Introduction to MapReduce

Getting maximum temperature

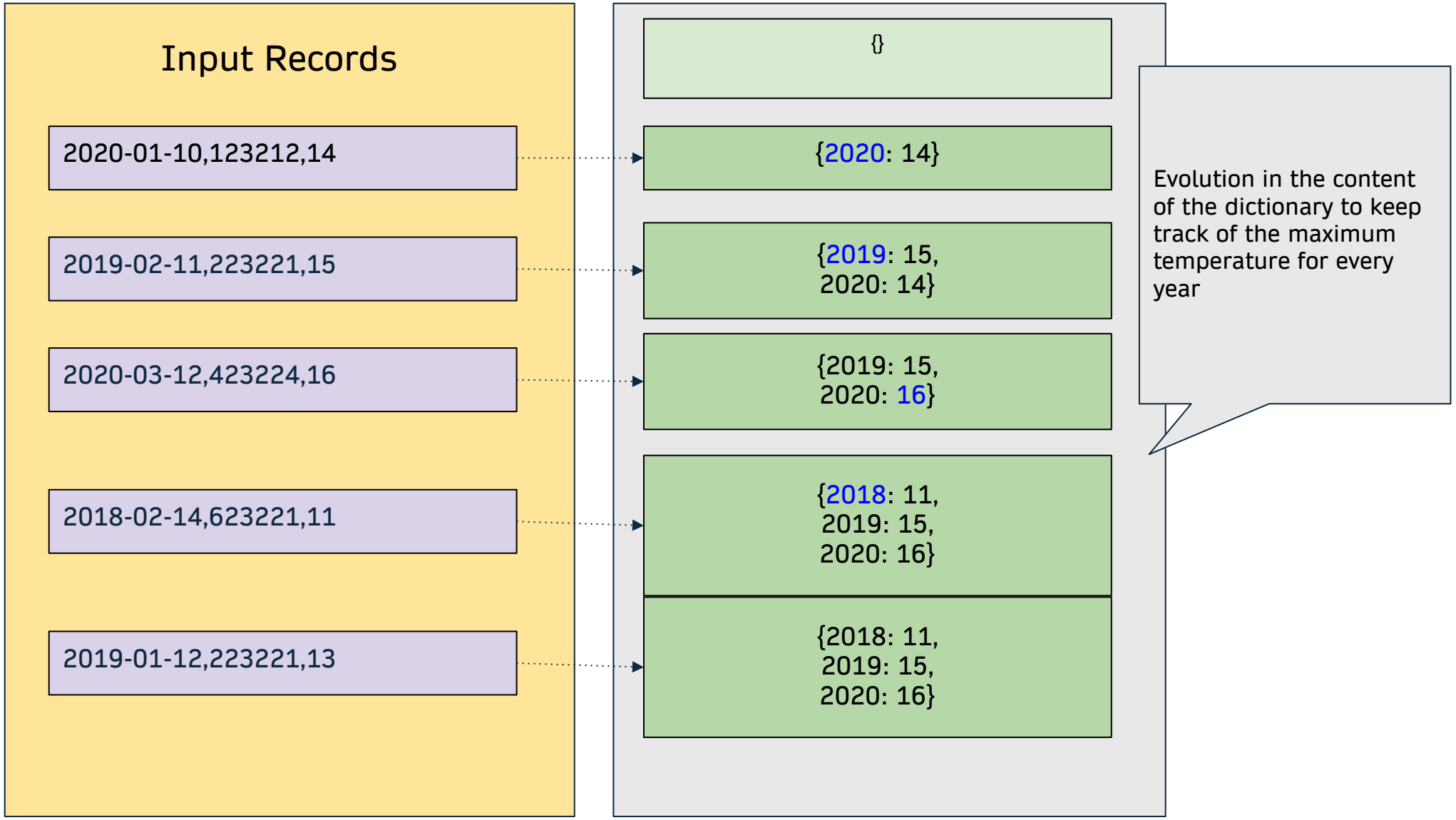
```
2020-01-10,123212,14
2019-02-11,223221,15
2020-03-12,423224,16
2018-02-14,623221,11
2020-01-11,332262,12
2019-01-12,223221,13
2020-02-10,123289,18
2019-02-10,423202,10
```

```
year_max_temp_dict = {}

for line in open('/tmp/weather.csv', 'r').readlines():
  full_date, zip, temperature = line.split(",")
  temperature = int(temperature)
  year, month, day = [int(field) for field in full_date.split("-")]
  if year not in year_max_temp_dict:
    curr_max_temp = temperature
  else:
    existing_temp = year_max_temp_dict[year]
    curr_max_temp = max(temperature, existing_temp)
  year_max_temp_dict[year] = curr_max_temp

for year in year_max_temp_dict:
  print("Maximum temperature for the year {} is {}".format(year, year_max_temp_dict[year]))
```

Getting maximum temperature



Issues with single process application

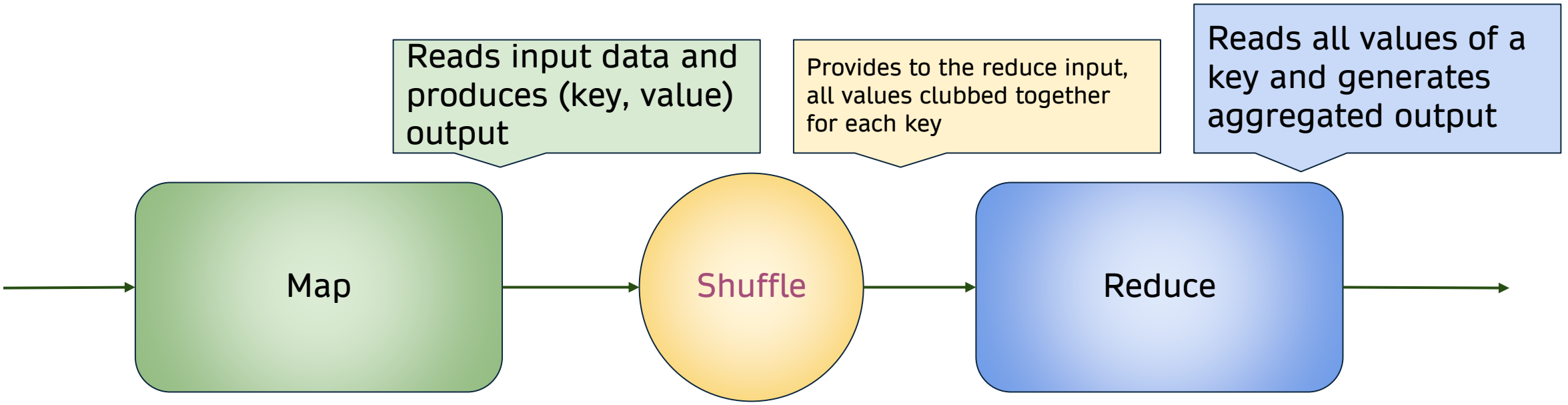
- Large file may not fit on the local machine
- Even if we find a way to read large file:
  - Low speed of execution
  - May not have enough memory to keep the dictionary if keys are in huge number
- Need a better way to perform complex computations on data at petabyte scale

Map Reduce: Flow of Operations

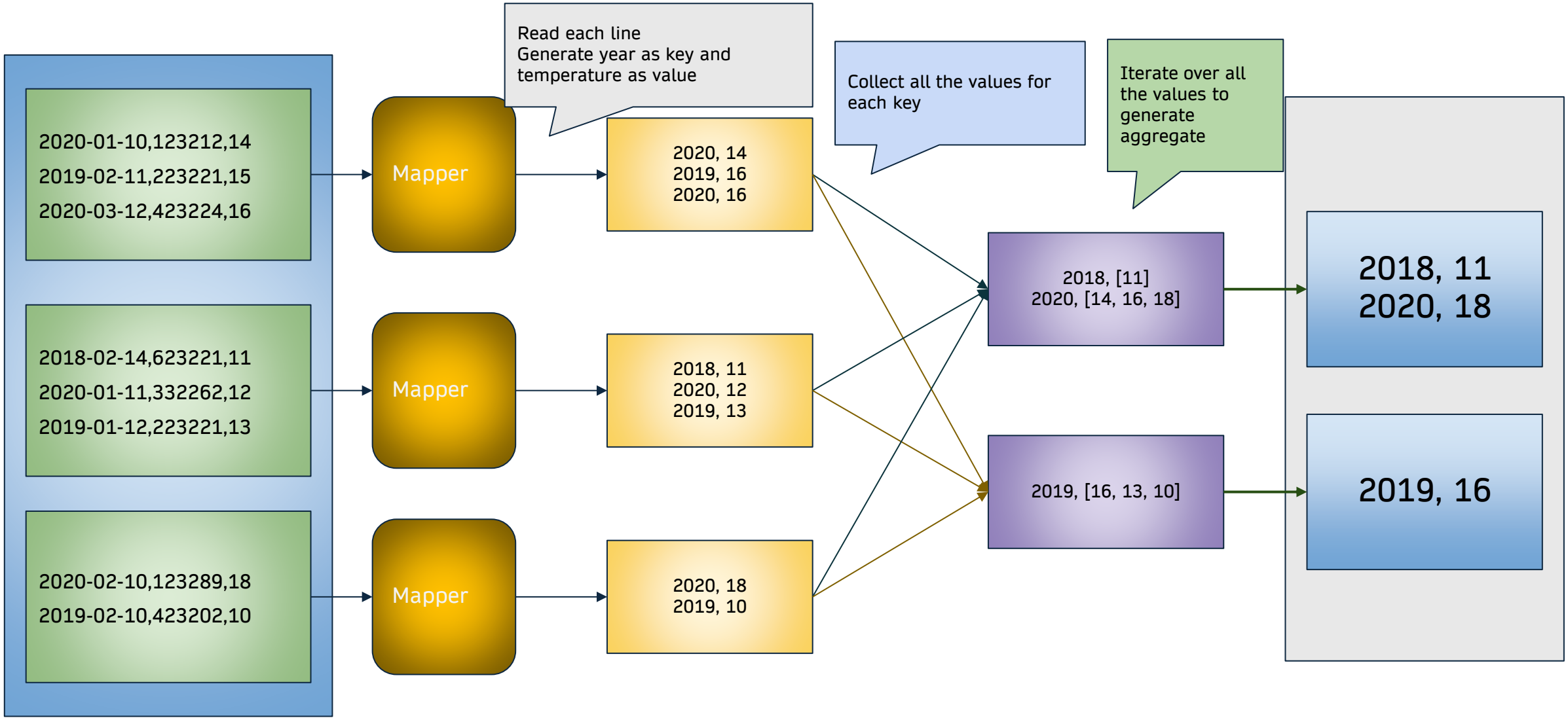
SELECT year, MAX(temperature) FROM <table> GROUP BY year;

1. Map Phase
   a. Work on the input dataset
   b. Run multiple processes in parallel spread across multiple machines of a cluster, each working on mutually exclusive portion of the data
   c. Produce output in the form of (key, value)
   d. Key: field over which aggregation should take place [here: it is year]
   e. Value: field to be aggregated [here: it is temperature]
2. Shuffle
   a. Collect all the values for each key and send them to the next phase ie. reduce
3. Reduce Phase
   a. Get the key and the list of the values
   b. Iterate over the values to generate the aggregate.

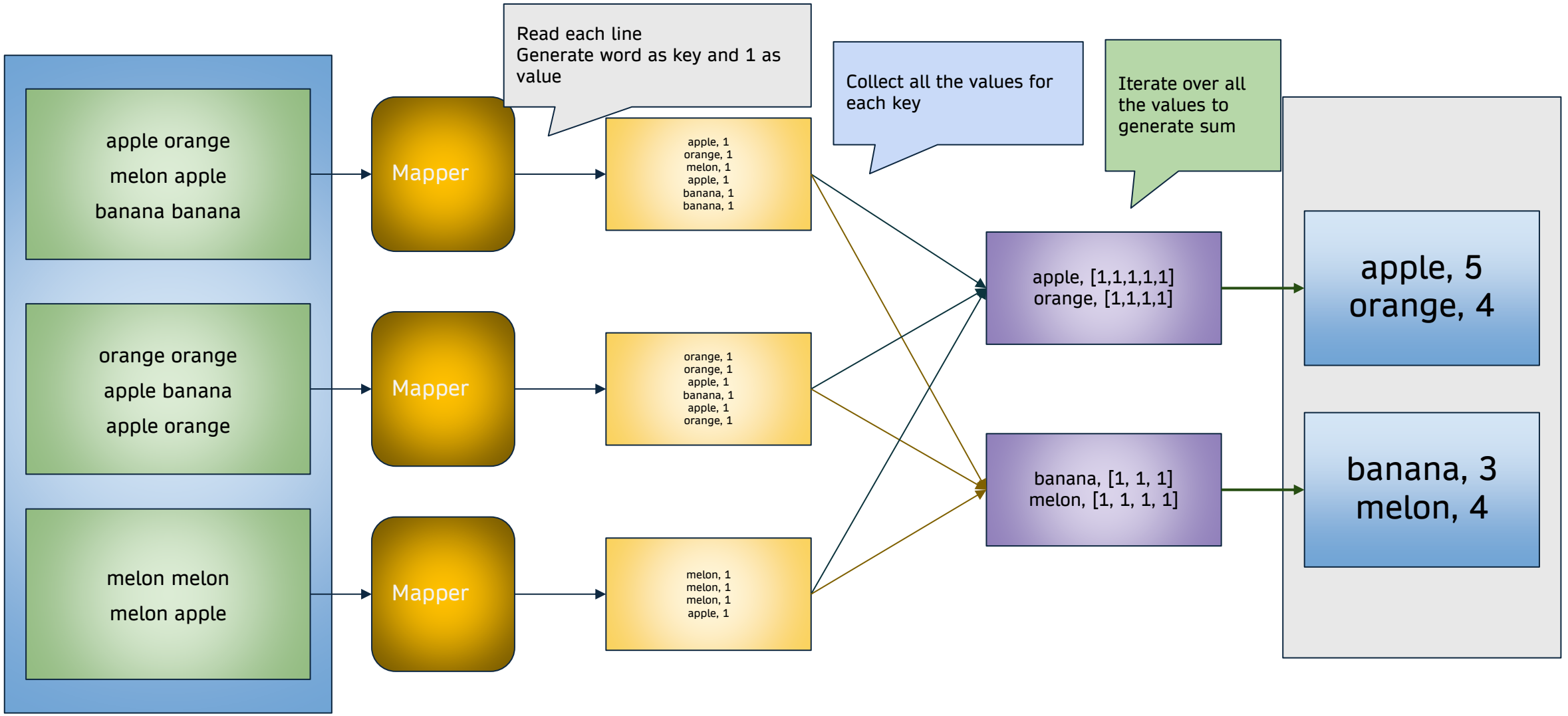
Map Reduce: Flow of Operations



Map Reduce:
Getting maximum temperature



Map Reduce: Generate word count



Why MapReduce

- Process massive amount of data
- Use of commodity hardware
- Jobs can succeed even if some tasks fail (as tasks are re-launched or re-tried)
- Speculative Execution
- Support for config to further tune or customize the behaviour
- Loose coupling with cluster manager

