A Quick Introduction to Apache Spark

Ahmad Al-Shishtawy
ahmadas@kth.se

Outline

- The MapReduce programming model
- Limitations and solutions
- The Spark approach
- Spark SQL
- Spark Streaming
- Extra Topics

The MapReduce programming model

Motivation

Started at Google

- Computations are mostly straightforward …
- … but the input data is usually large and the computations have to be distributed
- It is hard to parallelize the computation, distribute the data, and handle failures

MapReduce

- MapReduce is a programming model
- Does not require experience with parallel and distributed systems to program and utilize their resources
- Many real world tasks are expressible in this model
- Inspired by the map and reduce primitives present in Lisp

It All Started with a Paper…

in 2004

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared

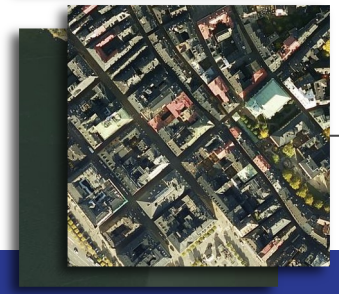
Count Buildings per City in Satellite Images

Map

('Stockholm', 5)
('Gothenburg', 7)
('Malmo', 3)
('Gothenburg', 2)
('Malmo', 6)
...

Map

('Stockholm', 6)
('Gothenburg', 3)
('Malmo', 3)
('Luleå', 5)
('Stockholm', 2)
('Stockholm', 8)
...



Map

('Stockholm', 9)
('Gothenburg', 5)
('Malmo', 8)
('Luleå', 4)
('Gothenburg', 4)
('Malmo', 2)
...

Shuffle

('Gothenburg', [7,2,3,5,4])
('Luleå', [5,4])
...

Reduce

('Gothenburg', 21)
('Luleå', 9)
...

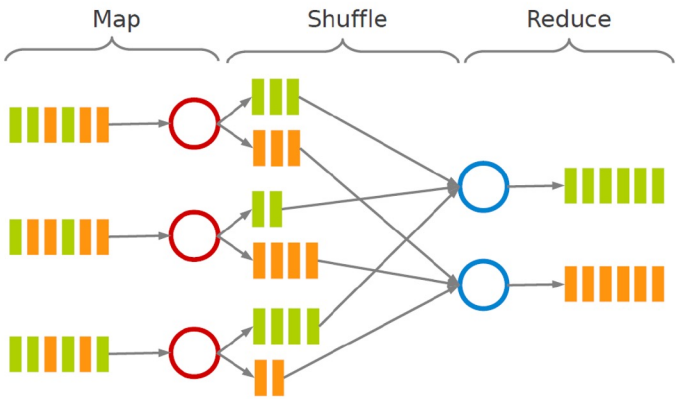
('Malmo', [3,6,3,8,2])
('Stockholm', [5,6,2,8,9])
...

Reduce

('Malmo', 22)
('Stockholm', 30)
...

MapReduce Programming Model

- Input file is divided into blocks and distributed on many machines (GFS & HDFS)
- Map: Process blocks in parallel and extract useful information (intermediate results)
  - No data movement between servers
  - Example: Number of buildings in a satellite image
- Shuffle: Group related data based on a user defined key
  - Data moves between servers but smaller than input
  - Example: All counts of buildings in Stockholm are grouped on the same server
- Reduce: Aggregates the grouped results using the user defined reduce function
  - Example: sum all counts of buildings in Stockholm



The Famous ~~HelloWorld~~ WordCount Example

- A simple application that counts the number of occurrences of each word in a given input file
- If the file fits on a single machine and in memory
  - No problem! We can use simple Linux commands
  - `cat doc.txt | tr ' ' '\n' | sort | uniq -c`
- If it doesn't fit??

WordCount in MapReduce

Consider doing a word count of the following example using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

The input to the map function is a key/value pair:

```
(1, ‘Hello World Bye World’)

(2, ‘Hello Hadoop Goodbye Hadoop’)
```

The Map Phase

- The map function reads a line and outputs (‘word’, 1) for each parsed input word.
- The map function is provided by the user.

The map function output is:

```
(‘Hello’, 1)
(‘World’, 1)
(‘Bye’, 1)
(‘World’, 1)
(‘Hello’, 1)
(‘Hadoop’, 1)
(‘Goodbye’, 1)
(‘Hadoop’, 1)
```

The Shuffle Phase

- The shuffle phase between the map and reduce phases creates a list of values associated with each key.
- The shuffle is done by the runtime system for you.

The result of the shuffle phase is:

```
('Bye', [1])
('Goodbye', [1])
('Hadoop', [1, 1])
('Hello', [1, 1])
('World', [1, 1])
```

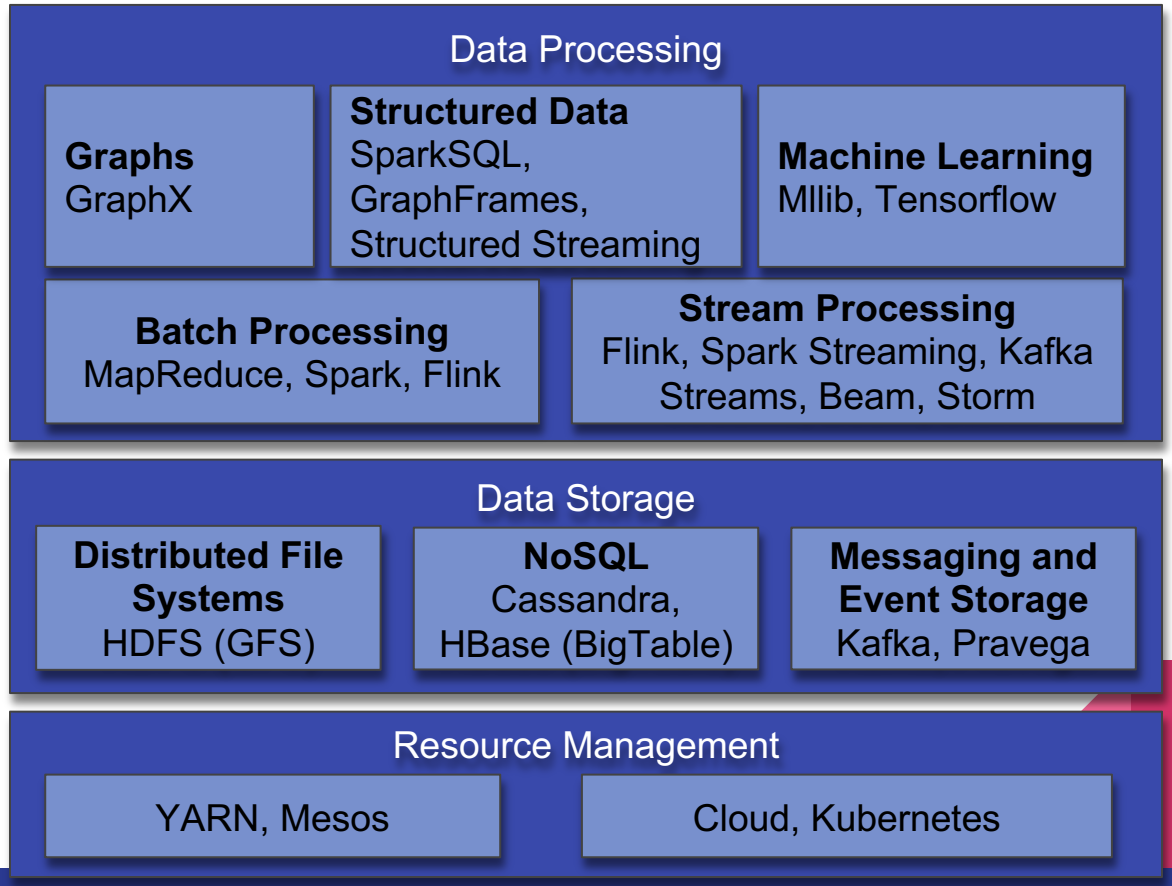
The Reduce Phase

- The reduce function sums the numbers in the list for each key and outputs (word, count) pairs.
- The reduce function is provided by the user.

The output of the reduce function is the output of the MapReduce job:

('Bye', 1)
('Goodbye', 1)
('Hadoop', 2)
('Hello', 2)
('World', 2)

Big Data Ecosystem (Simplified)



The Problem with MapReduce

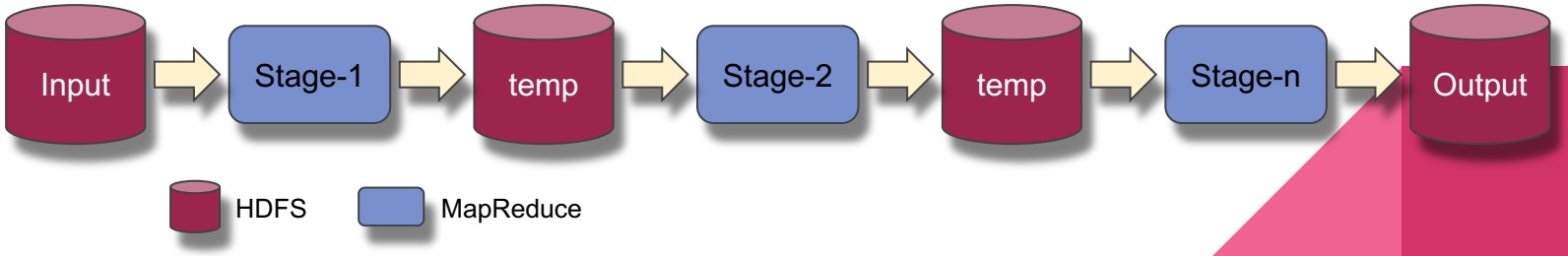
Limitations and solutions

MapReduce Limitations

- Pros:
  - **Simple abstraction:** Map to extract useful data and Reduce to aggregate and summarize
  - **Parallelism, robustness,** and **fault-tolerance** handled by the runtime
- Cons:
  - Powerful but low-level abstraction
  - Many real-world problems require multiple MapReduce stages
  - No automatic optimization
  - Hand optimizations makes it difficult to understand and extend
    - Require lots of experience
  - No separation between logical and physical plan

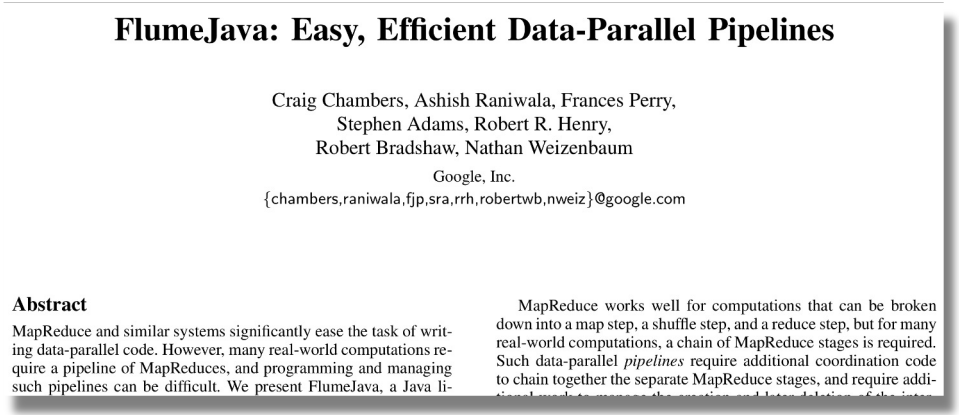
Chaining Multiple MapReduce Jobs

- Simple?
  - Map-1→ Reduce-1 → Map-2 → Reduce-2 → …
- Example: Top 10 words in a book longer that 5 characters?
  - Stage-1(word count) → Stage-2 (filter, rank, aggregate)
- Not if BigData + distributed system
  - Coordination code is needed
  - Syncing. Starting stage-2 after Stage-1 complete
  - Failures and Restarts
  - Intermediate “Big” results



Solutions?

