Large-scale text processing pipeline with Apache Spark

A. Svyatkovskiy, K. Imai, M. Kroeger, Y. Shiraito
Princeton University

***Abstract*—In this paper, we evaluate Apache Spark for a data-intensive machine learning problem. Our use case focuses on policy diffusion detection across the state legislatures in the United States over time. Previous work on policy diffusion has been unable to make an all-pairs comparison between bills due to computational intensity. As a substitute, scholars have studied single topic areas.**

We provide an implementation of this analysis workflow as a distributed text processing pipeline with Spark dataframes and Scala application programming interface. We discuss the challenges and strategies of unstructured data processing, data formats for storage and efficient access, and graph processing at scale.

***Index Terms*—Spark, Avro, Spark ML, Spark GraphFrames.**

I. Introduction

POLICY diffusion occurs when government decisions in a given jurisdiction are systematically influenced by prior policy choices made in other jurisdictions [1]. While policy diffusion can manifest in a variety of forms, we focus on a type of policy diffusion that can be detected by examining similarity of legislative bill texts. Our dataset is based on the LexisNexis StateNet [2] and contains a total of more than 7 million legislative bills from 50 US states from 1991 to 2016. We aim to identify groups of legislative bills from different states falling into the same diffusion topic, to perform an all-pairs comparison between the bills within each topic, and to identify paths connecting specific legislative proposals on a graph.

The causes and the extent to which policies spread across state legislatures is of substantive importance to political scientists, with implications for the states as laboratories of democracy [3–7]. Previous work has been unable to make an all-pairs comparison between bills for a lengthy time period, as we do in this paper, because of computational intensity: a brute-force all-pairs calculation between the texts of the state bills yields $O(10^{13})$ distinct pairs. As a substitute, scholars have studied single topic areas [6, 8–11], however, these areas are also the most likely to diffuse and thus tell us little about the extent to which law traverses across state borders.

Our analysis pipeline, which is summarized in Fig. 1, consists of the following five stages: (1) data ingestion, (2) pre-processing and feature extraction, (3) candidate document selection, (4) document pair similarity calculation and (5) policy diffusion detection by ranking or on a graph. Apache

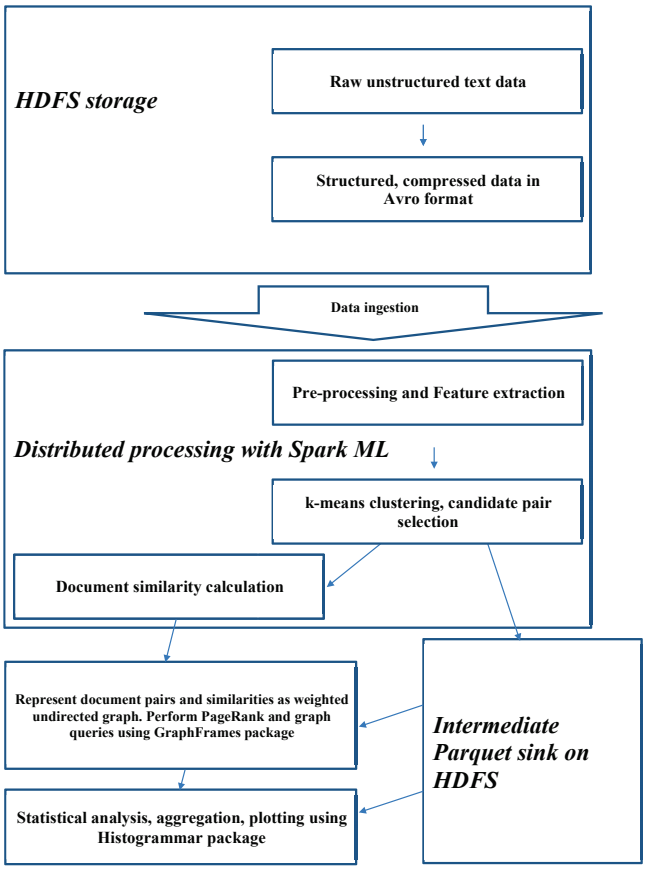

Fig. 1. The policy diffusion analysis pipeline.

