U O W

Data Analytics with Apache Spark

The slides are from the lecture notes of CSIC316 Big Data Techniques and Implementation

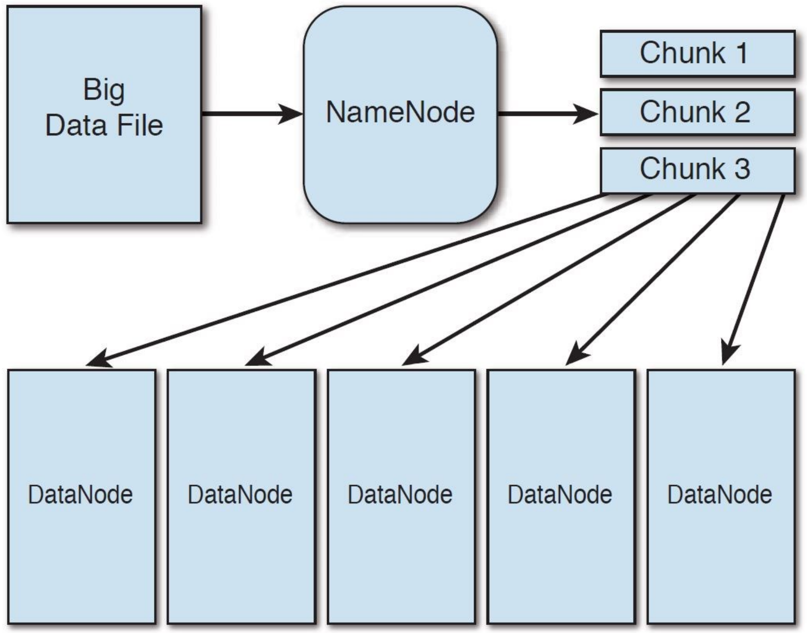
UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Agenda

- From MapReduce to Spark
- Parallel computation for Gradient-Based Optimization
- Spark's machine learning library

Massive Data Storage

- Massive datasets cannot be held in memory or even stored in a single hard disk of commodity hardware
  - Solution: Distributed storage (e.g., Google's distributed file system, Hadoop's HDFS)

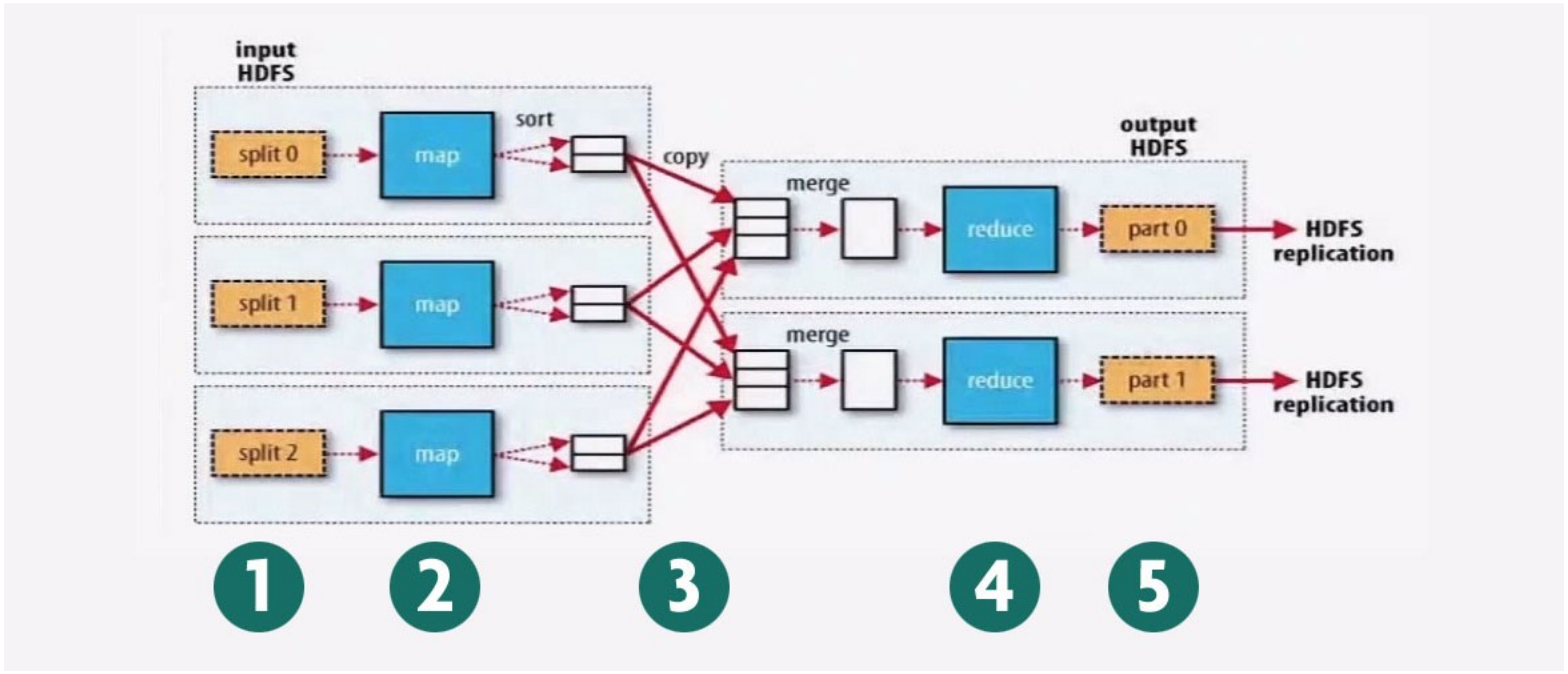


Processing Massive Data

- Processing massive datasets requires long runtime
  - One solution to speed up the computation is *parallelism (or distributed computing)*
- One preeminent model of parallel computation is **MapReduce**
- A very simple model:
  - One *Map* stage, which performs simple mapping-alike operations to produce key-value pairs,
  - One intermediate stage, which merges key-value pairs per key
  - One *Reduce* stage, which performs aggregation-alike operations per key
- However...
  - from the algorithm point of view: very powerful!
  - from the implementation point of view: very suitable for a computing cluster

