GENERAL ASSEMBLY

INTRO TO DATA SCIENCE
LECTURE 18: BIG DATA TECHNOLOGIES

Paul Burkard
08/27/2015

LAST TIME:

- TIME SERIES
  - AUTOREGRESSIVE MODELS
- DATA STREAM MINING
  - INCREMENTAL ALGORITHMS

QUESTIONS?

I. BIG DATA
II. HADOOP (MAPREDUCE + HDFS)
III. THE HADOOP ECOSYSTEM
IV. SPARK
HANDS-ON: PYSPARK
V. AWS

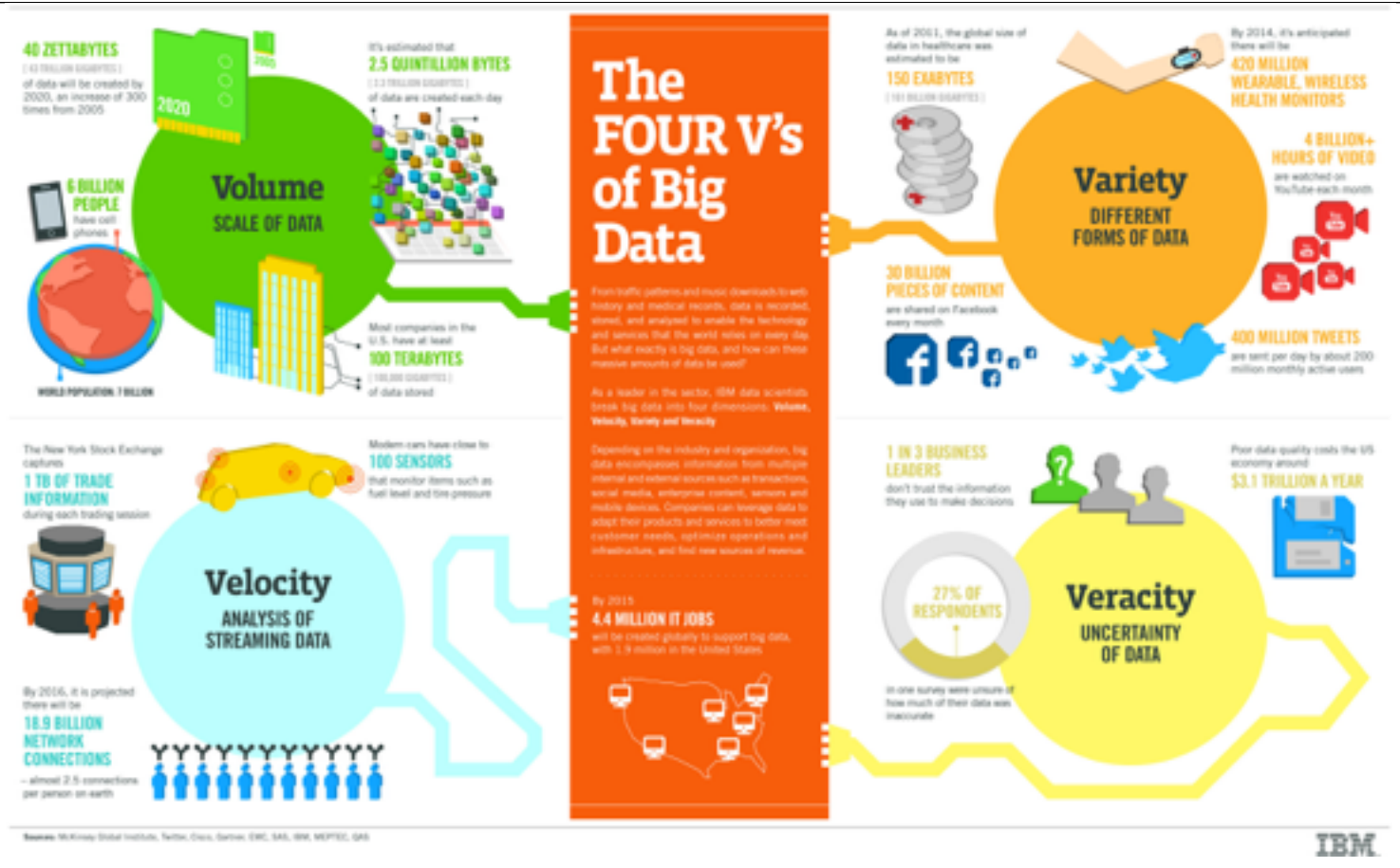
- What is big data?
  - What challenges does it pose?
- What is Hadoop?
  - What is MapReduce?
  - What is HDFS?
- What are the elements of the Hadoop stack and what do they do?
- What is the goal of Spark?
- What services does AWS provide?

I. BIG DATA

Q: ***Big Data*** *is a hot topic these days, what does it actually refer to?*

A:

- ***Volume****: more data than can fit in memory on 1 machine*
- ***Velocity****: data coming in faster than we can process (think Twitter)*
- ***Variety****: Data across all different types*
- ***Veracity****: Uncertainty of data*


40 ZETTABYTES
of data will be created by 2020, an increase of 300 times from 2005
2005
2020
6 BILLION PEOPLE
have cell phones
WORLD POPULATION: 7 BILLION
Volume
SCALE OF DATA
It's estimated that
2.5 QUINTILLION BYTES
of data are created each day
Most companies in the U.S. have at least
100 TERABYTES
of data stored
The FOUR V's of Big Data
From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?
As a leader in the sector, IBM data scientists break big data into four dimensions: Volume, Velocity, Variety and Veracity
Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.
By 2015
4.4 MILLION IT JOBS
will be created globally to support big data, with 1.9 million in the United States
As of 2011, the global size of data in healthcare was estimated to be
150 EXABYTES
Variety
DIFFERENT FORMS OF DATA
By 2014, it's anticipated there will be
420 MILLION WEARABLE, WIRELESS HEALTH MONITORS
4 BILLION+ HOURS OF VIDEO
are watched on YouTube each month
30 BILLION PIECES OF CONTENT
are shared on Facebook every month
400 MILLION TWEETS
are sent per day by about 200 million monthly active users
The New York Stock Exchange captures
1 TB OF TRADE INFORMATION
during each trading session
Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure
Velocity
ANALYSIS OF STREAMING DATA
By 2016, it is projected there will be
18.9 BILLION NETWORK CONNECTIONS
– almost 2.5 connections per person on earth
1 IN 3 BUSINESS LEADERS
don't trust the information they use to make decisions
27% OF RESPONDENTS
in one survey were unsure of how much of their data was inaccurate
Poor data quality costs the US economy around
$3.1 TRILLION A YEAR
Veracity
UNCERTAINTY OF DATA
IBM

*Q: **Big Data** is a hot topic these days, what does it actually refer to?*

A:

We're talking about more data than can fit on a single computer.

