Data Lakes and Spark

Rahul Dave(@rahuldave), Univ.Ai

Univ.AI

Data Engineering Lifecycle

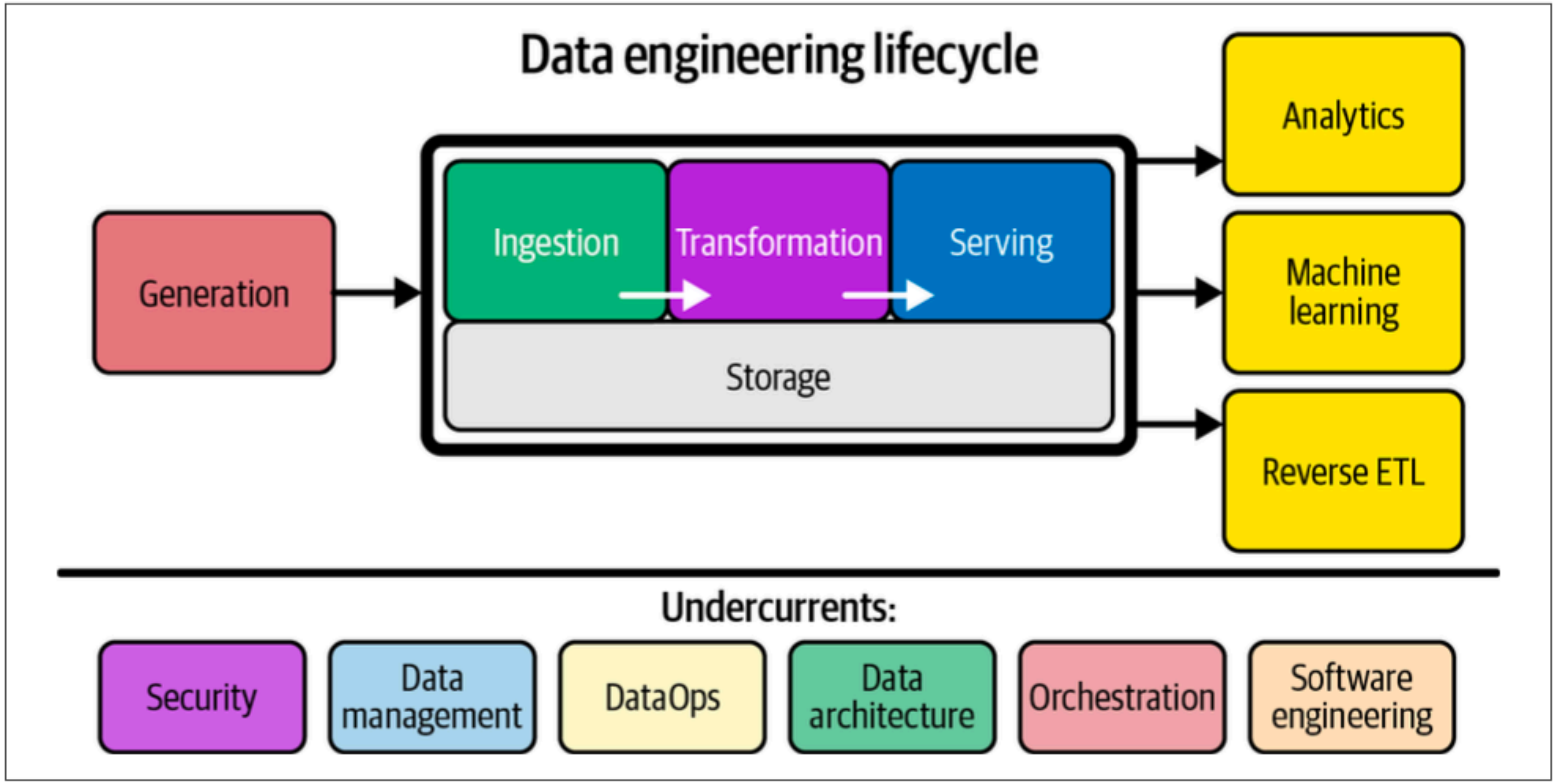
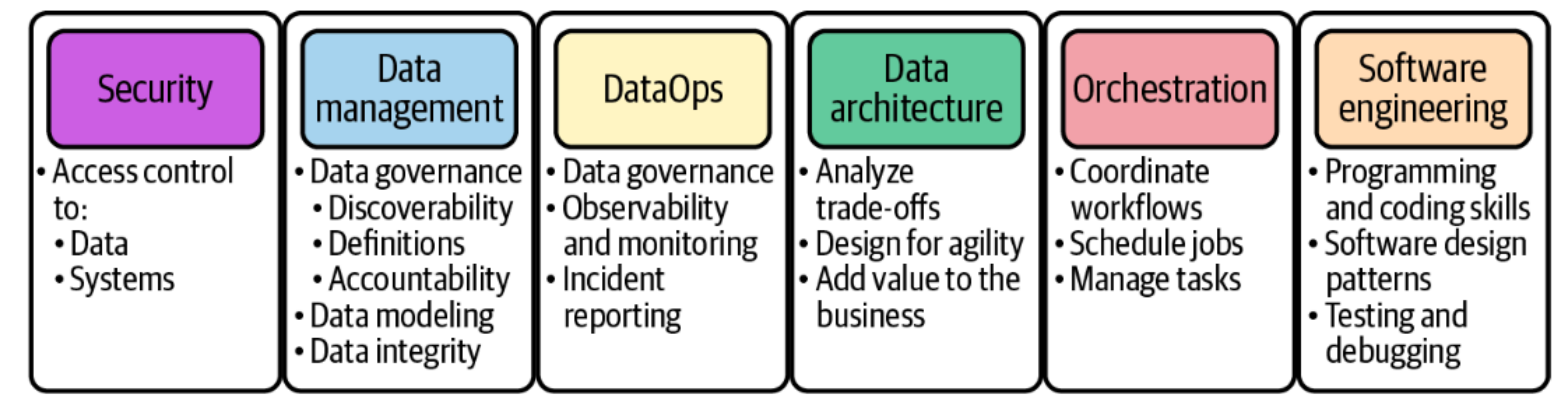


Figure 1-1. The data engineering lifecycle

Undercurrents



Security
- Access control to:
  - Data
  - Systems

Data management
- Data governance
  - Discoverability
  - Definitions
  - Accountability
- Data modeling
- Data integrity

DataOps
- Data governance
- Observability and monitoring
- Incident reporting

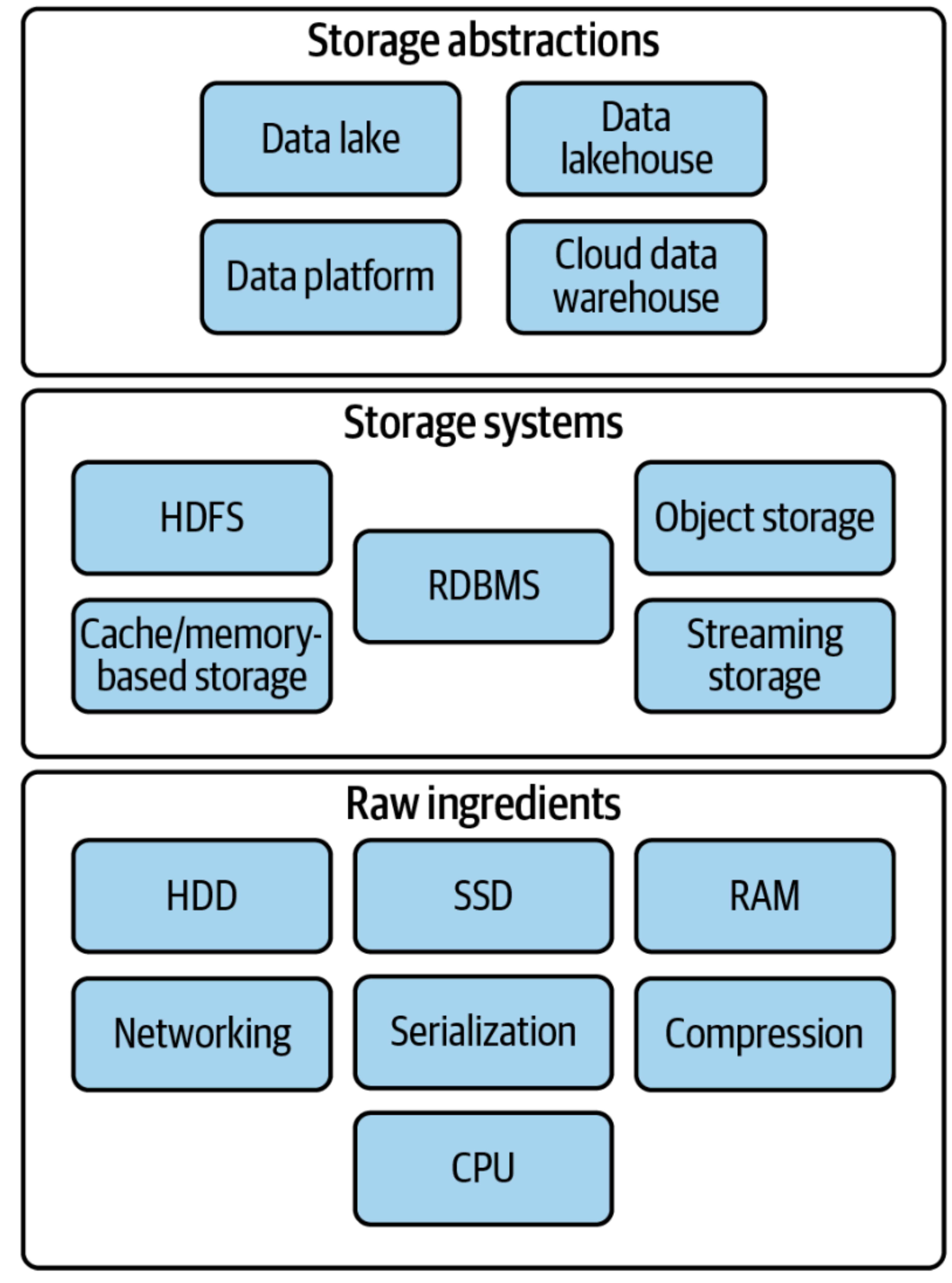
Data architecture
- Analyze trade-offs
- Design for agility
- Add value to the business

Orchestration
- Coordinate workflows
- Schedule jobs
- Manage tasks

Software engineering
- Programming and coding skills
- Software design patterns
- Testing and debugging

