Large-scale text processing pipeline with Apache Spark

Alexey Svyatkovskiy, Kosuke Imai, Mary Kroeger, Yuki Shiraito

Princeton University

Big NLP Workshop, IEEE Big Data conference, Washington DC, December 5th 2016

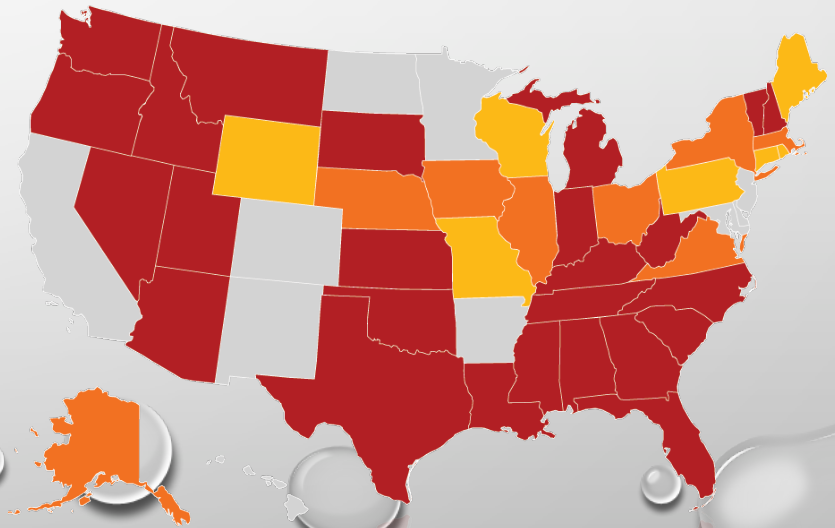
OUTLINE

- Policy diffusion detection in U.S. legislature: the problem
- Solving the problem at scale
  - Apache Spark, Hadoop, Scala
  - Text processing pipeline: core modules
- Text processing pipeline
  - Data ingestion. Apache Avro and Parquet data formats
  - Pre-processing and feature extraction
  - Candidate selection via clustering
- Policy diffusion detection modes
  - All-pairs similarity join
  - Reformulating problem as a network graph
- Interactive analysis with Histogrammar tool
- Conclusion and next steps

POLICY DIFFUSION DETECTION: THE PROBLEM

- Policy diffusion detection is a problem from a wider class of fundamental text mining problems of finding similar items, including plagiarism and mirror website detection
- It occurs when government decisions in a given jurisdiction are systematically influenced by prior policy choices made in other jurisdictions, in a different state on a different year
- Example: “Stand your ground” bills first introduced in Florida, Michigan and South Carolina 2005
  - A number of states have passed a form of SYG bills in 2012 after T. Martin’s death
- We focus on a type of policy diffusion that can be detected by examining similarity of bill texts

States that have passed SYG laws
States that have passed SYG laws since T. Martin’s death
States that have proposed SYG laws after T. Martin’s death



SSH

Login node

Spark
Hue
YP server
LDAP
…

Service node 1

Zookeeper
Journal node
Primary namenode
httpfs

Service node 2

Zookeeper
Journal node
Resource manager
Hive master

Service node 3

Zookeeper
Journal node
Standby namenode
History server

- A 10 node SGI Linux Hadoop cluster
  - Intel Xeon CPU E5-2680 v2 @ 2.80GHz CPU processors, 256 GB RAM
  - All the servers mounted on one rack and interconnected using a 10 Gigabit Ethernet switch
- Cloudera distribution of Hadoop configured in high-availability mode using two namenodes
  - Schedule Spark applications using YARN
  - Distributed file system (HDFS)

