



Aalto University  
School of Science

# Big Data Processing with MapReduce/Spark Programming Models

*Hong-Linh Truong*  
*Department of Computer Science*  
*[linh.truong@aalto.fi](mailto:linh.truong@aalto.fi), <https://rdsea.github.io>*

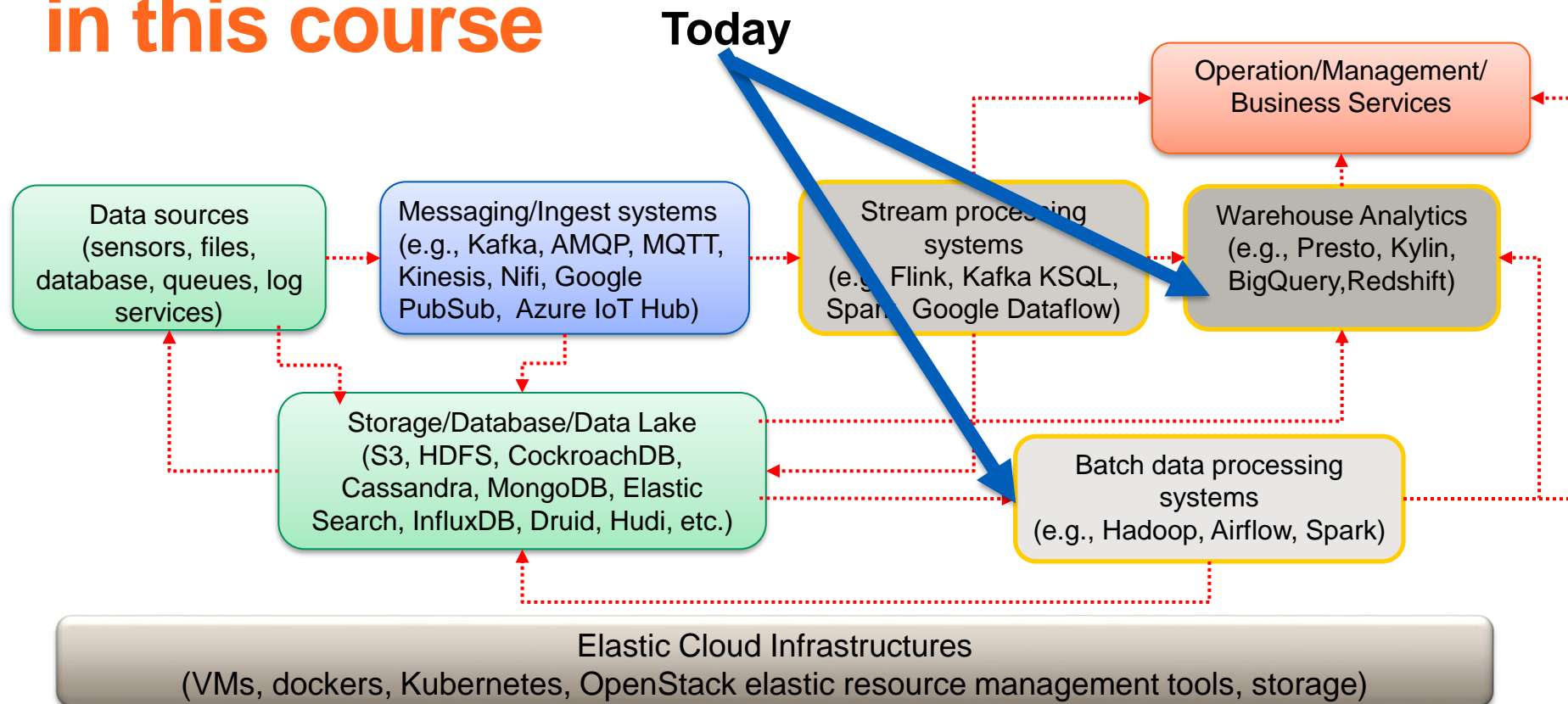
# Learning objectives

- **Be familiar with big data processing models using multiple nodes/clusters**
- **Understand MapReduce/Spark programming models for big data processing**
- **Able to perform practical programming features with MapReduce/Spark**
- **Able to design and apply MapReduce/Spark data processing with Hadoop and other frameworks**

# Big Data Processing in a platform

- **Data processing can be in different places**
  - Data ingestion and analytics
- **Simple/basic vs complex, example**
  - Basic transformations during ingestion
  - Basic queries of data from big data storage/databases
  - Complex, application-specific data analytics
    - *Realtime vs batch processing*

# Big data at large-scale: the big picture in this course

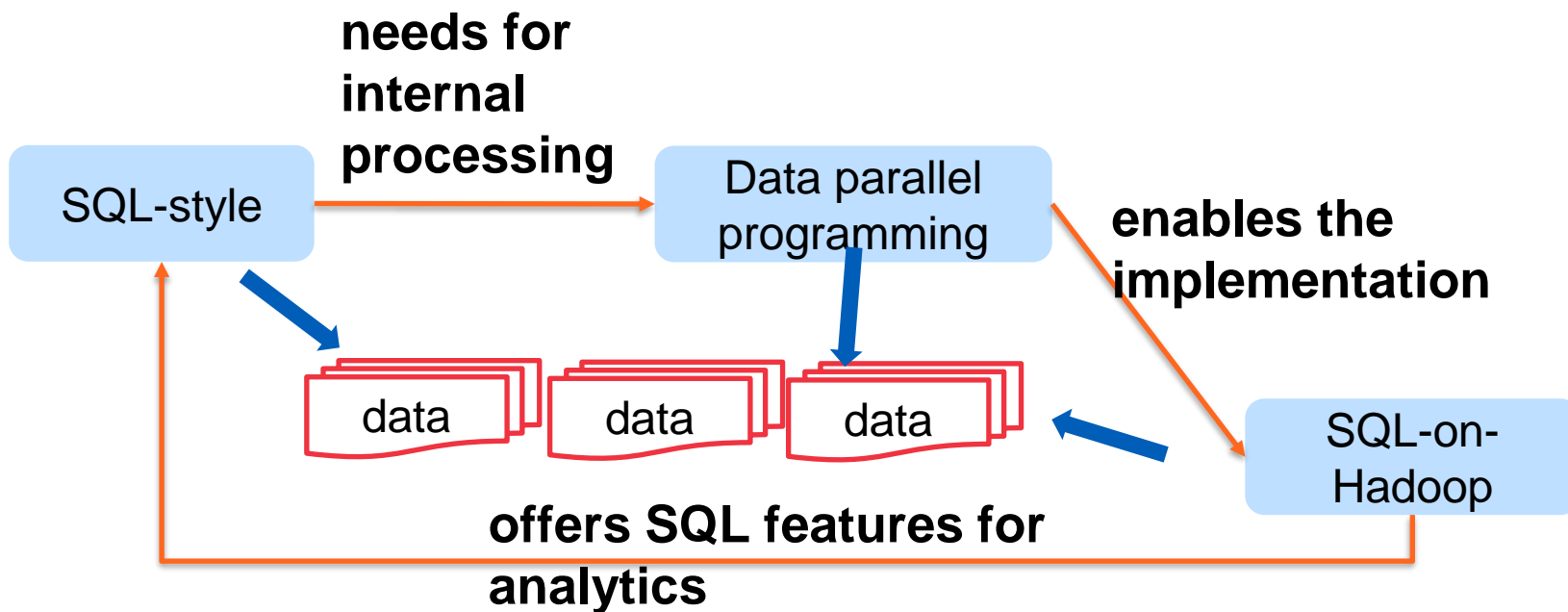


**Our first focus: big data  
analytics for data at rest**

# Recall: data at rest

- **Distributed file systems/object storages**
  - Files in Hadoop, distributed file systems, object storage
- **Data in a set of databases or in “datalake”**
- **Multiple types of big data analytics with high concurrent/parallel data writes/reads**
- **Dealing with different data access/analytics frequencies: hot, warm and cold data**
  - Different performance, volume, etc requirements

# Recall: SQL-style/data parallel processing for data in Hadoop FS



# Recall: SQL-Style

- **Analytics with big data databases**
  - NoSQL or NewSQL but they are very scale
  - E.g., Aurora, Cosmos, BigQuery
- **Analytics with federated databases**
  - Using scalable analytics engines to connect to different databases
  - Analytics using SQL-style queries or workflows
- **From the analytics: the developer is familiar with the traditional way**
- **Tools:**
  - built-in SQL features, Superset, Presto, etc.