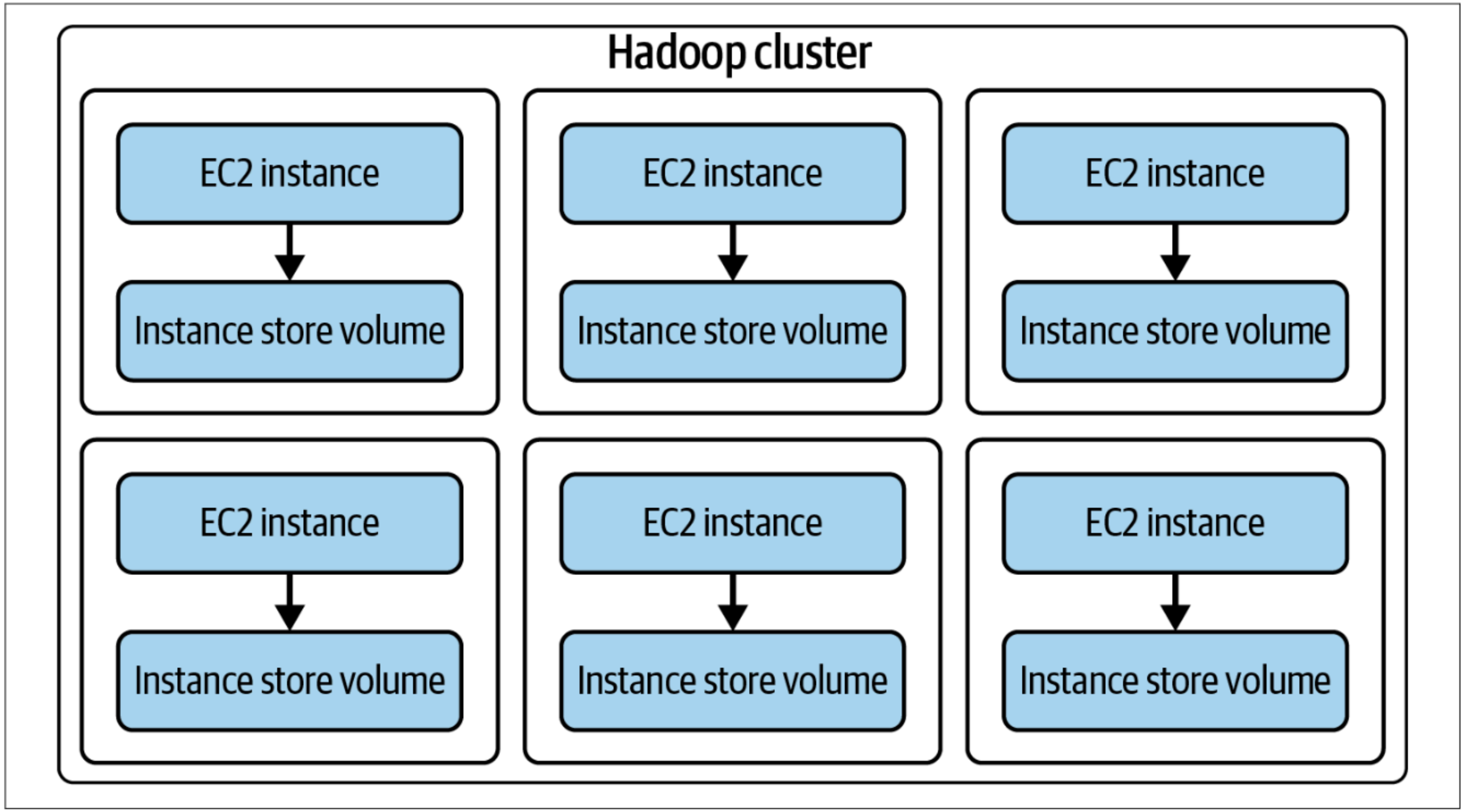
Abstractions

- There are macro storage abstractions
- Which come from combining storage systems and machines
- And the right hardware must be chosen for the right things: for example, slower disk may be ok for object storage
- But hdfs on an EMR cluster ought to use SSDs on the per-machine storage (EBS).



Univ.AI

Distributed Computing



To SQL or Not to SQL(Spark).
Thats the question...

- With object storage and SQL-on-file as base layer it is not entirely clear what is in the database and what is not

- There are two kinds of ELT: into warehouse, and into lake and (possibly) then with some T into warehouse (a kind of ETL).

- Dont focus on ETL/ELT, focus on the needs of the data

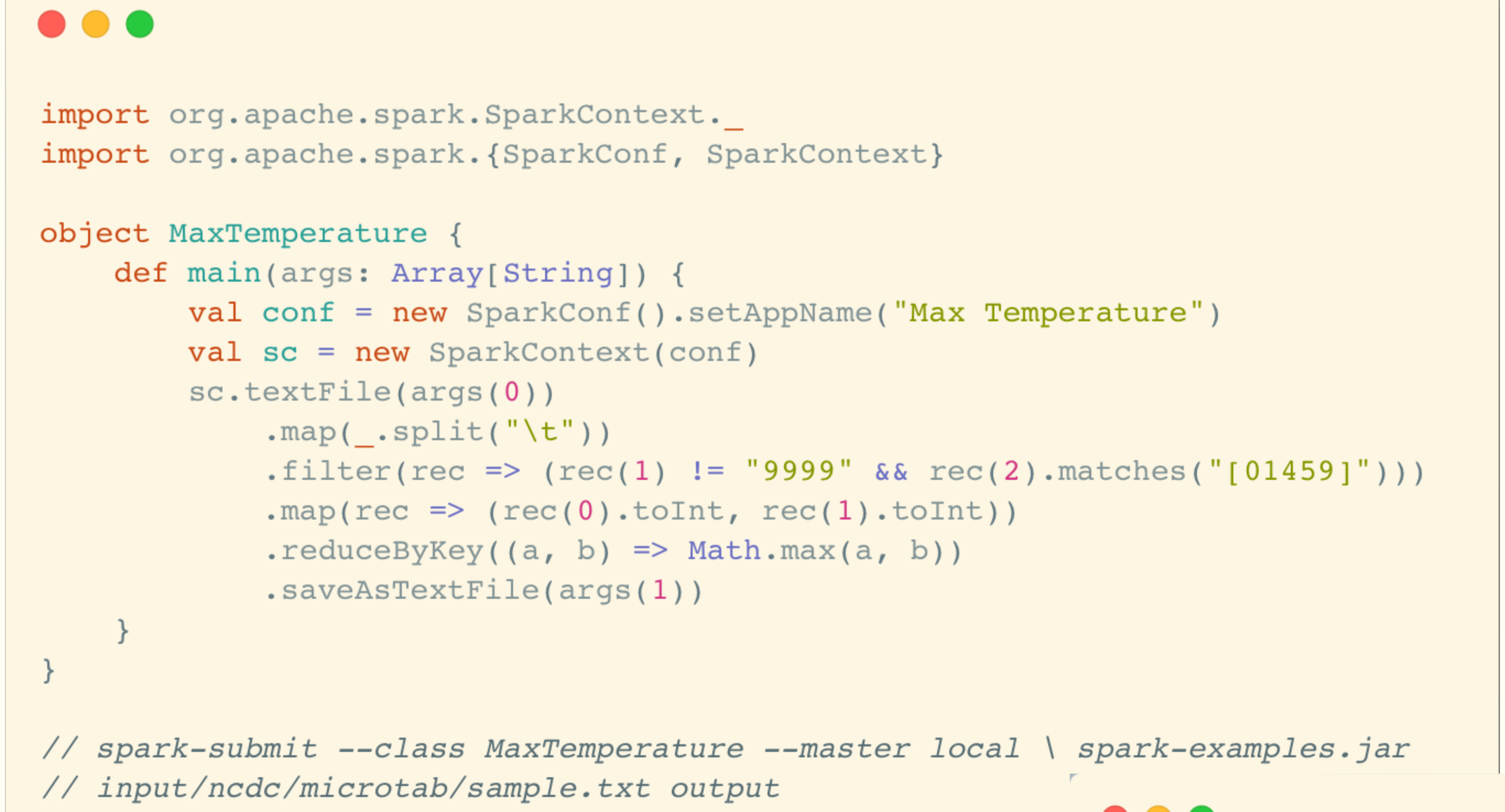
- We should think about SQL workflows vs non-SQL workflows instead. What transformations would be hard to express with SQL? There you want to use Spark and PySpark.

Spark (non sql) use cases

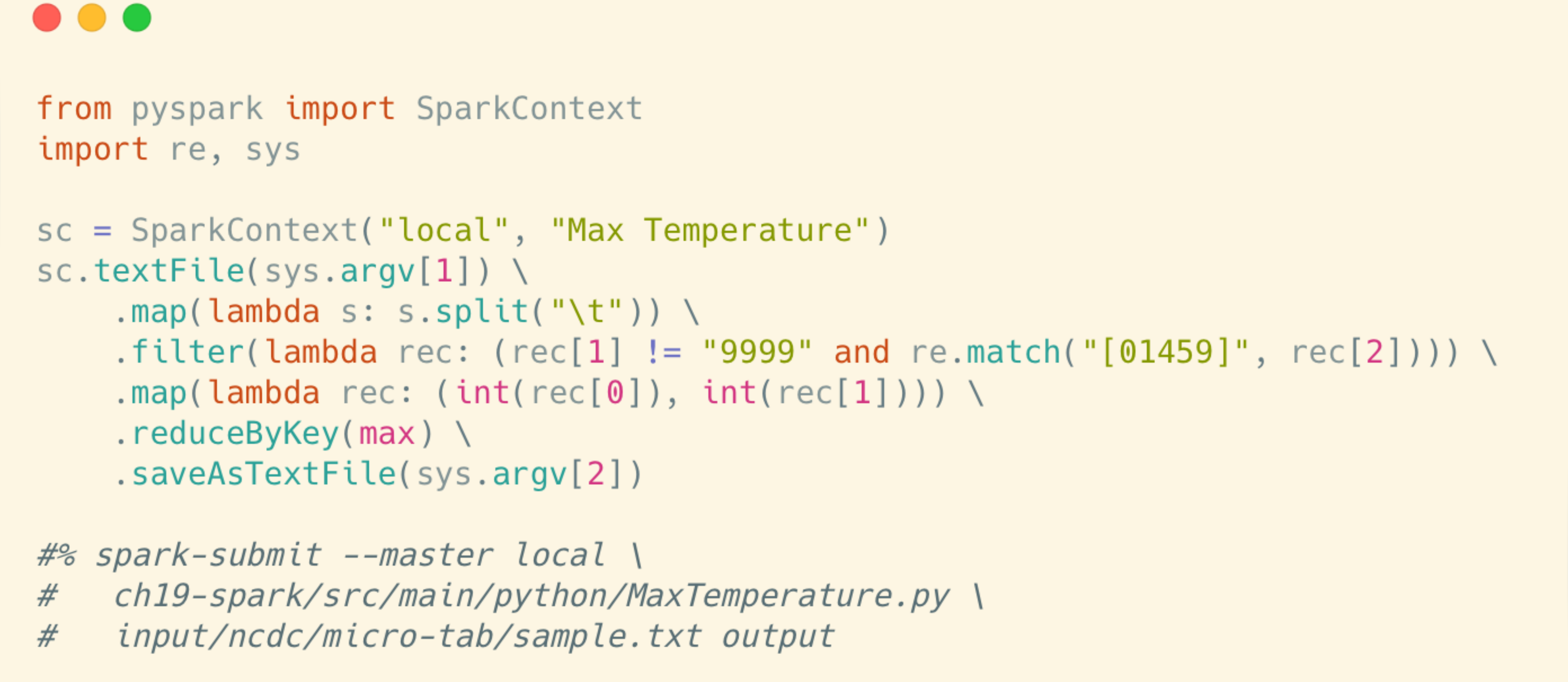
- word stemming
- algorithmic ml/transformations resulting in new data
- construction of re-usable libraries (although this is changing for sql, as we shall see)
- long running jobs that need careful optimization

