# IEMS5709 Spring 2016
# Advanced Topics in Information Processing:
# Big Data Processing Systems and Information Processing

# Spark Streaming, SparkSQL, MLlib, GraphX and Beyond

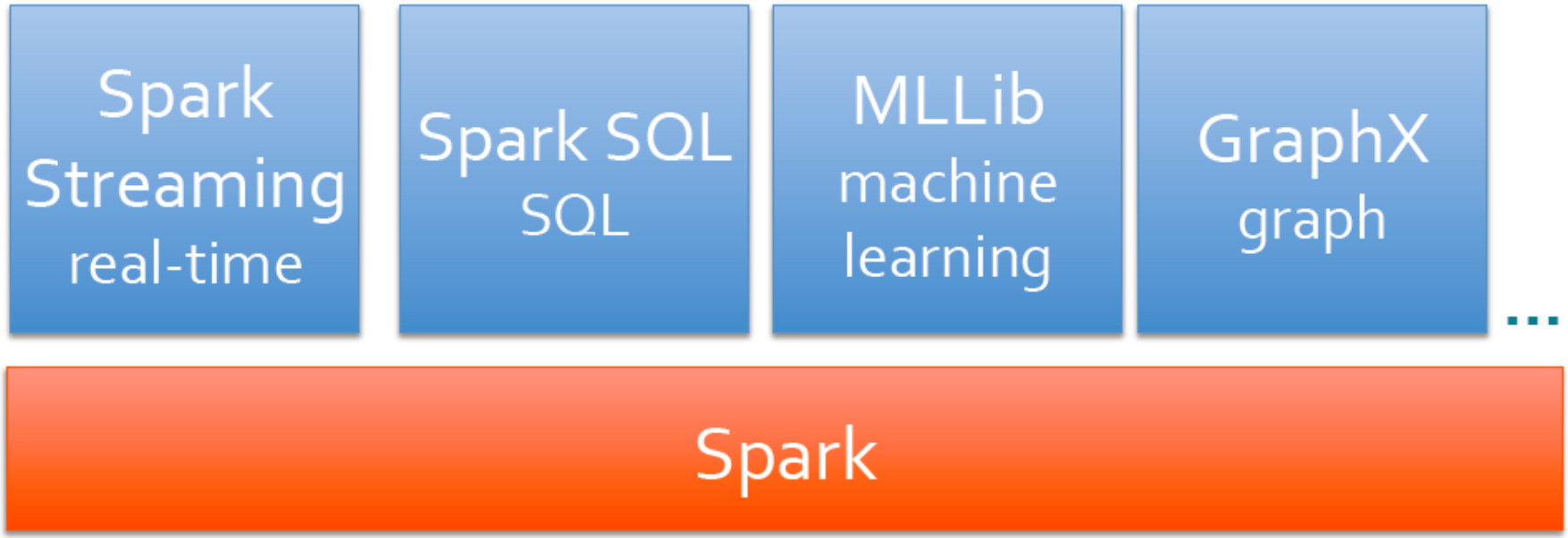
Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

# Acknowledgements

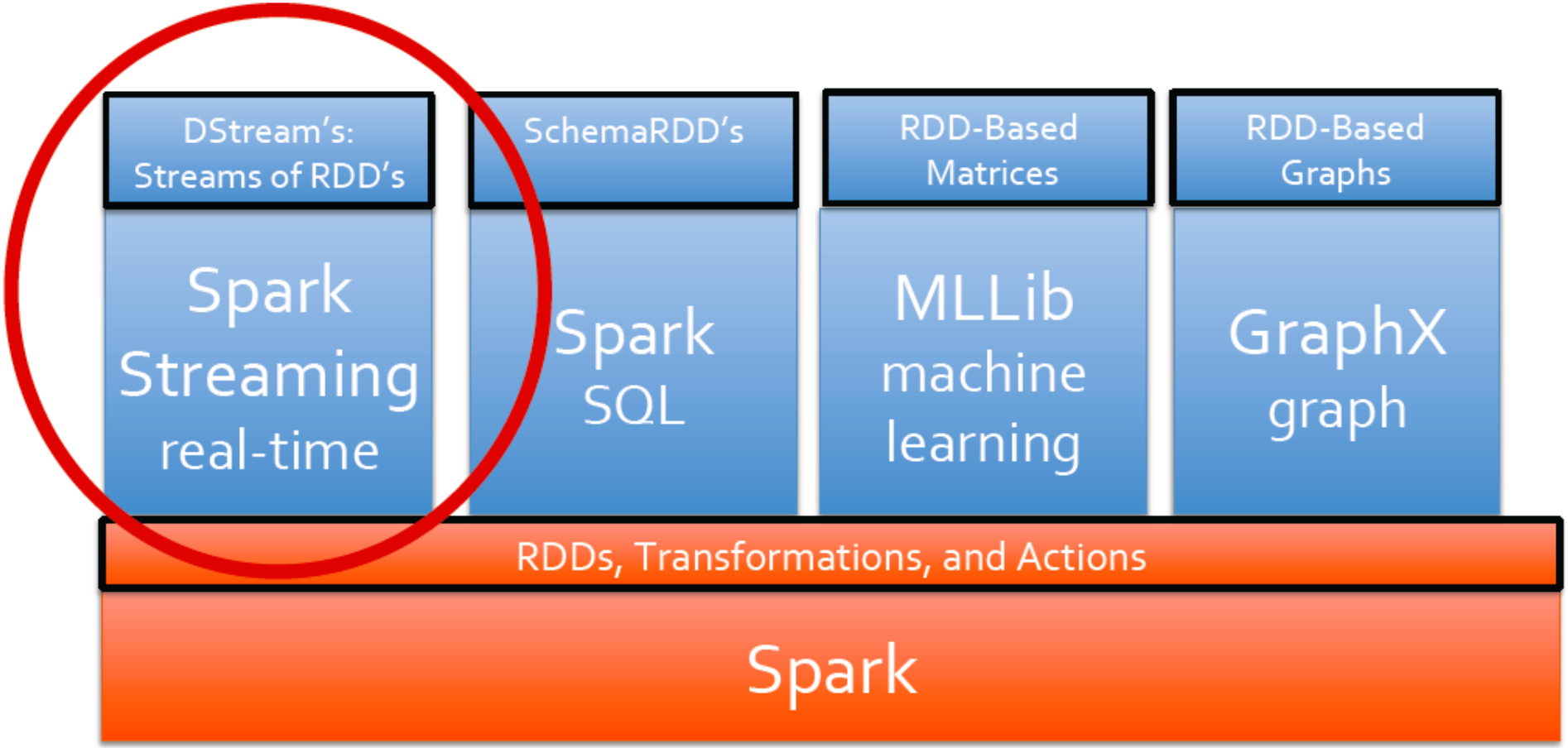

- Slides in this chapter are adapted from the following sources:
  - Matei Zaharia, “Spark 2.0,” Spark Summit East Keynote, Feb 2016.
  - Reynold Xin, “The Future of Real-Time in Spark,” Spark Summit East Keynote, Feb 2016.
  - Michael Armburst, “Structuring Spark: SQL, DataFrames, DataSets, and Streaming,” Spark Summit East Keynote, Feb 2016.
  - Ankur Dave, “GraphFrames: Graph Queries in Spark SQL,” Spark Summit East, Feb 2016.
  - Michael Armburst, “Spark DataFrames: Simple and Fast Analytics on Structured Data,” Spark Summit Amsterdam, Oct 2015.
  - Michael Armburst et al, “Spark SQL: Relational Data Processing in Spark,” ACM SIGMOD 2015 Talk.
  - Michael Armburst, “Spark SQL Deep Dive,” Melbourne Spark Meetup, June 2015.
  - Reynold Xin, “Spark,” Stanford CS347 Guest Lecture, May 2015.
  - Joseph K. Bradley, “Spark DataFrames and ML Pipelines,” MLconf Seattle, May 2015.
  - Ameet Talwalkar, “MLlib: Spark’s Machine Learning Library,” AMPCamps 5 Talk, Nov. 2014.
  - Shivaram Venkataraman, Zongheng Yang, “SparkR: Enabling Interactive Data Science at Scale,” AMPCamps 5 Talk, Nov. 2014.
  - Tathagata Das, “Spark Streaming: Large-scale near-real-time stream processing,” O’Reilly Strata Conference talk, 2013.
  - Joseph Gonzalez et al, “GraphX: Graph Analytics on Spark,” talk at AMPCAMP 3, 2013.

- 

# Generality of RDDs in Spark



# Generality of RDDs in Spark



# Motivation for Spark Streaming

- Many Important Applications must process Large Data Streams at second-scale latencies
  - Site Statistics, Intrusion Detection, Online ML, Fraud Detection
- To build and scale these applications require:
  - Integration: with Offline Analytic Stack
  - Fault-tolerance: to handle Crashes and Stragglers
  - Efficiency: low cost beyond base processing
  - Work with distributed collections as you would with local ones

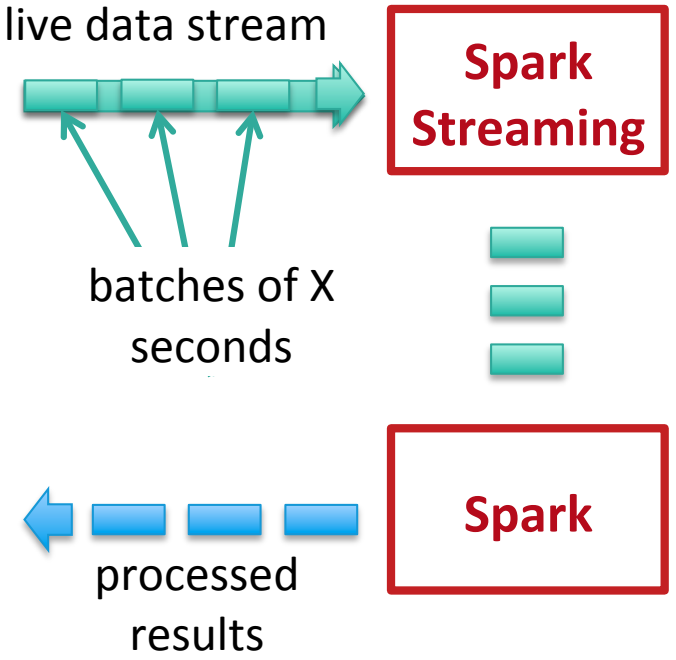
# Spark Streaming Overview

- Low Latency, High-throughput and Fault Tolerant
- DStream: Micro-batches of RDDs
  - Operations are similar to RDD
    - Lineage for Fault-Tolerance
- Leverage Core Components from Spark
  - RDD data model and API
  - Data Partitioning and Shuffles
  - Task Scheduling
  - Monitoring/ Instrumentation
  - Scheduling and Resource Allocation
- Support Flume, Kafka, Twitter, Kinesis, etc for Data Ingestion
- Long-running Spark Applications

# Discretized Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

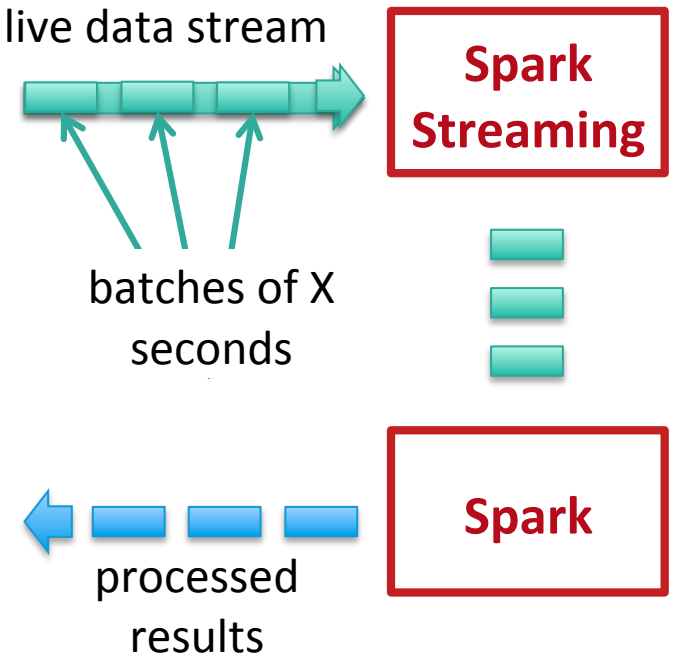
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



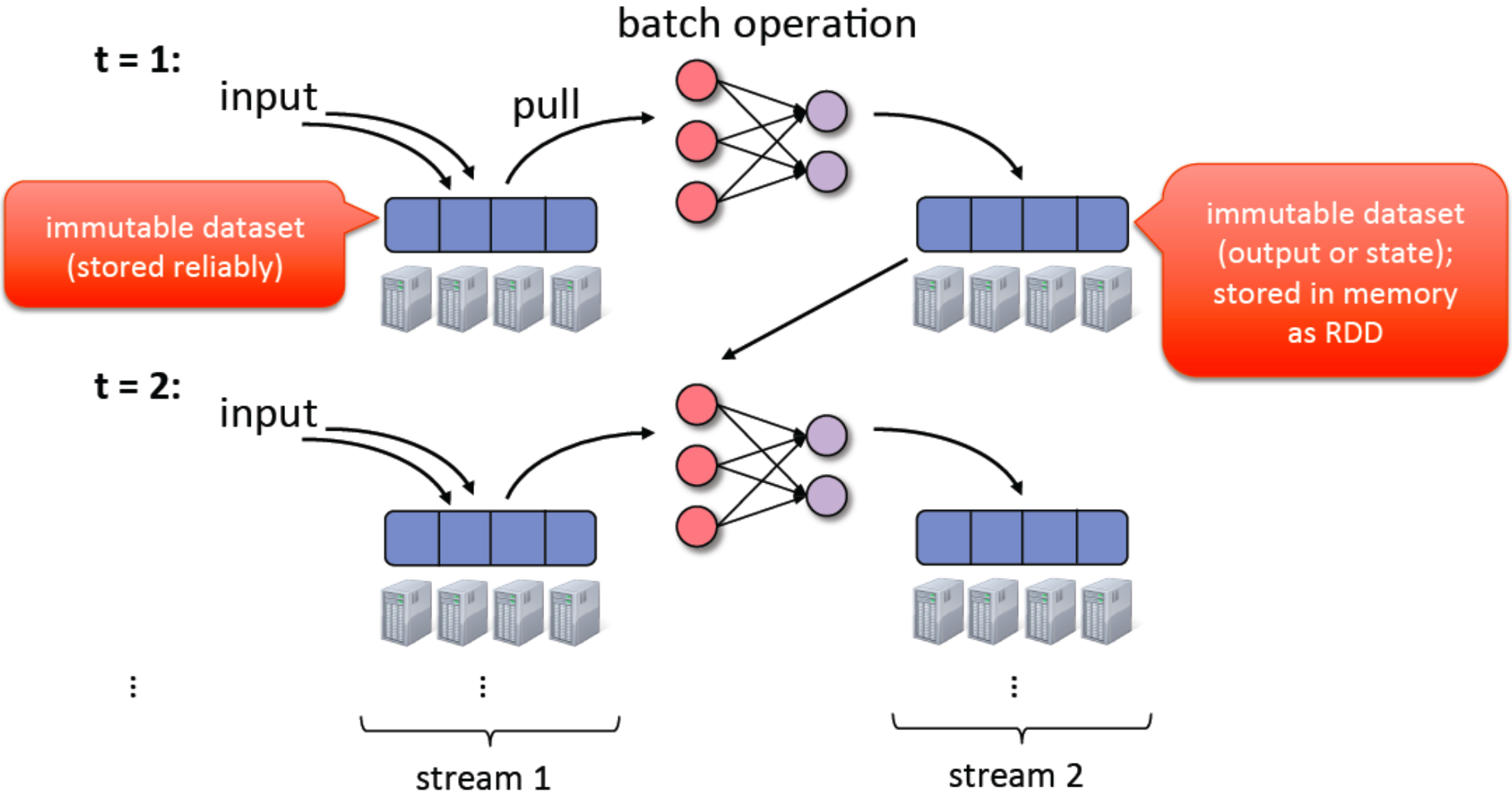
# Discretized Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



# Discretized Stream Processing (Micro-Batching)



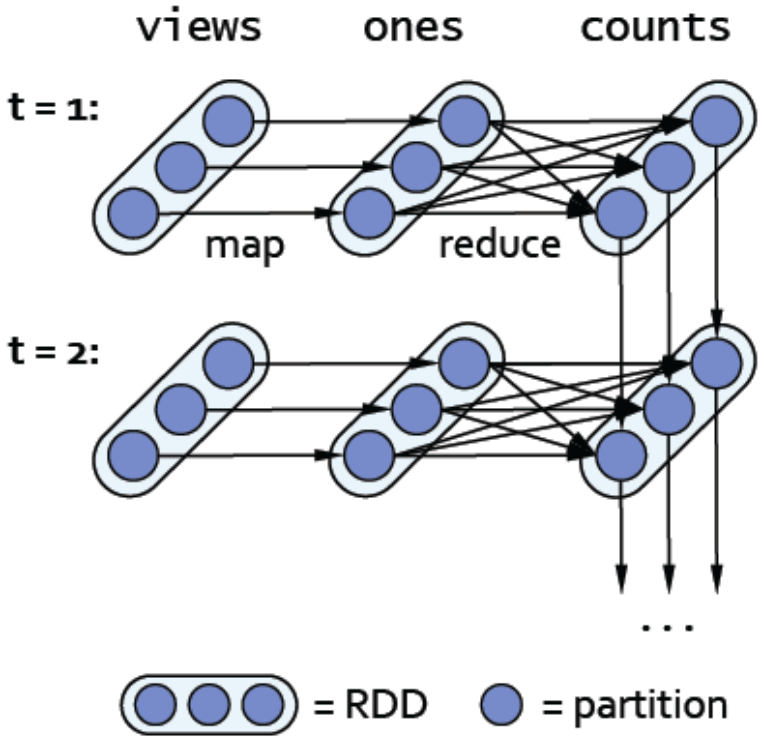
# Programming Interface

## Simple functional API

```
views = readStream("http:...", "1s")
ones = views.map(ev => (ev.url, 1))
counts = ones.runningReduce(_ + _)
```

## Interoperates with RDDs

```
// Join stream with static RDD
counts.join(historicCounts).map(...)

// Ad-hoc queries on stream state
counts.slice("21:00","21:05").topK(10)
```



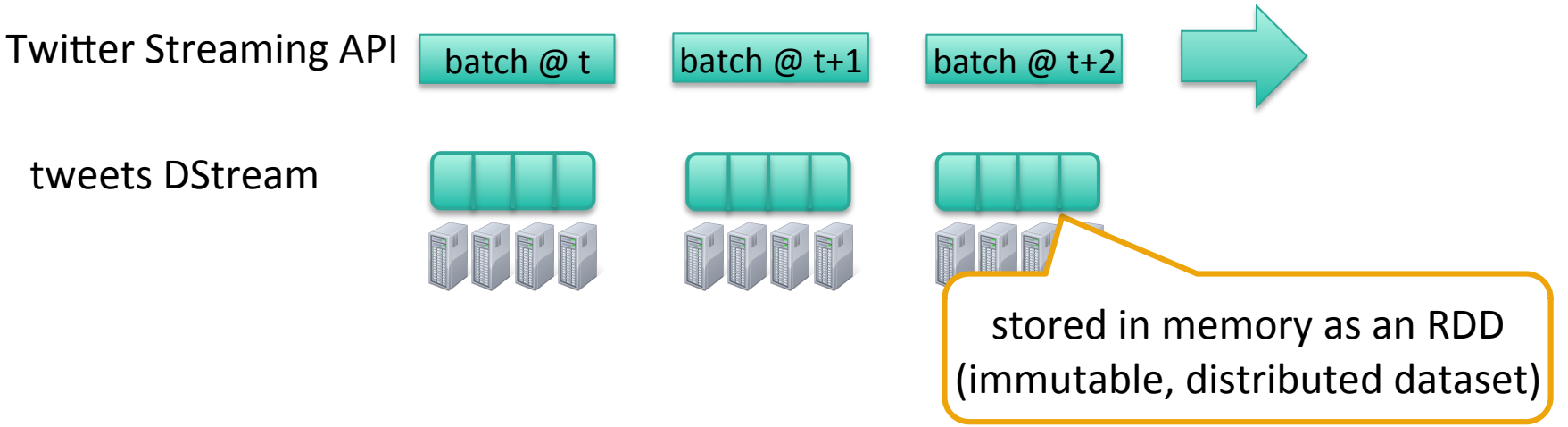
# Spark Streaming API

- **Transformations** – modify data from one DStream to another
  - Standard RDD operations – map, filter, distinct, countByValue, reduceByKey, join, …
  - Stateful, Sliding Window-based Operations
    - window, updateStateByKey, countByValueAndWindow
    - Window Size & Slide Interval
- **Output Operations –** send data to external entity
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results
- Checkpointing
- Register DStream as a SQL table

# Example 1: Get HashTags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

**DStream**: a sequence of distributed datasets (RDDs) representing a distributed stream of data

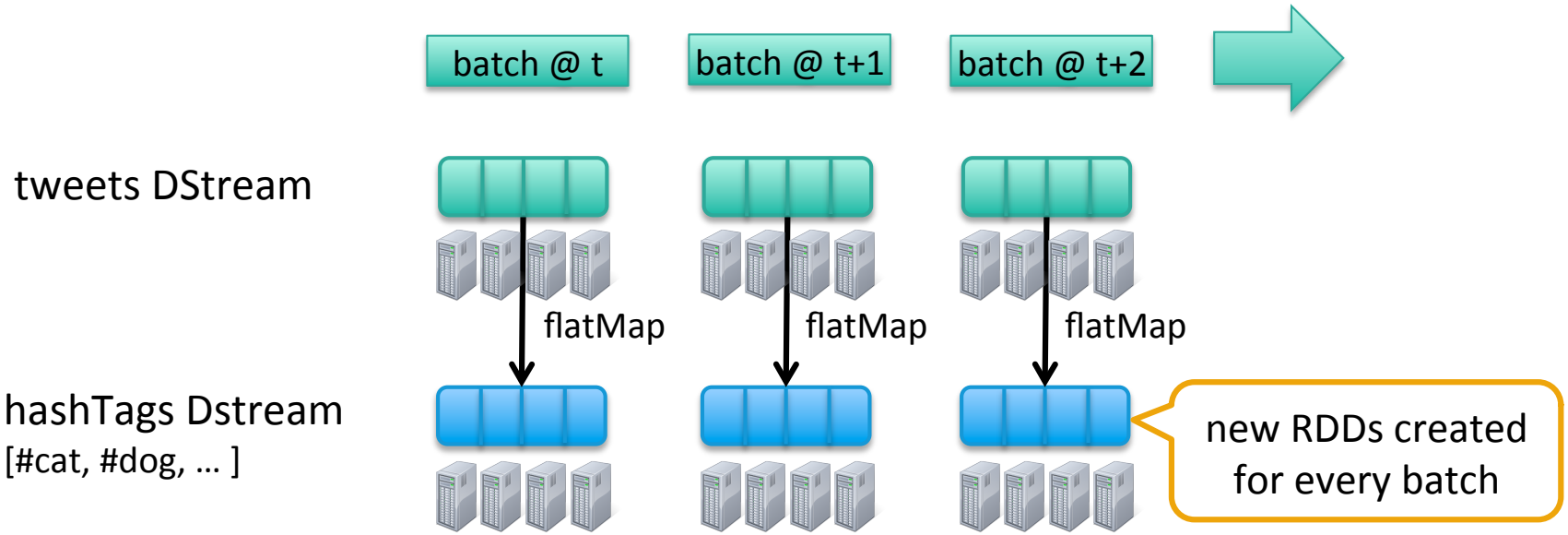


# Example 1: Get HashTags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

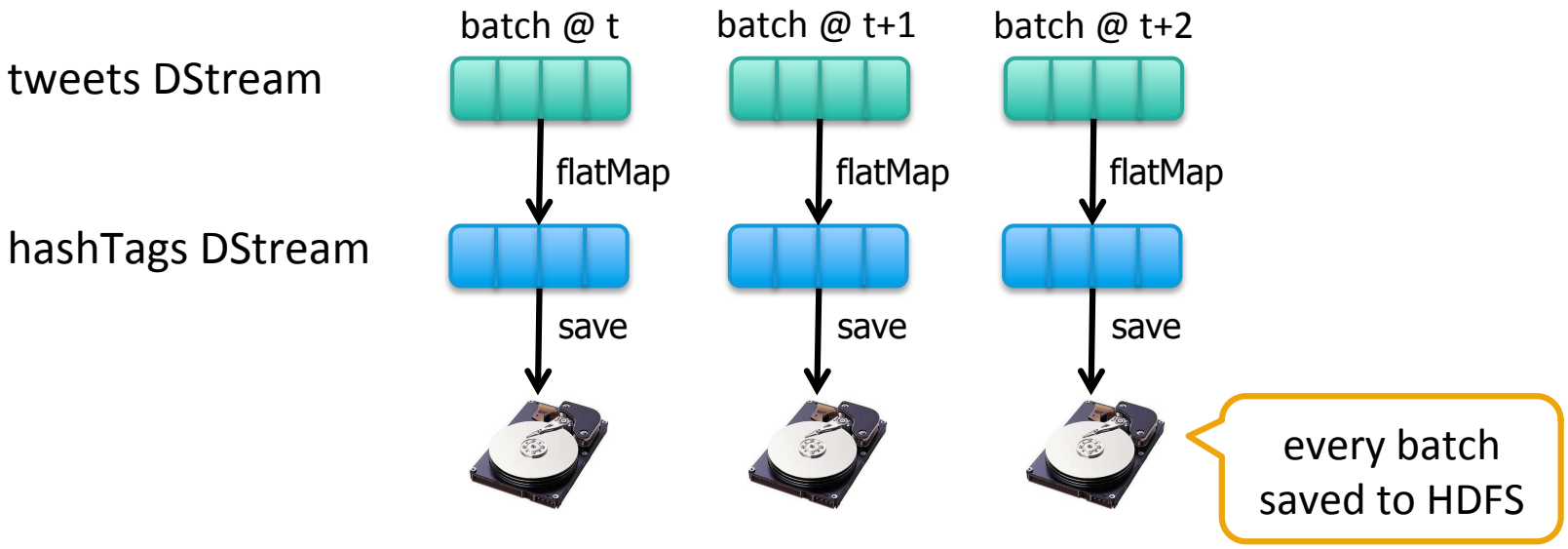
**transformation**: modify data in one DStream to create another DStream



# Example 1: Get HashTags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

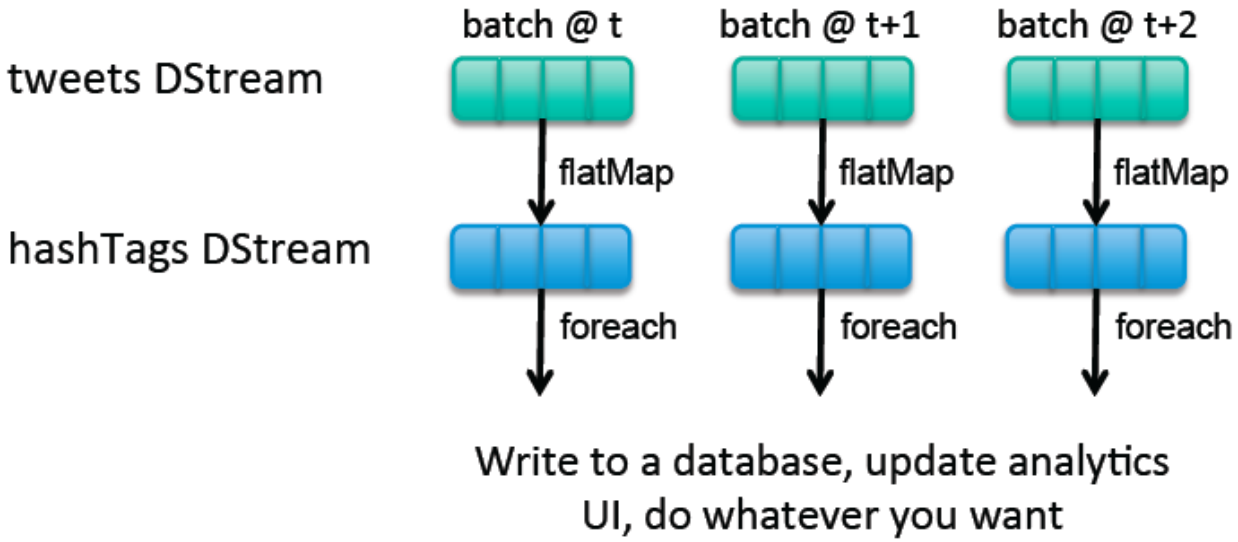
**output operation**: to push data to external storage



# Example 1: Get HashTags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreachRDD(hashTagRDD => { ... })
```

**foreach**: do whatever you want with the processed data



# Example 1 in Java vs. Scala

## Scala

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)

val hashTags = tweets.flatMap (status => getTags(status))

hashTags.saveAsHadoopFiles("hdfs://...")
```

## Java

```
JavaDStream<Status> tweets = ssc.twitterStream(<Twitter username>,
<Twitter password>)

JavaDstream<String> hashTags = tweets.flatMap(new Function<...> {  })

hashTags.saveAsHadoopFiles("hdfs://...")
```

Function object

# Spark program vs Spark Streaming program

## Spark Streaming program on Twitter stream

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

## Spark program on Twitter log file

```
val tweets = sc.hadoopFile("hdfs://...")
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFile("hdfs://...")
```

# Vision - *one stack to rule them all*

- Explore data interactively using Spark Shell / PySpark to identify problems
- Use same code in Spark stand-alone programs to identify problems in production logs
- Use similar code in Spark Streaming to identify problems in live log streams

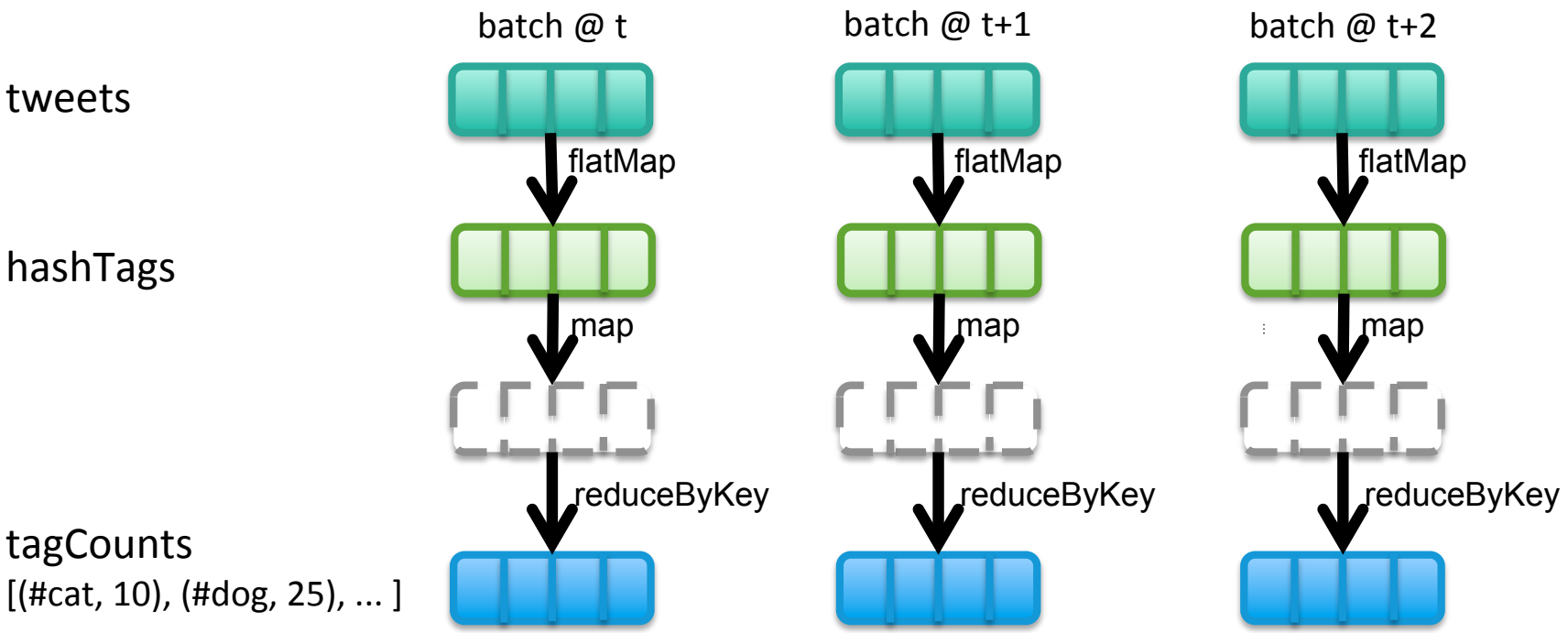
```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = file.map(...)
```

```
object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered =
file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
    ...
}
```

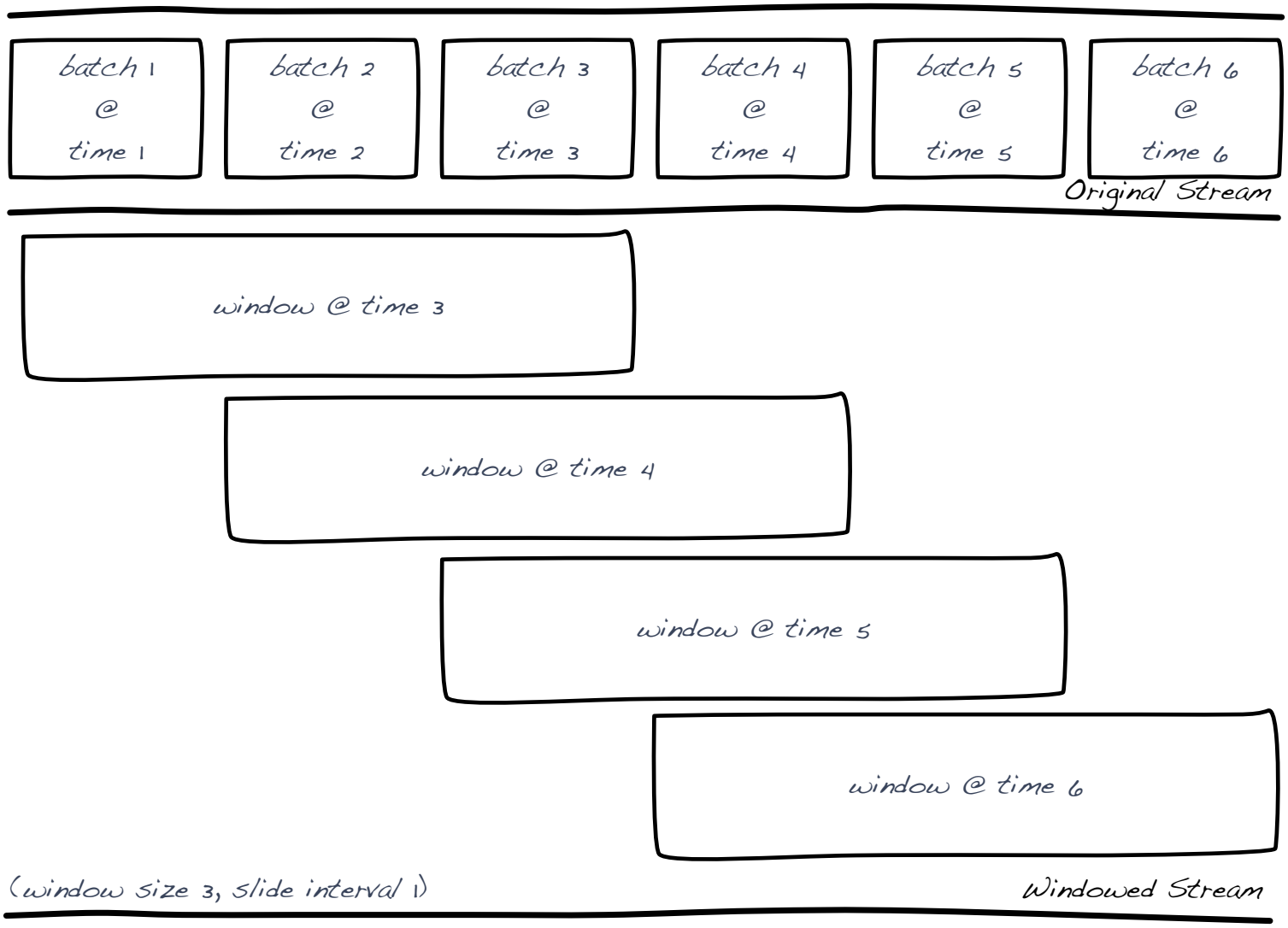
```
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered =
file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
    ...
  }
}
```

# Example 2: Count the HashTags

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.countByValue()
```



# Window-based Operations on DStreams



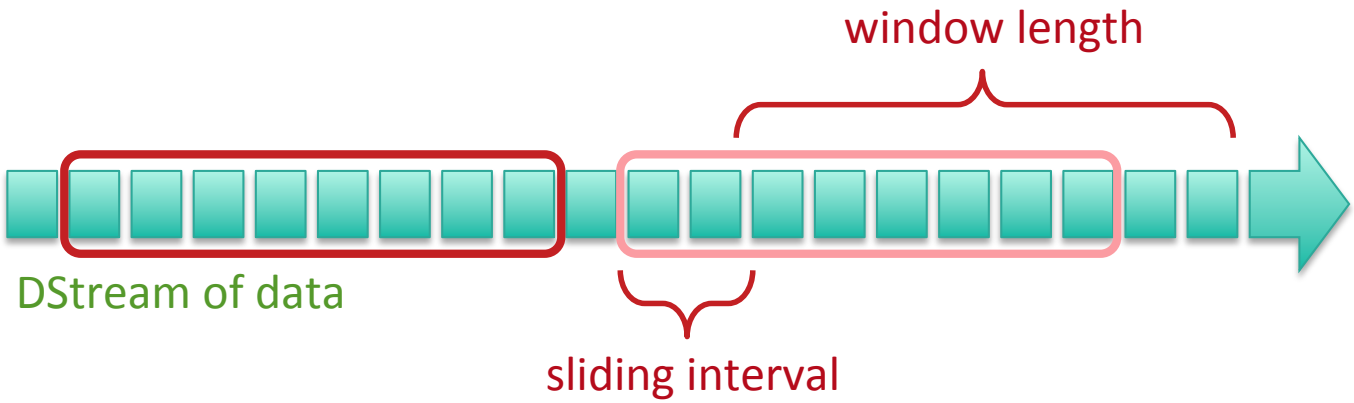
# Example 3: Count the HashTags over last 1 min

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(1)).countByValue()
```

sliding window operation

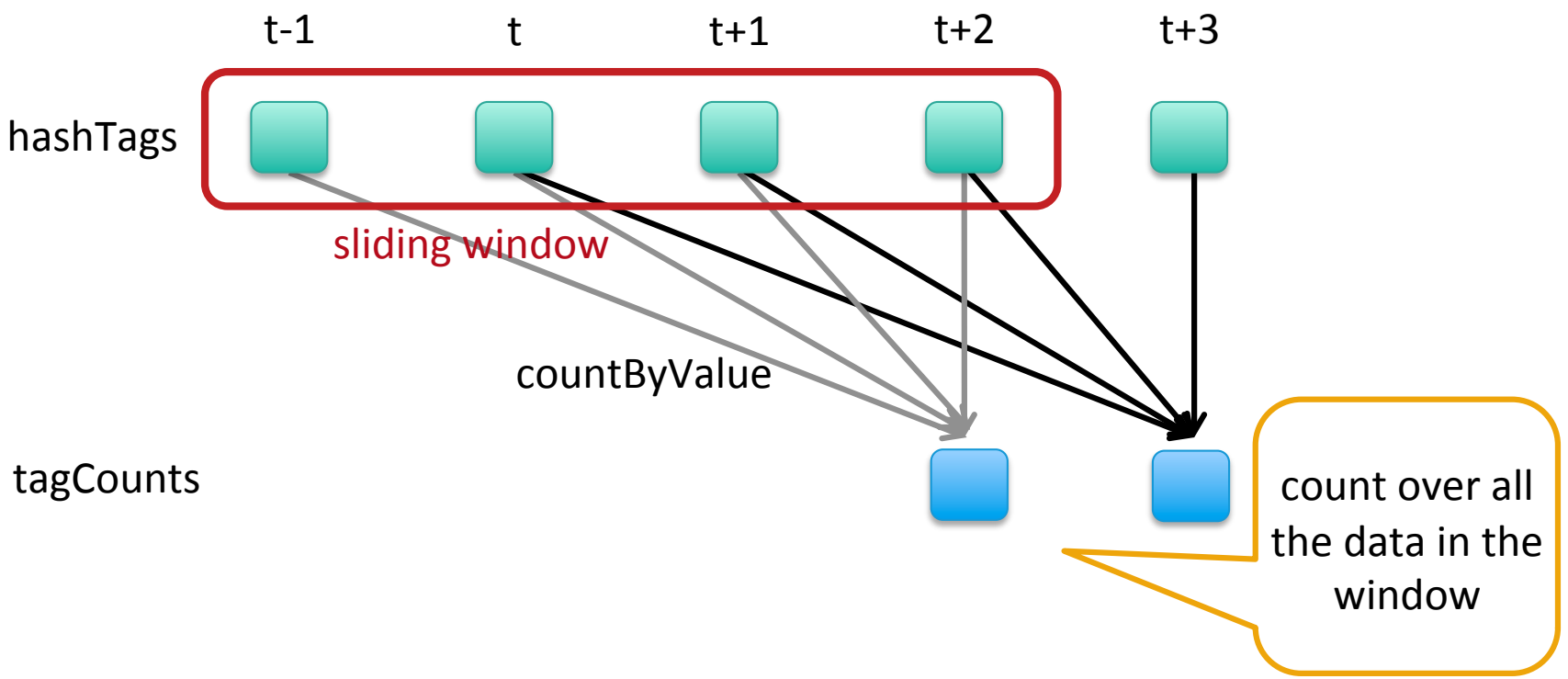
window length

sliding interval



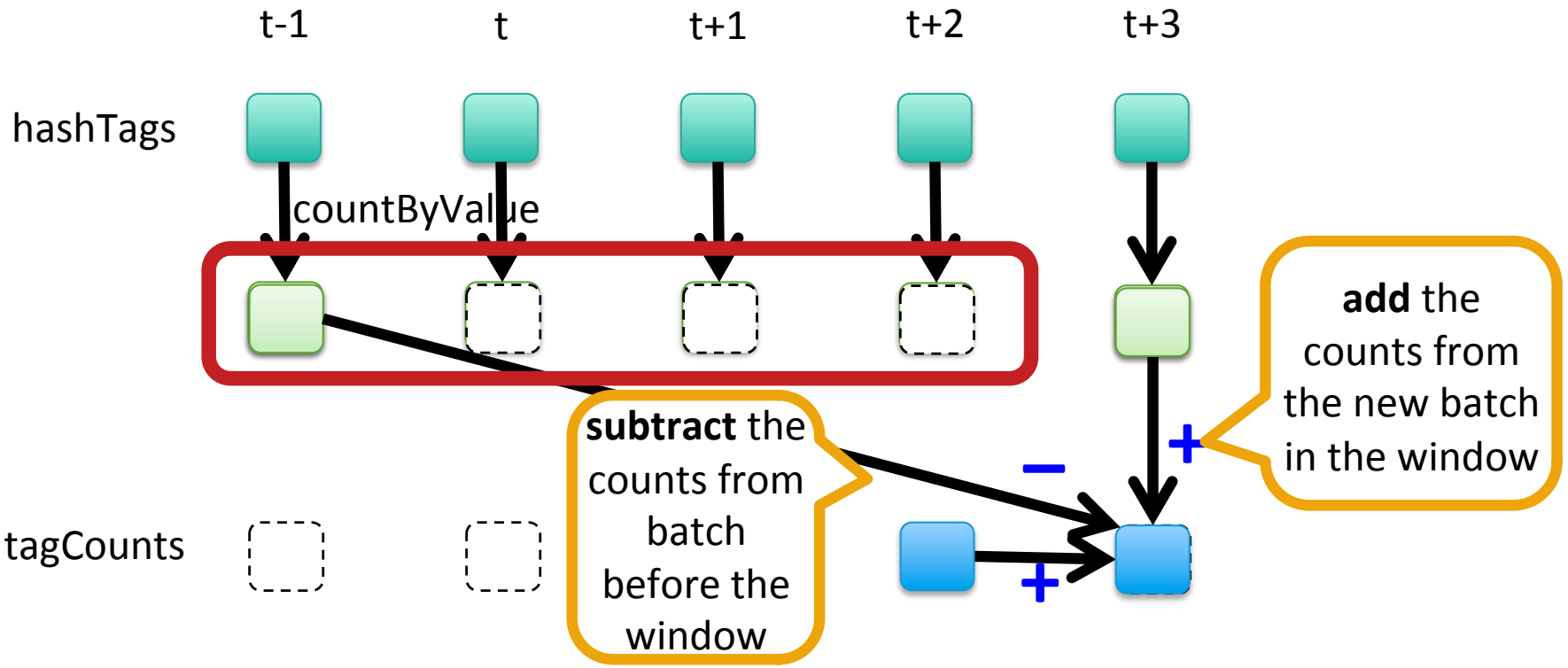
# Example 3: Count the HashTags over last 1 min

```
val tagCounts = hashTags.window(Minutes(1), Seconds(1)).countByValue()
```



# Example 3: Smart Window-based countByValue

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10),
Seconds(1))
```



# Smart window-based *reduce*

- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to "inverse reduce" ("subtract" for counting)

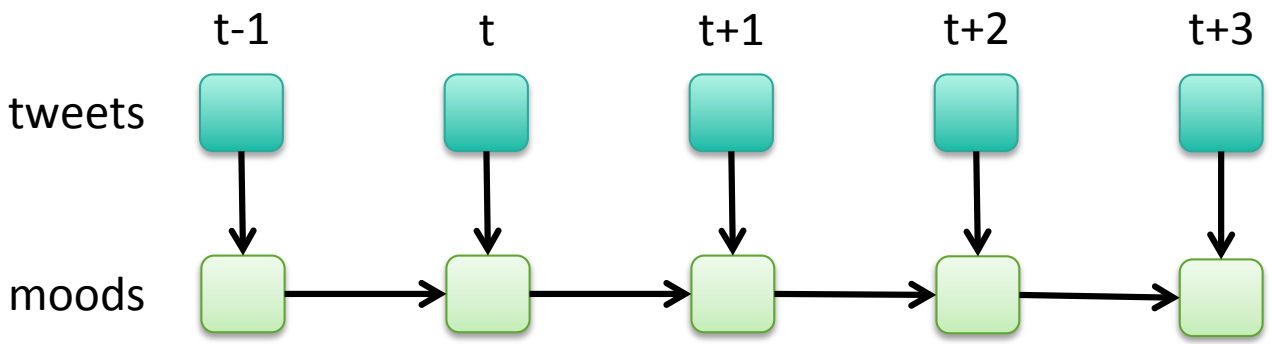
- Could have implemented counting as:

```
hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), …)
```

# Arbitrary Stateful Computations

- Maintain arbitrary state, track sessions and specify function to generate new state based on previous state and new data:
  - e.g. Maintain per-user mood as state, and update it with his/her tweets

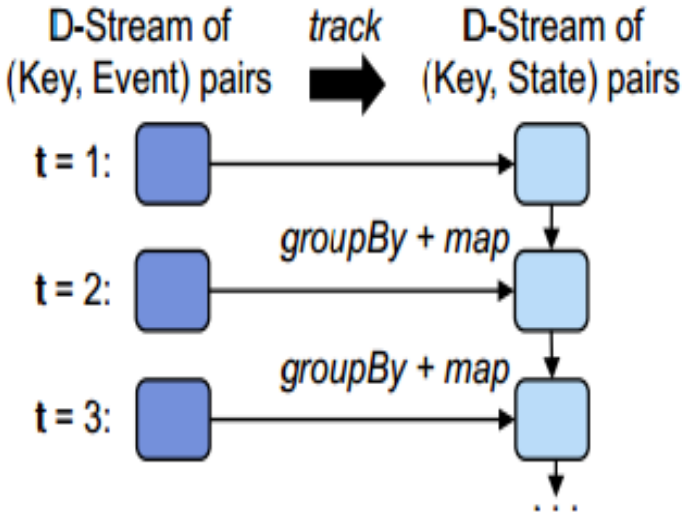
```
def updateMood(newTweets, lastMood) => newMood
val moods = tweets.updateStateByKey(tweet => updateMood( _ ))
```



# State Tracking

- A series of events → state changing

```
sessions = events.track(
(key, ev) => 1, // initialize function
(key, st, ev) => // update function
ev == Exit ? null : 1,
"30s") // timeout
counts = sessions.count() // a stream
of ints
```



# Another Example: Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))

val lines = KafkaUtils.createStream(context, ...)
```

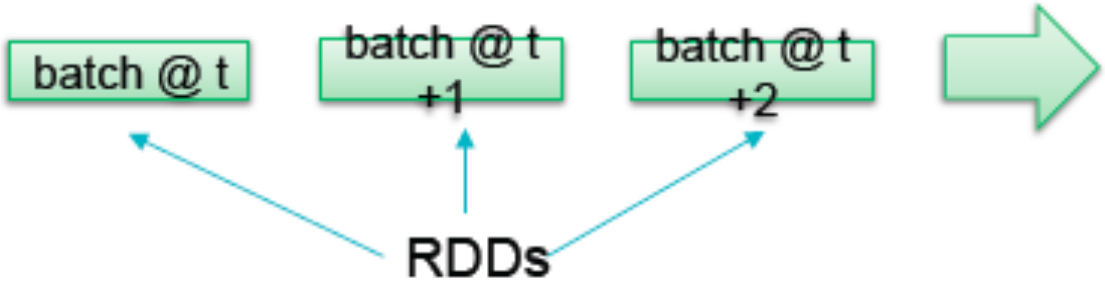
entry point of streaming functionality

create **DStream** from Kafka data

**Discretized Stream (DStream)** basic abstraction of Spark Streaming

series of RDDs representing a stream of data

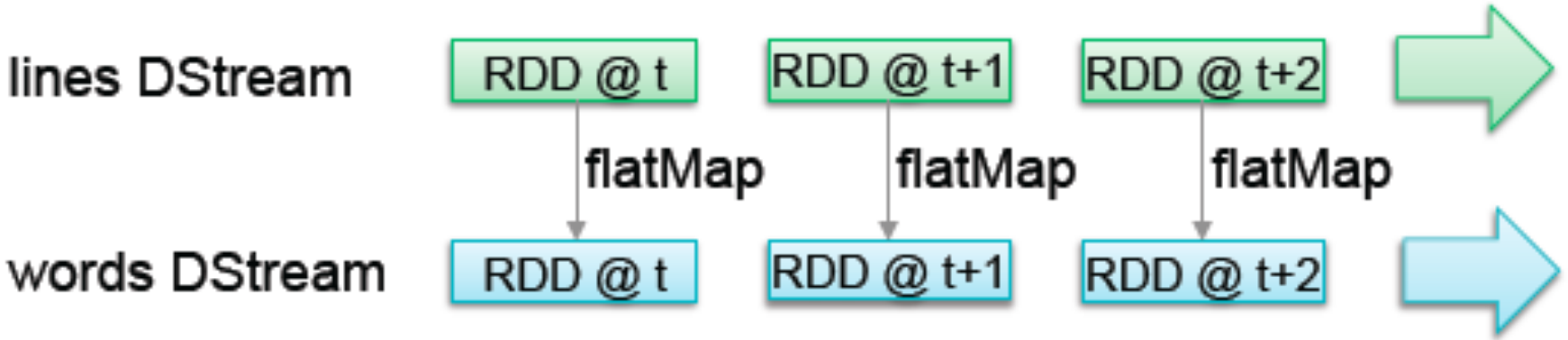
lines DStream



# Another Example: Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))

val lines = KafkaUtils.createStream(context, ...)

val words = lines.flatMap(_.split(" "))
```

split lines into words



# Another Example: Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))

val lines = KafkaUtils.createStream(context, ...)

val words = lines.flatMap(_.split(" "))

val wordCounts = words.map(x => (x, 1))
                      .reduceByKey(_ + _)

wordCounts.print()

context.start()
```

count the words

print some counts on screen

start receiving and transforming the data

# Another Example: Word Count with Kafka

```
val context = new StreamingContext(conf, Seconds(1))

val lines = KafkaUtils.createStream(context, ...)

val words = lines.flatMap(_.split(" "))

val wordCounts = words.map(x => (x, 1))
                      .reduceByKey(_ + _)

wordCounts.foreachRDD(rdd => /* do something */ )

context.start()
```

push data out to storage systems

# Many Transformations

## Window operations

```
words.map(x => (x, 1)).reduceByKeyAndWindow(_ + _, Minutes(1))
```

## Arbitrary stateful processing

```
def stateUpdateFunc(newData, lastState) => updatedState

val stateStream = keyValueDStream.updateStateByKey(stateUpdateFunc)
```

# Combine Batch and Stream Processing

- Inter-mix RDD and DStream operations
  - e.g., Join incoming tweets with a spam HDFS file to filter out bad tweets

```
tweets.transform(tweetsRDD => {
 tweetsRDD.join(spamHDFSFile).filter(...)
})
```

- Query streaming data using SQL, e.g.

```
select  *  from table_from_streaming_data
```

# Another Example

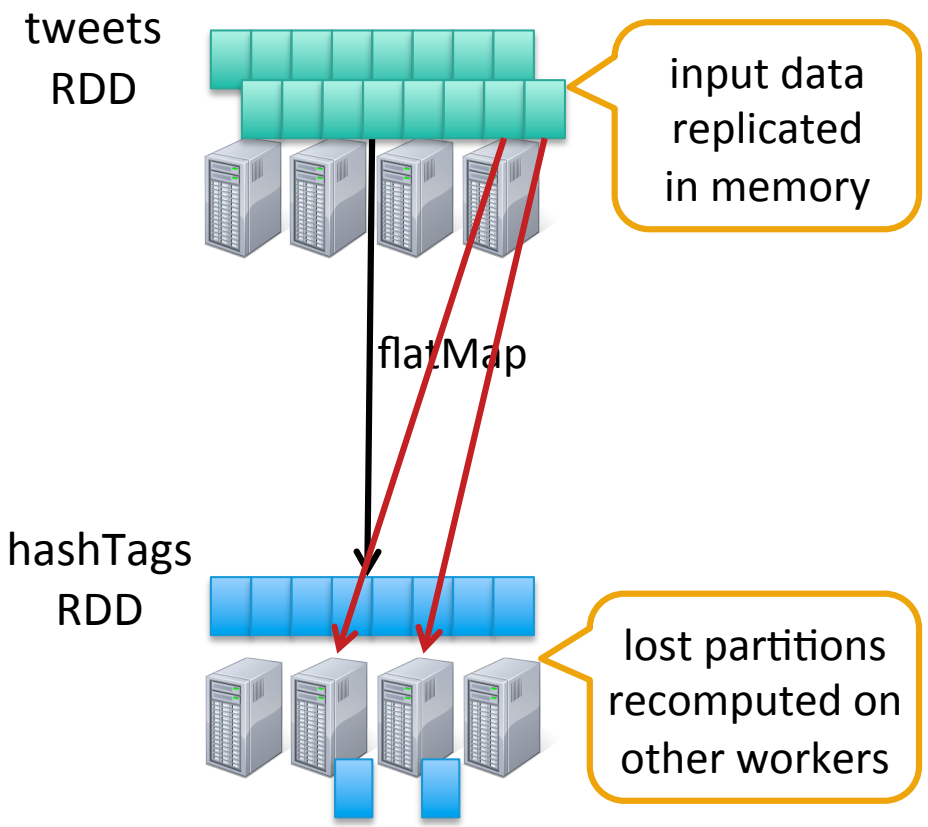
```
val ssc = new StreamingContext(sc, Seconds(5))
val sqlContext = new SQLContext(sc)
val tweets = TwitterUtils.createStream(ssc, auth)
val transformed = tweets.filter(isEnglish).window(Minutes(1))

transformed.foreachRDD { rdd =>
  // Tweet is a case class containing necessary
  rdd.map(Tweet.apply(_)).registerAsTable("tweets")
}
```

---

```
SELECT text FROM tweets WHERE similarity(tweet) > 0.01
SELECT getClosestCountry(lat, long) FROM tweets
```

# Fault Tolerance

- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- Therefore, all transformed data is fault-tolerant
- Exactly once semantics
  - No double counting



# Input Sources

- Out of the box support for:
  - Kafka, Flume, Akka Actors, Raw TCP sockets, HDFS, etc
- Developers can write additional custom *receiver(s)*
  - Just define what to and when receiver is started and stopped
- Can also generate one's own sequence of RDDs and push them in as a "Stream"

# Zero (Input) Data Loss during Streaming

For Non-replayable Sources, i.e. sources that do not support replay from any position (e.g. Flume, etc):

- Solved using Write Ahead Log (WAL) (since Spark 1.3)

For Replayable Sources, i.e. sources that allow data to be replayed from any position (e.g. Kafka, Kinesis, etc):

- Solved with more reliable Kafka and Kinesis Integrations (Spark 1.3-1.5)

# Write Ahead Log (WAL) [since Spark 1.3]

- All received data synchronously written to HDFS and replayed when necessary after failure
- WAL can be enabled by setting Spark configuration flag:

  `spark.streaming.receiver.writeAheadLog.enabled` to `TRUE`

- Can give end-to-end at least once guarantee for sources that can support acks, but do not support replays

# Reliable Kinesis [since Spark 1.5]

- Save record sequence numbers instead of data to WAL
- Replay from Kinesis using sequence numbers
- Higher throughput than using WAL
- Can give at least once guarantee

# Reliable Kafka [Spark 1.3, graduated in 1.5]

- New API: Direct Kafka stream:
  - Does not use receivers, does not use ZooKeeper to save offsets
  - Offset management (saving, replaying) by Spark Streaming
- Can provide up to 10x higher throughput than earlier receiver
  - https://spark-summit.org/2015/events/towards-benchmarking-modern-distributed-streaming-systems/
- Can give exactly-once guarantee
- Can run Spark batch jobs directly on Kafka
  - # of RDD partitions = # of Kafka partitions, easy to reason about
  - https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html

# Discretized Stream (DStream)

A sequence of RDDs representing
a stream of data

What does it take to define a DStream?

# DStream Interface

The DStream interface primarily defines how to generate an RDD in each batch interval

- List of *dependent (*parent) DStreams
- *Slide Interval*, the interval at which it will compute RDDs
- Function to *compute* RDD at a time t

# Example: Mapped DStream

- *Dependencies:* Single parent DStream
- *Slide Interval:* Same as the parent DStream
- *Compute function for time t:* Create new RDD by applying map function on parent DStream's RDD of time t

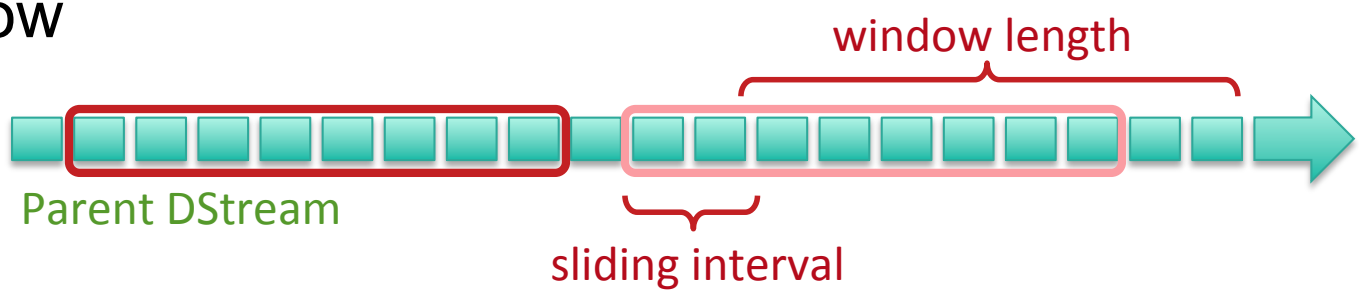
```
override def compute(time: Time): Option[RDD[U]] = {
    parent.getOrCompute(time).map(_.map[U](mapFunc))
}
```

Gets RDD of time t if already computed once, or generates it

Map function applied to generate new RDD

# Example: Windowed DStream

*Window* operation gather together data over a sliding window




*Dependencies:* Single parent DStream

*Slide Interval:* Window sliding interval

*Compute function for time t:* Apply union over all the RDDs of parent DStream between times t and (t – window length)

# Example: Network Input DStream

Base class of all input DStreams that receive data from the network

- *Dependencies:* None
- *Slide Interval:* Batch duration in streaming context
- *Compute function for time t:* Create a BlockRDD with all the blocks of data received in the last batch interval
- Associated with a Network Receiver object

# Network Receiver

Responsible for receiving data and pushing it into Spark's data management layer (Block Manager)

Base class for all receivers - Kafka, Flume, etc.

Simple Interface:

- What to do *on starting* the receiver
  - Helper object *blockGenerator* to push data into Spark
- What to do *on stopping* the receiver

# Example: Socket Receiver

- *On start*:

  Connect to remote TCP server

  While socket is connected,

    Receiving bytes and deserialize

    Deserialize them into Java objects

    Add the objects to *blockGenerator*

- *On stop*:

  Disconnect socket

# Other functions in DStream interface

- *parentRememberDuration* – defines how long should
  - Window-based DStreams have *parentRememberDuration = window length*

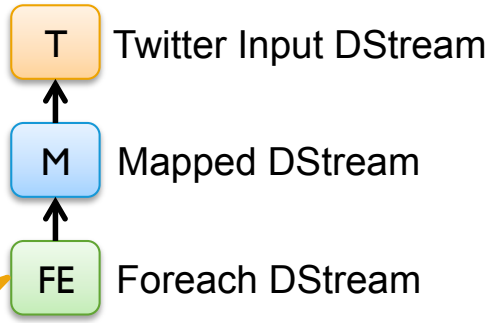
- *mustCheckpoint* – if set to true, the system will automatically enable periodic checkpointing
  - Set to true for stateful DStreams

# DStream Graph

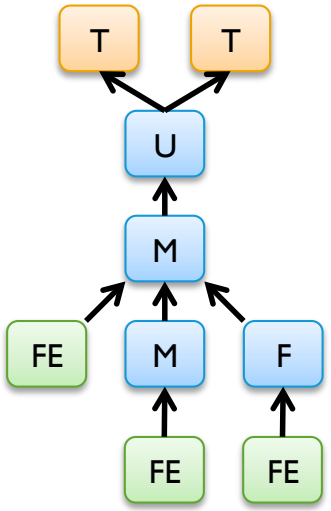
## Spark Streaming program

```
t = ssc.twitterStream("...")
        .map(...)
t.foreach(...)
```

## DStream Graph

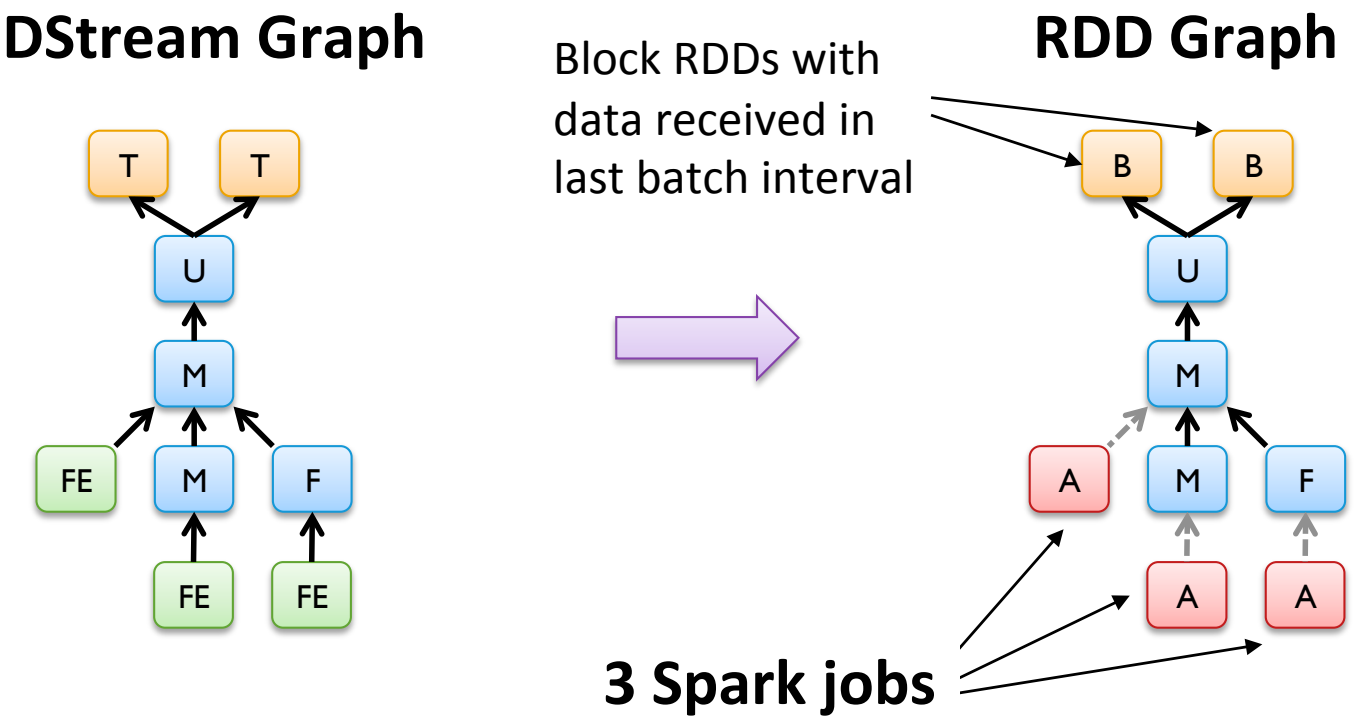



```
t1 = ssc.twitterStream("...")
t2 = ssc.twitterStream("...")

t = t1.union(t2).map(...)

t.saveAsHadoopFiles(...)
t.map(...).foreach(...)
t.filter(...).foreach(...)
```



# DStream Graph → RDD Graphs → Spark jobs

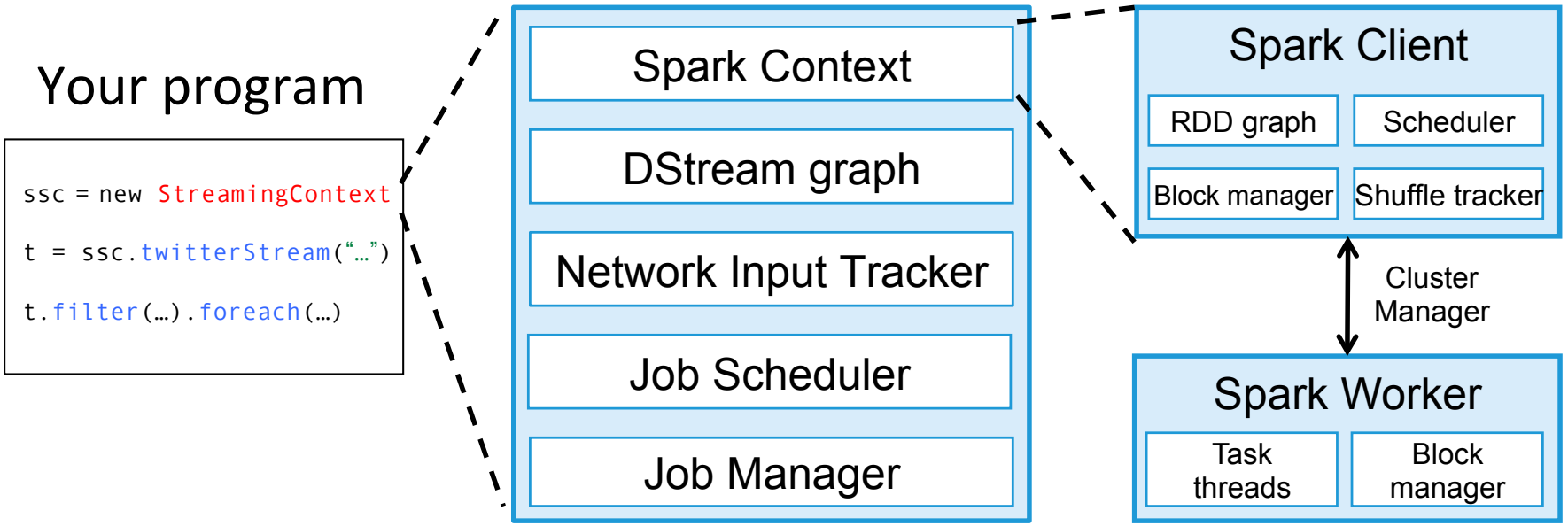
- Every interval, RDD graph is computed from DStream graph
- For each output operation, a Spark action is created
- For each action, a Spark job is created to compute it



# Agenda

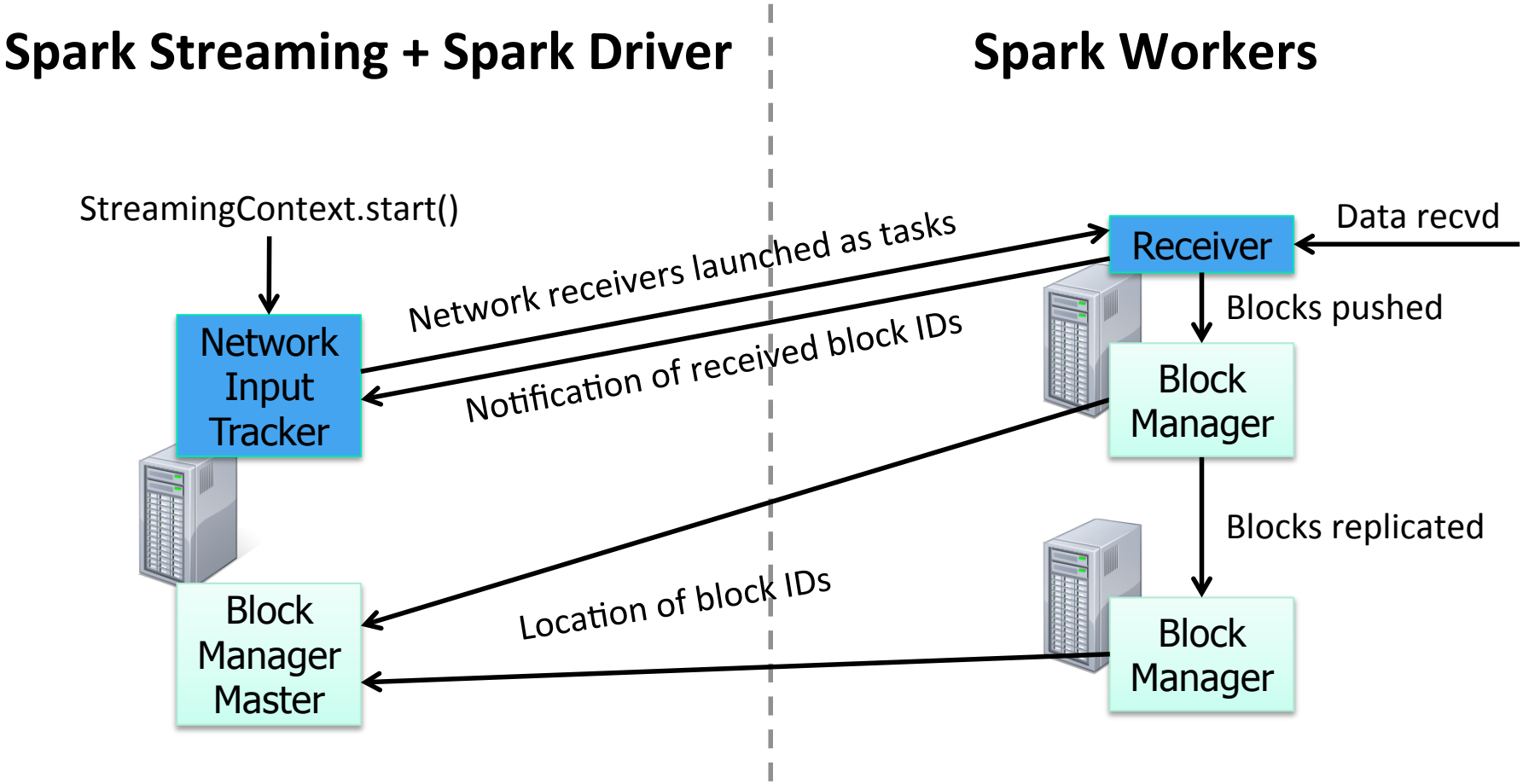
- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

# Components

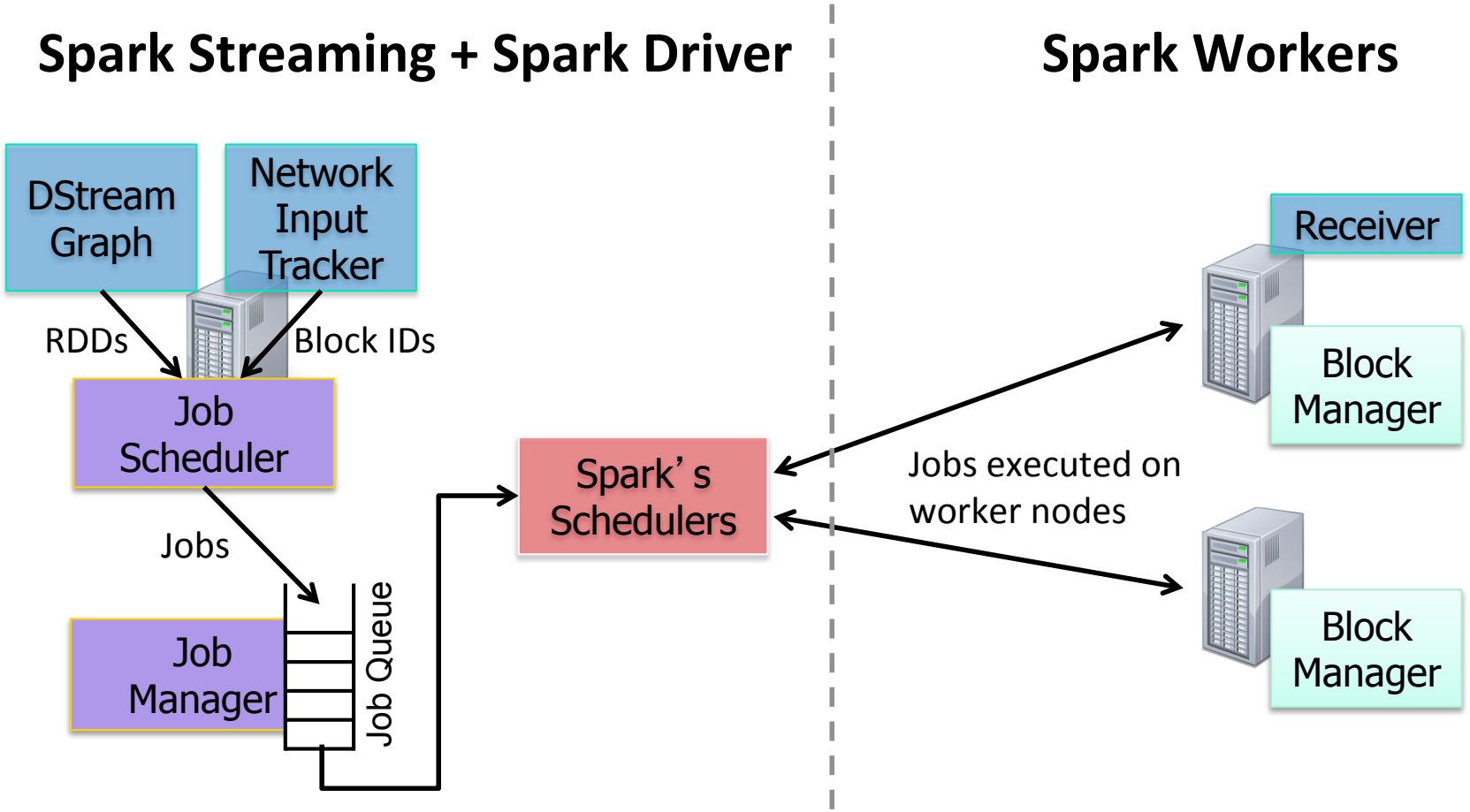



- **Network Input Tracker** – Keeps track of the data received by each network receiver and maps them to the corresponding input DStreams
- **Job Scheduler** – Periodically queries the DStream graph to generate Spark jobs from received data, and hands them to Job Manager for execution
- **Job Manager** – Maintains a job queue and executes the jobs in Spark

# Execution Model – Receiving Data



# Execution Model – Job Scheduling



# Job Scheduling

- Each output operation used generates a job
  - More jobs → more time taken to process batches → higher batch duration

- Job Manager decides how many concurrent Spark jobs to run
  - Default is 1, can be set using Java property `spark.streaming.concurrentJobs`
  - If you have multiple output operations, you can try increasing this property to reduce batch processing times and so reduce batch duration

# Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

# DStream Persistence

- If a DStream is set to persist at a storage level, then all RDDs generated by it set to the same storage level
- When to persist?
  - If there are multiple transformations / actions on a DStream
  - If RDDs in a DStream is going to be used multiple times
- Window-based DStreams are automatically persisted in memory

# DStream Persistence

- Default storage level of DStreams is `StorageLevel.MEMORY_ONLY_SER` (i.e. in memory as serialized bytes)
  - Except for input DStreams which have `StorageLevel.MEMORY_AND_DISK_SER_2`
  - Note the difference from RDD's default level (no serialization)
  - Serialization reduces random pauses due to GC providing more consistent job processing times

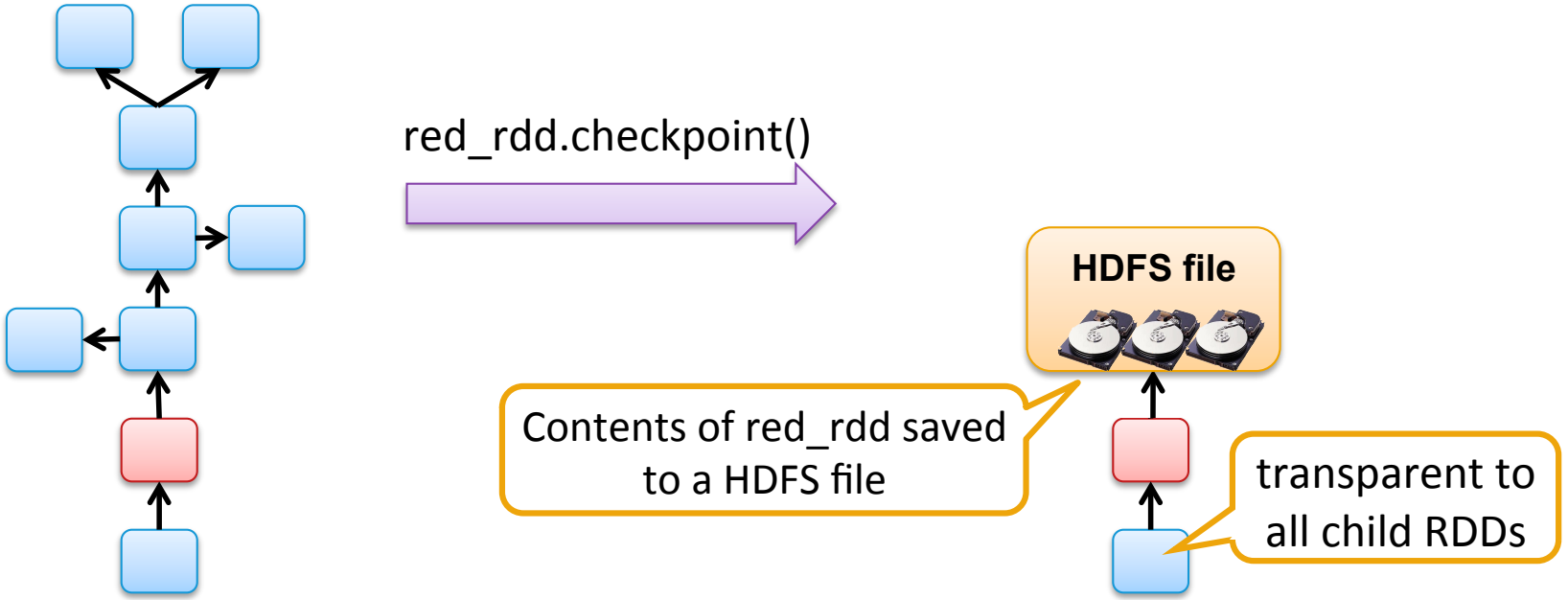
# Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

# What is RDD checkpointing?

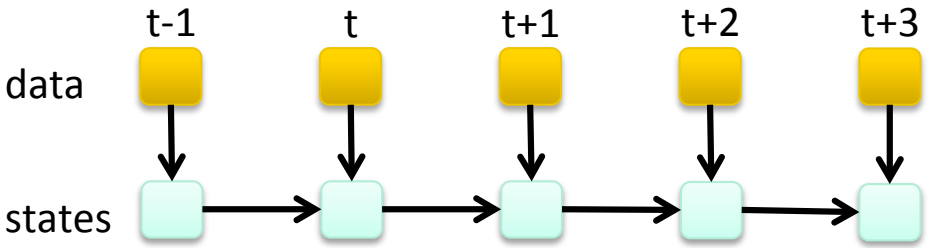
Saving RDD to HDFS to prevent RDD graph from growing too large

- Done internally in Spark transparent to the user program
- Done lazily, saved to HDFS the first time it is computed



# Why is RDD checkpointing necessary?

Stateful DStream operators can have infinite lineages

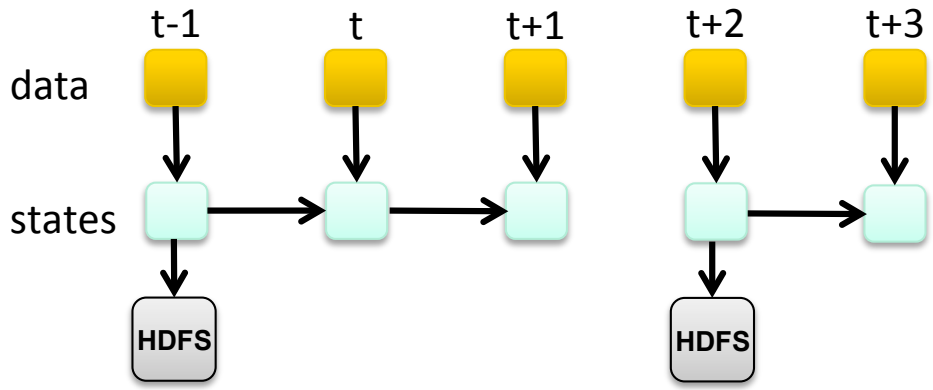



Large lineages lead to …

- Large closure of the RDD object → large task sizes → high task launch times
- High recovery times under failure

# Why is RDD checkpointing necessary?

Stateful DStream operators can have infinite lineages




Periodic RDD checkpointing solves this

Useful for iterative Spark programs as well

# RDD Checkpointing

- Periodicity of checkpoint determines a tradeoff
  - Checkpoint too frequent: HDFS writing will slow things down
  - Checkpoint too infrequent: Task launch times may increase
  - Default setting checkpoints at most once in 10 seconds
  - Try to checkpoint once in about 10 batches

# Agenda

- Overview
- DStream Abstraction
- System Model
- Persistence / Caching
- RDD Checkpointing
- Performance Tuning

# Performance Tuning

## Step 1

Achieve a stable configuration that can sustain the streaming workload

## Step 2

Optimize for lower latency

# Step 1: Achieving Stable Configuration

**How to identify whether a configuration is stable?**

- Look for the following messages in the log

```
Total delay: 0.01500 s for job 12 of time 1371512674000 …
```

- If the total delay is continuously increasing, then unstable as the system is unable to process data as fast as its receiving!

- If the total delay stays roughly constant and around 2x the configured batch duration, then stable

# Step 1: Achieving Stable Configuration

**How to figure out a good stable configuration?**

- Start with a low data rate, small number of nodes, reasonably large batch duration (5 – 10 seconds)
- Increase the data rate, number of nodes, etc.
- Find the bottleneck in the job processing
  - Jobs are divided into stages
  - Find which stage is taking the most amount of time

# Step 1: Achieving Stable Configuration

**How to figure out a good stable configuration?**

- If the first map stage on raw data is taking most time, then try …
  - Enabling delayed scheduling by setting property `spark.locality.wait`
  - Splitting your data source into multiple sub streams
  - Repartitioning the raw data into many partitions as first step
- If any of the subsequent stages are taking a lot of time, try…
  - Try increasing the level of parallelism (i.e., increase number of reducers)
  - Add more processors to the system

# Step 2: Optimize for Lower Latency

- Reduce batch size and find a stable configuration again
  - Increase levels of parallelism, etc.
- Optimize serialization overheads
  - Consider using Kryo serialization instead of the default Java serialization for both data and tasks
  - For data, set property

    `spark.serializer=spark.KryoSerializer`
  - For tasks, set

    `spark.closure.serializer=spark.KryoSerializer`
- Use Spark stand-alone mode rather than Mesos

# Step 2: Optimize for Lower Latency

- Using concurrent mark sweep GC `-XX:+UseConcMarkSweepGC` is recommended
  - Reduces throughput a little, but also reduces large GC pauses and may allow lower batch sizes by making processing time more consistent
- Try disabling serialization in DStream/RDD persistence levels
  - Increases memory consumption and randomness of GC related pauses, but may reduce latency by further reducing serialization overheads
- For a full list of guidelines for performance tuning
  - Spark Tuning Guide
  - Spark Streaming Tuning Guide

# System Stability

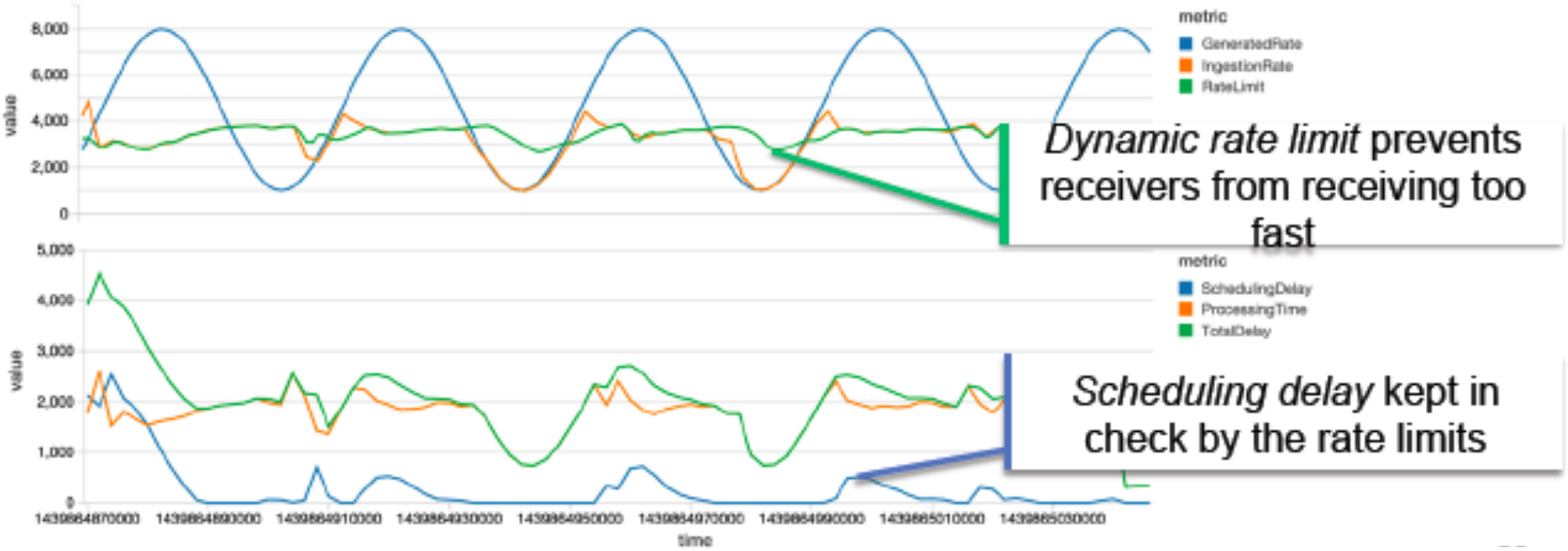
- Streaming applications may have to deal with variations in data rates and processing rates
- For stability, any streaming application must receive data only as fast as it can process
- Static rate limits on receivers [Spark 1.1]
  - But hard to figure out the right rate

# Backpressure [Spark 1.5]

- System automatically and dynamically adapts rate limits to ensure stability under any processing conditions
- If sinks slow down, then the system automatically pushes back on the source to slow down receiving
- System uses batch processing times and scheduling delays experienced to set rate limits
- Well known PID controller theory (used in industrial control systems) is used to calculate appropriate rate limit
  - Contributed by Typesafe
- Enabled by setting Spark configuration flag
  - `spark.streaming.backpressure.enabled` to `TRUE`

# Backpressure [Spark 1.5]

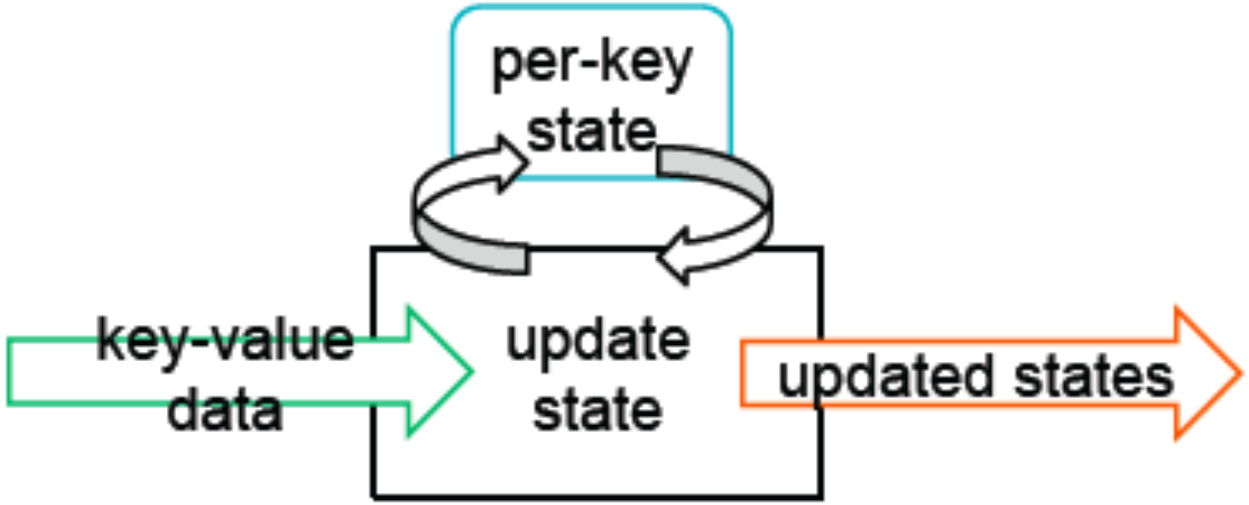
- System automatically and dynamically adapts rate limits



# Improved State Management

Earlier stateful stream processing done with `updateStateByKey`

```
def stateUpdateFunc(newData, lastState) => updatedState

val stateDStream = keyValueDStream.updateStateByKey(stateUpdateFunc)
```



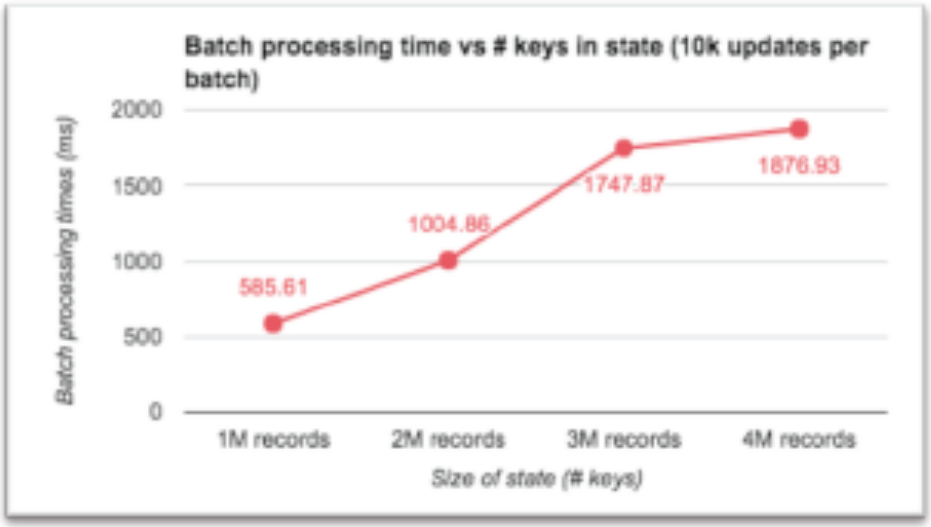
# Improved State Management

Feedback from community about `updateStateByKey`

Need to keep much larger state

- Processing times of batches increase with the amount state, limits performance

Need to expire keys that have received no data for a while



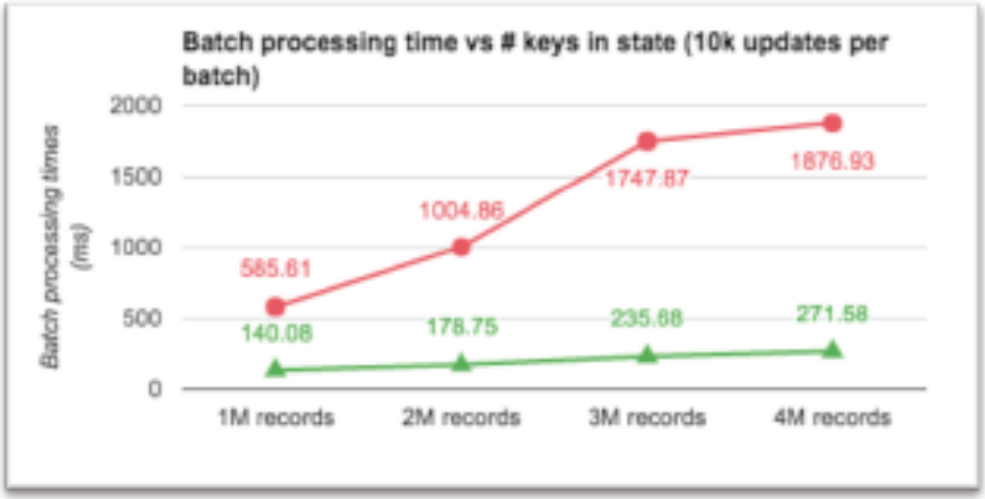
# Improved State Management

## New API with timeouts: `trackStateByKey`

```
def updateFunc(values, state) => emittedData
  // call state.update(newState) to update state

keyValueDStream.trackStateByKey(
  StateSpec.function(updateFunc).timeout(Minutes(10))
```

