General Purpose Computing Systems II: In-memory Computation & Spark

Shiow-yang Wu (吳秀陽)

CSIE, NDHU, Taiwan, ROC

Lecture material is mostly home-grown, partly taken with permission and courtesy from Professor Shih-Wei Liao of NTU.

Outline

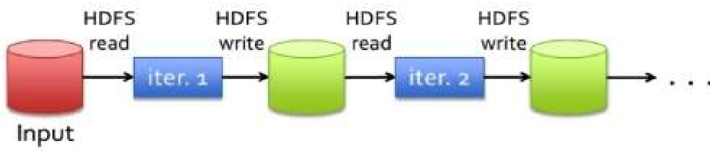
- Introduction
  - Motivation
  - Solution: In-memory computation
- Challenges
  - Designing a shared data abstraction with
    - Scalability
    - Data locality
    - Fault tolerance
- Resilient Distributed Datasets(RDD)
  - Design policy
  - Programming model
  - Implementation of RDD

Outline

- DataFrames
  - What and why
  - DataFrames vs RDDs
  - Operations/transformations
- Datasets
  - What and why
  - Datasets vs DataFrames
  - Compile-time type safety and object-oriented programming
- RDD vs DataFrame vs Dataset
- What's new in Spark 3.5?

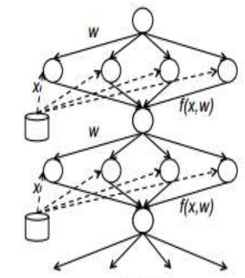
Problems with Hadoop MapReduce

- When doing iterative computation
  - Bad performance due to **replication & disk I/O**
  - Even worse: **Communication overheads** in the distributed file system



Problems with Hadoop MapReduce

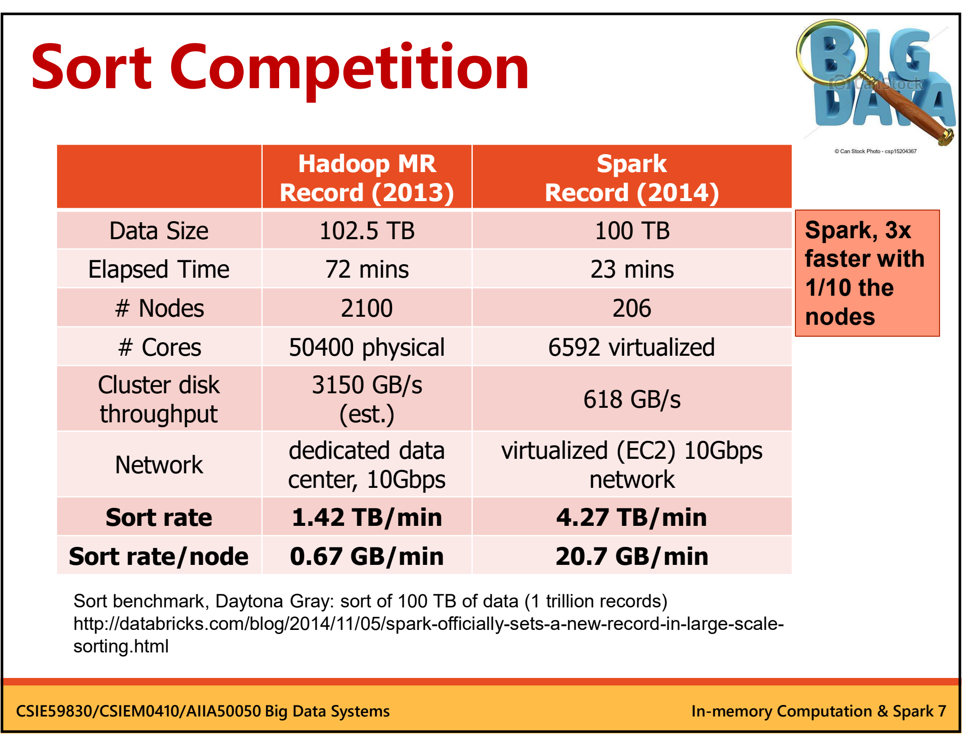
- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
  - More **complex**, **iterative** multi-pass analytics (e.g. ML, graph)
  - More **interactive** ad-hoc queries
- Requires intensive disk I/O
  - Intermediate data is always written to local disk make poor performance
  - solution: Apache Spark's in-memory computing



Solution: Keep the Data in Memory

- Apache Spark's in-memory computing
  - 10-100X faster than disk

- Sharing at memory speed
- Latest release: Spark 3.5.0 released (Sep 13, 2023)

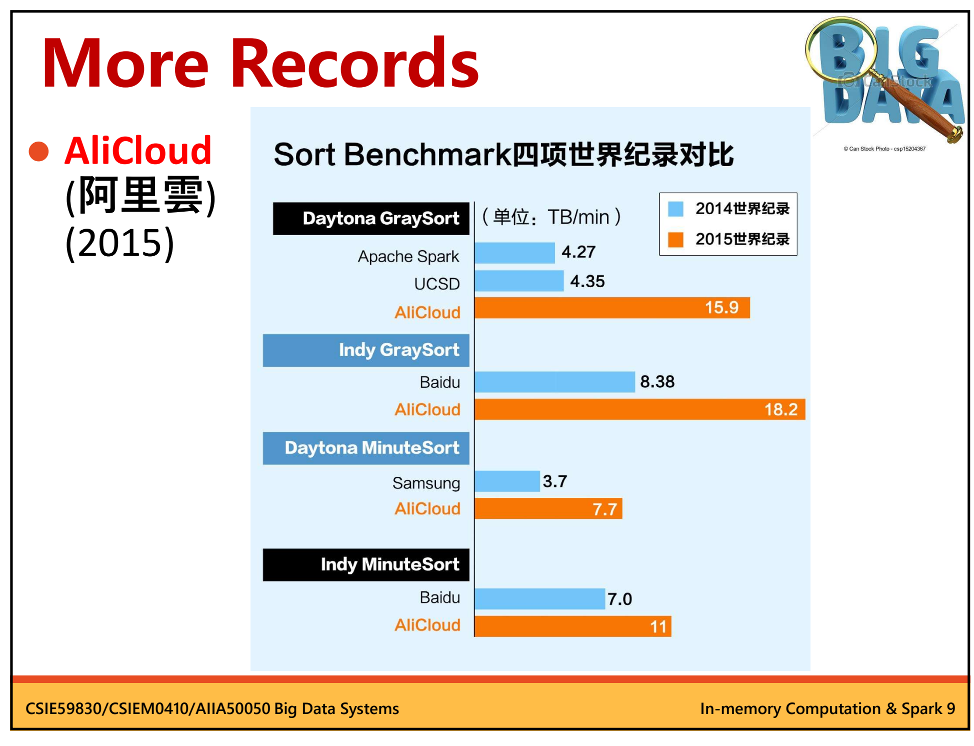


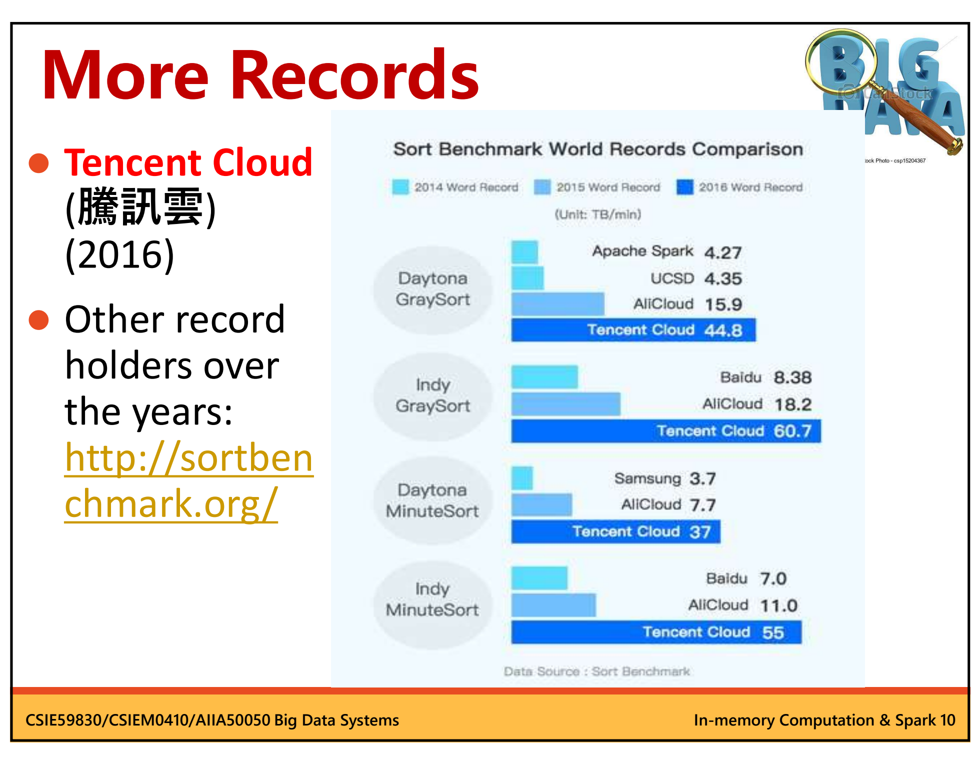
	Hadoop MR Record (2013)	Spark Record (2014)
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**

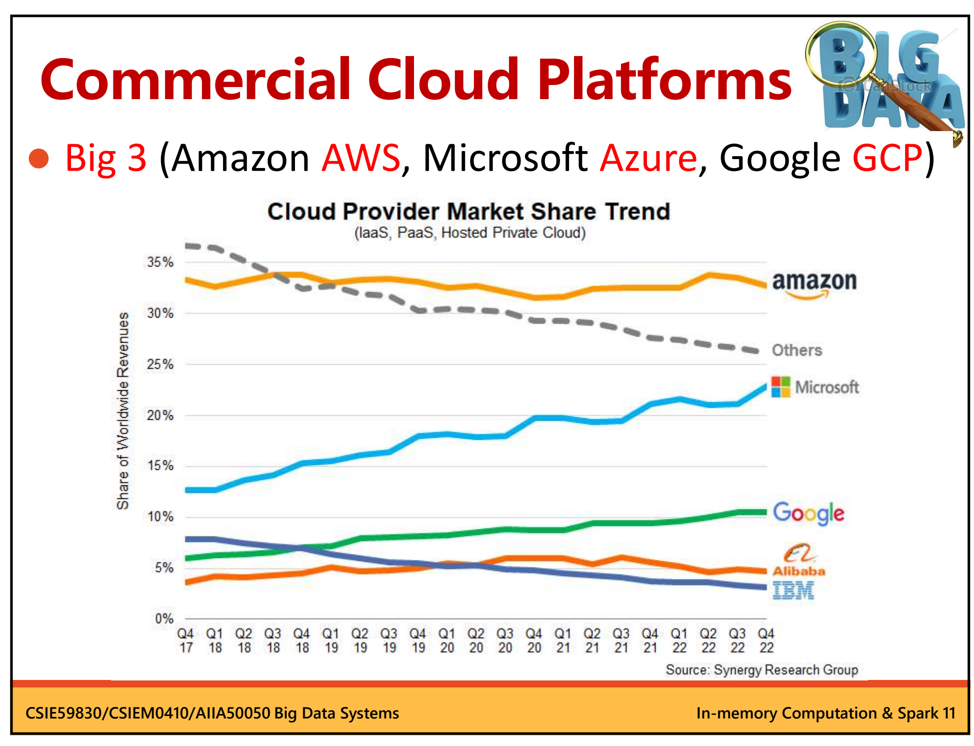
World Record on Sorting

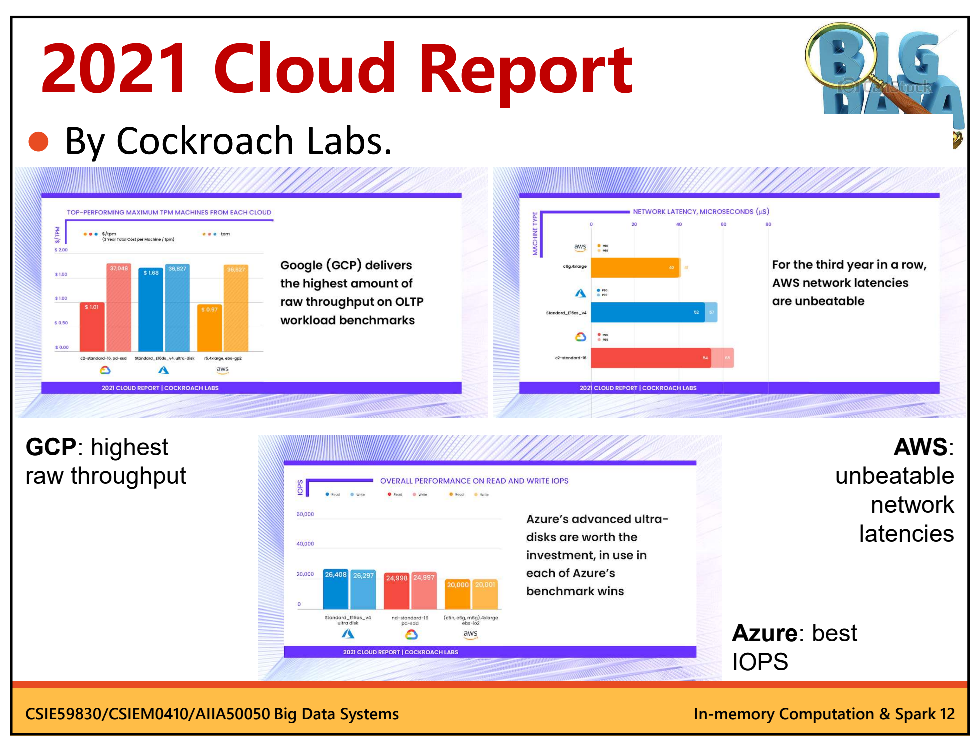
	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

CSIE59830/CSIEM0410/AIIA50050 Big Data Systems In-memory Computation & Spark 8


More Records
AliCloud
(阿里雲)
(2015)
Sort Benchmark四项世界纪录对比
（单位：TB/min）
2014世界纪录
2015世界纪录
Daytona GraySort
Apache Spark 4.27
UCSD 4.35
AliCloud 15.9
Indy GraySort
Baidu 8.38
AliCloud 18.2
Daytona MinuteSort
Samsung 3.7
AliCloud 7.7
Indy MinuteSort
Baidu 7.0
AliCloud 11
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 9


More Records
Tencent Cloud
(騰訊雲)
(2016)
Other record holders over the years: http://sortbenchmark.org/
Sort Benchmark World Records Comparison
2014 Word Record
2015 Word Record
2016 Word Record
(Unit: TB/min)
Daytona GraySort
Apache Spark 4.27
UCSD 4.35
AliCloud 15.9
Tencent Cloud 44.8
Indy GraySort
Baidu 8.38
AliCloud 18.2
Tencent Cloud 60.7
Daytona MinuteSort
Samsung 3.7
AliCloud 7.7
Tencent Cloud 37
Indy MinuteSort
Baidu 7.0
AliCloud 11.0
Tencent Cloud 55
Data Source : Sort Benchmark
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 10


Commercial Cloud Platforms
Big 3 (Amazon AWS, Microsoft Azure, Google GCP)
Cloud Provider Market Share Trend
(IaaS, PaaS, Hosted Private Cloud)
Share of Worldwide Revenues
35%
30%
25%
20%
15%
10%
5%
0%
amazon
Others
Microsoft
Google
Alibaba
IBM
Q4 17
Q1 18
Q2 18
Q3 18
Q4 18
Q1 19
Q2 19
Q3 19
Q4 19
Q1 20
Q2 20
Q3 20
Q4 20
Q1 21
Q2 21
Q3 21
Q4 21
Q1 22
Q2 22
Q3 22
Q4 22
Source: Synergy Research Group
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 11


2021 Cloud Report
By Cockroach Labs.
Google (GCP) delivers the highest amount of raw throughput on OLTP workload benchmarks
For the third year in a row, AWS network latencies are unbeatable
GCP: highest raw throughput
AWS: unbeatable network latencies
Azure's advanced ultra-disks are worth the investment, in use in each of Azure's benchmark wins
Azure: best IOPS
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 12

2022 Cloud Report

- Detailed comparison with 56 cloud instances and 3000+ benchmark runs.
- Testing Big 3 performance with benchmarks on OLTP, CPU, Network, and Storage I/O.
- No overall winner
- 6 key insights
- (https://www.cockroachlabs.com/guides/2022-cloud-report/)

The Working Set Idea

- Peter Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, May 1968.
  - http://dl.acm.org/citation.cfm?id=363141
- **Idea**: conventional programs generally exhibit a high degree of locality, returning to the same data over and over again.
- Operating system, virtual memory system, compiler, and micro architecture are designed around this assumption!
- Exploiting this observation makes programs run 100X faster than simply using plain old main memory in the obvious way.

Spark

- Exploit the working set idea
- Fast and expressive cluster computing system interoperable with Apache Hadoop
- Improves efficiency through:
  - In-memory computing primitives
  - General computation graphs (DAG)

  → Up to 100× faster (2-10× on disk)
- Improves usability through:
  - Rich APIs in Scala, Java, Python, R
  - Interactive shell

  → Often 2-10× less code
- The core processing engine of BDAS (Berkeley Data Analytics Stack)

Spark History

- 2008 – Yahoo! Hadoop team collaboration w Berkeley AMP/RAD Lab to begin the project
- 2009 – Spark example built for Nexus(a common substrate for cluster computing) -> Mesos(a distributed systems kernel)
- 2011 – "Spark is 2 years ahead of anything at Google" – Conviva(a company for online video optimization and analytics) seeing good results w Spark
- 2012 – Yahoo! working with Spark/Shark(now Spark SQL)
- 2013 – donated to Apache

Spark History

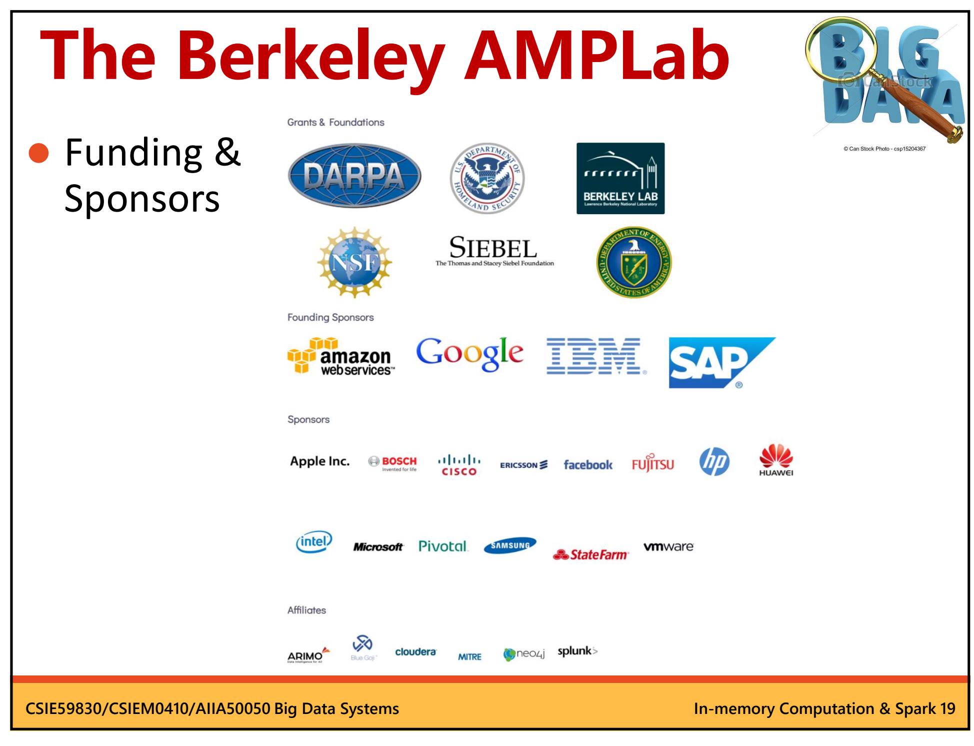
- 2014 – became a Top-Level Apache Project
- 2014(Nov) – Databricks(company) set a new world record in sorting using Spark
- 2015 – 1000+ contributors, one of most active Apache projects/open source big data projects
- 2016 – Spark 2.0 released. Spark SQL one of the best Big Data SQL engines, new Structured Streaming APIs
- 2020 – Spark 3.0 released. Spark SQL/Core, adaptive query execution, dynamic partition pruning, languages/systems upgrades, improved performance…
- 2021 and beyond – Blooming Spark Ecosystem
- Latest release: Spark 3.5.0 (Sep 13 2023)

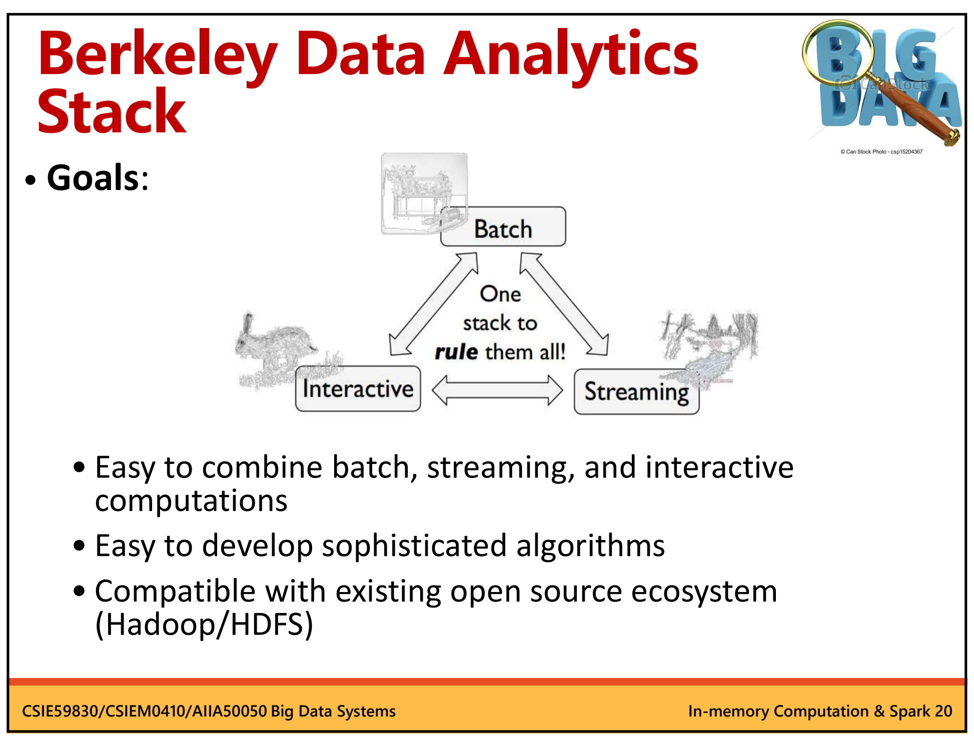
Berkeley Data Analytics Stack(BDAS)