Spark best practices

- Filter early and filter often
- rely heavily on the core spark api (this is like the dictum to use built-in numpy functions), or well maintained public libraries like mllib
- consider intermixing SQL
- be careful with User Defined Functions. PySpark is a wrapper for scala spark. Python UDF's incur the python boxing/unboxing penalty



```
import org.apache.spark.SparkContext._
import org.apache.spark.{SparkConf, SparkContext}

object MaxTemperature {
    def main(args: Array[String]) {
        val conf = new SparkConf().setAppName("Max Temperature")
        val sc = new SparkContext(conf)
        sc.textFile(args(0))
            .map(_.split("\t"))
            .filter(rec => (rec(1) != "9999" && rec(2).matches("[01459]")))
            .map(rec => (rec(0).toInt, rec(1).toInt))
            .reduceByKey((a, b) => Math.max(a, b))
            .saveAsTextFile(args(1))
    }
}

// spark-submit --class MaxTemperature --master local \ spark-examples.jar
// input/ncdc/microtab/sample.txt output
```



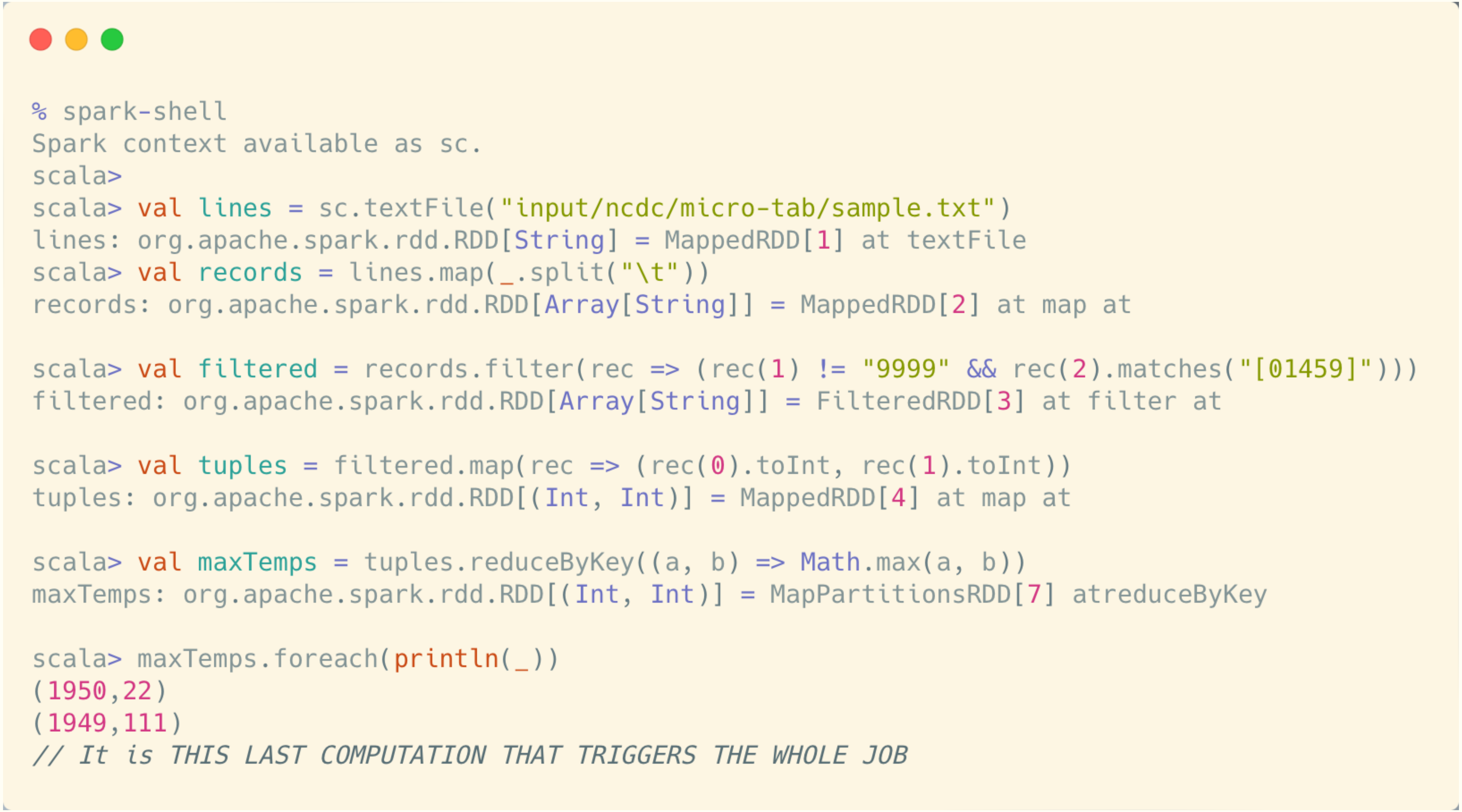
```
from pyspark import SparkContext
import re, sys

sc = SparkContext("local", "Max Temperature")
sc.textFile(sys.argv[1]) \
    .map(lambda s: s.split("\t")) \
    .filter(lambda rec: (rec[1] != "9999" and re.match("[01459]", rec[2]))) \
    .map(lambda rec: (int(rec[0]), int(rec[1]))) \
    .reduceByKey(max) \
    .saveAsTextFile(sys.argv[2])

#% spark-submit --master local \
#   ch19-spark/src/main/python/MaxTemperature.py \
#   input/ncdc/micro-tab/sample.txt output
```

How Spark works: RDDs and DAGs

- Spark uses a Directed Acyclic Graph (DAG) of tasks. By organizing a program into this DAG it can decide where to schedule the tasks, and provide resiliency guarantees on these tasks

- Spark keeps large working datasets in memory between these tasks. To do this, Spark has a central abstraction, the Resilient Distributed Dataset (RDD): a read-only collection of objects partitioned over the cluster.

- One or more RDDs are loaded as input. Transformations create a set of target RDDs which may be saved or used to compute a result



```
% spark-shell
Spark context available as sc.
scala>
scala> val lines = sc.textFile("input/ncdc/micro-tab/sample.txt")
lines: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile
scala> val records = lines.map(_.split("\t"))
records: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[2] at map at

scala> val filtered = records.filter(rec => (rec(1) != "9999" && rec(2).matches("[01459]")))
filtered: org.apache.spark.rdd.RDD[Array[String]] = FilteredRDD[3] at filter at

scala> val tuples = filtered.map(rec => (rec(0).toInt, rec(1).toInt))
tuples: org.apache.spark.rdd.RDD[(Int, Int)] = MappedRDD[4] at map at

scala> val maxTemps = tuples.reduceByKey((a, b) => Math.max(a, b))
maxTemps: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[7] atreduceByKey

scala> maxTemps.foreach(println(_))
(1950,22)
(1949,111)
// It is THIS LAST COMPUTATION THAT TRIGGERS THE WHOLE JOB
```

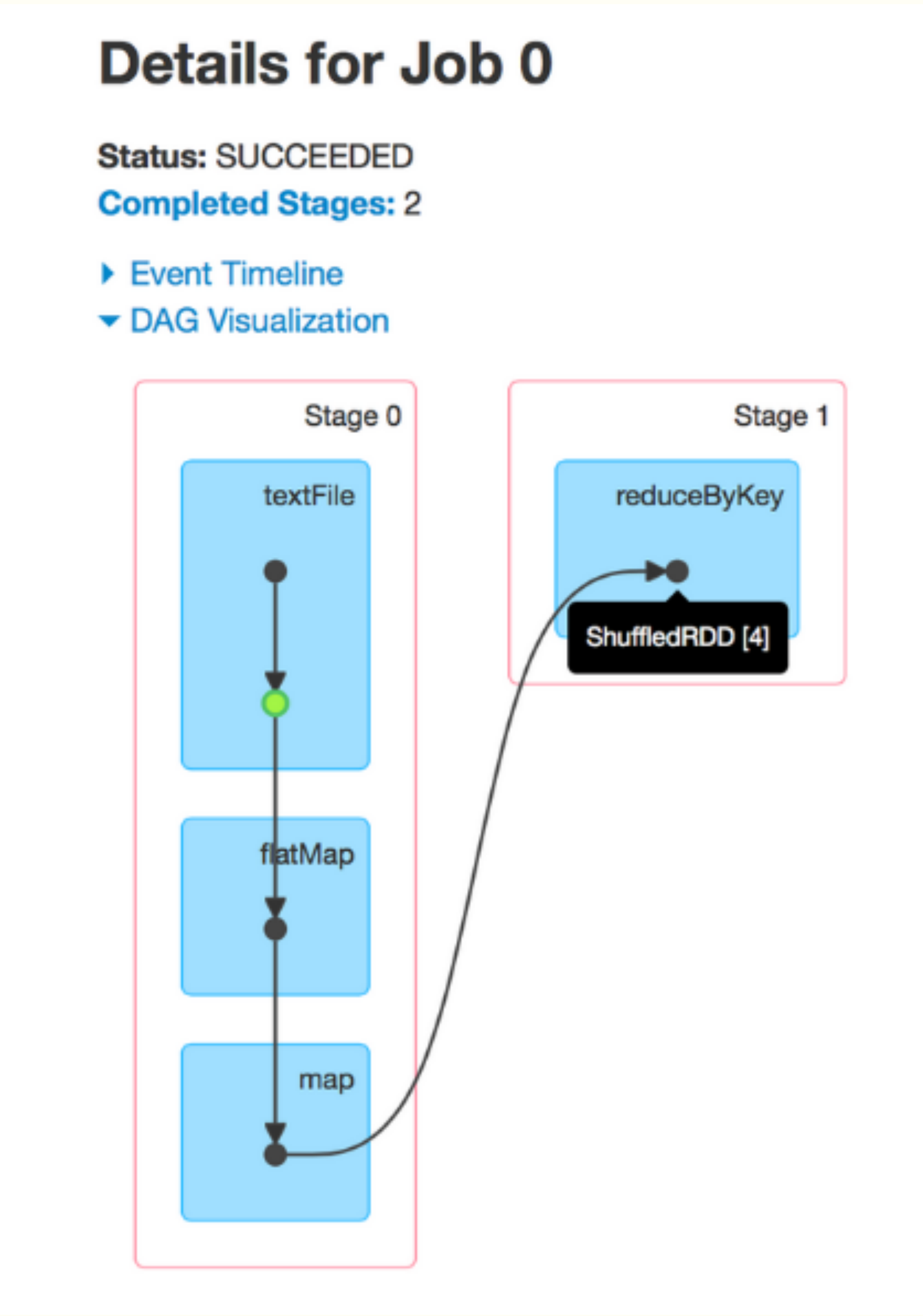
How Spark Works (contd)

- Spark is Lazy. No computations are done until they are absolutely needed

- When the final print is done, Spark walks back on the DAG, figures the tasks(stages) to run, and the order in which they need doing, and starts running them.

