Data sources and sinks in Spark

(CZ, chapter 9 + a little 20)

Data sources in Spark

- from Python objects:
  - df = spark.createDataFrame(list_of_tuples, list_of_col_names)
- from file(s):
  - df = spark.read.format(format).load(location)
  - core data formats:
    text, CSV, JSON lines, JDBC/ODBC, Parquet, ORC, AVRO
  - community-maintained sources:
    Cassandra, HBase, MongoDB, XML,
    also Bahir for Spark 2.x (but not structured, and Twitter API v1)
- from streams:
  - streaming_df = spark.readStream.format(format).load(location)
  - core streaming sources: socket, folder, HDFS, Kafka

spark.read

- df = spark.read.format(format).load(location)
- Generic format:

```
DataFrameReader \                    # i.e., spark.read
    .format(…) \                     # optional (default is Parquet)
    .option("key", "value") \        # some may be mandatory
    .schema(...) \                   # optional (schema inference on read)
    .load()
```

- Options depend on the format,
- Common options:

```
    .options('mode', …)              # permissive, dropMalformed, failFast
    .options('path', ...path...)     # path to file or folder
```

Spark schemas

- Schema:
  - schema inference, “schema-on-read”
  - schema from source (e.g., in Parquet file)
  - user-defined schema: df.schema(…schema...)
  - from pyspark.sql.type import StructType, StringType, LongType, ...
    ```
    schema = StructType() \
                .add('col_1_name', spark_type, is_null_allowed) \
                .add('col_2_name', spark_type, is_null_allowed) \
                ...
    ```
  - from pyspark.sql.type import StructType, StructField, StringType, …
    ```
    schema = StructType([
                StructField('col_1_name', StringType(), True),
                StructField('col_2_name', LongType(), False),
                ...
            ])
    ```

Parquet format

- Parquet:
  - an open source column-oriented data store
  - provides a variety of storage optimizations
    - suited for analytics workloads
  - provides columnar compression
    - saves storage space
    - allows for reading individual columns instead of entire files
  - Apache Spark's default file format
  - will always be more efficient than JSON or CSV
  - supports complex types (i.e., a column of arrays)

Data sinks in Spark

- to Python objects:
  - localCollection = df.collect()   # df.take(*n*), df.first()
  - iterator = df.toLocalIterator()
  - conversions: df.toPandas(), df.rdd, etc.
- to files:
  - df.write.format(format).save(folder_name)   # writes to a folder
- to streams:
  - streaming_df.writeStream. … .start()   # the usual action
    streaming_df.awaitTermination()
  - core streaming sinks:
    socket, console, memory, .foreach()-action, folder, HDFS, Kafka

spark.write

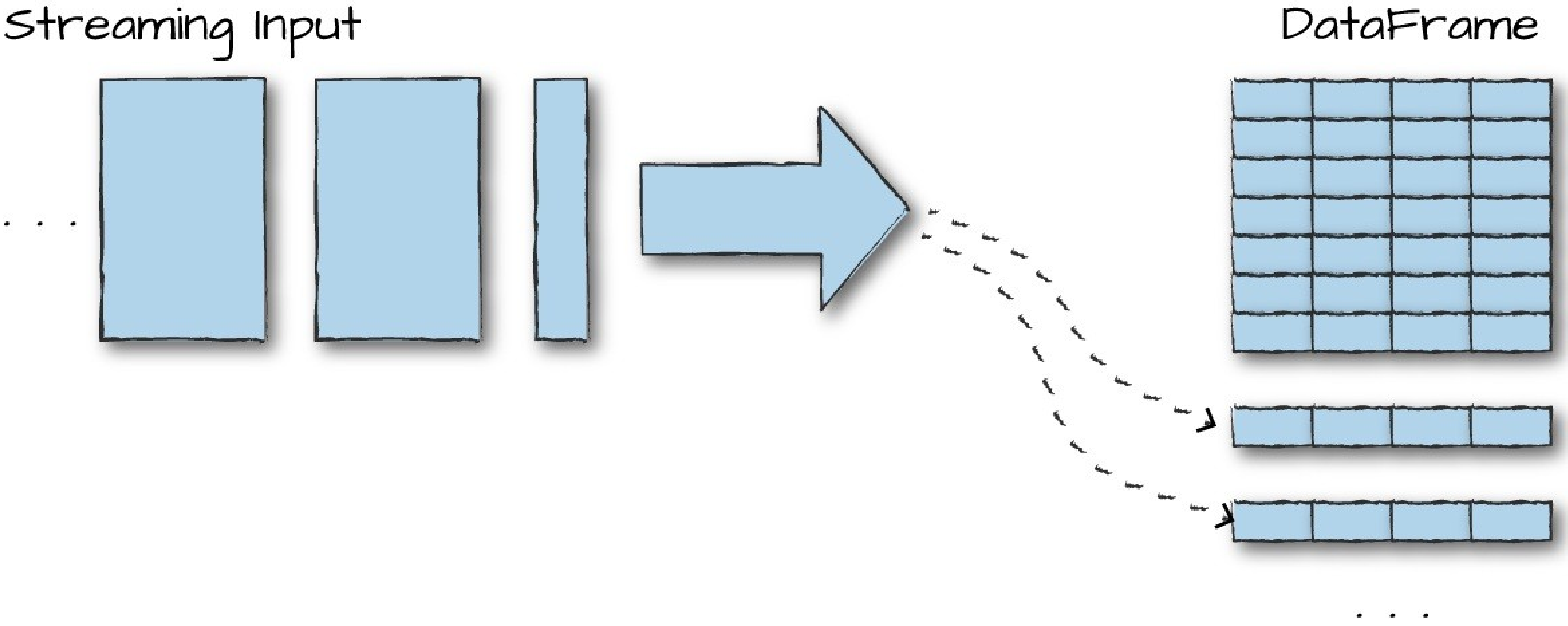
- df.write.format(format).save(folder_name) # writes a folder of files
- Generic format:

```
DataFrameWriter \                          # i.e., df.read
        .format(…) \
        .option(…) \
        .partitionBy(…) \                  # save to sub-folder per column value
        .bucketBy(…) \                     # split into files by column value
        .sortBy(…) \
        .save()
```

