**Python Interview Questions**

**Q1. What is the difference between list and tuples in Python?**

|  |  |
| --- | --- |
| **LIST vs TUPLES** | |
| **LIST** | **TUPLES** |
| Lists are mutable i.e they can be edited. | Tuples are immutable (tuples are lists which can’t be edited). |
| Lists are slower than tuples. | Tuples are faster than list. |
| Syntax: list\_1 = [10, ‘Chelsea’, 20] | Syntax: tup\_1 = (10, ‘Chelsea’ , 20) |

my\_tuple = ('sara', 6, 5, 0.97)

my\_list = ['sara', 6, 5, 0.97]

**print**(my\_tuple[0]) *# output => 'sara'*

**print**(my\_list[0]) *# output => 'sara'*

my\_tuple[0] = 'ansh' *# modifying tuple => throws an error*

my\_list[0] = 'ansh' *# modifying list => list modified*

**print**(my\_tuple[0]) *# output => 'sara'*

**print**(my\_list[0]) *# output => 'ansh'*

**Q: What are the built-in types provided by the Python?**

**Answer:**

Mutable built-in types:

* Lists
* Sets
* Dictionaries

Immutable built-in types:

* Strings
* Tuples
* Numbers

**Q2. What are the key features of Python?**

* Python is an **interpreted** language. That means that, unlike languages like *C* and its variants, Python does not need to be compiled before it is run. Other interpreted languages include *PHP* and *Ruby*.
* Python is **dynamically typed**, this means that you don’t need to state the types of variables when you declare them or anything like that. You can do things like x=111 and then x="I'm a string" without error
* Python is well suited to [**object orientated programming**](https://www.edureka.co/blog/python-class/) in that it allows the definition of classes along with composition and inheritance. Python does not have access specifiers (like C++’s public, private).
* In Python, [**functions**](https://www.edureka.co/blog/python-functions) are **first-class objects**. This means that they can be assigned to variables, returned from other functions and passed into functions. Classes are also first class objects
* **Writing Python code is quick** but running it is often slower than compiled languages. Fortunately，Python allows the inclusion of C-based extensions so bottlenecks can be optimized away and often are. The [numpy](https://www.edureka.co/blog/python-numpy-tutorial/) package is a good example of this, it’s really quite quick because a lot of the number-crunching it does isn’t actually done by Python
* Python finds **use in many spheres** – web applications, automation, scientific modeling, big data applications and many more. It’s also often used as “glue” code to get other languages and components to play nice.

**Q74.What is a dynamically typed language?**

Before we understand what a dynamically typed language, we should learn about what typing is. **Typing** refers to type-checking in programming languages. In a ***strongly-typed***  language, such as Python, **"1" + 2** will result in a type error, since these languages don't allow for **"type-coercion"** (implicit conversion of data types). On the other hand, a ***weakly-typed***  language, such as Javascript, will simply output **"12"** as result.

Type-checking can be done at two stages -

1. **Static -** Data Types are checked before execution.
2. **Dynamic -** Data Types are checked during execution.

Python being an interpreted language, executes each statement line by line and thus type-checking is done on the fly, during execution. Hence, Python is a Dynamically Typed language.

**Q3. What type of language is python? Programming or scripting?**

***Ans:*** Python is capable of scripting, but in general sense, it is considered as a general-purpose programming language. To know more about Scripting, you can refer to the [Python Scripting Tutorial](https://youtu.be/9F6zAuYtuFw).

**Q4.Python an interpreted language. Explain.**

***Ans:*** An interpreted language is any programming language which is not in machine-level code before runtime. Therefore, Python is an interpreted language.

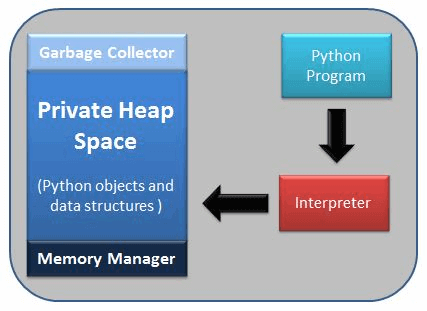
**Q5.What is pep 8?**

***Ans:*** PEP stands for **Python Enhancement Proposal.** It is a set of rules that specify how to format Python code for maximum readability.

**Q6. How is memory managed in Python?**

**Ans:** Memory is managed in Python in the following ways:

1. Memory management in python is managed by ***Python private heap space***. All Python objects and data structures are located in a private heap. The programmer does not have access to this private heap. The python interpreter takes care of this instead.
2. The allocation of heap space for Python objects is done by Python’s memory manager. The core API gives access to some tools for the programmer to code.
3. Python also has an inbuilt garbage collector, which recycles all the unused memory and so that it can be made available to the heap space.
4. Additionally, Python has an in-built garbage collection to recycle the unused memory for the private heap space.



**Q.What is the difference between .py and .pyc files?**

* .py files contain the source code of a program. Whereas, .pyc file contains the bytecode of your program. We get bytecode after compilation of .py file (source code). .pyc files are not created for all the files that you run. It is only created for the files that you import.
* Before executing a python program python interpreter checks for the compiled files. If the file is present, the virtual machine executes it. If not found, it checks for .py file. If found, compiles it to .pyc file and then python virtual machine executes it.
* Having .pyc file saves you the compilation time.

**Q7. What is namespace in Python? Why are they used?**

***Ans:*** A namespace is a naming system used to make sure that names are unique to avoid naming conflicts.

A namespace in Python ensures that object names in a program are unique and can be used without any conflict. Python implements these **namespaces as dictionaries** with 'name as key' mapped to a corresponding 'object as value'. This allows for multiple namespaces to use the same name and map it to a separate object. A few examples of namespaces are as follows:

* **Local Namespace** includes local names inside a function. the namespace is temporarily created for a function call and gets cleared when the function returns.
* **Global Namespace** includes names from various imported packages/ modules that is being used in the current project. This namespace is created when the package is imported in the script and lasts until the execution of the script.
* **Built-in Namespace** includes built-in functions of core Python and built-in names for various types of exceptions.

**Lifecycle of a namespace** depends upon the scope of objects they are mapped to. If the scope of an object ends, the lifecycle of that namespace comes to an end. Hence, it isn't possible to access inner namespace objects from an outer namespace.



**Q8. What is PYTHONPATH?**

***Ans:*** It is an environment variable which is used when a module is imported. Whenever a module is imported, PYTHONPATH is also looked up to check for the presence of the imported modules in various directories. The interpreter uses it to determine which module to load.

**Q9. What are python modules? Name some commonly used built-in modules in Python?**

***Ans:*** Python modules are files containing Python code. This code can either be functions classes or variables. A Python module is a .py file containing executable code.

Some of the commonly used built-in modules are:

* os
* sys
* math
* random
* data time
* JSON

**Q10.What are local variables and global variables in Python?**

**Global Variables:**

Variables declared outside a function or in global space are called global variables. These variables can be accessed by any function in the program.

**Local Variables:**

Any variable declared inside a function is known as a local variable. This variable is present in the local space and not in the global space.

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | a=2  def add():  b=3  c=a+b  print(c)  add() |

**Output:**5

When you try to access the local variable outside the function add(), it will throw an error.

**Q11. Is python case sensitive?**

***Ans:*** Yes. Python is a case sensitive language.

**Q12.What is type conversion in Python?**

***Ans:*** Type conversion refers to the conversion of one data type iinto another.

**int()** – converts any data type into integer type

**float()** – converts any data type into float type

**ord()** – converts characters into integer

**hex(**) – converts integers to hexadecimal

**oct()** – converts integer to octal

**tuple() –** This function is used to convert to a tuple.

**set() –** This function returns the type after converting to set.

**list() –**This function is used to convert any data type to a list type.

**dict() –**This function is used to convert a tuple of order (key,value) into a dictionary.

**str() –**Used to convert integer into a string.

**complex(real,imag) –** This functionconverts real numbers to complex(real,imag) number.

**Q13. How to install Python on Windows and set path variable?**

***Ans:*** To install Python on Windows, follow the below steps:

* Install python from this link: <https://www.python.org/downloads/>
* After this, install it on your PC. Look for the location where PYTHON has been installed on your PC using the following command on your command prompt: cmd python.
* Then go to advanced system settings and add a new variable and name it as PYTHON\_NAME and paste the copied path.
* Look for the path variable, select its value and select ‘edit’.
* Add a semicolon towards the end of the value if it’s not present and then type %PYTHON\_HOME%

**Q14. Is indentation required in python?**

***Ans:*** Indentation is necessary for Python. It specifies a block of code. All code within loops, classes, functions, etc is specified within an indented block. It is usually done using four space characters. If your code is not indented necessarily, it will not execute accurately and will throw errors as well.

**Q15. What is the difference between Python Arrays and lists?**

***Ans:*** Arrays and lists, in Python, have the same way of storing data. But, arrays can hold only a single data type elements whereas lists can hold any data type elements.

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5 | import array as arr  My\_Array=arr.array('i',[1,2,3,4])  My\_list=[1,'abc',1.20]  print(My\_Array)  print(My\_list) |

**Output:**

array(‘i’, [1, 2, 3, 4]) [1, ‘abc’, 1.2]

**Q.What is slicing in Python?**

* As the name suggests, ‘slicing’ is taking parts of.
* Syntax for slicing is **[start : stop : step]**
* **start** is the starting index from where to slice a list or tuple
* **stop** is the ending index or where to sop.
* **step** is the number of steps to jump.
* Default value for **start** is 0, **stop** is number of items, **step** is 1.
* Slicing can be done on **strings**, **arrays**, **lists**, and **tuples**.

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

**print**(numbers[1 : : 2])  *#output : [2, 4, 6, 8, 10]*

### Q.What are Dict and List comprehensions?

Python comprehensions, like decorators, are **syntactic sugar** constructs that help **build altered and filtered** lists, dictionaries or sets from a given list, dictionary or set. Using comprehensions, saves a lot of time and code that might be considerably more verbose (containing more lines of code). Let's check out some examples, where comprehensions can be truly beneficial:

**Performing mathematical operations on the entire list**

my\_list = [2, 3, 5, 7, 11]

squared\_list = [x\*\*2 **for** x **in** my\_list] *# list comprehension*

*# output => [4 , 9 , 25 , 49 , 121]*

squared\_dict = {x:x\*\*2 **for** x **in** my\_list} *# dict comprehension*

*# output => {11: 121, 2: 4 , 3: 9 , 5: 25 , 7: 49}*

**Performing conditional filtering operations on the entire list**

my\_list = [2, 3, 5, 7, 11]

squared\_list = [x\*\*2 **for** x **in** my\_list **if** x%2 != 0] *# list comprehension*

*# output => [9 , 25 , 49 , 121]*

squared\_dict = {x:x\*\*2 **for** x **in** my\_list **if** x%2 != 0} *# dict comprehension*

*# output => {11: 121, 3: 9 , 5: 25 , 7: 49}*

**Combining multiple lists into one**   
Comprehensions allow for multiple iterators and hence, can be used to combine multiple lists into one.

a = [1, 2, 3]

b = [7, 8, 9]

[(x + y) **for** (x,y) **in** **zip**(a,b)] *# parallel iterators*

*# output => [8, 10, 12]*

[(x,y) **for** x **in** a **for** y **in** b] *# nested iterators*

*# output => [(1, 7), (1, 8), (1, 9), (2, 7), (2, 8), (2, 9), (3, 7), (3, 8), (3, 9)]*

**Flattening a multi-dimensional list**   
A similar approach of nested iterators (as above) can be applied to flatten a multi-dimensional list or work upon its inner elements.

my\_list = [[10,20,30],[40,50,60],[70,80,90]]

flattened = [x **for** temp **in** my\_list **for** x **in** temp]

*# output => [10, 20, 30, 40, 50, 60, 70, 80, 90]*

***Note:*** *List comprehensions have the same effect as the map method in other languages. They follow the* ***mathematical set builder notation*** *rather than map and filter functions in Python.*

### Q. What are the common built-in data types in Python?

There are several built-in data types in Python. Although, Python doesn't require data types to be defined explicitly during variable declarations but type errors are likely to occur if the knowledge of data types and their compatibility with each other are neglected. Python provides type() and isinstance() functions to check the type of these variables. These data types can be grouped into the following catetgories-

* **None Type**   
  None keyword represents the null values in Python. Boolean equality operation can be performed using these NoneType objects.

|  |  |
| --- | --- |
| **Class Name** | **Description** |
| NoneType | Represents the **NULL** values in Python |

* **Numeric Types**   
  There are three distint numeric types - ***integers***, ***floating-point numbers***, and ***complex numbers***. Additionally, ***booleans*** are a sub-type of integers.

|  |  |
| --- | --- |
| **Class Name** | **Description** |
| int | Stores integer literals including hex, octal and binary numbers as integers |
| float | Stores literals containing decimal values and/or exponent sign as floating-point numbers |
| complex | Stores complex number in the form (A + Bj) and has attributes: real and imag |
| bool | Stores boolean value (True or False) |

* ***Note:*** *The standard library also includes* ***fractions*** *to store rational numbers and* ***decimal*** *to store floating-point numbers with user-defined precision.*
* Sequence Types   
  According to Python Docs, there are three basic Sequence Types - ***lists***, ***tuples***, and ***range objects***. Sequence types have the in and not in operators defined for their traversing their elements. These operators share the same priority as the comparison operations.

|  |  |
| --- | --- |
| **Class Name** | **Description** |
| list | Mutable sequence used to store collection of items. |
| tuple | Immutable sequence used to store collection of items. |
| range | Represents an immutable sequence of numbers generated during execution. |
| str | Immutable sequence of Unicode code points to store textual data. |

* ***Note:*** *The standard library also includes additional types for processing:   
  1.* ***Binary data*** *such as bytearray bytes memoryview , and   
  2.* ***Text strings*** *such as str .*
* Mapping Types   
  A mapping object can map *hashable values* to random objects in Python. Mappings objects are mutable and there is currently only one standard mapping type, the ***dictionary***.

|  |  |
| --- | --- |
| **Class Name** | **Description** |
| dict | Stores comma-separated list of **key: value** pairs |

* Set Types   
  Currently, Python has two built-in set types - **set** and **frozenset**. **set** type is mutable and supports methods like add() and remove(). **frozenset** type is immutable and can't be modified after creation.
* ***mbNote:*** *set is mutable and thus cannot be used as key for a dictionary. On the other hand, frozenset is immutable and thus, hashable, and can be used as a dictionary key or as an element of another set.*
* Modules   
  Module is an additional built-in type supported by the Python Interpreter. It supports one special operation, i.e., **attribute access**: mymod.myobj, where **mymod** is a module and **myobj** references a name defined in m's symbol table. The module's symbol table resides in a very special attribute of the module **\_\_dict\_\_**, but direct assignment to this module is neither possible nor recommended.
* Callable Types   
  Callable types are the types to which function call can be applied. They can be **user-defined functions**, **instance methods**, **generator functions**, and some other **built-in functions**, **methods** and **classes**.   
  Refer the documentation at [docs.python.org](https://docs.python.org/3/reference/datamodel.html) for a detailed view into the **callable types**.

**Q16. What are functions in Python?**

***Ans:*** A function is a block of code which is executed only when it is called. To define a [Python function](https://www.edureka.co/blog/python-functions), the **def** keyword is used.

**Example:**

|  |  |
| --- | --- |
| 1  2  3 | def Newfunc():  print("Hi, Welcome to Edureka")  Newfunc(); #calling the function |

**Output:** Hi, Welcome to Edureka

**Q17.What is \_\_init\_\_?**

***Ans:*** \_\_init\_\_ is a method or constructor in [Python](https://www.edureka.co/blog/python-programming-language). This method is automatically called to allocate memory when a new object/ instance of a class is created. All classes have the \_\_init\_\_ method.

Here is an example of how to use it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Employee:  def \_\_init\_\_(self, name, age,salary):  self.name = name  self.age = age  self.salary = 20000  E1 = Employee("XYZ", 23, 20000)  # E1 is the instance of class Employee.  #\_\_init\_\_ allocates memory for E1.  print(E1.name)  print(E1.age)  print(E1.salary) |

**Output:**

XYZ

23

20000

**Q18.What is a lambda function?**

Lambda is an anonymous function in Python, that can accept any number of arguments, but can only have a single expression. It is generally used in situations requiring an anonymous function for a short time period. Lambda functions can be used in either of the two ways:

Assigning lambda functions to a variable

mul = **lambda** a, b : a \* b

**print**(**mul**(2, 5)) *# output => 10*

Wrapping lambda functions inside another function

**def** **myWrapper**(n):

**return** **lambda** a : a \* n

mulFive = **myWrapper**(5)

**print**(**mulFive**(2)) *# output => 10*

**Output:**11

### ****Q.What do you understand by lambda function? Create a lambda function which will print the sum of all the elements in this list -> [5, 8, 10, 20, 50, 100]****

from functools import reduce

sequences = [5, 8, 10, 20, 50, 100]

sum = reduce (lambda x, y: x+y, sequences)

print(sum)

**Q19. What is self in Python?**

***Ans:*** Self is an instance or an object of a class. In Python, this is explicitly included as the first parameter. However, this is not the case in Java where it’s optional.  It helps to differentiate between the methods and attributes of a class with local variables.

The self variable in the init method refers to the newly created object while in other methods, it refers to the object whose method was called.

**Q20. How does break, continue and pass work?**

|  |  |
| --- | --- |
| Break | Allows loop termination when some condition is met and the control is transferred to the next statement. |
| Continue | Allows skipping some part of a loop when some specific condition is met and the control is transferred to the beginning of the loop |
| Pass | Used when you need some block of code syntactically, but you want to skip its execution. This is basically a null operation. Nothing happens when this is executed. |

pat = [1, 3, 2, 1, 2, 3, 1, 0, 1, 3]

**for** p **in** pat:

**pass**

**if** (p == 0):

current = p

**break**

**elif** (p % 2 == 0):

**continue**

**print**(p) *# output => 1 3 1 3 1*

**print**(current) *# output => 0*

**Q21. What does [::-1} do?**

***Ans:*** [::-1] is used to reverse the order of an array or a sequence.

*For example:*

|  |  |
| --- | --- |
| 1  2  3 | import array as arr  My\_Array=arr.array('i',[1,2,3,4,5])  My\_Array[::-1] |

**Output**: array(‘i’, [5, 4, 3, 2, 1])

[::-1] reprints a reversed copy of ordered data structures such as an array or a list. the original array or list remains unchanged.

**Q22. How can you randomize the items of a list in place in Python?**

**Ans:** Consider the example shown below:

|  |  |
| --- | --- |
| 1  2  3  4 | from random import shuffle  x = ['Keep', 'The', 'Blue', 'Flag', 'Flying', 'High']  shuffle(x)  print(x) |

The output of the following code is as below.

['Flying', 'Keep', 'Blue', 'High', 'The', 'Flag']

**Q23. What are python iterators?**

***Ans:*** Iterators are objects which can be traversed though or iterated upon.

Iterator in python is **an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets**. The iterator object is initialized using the iter() method. It uses the next() method for iteration

**Q24. How can you generate random numbers in Python?**

**Ans:** Random module is the standard module that is used to generate a random number. The method is defined as:

|  |  |
| --- | --- |
| 1  2 | import random  random.random |

The statement random.random() method return the floating point number that is in the range of [0, 1). The function generates random float numbers. The methods that are used with the random class are the bound methods of the hidden instances. The instances of the Random can be done to show the multi-threading programs that creates a different instance of individual threads. The other random generators that are used in this are:

1. randrange(a, b): it chooses an integer and define the range in-between [a, b). It returns the elements by selecting it randomly from the range that is specified. It doesn’t build a range object.
2. uniform(a, b): it chooses a floating point number that is defined in the range of [a,b).Iyt returns the floating point number
3. normalvariate(mean, sdev): it is used for the normal distribution where the mu is a mean and the sdev is a sigma that is used for standard deviation.
4. The Random class that is used and instantiated creates independent multiple random number generators.

**Q25. What is the difference between range & xrange?**

**Ans:** For the most part, xrange and range are the exact same in terms of functionality. They both provide a way to generate a list of integers for you to use, however you please. The only difference is that range returns a Python list object and x range returns an xrange object.

This means that xrange doesn’t actually generate a static list at run-time like range does. It creates the values as you need them with a special technique called yielding. This technique is used with a type of object known as generators. That means that if you have a really gigantic range you’d like to generate a list for, say one billion, xrange is the function to use.

This is especially true if you have a really memory sensitive system such as a cell phone that you are working with, as **range** will use as much memory as it can to create your array of integers, which can result in a Memory Error and crash your program. It’s a memory hungry beast.

**Q26. How do you write comments in python?**

***Ans:*** Comments in Python start with a # character. However, alternatively at times, commenting is done using docstrings(strings enclosed within triple quotes).

**Example:**

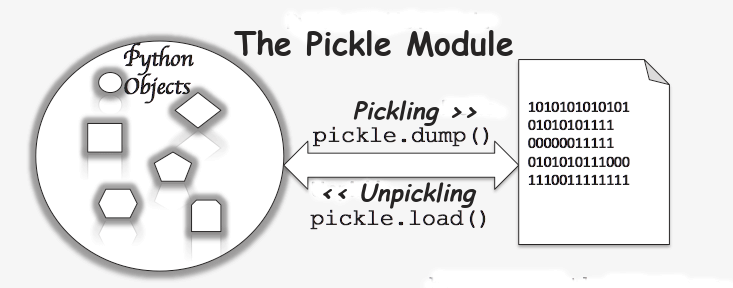
#Comments in Python start like this

print("Comments in Python start with a #")

**Output:** Comments in Python start with a #

**Q27. What is pickling and unpickling?**

**Ans:** Pickle module accepts any Python object and converts it into a string representation and dumps it into a file by using dump function, this process is called pickling. While the process of retrieving original Python objects from the stored string representation is called unpickling.



**Q28. What are the generators in python?**

***Ans:*** Functions that return an iterable set of items are called generators.

There are two terms involved when we discuss generators.

1. **Generator-Function :** A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the [yield keyword](https://www.geeksforgeeks.org/use-yield-keyword-instead-return-keyword-python/) rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

|  |
| --- |
| # A generator function that yields 1 for first time,  # 2 second time and 3 third time  def simpleGeneratorFun():      yield 1      yield 2      yield 3    # Driver code to check above generator function  for value in simpleGeneratorFun():      print(value) |

Output :

1

2

3

**Generator-Object :** Generator functions return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for in” loop (as shown in the above program).

|  |
| --- |
| # A Python program to demonstrate use of  # generator object with next()    # A generator function  def simpleGeneratorFun():      yield 1      yield 2      yield 3    # x is a generator object  x = simpleGeneratorFun()    # Iterating over the generator object using next  print(x.next()) # In Python 3, \_\_next\_\_()  print(x.next())  print(x.next()) |

1. Output :
2. 1
3. 2
4. 3

So a generator function returns an generator object that is iterable, i.e., can be used as an [Iterators](https://www.geeksforgeeks.org/iterators-in-python/) .

**Q29. How will you capitalize the first letter of string?**

***Ans:*** In Python, the capitalize() method capitalizes the first letter of a string. If the string already consists of a capital letter at the beginning, then, it returns the original string.

**Q30. How will you convert a string to all lowercase?**

***Ans:*** To convert a string to lowercase, lower() function can be used.

**Example:**

|  |  |
| --- | --- |
| 1  2 | stg='ABCD'  print(stg.lower()) |

**Output:** abcd

**Q31. How to comment multiple lines in python?**

***Ans:*** Multi-line comments appear in more than one line. All the lines to be commented are to be prefixed by a #. You can also a very good **shortcut method to comment multiple lines**. All you need to do is hold the ctrl key and **left click** in every place wherever you want to include a # character and type a # just once. This will comment all the lines where you introduced your cursor.

#Note –single line comment  
“””Note  
Note  
Note”””—–multiline comment

**Q32.What are docstrings in Python?**

***Ans:*** Docstrings are not actually comments, but, they are ***documentation strings***. These docstrings are within triple quotes. They are not assigned to any variable and therefore, at times, serve the purpose of comments as well.

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | """  Using docstring as a comment.  This code divides 2 numbers  """  x=8  y=4  z=x/y  print(z) |

**Output:** 2.0

**Q33. What is the purpose of is, not and in operators?**

***Ans:*** Operators are special functions. They take one or more values and produce a corresponding result.

is: returns true when 2 operands are true  (Example: “a” is ‘a’)

not: returns the inverse of the boolean value

in: checks if some element is present in some sequence

**Q34. What is the usage of help() and dir() function in Python?**

**Ans:** Help() and dir() both functions are accessible from the Python interpreter and used for viewing a consolidated dump of built-in functions.

1. Help() function: The help() function is used to display the documentation string and also facilitates you to see the help related to modules, keywords, attributes, etc.
2. Dir() function: The dir() function is used to display the defined symbols.

**Q35. Whenever Python exits, why isn’t all the memory de-allocated?**

**Ans:**

1. Whenever Python exits, especially those Python modules which are having circular references to other objects or the objects that are referenced from the global namespaces are not always de-allocated or freed.
2. It is impossible to de-allocate those portions of memory that are reserved by the C library.
3. On exit, because of having its own efficient clean up mechanism, Python would try to de-allocate/destroy every other object.

**Q36. What is a dictionary in Python?**

**Ans:** The built-in datatypes in Python is called dictionary. It defines one-to-one relationship between keys and values. Dictionaries contain pair of keys and their corresponding values. Dictionaries are indexed by keys.

Let’s take an example:

The following example contains some keys. Country, Capital & PM. Their corresponding values are India, Delhi and Modi respectively.

|  |  |
| --- | --- |
| 1 | dict={'Country':'India','Capital':'Delhi','PM':'Modi'} |
| 1 | print dict[Country] |

India

|  |  |
| --- | --- |
| 1 | print dict[Capital] |

Delhi

|  |  |
| --- | --- |
| 1 | print dict[PM] |

Modi

**Q37. How can the ternary operators be used in python?**

**Ans:** The Ternary operator is the operator that is used to show the conditional statements. This consists of the true or false values with a statement that has to be evaluated for it.

**Syntax**:

The Ternary operator will be given as:  
[on\_true] if [expression] else [on\_false]x, y = 25, 50big = x if x < y else y

**Example:**

The expression gets evaluated like if x<y else y, in this case if x<y is true then the value is returned as big=x and if it is incorrect then big=y will be sent as a result.

**Q38. What does this mean: \*args, \*\*kwargs? And why would we use it?**

**Ans:** We use \*args when we aren’t sure how many arguments are going to be passed to a function, or if we want to pass a stored list or tuple of arguments to a function. \*\*kwargs is used when we don’t know how many keyword arguments will be passed to a function, or it can be used to pass the values of a dictionary as keyword arguments. The identifiers args and kwargs are a convention, you could also use \*bob and \*\*billy but that would not be wise.

**\*args**

\*args is a special syntax used in function definition to pass variable-length argument.

“\*” means variable length and “args” is the name used by convention. You can use any other.

**def** **multiply**(a, b, \*argv):

mul = a \* b

**for** num **in** argv:

mul \*= num

**return** mul

**print**(**multiply**(1, 2, 3, 4, 5)) *#output: 120*

**\*\*kwargs**

* \*\*kwargs is a special syntax used in function definition to pass variable-length keyworded argument.
* Here, also, “kwargs” is used just by convention. You can use any other name.
* Keyworded argument means a variable which has a name when passed to a function.
* It is actually a dictionary of variable name and its value.

