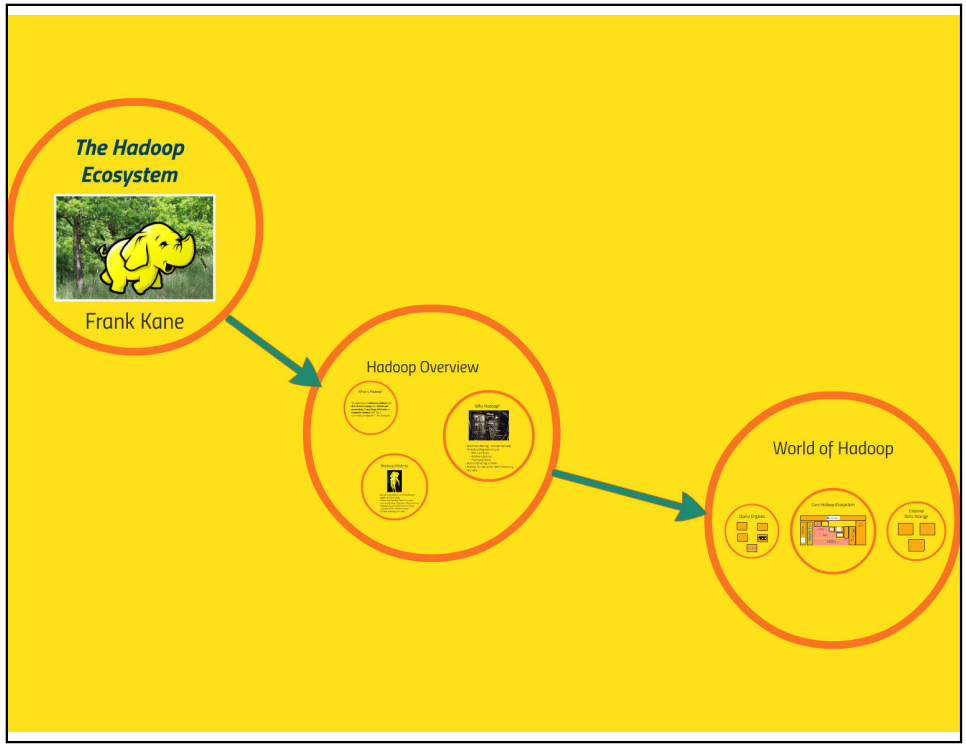
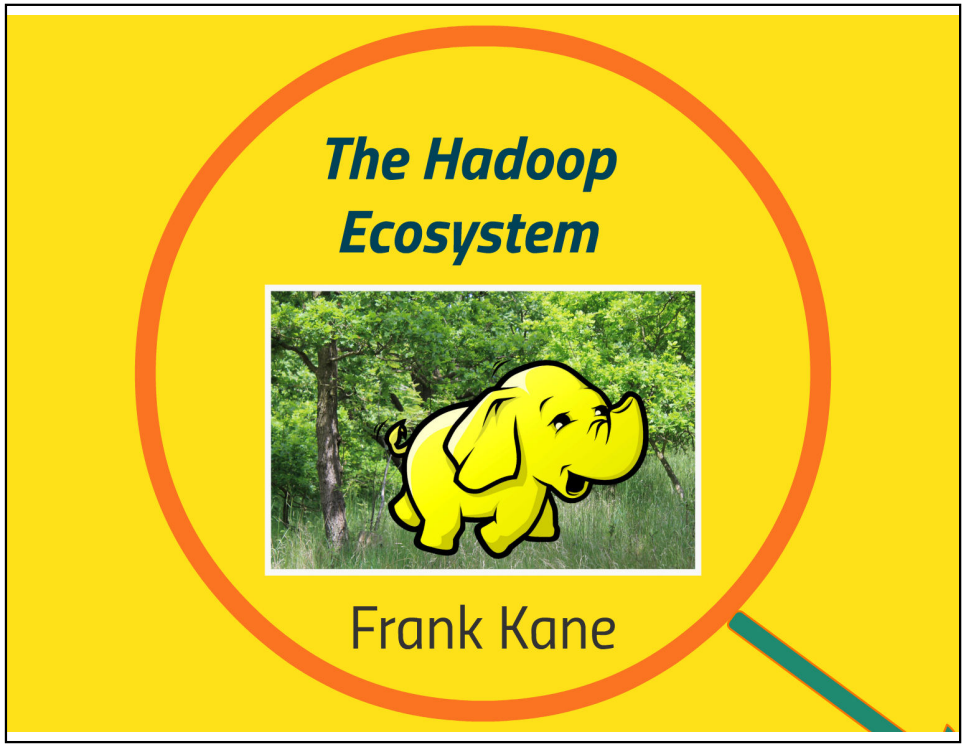




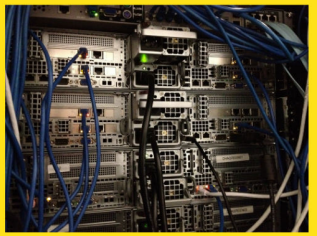
What is Hadoop?

"an open source **software platform** for **distributed storage** and **distributed processing** of **very large data sets** on **computer clusters** built from commodity hardware" - *Hortonworks*

Hadoop History

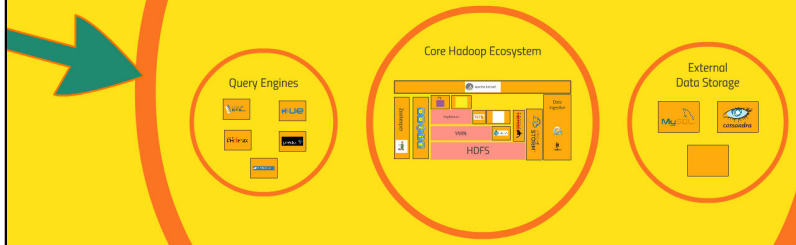
- Google published GFS and MapReduce papers in 2003-2004
- Yahoo! was building "Nutch," an open source web search engine at the same time
- Hadoop was primarily driven by Doug Cutting and Tom White in 2006
- It's been evolving ever since...

Why Hadoop?

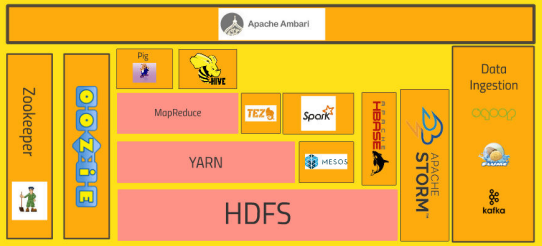


- Data's too darn big - terabytes per day
- Vertical scaling doesn't cut it
  - Disk seek times
  - Hardware failures
  - Processing times
- Horizontal scaling is linear
- Hadoop: It's not just for batch processing anymore

