Why use Apache Spark?

Apache Spark attributes

all operation happen within the memory or RAM

Spark is an open source in-memory application framework for distributed data processing and iterative analysis on massive data volumes

Apache Spark Attributes

support object-oriented and functional programming

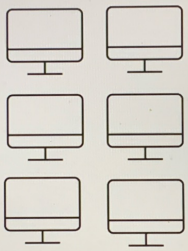
general purpose programming language

Spark is written predominantly in Scala and runs on Java virtual machines (JVMs)

What is Distributed Computing?

- A group, or cluster, of computers working together to appear as one system to the end user

behind the scences



What is Distributed Computing?

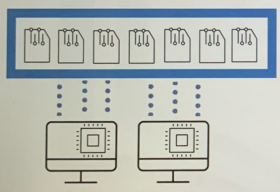
- The term distributed computing often used interchangeably with parallel computing as both are similar.

Distributed computing **and** Parallel computing **= or ≠ ?**

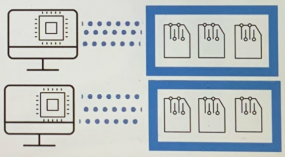
Parallel versus Distributed Computing

They are access memory differently

- Parallel computing processors access shared memory



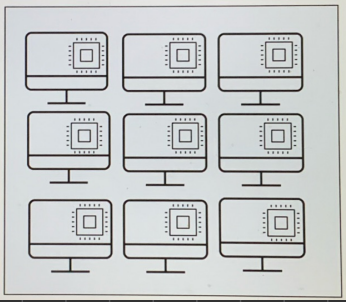
- Distributed computing processors usually have their own private or distributed memory



Distributed computing benefits

- Scalability and modular growth

a user can add additional machines to handle the increasing workload instead of repeatedly updating a single system



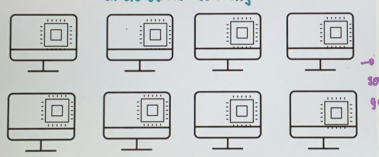
Distributed computing benefits

- Fault tolerance and redundancy

require

enable business continuity

as they use independent nodes that could fails

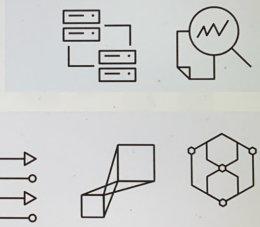


can run although some machines go offline

Your Data Center

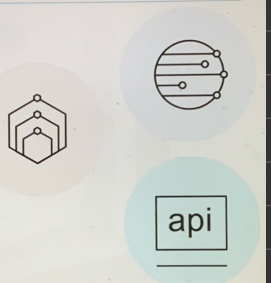
Spark Benefits

- Supports a computing framework for large scale data processing and analysis
- Provides parallel and distributed processing, scalability and fault tolerance on commodity hardware



Apache Spark Benefits

- Provides speed due to in-memory processing
- Creates a comprehensive, unified framework to manage big data processing
- Enables programming flexibility with easy-to-use Python, Scala, and Java APIs



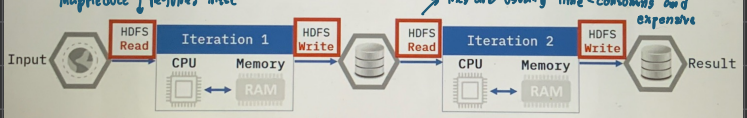
Apache Spark & MapReduce Compared

Traditional Approach:

- Create MapReduce jobs for complex jobs, interactive query, and online event-hub processing involves lots of (slow) disk I/O

MapReduce writes to HDFS

they are usually time-consuming and expensive

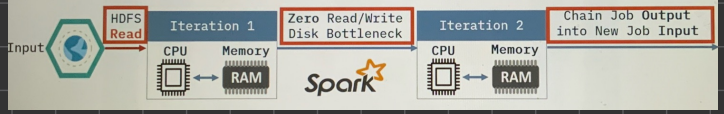


Apache Spark & MapReduce Compared

Solution:

