An Introduction to ~~Big Data~~
Data-Intensive Applications
Hadoop/Spark and others

Ismaël Mejía
@iemejia

Who am I ?

- Software Engineer (Practitioner)
- 10 years fighting with Software, particularly interested in Distributed Systems
- Apache Avro/Beam committer
- Now: Working on Open Source Big Data projects @Talend

Agenda

1) Big Data Concepts
2) Map Reduce / Hadoop
3) Big-Data Architectures

TPs about (2)

Part 1: Big Data Concepts

The need for massive data processing

- Internet scale (crawling and indexing)
- Cheap computing (The Cloud)
- Massive amount of information sources:
  - Social Networks
  - Sensors
- Text-retrieval and non-structured data

Why traditional DBs are not enough?

- Not horizontally scalable
- Schema on-write
- Not easy to work with semi-structured data
- Normalization makes partitioning harder
- SQL is fantastic but restrictive

What is Big Data? (Some definitions)

- Big data usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process data within a tolerable elapsed time. Big data "size" is a constantly moving target, as of 2012 ranging from a few dozen terabytes to many petabytes of data. Big data is a set of techniques and technologies that require new forms of integration to uncover large hidden values from large datasets that are diverse, complex, and of a massive scale.. [Wikipedia]
- Big Data is the result of collecting information at its most granular level.
- Big Data is an opportunity to gain a more complex understanding of the relationships between different factors and to uncover previously undetected patterns in data.
- Big data is when your business wants to use data to solve a problem, answer a question, produce a product, etc., but the standard, simple methods (maybe it’s SQL, maybe it’s k-means, maybe it’s a single server with a cron job) break down on the size of the data set, causing time, effort, creativity, and money to be spent crafting a solution to the problem that leverages the data without simply sampling or tossing out records.
- Big data refers to using complex datasets to drive focus, direction, and decision making within a company or organization.
- [Big data means] harnessing more sources of diverse data where “data variety” and “data velocity” are the key opportunities.
- Big data is data at a scale and scope that changes in some fundamental way (not just at the margins) the range of solutions that can be considered when people and organizations face a complex problem.

A revolution or just a buzzword?

- Let’s be honest the term has been abused, but a buzzword implies enthusiasm

- Really interesting use cases:
  - Web crawling
  - Social Network / Large Graph Analysis
  - Real-Time indexing / search
  - Internet of Things
  - Machine Learning, Recommendation Systems

- But... Inappropriate behavior from governments and companies are probably the worst side effect. Nothing is deleted anymore.

A simple Big Data definition

Big Data is data that can’t be fitted on a single computer.

so…

*Big Data is a new “disguise” for **data-intensive distributed systems***

Big Data != NoSQL != NewSQL

A ‘real’ data system is a Distributed System

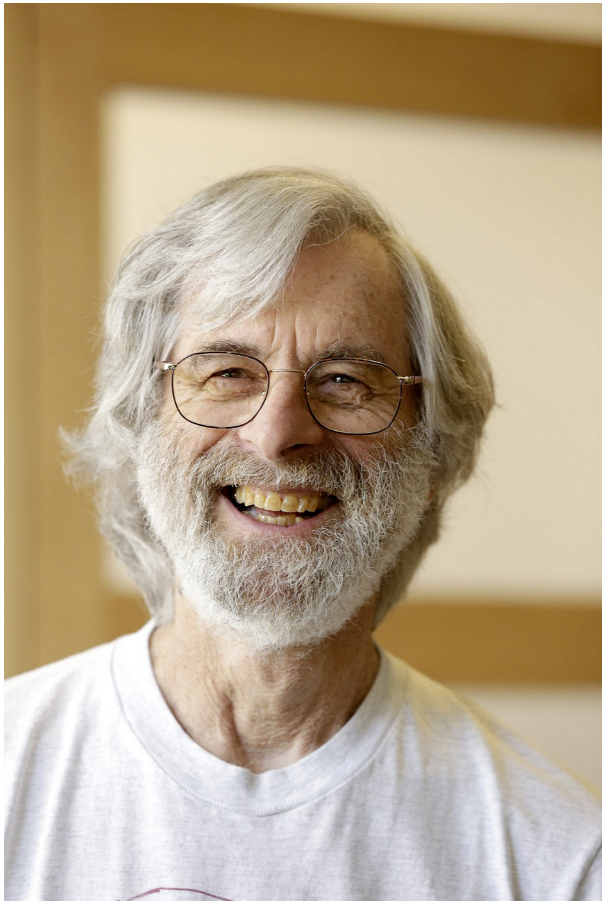
“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.”

Distributed systems are **HARD**!

but interesting :-)

A distributed systems problem

- Fault Tolerance
- Redundancy
- Scalability
- Latency
- Data Locality
- CAP Theorem*



Leslie Lamport
Distributed Systems **Jedi**

What if you had to resolve this problem?

Requirements:

- Support failures in multiple machines
- Hide system-level details from the app developer
- Seamless scalability
- Use commodity hardware
- Optimize for hardware constraints:
  - Machines break
  - Hard drives break
  - Seek time is costly
  - Latency matters

Some ideas:

- Partition for parallelism
- Replication for Integrity
- If data is huge sending function to data is better!

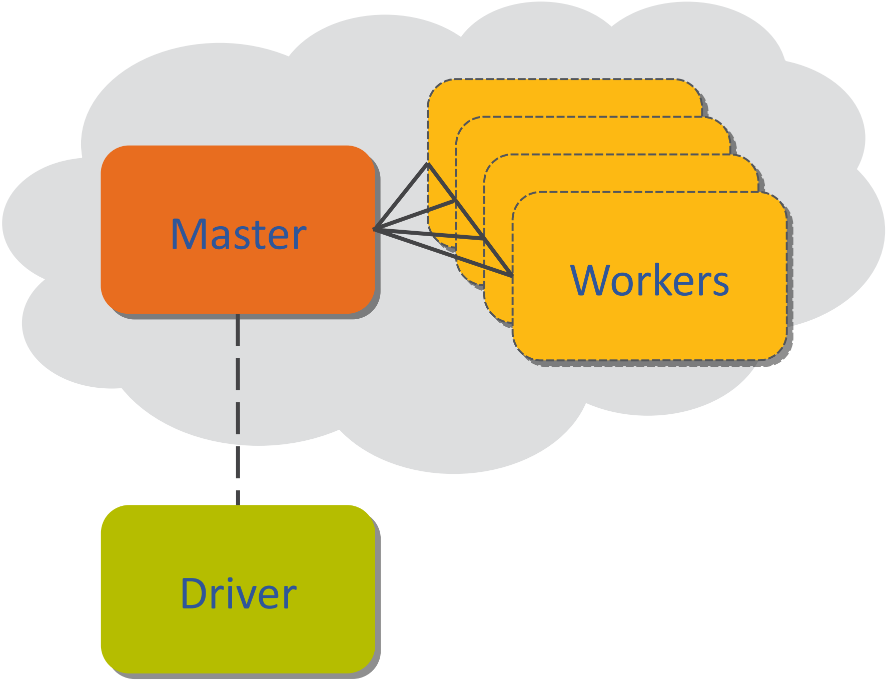
Latency Numbers Every Programmer Should Know

```
Latency Comparison Numbers
--------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1
cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2
cache, 200x L1 cache
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec
SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec
SSD, 4X memory
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x
datacenter roundtrip
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x
memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms

Notes
-----
1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns
```

Technical Definition (cop)

Big Data is a framework for distributed storage and parallel computation.

- Master/Workers are nodes in your cluster.
- Master provides the single point of access to the services, distributes resources and assign tasks to the workers.
- Need more tasks consuming more resources? Just add more workers (Velocity).
- Driver is the application / client requesting work to be done and receives results
- Driver can be launched from within the cluster (or outside)



Part 2: Map Reduce / Hadoop

A little bit of history of Map Reduce

- Google introduced Google File System (2003) and a way to process data M/R (2004)
- Process huge dataset in a scalable way. Google problem indexing the web.
- Based on the parallelism of Map and Reduce operators

Jeff Dean
Map Reduce co-author

How would you calc the sum of squares of the following list? *
Int[] xs = {1, 2, 3, 4, 5} = 1 + 2^2 + 3^2 + 4^2 + 5^2 = 55