- Many Platforms on top of MapReduce (Hadoop)
  - Solve and optimize specific problems
- Examples:
  - FlumeJava
  - Dryad



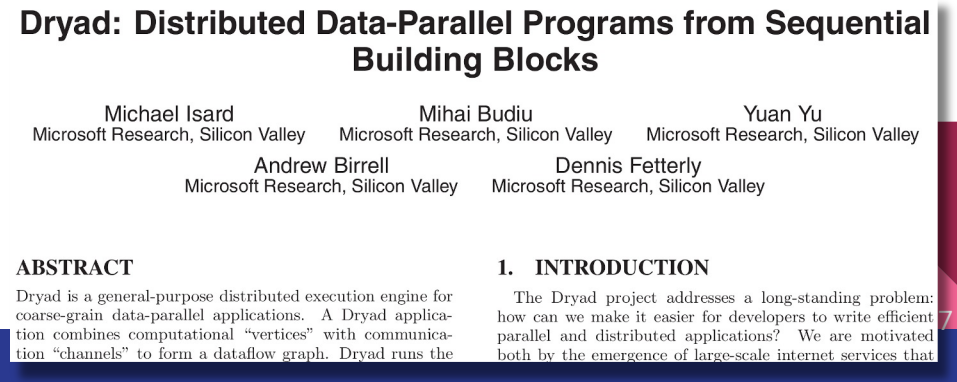
FlumeJava: Easy, Efficient Data-Parallel Pipelines

Craig Chambers, Ashish Raniwala, Frances Perry,
Stephen Adams, Robert R. Henry,
Robert Bradshaw, Nathan Weizenbaum
Google, Inc.
{chambers,raniwala,fjp,sra,rrh,robertwb,nweiz}@google.com

Abstract

MapReduce and similar systems significantly ease the task of writing data-parallel code. However, many real-world computations require a pipeline of MapReduces, and programming and managing such pipelines can be difficult. We present FlumeJava, a Java li-

MapReduce works well for computations that can be broken down into a map step, a shuffle step, and a reduce step, but for many real-world computations, a chain of MapReduce stages is required. Such data-parallel *pipelines* require additional coordination code to chain together the separate MapReduce stages, and require addi-



Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks

Michael Isard
Microsoft Research, Silicon Valley

Mihai Budiu
Microsoft Research, Silicon Valley

Yuan Yu
Microsoft Research, Silicon Valley

Andrew Birrell
Microsoft Research, Silicon Valley

Dennis Fetterly
Microsoft Research, Silicon Valley

ABSTRACT

Dryad is a general-purpose distributed execution engine for coarse-grain data-parallel applications. A Dryad application combines computational "vertices" with communication "channels" to form a dataflow graph. Dryad runs the

1. INTRODUCTION

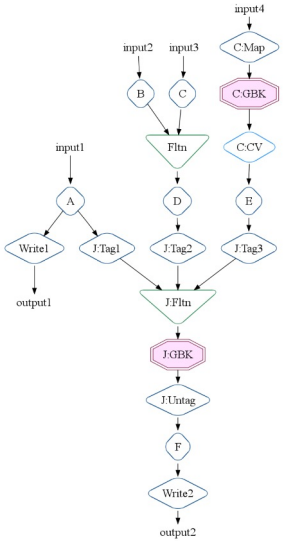
The Dryad project addresses a long-standing problem: how can we make it easier for developers to write efficient parallel and distributed applications? We are motivated both by the emergence of large-scale internet services that

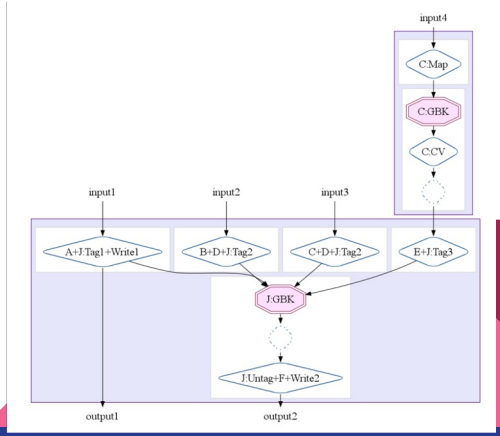
FlumeJava - Basic Concepts

- Library developed by Google to simply the creation of pipelined MapReduce jobs
- A few classes that represent parallel collections
  - abstract away details of where and how data is represented
  - `PCollection<T>`: immutable bag of elements of type T
  - `PTable<K, V>`: immutable multi-map with keys K and values V
- Data-parallel operation to manipulate (transform) parallel collections
  - `parallelDo()`: `PCollection<T> → PCollection<T>`
  - `groupByKey()`: `PTable<K, V> → PTable<K, Collection<V>>`
  - `combineValues()`: `PTable<K, Collection<V>> → PTable<K, V>`

FlumeJava - Execution

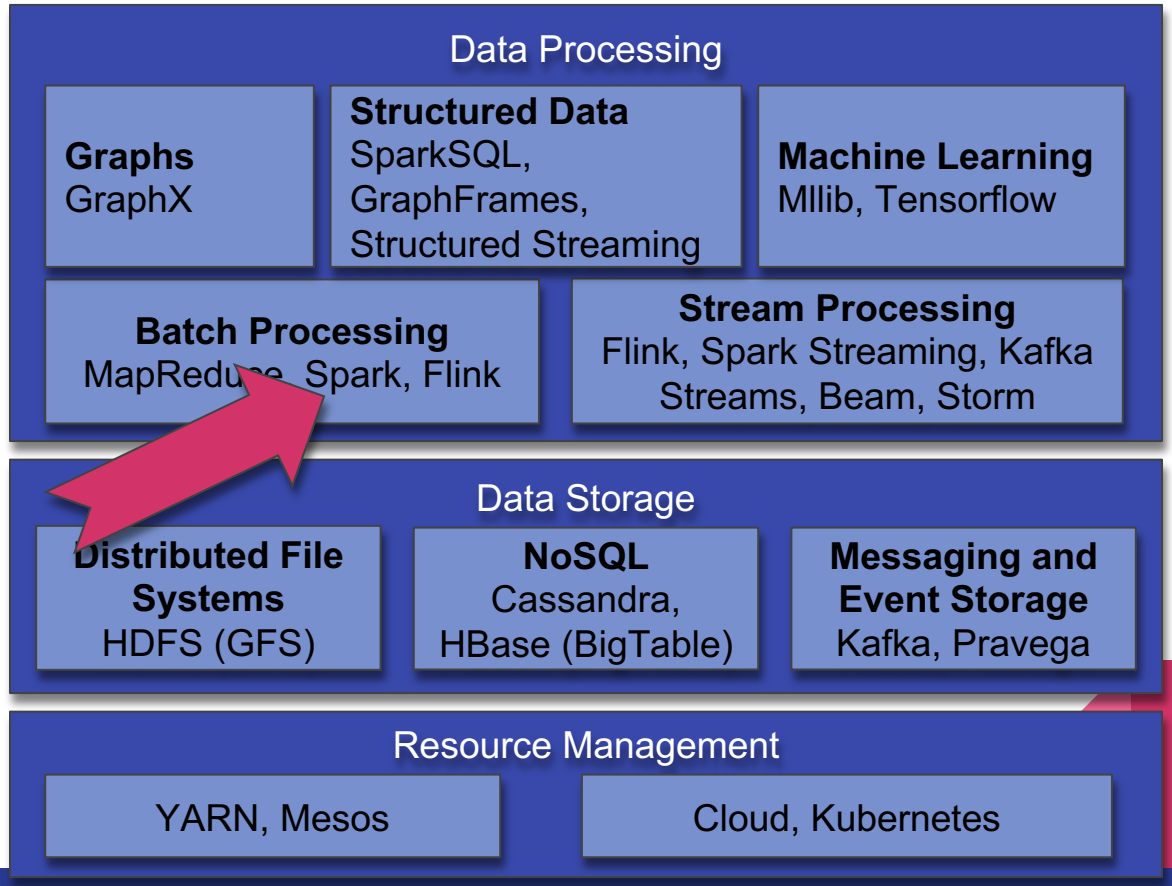
- Parallel operations are executed lazily
  - Materialized vs. deferred Parallel Collections
- You are building a “logical” execution plan
  - Directed acyclic graph of deferred Parallel Collections and Operations
- FlumeJava optimizes the execution plan before executing





Apache Spark

Big Data Ecosystem (Simplified)



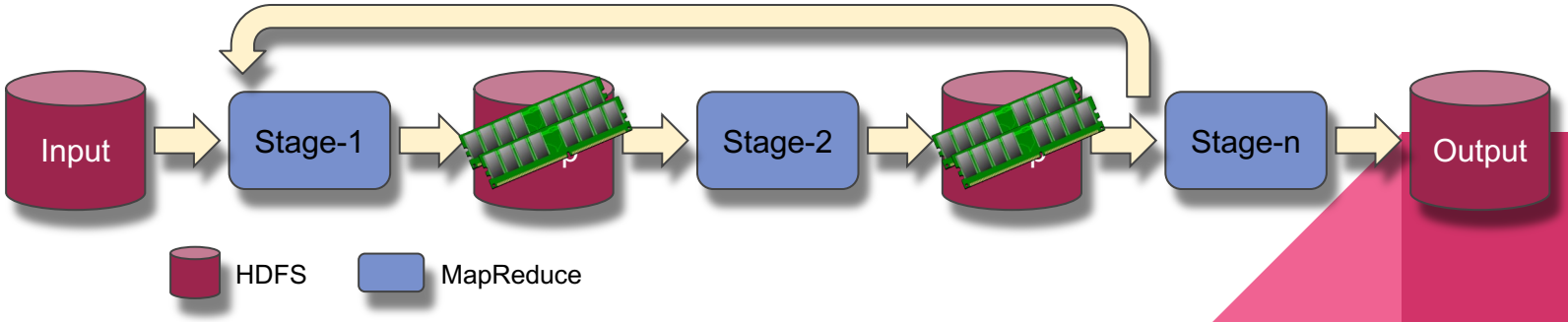
Spark Motivations

- Current platforms:
  - Provide abstractions for accessing a clusters computational resources
  - Reuse data between computations by writing it to disk
  - Lack abstractions for leveraging distributed memory

- Emerging applications
  - Reuse intermediate results across multiple computations
  - Iterative machine learning and graph algorithms
  - Interactive data mining and analytics

Spark Goals

- Extends MapReduce with more operators
- Support for advanced data flow graphs
- Fast and fault-tolerant in-memory processing



Challenge!

- Previous systems relied on the fault tolerance provided by the storage system
- How to design a distributed memory abstraction that is both fault tolerant and efficient?

Solution?

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

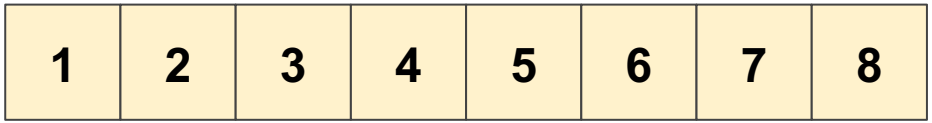
We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types

tion, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data

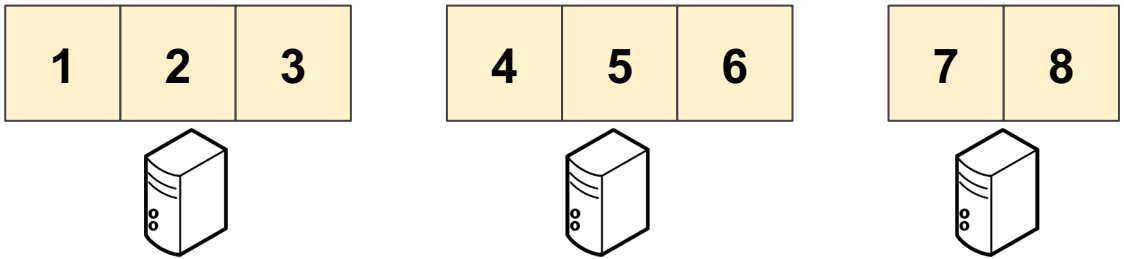
Resilient Distributed Datasets (RDD) (1/2)

- A distributed memory abstraction
- Immutable collections of objects spread across a cluster.
  - Think of: `myList = [1, 2, 3, 4, 5, 6, 7, 8]`
  - But very large and distributed!