**def** **tellArguments**(\*\*kwargs):

**for** key, value **in** kwargs**.items**():

**print**(key + ": " + value)

**tellArguments**(arg1 = "argument 1", arg2 = "argument 2", arg3 = "argument 3")

*#output:*

*# arg1: argument 1*

*# arg2: argument 2*

*# arg3: argument 3*

**Q39. What does len() do?**

***Ans:*** It is used to determine the length of a string, a list, an array, etc.

**Example:**

|  |  |
| --- | --- |
| 1  2 | stg='ABCD'  len(stg) |

**Q40. Explain split(), sub(), subn() methods of “re” module in Python.**

**Ans:** To modify the strings, Python’s “re” module is providing 3 methods. They are:

* split() – uses a regex pattern to “split” a given string into a list.
* sub() – finds all substrings where the regex pattern matches and then replace them with a different string
* subn() – it is similar to sub() and also returns the new string along with the no. of replacements.

**Q41. What are negative indexes and why are they used?**

**Ans:** The sequences in Python are indexed and it consists of the positive as well as negative numbers. The numbers that are positive uses ‘0’ that is uses as first index and ‘1’ as the second index and the process goes on like that.

The index for the negative number starts from ‘-1’ that represents the last index in the sequence and ‘-2’ as the penultimate index and the sequence carries forward like the positive number.

The negative index is used to remove any new-line spaces from the string and allow the string to except the last character that is given as S[:-1]. The negative index is also used to show the index to represent the string in correct order.

**Q42. What are Python packages?**

***Ans:*** Python packages are namespaces containing multiple modules.

**Q43.How can files be deleted in Python?**

***Ans:*** To delete a file in Python, you need to import the OS Module. After that, you need to use the os.remove() function.

**Example:**

|  |  |
| --- | --- |
| 1  2 | import os  os.remove("xyz.txt") |

**Q44. What are the built-in types of python?**

***Ans:*** Built-in types in Python are as follows –

* Integers
* Floating-point
* Complex numbers
* Strings
* Boolean
* Built-in functions

**Q45. What advantages do NumPy arrays offer over (nested) Python lists?**

**Ans:**

1. Python’s lists are efficient general-purpose containers. They support (fairly) efficient insertion, deletion, appending, and concatenation, and Python’s list comprehensions make them easy to construct and manipulate.
2. They have certain limitations: they don’t support “vectorized” operations like elementwise addition and multiplication, and the fact that they can contain objects of differing types mean that Python must store type information for every element, and must execute type dispatching code when operating on each element.
3. [NumPy](https://www.edureka.co/blog/python-numpy-tutorial/) is not just more efficient; it is also more convenient. You get a lot of vector and matrix operations for free, which sometimes allow one to avoid unnecessary work. And they are also efficiently implemented.
4. NumPy array is faster and You get a lot built in with NumPy, FFTs, convolutions, fast searching, basic statistics, linear algebra, [histograms](https://www.edureka.co/blog/python-matplotlib-tutorial/#Histogram), etc.

**Q46. How to add values to a python array?**

***Ans:*** Elements can be added to an array using the **append()**, **extend()** and the **insert (i,x)** functions.

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | a=arr.array('d', [1.1 , 2.1 ,3.1] )  a.append(3.4)  print(a)  a.extend([4.5,6.3,6.8])  print(a)  a.insert(2,3.8)  print(a) |

**Output:**

array(‘d’, [1.1, 2.1, 3.1, 3.4])

array(‘d’, [1.1, 2.1, 3.1, 3.4, 4.5, 6.3, 6.8])

array(‘d’, [1.1, 2.1, 3.8, 3.1, 3.4, 4.5, 6.3, 6.8])

**Q47. How to remove values to a python array?**

***Ans:*** Array elements can be removed using **pop()** or **remove()** method. The difference between these two functions is that the former returns the deleted value whereas the latter does not.

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5 | a=arr.array('d', [1.1, 2.2, 3.8, 3.1, 3.7, 1.2, 4.6])  print(a.pop())  print(a.pop(3))  a.remove(1.1)  print(a) |

**Output:**

4.6

3.1

array(‘d’, [2.2, 3.8, 3.7, 1.2])

**Q48. Does Python have OOps concepts?**

***Ans:*** Python is an object-oriented programming language. This means that any program can be solved in python by creating an object model. However, Python can be treated as procedural as well as structural language.

**Q49. What is the difference between deep and shallow copy?**

***Ans:****Shallow copy* is used when a new instance type gets created and it keeps the values that are copied in the new instance. Shallow copy is used to copy the reference pointers just like it copies the values. These references point to the original objects and the changes made in any member of the class will also affect the original copy of it. Shallow copy allows faster execution of the program and it depends on the size of the data that is used.

*Deep copy* is used to store the values that are already copied. Deep copy doesn’t copy the reference pointers to the objects. It makes the reference to an object and the new object that is pointed by some other object gets stored. The changes made in the original copy won’t affect any other copy that uses the object. Deep copy makes execution of the program slower due to making certain copies for each object that is been called.

**Q50. How is Multithreading achieved in Python?**

Multithreading is a threading technique in Python programming to run multiple threads concurrently by rapidly switching between threads with a CPU help (called context switching). Besides, it allows sharing of its data space with the main threads inside a process that share information and communication with other threads easier than individual processes. Multithreading aims to perform multiple tasks simultaneously, which increases performance, speed and improves the rendering of the application.

Note: The Python Global Interpreter Lock (GIL) allows running a single thread at a time, even the machine has multiple processors.

**Benefits of Multithreading in Python**

Following are the benefits to create a multithreaded application in Python, as follows:

1. It ensures effective utilization of computer system resources.
2. Multithreaded applications are more responsive.
3. It shares resources and its state with sub-threads (child) which makes it more economical.
4. It makes the multiprocessor architecture more effective due to similarity.
5. It saves time by executing multiple threads at the same time.
6. The system does not require too much memory to store multiple threads.

**When to use Multithreading in Python?**

It is a very useful technique for time-saving and improving the performance of an application. Multithreading allows the programmer to divide application **tasks** into sub-tasks and simultaneously run them in a program. It allows threads to communicate and share resources such as files, data, and memory to the same processor. Furthermore, it increases the user's responsiveness to continue running a program even if a part of the application is the length or blocked.

**How to achieve multithreading in Python?**

There are two main modules of multithreading used to handle threads in [Python](https://www.javatpoint.com/python-tutorial).

1. The thread module
2. The threading module

<https://www.javatpoint.com/multithreading-in-python-3>

**Q51. What is the process of compilation and linking in python?**

**Ans:** The compiling and linking allows the new extensions to be compiled properly without any error and the linking can be done only when it passes the compiled procedure. If the dynamic loading is used then it depends on the style that is being provided with the system. The python interpreter can be used to provide the dynamic loading of the configuration setup files and will rebuild the interpreter.

The steps that are required in this as:

1. Create a file with any name and in any language that is supported by the compiler of your system. For example file.c or file.cpp
2. Place this file in the Modules/ directory of the distribution which is getting used.
3. Add a line in the file Setup.local that is present in the Modules/ directory.
4. Run the file using spam file.o
5. After a successful run of this rebuild the interpreter by using the make command on the top-level directory.
6. If the file is changed then run rebuildMakefile by using the command as ‘make Makefile’.

**Q52. What are Python libraries? Name a few of them.**

Python libraries are a collection of Python packages. Some of the majorly used python libraries are – [Numpy](https://www.edureka.co/blog/python-numpy-tutorial/), [Pandas](https://www.edureka.co/blog/python-pandas-tutorial/), [Matplotlib](https://www.edureka.co/blog/python-matplotlib-tutorial/), [Scikit-learn](https://www.edureka.co/blog/scikit-learn-machine-learning/) and many more.

**Q53. What is split used for?**

The split() method is used to separate a given string in Python.

**Example:**

|  |  |
| --- | --- |
| 1  2 | a="edureka python"  print(a.split()) |

**Output:** [‘edureka’, ‘python’]

**Q54. How to import modules in python?**

Modules can be imported using the **import**keyword.  You can import modules in three ways-

**Example:**

|  |  |
| --- | --- |
| 1  2  3 | import array           #importing using the original module name  import array as arr    # importing using an alias name  from array import \*    #imports everything present in the array module |

**OOPS Python Interview Questions**

**Q55. Explain Inheritance in Python with an example.**

**Ans:** Inheritance allows One class to gain all the members(say attributes and methods) of another class. Inheritance provides code reusability, makes it easier to create and maintain an application. The class from which we are inheriting is called super-class and the class that is inherited is called a derived / child class.

They are different types of inheritance supported by Python:

1. Single Inheritance – where a derived class acquires the members of a single super class.
2. Multi-level inheritance – a derived class d1 in inherited from base class base1, and d2 are inherited from base2.

If class A inherits from B and C inherits from A it’s called multilevel inheritance.

class B(object):

def \_\_init\_\_(self):

self.b=0

class A(B):

def \_\_init\_\_(self):

self.a=0

class C(A):

def \_\_init\_\_(self):

self.c=0

1. Hierarchical inheritance – from one base class you can inherit any number of child classes
2. Multiple inheritance – a derived class is inherited from more than one base class.

**Q56. How are classes created in Python?**

**Ans:** Class in Python is created using the **class** keyword.

**Example:**

**Data Science Training**

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Employee:  def \_\_init\_\_(self, name):  self.name = name  E1=Employee("abc")  print(E1.name) |

**Output:** abc

**Q57. What is monkey patching in Python?**

**Ans:** In Python, the term monkey patch only refers to dynamic modifications of a class or module at run-time.

Consider the below example:

|  |  |
| --- | --- |
| 1  2  3  4 | # m.py  class MyClass:  def f(self):  print "f()" |

We can then run the monkey-patch testing like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import m  def monkey\_f(self):  print "monkey\_f()"    m.MyClass.f = monkey\_f  obj = m.MyClass()  obj.f() |

The output will be as below:

monkey\_f()

As we can see, we did make some changes in the behavior of*f()* in *MyClass*using the function we defined, *monkey\_f()*, outside of the module *m*.

**Q58. Does python support multiple inheritance?**

**Ans:** Multiple inheritance means that a class can be derived from more than one parent classes. Python does support multiple inheritance, unlike Java.

**Q59. What is Polymorphism in Python?**

**Ans:** Polymorphism means the ability to take multiple forms. So, for instance, if the parent class has a method named ABC then the child class also can have a method with the same name ABC having its own parameters and variables. Python allows polymorphism.

**Q60. Define encapsulation in Python?**

**Ans:** Encapsulation means binding the code and the data together. A Python class in an example of encapsulation.

**Q61. How do you do data abstraction in Python?**

**Ans:** Data Abstraction is providing only the required details and hiding the implementation from the world. It can be achieved in Python by using interfaces and abstract classes.

**Q62.Does python make use of access specifiers?**

**Ans:** Python does not deprive access to an instance variable or function. Python lays down the concept of prefixing the name of the variable, function or method with a single or double underscore to imitate the behavior of protected and private access specifiers.

**Q63. How to create an empty class in Python?**

**Ans:** An empty class is a class that does not have any code defined within its block. It can be created using the *pass*keyword. However, you can create objects of this class outside the class itself. IN PYTHON THE PASS command does nothing when its executed. it’s a null statement.

**For example-**

|  |  |
| --- | --- |
| 1  2  3  4  5 | class a:    &amp;amp;amp;nbsp; pass  obj=a()  obj.name="xyz"  print("Name = ",obj.name) |

**Output:**

Name = xyz

**Q64. What does an object() do?**

**Ans:** It returns a featureless object that is a base for all classes. Also, it does not take any parameters.

**Q70. Write a one-liner that will count the number of capital letters in a file. Your code should work even if the file is too big to fit in memory.**

**Ans:**  Let us first write a multiple line solution and then convert it to one-liner code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | with open(SOME\_LARGE\_FILE) as fh:  count = 0  text = fh.read()  for character in text:      if character.isupper():  count += 1 |

We will now try to transform this into a single line.

|  |  |
| --- | --- |
| 1 | count sum(1 for line in fh for character in line if character.isupper()) |

**Q71. Write a sorting algorithm for a numerical dataset in Python.**

**Ans:** The following code can be used to sort a list in Python:

|  |  |
| --- | --- |
| 1  2  3  4 | list = ["1", "4", "0", "6", "9"]  list = [int(i) for i in list]  list.sort()  print (list) |

**Q72: What are Python decorators?**

**Decorators** in Python are essentially functions that add functionality to an existing function in Python without changing the structure of the function itself. They are represented by the @decorator\_name in Python and are called in bottom-up fashion. For example:

*# decorator function to convert to lowercase*

**def** **lowercase\_decorator**(function):

**def** **wrapper**():

func = function()

string\_lowercase = func.lower()

**return** string\_lowercase

**return** wrapper

*# decorator function to split words*

**def** **splitter\_decorator**(function):

**def** **wrapper**():

func = function()

string\_split = func.split()

**return** string\_split

**return** wrapper

**@splitter\_decorator** *# this is executed next*

**@lowercase\_decorator** *# this is executed first*

**def** **hello**():

**return** 'Hello World'

**hello**() *# output => [ 'hello' , 'world' ]*

The beauty of the decorators lies in the fact that besides adding functionality to the output of the method, they can even **accept arguments** for functions and can further modify those arguments before passing it to the function itself. The **inner nested function**, i.e. 'wrapper' function, plays a significant role here. It is implemented to enforce **encapsulation** and thus, keep itself hidden from the global scope.

*# decorator function to capitalize names*

**def** **names\_decorator**(function):

**def** **wrapper**(arg1, arg2):

arg1 = arg1.capitalize()

arg2 = arg2.capitalize()

string\_hello = function(arg1, arg2)

**return** string\_hello

**return** wrapper

**@names\_decorator**

**def** **say\_hello**(name1, name2):

**return** 'Hello ' + name1 + '! Hello ' + name2 + '!'

**say\_hello**('sara', 'ansh') *# output => 'Hello Sara! Hello Ansh!'*

**Python Libraries – Python Interview Questions**

**Q72: What are Python modules?**

**Answer:** A file containing Python code like functions and variables is a Python module. A Python module is an executable file with a .py extension.

Python has built-in modules some of which are:

* os
* sys
* math
* random
* data time
* JSON

**Q73. Explain what Flask is and its benefits?**

**Ans:** Flask is a web microframework for Python based on “Werkzeug, Jinja2 and good intentions” BSD license. Werkzeug and Jinja2 are two of its dependencies. This means it will have little to no dependencies on external libraries.  It makes the framework light while there is a little dependency to update and fewer security bugs.

A session basically allows you to remember information from one request to another. In a flask, a session uses a signed cookie so the user can look at the session contents and modify. The user can modify the session if only it has the secret key Flask.secret\_key.

**Q74. Is Django better than Flask?**

**Ans:** Django and Flask map the URL’s or addresses typed in the web browsers to functions in Python.

Flask is much simpler compared to Django but, Flask does not do a lot for you meaning you will need to specify the details, whereas Django does a lot for you wherein you would not need to do much work. [Django](https://www.edureka.co/blog/django-tutorial/) consists of prewritten code, which the user will need to analyze whereas Flask gives the users to create their own code, therefore, making it simpler to understand the code. Technically both are equally good and both contain their own pros and cons.

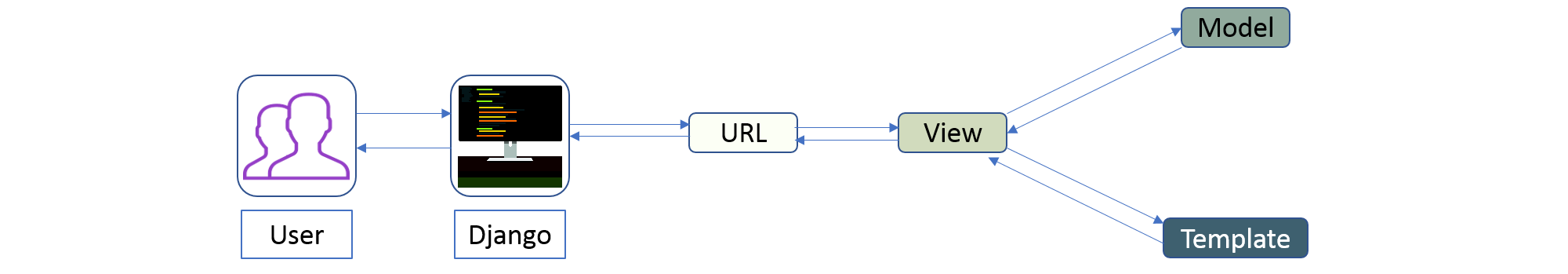
**Q75. Mention the differences between Django, Pyramid and Flask.**

**Ans:**

* Flask is a “microframework” primarily build for a small application with simpler requirements. In flask, you have to use external libraries. Flask is ready to use.
* Pyramid is built for larger applications. It provides flexibility and lets the developer use the right tools for their project. The developer can choose the database, URL structure, templating style and more. Pyramid is heavy configurable.
* Django can also be used for larger applications just like Pyramid. It includes an ORM.

**Q76. Discuss Django architecture.**

**Ans:** Django MVT Pattern:

**Figure:** *Python Interview Questions – Django Architecture*

The developer provides the Model, the view and the template then just maps it to a URL and Django does the magic to serve it to the user.

### ****Q. What are dataframes?****

A pandas dataframe is a data structure in pandas which is mutable. Pandas has support for heterogeneous data which is arranged across two axes.( rows and columns).

Reading files into pandas:-

|  |  |
| --- | --- |
| 1  2 | Import pandas as pd  df=p.read\_csv(“mydata.csv”) |

Here df is a pandas data frame. read\_csv() is used to read a comma delimited file as a dataframe in pandas.

### ****Q. What is a Pandas Series?****

Series is a one dimensional pandas data structure which can data of almost any type. It resembles an excel column. It supports multiple operations and is used for single dimensional data operations.

Creating a series from data:

**Code**

import pandas as pd

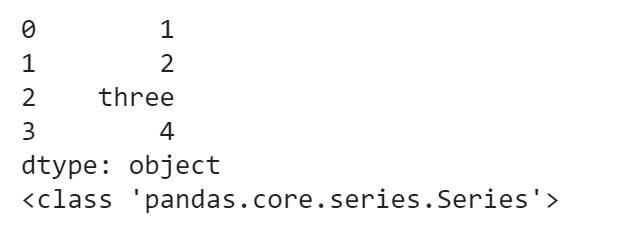
data=["1",2,"three",4.0]

series=pd.Series(data)

print(series)

print(type(series))

**Output**



### ****Q. What is pandas groupby?****

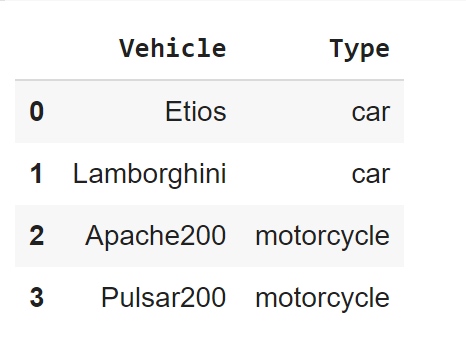
A pandas groupby is a feature supported by pandas which is used to split and group an object.  Like the sql/mysql/oracle groupby it used to group data by classes, entities which can be further used for aggregation. A dataframe can be grouped by one or more columns.

**Code**

df = pd.DataFrame({'Vehicle':['Etios','Lamborghini','Apache200','Pulsar200'], 'Type':["car","car","motorcycle","motorcycle"]})

df

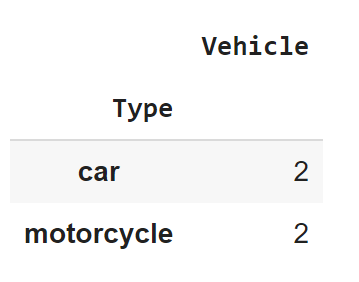
**Output**



To perform groupby type the following **code:**

df.groupby('Type').count()

**Output**



### ****Q. How to create a dataframe from lists?****

To create a dataframe from lists ,

1)create an empty dataframe

2)add lists as individuals columns to the list

**Code**

df=pd.DataFrame()

bikes=["bajaj","tvs","herohonda","kawasaki","bmw"]

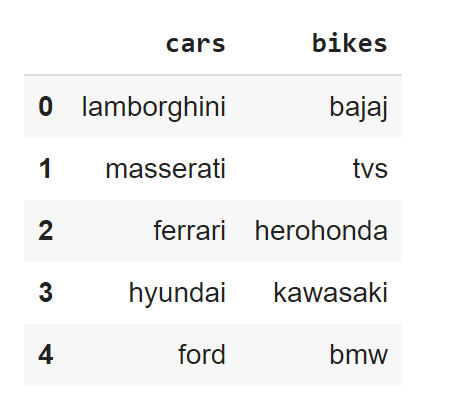
cars=["lamborghini","masserati","ferrari","hyundai","ford"]

df["cars"]=cars

df["bikes"]=bikes

df

**Output**



### ****14. How to create dataframe from a dictionary?****

A dictionary can be directly passed as an argument to the DataFrame() function to create the data frame.

Code

import pandas as pd

bikes=["bajaj","tvs","herohonda","kawasaki","bmw"]

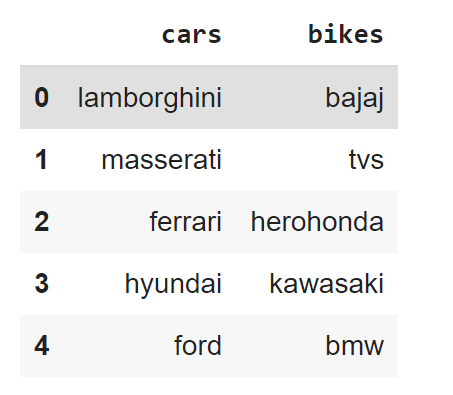
cars=["lamborghini","masserati","ferrari","hyundai","ford"]

d={"cars":cars,"bikes":bikes}

df=pd.DataFrame(d)

df

Output



### ****Q. How to combine dataframes in pandas?****

Two different data frames can be stacked either horizontally or vertically by the concat(), append() and join() functions in pandas.

Concat works best when the dataframes have the same columns and can be used for concatenation of data having similar fields and is basically vertical stacking of dataframes into a single dataframe.

Append() is used for horizontal stacking of dataframes. If two tables(dataframes) are to be merged together then this is the best concatenation function.

Join is used when we need to extract data from different dataframes which are having one or more common columns. The stacking is horizontal in this case.

Before going through the questions, here’s a quick video to help you refresh your memory on Python.

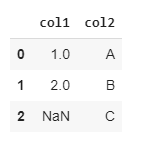
### ****Q. What kind of joins does pandas offer?****

Pandas has a left join, inner join, right join and an outer join.

### ****Q. How to merge dataframes in pandas?****

Merging depends on the type and fields of different dataframes being merged. If data is having similar fields data is merged along axis 0 else they are merged along axis 1.

### ****Q. Give the below dataframe drop all rows having Nan.****

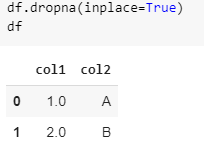


The dropna function can be used to do that.

df.dropna(inplace=True)

df

Output



### ****Q. How to access the first five entries of a dataframe?****

By using the head(5) function we can get the top five entries of a dataframe. By default df.head() returns the top 5 rows. To get the top n rows df.head(n) will be used.

### ****Q. How to access the last five entries of a dataframe?****

By using tail(5) function we can get the top five entries of a dataframe. By default df.tail() returns the top 5 rows. To get the last n rows df.tail(n) will be used.

### ****Q. How to fetch a data entry from a pandas dataframe using a given value in index?****

To fetch a row from dataframe given index x, we can use loc.

Df.loc[10] where 10 is the value of the index.

Code

import pandas as pd

bikes=["bajaj","tvs","herohonda","kawasaki","bmw"]

cars=["lamborghini","masserati","ferrari","hyundai","ford"]

d={"cars":cars,"bikes":bikes}

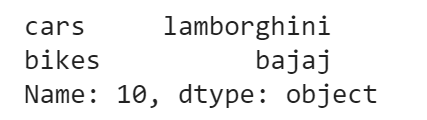
df=pd.DataFrame(d)

a=[10,20,30,40,50]

df.index=a

df.loc[10]

Output



### ****Q. How to create a new column in pandas by using values from other columns?****

We can perform column based mathematical operations on a pandas dataframe. Pandas columns containing numeric values can be operated upon by operators.

Code

import pandas as pd

a=[1,2,3]

b=[2,3,5]

d={"col1":a,"col2":b}

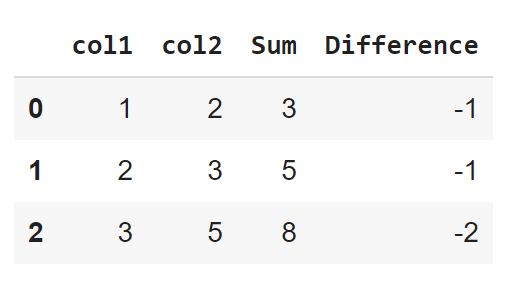
df=pd.DataFrame(d)

df["Sum"]=df["col1"]+df["col2"]

df["Difference"]=df["col1"]-df["col2"]

df

Output



### ****Q. What are the different functions that can be used by grouby in pandas ?****

grouby() in pandas can be used with multiple aggregate functions. Some of which are sum(),mean(), count(),std().

Data is divided into groups based on categories and then the data in these individual groups can be aggregated by the aforementioned functions.

### ****Q. How to select columns in pandas and add them to a new dataframe? What if there are two columns with the same name?****

If df is dataframe in pandas df.columns gives the list of all columns. We can then form new columns by selecting columns.

If there are two columns with the same name then both columns get copied to the new dataframe.

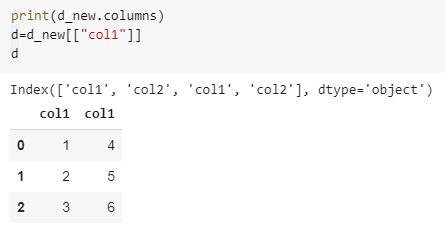
Code

print(d\_new.columns)

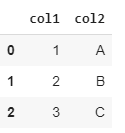
d=d\_new[["col1"]]

d

Output



### ****Q. How to delete a column or group of columns in pandas? Given the below dataframe drop column “col1”.****



drop() function can be used to delete the columns from a dataframe.

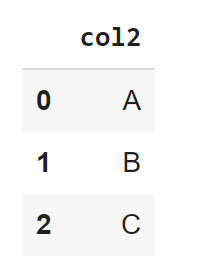
d={"col1":[1,2,3],"col2":["A","B","C"]}

df=pd.DataFrame(d)

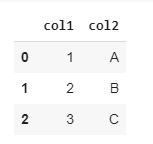
df=df.drop(["col1"],axis=1)

df

Output



### ****Q. Given the following data frame drop rows having column values as A.****



Code

d={"col1":[1,2,3],"col2":["A","B","C"]}

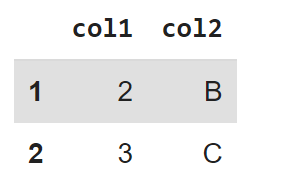
df=pd.DataFrame(d)

df.dropna(inplace=True)

df=df[df.col1!=1]

df

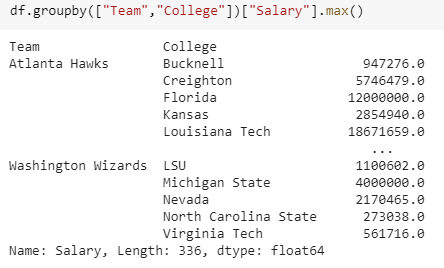
Output



### ****Q. Given the below dataset find the highest paid player in each college in each team.****



df.groupby(["Team","College"])["Salary"].max()

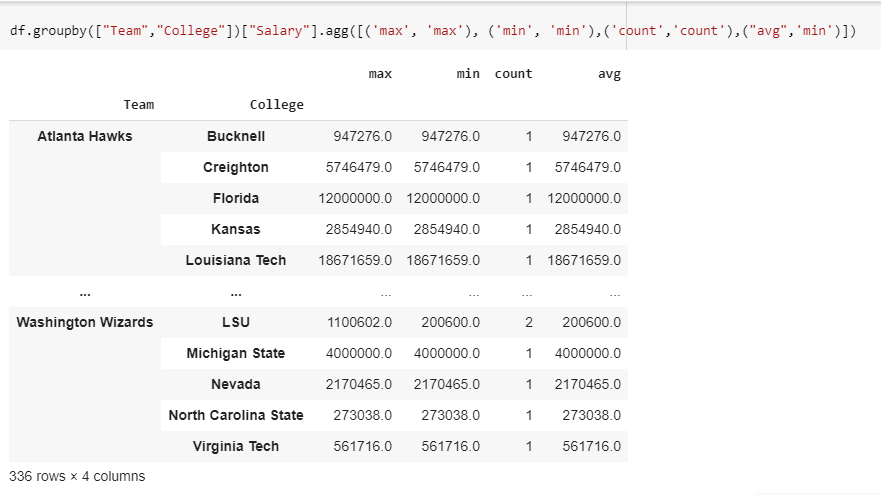


### ****Q. Given the above dataset find the min max and average salary of a player collegewise and teamwise.****

Code

df.groupby(["Team","College"])["Salary"].max.agg([('max','max'),('min','min'),('count','count'),('avg','min')])

Output



### ****Q. What is reindexing in pandas?****

Reindexing is the process of re-assigning the index of a pandas dataframe.