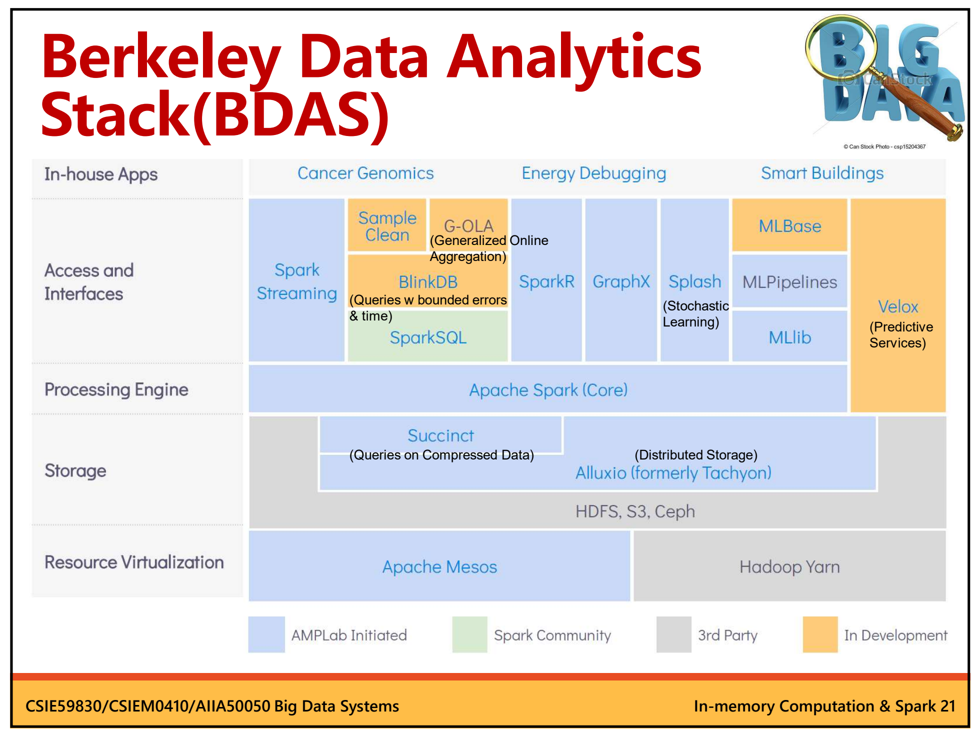
- An open source software stack that integrates software components being built by the Berkeley AMPLab to make sense of Big Data
- The AMPLab was launched at Jan 2011, not active after last publication at 2017.
- Goal: Next Generation of Analytics Data Stack
  - Berkeley Data Analytics Stack (BDAS)
  - Release as Open Source

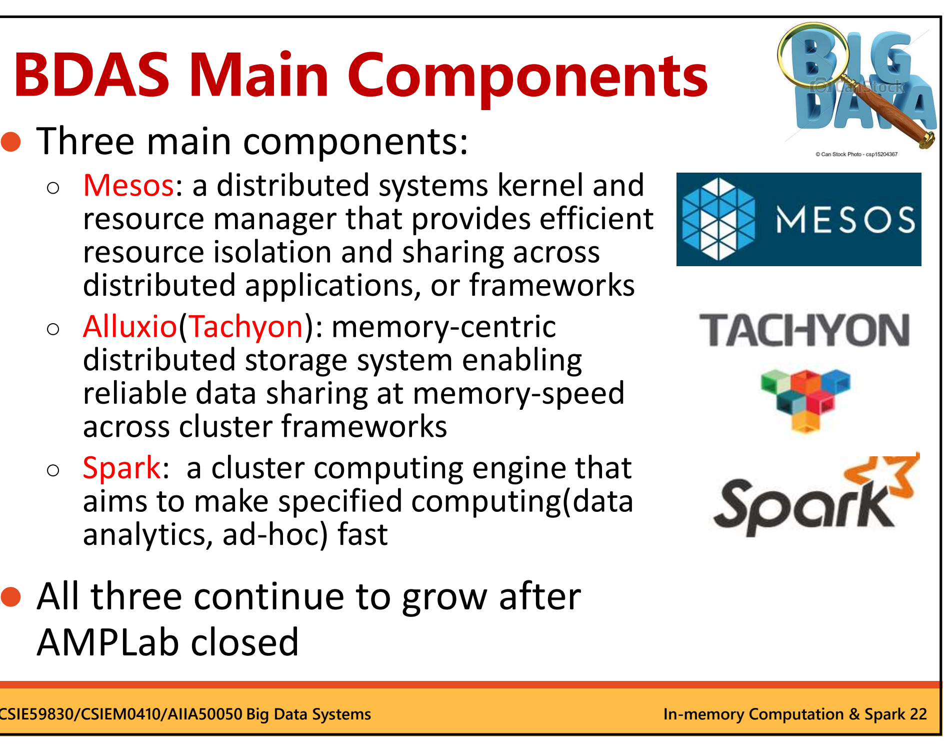
amplab UC BERKELEY

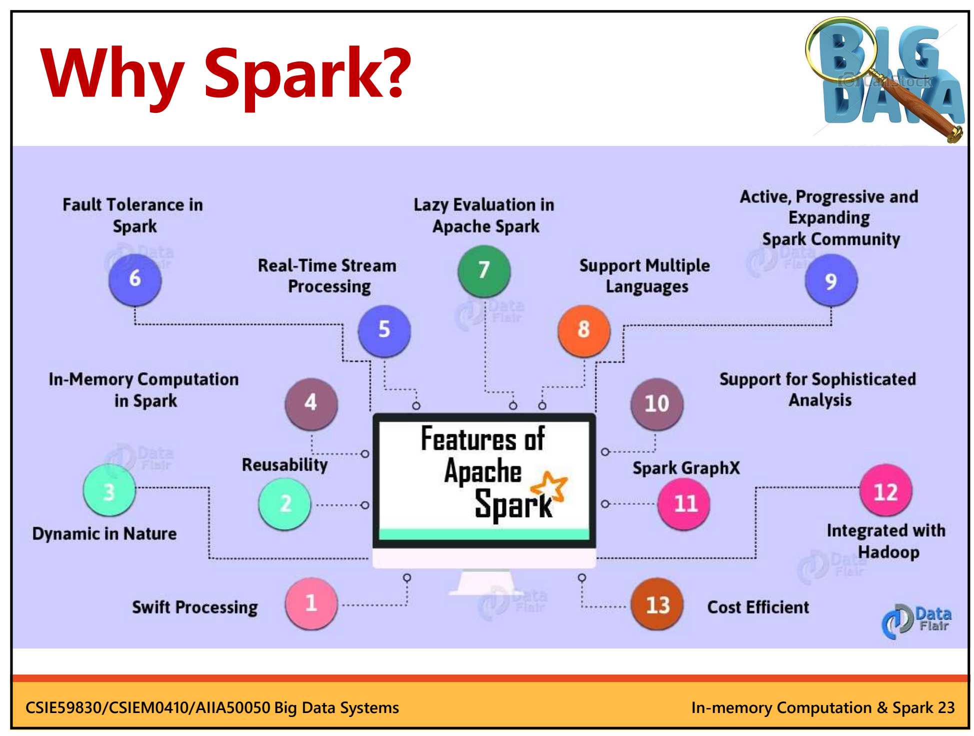
Algorithms
AMP
Machines
People

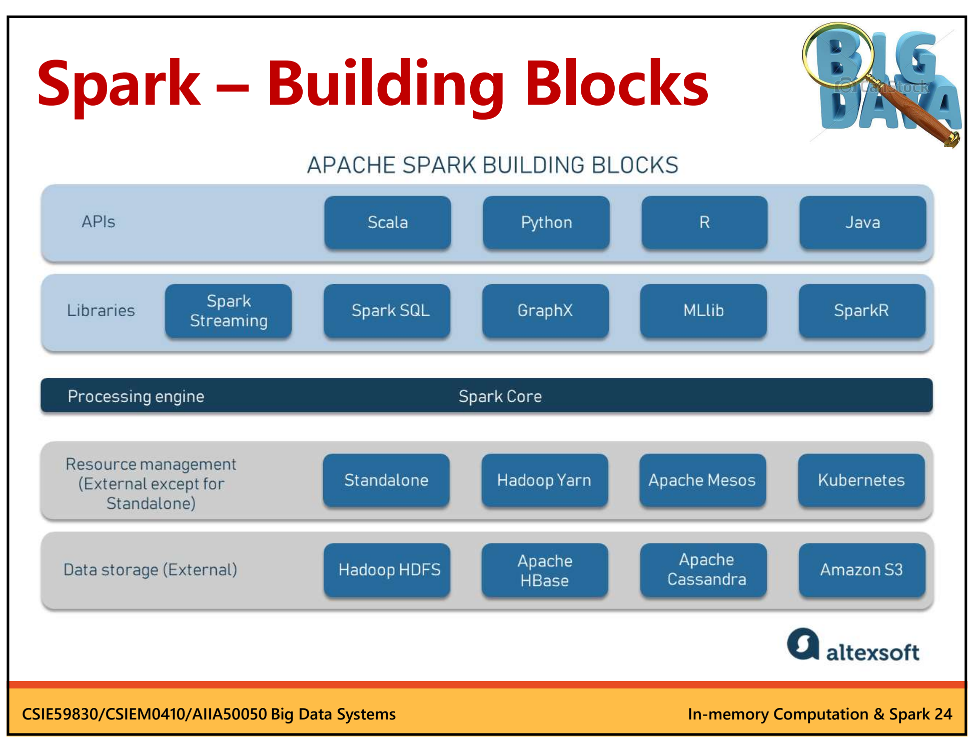

The Berkeley AMPLab
Funding & Sponsors
Grants & Foundations
DARPA
BERKELEY LAB
NSF
SIEBEL
The Thomas and Stacey Siebel Foundation
Founding Sponsors
amazon webservices
Google
IBM
SAP
Sponsors
Apple Inc.
BOSCH
CISCO
ERICSSON
facebook
FUJITSU
hp
HUAWEI
intel
Microsoft
Pivotal
SAMSUNG
StateFarm
vmware
Affiliates
ARIMO
cloudera
MITRE
neo4j
splunk>
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 19

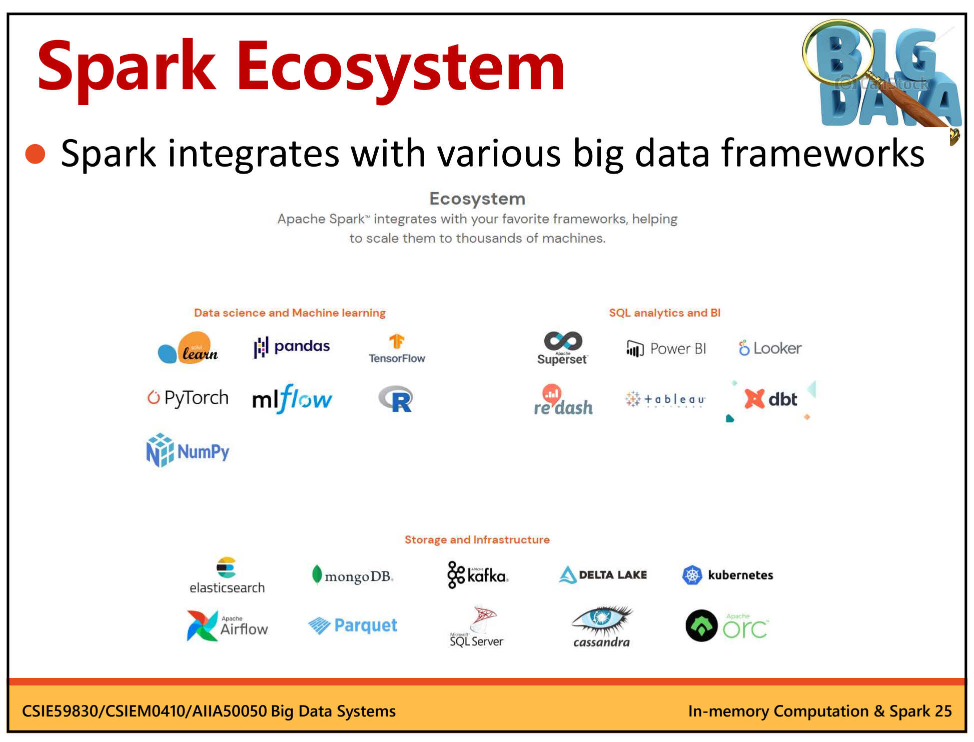

Berkeley Data Analytics Stack
Goals:
Batch
One stack to rule them all!
Interactive
Streaming
Easy to combine batch, streaming, and interactive computations
Easy to develop sophisticated algorithms
Compatible with existing open source ecosystem (Hadoop/HDFS)
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 20

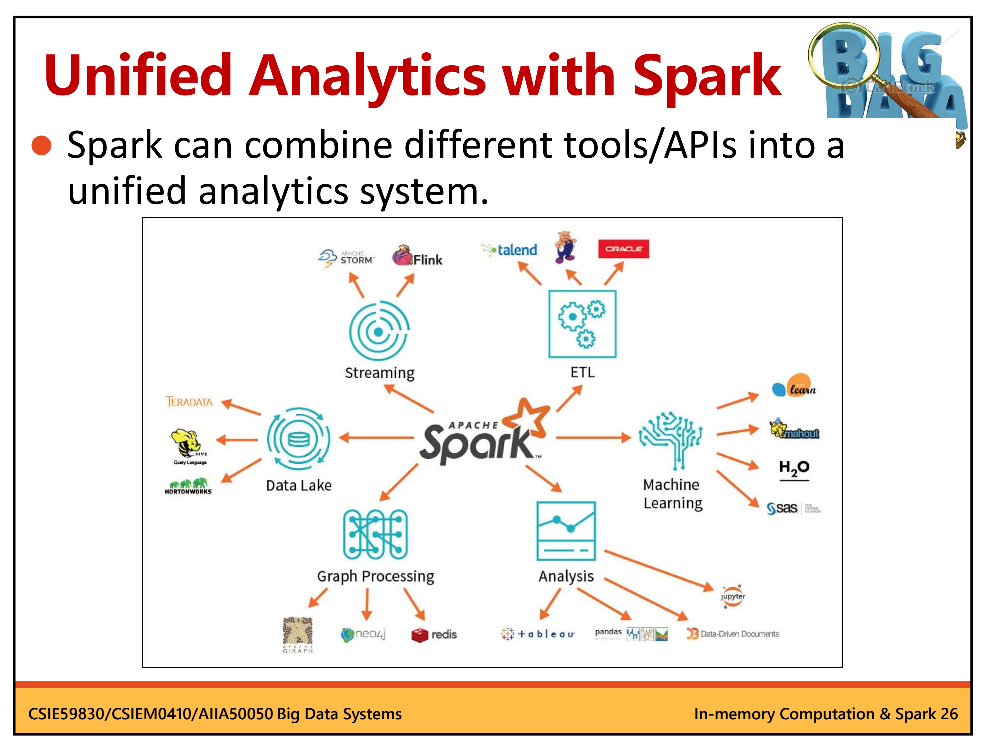

Berkeley Data Analytics Stack(BDAS)
In-house Apps
Cancer Genomics
Energy Debugging
Smart Buildings
Access and Interfaces
Spark Streaming
Sample Clean
G-OLA (Generalized Online Aggregation)
BlinkDB (Queries w bounded errors & time)
SparkSQL
SparkR
GraphX
Splash (Stochastic Learning)
MLBase
MLPipelines
MLlib
Velox (Predictive Services)
Processing Engine
Apache Spark (Core)
Storage
Succinct (Queries on Compressed Data)
(Distributed Storage) Alluxio (formerly Tachyon)
HDFS, S3, Ceph
Resource Virtualization
Apache Mesos
Hadoop Yarn
AMPLab Initiated
Spark Community
3rd Party
In Development
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 21


BDAS Main Components
Three main components:
Mesos: a distributed systems kernel and resource manager that provides efficient resource isolation and sharing across distributed applications, or frameworks
Alluxio(Tachyon): memory-centric distributed storage system enabling reliable data sharing at memory-speed across cluster frameworks
Spark: a cluster computing engine that aims to make specified computing(data analytics, ad-hoc) fast
All three continue to grow after AMPLab closed
MESOS
TACHYON
Spark
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 22


Why Spark?
Features of Apache Spark
1 Swift Processing
2 Reusability
3 Dynamic in Nature
4 In-Memory Computation in Spark
5 Real-Time Stream Processing
6 Fault Tolerance in Spark
7 Lazy Evaluation in Apache Spark
8 Support Multiple Languages
9 Active, Progressive and Expanding Spark Community
10 Support for Sophisticated Analysis
11 Spark GraphX
12 Integrated with Hadoop
13 Cost Efficient
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 23


Spark – Building Blocks
APACHE SPARK BUILDING BLOCKS
APIs
Scala
Python
R
Java
Libraries
Spark Streaming
Spark SQL
GraphX
MLlib
SparkR
Processing engine
Spark Core
Resource management (External except for Standalone)
Standalone
Hadoop Yarn
Apache Mesos
Kubernetes
Data storage (External)
Hadoop HDFS
Apache HBase
Apache Cassandra
Amazon S3
altexsoft
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 24


Spark Ecosystem
Spark integrates with various big data frameworks
Ecosystem
Apache Spark™ integrates with your favorite frameworks, helping
to scale them to thousands of machines.
Data science and Machine learning
learn
pandas
TensorFlow
PyTorch
mlflow
R
NumPy
SQL analytics and BI
Superset
Power BI
Looker
redash
tableau
dbt
Storage and Infrastructure
elasticsearch
mongoDB
kafka
DELTA LAKE
kubernetes
Airflow
Parquet
SQL Server
cassandra
orc
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 25


Unified Analytics with Spark
Spark can combine different tools/APIs into a unified analytics system.
STORM
Flink
talend
ORACLE
Streaming
ETL
TERADATA
Data Lake
APACHE Spark
Machine Learning
learn
mahout
H2O
sas
HORTONWORKS
Graph Processing
Analysis
jupyter
neo4j
redis
tableau
pandas
Data-Driven Documents
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 26

Spark Community

- The largest big data open source community, 2000+ contributors, used by 80% of the **Fortune 500**.
- 36,000+ stars to Spark project on **GitHub**.
- 80,000+ Spark related questions on **Stack Overflow**
- Various other Spark-focused online forums and groups.
- Active and growing community contributes to Spark's development, knowledge sharing, and troubleshooting assistance.

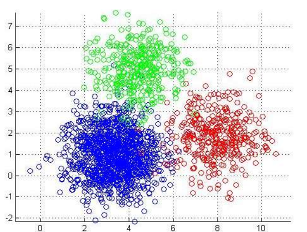
Key Advantages of Spark

- Speed - up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk
- Ease of Use - Write applications quickly in Java, Scala, Python, R, SQL.
- A Unified Engine - Combine SQL, streaming, and machine learning, graph & complex analytics.
- Runs Everywhere - Spark runs on Hadoop, Mesos, Kubernetes, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, MongoDB and S3.

Additional Goals of Spark

- Low latency (interactive) queries on historical data: enable faster decisions
  - E.g., identify why a site is slow and fix it
- Iterative Analytics: graph processing , machine learning, streaming, …
  - E.g., PageRank, MaxFlow, K-Means

Max flow in road network



Great but not Perfect

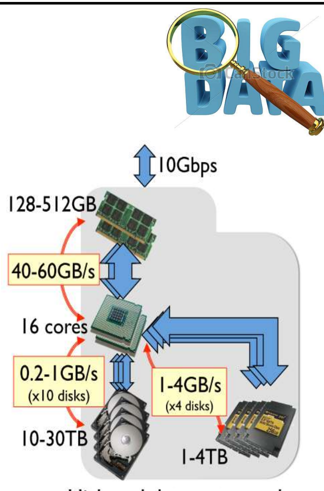
- High memory consumption which results in higher hardware and operational costs.
- Hard learning curve
- Complex performance parameters
  - Parameter settings significantly affect performance
  - Performance tuning/optimization not easy
- Generality can be a double-edged sword
  - Tools optimized for specific domain(s) (such as Presto, a distributed SQL engine for interactive analytic queries) are getting more and more popular.
- Limited support for real-time processing
  - Tools like Apache Flink or Apache Storm might be better for genuine real-time processing

The Working Set Idea in Spark

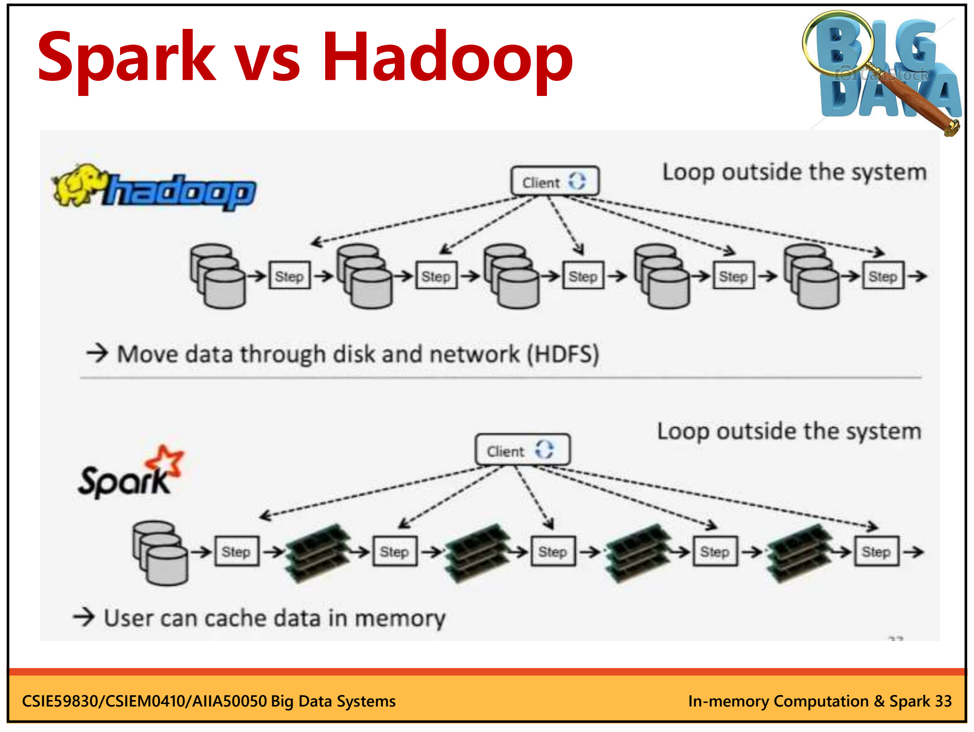
- The user should identify which datasets they want to access.
- Load datasets into memory, and use them multiple times.
- Keep newly created data in memory until explicitly told to store it.
- Master-Worker architecture: Master (driver) contains the main algorithmic logic, and the workers simply keep data in memory and apply functions to the distributed data.
- The master knows where data is located, so it can exploit locality.
- The driver is written in a functional programming language (Scala) which can be easily parallized.

Approach

- Aggregressive use of **Memory**
  - Memory transfer rate >> Disk transfer rate
  - Memory density (capacity) still grows with Moore's Law
    - RAM/SSD hybrid memories
  - Many datasets already fit into memory
    - The inputs of over 90% of jobs in Facebook, Yahoo!, and Bing clusters fit into memory
    - E.g., 1TB = 1 billion records @ 1 KB each



Spark vs Hadoop



Approach

- Trade between **result accuracy** and **response times**
- Why?
  - In-memory processing does not guarantee interactive query processing
  - E.g., ~10's sec just to scan 512 GB RAM!
  - Gap between memory capacity and transfer rate increasing
- Trade between response time, quality, and cost

Challenges for In-Memory Computation

- Provide distributed memory abstractions for clusters to support apps with working sets
- Retain the attractive properties of MapReduce:
  - Simple programming model
  - **Fault tolerance** (for crashes & stragglers)
  - Data locality
  - Scalability

Challenge: Fault Tolerance

- Existing in-memory storage systems have interfaces based on fine-grained updates
  - Read/Write to cells
  - E.g. Database, key-value store, distributed memory
- Requires replicating data or logs for fault tolerance
  - Very **inefficient & expensive** under **Big Data**

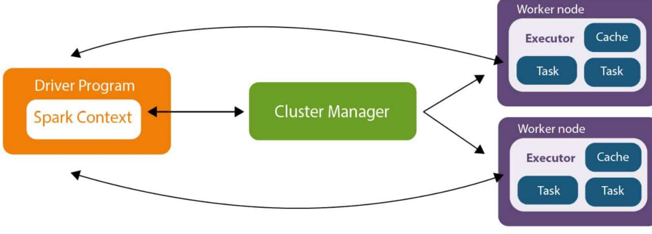
Challenge:
How to design a distributed memory abstraction that is both ***fault-tolerant*** and ***efficient***?
Solution: Augment data flow model with **RDD**

Spark Architecture

- The in-memory distributed computing is achieved through two key abstractions: RDD and DAG
- **Resilient Distributed Datasets (RDDs)** : immutable and distributed collections of obejcts that can be processed in parallel. (more details later)
- **Directed Acyclic Graph (DAG)** : an abstraction to model the transformations applied to RDDs.
- Programmers model data with RDDs and the application logic as a DAG. Then submit it to the Spark cluster for execution.