Map and Reduce types

```java
long total = 0;
int[] xs = {1, 2, 3, 4, 5};
for (int i = 0; i < xs.length; i++) {
    total += xs[i] * xs[i];
}
System.out.println(total);

55
```

```haskell
map :: (a -> b) -> [a] -> [b]
ys = map (^2) [1,2,3,4,5] = [1,4,9,16,25]
foldl (+) 0 ys

55


-- follow the types luke
-- maps
map :: (a -> b) -> [a] -> [b]
flatMap ~= (a -> [b]) -> [a] -> [[b]] = [b]
filter :: (a -> Bool) -> [a] -> [a]

-- reducers also called combiners or folds
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Associative combining operators are a VERY BIG DEAL!

Pure functions and you get parallelism for free !

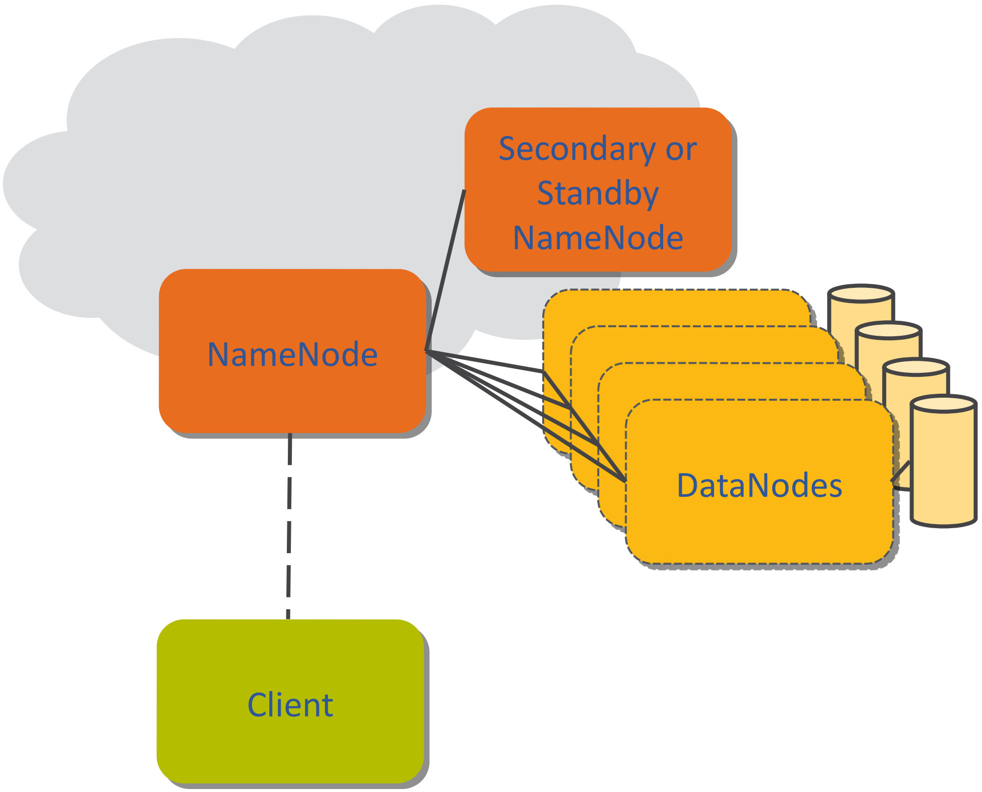
Apache Hadoop

- An open source version of Map Reduce
- Created as part of an open source search project (Nutch).
- Partition data in nodes of a cluster
- Replicas for integrity
- Written in Java
- Hides operational complexity (e.g. programming processing tasks in case of failure)
- Hadoop = HDFS + MR + YARN

HDFS – Hadoop Distributed File System

Distributed Storage

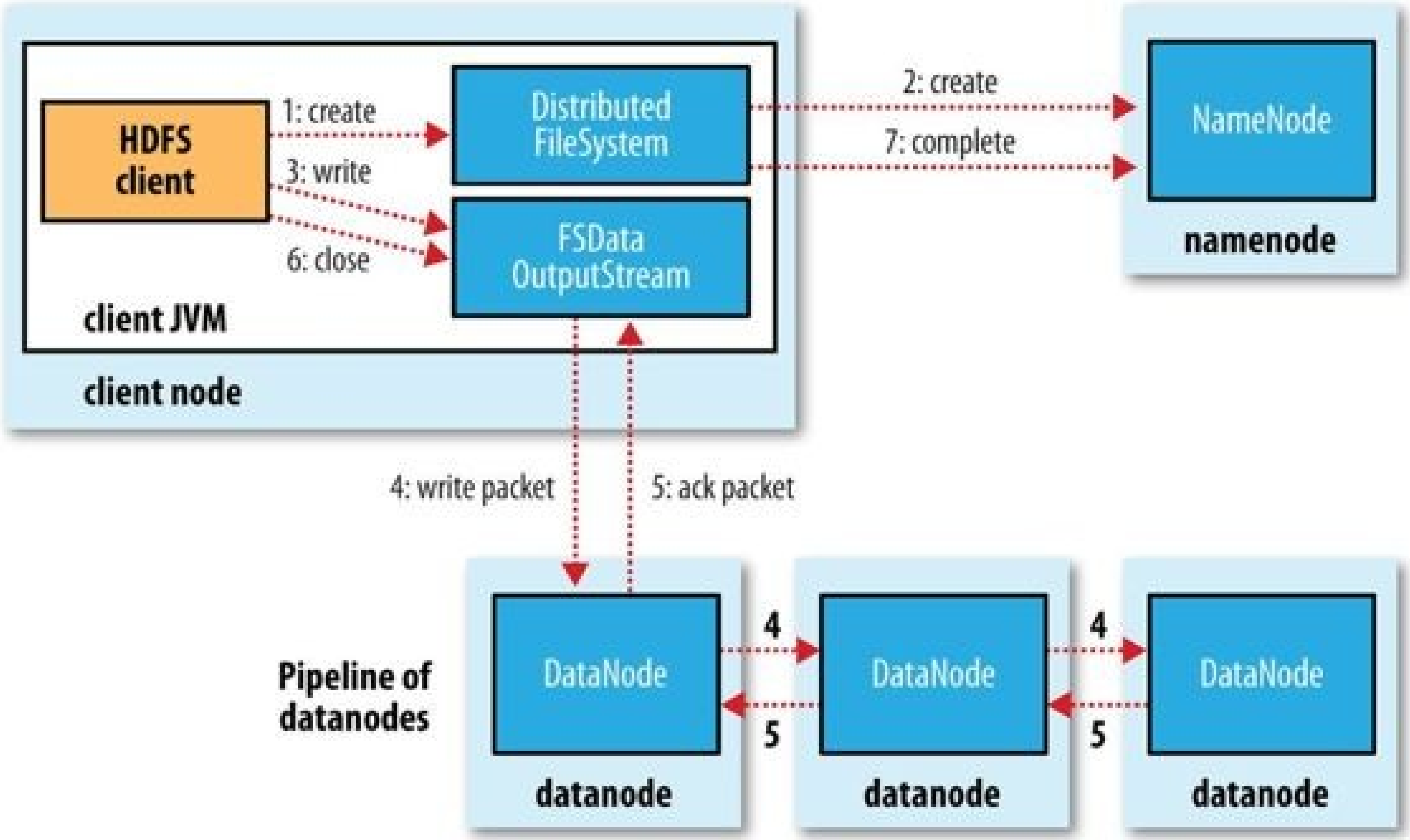
- **Namenode** (master) manages file metadata.
  - Commit logs and compaction for efficiency.
  - Secondary namenode for redundancy, OR High Availability in active/standby mode.
- **Datanodes** (workers) respond to file requests.
  - Distributed
  - Files are separated into large blocks (network transfer >> disk seek time)
  - A file can be larger than any single disk.
- Replicated (N=3)
  - Redundant (up to N–1 machines can fail with no data loss)
  - **Data-locality** (machines can be assigned to process the data on their local disk)



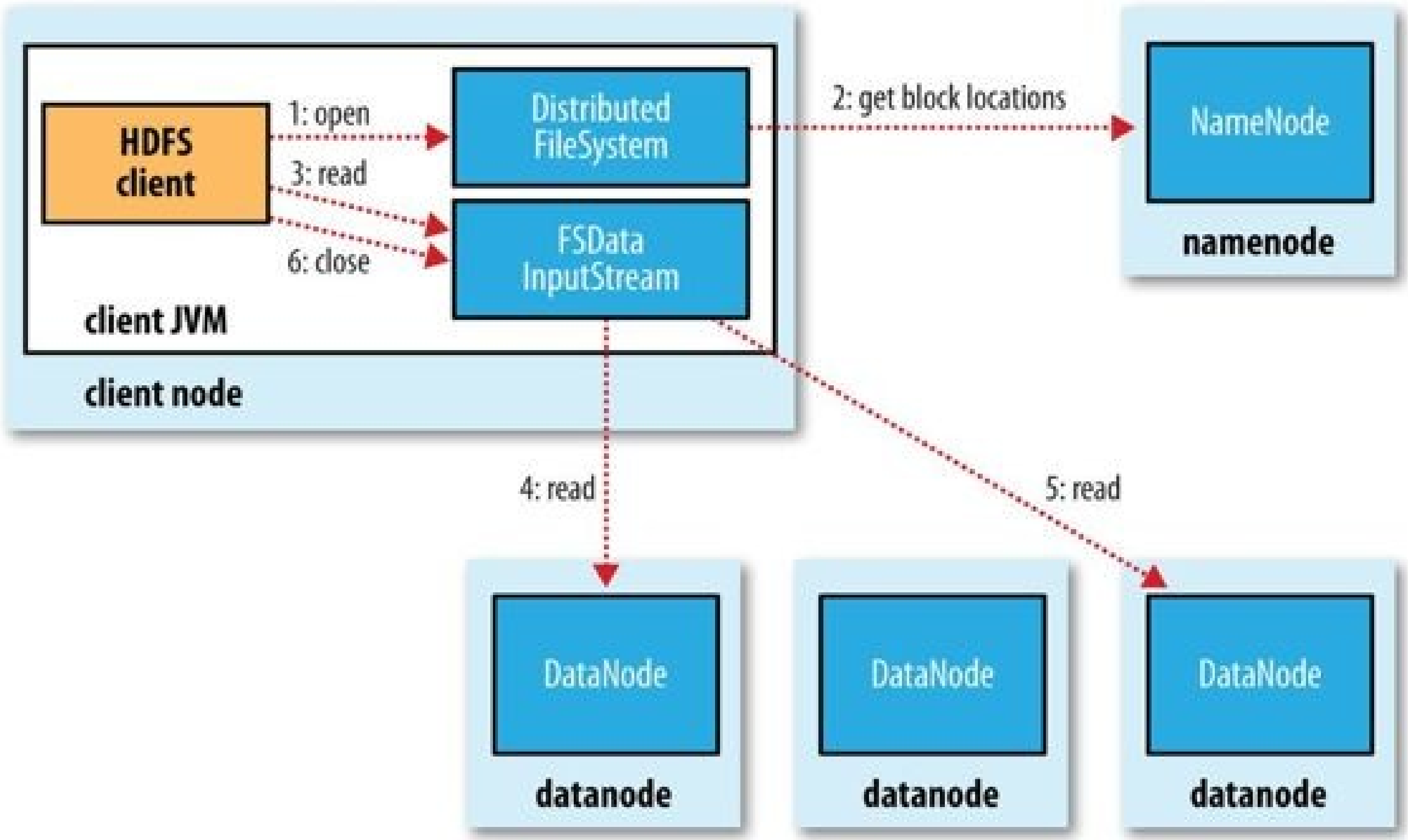
HDFS storage

- Implementation of Google File System (GFS)
- Like a virtual hard disk but distributed, not POSIX
- All files are physically split into fixed-size blocks (today 128 or 256MB)
- Once written not modifiable (Immutability*)
- Split strategy is determined based on record identification (default is line separator)
  - It is up to the RecordReader to handle records spanning several blocks
- Each block can be processed in parallel
- YARN interacts heavily with Namenode to determine best data locality to assign tasks where the block lives and minimize network transfers

HDFS Write


HDFS client
1: create
3: write
6: close
Distributed FileSystem
FSData OutputStream
client JVM
client node
2: create
7: complete
NameNode
namenode
4: write packet
5: ack packet
Pipeline of datanodes
DataNode
DataNode
DataNode
4
4
5
5
datanode
datanode
datanode

HDFS Read

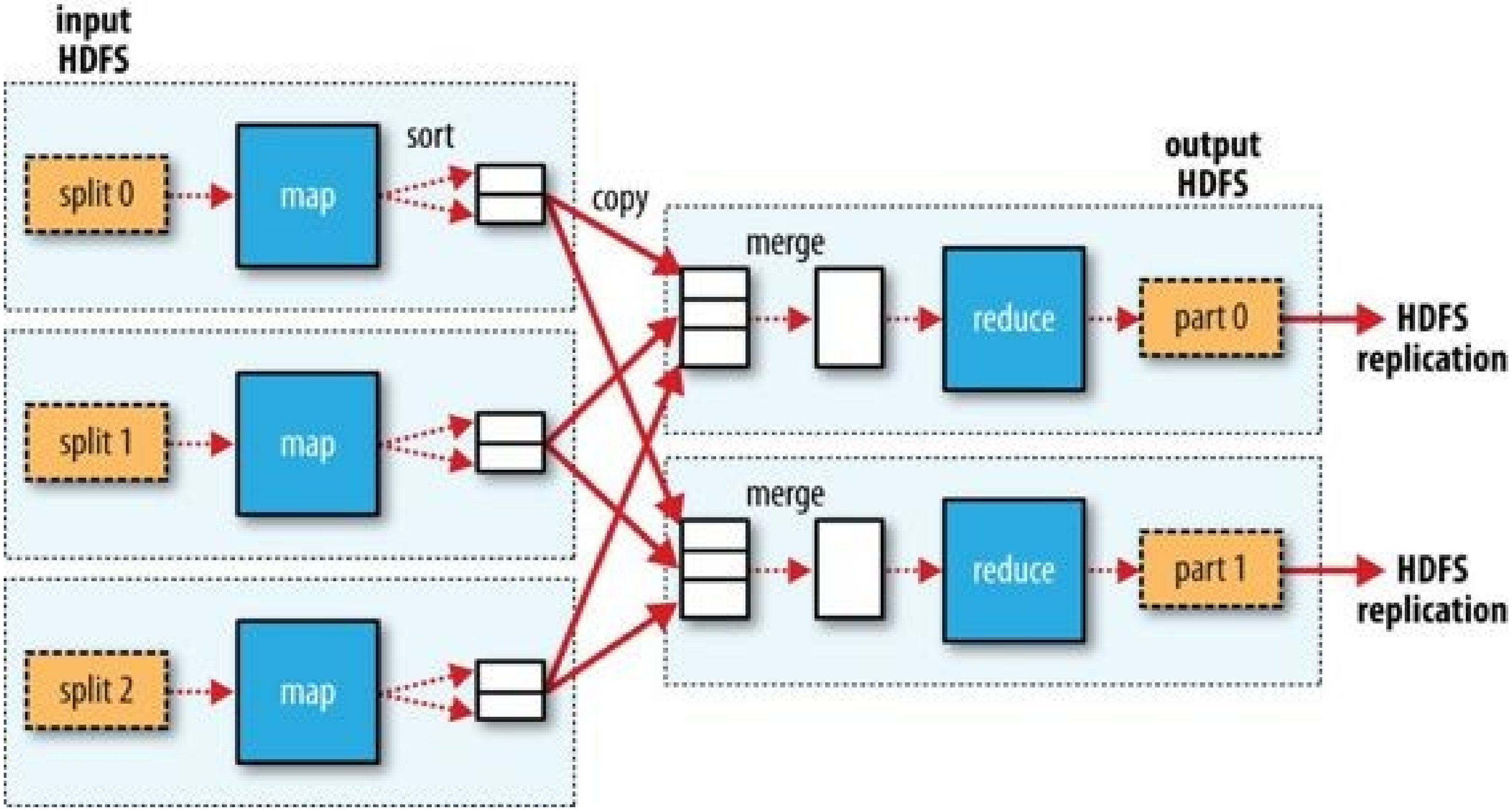


HDFS client
1: open
3: read
6: close
Distributed FileSystem
FSData InputStream
client JVM
client node
2: get block locations
NameNode
namenode
4: read
5: read
DataNode
datanode
DataNode
datanode
DataNode
datanode

HDFS use examples

```
hadoop fs -ls PATH

hadoop fs -mkdir PATH

hadoop fs -put local_folder/file
hdfs://folder/file
```

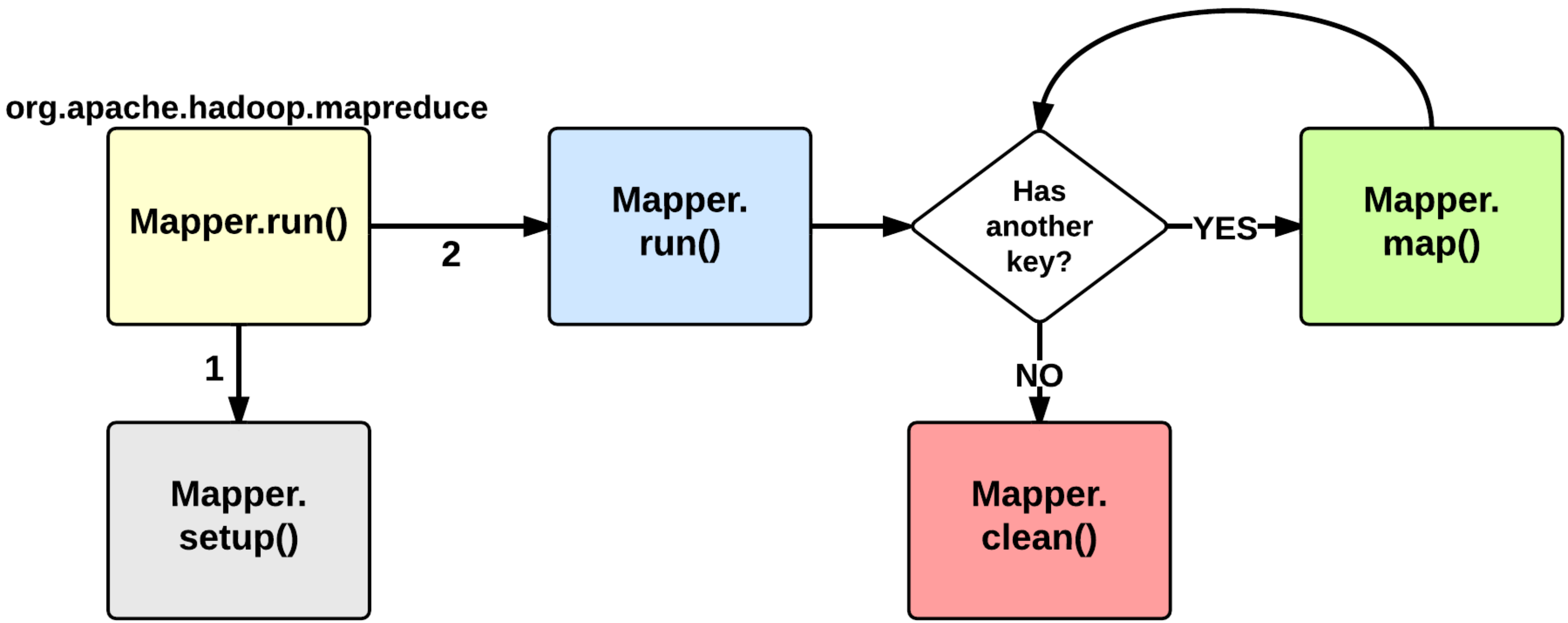
Map/Reduce on Hadoop



- Split a *huge* input into smaller chunks (input splits)
- Start many **map** tasks, assigning each an input split to scan into records
- Each record is processed independently into a `key / value` pair

- All of the pairs are sorted by key. All values for a key are grouped together (shuffle and sort)
- Start many **reduce** tasks, and partition the keys equitably
- Process each list of values into the output

Map task execution



Reduce follows a similar path

Map/Reduce Code

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

    Job job = Job.getInstance(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(MyMapper.class);
    job.setReducerClass(MyReducer.class);

    job.setJarByClass(WordCount.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
}
```