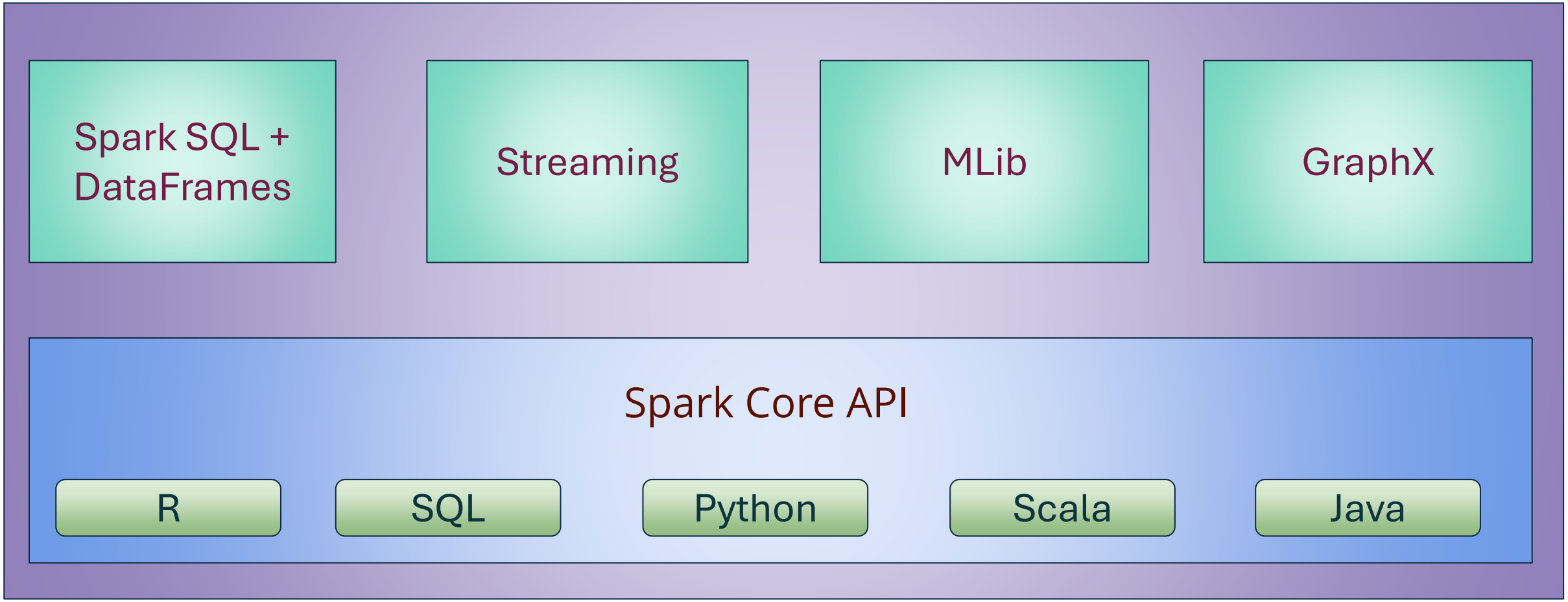
Why NOT MapReduce

- Increased latency due to high usage of local disk (for map output) or HDFS/similar storage (for reduce output)
  - Not optimised for iterative processing
  - Not suited for interactive queries
  - Not designed for low latency jobs like streaming/real time
- Need different tools for different processing
  - **Hive** of SQL
  - **Mahout** for Machine Learning
  - **Giraph** for Graph Processing

What is Spark

- **Computing Engine**
  - Provides fast, in-memory, distributed computation without being strongly coupled to a specific storage system
  - Can connect to wide variety of storage systems
- **Unified Data Processing**
  - Same engine for
    - Streaming
    - Machine learning, and
    - SQL
  - Consistent API
- **Set of libraries for big data**
  - Supports standard (like Spark SQL, MLib) libraries, and
  - third party libraries

Spark Ecosystem



Why Spark

- Evolution of Map Reduce, hence consists of the benefits of Map Reduce and tries to address the shortcomings.
- Parallel Processing
  - Can handle huge data volume by parallel processing on worker machines (called nodes)
  - Linearly scales with addition of more machines
  - Can work on commodity hardware, no requirement of any specialized machine
- Low disk I/O
  - Low usage of local disk or HDFS/similar storage (for intermediate output)
  - Hence, optimised for iterative processing
  - Designed for low latency jobs like streaming/real time
- Unified data processing
  - Supports wide variety of data processing requirements with the same computation engine

Why Spark

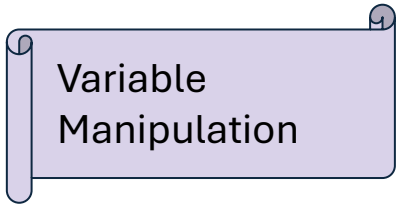
- Support multiple languages
  - Out of the box support for different languages: Scala, Python, Java, R and SQL
- Active
  - Most active open source project in its domain
- Integration
  - Seamless integration with wide variety of data storages
- Compact
  - Less lines of code, more focus on business logic

Next:
Concepts of Spark: RDD

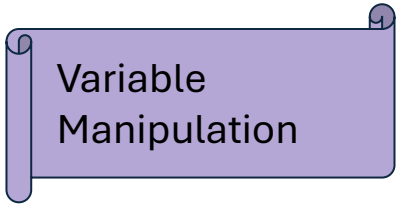
Terminology to be discovered

1. Resilient Distributed Dataset
    a. Partitions
2. Transformations
    a. Tasks
3. Lineage

Variable manipulation

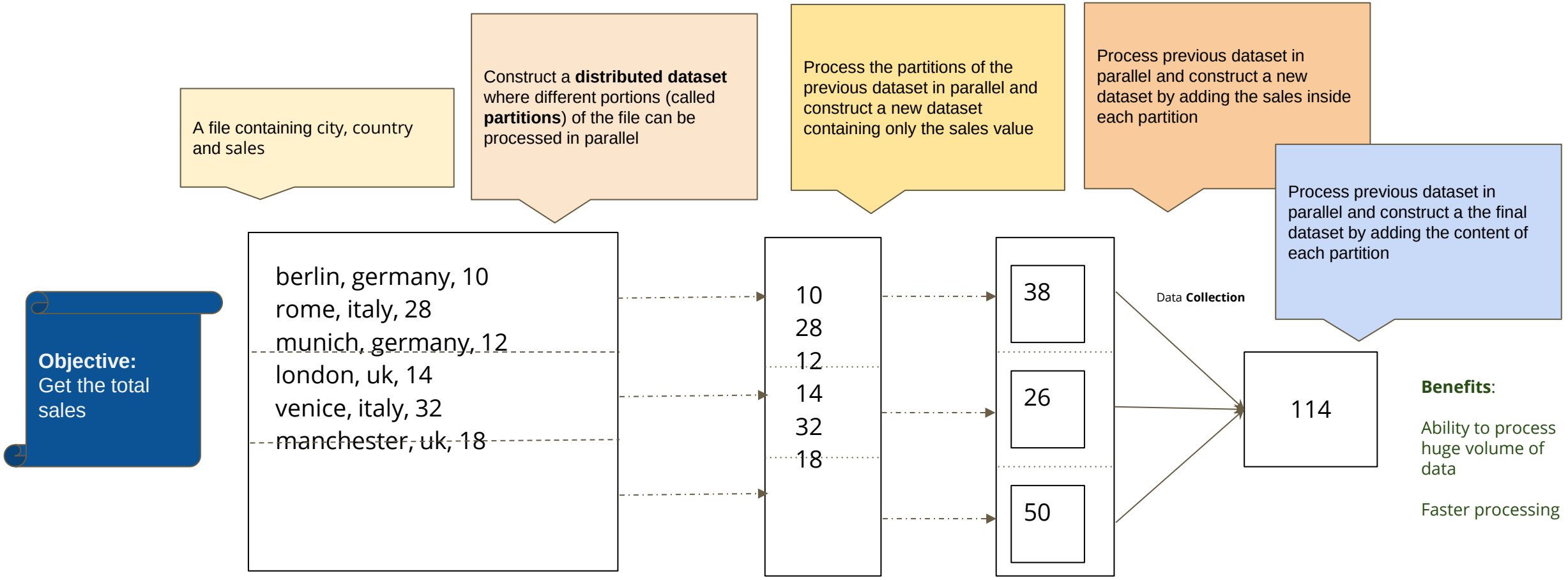



```
u = 1
v = u * 2
w = 10
x = v + w
```

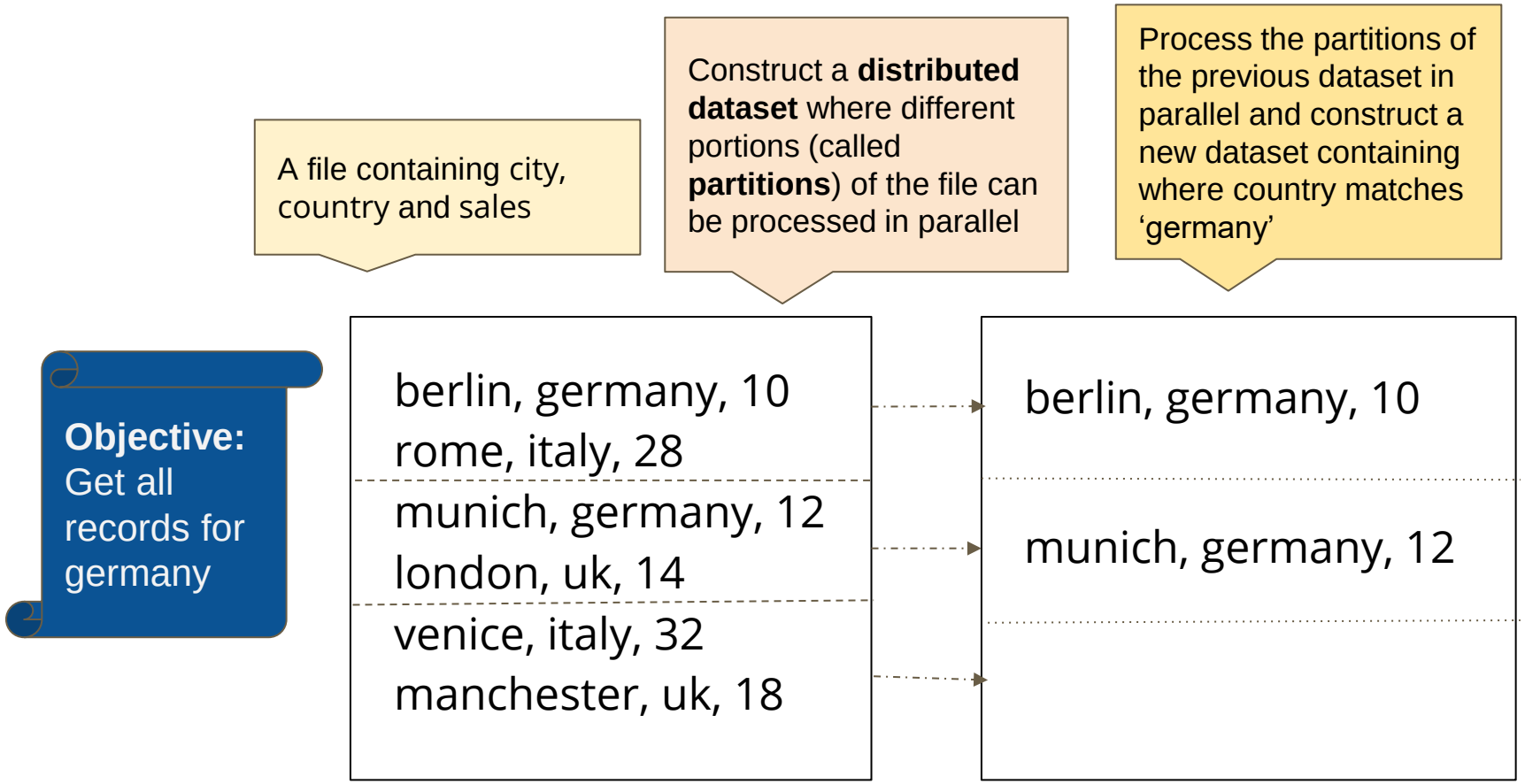



```
u = [10, 20, 12, 14]
v = map(lambda i: i*2, u)
w = [10, 11, -1, 18]
x = map(lambda (i, j): i + j, zip(v,w))
```