Spark [12], a high-performance distributed computing framework for large-scale data processing, is at the core of the pipeline implementation. Spark uses directed acyclic graph (DAG) instead of MapReduce execution engine, allowing to process multi-stage pipelines chained in one job. It is closely integrated with Apache Hadoop ecosystem and can run on YARN, use Hadoop file formats, and HDFS storage [13]. Spark provides an ability to cache large datasets in memory between stages of the calculation, allowing to reuse intermediate results of the computation in iterative algorithms like k-means clustering and improves fault tolerance by taking advantage of

Professor, Department of Politics and Center for Statistics and Machine Learning, Princeton University

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

the data replication in HDFS and check-pointing.

The framework is applicable to a wider class of fundamental text mining problems of finding similar items, including plagiarism and mirror website detection [14].

The implementation of the pipeline uses a mix of RDD based API along with the dataframes, allowing to take advantage of the Catalyst query optimizer and direct operations on serialized data, available in Spark 2.0.0 [15]. We use GraphFrames [16] for dataframe-based graph algorithms calculation and graph queries.

Scala is a high-level programming language for the Java virtual machine (JVM). We chose Scala over Python or R programming languages — commonly considered a standard in text analysis research communities — because the optimal performance of Spark is most likely to be achieved in that language. Spark code written in Python or R is often slower than equivalent code written in Scala, since it is statically typed and the cost of JVM communication from Python to Scala can be very high.

Big data applications require a mix of processing techniques, data sources and storage formats. We chose Apache Avro row-based format and Apache Parquet columnar format for this pipeline instead of data formats like CSV and JSON, commonly used by scholars in this area, to take advantage of advanced compression and serialization, crucial for big data applications implemented with Spark.

Interactive statistical analysis tools compatible with Spark have been implemented in the Histogrammar package [17]: a cross-platform suite of data aggregation primitives for making histograms, calculating descriptive statistics and plotting. Histogrammar allows to aggregate data using functional primitives, summarizing a large dataset with discretized distributions, using lambda functions and composition rather than a restrictive set of histogram types.

We observe stable execution of memory-intensive text processing jobs with large number of executor containers yielding efficiencies greater than 80% for the largest dataset considered in the study.

The paper is organized as follows: we start with the hardware specifications and details on the Spark cluster setup and Hadoop ecosystem; in the Section III the policy diffusion detection method and the core modules of the pipeline are described; Section IV introduces the Histogrammar tool for interactive data aggregation and plotting applied to the policy diffusion problem; finally, Section V summarizes the performance of the core modules of the pipeline and discusses the optimization. Section VI concludes the paper.

II. Hardware specifications

AN SGI Hadoop Linux cluster consisting of 6 worker nodes and 4 service nodes is used to deploy Apache Spark. The cluster is configured with all the servers mounted on one rack and interconnected using a 10 Gigabit Ethernet switch. Intel Xeon CPU E5-2680 v2 @ 2.80GHz CPU processors, with each worker node having 256 GB of memory and 10 TB of hard disk allow to achieve high performance and handle distributed workloads.

The Hadoop cluster is configured without single points of failure by using two separate machines as name nodes. Spark applications are scheduled using YARN resource manager with dynamic resource allocation enabled. HDFS distributed file system is chosen to improve data locality by means of replication.

III. Text processing pipeline: core modules

THIS section discusses the core modules of the text processing pipeline for policy diffusion detection.

A. Data ingestion

During ingestion step the raw unstructured data are converted into Apache Avro format having following schema:

```
{"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"}
 ]
}
```

where the $primary\_key$ field is a unique identifier of the elements in the dataset constructed from year, state and document version. The $year$, $state$ and $docversion$ fields are used to construct predicates and filter the data before the all-pairs similarity join calculation.

The $content$ field stores the entire legislative proposal as a unicode string. It is only used for feature extraction step, and is not read into memory during candidate selection and filtering steps, thanks to the Avro schema evolution property.

Avro schema is stored in a file along with the data. Thus, if the program reading the data expects a different schema this can be easily resolved by setting the $avro.input.schema.key$ in the Spark application, since the schemas of Avro writer and reader are both present.

B. Pre-processing and Feature extraction

The feature extraction step consists of a sequence of Spark ML transformers intended to produce numerical feature vectors as a dataframe column. The resulting dataframe is fed to Spark ML k-means estimator, later used to calculate the all-pairs join, and subsequently during the graph analysis step with GraphFrames.

1) Data cleaning and stop word removal: The raw text of legislative proposals from the StateNet dataset contains a lot of spurious white spaces and non-alphanumeric characters, which bare no meaning for analysis of legislative bills and often represent an obstacle for tokenization. The cleaner is implemented as a column-based user defined function (UDF).