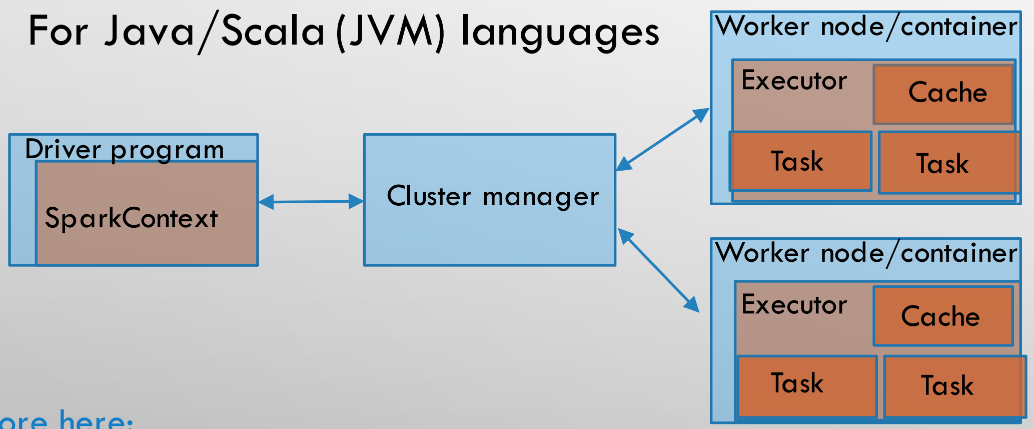
Datanodes

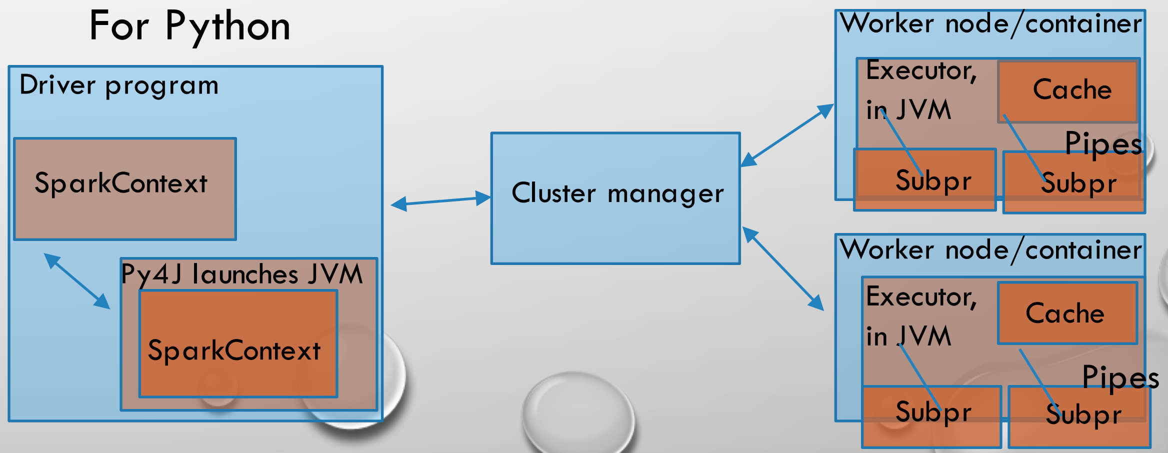
Spark
HDFS
Datanode service

hadoop

ANATOMY OF A SPARK APPLICATIONS

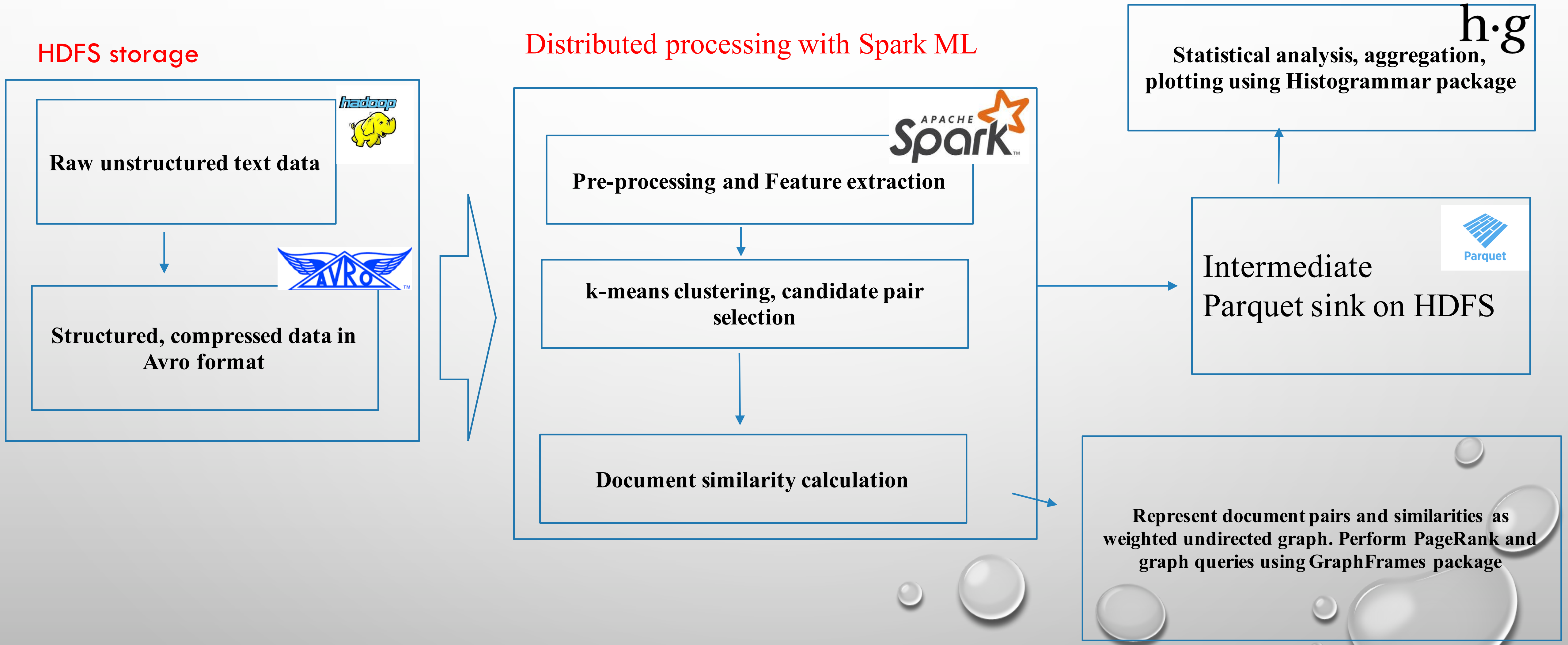
- Spark uses master/slave architecture with one central coordinator (driver) and many distributed workers (executors)
- Driver runs its own Java process, executors each run their own Java processes
- For PySpark, SparkContext uses Py4J to launch a JVM and create a JavaSparkContext
  - Executors all ran in JVMs, and Python subprocesses (tasks) which are launched communicate with them using pipes
- We choose Scala for our implementation because unlike Python and R it is statically typed, and the cost of JVM communication is minimal





See more here: https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals

TEXT PROCESSING PIPELINE: CORE MODULES


HDFS storage
Raw unstructured text data
Structured, compressed data in Avro format
Distributed processing with Spark ML
Pre-processing and Feature extraction
k-means clustering, candidate pair selection
Document similarity calculation
Intermediate Parquet sink on HDFS
Statistical analysis, aggregation, plotting using Histogrammar package
Represent document pairs and similarities as weighted undirected graph. Perform PageRank and graph queries using GraphFrames package

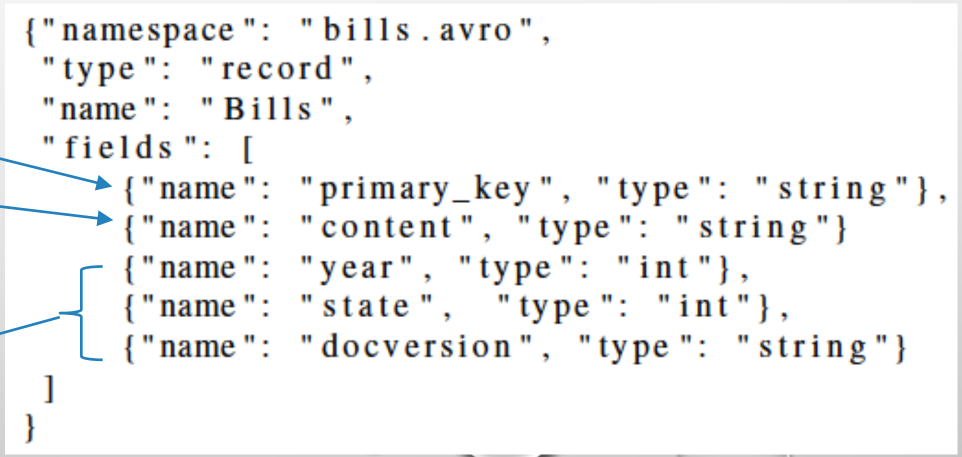
DATA INGESTION

- Our dataset is based on the LexisNexis StateNet dataset which contains a total of more than 7 million legislative bills from 50 US states from 1991 to 2016
- The initial dataset contains unstructured text documents sub-divided by year and state
- We use Apache Avro serialization framework to store the data and access it efficiently in Spark applications
  - Binary JSON meta-format with schema stored with the data
  - Row-oriented, good for nested schemas. Supports schema evolution

Unique identifier of the bill

Entire contents of the bill as a string, not read into memory during candidate selection and filtering steps, thanks to the Avro schema evolution property

Used to construct predicates and filter the data before calculating joins



```
{"namespace": "bills.avro",
 "type": "record",
 "name": "Bills",
 "fields": [
     {"name": "primary_key", "type": "string"},
     {"name": "content", "type": "string"}
     {"name": "year", "type": "int"},
     {"name": "state", "type": "int"},
     {"name": "docversion", "type": "string"}
 ]
}
```

MACHINE LEARNING: SPARK ML

- Spark ML standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline or workflow

Here are the basic components of a pipeline

- **Transformer**: A Transformer is an algorithm which can transform one DataFrame into another DataFrame. Each transformer class has a Transform method
  - E. g. tokenizer, stop word remover are transformers
- **Estimator**: An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a modeL. Each estimator class has a Fit method
  - E.g. RandomForest classifier
- **Parameter**: All Transformers and Estimators share a common API for specifying parameters
- The Pipeline concept is inspired by the scikit-learn project

PREPROCESSING AND FEATURE EXTRACTION (I)

- **Tokenize**: A simple way to tokenize a string (sentence) in Python is using the split() method
  - Spark ML has 2 tokenizers: the (default) Tokenizer, which tokenizes on whitespace and RegexTokenizer which allows to specify a custom pattern to tokenize on other than white space. The latter is more flexible
- **N-grams**: Instead of looking at just single words, it is also useful to look at n-grams, the n-word long sequences
- **Remove stopwords**: omit certain common words which bear no meaning of documents like "a", "an", and "the"
  - Spark ML contains a standard list of such stop words for English. One can include any custom stopwords, if need be
- **Stemming**: It would have been useful to identify words like "computer" and "computers" as one word. The process of replacing them by a common root, or **stem**, is called stemming - the stem will not, in general, be a full word itself

PREPROCESSING AND FEATURE EXTRACTION (II)

- **Feature hashing**: is a way of vectorizing features, E.g. turning text features into indices in a vector or matrix. It works by applying a hash function to the features and using their hash values as indices
  - A hash function takes a string as an input and spits out a number, with the desired property being that different inputs usually produce different outputs (no hash collisions)
- **TF-IDF weighting**: (term-frequency inverse document frequency) is a feature vectorization method in text mining to reflect importance of a term *t* to a document *d* in corpus *d*

$$IDF(t,D) = \log\frac{D+1}{DF(t,D)+1}$$

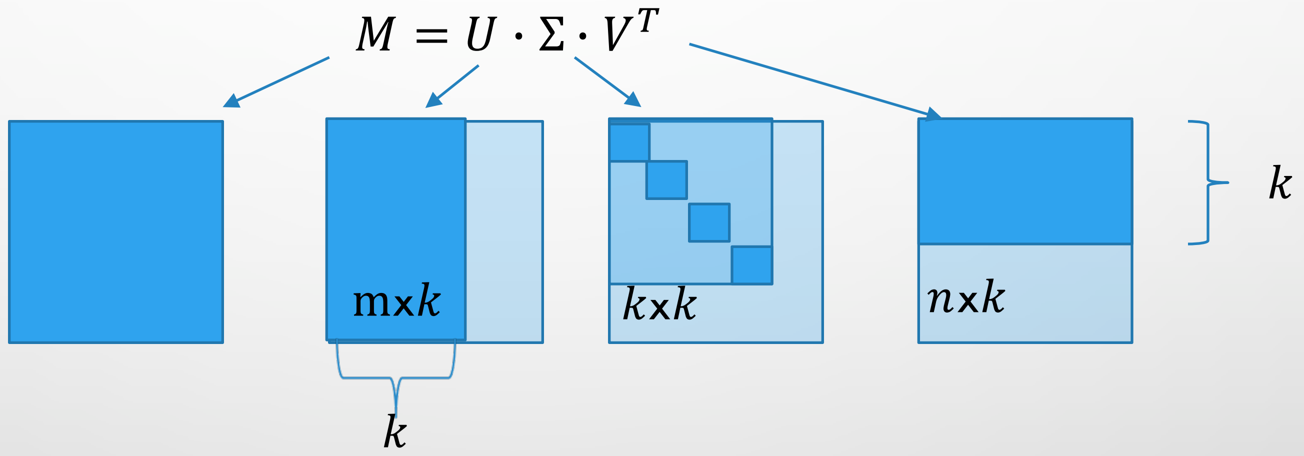
$$TFIDF(t,d,D) = TF(t,d) \bullet IDF(t,D)$$

