after Dark

Spark Streaming, Machine Learning, Graph Processing, Lambda Architecture, Probabilistic Data Structs/Algos, Approximations

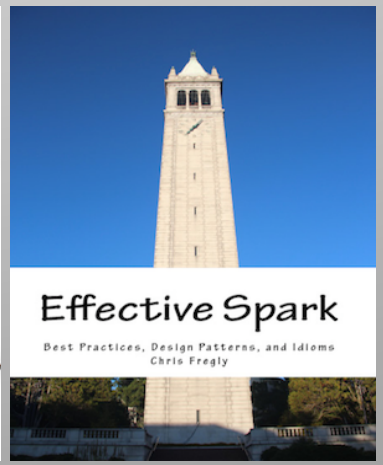
Chris Fregly

Who am I?

Former Netflix'er,
NetflixOSS Committer
(netflix.github.io)

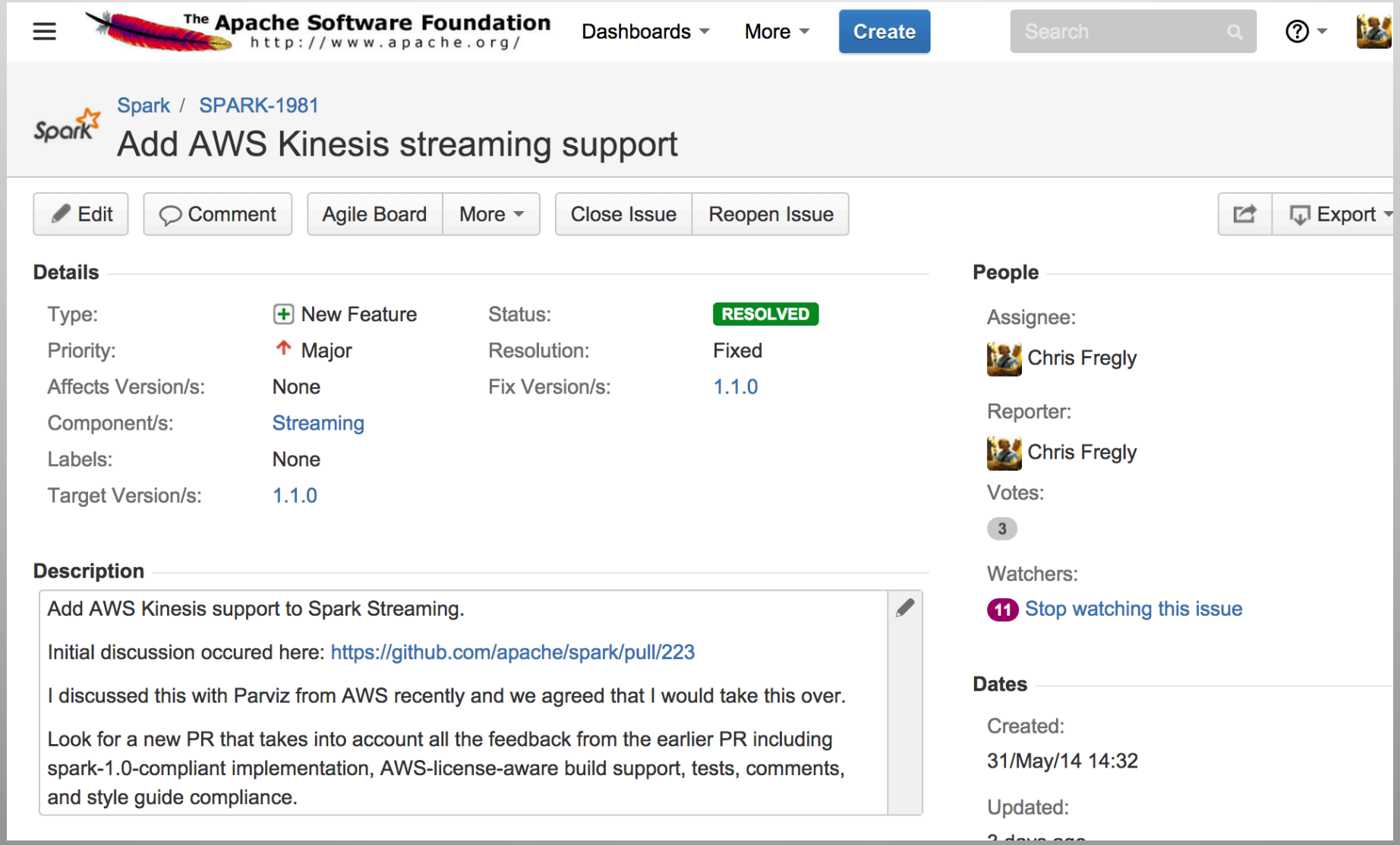
Spark Contributor
(github.com/apache/spark)

Author
(sparkinaction.com)
(effectivespark.com)



Field Engineering
(databricks.com)

Spark Streaming-Kinesis Jira



Not-so-Quick Poll

- Execution Engines
  - Spark, Hadoop, Hive, Pig
- File Formats
  - Parquet, Avro, RCFile, ORCFile
- Streaming
  - Spark Streaming, Storm, Flume, Kafka, Kinesis
- AWS Big Data
  - EMR, DynamoDB, Redshift, Kinesis
- Lambda Architecture
  - Combined Batch and Real-time Processing
- Machine Learning
  - Recommendations, Clustering, Classification
- Graph Processing
  - PageRank, Nearest Neighbor, Triangle Count
- Approximations
  - Bloom Filters, HyperLogLog, Count-min Sketch

Agenda

- Spark After Dark Overview
- Spark Overview
- Use Cases
- API and Libraries
- Machine Learning
- Graph Processing
- Execution Model
- Fault Tolerance
- Cluster Deployment
- Monitoring
- Scaling and Tuning
- Lambda Architecture
- Probabilistic Data Structs/Algos
- Approximations

DEMOS!!
DEMOS!!
DEMOS!!

DEMOS!!
DEMOS!!
DEMOS!!

DEMOS!!
DEMOS!!
DEMOS!!

DEMOS!!
DEMOS!!
DEMOS!!

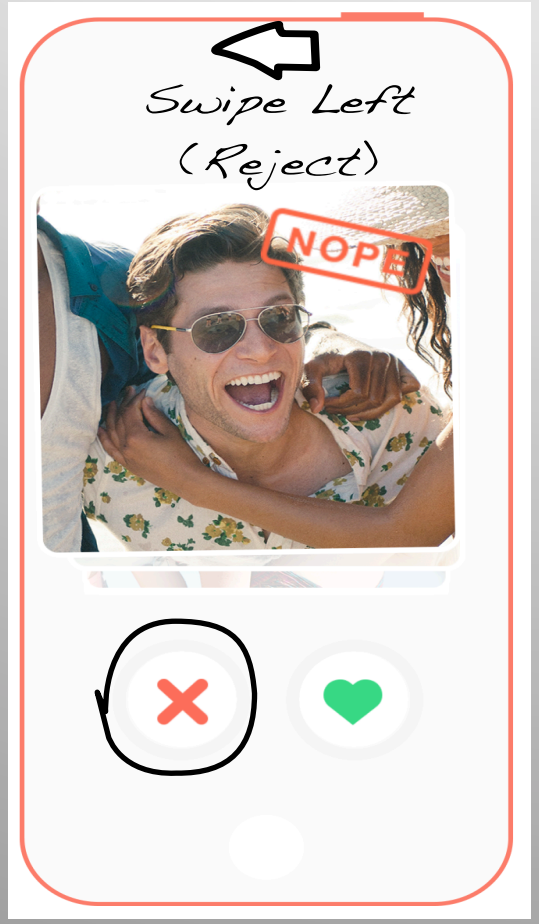
DEMOS!!
DEMOS!!
DEMOS!!

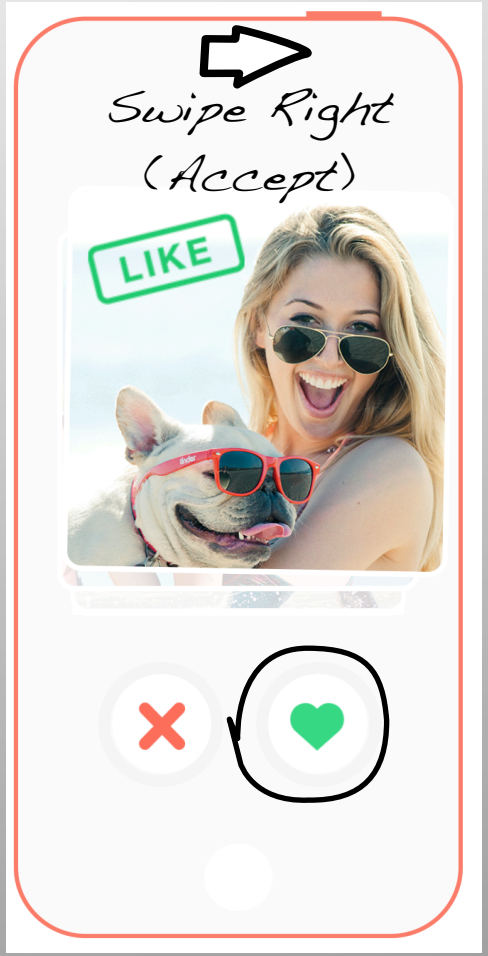
Spark After Dark (SAD) Overview

sparkafterdark.com

"Better Dating through Algorithms"

Spark After Dark (SAD) ~ Tinder*





*not affiliated with Tinder in any way

Spark After Dark Sample Dataset

Collaborative filtering dataset - dating agency

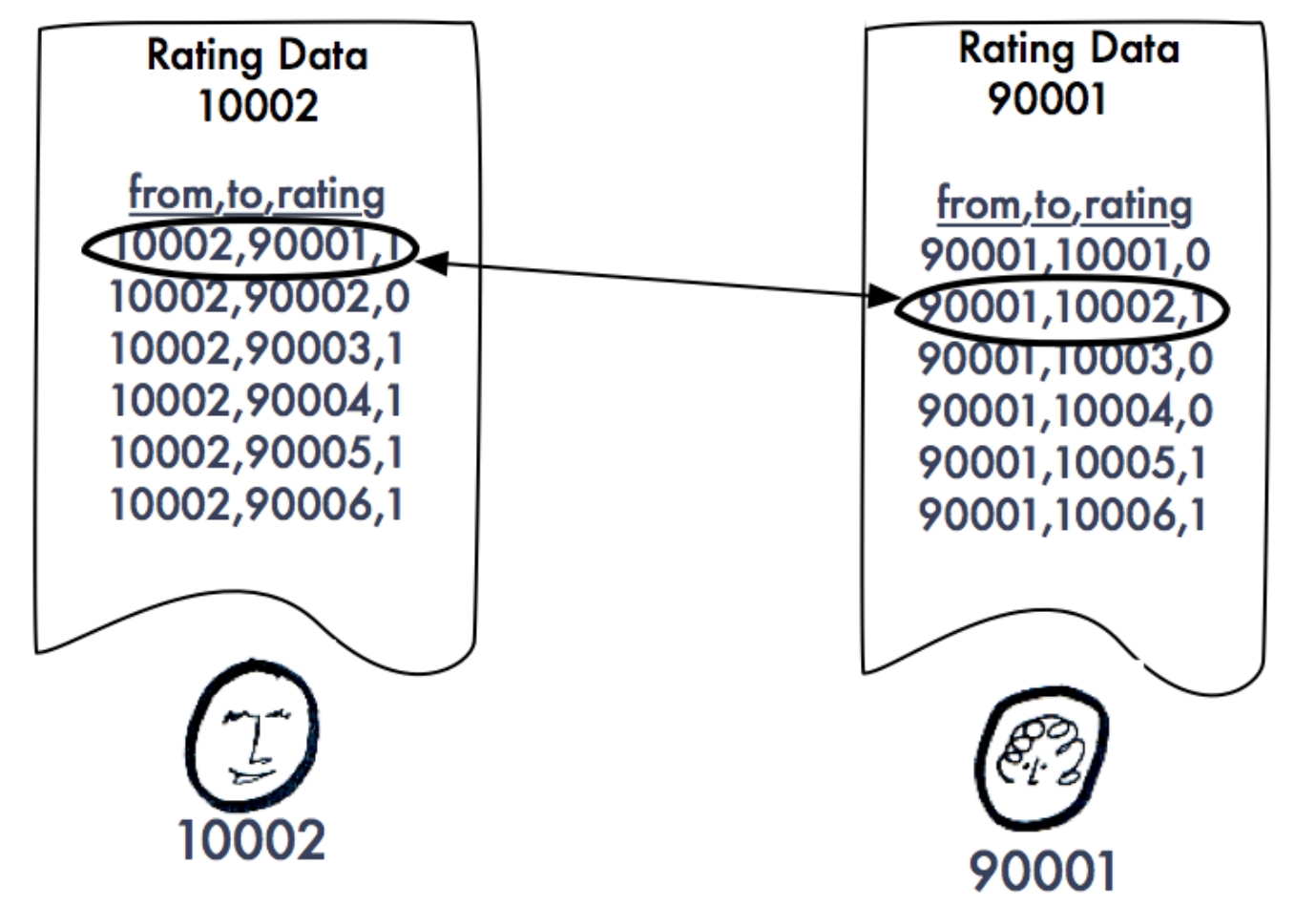
- Readme
- Users' gender
- Rating matrix
- Zipped rating and gender data

Questions and comments: Vaclav Petricek petricek(at)acm.org

```
SUMMARY
================================================================================

These files contain 17,359,346 anonymous ratings of 168,791 profiles
made by 135,359 LibimSeTi users as dumped on April 4, 2006.

The data is available from

          http://www.occamslab.com/petricek/data/
```

```
RATINGS FILE DESCRIPTION
================================================================================

All ratings are contained in the file "ratings.dat" and are in the
following format:

UserID,ProfileID,Rating

- UserID is user who provided rating
- ProfileID is user who has been rated
- UserIDs range between 1 and 135,359
- ProfileIDs range between 1 and 220,970 (not every profile has been rated)
- Ratings are on a 1-10 scale where 10 is best (integer ratings only)
- Only users who provided at least 20 ratings were included
- Users who provided constant ratings were excluded

USERS FILE DESCRIPTION
================================================================================

User gender information is in the file "gender.dat" and is in the following
format:

UserID,Gender

- Gender is denoted by a "M" for male and "F" for female and "U" for unknown
```

Goal: Increase Matches

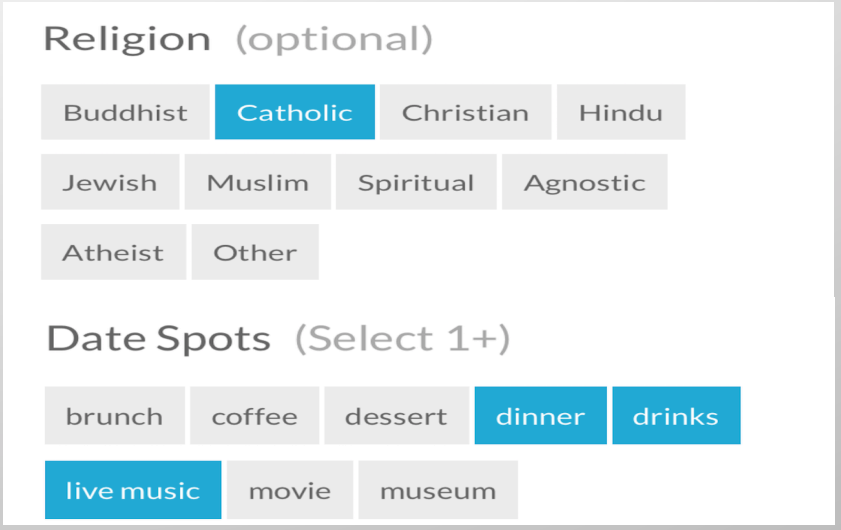


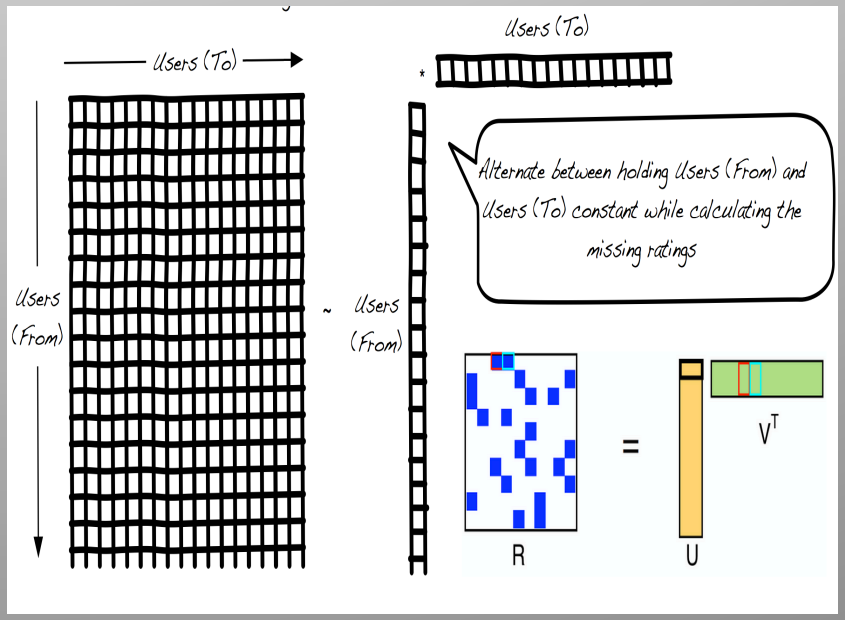
Matches: Spark SQL, BlinkDB

- Traditional rating aggregations
  - Most-active users (SUM)
  - Most-rated users (SUM)
  - Top-rated users (AVG)
  - Spark SQL
- Approximate online rating aggregations
  - Error-bound, time-bound
  - When exact results aren't needed
  - Near-real-time trend analysis
  - BlinkDB

Matches: MLlib, GraphX

- Find similar based on interests, religion, etc
  - MLlib: K-Means Cluster, DIMSUM Similarity, DecisionTree & RandomForest Classifier
- Recommendations
  - Fill in missing ratings
  - MLlib: Alternating Least Squares (ALS)



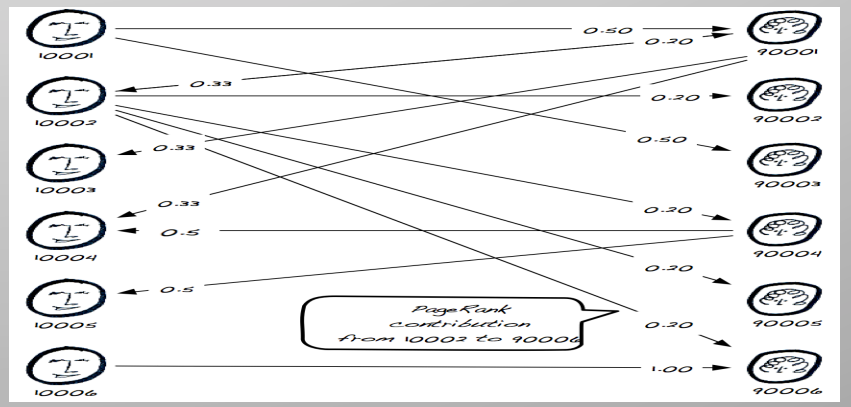


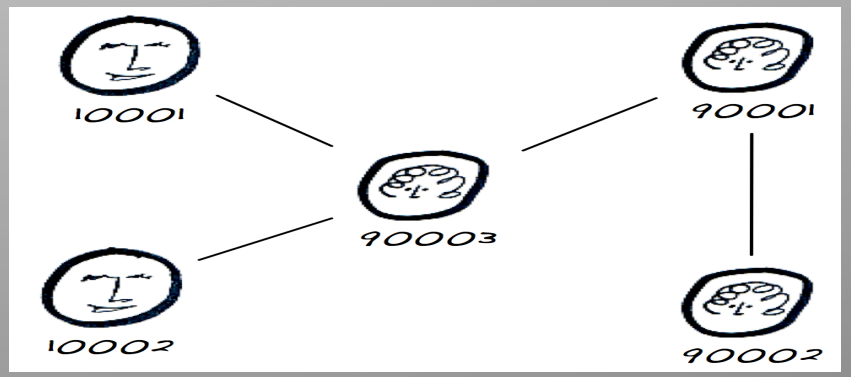
Matches: MLlib + GraphX

- Profile Text Analysis
  - MLlib: TF/IDF, N-gram, LDA Topics
  - GraphX: TextRank
- Top Influencers
  - Most desirable people
  - GraphX: PageRank
- People You May Know
  - More intimate matches
  - GraphX: Shortest Path, Connected Components

The six things I could never do without

1. Sonicare. I have 2: 1 for my shower, 1 for my sink.
2. Kindle app for my iPad. I read mostly tech stuff to stay on top of trends.
3. US Weekly. Comes directly to my iPad each week!
4. Music. I'm always listening to something from the moment I wake up. Mostly country, classic hip-hop southern rock, and even some hippie.
5. My new car. Little Red Whiting Hood is her name. Can you guess her nationality?
6. Cubs hat. I wear it everywhere. Fits my head perfectly. Love it!