# Data Analytics



## Query Engine

Data  
Analysis

REST API

JDBC

Tool-specific APIs

Client Libraries

Object-based  
storage  
(e.g. S3)

Relational  
Database  
(e.g. MySQL)

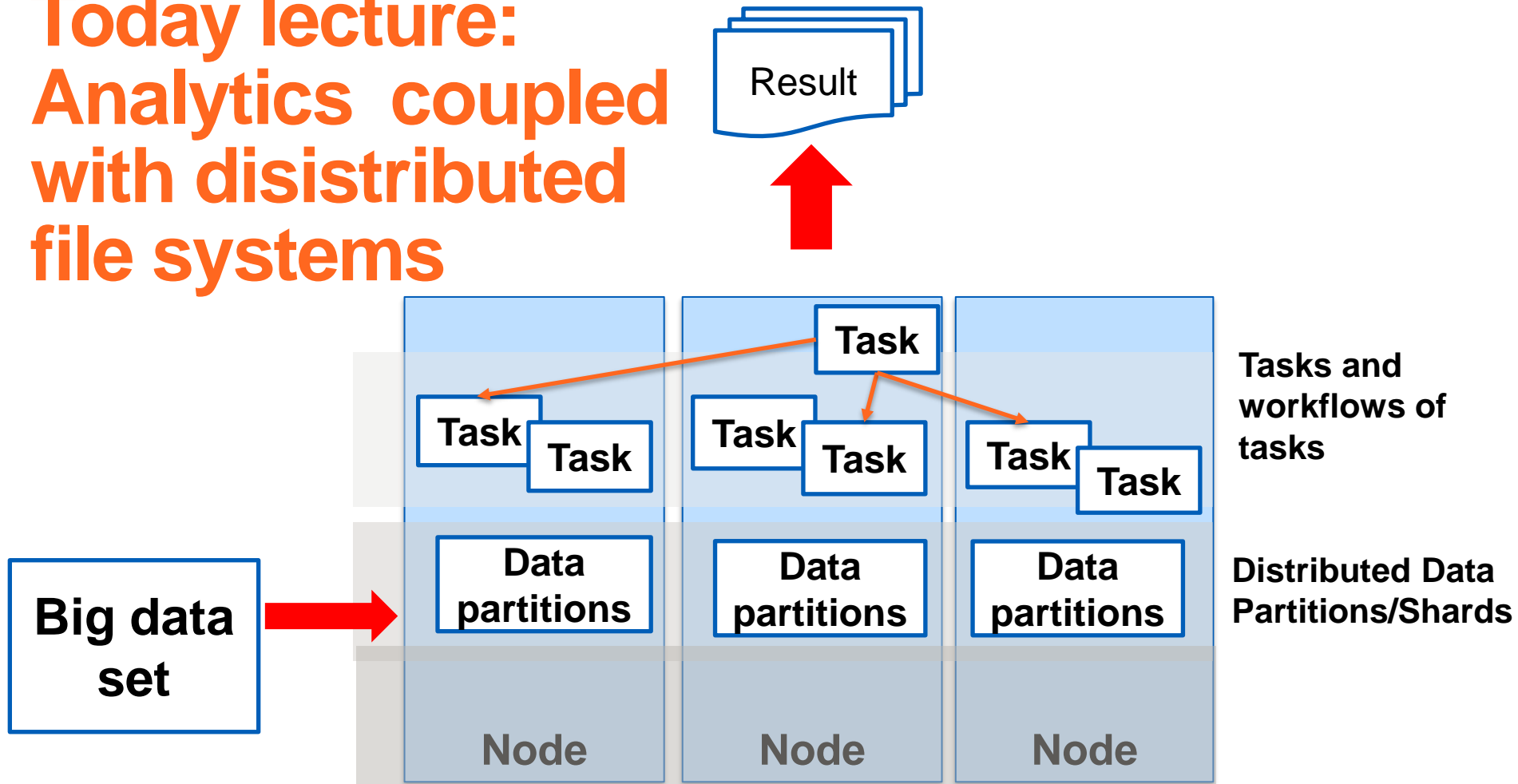
Distributed  
File Systems

NoSQL  
Database  
(e.g.  
MongoDB)

Real time  
data  
sources

**Presto as a query engine is an example for big data**

# Today lecture: Analytics coupled with distributed file systems



# Analysis of data in a DataFrame view

## Example taxi records

passenger_count	trip_distance	RatecodeID	store_and_fwd_flag	PULocationID	DOLocationID	payment_type	fare_amount	extra	mta_tax	tip_amount	tolls_amount	improvement_surcharge	total_amount
1	1.34	1	N	238	236	2	10.0	0.0	0.5	0.0	0.0	0.3	10.8
1	1.34	1	N	238	236	2	10.0	0.0	0.5	0.0	0.0	0.3	10.8
1	0.32	1	N	238	238	2	4.0	0.0	0.5	0.0	0.0	0.3	4.8
1	0.32	1	N	238	238	2	4.0	0.0	0.5	0.0	0.0	0.3	4.8
1	1.85	1	N	236	238	2	10.0	0.0	0.5	0.0	0.0	0.3	10.8
1	1.85	1	N	236	238	2	10.0	0.0	0.5	0.0	0.0	0.3	10.8
1	1.65	1	N	68	237	2	12.5	0.0	0.5	0.0	0.0	0.3	13.3
1	1.65	1	N	68	237	2	12.5	0.0	0.5	0.0	0.0	0.3	13.3
1	1.07	1	N	170	68	2	9.0	0.0	0.5	0.0	0.0	0.3	9.8
1	1.07	1	N	170	68	2	9.0	0.0	0.5	0.0	0.0	0.3	9.8
1	1.3	1	N	107	170	2	7.5	0.0	0.5	0.0	0.0	0.3	8.3
1	1.3	1	N	107	170	2	7.5	0.0	0.5	0.0	0.0	0.3	8.3
1	1.85	1	N	113	137	2	10.0	0.0	0.5	0.0	0.0	0.3	10.8
1	1.85	1	N	113	137	2	10.0	0.0	0.5	0.0	0.0	0.3	10.8
1	0.62	1	N	231	231	2	4.5	0.0	0.5	0.0	0.0	0.3	5.3
1	0.62	1	N	231	231	2	4.5	0.0	0.5	0.0	0.0	0.3	5.3
1	0.0	1	N	264	264	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.29	1	N	162	162	2	4.0	0.0	0.5	0.0	0.0	0.3	4.8
1	0.29	1	N	162	162	2	4.0	0.0	0.5	0.0	0.0	0.3	4.8
1	1.34	1	N	239	151	2	7.0	0.5	0.5	0.0	0.0	0.3	8.3

Very common we analyze **big data files** based on this view

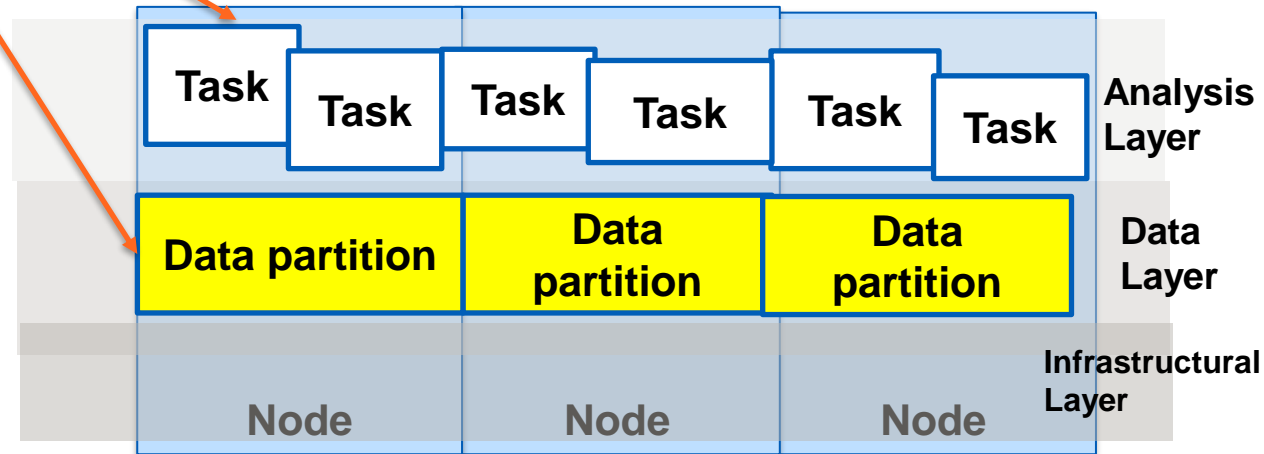
```

inputFile =args.input_file
## hadoop inputFile="hdfs://"
df =spark.read.csv(inputFile,header=True,inferSchema=True)
#df.show()
print("Number of trips", df.count())
#number of passenger count per vendor and total amount of money
passenger_exprs = {"passenger_count":"sum","total_amount":"sum"}
df2 = df.groupBy('VendorID').agg(passenger_exprs)
# Where do you want to write the output
df2.repartition(1).write.csv(args.output_dir,header=True)

```



What we need  
when we  
develop analysis  
programs for big  
data



# Big data processing techniques in our focus for data at rest

- **Programming models**
  - MapReduce/Spark
  - Workflows
- **Studied frameworks**
  - Apache Hadoop/Spark, Dask
  - Apache Airflow
- **Not in our focus:**
  - Bulk synchronous parallel (BSP)
  - HPC MPI (Message Passing Interface)

# MapReduce

<https://hadoop.apache.org>

# Several patterns

- Thinking if we have data that can be represented as **record=(key,value)**
  - Potentially millions of records, with millions of keys
- **Analytics**
  - Summarization/aggregation/filtering
    - *count, min, max, average, etc.*
  - Join data from big data set