- Options again depend on the format
- Common options:

```
        .options('mode', …)                # append, overwrite, errorIfExists, ignore
        .options('path', ...path…)         # path to folder
```

Streaming Spark

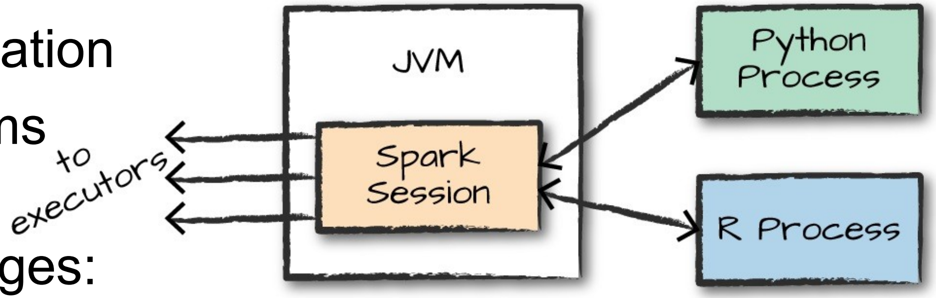


Spark execution

(CZ, chapter 9 and earlier)

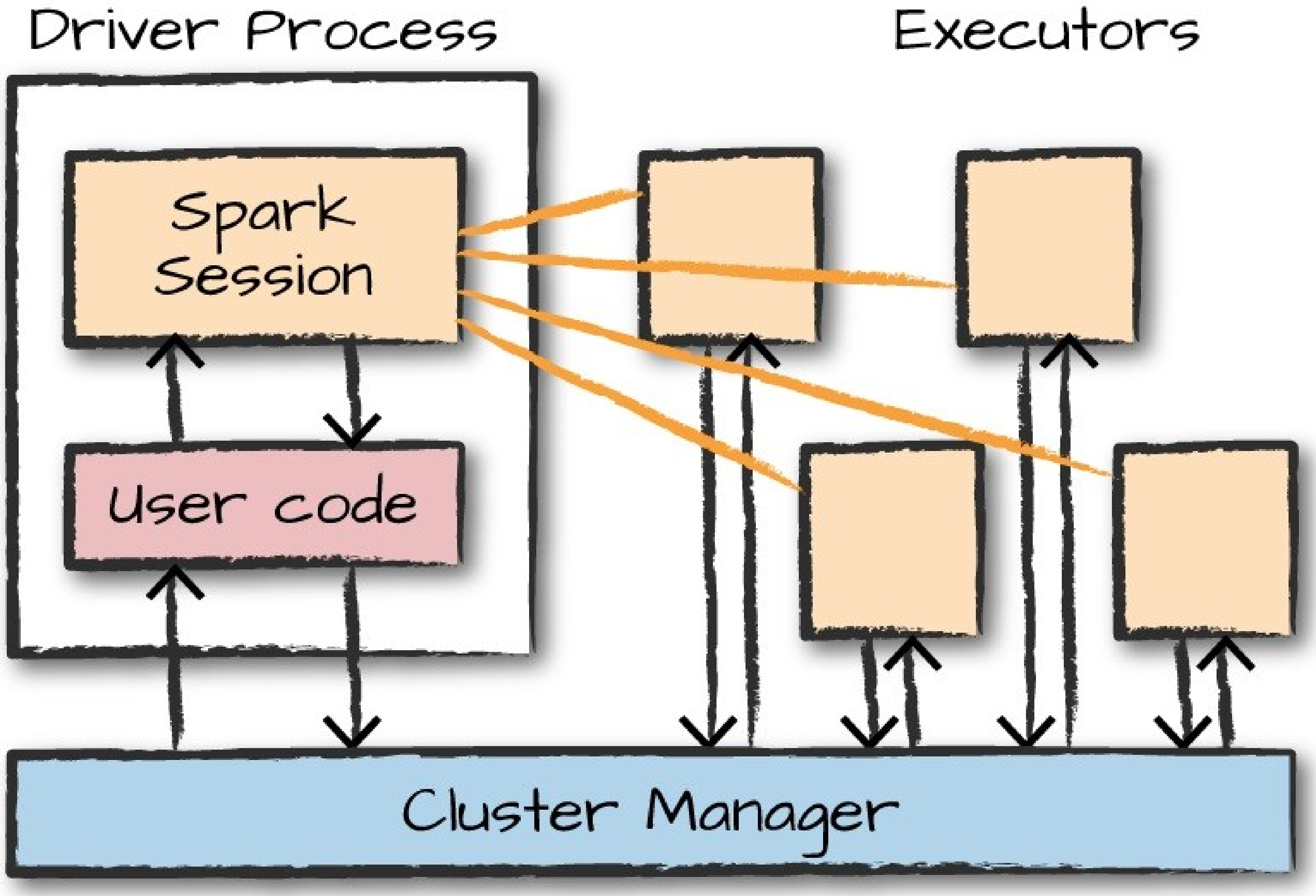
Applications, drivers and executors

- Spark Application (“user code”):
  - a *driver process* and (one or) many *executor processes*
- Driver process:
  - “the heart of the Spark Application” – runs the main() function
  - one-to-one with the SparkSession object
  - maintains information about the application
  - responds to input from users / programs
  - compiles, interprets, and translates Spark code written in different languages: Java, Scala, …, Python, R, SQL, ...
