Exercise Session 8 : YARN & Spark

Agenda :

① YARN
- What it is
- Architecture and components
- Selected exercises

② Spark
- Organization
- RDD
  - Design
  - Caching
  - Partitioning
- Querying options

MapReduce : Design

Session 7

1. Data replication
2. Task scheduling

1

Bottleneck!

(Daemon)

NameNode

/dir/file

D3

D4

1

2

JobTracker

T3 T5 ⇐ T6 ⇐ T4

T1

TaskTracker

2

Colocation

DataNode

T2

T3

task

T4 (busy)

replica1

T5

rare

(only 1 replica shown per block)

T6 (busy)

Network traffic

task replica 2

Split

Split

Split

Split

Split

Split

Split

Split

D1

D2

D3

D4

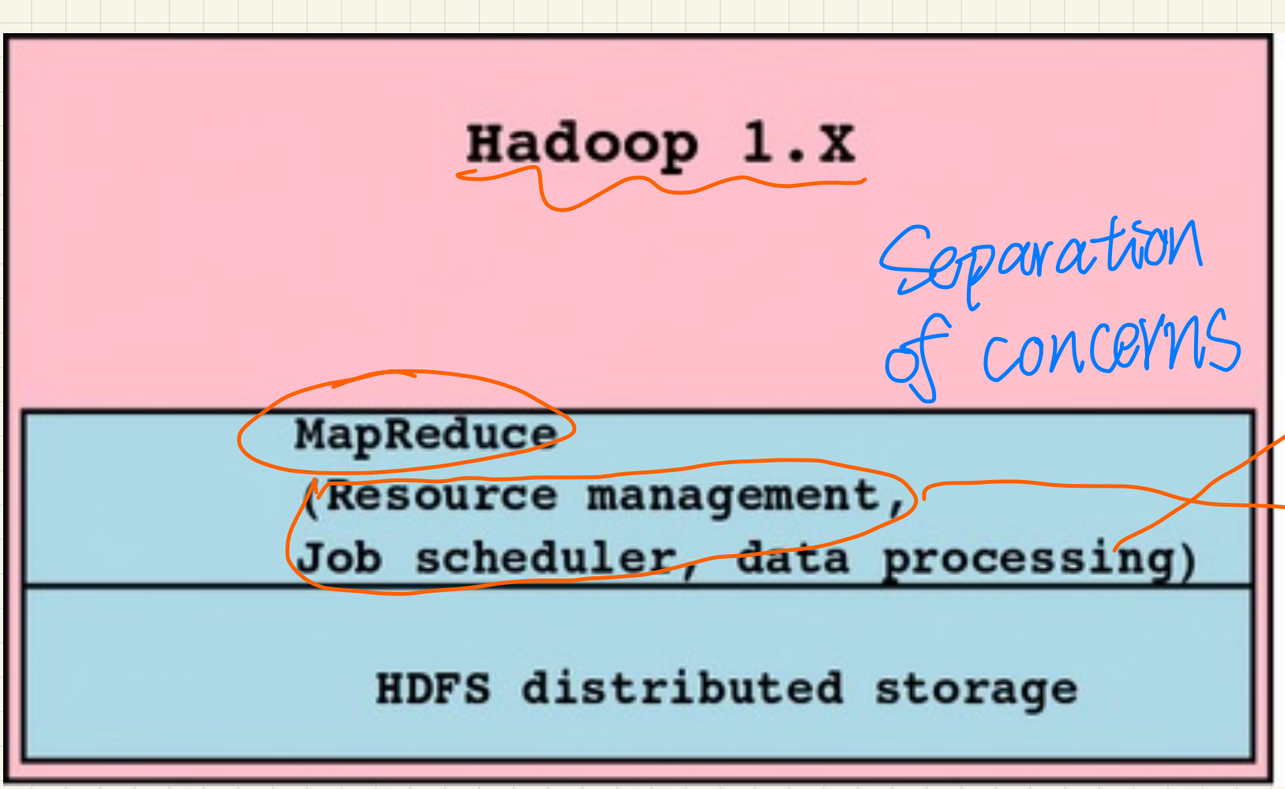
D5

D6

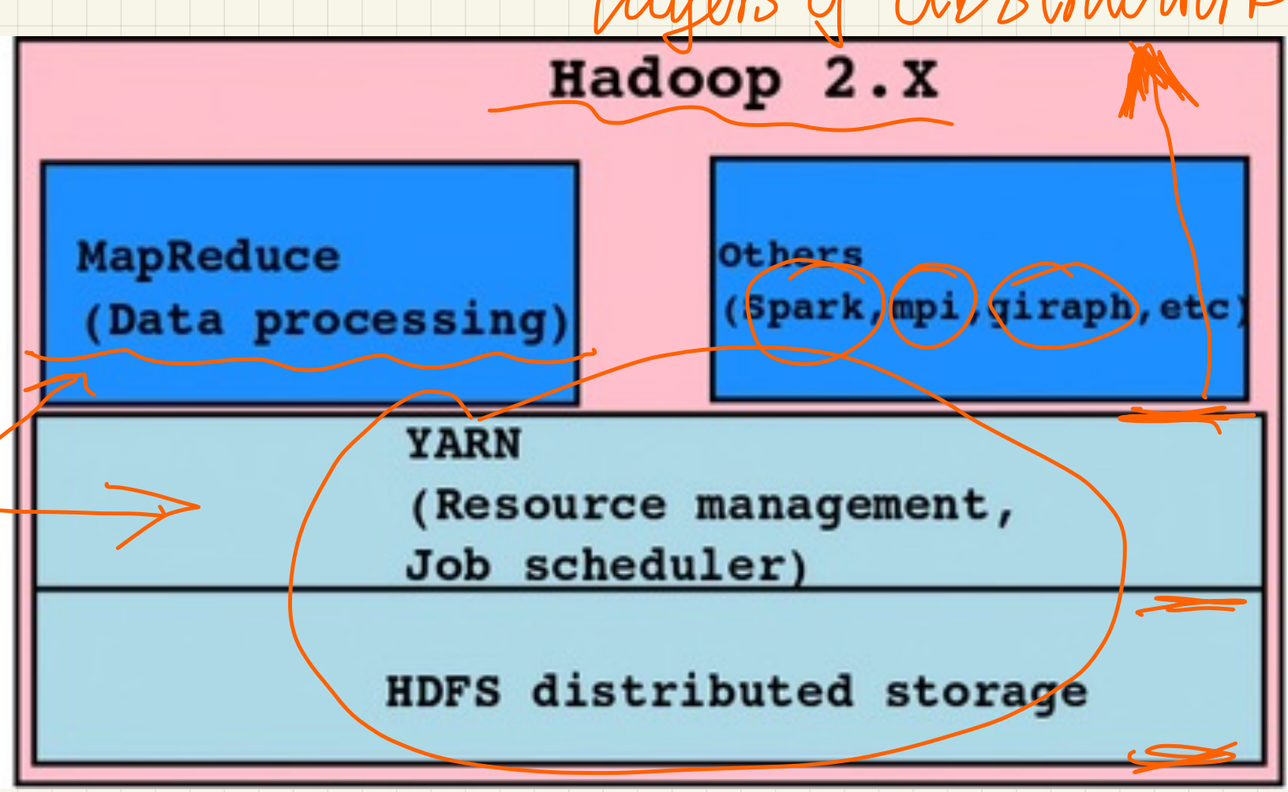
YARN

Separating resource management from data processing

V1: ~5K nodes, 40K concurrent tasks ⇒ 15 K nodes
(Yahoo!)
thousands of tasks...
layers of abstractions



Separation of concerns



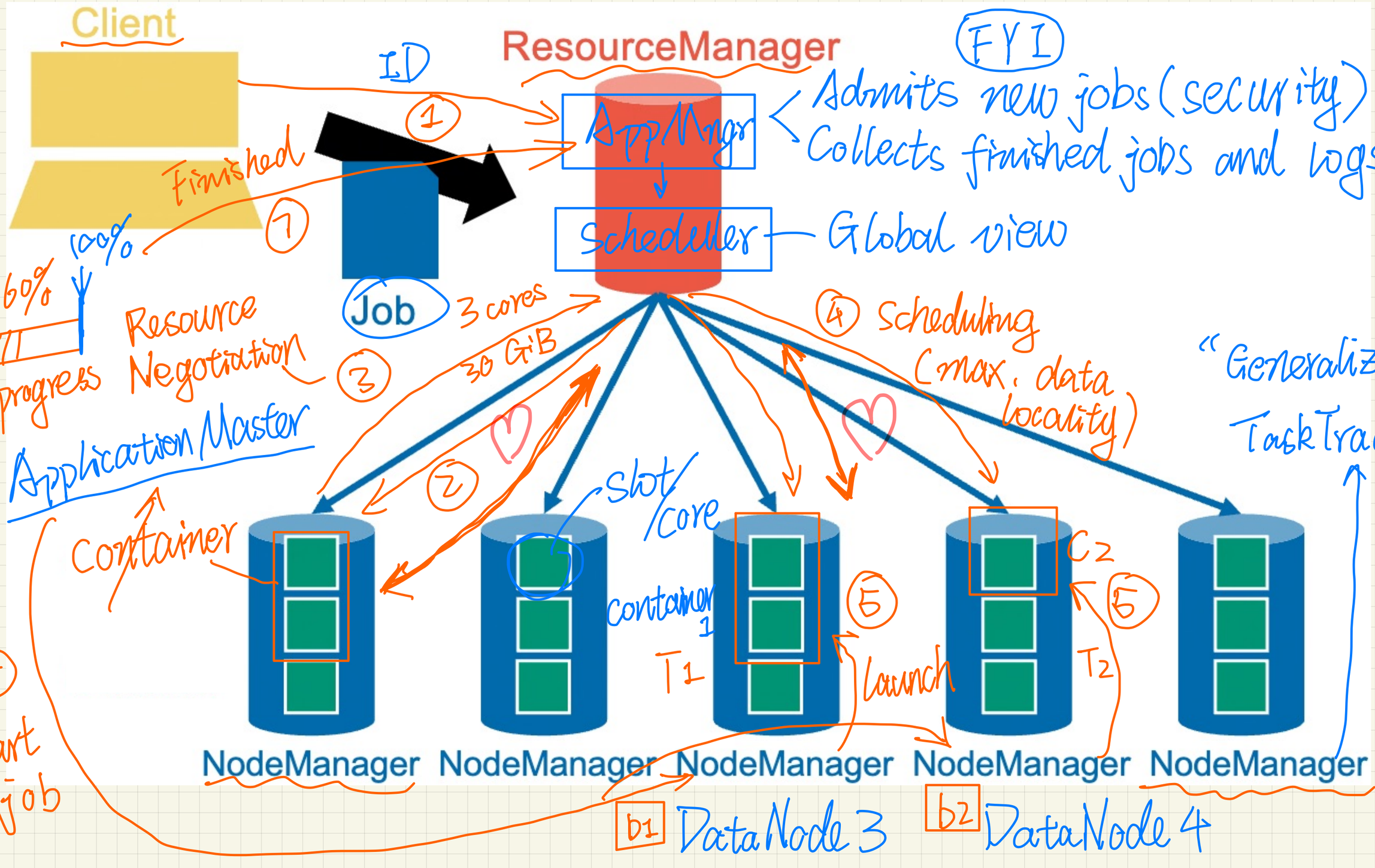
Added more details to 1.1

1.1 – List at least 3 main shortcomings of MapReduce v1, which are addressed by YARN design.

Answer

1. **Scalability Limitations in MapReduce v1**:
   - *Bottleneck*: MapReduce v1 suffered from a bottleneck with the JobTracker, responsible for resource scheduling and monitoring. This limited scalability, allowing for only around 5K nodes and up to 40K concurrent tasks (per Yahoo!).
   - *YARN Solution*: YARN's design decentralizes resource management, eliminating the JobTracker bottleneck. It allows for much greater scalability by enabling efficient resource allocation and management across the cluster, accommodating larger node counts (~10K) and higher concurrent task loads.
2. **Rigidity in Job Types**:
   - *MapReduce Specificity*: MapReduce v1 was confined to supporting only MapReduce-specific jobs. This limitation hindered the ability to schedule and execute diverse workloads such as MPI, graph processing, or user-specific code.
   - *YARN's Adaptability*: YARN's architecture is designed to accommodate various workloads beyond MapReduce, enabling the sharing of the cluster resources among different types of applications and frameworks. It allows for scheduling and executing non-MapReduce workloads like MPI, graph processing, and custom user code, enhancing the cluster's versatility.
3. **Resource Utilization Efficiency**:
   - *Idle Resources*: MapReduce v1 exhibited inefficiencies in resource utilization where reducers often waited for mappers to finish (and vice-versa), resulting in idle resources during these wait times.
   - *YARN's Dynamic Resource Allocation*: YARN optimizes resource utilization by allowing independent scheduling of resources for different tasks. It enables simultaneous execution of various tasks, ensuring that all available resources are utilized efficiently at any given time, reducing idle periods and maximizing overall cluster performance.
4. **Flexibility and Dynamic Configuration**:
   - *Static Mapper and Reducer Roles*: MapReduce v1 constrained the roles of mappers and reducers to configurations set at job initiation, lacking the flexibility to adapt roles during runtime.
   - *YARN's Dynamic Configuration*: YARN provides greater flexibility by allowing dynamic allocation and reallocation of resources during runtime. Applications can adapt their resource requirements and configurations dynamically, allowing for more efficient resource utilization and adaptive task handling based on changing workloads or conditions.

