Aalto University
School of Science

Programming Models for Big Data Processing

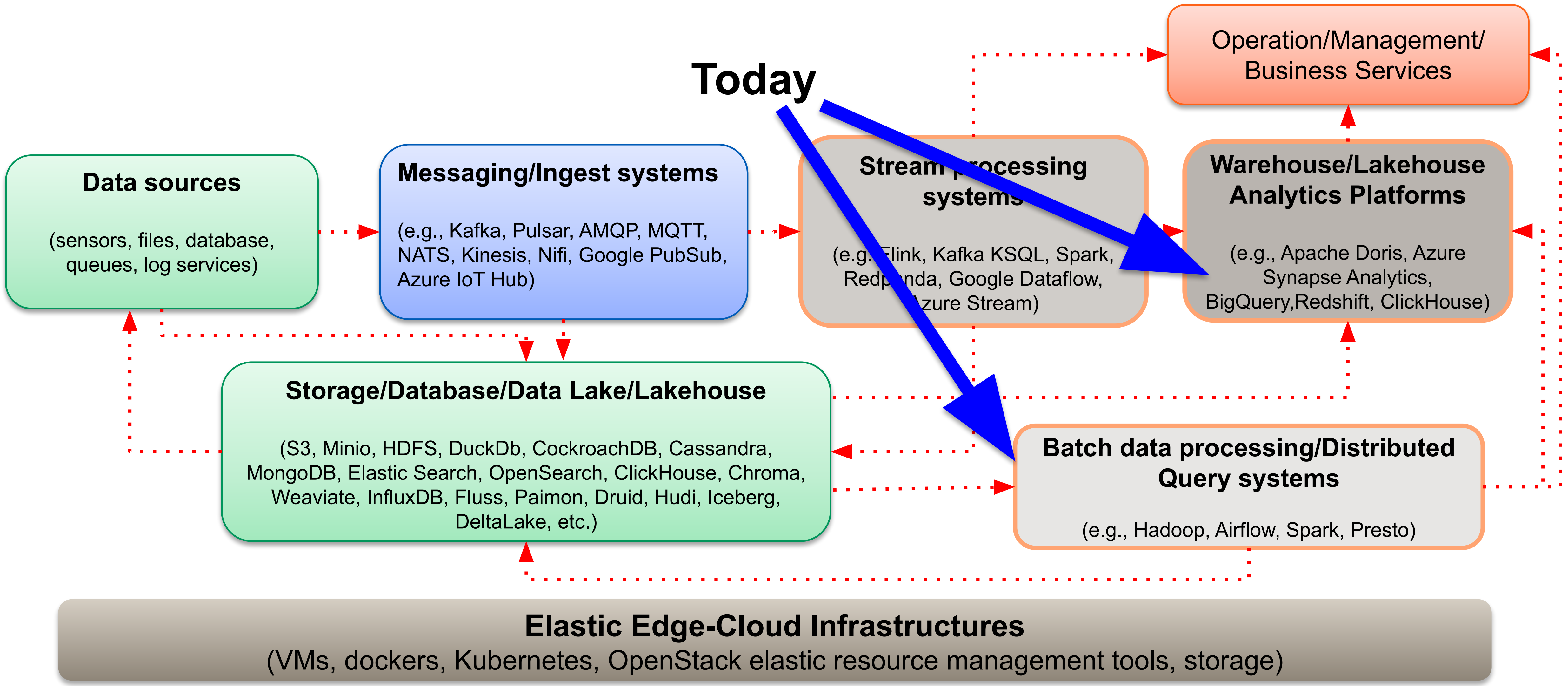
Hong-Linh Truong
Department of Computer Science
linh.truong@aalto.fi

11.2.2026

Learning objectives

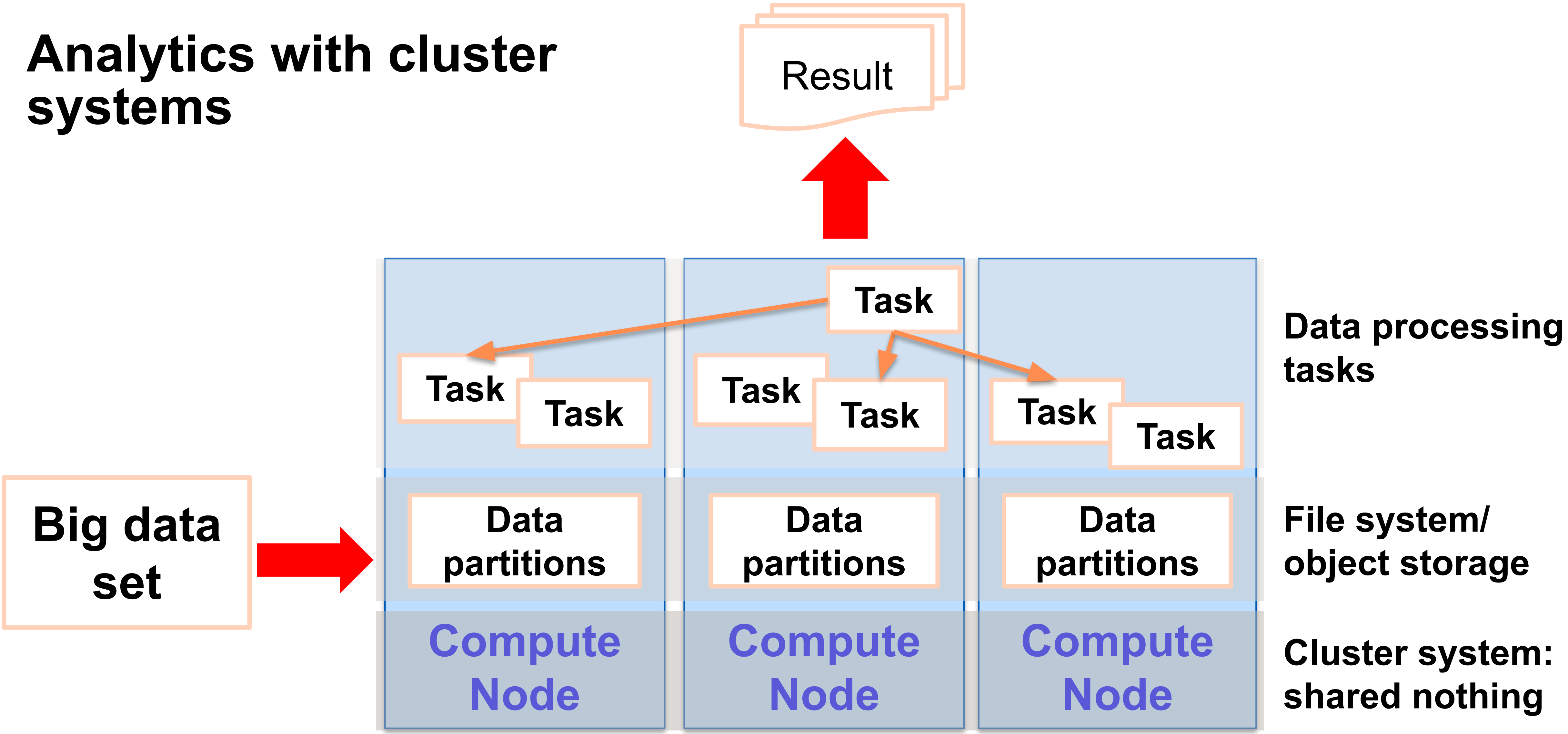
- Be familiar with key processing models and common techniques using multiple nodes/clusters for data processing
- Understand programming models and supports in Dask and Spark for data processing
- Able to perform practical programming features for data ingestion, transformation and analysis

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



Understanding common aspects

Analytics with cluster systems



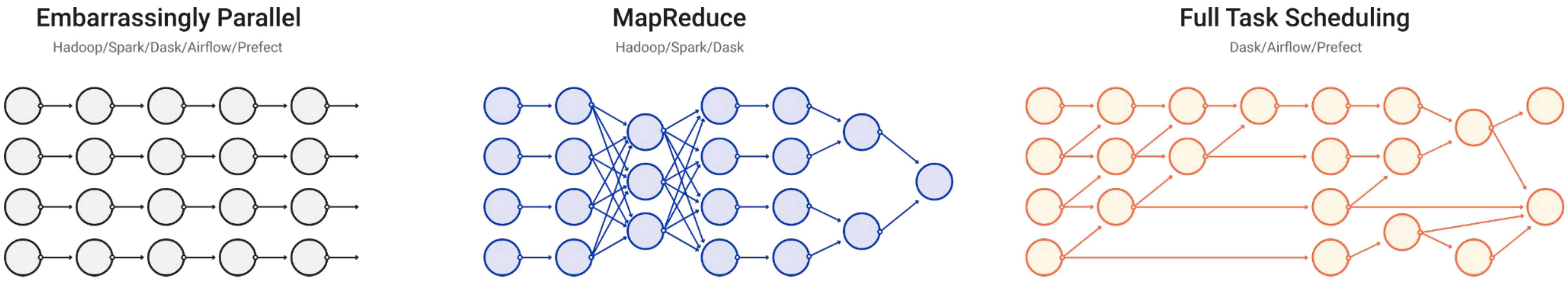
Compute Node: many cores with CPU/GPU

Parallel and distributed processing of data from distributed file systems/storage

- Process distributed data with different data formats
  - multiple types of data transformation/analytics with high concurrent/parallel data writes/reads
- Explore parallel and concurrent processing at different levels
  - data organization: different data access/analytics frequencies, e.g., data organized into hot, warm and cold data
  - individual data collection: items in a collection, e.g., a set of data files/tables, can be processed in parallel
  - parts of individual data file/table can be processed in parallel
- Leverage multiprocessing features from modern compute resources to speed up data processing
  - multi/many-cores and accelerators

Parallel and distributed data processing models

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



Choosing suitable programming models is based on use cases and ecosystem!