Resilient Distributed Dataset (RDD)



Resilient Distributed Dataset (RDD)



Resilient Distributed Dataset (RDD)

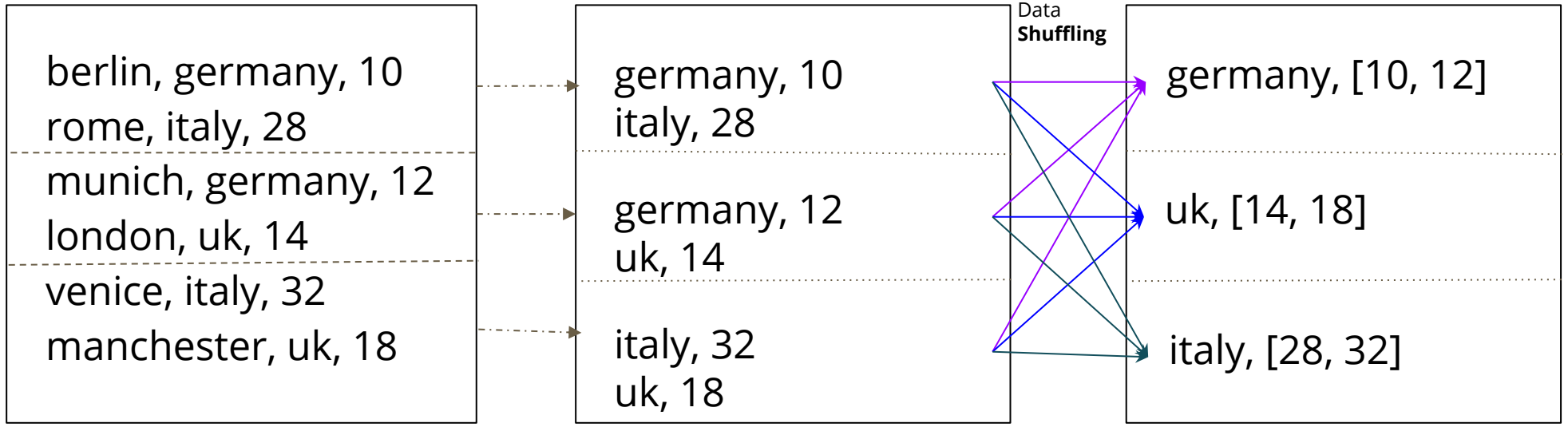
A file containing city, country and sales

Construct a **distributed dataset** where different portions (called **partitions**) of the file can be processed in parallel

Process the partitions of the previous dataset in parallel and construct a new dataset containing the country and sales value

Process previous dataset in parallel and construct a new dataset by collecting all the values for a country (involves **shuffle**)

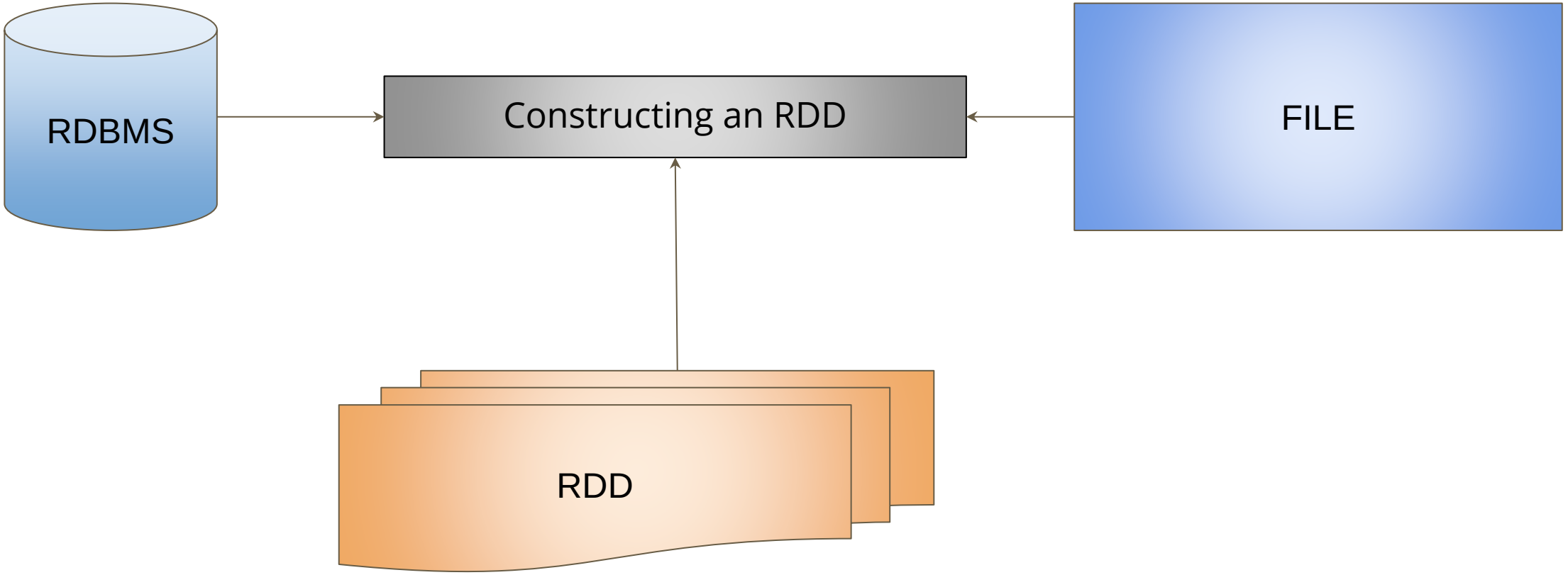
Objective: Get the total sales for each country



Defining RDD

- RDD stands for Resilient Distributed Dataset
- It defines functions to process/manipulate the data
- It partitions the input data, so that each partition can be processed in parallel.
- Different partitions can be processed on different machines in a cluster
- If processing of any partition fails due to temporary issues like network outage or machine getting down, it is automatically restarted, making it resilient to recoverable failures.
- RDDs are immutable, once created we need to construct another RDD if we wish to change the data referred by it.

Creating RDDs



Variable manipulation and lineage

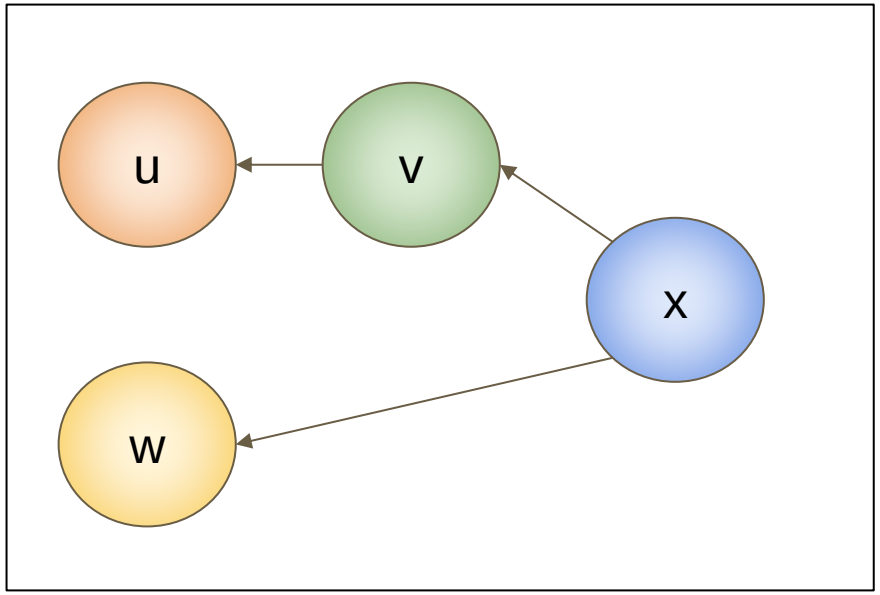
Variable Manipulation

```
u = 1
v = u * 2
w = 10
x = v + w
```

Variable Manipulation

```
u = [10, 20, 12, 14]
v = map(lambda i: i*2, u)
w = [10, 11, -1, 18]
x = map(lambda (i, j): i + j, zip(u,w))
```

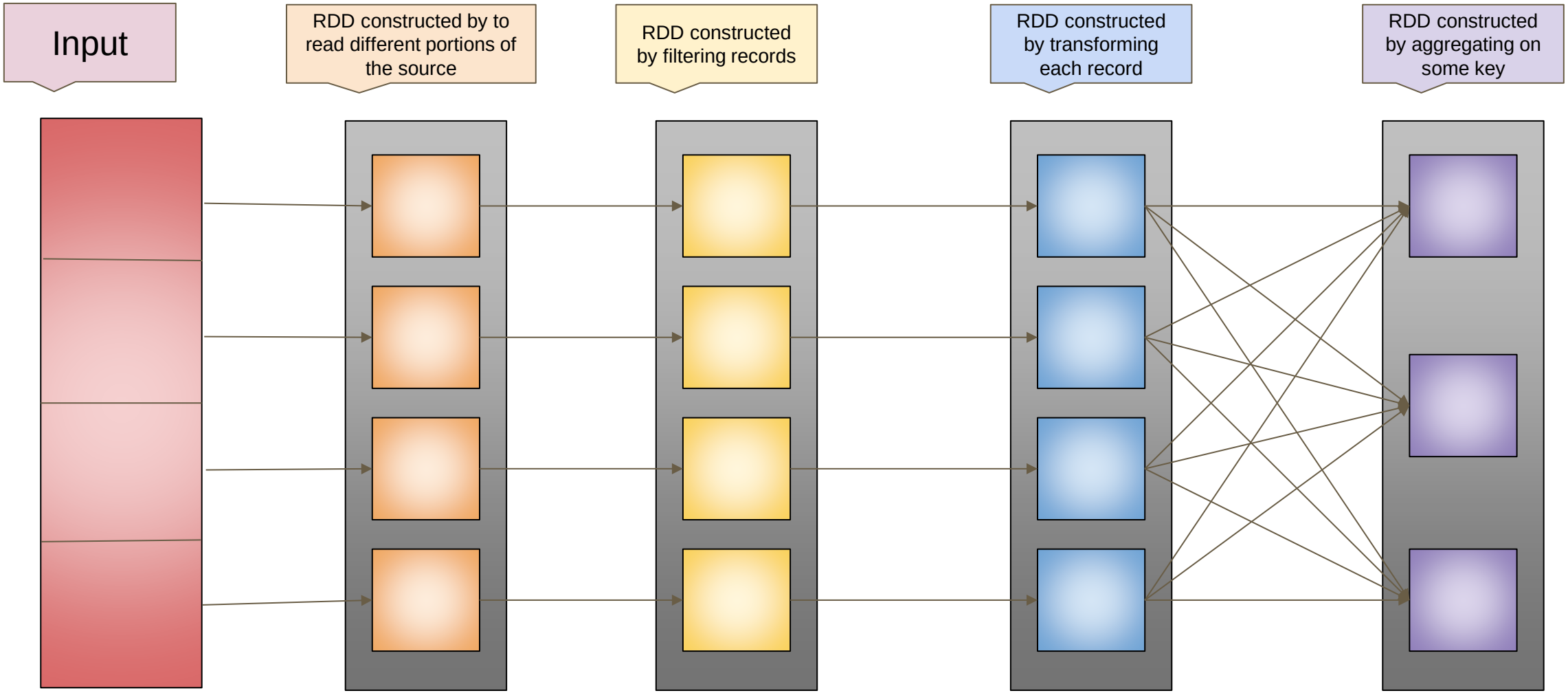
Lineage / Dependency



RDD lineage

- Whenever an RDD is created or transformed, the lineage information is updated and kept in-memory by spark.
- This lineage keeps the information about what has to be done when result is asked for.
  - There are two types of operations on an RDD, which we'll discuss in detail later:
    - Transformation: They update the lineage
    - Action: They execute the actual lineage
  - Lineage helps in this concept of lazy evaluation (more coming up in later videos)
- Lineage is essential to avoid reprocessing entire source dataset if one of the partitions of the RDD fails.

RDD Lineage


Input
RDD constructed by to read different portions of the source
RDD constructed by filtering records
RDD constructed by transforming each record
RDD constructed by aggregating on some key

RDD Q&A

Q1. Why do we need to create multiple RDDs. Why can't we update existing RDD?

RDDs are immutable, once created they can not be modified. To update the RDD, reconstruct it or create another RDD to transform it. Also, RDDs do not contain the data, and computation happens only when you need the final output (when you call the **action** [more to be discussed later]), so reconstructing an RDD before the trigger has no performance penalty. Also, Spark performs optimisation to compact operations together, many a times, adding extra transformation (not involving shuffle) also does not impact the performance.

Q2. If multiple RDDs are created, one corresponding to each transformation, wouldn't the data be duplicated across each RDD?

RDDs do not contain the data, rather they maintain the pointer to the actual data (for the RDD reading from the source) or the information about what transformation to perform on the parent RDD. The RDD lineage maintains the information about what transformation data will undergo when **action** [more to be discussed later] is called. When the user wants the final output, pipeline (lineage) of the RDD is triggered and data is processed on the fly. So, intermediate output may be produced, but they will be temporary and the entire data of the source won't be duplicated.

RDD Q&A

Q3. What does **resilience** mean in RDD?

Each partition is processed in parallel and one of the process fails, it is automatically restarted (for the configured number of times). Also, only the portion of the data that process was supposed to read will be scanned and the entire source won't be re-read. The lineage is traced back and not all the parent RDDs are recomputed, only the partitions of the parent RDD which had any role in creating the failed partition of the child RDD are recomputed. So, failure while processing a partition is handled transparently without user even noticing it on high level.

Q4. What is benefit of being **distributed** for an RDD?