Matches: Spark Streaming

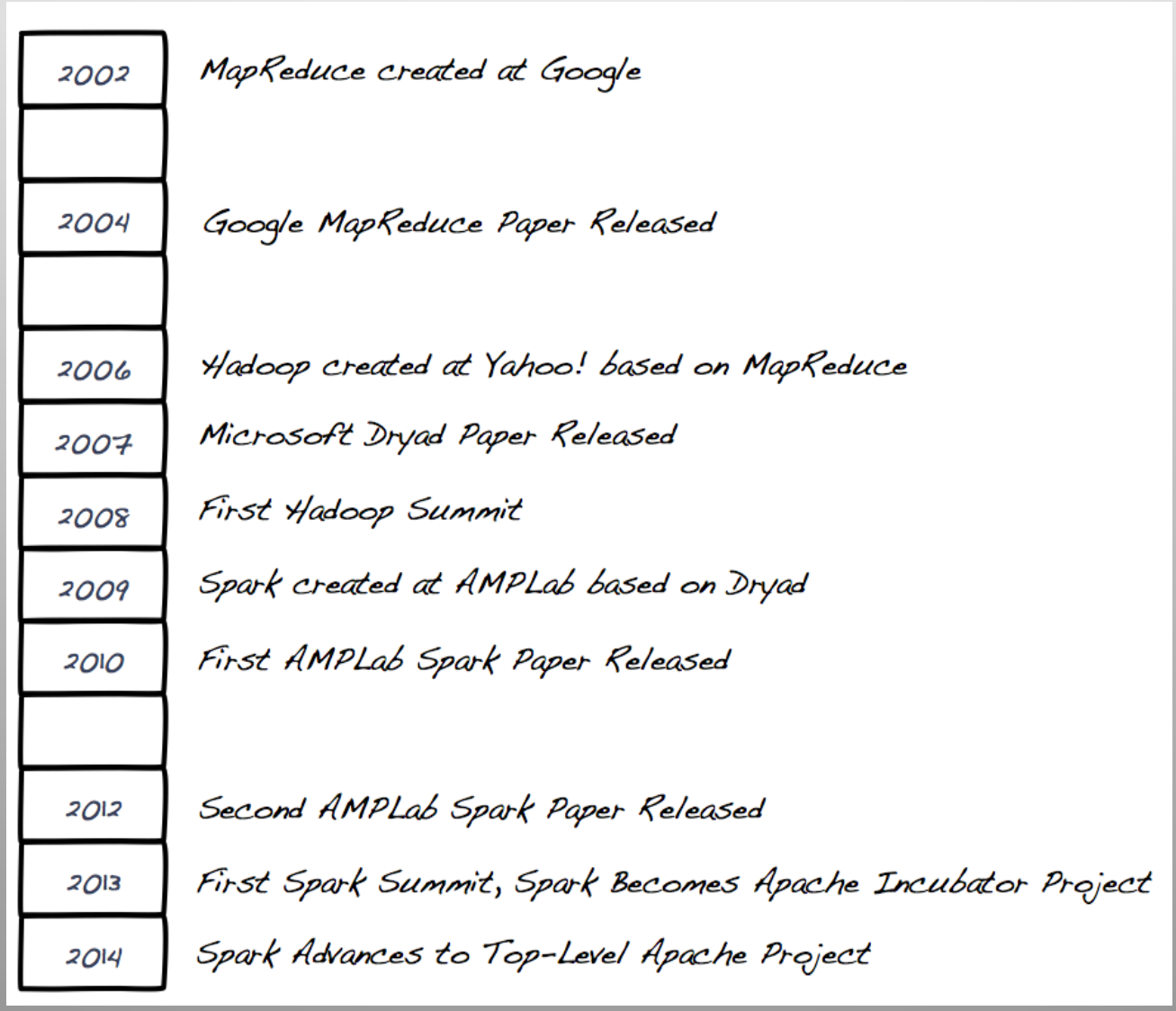
- Capture real-time, online, incremental trends!
- Spark Streaming + MLlib + GraphX
  - Clustering: StreamingKMeans.scala
  - Logistics Regression: StreamingLogR.scala
  - Linear Regression: StreamingLinR.scala
  - Naive-Bayes Classification: TODO
  - DecisionTree/RandomForest: TODO
  - PageRank: TODO
- Spark Streaming + Spark SQL
  - Lambda Architecture: real-time + batch

Spark Overview

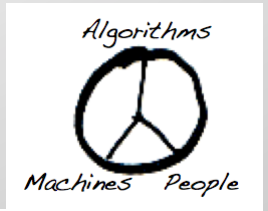
What is Spark?

- Based on 2007 Microsoft Dryad paper
- Written in Scala
- Supports Java, Python, SQL, and R
- Data fits in memory when possible, but not required
- Improved efficiency over MapReduce
  - 100x in-memory, 2-10x on-disk
- Compatible with Hadoop
  - File formats, SerDes, UDFs, Hive, and Pig

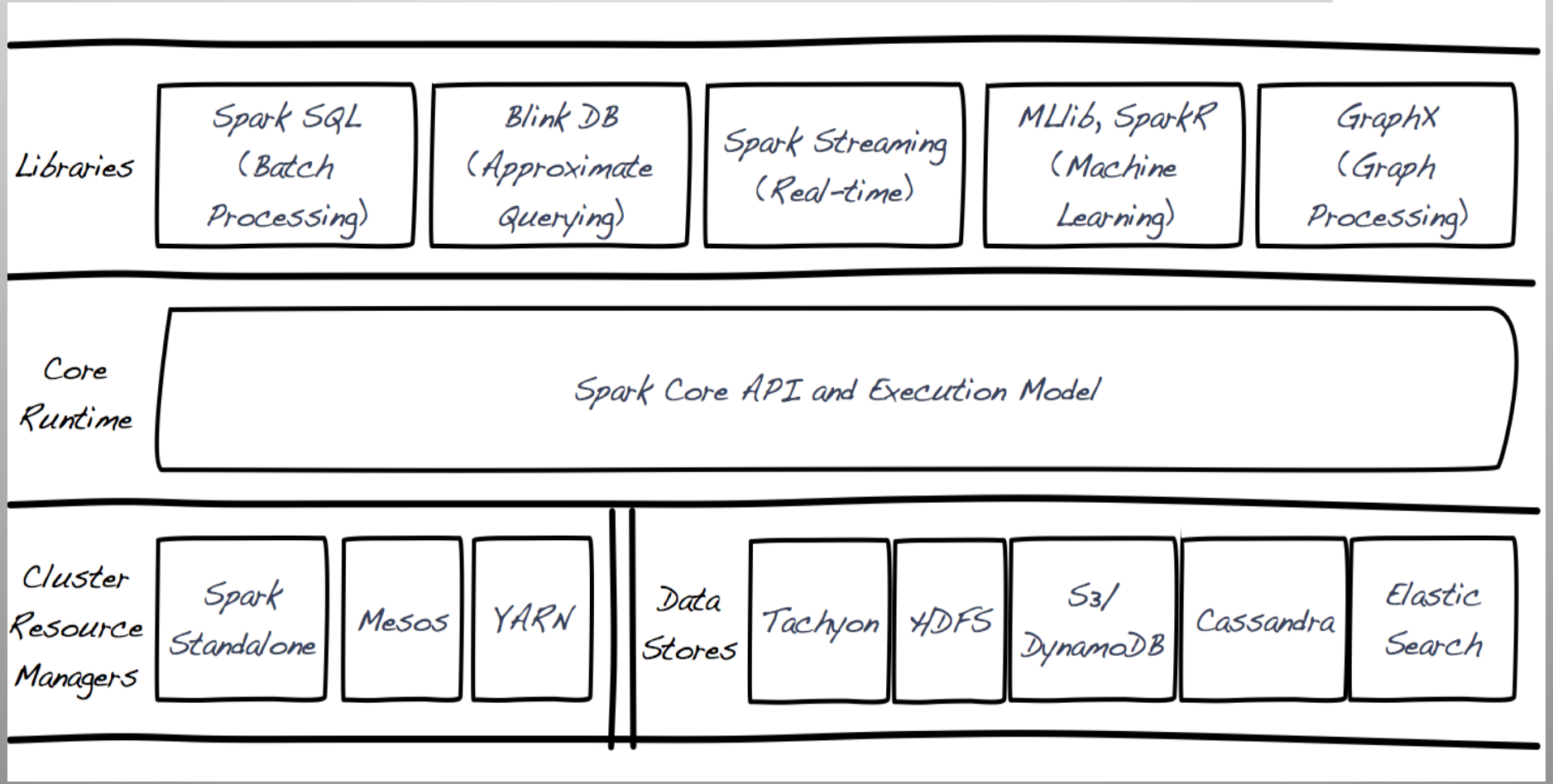
Timeline of Spark Evolution



Spark and Berkeley AMP Lab



Berkeley Data Analytics Stack (BDAS)



Advantages of Unified Libraries

- Advancements in higher-level libraries are pushed down into core and vice-versa
- Spark Core
  - Highly-optimized, low overhead, network-saturating shuffle
- Spark Streaming
  - GC, memory management, and cleanup improvements
- Spark GraphX
  - IndexedRDD for random access within a partition vs scanning entire partition
- MLlib
  - Statistics (Correlations, sampling, heuristics, etc)

Hadoop Support

- Shark
  - Hive on Spark
  - Used Hive execution engine - didn't allow for optimizations
  - *Deprecated in favor of Spark SQL + Thrift RPC Server for JDBC/ODBC connectivity
- Hive on Spark
  - Cloudera project
- Spork
  - Pig on Spark
  - Collabortion: Sigmoid Analytics and Twitter
- Parquet
  - Columnar File
  - Collaboration: Twitter and Cloudera (Impala)

Spark Use Cases

Spark SQL

- Ad hoc, exploratory, interactive analytics
- Traditional ETL, analytics, and reporting

Spark Streaming + MLlib

- Real-time traffic shifting based on CDN performance (video or music streaming)
- Proactively notify customers when errors are elevated (SaaS model)

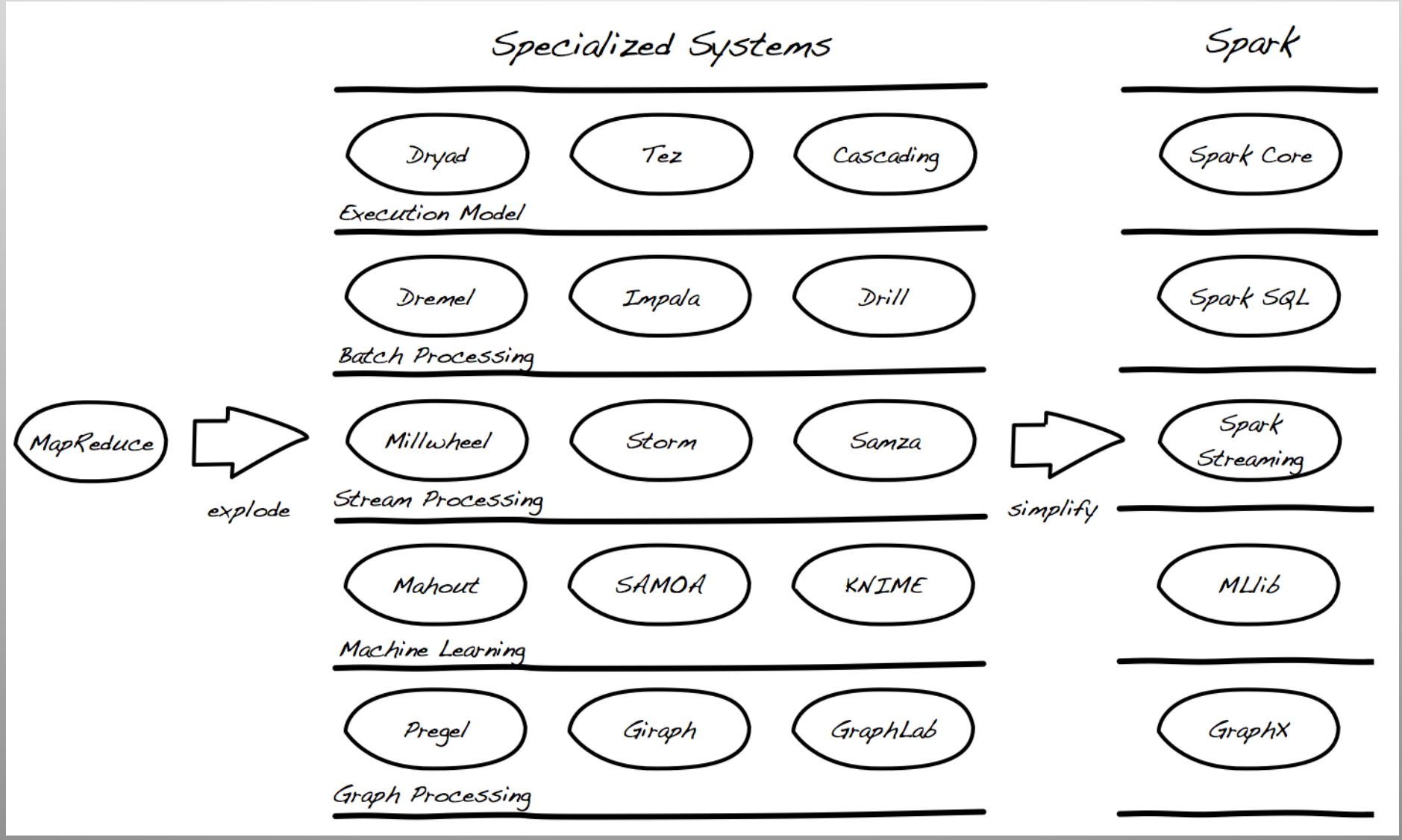
MLlib + GraphX

- Recommendations, anomaly/spam detection, related news
- Social analytics: "You May Know", top influencers

Spark Streaming + Spark SQL

- Lambda Architecture: batch + real-time

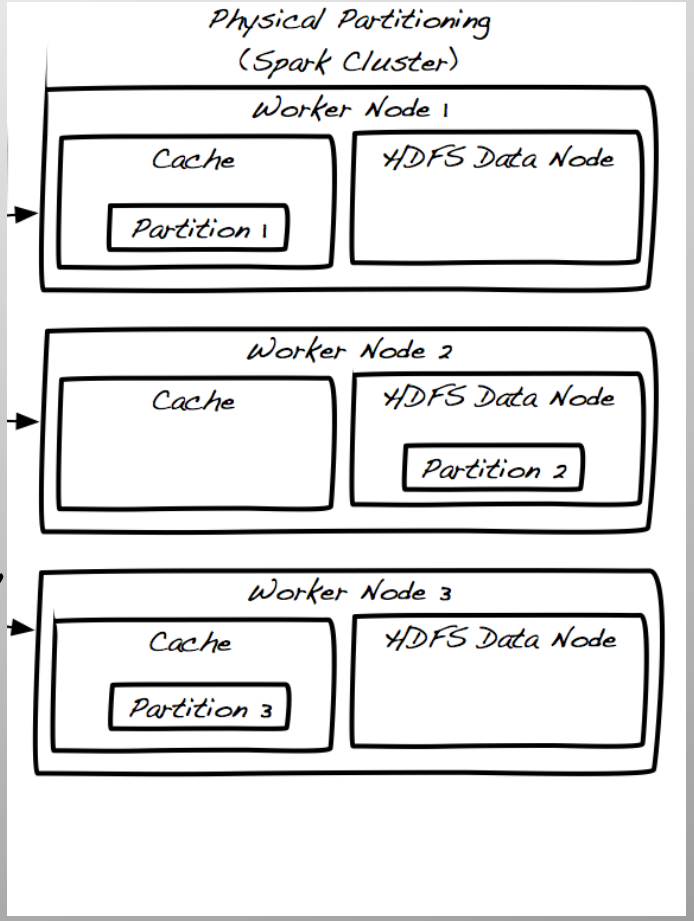
Explosion of Specialized Systems



Resilient Distributed Dataset (RDD)

RDD Overview

- Core Spark abstraction
- Represents partitions across the cluster nodes
- Enables parallel processing on data sets
- Partitions can be in-memory or on-disk
- Immutable, recomputable, fault tolerant
- Contains transformation history ("lineage") for whole data set



RDD Interface

Lineage (Required)

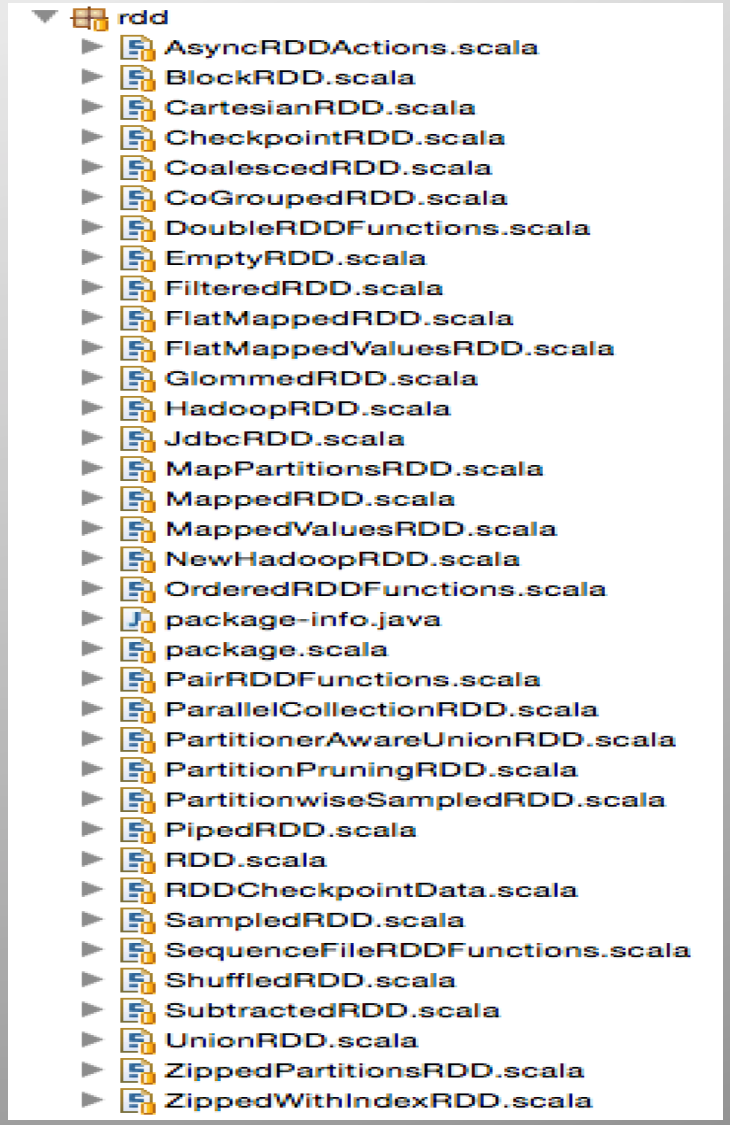
1. Set of partitions
2. List of dependencies (0, 1, many)
3. Compute function per partition

Execution optimizations (Optional)

4. Partitioner (ie. Hash, Range)
5. Preferred locations per partition

Types of RDDs

- Spark Out-of-the-Box
  - HadoopRDD
  - FilteredRDD
  - MappedRDD
  - PairRDD
  - ShuffledRDD
  - UnionRDD
  - DoubleRDD
  - JdbcRDD
  - JsonRDD
  - SchemaRDD
  - VertexRDD
  - EdgeRDD
- External
  - CassandraRDD (DataStax)
  - GeoRDD (Esri)
  - EsSpark (ElasticSearch)



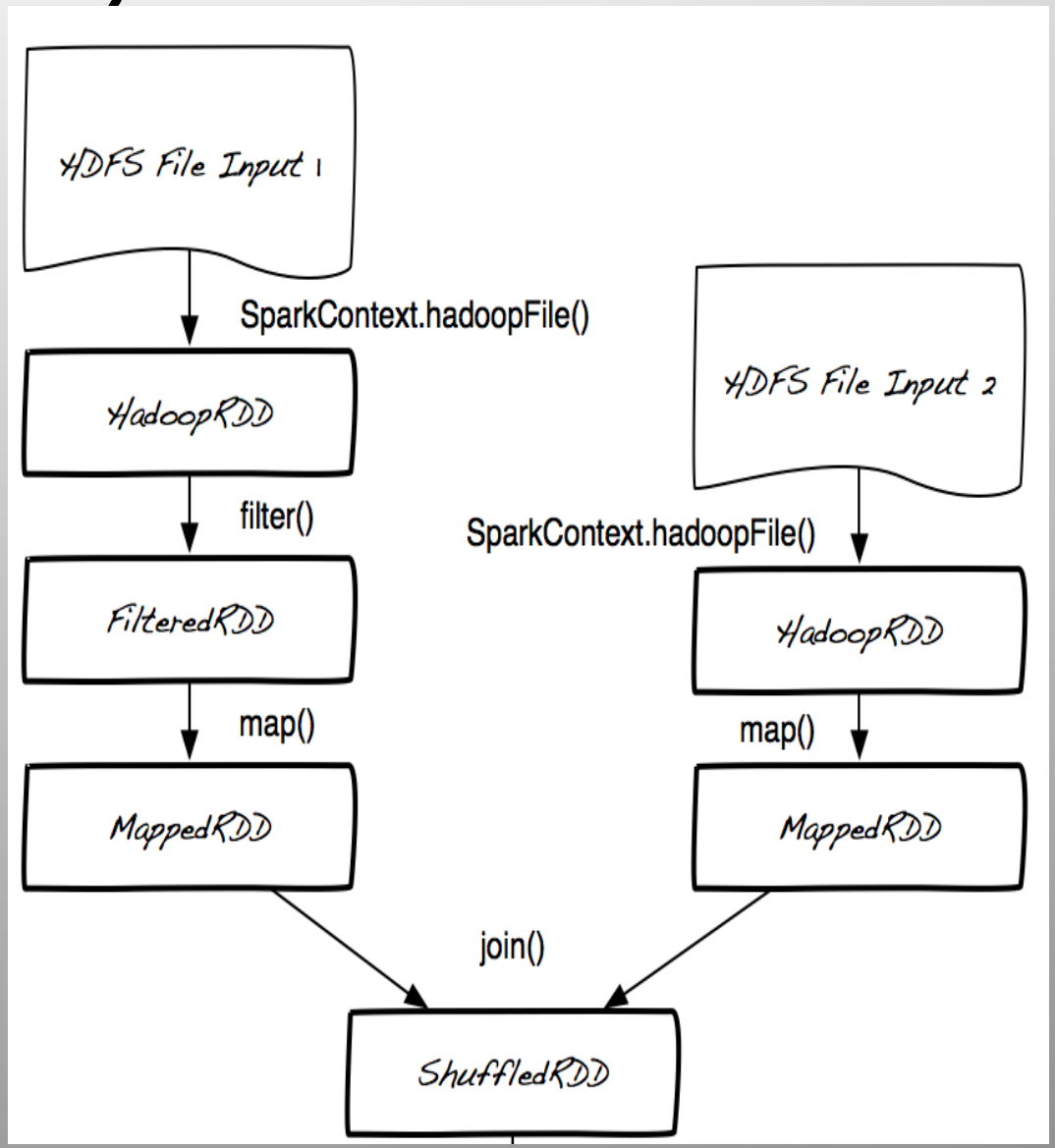
Hadoop RDD

Lineage (Required)