The words appearing very frequently in all the documents across the corpus (stop words) are excluded by means of

StopWordsRemover transformer from Spark ML, which takes a dataframe column of unicode strings and drops all the stop words from the input. The default list of stop words for English language is used in this study.

2) Bag-of-words and the N-gram model: In the bag-of-words model, text is represented as a multiset of words, disregarding grammar and word order but keeping multiplicity. The N-gram model, on the other hand, preserves the spatial information about the order within the multiset. Conceptually, the bag-of-word model can be viewed as a special case of the N-gram model with $N = 1$.

We use a regular expression based tokenizer which produces a dataframe column having an array of strings per row. The *NGram* transformer from Spark ML takes a sequence of strings from the output of tokenizer and converts it to a sequence of space-delimited strings of N consecutive words, which are optionally added to the bag-of-word features to improve accuracy.

3) Term frequency and inverse document frequency calculation: Term frequency-inverse document frequency (TF-IDF) is a feature vectorization method used to reflect the importance of a term to a document in the corpus. TF-IDF is implemented in two classes in Spark ML.

HashingTF class implements a transformer, which takes tokenized documents and converts them into fixed-length feature vectors by means of the hashing trick. A raw feature is mapped to an index by applying the *MurmurHash 3* hash function. The *IDF* estimator is fit on feature vectors created from *HashingTF*. It down-weights columns which appear frequently in the corpus.

4) Dimensionality reduction: Singular value decomposition (SVD) is applied to the TF-IDF document-feature matrix to extract concepts which are most relevant for classification [18].

SVD factorizes the document-feature matrix M ($m \times n$) into three matrices U,Σ and V, such that:

$$M = U \cdot \Sigma \cdot V^T, \tag{1}$$

having $m \times k$, $k \times k$ and $k \times n$ dimensions correspondingly, where m is the number of legislative bills ($O(10^6)$), k is the number of concepts, and n is the number of features (2^{14}). Following inequalities hold:

$$m \gg n \gg k. \tag{2}$$

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

C. Candidate selection and clustering

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 [14], as well as sampling techniques to select a subset of rows of TF-IDF matrix based on the sparsity [19]. 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.

The $features$ dataframe column is passed to the *KMeans* estimator which generates *KMeansModel* with a given number of cluster centroids. k-means clustering subdivides N vectors in the feature space into k clusters so that each vector belongs to a cluster with the nearest centroid, used to initialize the cluster.

Given an initial set of k cluster centroids $m_i^{(0)}$, where $i = 0...N$ the algorithm yields:

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 , \forall j, 1 \leq j \leq k \right\}, \tag{3}$$

where each x_p is assigned to exactly one $S^{(t)}$ during the iteration t.

During the update step, the means of the clusters are assigned to be the new clusters centroids on the next iteration:

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \tag{4}$$

and the procedure is repeated before the convergence based on the within-cluster sum of squares (WCSS) objective is reached.

The optimum number of clusters has been determined empirically, by repeating the calculating for a range of values of k and scoring them on a processing time versus WCSS plane. While processing time has been increasing with k, the WCSS gain has appeared to slow down significantly in the neighborhood of $k \approx 150$ for a 3 state subset and $k \approx 400$ for the entire dataset.

1) Number of permutations: Requesting algorithm to focus on the combinatorial pairs belonging to the same clusters reduces the number of comparisons in the all-pairs similarity join by 2-3 orders of magnitude, keeping the bill pairs belonging to the same diffusion topics with high probability.

Indeed, starting with a total of $N = 212768$ legislative proposals in a 3 state subset of the dataset, we would get a total of: $N \cdot (N-1)/2 = 2.26 \times 10^{10}$ distinct combinatorial pairs to compare. Considering $k = 150$ classes for k-means clustering and assuming a uniform distribution of samples among these clusters we would get: $M = N/k = 212768/150 \approx 1418$, resulting in $M \cdot (M-1)/2 \cdot k = 1418 \cdot (1418-1)/2 \cdot 150 \approx 1.5 \times 10^8$ combinatorial pairs, which is roughly 2 orders of magnitude less compared to the case with no clustering. The actual distribution among k-means clusters for this sample has shown a mean occupancy of 1467.3 documents per cluster, with the standard deviation of 9562.4, and the maximum occupancy of 110794 documents per cluster, yielding a good reduction in the number of pairwise comparisons.