Spark: Runtime Architecture

- Driver: Spark application program
- SparkContext: heart of Spark app to establish a connection to the Spark Execution environment
- Executors: Compute and store distributed data
- Cluster Manager: Maintain a cluster of machines consisting of master and workers to run Spark applications.

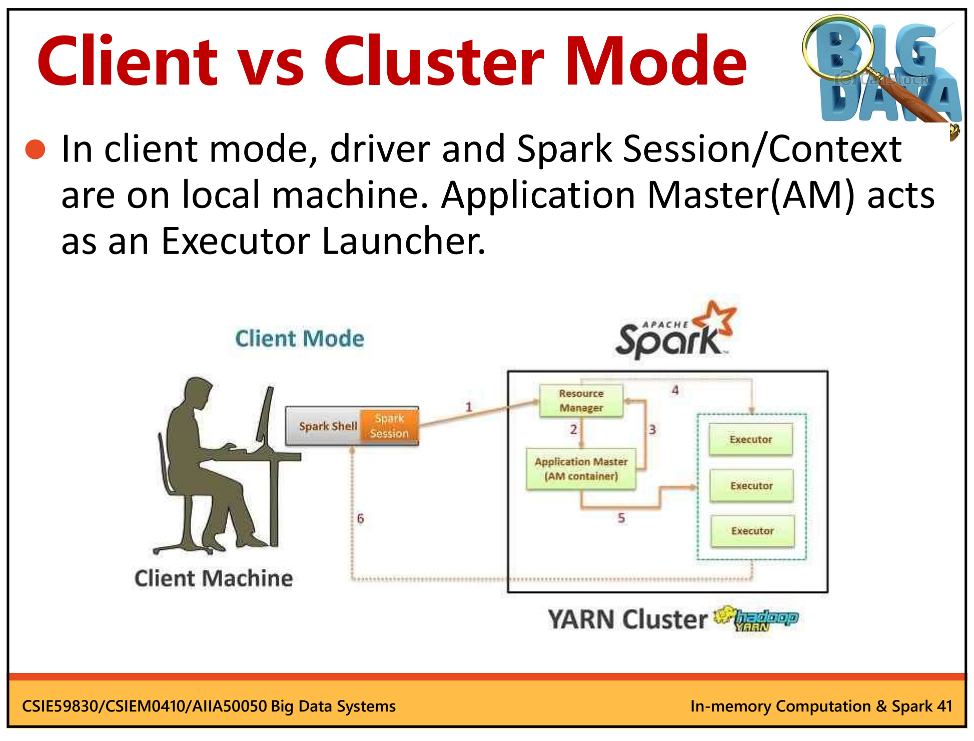


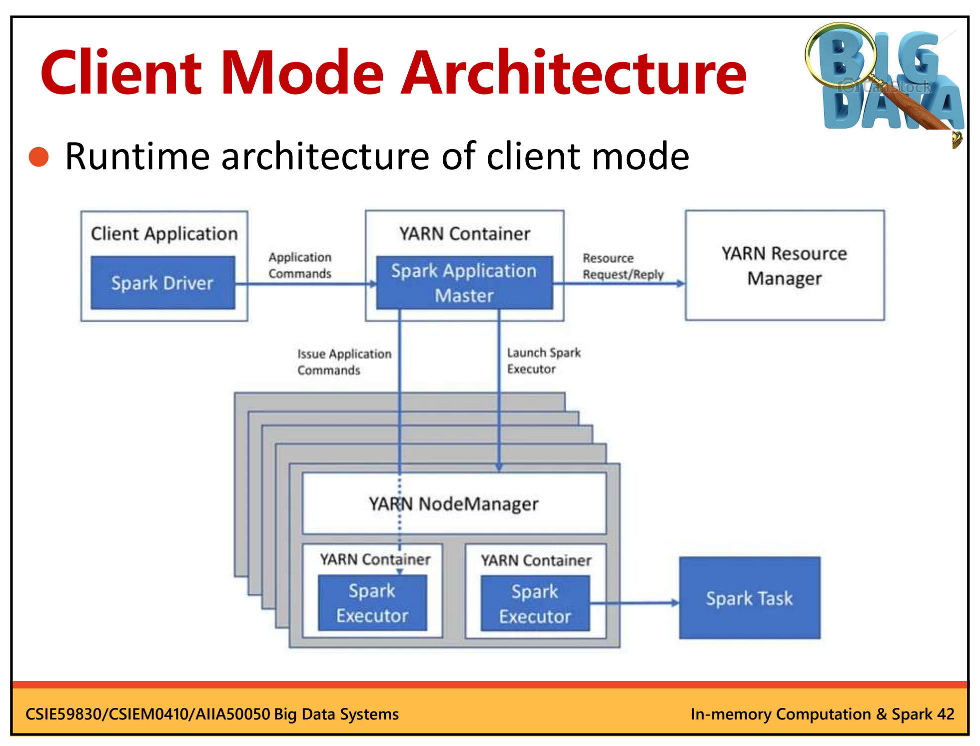
Modes of Execution

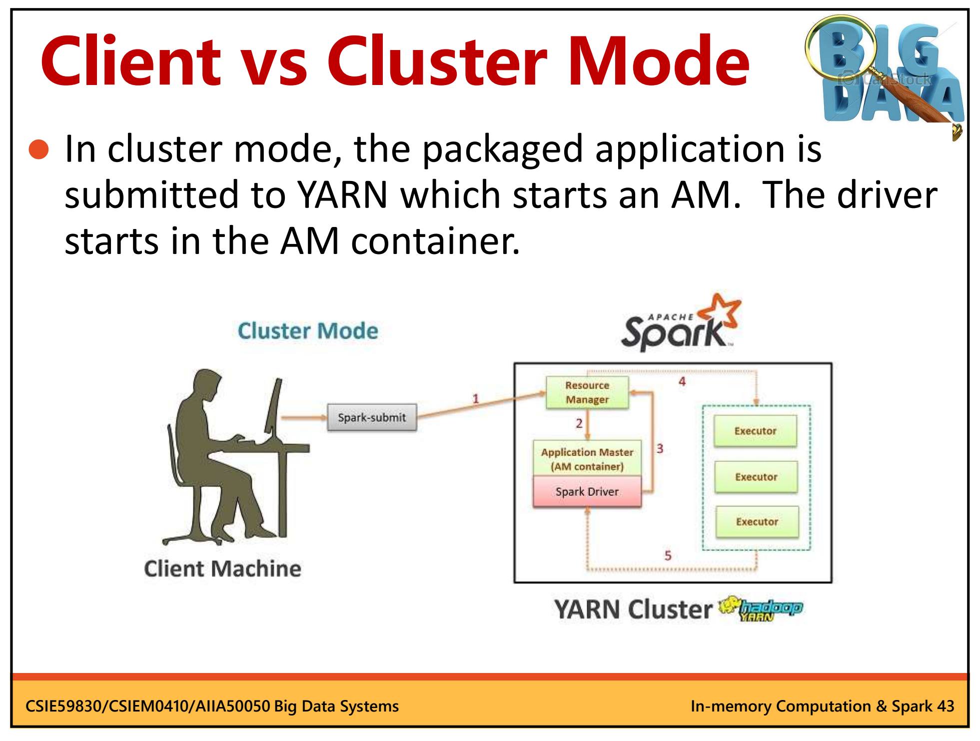
- Cluster Mode
  - Most common way of running Spark applications
  - A pre-compiled driver program (in Scalar, Python, JAR, R, …) is submitted to the cluster manager.
  - A driver process and executor processes are launched and maintained by the cluster manager on worker nodes.
- Client Mode
  - Almost the same as cluster mode except that the driver process remains on the client machine.
  - The cluster manage maintains the executor processes.

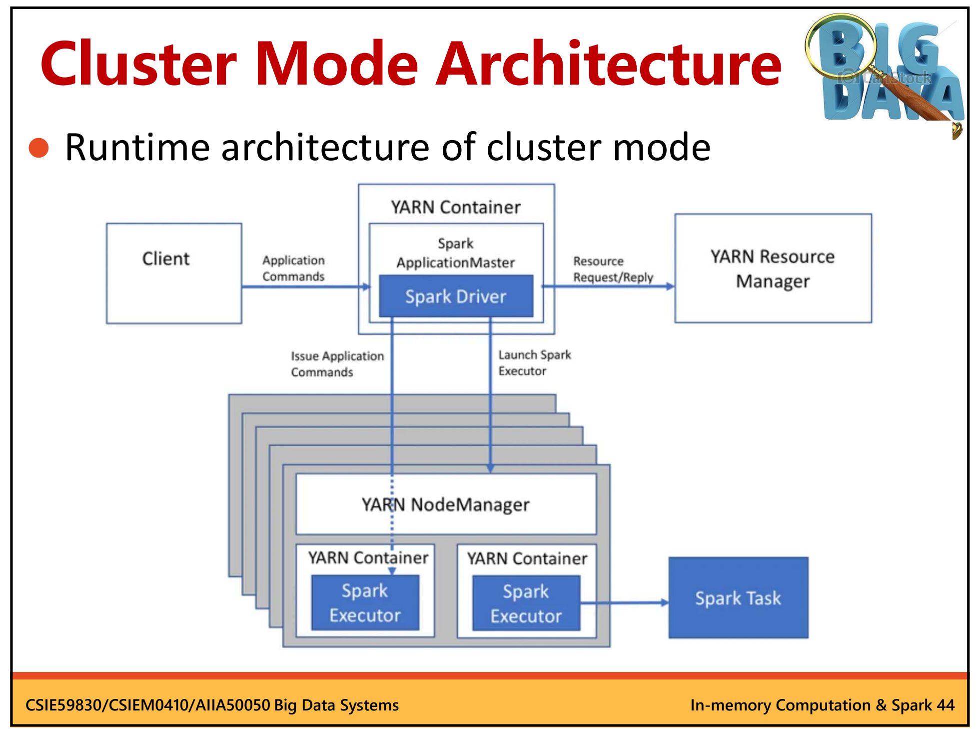
Modes of Execution

- Local Mode
  - The entire Spark application is run on a single machine
  - Still observers parallelism through threads
  - A common way for local development, testing and debugging
  - Not recommended for running production applications


Client vs Cluster Mode
In client mode, driver and Spark Session/Context are on local machine. Application Master(AM) acts as an Executor Launcher.
Client Mode
Spark
Spark Shell
Spark Session
Resource Manager
Application Master (AM container)
Executor
Executor
Executor
1
2
3
4
5
6
Client Machine
YARN Cluster
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 41


Client Mode Architecture
Runtime architecture of client mode
Client Application
Spark Driver
Application Commands
YARN Container
Spark Application Master
Resource Request/Reply
YARN Resource Manager
Issue Application Commands
Launch Spark Executor
YARN NodeManager
YARN Container
Spark Executor
YARN Container
Spark Executor
Spark Task
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 42


Client vs Cluster Mode
In cluster mode, the packaged application is submitted to YARN which starts an AM. The driver starts in the AM container.
Cluster Mode
Client Machine
Spark-submit
1
Resource Manager
2
Application Master (AM container)
Spark Driver
3
4
5
Executor
Executor
Executor
YARN Cluster
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 43


Cluster Mode Architecture
Runtime architecture of cluster mode
Client
Application Commands
YARN Container
Spark ApplicationMaster
Spark Driver
Resource Request/Reply
YARN Resource Manager
Issue Application Commands
Launch Spark Executor
YARN NodeManager
YARN Container
Spark Executor
YARN Container
Spark Executor
Spark Task
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 44

Resilient Distributed Datasets (RDD)

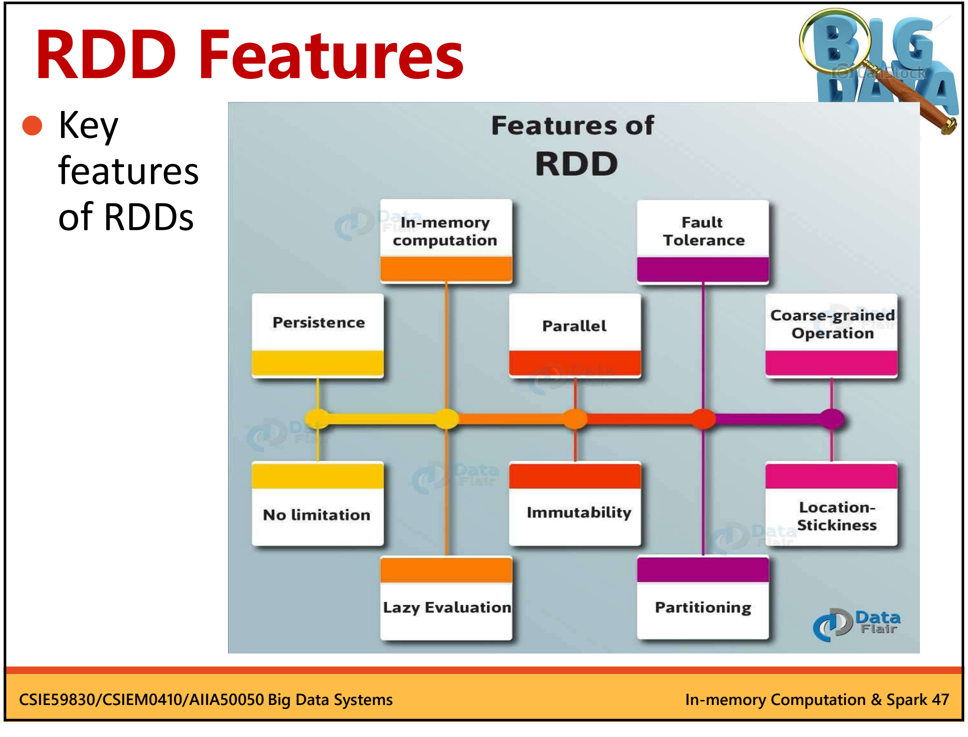
- **Distributed** data abstraction
  - for in-memory computation on large cluster
- **Read-only**, **partitioned** records
  - Only way to "write" is to create a new RDD
  - Partitions are scattered over the cluster
- Only **coarse-grained** operations are allowed
  - map, join, filter ...
  - operate on the whole dataset

Resilient Distributed Datasets (RDD)

- **Lazy Evaluation**
  - Two types of operations on RDD: Transformations & Actions
  - **Transformations**: create a new dataset from an existing one
    - do not compute right away but add this record to **Lineage**
    - only computed when an action requires a result
  - **Actions**: return a value after a computation on the dataset
    - It would execute all operation of the **Lineage**

RDD Features

- Key features of RDDs



DataFrames & SparkSQL

- A DataFrame is a distributed collection of rows under named columns. Better than RDDs for structured data.
- Similar to a SQL table in relational database, Python Pandas Dataframe or R's DataTables
  - Immutable once constructed
  - Lazy evaluation and track lineage
  - Enable distributed computations
- How to construct Dataframes
  - Read from files(e.g., CSV, JSON, Parquet, …)
  - Transforming an existing DFs(Spark or Pandas)
  - Parallelizing a python collection list
  - Apply transformations and actions
- Dataframes support a wide range of operations and transformations (e.g., filtering, aggregating, joingin, grouping)

DataFrame Examples

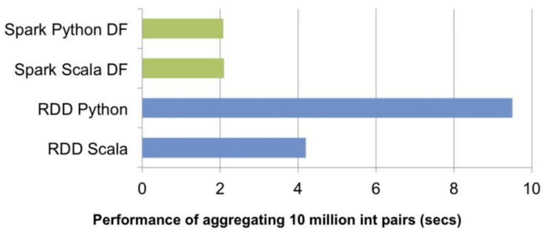
```
# import pyspark class Row from module sql
from pyspark.sql import *
# Create departments
dept1 = Row(id='123456', name='Computer Science')
dept2 = Row(id='789012', name='Pyhsics')

# Create Employees
Employee = Row("firstName", "lastName", "email", "salary")
emp1 = Employee('michael', 'armbrust', 'no-reply@Berkeley.edu', 100000)
emp2 = Employee('xiangrui', 'meng', 'no-reply@stanford.edu', 120000)
emp3 = Employee('matei', None, 'no-reply@waterloo.edu', 140000)
emp4 = Employee(None, 'wendell', 'no-reply@berkeley.edu', 160000)
```


DataFrame Examples

```
# Create the DepartmentWithEmployees instances from Departments and Employees
deptWithEmp1 = Row(department=dept1, employees=[emp1, emp2])
deptWithEmp2 = Row(department=dept2, employees=[emp3, emp4])
deptWithEmp3 = Row(department=dept3, employees=[emp1, emp4])
deptWithEmp4 = Row(department=dept4, employees=[emp2, emp3])

print(dept1)
print(emp2)
print(deptWithEmp1.employees[0].email)

# More on DataFrame later
```

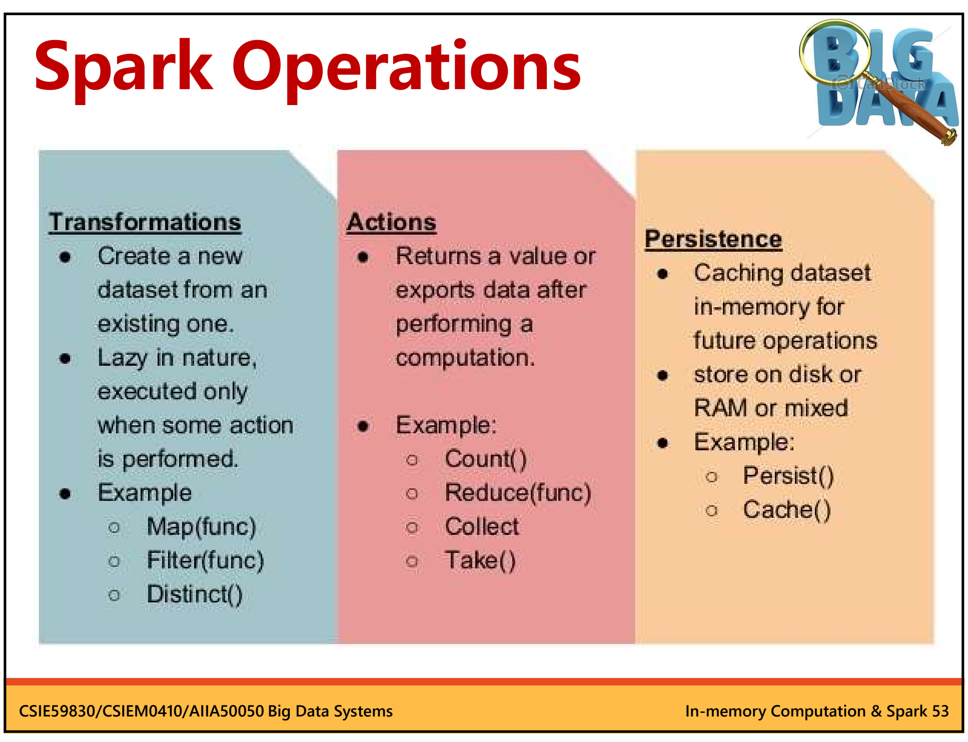
RDDs vs. DataFrames

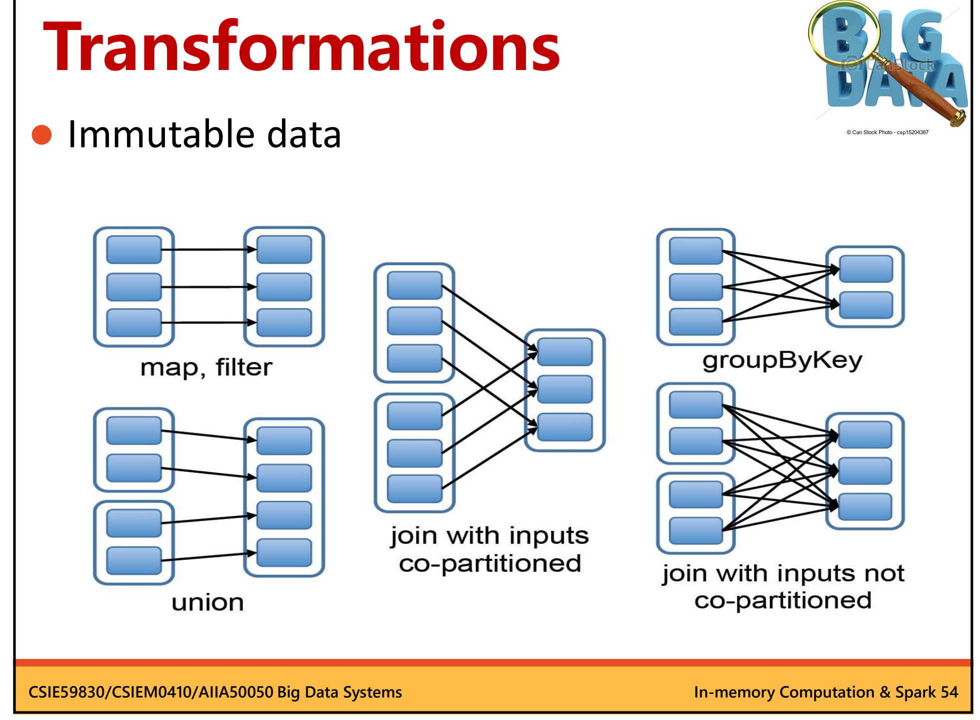
- RDDs provide a low level interface into Spark
- DataFrames have a schema
- DataFrames are cached and optimized by Spark
- DataFrames are built on top of the RDDs and the core Spark API

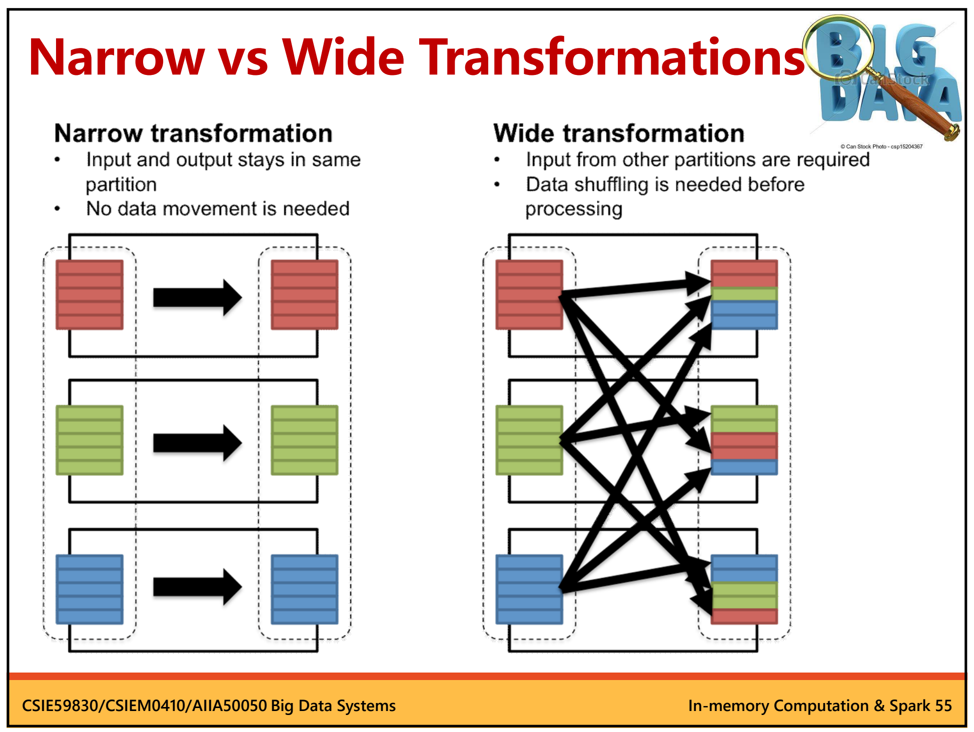


Dataset cs. DataFrames

- A Dataset is a distributed collection of data that provides the benefits of strong typing, compile-time type safety, and object-oriented programming.
- Essentially a strongly-typed version of DataFrame
- Each row is an object of a specific type
- Can be created from different sources (e.g., RDDs, DataFrames, structured data files, Hive tables, external databases, …)
- Compile-time type safety and OOP can help catch errors at compile time and improve code quality.

