An introduction

Pierre Lienhart - 09/12/2020

What is Apache Spark ?

Apache Spark is an:

- open-source
- in-memory
- general-purpose
- distributed-computing framework

Presentation outline

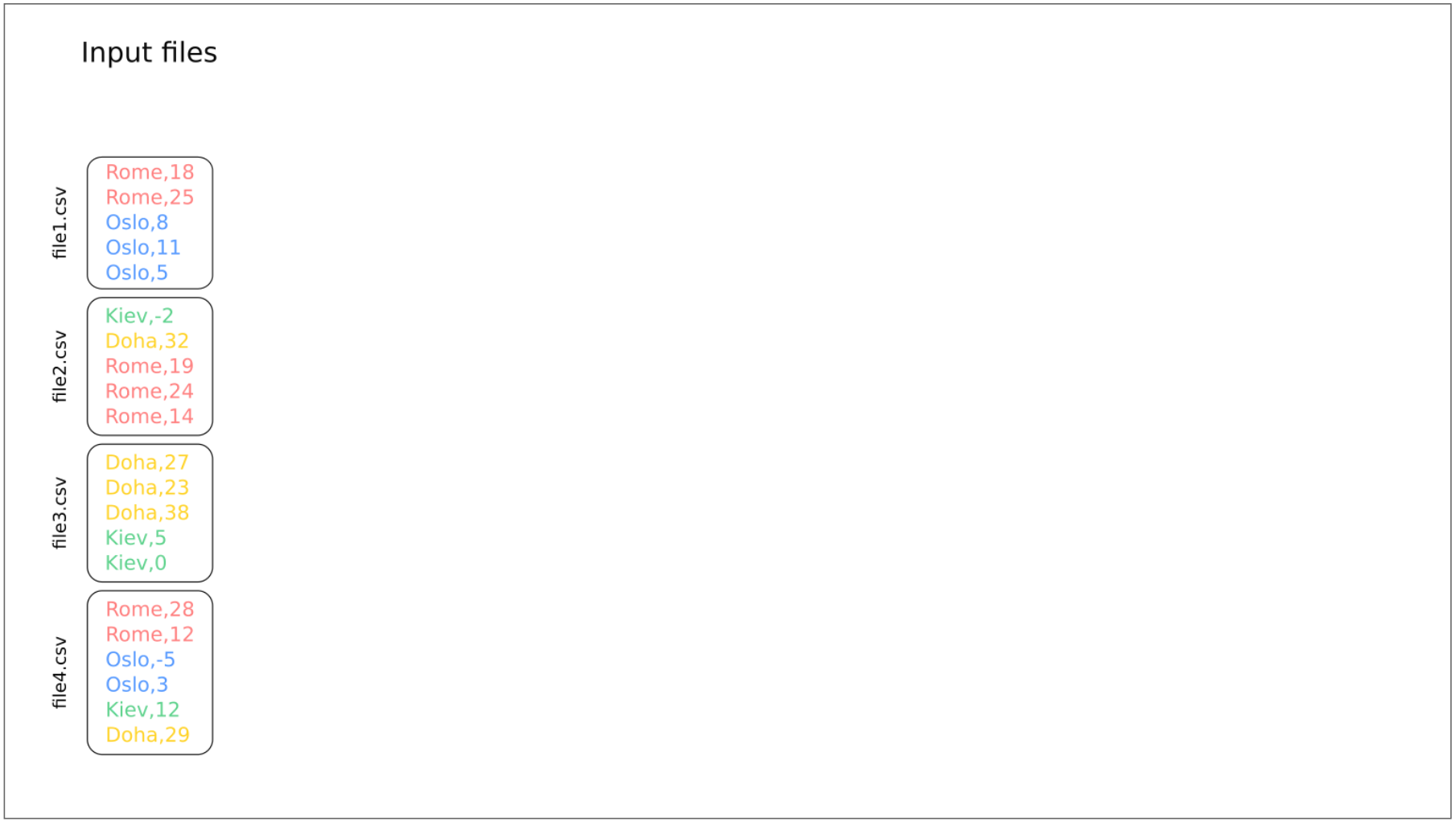
1. Introduction: From Hadoop MapReduce to Spark
2. Apache Spark: A general presentation
3. What is the architecture of a Spark application ?
4. Introducing Spark main features with SparkSQL
5. Working with Spark: useful things you should know about

1. Introduction: From Hadoop MapReduce to Spark

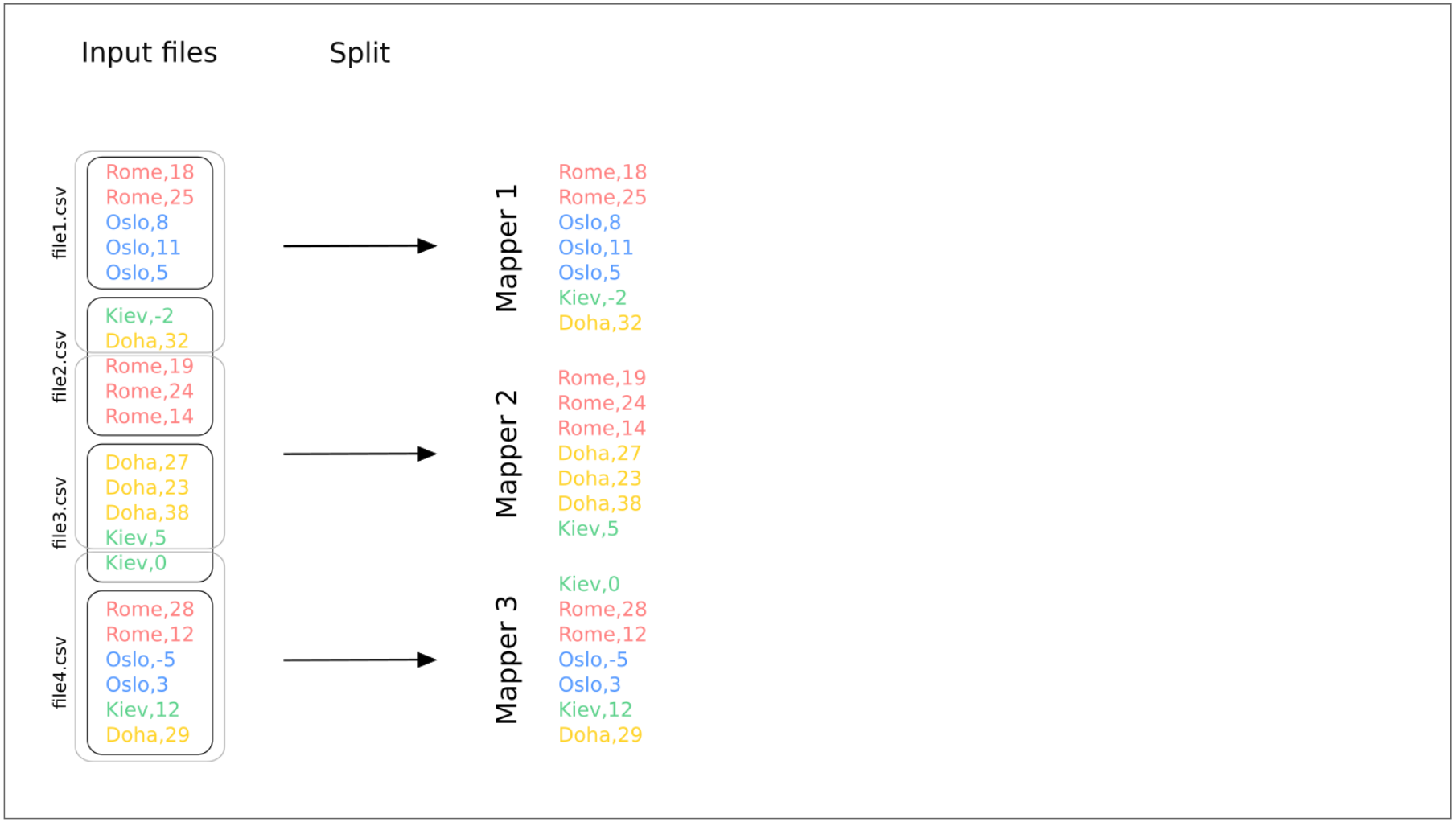
1.1 Distributing computations: MapReduce

- MapReduce is a distributed computing design pattern used to build scalable and fault-tolerant implementations of a given parallelizable algorithm or problem:
    - Scales linearly: if more data, just throw in more machines to keep the same execution time,
    - Gracefully handles partial failure.
- The model is a specialization of the split-apply-combine strategy for data analysis.
- 3 phases: map, shuffle & sort, reduce.
- 2 key components: mappers + reducers.
- Originates from *MapReduce: Simplified Data Processing on Large Clusters* (Google, 2004).

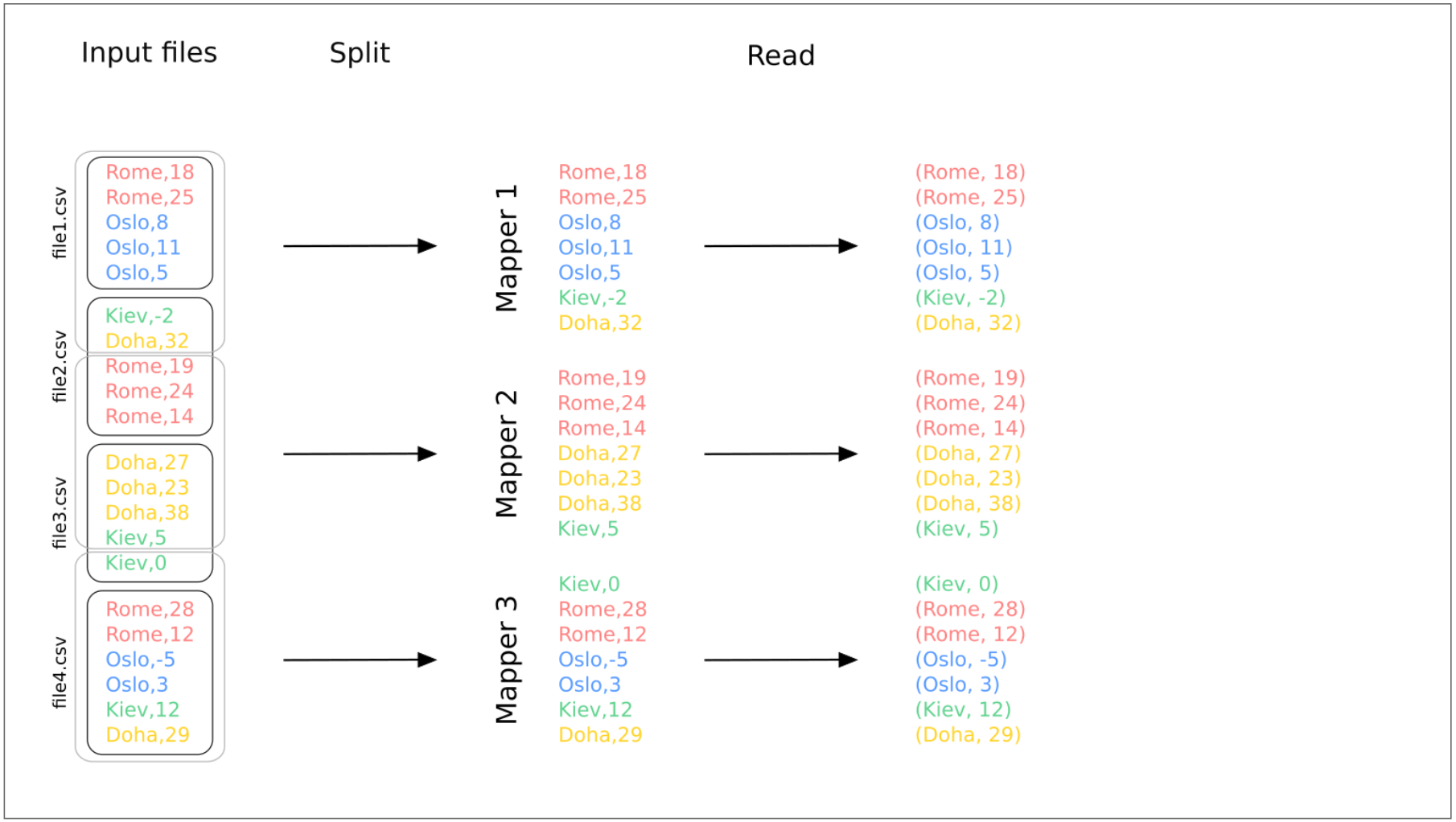
1.1 Distributing computations: MapReduce (Map phase)



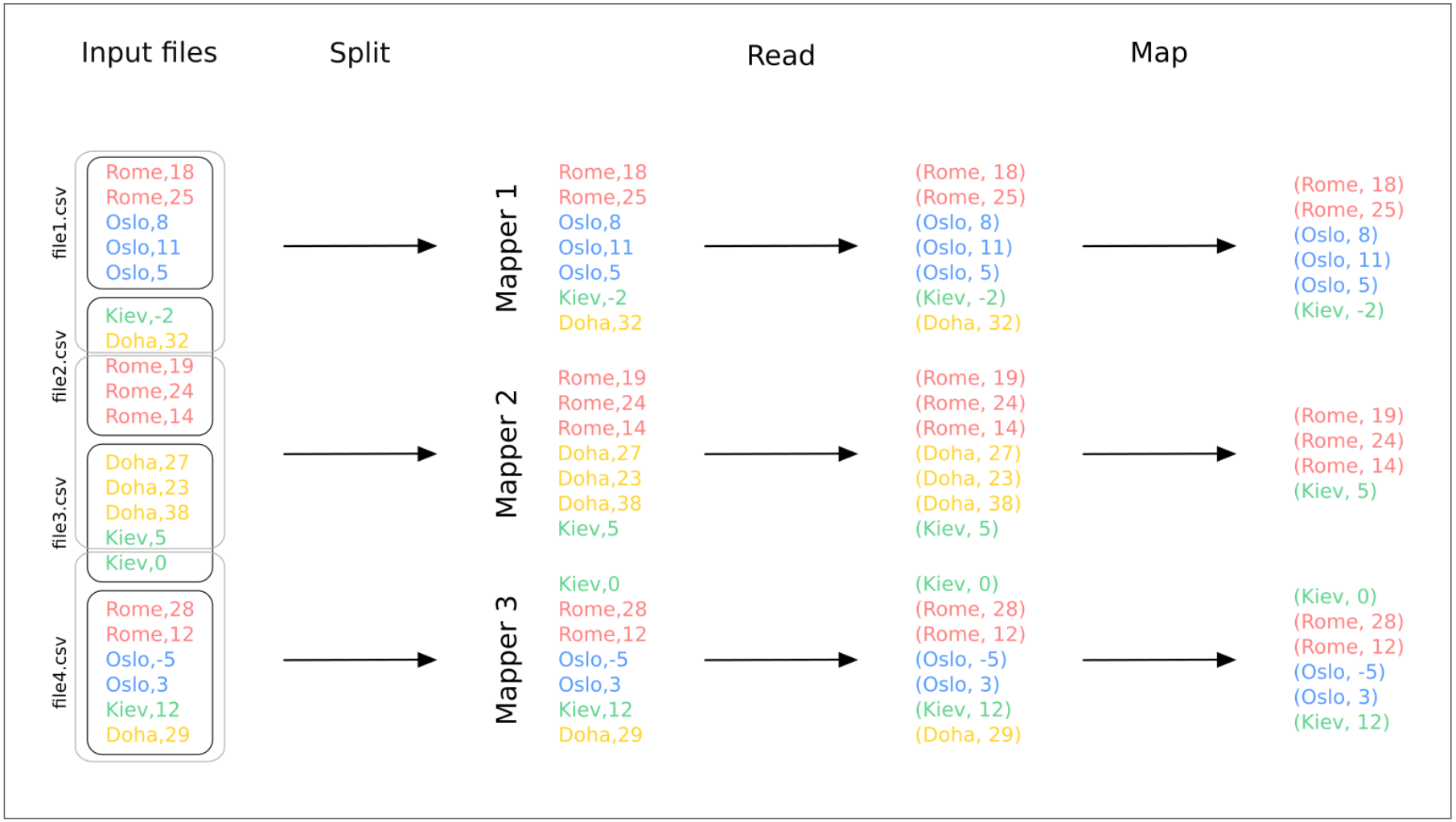
1.1 Distributing computations: MapReduce (Map phase)



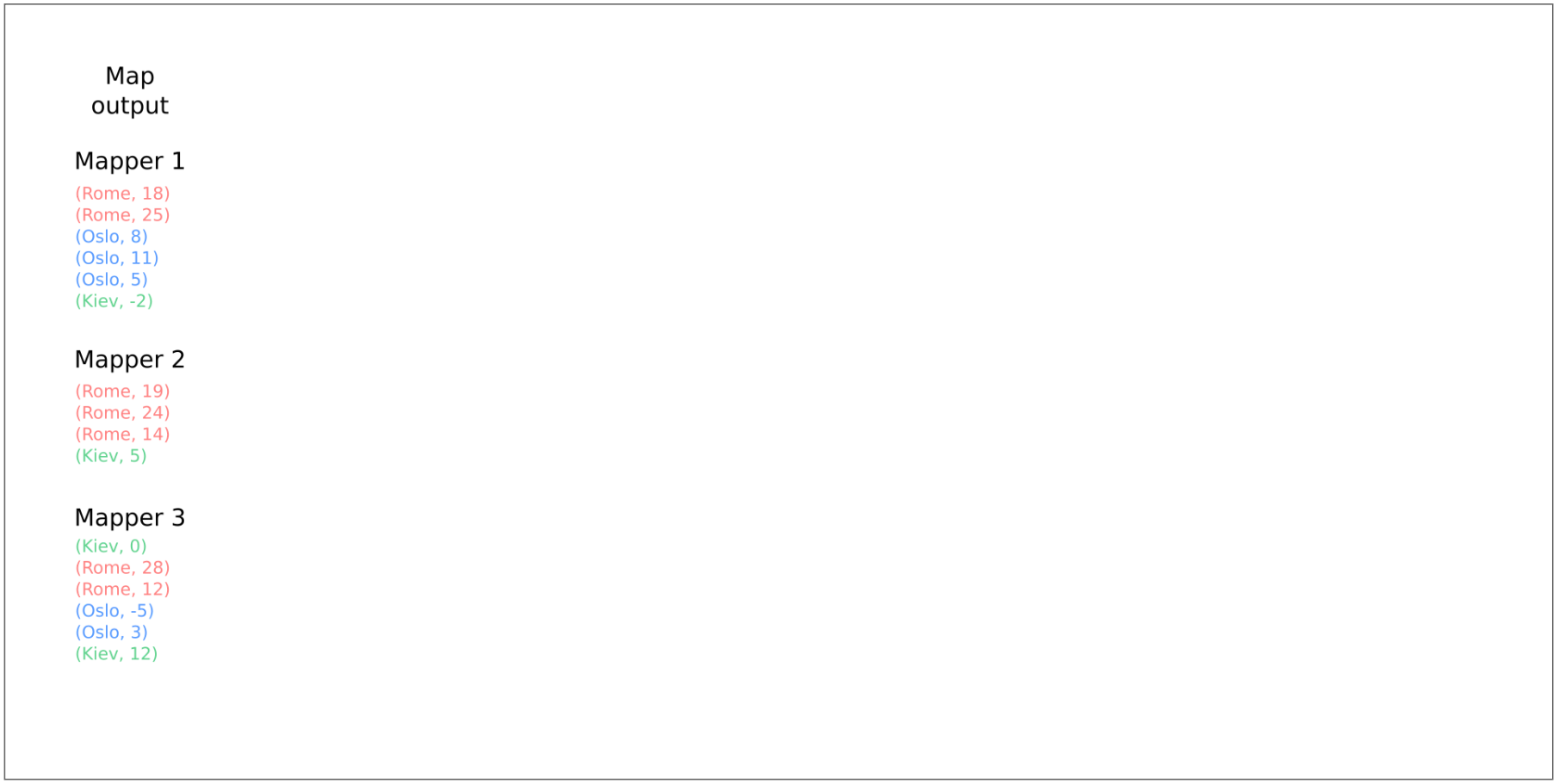
1.1 Distributing computations: MapReduce (Map phase)



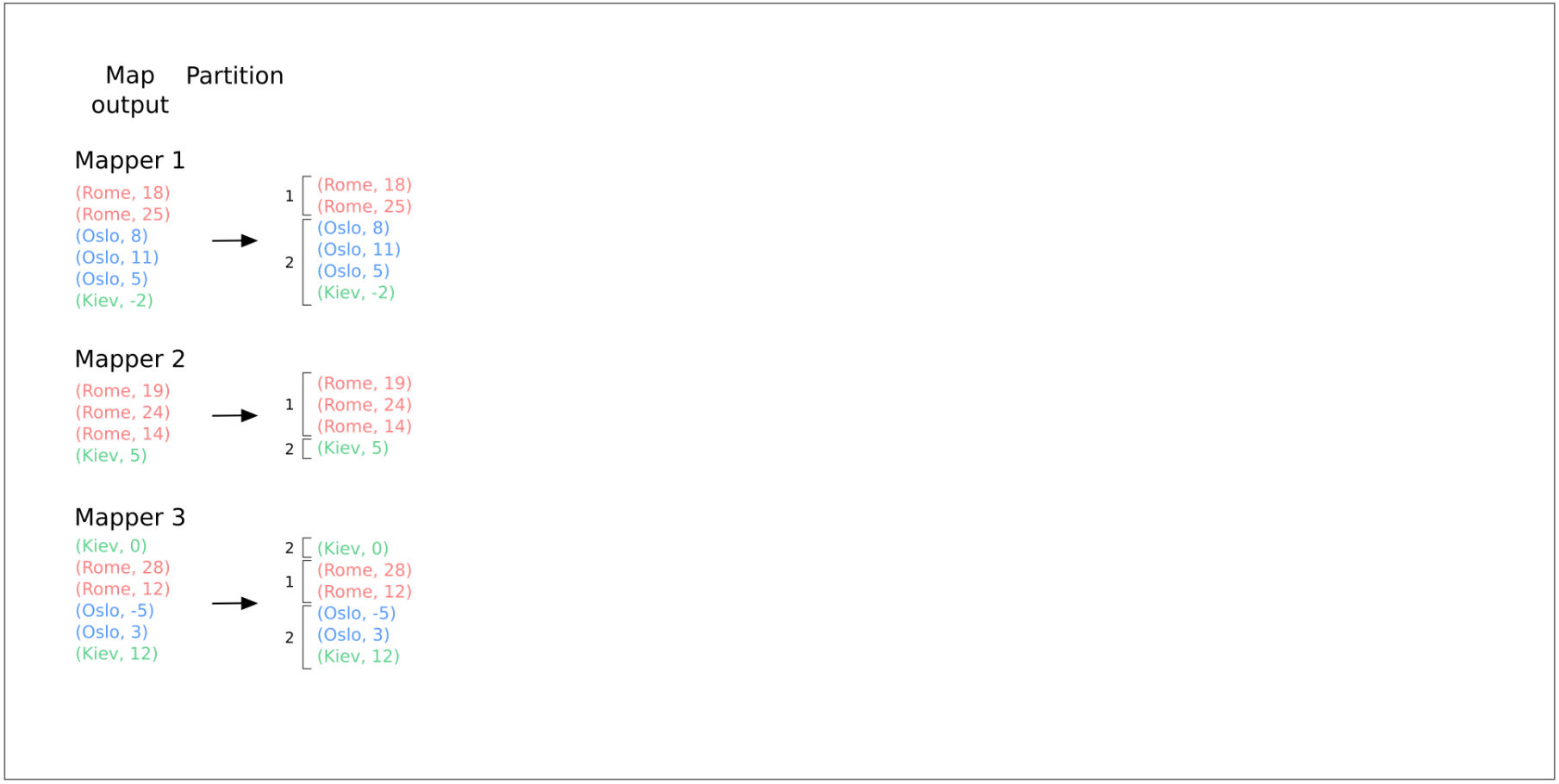
1.1 Distributing computations: MapReduce (Map phase)



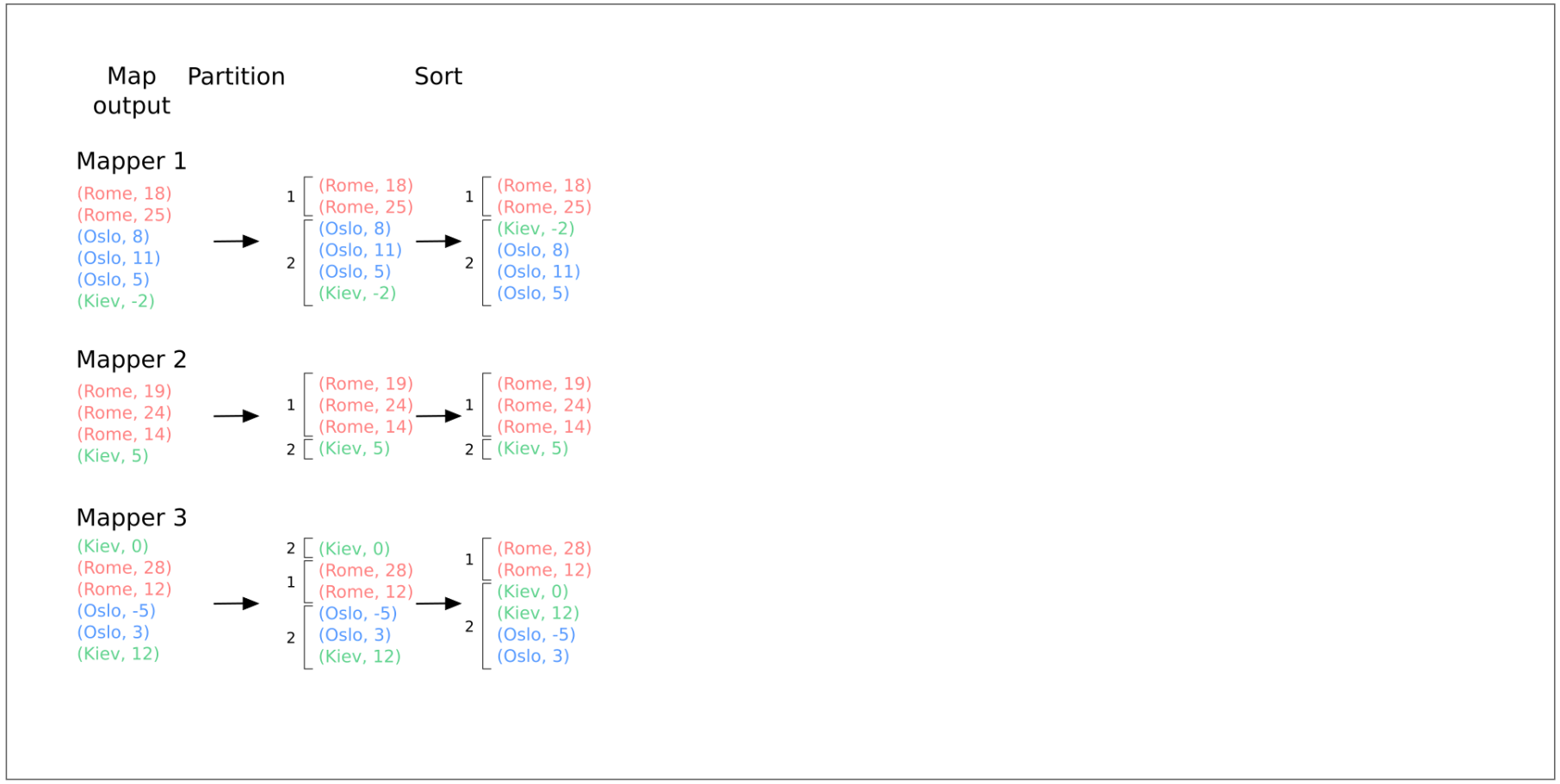
1.1 Distributing computations: MapReduce (Shuffle & sort phase)



