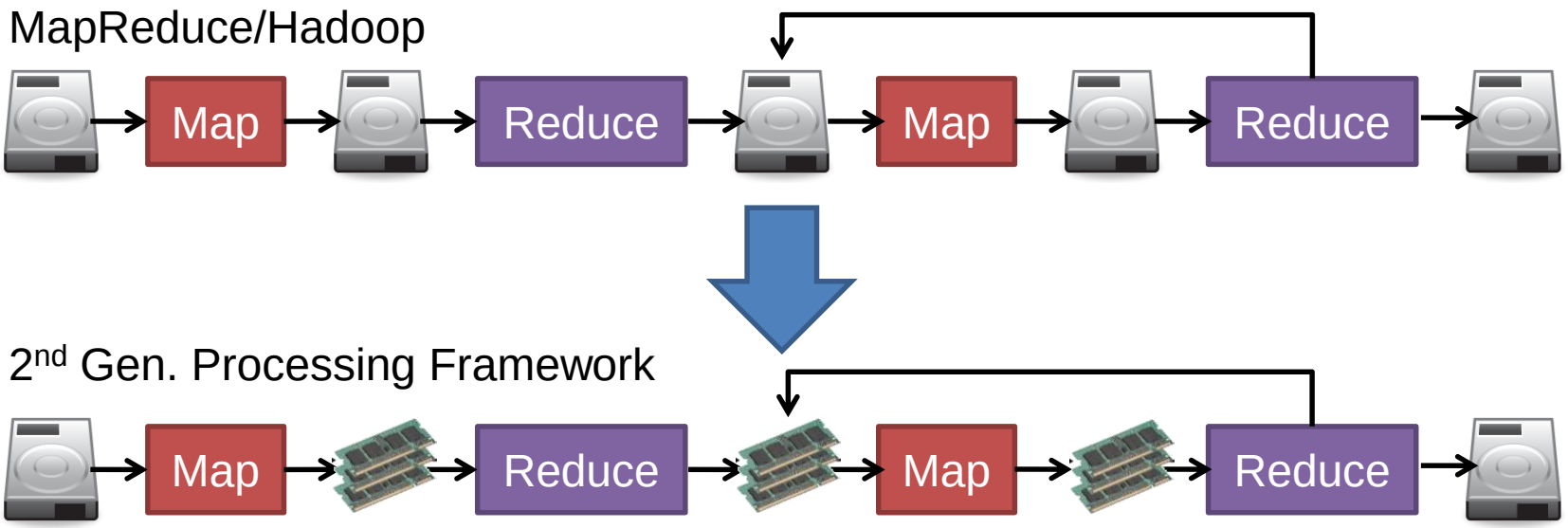
MapReduce Evolutions (1/3)

- Improved programming model with rich operator and functionality set
  - Joins, groupBys, filters, iterations and windows besides map and reduce operators
- This simplifies a wide array of computations, including:
  - Iterative machine learning,
  - Streaming and
  - Complex batch jobs



MapReduce Evolutions (2/3)

- Keep intermediate results in memory
  - Instead of writing them to disk
  - Can speed up iterative programs by 100x



MapReduce Evolutions (3/3)

- Generalize processing engine to handle streams
  - Use a batch engine but make batches really small
    - Also called Micro batches
    - Creates “illusion” of continuous stream
  - Use a real streaming engine
    - Continuous data flows rather than finite blocks
    - Batch becomes a special case of streaming

- Stream processing requires redefinition of operators
  - E.g. classic reduce function does not make sense
  - Instead: window operators (e.g. on all keys of last 5 sec.)

Successors of MapReduce

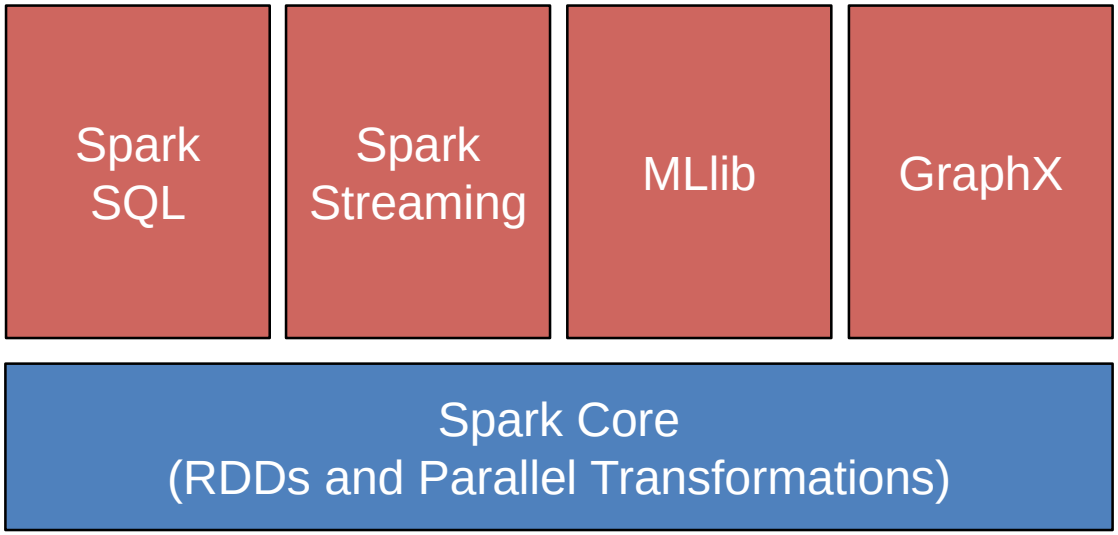
- MapReduce/Hadoop spawned several 2nd generation data processing frameworks
  - Address the shortcomings of classic MapReduce
  - Some are specialized, some general purpose

Apache Spark

- General purpose 2nd generation data processing framework with support for in-memory datasets and processing

- Open source-project, started as research project at UC Berkeley
- Apache top-level project since the beginning of 2014
  - Large user and developer base
  - Part of the platforms of Cloudera, Hortonworks and IBM

Apache Spark Stack



- At Spark's core are parallel transformations of Resilient Distributed Datasets (RDDs)
- Libraries built on-top of Spark for popular use cases

http://www-bcf.usc.edu/~minlanyu/teach/csci599-fall12/papers/nsdi_spark.pdf

Resilient Distributed Datasets (RDDs) (1/3)

- RDDs are read-only collections of objects:
  - distributed and partitioned across nodes
  - in-memory (intermediate results are not exchanged via disk like in MapReduce)

- Bulk operations (e.g. Map, Reduce, and Join) transform RDDs in parallel on workers

Resilient Distributed Datasets (RDDs) (2/3)

- RDDs are logical (lazy and ephemeral)
  - partitions of a dataset are materialized on demand when they are used
  - an RDD contains enough information to compute it starting from data in reliable storage
- Fault-tolerance through linage
  - Affected partitions are recomputed on failures (but partitions can depend on all partitions of preceding RDDs)
  - No overhead in failure-free case

