Chapter 5

Beyond MapReduce

Acknowledgements

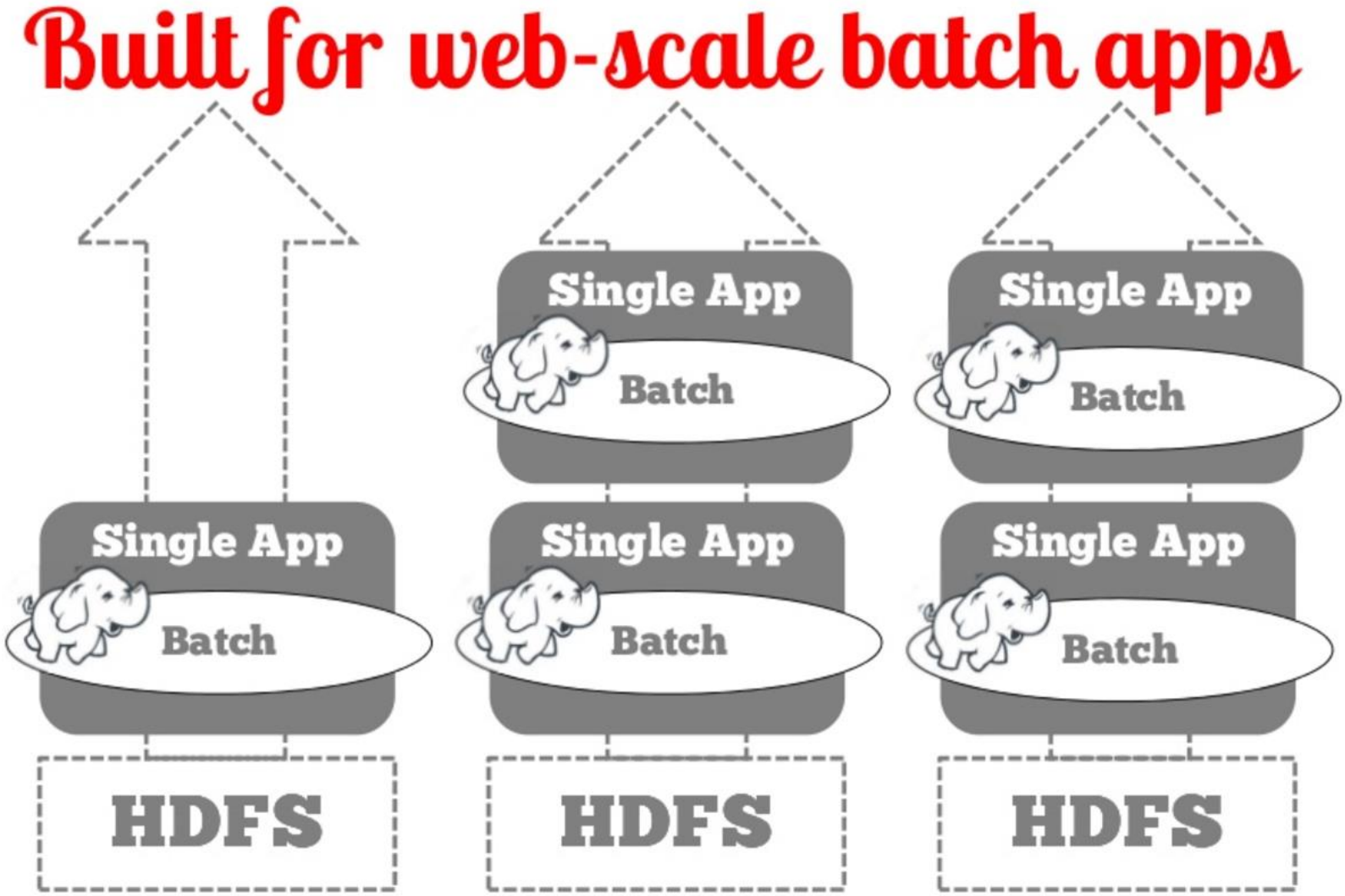
- Hadoop 2 – Beyond MapReduce. uwseiler@codecentric
- HaLoop: Efficient Iterative Data Processing On Large Scale Clusters. Yingyi Bu, Bill Howe, Magda Balazinska, Michael Ernst. VLDB'10.
- Twister: A Runtime for Iterative MapReduce. Jaliya Ekanayake. HPDC'10 MapReduce Workshop.

Chapter Outline

- Hadoop 2
  - HDFS 2
  - YARN
  - YARN Apps
  - HOYA
  - Write your own YARN app
  - Tez, Hive & Stinger Initiative
- Spark
- Iterative MapReduce

Why Hadoop 2

Hadoop 1


Built for web-scale batch apps
Single App
Batch
Single App
Batch
Single App
Batch
Single App
Batch
Single App
Batch
HDFS
HDFS
HDFS

Application Scenarios

- MapReduce is **good** for
  - Embrassassingly parallel algorithms
  - Summing, grouping, filtering, joining
  - Off-line batch jobs on massive data sets
  - Analyzing an entire large dataset
- MapReduce is **OK** for
  - Iterative jobs
    - i.e., graph algorithms
    - Each iteration must read/write data to disk
    - I/O and latency cost of an iteration is high
- MapReduce is **not good** for
  - Jobs that need shared state/coordination
    - Tasks are shared-nothing
    - Shared-state requires scalable state store
  - Low-latency jobs
  - Jobs on small datasets
  - Finding individual records

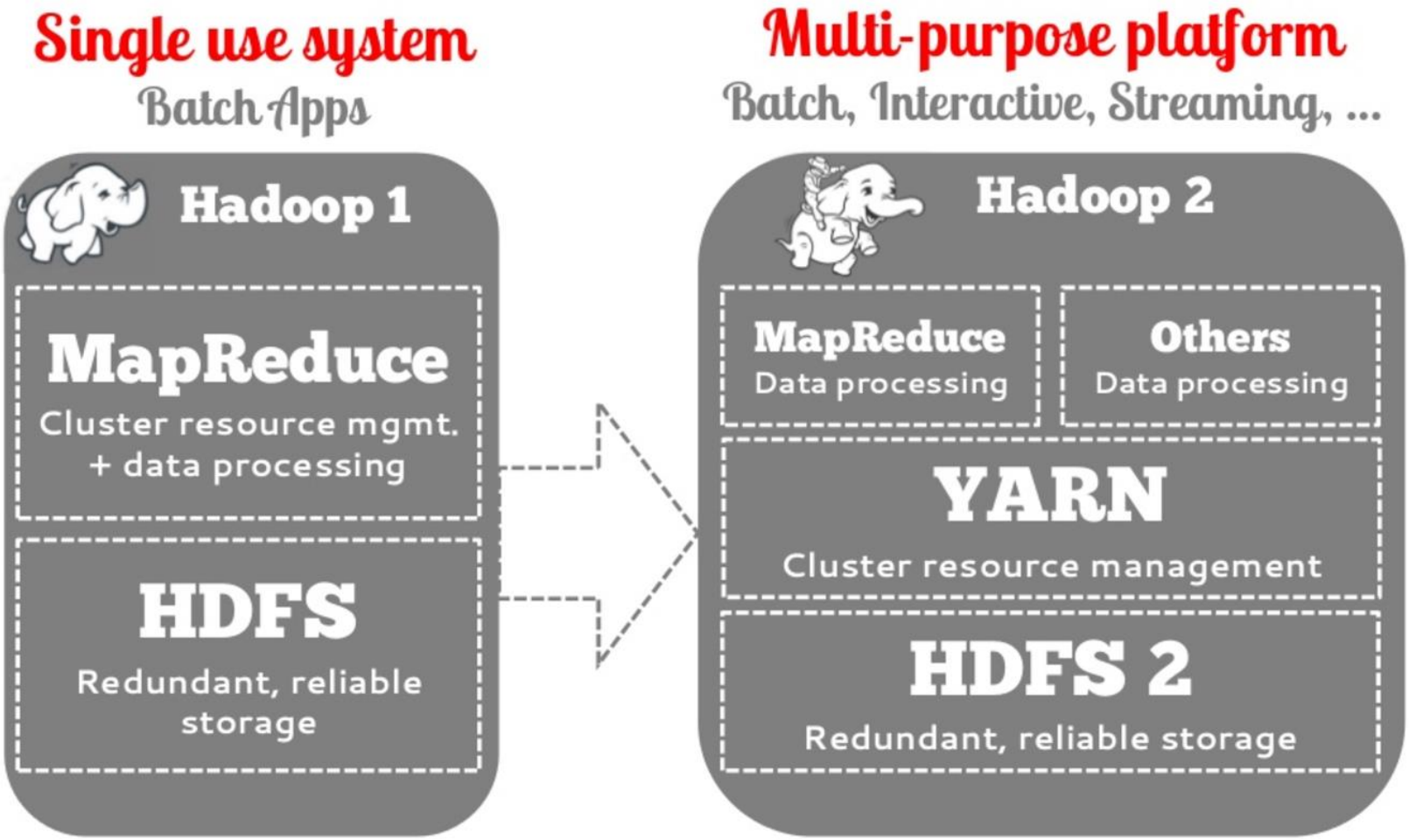
Hadoop 1 limitations

- **Scalability**
  - Maximum cluster size: 4500 nodes
  - Maximum concurrent tasks: 40000
  - Coarse synchronization in JobTracker
- **Availability**
  - Failure kills all queued and running jobs
- **Hard partition** of resources into map & reduce slots
  - Low resource utilization
- Lacks support for **alternate** paradigms and services
  - Iterative applications implemented using MapReduce are 10x slower

Brief history of Hadoop 2

- Originally conceived & architected by the team at Yahoo!
  - Arun Merthy created the original JIRA in 2008 and now is the Hadoop 2 release manager
- The community has been working on Hadoop 2 for over 4 years
- Hadoop 2 based architecture running at scale at Yahoo!
  - Deployed on 35,000+ nodes for 6+ months
- Version 2.2 was announced GA in October 2013

Hadoop 2: Next-gen platform



Taking Hadoop beyond batch

Store all data in one place

Interact with data in multiple ways

Applications run natively in Hadoop

Batch	Interactive	Online	Streaming	Graph	In-Memory	Other
MapReduce	Tez	HOYA	Storm, ...	Giraph	Spark	Search, ...