Hadoop MapReduce Framework

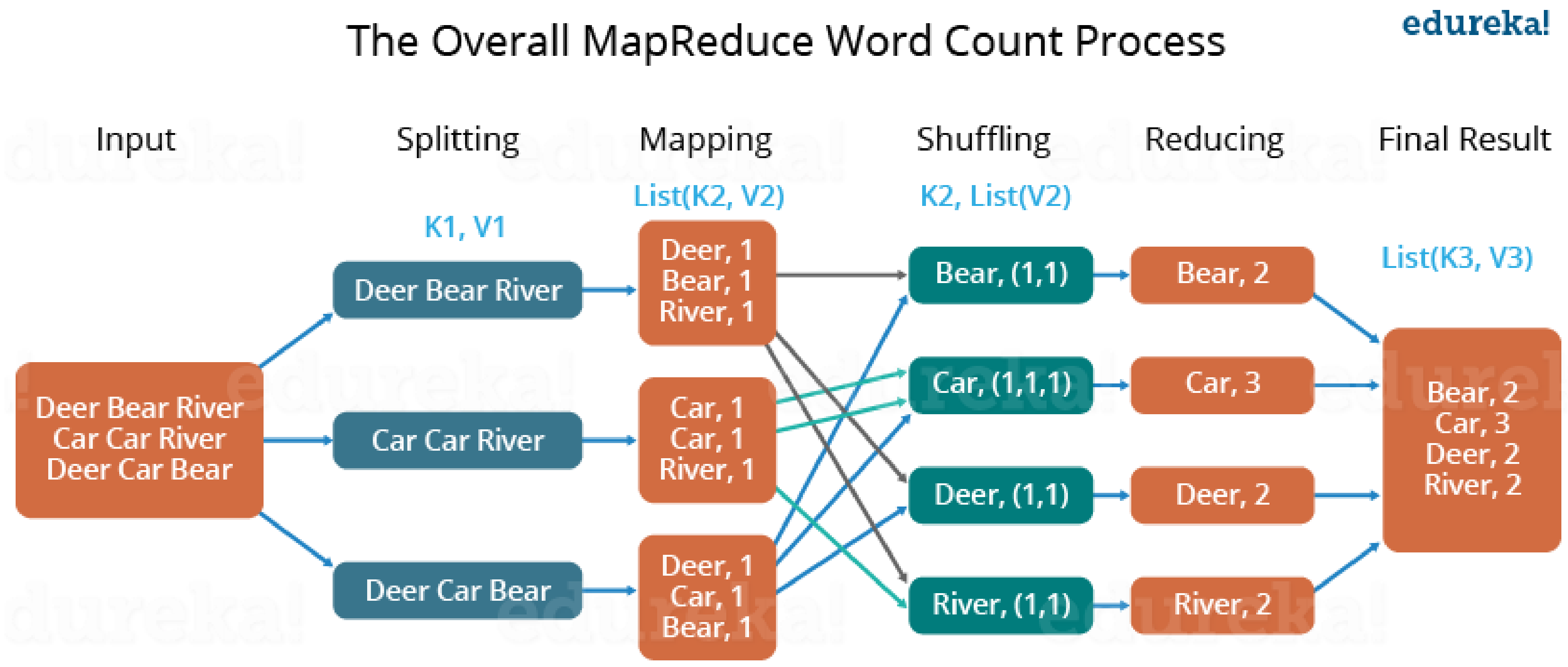
- The MapReduce:
  - (1) reading, (2) mapping, (3) shuffle-and-sort, (4) reducing and (5) writing



UNIVERSITY OF WOLLONGONG AUSTRALIA

MapReduce: The WordCount Example

- MapReduce uses (key, value) as the basic data structure.



Relational-Algebra Operations in MapReduce

- Although big data frameworks are not traditionally DB systems, **relational-algebra operations** are useful, especially in preprocessing.
- Recall that a *relation* is a table. We call the column headers as *attributes* and the rows as *tuples*. The bag of attributes of a relation is called its *schema.*
- We use $R[A_1, \ldots, A_n]$ to denote a relation R with schema $A_1, \ldots, A_n$.