We have exponentially growing data AND exponentially growing computing power.

- *The Large Hadron Collider produces about 30 petabytes of data each year*
- *A petabyte is 10^{15} bytes, or a million gigabytes*
- *Their data center contains over 10,000 servers with around 100,000 processor cores*
- *They can process around 1 petabyte of data per day!*

Q: How do we handle all of this data?

A: One approach would be to build a giant supercomputer.

But this has some obvious drawbacks:

- *expensive*
- *difficult to maintain*
- *scalability is bounded*

Instead of one huge machine, what if we got a bunch of regular (commodity) machines?

This has obvious benefits!
- cheaper
- easier to maintain
– scalability is unbounded (just add more nodes to the cluster)

This idea leads us to ***Hadoop***

II. HADOOP

*Q: What is **Hadoop**?*

*A: A platform/framework for **distributed computing**.*

Arose out of several papers from Google in the early 2000s.

Allows scaling big data analysis outwards over clusters of computers.

Handles fault tolerance, scheduling, monitoring

*Q: What is **Hadoop**?*

A: At its core, it consists of 2 components:

- ***Hadoop Distributed Filesystem (HDFS):***
  - *Filesystem optimized for dealing with big data on clusters of computers*
- ***MapReduce:***
  - *Functional programming paradigm for simplifying large scale highly parallelizable computations*

HDFS

*Q: What is **HDFS**?*

*A: **Hadoop Distributed Filesystem***

*– Based on **Google Filesystem (GFS)** from early 2000s paper.*

– Distributed filesystem across cluster of machines

– Highly fault tolerant

– Designed to be deployed on commodity hardware

– Can store and work with massive files across many computers

– A cluster is a group of machines (nodes) working together

NameNode: *Runs on a master node that tracks and directs the storage of the cluster*

DataNode: *Runs on slave nodes that make up the majority of the machines in the cluster. Handles storage of data files in blocks (much larger than usual file block size) which are replicated several times (3 by default) on data nodes across the cluster.*

Client Nodes: *Not actually storing data, but may run other Hadoop services and processes.*

MAPREDUCE

*Q: What is **MapReduce**?*

A: Functional programming paradigm designed to simplify appropriately parallelizable large-scale computations.

Originated from another early 2000s Google paper.

Great for certain "embarrassingly parallel problems", not so great for a lot of other types of problems.

- *Let's consider the word frequency problem…*
- *How would you count the words in a list containing, 10^3, 10^6, 10^9 words?*

```
def count_words(words_list):
    counts = {}
    for word in words_list:
        counts[word] += 1
```

- *What's the downside to this approach?*
  - *We have to hold this whole dictionary in memory.*
- *How can we solve this if the data is too big for that?*
  - *Use a cluster of many machines, send each a chunk of the data, then aggregate the results*
- *What issues might we still face?*
  - *Network bandwidth trying to move all that data around*
- *So how can we solve this last hurdle?*
  - ***Move the code to the data** rather than moving the data!*

- *With HDFS, we have our data effectively distributed across the cluster*
- *As best as possible, when we run a map reduce job, Hadoop tries to ensure that each individual worker node only operates on data that is local to it*

Divide and conquer is a fundamental algorithmic technique for solving a given task, whose steps include:

1) *split task into subtasks*
2) *solve these subtasks independently*
3) *recombine the subtask results into a final result*

Map-reduce leverages the divide and conquer approach by splitting a large dataset into several smaller datasets and performing a computation on each of these in parallel.

The defining characteristic of a problem that is suitable for the divide and conquer approach is that it can be broken down into independent subtasks.

Tasks that can be parallelized in this way include:

- *count, sum, average*
- *grep, sort, inverted index*
- *graph traversals,* ***some*** *ML algorithms*

NOTE

Parallelizing a ML algorithm can be a non-trivial exercise!

As we've discussed, the map-reduce approach involves splitting a problem into subtasks and processing these subtasks in parallel.

This takes place in (approximately) two phases:

1) the **mapper** *phase*
1.5) shuffle/sort
2) the **reducer** *phase*

To implement MapReduce, you have to (at a minimum) write 2 functions:

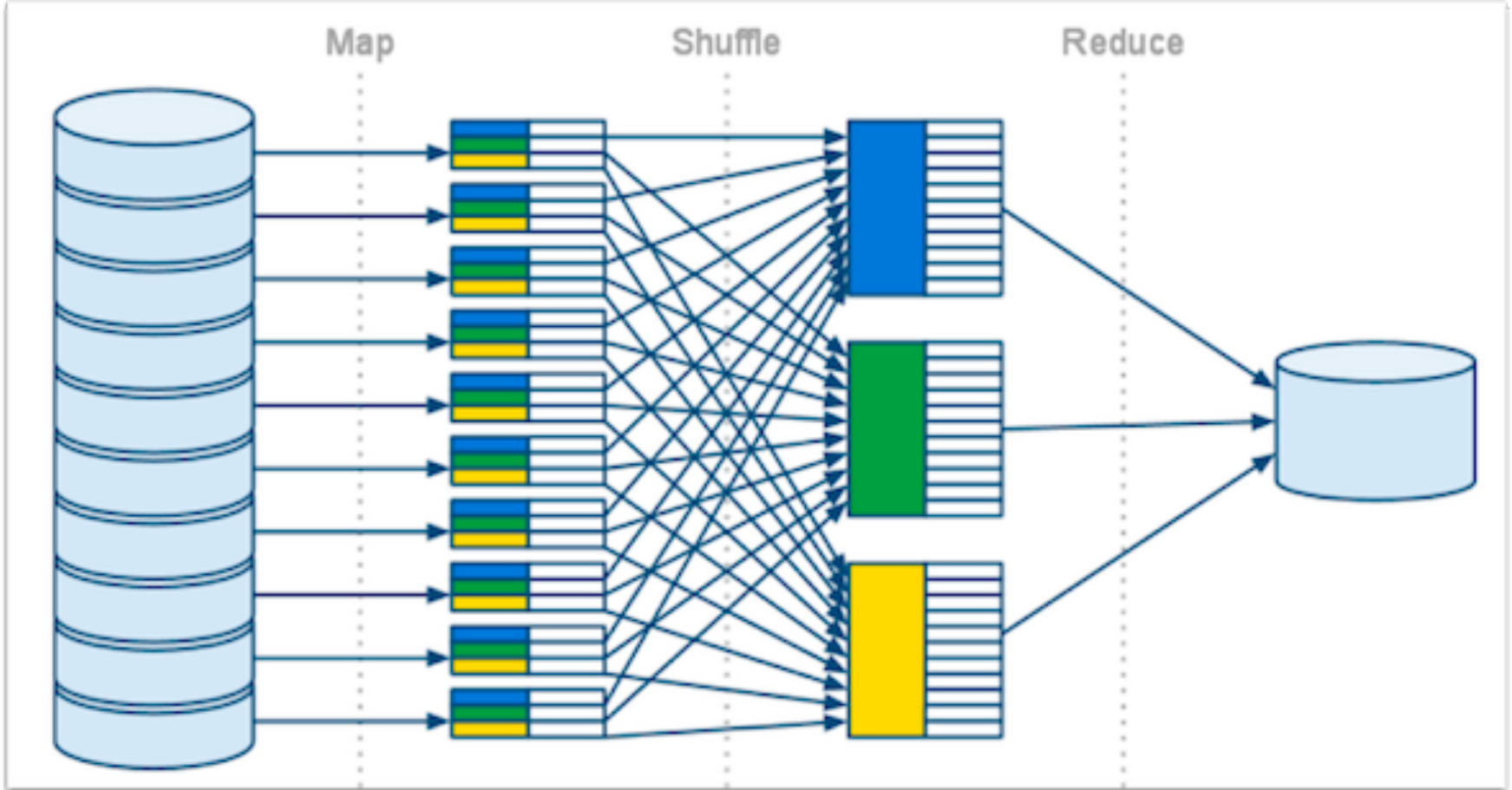
1) the **mapper***: filter and transform data*