- Keep more data in-memory with a new distributed execution engine

and avoiding expensive I/O, thus reducing the overall time by orders of magnitude



Spark and Big Data

Data engineering

- Core Spark engine
- Clusters and executors
- Cluster management
- SparkSQL
- Catalyst
  Tungsten DataFrames

Data science and Machine learning

- SparkML
- DataFrames
- Streaming

Functional Programming Basics

Objectives

After watching this video, you will be able to:

- Explain the term "functional programming"
- Explain Lambda functions
- Relate functional programming with Apache Spark

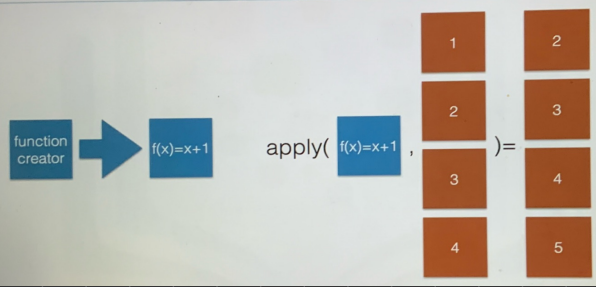
What is functional programming

- First implementation was LISt Programming Language (LISP) in the 1950s
- Scala most recent representative (Python, R and Java also provide rudimentary support for functional programming)
- Functions are first class citizens in Scala

What is functional programming

- A mathematical "function" programming style
- Follows a declarative programming model
- Emphasizes "What" instead of "How-to"
- Uses "expressions" instead of "statements"

A functional programming example

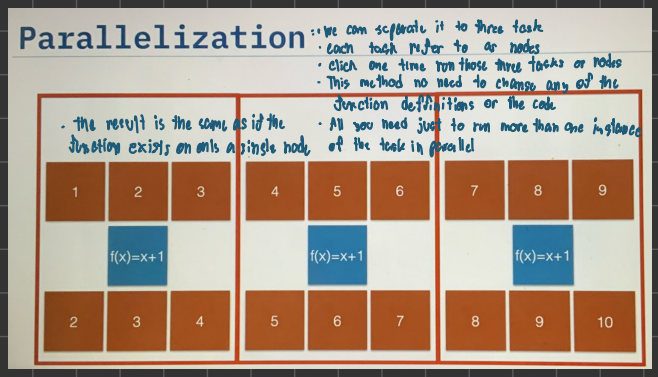


A traditional procedural example

A more traditional procedural way to achieve the same result:

```
// A Python function to add 1 to each element of a List
def increment(myList):
      N = size(myList)
      for i in range(N): #go over each element
            myList[i] += 1 #increment each element
      return myList
```

Parallelization



What are lambda functions?

- Lambda calculus functions, or operators, are anonymous functions that enable functional programming

Scala

```
// an example lambda operator in Scala to add 2 numbers
val add = (x:Int, y:Int) => x + y
println(add(1,2))
```

Python

```
// an example lambda function in Python to add 2 numbers
add = lambda x, y : x + y
print(add(1,2))
```

Lambda functions and Spark

- Spark parallelizes computations using the lambda calculus
- All functional Spark programs are inherently parallelizable

Parallel Programming using Resilient Distributed Datasets

Objectives

After watching this video, you will be able to:

- Define Resilient Distributed Datasets (RDDs)
- Define Parallel Programming
- Explain Resilience in Apache Spark
- Relate RDD and Parallel Programming with Apache Spark

What are RDDs

A resilient distributed dataset is:

- Spark's primary data abstraction
- A fault-tolerant collection of elements
- Partitioned across the nodes of the cluster
- Capable of accepting parallel operations
- Immutable : cannot change once created

Spark applications

Consist of a driver program that runs
The user's main functions
And multiple *parallel operations*
on a cluster.

