THE UNIVERSITY OF QUEENSLAND AUSTRALIA | CREATE CHANGE

Cloud Computing (INFS3208)

Lecture 8: Introduction to Spark Framework -- An in-Memory Analytics Tool for Data Science

Lecturer: AsPr Sen Wang

School of Electrical Engineering and Computer Science

Faculty of Engineering, Architecture and Information Technology

The University of Queensland

PhD Recruitment （1/4）

Research focuses on state-of-the-art topics in AI and Machine Learning. Successful candidates will have the opportunity to contribute to one or more of the following areas:

- **Multimodal Learning:** Developing models that can understand and reason about the world from multiple data sources, such as text, images, and audio.
- **AI for Edge Devices:** Designing efficient and powerful AI algorithms (e.g., model compression, quantization, and neural architecture search) that can run on resource-constrained devices like smartphones and IoT sensors.
- **Reinforcement Learning (RL):** Creating intelligent agents that learn to make optimal decisions through interaction and feedback, with applications in robotics, game theory, and autonomous systems.
- **Large Language Models (LLMs) and their Applications:** Exploring the frontiers of LLMs, including topics like instruction-tuning, retrieval-augmented generation (RAG), efficient fine-tuning, and developing novel agent-based systems.

PhD Recruitment （2/4）

Essential:

- Domestic applicants and those onshore international students who have completed a program at UQ in Semester 2 2025.
- Applicants must be onshore at the time that offers are issued.
- An outstanding academic record, typically a First Class Honours degree or a Master's degree with a significant research component in Computer/Data Science, Engineering, Mathematics, or a related discipline.
- A strong mathematical foundation and excellent programming skills (Python is essential; experience with frameworks like PyTorch or TensorFlow is highly valued).
- Excellent written and verbal communication skills in English.
- Strong motivation to conduct high-impact research and publish at top-tier conferences.

Desirable:

- Previous research experience (e.g., an honours thesis, research-based project, or publications).
- Demonstrated experience in one of the research areas listed above.

PhD Recruitment （3/4）

This position is contingent upon the applicant successfully securing a research scholarship through the UQ Graduate School. The UQ Graduate School Scholarship (UQGSS) provides a competitive living stipend and a full tuition fee waiver.

Please take note of the key dates for the upcoming UQ scholarship round:

- **Applications Open:** Monday, 25 August 2025
- **Applications Close:** Monday, 22 September 2025 (All documents must be submitted)
- **Scholarship Outcomes:** From Monday, 15 December 2025
- **Acceptance Deadline:** Wednesday, 7 January 2026

It is crucial to prepare your application well in advance of the deadline.

PhD Recruitment （4/4）

Interested candidates should first submit an **Expression of Interest (EOI)** via email to me at **sen.wang@uq.edu.au** with the subject line "PhD Application EOI".

Please include the following in your EOI:

- A **Cover Letter** introducing yourself, outlining your research interests, and explaining why you are a suitable candidate.
- A detailed **Curriculum Vitae (CV)**.
- Your **Academic Transcripts**.

Shortlisted candidates will be contacted for an interview and will be supported in preparing their formal application to the UQ Graduate School.

Associate Professor Sen Wang | UQ Experts

Recap

- Introduction to Vector Databases
- Fundamentals of Vector Databases
  - What are vectors?
  - Vector Similarity Measures
- Indexing Techniques for Vector Databases
  - Introduction to Indexing
  - Types of Indexing Techniques
- Vector Databases in the Cloud
- Applications:
  - Retrieval Augmented Generation
  - Anomaly Detection

THE UNIVERSITY OF QUEENSLAND AUSTRALIA

Outline

➡ • Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- How Spark works

Background – Digital Revolution

- Digital Revolution (aka the Third Industrial Revolution), is the shift from mechanical and analogue electronic technology to digital electronics.
- Milestones:
  - Digital computers (the late 1950s to the late 1970s)
  - Personal computers (1970s to 1980s)
  - WWW (1990s)
  - Cellular phones (2000s)
  - IoT and Cloud Computing (2010s)
  - Generative AI (2020s) – chatGPT, falcon, Llama 2, etc.

Information Processing

↓

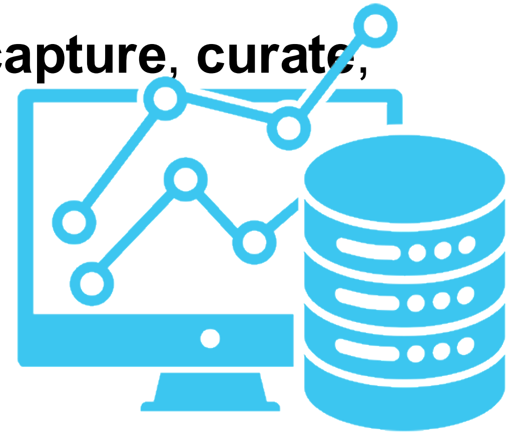
Information Transferring

↓

Information Explosion

Background – Big Data Era

- “Big Data” has been in use since 1990s.
- Data sets with sizes beyond the ability of commonly used software tools to **capture**, **curate**, **manage**, and **process** data within a tolerable elapsed time.
- Reasons of Big Data:
  - Hardware development: Storage (more cheaper), CPUs (more cores)
  - Internet bandwidth: 56kbps vs 1000Mbps
  - Data generation:
    - Transactional data (stock, shopping records in Woolworths/Coles)
    - User-centric data (videos/images)
    - Sensor-based data (cameras)



1 Bit	=	0 or 1
8 Bits	=	1 Byte
1024 Bytes	=	1 KB
1024 KBs	=	1 MB
1024 MBs	=	1 GB
1024 GBs	=	1 TB
1024 TBs	=	1 PB
1024 PBs	=	1 EB
1024 EBs	=	1 ZB
1024 ZBs	=	1 YB

https://en.wikipedia.org/wiki/Big_data
https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

Background – Big Data Era

- Characteristics of Big Data: 5Vs:
  - **V**olume (quantity) – from 4.4 trillion gigabytes to 44 trillion (more than doubles every two years).
  - **V**ariety (type) – structured vs non-structured
  - **V**elocity (speed)
  - **Veracity** (quality)
  - **V**alue
- Data must be **processed** with advanced tools to reveal **meaningful information**.
  - Data mining and machine learning
  - Cloud computing

1 Bit	=	0 or 1
8 Bits	=	1 Byte
1024 Bytes	=	1 KB
1024 KBs	=	1 MB
1024 MBs	=	1 GB
1024 GBs	=	1 TB
1024 TBs	=	1 PB
1024 PBs	=	1 EB
1024 EBs	=	1 ZB
1024 ZBs	=	1 YB

https://en.wikipedia.org/wiki/Big_data
https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

Background – Big Data Technologies

To solve big data problems, two types of technologies are required:

- Distributed storage
  - Huge volumes of data are stored on clusters of storage nodes
  - Distributed file systems & Databases (Cluster Relational DBs & NoSQL DBs)
  - GFS, BigTable (Google) and HDFS/HBase (open-source in Apache Hadoop)
- Distributed computing
  - Clusters of computing nodes process data in a parallel manner
  - Distributed computing models/frameworks
  - MapReduce in Hadoop and Apache Spark
- MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computers (nodes)
  - Map/Reduce
  - More I/O expensive than Spark (Hard disks vs Memories)

https://projects.apache.org
https://en.wikipedia.org/wiki/The_Apache_Software_Foundation

Outline

- Background
- ➡ Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- How Spark works

What Is Apache Spark?

- Apache Spark is an open-source distributed general-purpose cluster-computing framework.
- Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.
- Spark was initially started by Matei Zaharia at UC Berkeley's AMPLab in 2009, and open sourced in 2010 under a BSD license.
- Spark codebase was later donated to the Apache Software Foundation, which is now labelled as one of most active Apache projects.
- Spark aims to be fast: in-memory computing
- Spark vs MapReduce
  - Much faster (in-memory, low latency)
  - Wider range of workloads (e.g. iterative algorithms)

Dr Matei Zaharia

https://en.wikipedia.org/wiki/Apache_Spark

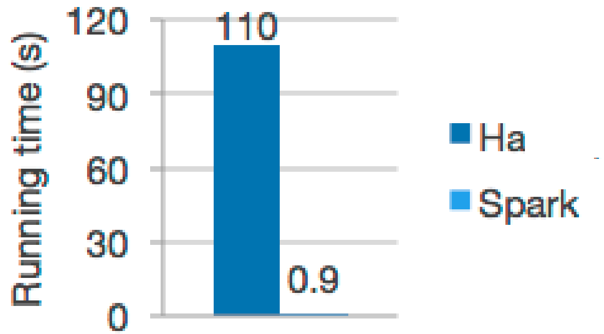
Characteristics of Spark – Speed

Apache Spark achieves high performance in terms of processing speed

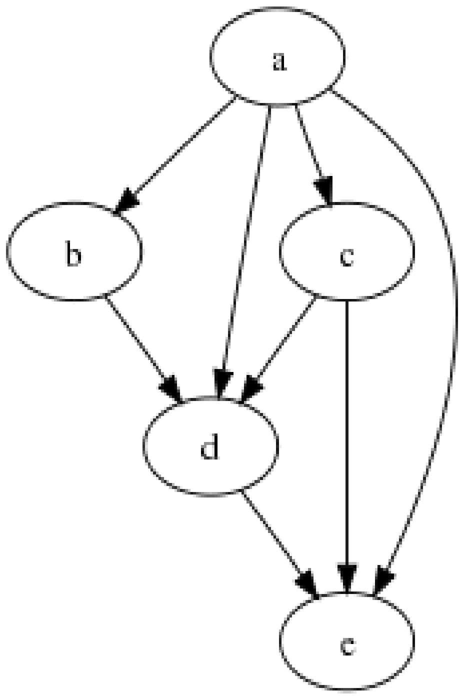
- Process data mainly in memory of working nodes
- Prevent unnecessary I/O operations on disks
- Use a state-of-the-art Directed Acyclic Graph (DAG) scheduler

Performance:

- Sorting Benchmark in 2014
  - 100TB data
  - 206 nodes in 23 mins (Spark)
  - 2,000 nodes in 72 mins (MapReduce)
  - 3x speed performance and 1/10 resources
- Machine Learning algorithm
  - 100x faster than Hadoop for logistic regression



Logistic regression in Hadoop and Spark



DAG

Characteristics of Spark – Ease of Use

Spark has provided many operators making it easy to build parallel apps.

- *map / reduce*
- *filter / groupByKey / join*
- and more.