YARN Architecture



More Details on the Components:

Components of YARN

1. **ResourceManager (RM):**

   - **Global Resource Manager**: Controls and manages the cluster's resources.
   - **Resource Allocation**: Allocates resources to various applications.
   - **Scheduling**: Schedules resources among competing applications based on policies and priorities.
   - **Manages NodeManagers**: Monitors NodeManagers and their resource availability.

2. **NodeManager (NM):**

   - **Node-level Resource Manager**: Manages resources on individual cluster nodes.
   - **Monitors Resource Usage**: Tracks resource utilization (CPU, memory, etc.) on the node.
   - **Execution and Monitoring**: Executes and monitors containers (fundamental units of execution) on the node.
   - **Reports to ResourceManager**: Regularly communicates with the ResourceManager, providing updates on available resources and container status.

3. **ApplicationMaster (AM):**

   - **Per-Application Coordinator**: Manages a specific application running on YARN.
   - **Resource Negotiation**: Negotiates resources from the ResourceManager for the application's tasks or containers.
   - **Application Lifecycle Management**: Coordinates the execution lifecycle, handling task execution, monitoring progress, and handling failures or retries.
   - **Optimizes Execution**: Optimizes task execution based on resource availability and application requirements.

4. **Containers:**

   - **Execution Units**: Containers encapsulate allocated resources (CPU, memory, etc.) for running specific tasks or components of an application.
   - **Isolated Execution Environment**: Provides a controlled environment for executing tasks without interference from other containers on the same node.
   - **Managed by NodeManager**: Managed and executed by the NodeManager on respective nodes.

5. **ResourceManager Web UI:**

   - **Visualization and Monitoring**: Provides a web-based interface for administrators and users to monitor resource utilization, application status, and cluster health.
   - **Insight into Resource Allocation**: Offers insights into resource allocation, application progress, and overall cluster performance.

More Details on Launching a Job:

Steps taken to launch a job in YARN

1. **Submitting the Job:**
   - A user submits their job/application to the YARN cluster. This submission might be through a command-line interface, an API call, or a higher-level framework built on top of YARN, such as Apache Hadoop MapReduce, Apache Spark, or others.
2. **ResourceManager Receives the Request:**
   - The ResourceManager (RM) receives the job/application submission request.
   - The RM is responsible for managing the cluster's resources and allocating resources to various applications.
3. **Resource Negotiation:**
   - The ResourceManager negotiates resources for the submitted job/application.
   - It considers the resource requirements specified by the user or the job/application (e.g., memory, CPU cores) and the current availability of resources in the cluster.
4. **Allocation of Containers:**
   - Upon resource allocation, the ResourceManager assigns containers to the ApplicationMaster (AM) for the submitted job/application.
   - Each container encapsulates the allocated resources (CPU, memory, etc.) needed to execute specific tasks.
5. **Launching the ApplicationMaster:**
   - The ResourceManager launches the ApplicationMaster on a chosen node in the cluster.
   - The ApplicationMaster is responsible for managing the execution of the user's job/application. It coordinates with the ResourceManager for resource needs and oversees task execution.

6. **Task Execution within Containers:**

   - The ApplicationMaster, once initialized, starts sending requests to the NodeManagers (NMs) to launch containers on specific nodes within the cluster.
   - NodeManagers launch containers and execute tasks associated with the job/application within these containers.

7. **Container Execution and Monitoring:**

   - NodeManagers monitor and manage the execution of containers.
   - They *report container status updates (start, finish, failure) to the ResourceManager, which in turn communicates this information to the ApplicationMaster.*

8. **Task Completion and Job Termination:**

   - As tasks within containers complete their execution, they report their status back to the ApplicationMaster.
   - The ApplicationMaster keeps track of task completion, aggregates results, and manages the overall progress of the job/application.

9. **Finalization and Resource Release:**

   - Once the job/application completes its tasks, the ApplicationMaster signals the ResourceManager about job completion.
   - The ResourceManager releases the allocated resources back to the cluster for other applications and awaits further job submissions.

Task 1.2

1.2 – State which of the following statements are true:

1. The ResourceManager has to provide fault tolerance for resources across the cluster
2. Container allocation/deallocation can take place in a dynamic fashion as the application progresses.
3. YARN plans to allow applications to only request resources in terms of memory usage and number of CPUs.
4. Communications between the ResourceManager and NodeManagers are heartbeat-based.
5. The ResourceManager does not have a global view of all usage of cluster resources. Therefore, it tries to make better scheduling decisions based on probabilistic prediction.
6. ResourceManager has the ability to request resources back from a running application.

AM + NM + HDFS

X GPUs, Disks, Custom resources

↳ "Resource Preemption" based on priorities, fairness, ...

Added more details to the answer for 1.2

1. The ResourceManager has to provide fault tolerance for resources across the cluster

   The statement is **false** because, in the context of YARN (Yet Another Resource Negotiator), the responsibility for fault tolerance of resources across the cluster does not lie primarily with the ResourceManager. Fault tolerance in YARN is primarily handled by other components:

   A. **NodeManagers**: These are responsible for managing resources on individual nodes in the cluster. They monitor resource usage, report to the ResourceManager, and manage container execution. NodeManagers are crucial in handling failures at the node level and ensuring that tasks or containers are rerun on other healthy nodes in case of node failures.

   B. **ApplicationMaster**: Each application running on YARN has its own ApplicationMaster. The ApplicationMaster is responsible for negotiating resources with the ResourceManager and managing the execution of the application's tasks or containers. It monitors the progress of the application and handles failures or retries at the application level.

   C. **HDFS**: For fault tolerance of data, YARN often relies on the underlying HDFS. HDFS replicates data across multiple nodes in the cluster to ensure that data remains available even if some nodes fail.

1. Container allocation/deallocation can take place in a dynamic fashion as the application progresses.

   True

2. YARN plans to allow applications to only request resources in terms of memory usage and number of CPUs.

   False, because because YARN, as a resource management framework, offers flexibility in the type of resources that applications can request beyond just memory and CPU.

