PUTTING IT ALL TOGETHER

We have seen a great deal of the structure at the edge of the cloud, and how it might “talk” to data sources of various kinds.

Tasks like machine learning and “big data” analytics occur at the back.

How are these connected together?

- Some tier two µ-services store data into files or append-only logs
- Periodically (delay depends on the use) we process in batches.
  … we batch the operations because of the massive scale.
  … idea is to compute “once” but answer many questions!
For example, imagine a smart city. We could collect a **batch** of new location data, then update all the “people locations” in some database in a single (parallel) computation.

- One computation reads a lot of files, and updates many data fields.
- The pattern is typical of today’s cloud.

Due to delays, batched computing not ideal for instant reaction.

... But it is adequate for tasks where a short delay is fine.
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.
Petabytes of data need to be processed or accessed everyday

➢ 300 Billion Google searches per day
➢ 300 Million photos uploaded on Facebook per day

Nature of the data causes it to be massive: 3 V’s

➢ Volume, Velocity, and Variety

The volume of unstructured data exploded in the past decade

➢ By 2020, it will be 53 ZettaBytes (53 trillion gigabytes) -- an increase of 10 times in 15 years (it was below 5 ZB until 2003)
COMMON BIG DATA USE CASES

➢ Extract/Transform/Load (ETL) – Huge Data Warehouses
➢ Text Mining
➢ Graph Creation and Analysis
➢ Prediction Models
➢ Analytics

To exploit batching, Google/Facebook etc., precompute the results for anticipated search queries. They analyze data in batches, then cache results.
BIG DATA PROCESSING?

Nature of the data forces us to use massive parallelism

➢ Recall: Huge Volumes, High Velocity, and Variety

Traditional Single Server Systems are far too weak for processing petabytes of data – insufficient compute and storage

The only option: Distribute data, obtain parallelism with multiple servers
TRADITIONAL DISTRIBUTED SYSTEMS

The Data Bottleneck:

➢ Data was historically first stored in a central location
➢ … then copied to processors at runtime
➢ Fine for limited amounts of data, breaks with massive data sets

Solution: A new style in which we process huge numbers of data files in parallel -- BigData Systems (e.g., Apache Hadoop, Apache Spark)
BIG DATA SYSTEMS

Two key ideas

➢ Distribute data right from the outset, when the data is initially stored
➢ Bring computation to the data rather than sending data to the computation

Scalable and economical data storage, processing and analysis

➢ Distributed and Fault-tolerant
➢ Harness the power of industry standard hardware
➢ Heavily inspired by Open Source technologies (HDFS, HBase, etc.)
A TYPICAL BIG DATA SYSTEM

Popular BigData Systems: Apache Hadoop, Apache Spark
APACHE HADOOP ECOSYSTEM

Map Reduce

Hive

Pig

Other Applications

Spark Stream

Yet Another Resource Negotiator (YARN)

Hadoop NoSQL Database (HBase)

Hadoop Distributed File System (HDFS)

Cluster

Data Ingest Systems e.g., Apache Kafka, Flume, etc
HADOOP ECOSYSTEM

• HDFS, HBase
• Yet Another Resource Negotiator
• MapReduce, Hive
• Kafka
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 IS FOR BATCH PROCESSING

Designed for batch processing rather than interactive
High throughput of data access rather than low latency
HDFS

Must never lose any data (Resilience)
Sits on top of native file systems (e.g., xfs, ext3, ext4) -- HDFS is written in Java, i.e., JVM → OS (file system) → Storage (disks)

Write-once read-many (or append-only) paradigm → Batch Processing → High Throughput

• Data is distributed when stored & move computation to the data

• Minimizes network congestion and increases throughput
HDFS: KEY FEATURES

- Scalable: HDFS is designed for massive scalability, so you can store unlimited amounts of data in a single platform.
- Flexible: Store data of any type -- structured, semi-structured, unstructured -- without any upfront modeling.
- Reliable: Multiple copies of your data are always available for access and protection from data loss (build-in fault tolerance).
HDFS: ARCHITECTURE

Master/slave architecture: NameNode cluster

Availability of Name Node is critical

The NameNode executes file system operations such as opening, closing etc.

The DataNodes (slaves) are responsible for serving read and write requests from the file system’s clients.

Image source: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
Data File is split into contiguous chunks, typically 64MB size, distributed at load time.

Each chunk is replicated on multiple “data” nodes (usually 3x).

“Name” node for a file stores metadata, location of all chunks, etc.