World of Hadoop



Core Hadoop Ecosystem



MapReduce

TEZ

Spark

HBASE

YARN

MESOS

HDFS

MapReduce

TEZ

YARN

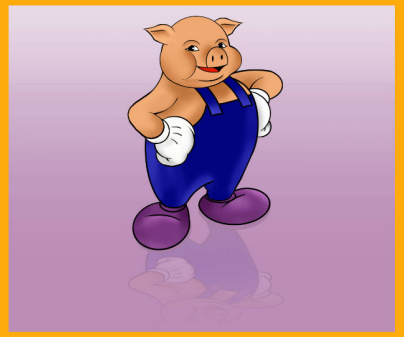
HDFS

HIVE

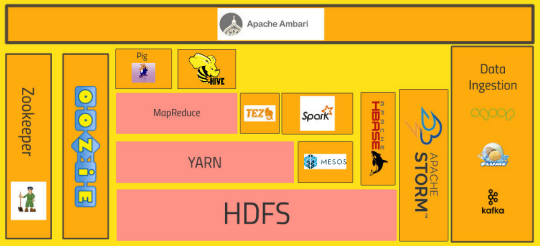
MapReduce

YARN

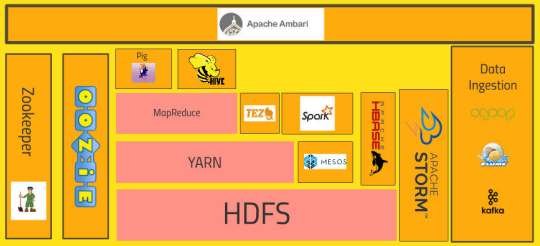
Pig

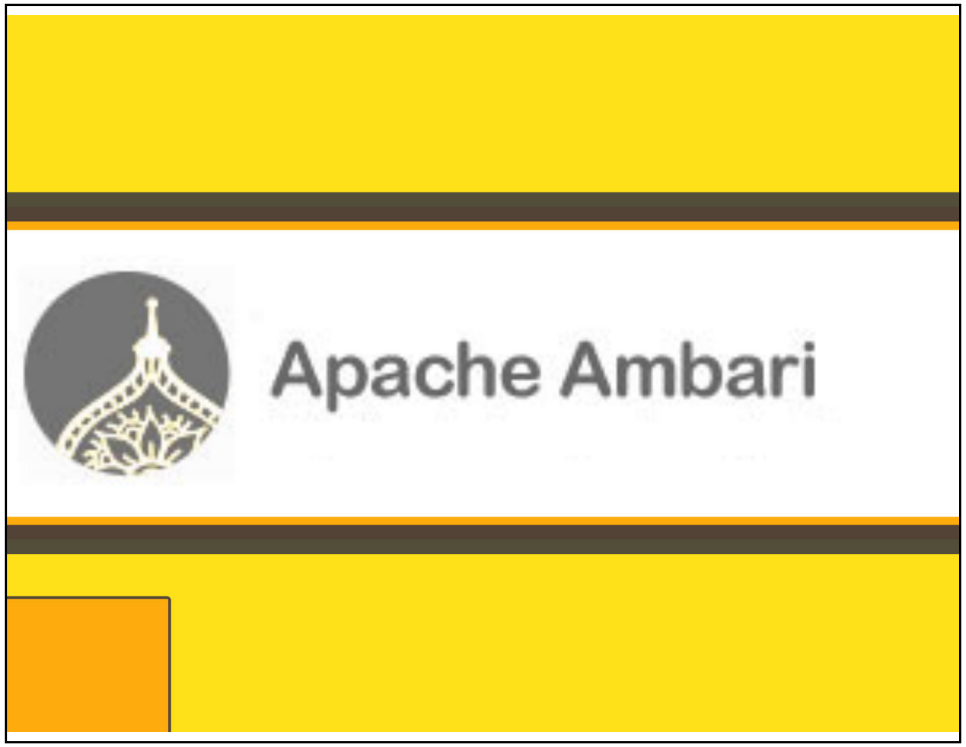


Core Hadoop Ecosystem

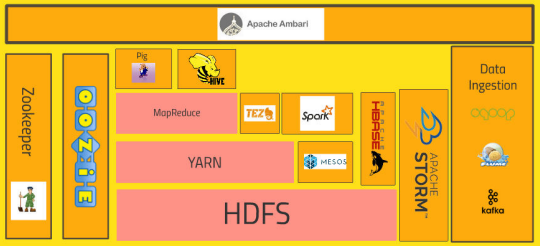


Core Hadoop Ecosystem



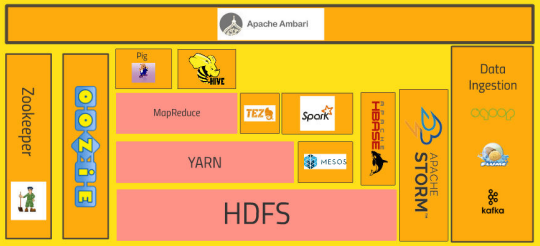


Core Hadoop Ecosystem



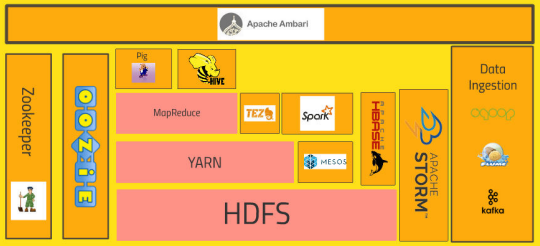
MESOS

Core Hadoop Ecosystem



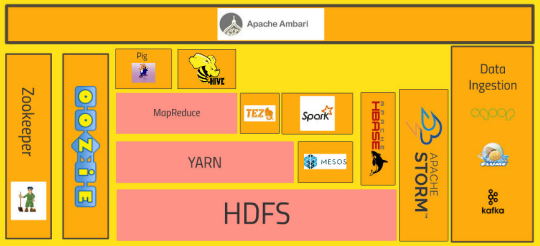
Spark

Core Hadoop Ecosystem

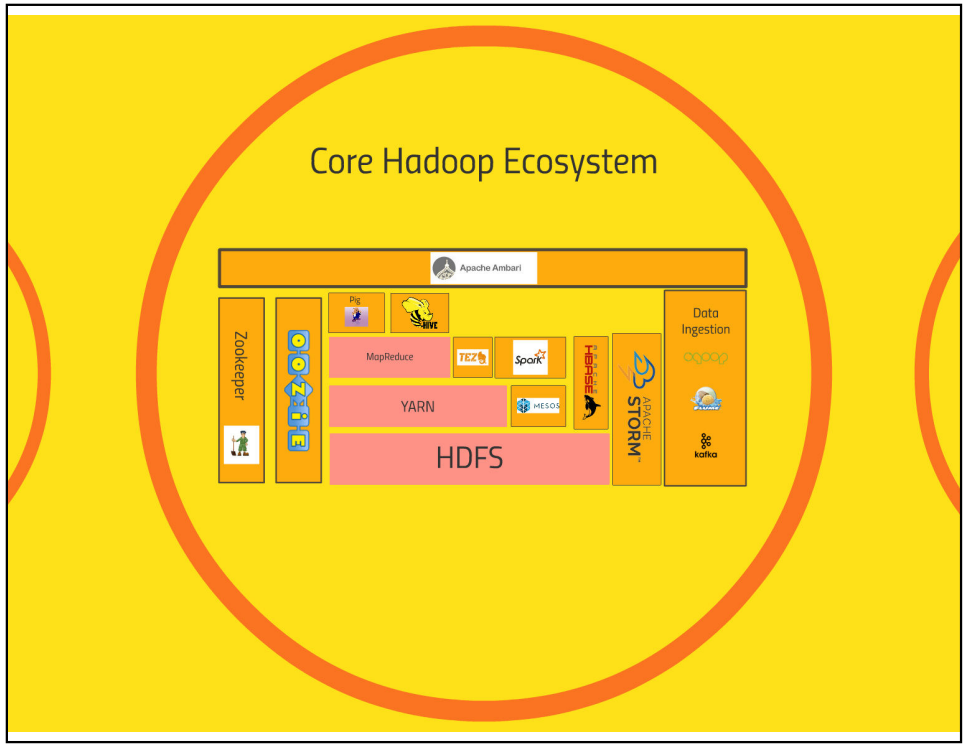


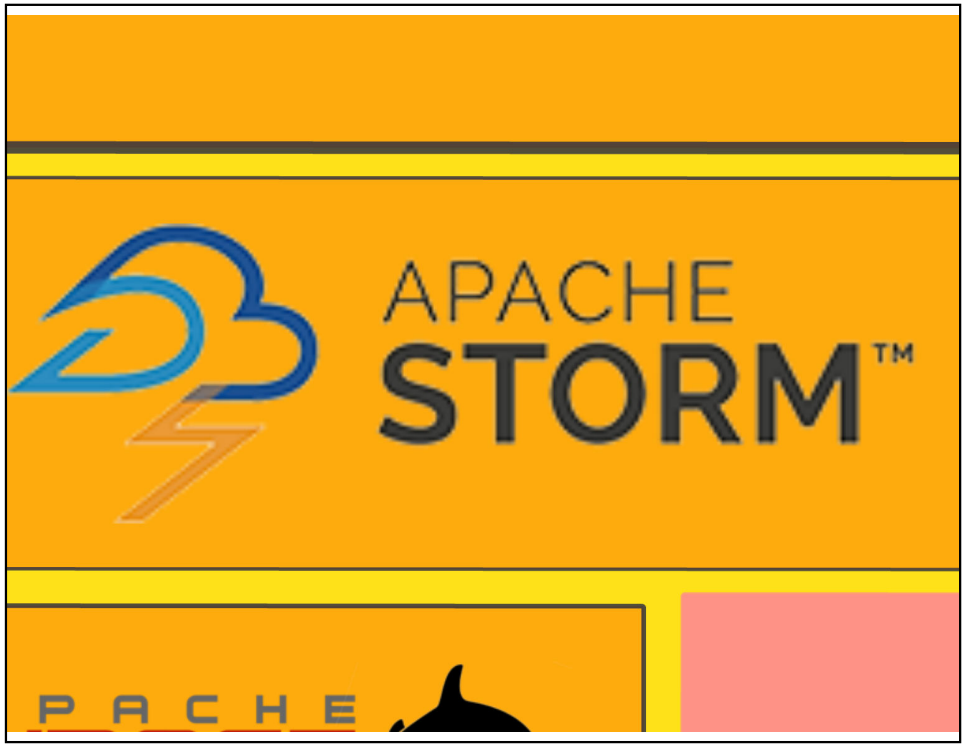
TEZ

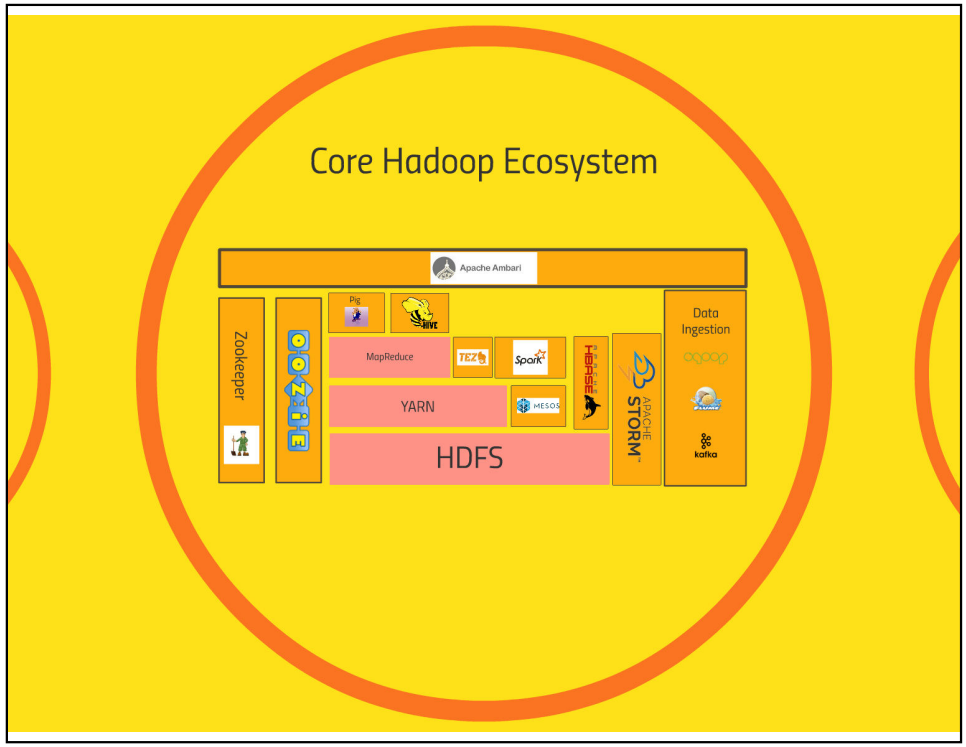
Core Hadoop Ecosystem

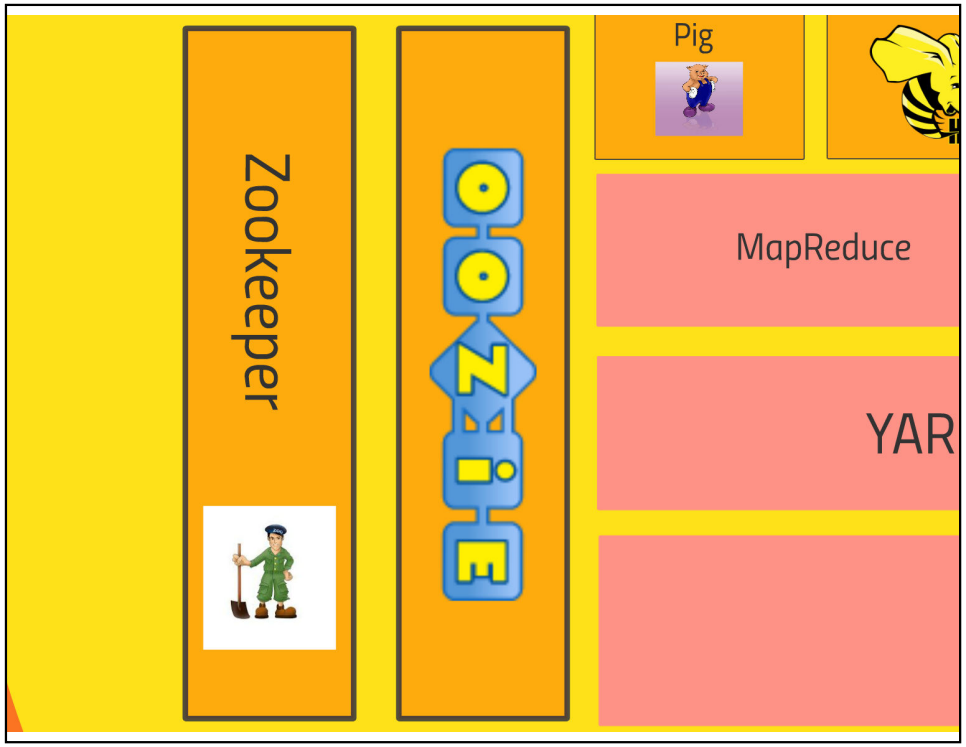


APACHE HBASE


Core Hadoop Ecosystem
Apache Ambari
Zookeeper
OOZIE
Pig
HIVE
MapReduce
TEZ
Spark
HBase
YARN
MESOS
HDFS
APACHE STORM™
Data Ingestion
kafka


APACHE
STORM™


Core Hadoop Ecosystem
Apache Ambari
Zookeeper
OOZIE
Pig
HIVE
MapReduce
TEZ
Spark
HBase
YARN
MESOS
HDFS
APACHE STORM™
Data Ingestion
kafka


