Aalto University
School of Science

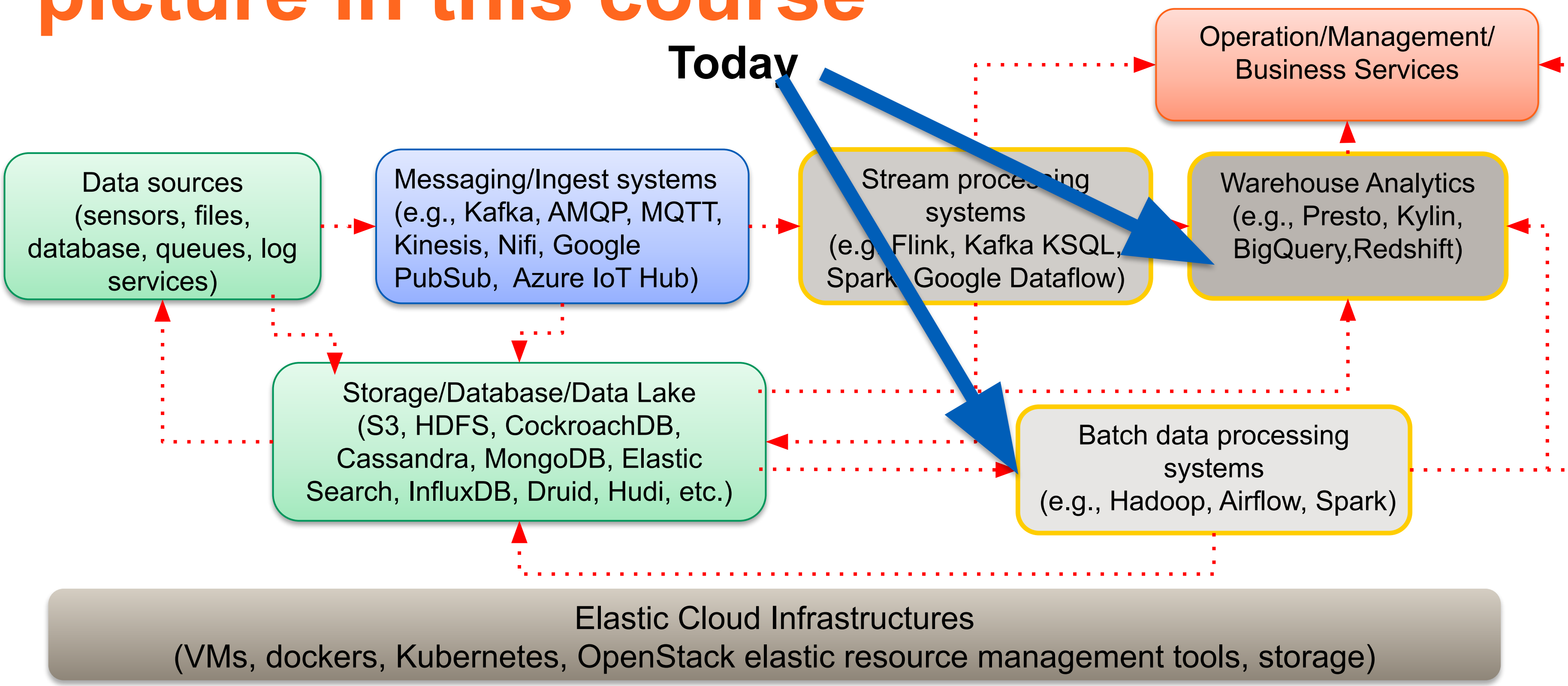
Big Data Processing - The Spark Programming Model

Hong-Linh Truong
Department of Computer Science
linh.truong@aalto.fi, https://rdsea.github.io