❖ *All common SQL queries can be implemented with MapReduce*

Relational-Algebra Operations in MapReduce

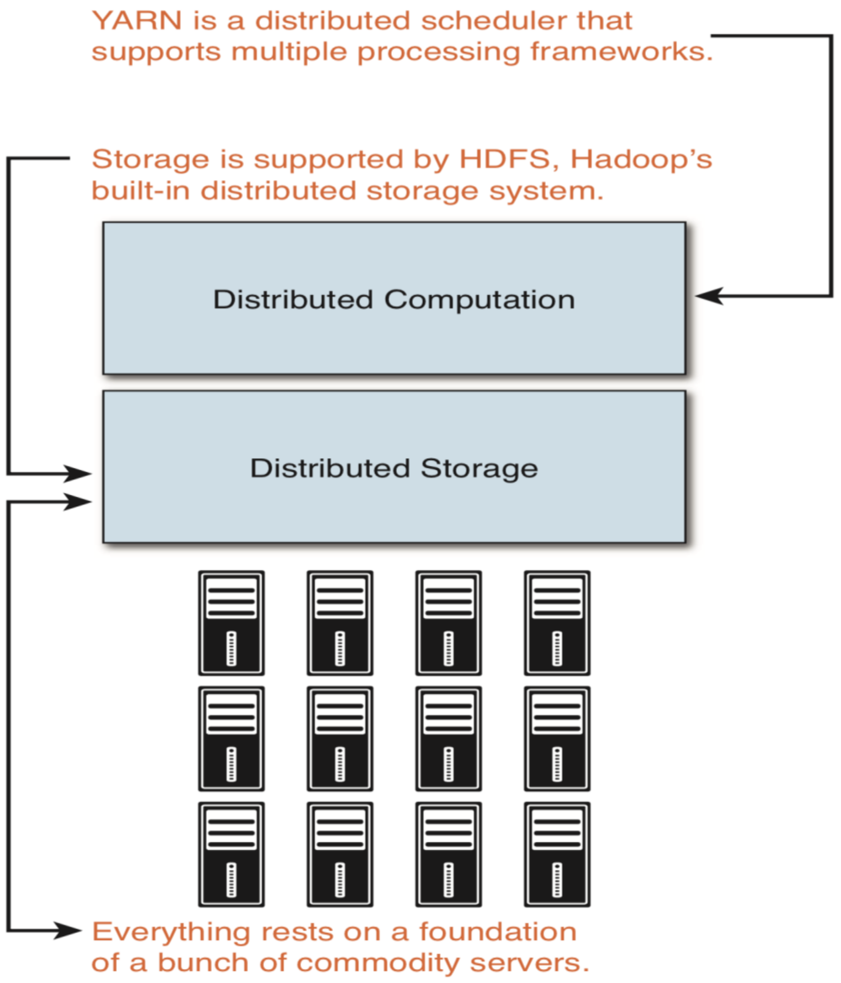
- **Example: Natural Join**: Given two relations, compare each pair of tuples, one from each relation. If the two tuples agree on all the attributes that are common to the two schemas, then produces a tuple that has components for each of the attributes in either schema or both.
- Computing Natural Join in MapReduce
  - The Map function: For each tuple (a, b) in $R[A, B]$, produce the key-value pair $(b, (R, a))$. For each tuple (b, c) in $S[B, C]$, produces the key-value pair $(b, (S, c))$.
  - The Reduce function: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). Construct ALL tuples (a, b, c) if both (R, a) and (S, c) appear in the list that b is associated with.

Matrix-Vector Multiplication in MapReduce

- **Matrix-vector multiplications** are widely used in machine learning.
- Let $\boldsymbol{M}$ be an $n \times n$ matrix whose element in row i and column j is m_{ij}; let $\boldsymbol{v}$ be an vector of length n whose jth element is v_j. And n is extremely large (e.g., in the order of 10^{12})
- The matrix-vector product is $x_i = \sum_{j=1}^{n} m_{ij} v_j$
- Process:
  - In each computing node, load $\boldsymbol{v}$ and a *chunk of* $\boldsymbol{M}$ into the memory,
  - For each m_{ij}, the map function computes a key-value pair $(i, m_{ij} v_j)$
  - For each key i, the reduce function computes $(i, \sum_j m_{ij} v_j)$
- If $\boldsymbol{v}$ cannot be loaded into the memory (but can be stored in the hard disk), a more sophisticate process achieve the same purpose
  - I.e., breaking the matrix and vector into stripes.

Hadoop MapReduce Framework

- Apache Hadoop is an open-source, Java-based, distributed computing platform
  - Build on commodity hardware
  - Fault-tolerance
  - Inexpensive
- Hadoop architecture:

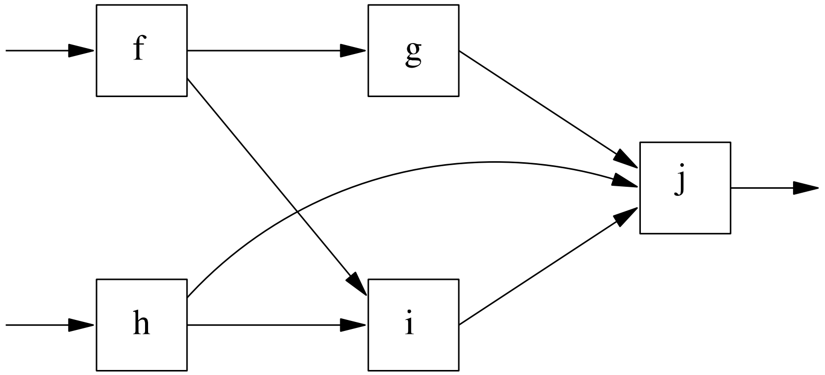

YARN is a distributed scheduler that supports multiple processing frameworks.
Storage is supported by HDFS, Hadoop's built-in distributed storage system.
Distributed Computation
Distributed Storage
Everything rests on a foundation of a bunch of commodity servers.

Hadoop MapReduce Framework

- MapReduce is a processing framework and programming model.
- Allows to process big files stored on HDFS in parallel.
- Developers focus on the implementation of "map" and "reduce" functions, not the low-level distributing computing subtilties (e.g., scheduling, IO).
- For a long period, people had believed that it was *the only* big data processing model -- it is not!
- Two important recent frameworks are Apache Spark and Google's TensorFlow.

From MapReduce to Workflow Systems

- Workflow systems extend MapReduce
  - from a simple two-step model (with a mapper and a reducer) to a orchestration of any steps that form an ***acyclic directed graph*** (DAG)
  - Although in theory is possible to pipeline MapReduce jobs to form any workflow, however…
  - You need to store the temporal output of intermediate jobs in HDFS (which is a natural idea in MapReduce) rather than keep it in memory
  - The idea of ***in-memory computing*** is the main feature of some workflow systems (e.g., Apache Spark)