Highly accessible with supported languages:

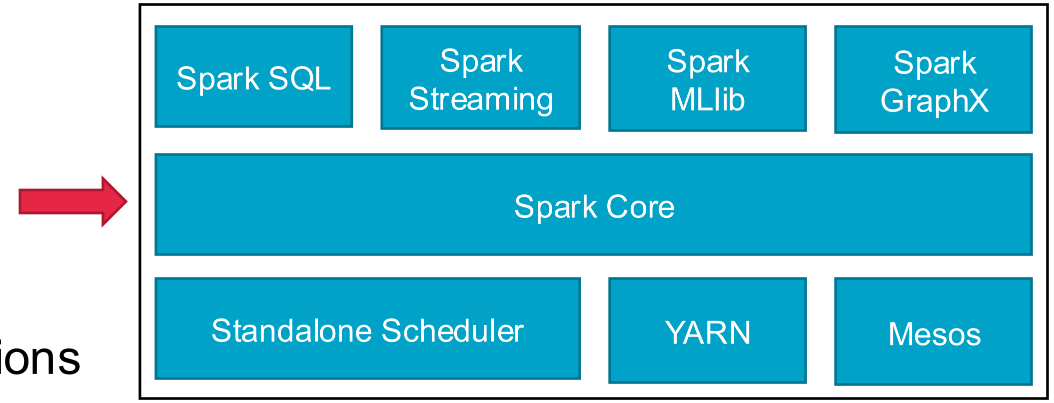
- Scala (interactive, fast)
- Python (interactive, slow)
- Java (non-interactive, fast)
- R and SQL shells (interactive, slow)

https://spark.apache.org

Characteristics of Spark – A Unified Stack

Spark contains multiple closely integrated components:

- Spark Core:
  - contains the basic functionality of Spark,
    - **task scheduling**,
    - **memory management**,
    - **fault recovery**,
    - **interacting** with storage systems,
    - and more.
  - Responsible for ***resilient distributed data*** (**RDDs**) definitions
    - RDDs are Spark's main programming abstraction.
    - RDDs represent a collection of items distributed across many compute nodes (in memory)
    - RDDs are central in Spark programming



Spark Stack

https://spark.apache.org

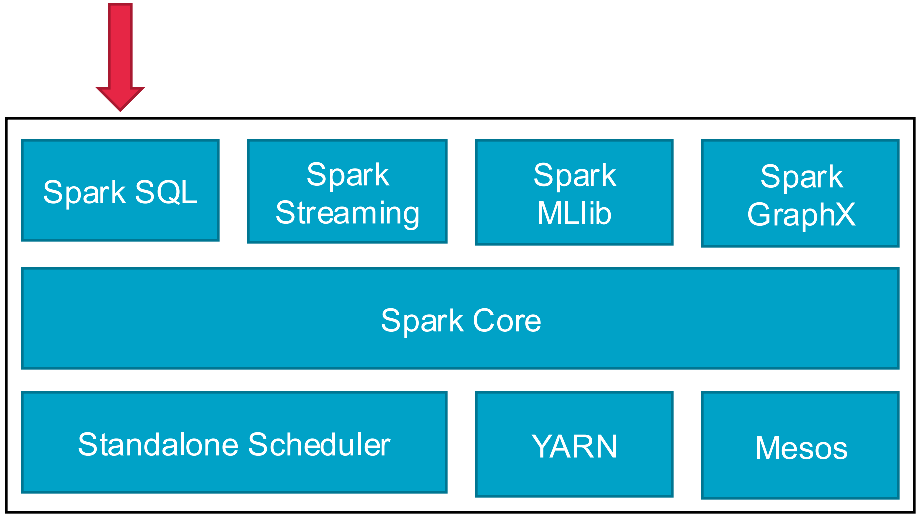
Characteristics of Spark – A Unified Stack

Spark contains multiple closely integrated components:

- Spark SQL is for working with structured data.
  - It allows querying data via SQL
  - it supports many sources of data, including Hive tables, JSON, etc.
  - Except for SQL interface to Spark, Spark SQL allows to intermix SQL queries in Python, Java, and Scala, all within a single application

```
spark.sql("SELECT * FROM people").show()
```

- When running SQL from within another programming language the results will be returned as a **Dataset/DataFrame**.



Spark Stack

https://spark.apache.org

Characteristics of Spark – A Unified Stack

Features of Spark SQL

- Integration With Spark:
  - Spark SQL queries are integrated with Spark programs, allowing to query structured data inside Spark programs, using SQL or a DataFrame API. Usable in Java, Scala, Python and R.
- Uniform Data Access:
  - DataFrames and SQL support a common way to access a variety of data sources, like Hive, JSON, and JDBC.
  - Joins the data across these different sources. This is very helpful to accommodate all the existing users into Spark SQL.
- Hive Compatibility:
  - Spark SQL runs unmodified Hive queries on current data.
  - It rewrites the Hive front-end and meta store, allowing full compatibility with current Hive data, queries, and UDFs.

https://www.edureka.co/blog/spark-sql-tutorial/
https://spark.apache.org

Characteristics of Spark – A Unified Stack

Features of Spark SQL

- Standard Connectivity:
  - JDBC or ODBC: JDBC and ODBC are the industry norms for connectivity for business intelligence tools.
- Performance And Scalability:
  - Due to the internal optimizers, Spark SQL can effectively and efficiently read from multiple sources (files, HDFS, JSON/Parquet files, existing RDDs, Hive, etc.) and conduct fast execution of existing Hive queries.
  - Much higher performance compared to Hadoop.

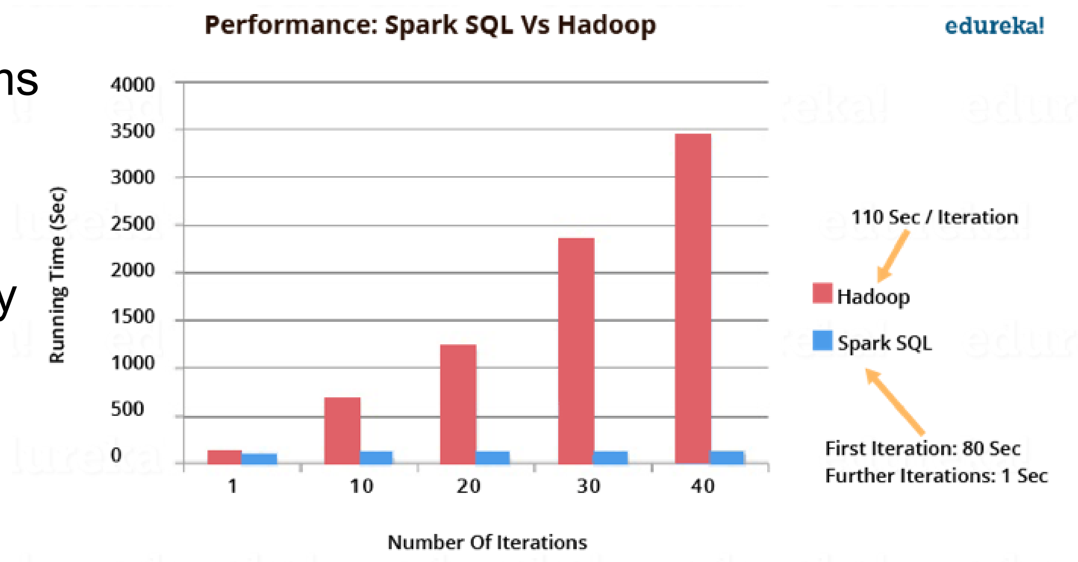


Figure: *Runtime of Spark SQL vs Hadoop. Spark SQL is faster*

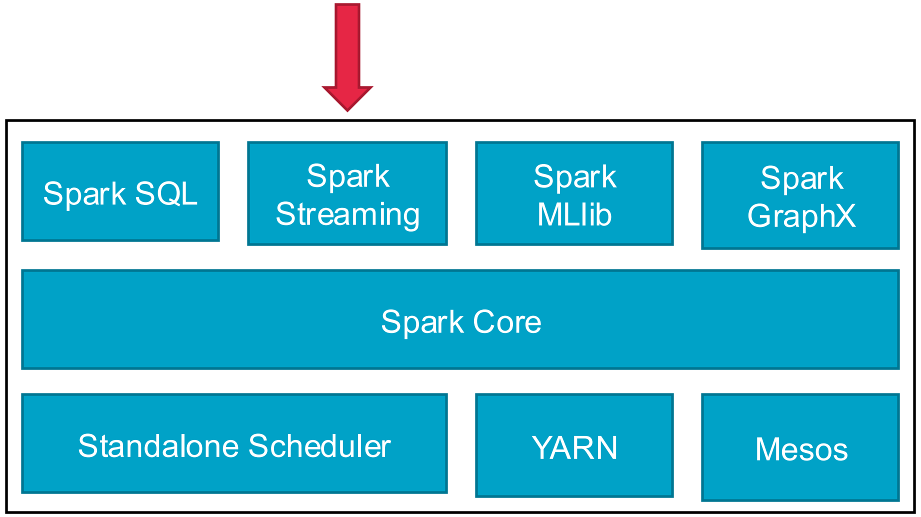
Source: *Cloudera Apache Spark Blog*

https://www.edureka.co/blog/spark-sql-tutorial/
https://spark.apache.org

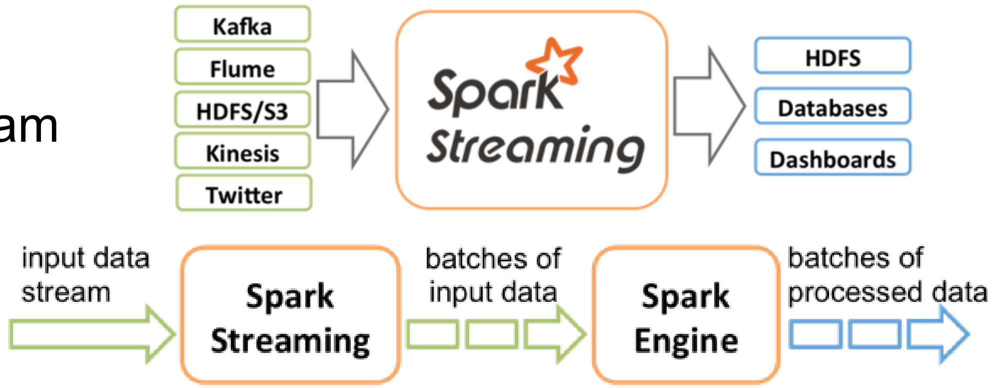
Characteristics of Spark – A Unified Stack

Spark contains multiple closely integrated components:

- Spark Streaming:
  - a component that enables scalable, high-throughput, fault-tolerant stream processing of live streams of data.
    - Logfiles generated by web servers
    - Web clicks & Advertising
    - Internet of Things: sensors
  - Data can be obtained from many sources like Kafka, Flume, or TCP sockets
  - Spark Streaming receives live input data streams and divides the data into smaller batches, which are then processed by the Spark engine to generate the final stream of results in batches.
  - Final results can be stored in file systems, databases, or even moved to live dashboards to be displayed



Spark Stack



https://spark.apache.org
https://spark.apache.org/docs/latest/streaming-programming-guide.html

Characteristics of Spark – A Unified Stack

Features of Spark Streaming

- **Dynamic load balancing**
  - Dividing the data into small micro-batches allows for fine-grained allocation of computations to resources.
- **Fast failure recovery**
  - In Spark, the computation is divided into small tasks that can run anywhere without affecting correctness.
  - Failed tasks are evenly distributed on all the other nodes in the cluster to perform the recomputations and recover from the failure faster than the traditional approach.

https://data-flair.training/blogs/apache-spark-streaming-tutorial/
https://spark.apache.org
https://spark.apache.org/docs/latest/streaming-programming-guide.html

Characteristics of Spark – A Unified Stack

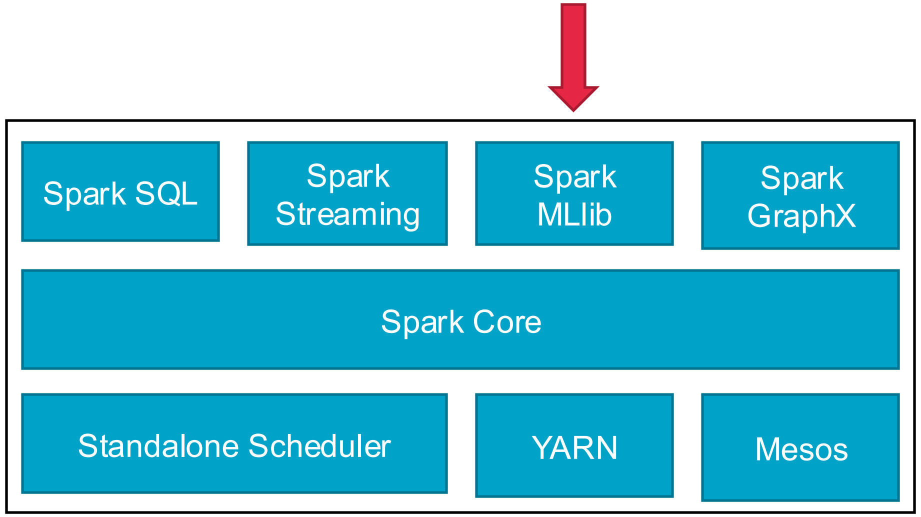
- **Interactive analytics on streaming data**
  - Arbitrary Apache Spark functions can be applied to each batch of streaming data. Since the batches of streaming data are stored in the Spark's worker memory, it can be interactively queried.
- **Advanced analytics like machine learning and interactive SQL**
  - Rich libraries like MLlib (machine learning) and SQL are available to micro-batches data.
  - Machine learning models generated offline with MLlib can apply to streaming data.
- **Performance**
  - Spark Streaming's ability to batch data can handle second-level tasks in many scenarios.
  - Spark Streaming can achieve latencies in subsecond (less than 1 sec).

https://data-flair.training/blogs/apache-spark-streaming-tutorial/
https://spark.apache.org
https://spark.apache.org/docs/latest/streaming-programming-guide.html

Characteristics of Spark – A Unified Stack

Spark contains multiple closely integrated components:

- Spark MLlib:
  - MLlib is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy.
  - ML Algorithms: common learning algorithms such as classification, regression, clustering, association rule mining, and collaborative filtering
  - Featurization: feature extraction, transformation, dimensionality reduction, and selection
  - some lower-level ML primitives, including a generic gradient descent optimization algorithm.
  - Utilities: e.g. model evaluation and data import tools.
- All of these methods are designed to scale out across a cluster.