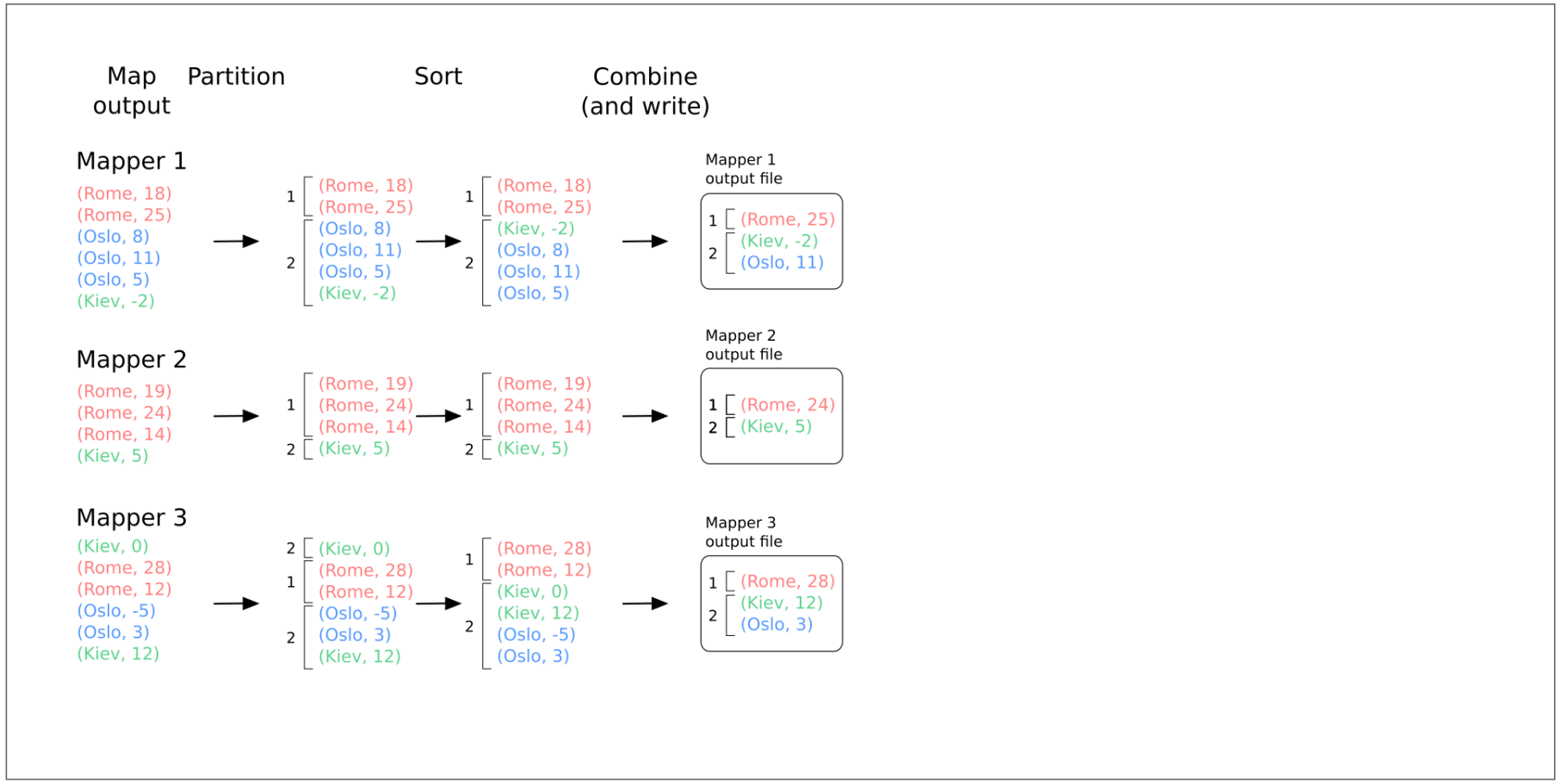
1.1 Distributing computations: MapReduce (Shuffle & sort phase)



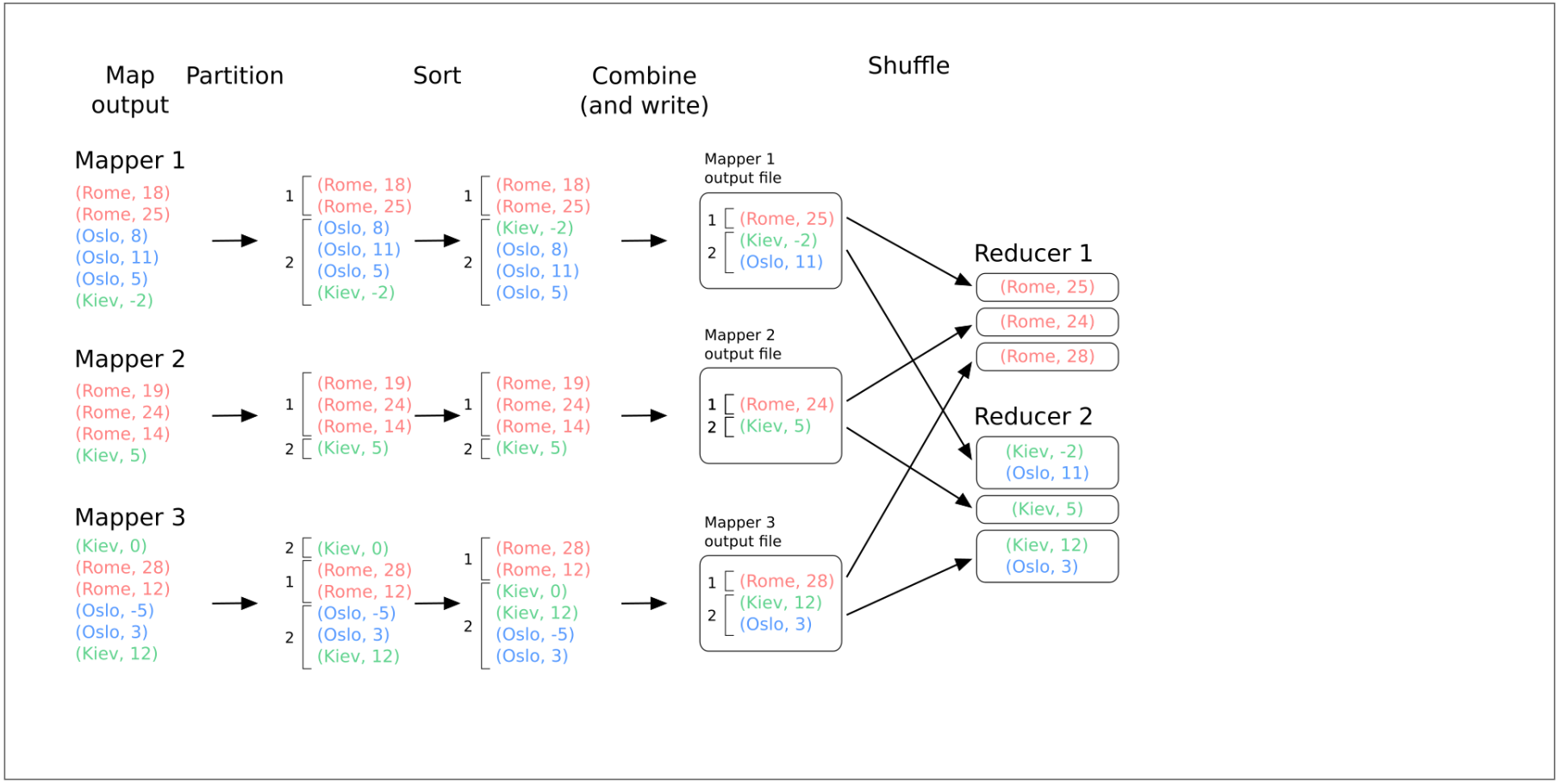
1.1 Distributing computations: MapReduce (Shuffle & sort phase)



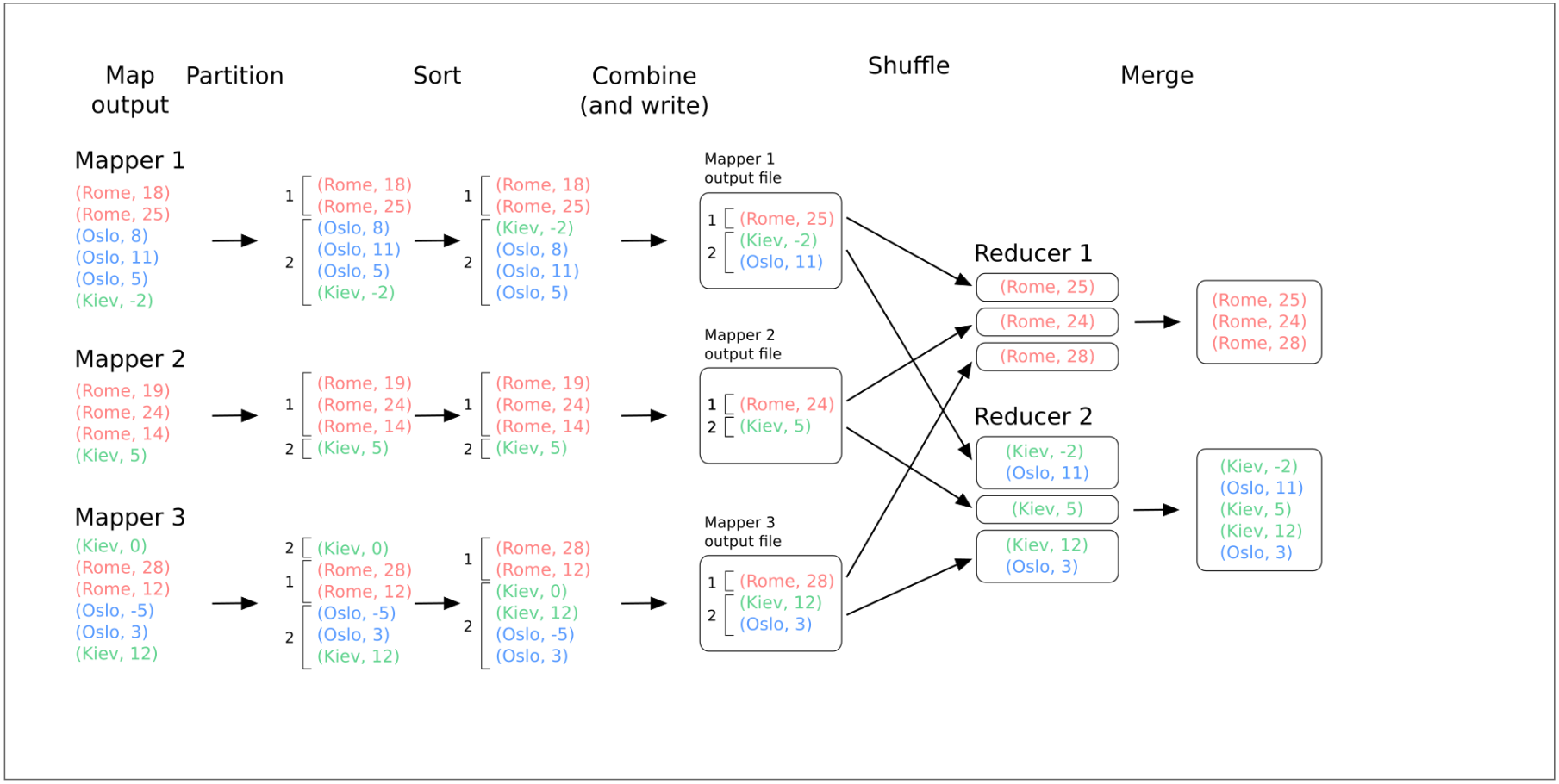
1.1 Distributing computations: MapReduce (Shuffle & sort phase)



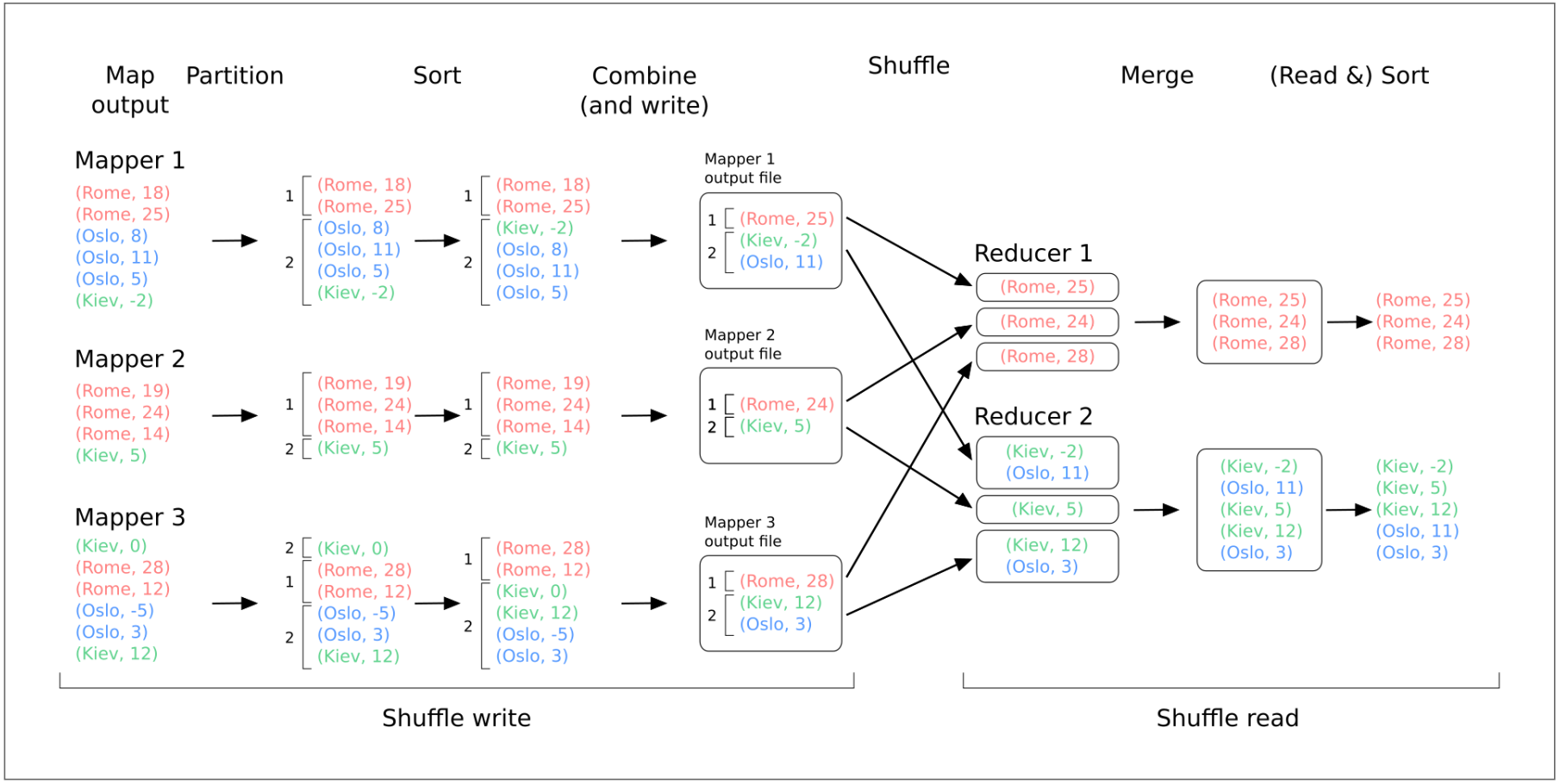
1.1 Distributing computations: MapReduce (Shuffle & sort phase)



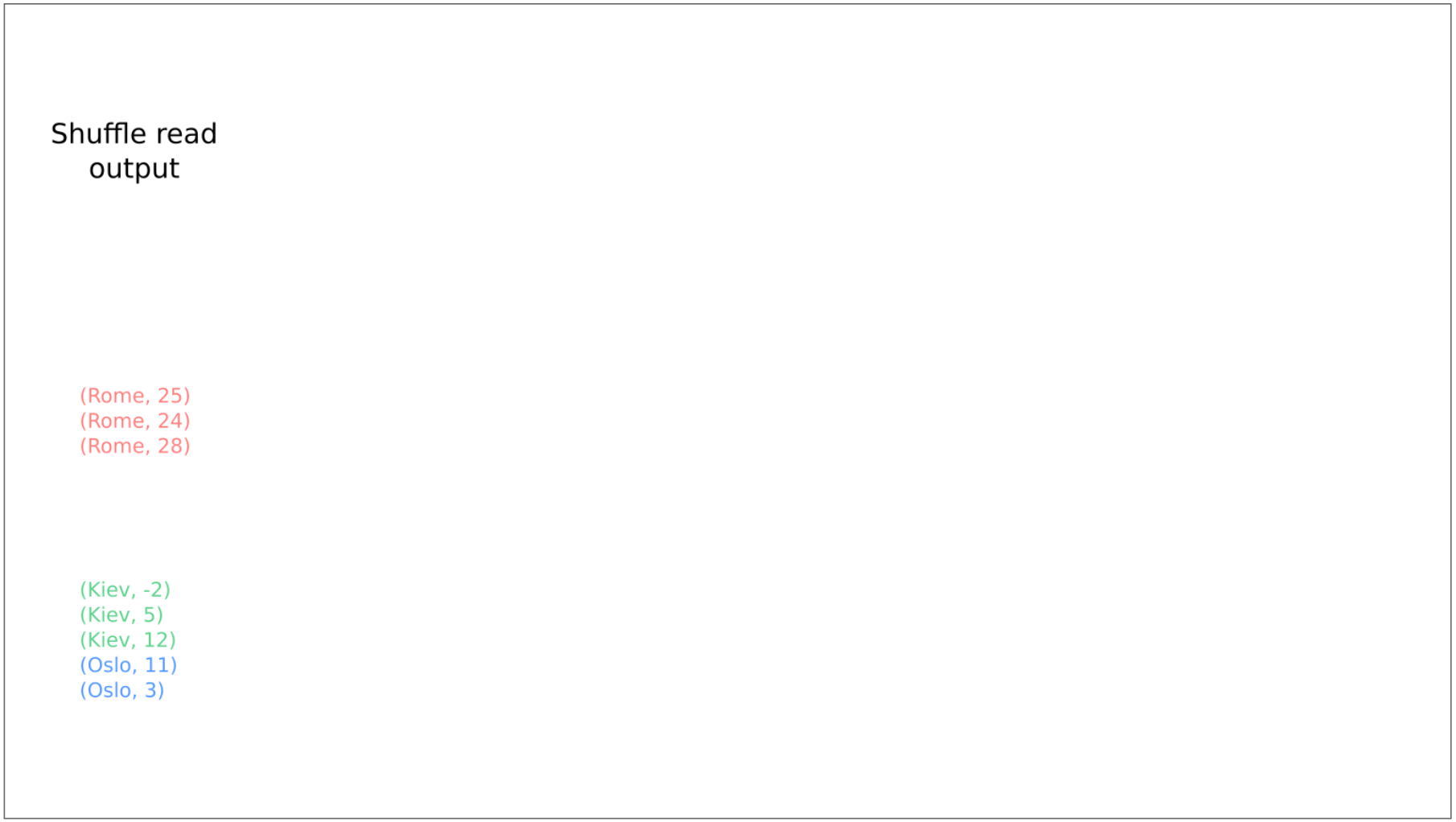
1.1 Distributing computations: MapReduce (Shuffle & sort phase)



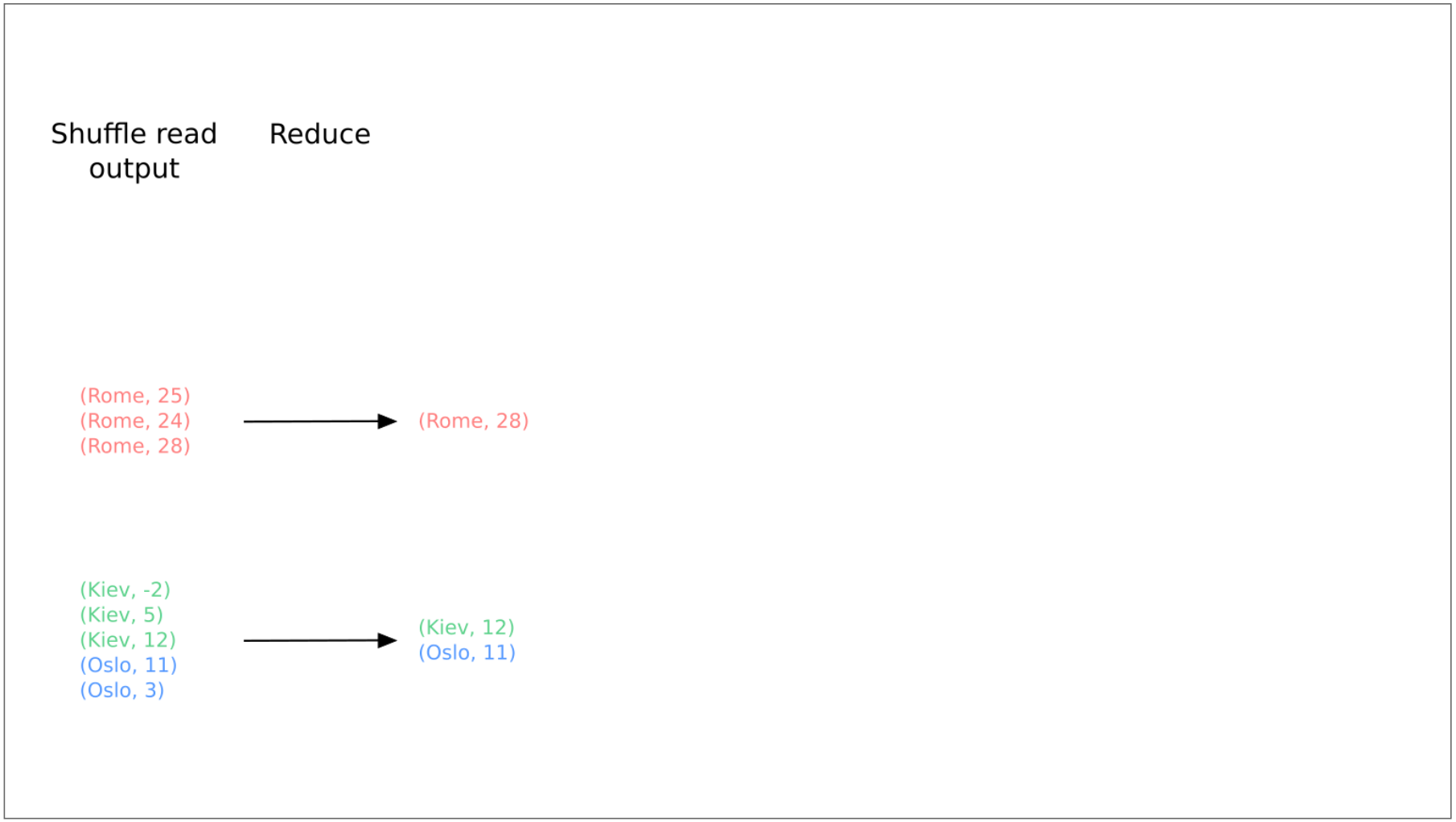
1.1 Distributing computations: MapReduce (Shuffle & sort phase)



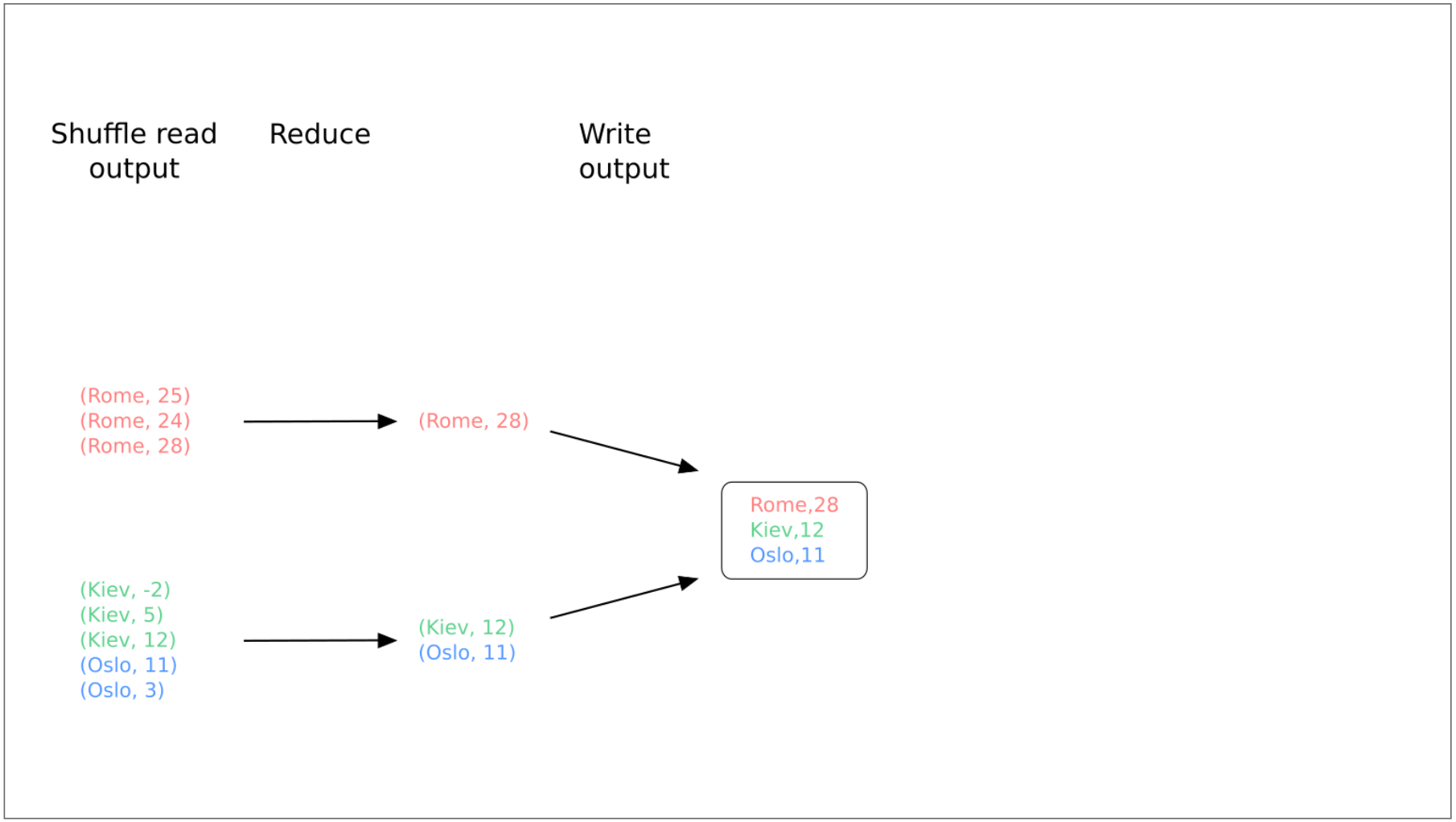
1.1 Distributing computations: MapReduce (Reduce phase)