Zookeeper
OOZIE
Pig
MapReduce
YAR

Zookeeper

OOZIE

Pig

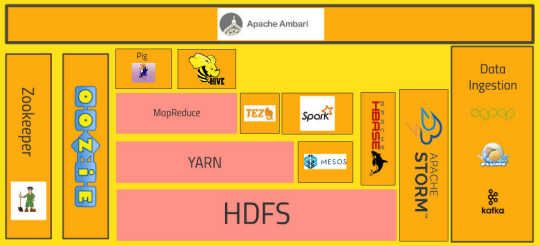
Data Ingestion

APACHE HBASE

APACHE STORM™

kafka

Core Hadoop Ecosystem

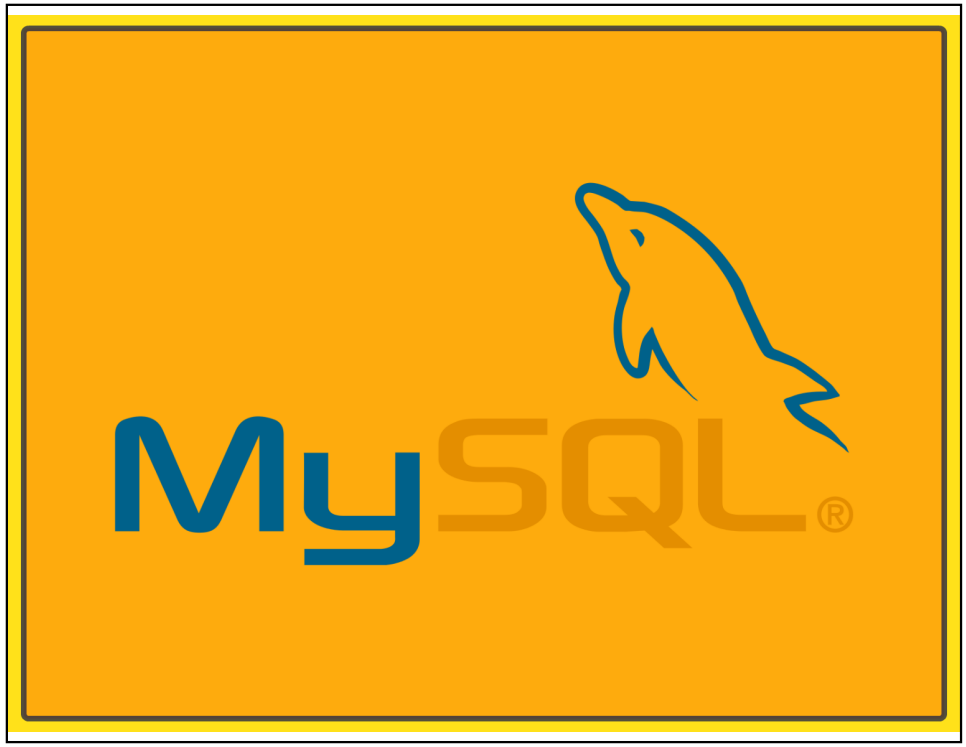


External Data Storage

MySQL

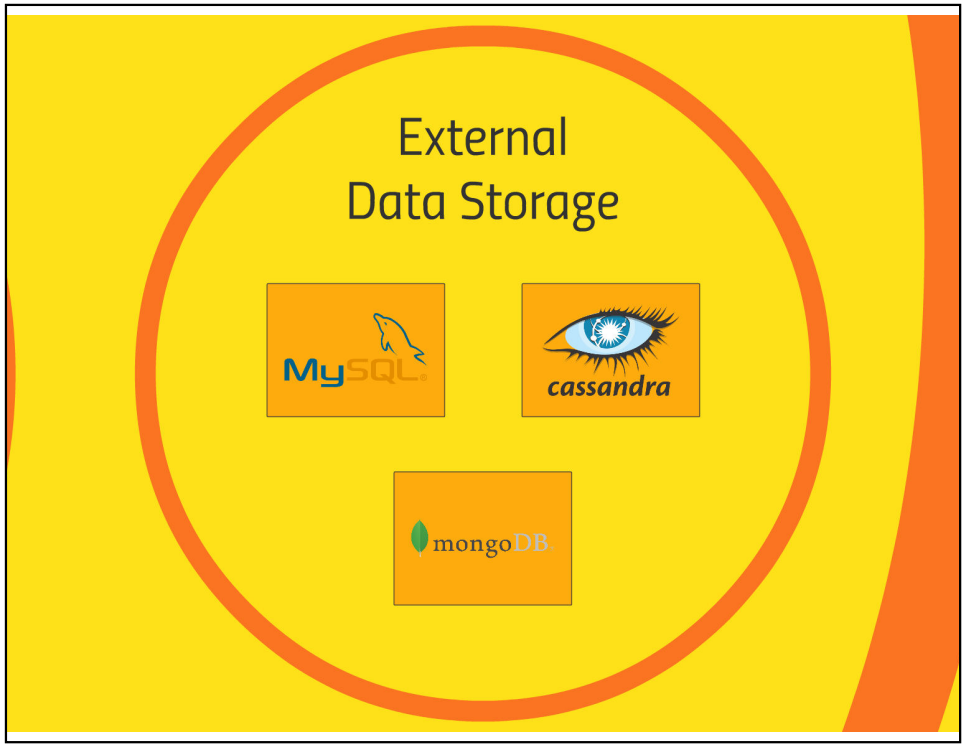
cassandra

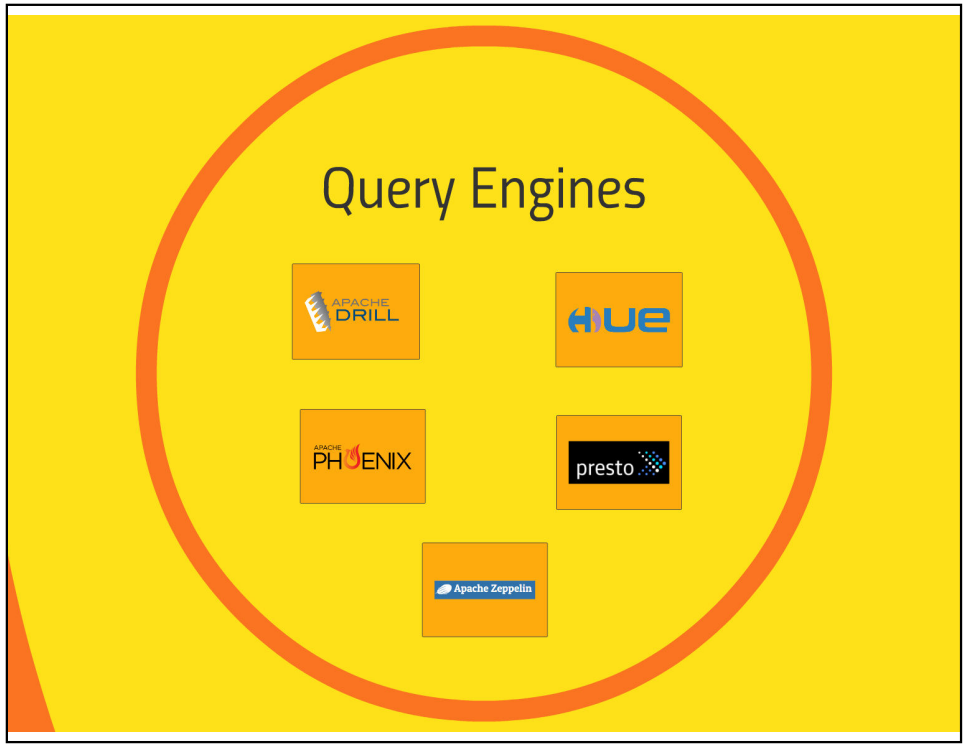
mongoDB


MySQL®


cassandra


mongoDB®


External
Data Storage
MySQL®
cassandra
mongoDB®


Query Engines
APACHE DRILL
HUE
APACHE PHOENIX
presto
Apache Zeppelin

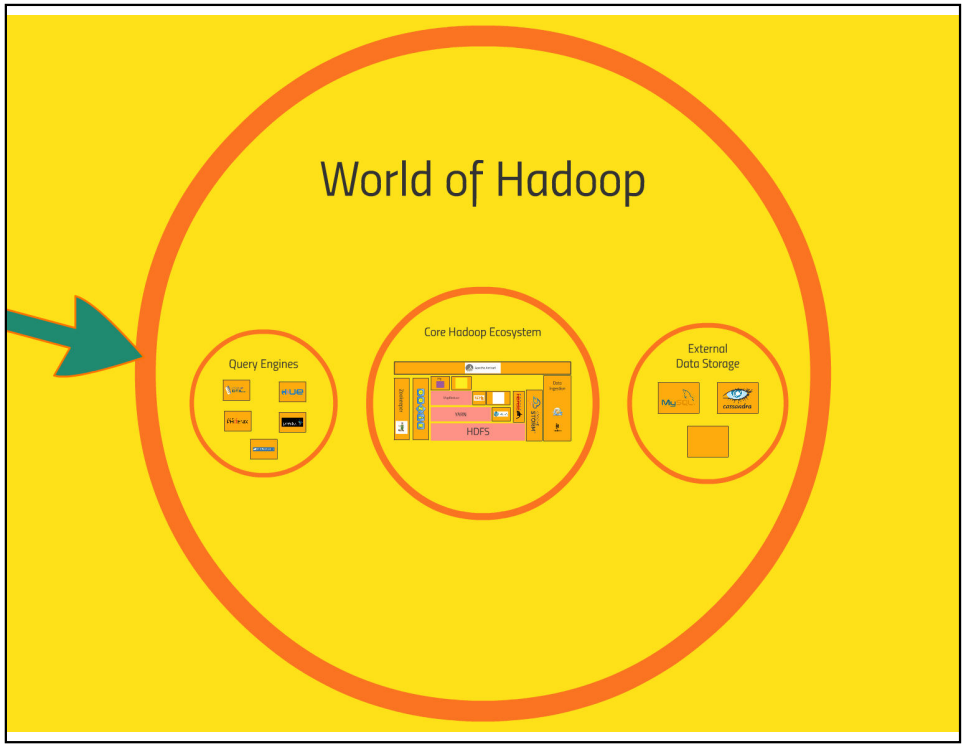

APACHE
DRILL

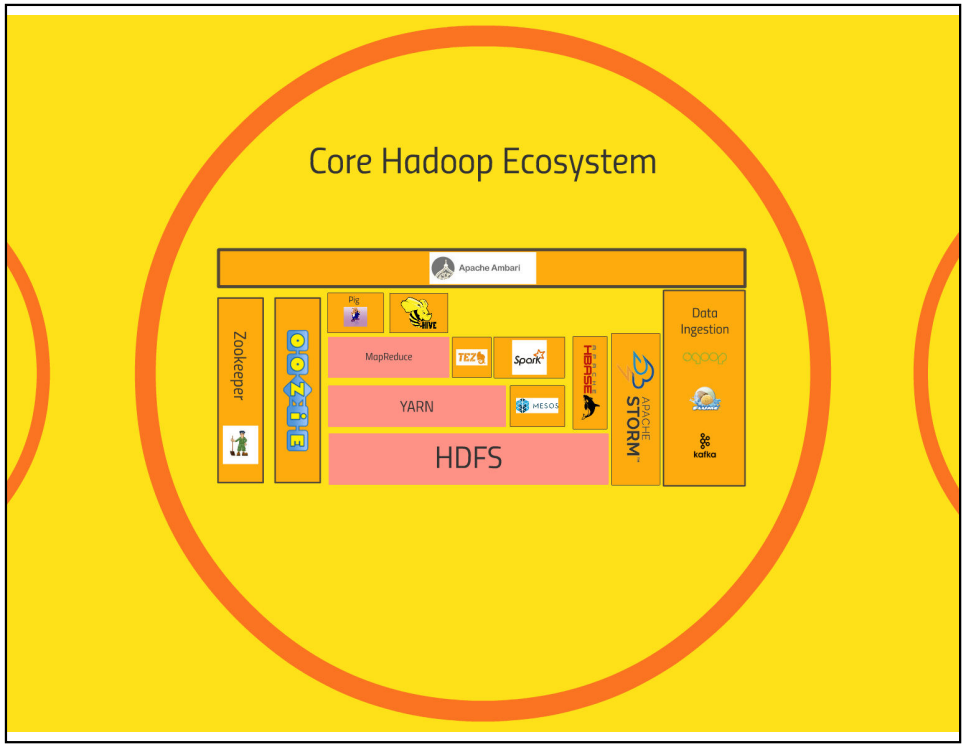

HUE


APACHE
PHOENIX

presto

Apache Zeppelin





External Data Storage

MySQL

cassandra

mongoDB

Query Engines

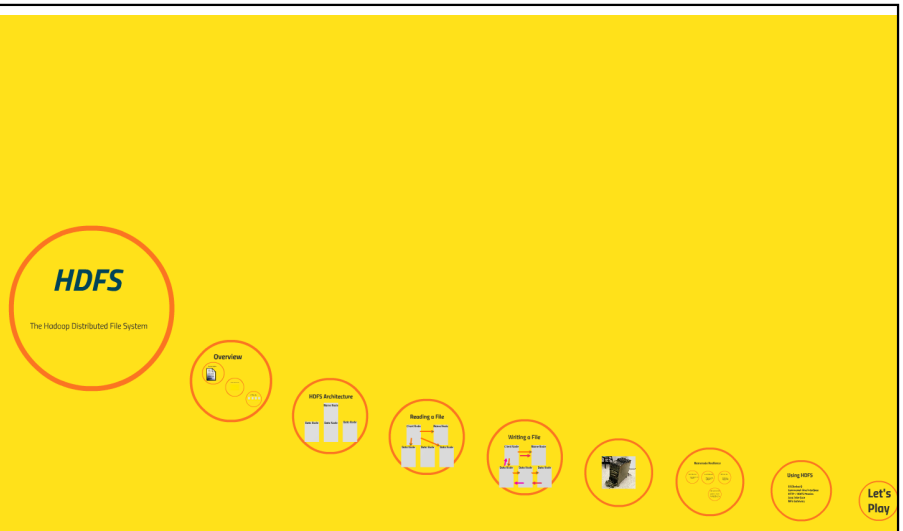
APACHE DRILL

HUE

APACHE PHOENIX

presto

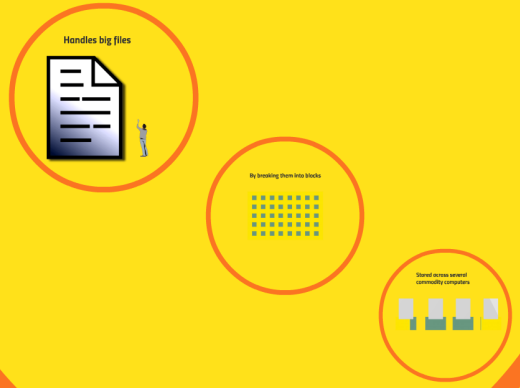
Apache Zeppelin



HDFS

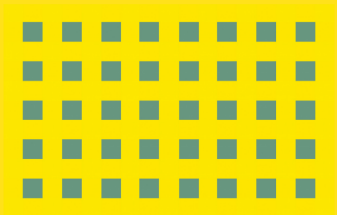
The Hadoop Distributed File System

Overview

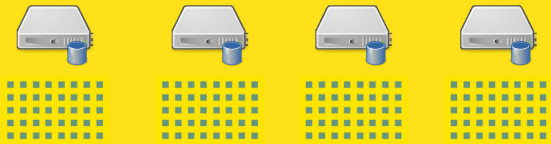


Handles big files

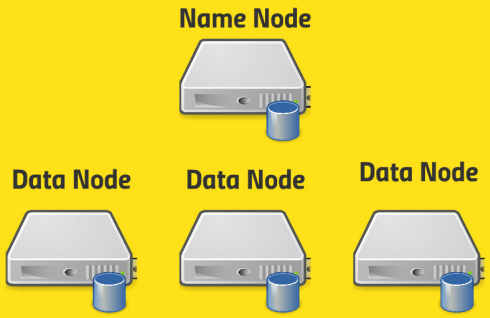
By breaking them into blocks



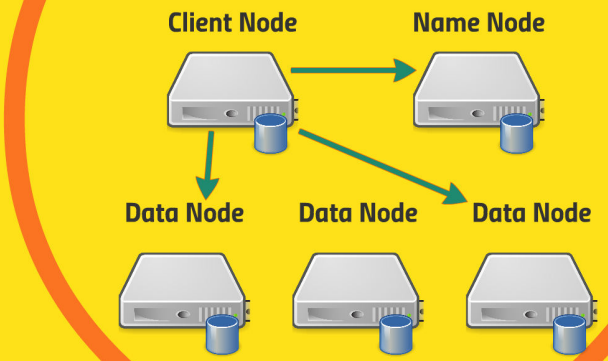
Stored across several commodity computers



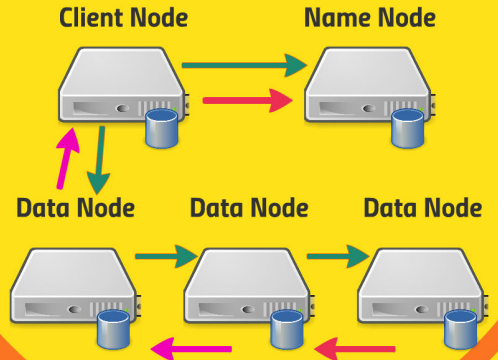
HDFS Architecture



Reading a File



Writing a File



Namenode Resilience

Back Up Metadata

Namenode writes to local disk and NFS

Secondary Namenode

Maintains merged copy of edit log you can restore from

HDFS Federation

Each namenode manages a specific namespace volume

HDFS High Availability

- Hot standby namenode using shared edit log
- Zookeeper tracks active namenode
- Uses extreme measures to ensure only one namenode is used at a time

Using HDFS

UI (Ambari)
Command-Line Interface
HTTP / HDFS Proxies
Java interface
NFS Gateway

Let's Play

MAPREDUCE FUNDAMENTAL CONCEPTS

Why MapReduce?