DIMENSION REDUCTION: TRUNCATED SVD

- SVD is applied to the TF-IDF document-feature matrix to perform semantic decomposition and extract concepts which are most relevant for classification
- SVD factorizes the document-feature matrix M $(\mathrm{m} \times n)$ into three matrices: U, Σ, and V, such that:

$$M = U \cdot \Sigma \cdot V^T$$

$$\mathrm{m} \gg n \gg k$$



Here m is the number of legislative bills (order of 10^6), k is the number of concepts, and n is the number of features (2^{14})

- The left singular matrix U is represented as distributed row-matrix, while Σ and V are sufficiently small to fit into Spark driver memory

ALL-PAIRS SIMILARITY (I)

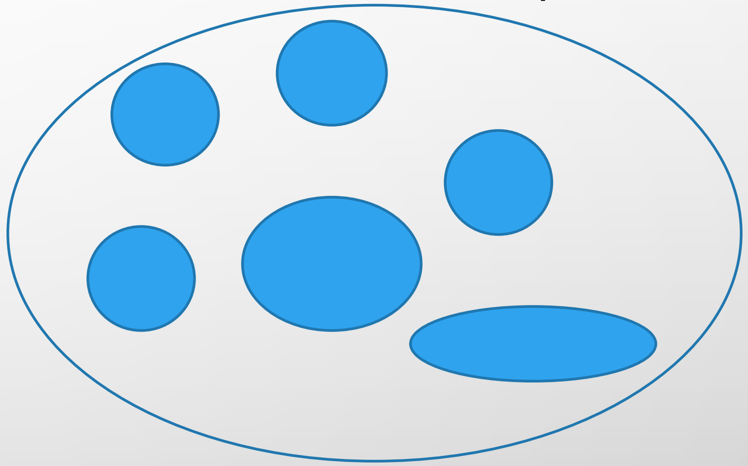
- Previous work in policy diffusion has been unable to make an all-pairs comparison between bills for a lengthy time period because of computational intensity
  - Brute-force all-pairs calculation between the texts of the state bills requires calculating a cross-join, yielding O(10^{13}) distinct pairs
  - As a substitute, scholars studied single topic areas
- Focusing on the document vectors which are likely to be highly similar is essential for all-pairs comparison at scale
- Modern studies employ variations of nearest-neighbor search, locality sensitive hashing (LSH), as well as sampling techniques to select a subset of rows of TF-IDF matrix based on the sparsity (DIMSUM)
- Our approach utilizes K-means clustering (details on the next slide)
  - We plan to include MinHash LSH in future

ALL-PAIRS SIMILARITY (II)

- Our approach utilizes K-means clustering to identify groups of documents which are likely to belong to the same diffusion topic, reducing the number of comparisons in the all-pairs similarity join calculation

Brute-force: just calculate all combinatorial pairs in the dataset

With K-means: first, split the dataset into K clusters (buckets), next only calculate all combinatorial pairs within each cluster

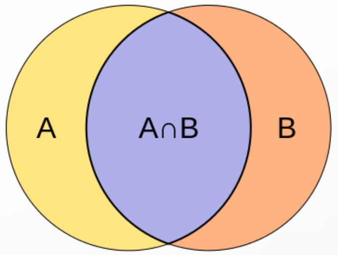


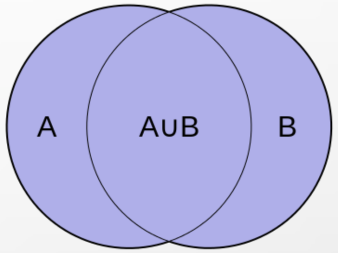
- Use within-set sum of squared errors as an objective of optimization (cost function)
- Tests show 40 iterations is necessary to get meaningful result
- Use 150 clusters for 3 states, 450 clusters for 50 states

SIMILARITY MEASURES: JACCARD

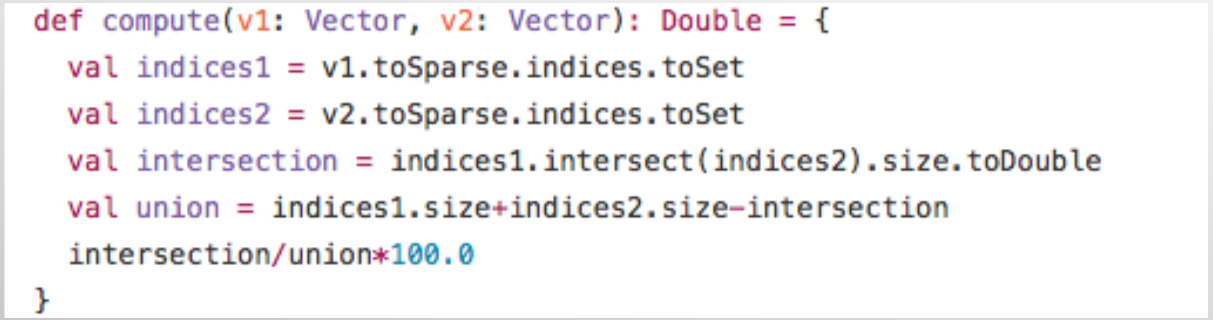
- Jaccard distance between sets:

$$D(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$



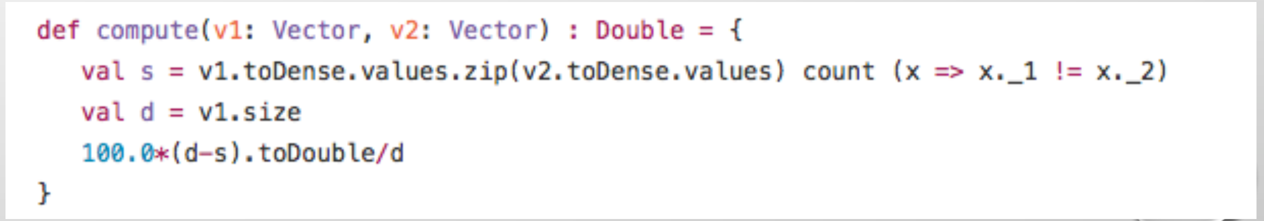


Key distance: consider feature vectors as sets of indices



```
def compute(v1: Vector, v2: Vector): Double = {
  val indices1 = v1.toSparse.indices.toSet
  val indices2 = v2.toSparse.indices.toSet
  val intersection = indices1.intersect(indices2).size.toDouble
  val union = indices1.size+indices2.size-intersection
  intersection/union*100.0
}
```

Hash distance: consider dense vector representations. Suitable when vectors are generated by hashing (like MinHash):



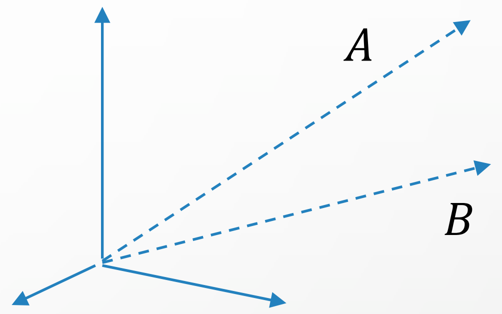
```
def compute(v1: Vector, v2: Vector) : Double = {
   val s = v1.toDense.values.zip(v2.toDense.values) count (x => x._1 != x._2)
   val d = v1.size
   100.0*(d-s).toDouble/d
}
```