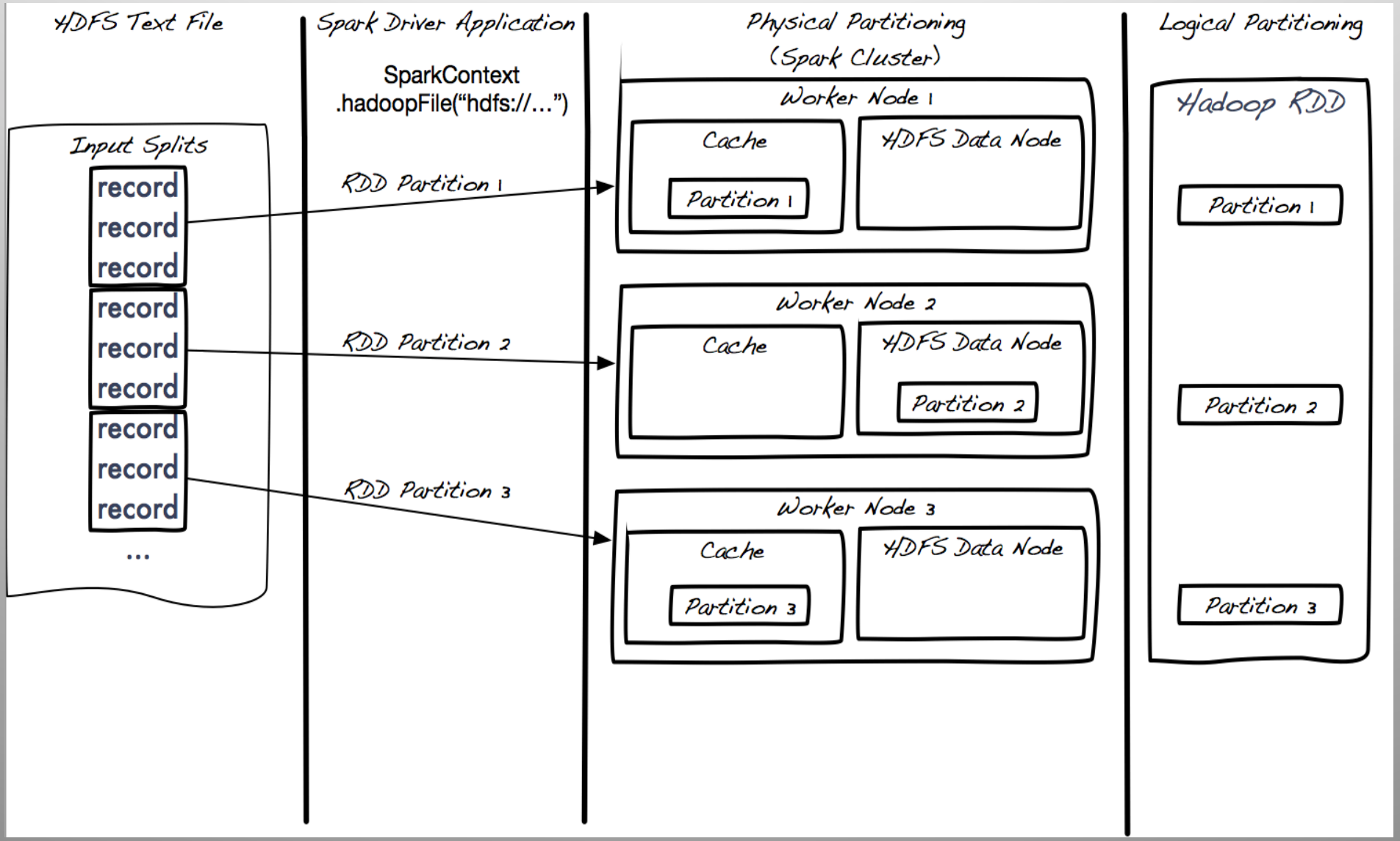
1. Set of partitions
   - 1 per HDFS block
2. List of dependencies
   - None
3. Compute function per partition
   - Read HDFS block

Execution optimizations (Optional)

4. Partitioner (ie. Hash, Range)
   - None
5. Preferred locations per partition
   - HDFS block location from NameNode



Hadoop RDD Partitions



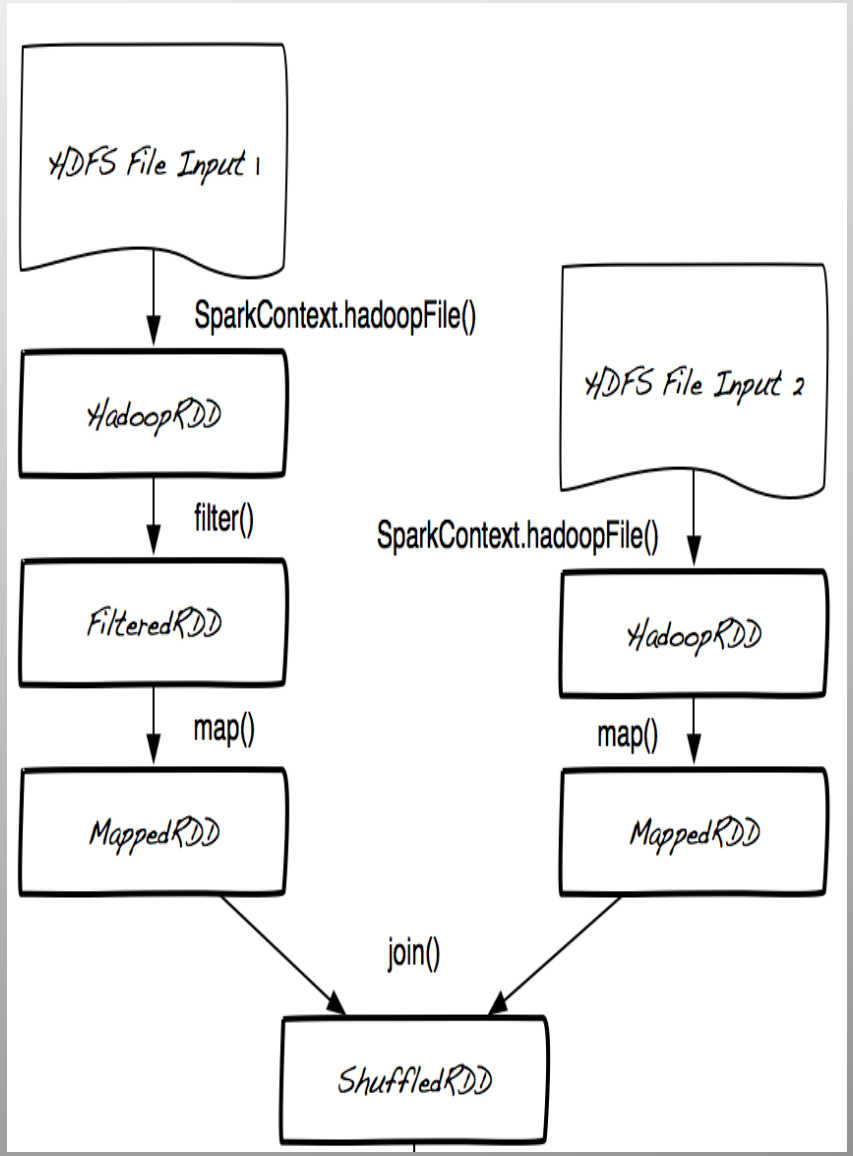
Filtered RDD

Lineage (Required)

1. Set of partitions
   - Same as parent RDD
2. List of dependencies
   - Parent RDD
3. Compute function
   - Compute parent and filter

Execution optimizations (Optional)

4. Partitioner (ie. Hash, Range)
   - None
5. Preferred locations per partition
   - Same as parent RDD



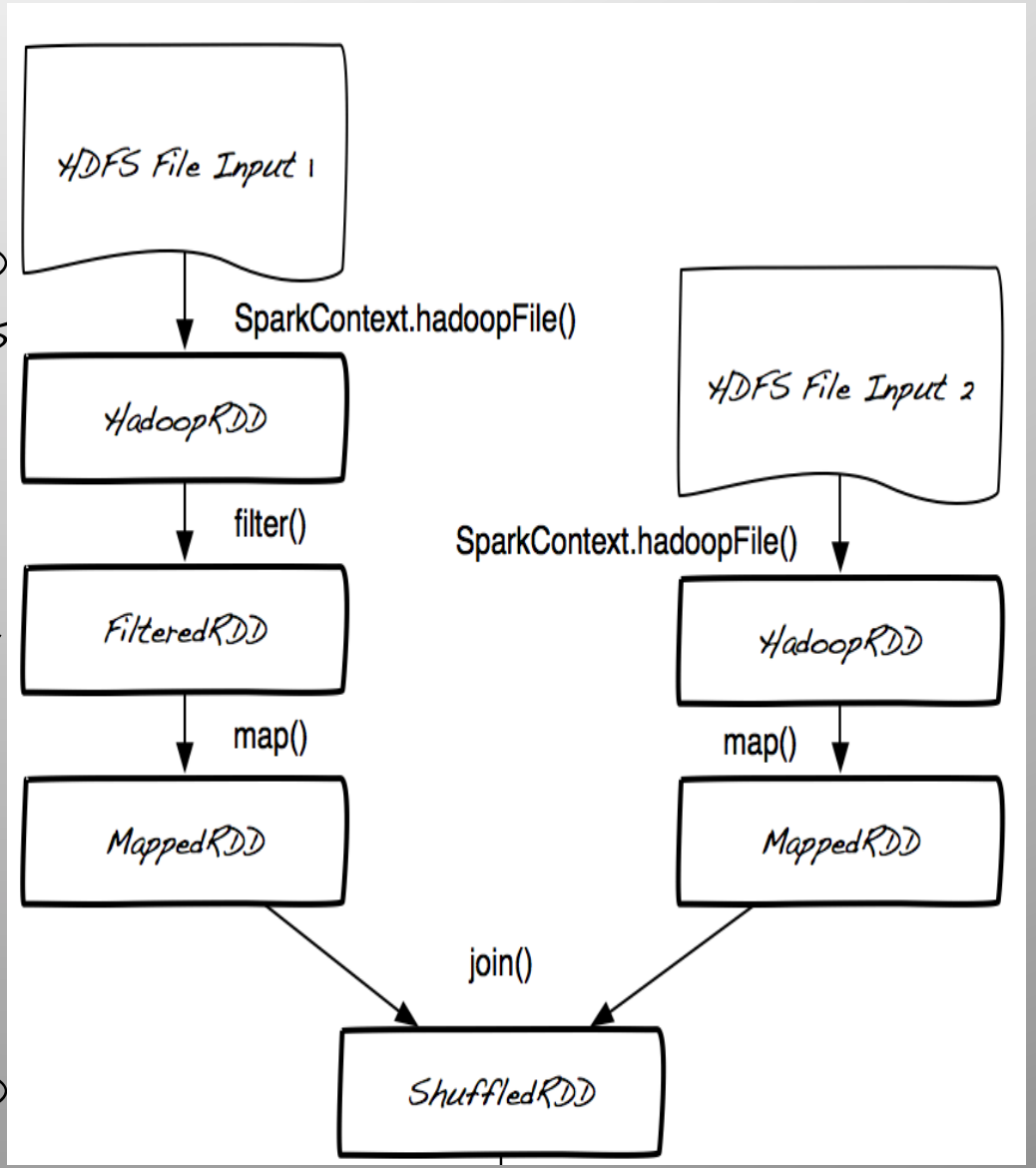
Mapped RDD

Lineage (Required)

1. Set of partitions
   - Same as parent RDD
2. List of dependencies
   - Parent RDD
3. Compute function
   - Compute parent, apply map function

Execution optimizations (Optional)

4. Partitioner (ie. Hash, Range)
   - None
5. Preferred locations per partition
   - Same as parent RDD



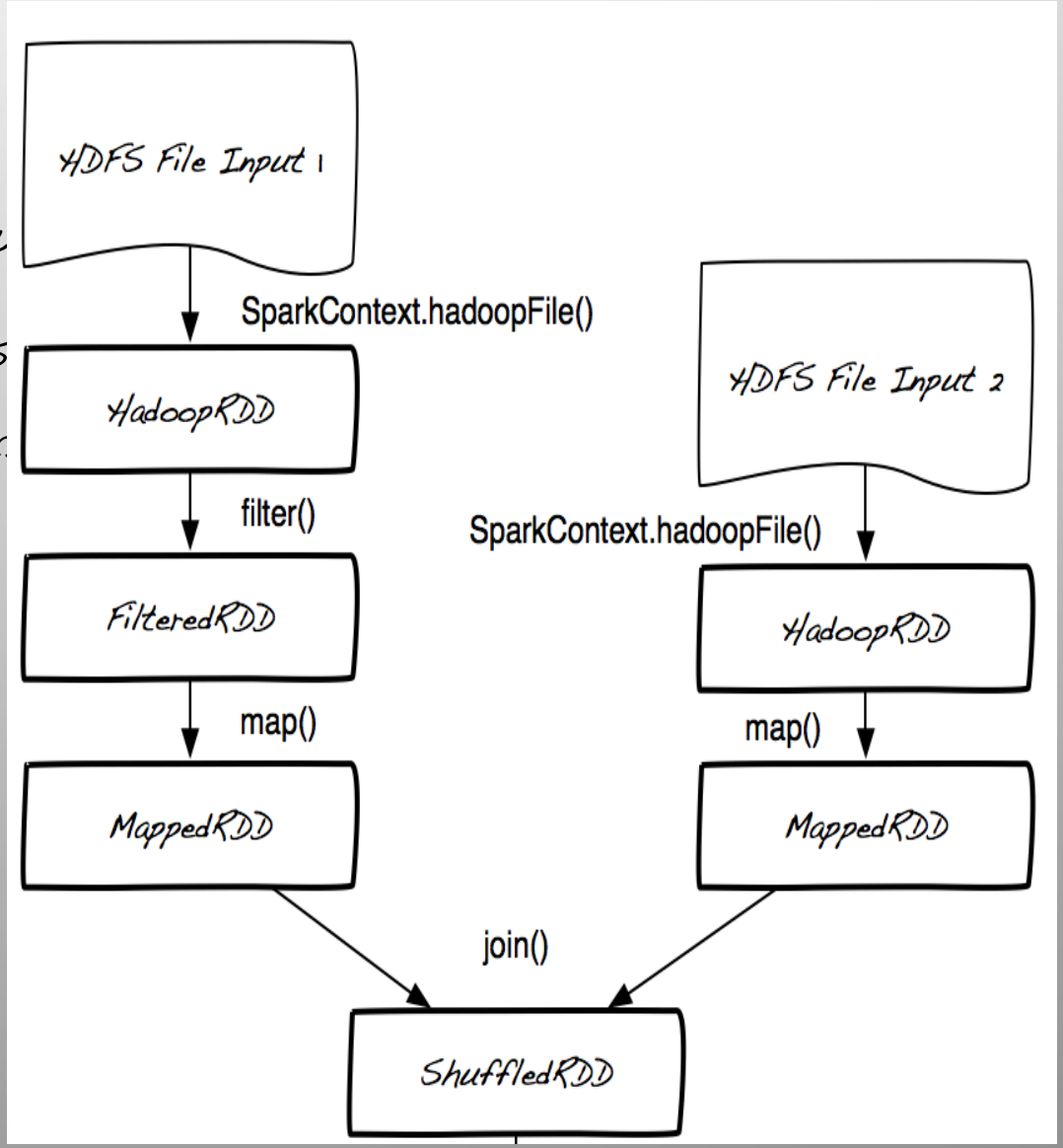
Joined RDD

Lineage (Required)

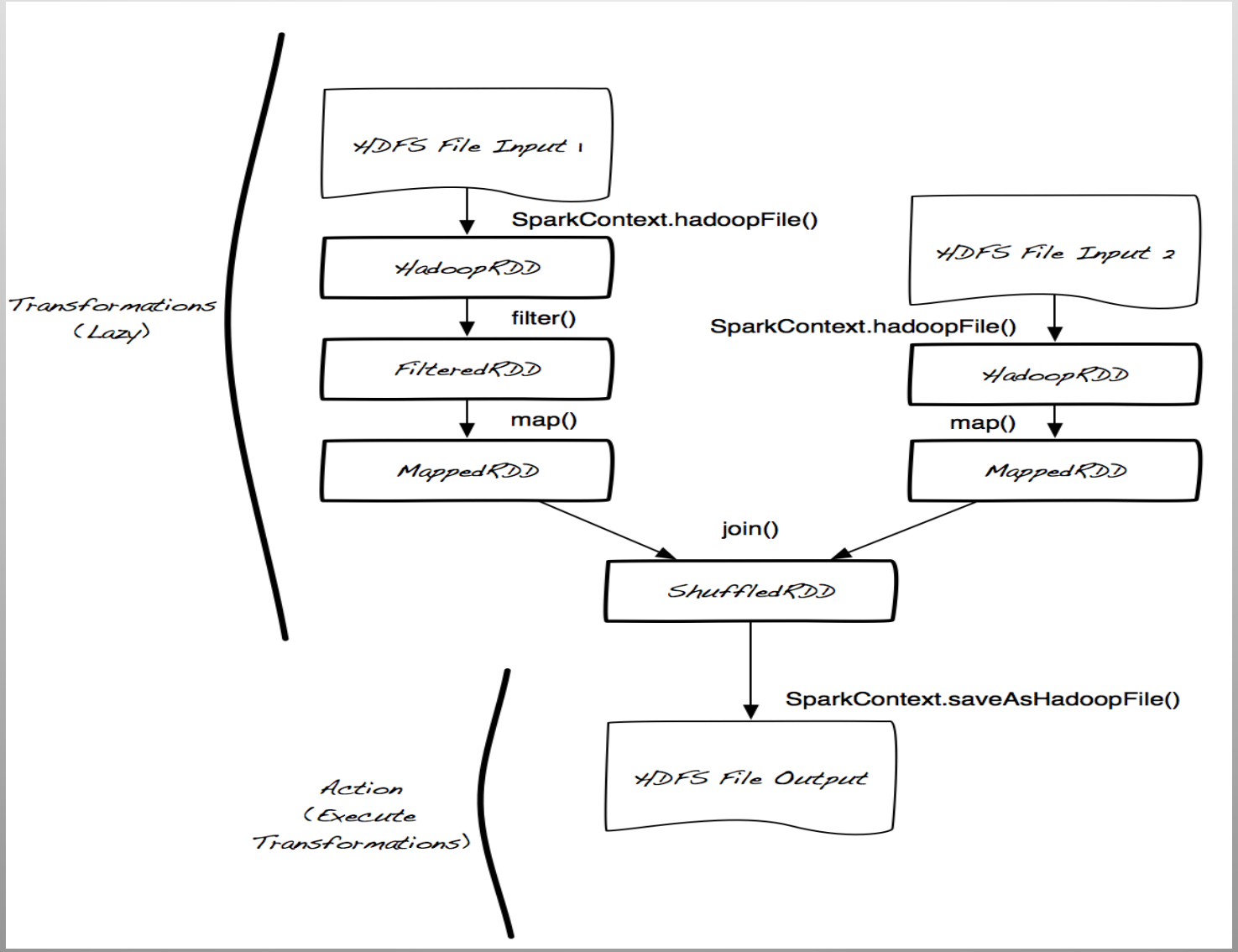
1. Set of partitions
   - One per reduce task (PART- file)
2. List of dependencies
   - Every parent (many to many shuf
3. Compute function
   - Read and join shuffled data

Execution optimizations (Optional)

4. Partitioner (ie. Hash, Range)
   - HashPartitioner
5. Preferred locations per partition
   - None



RDD Operator DAG

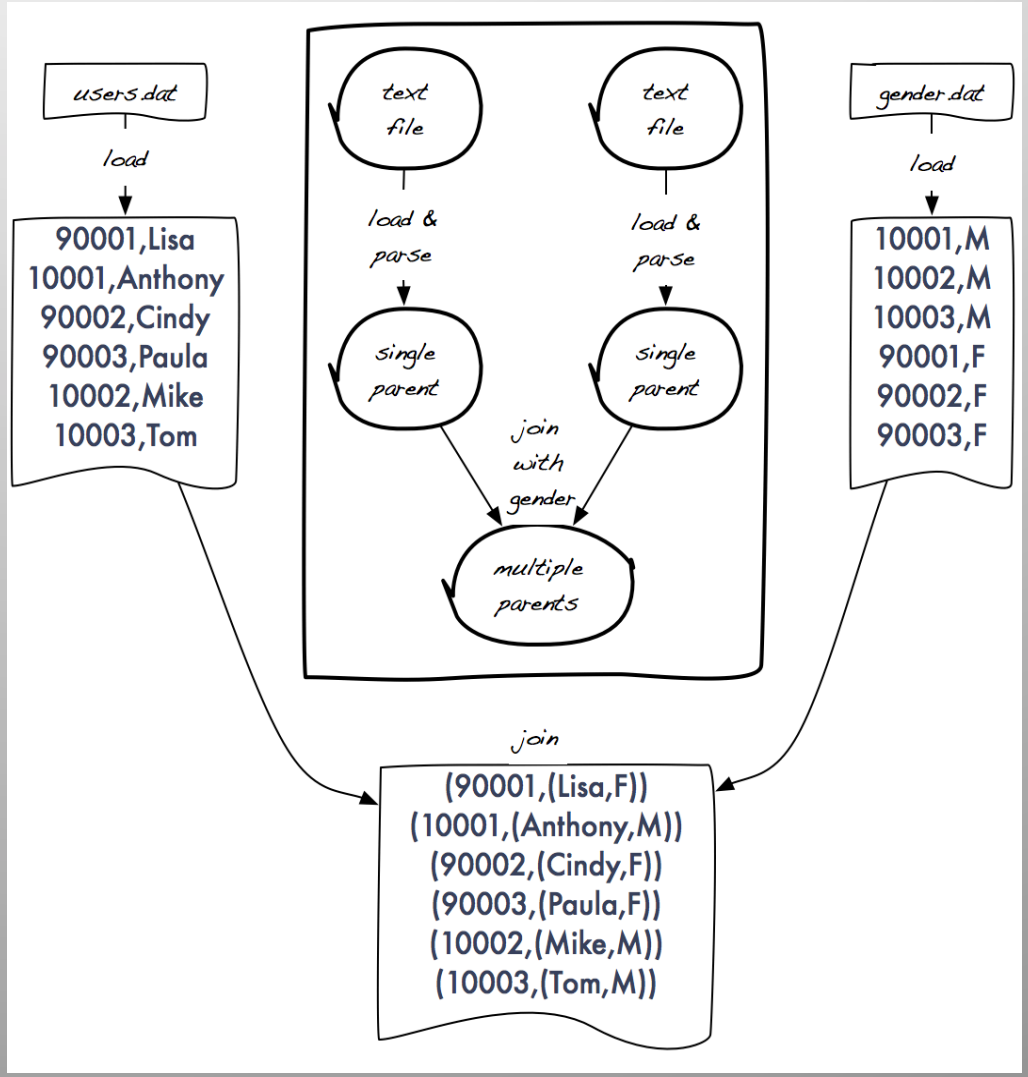


RDD Operator DAG

```
scala> messages.toDebugString
res5: String =
MappedRDD[4] at map at <console>:16 (3 partitions)
  MappedRDD[3] at map at <console>:16 (3 partitions)
    FilteredRDD[2] at filter at <console>:14 (3 partitions)
      MappedRDD[1] at textFile at <console>:12 (3 partitions)
        HadoopRDD[0] at textFile at <console>:12 (3 partitions)
```

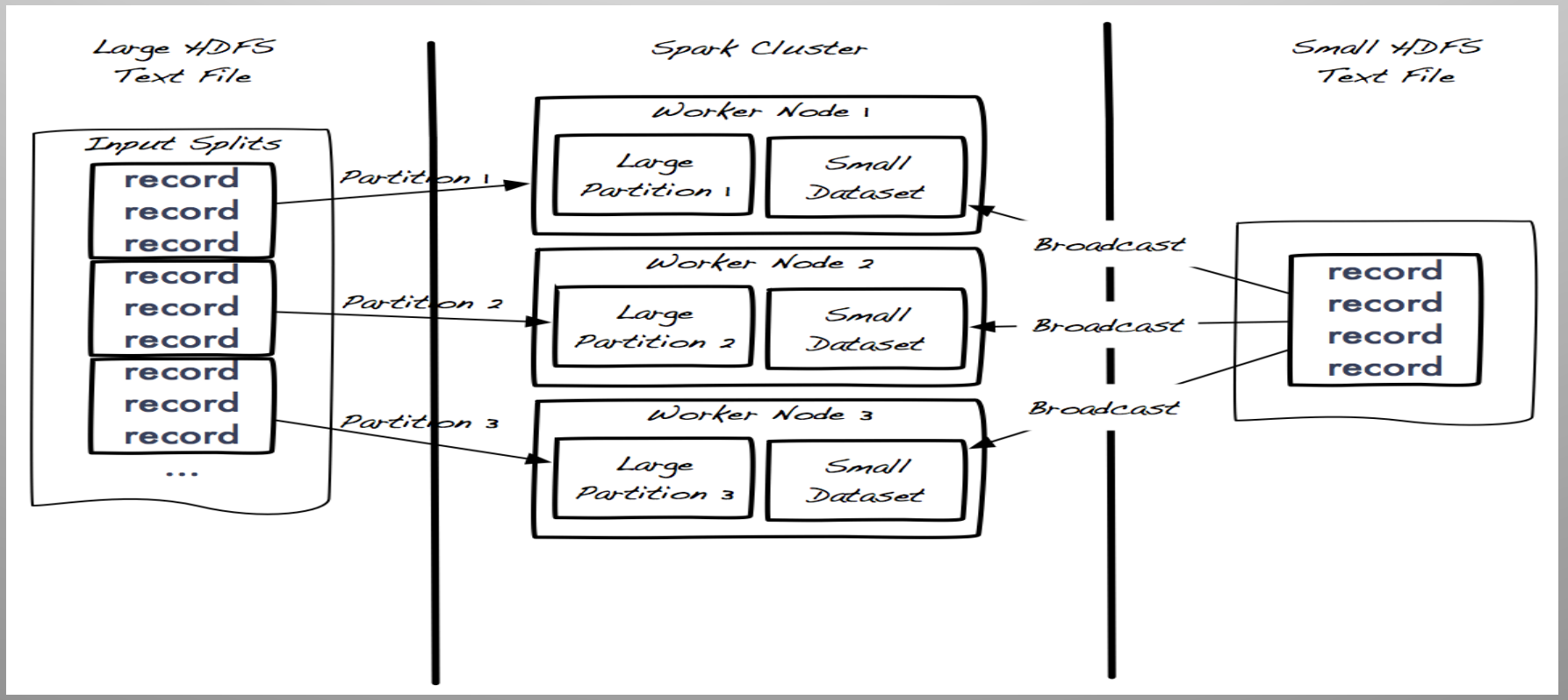
Demo: Joining Users and Gender

- Load user data
- Load gender data
- Join user data with gender data
- Analyze lineage



Join Optimizations

- When joining large dataset with small dataset (reference data)
- Broadcast small dataset to each node/partition of large dataset (one broadcast per worker node)



Spark Core API

Spark Core API Overview

- Richer, more expressive than MapReduce
- "Distributed Scala collections"
- Native support for Java, Scala, Python, SQL, and R* (coming soon)
- Unified API across all libraries
- Operations
  - Transformations (lazy evaluation DAG)
  - Actions (execute transformations)

Why Scala?