Resilient Distributed Datasets (RDD) (2/2)

- An RDD is divided into a number of partitions, which are atomic pieces of information
- Partitions of an RDD are stored on different nodes



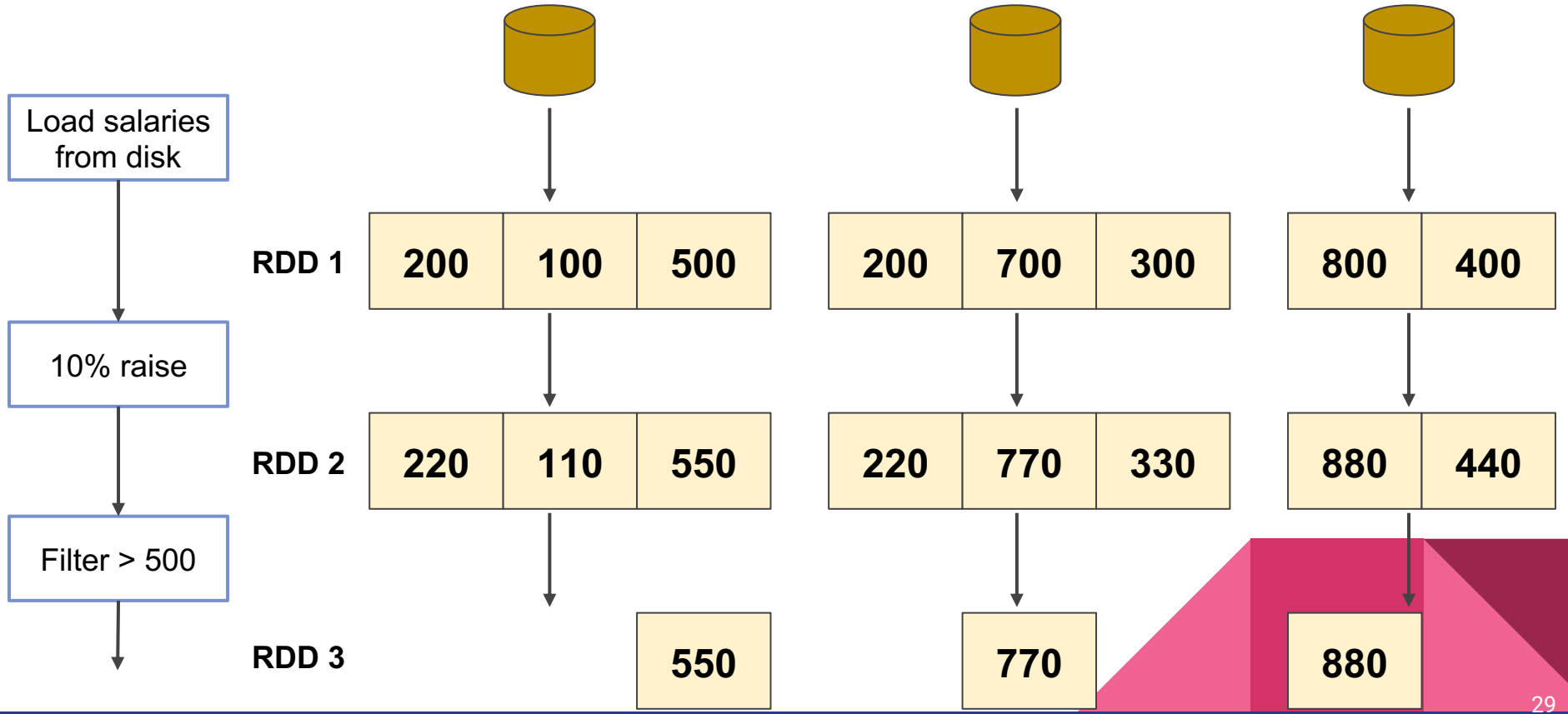
Why a New Abstraction?

- Existing abstractions examples
  - key-value stores
  - distributed shared memory
  - …
- Offer interface based on fine-grained updates to mutable state (e.g., cells in a table)
- Provide fault tolerance by replicating the data or logging updates across machines
- Expensive for data-intensive workloads
  - Require copying large amounts of data

RDDs Approach

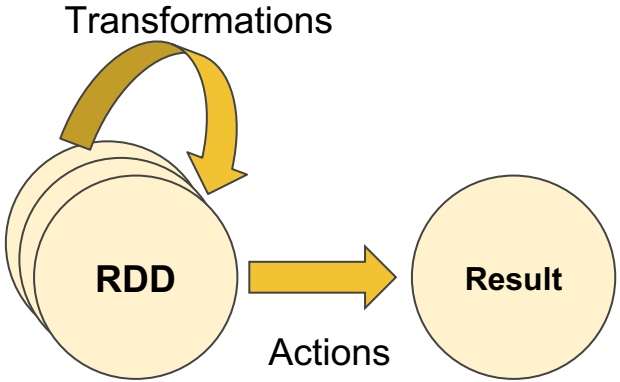
- Coarse-grained transformations
  - Apply the same operation to all data items
  - e.g., map, filter, join
- Efficiently provide fault tolerance
  - By logging the transformations used to build a dataset
  - Called its **lineage**
  - No replication of data in memory is required
- If an RDD partition (in memory) is lost
  - Trace back in its lineage to a parent partition (or data on disk)
  - Reapply the transformations to reconstruct the partition

Example for Transformations and Lineage



RDD Operations

- RDD operations can be either be **actions** or **transformations**
- **Transformations:**
  - **Lazy** evaluation (does not trigger computation)
  - Used to “describe” the program
  - Constructs in a DAG (directed acyclic graph) → Lineage
  - Each transformation produces a new RDD
- **Actions:**
  - Produce output (disk, on screen, or local memory)
  - Triggers computation to get the required output
  - Get data out of Spark



RDD Applications

- Applications suitable for RDDs
  - Data parallel applications
  - Apply the same operation to all elements of a dataset

- Applications not suitable for RDDs
  - Applications that make asynchronous fine-grained updates to shared

SparkContext

- Main entry point to Spark functionality
- A SparkContext represents the connection to a Spark cluster
- Can be created as following:

```
# master: the master URL to connect to, e.g.,
# "local", "local[4]", "spark://master:7077"
import pyspark
sc = pyspark.SparkContext('local[*]', 'MyApp')

# new API
from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local[*]').appName('MyApp').getOrCreate()
sc = spark.sparkContext
```

Creating RDDs

- Turn a collection into an RDD

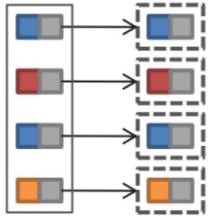
```
# Create an RDD from Python list
nums = sc.parallelize([1, 2, 3, 4, 5, 6])
```

- Load text file from local FS, HDFS, or S3

```
lines = sc.textFile('words.txt')
lines = sc.textFile('directory/*.txt')
lines = sc.textFile('hdfs://namenode:9000/path/file')
```

RDD Transformations - Map

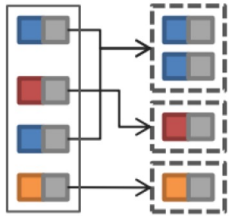
- All pairs are independently processed



```
# Map: Apply a function to all elements of an RDD
squares = nums.map(lambda x: x**2)  # [1, 4, 9, 16, 25, 36]

# Filter using a boolean function
even = nums.filter(lambda x: x % 2 == 0)  # [2, 4, 6]

# FlatMap generates zero or more elements for each input
many_nums = nums.flatMap(lambda x: list(range(0, x)))
# [0, 0, 1, 0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 5]

# Map is one to one
lists = nums.map(lambda x: list(range(0, x)))
# [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4, 5]]
```

RDD Transformations - Reduce

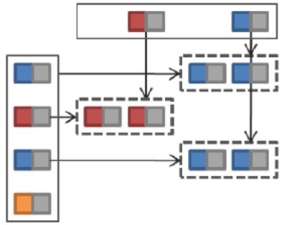
- Pairs with identical key are grouped
- Groups are independently processed



```
pets = sc.parallelize([('cat',1), ('dog',2), ('cat',3)])

pets.groupByKey().mapValues(list)
# [('cat', [1, 3]), ('dog', [2])]

pets.reduceByKey(lambda x, y: x+y)
# [('cat', 4), ('dog', 2)]

# or use Python operator module
from operator import add
pets.reduceByKey(add)
```

RDD Transformations - Join

- Return an RDD containing all pairs of elements with matching keys
- Only keys existing in both RDDs are returned



```
visits = sc.parallelize([('index.html','1.2.3.4'), \
                         ('about.html','3.2.3.4'), \
                         ('index.html','5.4.3.2'), \
                         ('help.html','7.6.1.2')])

pageName = sc.parallelize([('index.html','Home'), \
                           ('about.html','About'), \
                           ('prod.html','Products')])

visits.join(pageName)
# [('about.html', ('3.2.3.4', 'About')),
#  ('index.html', ('1.2.3.4', 'Home')),
#  ('index.html', ('5.4.3.2', 'Home'))]
```

Basic RDD Actions (1/2)

- Return all the elements of the RDD as an array

```
nums.collect()  # Out: [1, 2, 3, 4, 5, 6]
```

- Return an array with the first n elements of the RDD

```
nums.take(2)  # Out: [1, 2]
```

- Return the number of elements in the RDD

```
nums.count()  # Out: 6
```

- Lookup a value by key

```
pets.lookup('cat')  # Out: [1, 3]
```

Basic RDD Actions (2/2)

- Aggregate the elements of the RDD using the given function

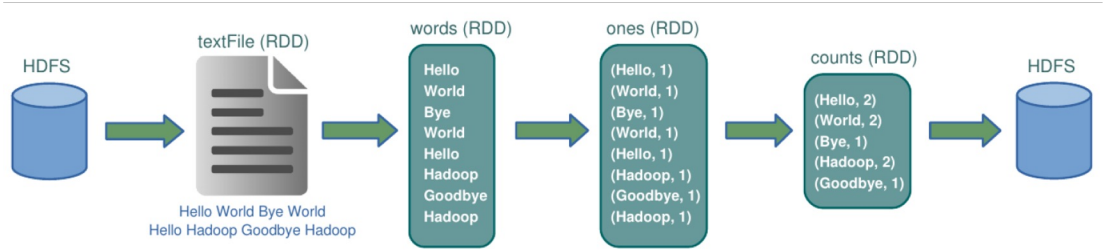
```
nums.reduce(lambda x,y: x+y)  # Out: 21
# or
from operator import add
nums.reduce(add)  # Out: 21
```

- Write the elements of the RDD as a text file

```
nums.saveAsTextFile('nums.txt')
```

Example: WordCount

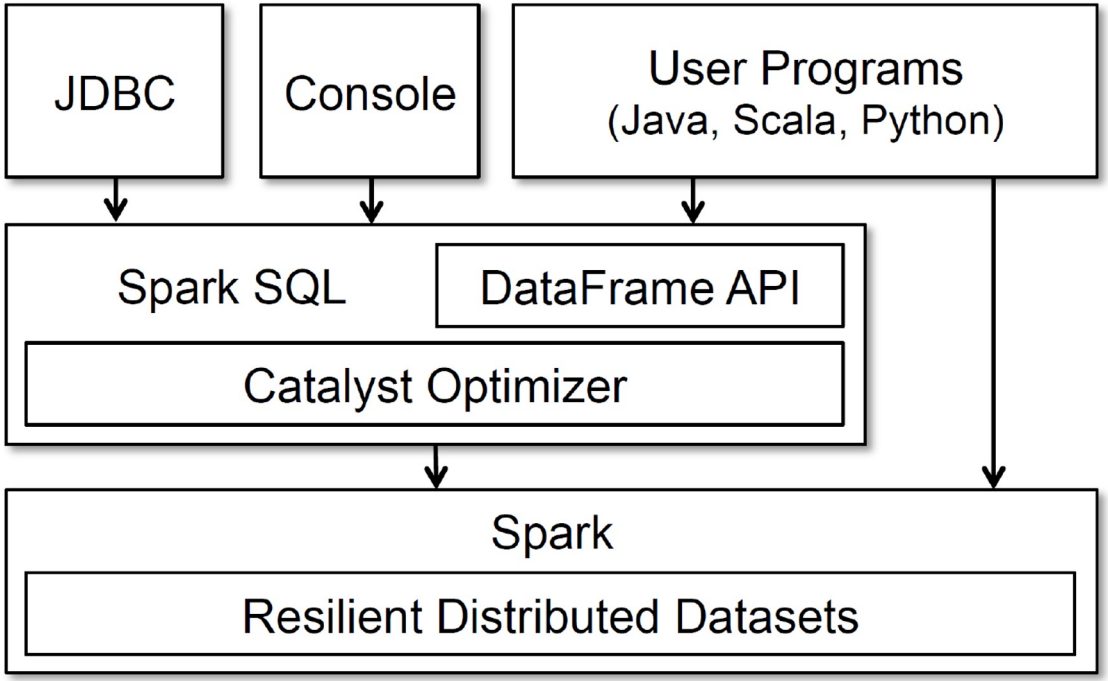
```
textFile = sc.textFile('hdfs://.../words.txt')
words = textFile.flatMap(lambda x: x.split(' '))
ones = words.map(lambda x: (x, 1))
counts = ones.reduceByKey(lambda x,y:x+y)

counts.saveAsTextFile("hdfs://...")
# or
counts.collect()
# Out: [('Hello', 2), ('World', 2), ('Goodbye', 1), ('Hadoop', 2), ('Bye', 1)]
```



Lineage: transformations used to build an RDD

Spark SQL

Spark vs. Spark SQL



Goals

- Support relational processing both within Spark programs (on native RDDs) and on external data sources
- Provide high performance using established DBMS techniques.
- Easily support new data sources, including semi-structured data and external databases
- Enable extension with advanced analytics algorithms such as graph processing and machine learning

DataFrame (1/2)

- A DataFrame is a distributed collection of rows with a homogeneous schema
  - Data is organized into named columns
  - Equivalent to a table in a relational database
- It can be manipulated:
  - With a relational API (SELECT, WHERE, ...)
  - With a procedural API (Map, Filter, ...) in a similar ways to RDDs (can be viewed as an RDD of Row objects)
- DataFrames are evaluated lazily
  - Represents a logical plan to compute a dataset
- DataFrames keep track of their schema

DataFrame (2/2)