Resilient Distributed Datasets (RDDs) (3/3)

- Caching: users can give explicit hints that datasets should be kept in-memory
- Enables faster iterative jobs and interactive analysis

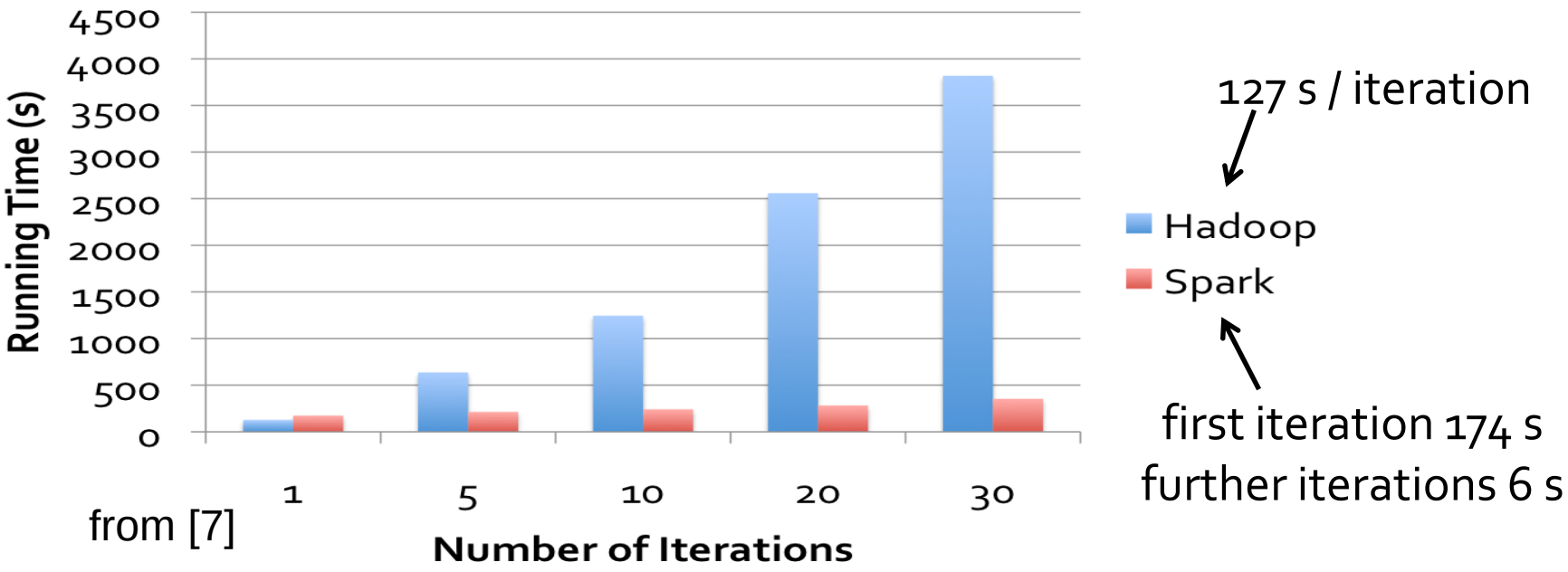
```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p => {…}).reduce(_ + _)
  w -= gradient
}
```

from [7]

- Model data kept in-memory across iterations!

Apache Spark Performance

- Logistic Regression on 29 GB dataset using 20 “m1.xlarge” EC2 nodes

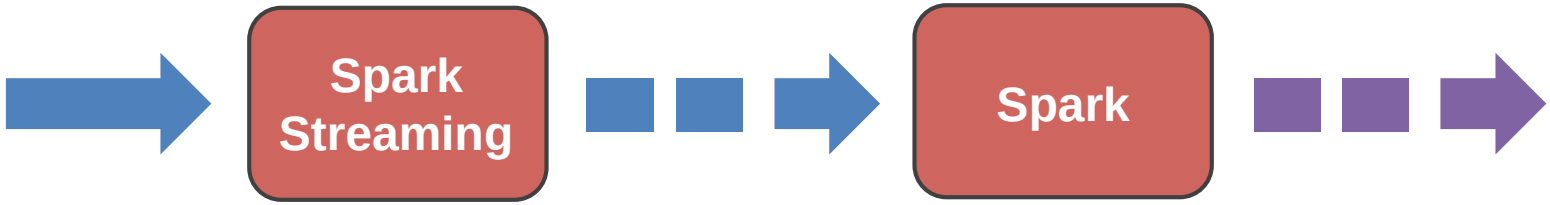


from [7]

- In-memory caching makes subsequent iterations fast

Spark Streaming

- Spark processes continuous inputs as Microbatches
  - Arriving input is processed in batches
  - Operations like aggregations and joins use these batches as their windows



Apache Flink

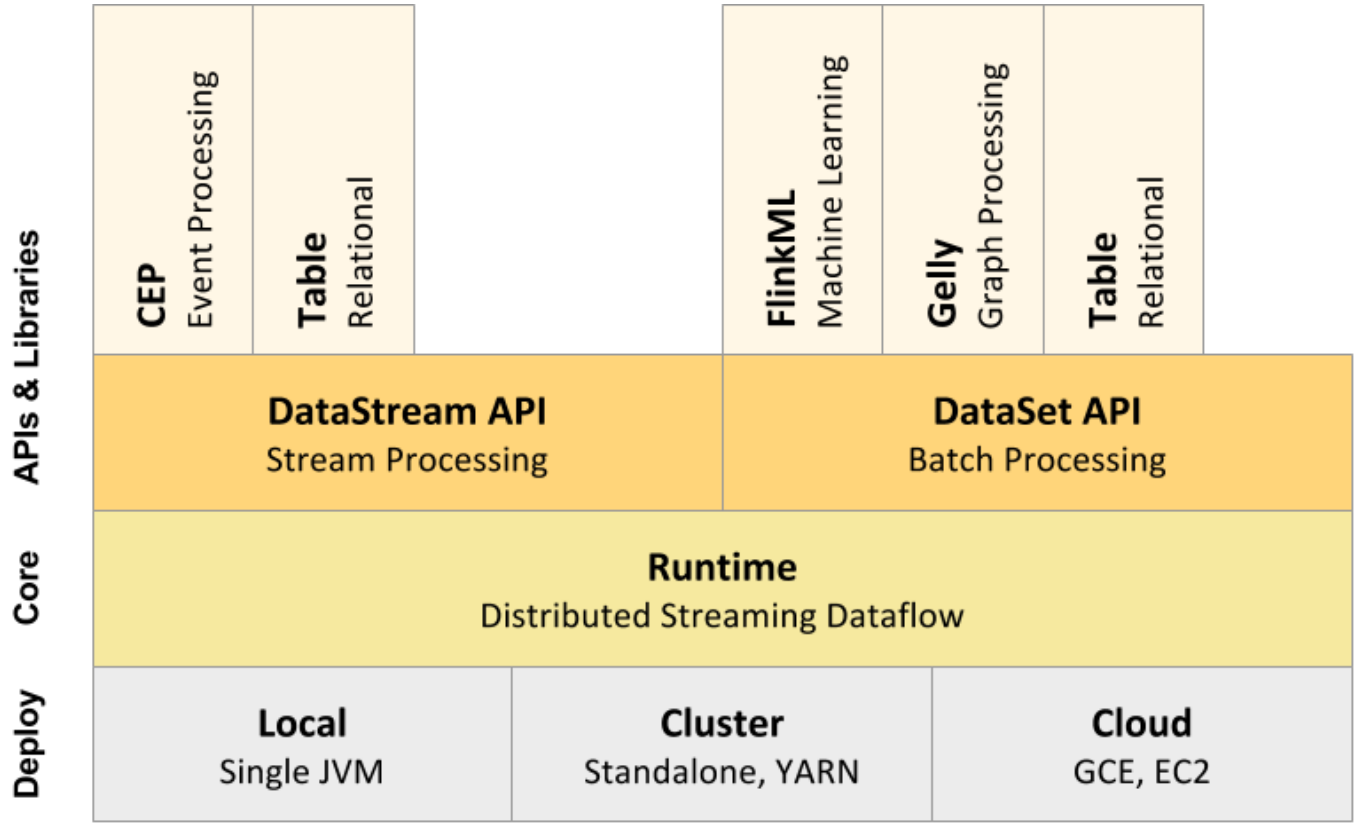
- Open-source project to unify batch and stream processing in one engine