Code

import pandas as pd

bikes=["bajaj","tvs","herohonda","kawasaki","bmw"]

cars=["lamborghini","masserati","ferrari","hyundai","ford"]

d={"cars":cars,"bikes":bikes}

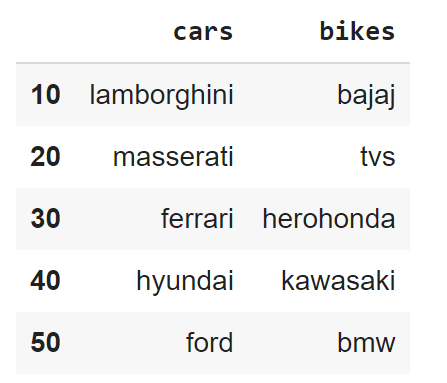
df=pd.DataFrame(d)

a=[10,20,30,40,50]

df.index=a

df

Output



**Data Analysis – Python Interview Questions**

**Q85. What is map function in Python?**

**Ans:** *map* function executes the function given as the first argument on all the elements of the iterable given as the second argument. If the function given takes in more than 1 arguments, then many iterables are given. #Follow the link to know more similar functions.

**Q86. Is python numpy better than lists?**

**Ans:** We use python numpy array instead of a list because of the below three reasons:

1. Less Memory
2. Fast
3. Convenient

For more information on these parameters, you can refer to this section – [Numpy Vs List](https://www.edureka.co/blog/python-numpy-tutorial/#NumpyVsList).

**Q87. How to get indices of N maximum values in a NumPy array?**

**Ans:** We can get the indices of N maximum values in a NumPy array using the below code:

|  |  |
| --- | --- |
| 1  2  3 | import numpy as np  arr = np.array([1, 3, 2, 4, 5])  print(arr.argsort()[-3:][::-1]) |

Output

[ 4 3 1 ]

**Q88. How do you calculate percentiles with Python/ NumPy?**

**Ans:** We can calculate percentiles with the following code

|  |  |
| --- | --- |
| 1  2  3  4 | import numpy as np  a = np.array([1,2,3,4,5])  p = np.percentile(a, 50) #Returns 50th percentile, e.g. median  print(p) |

Output

3

**Q89. What is the difference between NumPy and SciPy?**

**Ans:**

1. In an ideal world, NumPy would contain nothing but the array data type and the most basic operations: indexing, sorting, reshaping, basic elementwise functions, et cetera.
2. All numerical code would reside in SciPy. However, one of NumPy’s important goals is compatibility, so NumPy tries to retain all features supported by either of its predecessors.
3. Thus NumPy contains some linear algebra functions, even though these more properly belong in SciPy. In any case, SciPy contains more fully-featured versions of the linear algebra modules, as well as many other numerical algorithms.
4. If you are doing scientific computing with python, you should probably install both NumPy and SciPy. Most new features belong in SciPy rather than NumPy.

**Q90. How do you make 3D plots/visualizations using NumPy/SciPy?**

**Ans:** Like 2D plotting, 3D graphics is beyond the scope of NumPy and SciPy, but just as in the 2D case, packages exist that integrate with NumPy. Matplotlib provides basic 3D plotting in the mplot3d subpackage, whereas Mayavi provides a wide range of high-quality 3D visualization features, utilizing the powerful VTK engine.

**Python Program**

#### Q - Swap number in python?

x = 5

y = 10

x, y = y, x

print("x =", x)

print("y =", y)

#### Q - Suppose list1 = [3,4,5,2,1,0], what is list1 after list1.pop(1)?

|  |
| --- |
| list1 = [3,4,5,2,1] |
| list1 = [3,4,5,2,0] |
| list1 = [3,5,2,1,0] |
| list1 = [3,4,5,2] |

#### Q - What is the output of the following statement "Hello World"[::-1]?

|  |
| --- |
| "Hello World" |
| "World Hello" |
| "dlroW olleH" |
| "olleH dlroW" |

#### Q - What is the difference between lists and tuples?

|  |
| --- |
| List is a sequence data type, while tuple is not. |
| Tuples are mutable but lists are immutable. |
| Tuple is a sequence data type, while lists is not. |
| Lists are mutable but tuples are immutable. |

#### Q - Let func = lambda a, b : (a \*\* b), what is the output of func(float(10),20) ?

|  |
| --- |
| 100000000000000000000 |
| 1e+20 |
| 100000000000000000000.0 |
| 1.0e+20 |

#### Q - Which statement is false for \_\_init\_\_?

|  |
| --- |
| \_\_init\_\_ is called manually on object creation. |
| \_\_init\_\_ is a constructor method in Python. |
| All classes have a \_\_init\_\_ method associated with them. |
| \_\_init\_\_ allocates memory for objects. |

#### Q - Which of the following is the function responsible for pickling?

|  |
| --- |
| pickle.save() |
| pickle.store() |
| pickle.pickle() |
| pickle.dump() |

#### Q - Which of the following is a protected attribute?

|  |
| --- |
| \_\_sara\_\_ |
| \_\_ansh |
| \_sara\_ |
| ansh\_\_ |

#### Q - Which of the following is untrue for Python namespaces?

|  |
| --- |
| Python namespaces are implemented as a dictionary in Python. |
| Python namespaces have keys as addresses of the objects. |
| Lifecycle of a namespace depends upon the scope of the objects they are mapped to. |
| Namespaces ensure that object names in a program are unique. |

#### Q - Let list1 = ['s', 'r', 'a', 's'] and list2 = ['a', 'a', 'n', 'h'], what is the output of ["".join([i, j]) for i, j in zip(list1, list2)]?

|  |
| --- |
| ['s', 'a', 'r', 'a', 'a', 'n', 's', 'h'] |
| ['s', 'r', 'a', 's', 'a', 'a', 'n', 'h'] |
| ['sa', 'ra', 'an', 'sh'] |
| ['sa', 'sa', 'sn', 'sh', 'ra', 'ra', 'rn', 'rh', 'aa', 'aa', 'an', 'ah', 'sa', 'sa', 'sn', 'sh'] |

#### Q - time.time() in Python returns?

|  |
| --- |
| Current time. |
| Current time in milliseconds. |
| Current time in milliseconds since midnight, January 1, 1970. |
| Current time in milliseconds since midnight, January 1, 1970 GMT (the Unix time). |

## Multiple Choice Questions (MCQ) – Python Interview Questions

Q91. Which of the following statements create a dictionary? (Multiple Correct Answers Possible)

a) d = {}  
b) d = {“john”:40, “peter”:45}  
c) d = {40:”john”, 45:”peter”}  
d) d = (40:”john”, 45:”50”)

**Answer:** b, c & d.

Dictionaries are created by specifying keys and values.

Q92. Which one of these is floor division?

a) /  
b) //  
c) %  
d) None of the mentioned

Answer: b) //

When both of the operands are integer then python chops out the fraction part and gives you the round off value, to get the accurate answer use floor division. For ex, 5/2 = 2.5 but both of the operands are integer so answer of this expression in python is 2. To get the 2.5 as the answer, use floor division using //. So, 5//2 = 2.5

Q93. What is the maximum possible length of an identifier?

a) 31 characters  
b) 63 characters  
c) 79 characters  
d) None of the above

Answer: d) None of the above

Identifiers can be of any length.

Q94. Why are local variable names beginning with an underscore discouraged?

a) they are used to indicate a private variables of a class  
b) they confuse the interpreter  
c) they are used to indicate global variables  
d) they slow down execution

Answer: a) they are used to indicate a private variable of a class

As Python has no concept of private variables, leading underscores are used to indicate variables that must not be accessed from outside the class.

Q95. Which of the following is an invalid statement?

a) abc = 1,000,000  
b) a b c = 1000 2000 3000  
c) a,b,c = 1000, 2000, 3000  
d) a\_b\_c = 1,000,000

Answer: b) a b c = 1000 2000 3000

Spaces are not allowed in variable names.

Q96. What is the output of the following?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | try:      if '1' != 1:          raise "someError"      else:          print("someError has not occured")  except "someError":      print ("someError has occured") |

a)some Error has occured  
b)some Error has not occured  
c)invalid code  
d) none of the above

Answer: c) invalid code

A new exception class must inherit from a BaseException. There is no such inheritance here.

Q97. Suppose list1 is [2, 33, 222, 14, 25], What is list1[-1] ?

a)Error  
b)None  
c)25  
d) 2

Answer: c) 25

The index -1 corresponds to the last index in the list.

Q98. To open a file c:scores.txt for writing, we use

a) outfile = open(“c:scores.txt”, “r”)  
b) outfile = open(“c:scores.txt”, “w”)  
c) outfile = open(file = “c:scores.txt”, “r”)  
d) outfile = open(file = “c:scores.txt”, “o”)

Answer: b) The location contains double slashes ( ) and w is used to indicate that file is being written to.

Q99. What is the output of the following?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | f = None    for i in range (5):      with open("data.txt", "w") as f:          if i &amp;amp;gt; 2:              break    print f.closed |

a) True  
b) False  
c) None  
d) Error

Answer: a) True

The WITH statement when used with open file guarantees that the file object is closed when the with block exits.

Q100. When will the else part of try-except-else be executed?

a) always  
b) when an exception occurs  
c) when no exception occurs  
d) when an exception occurs into except block

**Answer:** c) when no exception occurs

The else part is executed when no exception occurs.

**Spark**

### ****1. Compare Hadoop and Spark.****

We will compare Hadoop MapReduce and Spark based on the following aspects:

|  |  |  |
| --- | --- | --- |
| ****Apache Spark vs. Hadoop**** | | |
| **Feature Criteria** | **Apache Spark** | **Hadoop** |
| **Speed** | 100 times faster than Hadoop | Decent speed |
| **Processing** | Real-time & Batch processing | Batch processing only |
| **Difficulty** | Easy because of high level modules | Tough to learn |
| **Recovery** | Allows recovery of partitions | Fault-tolerant |
| **Interactivity** | Has interactive modes | No interactive mode except Pig & Hive |

**Table:** Apache Spark versus Hadoop

Let us understand the same using an interesting analogy.

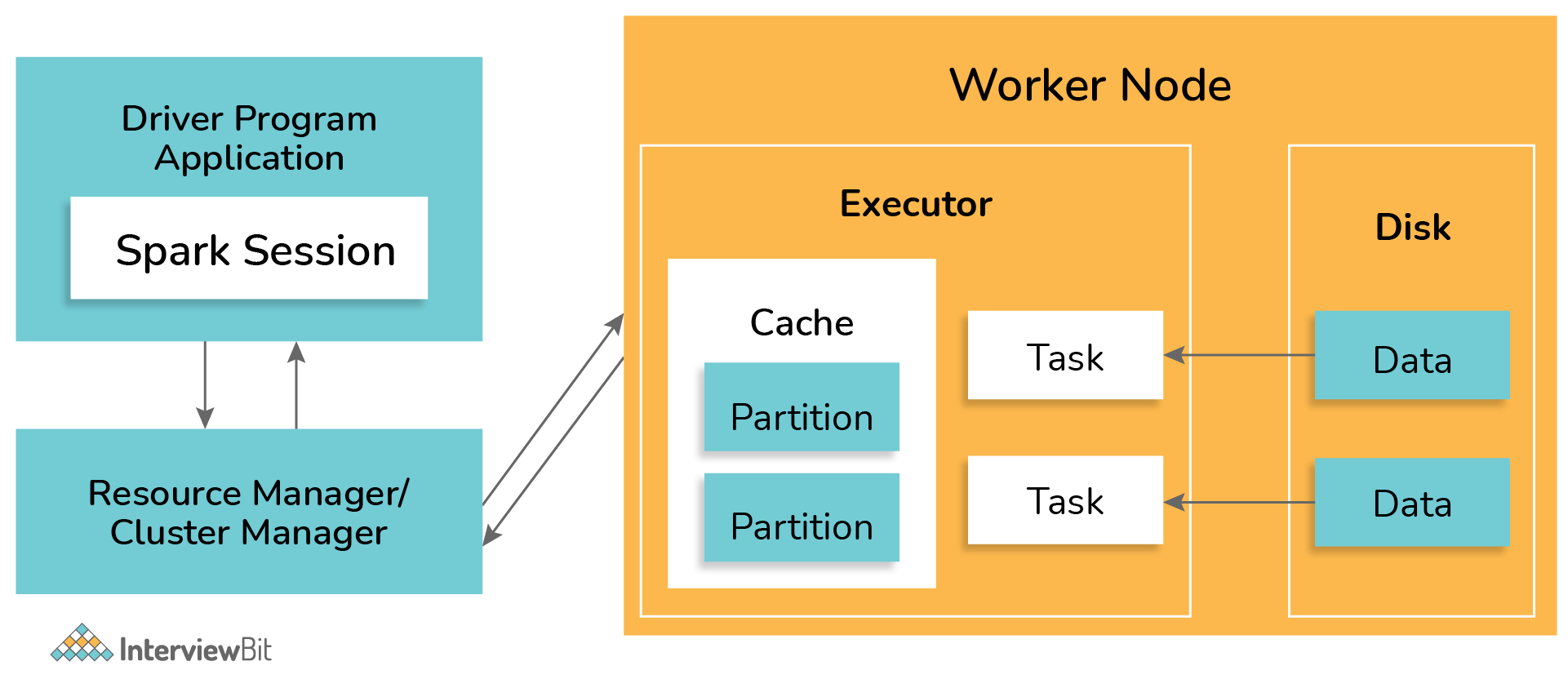
“Single cook cooking an entree is regular computing. Hadoop is multiple cooks cooking an entree into pieces and letting each cook her piece.Each cook has a separate stove and a food shelf. The first cook cooks the meat, the second cook cooks the sauce. This phase is called “Map”. A the end the main cook assembles the complete entree. This is called “Reduce”. For Hadoop, the cooks are not allowed to keep things on the stove between operations. Each time you make a particular operation, the cook puts results on the shelf. This slows things down.For Spark, the cooks are allowed to keep things on the stove between operations. This speeds things up. Finally, for Hadoop the recipes are written in a language which is illogical and hard to understand. For Spark, the recipes are nicely written.” – Stan Kladko*, Galactic Exchange.io*

### How is Apache Spark different from MapReduce?

| **MapReduce** | **Apache Spark** |
| --- | --- |
| MapReduce does only batch-wise processing of data. | Apache Spark can process the data both in real-time and in batches. |
| MapReduce does slow processing of large data. | Apache Spark runs approximately 100 times faster than MapReduce for big data processing. |
| MapReduce stores data in HDFS (Hadoop Distributed File System) which makes it take a long time to get the data. | Spark stores data in memory (RAM) which makes it easier and faster to retrieve data when needed. |
| MapReduce highly depends on disk which makes it to be a high latency framework. | Spark supports in-memory data storage and caching and makes it a low latency computation framework. |
| MapReduce requires an external scheduler for jobs. | Spark has its own job scheduler due to the in-memory data computation. |

### Q. Explain the working of Spark with the help of its architecture.

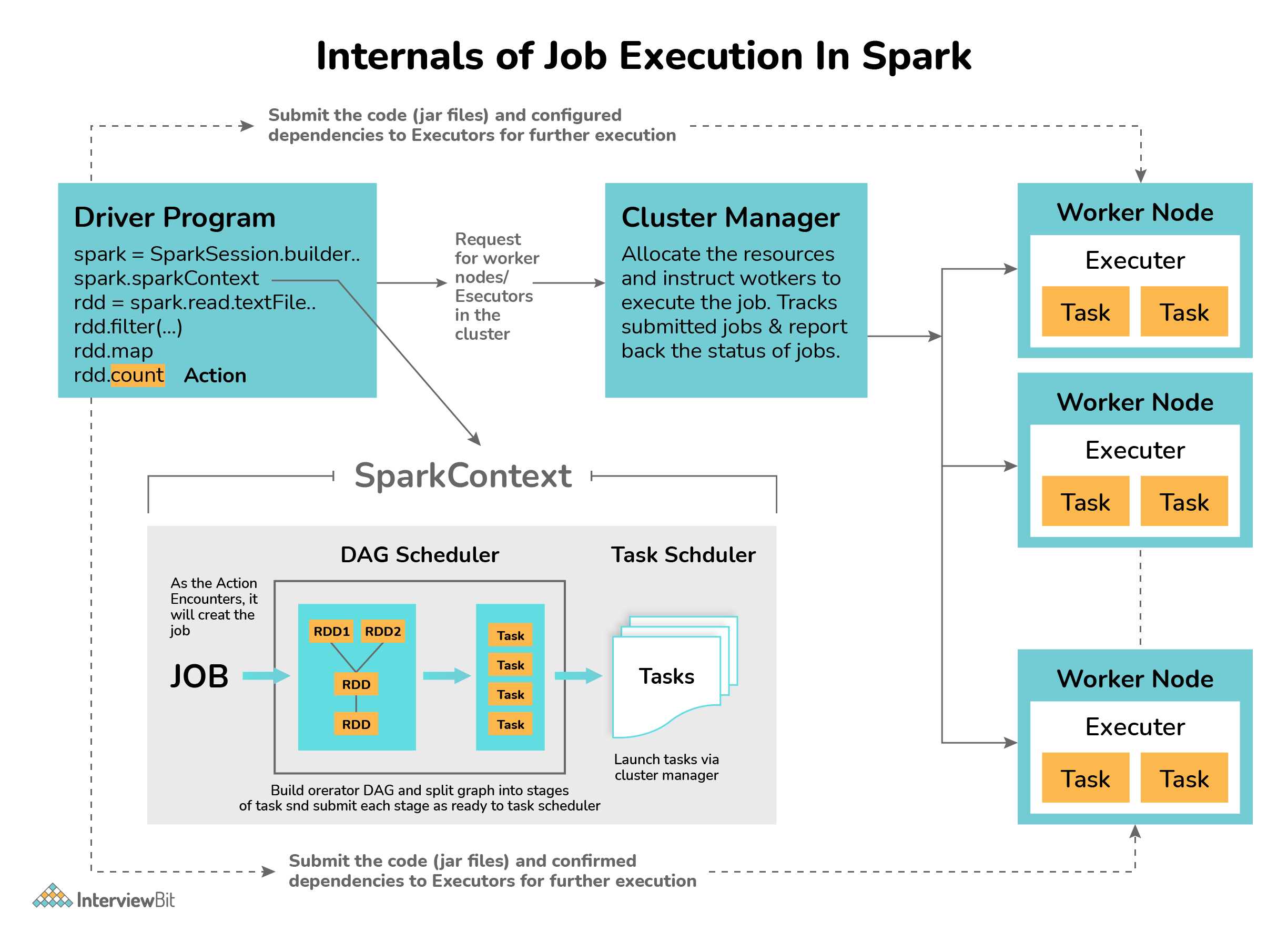
Spark applications are run in the form of independent processes that are well coordinated by the Driver program by means of a SparkSession object. The cluster manager or the resource manager entity of Spark assigns the tasks of running the Spark jobs to the worker nodes as per one task per partition principle. There are various iterations algorithms that are repeatedly applied to the data to cache the datasets across various iterations. Every task applies its unit of operations to the dataset within its partition and results in the new partitioned dataset. These results are sent back to the main driver application for further processing or to store the data on the disk. The following diagram illustrates this working as described above:



### What is the working of DAG in Spark?

DAG stands for Direct Acyclic Graph which has a set of finite vertices and edges. The vertices represent RDDs and the edges represent the operations to be performed on RDDs sequentially. The DAG created is submitted to the DAG Scheduler which splits the graphs into stages of tasks based on the transformations applied to the data. The stage view has the details of the RDDs of that stage.

The working of DAG in spark is defined as per the workflow diagram below:



* The first task is to interpret the code with the help of an interpreter. If you use the Scala code, then the Scala interpreter interprets the code.
* Spark then creates an operator graph when the code is entered in the Spark console.
* When the action is called on Spark RDD, the operator graph is submitted to the DAG Scheduler.
* The operators are divided into stages of task by the DAG Scheduler. The stage consists of detailed step-by-step operation on the input data. The operators are then pipelined together.
* The stages are then passed to the Task Scheduler which launches the task via the cluster manager to work on independently without the dependencies between the stages.
* The worker nodes then execute the task.

Each RDD keeps track of the pointer to one/more parent RDD along with its relationship with the parent. For example, consider the operation val childB=parentA.map() on RDD, then we have the RDD childB that keeps track of its parentA which is called **RDD lineage**.

### 14. Under what scenarios do you use Client and Cluster modes for deployment?

* In case the client machines are not close to the cluster, then the Cluster mode should be used for deployment. This is done to avoid the network latency caused while communication between the executors which would occur in the Client mode. Also, in Client mode, the entire process is lost if the machine goes offline.
* If we have the client machine inside the cluster, then the Client mode can be used for deployment. Since the machine is inside the cluster, there won’t be issues of network latency and since the maintenance of the cluster is already handled, there is no cause of worry in cases of failure.

### Q.Pyspark UDF

PySpark UDF (a.k.a User Defined Function) is the most useful feature of Spark SQL & DataFrame that is used to extend the PySpark build in capabilities. In this article, I will explain what is UDF? why do we need it and how to create and use it on DataFrame select(), [withColumn()](https://sparkbyexamples.com/pyspark/pyspark-dataframe-withcolumn/) and SQL using PySpark

**Why do we need a UDF?**

UDF’s are used to extend the functions of the framework and re-use these functions on multiple DataFrame’s. For example, you wanted to convert every first letter of a word in a name string to a capital case; PySpark build-in features don’t have this function hence you can create it a UDF and reuse this as needed on many Data Frames. UDF’s are once created they can be re-used on several DataFrame’s and SQL expressions.

Before you create any UDF, do your research to check if the similar function you wanted is already available in [Spark SQL Functions](https://sparkbyexamples.com/spark/spark-sql-functions-understanding/). PySpark SQL provides several predefined common functions and many more new functions are added with every release. hence, It is best to check before you reinventing the wheel.

<https://sparkbyexamples.com/pyspark/pyspark-udf-user-defined-function/#udf-need>

When you creating UDF’s you need to design them very carefully otherwise you will come across optimization & performance issues.

import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, udf

from pyspark.sql.types import StringType

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

columns = ["Seqno","Name"]

data = [("1", "john jones"),

("2", "tracey smith"),

("3", "amy sanders")]

df = spark.createDataFrame(data=data,schema=columns)

df.show(truncate=False)

def convertCase(str):

resStr=""

arr = str.split(" ")

for x in arr:

resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "

return resStr

""" Converting function to UDF """

convertUDF = udf(lambda z: convertCase(z))

df.select(col("Seqno"), \

convertUDF(col("Name")).alias("Name") ) \

.show(truncate=False)

def upperCase(str):

return str.upper()

upperCaseUDF = udf(lambda z:upperCase(z),StringType())

df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \

.show(truncate=False)

""" Using UDF on SQL """

spark.udf.register("convertUDF", convertCase,StringType())

df.createOrReplaceTempView("NAME\_TABLE")

spark.sql("select Seqno, convertUDF(Name) as Name from NAME\_TABLE") \

.show(truncate=False)

spark.sql("select Seqno, convertUDF(Name) as Name from NAME\_TABLE " + \

"where Name is not null and convertUDF(Name) like '%John%'") \

.show(truncate=False)

""" null check """

columns = ["Seqno","Name"]

data = [("1", "john jones"),

("2", "tracey smith"),

("3", "amy sanders"),

('4',None)]

df2 = spark.createDataFrame(data=data,schema=columns)

df2.show(truncate=False)

df2.createOrReplaceTempView("NAME\_TABLE2")

spark.udf.register("\_nullsafeUDF", lambda str: convertCase(str) if not str is None else "" , StringType())

spark.sql("select \_nullsafeUDF(Name) from NAME\_TABLE2") \

.show(truncate=False)

spark.sql("select Seqno, \_nullsafeUDF(Name) as Name from NAME\_TABLE2 " + \

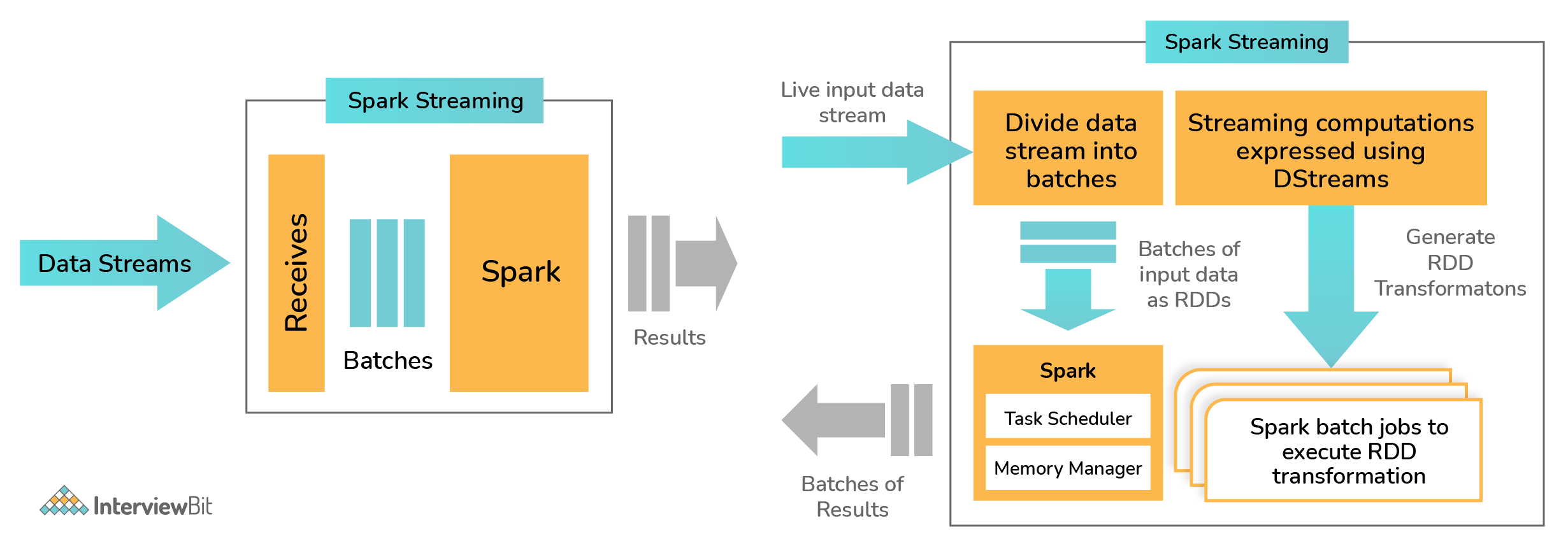
" where Name is not null and \_nullsafeUDF(Name) like '%John%'") \

.show(truncate=False)

### 15. What is Spark Streaming and how is it implemented in Spark?

Spark Streaming is one of the most important features provided by Spark. It is nothing but a Spark API extension for supporting stream processing of data from different sources.