Using general purpose programming languages: 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

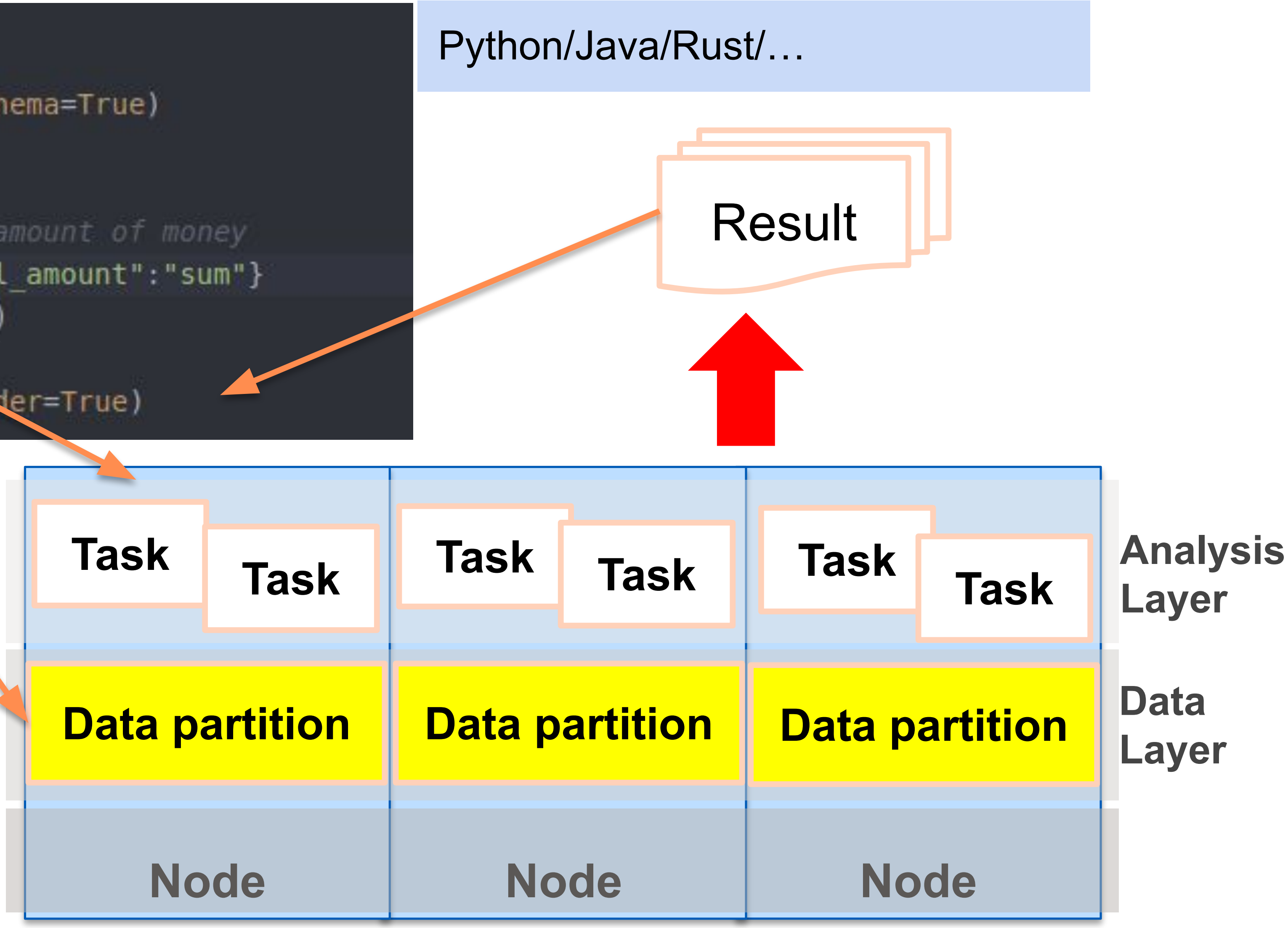
- Very common we analyze big data files based on this view
- Streaming data can be also represented as unbounded tables

pandas *https://pandas.pydata.org/docs/*

Polars *https://github.com/pola-rs/polars*

MODIN *https://github.com/modin-project/modin*

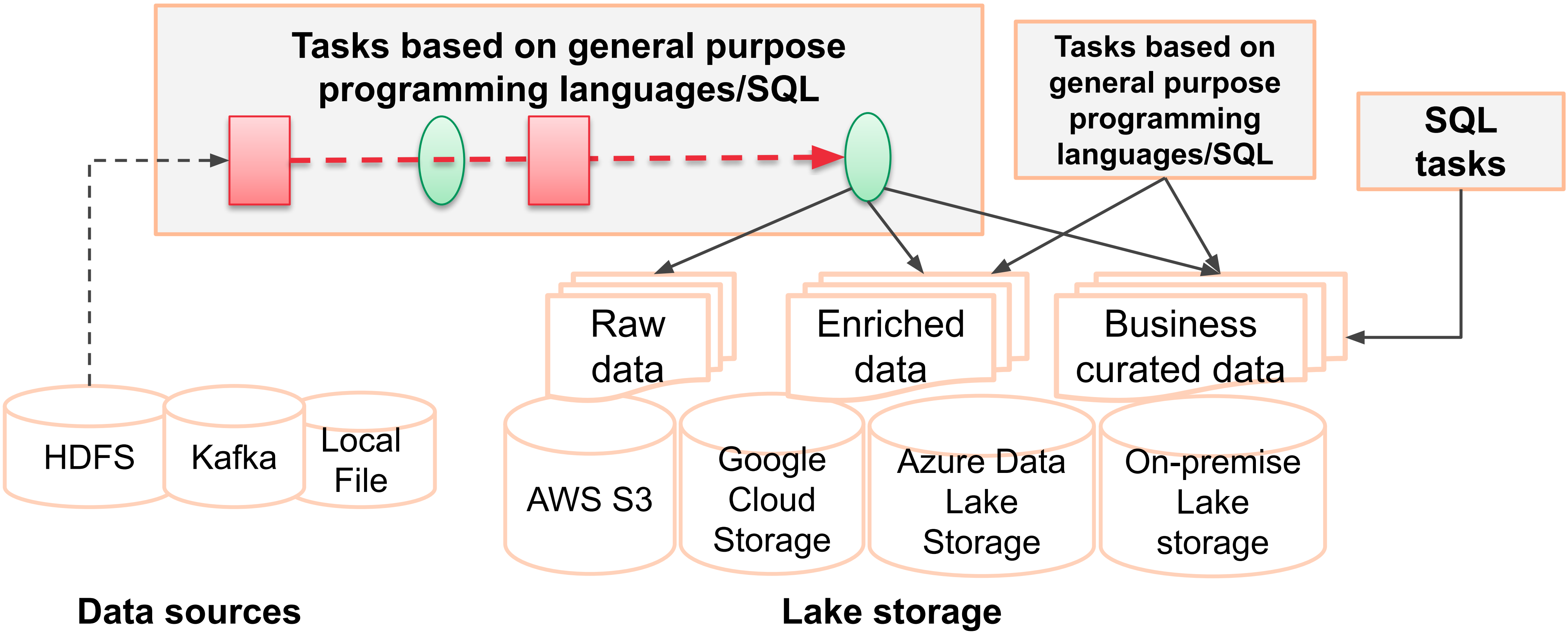
```python
inputFile =args.input_file
## hadoop inputFile="hdfs://"
df =spark.read.csv(inputFile,header=True,inferSchema=True)
#df.show()
print("Number of trips", df.count())
#number of passenger count per vendor and total amount of money
passenser_exprs = {"passenger_count":"sum","total_amount":"sum"}
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)
```



Using general purpose programming languages:

what we want when developing analysis programs for big data

General purpose programming languages +SQL for Data Lake/Lakehouse



Example with writing data to data lakes

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 (https://delta.io/)