- Best results achieved for TF features with a relatively large N-gram granularity and key-distance

SIMILARITY MEASURES: COSINE

- Cosine distance between feature vectors

$$D(A,B) = \frac{(A \cdot B)}{|A| \cdot |B|}$$



- We convert all distances (D) to similarities (S) assuming inverse proportionality, and rescaling it to [0,100] range and add a regularization term:

$$S = \frac{100}{1 + D}$$

- Similarity calculation between feature vectors reaches into Spark's private linear algebra code, to use BLAS dot product
- Best results achieved for TF-IDF features with truncated SVD applied, a relatively small N-gram granularity or a unigram

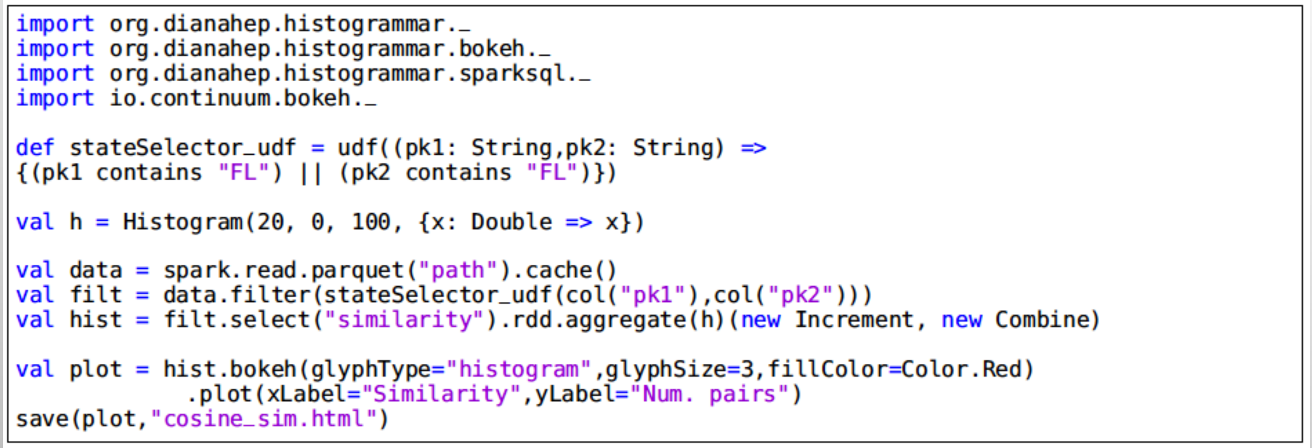
INTERACTIVE ANALYSIS WITH HISTOGRAMMAR TOOL

- Histogrammar is a suite of composable aggregators with
  - A language-independent specification
  - Several language versions (Python and Scala are the most complete)
  - An interchangeable JSON format
  - Multiple filling back-ends (examples follow),
  - No built-in plotting, but has Matplotlib, Bokeh front-ends

h·g

http://histogrammar.org/docs/

Contact Jim Pivarski
or me to contribute!



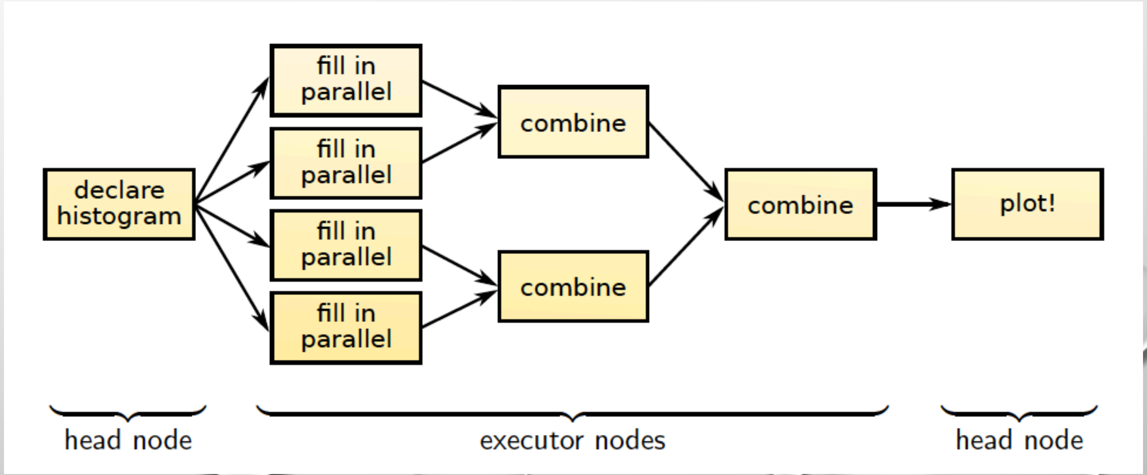
```
import org.dianahep.histogrammar._
import org.dianahep.histogrammar.bokeh._
import org.dianahep.histogrammar.sparksql._
import io.continuum.bokeh._

def stateSelector_udf = udf((pk1: String,pk2: String) =>
{(pk1 contains "FL") || (pk2 contains "FL")})

val h = Histogram(20, 0, 100, {x: Double => x})

val data = spark.read.parquet("path").cache()
val filt = data.filter(stateSelector_udf(col("pk1"),col("pk2")))
val hist = filt.select("similarity").rdd.aggregate(h)(new Increment, new Combine)

val plot = hist.bokeh(glyphType="histogram",glyphSize=3,fillColor=Color.Red)
            .plot(xLabel="Similarity",yLabel="Num. pairs")
save(plot,"cosine_sim.html")
```

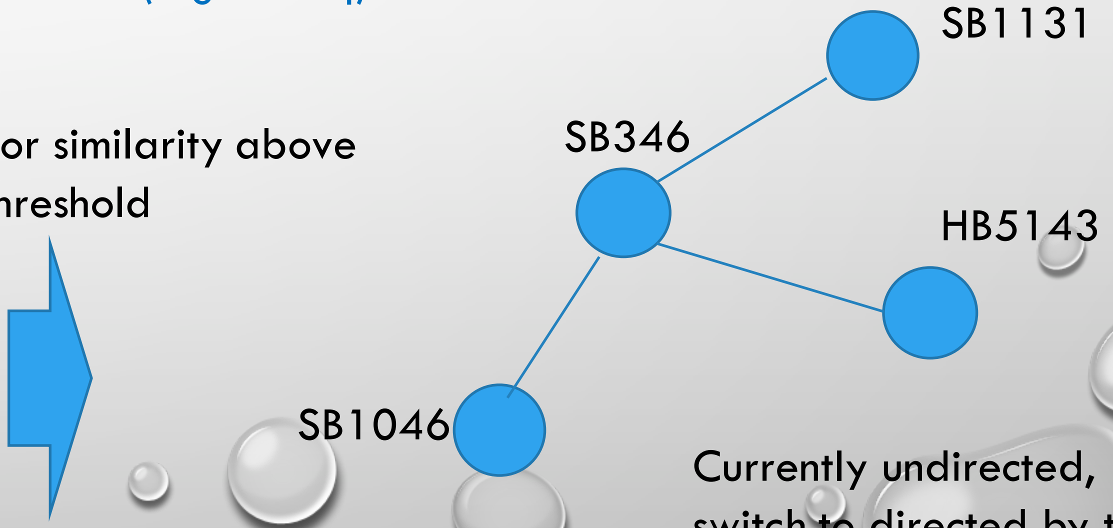


GRAPHFRAMES

- GraphFrames is an extension of Spark allowing to perform graph queries and graph algorithms on Spark dataframes
  - A GraphFrame is constructed using two dataframes (a dataframe of nodes and an edge dataframe), allowing to easily integrate the graph processing step into the pipeline along with Spark ML
  - Graph queries: like a Cypher query on a graph database (e.g. Neo4j)
  - Graph algorithms: PageRank, Dijkstra
  - Possibility to eliminate joins

bill1	bill2	similarity
FL/2005/SB436	MI/2005/SB1046	91.38
FL/2005/SB436	MI/2005/HB5143	91.29
FL/2005/SB436	SC/2005/SB1131	82.89

