Methods of Cloud Computing

Programming Cloud Resources 2: Data-Intensive Applications

Complex and Distributed Systems
Faculty IV
Technische Universität Berlin

Operating Systems and Middleware
Hasso-Plattner-Institut
Universität Potsdam

Overview

- **Introduction**
- Distributed Storage
  - GFS
  - HDFS
  - Bigtable
- Distributed Processing
  - MapReduce
  - Spark
  - Flink

Motivation

- Amount of digitally available data grows rapidly
  - WWW
    - Google processes hundreds of TB of Web pages and millions of clicks for its services[1,2]
  - Data warehousing
    - HP built 4 PB data store for Wal-Mart[3]
  - Scientific data
    - LHC produces 60 TB per day[3]
- Traditionally domain of Internet/search companies, e.g.
  - Google: GFS, MapReduce, Dataflow/Beam
  - Microsoft: Dryad, SCOPE

Data-Driven Applications

lifecycle management

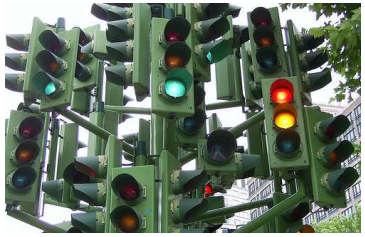
home automation

health

water management

market research

information marketplaces



traffic management

energy management

- Trends: Digitalization, Internet of Things, Industry 4.0

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

- Requirements for data-intensive applications out-scaled several traditional solutions
  - Commercial parallel databases and HPC computers often also prohibitively expensive

- New distributed systems for commodity clusters
  - Both storage and computing across commodity nodes, which allows to keep computation close to the data
  - Individual servers are less reliable, so fault tolerance must be provided by distributed systems

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

- Cloud computing (particularly IaaS) is a good fit for data-intensive applications
  - Looks like a large pool of cluster nodes to customer
  - Possibility of elastic scaling
  - 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 and infrequent users, while heavy users might prefer dedicated clusters

Challenges of Large-Scale Data Processing

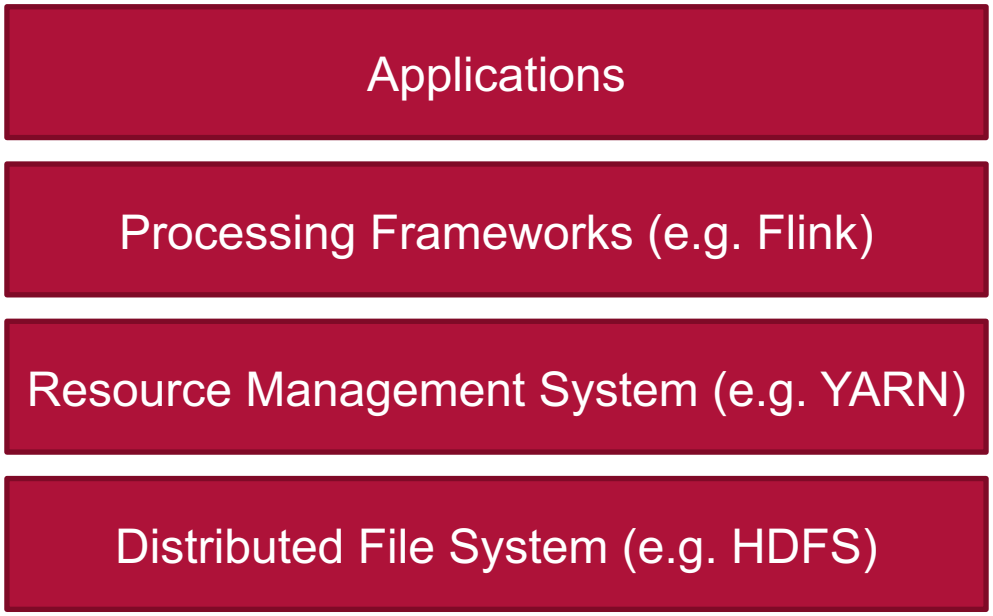
- Large clusters/clouds have 100s to 1000s of servers
  - Extremely high performance/throughput possible
  - But jobs must be written to take advantage of the massively parallel and distributed environment
- Writing efficient parallel and distributed processing jobs is hard
  - Most developers are no experts in parallel programming, distributed systems, and data processing
  - Developers also don’t want to care about concurrency and failures, but about extracting new information
- Needed: Suitable abstraction layers for developers

Distributed Systems for Data-Intensive Applications

- Requirements for distributed data processing systems
  1. Developers don‘t have to think about parallelization and distributed execution
     - Should be able to write sequential code that is independent of the degree of parallelism at runtime
     - *System* schedules and manages distributed tasks
  2. Developers don‘t have to think about load balancing
     - *System* distributes the work evenly across workers
  3. Developers don‘t have to think about fault tolerance
     - *System* detects failed nodes / executors
     - *System* re-executes any lost computations

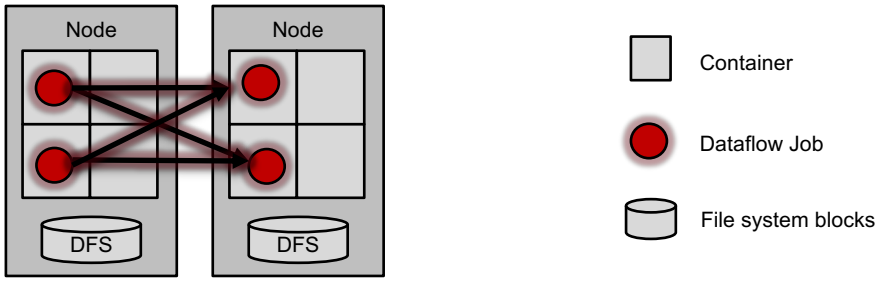
Analytics Cluster Setup (1/2)

- Analytics clusters often shared by multiple users and applications
- Shared via resource management and distributed file systems



Analytics Cluster Setup (2/2)

- Users reserve resources temporarily in containers from large sets of managed commodity nodes
- Containers: bundle of resources on a specific node assigned to a specific distributed job, often without any isolation (i.e. resource management “containers”, not necessarily OS-level containers)



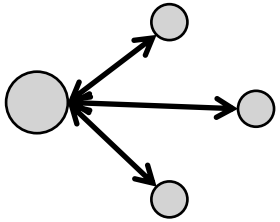
Comparison to HPC (1/2)

- Largely distinct communities and tools

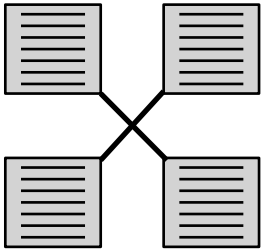
High-Performance Computing	Data-intensive Applications / Distributed Dataflows
Generic and low-level programming abstractions, explicit parallelism and synchronization and communication, optimized for specific hardware	High-level declarative abstractions, restricted programming models, comprehensive frameworks and runtime environments
Often more tightly coupled dependent computations, messaging and fine-grained updates to distributed state	Typically data-parallel problems with little coupling, parallelizable work that exceeds distributed communication
High-performance parallel computers and high-speed interconnects	Commodity nodes, used for both compute and storage, and commodity network technology

Comparison to HPC (2/2)

Flexible and fast low-level code

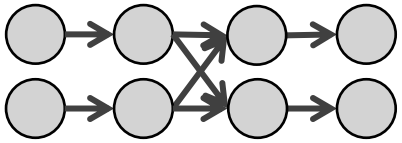


Architecture-specific implementations

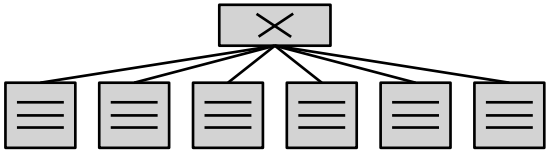


High-performance hardware

High-level dataflows



Scalable fault-tolerant distributed engines



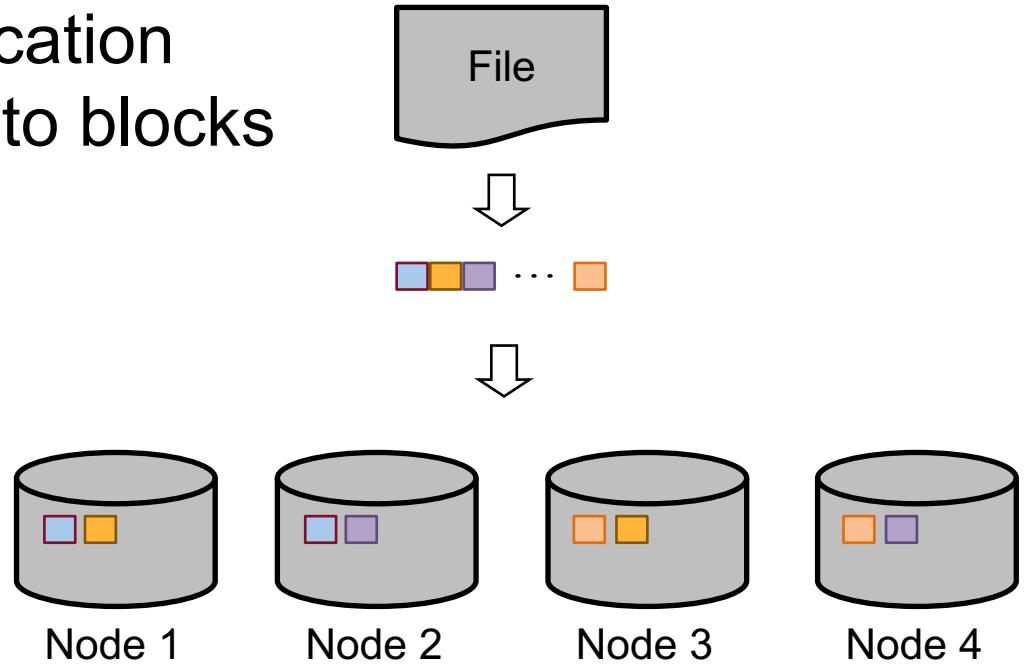
Commodity clusters

Overview

- Introduction
- **Distributed Storage**
  - GFS
  - HDFS
  - Bigtable
- Distributed Processing
  - MapReduce
  - Spark
  - Flink

Distributed Data

- Distributed file systems and systems on top (e.g. Bigtable on GFS), running on commodity servers
- Fault tolerance and parallel access through replication of data that is split into blocks



Overview

- Introduction
- Distributed Storage
  - **GFS**
  - HDFS
  - Bigtable
- Distributed Processing
  - MapReduce
  - Spark
  - Flink

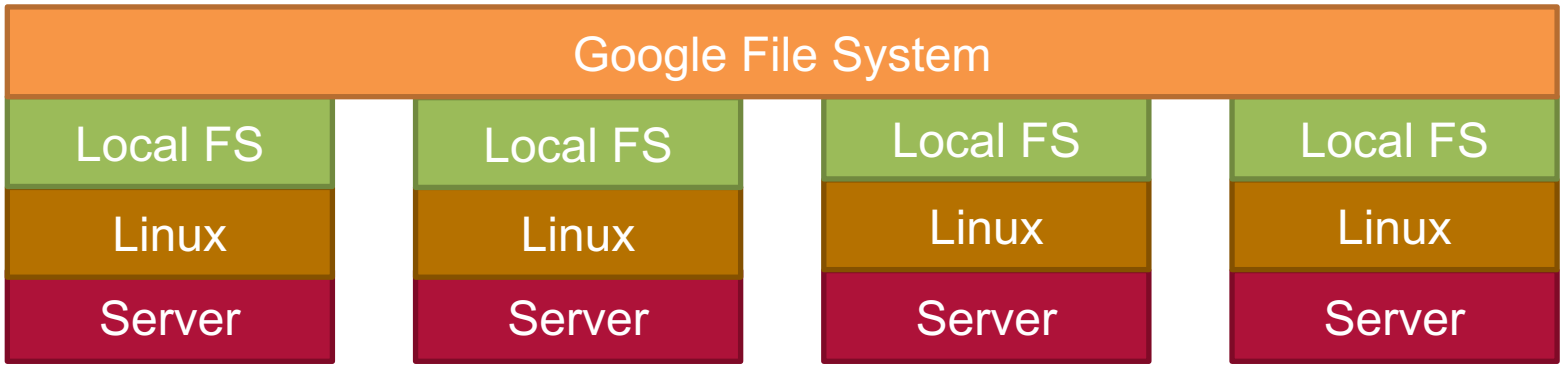
Google File System[4] Design (1/2)

- GFS was designed to meet the following criteria
  - Scalability: Must store hundreds of terabytes
  - High performance: High throughput (over low latency)
  - Support for commodity clusters (1000s of servers)
  - Fault-tolerance: Compensate individual server failures

- Google‘s workload characteristics
  - Individual files are expected to be big (MBs to GBs)
  - Billions of files must be managed by the file system
  - Most files are written only once, then read sequentially

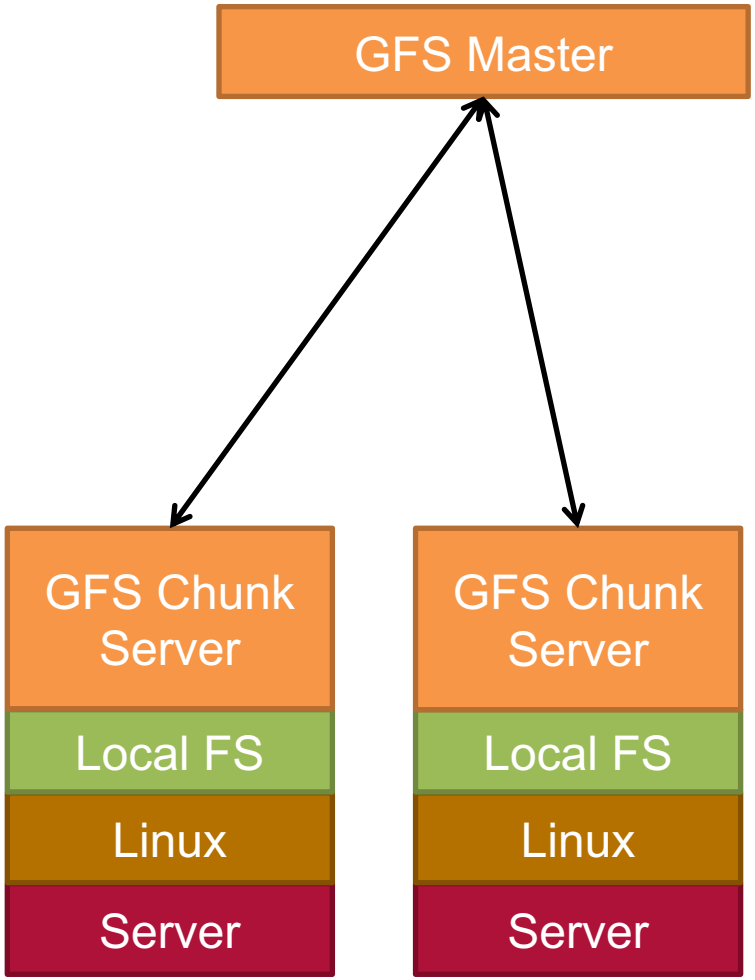
Google File System[4] Design (2/2)

- GFS is not a POSIX-compliant file system
  - Limited support for random writes (expected to happen rarely), data can be efficiently appended instead
  - Sits on top of regular file systems
  - Custom API to the developer



Google File System Architecture

- GFS follows master-worker architecture
  - Master stores metadata
  - Workers store actual data
    - Called chunk server
    - One chunk server per node
- Files are split into fixed-sized chunks
  - Identified by GUIDs
  - Replicated across machines