- (key, value) ——> (key, value)

2) the **reducer***: outputs a new (key, value) by aggregating pairs with matching keys via some user-defined aggregation function*

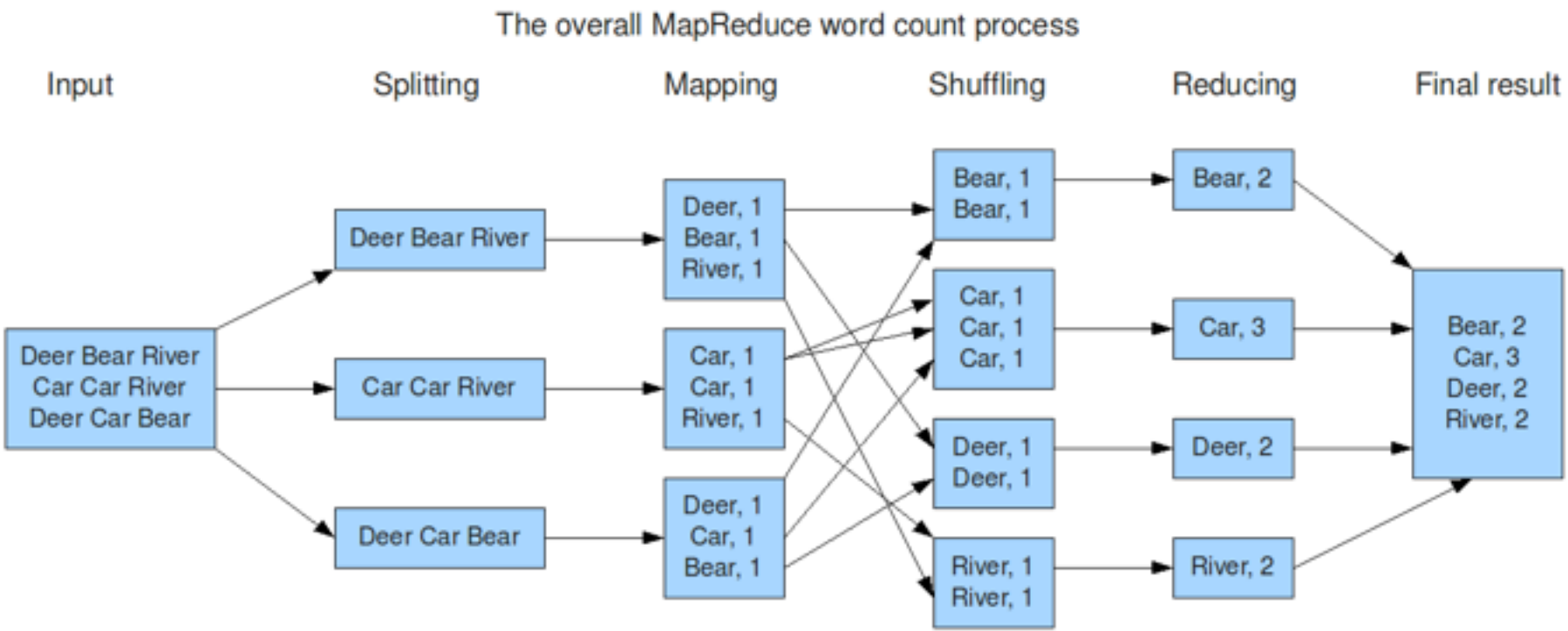
- (key, List<values>) ——> (key, value)

Hadoop will handle everything else for you if you want.


Map
Shuffle
Reduce

As our earlier diagram suggests, there are additional intermediate steps in a map-reduce workflow.

mappers – *filter & transform data*
combiners – *perform reducer operations on the mapper node (optional step, to reduce network traffic and disk I/O).*
partitioners – *shuffle/sort/redirect mapper output*
reducers – *aggregate results*


The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

Map-reduce processes data in terms of key-value pairs:

```
input              <k1, v1>

mapper             <k1, v1> → <k2, v2>

(partitioner)      <k2, v2> → <k2, [all k2 values]>

reducer            <k2, [all k2 values]> → <k3, v3>
```

Using the following input, we can implement the "Hello World" of map-reduce: a word count.

```
where
where in
where in the
where in the world
where in the world is
where in the world is carmen
where in the world is carmen sandiego
```

The first processing primitive is the mapper, which filters & transforms the input data, and emits transformed key-value pairs.

```
mapper(k1, v1):
// k1 = line number
// v1 = line contents (eg, space-delimited string)

    words = tokenize(v1)   // split string into words
    for word in words:
        emit (word, 1)
```

The mapper emits key-value pairs for each word encountered in the input data.

```
where 1
where 1
in    1
where 1
in    1
the   1
...
```