Each partition is processed in parallel and one of the process fails. For each partition, one **task** [more to be discussed later] is executed by **executor** [more to be discussed later] (a process running on the worker machines). These tasks can be spread across multiple machines of the cluster. Hence, being distributed helps getting the results of data analysis faster or meeting the timelines when performing heavy-weight data analysis by scaling out the cluster.

Next:
Concepts of Spark: Architecture

Terminology to be discovered

1. SparkContext
    1. Config
2. SparkSession
    1. Config
3. Worker Nodes
4. Executors
5. Driver Machine
6. Cluster Managers

Spark Context

- Main entry point for Spark functionality.
- A **SparkContext** represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
- With default constructor, it loads settings from system properties
  - **SparkContext**()
- It can be constructed with a Spark Config object as a parameter describing the application configuration.
  - **SparkContext**(**SparkConf** config)
    - Any settings in this config overrides the default configs as well as system properties.

Spark Context

- Only one SparkContext should be active per JVM.
- You must stop() the active SparkContext before creating a new one.
- Some of the methods available on the SparkContext are:
  - **textFile**(String path, int minPartitions)
    - Read a text file from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI, and return it as an RDD of Strings.
  - **addJar**(String path)
    - Adds a JAR dependency for all tasks to be executed on this SparkContext in the future.
  - **addFile**(String path)
    - Add a file to be downloaded with this Spark job on every node.

Spark Session

Overview:

•SparkSession is the primary entry point to Spark functionality, introduced in Spark 2.0.

•It simplifies access to Spark's features, integrating previous contexts like SparkContext, SQLContext, and HiveContext.

Key Features:

•Creates and manages DataFrames and Datasets.

•Allows execution of SQL queries and integration with machine learning and streaming capabilities.

•Single point of entry for all Spark operations.

Example: Creating a SparkSession

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder \
    .appName("DataFrameExample") \
    .getOrCreate()

# Example DataFrame
data = [("John", 25), ("Alice", 30), ("Bob", 35)]
df = spark.createDataFrame(data, ["Name", "Age"])

# Show and filter the DataFrame
df.show()
filtered_df = df.filter(df.Age > 30)
filtered_df.show()

# Aggregation
agg_df = df.groupBy("Name").avg("Age")
agg_df.show()

# Stop SparkSession
spark.stop()
```

Spark Context Vs Spark Session

Feature	SparkContext	SparkSession
API Level	Lower-level API (RDDs)	Higher-level API (DataFrames, Datasets)
Functionality	Core distributed computing	Combines SQL, streaming, and ML capabilities
Ease of Use	Requires more setup code	Simpler, less boilerplate needed
Version Compatibility	Primary in Spark 1.x	Introduced in Spark 2.x, recommended for newer versions
Multiple Instances	One per JVM	Multiple can exist in a single application
Data Abstraction	Works with RDDs	Works with DataFrames, Datasets, and SQL

• **Preference for SparkSession**: In modern Spark applications (2.x and later), SparkSession is recommended due to its unified interface and ease of use.
•**Access to SparkContext**: Although SparkSession is preferred, you can still access SparkContext for specific RDD operations when needed.

Configuring SparkSession

Example: Basic Configuration

```python
from pyspark.sql import SparkSession

# Create a SparkSession with configurations
spark = SparkSession.builder \
    .appName("ConfigExample") \
    .config("spark.sql.shuffle.partitions", "50") \
    .config("spark.executor.memory", "4g") \
    .config("spark.driver.memory", "2g") \
    .getOrCreate()
```

Common Configuration Options:

- `spark.sql.shuffle.partitions`: Number of partitions to use when shuffling data.
- `spark.executor.memory`: Amount of memory allocated for each executor.
- `spark.driver.memory`: Amount of memory allocated for the driver.
- `spark.sql.warehouse.dir`: Directory for Spark SQL warehouse.

Tips for Configuration:

- Adjust configurations based on your workload and cluster resources.
- Use Spark UI to monitor performance and make adjustments as needed.

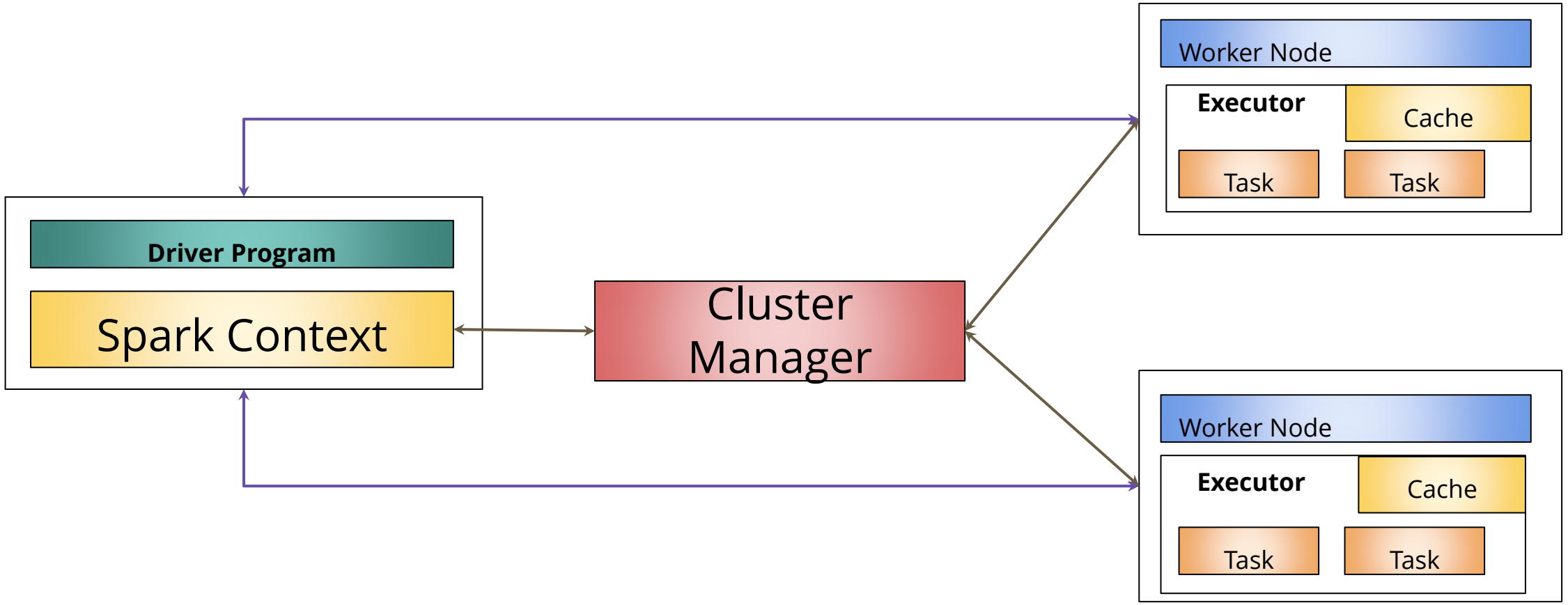
Cluster Manager

- An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
- The system currently supports several cluster managers:
  - **Standalone**
    - simple cluster manager included with Spark that makes it easy to set up a cluster.
  - **Apache Mesos**
    - a general cluster manager that can also run Hadoop MapReduce and service applications.
  - **Hadoop YARN**
    - the resource manager in Hadoop 2.
  - **Kubernetes**
    - an open-source system for automating deployment, scaling, and management of containerized applications.
  - A third-party project (not supported by the Spark project) exists to add support for **Nomad** as a cluster manager (now deprecated).

Spark Architecture

- There are lots of worker nodes where a service called executor runs. It owns resources and runs tasks within it. There could be multiple executors running on one machine.
- To run on a cluster, the SparkContext, created on the driver node can **connect** to several types of **cluster managers** (either Spark's own standalone cluster manager, Mesos or YARN), which allocate resources across applications.
- Once connected, Spark acquires **executors** on nodes in the cluster, that run computations and store data for the application.
- Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors.
- Finally, SparkContext sends tasks to the executors to run.

Spark Architecture



Core Concepts of Spark

- **RDD**
  - Resilient (fault tolerant)
  - Distributed
  - Dataset
- **Spark Context**
  - Represents the connection to a Spark cluster
  - Used to create RDDs, accumulators and broadcast variables on that cluster
- **Cluster Manager**
  - Allocates resources to the application
  - Provides scheduling of the applications on shared cluster

Next:
Tips on tuning Spark Application
[Choosing size of executors]

Setting configurations in Spark Application

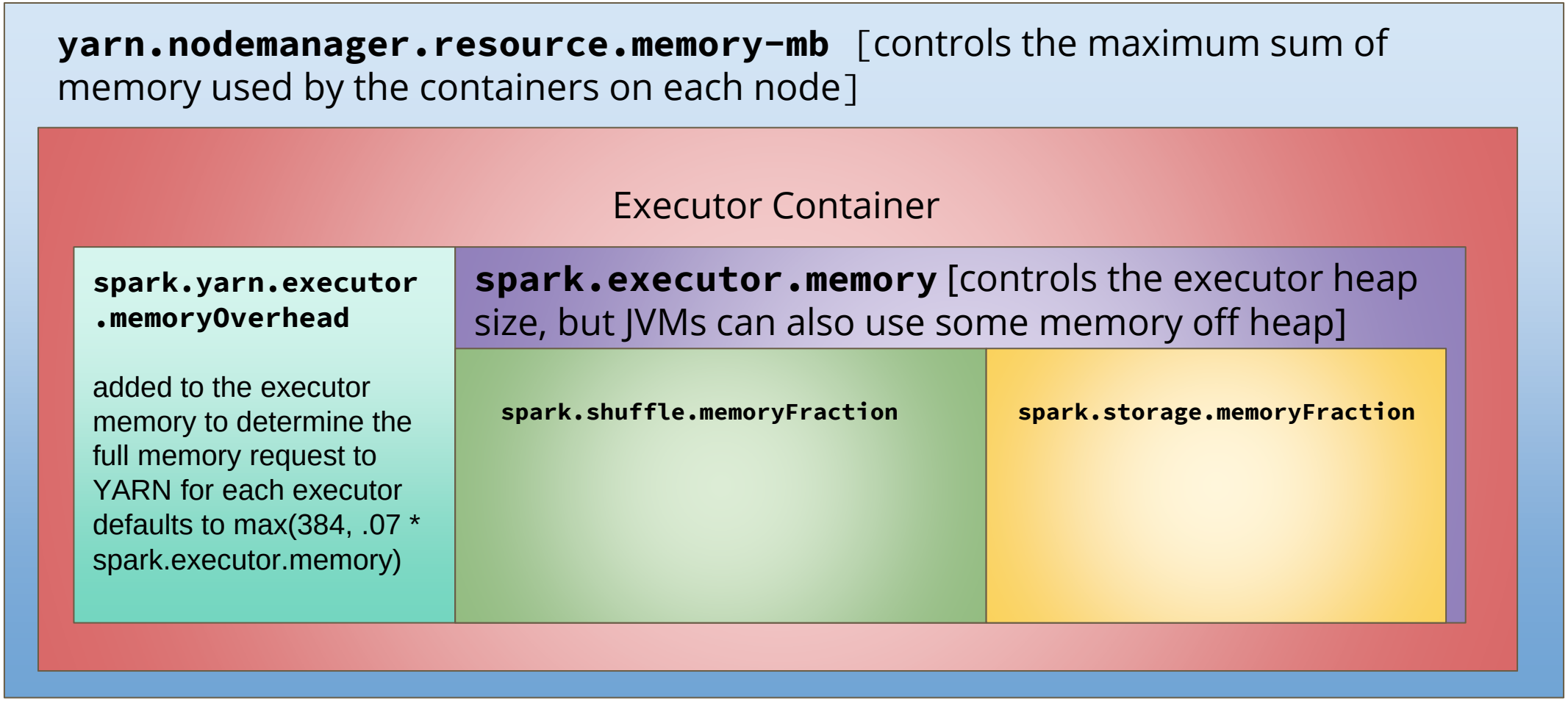
In an application

```
conf = SparkConf()
conf.set("spark.task.maxFailures", "2")
conf.set("spark.speculation", "true")
sc = SparkContext(conf) ...
```

At runtime (spark-submit)