Flink

- Started as joint research project from TU Berlin, HU Berlin and HPI Potsdam

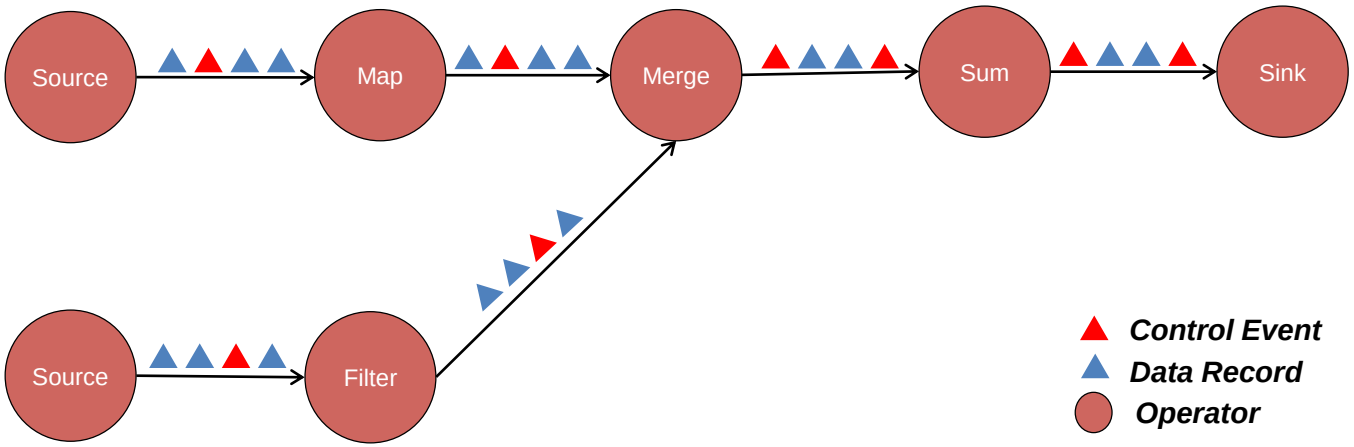
- Apache top-level project since beginning of 2015
  - Well-integrated in big data ecosystem
  - Vivid user and developer community worldwide

Apache Flink Stack



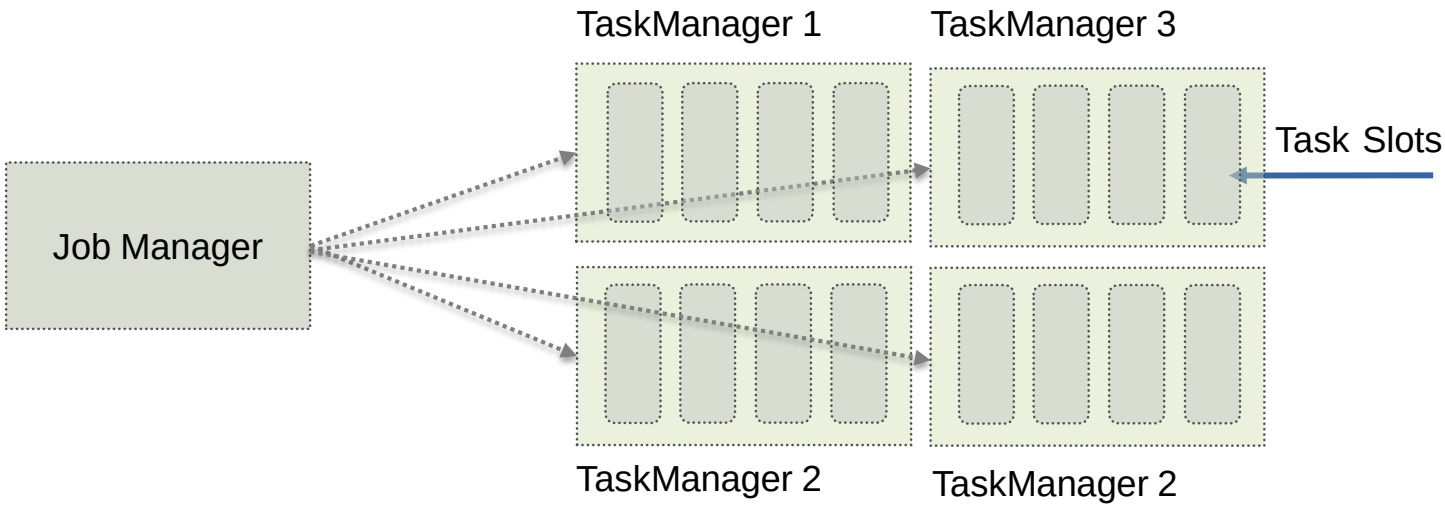
- Flink's core is a **streaming dataflow engine** that supports batch and streaming functionalities

Apache Flink Execution Model (1/3)