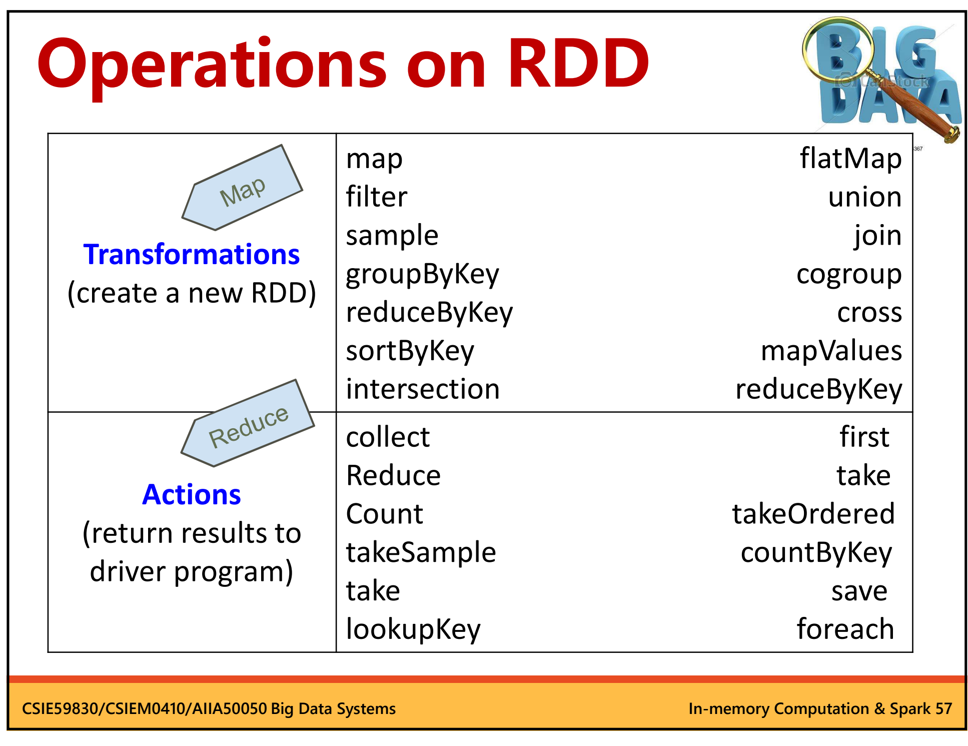

Spark Operations
Transformations
Create a new dataset from an existing one.
Lazy in nature, executed only when some action is performed.
Example
Map(func)
Filter(func)
Distinct()
Actions
Returns a value or exports data after performing a computation.
Example:
Count()
Reduce(func)
Collect
Take()
Persistence
Caching dataset in-memory for future operations
store on disk or RAM or mixed
Example:
Persist()
Cache()
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 53

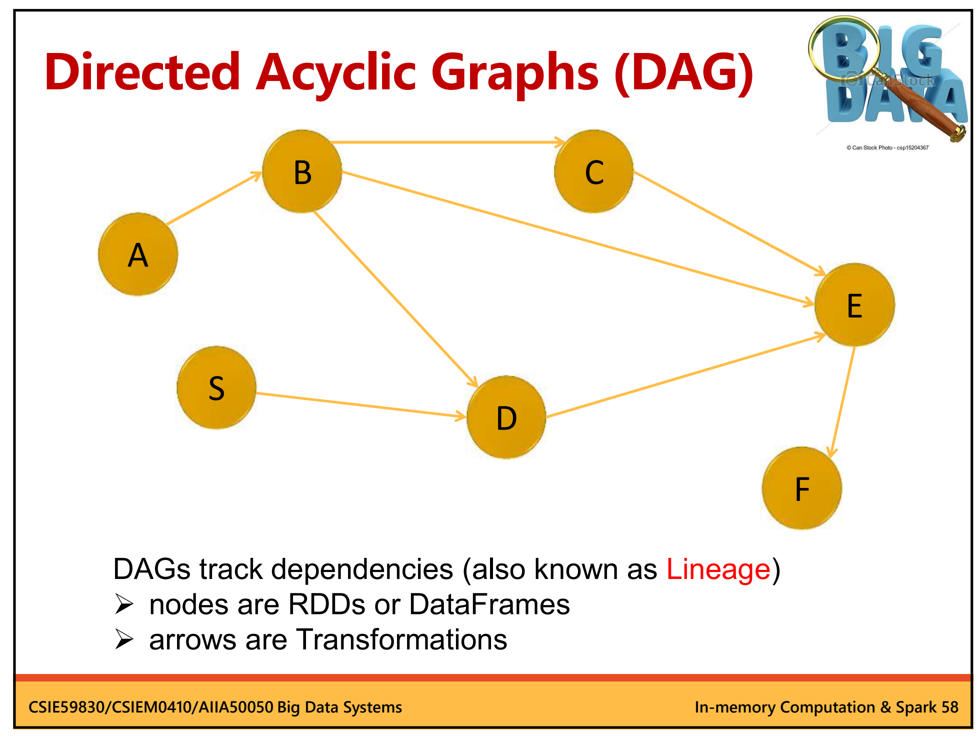

Transformations
Immutable data
map, filter
union
join with inputs co-partitioned
groupByKey
join with inputs not co-partitioned
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 54



Actions

- What is an action
  - Aggregates distributed data into values
  - Modeling a summary stage of the workflow
  - Triggers the execution of the DAG
  - Returns the results to the driver
  - Writes the data to HDFS or to a file


Operations on RDD
Map
Transformations
(create a new RDD)
map
filter
sample
groupByKey
reduceByKey
sortByKey
intersection
flatMap
union
join
cogroup
cross
mapValues
reduceByKey
Reduce
Actions
(return results to driver program)
collect
Reduce
Count
takeSample
take
lookupKey
first
take
takeOrdered
countByKey
save
foreach
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 57

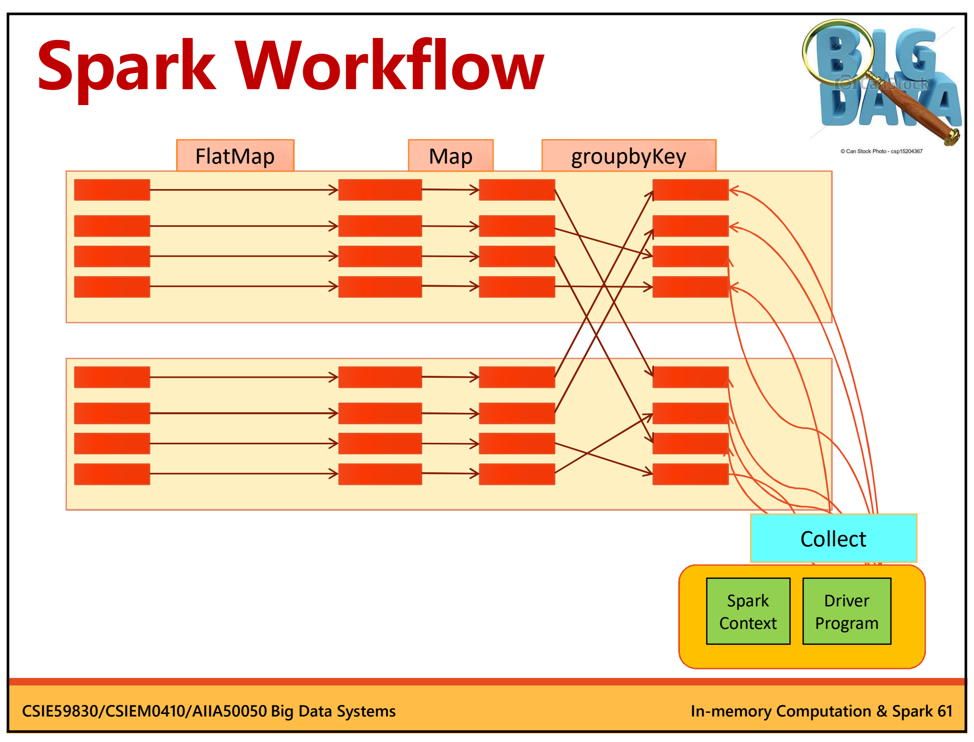

Directed Acyclic Graphs (DAG)
A
B
C
S
D
E
F
DAGs track dependencies (also known as Lineage)
nodes are RDDs or DataFrames
arrows are Transformations
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 58

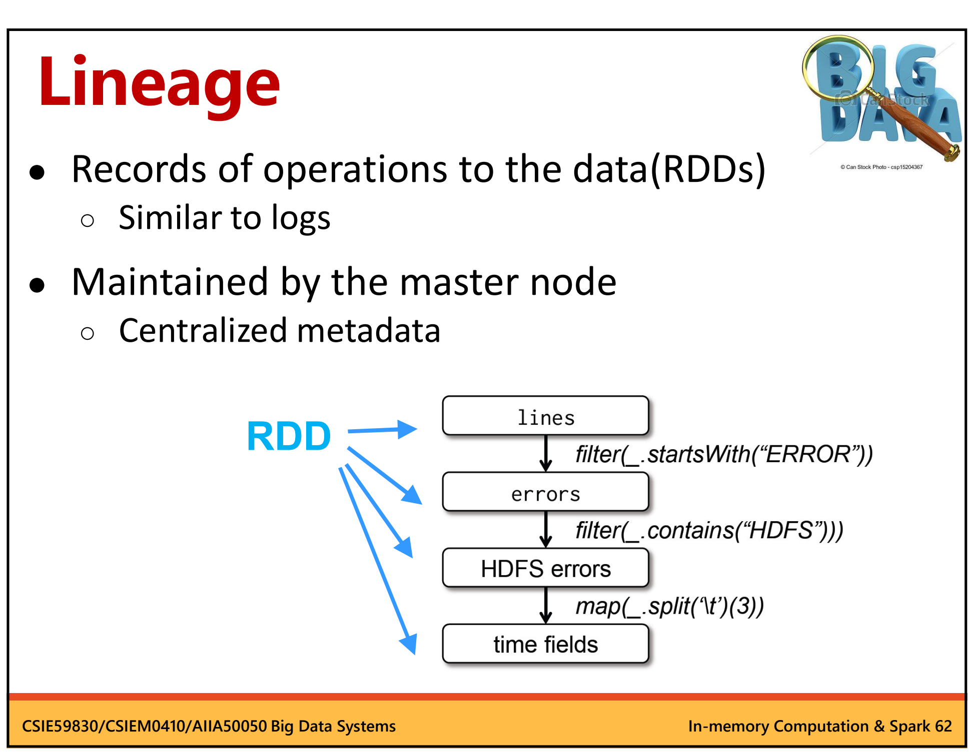
Generality of RDDs

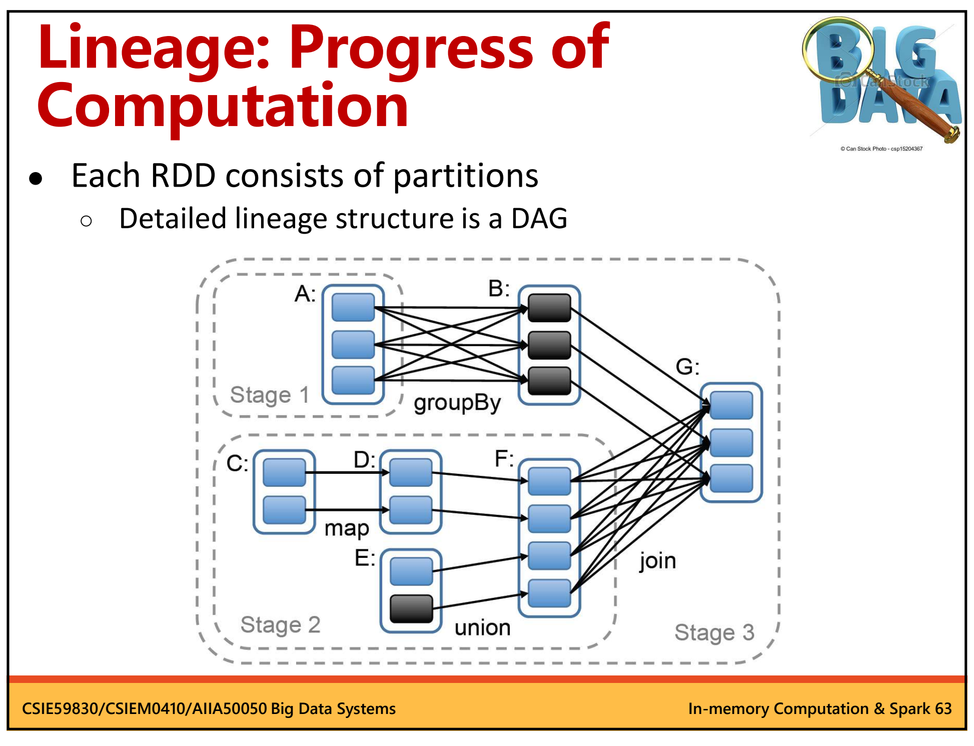
- Spark's combination of data flow with RDDs/DataFrames/Datasets unifies many proposed cluster programming models
  - *General data flow models:* MapReduce, Dryad, SQL
  - *Specialized models for stateful apps:* Pregel (BSP), HaLoop (iterative MR), Continuous Bulk Processing
- Instead of specialized APIs for one type of app, give user first-class control of distributed datasets

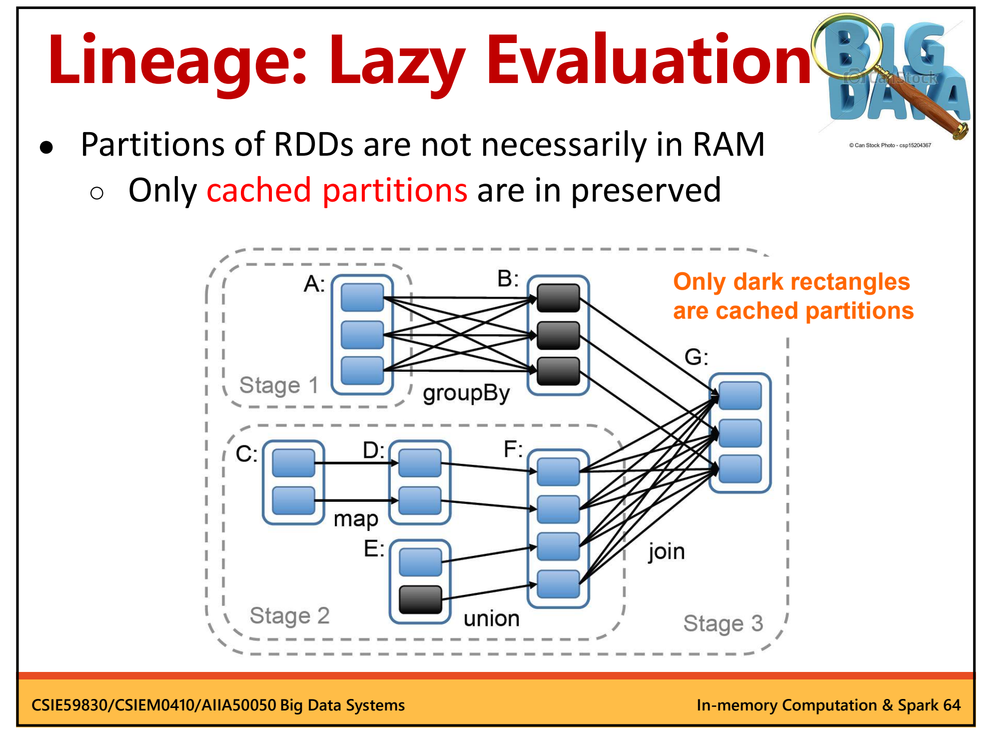
RDDs vs. DSM

Aspect	RDDs	Distr. Shared Mem.
Reads	Bulk or fine-grained	Fine-grained
Writes	Bulk transformations	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)


Spark Workflow
FlatMap
Map
groupbyKey
Collect
Spark Context
Driver Program
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 61

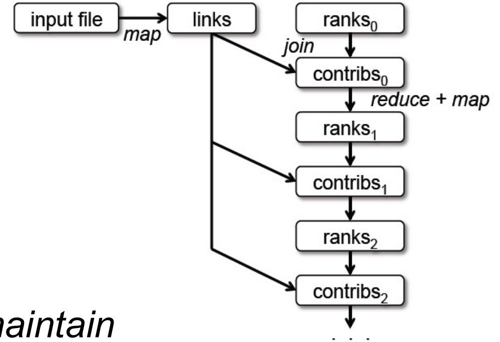

Lineage
Records of operations to the data(RDDs)
Similar to logs
Maintained by the master node
Centralized metadata
RDD
lines
filter(_.startsWith("ERROR"))
errors
filter(_.contains("HDFS")))
HDFS errors
map(_.split('\t')(3))
time fields
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 62


Lineage: Progress of Computation
Each RDD consists of partitions
Detailed lineage structure is a DAG
A:
B:
G:
Stage 1
groupBy
C:
D:
F:
map
E:
join
Stage 2
union
Stage 3
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 63


Lineage: Lazy Evaluation
Partitions of RDDs are not necessarily in RAM
Only cached partitions are in preserved
A:
B:
Only dark rectangles are cached partitions
G:
Stage 1
groupBy
C:
D:
F:
map
E:
join
Stage 2
union
Stage 3
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 64

Fault Tolerance using Lineage

- RDD can only be created (written) from
  - Static Storage
  - Other RDDs
- Only coarse-grained opeations



➡ *Less information to maintain*

➡ *Lost partitions can be re-computed efficiently*

Core Concepts

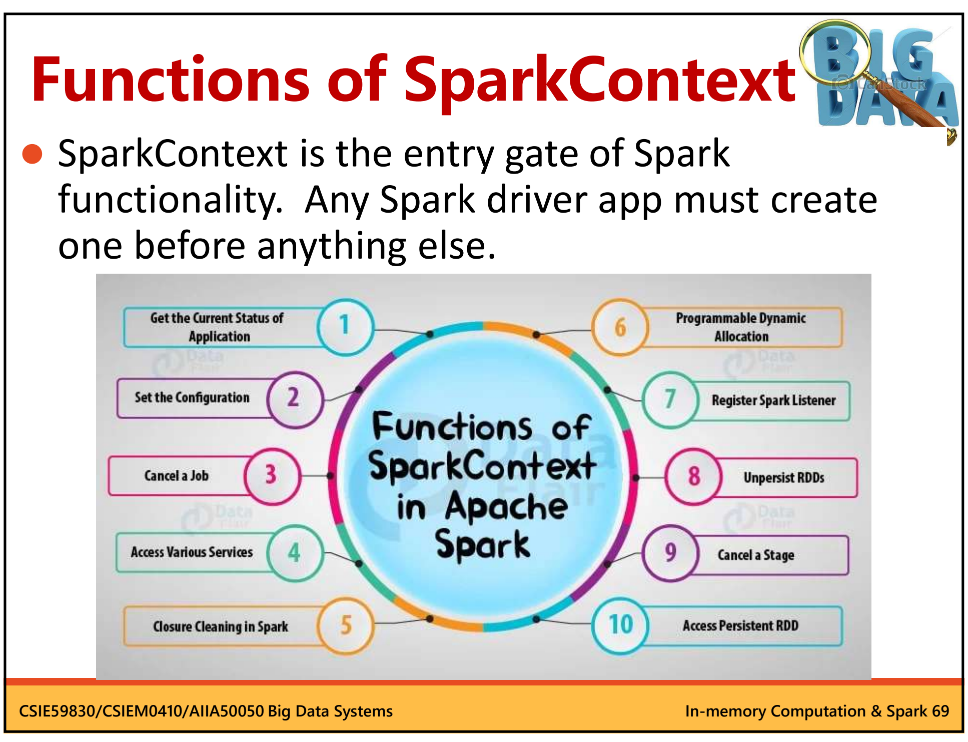
- Job: A piece of code which reads some input data from HDFS or local, performs some computation and writes some output data.
- Stages: Jobs are divided into stages. Stages are divided based on computational boundaries. All computations (operators) cannot be updated in a single stage. It happens over many stages.
- Tasks: Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- DAG: DAG stands for Directed Acyclic Graph, which represents the flow and relationships of operators.
- Executor: The process responsible for executing a task.
- Master: The machine on which the Driver program runs
- Slave: The machine on which the Executor program runs

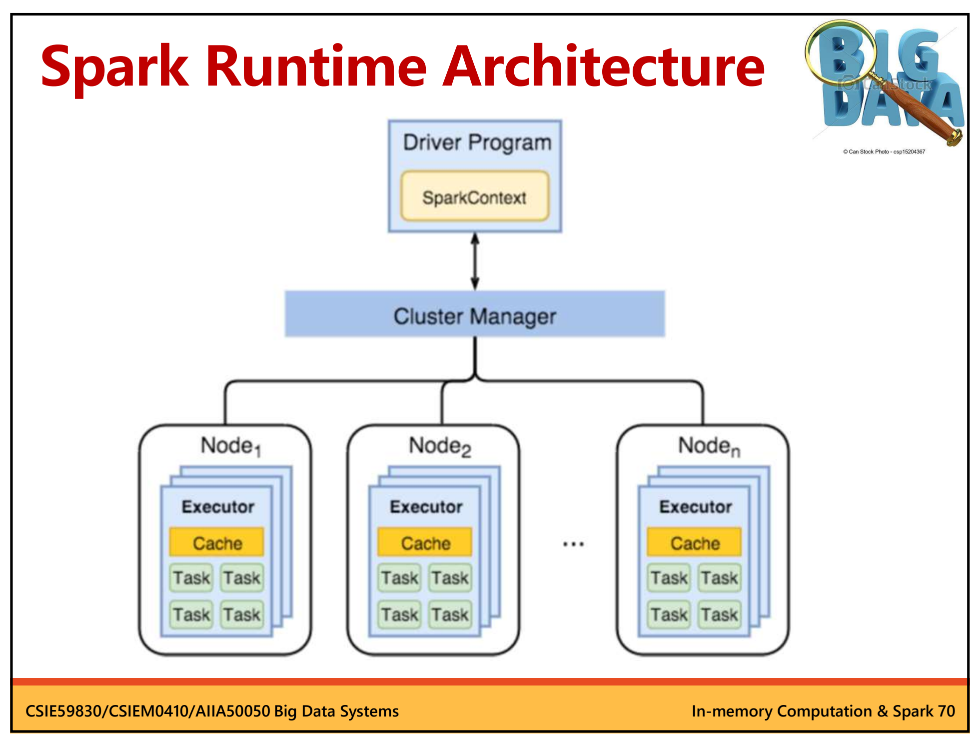
Architectural Components

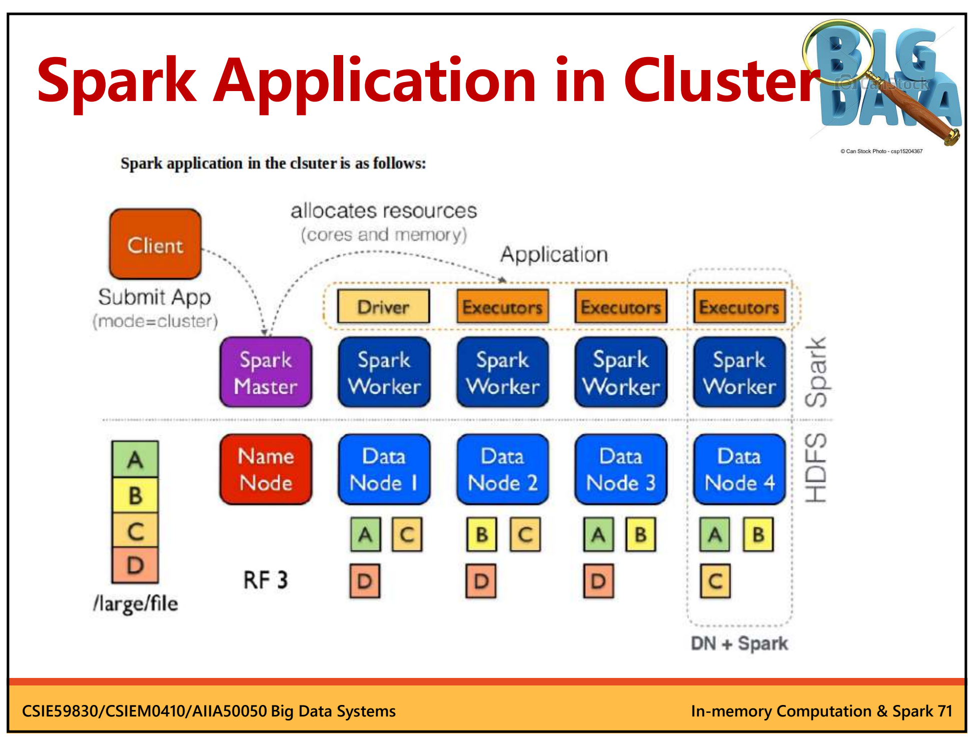
- Spark cluster: a collection of machines connected to each other running Spark.
- Spark Master: the node that schedules and monitors the jobs assigned to the Workers.
- Spark Worker: receive commands from the Master, launch executors, execute the assigned tasks.
- Spark Executor: an executor inside a worker which executes the assigned tasks.
- Spark Driver: the coordinator between master and workers, distribute tasks to executors.

