



# **CS5412 / LECTURE 20**

## **APACHE ARCHITECTURE**

**Ken Birman & Kishore  
Pusukuri, Spring 2021**

# BATCHED, SHARDED COMPUTING ON BIG DATA WITH APACHE



Last time we heard about ways of computing on collections, but only touched on the issue that with big data this computing needs to be spread over many shards (IoT will make things even bigger!).

Today's non-IoT systems shard the data and store it in files or other forms of databases. We use SQL expressions in massively parallel programs to search or compute functions over these huge data sets.

Apache is the most widely used big data processing framework

# IN STANDARD BIG DATA SYSTEMS, BATCHING IS THE KEY. WHY BATCH?

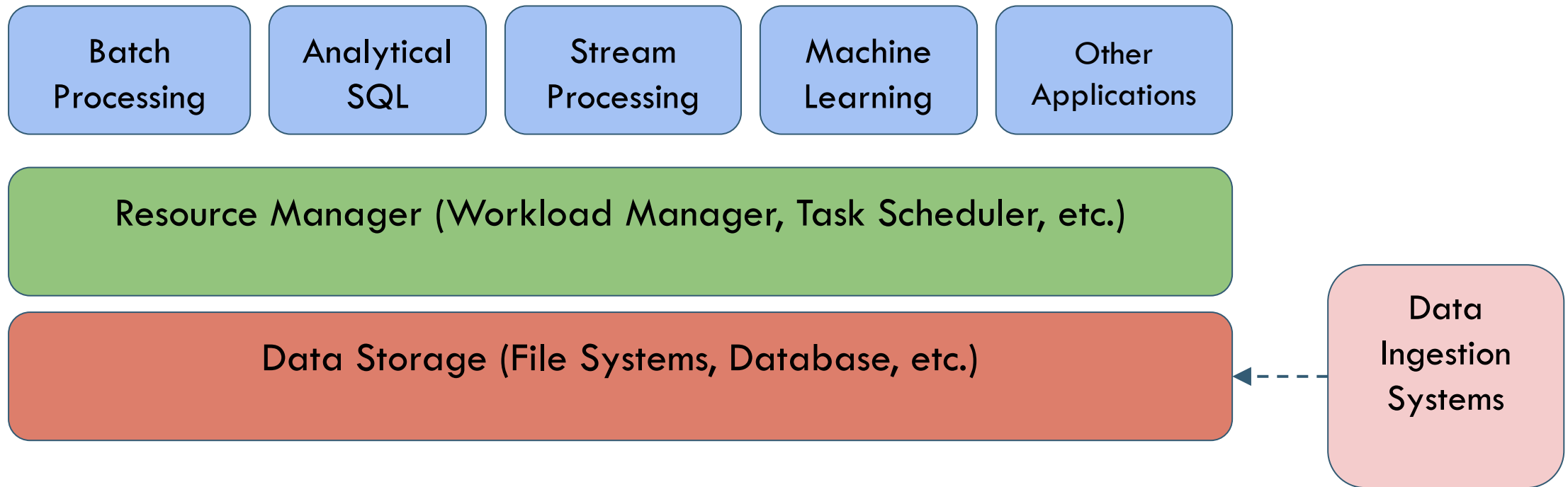
The core issue is overhead. Doing things one by one incurs high overheads.

Updating data in a batch pays the overhead once on behalf of many events, hence we “amortize” those costs. The advantage can be huge.

But batching must accumulate enough individual updates to justify running the big parallel batched computation. Tradeoff: *Delay versus efficiency.*

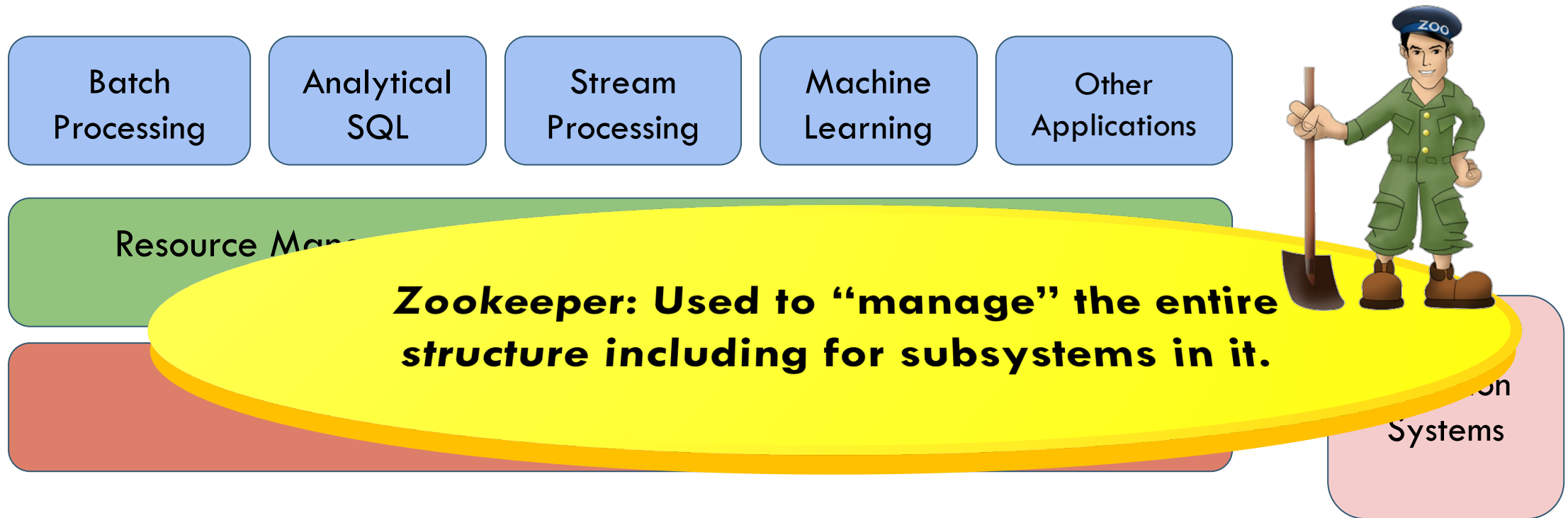
***For IoT, this delay is a big concern... motivating projects like Cascade.***

# A TYPICAL BIG DATA SYSTEM FOR BATCHED WORKLOADS



Popular BigData Systems: **Apache Hadoop, Apache Spark**

# A TYPICAL BIG DATA SYSTEM



Popular BigData Systems: **Apache Hadoop, Apache Spark**

# SEVERAL MODELS EMPLOYED

**Kafka:** For event-streams ([check out the cool video here](#)). Events are captured into logs (“queues”), and applications can filter on topics (“event names”) to consume specific events of interest.

**Databases:** They receive updates and hold data and support full SQL.

**File systems:** They hold documents, web pages, etc. You use a file system API.

**Key-value stores:** They hold (key,value) tuples, perhaps with versioning (logs with one record per version). They use the NoSQL model.

# UNIFYING IDEAS

All are built in similar ways. In Apache, all are structured “around” Zookeeper. Each is optimized for a particular style of use.

All use sharding for scalability.

And in fact all are accessible through similar APIs – most Apache tools support more than one kind of API (for example, as a file system, or as a DHT – but with the same contents in both cases).

# APACHE HAS MULTIPLE VERSIONS OF SOME ELEMENTS. FOR EXAMPLE, “FILE SYSTEMS”

Before we discuss Zookeeper, let's think about file systems. *Clouds have many!* One is for bulk storage: some form of “global file system” or GFS.