- Supports all major SQL data types
  - boolean, integer, double, decimal, string, date, and, timestamp.
  - complex data types: structs, arrays, maps, and unions.
- Support various relational operations that lead to more optimized execution
  - Similar DataFremes in R and Python Pandas
  - All common relational operators, including projection (select), filter (where), join, and aggregations (groupBy)
  - Operators take expression in a limited domain-specific language (DSL) that lets Spark capture the structure of the expression. (col('Age') > 20)
- DataFrames can be constructed:
  - from tables in a system catalog (based on external data sources)
  - from existing RDDs of native Java/Python objects

Starting Point: SparkSession

- Spark <2.0
  - Multiple entry points for different APIs
  - SparkContext, SQLContext, HiveContext
- Spark 2.0+
  - Single entry point into all functionality in Spark
  - SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .master('local[*]') \
    .appName('My Data Intensive App') \
    .config('spark.driver.memory', '5g') \
    .getOrCreate()

# optional: .config('key', 'value')
# optional .enableHiveSupport()
```

Main SparkSession Usage

- Create DataFrame
- Read data from file (csv, json, parquet, jdbc, ...)
- Execute sql queries
- Get the underlying sparkContext
- Stop the underlying context

Creating DataFrames (1/3)

- From Java/Python collections

```
l = [('Alice', 1)]
spark.createDataFrame(l, ['name', 'age']).collect()
# [Row(name='Alice', age=1)]

d = [{'name': 'Alice', 'age': 1}]
spark.createDataFrame(d).collect()
# [Row(age=1, name='Alice')]

pandas_df = df.toPandas()
spark.createDataFrame(pandas_df).collect()
# [Row(name='Alice', age=1)]
```

Creating DataFrames (2/3)

- From an existing RDD
- Java/Scala: can automatically infer the schema using reflection
- Python:
  - Samples the dataset to perform schema inference due to the dynamic type system
  - Or you can explicitly specify the schema

```
l = [('Alice', 1)]
rdd = spark.sparkContext.parallelize(l)
spark.createDataFrame(rdd, ['name', 'age']).collect()
# [Row(name='Alice', age=1)]

schema = StructType([
       StructField('name', StringType(), True),
       StructField('age', IntegerType(), True)])
spark.createDataFrame(rdd, schema).collect()
# [Row(name='Alice', age=1)]
```

Creating DataFrames (3/3)

- By reading a file

```
# spark is an existing SparkSession
df = spark.read.json('people.json')
# Displays the content of the DataFrame to stdout
df.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

DataFrame Operations (1/3)

```
# Print the schema in a tree format
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)

# Select only the 'name' column
df.select('name').show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+
```

DataFrame Operations (2/3)

```
# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
# +-------+---------+
# |   name|(age + 1)|
# +-------+---------+
# |Michael|     null|
# |   Andy|       31|
# | Justin|       20|
# +-------+---------+

# Select people older than 21
df.filter(df['age'] > 21).show()
# +---+----+
# |age|name|
# +---+----+
# | 30|Andy|
# +---+----+
```

DataFrame Operations (3/3)

```
# Count people by age
df.groupBy('age').count().show()
# +----+-----+
# | age|count|
# +----+-----+
# |  19|    1|
# |null|    1|
# |  30|    1|
# +----+-----+
```

SQL Queries

- Running SQL queries programmatically and returns the result as a DataFrame
- Using the sql function on a SparkSession

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView('people')

sqlDF = spark.sql('SELECT * FROM people')
sqlDF.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

Interoperating with RDDs (1/2)

```
from pyspark.sql import Row

sc = spark.sparkContext

# Load a text file and convert each line to a Row.
lines = sc.textFile('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 DataFrame as a table.
schemaPeople = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView('people')
```

Interoperating with RDDs (2/2)

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

# The results of SQL queries are Dataframe objects.
# rdd returns the content as an :class:`pyspark.RDD` of :class:`Row`.
teenNames = teenagers.rdd.map(lambda p: 'Name: ' + p.name).collect()
for name in teenNames:
    print(name)
# Name: Justin
```

Load/Save Data

- Built in data sources: json, parquet, jdbc, orc, libsvm, csv, text
- Parquet is a columnar format that is supported by many other data processing systems