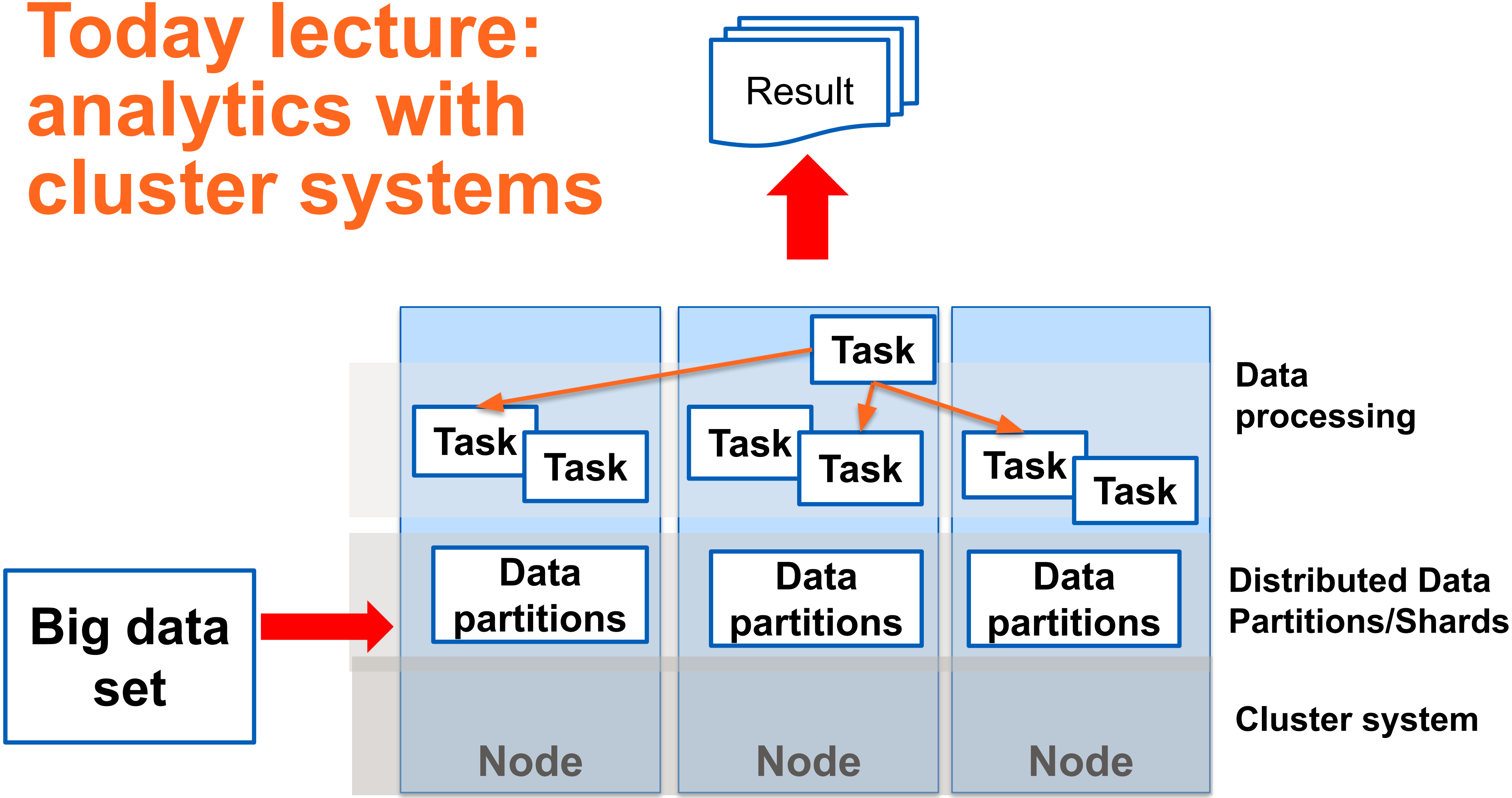
Learning objectives

- Be familiar with big data processing models using multiple nodes/clusters
- Understand the Spark programming model for big data processing
- Able to perform practical programming features with Apache Spark
- Able to design and apply Spark data processing for data in lake storage

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



Today lecture: analytics with cluster systems



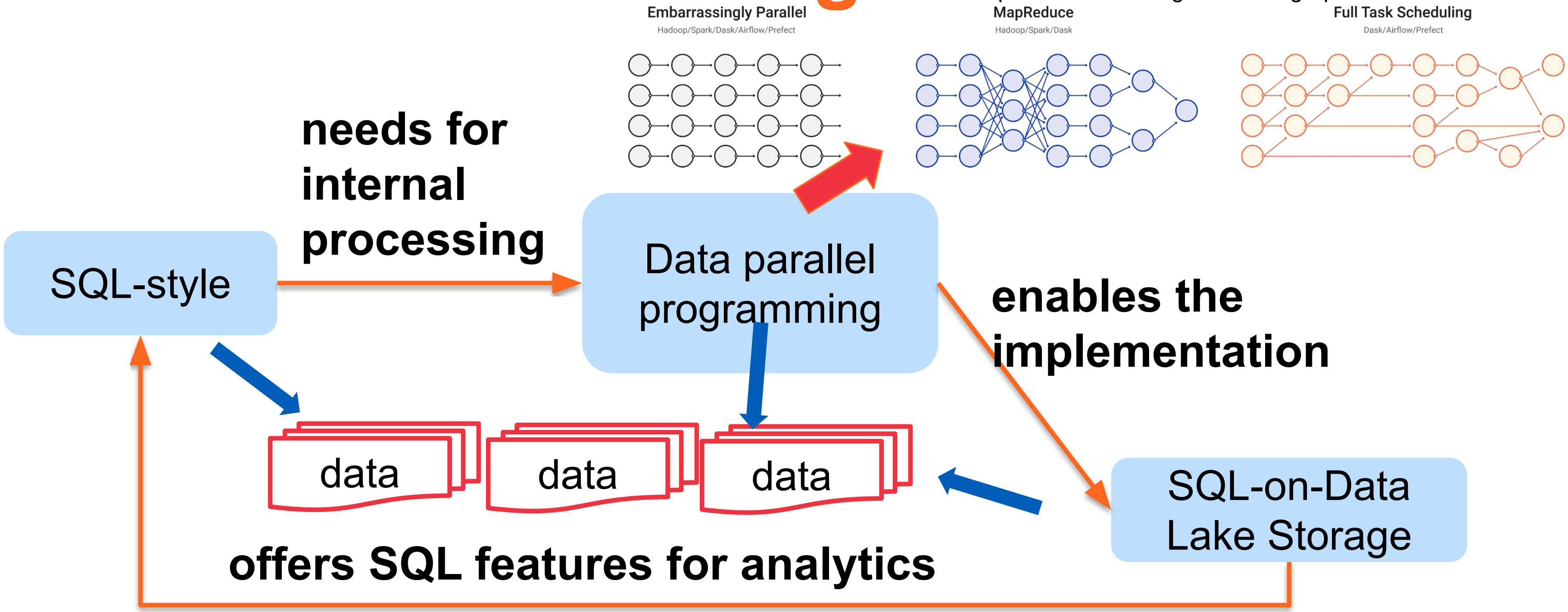
Our first focus: big data analytics for data at rest