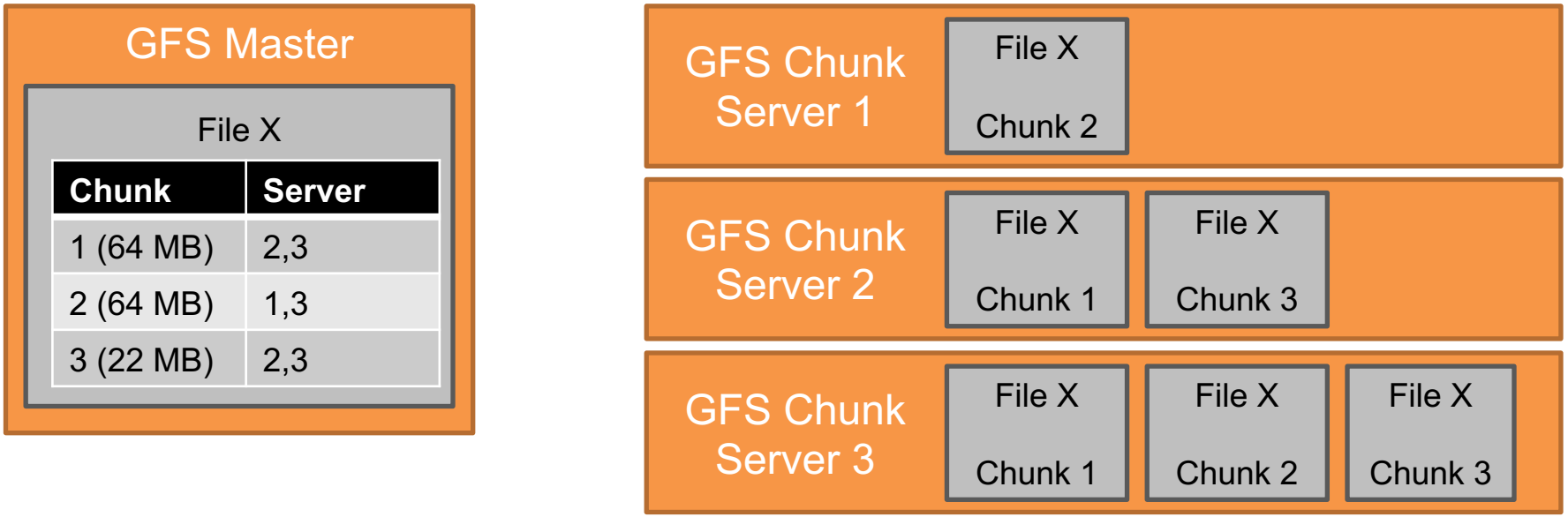


Example of Data Storage in Google File System

- Lets assume, we have file X with a size of 150 MB
  - Default chunk size is 64 MB

→File is stored in three chunks

→Chunks are stored across the available chunk servers



Reading File X From GFS

- Lets assume client wants to read X starting at MB 100
  1. Client knows chunk size, converts offset to chunk index
  2. Client contacts master with filename and chunk index
  3. Master returns chunk handle and list of replicas
  4. Client sends read request to closest replica
  5. Client caches chunk information so further reads of the same chunk require no more client-master interaction (until cached information expires)

- Note: Master not involved in actual data transfer!

Design Considerations for the GFS Master

- Master has global knowledge of the file system
  - Simplifies design and reduces response times
  - Easy garbage collection and data reorganization

- Meta data is kept in main memory for performance
  - Meta data for each chunk consumes 64 bytes

  → Chunk size is an important parameter of GFS

    → Determines the amount of meta data that fits into memory

    → Determines the frequency of the client requests

What Happens When the Master Fails?

- Meta data is crucial to the functionality of the FS
- GFS knows three types of meta data
  - Namespace mappings (files, directories)
  - File-to-chunk mappings
  - Replica locations

- Namespace/file-to-chunk mappings are written to log
  - Operation log is persistent on master‘s hard disk
  - Replicated to remote machine
- Replica locations are requested from chunk servers when the master starts or new chunk server joins

Operation Log

- Serves as logical timeline that defines the order of concurrent operations
  - Files and chunks (as well as their version) are all uniquely and eternally identified by their logical creation times
  - Changes to the file system are only made visible to clients after the log has been flushed locally and remotely
- Master can recover FS state by replaying log
- When log grows beyond certain size the master creates snapshot of meta data to improve startup times

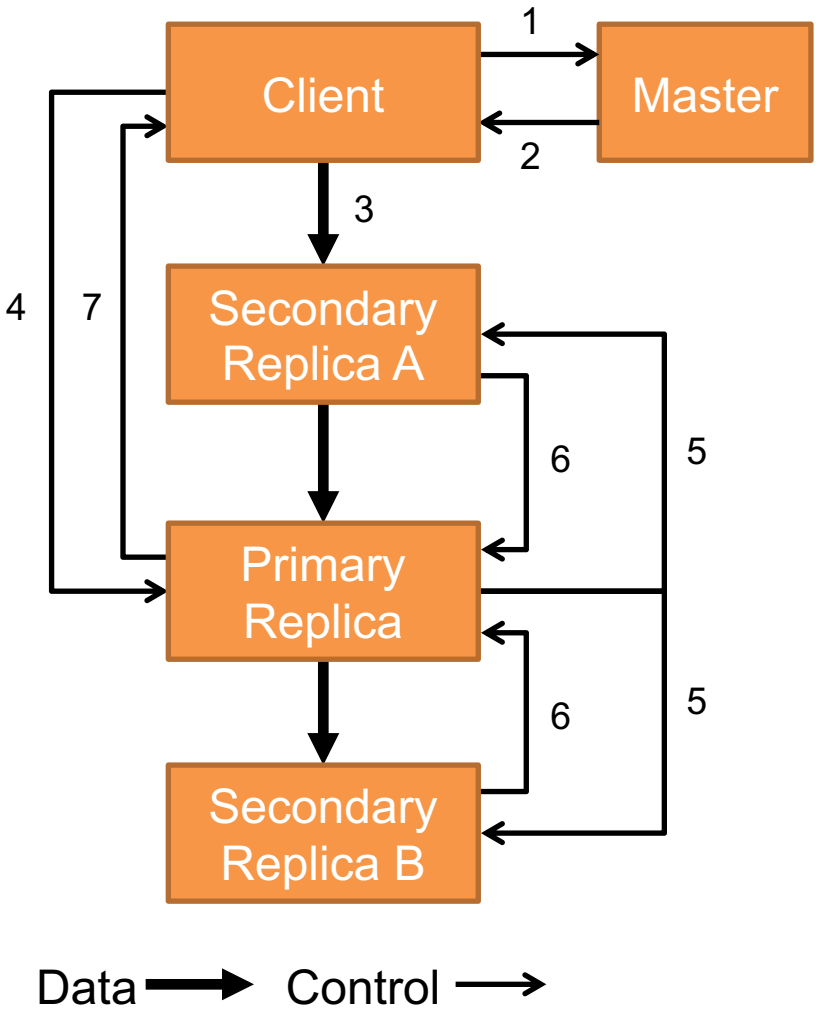
GFS Leases and Mutation Order

- A mutation (write/append) is performed at all the chunk‘s replicas
- Master grants chunk lease to one replica to ensure consistent mutation order across all other replicas
  - Replica with lease is called primary
- Primary chooses serial order for mutations
  - All other replicas follow this order

→ Global mutation order is defined by

1. Lease grant order chosen by the master
2. Serial order chosen by primary within the lease

GFS Control/Data Flow for Writes/Appends

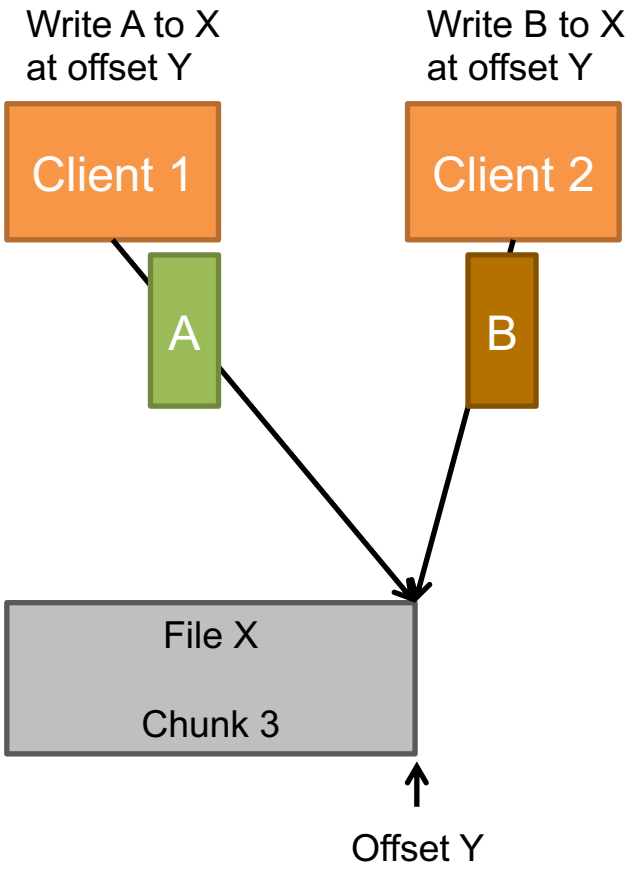


1. Request primary, list of replicas from master
2. Master‘s response
3. Data transfer to arbitrary replica server
4. Mutation request to primary, primary determines order
5. Replicas apply mutations in primary‘s serial order
6. Ack. to primary
7. Ack. to client

GFS Support for Atomic Appends (1/2)

- Technically, append is writing to a file at specific offset
  - Offset traditionally specified by client
  - Challenging in presence of concurrency
    - Example: Two applications concurrently specify to write record at offset Y

    →One record would be destroyed
  - Applications depend on atomic append for no loss of data



GFS Support for Atomic Appends (2/2)

- Special GFS operation to append data in atomic units
  - Operation guarantees that data is appended *at least once*
  - Append procedure
    1. Client specifies append operation and data (but no offset!)
    2. Primary serializes requests, chooses offsets
    3. Returns offsets to clients after data has been appended
  - Data appended to next chunk if size exceeded otherwise
    → Rest of previous chunk filled with padding bits!
  - Client retries operation if append fails at any replica
    → Appended data may occur more than once in some replicas
    → Replicas are not guaranteed to be bitwise identical!

Overview

- Introduction
- Distributed Storage
  - GFS
  - **HDFS**
  - Bigtable
- Distributed Processing
  - MapReduce
  - Spark
  - Flink

Hadoop Distributed File System (HDFS)

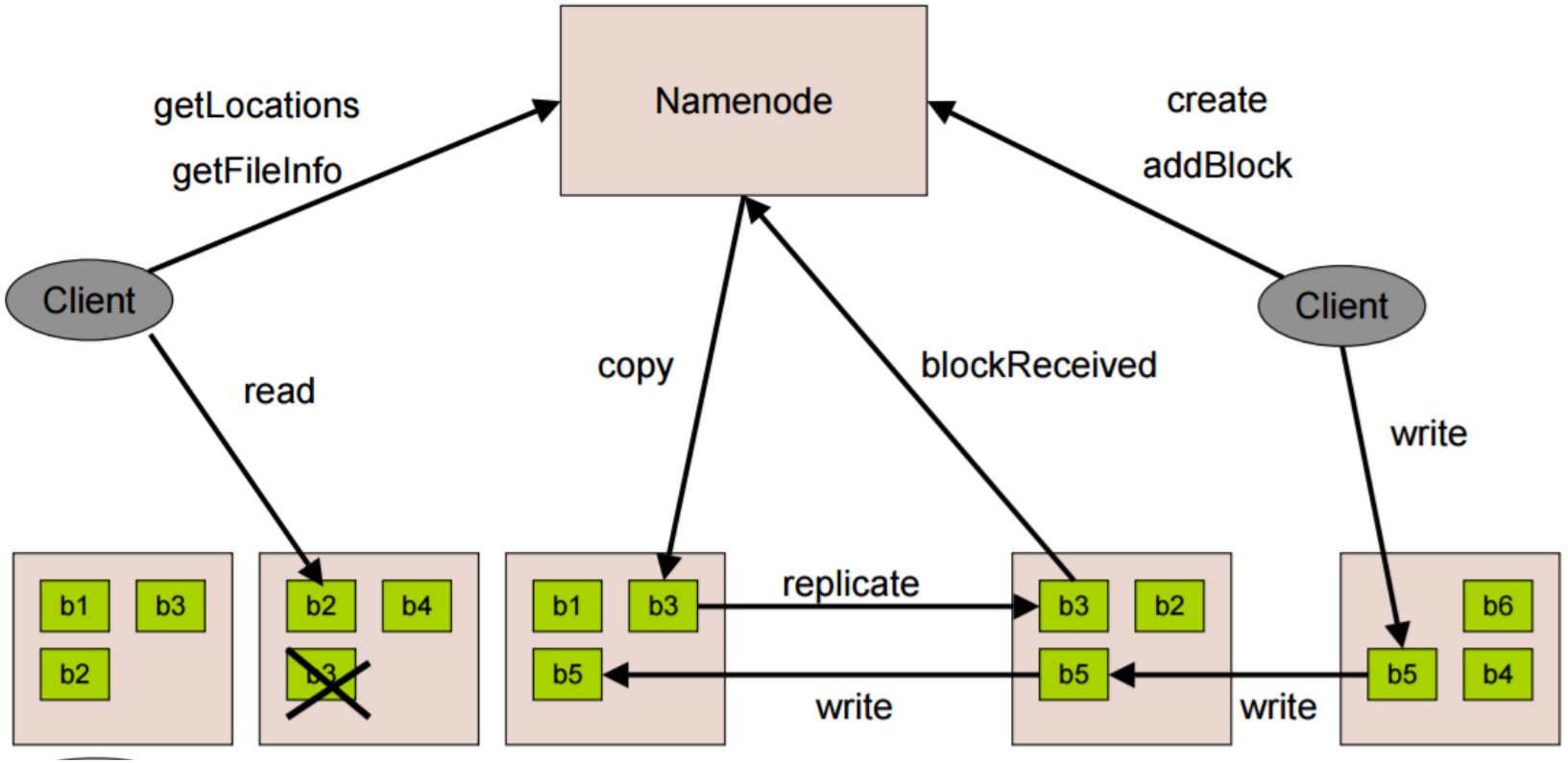
- Distributed filesystem inspired by GFS: All data is stored in blocks, replicated on multiple Data Nodes
- Each node is Linux compute node with hard disks
- One NameNode maintains the file locations, many DataNodes store actual data (1000+)
- Any piece of data is available if at least one data node replica is up and running
- Rack-optimized by default: one replica written locally, second in same rack, third replica in another rack
- Uses large block size, 128 MB is the default, designed for batch processing

Hadoop Distributed File System (HDFS)

- For scalability: Write-once-read-many filesystem
- Applications need to be re-engineered to only do sequential writes

- Example systems working on top of HDFS
  - HBase (Hadoop Database), a database system with only sequential writes, inspired by Google Bigtable
  - MapReduce batch processing system
  - Pig and Hive data mining tools
  - Mahout machine learning libraries
  - Spark and Flink dataflow systems

HDFS Architecture



http://assets.en.oreilly.com/1/event/12/HDFS%20Under%20the%20Hood%20Presentation%201.pdf

Hadoop Distributed File System (HDFS)