* Data from sources like Kafka, Kinesis, Flume, etc are processed and pushed to various destinations like databases, dashboards, machine learning APIs, or as simple as file systems. The data is divided into various streams (similar to batches) and is processed accordingly.
* Spark streaming supports highly scalable, fault-tolerant continuous stream processing which is mostly used in cases like fraud detection, website monitoring, website click baits, IoT (Internet of Things) sensors, etc.
* Spark Streaming first divides the data from the data stream into batches of X seconds which are called Dstreams or Discretized Streams. They are internally nothing but a sequence of multiple RDDs. The Spark application does the task of processing these RDDs using various Spark APIs and the results of this processing are again returned as batches. The following diagram explains the workflow of the spark streaming process.



### 16. Write a spark program to check if a given keyword exists in a huge text file or not?

def keywordExists(line):

if (line.find(“my\_keyword”) > -1):

return 1

return 0

lines = sparkContext.textFile(“test\_file.txt”);

isExist = lines.map(keywordExists);

sum = isExist.reduce(sum);

print(“Found” if sum>0 else “Not Found”)

### 17. What can you say about Spark Datasets?

Spark Datasets are those data structures of SparkSQL that provide JVM objects with all the benefits (such as data manipulation using lambda functions) of RDDs alongside Spark SQL-optimised execution engine. This was introduced as part of Spark since version 1.6.

* Spark datasets are strongly typed structures that represent the structured queries along with their encoders.
* They provide type safety to the data and also give an object-oriented programming interface.
* The datasets are more structured and have the lazy query expression which helps in triggering the action. Datasets have the combined powers of both RDD and Dataframes. Internally, each dataset symbolizes a logical plan which informs the computational query about the need for data production. Once the logical plan is analyzed and resolved, then the physical query plan is formed that does the actual query execution.

Datasets have the following features:

* **Optimized Query feature**: Spark datasets provide optimized queries using Tungsten and Catalyst Query Optimizer frameworks. The Catalyst Query Optimizer represents and manipulates a data flow graph (graph of expressions and relational operators). The Tungsten improves and optimizes the speed of execution of Spark job by emphasizing the hardware architecture of the Spark execution platform.
* **Compile-Time Analysis**: Datasets have the flexibility of analyzing and checking the syntaxes at the compile-time which is not technically possible in RDDs or Dataframes or the regular SQL queries.
* **Interconvertible**: The type-safe feature of datasets can be converted to “untyped” Dataframes by making use of the following methods provided by the Datasetholder:
  + toDS():Dataset[T]
  + toDF():DataFrame
  + toDF(columName:String\*):DataFrame
* **Faster Computation:** Datasets implementation are much faster than those of the RDDs which helps in increasing the system performance.
* **Persistent storage qualified**: Since the datasets are both queryable and serializable, they can be easily stored in any persistent storages.
* **Less Memory Consumed**: Spark uses the feature of caching to create a more optimal data layout. Hence, less memory is consumed.
* **Single Interface Multiple Languages**: Single API is provided for both Java and Scala languages. These are widely used languages for using Apache Spark. This results in a lesser burden of using libraries for different types of inputs.

### 18. Define Spark DataFrames.

Spark Dataframes are the distributed collection of datasets organized into columns similar to SQL. It is equivalent to a table in the relational database and is mainly optimized for big data operations.  
Dataframes can be created from an array of data from different data sources such as external databases, existing RDDs, Hive Tables, etc. Following are the features of Spark Dataframes:

* Spark Dataframes have the ability of processing data in sizes ranging from Kilobytes to Petabytes on a single node to large clusters.
* They support different data formats like CSV, Avro, elastic search, etc, and various storage systems like HDFS, Cassandra, MySQL, etc.
* By making use of SparkSQL catalyst optimizer, state of art optimization is achieved.
* It is possible to easily integrate Spark Dataframes with major Big Data tools using SparkCore.

### 19. Define Executor Memory in Spark

The applications developed in Spark have the same fixed cores count and fixed heap size defined for spark executors. The heap size refers to the memory of the Spark executor that is controlled by making use of the property spark.executor.memory that belongs to the -executor-memory flag. Every Spark applications have one allocated executor on each worker node it runs. The executor memory is a measure of the memory consumed by the worker node that the application utilizes.

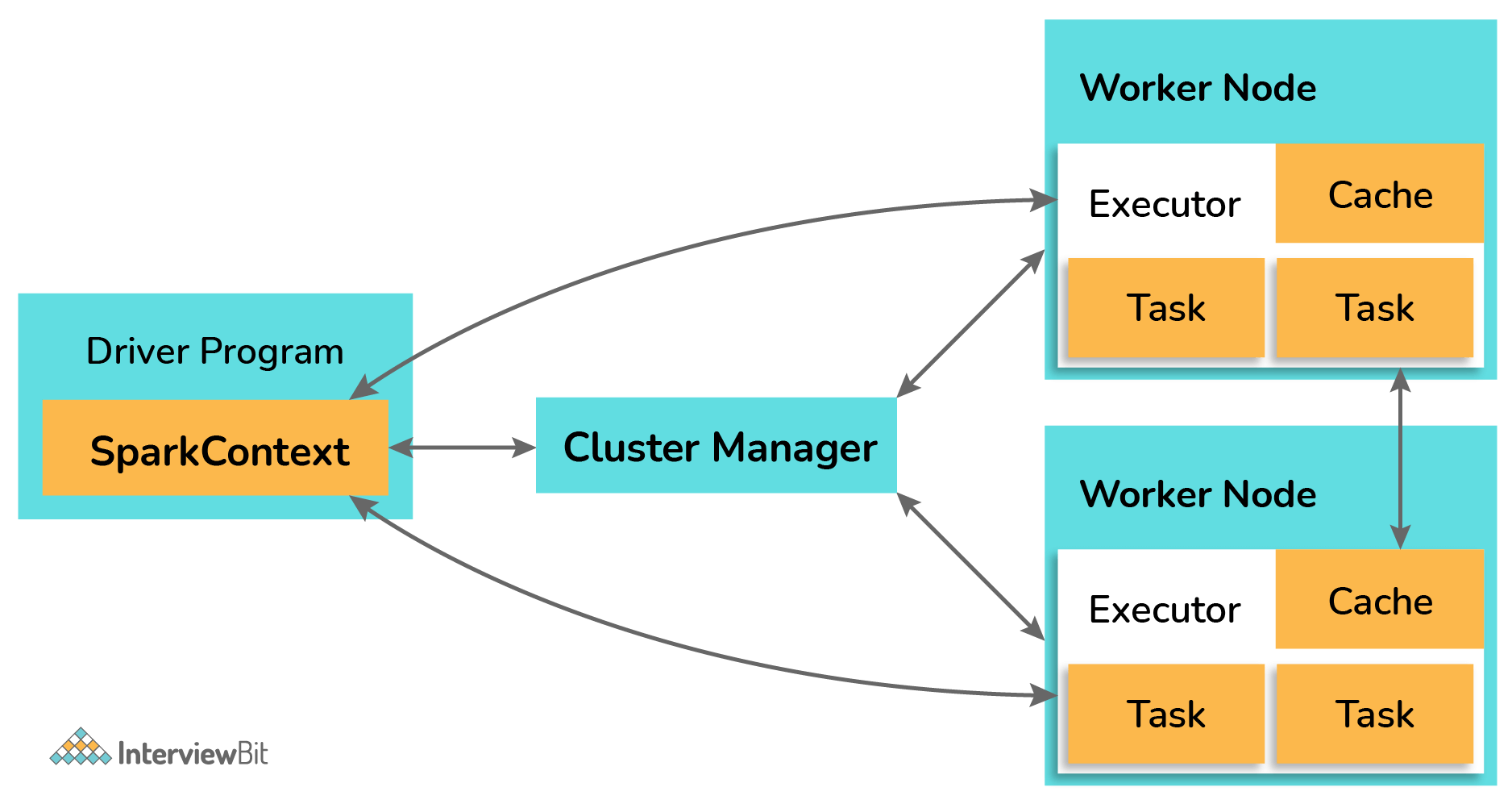
### 20. What are the functions of SparkCore?

SparkCore is the main engine that is meant for large-scale distributed and parallel data processing. The Spark core consists of the distributed execution engine that offers various APIs in Java, Python, and Scala for developing distributed ETL applications.  
Spark Core does important functions such as memory management, job monitoring, fault-tolerance, storage system interactions, job scheduling, and providing support for all the basic I/O functionalities. There are various additional libraries built on top of Spark Core which allows diverse workloads for SQL, streaming, and machine learning. They are responsible for:

* Fault recovery
* Memory management and Storage system interactions
* Job monitoring, scheduling, and distribution
* Basic I/O functions

### 21. What do you understand by worker node?

Worker nodes are those nodes that run the Spark application in a cluster. The Spark driver program listens for the incoming connections and accepts them from the executors addresses them to the worker nodes for execution. A worker node is like a slave node where it gets the work from its master node and actually executes them. The worker nodes do data processing and report the resources used to the master. The master decides what amount of resources needs to be allocated and then based on their availability, the tasks are scheduled for the worker nodes by the master.



### 22. What are some of the demerits of using Spark in applications?

Despite Spark being the powerful data processing engine, there are certain demerits to using Apache Spark in applications. Some of them are:

* Spark makes use of more storage space when compared to MapReduce or Hadoop which may lead to certain memory-based problems.
* Care must be taken by the developers while running the applications. The work should be distributed across multiple clusters instead of running everything on a single node.
* Since Spark makes use of “in-memory” computations, they can be a bottleneck to cost-efficient big data processing.
* While using files present on the path of the local filesystem, the files must be accessible at the same location on all the worker nodes when working on cluster mode as the task execution shuffles between various worker nodes based on the resource availabilities. The files need to be copied on all worker nodes or a separate network-mounted file-sharing system needs to be in place.
* One of the biggest problems while using Spark is when using a large number of small files. When Spark is used with Hadoop, we know that HDFS gives a limited number of large files instead of a large number of small files. When there is a large number of small gzipped files, Spark needs to uncompress these files by keeping them on its memory and network. So large amount of time is spent in burning core capacities for unzipping the files in sequence and performing partitions of the resulting RDDs to get data in a manageable format which would require extensive shuffling overall. This impacts the performance of Spark as much time is spent preparing the data instead of processing them.
* Spark doesn’t work well in multi-user environments as it is not capable of handling many users concurrently.

### 23. How can the data transfers be minimized while working with Spark?

The user needs to minimize data transfers and avoid shuffling to write Spark programs that run in a fast and reliable manner. There are a number of ways to minimize data transfers in Apache Spark:  
• Using Broadcast Variable- The broadcast variable enhances the efficiency of joins between small and large RDDs.  
• Using Accumulators – Accumulators help to update the values of variables in parallel while executing.  
The most common way to minimize data transfer is to avoid operations ByKey, repartition or any other operations which trigger any form of shuffles.

### ****Q.What are the various types of shared variable in Apache Spark?****

* There are two types of shared variables available in [**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/):  
  (1) **Accumulators**: used to Aggregate the Information.  
  (2) **Broadcast variable**: to efficiently distribute large values.
* When we pass the function to Spark, say filter(), this function can use the variable which defined outside of the function but within the Driver program but when we submit the task to Cluster, each worker node gets a new copy of variables and update from these variables not propagated back to Driver program.
* Accumulators and Broadcast variable are used to remove above drawback ( i.e. we can get the updated values back to our Driver program)

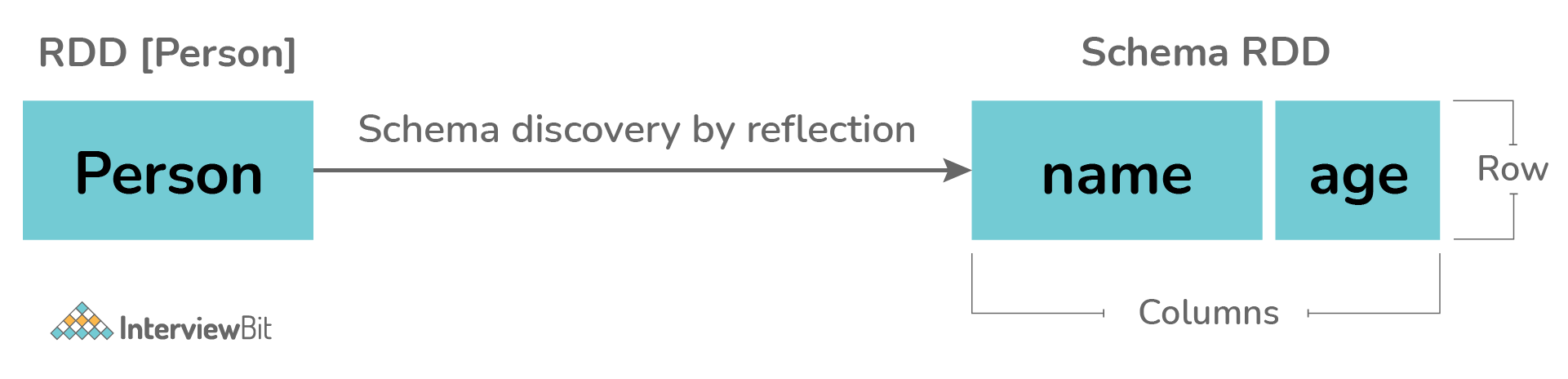
### Q. Define Partitions in Apache Spark.

**Answer:** A partition is a smaller and logical division of data. It is a logical chunk of a large distributed data set. Partitioning is the process to derive logical units of data to speed up the processing process.  
Spark manages data using partitions that help to parallelize distributed data processing with minimal network traffic for sending data between executors. By default, Spark tries to read data into an RDD from the nodes that are close to it. Since Spark usually accesses distributed partitioned data, to optimize transformation operations it creates partitions to hold the data chunks.

### 24. What is SchemaRDD in Spark RDD?

SchemaRDD is an RDD consisting of row objects that are wrappers around integer arrays or strings that has schema information regarding the data type of each column. They were designed to ease the lives of developers while debugging the code and while running unit test cases on the SparkSQL modules. They represent the description of the RDD which is similar to the schema of relational databases. SchemaRDD also provides the basic functionalities of the common RDDs along with some relational query interfaces of SparkSQL.

Consider an example. If you have an RDD named Person that represents a person’s data. Then SchemaRDD represents what data each row of Person RDD represents. If the Person has attributes like name and age, then they are represented in SchemaRDD.

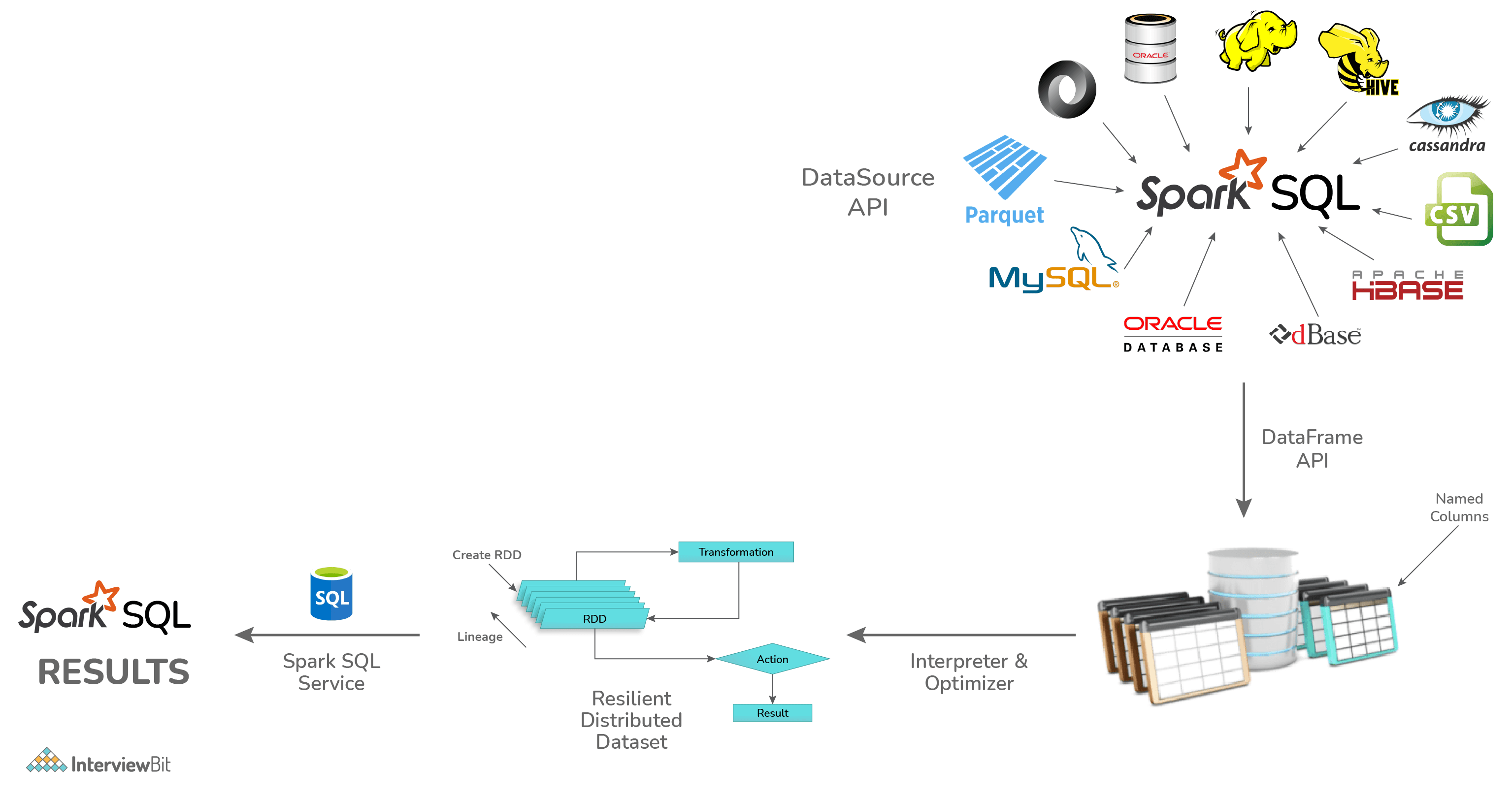


### 25. What module is used for implementing SQL in Apache Spark?

Spark provides a powerful module called SparkSQL which performs relational data processing combined with the power of the functional programming feature of Spark. This module also supports either by means of SQL or Hive Query Language. It also provides support for different data sources and helps developers write powerful SQL queries using code transformations.  
The four major libraries of SparkSQL are:

* Data Source API
* DataFrame API
* Interpreter & Catalyst Optimizer
* SQL Services

Spark SQL supports the usage of structured and semi-structured data in the following ways:

* Spark supports DataFrame abstraction in various languages like Python, Scala, and Java along with providing good optimization techniques.
* SparkSQL supports data read and writes operations in various structured formats like JSON, Hive, Parquet, etc.
* SparkSQL allows data querying inside the Spark program and via external tools that do the JDBC/ODBC connections.
* It is recommended to use SparkSQL inside the Spark applications as it empowers the developers to load the data, query the data from databases and write the results to the destination.
* 

### 26. What are the different persistence levels in Apache Spark?

Spark persists intermediary data from different shuffle operations automatically. But it is recommended to call the persist() method on the RDD. There are different persistence levels for storing the RDDs on memory or disk or both with different levels of replication. The persistence levels available in Spark are:

* **MEMORY\_ONLY**: This is the default persistence level and is used for storing the RDDs as the deserialized version of Java objects on the JVM. In case the RDDs are huge and do not fit in the memory, then the partitions are not cached and they will be recomputed as and when needed.
* **MEMORY\_AND\_DISK**: The RDDs are stored again as deserialized Java objects on JVM. In case the memory is insufficient, then partitions not fitting on the memory will be stored on disk and the data will be read from the disk as and when needed.
* **MEMORY\_ONLY\_SER**: The RDD is stored as serialized Java Objects as One Byte per partition.
* **MEMORY\_AND\_DISK\_SER**: This level is similar to MEMORY\_ONLY\_SER but the difference is that the partitions not fitting in the memory are saved on the disk to avoid recomputations on the fly.
* **DISK\_ONLY**: The RDD partitions are stored only on the disk.
* **OFF\_HEAP**: This level is the same as the MEMORY\_ONLY\_SER but here the data is stored in the off-heap memory.

The syntax for using persistence levels in the persist() method is:

df.persist(StorageLevel.<level\_value>)

The following table summarizes the details of persistence levels:

| **Persistence Level** | **Space Consumed** | **CPU time** | **In-memory?** | **On-disk?** |
| --- | --- | --- | --- | --- |
| MEMORY\_ONLY | High | Low | Yes | No |
| MEMORY\_ONLY\_SER | Low | High | Yes | No |
| MEMORY\_AND\_DISK | High | Medium | Some | Some |
| MEMORY\_AND\_DISK\_SER | Low | High | Some | Some |
| DISK\_ONLY | Low | High | No | Yes |
| OFF\_HEAP | Low | High | Yes (but off-heap) | No |

### 27. What are the steps to calculate the executor memory?

Consider you have the below details regarding the cluster:

Number of nodes = 10

Number of cores in each node = 15 cores

RAM of each node = 61GB

To identify the number of cores, we follow the approach:

Number of Cores = number of concurrent tasks that can be run parallelly by the executor. The optimal value as part of a general rule of thumb is 5.

Hence to calculate the number of executors, we follow the below approach:

Number of executors = Number of cores/Concurrent Task

= 15/5

= 3

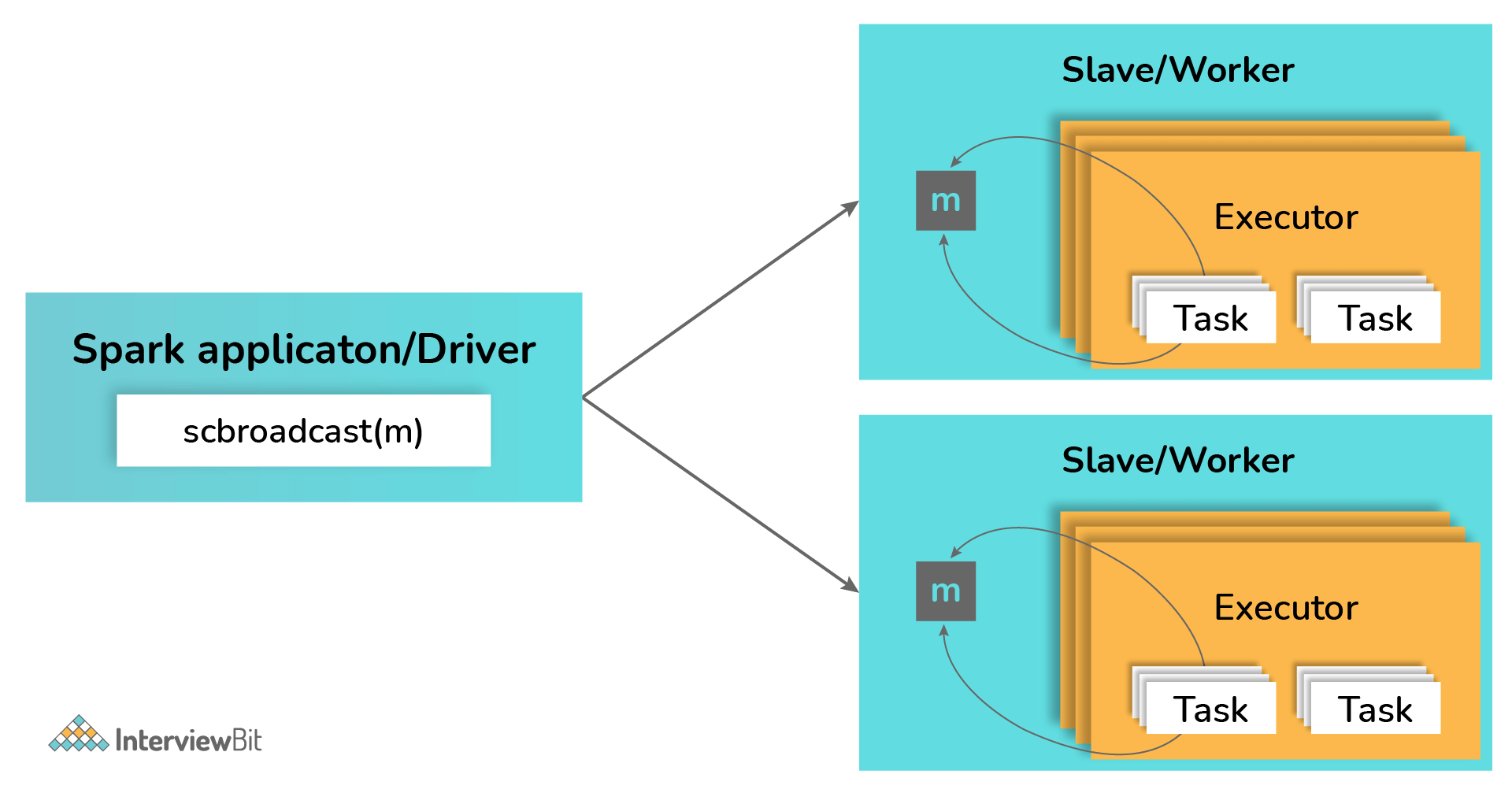
Number of executors = Number of nodes \* Number of executor in each node

= 10 \* 3

= 30 executors per Spark job

### 28. Why do we need broadcast variables in Spark?

Broadcast variables let the developers maintain read-only variables cached on each machine instead of shipping a copy of it with tasks. They are used to give every node copy of a large input dataset efficiently. These variables are broadcasted to the nodes using different algorithms to reduce the cost of communication.