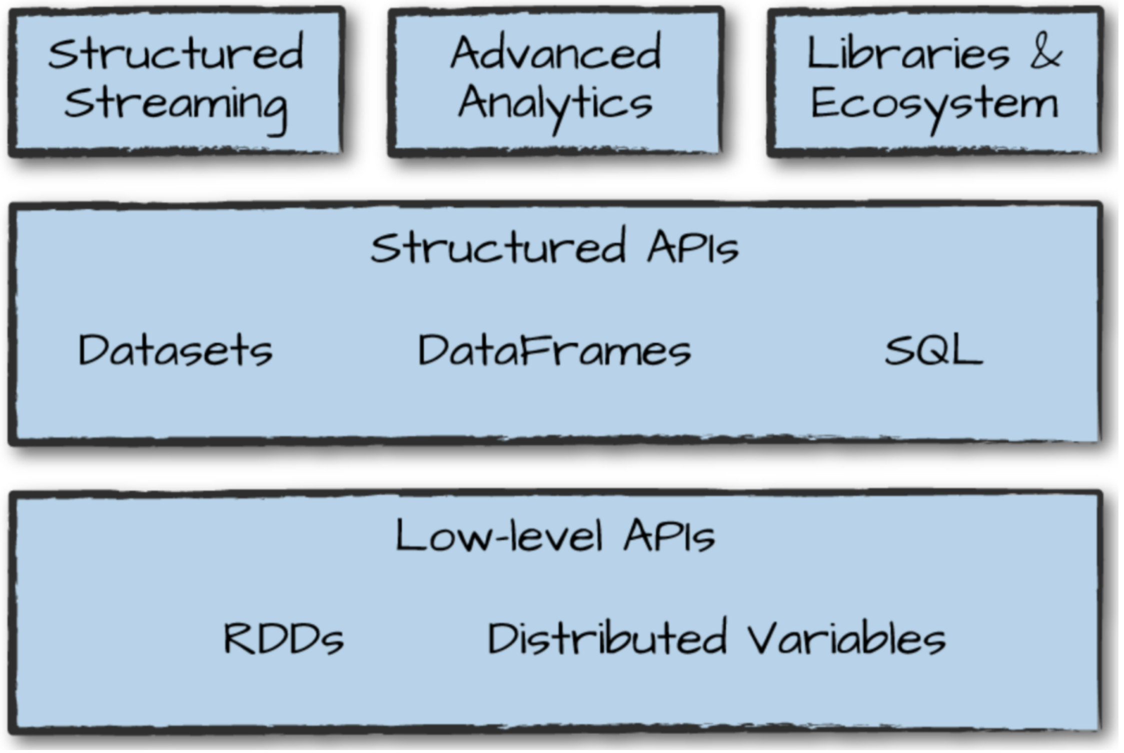


UNIVERSITY OF WOLLONGONG AUSTRALIA

Apache Spark

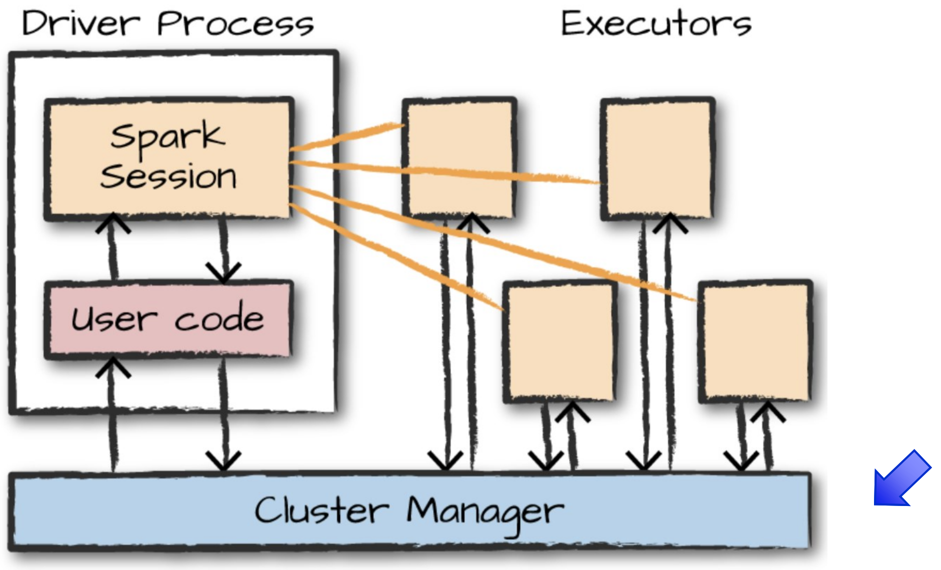
- A *unified* development environment
  - Supports a wide range of data analytics tasks, from SQL queries and ETL to stream processing, and to advanced data analytics and machine learning
  - From small-scale validation of programs to application development for production
  - Multiple language support
    - e.g., Python, Scala, Java, SQL and R
- A computing engine
  - Integrates with other software frameworks (e.g., Hadoop, Kakfa and Kubernetes)