Optimized for large, streaming reads of files (rather than random reads).

Files are “write once” -- No random writes to files allowed -- because of HDFS’s batch roots, it was only designed to handle append-only formats.
How to choose number of replicas (replication factor)?
The client sends a request to the NameNode to read a file.

The NameNode determines which blocks are involved and chooses the most efficient access path.

The client then accesses the blocks using the addresses provided by the NameNode.
HDFS: WRITING DATA (1)

Get a Lease → Write Data → Close the Lease

Getting Lease:

- The client sends a request to the NameNode to create a new file.
- The NameNode determines how many blocks are needed, and the client is granted a lease for creating these new file blocks in the cluster.
HDFS: WRITING DATA (2)

Get a Lease → Write Data → Close the Lease

Write Data:

• The client then writes the first copies of the file blocks to the slave nodes using the lease assigned by the NameNode.

• As each block is written to HDFS, a special background task duplicates the updates to the other slave nodes identified by the NameNode.
HDFS: WRITING DATA (3)

Get a Lease → Write Data → Close the Lease

Close Lease:

- The DataNode daemons acknowledge the file block replicas have been created,
- The client application closes the file and notifies the NameNode
- NameNode closes the open lease. Updates become visible at this point.
HDFS: SOME LIMITATIONS

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

The Persistence of File System Metadata?
HADOOP DATABASE (HBASE)

A NoSQL database built on HDFS
A table can have thousands of columns
Supports very large amounts of data and high throughput
HBase $\rightarrow$ Strong Consistency
Random access, low latency
HBase

HBase is based on Google’s Bigtable
A NoSQL distributed database/map built on top of HDFS
Designed for Distribution, Scale, and Speed
Relational Database (RDBMS) vs NoSQL Database:
RDBMS → vertical scaling (expensive) → not appropriate for BigData
NoSQL → horizontal scaling / sharding (cheap) → appropriate for BigData
RDBMS VS NOSQL (1)

• BASE not ACID:
  ➢ RDBMS (ACID): Atomicity, Consistency, Isolation, Durability
  ➢ NoSQL (BASE): Basically Available Soft state Eventually consistency

• The idea is that by giving up ACID constraints, one can achieve much higher availability, performance, and scalability
  ➢ e.g. most of the systems 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
• NoSQL $\rightarrow$ eventual consistency (e.g., CouchDB) or strong consistency (HBase)
## HBASE: DATA MODEL (1)

### Data model

<table>
<thead>
<tr>
<th>Row key</th>
<th>info:name</th>
<th>info:age</th>
<th>comp:base</th>
<th>comp:stocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>121</td>
<td>‘tom’</td>
<td>‘28’</td>
<td>‘125k’</td>
<td></td>
</tr>
<tr>
<td>145</td>
<td>‘bob’</td>
<td>‘32’</td>
<td>‘110k’</td>
<td>‘50’ (ts=2012)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>‘100’ (ts=2014)</td>
</tr>
</tbody>
</table>

- **Row keys**
- **Columns**
- **Cells**
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

<table>
<thead>
<tr>
<th>Unique id</th>
<th>Name</th>
<th>price</th>
<th>weight</th>
<th>store1</th>
<th>store2</th>
<th>store3</th>
</tr>
</thead>
<tbody>
<tr>
<td>“1000000”</td>
<td>snickers</td>
<td>$9.99</td>
<td>4 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“3000000”</td>
<td>almonds</td>
<td>$9.99</td>
<td>8 Oz</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>“8000000”</td>
<td>coke</td>
<td>$9.99</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“4000000”</td>
<td>foo</td>
<td>$34.63</td>
<td>16 Oz</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“5000000”</td>
<td>bar</td>
<td>$22.54</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“9000000”</td>
<td>new1</td>
<td>$2.5</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“7000000”</td>
<td>new2</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“2000000”</td>
<td>new3</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
# HBASE: HORIZONTAL SPLITS (REGIONS)

### Region 1: ["", "5000000"]

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Name</th>
<th>brand</th>
<th>price</th>
<th>weight</th>
<th>store1</th>
<th>store2</th>
<th>store3</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;1000000&quot;</td>
<td>snickers</td>
<td>xxx</td>
<td>$9.99</td>
<td>4 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;2000000&quot;</td>
<td>new3</td>
<td>xxx</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;3000000&quot;</td>
<td>almonds</td>
<td>xxx</td>
<td>$9.99</td>
<td>8 Oz</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;4000000&quot;</td>
<td>foo</td>
<td>xxx</td>
<td>$34.63</td>
<td>16 Oz</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Region 2: ["5000000", "]"