Recall: Data at rest

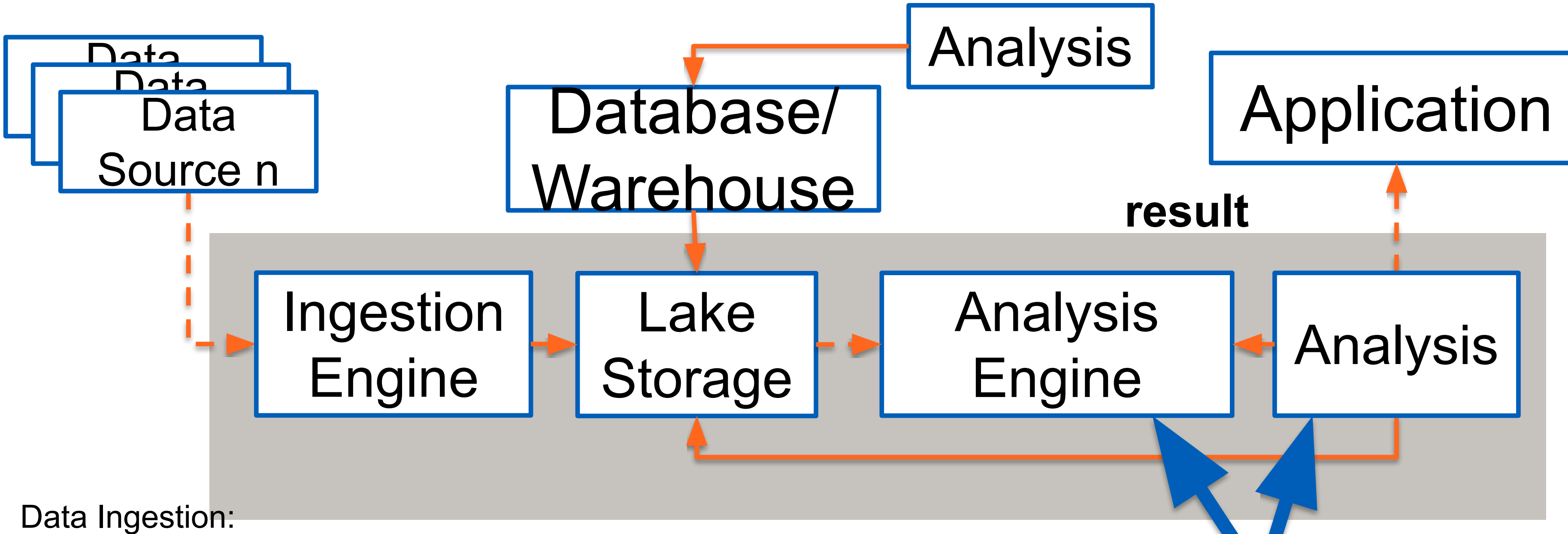
- **At rest**
  - distributed file systems/object storages
    - *in big data we have a lot of files with different data formats*
  - data in a set of databases
- **Multiple types of big data analytics with high concurrent/parallel data writes/reads**
- **Dealing with different data access/analytics frequencies:**
  - e.g., data organized into **hot, warm and cold data**

SQL-style/data parallel processing for data in lake storage

Figure source:
https://docs.dask.org/en/stable/graphs.html



ETL and Analytics with Lake Storage



- Data Ingestion:
  - Spark Streaming
  - Kafka Connect
  - Apache Nifi
- HDFS, AWS S3, Google Storage, Azure Data Lake Storage, etc., as storage
- Computing/Data Processing Framework
  - Apache Spark
  - Hadoop MapReduce