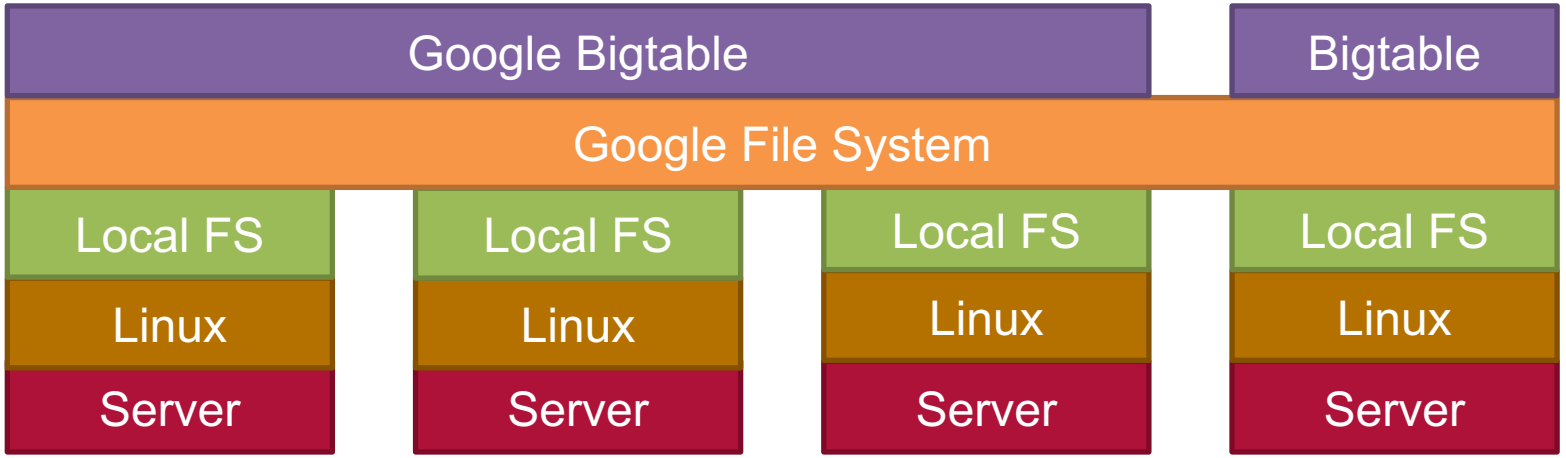
- NameNode is a single computer that maintains the namespace (meta data) of the filesystem
  - Keeps all meta data in memory, writes logs, and does periodic snapshots to the disk
- Rules
  - Replica writes are done in a pipelined fashion to maximize network utilization
  - Reads are done from the nearest replica
  - Replication and block placement: file's replication factor can be changed dynamically (default 3)
  - Block placement is rack-aware (by default, when racks are configured)

Overview

- Introduction
- Distributed Storage
  - GFS
  - HDFS
  - **Bigtable**
- Distributed Processing
  - MapReduce
  - Spark
  - Flink

Google Bigtable[5]

- GFS offers tremendous storage capacity, but limited support to efficiently retrieve structured data (== no indices!)
- Solution: Google Bigtable
  - High-throughput NoSQL store (wide column store), through multi-dimensional sorted map on top of GFS
  - Many use cases at Google, e.g. Web indexes, Gmail, Maps
  - Paper regarded as majorly influential (SIGOPS Hall of Fame)
  - Open source clones: Hbase on HDFS and Cassandra

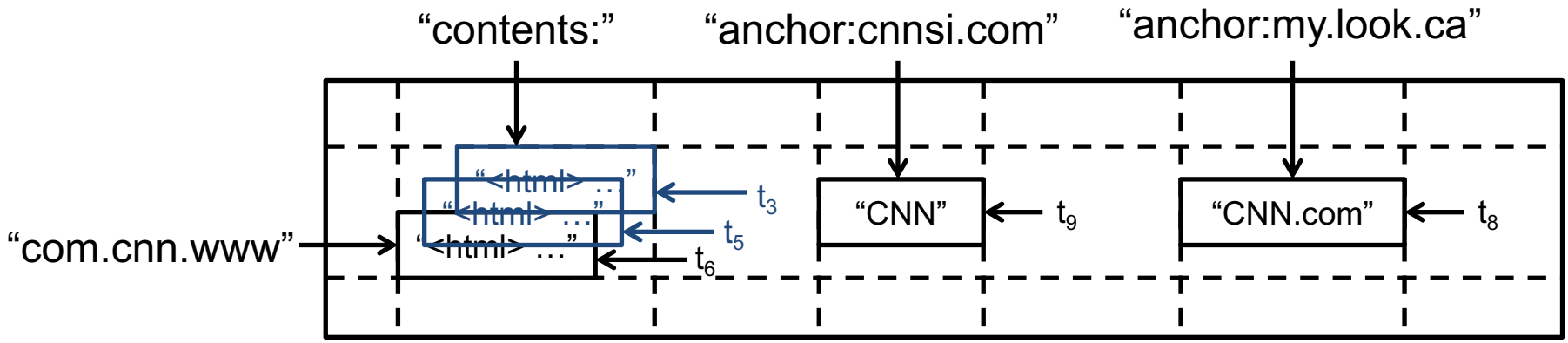


Google Bigtable

- Sequential writes are much faster than random writes
- Data is stored sorted by the row key, and the data is automatically sharded (split to different servers) in large blocks sorted by the row key
  - Allowing for efficient scans in increasing alphabetical row key order
- Columns are grouped in “Column families” and all columns in a single column family are stored together in compressed form on disk
  - Some queries might not access all columns → column families can be easily skipped
- Timestamp field keeps track of versions, e.g. Website content changes over time

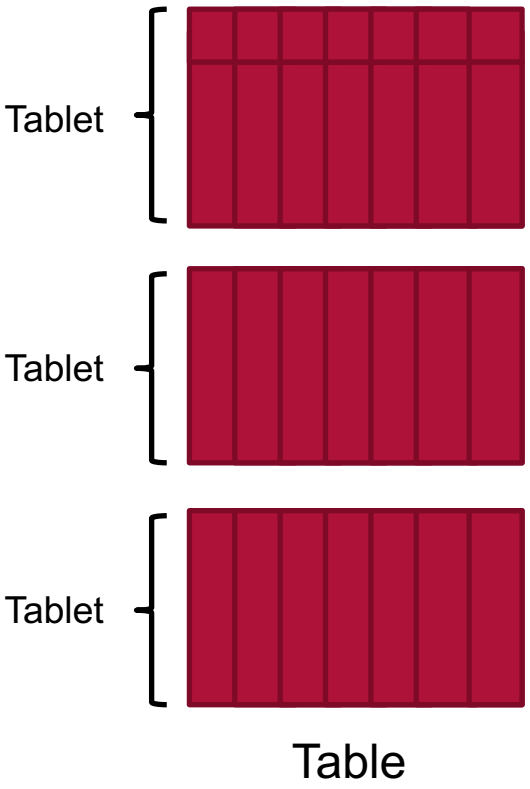
Bigtable Data Model

- (row:string, column:string, time:int64) → string
- No schema, each row can have arbitrary columns
- Columns are grouped to column families
- Example:
  - Row key = reverse of URL to group pages from the same site
  - “Contents” column family stores Web site contents (certain version)
  - “Anchor” column family stores links pointing to the Web page
  - Easy to skip for example reading the Web page contents if we are only interested in links between Web pages



Bigtable Terminology

- Bigtable clusters store several tables
- Rows with consecutive row keys are grouped together to “tablets” = unit of distribution and load balancing
- A table consists of a set of *tablets*
  - Tablet contains all data associated with a row range
  - Initially, each table has one tablet
  - When tablet grows, it is automatically split into several tablets
    - Avg. tablet size is 100-200 MB
    - Servers typically hold about 100 tablets



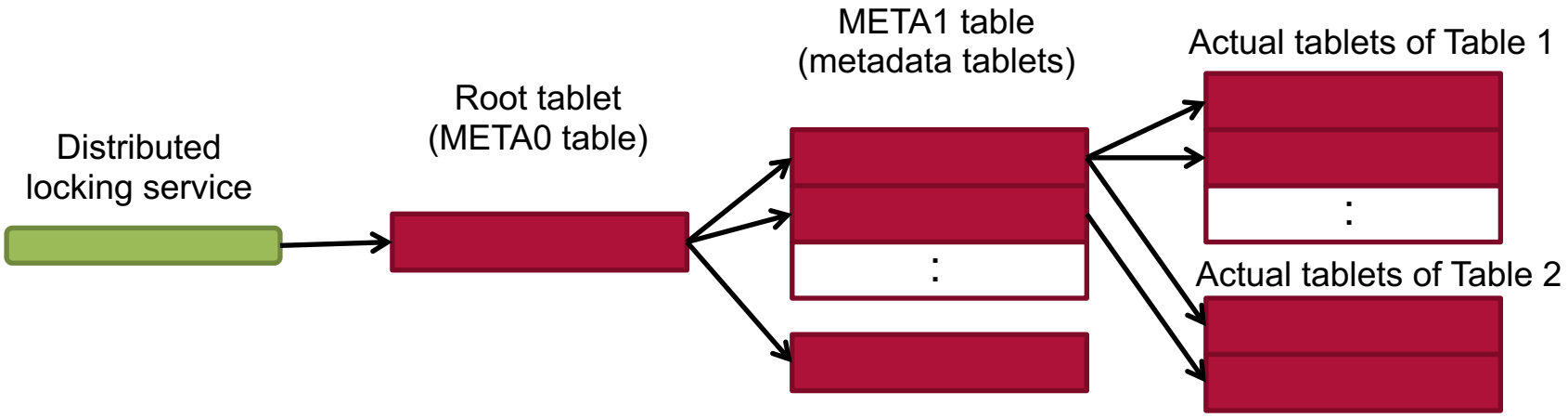
Bigtable Architecture

- Client
  - Sending requests (row, column, time), expecting value
- Master
  - Can manage several tables
  - Assigning tablets to tablet servers
  - Keeps track of addition/expiration of tablet servers
  - Load balancing, garbage collection
- Tablet servers
  - Store the actual tablets
  - Serve client requests
    - Clients rarely communicate with the master

Locating Tablets

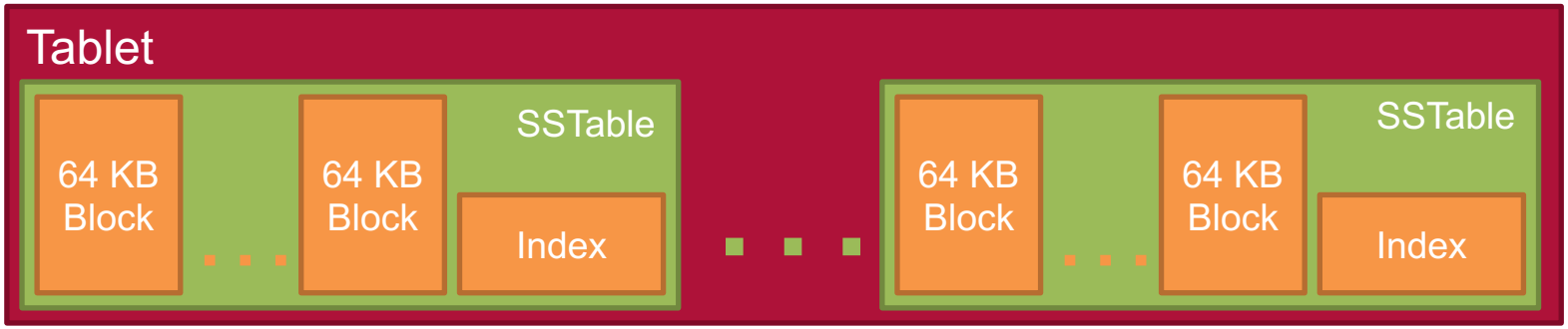
- Tree-level hierarchy to locate correct table
  - Client specifies table and row key
  - First lookup in root tablet (pointer in dist. locking service)
- Information is stored in special METADATA tables
  - Each row in METADATA table consumes 1 KB

  →2^{34} tablets addressable (assuming 128 MB tablet size)



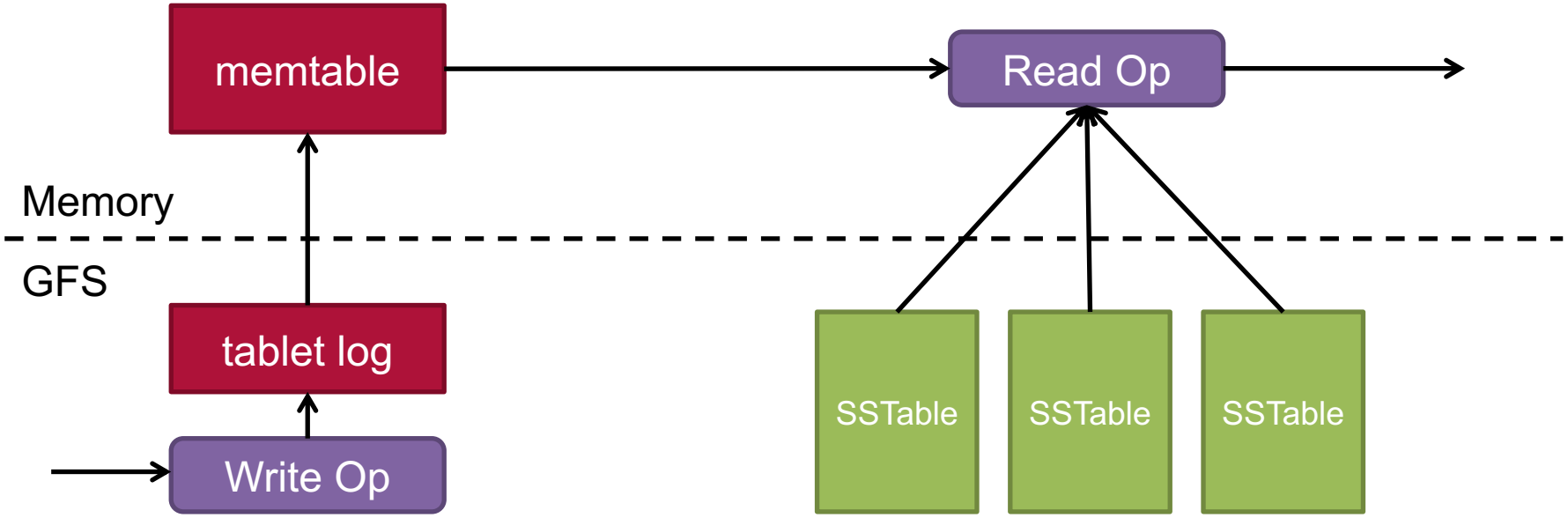
Internal Tablet Structure

- Actual data of a tablet is stored in a structure called SSTable (Sorted String Table)
  - Persistent, ordered, immutable map
  - Each SSTable is a file in GFS
- Each SSTable consists of several blocks and index
  - Each block has 64 KB size
  - Index is loaded in main memory when SSTable is loaded
    - Blocks can be loaded with a single disk seek



Serving Tablets

- Updates are stored to a commit log in GFS
  - When committed they are stored in a *memtable*
  - Older updates are stored in a sequence of SSTables
- Reads are served by merging memtable and SSTables



Compaction

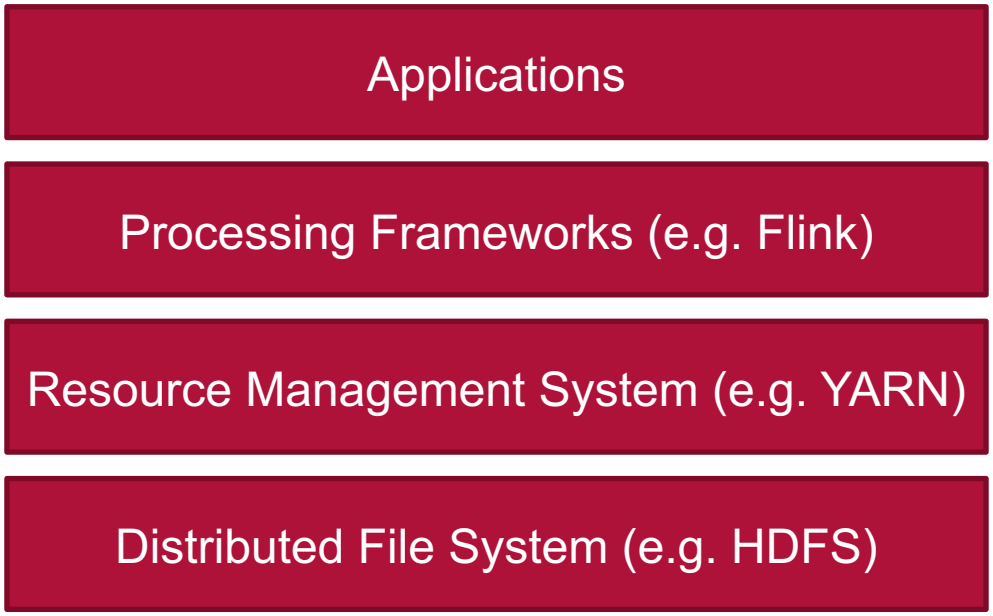
- Minor compaction: Convert memtable into an SSTable
  - Reduces memory usage
  - Reduces reconstruction effort of memtable on restart
- Merging compaction
  - Reduces number of SSTables
  - Happens periodically in the background
- Major compaction
  - Merging compaction that results in only one SSTable
  - Special case of merging compaction that purges deleted data

