Distributed Data Processing

SDU

Agenda

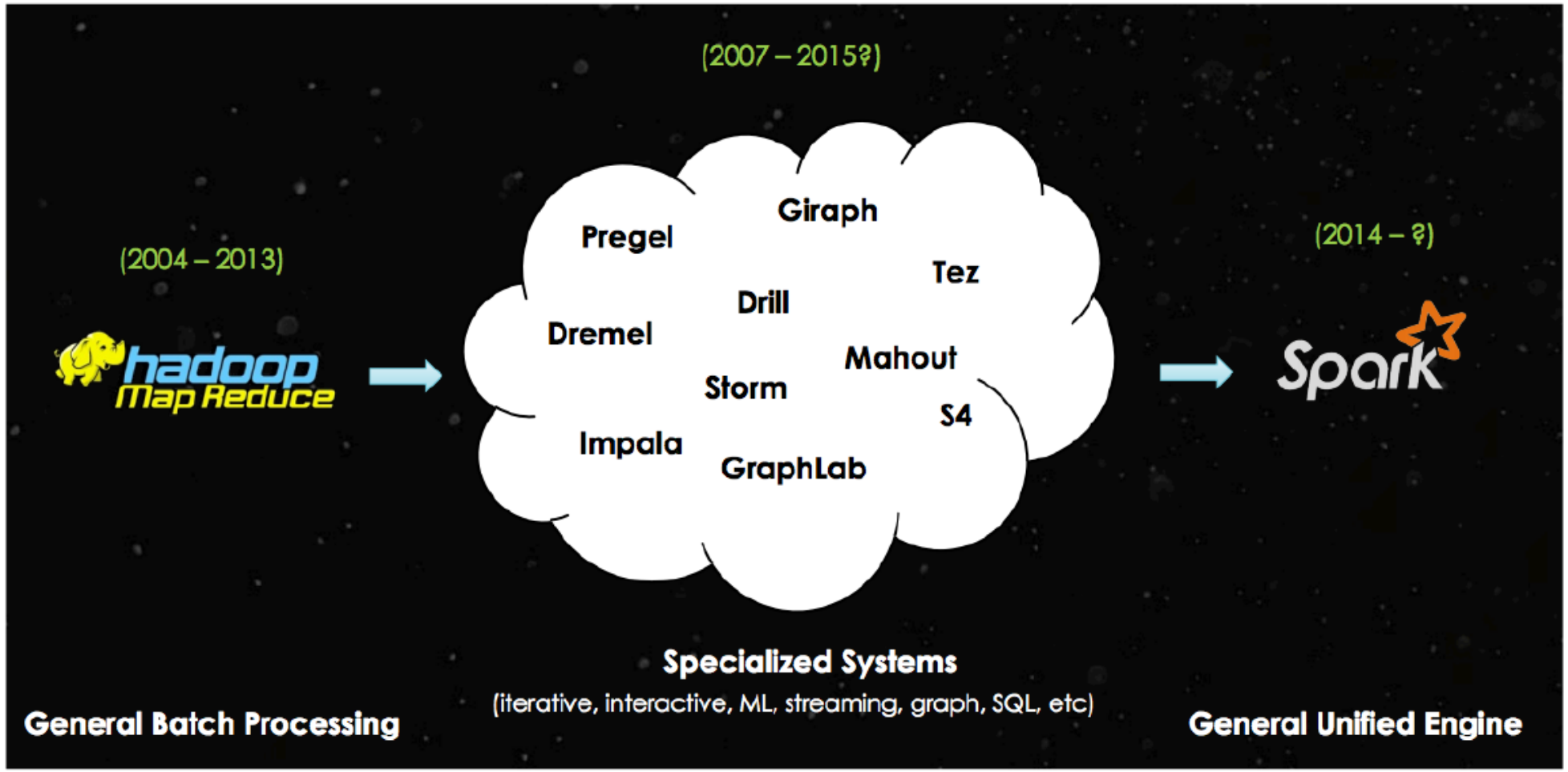
→ Spark (Historical Data)