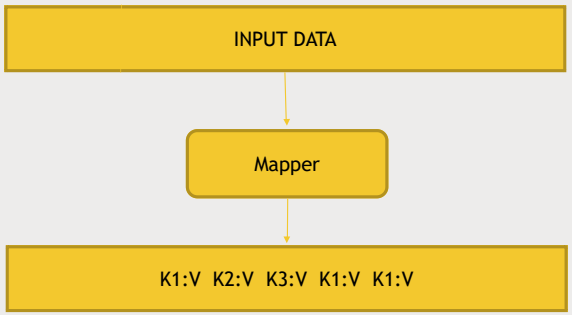
- Distributes the processing of data on your cluster
- Divides your data up into partitions that are MAPPED (transformed) and REDUCED (aggregated) by mapper and reducer functions you define
- Resilient to failure – an application master monitors your mappers and reducers on each partition

Let's illustrate with an example

- How many movies did each user rate in the MovieLens data set?

How MapReduce Works: Mapping

- The MAPPER converts raw source data into **key/value** pairs



Example: MovieLens Data (u.data file)

USER ID	MOVIE ID	RATING	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

Map users to movies they watched

USER ID | MOVIE ID | RATING | TIMESTAMP

196 242 3 881250949
186 302 3 891717742
196 377 1 878887116
244 51 2 880606923
166 346 1 886397596
186 474 4 884182806
186 265 2 881171488

196:242 186:302 196:377 244:51 166:346 186:274 186:265

Extract and Organize What We Care About

196:242 186:302 196:377 244:51 166:346 186:274 186:265

MapReduce Sorts and Groups the Mapped Data ("Shuffle and Sort")

196:242 186:302 196:377 244:51 166:346 186:274 186:265

166:346 186:302,274,265 196:242,377 244:51

The REDUCER Processes Each Key's Values

166:346 186:302,274,265 196:242,377 244:51

len(movies)

166:1 186:3 196:2 244:1

Putting it All Together

USER ID	MOVIE ID	RATING	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

↓

MAPPER

↓

196:242 186:302 196:377 244:51 166:346 186:274 186:265

↓

SHUFFLE AND SORT

↓

166:346 186:302,274,265 196:242,377 244:51

↓

REDUCER

↓

166:1 186:3 196:2 244:1

MAPREDUCE ON A CLUSTER

How MapReduce Scales

Putting it All Together

USER ID	MOVIE ID	RATING	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

MAPPER

196:242 186:302 196:377 244:51 166:346 186:274 186:265

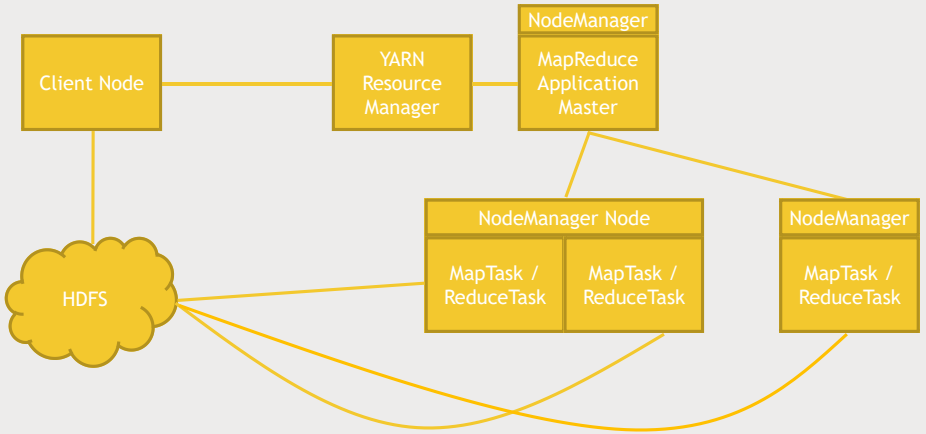
SHUFFLE AND SORT

166:346 186:302,274,265 196:242,377 244:51

REDUCER

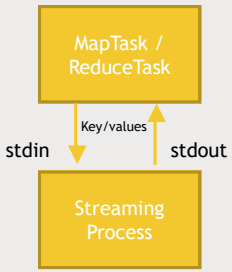
166:1 186:3 196:2 244:1

What's Happening



How are mappers and reducers written?

- MapReduce is natively Java
- STREAMING allows interfacing to other languages (ie Python)



Handling Failure

- Application master monitors worker tasks for errors or hanging
  - *Restarts as needed*
  - *Preferably on a different node*
- What if the application master goes down?
  - *YARN can try to restart it*
- What if an entire Node goes down?
  - *This could be the application master*
  - *The resource manager will try to restart it*
- What if the resource manager goes down?
  - *Can set up "high availability" (HA) using Zookeeper to have a hot standby*

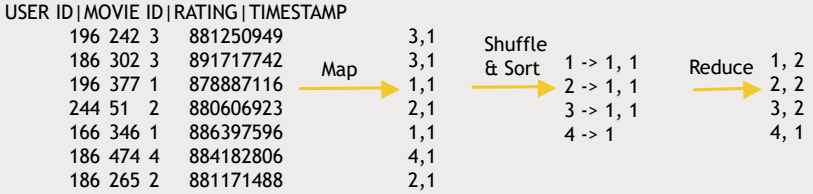
MAPREDUCE: A REAL EXAMPLE

How many of each rating type exist?

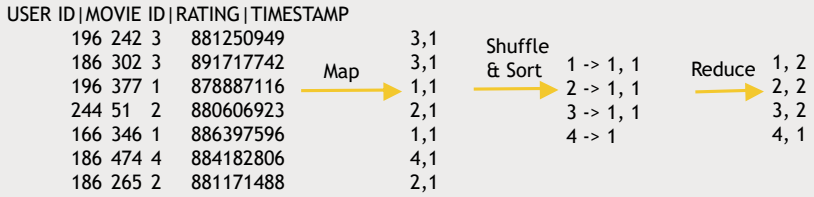
How many of each movie rating exist?

Making it a MapReduce problem

- MAP each input line to (rating, 1)
- REDUCE each rating with the sum of all the 1's

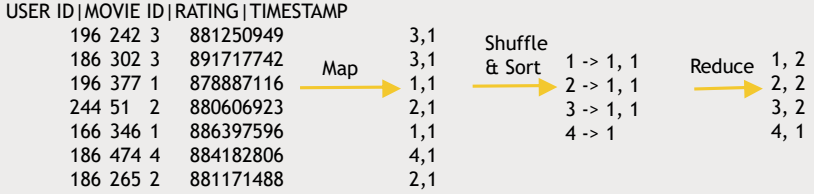


Writing the Mapper




```
def mapper_get_ratings(self, _, line):
    (userID, movieID, rating, timestamp) = line.split('\t')
    yield rating, 1
```

Writing the Reducer




```
def reducer_count_ratings(self, key, values):
    yield key, sum(values)
```

Putting it all together

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class RatingsBreakdown(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_ratings,
                   reducer=self.reducer_count_ratings)
        ]

    def mapper_get_ratings(self, _, line):
        (userID, movieID, rating, timestamp) = line.split('\t')
        yield rating, 1

    def reducer_count_ratings(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    RatingsBreakdown.run()
```

RUNNING MAPREDUCE WITH MRJOB

Run our MapReduce job in our Hadoop installation

Installing what we need

- PIP
  - *Utility for installing Python packages*
  - `su root`
    `yum install python-pip`
- Nano
  - `yum install nano`
- MRJob
  - `pip install mrjob`
    `exit`
- Data files and the script
  - `wget http://media.sundog-soft.com/hadoop/ml-100k/u.data`
  - `wget http://media.sundog-soft.com/hadoop/RatingsBreakdown.py`

Running with mrjob

- Run locally
  - *python RatingsBreakdown.py u.item*
- Run with Hadoop
  - *python MostPopularMovie.py -r hadoop --hadoop-streaming-jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar u.data*

YOUR CHALLENGE

Sort movies by popularity with Hadoop

Challenge exercise

- Count up ratings given for each movie
  - *All you need is to change one thing in the mapper – we don't care about ratings now, we care about movie ID's!*
  - *Start with this and make sure you can do it.*
  - *You can use nano to just edit the existing RatingsBreakdown.py script*

Stretch goal

- Sort the movies by their numbers of ratings
- Strategy:
  - *Map to (movieID, 1) key/value pairs*
  - *Reduce with output of (rating count, movieID)*
  - *Send this to a second reducer so we end up with things sorted by rating count!*
- Gotchas:
  - *How do we set up more than one MapReduce step?*
  - *How do we ensure the rating counts are sorted properly?*

Multi-stage jobs

- You can chain map/reduce stages together like this:

```
def steps(self):
    return [
        MRStep(mapper=self.mapper_get_ratings,
               reducer=self.reducer_count_ratings),
        MRStep(reducer=self.reducer_sorted_output)
    ]
```

Ensuring proper sorting

- By default, streaming treats all input and output as strings. So things get sorted as strings, not numerically.
- There are different formats you can specify. But for now let's just zero-pad our numbers so they'll sort properly.
- The second reducer will look like this:

```
def reducer_count_ratings(self, key, values):
    yield str(sum(values)).zfill(5), key
```

Iterating through the results

- Spoiler alert!

```
def reducer_sorted_output(self, count, movies):
    for movie in movies:
        yield movie, count
```

CHECK YOUR RESULTS

Did it work?

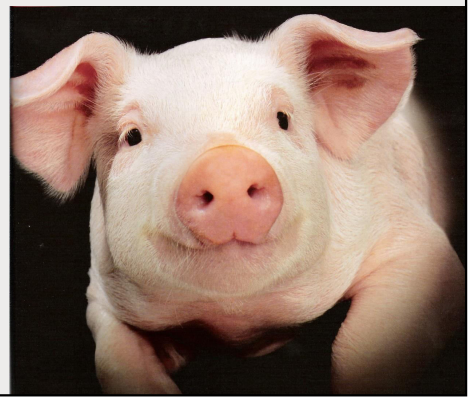
My solution

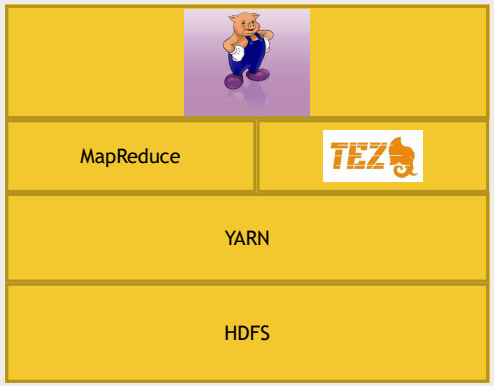
```
from mrjob.job import MRJob
from mrjob.step import MRStep

class RatingsBreakdown(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_ratings,
                   reducer=self.reducer_count_ratings),
            MRStep(reducer=self.reducer_sorted_output)
        ]

    def mapper_get_ratings(self, _, line):
        (userID, movieID, rating, timestamp) = line.split('\t')
        yield movieID, 1

    def reducer_count_ratings(self, key, values):
        yield str(sum(values)).zfill(5), key

    def reducer_sorted_output(self, count, movies):
        for movie in movies:
            yield movie, count

if __name__ == '__main__':
    RatingsBreakdown.run()
```

APACHE PIG

Why Pig?

- Writing mappers and reducers by hand takes a long time.
- Pig introduces *Pig Latin*, a scripting language that lets you use SQL-like syntax to define your map and reduce steps.
- Highly extensible with user-defined functions (UDF's)





Running Pig

- Grunt
- Script
- Ambari / Hue

An example

- Find the oldest 5-star movies

```
ratings = LOAD '/user/maria_dev/ml-100k/u.data' AS
      (userID:int, movieID:int, rating:int, ratingTime:int);
```

This creates a *relation* named “ratings” with a given *schema*.

(660,229,2,891406212)
(421,498,4,892241344)
(495,1091,4,888637503)
(806,421,4,882388897)
(676,538,4,892685437)
(721,262,3,877137285)

Use PigStorage if you need a different delimiter.