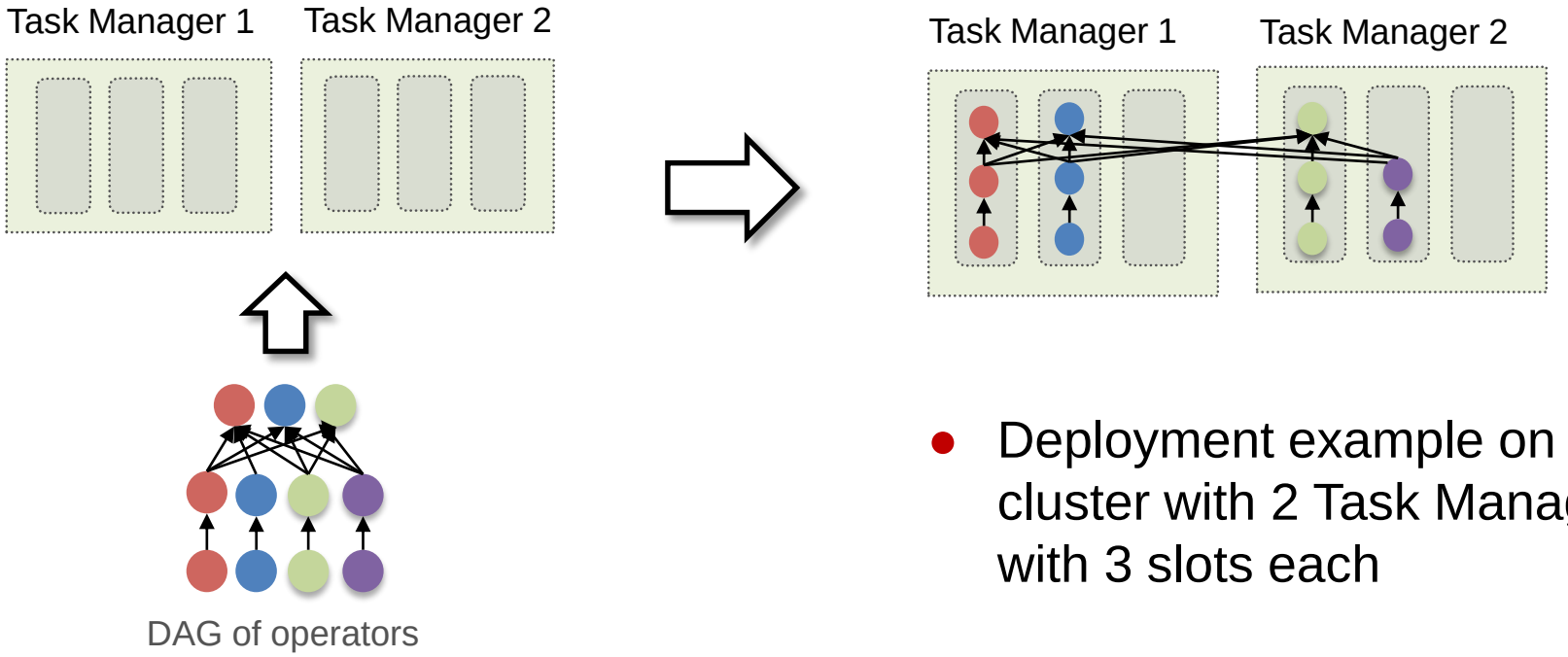
- A job consists of:
  - A directed acyclic graph (DAG) of operators and
  - Intermediate streams of data records and control events flowing through the DAG of operators
- ➢ Pipelined/Streaming engine

Apache Flink Execution Model (2/3)



- Follows a master-slave paradigm
  - Job Manager keeps track of all Task Managers and job execution
- Execution resources are defined through Task Slots
  - A Task Manager will have one or more Task Slots
  - Typically, equal to the number of cores per Task Manager

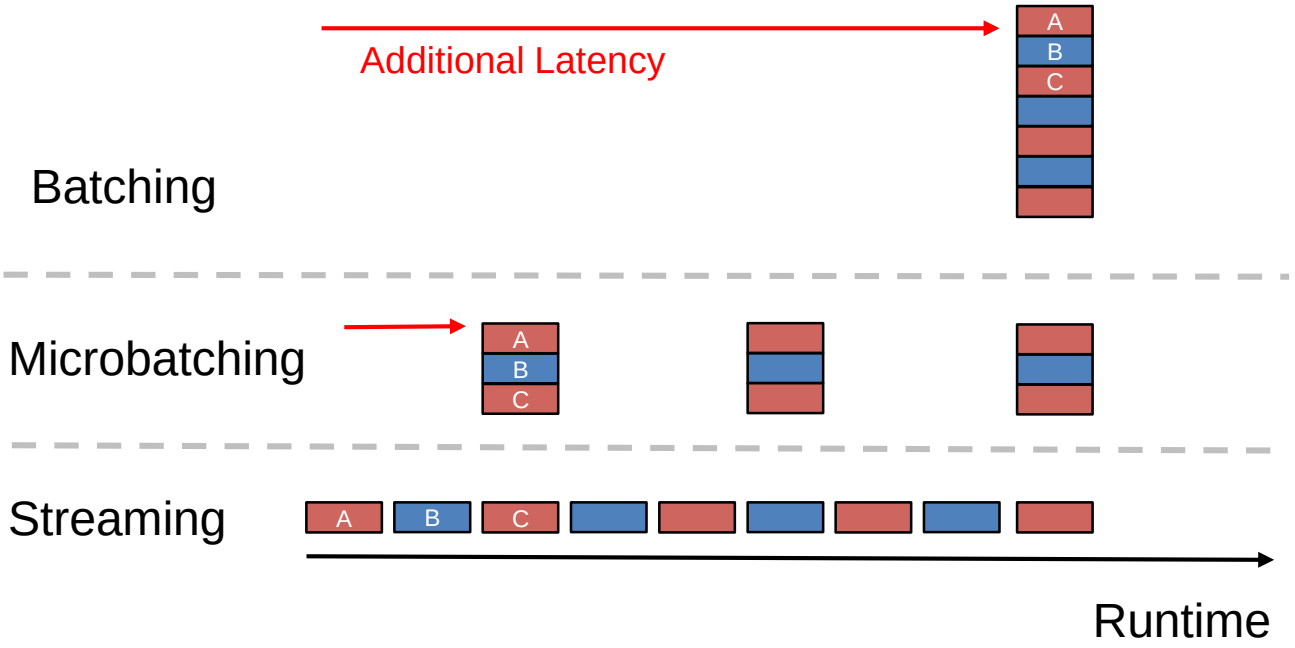
Apache Flink Execution Model (3/3)



- Deployment example on a cluster with 2 Task Managers with 3 slots each

- Complete DAG of operators are deployed distributed on all Task Managers
  - A Task Slot executes a pipeline of tasks (operators)
  - Often execute successive operators concurrently