D. Document similarity calculation

The k-means clustering algorithm assigns vectors in the feature space to clusters by minimizing the WCSS objective. The step after that – all-pairs document similarity calculation – is performed within each cluster. Cosine, Jaccard, manhattan and Hamming similarity measures are considered.

The *SimilarityMeasure* trait provides a common interface for similarity calculation between feature vectors reaching into Spark's private linear algebra code to use *BLAS* dot product.

Each similarity measure is implemented as an object extending the *SimilarityMeasure* class and each implementing its own $compute$ method for dot product.

We convert Cosine, manhattan and Hamming distances to similarities assuming inverse proportionality, and re-scale all similarities to a common range:

$$S_M = \frac{100}{1 + D_M} \tag{5}$$

an extra additive term in the denominator serves as a regularization parameter for the case of identical vectors.

E. Policy diffusion detection

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,
- discovery of diffusion topics,
- identification of minimum cost paths connecting two specific legislative proposals on a graph, and, possibly,
- identification of the most influential US states for policy diffusion.

We use supervised pre-training on a set of diffusion topics labeled by an expert to tune the classification algorithm to achieve a better accuracy.

Below is an example output of the classifier for the test performed on a subset of legislative proposals having "Stand your ground" diffusion topic.

Input: A set of bills on the topic: FL/2005/SB436, MI/2005/HB5153, MI/2005/HB5143, SC/2005/HB4301, MI/2005/SB1046, and a probe bill: FL/2005/SB436.

Output: A set of top similarity bills from the test set contained all the samples labeled as having "Stand your ground" diffusion topic by an expert:

```
FL/2005/SB436, MI/2005/SB1046: 91.38,
FL/2005/SB436, MI/2005/HB5143: 91.29,
FL/2005/SB436, MI/2005/HB5153: 91.18,
FL/2005/SB436, SC/2005/SB1131: 82.89,
FL/2005/SB436, SC/2005/HB4301: 81.86,
FL/2005/SB436, SC/2011/SB1415: 77.11.
```

F. Reformulating the problem as a network (graph) problem

Some policy diffusion questions are easier answered if the problem is formulated as a graph analysis problem. The dataframe output of the document similarity step is mapped onto a weighted undirected graph, considering each unique legislative proposal as a node and a presence of a document with similarity above a certain threshold as an edge with a weight attribute equal to the similarity. The PageRank and Dijkstra minimum cost path algorithms are applied to detect events of policy diffusion and the most influential states.

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, without a need to move the results of previous steps manually and feeding them to the graph processing module from an intermediate sink, like with isolated graph analysis systems.

IV. Interactive analysis

THIS section describes the tools and techniques used for interactive part of the analysis in read-eval-print loop (REPL) shell.

Histogrammar [17] is a suite of data aggregation primitives for making histograms, calculating descriptive statistics and plotting. A few composable functions can generate many different types of plots, and these functions are reimplemented in multiple languages and serialized to JSON for cross-platform compatibility. Histogrammar allows to aggregate data using cross-platform, functional primitives, summarizing a large dataset with discretized distributions, using lambda functions and composition rather than a restrictive set of histogram types.

Histogrammar primitives are order-independent commutative monoids designed for distributed computing and cross-platform compatibility. As a data analyst, you just express your data aggregation in terms of nested Histogrammar primitives and pass it to any system for evaluation. Since all of the logic of what to fill is encoded in your lambda functions, the aggregation phase is automatic.

Moving the logic of data analysis out of the for loop allows the analyst to describe an entire analysis declaratively. A whole analysis can be wrapped up in subdirectories like

```
Label(
   dir1 = Label(
      hist1 = Bin(...),
      hist2 = Bin(...)),
   dir2 = ...)
```

This tree gets filled the same way as a single histogram, because the Label collection is a primitive just like Bin.

Thus, analysis code can be independent of where the data are analyzed. This is especially helpful for aggregating data in hard to reach places: across a distributed system like Apache Spark, on a GPU coprocessor, or through a thin bandwidth connection.

Histogrammar has front-end extensions to pass its aggregated data to many different plotting libraries, including Bokeh and Matplotlib.

Histogrammar also has back-end extensions for aggregating data from different frameworks. It can therefore be thought of as a common language for aggregating and then plotting data, so that every plotting library doesn't have to have individual hooks for every kind of aggregation system.