RDD supported files

Supported File types

- Text
- SequenceFiles
- Avro
- Parquet
- Hadoop input formats

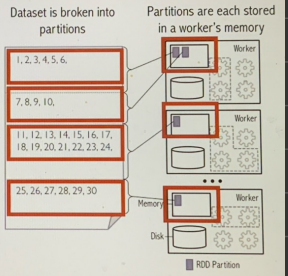
Supported File formats

- Local
- Cassandra
- HBase
- HDFS
- Amazon S3
- and others
- SQL and NoSQL

Creating an RDD in Spark

Use an external or local file from Hadoop-supported file system such as:

- HDFS
- Cassandra
- Hbase
- Amazon S3



Create an RDD from a collection

↓ apply to the parallelize function
↳ supported high level APIs, including JAVA, PYTHON and Scala

Simple examples of creating a RDD from a list in Scala and Python

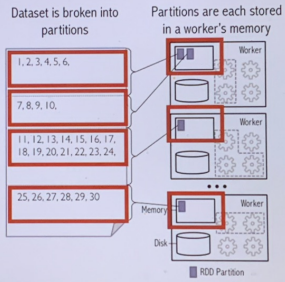
- one important for parallel collections is the number of partitions specified to cut the dataset
- Spark run one task for each partition of the cluster
- want 2 to 4 partitions for each CPU
- Spark tries to automatically set the number of partitions based on cluster
- Can set partition manually by passing the number as second parameter to the parallelize function

```
// Scala example
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

```
// Python example
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Creating an RDD in Spark

Apply a transformation on an existing RDD to create a new RDD

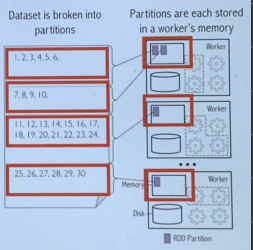


What is Parallel Programming

- Parallel programming:
- Is the simultaneous use of multiple compute resources to solve a computational problem
- Breaks problems into discrete parts that can be solved concurrently
- Runs simultaneous instructions on multiple processors
- Employs an overall control/coordination mechanism

RDDs & Parallel Programming

- You can create an RDD by parallelizing an array of objects, or by splitting a dataset into partitions
- Spark runs one task for each partition of the cluster
  based on how RDDs are created



Resilience and Spark

RDD's provide resilience in Spark through immutability and caching

Resilient Distributed Datasets

- Are always recoverable as they are immutable
- Can persist or cache datasets in memory across operations, which speeds iterative operations

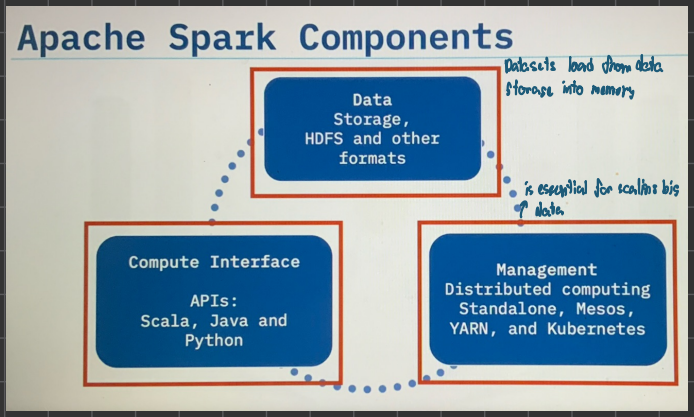
Scale out / Data Parallelism in Apache Spark

Objectives

After watching this video, you will be able to:

- Describe Apache Spark components
- Describe how Apache Spark scales with big data