Overview

- Introduction
- Distributed Storage
  - GFS
  - HDFS
  - Bigtable
- **Distributed Processing**
  - MapReduce
  - Spark
  - Flink

RECAP: Analytics Cluster Setup

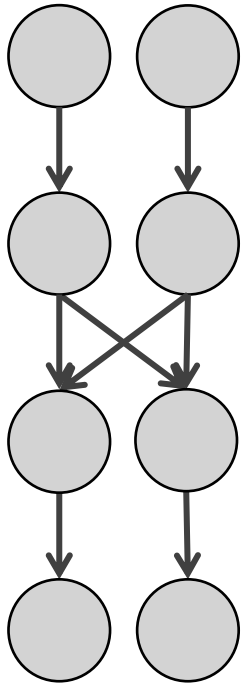
- Analytics clusters often shared by multiple users and applications
- Shared via resource management and distributed file systems



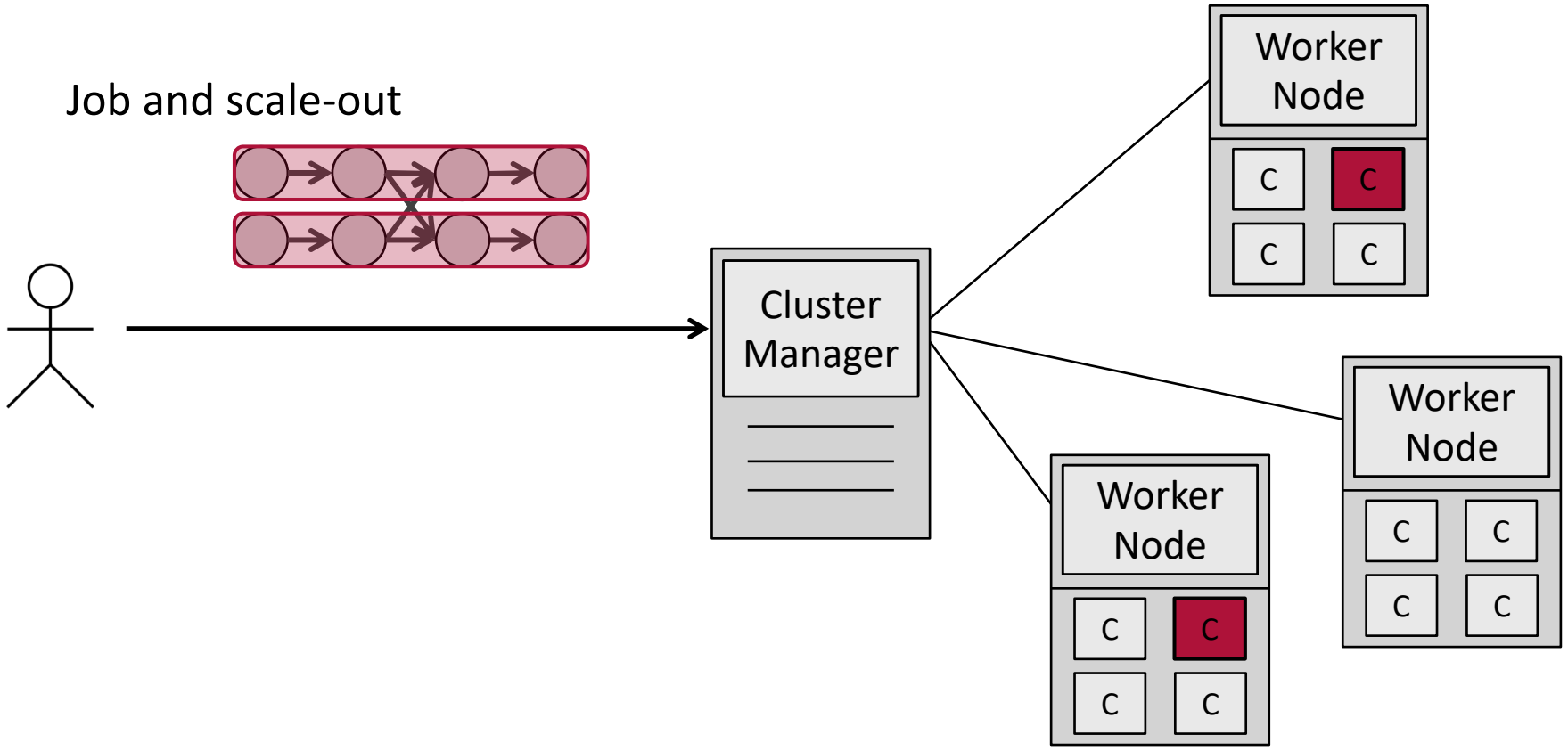
Dataflows Compared to HPC

- High-level programming abstractions
- Comprehensive frameworks and distributed execution engines

- Usable even without extensive knowledge of parallel and distributed programming



Distributed Execution on Commodity Nodes



Overview

- Introduction
- Distributed Storage
  - GFS
  - HDFS
  - Bigtable
- Distributed Processing
  - **MapReduce**
  - Spark
  - Flink

MapReduce[2]

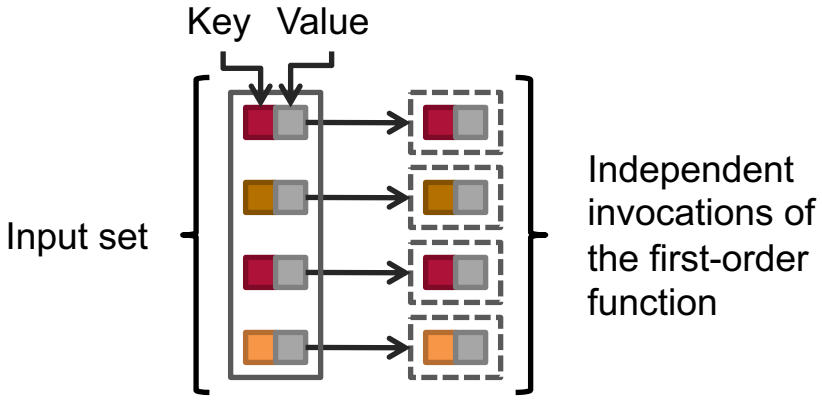
- System published by Google in 2004
  - Introduces MapReduce programming model
  - Illustrates how model helps to meet the discussed requirements (sequential code, fault tolerance, …)
- Google used MapReduce for various things
  - Process crawled documents, logs
  - Computing inverted indices
  - …
- Publication has spawned significant research activities
  - Paper is among most cited research paper in last 15 years
- System has been replaced by successors by now

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 specify how to build KV pairs from input data

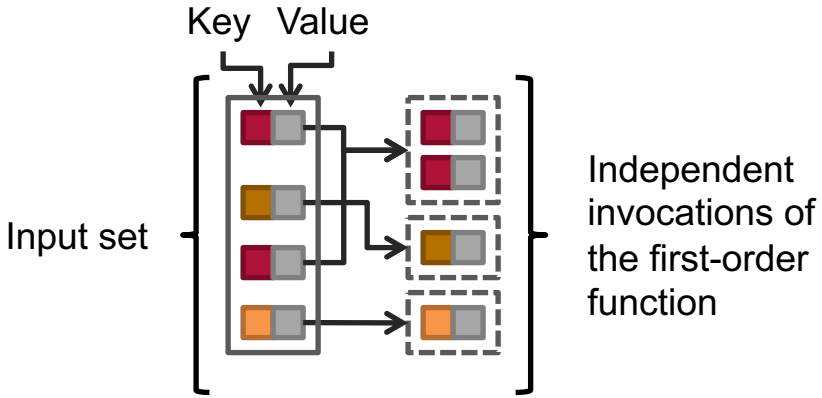
The Map Function

- Signature: $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$
- 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, …

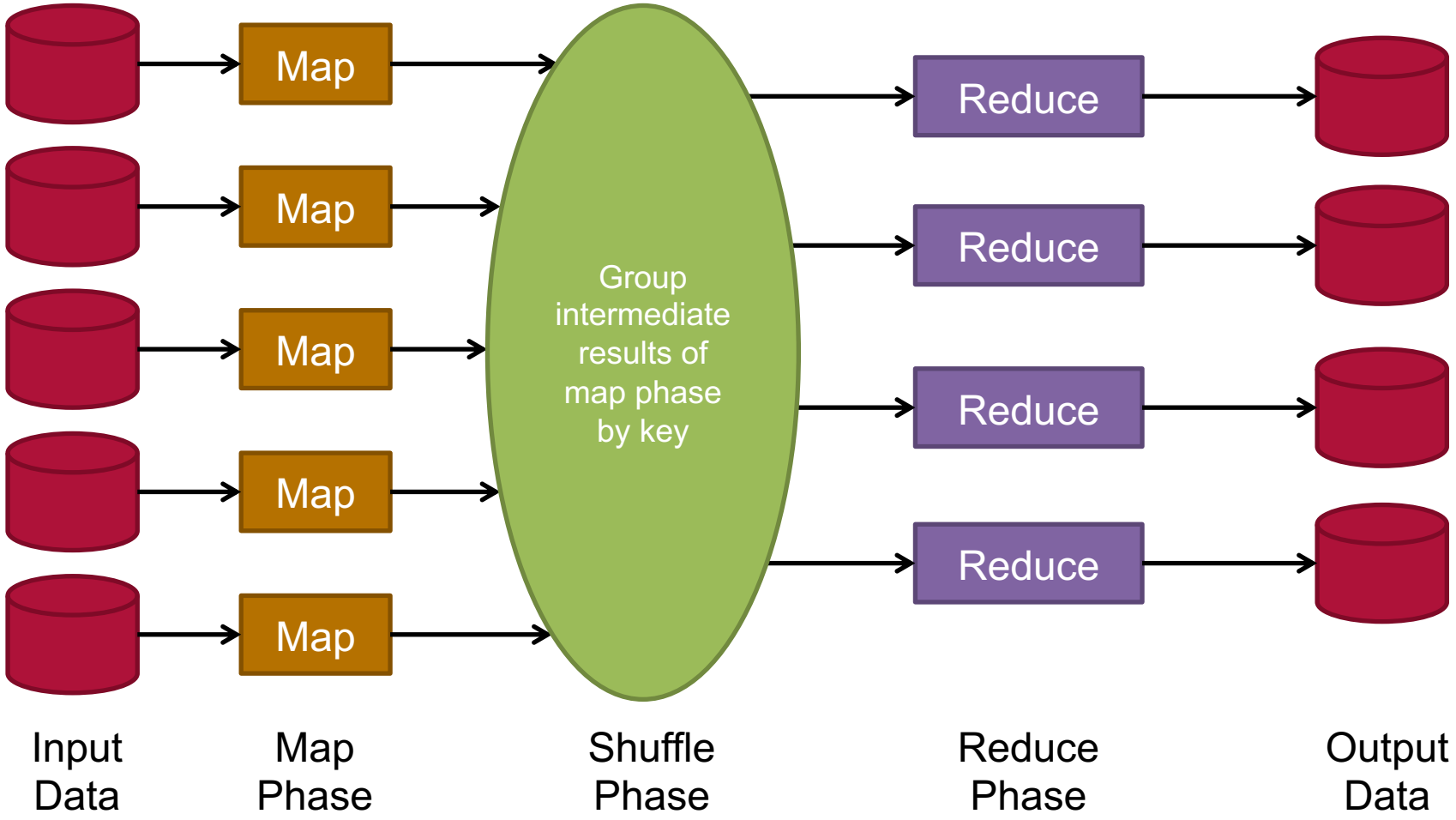


The Reduce Function

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

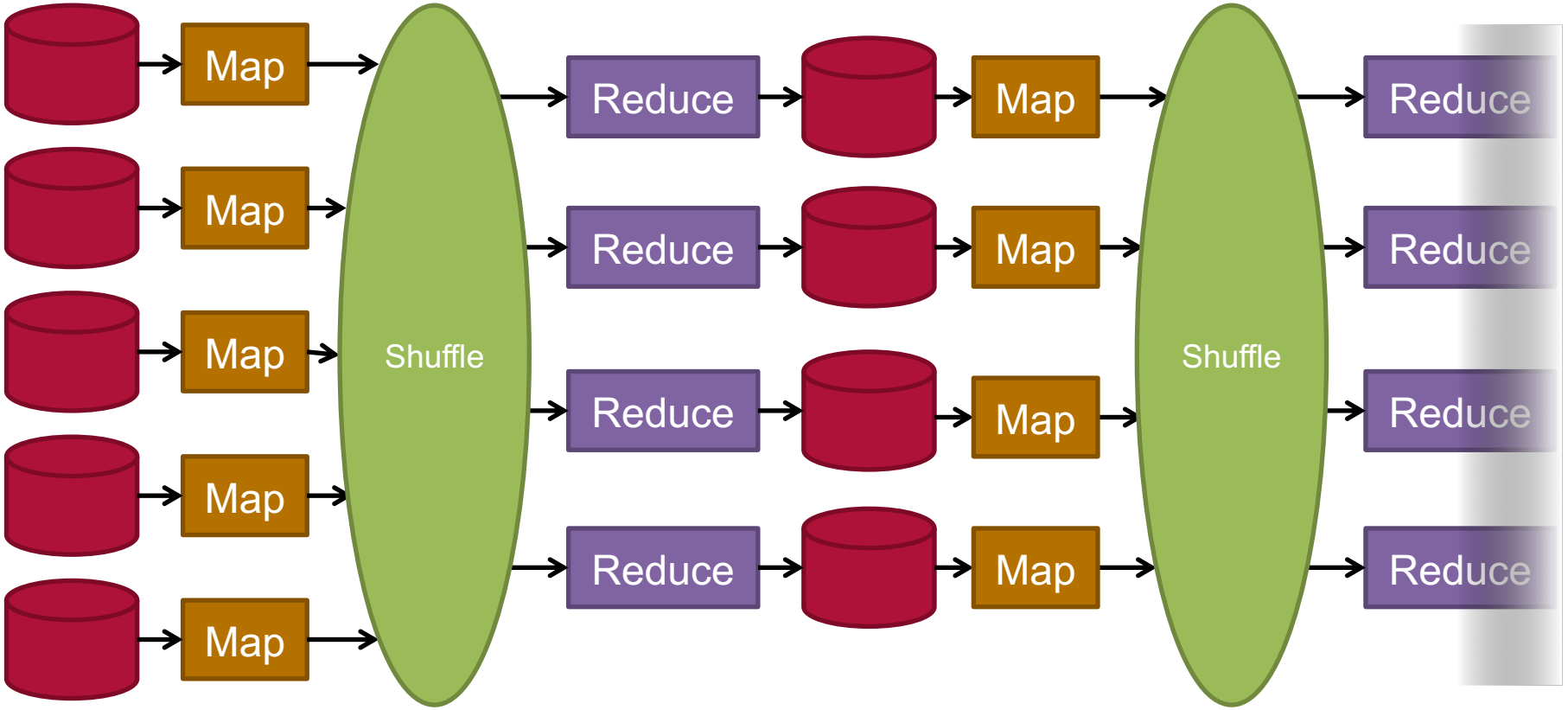
```
Reduce(String word, Iterator partialCounts) {
    int result = 0;
    for each v in partialCounts {
        result +=v;
    }
    Emit(word, result);
}
```