  - analyses, distributes, and schedules work to the executors
  - interfaces with the *cluster manager* to launch executors



Applications, drivers and executors

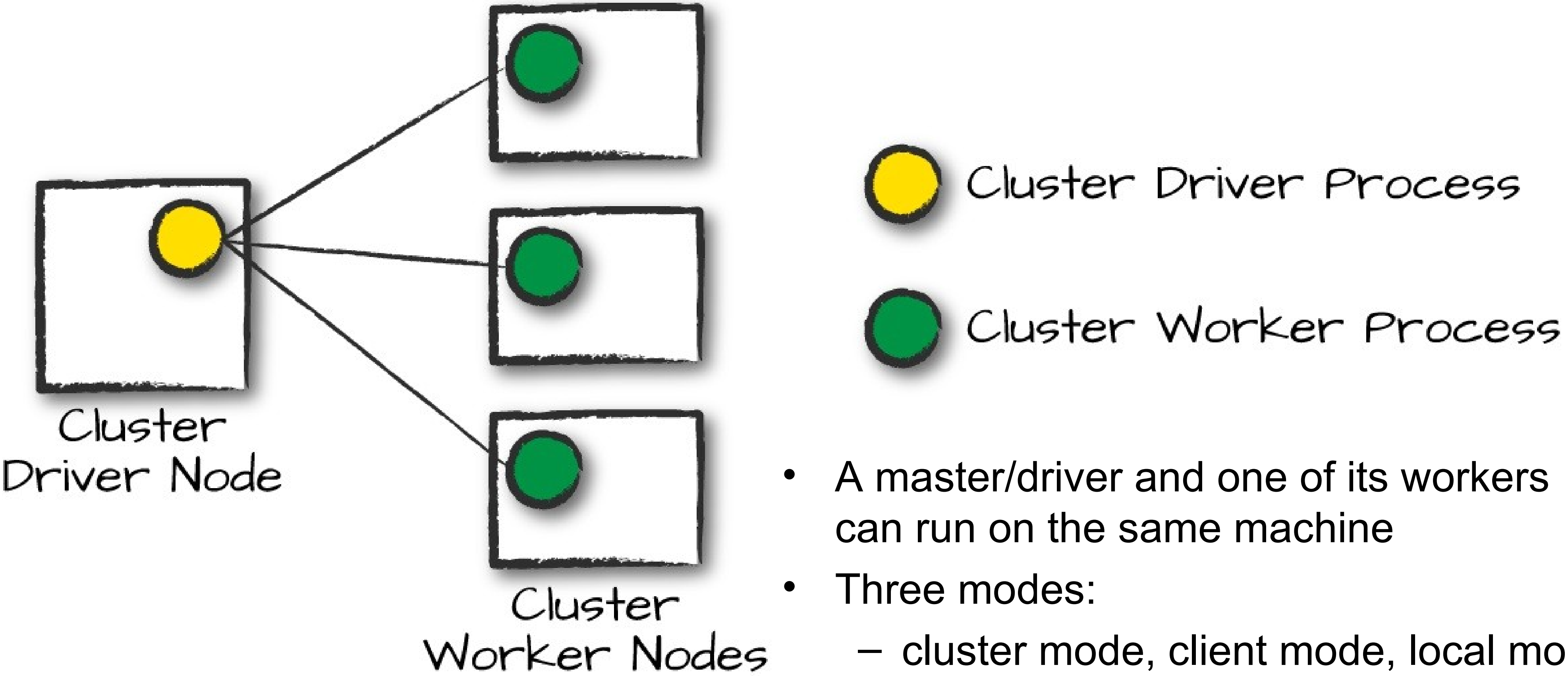
- Spark Application ("user code"):
  - a *driver process* and (one or) many *executor processes*
- Executor processes:
  - responsible for actually carrying out the work assigned by the driver
  - executing code assigned to it by the driver
    - mostly run Java bytecode
      - compiled from Java, Scala, ...
      - perhaps from Python, R, …
    - can also run other code
      - but that can make the job harder for the cluster manager
  - reporting the state of its computation back to the driver process
  - an executor belongs to only one driver