→ Spark SQL

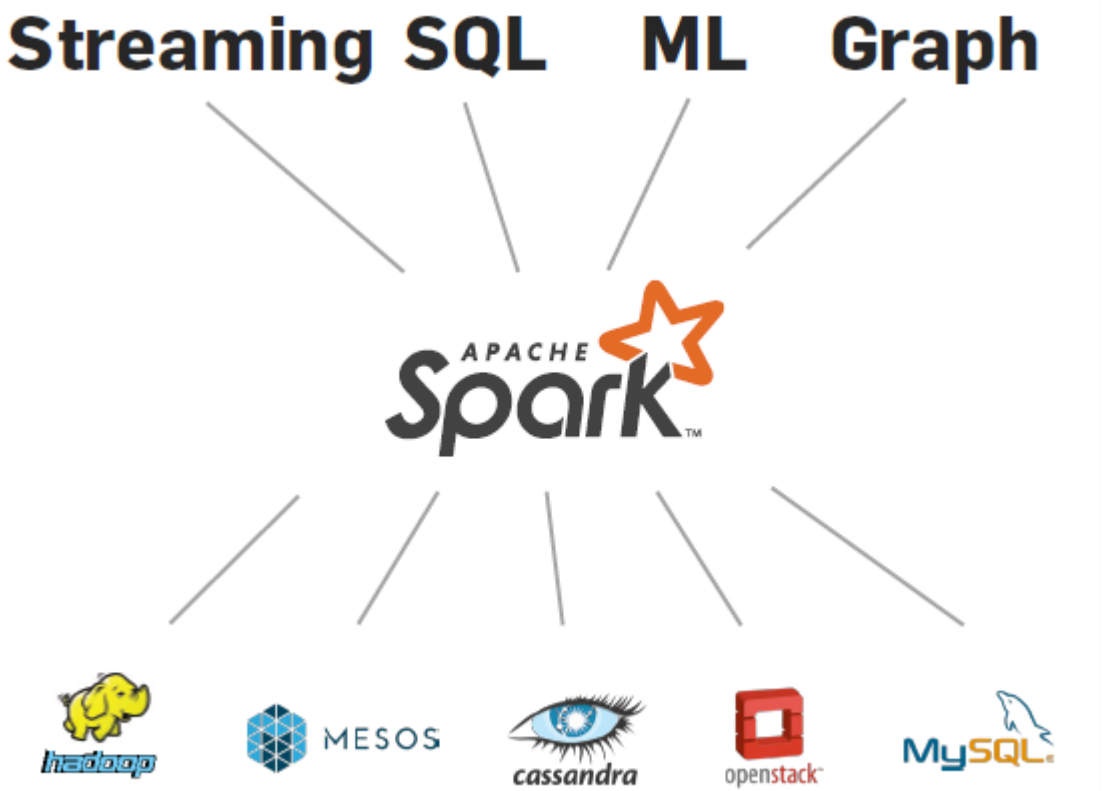
→ Spark Streaming (Live Data)

Spark (Historical Data)

The history before Spark



What is Spark



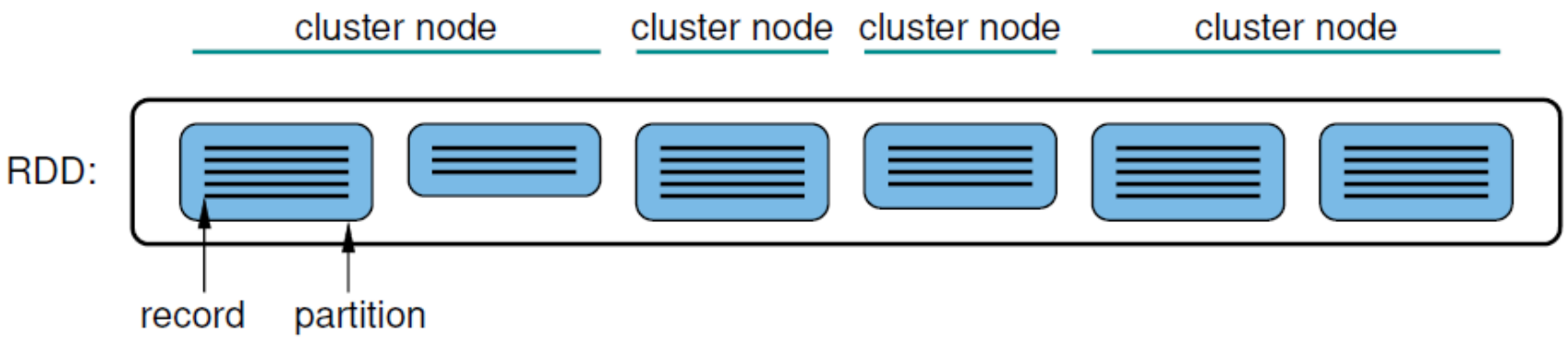
"Engine" for distributed data processing over a cluster:

- programming language [a]
- libraries
- underlying engine:
  - interfaces with storage
  - assigns tasks to servers
  - moves data in the network
  - deals with failures

---

[a] Python, Scala, Java, R

RDD (Resilient Distributed Dataset)



→ RDDs are **immutable**!

→ RDDs are **In Memory**!

→ RDD records are partitioned:

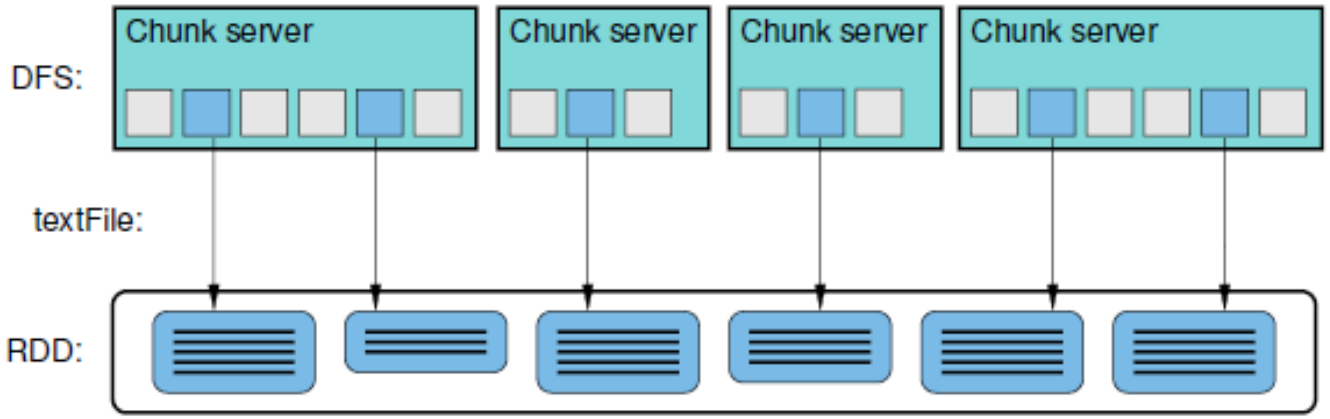
→ A partition:

- is a batch of data, operated on in **parallel**
- the number of partitions:
  - configured by you
  - "capped" by the cluster resources given to the job
  - determined by the number of data chunks in input.

SDU

The location of partitions

- A partition has a **preferred location**
  (e.g., on the machine holding the initial data chunk):



- New partitions are **computed** from previous partitions.
- At machine failure, a partition is **recomputed** on another machine.

Partitioning function (configurable): determines the location of records

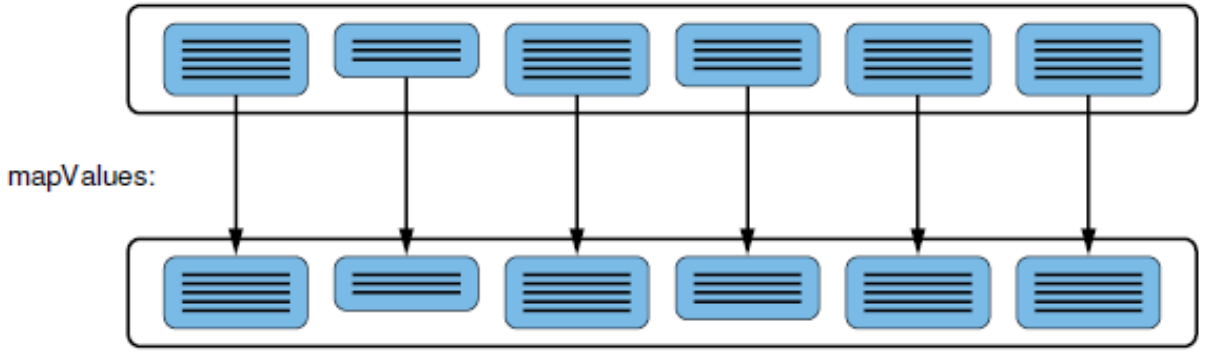
- Some operations **don't impose any partitioning**
  → `textFile, wholeTextFile, map`