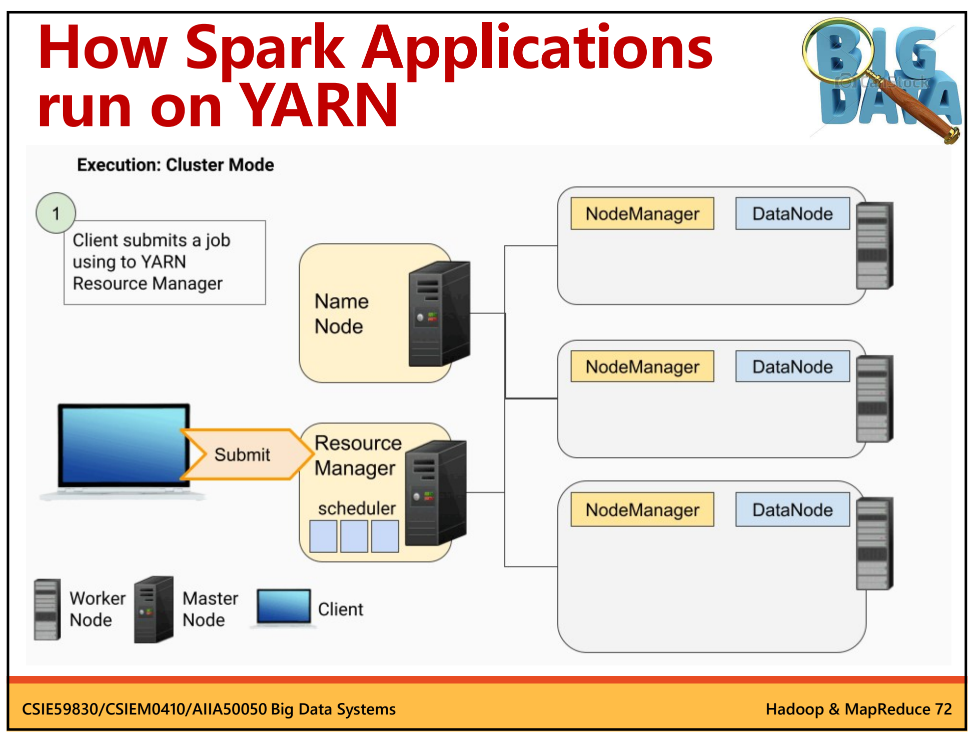
SparkContext (SC)

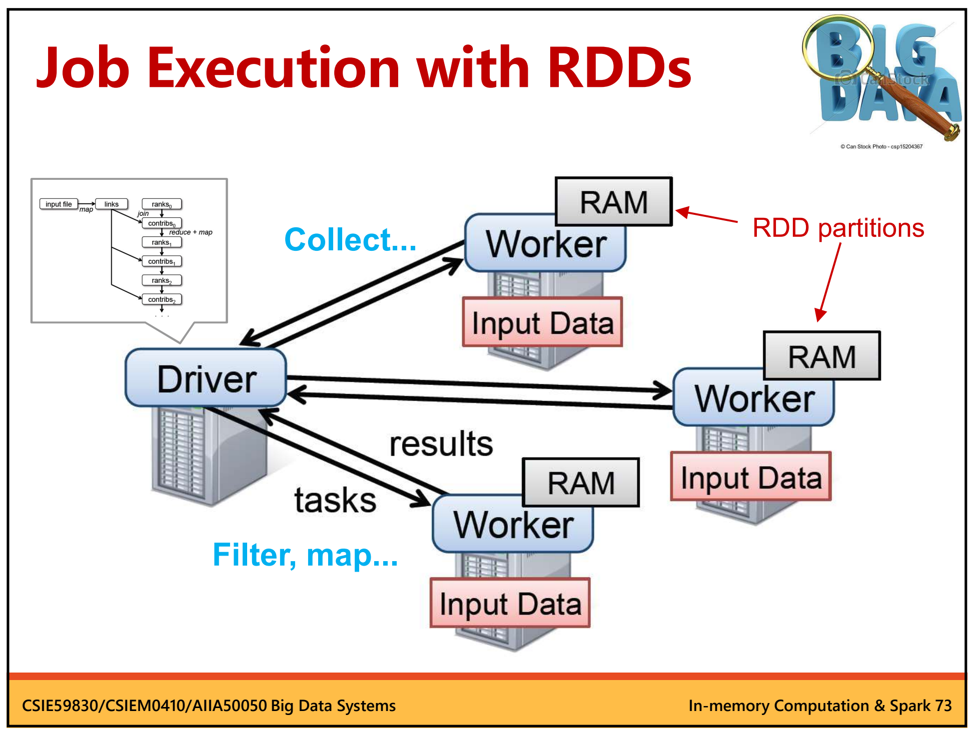
- The master of any Spark application
- Enable Spark Driver to access the cluster through resource manager(RM)
- RM can be any of YARN, MESOS, Spark's cluster manager, etc.
- Provide functions for getting/setting configuration, creating objects, scheduling/canceling jobs, etc.
- In Spark shell, it is automatically created for you.
- In your app, you need to set it explicitly.


Functions of SparkContext
SparkContext is the entry gate of Spark functionality. Any Spark driver app must create one before anything else.
Functions of SparkContext in Apache Spark
Get the Current Status of Application
Set the Configuration
Cancel a Job
Access Various Services
Closure Cleaning in Spark
Programmable Dynamic Allocation
Register Spark Listener
Unpersist RDDs
Cancel a Stage
Access Persistent RDD
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 69


Spark Runtime Architecture
Driver Program
SparkContext
Cluster Manager
Node1
Node2
Noden
Executor
Cache
Task
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 70

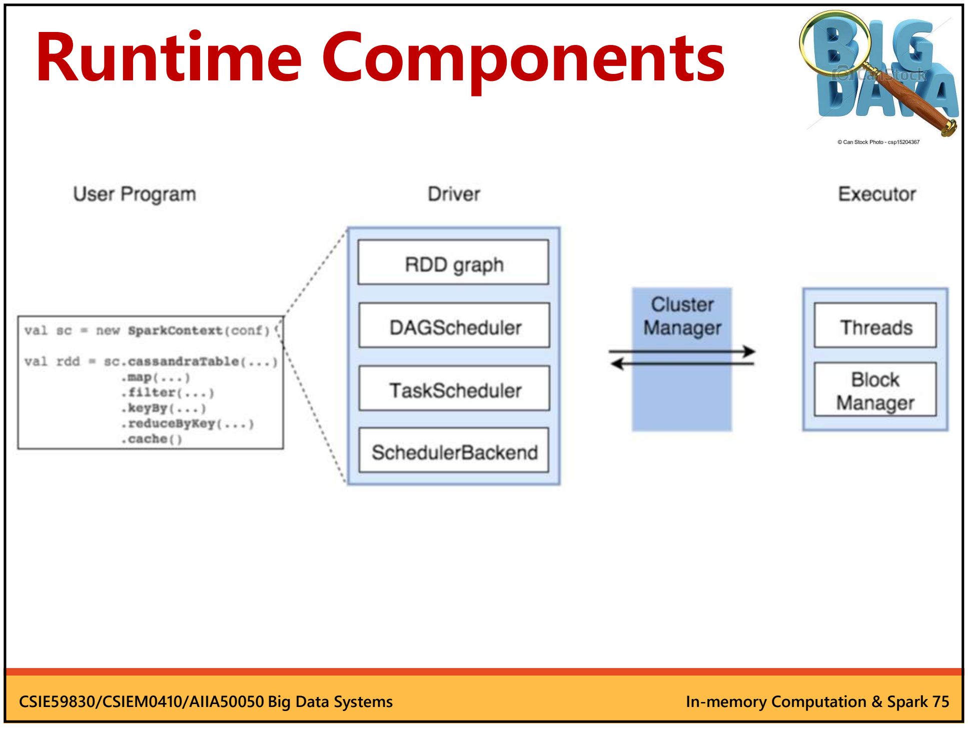

Spark Application in Cluster
Spark application in the clsuter is as follows:
Client
Submit App
(mode=cluster)
allocates resources
(cores and memory)
Application
Driver
Executors
Executors
Executors
Spark Master
Spark Worker
Spark Worker
Spark Worker
Spark Worker
Spark
HDFS
Name Node
Data Node 1
Data Node 2
Data Node 3
Data Node 4
A
B
C
D
/large/file
RF 3
DN + Spark
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 71


How Spark Applications run on YARN
Execution: Cluster Mode
1
Client submits a job using to YARN Resource Manager
Name Node
Submit
Resource Manager
scheduler
NodeManager
DataNode
NodeManager
DataNode
NodeManager
DataNode
Worker Node
Master Node
Client
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
Hadoop & MapReduce 72



Runtime Components

- **Spark Driver**:
  - 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/disk or off-heap
  - interact with storage systems
- **Cluster Manager**:
  - Spark Standalone
  - Mesos
  - YARN
  - Kubernetes



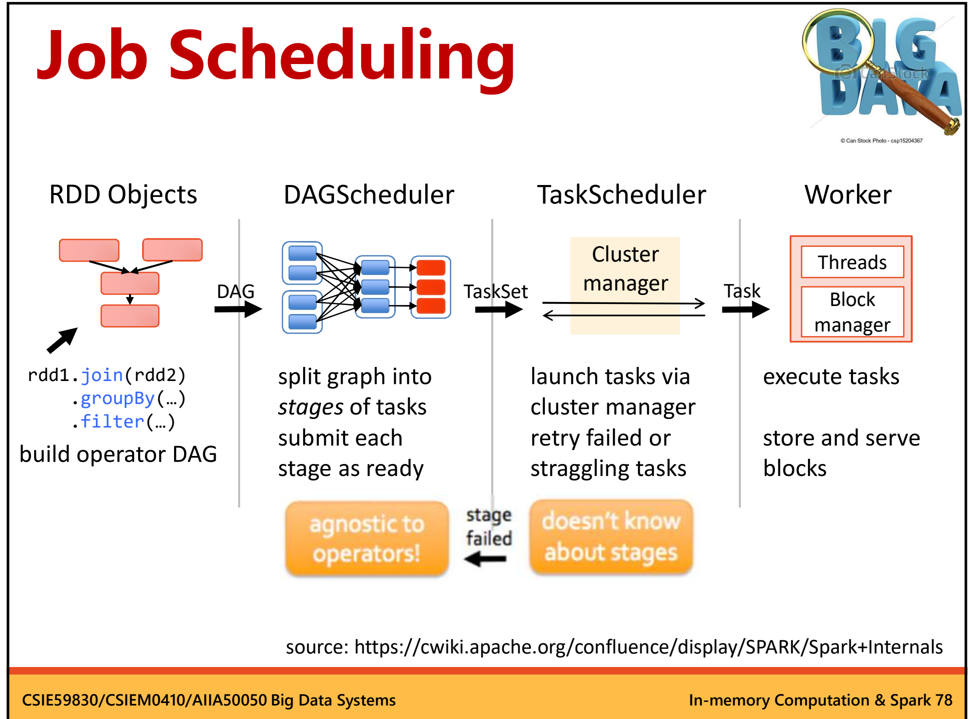
Component Details

- **SparkContext**
  - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster
- **DAGScheduler**
  - computes a DAG of stages for each job and submits them to TaskScheduler, determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs
- **TaskScheduler**
  - sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers

Component Details

- **SchedulerBackend**
  - backend interface for scheduling systems that allows plugging in different implementations (Mesos, YARN, Standalone, local)
- **BlockManager**
  - provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

Job Scheduling



source: https://cwiki.apache.org/confluence/display/SPARK/Spark+Internals

Other Issue: Dealing with Stragglers

- Speculative Execution
  - Observe the process of the tasks of a job
  - Launch duplicates of those tasks that are slower
  - It then becomes a race between the original and the speculative copies

Other Issue: Dependency

Narrow Dependency:

- 1/N-to-1

map, filter

join with inputs co-partitioned

union

- Execution can be pipelined
- Faster to recompute

Wide Dependency:

- N-to-N

groupByKey

join with inputs not co-partitioned

Other Issue: Memory Management

- **Problem**:
  - Some RDDs (partitions) are too large to store in some worker's memory
  - These RDDs are costly to re-compute
- **Solution**: Use hard disks
  - Swap RDDs out under LRU eviction policy
  - Users can set persistence priority to RDDs

Other Issue: Optimization

- Persistence
  - Users can indicate which RDDs they will reuse => save them in memory rather than recomputed
- Partitioning
  - Utilize data locality to optimize transformations
  - Similar to the *partition function* in MapReduce when mapping
  - e.g. partition URLs by domain name

Spark in the Real World (I)

- **Uber** – the online taxi company gathers terabytes of event data from its mobile users every day.
  - By using Kafka, Spark Streaming, and HDFS, to build a continuous ETL(Extract-Transform-Load) pipeline
  - Convert raw unstructured event data into structured data as it is collected
  - Uses it further for more complex analytics and optimization of operations
- **Pinterest** (social network for sharing pinboards) – Uses a Spark ETL pipeline
  - Leverages Spark Streaming to gain immediate insight into how users all over the world are engaging with Pins—in real time.
  - Can make more relevant recommendations as people navigate the site
  - Recommends related Pins
  - Determine which products to buy, or destinations to visit

Spark in the Real World (II)

Here are Few other Real World Use Cases:

- **Conviva** (video AI platform)– 4 million video feeds per month
  - This streaming video company is second only to YouTube.
  - Uses Spark to reduce customer churn by optimizing video streams and managing live video traffic
  - Maintains a consistently smooth, high quality viewing experience.
- **Capital One** (online account service) – is using Spark and data science algorithms to understand customers in a better way.
  - Developing next generation of financial products and services
  - Find attributes and patterns of increased probability for fraud
- **Netflix** – leveraging Spark for insights of user viewing habits and then recommends movies to them.
  - User data is also used for content creation

Application Development with Spark

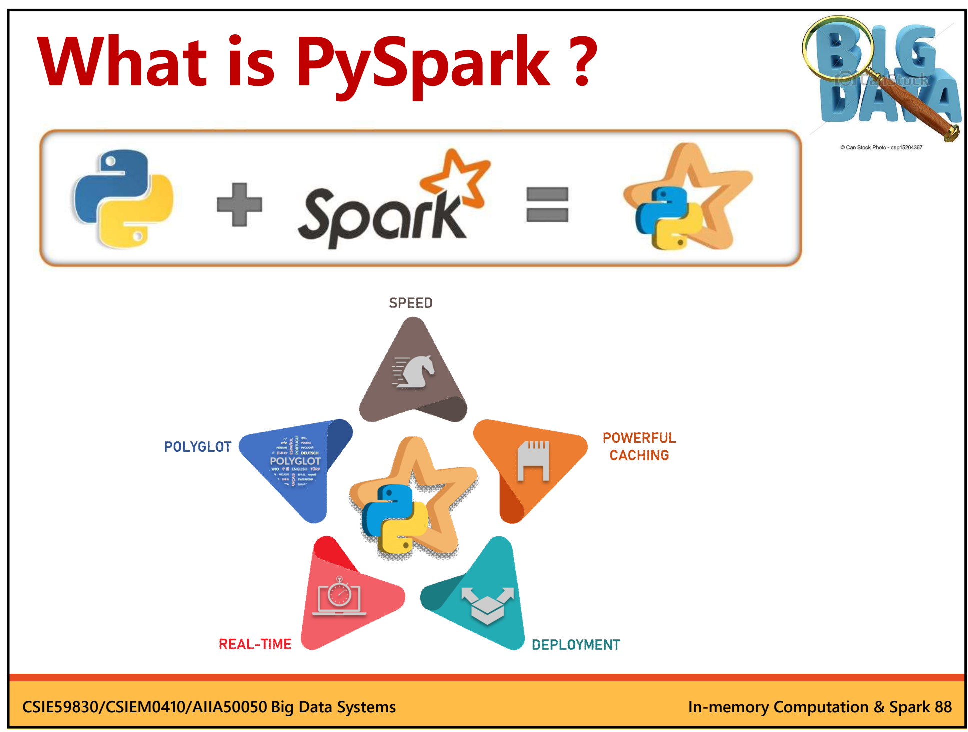
Spark Application Development

- Spark is polyglot (works with multiple programming languages) with APIs to Scala, Java, Python, R, and SQL.
- Most Spark applications (> 70%) were developed with Scala which is the primary language.
- With the availability of PySpark and continuous improvement over the years, Python is getting popular among all types of programmers.
- In general, Scala is faster than Python but harder to learn and master.

Functional Programming and Stateless with Scala

- Using **Scala,** a functional programming language which runs on JVM.
- Recall from the MapReduce session: **stateless properties** of functional programming language is good for parallelization.
- Scala is a natural choice to work with RDDs.

What is PySpark ?



PySpark(Spark with Python)

- PySpark is the Python API for Apache Spark.
- Provides PySpark shell for interactive analytics.
- PySpark supports all of Spark's features

Spark SQL and DataFrames	Pandas API on Spark	Structured Streaming	Machine Learning MLlib
Spark Core and RDDs			

Advantages of PySpark

- Python is far better than Scala in available libraries.
- For Scala, there are no good visualization tool which is not a problem with Python.
- The learning curve is less in Python.
- Python is easy to use.
- PySpark works well with RDDs.
- Inbuild-optimization when using DataFrames.
- Supports ANSI SQL.

Python or Scala ?

- Apache **Spark** is written in **Scala** as it is more scalable on JVM.
- If you know Scala, then use Scala!
- If both are new to you, Python is easier to learn.
- Python is easy to use: Code readability, maintainability and familiarity is far beter with Python
- Python comes with great libraries !!
- Scala is faster than Python but if your Python code just calls Spark libraries, the performance differences is minimal.
- **Reminder**: New feature added in Spark API will be available in Scala first!

Python vs Scala

- Detailed comparison of Python vs Scala

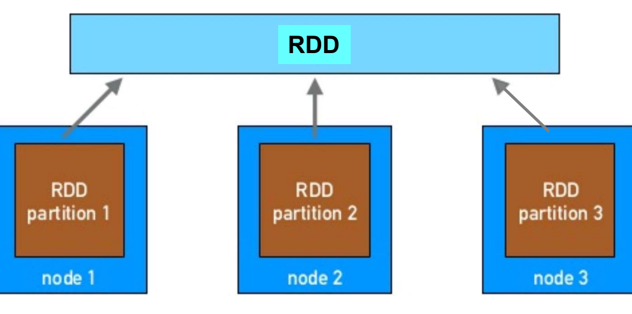
Sr.	Python	Scala
1.	Python is an interpreted, dynamic programming language.	Scala is a statically typed language.
2.	Python is Object Oriented Programming language.	In Scala, we need to specify the type of variable and objects.
3.	Python is easy to learn and use.	Scala is slightly difficult to learn than Python.
4.	Python is slower than Scala because it is an interpreted language.	Scala is 10 times faster than Python.
5.	Python is an Open-Source language and has a huge community to make it better.	Scala also has an excellent community but lesser than Python.
6.	Python contains a vast number of libraries and the perfect tool for data science and machine learning.	Scala has no such tool.

Programming Model

- RDDs/DataFrames/Datasets
  - Constructed from HDFS files, "parallelized" collections
  - Can be transformed with map and filter
  - *Can be cached across parallel operations*
- Parallel operations
  - Foreach, reduce, collect, …
- Shared variables
  - Accumulators (add-only)
  - Broadcast variables (read-only)

Resilient Distributed Datasets (RDDs)

- RDDs are the core data structure in Spark
- Distributed, resilient, immutable, can store unstructured and structured data, lazy evaluated



Resilient Distributed Datasets (RDDs)

- In Spark, there are **four** ways to construct RDDs:
  - From files in a shared filesystem, such as HDFS.
  - An existing collection (e.g., an array) of objects
  - Transforming existing RDDs
  - Changing the persistence of existing RDDs, RDDs by default are *lazy* and *ephemeral*(短暫的)
    - cache: hint that the data need to be cache after the first time
    - save: save the dataset to distributed file system (HDFS)

Parallel Operations

- Several parallel operations can be performed on RDD
  - **reduce**: combines dataset elements using an associative function to produce a result at the driver program.
  - **collect**: sends all elements of the dataset to the driver program.
  - **foreach**: Passes each element through a user provided function.