- The job runs in the context of an application, represented by a SparkContext instance

- Because each RDD is mutable, a stage that fails can be run again by obtaining the starting RDD as the finishing RDD of the previous stage.

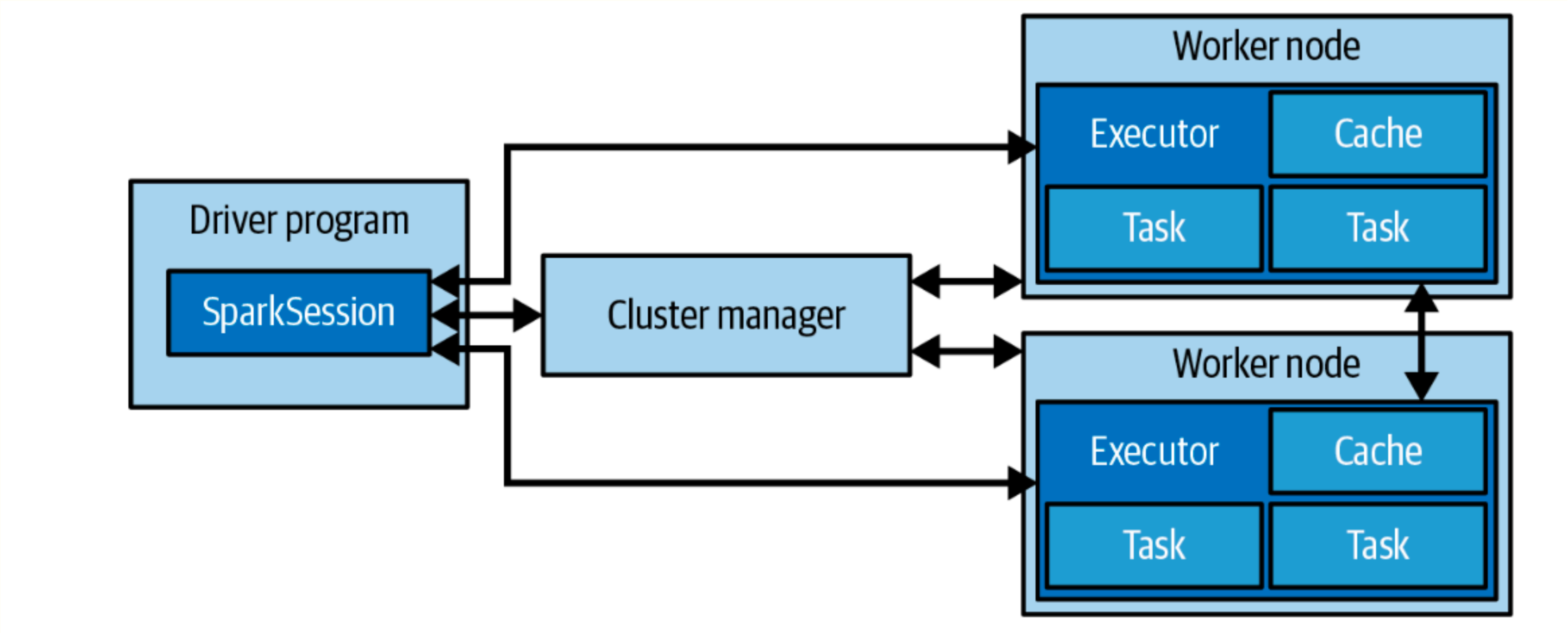


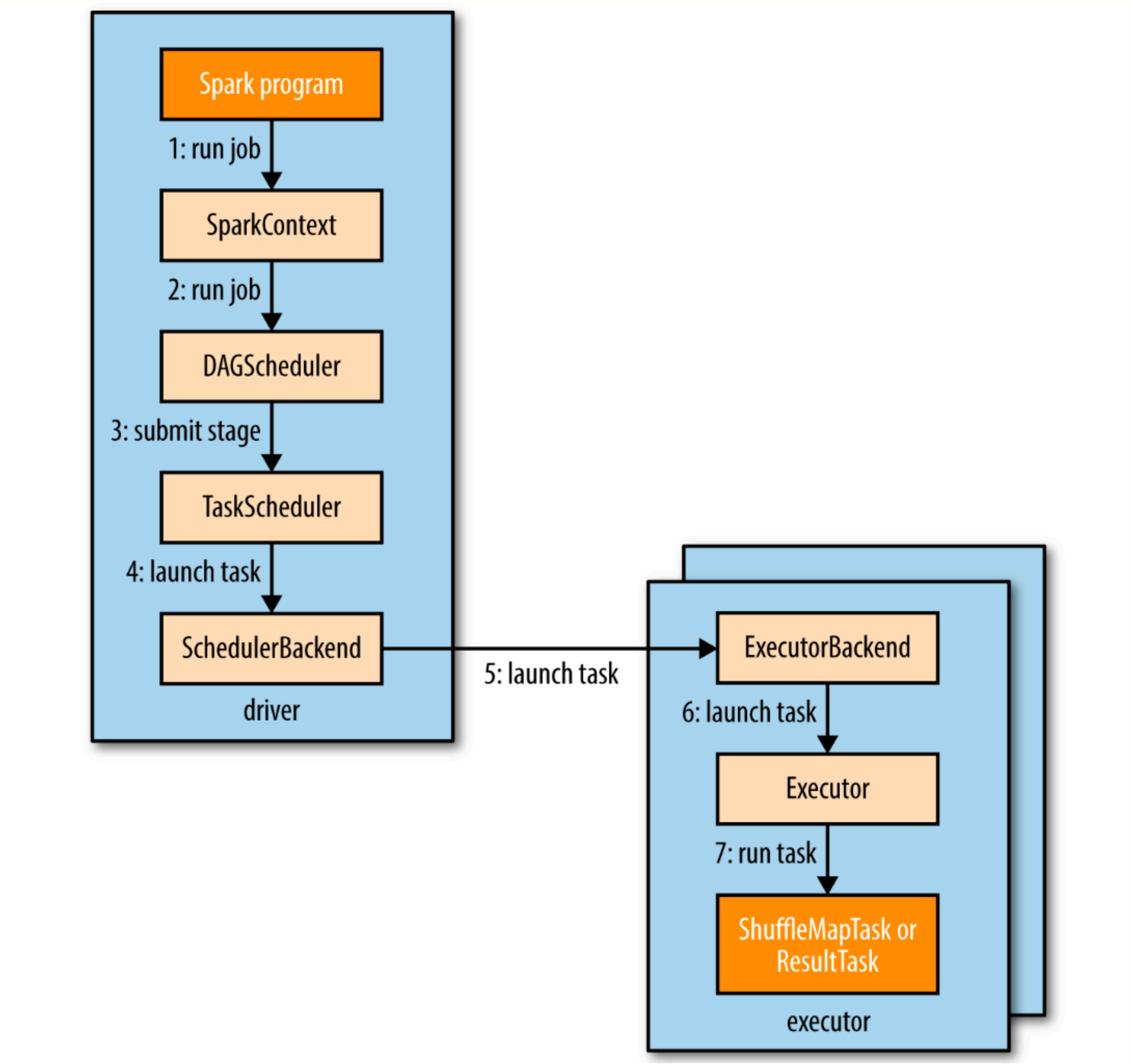
Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization

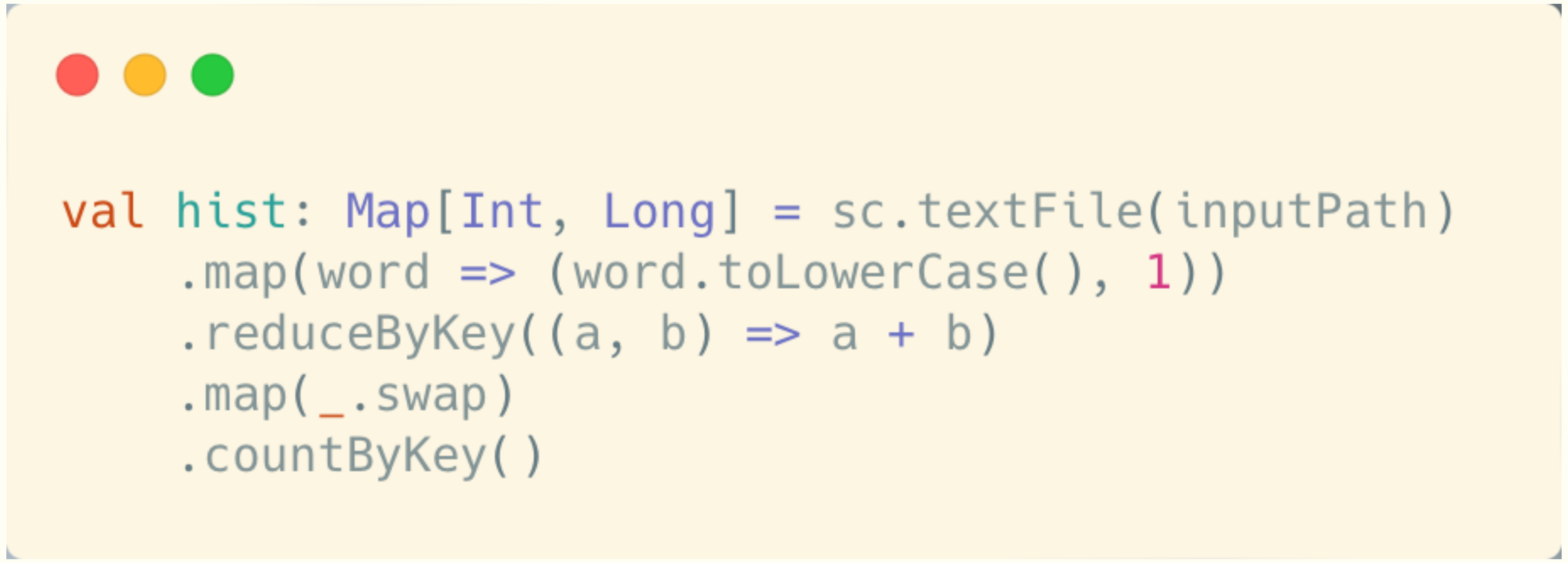
Spark Execution





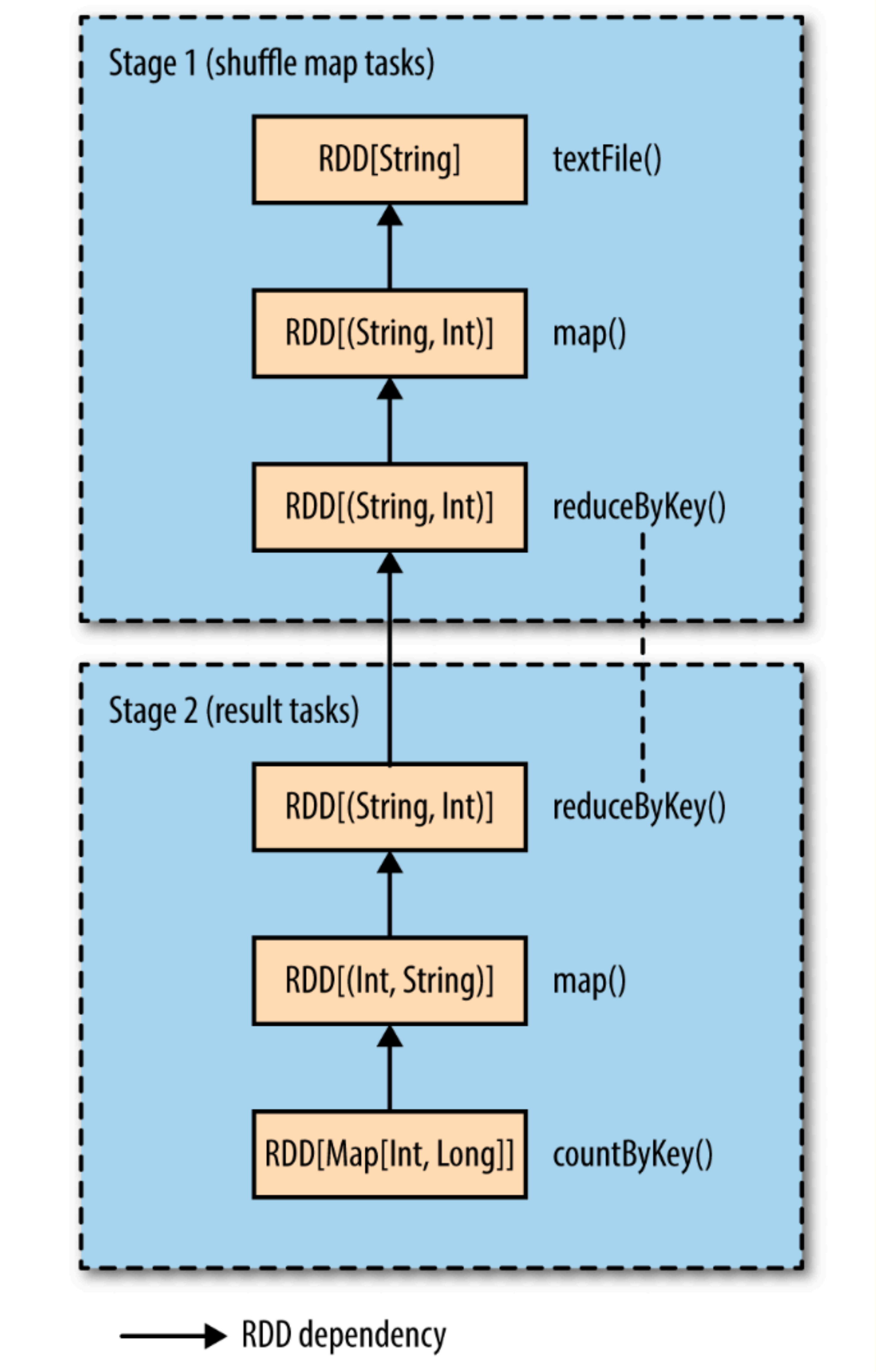
Univ.AI

Spark Execution (contd)



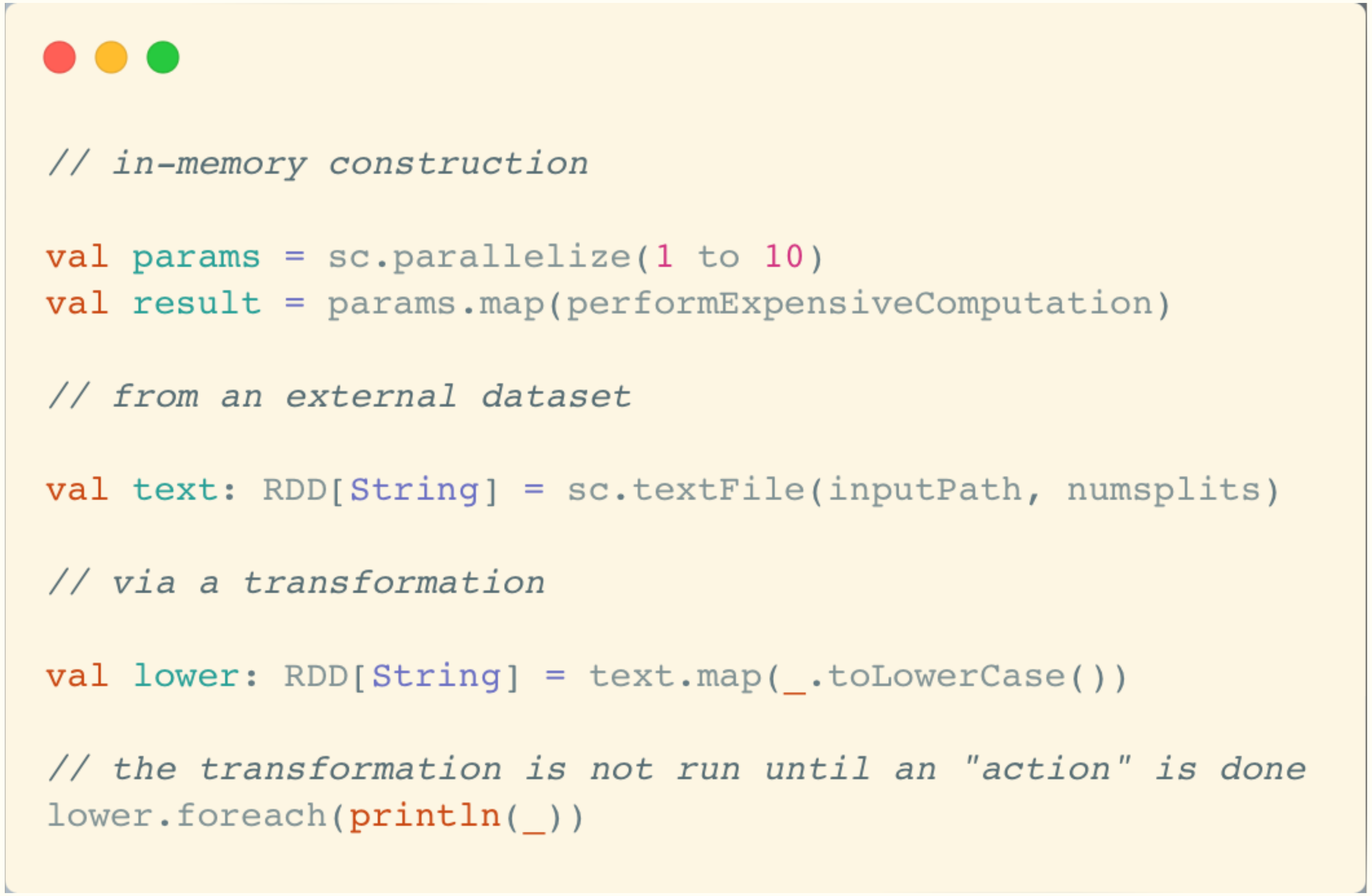
```
val hist: Map[Int, Long] = sc.textFile(inputPath)
    .map(word => (word.toLowerCase(), 1))
    .reduceByKey((a, b) => a + b)
    .map(_.swap)
    .countByKey()
```

reduceByKey creates a shuffle op, so 2 stages



More Spark

- the number of parallel jobs depends on spark.default.parallelism with the default in local mode and cluster mode being the number of cores
- Spark RDDs can be constructed in 3 ways: (1) from an in-memory collection of objects (2) creating a reference to an external dataset (3) transforming an existing RDD
- if the return type of an operation is a RDD, we have a transformation; else, an action.



```
// in-memory construction

val params = sc.parallelize(1 to 10)
val result = params.map(performExpensiveComputation)

// from an external dataset

val text: RDD[String] = sc.textFile(inputPath, numsplits)

// via a transformation

val lower: RDD[String] = text.map(_.toLowerCase())

// the transformation is not run until an "action" is done
lower.foreach(println(_))
```

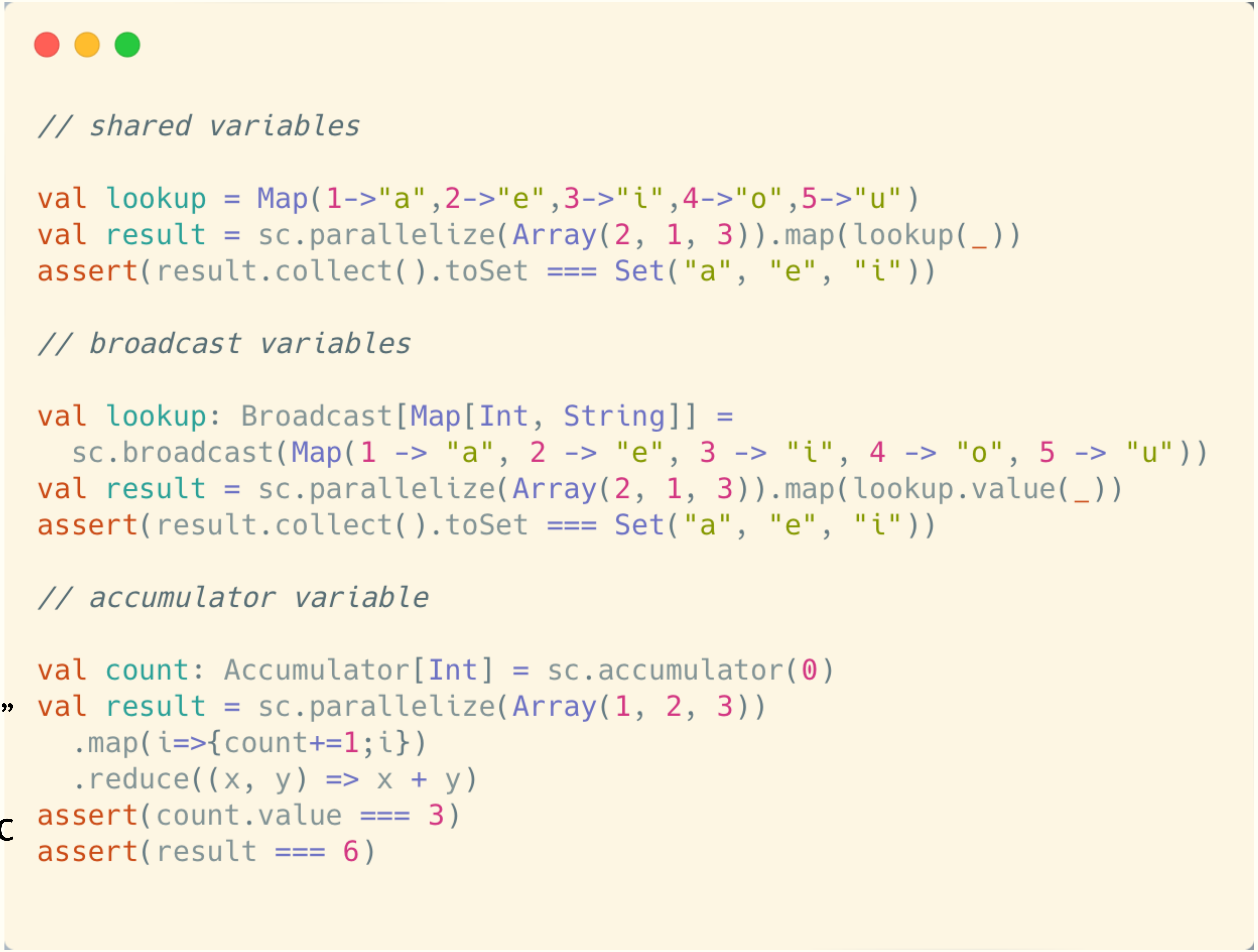
You can memory-cache a RDD by adding .cache() to it. Each partition will be cached in its executors memory. For example, text.cache()

More spark concepts

- disk persistence can be achieved by calling ".persist()" on a RDD. The serialization for this is done using regular java object serialization by default. You can serialize functions as well.