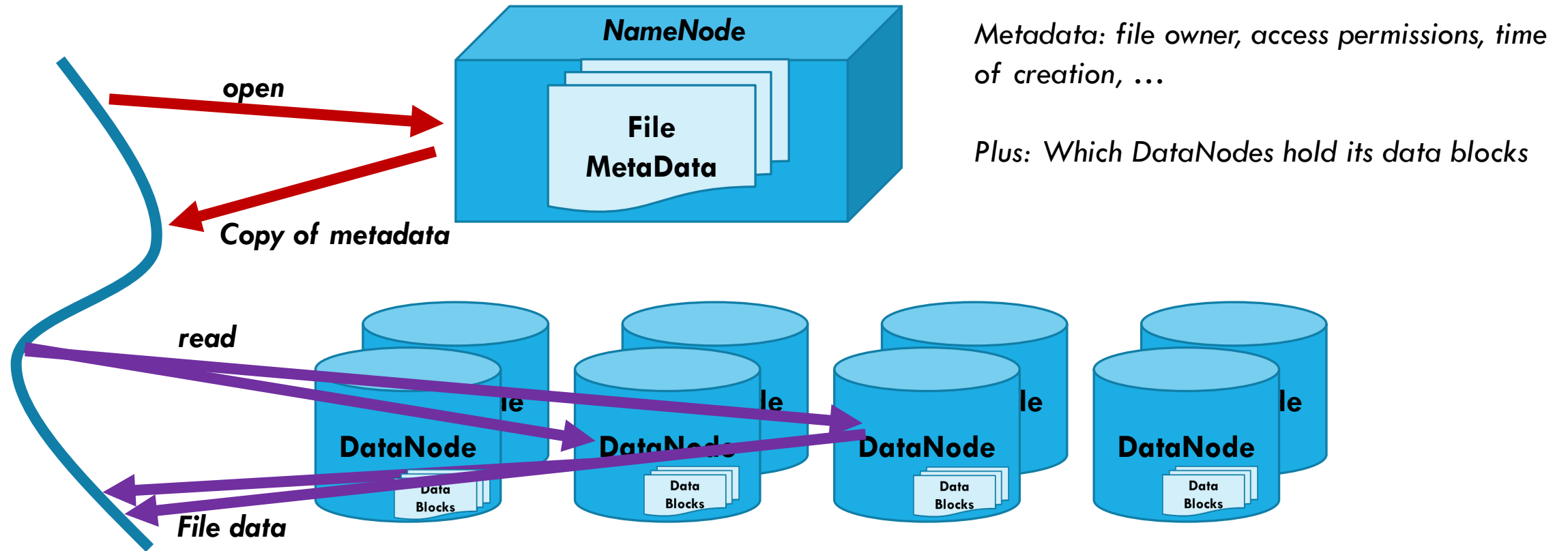
- At Google, it is actually called GFS. HDFS (which we will study) is an open-source version of GFS.
- At Amazon, S3 plays this role
- Azure uses “Azure storage fabric”
- Derecho can be used as a file system too (Cascade service)



# HOW DO THEY (ALL) WORK?

- A “Name Node” service runs, fault-tolerantly, and tracks file meta-data (like a Linux inode): Name, create/update time, size, seek pointer, etc.
- The name node also tells your application which data nodes hold the file.
- Very common to use a simple DHT scheme to fragment the NameNode into subsets, hopefully spreading the work around. DataNodes are hashed at the block level (large blocks)
- Some form of primary/backup scheme for fault-tolerance, like chain replication. Writes are automatically forwarded from the primary to the backup.

# HOW DO THEY WORK?



# **MANY FILE SYSTEMS THAT SCALE REALLY WELL AREN'T GREAT FOR LOCKING/CONSISTENCY**

The majority of sharded and scalable file systems turn out to be slow or incapable of supporting consistency via file locking, for many reasons.

So many application use two file systems: one for bulk data, and Zookeeper for configuration management, coordination, failure sensing.

This permits some forms of consistency even if not everything.

# APACHE ZOOKEEPER AND $\mu$ -SERVICES

Zookeeper can manage information in your system

IP addresses, version numbers, and other configuration information of your  $\mu$ -services.

The health of the  $\mu$ -service.

The step count for an iterative calculation.

Group membership



# ZOOKEEPER USE CASES

The need in many systems is for a place to store configuration, parameters, lists of which machines are running, which nodes are “primary” or “backup”, etc.

We desire a file system interface, but “strong, fault-tolerant semantics”

Zookeeper is widely used in this role. Stronger guarantees than GFS.

- Data lives in (small) files. Versioning is automatic (it can keep every version, or just the most recent version).
- **Zookeeper is quite slow and not very scalable.** Facebook has reimplemented it and has a faster, API-compliant version in-house.

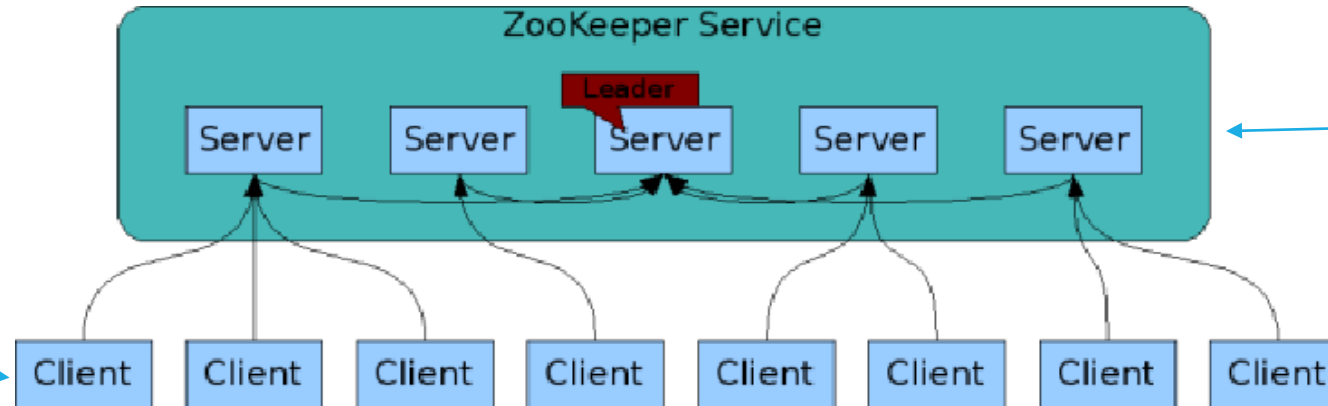
# MOST POPULAR ZOOKEEPER API?

They offer a novel form of “conditional file replace”

- Exactly like the conditional “put” operation in Cascade.
- Files have version numbers in Zookeeper.
- A program can read version 5, update it, and tell the system to replace the file *creating version 6*. But this can fail if there was a race and you lost the race. You could would just loop and retry from version 6.
- It avoids the need for locking and this helps Zookeeper scale better.

# THE ZOOKEEPER SERVICE

Each  $\mu$ -service has a leader that talks to Zookeeper. Its other nodes have connections too, but just passive (for fault detection)



Zookeeper is itself an interesting distributed system

ZooKeeper Service is replicated over a set of machines, usually 5 or 7.

All machines store a copy of the data **in memory** (!). Checkpointed to disk if you wish.

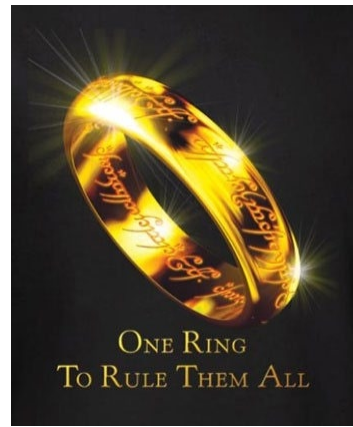
A leader is elected on service startup

Clients only connect to a single ZooKeeper server & maintains a TCP connection.

Client can read from any Zookeeper server.

Writes go through the leader & need majority consensus.