```
spark-submit \
--conf spark.task.maxFailures=2 \
--conf spark.speculation=true ...
```

Spark Memory Model



Starting point of the number of executors

Deciding the number and size of executors

- **--num-executors**
- **--executor-cores**
- **--executor-memory**

Take care of below points:

- **--executor-cores** decides number of tasks running in that executor in parallel

Starting point of the number of executors

- For better **HDFS throughput**, recommendation is to use 4-5 --executor-cores
- A **thin** executor with lesser --executor-core and --executor-memory, **cache** of the executors will be **underutilised.** We won't benefit from running multiple tasks in the JVM. Eg: multiple copies of the broadcast variables will be sent.
- A **fat** executor with higher --executor-core and --executor-memory will lead to **poor throughput**
- In YARN
  - **Application Master** will also need 1 CPU core
  - **7%** overhead memory is provided to the executors

Starting point of the number of executors: Sample

- With 6 machines each having 16 cores and 64 GB of RAM
- Leave out 1 core and 1 GB of RAM for the OS related process on those machines
- With 5 tasks per executor, we can choose to run 3 executors on each machine, so that each task on the executor can use 1 core. Hence **--executor-cores** can be **5**
- A total of 18 executors can be used. Since, application master also uses one core, we can use **--num-executors** to be **17**
- 63 GB of RAM across 3 executors, means each executor can have 63/3 GB (= 21GB) of memory, since YARN introduces overhead of 7%, we can choose **--executor-memory** to be **19 GB**. (exact number is around 19.5GB)

Things we overlooked

- We are giving **one executor resource** (5 cores and 19 GB of RAM) to Application Master which is a lot. Most of the time AM is fine with 1 GB and 1 core. So, update the calculation accordingly.
- If job is **not CPU intensive**, each task in the executor need not have 1 CPU core. Hence, even if one executor gets 2 physical CPU cores, it can run **more than** 2 tasks.
- **Monitor** the performance the **RAM utilisation** (using spark JVM metrics) to tune the numbers further.
- If a machine has 22 GB of RAM and we are running 2 executors each using 8 GB (including overhead), then a request for running third executor with 8 GB RAM will not granted and the remaining 6 GB will be unutilised. So, ensure RAM on the machine is a close multiple of executor memory (with overhead)