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Name</th>
<th>brand</th>
<th>price</th>
<th>weight</th>
<th>store1</th>
<th>store2</th>
<th>store3</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;5000000&quot;</td>
<td>bar</td>
<td>xxx</td>
<td>$22.54</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;7000000&quot;</td>
<td>new2</td>
<td>xxx</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;8000000&quot;</td>
<td>coke</td>
<td>xxx</td>
<td>$9.99</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;9000000&quot;</td>
<td>new1</td>
<td>xxx</td>
<td>$2.5</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## HBASE ARCHITECTURE (REGION SERVER)

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Name</th>
<th>price</th>
<th>weight</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>“1000000”</td>
<td>snickers</td>
<td>$9.99</td>
<td>4 Oz</td>
<td>....</td>
</tr>
<tr>
<td>“2000000”</td>
<td>new3</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>....</td>
</tr>
<tr>
<td>“3000000”</td>
<td>almonds</td>
<td>$9.99</td>
<td>8 Oz</td>
<td>....</td>
</tr>
<tr>
<td>“4000000”</td>
<td>foo</td>
<td>$34.63</td>
<td>16 Oz</td>
<td>....</td>
</tr>
</tbody>
</table>

Server 12

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Name</th>
<th>price</th>
<th>weight</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>“5000000”</td>
<td>bar</td>
<td>$22.54</td>
<td>16 Oz</td>
<td>....</td>
</tr>
<tr>
<td>“7000000”</td>
<td>new2</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>....</td>
</tr>
<tr>
<td>“8000000”</td>
<td>coke</td>
<td>$9.99</td>
<td>16 Oz</td>
<td>....</td>
</tr>
<tr>
<td>“9000000”</td>
<td>new1</td>
<td>$2.5</td>
<td>16 Oz</td>
<td>....</td>
</tr>
</tbody>
</table>

Server 7
# HBASE ARCHITECTURE

<table>
<thead>
<tr>
<th>Unique id</th>
<th>Name</th>
<th>price</th>
<th>weight</th>
<th>store1</th>
<th>store2</th>
<th>store3</th>
</tr>
</thead>
<tbody>
<tr>
<td>“1000000”</td>
<td>snickers</td>
<td>$9.99</td>
<td>4 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“3000000”</td>
<td>almonds</td>
<td>$9.99</td>
<td>8 Oz</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>“8000000”</td>
<td>coke</td>
<td>$9.99</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“4000000”</td>
<td>foo</td>
<td>$34.63</td>
<td>16 Oz</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“5000000”</td>
<td>bar</td>
<td>$22.54</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“9000000”</td>
<td>new1</td>
<td>$2.5</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“7000000”</td>
<td>new2</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“2000000”</td>
<td>new3</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Row Key</td>
<td>info: name</td>
<td>info: price</td>
<td>info: weight</td>
<td>availability: store1</td>
<td>availability: store2</td>
<td>availability: store3</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
<td>--------------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>“1000000”</td>
<td>snickers</td>
<td>$9.99</td>
<td>4 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“2000000”</td>
<td>new3</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“3000000”</td>
<td>almonds</td>
<td>$9.99</td>
<td>8 Oz</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>“4000000”</td>
<td>foo</td>
<td>$34.63</td>
<td>16 Oz</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“5000000”</td>
<td>bar</td>
<td>$22.54</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“7000000”</td>
<td>new2</td>
<td>$6.4</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“8000000”</td>
<td>coke</td>
<td>$9.99</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“9000000”</td>
<td>new1</td>
<td>$2.5</td>
<td>16 Oz</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
HBASE ARCHITECTURE: COLUMN FAMILY

<table>
<thead>
<tr>
<th>Column Family</th>
<th>info: name</th>
<th>info: price</th>
<th>info: weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>“1000000”</td>
<td>snickers</td>
<td>$9.99</td>
<td>4 Oz</td>
</tr>
<tr>
<td>“2000000”</td>
<td>new3</td>
<td>$6.4</td>
<td>16 Oz</td>
</tr>
<tr>
<td>“3000000”</td>
<td>almonds</td>
<td>$9.99</td>
<td>8 Oz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column Family</th>
<th>available: store1</th>
<th>available: store2</th>
<th>available: store3</th>
</tr>
</thead>
<tbody>
<tr>
<td>“1000000”</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“2000000”</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>“3000000”</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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 is a key-value store designed for distribution, scale, and speed