- 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>

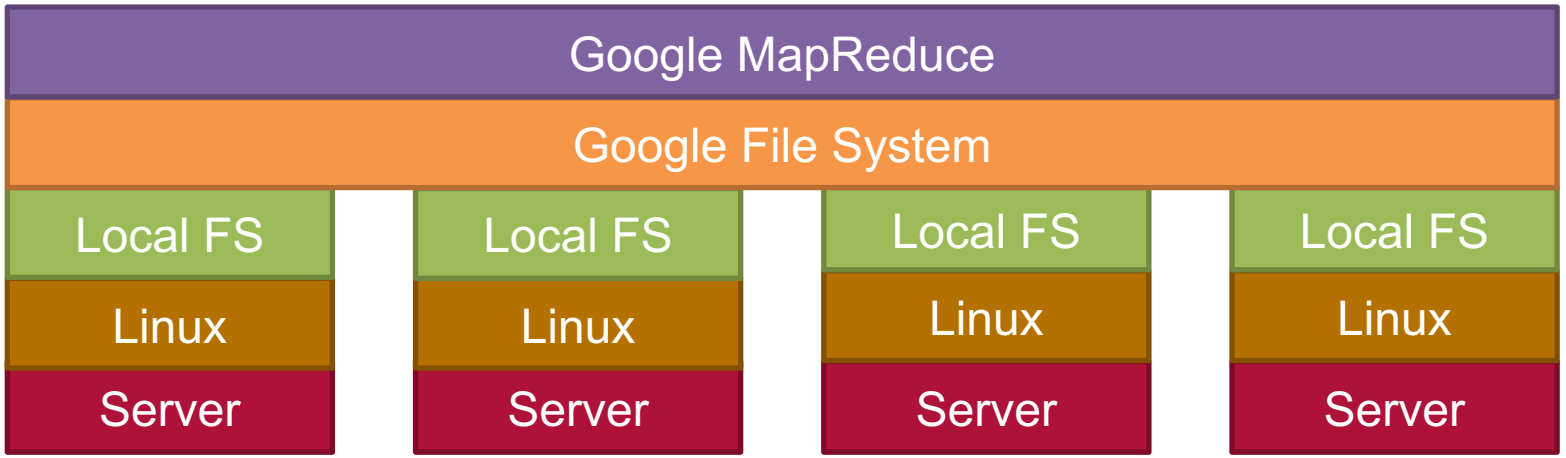
Combining Multiple MapReduce Programs

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



MapReduce Implementation (1/2)

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



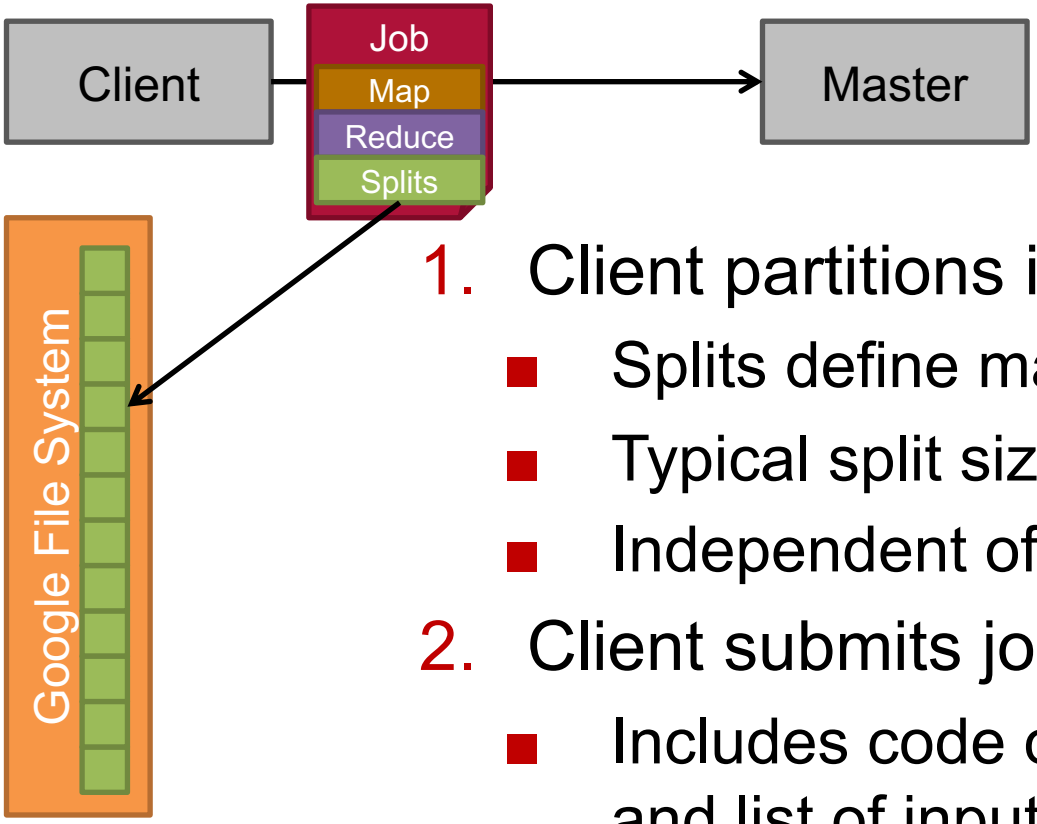
MapReduce Implementation (2/2)

- MapReduce follows master-worker pattern
- Master
  - Job scheduling
  - Load balancing
  - Monitoring of worker nodes and failure detection
- Workers
  - Execution of map and reduce functions
  - Reading and storing of data
  - Periodically reporting availability to master node

Some MapReduce Terminology

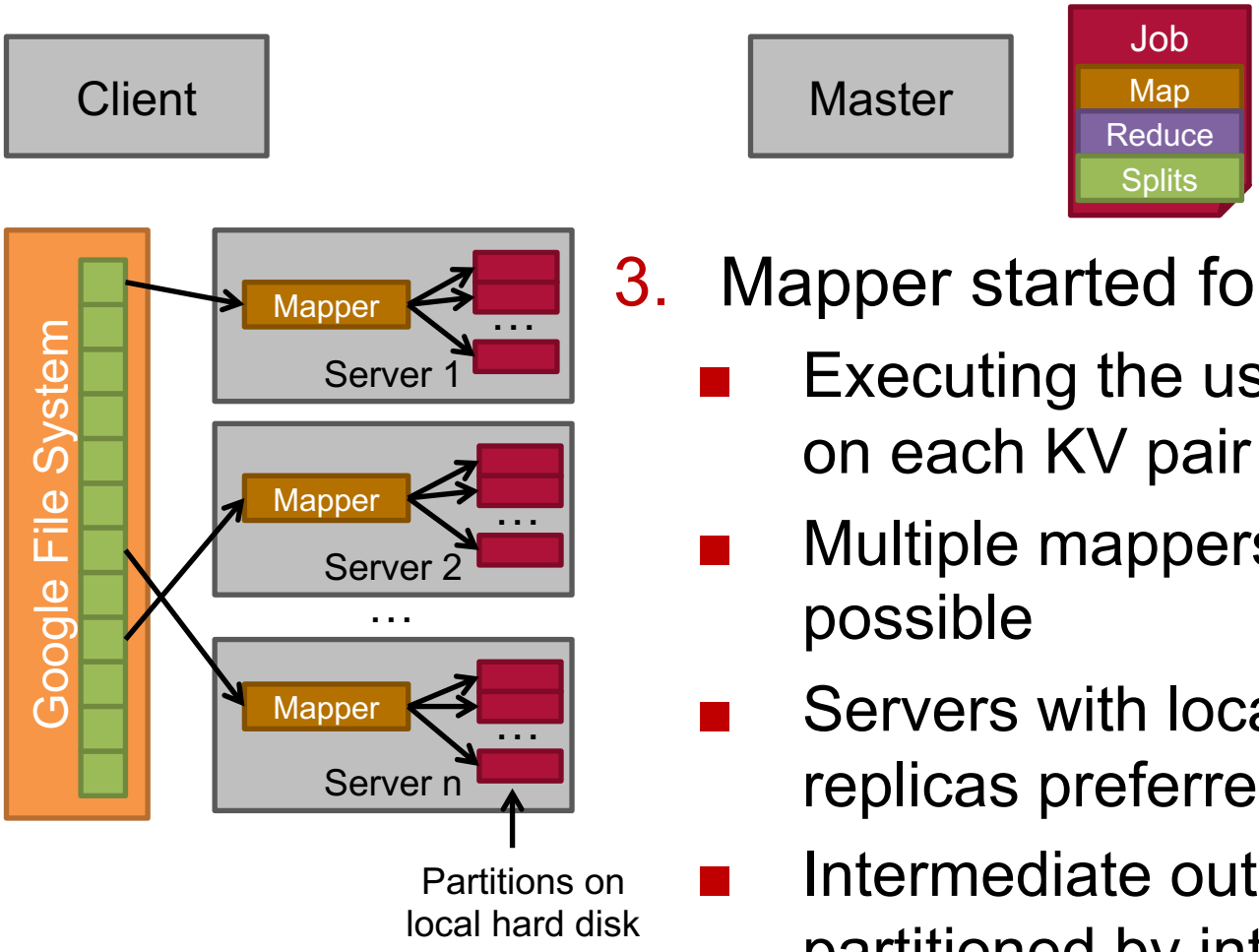
- Map function
  - First-order function provided by user
  - Specifies what happens to the data in job‘s map phase
- Mapper
  - Worker process invoking map function on each KV pair
- Reduce function
  - First-order function provided by user
  - Specifies what happens to the data in job‘s reduce phase
- Reducer
  - Worker process invoking reduce function on KV pair groups

Distributed Execution of MapReduce Program (1/3)



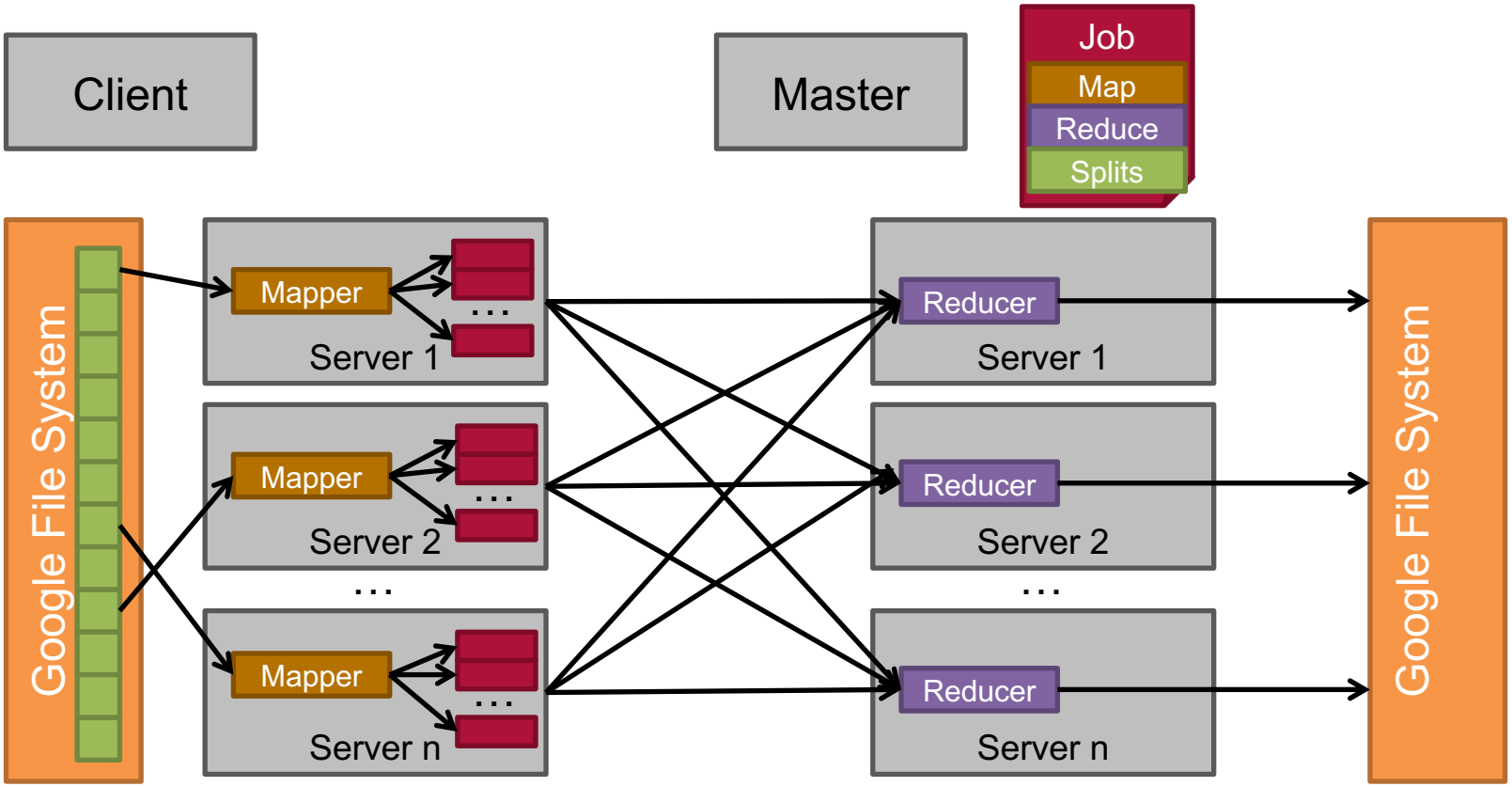
1. Client partitions input file into input splits
   - Splits define maximal scale-out
   - Typical split size is 16-64 MB
   - Independent of GFS block size
2. Client submits job to master
   - Includes code of map/reduce functions and list of input splits (file boundaries)
   - Master tries to find 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 on each KV pair in input split
   - Multiple mappers per server possible
   - Servers with locally available replicas preferred (data locality)
   - 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, before partitions go to disk
  - 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, continues with reading 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 re-compute lost intermediate results

Apache Hadoop

- Open source implementation of Google's MapReduce
  - Written in Java
  - Many prominent users, e.g. Yahoo, Facebook, Twitter

- Also includes HDFS and YARN, a resource management system
- Today, HDFS and YARN are more relevant than MapReduce as they are used with MR successors
- Hadoop still basis of many commercial distributions
  - e.g. Cloudera, Hortonworks, …

MapReduce Limitations

- MapReduce is powerful but has two major limitations
  1. Assumes finite input (batch processing only)
  2. Data between stages and jobs goes to file system for fault tolerance

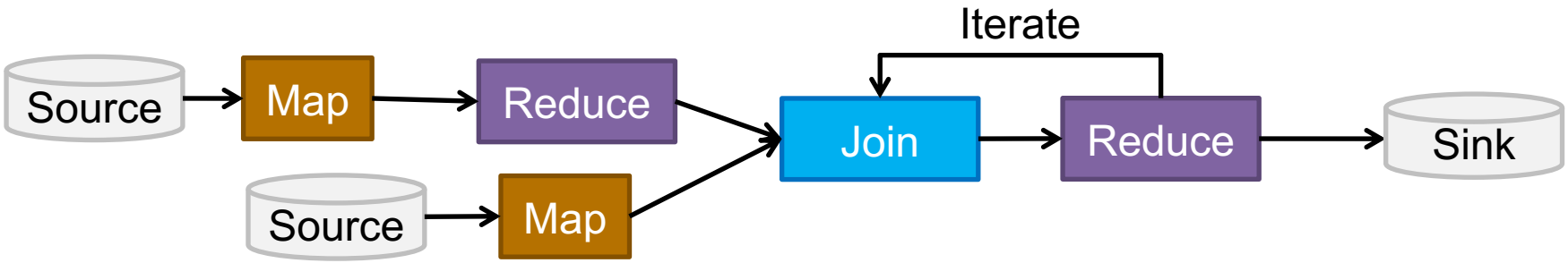
- Limitation of finite input prevents streaming processing
  - Difficult to respond to real-time events without large delays, e.g. click streams, IT systems monitoring …
- Constraint to write to GFS especially costly for iterative algorithms
  - e.g. graph processing, machine learning, …