For similarity above threshold



Currently undirected, switch to directed by time

APPLICATIONS OF POLICY DIFFUSION DETECTION TOOL

- The policy diffusion detection tool can be used in a number of modes:
  - Identification of groups of diffused bills in the dataset

    given a diffusion topic (for instance, “Stand your ground” policy, cyberstalking, marijuana laws etc)
  - Discovery of diffusion topics: consider top-similarity bills within each cluster, careful filtering of uniform bills and interstate compact bills is necessary as they would show high similarity as well
  - Identification of minimum cost paths connecting two specific legislative proposals on a graph
  - Identification of the most influential US states for policy diffusion
- The research paper on applications of the tool is currently in progress

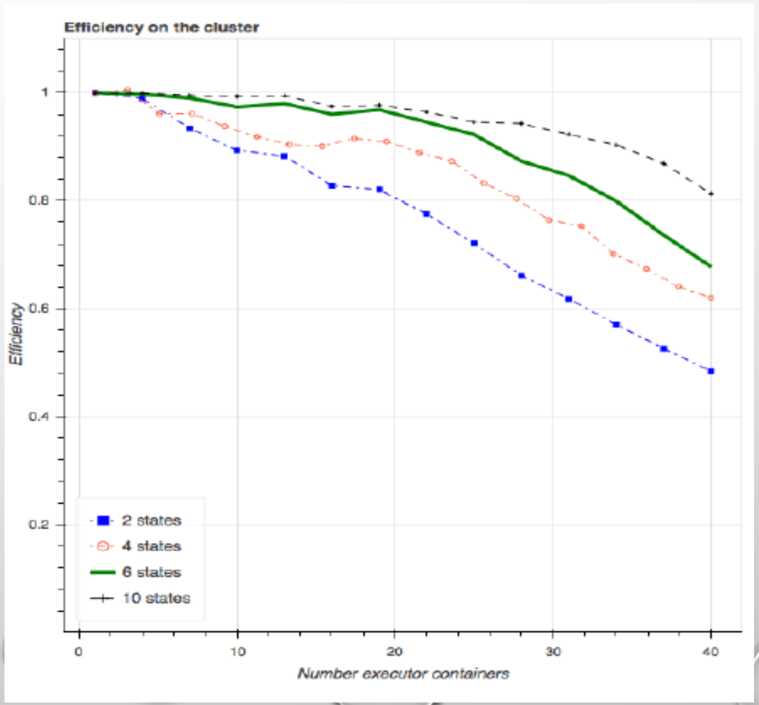
PERFORMANCE SUMMARY

- Spark applications have been deployed on Hadoop cluster using YARN
  - 40 executor containers, each using 3 executor cores and 15 GB of RAM per JVM
  - Use external shuffle service inside the YARN node manager to improve stability of memory-intensive jobs with larger number of executor containers
- Calculate efficiency of parallel execution as

$$E = \frac{T_0}{N_{exec} \cdot T_N}$$

Single executor case

Time using N_{exec} executors



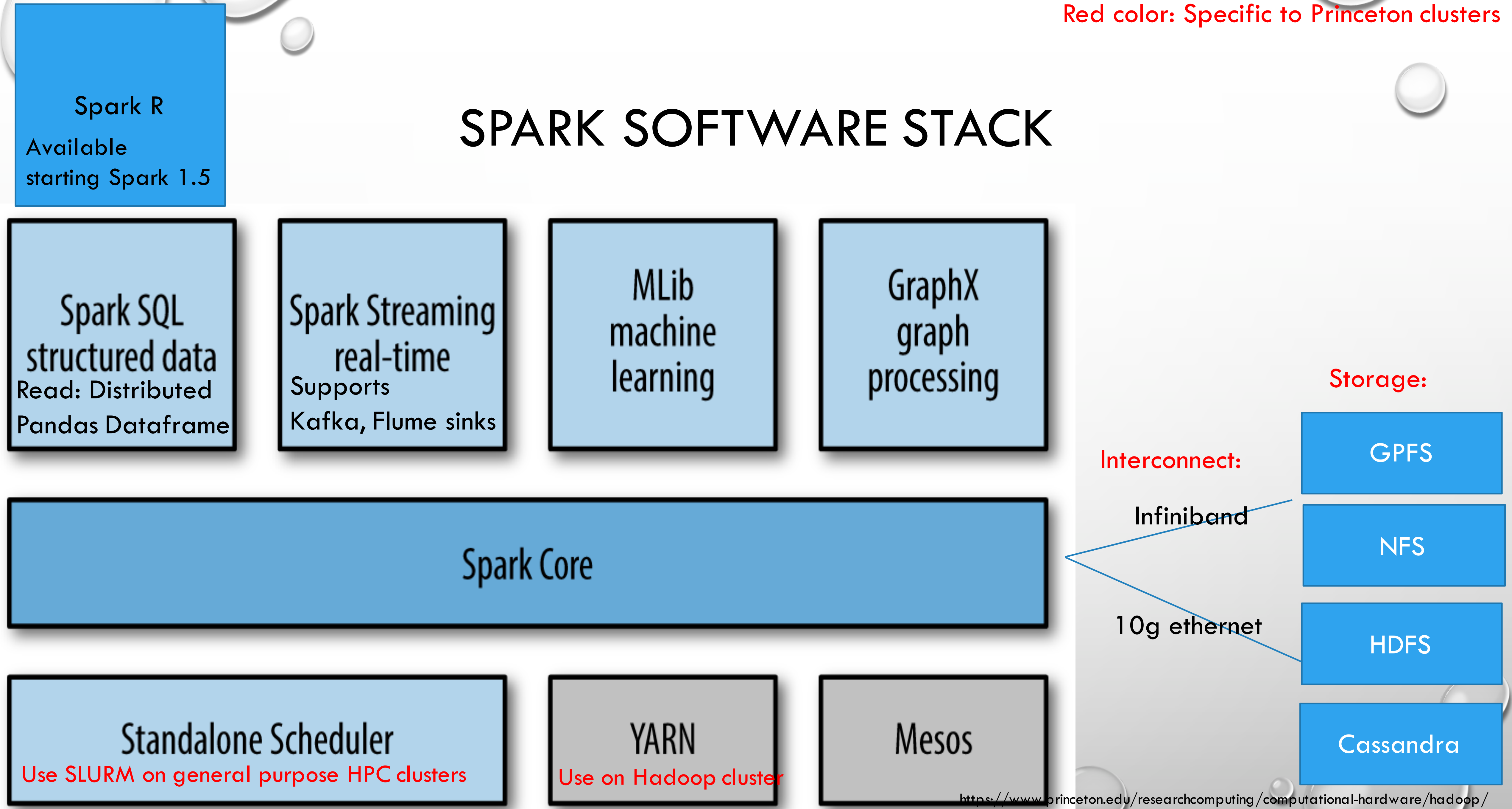
CONCLUSIONS

- Evaluated Apache Spark framework for the case of data-intensive machine learning problem of policy diffusion detection.
  - Provided a scalable method to calculate all-pairs similarity based on K-means clustering
  - Implemented a text processing pipeline utilizing Apache Avro serialization framework, Spark ML, GraphFrames, and Histogrammar suite of data aggregation primitives
  - Efficiently calculate all-pairs comparison between legislative bills, estimate relationships between bills on a graph, potentially applicable to a wider class of fundamental text mining problems of finding similar items
- Tuned Spark internals (partitioning, shuffle) to obtain good scaling up to O(100) processing cores, yielding 80% parallel efficiency
- Utilized Histogrammar tool as a part of the framework to enable interactive analysis, allows a researcher to perform analysis in Scala language, integrating well with Hadoop ecosystem