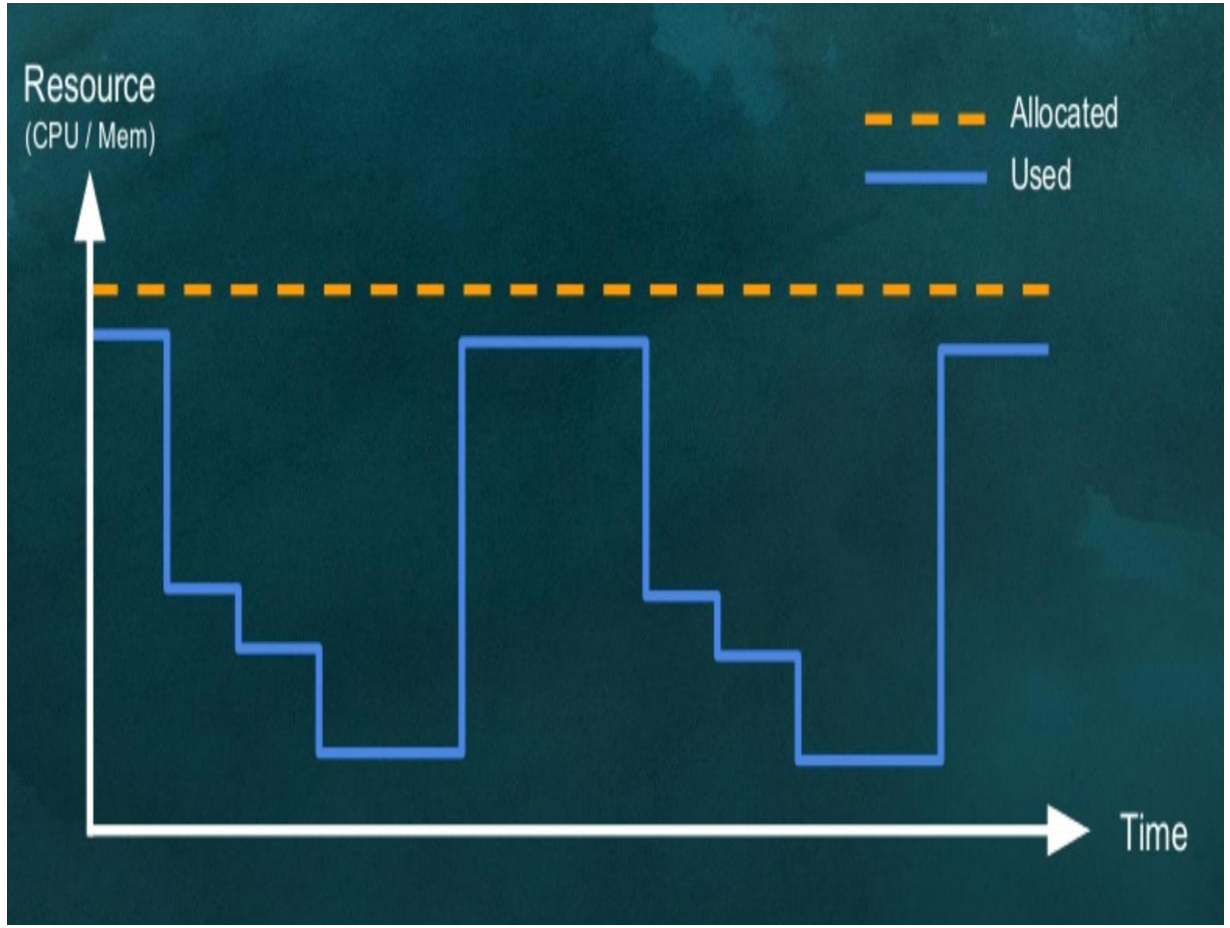
Next:
Tuning Spark Application [Dynamic Allocation]

Dynamic Allocation

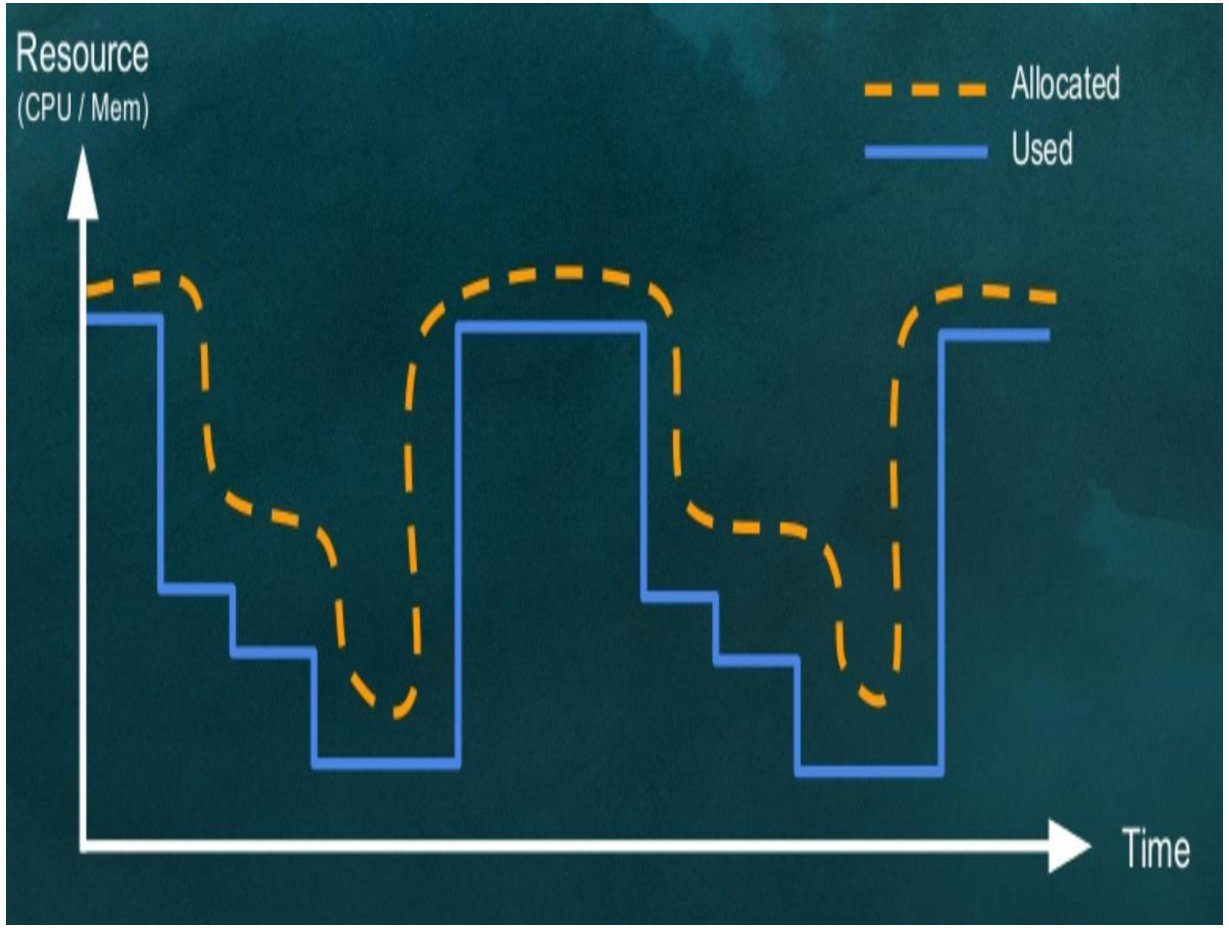
- One spark application consists of multiple jobs, each can have different requirement for the number of executors
- Static values of number of executors can be many a time too high or too low
- Driver should request for more executors if there are tasks pending or kill executors if idle for long.

Dynamic Allocation

Static Allocation

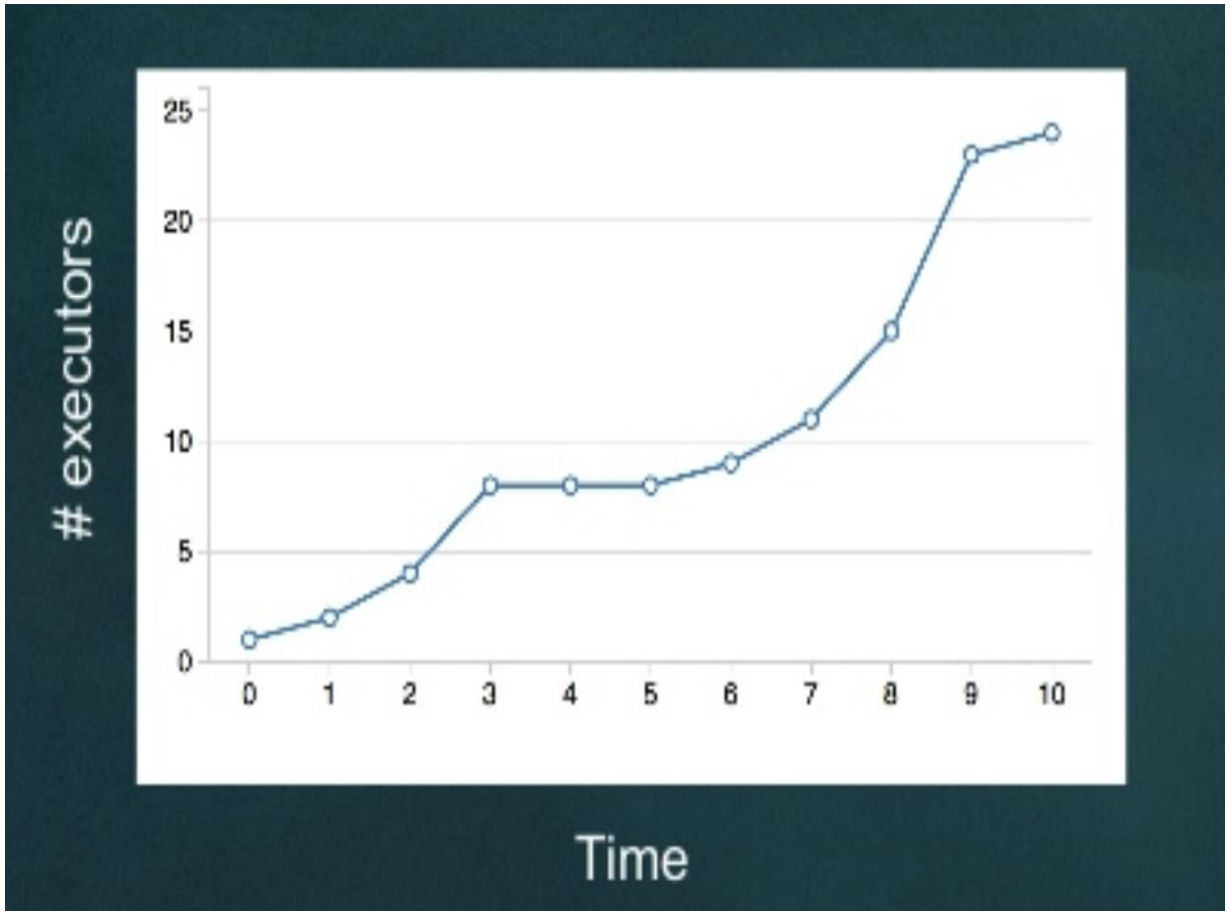


Dynamic Allocation

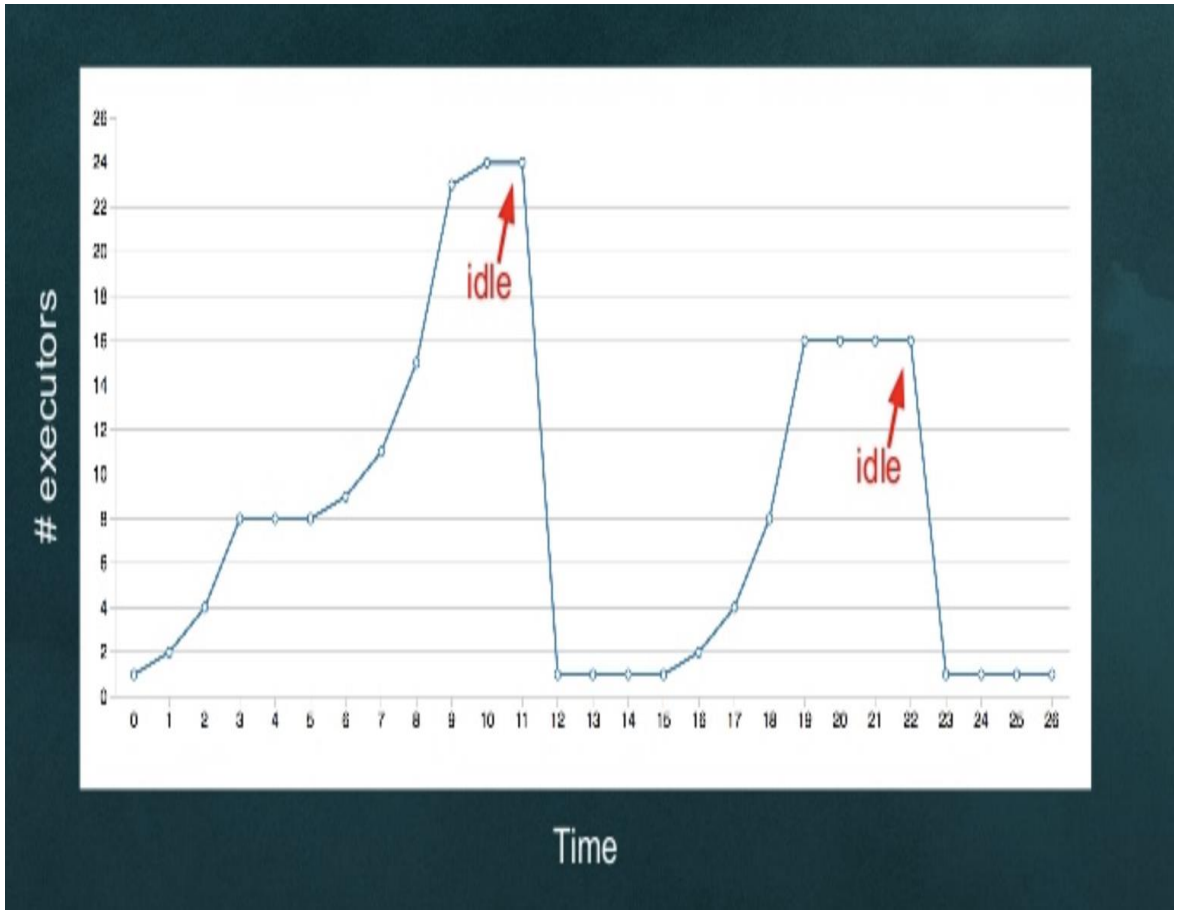


Dynamic Allocation

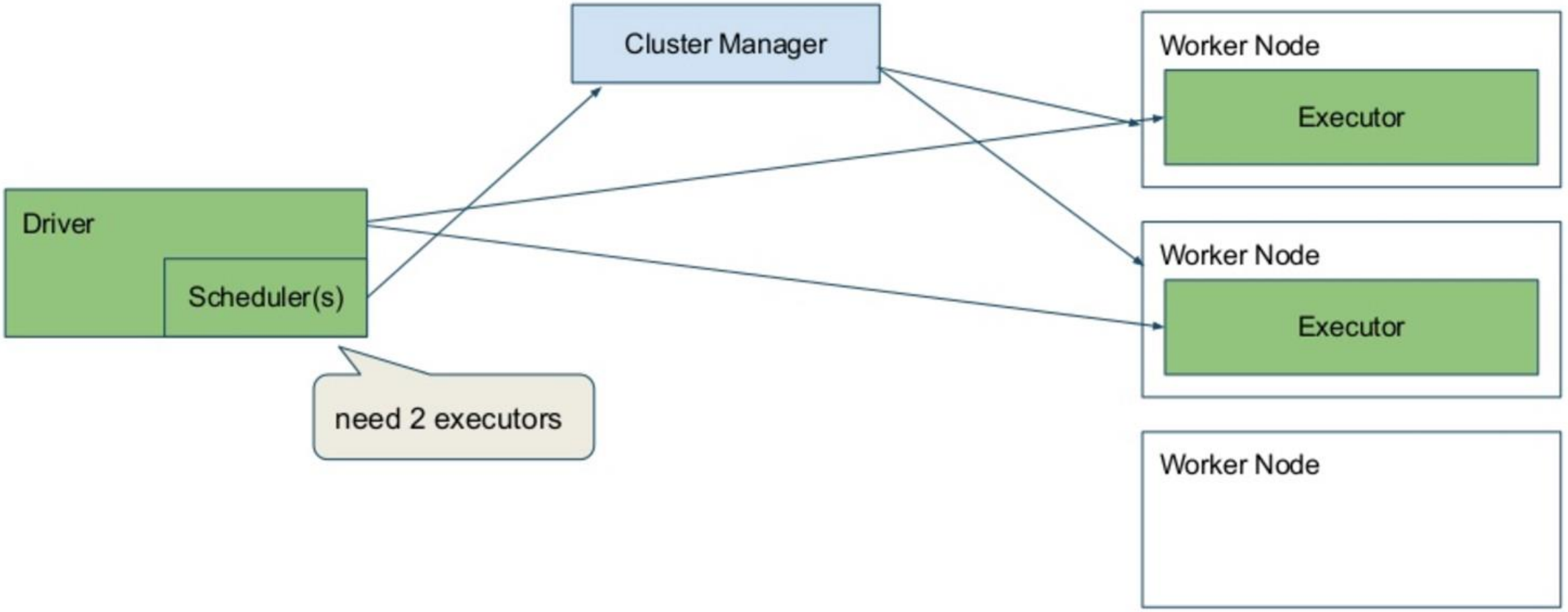
Add executors when more tasks are pending



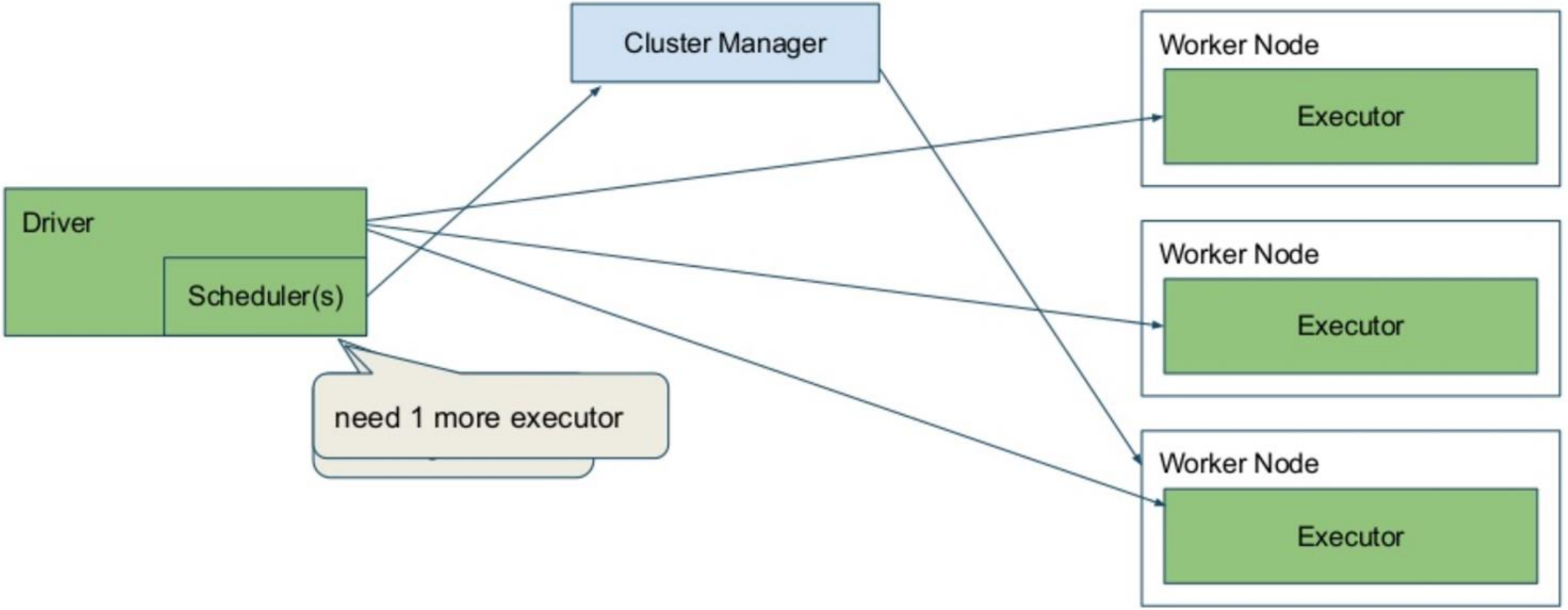
Remove executors when it remains idle for long



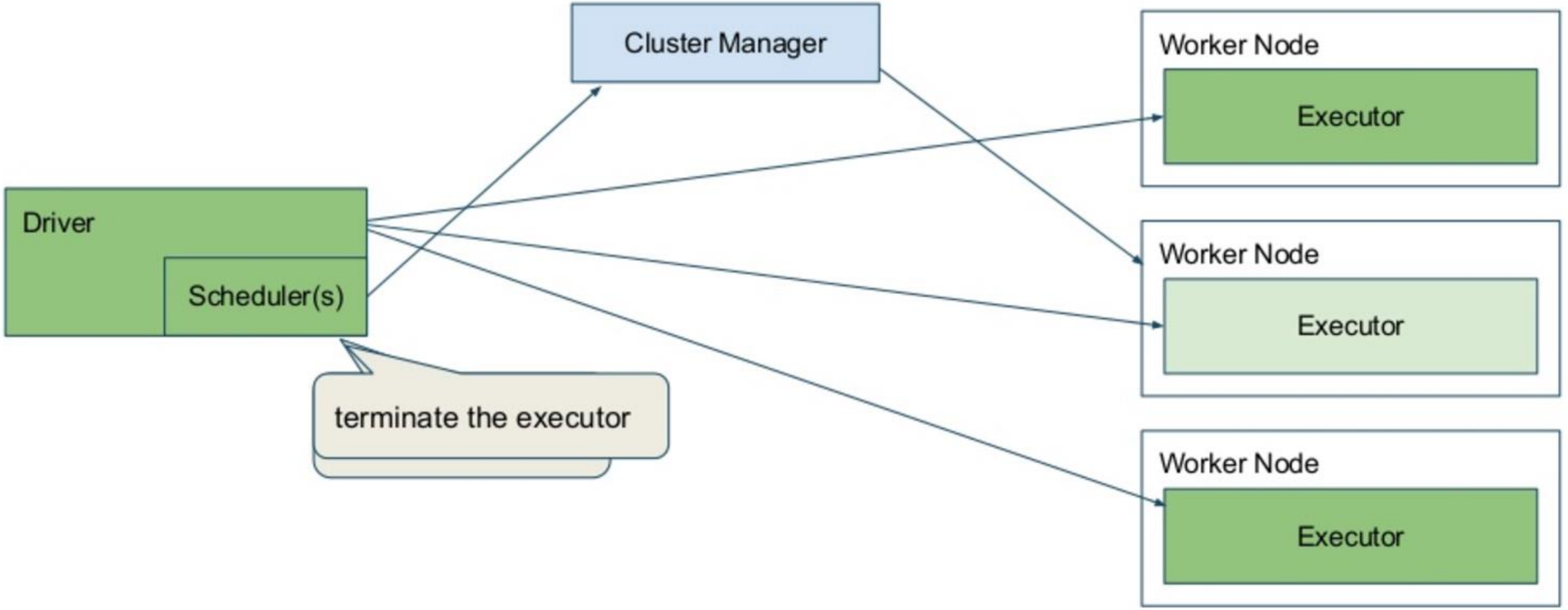
Dynamic Allocation



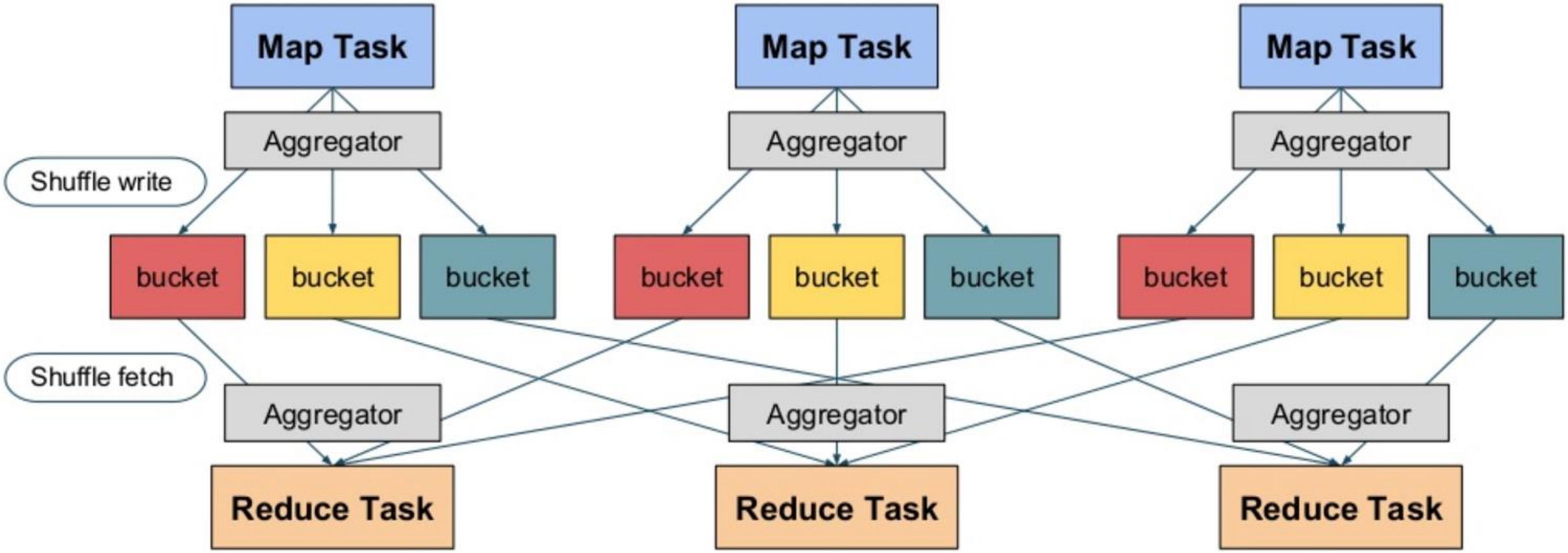
Dynamic Allocation



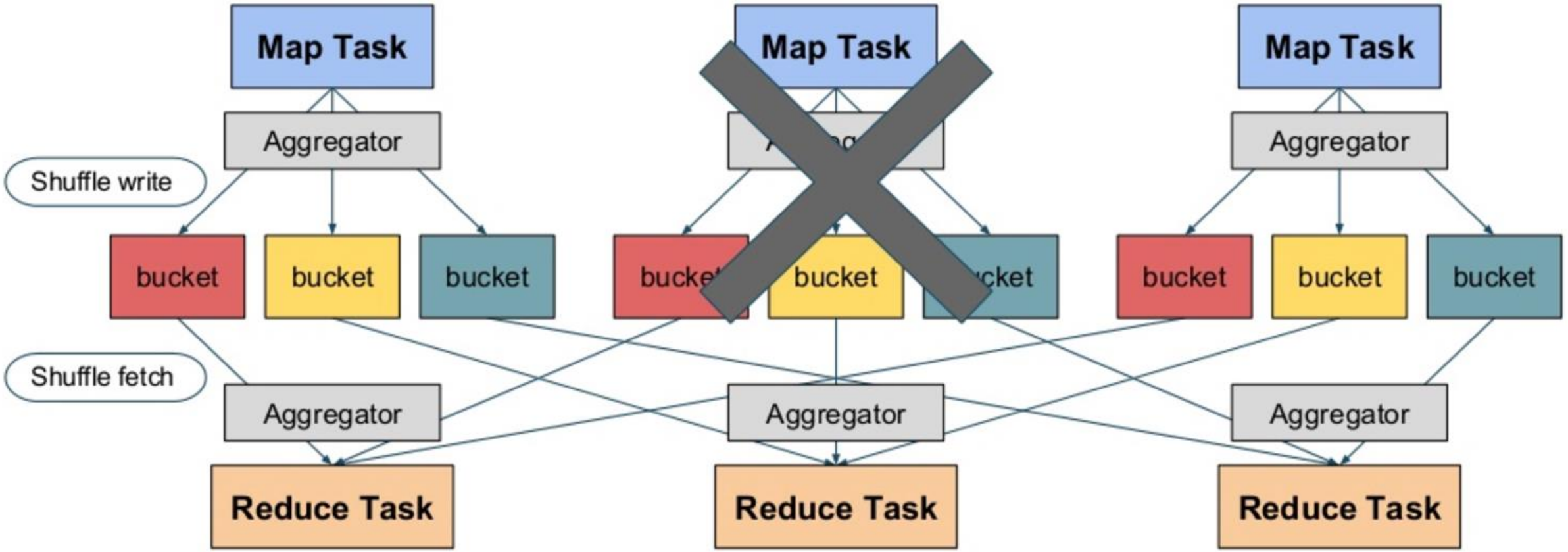
Dynamic Allocation



Dynamic Allocation: External Shuffle Service



Dynamic Allocation: External Shuffle Service



Dynamic Allocation: External Shuffle Service

- If an executor is killed, the **shuffle data** it had also gets **deleted**
- This may impact **further stages** of the job
- **Solution**: Extract shuffle service away from executors
- External shuffle service will manage local aggregated data across shuffles
- **Maintain data** until application is done
- Use carefully for **streaming/long** running tasks, since the disk can run out of space

Dynamic Allocation: Configurations

Configuration	Description
`spark.dynamicAllocation.enabled`	Whether to use dynamic resource allocation
`spark.dynamicAllocation.initialExecutors`	Initial number of executors to run if dynamic allocation is enabled
`spark.dynamicAllocation.maxExecutors`	Upper bound for the number of executors if dynamic allocation is enabled
`spark.dynamicAllocation.minExecutors`	Lower bound for the number of executors if dynamic allocation is enabled

Dynamic Allocation: Configurations

Configuration	Description
`spark.dynamicAllocation.executorIdleTimeout`	If dynamic allocation is enabled and an executor has been idle for more than this duration, the executor will be removed
`spark.dynamicAllocation.schedulerBacklogTimeout`	If dynamic allocation is enabled and there have been pending tasks backlogged for more than this duration, new executors will be requested