Successors of MapReduce

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

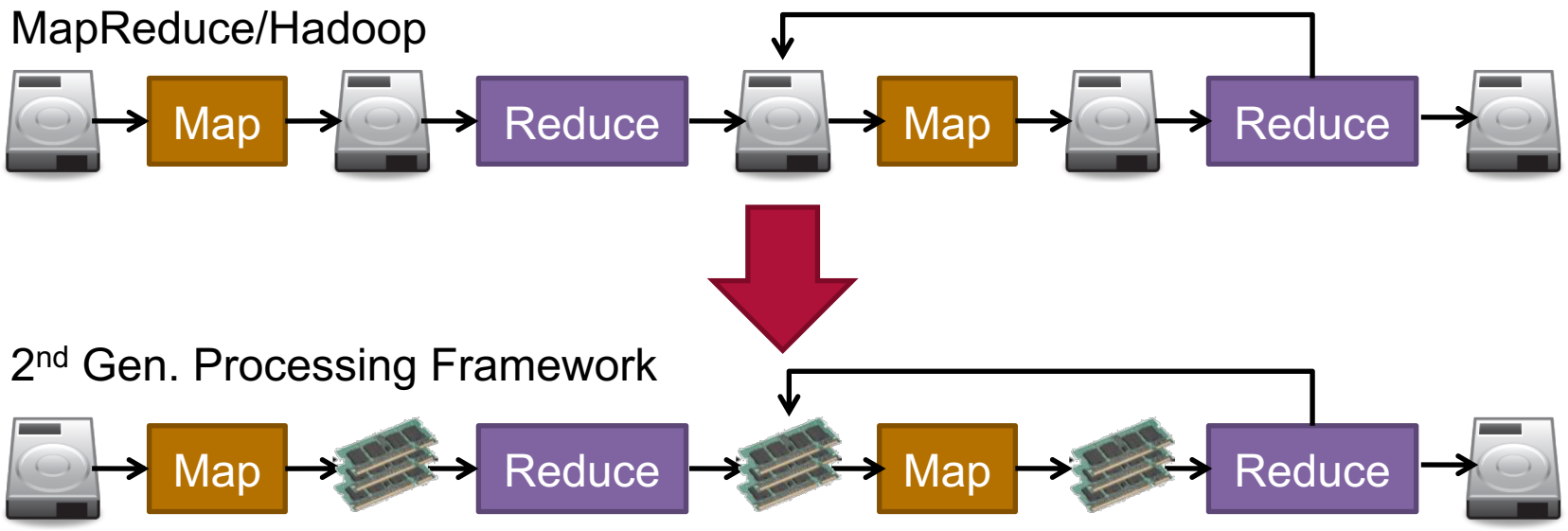
MapReduce Evolutions (1/3)

- Improved programming model with rich operator and feature set
  - Joins, groupBys, filters, iterations, and windows besides map and reduce operators
- This simplifies a wide array of computations, including:
  - Relational batch jobs
  - Iterative machine learning and graph processing
  - Continuous stream processing



MapReduce Evolutions (2/3)

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



MapReduce Evolutions (3/3)

- Generalized processing engine that handles streams
  - Using a batch engine but really small batches
    - Also called “Microbatching”
    - Creates “illusion” of continuous processing
  - Using a real streaming engine
    - Continuously deployed dataflows rather than multiple batch jobs
    - Batch becomes a special case of streaming
- Stream processing requires re-definition of operators
  - e.g. classic reduce function does not make sense
  - Instead: window operators (e.g. all tuples of last 5 sec.)

Overview

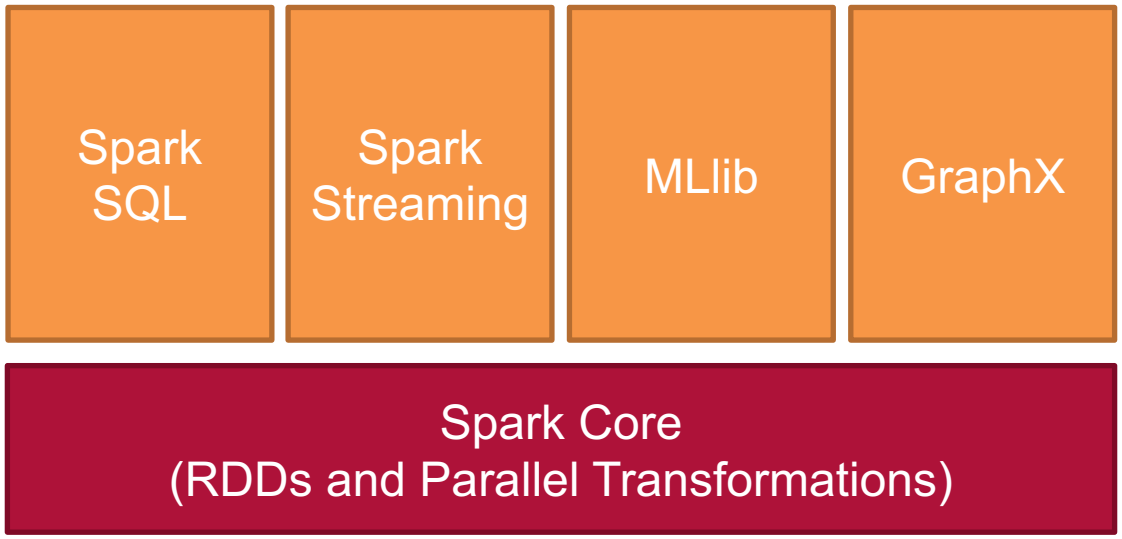
- Introduction
- Distributed Storage
  - GFS
  - HDFS
  - Bigtable
- Distributed Processing
  - MapReduce
  - **Spark**
  - Flink

Apache Spark[7]

- 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

Resilient Distributed Datasets (RDDs) (1/3)