BACKUP SLIDES

Red color: Specific to Princeton clusters

SPARK SOFTWARE STACK

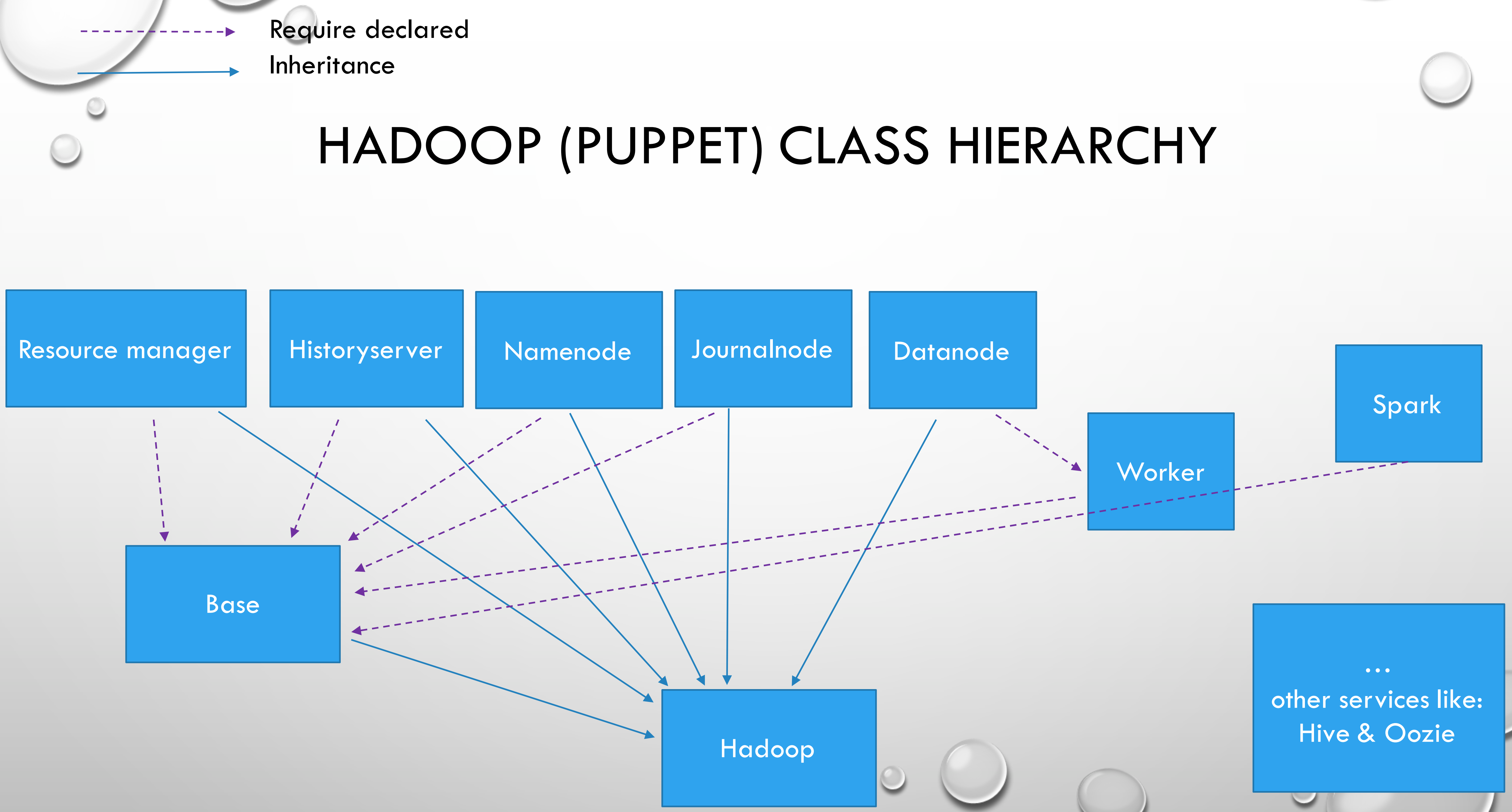

Spark R
Available starting Spark 1.5
Spark SQL structured data
Read: Distributed Pandas Dataframe
Spark Streaming real-time
Supports Kafka, Flume sinks
MLib machine learning
GraphX graph processing
Spark Core
Standalone Scheduler
Use SLURM on general purpose HPC clusters
YARN
Use on Hadoop cluster
Mesos
Interconnect:
Infiniband
10g ethernet
Storage:
GPFS
NFS
HDFS
Cassandra

https://www.princeton.edu/researchcomputing/computational-hardware/hadoop/
https://www.princeton.edu/researchcomputing/computational-hardware/machine-2/

HADOOP CLUSTER COMPONENTS

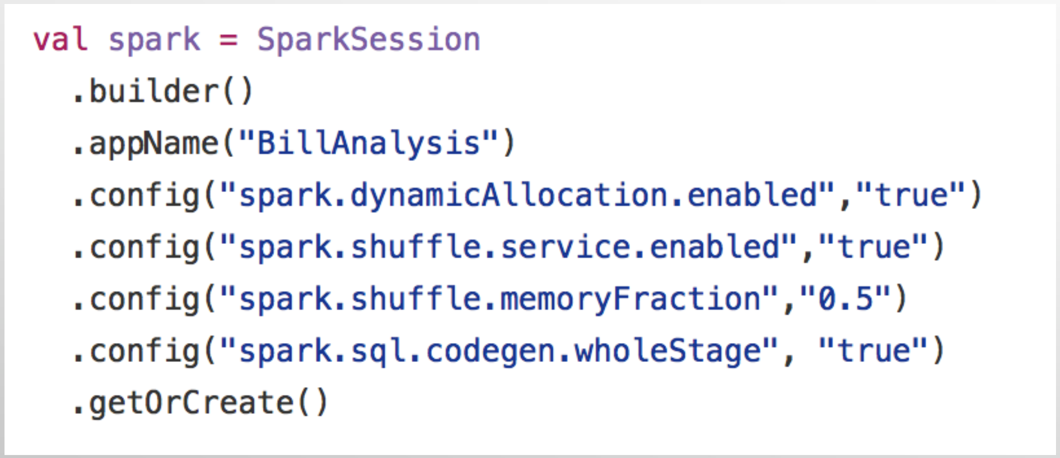
- RESOURCE MANAGER – THE YARN RESOURCE MANAGER (RM) IS RESPONSIBLE FOR TRACKING THE RESOURCES IN A CLUSTER AND SCHEDULING APPLICATIONS (FOR EXAMPLE, SPARK JOBS)
- (JOB) HISTORY SERVER – YARN REQUIRES A STAGING DIRECTORY FOR TEMPORARY FILES CREATED WHILE RUNNING JOBS. ALSO, DIRECTORY FOR LOG FILES OF FINISHED JOBS
- NAMENODES – HADOOP CLUSTER FOLLOWS A MASTER-WORKER PATTERN, HERE NAMENODE IS THE MASTER TO DATANODES (WORKERS). NAMENODES MANAGE THE FILESYSTEM TREE AND METADATA. IN THE HIGH AVAILABILITY CONFIGURATION, 2 OR MORE NAMENODES IS RAN ON THE SAME CLUSTER, IN AN ACTIVE/PASSIVE CONFIGURATION. THESE ARE REFERRED TO AS THE **ACTIVE** NAMENODE AND THE **STANDBY** NAMENODE(S).
- JOURNAL NODES - IN ORDER FOR A STANDBY NODE TO KEEP ITS STATE SYNCHRONIZED WITH THE ACTIVE NODE IN THIS IMPLEMENTATION, BOTH NODES COMMUNICATE WITH A GROUP OF SEPARATE DAEMONS CALLED **JOURNALNODES**. WHEN ANY FILESYSTEM MODIFICATION IS PERFORMED BY THE ACTIVE NODE, IT LOGS A RECORD OF THE MODIFICATION TO A MAJORITY OF THESE JOURNALNODES. THE STANDBY NODE IS CAPABLE OF READING THE EDITS FROM THE JOURNALNODES, AND IS CONSTANTLY WATCHING THEM FOR CHANGES TO THE EDIT LOG. AS THE STANDBY NODE SEES THE EDITS, IT APPLIES THEM TO ITS OWN NAMESPACE. IN THE EVENT OF A FAILOVER, THE STANDBY WILL ENSURE THAT IT HAS READ ALL OF THE EDITS FROM THE JOURNALNODES BEFORE PROMOTING ITSELF TO THE ACTIVE STATE.
- DATANODE – WORKER NODE. DAEMONS RUNNING ON DATANODES INCLUDE JOB TRACKER AND HDFS
- OTHER (OPTIONAL) CLUSTER COMPONENTS INCLUDE: SPARK, HIVE, OOZIE, HBASE, HUE…