# MapReduce programming model

- **MapReduce is a programming model original from Google**
  - Various implementations/frameworks support MapReduce
  - Apache Hadoop (originally from Yahoo!) is the most famous one
  - **<https://hadoop.apache.org>**
- **Support batch data processing for very large datasets**
- **Suitable for batch jobs in big data, e.g.,**
  - Web search, document processing, ecommerce information
  - Extract, transform, data wrangling, and data cleansing tasks



# Map & Reduce

- **Map: map data into (key, value)**
  - Receives `<key,value>`
  - Outputs `<key,value>` - new set of `<key,value>`
- **Reducer: compute results from the same key**
  - Receives `<key, Iterable[value]>`
  - Outputs `<key,value>`

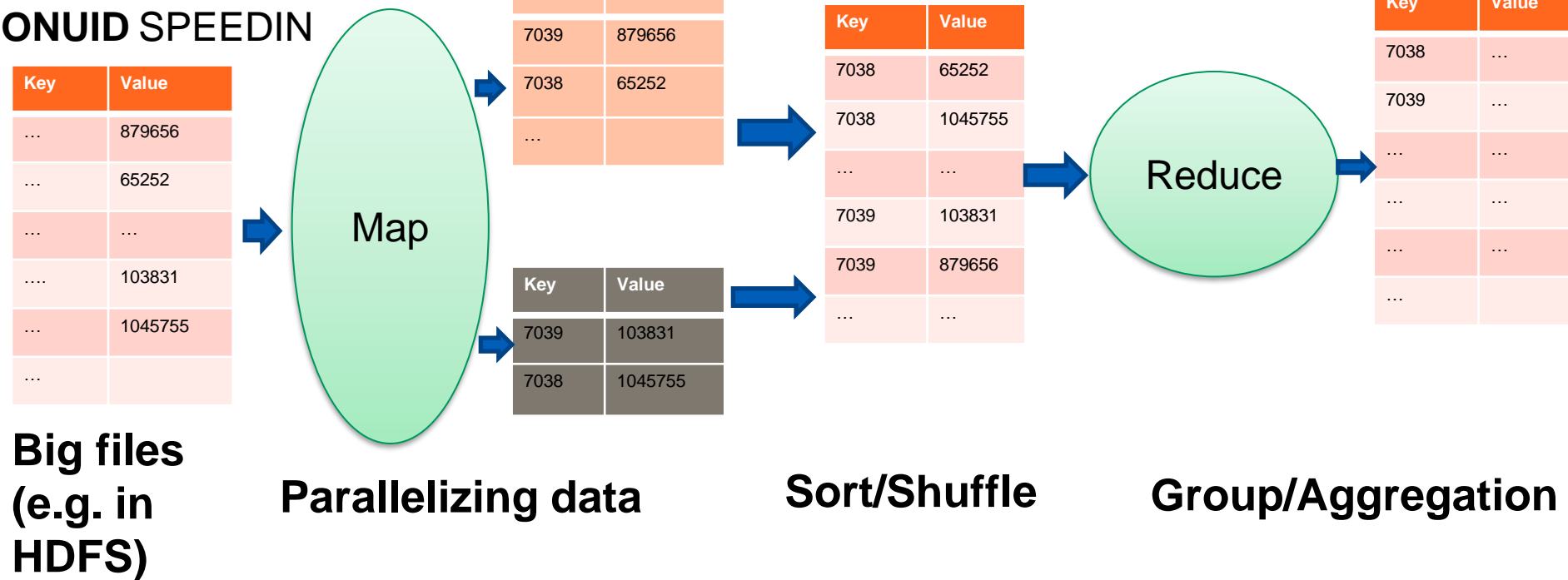
# Example of a real data

## Look at the network monitoring data

```
PROVINCECODE,DEVICEID,IFINDEX,FRAME,SLOT,PORT,ONUINDEX,ONUID,TIME,SPEEDIN,SPEEDOUT
XXN,10XXXXXX023,26XXXXXX8,1,2,7,39,100XXXXXX2310207039,01/08/2019 00:04:07,148163,49018
XXN,10XXXXXX023,26XXXXXX8,1,2,7,38,100XXXXXX2310207038,01/08/2019 00:04:07,1658,1362
XXN,10XXXXXX023,26XXXXXX8,1,2,7,9,100XXXXXX2310207009,01/08/2019 00:04:07,6693,5185
```

Sample: <https://version.aalto.fi/gitlab/bigdataplatfoms/cs-e4640/-/tree/master/data/onudata>

# Understand the MapReduce programming model



# Key ideas of MapReduce

- **Data can be divided by “Map” operators**
  - data processing tasks extract “intermediate results”
- **Intermediate results can be aggregated through “Reduce” operators**
  - data processing tasks produce a result from “intermediate results”
- **We can glue “Map” and “Reduce” operators into a multi-stage data flow model**
- **Other possible operators:**
  - **Combiner**: performs “Reduce” at local nodes
  - **Partitioner**: decides key/value for Reduce

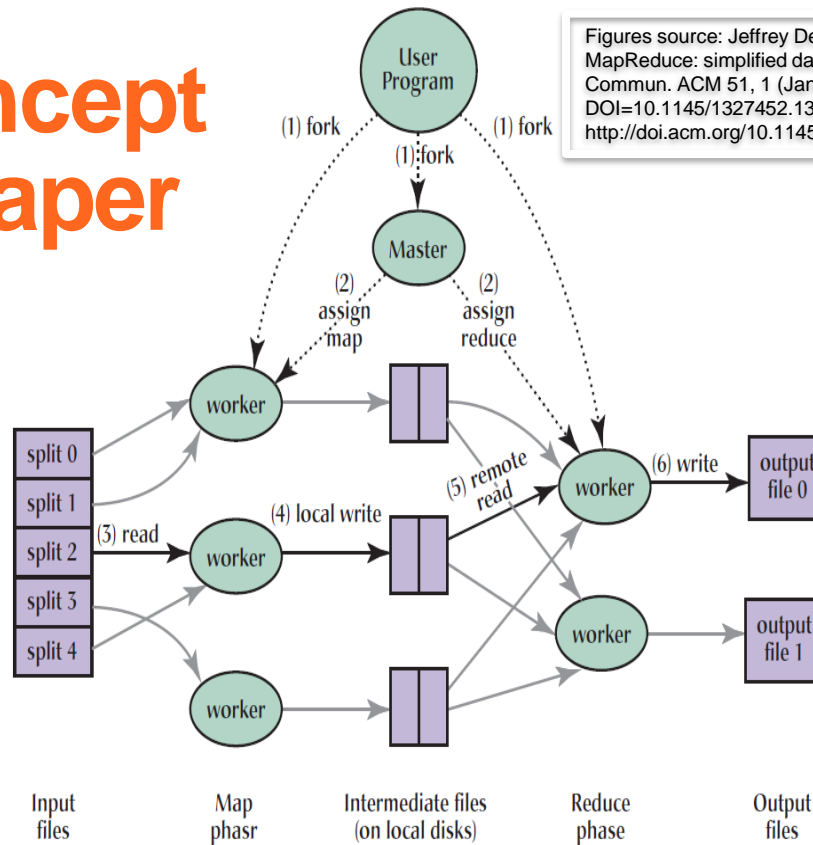
# Key ideas of MapReduce

- **Key points for the developers**
  - should write only the main “logic”: Map and Reduce operators
- **The runtime framework will**
  - handle data movement and input/output management for Map/Reduce tasks
  - parallelizing tasks in multiple nodes

# MapReduce concept in the original paper

```
map(String key, String value):  
    // key: document name  
    // value: document contents  
    for each word w in value:  
        EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```



Figures source: Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107-113. DOI=10.1145/1327452.1327492 <http://doi.acm.org/10.1145/1327452.1327492>

**Key point: parallelize workers to process a lot of input files and produce a lot of output files**

# Hadoop MapReduce

- **Hadoop supports the MapReduce programming model**
  - Use cluster nodes for data processing tasks
  - Access data in HDFS and data partitioned in different nodes
  - Hadoop runtime automatically creates parallel tasks
  - YARN is used to run jobs of MapReduce applications
- **Data management (HDFS) and data processing (MapReduce) is aligned nicely**
  - Run in the same nodes → data locality optimization

# Map/Reduce tasks and data/node partitions

- **A Map task can handle a data partition in the same node**
  - E.g., a Map task handles a HDFS data block → local data optimization: no data movement - local processing
  - Results from a Map task are **intermediate** → to where a task will store them?
  - *What if a Map task fails?*
- **Reduce Task**
  - Need to deal with data produced from different Map tasks → *where to run the Reduce tasks?*



# Examples - Map

```
public static class SpeedInMapper
    extends Mapper<Object, Text, LongWritable, AverageWritable> {
    private LongWritable id = new LongWritable();
    private AverageWritable averagecount = new AverageWritable();
    public void map(Object key, Text value, Context output)
        throws IOException, InterruptedException {