Spark’s Component Stacks



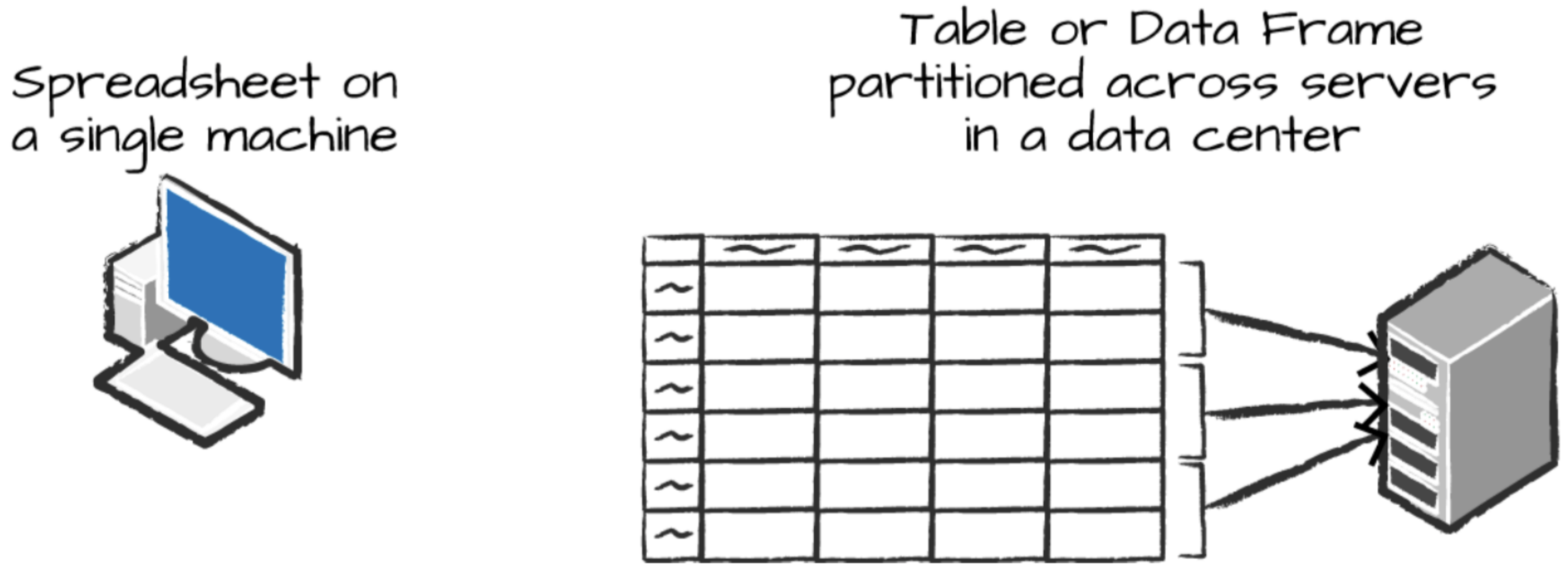
Architecture of Spark

- Each Spark application consists of a ***driver*** process and a set of ***executor*** processes.
  - The driver process runs your main() function
  - The executors carry out the actual work of the application (job)

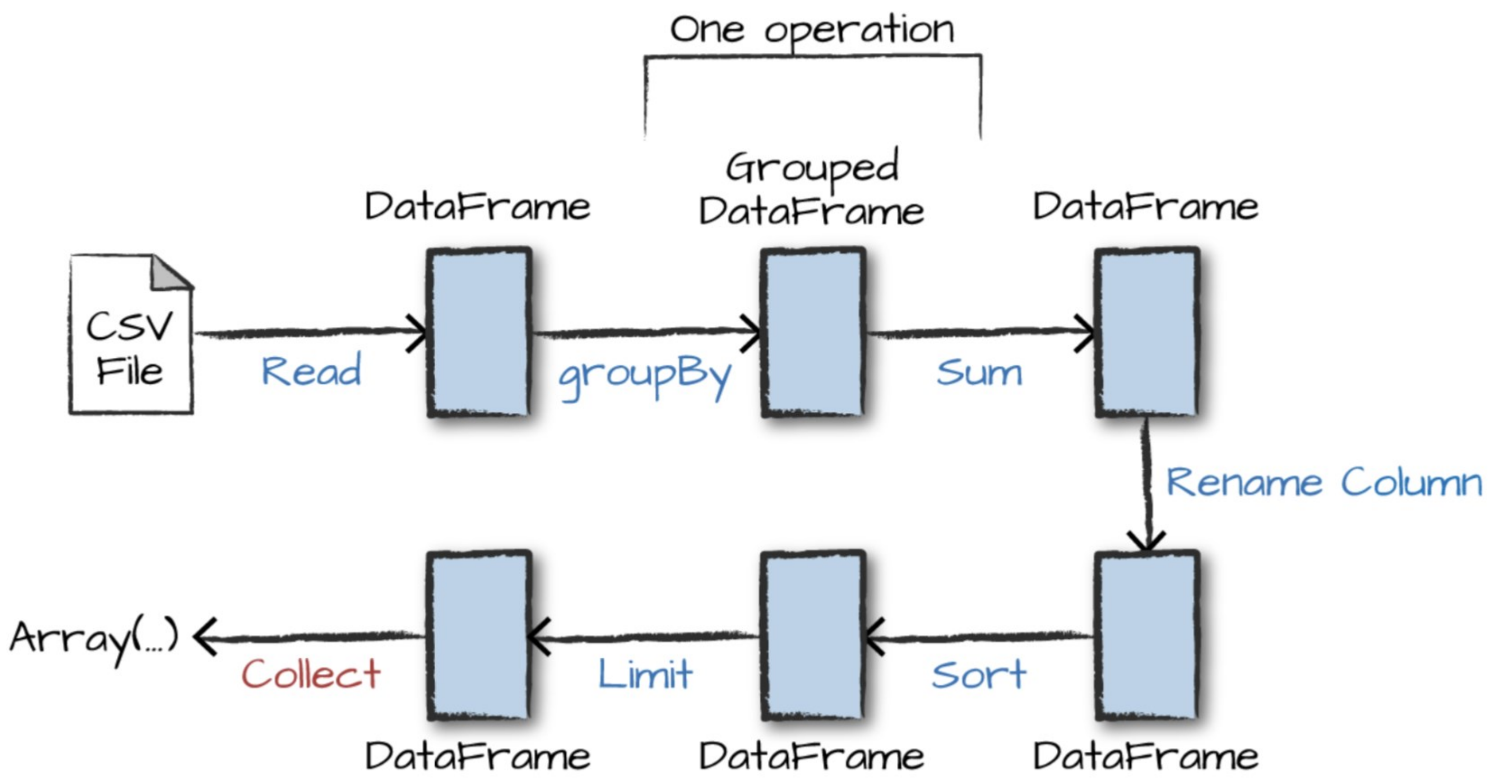


E.g. Hadoop YARN

Distributed vs. single-machine computing


Spreadsheet on a single machine
Table or Data Frame partitioned across servers in a data center