YARN

Cluster resource management

HDFS 2

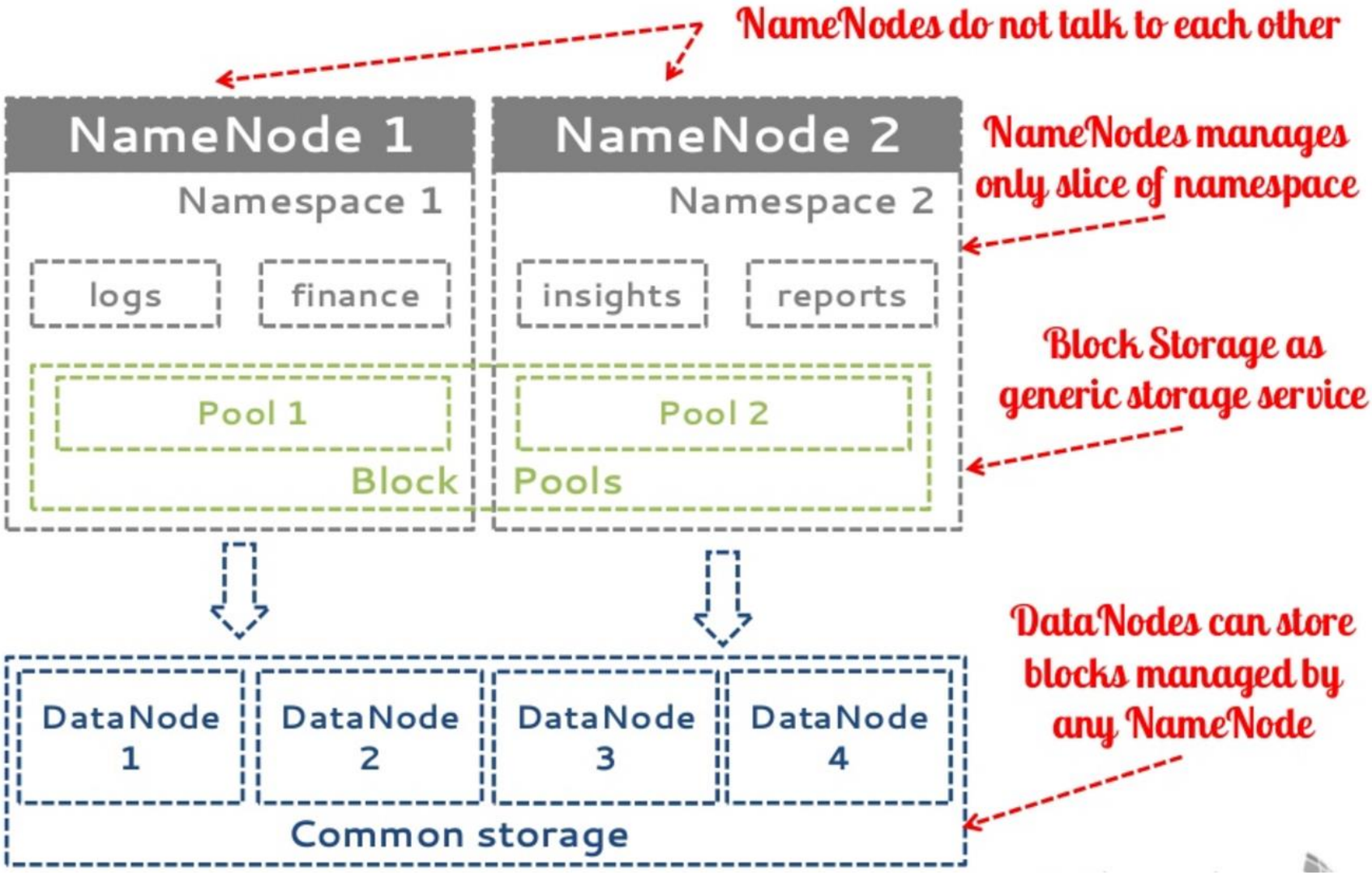
Redundant, reliable storage

HDFS 2

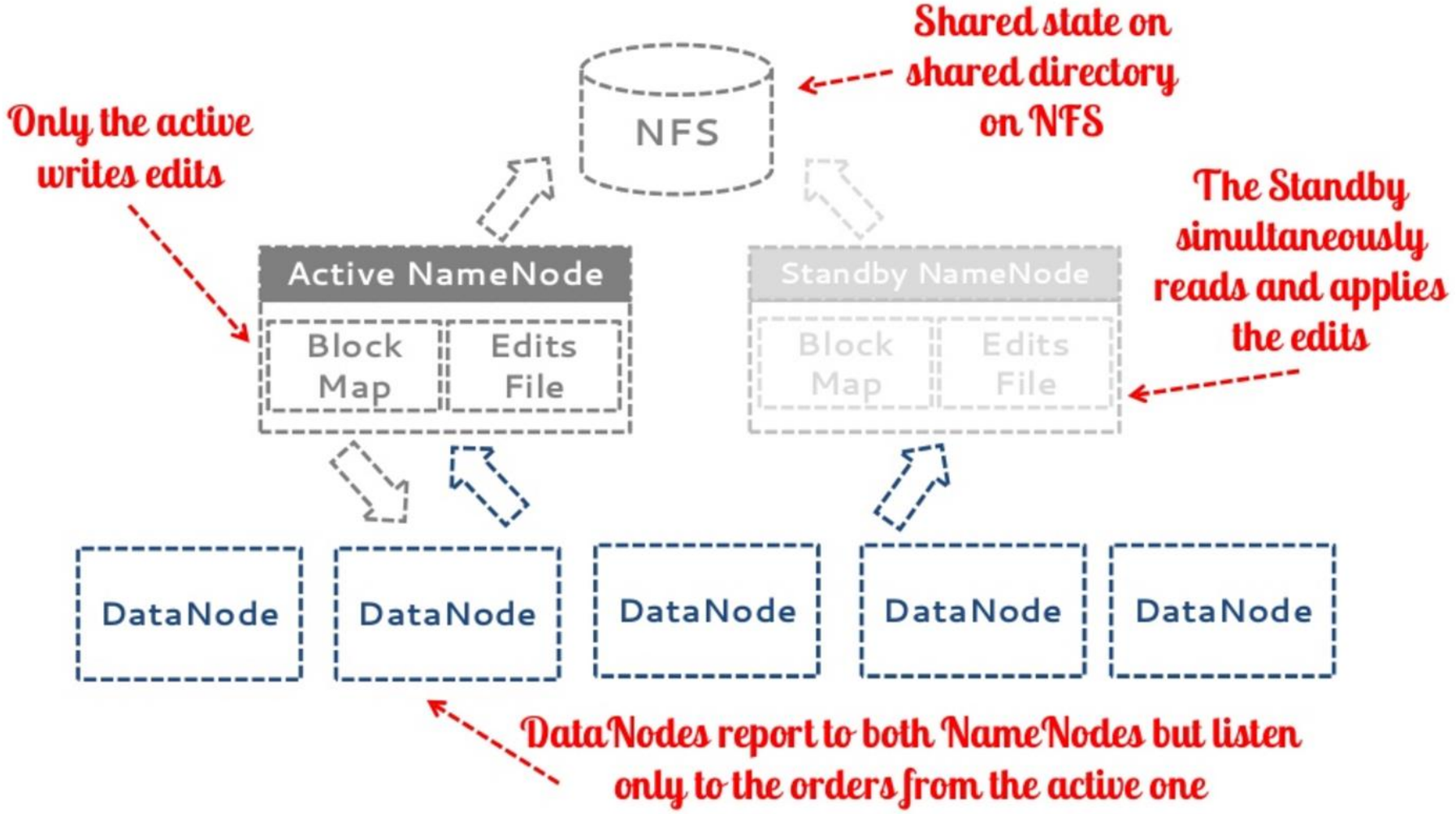
In a nutshell

- Removes tight coupling of Block Storage and Namespace
- High availability
- Scalability & Isolation
- Increased performance

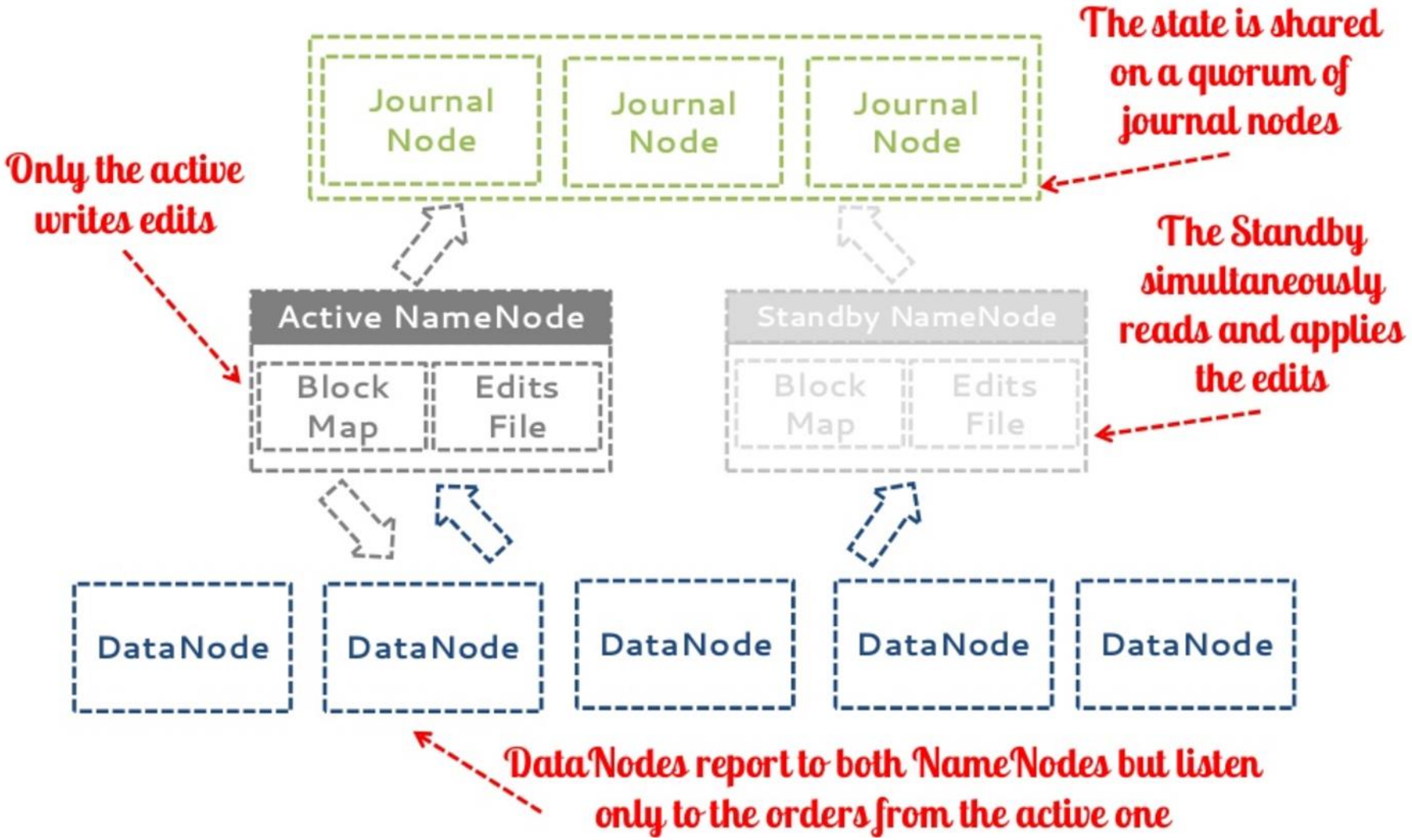
Federation


NameNodes do not talk to each other
NameNode 1
NameNode 2
Namespace 1
Namespace 2
logs
finance
insights
reports
Pool 1
Pool 2
Block Pools
NameNodes manages only slice of namespace
Block Storage as generic storage service
DataNode 1
DataNode 2
DataNode 3
DataNode 4
Common storage
DataNodes can store blocks managed by any NameNode

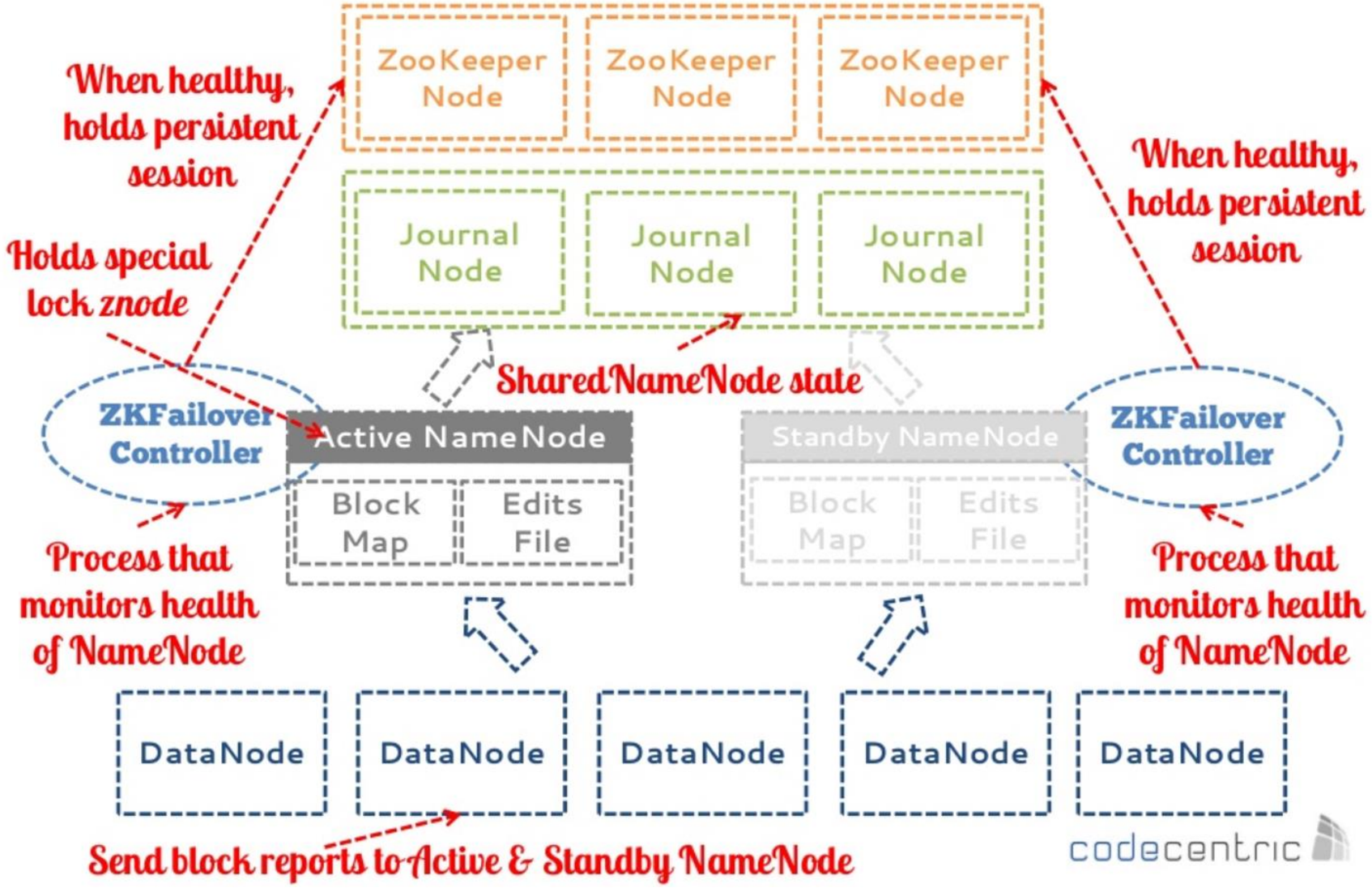
Architecture



Quorum based storage



High availability



Snapshots

- Admin can create point in time snapshots of HDFS
  - Of the entire file system
  - Of a specific data-set (sub-tree directory of file system)
- Restore state of entire file system or data-set to a snapshot (like Apple Time Machine)
  - Protect against user errors
- Snapshot diffs identify changes made to data set

NFS Gateway

- Supports NFS v3 (NFS v4 is work in progress)
- Supports all HDFS commands
  - List files
  - Copy, move files
  - Create and delete directories
- Ingest for large scale analytical workloads
  - Load immutable files as source for analytical processing
  - No random writes
- Stream files into HDFS
  - Log ingest by applications writing directly to HDFS client mount

Performance

- Many improvements
  - Write pipeline (e.g. new primitives *hflush*, *hsync*)
  - Read path improvements for fewer memory copies
  - Short-circuit local reads for 2-3x faster random reads
  - I/O improvements using *posix_fadvise()*
  - *libhdfs* improvements for zero copy reads
- Significant improvements: I/O 2.5-5x faster

YARN

Design Goals

- Build a new abstraction layer by splitting up the two major functions of the JobTracker
  - Cluster resource management
  - Application life-cycle management
- Allow other processing paradigms
  - Flexible API for implementing YARN apps
  - MapReduce becomes YARN app
  - Lots of different YARN apps

Concepts

- **Application**
  - Application is a job committed to the YARN framework
  - Example: MapReduce job, Storm topology, …
- **Container**
  - Basic unit of allocation
  - Fine-grained resource allocation across multiple resource types
    - RAM, CPU, Disk, Network, GPU, etc.
    - container_0 = 4 GB, 1 CPU
    - container_1 = 512MB, 6 CPU's
  - Replaces the fixed map/reduce slots

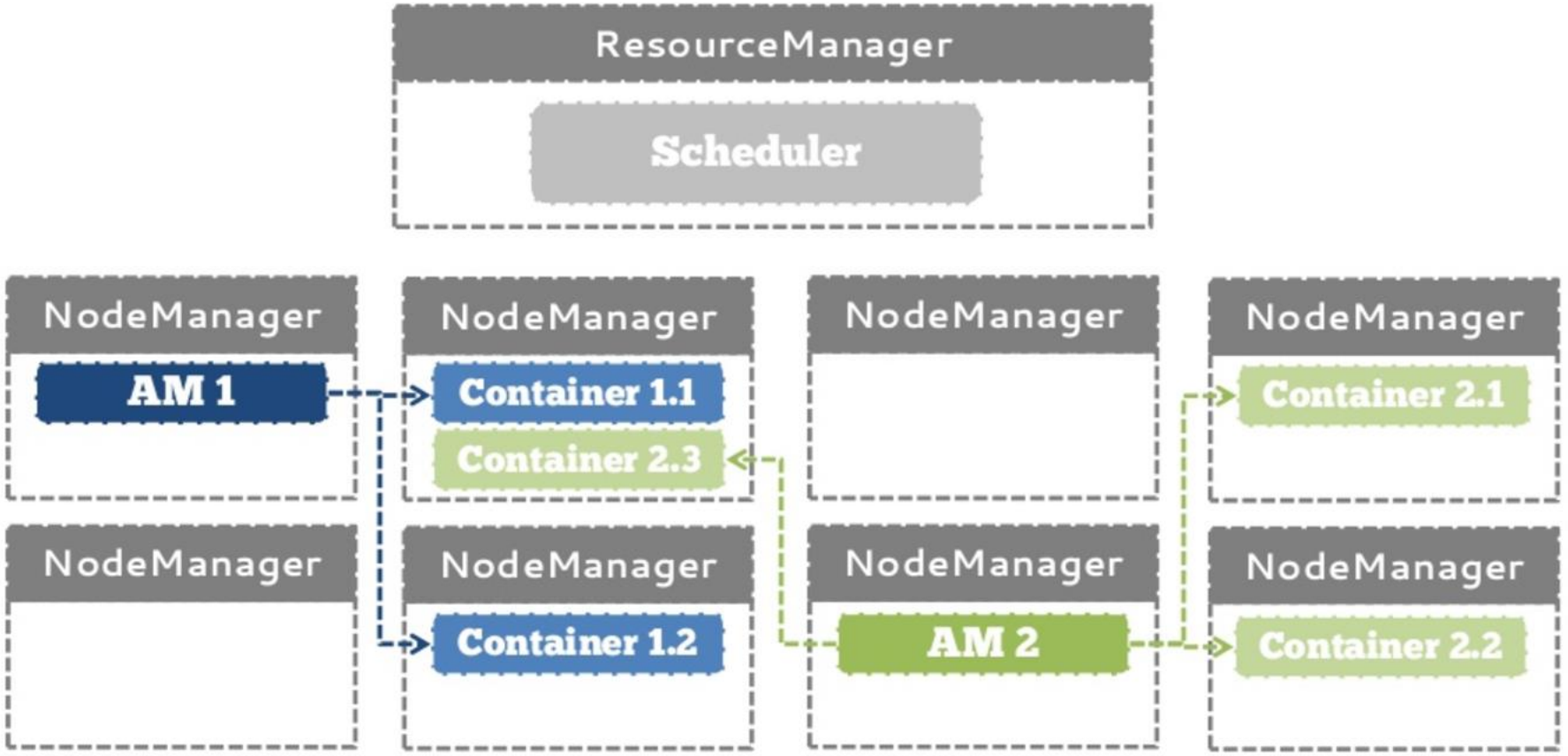
Architecture

- Resource Manager
  - Global resource scheduler
  - Hierarchical queues
- Node Manager
  - Per-machine agent
  - Manages the life-cycle of container
  - Container resource monitoring
- Application Master
  - Per-application
  - Manages application scheduling and task execution
  - E.g. MapReduce Application Master