        String valueString = value.toString();
        String[] record = valueString.split(",");
        id.set(Long.parseLong(record[7]));
        averagecount.setAverage(Float.parseFloat(record[9]));
        averagecount.setCount(1);
        output.write(id, averagecount);
    }
}
```

Input

Output

Parse the data to  
get ONUID and  
SPEEDIN

Map (ONUID,(SPEEDIN, count))

# Example - Reduce

Input

Output

```
public static class SpeedInAverageReducer
    extends Reducer<LongWritable, AverageWritable, LongWritable, FloatWritable> {
    private FloatWritable new_result = new FloatWritable();

    public void reduce(LongWritable key, Iterable<AverageWritable> values,
        Context context
        ) throws IOException, InterruptedException {

        float avg = 0;
        int count = 0;
        for (AverageWritable val : values) {
            float current_avg = val.getAverage();
            int current_count = val.getCount();
            avg = avg + (current_avg * current_count);
            count = count + current_count;
        }

        new_result.set(avg / count);
        context.write(key, new_result);
    }
}
```

Simple way to  
determine the  
average as  
“Reduce” operator

Reduce (ONUID,AVG)

# Driver: connect Map and Reduce operators

```
public static void main(String[] args) throws Exception {  
    Configuration conf = new Configuration();  
    Job job = Job.getInstance(conf, "simpleaverage");  
    job.setJarByClass(SimpleAverage.class);  
    job.setMapperClass(SpeedInMapper.class);  
    job.setCombinerClass(SpeedInAverageCombiner.class);  
    job.setReducerClass(SpeedInAverageReducer.class);  
    job.setMapOutputKeyClass(LongWritable.class);  
    job.setMapOutputValueClass(AverageWritable.class);  
    job.setOutputKeyClass(LongWritable.class);  
    job.setOutputValueClass(FloatWritable.class);  
    FileInputFormat.addInputPath(job, new Path(args[0]));  
    FileOutputFormat.setOutputPath(job, new Path(args[1]));  
    System.exit(job.waitForCompletion(true) ? 0 : 1);  
}
```

← **Combiner**

# Example with Python using MRJob

```
class ONUSpeedinAverage(MRJob):
    def mapper(self, _, entry):
        provincecode,deviceid,ifindex,frame,slot,port,onuindex,onuid,timestamp,speedin,speedout= entry.split(",")
        #average speedin with count = 1
        yield (onuid, (float(speedin),1))

    ## recalculate the new speedin average through an array of speedin average values
    def _recalculate_avg(self, onuid, speedin_avg_values):
        current_speedin_total = 0
        new_avg_count = 0
        for speedin_avg, avg_count in speedin_avg_values:
            current_speedin_total = current_speedin_total +(speedin_avg*avg_count)
            new_avg_count = new_avg_count + avg_count
        new_speedin_avg = current_speedin_total/new_avg_count
        return (onuid, (new_speedin_avg, new_avg_count))

    def combiner(self, onuid, speedin_avg_values):
        yield self._recalculate_avg(onuid, speedin_avg_values)

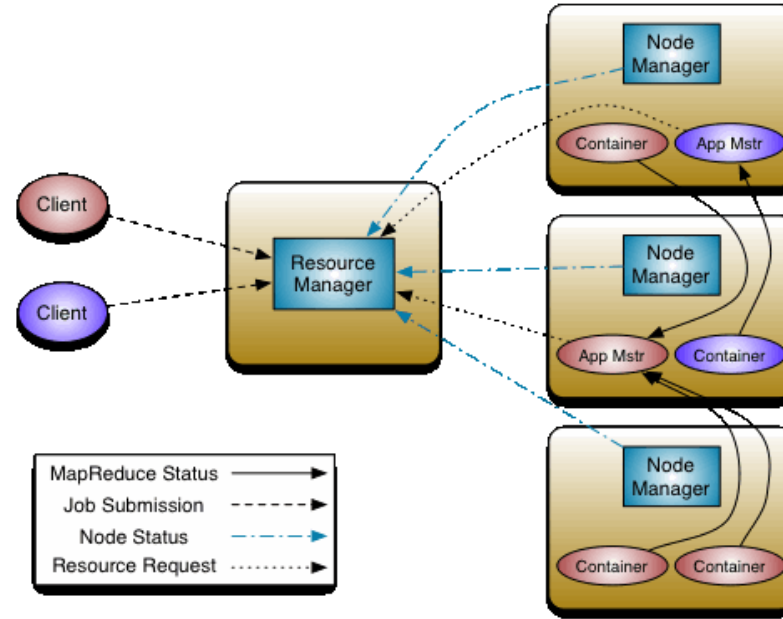
    def reducer(self, onuid,speedin_avg_values):
        onuid, (speedin_avg, avg_count) = self._recalculate_avg(onuid,speedin_avg_values)
        yield (onuid, speedin_avg)