- Java/JVM is used by all popular big data frameworks
  - Lucene, SOLR, Hadoop, Mahout, Flume, Cassandra, HBase, Giraph, Storm, Avro, Protocol Buffers, Thrift, Parquet, etc
- Supports closures
  - Allows shipping code to data
- Functional, but not obscure
- Typing allows performance optimizations
- Leverage Spark REPL
- It's cool!

Transformations (Lazy)

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `combineByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are also supported through `leftOuterJoin` and `rightOuterJoin`.

Actions (Force Tranformations)

Actions

The following table lists some of the common actions supported by Spark. Refer to the RDD API doc (Scala, Java, Python) and pair RDD functions doc (Scala, Java) for details.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
takeSample(*withReplacement*, *num*, *seed*)	Return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

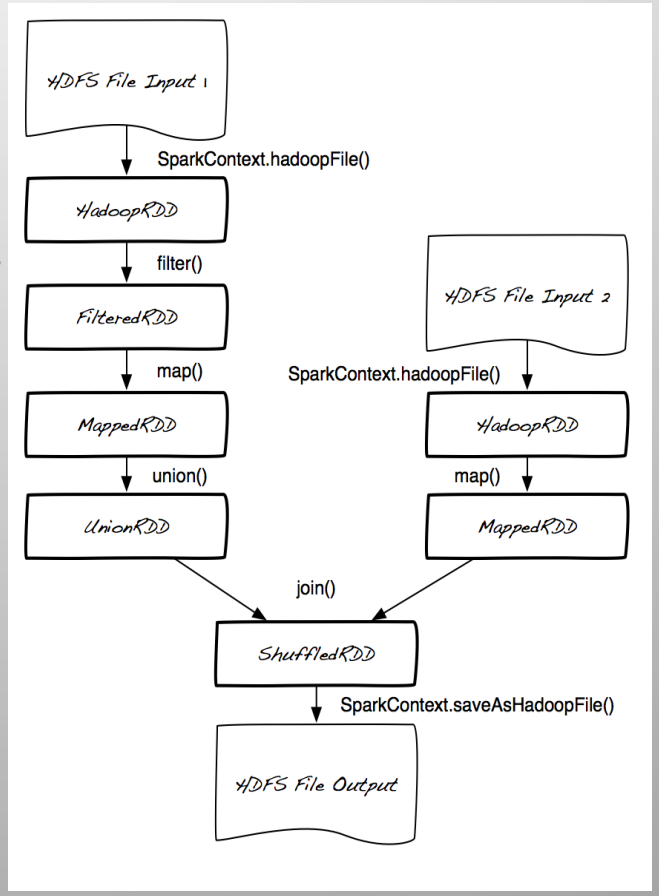
Spark Execution Model

Job Scheduler

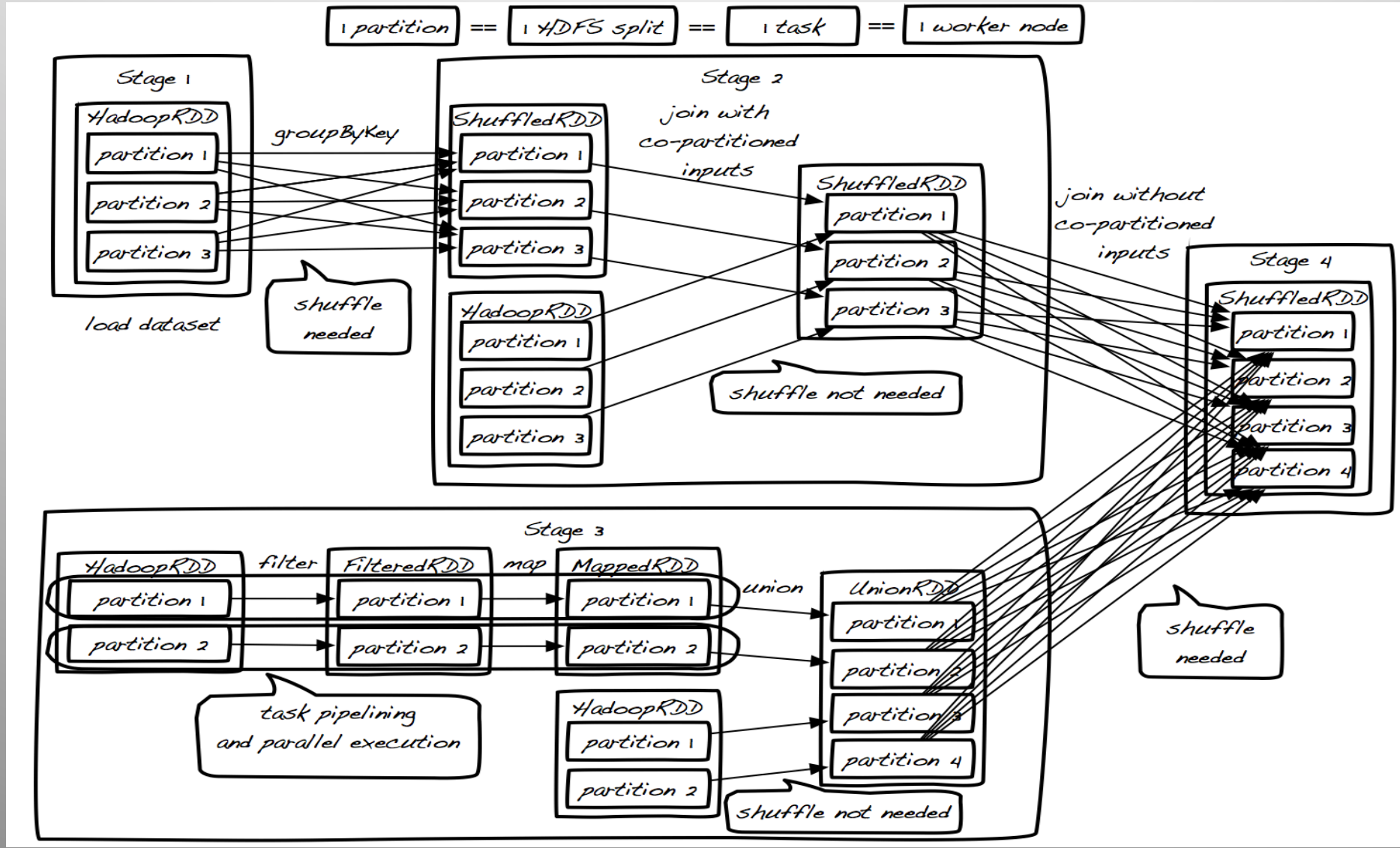
- Job
  - Contains many stages
- Stages
  - Contains many tasks
- Job resource allocation is aggressive
  - FIFO
    - Long-running jobs may starve resources for other jobs
  - Fair
    - Round-robin to prevent resource starvation
- Fair Scheduler Pools
  - High-priority pool for important jobs
  - Separate user pools
  - FIFO within the pool
  - Modeled after Hadoop Fair Scheduler

Spark Execution Model Overview

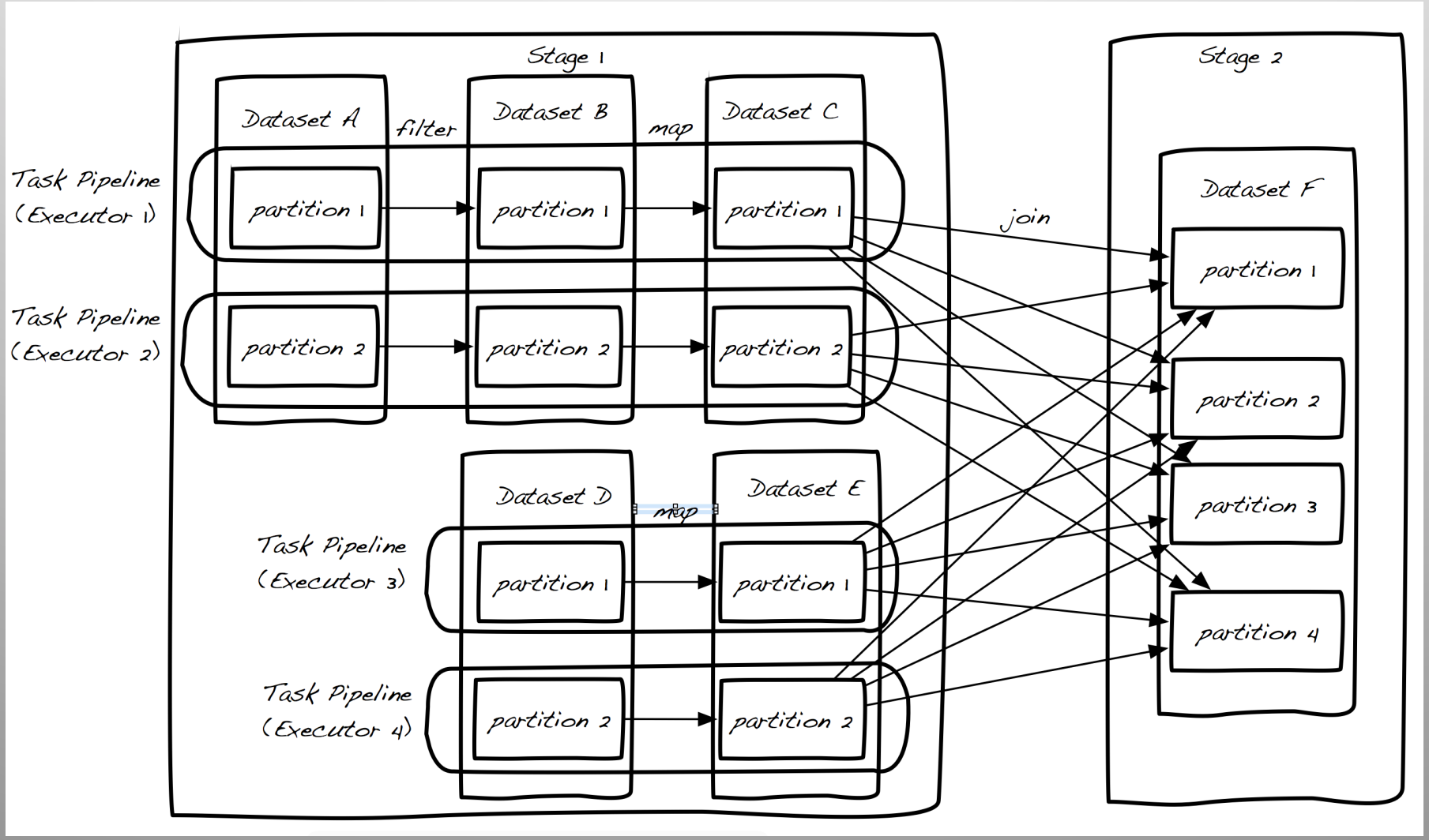
- Parallel, distributed
- DAG-based, lazy evaluation
- Allows optimizations
  - Reduce disk I/O
  - Reduce shuffle I/O
  - Single pass through dataset
  - Parallel execution
  - Task pipelining
- Data locality and rack awareness
  - PROCESS, NODE, RACK, ANY
- Fault tolerance: RDD lineage graphs



Execution Optimizations



Task Pipelining



Daytona GraySort Contest

250,000 partitions, on-disk (No caching)

	Hadoop MR Record	**Spark Record**	**Spark 1 PB**
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Improved Shuffle & Sort

- aka "Sort-based Shuffle"
- Released across both Spark 1.1 and 1.2
- Improves performance for large number of partitions/reducers (>10,000)
- Reduces memory pressure, GC, and OOMs
- Reduces OS open files
- Sequential versus random disk I/O
- Saturate the network link
  - ~125 MB/s (1 GB ethernet)
  - 1.25 GB/s (10 GB ethernet)

Improved Shuffle

- Sort-based Shuffle (SPARK-2045)
  - 1 file for all partitions (grouped by partition key)
    - Versus many files (1 per partition key)
  - Sorted in-memory, sequential file write I/O
    - Versus random file write I/O (1 per partition key)
- Java-Netty Network Module (SPARK-2468)
  - Replaces old nio socket-level code
  - epoll, zero-copy, kernel-space
  - Self-managed memory pool, less GC
- External Shuffle Service (SPARK-3796)
  - Decoupled from the Executor JVMs
  - Uses Network Module
  - Can serve shuffle files after an Executor crash or during GC

Improved Sort

- TimSort
  - Combo of merge and insertion sort
  - Optimized for partially-ordered datasets
  - Used in both map and reduce phases
- Reduced Data Movement, Cache Misses
  - Sort using (key, pointer) pairs
  - GraySort keys are 10 bytes (2^3.162277766??)
  - Compressed Oops = 4 bytes
  - 10 key bytes + 2 pad bytes + 4 pointer bytes = 16 bytes (2^4!!)
  - AlphaSort Paper ~1995 (Chris Nyberg, Jim Gray)

Elastic Spark

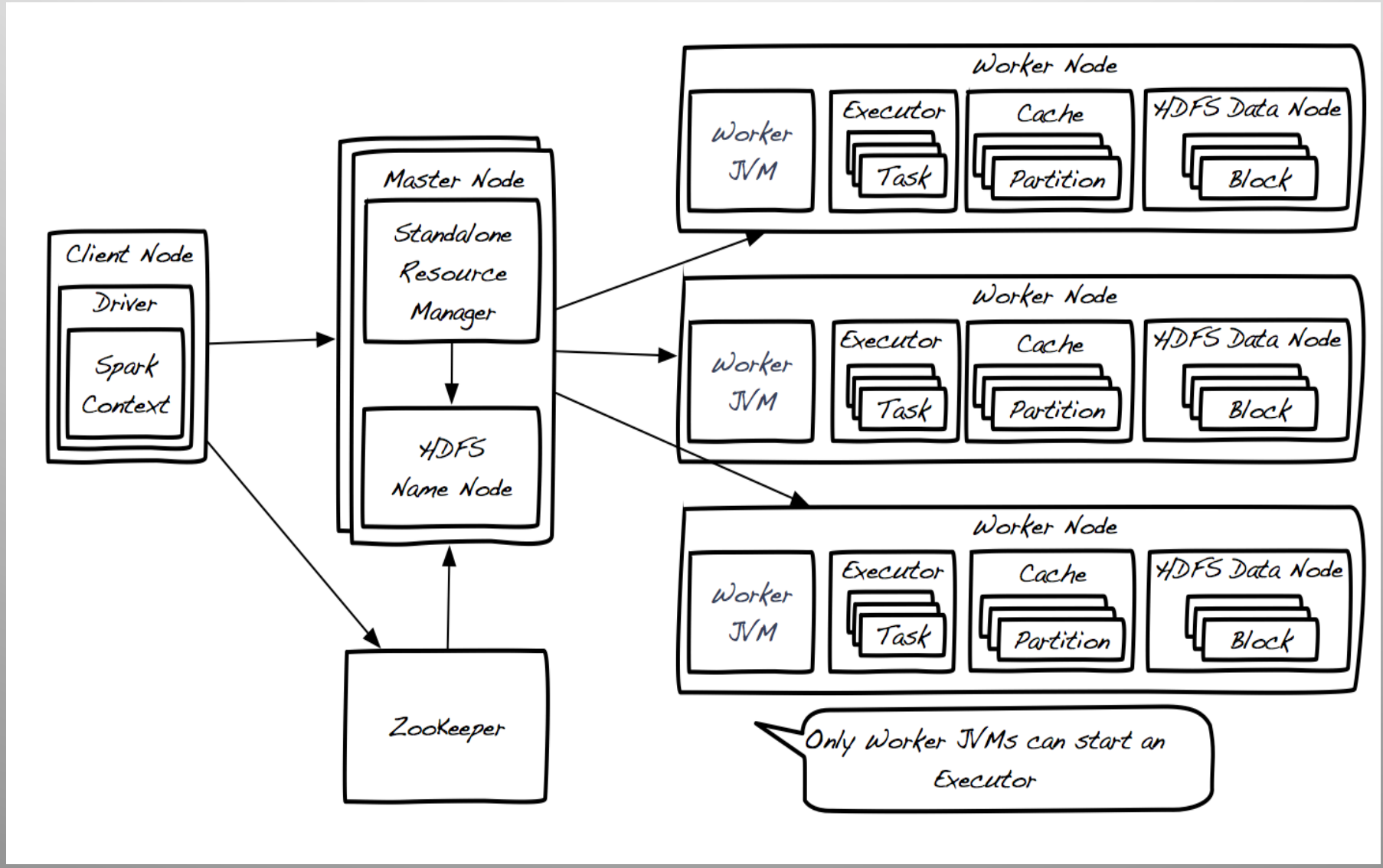
- Scaling up/down based on resource idle threshold
- Common problem
  - Long-running apps hold resources even when idle
  - Spark Shell is a long-running Spark app
  - Spark Streaming is a long running Spark app
- Best practice
  - Scale up quickly (exponentionall)
  - Scale down slowly

Spark Cluster Deployments

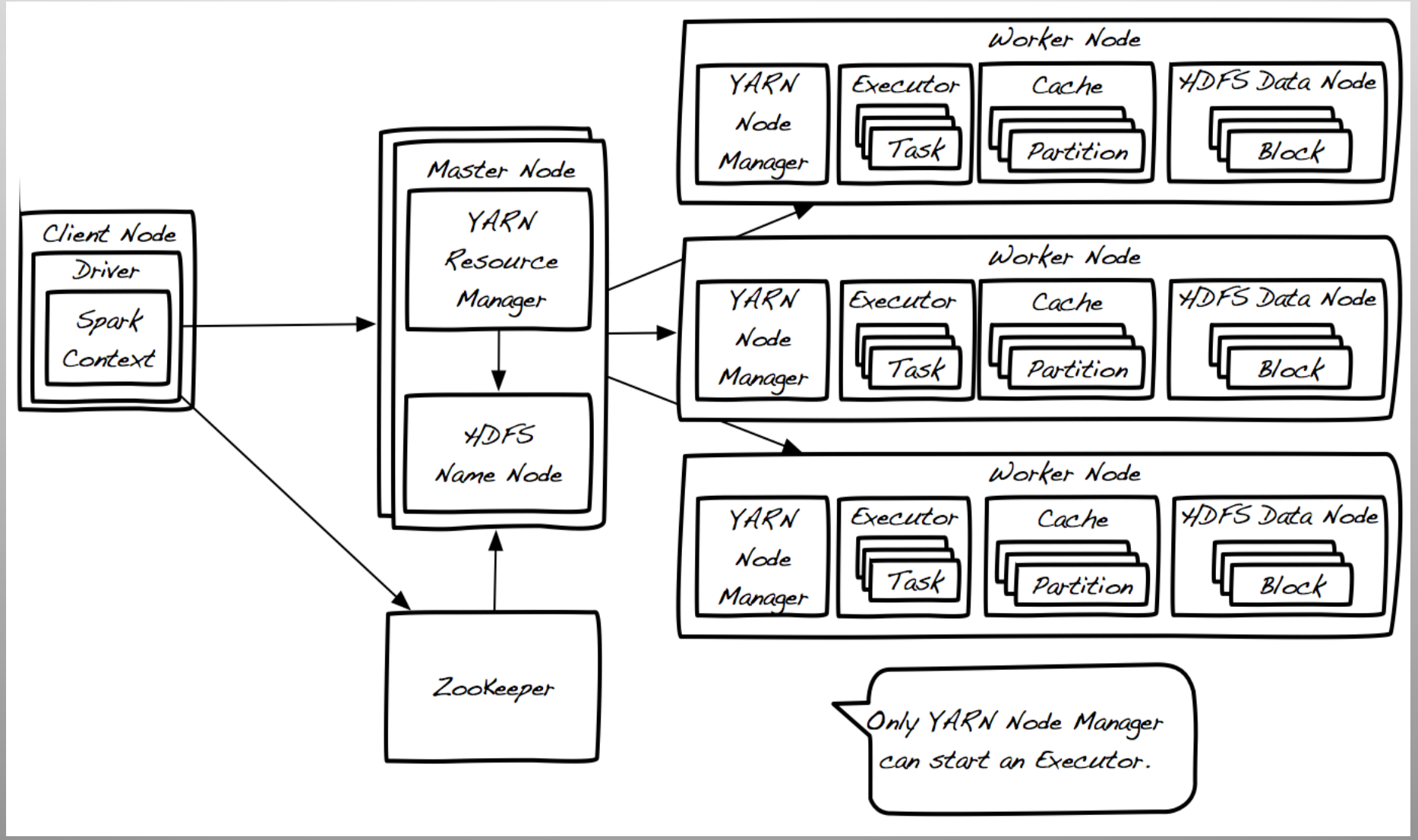
Cluster Resource Managers

- Spark Standalone (AMPLab, Spark default)
  - Suitable for a lot of production workloads
  - Only Spark workloads
- YARN (Hadoop)
  - Allows hierarchies of resources
  - Kerberos integration
  - Multiple workloads from different execution frameworks
    - Hive, Pig, Spark, MapReduce, Cascading, etc
- Mesos (AMPLab)
  - Similar to YARN, but allows elastic allocation to disparate execution frameworks
  - Coarse-grained
    - Single, long-running Mesos tasks runs Spark mini tasks
  - Fine-grained
    - New Mesos task for each Spark task
    - Higher overhead, not good for long-running Spark jobs (ie. Streaming)