Can provide order of magnitude higher performance than updateStateByKey

https://issues.apache.org/jira/browse/SPARK-2629



# Visualizations

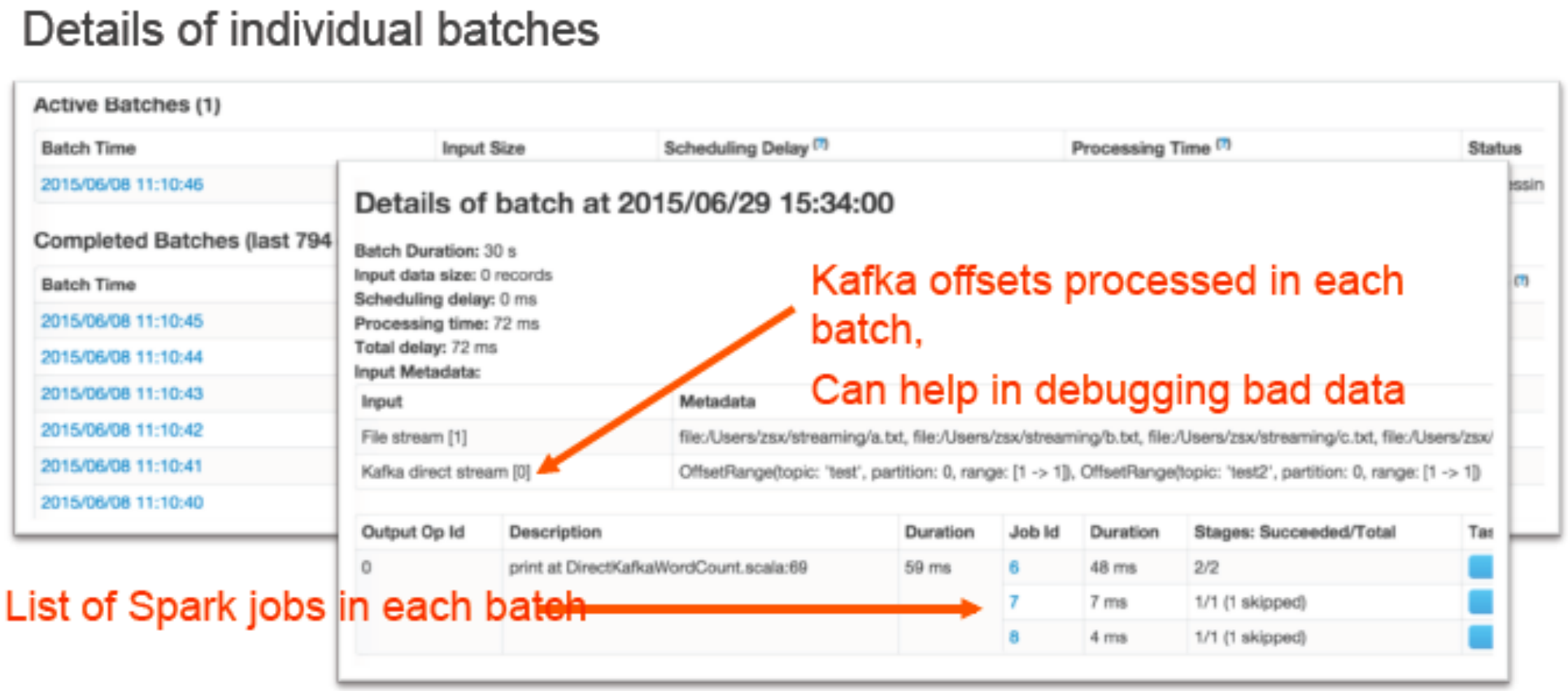

Stats over last 1000 batches

For stability

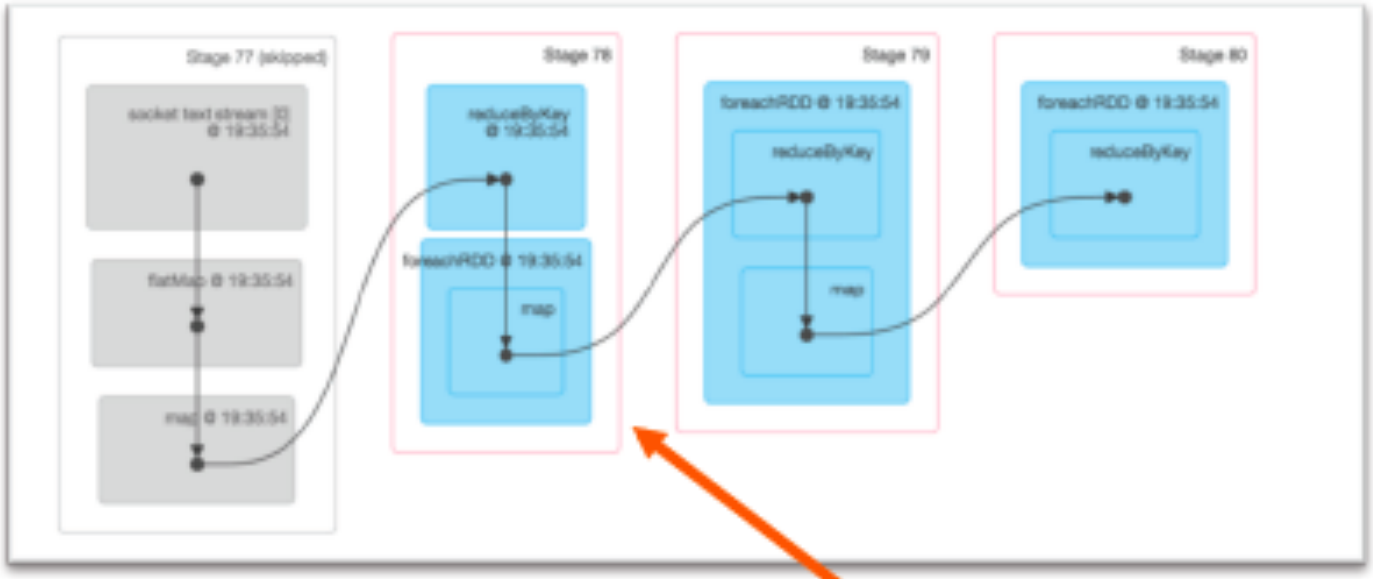
Scheduling delay should be approx 0

Processing Time approx < batch interval

# Visualizations



# Visualizations

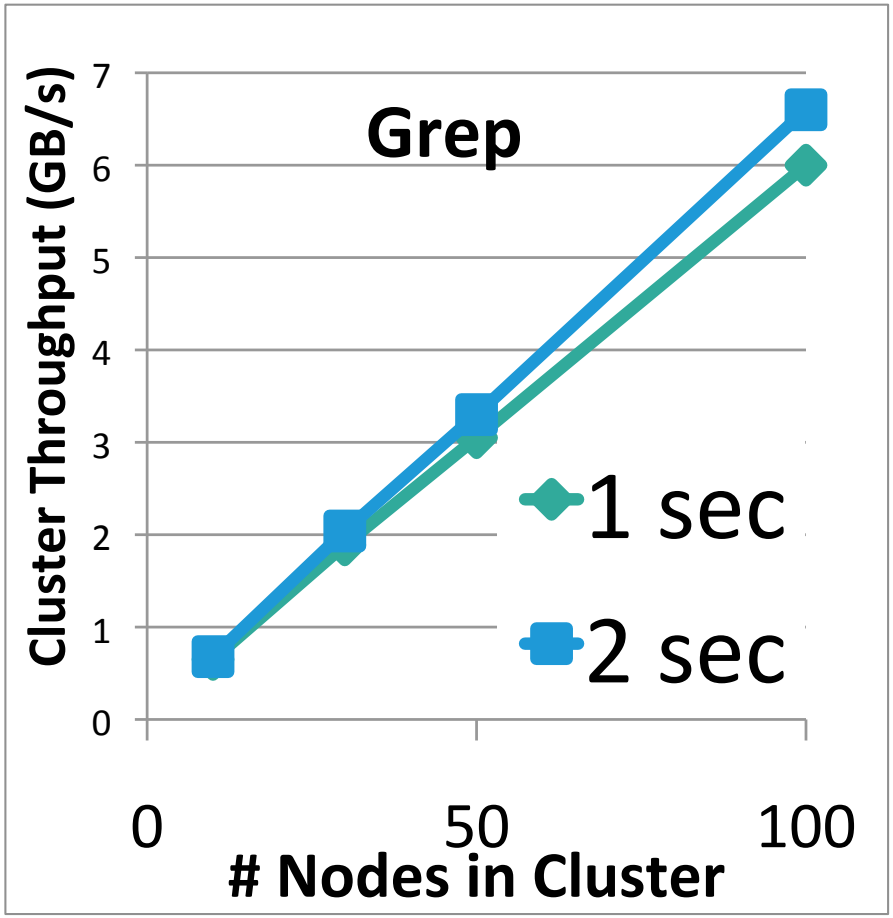


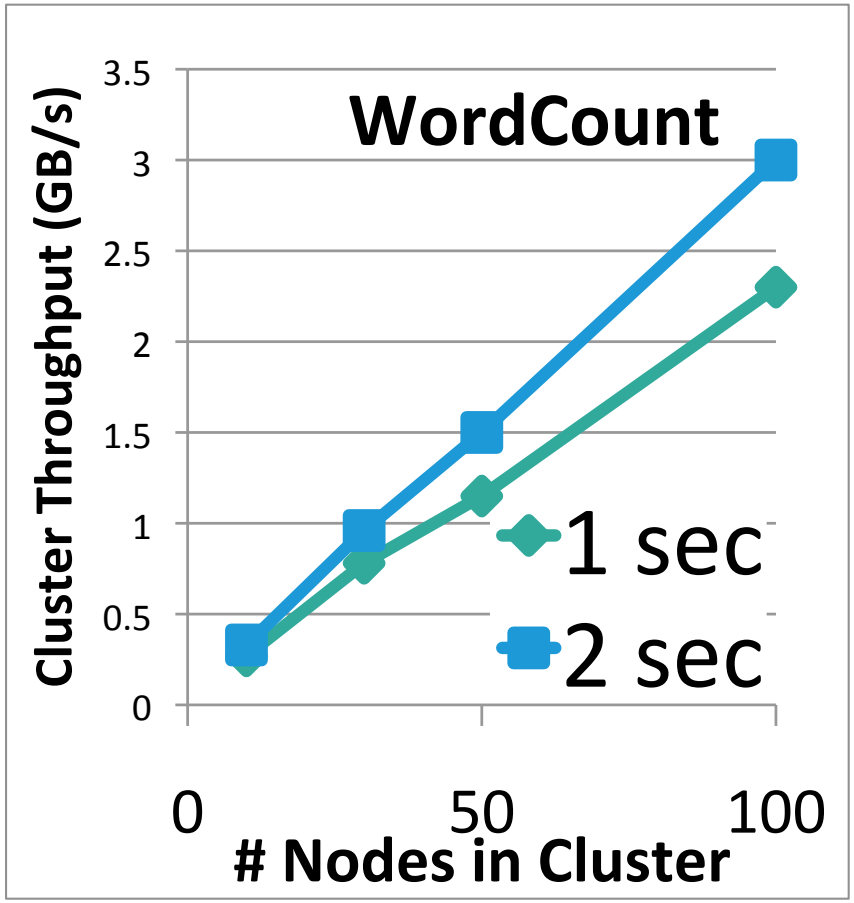

Full DAG of RDDs and stages generated by Spark Streaming

# Performance

Can process **6 GB/sec (60M records/sec**) of data on 100 nodes at **sub-second** latency

- Tested with 100 streams of data on 100 EC2 instances with 4 cores each

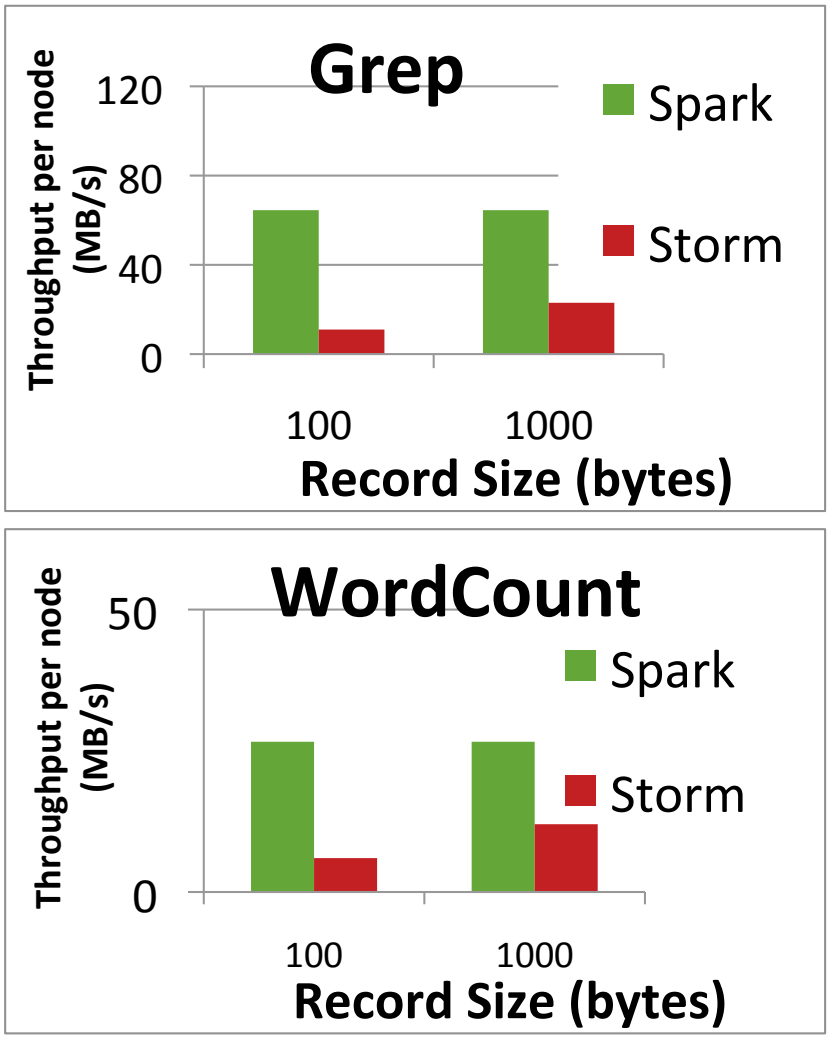


# Comparison with Storm and S4

## Higher throughput than Storm

- Spark Streaming: **670k** records/second/node
- Storm: **115k** records/second/node
- Apache S4: 7.5k records/second/node



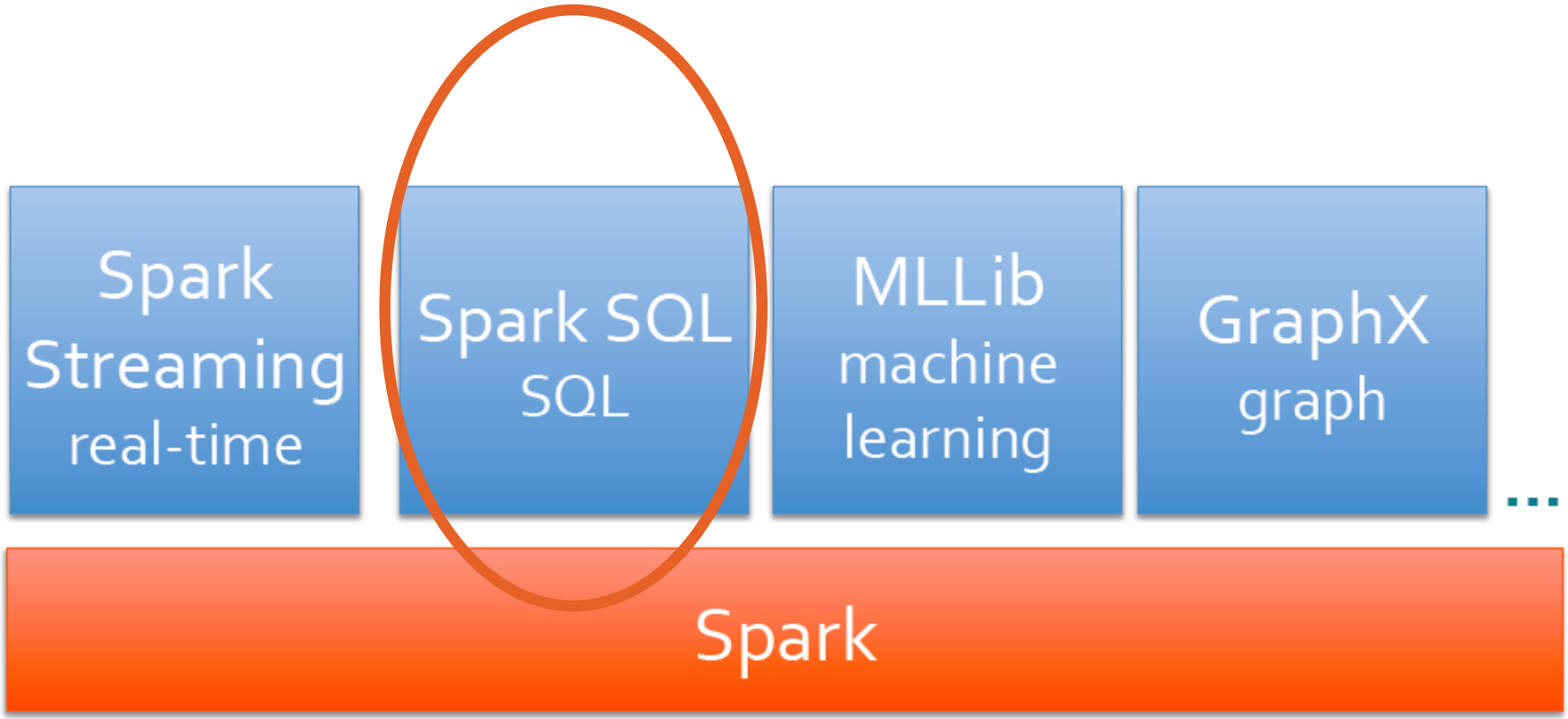
# Small code base

- 5000 LOC for Scala API (+ 1500 LC for Java API)
  - Most DStream code mirrors the RDD code

# Future Extensions on Spark Streaming

- API and Libraries Support of Streaming DataFrames
  - Logical-to-physical plan optimizations
  - Tungsten-based binary optimizations
  - Support for event-time based windowing
  - Support for Out-of-Order Data
- Add Nature Infrastructure support for Dynamic Allocation for Streaming
  - Dynamically scale the cluster resources based on processing load
  - Need to work with Backpressure to scale up/down while maintaining stability
- Programmable monitoring by exposing more info via Streaming Listener
- Performance Enhancement: higher throughput and lower latency, specially for Stateful Ops, e.g. trackStateByKey

# Generality of RDDs in Spark



# Before SQL support was available from Spark


- The Spark Core Engine does not understand the structure of the data in RDDs or the semantics of user functions → limited optimization.
- However, most data is structured, e.g. JSON, CSV, Avro, Parquet, Hive, etc

=> Programming/ Operations via the RDD API inevitably ends up with a lot of tuples ( _1, _2, …)

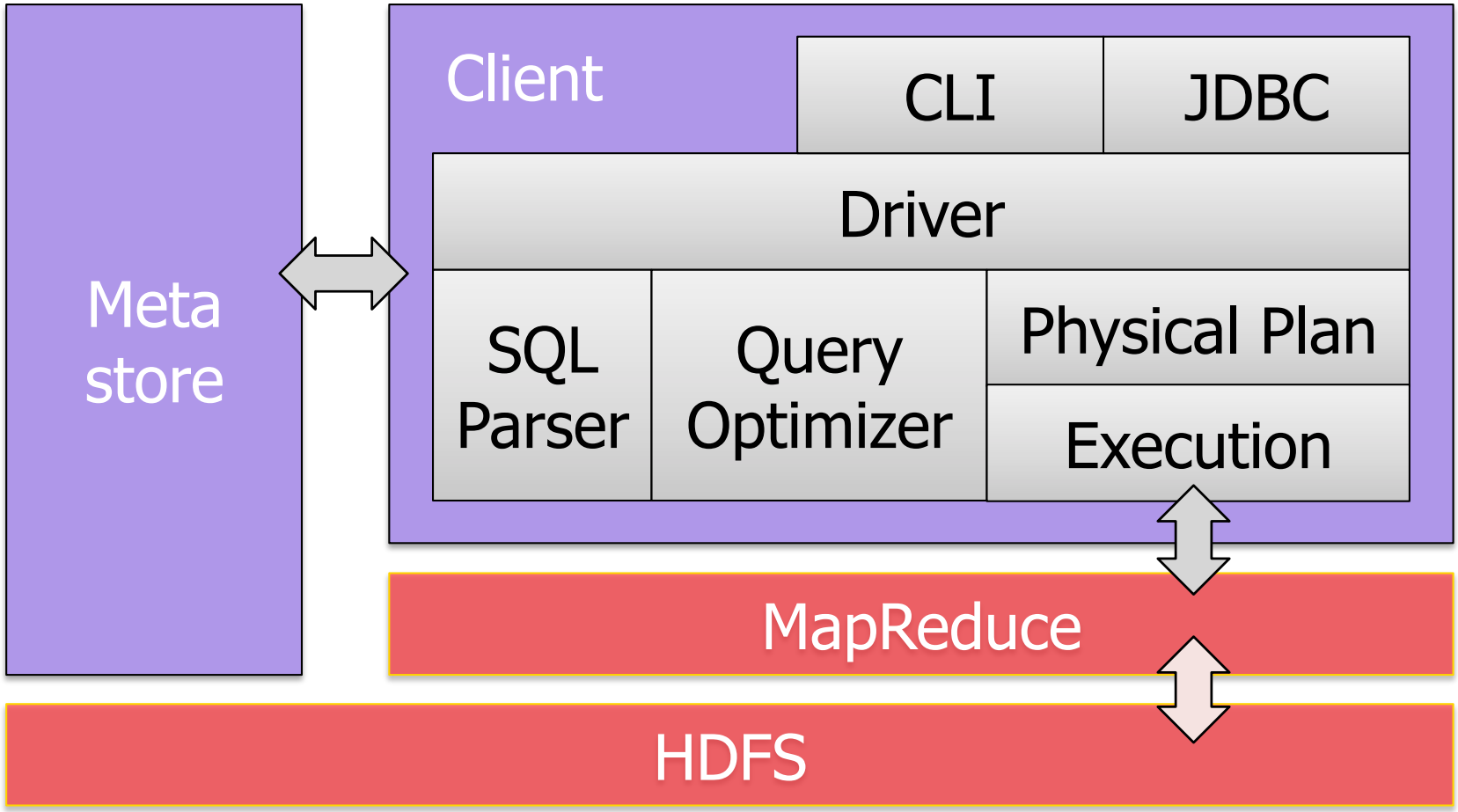
- Functional Transformations, e.g. Map/Reduce are still not as Intuitive as SQL for a lot of Experienced System/Data Analysts.

# SQL support in Spark - Take 1: The Shark Story

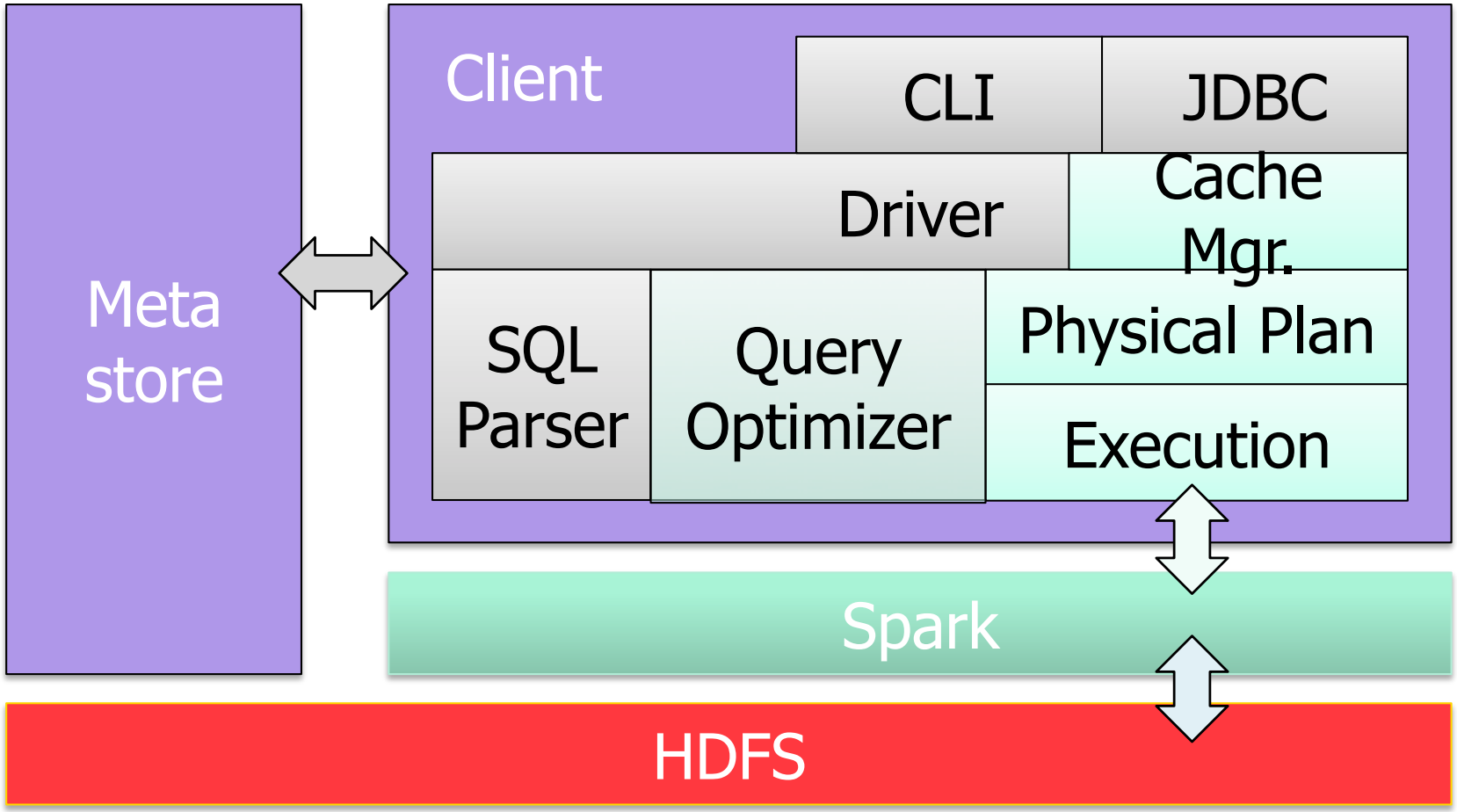
- Hive is great, but Hadoop's execution engine makes even the smallest queries take minutes
- Scala is good for programmers, but many data users only know SQL
- **Initial Approach: Make Hive to run on Spark**


**= Hive on Spark**

# Original Hive Architecture



# Shark Architecture

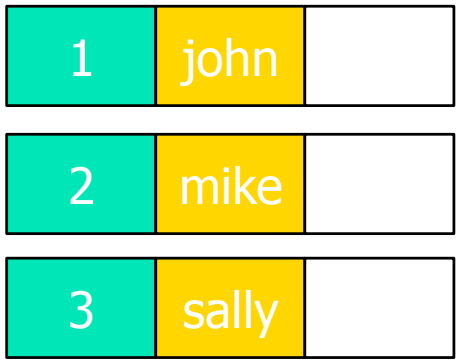



[Engle et al, SIGMOD 2012]

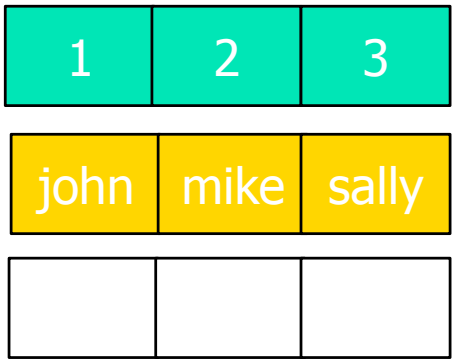
# Efficient In-Memory Storage

- Simply caching Hive records as Java objects is inefficient due to high per-object overhead
- Instead, Shark employs column-oriented storage using **arrays of primitive types**

**Row Storage**

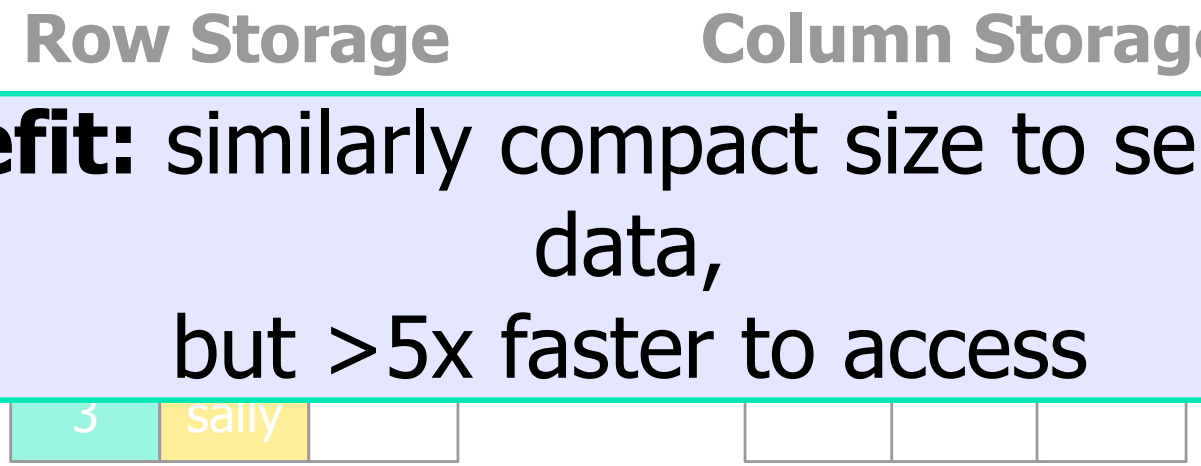



**Column Storage**



# Efficient In-Memory Storage

- Simply caching Hive records as Java objects is inefficient due to high per-object overhead
- Instead, Shark employs column-oriented storage using **arrays of primitive types**




**Benefit:** similarly compact size to serialized data,
but >5x faster to access

# But Shark was short-lived (2011-2014)

Limitations of SHARK

- Can only be used to query external data in Hive catalog → limited data sources
- Can only be invoked via SQL string from Spark
  → error prone
- Hive optimizer tailored for MapReduce
  → difficult to extend

- As a result, BDAS Project decided to build switch to Spark SQL and stopped development of Shark in 2014
  - The Apache Hive community still runs a Hive-over-Spark effort, as well as the Stinger/ Stinger.Next efforts to make Hive/HiveQL to be SQL compatible and low-latency

# Take 2: Spark SQL Overview

- Part of the core distribution since Spark 1.0 (April 2014)
  - Optionally alongside or replacing existing Hive deployments
- Run SQL/ HiveQL queries including UDFs, UDAFs and SerDes, e.g.


```
SELECT COUNT(*)
FROM hiveTable
WHERE hive_udf(data)
```

- Connect existing Business Intelligence (BI) tools to Spark through JDBC




- Bindings in Python, Scala and Java



# The Approach of Spark SQL

- Introduce a Tightly Integrated way to work with a new abstraction of Structured Data called SchemaRDD, which is a Distributed Collection of Rows (i.e. a Table) with Named Columns
  - SchemaRDD was renamed to DataFrame in Spark 1.3

- Support the Transformation of RDDs using SQL: In particular, DataFrames (aka SchemaRDDs) is an abstraction which supports:
  - Selecting, Filtering, Aggregating and Plotting Structured data (cf. R or Python-based Pandas)
- Evaluated lazily → unmaterialized *logical* plan

- Data source integration Support for: Hive, Parquet, JSON and …

# Relationship between Spark SQL and Shark

- Shark modified the Hive backend to run over Spark but had two challenges:
  - Limited integration with Spark programs
  - Hive Optimizer not designed for Spark

- Spark SQL reuses some parts of Shark by

Borrowing:

  - Hive Data Loading
  - In-memory Column-store

while Adding:

  - RDD-aware Optimizer
  - Richer Language Interfaces

# What is an RDD ?

- Dependencies
- Partitions (with optional locality information)
- Compute Function: `Partition=>Iterator[T]`

Opaque Computation

# What is an RDD ?

- Dependencies
- Partitions (with optional locality information)
- Compute Function: `Partition=>Iterator[T]`

Opaque Data

# Why Structure ?

- What do we mean by “Structure” [verb] ?:
  - Construct or Arrange according to a plan ; Give a pattern or organization to.
- By definition, structure will LIMIT what can be expressed.
- In practice, it is still possible to accommodate a vast majority of computations

BUT

- By Limiting the space of what can be expressed ENABLES Optimization

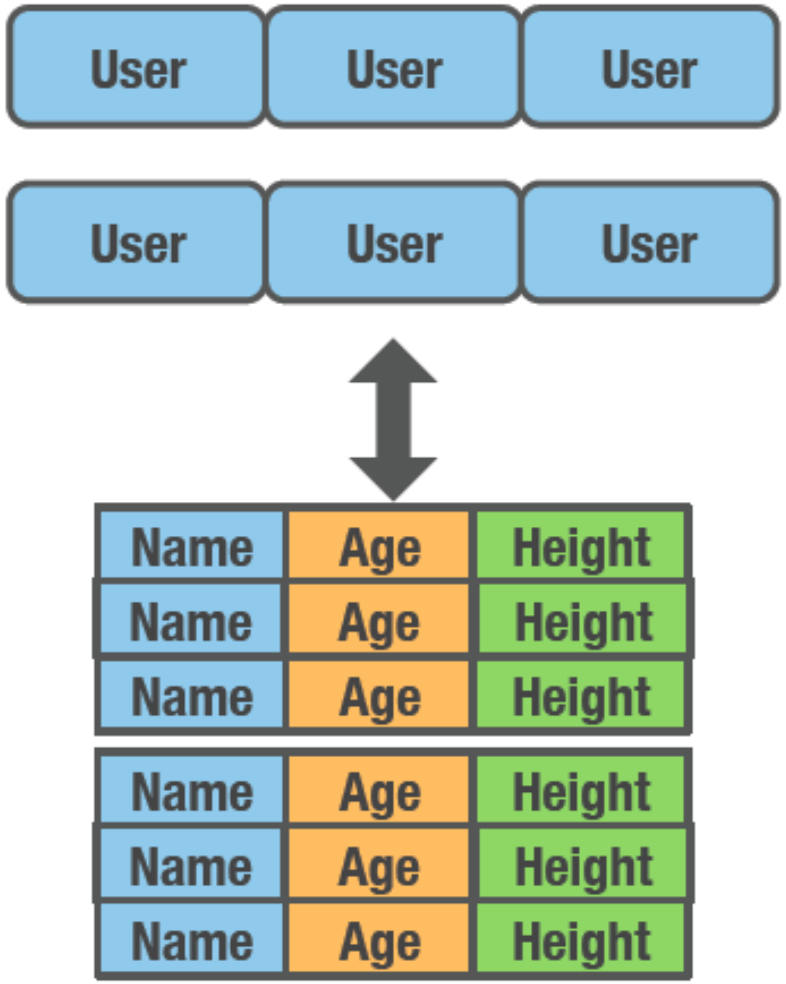
# Adding Schema to RDDs

Spark + RDDs

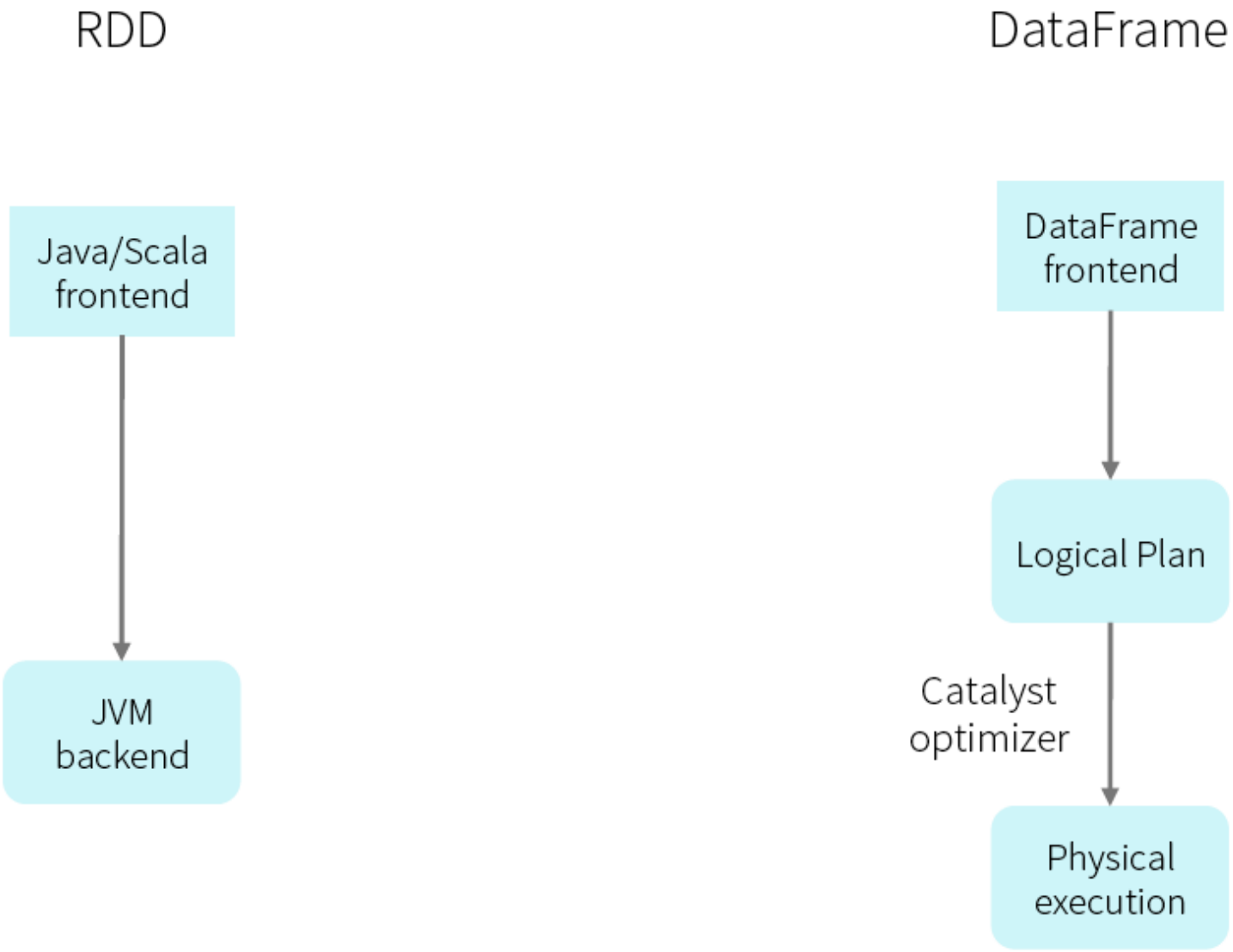
- Functional transformations on Partitioned Collections of *Opaque Objects*

SQL + DataFrames (aka SchemaRDDs)

- Declarative transformations on Partitioned Collections of *Tuples*



# Comparing the Approaches of RDD vs. DataFrame



# Data Model for DataFrame

- Nested data model
- Supports both primitive SQL types (boolean, integer, double, decimal, string, data, timestamp) and complex types (structs, arrays, maps, and unions); also user defined types.
- First class support for complex data types

# DataFrame Operations

- Relational operations (select, where, join, groupBy) via a DSL
- Operators take *expression* objects
- Operators build up an abstract syntax tree (AST), which is then optimized by *Catalyst*.

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

- Alternatively, register as temp SQL table and perform traditional SQL query strings

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

# Programming Interface for Spark SQL

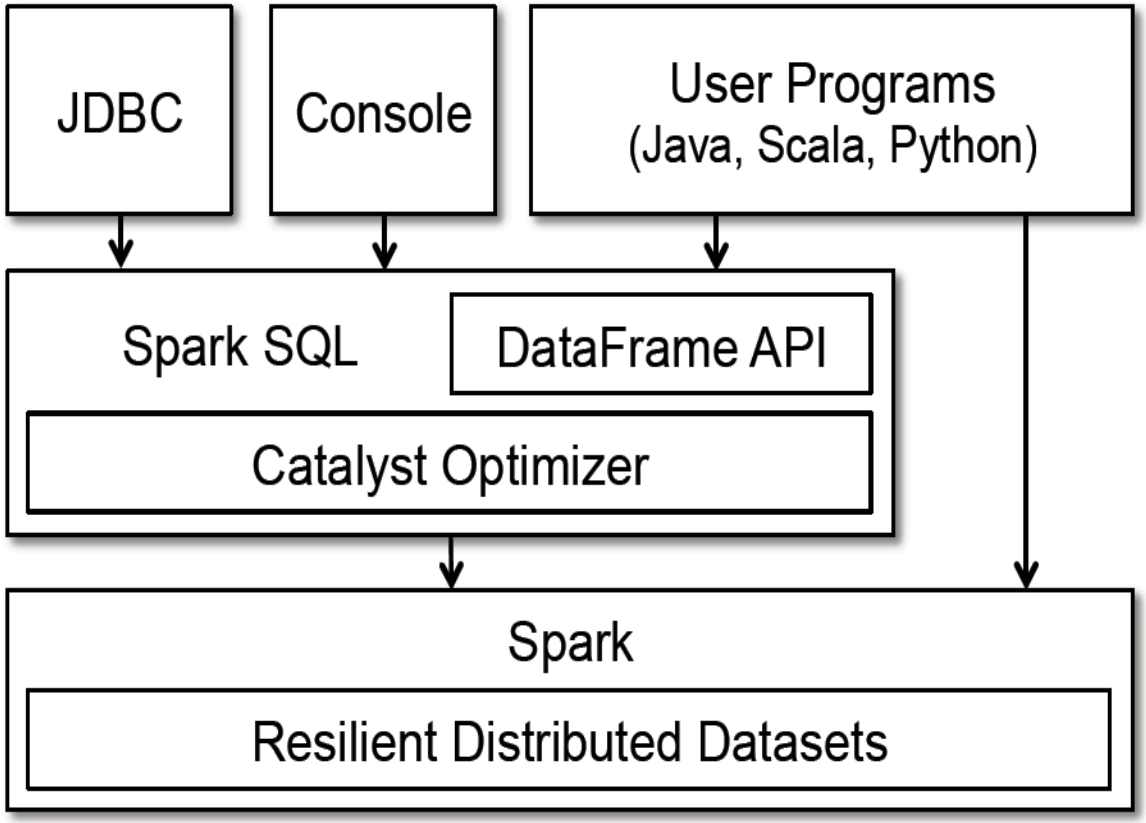
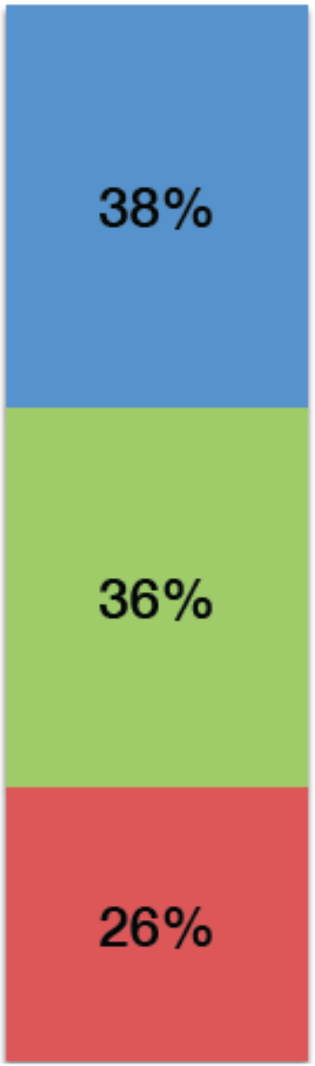



**Figure 1: Interfaces to Spark SQL, and interaction with Spark.**

# Spark SQL Components




- Catalyst Optimizer
  - Relational algebra + expressions
  - Query optimization
- Spark SQL Core
  - Execution of queries as RDDs
  - Reading in Parquet, JSON …
- Hive Support
  - HQL, MetaStore, SerDes, UDFs

# Getting Started: Spark SQL

- `SQLContext/ HiveContext`
  - Entry point for all SQL functionality
  - Wraps/Extend existing Spark Context

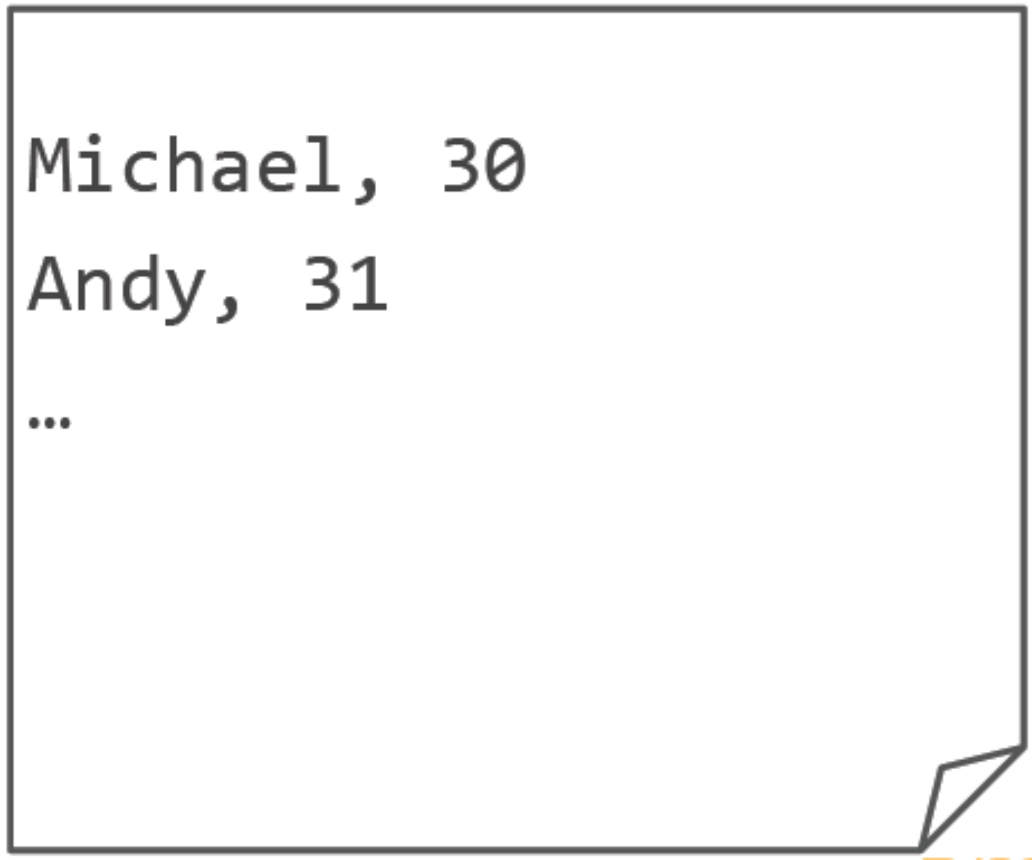
```
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
```

OR

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```

# Sample Input Data

- A text file filled with people's names and ages:



# RDDs as Relations (Scala)

```scala
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._

// Define the schema using a case class.
case class Person(name: String, age: Int)
// Create an RDD of Person objects and register it as a table.
val people =
  sc.textFile("examples/src/main/resources/people.txt")
    .map(_.split(","))
    .map(p => Person(p(0), p(1).trim.toInt))

people.registerAsTable("people")
```

# RDDs as Relations (Python)

```
# Load a text file and convert each line to a dictionary.
lines = sc.textFile("examples/…/people.txt")

parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0],age=int(p[1])))

# Infer the schema, and register the SchemaRDD as a table
peopleTable = sqlCtx.inferSchema(people)
peopleTable.registerAsTable("people")
```