   YARN is designed to accommodate diverse workloads and varied resource requirements beyond memory and CPU. It supports the notion of "resource types," which enables applications to request and utilize different kinds of resources based on their specific needs. These resource types can include but are not limited to:

   A. **Memory:** Applications can request a certain amount of memory for their tasks or containers.
   B. **CPU:** They can also specify the number of CPU cores required for their execution.
   C. **GPUs:** For applications that benefit from GPU acceleration, YARN allows for GPU resource requests.
   D. **Disk Space:** Applications might need specific disk space for temporary storage or data processing.
   E. **Custom Resources:** YARN allows for defining custom resource types to cater to specific application requirements that go beyond standard memory and CPU, such as specialized hardware or software licenses.

   By enabling applications to request various types of resources, YARN supports a wide range of workloads, including machine learning, data analytics, scientific computing, and more, each with distinct resource needs beyond memory and CPU. Therefore, the statement is false because YARN's design is not limited solely to memory usage and the number of CPUs when it comes to resource requests.

3. Communications between the ResourceManager and NodeManagers are heartbeat-based.

   True: NodeManagers periodically send heartbeats (every a few seconds) to the ResourceManager to provide status updates and information about the resources available on their respective nodes. These heartbeats serve as a way for NodeManagers to communicate their health, report resource utilization, and update the ResourceManager about the status of containers they are managing.

   The ResourceManager relies on these heartbeat messages from NodeManagers to keep track of node health, available resources, and to detect any issues or failures at the node level. By maintaining a regular heartbeat exchange, the ResourceManager can dynamically manage resources, allocate or reallocate resources as needed, and make decisions based on the most up-to-date information about the cluster's state. This heartbeat-based communication ensures that the ResourceManager stays informed about the health and status of the nodes in the cluster, allowing for effective resource allocation, fault tolerance, and overall cluster management within the YARN framework.

4. The ResourceManager does not have a global view of all usage of cluster resources. Therefore, it tries to make better scheduling decisions based on probabilistic prediction.

   False (last session)

5. ResourceManager has the ability to request resources back from a running application.

   True: In the YARN (Yet Another Resource Negotiator) framework, the ResourceManager (RM) has the capability to reclaim or request resources back from a running application under certain circumstances.

   This feature is known as "resource preemption." Preemption allows the ResourceManager to reclaim resources from applications that are utilizing more resources than their fair share or if there's a need to allocate resources to higher-priority applications.

   A. **Queue-based Preemption:** YARN supports resource allocation based on queues with varying priorities. If a higher-priority queue or application needs resources that are currently being used by a lower-priority application, the ResourceManager can reclaim resources from the lower-priority application to satisfy the higher-priority request.

   B. **Over-allocation or Fairness:** If an application is using resources significantly beyond its allocated share or if it's causing other applications to be starved of resources, the ResourceManager can step in and reclaim resources to maintain fairness and prevent resource monopolization.

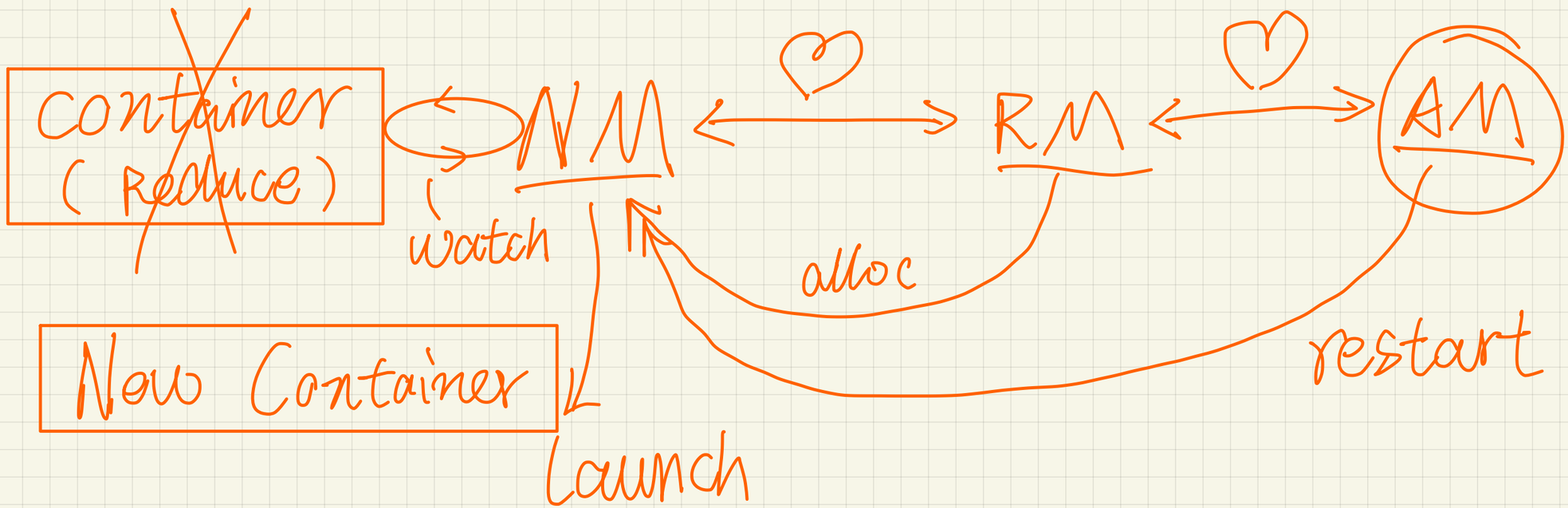
   C. **Dynamic Resource Demands:** In environments where resource demands fluctuate dynamically, the ResourceManager can adjust resource allocations across applications to ensure optimal resource utilization and meet changing demands efficiently.

   Resource preemption ensures that resources are distributed fairly and efficiently among competing applications in the cluster. However, preemption is typically used judiciously to avoid disrupting long-running jobs or causing unnecessary interference with the application's execution, aiming to balance fairness and cluster efficiency.

Task 1.3

1.3 – Whose responsibility is it? Say which component of YARN is resposible for each of the following tasks.