The partitioner is internal to the map-reduce framework, so we don't have to write this ourselves. It shuffles & sorts the mapper output, and redirects all intermediate results for a given key to a single reducer.

```
where       [1, 1, 1, 1, 1, 1, 1]
in          [1, 1, 1, 1, 1, 1]
the         [1, 1, 1, 1, 1]
world       [1, 1, 1, 1]
is          [1, 1 ,1]
carmen      [1, 1]
sandiego    [1]
```

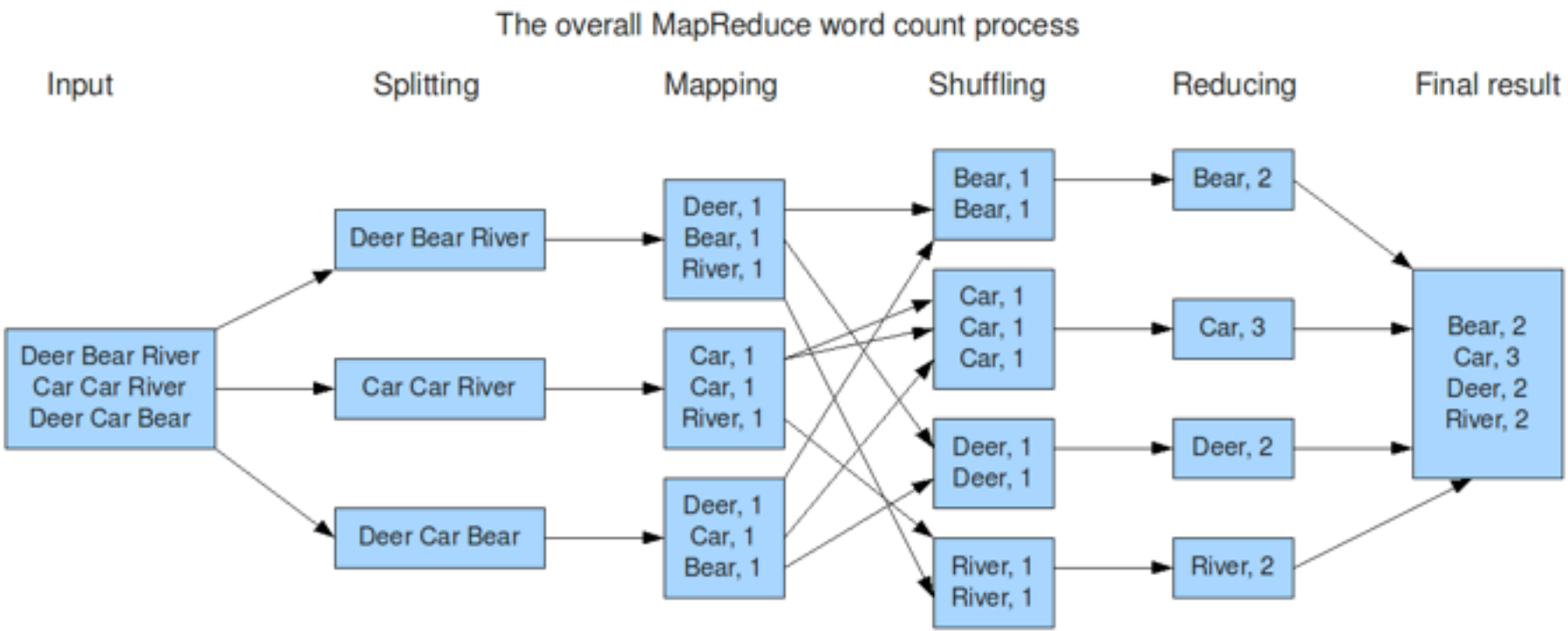
Finally, the reducer receives all values for a given key and aggregates (in this case, sums) the results.

```
reducer(k2, k2_vals):
// k2 = word
// k2_vals = word counts

    emit k2, sum(k2_vals)
```

Reducer output is aggregated & sorted by key.

```
carmen      2
is          3
in          6
the         5
sandiego    1
where       7
world       4
```


The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

The map-reduce framework handles a lot of messy details for you:
- parallelization & distribution (eg, input splitting)
- partitioning (shuffle/sort/redirect)
- fault-tolerance (fact: tasks/nodes will fail!)
- I/O scheduling
- status and monitoring

This (along with the functional semantics) allows you to focus on solving the problem instead of accounting & housekeeping details.

III. THE HADOOP STACK

It's possible to overlay the map-reduce framework with an additional declarative syntax.

This makes operations like select & join easier to implement and less error prone.

*Popular examples include **Pig** and **Hive**.*

Apache Hive

- *SQL language to query data on HDFS*
- *Queries are translated behind the scenes into map-reduce jobs*
- *Data is stored on HDFS, but a metadata database contains the table schemas*

Cloudera Impala

- ***ANOTHER*** *SQL language to query data on HDFS*

- Similar interface to Hive

BUT:

- Impala contains its own scheduling engine, queries are not translated map-reduce jobs