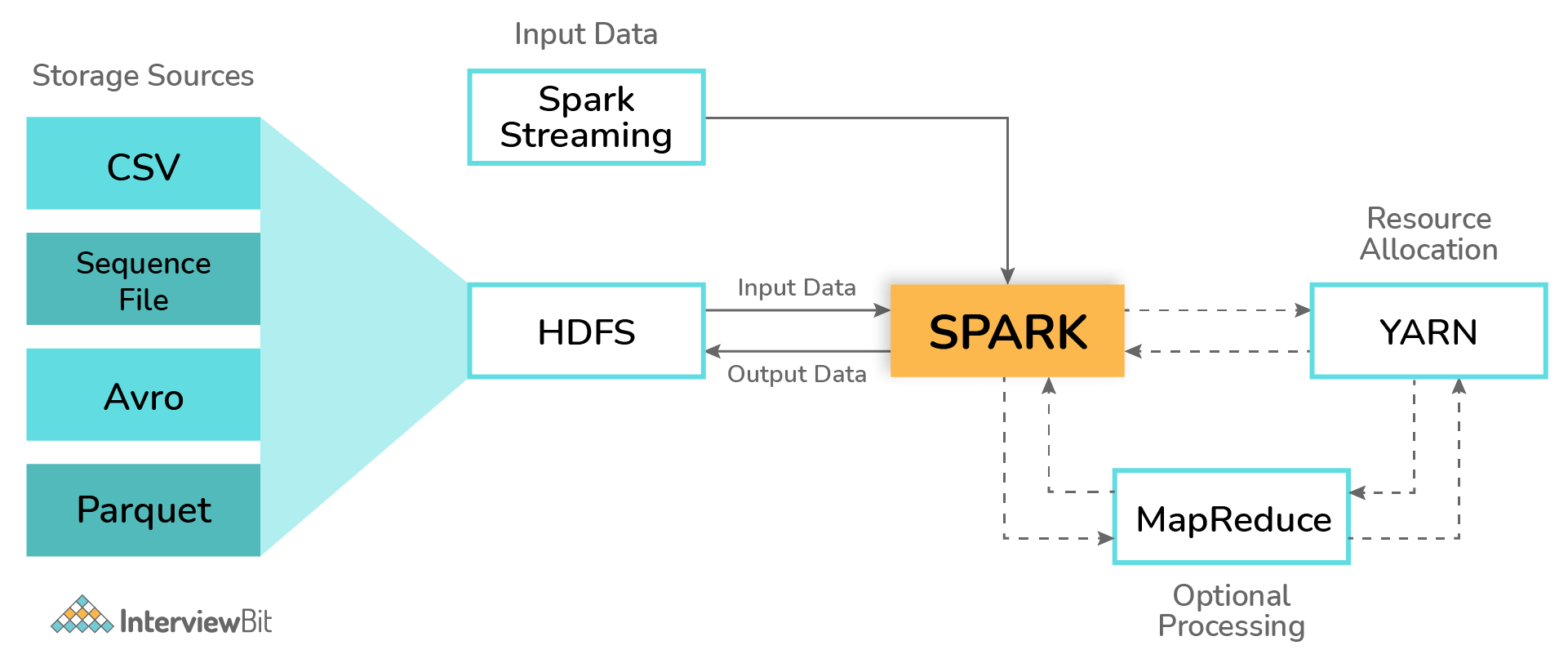


### 29. Differentiate between Spark Datasets, Dataframes and RDDs.

| **Criteria** | **Spark Datasets** | **Spark Dataframes** | **Spark RDDs** |
| --- | --- | --- | --- |
| **Representation of Data** | Spark Datasets is a combination of Dataframes and RDDs with features like static type safety and object-oriented interfaces. | Spark Dataframe is a distributed collection of data that is organized into named columns. | Spark RDDs are a distributed collection of data without schema. |
| **Optimization** | Datasets make use of catalyst optimizers for optimization. | Dataframes also makes use of catalyst optimizer for optimization. | There is no built-in optimization engine. |
| **Schema Projection** | Datasets find out schema automatically using SQL Engine. | Dataframes also find the schema automatically. | Schema needs to be defined manually in RDDs. |
| **Aggregation Speed** | Dataset aggregation is faster than RDD but slower than Dataframes. | Aggregations are faster in Dataframes due to the provision of easy and powerful APIs. | RDDs are slower than both the Dataframes and the Datasets while performing even simple operations like data grouping. |

### 30. Can Apache Spark be used along with Hadoop? If yes, then how?

Yes! The main feature of Spark is its compatibility with Hadoop. This makes it a powerful framework as using the combination of these two helps to leverage the processing capacity of Spark by making use of the best of Hadoop’s YARN and HDFS features.



Hadoop can be integrated with Spark in the following ways:

* **HDFS**: Spark can be configured to run atop HDFS to leverage the feature of distributed replicated storage.
* **MapReduce**: Spark can also be configured to run alongside the MapReduce in the same or different processing framework or Hadoop cluster. Spark and MapReduce can be used together to perform real-time and batch processing respectively.
* **YARN**: Spark applications can be configured to run on YARN which acts as the cluster management framework.

### 31. What are Sparse Vectors? How are they different from dense vectors?

Sparse vectors consist of two parallel arrays where one array is for storing indices and the other for storing values. These vectors are used to store non-zero values for saving space.

val sparseVec: Vector = Vectors.sparse(5, Array(0, 4), Array(1.0, 2.0))

* In the above example, we have the vector of size 5, but the non-zero values are there only at indices 0 and 4.
* Sparse vectors are particularly useful when there are very few non-zero values. If there are cases that have only a few zero values, then it is recommended to use dense vectors as usage of sparse vectors would introduce the overhead of indices which could impact the performance.
* Dense vectors can be defines as follows:

val denseVec = Vectors.dense(4405d,260100d,400d,5.0,4.0,198.0,9070d,1.0,1.0,2.0,0.0)

* Usage of sparse or dense vectors does not impact the results of calculations but when used inappropriately, they impact the memory consumed and the speed of calculation.

### 32. How are automatic clean-ups triggered in Spark for handling the accumulated metadata?

The clean-up tasks can be triggered automatically either by setting spark.cleaner.ttl parameter or by doing the batch-wise division of the long-running jobs and then writing the intermediary results on the disk.

### 33. How is Caching relevant in Spark Streaming?

Spark Streaming involves the division of data stream’s data into batches of X seconds called DStreams. These DStreams let the developers cache the data into the memory which can be very useful in case the data of DStream is used for multiple computations. The caching of data can be done using the cache() method or using persist() method by using appropriate persistence levels. The default persistence level value for input streams receiving data over the networks such as Kafka, Flume, etc is set to achieve data replication on 2 nodes to accomplish fault tolerance.

* Caching using cache method:

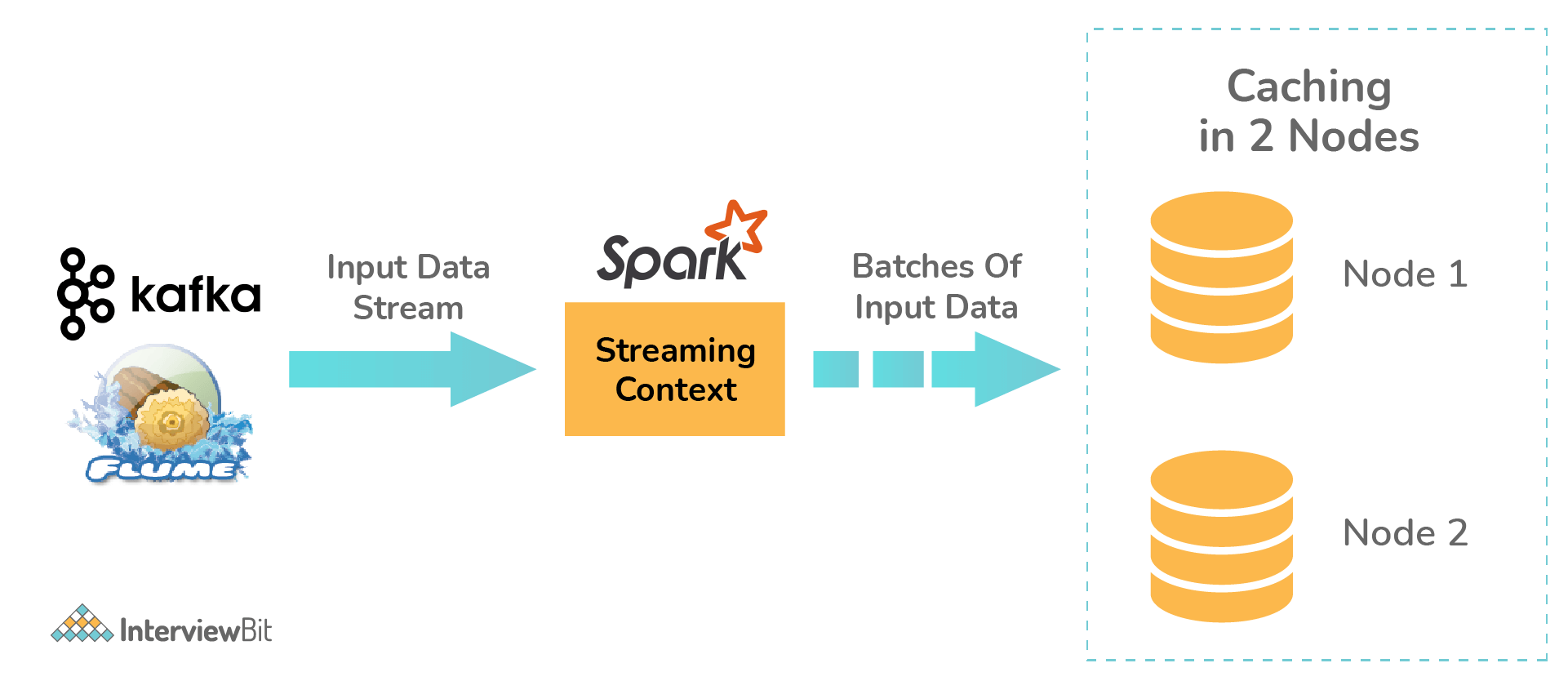
val cacheDf = dframe.cache()

* Caching using persist method:

val persistDf = dframe.persist(StorageLevel.MEMORY\_ONLY)

The main advantages of caching are:

* **Cost efficiency**: Since Spark computations are expensive, caching helps to achieve reusing of data and this leads to reuse computations which can save the cost of operations.
* **Time-efficient**: The computation reusage leads to saving a lot of time.
* **More Jobs Achieved**: By saving time of computation execution, the worker nodes can perform/execute more jobs.

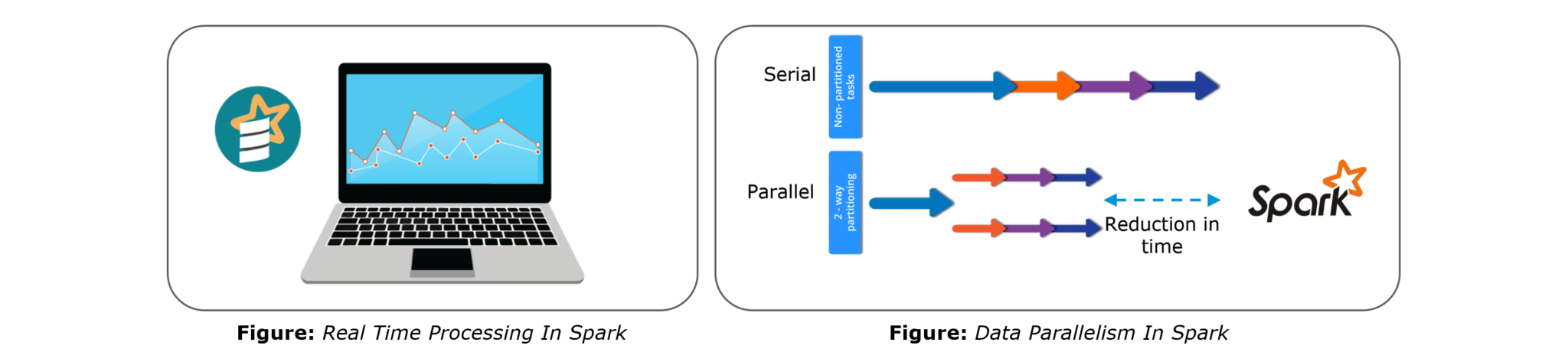


### 34. Define Piping in Spark.

Apache Spark provides the pipe() method on RDDs which gives the opportunity to compose different parts of occupations that can utilize any language as needed as per the UNIX Standard Streams. Using the pipe() method, the RDD transformation can be written which can be used for reading each element of the RDD as String. These can be manipulated as required and the results can be displayed as String.

### ****2. What is Apache Spark?****

* [***Apache Spark***](https://www.edureka.co/blog/spark-tutorial/) is an open-source cluster computing framework for real-time processing.
* It has a thriving open-source community and is the most active Apache project at the moment.
* Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.

Spark is of the most successful projects in the Apache Software Foundation. Spark has clearly evolved as the market leader for Big Data processing. Many organizations run Spark on clusters with thousands of nodes. Today, Spark is being adopted by major players like Amazon, eBay, and Yahoo!

### ****3. Explain the key features of Apache Spark.****

The following are the key features of Apache Spark:

1. **Polyglot**
2. **Speed**
3. **Multiple Format Support**
4. **Lazy Evaluation**
5. **Real Time Computation**
6. **Hadoop Integration**
7. **Machine Learning**

Let us look at these features in detail:

1. **Polyglot**: Spark provides high-level APIs in Java, Scala, Python and R. Spark code can be written in any of these four languages. It provides a shell in Scala and Python. The Scala shell can be accessed through **./bin/spark-shell** and Python shell through **./bin/pyspark** from the installed directory.
2. **Speed**: Spark runs upto 100 times faster than Hadoop MapReduce for large-scale data processing. Spark is able to achieve this speed through controlled partitioning. It manages data using partitions that help parallelize distributed data processing with minimal network traffic.
3. **Multiple Formats**: Spark supports multiple data sources such as Parquet, JSON, Hive and Cassandra. The Data Sources API provides a pluggable mechanism for accessing structured data though Spark SQL. Data sources can be more than just simple pipes that convert data and pull it into Spark.
4. **Lazy Evaluation**: Apache Spark delays its evaluation till it is absolutely necessary. This is one of the key factors contributing to its speed. For transformations, Spark adds them to a DAG of computation and only when the driver requests some data, does this DAG actually gets executed.
5. **Real Time Computation**: Spark’s computation is real-time and has less latency because of its in-memory computation. Spark is designed for massive scalability and the Spark team has documented users of the system running production clusters with thousands of nodes and supports several computational models.
6. **Hadoop Integration**: Apache Spark provides smooth compatibility with Hadoop. This is a great boon for all the Big Data engineers who started their careers with Hadoop. Spark is a potential replacement for the MapReduce functions of Hadoop, while Spark has the ability to run on top of an existing Hadoop cluster using YARN for resource scheduling.
7. **Machine Learning**: Spark’s MLlib is the machine learning component which is handy when it comes to big data processing. It eradicates the need to use multiple tools, one for processing and one for machine learning. Spark provides data engineers and data scientists with a powerful, unified engine that is both fast and easy to use.

### ****Q. Why is transformation lazy operation in Apache Spark RDD? How is it useful?****

Lazy evaluation means that [Spark](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) does not evaluate each [transformation](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/) as they arrive, but instead queues them together and evaluate all at once, as an Action is called.

The benefit of this approach is that Spark can make optimization decisions after it had a chance to look at the [**DAG**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/) in entirety. This would not be possible if it were to execute everything as soon as it got it. As a result, a large volume of Network I/O can be avoided which, otherwise, could have caused a serious bottleneck.

**Example:**  
Suppose we have a file **words.txt** containing the following lines:

line1 word1

line2 word2 word1

line3 word3 word4

line4 word1

Next, we apply the following operations.

scala> val lines = sc.textFile("words.txt")

scala> val filtered = lines.filter(line => line.contains("word1"))

scala> filtered.first()

res0: String = line1 word1

If Spark were to evaluate each line immediately, it would end up reading the whole file, then applying a filter transformation and then displaying the first line from the filtered result. This would mean a lot of extra work and unnecessary memory utilization.

On the other hand, in Lazy evaluation mode, Spark first builds the entire DAG and then, using optimization techniques it understands that reading the entire file is not necessary. The same result can be achieved by just reading the first line of the file.

### ****4. What are the languages supported by Apache Spark and which is the most popular one?****

Apache Spark supports the following four languages: Scala, Java, Python and R. Among these languages, Scala and Python have interactive shells for Spark. The Scala shell can be accessed through **./bin/spark-shell**and the Python shell through **./bin/pyspark**. Scala is the most used among them because Spark is written in Scala and it is the most popularly used for Spark.

### ****5. What are benefits of Spark over MapReduce?****

Spark has the following benefits over MapReduce:

1. Due to the availability of in-memory processing, Spark implements the processing around 10 to 100 times faster than Hadoop MapReduce whereas MapReduce makes use of persistence storage for any of the data processing tasks.
2. Unlike Hadoop, Spark provides inbuilt libraries to perform multiple tasks from the same core like batch processing, Steaming, Machine learning, Interactive SQL queries. However, Hadoop only supports batch processing.
3. Hadoop is highly disk-dependent whereas Spark promotes caching and in-memory data storage.
4. Spark is capable of performing computations multiple times on the same dataset. This is called iterative computation while there is no iterative computing implemented by Hadoop.

### ****6. What is YARN?****

Similar to Hadoop, YARN is one of the key features in Spark, providing a central and resource management platform to deliver scalable operations across the cluster. YARN is a distributed container manager, like Mesos for example, whereas Spark is a data processing tool. Spark can run on YARN, the same way Hadoop Map Reduce can run on YARN. Running Spark on YARN necessitates a binary distribution of Spark as built on YARN support.

### ****7. Do you need to install Spark on all nodes of YARN cluster?****

No, because Spark runs on top of YARN. Spark runs independently from its installation. Spark has some options to use YARN when dispatching jobs to the cluster, rather than its own built-in manager, or Mesos. Further, there are some configurations to run YARN. They include master, deploy-mode, driver-memory, executor-memory, executor-cores, and queue.

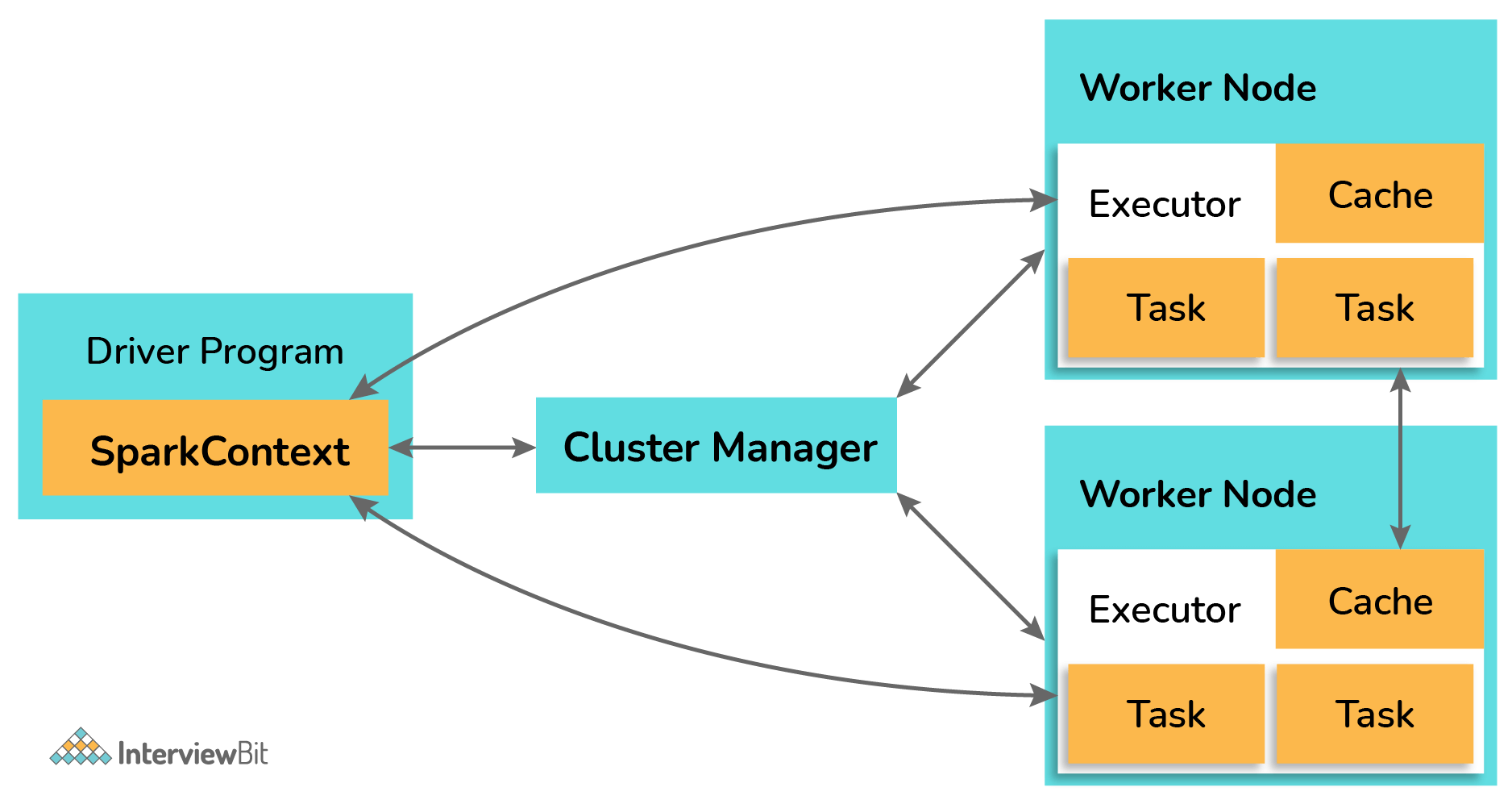
### ****8. Is there any benefit of learning MapReduce if Spark is better than MapReduce?****

Yes, MapReduce is a paradigm used by many big data tools including Spark as well. It is extremely relevant to use MapReduce when the data grows bigger and bigger. Most tools like Pig and Hive convert their queries into MapReduce phases to optimize them better.

**Q.List the types of Deploy Modes in Spark.**

There are 2 deploy modes in Spark. They are:

* **Client Mode**: The deploy mode is said to be in client mode when the spark driver component runs on the machine node from where the spark job is submitted.
  + The main disadvantage of this mode is if the machine node fails, then the entire job fails.
  + This mode supports both interactive shells or the job submission commands.
  + The performance of this mode is worst and is not preferred in production environments.
* **Cluster Mode**: If the spark job driver component does not run on the machine from which the spark job has been submitted, then the deploy mode is said to be in cluster mode.
  + The spark job launches the driver component within the cluster as a part of the sub-process of ApplicationMaster.
  + This mode supports deployment only using the spark-submit command (interactive shell mode is not supported).
  + Here, since the driver programs are run in ApplicationMaster, in case the program fails, the driver program is re-instantiated.
  + In this mode, there is a dedicated cluster manager (such as stand-alone, YARN, Apache Mesos, Kubernetes, etc) for allocating the resources required for the job to run as shown in the below architecture.



Apart from the above two modes, if we have to run the application on our local machines for unit testing and development, the deployment mode is called “**Local Mode**”. Here, the jobs run on a single JVM in a single machine which makes it highly inefficient as at some point or the other there would be a shortage of resources which results in the failure of jobs. It is also not possible to scale up resources in this mode due to the restricted memory and space

### Q.What is the difference between repartition and coalesce?

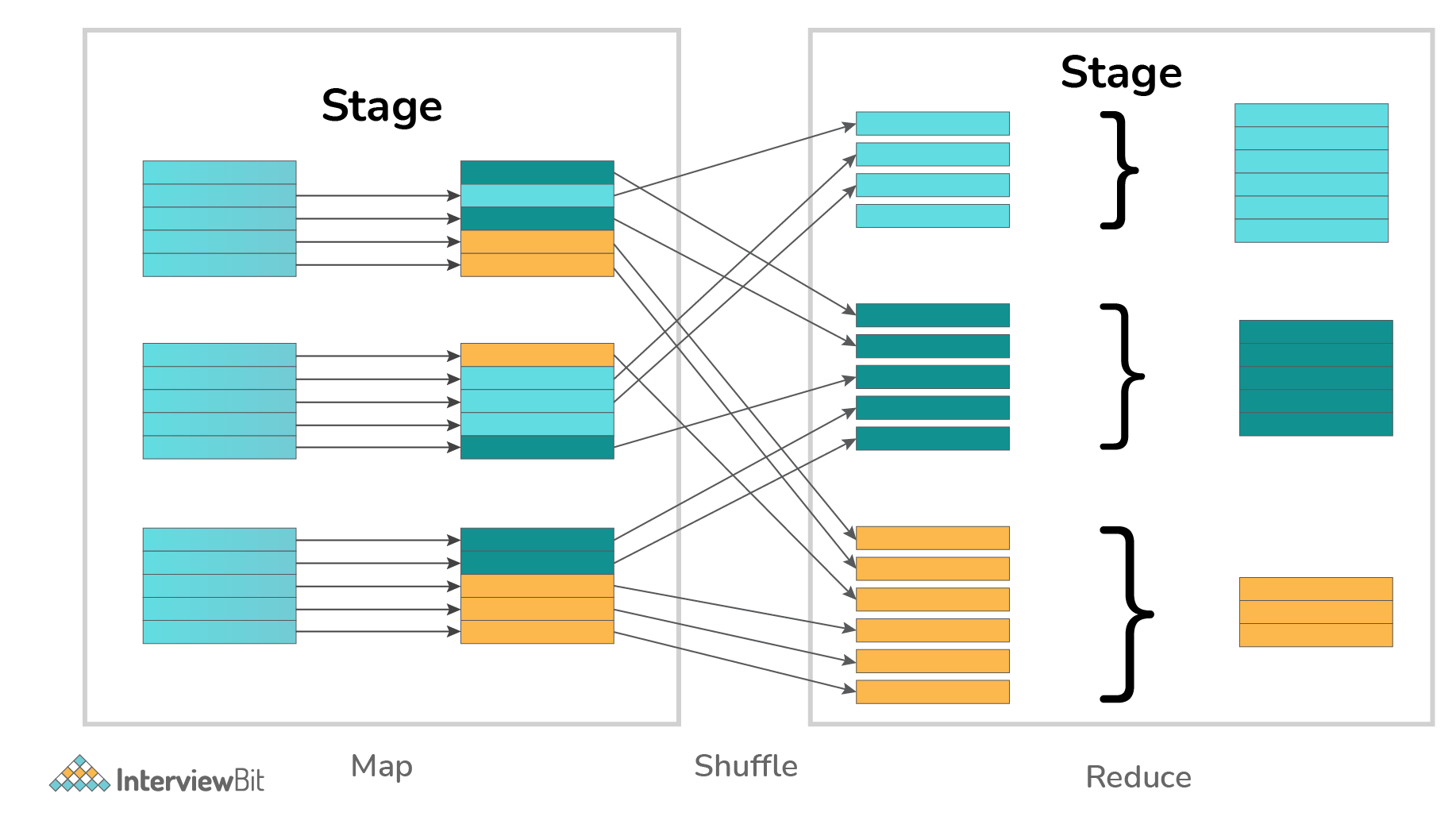
| **Repartition** | **Coalesce** |
| --- | --- |
| Usage repartition can increase/decrease the number of data partitions. | Spark coalesce can only reduce the number of data partitions. |
| Repartition creates new data partitions and performs a full shuffle of evenly distributed data. | Coalesce makes use of already existing partitions to reduce the amount of shuffled data unevenly. |
| Repartition internally calls coalesce with shuffle parameter thereby making it slower than coalesce. | Coalesce is faster than repartition. However, if there are unequal-sized data partitions, the speed might be slightly slower. |

### Q. What are the data formats supported by Spark?

Spark supports both the raw files and the structured file formats for efficient reading and processing. File formats like paraquet, JSON, XML, CSV, RC, Avro, TSV, etc are supported by Spark.

### Q. What do you understand by Shuffling in Spark?

The process of redistribution of data across different partitions which might or might not cause data movement across the JVM processes or the executors on the separate machines is known as shuffling/repartitioning. Partition is nothing but a smaller logical division of data.



It is to be noted that Spark has no control over what partition the data gets distributed across.

### ****9. Explain the concept of Resilient Distributed Dataset (RDD).****

RDD stands for Resilient Distribution Datasets. An RDD is a fault-tolerant collection of operational elements that run in parallel. The partitioned data in RDD is immutable and distributed in nature. There are primarily two types of RDD:

1. Parallelized Collections: Here, the existing RDDs running parallel with one another.
2. Hadoop Datasets: They perform functions on each file record in HDFS or other storage systems.

RDDs are basically parts of data that are stored in the memory distributed across many nodes. RDDs are lazily evaluated in Spark. This lazy evaluation is what contributes to Spark’s speed.

### ****10. How do we create RDDs in Spark?****

Spark provides two methods to create RDD:

1. By parallelizing a collection in your Driver program.

2. This makes use of SparkContext’s ‘parallelize’

|  |  |
| --- | --- |
| 1  2  3 | method val DataArray = Array(2,4,6,8,10)    val DataRDD = sc.parallelize(DataArray) |

3. By loading an external dataset from external storage like HDFS, HBase, shared file system.

### ****11. What is Executor Memory in a Spark application?****

Every spark application has same fixed heap size and fixed number of cores for a spark executor. The heap size is what referred to as the Spark executor memory which is controlled with the spark.executor.memory property of the **–executor-memory** flag. Every spark application will have one executor on each worker node. The executor memory is basically a measure on how much memory of the worker node will the application utilize.