# IS ZOOKEEPER USING PAXOS?



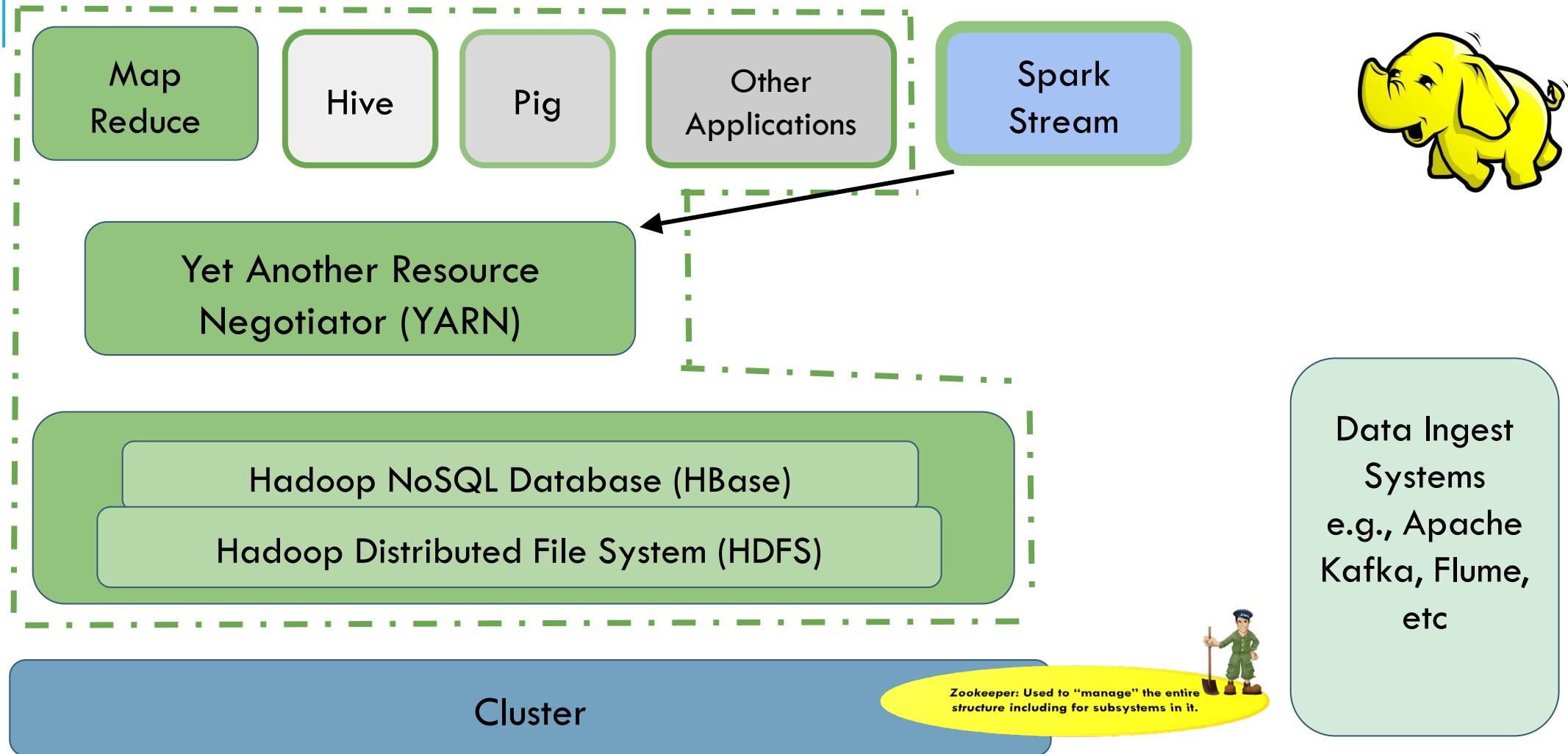
Early work on Zookeeper actually did use Paxos, but it was too slow

They settled on a model that uses atomic multicast with dynamic membership management and in-memory data (like virtual synchrony).

But they also checkpoint Zookeeper every 5s if you like (you can control the frequency), so if it crashes it won't lose more than 5s of data.



# REST OF THE APACHE HADOOP ECOSYSTEM



# HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

HDFS is the storage layer for Hadoop BigData System

HDFS is based on the Google File System (GFS)

Fault-tolerant distributed file system

Designed to turn a computing cluster (a large collection of loosely connected compute nodes) into a massively scalable pool of storage

Provides redundant storage for massive amounts of data -- scales up to 100PB and beyond

# HDFS: SOME LIMITATIONS

Files can be created, deleted, and you can write to the end, but not update them in the middle.

A big update might not be atomic (if your application happens to crash while writes are being done)

Not appropriate for real-time, low-latency processing -- have to close the file immediately after writing to make data visible, hence a real time task would be forced to create too many files

Centralized metadata storage -- multiple single points of failures

**Name node is a scaling (and potential reliability) weak spot.**

# HADOOP DATABASE (HBASE) IS A THINK LAYER DIRECTLY OVER HDFS

HBASE is used like a NoSQL database. It maps directly to HDFS

It holds tables. One table can have thousands of columns

Supports very large amounts of data and high throughput

HBase has a weak consistency model, but there are ways to use it safely

Random access, low latency

# HBASE

The Hbase design actually is based on Google's Bigtable

Designed for Distribution, Scale, and Speed

Full relational Database (RDBMS) vs NoSQL Database:

- RDBMS → vertical scaling (expensive) → not appropriate for BigData
- NoSQL → horizontal scaling / sharding (cheap) → better for BigData

# REMINDER: SQL VS NOSQL (1)

- Full RDMS systems support SQL, but Jim Gray's analysis applies
- So at scale we shard (the BASE methodology), lose ACID:
  - RDBMS (ACID): Atomicity, Consistency, Isolation, Durability
  - NoSQL (BASE): Basically Available Soft state Eventually consistency
- Giving up ACID enables availability, performance, and scalability
  - Most Apache subsystems call themselves “eventually consistent”, meaning that updates are eventually propagated to all nodes

## RDBMS VS NOSQL (2)

- NoSQL (e.g., CouchDB, HBase) is a good choice for 100 Millions/Billions of rows
- RDBMS (e.g., mysql) is a good choice for a few thousand/millions of rows
- HBase is a NoSQL table store and actually is “consistent” but only if used in specific ways.

# HBASE: DATA MODEL (1)

## Data model

Columns

Row key	info:name	info:age	comp:base	comp:stocks
121	'tom'	'28'	'125k'	
145	'bob'	'32'	'110k'	'50' (ts=2012) '100' (ts=2014)

Cells

Row keys



# HBASE: DATA MODEL (2)

- Sorted rows: support billions of rows
- Columns: Supports millions of columns
- Cell: intersection of row and column
  - Can have multiple values (which are time-stamped)
  - Can be empty. No storage/processing overheads

# HBASE: TABLE

Unique id	Name	price	weight	store1	store2	store3
"1000000"	snickers	\$9.99	4 Oz	Yes	Yes	Yes
"3000000"	almonds	\$9.99	8 Oz	Yes	No	Yes
"8000000"	coke	\$9.99	16 Oz	Yes	Yes	Yes
"4000000"	foo	\$34.63	16 Oz	No	Yes	Yes
"5000000"	bar	\$22.54	16 Oz	Yes	Yes	Yes
"9000000"	new1	\$2.5	16 Oz	Yes	Yes	Yes
"7000000"	new2	\$6.4	16 Oz	Yes	Yes	Yes
"2000000"	new3	\$6.4	16 Oz	Yes	Yes	Yes

# HBASE: HORIZONTAL SPLITS (REGIONS)

[ "", "5000000" )

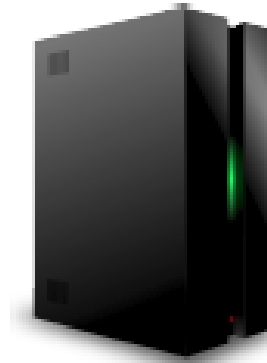
Row Key	Name	brand	price	weight	store1	store2	store3
"1000000"	snickers	xxx	\$9.99	4 Oz	Yes	Yes	Yes
"2000000"	new3	xxx	\$6.4	16 Oz	Yes	Yes	Yes
"3000000"	almonds	xxx	\$9.99	8 Oz	Yes	No	Yes
"4000000"	foo	xxx	\$34.63	16 Oz	No	Yes	Yes

[ "5000000", "" )

Row Key	Name	brand	price	weight	store1	store2	store3
"5000000"	bar	xxx	\$22.54	16 Oz	Yes	Yes	Yes
"7000000"	new2	xxx	\$6.4	16 Oz	Yes	Yes	Yes
"8000000"	coke	xxx	\$9.99	16 Oz	Yes	Yes	Yes
"9000000"	new1	xxx	\$2.5	16 Oz	Yes	Yes	Yes

# HBASE ARCHITECTURE (REGION SERVER)

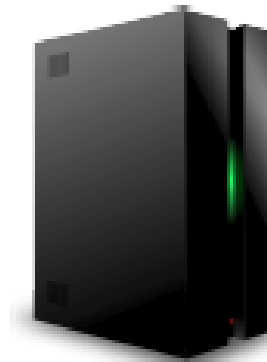
Row Key	Name	price	weight	....
"1000000"	snickers	\$9.99	4 Oz	....
"2000000"	new3	\$6.4	16 Oz	....
"3000000"	almonds	\$9.99	8 Oz	....
"4000000"	foo	\$34.63	16 Oz	....



Server 12

*Sharding is automatic*

Row Key	Name	price	weight	....
"5000000"	bar	\$22.54	16 Oz	....
"7000000"	new2	\$6.4	16 Oz	....
"8000000"	coke	\$9.99	16 Oz	....
"9000000"	new1	\$2.5	16 Oz	....