Map/Reduce Code

```
public static class MyMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {
     private final static IntWritable one = new IntWritable(1);

     public void map(LongWritable key, Text value, Context
context)
             throws IOException, InterruptedException {
             // code

             context.write(word, one);
     }
}
```

Map/Reduce Code

```
public static class MyMapper
   extends Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value, Context
context)
            throws IOException,InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
```

Map/Reduce Code

```
public static class MyReducer
   extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values,
Context context)
            throws IOException, InterruptedException {
        // code


        context.write(key, new IntWritable(sum));
    }
}
```

Map/Reduce Code

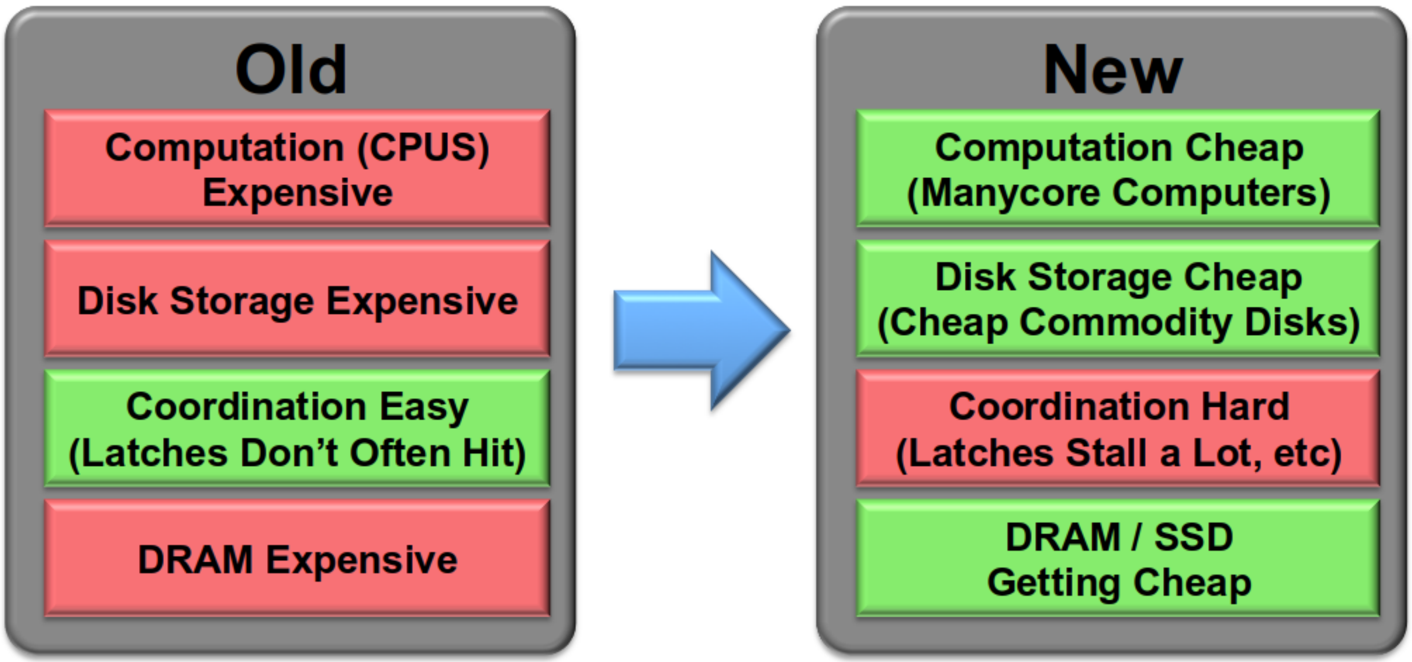
```
public static class MyReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values,
Context context)
            throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

Optimizations

- Avoid seeks, process data sequentially
- Move processing to the data
- Rack awareness
- Combiners: Local reducers (associative + commutative)*
- Try to avoid shuffling if you can.
- Bloom Filters and other stuff

Immutability changes everything

Some Industry Trends to Consider



We Can Afford to Keep Immutable Copies of Lots of Data

We Need Immutability to Coordinate with Fewer Challenges

Immutability is a huge step to make Human-Tolerant Systems

Algebraic Properties are important

- Associative: grouping doesn't matter!
- Commutative: order doesn't matter!
- Idempotency: duplicates don't matter!
- Identity: this value doesn't matter!
- Zero: other values don't matter!

Guy Steele
PL Guru (LISP/Java/Fortress)

And don't forget Immutability changes everything!

Data formats

Type	Orientation	Splittable	Compressable	Hierarchical	Notes
Plain text	Row	Yes	Yes (expensive)	No	Deal with encoding
XML / JSON	Row	No (expensive)	Yes	Yes	Very difficult to split without cheating.
SequenceFile	Row	Yes	Yes (block, record)	Yes (expensive)	Uses handwritten Hadoop-style serialization. Fast/customizable but harder to develop/maintain (especially for Hierarchical data).
MapFile	Row	Yes	Yes	Yes (expensive)	Ordered, indexed SequenceFile for fast lookups.
Avro	Row	Yes	Yes	Yes	Widely used. Efficient binary layout of structured data. JSON schema embedded with data.
ORC	Column	Yes	Yes	Yes	Hive-initiated. Only accessible via Hive or HCatalog
Parquet	Column	Yes	Yes	Yes	Hive-initiated. Connectors available in all frameworks

Data format recommendations

When data arrives in Hadoop in a “Document”-like format (xml, json, nested flat files), **it is more efficient to store it in optimized format like Avro or Parquet**

Columnar vs Row storage

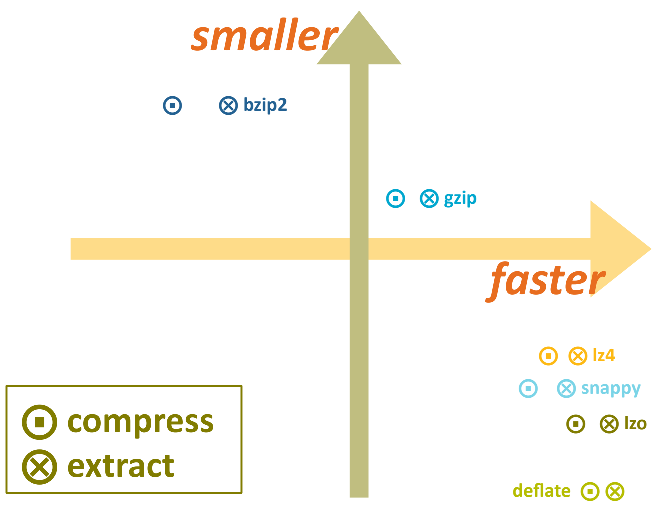
- Columnar data stores are good when not all columns for a record are needed (like aggregation queries)