1.1 Distributing computations: MapReduce (Reduce phase)



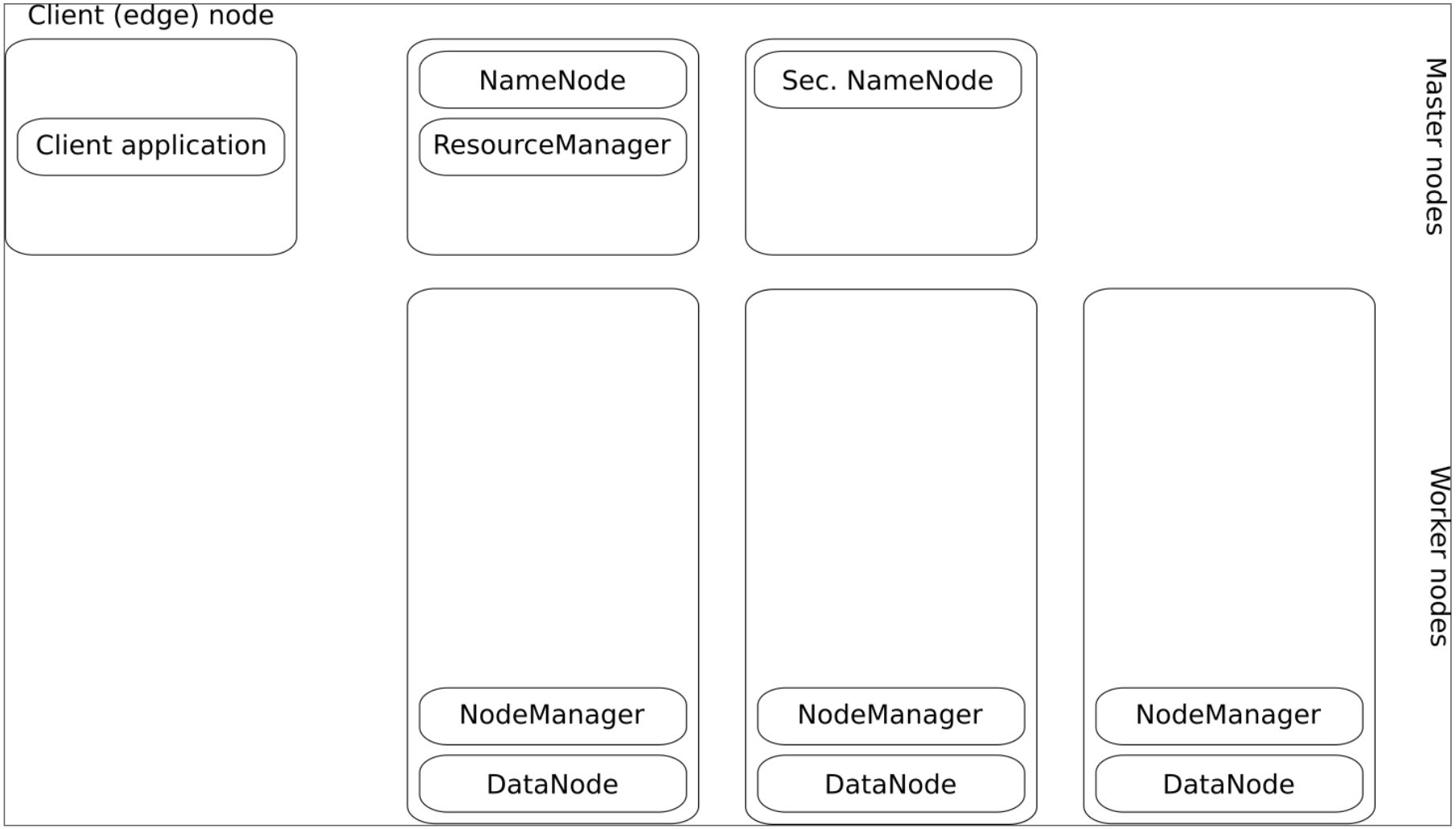
1.1 Distributing computations: MapReduce (Reduce phase)



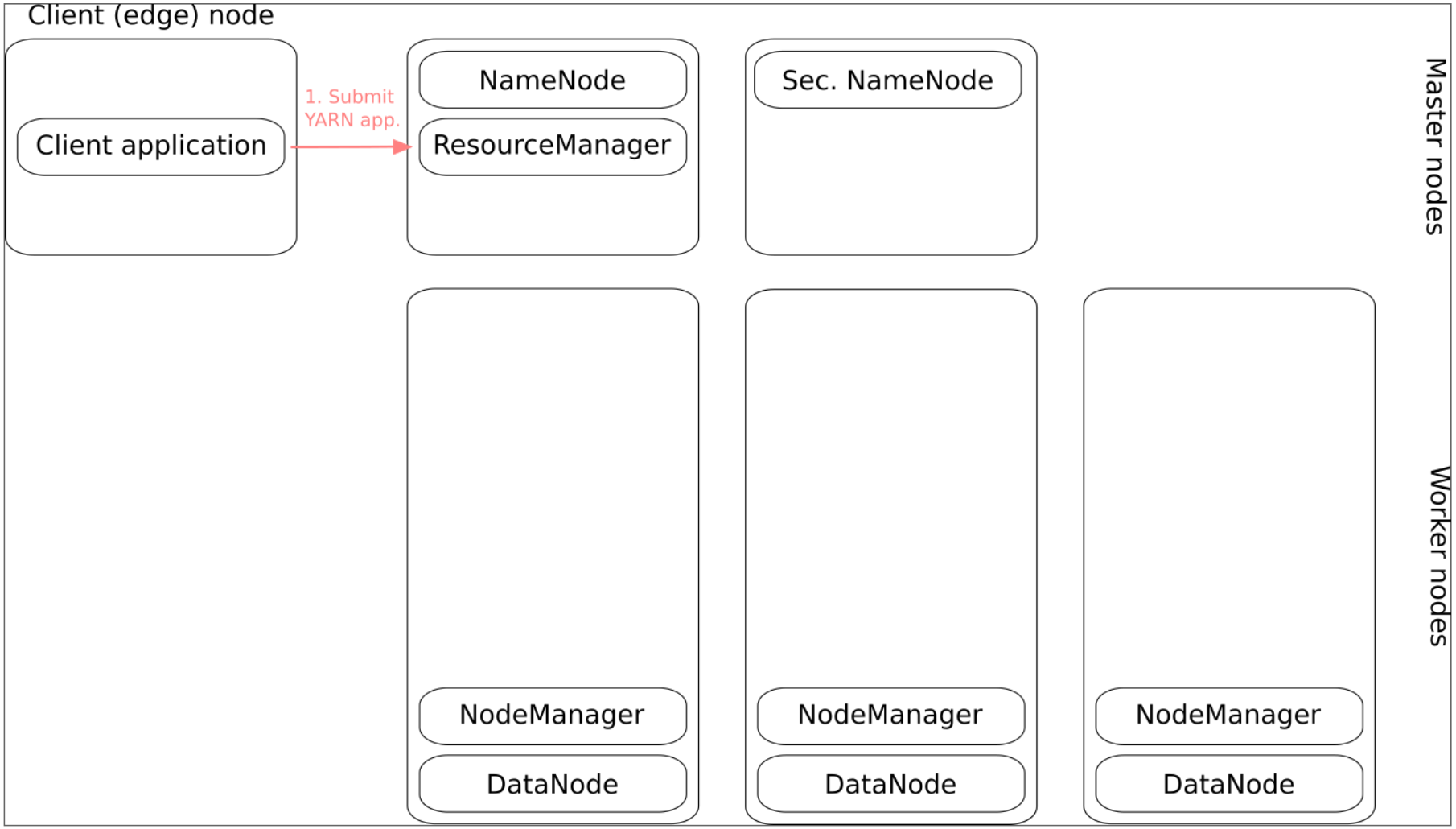
1.2 Apache Hadoop

- Hadoop's origins are rooted into two fundamental papers:
    - The Google File System (Google, 2003)
    - MapReduce: Simplified Data Processing on Large Clusters (Google, 2004)
- on which are based Hadoop's two fundamental services:
    - A distributed filesystem: Hadoop Distributed File System (HDFS)
    - A distributed computing service: Hadoop MapReduce
- Hadoop 2.0 (2013) introduced the third and last key component of Hadoop: a cluster resource manager called YARN (Yet Another Resource Negociator):
    - YARN is the product of a deep refactoring of MapReduce which split resource management (delegated to YARN) and job monitoring (delegated to dedicated applications called Application Masters)
    - Opened the door for the execution of non-MapReduce jobs.

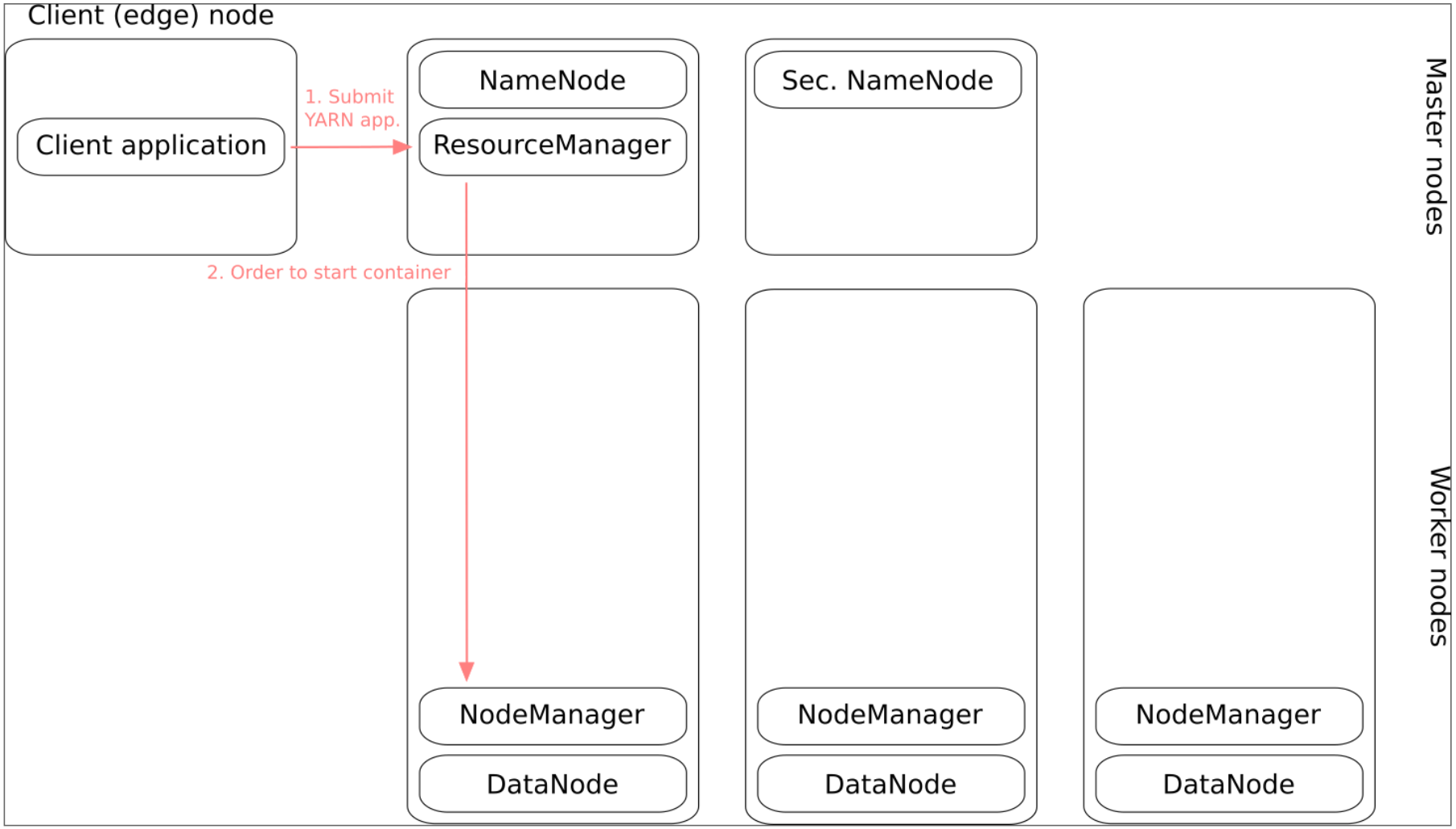
1.2 Apache Hadoop: Submitting a YARN job



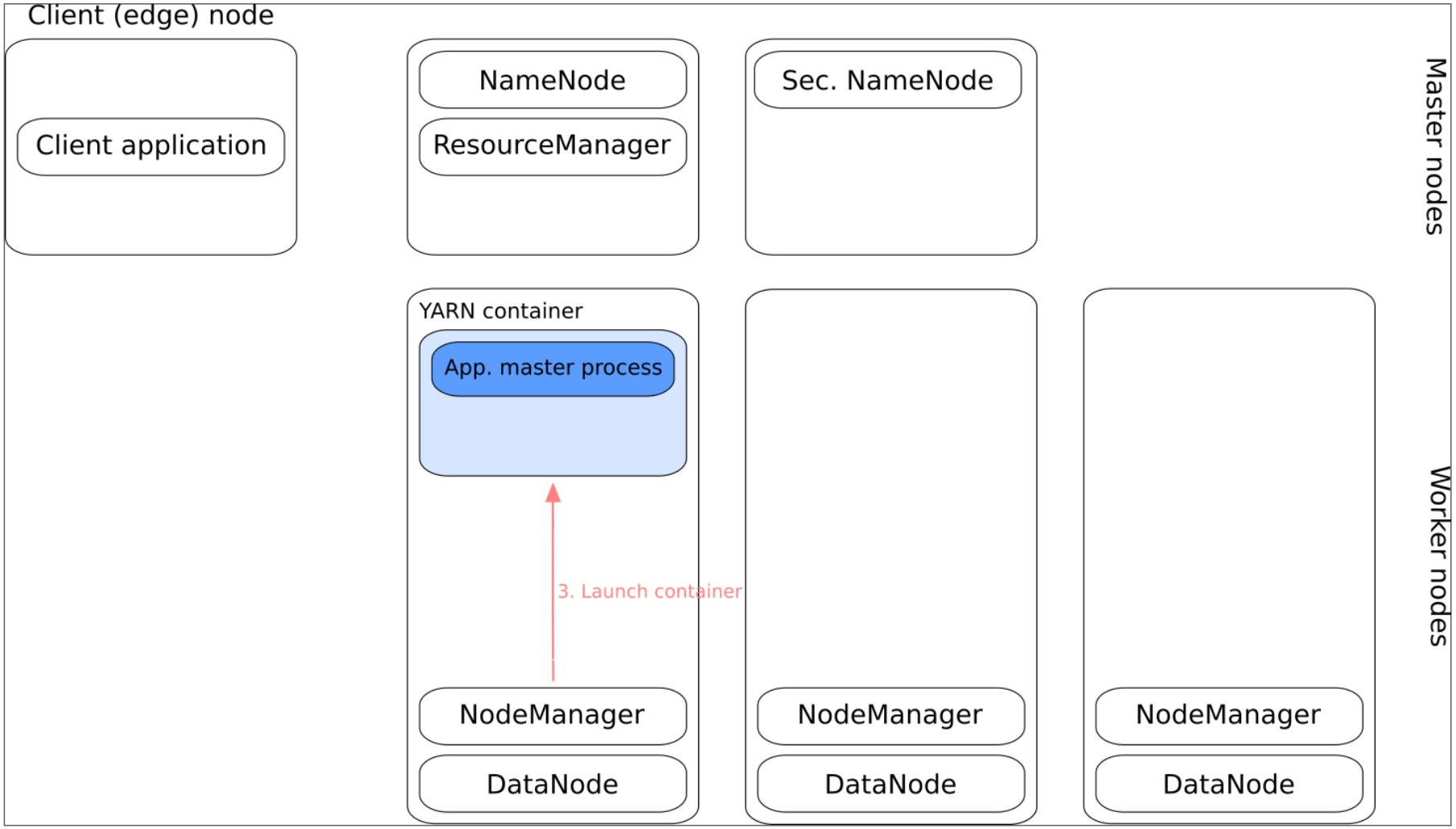
1.2 Apache Hadoop: Submitting a YARN job



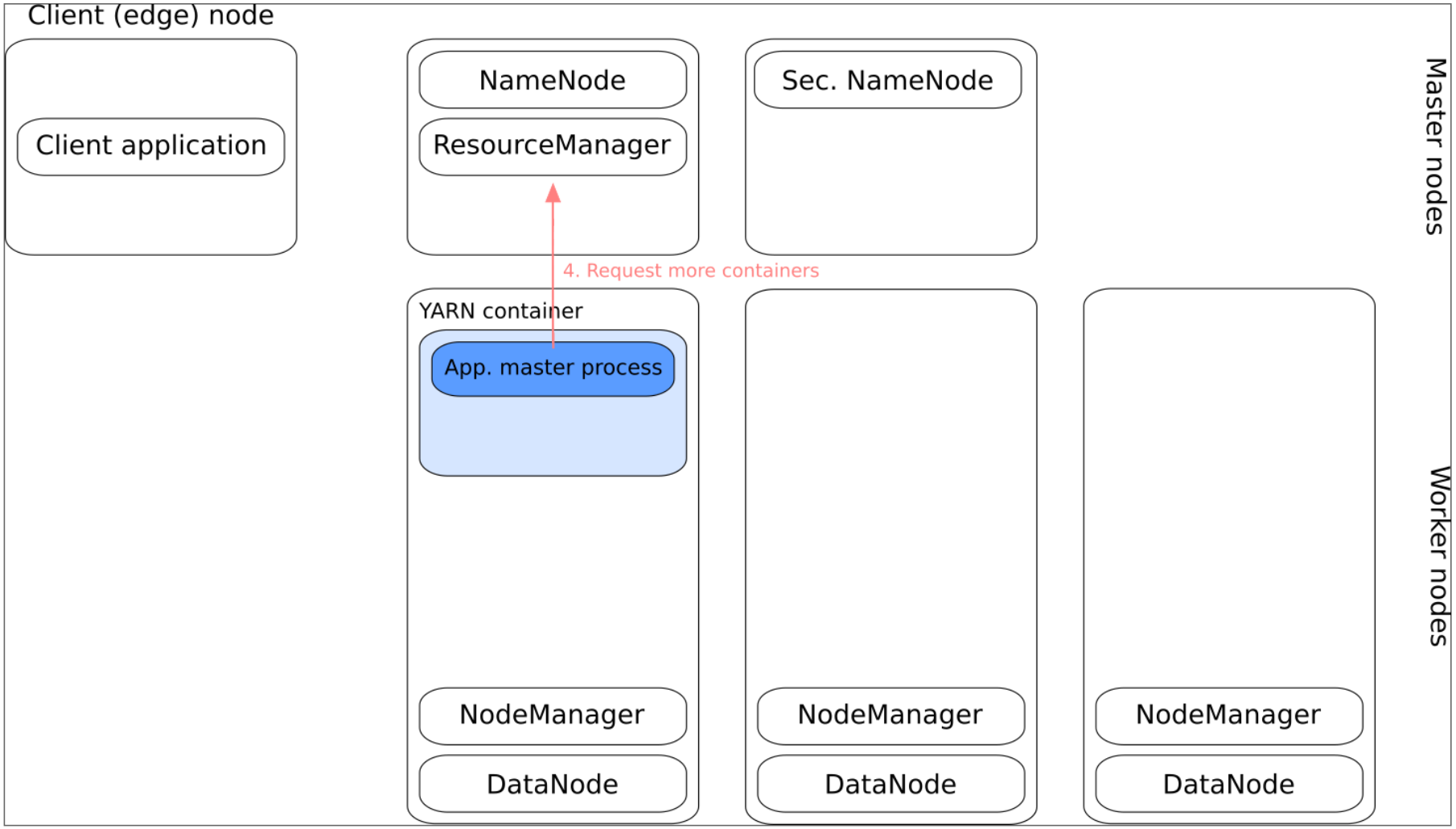
1.2 Apache Hadoop: Submitting a YARN job



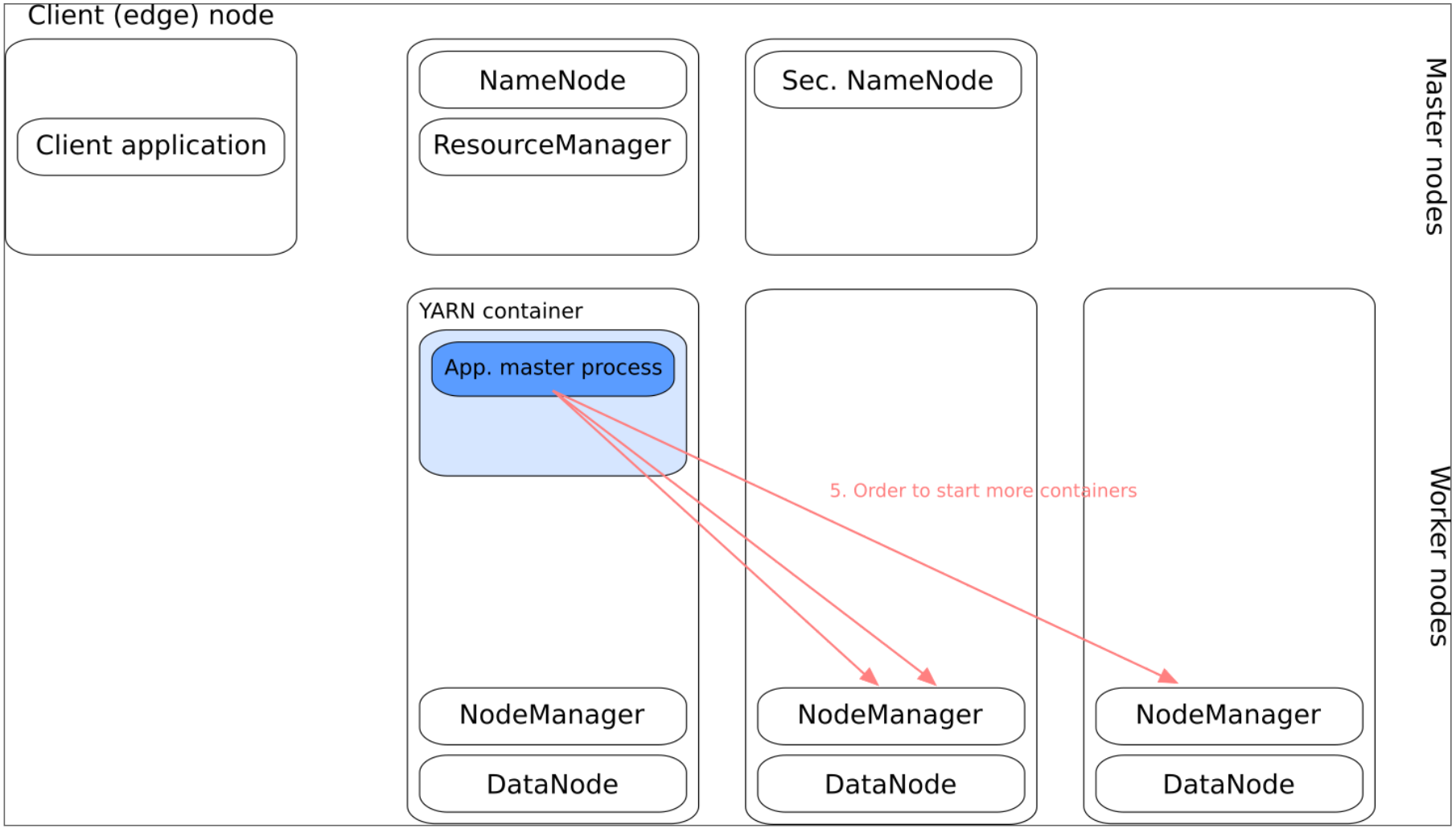
1.2 Apache Hadoop: Submitting a YARN job



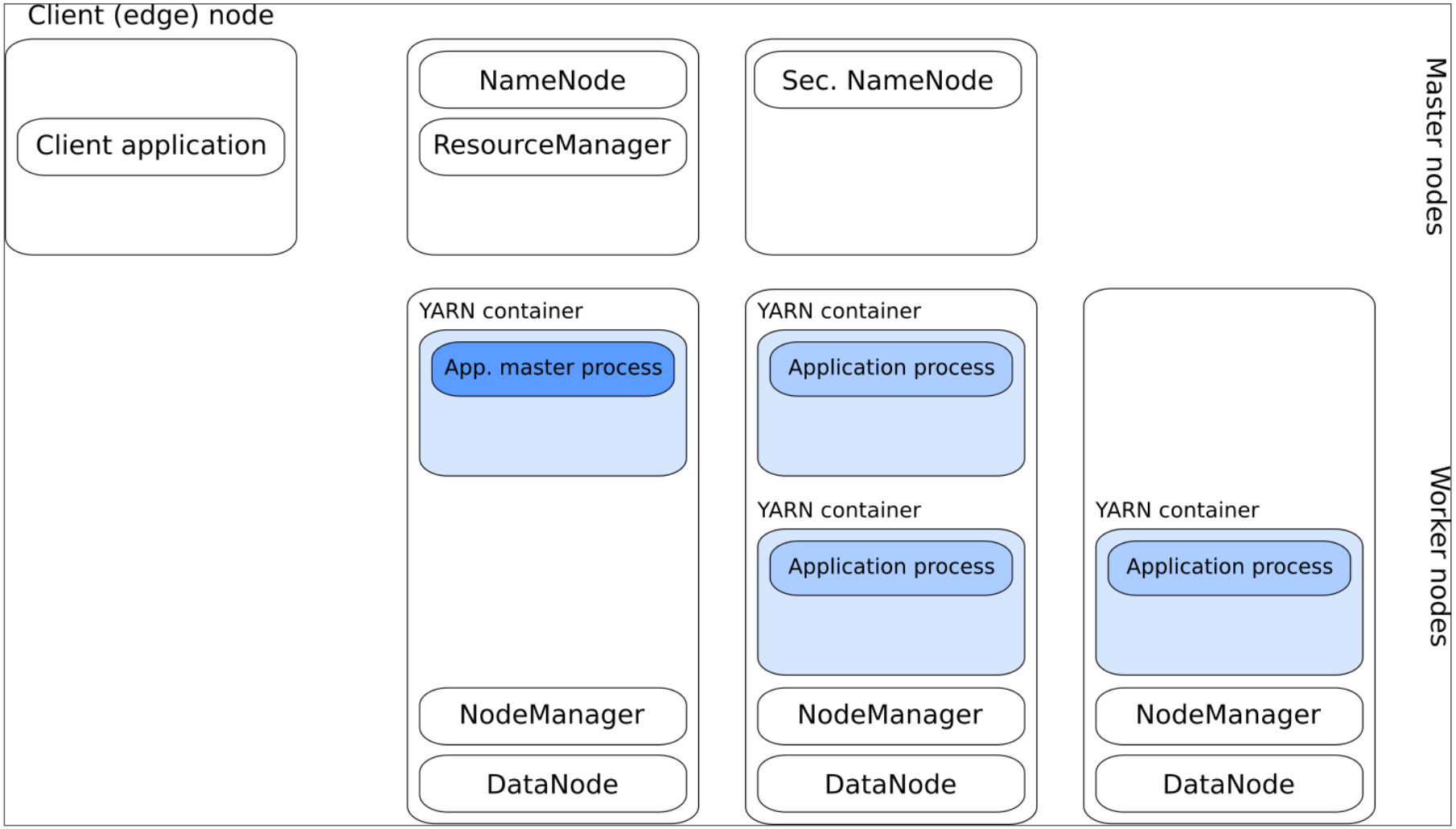
1.2 Apache Hadoop: Submitting a YARN job