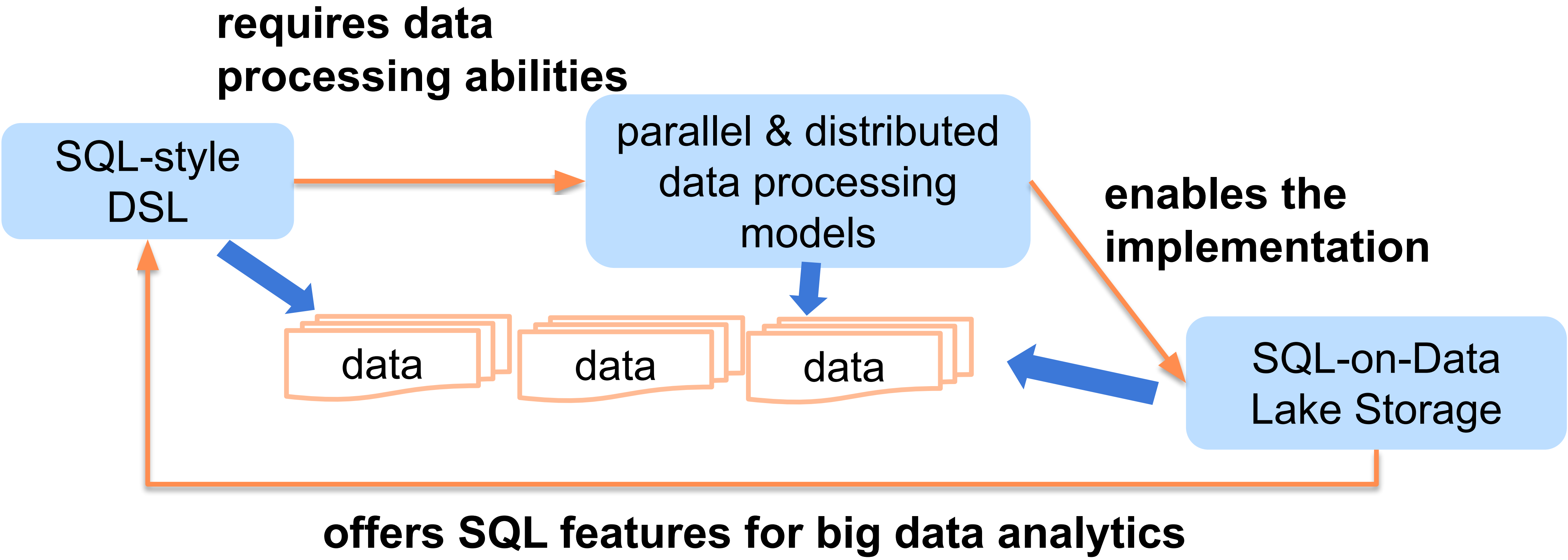
```
## hadoop inputFile="hdfs://"
spark_df =spark.read.csv(inputFile,header=True,inferSchema=True)
print(spark_df.head(10))
#do many things, before producing data for datalake
spark_df.write.format("delta").mode("append").save(lake_table_path)
```

Data lake storage with Iceberg tables, Pyarrow an PyIceberg
(https://py.iceberg.apache.org/)

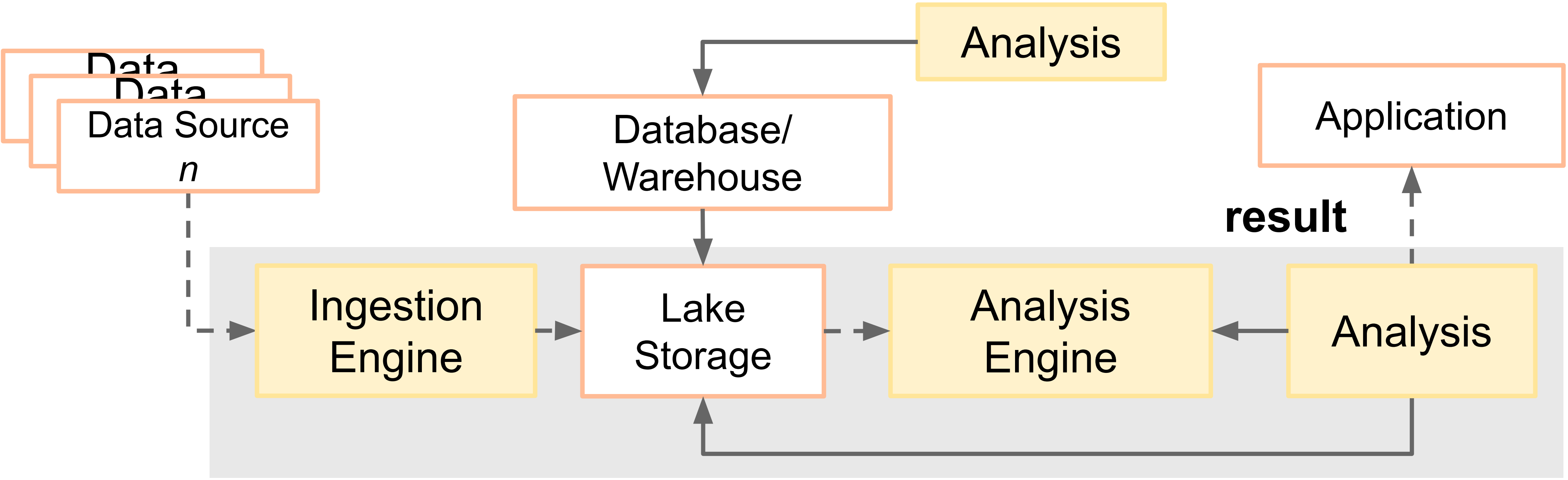
```
catalog = SqlCatalog(
    catalog_name,
    **catalog_config["catalog"][catalog_name],
)

if data_type == ".parquet":
    df = pq.read_table(input_data)
else:
    df = csv.read_csv(input_data)
catalog.create_namespace_if_not_exists(namespace)
logger.info(f'Existing namespaces: {catalog.list_namespaces()}')
full_tablename=f'{namespace}.{table_name}'
if not catalog.list_namespaces((namespace)):
    catalog.create_namespace(namespace)
table = catalog.create_table_if_not_exists(
    full_tablename,
    schema=df.schema,
)
table.append(df)
```

Enabling SQL-style with parallel/distributed data processing



Where is distributed/parallel data processing needed?



Data Ingestion:

- Spark Streaming
- Kafka Connect
- Apache Nifi
- etc.

Storage:

- HDFS, AWS S3, Google Storage, Azure Data Lake Storage, Iceberg tables, etc., as storage

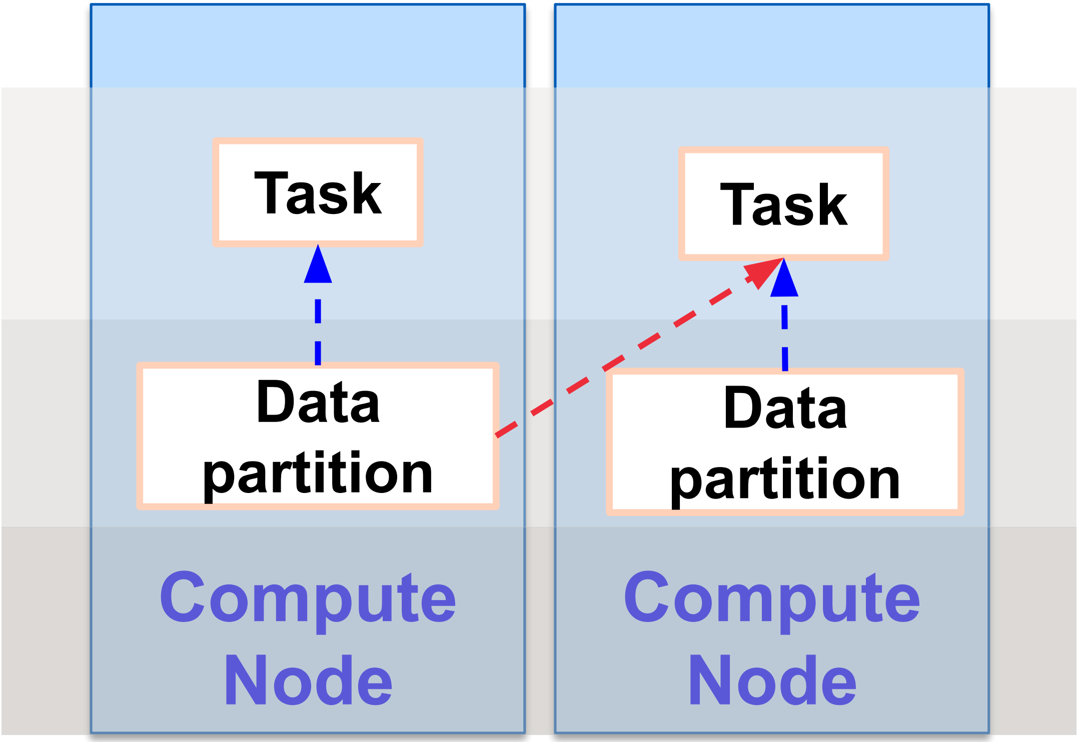
Programming Frameworks for data processing:

- Apache Spark
- Hadoop MapReduce
- Dask, Ray, etc.