# RDDs as Relations (Java)

```
public class Person implements Serializable {
  private String _name;
  private int _age;
  public String getName() { return _name;  }
  public void setName(String name) { _name = name; }
  public int getAge() { return _age; }
  public void setAge(int age) { _age = age; }
}

JavaSQLContext ctx = new org.apache.spark.sql.api.java.JavaSQLContext(sc)
JavaRDD<Person> people = ctx.textFile("examples/src/main/resources/
people.txt").map(
  new Function<String, Person>() {
    public Person call(String line) throws Exception {
      String[] parts = line.split(",");
      Person person = new Person();
      person.setName(parts[0]);
      person.setAge(Integer.parseInt(parts[1].trim()));
      return person;
    }
  });

JavaSchemaRDD schemaPeople = sqlCtx.applySchema(people, Person.class);
```

# Querying using Spark SQL (Python)

```
# SQL can be run over SchemaRDDs that have been registered
# as a table.
teenagers = sqlCtx.sql("""
  SELECT name FROM people WHERE age >= 13 AND age <= 19""")

# The results of SQL queries are RDDs and support all the normal
# RDD operations.
teenNames = teenagers.map(lambda p: "Name: " + p.name)
```

# Support of Existing Tools, and New Data Sources

- SparkSQL includes a server that exposes its data using JDBC/ODBC
  - Query data from HDFS/S3
  - Including formats like Hive/Parquet/JSON
  - Support for caching data IN-MEMORY

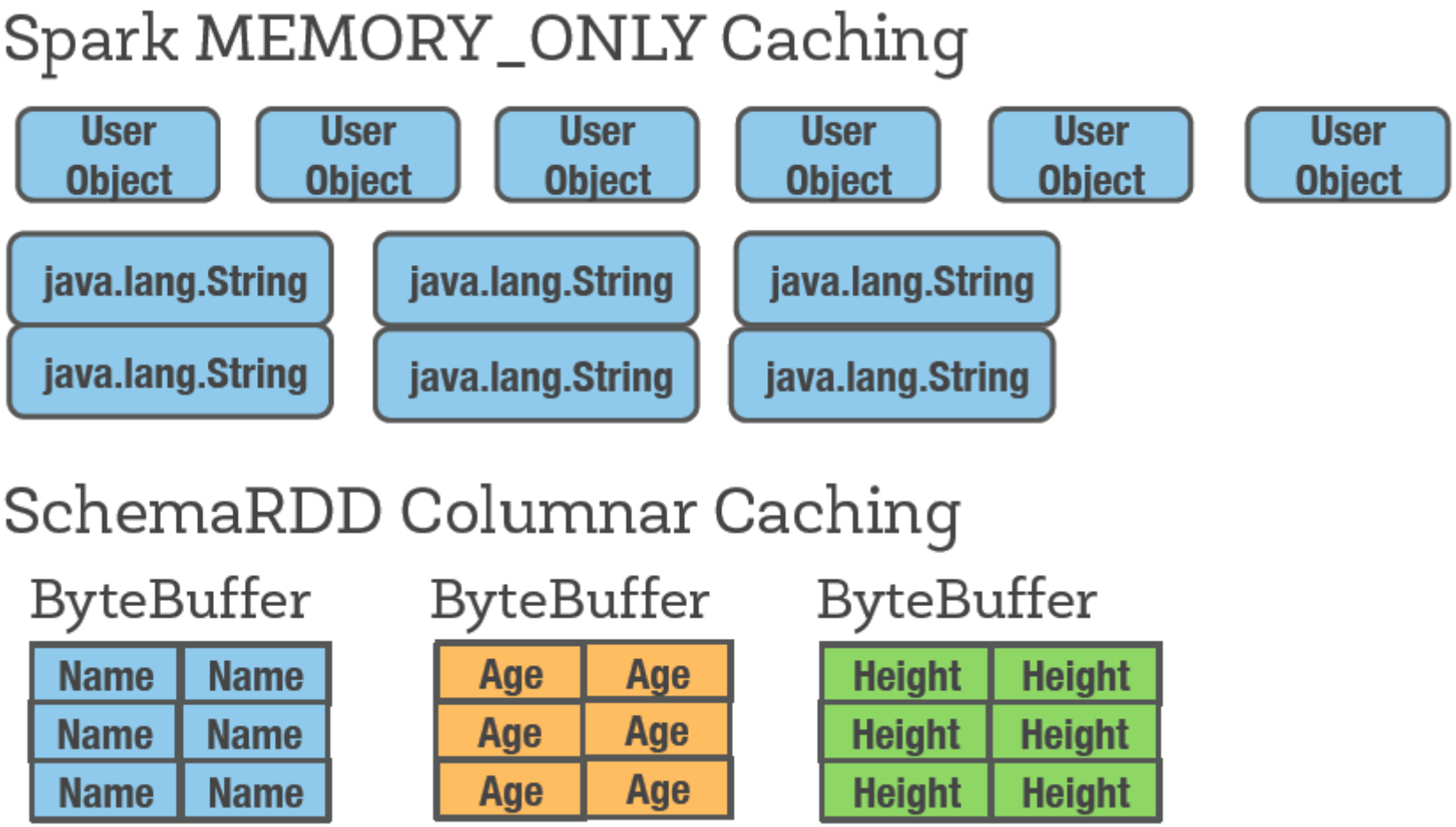
# Caching Tables In-Memory

- SparkSQL can cache tables using an in-memory columnar format:
  - Scan only required columns
  - Fewer allocated objects (less Garbage Collection)
  - Automatically selects best compression

  e.g.

  cacheTable(“people”)  or dataframe.cache( )

# Caching Comparison



# Language Integrated UDFs

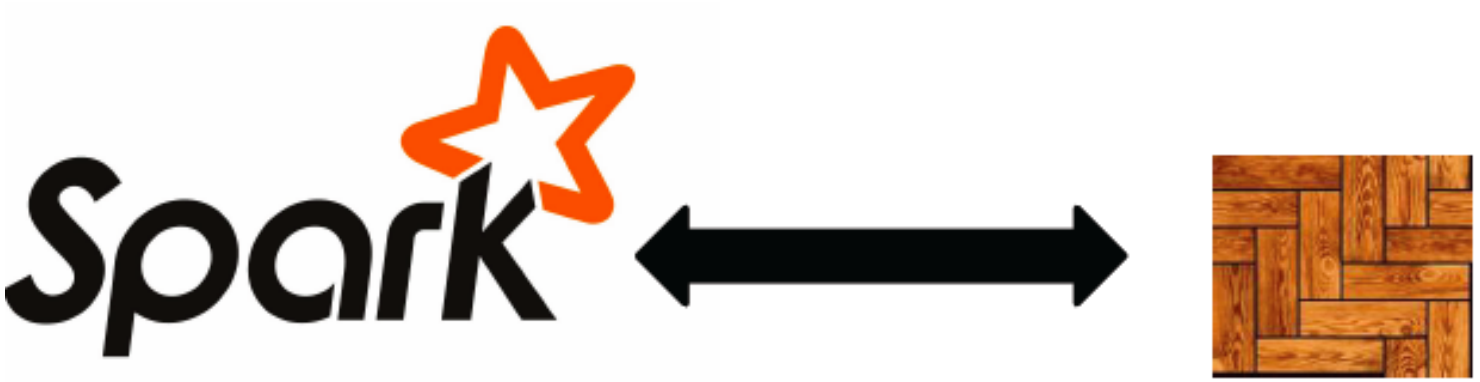
```
registerFunction("countMatches",
  lambda (pattern, text):
    re.subn(pattern, '', text)[1])

sql("SELECT countMatches('a', text)…")
```

# Reading Data stored in Hive

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)

hiveCtx.hql("""
  CREATE TABLE IF NOT EXISTS src (key INT, value STRING)""")

hiveCtx.hql("""
  LOAD DATA LOCAL INPATH 'examples/…/kv1.txt' INTO TABLE src""")

# Queries can be expressed in HiveQL.
results = hiveCtx.hql("FROM src SELECT key, value").collect()
```

# Parquet Compatibility

- Native support for reading data in Parquet
  - Columnar storage avoids reading unneeded data
  - RDDs can be written to Parquet files, preserving the schema
  - Convert other slower formats into Parquet for repeated querying



# Using Parquet

```
# SchemaRDDs can be saved as Parquet files, maintaining the
# schema information.
peopleTable.saveAsParquetFile("people.parquet")

# Read in the Parquet file created above.  Parquet files are
# self-describing so the schema is preserved. The result of
# loading a parquet file is also a SchemaRDD.
parquetFile = sqlCtx.parquetFile("people.parquet”)

# Parquet files can be registered as tables used in SQL.
parquetFile.registerAsTable("parquetFile”)
teenagers = sqlCtx.sql("""
  SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19""")
```

# JSON Support

- Use jsonFile or jsonRDD to convert a collection of JSON objects into a DataFrame
- Infer and Union the schema of each record
- Maintain nested structures and arrays

# JSON Example

```
# Create a SchemaRDD from the file(s) pointed to by path
people = sqlContext.jsonFile(path)

# Visualized inferred schema with printSchema().
people.printSchema()
# root
#  |-- age: integer
#  |-- name: string

# Register this SchemaRDD as a table.
people.registerTempTable("people")
```

# Data Sources API

- Allow easy integration with new sources of structured data:

```
CREATE TEMPORARY TABLE episodes
USING com.databricks.spark.avro
OPTIONS (
  path ”./episodes.avro”
)
```

https://github.com/databricks/spark-avro

# Much More than SQL: DataFrames as A Unified Interface for the Processing of Structured Data

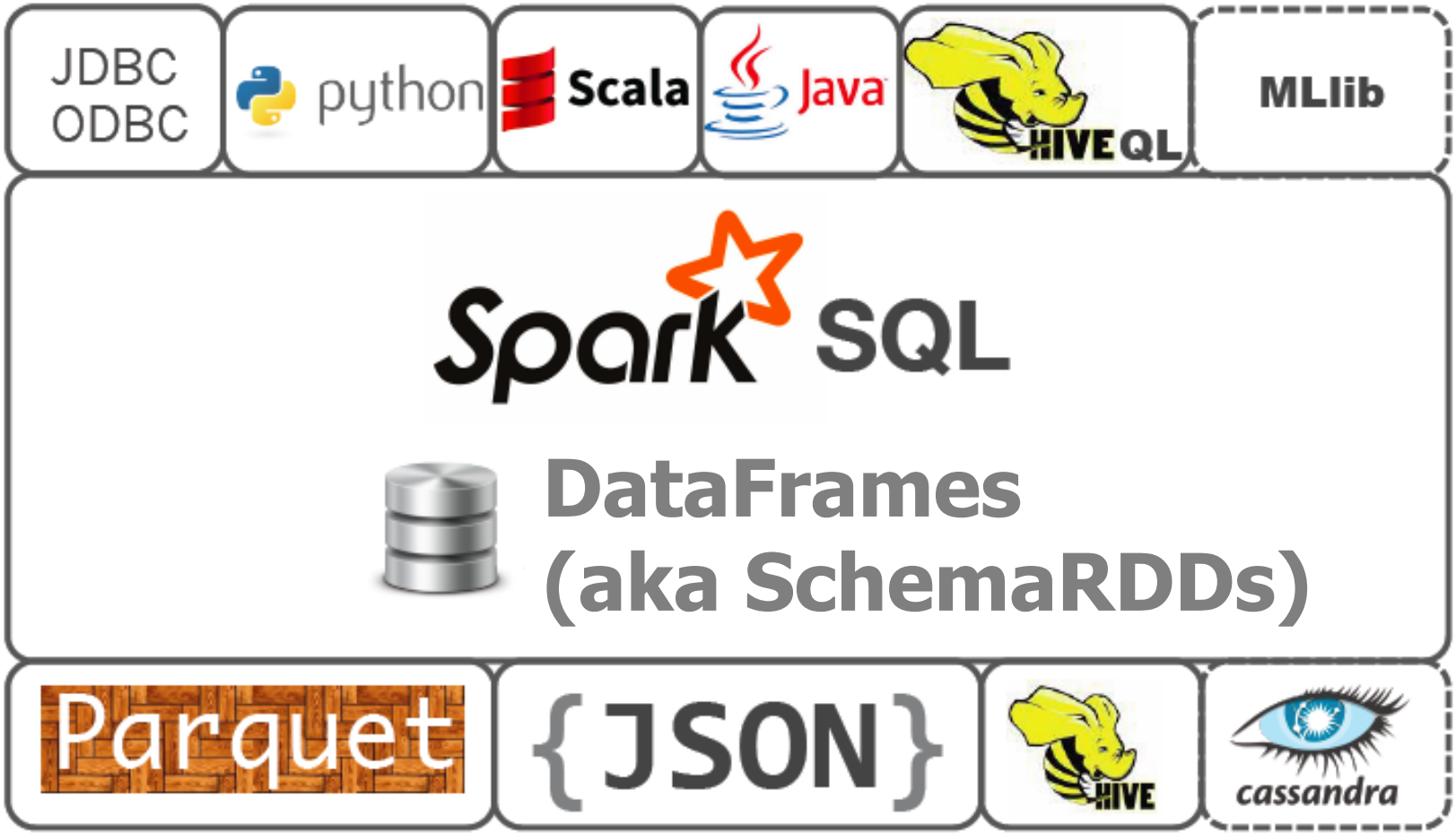


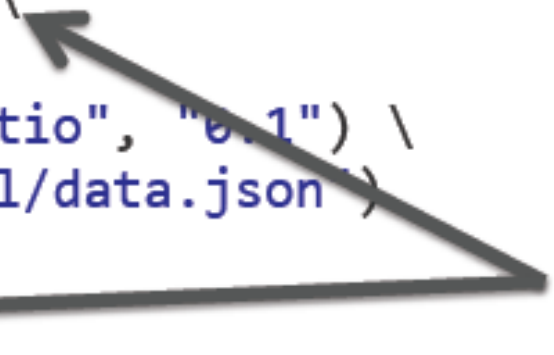

Image credit: http://barrymieny.deviantart.com/

# Much More than SQL: Simplifying Inputs and Outptuts

Spark SQL's Data Source API can read and write DataFrames using a variety of formats.

# Unified and Simplified Interface to Read/ Write Data in Many different Formats

```
df = sqlContext.read \
  .format("json") \
  .option("samplingRatio", "0.1") \
  .load("/home/michael/data.json")

df.write \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```



`read` and `write` functions create new builders for doing I/O

# Unified and Simplified Interface to Read/ Write Data in Many different Formats

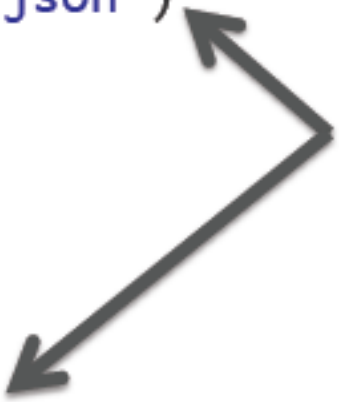
```
df = sqlContext.read \
  .format("json") \
  .option("samplingRatio", "0.1") \
  .load("/home/michael/data.json")

df.write \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```

Builder methods are used to specify:

- Format
- Partitioning
- Handling of existing data
- and more

# Unified and Simplified Interface to Read/ Write Data in Many different Formats

```
df = sqlContext.read \
  .format("json") \
  .option("samplingRatio", "0.1") \
  .load("/home/michael/data.json")

df.write \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```



`load(…)`, `save(…)` or `saveAsTable(…)` functions create new builders for doing I/O

# ETL using Custom Data Sources

```
sqlContext.read
  .format("com.databricks.spark.jira")
  .option("url", "https://issues.apache.org/jira/rest/api/latest/search")
  .option("user", "marmbrus")
  .option("password", "********")
  .option("query", """
    |project = SPARK AND
    |component = SQL AND
    |(status = Open OR status = "In Progress" OR status = Reopened)""".stripMargin)
  .load()
  .repartition(1)
  .write
  .format("parquet")
  .saveAsTable("sparkSqlJira")
```

# Write Less Codes with DataFrames

- Common operations can be expressed concisely as higher level operation calls to the DataFrame API:
  - Selecting required Columns
  - Joining Different Data Sources
  - Aggregation (Count, Sum, Average, etc)
  - Filtering

# Write Less Codes: An Example of Computing Average


```
private IntWritable one =
  new IntWritable(1)
private IntWritable output =
  new IntWritable()
proctected void map(
    LongWritable key,
    Text value,
    Context context) {
  String[] fields = value.split("\t")
  output.set(Integer.parseInt(fields[1]))
  context.write(one, output)
}

IntWritable one = new IntWritable(1)
DoubleWritable average = new DoubleWritable()

protected void reduce(
    IntWritable key,
    Iterable<IntWritable> values,
    Context context) {
  int sum = 0
  int count = 0
  for(IntWritable value : values) {
     sum += value.get()
     count++
    }
  average.set(sum / (double) count)
  context.Write(key, average)
}
```

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [x.[1], 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] +
y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

# Write Less Code: Example of Computing Average

- Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 1])) \
    .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
    .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
    .collect()
```

## Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

## Using Pig

```
P = load '/people' as (name, name);
G = group P by name;
R = foreach G generate … AVG(G.age);
```

## Using DataFrames

```
sqlCtx.table("people") \
    .groupBy("name") \
    .agg("name", avg("age")) \
    .collect()
```

# Read Less Data with DataFrames & SparkSQL

"The fastest way to process big data is to never read it."

- SparkSQL can help the program to read less data automatically by performing BEYOND naïve scanning:
  - Using Columnar formats (e.g. Parquet) and prune irrelevant Columns and Blocks of data
  - Push filters to the source
  - Converting to more efficient formats, e.g. turning string comparisons into integer comparisons for dictionary encoded data
  - Using Partitioning (i.e., /year=2-14/month=02/.. )
  - Skipping data using statistics (i.e. min, max)
  - Pushing predicates into storage systems (i.e. JDBC)

# Intermix DataFrame Operations with Custom Codes (Python, Java, R, Scala)

```
zipToCity = udf(lambda zipCode: <custom logic here>)

def add_demographics(events):
    u = sqlCtx.table("users")
    events \
      .join(u, events.user_id == u.user_id) \
      .withColumn("city", zipToCity(df.zip))
```

Augments any DataFrame that contains `user_id`

Takes and returns a DataFrame

# Integration with RDDs

- Internally, DataFrame execution is done with Spark RDDs

=> Easy Interoperation with outside sources and custom algorithms

## External Input

```
def buildScan(
    requiredColumns: Array[String],
    filters: Array[Filter]): RDD[Row]
```

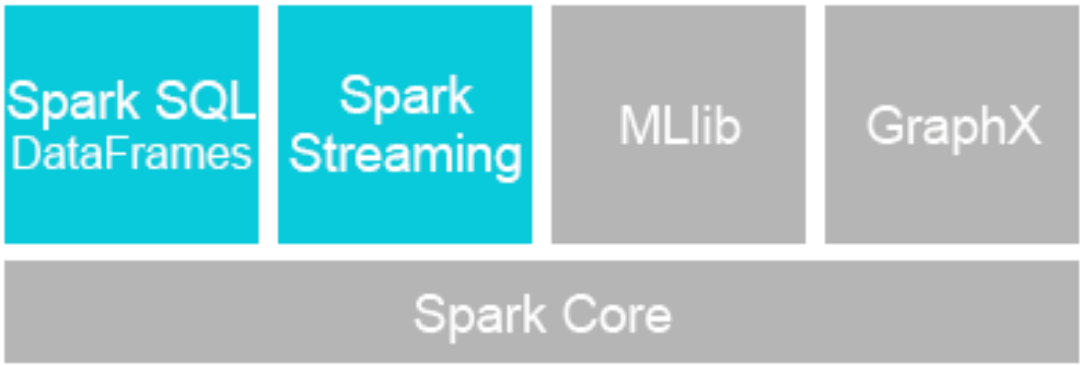
## Custom Processing

```
queryResult.rdd.mapPartitions { iter =>

  … Your code here …

}
```

# Combine SQL with Streaming

- Interactively query streaming data with SQL and Dataframes

```
// Register each batch in stream as table
kafkaStream.foreachRDD { batchRDD =>
  batchRDD.toDF.registerTempTable("events")
}

// Interactively query table
sqlContext.sql("select * from events")
```



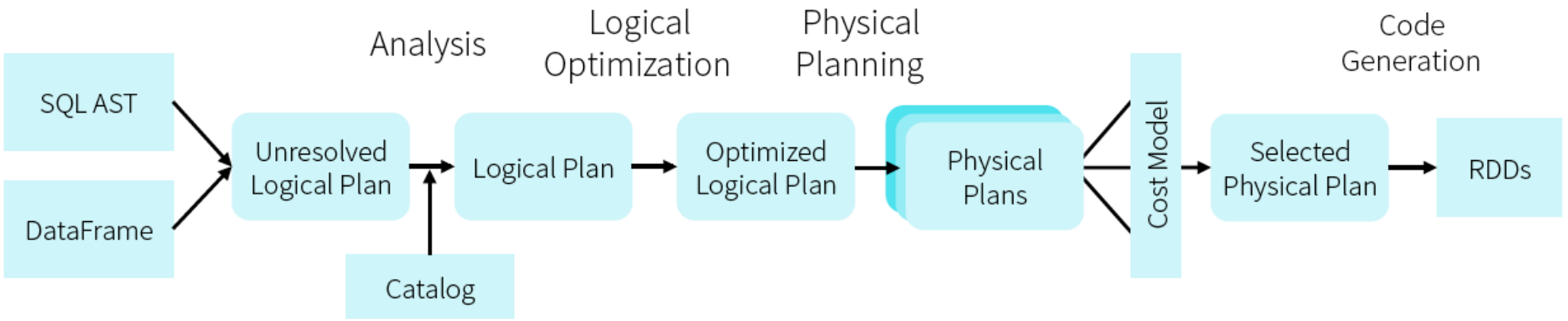
# DataFrame & SparkSQL Demo

Demo:

- *Using Spark SQL to read, write, and transform data in a variety of formats:*

*http://people.apache.org/~marmbrus/talks/dataframe.demo.pdf*

# Plan Optimization and Execution for the entire Pipelines



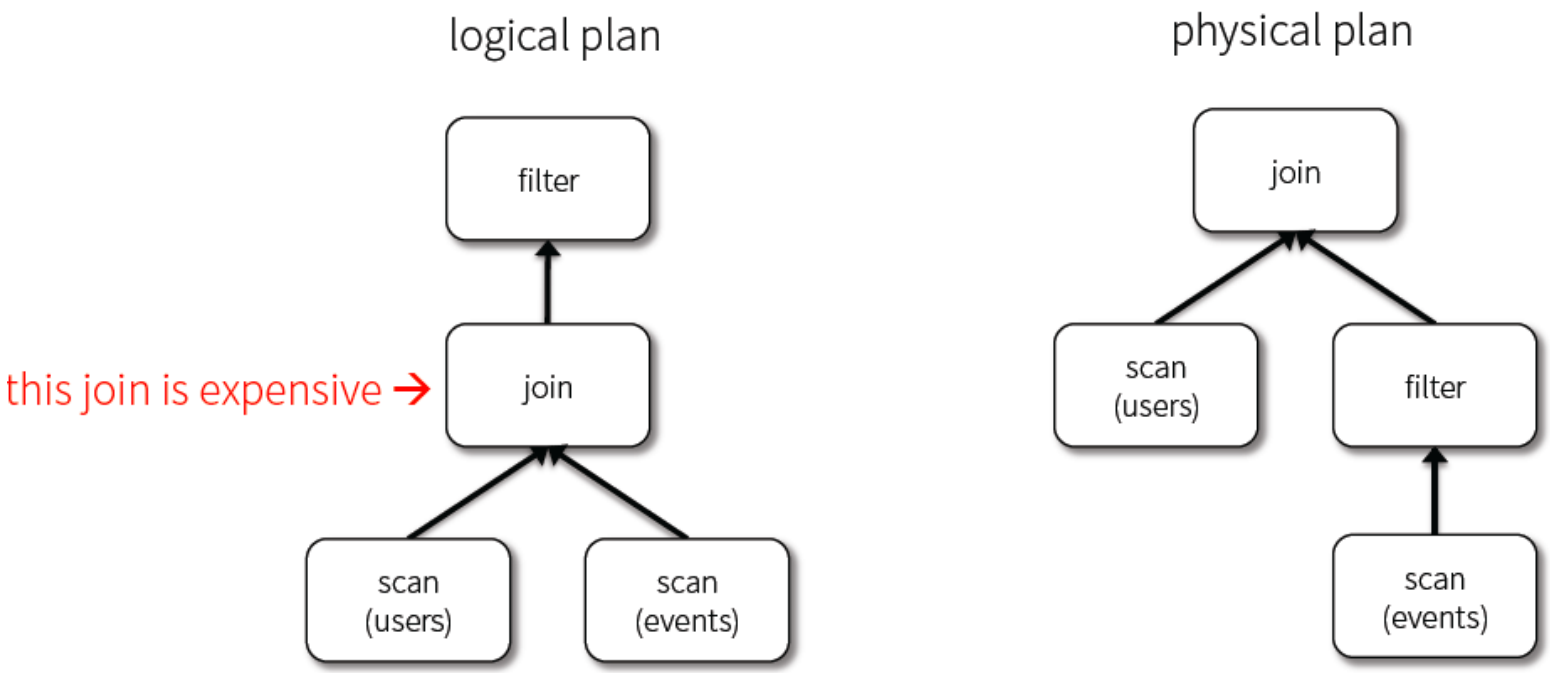

DataFrames and SQL share the same optimization/execution pipeline

- Optimization happens as late as possible

=> Spark SQL can optimize *even across different* functions !

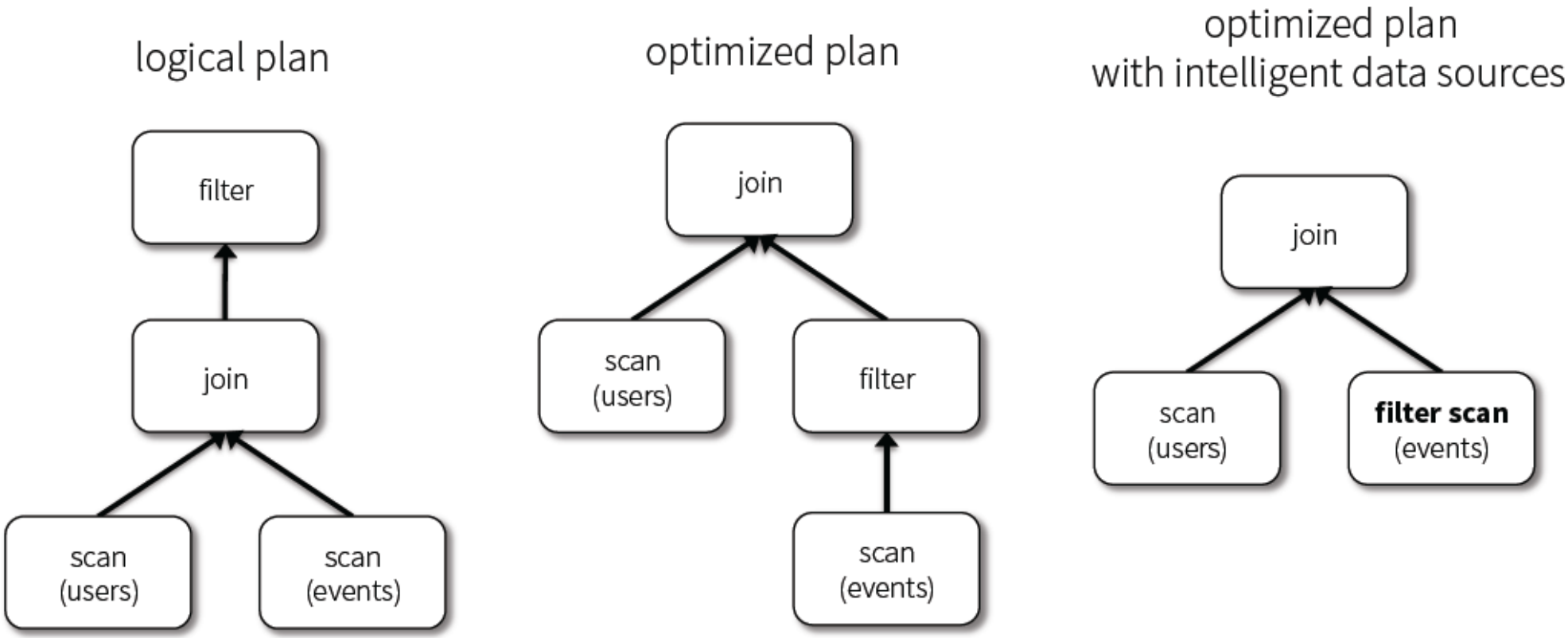
# Optimization Example

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



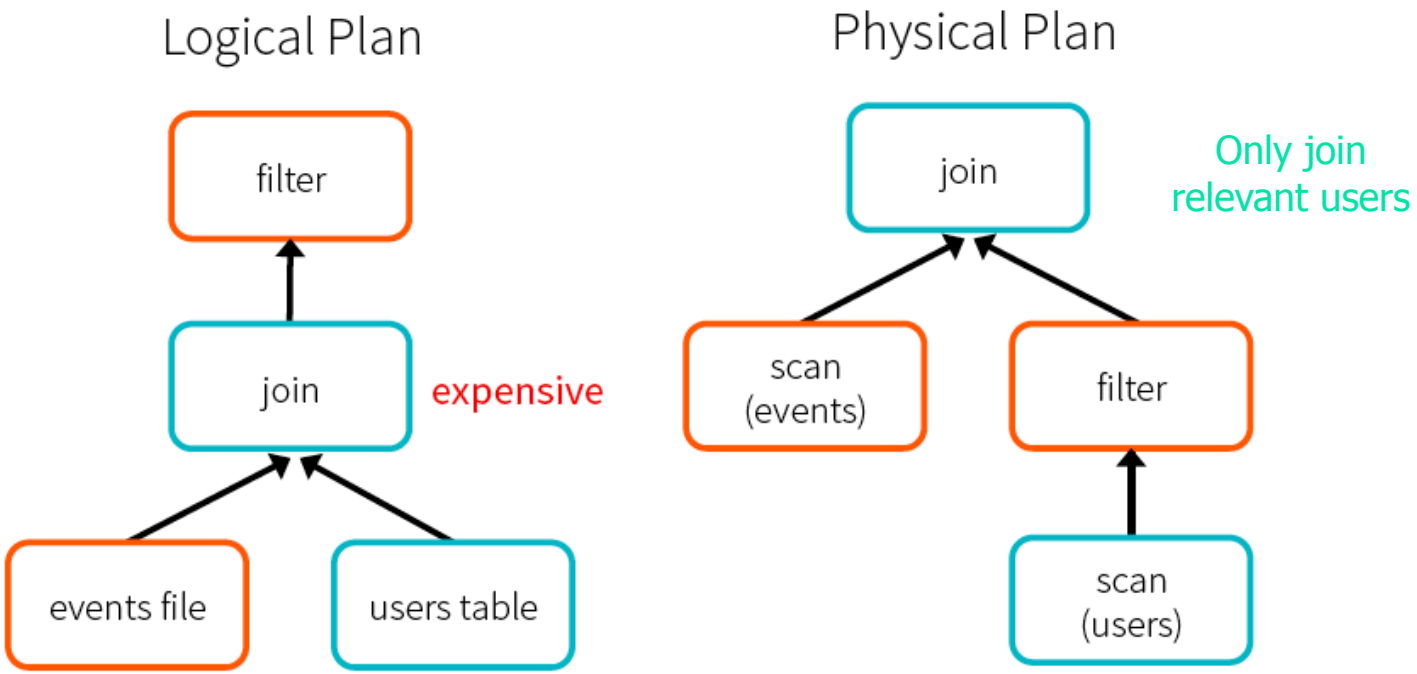
# Optimization Example

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date > "2015-01-01")
```



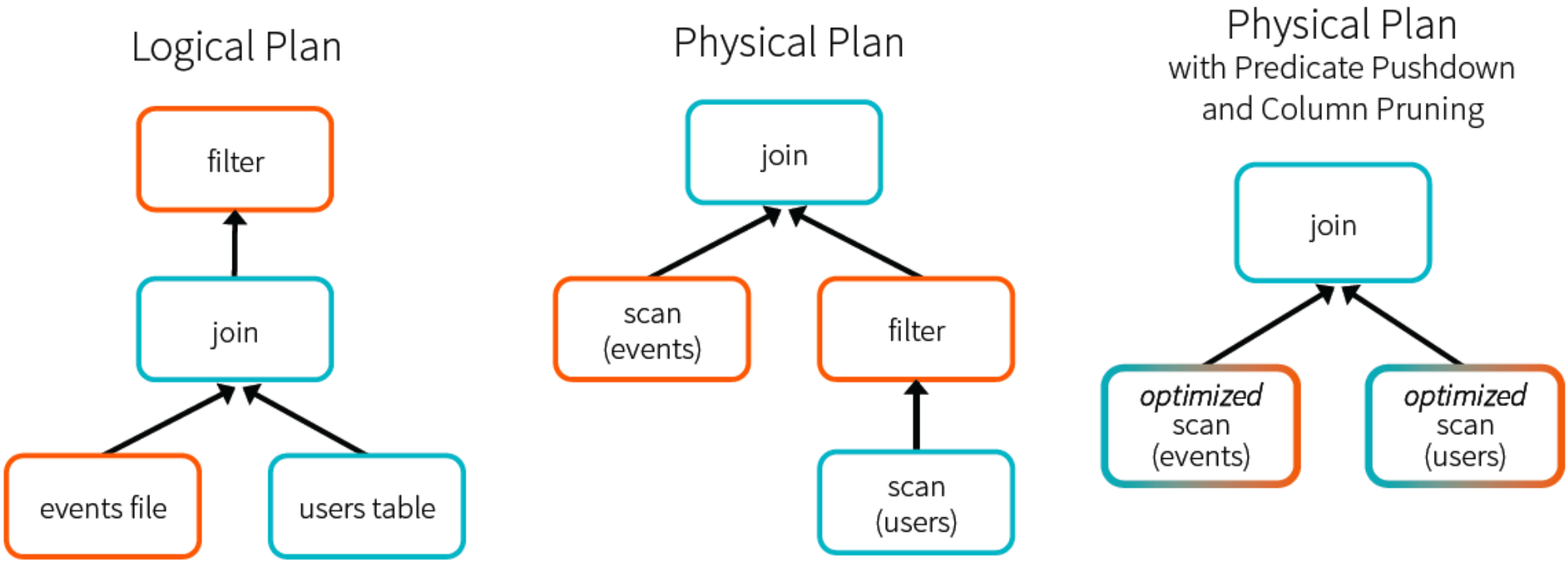
# Optimization Example

```
def add_demographics(events):
    u = sqlCtx.table("users")                          # Load Hive table
    events \
      .join(u, events.user_id == u.user_id) \          # Join on user_id
      .withColumn("city", zipToCity(df.zip))           # Run udf to add city column

events = add_demographics(sqlCtx.load("/data/events", "json"))
training_data = events.where(events.city == "New York").select(events.timestamp).collect()
```

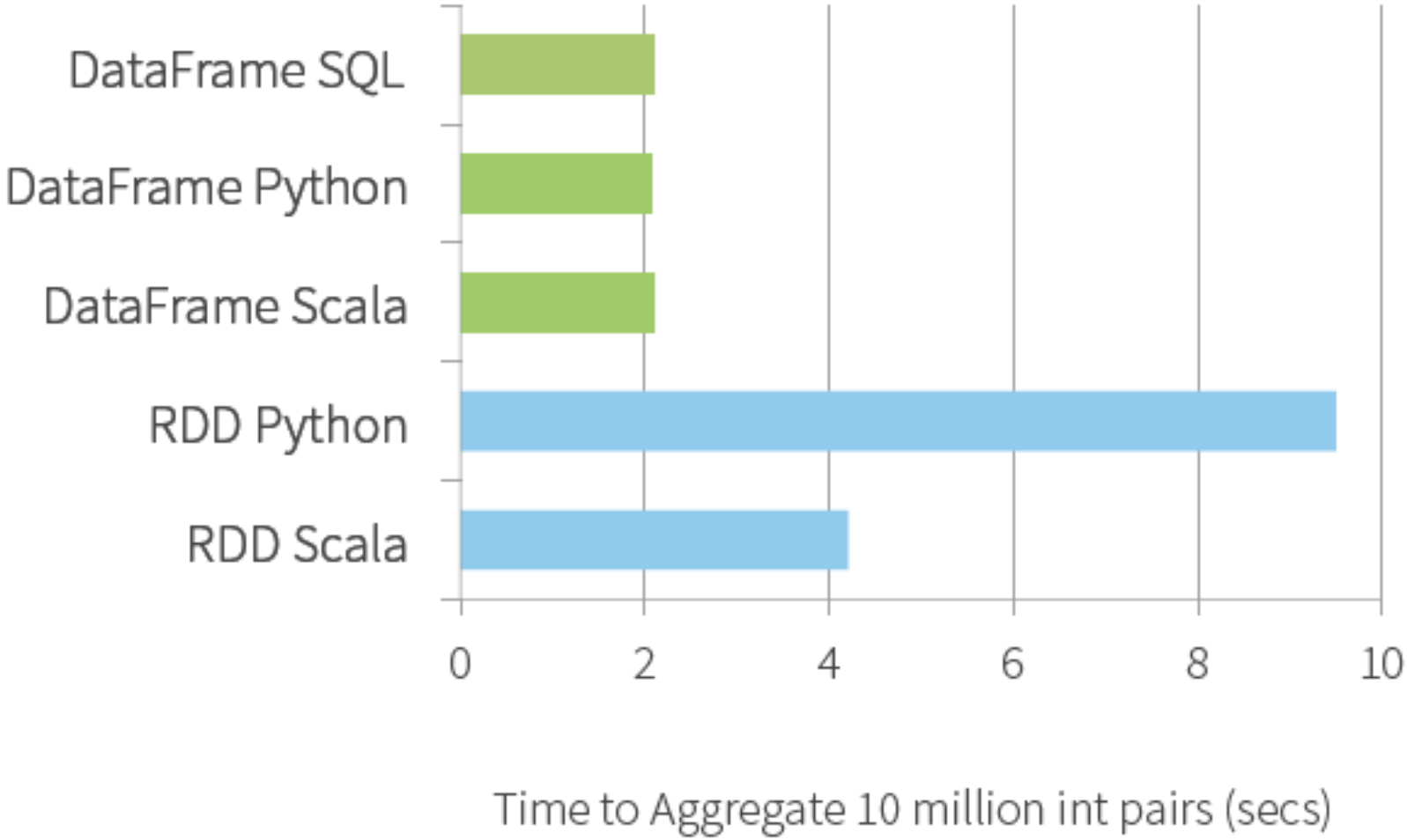


# Optimization Example

```
def add_demographics(events):
   u = sqlCtx.table("users")                          # Load partitioned Hive table
   events \
     .join(u, events.user_id == u.user_id) \          # Join on user_id
     .withColumn("city", zipToCity(df.zip))           # Run udf to add city column

events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "New York").select(events.timestamp).collect()
```



# Named Columns (vs. Opaque Objects in RDDs) Enable Performance Optimization



# More Performance Comparison

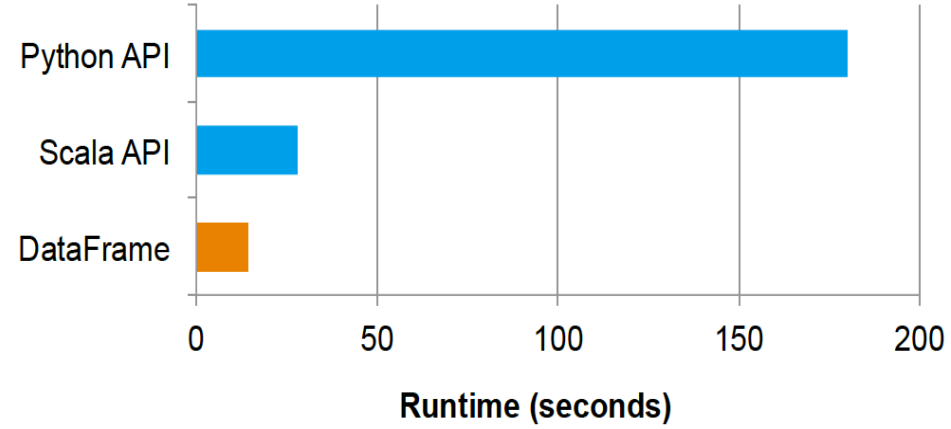



**Figure 9: Performance of an aggregation written using the native Spark Python and Scala APIs versus the DataFrame API.**

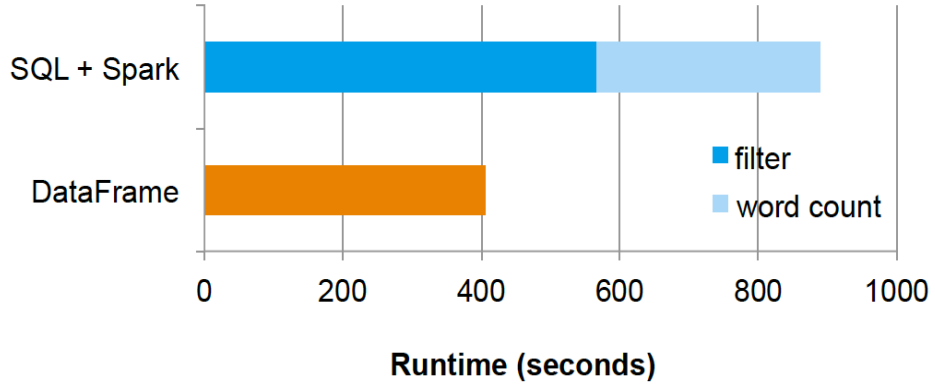
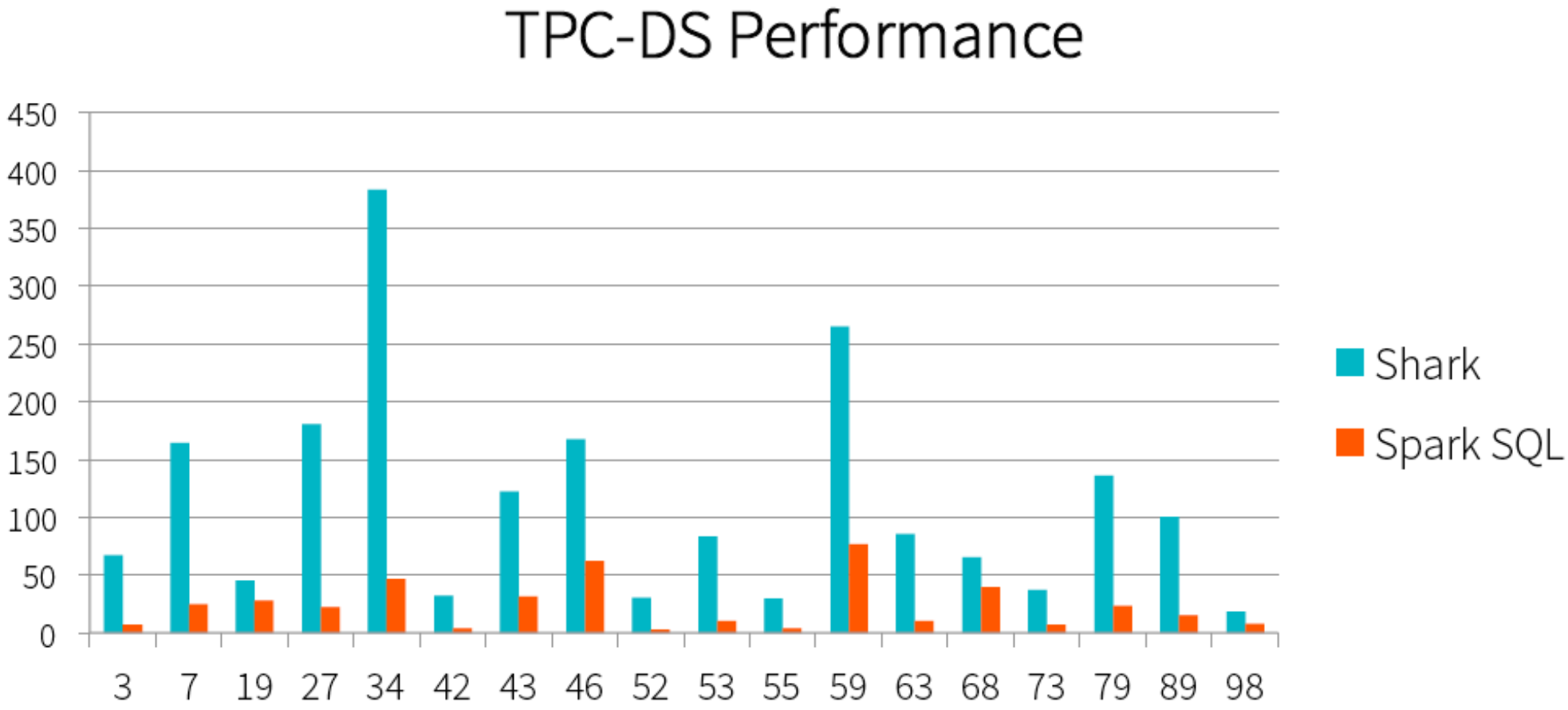



**Figure 10: Performance of a two-stage pipeline written as a separate Spark SQL query and Spark job (above) and an integrated DataFrame job (below).**

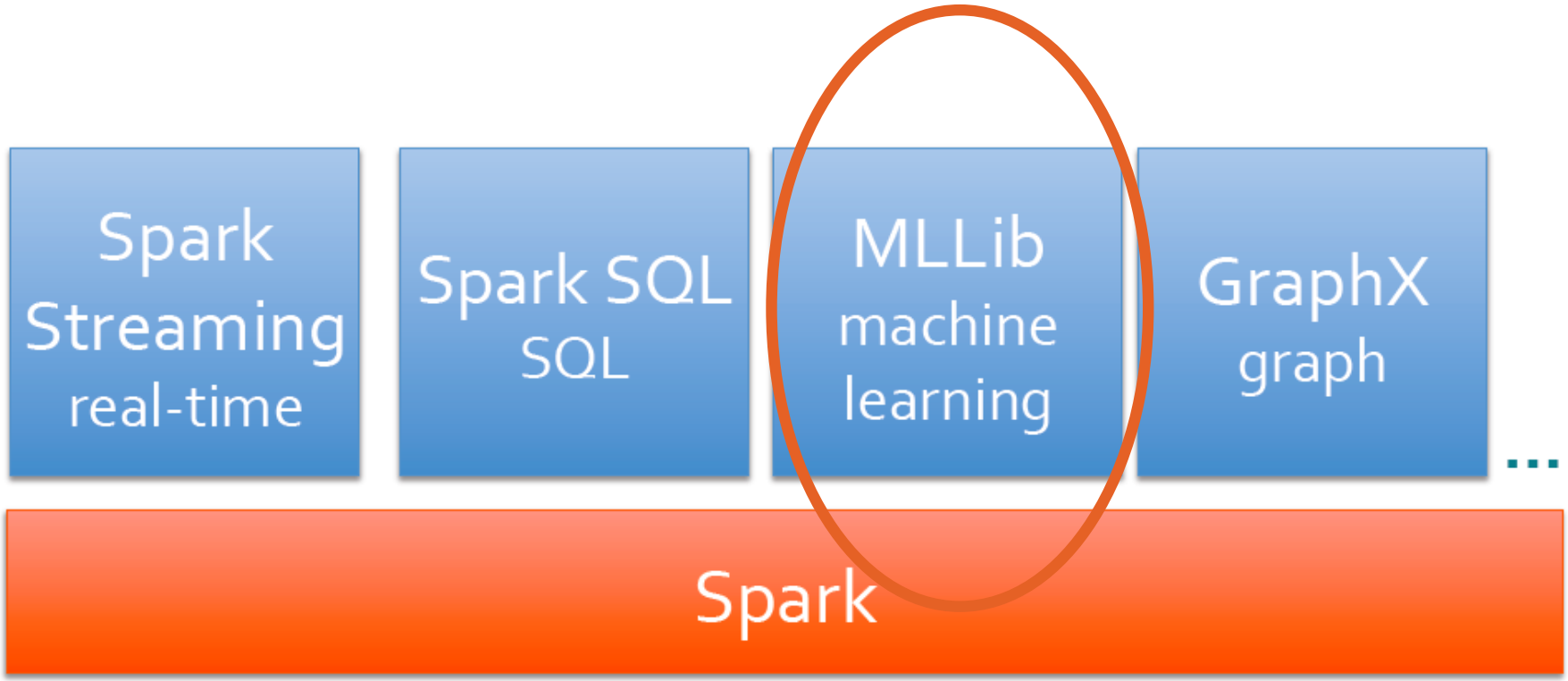
# Performance Comparison: Spark SQL vs. Shark



# Lessons Learnt from Spark SQL

- SQL is wildly popular and important for real-world customers
- Schema is very useful
  - In most data pipelines, even the ones that start with unstructured data end up having some implicit structure
  - Key-value abstraction (under RDD) is too limited
  - Nevertheless, Support for Semi/Un-structured data is critical !
- Separation of Logical vs. Physical Plan is important for Performance Optimizations, e.g. join selection.