- Variables can be shared among executors by creating a mask, or even better, by using a "broadcast"

- Shared variables can have specific types, such as accumulators.



```
// shared variables

val lookup = Map(1->"a",2->"e",3->"i",4->"o",5->"u")
val result = sc.parallelize(Array(2, 1, 3)).map(lookup(_))
assert(result.collect().toSet === Set("a", "e", "i"))

// broadcast variables

val lookup: Broadcast[Map[Int, String]] =
  sc.broadcast(Map(1 -> "a", 2 -> "e", 3 -> "i", 4 -> "o", 5 -> "u"))
val result = sc.parallelize(Array(2, 1, 3)).map(lookup.value(_))
assert(result.collect().toSet === Set("a", "e", "i"))

// accumulator variable

val count: Accumulator[Int] = sc.accumulator(0)
val result = sc.parallelize(Array(1, 2, 3))
  .map(i=>{count+=1;i})
  .reduce((x, y) => x + y)
assert(count.value === 3)
assert(result === 6)
```

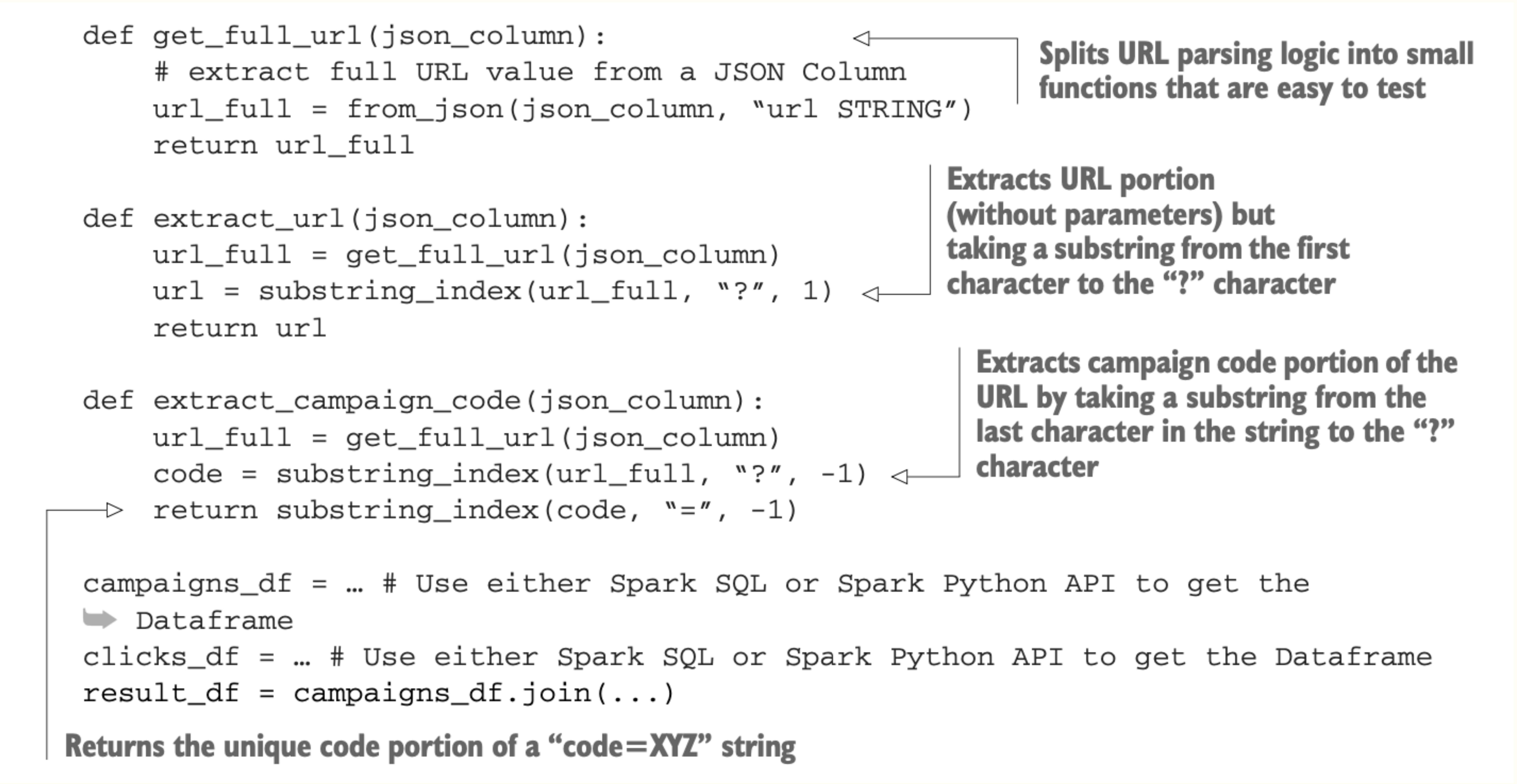
The advantages of Spark

- Write modular code, which can be checked into version control and tested

- use dataframe api or SQL to access data and turn it into RDDs

- spark native is usually faster than SQL

- access to all file formats (parquet/avro/ORC) and newer transactional formats (iceberg/hudi/delta-lake)



```
def get_full_url(json_column):
    # extract full URL value from a JSON Column
    url_full = from_json(json_column, “url STRING”)
    return url_full

def extract_url(json_column):
    url_full = get_full_url(json_column)
    url = substring_index(url_full, “?”, 1)
    return url

def extract_campaign_code(json_column):
    url_full = get_full_url(json_column)
    code = substring_index(url_full, “?”, -1)
    return substring_index(code, “=”, -1)

campaigns_df = … # Use either Spark SQL or Spark Python API to get the
➥ Dataframe
clicks_df = … # Use either Spark SQL or Spark Python API to get the Dataframe
result_df = campaigns_df.join(...)
```

Splits URL parsing logic into small functions that are easy to test

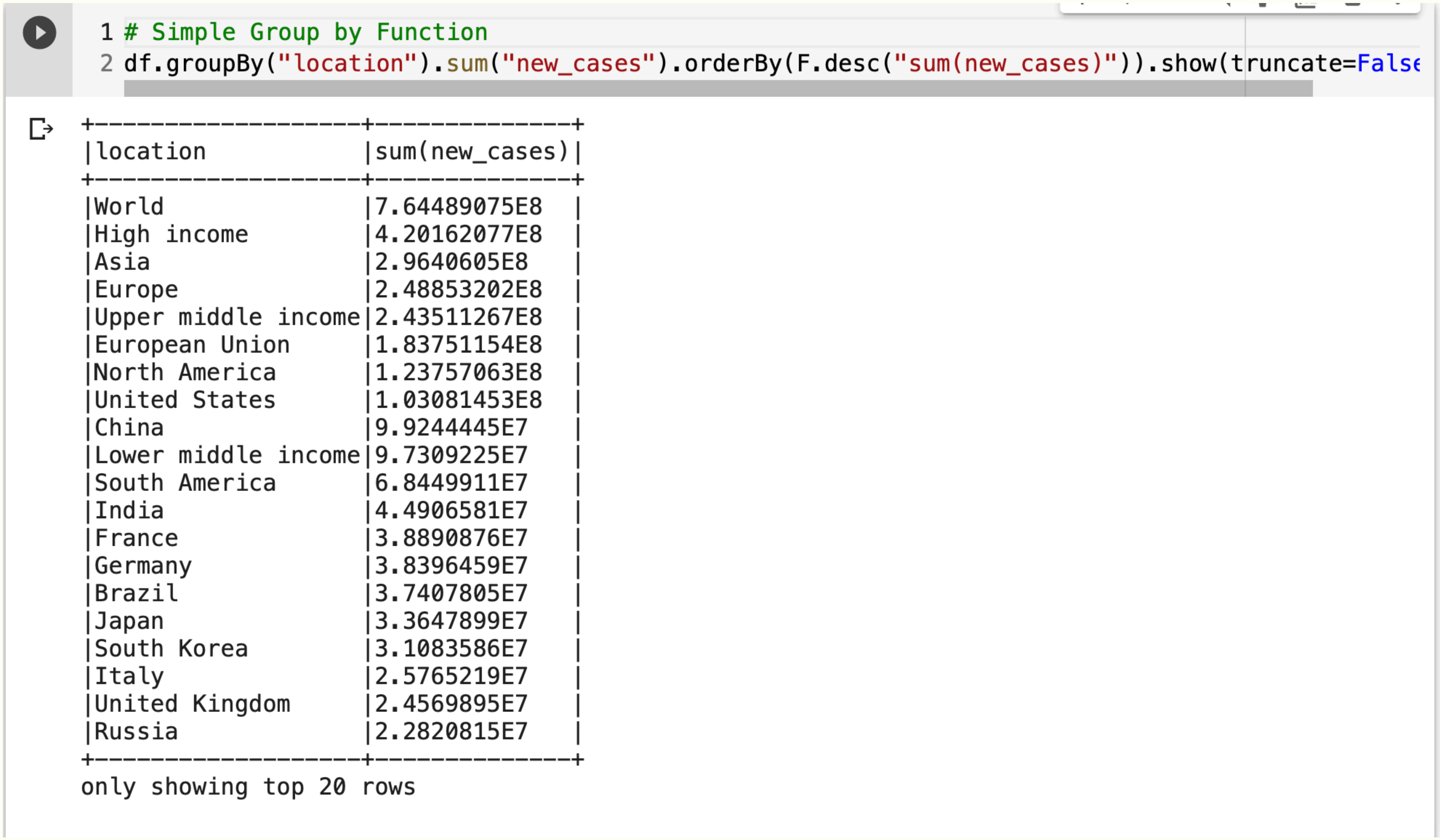
Extracts URL portion (without parameters) but taking a substring from the first character to the “?” character

Extracts campaign code portion of the URL by taking a substring from the last character in the string to the “?” character