DAG of A Spark Application



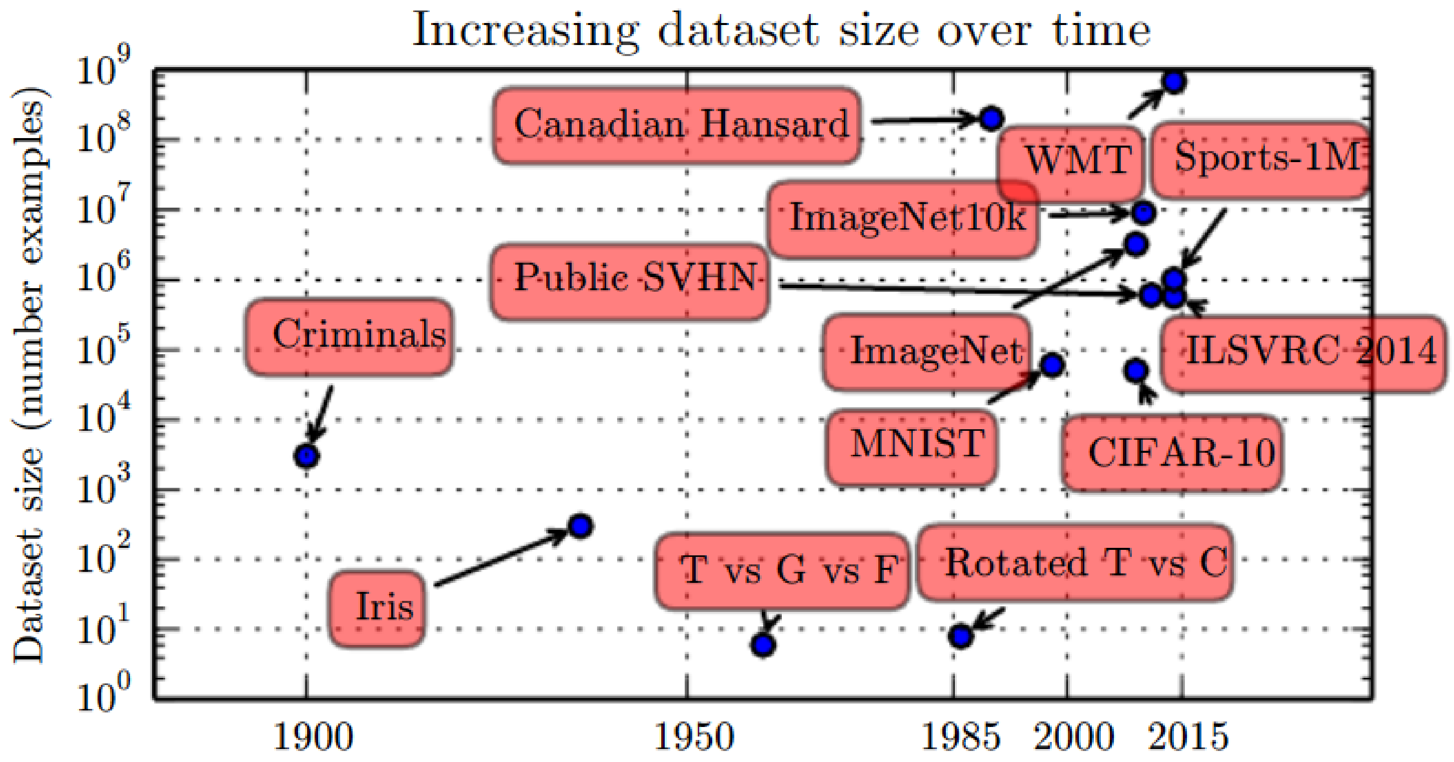
More flexible than the MapReduce model in developing a data analytics pipeline.

Handling Massive Datasets with Spark

- What if the dataset is too large to fit the machine's main memory?
  - If a single computer cannot do the job, use a cluster (e.g. a Hadoop cluster)
- ➢ Use Spark as a "heavy lifter"
  - SQL-like data query, ETL, etc.
  - clean raw datasets, exploratory analysis (e.g., statistics).
  - It makes use of the Spark's distributed computing capability
  - Spark provides a rich set of programming APIs
- ➢ Large-scale machine learning with Spark
  - Use the native MLlib Library in Spark

Large-Scale ML and Parallel Computation

Machine Learning Meets Big Data



UNIVERSITY OF WOLLONGONG AUSTRALIA

How can ML benefit from parallelism

- Common relational and algebraic operations can benefit from parallel computation (with MapReduce and Spark); *how about machine learning*?
- Recall what is ML
  - (Supervised learning) Given a set of observations $\boldsymbol{X}$ with labels y, learn a model $\mathcal{M}$ that predicts the labels of $\boldsymbol{X}$ as good as possible in items of some *cost function*
  - Most machine learning algorithms involve *Gradient-based Optimisation* and, in particular, *Gradient Descent*.

ML Example: Linear Regression

- Problem: minimise the MSE (mean square error) between the prediction value of linear regression and the target value
- Objective function:

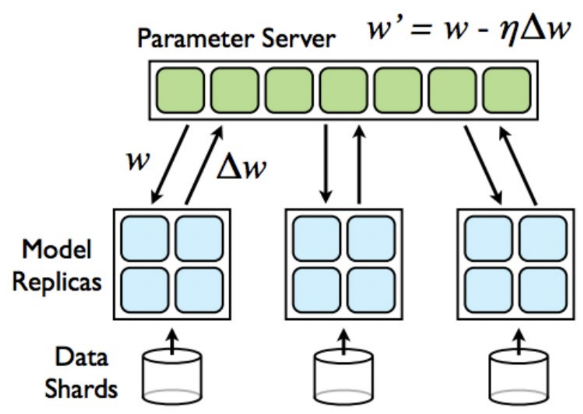
$$J(\boldsymbol{w}) = ||\boldsymbol{w}^T \boldsymbol{X} - \boldsymbol{y}||_2^2 = \sum_{j=1}^{m} (\boldsymbol{w}^T \boldsymbol{x}^{(j)} - y^{(j)})^2$$

- In order to minimise $J(\boldsymbol{w})$, use gradient descent (vector update):