Common principles and techniques

- Data input/output connectors
  - for reading data from sources and writing data into data sinks
- Data collections as abstract (big/distributed) data structures
  - for modeling/representing data in suitable views for processing
- Data partitioning
  - in the view of processing, a similar principle in data storage
- Data operations
  - operations applied to data in data collections
- Execution models
  - tasks and workflows
  - job scheduling; lazy vs eager execution; future execution
- Task fault-tolerance and data exchange among tasks in distributed processes/machines

Data Shuffle



Which operation/analysis could lead to the data shuffle?

Shuffling data from one node to another node for another task is expensive!

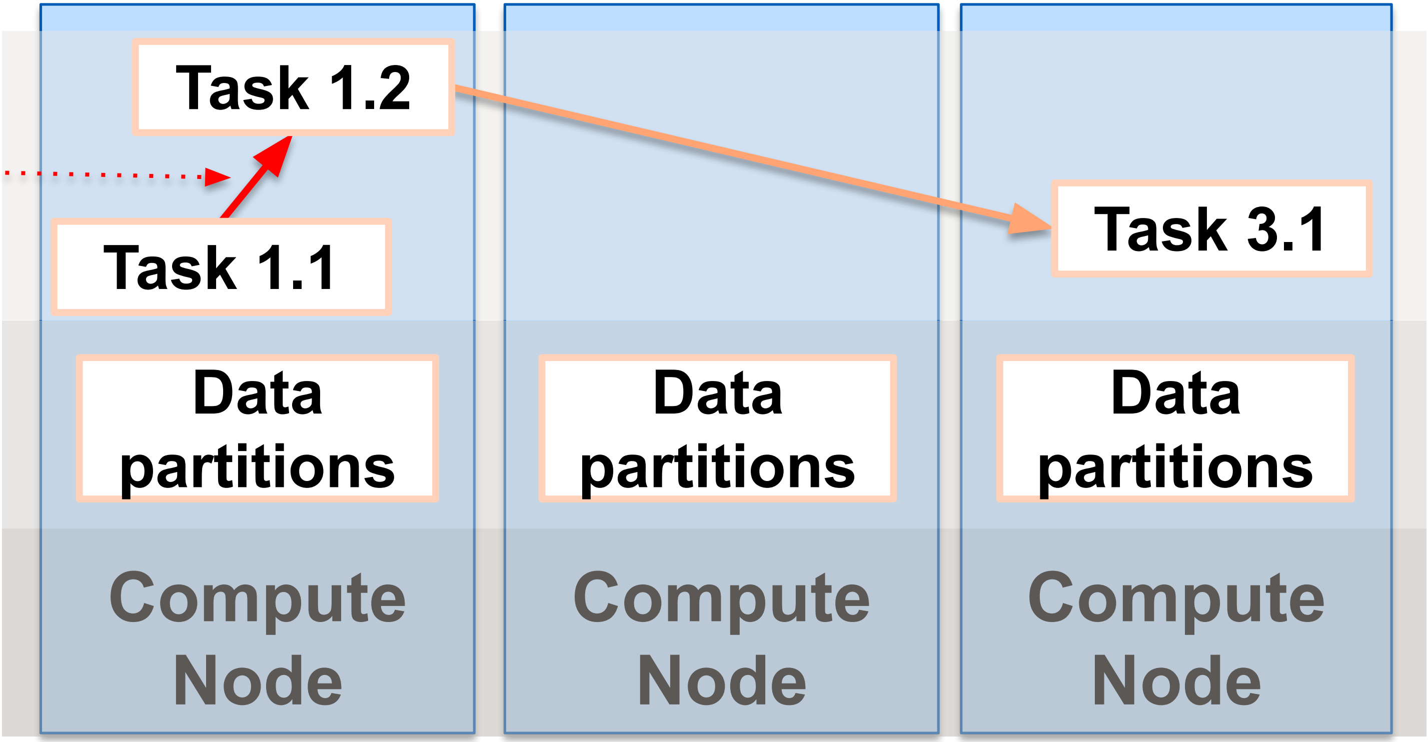
- via memory and network?
- via local disk and network?
- via distributed/shared file systems and network?

Which one we shall avoid?

Exchange data possibilities

Try to analyze and identify this problem with a framework of your choice (and fault management)

Need to exchange data



Programing frameworks in our focus

- Programming models
  - OS-based multi-threads/processes
  - embarrassingly parallel programming
  - MapReduce/Spark
  - workflows
  - (distributed) SQL processing with MPP (Massive Parallel Processing)
- Programming frameworks
  - Apache Hadoop/Spark, Dask, Polars, Apache Airflow
- Not in our focus:
  - HPC MPI (Message Passing Interface), NCCL (NVIDIA Collective Communications Library)

e.g., IBIS
https://ibis-project.org/

Unified APIs

Programming Frameworks

Programming Models

Data processing workload and task graph dependencies with Dask

https://www.dask.org/

Key features

- Data input/output connectors
  - file types: CSV, Parquet, HDF5, ORC, Json
  - source: Cloud storage (S3, Google), HDFS, Snowflake, BigQuery, Delta Lake
- Data Collections
  - Array (like numpy array), Bag/Multiset (suitable for unstructured data, like text), DataFrame
- Operations:
  - joins, concatenation, aggregation (`first, sum`, ...)
  - grouping/resampling, SQL-alike support
  - functions/computation suitable for arrays
- Execution modes
  - lazy by default; and support specific delayed and future tasks

Key features

- Data can be splitted and processed in parallel tasks
    - many operations on dataframes/tables can be parallelized, with little/without dependency among tasks
    - using directed acyclic graph (DAG) to represent tasks
    - little communication among them, little data shuffle between tasks
- Single and multiple compute nodes for processing
    - multiprocessing in single node vs distributed nodes
    - scheduling graphs using OS threads and processes to execute tasks
    - data exchange among tasks using shared memory, direct communication or network file systems
    - using different resource management systems: Kubernetes, SLURM, PBS, etc.
    - spill data into disks when running out of memory

Parallelizing dataframe → embarrassingly

- A big dataset can be presented as a Dask dataframe
  - a Dask dataframe can be partitioned into different partitions
- Perform operations on data partitions with lazy principles
  - explicitly call `compute()` method → computation