```
metadata = LOAD '/user/maria_dev/ml-100k/u.item' USING
      PigStorage('|')AS (movieID:int, movieTitle:chararray,
      releaseDate:chararray, videoRelease:chararray,
      imdbLink:chararray);

DUMP metadata;
```

(1,Toy Story (1995),01-Jan-1995,,http://us.imdb.com/M/title-exact?Toy%20Story%20(1995))
(2,GoldenEye (1995),01-Jan-1995,,http://us.imdb.com/M/title-exact?GoldenEye%20(1995))
(3,Four Rooms (1995),01-Jan-1995,,http://us.imdb.com/M/title-exact?Four%20Rooms%20(1995))
(4,Get Shorty (1995),01-Jan-1995,,http://us.imdb.com/M/title-exact?Get%20Shorty%20(1995))
(5,Copycat (1995),01-Jan-1995,,http://us.imdb.com/M/title-exact?Copycat%20(1995))

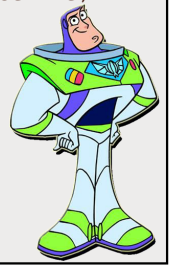
Creating a relation from another relation; FOREACH / GENERATE

```
metadata = LOAD '/user/maria_dev/ml-100k/u.item' USING PigStorage('|')
      AS (movieID:int, movieTitle:chararray, releaseDate:chararray,
      videoRelease:chararray, imdbLink:chararray);

nameLookup = FOREACH metadata GENERATE movieID, movieTitle,
      ToUnixTime(ToDate(releaseDate, 'dd-MMM-yyyy')) AS releaseTime;
```

(1,Toy Story (1995),01-Jan-1995,,http://us.imdb.com/M/title-exact?Toy%20Story%20(1995))

↓

(1,Toy Story (1995),788918400)



Group By

```
ratingsByMovie = GROUP ratings BY movieID;
DUMP ratingsByMovie;
```

(1,{(807,1,4,892528231),(554,1,3,876231938),(49,1,2,888068651), … }
(2,{(429,2,3,882387599),(551,2,2,892784780),(774,2,1,888557383), … }

ratingsByMovie: {group: int,ratings: {(userID: int,movieID: int,rating: int,ratingTime: int)}}

```
avgRatings = FOREACH ratingsByMovie GENERATE group AS movieID,
      AVG(ratings.rating) AS avgRating;
DUMP avgRatings;
```

(1,3.8783185840707963)
(2,3.2061068702290076)
(3,3.033333333333333)
(4,3.550239234449761)
(5,3.302325581395349)

```
DESCRIBE ratings;
DESCRIBE ratingsByMovie;
DESCRIBE avgRatings;
```

ratings: {userID: int,movieID: int,rating: int,ratingTime: int}

ratingsByMovie: {group: int,ratings: {(userID: int,movieID: int,rating: int,ratingTime: int)}}

avgRatings: {movieID: int,avgRating: double}

FILTER

```
fiveStarMovies = FILTER avgRatings BY avgRating > 4.0;
```

(12,4.385767790262173)
(22,4.151515151515151)
(23,4.1208791208791204)
(45,4.05)

JOIN

```
DESCRIBE fiveStarMovies;
DESCRIBE nameLookup;
fiveStarsWithData = JOIN fiveStarMovies BY movieID, nameLookup BY movieID;
DESCRIBE fiveStarsWithData;
DUMP fiveStarsWithData;
```

fiveStarMovies: {movieID: int,avgRating: double}
nameLookup: {movieID: int,movieTitle: chararray,releaseTime: long}

fiveStarsWithData: {fiveStarMovies::movieID: int,fiveStarMovies::avgRating: double,
nameLookup::movieID: int,nameLookup::movieTitle: chararray,nameLookup::releaseTime: long}

(12,4.385767790262173,12,Usual Suspects, The (1995),808358400)
(22,4.151515151515151,22,Braveheart (1995),824428800)
(23,4.1208791208791204,23,Taxi Driver (1976),824428800)

ORDER BY

```
oldestFiveStarMovies = ORDER fiveStarsWithData BY
      nameLookup::releaseTime;
DUMP oldestFiveStarMovies;
```

(493,4.15,493,Thin Man, The (1934),-1136073600)
(604,4.012345679012346,604,It Happened One Night (1934),-1136073600)
(615,4.0508474576271185,615,39 Steps, The (1935),-1104537600)
(1203,4.0476190476190474,1203,Top Hat (1935),-1104537600)

Putting it all together

```
ratings = LOAD '/user/maria_dev/ml-100k/u.data' AS (userID:int, movieID:int, rating:int, ratingTime:int);

metadata = LOAD '/user/maria_dev/ml-100k/u.item' USING PigStorage('|')
    AS (movieID:int, movieTitle:chararray, releaseDate:chararray, videoRelease:chararray, imdbLink:chararray);

nameLookup = FOREACH metadata GENERATE movieID, movieTitle,
    ToUnixTime(ToDate(releaseDate, 'dd-MMM-yyyy')) AS releaseTime;

ratingsByMovie = GROUP ratings BY movieID;

avgRatings = FOREACH ratingsByMovie GENERATE group AS movieID, AVG(ratings.rating) AS avgRating;

fiveStarMovies = FILTER avgRatings BY avgRating > 4.0;

fiveStarsWithData = JOIN fiveStarMovies BY movieID, nameLookup BY movieID;

oldestFiveStarMovies = ORDER fiveStarsWithData BY nameLookup::releaseTime;

DUMP oldestFiveStarMovies;
```

Let's run it

Pig Latin: Diving Deeper
Things you can do to a relation

- LOAD STORE DUMP
  - *STORE ratings INTO 'outRatings' USING PigStorage(':');*
- FILTER DISTINCT FOREACH/GENERATE MAPREDUCE STREAM SAMPLE
- JOIN COGROUP GROUP CROSS CUBE
- ORDER RANK LIMIT
- UNION SPLIT

Diagnostics

- DESCRIBE
- EXPLAIN
- ILLUSTRATE

UDF's

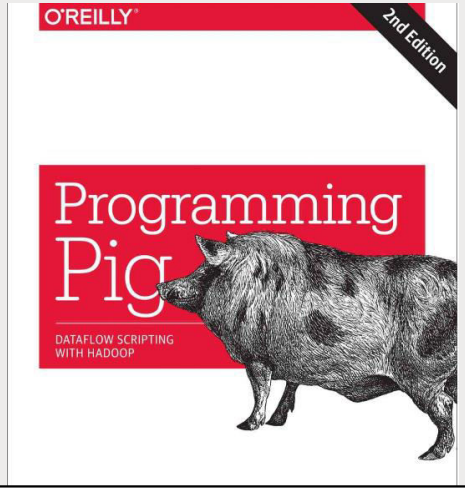
- REGISTER
- DEFINE
- IMPORT

Some other functions and loaders

- AVG CONCAT COUNT MAX MIN SIZE SUM

- PigStorage
- TextLoader
- JsonLoader
- AvroStorage
- ParquetLoader
- OrcStorage
- HBaseStorage

Learning more



PIG CHALLENGE

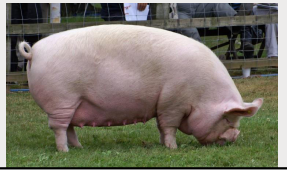
Find the most popular bad movies

Defining the problem

- Find all movies with an average rating less than 2.0
- Sort them by the total number of ratings

Hint

- We used everything you need in our earlier example of finding old movies with ratings greater than 4.0
- Only new thing you need is COUNT(). This lets you count up the number of items in a bag.
  - *So just like you can say AVG(ratings.rating) to get the average rating from a bag of ratings,*
  - *You can say COUNT(ratings.rating) to get the total number of ratings for a given group's bag.*



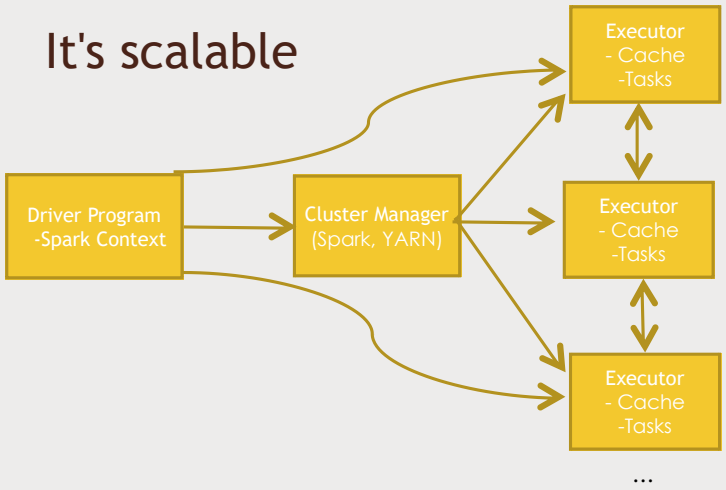
INTRODUCTION TO SPARK

Frank Kane

What is spark?

- "A fast and general engine for large-scale data processing"

It's scalable



It's fast

- "Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk."
- DAG Engine (directed acyclic graph) optimizes workflows

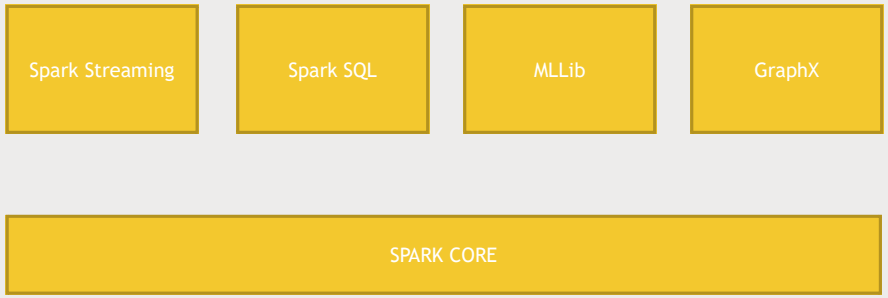
It's hot

- Amazon
- Ebay: log analysis and aggregation
- NASA JPL: Deep Space Network
- Groupon
- TripAdviser
- Yahoo
- Many others: https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark

It's not that hard

- Code in Python, Java, or Scala
- Built around one main concept: the Resilient Distributed Dataset (RDD)

Components of spark



Let's Use Python

- Why Python?
  - *It's a lot simpler, and this is just an overview.*
  - *Don't need to compile anything, deal with JAR's, dependencies, etc*
- But...
  - *Spark itself is written in Scala*
  - *Scala's functional programming model is a good fit for distributed processing*
  - *Gives you fast performance (Scala compiles to Java bytecode)*
  - *Less code & boilerplate stuff than Java*
  - *Python is slow in comparison*

FEAR NOT

- Scala code in Spark looks a LOT like Python code.

Python code to square numbers in a data set:

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
```

Scala code to square numbers in a data set:

```
val nums = sc.parallelize(List(1, 2, 3, 4))
val squared = nums.map(x => x * x).collect()
```

INTRODUCING RDD'S

Frank Kane

RDD

- Resilient
- Distributed
- Dataset

The SparkContext

- Created by your driver program
- Is responsible for making RDD's resilient and distributed!
- Creates RDD's
- The Spark shell creates a "sc" object for you

Creating RDD's

- nums = parallelize([1, 2, 3, 4])
- sc.textFile("file:///c:/users/frank/gobs-o-text.txt")
  - *or s3n:// , hdfs://*
- hiveCtx = HiveContext(sc) rows = hiveCtx.sql("SELECT name, age FROM users")
- Can also create from:
  - *JDBC*
  - *Cassandra*
  - *HBase*
  - *Elastisearch*
  - *JSON, CSV, sequence files, object files, various compressed formats*

Transforming RDD's

- map
- flatmap
- filter
- distinct
- sample
- union, intersection, subtract, cartesian

map example

- rdd = sc.parallelize([1, 2, 3, 4])
- squaredRDD = rdd.map(lambda x: x*x)

- This yields 1, 4, 9, 16

What's that lambda thing?

Many RDD methods accept a *function* as a parameter

```
rdd.map(lambda x: x*x)
```

Is the same thing as

```
def squareIt(x):
    return x*x

rdd.map(squareIt)
```

There, you now understand functional programming.

RDD actions

- collect
- count
- countByValue
- take
- top
- reduce
- … and more …

Lazy evaluation

- Nothing actually happens in your driver program until an action is called!

SPARK SQL

DataFrames and DataSets

Working with structured data

- Extends RDD to a "DataFrame" object
- DataFrames:
  - *Contain Row objects*
  - *Can run SQL queries*
  - *Has a schema (leading to more efficient storage)*
  - *Read and write to JSON, Hive, parquet*
  - *Communicates with JDBC/ODBC, Tableau*

Using SparkSQL in Python

- from pyspark.sql import SQLContext, Row
- hiveContext = HiveContext(sc)
- inputData = spark.read.json(dataFile)
- inputData.createOrReplaceTempView("myStructuredStuff")
- myResultDataFrame = hiveContext.sql("""SELECT foo FROM bar ORDER BY foobar""")

Other stuff you can do with dataframes

- myResultDataFrame.show()
- myResultDataFrame.select("someFieldName")
- myResultDataFrame.filter(myResultDataFrame("someFieldName" > 200)
- myResultDataFrame.groupBy(myResultDataFrame("someFieldName")).mean()
- myResultDataFrame.rdd().map(mapperFunction)

Datasets

- In Spark 2.0, a DataFrame is really a DataSet of Row objects
- DataSets can wrap known, typed data too. But this is mostly transparent to you in Python, since Python is dynamically typed.
- So – don't sweat this too much with Python. But the Spark 2.0 way is to use DataSets instead of DataFrames when you can.

Shell access

- Spark SQL exposes a JDBC/ODBC server (if you built Spark with Hive support)
- Start it with sbin/start-thriftserver.sh
- Listens on port 10000 by default
- Connect using bin/beeline -u jdbc:hive2://localhost:10000
- Viola, you have a SQL shell to Spark SQL
- You can create new tables, or query existing ones that were cached using hiveCtx.cacheTable("tableName")

User-defined functions (UDF's)

```
from pyspark.sql.types import IntegerType
hiveCtx.registerFunction("square", lambda x: x*x, IntegerType())
df = hiveCtx.sql("SELECT square('someNumericFiled') FROM tableName)
```

YOUR CHALLENGE

Filter out movies with very few ratings

The problem

- Our examples of finding the lowest-rated movies were polluted with movies only rated by one or two people.
- Modify one or both of these scripts to only consider movies with at least ten ratings.