1. Fault Tolerance of running applications *[ResourceManager | ApplicationMaster | NodeManager]* → Node-level monitoring
2. Asking for resources needed for an application *[ResourceManager | ApplicationMaster | NodeManager]* Resource Negotiation
3. Providing leases to use containers *[ResourceManager | ApplicationMaster | NodeManager]*
4. Tracking status and progress of running applications *[ResourceManager | ApplicationMaster | NodeManager]*
5. (Bonus) Handling container failures. *[ResourceManager | ApplicationMaster | NodeManager]*



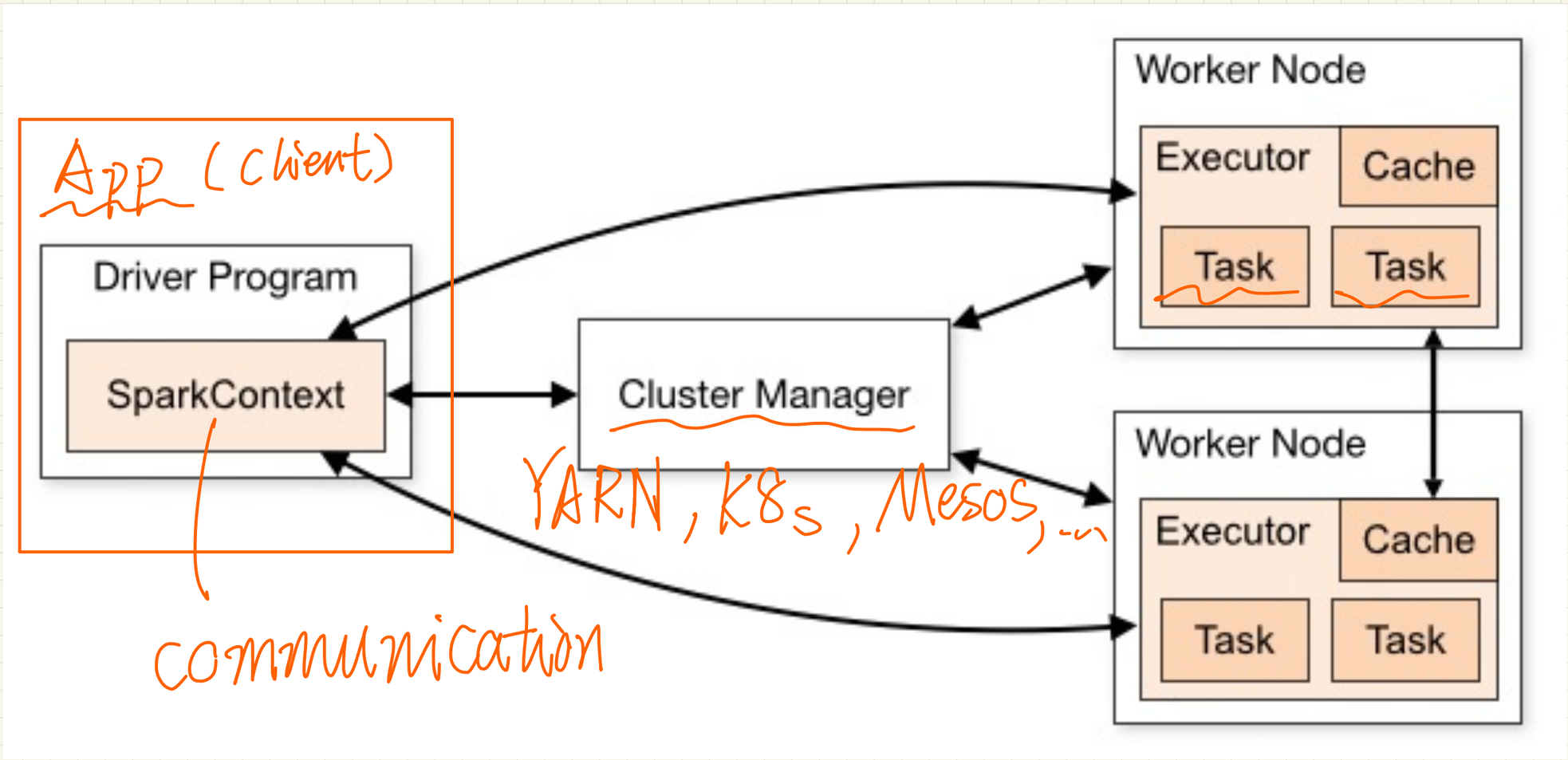
More details on 1.3 Bonus

RM + AM + NM

1. **Container Failure Detection:** When a container fails, the NodeManager detects the failure and reports it to the ResourceManager.
2. **ResourceManager's Role:** The ResourceManager receives information about container failures and manages the cluster's resources. It reallocates resources and attempts to recover from the failure by potentially rerunning the failed container on a different node if necessary. *The ResourceManager ensures that the resources requested by the ApplicationMaster are made available for the failed task to be retried.*
3. **ApplicationMaster's Response:** The ApplicationMaster, which is responsible for managing the specific application's execution, receives updates from the ResourceManager about container status changes. Upon detecting a container failure, the ApplicationMaster can take action, such as:
    - **Retrying Failed Tasks:** The ApplicationMaster might initiate a retry of the failed task by requesting the ResourceManager to allocate resources for a new container to replace the failed one. This retry can occur on the same or a different node.
    - **Handling Failures:** The ApplicationMaster might implement error handling strategies, such as logging the failure, updating the job status, resuming from a checkpoint, or taking other appropriate actions based on the application's logic and requirements.

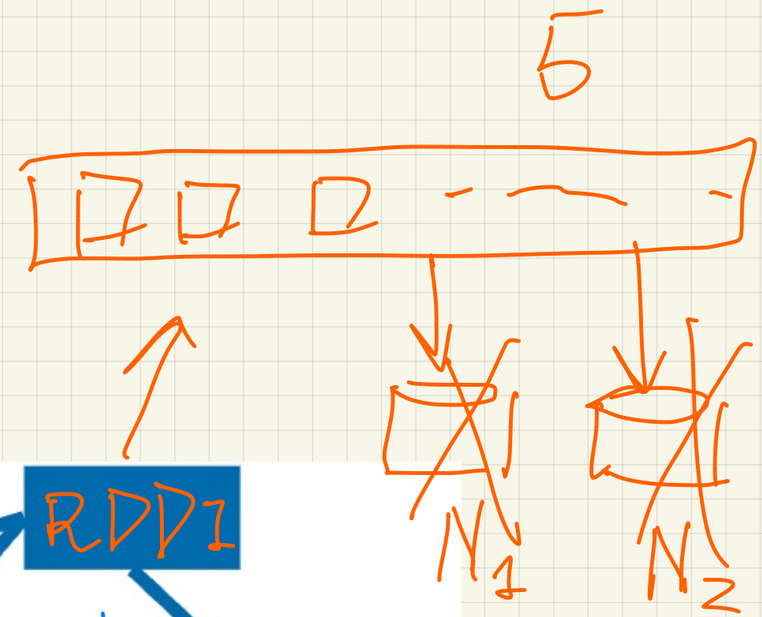
YARN's design allows for targeted recovery by rerunning specific failed tasks or containers, minimizing the impact of failures on the overall job execution. The ApplicationMaster is responsible for managing these recovery actions based on the status updates received from the ResourceManager.