- Largely reduces seek/scan time
- Usually split on chunks of rows, then organized by columns
- Optional indexes and aggregate statistics of rows in a chunk
- Columnar storage have a better compression ratio and is more suited on very high volume requirements
- Prefer Parquet over ORC, thanks to native compatibility with almost all the ecosystem whereas ORC is only for Hive / Hcatalog

http://fr.slideshare.net/julienledem/parquet-hadoop-summit-2013

Data compression

- Prefer Snappy or LZO (if LZO is correctly installed by Sysadmins)
- Forbid unsplittable formats on HDFS (gzip), it will limit to 1 mapper / file, highly impacting performance if files > 64MB

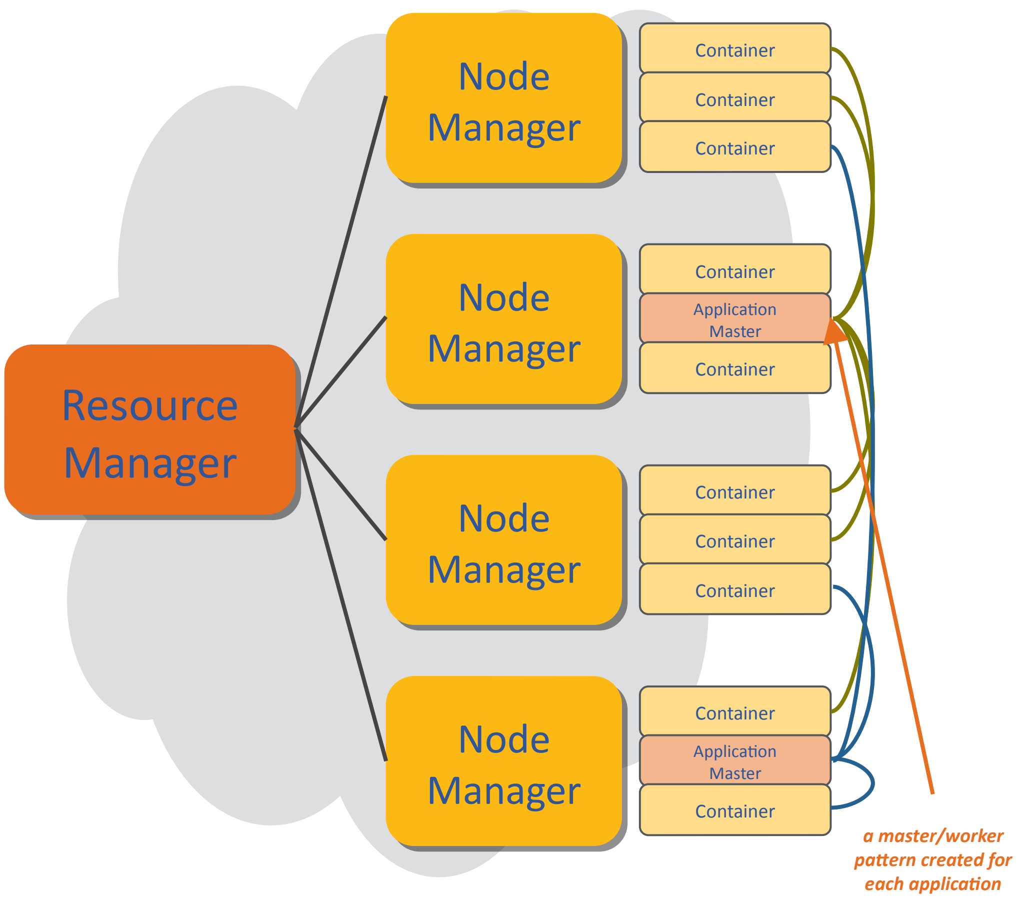
Compression Format	Tool	Algo	Extension	Splittable?
DEFLATE	compress	DEFLATE	.deflate/.Z	no
gzip	gzip	DEFLATE	.gz	no
bzip2	bzip2	bzip2	.bz2	yes
LZO	lzop	LZO	.lzo	yes(if encoded by records)
LZ4	n/a	LZ4	.lz4	no
Snappy	n/a	Snappy	.snappy	yes(if encoded by records)



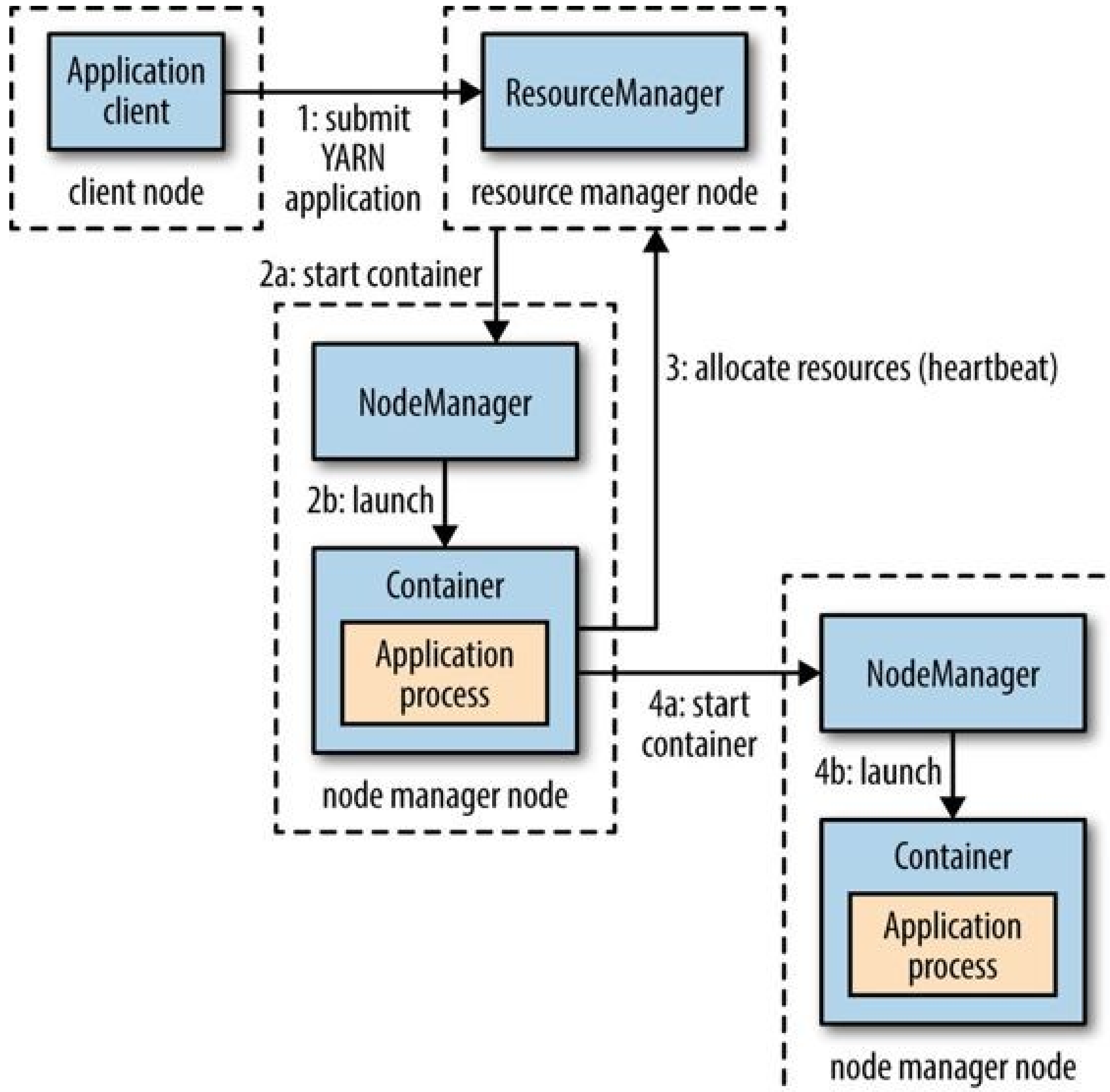
Yet Another Resource Negotiator

Distributed Computations

- **ResourceManager** (master) is the entry point for all YARN application tasks.
  - An application is a set of tasks (long running or not) that consume cluster resources (cpu/memory/disk/network) to perform work.
  - Has **ApplicationsManager** to submit/monitor applications and *Scheduler* to allocate resources.
- **NodeManagers** (workers) manages **containers** on each node.
  - Application-specific tasks are launched in containers.
  - Each application has one **ApplicationMaster**, which coordinates the efforts of all its containers.
  - A container can fail and its work will be rescheduled.
- Replaces MapReduce v1, which had scalability issues (and only supported MapReduce computations).
- Other job types include **Tez**, **Spark**, **DataTorrent,** and **???** (open-ended).

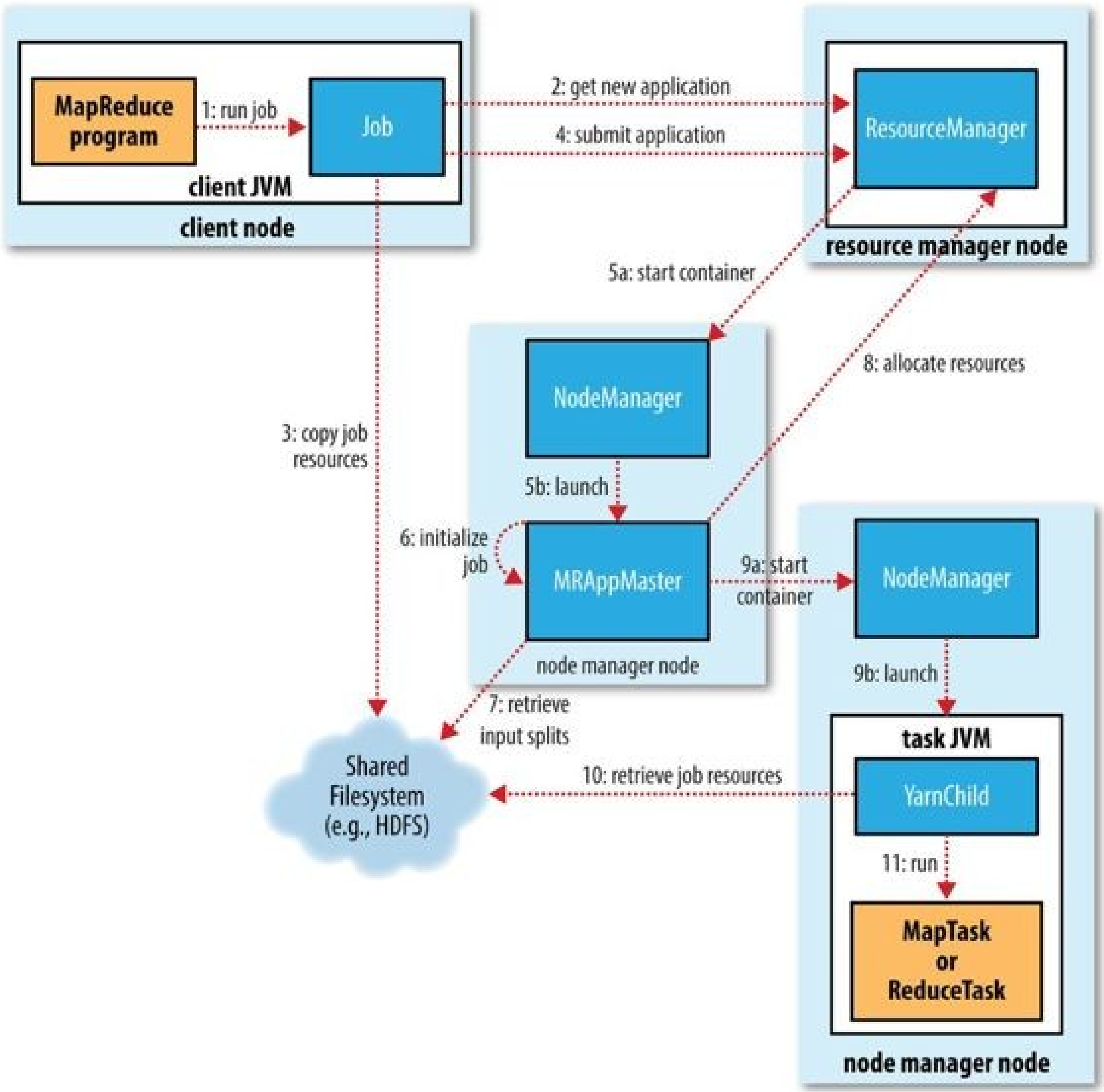


YARN



Application client
1: submit YARN application
ResourceManager
client node
resource manager node
2a: start container
NodeManager
3: allocate resources (heartbeat)
2b: launch
Container
Application process
node manager node
4a: start container
NodeManager
4b: launch
Container
Application process
node manager node

YARN



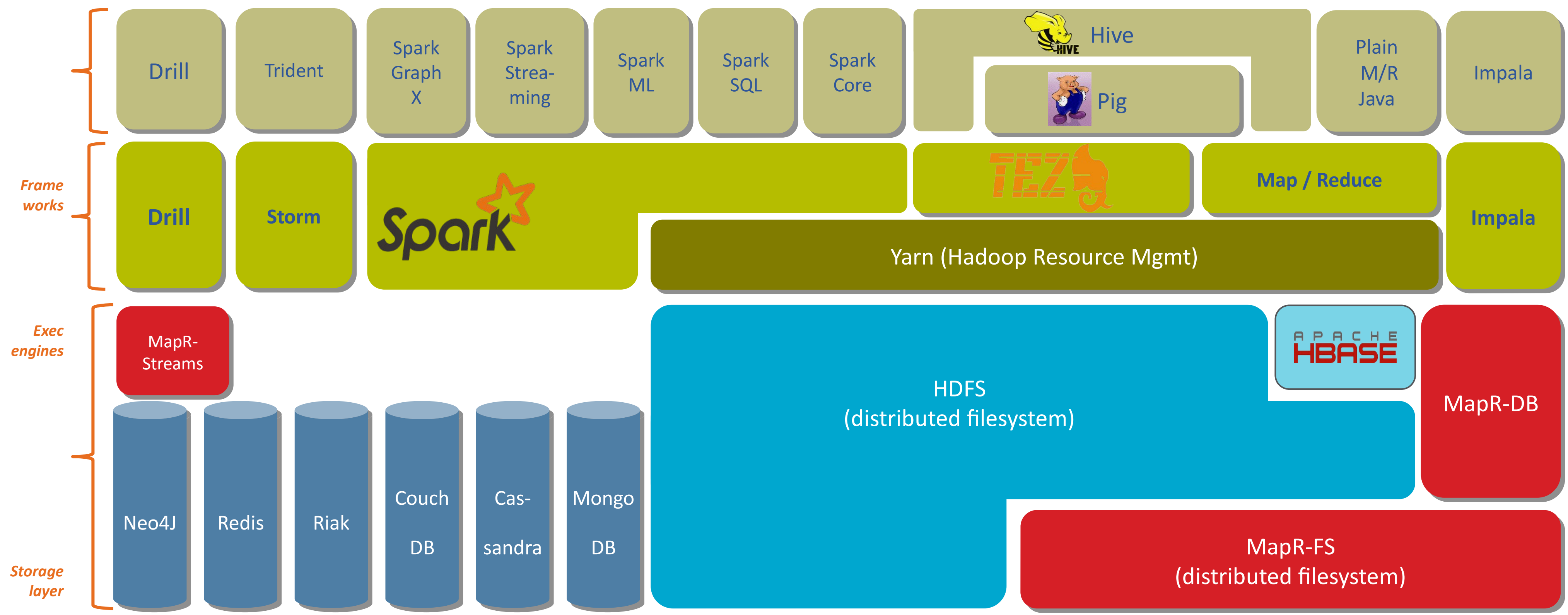
MapReduce program
1: run job
Job
client JVM
client node
2: get new application
4: submit application
ResourceManager
resource manager node
5a: start container
NodeManager
8: allocate resources
3: copy job resources
5b: launch
6: initialize job
MRAppMaster
9a: start container
NodeManager
node manager node
9b: launch
7: retrieve input splits
task JVM
Shared Filesystem (e.g., HDFS)
10: retrieve job resources
YarnChild
11: run
MapTask or ReduceTask
node manager node

Hadoop/MR Issues

- Not good for random access
- Lacks: indexing, metadata layer, query optimizer, memory management
- No ACID support
- Not suited for Iterative algorithms (most of ML)
- Latency

The Hadoop ecosystem

Layered architecture

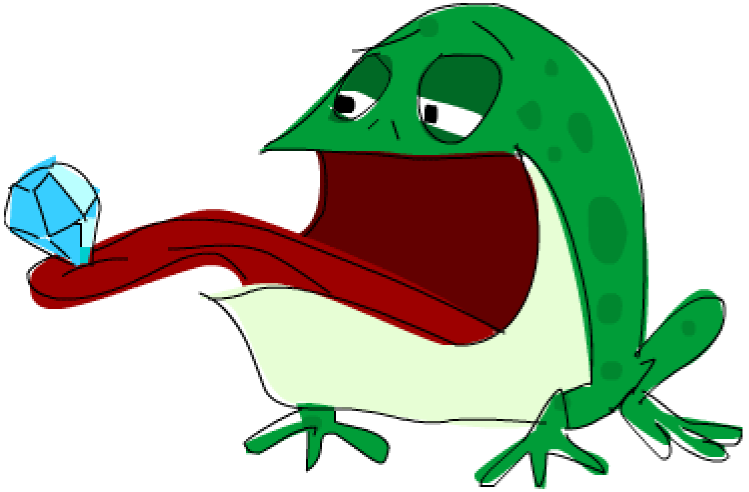