Hints

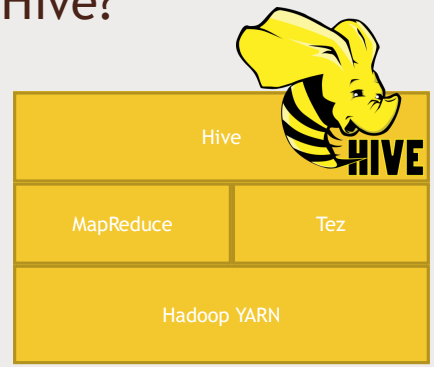
- RDD's have a filter() function you can use
  - *It takes a function as a parameter, which accepts the entire key/value pair*
    - So if you're calling filter() on an RDD that contains (movieID, (sumOfRatings, totalRatings)) – a lambda function that takes in "x" would refer to totalRatings as x[1][1]. x[1] gives us the "value" (sumOfRatings, totalRatings) and x[1][1] pulls out totalRatings.
  - *This function should be an expression that returns True if the row should be kept, or False if it should be discarded*
- DataFrames also have a filter() function
  - *It's easier – you just pass in a string expression for what you want to filter on.*
  - *For example: df.filter("count > 10") would only pass through rows where the "count" column is greater than 10.*

GOOD LUCK

HIVE

Distributing SQL queries with Hadoop

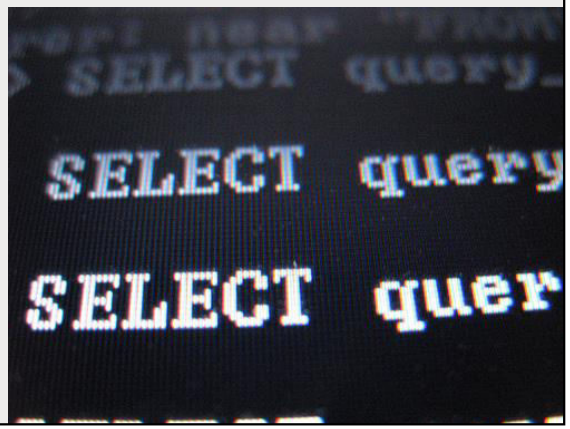
What is Hive?



Translates SQL queries to MapReduce or Tez jobs on your cluster!

Why Hive?

- Uses familiar SQL syntax (HiveQL)
- Interactive
- Scalable - works with "big data" on a cluster
  - *Really most appropriate for data warehouse applications*
- Easy OLAP queries - WAY easier than writing MapReduce in Java
- Highly optimized
- Highly extensible
  - *User defined functions*
  - *Thrift server*
  - *JDBC / ODBC driver*



Why not Hive?

- High latency - not appropriate for OLTP
- Stores data de-normalized
- SQL is limited in what it can do
  - *Pig, Spark allows more complex stuff*
- No transactions
- No record-level updates, inserts, deletes

HiveQL

- Pretty much MySQL with some extensions
- For example: views
  - *Can store results of a query into a "view", which subsequent queries can use as a table*
- Allows you to specify how structured data is stored and partitioned

Let's just dive into an example.

HOW HIVE WORKS

Schema On Read

- Hive maintains a "metastore" that imparts a structure you define on the unstructured data that is stored on HDFS etc.

```
CREATE TABLE ratings (
    userID INT,
    movieID    INT,
    rating INT,
    time    INT)
ROW FORMAT DELIMTED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '${env:HOME}/ml-100k/u.data'
OVERWRITE INTO TABLE ratings;
```

Where is the data?

- LOAD DATA
  - *MOVES data from a distributed filesystem into Hive*
- LOAD DATA LOCAL
  - *COPIES data from your local filesystem into Hive*
- Managed vs. External tables

```
CREATE EXTERNAL TABLE IF NOT EXISTS ratings (
    userID    INT,
    movieID INT,
    rating    INT,
    time      INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LOCATION '/data/ml-100k/u.data';
```

Partitioning

- You can store your data in partitioned subdirectories
  - *Huge optimization if your queries are only on certain partitions*

```
CREATE TABLE customers (
      name     STRING,
      address  STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
PARTITIONED BY (country STRING);
```

```
…/customers/country=CA/
…/customers/country=GB/
```

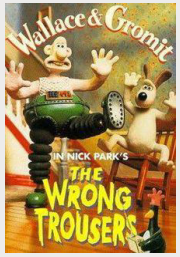
Ways to use Hive

- Interactive via hive> prompt / Command line interface (CLI)
- Saved query files
  - *hive -f /somepath/queries.hql*
- Through Ambari / Hue
- Through JDBC/ODBC server
- Through Thrift service
  - *But remember, Hive is not suitable for OLTP*
- Via Oozie

HIVE CHALLENGE

Find the movie with the highest average rating

- Hint: AVG() can be used on aggregated data, like COUNT() does.
- Extra credit: only consider movies with more than 10 ratings



INTEGRATING MYSQL & HADOOP

Fun with Sqoop

What's MySQL?

- Popular, free relational database
- Generally monolithic in nature
- But, can be used for OLTP - so exporting data into MySQL can be useful
- Existing data may exist in MySQL that you want to import to Hadoop

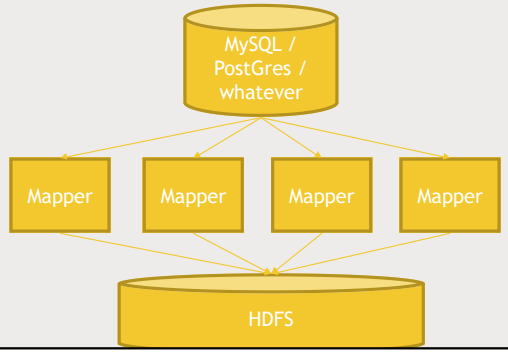
MySQL®

Sqoop to the rescue

sqoop

Sqoop can handle BIG data

- Actually kicks off MapReduce jobs to handle importing or exporting your data!



Sqoop: Import data from MySQL to HDFS

sqoop import --connect jdbc:mysql://localhost/movielens --driver com.mysql.jdbc.Driver --table movies

Sqoop: Import data from MySQL directly into Hive!

- sqoop import --connect jdbc:mysql://localhost/movielens --driver com.mysql.jdbc.Driver --table movies --hive-import

Incremental imports

- You can keep your relational database and Hadoop in sync
- --check-column and --last-value

Sqoop: Export data from Hive to MySQL

- sqoop export --connect jdbc:mysql://localhost/movielens -m 1 --driver com.mysql.jdbc.Driver --table exported_movies --export-dir /apps/hive/warehouse/movies --input-fields-terminated-by '\0001'
- Target table must already exist in MySQL, with columns in expected order

Let's play with MySQL and Sqoop

- Import MovieLens data into a MySQL database
- Import the movies to HDFS
- Import the movies into Hive
- Export the movies back into MySQL

NOSQL

When RDBMS's don't cut it

Random Access to Planet-Size Data

Scaling up MySQL etc. to massive loads requires extreme measures

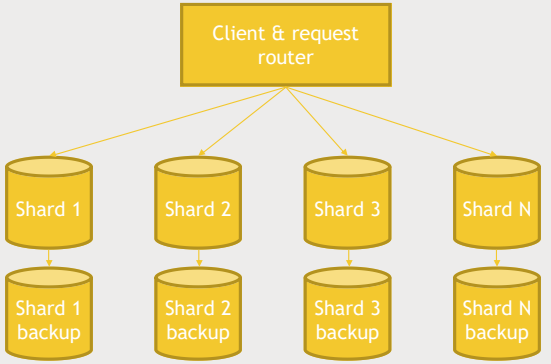
- Denormalization
- Caching layers
- Master/slave setups
- Sharding
- Materialized views
- Removing stored procedures

Support nightmare

Do you really need SQL?

- Your high-transaction queries are probably pretty simple once de-normalized
- A simple get / put API may meet your needs
- Looking up values for a given key is simple, fast, and scalable
- You can do both…

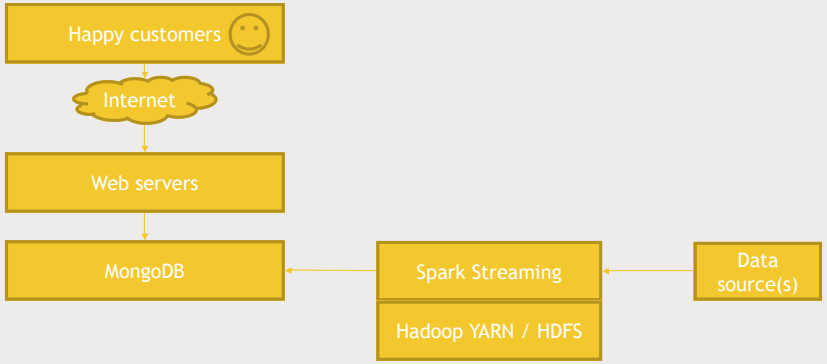
Sample architecture



Use the right tool for the job

- For analytic queries, Hive, Pig, Spark, etc. work great.
- Exporting data to MySQL is plenty fast for most applications too.
- But if you work at giant scale – export your data to a non-relational database for fast and scalable serving of that data to web applications, etc.

Sample application architecture (greatly simplified!)



Every box represents many machines!

Choose a database design that meets your usage patterns

HBASE

Non-relational, scalable database built on HDFS

Based on Google's BigTable

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

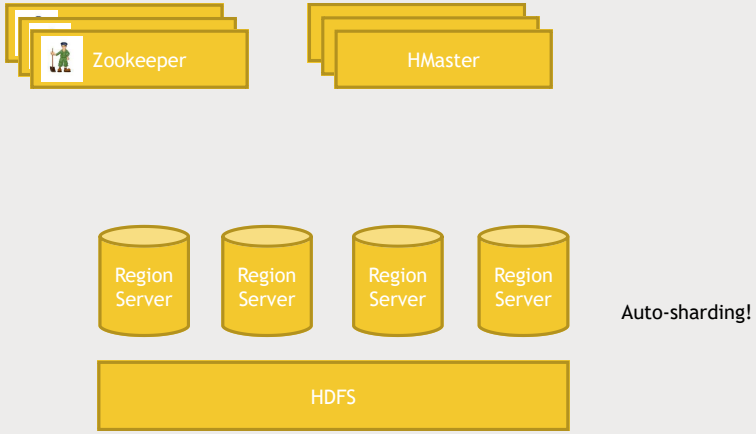
2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map

CRUD

- Create
- Read
- Update
- Delete
- There is no query language, only CRUD API's!

HBase architecture

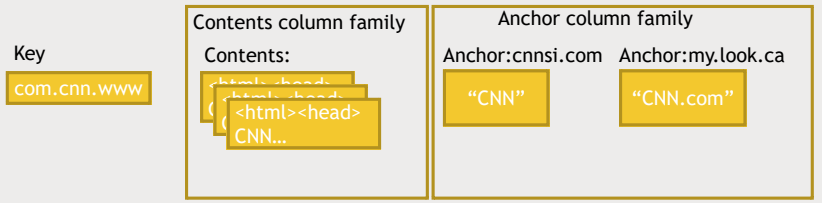


Auto-sharding!

HBase data model

- Fast access to any given ROW
- A ROW is referenced by a unique KEY
- Each ROW has some small number of COLUMN FAMILIES
- A COLUMN FAMILY may contain arbitrary COLUMNS
- You can have a very large number of COLUMNS in a COLUMN FAMILY
- Each CELL can have many VERSIONS with given timestamps
- Sparse data is A-OK – missing columns in a row consume no storage.

Example: One row of a web table



Some ways to access HBase

- HBase shell
- Java API
  - *Wrappers for Python, Scala, etc.*
- Spark, Hive, Pig
- REST service
- Thrift service
- Avro service

LET'S PLAY WITH HBASE

Creating a HBase table with Python via REST

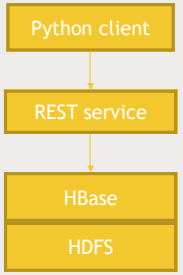
What are we doing?

- Create a HBase table for movie ratings by user
- Then show we can quickly query it for individual users
- Good example of sparse data

Column family: rating

UserID	Rating:50	Rating:33	Rating:223
	1	5	5

How are we doing it?