if __name__ == '__main__':
    ONUSpeedinAverage.run()
```

Note: see code examples in our GIT

# Recall: Resource management and execution in Hadoop YARN

A cluster of computing nodes can be managed by YARN or Mesos



Source:

<http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html>

# Schedule and monitoring for MapReduce

- **A MapReduce program runs → MapReduce Job**
  - includes many tasks (Map and Reduce processes + others)
- **JobTracker**
  - monitors the whole job (all tasks of a MapReduce program)
- **TaskTracker**
  - performs a task of the MapReduce applications
  - informs JobTracker about the state of the tasks

# Monitoring MapReduce Jobs



Logged in as: dr.who

## Application application\_1570429323498\_0008

▼ Cluster

- About
- Nodes
- Node Labels
- Applications
- NEW
- NEW\_SAVING
- SUBMITTED
- ACCEPTED
- RUNNING
- FINISHED
- FAILED
- KILLED
- Scheduler

► Tools

Application Overview

User:	mybdbp
Name:	cse4640-nytaxicount
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Fri Oct 25 19:22:08 +0000 2019
Elapsed:	3mins, 6sec
Tracking URL:	History
Log Aggregation Status:	DISABLED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	5039065 MB-seconds, 973 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Show 20 ▼ entries

Search:

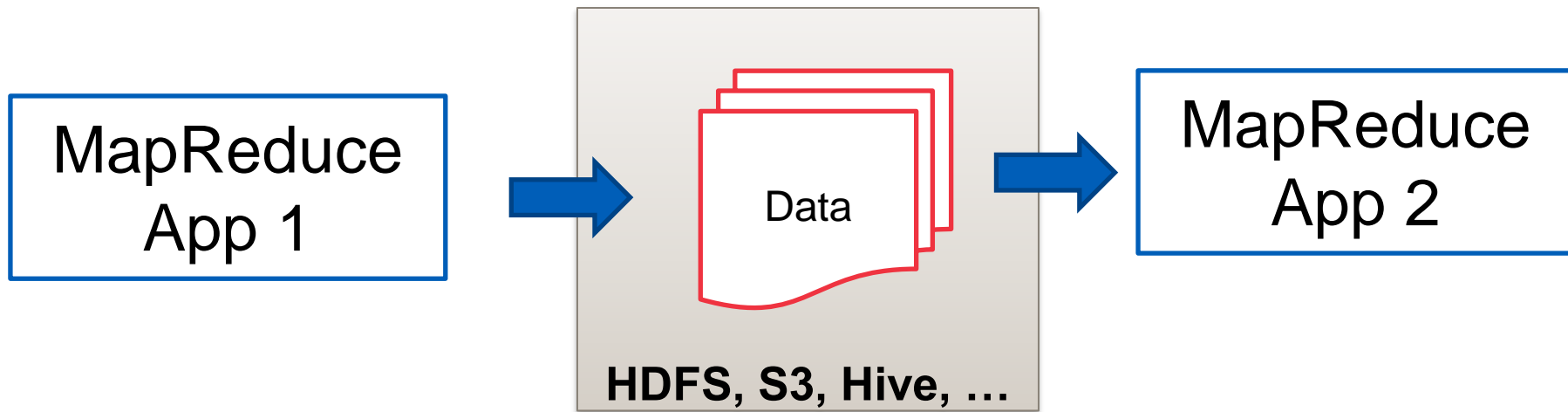
Attempt ID	Started	Node	Logs	Nodes blacklisted by the app	Nodes blacklisted by the system
appattempt_1570429323498_0008_000001	Fri Oct 25 22:22:08 +0300 2019	<a href="http://cluster-bdp-w-3.c.bigmultidatstore.internal:8042">http://cluster-bdp-w-3.c.bigmultidatstore.internal:8042</a>	<a href="#">Logs</a>	0	0

Showing 1 to 1 of 1 entries

First Previous 1 Next Last

# Connecting MapReduce applications

**Build complex MapReduce pipelines**



**Using big data storage/database as data exchange**

**We can use workflows to coordinate different MapReduce apps**



# Problems with MapReduce

- **Strict Map & Reduce tasks connection → limitation**
- **Need more flexible in processing big data workloads**
  - Batch data flows and streaming data flows
- **Programming diversity support**
  - Software engineering productivity

# Apache Spark

<https://spark.apache.org/>

# Apache Spark

- **Cluster-based high-level computing framework**
- **“unified engine” for different types of big data processing**
  - SQL/structured data processing
  - Machine learning
  - Graph processing
  - Streaming processing
- **It is a powerful computing framework and system → an important service that a big data platform should support**

# Apache Spark

Can be run a top

- Hadoop (using HDFS and YARN)
- Mesos cluster
  - <http://mesos.apache.org/>
- Kubernetes
- Standalone machines

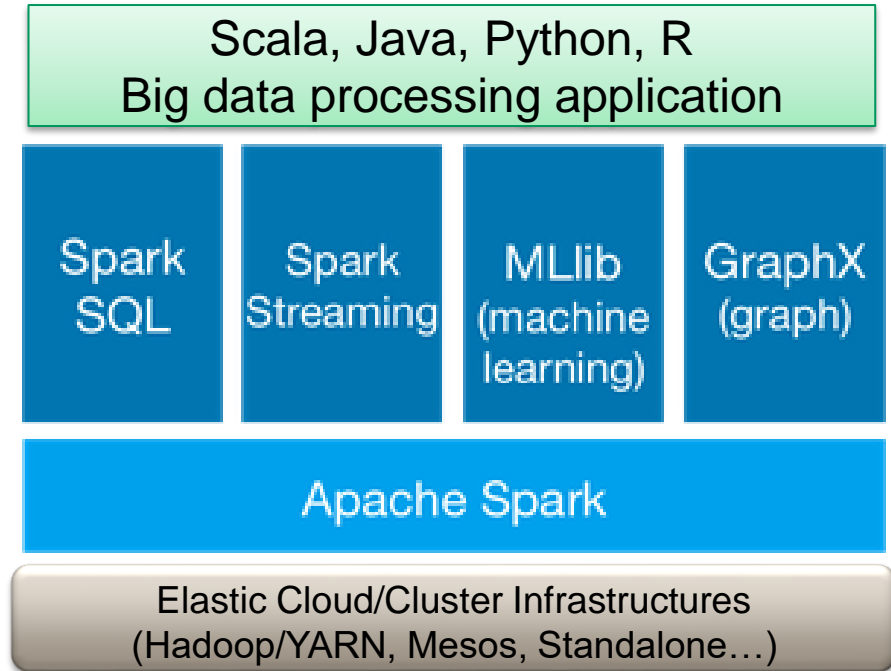


Figure source: <http://spark.apache.org/>

# Key features

- **Data is distributed in different nodes for processing**
  - Like data distributed in different nodes in big storage/database
- **Leverage parallel computing concepts to run multiple tasks**
  - Parallel tasks, task pipeline, DAG of processing stages
- **Employ in-memory big data processing**
- **Program driver steers the execution of parallel tasks**
  - Tasks are paralleled automatically and are scheduled with different underlying schedulers
- **Key data operators**
  - Transformations and actions on data

# Spark Program: programming elements

- **SparkSession**

- Act as a program driver to manage the execution of tasks
- SparkContext: manages connection to cluster, manage internal services

- **Data APIs**

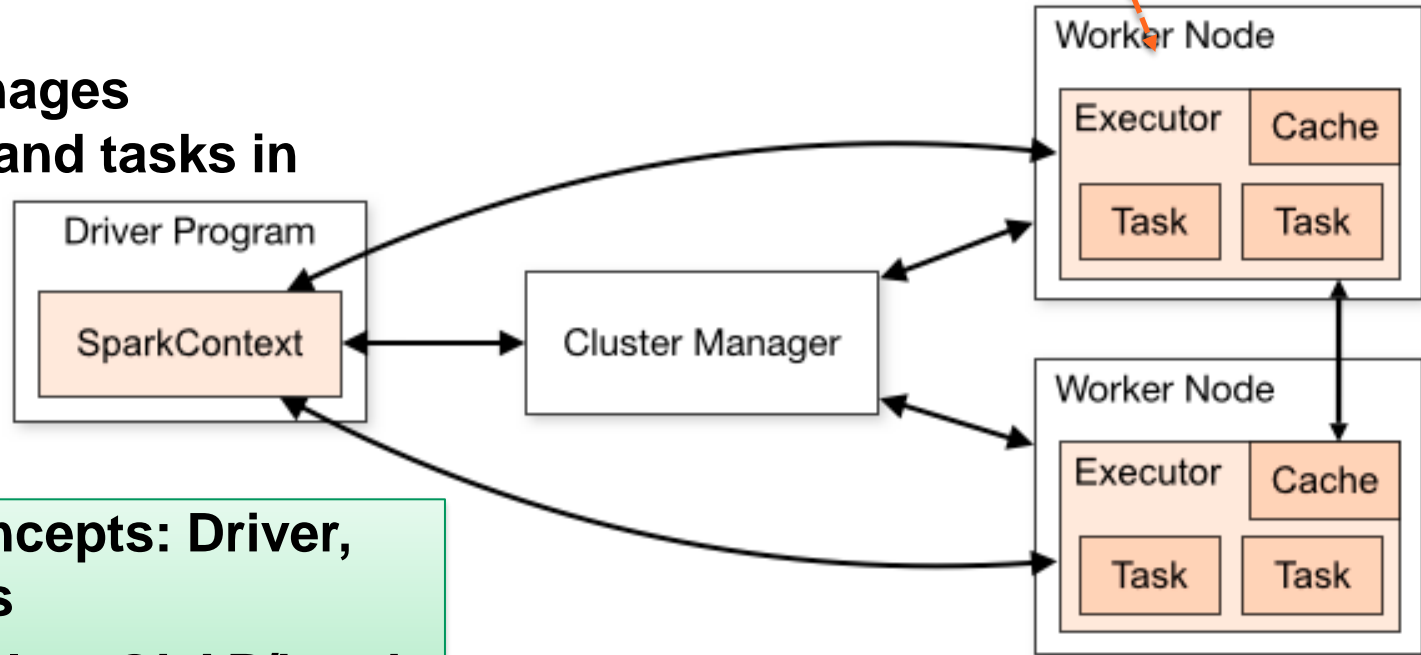
- Low-level Resilient Distributed Dataset (RDD)
- High-level DataFrames/DataSets
- Load and hold distributed data
- Transformation and action functions

- **ML, graph and streaming functions and pipelines**

# Execution Model

Driver manages operators and tasks in nodes

Map into a resource in a cluster node



**Common concepts: Driver, Nodes, Tasks**

**Workload styles: OLAP/batch jobs with a lot of data**

Figure source: <http://spark.apache.org/docs/latest/cluster-overview.html>

# Spark application management: high-level view

- **Submission/Request**
    - Submit the Spark application for running
    - Resource is provided for running the Driver
  - **Launch**
    - The Driver requests resources for executors (through SparkContext)
    - Establish executors across worker nodes
  - **Execution**
    - The driver starts to execute code and move data
  - **Finish/Completion:**
    - Finish, release executors
-

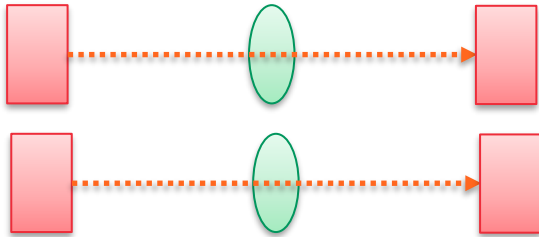


# Spark program logic: typical steps

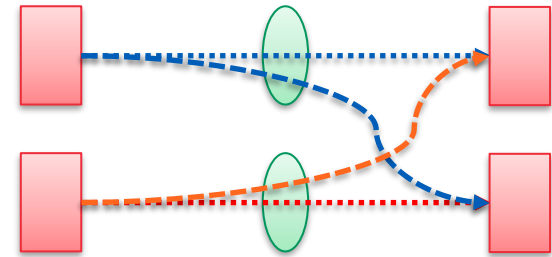
- **Load data and distribute data**
  - Data is **immutable** after created
  - Data partition in Spark: a partition is allocated in a node
- **Perform **transformation and action** operators**
  - *Transformations*: build plans for transforming data models
  - *Actions*: perform computation on data
- **The developer mostly focuses on loading data and performing operators**

# Transformation operators

- Transformation:
  - Instructions about how to transform a data in a form to another form → it will not change the original data (immutability)
- Only tell what to do: to build a DAG (direct acyclic graph): a lineage of what to do
- lazy approach → real transformation will be done at **action operators**



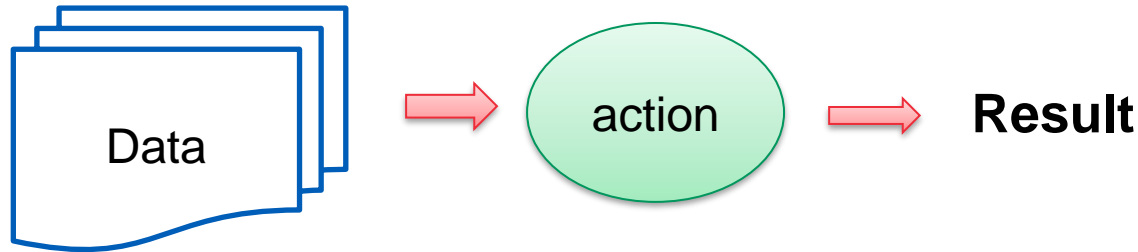
**Narrow transformation,  
no data shuffle**



**Wide transformation, cross data  
partitions, require shuffles**

# Action operators

- **Compute the results for a set of transformations**
  - Examples: count or average
- **Actions: view, collect, write, calculation**



**Lazy approach: an action triggers execution of transformation operators → enable various types of optimization**

# Resilient distributed dataset (RDD)

- **Low-level data structure**
  - Collection of data elements partitioned across nodes in the cluster
  - With data sharing, parallel operations, fault-tolerant features
- **Create RDD**
  - Created by loading data from files (text, sequence file) including local file systems, HDFS, Cassandra, HBase, Amazon S3, etc.
- **Persist RDD**
  - In memory or to files

# RDD transformations and actions

## Transformations

- **map**
- **filter**
- **sample**
- **intersection**
- **groupByKey**

## Actions

- **reduce()**
- **collect()**
- **count()**
- **saveAs...File()**

# Shared variables

- **A function is executed a remote and various tasks running in parallel**
  - How do tasks share variables? common patterns in parallel computing: *broadcast and global counter*
- **Variables used in parallel operations**
  - variables are copied among parallel tasks
  - shared among tasks or between tasks and the driver
- **Types of variables**
  - broadcast variables: cache a value in all nodes
  - accumulators: a global counter shared across processes

# Examples