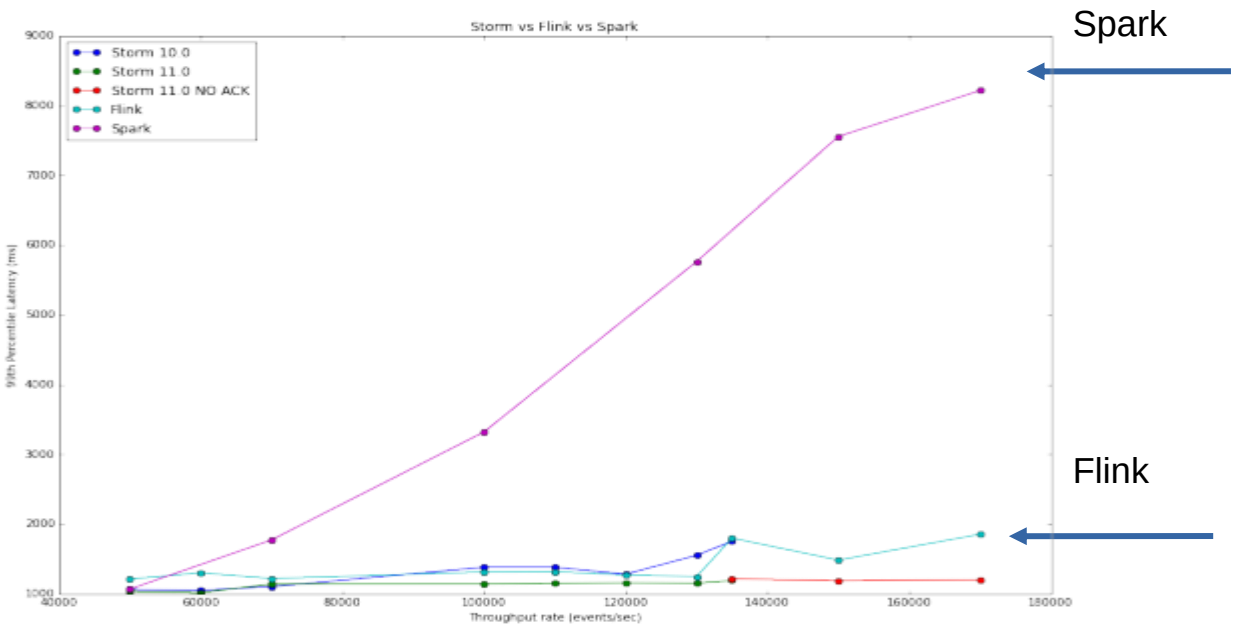
Performance Comparison of Streaming Approaches (1/3)



- Micro batches: Processing of small batches of tuples
- “Real” Streaming: Tuple-wise processing (lower latency possible)

Performance Comparison of Streaming Approaches (2/3)

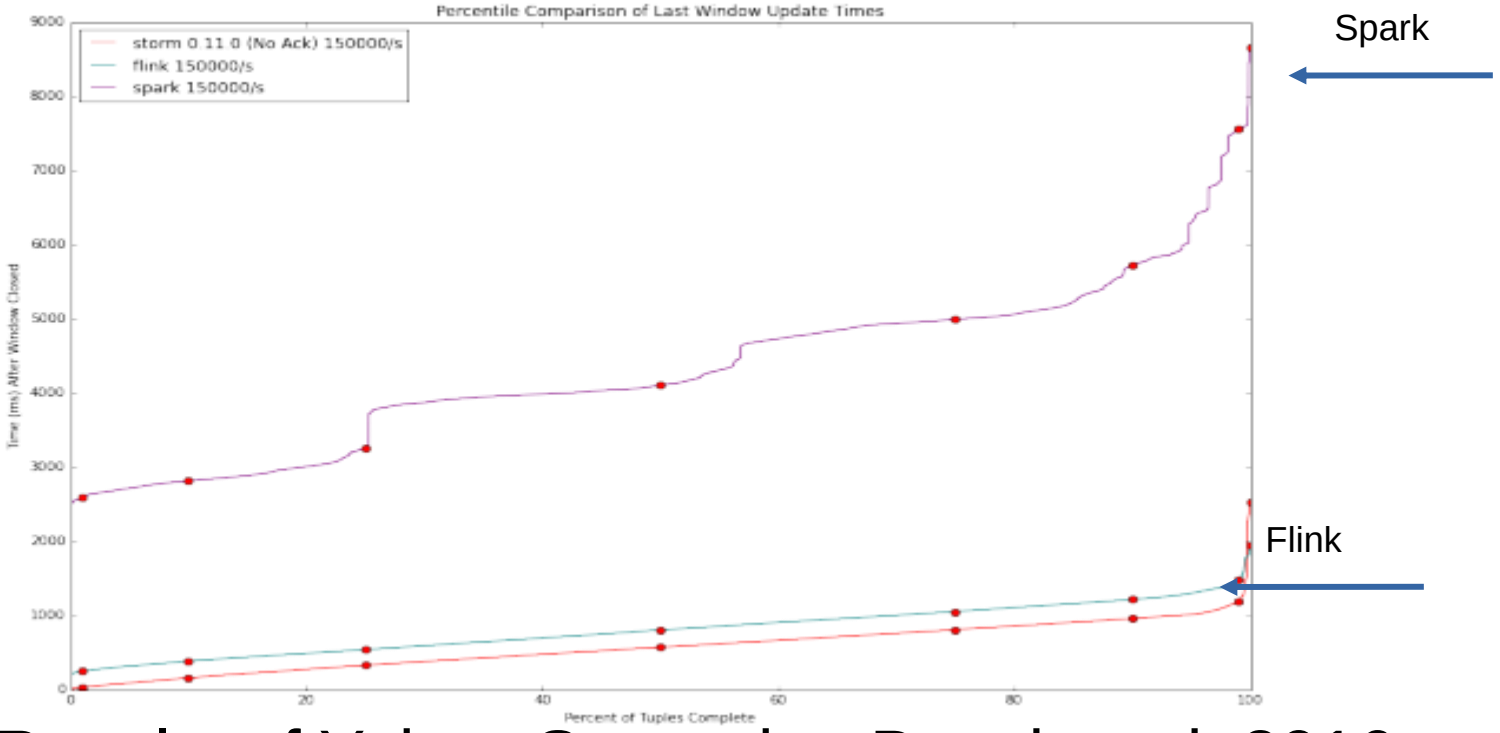
- Flink has lower latency (y-axis)
  - It processes an event as it becomes available



- Results of Yahoo Streaming Benchmark 2016:
  - https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Performance Comparison of Streaming Approaches (3/3)

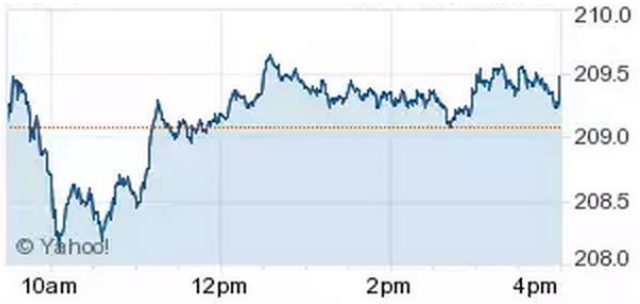
- Spark has higher throughput (y-axis)
  - Less overhead and meta data with microbatches