1.2 Apache Hadoop: Submitting a YARN job

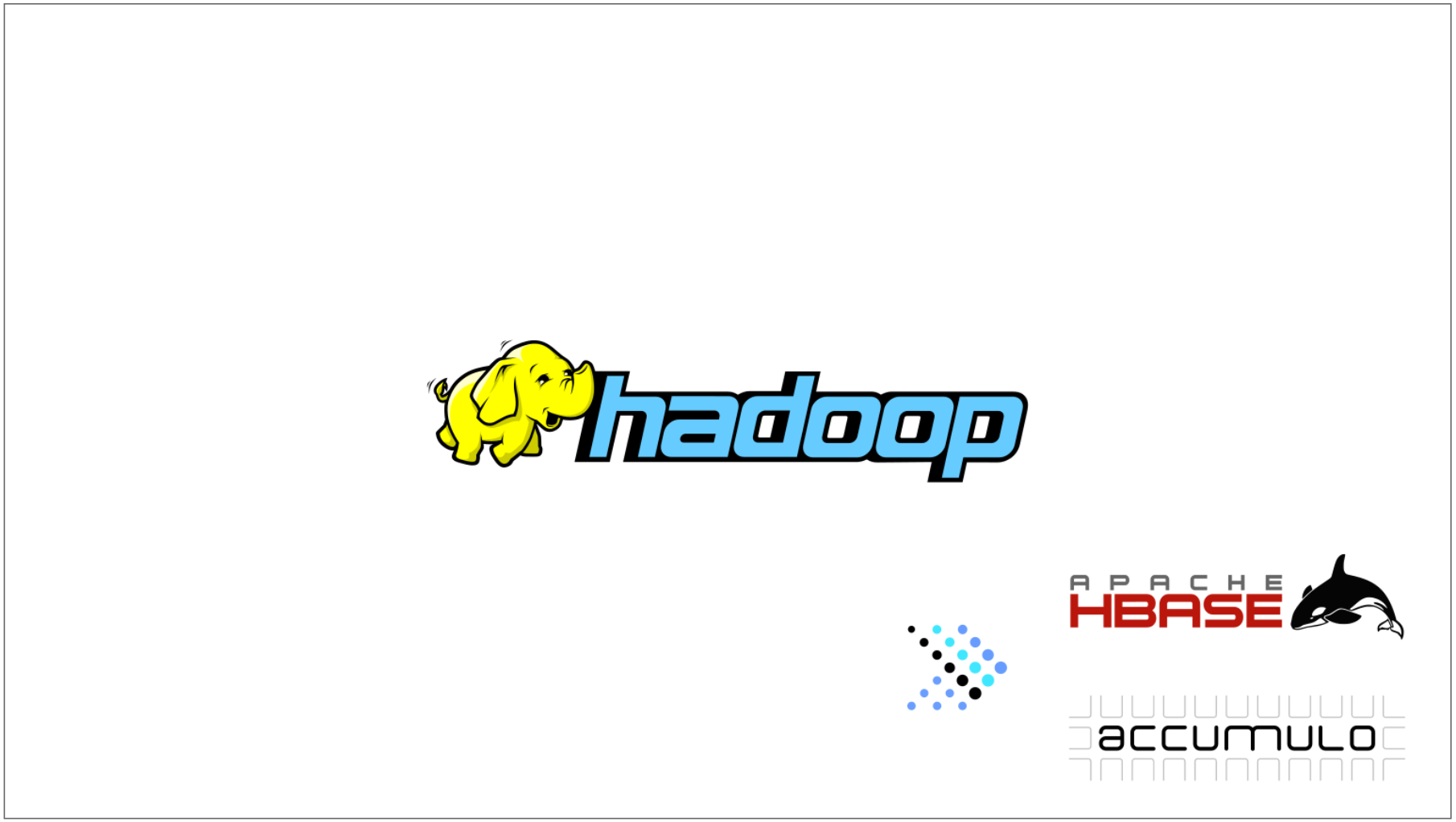


1.2 Apache Hadoop: Submitting a YARN job

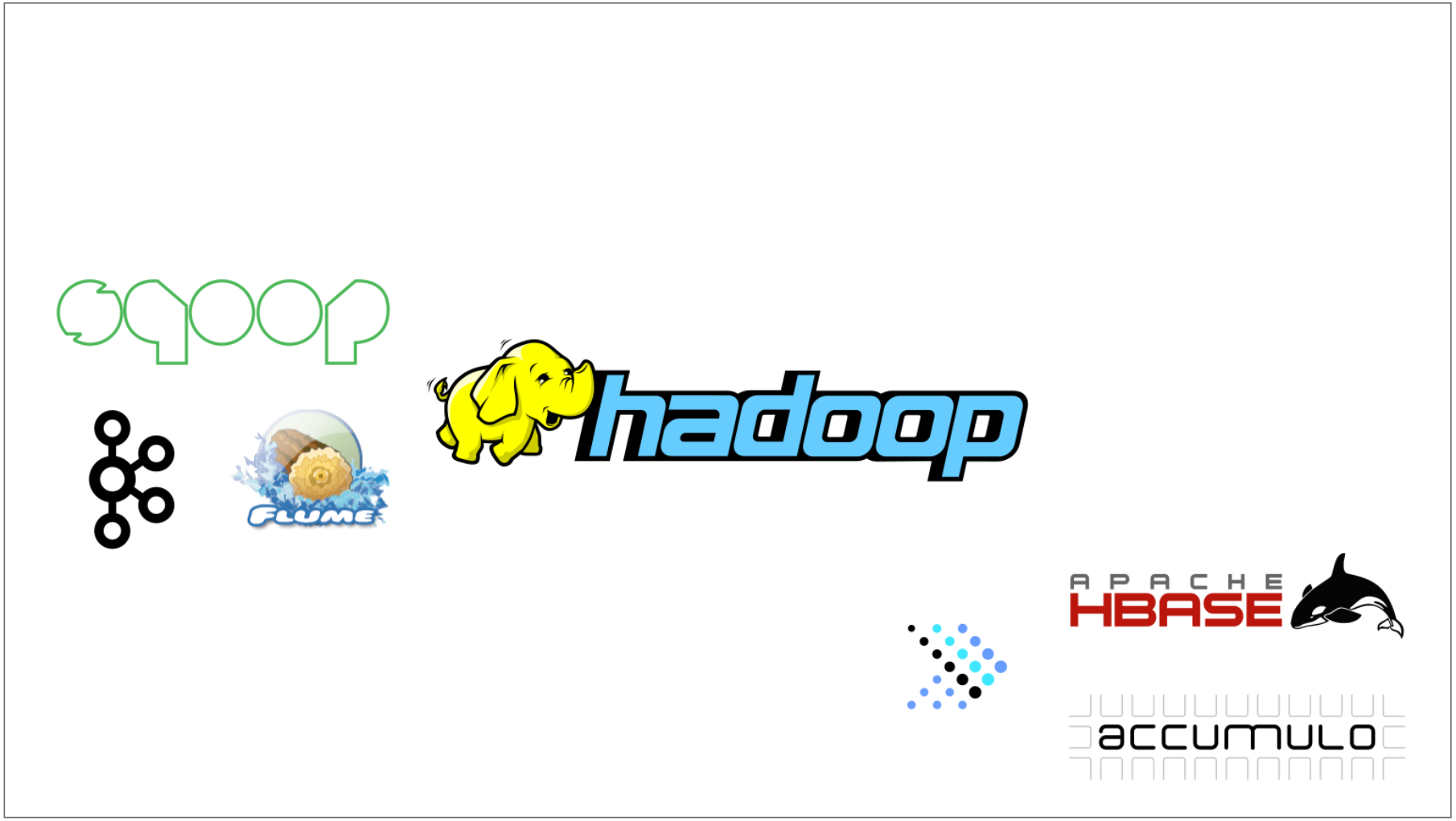


1.2 Apache Hadoop: The Hadoop ecosystem

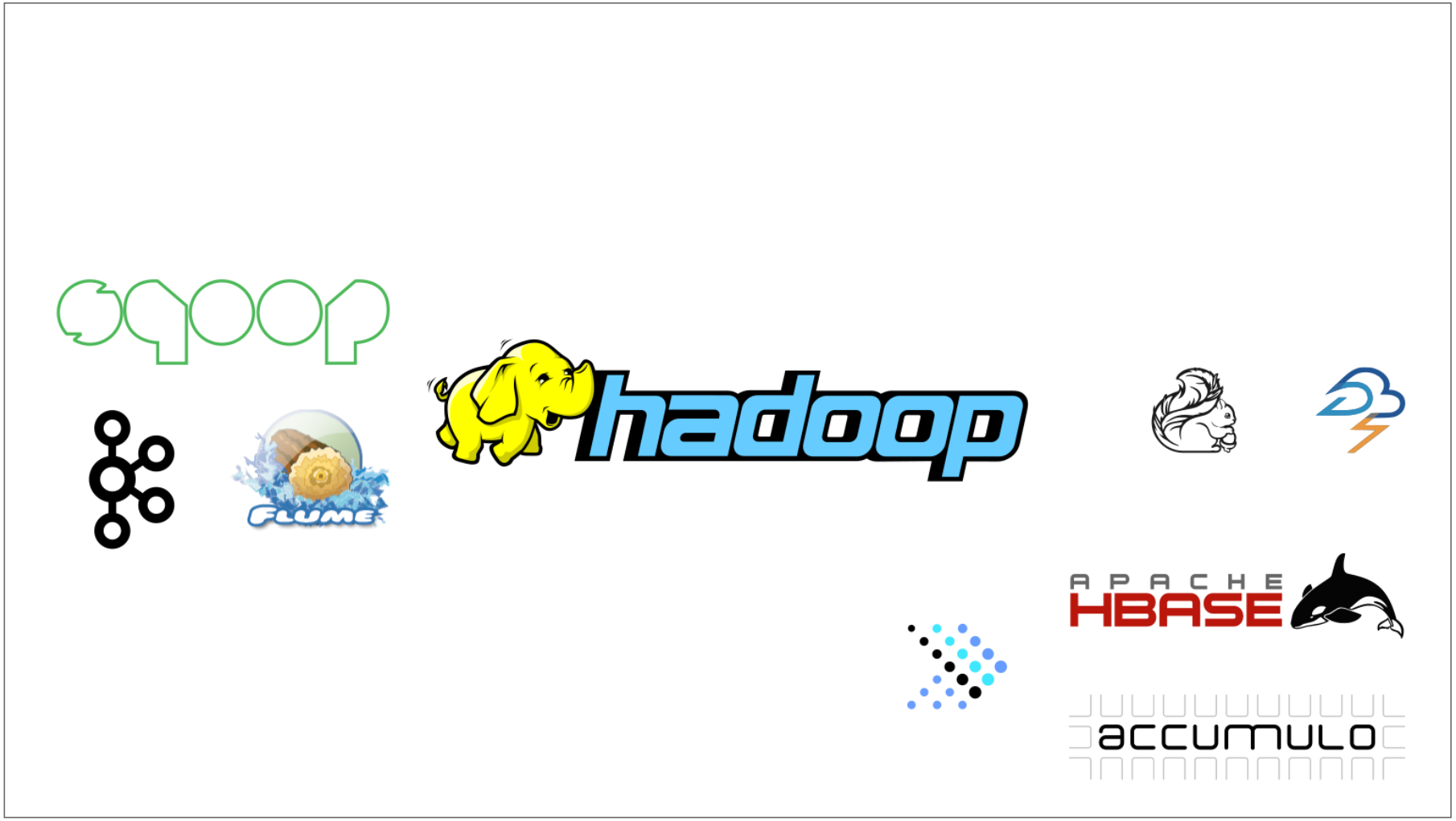
1.2 Apache Hadoop: The Hadoop ecosystem



1.2 Apache Hadoop: The Hadoop ecosystem



1.2 Apache Hadoop: The Hadoop ecosystem



1.2 Apache Hadoop: The Hadoop ecosystem

1.2 Apache Hadoop: The Hadoop ecosystem

1.2 Apache Hadoop: The Hadoop ecosystem

1.2 Apache Hadoop: The Hadoop ecosystem

1.2 Apache Hadoop: The Hadoop ecosystem

1.2 Apache Hadoop: The Hadoop ecosystem

1.2 Apache Hadoop: The Hadoop ecosystem

1.2 Apache Hadoop: The Hadoop ecosystem

1.2 Apache Hadoop: The Hadoop ecosystem

1.2 Apache Hadoop: The Hadoop ecosystem

1.3 Hadoop MapReduce weaknesses

- Rather heavy and rigid framework:
    - The developper must provide a map-reduce implementation of even the simpliest tasks (alleviated by higher-level tools)
    - Jobs must be written in Java (alleviated by higher-level tools)
- Complex workflows require to chain dozens (or many more) of MapReduce jobs:
    - Rather big job codebase with heavy maintenance
    - Global job optimization is left to the developper
    - Complex jobs are slow:
        - Each job's partial result must be written to and then read from disk
        - Very poor performance on iterative algorithms (ex: machine learning)

1.3 Hadoop MapReduce weaknesses: how does Apache Spark address these issues ?

- Provides high-level abstractions for many basic operations that can be combined with few lines of code to build complex workflows (including using scripting languages): analysts can focus on their business problems rather than being absorbed by implementation details. Spark is easy to use.
- Embed workflow optimizations which aim at minimizing data movements: Spark is fast.
- Operations are carried out in-memory as much as possible: Spark is much faster.

2. Apache Spark: A general presentation

2.1 General project information

- Started in 2009 at UC Berkeley AMPLab (Matei Zaharia PhD thesis).
- Handed over to the Apache Software Foundation in 2013, Apache top-level project since 2014.
- Very active community: 1000+ contributors, 30k+ JIRA tickets, 20k+ Github stars, Spark&AI summits.
- Contributions from most of the top tech companies. Most contributions come from Databricks.
- Written in Scala.
- Spark 2.0 released in July 2016, Spark 3.0 released in September 2020.

2.2 Apache Spark: 4 main libraries

2.3 Apache Spark: 4 supported languages

`org.apache.spark`

`pyspark`
`koalas`

`sparkr`
`sparklyr`

2.4 Apache Spark: 4 supported resource managers

2.5 A great variety of supported data sources

- Storage: local FS, HDFS, cloud storage (AWS S3, Azure Storage, Google Cloud Storage, etc.)
- Databases/data warehouses: Apache Hive, many available connectors (Cassandra, CouchBase, Neo4j, etc.)
- Data streams: Apache Kafka, AWS Kinesis, etc.
- File formats: CSV, JSON, Parquet, Avro, ORC, etc.

2.6 What is Apache Spark ? (continued)

- "Apache Spark is an open-source in-memory general-purpose distributed-computing framework"
- "Apache Spark is a unified engine for big data processing, data science, machine learning and data analytics workloads"

3. What is the architecture of a Spark application ?

3.1 The Spark application: A two-level distributed architecture

- A Spark application is a hierachical cluster of individual entities.
- One program is called the **Spark driver** (one per application) which is in charge of translating the application's code into tasks, optimizing the tasks, allocating the tasks, monitoring their execution, relauching failed tasks.
- All the other entities are called **Spark executors** and their sole purpose is to execute the tasks the driver assigned to them.

3.2 Configuring a Spark application

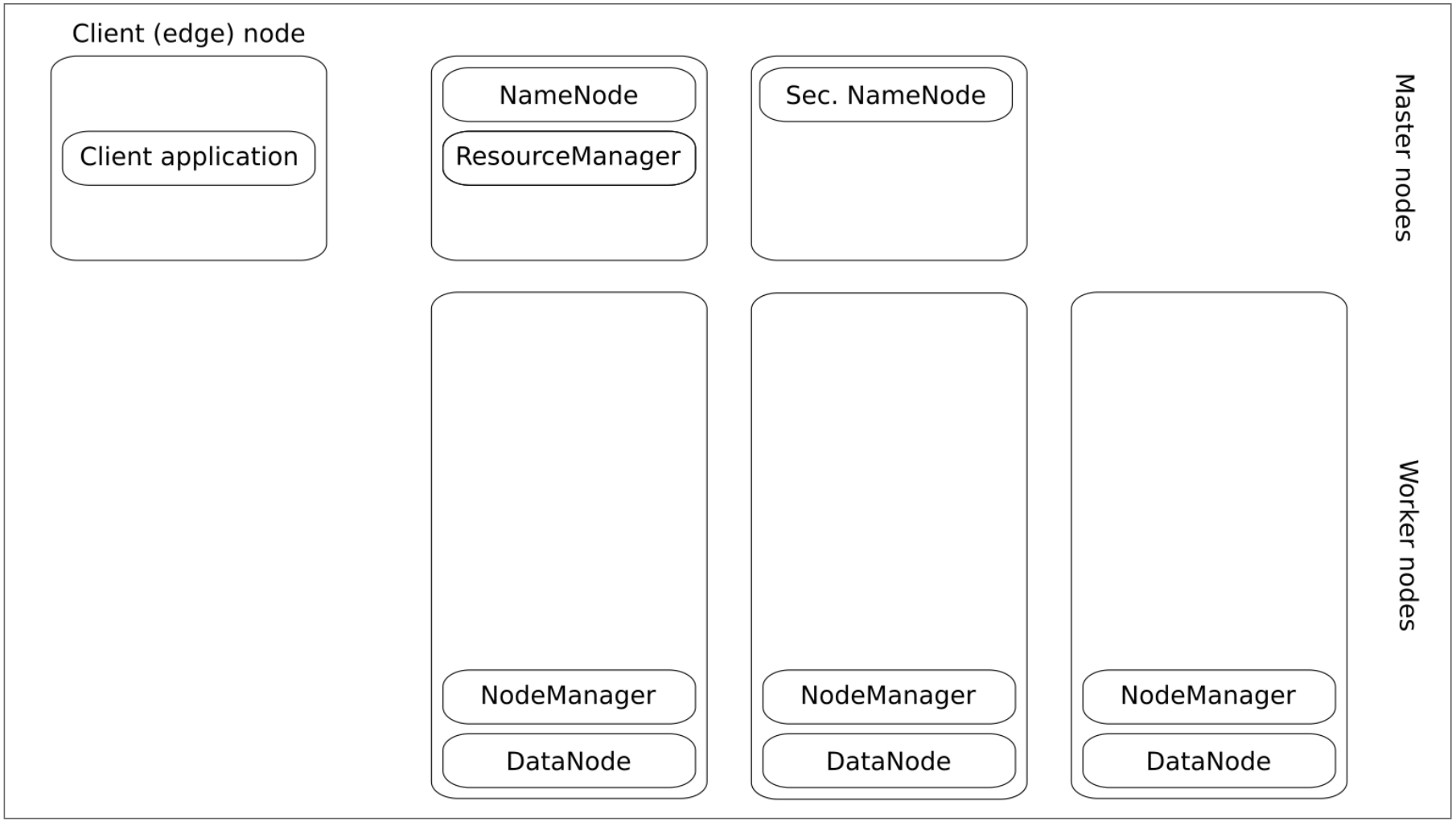
- The Spark driver and the Spark executors are JVMs.
- When running on a cluster, Spark JVMs run in ad-hoc containers.

The essential part of configuring a Spark application resides in choosing:

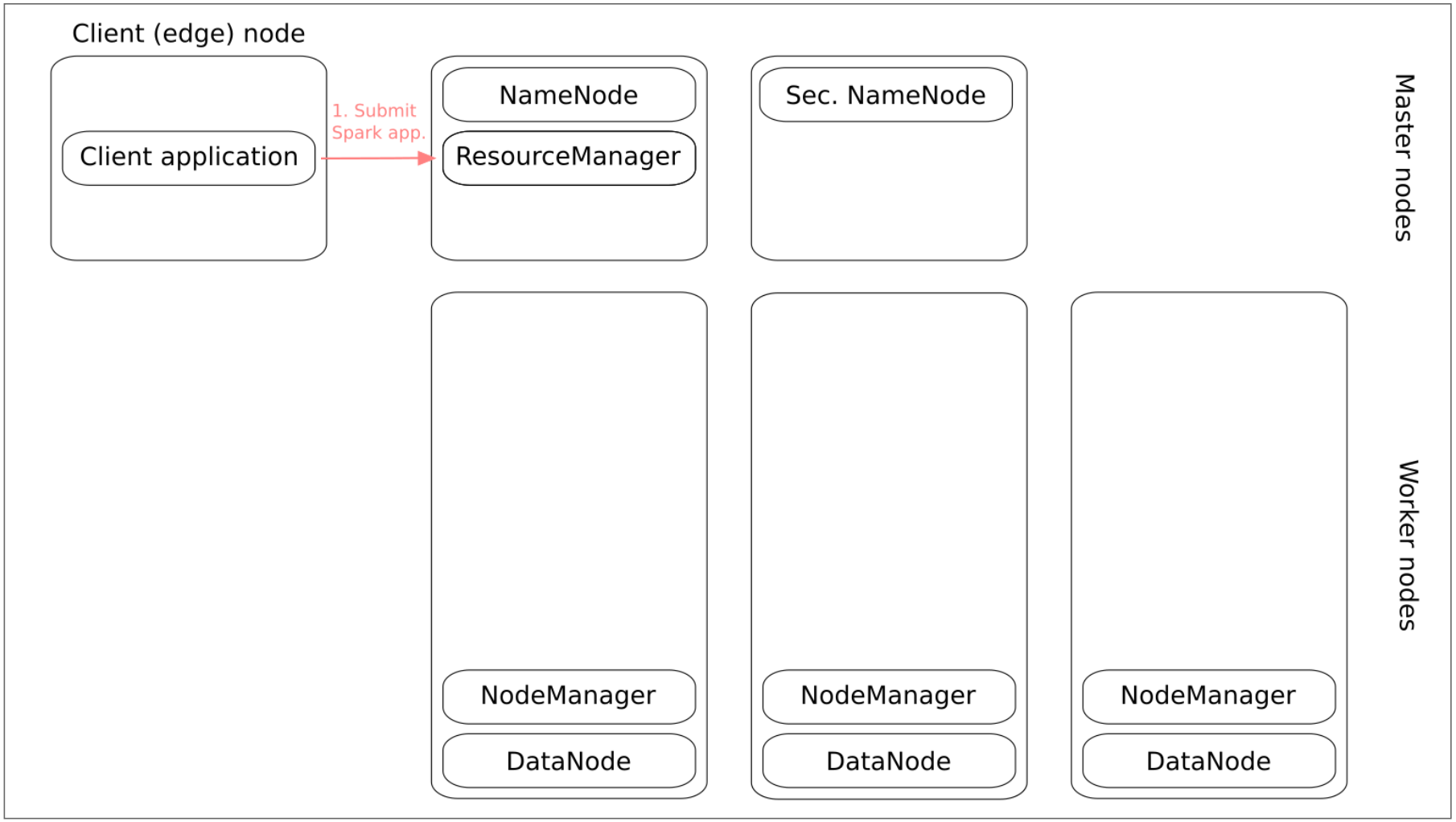
- The number of executors.
- The amount of CPU resources allocated to the driver and to each of the executors.
- The amount of memory resources allocated to the driver and to each of the executors (JVM + overhead).
- The deploy mode: does the driver run on (cluster mode) or outside (client mode) from the cluster ?

Notice: Each JVM can take advantage of several cores. Each working thread is called a **Spark worker**.

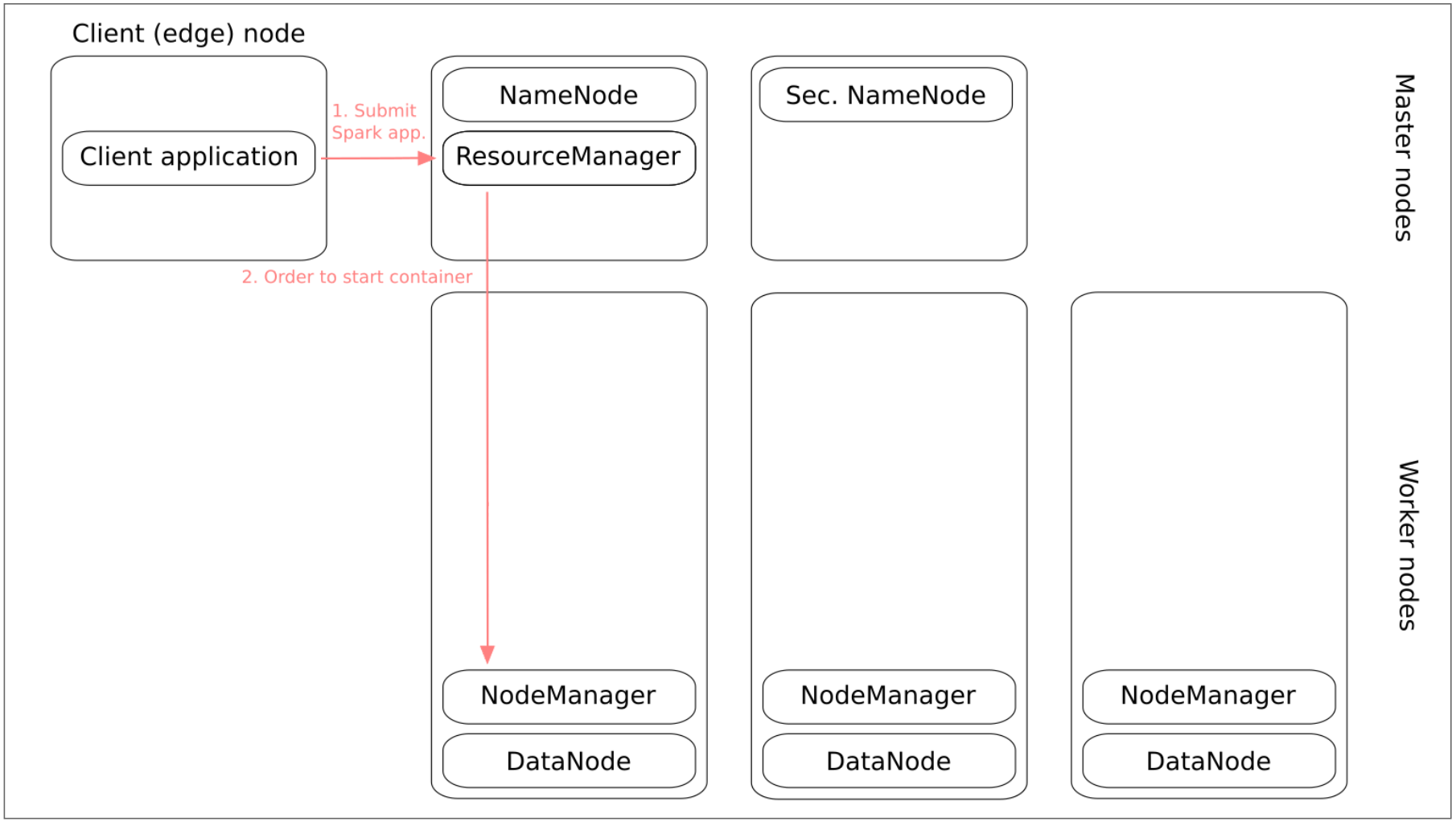
3.3 How does a YARN Spark application run in cluster mode ?