Spark Standalone Deployment



Spark YARN Deployment



Spark Master High Availability

- Multiple Master Nodes
- ZooKeeper maintains current Master
- Existing applications and workers will be notified of new Master election
- New applications and workers need to explicitly specify current Master
- Alternatives (Not recommended)
  - Local filesystem
  - NFS Mount

Spark Library APIs

Spark SQL

HiveQL (v13)

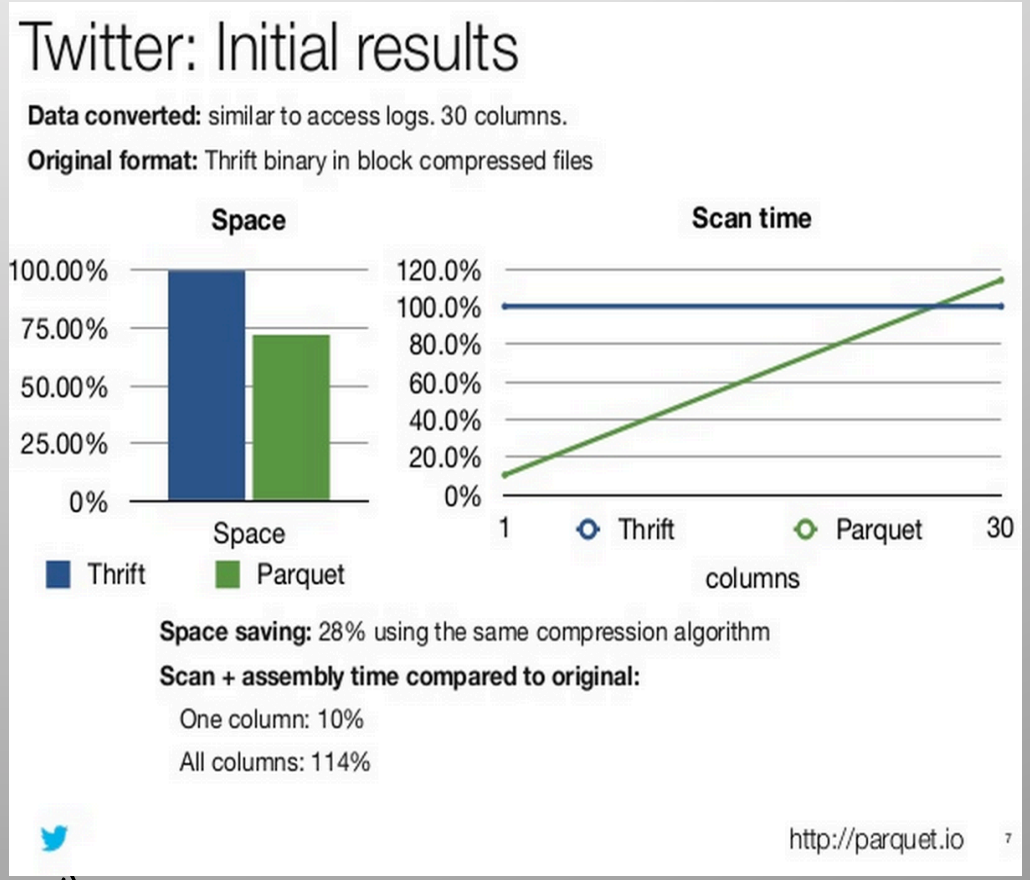
- Hive query statements, including:
  - `SELECT`
  - `GROUP BY`
  - `ORDER BY`
  - `CLUSTER BY`
  - `SORT BY`
- All Hive operators, including:
  - Relational operators (=, ⇔, ==, <>, <, >, >=, <=, etc)
  - Arithmetic operators (+, −, *, /, %, etc)
  - Logical operators (`AND`, &&, `OR`, ||, etc)
  - Complex type constructors
  - Mathematical functions (`sign`, `ln`, `cos`, etc)
  - String functions (`instr`, `length`, `printf`, etc)
- User defined functions (UDF)
- User defined aggregation functions (UDAF)
- User defined serialization formats (SerDes)
- Joins
  - `JOIN`
  - `{LEFT|RIGHT|FULL} OUTER JOIN`
  - `LEFT SEMI JOIN`
  - `CROSS JOIN`
- Unions
- Sub-queries
  - `SELECT col FROM ( SELECT a + b AS col from t1) t2`
- Sampling
- Explain
- Partitioned tables
- View
- All Hive DDL Functions, including:
  - `CREATE TABLE`
  - `CREATE TABLE AS SELECT`
  - `ALTER TABLE`
- Most Hive Data types, including:
  - `TINYINT`
  - `SMALLINT`
  - `INT`
  - `BIGINT`
  - `BOOLEAN`
  - `FLOAT`
  - `DOUBLE`
  - `STRING`
  - `BINARY`
  - `TIMESTAMP`
  - `DATE`
  - `ARRAY<>`
  - `MAP<>`
  - `STRUCT<>`

Hive In-Memory Caching

- In-memory columnar storage format
- Physically stores columns together (instead of rows)
- Optimized for fast column-based aggregations
  - SELECT AVG(age) FROM users
  - Not "SELECT *" which is row-based
- Predicate pushdowns (WHERE age >= 21)
- Column filtering (SELECT age FROM users)
- Partition pruning (WHERE date="2015-01")

Parquet File Format

- On-disk columnar storage format
- Heavy compression:
  - Run Length Encoding
  - Dictionary Encoding
  - Bit packing
- Nested structures
- Self-describing, evolving schema
- Optimized for:
  - Column-based aggregations
  - Data at rest
- Not optimized for:
  - SELECT * (row-based)
  - INSERTS/UPDATES (entire dataset needs re-encoding)



Demo: Rating Aggregations (SparkSQL, Parquet)

- Load CSV
- Convert to JSON
- Query JSON using SQL (slow)
- Convert to Parquet file
- Save Parquet file to disk
- Query Parquet using SQL (fast!)

MLlib and GraphX (Machine Learning and Graph Processing)

Machine Learning Algos

- Supervised
  - Prediction: Train a model with existing data + label, predict label for new data
    - Classification (categorical or numeric)
    - Regression (continuous numeric)
  - Recommendation: recommend to similars
    - User-user, item-item, user-item similarity
- Unsupervised
  - Clustering: Find natural clusters in data based on similarities (No training needed)
- Must be parallelizable!

ML Model Training

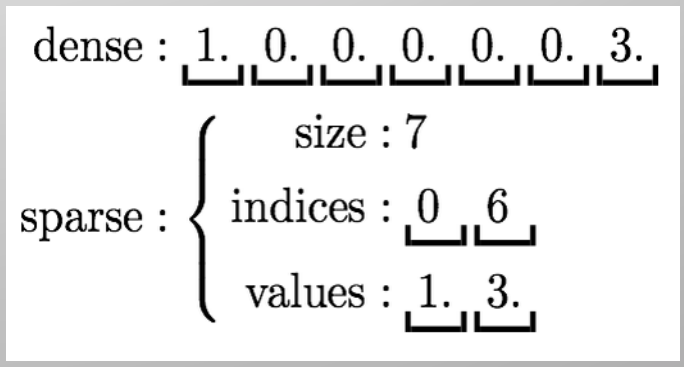
- Input Data (Instances)
  - Training data (~80%)
  - Hold-out data: model validation (~10%) & testing (~10%)
  - Shape/density: Tall & skinny, short & wide, dense vs. sparse?
- Features
  - Columns/attributes of instances
  - Types: Categorical, numeric (categorical & continuous)
  - Recommender: user id, product id
  - Clustering: height, religion, interest
  - Combine features to derive new ones
  - Requires domain knowledge
- Labels
  - Supervised: predictions, classifications
- Model Hyperparameters
  - Convergence threshold, number of iterations, etc

ML Model Validation & Testing

- Use hold-out data (20%)
  - Compare known validation label to model-predicted label
- Measure the difference with evaluation criteria:
  - Root Mean Squared Error (RMSE)
  - Mean Absolute Error (MAE)
- Optimize evaluation criteria
  - Choose the features and hyperparameters with the best evaluation criteria
  - ML Optimizer/ML Pipelines (AMPLab)
- K-folds validation
  - Split input data k-ways
  - Alternate the role of each split: training, validation, testing
  - Especially useful for smaller datasets
- Avoid overfitting to a particular dataset!

Shape and Density of Input

- Convert data to matrix
  - Rows (instances) x Columns (features)
- SparseVector vs. DenseVector
  - Sparse: O(12nnz+4);
    nnz = number of non-zeros
  - Dense requires O(8n);
    n = size of vector



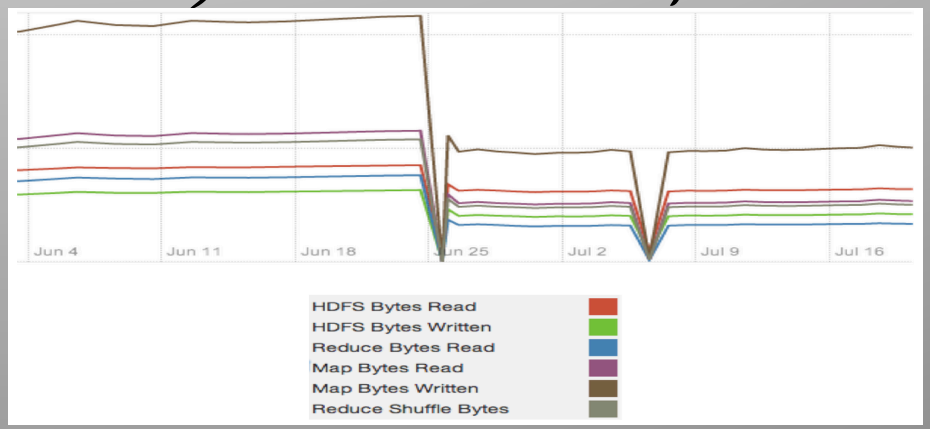
- SparseVector: wide rows
  - Lots of 0's (most large distributed datasets)
  - LIBSVM Format
- Some ML algos optimized for tall and skinny, some for short and wide
  - Depends on parallelism/shuffle characteristics

Categorical Features

- ML algos require data to be numeric (Double)
- Categorical is usually not numeric
  - ie. NFL Teams: Bears, 49`ers, Steelers
- Index? 1->Bears 2->49`ers 3-> Steelers
  - No! ML algo will assign irrelevant meaning to number
- Instead, use "one-hot" or "1 of n" encoding
  - Convert categories to unique combos of 0`s and 1`s
  - ie. Bears -> 1,0,0
    49`ers -> 0,1,0
    Steelers-> 0,0,1
- Lots of open Spark PR`s for one-hot impls

Similarity Algos (1/2)

- DIMSUM
  - "Dimension Independent Matrix Square Using MapReduce"
  - From Twitter
  - Merged into MLlib as RowMatrix.columnSimilarities()
  - Similar to Cosine Similarity, but overcomes O(m*n^2) shuffle by sampling
  - Becomes O(n^2/Epsilon^2); n = columns, Epsilon is threshold
  - Sampling requires shuffling
  - 40% improvement in Twitter Ad similarity computation



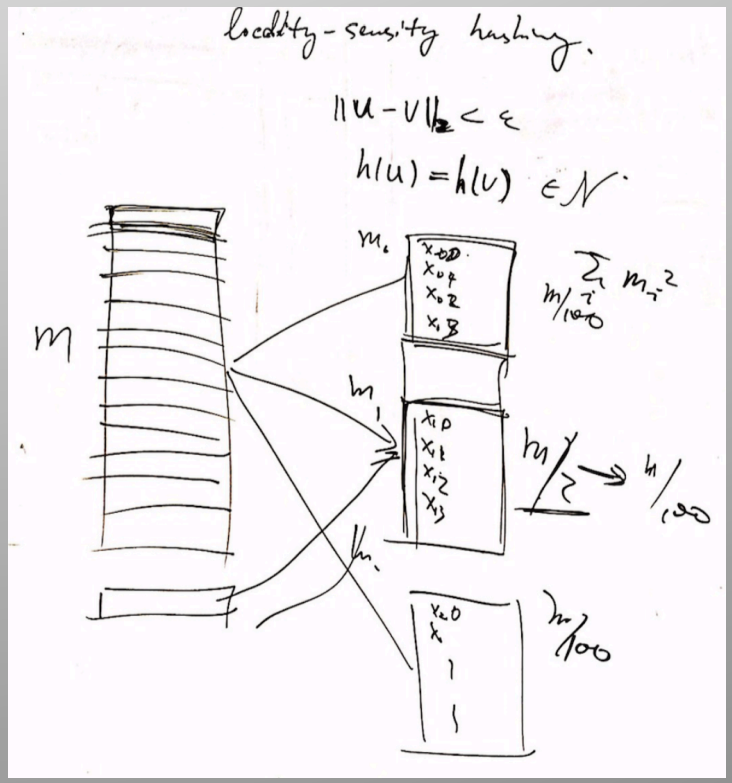
https://blog.twitter.com/2014/all-pairs-similarity-via-dimsum

Similarity Algos (2/2)

- LSH
  - Locality-Sensitive Hash
  - Hashes values and creates buckets of similar values
  - Hash function takes into account similarity using euclidean distance
  - Requires pre-processing to create buckets
  - Could suffer from skew
    - Reduce the space and return approximation
  - O(n^2) brute force w/o LSH
  - O(n/m * m^2) w/ LSH

Common LSH Use Case

- "I have a 500k x 500k matrix and I want to calculate pair-wise similarities"
  - O(n^2), cartesian, lots of shuffle!
- Use LSH!
  - Divide similar data into m buckets
  - Compare between buckets
  - Reduces down to O(n/m * m^2); where m = # of buckets
- O(250,000,000,000) vs. O(25,000,000); m=50

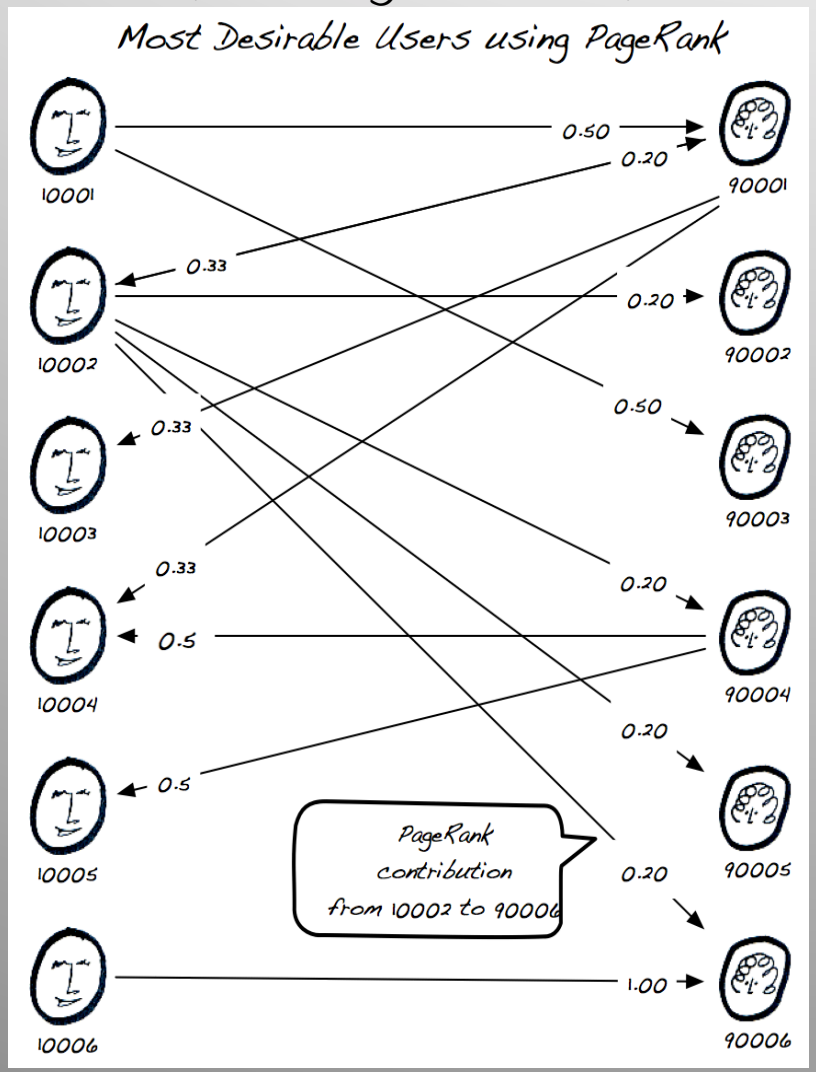


Serving Model Results

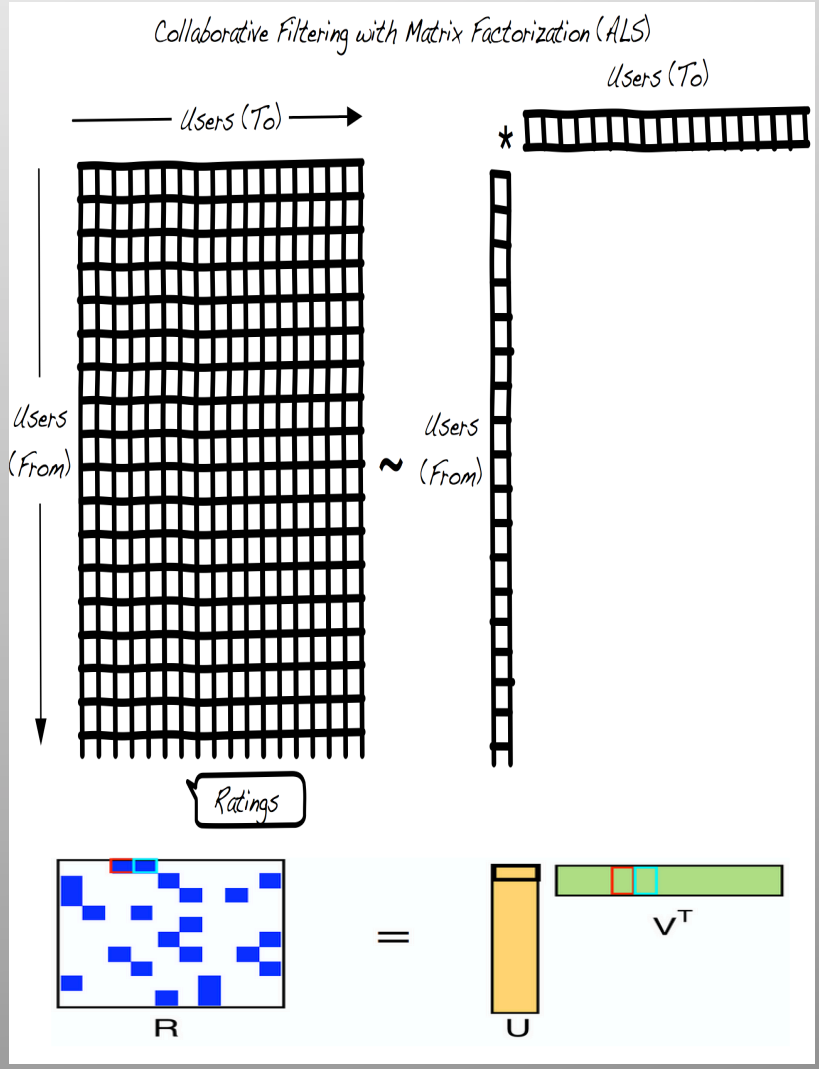
- Cold Start Problem
  - New users suffer cold start
  - Need suitable starting points
    - Top-rated users from Spark SQL
    - Most-desirable users from PageRank
- Randomization
  - Don't underestimate the power of randomizing
  - Need to surface results that wouldn't otherwise surface
- A/B test diff models to determine actual effectiveness
- Combine model results
  - Track which model the result came from
  - Remember that position in results matters

Demo: Generating Matches (PageRank & ALS)

GraphX: PageRank (TopK)

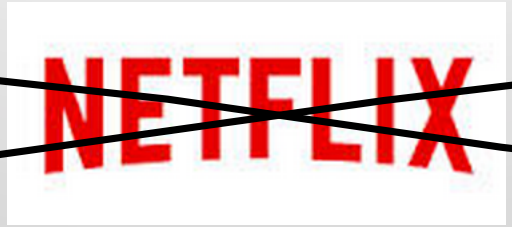


MLlib: ALS (Recommend)



Spark Streaming

"Streaming"