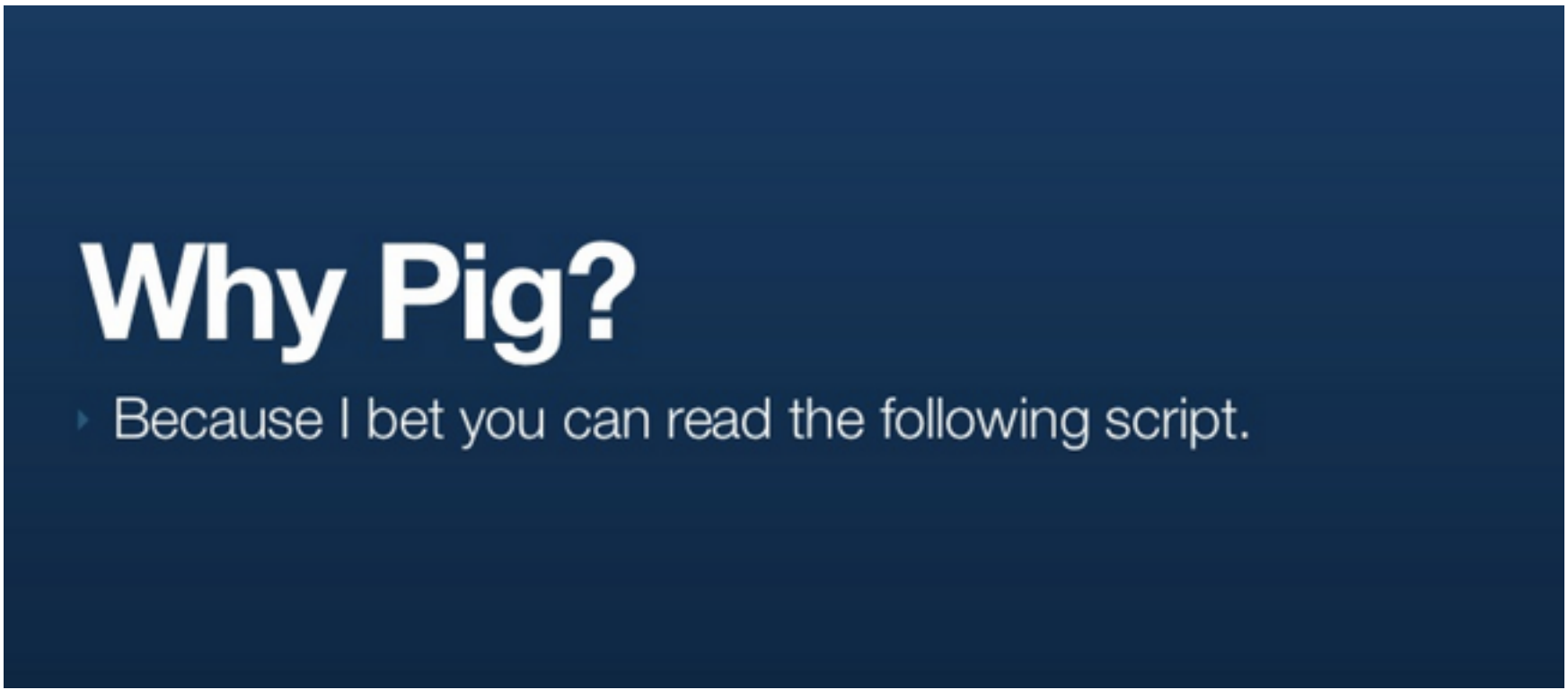
- Leads to faster queries, but no fault tolerance

Apache Pig

- Scripting language to operate on data on HDFS

- Queries are translated behind the scenes into map-reduce jobs

- Allows for similar declarative functionality to hive, but often in a simpler and more intuitive scripting format



source: http://www.slideshare.net/kevinweil/hadoop-pig-and-twitter-nosql-east-2009

A Real Pig Script

top_5.pig

```
users = load 'users.csv' as (username: chararray, age: int);
users_1825 = filter users by age >= 18 and age <= 25;

pages = load 'pages.csv' as (username: chararray, url: chararray);

joined = join users_1825 by username, pages by username;
grouped = group joined by url;
summed = foreach grouped generate group as url, COUNT(joined) AS views;
sorted = order summed by views desc;
top_5 = limit sorted 5;

store top_5 into 'top_5_sites.csv';
```

- Now, just for fun... the same calculation in vanilla Hadoop MapReduce.


No, seriously.

Apache Sqoop

- *Utility for moving data between relational databases and Distributed file systems (import/export)*
- *Automatically generates efficient map-reduce jobs for your bulk move*