- RDDs are distributed 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

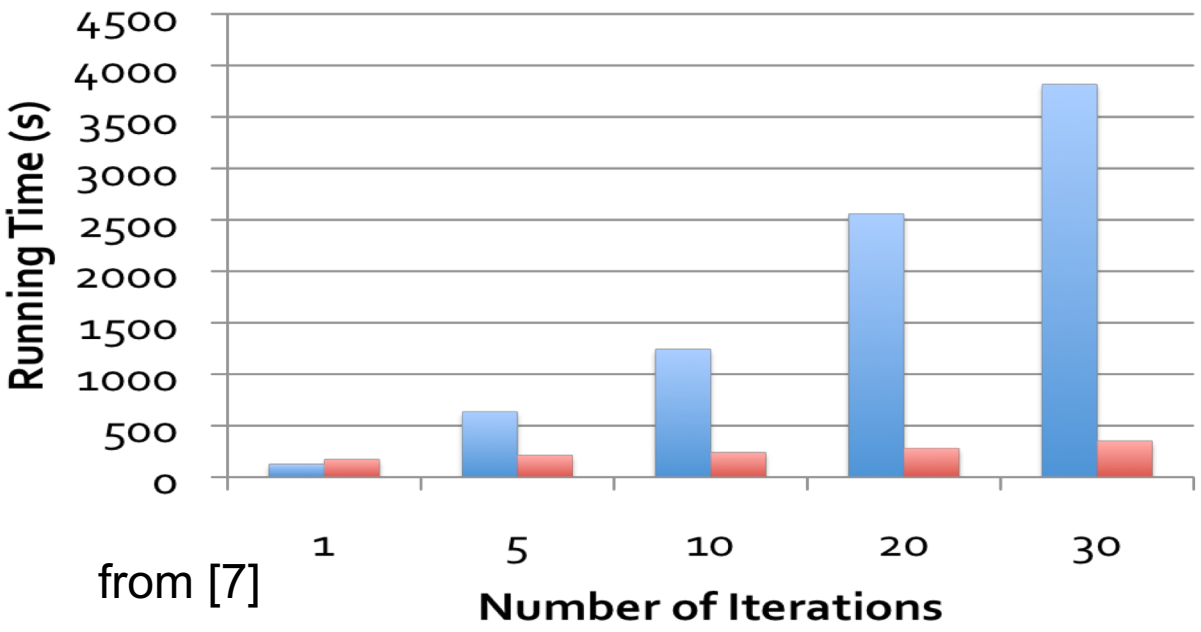
```
val data = spark.textFile(...).map(readPoint).cache()

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

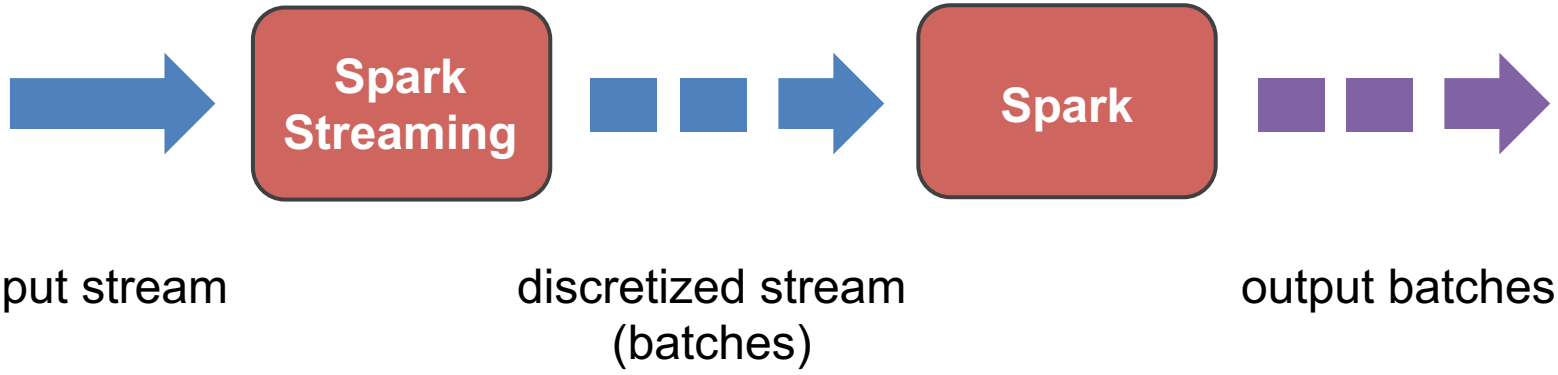


from [7]

- In-memory caching makes subsequent iterations fast

Spark Streaming

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

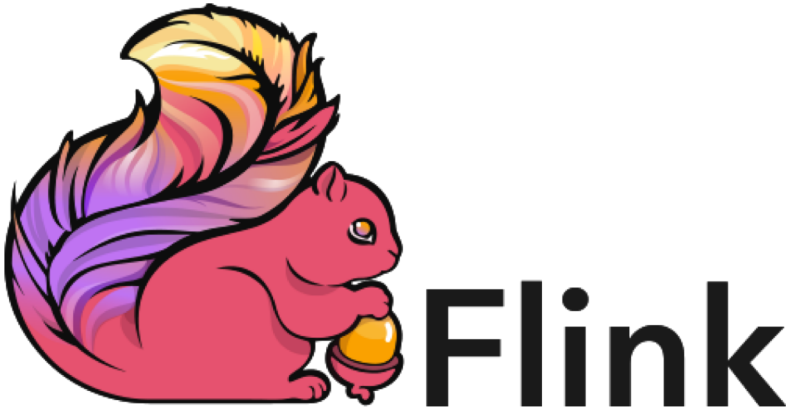


Overview

- Introduction
- Distributed Storage
  - GFS
  - HDFS
  - Bigtable
- Distributed Processing
  - MapReduce
  - Spark
  - **Flink**

Apache Flink[8]

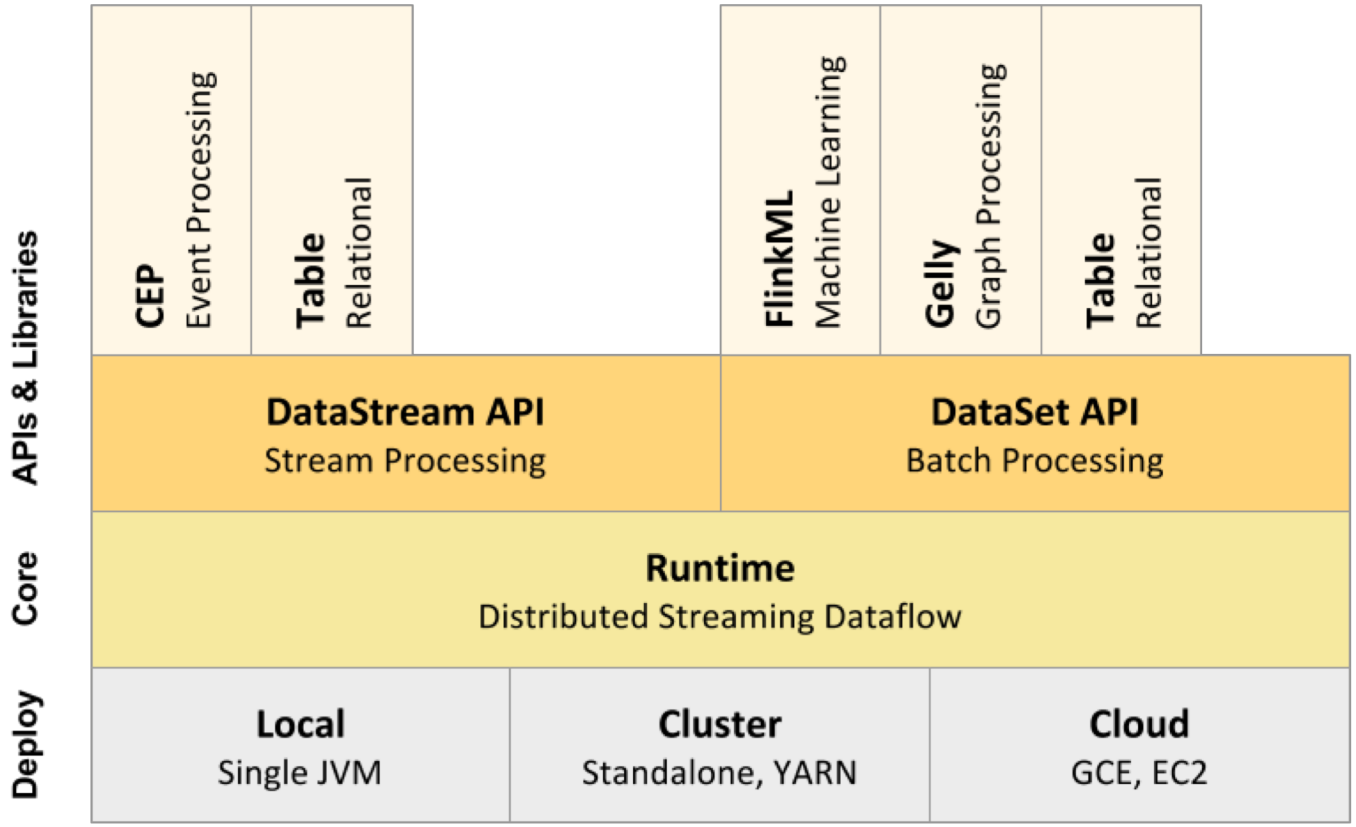
- Open source project that unifies batch and stream processing in one engine



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

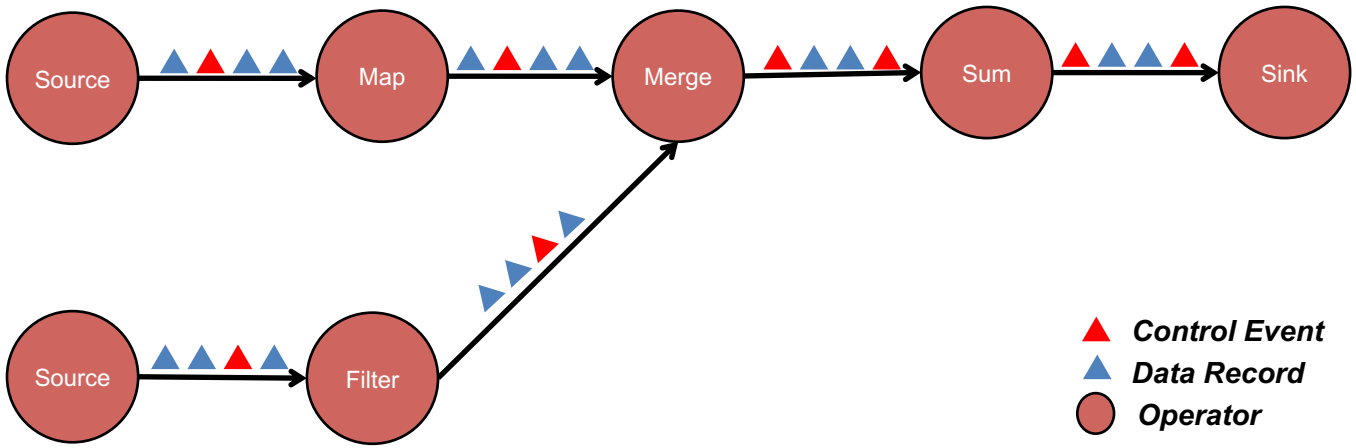
- Apache top-level project since beginning of 2015
  - Well-integrated in big data ecosystem
  - Vivid user and developer community worldwide

Apache Flink Stack



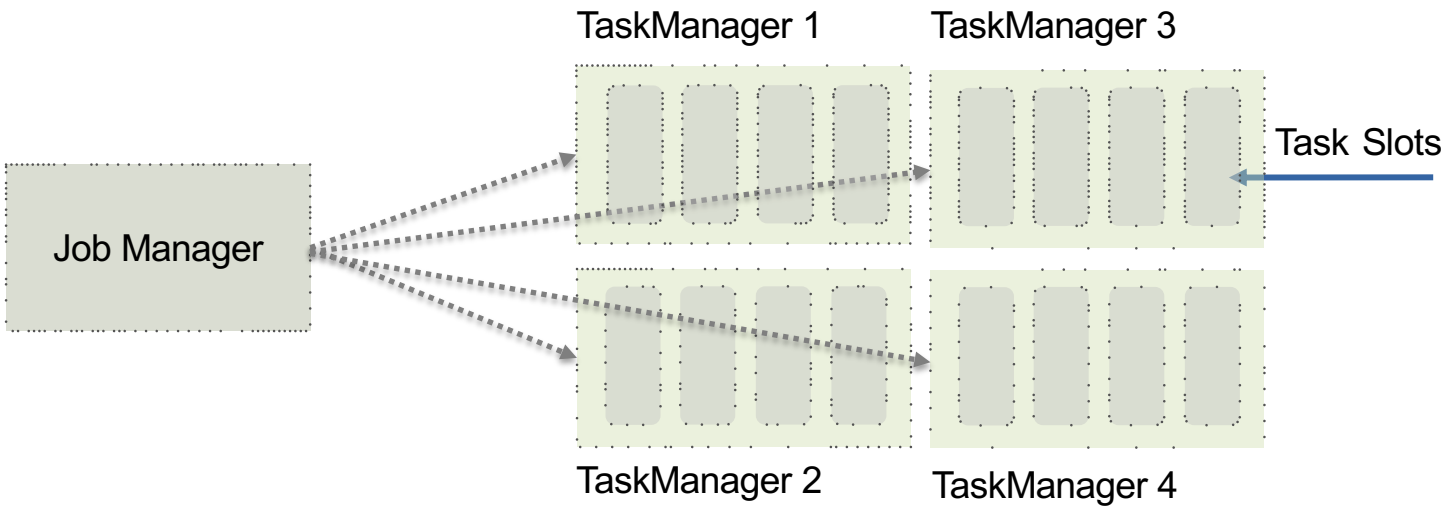
- Flink's core is a streaming dataflow engine that supports both batch and stream processing

Apache Flink Execution Model (1/3)



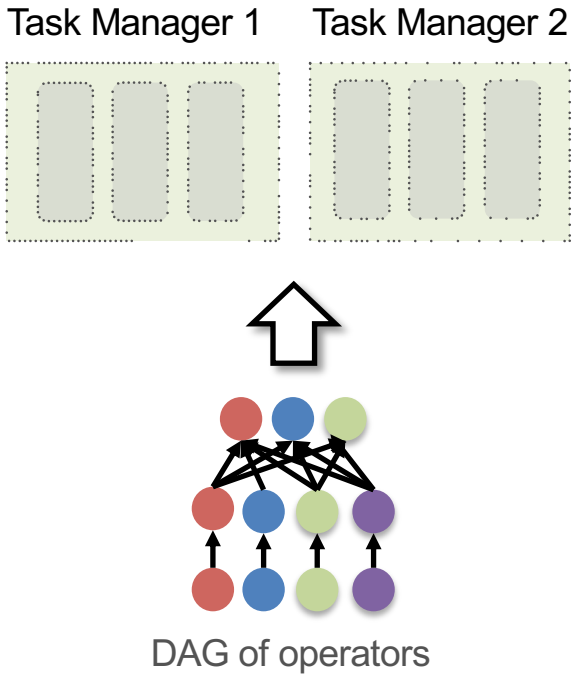
- A job consists of:
  - A directed acyclic graph (DAG) of operators and
  - Intermediate streams of data records and control events flowing through the DAG of operators
- Pipelined/Streaming engine

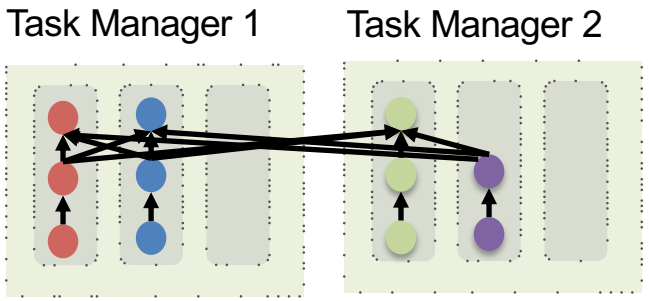
Apache Flink Execution Model (2/3)



- 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)

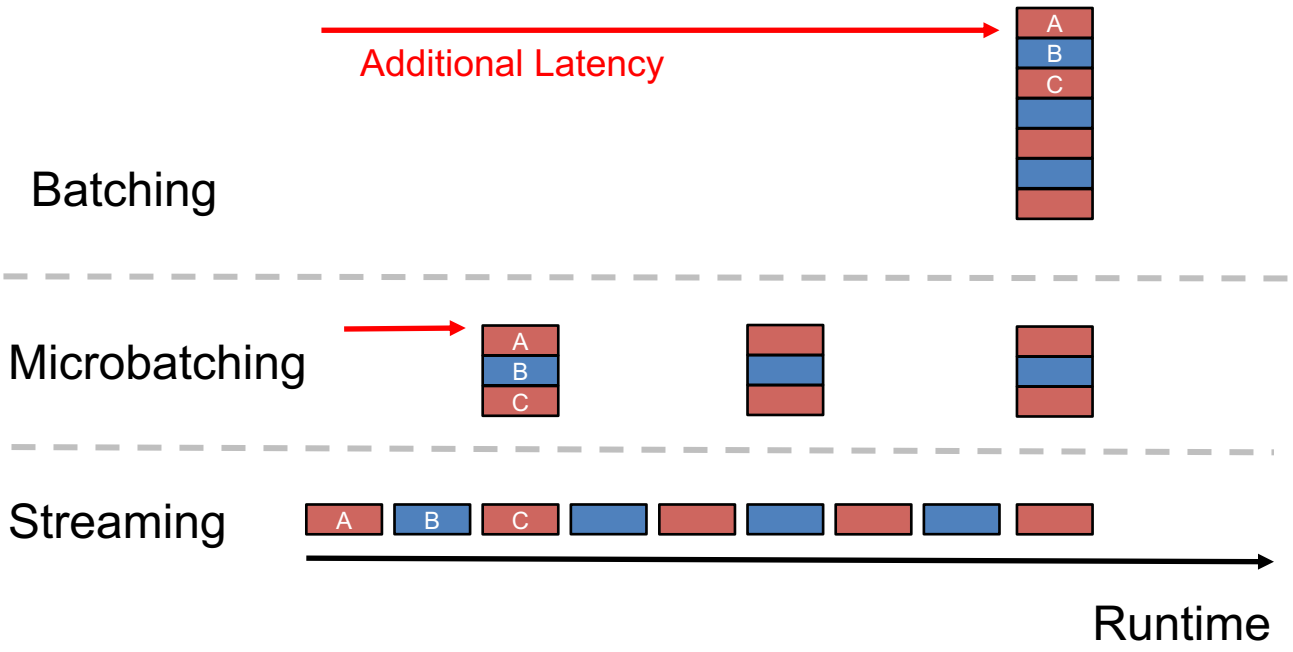




- Deployment example on a cluster with 2 Task Managers with 3 slots each

- 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

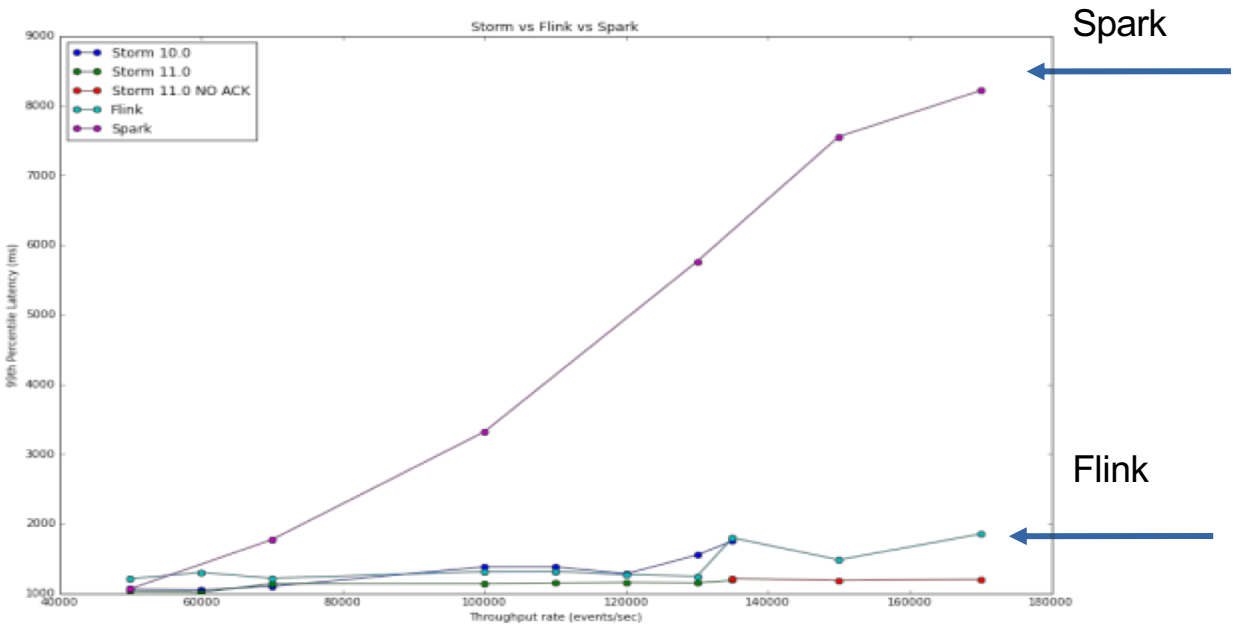
Performance Comparison of Streaming Approaches (1/3)



- Microbatches: Processing of small batches of tuples
- “Real” Streaming: Tuple-wise processing (lower latencies possible, but might cost throughput)

Performance Comparison of Streaming Approaches (2/3)

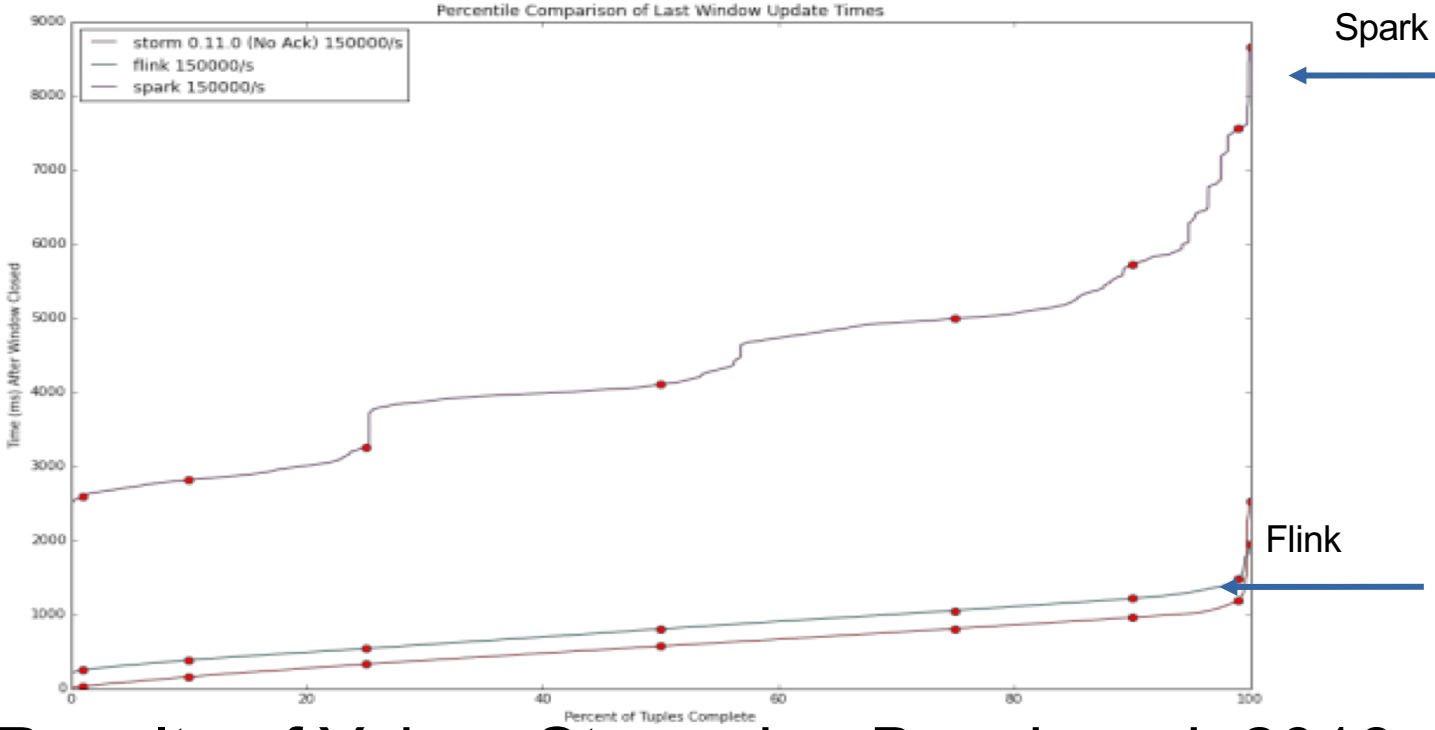
- Flink has lower latency (y-axis)
  - It processes an event as it becomes available



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

Performance Comparison of Streaming Approaches (3/3)

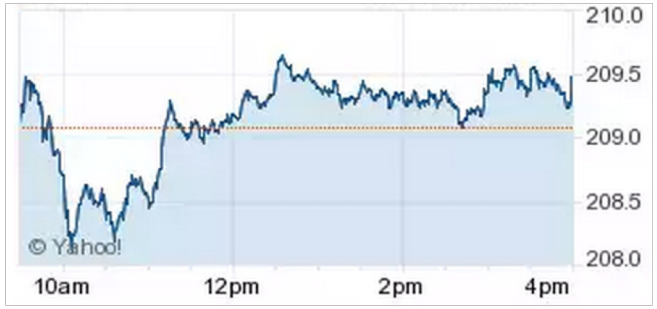
- Spark has higher throughput (y-axis)
  - Less overhead and meta data with microbatches



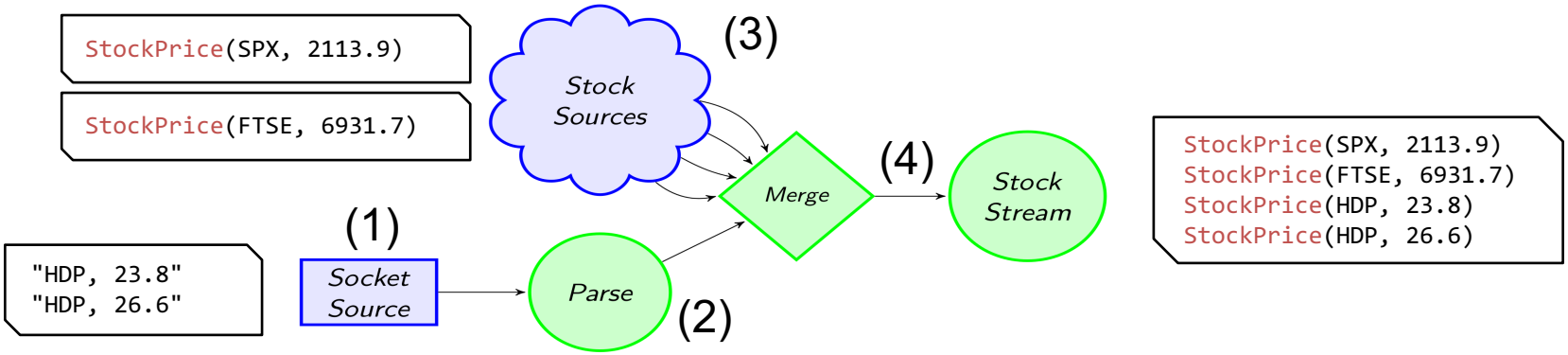
- Results of Yahoo Streaming Benchmark 2016:
  - https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

Flink Streaming: Stock Example

- 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

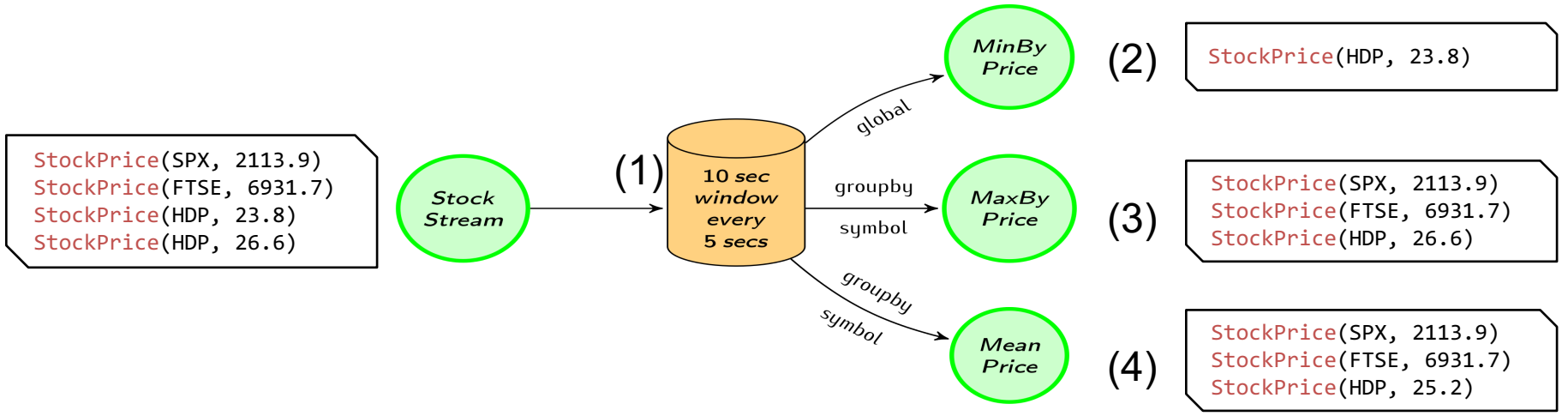


Flink Streaming: Reading from Multiple Inputs



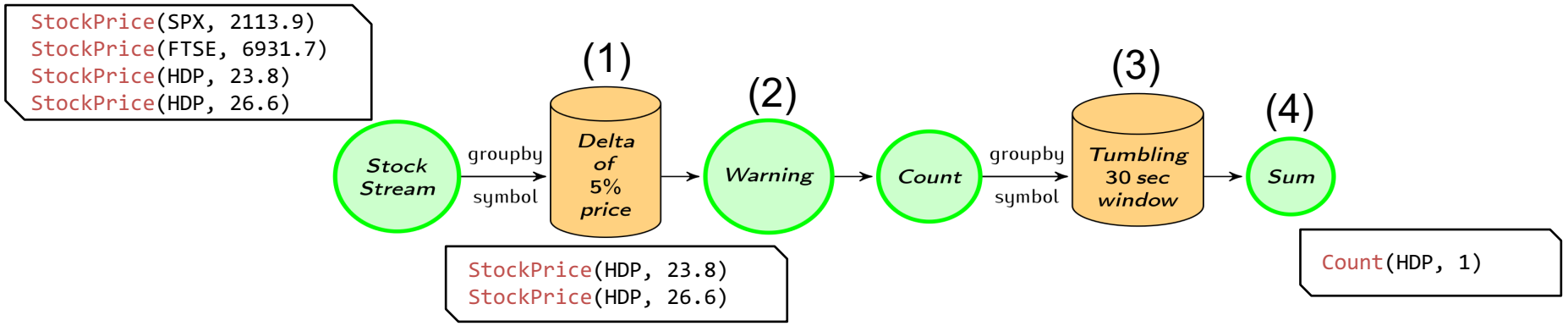