Server 7

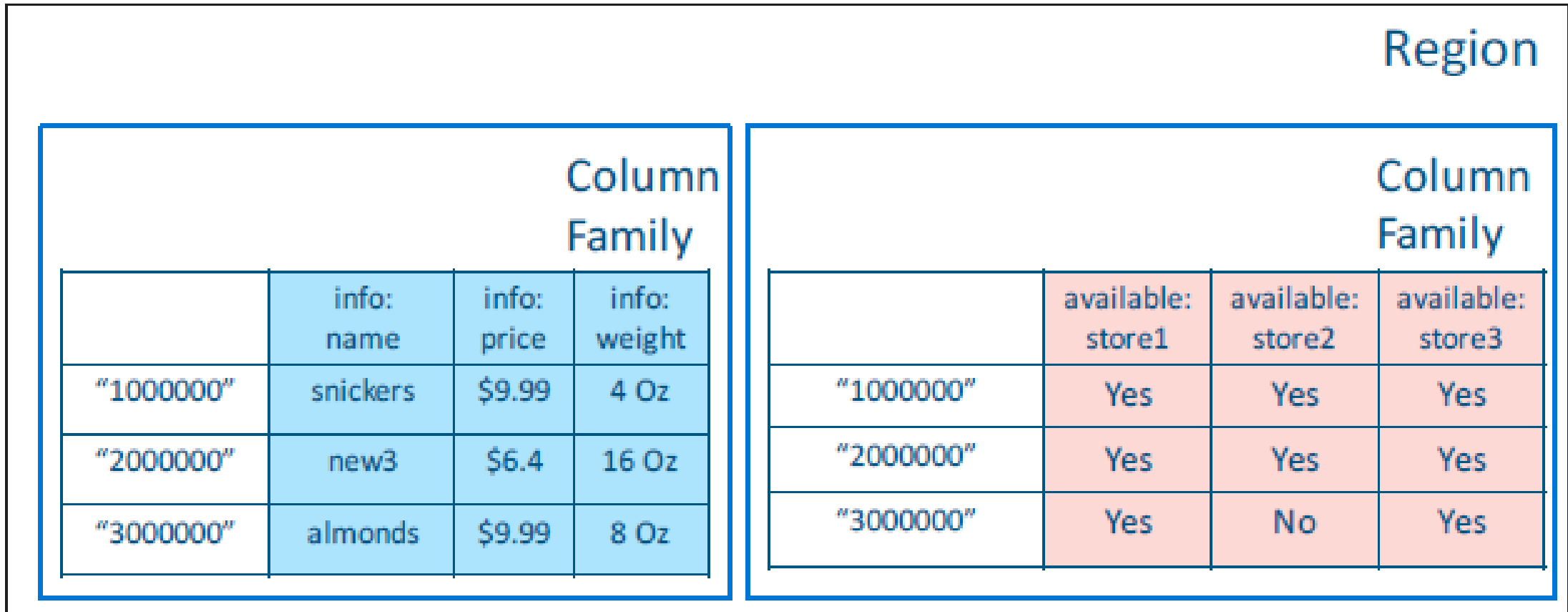
# HBASE ARCHITECTURE

Unique id	Name	price	weight	store1	store2	store3
"1000000"	snickers	\$9.99	4 Oz	Yes	Yes	Yes
"3000000"	almonds	\$9.99	8 Oz	Yes	No	Yes
"8000000"	coke	\$9.99	16 Oz	Yes	Yes	Yes
"4000000"	foo	\$34.63	16 Oz	No	Yes	Yes
"5000000"	bar	\$22.54	16 Oz	Yes	Yes	Yes
"9000000"	new1	\$2.5	16 Oz	Yes	Yes	Yes
"7000000"	new2	\$6.4	16 Oz	Yes	Yes	Yes
"2000000"	new3	\$6.4	16 Oz	Yes	Yes	Yes

# HBASE ARCHITECTURE: COLUMN FAMILY (1)

Row Key	info: name	info: price	info: weight	availability: store1	availability: store2	availability: store3
"1000000"	snickers	\$9.99	4 Oz	Yes	Yes	Yes
"2000000"	new3	\$6.4	16 Oz	Yes	Yes	Yes
"3000000"	almonds	\$9.99	8 Oz	Yes	No	Yes
"4000000"	foo	\$34.63	16 Oz	No	Yes	Yes
"5000000"	bar	\$22.54	16 Oz	Yes	Yes	Yes
"7000000"	new2	\$6.4	16 Oz	Yes	Yes	Yes
"8000000"	coke	\$9.99	16 Oz	Yes	Yes	Yes
"9000000"	new1	\$2.5	16 Oz	Yes	Yes	Yes

# HBASE ARCHITECTURE: COLUMN FAMILY



# HBASE ARCHITECTURE: COLUMN FAMILY (3)

- Data (column families) stored in separate files (Hfiles)
- Tune Performance
  - In-memory
  - Compression
- Needs to be specified by the user

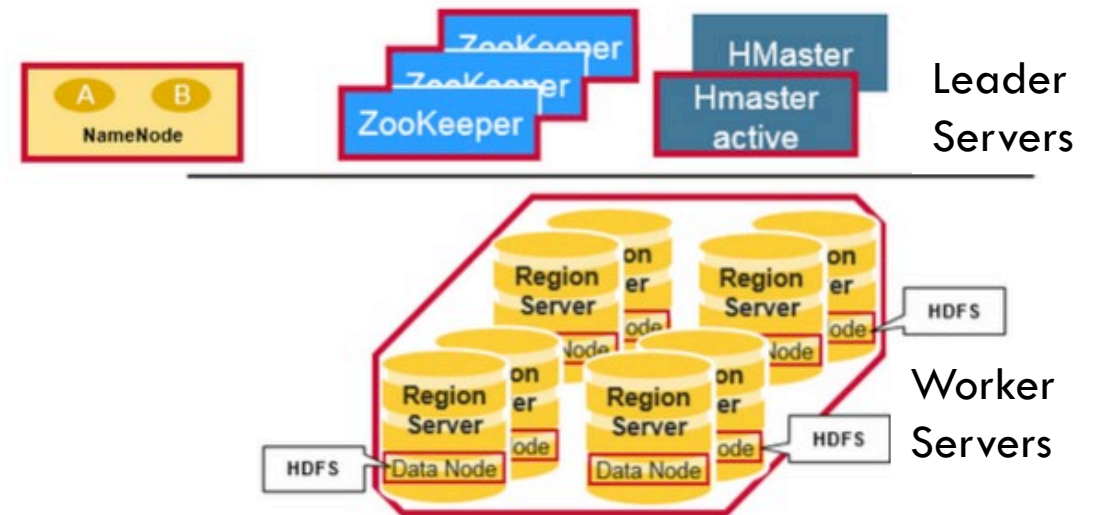


# HBASE ARCHITECTURE (1)

HBase is composed of three types of servers in a leader/worker type of architecture: **Region Server, Hbase Master, ZooKeeper**.

## Region Server:

- Clients communicate with RegionServers (workers) directly for accessing data
- Serves data for reads and writes.
- These region servers are assigned to the HDFS data nodes to preserve data locality.



# HBASE ARCHITECTURE (2)

**HBase Leader (HMaster):** coordinates region servers, handles DDL (create, delete tables) operations.

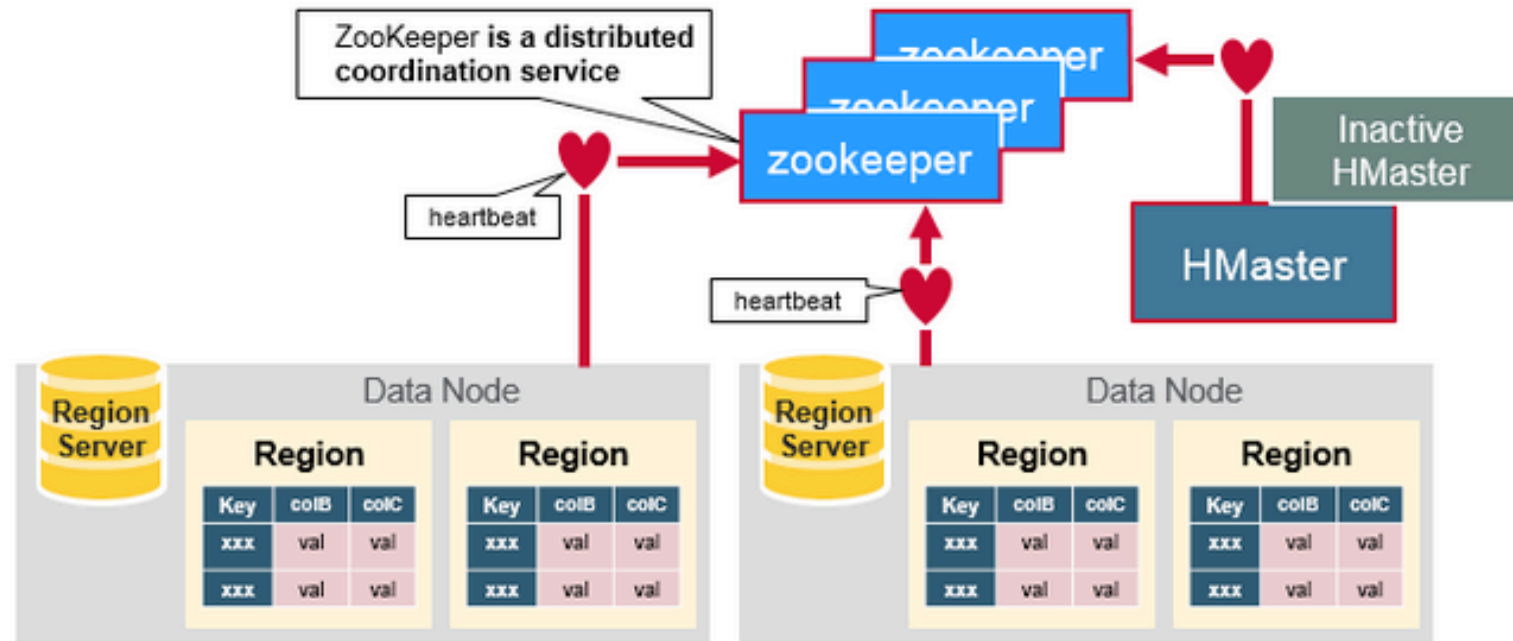
**Zookeeper:** HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster.