Spark !

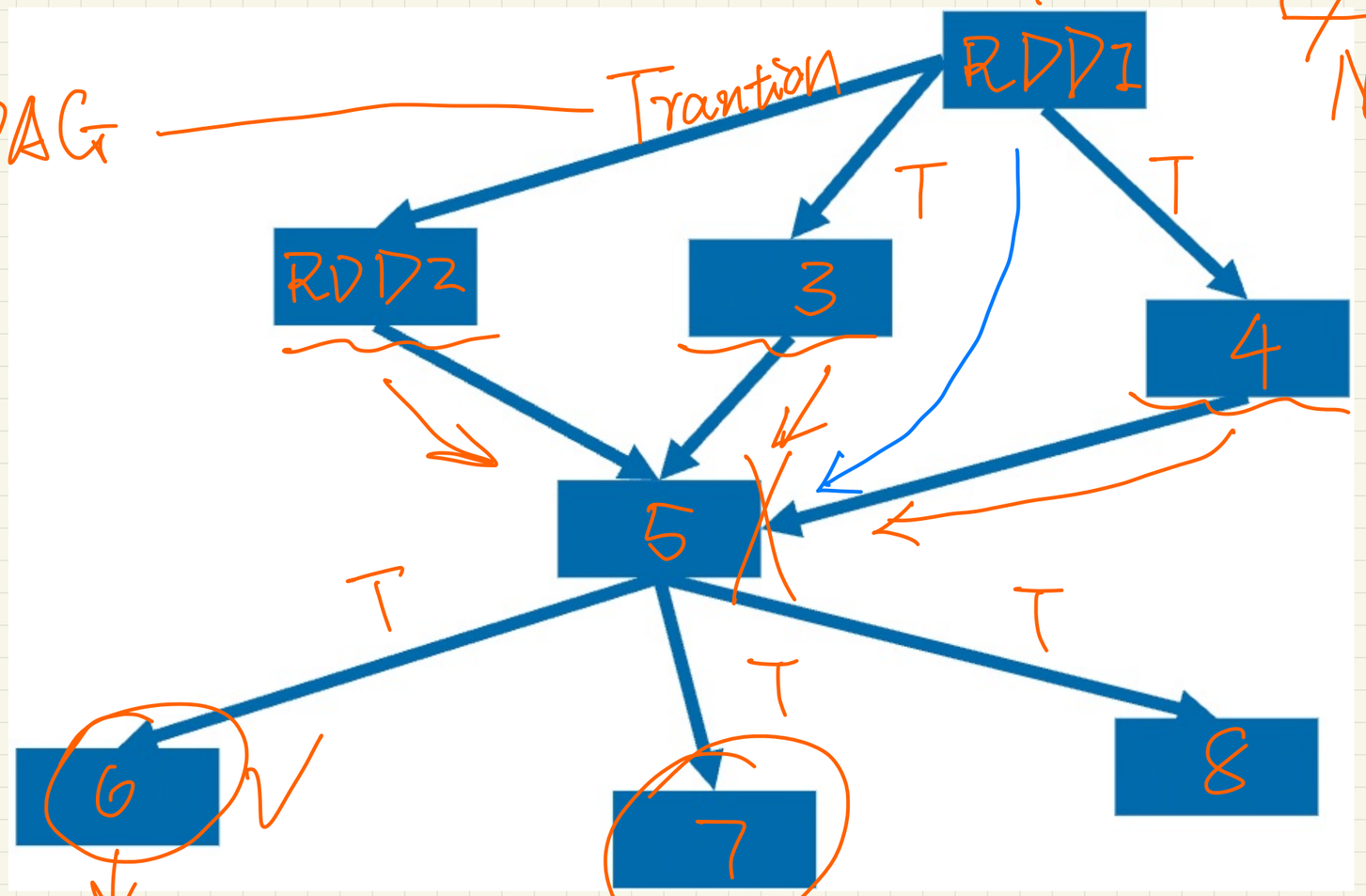

App (client)
Driver Program
SparkContext
Cluster Manager
YARN, K8s, Mesos, ...
communication
Worker Node
Executor
Cache
Task
Task
Worker Node
Executor
Cache
Task
Task

Design of Resilient Distributed Datasets (RDD)

- Immutable : Lineage ⇒ Resilience
- "Lazy" : Optizing execution



Drawing the DAG



Action

Why is Spark faster than MapReduce?

1. **In-Memory Processing:** Spark performs computations in memory, reducing the need to read and write to disk after each step, unlike Hadoop MapReduce, which writes to disk between map and reduce phases. This in-memory processing significantly reduces I/O overhead and speeds up data processing.

2. **DAG Execution Model:** Spark uses a Directed Acyclic Graph (DAG) execution model, allowing it to optimize and chain together multiple operations or transformations into a single job. This eliminates the overhead of writing intermediate results to disk after each stage, improving efficiency.

3. **Resilient Distributed Datasets (RDDs):** Spark's RDDs are a fundamental abstraction that allows for fault tolerance and distributed data processing. RDDs can be cached in memory across multiple operations, reducing data replication and improving the speed of subsequent computations.

4. **Lazy Evaluation:** Spark uses lazy evaluation, deferring the actual execution of operations until an action is triggered. This optimization allows it to optimize the execution plan and minimize unnecessary computations, resulting in faster processing.

5. **Rich APIs and Libraries:** Spark offers a wide range of high-level APIs in Python, Scala, Java, and R, along with various libraries (e.g., Spark SQL, MLlib, GraphX), enabling diverse and complex data processing tasks to be performed more efficiently compared to the limited capabilities of MapReduce.

6. **Interactive and Iterative Processing:** Spark is well-suited for interactive data analysis and iterative machine learning algorithms due to its in-memory computations and iterative processing capabilities, resulting in faster response times for these workloads compared to MapReduce.

7. **Dynamic Partitioning and Pipelining:** Spark provides dynamic partitioning, enabling users to control how data is distributed across nodes, and pipelining optimizations that minimize shuffling and data movement, leading to improved performance.