`wholeTextFiles`(*path*, *minPartitions=None*, *use_unicode=True*) [source]

Read a directory of text files from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. Each file is read as a single record and returned in a key-value pair, where the key is the path of each file, the value is the content of each file.

`map`(*f*, *preservesPartitioning=False*) [source]

Return a new RDD by applying a function to each element of this RDD.



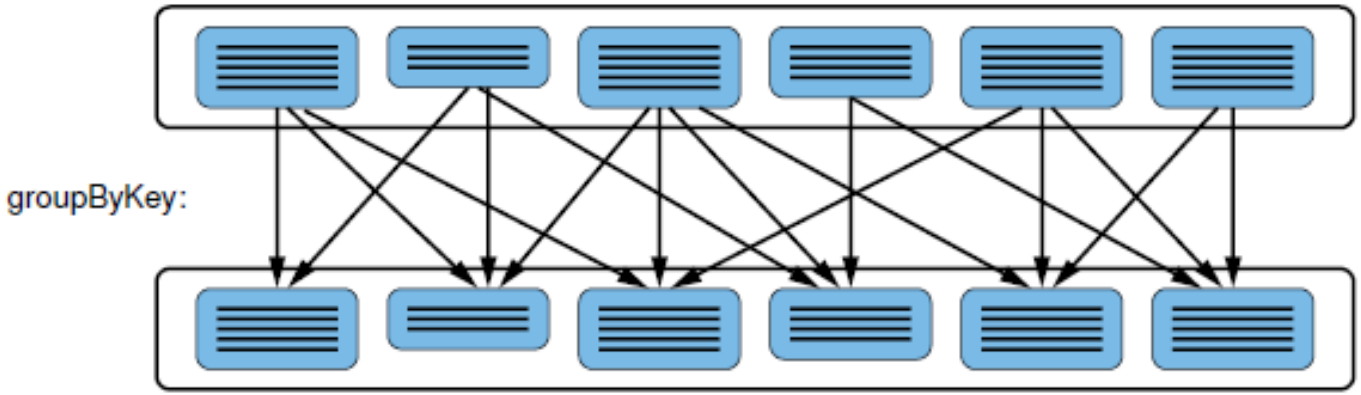
- Some operations **impose a partitioning**
  - → `groupByKey (hash by key)`
  - → `sortBy(Key) (range partitioning)`

groupByKey(*numPartitions=None, partitionFunc=<function portable_hash>*) [source]

Group the values for each key in the RDD into a single sequence. Hash-partitions the resulting RDD with numPartitions partitions.

sortBy(*keyfunc, ascending=True, numPartitions=None*) [source]