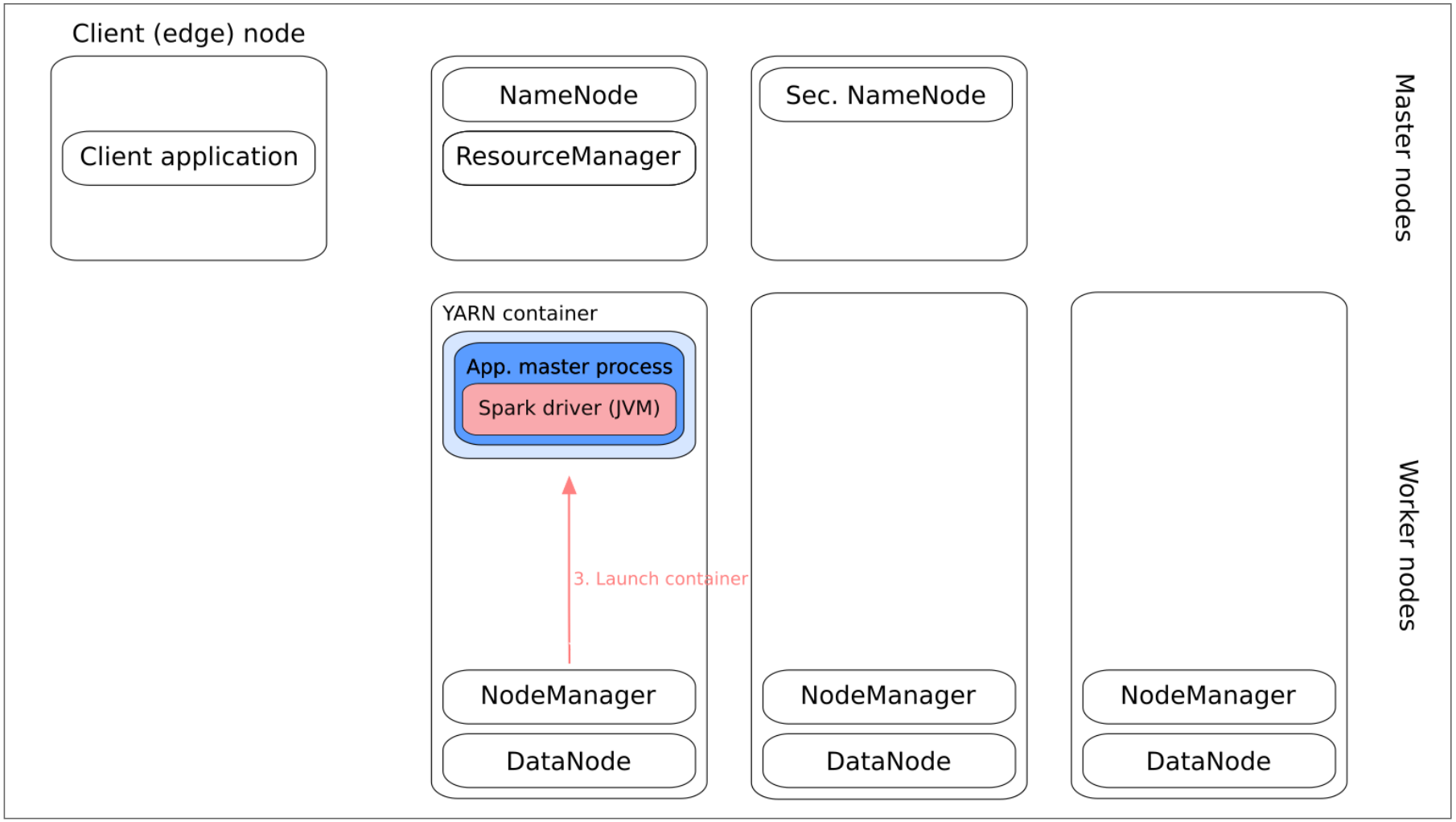
3.3 How does a YARN Spark application run in cluster mode ?



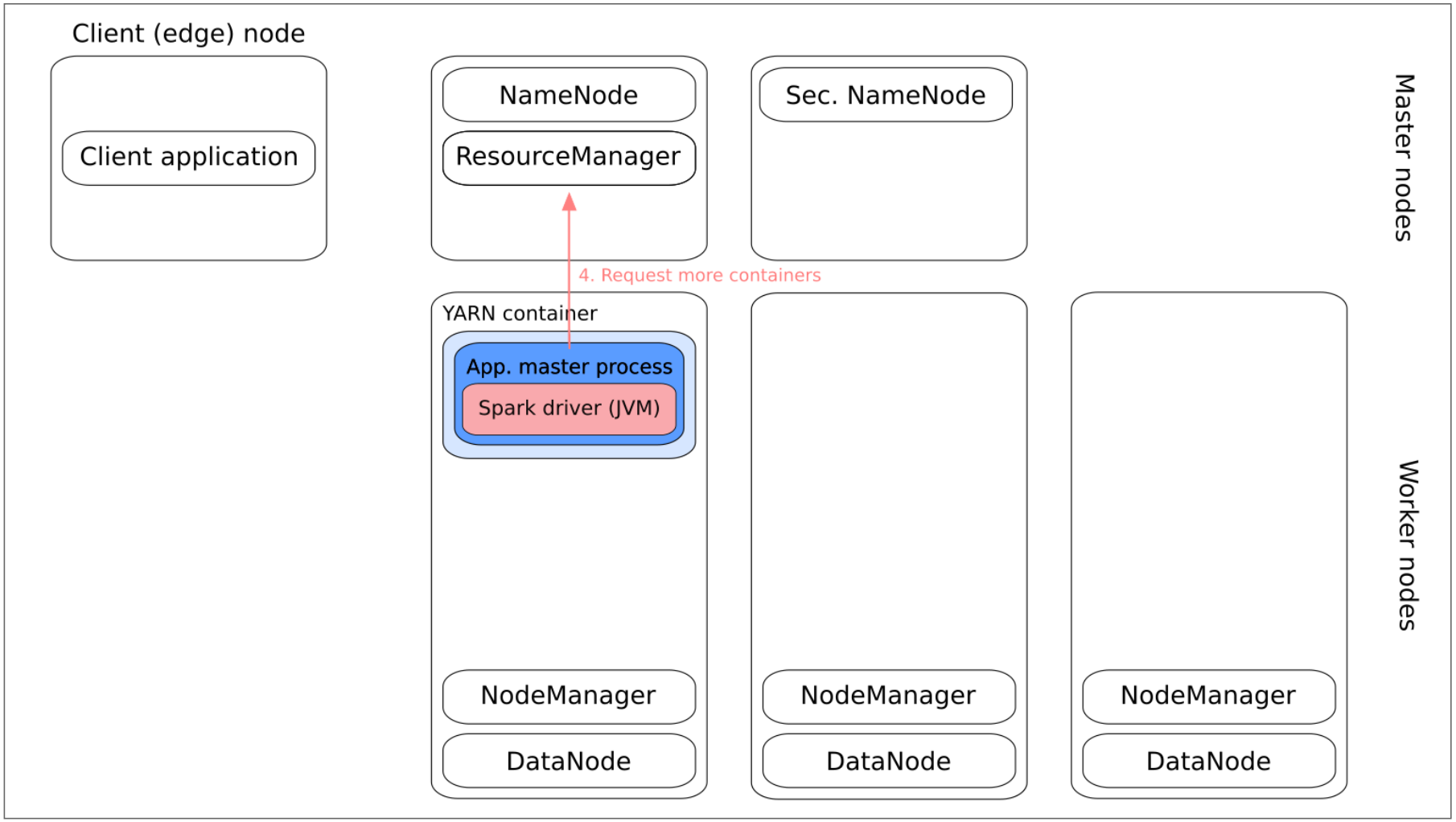
3.3 How does a YARN Spark application run in cluster mode ?



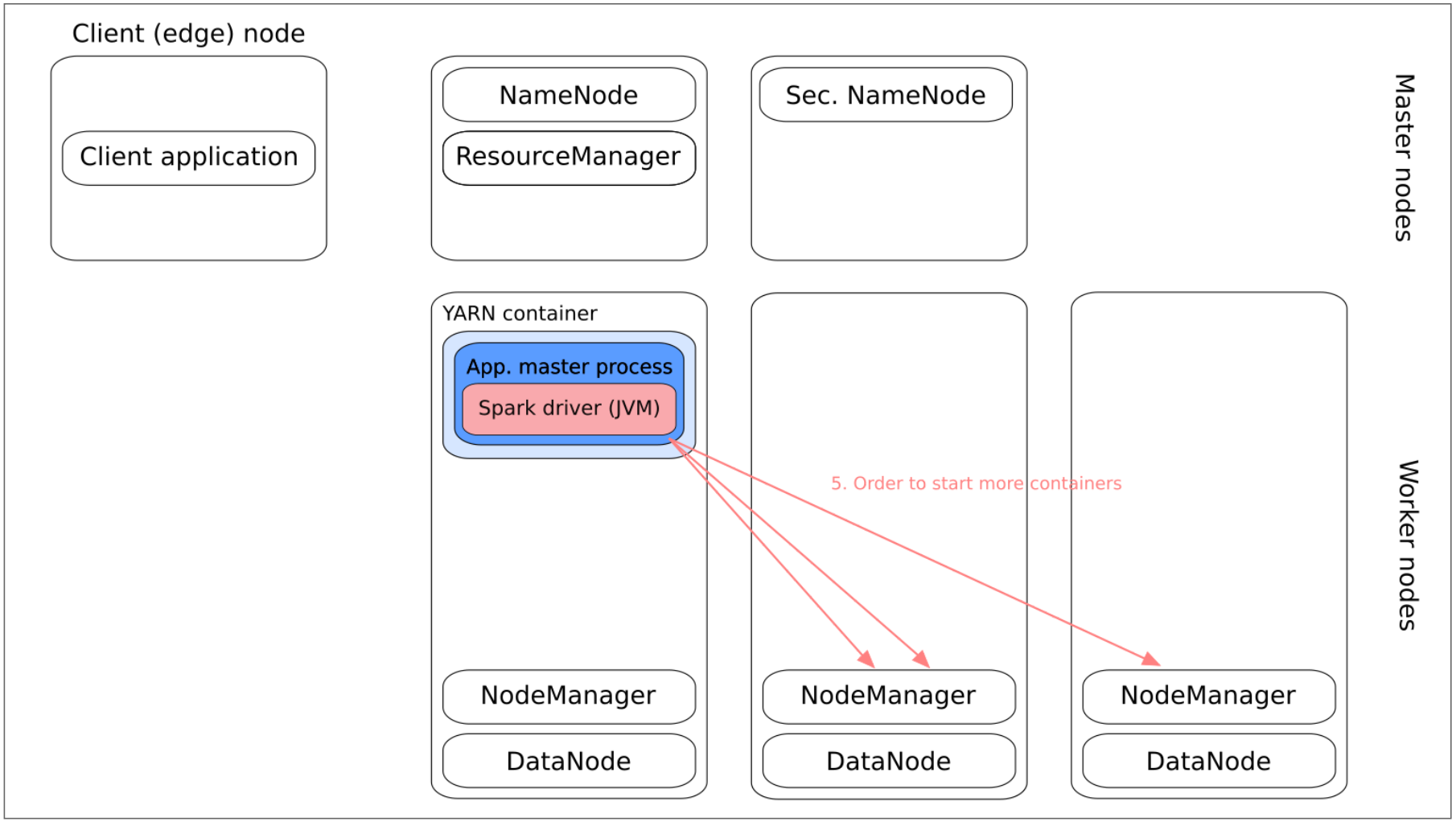
3.3 How does a YARN Spark application run in cluster mode ?



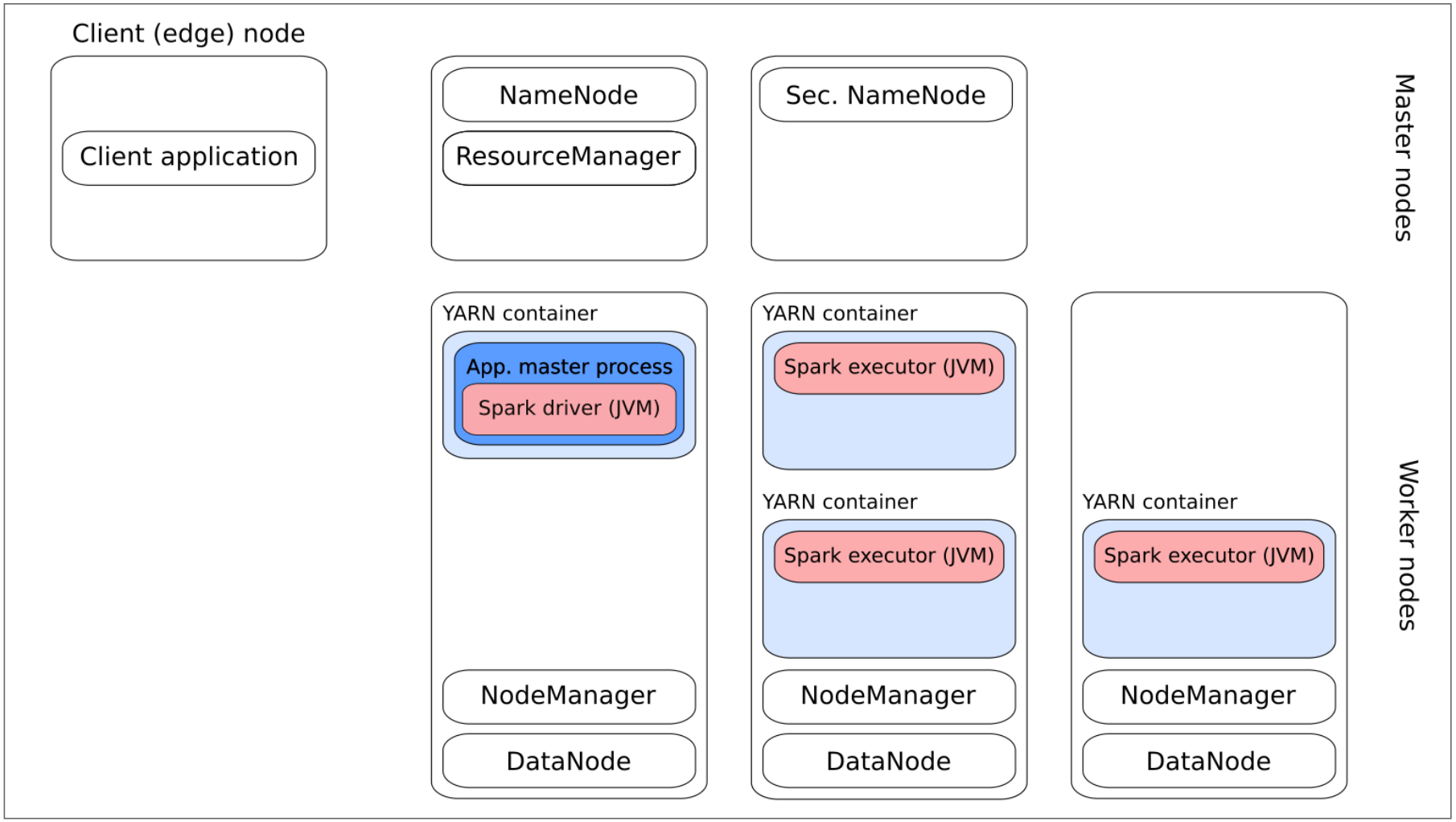
3.3 How does a YARN Spark application run in cluster mode ?



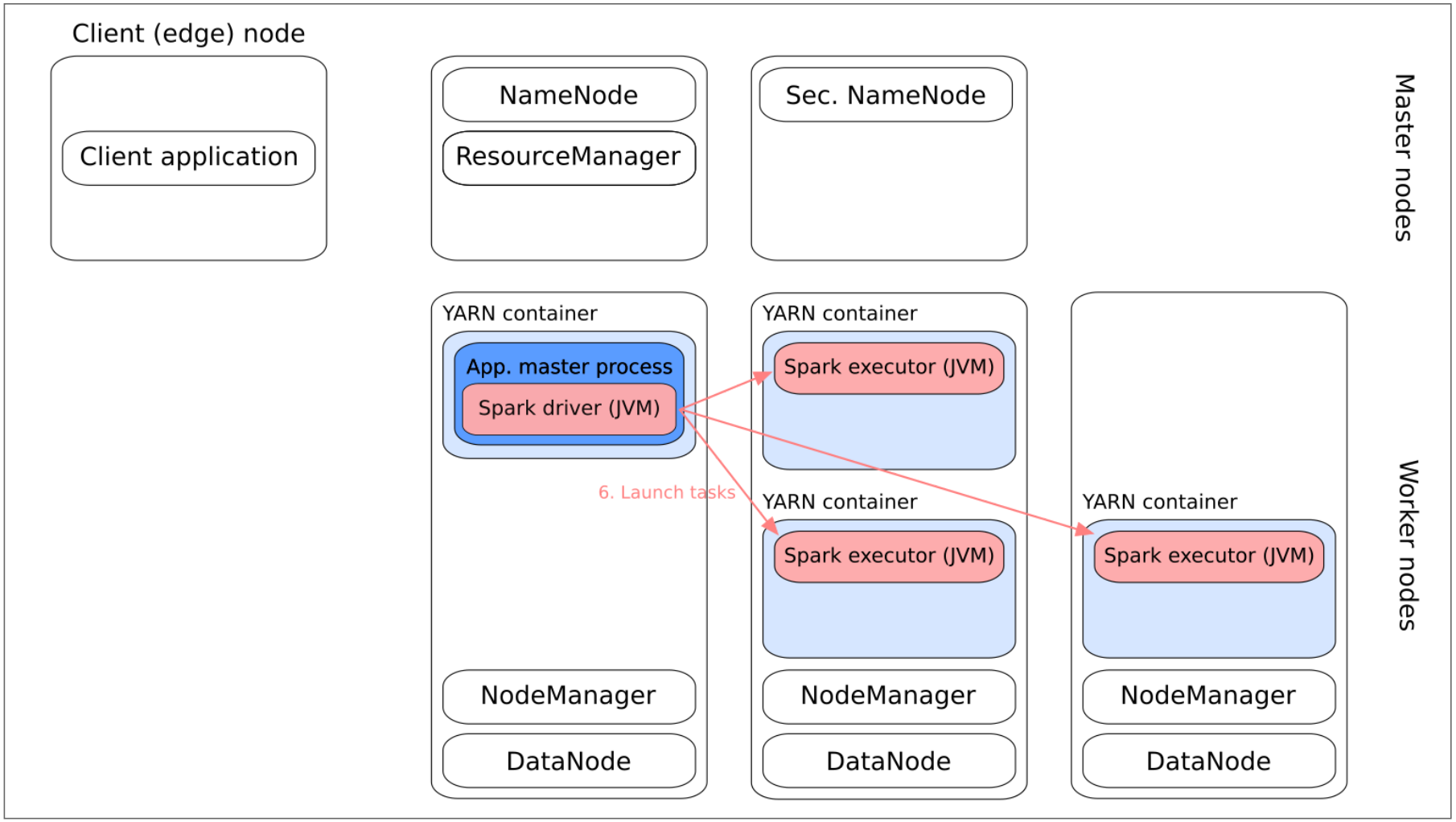
3.3 How does a YARN Spark application run in cluster mode ?



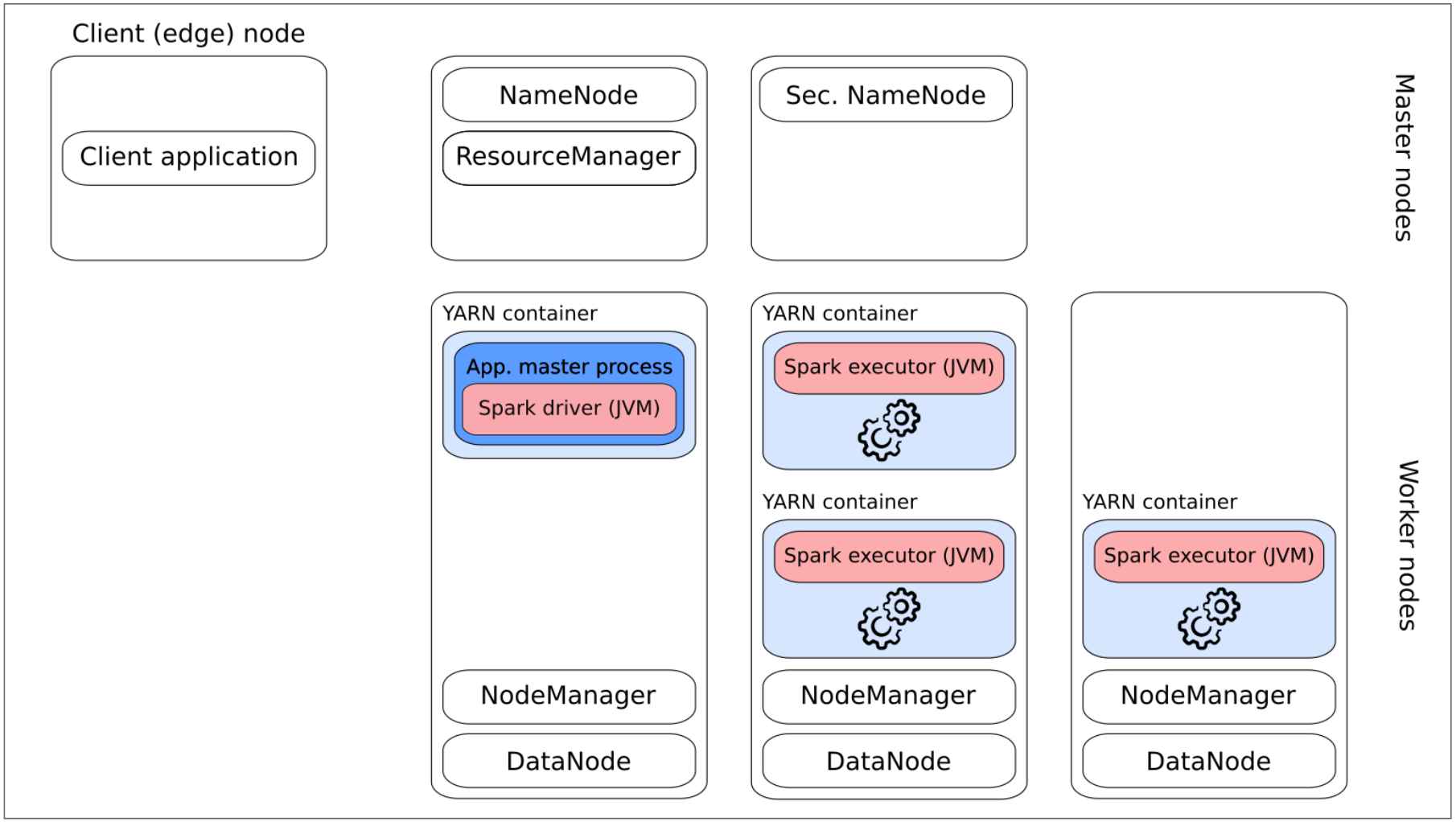
3.3 How does a YARN Spark application run in cluster mode ?



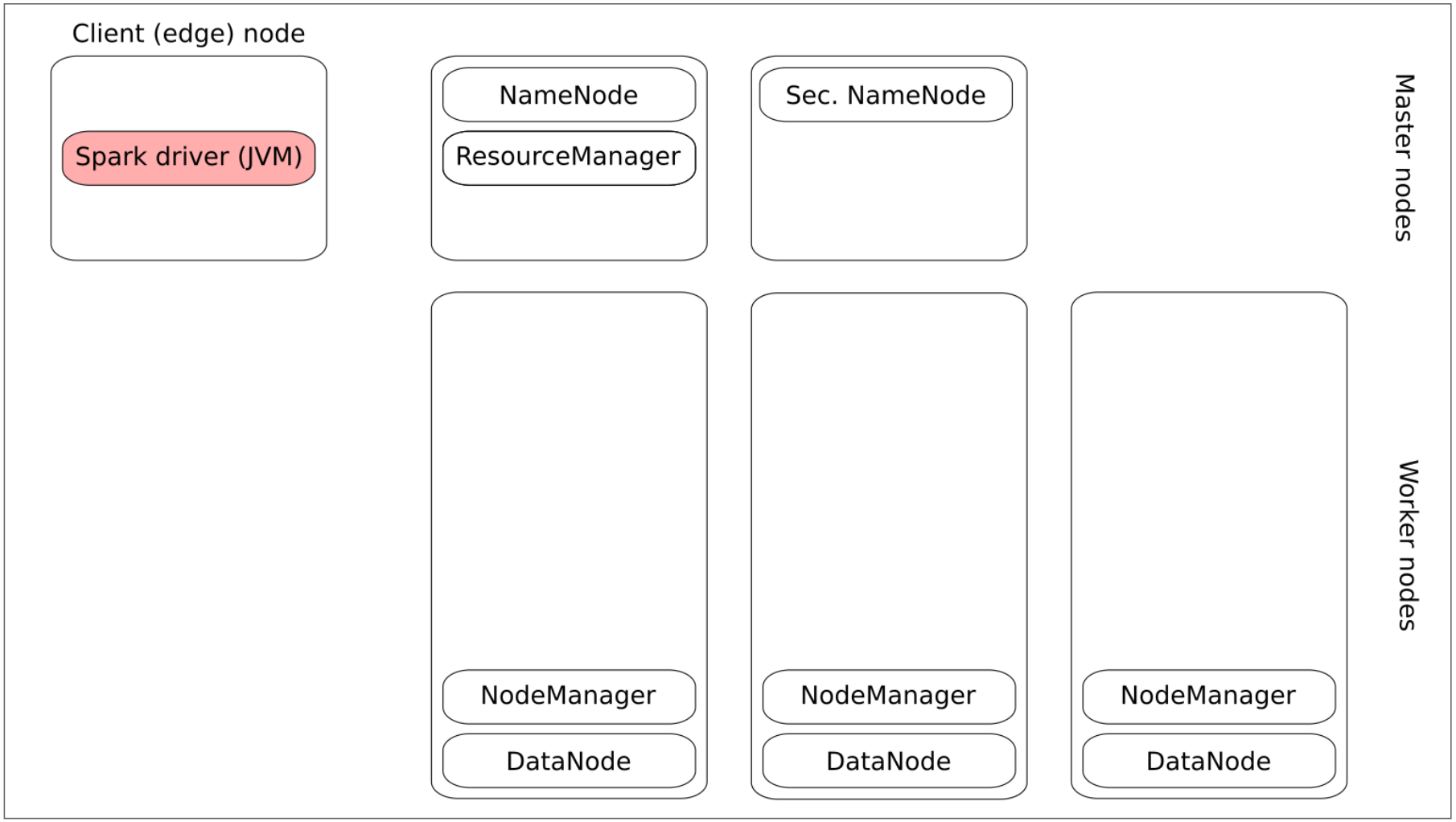
3.3 How does a YARN Spark application run in cluster mode ?