An example interactive analysis in spark-shell REPL is provided in Appendix A.

V. Performance evaluation

THIS section discusses the core algorithms, types of transformations used in the analysis, partitioning, checkpointing and shuffle among the stages of calculation.

The policy diffusion analysis involves **map**, **filter**, **join** and **aggregateByKey** transformations. All-pairs similarity calculation involves a two-sided **join** transformation with wide dependencies among partitions, resulting in an order of 100 TB of intermediate data and intensive shuffles, making the analysis challenging. The cost of failure for a partition with wide dependencies is rather high, since it requires a number

procedure JOIN ON THE LEFT KEY($\tilde{D}$)
    for each (pk_j, pk_k) in RDD calculate $(pk_j, (pk_j, pk_k))$
    join with the dataset $\tilde{D}$ on the left key
end procedure
procedure JOIN ON THE RIGHT KEY($\tilde{D}$)
    for each (pk_j, pk_k) and feature vector v_j in RDD calculate $(pk_k, ((pk_j, pk_k), v_j)$
    join with the dataset $\tilde{D}$ on the right key
end procedure
procedure CALCULATE SIMILARITIES($threshold$)
    for each (pk_j, pk_k) and feature vector (v_j, v_k) in RDD calculate $((pk_j, pk_k), (v_j, v_k))$
    for each $((pk_j, pk_k), (v_j, v_k))$ calculate similarity between v_j and v_k: $((pk_j, pk_k), S_{jk})$ Filter $S_{jk} >$ threshold
end procedure

Fig. 2. Two-sided join and all-pairs similarity calculation.

of partitions to be re-computed, especially if the lineage graph is rather long. An intermediate Parquet sink is introduced between the two main steps of the computation (separating the feature extraction and document classification steps) to break the RDD's lineage.

A. All-pairs similarity calculation

The most compute and shuffle intensive part of the pipeline is the all-pairs document similarity calculation. To scale the solution up to large dataset sizes an efficient candidate selection step via k-means clustering is introduced (III-C).

Once all rows of the pre-processed dataset D are subdivided into k clusters, a copy of the clustered dataset $\tilde{D}$ is broadcasted to each partition across the nodes of Spark cluster. All combinatorial pairs of primary keys (pk_j, pk_k) corresponding to the documents are calculated in each partition, filtered by state and predicted cluster label. The result is then aggregated into an array of pairs of primary keys (pk_j, pk_k) and combined. The RDD checkpoint is introduced following this step.

Next, the two-sided join is performed to calculate the and all-pairs similarity as described in V-A. The DAG visualization of two-sided join and all-pairs similarity calculation is provided in Appendix B.

1) Shuffle and partitioning: Spark applications have been deployed on the cluster (II) with up to 40 executor containers each using 3 executor cores and 15 GB of RAM per JVM. We use external shuffle service inside the YARN node manager, observing an improved stability of memory-intensive jobs with larger number of executor containers. Fig. 3 shows the efficiency of the computation on the Spark cluster, defined as:

$$E = \frac{T_0}{N_{exec} \cdot T_N} \tag{6}$$

where T_0 is the processing time on a single executor and T_N is the processing time using N_{exec} executor containers. The total processing time is composed of the executor compute time, shuffle read-write time, task serialization and deserialization times, excluding scheduler delays. The efficiency is calculated for four distinct samples containing legislative proposals from 2, 4, 6 and 10 states respectively as a function of the number of executor containers used in the calculation. As seen, the efficiency decreases down to 50% for the case of 2 state sample, which is due to a relatively small problem size. The efficiency in the high-executor region is improved as the sample size increases, staying above 80% for the 10 state sample.