# HDFS USES ZOOKEEPER AS ITS COORDINATOR

Maintains region server state in the cluster

Provides server failure notification

Uses consensus to guarantee common shared state

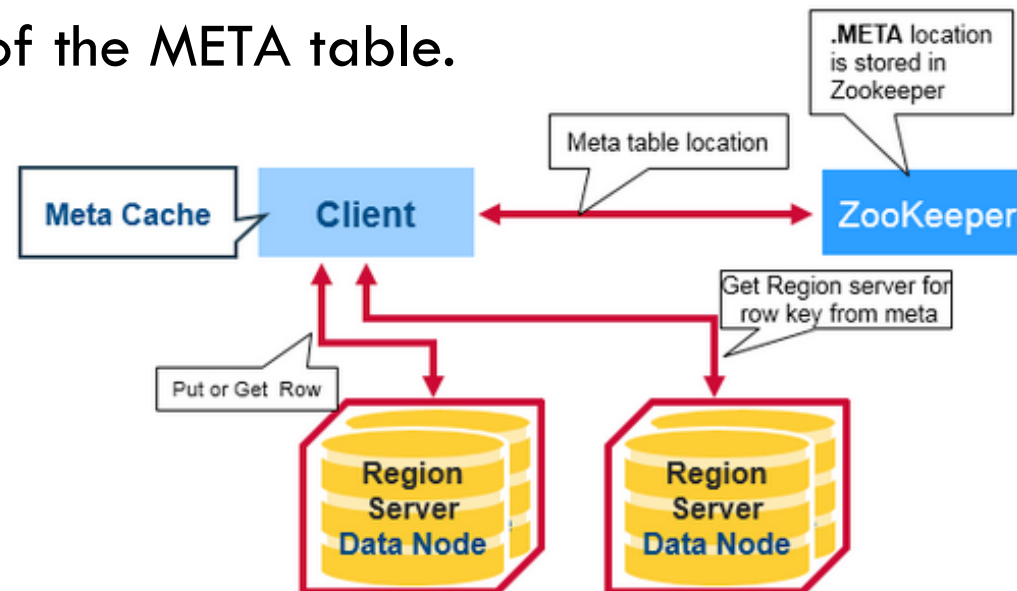


# HOW DO THESE COMPONENTS WORK TOGETHER?

Region servers and the active HBase Leaders connect with a session to ZooKeeper

A special HBase Catalog table “META table” → Holds the location of the regions in the cluster.

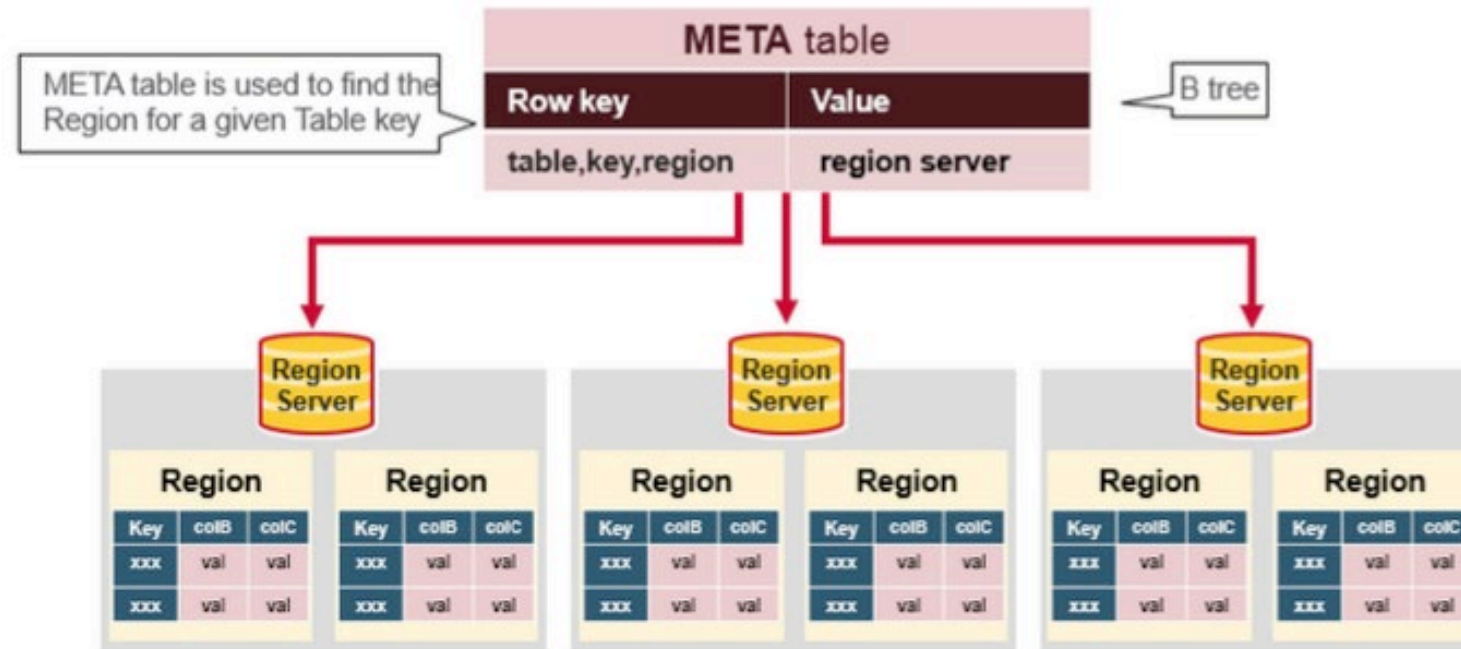
ZooKeeper stores the location of the META table.



# HBASE: META TABLE

The META table is an HBase table that keeps a list of all regions in the system.

This META table is like a B Tree

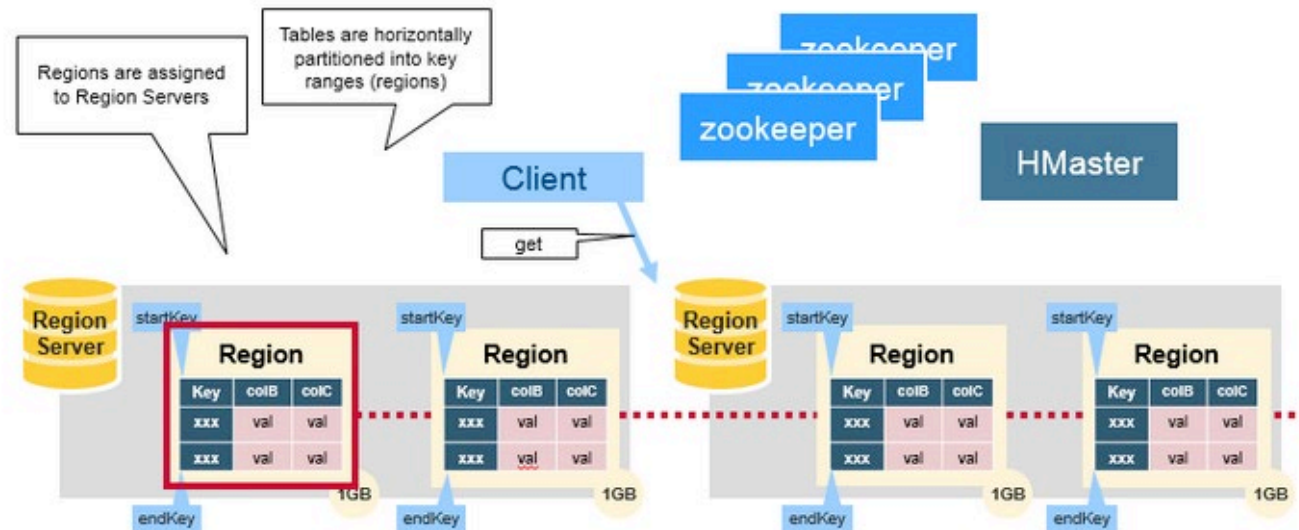


# HBASE: READS/WRITES

The client gets the Region server that hosts the META table from ZooKeeper

The client will **query (get/put)** the META server to get the region server corresponding to the **rowkey** it wants to access

It will get the Row from the corresponding Region Server.