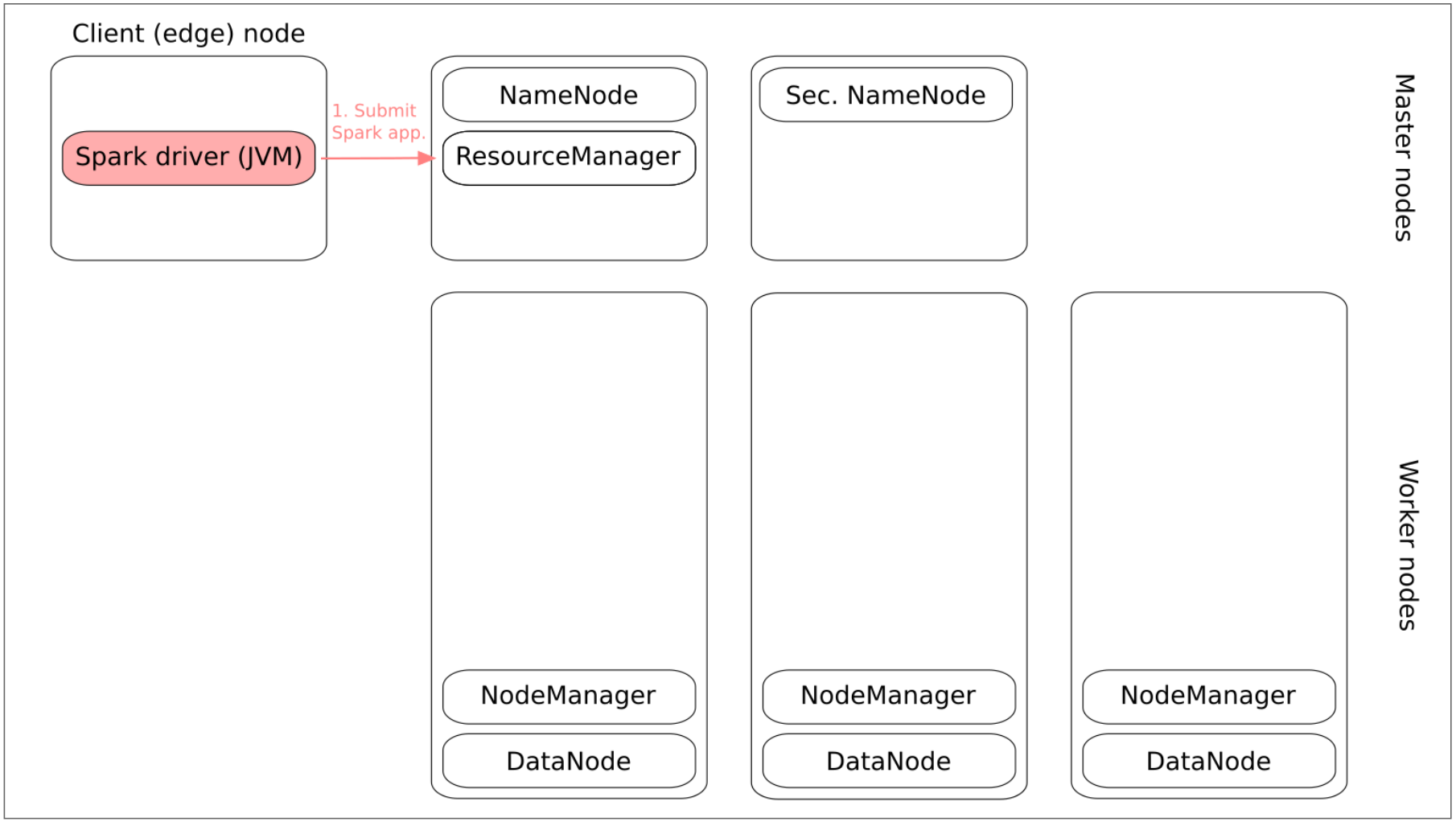
3.3 How does a YARN Spark application run in cluster mode ?



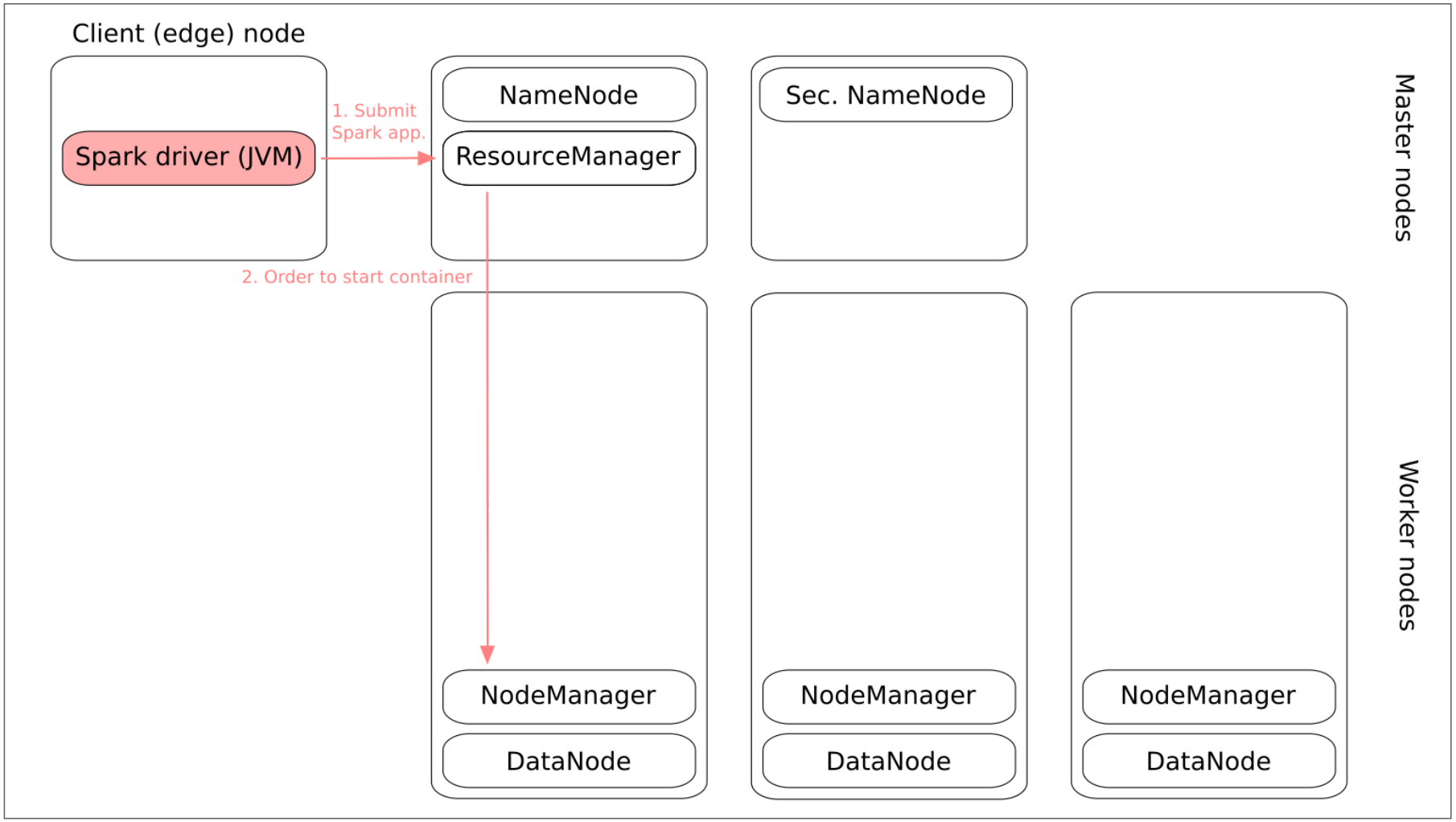
3.4 How does a YARN Spark application run in client mode ?



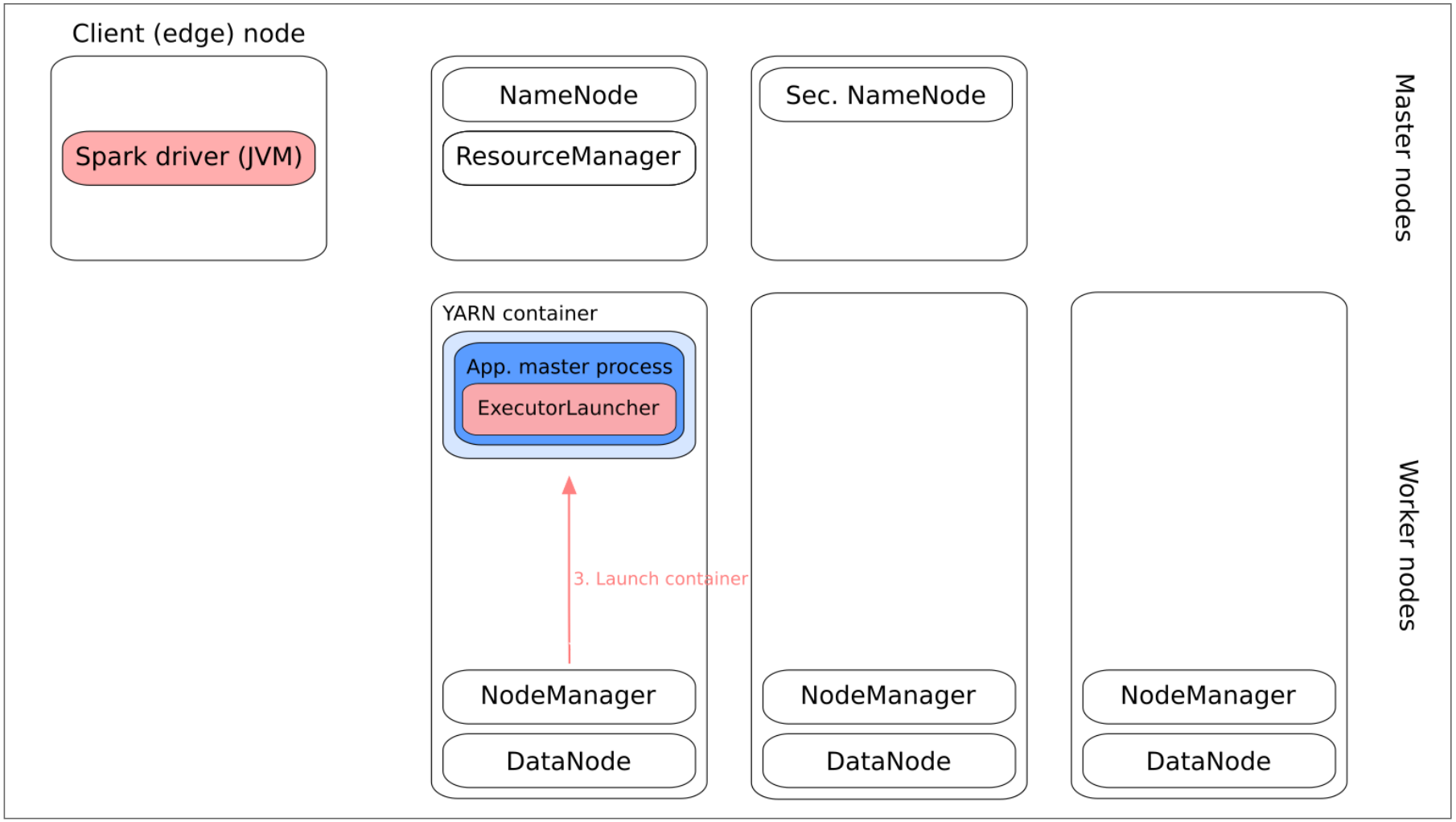
3.4 How does a YARN Spark application run in client mode ?



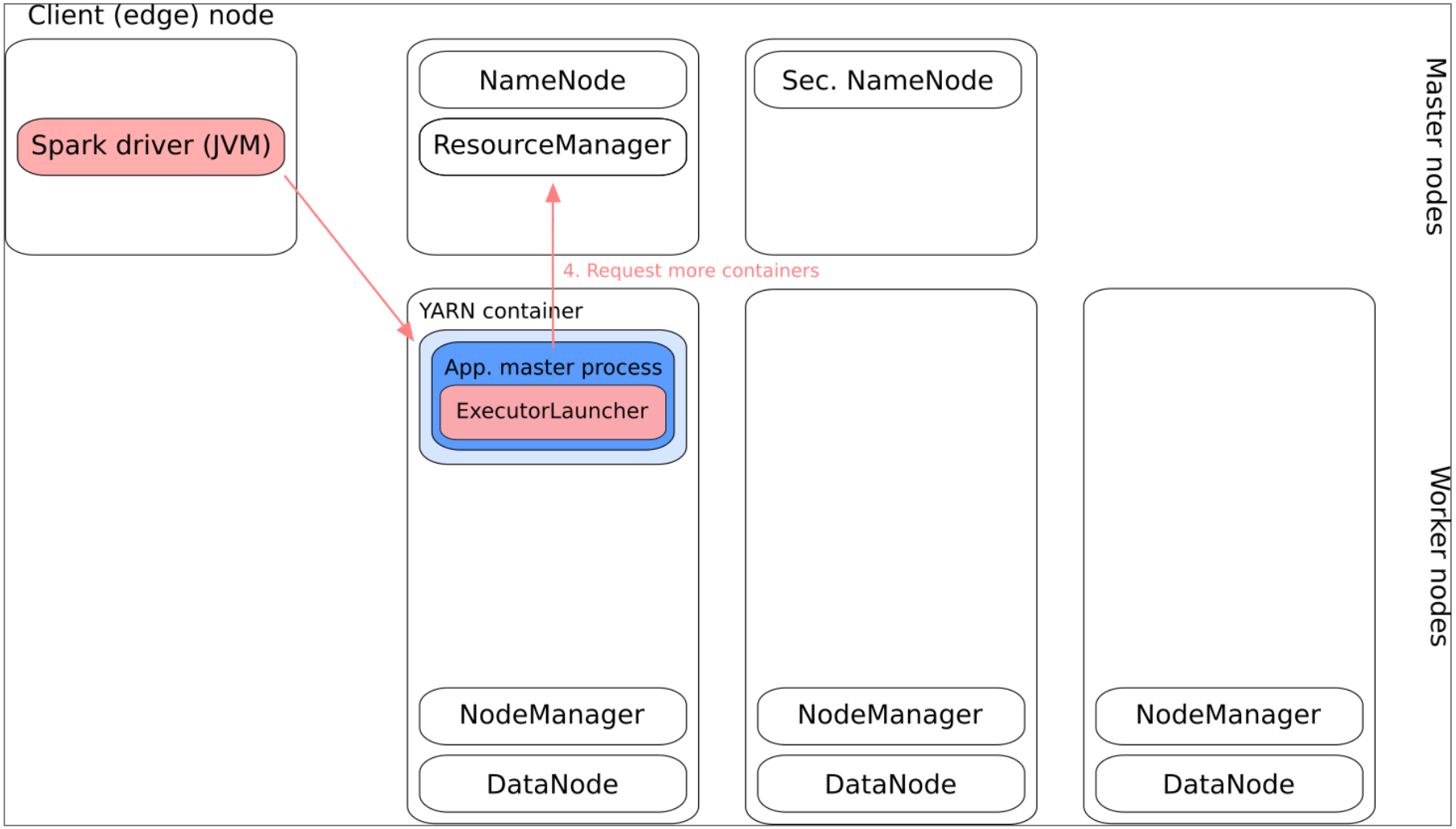
3.4 How does a YARN Spark application run in client mode ?



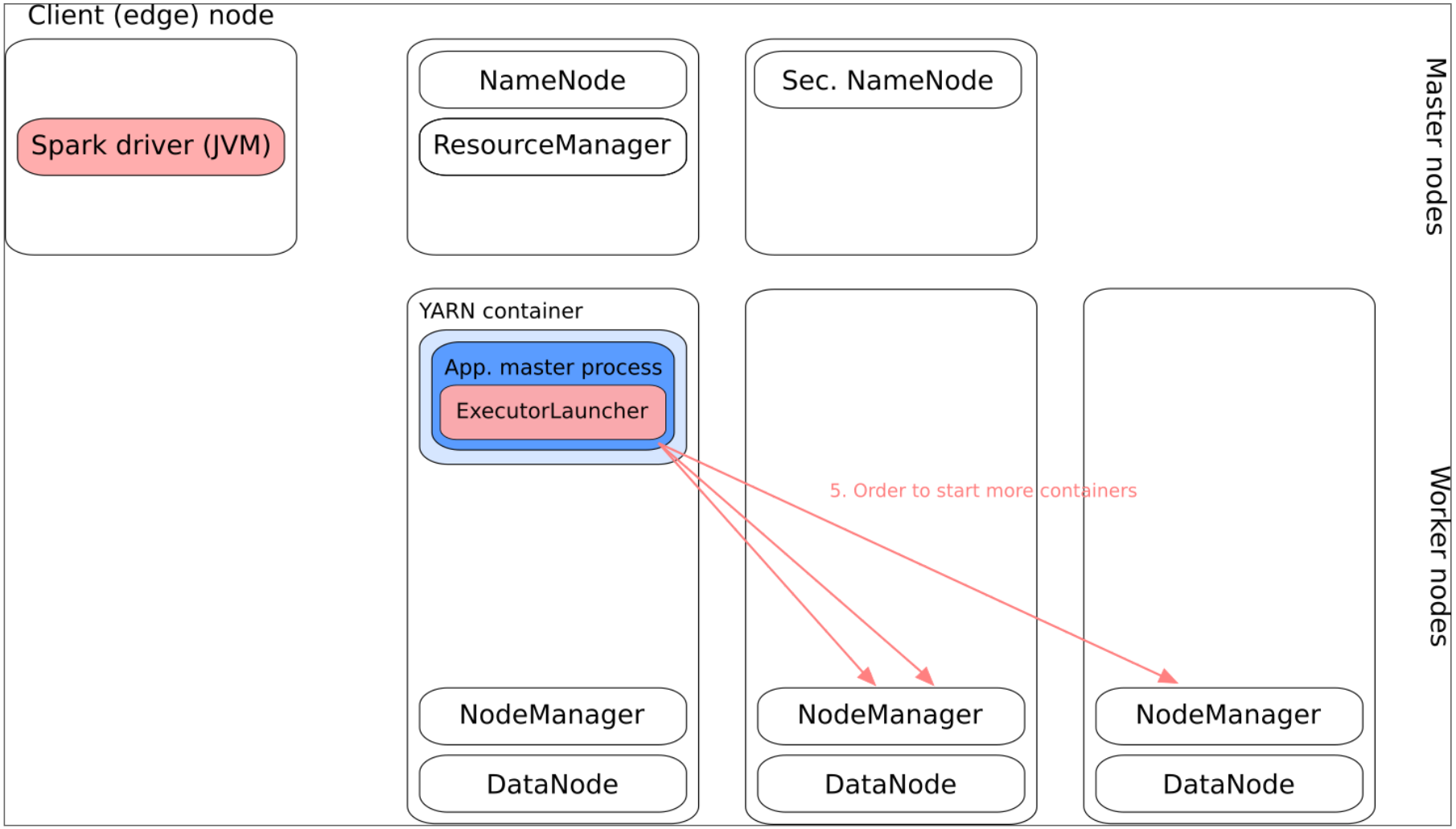
3.4 How does a YARN Spark application run in client mode ?



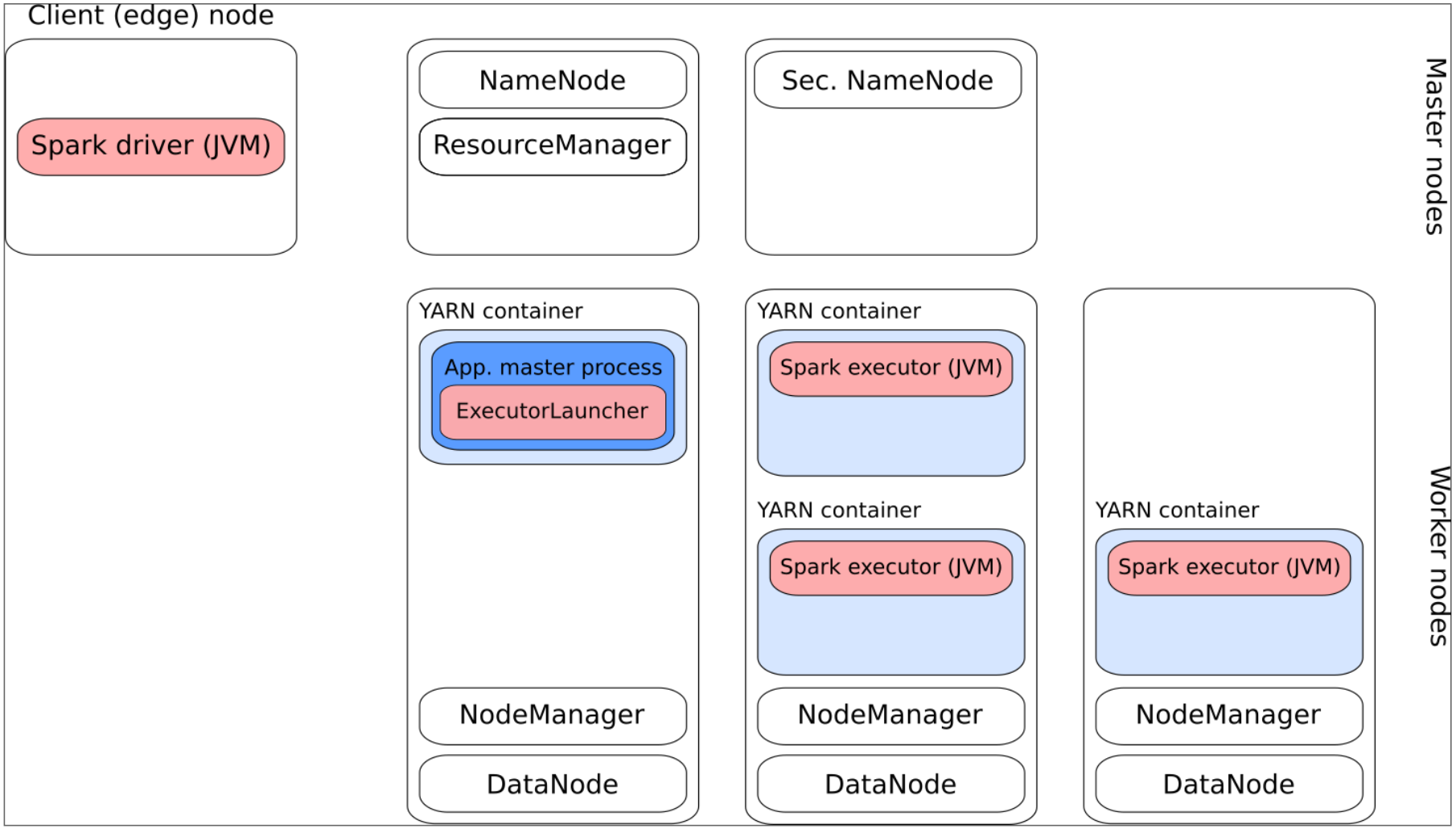
3.4 How does a YARN Spark application run in client mode ?



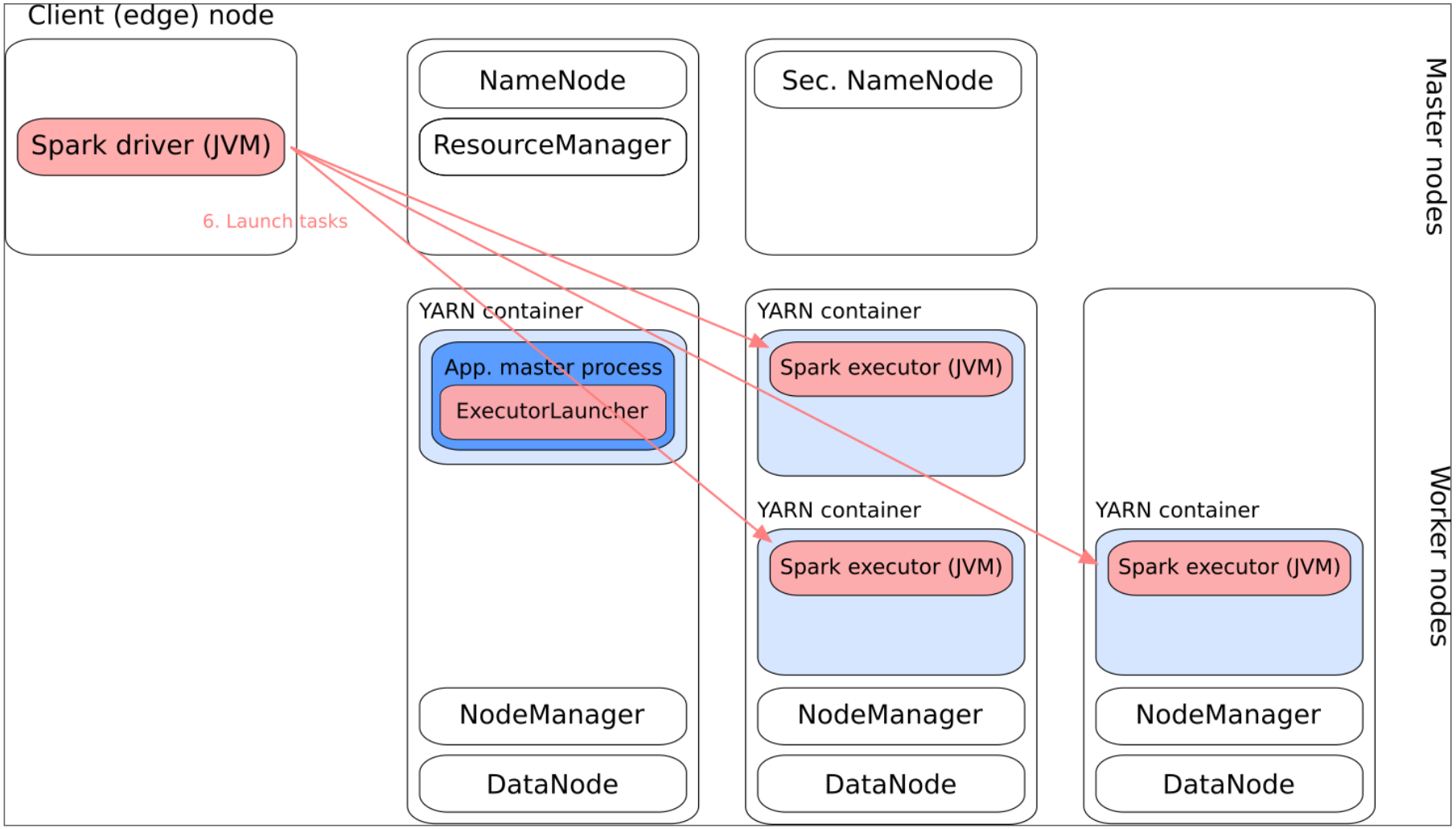
3.4 How does a YARN Spark application run in client mode ?