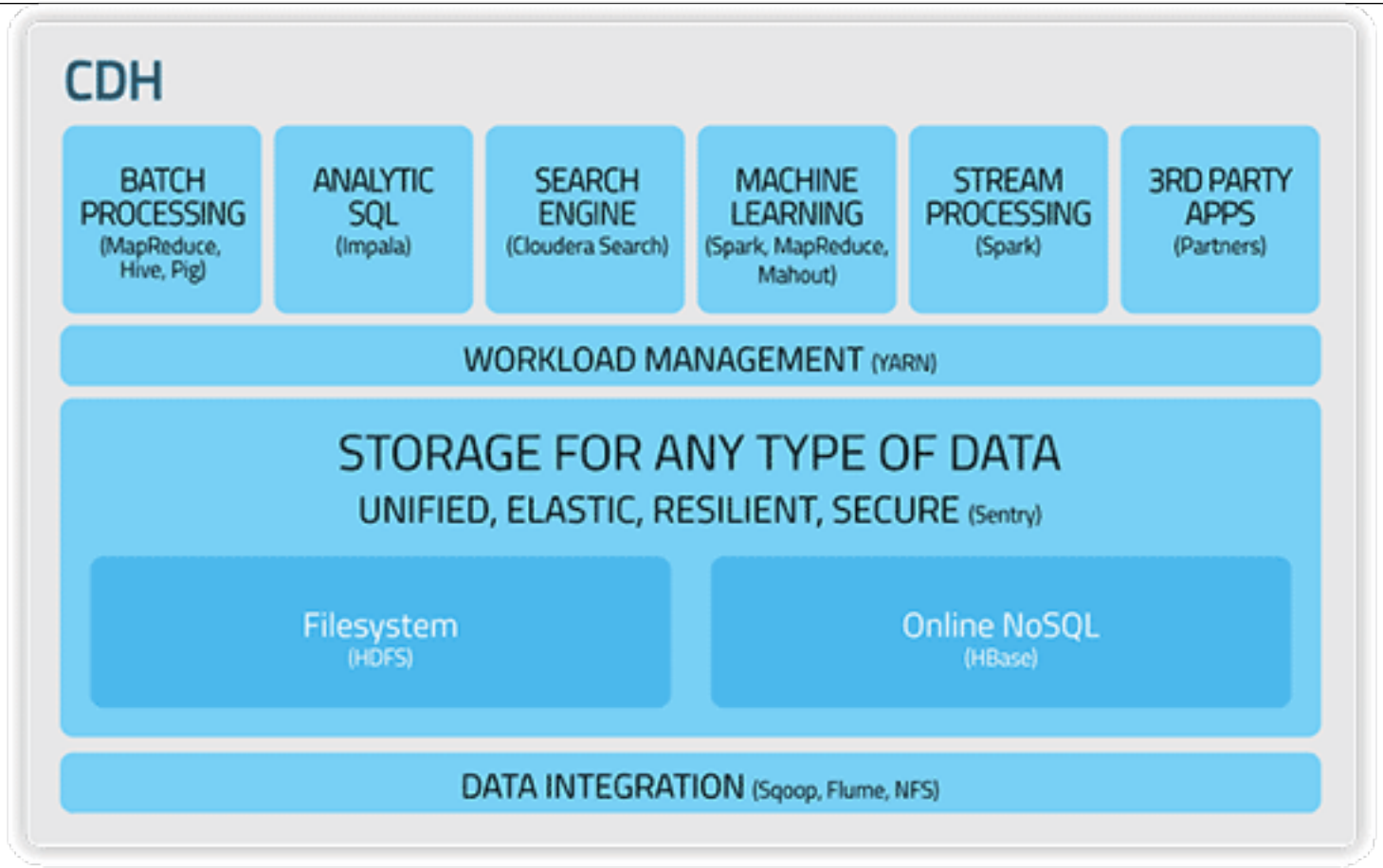
Apache Oozie

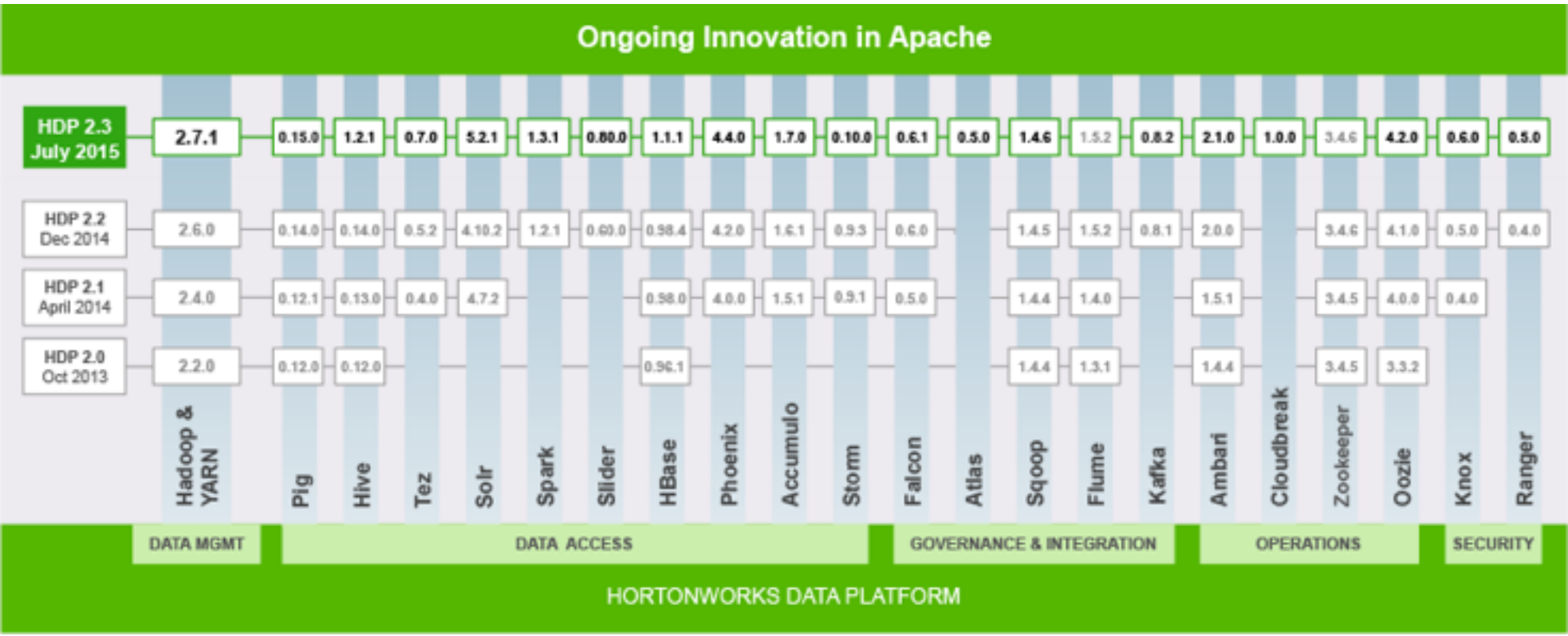
- *Job-chaining utility*
- *Allows you to plan directed graphs of sequential jobs with conditional path execution*

Apache Zookeeper

- Central coordination service used by many of the services in the Hadoop ecosystem to stay in lockstep

HADOOP PROVIDERS


CDH
BATCH PROCESSING (MapReduce, Hive, Pig)
ANALYTIC SQL (Impala)
SEARCH ENGINE (Cloudera Search)
MACHINE LEARNING (Spark, MapReduce, Mahout)
STREAM PROCESSING (Spark)
3RD PARTY APPS (Partners)
WORKLOAD MANAGEMENT (YARN)
STORAGE FOR ANY TYPE OF DATA
UNIFIED, ELASTIC, RESILIENT, SECURE (Sentry)
Filesystem (HDFS)
Online NoSQL (HBase)
DATA INTEGRATION (Sqoop, Flume, NFS)


Ongoing Innovation in Apache
HDP 2.3 July 2015
2.7.1
0.15.0
1.2.1
0.7.0
5.2.1
1.3.1
0.80.0
1.1.1
4.4.0
1.7.0
0.10.0
0.6.1
0.5.0
1.4.6
1.5.2
0.8.2
2.1.0
1.0.0
3.4.6
4.2.0
0.6.0
0.5.0
HDP 2.2 Dec 2014
2.6.0
0.14.0
0.14.0
0.5.2
4.10.2
1.2.1
0.60.0
0.98.4
4.2.0
1.6.1
0.9.3
0.6.0
1.4.5
1.5.2
0.8.1
2.0.0
3.4.6
4.1.0
0.5.0
0.4.0
HDP 2.1 April 2014
2.4.0
0.12.1
0.13.0
0.4.0
4.7.2
0.98.0
4.0.0
1.5.1
0.9.1
0.5.0
1.4.4
1.4.0
1.5.1
3.4.5
4.0.0
0.4.0
HDP 2.0 Oct 2013
2.2.0
0.12.0
0.12.0
0.96.1
1.4.4
1.3.1
1.4.4
3.4.5
3.3.2
Hadoop & YARN
Pig
Hive
Tez
Solr
Spark
Slider
HBase
Phoenix
Accumulo
Storm
Falcon
Atlas
Sqoop
Flume
Kafka
Ambari
Cloudbreak
Zookeeper
Oozie
Knox
Ranger
DATA MGMT
DATA ACCESS
GOVERNANCE & INTEGRATION
OPERATIONS
SECURITY
HORTONWORKS DATA PLATFORM

IV. SPARK

- *With distributed systems like Hadoop, we have tools for collecting, storing, and processing information at scale.*
- *Having access to all this data is one thing, being able to do something useful with it is something entirely different.*
- *Doing something useful means placing a schema over the data and using, say, SQL, to answer questions like: "of the millions of users who made it to the 3rd page of our registration process, how many were over 25?"*

- *In general, "doing something useful" means more than just SQL*
- *We want richer functionality in areas like ML and statistics*

- *The sorts of things we have been doing with data in our IPython notebooks are the sorts of things we would like to do to leverage the huge datasets residing on our HDFS, which is spread over our cluster*
- *One idea might be to push our ML framework onto the machines, having rewritten them to work in a distributed setting.*
- *The problem is that many algorithms have wide data dependencies, and network transfer rates are considerably slower than in-memory access.*