- Data that is accessed together is stored together → faster for scaling
- Grouping the data by key (here rowkey) is central to running on a cluster and sharding -- the key acts as the atomic unit for updates
- Each record/row is indexed by a key (the rowkey) that you can use for lookup. The rowkey is like a primary key from a relational database.
- Records in HBase are stored in sorted order according to the rowkey. This is a critical semantic used in HBase schema design.
HBASE CONCEPTS (2)

Horizontal splits/sharding:
- Tables are divided into sequences of rows, by “key range”, called regions.
- These regions are then assigned to the data nodes (HDFS) in the cluster called “RegionServers” → Preserving data locality.
- Scales read and write capacity by spreading “regions” across the cluster.
- Here HBase maps (rowkey, column family, column, timestamp) to a “value”.
HBASE ARCHITECTURE (1)

HBase is composed of three types of servers in a master slave type of architecture: Region Server, Hbase Master, ZooKeeper.

Region Server:
- Clients communicate with RegionServers (slaves) 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 Master: 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.
HOW DO THESE COMPONENTS WORK TOGETHER?

Region servers and the active HBase Master 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.
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.
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.
ZOOKEEPER: THE COORDINATOR

Maintains region server state in the cluster
Provides server failure notification
Uses consensus to guarantee common shared state
HBASE: SOME LIMITATIONS

Not ideal for large objects (>50MB 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 store data chronologically (time as primary index), e.g., machine logs organized by time-stamps cause write hot-spots.
HBASE VS HDFS

Hbase is a NoSQL distributed store layer (on top of HDFS). It is for faster random, realtime read/write access to the big data stored in HDFS.

**HBase**
- Stores data as key-value stores in columnar fashion. Records in HBase are stored according to the rowkey and that 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
Yet Another Resource Negotiator (YARN)

➢ YARN is a core component of Hadoop, manages all the resources of a Hadoop cluster.

➢ 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)

- Map Reduce
- Yet Another Resource Negotiator (YARN)
- Hive
- Pig
- Other Applications
- Spark Stream
- Hadoop Distributed File System (HDFS)
- Hadoop NoSQL Database (HBase)
- Data Ingest Systems e.g., Apache Kafka, Flume, etc.

Resource manager
YARN CONCEPTS (1)

Container:

- YARN uses an abstraction of resources called a container for managing resources -- an unit of computation of a slave node, i.e., a certain amount of CPU, Memory, Disk, etc., 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.
Three Main components of YARN:

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

➢ Application Master:
  ○ 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 slave node. Responsible for monitoring and reporting on local container status (all containers on slave node).
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.
How do the components of YARN work together?

Image source: http://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/YARN.html
HADOOP ECOSYSTEM (PROCESSING LAYER)

Map Reduce
Hive
Pig
Other Applications
Yet Another Resource Negotiator (YARN)
Spark Stream
Hadoop Distributed File System (HDFS)
Hadoop NoSQL Database (HBase)

Data Ingest Systems e.g., Apache Kafka, Flume, etc.
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 frameworks:

➢ MapReduce: used for individual batch (long running) jobs
➢ Spark: for streaming, interactive, and iterative batch jobs

Note: Spark is more than a framework. We will learn more about this in future lectures
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 functions
- Invisible management of hardware and software failures
- Easy management of very large-scale data
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.
The Problem:

➢ 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.
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

Input:
- the cat sat on the mat
- the aardvark sat on the sofa

Map:
- the 1
- cat 1
- sat 1
- on 1
- the 1
- mat 1

Map:
- the 1
- aardvark 1
- sat 1
- on 1
- the 1
- sofa 1
WORD COUNT: SHUFFLE & SORT

Mapper Output
- the 1
- cat 1
- sat 1
- on 1
- mat 1
- aardvark 1
- sat 1
- on 1
- the 1
- sofa 1

Intermediate Data
- aardvark 1
- cat 1
- mat 1
- on 1,1
- sat 1,1
- sofa 1
- the 1,1,1,1

Shuffle & Sort
WORD COUNT: REDUCER

Intermediate Data:
- aardvark 1
- cat 1
- mat 1
- on 1,1
- sat 1,1
- sofa 1
- the 1,1,1,1

Reducer Output:
- aardvark 1
- cat 1
- mat 1
- on 2
- sat 2
- sofa 1
- the 4

Result:
- aardvark 1
- cat 1
- mat 1
- on 2
- sat 2
- sofa 1
- the 4
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.
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.