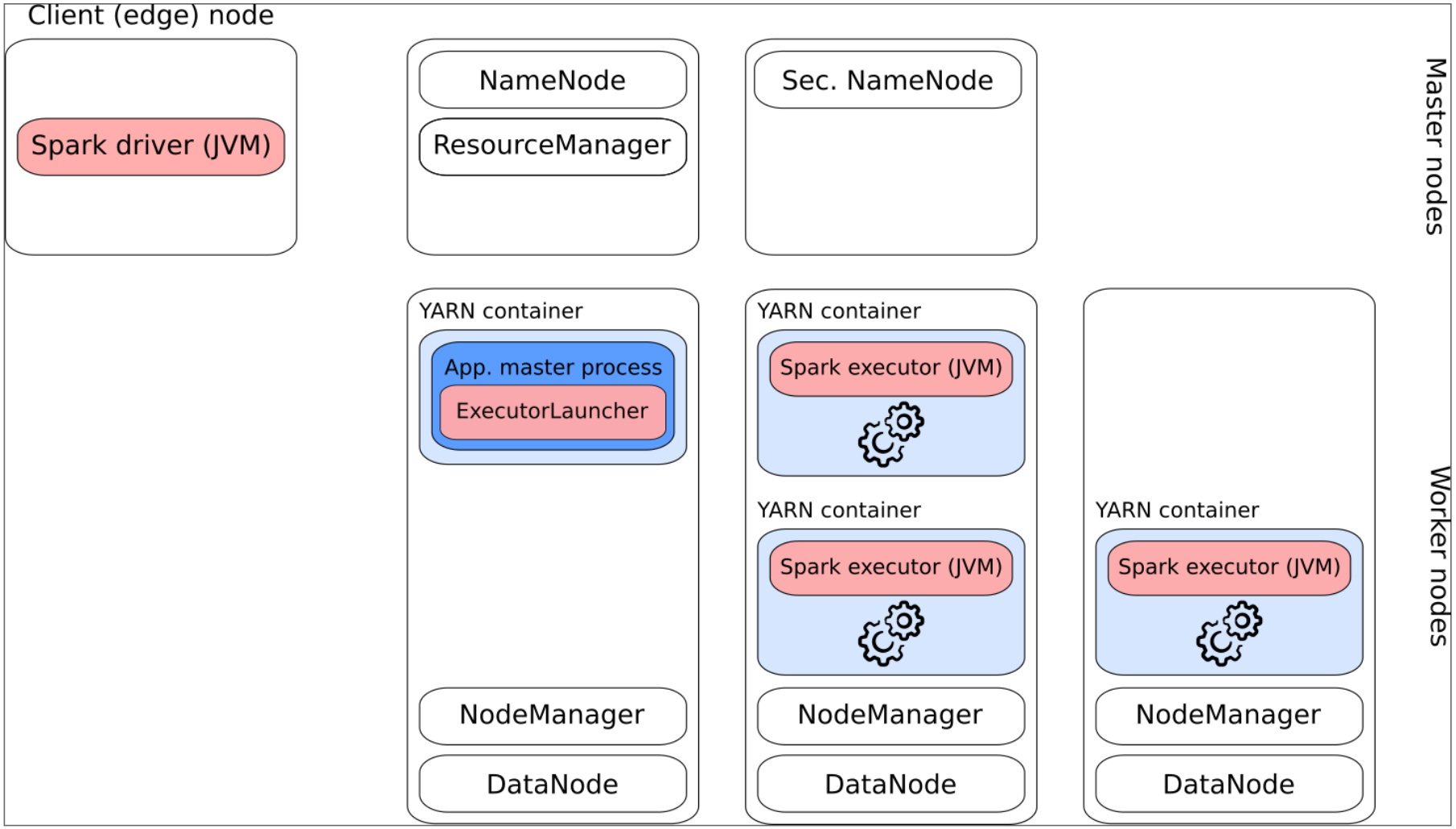
3.4 How does a YARN Spark application run in client mode ?



3.4 How does a YARN Spark application run in client mode ?



3.4 How does a YARN Spark application run in client mode ?



4. Introducing Spark main features with SparkSQL

4.1 Interacting with Spark: The Spark session

In [2]:

```
import pyspark

spark = pyspark.sql.SparkSession.builder\
    .config('spark.app.name', 'mpdata-talk')\
    .config('spark.master', 'local[*]')\
    .getOrCreate()
```

A Spark application can also be launched from the command line using the `spark-submit` command.

4.1 Interacting with Spark: The Spark session

In [3]:

```
spark
```

Out[3]:

SparkSession - in-memory

SparkContext

Spark UI

Version	`v2.4.7`
Master	`local[*]`
AppName	`mpdata-talk`

4.2 Spark's fundamental data structure: the Spark DataFrame

- Spark data is represented using an abstraction called a DataFrame (Python API).
- A Spark DataFrame object does not contain data: a Spark DataFrame reprensents a collection of operations to be performed on one or several data sources.
- Spark DataFrames are built on a lower-level abstraction called Resilient Distributed Datasets (RDDs) which are no longer part of the public API since Spark 2.0 and should not be used directly.
- Spark uses indeed **lazy evaluation**:
    - Computations are triggered only by special methods called **actions**.
    - Lazy evaluation allows to perform a global optimization of the operations to be performed.

4.2 Spark's fundamental data structure: the Spark DataFrame

- Spark distinguishes two types of Dataframe operations (methods):
    - **Actions**: An action triggers the computation of the DataFrame on which they were called. Calling an action submits a job. Each Spark job has been launched by a call to an action. Actions: `show`, `collect`, `toPandas`, `count`, `take`, write operations.
    - **Transformations**: A transformation simply returns a new DataFrame. Example: `select`, `filter`, `groupBy`, `join`, `withColumn`, etc.

4.2 Spark's fundamental data structure: the Spark DataFrame

In [4]:

```
spark.read\
    .parquet('data/flights.parquet')\
    .show(5)
```

```
+----------+---------+-------+--------------+-------
------+--------+--------+
|DayofMonth|DayOfWeek|Carrier|OriginAirportID|DestAir
portID|DepDelay|ArrDelay|
+----------+---------+-------+--------------+-------
------+--------+--------+
|        16|        2|     DL|          14635|
10397|      -5|     -20|
|        28|        7|     UA|          13930|
12266|      -1|     -18|
|        10|        3|     UA|          11292|
11042|      -1|      -7|
|        13|        6|     WN|          14107|
14869|      25|      22|
|        10|        3|     OO|          12892|
13851|      -4|      12|
+----------+---------+-------+--------------+-------
```

```
------+--------+--------+
only showing top 5 rows
```

4.2 Spark's fundamental data structure: the Spark DataFrame

In [5]:

```
spark.read\
    .parquet('data/flights.parquet')\
    .select('DayofMonth', 'DayOfWeek', 'Carrier', 'DepDelay')  # Transformation
```

Out[5]:

```
DataFrame[DayofMonth: int, DayOfWeek: int, Carrier: s
tring, DepDelay: int]
```

- The Spark reader `spark.read.parquet` returns a `spark.sql.DataFrame` object.
- The `select` method is a transformation and therefore returns a new `spark.sql.DataFrame`.

4.2 Spark's fundamental data structure: the Spark DataFrame

In [6]:

```
spark.read\
    .parquet('data/flights.parquet')\
    .select('DayofMonth', 'DayOfWeek', 'Carrier', 'DepDelay')\
    .show(5)  # Action
```

```
+----------+---------+-------+--------+
|DayofMonth|DayOfWeek|Carrier|DepDelay|
+----------+---------+-------+--------+
|        16|        2|     DL|      -5|
|        28|        7|     UA|      -1|
|        10|        3|     UA|      -1|
|        13|        6|     WN|      25|
|        10|        3|     OO|      -4|
+----------+---------+-------+--------+
only showing top 5 rows
```

- The Spark reader `spark.read.parquet` returns a `spark.sql.DataFrame` object.
- The `select` method is a transformation and therefore returns a new `spark.sql.DataFrame`.
- The `show` method is an action and therefore triggers a Spark job and returns the results.

4.3 Spark’s fundamental data processing unit: the partition

- Spark splits its data into chunks called **partitions**
- All records within a given partition will be processed together by the same core (Spark worker) within the same Spark executor.
- Data partitions are the smallest data processing unit: the processing of one data partition by a single Spark worker thread defines Spark's elemental processing unit: the **task**.

Notices:

- A Spark executor (JVM) gathers one or more workers (threads): each worker run on its own core.
- When asked to write its output, Spark creates one file per partition.

Warning: Row ordering "does not make sense" any longer with ditributed data: be careful with sort-based operations

4.3 Spark's fundamental data processing unit: the partition

In [9]:

```
flight_data = spark.read\
    .parquet('data/flights.parquet')\
    .persist(pyspark.StorageLevel.MEMORY_AND_DISK)

flight_data.count()
```

Out[9]:

```
2702218
```

4.4 Spark's Dataframe operations: narrow vs. wide

- There are two types of DataFrame operations: narrow and wide operations.
- Can the operation be performed independently on each partition ?
    - **Yes**: the operation is a narrow operation: `select`, `filter`, simple map operations, etc.
    - **No**: the operation requires exchange of information and is called a wide operation: `groupBy.agg`, `join`, `orderBy`, window functions, etc.
- Wide operations imply suffling data with the associated impact on performance: wait for other executors, disk + network I/O.
- **Never forget that operations are distributed: do not code with Spark like you would with `pandas`.**

4.4 Spark's Dataframe operations: narrow vs. wide

- Describing Spark computations:
    - **Task**: Processing of one partition by one Spark worker (thread).
    - **Stage**: Block of narrow operations. Stages are delimited by shuffles. Completing a stage required to complete as many tasks as partitions at the beginning of the stage.
    - **Job**: Computation triggered by a Spark action. Can consist of one or several stages.
    - **Session**: A session is tied to the application's life. Spark allows to run more than one job per session.

Notice: The number of partitions is set when reading data and can change after a shuffle or at the user's command (`repartition`).

4.4 Spark's Dataframe operations: narrow vs. wide

In [10]:

```
spark.read\
    .parquet('data/flights.parquet')\
    .select('DayofMonth', 'DayOfWeek', 'Carrier', 'DepDelay')\
    .show(5)
```

```
+----------+---------+-------+--------+
|DayofMonth|DayOfWeek|Carrier|DepDelay|
+----------+---------+-------+--------+
|         16|        2|     DL|      -5|
|         28|        7|     UA|      -1|
|         10|        3|     UA|      -1|
|         13|        6|     WN|      25|
|         10|        3|     OO|      -4|
+----------+---------+-------+--------+
only showing top 5 rows
```

4.4 Spark's Dataframe operations: narrow vs. wide

In [11]:

```
import pyspark.sql.functions as sqlf

spark.read\
    .parquet('data/flights.parquet')\
    .groupBy('Carrier', 'DepDelay')\
    .agg(sqlf.max('DepDelay').alias('max_dep_delay'))\
    .show(5)
```

```
+-------+--------+--------------+
|Carrier|DepDelay|max_dep_delay|
+-------+--------+--------------+
|     WN|       0|             0|
|     OO|     -11|           -11|
|     DL|     193|           193|
|     UA|     160|           160|
|     B6|     129|           129|
+-------+--------+--------------+
only showing top 5 rows
```

4.5 Spark SQL query optimizer: Catalyst

In [12]:

```
airports = spark.read\
    .option('header', True)\
    .option('inferSchema', True)\
    .csv('data/airports.csv')\
    .select('airport_id', 'state', sqlf.col('name').alias('airport_name'))

airports.show(5)
```

```
+----------+-----+--------------------+
|airport_id|state|        airport_name|
+----------+-----+--------------------+
|     10165|   AK|                Adak|
|     10299|   AK|Ted Stevens Ancho...|
|     10304|   AK|       Aniak Airport|
|     10754|   AK|Wiley Post/Will R...|
|     10551|   AK|      Bethel Airport|
+----------+-----+--------------------+
only showing top 5 rows
```

4.5 Spark SQL query optimizer: Catalyst

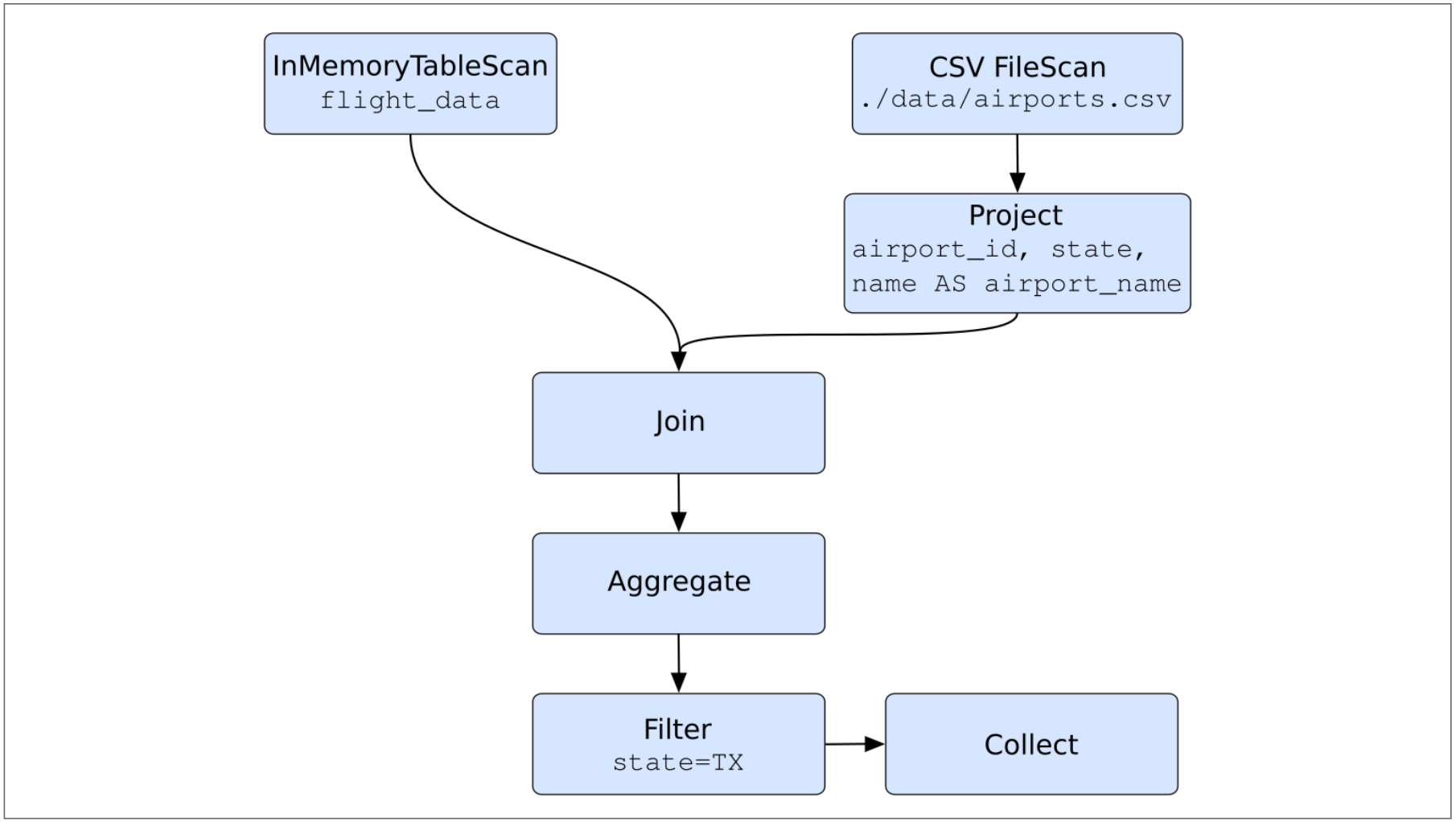
In [13]:

```
texas_delays = flight_data.join(airports,
                                flight_data.OriginAirportID==airports.airport_id,
                                how='left')\
    .groupBy('state', 'airport_name')\
    .agg(sqlf.max('DepDelay').alias('max_dep_delay'))\
    .filter(sqlf.col('state')=='TX')

texas_delays.show(5)
```

```
+-----+--------------------+-------------+
|state|        airport_name|max_dep_delay|
+-----+--------------------+-------------+
|   TX|San Antonio Inter...|         1213|
|   TX|George Bush Inter...|         1252|
|   TX|     William P Hobby|         1113|
|   TX|Dallas/Fort Worth...|         1145|
|   TX|   Dallas Love Field|          598|
+-----+--------------------+-------------+
only showing top 5 rows
```

4.5 Spark SQL query optimizer: Catalyst



4.5 Spark SQL query optimizer: Catalyst

In [14]:

```
texas_delays.explain(extended=False)
```

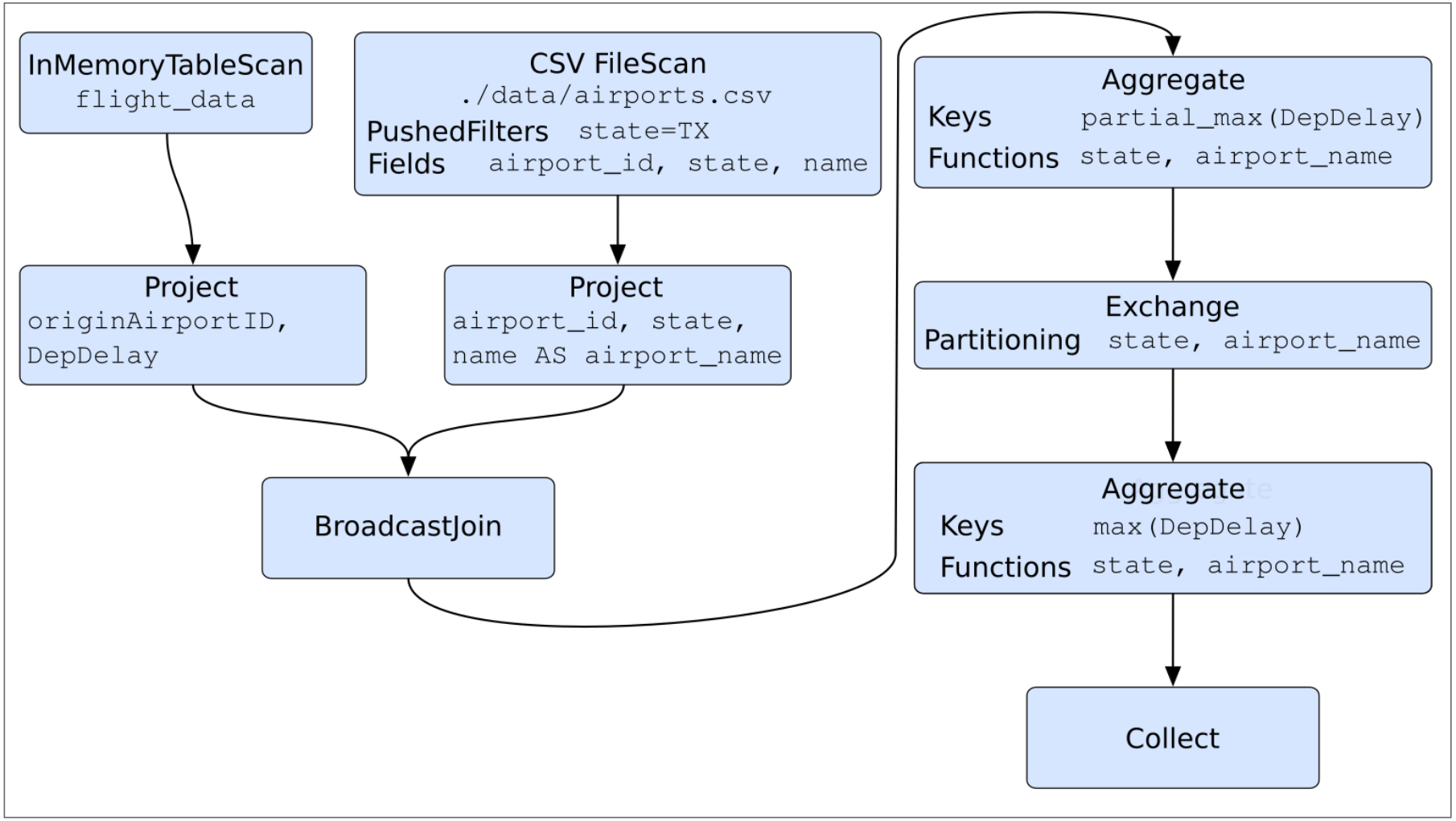
```
== Physical Plan ==
*(3) HashAggregate(keys=[state#454, airport_name#46
0], functions=[max(DepDelay#247)])
+- Exchange hashpartitioning(state#454, airport_name#
460, 200)
   +- *(2) HashAggregate(keys=[state#454, airport_nam
e#460], functions=[partial_max(DepDelay#247)])
      +- *(2) Project [DepDelay#247, state#454, airpo
rt_name#460]
         +- *(2) BroadcastHashJoin [OriginAirportID#2
45], [airport_id#452], Inner, BuildRight
            :- *(2) Filter isnotnull(OriginAirportID#
245)
            :  +- InMemoryTableScan [OriginAirportID#
245, DepDelay#247], [isnotnull(OriginAirportID#245)]
            :        +- InMemoryRelation [DayofMonth#
242, DayOfWeek#243, Carrier#244, OriginAirportID#245,
DestAirportID#246, DepDelay#247, ArrDelay#248], Stora
```

```
geLevel(disk, memory, 1 replicas)
            :                 +- *(1) FileScan parquet
[DayofMonth#85,DayOfWeek#86,Carrier#87,OriginAirportI
D#88,DestAirportID#89,DepDelay#90,ArrDelay#91] Batche
d: true, Format: Parquet, Location: InMemoryFileIndex
[file:/home/pierre/Documents/notes/spark/talk/data/fl
ights.parquet], PartitionFilters: [], PushedFilters:
[], ReadSchema: struct<DayofMonth:int,DayOfWeek:int,C
arrier:string,OriginAirportID:int,DestAirportID:int,D
epDelay...
            +- BroadcastExchange HashedRelationBroadc
astMode(List(cast(input[0, int, true] as bigint)))
               +- *(1) Project [airport_id#452, state
#454, name#455 AS airport_name#460]
                  +- *(1) Filter ((isnotnull(state#45
4) && (state#454 = TX)) && isnotnull(airport_id#452))
                     +- *(1) FileScan csv [airport_id
#452,state#454,name#455] Batched: false, Format: CSV,
Location: InMemoryFileIndex[file:/home/pierre/Documen
ts/notes/spark/talk/data/airports.csv], PartitionFilt
ers: [], PushedFilters: [IsNotNull(state), EqualTo(st
ate,TX), IsNotNull(airport_id)], ReadSchema: struct<a
irport_id:int,state:string,name:string>
```

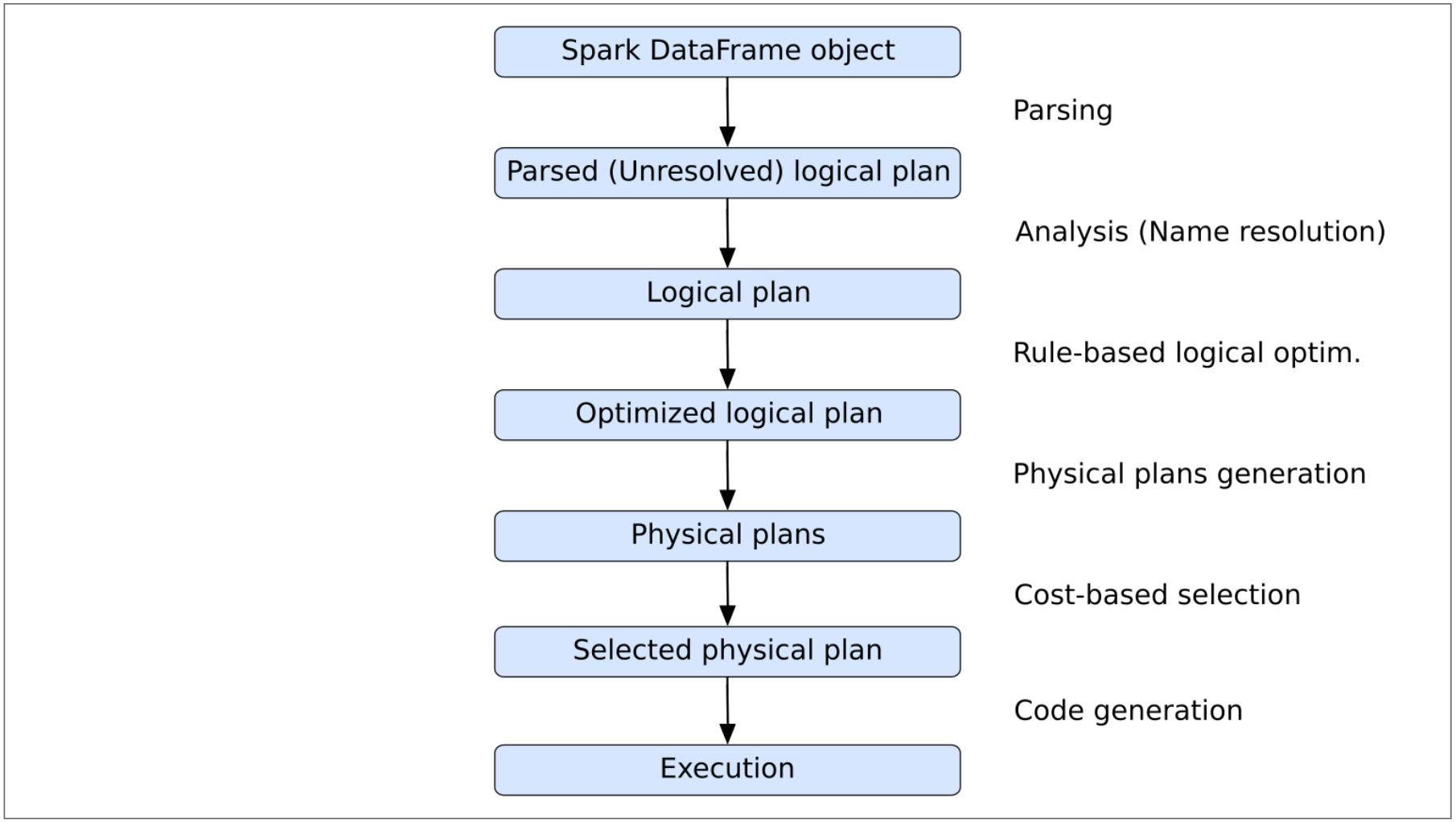
4.5 Spark SQL query optimizer: Catalyst

- Spark lazy evaluation enable a global optimization of the computations to be performed.
- Optimizations are performed on Spark DataFrames (not RDDs) by its embedded query optimizer: Catalyst
- Catalyst was introduced in Spark 2.0 and is still a major source of new developments and performance gains.
- Philosophy:
    - Only load the data you need.
    - Minimize costly data shuffles.
- Operations to be performed are represented using a (directed acyclic) graph (DAG) which is used as input by the optimizer.
- Performance gains can be very substantial.