- *Thus there is a need for a programming paradigm that is by its very nature highly parallel*
- *Genomics has had **HPC**, or high-performance computing for decades*
- *The problems with HPC are in its difficulty in use*
- *Hadoop gives the mainstream user many of the features of HPC at a far cheaper cost...*
- *So how do we do data science on massive data?*

1. *Most of our work is in preprocessing the data*
   - *You will have to preprocess data that you can't inspect directly*
   - *Preprocessing becomes even more computationally dependent*
2. *Iteration is fundamental to data science*
   - *There is a requirement for multiple passes over the data*
   - *Experimentation intrinsic (feature/model selection, optimization, etc)*
3. *The task isn't over when a good model has been built*
   - *Models must be productionized and maintained*
   - *Ideally, we need a **REPL**, (Read-Evaluate-Print-Loop) environment*

- *Spark aims to provide the solution to this*

*Q: What is **Spark**?*

A: An open source framework that combines an engine to distribute programs across clusters of machines with an elegant model for writing programs on top of it.

Arguably the first open source software that makes distributed programming truly accessible to the data science workflow!

- ***Spark Core:*** *contains the basic functionality of Spark*
  - *APIs that define Resilient Distributed Datasets (RDDs) and operations and actions that can be performed on them*
  - *The rest of Spark's libraries are built on top of the RDD and Spark Core*

- ***Spark SQL:***
  - *APIs for interacting with data in Spark via the Apache Hive variant of SQL* ***HiveQL*** *(Hive Query Language)*
  - *Every DB table is represented as an RDD and Spark SQL queries are translated to Spark Operations*
  - *Can function as drop-in replacement for Hive*

- ***Spark Streaming:***
  - *Enables the processing and manipulation of live data streams in real time*
  - *Many streaming data libraries (e.g. Apache Storm) exist for handling streaming data in real-time*
  - *Spark Streaming enables programs to leverage this data similar to how you would interact with a normal RDD as data is flowing in*

- ***Spark MLLib:***
  - *A library of common ML algorithms implemented as Spark operations on RDDs*
  - *Contains scalable algorithms like classifications, regressions, etc*
  - ***Mahout**, the former choice for this task, will move to Spark for future things*

- *Spark maintains MapReduce's linear scalability and fault tolerance, but extends it in 3 ways:*
  - *Does not rely on a rigid map-then-reduce paradigm. Its engine can implement a Directed Acyclic Graph (DAG) of operators.*
  - *It complements this with a rich set of transformations that allow users to express computations more naturally (developer focused)*
  - *It has in-memory processing; it has RDD abstraction, meaning future steps requiring the same data do NOT need to be recomputed or reloaded from disk*

- *Highly suited for iterative algorithms requiring multiple data passes*
- *Aim is to bring data science into a single programming environment*
- *It aims to be the **Python of big data**!*
- *Native language is **Scala**, but has Python API **PySpark***
- *With Scala, you can perform the entire data science pipeline. You don't have to query to get the data and then do your analysis in R or Python, it's all unified and the details invisible to the user.*

- *Read/write data in all formats supported by MR (Avro, Parquet, CSV)*
- *Read/write from/to NoSQL DBs like HBase or Cassandra*
- *Spark streaming can ingest data continuously from streaming sources like Flume*
- *SparkSQL can use the underlying Hive engine for data interaction*
- *Can run inside YARN, Hadoop's scheduler and resource manager*

- *STILL A WORK IN PROGRESS*
- *Hasn't surpassed MR as the workhorse of batch processing*
- *SQL, MLLib, Streaming, GraphX all still under development*
- *Nowhere near the full-featured ML of R, Python, etc*
- *SQL is rich, but still lags behind Hive*

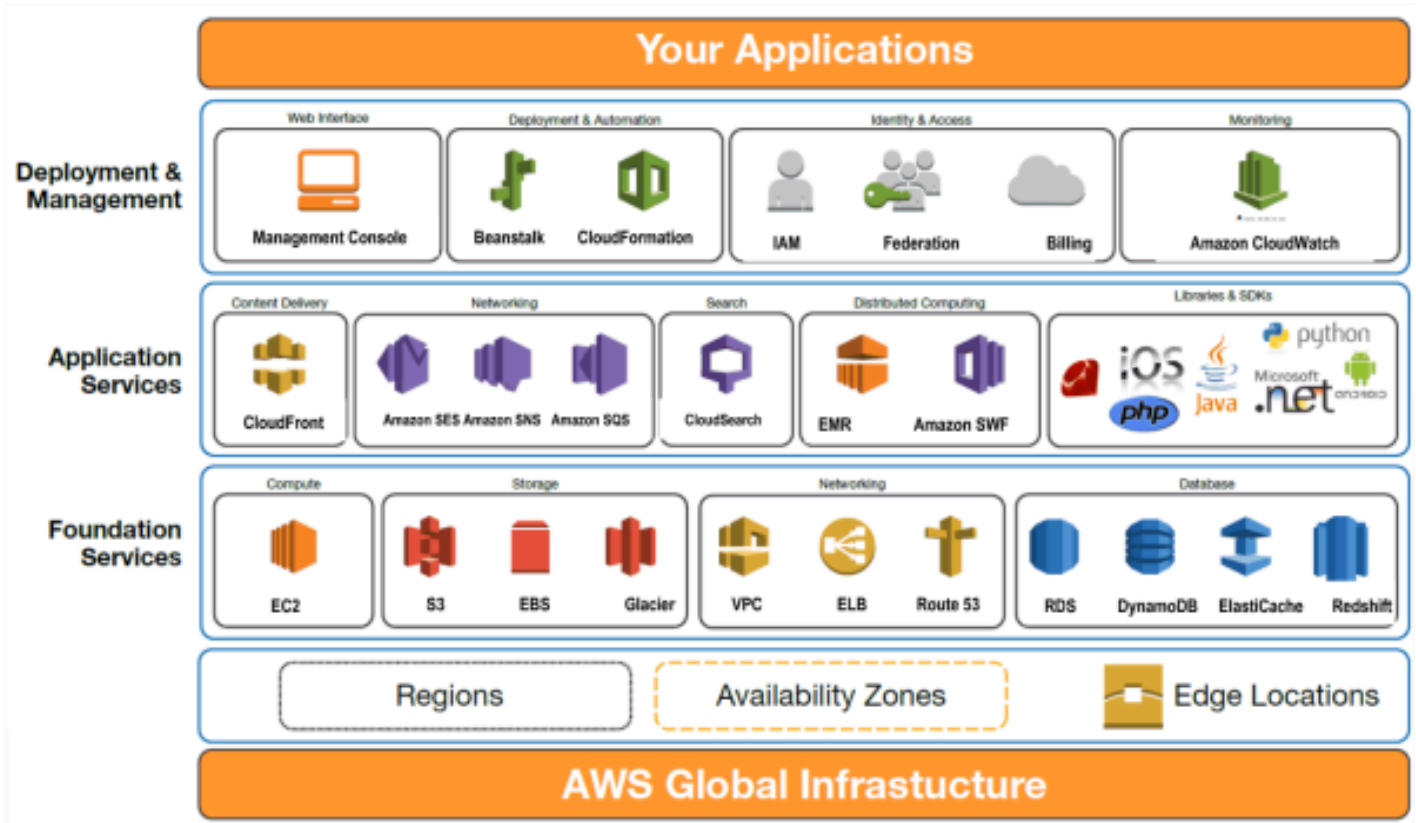
- *Writing a Spark program usually consists of:*
  - *Defining a set of transformations on input datasets*
  - *Invoking actions that output the transformed datasets into persistent storage or return results to the driver's local memory*
  - *Running local computations that operate on the results computed in a distributed fashion. These can help you decide what actions and transformations to undertake next*