# HBASE: SOME LIMITATIONS

Not ideal for large objects ( $>50\text{MB}$  per cell), e.g., videos -- the problem is “write amplification” -- when HDFS reorganizes data to compact large unchanging data, extensive copying occurs

Not ideal for storing data chronologically (time as primary index), e.g., machine logs organized by time-stamps cause write hot-spots.

# HBASE VS HDFS

Hbase is a way of “talking to” HDFS. We use it for massive tables that wouldn’t fit into a single HDFS file.

## HBase

- Stores data as key-value objs in column-families. Records in HBase are stored according to the rowkey and sequential search is common
- Provides low latency access to small amounts of data from within a large data set
- Provides flexible data model

## HDFS

- Stores data as flat files
- Optimized for streaming access of large files -- doesn’t support random read/write
- Follows write-once read-many model
- Supports log-style files (append-only).

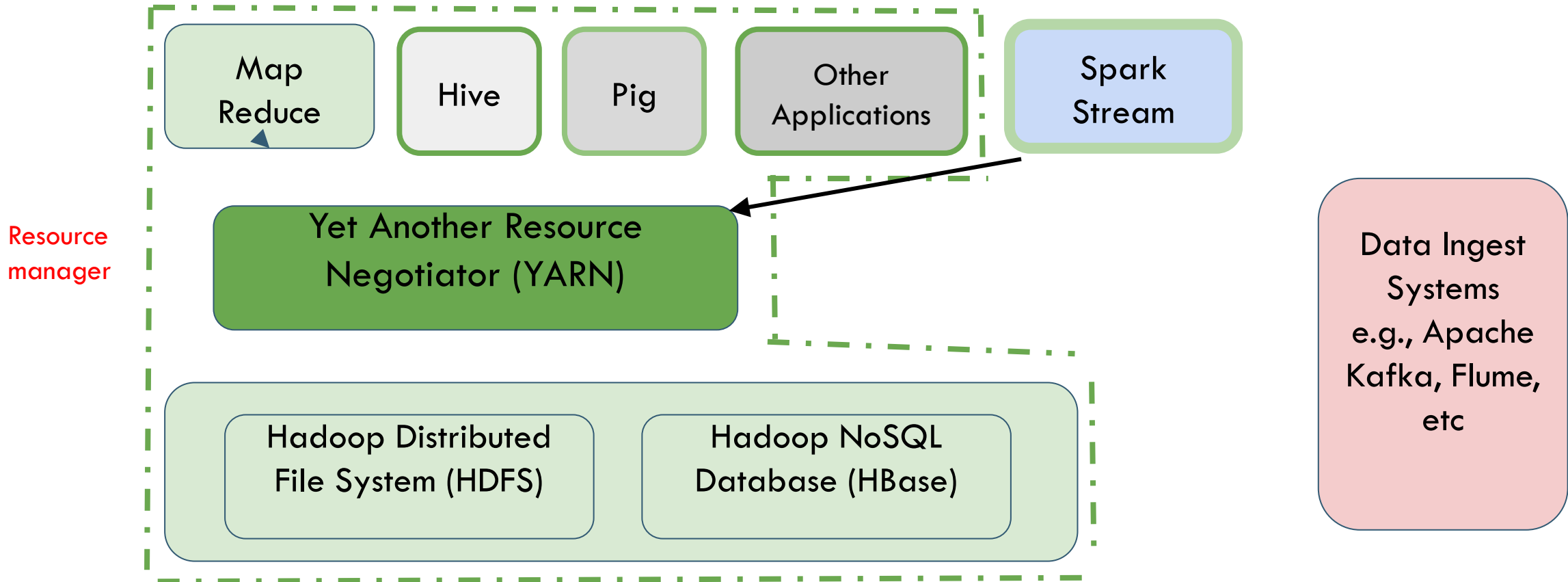


# HADOOP RESOURCE MANAGEMENT

## Yet Another Resource Negotiator (**YARN**)

- YARN is a core component of Hadoop, manages all the resources of a Hadoop cluster (CPUs, memory, GPUs, networking connections, etc).
- Using selectable criteria such as fairness, it effectively allocates resources of Hadoop cluster to multiple data processing jobs
  - Batch jobs (e.g., MapReduce, Spark)
  - Streaming Jobs (e.g., Spark streaming)
  - Analytics jobs (e.g., Impala, Spark)

# HADOOP ECOSYSTEM (RESOURCE MANAGER)



# YARN CONCEPTS (1)

## Container:

- YARN uses an abstraction of resources called a **container** for managing resources -- an unit of computation of a resource node, i.e., a certain amount of CPU, Memory, Disk, etc., ASIC resources. Tied to Mesos container model.
- A single job may run in one or more containers – a set of containers would be used to encapsulate highly parallel Hadoop jobs.
- The main goal of YARN is effectively allocating containers to multiple data processing jobs.

# YARN CONCEPTS (2)

Three Main components of YARN:

Application Leader, Node Manager, and Resource Manager (a.k.a. YARN Daemon Processes)

➤ **Application Leader:**

- Single instance per job.
- Spawned within a container when a new job is submitted by a client
- Requests additional containers for handling of any sub-tasks.

➤ **Node Manager:** Single instance per worker node. Responsible for monitoring and reporting on local container status (all containers on worker node).

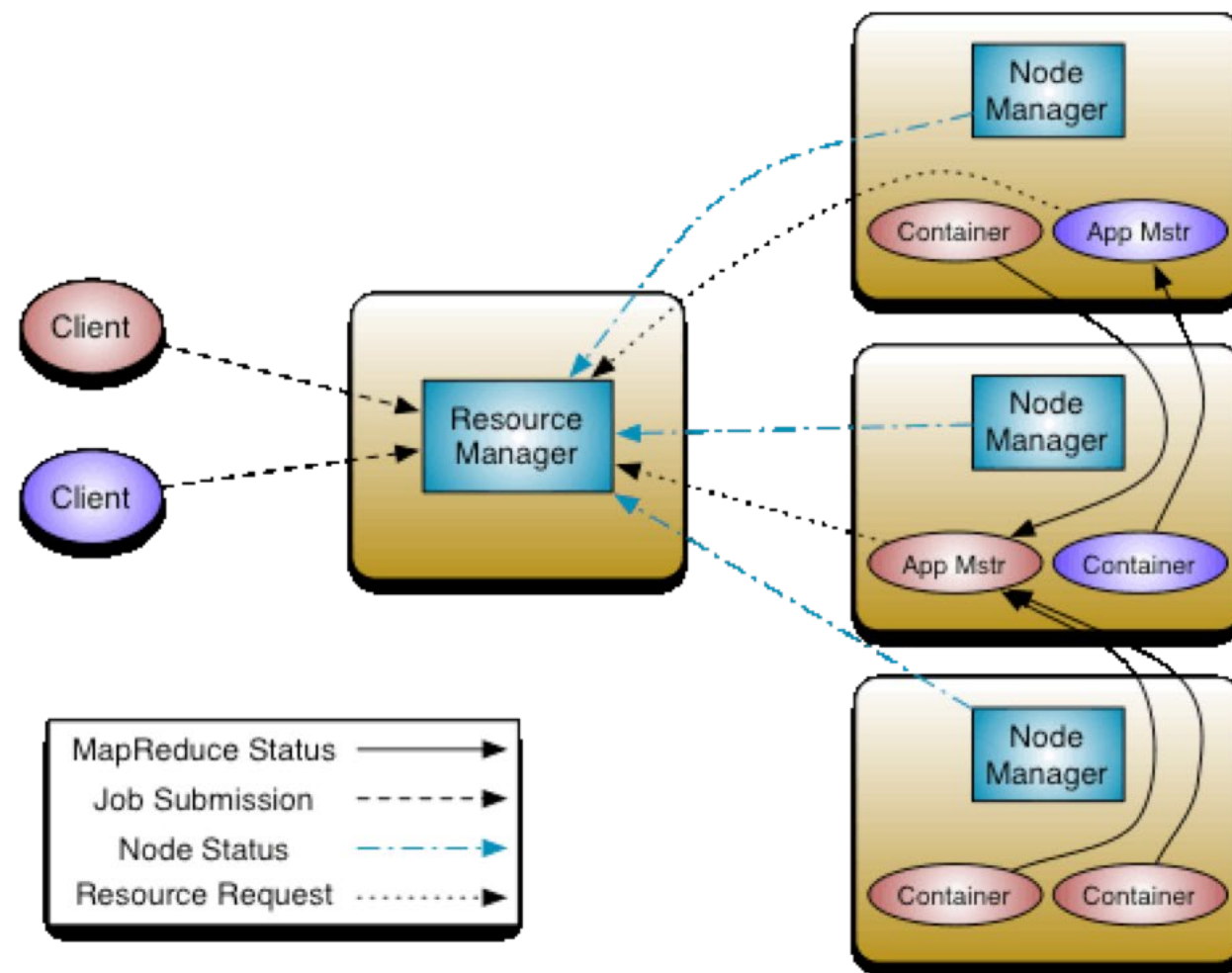
# YARN CONCEPTS (3)