HADOOP (PUPPET) CLASS HIERARCHY

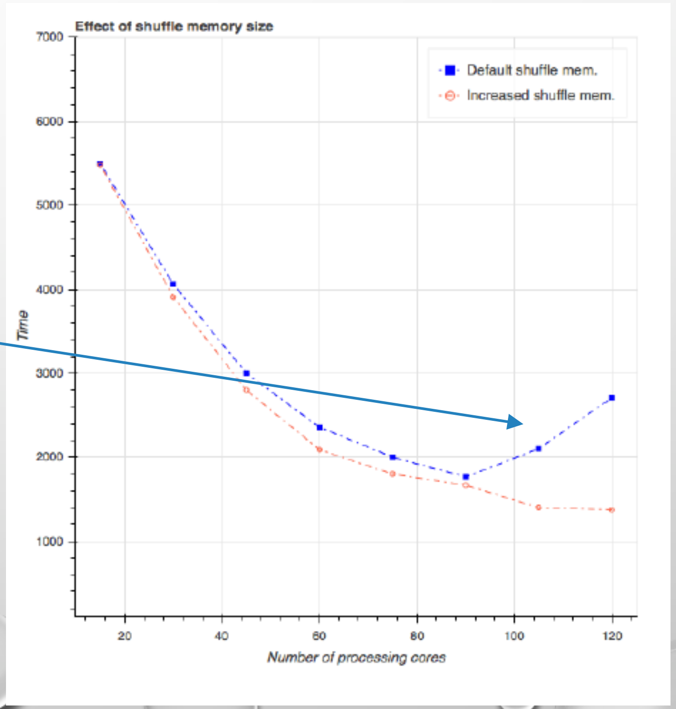


SHUFFLE

- An intensive shuffle across partitions of the dataset has been identified as the main cause of efficiency decrease
  - Adjusting the fraction of Java heap space used during shuffles allowed to maintain a good scaling beyond 90 processing cores



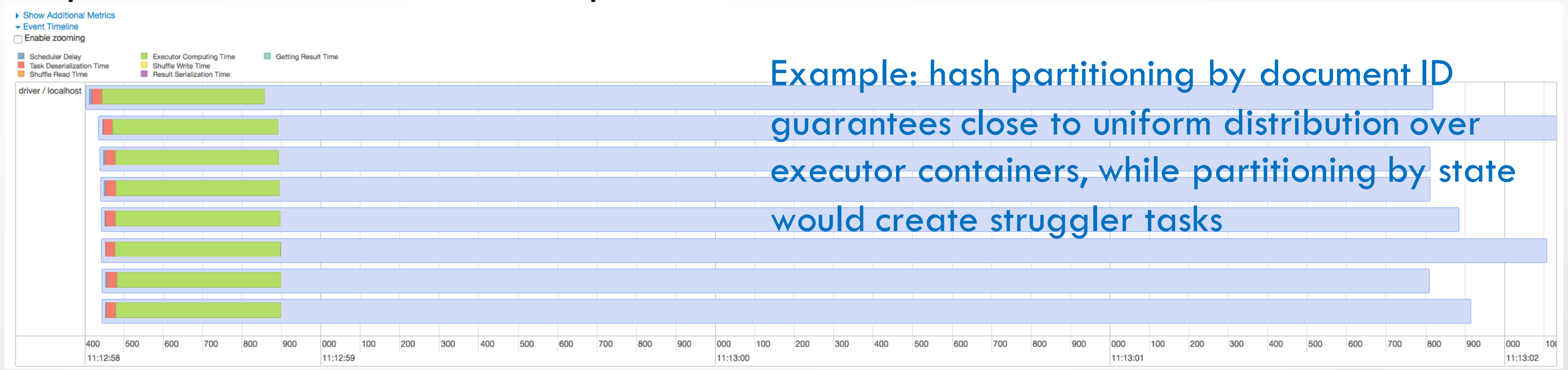
```
val spark = SparkSession
  .builder()
  .appName("BillAnalysis")
  .config("spark.dynamicAllocation.enabled","true")
  .config("spark.shuffle.service.enabled","true")
  .config("spark.shuffle.memoryFraction","0.5")
  .config("spark.sql.codegen.wholeStage", "true")
  .getOrCreate()
```



PERFORMANCE TUNING FOR SPARK APPLICATIONS: PARTITIONING

- Partition in Spark is a unit of parallel execution that corresponds to one task

The number of tasks that can be executed concurrently is limited by the total number of executor cores in the Spark cluster



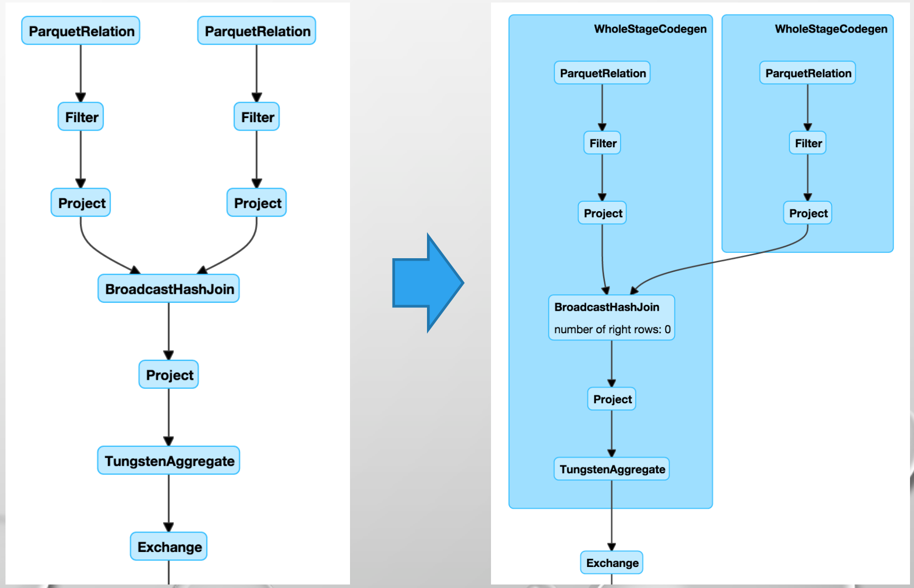
Example: hash partitioning by document ID guarantees close to uniform distribution over executor containers, while partitioning by state would create struggler tasks

- The Partitioner object defines a function from an element of a pair RDD to a partition via a mapping from each record to partition number (Examples: HashPartitioner, RangePartitioner)
  - By assigning a partitioner to an RDD we can control how data is distributed across the nodes on a cluster and minimize shuffle during joins
  - repartition and coalesce transformations are the ways to control the number of partitions, while the Partitioner object controls “how”
  - coalesce is a narrow transformation if the number of partitions is reduced: no shuffle!

WHY DATAFRAMES?

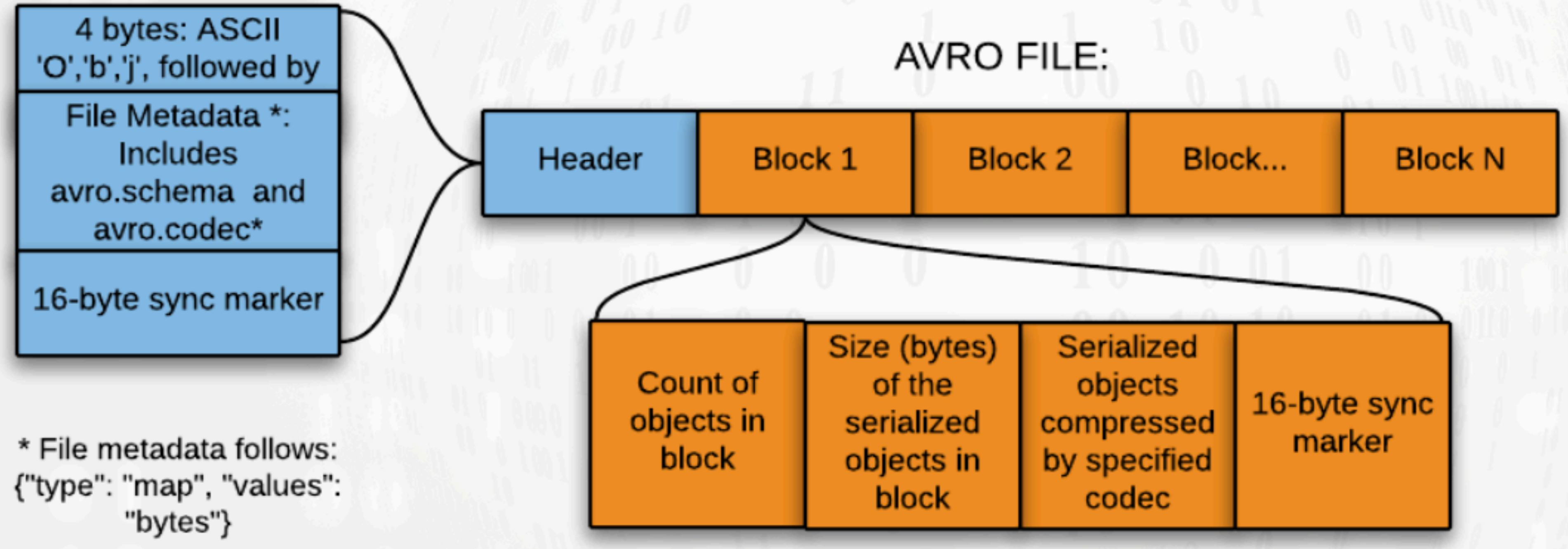
- Spark SQL and its DataFrames are essential for Spark performance with more efficient storage options, advanced optimizer, and direct operations on serialized data
  - Introduced in Spark 1.3 (used to be called schema RDD and inherited directly from RDD)
- Like RDDs, DataFrames represent distributed collections, with additional schema information not found in RDDs
  - This additional schema information is used to provide a more efficient storage layer and in the optimizer
- Compared to working with RDDs, DataFrames allow Spark's Catalyst optimizer to better understand our code and our data