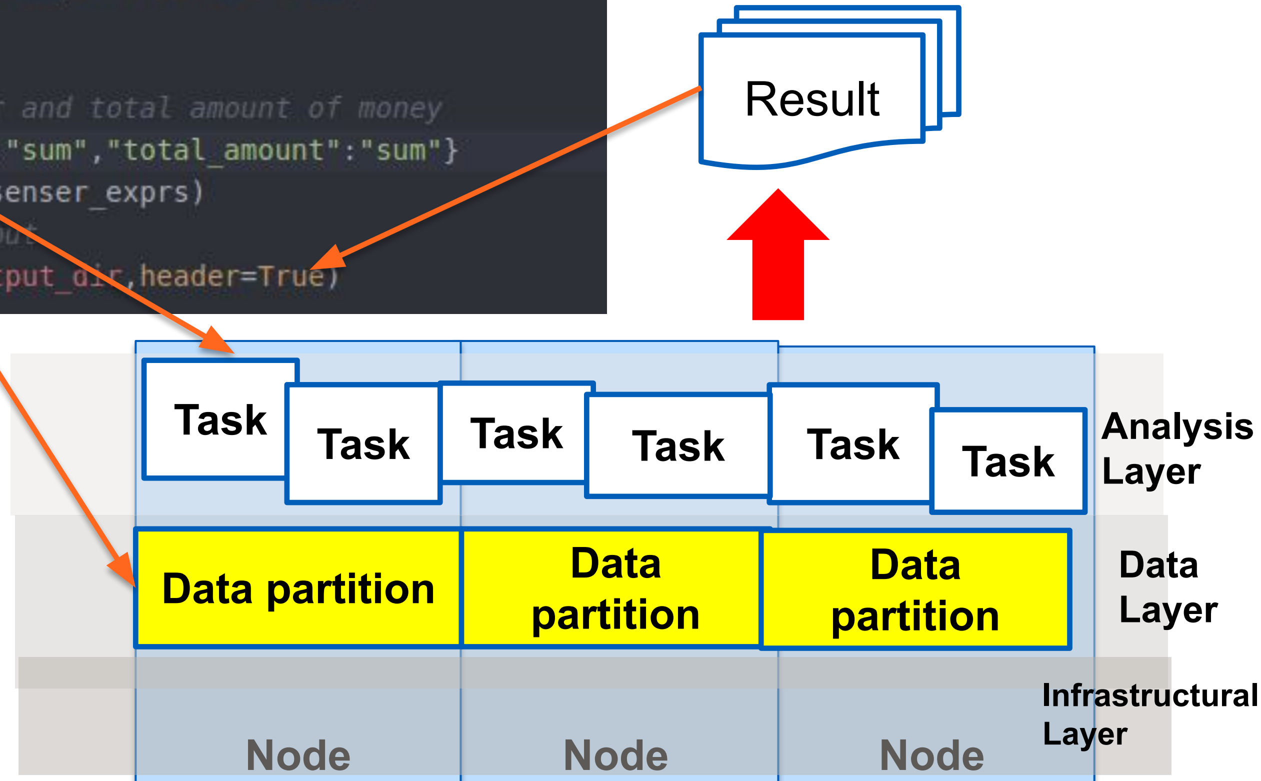
DataFrame/Table view of data

Example taxi records: named columns

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
- Streaming data can be also represented as **unbounded tables**

```
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
passenser_exprs = {"passenger_count":"sum","total_amount":"sum"}
df2 = df.groupBy('VendorID').agg(passenser_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
  - (Distributed) SQL-style processing
- **Studied frameworks**
  - Apache Hadoop/Spark, Dask
  - Apache Airflow
- **Not in our focus:**
  - Bulk synchronous parallel (BSP)
  - HPC MPI (Message Passing Interface)

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**
  - public cloud: Google DataProc, Azure HDInsight, Amazon EMR
  - data lake systems: e.g., Hudi and Delta Lake

Apache Spark

Can be run a top

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

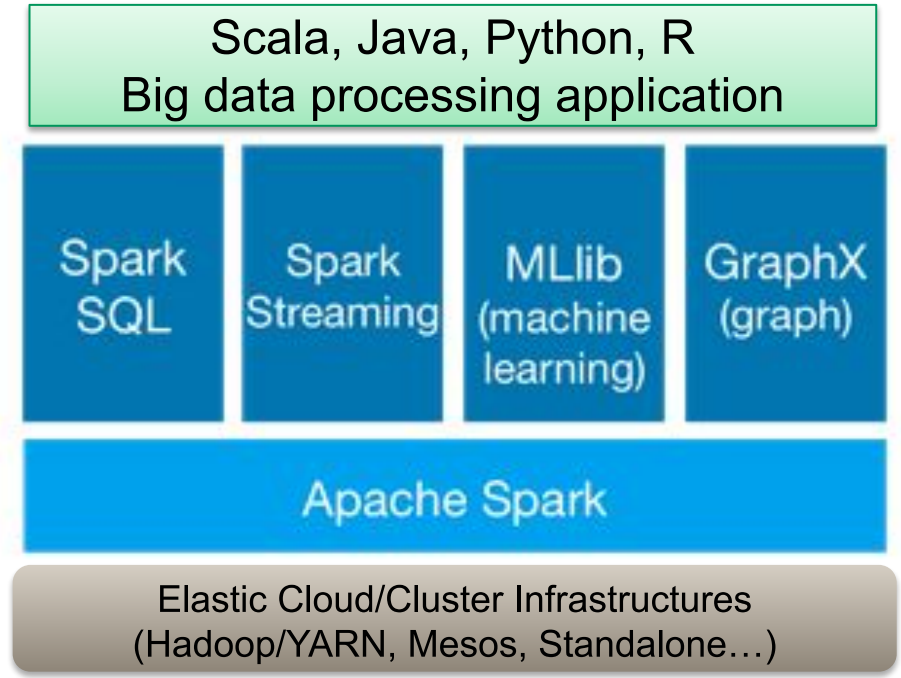
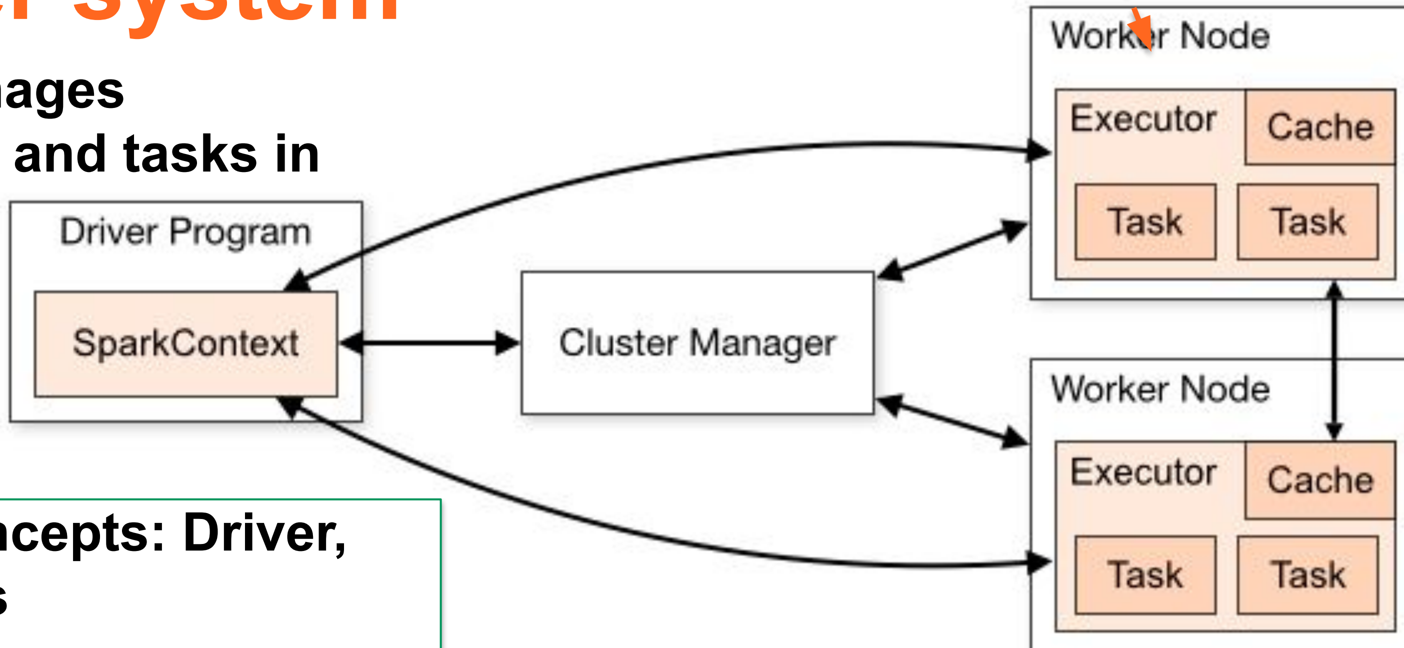


Figure source: http://spark.apache.org/

Execution model in a cluster system

Computing resources in a cluster node

Driver manages operations and tasks in nodes

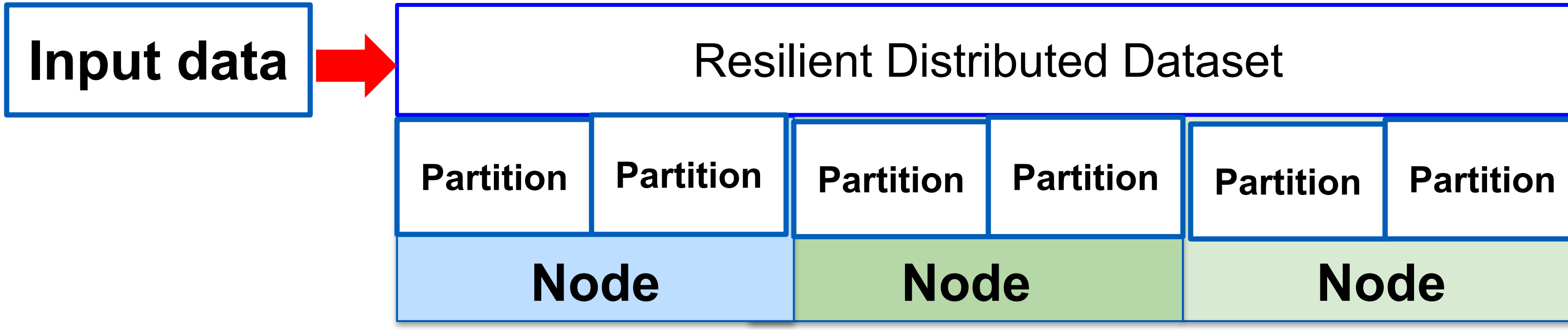


Common concepts: Driver, Nodes, Tasks

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

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

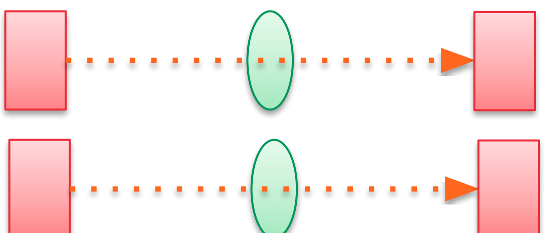
Key features



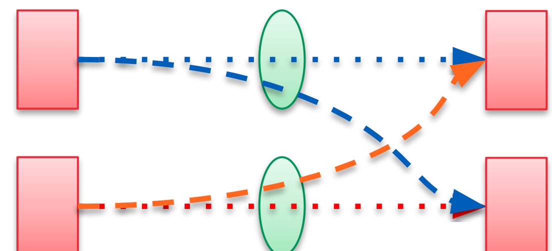
- **Input data is distributed in different nodes for processing**
  - Support partitions for data processing: a node keeps one or n partitions, a partition resides only in a node ⇒ for computing
- **Key operations: transformations and actions on data**
- **Leverage parallel computing concepts to run multiple tasks**
  - Operation -> task executed by executor
  - Parallel tasks, task pipeline, DAG of processing stages
- **Persistent data in memory/disk for operations**

Transformation operations

- **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**



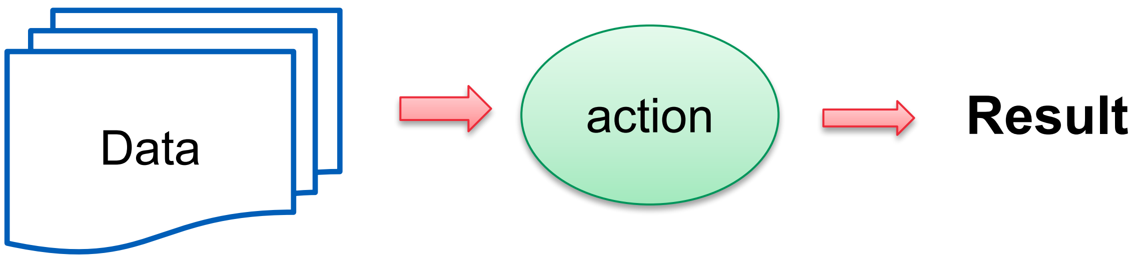
Narrow transformation, no data shuffle



Wide transformation, cross data partitions, requires a shuffle

Action operations

- **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 operations ⇒ enable various types of optimization

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**

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 transformations and actions operations**
  - *transformations:* build plans for transforming data models
  - *actions:* perform computation on data

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()

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)
```