```
df = spark.read.load('people.json', format='json')
teenagers = df.select('name', 'age').where('age >= 13 AND age <= 19')
teenagers.write.save('namesAndAges.parquet', format='parquet')
```

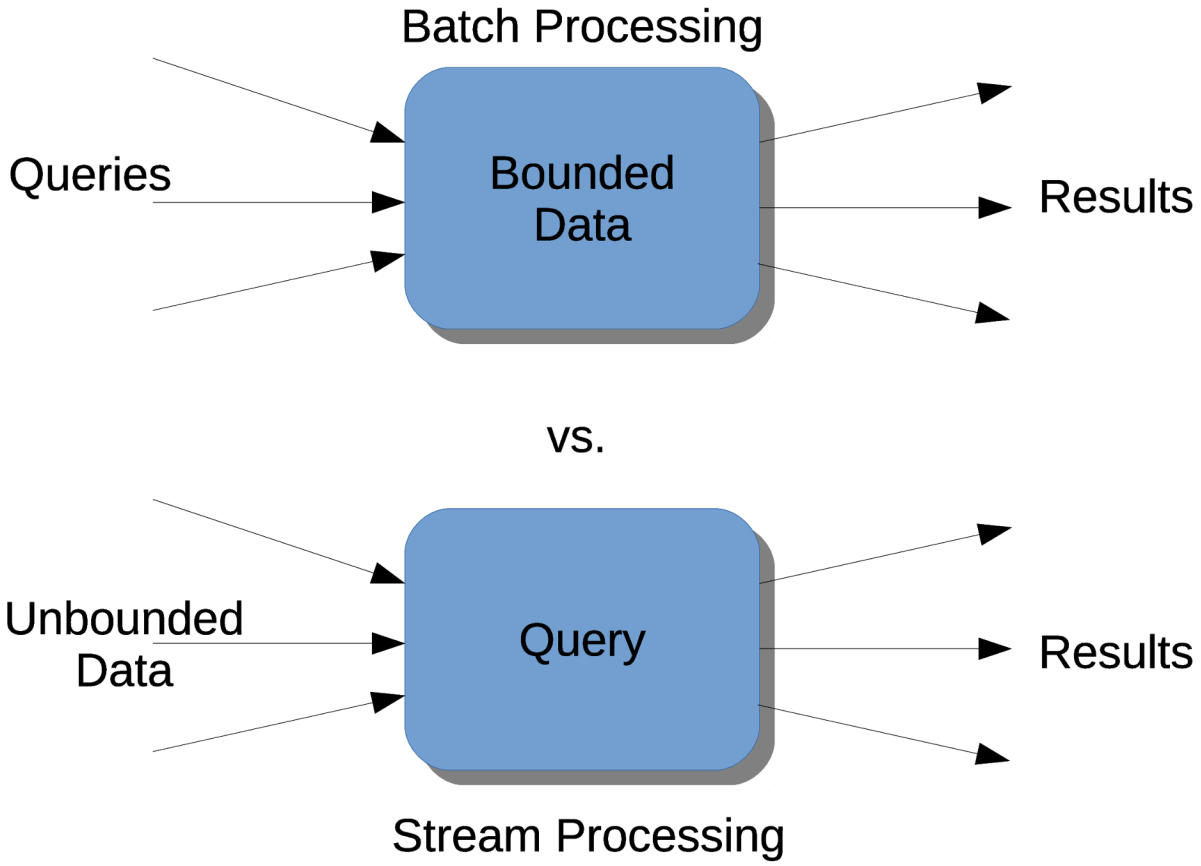
Stream Processing

Basic Concepts

- Bounded Data (data-at-rest): collected, stored, and indexed before processing
- Unbounded Data (data-in-motion): no clear beginning or end, new data added continuously
- Both batch and streaming systems are capable of processing both bounded and unbounded data

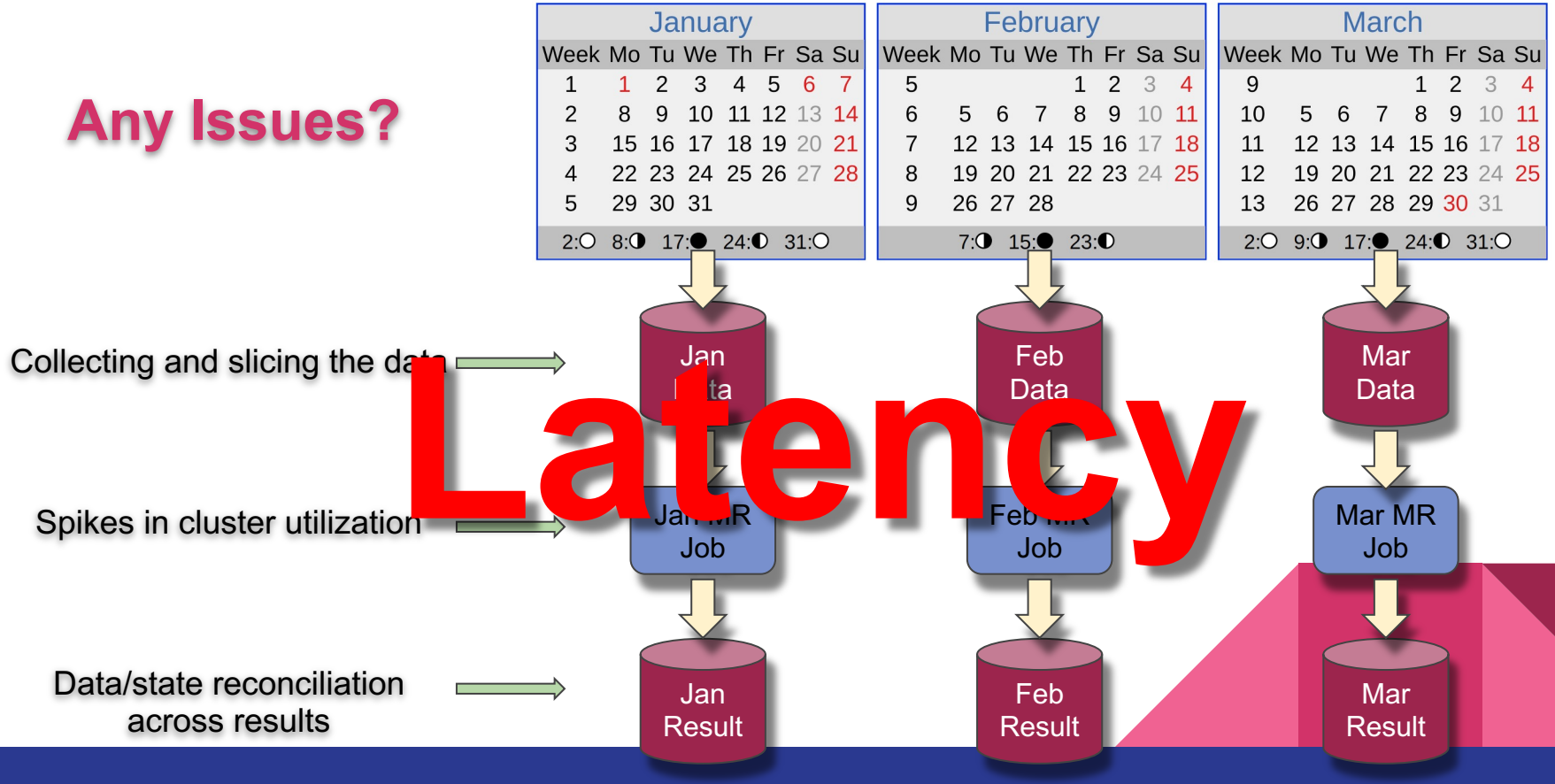
  ... but there are some differences and challenges that we have to address

A Different Way of Thinking!



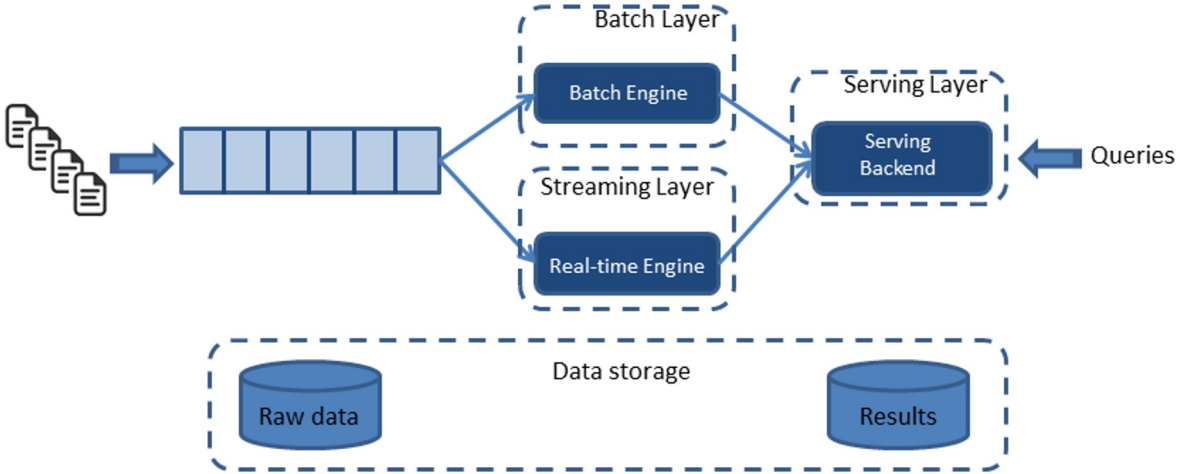
Batch System and Unbounded Data!

Any Issues?



Lambda Architecture

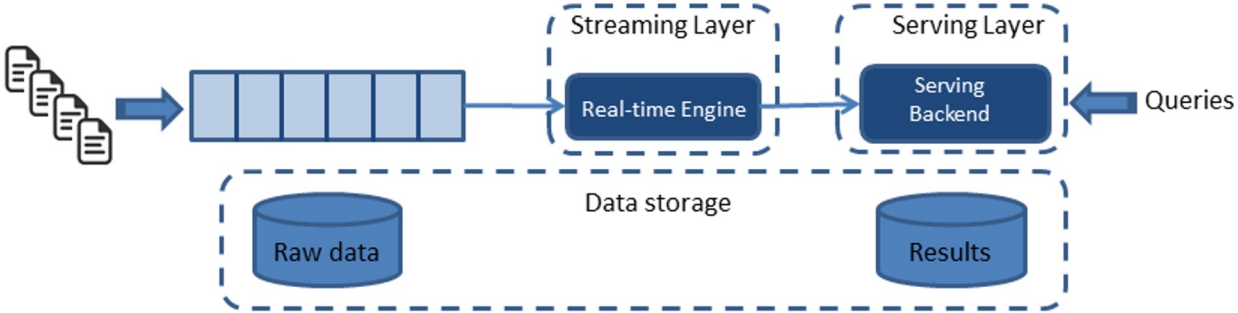
- Use streaming layer to get fast but approximate results
- Use batch layer to compute the final correct result



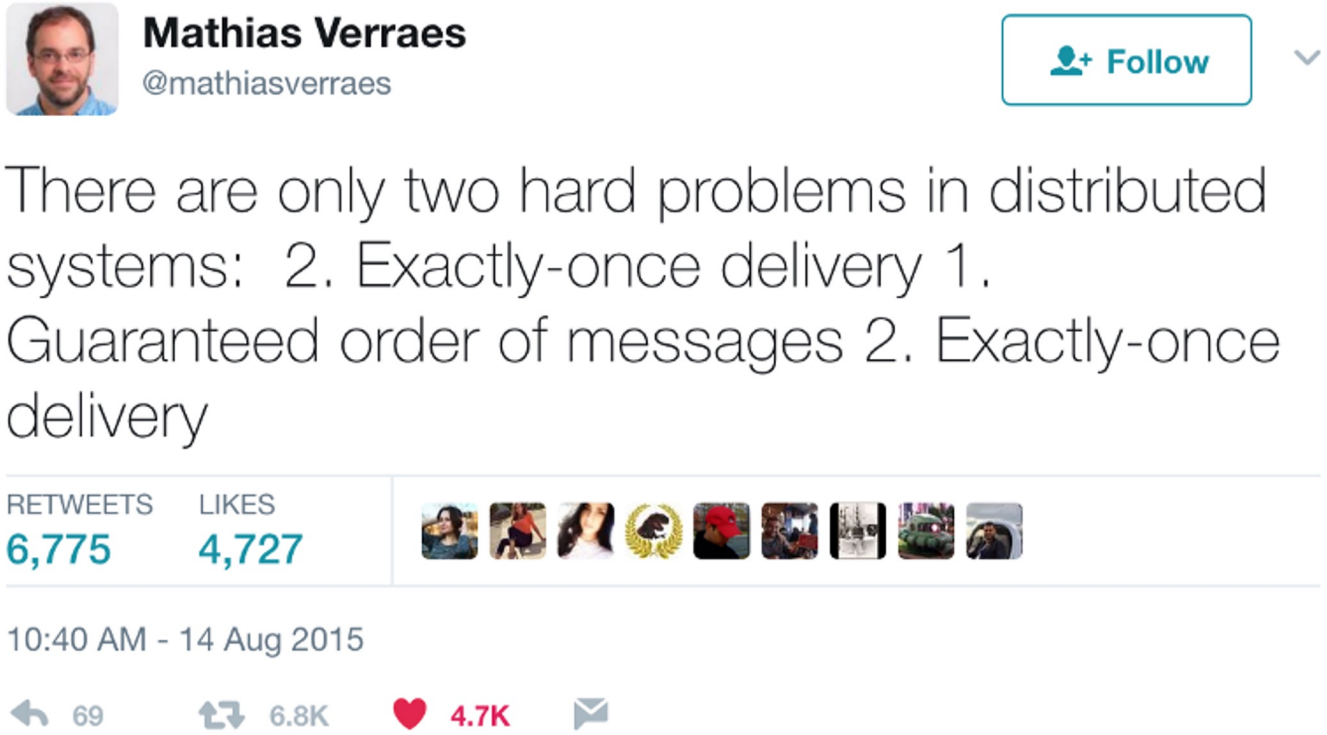
Kappa Architecture

- Everything is a stream
- Use only stream processing

- … but what changed? why now?



Streaming is Hard



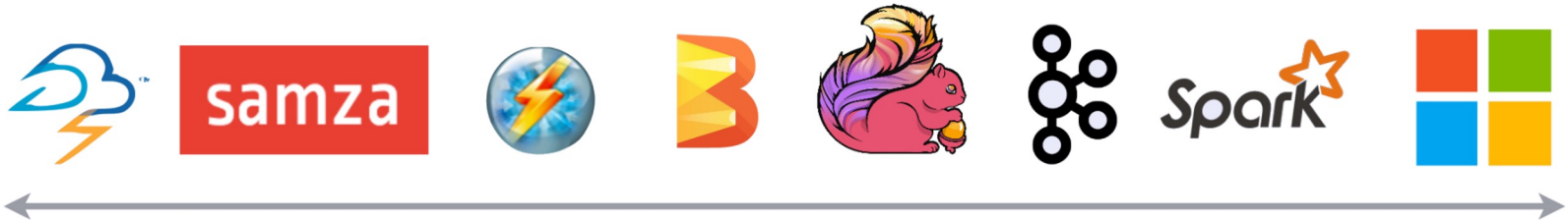
Event Processing Guarantees

- **At-least-once:** all events are processed, but some might be processed more than once
- **At-most-once:** some events might be lost and never processed
- **Exactly-once:** Events are never lost and processed only once

Programming Level

Compositional (how)

Declarative (what)



- Physical Representations
- Offer basic building blocks(Operators/Data Exchange)
- Custom Optimization/Tuning
- Direct access to the execution graph / topology
- Dataflow Programming

- Logical Representations
- Operators are transformations on abstract data types
- Advanced behavior such as windowing is supported
- Self-Optimization
- Meta-programming

The Evolution of Stream Processing Systems

- Early systems exposed compositional API with at-least-once semantics
- Current systems provide high level declarative API with exactly-once semantics
- Streaming systems are moving towards
  - End-to-end exactly one
  - Structured data
  - SQL support

Batch or Streaming

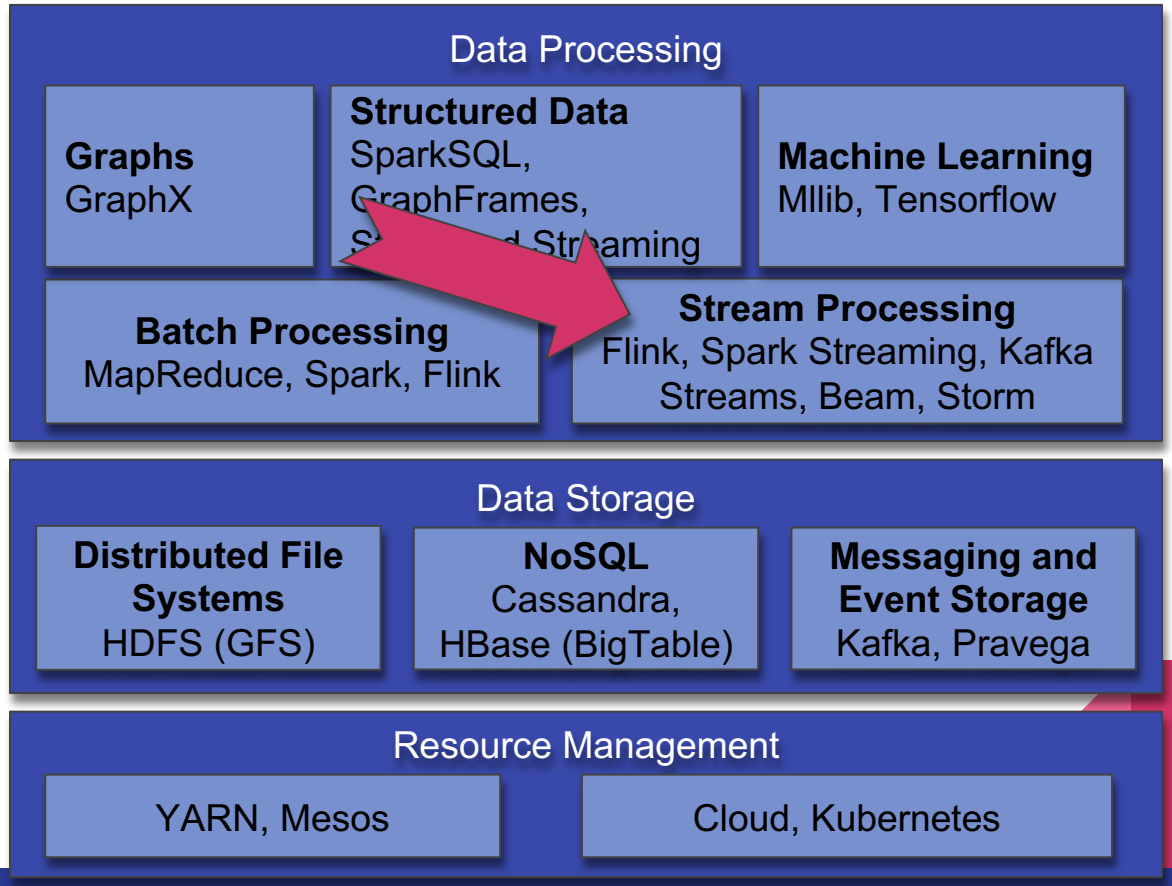
“What changes faster? Your code or your data?

Data: You got a data stream processing problem,

Code: You got an exploration problem. Eventually it will become a stream processing problem.”

- Joe Hellerstein

Big Data Ecosystem (Simplified)



Spark Streaming

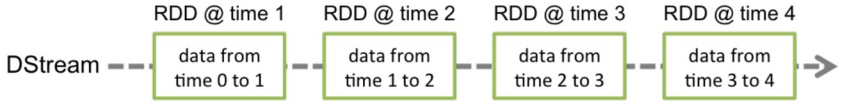
Spark Streaming

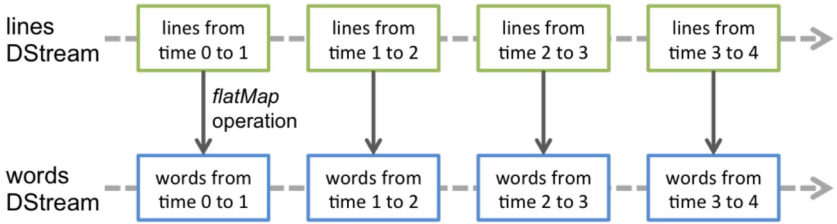
Run a streaming computation as a series of very small, deterministic batch jobs

- 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 (DStream)

DStream

- **DStream:** sequence of RDDs representing a stream of data
- Any operation applied on a DStream translates to operations on the underlying RDDs



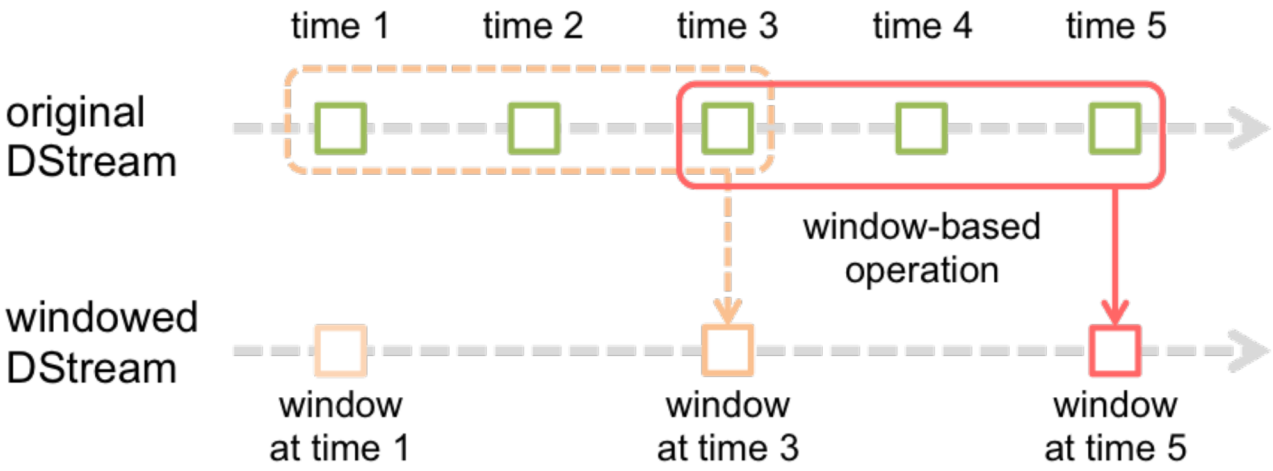


DStream Transformations

- Transformations: modify data from one DStream to a new DStream
- Standard RDD operations, e.g., map, join, …
- DStream operations, e.g., window operations

Window Operations

- Apply transformations over a sliding window of data
  - Window length and slide interval



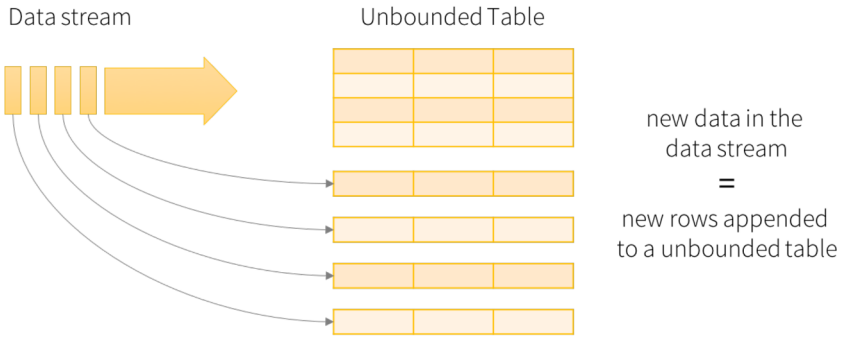
Spark Structured Streaming

Structured Streaming

- Structured streaming is a new high-level API to support continuous applications
- A higher-level API than Spark streaming
- Built on the Spark SQL engine
- Perform database-like query optimizations

Stream as an Unbounded Table

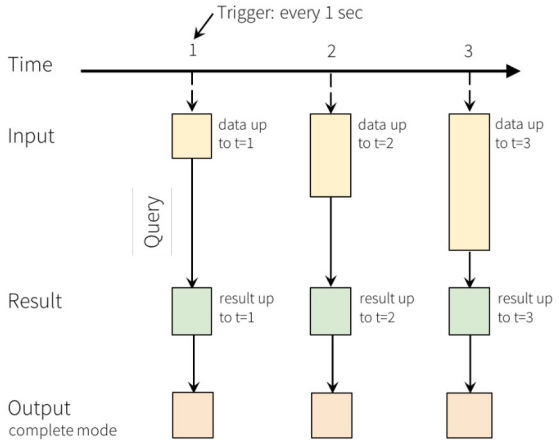
- Treating a live data stream as a table that is being continuously appended
- Users can express their streaming computation as standard batch-like query as on a static table
- Spark runs it as an incremental query on the unbounded input table



Data stream as an unbounded table

Programming Model

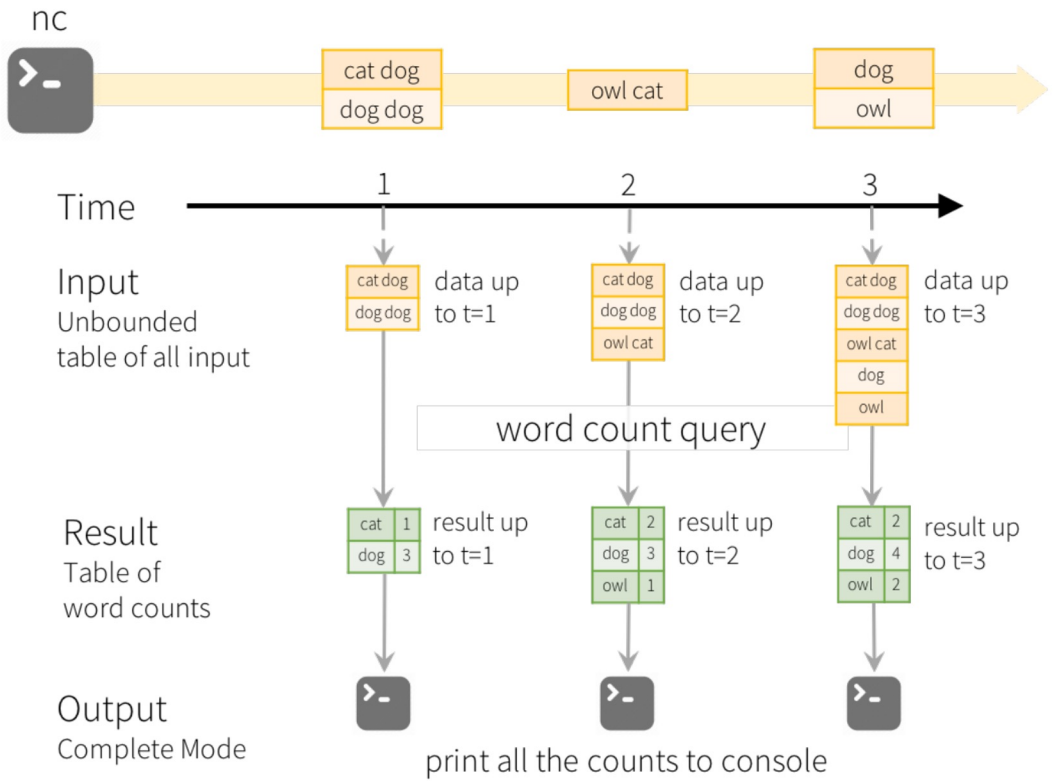
- A query on the input will generate the Result Table
- Every trigger interval (e.g., every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table
- Whenever the result table gets updated, we can write the changed result rows to an external sink



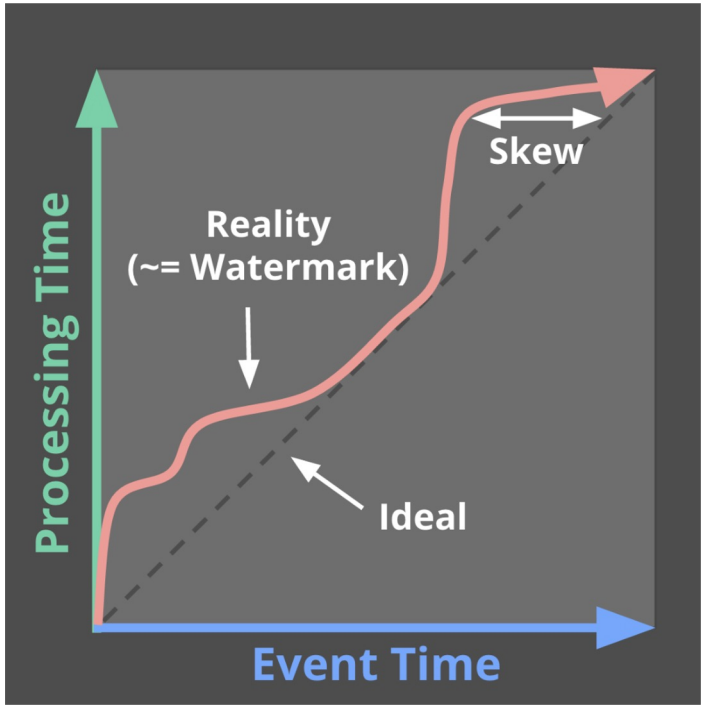
Example (1/2)