Dataframe in partitions

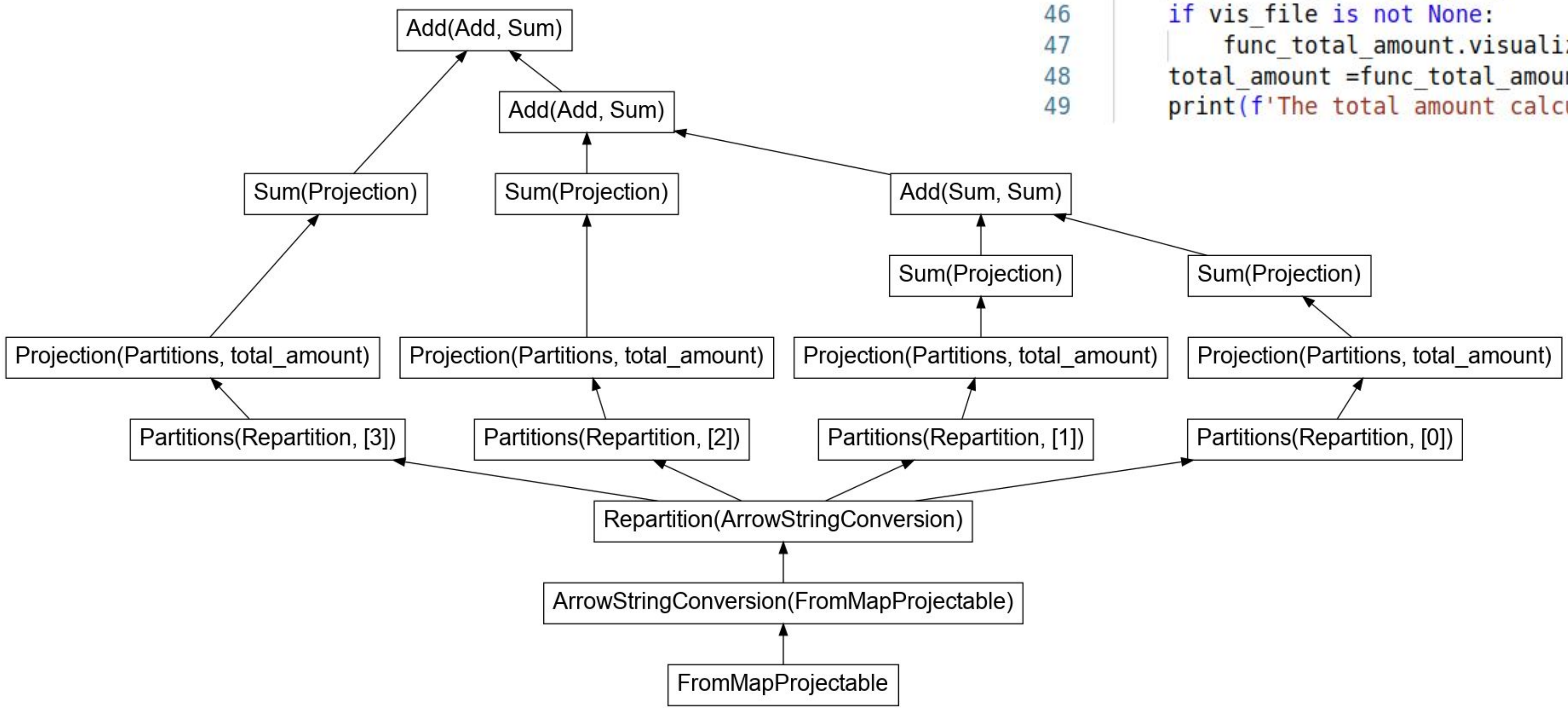
Dask Dataframe

	VendorID	total_amount
0	1.0	11.80
1	1.0	4.30
2	1.0	51.95
3	1.0	36.35
4	2.0	24.36
..	...	...
95	2.0	21.96
96	2.0	17.30
97	2.0	15.36
98	2.0	24.80
99	2.0	13.30

```
Data records: 1369765
Data has 4 partitions
Partition 0 has 342441
Partition 1 has 342441
Partition 2 has 342441
Partition 3 has 342442
```

Example

```python
from dask.distributed import Client
# make sure that dask scheduler and worker running
client = Client(f'{dask_scheduler_host}:{dask_scheduler_port}')
taxi_df = dd.read_csv(input_file, dtype = dtype,
                      assume_missing=True,
                      low_memory=False)
print(f'Total records: {len(taxi_df)}')
p_taxi_df = taxi_df.repartition(npartitions=num_partitions)
func_total_amount = p_taxi_df.get_partition(0)["total_amount"].sum()
for i in range(0,num_partitions):
    func_total_amount = func_total_amount \
        + p_taxi_df.get_partition(i)["total_amount"].sum()
if vis_file is not None:
    func_total_amount.visualize(filename=vis_file)
total_amount =func_total_amount.compute()
print(f'The total amount calculated from this file is {total_amount}')
```

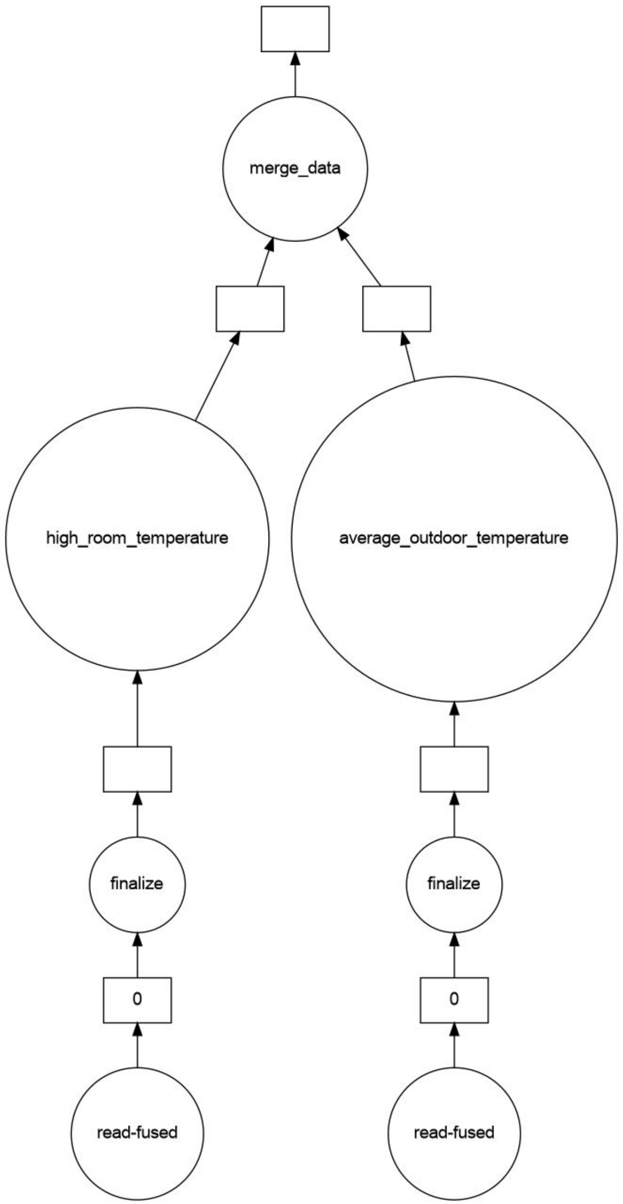


Task dependency based on DAG

- Flexible to define task graphs
  - as a directed acyclic graph
- Explicitly lazy, deferred execution
  - using `dask.delayed()/@dask.delayed` to declare delayed tasks
- Concurrent, asynchronous eager execution
  - using future tasks
- Suitable for problems cannot be solved with Dask Dataframe

Task dependency based on DAG

```python
if delayed_mode:
    # delayed tasks
    task11 = dask.delayed(high_room_temperature)(bts_alarm_df)
    task12 = dask.delayed(average_outdoor_temperature)(bts_parameter_df)
    final_task = dask.delayed(merge_data)(task11,task12)
    if vis_file is not None:
        final_task.visualize(filename=vis_file)
    final_result = final_task.compute()
    print(f'First 100 elements\n: {final_result.head(100)}')
```



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

Apache Spark

Can be run a top

- Clusters of shared-nothing compute nodes with Hadoop (using HDFS and YARN)
- Kubernetes
- A set of standalone machines in a master-worker architecture