### ****12. Define Partitions in Apache Spark.****

As the name suggests, partition is a smaller and logical division of data similar to ‘split’ in MapReduce. It is a logical chunk of a large distributed data set. Partitioning is the process to derive logical units of data to speed up the processing process. Spark manages data using partitions that help parallelize distributed data processing with minimal network traffic for sending data between executors. By default, Spark tries to read data into an RDD from the nodes that are close to it. Since Spark usually accesses distributed partitioned data, to optimize transformation operations it creates partitions to hold the data chunks. Everything in Spark is a partitioned RDD.

### ****13. What operations does RDD support?****

RDD (Resilient Distributed Dataset) is main logical data unit in Spark. An RDD has distributed a collection of objects. Distributed means, each RDD is divided into multiple partitions. Each of these partitions can reside in memory or stored on the disk of different machines in a cluster. RDDs are immutable (Read Only) data structure. You can’t change original RDD, but you can always transform it into different RDD with all changes you want.

RDDs support two types of operations: transformations and actions.

Transformations: Transformations create new RDD from existing RDD like map, reduceByKey and filter we just saw. Transformations are executed on demand. That means they are computed lazily.

Actions: Actions return final results of RDD computations. Actions triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system.

### ****14. What do you understand by Transformations in Spark?****

Transformations are functions applied on RDD, resulting into another RDD. It does not execute until an action occurs. map() and filter() are examples of transformations, where the former applies the function passed to it on each element of RDD and results into another RDD. The filter() creates a new RDD by selecting elements from current RDD that pass function argument.

Ex: filter(), union()

|  |  |
| --- | --- |
| 1  2  3 | val rawData=sc.textFile("path to/movies.txt")  val moviesData=rawData.map(x=>x.split("  ")) |

As we can see here, rawData RDD is transformed into moviesData RDD. Transformations are lazily evaluated.

### ****15. Define Actions in Spark.****

An action helps in bringing back the data from RDD to the local machine. An action’s execution is the result of all previously created transformations. Actions triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system.

reduce() is an action that implements the function passed again and again until one value if left. take() action takes all the values from RDD to a local node.

|  |  |
| --- | --- |
|  | moviesData.saveAsTextFile(“MoviesData.txt”) |

As we can see here, moviesData RDD is saved into a text file called MoviesData.txt.

### ****16. Define functions of SparkCore.****

Spark Core is the base engine for large-scale parallel and distributed data processing. The core is the distributed execution engine and the Java, Scala, and Python APIs offer a platform for distributed ETL application development. SparkCore performs various important functions like memory management, monitoring jobs, fault-tolerance, job scheduling and interaction with storage systems. Further, additional libraries, built atop the core allow diverse workloads for streaming, SQL, and machine learning. It is responsible for:

1. Memory management and fault recovery
2. Scheduling, distributing and monitoring jobs on a cluster
3. Interacting with storage systems

### ****17. What do you understand by Pair RDD?****

Apache defines PairRDD functions class as

|  |  |
| --- | --- |
| 1 | class PairRDDFunctions[K, V] extends Logging with HadoopMapReduceUtil with Serializable |

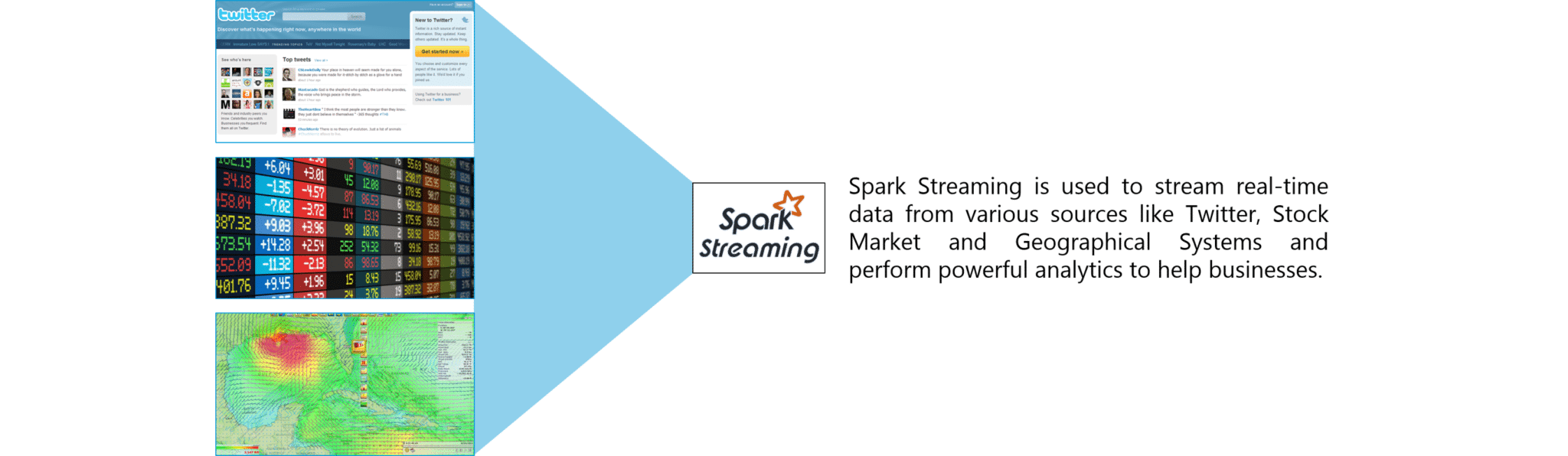
Special operations can be performed on RDDs in Spark using key/value pairs and such RDDs are referred to as Pair RDDs. Pair RDDs allow users to access each key in parallel. They have a reduceByKey() method that collects data based on each key and a join() method that combines different RDDs together, based on the elements having the same key.

### ****18. Name the components of Spark Ecosystem.****

1. **Spark Core**: Base engine for large-scale parallel and distributed data processing
2. **Spark Streaming**: Used for processing real-time streaming data
3. **Spark SQL**: Integrates relational processing with Spark’s functional programming API
4. **GraphX**: Graphs and graph-parallel computation
5. **MLlib**: Performs machine learning in Apache Spark

### ****19. How is Streaming implemented in Spark? Explain with examples.****

Spark Streaming is used for processing real-time streaming data. Thus it is a useful addition to the core Spark API. It enables high-throughput and fault-tolerant stream processing of live data streams. The fundamental stream unit is DStream which is basically a series of RDDs (Resilient Distributed Datasets) to process the real-time data. The data from different sources like Flume, HDFS is streamed and finally processed to file systems, live dashboards and databases. It is similar to batch processing as the input data is divided into streams like batches.

****

**Figure:** Spark Interview Questions – Spark Streaming

### ****23. Is there a module to implement SQL in Spark? How does it work?****

Spark SQL is a new module in Spark which integrates relational processing with Spark’s functional programming API. It supports querying data either via SQL or via the Hive Query Language. For those of you familiar with RDBMS, Spark SQL will be an easy transition from your earlier tools where you can extend the boundaries of traditional relational data processing.

Spark SQL integrates relational processing with Spark’s functional programming. Further, it provides support for various data sources and makes it possible to weave SQL queries with code transformations thus resulting in a very powerful tool.

The following are the four libraries of Spark SQL.

1. Data Source API
2. DataFrame API
3. Interpreter & Optimizer
4. SQL Service

### Spark SQL - Spark Interview Questions - Edureka****24. What is a Parquet file?****

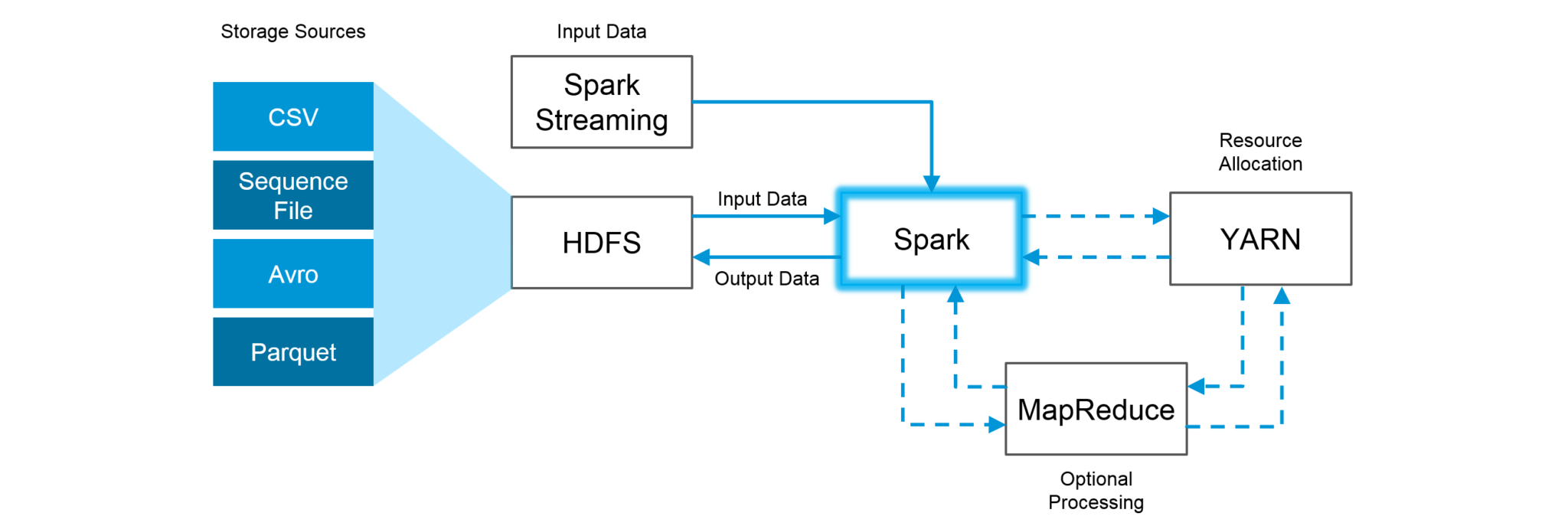
Parquet is a columnar format file supported by many other data processing systems. Spark SQL performs both read and write operations with Parquet file and consider it be one of the best big data analytics formats so far.

Parquet is a columnar format, supported by many data processing systems. The advantages of having a columnar storage are as follows:

1. Columnar storage limits IO operations.
2. It can fetch specific columns that you need to access.
3. Columnar storage consumes less space.
4. It gives better-summarized data and follows type-specific encoding.

### ****25. How can Apache Spark be used alongside Hadoop?****

The best part of Apache Spark is its compatibility with Hadoop. As a result, this makes for a very powerful combination of technologies. Here, we will be looking at how Spark can benefit from the best of Hadoop. Using Spark and Hadoop together helps us to leverage Spark’s processing to utilize the best of Hadoop’s HDFS and YARN.

**Figure:**Using Spark and Hadoop

Hadoop components can be used alongside Spark in the following ways:

1. **HDFS**: Spark can run on top of HDFS to leverage the distributed replicated storage.
2. **MapReduce**: Spark can be used along with MapReduce in the same Hadoop cluster or separately as a processing framework.
3. **YARN**: Spark applications can also be run on YARN (Hadoop NextGen).
4. **Batch & Real Time Processing**: MapReduce and Spark are used together where MapReduce is used for batch processing and Spark for real-time processing.

### ****26. What is RDD Lineage?****

Spark does not support data replication in the memory and thus, if any data is lost, it is rebuild using RDD lineage. RDD lineage is a process that reconstructs lost data partitions. The best is that RDD always remembers how to build from other datasets.

### ****27. What is Spark Driver?****

Spark Driver is the program that runs on the master node of the machine and declares transformations and actions on data RDDs. In simple terms, a driver in Spark creates SparkContext, connected to a given Spark Master.  
The driver also delivers the RDD graphs to Master, where the standalone cluster manager runs.

### ****28. What file systems does Spark support?****

The following three file systems are supported by Spark:

1. Hadoop Distributed File System (HDFS).
2. Local File system.
3. Amazon S3

### ****29. List the functions of Spark SQL.****

Spark SQL is capable of:

1. Loading data from a variety of structured sources.
2. Querying data using SQL statements, both inside a Spark program and from external tools that connect to Spark SQL through standard database connectors (JDBC/ODBC). For instance, using business intelligence tools like Tableau.
3. Providing rich integration between SQL and regular Python/Java/Scala code, including the ability to join RDDs and SQL tables, expose custom functions in SQL, and more.

### ****30. What is Spark Executor?****

When SparkContext connects to a cluster manager, it acquires an Executor on nodes in the cluster. Executors are Spark processes that run computations and store the data on the worker node. The final tasks by SparkContext are transferred to executors for their execution.

### ****Q:What do you mean by PySpark SparkContext?****

Ans. In simple words, an entry point to any spark functionality is what we call SparkContext. While it comes to **PySpark, SparkContext** uses Py4J(library) in order to launch a JVM. In this way, it creates a JavaSparkContext. However, PySpark has SparkContext available as ‘sc’, by default.

### ****Q. Explain PySpark SparkConf?****

Ans. Mainly, we use **SparkConf** because we need to set a few configurations and parameters to run a Spark application on the local/cluster. In other words, SparkConf offers configurations to run a Spark application.

### ****Q.Name parameter of SparkContext?****

Ans. The parameters of a SparkContext are:

* **Master** − URL of the cluster from which it connects.
* **appName** − Name of our job.
* **sparkHome** − Spark installation directory.
* **pyFiles** − It is the .zip or .py files, in order to send to the cluster and also to add to the PYTHONPATH.
* **Environment** − Worker nodes environment variables.
* **Serializer** − RDD serializer.
* **Conf** − to set all the Spark properties, an object of L{SparkConf}.
* **JSC** − It is the JavaSparkContext instance.

### ****Q.Name attributes of SparkConf.****

Ans. **Attributes of SparkConf** −

1. set(key, value) − This attribute helps to set a configuration property.
2. setMaster(value) − It helps to set the master URL.
3. setAppName(value) − This helps to set an application name.
4. get(key, defaultValue=None) − This attribute helps to get a configuration value of a key.
5. setSparkHome(value) − It helps to set Spark installation path on worker nodes.

**In what ways SparkSession different from SparkContext?**

[**Spark Context:**](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/)  
Prior to [**Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) 2.0.0 sparkContext was used as a channel to access all spark functionality.  
The spark driver program uses spark context to connect to the cluster through a resource manager ([**YARN**](http://data-flair.training/blogs/category/yarn/) orMesos..).  
sparkConf is required to create the spark context object, which stores configuration parameter like appName (to identify your spark driver), application, number of core and memory size of executor running on worker node.

In order to use APIs of [**SQL**](http://data-flair.training/blogs/spark-sql-tutorial/)**,**[**HIVE**](http://data-flair.training/blogs/category/hive/)**, and**[**Streaming**](http://data-flair.training/blogs/apache-spark-streaming-comprehensive-guide/), separate contexts need to be created.

**Example:**  
creating sparkConf :

val conf = new SparkConf().setAppName(“RetailDataAnalysis”).setMaster(“spark://master:7077”).set(“spark.executor.memory”, “2g”)

creation of sparkContext:

val sc = new SparkContext(conf)

**Spark Session:**

SPARK 2.0.0 onwards, SparkSession provides a single point of entry to interact with underlying Spark functionality and  
allows programming Spark with **[DataFrame](http://data-flair.training/blogs/apache-spark-dataframe-tutorial/)** and Dataset APIs. All the functionality available with sparkContext are also available in sparkSession.

In order to use APIs of SQL, HIVE, and Streaming, no need to create separate contexts as sparkSession includes all the APIs.

Once the SparkSession is instantiated, we can configure Spark’s run-time config properties.

**Example:**

Creating Spark session:  
val spark = SparkSession  
.builder  
.appName(“WorldBankIndex”)  
.getOrCreate()

Configuring properties:  
spark.conf.set(“spark.sql.shuffle.partitions”, 6)  
spark.conf.set(“spark.executor.memory”, “2g”)

Spark 2.0.0 onwards, it is better to use sparkSession as it provides access to all the spark Functionalities that sparkContext does. Also, it provides APIs to work on DataFrames and Datasets.

### ****31. Name types of Cluster Managers in Spark.****

The Spark framework supports three major types of Cluster Managers:

1. **Standalone**: A basic manager to set up a cluster.
2. **Apache Mesos**: Generalized/commonly-used cluster manager, also runs Hadoop MapReduce and other applications.
3. **YARN**: Responsible for resource management in Hadoop.

### ****32. What do you understand by worker node?****

Worker node refers to any node that can run the application code in a cluster. The driver program must listen for and accept incoming connections from its executors and must be network addressable from the worker nodes.

Worker node is basically the slave node. Master node assigns work and worker node actually performs the assigned tasks. Worker nodes process the data stored on the node and report the resources to the master. Based on the resource availability, the master schedule tasks.

### ****33. Illustrate some demerits of using Spark.****

The following are some of the demerits of using Apache Spark:

1. Since Spark utilizes more storage space compared to Hadoop and MapReduce, there may arise certain problems.
2. Developers need to be careful while running their applications in Spark.
3. Instead of running everything on a single node, the work must be distributed over multiple clusters.
4. Spark’s “in-memory” capability can become a bottleneck when it comes to cost-efficient processing of big data.
5. Spark consumes a huge amount of data when compared to Hadoop.

### ****34. List some use cases where Spark outperforms Hadoop in processing.****

1. **Sensor Data Processing**: Apache Spark’s “In-memory” computing works best here, as data is retrieved and combined from different sources.
2. **Real Time Processing**: Spark is preferred over Hadoop for real-time querying of data. e.g. Stock Market Analysis, Banking, Healthcare, Telecommunications, etc.
3. **Stream Processing**: For processing logs and detecting frauds in live streams for alerts, Apache Spark is the best solution.
4. **Big Data Processing**:Spark runs upto 100 times faster than Hadoop when it comes to processing medium and large-sized datasets.

### ****36. Can you use Spark to access and analyze data stored in Cassandra databases?****

Yes, it is possible if you use Spark Cassandra Connector.To connect Spark to a Cassandra cluster, a Cassandra Connector will need to be added to the Spark project. In the setup, a Spark executor will talk to a local Cassandra node and will only query for local data. It makes queries faster by reducing the usage of the network to send data between Spark executors (to process data) and Cassandra nodes (where data lives).

### ****37. Is it possible to run Apache Spark on Apache Mesos?****

Yes, Apache Spark can be run on the hardware clusters managed by Mesos. In a standalone cluster deployment, the cluster manager in the below diagram is a Spark master instance. When using Mesos, the Mesos master replaces the Spark master as the cluster manager. Mesos determines what machines handle what tasks. Because it takes into account other frameworks when scheduling these many short-lived tasks, multiple frameworks can coexist on the same cluster without resorting to a static partitioning of resources.

### ****38. How can Spark be connected to Apache Mesos?****

To connect Spark with Mesos:

1. Configure the spark driver program to connect to Mesos.
2. Spark binary package should be in a location accessible by Mesos.
3. Install Apache Spark in the same location as that of Apache Mesos and configure the property ‘spark.mesos.executor.home’ to point to the location where it is installed.

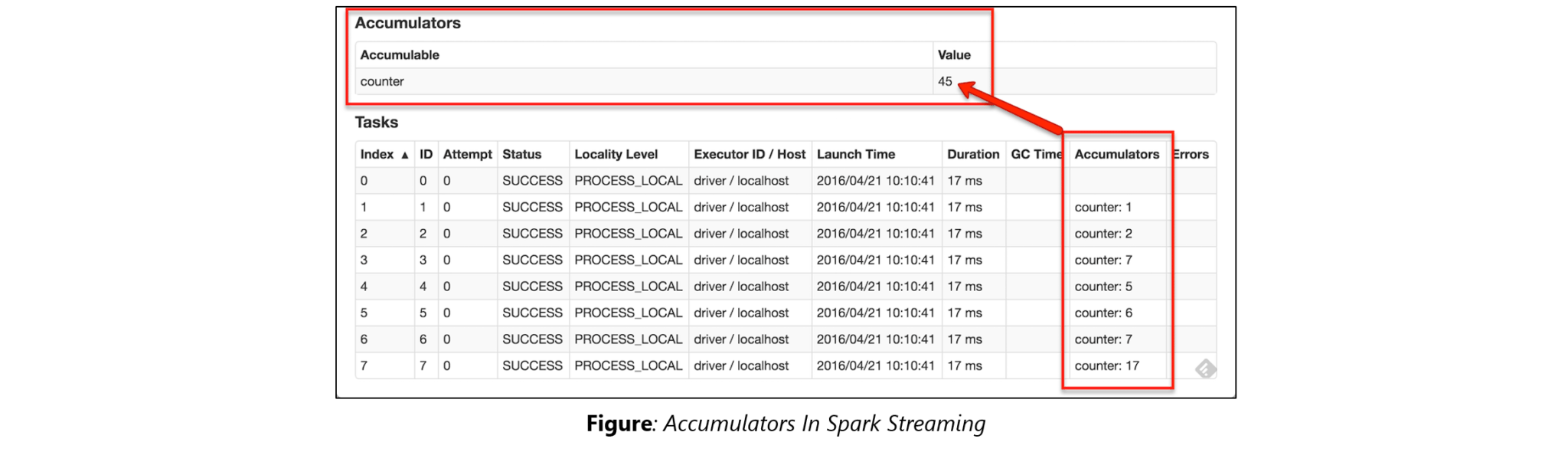
### ****40. What are broadcast variables?****

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

### Broadcast Variables - Spark Interview Questions - Edureka

### ****41. Explain accumulators in Apache Spark.****

Accumulators are variables that are only added through an associative and commutative operation. They are used to implement counters or sums. Tracking accumulators in the UI can be useful for understanding the progress of running stages. Spark natively supports numeric accumulators. We can create named or unnamed accumulators.



#### Creating Accumulator Variable

Below is an example of how to create an accumulator variable “**accum**” of type int and using it to sum all values in an RDD.

accum=sc.accumulator(0)

rdd=spark.sparkContext.parallelize([1,2,3,4,5])

rdd.foreach(lambda x:accum.add(x))

print(accum.value) #Accessed by driver

Here, we have created an accumulator variable **accum** using **spark.sparkContext.accumulator(0)** with initial value 0. Later, we are [iterating each element in an rdd using foreach() action](https://sparkbyexamples.com/spark/spark-foreach-usage-with-examples/) and adding each element of rdd to accum variable. Finally, we are getting accumulator value using **accum.value** property.

Note that, In this example, rdd.foreach() is executed on workers and accum.value is called from PySpark driver program.

### ****42. Why is there a need for broadcast variables when working with Apache Spark?****

Broadcast variables are read only variables, present in-memory cache on every machine. When working with Spark, usage of broadcast variables eliminates the necessity to ship copies of a variable for every task, so data can be processed faster. Broadcast variables help in storing a lookup table inside the memory which enhances the retrieval efficiency when compared to an RDD lookup().

### ****43. How can you trigger automatic clean-ups in Spark to handle accumulated metadata?****

You can trigger the clean-ups by setting the parameter ‘spark.cleaner.ttl’ or by dividing the long running jobs into different batches and writing the intermediary results to the disk.

### ****44. What is the significance of Sliding Window operation?****

Sliding Window controls transmission of data packets between various computer networks. Spark Streaming library provides windowed computations where the transformations on RDDs are applied over a sliding window of data. Whenever the window slides, the RDDs that fall within the particular window are combined and operated upon to produce new RDDs of the windowed DStream.

### DStream Sliding Window - Spark Interview Questions - Edureka****45. What is a DStream in Apache Spark?****

***Discretized Stream***(DStream) is the basic abstraction provided by Spark Streaming. It is a continuous stream of data. It is received from a data source or from a processed data stream generated by transforming the input stream. Internally, a DStream is represented by a continuous series of RDDs and each RDD contains data from a certain interval. Any operation applied on a DStream translates to operations on the underlying RDDs.

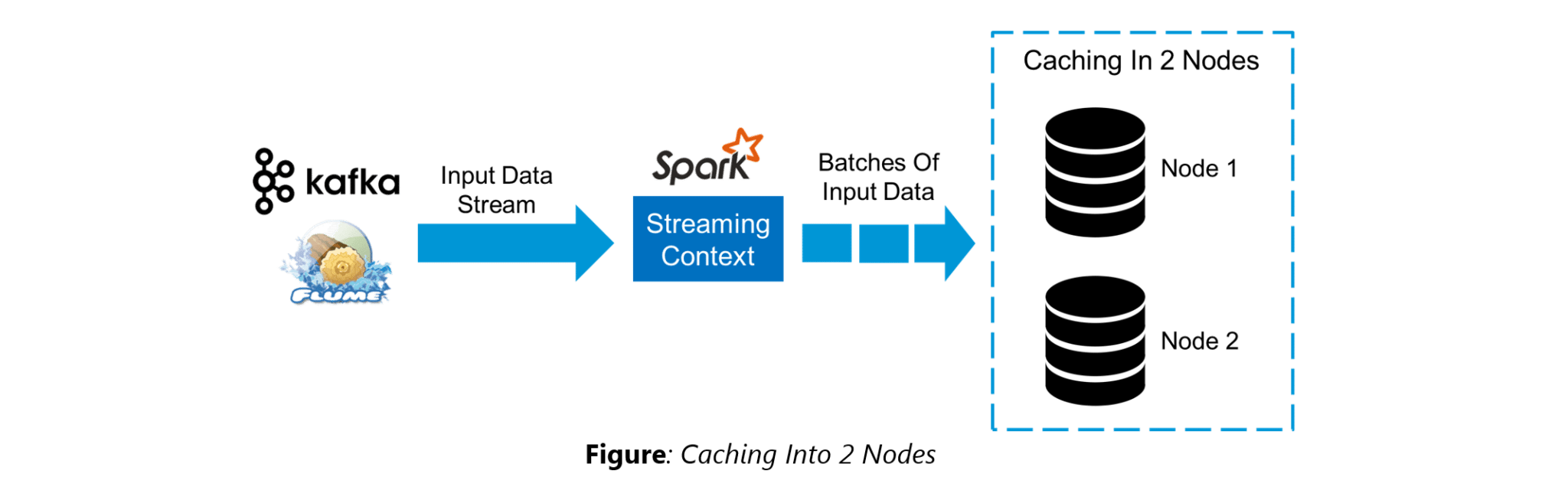
DStreams can be created from various sources like Apache Kafka, HDFS, and Apache Flume. DStreams have two operations:

1. Transformations that produce a new DStream.
2. Output operations that write data to an external system.

There are many DStream transformations possible in Spark Streaming. Let us look at **filter(*func*)**. filter(*func*) returns a new DStream by selecting only the records of the source DStream on which func returns true.

### DStream Filter - Spark Interview Questions - Edureka****46. Explain Caching in Spark Streaming.****

DStreams allow developers to cache/ persist the stream’s data in memory. This is useful if the data in the DStream will be computed multiple times. This can be done using the persist() method on a DStream. For input streams that receive data over the network (such as Kafka, Flume, Sockets, etc.), the default persistence level is set to replicate the data to two nodes for fault-tolerance.

****

### ****47. When running Spark applications, is it necessary to install Spark on all the nodes of YARN cluster?****

Spark need not be installed when running a job under YARN or Mesos because Spark can execute on top of YARN or Mesos clusters without affecting any change to the cluster.

### ****48. What are the various data sources available in Spark SQL?****

Parquet file, JSON datasets and Hive tables are the data sources available in Spark SQL.