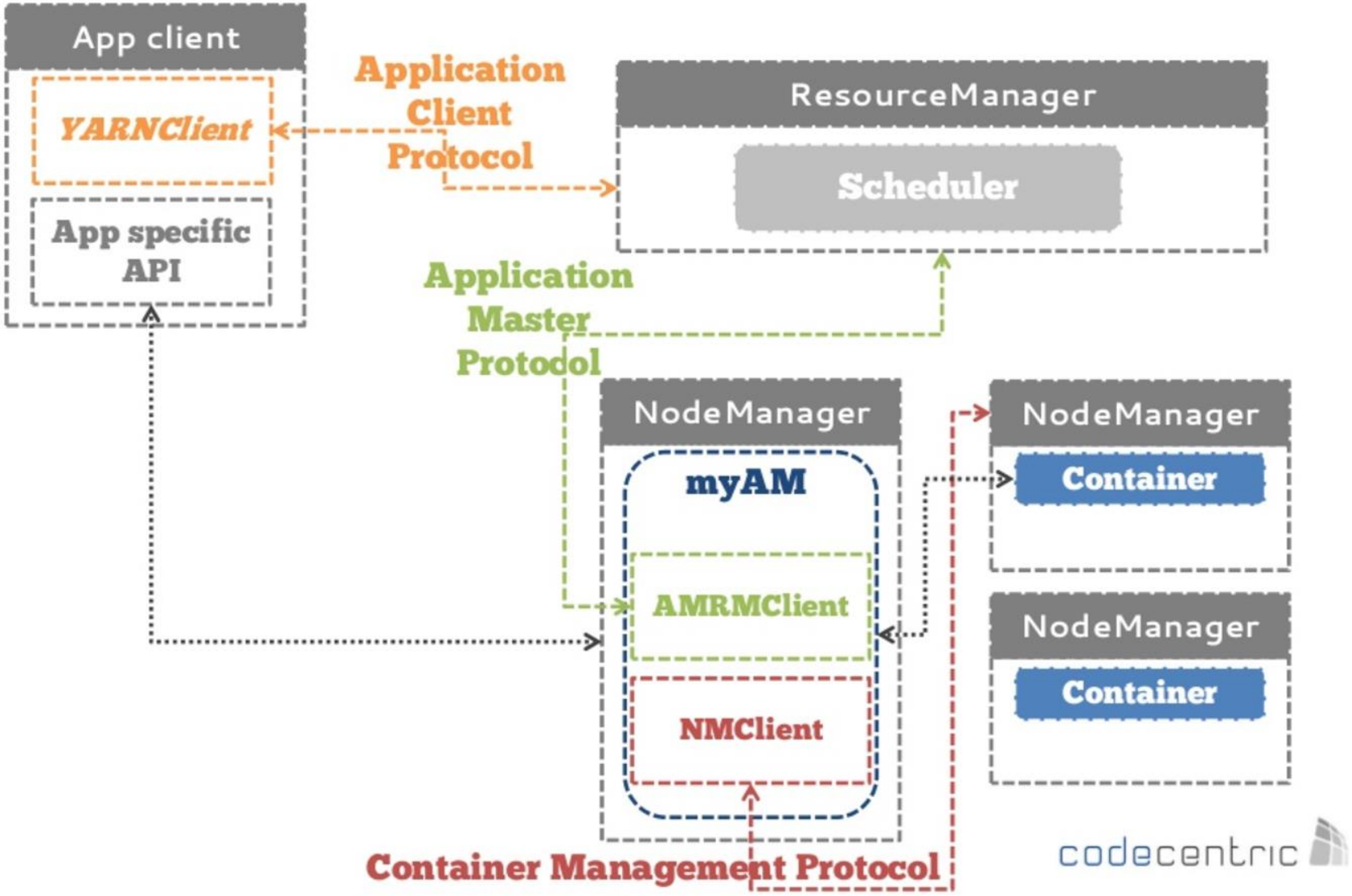
Architectural Overview

Split up the two major functions of the JobTracker

Cluster resource management & Application life-cycle management



YARN: Application flow



YARN Apps

Overview

• MapReduce 2	Batch
• HOYA	Hbase on YARN
• Storm	Stream Processing
• Samza	Stream Processing
• Apache S4	Stream Processing
• Spark	Iterative/Interactive
• Apache Giraph	Graph Processing
• Apache Hama	Bulk Synchronous Parallel
• Elastic Search	Scalable Search
• Cloudera Llama	Impala on YARN

MapReduce 2: In a nutshell

- MapReduce is now a **YARN app**
  - No more map and reduce slots, it's containers now
  - No more JobTracker, it's YarnAppmaster library now
- **Multiple versions** of MapReduce
  - The older mapred APIs work without modification or recompilation
  - The newer mapreduce APIs may need to be recompiled
- Still has **one master** server component:
  - The Job History Server
    - The Job History Server stores the execution of jobs
    - Used to audit prior execution of jobs
    - Used by YARN framework to store charge backs at that level
- Better **cluster utilization**
- Increased **Scalability** & **availability**

MapReduce 2: Shuffle

- **Faster Shuffle**
  - Better embedded server: Netty
- **Encrypted Shuffle**
  - Secure the shuffle phase as data moves across the cluster
  - Requires 2 way HTTPS, certificates on both sides
  - Causes significant CPU overhead, reserve 1 core for this work
  - Certificates stored on each node (Provision with the cluster), refreshed every 10 secs
- **Pluggable Shuffle Sort**
  - Shuffle is the first phase in MapReduce that is guaranteed to not be data-local
  - Pluggable Shuffle/Sort allows application developers or hardware developers to intercept the network-heavy workload and optimize it
  - Typical implementations have hardware components like fast networks and software components like sorting algorithms
  - API will change with future versions of Hadoop

MapReduce 2: Performance

- **Key Optimizations**
  - No hard segmentation of resource into map and reduce slots
  - YARN scheduler is more efficient
  - MR2 framework has become more efficient than MR1
    - Shuffle phase in MRv2 is more performant with the usage of Netty

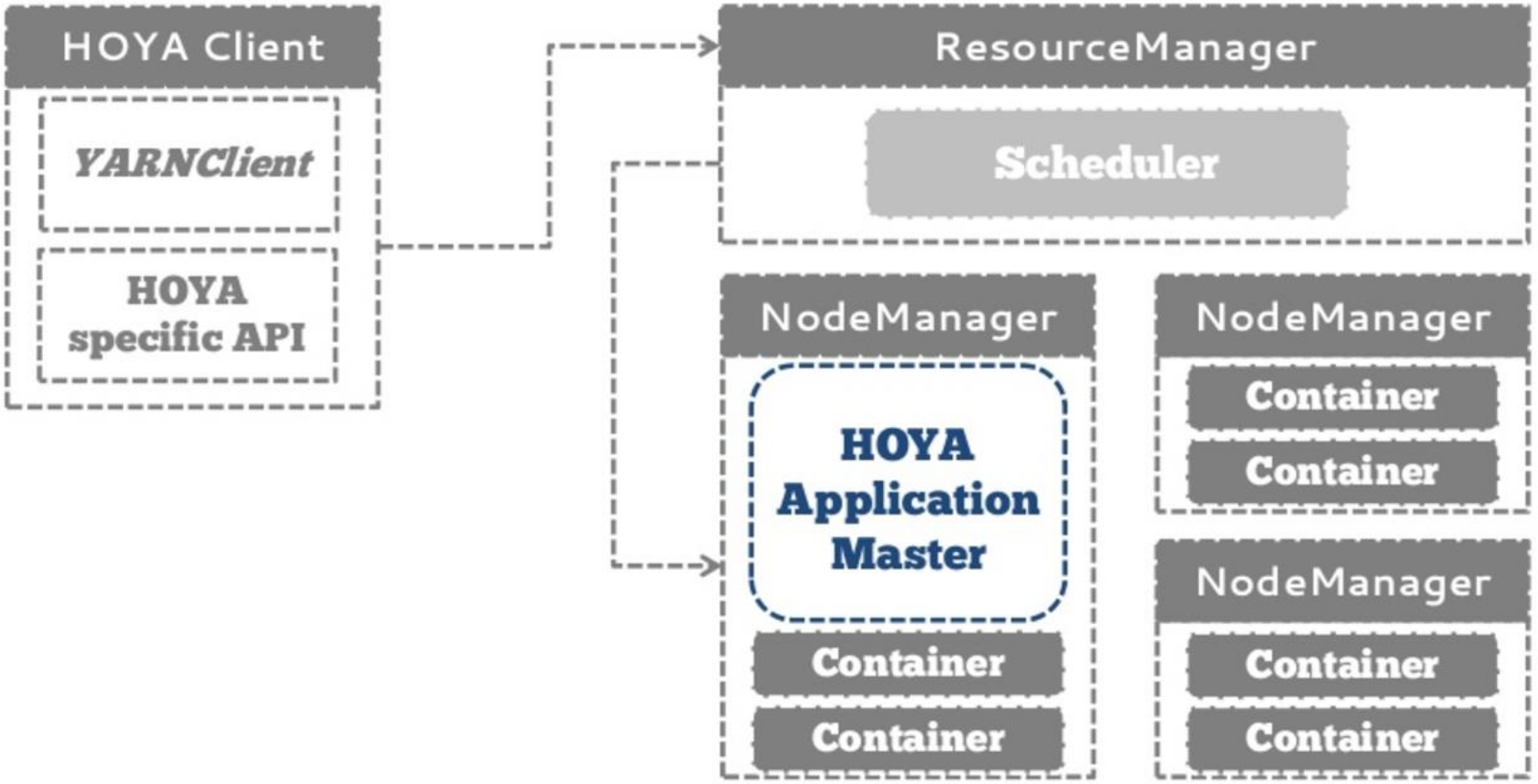
- 40,000+ nodes running YARN across over 365PB of data
- About 400,000 jobs per day for about 10 million hours of compute time
- Estimated 60% - 150% improvement on node usage per day
- Got rid of a whole 10,000 node datacenter because their increased utilization

HOYA

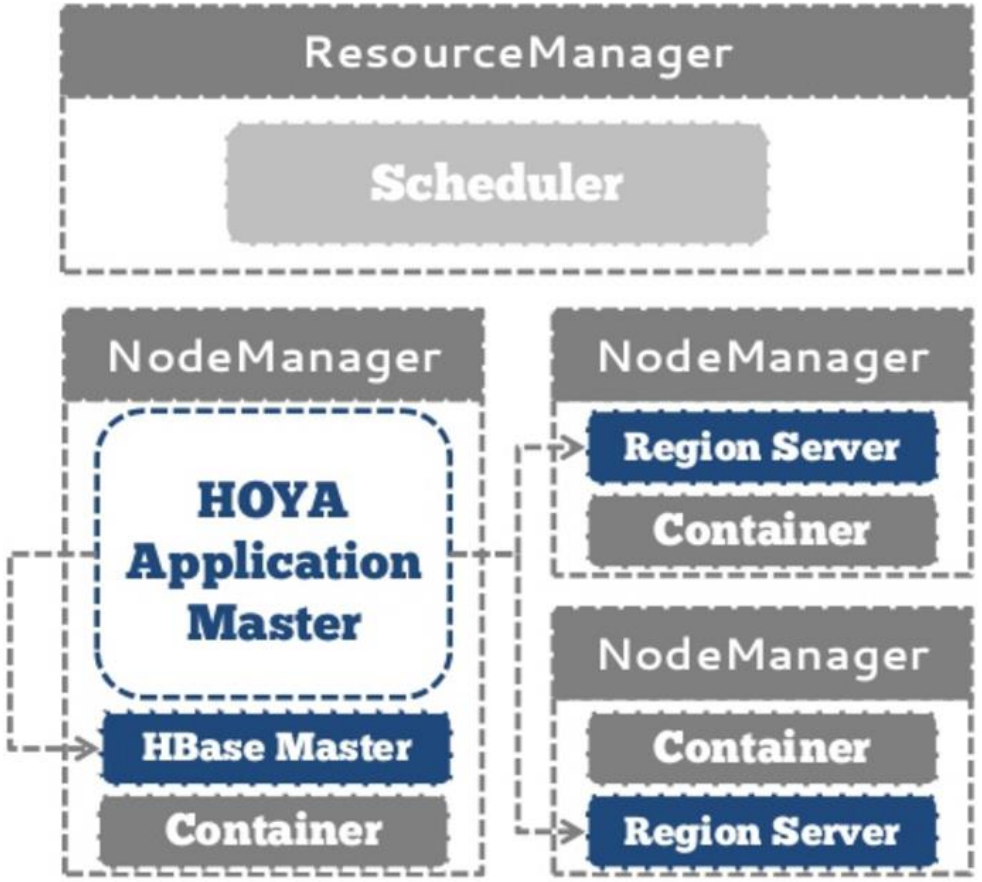
HOYA: In a nutshell

- Create **on-demand** HBase clusters
  - **Small** HBase cluster in large YARN cluster
  - **Dynamic** HBase clusters
  - **Self-healing** HBase cluster
  - **Elastic** HBase clusters
  - **Transient/intermittent** clusters for workflows
- Configure **custom** configurations & versions
- Better **isolation**
- More efficient **utilization**/sharing of cluster