```
spark = SparkSession.builder.appName("StructuredNetworkWordCount").getOrCreate()

# Create DataFrame representing the stream of input lines from connection to localhost:999
lines = spark.readStream.format("socket").option("host", "localhost") \
    .option("port", 9999).load()

# Split the lines into words
words = lines.select(explode(split(lines.value, ' ')).alias('word'))

# Generate running word count
wordCounts = words.groupBy('word').count()

# Start running the query that prints the running counts to the console
query = wordCounts.writeStream.outputMode("complete").format("console").start()
query.awaitTermination()
```

Example (2/2)


nc
cat dog
dog dog
owl cat
dog
owl
Time
1
2
3
Input
Unbounded
table of all input
cat dog
dog dog
data up
to t=1
cat dog
dog dog
owl cat
data up
to t=2
cat dog
dog dog
owl cat
dog
owl
data up
to t=3
word count query
Result
Table of
word counts
cat 1
dog 3
result up
to t=1
cat 2
dog 3
owl 1
result up
to t=2
cat 2
dog 4
owl 2
result up
to t=3
Output
Complete Mode
print all the counts to console

Processing Time vs. Event Time

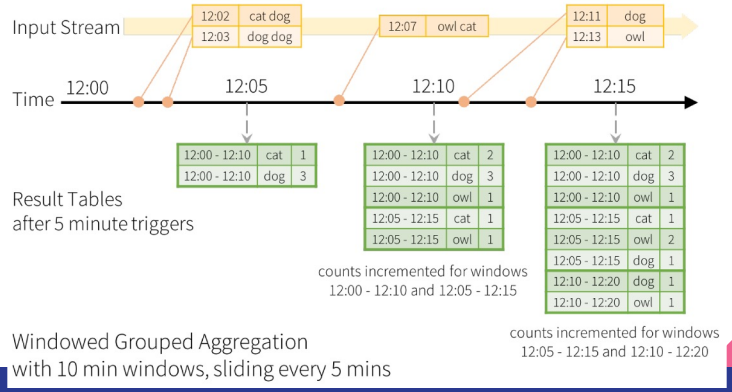


https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

Event Time

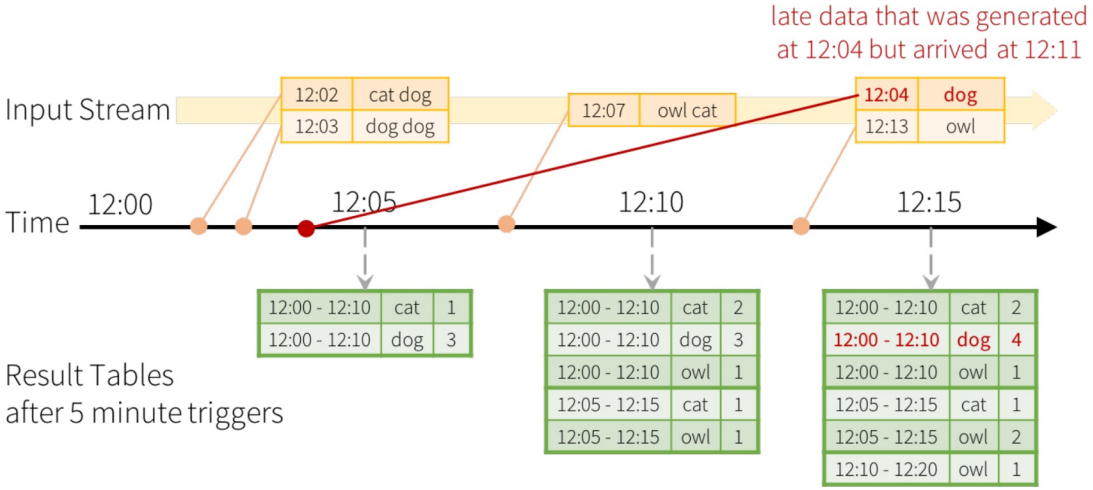
- Aggregations over a sliding event-time window
- Event-time is the time embedded in the data itself, not the time Spark receives them