`spark.dynamicAllocation.sustainedSchedulerBacklogTimeout`	Time to wait between subsequent executor requests

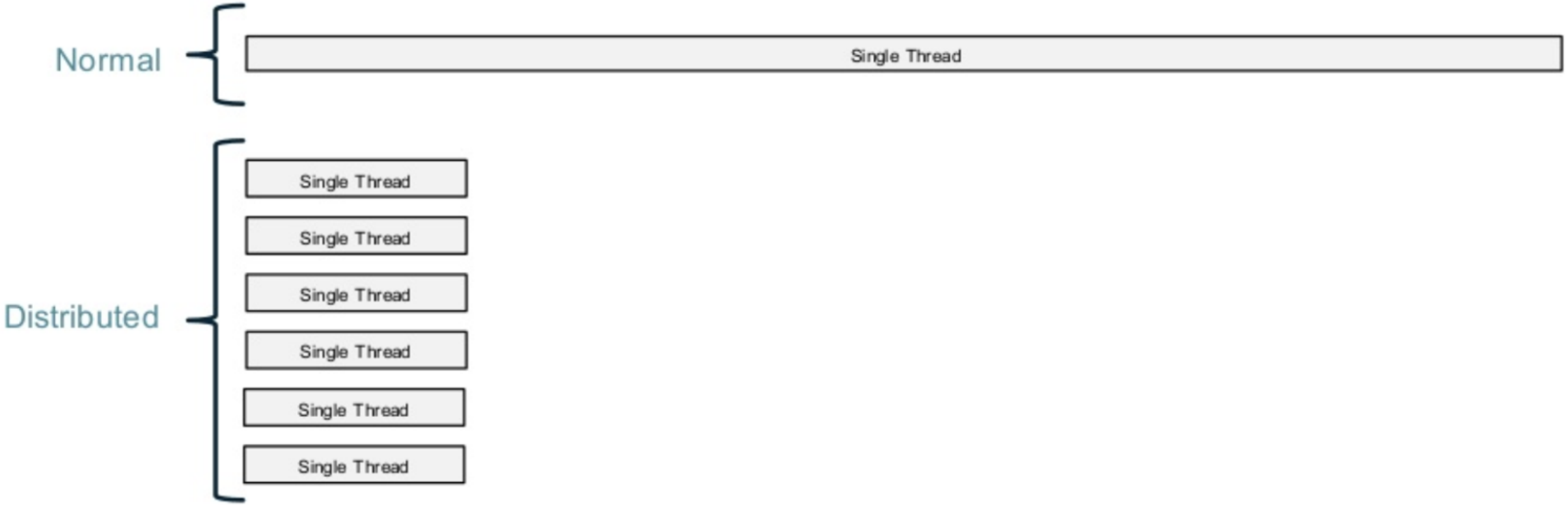
Dynamic Allocation: Configurations

Configuration	Description
`spark.shuffle.service.enabled`	Enables the external shuffle service
`spark.shuffle.service.port`	Port on which the external shuffle service will run

Next:
Tuning Spark Application [Handling Skewness]

Handling Skewness

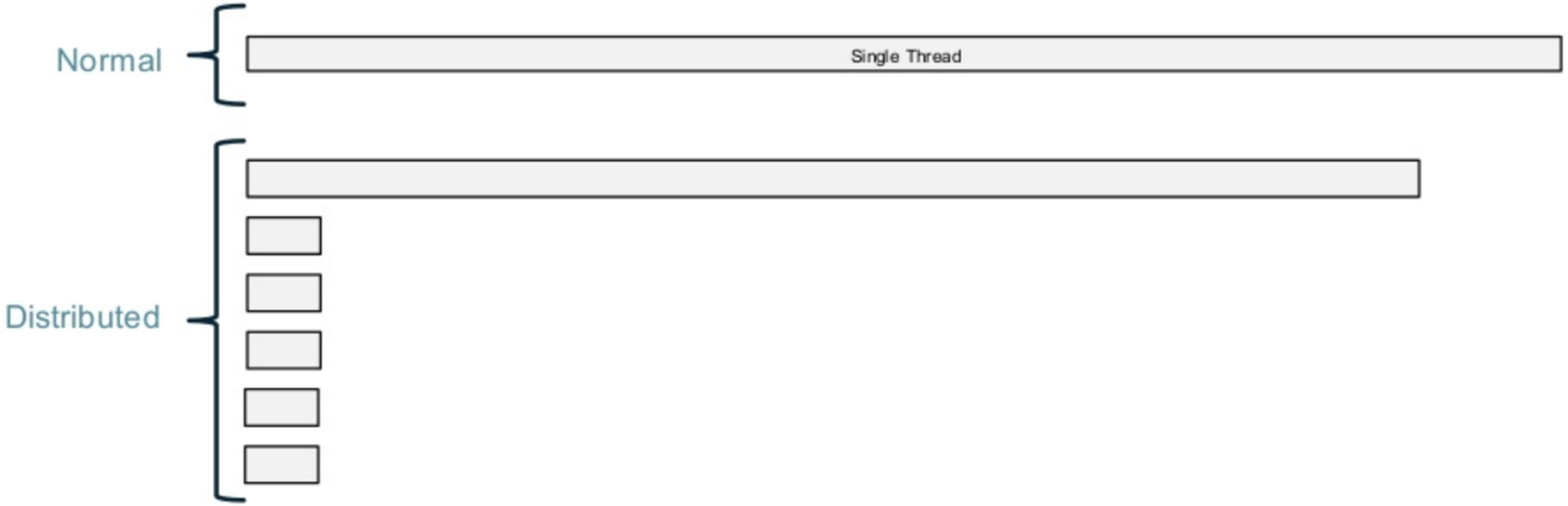
Expectation from distributed processing



Handling Skewness

Result of skewness

- One machine doing the most of the work

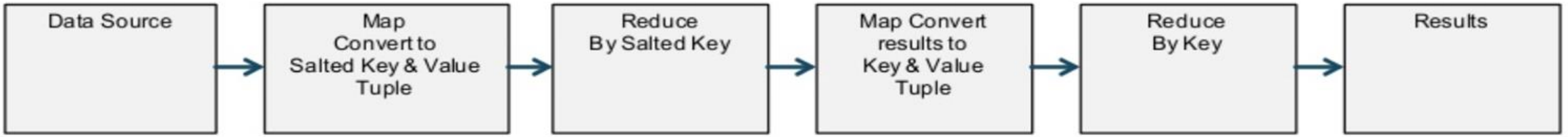


Handling skewness in processing

1. **Repartitioning**
    a. Increase the number of partitions using **repartition()** method over the RDD.
    b. When deciding the number of partitions in the output child RDD, as a rule of thumb 128 MB or 256 MB should be the size of each child partition.
    c. If you happen to use close to but less than 2000 partitions, then use at least 2000 partitions for better compression of the shuffle block
2. **Use of combiner**
    a. If there are many records of the same key, repartition does not help
    b. Combiners sharply decrease the size of shuffle data and reduces the load during aggregation.

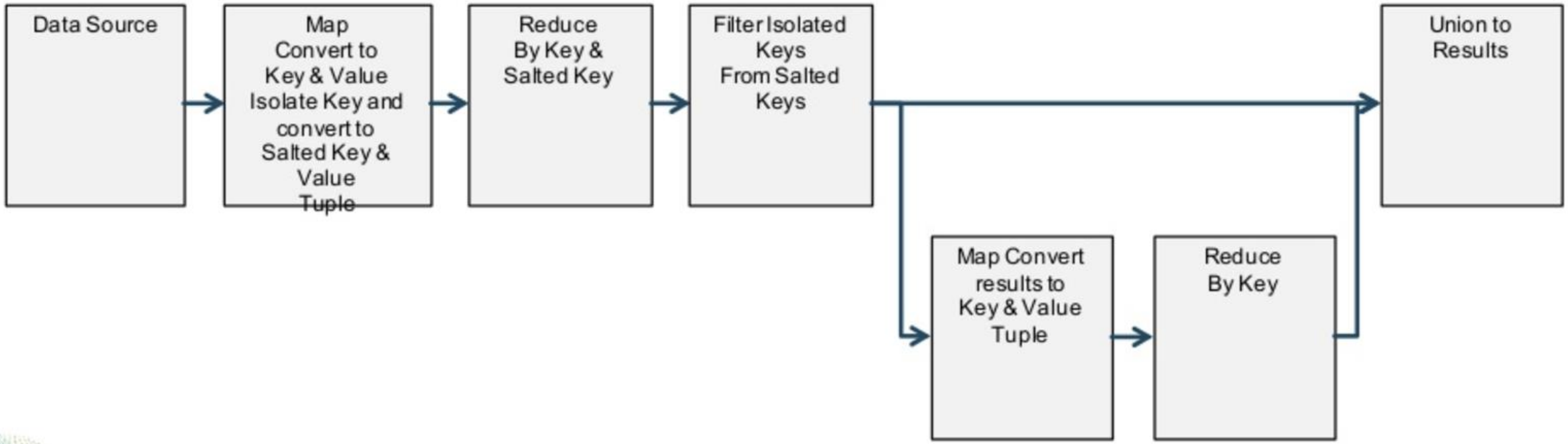
Handling skewness in aggregates: Salting

- If control over combiner and how many times it runs is limited, use **two phase aggregation** by using salting
- Salting is adding some random prefix or suffix in the original key
- Eg: if key is “Stockholm” with radomness of 10 integers, we can get keys Stockholm_0, Stockholm_1, ..., Stockholm_9. Here, _0, _1 etc are the salts.
- With salting, a single key can get spread across multiple machines.
- Do the aggregation and in the **second** phase **remove the salt** to do **final aggregation** over the actual key. The first phase of the aggregation acts like the combiner and reduces load on the second phase.



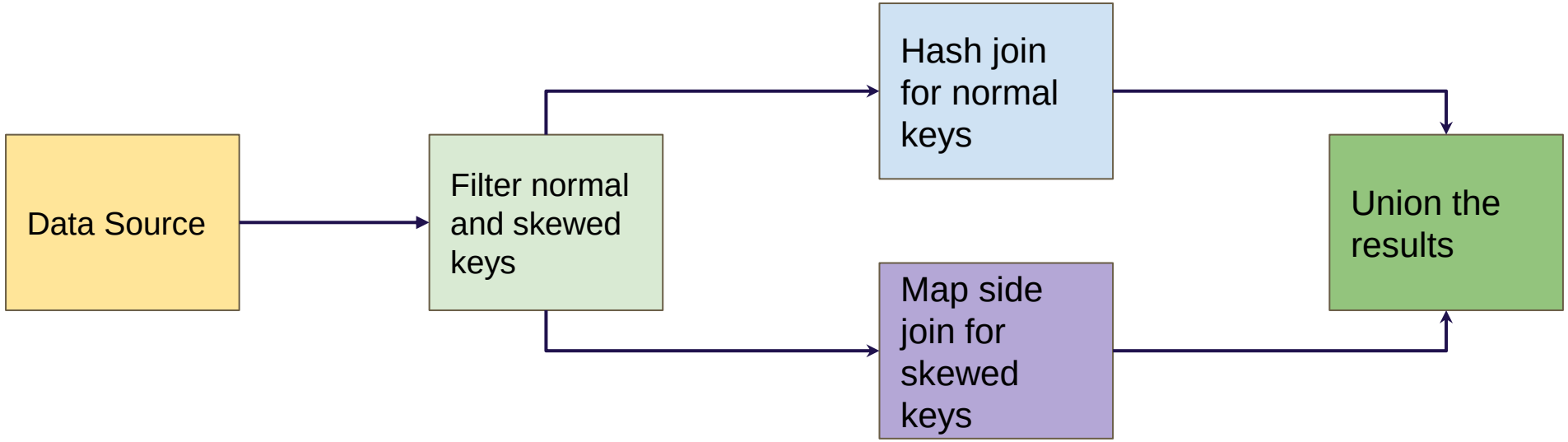
Handling skewness in aggregates: Salting

- Instead of salting all the keys, construct **two RDDs** one with normal keys and one with skewed keys
- Perform **two-phase aggregation** only for the salted keys and merge the results.



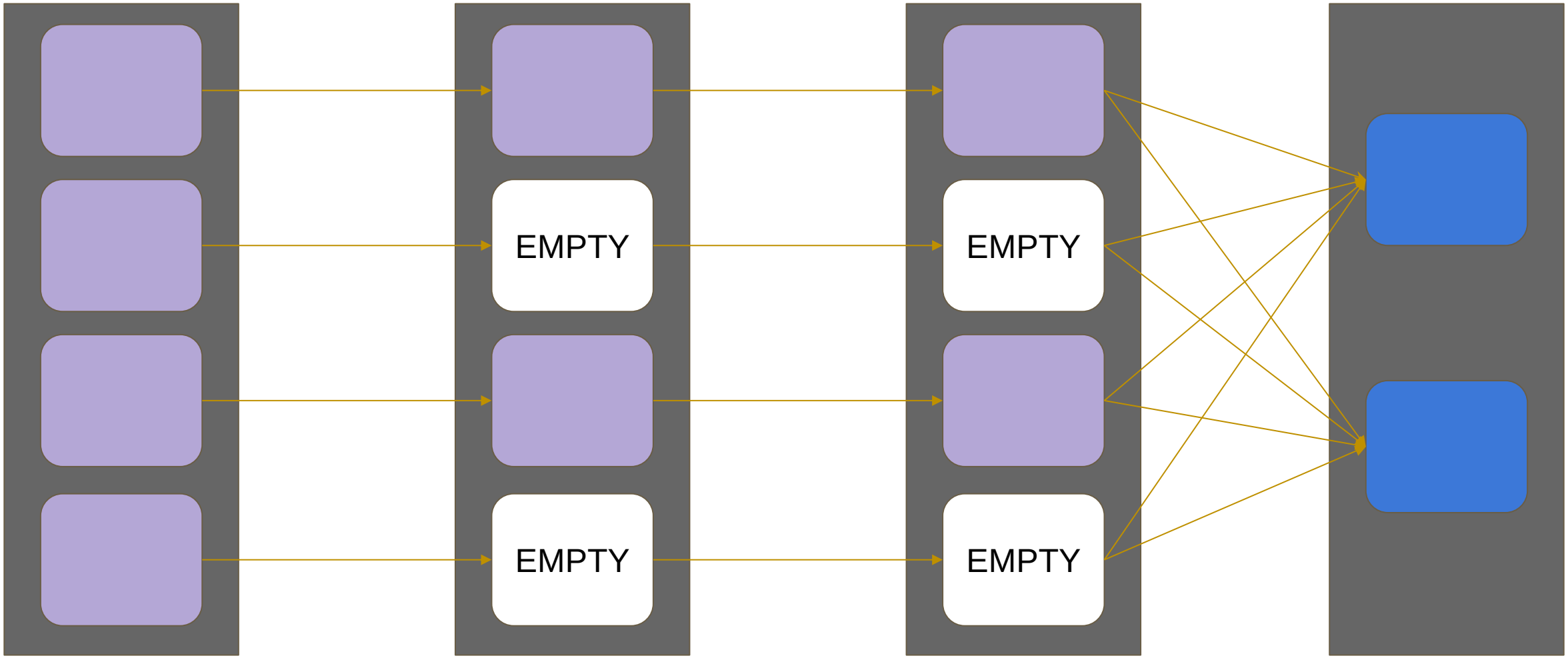
Handling skewness in joins: MapSide Join

- Construct **2 RDDs** for **each** source
  - One with normally distributed keys
  - One with skewed keys
- Apply **corresponding filter** on the **other RDD** as well