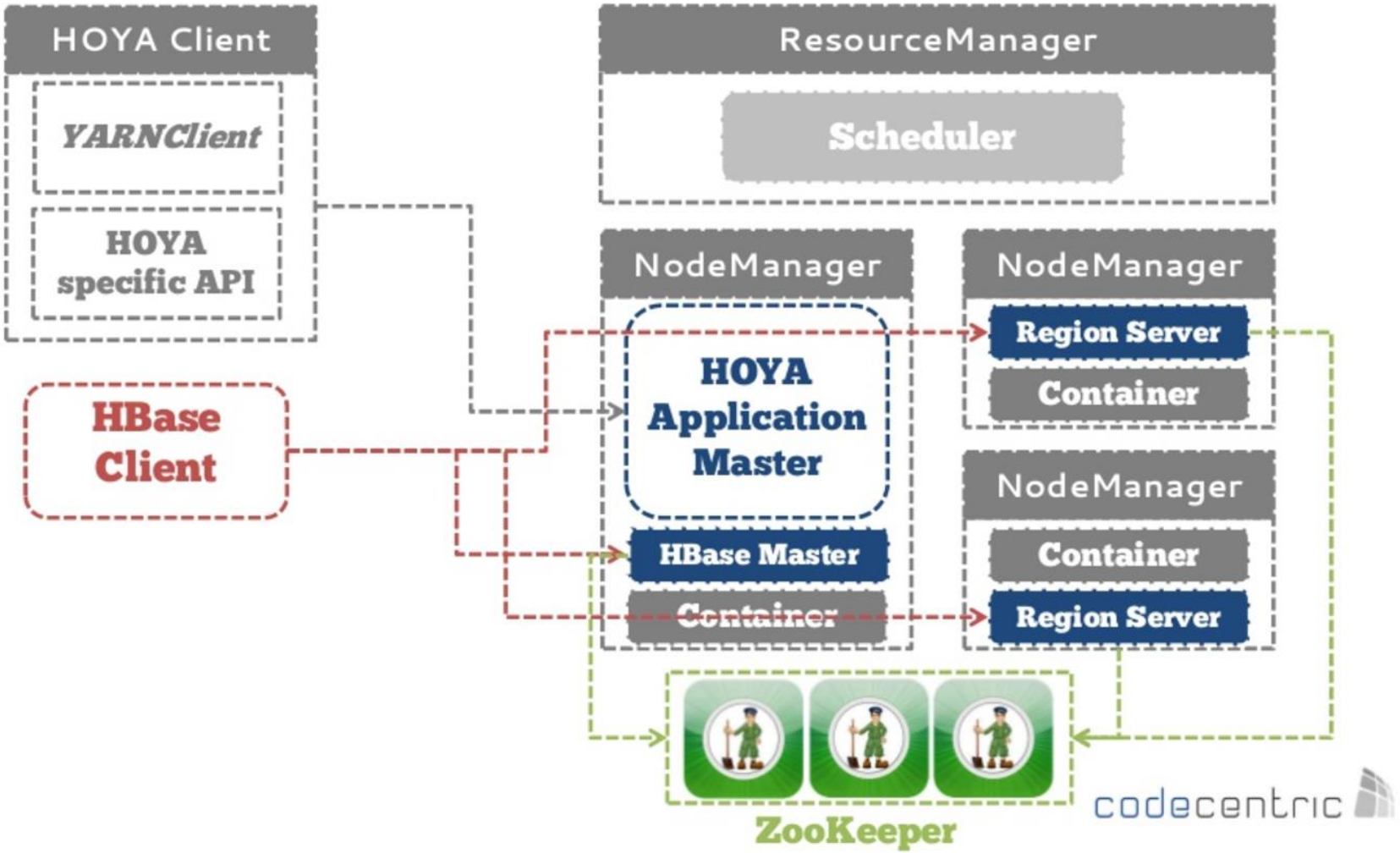
HOYA: Creation of AppMaster



HOYA: Deployment of HBase



HOYA: Bind via ZooKeeper

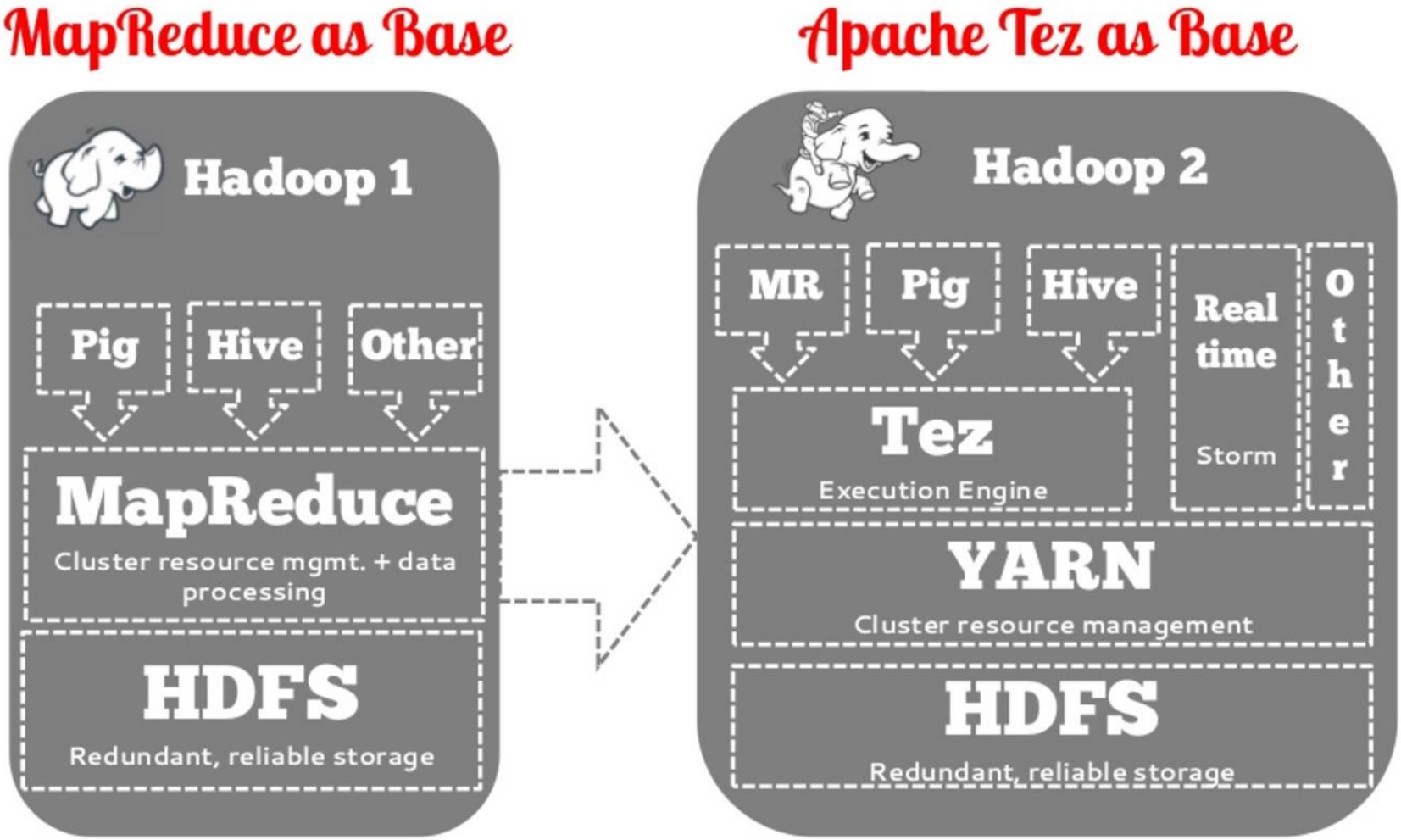


Tez, Hive & Stinger Initiative

Apache Tez: In a netshell

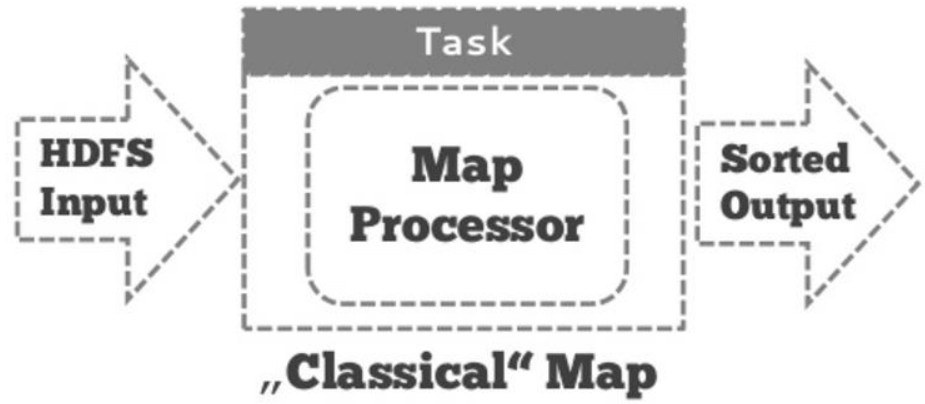
- **Distributed execution framework** that works on computations represented as dataflow graphs
- Tez is *Hindi* for "**speed**"
- Naturally **maps to execution plans** produced by query optimizers
- **Highly customizable** to meet a broad spectrum of use cases and to enable dynamic performance optimizations at runtime
- Built **on top of YARN**

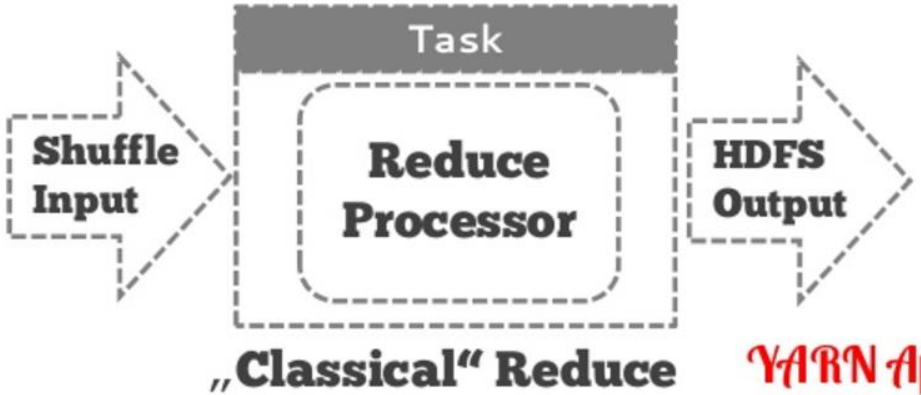
Apache Tez: The new primitive

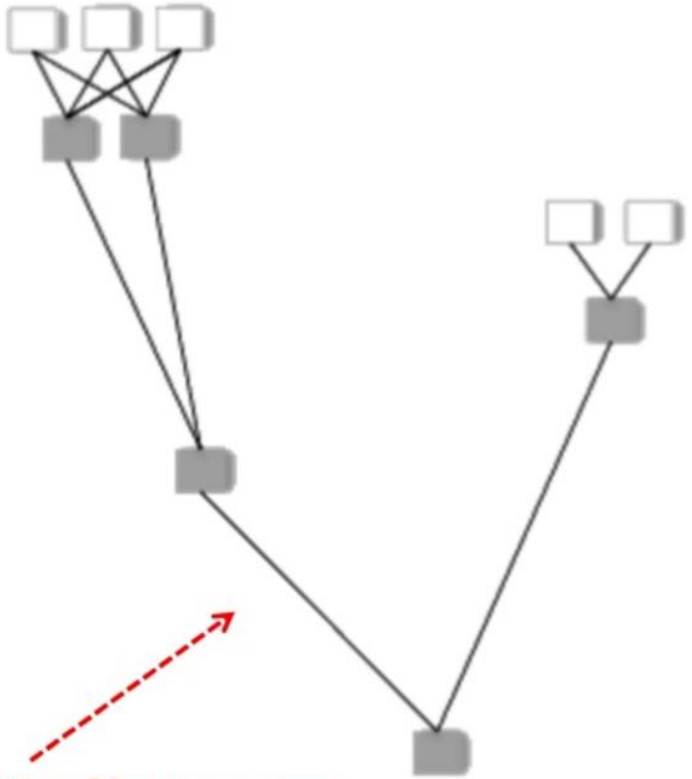


Apache Tez: Architecture

- Task with pluggable Input, Processor & Output





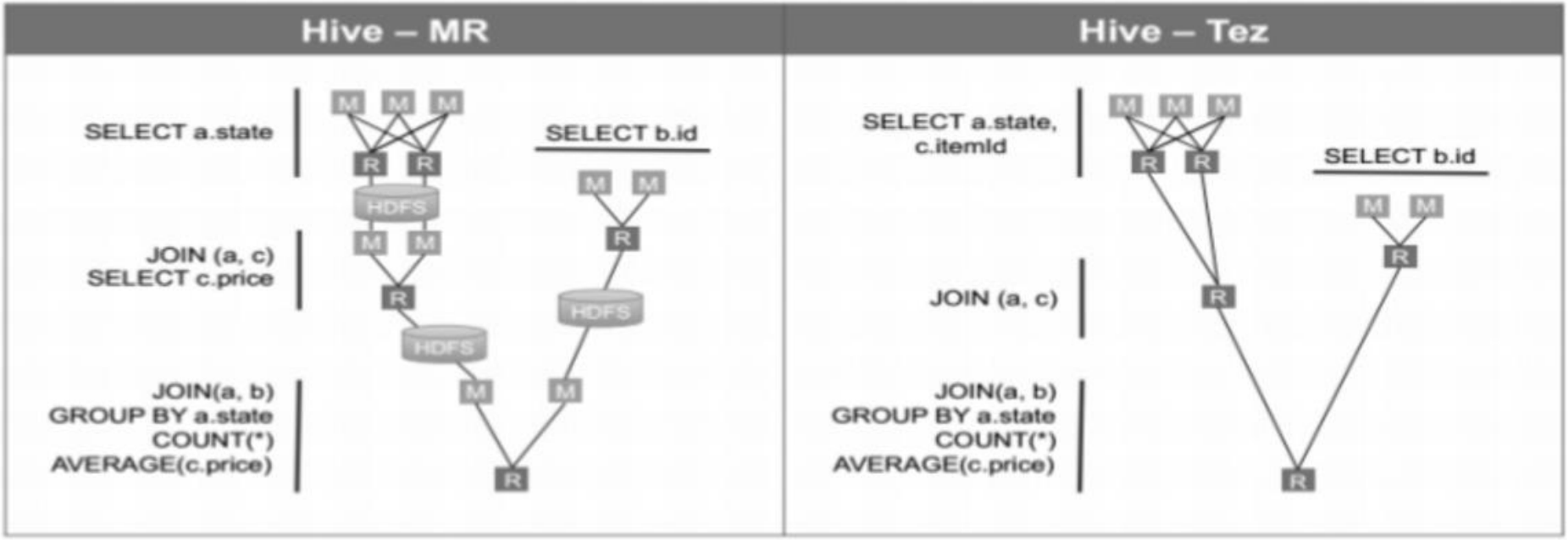


YARN ApplicationMaster to run DAG of Tez Tasks

codecentric

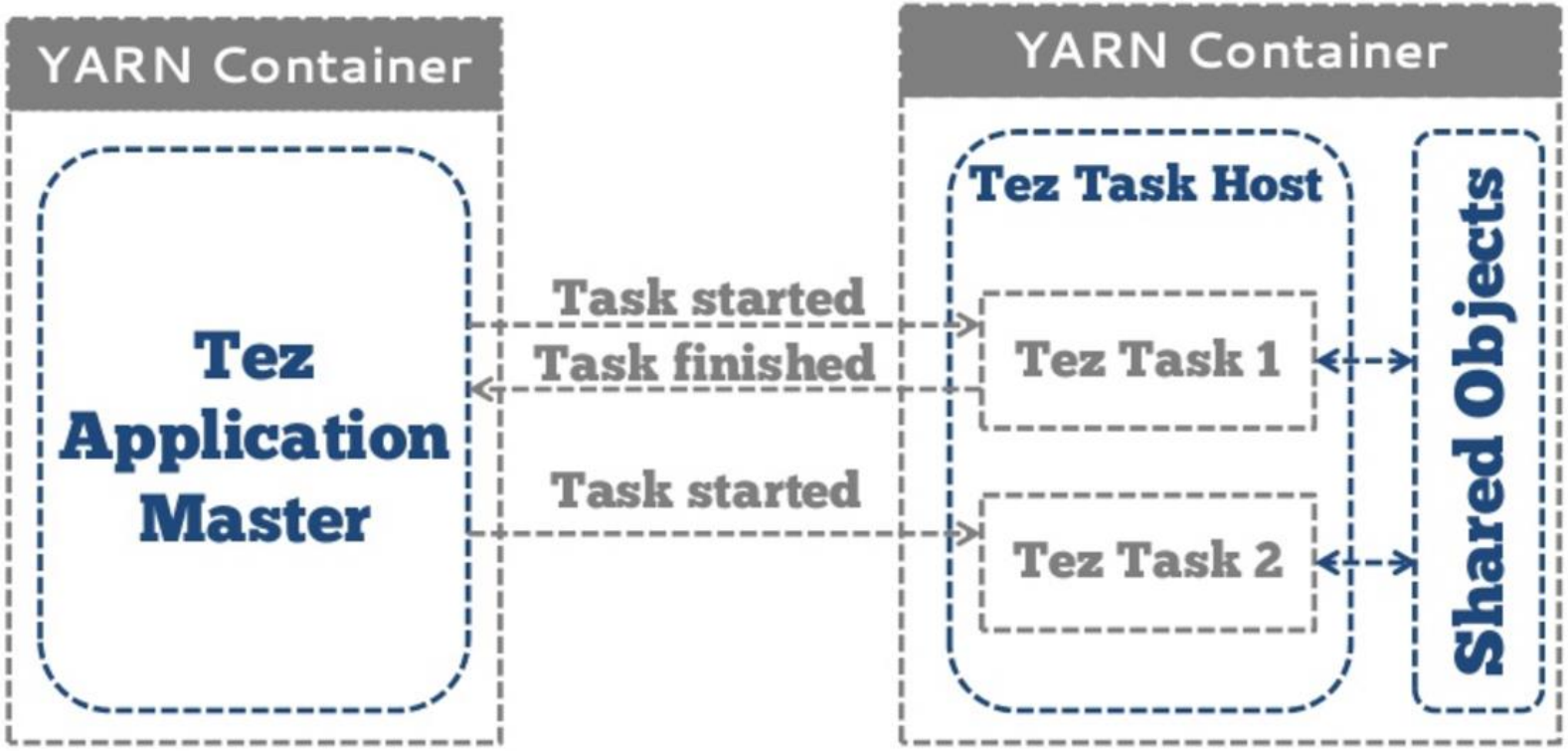
Apache Tez: Performance

- Performance gains over MapReduce
  - Eliminate replicated write barrier between successive computations
  - Eliminate job launch overhead of workflow jobs
  - Eliminate extra stage of map reads in every workflow job
  - Eliminate queue & resource contention suffered by workflow jobs that are started after a predecessor job completes