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 variable/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("cse4640-broadcast").setMaster(args.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**

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

```python
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
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	39	10[redacted]07039	01/08/2019 00:04:07	148163	49018
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	38	10[redacted]07038	01/08/2019 00:04:07	1658	1362
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	9	10[redacted]07009	01/08/2019 00:04:07	6693	5185
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	8	10[redacted]07008	01/08/2019 00:04:07	640	544
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	11	10[redacted]07011	01/08/2019 00:04:07	118	114
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	10	10[redacted]07010	01/08/2019 00:04:07	28514	12495
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	13	10[redacted]07013	01/08/2019 00:04:07	868699	23400
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	15	10[redacted]07015	01/08/2019 00:04:07	1822	1120
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	17	10[redacted]07017	01/08/2019 00:04:07	998069	117345
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	16	10[redacted]07016	01/08/2019 00:04:07	22402	1804
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	19	10[redacted]07019	01/08/2019 00:04:07	640	791
[redacted]YN	1[redacted]3023	[redacted]760	1	1	10	49	10[redacted]10049	01/08/2019 00:04:07	662	494
[redacted]YN	1[redacted]3023	[redacted]760	1	1	10	48	10[redacted]10048	01/08/2019 00:04:07	2158	759
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	21	10[redacted]07021	01/08/2019 00:04:07	0	0
[redacted]YN	1[redacted]3023	[redacted]760	1	1	10	51	10[redacted]10051	01/08/2019 00:04:07	2600890	54153
[redacted]YN	1[redacted]3023	[redacted]528	1	2	7	20	10[redacted]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, cloud object storage (AWS S3, Google Cloud Storage, Azue Blob Storage), local files, etc.

Figure 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