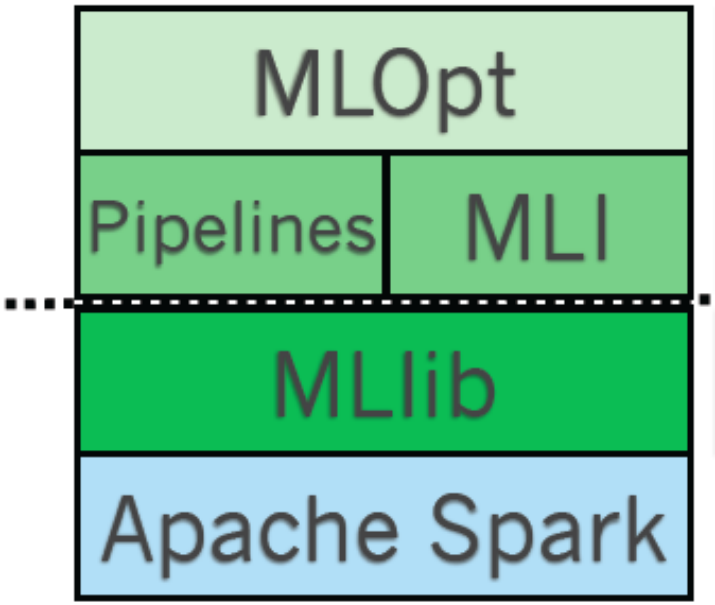
# About Spark for MLlib



# Machine Learning Support for Spark

- MLlib: Spark's core ML library

- The spark.ml package: High-level API built on the top of DataFrames for constructing Machine Learning Pipelines
  - Estimators, transformers, pipelines
- KeystoneML framework: Provide a richer set of operators than those in spark.ml
  - Featurizers for images, text, speech and example pipelines to reproduce state-of-the-art academic results in ML
  - Other related research projects:
    - MLBase: MLOpt, MLI (UCB AMPLabs)
    - Velox (UCB AMPLabs)
    - Parameter Server (CMU)



# Machine Learning Support for Spark

- DataFrames
  - Structured Data
  - Familiar API based on R & Python Pandas
  - Distributed, Optimized Implementation
- Machine Learning Pipelines
  - Simple construction and Tuning of Machine Learning workflows

# Machine Learning Support for Spark (via MLlib or spark.ml)

## Classification

- Logistic regression w/ elastic net
- Naive Bayes
- Streaming logistic regression
- Linear SVMs
- Decision trees
- Random forests
- Gradient-boosted trees
- Multilayer perceptron
- One-vs-rest

## Regression

- Least squares w/ elastic net
- Isotonic regression
- Decision trees
- Random forests
- Gradient-boosted trees
- Streaming linear methods

## Recommendation

- Alternating Least Squares

## Frequent itemsets

- FP-growth
- Prefix span

## Feature extraction & selection

- Binarizer
- Bucketizer
- Chi-Squared selection
- CountVectorizer
- Discrete cosine transform
- ElementwiseProduct
- Hashing term frequency
- Inverse document frequency
- MinMaxScaler
- Ngram
- Normalizer
- One-Hot Encoder
- PCA
- PolynomialExpansion
- RFormula
- SQLTransformer
- Standard scaler
- StopWordsRemover
- StringIndexer
- Tokenizer
- StringIndexer
- VectorAssembler
- VectorIndexer
- VectorSlicer
- Word2Vec

## Clustering

- Gaussian mixture models
- K-Means
- Streaming K-Means
- Latent Dirichlet Allocation
- Power Iteration Clustering

## Statistics

- Pearson correlation
- Spearman correlation
- Online summarization
- Chi-squared test
- Kernel density estimation

## Linear algebra

- Local dense & sparse vectors & matrices
- Distributed matrices
  - Block-partitioned matrix
  - Row matrix
  - Indexed row matrix
  - Coordinate matrix
- Matrix decompositions

## Model import/export

## Pipelines

*List based on Spark 1.5*

# Machine Learning Workflows are Complex

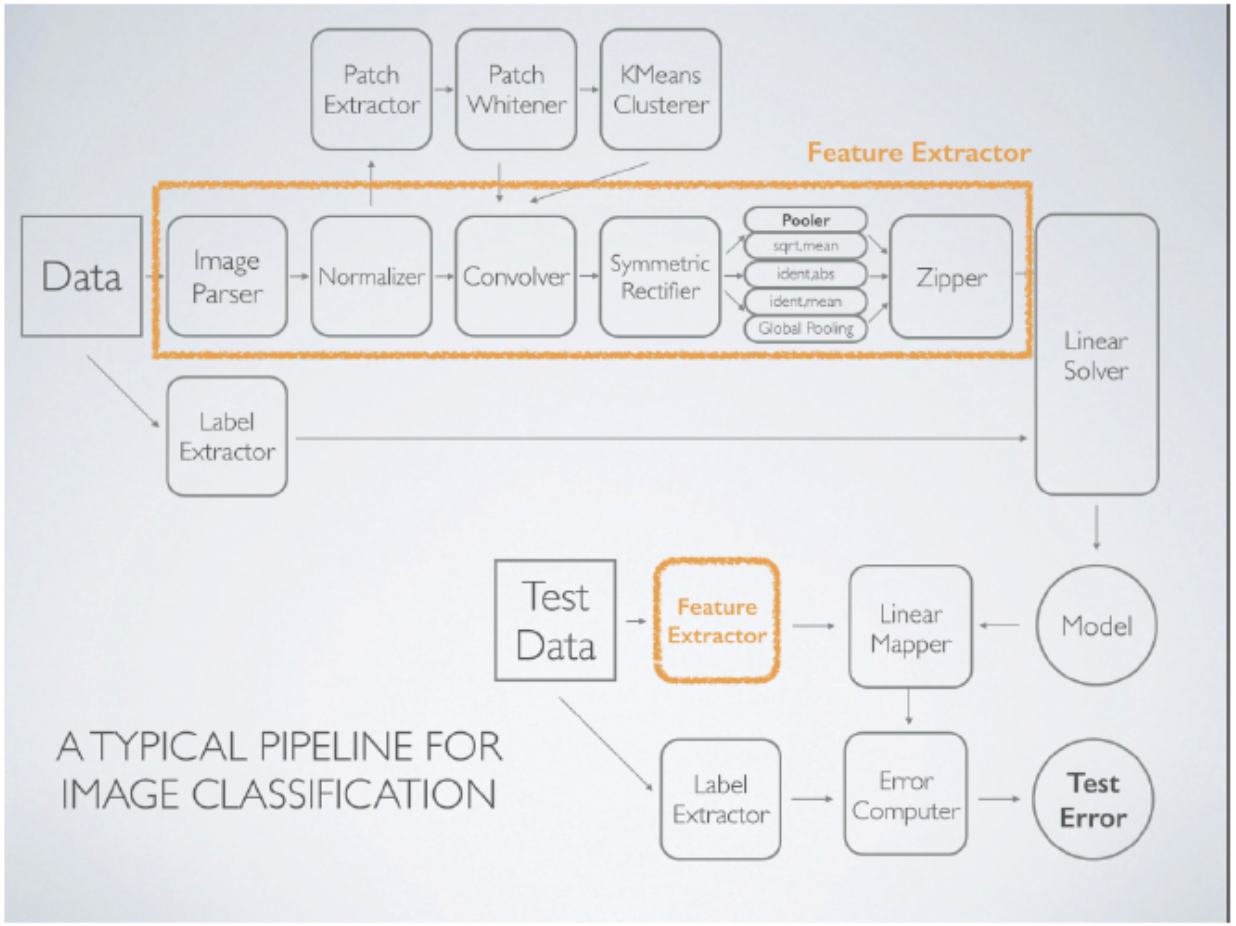



Image classification pipeline*

→ Specify pipeline
→ Inspect & debug
→ Re-run on new data
→ Tune parameters

** Evan Sparks. "ML Pipelines."*
amplab.cs.berkeley.edu/ml-pipelines

# Example: Text Classification

Goal: Given a text document, predict its topic.

**Features**

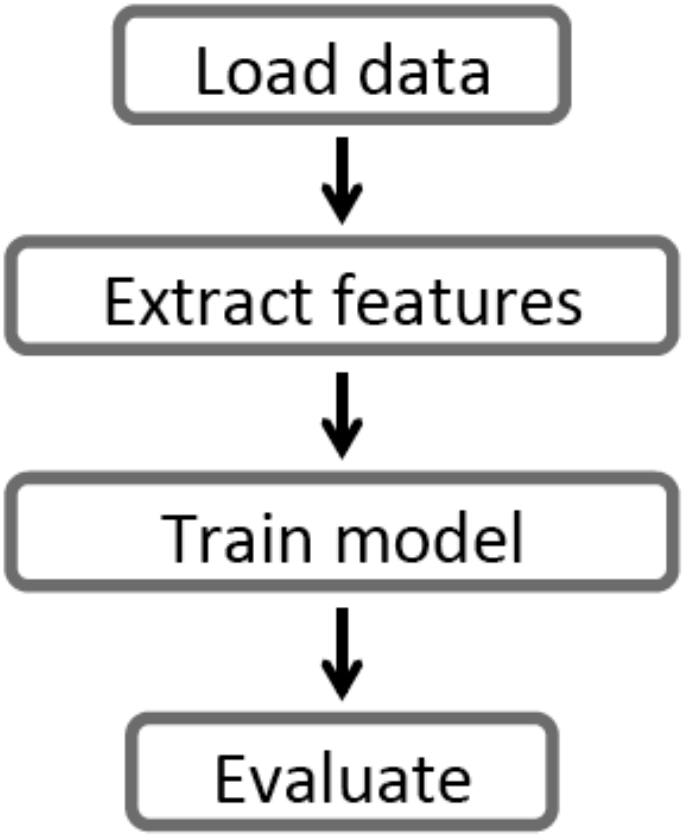
```
Subject: Re: Lexan Polish?
Suggest McQuires #1 plastic
polish.  It will help somewhat
but nothing will remove deep
scratches without making it
worse than it already is.
McQuires will do something...
```

**Label**

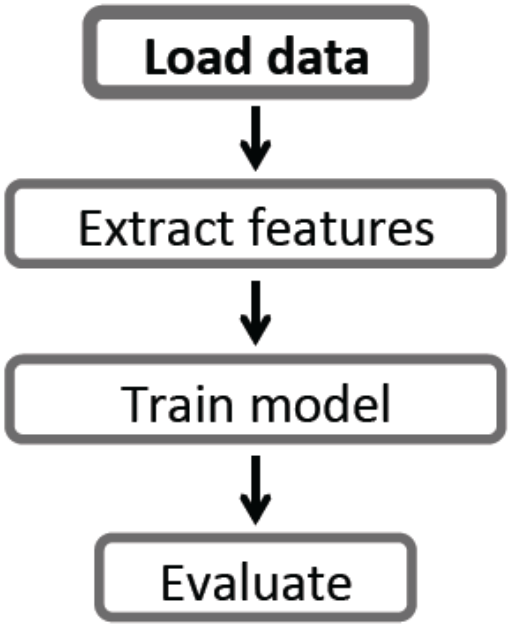
1: about science
0: not about science

Dataset: "20 Newsgroups"
*From UCI KDD Archive*

# Machine Learning Workflow



# Load Data




## Data sources for DataFrames

### built-in

Parquet

JDBC

{ JSON }

PostgreSQL

HIVE

MySQL

HDFS

amazon web services S3

H2

### external

AVRO

CSV

dBase

APACHE HBASE

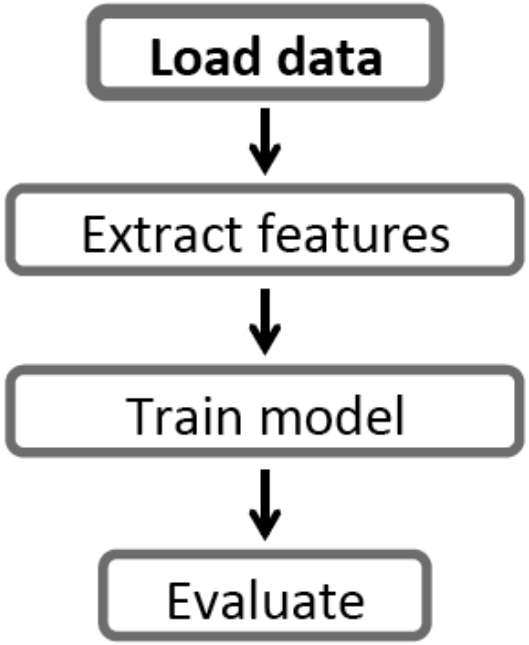
elasticsearch.

cassandra

amazon web services Amazon Redshift

and more ...

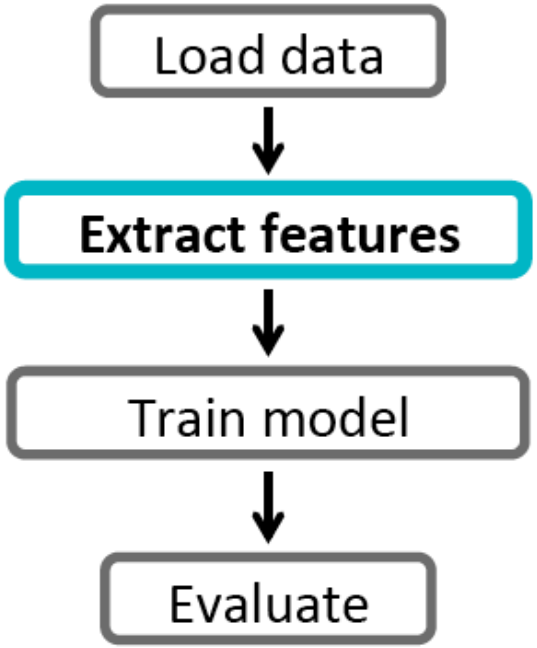
# Load Data




Current data schema

```
label: Int
text: String
```

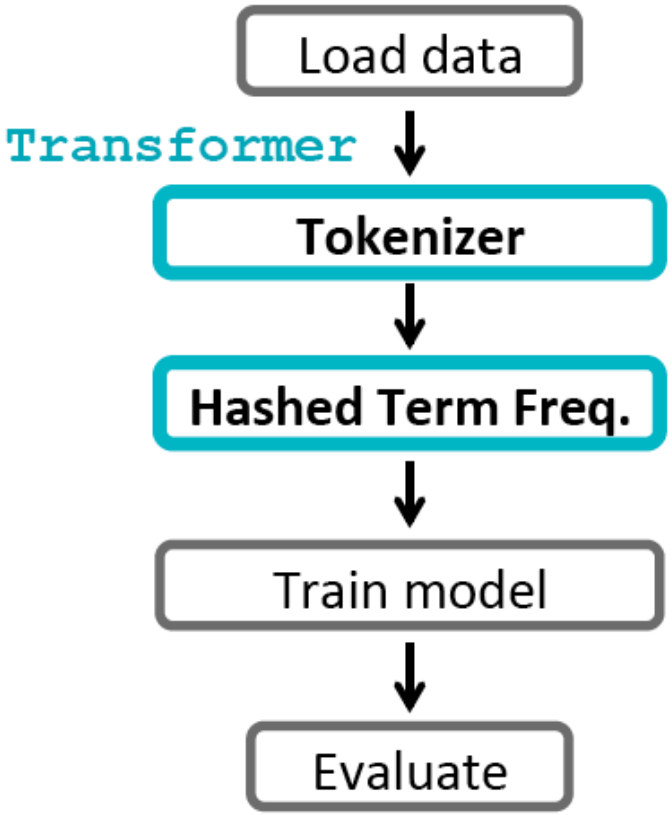
# Extract Features




Current data schema

```
label: Int
text: String
```

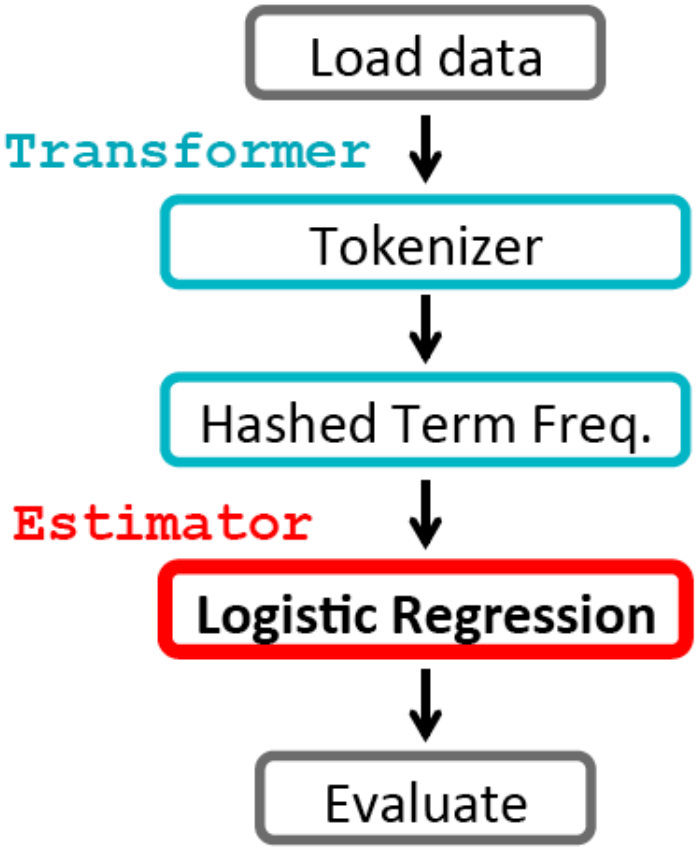
# Extract Features




Current data schema

```
label: Int
text: String

words: Seq[String]

features: Vector
```

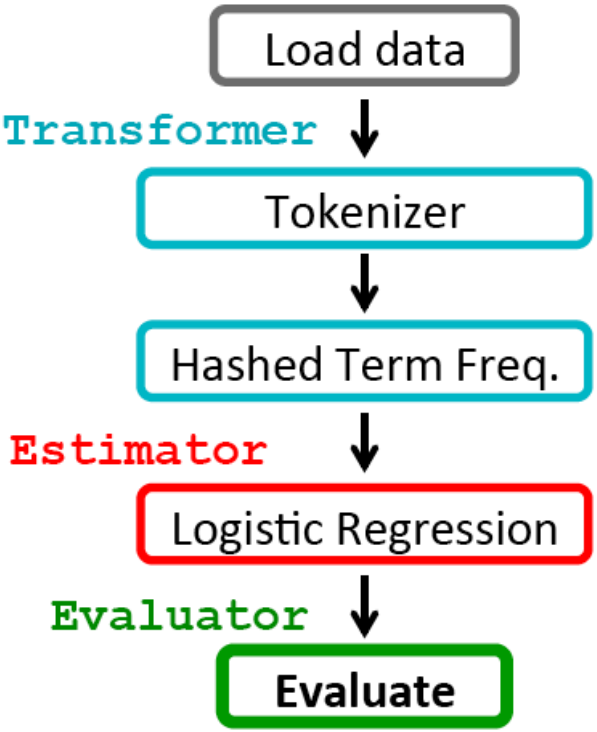
# Train a Model




Current data schema

```
label: Int
text: String

words: Seq[String]

features: Vector

prediction: Int
```

# Evaluate the Model




Current data schema

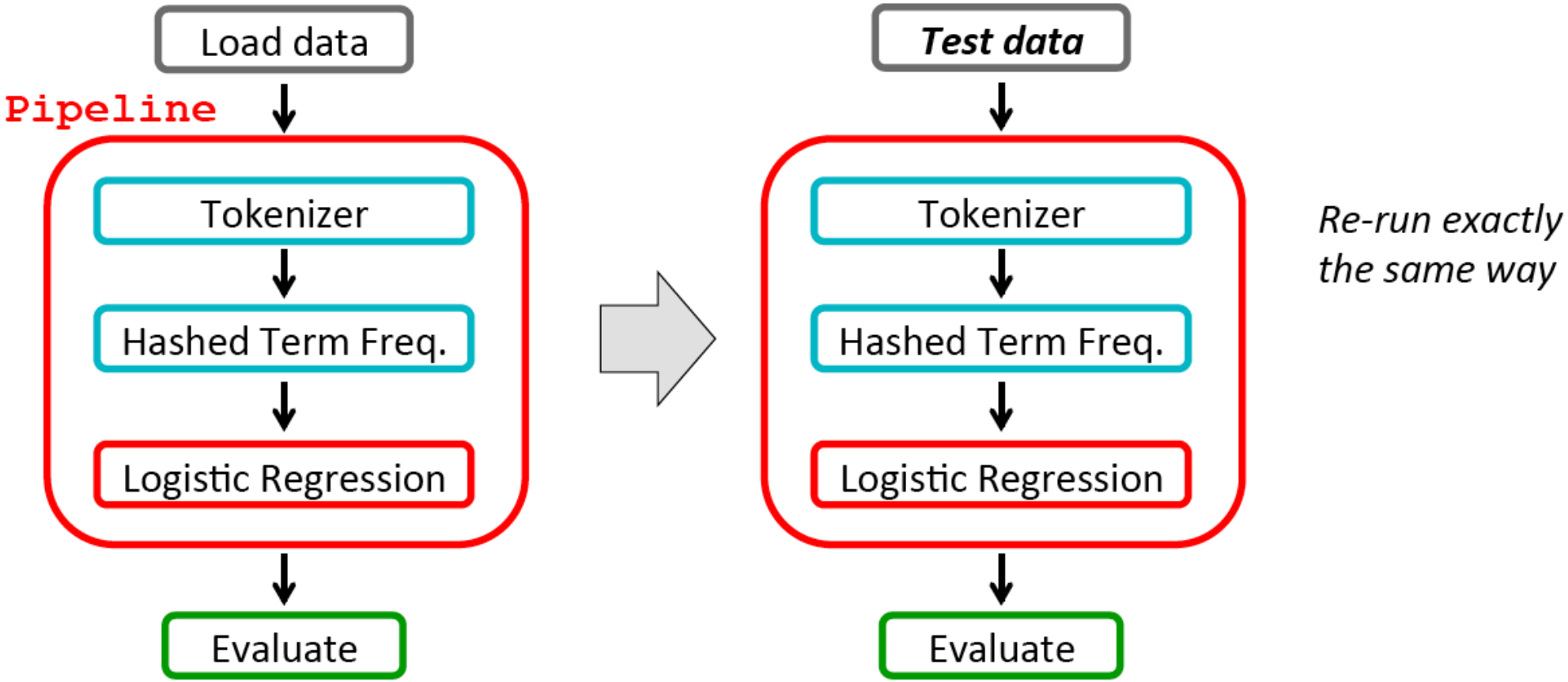
```
label: Int
text: String

words: Seq[String]

features: Vector

prediction: Int
```

By default, always append new columns

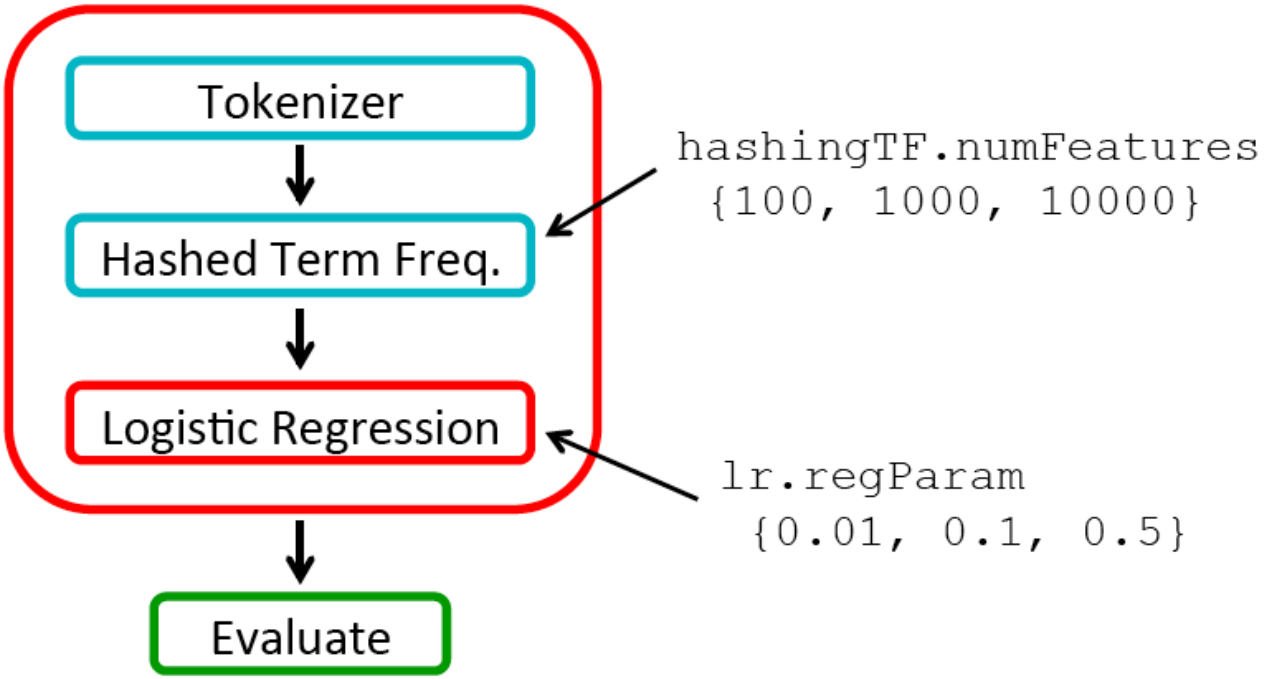
→ Can go back & inspect intermediate results

→ Made efficient by DataFrame optimizations

# Machine Learning Pipelines



# Parameter Tuning




**CrossValidator**

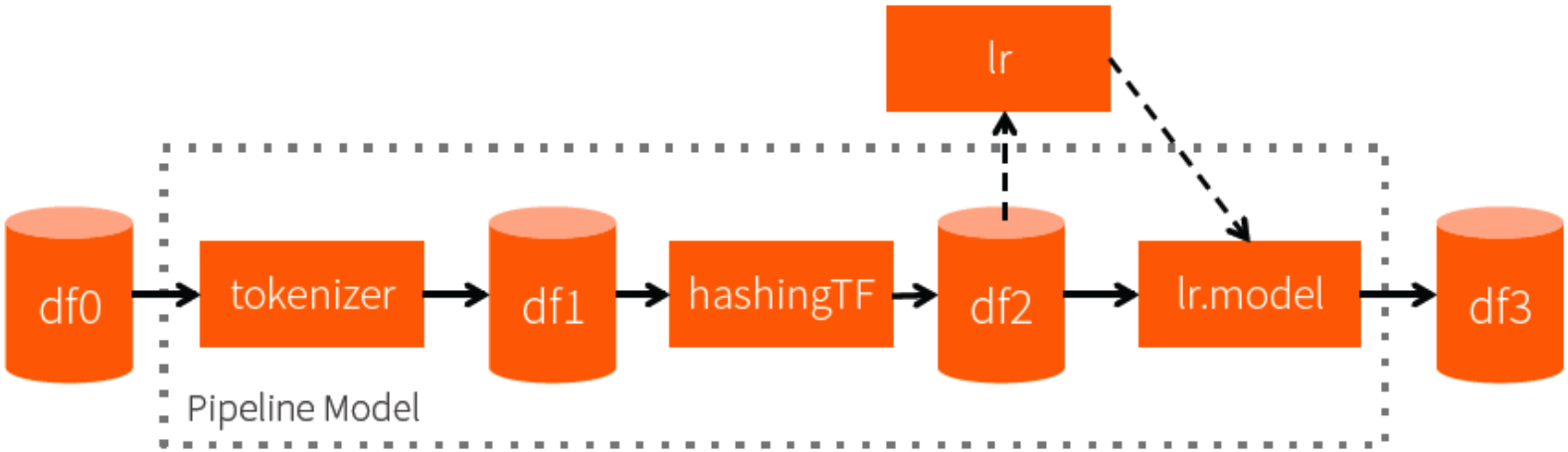
Given:

- Estimator
- Parameter grid
- Evaluator

Find best parameters

# DataFrame for Machine Learning Pipelines

```
tokenizer = Tokenizer(inputCol="text", outputCol="words”)
hashingTF = HashingTF(inputCol="words", outputCol="features”)
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
```



# SQL and Machine Learning

```
training_data_table = sql("""
  SELECT e.action, u.age, u.latitude, u.logitude
    FROM Users  u
    JOIN Events e ON u.userId = e.userId""")

def featurize(u):
    LabeledPoint(u.action, [u.age, u.latitude, u.longitude])


// SQL results are RDDs so can be used directly in Mllib.
training_data = training_data_table.map(featurize)

model = new LogisticRegressionWithSGD.train(training_data)
```

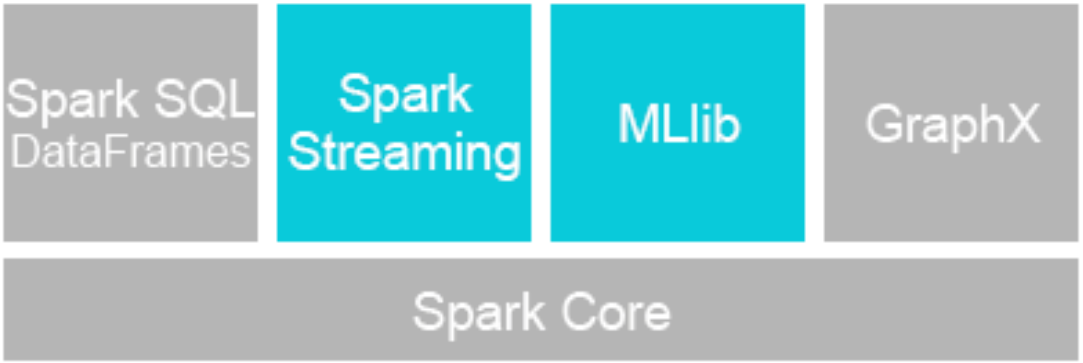
# Machine Learning Pipelines

```
// training:{eventId:String, features:Vector, label:Int}
val training = parquetFile("/path/to/training")
val lr = new LogisticRegression().fit(training)

// event: {eventId: String, features: Vector}
val event = parquetFile("/path/to/event")
val prediction =
  lr.transform(event).select('eventId, 'prediction)

prediction.saveAsParquetFile("/path/to/prediction”)
```

# Combine Machine Learning with Streaming

- Learn models offline, apply them online

```
// Learn model offline
val model = KMeans.train(dataset, ...)

// Apply model online on stream
kafkaStream.map { event =>
    model.predict(event.feature)
}
```



# Streaming MLlib Algorithms

## Continuous learning and prediction on streaming data

StreamingLinearRegression, StreamingKMeans,
StreamingLogisticRegression

```
val model = new StreamingKMeans()
              .setK(10).setDecayFactor(1.0).setRandomCenters(4, 0.0)

model.trainOn(trainingDStream) // Train on one DStream

// Predict on another DStream
model.predictOnValues( testDStream.map { lp => (lp.label, lp.features) } )
```

https://databricks.com/blog/2015/01/28/introducing-streaming-k-means-in-spark-1-2.html

```
lines = KafKaUtils.createStream(
     streamingContext, kafkaTopics, kafkaParams)

counts = lines.flatMap(lambda line: line.split(" "))
```

# Summary of DataFrames and ML Workflow

DataFrames

- Structured Data
- Familiar API based on R & Python Pandas
- Distributed, Optimized Implementation

Schema validation

User-defined Transformers & Estimators

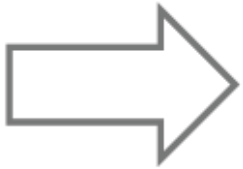
Machine Learning Pipelines

- Integration with DataFrames
- Familiar API based on scikit-learn
- Simple Parameter Tuning
  - Ongoing Research on Auto-Tuning Models

Composable & DAG Pipelines

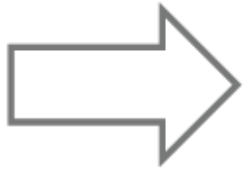
# Enabling Interactive (Big) Data Science with SparkR

Spark early adopters




Users

Understands
MapReduce
& functional APIs



Data Engineers
Data Scientists
Statisticians
R users
PyData …

Fast! Statistical!

Scalable  +  Packages

Flexible Interactive

# SparkR – R package for Spark

- R Interface support via SparkR (R with RDD = R2D2) since Spark 1.4 (released since June 2015)
  - Exposes DataFrames and MLlib in R:

```
df = jsonFile("tweets.json")

summarize(
  group_by(
    df[df$user == "matei",],
    "date"),
  sum("retweets"))
```

# SparkR – R package for Spark

**SparkR**

RDD → distributed lists

Run R on clusters

Re-use existing packages

**Combine scalability & utility**

# Getting closer to Idiomatic R

*Q: How can I use a loop to [...insert task here...] ?*

*A: Don't. Use one of the apply functions.*

From: http://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/

# SparkR

R + RDD = RRDD

lapply
lapplyPartition
groupByKey
reduceByKey
sampleRDD
collect
cache
…

broadcast
includePackage
textFile
parallelize

# Example: Word Counting with SparkR

```
lines <- textFile(sc, "hdfs://my_text_file")

words <- flatMap(lines,
                 function(line) {
                   strsplit(line, " ")[[1]]
                 })
wordCount <- lapply(words,
                    function(word) {
                        list(word, 1L)
                    })
counts <- reduceByKey(wordCount, "+", 2L)
output <- collect(counts)
```

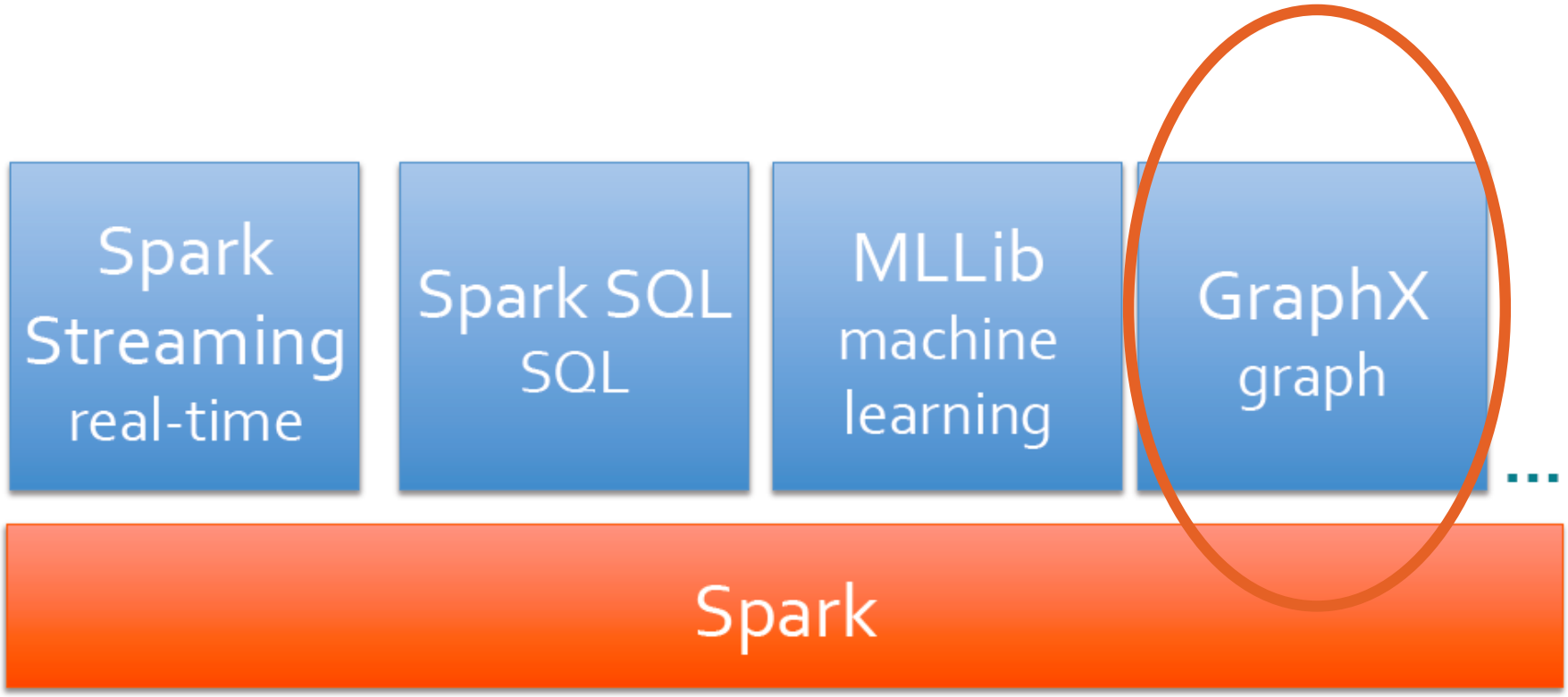
# Example: Logistic Regression with SparkR

```
pointsRDD <- textFile(sc, "hdfs://myfile")
weights <- runif(n=D, min = -1, max = 1)

# Logistic gradient
gradient <- function(partition) {
  X <- partition[,1]; Y <- partition[,-1]
  t(X) %*% (1/(1 + exp(-Y * (X %*% weights))) - 1) * Y
}

# Iterate
weights <- weights - reduce(
      lapplyPartition(pointsRDD, gradient), "+")
```

# SparkR Implementation

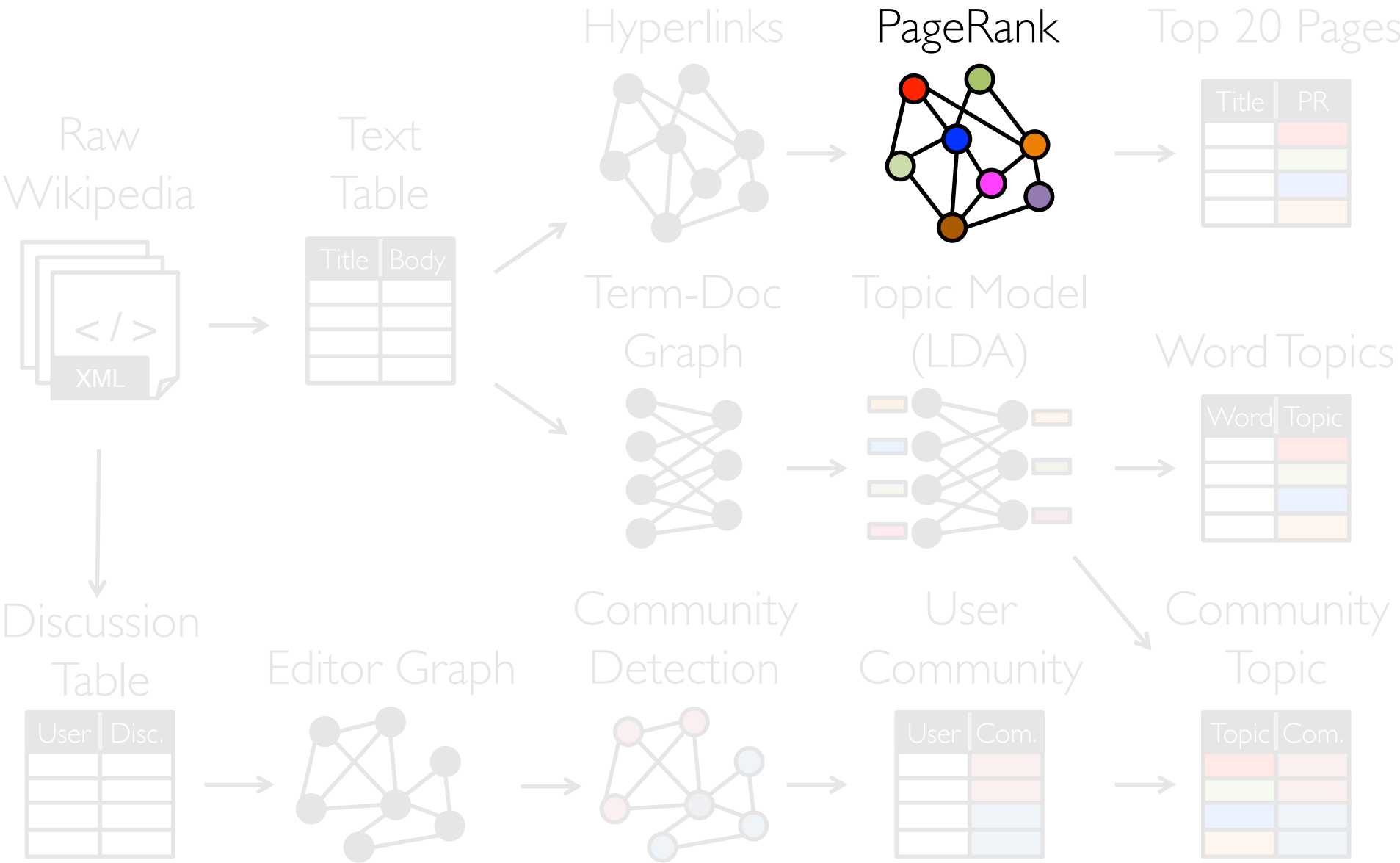
- Very similar to PySpark
- Relatively easy to extend Spark
  - 329 lines of Scala code
  - 2079 lines of R code
  - 693 lines of Test code in R

# Generality of RDDs in Spark



# GraphX:
# *Unifying Data-Parallel and Graph-Parallel Analytics*

# Graphs are Central to Analytics



# PageRank: Identifying Leaders

$$R[i] = 0.15 + \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j]$$

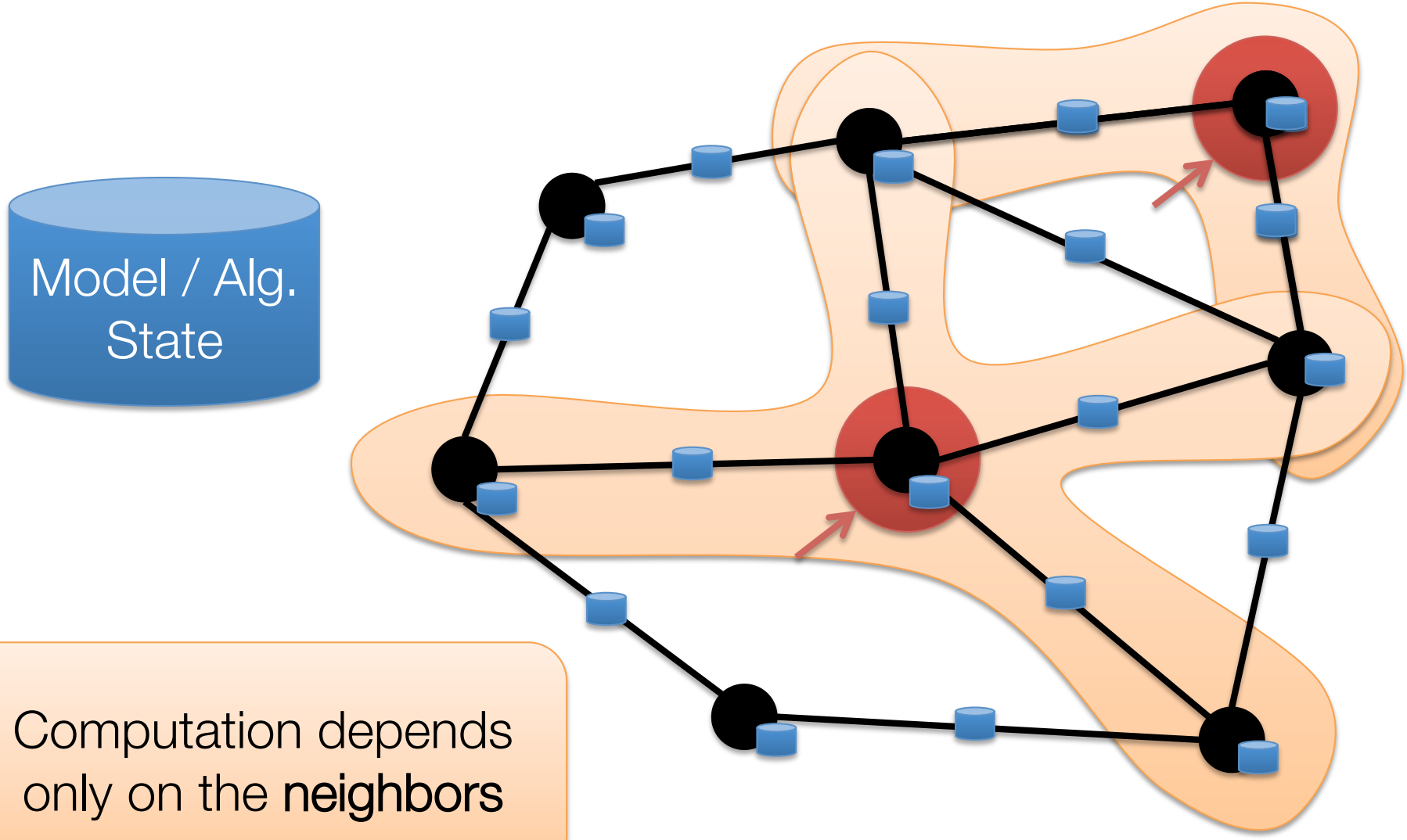
Rank of user *i*

Weighted sum of neighbors' ranks

Update ranks in parallel

Iterate until convergence

# The Graph-Parallel Pattern



# Many Graph-Parallel Algorithms

- Collaborative Filtering
  - Alternating Least Squares
  - Stochastic Gradient Descent
  - Tensor Factorization
- Structured Prediction
  - Loopy Belief Propagation
  - Max-Product Linear Programs
  - Gibbs Sampling
- Semi-supervised ML
  - Graph SSL
  - CoEM
- Community Detection
  - Triangle-Counting
  - K-core Decomposition
  - K-Truss
- Graph Analytics
  - PageRank
  - Personalized PageRank
  - Shortest Path
  - Graph Coloring
- Classification
  - Neural Networks