Driver Process
Executors
Spark Session
User code
Cluster Manager

Master/driver and worker nodes

- Cluster manager:
  - keeps track of the resources available
  - one cluster *master/driver* and one (or more) *workers (slaves)*
  - each runs on a separate machine (metal or virtual)
    - called a *node*
- Master/driver node:
  - runs process that create and manage worker processes on other nodes
- Worker (slave) nodes:
  - runs processes that do the actual work, for example
  - runs the Spark driver and executor processes
- Available cluster managers:
  - standalone (built-in), Mesos, (Hadoop) YARN, Kubernetes

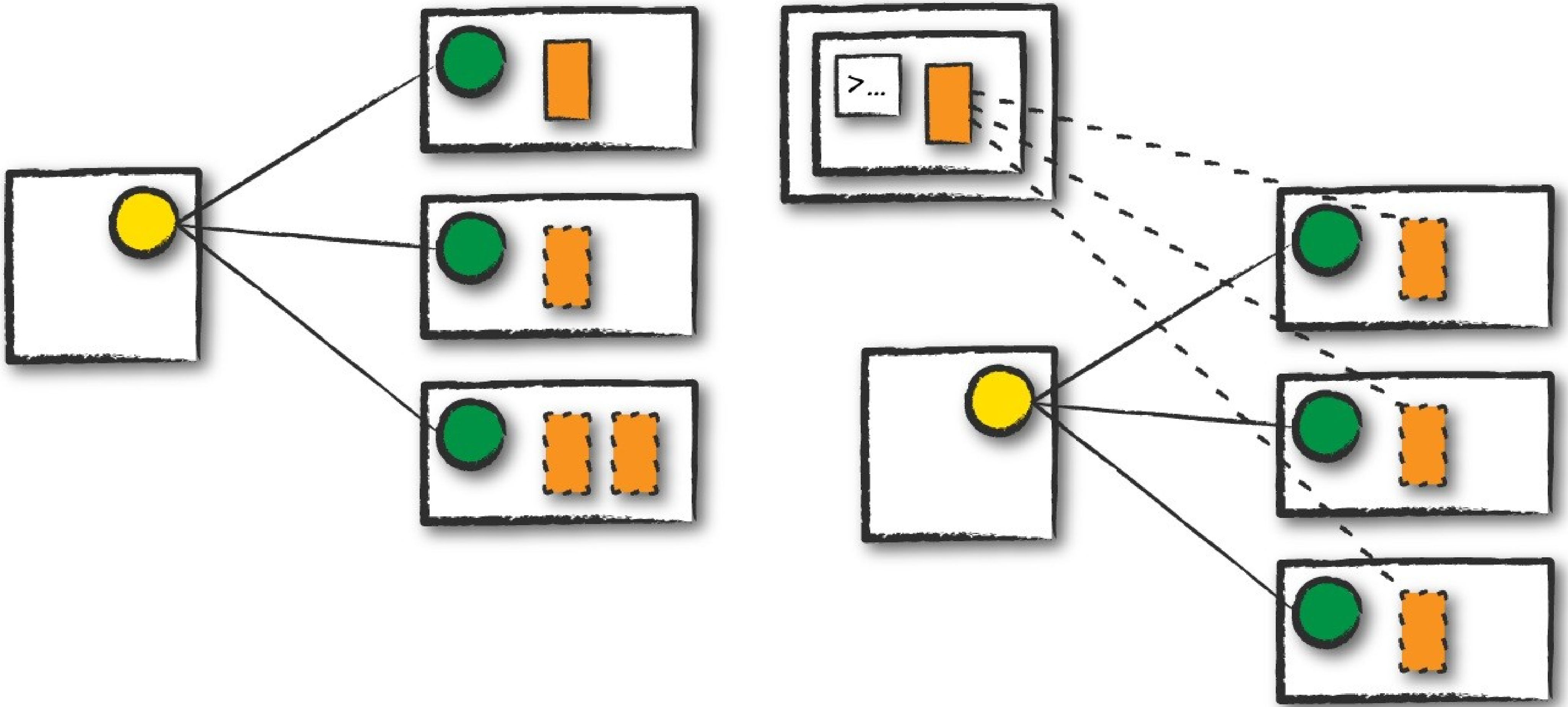
Master/driver and worker processes



- A master/driver and one of its workers can run on the same machine
- Three modes:
  - cluster mode, client mode, local mode

Cluster mode

Client mode



Managers and workers

Logical:	Cluster level	Application level		Detail level
HDFS	Name node			Data node
Hadoop	JobTracker			TaskTracker
YARN	ResourceManager	MRAppManager		TaskManager
Spark		Spark driver		Executor & JVM worker
Cluster	Master/driver node			Worker (slave) node
Machine:	Master/driver machine		Outside	Master/worker machine

Partitions and executors

- DataFrames are divided into partitions
- *Partition:*
  - a collection of rows that sit on the same executor
  - each partition resides in the memory of a single executor
  - allow executors to perform work in parallel
- *Executors:*
  - each executor can (and often should) harbour several partitions
- *Parallelism* is bounded by both
  - number of partitions
  - number of executors