Example: Log Error Counting

- To count the lines containing errors in a large log file stored in HDFS (Python)

```
file = sc.textFile("hdfs://master:9000/.../tst.dat")
errs = file.filter(lambda x: "ERROR" in x))
ones = errs.map(lambda x : ("errorline", 1))
count = ones.reduceByKey(lambda x,y: x + y)
count.saveAsTextFile("hdfs://master:9000/.../output")
```

- RDDs are lazy that are not materialized immediately. They can be made by

```
cachedErrs = errs.cache()
```


Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns (Scala)

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```


Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

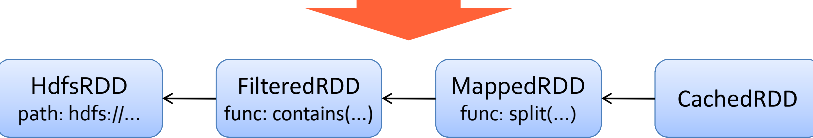
RDDs Revisited

- An RDD is an immutable, partitioned, logical collection of records
  - Need not be materialized, but rather contains information to rebuild a dataset from stable storage
- Partitioning can be based on a key in each record (using hash or range partitioning)
- Built using bulk transformations on other RDDs
- Can be cached for future reuse

RDD Fault Tolerance

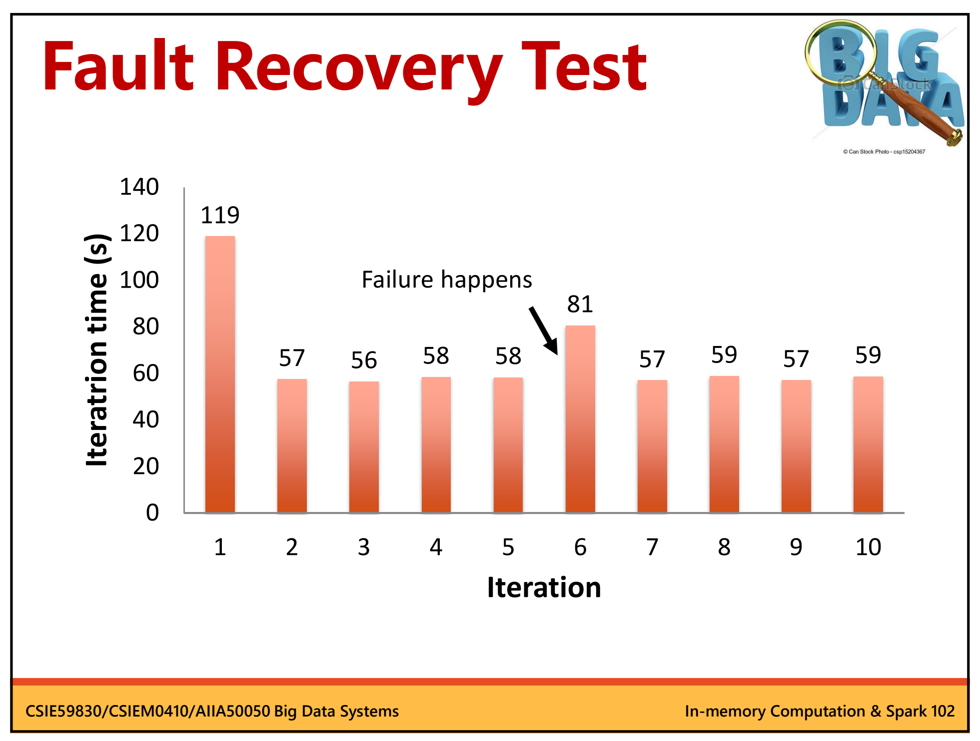
- RDDs maintain ***lineage*** information that can be used to reconstruct the exact lost partitions
- Ex:

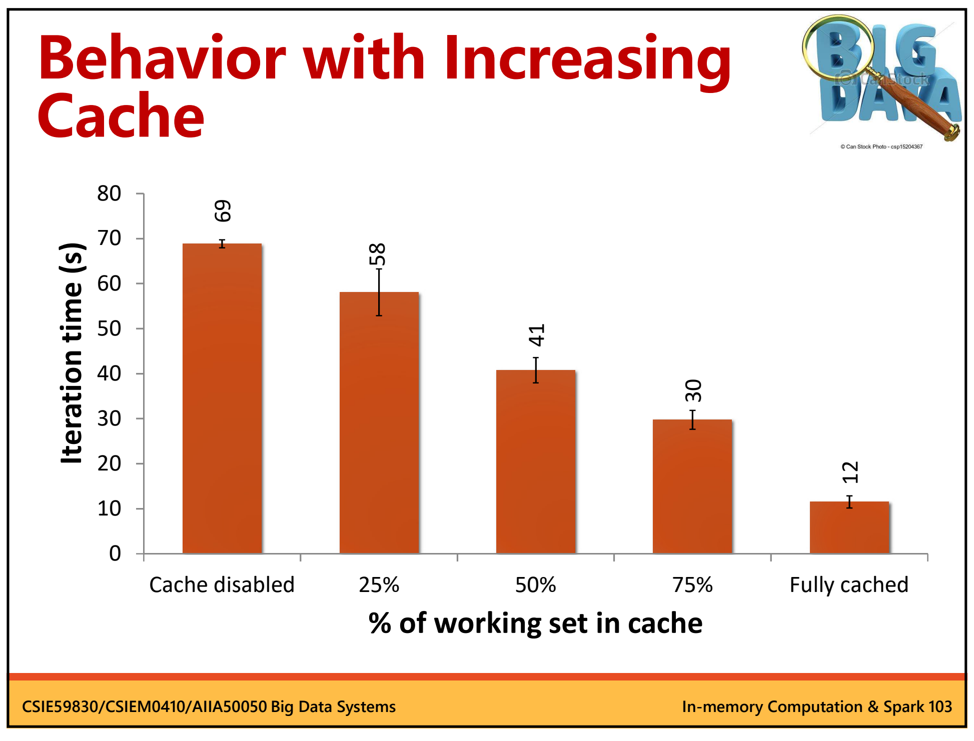
```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```



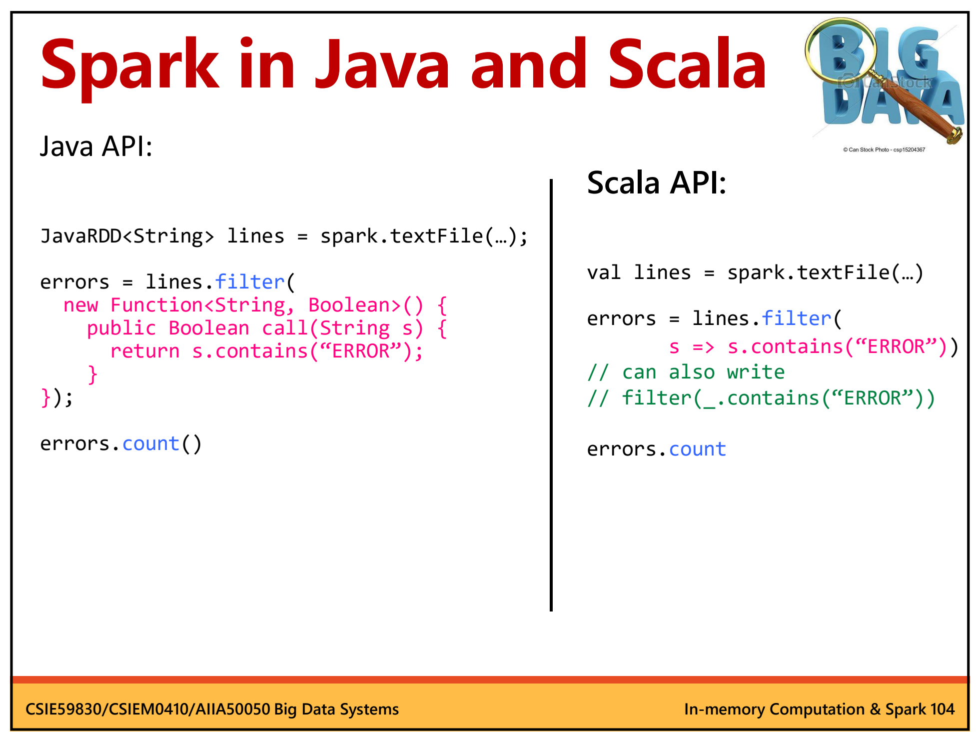
Benefits of RDD Model

- Consistency is easy due to immutability
- Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- Locality-aware scheduling of tasks on partitions
- High performance with in-memory computation
- Despite being restricted, model seems applicable to a broad variety of applications





Behavior with Increasing Cache
Iteration time (s)
80
70
60
50
40
30
20
10
0
69
58
41
30
12
Cache disabled
25%
50%
75%
Fully cached
% of working set in cache
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 103



Spark in Java and Scala
Java API:
JavaRDD<String> lines = spark.textFile(…);
errors = lines.filter(
new Function<String, Boolean>() {
public Boolean call(String s) {
return s.contains("ERROR");
}
});
errors.count()
Scala API:
val lines = spark.textFile(…)
errors = lines.filter(
s => s.contains("ERROR"))
// can also write
// filter(_.contains("ERROR"))
errors.count
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 104

Which Language to Use?

- Standalone programs can be written in any APIs
- On console (interactive shell), use different commands (spark-shell, pyspark, sparkr) for different languages (Scala, Python, R).
- **Python developers:** can stay with Python for both
- **Java developers:** consider using Scala for console (to learn the API)
- Performance: Scala/Java will be faster (statically typed), but Python can do well for numerical work with NumPy

Scala Cheat Sheet

Variables:

```
var x: Int = 7
var x = 7      // type inferred

val y = "hi"   // read-only
```

Functions:

```
def square(x: Int): Int = x*x

def square(x: Int): Int = {
  x*x    // last line returned
}
```

Collections and closures:

```
val nums = Array(1, 2, 3)

nums.map((x: Int) => x + 2) // => Array(3, 4, 5)

nums.map(x => x + 2)  // => same
nums.map(_ + 2)       // => same

nums.reduce((x, y) => x + y) // => 6
nums.reduce(_ + _)           // => 6
```

Java interop:

```
import java.net.URL

new URL("http://cnn.com").openStream()
```

More details: scala-lang.org

Learning Spark

- Easiest way: Spark interpreter (`spark-shell` or `pyspark`)
  - Special Scala and Python consoles for cluster user
- Runs in local mode on 1 thread by default, but can control with `--master` option:

```
$ pyspark --master local                # local, 1 thread
$ pyspark --master local[4]             # local, 4 threads
$ pyspark --master spark://master:9000  # master URL
```


First Step: SparkContext

- Main entry point to Spark functionality
- Created for you in Spark shells as variable `sc`
- In standalone programs, you'd make your own with

```
from pyspark import SparkContext
sc = SparkContext("local[2]", "PySparkErrorCount")
```

Cluster URL, or local / local[N]

App name

Creating RDDs

```
// Turn local objects into an RDDs
dat = sc.parallelize([1, 5, 60, 'a', 9, 'c', 4])
pair = sc.parallelize([('a', 6), ('a', 1), ('b', 2),
('c', 5), ('c', 8), ('c', 11)])

// Load text file from local FS, HDFS, or S3
distFile = sc.textFile("data.txt")
// sc.textFile("directory/*.txt")
// sc.textFile("hdfs://namenode:9000/path/file")

// Use any existing Hadoop InputFormat
sc.hadoopFile(path, inputFmtClass, keyClass,
valClass, valueConverter)
```


Common Transformations

- `map(func)`: Applies func to each element of the RDD.
- `filter(func)`: Returns a new RDD containing only the elements that satisfy the given predicate.
- `reduceByKey(func)`: Performs a reduction(func) on the elements with the same key.
- `flatMap(func)`: Similar to map where each input item is mapped to zero or more output items, then all output items are flattened.
- `mapValues(func)`: Applies func to the values of each key-value pair while keeping the key unchanged.

Transformations: Examples

```
nums = sc.parallelize([1, 2, 3])

// Pass each element through a function
squares = nums.map(lambda x: x*x)   // => {1, 4, 9}

// Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0)   // => {4}

// Map each element to zero or more others
Rng = nums.flatMap(lambda x: range(1,x))  // => {1, 1, 2}
```


Common Actions

- `collect()`: Retrieves all elements of an RDD and brings them to the driver program.
- `count()`: Returns the number of elements in an RDD.
- `first()`: Returns the first element of an RDD.
- `take(n)`: Returns the first n elements of an RDD.
- `reduce(func)`: Aggregates the elements of an RDD using a specified function.

Actions: Examples

```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```


Working with Key-Value Pairs

- Spark's "distributed reduce" transformations act on RDDs of *key-value pairs*
- Python:
```
pair = (a, b)
    pair[0] # => a
    pair[1] # => b
```
- Scala:
```
val pair = (a, b)
    pair._1 // => a
    pair._2 // => b
```
- Java:
```
Tuple2 pair = new Tuple2(a, b);  // scala.Tuple2
    pair._1 // => a
    pair._2 // => b
```

Some Key-Value Operations

```
pets = sc.parallelize([(“cat”, 1), (“dog”, 1), (“cat”, 2)])

pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}

pets.groupByKey()
# => {(cat, Seq(1, 2)), (dog, Seq(1)}

pets.sortByKey()
# => {(cat, 1), (cat, 2), (dog, 1)}
```

reduceByKey also automatically implements combiners on the map side

Spark for MapReduce

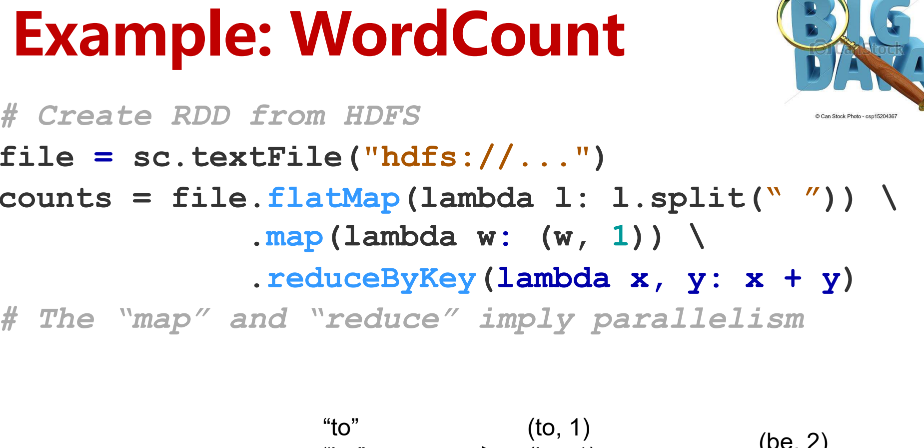
- MapReduce data flow can be expressed using RDD transformations

```
res = data.flatMap(lambda rec : myMapFunc(rec))
          .groupByKey()
          .map(lambda(key, vals) : myRdcFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(lambda rec : myMapFunc(rec))
          .reduceByKey(lambda x : myCombiner(x))
          .map(lambda key, vals : myReduceFunc(key, vals))
```

Example: WordCount



```
# Create RDD from HDFS
file = sc.textFile("hdfs://...")
counts = file.flatMap(lambda l: l.split(" ")) \
             .map(lambda w: (w, 1)) \
             .reduceByKey(lambda x, y: x + y)
# The "map" and "reduce" imply parallelism
```


WordCount Complete

```
from pyspark import SparkContext
# create Spark context with necessary configuration
sc = SparkContext("local[2]", "pyWordCount")

# read, split, map, and reduce all together
counts = sc.textFile("hdfs://master:9000/home/hadoop/input.txt") \
           .flatMap(lambda line: line.split(" ")) \
           .map(lambda word: (word,1)) \
           .reduceByKey(lambda a,b: a+b)

# save the counts to output
counts.saveAsTextFile("hdfs://master:9000/home/hadoop/output/")
```

ErrorCount Complete Example

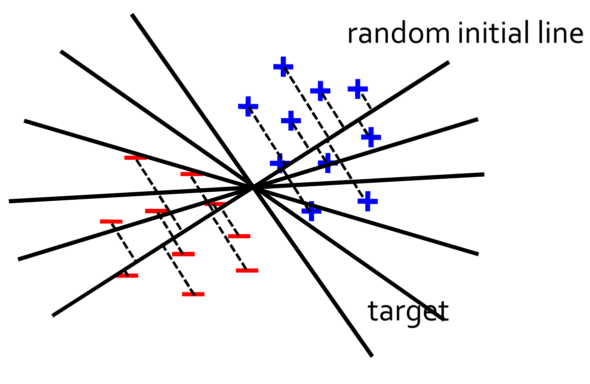
```
from pyspark import SparkContext

# setting Spark Context
sc = SparkContext("local[2]", "PySparkErrorCount")

# reading source text file from HDFS
file = sc.textFile("hdfs://master:9000/user/hadoop/tmp.txt")

# filter ERROR message
errs = file.filter(lambda x : "ERROR" in x)

# each line is counted once and sum all
ones = errs.map(lambda x : ("errorline", 1))
count = ones.reduceByKey(lambda x,y: x + y)

# save to HDFS
count.saveAsTextFile("hdfs://master:9000/user/hadoop/output")
```


Other Key-Value Operations

```
visits = sc.parallelize([ (“index.html”, “1.2.3.4”),
                          (“about.html”, “3.4.5.6”),
                          (“index.html”, “1.3.3.1”) ])
pageNames = sc.parallelize([ (“index.html”, “Home”),
                             (“about.html”, “About”) ])
visits.join(pageNames)
  # (“index.html”, (“1.2.3.4”, “Home”))
  # (“index.html”, (“1.3.3.1”, “Home”))
  # (“about.html”, (“3.4.5.6”, “About”))
visits.cogroup(pageNames)
  # (“index.html”, ([“1.2.3.4”, “1.3.3.1”], [“Home”]))
  # (“about.html”, ([“3.4.5.6”], [“About”]))
```

Example: Logistic Regression

- Goal: find best line separating two sets of points



Example: Logistic Regression

- An iterative classification algorithm to find a hyperplane *w* that best separates two sets of points.
- Popular binary classifier in machine learning
- Gradient Descent
  - ***ITERATIVELY*** minimizes the error by computing the gradient over ***all data points***
  - Computing among data points: parallelization
  - But the iterative instrinsic is another bottleneck

Logistic Regression Algorithm

```
w = random(D) // D-dimensional vector
for i from 1 to ITERATIONS do {
  //Compute gradient
  g = 0  // D-dimensional zero vector
  for every data point (sn, pn) do {
    // pn is a vector, sn is +1 or -1
    g += sn * pn / (1 + exp(sn * w * pn))
  }
  w -= LEARNING_RATE * g
}
```

Very big!!!!!!

Iterative Version

```
import numpy as np  # import the numeric lib
from math import exp
# Read and create points from a text file
points = sc.textFile(...)...
# Initialize w to a random D-dimensional vector
w = np.random.rand(D)
# Run multiple iterations to update w
for i in range(ITERATIONS):
  grad = np.zeros(D)
  for v in points.collect():
    d = v.s/(1+ exp(v.s * np.dot(w, v.p)))
    grad += np.dot(d, v.p)
  w -= LEARNING_RATE * grad
```

Spark: with accumulator

```
import ...
# Read points from a text file and cache them
points = sc.textFile(...).map(parsePoint).cache()
# Initialize w to a random D-dimensional vector
w = np.random.rand(D)
# Run multiple iterations to update w
for i in range(ITERATIONS):
  grad = sc.accumulator(np.zeros(D))
  points.foreach(lambda v:  # Run in parallel
    d = v.s/(1+ exp(v.s * np.dot(w, v.p)))
    grad += np.dot(d, v.p))
  w -= LEARNING_RATE * grad

# Need to define proper accumulator first
```

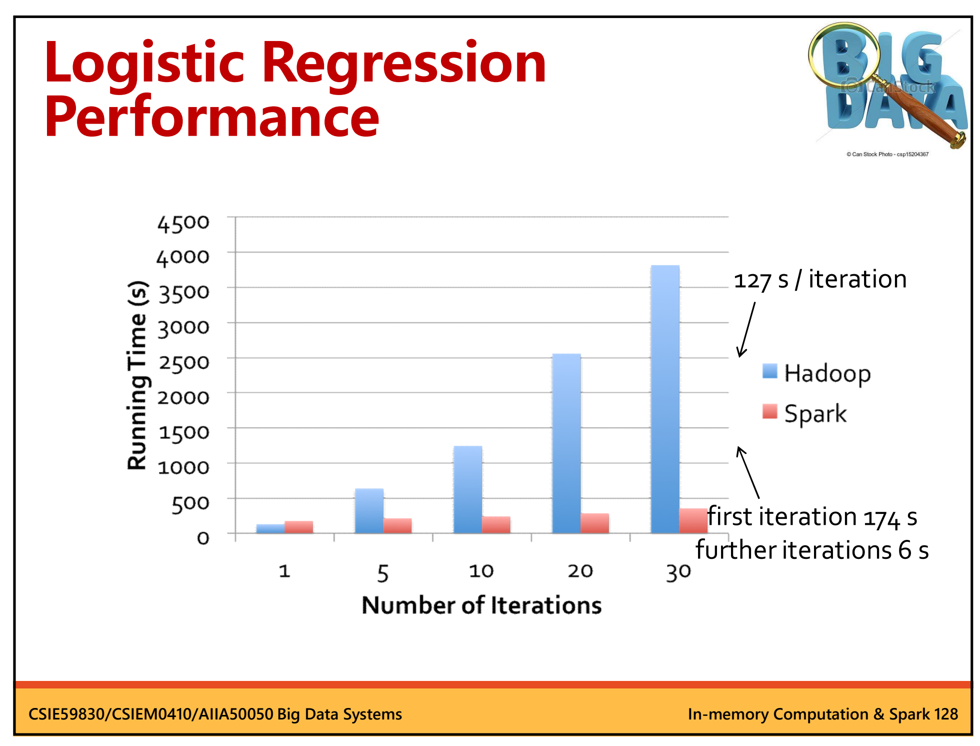

Spark: MP Version

```
import ...
# Read points from a text file and cache them
points = sc.textFile(...).map(parsePoint).cache()

# Initialize w to a random D-dimensional vector
w = np.random.rand(D)

# Run multiple iterations to update w
for i in range(ITERATIONS):
  grad = points.map(lambda v: v.s/(1+ exp(v.s *
np.dot(w, v.p)))).reduce(lambda x,y: np.add(x,y))
  w -= LEARNING_RATE * grad
```

Some Spark Features

- `points.foreach(body)` is an invocation of the Spark's parallel foreach operation
- Accumulator allows results of tasks running on clusters to be accumulated using operators like +=
- But defult accumulator only suppor numbers.
- Can define our own accumulator.
- Only the driver program can read the accumulator's value

Logistic Regression Performance

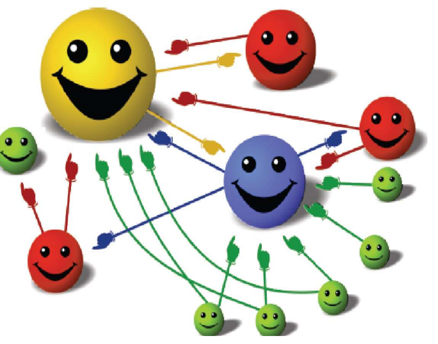


Example: PageRank

- Use PageRank as a Spark example
- Good example of a more complex algorithm
  - Multiple stages of map & reduce
- Benefits from Spark's in-memory computation
  - Multiple iterations over the same data

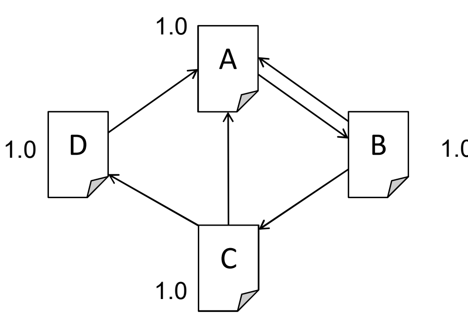
Basic Idea

- Give pages ranks based on links to them
  - Links from mamy pages -> high rank
  - Links from a high ranking page -> high rank



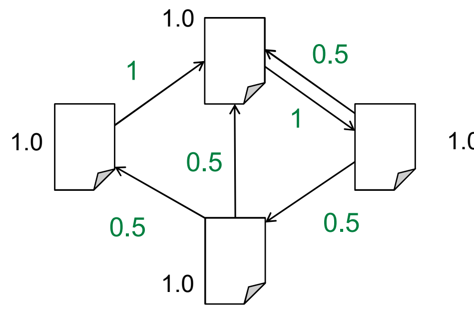
PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times contribs$