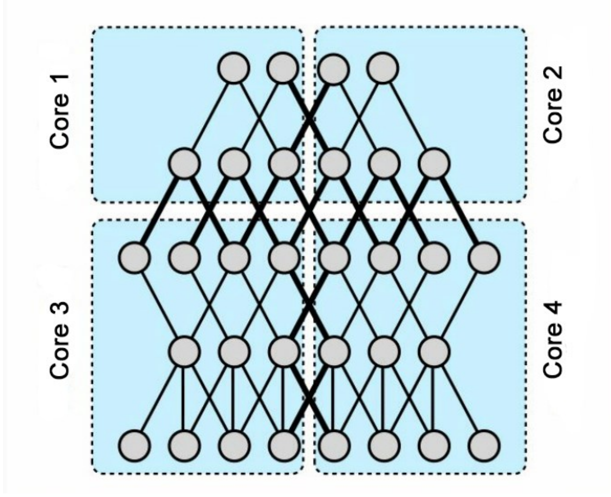
$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha_i \sum_{j=1}^{m} \left(\boldsymbol{w}_i^T \boldsymbol{x}^{(j)} - y^{(j)}\right) \boldsymbol{x}^{(j)}$$

Large-Scale ML

- Recall that ML updates a model $\mathcal{M}$ based on data $\boldsymbol{X}$
  - For linear regression (with the MSE cost function, without regularisation), the model is expressed by the vector $\boldsymbol{w}$.
- <u>Big data</u>: the input data $\boldsymbol{X}$ is too large to hold in the main memory.
- <u>Big model</u>: the model $\mathcal{M}$ is too large to hold in the main memory
- *Data parallelism* and *model parallelism*



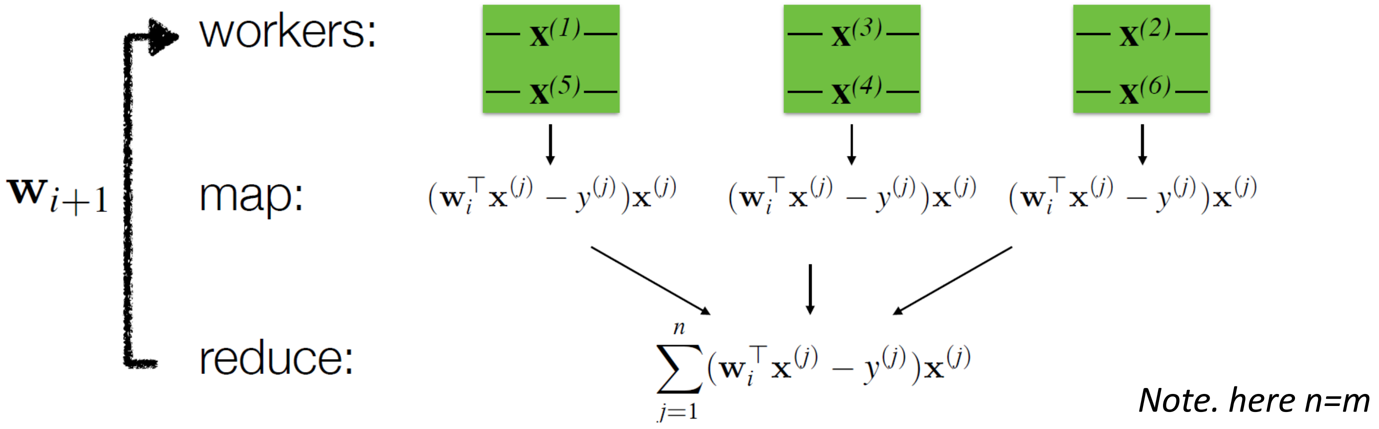
Data parallelism



Model parallelism

Gradient Descent: Big m, Big d

- Vector update: $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha_i \sum_{j=1}^{m} \left(\boldsymbol{w}_i^T \boldsymbol{x}^{(j)} - y^{(j)} \right) \boldsymbol{x}^{(j)}$
- By data parallelism, we compute summands in parallel on workers receiving all $\boldsymbol{w}_i$ at every iteration.
- For example, let $m = 6$ and the number of works is 3:


workers:
map:
reduce:
$\mathbf{w}_{i+1}$
$\mathbf{x}^{(1)}$
$\mathbf{x}^{(5)}$
$\mathbf{x}^{(3)}$
$\mathbf{x}^{(4)}$
$\mathbf{x}^{(2)}$
$\mathbf{x}^{(6)}$
$(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$
$(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$
$(\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$
$\sum_{j=1}^{n} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$
Note. here n=m

- Bottleneck: the transfer of $\boldsymbol{w}_i$ between the driver (or parameter server) and the workers.

Gradient Descent: Big m, Huge d

- In linear regression, the data size (i.e., $m \times d$) is much larger than the model size (i.e., d). But in other ML algorithms, the model size can be huge even compared with the data size.
  - As an example, consider a polynomial regression model with only two data records: $f(\boldsymbol{w}) = \sum_{i,j \le d} w_{i,j} x^i y^j$
  - In deep learning, the number of parameters can reach billions.
- In this case, model parallelism separates the models across different computing nodes and let different partitions of parameters be processed by different CPUs and GPUs.
- Divide and Conquer: fully update each partition locally and minimise the global communication.

Minibatch Gradient Descent

- Divide the data set into small batches of examples, compute the gradient using a single batch, makes an update, then move to the next batch of examples.
  - Computing the gradient simultaneously using the matrix-matrix multiplications which are efficient, especially on GPUs
  - Practice also shows that minibatch GD helps the algorithm jump out of local minima and find the global minima.
- E.g., $m = 10$,

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha_i \sum_{j=1}^{10} \left(\boldsymbol{w}_i^T \boldsymbol{x}^{(j)} - y^{(j)}\right) \boldsymbol{x}^{(j)}$$

Synchronous vs. Asynchronous GD

- In *synchronous* (minibatch) GD, the Spark Driver (or a parameter server) will wait until all parallel workers have returned their updated model before continuing to the next iteration.
  – Fast workers wait for slow workers
  – Can guarantee fault-tolerance
- In (parallel) *asynchronous* GD (or minibatch GD), a parameter server will apply model updates from parallel workers immediately, whereupon the work can immediately get new copy of the model to work on a new mini-batch.
  – Workers train the model concurrently on mini-batches without blocking
  – Needs to work with stale gradients