Hadoop vendors

Vendor	Short (Hadoop distribution)	Main theme
Cloudera / Hortonworks	CLD/CDH HWX/HDP now CDP	OSS with proprietary tooling (Cloudera Manager / Cloudera Navigator
MapR	MAPR	Only API are OSS, all subsystems are full commercial rewrite (C++)
Amazon	EMR	Fully OSS in terms of Hadoop services but fully managed
Microsoft	Azure HDInsights	Based on HDP + REST-only services (WebHDFS, WebHCat, WebSpark with Livy, etc.)
Google	Dataproc	Fully OSS But their own systems too

3) Big Data Architectures

A weird intuition



What if we think of big data as a database that exploded ?

Big Data Elements

- ***Distributed Filesystems:*** HDFS / S3 / GS
- ***Processing Framework:*** Hadoop / Spark / Flink / Beam-Dataflow
- **Distributed Coordination:** Zookeeper / Consul / etcd
- **Query Engines:** Hive / Pig / Spark SQL
- **Distributed Databases:** HBase / Cassandra / Mongo?
- **Distributed Log (Broker):** Kafka / Kinesis / PubSub
- **In-Memory Caches:** Memcache / Redis
- **Search Engine / Indexer:** Elasticsearch / Solr
- **Cluster Framework:** Mesos / Kubernetes

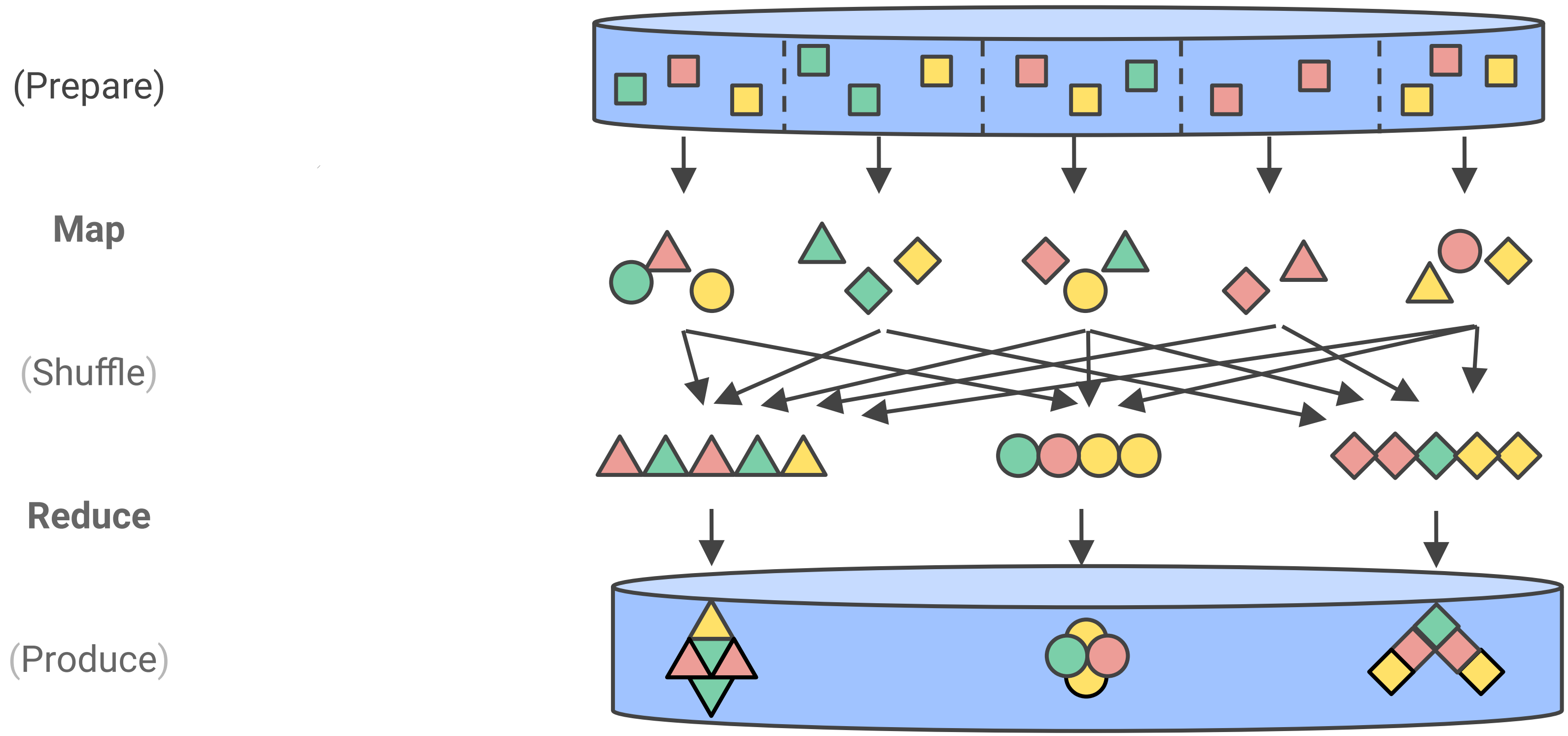
Some people prefer the term DataStore to unify most of these

Cluster Glossary

Technology	Master	Workers
HDFS	Name Node	Data Nodes
YARN	Resource Manager (Applications Manager / Scheduler)	Node Managers
YARN application	Application Master	Containers
MapReduce v1	Job Tracker	Task Tracker
MapReduce v2	MRApplicationMaster	Map / Reduce tasks (Yarn Child)
Spark	ClusterManager	Executors
Storm	Nimbus	Supervisors
HBase	Master	Region Server
Zookeeper	Leader (elected)	Followers
Kafka	Leader (per topic)	Replicas

3.1. Batch / Bounded Data

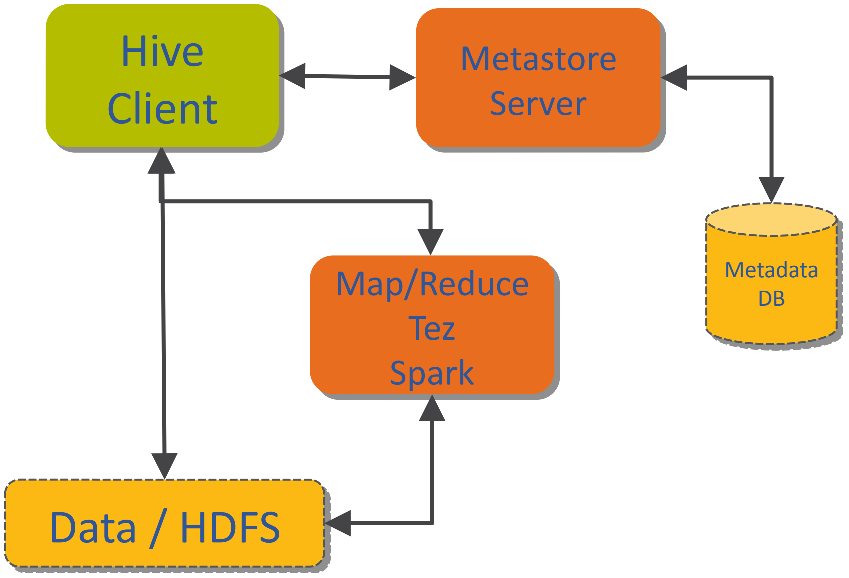
MapReduce: Batch Processing / Bounded Data



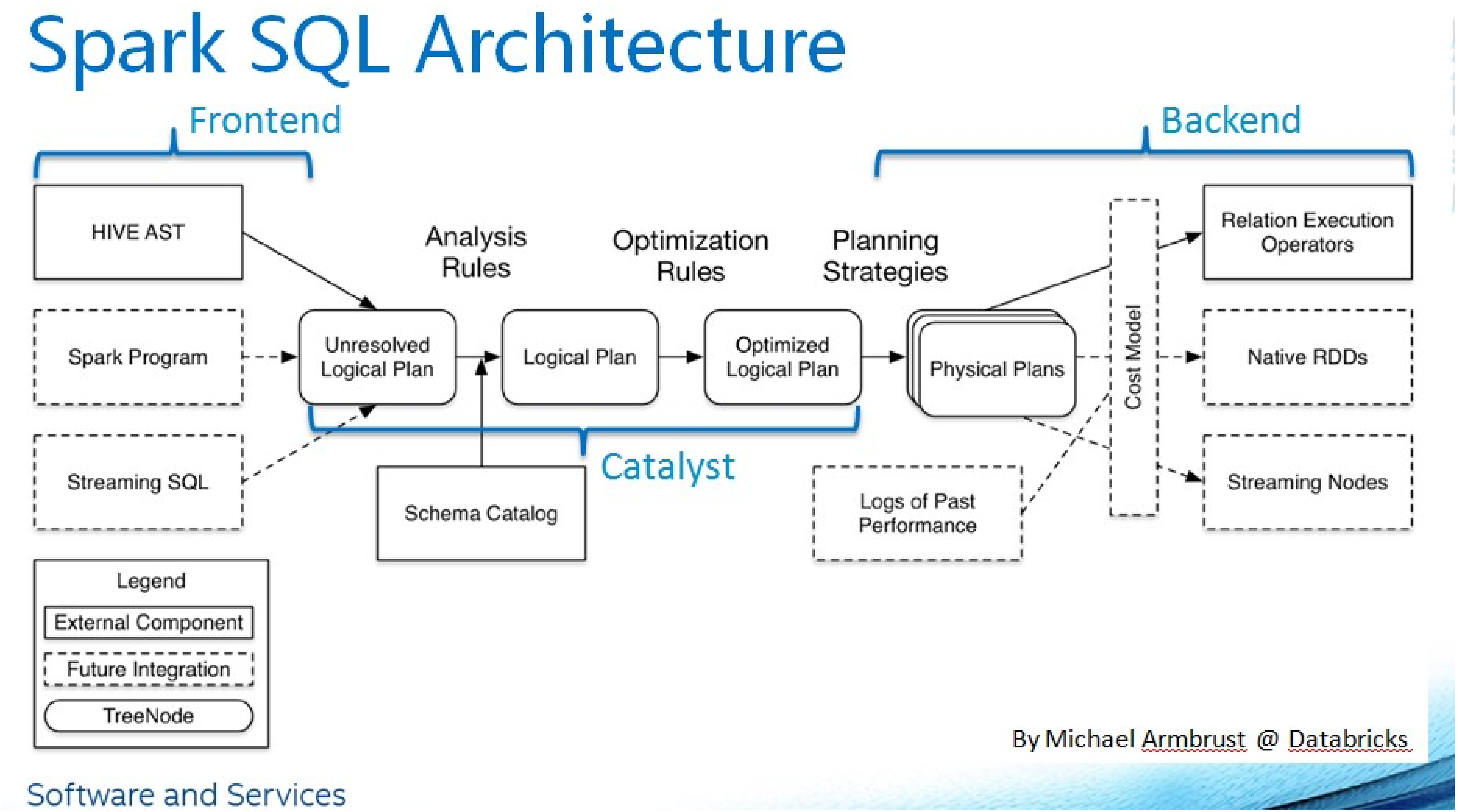
Hive

SQL-like Query Engine

- Hive describes a data warehousing infrastructure for storing data, schemas and other metadata (**HCatalog**).
- **U**ses the **HQL** query language (SQL-ish, accessible through JDBC).
- **HiveServer** for making metadata queries.
- **Beeline** shell for interactive queries. (in Hive 2)



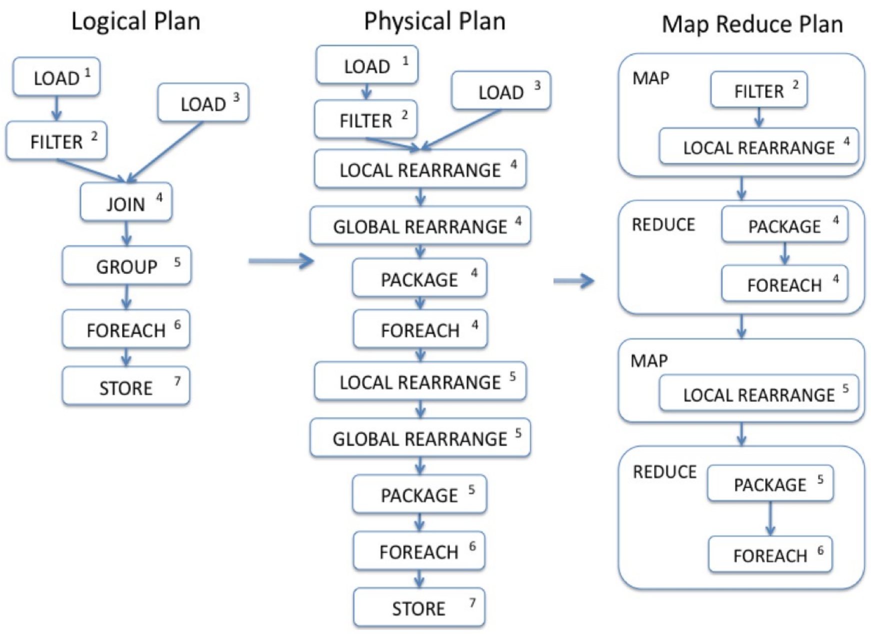
SQL like systems



Pig

Data Flow Engine

- High-level dataflow system aiming at a sweet spot between SQL and M/R
- From 0.14 (and 0.16 for Spark) supports generation of Tez Plan in parallel of M/R Plan, leading to high increase in performance without any change in the Pig Latin
- Strength are on its fast coding capabilities and relatively small performance overhead
- Its main weakness is on the maintainability aspect, which is difficult (no particular tooling for automation)

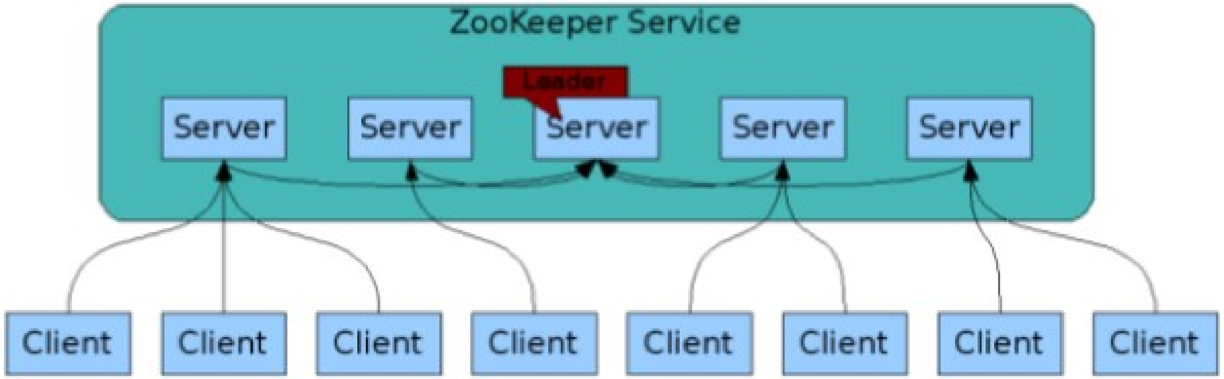


Zookeeper

Distributed consistent key-value store used to coordinate distributed systems

- Distributed coordination
  - Group Membership
  - Leader Election
  - Locking
  - Synchronization
  - Publish / Subscribe
- Use cases
  - Config Management
  - Cluster status
  - DNS (Name Service)
  - Service Registry

Zookeeper Architecture