HANDS-ON: PYSPARK

V. AMAZON WEB SERVICES

Amazon provides a suite of services to allow you to easily manage all of your servers (hardware, software, OS) completely in the Amazon cloud.

This is called Amazon Web Services (AWS)


Your Applications
Deployment & Management
Web Interface
Management Console
Deployment & Automation
Beanstalk
CloudFormation
Identity & Access
IAM
Federation
Billing
Monitoring
Amazon CloudWatch
Application Services
Content Delivery
CloudFront
Networking
Amazon SES
Amazon SNS
Amazon SQS
Search
CloudSearch
Distributed Computing
EMR
Amazon SWF
Libraries & SDKs
ios
python
Java
Microsoft
.net
php
android
Foundation Services
Compute
EC2
Storage
S3
EBS
Glacier
Networking
VPC
ELB
Route 53
Database
RDS
DynamoDB
ElastiCache
Redshift
Regions
Availability Zones
Edge Locations
AWS Global Infrastucture

Compute

- ec2: resizable compute capacity in the cloud

- lambda: run code immediately in response to events like website clicks, service requests

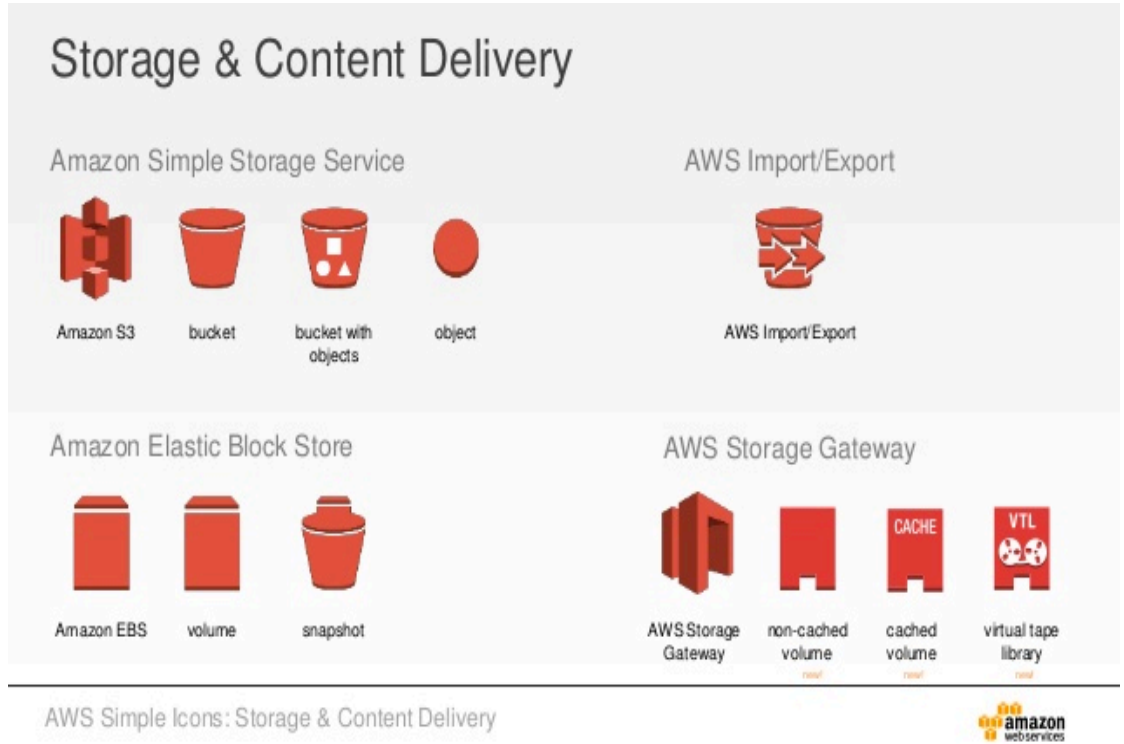
Storage

- s3: secure, durable, scalable object storage

- Storage Gateway: connect on-premises hardware with cloud

- Glacier: long term archival storage

- Cloudfront: deliver content to users



Databases

- RDS: run relational DBs

- DynamoDB: NoSQL DB in the cloud

- ElastiCache: In-memory key-value cache

- Redshift: Data Warehousing in the cloud

Amazon RDS

DynamoDB

Amazon ElastiCache

Networking

- VPC: provision segment of cloud for private network

- Direct Connect: private connection directly to AWS cloud

- Route 53: DNS Server


Amazon VPC


AWS Direct Connect


Amazon Route 53

Admin/Security

- *Directory Service: use Microsoft ActiveDirectory*
- *IAM: Control access to resources/services*
- *Trusted Advisor: advise on best practices*
- *CloudTrail: log AWS API calls*
- *Config: Configuration management*
- *CloudWatch: monitor resources*

Deployment/Management

- *Elastic Beanstalk: Web application deployment*
- *OpsWorks: Custom application deployment*
- *CloudFormation: Provision AWS resources*
- *CodeDeploy: Automated code deployment*

Amazon Elastic Beanstalk

AWS CloudFormation

Analytics

- *EMR: Managed Hadoop, Spark framework*
- *Kinesis: Data stream processing*
- *DataPipeline: Data processing/transfer at regular intervals*