Spark Stack

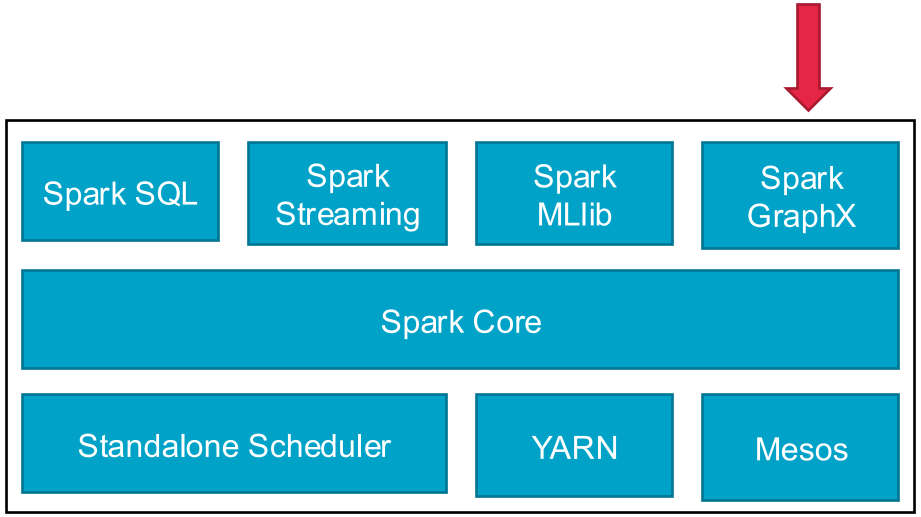
Classification: SVMs, Decision trees, Naïve Bayesian, etc.
Regression
Collaborative filtering
Clustering
Frequent pattern

https://spark.apache.org
https://spark.apache.org/docs/latest/streaming-programming-guide.html

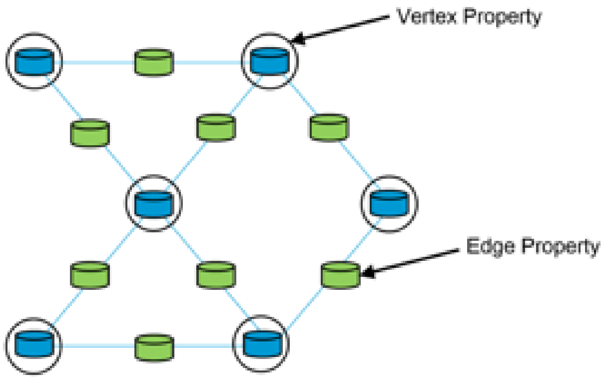
Characteristics of Spark – A Unified Stack

Spark contains multiple closely integrated components:

- Spark GraphX:
  - is a new component in Spark for graphs and graph-parallel computation.
  - At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction:
    - a directed multigraph with properties attached to each vertex and edge.
  - To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages).
  - In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.



Spark Stack



https://spark.apache.org
https://spark.apache.org/docs/latest/graphx-programming-guide.html

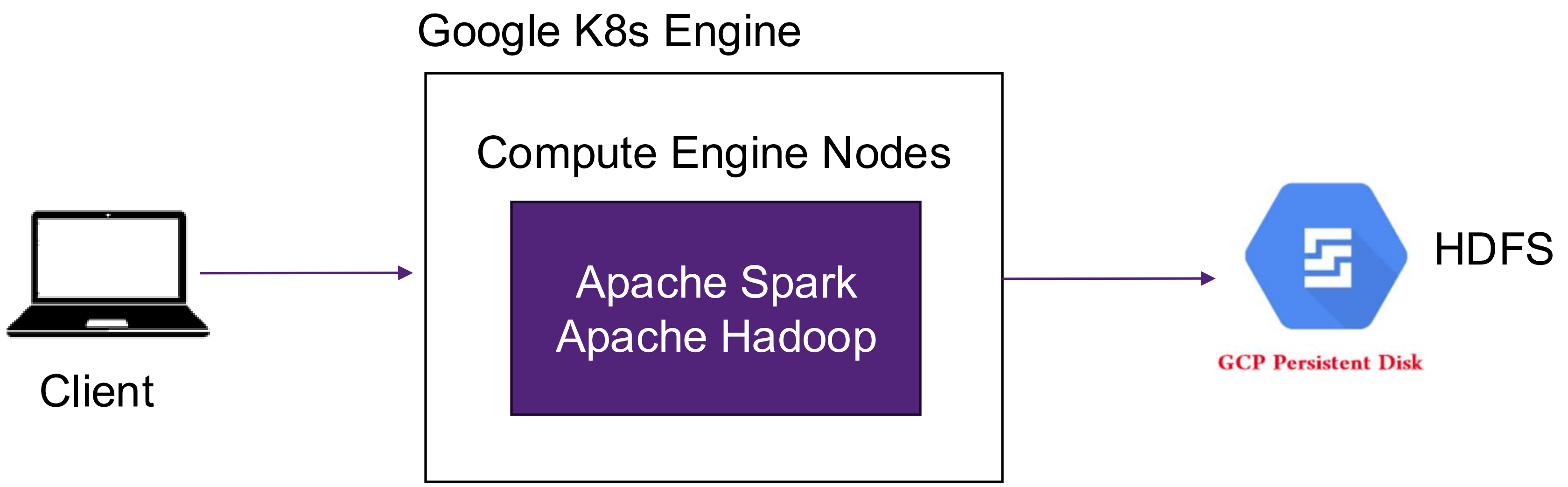
Characteristics of Spark – Runs Everywhere

- Under the hood, Spark is designed to efficiently scale up from one to many thousands of compute nodes.
- To achieve this while maximizing flexibility, Spark can run over a variety of *cluster managers*:
  - Hadoop YARN,
  - Apache Mesos,
  - Kubernetes,
  - and a simple cluster manager included in Spark itself (Standalone Scheduler).
- Can access diverse data sources:
  - HDFS, Cassandra, Hbase, Apache Hive, and hundreds of other data sources

Running Spark

Option 1: Create a Cluster using **Docker Swarm / K8s / GKE** and Run Spark on it.

You start with a general-purpose container orchestration platform (your GKE cluster). Then, you deploy and manage your Spark application on it, just like any other containerized microservice. You are responsible for configuring how Spark runs, managing its resources, and setting up its integrations.



You have **strong DevOps and K8s expertise**. Your team is comfortable with kubectl, Docker, and YAML. You require **fine-grained control** and **customization** over your Spark environment and dependencies.

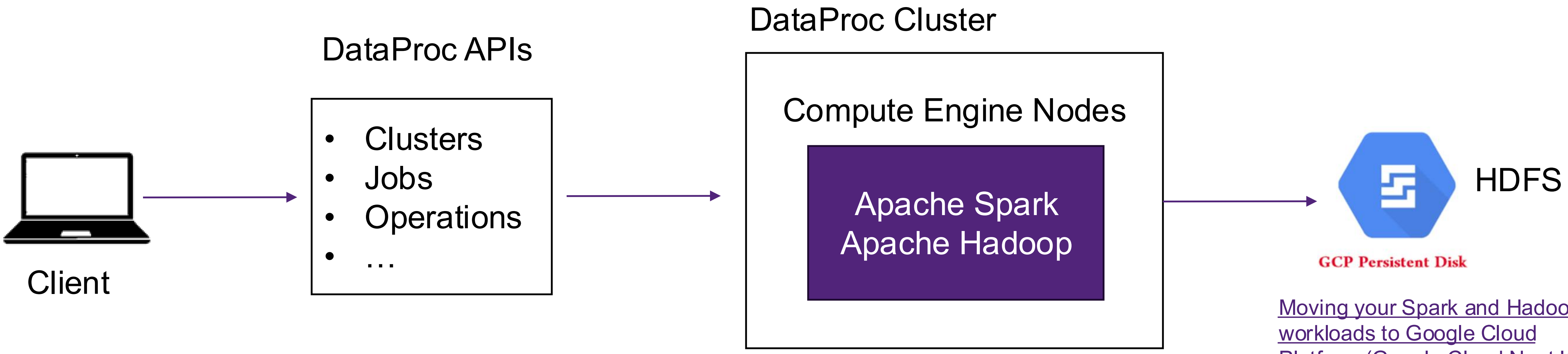
https://spark.apache.org

THE UNIVERSITY OF QUEENSLAND AUSTRALIA

Running Spark

Option 2: Create a dedicated service (DataProc/EMR/HDInsight) for administering Spark/Hadoop.

You press a button, and Google gives you a perfectly configured, optimized, and ready-to-go cluster for your Spark jobs. It handles the installation, integration with other GCP services (like BigQuery, Cloud Storage), monitoring, and scaling.



[Moving your Spark and Hadoop workloads to Google Cloud Platform (Google Cloud Next '17)]

Your team (e.g., data scientists, analysts) **wants to focus on** writing Spark code, not managing infrastructure. You need **deep, out-of-the-box integration** with the GCP data ecosystem (BigQuery, Cloud Storage, etc.).

https://spark.apache.org

Demo - Spark on DataProc

- Create a DataProc Cluster enabling a specific version of Apache Spark
- Use NYC Taxi and Limousine Commission (TLC) Trips Dataset (big dataset) available in BigQuery
  - tlc_yellow_trips_2022
  - **Table Size:** Approximately **8.85 GB**
  - **Number of Rows:** Approximately **39.7 million rows**
- Task 1: Big Data Query and Aggregation – Spark's SQL
  - Find the **average trip distance** and **total fare amount** by passenger **count #** for trips with a positive fare amount.
  - Aggregate big data using GROUP BY
- Task 2: Machine Learning for Fare Prediction – Spark's Mllib
  - Extract feature, train a regression model, and test the model's performance
  - Machine Learning model is applied
- **Both tasks CANNOT be processed on a single laptop due to Big Data!!**

https://spark.apache.org

Demo – Create a Dataproc Cluster (1/3)

- Open console and check project ID:

```
uqteaching@cloudshell:~$ gcloud projects list
PROJECT_ID: infs3208-437208
NAME: INFS3208
PROJECT_NUMBER: 644867479445
```

- Set a valid project:

```
uqteaching@cloudshell:~$ gcloud config set project infs3208-437208
Updated property [core/project].
```

- Enable the Dataproc, BigQuery, and Cloud Storage APIs:

```
uqteaching@cloudshell:~ (infs3208-437208)$ gcloud services enable dataproc.googleapis.com \
    compute.googleapis.com \
    storage.googleapis.com \
    bigquery.googleapis.com
Operation "operations/acat.p2-644867479445-f197f253-b8f5-4d4f-90bd-410d1612b971" finished successfully.
uqteaching@cloudshell:~ (infs3208-437208)$
```

Demo – Create a Dataproc Cluster (2/3)

- Create a GCS Bucket (Cloud Storage): a Cloud Storage bucket is used for staging files and storing job output. Note that the name of the bucket **must be globally unique**.

gsutil mb gs://your-unique-bucket-name/

```
uqteaching@cloudshell:~ (infs3208-437208)$ gsutil mb gs://uq-infs3208-dataproc-demo-bucket/
Creating gs://uq-infs3208-dataproc-demo-bucket/...
uqteaching@cloudshell:~ (infs3208-437208)$
```