Three Main components of YARN: Application Master, Node Manager, and Resource Manager (aka The YARN Daemon Processes)

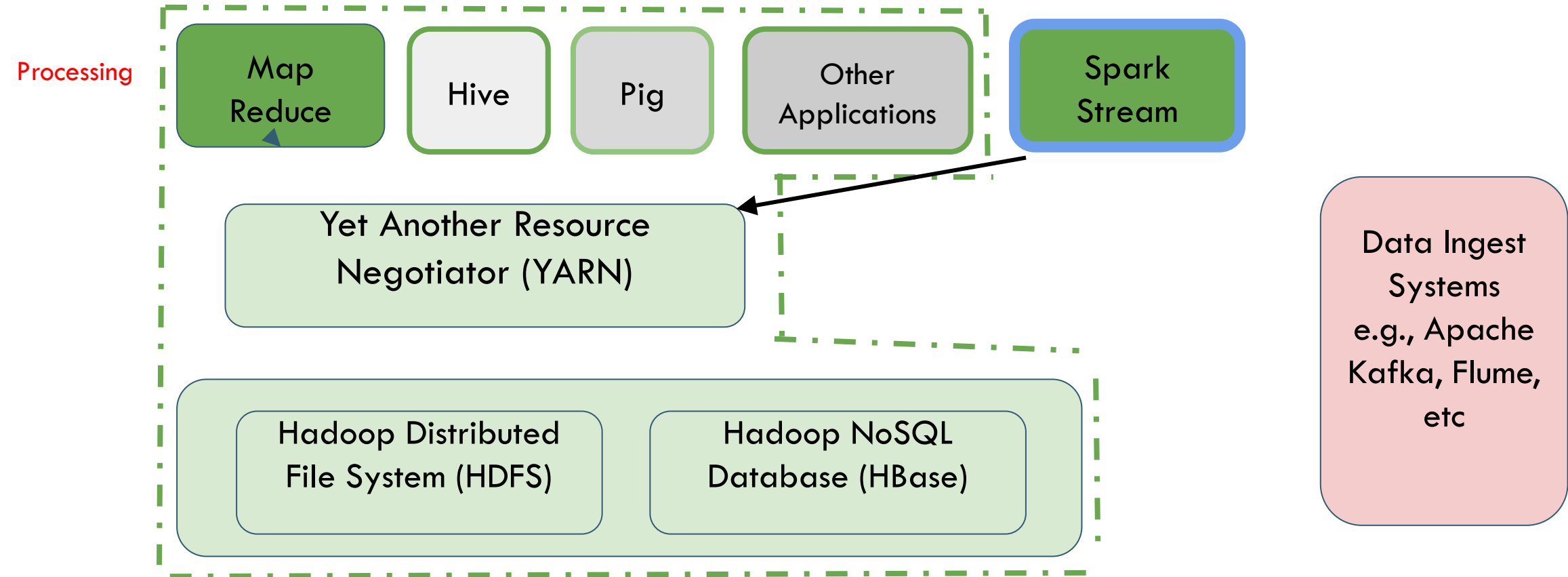
- **Resource Manager**: arbitrates system resources between competing jobs. It has two main components:
  - **Scheduler** (Global scheduler): Responsible for allocating resources to the jobs subject to familiar constraints of capacities, queues etc.
  - **Application Manager**: Responsible for accepting job-submissions and provides the service for restarting the ApplicationMaster container on failure.

# YARN CONCEPTS (4)

How do the components of YARN work together?



# HADOOP ECOSYSTEM (PROCESSING LAYER)



# HADOOP DATA PROCESSING FRAMEWORKS

Hadoop data processing (software) framework:

- Abstracts the complexity of distributed programming
- For easily writing applications which process vast amounts of data in-parallel on large clusters

Two popular implementations:

- **MapReduce**: used for individual batch (long running) jobs
- **Spark**: All of the same, plus streaming, interactive, and iterative batch jobs



# MAPREDUCE (“JUST A TASTE”)

MapReduce allows a style of parallel programming designed for:

- Distributing (parallelizing) a task easily across multiple nodes of a cluster
  - Allows programmers to describe processing in terms of simple **map** and **reduce** operations, which are inherently parallel
- Invisible management of hardware and software failures
- Easy management of very large-scale data

# MAPREDUCE: TERMINOLOGY

- A MapReduce job starts with a collection of input elements of a single type  
-- technically, all types are key-value pairs
- A MapReduce job/application is a complete execution of Mappers and Reducers over a dataset
  - Mapper applies the map functions to a single input element
  - Application of the reduce function to one key and its list of values is a Reducer
- Many mappers/reducers grouped in a Map/Reduce task (the unit of parallelism)

# MAPREDUCE: PHASES

## Map

- Each Map task (typically) operates on a single HDFS block -- Map tasks (usually) run on the node where the block is stored
- The output of the Map function is a set of 0, 1, or more key-value pairs

## Shuffle and Sort

- Sorts and consolidates intermediate data from all mappers -- sorts all the key-value pairs by key, forming key-(list of values) pairs.
- Happens as Map tasks complete and before Reduce tasks start

## Reduce

- Operates on shuffled/sorted intermediate data (Map task output) -- the Reduce function is applied to each key-(list of values). Produces final output.

# EXAMPLE: WORD COUNT (1)

The use case scenario: Start with standard WC for one file.

We have a large file of documents (the input elements)

Documents are words separated by whitespace.

Count the number of times each distinct word appears in the file.

... with MapReduce we can extend this concept to huge numbers of files.

# EXAMPLE: WORD COUNT (2)

Why Do We Care About Counting Words?

- Word count is challenging over massive amounts of data
  - Using a single compute node would be too time-consuming
  - Using distributed nodes requires moving data
  - Number of unique words can easily exceed available memory -- would need to store to disk
- Many common tasks are very similar to word count, e.g., log file analysis

# WORD COUNT USING MAPREDUCE (1)

**map(key, value):**

// key: document ID; value: text of document

FOR (each word w IN value)

emit(w, 1);

**reduce(key, value-list):**

// key: a word; value-list: a list of integers

result = 0;

FOR (each integer v on value-list)

result += v;

emit(key, result);