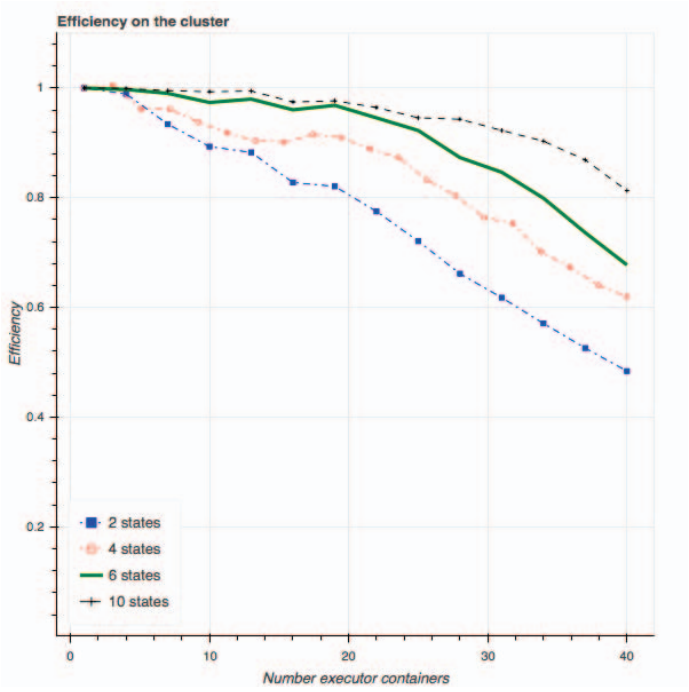

Fig. 3. Efficiency as a function of the number of executor containers used in the calculation for different dataset sizes.

An intensive shuffle across partitions of the dataset has been identified as the main cause of efficiency decrease. Fig. 4, shows the scaling of the document similarity calculation step as a function of number of processing cores, as well as the effects of changing the fraction of Java heap space used during shuffles. If the specified threshold for in-memory maps used for shuffles is exceeded, the contents will begin to spill to disk. Increasing the value of the memory fraction to 50% of the executor memory allowed to maintain a good scaling beyond 90 processing cores.

Partition is a unit of parallelism in Spark. Proper partitioning is necessary to speed-up the computation and avoid out-of-memory errors. The algorithm described in section (V-A) avoids grouping by key, thus minimizing the shuffle and eliminating "struggler tasks" which arise due to an uneven distribution of documents over key-groups.

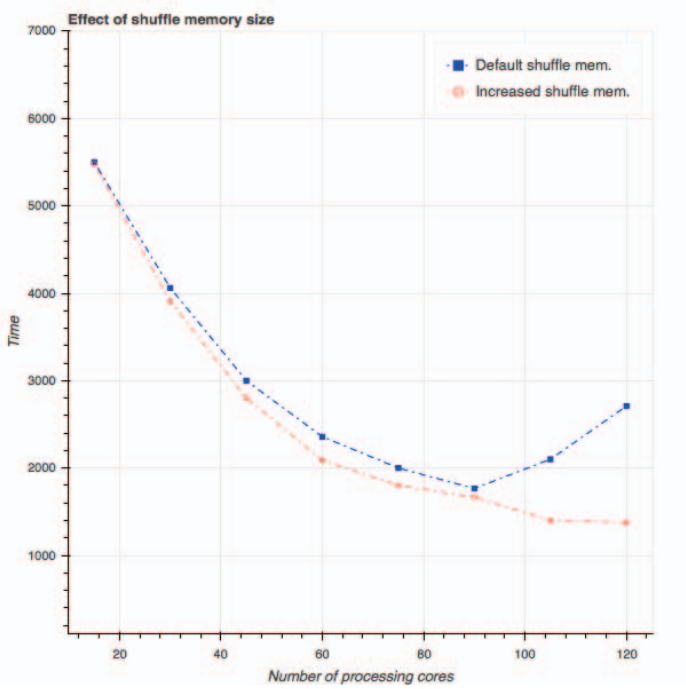

Fig. 4. Processing time as a function of number of virtual cores in the Spark cluster. Comparison of two curves illustrate the effect of the shuffle memory spill fraction.

VI. Conclusion

The Apache Spark framework has been evaluated for the case of data-intensive machine learning problem. A text processing pipeline utilizing Avro serialization framework, Spark ML, GraphFrames, and Histogrammar suite of data aggregation primitives has been proposed in application to a policy diffusion problem.

The proposed framework allows to efficiently calculate all-pairs comparison between legislative bills, estimate relationships between bills on a graph, and is potentially applicable to a wider class of fundamental text mining problems of finding similar items.

Histogrammar tool, adopted as a part of the framework to enable interactive analysis, allows a researcher to perform analysis in Scala language, integrating well with Hadoop ecosystem.

VII. Appendix A

Histogrammar is available on Maven Central, a publicly accessible Java/Scala repository with dependency management. To use it in Spark 2.0, one does not have to download anything. Following will start spark-shell and include the Histogrammar core as well as the packages needed to work with Spark SQL and Bokeh (VII).