- Create a Cluster:

gcloud dataproc clusters create taxi-demo-cluster \

--region=australia-southeast1 *--num-workers*=3 \

--worker-machine-type=n2-standard-4 \

--master-machine-type=n2-standard-4 \

--image-version=2.1-debian11 \

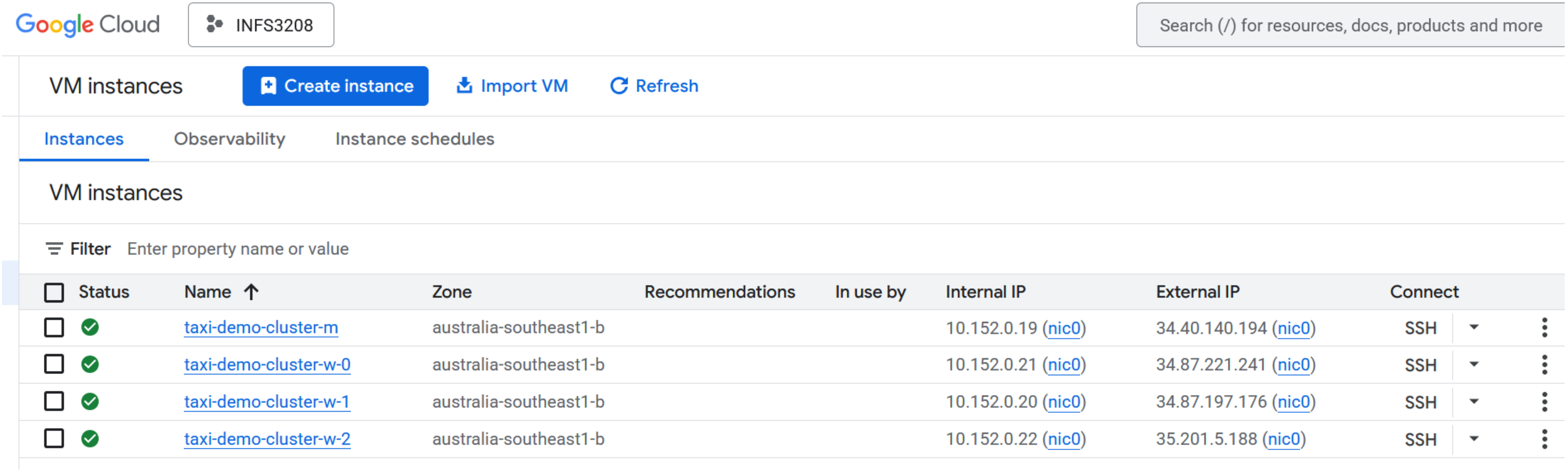
--master-boot-disk-size=50GB *--worker-boot-disk-size*=50GB

https://spark.apache.org

THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Demo – Create a Dataproc Cluster (3/3)

- After the creation of the cluster, you will see the corresponding VMs in GCE, one master node and three worker nodes.



DELETE the cluster to avoid incurring further charges.

```
gcloud dataproc clusters delete taxi-demo-cluster --region=australia-southeast1
```

Demo – Big Data Query and Aggregation (T1)

Create a local Python file named taxi_query_job.py:

- Read the NYC Yellow Taxi trip data for the year 2022 from a public BigQuery table.
- Perform a GROUP BY operation to aggregate data.
- Write the aggregated result to your GCS bucket in Parquet format.
- Output:

passenger_count	trip_count	avg_distance	total_fare
1	25702970	3.3944167300000	364911278.980000000
2	5298235	3.9966738580000	85679423.800000000
3	1389687	3.8205254920000	22654264.660000000
4	620920	4.0977473590000	10860404.080000000
5	633917	3.4099003020000	8873865.220000000
6	425746	3.4838078340000	6051653.970000000
7	205	1.6563414630000	14402.060000000
8	130	3.8932307690000	9166.670000000
9	39	3.7094871790000	3171.090000000

```
query = """
    SELECT
        passenger_count,
        COUNT(*) as trip_count,
        AVG(trip_distance) as avg_distance,
        SUM(fare_amount) as total_fare
    FROM
        trips
    WHERE
        passenger_count > 0 AND fare_amount > 0
    GROUP BY
        passenger_count
    ORDER BY
        passenger_count
"""
aggregated_data = spark.sql(query)
```

https://spark.apache.org

Demo – Machine Learning for Fare Prediction (T2)

Create a local Python file named taxi_ml_job.py:

- Read taxi data from BigQuery.
- Perform data cleaning and feature engineering.
- Train a Gradient-Boosted Tree (GBT) regression model.
- Evaluate the model's performance.

Output:

```
--------------------------------------------------
Model training complete.
Root Mean Squared Error (RMSE) on test data = 1.9940
--------------------------------------------------
```

```python
# --- 2. FEATURE ENGINEERING ---
# Combine feature columns into a single vector column
feature_cols = ["passenger_count", "trip_distance", "fare_amount", "PULocationID", "DOLocationID"]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
model_df = assembler.transform(model_df)

# --- 3. TRAIN the Model ---
# Split data into training and testing sets
(training_data, test_data) = model_df.randomSplit([0.8, 0.2], seed=42)

print(f"Training on {training_data.count()} records, testing on {test_data.count()} records.")

# Create and train the GBT Regressor model
gbt = GBTRegressor(featuresCol="features", labelCol="label", maxIter=10)
model = gbt.fit(training_data)

# --- 4. EVALUATE the Model ---
# Make predictions on the test data
predictions = model.transform(test_data)

# Evaluate the model using Root Mean Squared Error (RMSE)
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
```

https://spark.apache.org

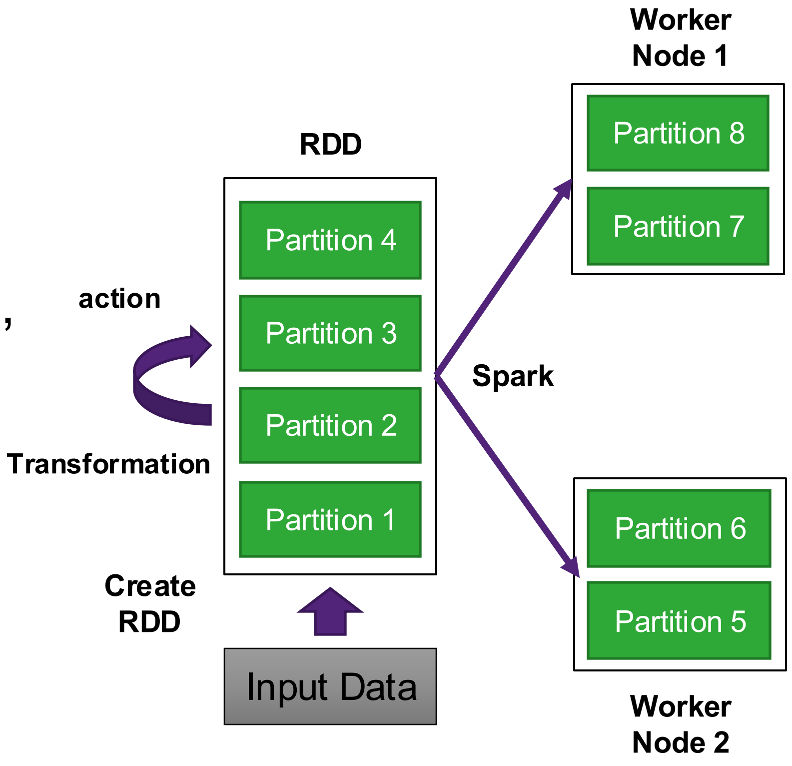
Outline

- Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- How Spark works

Resilient Distributed Dataset (RDD)

Resilient Distributed Dataset (RDD)

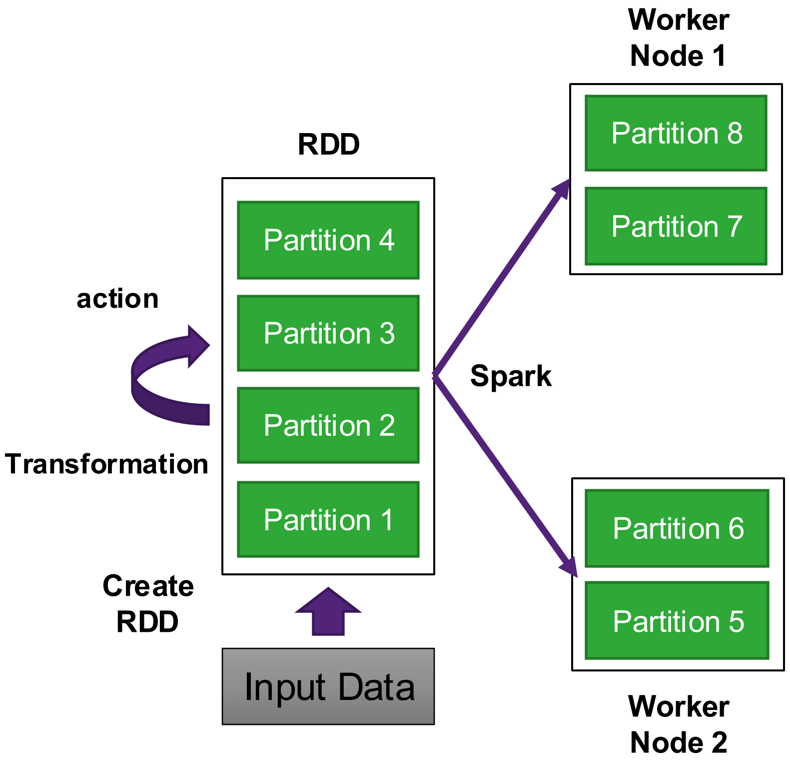
- RDD is a fundamental data structure of Spark.
- RDD is a read-only (i.e. immutable) distributed collection of objects/elements.
- Distributed: Each dataset in RDD is divided into logical partitions, which are computed by many worker nodes (computers) in the cluster.
- Resilient: RDD can be self-recovered in case of failure (support rebuild if a partition is destroyed).
- Datasets: JSON file, CSV file, text file etc.



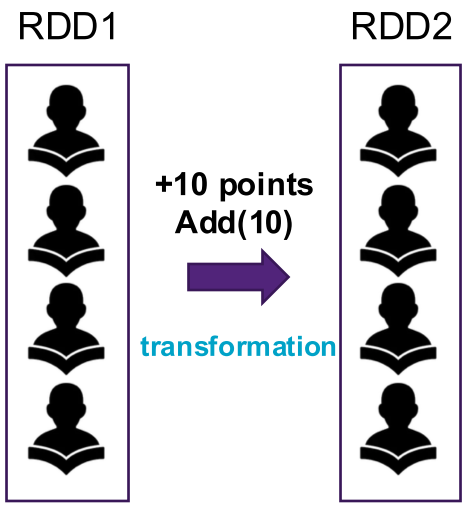
Resilient Distributed Dataset (RDD)

Resilient Distributed Dataset (RDD)

- In Spark, all work is expressed as
  - creating new RDDs or,
  - transforming existing RDDs or,
  - action on RDDs to compute a result.
- Data manipulation in Spark is heavily based on RDDs.
- RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.
- Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them.



RDD Operations



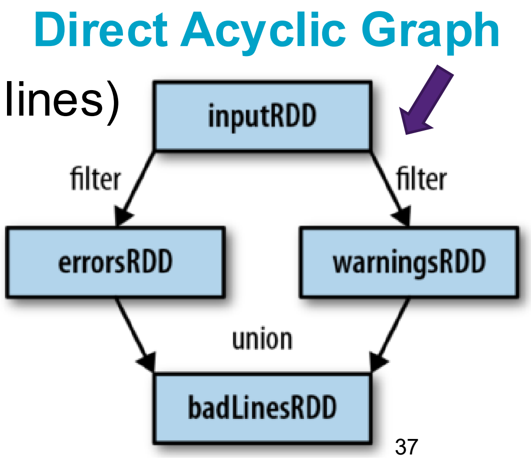
There are two types of RDD Operations: Transformation and Action

Transformations

- Transformations are operations on RDDs that return a new RDD.
- Transformed RDDs are computed lazily (only when you use them in an action)
- Many transformations are *element-wise* (working on one element at a time)
- introduce dependencies between RDDs to generate lineage graph

Example: a huge logfile (100TB), *log.txt*, with lines of messages, including normal, warning, and error messages. We want to display the warning and error messages.