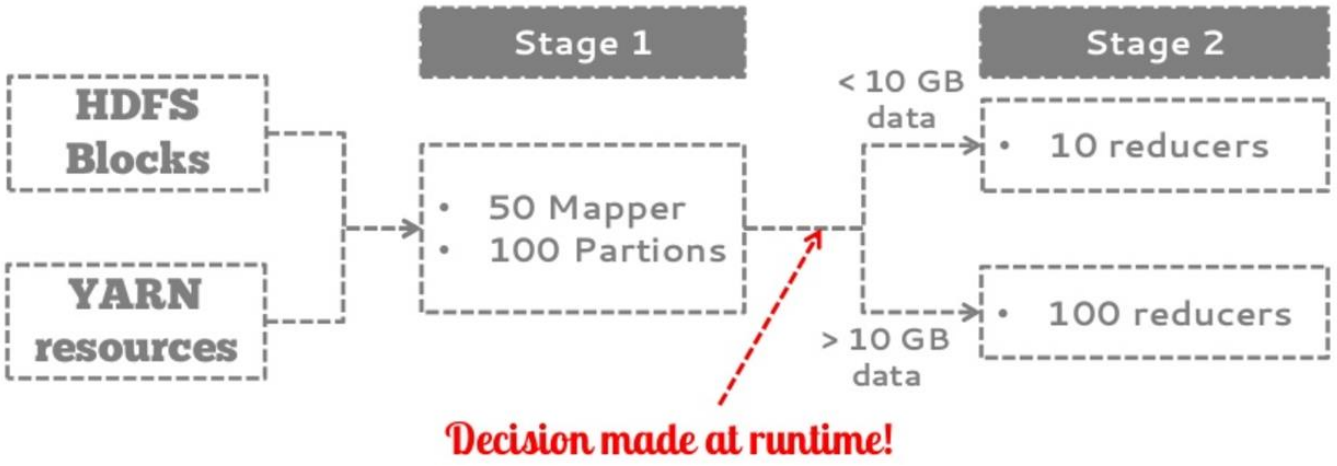
Apache Tez: Performance

- Optimal resource management
  - Reuse YARN containers to launch new tasks
  - Reuse YARN containers to enable shared objects across tasks



Apache Tez: Performance

- Execution plan reconfiguration **at runtime**



- **Dynamic** physical data flow decisions

Apache Tez: Performance

```
SELECT a.state, COUNT(*),
    AVERAGE(c.price)
        FROM a
    JOIN b ON (a.id = b.id)
JOIN c ON (a.itemId = c.itemId)
    GROUP BY a.state
```

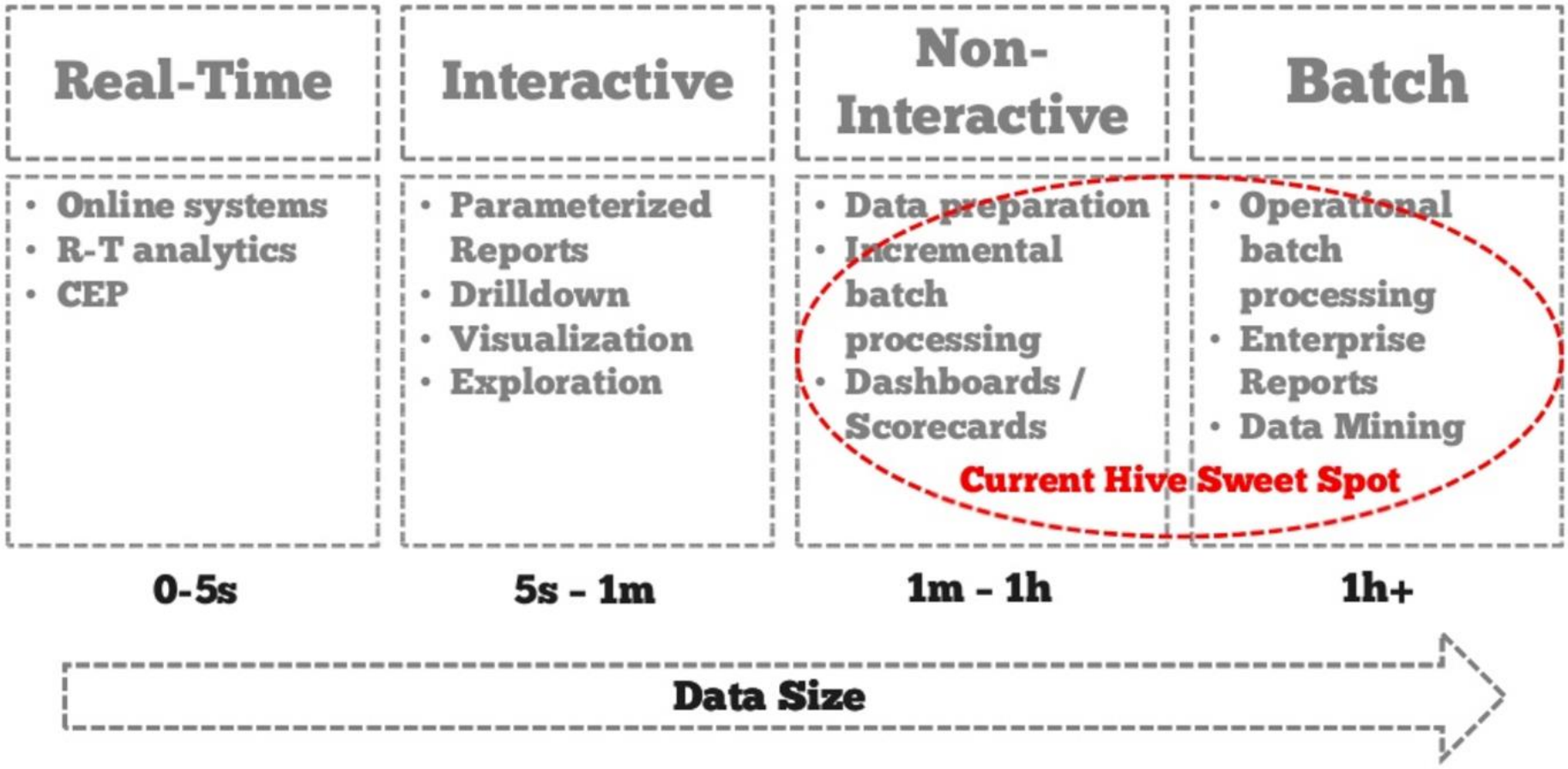
Existing Hive	
Parse Query	0.5s
Create Plan	0.5s
Launch Map-Reduce	20s
Process Map-Reduce	10s
Total	31s

Hive/Tez	
Parse Query	0.5s
Create Plan	0.5s
Launch Map-Reduce	20s
Process Map-Reduce	2s
Total	23s

Tez & Hive Service	
Parse Query	0.5s
Create Plan	0.5s
Submit to Tez Service	0.5s
Process Map-Reduce	2s
Total	3.5s

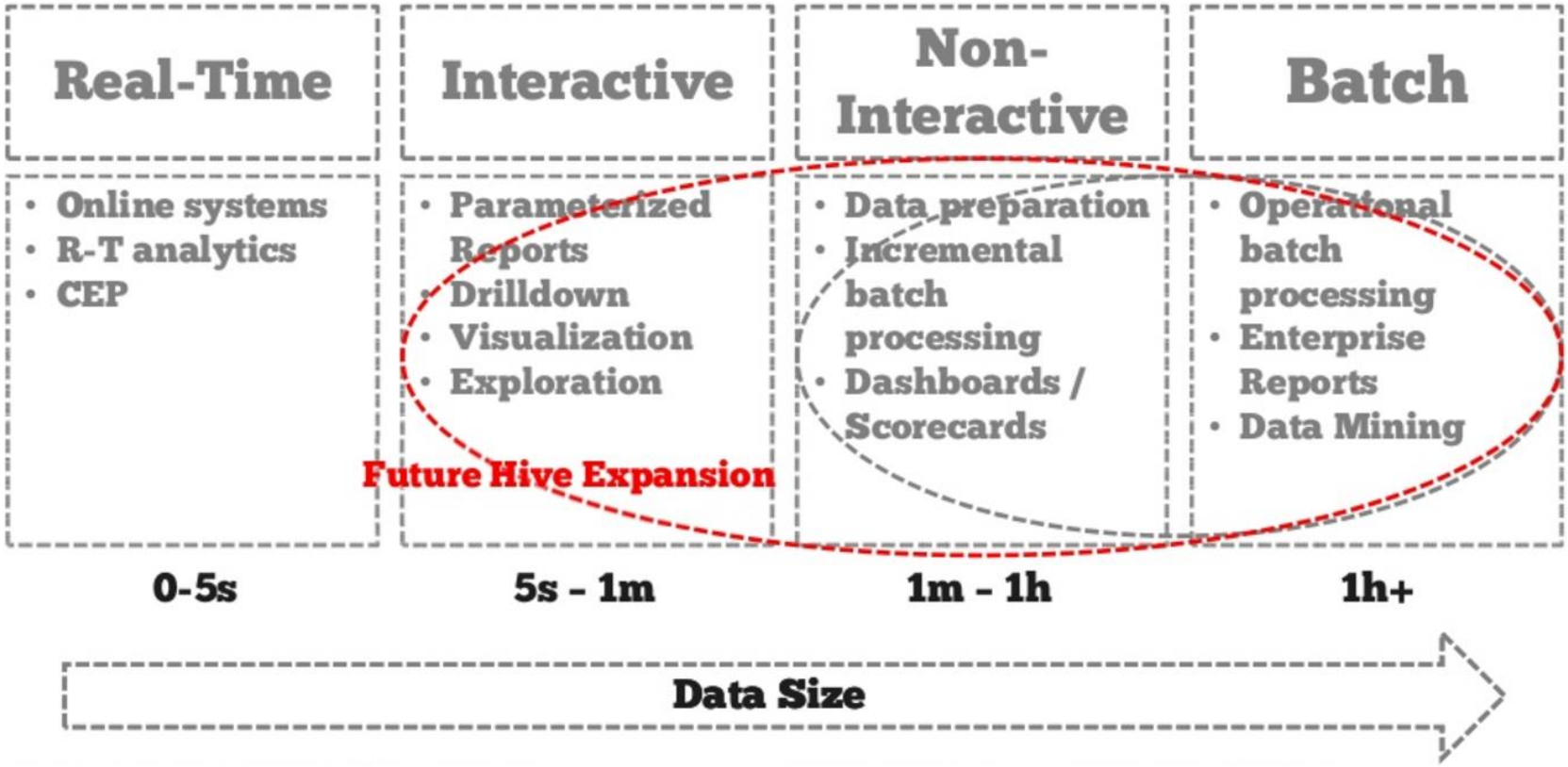
*** No exact numbers, for illustration only**

Hive: Current Focus Area

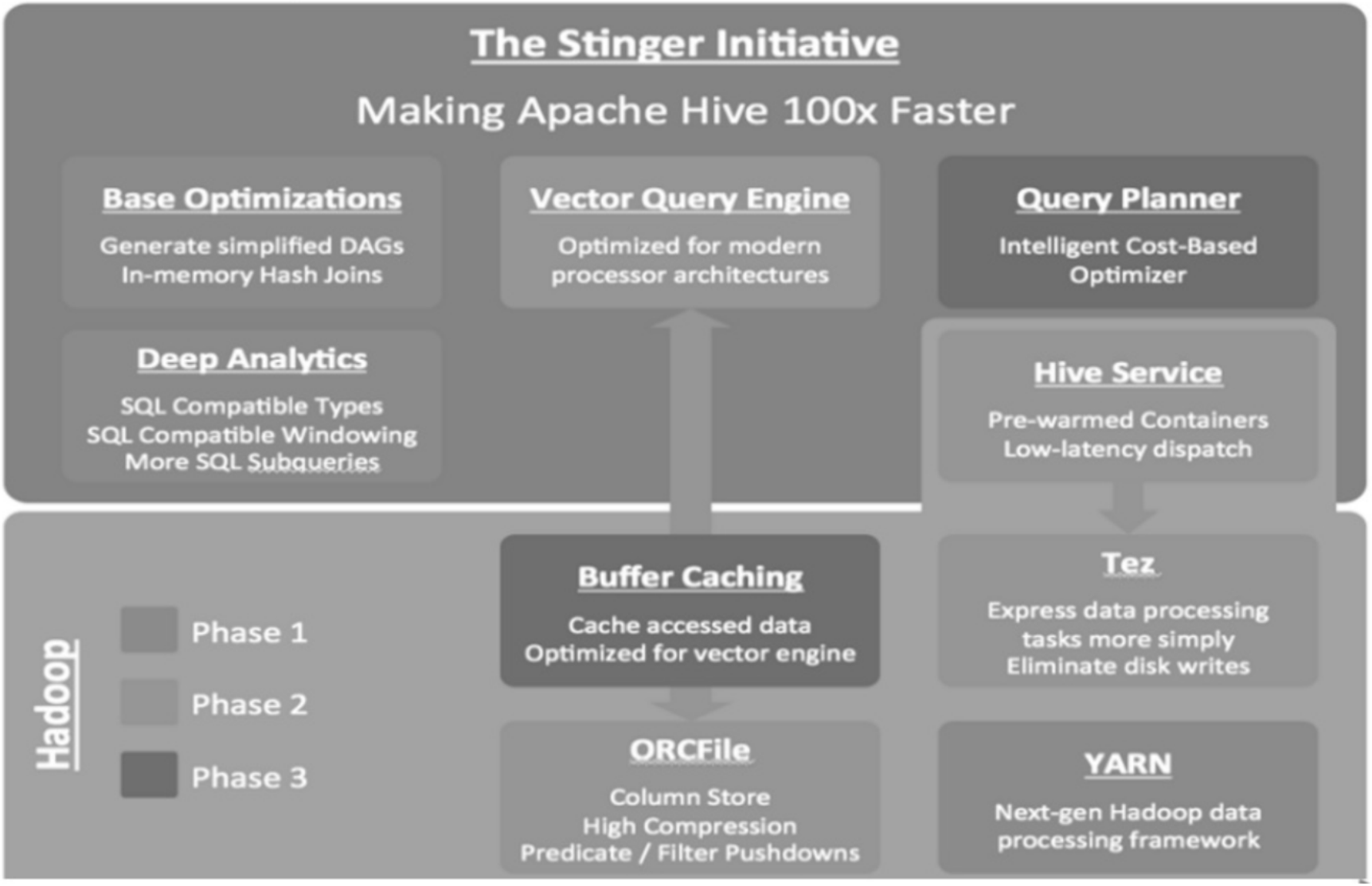


Stinger: Extending the sweet spot

- Improve Latency & Throughput
  - Query engine Improvement
  - New “Optimized RCFile” column store
  - Nex-gen runtime (elim’s M/R latency)
- Extend Deep Analytical Ability
  - Analytics functions
  - Improved SQL coverage
  - Continued focus on core Hive use cases