### ****49. What are the various levels of persistence in Apache Spark?****

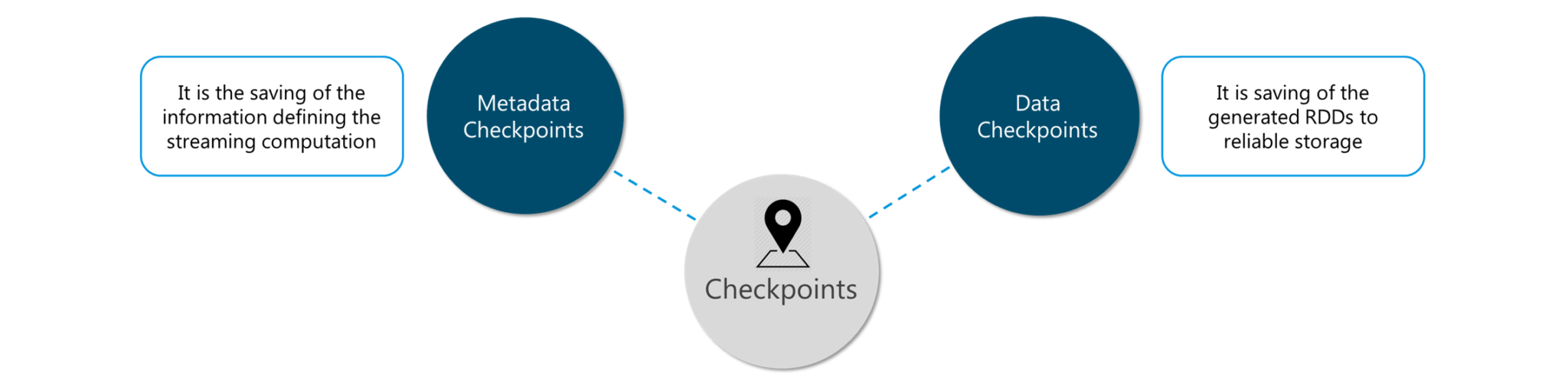
Apache Spark automatically persists the intermediary data from various shuffle operations, however, it is often suggested that users call persist () method on the RDD in case they plan to reuse it. Spark has various persistence levels to store the RDDs on disk or in memory or as a combination of both with different replication levels.

The various storage/persistence levels in Spark are:

1. MEMORY\_ONLY: Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they’re needed. This is the default level.
2. MEMORY\_AND\_DISK: Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don’t fit on disk, and read them from there when they’re needed.
3. MEMORY\_ONLY\_SER: Store RDD as *serialized* Java objects (one byte array per partition).
4. MEMORY\_AND\_DISK\_SER: Similar to MEMORY\_ONLY\_SER, but spill partitions that don’t fit in memory to disk instead of recomputing them on the fly each time they’re needed.
5. DISK\_ONLY: Store the RDD partitions only on disk.
6. OFF\_HEAP: Similar to MEMORY\_ONLY\_SER, but store the data in off-heap memory.

### ****50. Does Apache Spark provide checkpoints?****

Checkpoints are similar to checkpoints in gaming. They make it run 24/7 and make it resilient to failures unrelated to the application logic.

**Figure:** Spark Interview Questions – Checkpoints

Lineage graphs are always useful to recover RDDs from a failure but this is generally time-consuming if the RDDs have long lineage chains. Spark has an API for checkpointing i.e. a REPLICATE flag to persist. However, the decision on which data to checkpoint – is decided by the user. Checkpoints are useful when the lineage graphs are long and have wide dependencies.

### ****51. How Spark uses Akka?****

Spark uses Akka basically for scheduling. All the workers request for a task to master after registering. The master just assigns the task. Here Spark uses Akka for messaging between the workers and masters.

### ****53. What do you understand by SchemaRDD in Apache Spark RDD?****

SchemaRDD is an RDD that consists of row objects (wrappers around the basic string or integer arrays) with schema information about the type of data in each column.

SchemaRDD was designed as an attempt to make life easier for developers in their daily routines of code debugging and unit testing on SparkSQL core module. The idea can boil down to describing the data structures inside RDD using a formal description similar to the relational database schema. On top of all basic functions provided by common RDD APIs, SchemaRDD also provides some straightforward relational query interface functions that are realized through SparkSQL.

Now, it is officially renamed to DataFrame API on Spark’s latest trunk.

### ****54. How is Spark SQL different from HQL and SQL?****

Spark SQL is a special component on the Spark Core engine that supports SQL and Hive Query Language without changing any syntax. It is possible to join SQL table and HQL table to Spark SQL.

### ****Q.Explain the difference between Spark SQL and Hive.****

* Spark SQL is faster than Hive.
* Any Hive query can easily be executed in Spark SQL but vice-versa is not true.
* Spark SQL is a library whereas Hive is a framework.
* It is not mandatory to create a metastore in Spark SQL but it is mandatory to create a Hive metastore.
* Spark SQL automatically infers the schema whereas in Hive schema needs to be explicitly declared.

### ****Q.What are scalar and aggregate functions in Spark SQL?****

In Spark SQL, Scalar functions are those functions that return a single value for each row. Scalar functions include built-in functions including array functions and map functions. Aggregate functions return a single value for a group of rows. Some of the built-in aggregate functions include min(), max(), count(), countDistinct(), avg(). Users can also create their own scalar and aggregate functions.

### ****Q.Differentiate between temp view and global temp view on Spark SQL.****

* Temp views in Spark SQL are tied to the Spark session that created the view, and will no longer be available upon termination of the Spark session.
* Global temp views in Spark SQL are not tied to a particular Spark session, but can be shared across multiple Spark sessions. They are tied to a system database and can only be created and accessed using the qualified name “global\_temp”. Global temporary views remain available until the Spark session is terminated.

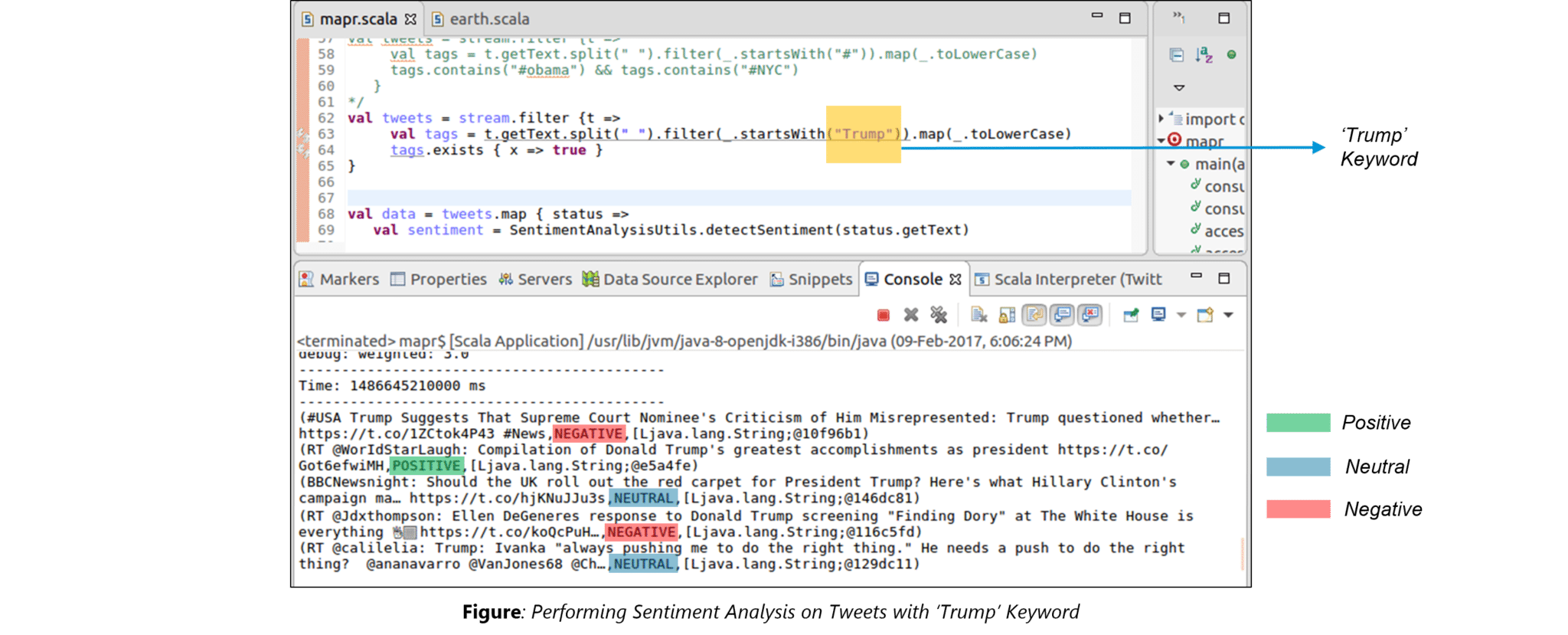
### ****55. Explain a scenario where you will be using Spark Streaming.****

When it comes to Spark Streaming, the data is streamed in real-time onto our Spark program.

Twitter Sentiment Analysis is a real-life use case of Spark Streaming. Trending Topics can be used to create campaigns and attract a larger audience. It helps in crisis management, service adjusting and target marketing.

Sentiment refers to the emotion behind a social media mention online. Sentiment Analysis is categorizing the tweets related to a particular topic and performing data mining using Sentiment Automation Analytics Tools.

Spark Streaming can be used to gather live tweets from around the world into the Spark program. This stream can be filtered using Spark SQL and then we can filter tweets based on the sentiment. The filtering logic will be implemented using MLlib where we can learn from the emotions of the public and change our filtering scale accordingly.



The above figure displays the sentiments for the tweets containing the word ‘Trump’.

### 56. What API is used for Graph Implementation in Spark?

Spark provides a powerful API called GraphX that extends Spark RDD for supporting graphs and graph-based computations. The extended property of Spark RDD is called as Resilient Distributed Property Graph which is a directed multi-graph that has multiple parallel edges. Each edge and the vertex has associated user-defined properties. The presence of parallel edges indicates multiple relationships between the same set of vertices. GraphX has a set of operators such as subgraph, mapReduceTriplets, joinVertices, etc that can support graph computation. It also includes a large collection of graph builders and algorithms for simplifying tasks related to graph analytics.

### ****Is there an API for implementing graphs in Spark?****

GraphX is the Spark API for graphs and graph-parallel computation. Thus, it extends the Spark RDD with a Resilient Distributed Property Graph.

The property graph is a directed multi-graph which can have multiple edges in parallel. Every edge and vertex have user defined properties associated with it. Here, the parallel edges allow multiple relationships between the same vertices. At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge.

To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and mapReduceTriplets) as well as an optimized variant of the Pregel API. In addition, GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.

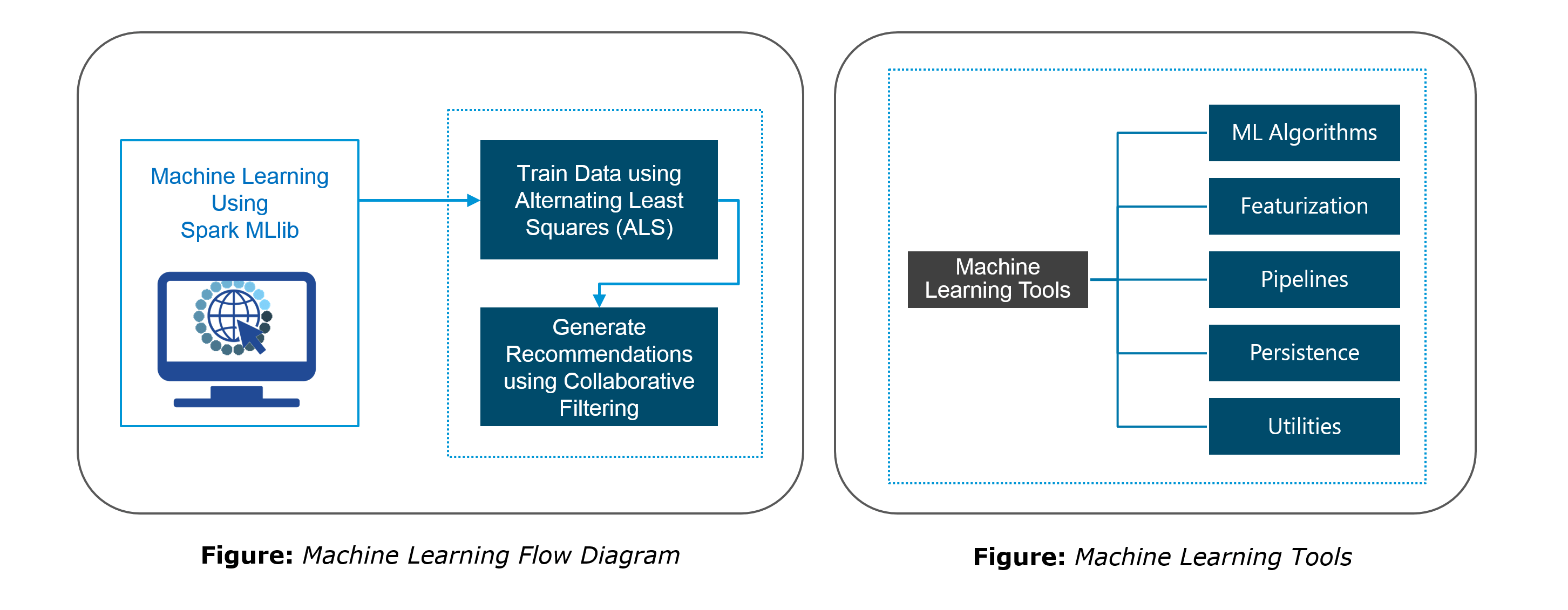
### ****21. What is PageRank in GraphX?****

PageRank measures the importance of each vertex in a graph, assuming an edge from u to v represents an endorsement of v’s importance by u. For example, if a Twitter user is followed by many others, the user will be ranked highly.

GraphX comes with static and dynamic implementations of PageRank as methods on the PageRank Object. Static PageRank runs for a fixed number of iterations, while dynamic PageRank runs until the ranks converge (i.e., stop changing by more than a specified tolerance). GraphOps allows calling these algorithms directly as methods on Graph.

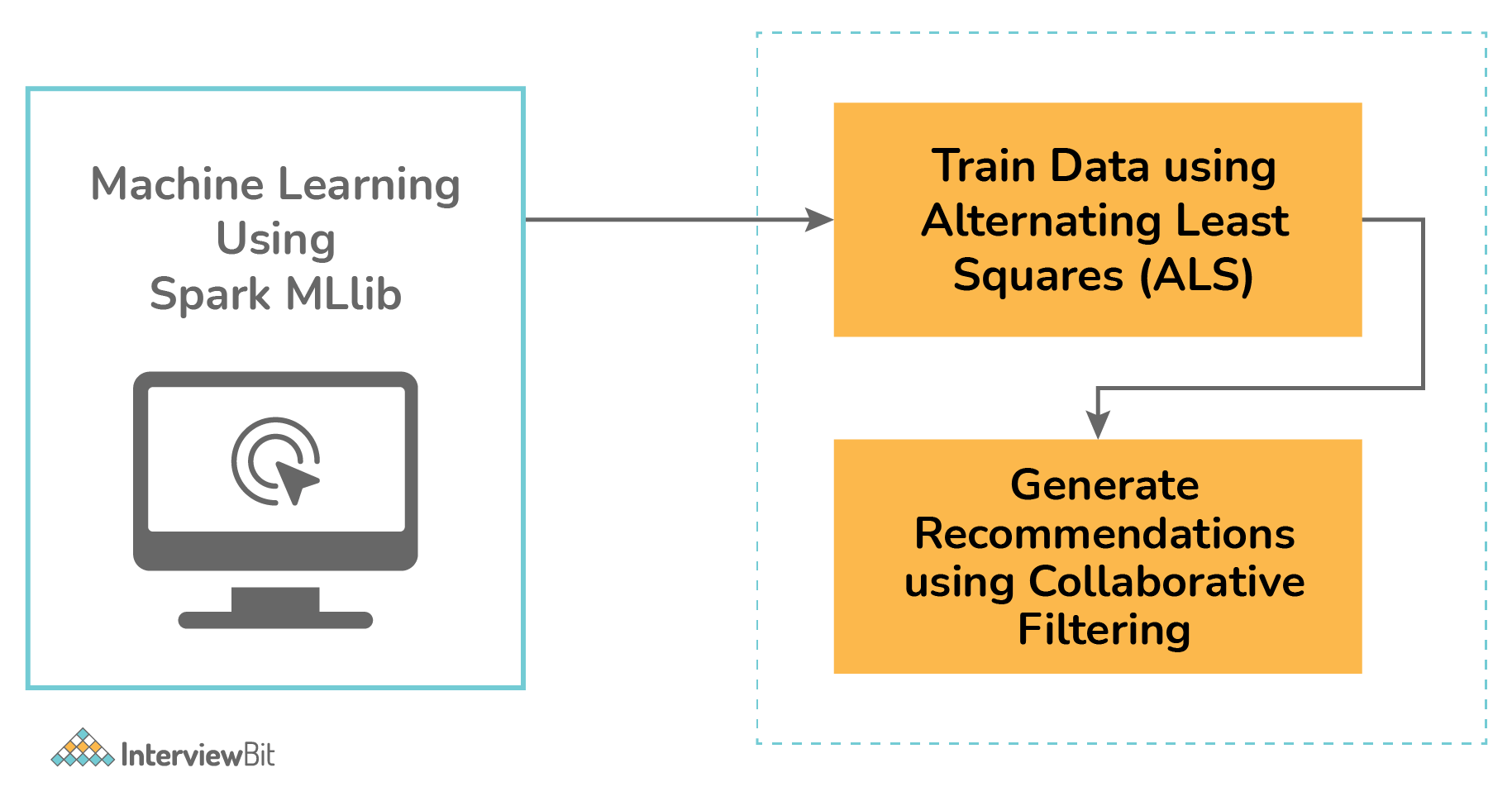
### ****22. How is machine learning implemented in Spark?****

MLlib is scalable machine learning library provided by Spark. It aims at making machine learning easy and scalable with common learning algorithms and use cases like clustering, regression filtering, dimensional reduction, and alike.

****

### 57. How can you achieve machine learning in Spark?

Spark provides a very robust, scalable machine learning-based library called MLlib. This library aims at implementing easy and scalable common ML-based algorithms and has the features like classification, clustering, dimensional reduction, regression filtering, etc. More information about this library can be obtained in detail from Spark’s official documentation site here: <https://spark.apache.org/docs/latest/ml-guide.html>



# 99 Apache Spark Interview Questions for Professional

1. What is the difference between Spark and Hadoop?   
2. What are the differences between functional and imperative languages, and why is functional programming important?   
3. What is a resilient distributed dataset (RDD), explain showing diagrams?   
4. Explain transformations and actions (in the context of RDDs)   
5. What are the Spark use cases?   
6. Why do we need transformations? What is lazy evaluation and why is it useful?   
7. What is ParallelCollectionRDD?   
8. Explain how ReduceByKey and GroupByKey work?   
9. What is the common workflow of a Spark program?

10. Explain Spark environment for driver. Ref   
11. What are the transformations and actions that you have used in Spark?   
12. How can you minimize data transfers when working with Spark?   
13. What is a lineage graph?   
14. Describe the major libraries that constitute the Spark Ecosystem   
15. What are the different file formats that can be used in SparkSql?   
16. What are Pair RDDs?  
17. What is the difference between persist() and cache()  
18. What are the various levels of persistence in Apache Spark? Ref   
19. Which Storage Level to choose?

20. Explain advantages and drawbacks of RDD

21. Explain why dataset is preferred over RDDs?   
22. How to share data from Spark RDD between two applications?   
23. Does Apache Spark provide check pointing?   
24. Explain the internal working of caching?   
25. What is the function of Block manager?   
26. Why does Spark SQL consider the support of indexes unimportant?   
27. How to convert existing UDTFs in Hive to Scala functions and use them from Spark SQL? Explain with example   
28. Why use dataframes and datasets when we have RDD? Ref   
29. What is a Catalyst and how does it work?   
30. What are the top challenges developers faces while writing Spark applications?

31. Explain the difference in implementation between DataFrames and DataSet?   
32. How is memory handled in Datasets?   
33. What are the limitations of dataset?   
34. What are the contentions with memory?

35. Show Command to run Spark in YARN client mode?

36. Show Command to run Spark in YARN cluster mode?   
37. What is Standalone and YARN mode?

38. Explain client mode and cluster mode in Spark?   
39. Which cluster managers are supported by Spark?  
40. What is Executor memory?

41. What is DStream and what is the difference between batch and Dstream in Spark streaming?

42. How does Spark Streaming work?   
43. Difference between map() and flatMap()?

44. What is reduce() action, Is there any difference between reduce() and reduceByKey()?  
45. What is the disadvantage of reduce() action and how can we overcome this limitation?  
46. What are Accumulators and when are accumulators truly reliable?

47. What is Broadcast Variables and what advantage do they provide?   
48. What is piping? Demonstrate with an example of a data pipeline.   
49. What is a driver?   
50. What does a Spark Engine do?   
51. What are the steps that occur when you run a Spark application on a cluster?   
52. What is a schema RDD/DataFrame?   
53. What are Row objects?   
54. How does Spark achieve fault tolerance?   
55. What parameter is set if cores need to be defined across executors?

56. Name few Spark Master system properties?

57. Define Partitions in reference to Spark implementation?   
58. Differences between how Spark and MapReduce manage cluster resources under YARN

59. What is GraphX and what is PageRank?   
60. What does MLlib do?   
61. What is a Parquet file?

62. Why is Parquet used for Spark SQL?

63. What is schema evolution and what is its disadvantage, explain schema merging in reference to parquet file?   
64. Will Spark replace MapReduce?   
65. What is Spark Executor?   
66. Name the different types of Cluster Managers in Spark.   
67. How many ways we can create RDDs, show example?   
68. How do you flatten rows in Spark? Explain with example.

69. What is Hive on Spark?   
70. Explain Spark Streaming Architecture? 71. What are the types of Transformations on DStreams?   
72. What is Receiver in Spark Streaming, and can you build custom receivers?   
73. Explain the process of Live streaming storing DStream data to database?   
74. How is Spark streaming fault tolerant?

75. Explain transform() method used in dSteam?   
76. What file systems does Spark support?   
77. How is data security achieved in Spark?   
78. Explain Kerberos security?   
79. Name the various types of distributing that Spark supports?   
80. Show some example queries using the Scala DataFrame API.

81. What are the conditions where Spark driver can parallelize dataSets as RDDs?   
82. Can repartition() operation decrease the number of partitions?   
83. What is the drawback of repartition() and coalesce() operations?   
84. In a join operaton for example val joinVal = rddA.join(rddB) will it generate partition?

85. Consider the following code in Spark, what is the final value in fVal variable?   
86. Scala pattern matching - Show various ways code can be written?   
87. What is the return result when a query is executed using Spark SQL or HIVE? Hint: RDD or dataframe/dataset?   
88. If we want to display just the schema of a dataframe/dataset what method is called?   
89. Show various implementations for the following query in Spark?   
90. What are the most important factors you want to consider when you start machine learning project?................... 80   
91. As a data scientist, which algorithm would you suggest if legal aspects and ease of explanation to non technical   
people are the main criteria? ...................................................................................................................................... 80   
92. For the supervised learning algorithm, what percentage of data is split between training and test dataset?   
93. Compare performance of Avro and parquet file formats and their usage (in the context of Spark)   
94. Spark Master exposes a set Rest API to submit am monitor application. Which data format is used for these web services?  
95. When you should not use Spark?   
96. Can you use Spark to access and analyze data stored in Cassandra databases? 97. With which mathematical properties can you achieve parallelism?   
98. What are various types of Partitioning in Apache Spark?   
99. How to set partitioning for data in Apache Spark?

# Hadoop

**10) What command will you use to copy data from one node in Hadoop to another?**

hdfs dfs -distcp hdfs://source\_namenode/apache\_hadoop hdfs://dest\_namenodeB/Hadoop

**11) How can you kill an application running on YARN?**

Use:

yarn application -list

To list all the applications that are running on YARN.

To kill the application that you want to kill, identify its application ID and use the following command to kill it:

yarn application -kill appid

**Suppose you want to get an HDFS file into a local directory; how would you go about it?**

There are two commands that can be used to get HDFS files into the local system:

hadoop fs -get

hadoop fs -copyToLocal

# Spark Performance

# Q.how to decide various parameters in spark submit

Resource Allocation is an important aspect during the execution of any spark job. If not configured correctly, a spark job can consume entire cluster resources and make other applications starve for resources.

This blog helps to understand the basic flow in a Spark Application and then how to configure the number of executors, memory settings of each executors and the number of cores for a Spark Job. There are a few factors that we need to consider to decide the optimum numbers for the above three, like:

* The amount of data
* The time in which a job has to complete
* Static or dynamic allocation of resources
* Upstream or downstream application

**Introduction**

Let’s start with some basic definitions of the terms used in handling Spark applications.

**Partitions** : A partition is a small chunk of a large distributed data set. Spark manages data using partitions that helps parallelize data processing with minimal data shuffle across the executors.

**Task** : A task is a unit of work that can be run on a partition of a distributed dataset and gets executed on a single executor. The unit of parallel execution is at the task level.All the tasks with-in a single stage can be executed in parallel

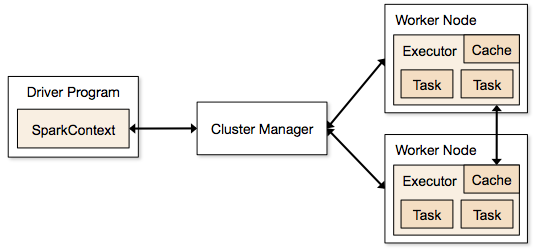
**Executor** : An executor is a single JVM process which is launched for an application on a worker node. Executor runs tasks and keeps data in memory or disk storage across them. Each application has its own executors. A single node can run multiple executors and executors for an application can span multiple worker nodes. An executor stays up for the  
duration of the Spark Application and runs the tasks in multiple threads. The number of executors for a spark application can be specified inside the SparkConf or via the flag –num-executors from command-line.

**Cluster Manager** : An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN). Spark is agnostic to a cluster manager as long as it can acquire executor processes and those can communicate with each other.We are primarily interested in Yarn as the cluster manager. A spark cluster can run in either yarn cluster or yarn-client mode:

**yarn-client mode** – A driver runs on client process, Application Master is only used for requesting resources from YARN.

**yarn-cluster mode** – A driver runs inside application master process, client goes away once the application is initialized

**Cores** : A core is a basic computation unit of CPU and a CPU may have one or more cores to perform tasks at a given time. The more cores we have, the more work we can do. In spark, this controls the number of parallel tasks an executor can run.



**Steps involved in cluster mode for a Spark Job**

1. From the driver code, SparkContext connects to cluster manager (standalone/Mesos/YARN).
2. Cluster Manager allocates resources across the other applications. Any cluster manager can be used as long as the executor processes are running and they communicate with each other.
3. Spark acquires executors on nodes in cluster. Here each application will get its own executor processes.
4. Application code (jar/python files/python egg files) is sent to executors
5. Tasks are sent by SparkContext to the executors.

From the above steps, it is clear that the number of executors and their memory setting play a major role in a spark job. Running executors with too much memory often results in excessive garbage collection delays

Now we try to understand, how to configure the best set of values to optimize a spark job.

There are two ways in which we configure the executor and core details to the Spark job. They are:

1. Static Allocation – The values are given as part of spark-submit
2. Dynamic Allocation – The values are picked up based on the requirement (size of data, amount of computations needed) and released after use. This helps the resources to be re-used for other applications.

**Static Allocation**

Different cases are discussed varying different parameters and arriving at different combinations as per user/data requirements.