- use a filter() transformation from inputRDD to errorsRDD (only transform error lines)
- use a filter() transformation from inputRDD to warningsRDD (only transform warning lines)
- use a union() transformation to display the final results.
- filter() has one ascendant, while union() has two ascendants.
  - Transformations can actually operate on any number of RDDs.

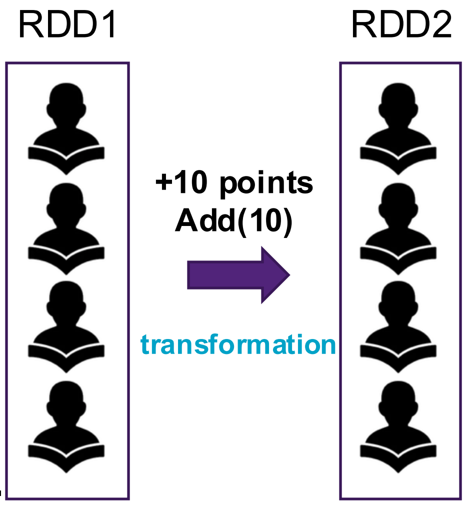


RDD Operations

There are two types of RDD Operations:

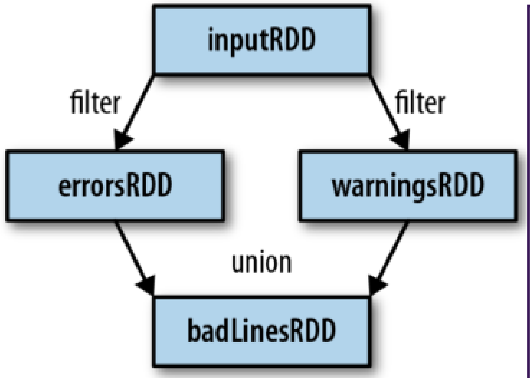
Actions

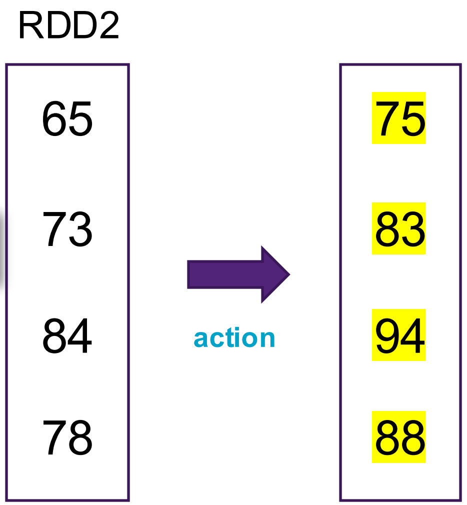
- trigger job execution that forces the evaluation of all the transformations
- must return a final value
- the values of action are stored to drivers or to the external storage system.
- It brings laziness of RDD into motion.



To print out some information about the badLinesRDD:

- count() action – returns the count as a number
- take() action – collects a number of elements from the RDD





THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Spark – Transformation

Common Transformations (15+) supported by Spark

Transformation	**Meaning**
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
sortByKey([ascending], [numPartitions])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join(otherDataset, [numPartitions])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

Spark – Action

Common Actions (10+) supported by Spark

Action	Meaning
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.

Outline

- Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- ➡ Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- How Spark works

Lazy Evaluation

Transformations on RDDs are evaluated or computed in a lazy manner:

- Spark will not begin to execute until it sees an action.

Lazy evaluation means that when a transformation on an RDD is called, the operation is not immediately performed.

Spark internally records metadata to indicate that this operation has been requested.

Spark can decide what the best way is to perform a series of transformations that are recorded.

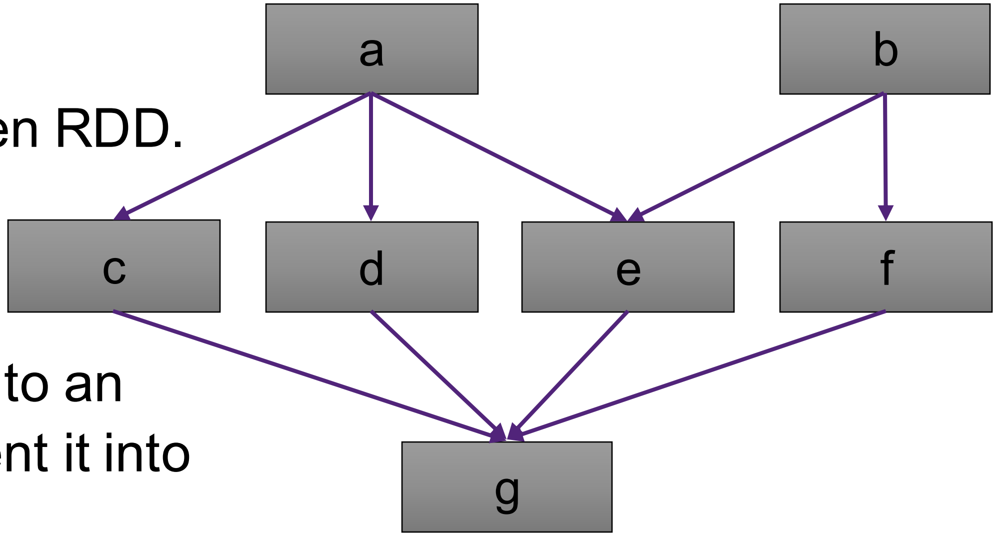
Spark uses lazy evaluation to reduce the number of passes (storage on disk)

- *Hadoop MapReduce, developers often have to spend a lot of time considering how to group together operations to minimize the number of MapReduce passes.*

https://www.wedowe.org/blog/theres-no-such-thing-as-laziness

RDD Lineage Graph

- Because of the lazy nature of RDD, dependencies between RDDs are logged in a lineage graph (or RDD operator graph or RDD dependency graph).
- Lineage graph can be regarded as a family tree of a given RDD.
- To get lineage graph:
  - METHOD: toDebugString:String
- When you run into an action, this local plan is submitted to an optimizer, which is going to do optimization and implement it into a physical plan (DAG) containing stages.
- Spark logs all transformations with a graph structure which can be optimized by graph optimization technology.
- Lineage graph can be used to re-build the RDDs



Outline

- Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- ➡ RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- How Spark works

RDD Persistence and Caching

- RDD persistence is an optimization technique in which saves the result of RDD evaluation.
- To significantly reduce computation overhead, we use RDD persistence to save the intermediate result so that we can use it further if required.
- Methods:
  - cache(): store all the RDD in-memory (default storage level: MEMORY_ONLY - executor's JVM heap as deserialized Java/Scala objects).
  - persist(level): can cache in memory, on disk, or off-heap memory.
- It is a key tool for iterative and interactive algorithms.
  - persistence process speeds up the further computation many times.
- The cache memory of the Spark is fault tolerant

RDD Persistence and Caching

When to use persistence and caching?

Caching is recommended in the following situations:

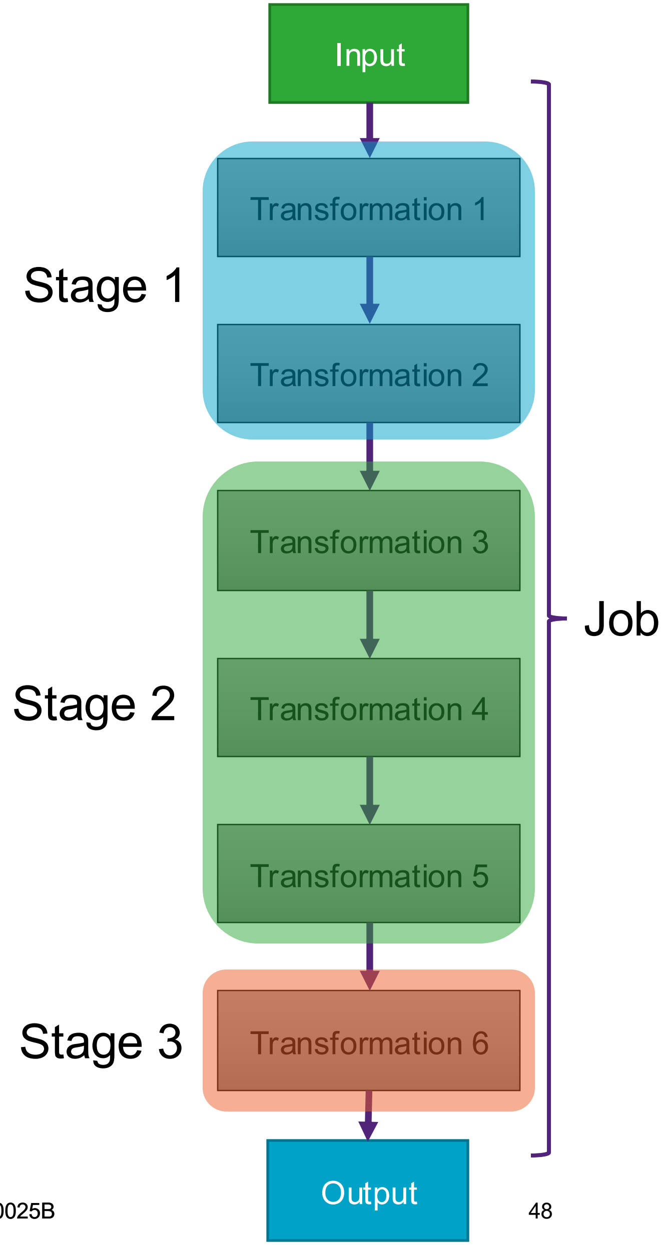
- For RDD re-use in iterative machine learning applications
  - E.g. in machine learning, some computed dataset is reused in each iteration (gradient-based)
- For RDD re-use in standalone Spark applications
  - In standalone Spark applications, multiple actions will perform on the same RDD
- When too many transformations on RDD or some computations are very expensive
  - Too many transformations will have a very long chain (long list of records)
  - Some expensive computations (transformations) take time to finish
  - caching can help in reducing the cost of recovery in the case one executor fails

Outline

- Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- ➡ Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- How Spark works

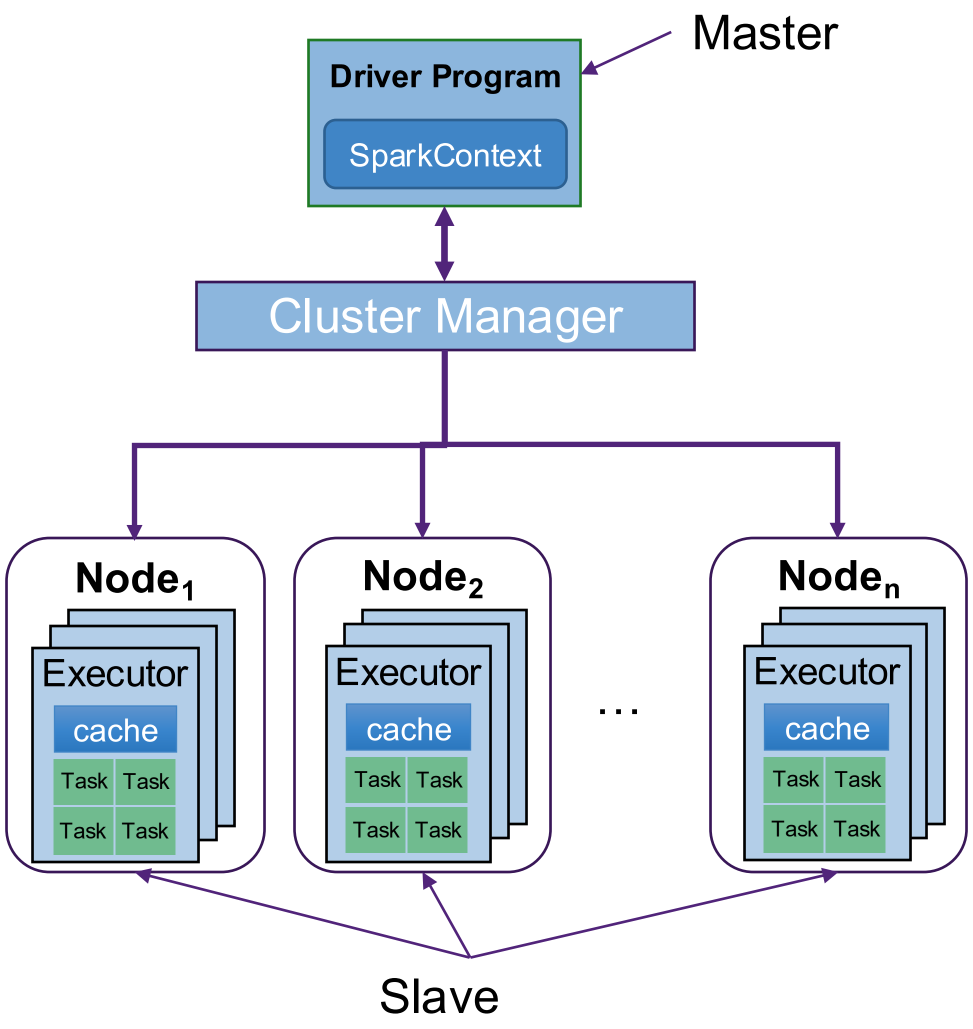
Terms in Spark

- Job:
  - A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- Stages:
  - Jobs are divided into stages.
  - E.g. Map or Reduce stages (similar with Hadoop).
  - Stages are divided based on computational boundaries.
- Tasks:
  - Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor.