```
conf = SparkConf().setAppName("CS-E4640-Broadcast").setMaster("ygs:master")
sc = SparkContext(conf=conf)
bVar = sc.broadcast([5,10])
print("The value of the broadcast",bVar.value,sep=" ")
counter = sc.accumulator(0)
sc.parallelize([1, 2, 3, 4]).foreach(lambda x: counter.add(bVar.value[0]))
print("The value of the counter is ",counter.value,sep=" ")
```

## Use cases:

- Broadcast variables: lookup tables
- Accumulators: monitoring/checkpoint counters

# Example with RDD

VendorID,tpep\_pickup\_datetime,tpep\_dropoff\_datetime,passenger\_count,trip\_distance,RatecodeID,store\_and\_fwd\_flag,PULocationID,DOLocationID,payment\_type,fare\_amount,extra,mta\_tax,tip\_amount,tolls\_amount,improvement\_surcharge,total\_amount

2,11/04/2084 12:32:24 PM,11/04/2084 12:47:41 PM,1,1.34,1,N,238,236,2,10,0,0.5,0,0,0.3,10.8

2,11/04/2084 12:32:24 PM,11/04/2084 12:47:41 PM,1,1.34,1,N,238,236,2,10,0,0.5,0,0,0.3,10.8

2,11/04/2084 12:25:53 PM,11/04/2084 12:29:00 PM,1,0.32,1,N,238,238,2,4,0,0.5,0,0,0.3,4.8

as a text file

```
conf = SparkConf().setAppName("cse4640-rddshow").setMaster(args.master)
sc = SparkContext(conf=conf)
##modify the input data
rdd=sc.textFile(args.input_file)
## if there is a header we can filter it otherwise comment two lines
csvheader = rdd.first()
rdd = rdd.filter(lambda csventry: csventry != csvheader)
## using map to parse csv text entry
rdd=rdd.map(lambda csventry: csventry.split(","))
rdd.repartition(1)
rdd.saveAsTextFile(args.output_dir)
```



# Spark SQL and DataFrames

- **High-level APIs**
  - Design with common programming patterns in data analysis, multi-language support
- **SparkSQL: enable dealing with structured data**
  - SQL query execution, Hive, JDBC/ODBC
- **DataFrame**
  - Distributed data organized into named columns, similar to a table in relational database
  - Pandas and Spark DataFrames have similar design concepts

# DataFrame

```
inputFile = args.input_file
df = spark.read.csv(inputFile, header=True, inferSchema=True)
print("Number of partition", df.rdd.getNumPartitions())
df.show()
```

PROVINCECODE	DEVICEID	IFINDEX	FRAME	SLOT	PORT	ONUINDEX	ONUID	TIME	SPEEDIN	SPEEDOUT
YN 1	3023	528	1	2	7	39 10	07039	01/08/2019 00:04:07	148163	49018
YN 1	3023	528	1	2	7	38 10	07038	01/08/2019 00:04:07	1658	1362
YN 1	3023	528	1	2	7	9 10	07009	01/08/2019 00:04:07	6693	5185
YN 1	3023	528	1	2	7	8 10	07008	01/08/2019 00:04:07	640	544
YN 1	3023	528	1	2	7	11 10	07011	01/08/2019 00:04:07	118	114
YN 1	3023	528	1	2	7	10 10	07010	01/08/2019 00:04:07	28514	12495
YN 1	3023	528	1	2	7	13 10	07013	01/08/2019 00:04:07	868699	23400
YN 1	3023	528	1	2	7	15 10	07015	01/08/2019 00:04:07	1822	1120
YN 1	3023	528	1	2	7	17 10	07017	01/08/2019 00:04:07	998069	117345
YN 1	3023	528	1	2	7	16 10	07016	01/08/2019 00:04:07	22402	1804
YN 1	3023	528	1	2	7	19 10	07019	01/08/2019 00:04:07	640	791
YN 1	3023	760	1	1	10	49 10	10049	01/08/2019 00:04:07	662	494
YN 1	3023	760	1	1	10	48 10	10048	01/08/2019 00:04:07	2158	759
YN 1	3023	528	1	2	7	21 10	07021	01/08/2019 00:04:07	0	0
YN 1	3023	760	1	1	10	51 10	10051	01/08/2019 00:04:07	2600890	54153
YN 1	3023	528	1	2	7	20 10	07020	01/08/2019 00:04:07	330	184

# Create DataFrame

**DataFrames can be created from a Hive table, from Spark data sources, or another DataFrame**

## Load and save

- From Hive, JSON, CSV
- HDFS, local file, etc.



Formats and Sources supported by DataFrames

Source: <https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html>

# DataFrame Transformations & Actions

- **Several transformations can be done**

- *Think transformation for relational database or matrix*

- **Select**

- *df.select*

- **Filter**

- *df.filter*

- **Groupby**

- *df.groupBy*

- **Handle missing data**

- *Drop duplicate rows, drop rows with NA/null data*
- *Fill NA/null data*

## Actions

- Return values calculated from DataFrame

## Examples

- reduce, max, min, sum, variance and stdev

→ Distributed and parallel processing but it is done by the framework

# Example of a Spark program

```
#!/usr/bin/env python2
#encoding: UTF-8
# CS-E4640
import csv
import sys
from datetime import datetime
from pyspark.sql import SparkSession
import numpy as np
from pyspark.sql import functions as F
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--input_file', help='input data file')
parser.add_argument('--output_dir', help='output dir')
args = parser.parse_args()