Video Streaming

Piping

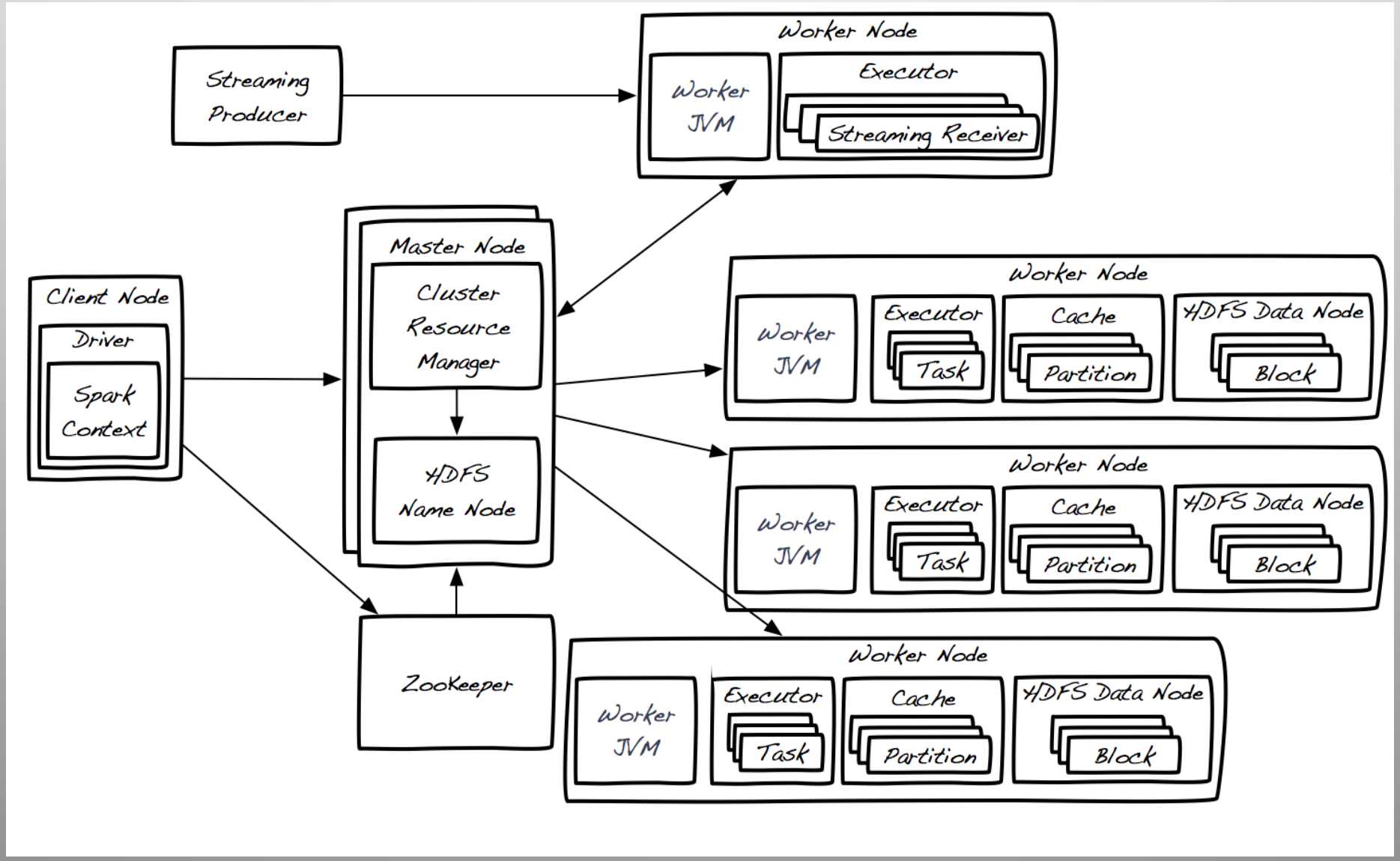
Big Data Streaming

Kinesis

Spark Streaming Overview

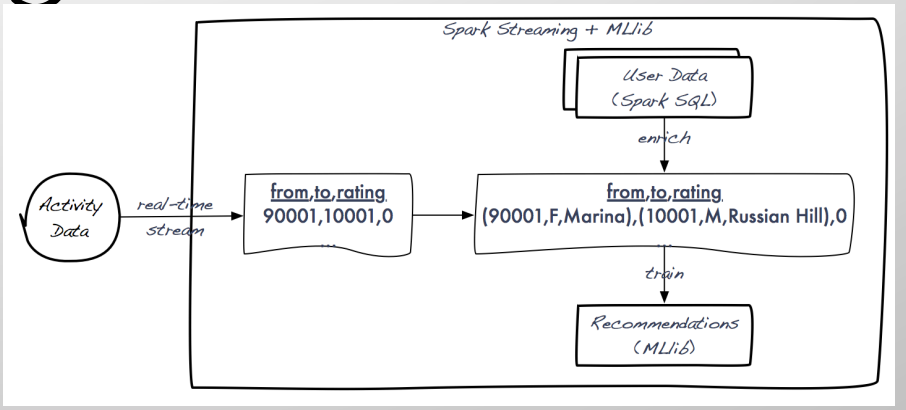
- Low latency, high throughput, fault-tolerance (mostly)
- Long-running Spark application
- Supports Flume, Kafka, Twitter, Kinesis, Socket, File, etc.
- Graceful shutdown, in-flight message draining
- Uses Spark Core, DAG Execution Model, and Fault Tolerance
- Submits micro-batch jobs to the cluster

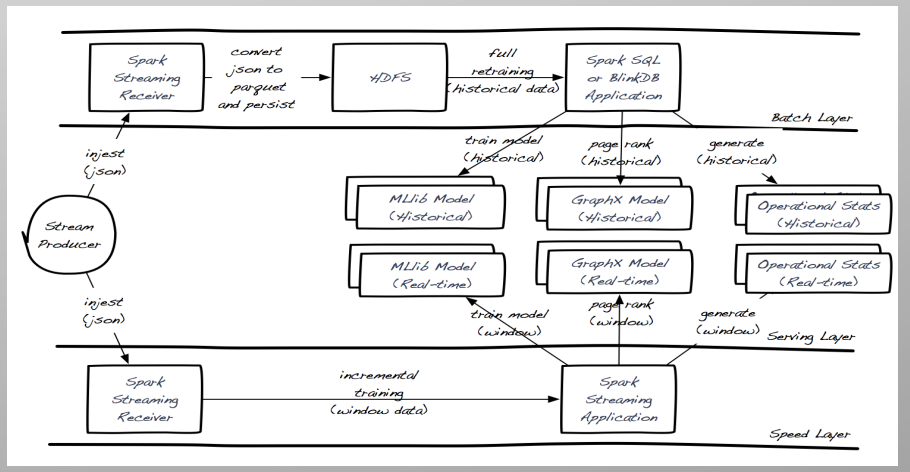
Spark Streaming Overview

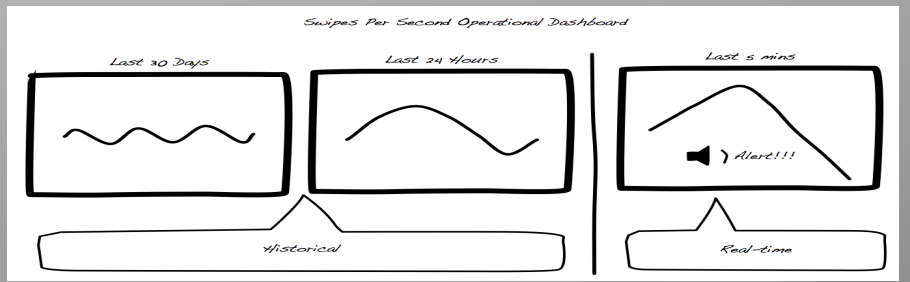


Spark Streaming Use Cases

- ETL and enrichment of streaming data on injest
  - Spark Streaming, Spark SQL, MLlib
- Lambda Architecture
  - Spark Streaming, Spark SQL, MLlib, GraphX
  - Everything!!
- Operational dashboards
  - Lambda Architecture
  - Spark Streaming, Spark SQL

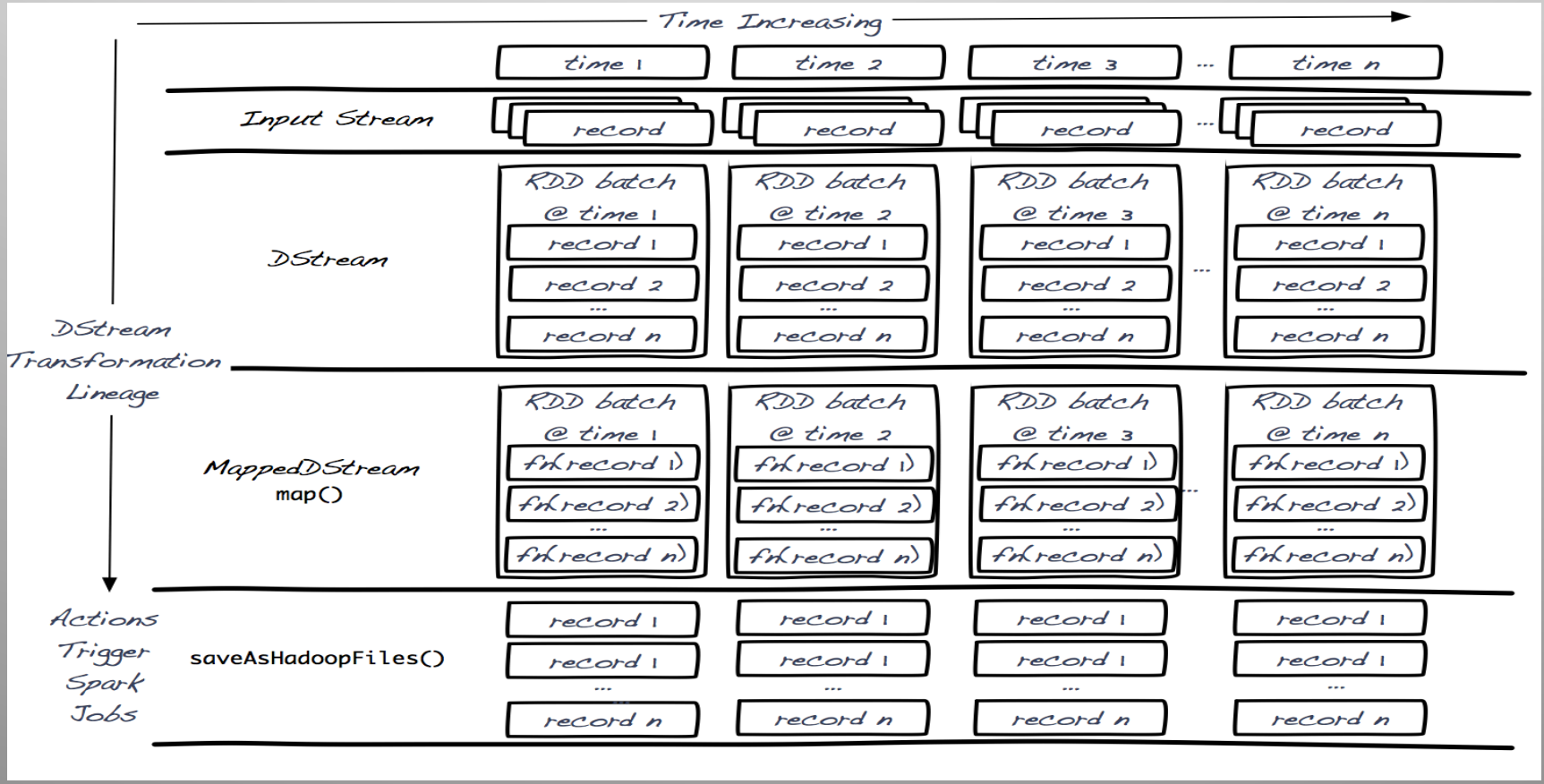






Discretized Stream (DStream)

- Core Spark Streaming abstraction
- Micro-batches of RDDs
- Operations similar to RDD
- Operates on all underlying RDDs of DStream



Spark Streaming API Overview

- Rich, expressive API similar to core
- Operations
  - Transformations (lazy)
  - Actions (execute transformations)
- Window and Stateful Operations
  - Current state depends on previous state
- Requires 2 types of durable checkpointing (HDFS) to snip long-running lineages
  - Metadata: DStream DAG, queue of jobs,
  - Data: input data, window and stateful data
- Register DStream as a Spark SQL table for querying?! Wow.

DStream Transformations

Transformations

DStreams support many of the transformations available on normal Spark RDD's. Some of the common ones are as follows.

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property `spark.default.parallelism`) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.
join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
cogroup(*otherStream*, [*numTasks*])	When called on DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
transform(*func*)	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
updateStateByKey(*func*)	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

DStream Actions

Output Operations

When an output operator is called, it triggers the computation of a stream. Currently the following output operators are defined:

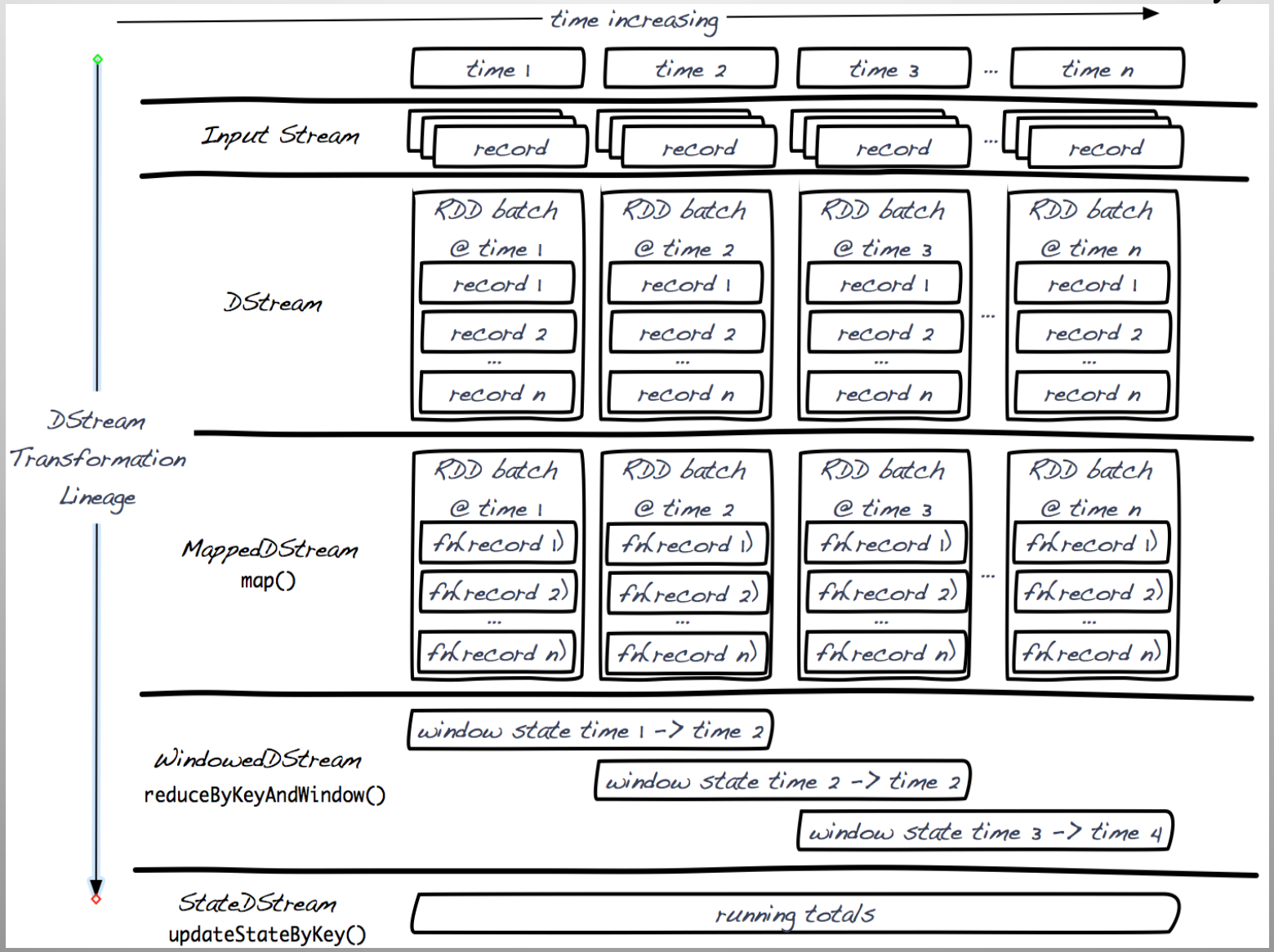
Output Operation	Meaning
print()	Prints first ten elements of every batch of data in a DStream on the driver.
foreachRDD(*func*)	The fundamental output operator. Applies a function, *func*, to each RDD generated from the stream. This function should have side effects, such as printing output, saving the RDD to external files, or writing it over the network to an external system.
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as a `SequenceFile` of serialized objects. The file name at each batch interval is generated based on *prefix* and *suffix*: "*prefix-TIME_IN_MS[.suffix]*".
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as a text files. The file name at each batch interval is generated based on *prefix* and *suffix*: "*prefix-TIME_IN_MS[.suffix]*".
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as a Hadoop file. The file name at each batch interval is generated based on *prefix* and *suffix*: "*prefix-TIME_IN_MS[.suffix]*".

Window and State DStream Operations

Transformation	Meaning
window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.
countByWindow(*windowLength*, *slideInterval*)	Return a sliding window count of elements in the stream.
reduceByWindow(*func*, *windowLength*, *slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative so that it can be computed correctly in parallel.
reduceByKeyAndWindow(*func*, *windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function *func* over batches in a sliding window. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property `spark.default.parallelism`) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKeyAndWindow(*func*, *invFunc*, *windowLength*, *slideInterval*, [*numTasks*])	A more efficient version of the above `reduceByKeyAndWindow()` where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enter the sliding window, and "inverse reducing" the old data that leave the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable to only "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter *invFunc*. Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument.
countByValueAndWindow(*windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument.

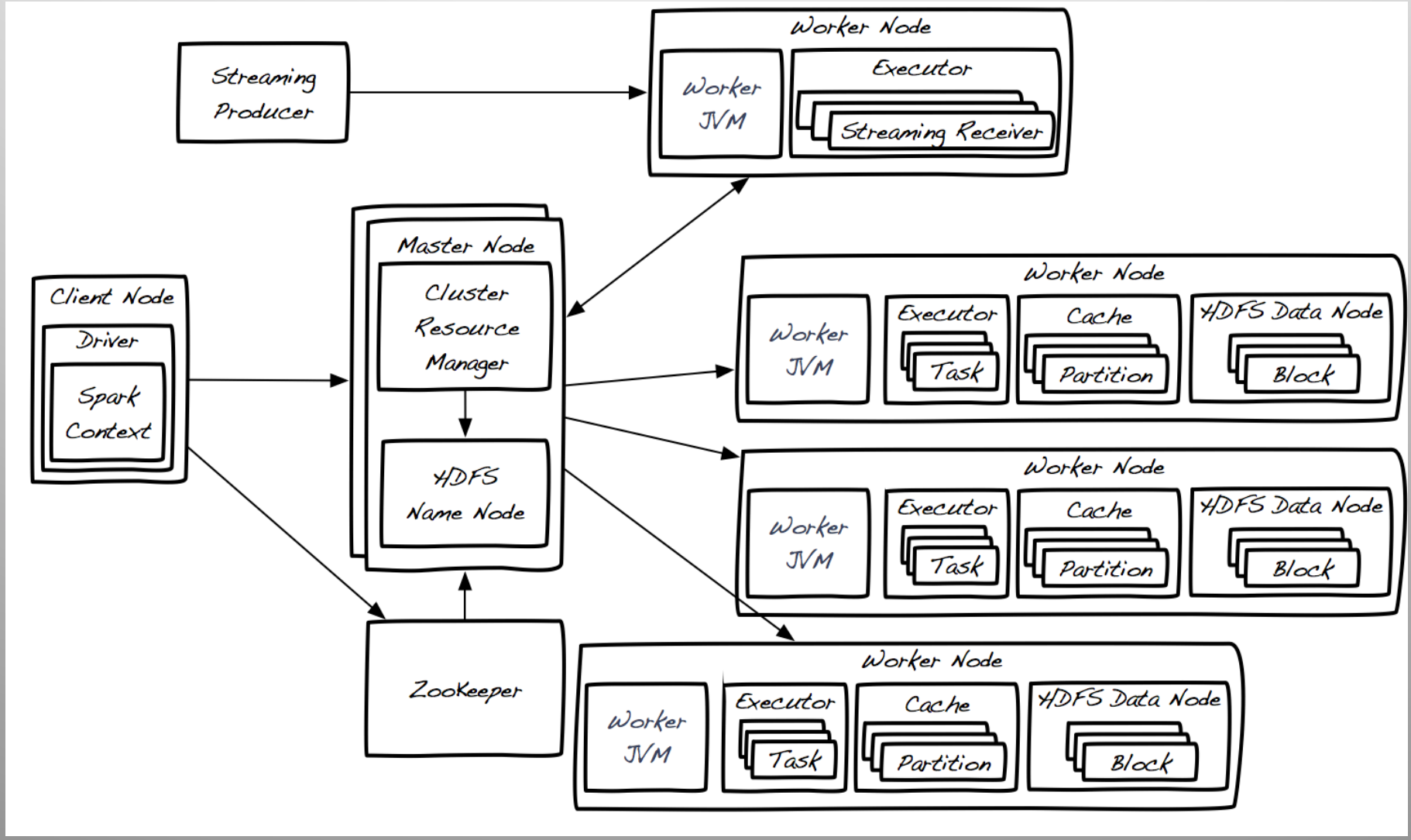
updateStateByKey(*func*)	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