4.5 Spark SQL query optimizer: Catalyst



4.5 Spark SQL query optimizer: Catalyst



4.5 Spark SQL query optimizer: Catalyst

In [15]:

```
texas_delays.explain(extended=True)
```

```
== Parsed Logical Plan ==
'Filter ('state = TX)
+- Aggregate [state#454, airport_name#460], [state#45
4, airport_name#460, max(DepDelay#247) AS max_dep_del
ay#518]
   +- Join LeftOuter, (OriginAirportID#245 = airport_
id#452)
      :- Relation[DayofMonth#242,DayOfWeek#243,Carrie
r#244,OriginAirportID#245,DestAirportID#246,DepDelay#
247,ArrDelay#248] parquet
      +- Project [airport_id#452, state#454, name#455
AS airport_name#460]
         +- Relation[airport_id#452,city#453,state#45
4,name#455] csv

== Analyzed Logical Plan ==
state: string, airport_name: string, max_dep_delay: i
nt
```

```
Filter (state#454 = TX)
+- Aggregate [state#454, airport_name#460], [state#45
4, airport_name#460, max(DepDelay#247) AS max_dep_del
ay#518]
   +- Join LeftOuter, (OriginAirportID#245 = airport_
id#452)
      :- Relation[DayofMonth#242,DayOfWeek#243,Carrie
r#244,OriginAirportID#245,DestAirportID#246,DepDelay#
247,ArrDelay#248] parquet
      +- Project [airport_id#452, state#454, name#455
AS airport_name#460]
         +- Relation[airport_id#452,city#453,state#45
4,name#455] csv

== Optimized Logical Plan ==
Aggregate [state#454, airport_name#460], [state#454,
airport_name#460, max(DepDelay#247) AS max_dep_delay#
518]
+- Project [DepDelay#247, state#454, airport_name#46
0]
   +- Join Inner, (OriginAirportID#245 = airport_id#4
52)
      :- Project [OriginAirportID#245, DepDelay#247]
      :  +- Filter isnotnull(OriginAirportID#245)
      :     +- InMemoryRelation [DayofMonth#242, DayO
```

```
fWeek#243, Carrier#244, OriginAirportID#245, DestAirp
ortID#246, DepDelay#247, ArrDelay#248], StorageLevel
(disk, memory, 1 replicas)
      :           +- *(1) FileScan parquet [DayofMont
h#85,DayOfWeek#86,Carrier#87,OriginAirportID#88,DestA
irportID#89,DepDelay#90,ArrDelay#91] Batched: true, F
ormat: Parquet, Location: InMemoryFileIndex[file:/hom
e/pierre/Documents/notes/spark/talk/data/flights.parq
uet], PartitionFilters: [], PushedFilters: [], ReadSc
hema: struct<DayofMonth:int,DayOfWeek:int,Carrier:str
ing,OriginAirportID:int,DestAirportID:int,DepDelay...
      +- Project [airport_id#452, state#454, name#455
AS airport_name#460]
         +- Filter ((isnotnull(state#454) && (state#4
54 = TX)) && isnotnull(airport_id#452))
            +- Relation[airport_id#452,city#453,state
#454,name#455] csv

== Physical Plan ==
*(3) HashAggregate(keys=[state#454, airport_name#46
0], functions=[max(DepDelay#247)], output=[state#454,
airport_name#460, max_dep_delay#518])
+- Exchange hashpartitioning(state#454, airport_name#
460, 200)
   +- *(2) HashAggregate(keys=[state#454, airport_nam
```

```
e#460], functions=[partial_max(DepDelay#247)], output
=[state#454, airport_name#460, max#569])
      +- *(2) Project [DepDelay#247, state#454, airpo
rt_name#460]
         +- *(2) BroadcastHashJoin [OriginAirportID#2
45], [airport_id#452], Inner, BuildRight
            :- *(2) Filter isnotnull(OriginAirportID#
245)
            :  +- InMemoryTableScan [OriginAirportID#
245, DepDelay#247], [isnotnull(OriginAirportID#245)]
            :        +- InMemoryRelation [DayofMonth#
242, DayOfWeek#243, Carrier#244, OriginAirportID#245,
DestAirportID#246, DepDelay#247, ArrDelay#248], Stora
geLevel(disk, memory, 1 replicas)
            :              +- *(1) FileScan parquet
[DayofMonth#85,DayOfWeek#86,Carrier#87,OriginAirportI
D#88,DestAirportID#89,DepDelay#90,ArrDelay#91] Batche
d: true, Format: Parquet, Location: InMemoryFileIndex
[file:/home/pierre/Documents/notes/spark/talk/data/fl
ights.parquet], PartitionFilters: [], PushedFilters:
[], ReadSchema: struct<DayofMonth:int,DayOfWeek:int,C
arrier:string,OriginAirportID:int,DestAirportID:int,D
epDelay...
            +- BroadcastExchange HashedRelationBroadc
astMode(List(cast(input[0, int, true] as bigint)))
```

```
                  +- *(1) Project [airport_id#452, state
#454, name#455 AS airport_name#460]
                     +- *(1) Filter ((isnotnull(state#45
4) && (state#454 = TX)) && isnotnull(airport_id#452))
                        +- *(1) FileScan csv [airport_id
#452,state#454,name#455] Batched: false, Format: CSV,
Location: InMemoryFileIndex[file:/home/pierre/Documen
ts/notes/spark/talk/data/airports.csv], PartitionFilt
ers: [], PushedFilters: [IsNotNull(state), EqualTo(st
ate,TX), IsNotNull(airport_id)], ReadSchema: struct<a
irport_id:int,state:string,name:string>
```

5. Working with Spark: useful things you should know about

5.1 Storing data on HDFS: Use the Apache Parquet format

- Binary columnar encoded compressed format with embedded metadata (like schema).
- Spark default file format, supported by many applications within the Hadoop ecosystem.
- Spark Parquet reader leverages Parquet features for very high read performance: filter pushdown, partition pruning, metadata filtering.
- Writing Parquet files:
    - Remember that Spark creates one file per partition: beware of not writing many small files on HDFS (HDFS small files problem)
    - Leverage features such as Parquet partitioning, bucketing and sorting for faster reads.

5.2 Caching

- In complex workflows, appropriatly caching or checkpointing intermediate data avoids recomputing that data more than once.
- See the `cache`, `persist` DataFrame methods.
- **Warning:** `cache` and `persist` are transformations and not actions.

5.3 Beware of unbalanced partitioning

- If data in not evenly spread accross partitions (data skew), some tasks may take much longer than others forcing dependent subsequent tasks to wait.
- Reminder: When is the data (re)partioned ?
    - After reading.
    - After shuffling.
- Solution: the user forces a repartitioning using an appropriate partitioning key using the `repartition` method.
- **Warning:**
    - This will cost a full shuffle.
    - Find a balance between too many partitions (book-keeping overhead) and too few partitions (executors will suffer or even crash).
- Spark 3.0 Adaptative Query Execution introduces an automatic response to this problem.
- **Warning** on `collect`/`toPandas`, `repartition(1)` and `coalesce(1)`: all data ends up on the same JVM and can kill it.

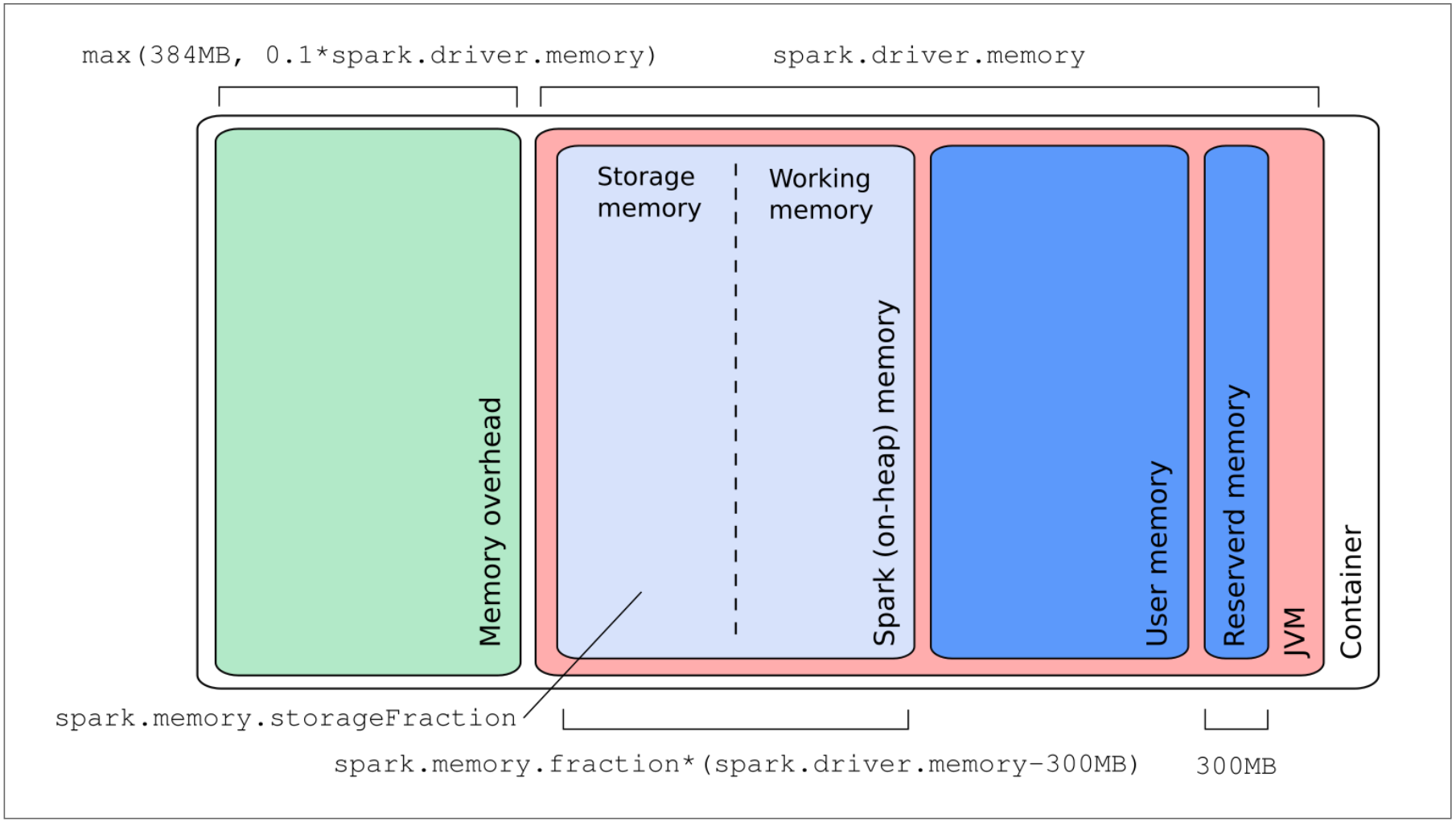
5.4 Minimize wide operations

- Distributed computing wide operations (aggregations, join, sort) have high performance costs due to shuffling (disk and network I/O).
- Spark Catalyst optimizes as much as it can but:
    - Do not use these operations (Ex: cosmetic `orderBy`) if you can find a workaround.
    - Catalyst may miss possible optimizations (especially when using window functions) but is constantly improved.
- **Notice**: Sorting the entire dataset has little meaning in distributed computing and can lead to missleading results and degrade performance (sorting requires to shuffle):
    - They are very few situation outside from window functions and data writes where the user needs to call `orderBy` or any equivalent operation.
    - Ex: First row of each group.

5.5 Sizing you application

- People's default behavior when memory issues arise: "If it does not work, just keep throwing more memory in".
- Instead, when your job keeps crashing for memory issues:
    - Increase parallelism: more executors, more cores per executors (total number of cores = total number of Spark workers = total number of simultaneously executed tasks).
    - **At the executor level: balance RAM and the number of cores**.
    - Check your data partitioning: data skew ? too many or too few partitions: change the partitioning settings and/or adapt your code.
- Key advice: Cache your data and go to the Spark UI to assess its size once in memory.
- Key ratio: memory per core.
- Application sizing advice:
    - Data size => Partition count => Spark worker count (cores and executors).
    - Spark worker count and partition size => Appropriate amount of memory.
- Great variability in input data size: dynamic allocation of executors

5.5 Sizing you application: Spark memory layout



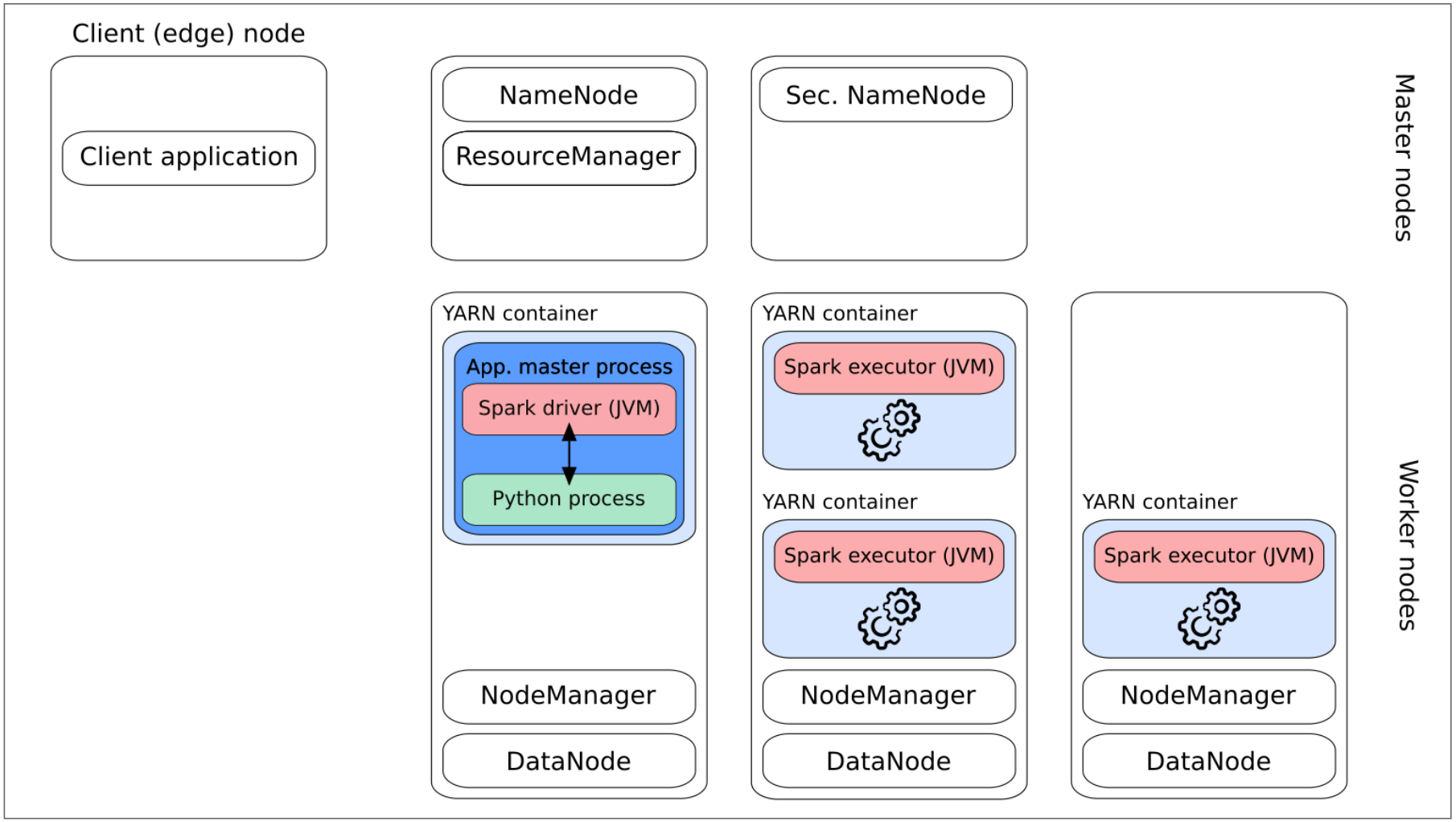
5.6 Python on Spark (1/2)

- Spark is written in Scala so where is my Python executed ?
    - You Python application code is executed by a Python interpreter **next to the Spark driver's JVM**.
    - Spark executors only run Python code when you resort to Spark Python user defined functions (UDFs).
- On Python UDFs:
    - Python UDFs can greatly hamper performance (serialization/deserialization costs), use them when you have no alternative.
    - Major field of improvements since Spark 2.3 (`pandas` UDFs, `pyarrow`).
- Which Python interpreter is being used? Set using `spark.pyspark.python`/`spark.pyspark.driver.python` or the corresponding `PYSPARK_PYTHON`/`PYSPARK_DRIVER_PYTHON` environment variables.

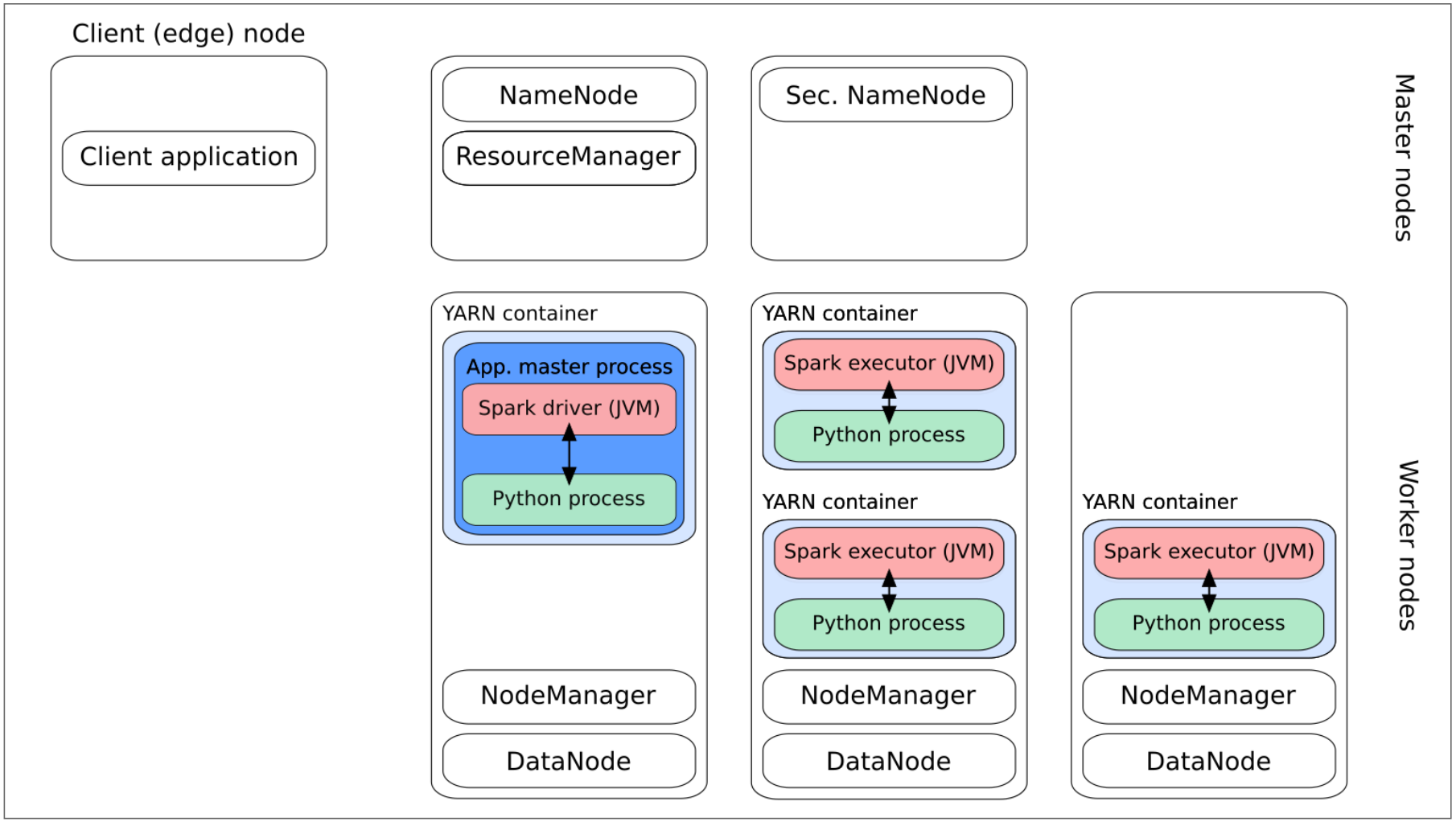
5.6 Python on Spark (2/2)

- What about my dependencies?
    - They need to be installed on each node of the cluster.
    - Depending on your type of cluster, you can use Docker images to manage your Python environment.
    - You can upload additional dependencies at application submission time using `spark.submit.pyFiles` / `--py-files` which are automatically added to `PYTHONPATH`.
- Make sure to allocate enough memory for the Python interpreter especially in the driver's container if you collect all your data in a `pandas.DataFrame`.

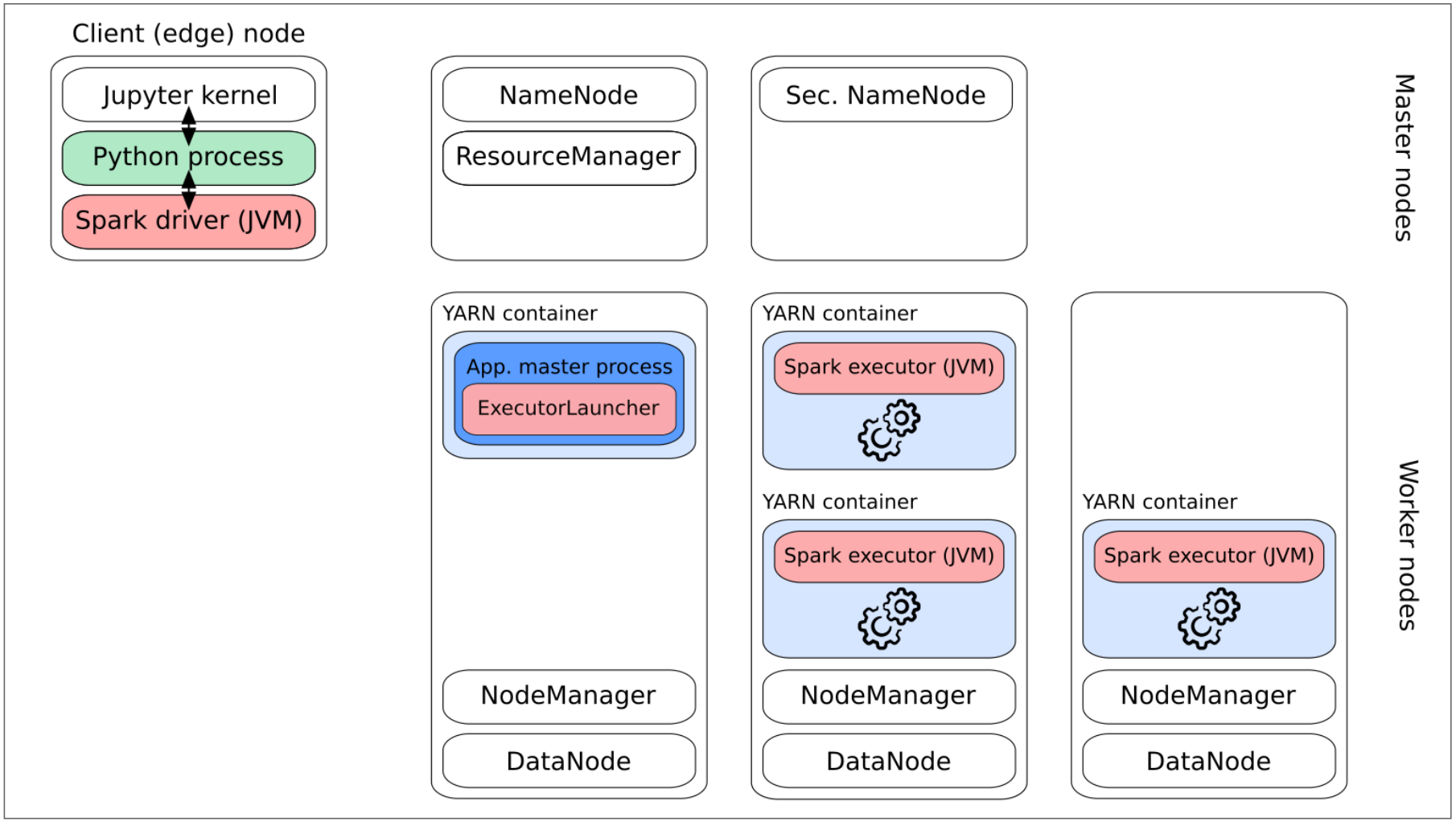
5.6 `pyspark` YARN application (cluster mode)



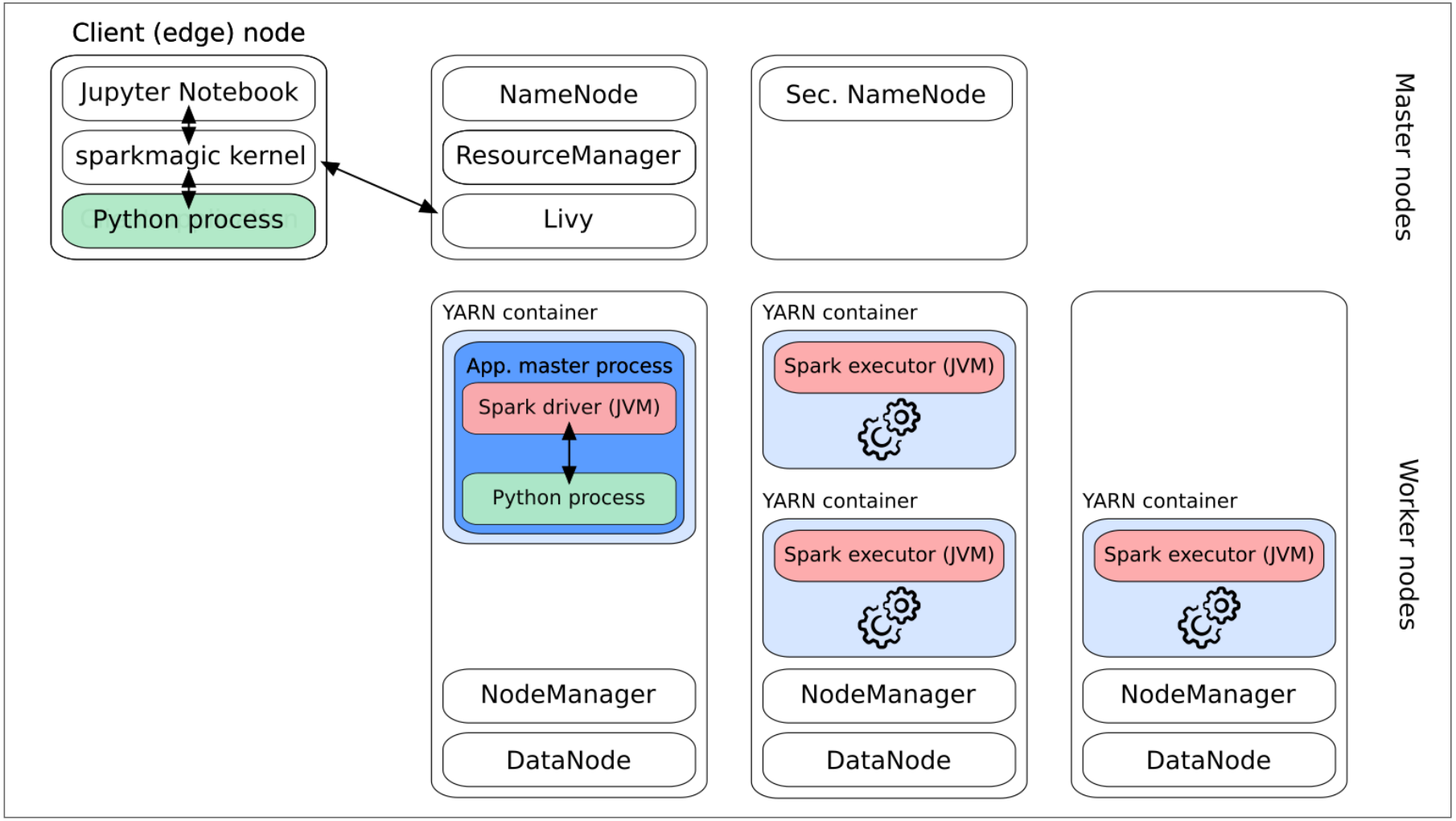
`pyspark` YARN application with Python UDFs (cluster mode)



5.6 Using `pyspark` with Jupyter notebooks



5.6 Using `pyspark` with Jupyter notebooks & Apache Livy



In [16]:

```
spark.stop()
```