Returns the unique code portion of a “code=XYZ” string

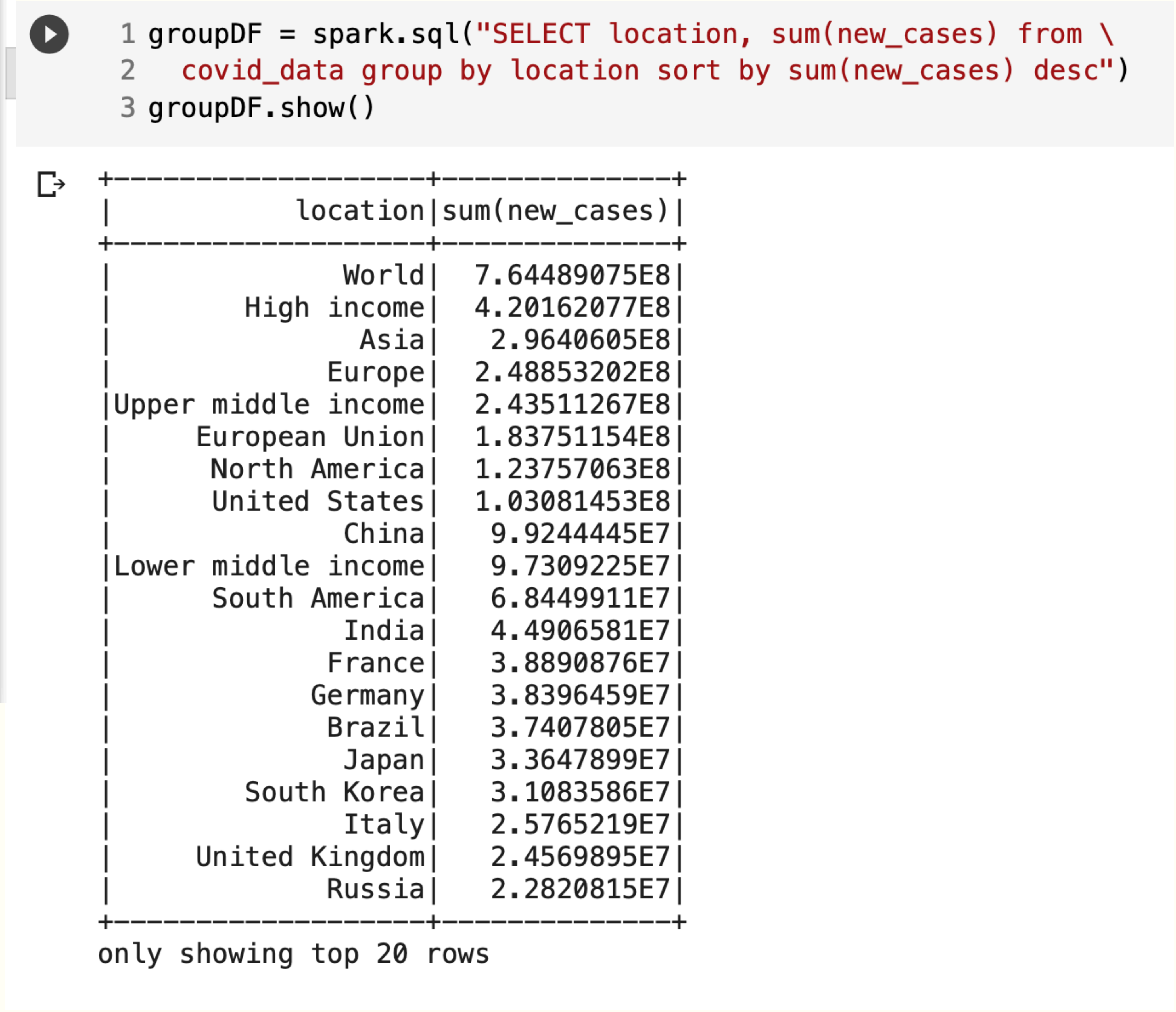
Univ.AI

Spark: Dataframes and SQL



```
# Simple Group by Function
df.groupBy("location").sum("new_cases").orderBy(F.desc("sum(new_cases)")).show(truncate=False
```

```
+-------------------+--------------+
|location           |sum(new_cases)|
+-------------------+--------------+
|World              |7.64489075E8  |
|High income        |4.20162077E8  |
|Asia               |2.9640605E8   |
|Europe             |2.48853202E8  |
|Upper middle income|2.43511267E8  |
|European Union     |1.83751154E8  |
|North America      |1.23757063E8  |
|United States      |1.03081453E8  |
|China              |9.9244445E7   |
|Lower middle income|9.7309225E7   |
|South America      |6.8449911E7   |
|India              |4.4906581E7   |
|France             |3.8890876E7   |
|Germany            |3.8396459E7   |
|Brazil             |3.7407805E7   |
|Japan              |3.3647899E7   |
|South Korea        |3.1083586E7   |
|Italy              |2.5765219E7   |
|United Kingdom     |2.4569895E7   |
|Russia             |2.2820815E7   |
+-------------------+--------------+
only showing top 20 rows
```



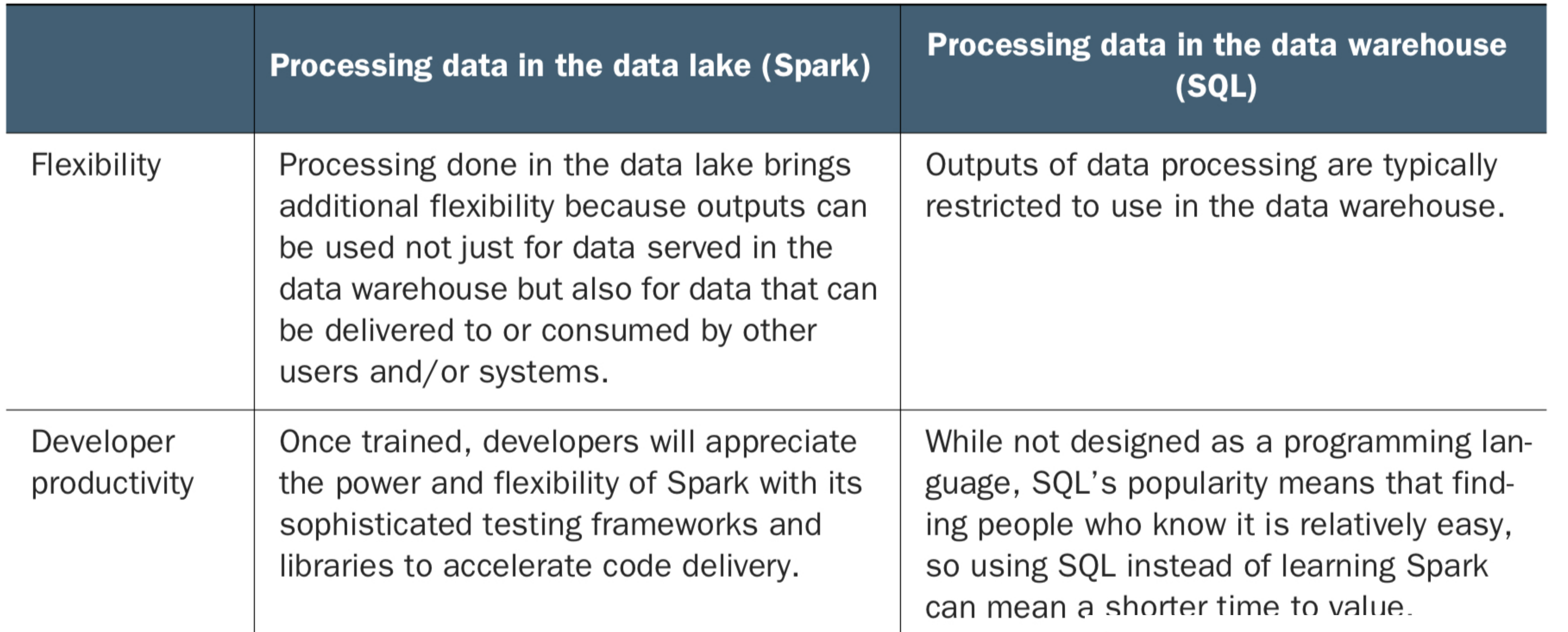
```
groupDF = spark.sql("SELECT location, sum(new_cases) from \
  covid_data group by location sort by sum(new_cases) desc")
groupDF.show()
```

```
+-------------------+--------------+
|           location|sum(new_cases)|
+-------------------+--------------+
|              World|  7.64489075E8|
|        High income|  4.20162077E8|
|               Asia|   2.9640605E8|
|             Europe|  2.48853202E8|
|Upper middle income|  2.43511267E8|
|     European Union|  1.83751154E8|
|      North America|  1.23757063E8|
|      United States|  1.03081453E8|
|              China|   9.9244445E7|
|Lower middle income|   9.7309225E7|
|      South America|   6.8449911E7|
|              India|   4.4906581E7|
|             France|   3.8890876E7|
|            Germany|   3.8396459E7|
|             Brazil|   3.7407805E7|
|              Japan|   3.3647899E7|
|        South Korea|   3.1083586E7|
|              Italy|   2.5765219E7|
|     United Kingdom|   2.4569895E7|
|             Russia|   2.2820815E7|
+-------------------+--------------+
only showing top 20 rows
```

Spark can also speak jdbc to create dataframes or sql queries on existing databases. So you can access souce databases, warehouses, and parquet files as well.

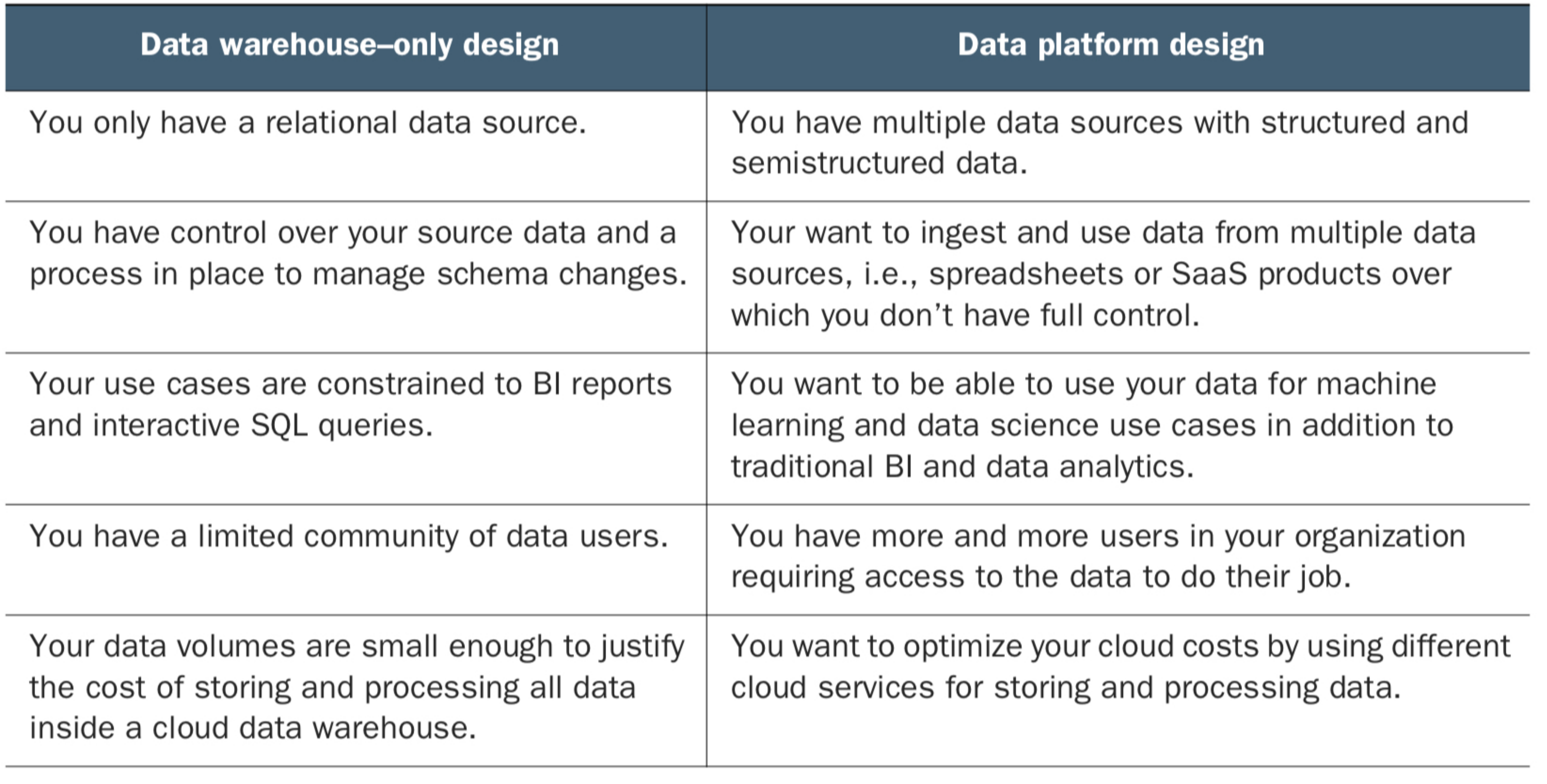
Univ.AI

Warehouse, or Lake, or Lakehouse or Platform?