```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }

# Group the data by window and word and compute the count of each group
windowedCounts = words.groupBy(
    window(words.timestamp, "10 minutes", "5 minutes"),
    words.word
).count()
```



Late Data

- Aggregations over a sliding event-time window
- Event-time is the time embedded in the data itself, not the time Spark receives them

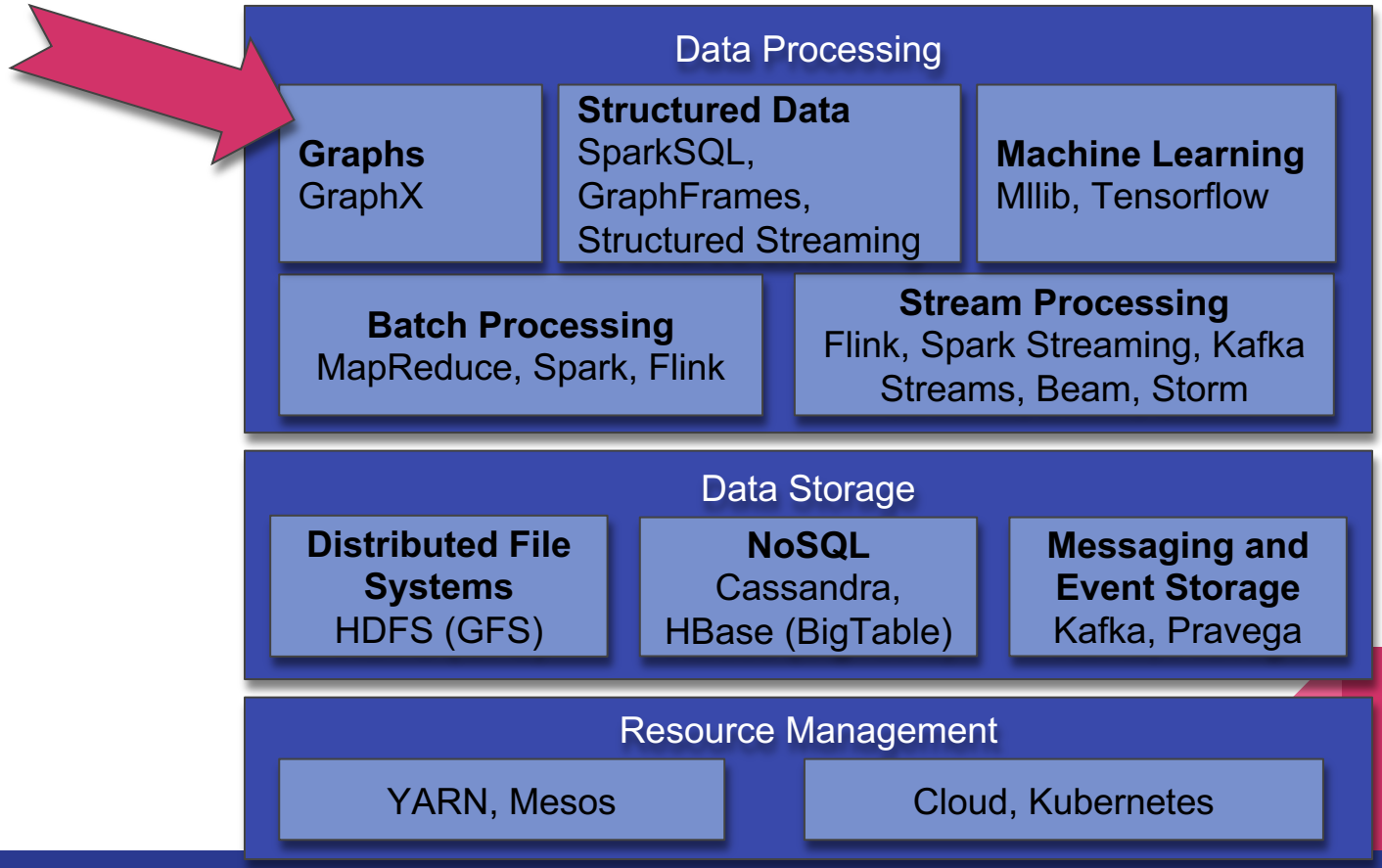


Late data handling in Windowed Grouped Aggregation

Extra Topics...

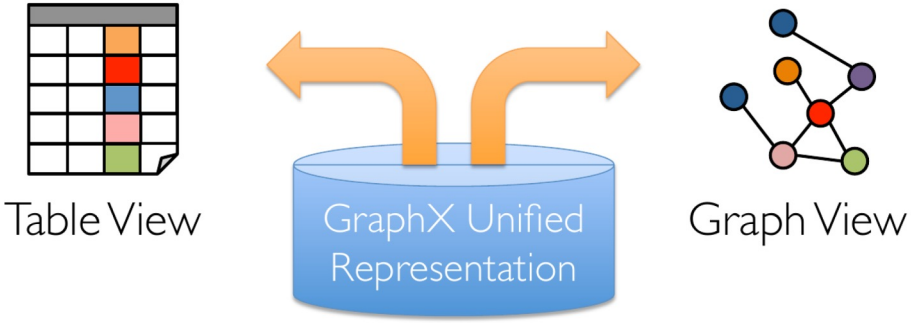
GraphX

Big Data Ecosystem (Simplified)



GraphX

- Unites data-parallel and graph-parallel systems
- Tables and Graphs are composable views of the same physical data
- Implemented on top of Spark



Programming Model

- Gather-Apply-Scatter (GAS)
- Input data (Property Graph): represented using two Spark RDDs:
  - Edge collection: VertexRDD
  - Vertex collection: EdgeRDD

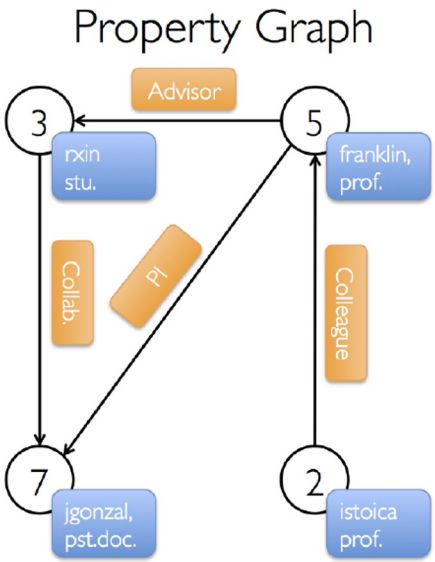
```
// VD: the type of the vertex attribute
// ED: the type of the edge attribute
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

Execution Model

GAS decomposition:

- Gather: the groupBy stage gathers messages destined to the same vertex
- Apply: an intervening map operation applies the message sum to update the vertex property
- Scatter: the join stage scatters the new vertex property to all adjacent vertices

Example (1/3)



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Example (2/3)

```
val sc: SparkContext

// Create an RDD for the vertices
val users: VertexRDD[(String, String)] = sc.parallelize(
    Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
          (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// Create an RDD for edges
val relationships: EdgeRDD[String] = sc.parallelize(
    Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
          Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))

// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")

// Build the initial Graph
val userGraph: Graph[(String, String), String] =
    Graph(users, relationships, defaultUser)
```

Example (3/3)

```
// Constructed from above
val userGraph: Graph[(String, String), String]

// Count all users which are postdocs
userGraph.vertices.filter((id, (name, pos)) => pos == "postdoc").count

// Count all the edges where src > dst
userGraph.edges.filter(e => e.srcId > e.dstId).count

// Use the triplets view to create an RDD of facts
val facts: RDD[String] = graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " +
    triplet.attr + " of " + triplet.dstAttr._1)

// Remove missing vertices as well as the edges to connected to them
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")

facts.collect.foreach(println(_))
```

GraphFrames

GraphFrames

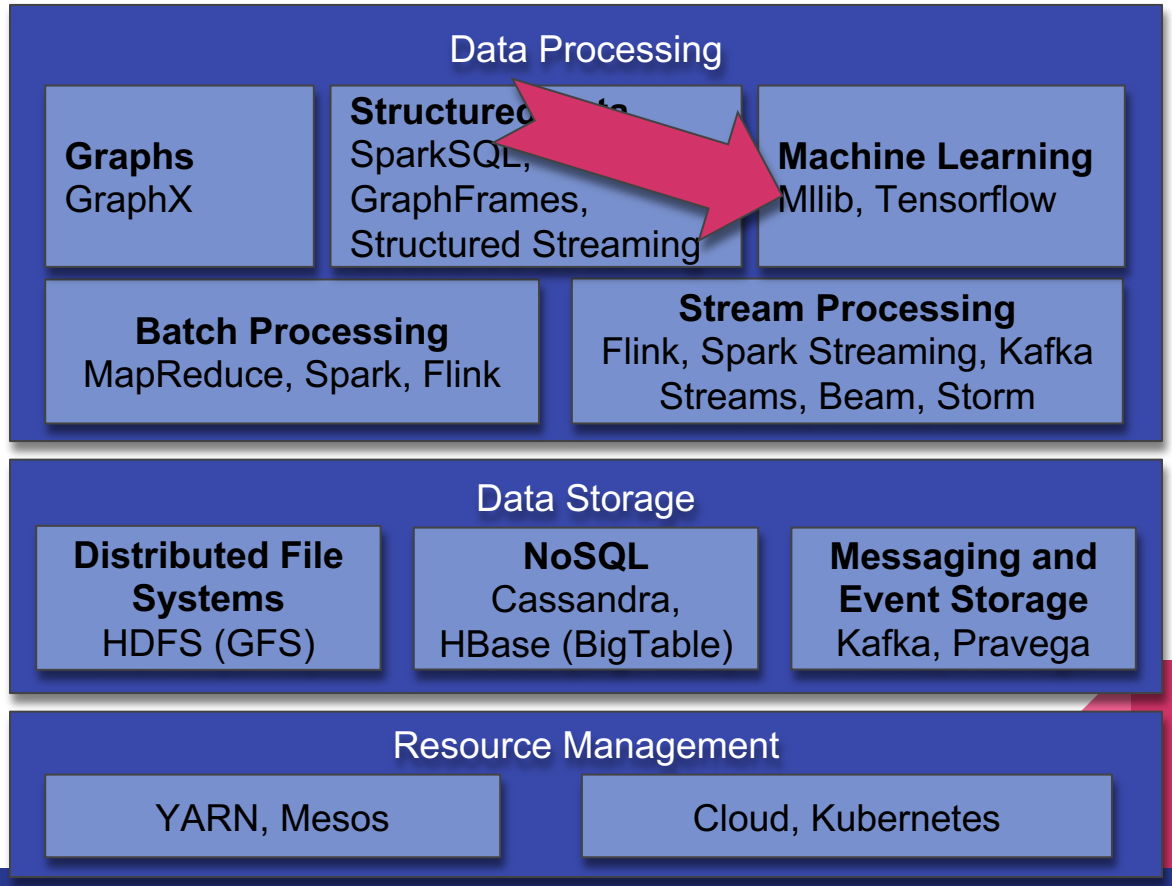
- A package for Apache Spark which provides DataFrame-based Graphs
- Provides high-level APIs in Scala, Java, and Python
- GraphX is to RDDs as GraphFrames are to DataFrames

GraphFrames Example

```
# Create a Vertex DataFrame with unique ID column 'id'
v = sqlContext.createDataFrame([('a', 'Alice', 34),
  ('b', 'Bob', 36), ('c', 'Charlie', 30),], ['id', 'name', 'age'])

# Create an Edge DataFrame with 'src' and 'dst' columns
e = sqlContext.createDataFrame([('a', 'b', 'friend'),
  ('b', 'c', 'follow'),  ('c', 'b', 'follow'),], ['src', 'dst', 'relationship'])

# Create a GraphFrame
g = GraphFrame(v, e)

# Query: Get in-degree of each vertex.
g.inDegrees.show()

# Query: Count the number of 'follow' connections in the graph.
g.edges.filter('relationship = "follow"').count()

# Run PageRank algorithm, and show results.
results = g.pageRank(resetProbability=0.01, maxIter=20)
results.vertices.select('id', 'pagerank').show()
```

Spark MLlib

Big Data Ecosystem (Simplified)



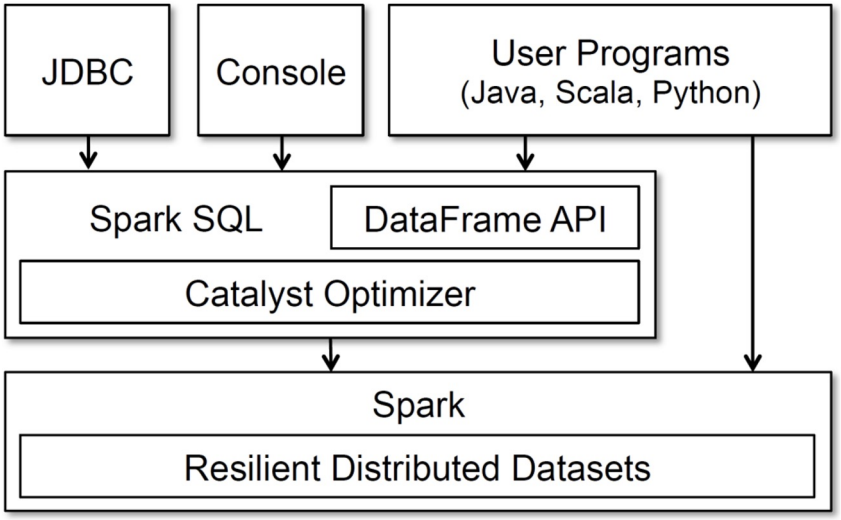
What is Spark MLlib

Goal: Make practical machine learning scalable and easy!