Terms in Spark

- Spark Driver (program driver)
  - A separate process to execute user applications
  - creates SparkContext to schedule jobs execution and negotiate with cluster manager
- Executors
  - run tasks scheduled by driver
  - store computation results in memory, on disk or off-heap memory
  - interact with storage systems
- Master: The machine on which the Driver program runs
- Slave: The machine on which the Executor program runs



Terms in Spark

- SparkContext
  - The first step to create Apache Spark SparkContext, which is the main entry point to spark functionality
  - Configurable parameters of SparkContext for applications.
    - Spark use some of them to allocate resources on a cluster by executors (e.g. memory size, and cores).
  - Once SparkContext is created, it can be used to create RDDs (e.g. textfile method), broadcast variable, accumulator, and run jobs until SparkContext is stopped.
  - Functionalities:
    - Get the current status of application and set the configuration
    - Cancel a job/stage and closure cleaning in Spark
    - Access various services and programmable dynamic allocation (request/kill executors)
    - Access persistent RDDs and unpersist RDDs
    - etc.

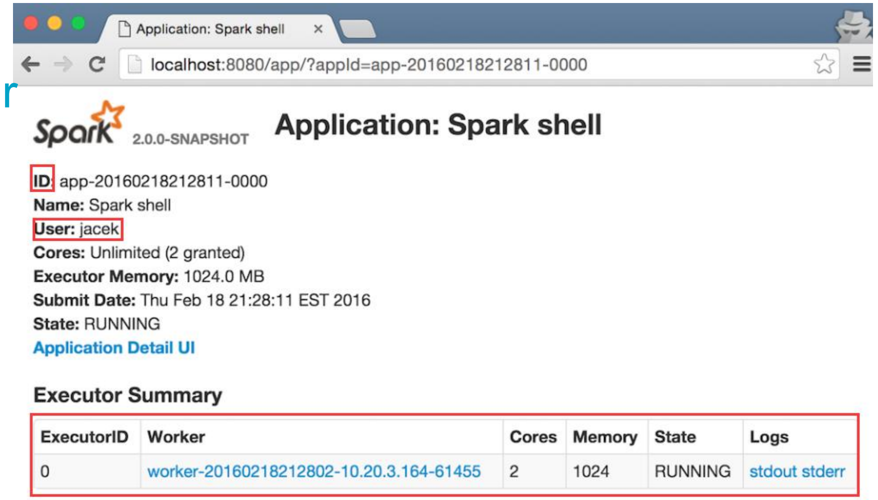
Cluster Manager

- Apache Spark is an in-memory computing engine to process Big Data and it can run a cluster, which has master and slave nodes (worker nodes).
- Cluster manager is to divide resources across applications and works as an external service for acquiring resources on the cluster.
- Spark supports pluggable cluster management, which handles starting executor processes, including:
  - Standalone Cluster Manager
  - Hadoop YARN
  - Apache Mesos

Cluster Manager – Spark Standalone

- Standalone mode is a simple cluster manager incorporated with Spark.
  - It makes it easy to setup a cluster that Spark itself manages and can run on Linux, Windows, or Mac OSX.
  - Often it is the simplest way to run Spark application in a clustered environment.
- It has masters and number of workers with configured amount of memory and CPU cores.
- Spark allocates resources based on the core and an application will grab all the cores in the cluster by default.
- To check the application, each Apache Spark application has a Web User Interface
  - provides information of executors, storage usage, running task in the application.
  - monitors cluster and job statistics
  - checks detailed log output for each job.
  - can reconstruct the application's UI after the application exits, if an application has logged event for its lifetime.

Standalone Cluster



Cluster Manager – Apache Mesos

- Apache Mesos is an open-source cluster manager developed in AMPLab at UC Berkeley.
- Apache Mesos abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively.
- Mesos runs on nodes of a cluster and provides an APIs to applications for managing and scheduling resources.
- Mesos is one of the cluster configurations in which Spark can operate.
- Mesos vs Spark Standalone vs YARN
  - There are differences in terms of **high availability, security, monitoring** (depends on your goals)
  - They have provided almost all the features as a cluster manager
  - Standalone is the easiest for beginners
  - Mesos has richer resource scheduling capabilities (allows fine-grained sharing options)
  - YARN can share and configure the same pool of cluster resources between all frameworks that are running on YARN

https://data-flair.training/blogs/apache-spark-cluster-managers-tutorial/

Outline

- Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- How Spark works

Directed Acyclic Graph (DAG)

- Directed Acyclic Graph (DAG) is a set of vertices and edges
  - vertices represent the RDDs;
  - edges represent the operation to be applied on RDD.
- DAG is a finite directed graph (finite vertices and edges) with no directed cycles.
- It contains a sequence of vertices such that every edge is directed from earlier to later in the sequence.
- DAG operations can do better global optimization than other systems like MapReduce.
- On the calling of Action, the created DAG submits to DAG Scheduler which further splits the graph into the stages of the job.
- The DAGScheduler splits the Spark RDD into stages based on applied transformation.
- DAG vs RDD Lineage Graph: Lineage is a portion of a DAG that leads to the creation of that particular RDD.

Lineage Graph v.s. Directed Acyclic Graph (DAG)

Lineage Graph:

- Captures the sequence of transformations applied to the data.
- A lineage graph is a record of how a particular piece of data was derived. For each RDD (Resilient Distributed Dataset), there's a lineage graph that traces back the sequence of transformations (like `map`, `filter`, etc.) that produced it from the source data.
- The lineage graph aids in recovery during data loss. If a partition of an RDD is lost, Spark can recompute it from the source data using the lineage graph, thus ensuring fault tolerance without data replication.
- A lineage graph represents individual transformations and their dependencies.

Directed Acyclic Graph (DAG):

- Models the entire execution plan for a Spark job.
- When an action (like `count`, `collect`, etc.) is called in Spark, it constructs a DAG to model the stages and tasks required for that action. This DAG represents the optimized execution plan for the computation.
- The DAG scheduler divides the DAG into stages. Each stage contains as many transformations as possible that have narrow dependencies. Wide dependencies introduce stage boundaries. This means that transformations which can be done in parallel without shuffling data are grouped into a single stage.
- The DAG represents the entire job, and Spark executes the job by executing these stages in topological order. If one stage fails, only that stage is recomputed, leveraging the DAG's ability to identify the minimum set of tasks to recompute.

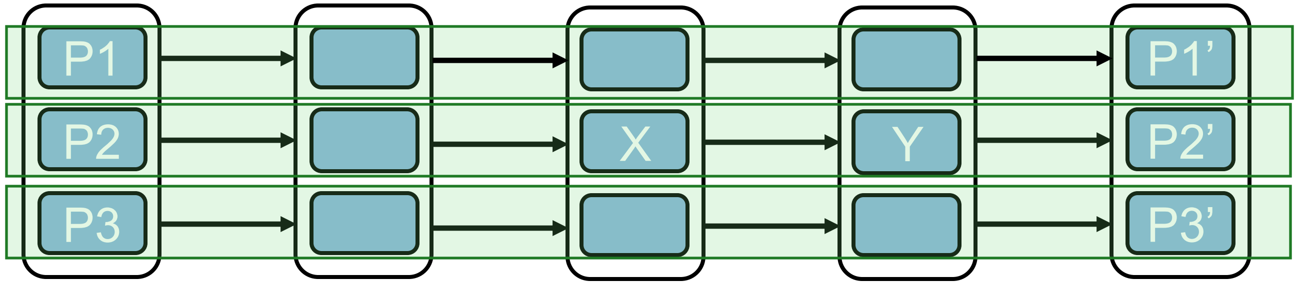
Outline

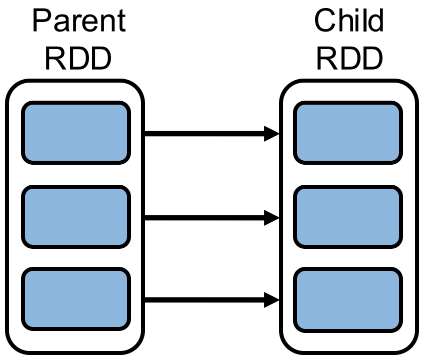
- Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- ➡ Narrow and Wide Dependencies
- Shuffle
- How Spark works

Narrow and Wide Dependencies

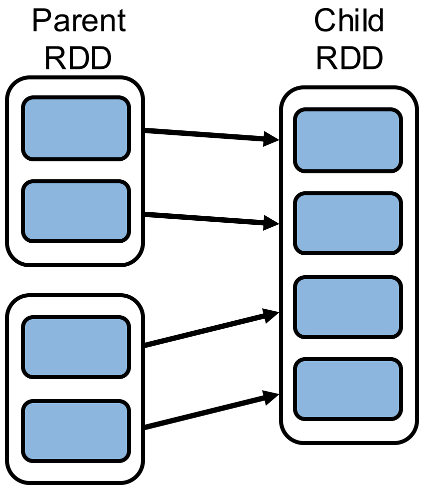
There are two types of transformations:

- **Narrow transformation (dependencies)**
  - each partition of the parent RDD is used by at most one partition of the child RDD
  - allow for pipelined execution on one cluster node
  - failure recovery is more efficient as only lost parent partitions need to be recomputed
  - *Example: map, flatmap, filter, sample, union, etc.*