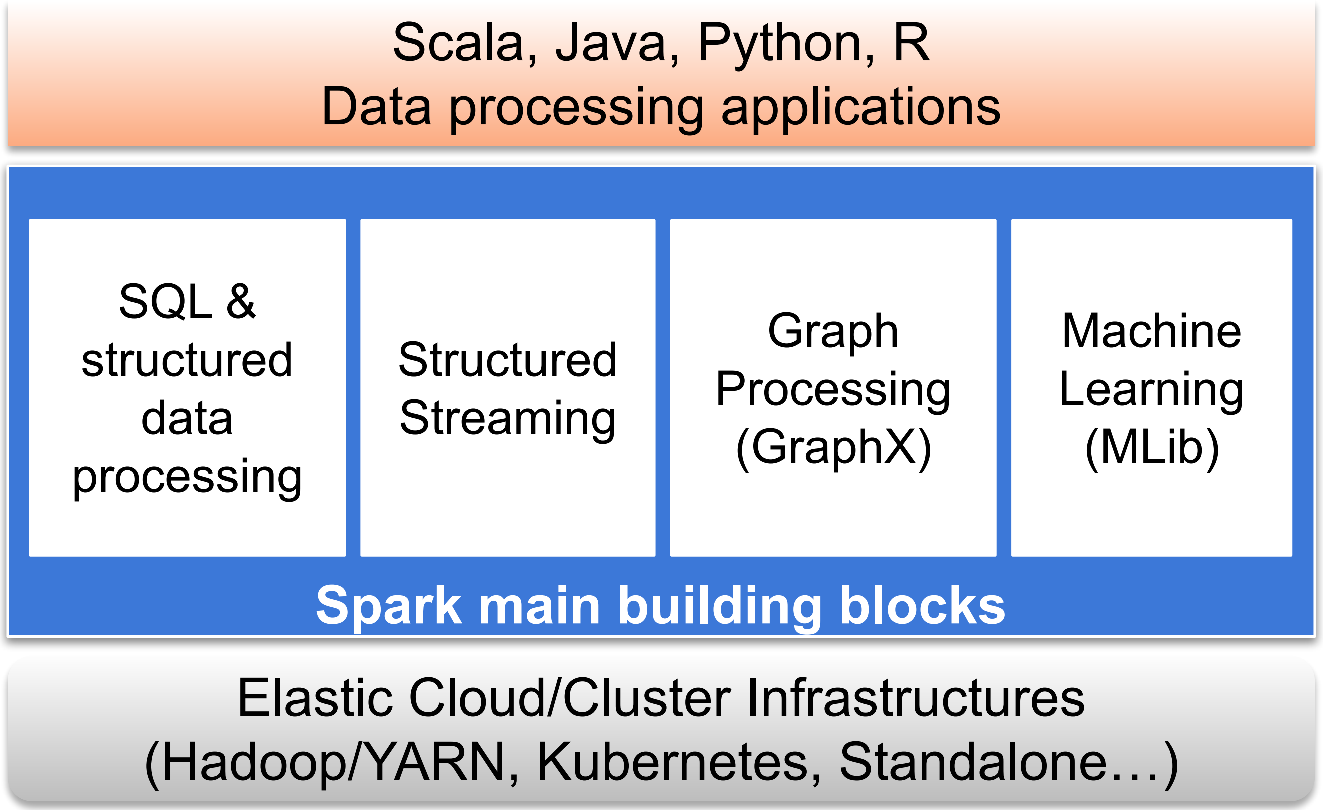
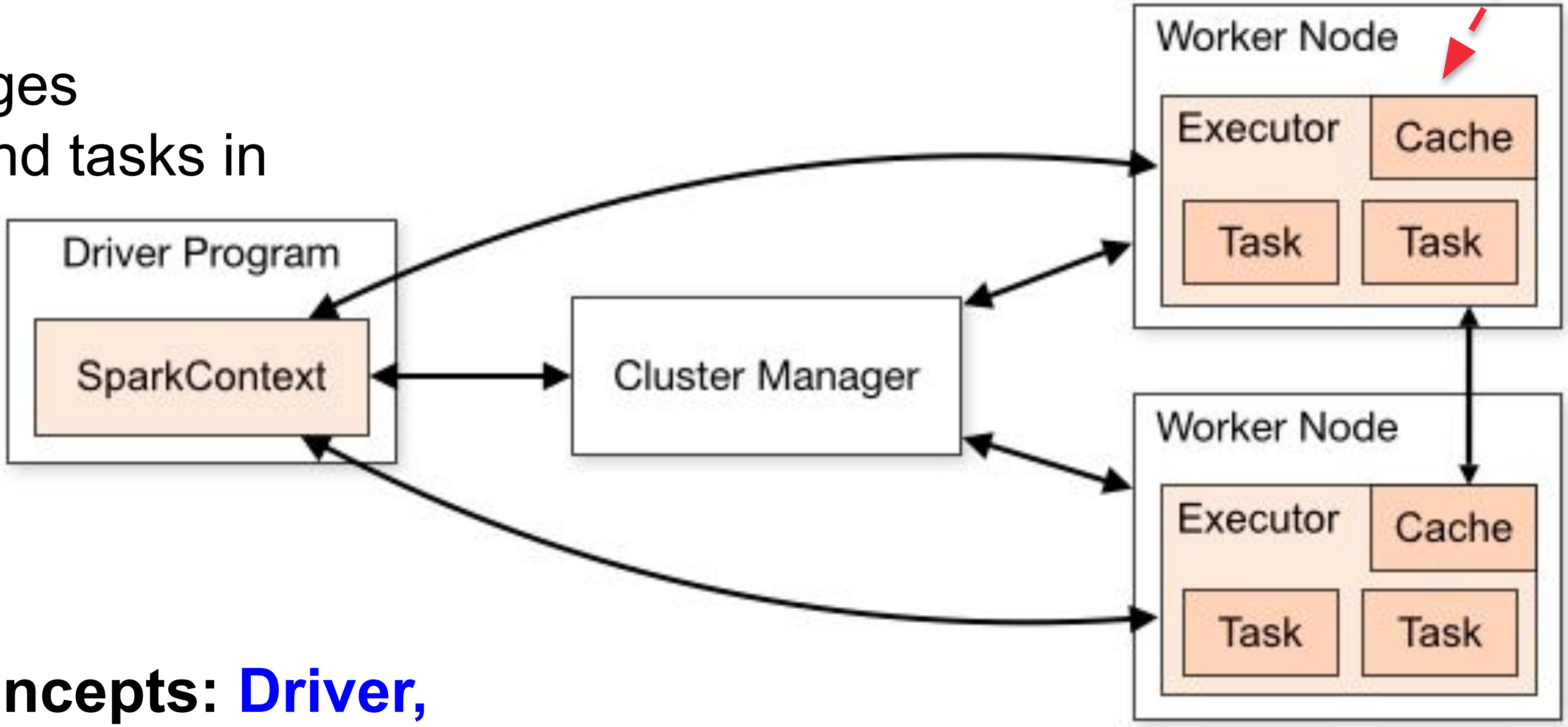


Figure source: *http://spark.apache.org/*

Execution model in a cluster system

Computing resources in a cluster node

Driver manages operations and tasks in nodes



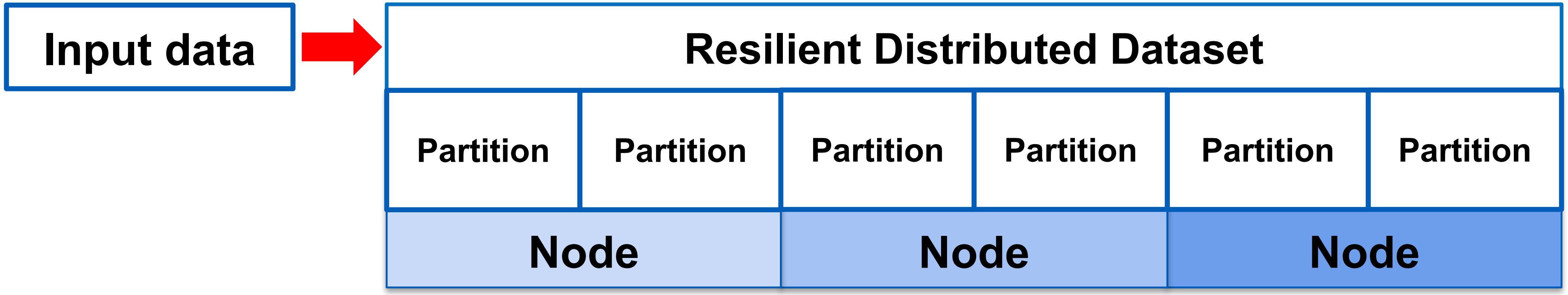
Common concepts: Driver, Node, Task and Executor

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

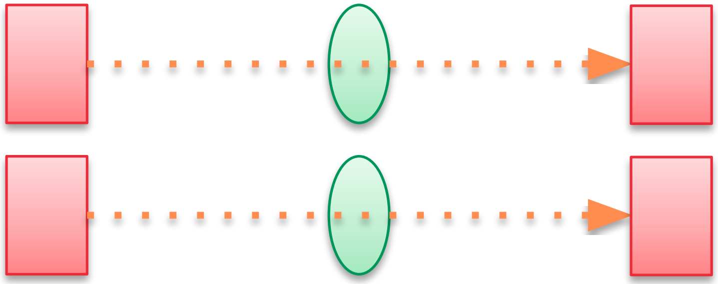
Key features



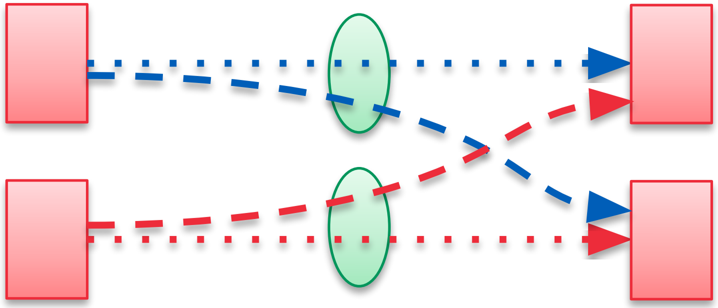
- Input data is distributed in different nodes for processing
  - support partitions for data processing: a node keeps one or n partitions, a partition resides only in a node ⇒ for computing
- Key operations: transformations and actions on data
- Leverage parallel computing concepts to run multiple tasks
  - data 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 (directed acyclic graph) → a lineage of what to do
- Lazy approach ⇒ real transformations will be done when action operations are triggered