- ✓ **distributed** over a set of machines and **replicated**.
- ✓ all servers store a **copy of the data** (in memory as well as local file system)
- ✓ a **LEADER** is elected at the startup
- ✓ LEADER will do **atomic broadcast** to all other servers (**Z**ooKeeper**A**tomic**B**roadcast)
- ✓ strong **ordering guarantees**
- ✓ **no partial** read/writes

https://events.linuxfoundation.org/sites/events/files/slides/ZooKeeper%20in%20the%20Wild.pdf

Apache HBase

- NoSQL datastore on top of HDFS
- Based on Google’s BigTable
- Random Reads/Writes
- No SQL, No Joins
- Design around data access
- CP
- Users: Yahoo, Uber, etc
- Use if > 10k ops per second on huge data and access patterns are well-known

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7, 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

`info` **Column Family**

Row key	Column key	Timestamp	Cell value
cutting	info:height	1273516197868	9ft
cutting	info:state	1043871824184	CA
tlipcon	info:height	1273878447049	5ft7
tlipcon	info:state	1273616297446	CA

`roles` **Column Family**

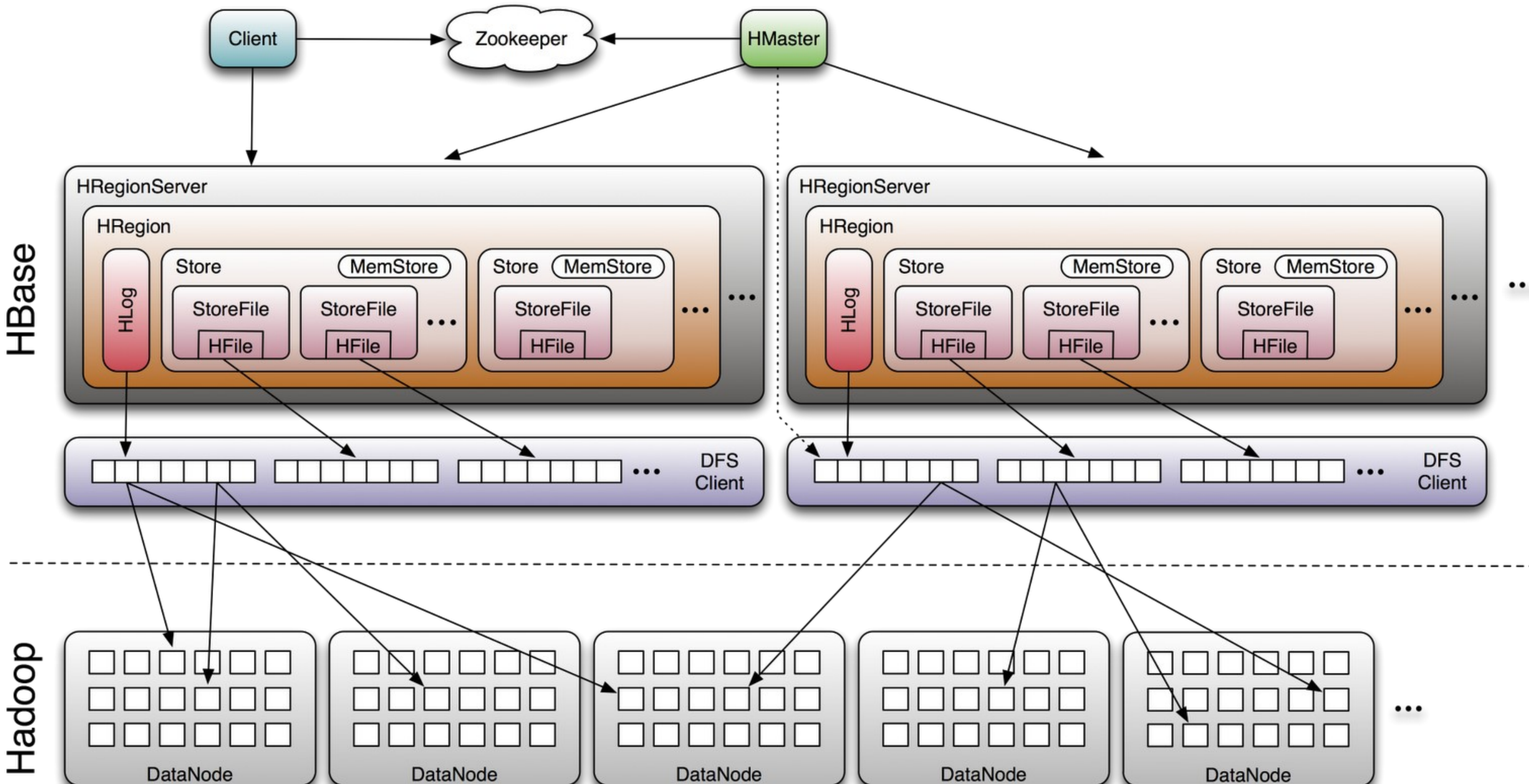
Row key	Column key	Timestamp	Cell value
cutting	roles:ASF	1273871823022	Director
cutting	roles:Hadoop	1183746289103	Founder
tlipcon	roles:Hadoop	1300062064923	PMC
tlipcon	roles:Hadoop	1293388212294	Committer
tlipcon	roles:Hive	1273616297446	Contributor

Sorted on disk by Row key, Col key, descending timestamp

Milliseconds since unix epoch

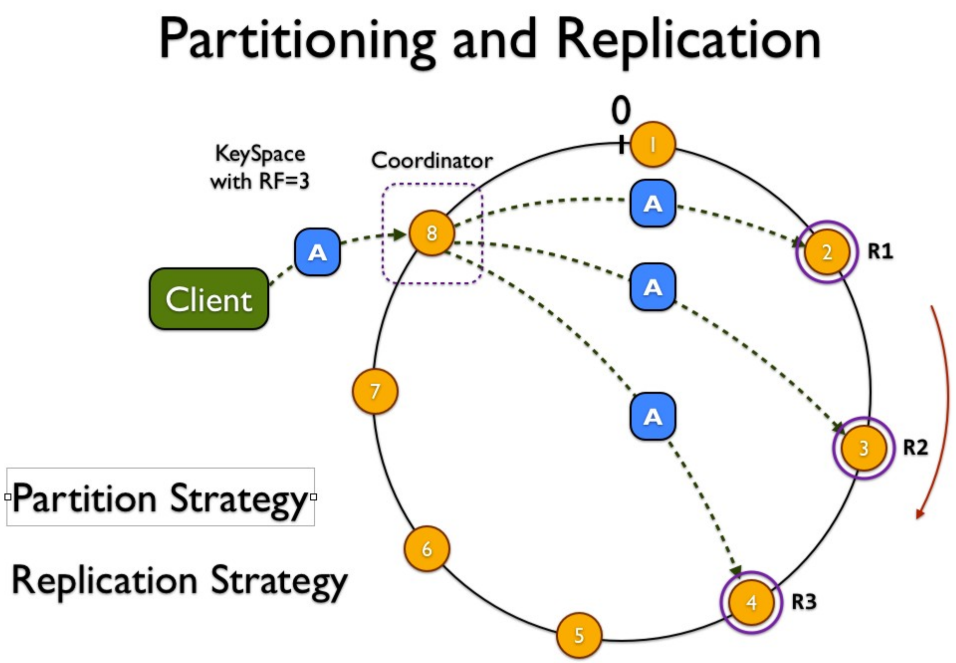
cloudera

Hbase Big Picture



Apache Cassandra

- Amazon Dynamo + Google BigTable made by facebook
- Peer-to-peer (gossip) !
  True Horizontal Scalability
- Easier to admin (No ZK)
- CQL (SQL-like)
- Sorted Map of sorted maps
- Eventual Consistency (AP) for Low Latency
- Users: Netflix, Twitter
- Use when you need faster writes and you don't care about consistency and you need linear scalability.



Apache Spark

"Spark is a fast and general engine for large-scale data processing".
Write programs in terms of **transformations** on **distributed datasets.**

Resilient Distributed Datasets (RDDs)

- Collections of objects spread across a cluster
  - stored in RAM or on Disk (or both) via API
  - user can decide when to keep in memory and when to persist
- Built through parallel transformations
- Automatically rebuilt on failure

Operations

- **Transformations** (e.g. map, flatMap, filter, groupBy)
- **Actions** (e.g. count, collect, save)

http://spark.apache.org/docs/latest/programming-guide.html

Spark bring interactive data mining and iterative algorithms. Huge Impact !

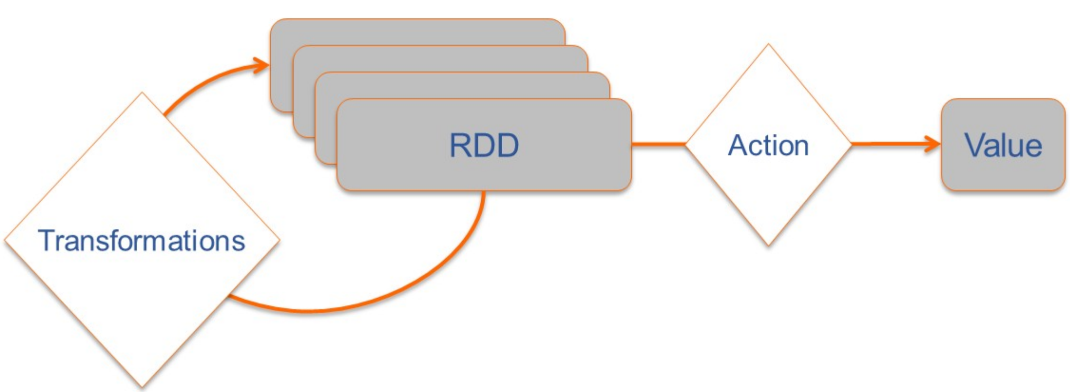
Spark Programming Model

Resilient distributed datasets (RDDs)

- » Immutable, partitioned collections of objects
- » Created through parallel *transformations* (map, filter, groupBy, join, …) on data in stable storage
- » Can be *cached* for efficient reuse

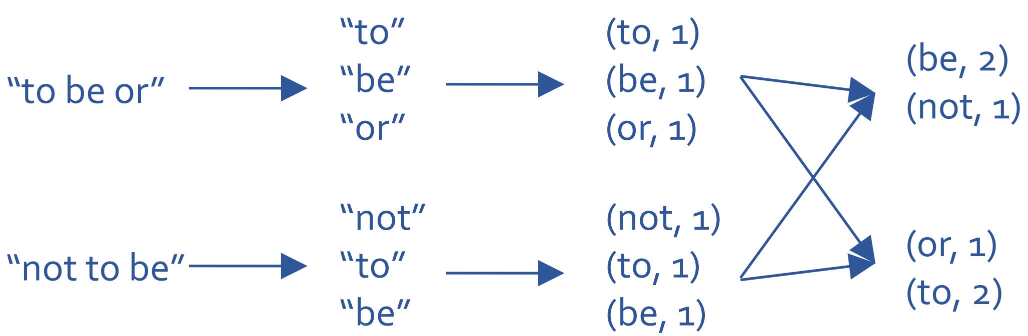
Actions on RDDs

- » Count, reduce, collect, save, …



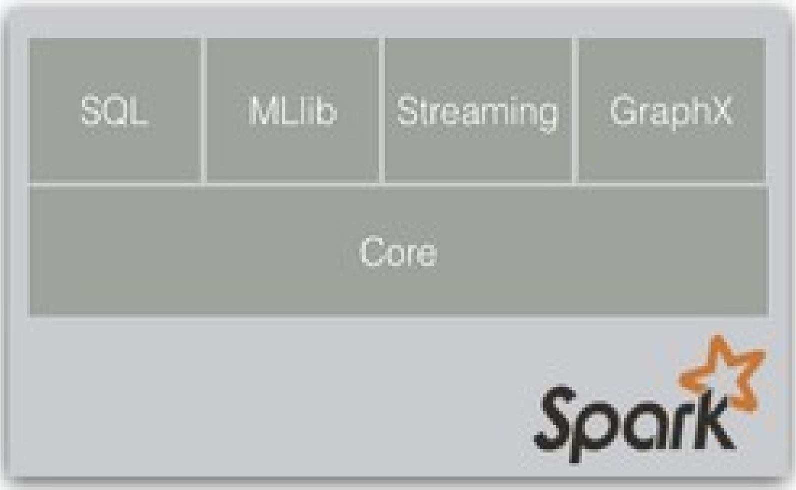
Spark Word Count