Gradient-Based Optimisation in Spark

- The architecture of Spark is well-suited to *data parallelism* and *synchronise* model update (including gradient-based model update).
  - Model parallelism and asynchronies update are not really supported in a standard Spark architecture
  - Also, the model must fit in the Spark Driver's memory.
  - A parameter server can handle large models and asynchronies update (currently a hot research topic)

❖ References:

- ❖ Gradient-based optimisation methods implemented in Spark ML: https://spark.apache.org/docs/latest/mllib-optimization.html
- ❖ Spark also use Gradient-based methods (e.g., LBFGS) from the Scala scientific computing library Breeze: https://github.com/scalanlp/breeze/blob/master/math/src/main/scala/breeze/optimize/LBFGS.scala

Spark MLlib

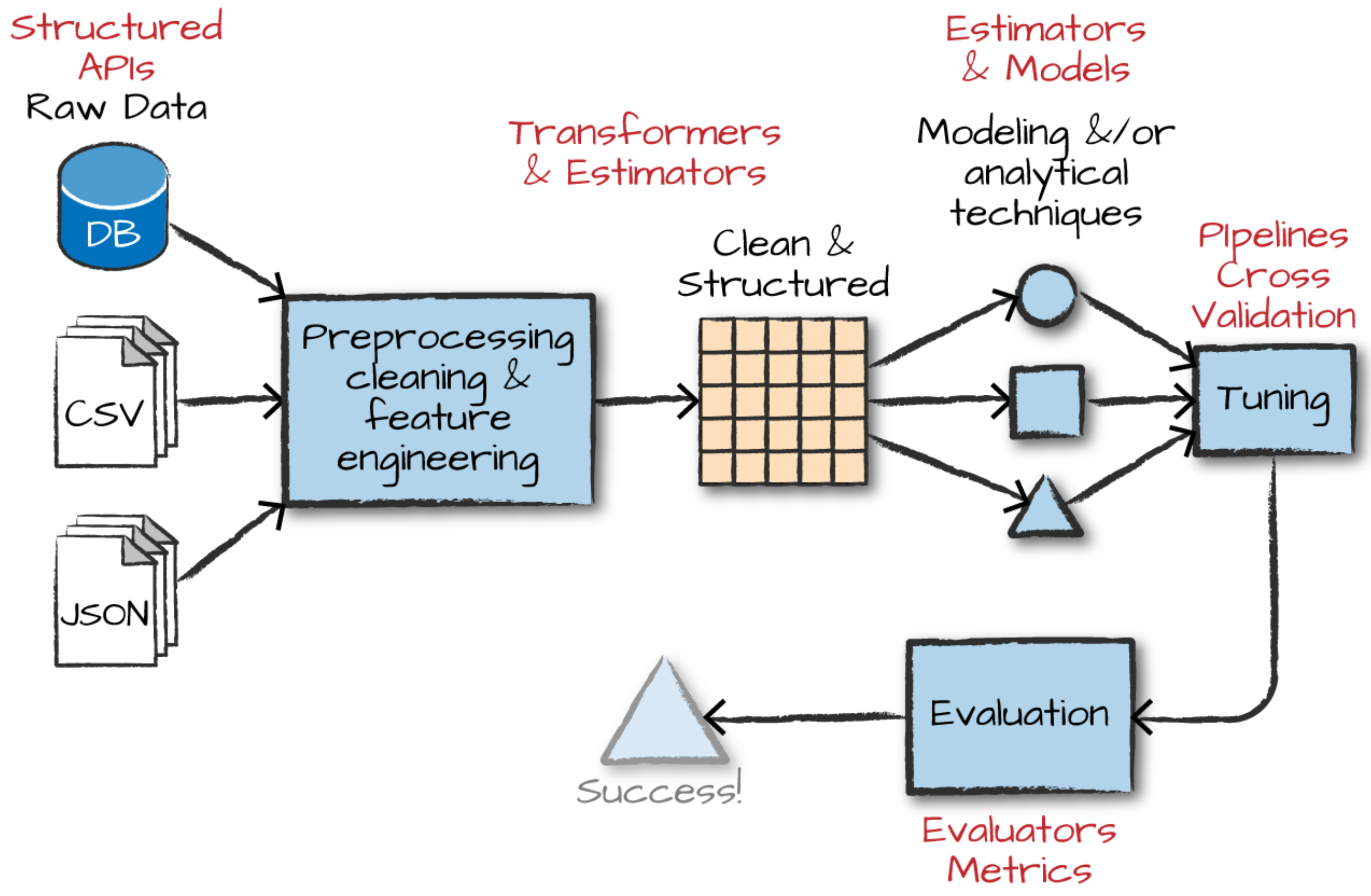
—Spark’s machine learning library, which leverages the parallel computation power of Spark to implement large-scale machine learn project

The Process of End-to-End ML Project

- The key steps in the process of an end-to-end project:
    1. Gathering and collecting the relevant data for your task.
    2. Cleaning and inspecting the data to better understand it.
    3. Performing feature engineering to allow the algorithm to leverage the data in a suitable form (e.g., converting the data to numerical vectors).
    4. Using a portion of this data as a training set to train one or more algorithms to generate some candidate models.
    5. Evaluating and comparing models by objectively measuring results on a subset of the same data that was not used for training.
    6. Leveraging the insights from the above process and/or using the model to make predictions, detect anomalies, or solve more general business challenges.

High-Level MLlib Concepts

- Spark supports most steps in the e2e ML project


Structured APIs
Raw Data
DB
CSV
JSON
Transformers & Estimators
Preprocessing cleaning & feature engineering
Clean & Structured
Estimators & Models
Modeling &/or analytical techniques
Pipelines Cross Validation
Tuning
Evaluation
Evaluators Metrics
Success!

Question?

UNIVERSITY
OF WOLLONGONG
AUSTRALIA