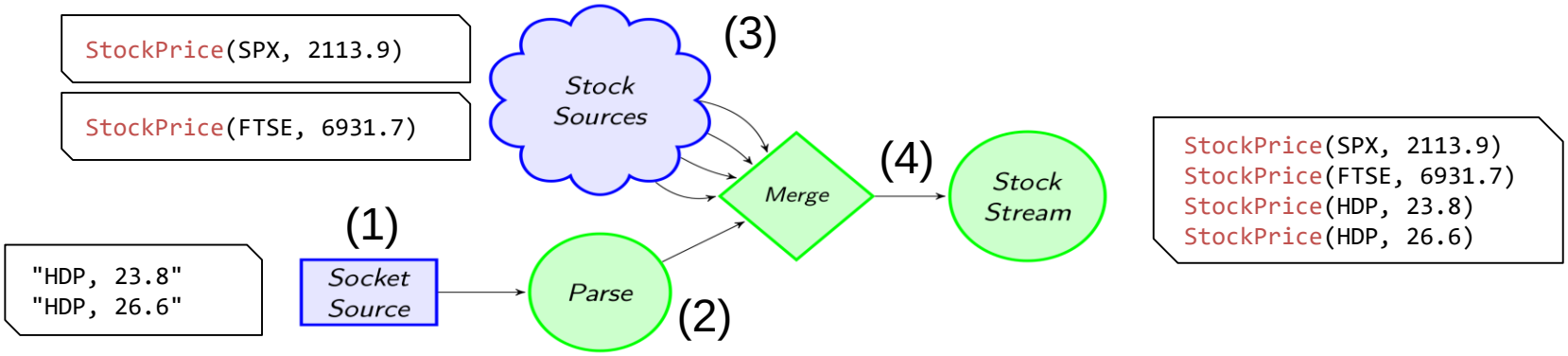
- Results of Yahoo Streaming Benchmark 2016:
  - https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Flink Streaming: Use Case

- Reading from multiple inputs
  - Merge stock data from various sources
- Window aggregations
  - Compute simple statistics over windows of data
- Data-driven windows
  - Define arbitrary windowing semantics
- Detailed explanation and source code on:
  - http://flink.apache.org/news/2015/02/09/streaming-example.html

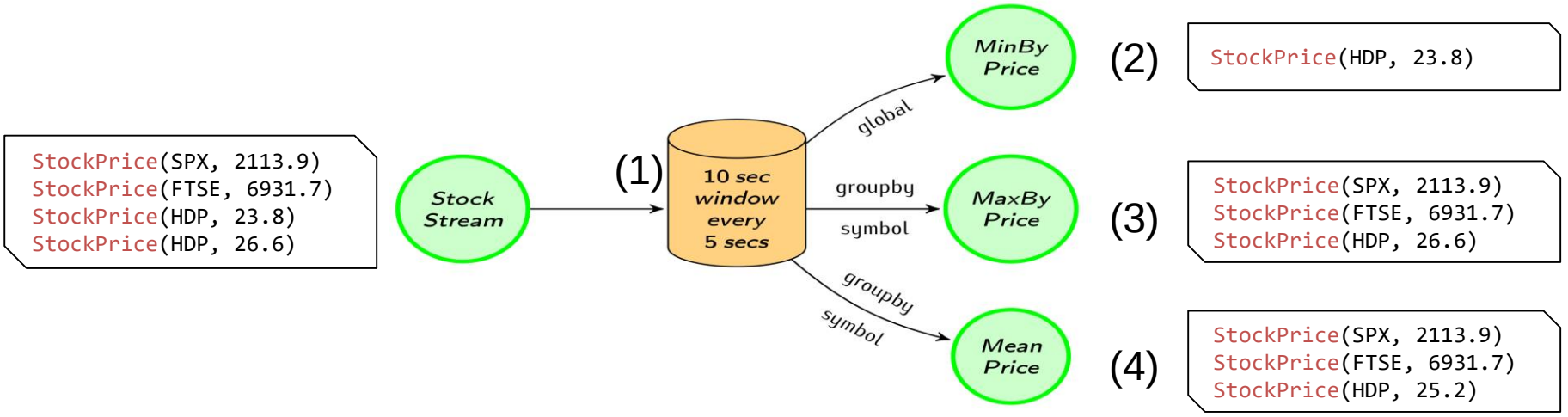


Flink Streaming: Reading from Multiple Inputs



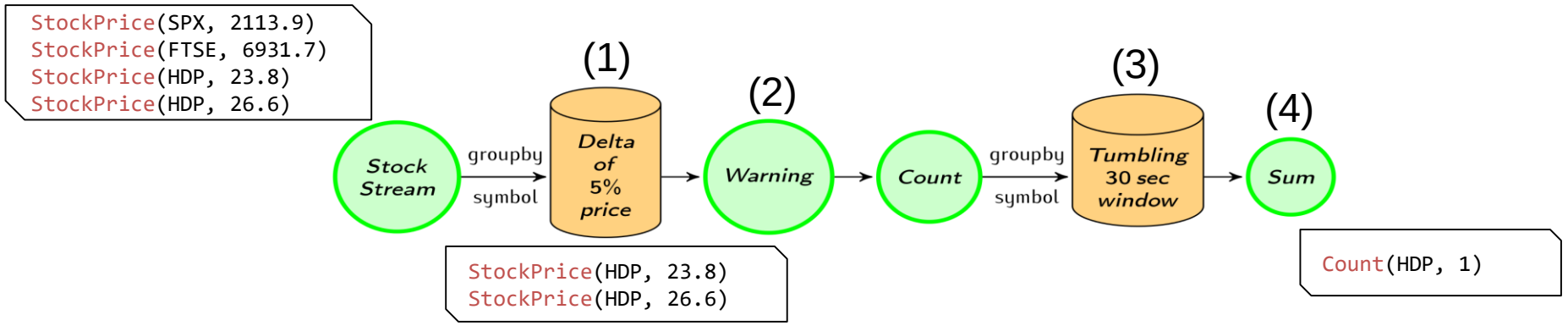

```
case class StockPrice(symbol : String, price : Double)
val env = StreamExecutionEnvironment.getExecutionEnvironment

(1) val socketStockStream = env.socketTextStream("localhost", 9999)
(2)     .map(x => { val split = x.split(",")
            StockPrice(split(0), split(1).toDouble) })

(3) val SPX_Stream = env.addSource(generateStock("SPX")(10) _)
    val FTSE_Stream = env.addSource(generateStock("FTSE")(20) _)
(4) val stockStream = socketStockStream.merge(SPX_Stream, FTSE_STREAM)
```

Flink Streaming: Window Aggregations




```
    val windowedStream = stockStream
(1)   .window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))

(2) val lowest = windowedStream.minBy("price")
(3) val maxByStock = windowedStream.groupBy("symbol").maxBy("price")
(4) val rollingMean = windowedStream.groupBy("symbol").mapWindow(mean _)
```

Flink Streaming: Data-driven Windows




```
case class Count(symbol : String, count : Int)

val priceWarnings = stockStream.groupBy("symbol")
(1)     .window(Delta.of(0.05, priceChange, defaultPrice))
(2)     .mapWindow(sendWarning _)

val warningsPerStock = priceWarnings.map(Count(_, 1)) .groupBy("symbol")
(3)     .window(Time.of(30, SECONDS))
(4)     .sum("count")
```