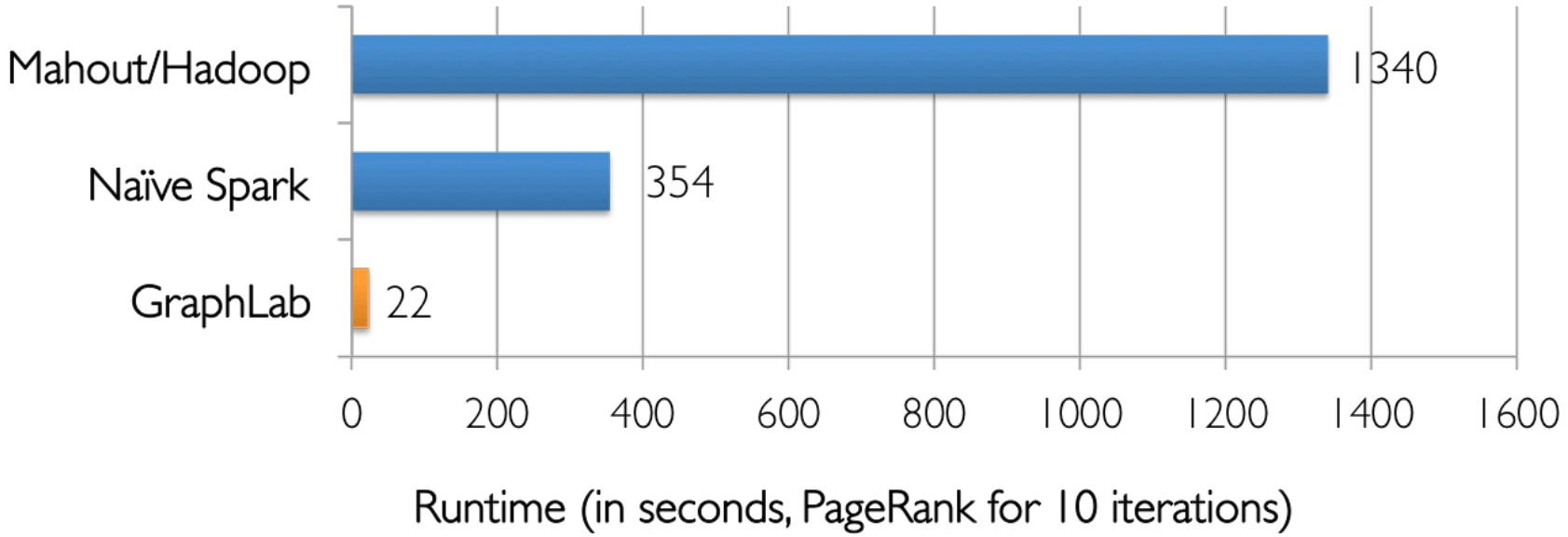
# Graph-Parallel Systems




*Expose specialized APIs to simplify graph programming.*

*Exploit graph structure to achieve orders-of-magnitude performance gains over more general data-parallel systems.*

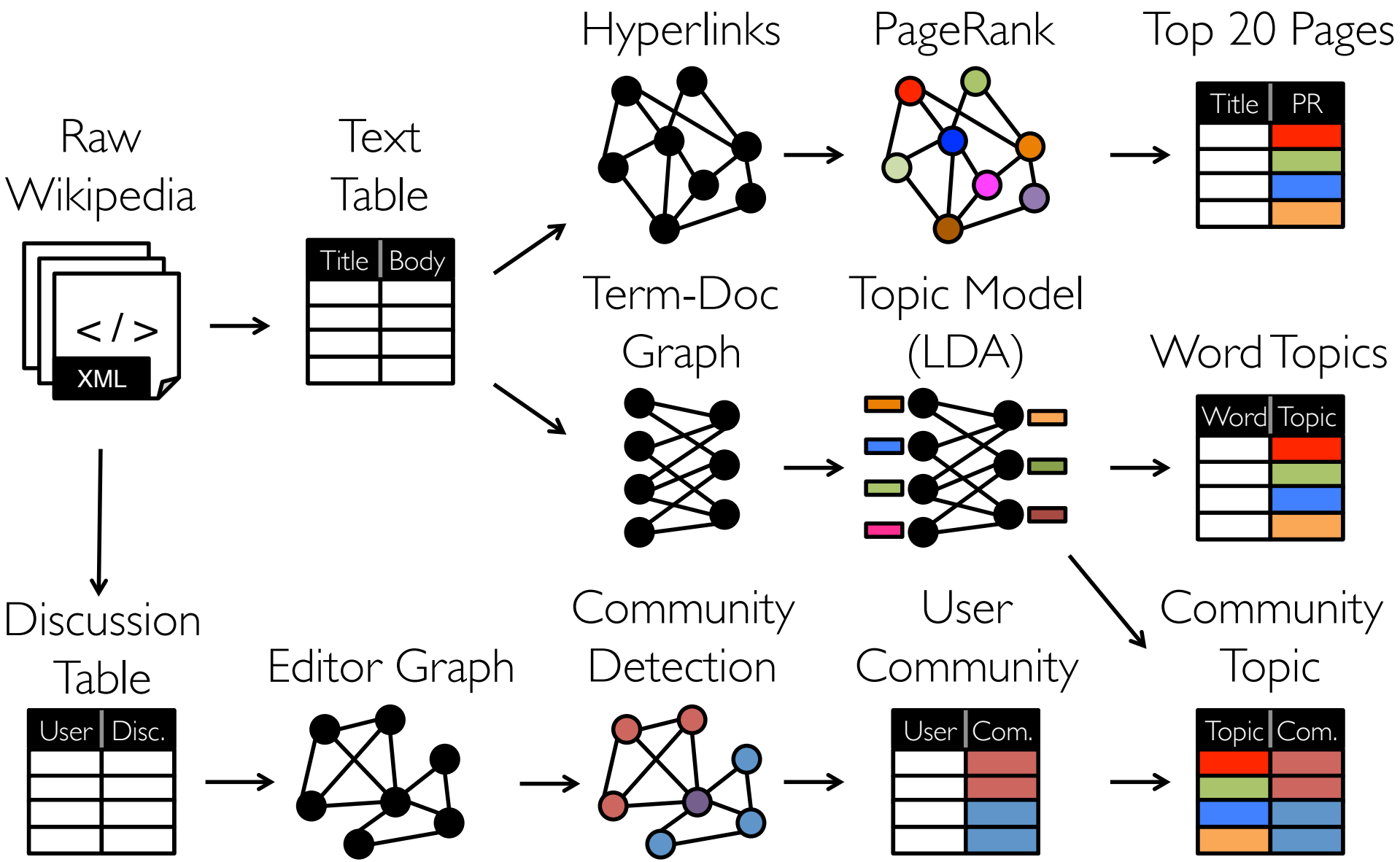
# PageRank on the Live-Journal Graph




GraphLab is *60x faster* than Hadoop

GraphLab is *16x faster* than Spark

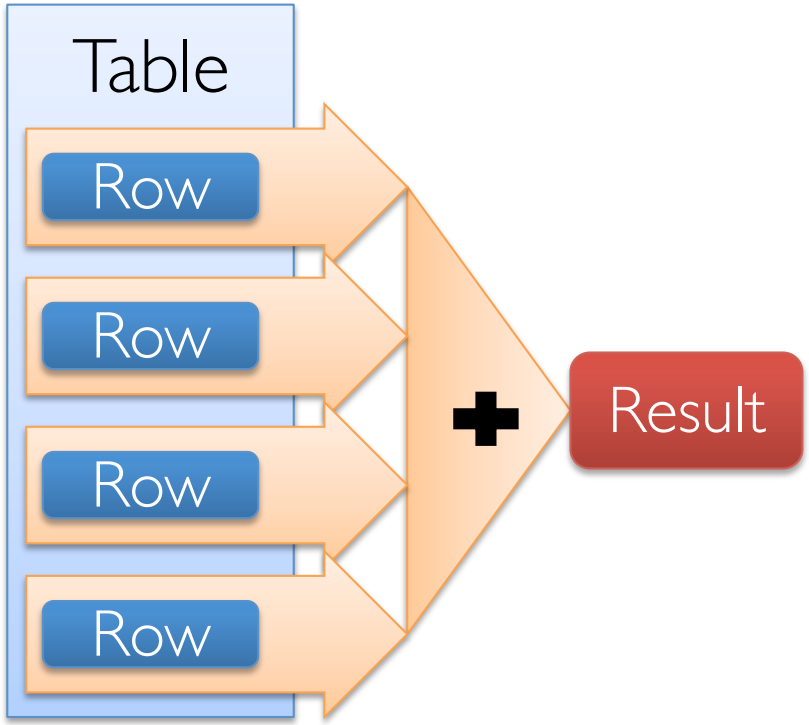
# Graphs are Central to Analytics



# Separate Systems to Support Each View

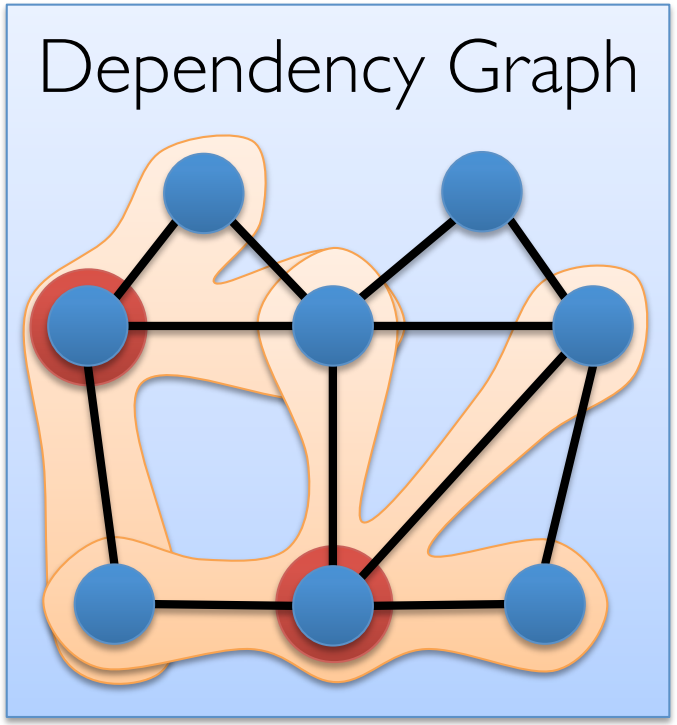
## Table View






## Graph View

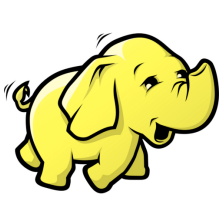




*Having separate systems for each view is difficult to use and inefficient*

# Difficult to Program and Use

Users must *Learn*, *Deploy*, and *Manage* multiple systems

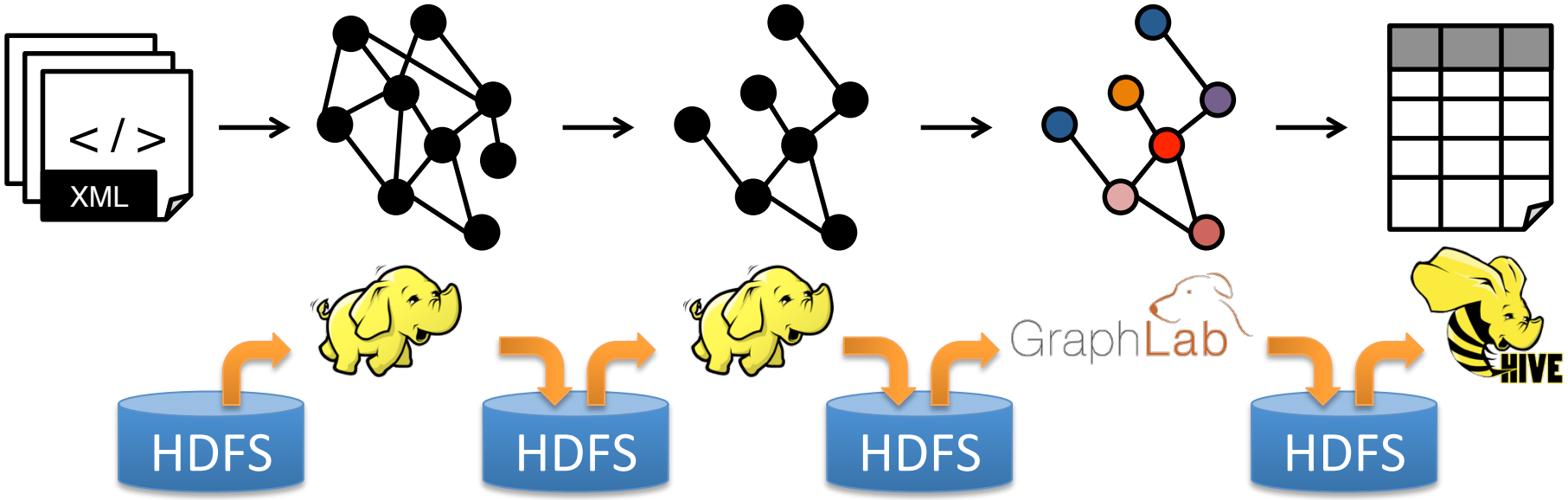




Leads to brittle and often complex interfaces

# Inefficient

Extensive data movement and duplication across the network and file system

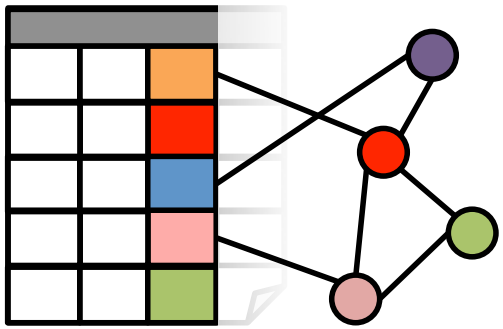



Limited reuse internal data-structures across stages

# *Solution*: The GraphX Unified Approach

## New API

*Blurs the distinction between Tables and Graphs*



## New System

*Combines Data-Parallel Graph-Parallel Systems*




Enabling users to easily and efficiently express the entire graph analytics pipeline

# Tables and Graphs are composable *views* of the *same physical* data

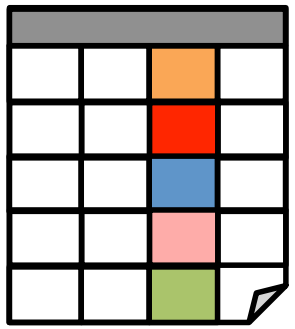
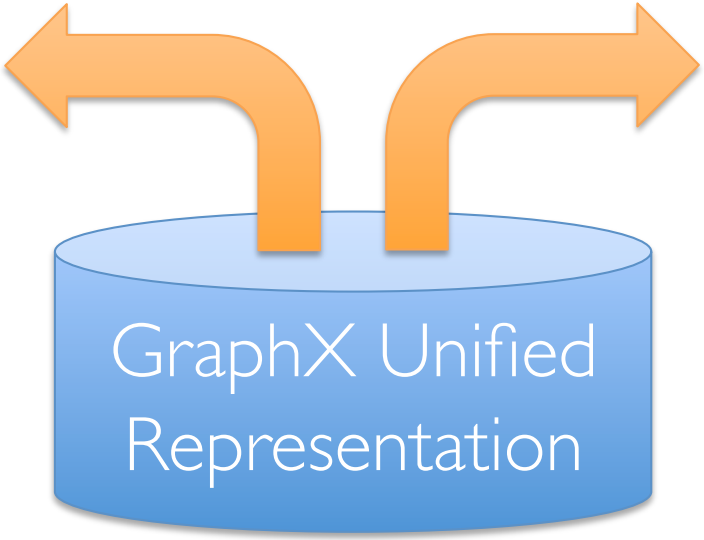
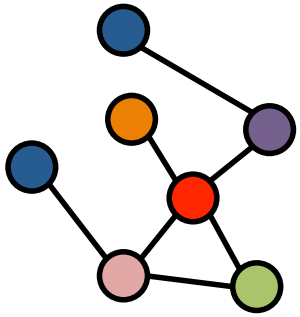


Table View



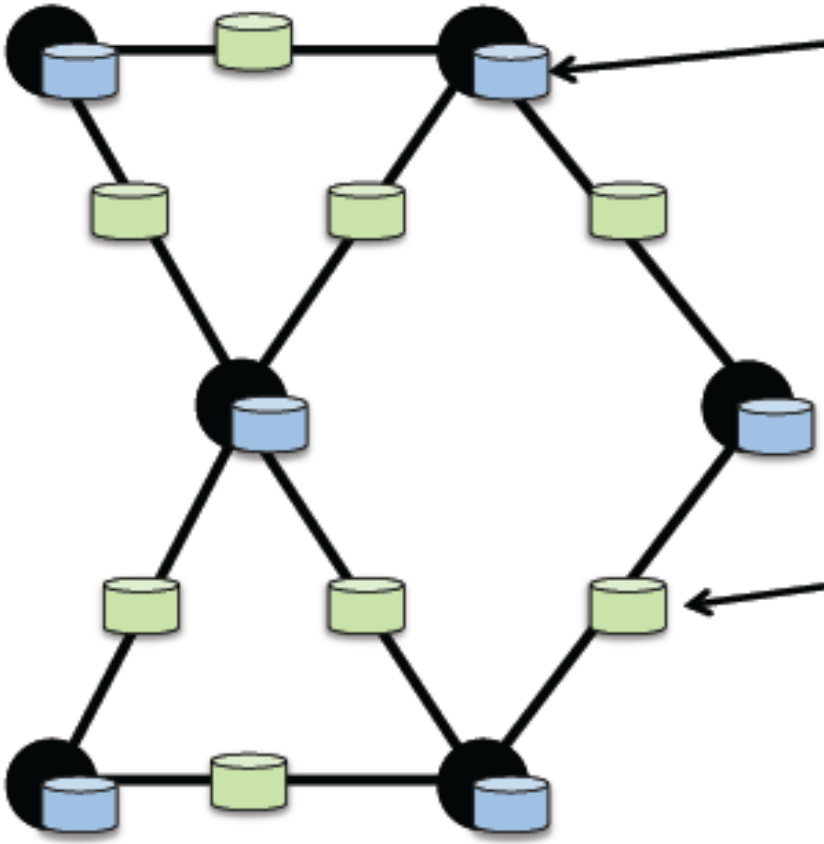



Graph View

Each view has its own operators that exploit the semantics of the view to achieve efficient execution

# The GraphX API

# Property Graphs



## Vertex Property:

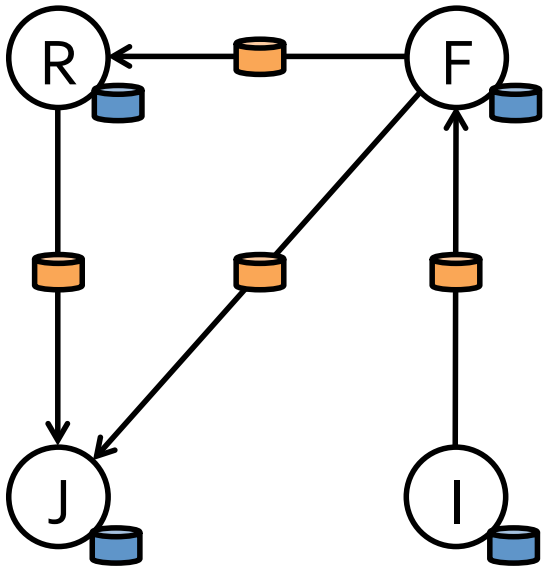
- User Profile
- Current PageRank Value

## Edge Property:

- Weights
- Relationships
- Timestamps

# View a Graph as a Table

## Property Graph




## Vertex Property Table

| Id | Property (V) |
|---|---|
| Rxin | (Stu., Berk.) |
| Jegonzal | (PstDoc, Berk.) |
| Franklin | (Prof., Berk) |
| Istoica | (Prof., Berk) |

## Edge Property Table

| SrcId | DstId | Property (E) |
|---|---|---|
| rxin | jegonzal | Friend |
| franklin | rxin | Advisor |
| istoica | franklin | Coworker |
| franklin | jegonzal | PI |

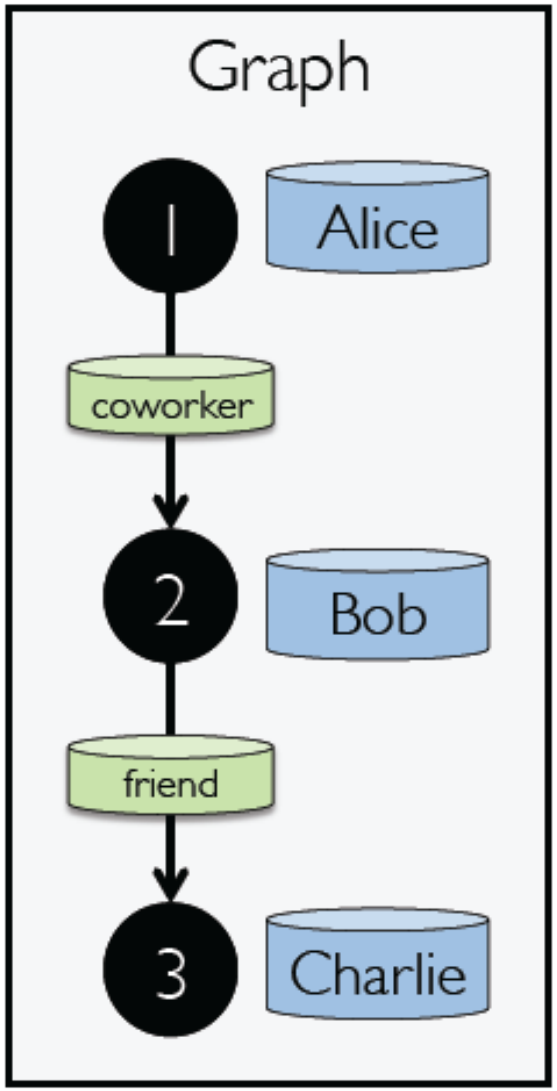
# Table Operators

Table (RDD) operators are inherited from Spark:

| | | |
|---|---|---|
| `map` | `reduce` | `sample` |
| `filter` | `count` | `take` |
| `groupBy` | `fold` | `first` |
| `sort` | `reduceByKey` | `partitionBy` |
| `union` | `groupByKey` | `mapWith` |
| `join` | `cogroup` | `pipe` |
| `leftOuterJoin` | `cross` | `save` |
| `rightOuterJoin` | `zip` | `...` |

# Creating a Graph (Scala)

```
type VertexId = Long

val vertices: RDD[(VertexId, String)] =
  sc.parallelize(List(
    (1L, “Alice”),
    (2L, “Bob”),
    (3L, “Charlie”)))

class Edge[ED](
  val srcId: VertexId,
  val dstId: VertexId,
  val attr: ED)

val edges: RDD[Edge[String]] =
  sc.parallelize(List(
    Edge(1L, 2L, “coworker”),
    Edge(2L, 3L, “friend”)))

val graph = Graph(vertices, edges)
```



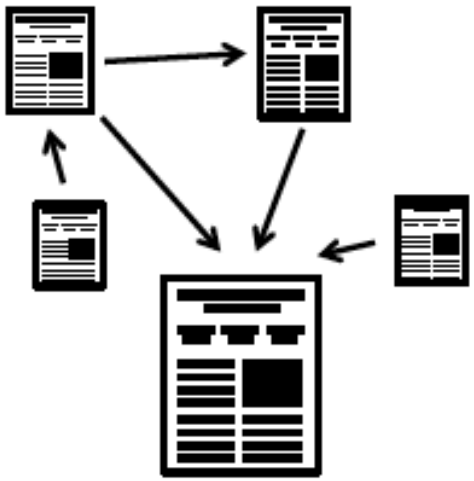
# Graph Operations (Scala)

```
class Graph[VD, ED] {
    // Table Views -----------------------
    def vertices: RDD[(VertexId, VD)]
    def edges: RDD[Edge[ED]]
    def triplets: RDD[EdgeTriplet[VD, ED]]
    // Transformations ------------------------------------
    def mapVertices[VD2](f: (VertexId, VD) => VD2): Graph[VD2, ED]
    def mapEdges[ED2](f: Edge[ED] => ED2): Graph[VD2, ED]
    def reverse: Graph[VD, ED]
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
    // Joins ----------------------------------------
    def outerJoinVertices[U, VD2]
        (tbl: RDD[(VertexId, U)])
        (f: (VertexId, VD, Option[U]) => VD2): Graph[VD2, ED]
    // Computation ----------------------------------
    def mapReduceTriplets[A](
        sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
        mergeMsg: (A, A) => A): RDD[(VertexId, A)]
```

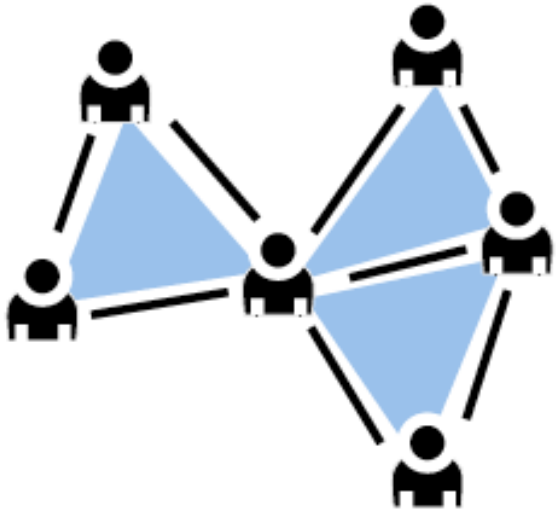
# Built-in Algorithms (Scala)

```
    // Continued from previous slide
    def pageRank(tol: Double): Graph[Double, Double]
    def triangleCount(): Graph[Int, ED]
    def connectedComponents(): Graph[VertexId, ED]
    // ...and more: org.apache.spark.graphx.lib
}
```

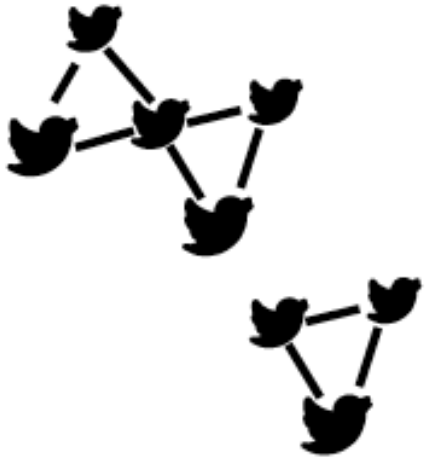
PageRank



Triangle Count

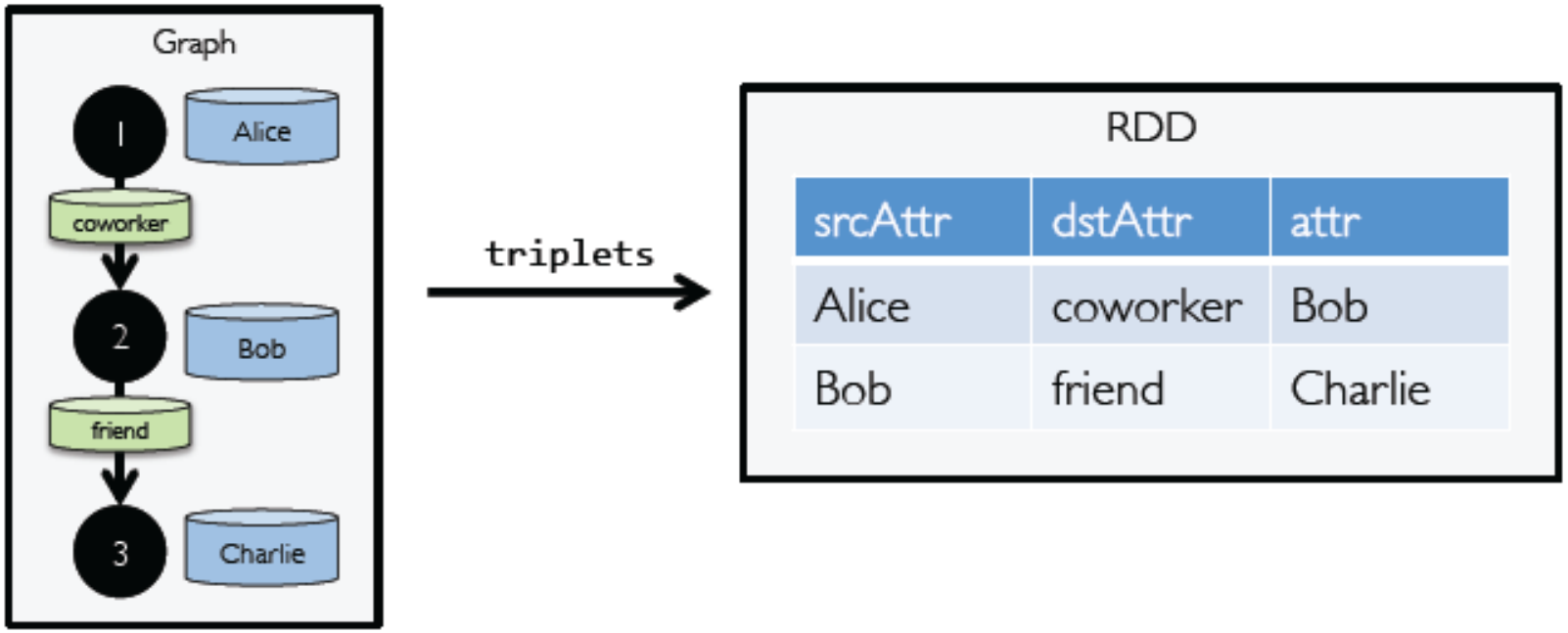


Connected Components



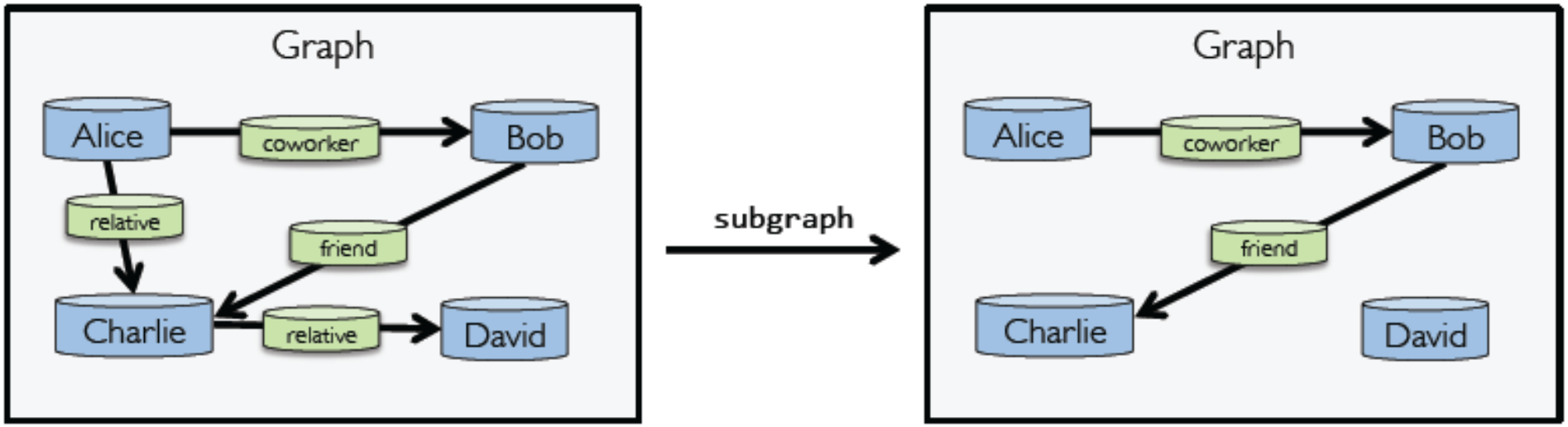
# The "triplets" view

```
class Graph[VD, ED] {
    def triplets: RDD[EdgeTriplet[VD, ED]]
}

class EdgeTriplet[VD, ED](
  val srcId: VertexId, val dstId: VertexId, val attr: ED,
  val srcAttr: VD, val dstAttr: VD)
```



# The `subgraph` transformation

```
class Graph[VD, ED] {
    def subgraph(epred: EdgeTriplet[VD, ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
}

graph.subgraph(epred = (edge) => edge.attr != "relative")
```

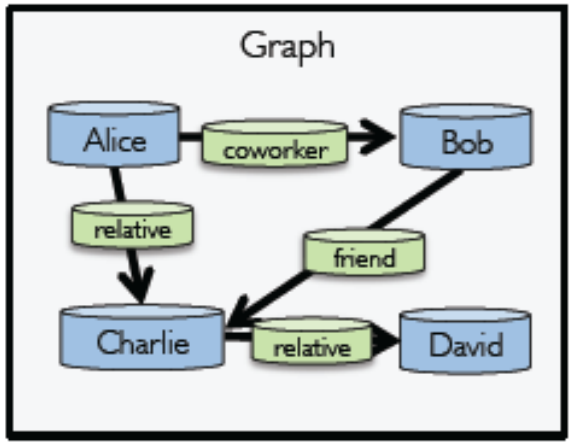


# Computation w/ `mapReduceTriplets`

```
class Graph[VD, ED] {
  def mapReduceTriplets[A](
      sendMsg: (EdgeTriplet[VD, ED]) => Iterator[(VertexId, A)],
      mergeMsg: (A, A) => A): RDD[(VertexId, A)]
}
```

upgrade to `aggregateMessages` in Spark 1.2.0

```
graph.mapReduceTriplets(
  edge => Iterator(
    (edge.srcId, 1),
    (edge.dstId, 1)),
  _ + _)
```




`mapReduceTriplets`

RDD

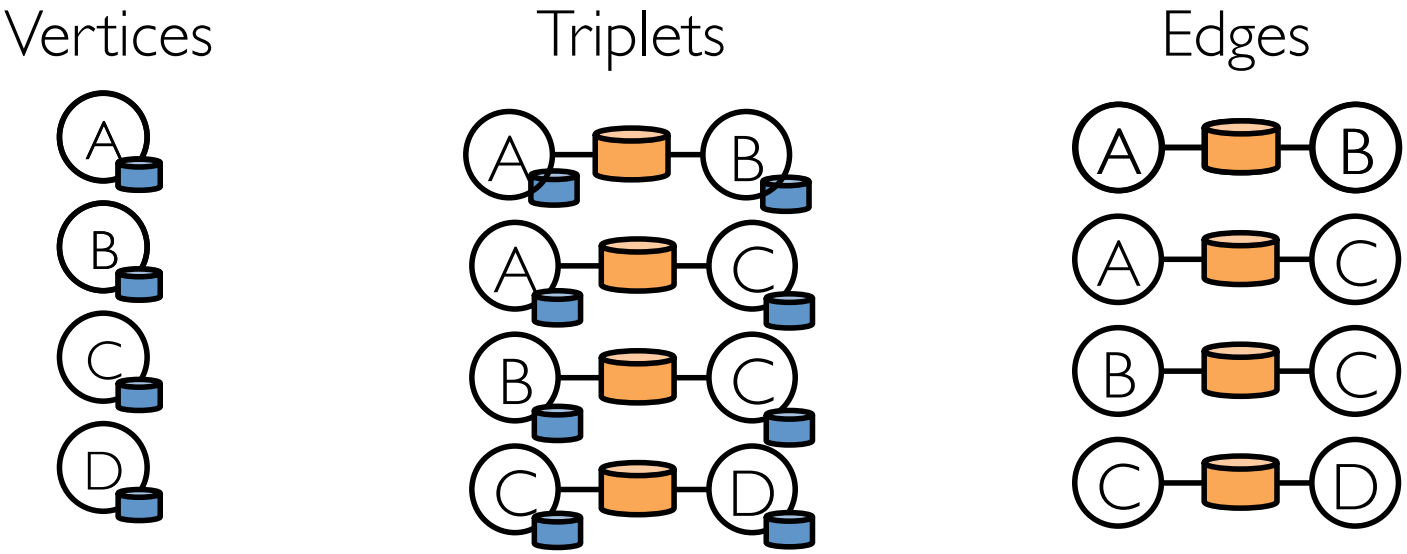
| vertex id | degree |
|---|---|
| Alice | 2 |
| Bob | 2 |
| Charlie | 3 |
| David | 1 |

# More (newer) Graph Operators

```
class Graph [ V, E ] {
  def Graph(vertices: Table[ (Id, V) ],
            edges: Table[ (Id, Id, E) ])
  // Table Views -----------------
  def vertices: Table[ (Id, V) ]
  def edges: Table[ (Id, Id, E) ]
  def triplets: Table [ ((Id, V), (Id, V), E) ]
  // Transformations ------------------------------
  def reverse: Graph[V, E]
  def subgraph(pV: (Id, V) => Boolean,
               pE: Edge[V,E] => Boolean): Graph[V,E]
  def mapV(m: (Id, V) => T ): Graph[T,E]
  def mapE(m: Edge[V,E] => T ): Graph[V,T]
  // Joins ----------------------------------------
  def joinV(tbl: Table [(Id, T)]): Graph[(V, T), E ]
  def joinE(tbl: Table [(Id, Id, T)]): Graph[V, (E, T)]
  // Computation ----------------------------------
  def mrTriplets(mapF: (Edge[V,E]) => List[(Id, T)],
                 reduceF: (T, T) => T): Graph[T, E]
}
```

# Triplets Join Vertices and Edges

The *triplets* operator joins vertices and edges:

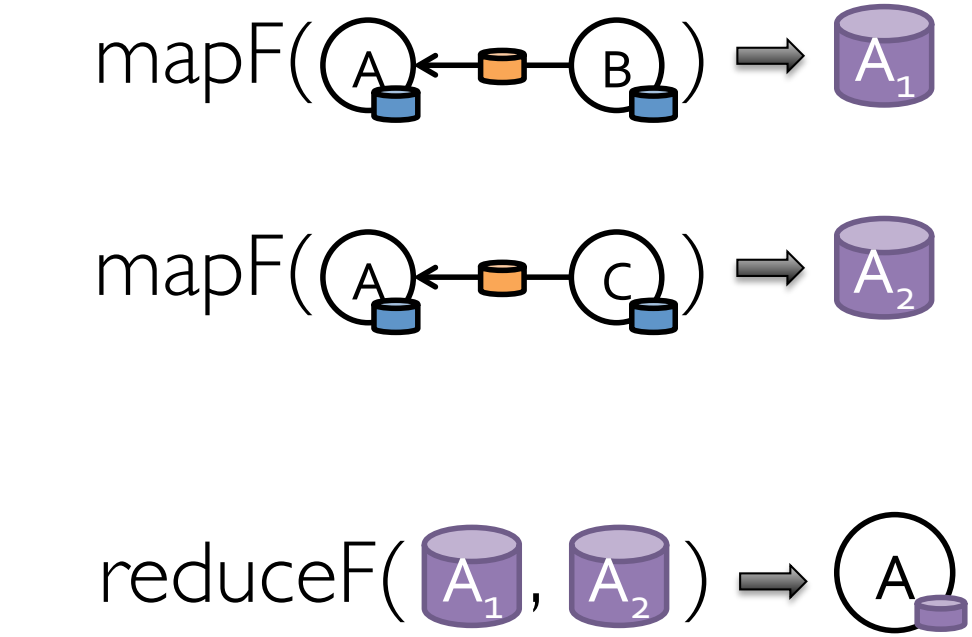


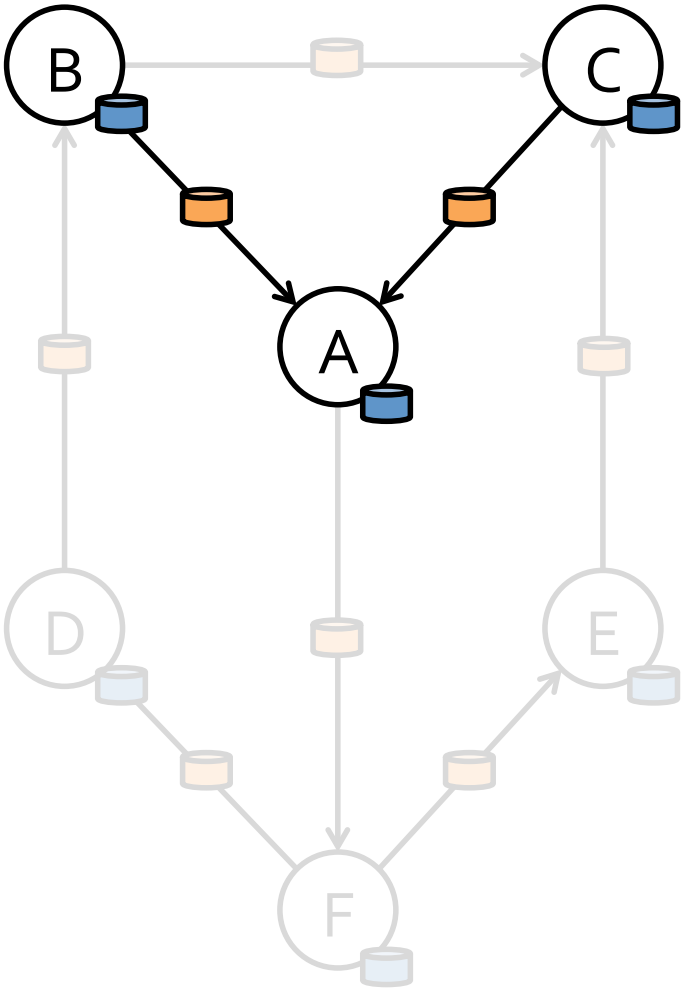

The *mrTriplets* operator sums adjacent triplets.

**SELECT** t.dstId, *reduceUDF*( *mapUDF*(t) ) **AS** sum
**FROM** triplets **AS** t **GROUPBY** t.dstId

# Map Reduce Triplets

*Map-Reduce* for each vertex

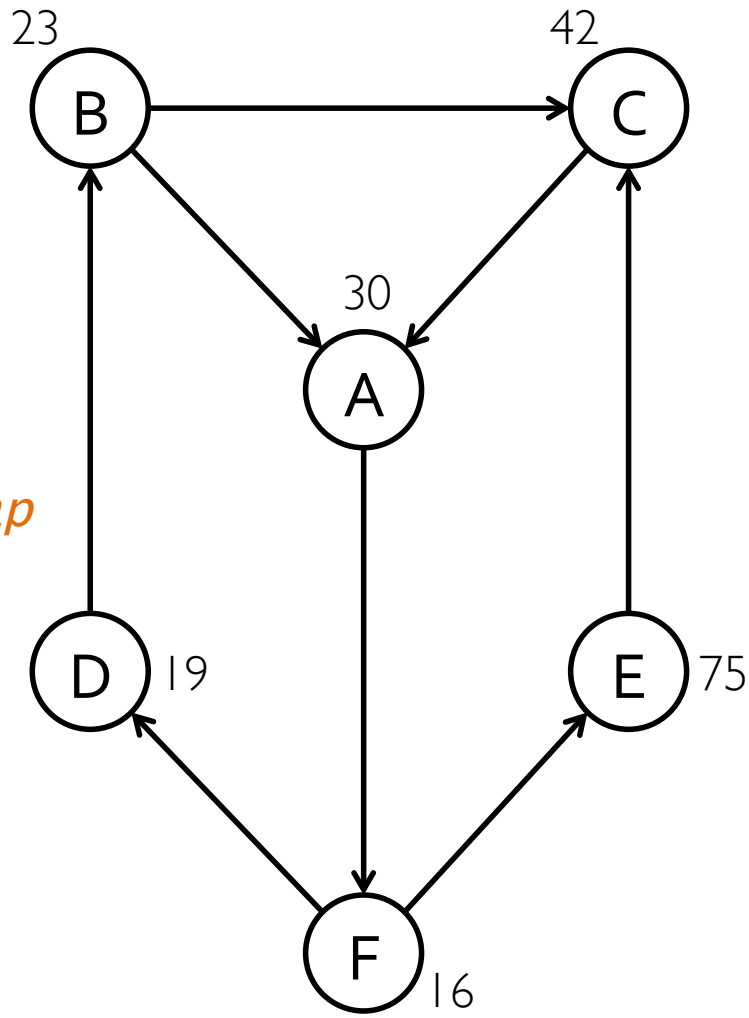


# Example: Oldest Follower

*What is the age of the oldest follower for each user?*

```
val oldestFollowerAge = graph
  .mrTriplets(
    e=> (e.dst.id, e.src.age),//Map
    (a,b)=> max(a, b) //Reduce
  )
  .vertices
```

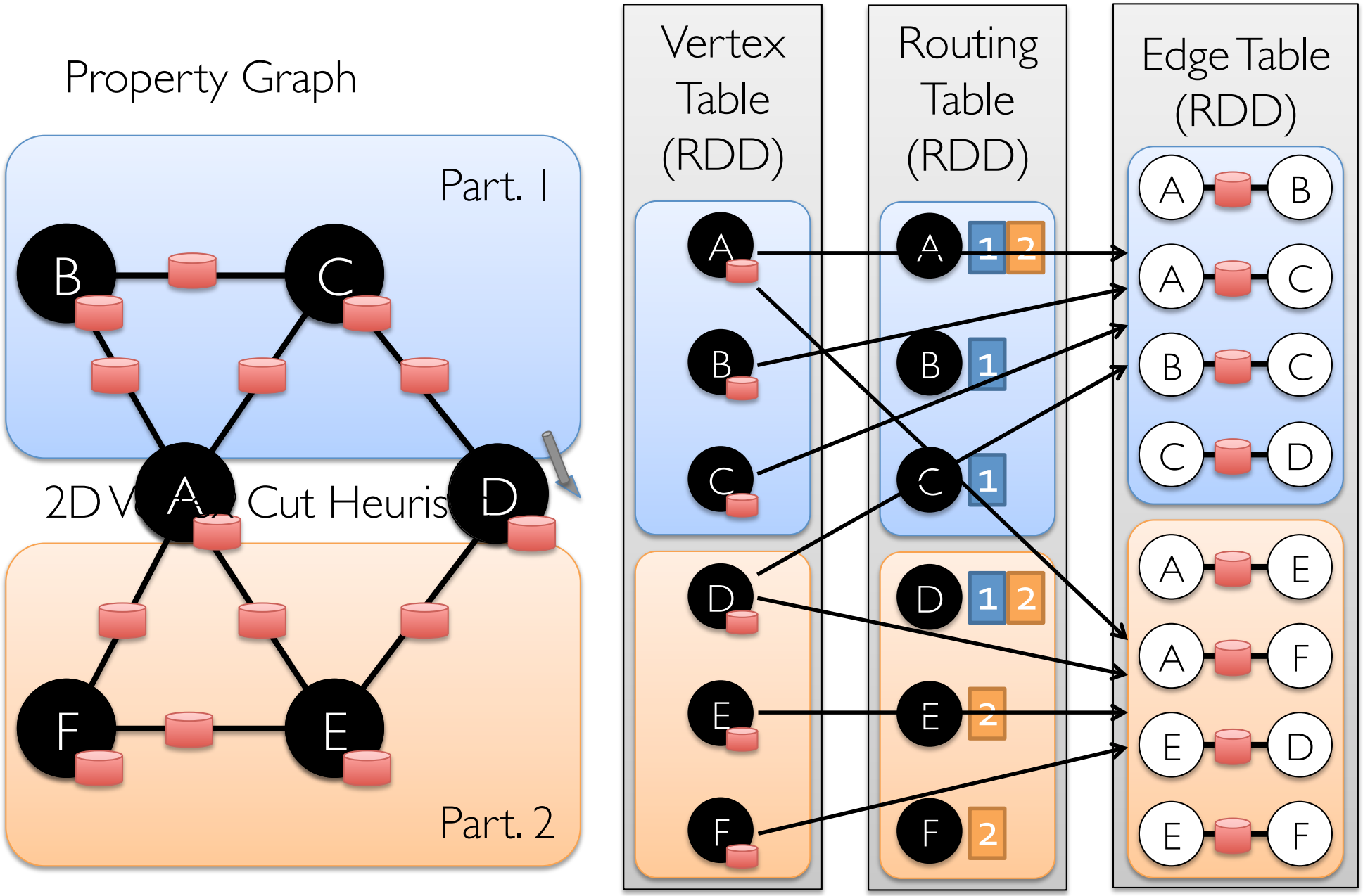


Have Expressed the Pregel and GraphLab abstractions using the GraphX operators in less than 50 lines of code!