Let's do this

HBASE / PIG INTEGRATION

Populating HBase at scale

Integrating Pig with HBase

- Must create HBase table ahead of time
- Your relation must have a unique key as its first column, followed by subsequent columns as you want them saved in Hbase
- USING clause allows you to STORE into an HBase table
- Can work at scale – Hbase is transactional on rows

Let's do this

CASSANDRA

A distributed database with no single point of failure

Cassandra – NoSQL with a twist

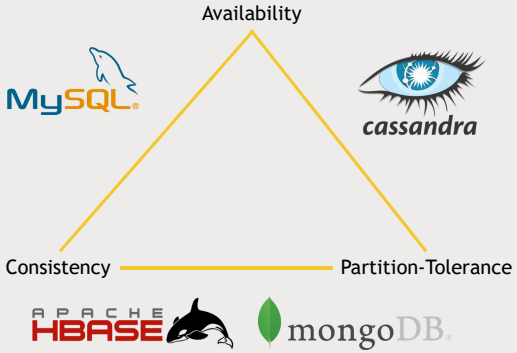
- Unlike HBase, there is no master node at all – every node runs exactly the same software and performs the same functions
- Data model is similar to BigTable / Hbase
- It's non-relational, but has a limited CQL query language as its interface

cassandra

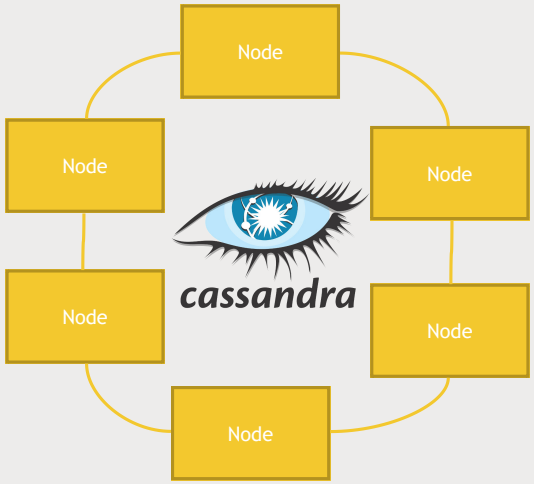
Cassandra's Design Choices

- The CAP Theorem says you can only have 2 out of 3: consistency, availability, partition-tolerance
  - *And partition-tolerance is a requirement with "big data," so you really only get to choose between consistency and availability*
- Cassandra favors availability over consistency
  - *It is "eventually consistent"*
  - *But you can specify your consistency requirements as part of your requests. So really it's "tunable consistency"*

Where Cassandra Fits in CAP tradeoffs

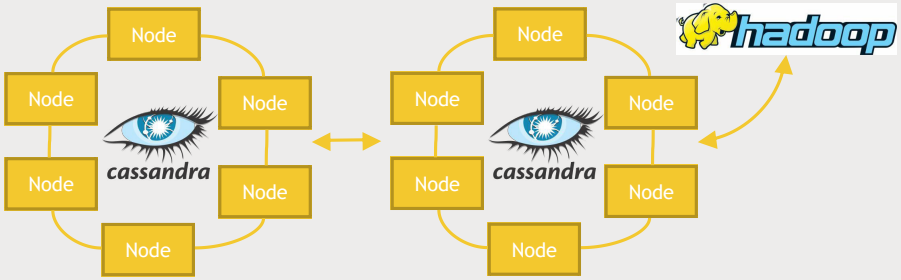


Cassandra architecture



Cassandra and your cluster

- Cassandra's great for fast access to rows of information
- Get the best of both worlds – replicate Cassandra to a another ring that is used for analytics and Spark integration



CQL (Wait, I thought this was NoSQL!)

- Cassandra's API is CQL, which makes it easy to look like existing database drivers to applications.
- CQL is like SQL, but with some big limitations!
  - *NO JOINS*
    - Your data must be de-normalized
    - So, it's still non-relational
  - *All queries must be on some primary key*
    - Secondary indices are supported, but…
- CQLSH can be used on the command line to create tables, etc.
- All tables must be in a *keyspace* - keyspaces are like databases

Cassandra and Spark

- DataStax offers a Spark-Cassandra connector
- Allows you to read and write Cassandra tables as DataFrames
- Is smart about passing queries on those DataFrames down to the appropriate level
- Use cases:
  - *Use Spark for analytics on data stored in Cassandra*
  - *Use Spark to transform data and store it into Cassandra for transactional use*

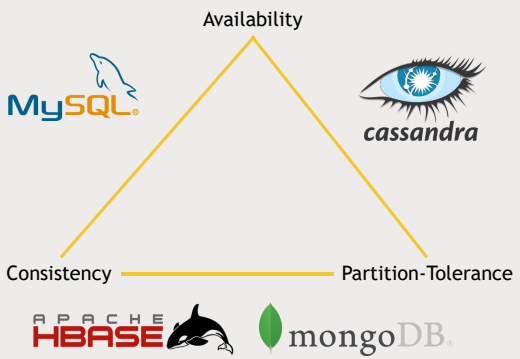
Let's Play

- Install Cassandra on our virtual Hadoop node
- Set up a table for MovieLens users
- Write into that table and query it from Spark!

MONGODB

Managing HuMONGOus data

Where are we?



Document-based data model

Looks like JSON. Example:

```
{
        "_id" : ObjectID("7b33e366ae32223aee34fd3"),
        "title" : "A blog post about MongoDB",
        "content" : "This is a blog post about MongoDB",
        "comments": [
                {
                        "name" : "Frank",
                        "email" : fkane@sundog-soft.com,
                        "content" : "This is the best article ever written!"
                        "rating" : 1
                }
        ]
}
```

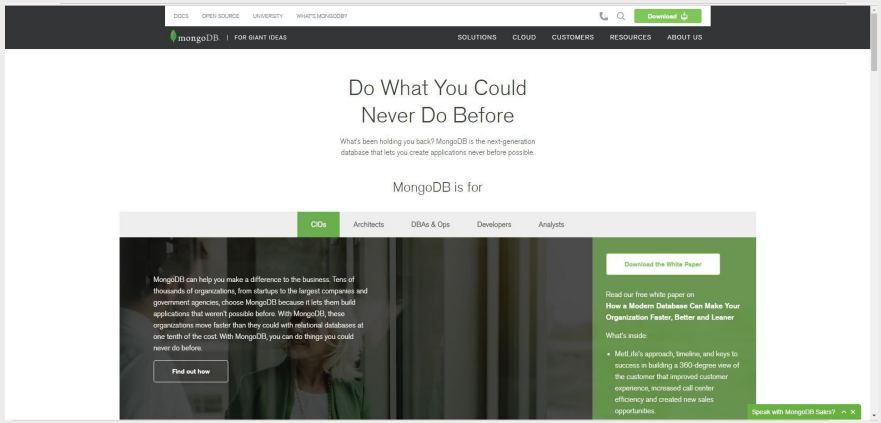
No real schema is enforced.

- You can have different fields in every document if you want to
- No single "key" as in other databases
  - *But you can create indices on any fields you want, or even combinations of fields.*
  - *If you want to "shard", then you must do so on some index.*
- Results in a lot of flexibility
  - *But with great power comes great responsibility*

MongoDB terminology

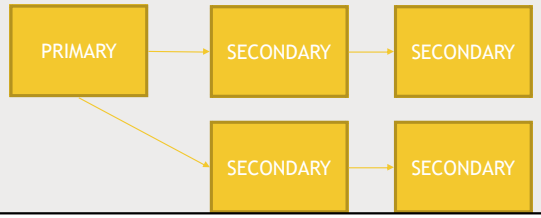
- Databases
- Collections
- Documents

It’s kinda corporate-y



Replication Sets

- Single-master!
- Maintains backup copies of your database instance
  - *Secondaries can elect a new primary within seconds if your primary goes down*
  - *But make sure your operation log is long enough to give you time to recover the primary when it comes back…*

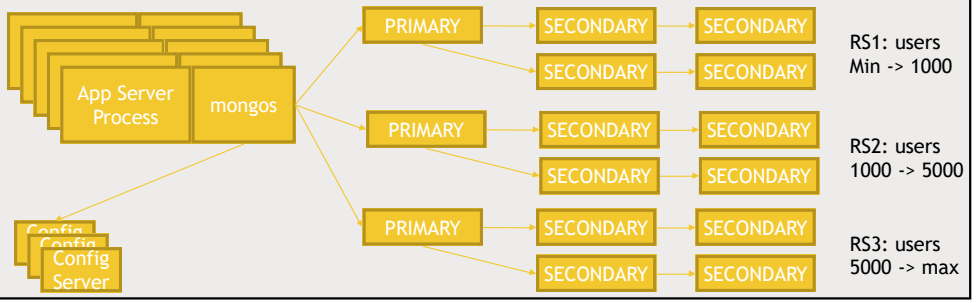


Replica Set Quirks

- A majority of the servers in your set must agree on the primary
  - *Even numbers of servers (like 2) don’t work well*
- Don’t want to spend money on 3 servers? You can set up an ‘arbiter’ node
  - *But only one*
- Apps must know about enough servers in the replica set to be able to reach one to learn who’s primary
- Replicas only address durability, not your ability to scale
  - *Well, unless you can take advantage of reading from secondaries - which generally isn’t recommended*
  - *And your DB will still go into read-only mode for a bit while a new primary is elected*
- Delayed secondaries can be set up as insurance against people doing dumb things

Sharding

- Finally - “big data”
- Ranges of some indexed value you specify are assigned to different replica sets



Sharding Quirks

- Auto-sharding sometimes doesn't work
  - *Split storms, mongos processes restarted too often*
- You must have 3 config servers
  - *And if any one goes down, your DB is down*
  - *This is on top of the single-master design of replica sets*
- MongoDB's loose document model can be at odds with effective sharding

Neat Things About MongoDB

- It's not just a NoSQL database - very flexible document model
- Shell is a full JavaScript interpreter
- Supports many indices
  - *But only one can be used for sharding*
  - *More than 2-3 are still discouraged*
  - *Full-text indices for text searches*
  - *Spatial indices*
- Built-in aggregation capabilities, MapReduce, GridFS
  - *For some applications you might not need Hadoop at all*
  - *But MongoDB still integrates with Hadoop, Spark, and most languages*
- A SQL connector is available
  - *But MongoDB still isn't designed for joins and normalized data really.*

Let's Mess Around

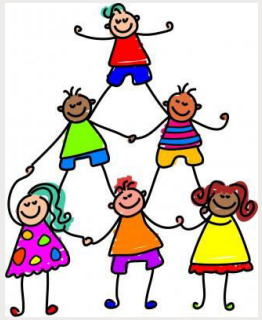
- We'll integrate MongoDB with Spark
- Then play around with the resulting database in the mongo shell

CHOOSING YOUR DATABASE

Integration considerations

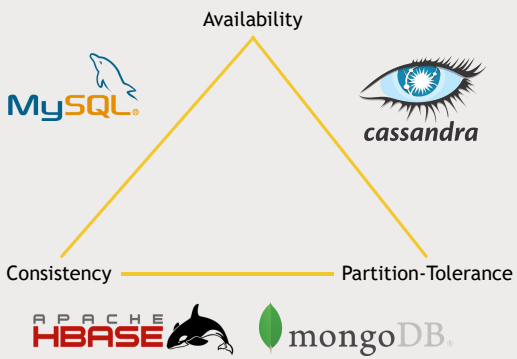
Scaling requirements

Support considerations



Budget considerations? Probably not.

CAP considerations



Simplicity

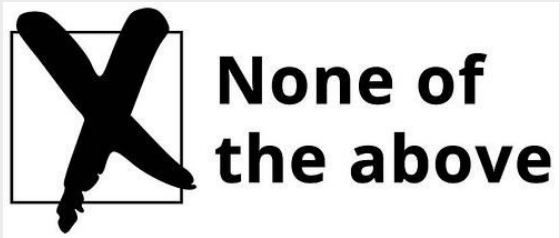
An example

- You're building an internal phone directory app
  - *Scale: limited*
  - *Consistency: Eventual is fine*
  - *Availability requirements: not mission critical*
  - *MySQL is probably already installed on your web server…*

MySQL®

Another example

- You want to mine web server logs for interesting patterns
- What are the most popular times of day? What's the average session length? Etc.



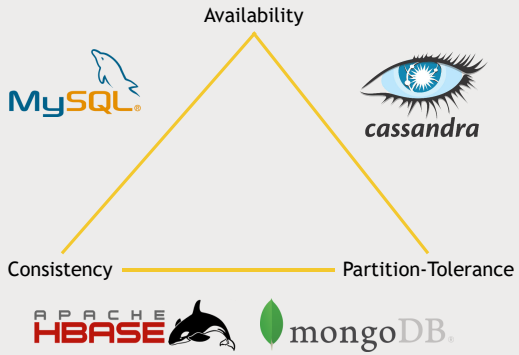
Another example

- You have a big Spark job that produces movie recommendations for end users nightly
- Something needs to vend this data to your web applications
- You work for some huge company with massive scale
- Downtime is not tolerated
- Must be fast
- Eventual consistency OK – it's just reads

You try it!

- You're building a massive stock trading system
- Consistency is more important than anything
- "Big data" is present
- It's really, really important – so having access to professional support might be a good idea. And you have enough budget to pay for it.

CAP considerations



APACHE DRILL

SQL for noSQL

What is Apache Drill?

- A SQL query engine for a variety of non-relational databases and data files
  - *Hive, MongoDB, HBase*
  - *Even flat JSON or Parquet files on HDFS, S3, Azure, Google cloud, local file system*
- Based on Google's Dremel

It's real SQL

- Not SQL-Like
- And it has a ODBC / JDBC driver so other tools can connect to it just like any relational database

It's fast and pretty easy to set up.

- But remember, these are still non-relational databases under the hood!
- Allows SQL analysis of disparate data source without having to transform and load it first
  - *Internally data is represented as JSON and so has no fixed schema*

You can even do joins across different database technologies

- Or with flat JSON files that are just sitting around

Think of it as SQL for your entire ecosystem

Let's drill

- We'll import data into Hive and MongoDB
- Set up Drill on top of both
- And do some queries!

APACHE PHOENIX

SQL for HBase

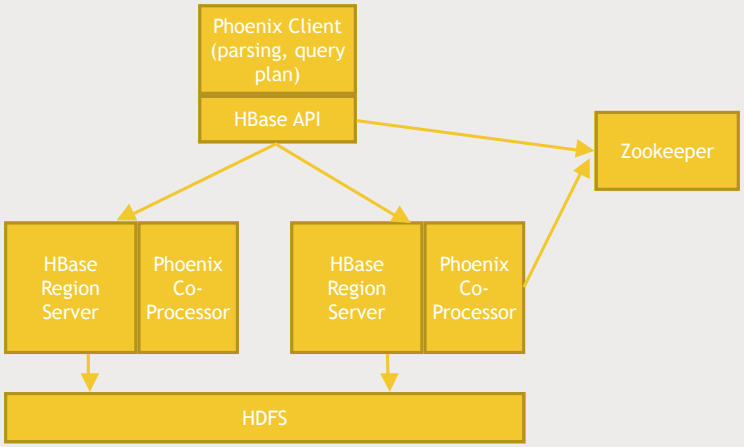
What is Phoenix?