While Spark demonstrates faster performance compared to Hadoop MapReduce in many scenarios, the choice between the two depends on specific use cases, data sizes, resource availability, and processing requirements. In certain situations, such as handling large-scale batch processing or when fault tolerance is critical, MapReduce might still be a suitable choice despite Spark's performance advantages.

```python
from pyspark.sql import SparkSession
from pyspark import SparkConf
#* use spark session
spark = SparkSession.builder.master('local').getOrCreate()
sc = spark.sparkContext
```

```python
# Create an RDD
fruits = sc.textFile('fruits.txt')
yellowThings = sc.textFile('yellowthings.txt')
```

RDD transformations

```python
# map
fruitsReversed = fruits.map(lambda fruit: fruit[::-1])
fruitsReversed.collect()
```

```
['elppa',
 'ananab',
 'nolem yranac',
 'eparg',
 'nomel',
 'egnaro',
 'elppaenip',
 'yrrebwarts']
```

```python
# filter
shortFruits = fruits.filter(lambda fruit: len(fruit) <= 5)
shortFruits.collect()
```

```
['apple', 'grape', 'lemon']
```

```python
# flatMap
characters = (fruits
              .filter(lambda fruit: len(fruit) <= 5)
              .flatMap(lambda fruit: list(fruit)))
characters.collect()
```

```
['a', 'p', 'p', 'l', 'e', 'g', 'r', 'a', 'p', 'e', 'l', 'e', 'm', 'o', 'n']
```

```python
# union between fruits and yellowthings datasets
fruitsAndYellowThings = fruits.union(yellowThings)
fruitsAndYellowThings.collect()
```

```
{' ',
 'a',
 'b',
 'c',
 'e',
 'g',
 'i',
 'l',
 'm',
 'n',
 'o',
 'p',
 'r',
 's',
 't',
 'w',
 'y'}
```

RDD actions

```python
# collect
fruitsArray = fruits.collect()
yellowThingsArray = yellowThings.collect()
print(fruitsArray)
print(yellowThingsArray)
```

```
['apple', 'banana', 'canary melon', 'grape', 'lemon', 'orange', 'pineapple', 'strawberry']
['banana', 'bee', 'butter', 'canary melon', 'gold', 'lemon', 'pineapple', 'sunflower']
```

```python
# count - how many fruits are
fruits.count()
```

```
8
```

```python
# take - show the first three fruits
fruits.take(3)
```

```
['apple', 'banana', 'canary melon']
```

```python
# reduce - what letters are used
fruits.map(lambda fruit: set(fruit)).reduce(lambda x, y: x.union(y))
```

Transformations & Actions

Explicit Caching of (intermediate) RDDs

```
1 from pyspark.storagelevel import StorageLevel
```

```
1 #* Example 1
2 fruits.cache()
3 fruits.count(), fruits.take(3)
```

```
(8, ['apple', 'banana', 'canary melon'])
```

```
1 #* Memory-cached deserialized object + (Memory-cached) serialized Java object
2 fruits.is_cached, str(fruits.getStorageLevel())
```

```
(True, 'Memory Serialized 1x Replicated')
```

```
1 fruits.unpersist()
2 #* Memory-cached serialized Java object
3 fruits.is_cached, str(fruits.getStorageLevel())
```

```
(False, 'Serialized 1x Replicated')
```

```python
#* Example 2
words = fruits.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
result.persist(StorageLevel.MEMORY_AND_DISK_2)
result.collect()
```

```
[('apple', 1),
 ('banana', 1),
 ('canary', 1),
 ('melon', 1),
 ('grape', 1),
 ('lemon', 1),
 ('orange', 1),
 ('pineapple', 1),
 ('strawberry', 1)]
```

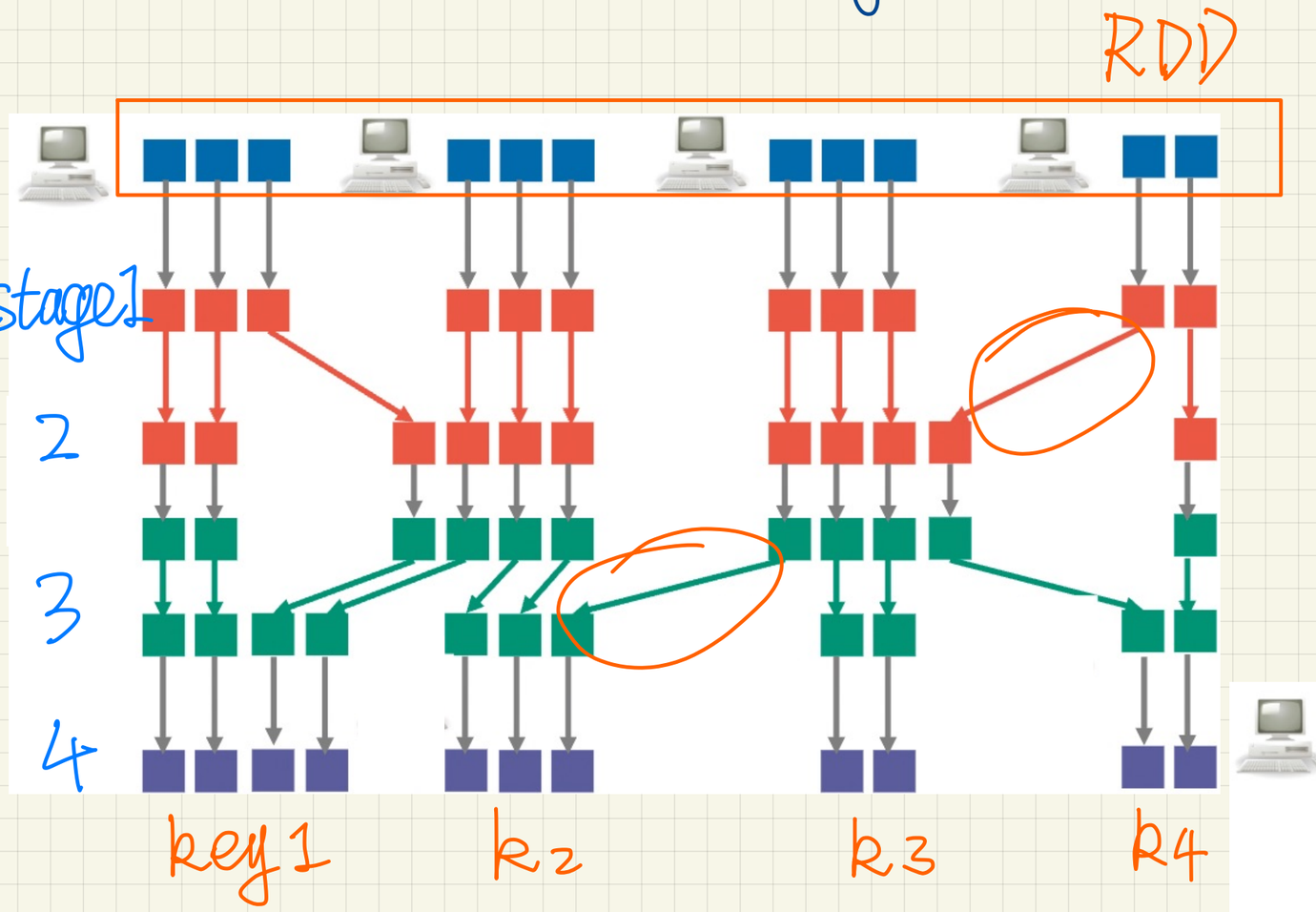
```python
#* Disk (if falls out of mem) +
#* Memory-cached deserialized object +
#* Memory-cached serialized Java object +
#* Replicated on 2 nodes
result.is_cached, str(result.getStorageLevel())
```

```
(True, 'Disk Memory Serialized 2x Replicated')
```

```python
result.unpersist()
result.is_cached, str(result.getStorageLevel())
```