Following lines of Scala code in the interactive spark-shell produce an editable HTML file with the similarity distribution plot for all the bill pairs having at least one bill from Florida (VII).

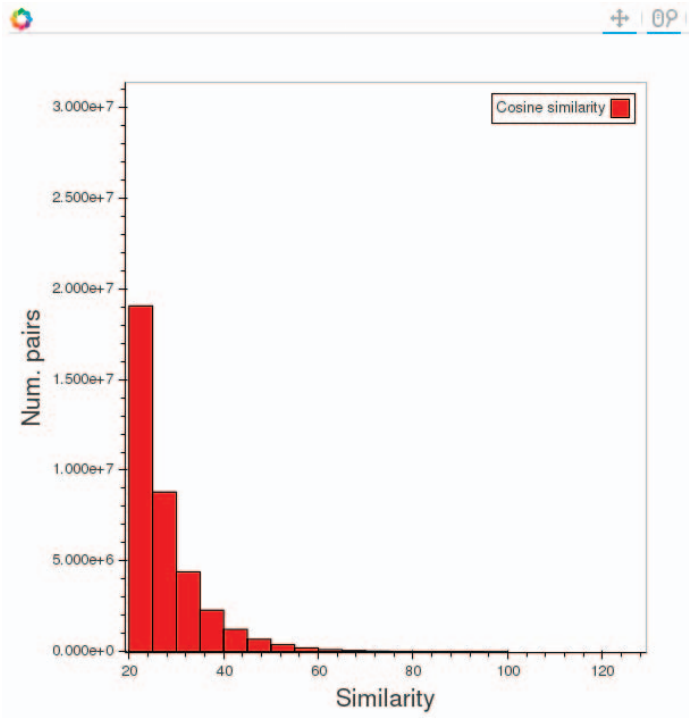

Fig. 5. Similarity distribution produced with Histogrammar using Bokeh plotting front-end.

VIII. Appendix B

References

[1] F. Gilardi, "Four ways we can improve policy diffusion research," *State Politics and Policy Quarterly*, p. 1532440015608761, 2015.

[2] "LexisNexis StateNet," http://www.lexisnexis.com/en-us/products/state-net.page, accessed: 2016-08-30.

[3] J. Walker, "The diffusion of innovations among the american states," *American Political Science Review*, vol. 63, no. 3, pp. 880–899, 1969.

[4] V. Gray, "Innovation in the states: A diffusion study," *American Political Science Review*, vol. 67, no. 4, pp. 1174–1185, 1973.

[5] F. S. Berry and W. Berry, "State lottery adoptions as policy innovations: An event history analysis," *American Political Science Review*, vol. 84, no. 2, pp. 395–415, 1990.

```
spark-shell --packages "org.diana-hep:histogrammar_2.11:1.0.3",
"org.diana-hep:histogrammar-bokeh_2.11:1.0.3",
"org.diana-hep:histogrammar-sparksql_2.11:1.0.3"
```

```
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("path2").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_j.bokeh(glyphType="histogram",glyphSize=3,fillColor=Color.Red)
            .plot(xLabel="Similarity",yLabel="Num. pairs")
save(plot,"cosine_sim.html")
```


[6] S. J. Balla, "Interstate professional associations and the diffusion of policy innovations," *American Politics Research*, vol. 29, no. 3, pp. 221–245, 2001.

[7] C. Volden, "States as policy laboratories: Emulating success in the children's health insurance program," *American Journal of Political Science*, vol. 50, no. 2, pp. 294–312, 2006. [Online]. Available: http://dx.doi.org/10.1111/j.1540-5907.2006.00185.x

[8] R. J. Kreitzer, "Politics and morality in state abortion policy," *State Politics and Policy Quarterly*, vol. 15, no. 1, pp. 41–66, 2015.

[9] K. Mossberger, *The politics of ideas and the spread of enterprise zones*. Georgetown University Press, 2000.

[10] J. Pacheco, "Attitudinal policy feedback and public opinion: the impact of smoking bans on attitudes towards smokers, secondhand smoke, and antismoking policies," *Public opinion quarterly*, p. nft027, 2013.