- A SQL driver for HBase that supports transactions
- Fast, low-latency – OLTP support
- Originally developed by Salesforce, then open-sourced
- Exposes a JDBC connector for HBase
- Supports secondary indices and user-defined functions
- Integrates with MapReduce, Spark, Hive, Pig, and Flume

Why Phoenix?

- It's really fast. You probably won't pay a performance cost from having this extra layer on top of Hbase.
- Why Phoenix and not Drill?
  - *Well, choose the right tool for the job.*
- Why Phoenix and not HBase's native clients?
  - *Your apps, and analysts, may find SQL easier to work with.*
  - *Phoenix can do the work of optimizing more complex queries for you*
    - But remember HBase is still fundamentally non-relational!

Phoenix architecture



Using Phoenix

- Command-Line Interface (CLI)
- Phoenix API for Java
- JDBC Driver (thick client)
- Phoenix Query Server (PQS) (thin client)
  - *Intended to eventually enable non-JVM access*
- JAR's for MapReduce, Hive, Pig, Flume, Spark

Let's Play

- Install Phoenix on our Hortonworks Sandbox
- Mess around with the CLI
- Set up a users table for MovieLens
- Store and load data to it through the Pig integration

One more cool Phoenix picture

PRESTO

Distributing queries across different data stores

What is Presto

- It's a lot like Drill
  - *It can connect to many different "big data" databases and data stores at once, and query across them*
  - *Familiar SQL syntax*
  - *Optimized for OLAP - analytical queries, data warehousing*
- Developed, and still partially maintained by Facebook
- Exposes JDBC, Command-Line, and Tableau interfaces

Why Presto

- Vs. Drill? Well, it has a Cassandra connector for one thing.
- If it's good enough for Facebook…
  - "Facebook uses Presto for interactive queries against several internal data stores, including their 300PB data warehouse. Over 1,000 Facebook employees use Presto daily to run more than 30,000 queries that in total scan over a petabyte each per day."
  - Also used by DropBox and AirBNB
- "A single Presto query can combine data from multiple sources, allowing for analytics across your entire organization."
- "Presto breaks the false choice between having fast analytics using an expensive commercial solution or using a slow "free" solution that requires excessive hardware."

What can Presto connect to?

- Cassandra (It's Facebook, after all)
- Hive
- MongoDB
- MySQL
- Local files
- And stuff we haven't talked about just yet:
  - *Kafka, JMX, PostgreSQL, Redis, Accumulo*

Let's just dive in

- Set up Presto
- Query our Hive ratings table using Presto
- Spin Cassandra back up, and query our users table in Cassandra with Presto
- Execute a query that joins users in Cassandra with ratings in Hive!

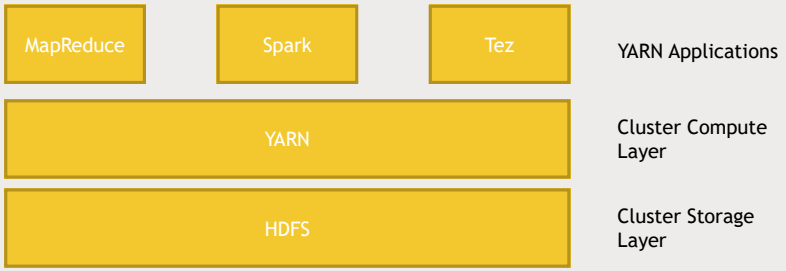
HADOOP YARN

Yet Another Resource Negotiator

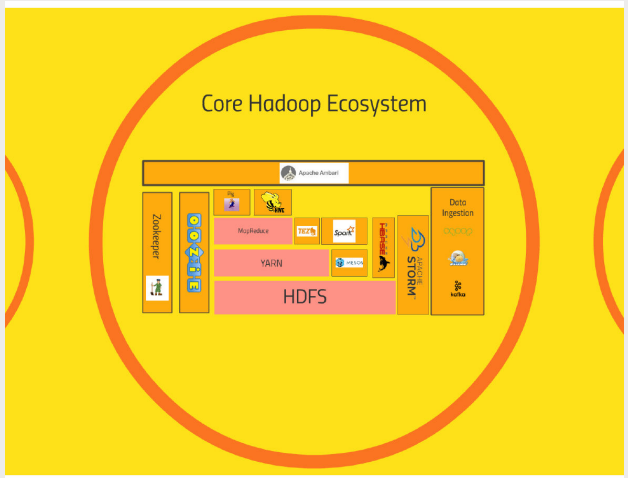
What is YARN?

- Yet Another Resource Negotiator
  - *Introduced in Hadoop 2*
  - *Separates the problem of managing resources on your cluster from MapReduce*
  - *Enabled development of MapReduce alternatives (Spark, Tez) built on top of YARN*
- It's just there, under the hood, managing the usage of your cluster
  - *I can't think of a reason why you'd need to actually write code against it yourself in this day and age. But you can.*

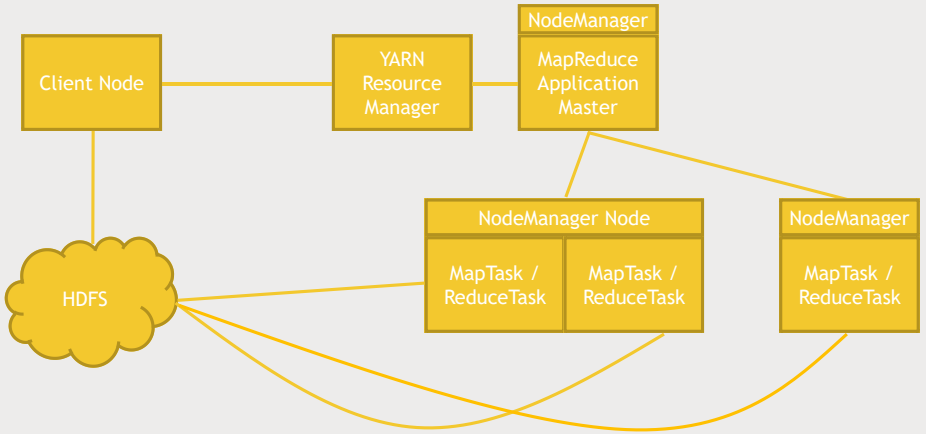
Where YARN fits in



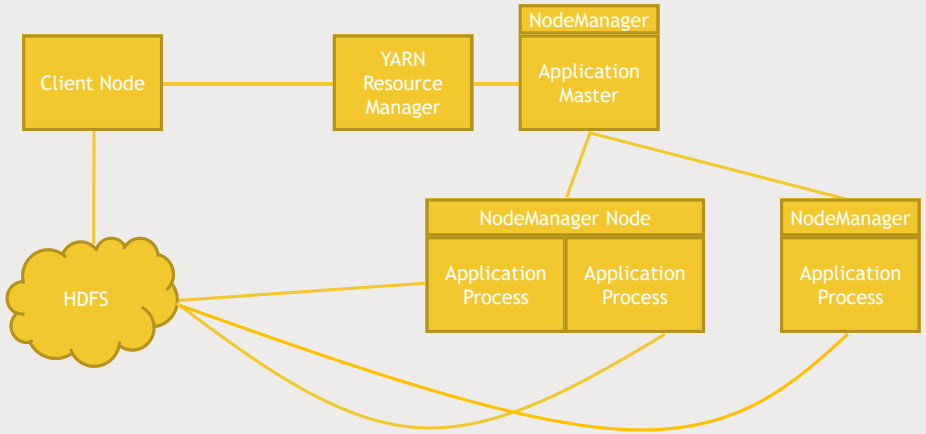
Where YARN fits in



Remember how MapReduce works



YARN just generalizes this



How YARN works

- Your application talks to the Resource Manager to distribute work to your cluster
- You can specify data locality - which HDFS block(s) do you want to process?
  - *YARN will try to get your process on the same node that has your HDFS blocks*
- You can specify different scheduling options for applications
  - *So you can run more than one application at once on your cluster*
  - *FIFO, Capacity, and Fair schedulers*
    - FIFO runs jobs in sequence, first in first out
    - Capacity may run jobs in parallel if there's enough spare capacity
    - Fair may cut into a larger running job if you just want to squeeze in a small one

Building new YARN applications

- Why? There are so many existing projects you can just use
  - *Need a DAG*-based application? Build it on Spark or Tez*
    - (*Directed Acyclic Graph)
- But if you really really need to
  - *There are frameworks: Apache Slider, Apache Twill*
  - *And there are some books on the topic.*

And that's really all there is to say.

- Want to practice "using YARN?" Well, we already did that with MapReduce and Spark!
- You just need to know it's there, under the hood, managing your cluster's resources for you
- Thanks YARN!

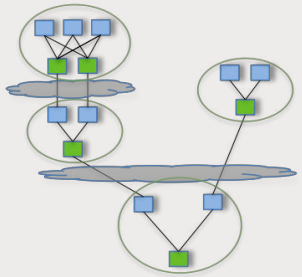
APACHE TEZ

Directed Acyclic Graph Framework

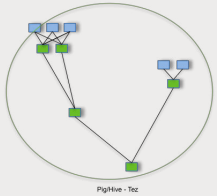
What is Tez?

- Another bit of infrastructure you can just use
  - *Makes your Hive, Pig, or MapReduce jobs faster!*
  - *It's an application framework clients can code against as a replacement for MapReduce*
- Constructs Directed Acyclic Graphs (DAGs) for more efficient processing of distributed jobs
  - *Relies on a more holistic view of your job; eliminates unnecessary steps and dependencies.*
- Optimizes physical data flow and resource usage

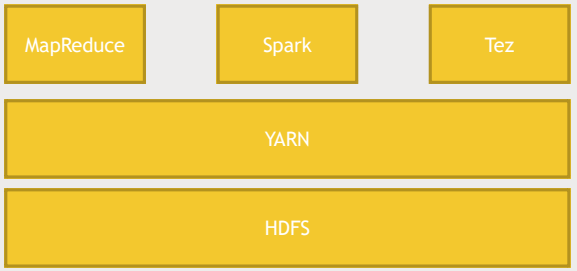
Directed Acyclic Graphs



Pig/Hive - MR



Where Tez sits



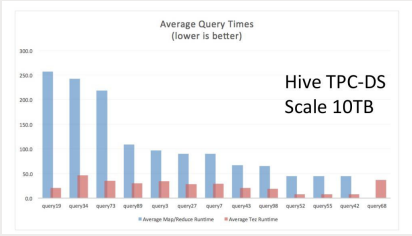
YARN Applications

Cluster Compute Layer

Cluster Storage Layer

Just tell Hive or Pig to use it.

- It probably does by default anyhow.
- It really is a lot faster!



Let's try it out

- Compare performance of a Hive query using Tez vs. MapReduce

MESOS

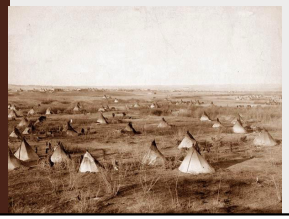
Yes, yet another resource negotiator

What is Mesos?

- Came out of Twitter – it's a system that manages resources across your data center(s)
- Not just for big data stuff – it can allocate resources for web servers, small scripts, whatever.
- Meant to solve a more general problem than YARN – it's really a general container management system

We're kind of going off the reservation

- Mesos isn't really part of the Hadoop ecosystem per se, but it can play nice with it
  - *Spark and Storm may both run on Mesos instead of YARN*
  - *Hadoop YARN may be integrated with Mesos using Myriad*
    - So you don't necessarily need to partition your data center between your Hadoop cluster and everything else!



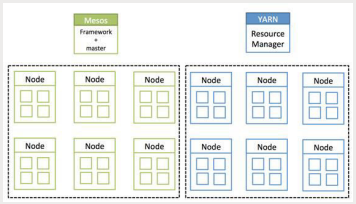
Differences between Mesos and YARN

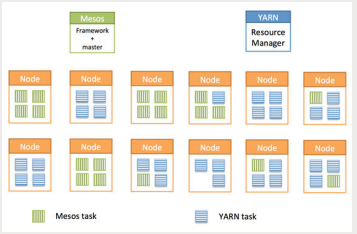
- YARN is a monolithic scheduler – you give it a job, and YARN figures out where to run
- Mesos is a two-tiered system
  - *Mesos just makes offers of resources to your application ("framework")*
  - *Your framework decides whether to accept or reject them*
  - *You also decide your own scheduling algorithm*
- YARN is optimized for long, analytical jobs like you see in Hadoop
- Mesos is built to handle that, as well as long-lived processes (servers) and short-lived processes as well.

How Mesos fits in

- If you're looking for an architecture you can code all of your organization's cluster applications against – not just Hadoop stuff – Mesos can be really useful
  - *You should also look at Kuberentes / Docker*
- If all you are about is Spark and Storm from the Hadoop-y world, Mesos is an option
  - *But YARN's probably better, especially if your data is on HDFS*
  - *Spark on Mesos is limited to one executor per slave (node)*

Siloed vs resource sharing





Images from Mesosphere

When to use Mesos?

- If your organization as a whole has chosen to use Mesos to manage its computing resources in general
  - *Then you can avoid partitioning off a Hadoop cluster by using Myriad*
  - *There is also a "Hadoop on Mesos" package for Cloudera that bypasses YARN entirely*
- Otherwise – probably not. I just want you know what Mesos is and how it's different from YARN.