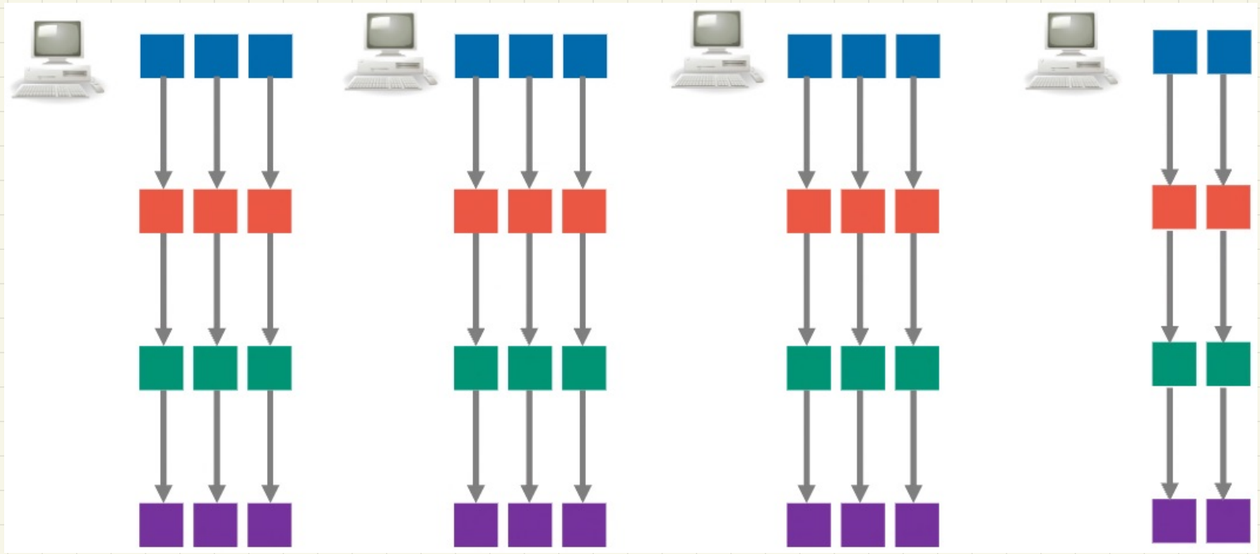
```
(False, 'Serialized 1x Replicated')
```

RDD Partitioning



Partitioning
⇓
Reduce shuffling
⇓
Network traffic ↓ perf ↑

k1 k2 k3 k4



Partitioning

RDD

```
1 nums = [(k, str(v)) for k in range(3) for v in range(2)]
2 pairs = sc.parallelize(nums)
```

RDD

```
1 print("Number of partitions: ", pairs.getNumPartitions())
2 pairs.glom().collect()
```

```
Number of partitions:  1
[[(0, '0'), (0, '1'), (1, '0'), (1, '1'), (2, '0'), (2, '1')]]
```

RDD

key value

Hash partitioning (default)

```
1 pairs = sc.parallelize(nums).partitionBy(3, lambda k: hash(k))
```

RDD func

```
1 print("Number of partitions: ", pairs.getNumPartitions())
2 pairs.glom().collect()
```

```
Number of partitions:  3
[[(0, '0'), (0, '1')], [(1, '0'), (1, '1')], [(2, '0'), (2, '1')]]
```

Range partitioning

```
1 pairs = sc.parallelize(nums).partitionBy(3, lambda k: k > 1)
```

func

```
1 print("Number of partitions: ", pairs.getNumPartitions())
2 pairs.glom().collect()
```

```
Number of partitions:  3
[[(0, '0'), (0, '1'), (1, '0'), (1, '1')], [(2, '0'), (2, '1')], []]
```

<=1 >1 empty

Comparing RDD, DF, and DS :

In Common :

① Structured and unstructured data

② txt, csv, JSON, Parquet, ...

③ Immutable

④ "Lazy" !

Aspect	RDDs (v1.0)	DataFrames (v1.3)	Datasets (v1.6)
Abstraction	Low-level API	Higher-level API	Hybrid API
Type Safety	No inherent type safety	Type-safe within columns	Type-safe across objects
Performance	Lower optimization	Optimized operations	Optimized operations
Ease of Use	Requires more code	SQL-like operations	Improved usability
Schema	No fixed schema	Has a structured schema	Allows custom schemas
Serialization	Uses Java Serialization or Kryo	Encoders for efficient serialization	Encoders for efficient serialization
API	Functional operations	DataFrame API	DataFrame API
Optimization	Manual optimization	Catalyst query optimizer	Catalyst query optimizer
Interoperability	Works with any data	Various data sources	Various data sources
Supported Languages	Scala, Java, Python, R	Scala, Java, Python, R (alias DS)	Scala, Java
Use Cases	Low-level control, complex processing	Data manipulation, querying	Structured & unstructured data

Different Querying Options

① Functional APIs :

DF

~~RDD~~

```
df = spark.read.json('hdfs:///dataset.json')

df2 = df.filter(df['name'] = Ramanujan')

df3 = df.sortBy(asc("theory"), desc("date"))

df4 = df.select('year')

result = df4.take(10)
```

② Structured API:

Declarative !

DF, DS ⇒ Catalyst

Databricks

```
df = spark.read.json('hdfs:///dataset.json')

df.createOrReplaceTempView("dataset")

df2 = df.sql("SELECT * FROM dataset "
             "WHERE guess = target "
             "ORDER BY target ASC, country DESC, date DESC")

result = df2.take(10)
```

Spark SQL