```
#!/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

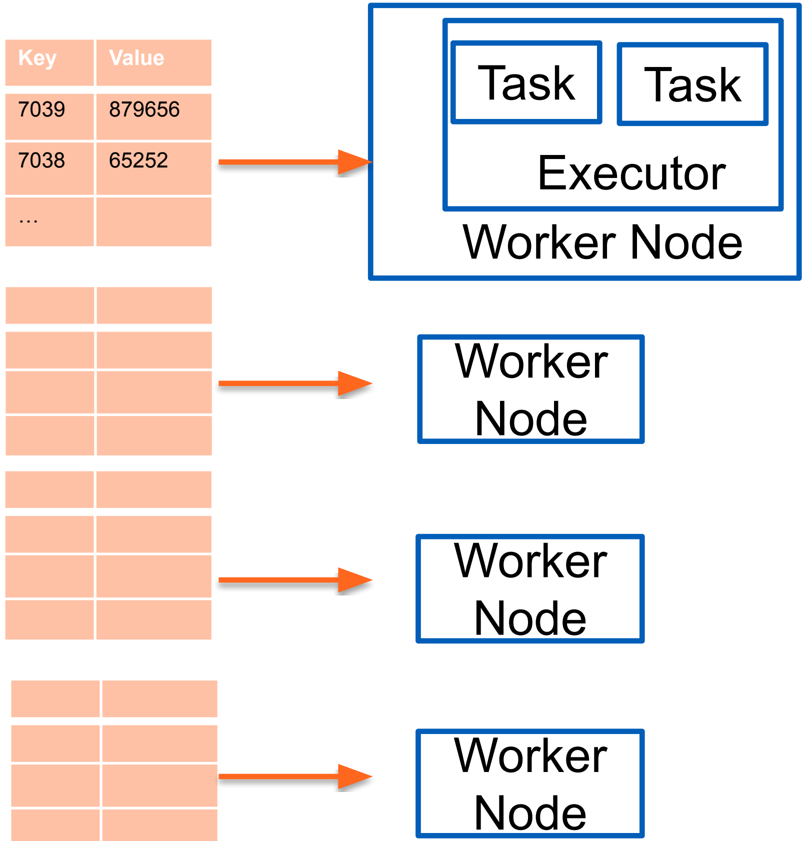
Read data

Apply operations

Spark application runtime view

- **Tasks:**
  - a unit of work executed in an executor: e.g., performing transformations of a data partition
- **Stage: Shuffle Map Stage & Result Stage**
  - a set of tasks executed in many nodes for performing the same operation
  - move to a new stage: through a shuffle to produce output partitions or an action to produce results
- **Job**
  - runtime view of an action operation (actual computation produces a result), includes many stages of tasks

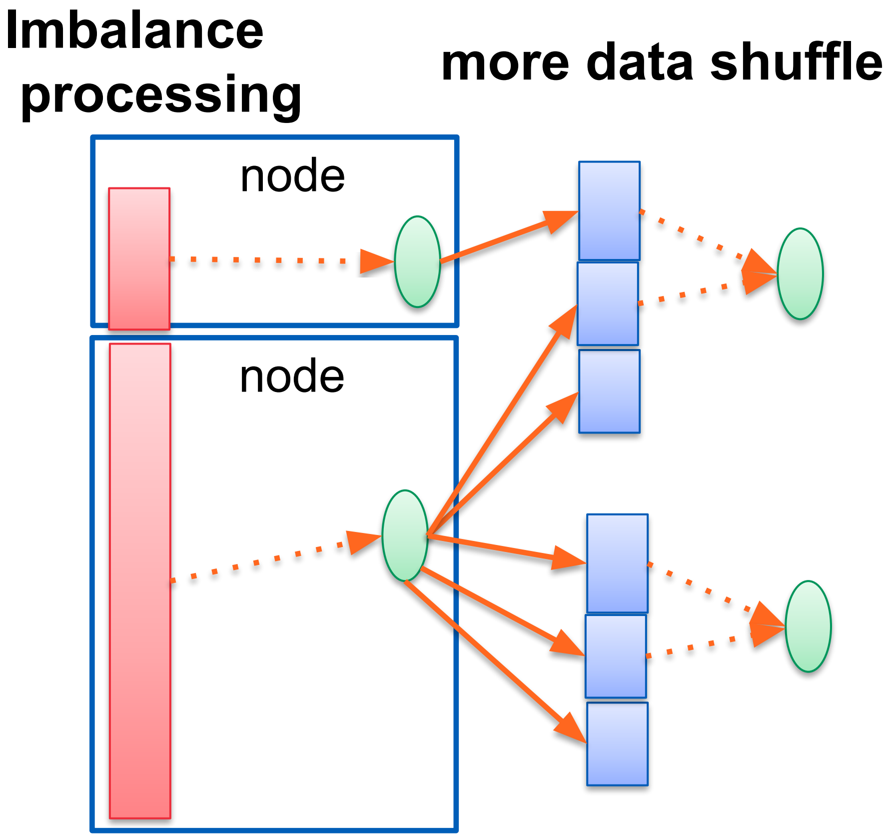
Data Distribution



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



- **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
  - 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*