Example: Planner and Catalyst

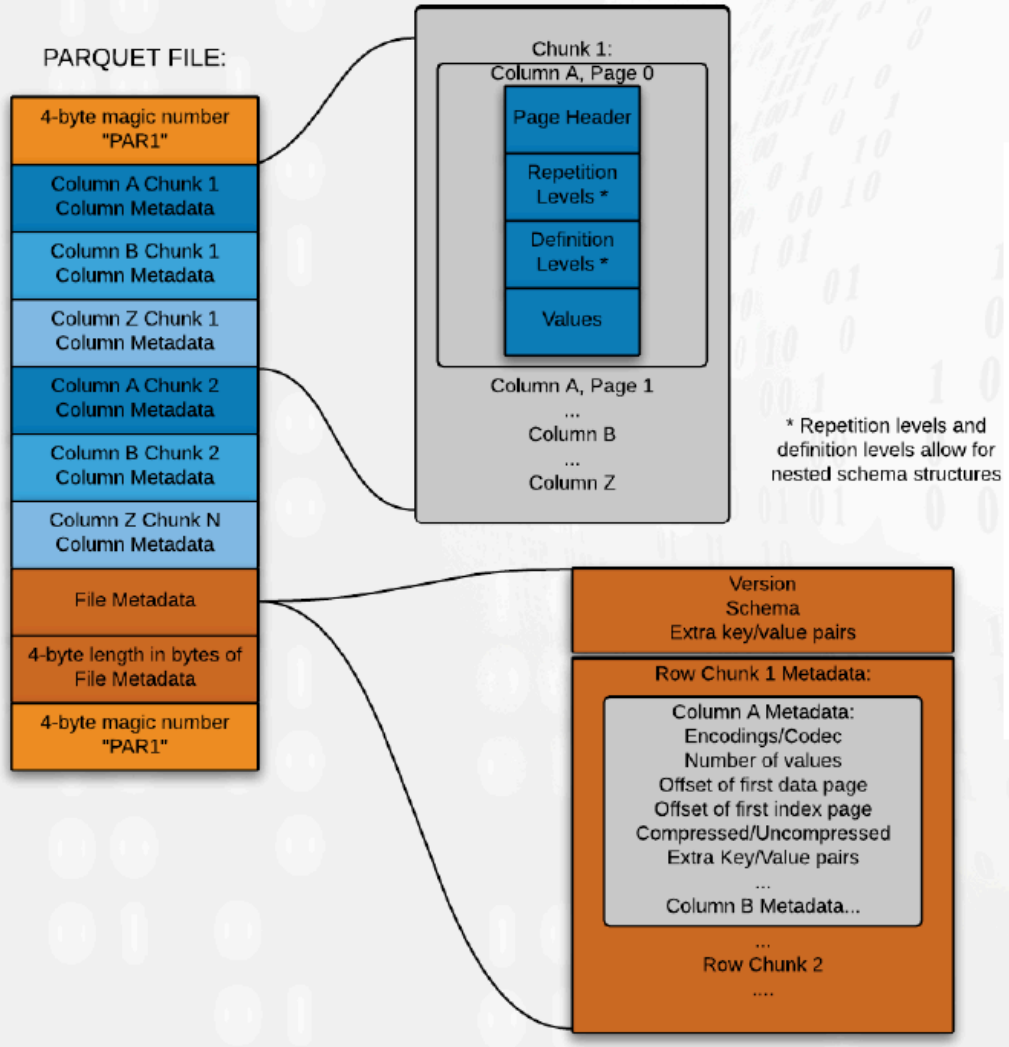


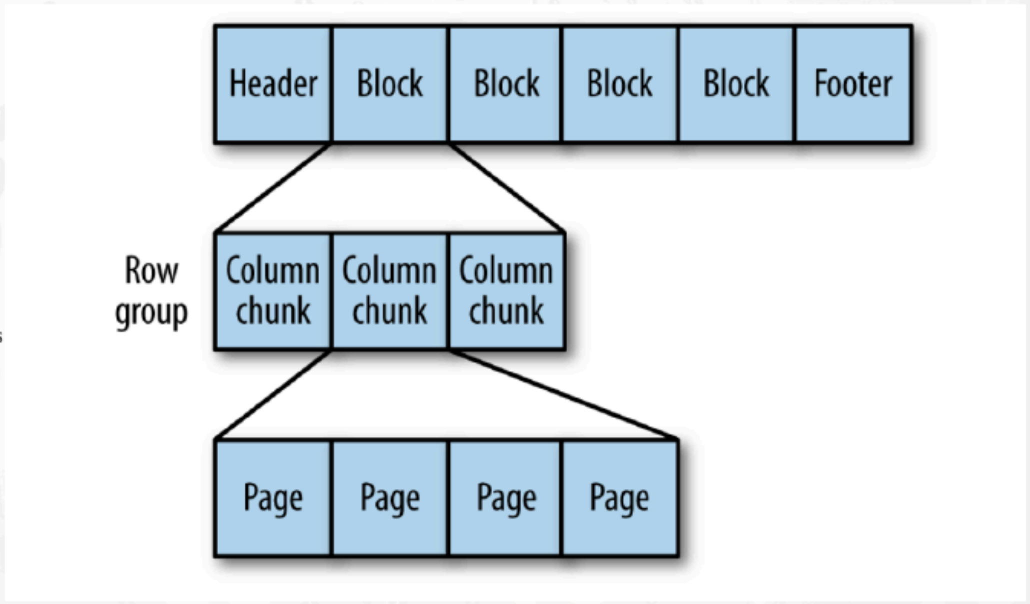
Based on "High performance Spark" by H. Karau et al and personal experience

File Structure - Avro


4 bytes: ASCII 'O','b','j', followed by
File Metadata *: Includes avro.schema and avro.codec*
16-byte sync marker
* File metadata follows: {"type": "map", "values": "bytes"}
AVRO FILE:
Header
Block 1
Block 2
Block...
Block N
Count of objects in block
Size (bytes) of the serialized objects in block
Serialized objects compressed by specified codec
16-byte sync marker

File Structures - Parquet


PARQUET FILE:
4-byte magic number "PAR1"
Column A Chunk 1 Column Metadata
Column B Chunk 1 Column Metadata
Column Z Chunk 1 Column Metadata
Column A Chunk 2 Column Metadata
Column B Chunk 2 Column Metadata
Column Z Chunk N Column Metadata
File Metadata
4-byte length in bytes of File Metadata
4-byte magic number "PAR1"
Chunk 1:
Column A, Page 0
Page Header
Repetition Levels *
Definition Levels *
Values
Column A, Page 1
...
Column B
...
Column Z
* Repetition levels and definition levels allow for nested schema structures
Version
Schema
Extra key/value pairs
Row Chunk 1 Metadata:
Column A Metadata:
Encodings/Codec
Number of values
Offset of first data page
Offset of first index page
Compressed/Uncompressed
Extra Key/Value pairs
...
Column B Metadata...
...
Row Chunk 2
....



Parquet