**Case 1 Hardware – 6 Nodes and each node have 16 cores, 64 GB RAM**

First on each node, 1 core and 1 GB is needed for Operating System and Hadoop Daemons, so we have 15 cores, 63 GB RAM for each node

***We start with how to choose number of cores*:**

*Number of cores = Concurrent tasks an executor can run*

So we might think, more concurrent tasks for each executor will give better performance. But research shows that any application with more than 5 concurrent tasks, would lead to a bad show. So the optimal value is 5.

This number comes from the ability of an executor to run parallel tasks and not from how many cores a system has. So the number 5 stays same even if we have double (32) cores in the CPU

***Number of executors:***

Coming to the next step, with 5 as cores per executor, and 15 as total available cores in one node (CPU) – we come to 3 executors per node which is 15/5. We need to calculate the number of executors on each node and then get the total number for the job.

So with 6 nodes, and 3 executors per node – we get a total of 18 executors. Out of 18 we need 1 executor (java process) for Application Master in YARN. So final number is 17 executors

This 17 is the number we give to spark using –num-executors while running from spark-submit shell command

***Memory for each executor:***

From above step, we have 3 executors per node. And available RAM on each node is 63 GB

So memory for each executor in each node is 63/3 = 21GB.

However small overhead memory is also needed to determine the full memory request to YARN for each executor.

The formula for that overhead is max(384, .07 \* spark.executor.memory)

Calculating that overhead:  .07 \* 21 (Here 21 is calculated as above 63/3) = 1.47

Since 1.47 GB > 384 MB, the overhead is 1.47

Take the above from each 21 above => 21 – 1.47 ~ 19 GB

So executor memory – 19 GB

*Final numbers – Executors – 17, Cores 5, Executor Memory – 19 GB*

**Case 2 Hardware – 6 Nodes and Each node have 32 Cores, 64 GB**

**Number of cores** of 5 is same for good concurrency as explained above.

**Number of executors** for each node = 32/5 ~ 6

So total executors = 6 \* 6 Nodes = 36. Then final number is 36 – 1(for AM) = 35

**Executor memory**:

6 executors for each node. 63/6 ~ 10. Overhead is .07 \* 10 = 700 MB. So rounding to 1GB as overhead, we get 10-1 = 9 GB

*Final numbers – Executors – 35, Cores 5, Executor Memory – 9 GB*

**Case 3 – When more memory is not required for the executors**

The above scenarios start with accepting number of cores as fixed and moving to the number of executors and memory.

Now for the first case, if we think we do not need 19 GB, and just 10 GB is sufficient based on the data size and computations involved, then following are the numbers:

Cores: 5

Number of executors for each node = 3. Still 15/5 as calculated above.

At this stage, this would lead to 21 GB, and then 19 as per our first calculation. But since we thought 10 is ok (assume little overhead), then we cannot switch the number of executors per node to 6 (like 63/10). Because with 6 executors per node and 5 cores it comes down to 30 cores per node, when we only have 16 cores. So we also need to change number of cores for each executor.

So calculating again,

The magic number 5 comes to 3 (any number less than or equal to 5). So with 3 cores, and 15 available cores – we get 5 executors per node, 29 executors ( which is  (5\*6 -1)) and memory is 63/5 ~ 12.

Overhead is 12\*.07=.84. So executor memory is 12 – 1 GB = 11 GB

*Final Numbers are 29 executors, 3 cores, executor memory is 11 GB*

**Summary Table**

**Dynamic Allocation**

*Note: Upper bound for the number of executors if dynamic allocation is enabled is infinity. So this says that spark application can eat away all the resources if needed. In a cluster where we have other applications running and they also need cores to run the tasks, we need to make sure that we assign the cores at cluster level.*

*This means that we can allocate specific number of cores for YARN based applications based on user access. So we can create a spark\_user and then give cores (min/max) for that user. These limits are for sharing between spark and other applications which run on YARN.*

To understand dynamic allocation, we need to have knowledge of the following properties:

*spark.dynamicAllocation.enabled –*when this is set to true we need not mention executors. The reason is below:

The static parameter numbers we give at spark-submit is for the entire job duration. However if dynamic allocation comes into picture, there would be different stages like the following:

**What is the number for executors to start with:**

Initial number of executors (*spark.dynamicAllocation.initialExecutors*) to start with

**Controlling the number of executors dynamically:**

Then based on load (tasks pending) how many executors to request. This would eventually be the number what we give at spark-submit in static way. So once the initial executor numbers are set, we go to min (*spark.dynamicAllocation.minExecutors*) and max (*spark.dynamicAllocation.maxExecutors*) numbers.

**When to ask new executors or give away current executors:**

When do we request new executors (*spark.dynamicAllocation.schedulerBacklogTimeout*) – This means that there have been pending tasks for this much duration. So the request for the number of executors requested in each round increases exponentially from the previous round. For instance, an application will add 1 executor in the first round, and then 2, 4, 8 and so on executors in the subsequent rounds. At a specific point, the above property max comes into picture.

When do we give away an executor is set using *spark.dynamicAllocation.executorIdleTimeout.*

To conclude, if we need more control over the job execution time, monitor the job for unexpected data volume the static numbers would help. By moving to dynamic, the resources would be used at the background and the jobs involving unexpected volumes might affect other applications.

<https://towardsdatascience.com/basics-of-apache-spark-configuration-settings-ca4faff40d45>

# Q.if we have 50gb memory and 100gb data how spark will process it

Simple Method to choose Number of Partitions in Spark

At the end of this article, you will able to analyze your Spark Job and identify whether you have the right configurations settings for your spark environment and whether you utilize all your resources.

Whenever you work on a spark job, you should consider 2 things.

* Avoid Spill
* Maximize Parallelism by utilizing all the cores.

Both of these go hand in hand and you should be able to use them to their fullest.

**Spills:**Spill happens whenever there is Shuffle and the data has to be moved around and the executor is not able to hold the data in its memory. So it has to use the storage to save the data in memory for a certain time.   
When we don’t right size partitions, we get spills. Always avoid Spills. For reading 50 GB, the spill may go as high as 500 GB.

**Partitions:**

Let's start with some basic default and desired spark configuration parameters.

* Default Spark Shuffle Partitions — 200
* Desired Partition Size (Target Size)= 100 or 200 MB
* No Of Partitions = Input Stage Data Size / Target Size

Below are examples of how to choose the partition count.

**Case 1** :

* Input Stage Data 100GB
* Target Size = 100MB
* Cores = 1000
* Optimal Count of Partitions = 100,000 MB / 100 = 1000 partitions
* Spark.conf.set(“spark.sql.shuffle.partitions”,1000)
* **Partitions should not be less than number of cores**

**Case 2**:

* Input Size Data — 100GB
* Target Size = 100MB
* Cores = 96
* Optimal Count of Parititons = 100,000 MB / 100 MB = 1000 partitions
* Spark.conf.set(“spark.sql.shuffle.partitions”,960)
* When partition count is greater than Core Count, partitions should be a factor of the core count. Else we would be not utilizing the cores in the last run.

**Input:**

* Read the input data with the number of partitions, that matches your core count
* Spark.conf.set(“spark.sql.files.maxPartitionBytes”, 1024 \* 1024 \* 128) — setting partition size as 128 MB
* Apply this configuration and then read the source file. It will partition the file into multiples of 128MB
* To verify df.rdd.partitions.size
* By doing this, we will be able to read the file pretty fast

**Output:**

* When saving down the data, try to utilize all the cores.
* If the number of partitions matches the core count or is a factor of core count, we will achieve parallelism which in turn will reduce the time.

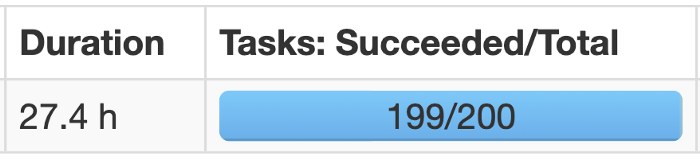
**Takeaways:**

* Too few partitions will lead to less concurrency.
* Too many partitions will lead to a lot of shuffles.
* Partition count in common lies between 100 and 10,000.
* Lower Bound: At least ~2x number of cores in the cluster.
* Upper Bound: Ensure tasks take at least 100ms.

# Q.Art of joining in Spark

I’ve met Apache Spark a few months ago and it has been love at first sight. My first thought was: “it’s incredible how something this powerful can be so easy to use, I just need to write a bunch of SQL queries!”. Indeed starting with Spark is very simple: it has very nice APIs in multiple languages (e.g. Scala, Python, Java), it’s virtually possible to just use SQL to unleash all of its power and it has a widespread community and tons of documentation. My starting point has been a book, [**Spark: The Definitive Guide**](https://amzn.to/39TIOYN) **(<- affiliate link** [**US**](https://amzn.to/39TIOYN)**,** [**UK**](https://amzn.to/2K82rBO)**)**, I believe it’s a good introduction to the tool: it is authored by [Bill Chambers](https://databricks.com/speaker/bill-chambers) (Product Manager at Databricks) and [Matei Zaharia](https://databricks.com/speaker/matei-zaharia) (Chief Technologist at Databricks and creator of Spark).

**Very soon I realized that things are not that easy as I used to believe**. And that same discovery is probably the reason why you landed on this article. For example, I would bet that you will find the following image quite familiar:



As you may already know the above is a typical manifestation of data skewness during a join operation: one task will take forever to complete just because all the effort of the join is put on a single executor process.

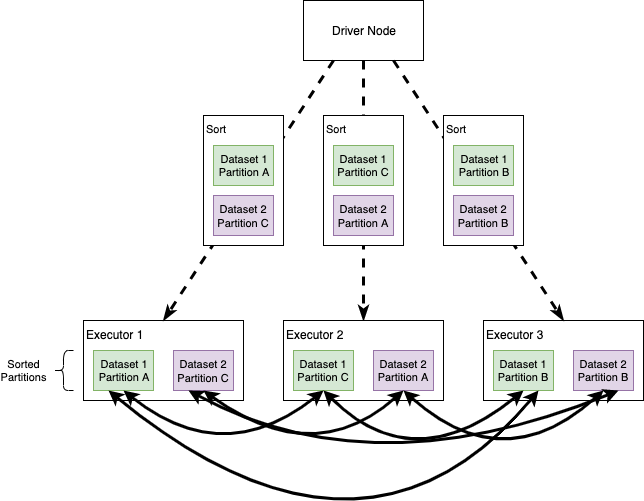
**Oversimplifying how Spark joins tables**

Looking at what tables we usually join with Spark, we can identify two situations: we may be joining **a big table with a small table** or, instead, **a big table with another big table**. Of course, during Spark development we face all the shades of grey that are between these two extremes!

Sticking to use cases mentioned above, Spark will perform (or be forced by us to perform) joins in two different ways: either using **Sort Merge Joins** if we are joining two big tables, or **Broadcast Joins** if at least one of the datasets involved is small enough to be stored in the memory of the single all executors. Note that there are other types of joins (e.g. Shuffle Hash Joins), but those mentioned earlier are the most common, in particular from Spark 2.3.

## ****Sort Merge Joins****

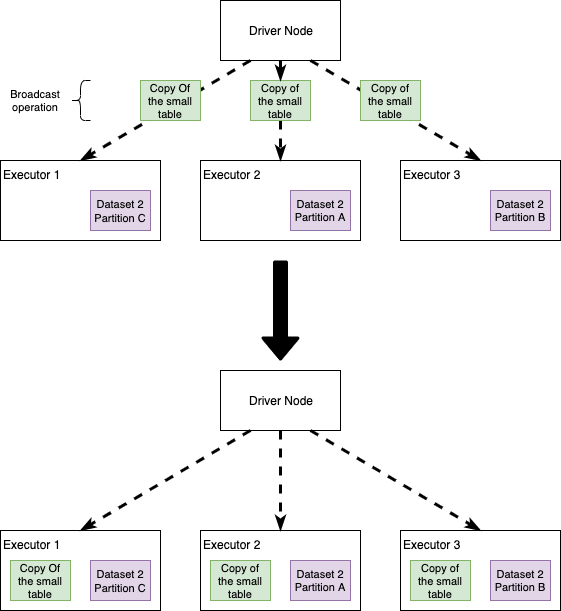
When Spark translates an operation in the execution plan as a Sort Merge Join it enables an **all-to-all communication strategy among the nodes**: the Driver Node will orchestrate the Executors, each of which will hold a particular set of joining keys. **Before running the actual operation, the partitions are first sorted** (this operation is obviously heavy itself). As you can imagine this kind of strategy can be expensive: nodes need to use the network to share data; **note that Sort Merge Joins tend to minimize data movements in the cluster, especially compared to Shuffle Hash Joins**.



In a Sort Merge Join partitions are sorted on the join key prior to the join operation.

## ****Broadcast Joins****

Broadcast joins happen when Spark decides to send **a copy of a table to all the executor nodes**. The intuition here is that, if we broadcast one of the datasets, Spark no longer needs an all-to-all communication strategy and **each Executor will be self-sufficient in joining the big dataset records in each node, with the small (broadcasted) table**. We’ll see that this simple idea improves performance… usually.



In a Broadcast Join a copy of the small table is sent to all the Executors. Each executor will then perform the join without the need of network communication

## ****Wrapping up our enemies****

Some of the biggest villains that we may face during join operations are:

* **Data Skewness**: the key on which we are performing the join is not evenly distributed across the cluster, causing one of the partitions to be very large and not allowing Spark to execute operations in parallel and/or congesting the network. Note that **Skewness is a problem that affects many parallel computation systems**: the keyword here is “parallel”, we can take advantage of these tools only if we are able to execute multiple operations at the same time, so any data processing system that finds itself in some kind of skewed operation will suffer from similar problems (e.g. it also happens running Branch-and-Bound algorithms for Mixed-Integer Linear Programming Optimization).
* **All-to-all communication strategy**
* **Limited executors memory**

**An important note before delving into some ideas to optimize joins**: sometimes I will use the execution times to compare different join strategies. Actually, a lower absolute execution time does not imply that one method is absolutely better than the other. Performance also depends on the Spark session configuration, the load on the cluster and **the synergies among configuration and actual code**. So, read what follows with the intent of gathering some ideas that you’ll probably need to tailor on your specific case!

## Broadcasting or not broadcasting

First of all, let’s see what happens if we decide to broadcast a table during a join. Note that the Spark execution plan could be automatically translated into a broadcast (without us forcing it), although this can vary depending on the Spark version and on how it is configured.

We will be joining two tables: fact\_table and dimension\_table. First of all, let’s see how big they are:

fact\_table.count // #rows 3,301,889,672  
dimension\_table.count // #rows 3,922,556

In this case, the data are not skewed and the partitioning is all right — you’ll have to trust my word. Note that the dimension\_table is not exactly “small” (although size is not information that we can infer by only observing the number of rows, we’d rather prefer to look at the file size on HDFS).

By the way, let’s try to join the tables without broadcasting to see how long it takes:

Output: Elapsed time: 215.115751969s

Now, what happens if we broadcast the dimension table? By a simple addition to the join operation, i.e. replace the variable dimension\_table with broadcast(dimension\_table), **we can force Spark to handle our tables using a broadcast**:

Output: Elapsed time: 61.135962017s

The broadcast made the code run **71% faster**! Again, read this outcome having in mind what I wrote earlier about absolute execution time.

Is broadcasting always good for performance? **Not at all!** If you try to execute the snippets above giving more resources to the cluster (in particular more executors), the non-broadcast version will run faster than the broadcast one! **One reason why this happens is because the broadcasting operation is itself quite expensive** (it means that all the nodes need to receive a copy of the table), so it’s not surprising that **if we increase the amount of executors that need to receive the table, we increase the broadcasting cost**, which suddenly may become higher than the join cost itself.

It’s important to remember that when we broadcast, we are hitting on the memory available on each Executor node (here’s a [brief article about Spark memory](https://0x0fff.com/spark-memory-management/)). **This can easily lead to Out Of Memory exceptions** or make your code unstable: imagine to broadcast a medium-sized table. You run the code, everything is fine and super fast. A couple of months later you suddenly find out that your code breaks, OOM. After some hours of debugging, you may discover that the medium-sized table you broadcast to make your code fast is not that “medium” anymore. Takeaway, **if you broadcast a medium-sized table, you need to be sure it will remain medium-sized in the future!**

**Skew it! This is taking forever!**

**Skewness is a common issue when you want to join two tables**. We say a join is skewed when the join key is not uniformly distributed in the dataset. During a skewed join, Spark cannot perform operations in parallel, **since the join’s load will be distributed unevenly across the Executors**.

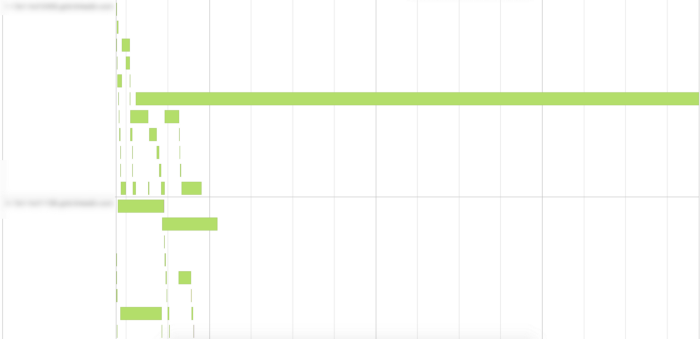
Let’s take our old fact\_table and a new dimension:

fact\_table.count // #rows 3,301,889,672  
dimension\_table2.count // #rows 52

Great our dimension\_table2 is very small and we can decide to broadcast it straightforward! Let’s join and see what happens:

Output: Elapsed time: 329.991336182s

Now, observe on the SparkUI what happened to the tasks during the execution:



As you can see in the image above, **one of the tasks took much more time to complete compared to the others**. This is clearly an indication of **skewness** in the data — and this conjecture would be easily verifiable by looking at the distribution of the join key in the fact\_table.

To make things work, we need to find a way to redistribute the workload to improve our join’s performance. I want to propose two ideas:

* **Option 1**: we can try to **repartition our fact table**, in order to distribute the effort in the nodes
* **Option 2**: we can artificially create a repartitioning key (key salting)

## ****Option 1: Repartition the table****

We can select a column that is uniformly distributed and repartition our table accordingly; **if we combine this with broadcasting**, we should have achieved the goal of redistributing the workload:

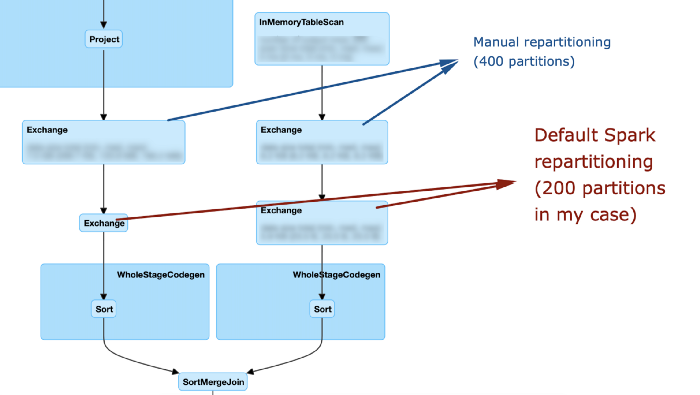
Output: Elapsed time: 106.708180448s

Note that we want to choose a column also looking at the cardinality (e.g. I wouldn’t choose a key with “too high” or “too low” cardinality, I let you quantify those terms).

Important note: if you cannot broadcast the dimension table and you still want to use this strategy, **the left side and the right side of the join need to be repartitioned using the same partitioner!** Let’s see what happens if we don’t.

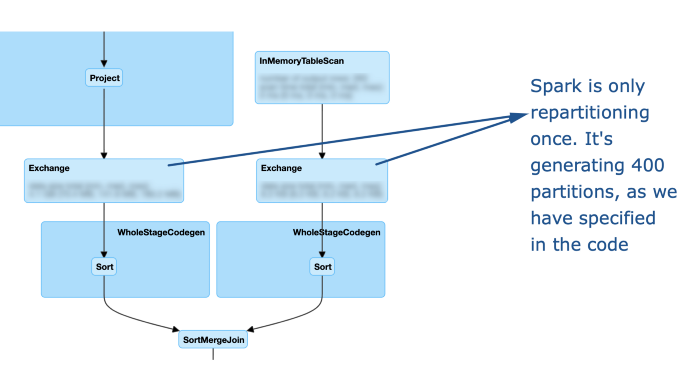
Consider the following snippet and let’s look at the DAG on the Spark UI

If we don’t specify a partitioner, Spark may decide to perform a default repartitioning before the join



As you can see, it this case my repartitioning is basically ignored: **after it is performed, spark still decides to re-exchange the data using the default configuration**. Let’s look at how the DAG changes if we use the same partitioner:

Using the same partitioner allows Spark to actually perform the join using our custom options

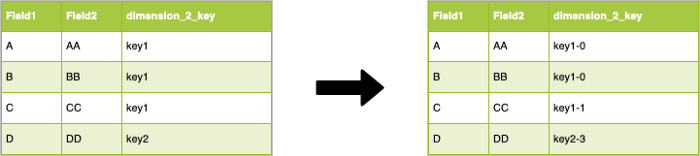


## ****Option 2: Key salting****

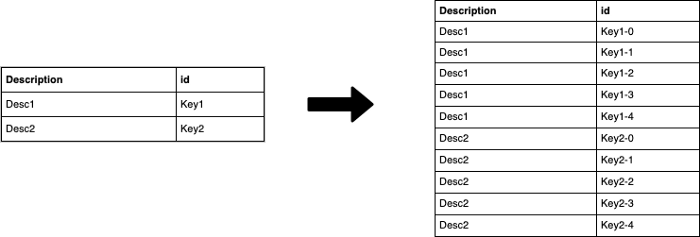
Another strategy is to **forge a new join key!**

We still want to force spark to do a uniform repartitioning of the big table; in this case, we can also combine Key salting with broadcasting, since the dimension table is very small.

The join key of the left table is stored into the field dimension\_2\_key, which is not evenly distributed. The first step is to make this field more “uniform”. An easy way to do that is to randomly append a number between 0 and N to the join key, e.g.:



As you can see we modified the dimension\_2\_key which is now “uniformly” distributed, **we are on the right path to a better workload on the cluster.** We have modified the join key, so **we need to do the same operation on the dimension table**. To do so, we create for each “new” key value in the fact table, a corresponding value in the dimension: for each value of the id in the dimension table we generate N values **in which we append to the old ids the numbers in the [0,N] interval.** Let’s make this clearer with the following image:



**At this point, we can join the two datasets using the “new” salted key.**

This simple trick will **improve the degree of parallelism of the DAG execution**. Of course, we have increased the number of rows of the dimension table (in the example N=4). A higher N (e.g. 100 or 1000) will result in a more uniform distribution of the key in the fact, **but in a higher number of rows for the dimension table!**

Let’s code this idea.

First, we need to append the salt to the keys in the fact table. This is a surprisingly challenging task, or, better, it’s a decision point:

* **We can use a UDF**: easy, but can be slow because Catalyst is not very happy with UDFs!
* **We can use the “rand” SQL operator**
* **We can use the monotonically\_increasing\_id function**

Just for fun, let’s go with this third option (it also appear to be a bit faster)

Now we need to “explode” the dimension table with the new key. The fastest way that I have found to do so is to create a dummy dataset containing the numbers between 0 and N (in the example between 0 and 1000) and cross-join the dimension table with this “dummy” dataset:

Finally, we can join the tables using the salted key and see what happens!

Output: Elapsed time: 182.160146932s

Again, execution time is not really a good indicator to understand our improvement, so let’s look at the event timeline:



As you can see we greatly increased the parallelism.

In this case, a simple repartitioning plus broadcast, worked better than crafting a new key. Note that **this difference is not due to the join, but to the random number generation during the fact table lift.**

## Takeaways

* Joins can be difficult to tune since performance are bound to both the code and the Spark configuration (number of executors, memory, etc.)
* Some of the most common issues with joins are all-to-all communication between the nodes and data skewness
* We can avoid all-to-all communication using broadcasting of small tables or of medium-sized tables if we have enough memory in the cluster
* Broadcasting is not always beneficial to performance: we need to have an eye for the Spark config
* Broadcasting can make the code unstable if broadcast tables grow through time
* Skewness leads to an uneven workload on the cluster, resulting in a very small subset of tasks to take much longer than the average
* There are multiple ways to fight skewness, one is repartitioning.
* We can create our own repartitioning key, e.g. using the key salting technique

# Q.How is fault tolerance achieved in Apache Spark?

The basic semantics of fault tolerance in[**Apache Spark**](http://data-flair.training/blogs/apache-spark-introduction-spark-comprehensive-tutorial/) is, all the [**Spark RDDs**](http://data-flair.training/blogs/rdd-in-apache-spark/) are immutable. It remembers the dependencies between every RDD involved in the operations, through the lineage graph created in the [**DAG**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/), and in the event of any failure, Spark refers to the lineage graph to apply the same operations to perform the tasks.

There are two types of failures – Worker or driver failure. In case if the worker fails, the executors in that worker node will be killed, along with the data in their memory. Using the lineage graph, those tasks will be accomplished in any other worker nodes. The data is also replicated to other worker nodes to achieve fault tolerance. There are two cases:

1.**Data received and replicated** – Data is received from the source, and replicated across worker nodes. In the case of any failure, the data replication will help achieve fault tolerance.

2.**Data received but not yet replicated**– Data is received from the source but buffered for replication. In the case of any failure, the data needs to be retrieved from the source.

For stream inputs based on receivers, the fault tolerance is based on the type of receiver:

<li style=”list-style-type: none”>

* **Reliable receiver** – Once the data is received and replicated, an acknowledgment is sent to the source. In case if the receiver fails, the source will not receive acknowledgment for the received data. When the receiver is restarted, the source will resend the data to achieve fault tolerance.
* **Unreliable receiver** – The received data will not be acknowledged to the source. In this case of any failure, the source will not know if the data has been received or not, and it will nor resend the data, so there is data loss.

To overcome this data loss scenario, Write Ahead Logging (WAL) has been introduced in Apache Spark 1.2. With WAL enabled, the intention of the operation is first noted down in a log file, such that if the driver fails and is restarted, the noted operations in that log file can be applied to the data. For sources that read streaming data, like Kafka or Flume, receivers will be receiving the data, and those will be stored in the executor’s memory. With WAL enabled, these received data will also be stored in the log files.

WAL can be enabled by performing the below:

* Setting the checkpoint directory, by using streamingContext.checkpoint(path)
* Enabling the WAL logging, by setting spark.stream.receiver.WriteAheadLog.enable to True.

**to know more about fault tolerance refer:**[**Fault Tolerance in Spark**](http://data-flair.training/blogs/apache-spark-streaming-fault-tolerance/)

# USEFUL LINKS

* <https://towardsdatascience.com/the-art-of-joining-in-spark-dcbd33d693c>
* <https://medium.com/@brajendragouda/5-key-factors-to-keep-in-mind-while-optimising-apache-spark-in-aws-part-2-c0197276623c>
* <https://medium.com/teads-engineering/spark-performance-tuning-from-the-trenches-7cbde521cf60>
* <https://medium.com/tblx-insider/how-we-reduced-our-apache-spark-cluster-cost-using-best-practices-ac1f176379ac>
* <https://medium.com/expedia-group-tech/part-3-efficient-executor-configuration-for-apache-spark-b4602929262>
* <https://towardsdatascience.com/about-joins-in-spark-3-0-1e0ea083ea86>
* <https://towardsdatascience.com/be-in-charge-of-query-execution-in-spark-sql-c83d1e16b9b8>
* <https://ch-nabarun.medium.com/apache-spark-optimization-techniques-54864d4fdc0c>
* <https://changhsinlee.com/pyspark-dataframe-basics/>
* <https://robertovitillo.com/spark-best-practices/>
* <https://luminousmen.com/post/spark-tips-partition-tuning>