DStream Window and State Example

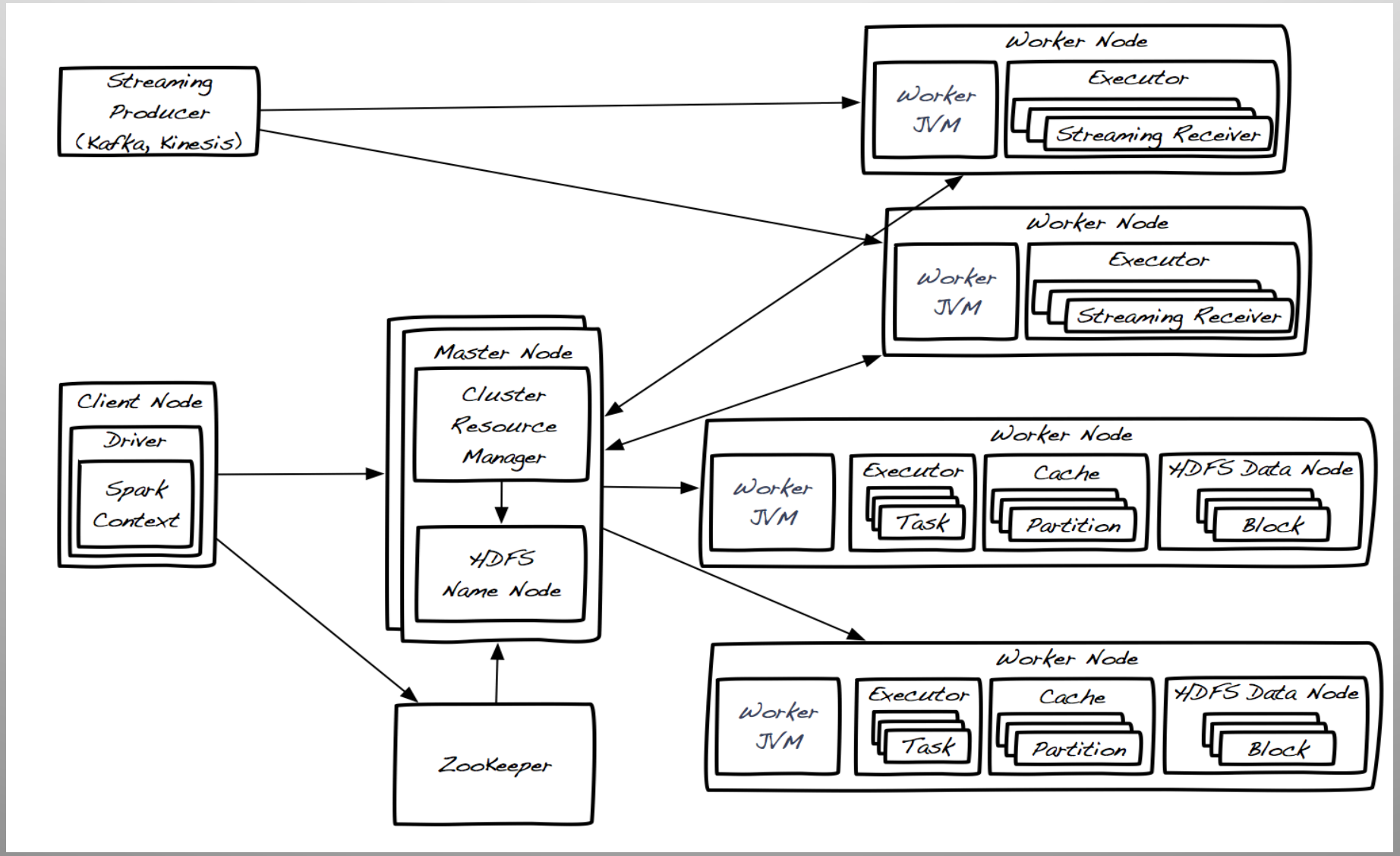


Spark Streaming Cluster Deployment

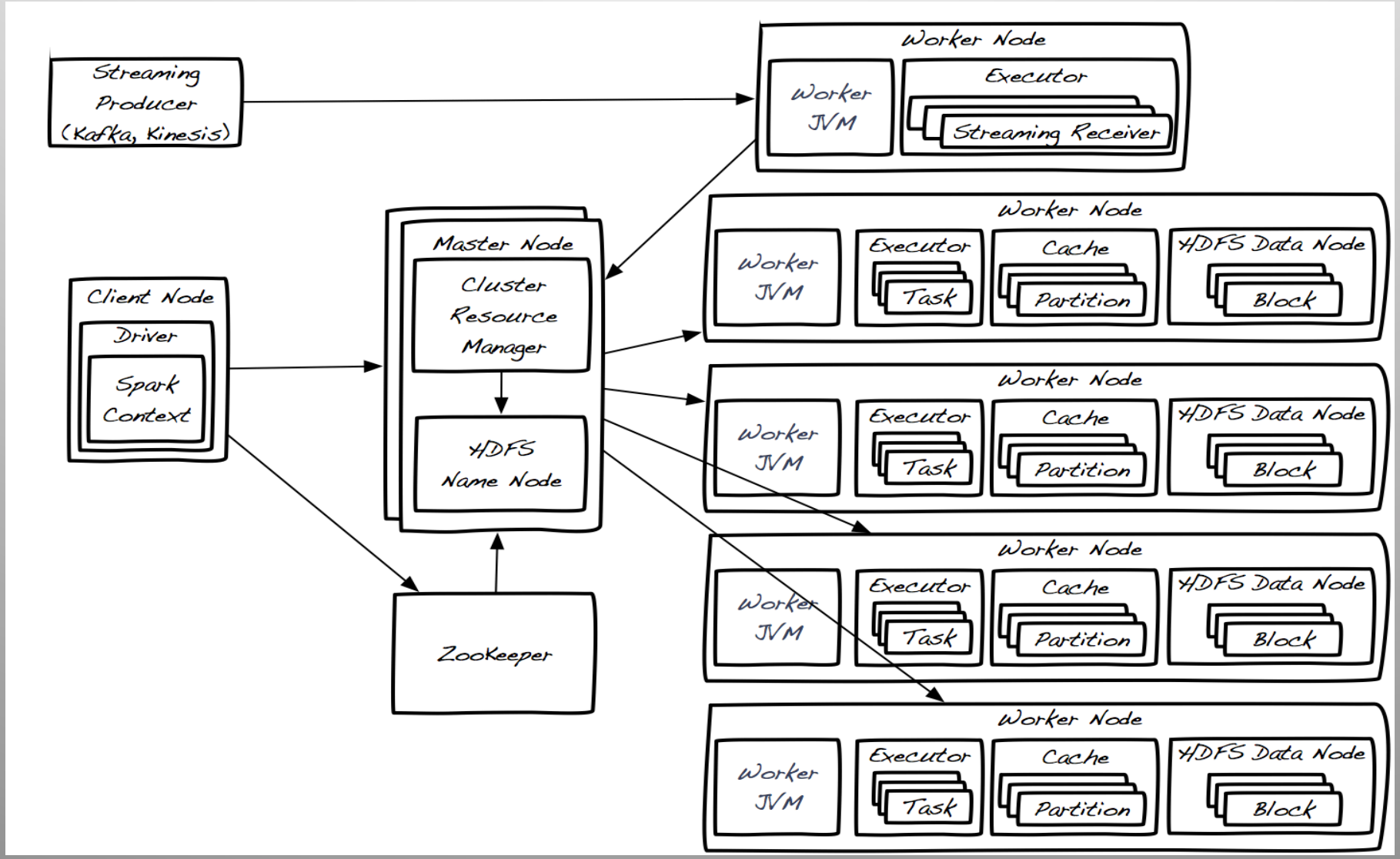
Streaming Cluster Deployment



Adding Receivers

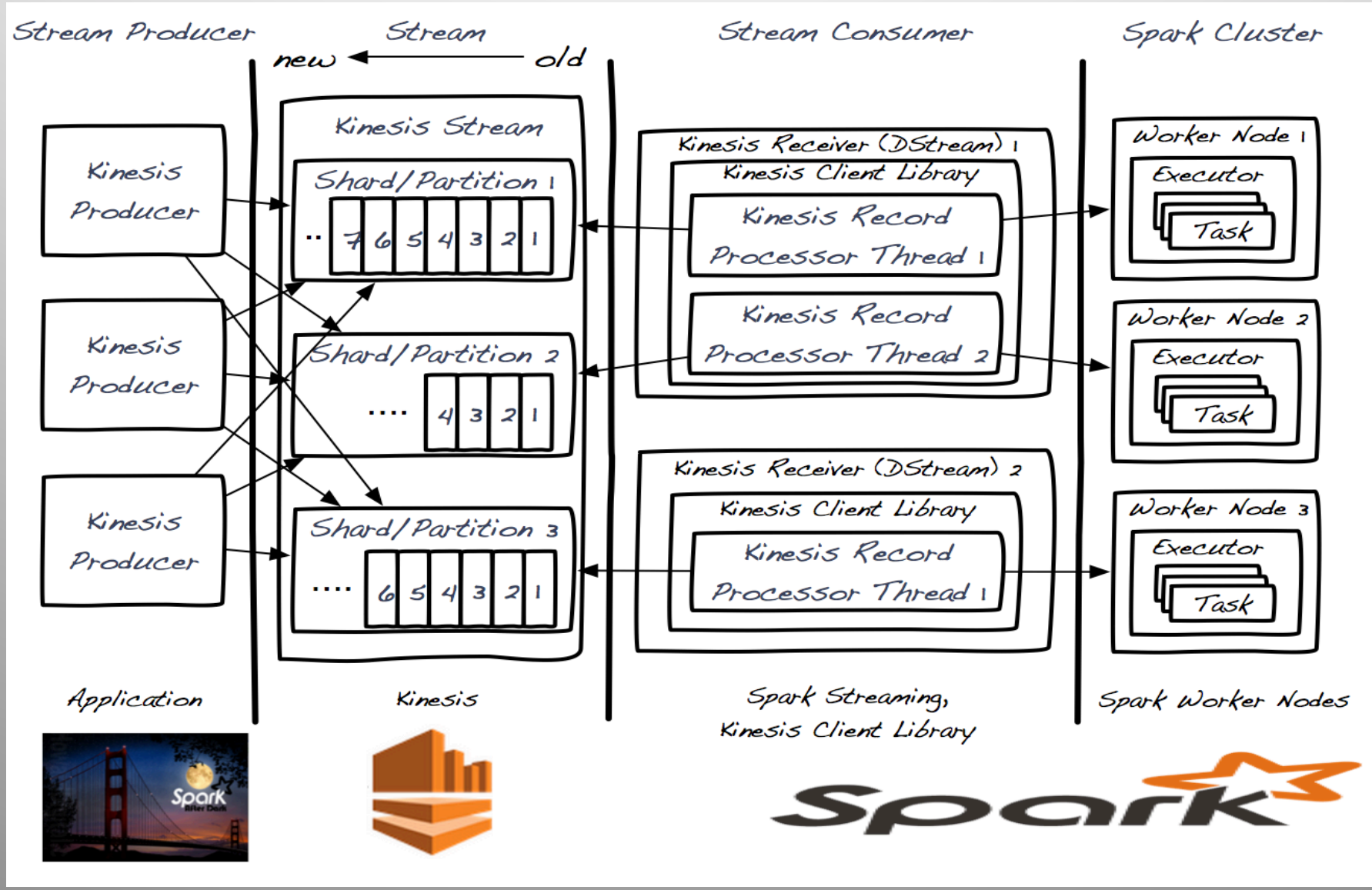


Adding Workers

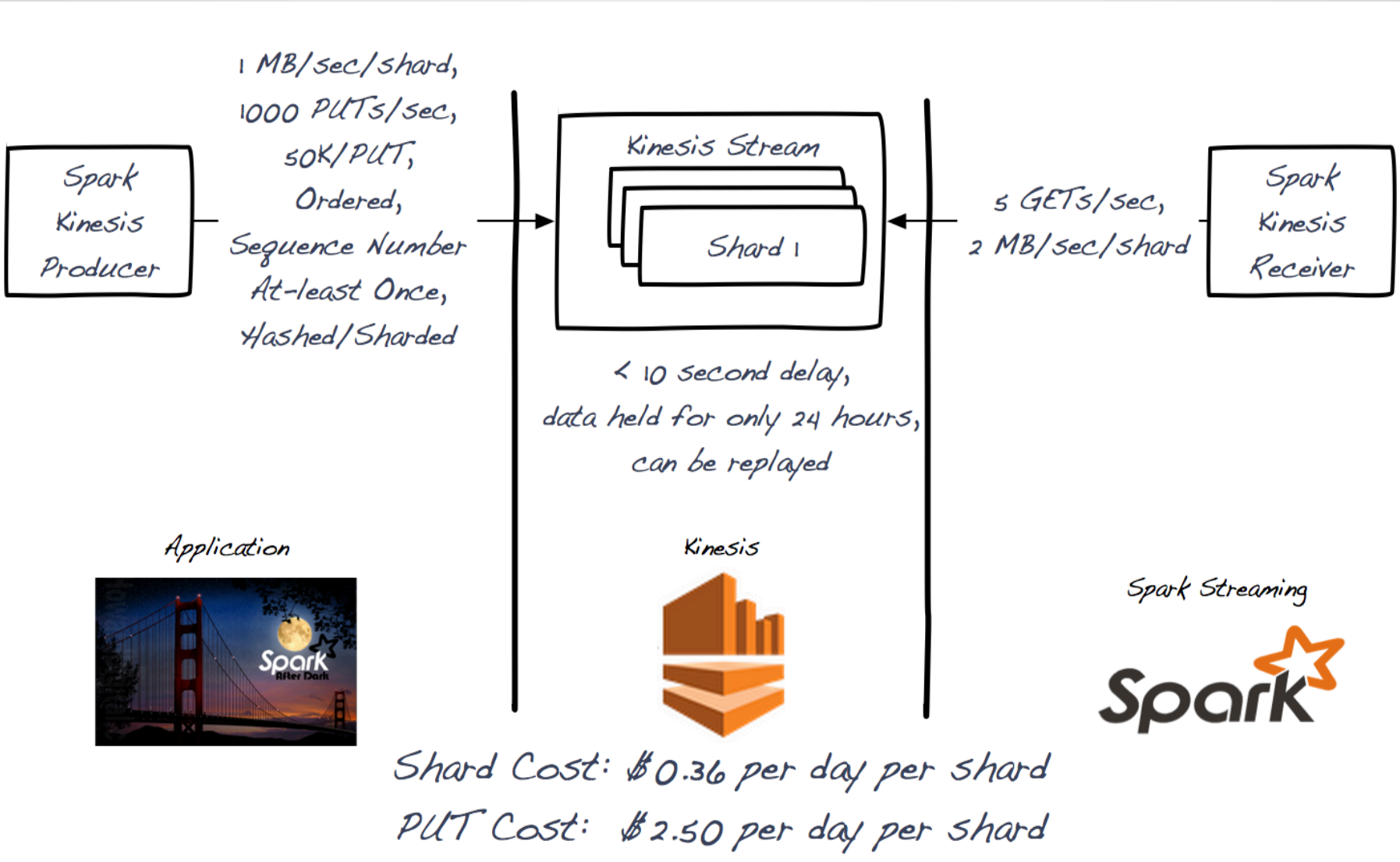


Spark Streaming
+
Kinesis

Spark Streaming + Kinesis Architecture



Throughput and Pricing



Demo: Real-time ALS Matches

https://github.com/apache/spark/blob/master/extras/kinesis-asl/src/main/…

Scala: …/scala/org/apache/spark/examples/streaming/KinesisWordCountASL.scala
Java: …/java/org/apache/spark/examples/streaming/JavaKinesisWordCountASL.java

Kinesis Best Practices

- Avoid resharding
  - Over-provision shards to avoid splitting/merging & increase throughput
  - Remember to include egress rate when provisioning - not just ingest
  - Choose high-cardinality partition hash key to avoid hot spots/shards (ie. CustomerID, etc)
- Avoid excess PUTs
  - Batch vs. single records
  - Kinesis Mobile Client supports batch to reduce battery consumption
  - Increases throughput
  - Filter on client-side
  - Async producer like FluentD, Log4J, Flume
- Avoid data loss
  - Use an archiver consumer app to move the data into S3 for longer-term durability past 24 hours
  - Write to durable storage (Spark 1.2 WAL) before checkpointing
  - De-dupe as deep into the pipeline as possible - allows retry
  - Use deterministic checkpoint strategy to get repeatable failover results (ie. number of records since last checkpoint)
  - Idempotent processing where possible

Spark Streaming Fault Tolerance

Types of Streaming Sources

- Batched
  - Flume, Kafka, Kinesis
  - Improves throughput at expense of possible duplication during failure replay
- Partitioned
  - Kafka, Kinesis
  - Different receivers dedicated to different partitions
  - Improves throughput through parallelism
  - Order is only guaranteed within a single partition
- Buffered
  - Flume, Kafka, Kinesis
  - Allows back pressure, rate limiting, etc
- Reliable (Checkpointed)
  - Kafka, Kinesis
  - Allows replay from specific checkpoint
  - Requires ACK after each batch is durably stored within Spark Streaming
- Transactional
  - Flume
  - Start, Commit semantics

Message Delivery Guarantees

- Exactly once [1]
  - No loss
  - No redeliver
  - Perfect delivery
  - Incurs higher latency for transactional semantics
  - Spark default per batch using DStream lineage
  - Degrades to less guarantees depending on source
- At least once [1..n]
  - No loss
  - Possible redeliver
- At most once [0,1]
  - Possible loss
  - No redeliver
  - *Best configuration if some data loss is acceptable
- Ordered
  - Per partition: Kafka, Kinesis
  - Global across all partitions: Hard to scale

Types of Checkpoints

Spark

1. Metadata: Spark checkpointing of StreamingContext DStreams and metadata
2. Data: Lineage of input data, state, and window DStream operations

Kinesis

3. Kinesis Client Library (KCL) checkpoints current position within shard
   - Checkpoint info is stored in DynamoDB per Kinesis application keyed by shard

Fault Tolerance

- Points of Failure
  - Receiver
    - Should restart, replace automatically
    - Possible data loss if not using Write Ahead Log (WAL)
  - Driver
    - Application is torn down including worker allocations
  - Worker/Executor
    - Normal Spark partition-level recovery occurs
- Possible Solutions
  - Use HDFS File Source for durability
  - Data Replication
  - Secondary/Backup Nodes
  - Checkpoints
    - Stream, Window, and State info

Streaming Receiver Failure

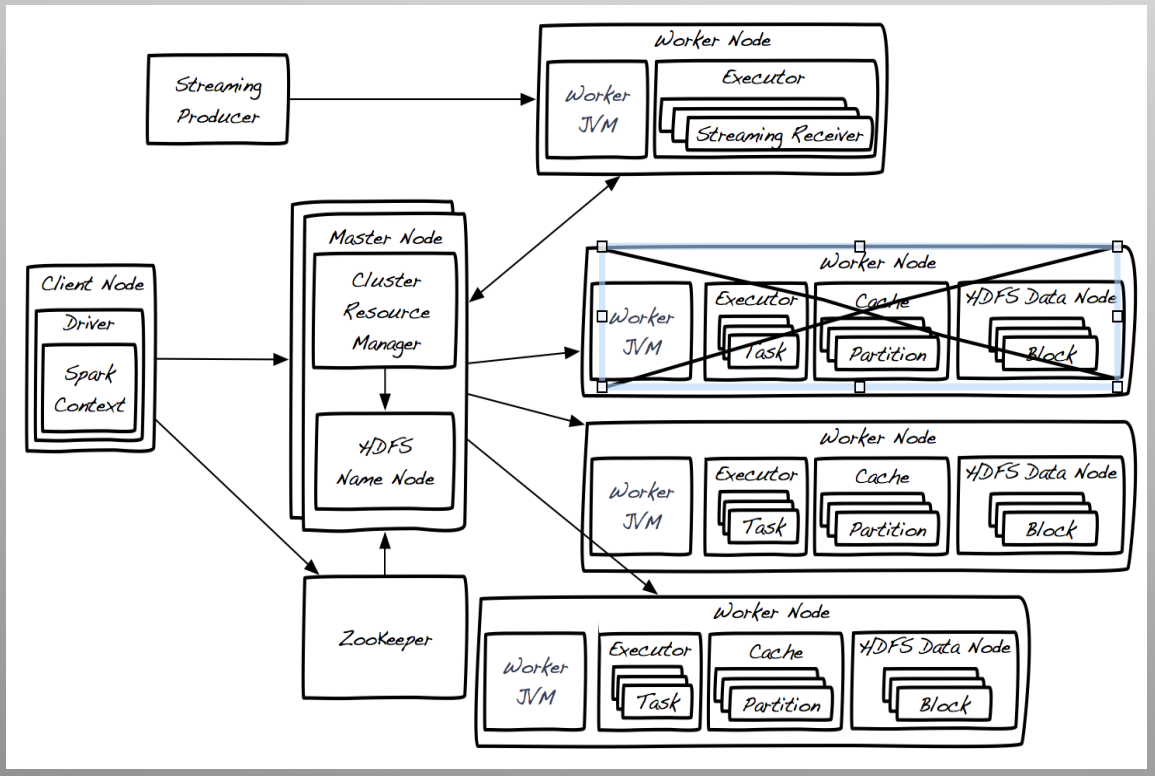
- Use multiple receivers pulling from multiple shards
- Use checkpoint-enabled, reliable, sharded source (ie. Kafka, Kinesis) for replay
- Data is replicated to 1 other node in-memory immediately upon ingestion
  - 2nd node is chosen at random
  - Spills to disk if data doesn't fit in memory
- Write Ahead Log (WAL)
  - Prevents losing in-flight metadata and data (received, but not processed)
  - Stored in HDFS
  - ACK back to source when safely stored in WAL
  - Spark processes the stored-but-not-processed data upon failover

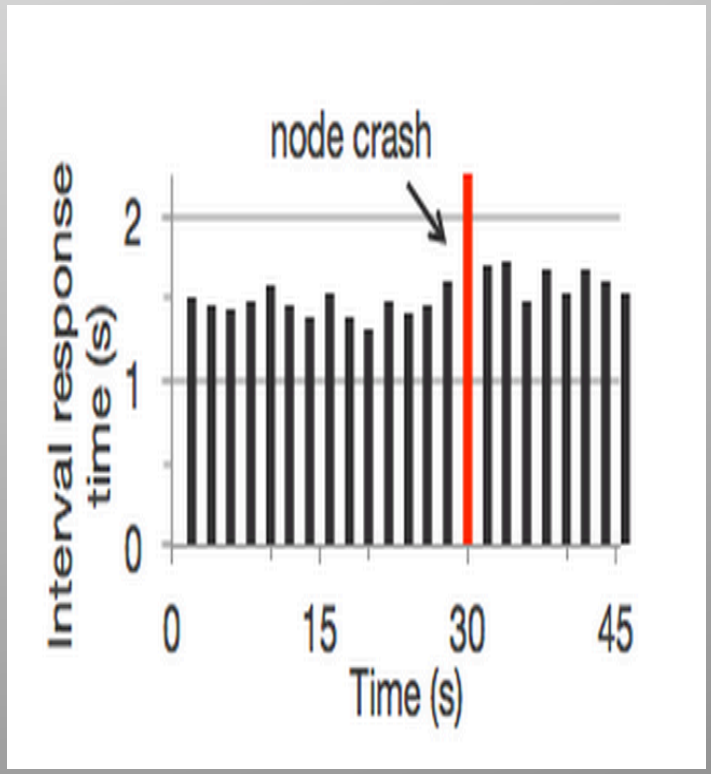
Streaming Driver Failure

- Use a backup Driver
  - Use DStream metadata checkpoint info to recover
  - StreamingContext.getOrCreate()
- Single point of failure
  - Interrupts stream processing
  - State and Window data is lost
  - Use checkpointing to avoid data loss
- Streaming Driver is a long-running Spark application
  - Schedules long-running receivers
  - Checkpointing prevents infinite DStream lineage for long-running streaming app

Stream Worker/Processor Failure

- No problem!
- DStream RDD partitions will be recalculated from lineage
- Causes temporary processing blip during failover



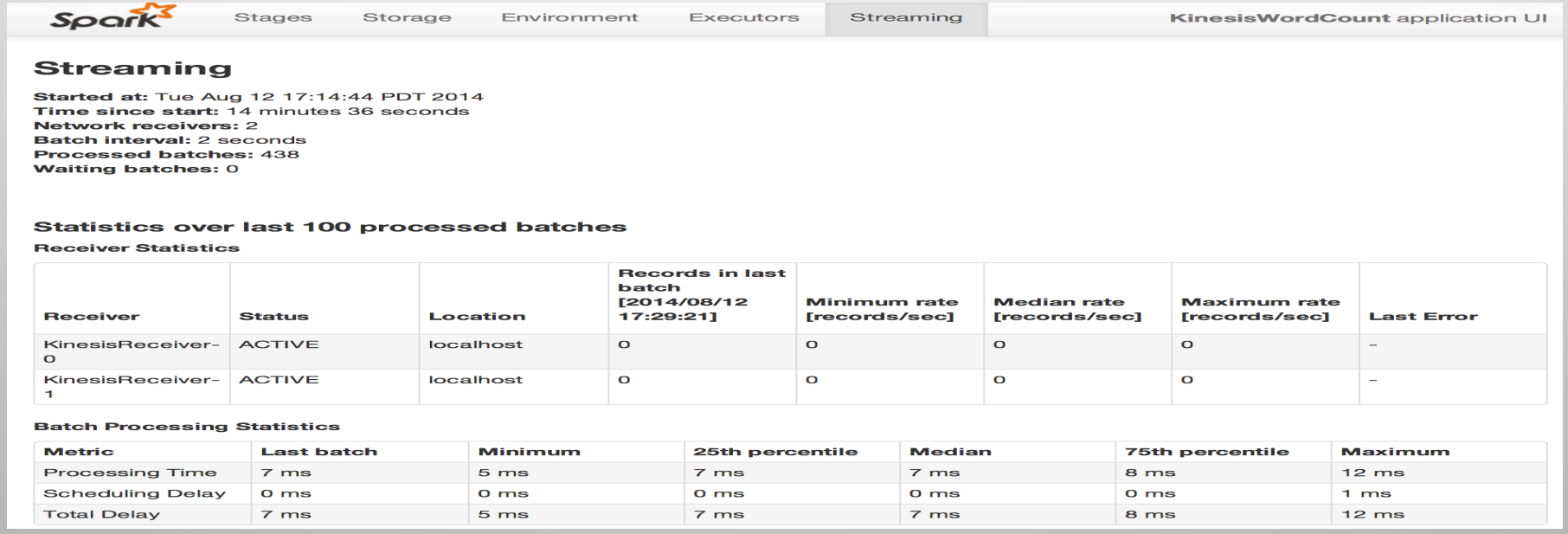


Spark Streaming Monitoring and Tuning

Monitoring

- Monitor driver, receiver, worker nodes, and streams
- Alert upon failure or unusually high latency
- Spark Web UI
  - Streaming tab
- Ganglia, CloudWatch (AWS)
- Custom StreamingListener callback hooks during DStream processing