# WORD COUNT USING MAPREDUCE (2)

Input

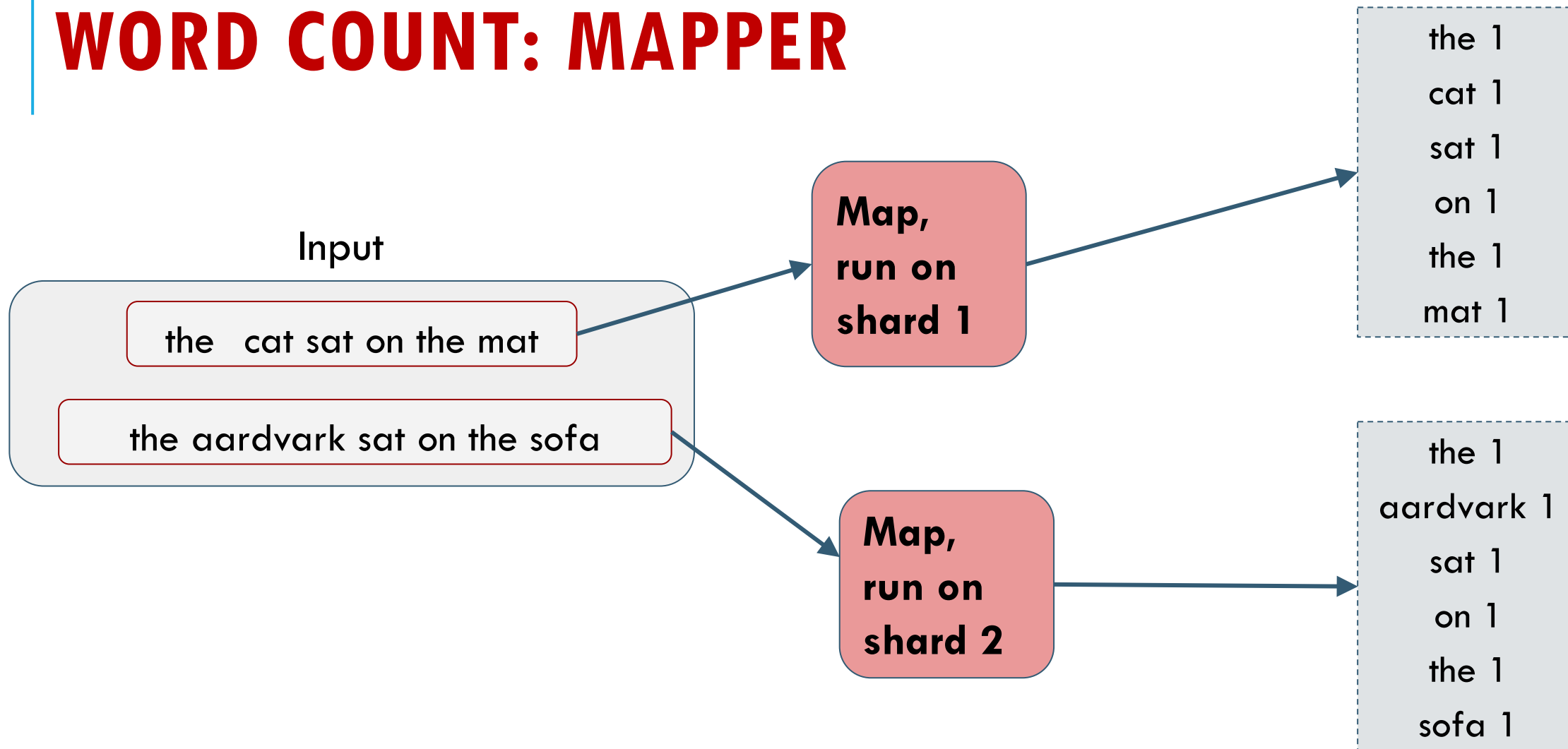
the cat sat on the mat  
the aardvark sat on the sofa

Map & Reduce

Result

aardvark 1  
cat 1  
mat 1  
on 2  
sat 2  
sofa 1  
the 4

# WORD COUNT: MAPPER





# WORD COUNT: SHUFFLE & SORT

Mapper  
Output

the 1
cat 1
sat 1
on 1
the 1
mat 1
the 1
aardvark 1
sat 1
on 1
the 1
sofa 1

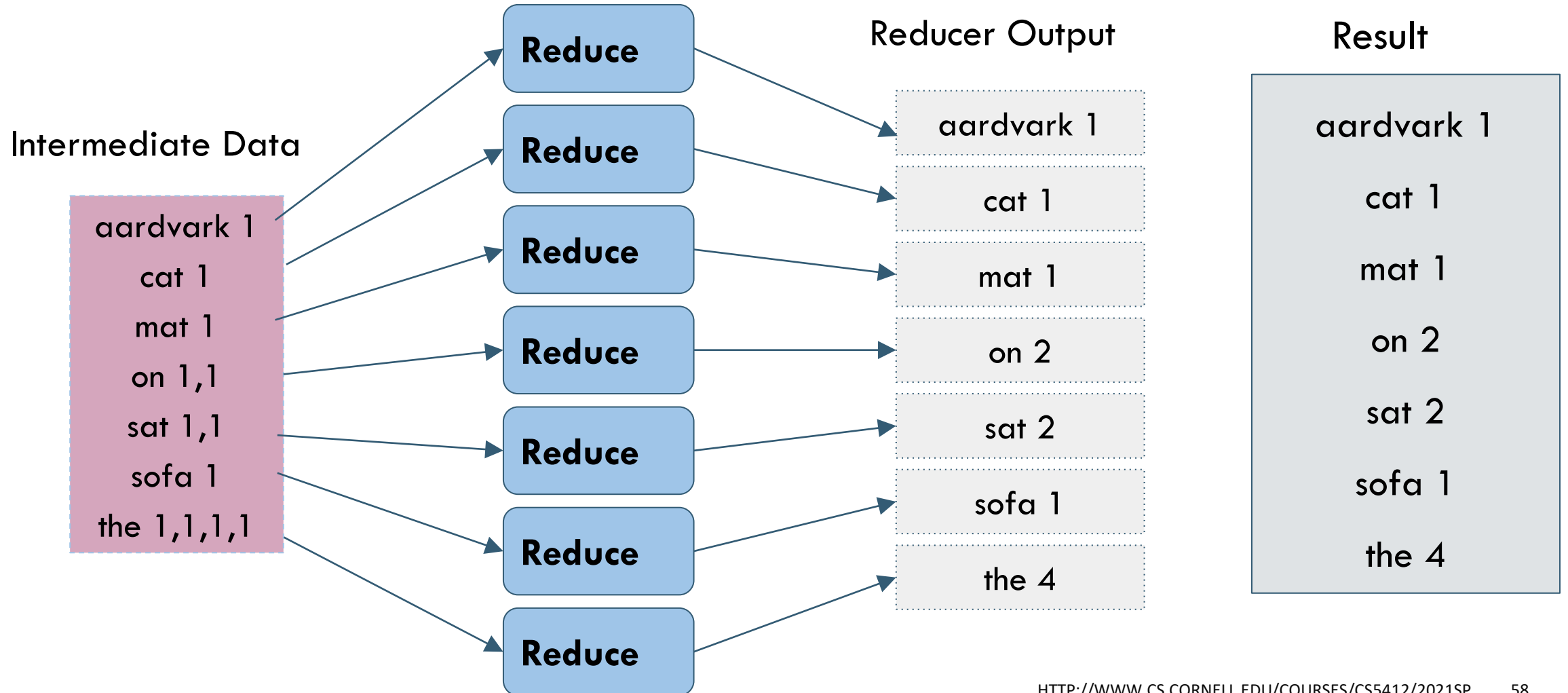
Shuffle & Sort



Intermediate Data

aardvark 1
cat 1
mat 1
on 1,1
sat 1,1
sofa 1
the 1,1,1,1

# WORD COUNT: REDUCER



# THE SLIDE WAS A LIE!

In fact the reducer output might never be combined into a single file, the way the right-hand side of that slide made it seem.

With MapReduce we often start with sharded data, then “transform” it via one or more map-reduce stages, but end up with a sharded result.

The reason is that often the result itself is also a massive object. It might be far too big to fit into any single machine. So we keep it sharded!

# COPING WITH FAILURES

- MapReduce is designed to deal with compute nodes failing to execute a Map task or Reduce task.
- Re-execute failed tasks, not whole jobs/applications.
- Key point: MapReduce tasks produce no visible output until the entire set of tasks is completed. If a task or sub task somehow completes more than once, only the earliest output is retained.
- Thus, we can restart a Map task that failed without fear that a Reduce task has already used some output of the failed Map task.

# MORE BIG DATA STUFF

Apache has many more tools that we haven't discussed today (and won't cover in CS5412 – they belong in a “big data” course, not CS5412).

They include software for automatically extracting structured data from unstructured web pages or documents, ways to treat files as massive databases or spreadsheets but for limited kinds of retrieval and search tasks, etc. Names include HBASE, RocksDB, PIG, HIVE, etc.

Many of them are implemented “over” Hadoop

# SUMMARY

With really huge data sets, or changing data collected from huge numbers of clients, it often is not practical to use a classic database model where each incoming event triggers its own updates.

So we shift towards batch processing, highly parallel: many updates and many “answers” all computed as one task.

Then cache the results to enable fast tier-one/two reactions later.