[11] C. R. Shipan and C. Volden, "Policy diffusion: Seven lessons for scholars and practitioners," *Public Administration Review*, vol. 72, no. 6, pp. 788–796, 2012.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[13] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[14] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742797

[16] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia, "Graphframes: An integrated api for mixing graph and relational queries," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '16. New York, NY, USA: ACM, 2016, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2960414.2960416

[17] J. Pivarski and A. Svyatkovskiy, "histogrammar-scala: 1.0.0," Sep. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.61344

[18] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log," *Empirical Software Engineering*, vol. 17, no. 4, pp. 424–466, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10664-010-9144-6

[19] R. Zadeh and G. Carlsson, "Dimension independent matrix square using mapreduce," 2013.

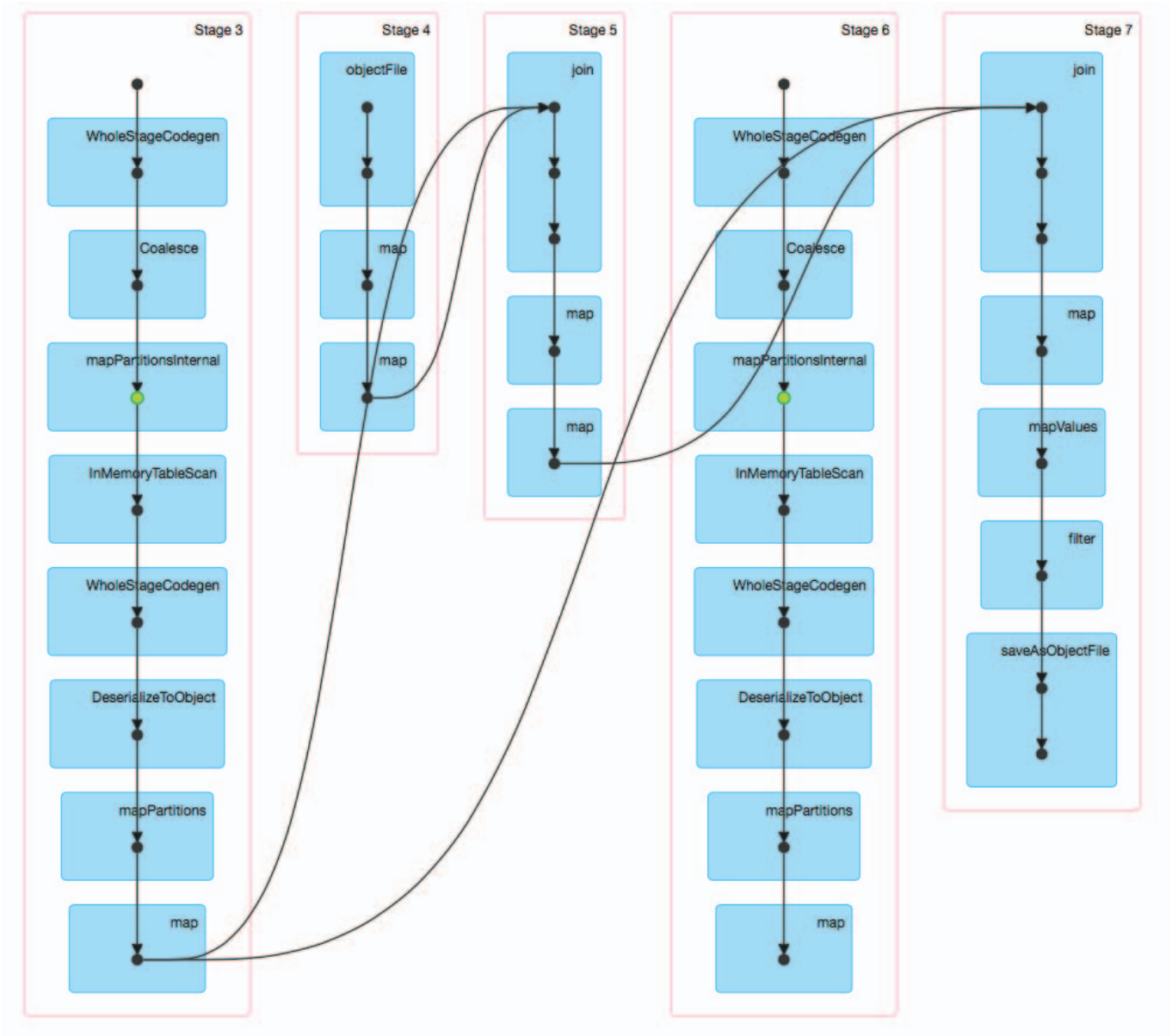

Fig. 6. DAG visualization of two-sided join and all-pairs similarity calculation.