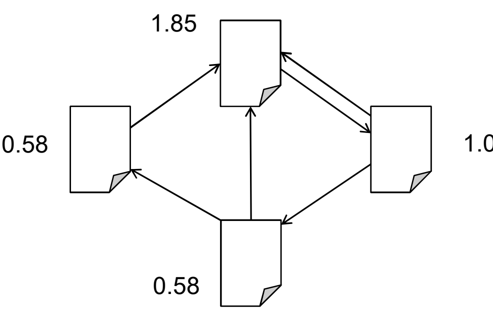
PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times contribs$



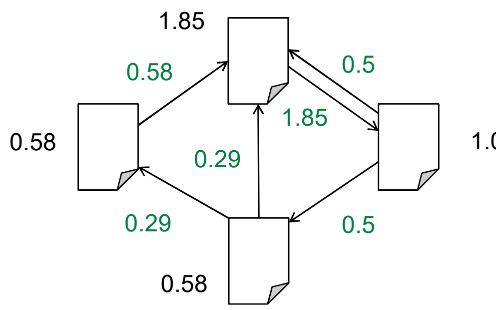
PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times contribs$



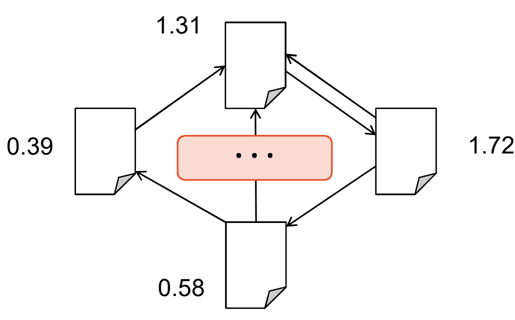
PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times contribs$



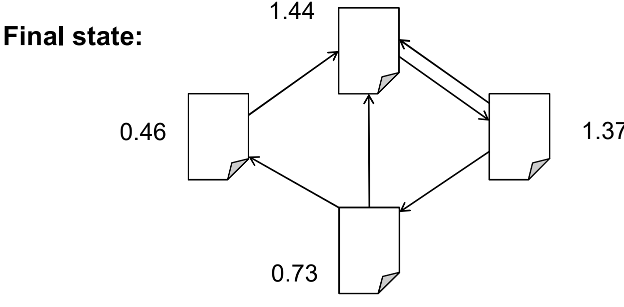
PageRank Algorithm

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times contribs$



PageRank Algorithm

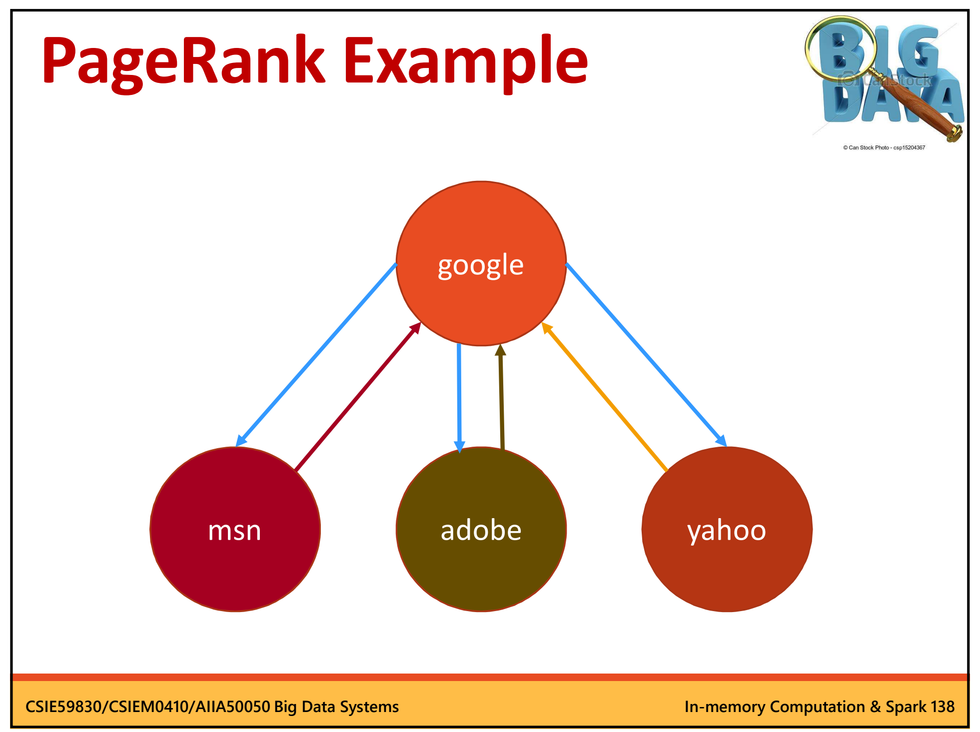
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p / |neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times contribs$

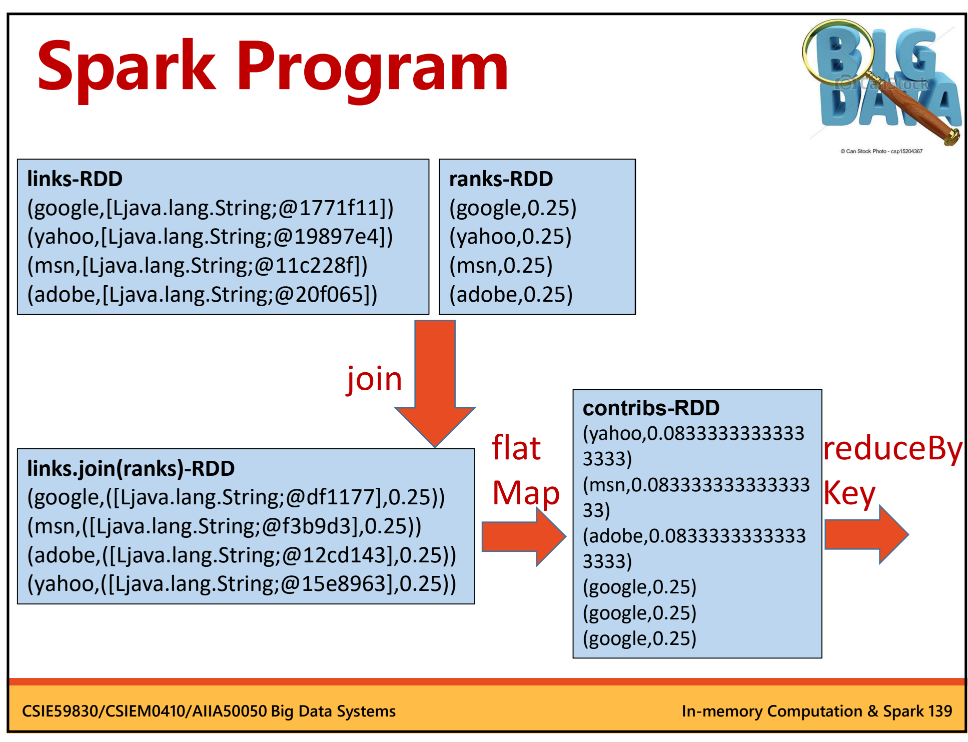


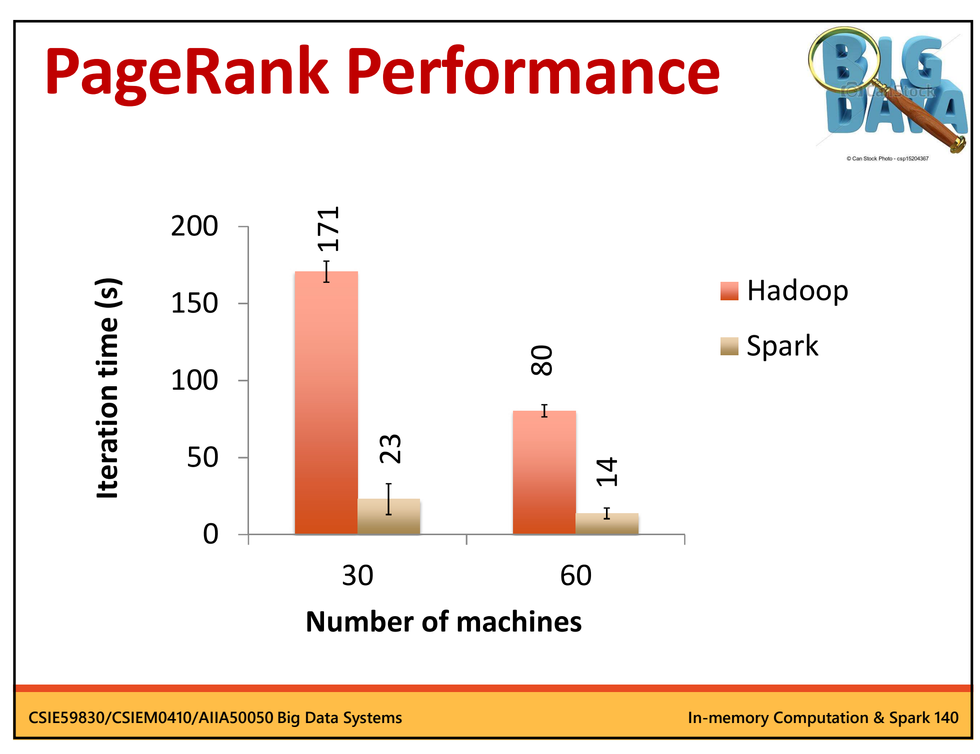
Spark Program

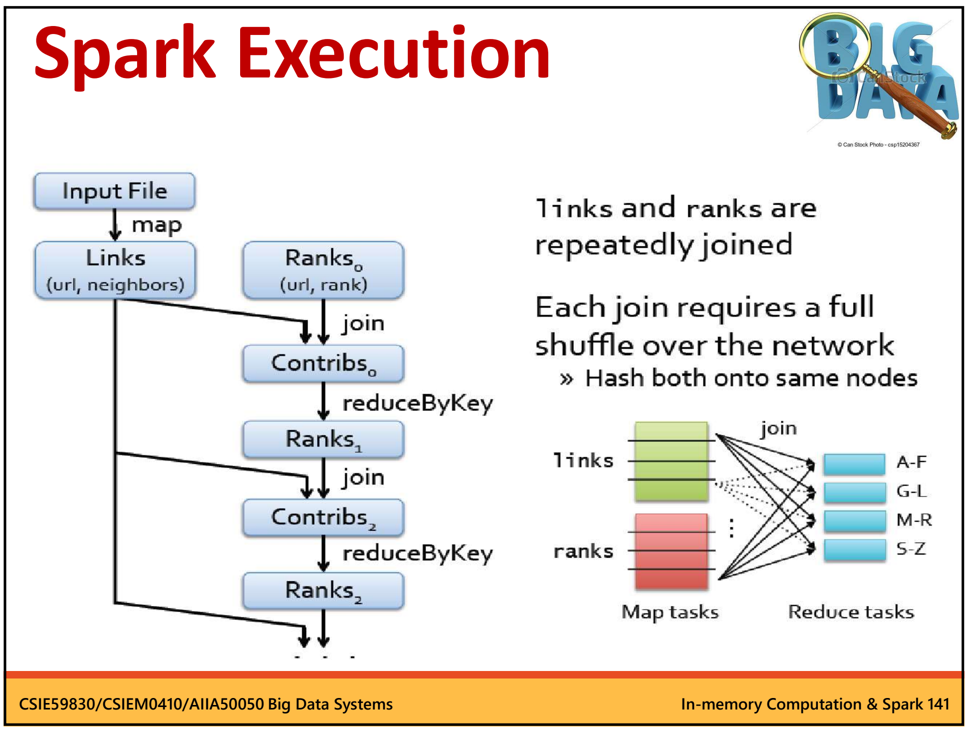
```
import . . .
# function to calculate rank contributions
def rCalc(neighbors, rank):
  n = len(neighbors)  # no. of neighbors
  for url in neighbors:
    yield(url, rank/n)
. . . # Read input graph from text file
links = ...cache()  # RDD of (url, neighbors) pairs
ranks = ...         # RDD of (url, rank) pairs
# perform rank update for ITERATIONS rounds
for i in range(ITERATIONS):
  contribs = links.join(ranks).flatMap(  # u_nr is (url, (neighbors, rank))
    lambda u_nr: rCalc(u_nr[1][0], u_nr[1][1]))
  ranks = contribs.reduceByKey(add).mapValues(lambda r: 0.15 + 0.85 * r)
ranks.saveAsTextFile(...)
```


PageRank Example




Spark Program
links-RDD
(google,[Ljava.lang.String;@1771f11])
(yahoo,[Ljava.lang.String;@19897e4])
(msn,[Ljava.lang.String;@11c228f])
(adobe,[Ljava.lang.String;@20f065])
ranks-RDD
(google,0.25)
(yahoo,0.25)
(msn,0.25)
(adobe,0.25)
join
links.join(ranks)-RDD
(google,([Ljava.lang.String;@df1177],0.25))
(msn,([Ljava.lang.String;@f3b9d3],0.25))
(adobe,([Ljava.lang.String;@12cd143],0.25))
(yahoo,([Ljava.lang.String;@15e8963],0.25))
flat Map
contribs-RDD
(yahoo,0.08333333333333333)
(msn,0.08333333333333333)
(adobe,0.08333333333333333)
(google,0.25)
(google,0.25)
(google,0.25)
reduceBy Key
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 139


PageRank Performance
Iteration time (s)
200
150
100
50
0
171
23
80
14
30
60
Number of machines
Hadoop
Spark
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 140



Solution: Controlled Partitioning

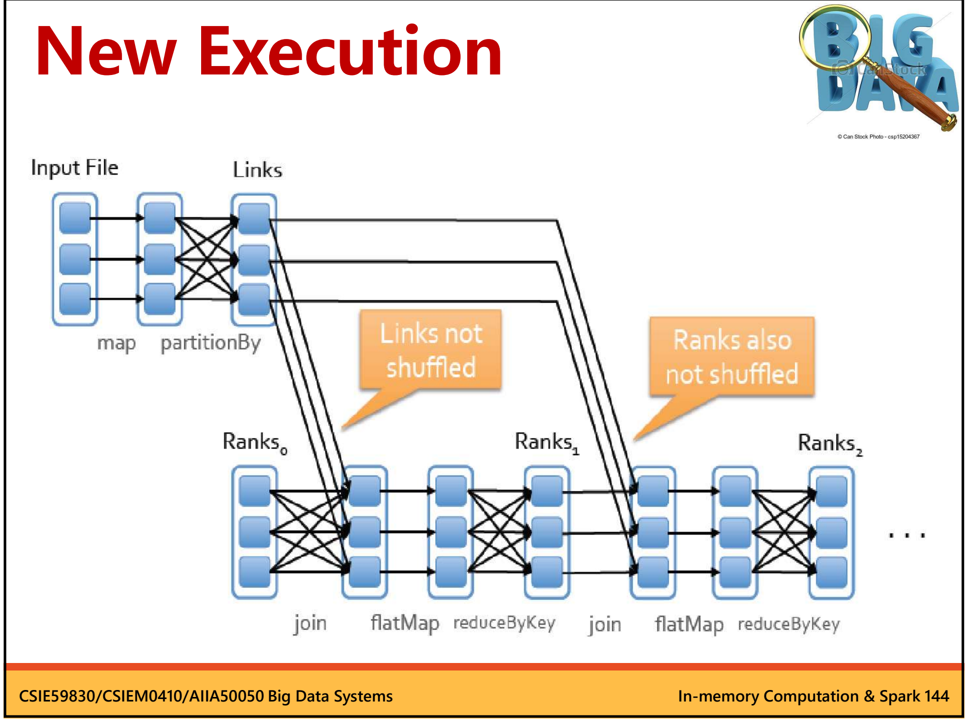
- Network bandwidth is ~100× as expensive as memory bandwidth
- *Pre-partition* the **links RDD** -

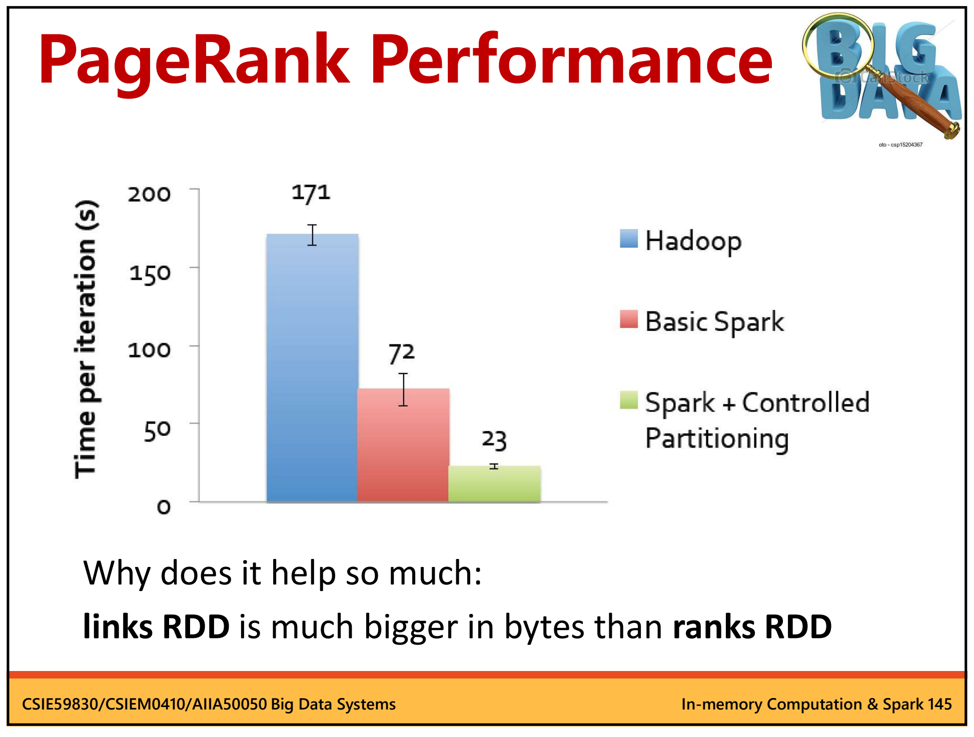
  so that links for URLs with the same hash code are on the same node

Controlled Partitioning

```
. . . # Read input graph from text file
links = ...partitionBy(4).cache()
ranks = ... # RDD of (url, rank) pairs
```


New Execution





PageRank Test

- Input data : simple.dat

  google: yahoo msn adobe

  yahoo: google

  msn: google

  adobe: google

- *numberIterations = 30, usePartitioner = false*
- *numberIterations = 30, usePartitioner = true*
- *numberIterations = 45, usePartitioner = false*
- *numberIterations = 45, usePartitioner = true*

Example: Alternating Least Squares (ALS)

- ALS is for collaborative filtering(CF) such as predicting u users' ratings for m movies based on past rating history.
- Both movies and user's preferences are represented as k-dim feature vectors.
- A user's rating to a movie is the dot product of the user's feature vector with the movie's.
- Let **M** be a m × k matrix and **U** be a k × u matrix of feature vectors, the rating **R** can be represented as **M** × **U**.

ALS Algorithm

- ALS algorithm: (detail not listed)
  1. Initialize M to a random value.
  2. Optimize U given M to minimize error on R.
  3. Optimize M given U to minimize error on R.
  4. Repeat steps 2 and 3 until convergence.
- All steps need R. It is helpful to make R a broadcast variable so that it does not re-sent to each node on each step.

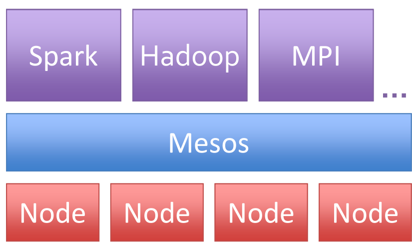
(https://spark.apache.org/docs/latest/ml-collaborative-filtering.html)

ALS Program in Spark

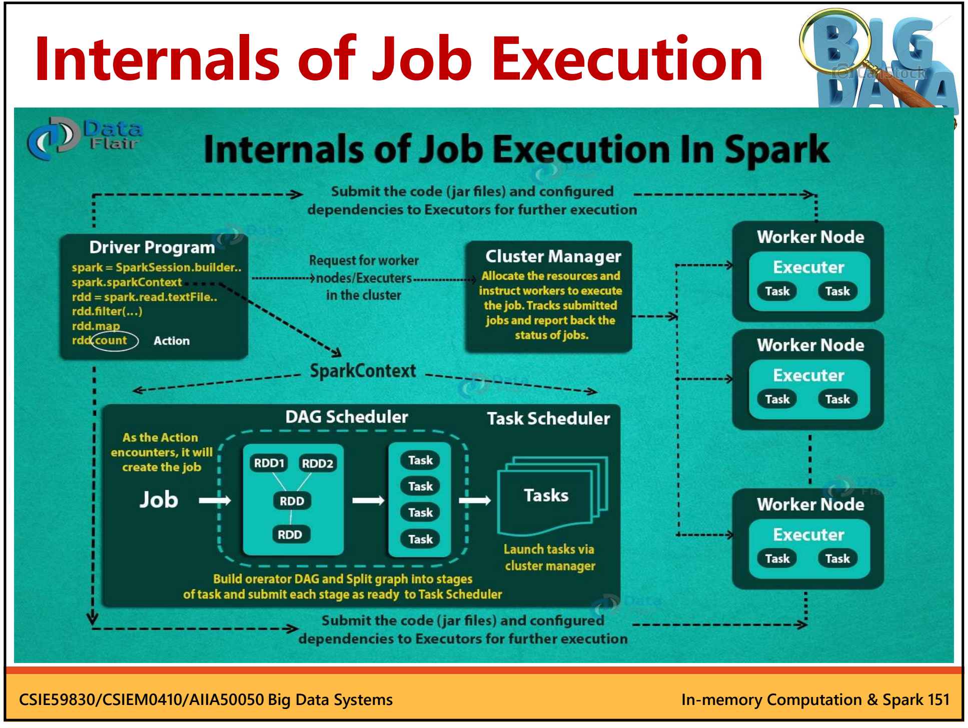
```
Rb = sc.broadcast(R)
for n in range(ITERATIONS):
  U = sc.parallelize(0 until u) \
        .map(lambda j: updateU(j, Rb, M)) \
        .collect()
  M = sc.parallelize(0 until m) \
        .map(lambda j: updateM(j, Rb, U)) \
        .collect()
```


Spark Implementation Overview

- Initially, Spark runs on the cluster manager (eg. Mesos) , to share resources with Hadoop & others
- Can read from any Hadoop input source (e.g. HDFS)



~6000 lines of Scala code thanks to building on Mesos



Language Integration

- Scala closures are Serializable Java objects
  - Serialize on driver, load & run on workers
- Not quite enough
  - Nested closures may reference entire outer scope
  - May pull in non-Serializable variables not used inside
  - Solution: bytecode analysis + reflection
- Shared variables implemented using custom serialized form (e.g. broadcast variable contains pointer to BitTorrent tracker)
- PySpark(with the Py4j library) allow easy integration of Python with Spark and JVM objects.

Interactive Spark

- Shell interpreter allows Spark to be used interactively from the command line
- Required two changes:
  - Modified wrapper code generation so that each "line" typed has references to objects for its dependencies
  - Place generated classes in distributed file system
- Enables in-memory exploration of big data

Spark Initial Remarks

- By making distributed datasets a first-class primitive, Spark provides a simple, efficient programming model for stateful data analytics
- RDDs provide:
  - Lineage info for fault recovery and debugging
  - Adjustable in-memory caching
  - Locality-aware parallel operations
- Spark can be the basis of a suite of batch and interactive data analysis tools

DataFrames

- DataFrames are a later addition to Spark (early 2015).
- The DataFrames API:
  - intended to enable wider audiences beyond "Big Data" engineers to leverage the power of distributed processing
  - inspired by data frames in R and Python (Pandas)
  - designed from the ground-up to support modern big data and data science applications
  - an extension to the existing RDD API

DataFrames: Features

DataFrames have the following features:

- Ability to scale from kilobytes of data on a single laptop to petabytes on a large cluster
- Support for a wide array of data formats and storage systems
- State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer
- Seamless integration with all big data tooling and infrastructure via Spark
- APIs for Python, Java, Scala, and R

DataFrames: Features

- For new users familiar with data frames in other programming languages, this API should make them feel at home.
- For existing Spark users, the API will make Spark easier to program.
- For both sets of users, DataFrames will improve performance through intelligent optimizations and code-generation.