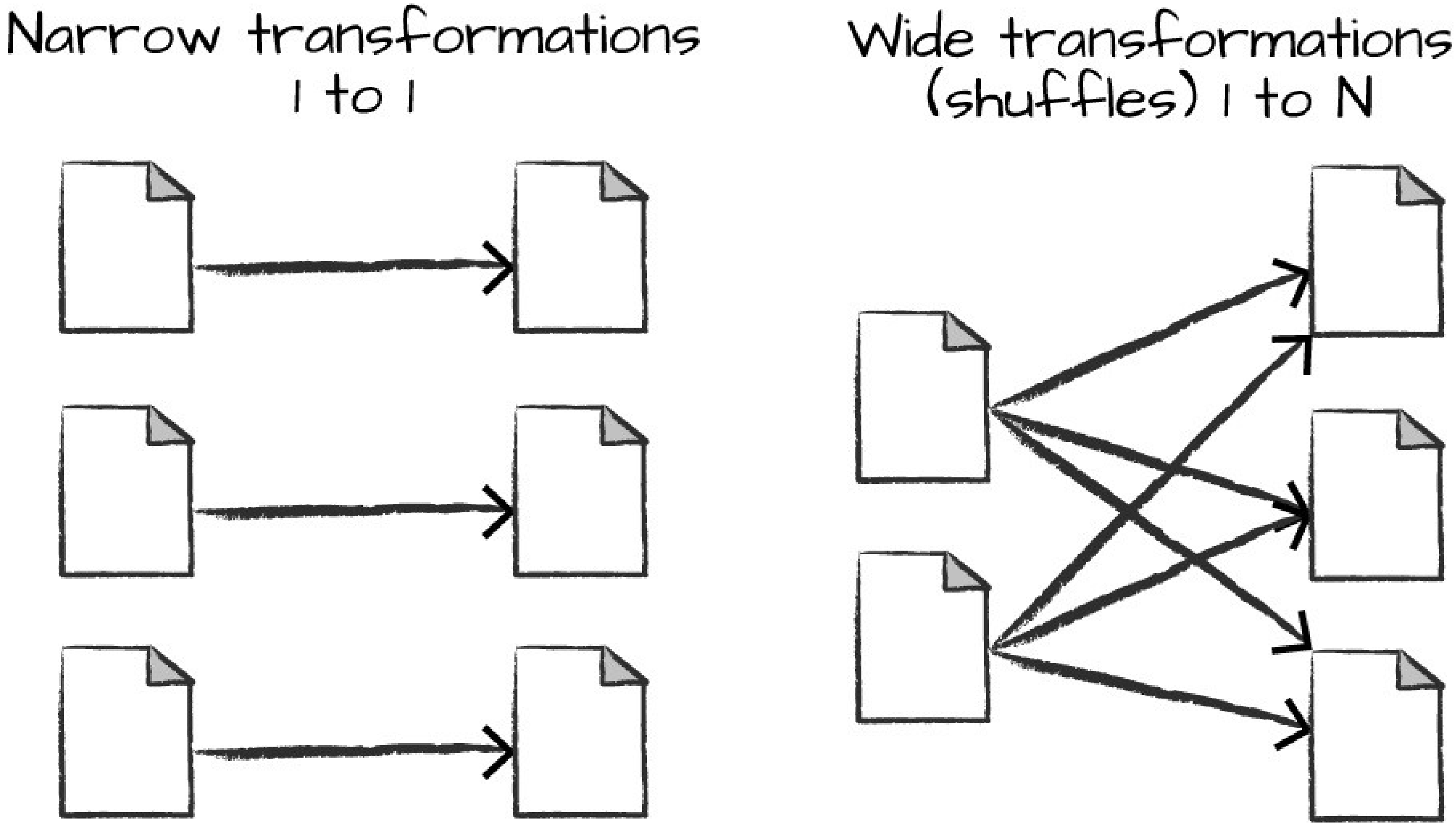
Repartitioning and coalescing

- *Repartition:*
  - control the physical layout of data across the cluster
  - according to either
    - a new number of partitions: df.repartition(*n*)
    - frequently filtered columns: df.repartition(...*col*...)
    - both: df.repartition(*n*, ...*col*...)
  - incurs a full *shuffle* of the data
    - regardless of whether one is necessary
    - (unless you repartition to a smaller number of partitions)
- Coalesce:
  - tries to combine partitions: df.coalesce(*n*)
  - does not incur shuffle

Pipelining and shuffling

- Narrow and wide transformations
  - narrow transformations consist of narrow dependencies