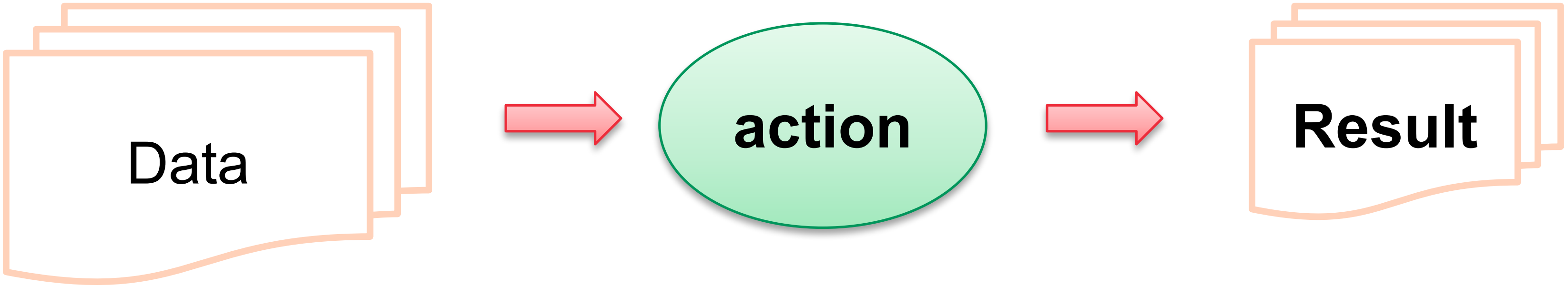
Narrow transformation, no data shuffle



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
  - acts as a program driver to manage the execution of tasks
  - SparkContext: manages connection to a cluster and internal services
- Data APIs
  - low-level Resilient Distributed Dataset (RDD) & shared variables
  - high-level DataFrames/DataSets
  - load and hold distributed data
  - transformation and action functions
- ML, graph and streaming functions and pipelines

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

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

```
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]	10[redacted]23	26[redacted]	1	2	7	39	100560530[redacted]	[redacted]08/2019 00:04:07	148163	49018
[redacted]	10[redacted]23	26[redacted]	1	2	7	38	100560530[redacted]	[redacted]08/2019 00:04:07	1658	1362
[redacted]	10[redacted]23	26[redacted]	1	2	7	9	100560530[redacted]	[redacted]08/2019 00:04:07	6693	5185
[redacted]	10[redacted]23	26[redacted]	1	2	7	8	100560530[redacted]	[redacted]08/2019 00:04:07	640	544
[redacted]	10[redacted]23	26[redacted]	1	2	7	11	100560530[redacted]	[redacted]08/2019 00:04:07	118	114
[redacted]	10[redacted]23	26[redacted]	1	2	7	10	100560530[redacted]	[redacted]08/2019 00:04:07	28514	12495
[redacted]	10[redacted]23	26[redacted]	1	2	7	13	100560530[redacted]	[redacted]08/2019 00:04:07	868699	23400
[redacted]	10[redacted]23	26[redacted]	1	2	7	15	100560530[redacted]	[redacted]08/2019 00:04:07	1822	1120
[redacted]	10[redacted]23	26[redacted]	1	2	7	17	100560530[redacted]	[redacted]08/2019 00:04:07	998069	117345
[redacted]	10[redacted]23	26[redacted]	1	2	7	16	100560530[redacted]	[redacted]08/2019 00:04:07	22402	1804
[redacted]	10[redacted]23	26[redacted]	1	2	7	19	100560530[redacted]	[redacted]08/2019 00:04:07	640	791
[redacted]	10[redacted]23	26[redacted]	1	1	10	49	100560530[redacted]	[redacted]08/2019 00:04:07	662	494
[redacted]	10[redacted]23	26[redacted]	1	1	10	48	100560530[redacted]	[redacted]08/2019 00:04:07	2158	759
[redacted]	10[redacted]23	26[redacted]	1	2	7	21	100560530[redacted]	[redacted]08/2019 00:04:07	0	0
[redacted]	10[redacted]23	26[redacted]	1	1	10	51	100560530[redacted]	[redacted]08/2019 00:04:07	2600890	54153
[redacted]	10[redacted]23	26[redacted]	1	2	7	20	100560530[redacted]	[redacted]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, Azure Blob Storage), Delta Lake, local files, etc.

Formats and Sources supported by DataFrames

Figure source: *https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html*

DataFrame Transformations & Actions

Transformations

- Several operations, think transformation for a relational table or a matrix

Example

- 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)
```

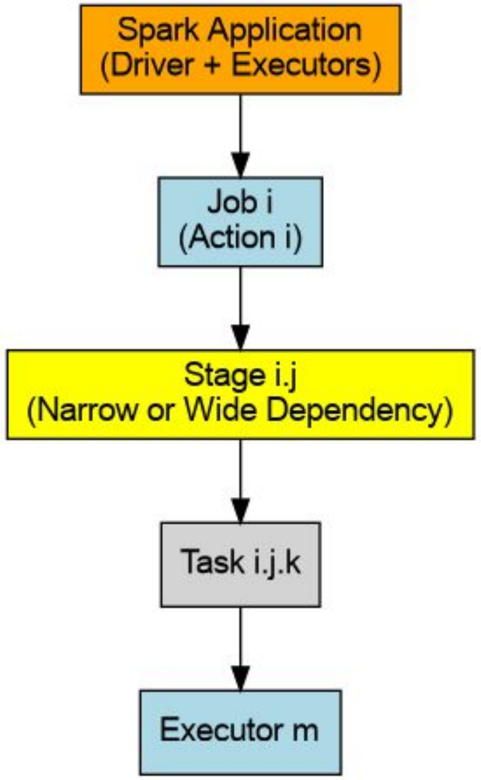
Session/Driver

Read data

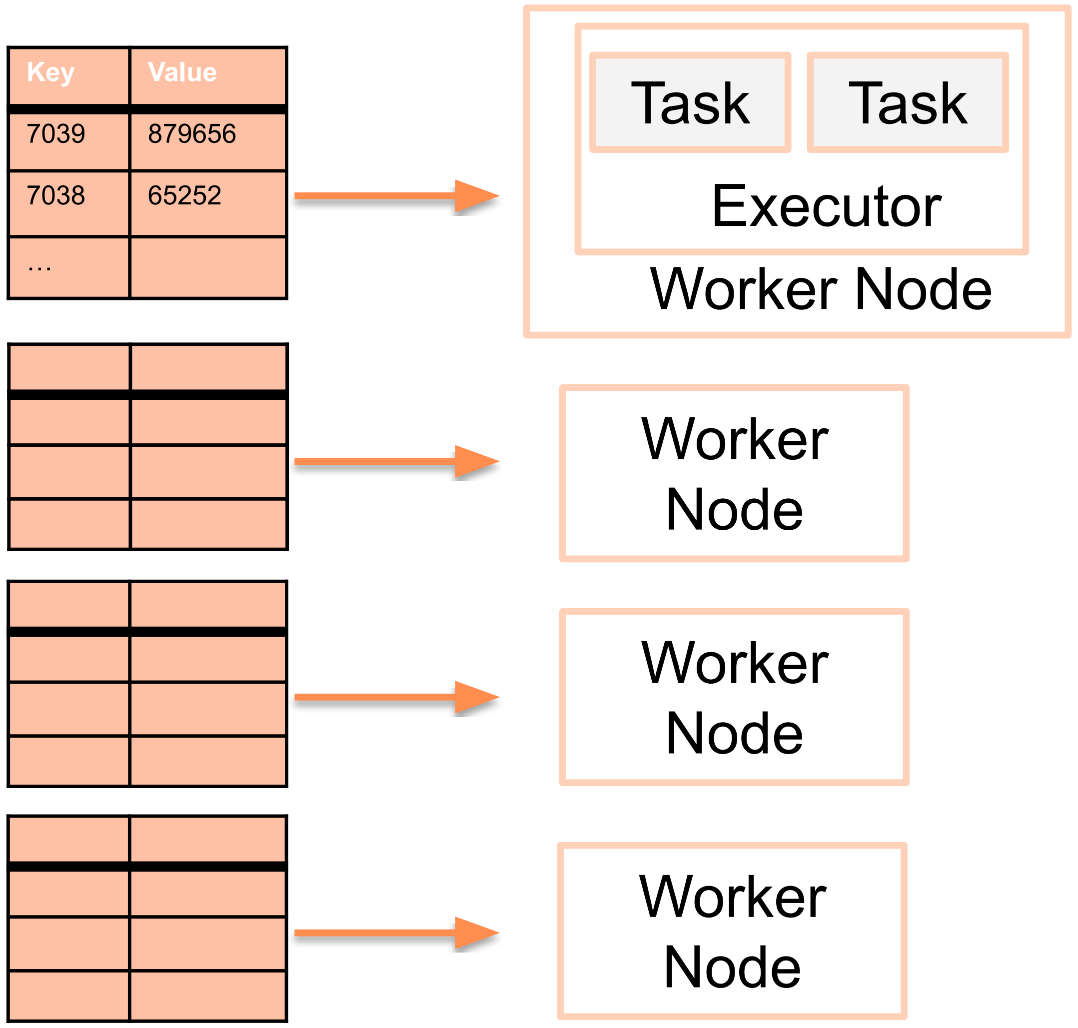
Apply operations

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 which does not lead to a data shuffle
  - 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