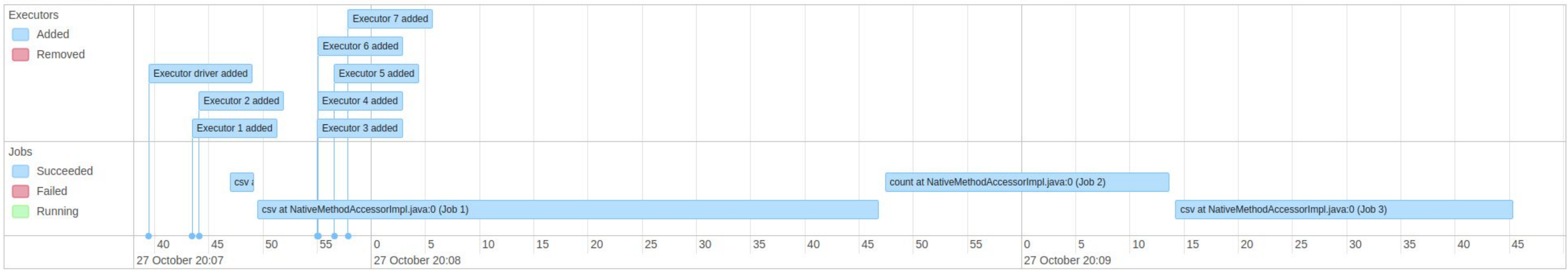
Monitoring Spark: Executors and tasks

Spark 2.3.3 | 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

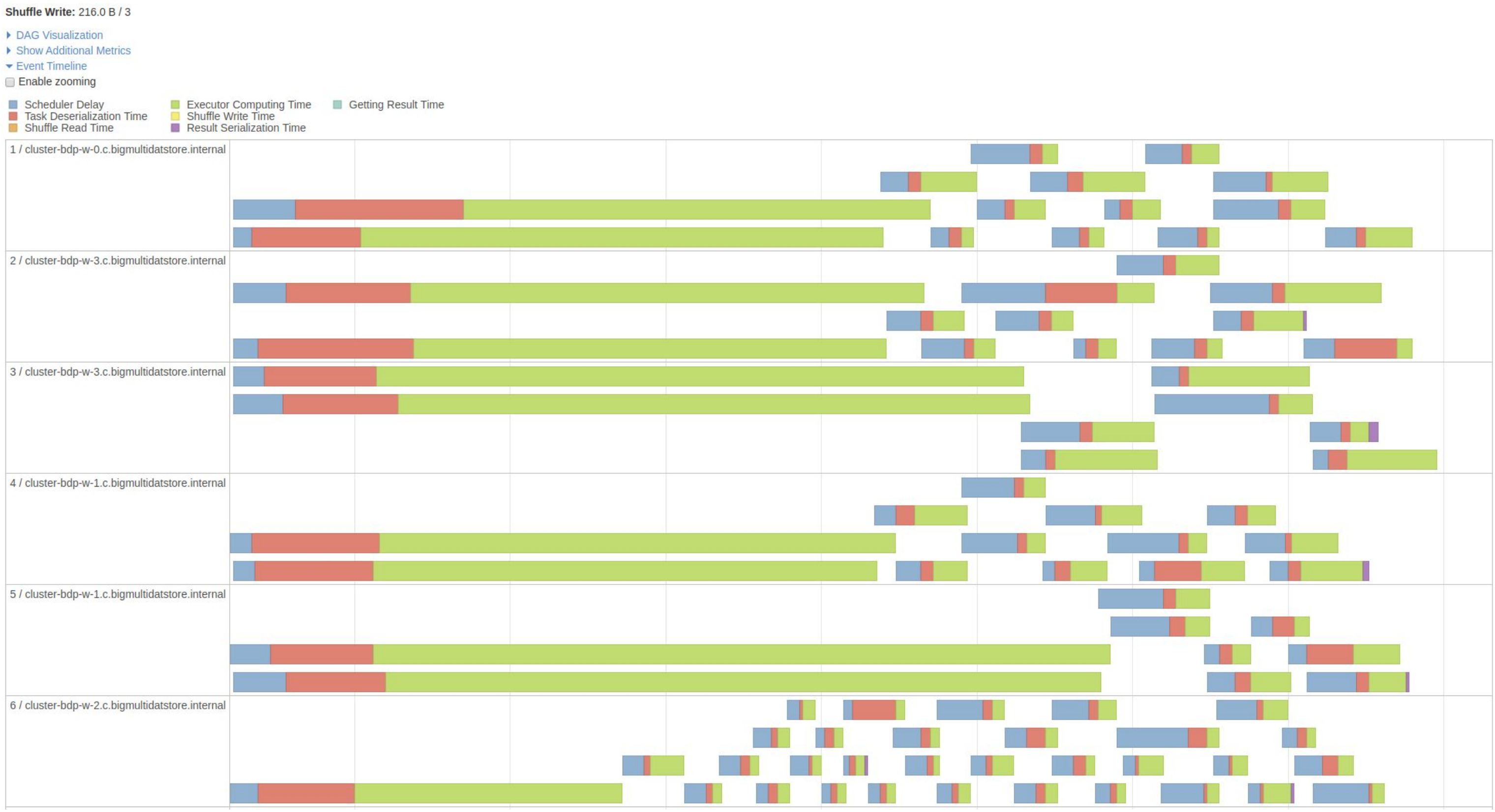
▾ Event Timeline
☐ Enable zooming



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



Other important support of Spark

- **MLlib - Machine learning**
  - Distributed and parallel machine learning algorithms with big data and clusters
- **Streaming: data processing in near real-time**
  - *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/)
  - PyDeequ (https://pydeequ.readthedocs.io/en/latest/README.html#) Data quality
    - check our example: https://github.com/rdsea/bigdataplatforms/tree/master/tutorials/dataquality