Stinger Initiative: In a nutshell



Stinger: Enhancing SQL Semantics

Hive SQL Datatypes

- INT
- TINYINT/SMALLINT/BIGINT
- BOOLEAN
- FLOAT
- DOUBLE
- STRING
- TIMESTAMP
- BINARY
- DECIMAL
- ARRAY, MAP, STRUCT, UNION
- DATE
- VARCHAR
- CHAR

Hive SQL Semantics

- SELECT, INSERT
- GROUP BY, ORDER BY, SORT BY
- JOIN on explicit join key
- Inner, outer, cross and semi joins
- Sub-queries in FROM clause
- ROLLUP and CUBE
- UNION
- Windowing Functions (OVER, RANK, etc)
- Custom Java UDFs
- Standard Aggregation (SUM, AVG, etc.)
- Advanced UDFs (ngram, Xpath, URL)
- Sub-queries in WHERE, HAVING
- Expanded JOIN Syntax
- SQL Compliant Security (GRANT, etc.)
- INSERT/UPDATE/DELETE (ACID)

SQL Compliance

Hive 12 provides a wide array of SQL datatypes and semantics so your existing tools integrate more seamlessly with Hadoop

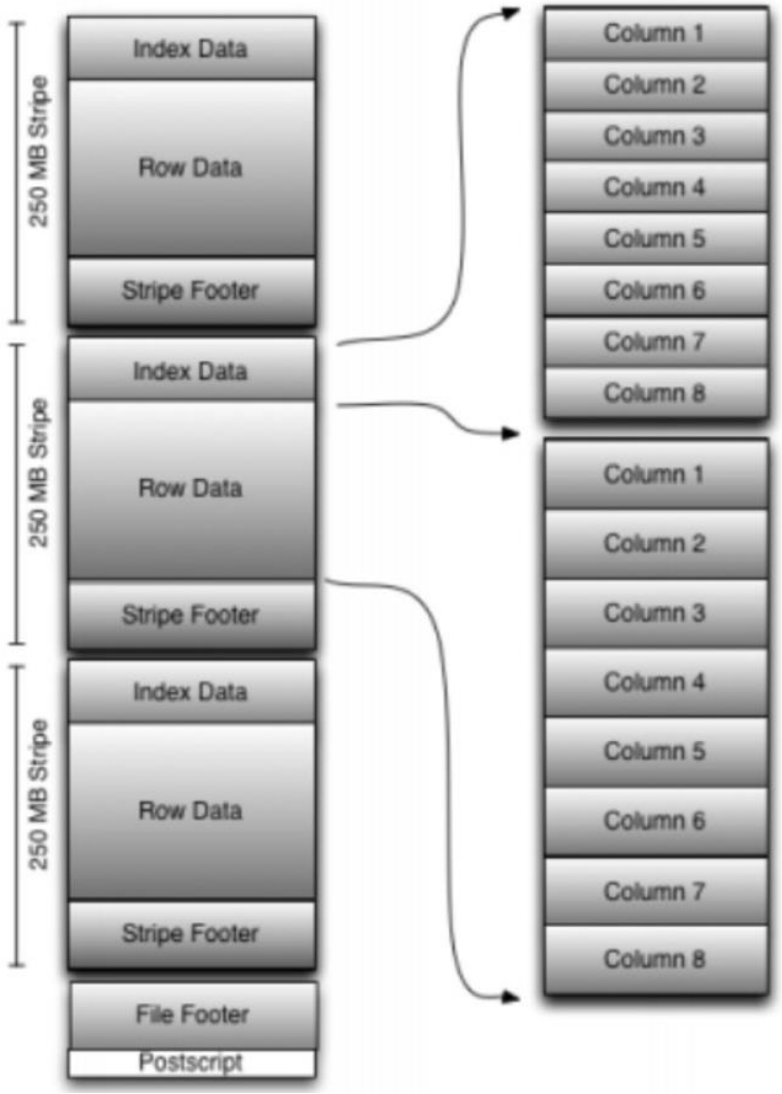
- Available
- Hive 0.12
- Roadmap

String: ORCFile in a nutshell

- **O**ptimized **R**ow **C**olumnar File
  - Columns stored separatedly
- Knows types
  - Uses type-specific encoders
  - Stores statistics (min, max, sum, count)
- Has light-weight index
  - Skip over blocks of rows that don't matter
- Larger blocks
  - 256MB by default
  - Has an index for block boundaries

Stinger: ORCFile Layout

Large block size well suited for HDFS



Columnar format arranges columns adjacent within the file for compression and fast access.

codecentric

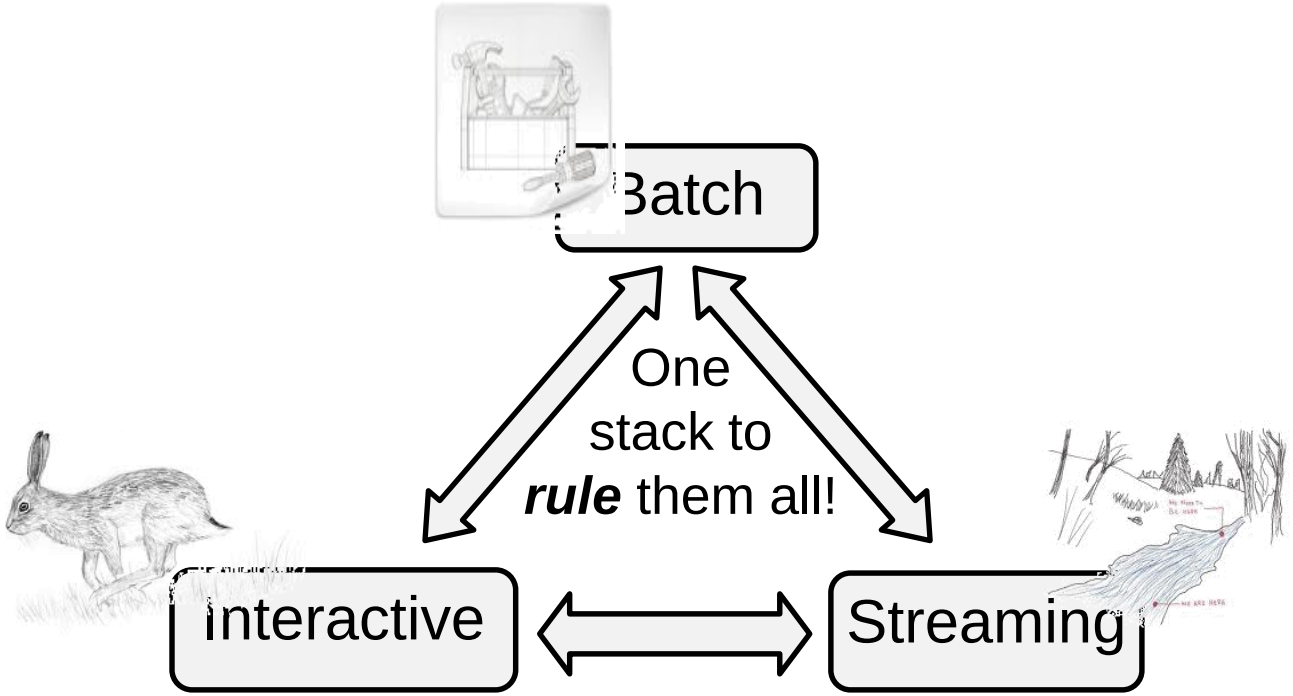
Stinger: ORCFile advantages

- **High compression**
  - Many tricks used out-of-the-box to ensure high compression rates
  - RLE, dictionary encoding, *etc*.
- **High performance**
  - Inline indexes record value ranges within blocks of ORCFile data
  - Filter pushdown allows efficient scanning during precise queries
- **Flexible data model**
  - All hive types including maps, structs and unions
- **Join optimizations**
  - New Join Types added or improved in Hive 0.11
  - More Efficient Query Plan Generation
- **Vectorization**
  - Make the most use of L1 and L2 caches
  - Avoid branching whenever possible
- **Optimized query planner**
  - Automatically determine optimal execution parameters
- **Buffering**
  - Cache hot data in memory

Spark

Spark: Motivation

- Easy to combine batch, streaming, and interactive computations
- Easy to develop sophisticated algorithms
- Compatible with existing open source ecosystem (Hadoop/HDFS)

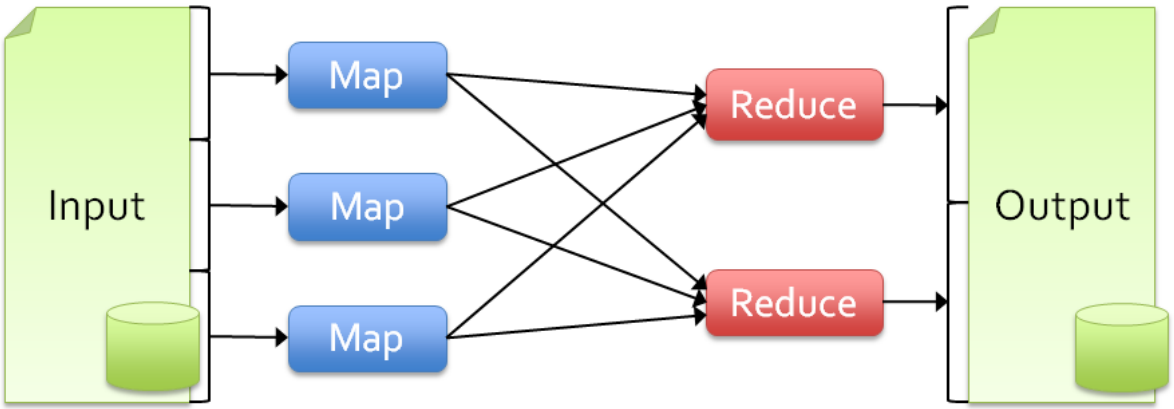


Spark: Motivation

- MapReduce greatly simplified big data analysis on large, unreliable clusters. It is great at **one-pass** computation.
- But as soon as it got popular, users wanted more:
  - More **complex**, **multi-pass** analytics (e.g. ML, graph)
  - More **interactive** ad-hoc queries
  - More **real-time** stream processing
- All 3 need faster data sharing across parallel jobs
  - One reaction: **specialized models** for some of these apps, e.g.,
    - Pregel (graph processing)
    - Storm (stream processing)

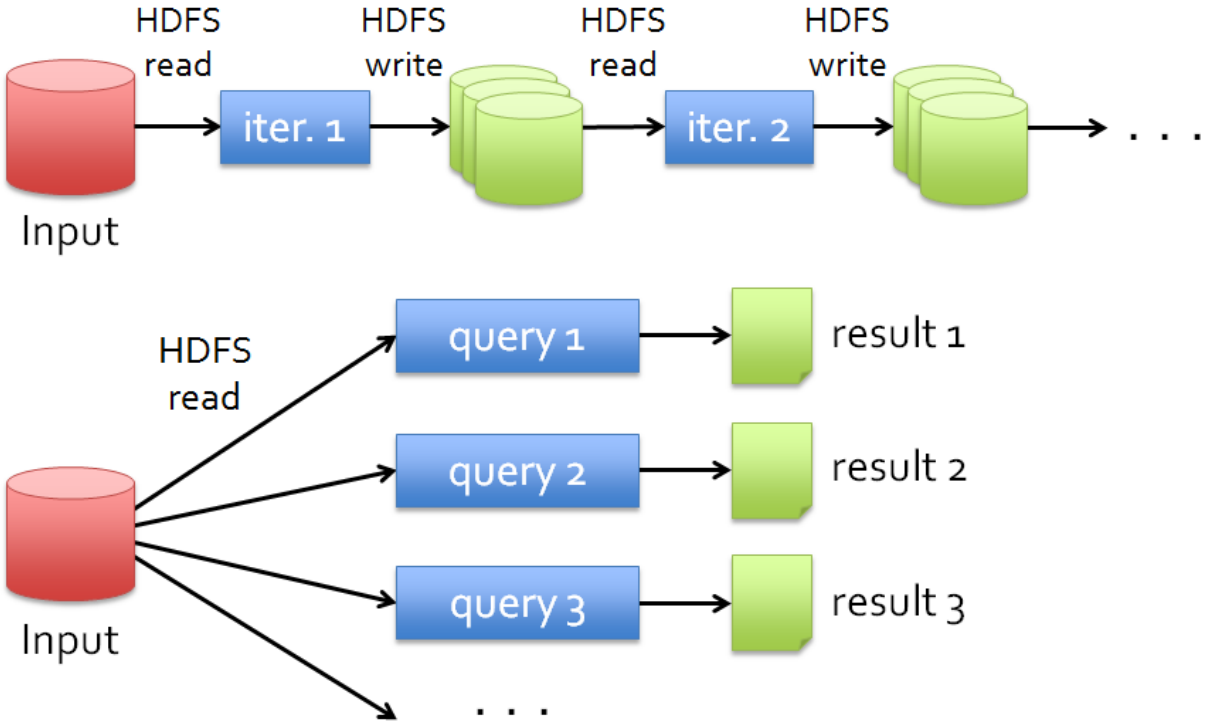
Spark: Limitations of MR

- As a general programming model:
  - It is more suitable for **one-pass** computation on a large dataset
  - Hard to **compose** and **nest** multiple operations
  - No means of expressing **iterative** operations
- As implemented in Hadoop
  - All datasets are read from **disk**, then stored back on to **disk**
  - All data is (usually) **triple-replicated** for reliability
  - Not easy to write MapReduce programs using **Java**



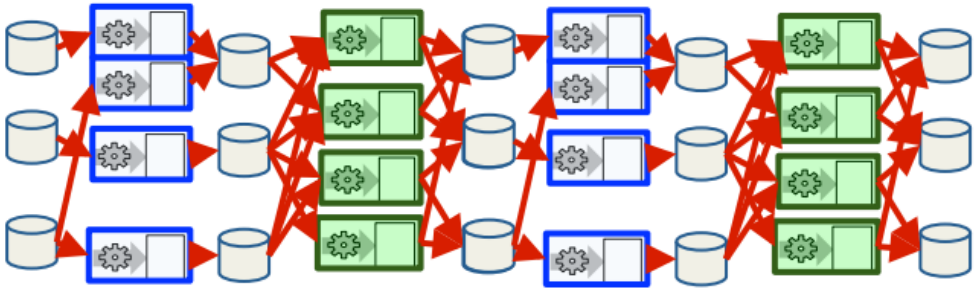
Spark: Data Sharing in MR

- Slow due to **replication**, **serialization**, and **disk IO**
- Complex apps, streaming, and interactive queries all need one thing that MapReduce lacks:
  - **Efficient primitives for data sharing**

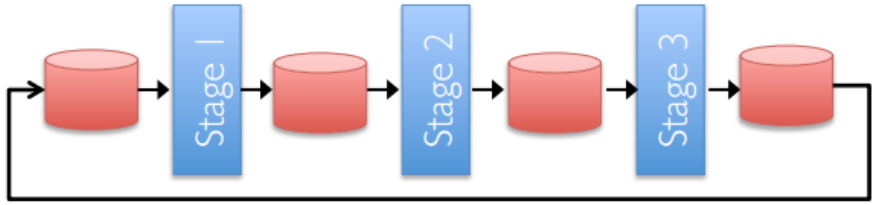


Spark: Data Sharing in MR

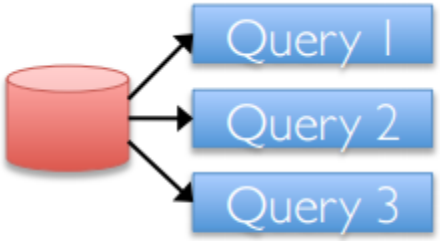
- **Iterative jobs** involve a lot of disk I/O for each repetition



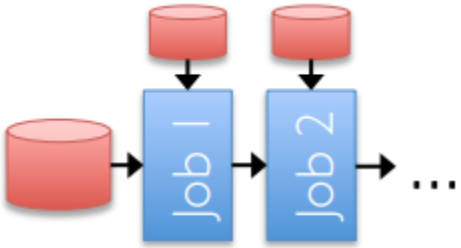
Disk I/O is very slow!