    - each input partition contributes to only one output partition
    - automatic in-memory pipelining (combining transformations)
    - no data exchange between executors
  - wide transformation consist of wide dependencies
    - each input partition contributes to many output partitions
    - data is exchanged between executors
    - shuffling is *disk-based*
      - previous stages do not have to be repeated
      - simpler recovery from executor failure
      - can explicitly write to disk with the *.cache()*-function

Narrow and wide transformations



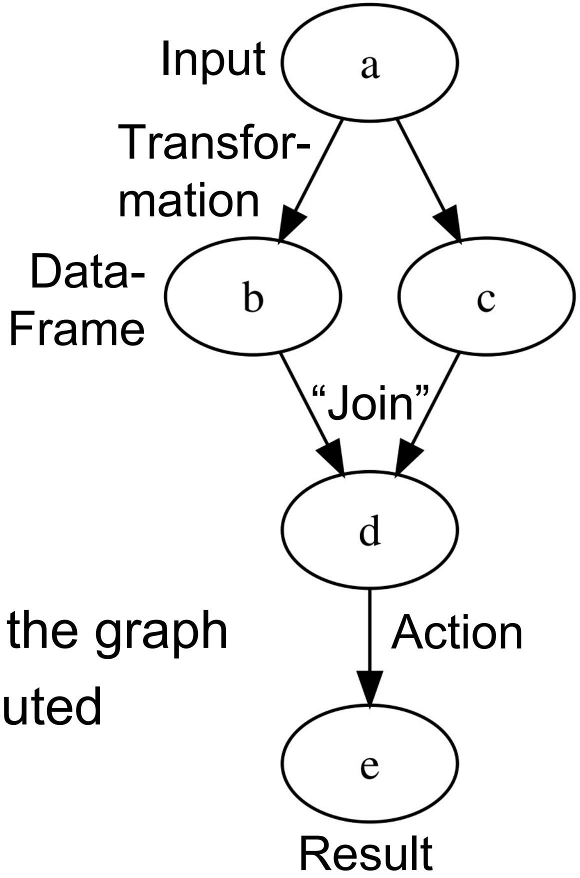
Wide trans-
formations
persist the
data to disk