By composing these operators we can construct entire graph-analytics pipelines.

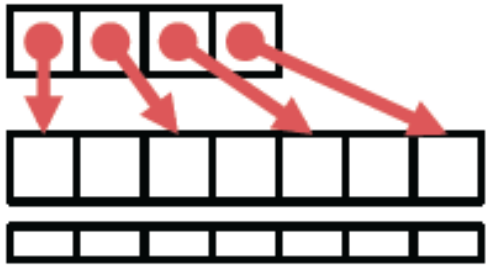
# GraphX System Design

# Distributed Graphs as Tables (RDDs)

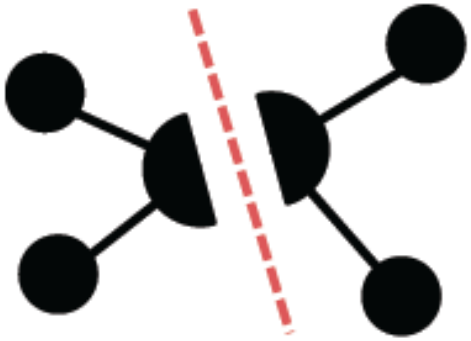


# Graph System Optimizations

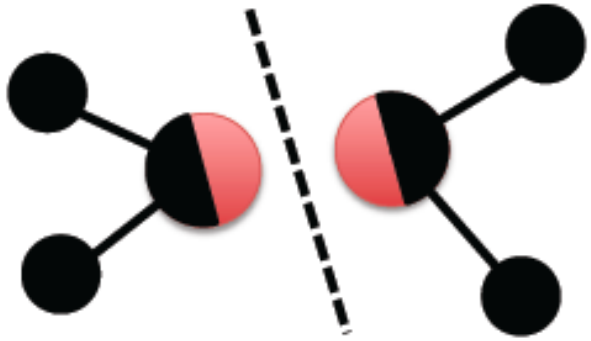
Specialized Data-Structures



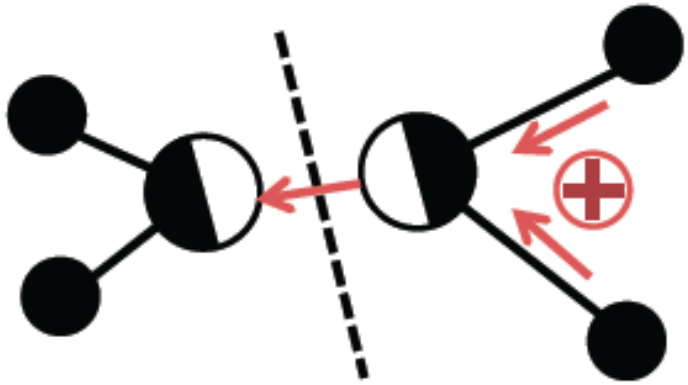
Vertex-Cuts Partitioning



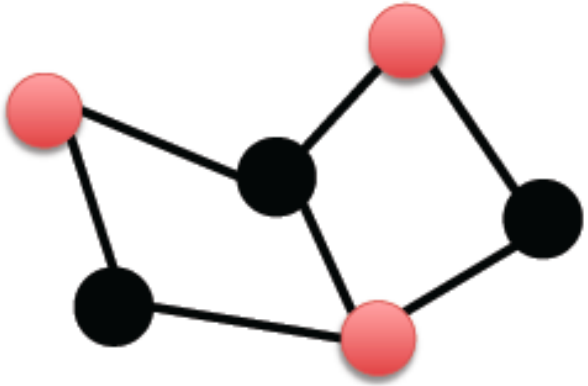
Remote Caching / Mirroring



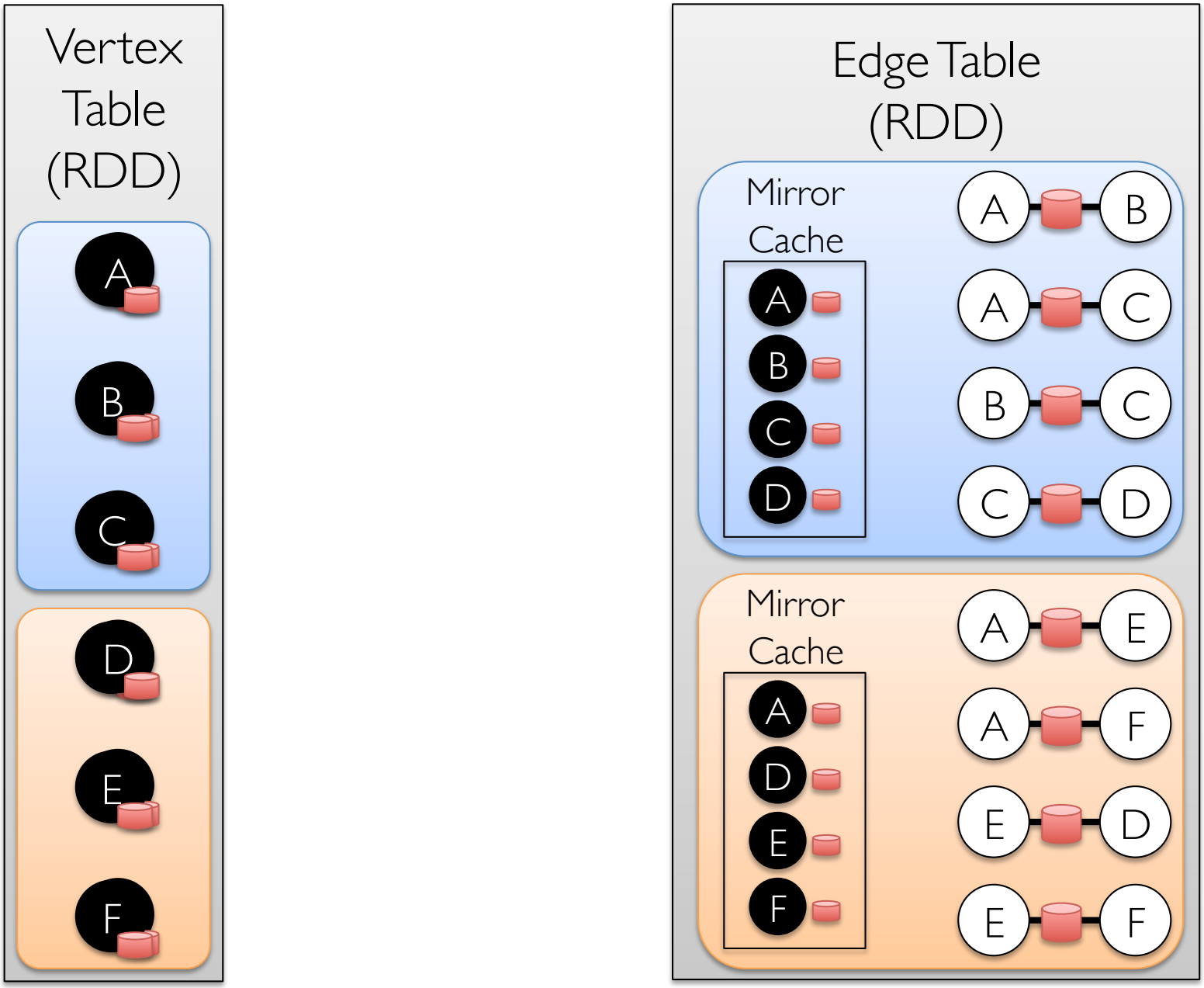
Message Combiners



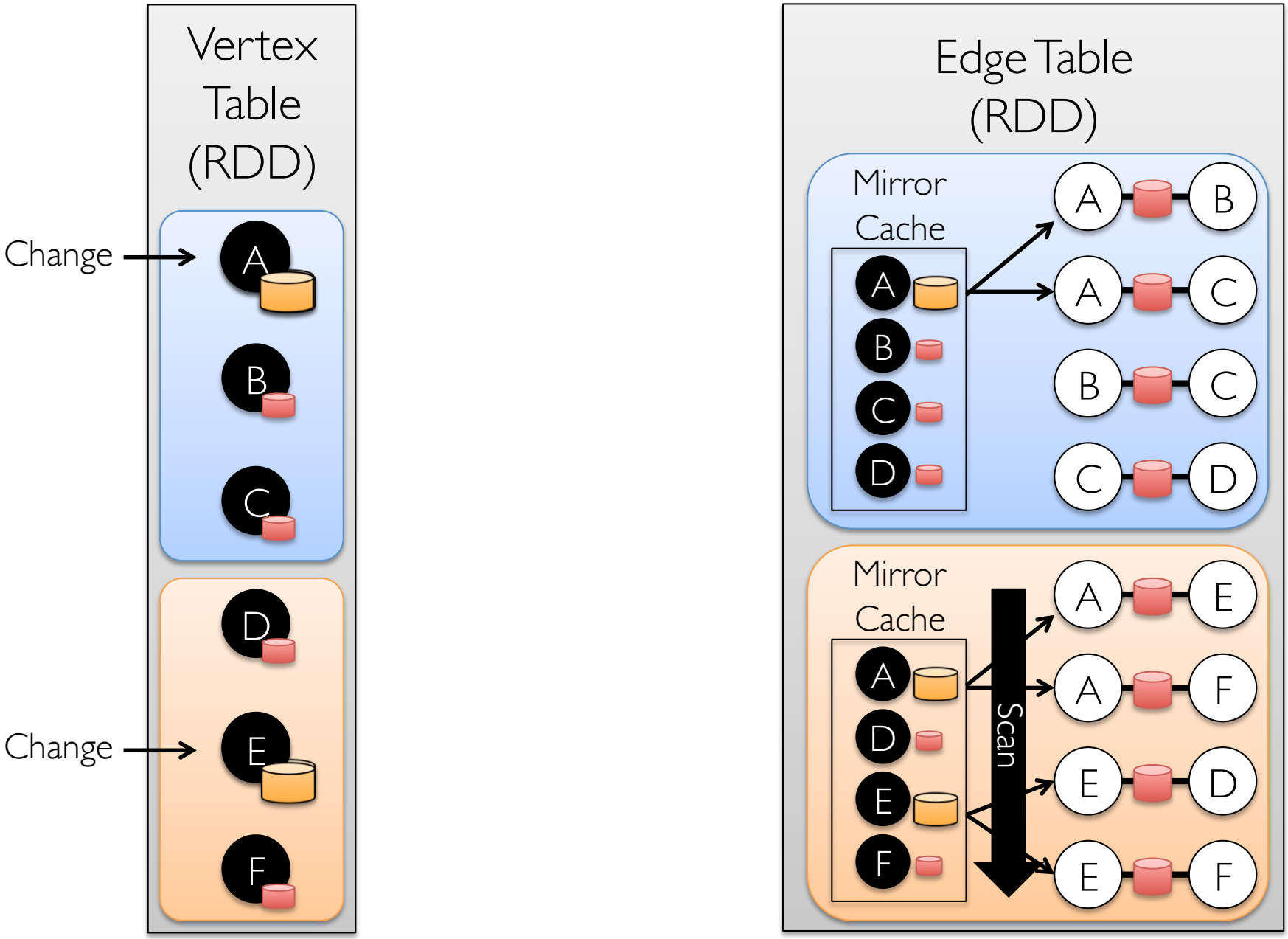
Active Set Tracking



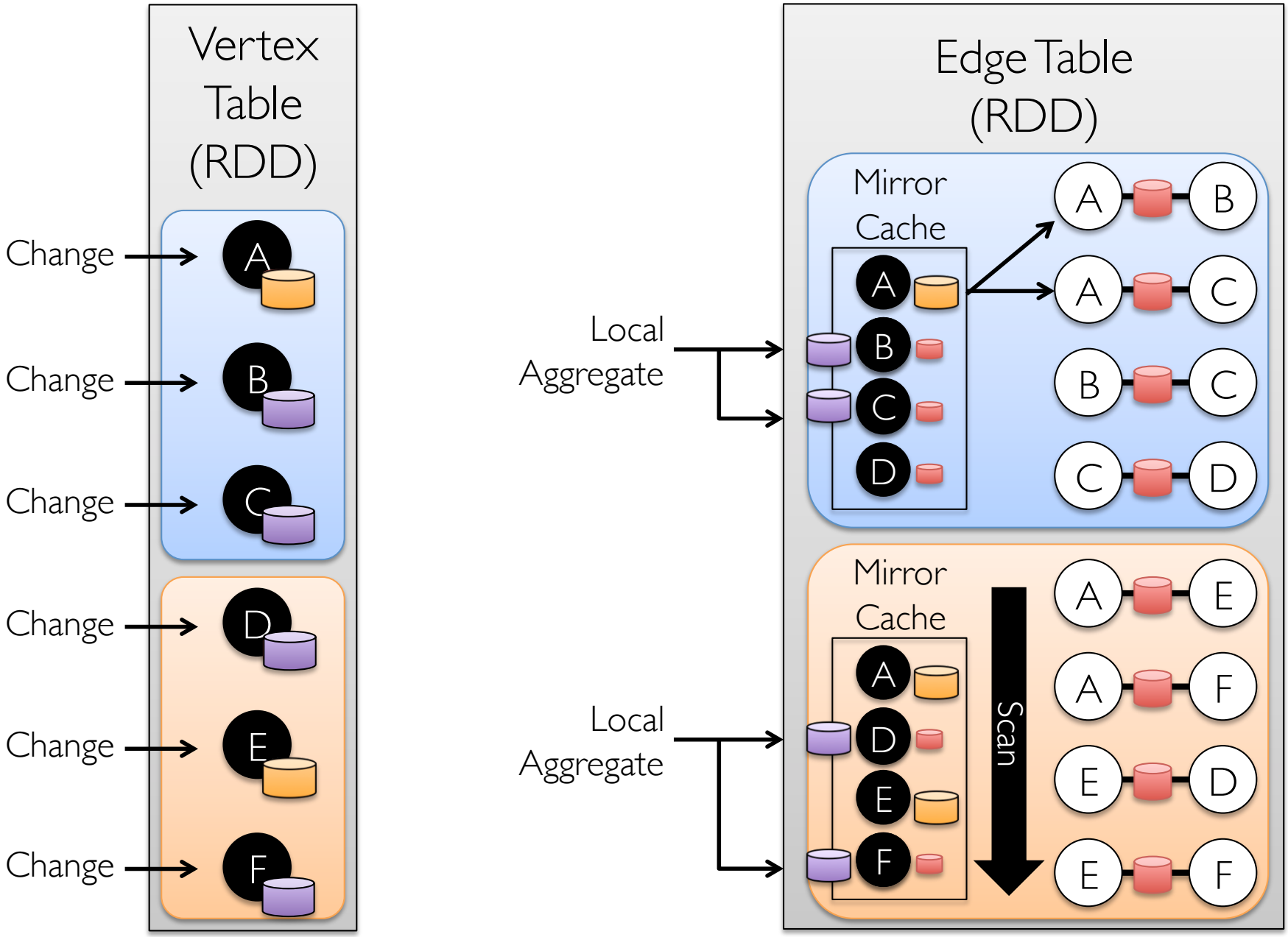
# Caching for Iterative mrTriplets



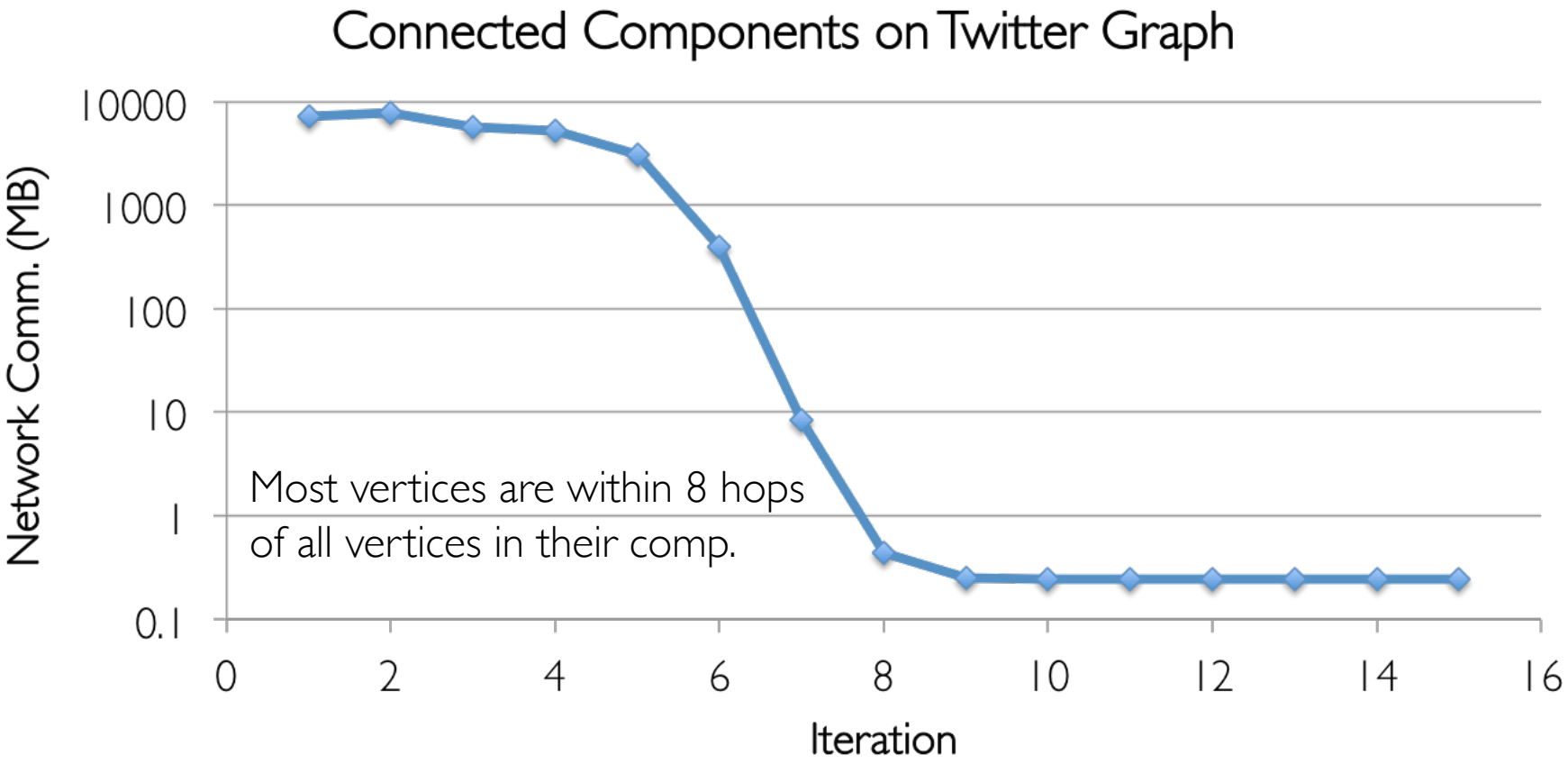
# Incremental Updates for Iterative mrTriplets



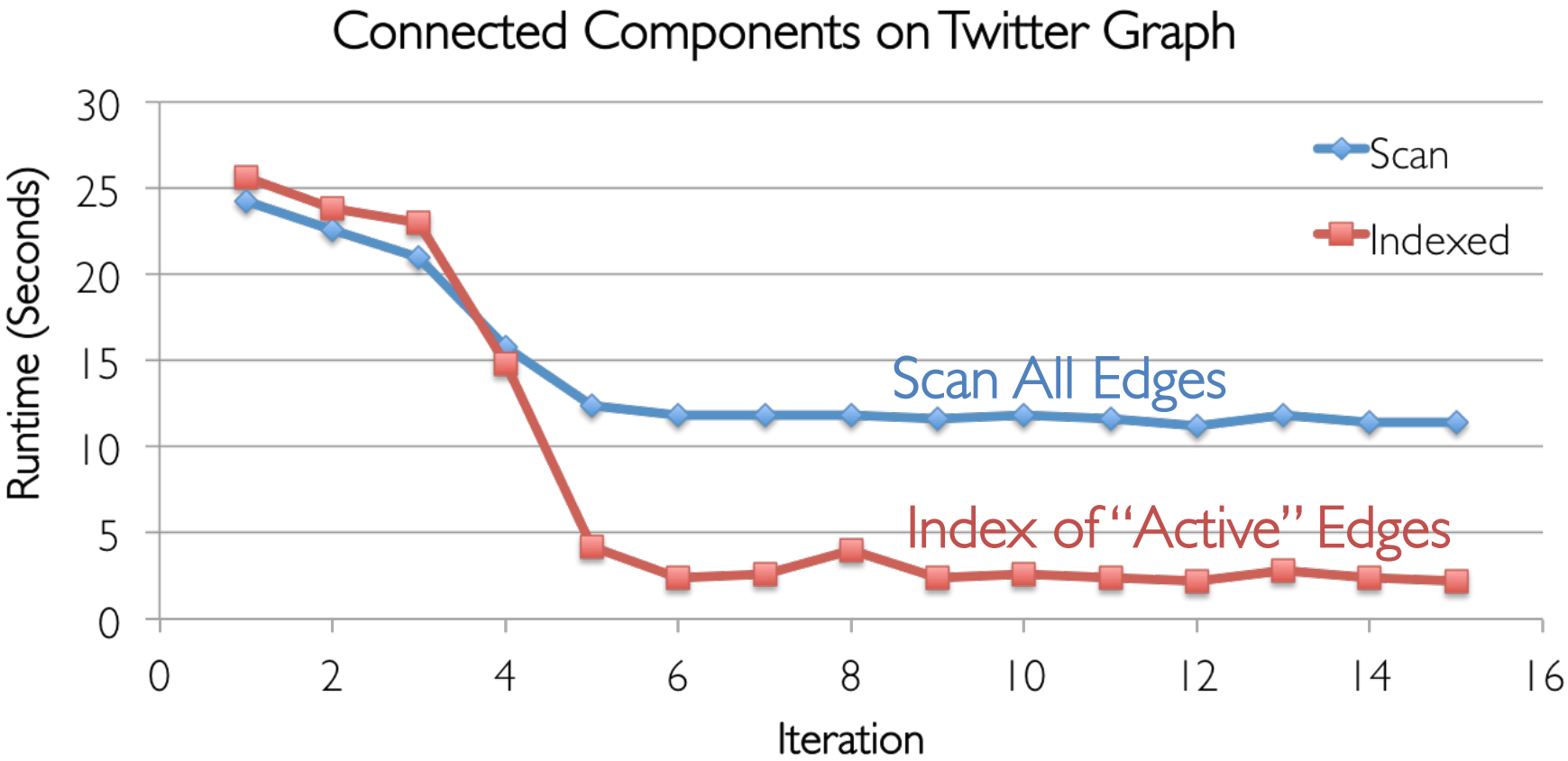
# Aggregation for Iterative mrTriplets



# Reduction in Communication Due to Cached Updates



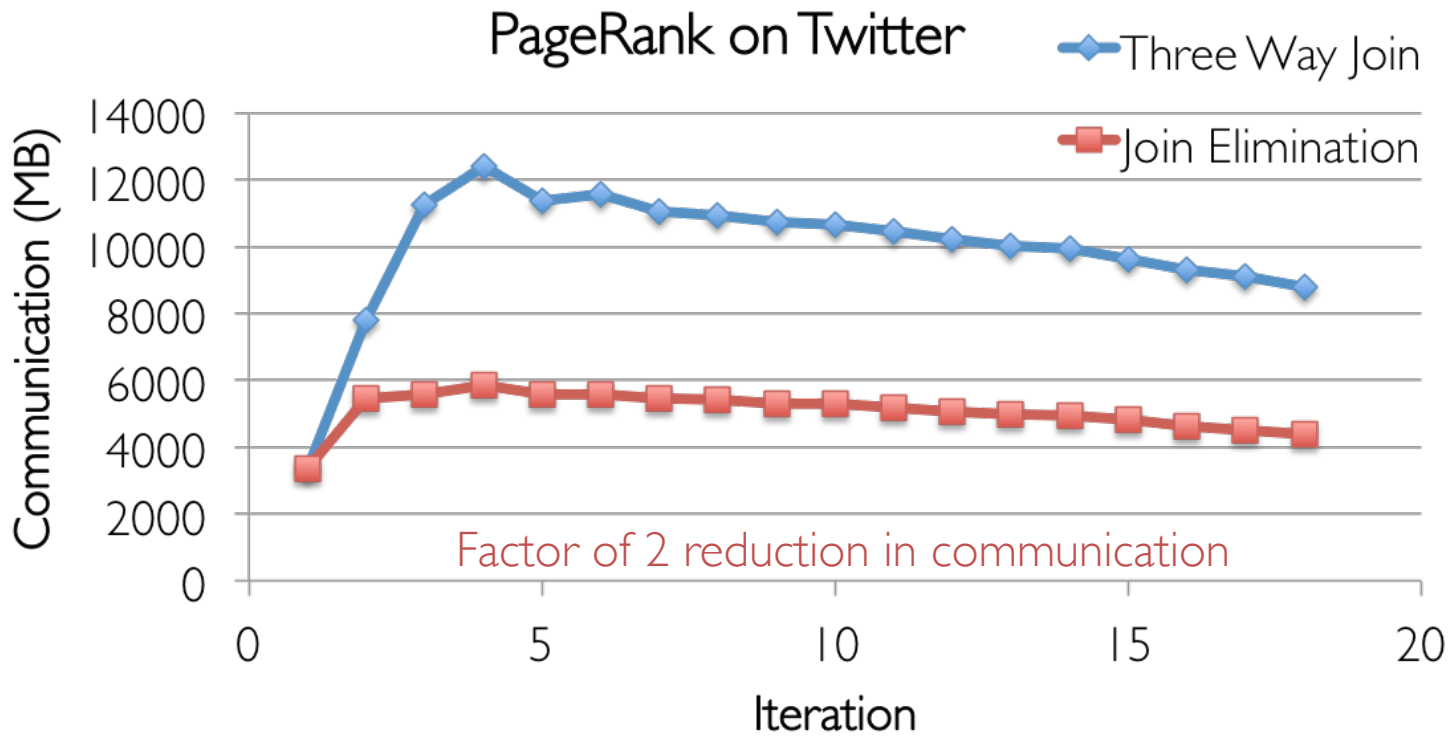
# Benefit of Indexing *Active* Edges



# Join Elimination

Identify and bypass joins for unused triplets fields

»*Example:* PageRank only accesses source attribute



# Additional Query Optimizations

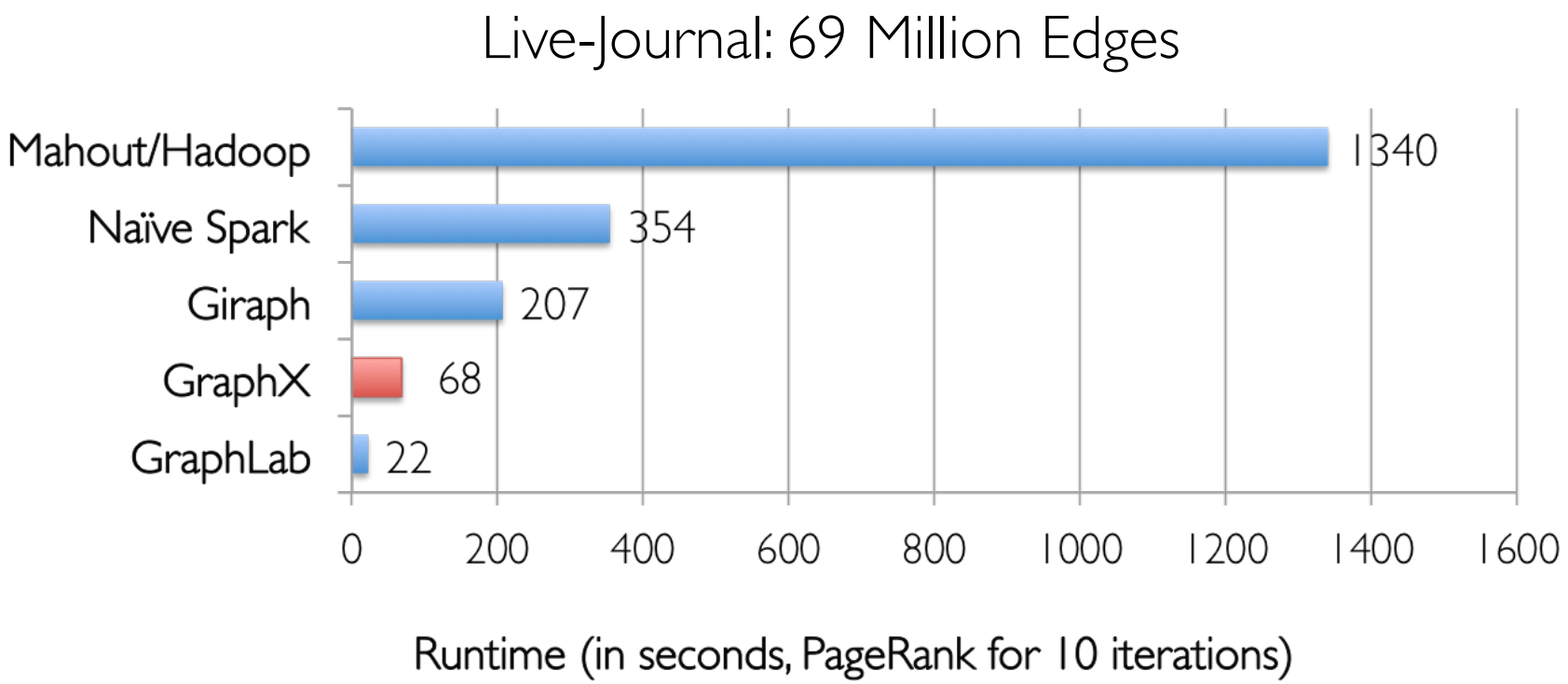
Indexing and Bitmaps:

- To accelerate joins across graphs
- To efficiently construct sub-graphs

Substantial Index and Data Reuse:

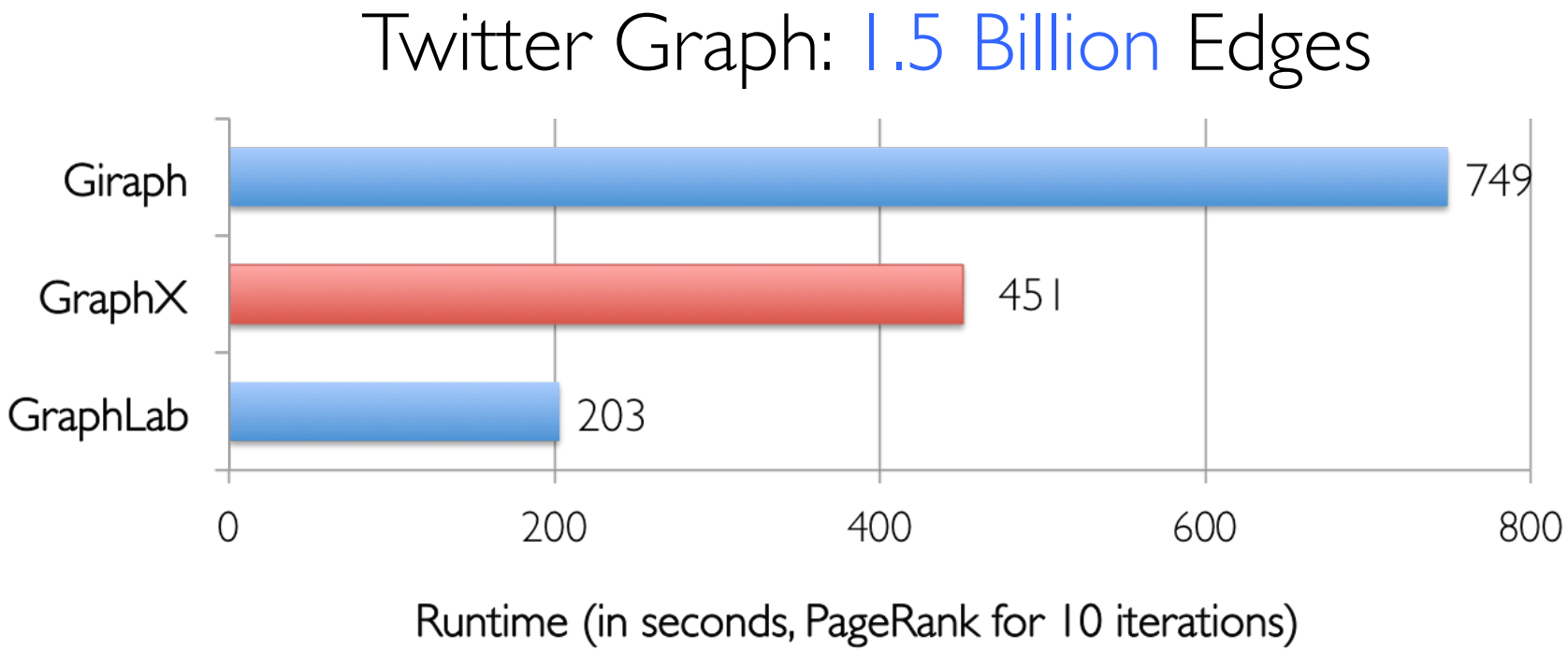
- Reuse routing tables across graphs and sub-graphs
- Reuse edge adjacency information and indices

# Performance Comparisons




GraphX is roughly 3x slower than GraphLab

# GraphX scales to larger graphs



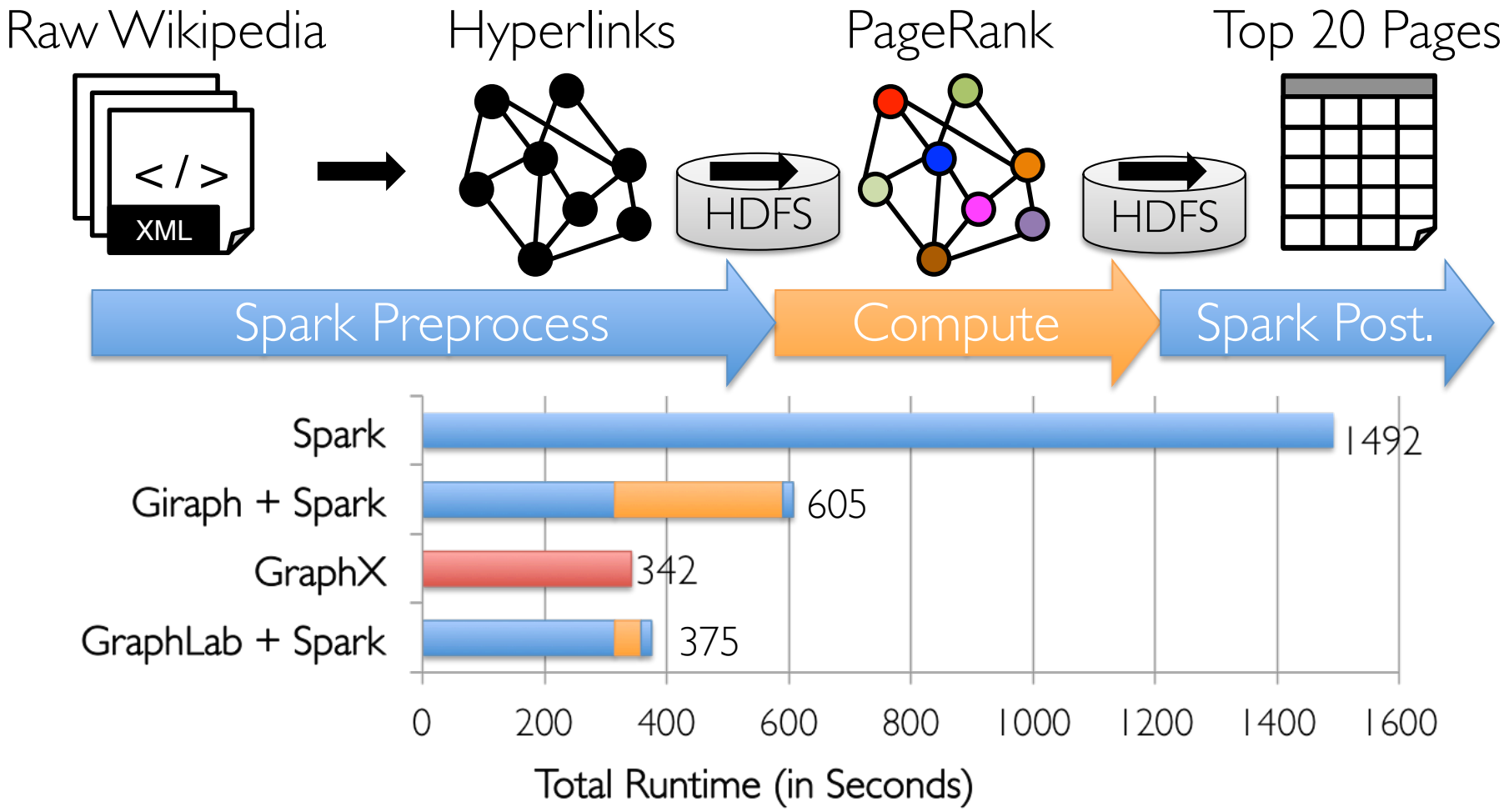

GraphX is roughly 2x slower than GraphLab

- Scala + Java overhead: Lambdas, GC time, …
- No shared memory parallelism: 2x increase in comm.

PageRank is just one stage….

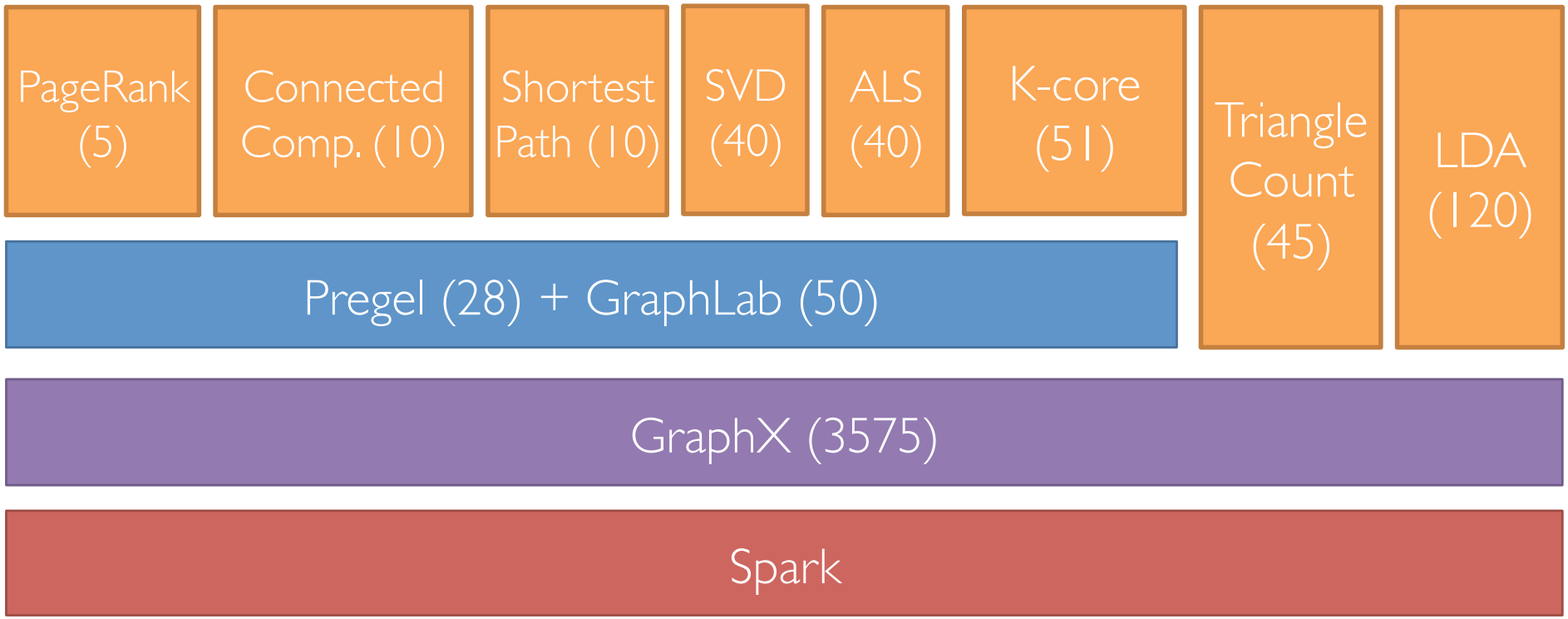
What about a pipeline?

# A Small Pipeline in GraphX




Timed end-to-end GraphX is *faster* than GraphLab

# The GraphX Stack (Lines of Code)



# Conclusion and Observations

Domain specific views: *Tables* and *Graphs*

- » tables and graphs are first-class composable objects
- » specialized operators which exploit view semantics

Single system that efficiently spans the pipeline

- » minimize data movement and duplication
- » eliminates need to learn and manage multiple systems

Graphs through the lens of database systems

- » Graph-Parallel Pattern → Triplet joins in relational alg.
- » Graph Systems → Distributed join optimizations

# Active Research

Static Data → Dynamic Data

- » Apply GraphX unified approach to time evolving data
- » Model and analyze relationships over time

Serving Graph Structured Data

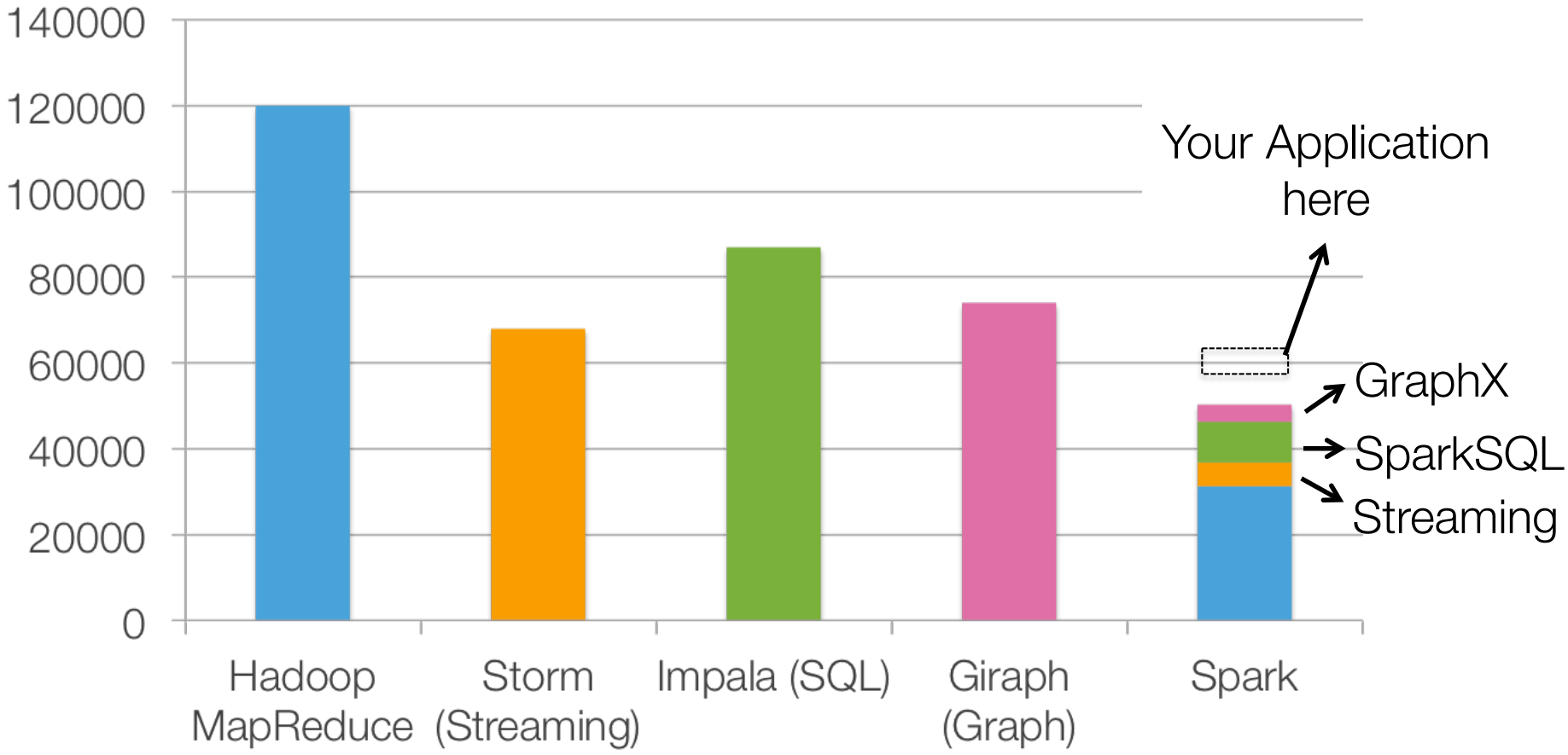
- » Allow external systems to interact with GraphX
- » Unify distributed graph databases with relational database technology

# More Info at:

http://amplab.github.io/graphx/

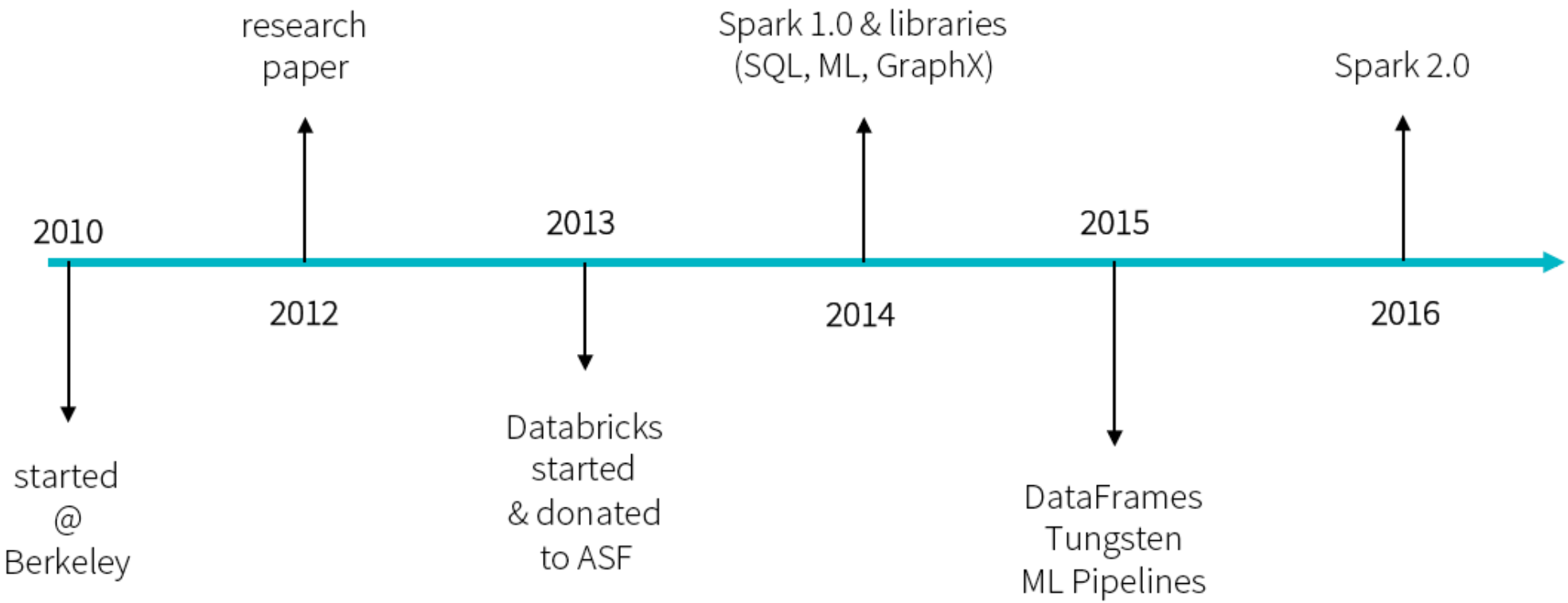
ankurd@eecs.berkeley.edu
crankshaw@eecs.berkeley.edu
rxin@eecs.berkeley.edu
jegonzal@eecs.berkeley.edu