Construct a DataFrame

- Python

```
# Construct a DataFrame from a "users" table in Hive.
df = sqlContext.table("users")
# Construct a DataFrame from a log file in S3.
df = sqlContext.load("s3n://someBucket/path/to/data.json", "json")
```

- Scala

```
val people = sqlContext.read.parquet("...")
```

- Java

```
DataFrame people = sqlContext.read().parquet("...")
```

Using DataFrames

```
# Create a new DataFrame that contains only "young" users
young = users.filter(users["age"] < 21)
# Alternatively, using a Pandas-like syntax
young = users[users.age < 21]
# Increment everybody's age by 1
young.select(young["name"], young["age"] + 1)
# Count the number of young users by gender
young.groupBy("gender").count()
# Join young users with another DataFrame, logs
young.join(log, logs["userId"] == users["userId"], "left_outer")
```


DataFrames and Spark SQL

- DataFrames are fundamentally tied to Spark SQL.
- The DataFrames API provides a programmatic interface—a domain-specific language (DSL)—for interacting with your data
- Spark SQL provides a SQL-like interface.
- What you can do in Spark SQL, you can do in DataFrames and vice versa.

```
young.registerTempTable("young")
sqlContext.sql("SELECT count(*) FROM young")
```

Datasets

- Dataset API provides a type-safe, object-oriented programming interface.
- DataFrame is an alias for untyped Dataset.
- Provide compile-time type safety.
- Offer high-level domain-specific language operations like sum(), join(), select(), groupBy().
- Making code safer, easier and more natural.
- Provide the benefits of RDDs and Spark SQL's optimized execution engine.
- Available in Scala and Java.

Create Datasets

- Two ways: dynamically or read from external files
- Example: Create 100 integers as Dataset[Long]

```
// range of 100 numbers to create a Dataset.
val range100 = spark.range(100)
// try range100.collect() to see it
```

- Example: Read from an external JSON file

```
val df =
spark.read.json("/samples/people.json")
```

- Also from CSV, Text, Parquet, ORC, etc.

Read with Schema

- Can define the schema before reading

```
// Define a class to represent a type-specific obj
case class Person(id:Int, name: String, age: Long)
```

- Read a JSON file into the class format

```
val ds =
spark.read.json("/samples/people.json").as[Person]
```

- Upon reading, will create a generic DataFrame=Dataset[Rows] which convert a DataFrame into a type-specific object.

Create Datasets

- Can create a new Dataset from existing Datasets

```
// in Scala; names is a Dataset[String]
val names = ds.map(_.name)
```

- The most common way is to read from files first and transform them if necessary.

```
val young = ds.filter(d => d.age < 25)
              .map(d => (d.ID, d.name))
```

Print the Content

- Can print the content with standard Spark commands

```
ds.take(10).foreach(println(_))
```

- The above will print the first 10 rows of the ds

Using SQL-like Query

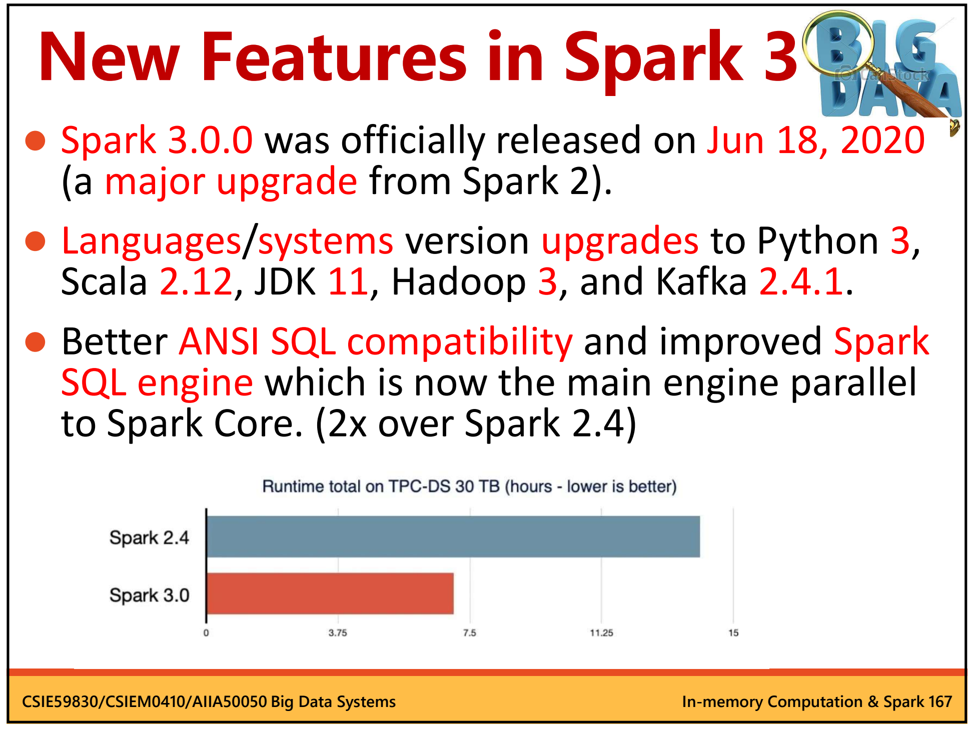
- Can process a Dataset with SQL-like query
- Select a named column

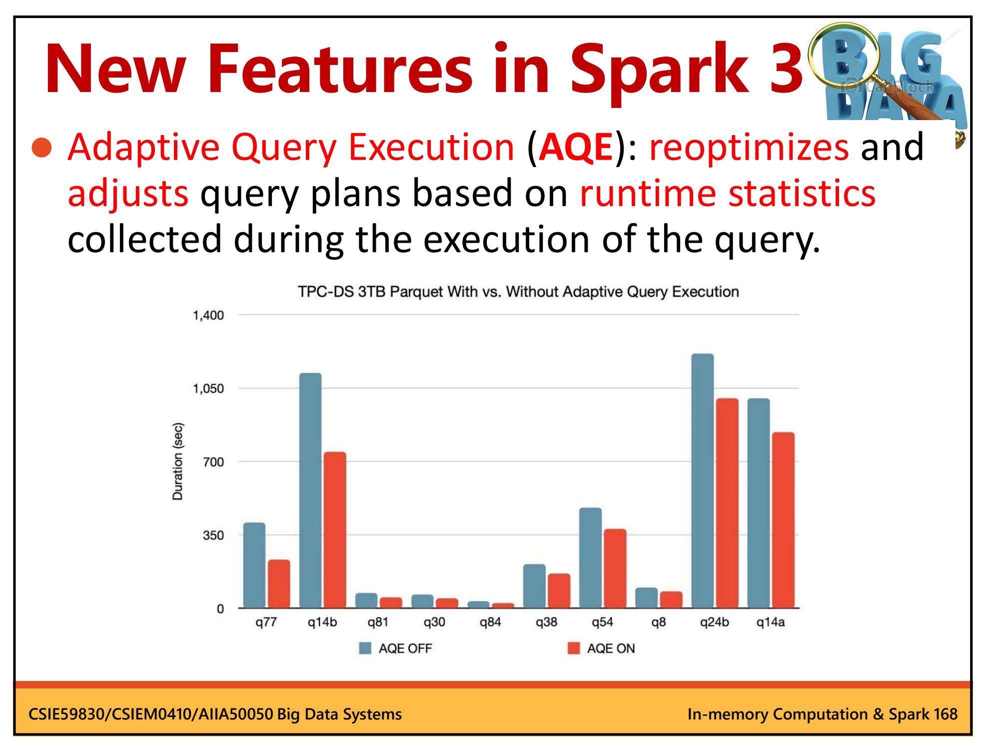
```
val ageCol = ds("age")
```

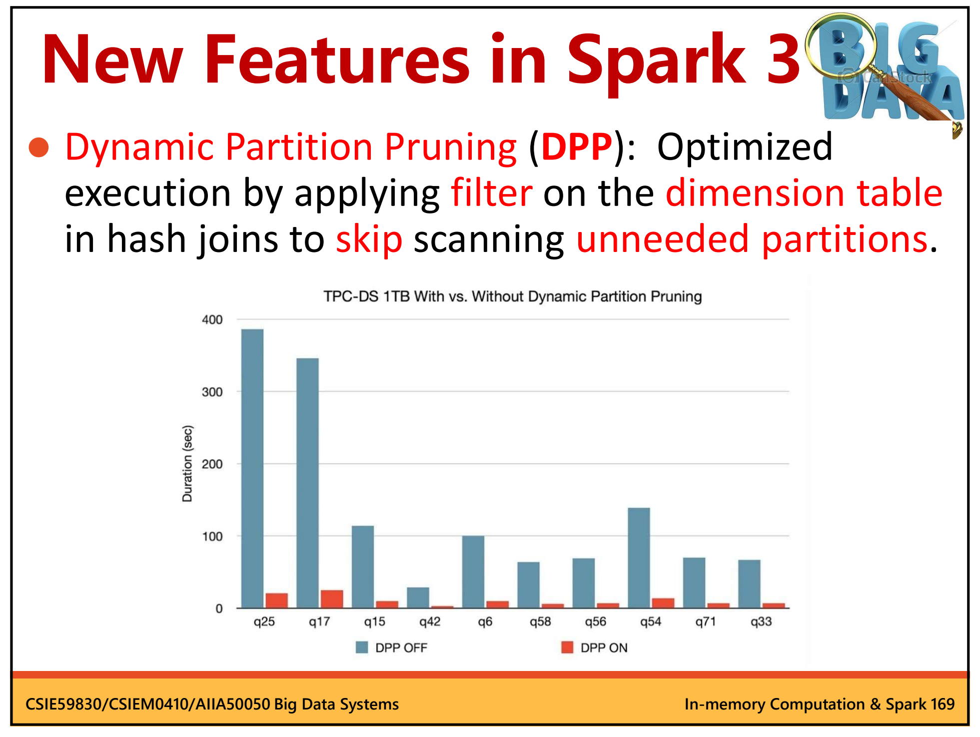
- Can use SQL-like query

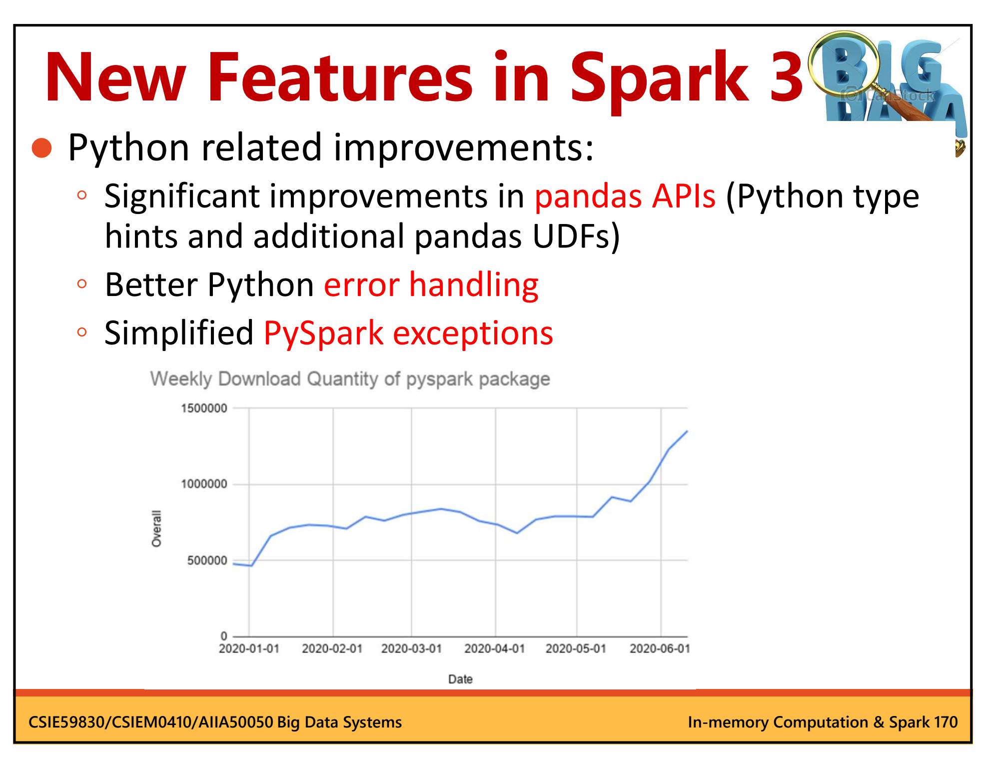
```
val old = ds.select($“name", $“age")
            .where($"age" > 60)
            .sort($"name")
```

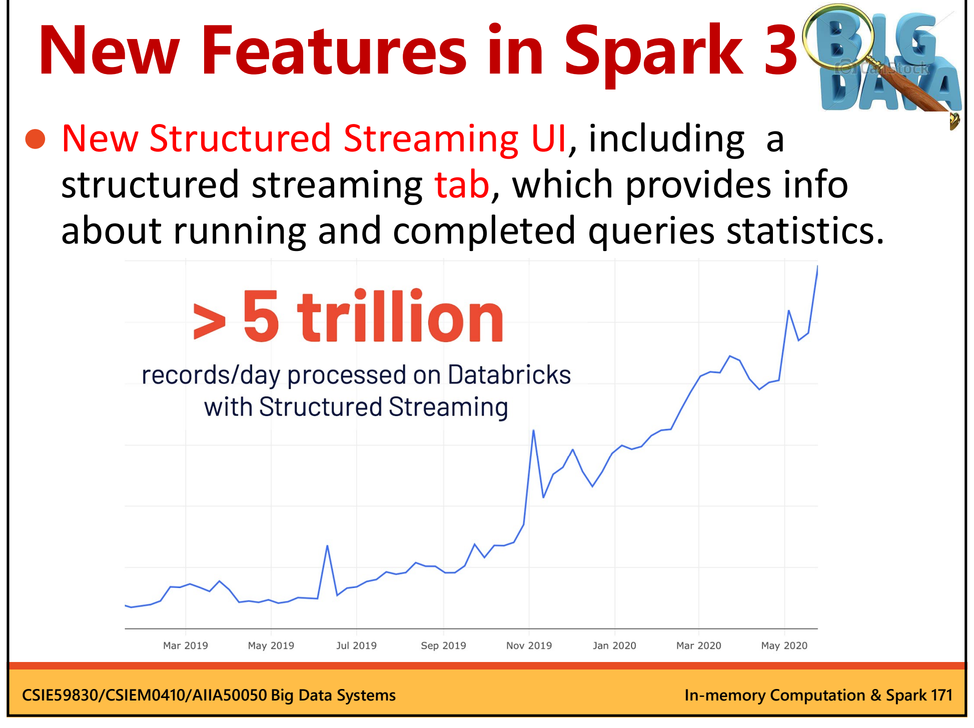
- In general, Datasets are powerful and friendly


New Features in Spark 3
Spark 3.0.0 was officially released on Jun 18, 2020 (a major upgrade from Spark 2).
Languages/systems version upgrades to Python 3, Scala 2.12, JDK 11, Hadoop 3, and Kafka 2.4.1.
Better ANSI SQL compatibility and improved Spark SQL engine which is now the main engine parallel to Spark Core. (2x over Spark 2.4)
Runtime total on TPC-DS 30 TB (hours - lower is better)
Spark 2.4
Spark 3.0
0
3.75
7.5
11.25
15
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 167


New Features in Spark 3
Adaptive Query Execution (AQE): reoptimizes and adjusts query plans based on runtime statistics collected during the execution of the query.
TPC-DS 3TB Parquet With vs. Without Adaptive Query Execution
1,400
1,050
700
350
0
Duration (sec)
q77
q14b
q81
q30
q84
q38
q54
q8
q24b
q14a
AQE OFF
AQE ON
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 168

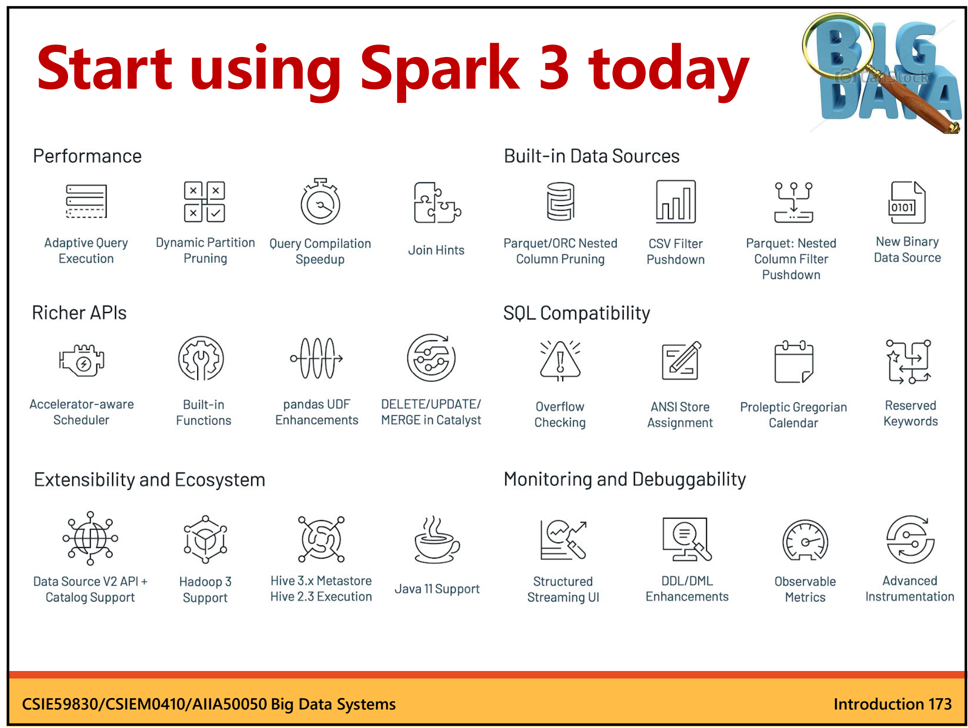

New Features in Spark 3
Dynamic Partition Pruning (DPP): Optimized execution by applying filter on the dimension table in hash joins to skip scanning unneeded partitions.
TPC-DS 1TB With vs. Without Dynamic Partition Pruning
Duration (sec)
400
300
200
100
0
q25
q17
q15
q42
q6
q58
q56
q54
q71
q33
DPP OFF
DPP ON
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 169


New Features in Spark 3
Python related improvements:
Significant improvements in pandas APIs (Python type hints and additional pandas UDFs)
Better Python error handling
Simplified PySpark exceptions
Weekly Download Quantity of pyspark package
1500000
1000000
500000
0
Overall
2020-01-01
2020-02-01
2020-03-01
2020-04-01
2020-05-01
2020-06-01
Date
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems
In-memory Computation & Spark 170



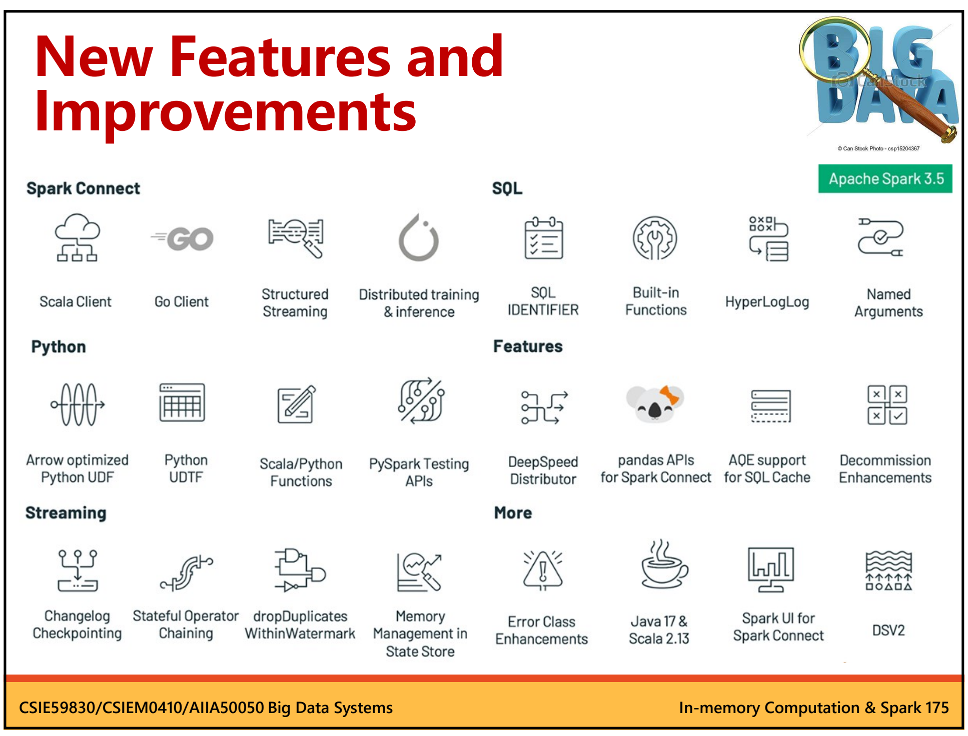
New Features in Spark 3

- Accelerator-aware Scheduling: Users can specify GPU accelerators via configuration and call new RDD APIs to leverage them.
- New Spark built-in functions (32 functions)
- Datasource format "binaryFile" to read binary files
- Up to 40x speedups for calling R user-defined functions.
- A whole new module **Spark Graph** with major features (eg. query language Cypher) for Graph processing.



Spark 3.5 – What's New

- Spark Connect: General availability of the Scala client, support for distributed training and inference, parity of Pandas API on SPARK.
- New PySpark and SQL functionality: SQL IDENTIFIER clause, named argument support for SQL function calls, SQL function support for HyperLogLog approximate aggregations, and Python user-defined table functions.
- Distributed training with DeepSpeed: Simplified configuration and improved performance.
- Performance and stability improvements in the **RocksDB state store provider**
- **Structured Streaming:** improved compatibility
- English SDK for Apache Spark enables users to utilize plain English as their programming language, making data transformations more accessible and user-friendly.



Assignment 2a

1. Implement the PageRank algorithm with Spark and provide suitable input to test it.
2. Given a file with store sales records in the format:

   StoreID ItemID1 #sold1 ItemID2 #sold2 …

   and a file of item prices in the format:

   ItemID1 price1 ItemID2 price2 …

   Write a Spark program to compute the total sales of each store, the total number sold of each item, the average total sales and the grand total sales of all stores.
3. Write a Spark program to compute the inverted index and frequency counts of keywords on a set of documents. More specifically, given a set of (DocumentID, text) pairs, output a list of (word, ((doc1, #1), (doc2, #2) …)) pairs.

Assignment 2b

4. Given a text file of purchase records and a threshold θ, write a Spark program to find all sets of frequent items that are purchased together. Each line of the input is a transaction of the format

   <tid> $item_1$ $item_2$ . . .

   where <tid> is the transaction ID and $item_i$ are the purchased items (all represented by integer IDs). A set of items is considered frequent if it appears in at least θ transactions. Keep in mind that purchase order is irrelevant. {A, B} is the same as {B, A}. If a set appears in a transaction, it is only counted once no matter how many times it appears in that transaction.

- Due date: 3 weeks from now