```
val file = sc.textFile("hdfs://.../hamlet.txt")

val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://.../hamletCount.txt")
```



API Like Distributed Collections, a little bit LINQ like (based on Microsoft DryadLINQ)

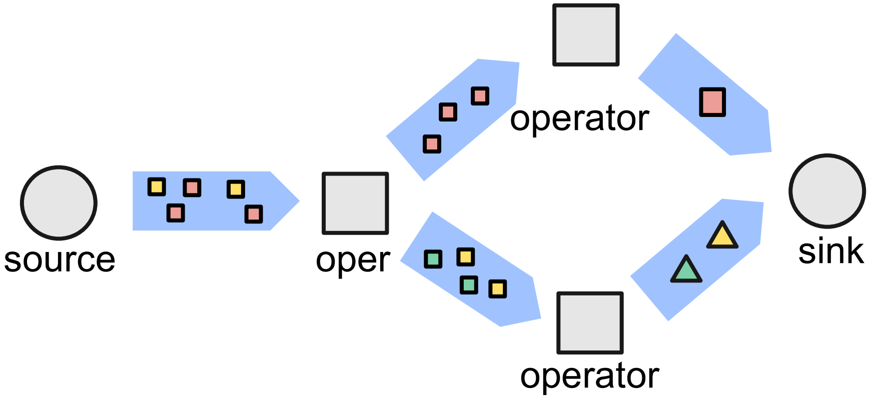
Spark



- **Spark Core**
  - Set of APIs to support workflow transformations on RDDs
- **SQL**
  - Allows relational queries expressed in SQL, HiveQL or Scala to be executed through a data abstraction called a SchemaRDD.
  - Supports Parquet files, JSON, data stored in Hive
- **MLlib** (said M-L-lib for Machine Learning library)
  - Algorithms include support vector machines (SVM), logistic regression, decision trees, naïve Bayes and k-means clustering.
- **Streaming**
  - Stream processing ingested from sources like Kafka, Flume, Twitter, TCP
- **GraphX**
  - Graph analytics for applications like social networks.

3.2. Stream Processing / Unbounded Data

Process data *immediately* on arrival
Continuous **Unbounded** data
Continuous Operations



Advantages:
Low latency

Projects:
Apache Storm, Flink

Issues:
Correctness compromised:
- When to aggregate ?
- How to compute aggregates?

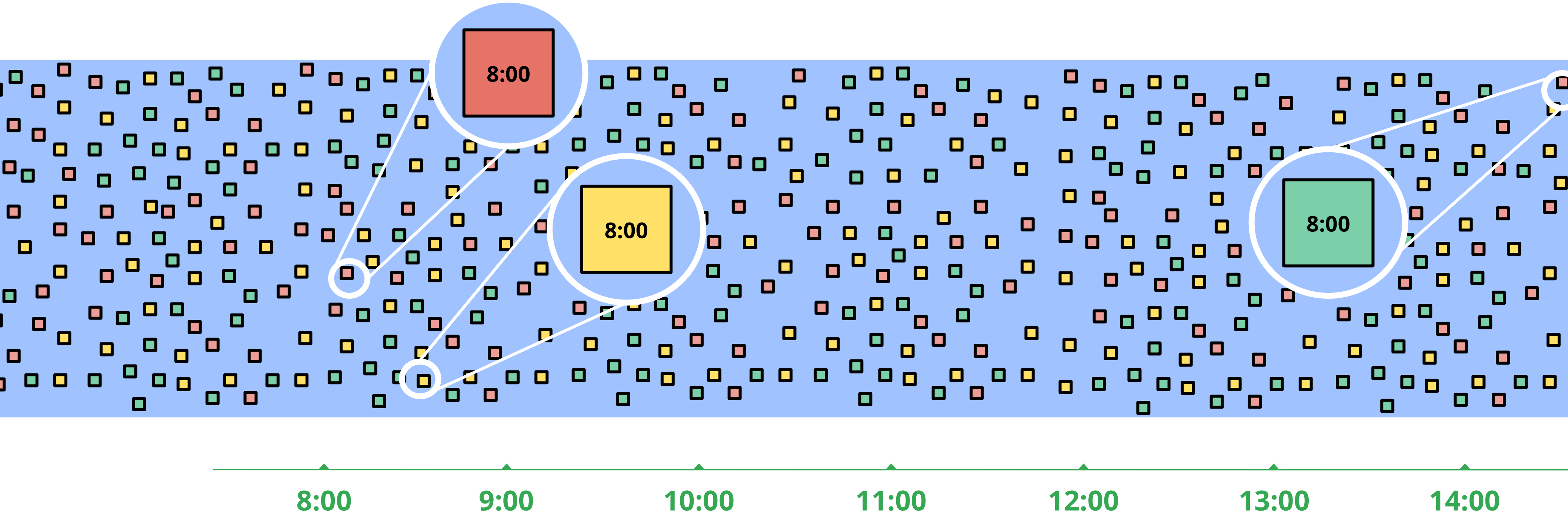
Nathan Marz
Storm creator

Guarantees in streaming are harder to achieve

Guaranteeing Message Processing

- ***at-most-once delivery***,
  - Drops messages if they are not processed correctly, or if the machine doing the processing fails
  - Processes messages in the order they were produced
  - Could lead to loss of data, but ok if doing approximations
- ***at-least-once delivery***,
  - tracks whether each input tuple (and any downstream tuples it generates) was successfully processed within a configured timeout
  - any tuples that are not fully processed within the timeout are re-emitted.
  - This implies the same tuple can be processed more than once, and that messages can be processed out-of-order.
- ***exactly-once semantics***
  - extends at-least-once mode
  - but the state implementation allows duplicates to be detected and ignored.
  - batches are processed in a strictly sequential order
  - **This mode is possible with Spark only if you manage yourself the partition offsets**

Streaming – Hardcore issue → late data

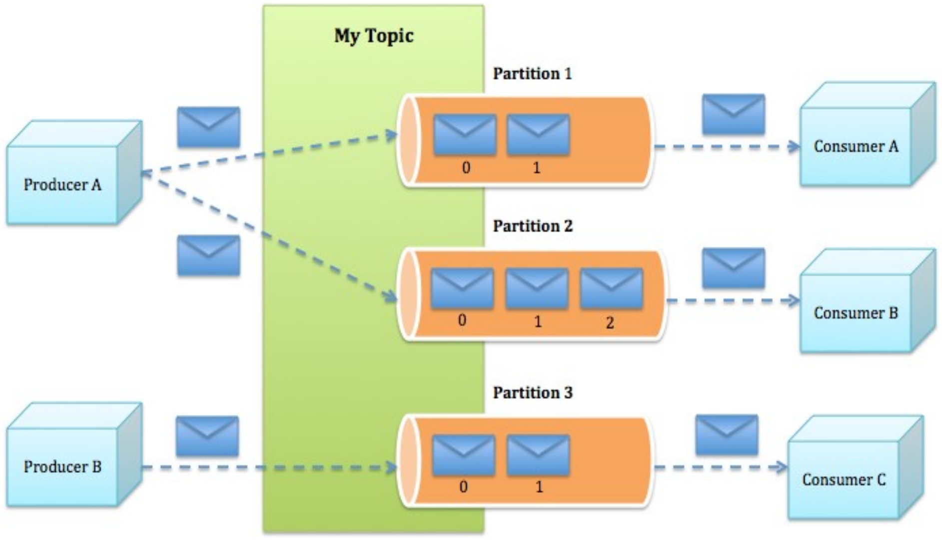


We will see this later on more detail when we discuss about Beam!

Kafka

Distributed message broker : the backbone component of the Lambda architecture

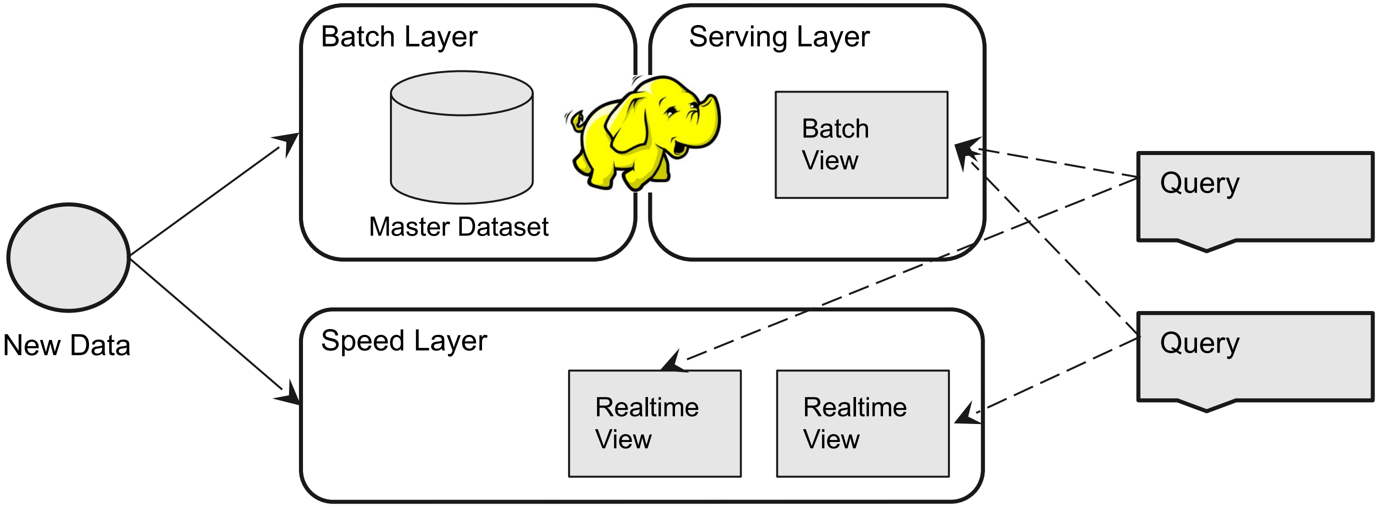
- Distributed message broker (Similar to JMS/ActiveMQ/Rabbit MQ)
  - Organized as a replicated, partitioned commit log.
- **Brokers** store partitions of **messages** per **topic**.
  - One broker is elected **leader**, other brokers are **replicas**.
- **Producers** generate messages, sent and stored into the broker's commit log.
- **Consumers** fetch messages as they arrive in a topic, from beginning or maintaining a client-side index.
- Strengths
  - High throughput for pub-sub achieved thanks to in-memory end-to-end communication
  - Can easily scale and store terabytes of messages and serve as backlog
- Weaknesses
  - Heavy use of Zookeeper for coordination
  - API still young (consumer API refactored in 0.9)



- A consumer belongs to a single consumer group
- Consumer groups control one offset per partition per topic
- Offsets can be stored either in Zookeeper (default), in Kafka (_offsets) or on the consumer side (ex: Own jdbc)

Lambda Architecture

How can we have the best of both models (correctness + low latency) ?



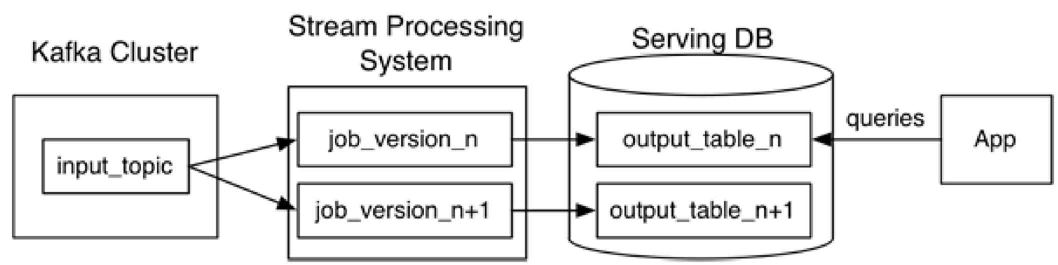
Issues: Two different programming models to maintain

Streaming First / Kappa

Batch is a subset of streaming argument

More powerful stream processing engines
- Deal with **state**, checkpoints, versions

Kappa Architecture via the distributed log abstraction (Kafka)



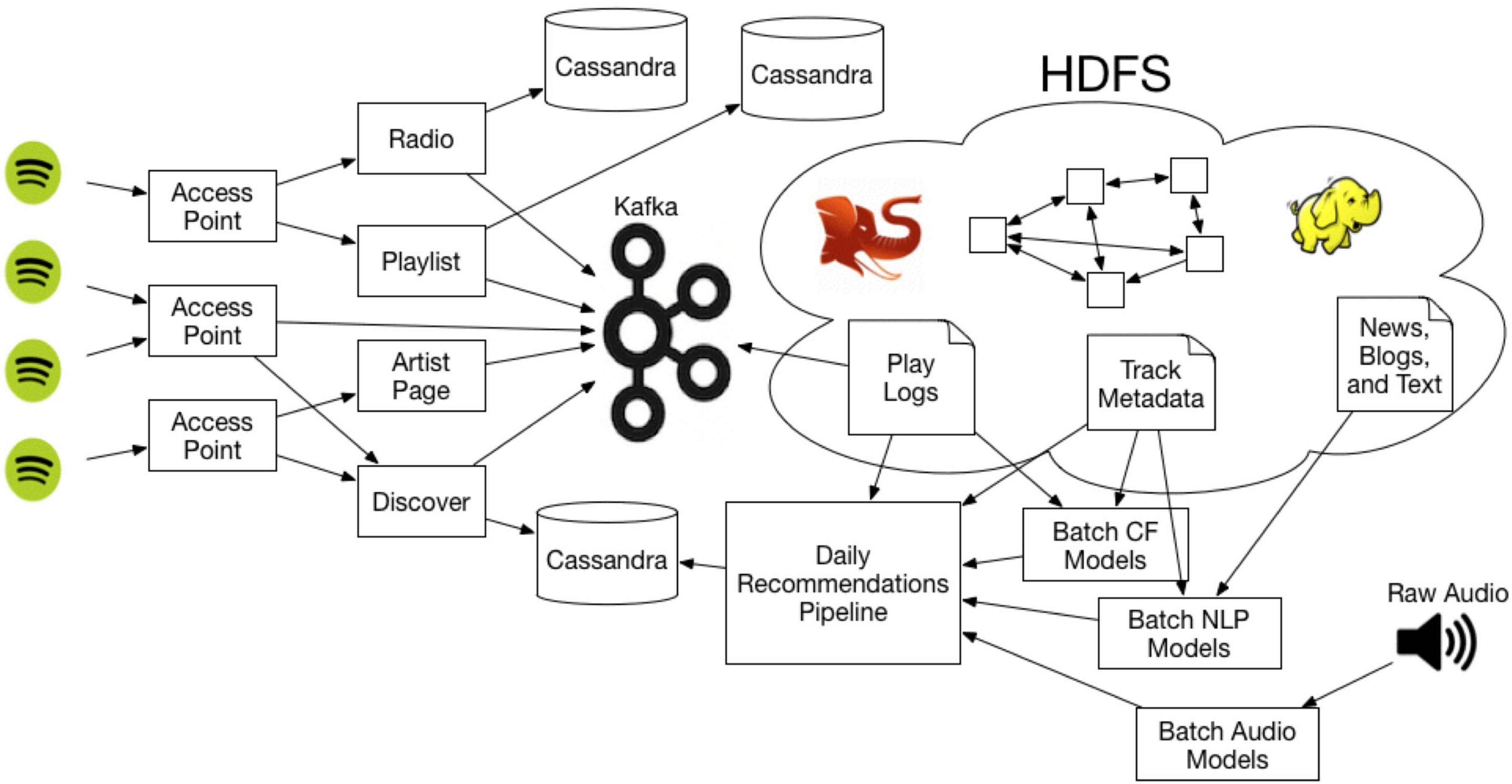
Issues:
Too many different stream processing frameworks
Infinite, out of order data sources require more advanced semantics

Beam

More details:

https://docs.google.com/presentation/d/17eq17-4KYvF1-2sCOo0sSUdm6gj4h6sWLhLDUYOe1cU/edit?usp=drive_web

An example – Music Recommendation at Spotify



http://www.slideshare.net/eshvk/spotifys-music-recommendations-lambda-architecture

Current trends (2019)

Is Hadoop still a thing?

What ‘killed’ Hadoop?

- Hype! Hadoop as the solution to everything
- Operational Complexity
- Evolution (this is positive)
  - Richer programming models
  - Improved faster execution models (memory based)