Research at CIT

Nephele

StratoSphere
Above the Clouds

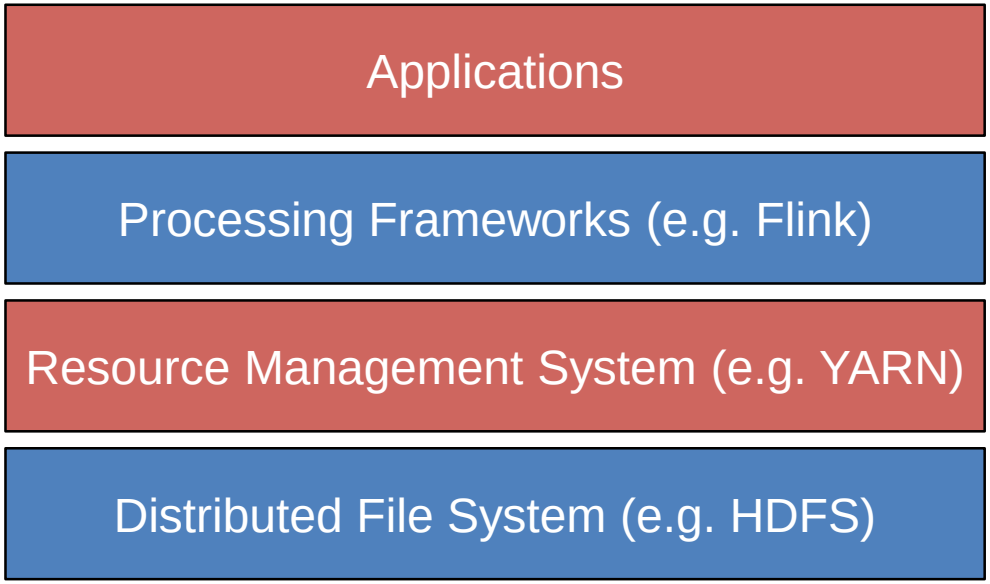
Flink

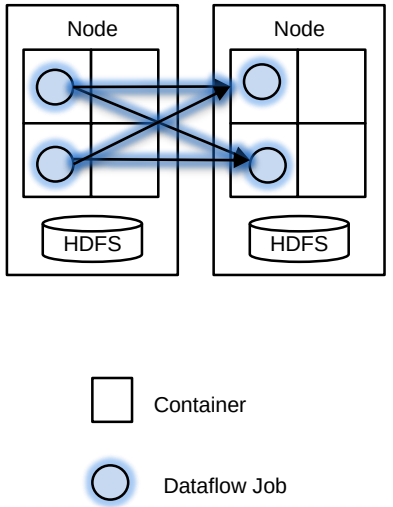
BBDC
BERLIN BIG DATA CENTER

- Current Research at CIT:
  - Adaptive resource management
  - Adaptive fault-tolerance
  - Stream processing with latency guarantees
- Available theses: http://www.cit.tu-berlin.de/menue/theses/

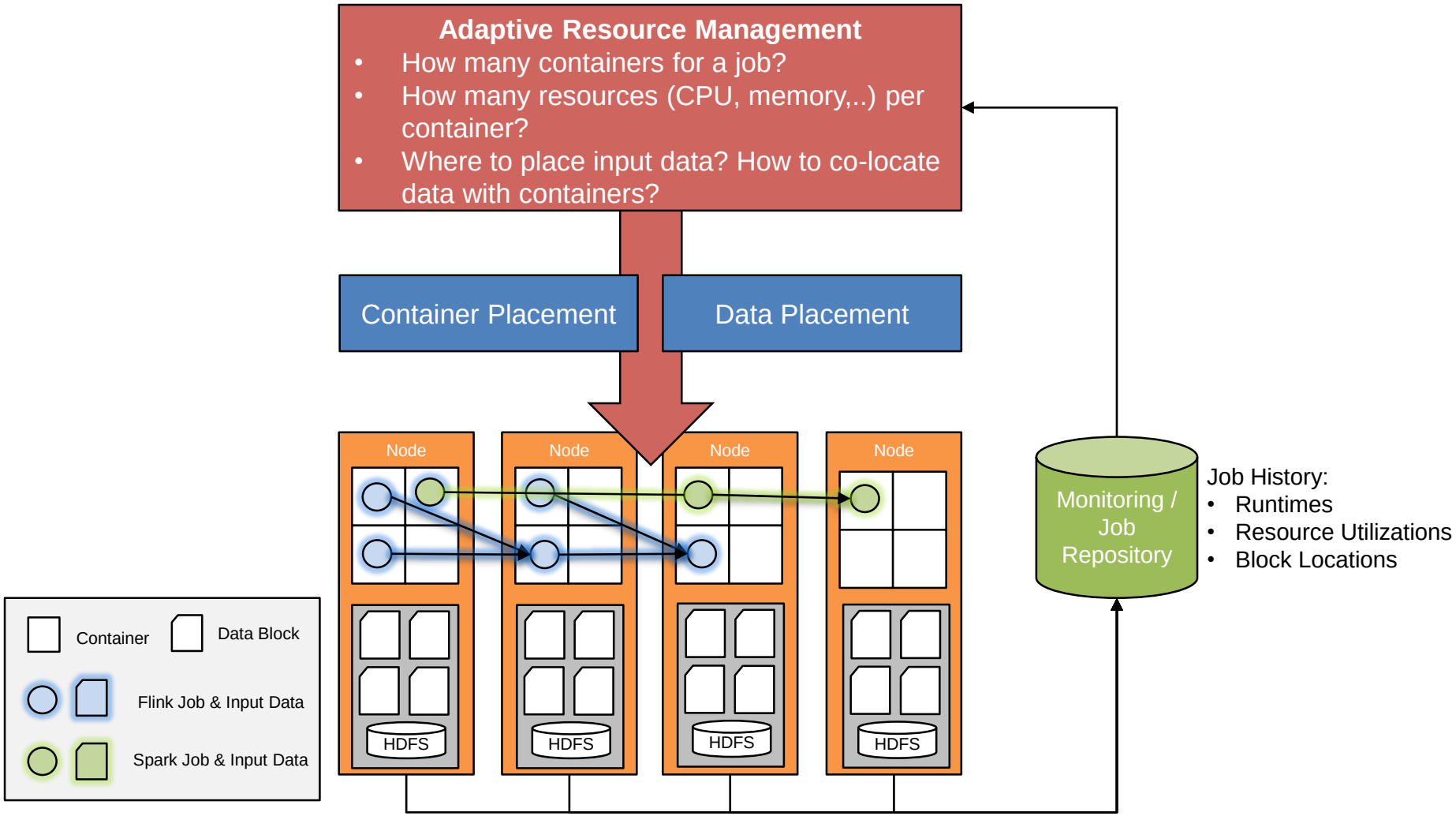
Resource Management

- Resource management systems allow multiple applications, users, and frameworks to share a cluster through containers