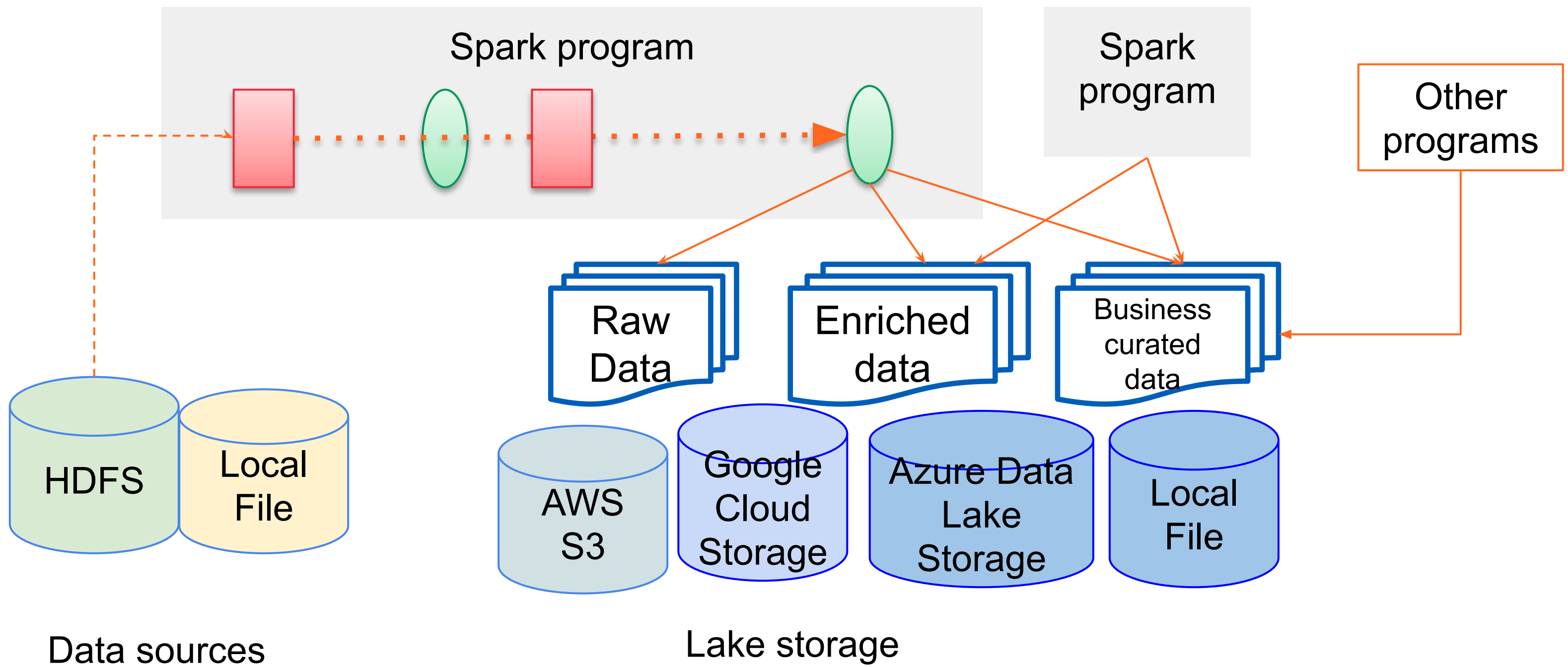
Spark as a key programming model/analytic engine for Data Lake

- **Modern lake data: cloud or on-premise**
  - *multiple types of data* from different sources (databases, files, sensors, etc)
  - *different forms in storage*: raw data, enriched/processed/cleansing, application-/business curated data, sandbox data (for testing, collaboration)
  - *common, standard, cost optimal storage*: object storage (S3, Azure), (distributed) file storage (Hadoop FS), ...
- **Data Lake Core**
  - Data tables, metadata and catalogs
  - Open standards: Parquet, ORC, Iceberg tables, Delta Lake formats
  - Many processing and governance tasks

Spark as a key programming model/analytic engine for Data Lake

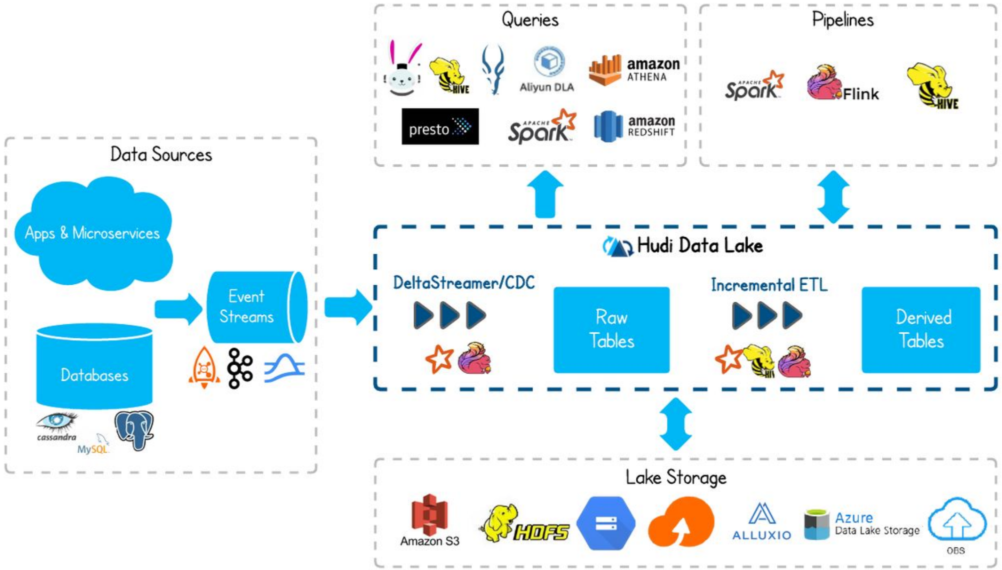
- **Many tasks required:**
  - Ingestion (insert/update)
  - Transformation
  - Query
  - Quality controls
- **Spark as an important engine**
  - for batch processing and stream processing (next lecture)
  - deal with different data formats
  - work with different lake storage
  - still in the same framework
- Core engine for Data lake platforms: Apache Hudi, Delta Lake

Spark programs for ingestion/analytics of lake data



Example: Hudi Data Lake

Figure source: https://hudi.apache.org/



different analytics

lake core

different storage

Problems: hard to govern, technology-centric, lack of considerations of data products, cost of effort, etc.

Example

Spark program with Spark Delta for processing data and store the processed data into a cloud data lake storage

E.g., Data lake storage based on Google Cloud Storage

```
## hadoop inputFile="hdfs://"
spark_df =spark.read.csv(inputFile,header=True,inferSchema=True)
print(spark_df.head(10))
#do many things, before producing data for datalake
spark_df.write.format("delta").mode("append").save(lake_table_path)
```

A program to read and write data from/to the same lake (delta-rs package, not Spark)

```
if args.read_only !="yes":
    # read data from csv file, no error checking
    df = pd.read_csv(args.input_file)
    write_deltalake(args.lake_table_path, df)
# Read from the lake and print out the first 100 entries
# Load data from the delta table
lake_table_data= DeltaTable(args.lake_table_path)
df_result = lake_table_data.to_pandas()
print(df_result.head(100))
```

Summary

- **Facts:**
  - Spark is an important framework
  - A user/developer needs to learn to develop 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