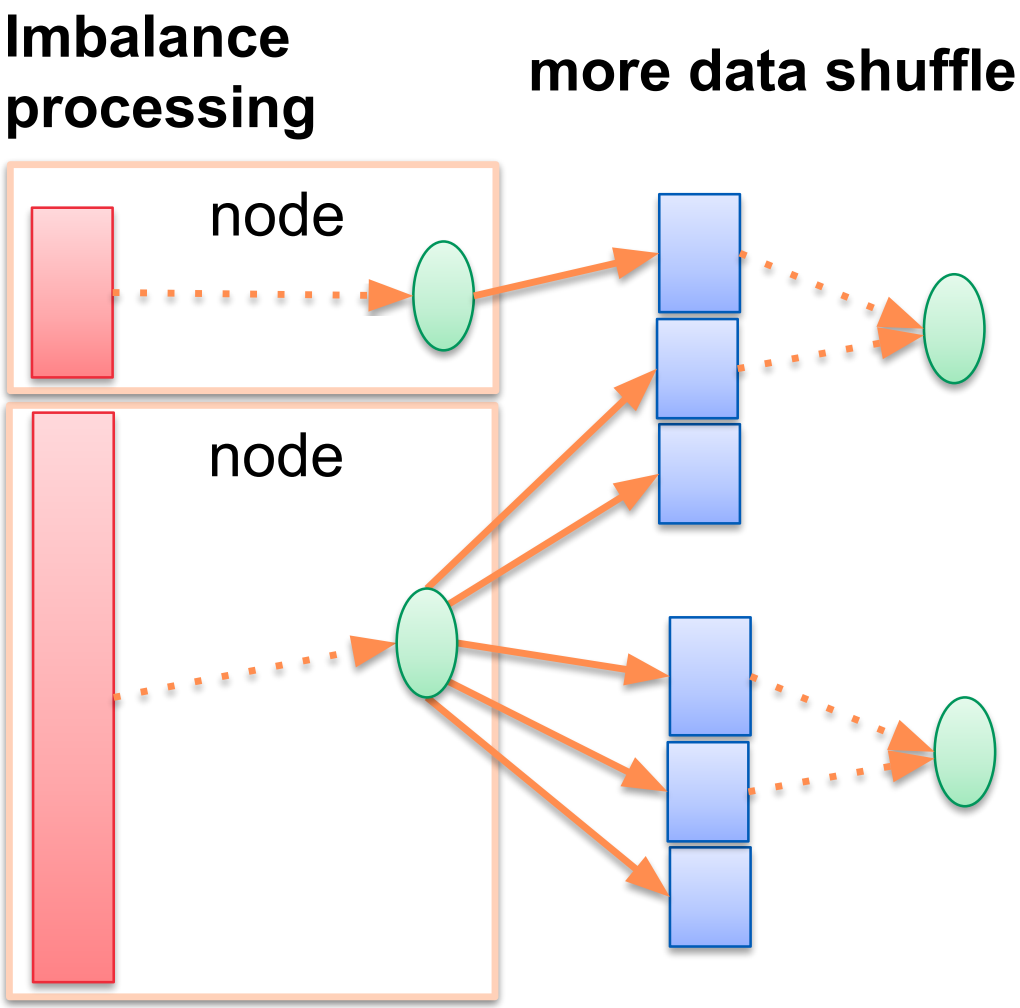
Data Distribution



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

Massive parallel processing for distributed query engines

Massive parallel processing employed by distributed query engine

- Key concepts
  - using SQL as a way to query different types of data sources like data lake, warehouse, and databases
    - managing catalogs and schemas about data sources
  - the query engine is decoupled from data sources/storage
  - using massive parallel processing (MPP) to support parallel tasks accessing different data sources at a large-scale with many compute nodes
- Complex fault tolerance and optimization:
  - failure management, query and data movement costs, …
- Mostly for analytics: interactive analytics, seconds – minutes

Example distributed SQL engine: Presto/Trino

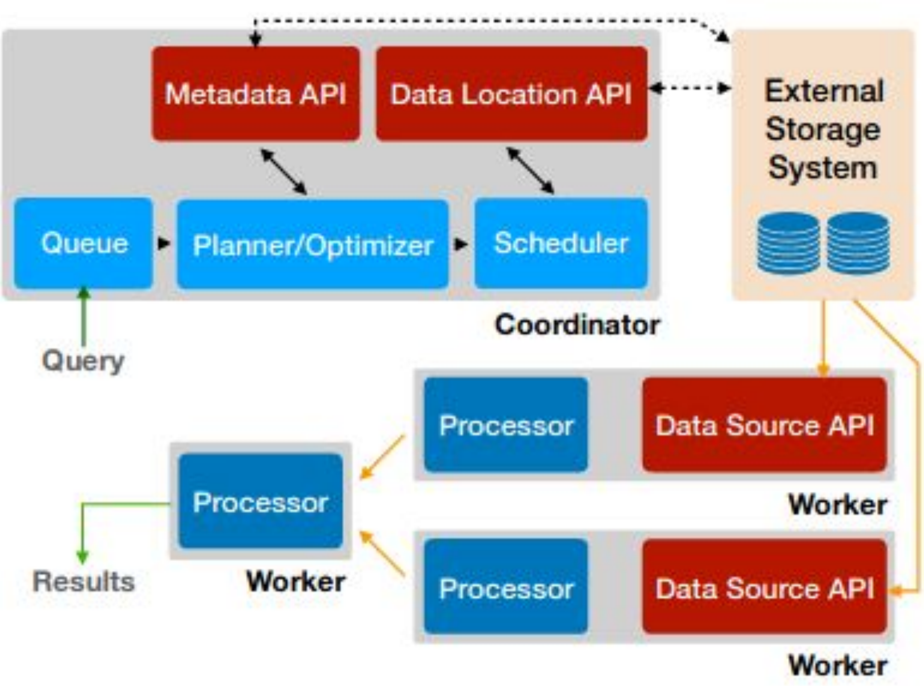
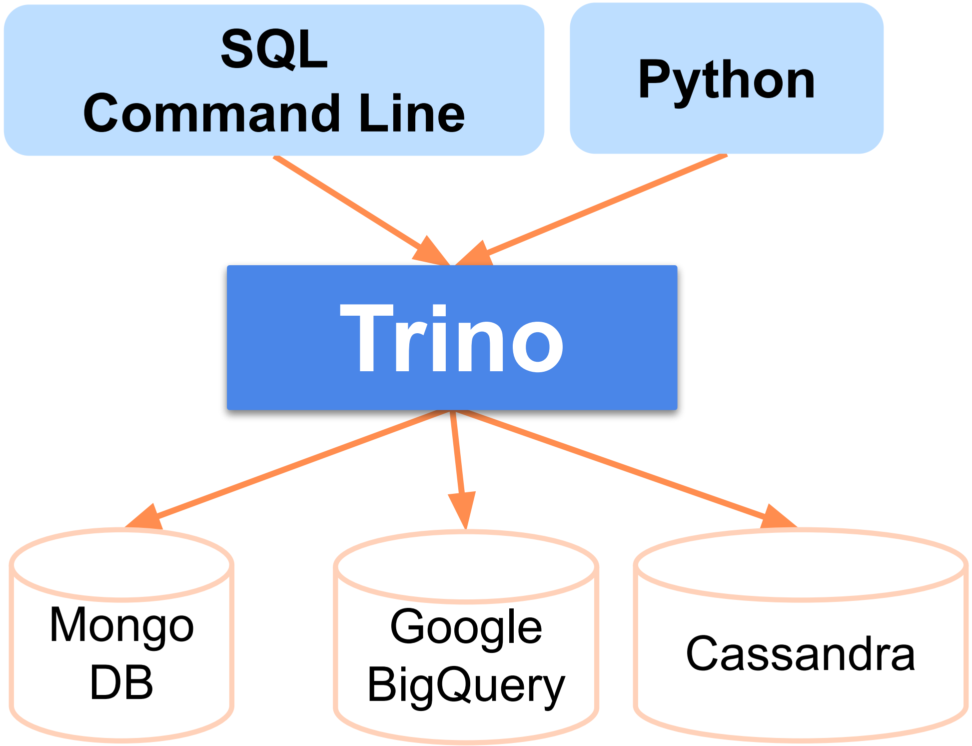


Figure source: *Presto: SQL on Everything https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8731547 &tag=1*

Small exercise
(see course git)



Trino (*https://trino.io/*): from a fork of Presto

Summary

- Different programming models for data processing
  - models and tools selected based on data workload and ecosystems, including underlying compute resource management
  - both developers and platform operator/provider must carefully decide the programming models for data processing
- Effects of modernization and composability in data platforms
  - Spark is powerful but many emerging ones, e.g., Polars and DuckDB, which may be suitable due to learning curves, management, and data load
- Rich ecosystems
  - combine data, different distributed programming supports for big data platforms

Thanks!

Hong-Linh Truong
Department of Computer Science

rdsea.github.io

Kiitos

aalto.fi