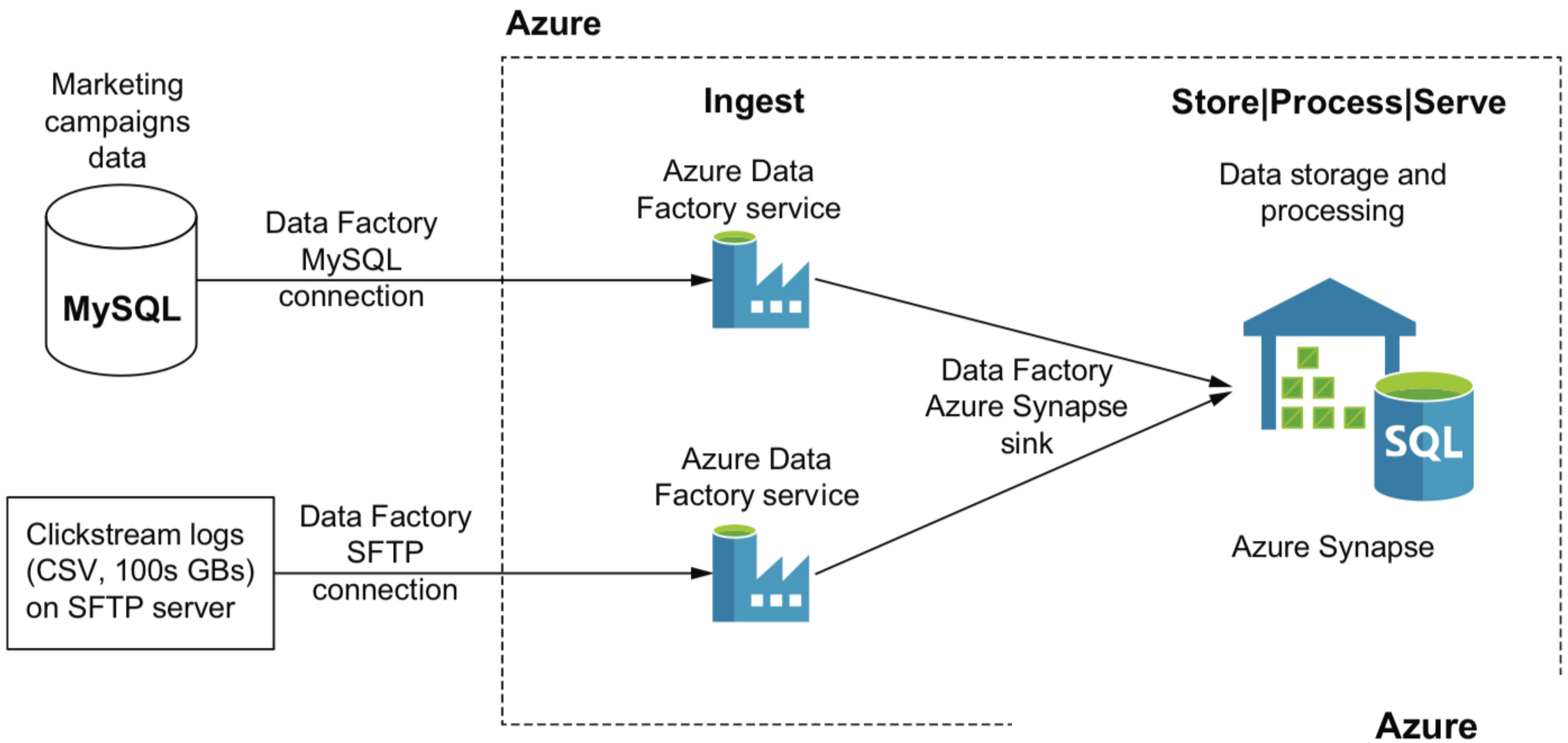
	Processing data in the data lake (Spark)	Processing data in the data warehouse (SQL)
Flexibility	Processing done in the data lake brings additional flexibility because outputs can be used not just for data served in the data warehouse but also for data that can be delivered to or consumed by other users and/or systems.	Outputs of data processing are typically restricted to use in the data warehouse.
Developer productivity	Once trained, developers will appreciate the power and flexibility of Spark with its sophisticated testing frameworks and libraries to accelerate code delivery.	While not designed as a programming language, SQL's popularity means that finding people who know it is relatively easy, so using SQL instead of learning Spark can mean a shorter time to value.

- Warehouse has a columnar database
- Lake has parquet and other files on object storage
- Lakehouse adds a table structure to the lake
- Platform adds warehouse/lake/etc

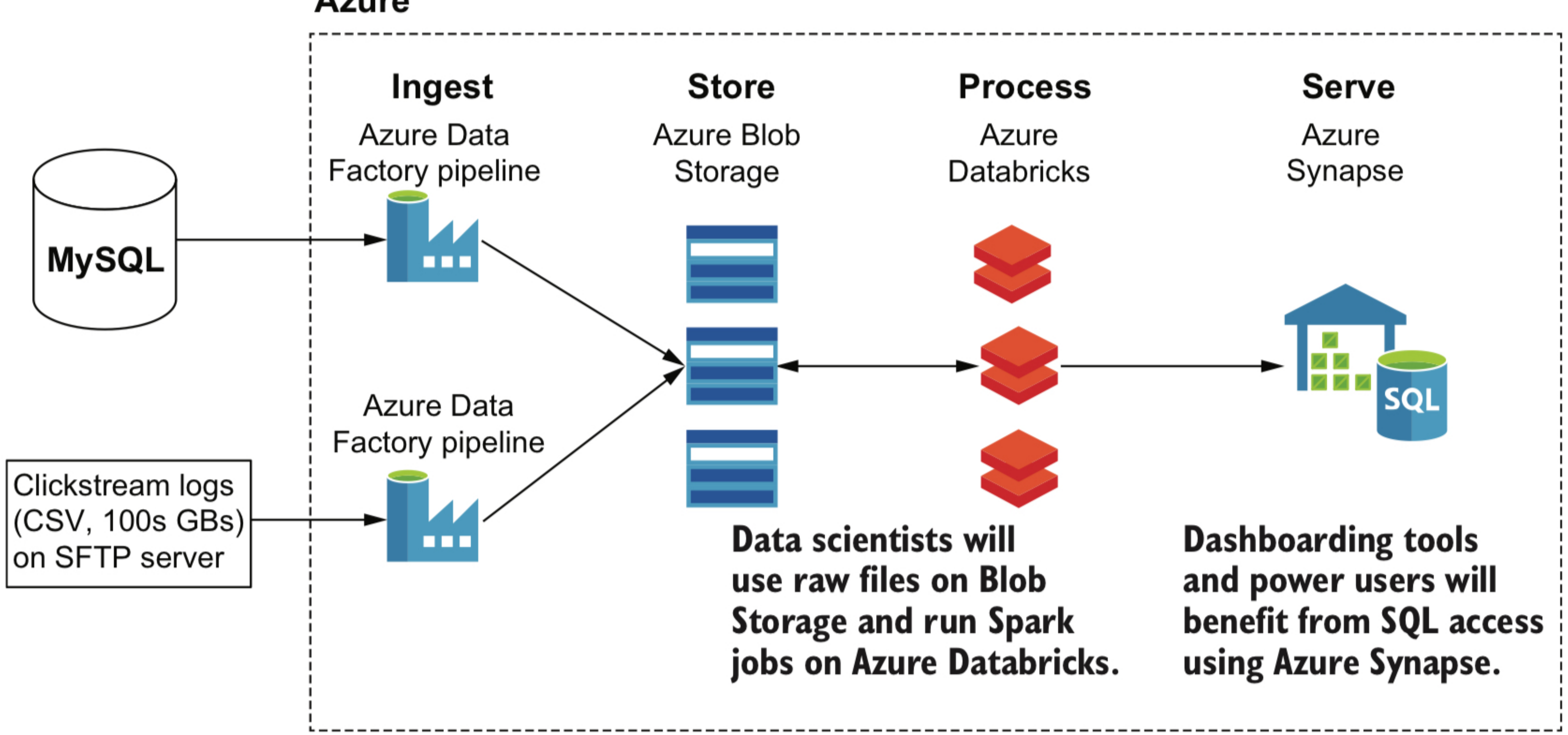


Data warehouse–only design	Data platform design
You only have a relational data source.	You have multiple data sources with structured and semistructured data.
You have control over your source data and a process in place to manage schema changes.	Your want to ingest and use data from multiple data sources, i.e., spreadsheets or SaaS products over which you don't have full control.
Your use cases are constrained to BI reports and interactive SQL queries.	You want to be able to use your data for machine learning and data science use cases in addition to traditional BI and data analytics.
You have a limited community of data users.	You have more and more users in your organization requiring access to the data to do their job.
Your data volumes are small enough to justify the cost of storing and processing all data inside a cloud data warehouse.	You want to optimize your cloud costs by using different cloud services for storing and processing data.

From Warehouse


Azure
Ingest
Store|Process|Serve
Marketing campaigns data
MySQL
Data Factory MySQL connection
Azure Data Factory service
Data storage and processing
Data Factory Azure Synapse sink
SQL
Azure Synapse
Clickstream logs (CSV, 100s GBs) on SFTP server
Data Factory SFTP connection
Azure Data Factory service

To Data Platform


Azure
Ingest
Azure Data Factory pipeline
Store
Azure Blob Storage
Process
Azure Databricks
Serve
Azure Synapse
MySQL
SQL
Azure Data Factory pipeline
Clickstream logs (CSV, 100s GBs) on SFTP server

Data scientists will use raw files on Blob Storage and run Spark jobs on Azure Databricks.

Dashboarding tools and power users will benefit from SQL access using Azure Synapse.