Spark Web UI



Spark | Stages | Storage | Environment | Executors | Streaming | KinesisWordCount application UI

Streaming

Started at: Tue Aug 12 17:14:44 PDT 2014
Time since start: 14 minutes 36 seconds
Network receivers: 2
Batch interval: 2 seconds
Processed batches: 438
Waiting batches: 0

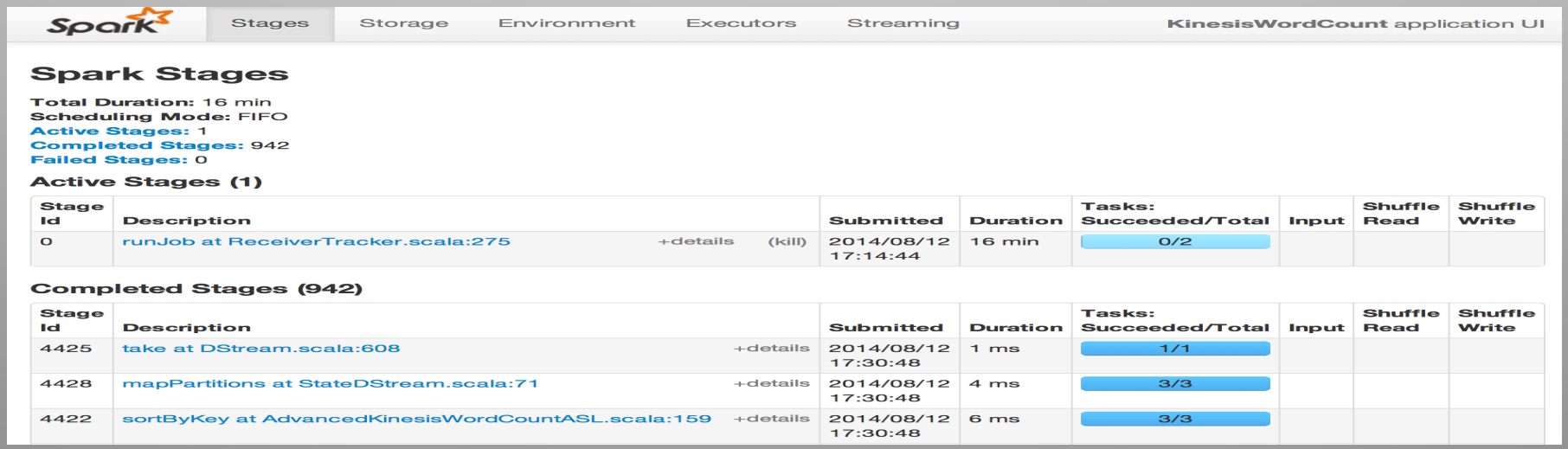
Statistics over last 100 processed batches

Receiver Statistics

Receiver	Status	Location	Records in last batch [2014/08/12 17:29:21]	Minimum rate [records/sec]	Median rate [records/sec]	Maximum rate [records/sec]	Last Error
KinesisReceiver-0	ACTIVE	localhost	0	0	0	0	-
KinesisReceiver-1	ACTIVE	localhost	0	0	0	0	-

Batch Processing Statistics

Metric	Last batch	Minimum	25th percentile	Median	75th percentile	Maximum
Processing Time	7 ms	5 ms	7 ms	7 ms	8 ms	12 ms
Scheduling Delay	0 ms	0 ms	0 ms	0 ms	0 ms	1 ms
Total Delay	7 ms	5 ms	7 ms	7 ms	8 ms	12 ms



Spark | Stages | Storage | Environment | Executors | Streaming | KinesisWordCount application UI

Spark Stages

Total Duration: 16 min
Scheduling Mode: FIFO
Active Stages: 1
Completed Stages: 942
Failed Stages: 0

Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
0	runJob at ReceiverTracker.scala:275 +details (kill)	2014/08/12 17:14:44	16 min	0/2			

Completed Stages (942)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
4425	take at DStream.scala:608 +details	2014/08/12 17:30:48	1 ms	1/1			
4428	mapPartitions at StateDStream.scala:71 +details	2014/08/12 17:30:48	4 ms	3/3			
4422	sortByKey at AdvancedKinesisWordCountASL.scala:159 +details	2014/08/12 17:30:48	6 ms	3/3			

Tuning (1/2)

- DStream batch time interval
  - Higher: reduces overhead by submitting less tasks overall
  - Lower: keeps latencies low
  - Sweet spot: DStream job time (scheduling + processing) is steady and less than batch interval

- Checkpoint time interval
  - Higher: reduces load on checkpoint overhead
  - Lower: reduces amount of data loss on failure
  - Recommendation: 5-10x sliding window interval

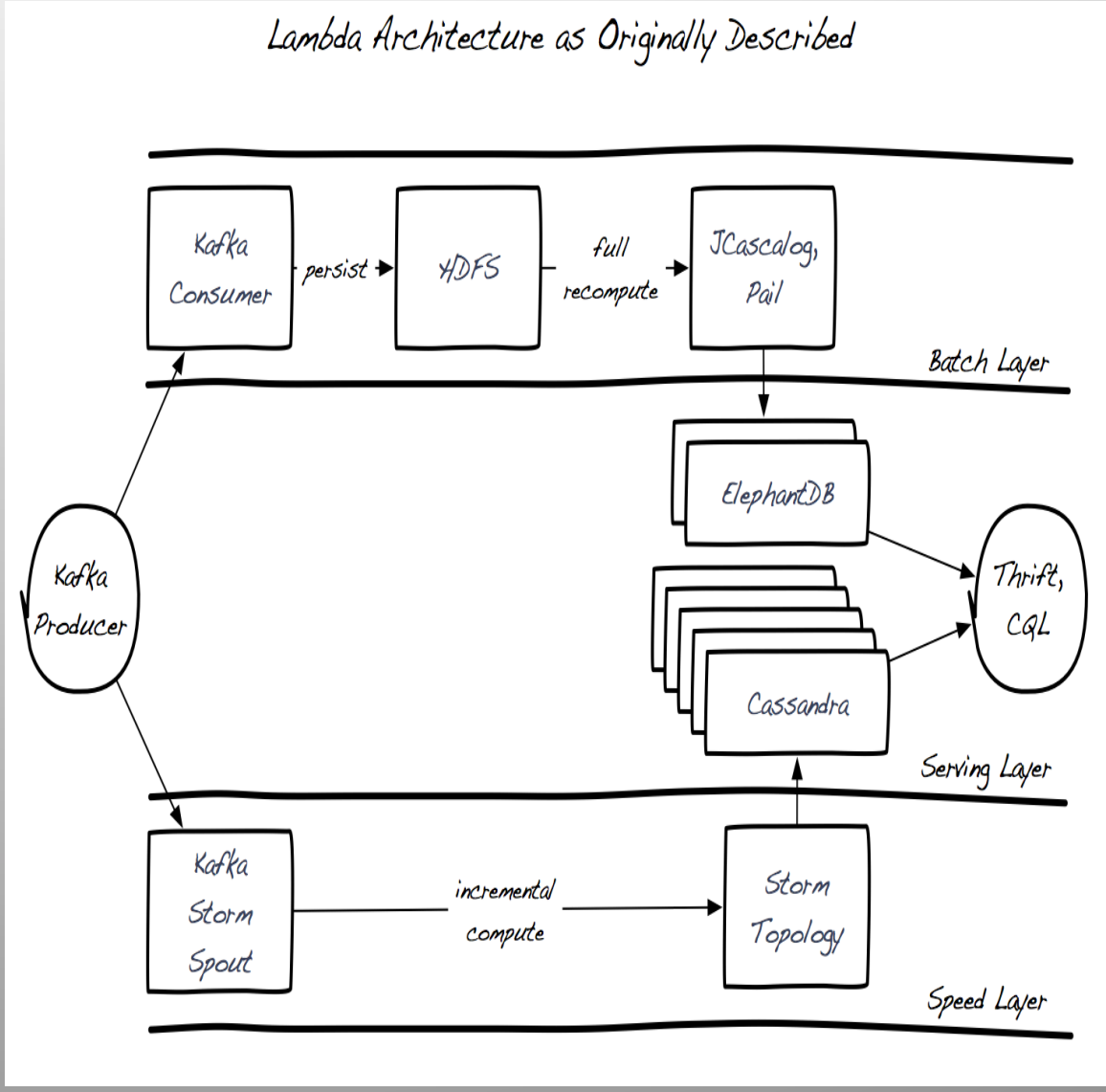
Tuning (2/2)

- Use DStream.repartition()
  - Increase parallelism of processing DStream jobs across more cluster nodes
- Use spark.streaming.unpersist=true
  - Let's the Streaming Framework figure out when to unpersist out-of-scope resources
- Use Concurrent Mark and Sweep (CMS) GC Strategy
  - For consistent processing times
  - Note: Default stores data as serialized byte arrays to minimize GC
- Consistency is key!!

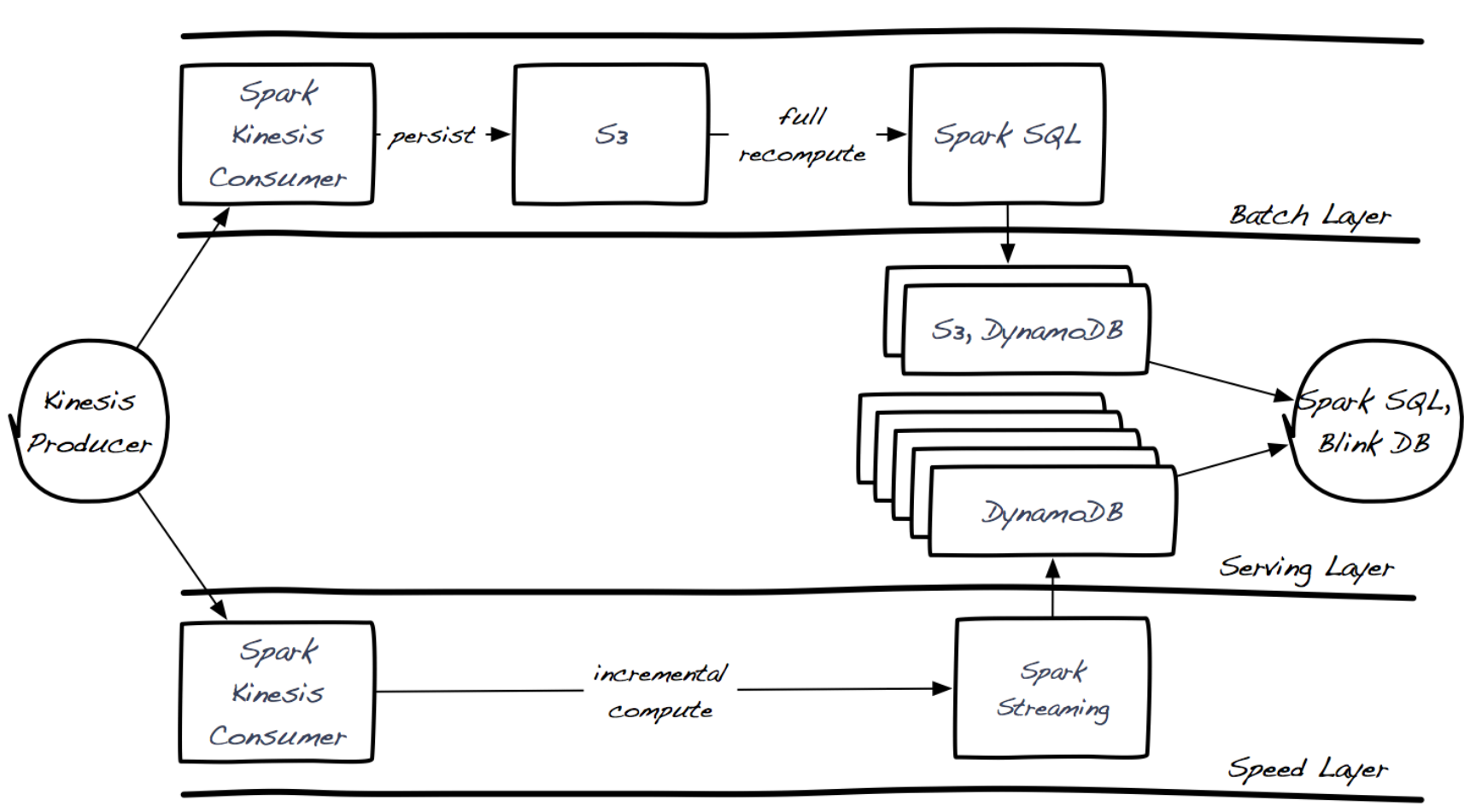
Spark and the Lambda Architecture

Lambda Architecture Overview

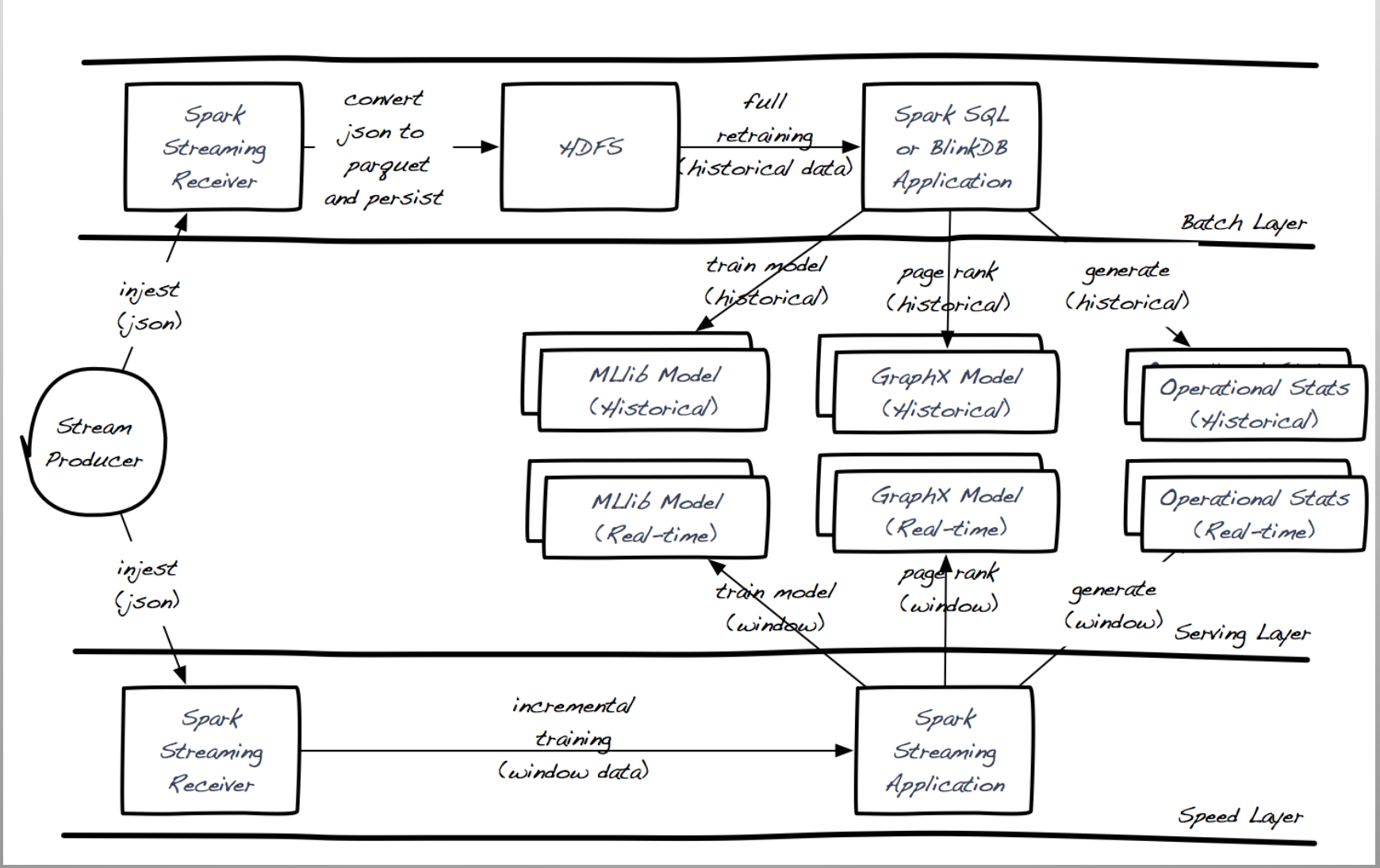
- Batch Layer
  - Immutable, Batch read, Append-only write
  - Source of truth
  - ie. HDFS
- Speed Layer
  - Mutable, Random read/write
  - Most complex
  - Recent data only
  - ie. Cassandra
- Serving Layer
  - Immutable, Random read, Batch write
  - ie. ElephantDB



Spark + AWS + Lambda



Spark + Lambda + GraphX + MLlib



Approximations

Approximation Overview

- Required for scaling efficiently
- Speed up analysis of large datasets
- Reduce size of working dataset
- Data is messy
- Collection of data is messy
- Exact isn't always necessary
- "Approximate is the new Exact"

Probabilistic Data Structs/Algos

- HyperLogLog
  - Count distinct elements for a date range
  - 16K for 99% accuracy
- Count-min Sketch
  - Heavy hitters, top-k, most-frequent elements
  - Supports streaming data
  - Fixed amount of space for all counts
- Bloom Filter
  - Search/filter elements
  - Bitset representation of hash
  - "Maybe yes, definitely no"

Benefits of Approximations

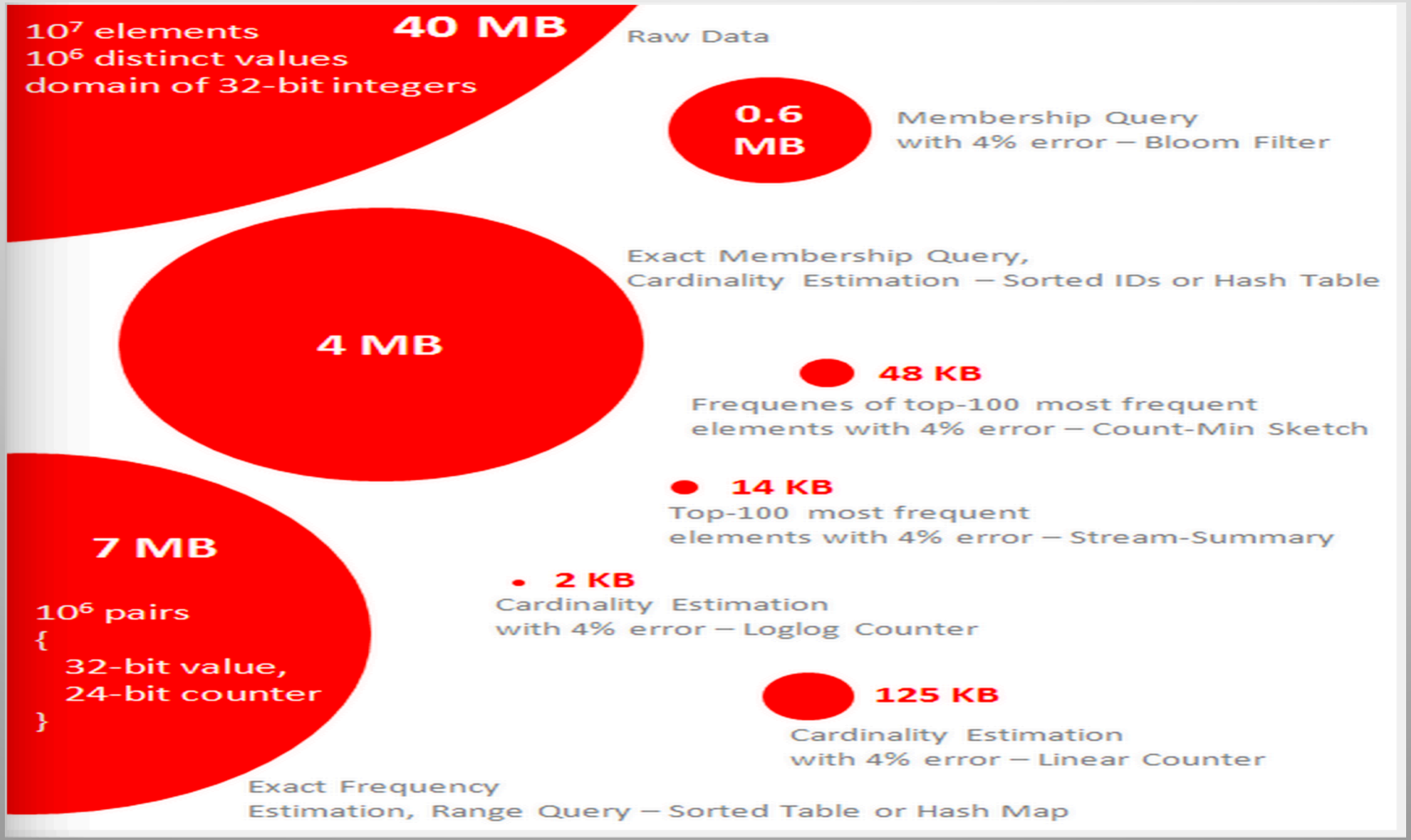


Figure: Memory Savings with Approximation Techniques
(http://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/)

Spark's Approximation Methods

- BlinkDB
  - Approximate error-bound, time-bound "online aggregation" queries
- RDD: sample(), takeSample()
  - Bernouilli and Poisson Sampling
- PairRDD: countApproxDistinctByKey()
  - HyperLogLog
- Spark Streaming and Twitter Algebird
  - Count-min Sketch
- MLlib RowMatrix.columnSimilarities()
  - DIMSUM approximation
- Bloom Filters
  - Everywhere!

Spark's Statistics Library

- Correlations
  - Dependence between 2 random variables
  - Pearson, Spearman
- Hypothesis Testing
  - Measure of statistical significance
  - Chi-squared test
- Stratified Sampling
  - Sample separately from different sub-populations
  - Bernoulli and Poisson sampling
  - With and without replacement
- Random data generator
  - Uniform, standard normal, and Poisson distribution

Upcoming Spark Conferences

Summary

- Spark, Spark Streaming Overview
- Use Cases
- API and Libraries
- Machine Learning
- Graph Processing
- Execution Model
- Fault Tolerance
- Cluster Deployment
- Monitoring
- Scaling and Tuning
- Lambda Architecture
- Probabilistic Data Structs/Algos
- Approximations

http://sparkinaction.com
http://effectivespark.com

Thanks!!

Chris Fregly
@cfregly