  - The rise of containers (better support for non Java jobs)

But not dead yet: The Rise of the data lake

- A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale.

Hadoop’s HDFS and its ecosystem tools fit this definition nicely

Basic elements in a data processing system (Hadoop case)

- Storage System: HDFS
- Programming Model: MapReduce
- Execution System: YARN

So Hadoop is dead what now?

- HDFS replaced by Cloud Filesystems*
- MapReduce replaced by modern programming models e.g. Spark, Beam
- YARN replaced by Kubernetes or by Cloud self-managed systems (Databricks Managed Spark + Google Dataflow)

Rise of cloud services

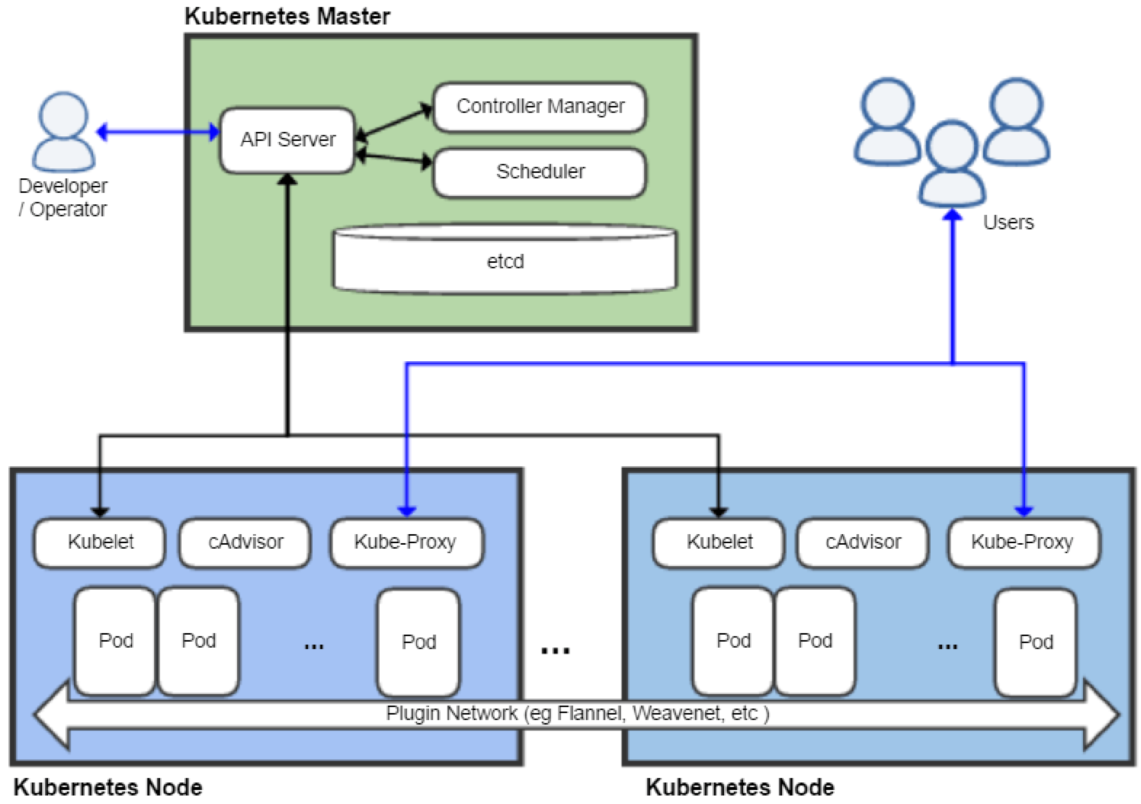
To deal with Hadoop operational issues

- Cheaper* and easier storage
- On demand clusters in minutes
  - NY Times transformed 150 years old archive of articles and images into web form on 36h [1]
- Self-managed job execution making operations trivial
- Serverless models (FAAS)

[1] https://open.blogs.nytimes.com/2008/05/21/the-new-york-times-archives-amazon-web-services-timesmachine/
* (until you want data out)

Kubernetes

- Kubernetes is a portable, extensible, open-source platform for managing **containerized** workloads and services.
- ‘Resource management system’ (cluster)

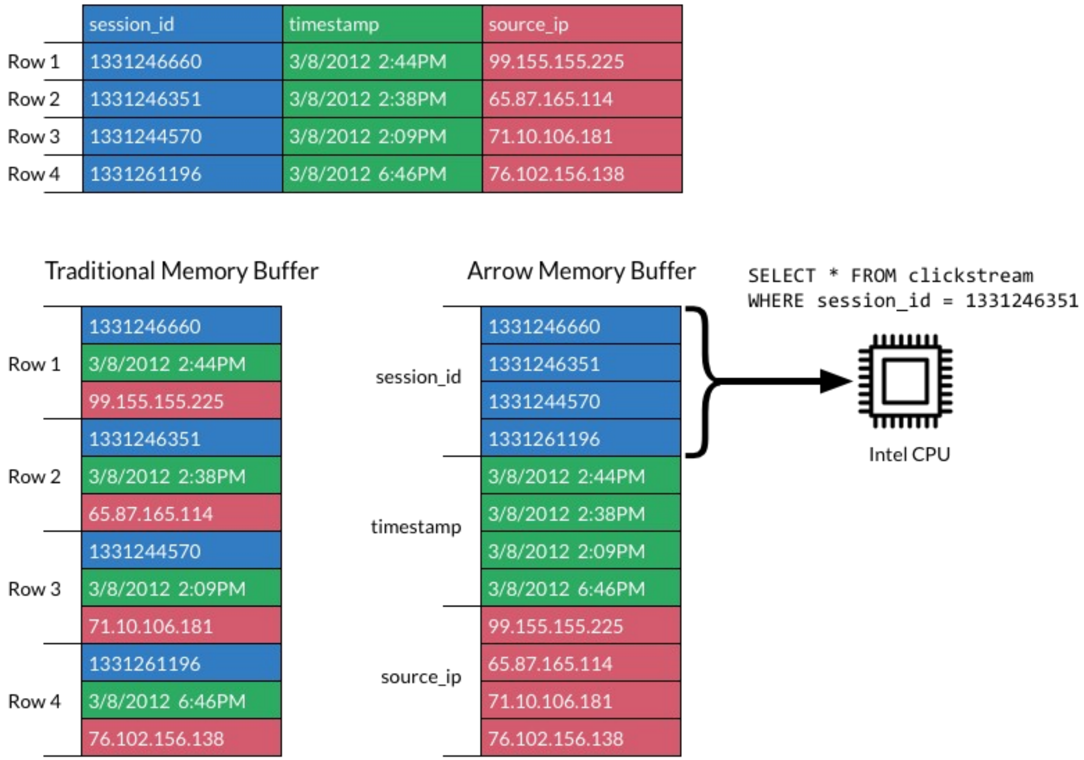


Rise of polyglotism

- Most Big Data systems are Java/JVM based
- Java is not good for interactive data exploration
- Machine learning and ecosystem tools
  - Interactive Notebooks
  - Dataframe-like frameworks

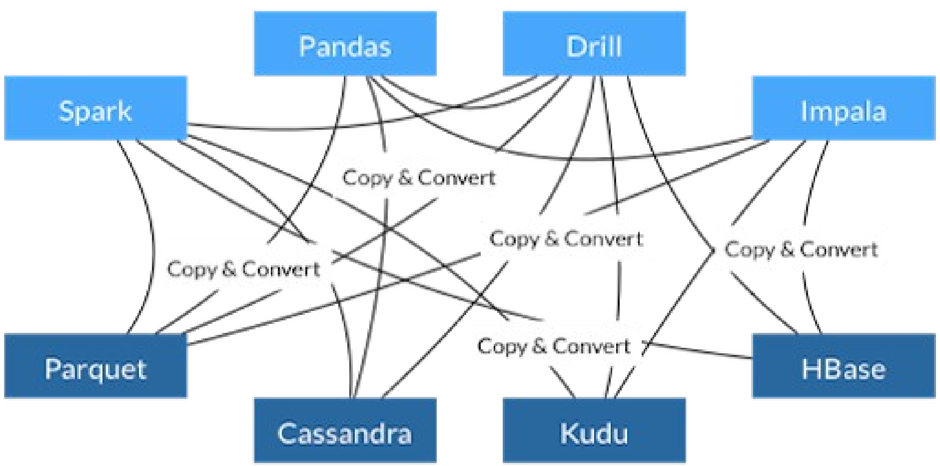
Apache Arrow

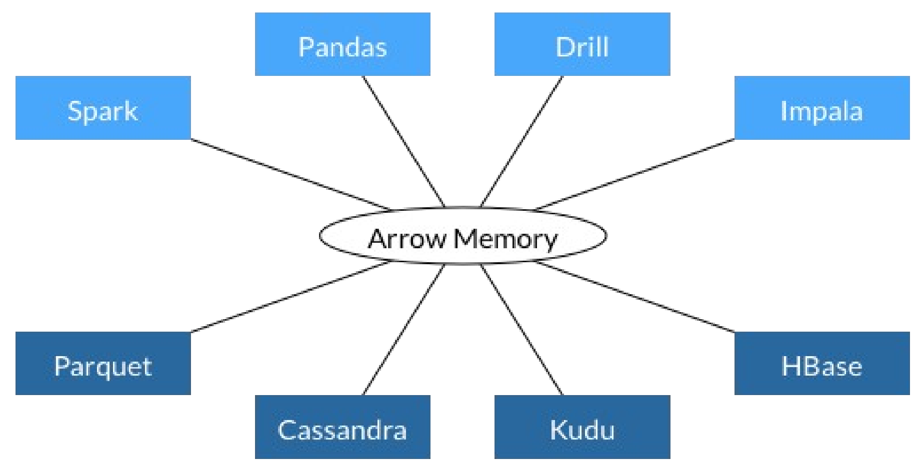
- Language-independent columnar memory format for flat and hierarchical data



Apache Arrow

- Idea: Finish the waste of CPU cycles on (de)serialization



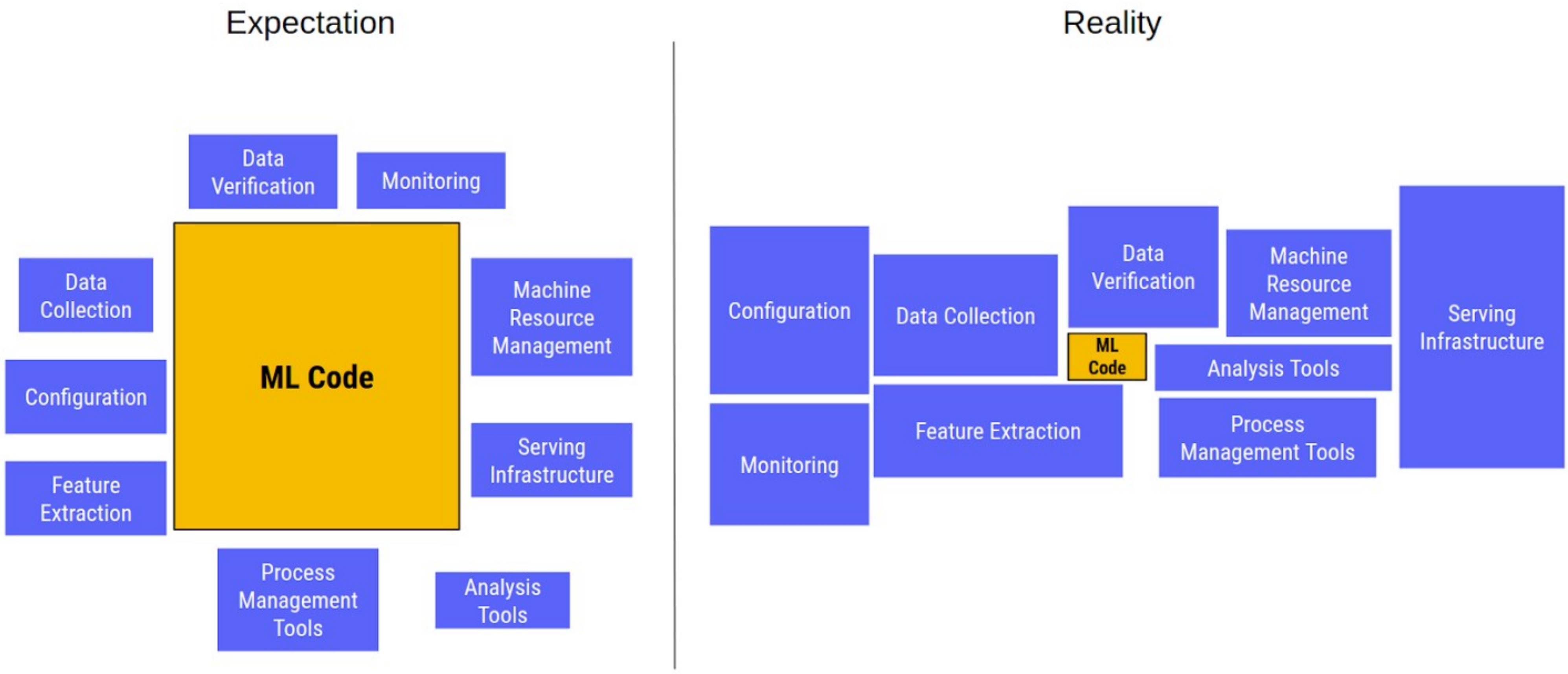


Delta vs Apache Iceberg

- Table formats on top of distributed file systems
- Solve many of the Data lake scenario issues:
  - Schema evolution supports add, drop, update, or rename, and has no side-effects
  - Hidden partitioning prevents user mistakes that cause silently incorrect results or extremely slow queries
  - Partition layout evolution can update the layout of a table as data volume or query patterns change
  - Time travel enables reproducible queries that use exactly the same table snapshot, or lets users easily examine changes
  - Correctness issues with eventually-consistent dfs
- Mostly are clever representations on top of Parquet

Machine Learning is the new deal

- Big Data is the enabler of modern ML



Conclusions

- The design space is huge
- No tool solves all the problems
- Use the right tool for the right problem
  - If you can deal with structured data or KV structured data just do it.
  - If you have multiple sources or need a more powerful data processing model use it
- Trade-offs and constraints are all around
- Streaming is the new frontier (since 2012)
- Complexity is the ultimate enemy !
- Technology evolve but fundamental principles are the same
- There is no escape :-) these systems are here to stay

References

- Papers:
  - MapReduce by Google
  - Big Table by Google
  - Dynamo/Cassandra (this one is the greatest hits of DS)
  - Immutability changes everything by Pat Helland
- Books:
  - Designing Data-Intensive Applications. Martin Kleppmann
  - Big Data principles and best practices. Nathan Marz
  - The Hadoop Definitive Guide. Tom White.

Extra Slides

CAP Theorem

The CAP Theorem states that, in a distributed system, you can only have two out of the following three guarantees across a write/read pair:

- *C_onsistency_* - A read is guaranteed to return the most recent write for a given client. *All clients have the same view of data.*
- *A_vailability_* - A non-failing node will return a reasonable response within a reasonable amount of time (no error or timeout). *Writable in the face of a node failure.*
- *P_artition Tolerance_* - The system will continue to function when network partitions occur. *Processing can continue in the face of network failure.*

In reality you **cannot** sacrifice **P**, so is it **CP vs AP**

Eric Brewer
CAP author

ACID vs BASE

- ACID properties of transactions
  - Atomicity, Consistency, Isolation, Durability
- You can’t guarantee in every system.
  - Trade guarantees for performance
- BASE: Basically Available Soft-State, Eventually Con sistent
- Drop consistency and isolation to improve availability and performance.
- ACID vs BASE are our design spectrum