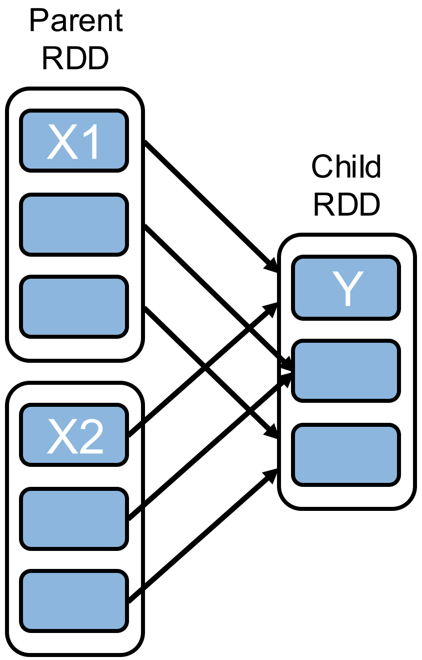




map, filter



union

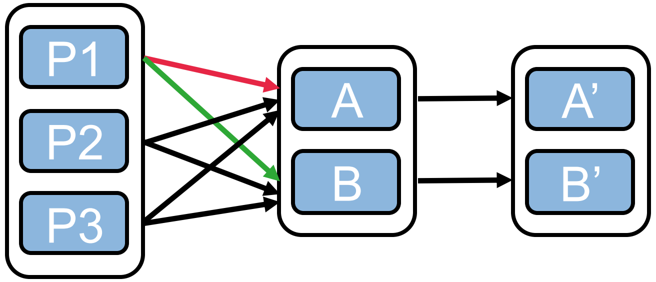


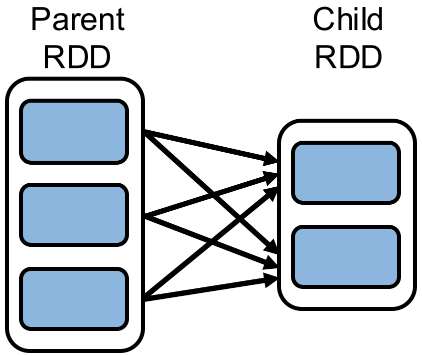
Join with inputs co-partitioned

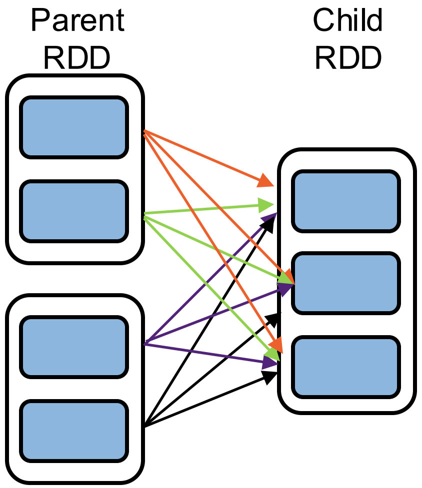
Narrow and Wide Dependencies

There are two types of transformations:

- **Wide transformation (dependencies)**
  - multiple child partitions may depend on one parent partition
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - a complete re-computation is needed, if some partition is lost from all the ancestors
  - *Example: groupbyKey()* and *reducebyKey()*.







Outline

- Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- How Spark works

Shuffle

Calculate how many items are sold for each store?

trasactionID	itemID	itemPrice	storeID
00000001	131	13.49	1
00000002	211	55.99	74
00000003	432	109.99	54
…	…	…	…
99999999	123	10.0	41
100000000	411	3.0	99

Example: Given 100k sales records of 100 stores, how to calculate how many items have been sold for each store?

Solution – Database manner:

- use groupby with a key dividing the entire data sheet into different groups: e.g. all records of sold items for storeID=1 will be in one group.
- Use aggregate function (count()) to calculate how many rows within each group: e.g. 10k rows for storeID=1 and 12k rows for storeID=99.
- Return the values as the final results

It's feasible for modern databases, such as MS SQL Server, MySQL, PostgreSQL, etc.

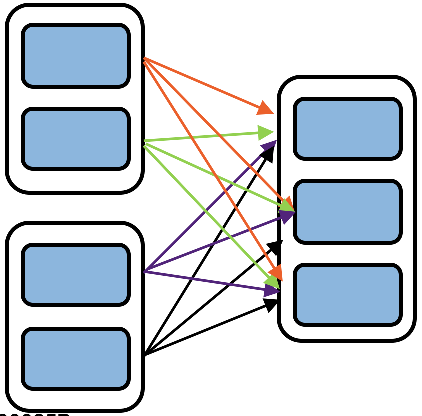
What if we have much more data? E.g. 1,000 times more?

Shuffle

- In the traditional database, we divide the entire data into different groups and calculate result of each group.
- In RDD, data are distributed across multiple partitions on nodes, traditional groupby operation can cause expensive communication cost.
- In Spark, we can group the records with the same key to one partition for further operation (e.g. reduceByKey, groupByKey).
- **Example**: there are 10 worker nodes and each node calculates records of 10 stores: Node1 – Store 1 to Store 10, Node 2 – Store 11 to Store 20, …
- To do this computation, we need to re-distribute (shuffle) data so that it's grouped differently across partitions.
- In Spark, the huge database is distributedly stored in RDDs across partitions in different nodes, some transformation operations will trigger shuffle (re-distributing) data.
- Shuffle typically involves copying data across executors and machines, making the shuffle a complex and costly operation.

Calculate how many items are sold for each store?

trasactionID	itemID	itemPrice	storeID
00000001	131	13.49	1
00000002	211	55.99	74
00000003	432	109.99	54
…	…	…	…
99999999	123	10.0	41
100000000	411	3.0	99



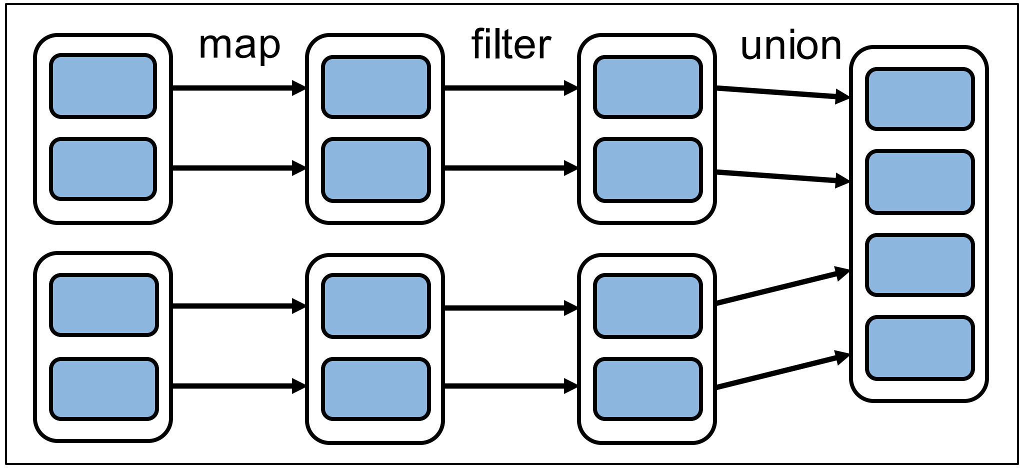
Shuffle

- Shuffle is an expensive operation since it involves disk I/O, data serialization, and network I/O.
- Shuffle also generates a large number of intermediate files on disk.
- Transformations that ***cause communications across nodes when repartitioning*** will cause a shuffle in Spark:
  - join
  - groupByKey
  - reduceByKey
  - combineByKey
  - etc.
- One direct method to check if the shuffle will occur: .toDebugString

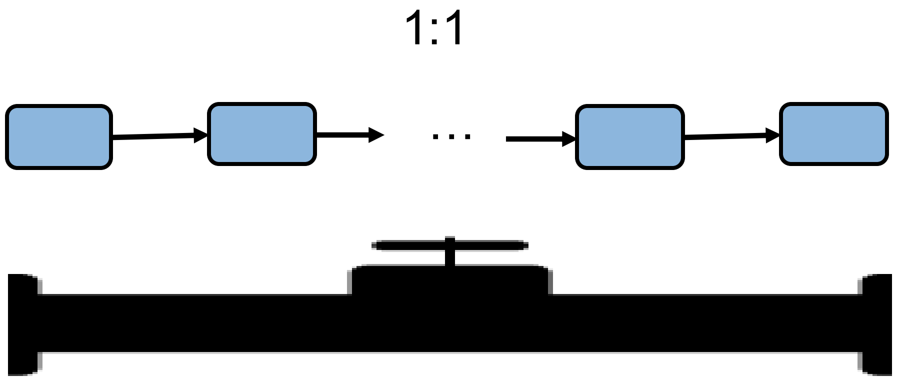
you should always assume that all wide transformations will trigger a shuffle.

How Stage Generated

- When the program runs into an action (like collect), the graph of transformations is submitted to a DAG Scheduler.
- DAG scheduler aims to optimize the performance by dividing transformation graph into different stages.
- DAG scheduler pipelines transformation operations together to optimize the graph.
- E.g. Many map operators can be scheduled in a single stage.



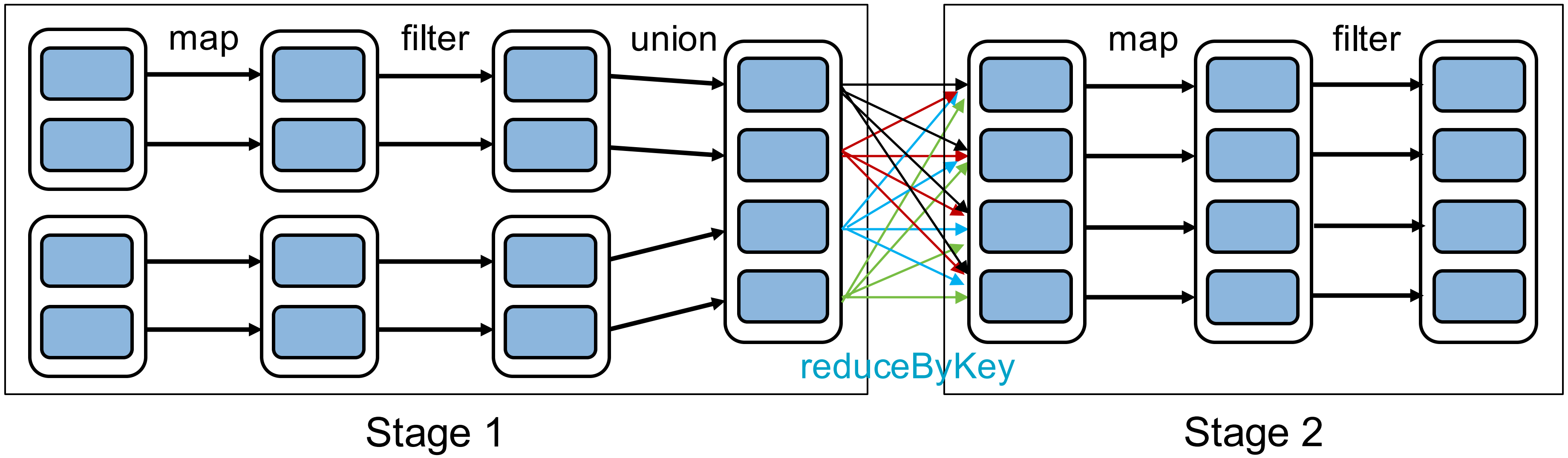
One stage



One stage

How Stage Generated

- When the transformations can trigger shuffle, DAG Scheduler will divide the transformations into different stages.
- In the following example, reduceByKey triggers shuffle and DAG Scheduler divide the pipeline into two stages.



How Stage Generated

The DAG scheduler divides operator graph into stages according to the types of dependencies:

- Narrow dependency – all the transformations can be pipelined into one stage;
- Wide dependency – divide the transformations into different stages.

The final result of a DAG scheduler is a set of stages.

The stages are passed on to the Task Scheduler.