##define a context
spark = SparkSession.builder.appName("cse4640-onu").getOrCreate()
#NOTE: using hdfs:///..... for HDFS file or file:///
inputFile = args.input_file
df = spark.read.csv(inputFile, header=True, inferSchema=True)
#df.show()
print("Number of records", df.count())
exprs = {"SPEEDIN": "avg"}
df2 = df.groupBy('ONUID').agg(exprs)
df2.repartition(1).write.csv(args.output_file, header=True)
```

Session/Driver



Read data



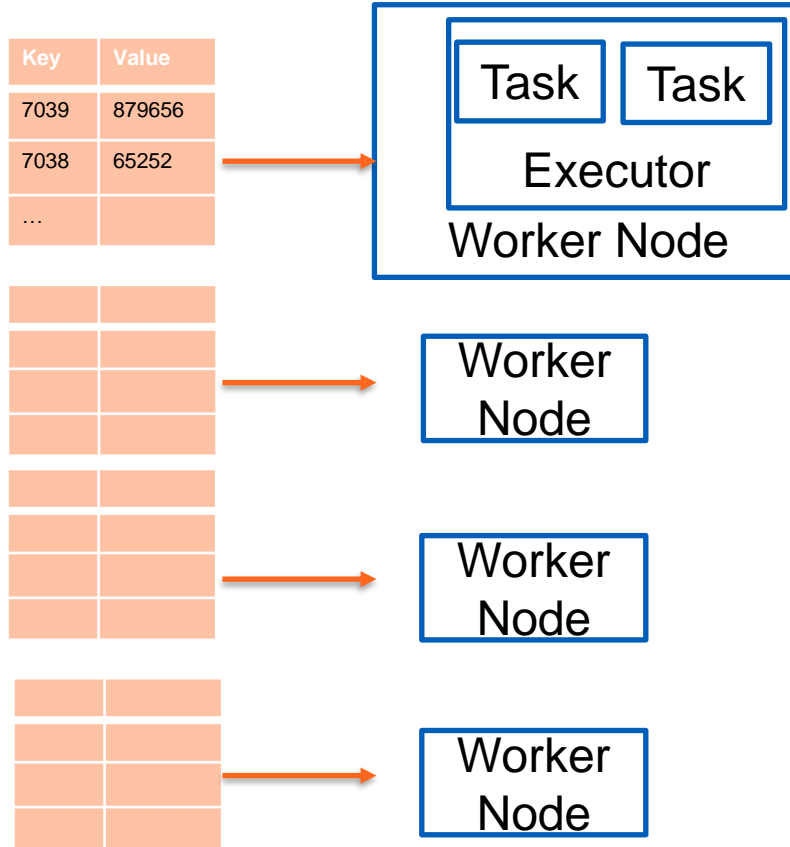
Apply operators



# Spark application runtime view

- **Tasks:**
  - A unit of work executed in an executor: e.g., set of transformations for a data partition
- **Stage**
  - A set of tasks executed in many nodes for computing the same operation
  - Move to a new stage: through shuffle operations
- **Job**
  - Runtime view of an action operator (produce a result), includes many stages

# Data Distribution



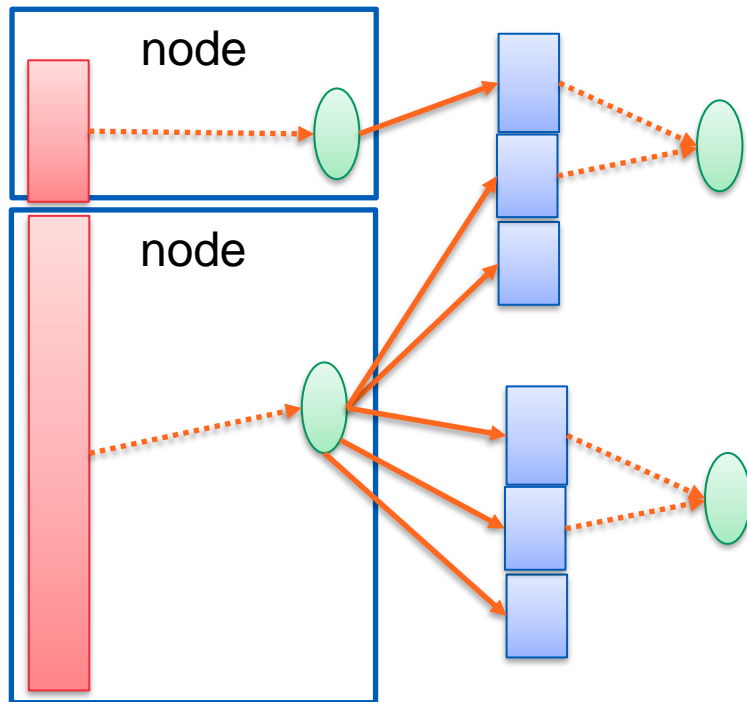
**One task works on a partition at a time**

**→ Parallelism and performance are strongly dependent on number of partitions, tasks, CPU cores**

# Data Distribution: Load balance

Imbalance  
processing

more data shuffle



- It is important to have well-balanced data distribution across nodes
- **Detection:**
  - look at runtime execution time to see problems or check your data
- **Examples of solution:**
  - Repartition
  - Broadcast
  - ~~Change group keys~~



# Pipelining, Shuffle and DAG

- **Operations work in a pipeline without moving data across nodes**
  - E.g., map->filter, select->filter
- **Shuffle persistent**
  - Shuffle needs move data across nodes
  - Source tasks save shuffle files into local disks for data shuffle, then the target tasks will read data from source nodes
    - *Save time, recovery, fault tolerance*

# Monitoring Spark: Executors and tasks



Jobs

Stages

Storage

Environment

Executors

SQL

cse4640-nytaxicount application UI

## Spark Jobs (?)

User: truong

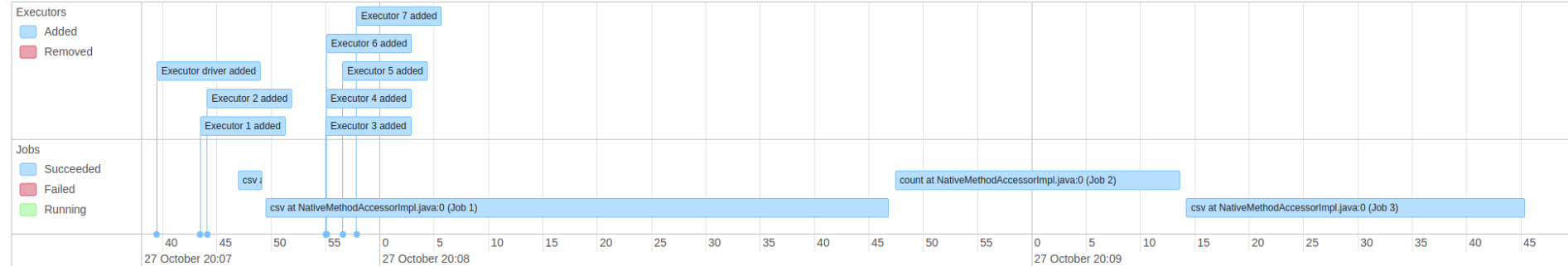
Total Uptime: 2.1 min

Scheduling Mode: FAIR

Completed Jobs: 4

▼ Event Timeline

▣ Enable zooming



## Completed Jobs (4)

Job Id ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2019/10/27 20:09:14	31 s	3/3	279/279