- For normal keys, use **Hash Join** and **Map Join** for skewed key



Next:
Tuning Spark Application [Miscellaneous]

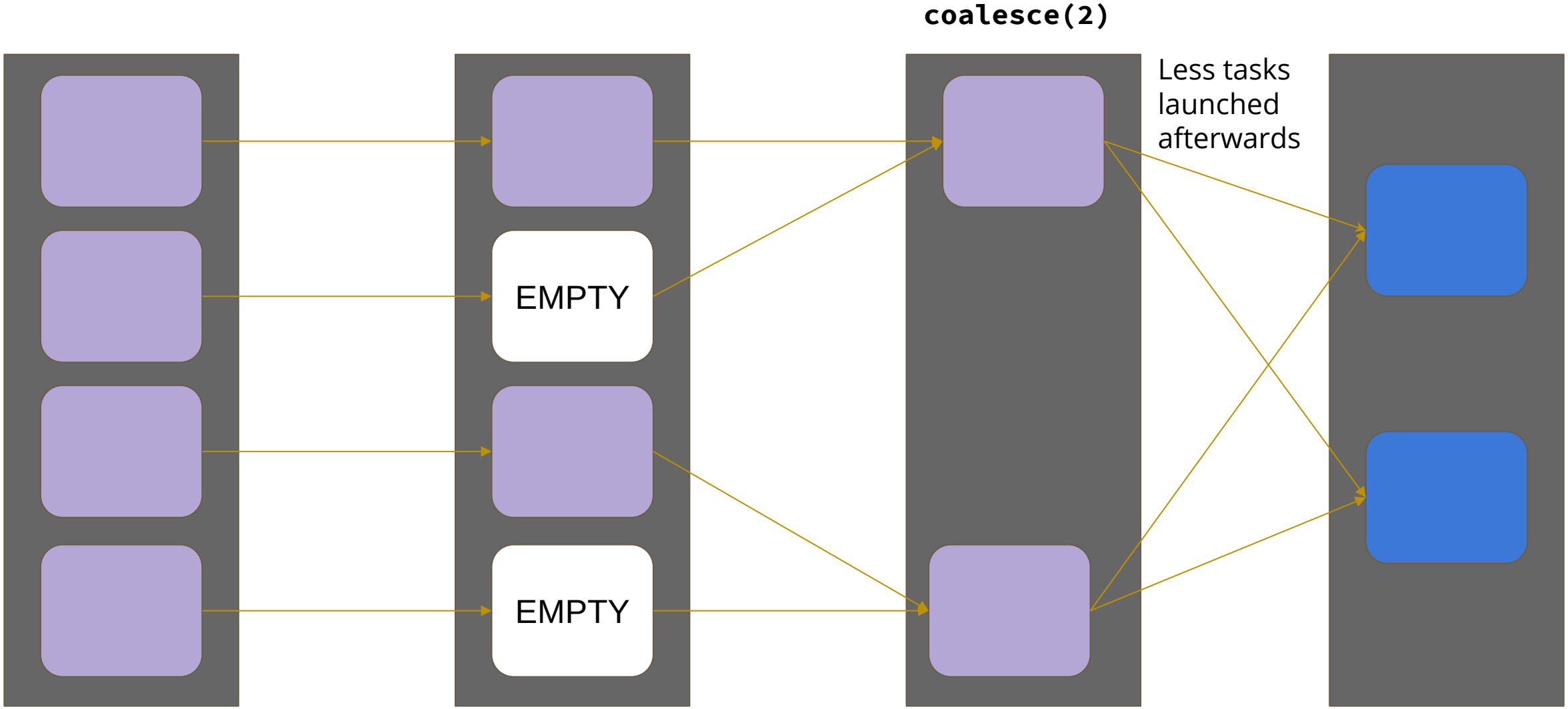
Using coalesce



Using coalesce

- When some **empty** or **light** files are produced after the final processing, use coalesce.
- **coalesce** is preferred over **repartition** if we have to reduce the count of partitions in an RDD.
- As a rule of thumb, each output file should be around 128 MB or 256 MB, if that is not the case, use coalesce in the stage of the pipeline where partitions in the output have much lesser or no data at all.
- This avoids running unnecessary tasks which do almost nothing.

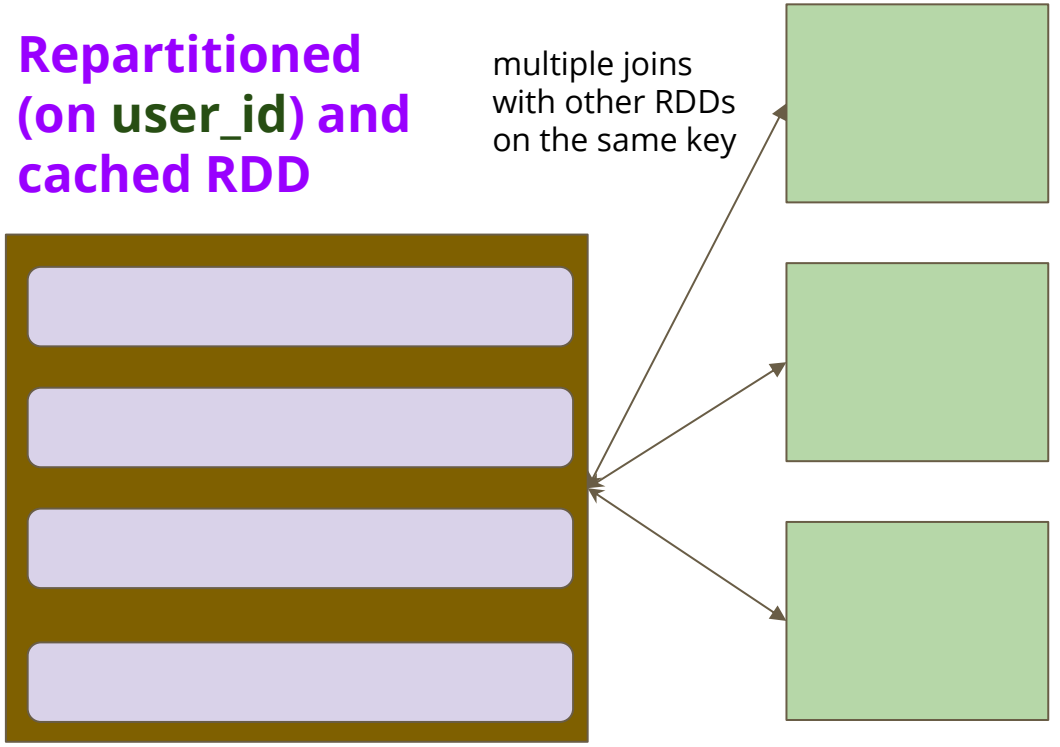
Using coalesce



Avoiding Shuffles

When joining a bigger dataset frequently with smaller datasets on the same key, look for opportunity to avoid shuffle

- Repartition the bigger RDD on the join key
- Cache the bigger RDD
- On joining with the other dataset, repartition it on the join key before joining with bigger RDD



Avoiding Shuffles

- Prefer **secondary sort** for operations like **topN** for each key
- Prefer **reduceByKey** or **combineByKey** over **groupByKey** as the former reduces the amount of data that gets shuffled

Avoid collecting larger results in memory

- Avoid using collect
  - Can lead to out of memory on driver
- Alternatives to view limited data:
  - sample
  - take

Avoid unnecessary actions

- Each action triggers one Spark job
- Think of caching the RDD if multiple actions are needed
- Metrics which can be collected via accumulators can avoid additional actions
- If we need to check if an RDD is empty or not:
  - Instead of using **`if rdd.count() > 0`**
  - Use, **`if len(rdd.take(1)) > 0`**
- count will scan entire RDD, whereas take() will just read only those partitions from where results can be retrieved.

Choosing better storage levels while caching

- **MEMORY_ONLY** is the fastest way to read the RDD
- Think of **MEMORY_ONLY_SER** or **MEMORY_AND_DISK**, if RAM is not enough
- Use the replicated storage levels like **MEMORY_ONLY_2** if the RDD is critical for downstream RDDs and RAM is sufficient
- If multiple actions can be avoided, caching the RDD should also be considered not to be used.

Testing on local than on cluster

- Usually cluster is shared across multiple applications
- Test the logic on the local using **`pyspark --master local[1]`**
- Encourage writing unit tests while developing the application