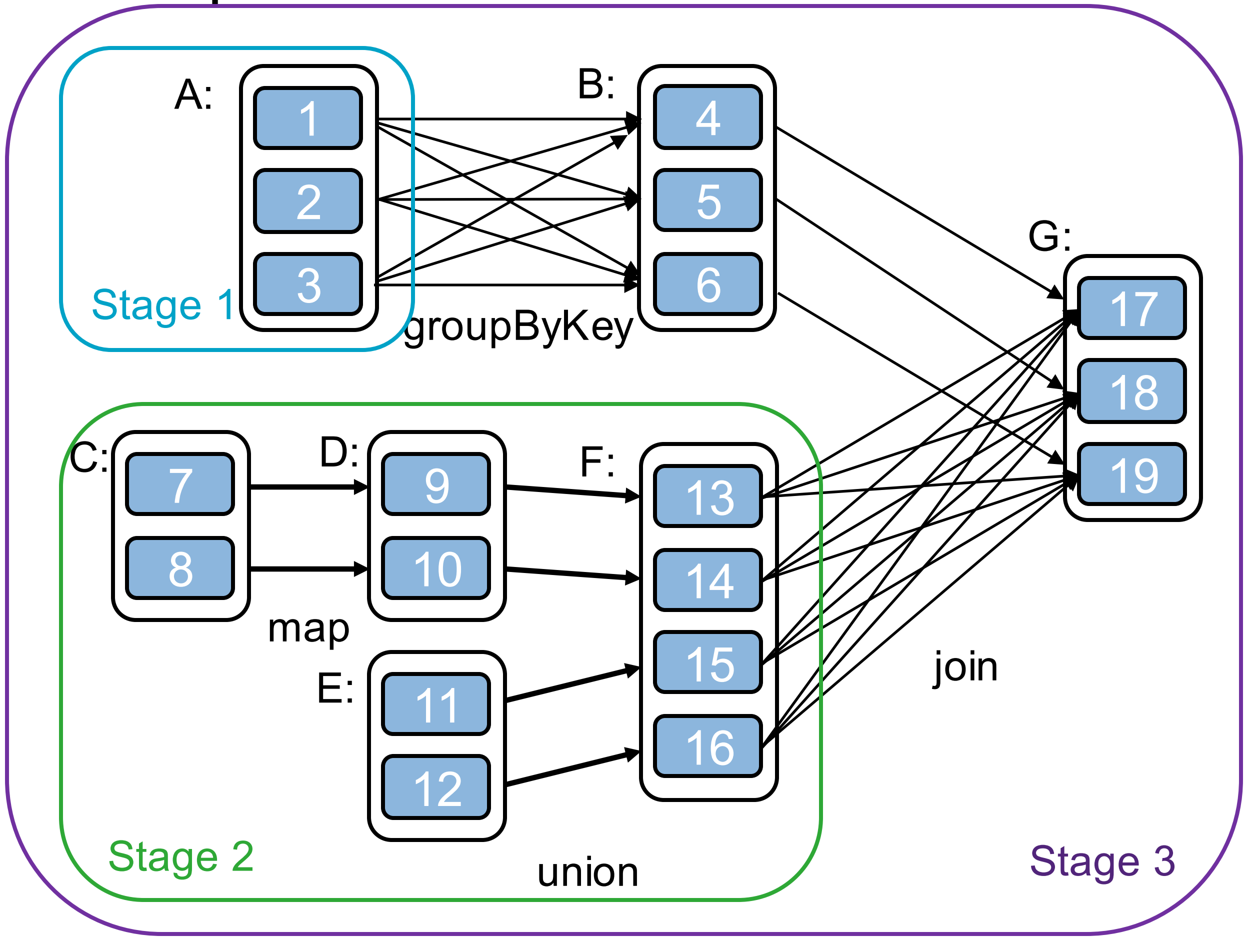
The task scheduler launches tasks via cluster manager. (Spark Standalone or Hadoop YARN or Apache Mesos).

The task scheduler doesn't know about dependencies among stages.

How Stage Generated

Let's look at another complicated example:

- RDDs: A – G
- Partitions: 1 – 19
- Narrow transformations:
  - map, union
- Wide transformation:
  - groupByKey, join



Outline

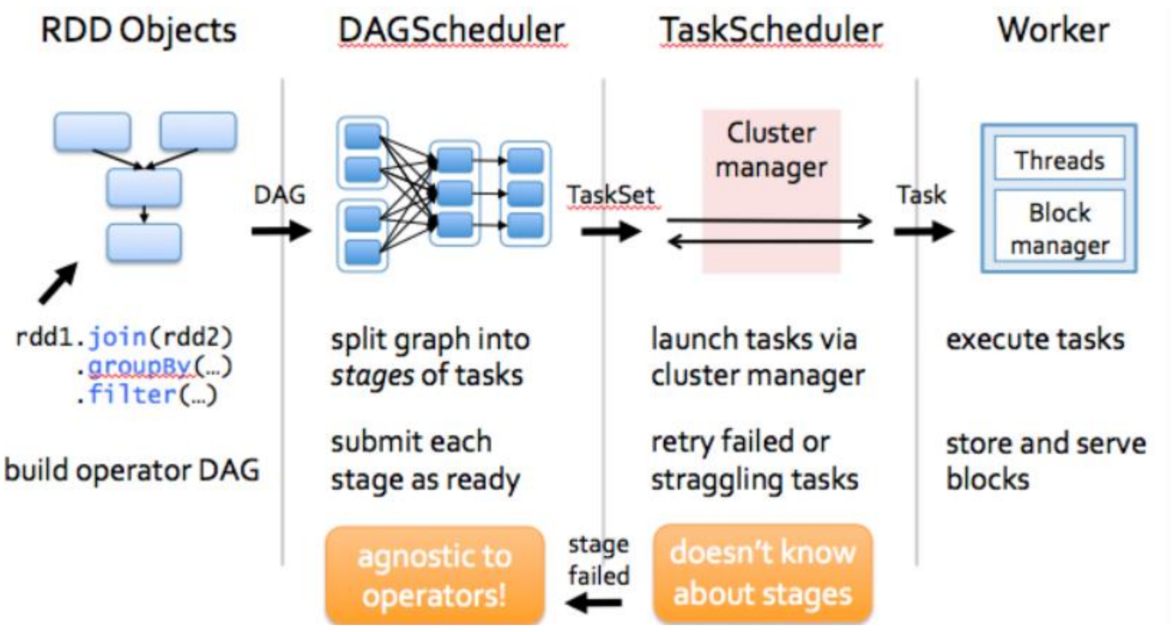
- Background
- Apache Spark and its Characteristics
- Resilient Distributed Dataset (RDD) and its operations: Transform & Action
- Lazy evaluation and RDD Lineage graph
- RDD Persistence and Caching
- Terms in Spark
- Directed Acyclic Graph (DAG)
- Narrow and Wide Dependencies
- Shuffle
- ➡ How Spark works

The University of Queensland
Australia

How Spark Works

Based on aforementioned RDD contents, let us have a look at the summary of the RDD running process in the Spark architecture

1. Create an RDD object;
2. SparkContext is responsible for calculating the dependencies between RDDs and building DAGs;
3. DAG Scheduler is responsible for decomposing the DAG graph into multiple stages, each stage containing multiple tasks,
4. Task scheduler launches tasks to distribute across the worker nodes via cluster manager (Standalone or Mesos or YARN). The task scheduler does not know about dependencies among stages.



http://web.utk.edu/~wfeng1/spark/introduction.html

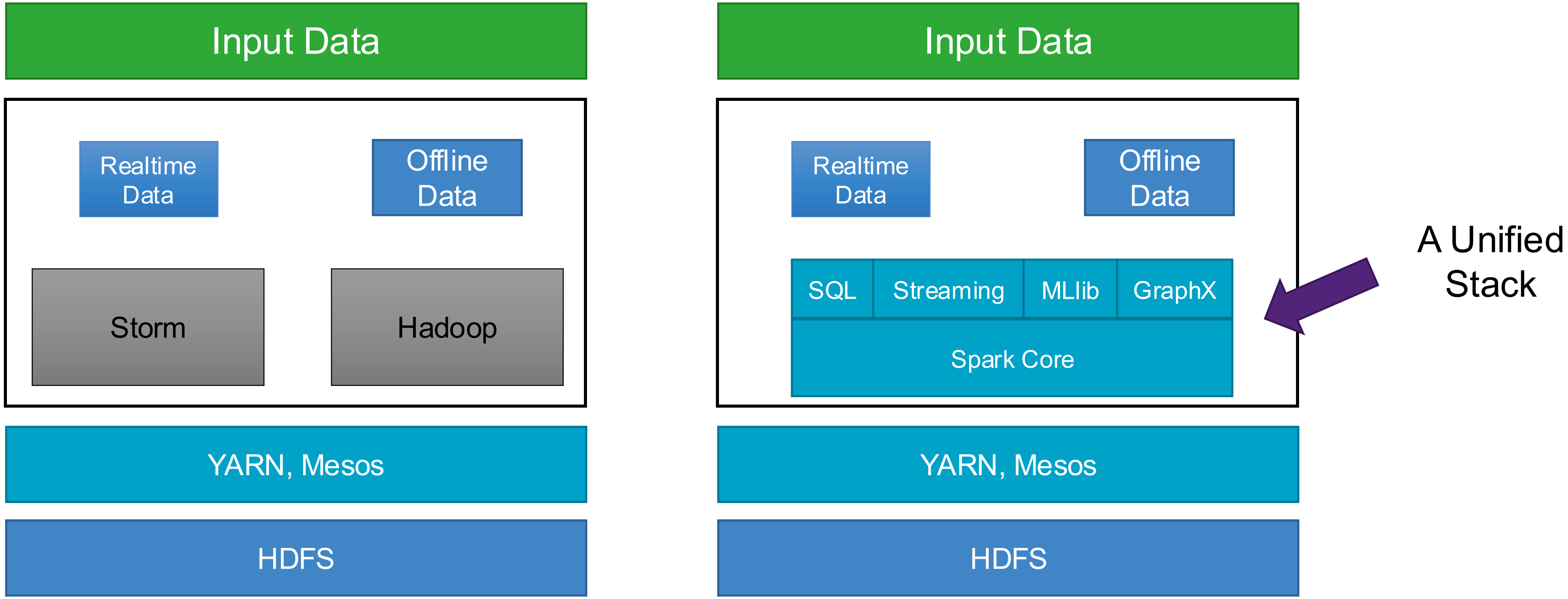
Fault Tolerance

Apache Spark fault tolerance property :

- RDD has a capability of handling if any loss/fault/failure occurs.
- Transformations applied to RDDs:
  - Narrow dependency: if one partition fails, simply re-calculate the corresponding partition in the parent RDD.
  - Wide dependency: if one partition fails, need to re-calculate all computations in the parent RDD.
- lineage graph (logical execution plan)
  - Too long or some transformations are too expensive (checkpoint: write RDDs on Disks)
  - Checkpoint for wide dependent transformation.

Spark Deployment

Deployment architecture comparison



Spark vs. Hadoop

Parameters	Spark	Hadoop
Data Storage	Spark stores data in-memory.	Hadoop stores data on disk.
Fault tolerance	Spark's data storage model, resilient distributed datasets (RDD) guarantees fault tolerance.	It uses replication to achieve fault tolerance.
Line of code	Apache Spark is project of 20,000 Line of code.	Hadoop 2.0 has 1,20,000 Line of code
Speed	It is Faster due to In-memory computation.	It is relatively slower than Spark.
OS Support	• Linux • Windows • Mac OS	• Linux
High level language	• Scala • Python • Java • R	• Java
Streaming data	Spark can be used to process as well as modify real-time data with Spark streaming.	With Hadoop Map-Reduce one can process batch of stored data.
Machine Learning	Spark has its own set of Machine learning libraries (MLib).	Hadoop requires interface with other Machine learning library. Eg: Apache Mahout.

https://datasciencegyan.com/spark-vs-hadoop/

Pros and Cons of Apache Spark

Advantages:

- Spark is fast in data processing (In-memory computation technology).
- Spark has significantly less computation resources than MapReduce.
- Spark has more complicated computing operations than MapReduce.
- Spark support many languages: Java, Python, R and Scala.
- Spark can easily integrate with almost all Big data technologies (incorporated with Hadoop ecosystem).
- It is fault tolerant (RDD) and easily scalable.
- Spark provides 128-bit encryption and SSL support for its network.

https://youthgiri.com/it-world/advantages-and-disadvantages-of-spark/12936/.html/

Pros and Cons of Apache Spark

Disadvantages:

- In-memory computing needs large memory to store all the data which makes hardware very pricy.
- Spark does not support genuine real-time processing (micro-batch). It processes data in the micro-batch which we can make it as small as 1 second (second-level). Apache Storm can perform at millisecond level.
- Spark doesn't have its own file system. It uses the file system of other technology like HDFS, Hive, etc. (Small file problem)

https://youthgiri.com/it-world/advantages-and-disadvantages-of-spark/12936/.html/

Reading Materials

1. https://data-flair.training/blogs/rdd-lineage/
2. http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/
3. http://web.utk.edu/~wfeng1/spark/introduction.html
4. https://www.tutorialspoint.com/apache_spark/index.htm
5. https://techvidvan.com/tutorials/spark-tutorial/

Next (Week 9) Topic:

Spark Applications