2	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2019/10/27 20:08:47	26 s	2/2	79/79
1	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2019/10/27 20:07:49	57 s	1/1	78/78
0	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2019/10/27 20:07:46	2 s	1/1	1/1

# Executors and tasks

Shuffle Write: 216.0 B / 3

▶ DAG Visualization

▶ Show Additional Metrics

▼ Event Timeline

☐ Enable zooming

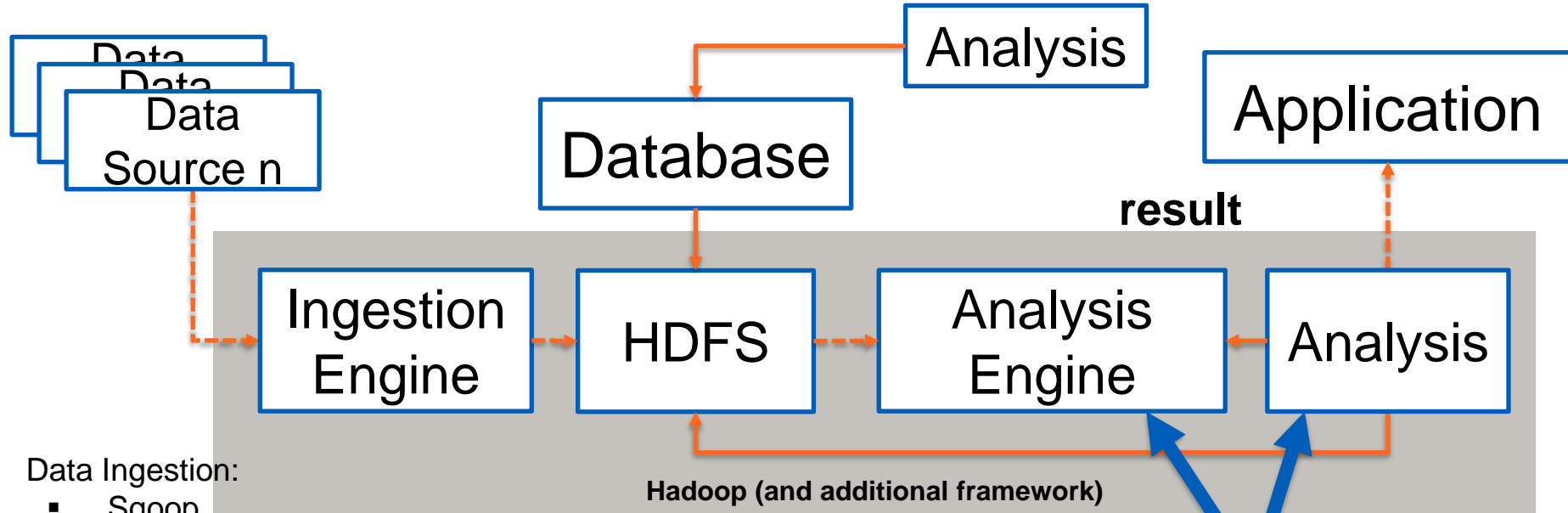
■ Scheduler Delay ■ Executor Computing Time ■ Getting Result Time  
■ Task Deserialization Time ■ Shuffle Write Time  
■ Shuffle Read Time ■ Result Serialization Time



# Other important support of Spark

- **MLlib - Machine learning**
  - Distributed and parallel machine learning algorithms with big data and clusters
- **Streaming: data processing in near-realtime**
  - *Related to our topic: stream data processing*
- **Graph Processing: Spark GraphX**
  - Parallel computation for graphs
- **Many third-party frameworks, e.g.,**
  - SparkOCR (<https://www.johnsnowlabs.com/spark-ocr/>),  
SparkNLP (<https://nlp.johnsnowlabs.com/>)

# Integration patterns: ETL and Analytics



## Data Ingestion:

- Sqoop
- HDFS Client/Hadoop Streaming
- Spark Streaming
- Apache Kafka
- Apache Nifi

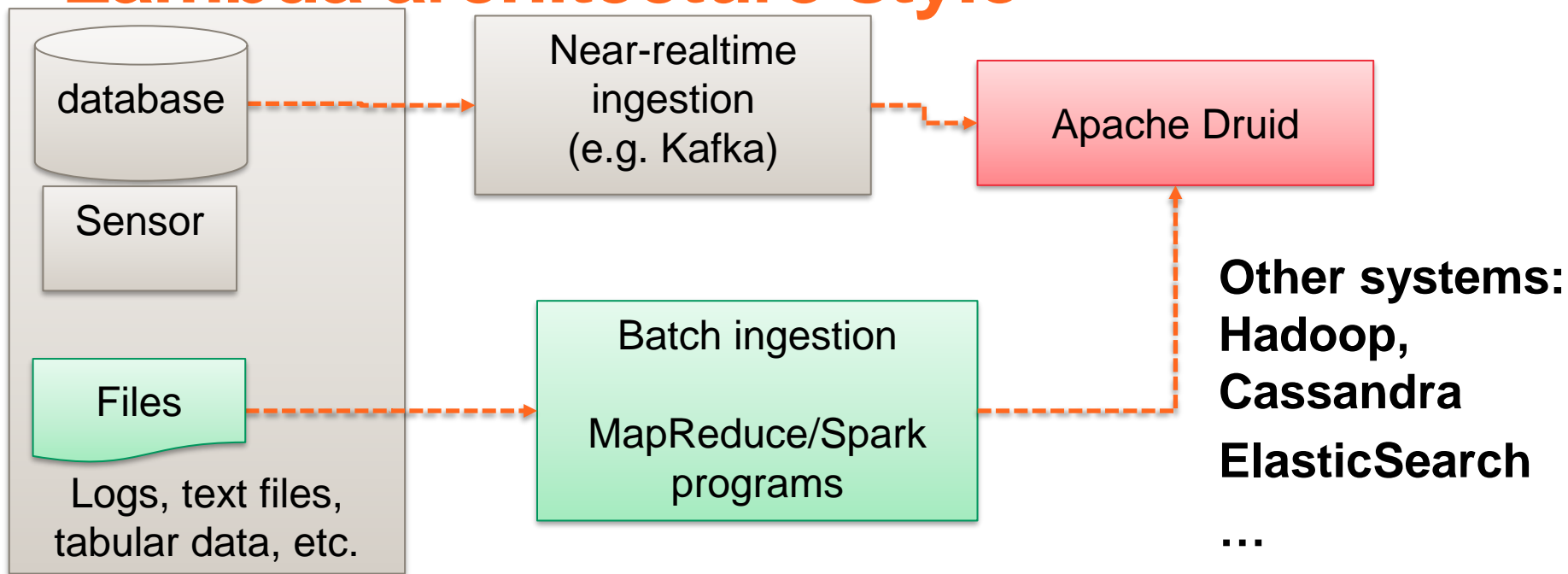
## HDFS as storage for databases

- Accumulo, Druid, etc.

## Computing/Data Processing Framework

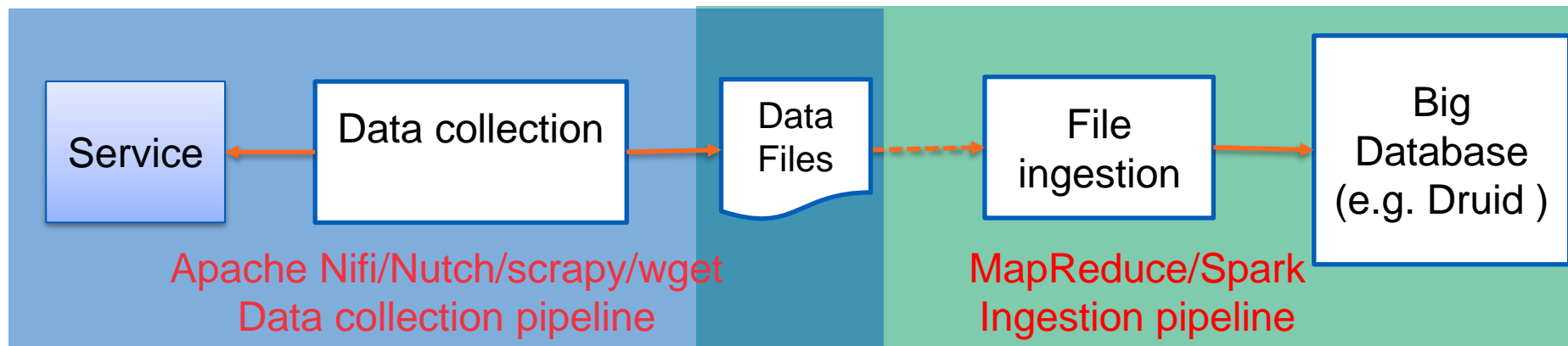
- Apache Spark
- Hadoop MapReduce
- Apache Tez

# Integration patterns: ingestion in Lambda architecture style



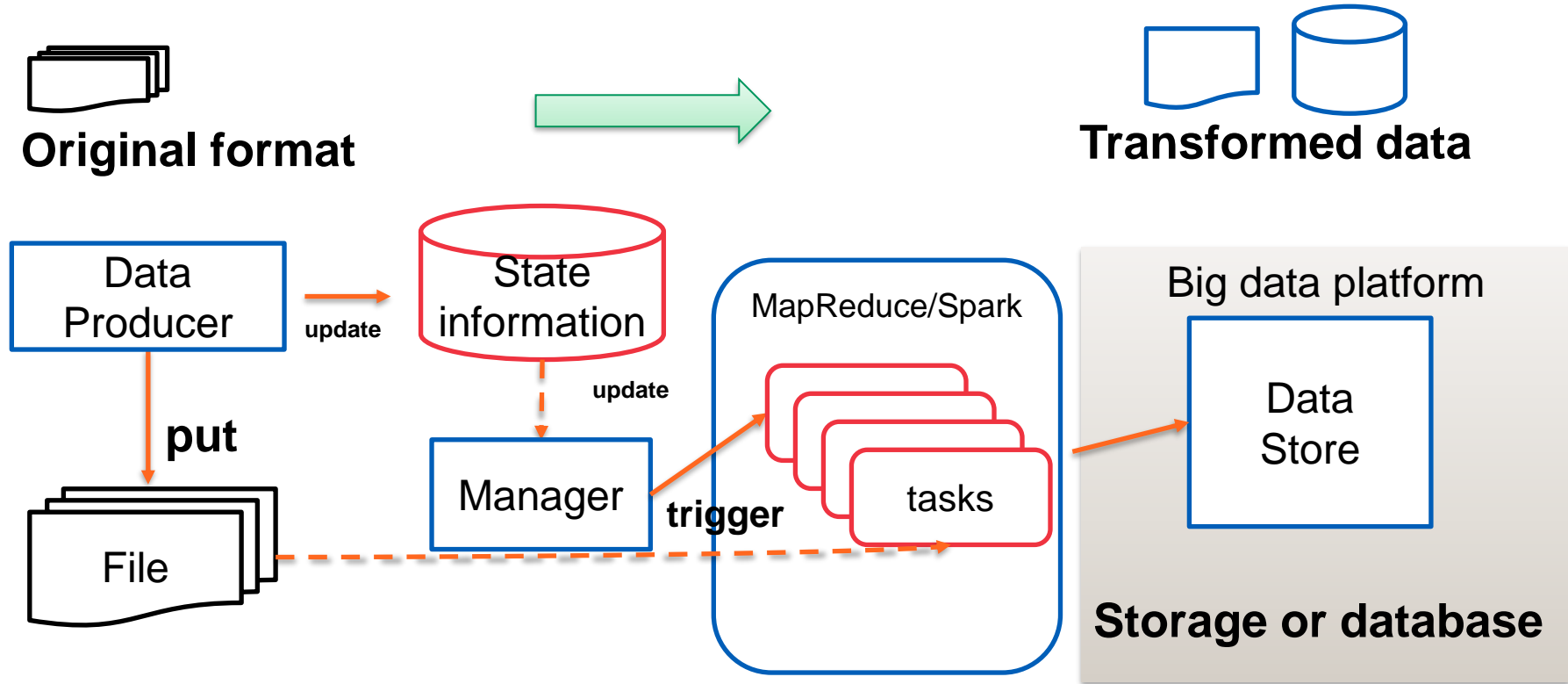
**More pipelines dealing with different sources**

# Integration patterns: Connecting different pipelines



**both pipelines and their connection are complex**

# Integration patterns: ingestion triggers





# Summary

- **Facts:**

- MapReduce and Spark are important frameworks
- A user/developer needs to learn to develop MapReduce/Spark applications
- A platform operator/provider offer services for managing resources, processing and monitoring

- **Thoughts:**

- Think about the success of Apache Spark: rich ecosystems!
- Think if you combine data, different distributed programming supports for your big data platform

# Thanks!

**Hong-Linh Truong**  
**Department of Computer Science**

**rdsea.github.io**