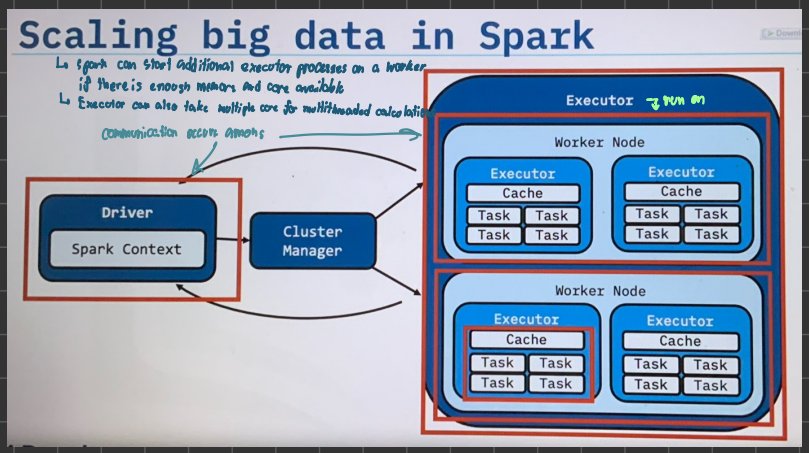
Apache Spark Components



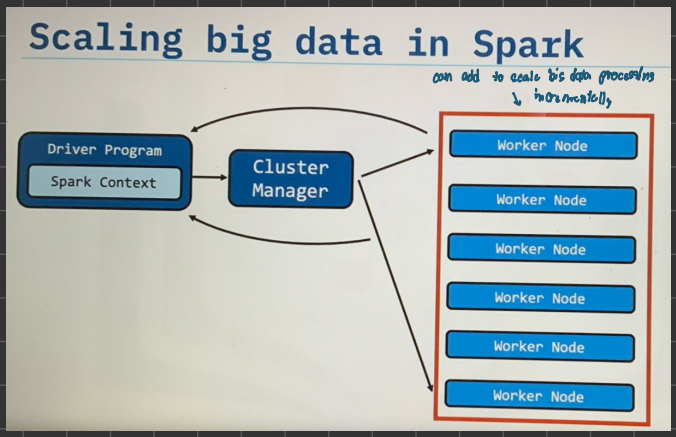
Spark Core

- Is a base engine
- Is fault-tolerant
- Performs large scale parallel and distributed data processing
- Manages memory
- Schedules tasks
- Houses APIs that define RDDs
- Contains a distributed collection of elements that are parallelized across the cluster

Scaling big data in Spark



Scaling big data in Spark



Spark Application Architecture

DataFrames and SparkSQL

Objectives

After watching this video, you will be able to:

- Describe what Spark SQL is, define the parts of a Spark SQL query, and explain benefits of using Spark SQL
- Describe what DataFrames are, define the parts of a DataFrame query and explain the benefits of using a DataFrame

Spark SQL

- Is a Spark module for structured data processing
- Used to query structured data inside Spark programs, using either SQL or a familiar DataFrame API
- Usable in Java, Scala, Python and R
- Runs SQL queries over imported data and existing RDDs independently of API or programming language

Spark SQL example

Spark SQL query using Python

```
results = spark.sql(
  "SELECT * FROM people")
names = results.map(lambda p:
p.name)
```

Spark SQL - Benefits

- Includes a cost-based optimizer, columnar storage, and code generation to make queries fast
- Scales to thousands of nodes and multi-hour queries using the Spark engine, which provides full mid-query fault tolerance
- Provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine

DataFrames

- Distributed collection of data organized into named columns
- Conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations
- Built on top of the RDD API
- Uses RDDs
- Performs relational queries

DataFrame

Python code snippet to read from a JSON file and create a simple DataFrame.

```
df = spark.read.json("people.json")
df.show()
df.printSchema()

# Register the DataFrame as a SQL temporary view
df.createTempView("people")
```

DataFrame example

Input JSON file

```
{"name":"Michael"}
{"name":"Andy",
"age":30}
{"name":"Justin",
"age":19}
```

Created DataFrame

```
+----+-------+
| age| name  |
+----+-------+
|null|Michael|
|  30|  Andy |
|  19| Justin |
+----+-------+
```

DataFrame Benefits

- 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, which is in development via Spark R

DataFrame + Spark SQL

SQL Query

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

DataFrame Python API

```
df.select("name").show()
df.select(df["name"]).show()
```

Result

```
+-------+
| name  |
+-------+
|Michael|
|  Andy |
| Justin|
+-------+
```

DataFrame + Spark SQL

SQL Query

```
spark.sql("SELECT age, name
FROM people WHERE age >
21").show()
```

DataFrame Python API

```
df.filter(df["age"]>21).show()
```

Result

```
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
```