- ML Algorithms: common learning algorithms
  - Classification, regression, clustering, and collaborative filtering
- Featurization:
  - Feature extraction, transformation, dimensionality reduction, and selection
- Pipelines:
  - Tools for constructing, evaluating, and tuning ML Pipelines
- Persistence:
  - Saving and load algorithms, models, and Pipelines
- Utilities:
  - Linear algebra, statistics, data handling, etc

MLlib: RDD vs. DataFrames

- MLlib originally based on Spark's RDD API
- Spark introduced DataFrame API
  - Added structure and schema to the data
  - User-friendly API
  - SQL like operators and queries
  - Better optimizations (Tungsten and Catalyst)
  - Unified API across languages
- MLlib switched to DataFrame-based API
- Both are available but RDD-based api is expected to be removed in Spark 3.0

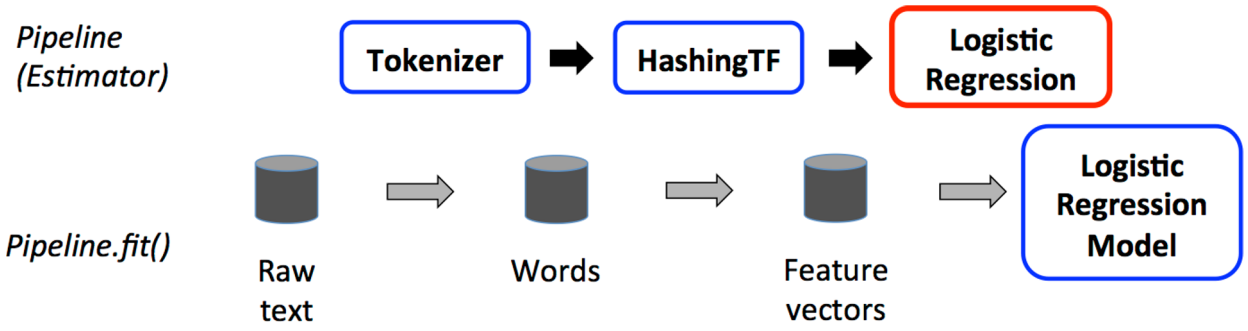


MLlib API

- Two major types of algorithms:
  - **Transformers:**
    - Take an input dataset and modify it via a transform() function to produce an output dataset. E.g., Binarizer
    - transform one DataFrame into another DataFrame
  - **Estimators:**
    - Take a training dataset, use a fit() function to train an ML model, and output that model
    - The model is itself a Transformer. Calling transform() will add a new column of predictions. E.g., Logistic Regression
    - Fit on a DataFrame to produce a Transformer
  - Transformer.transform()s and Estimator.fit()s are both stateless

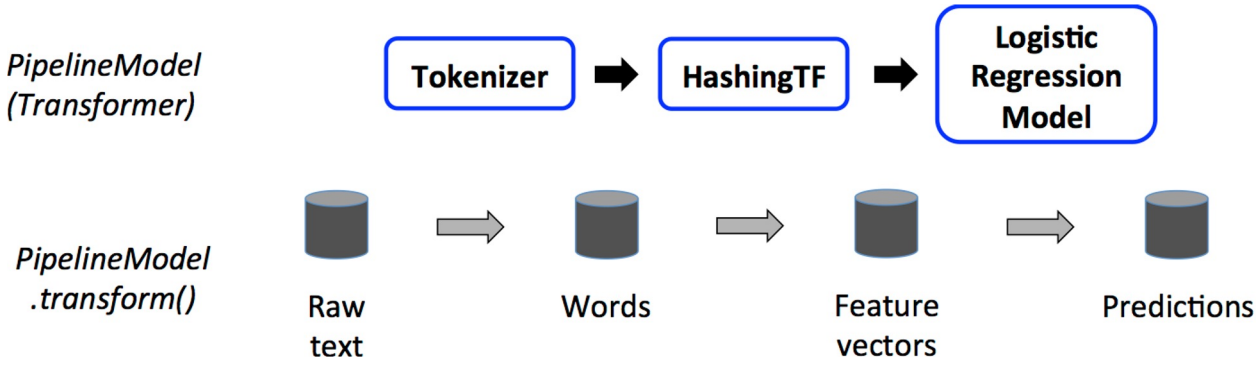
ML Pipelines

- Mostly inspired by the scikit-learn project
- Combine multiple Transformers and Estimators into a data analytics workflow



PipelineModel

- A pipeline is an Estimator
- Pipeline's fit() method produces a PipelineModel
- PipelineModel is a Transformer



Example (1/2)

```
# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer,
hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

Example (2/2)

```
# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s, prediction=%f" % (rid, text,
str(prob), prediction))
```

Output

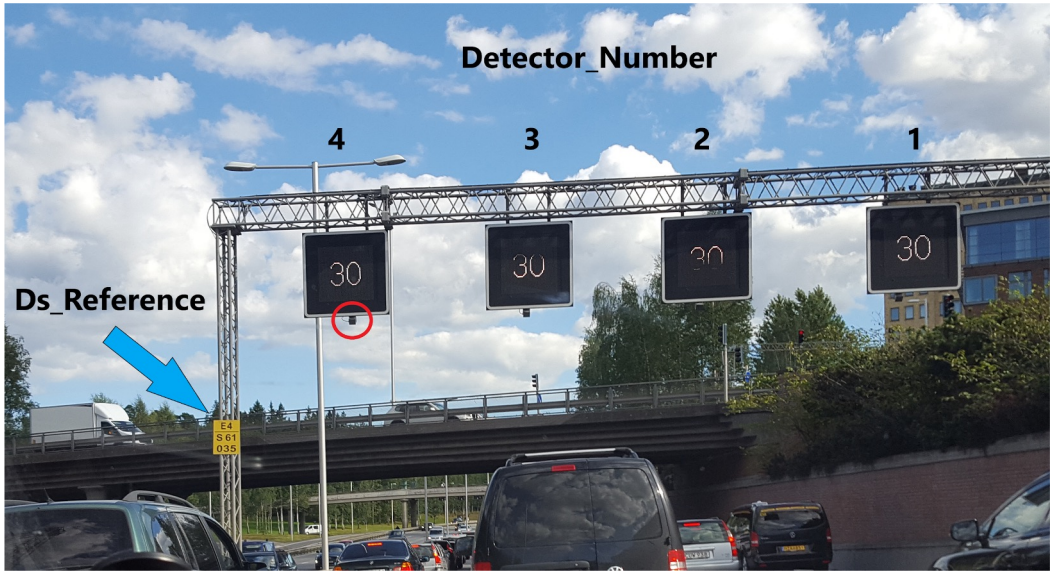
```
(4, spark i j k) --> prob=[0.155543713844,0.844456286156], prediction=1.000000
(5, l m n) --> prob=[0.830707735211,0.169292264789], prediction=0.000000
(6, spark hadoop spark) --> prob=[0.0696218406195,0.93037815938], prediction=1.000000
(7, apache hadoop) --> prob=[0.981518350351,0.018481649649], prediction=0.000000

prediction.show()
+---+-----------------+-------------------+-------------------+--------------------+--------------------+----------+
| id|             text|              words|           features|       rawPrediction|         probability|prediction|
+---+-----------------+-------------------+-------------------+--------------------+--------------------+----------+
|  4|      spark i j k|    [spark, i, j, k]|(262144,[20197,24...|[-1.6917661618366...|[0.15554371384424...|       1.0|
|  5|            l m n|          [l, m, n]|(262144,[18910,10...|[1.59065143175960...|[0.83070773521117...|       0.0|
|  6|spark hadoop spark|[spark, hadoop, s...|(262144,[155117,2...|[-2.5925128063687...|[0.06962184061952...|       1.0|
|  7|    apache hadoop|    [apache, hadoop]|(262144,[66695,15...|[3.97232238082365...|[0.98151835035101...|       0.0|
+---+-----------------+-------------------+-------------------+--------------------+--------------------+----------+

test.show()
+---+-----------------+
| id|             text|
+---+-----------------+
|  4|      spark i j k|
|  5|            l m n|
|  6|spark hadoop spark|
|  7|    apache hadoop|
+---+-----------------+
```

Example:
Processing Road Traffic with Spark

Road Traffic Flow Analysis with Spark

Getting Started Guide:

https://alshishtawy.github.io/traffic-flow-analysis.html



Data Formats

Storage Data Formats

- Many formats to choose from:
  - csv
  - Json
  - Avro
  - Parquet
  - ORC
  - ….
- To compress, or not to compress
- Which is better for data science
  - why?
  - How big is the difference
- Storage Efficiency vs. Compute Efficiency

Storage Efficiency

Test using a month of Stockholm's highway traffic data

Format	Size (MB)	% of Original	Convert & Save (sec)	Notes
Original CSV	5064	100.00%	-	Row, Txt, No Schema
Original Zipped CSV	480	9.48%	-	Row, Txt, No Schema, Compressed
Cleaned CSV	4937	97.49%	162	Row, Txt, No Schema
Json	20919	413.09%	410	Row, Txt, Schema
Parquet	175	3.46%	166	Col, Bin, Schema Compressed

Compute Efficiency (1/2)

Test using the two queries below

Query-1: Count distinct detectors in the dataset

```
%%time
spark.read.parquet('../../parquet/mcs_201611').select('Road', 'Km_Ref', 'Detector_Number') \
            .distinct().count()
```

Query-2: Calculate Density for E4N and plot histogram

```
%%time
q2 = spark.read.parquet('../../parquet/mcs_201611') \
            .select('Flow_In', 'Average_Speed') \
            .where('Status == 3 AND Road == "E4N"') \
            .withColumn('Density', col('Flow_In')*60/col('Average_Speed')) \
            .select('Density').rdd.flatMap(lambda x: x).histogram(list(range(0,55,5)))
```

Compute Efficiency (2/2)

File Format	Query 1		Query 2	
	Avg. (sec)	% of CSV	Avg. (sec)	% CSV
CSV	23	100%	29	100%
Json	95	413%	101	354%
Parquet	3	14%	13	46%

Why Parquet is Much Better?

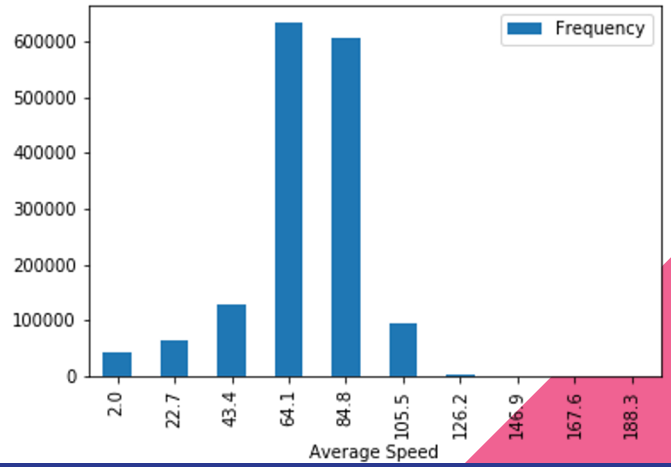
- Column vs row storage
  - column pruning, better compression
- Binary format is more efficient
- Fast compression (using Snappy)
  - Disks are the bottleneck in data intensive applications
  - Standard Zip can not be read in parallel
- Predicate/filter push down
  - evaluating predicates against metadata
  - e.g., minimum and maximum value for a column chunk

Visualization Hints

Visualization Hints

… from the road traffic analysis use cases

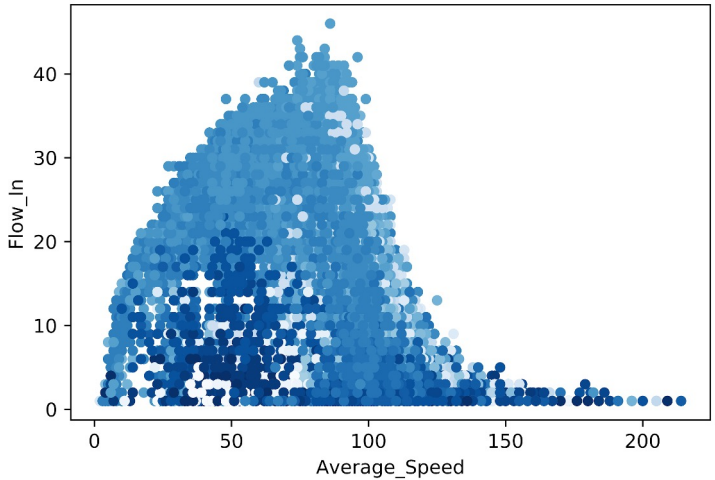
- Don’t be surprised if your computer crashes when plotting millions of points
- Option 1: plot a sample
  - but be very careful how you select your sample
- Option 2: summarize the data
  - let spark do the heavy lifting
  - e.g., plot a histogram



Example

Plotting one sensor for one month is OK

Exercise: How to plot a year?



Acknowledgements

Acknowledgements

- Some of the slides are based on slides by Amir Payberah
- Some of the slides are based on slides by Paris Carbone
- Some code example and images are from the official Spark documentation

Questions?