```
case class StockPrice(symbol : String, price : Double)
val env = StreamExecutionEnvironment.getExecutionEnvironment

(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")
```

Overview

- Introduction
- Distributed Storage
  - GFS
  - HDFS
  - Bigtable
- Distributed Processing
  - MapReduce
  - Spark
  - Flink

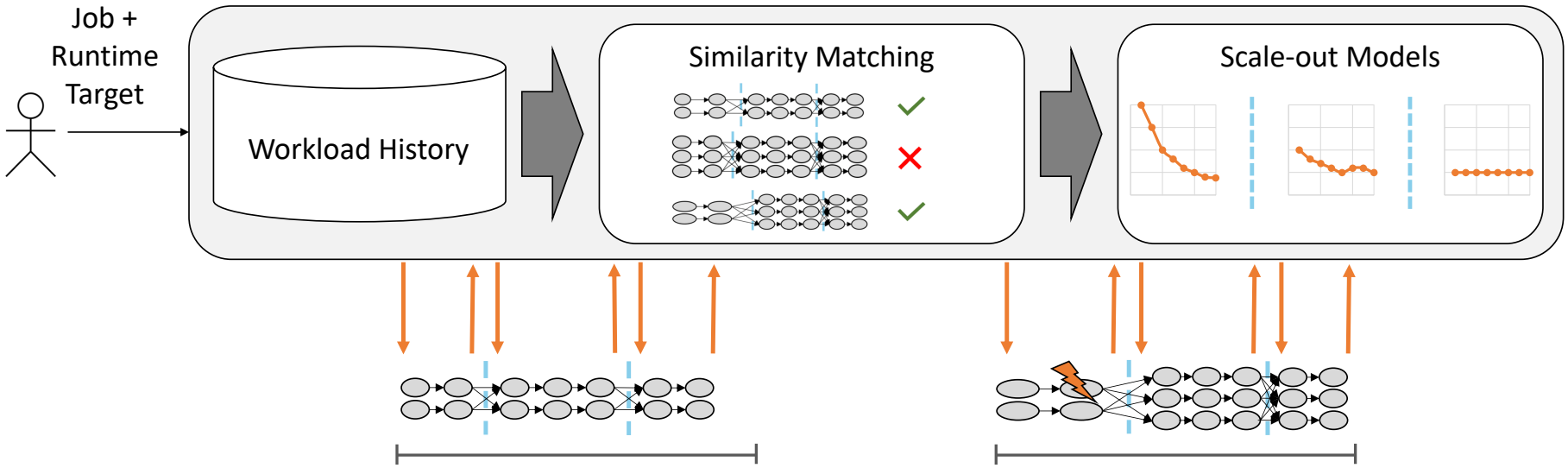
Summary

- Major trends of the last two decades
  - Single processors stopped getting faster as previously
  - Clusters of commodity resources and access via VMs or containers in public clouds
  - More data is recorded and stored
- New distributed systems to utilize cluster resources
  - Distributed storage systems for files and also more structured data
  - Distributed analytics with dataflow systems for search and relational processing, graph analysis, machine learning etc.

Add-on: Current Related Research at CIT

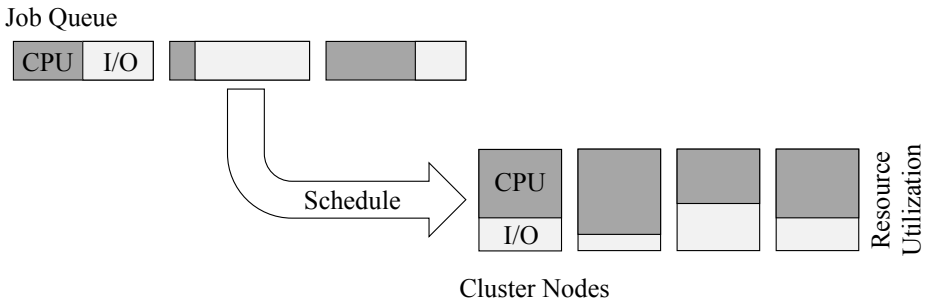
CIT Research: Dynamic Resource Allocation for Runtime Targets[9]

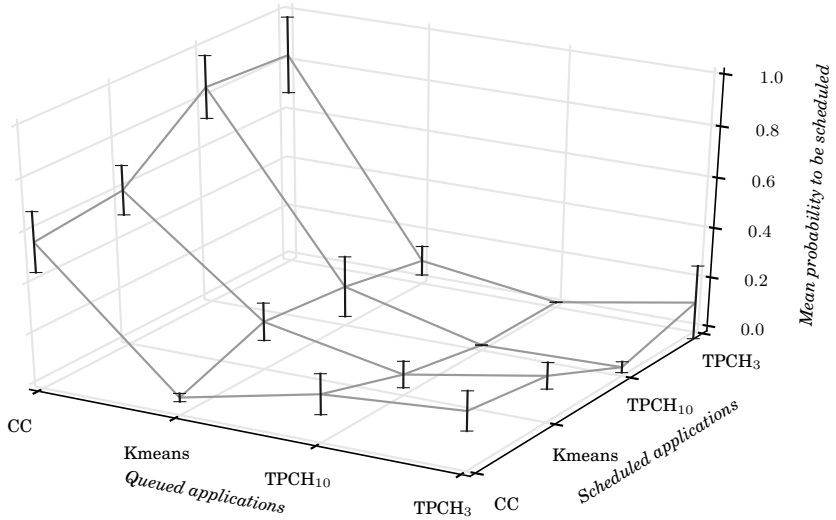
- Automatic resource provisioning and dynamic scaling using scale-out models based on similar previous executions of recurring batch jobs



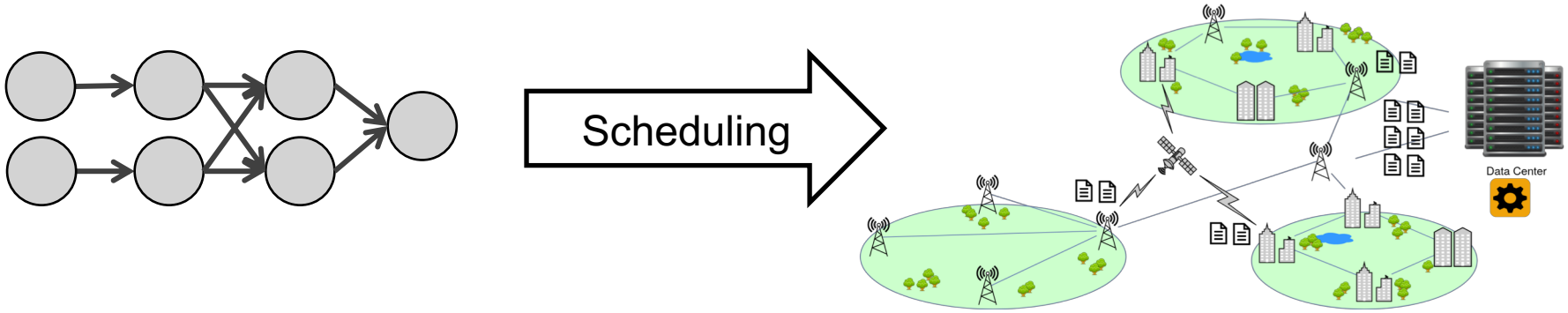
CIT Research: Continuously Improving Job Scheduling[10]

- Using reinforcement learning to learn "co-location goodness" of recurring batch jobs over time and then schedule jobs with complimentary resource needs
  - For resource utilization: CPU, disk, and network
  - For interference: I/O wait





CIT Research: Adaptive Resource Management for Dataflows in IoT [11]



- Heterogeneous, geo-distributed, dynamically changing environments (instead of homogeneous clusters)
- Match streaming jobs (with their varying requirements for computation and communication) to topologies (with heterogenous nodes, large networks, and changing connections, i.e. latency and bandwidth)

Literature and References

- Literature: M. Kleppmann, “Designing Data-Intensive Applications”, 2017, Chapter 10 and 11

- References:

[1] A. S. Das, M. Datar, A. Garg, and S. Rajaram, “Google News Personalization: Scalable Online Collaborative Filtering”, In Proceedings of the 16th International Conference on World Wide Web. WWW ’07. ACM, 2007.

[2] J. Dean, S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Communications of the ACM, 51 (1), 2008.

[3] 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.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google File System”. in Proc. of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP ‘03), ACM, 2003.

[5] F. Chang, J .Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber: “Bigtable: A Distributed Storage System for Structured Data”, ACM Transactions on Computer Systems, 26 (2), 2008

[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.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine”, in IEEE Data Engineering Bulletin iss. 36, no. 4, 2015.

[9] L. Thamsen, “Dynamic Resource Allocation for Distributed Dataflows”, PhD thesis at TU Berlin, 2018.

[10] L. Thamsen, B. Rabier, F. Schmidt, T. Renner, and O. Kao, “Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference”, in Proc. of the 6th IEEE BigData Congress, IEEE, 2017.

[11] G. Janßen, I. Verbitskiy, T. Renner, and L. Thamsen, “Scheduling Stream Processing Tasks on Geo-Distributed Heterogeneous Resources”, in Proc. of the 2018 IEEE Int. Conf. on Big Data (IEEE BigData), IEEE, 2018.