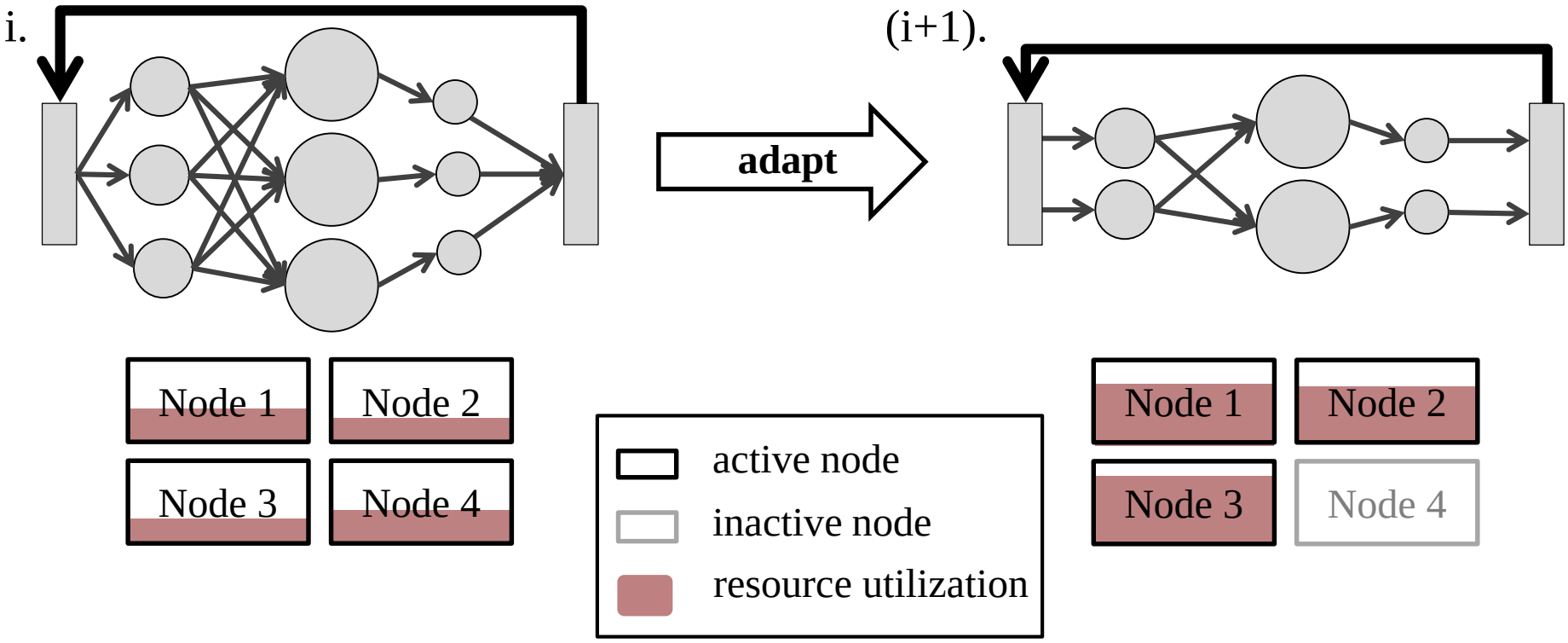


Adaptive Resource Management



Dynamically Scaling Iterative Dataflow Jobs (1/2)

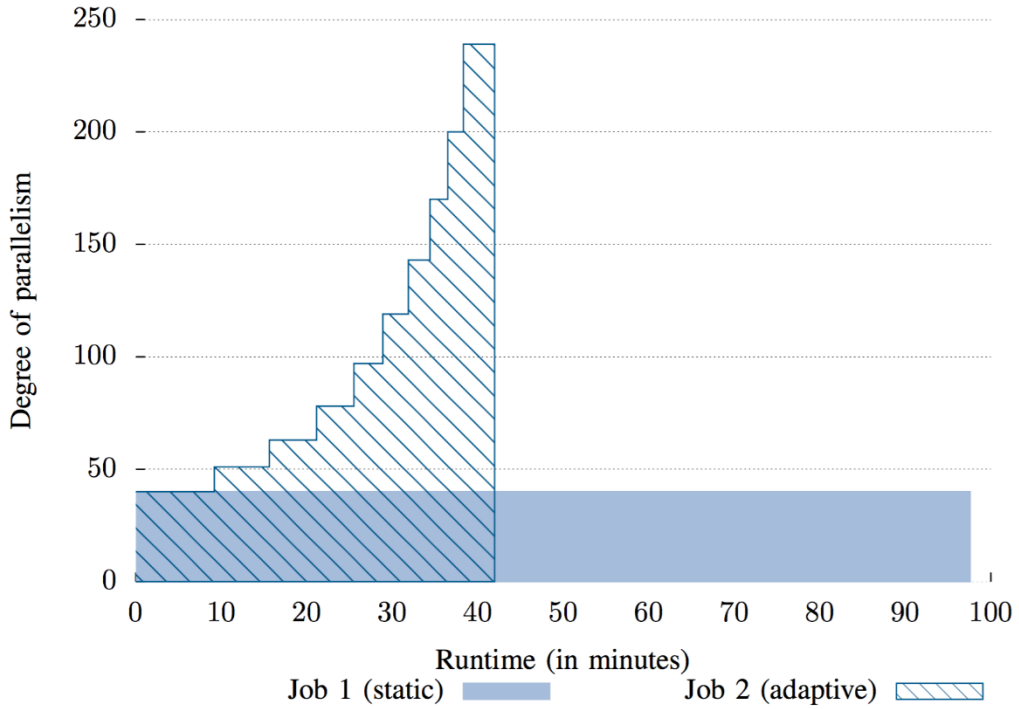
- Adapting resource allocation in-between iterations based on actual resource utilization



Dynamically Scaling Iterative Dataflow Jobs (2/2)

- Scaling K-means automatically based on CPU utilization: grouping a 100 GB of three-dimensional points into 10 clusters

- More than twice as fast using roughly the same amount of compute resources (number of seconds spent on nodes)



References

[1] R. Ramakrishnan and A. Tomkins. “Toward a peopleweb”. IEEE Computer, 40(8):63–72, 2007.

[2] R.E. Bryant, R.H. Katz, E.D. Lazowska: “Big-Data Computing: Creating revolutionary breakthroughs in commerce, science, and society”, in Computing Research Initiatives for the 21st Century. Computing Research Association, 2008

[3] J. Fenn, H. LeLong: “Hype Cycle for Emerging Technologies”, Gartner, 2011

[4] J. Dean, S. Ghemawat, “MapReduce: simplified data processing on large clusters”, Communications of the ACM, 51 (1), 2008

[5] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins: “Pig latin: a not-so-foreign language for data processing”, in Proc. of the 2008 ACM SIGMOD International Conference on Management of Data, 2008

[6] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, R. Murthy: “Hive –A Petabyte Scale Data Warehouse Using Hadoop”, in Proc. of the 26th International Conference on Data Engineering (ICDE), 2010

[7] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica. Spark: Cluster Computing with Working Sets, in Proc. of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud’2010), 2010.