- **Interactive queries** and **online processing** involves lots of disk I/O



Interactive mining



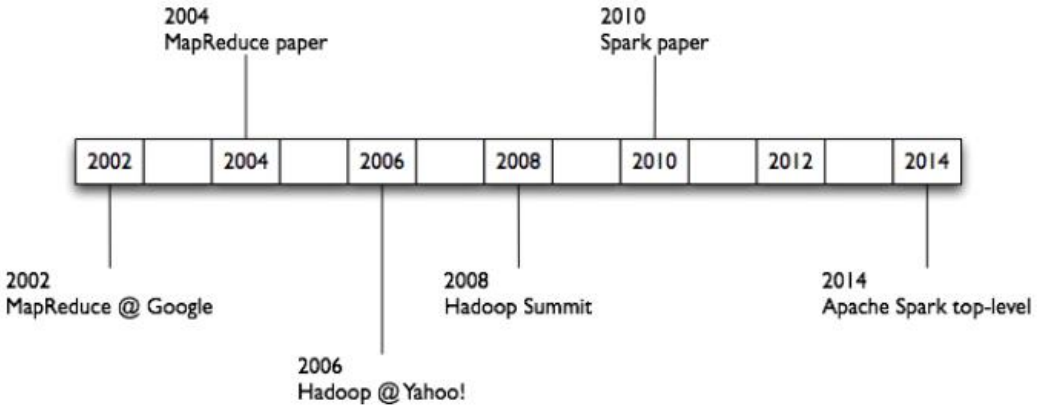
Stream processing

Spark: Goals

- Keep more data **in-memory** to improve the performance!
- Extend the MapReduce model to better support two common classes of analytics apps:
  - **Iterative** algorithms (machine learning, graphs)
  - **Interactive** data mining
- Enhance **programmability**:
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter

Spark: What is Spark

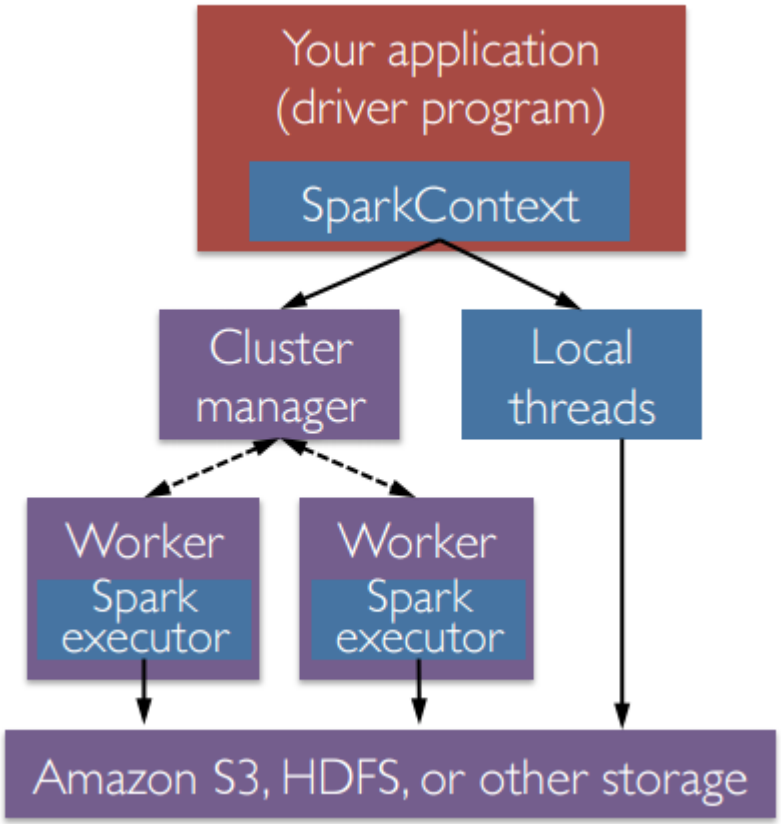
- One popular answer to “What’s beyond MapReduce?”
- Open-source engine for large-scale data processing
  - Supports generalized dataflows
  - Written in Scala, with bindings in Java and Python
- Spark:
  - is not a modified version of Hadoop
  - is dependent on Hadoop because it has its own cluster management
  - uses Hadoop for storage purpose only



Spark: Ideas

- **Expressive** computing system, not limited to map-reduce model
- Facilitate system **memory**
  - avoid saving intermediate results to disk
  - cache data for repetitive queries (e.g. for machine learning)
- Layer an in-memory system **on top of Hadoop**
- Achieve fault-tolerance by **re-execution** instead of replication

Spark: Components



- A Spark program first creates a SparkContext object
  - Tells Spark how and where to access a cluster
  - Connect to several types of cluster managers (e.g., YARN or its own manager)
- Cluster manager:
  - Allocate resources across applications
- Spark executor:
  - Run computations
  - Access data storage

Spark: Key Solution

- Resilient Distributed Dataset (RDD)
  - **Distributed collections of objects** that can be cached in memory across cluster
  - Manipulated through **parallel operators**
  - **Automatically recomputed** on failure based on lineage
  - RDDs can express many parallel algorithms, and capture many current programming models
    - Data flow models: MapReduce, SQL, …
    - Specialized models for iterative apps: Pregel, …

Spark: RDD

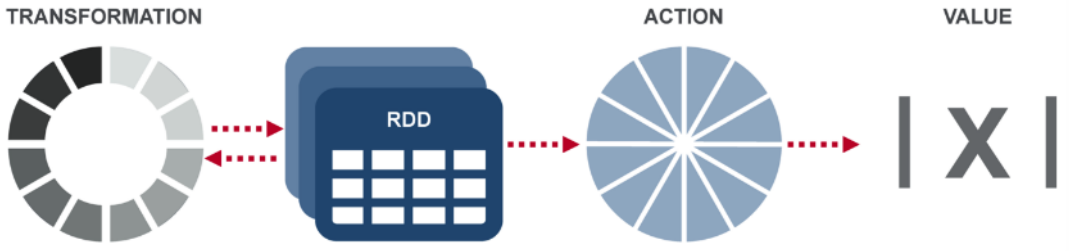
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, et al. NSDI'12
  - **RDD** is a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.
- **Resilient**: Fault-tolerant, is able to recompute missing or damaged partitions due to node failures.
- **Distributed**: Data residing on multiple nodes in a cluster.
- **Dataset**: A collection of partitioned elements, e.g. tuples or other objects (that represent records of the data you work with).

Spark: Traits

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable or Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed**, i.e. values in a RDD have types, e.g. RDD[Long] or RDD[(Int, String)].
- **Partitioned**, i.e. the data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

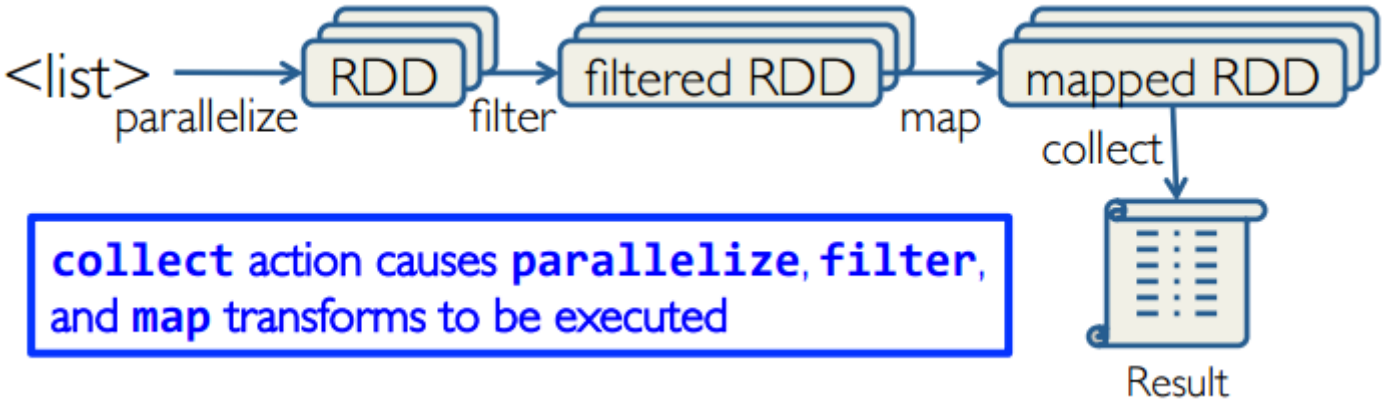
Spark: RDD Operations

- **Transformation**: returns a new RDD.
  - Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.
  - Transformation functions include *map*, *filter*, *flatMap*, *groupByKey*, *reduceByKey*, *aggregateByKey*, *filter*, *join*, *etc*.
- **Action**: evaluates and returns a new value.
  - When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
  - Action operations include *reduce*, *collect*, *count*, *first*, *take*, *countByKey*, *foreach*, *saveAsTextFile*, *etc*.



Spark: Working with RDDs

- Create an RDD from a data source
  - by parallelizing existing Python collections (lists)
  - by transforming an existing RDDs
  - from files in HDFS or any other storage system
- Apply transformations to an RDD: e.g., *map*, *filter*
- Apply actions to an RDD: e.g., *collect*, *count*



- Users can control two other aspects:
  - Persistence
  - Partitioning

Spark: vs. Hadoop MapReduce

- **Performance**: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it often needs lots of memory to do well; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- **Ease of use**: Spark is easier to program
- **Data processing**: Spark is more general
- **Maturity**: Spark maturing, Hadoop MapReduce mature

"Spark vs. Hadoop MapReduce" by Saggi Neumann (November 24, 2014)
https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/

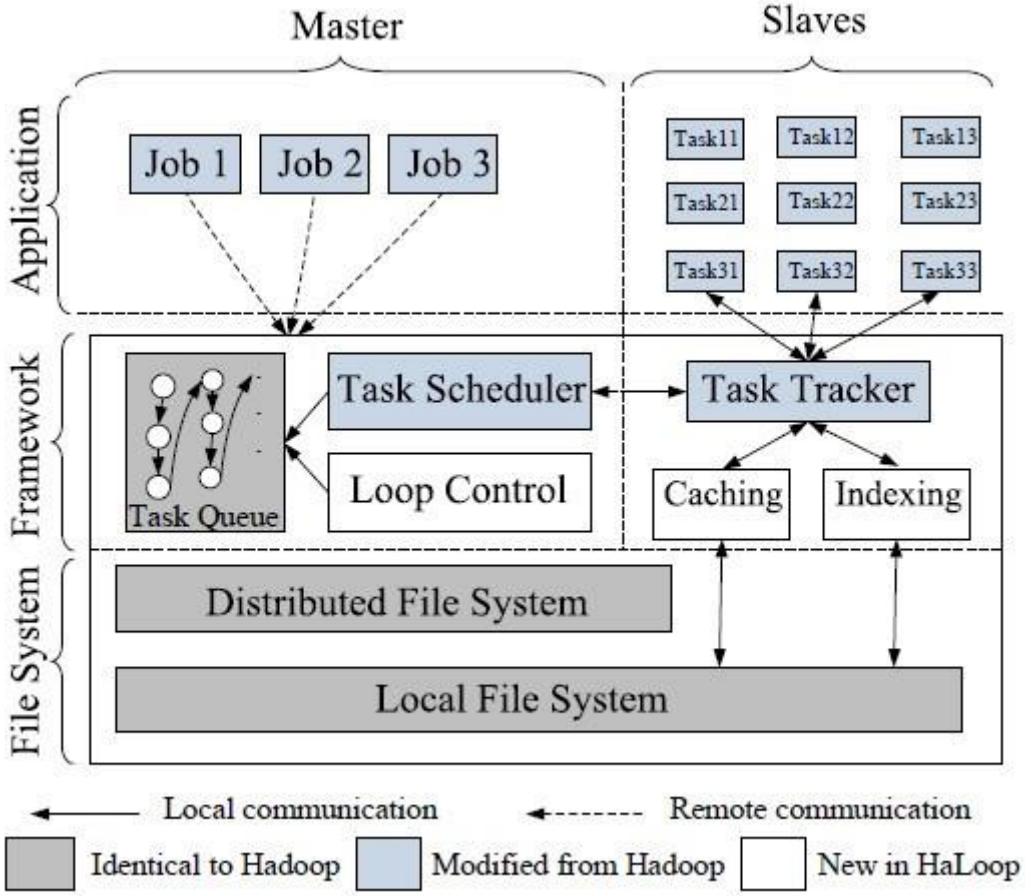
Iterative MapReduce

Naïve Implementation

- Straight forward
  - Join multiple MapReduce jobs into a job chain
  - One job after another
- User driver
  - Termination criteria
    - Maximum iteration numbers
    - Threshold (e.g. difference between the results of two successive iterations)
  - Input / Output
    - The output of last iteration as the input of the next iteration
- Improved frameworks
  - HaLoop
  - Twister