Jobs, tasks, and stages

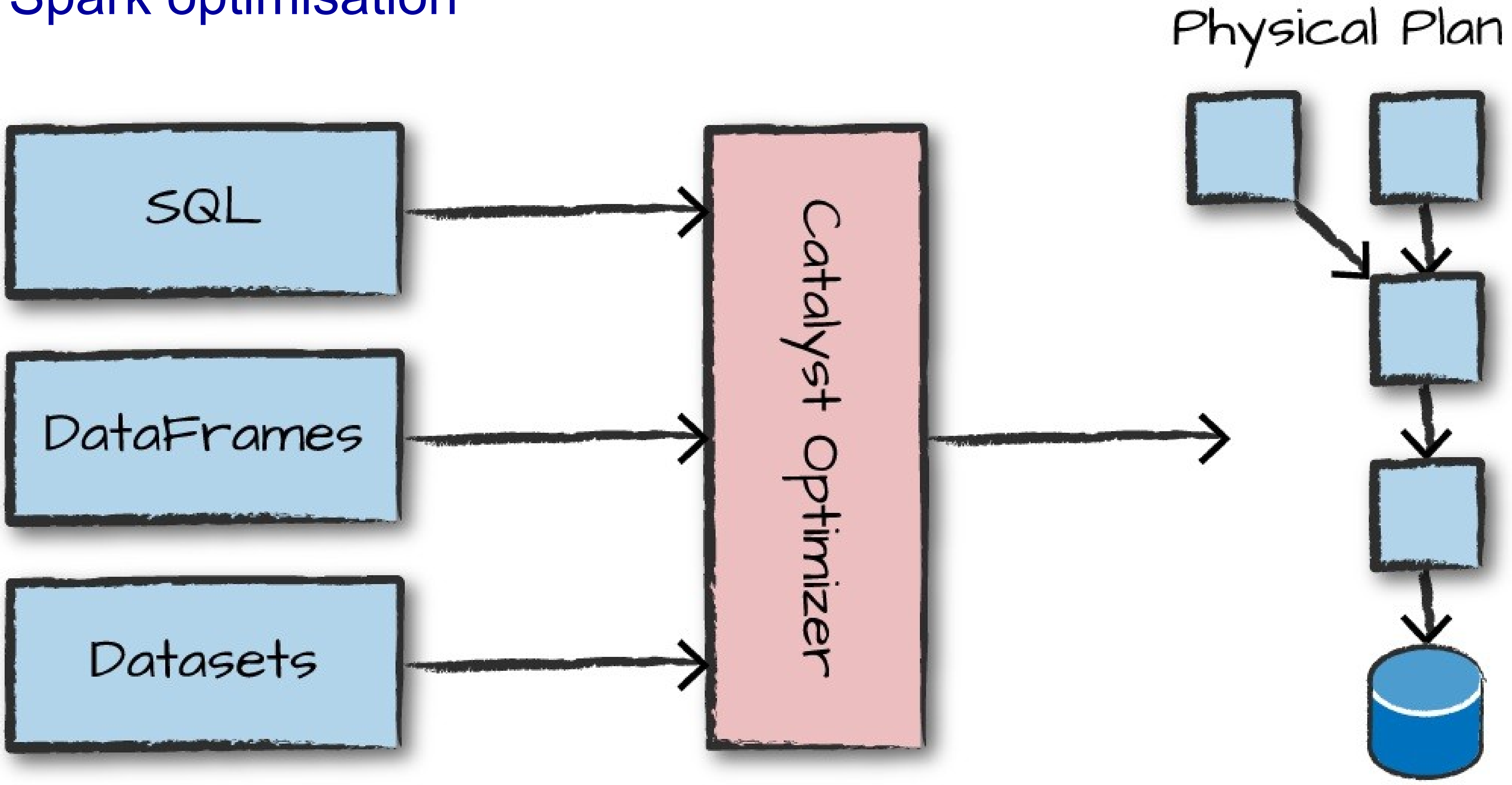
- *Spark application* (“user code”) :
  - a series of *jobs*
  - one job corresponds to one *action*
  - a *job* is a series of *stages*
- *Stage:*
  - the processing that goes on between two shuffle operations
  - a group of *tasks* that can be executed together to compute the same operations (pipeline) on multiple executors
- *Task:*
  - run on a single executor
  - a unit of computation (pipeline) applied to a unit of data (partition)
  - a combination of blocks of data and a set of transformations