# Powerful Stack – Agile Development




non-test, non-example source lines

# Spark/ BDAS Timeline so far…



# Major Features in Spark 2.0

Tungsten Phase 2
speedups of 5-10x

Structured Streaming
real-time engine
on SQL/DataFrames

Unifying Datasets
and DataFrames

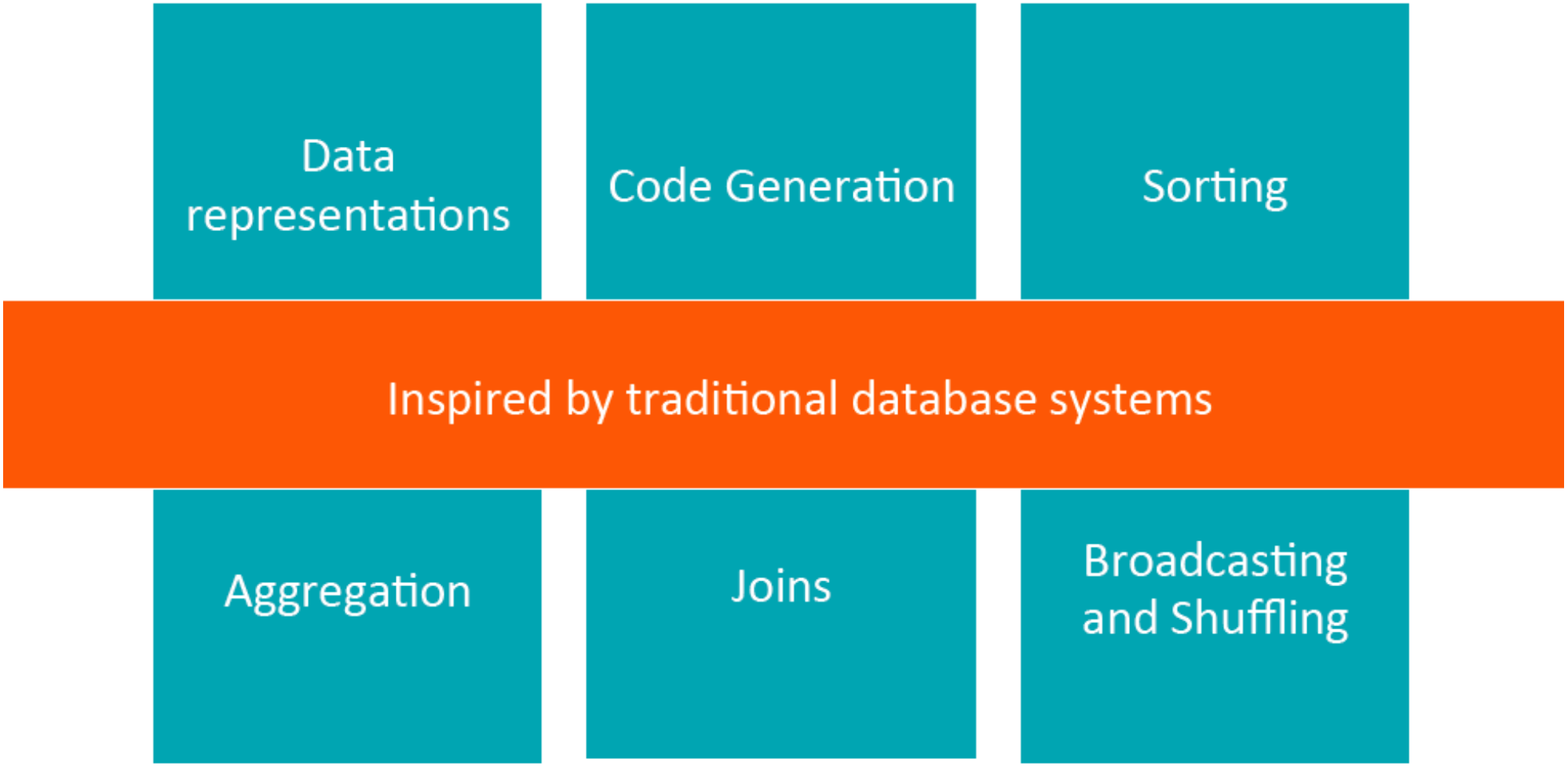
# Boosting Spark Performance via Project Tungsten

**Goal**

To Overcome JVM Performance limitations and bring Spark performance closer to Bare Metal via:

- **Native Memory Management and Binary Processing**: leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection
- **Cache-aware computation**: algorithms and data structures to exploit memory hierarchy
- **Runtime Code generation**: using code generation to exploit modern compilers and CPUs

# Project Tungsten: Key areas of Optimization

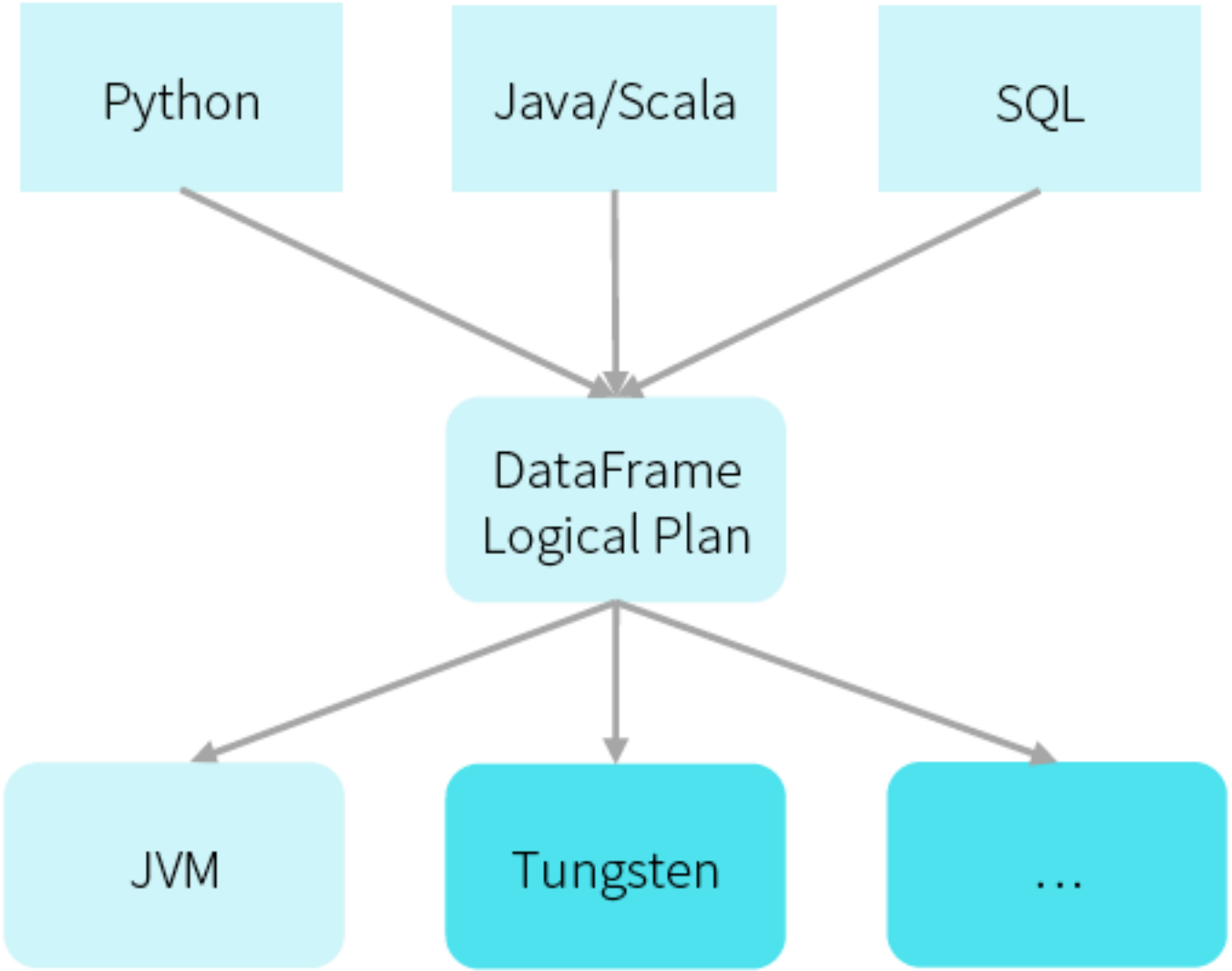


# Optimized Data Representations

- Java Objects have two downsides:
  - Space overheads
  - Garbage collection overheads

- Tungsten sidesteps these problems by performing its own manual memory management

# Further Performance Optimization via Project Tungsten



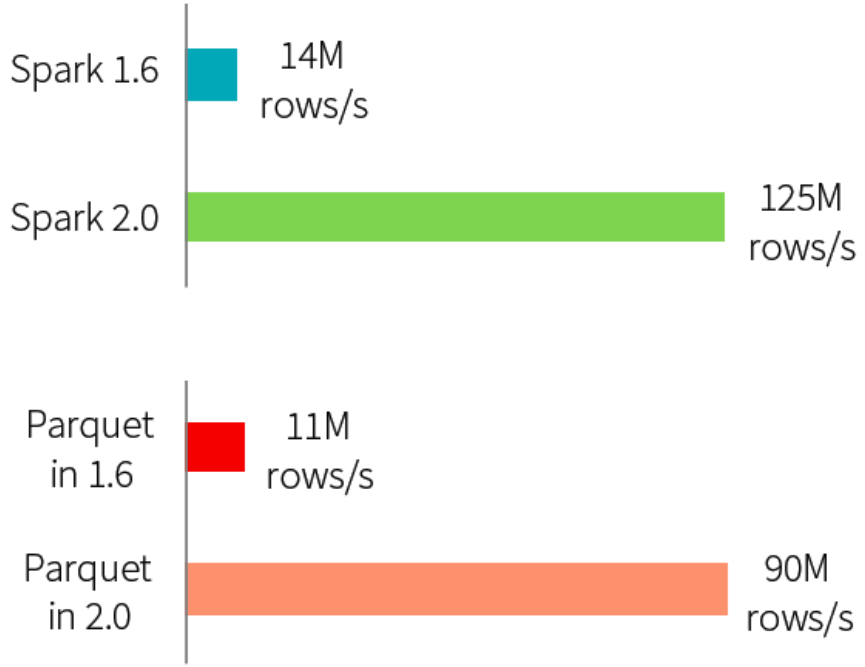
# Status of Tungsten (as of Feb 2016)

In Spark 1.4-1.6

- Added Binary Storage and Basic Code Generation
- DataFrame + Dataset APIs enable Tungsten in User Programs
- Tungsten also being used under SparkSQL + parts of MLlib

For Spark 2.0 (1H2016 Release)

- Whole-stage Code Generation
  - Remove expensive Iterator calls
  - Fuse across multiple operators
- Optimized Input/Output
  - Parquest + Built-in Cache




Automatically applies to SQL, DataFrames, Datasets

# Datasets and DataFrames

- DataFrames are collections of rows with a schema
- Datasets add static types, e.g. Dataset[Person]
- Both run on Tungsten
- Spark 2.0 will merge these APIs:

```
Dataframe = Dataset[Row]
```

Benefits of Merging

- Simpler to understand
  - Only kept Dataset separate to keep binary compatibility in Spark 1.x
- Libraries can take data of both forms
- With Streaming, same API will also work on streams

# Example for Datasets and DataFrames

```
case class User(name: String, id: Int)
case class Message(user: User, text: String)

dataframe = sqlContext.read.json("log.json")      // DataFrame, i.e. Dataset[Row]
messages = dataframe.as[Message]                  // Dataset[Message]

users = messages.filter(m => m.text.contains("Spark"))
                .map(m => m.user)                 // Dataset[User]

pipeline.train(users)                  // MLlib takes either DataFrames or Datasets
```

# Datasets API

- Type-safe: Operate on domain objects with compiled lambda fucntions

```
val df = ctx.read.json("people.json")

// Convert data to domain objects.
case class Person(name: String, age: Int)
val ds: Dataset[Person] = df.as[Person]
ds.filter(_.age > 30)

// Compute histogram of age by name.
val hist = ds.groupBy(_.name).mapGroups {
  case (name, people: Iter[Person]) =>
    val buckets = new Array[Int](10)
    people.map(_.age).foreach { a =>
      buckets(a / 10) += 1
    }
    (name, buckets)
}
```

# Datasets: Another new Structured Data Abstraction and its API in Spark

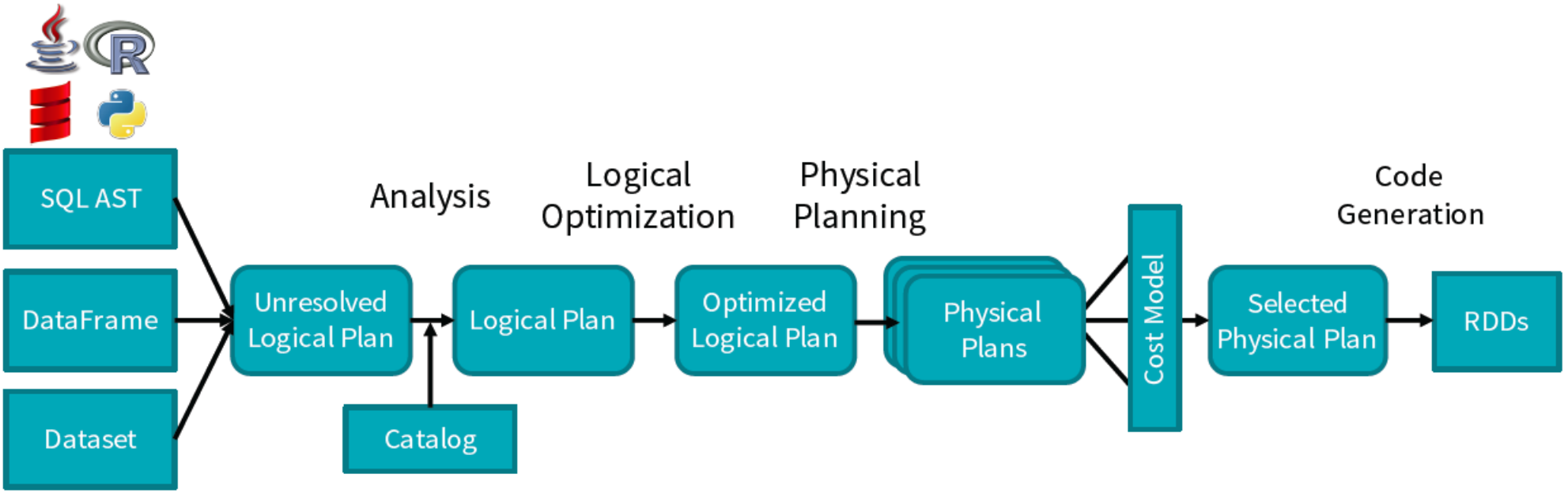
| | SQL | DataFrames | Datasets |
|---|---|---|---|
| Syntax Errors | Runtime | Compile Time | Compile Time |
| Analysis Errors | Runtime | Runtime | Compile Time |

**Analysis errors reported before a distributed job starts**

# Long-Term Direction

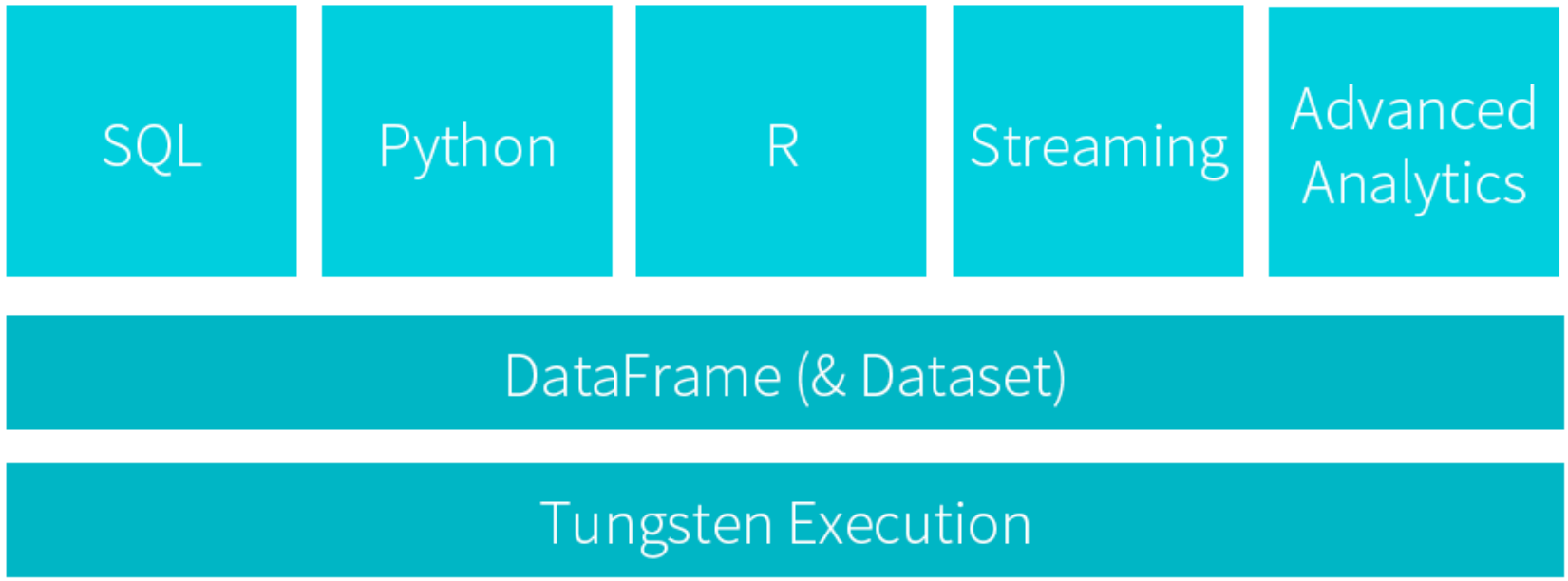
- RDD will remain the low-level API in Spark
- Datasets and DataFrames give richer semantics and optimizations
  - New libraries will increasingly use these as interchange format, e.g. Structured Streaming, MLlib and GraphFrames

# Shared Optimization and Execution




DataFrames, Datasets and SQL
share the same optimization/execution pipeline

# Execution Architecture for Spark Going Forward (Ver. 2.0 and Beyond)

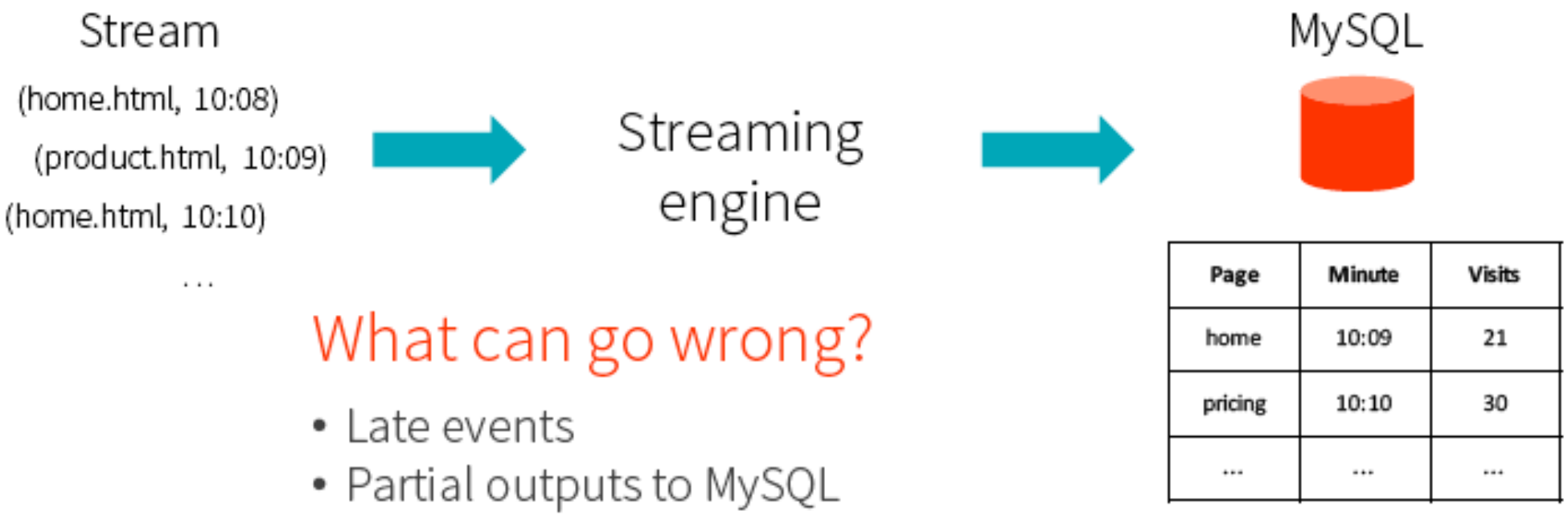


# Motivation for Structured Streaming

- Real-time processing is increasingly important
- Most applications need to combine it with Batch & Interactive queries, e.g.
  - Track state using a stream, then run SQL queries
  - Train an Machine Learning model offline, then update it with new, online data

Not just streaming, but "continuous applications"

# Challenges of Integrating Streaming into a Real-world Application Infrastructure




## What can go wrong?

- Late events
- Partial outputs to MySQL
- State recovery on failure
- Distributed reads/writes
- ...

# Complex Programming Models

**Data**

Late arrival, varying distribution over time, …

**Processing**

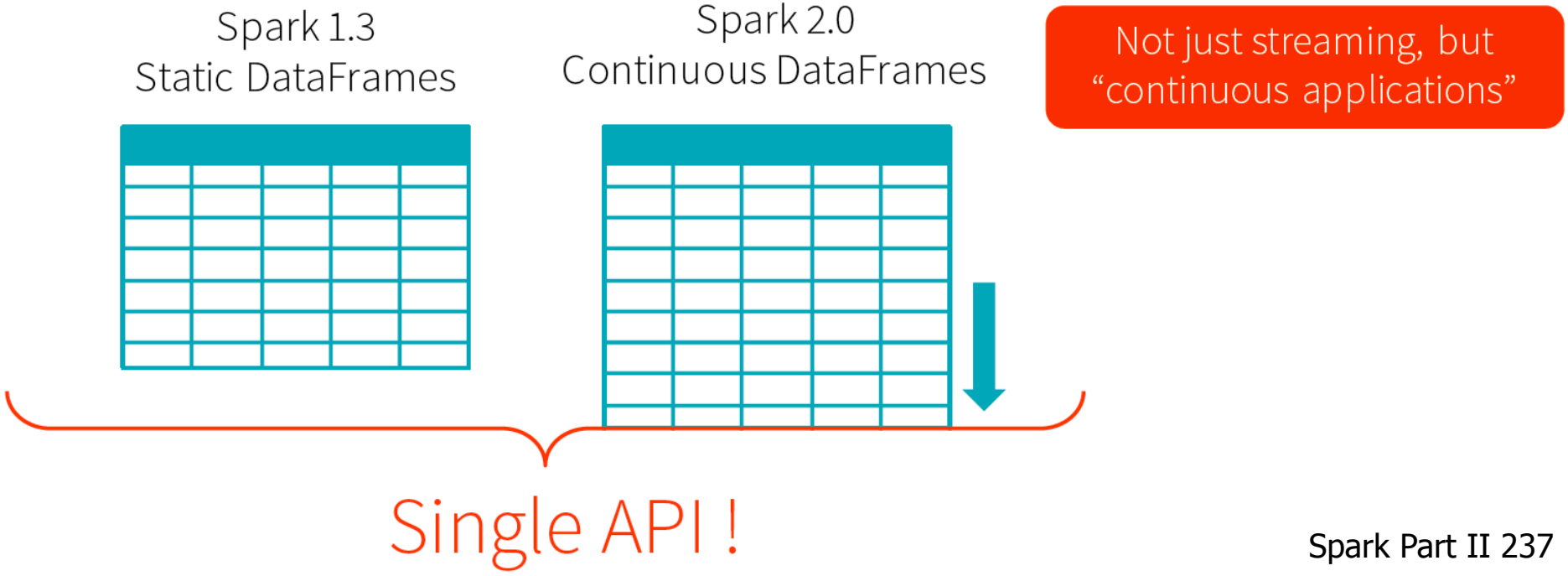
Business logic change & new ops
(windows, sessions)

**Output**

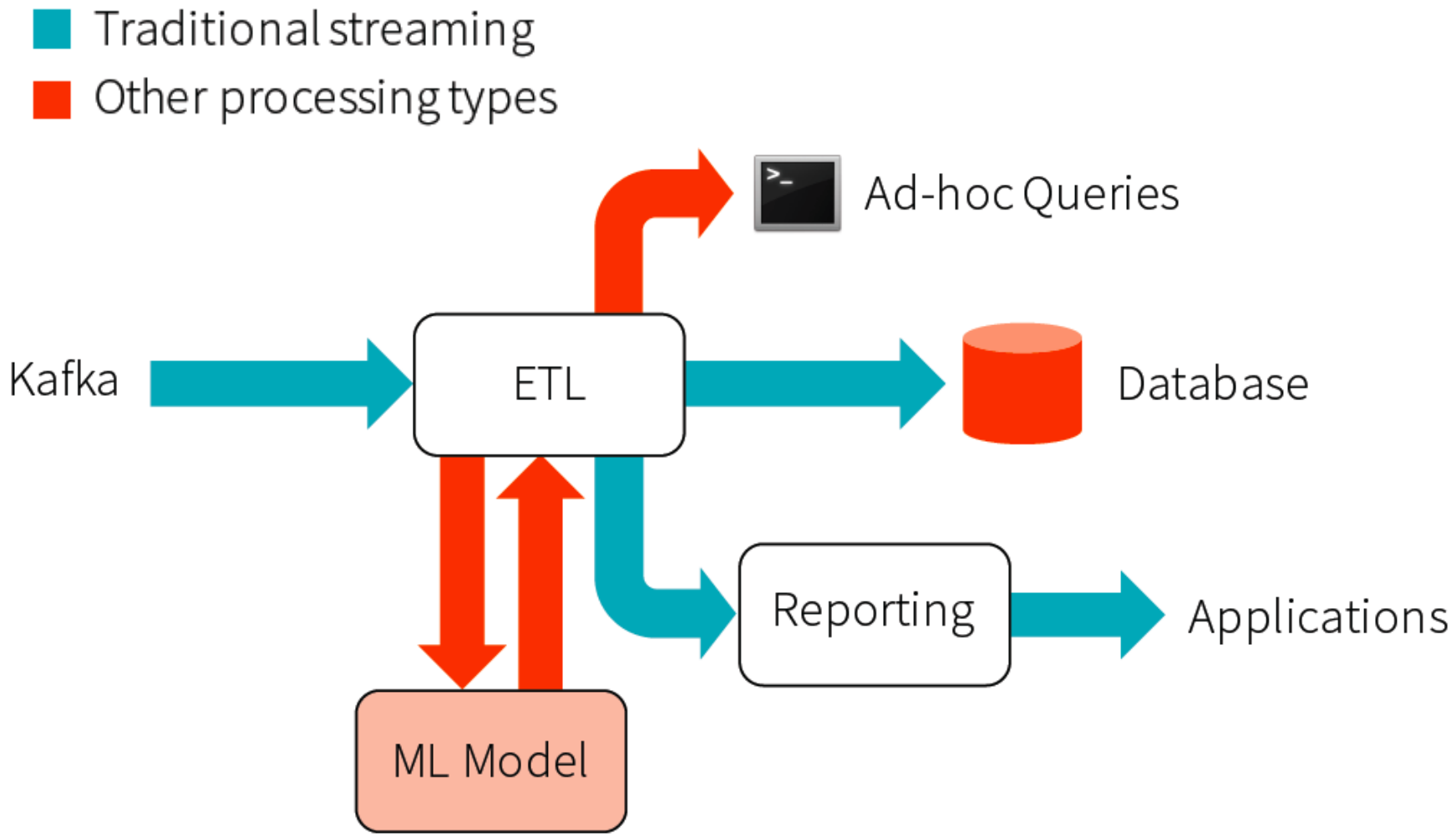
How do we define
output over time & correctness?

# Structured Streaming

- High-level Streaming API built on Spark SQL engine
  - Run the same queries on DataFrames
  - Event-time, windowing, sessions, source and sinks
- Unify Streaming, Interactive and Batch Queries
  - Aggregate data in a stream, then serve using JDBC
  - Change queries at runtime
  - Build and Apply Machine Learning models

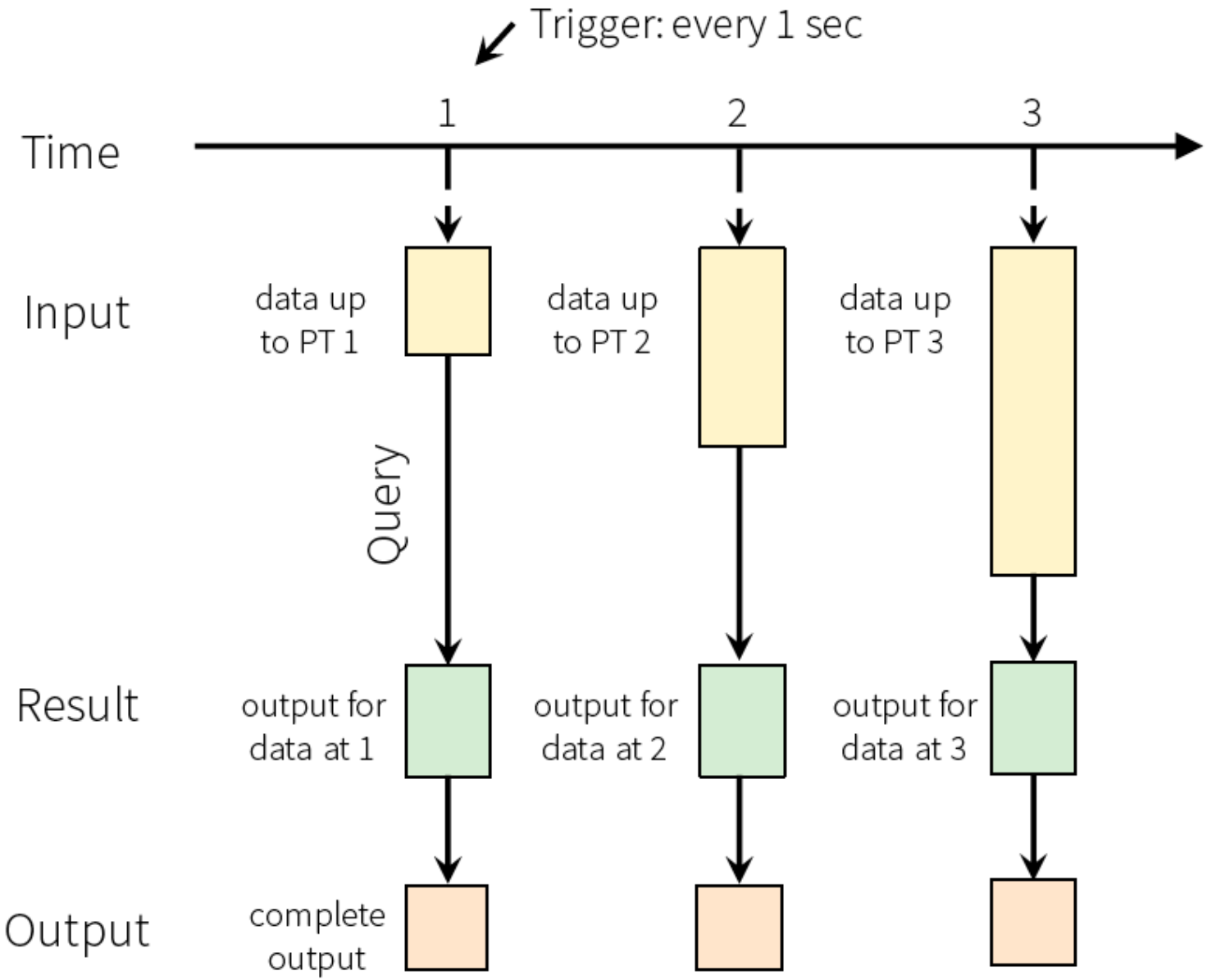


# An Example

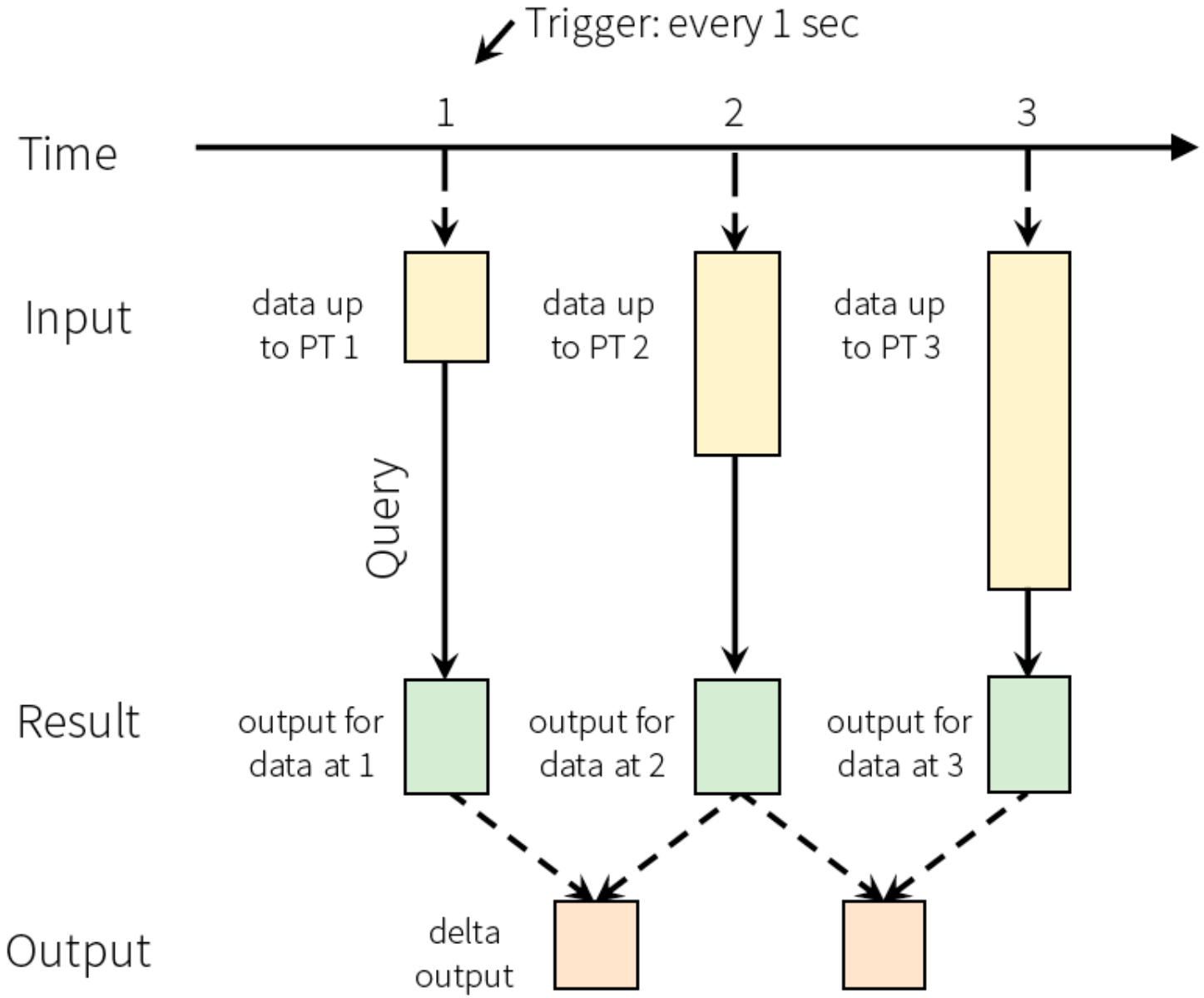



Goal: end-to-end continuous applications

# Model for Structured Streaming



# Model for Structured Streaming



# Model Details for Structured Streaming

- **Input Sources:** Append-Only Tables
- **Queries:** New operators for Windowing, Sessions, etc
- **Triggers:** based on time (e.g. every 1 sec)
- **Output modes:** Complete, Deltas, Update-in-Place

# An ETL Example

- **Input:** files in S3
- **Query:** map (transform each record)
- **Trigger:** “every 5 sec”
- **Output mode:** “new records”, into S3 sink

# A Page View Count Example

- **Input:** records in Kafka
- **Query:** select count (*) group by page, minute(evtime)
- **Trigger:** “every 5 sec”
- **Output mode:** “update-in-place”, into MySQL sink

NOTE: This will automatically update “old” records on late data

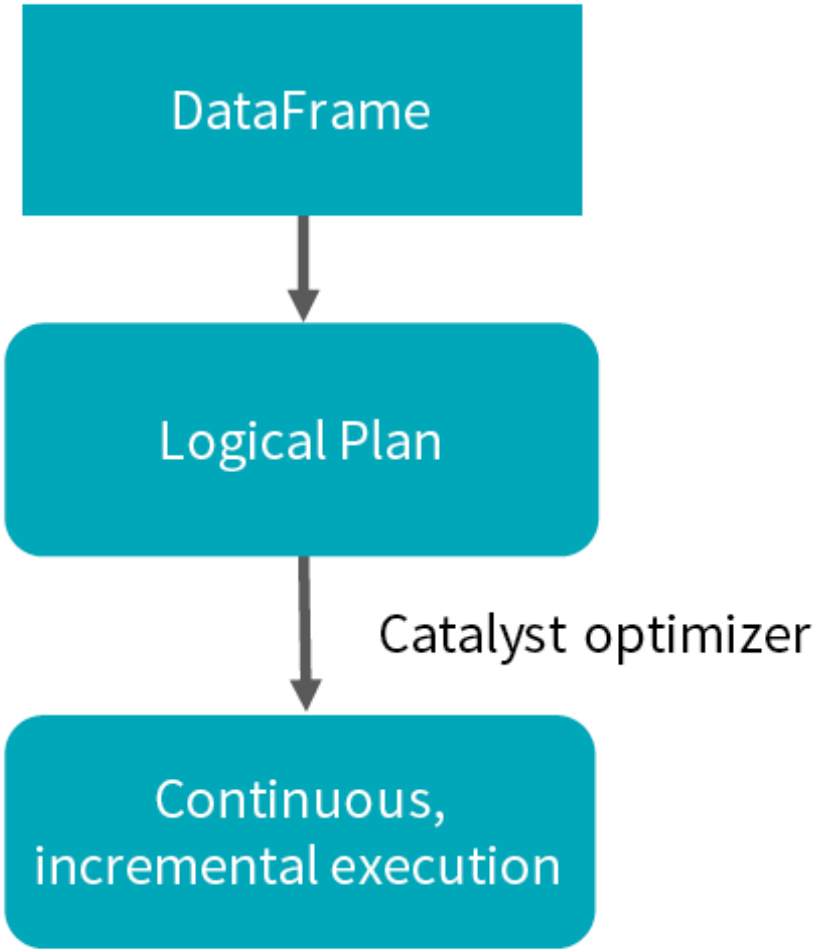
# Execution for Structured Streaming

Logically:

- DataFrame operations on static data (i.e. as easy to understand as batch)

Physically:

- Spark automatically runs the query in Streaming fashion (i.e. incrementally and continuously)



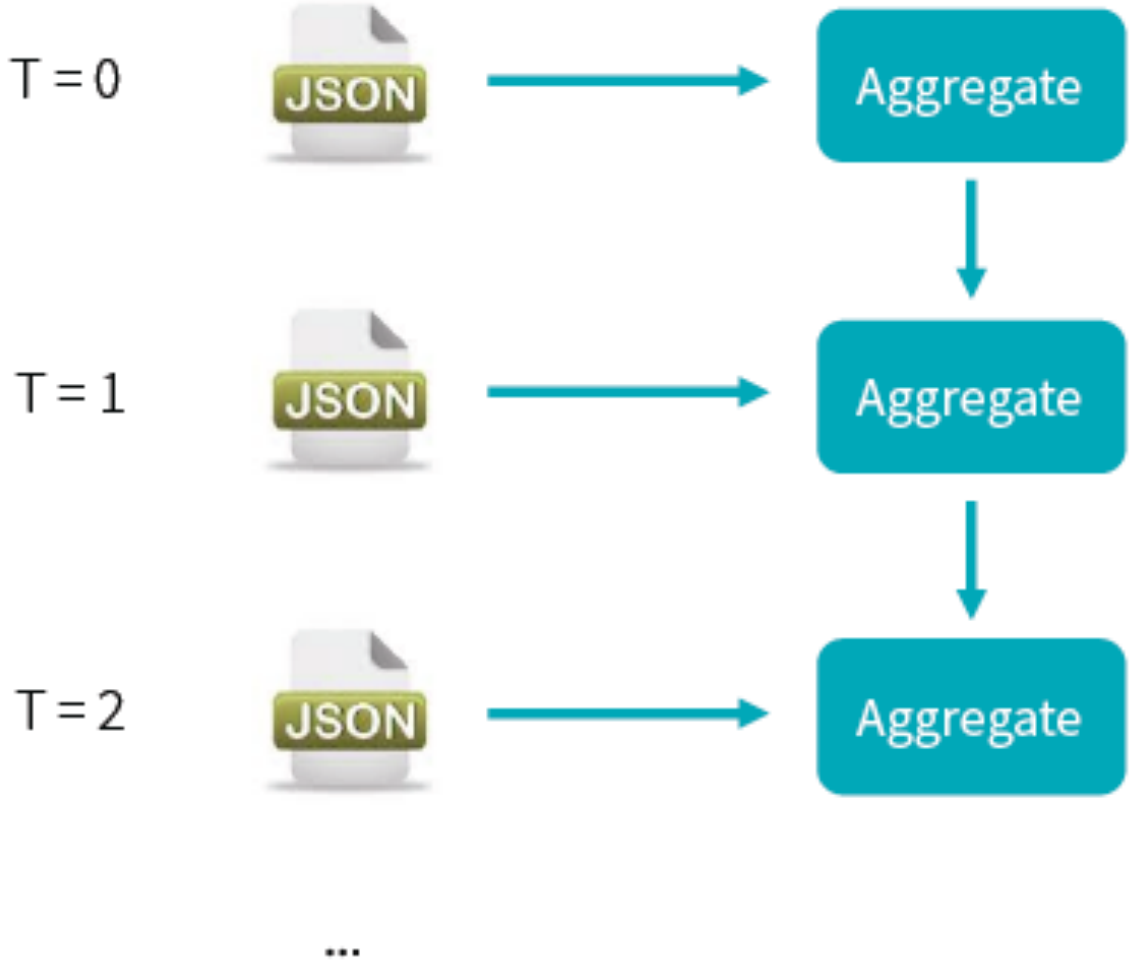
# Example: Batch Aggregation

```
logs  = ctx.read.format("json").open("s3://logs")

logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .save("jdbc:mysql//...")
```

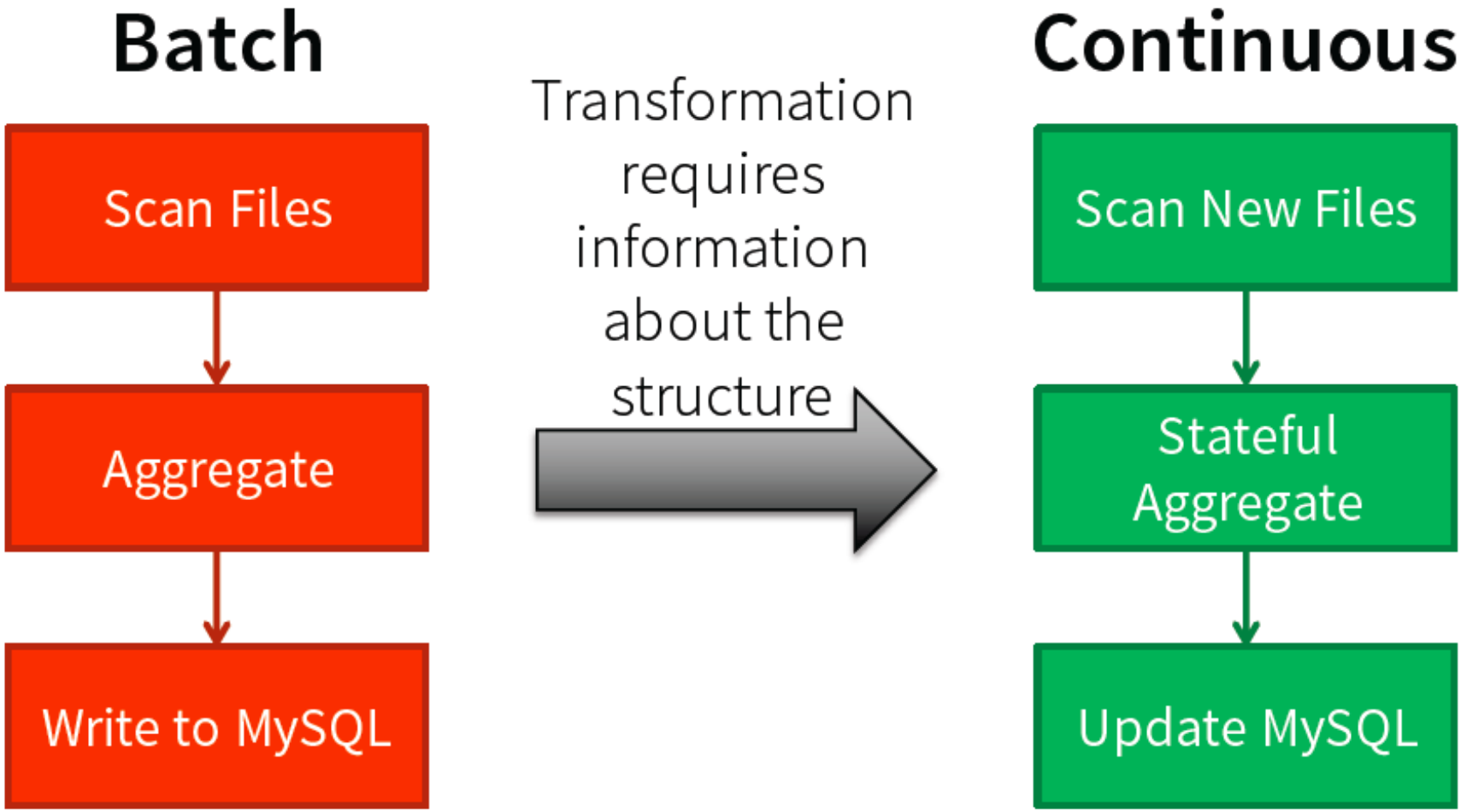
# Example: Continuous Aggregation

```
logs  = ctx.read.format("json").stream("s3://logs")

logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .stream("jdbc:mysql//...")
```

# Automatic Incremental Execution



# Incrementalized by Spark



# Rest of Spark will follow

- Interactive queries should just work
- Spark's data source API will be updated to support seamless streaming integration
  - Exactly once semantics end-to-end
  - Different Output modes (complete, delta, update-in-place)
- Machine Learning algorithms will be updated according to this new model

# Going Forward: What to expect from Spark ?

- Spark 2.0
  - Unification of the APIs
  - Basic Streaming API
  - Event-time Aggregations
- Spark 2.1+
  - Other Streaming sources/sinks
  - Machine Learning
- Structure in other libraries, e.g.
  - MLlib
  - GraphFrames