HaLoop

Architecture

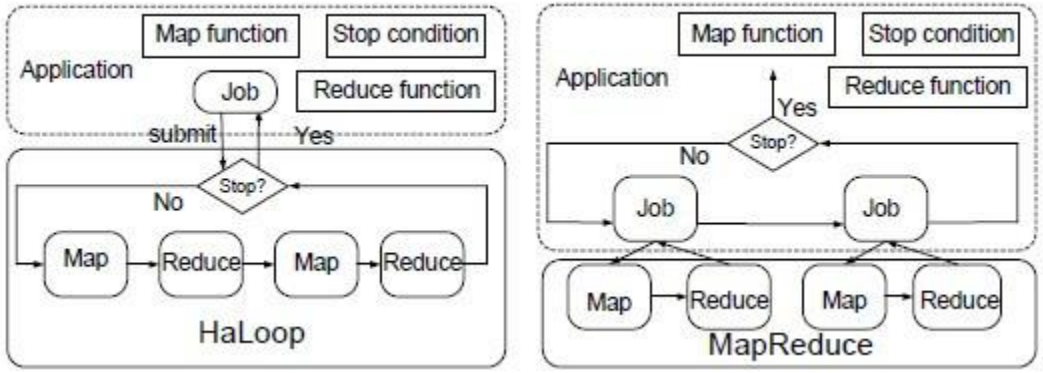


Improvement

- New programming interface for iterations

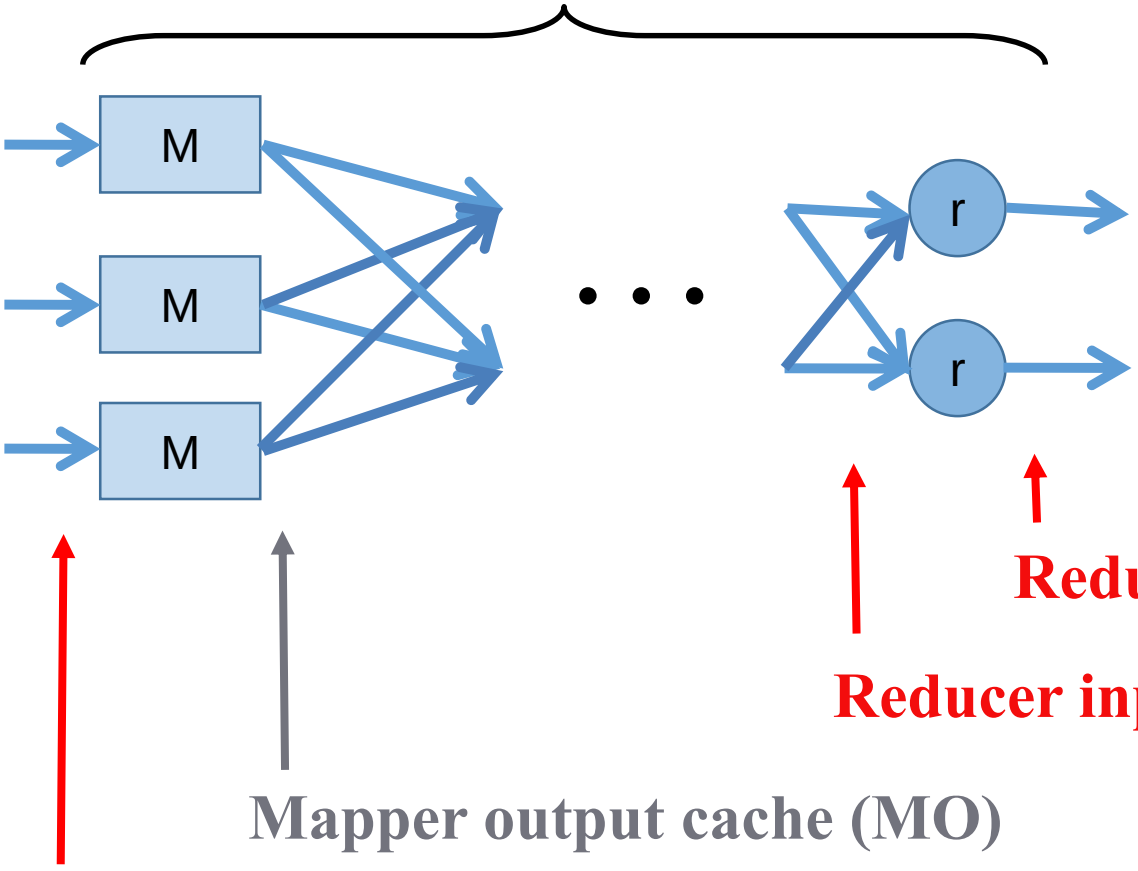
$$R_{i+1} = R_0 \cup (R_i \bowtie L)$$

- Master controls the iteration in the MR job



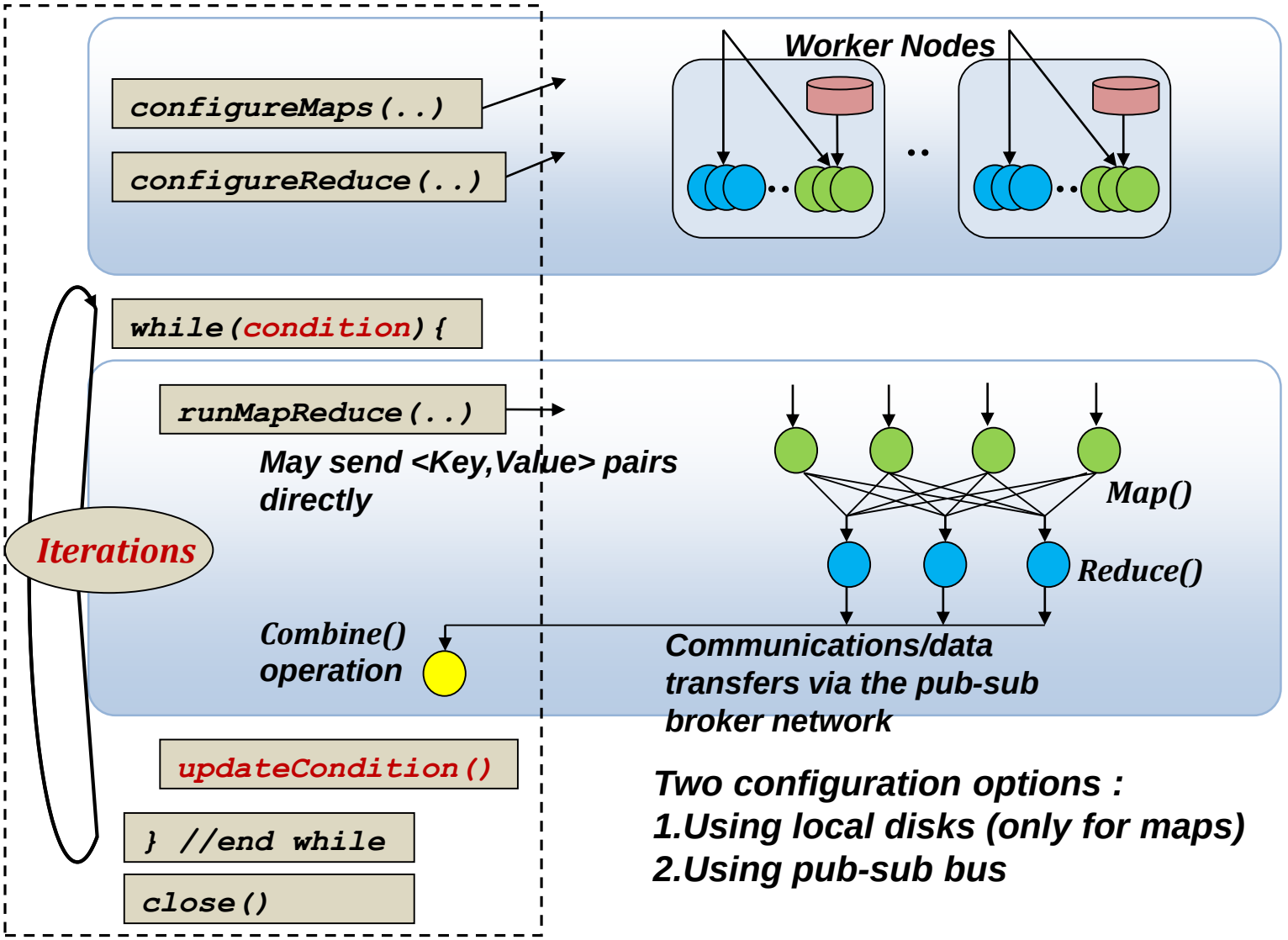
- Data locality is important for task scheduler
- Data is cached and index by the slave nodes

Approach: Inter-iteration caching



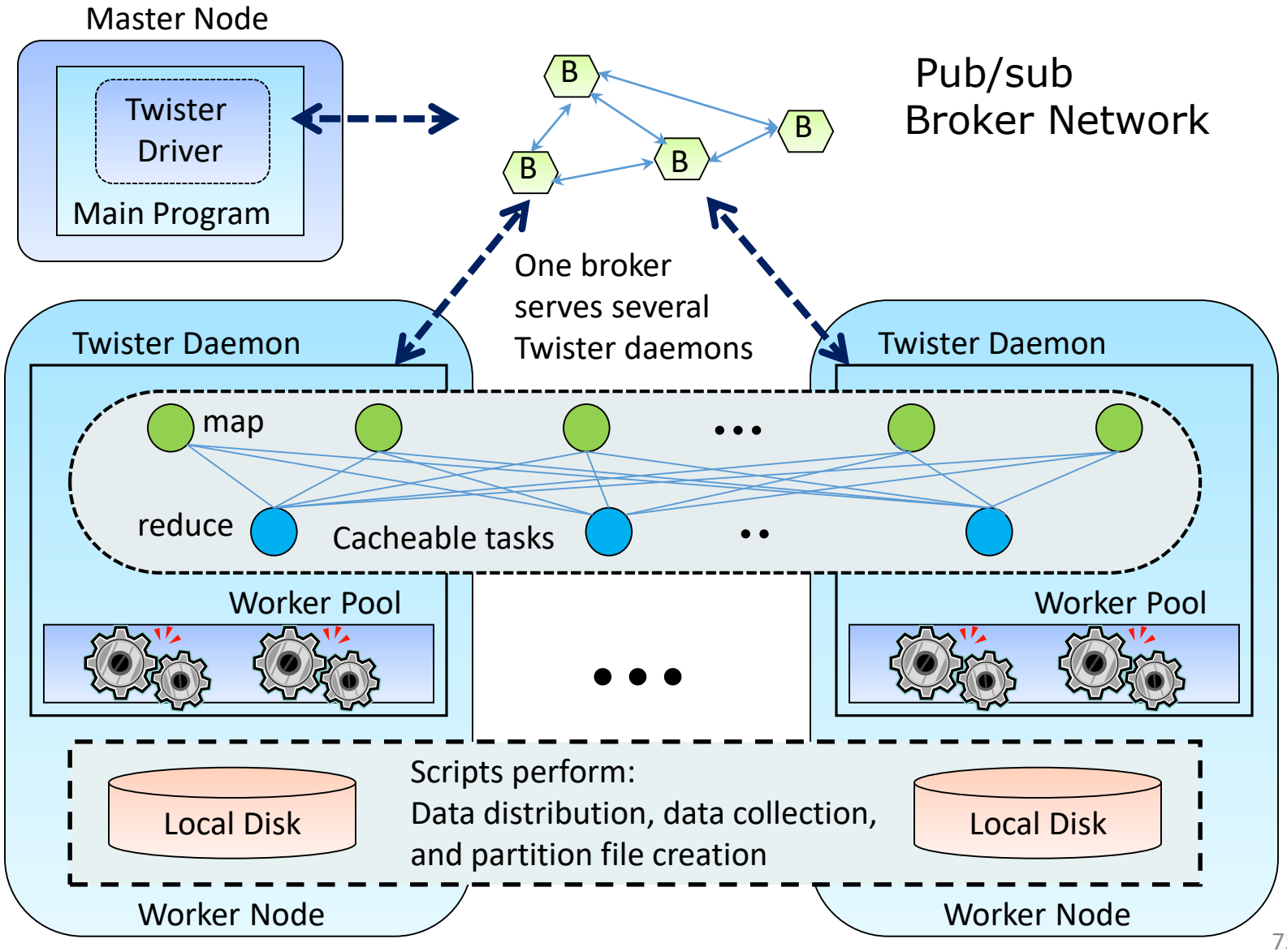
Twister

Twister Programming Model

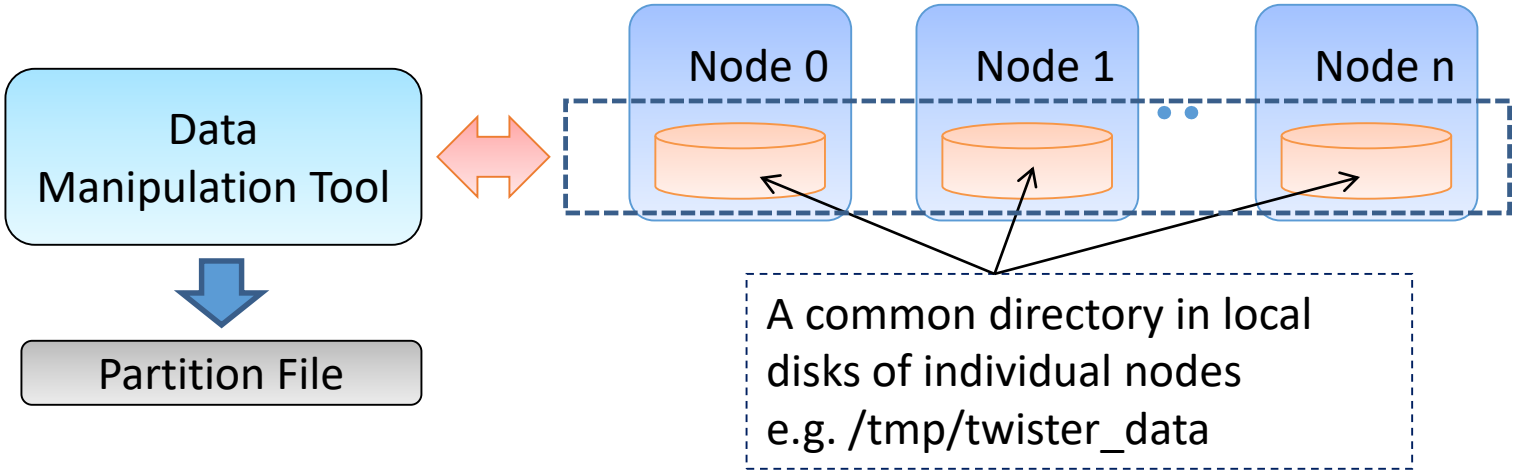


Two configuration options :
1.Using local disks (only for maps)
2.Using pub-sub bus

Twister Architecture



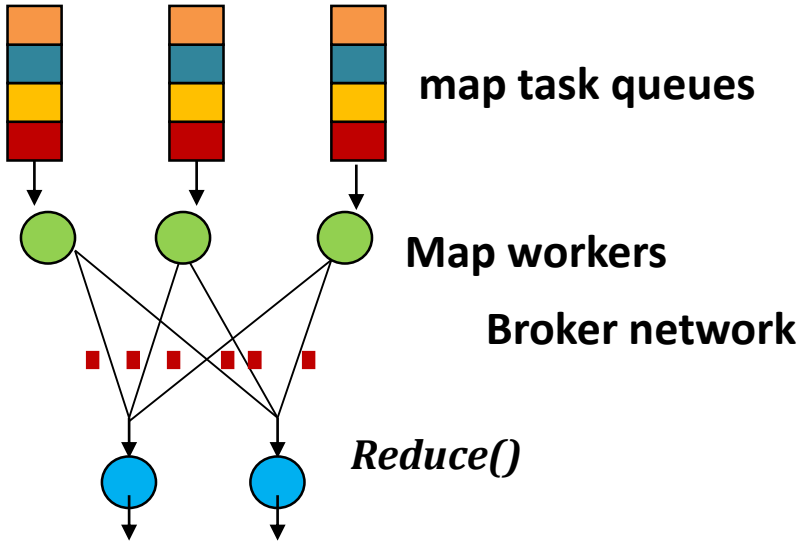
Input/Output Handling



- Data Manipulation Tool:
  - Provides basic functionality to manipulate data across the local disks of the compute nodes
  - Data partitions are assumed to be files (Contrast to fixed sized blocks in Hadoop)
  - Supported commands:
    - mkdir, rmdir, put,putall,get,ls,
    - Copy resources
    - Create Partition File

The use of pub/sub messaging

- Intermediate data transferred via the broker network
- Network of brokers used for load balancing
  - Different broker topologies
- Interspersed computation and data transfer minimizes large message load at the brokers
- Currently supports
  - NaradaBrokering
  - ActiveMQ



End of Chapter 5