Plans and jobs

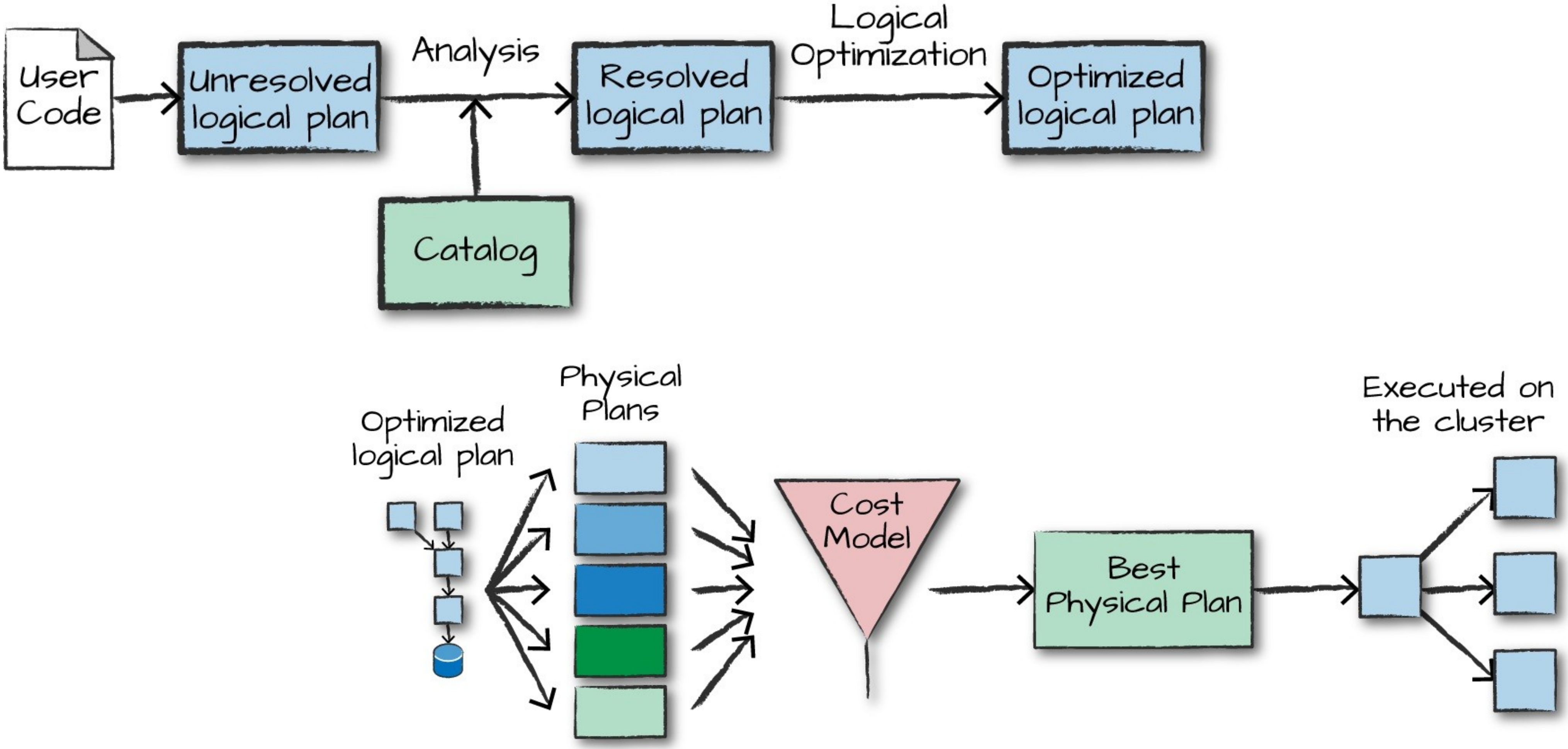
- Spark application ("user code"):
  - a directed acycilic graph of operations
  - transformations and actions are edges
  - DataFrames are internal nodes
  - data sources, results and sinks are leaf nodes
- Lazy evaluation
  - Spark will wait until the very last moment to execute the graph
  - the graph is compiled to an optimised plan and executed
    - only the necessary parts are executed
    - predicate pushdown
    - the .explain() -method



Spark optimisation



Spark optimisation



Spark UI

- Spark user interface (UI)
- Default localhost:4040 (but check the start-up message)
- Displays information
  - the state of your Spark jobs
  - its environment
  - cluster state

What to do
in two weeks?
...and in the meantime :-)

- Exercise 2:
  - streaming data from the Twitter API with tweepy
  - saving to file, sending to socket
  - receiving as streaming Spark
  - combine with your pipeline from exercise 1
- *Project ideas and plans!*
- Essay ideas
- Session 3:
  - streaming Spark
  - Kafka