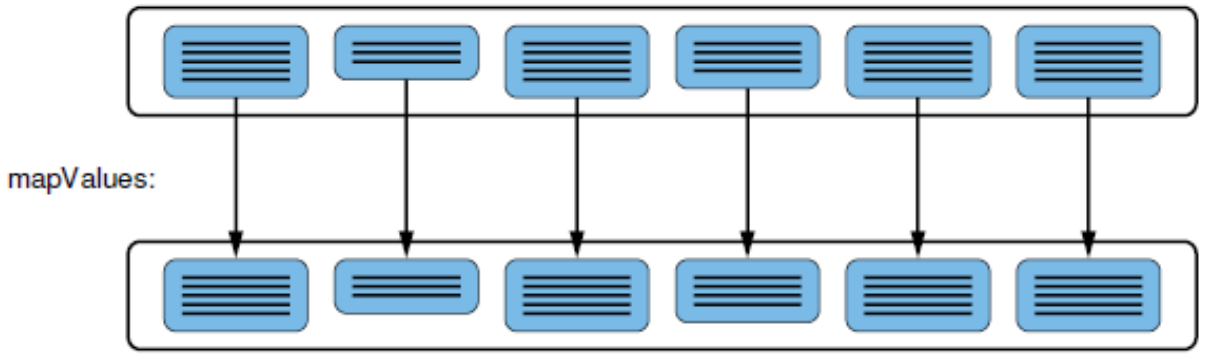
Sorts this RDD by the given keyfunc



Dependencies between RDDs

Narrow dependencies

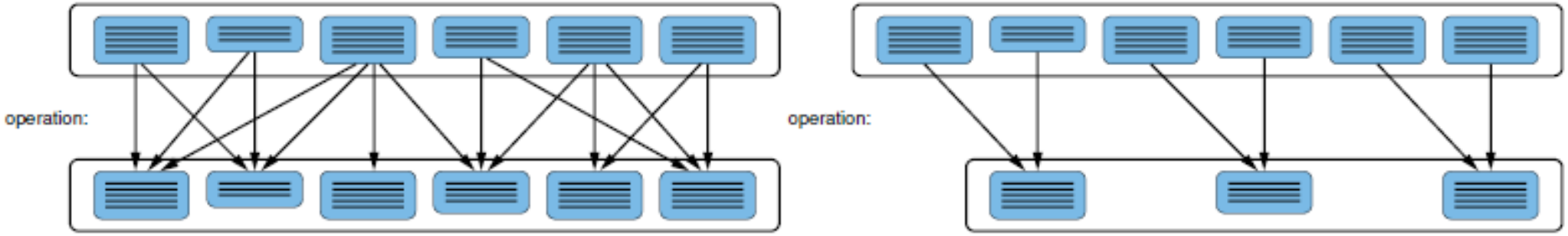
Each partition of the parent RDD(s) is used by **at most one** partition of the child RDD.



→ *Why you care*:
→ (1) operations with narrow dependencies
→ are “pipelined” **local**ly on one cluster node;
→ (2) if one partition is lost, recomputation is also local.

Wide dependencies

A parent partition has **multiple** child partitions depending on it.



(left: wide; right: narrow!)

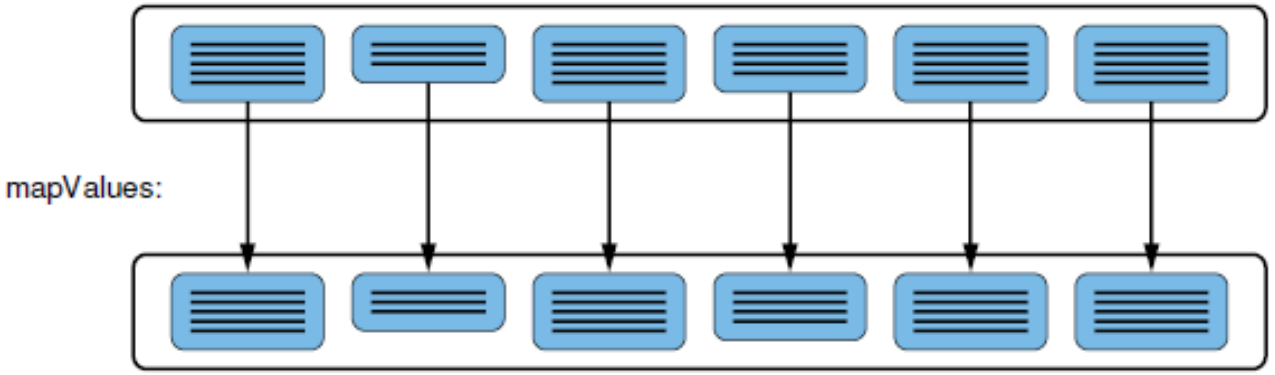
→ *Why you care*:
→ (1) operations with wide dependencies require data from all parent
→ partitions to be **shuffled** on the network (MapReduce-like);
→ (2) if one partition is lost, you may need to recompute the entire
→ program!

map

`map`(*f*, *preservesPartitioning=False*)

Return a new RDD by applying a function to each element of this RDD.

This operation may change partitioning
→ (e.g., if parent RDD was partitioned by key and *f* changes keys),
→ but doesn't impose a partitioner on the child RDD,
→ so locations need not change.



(narrow)

flatMap

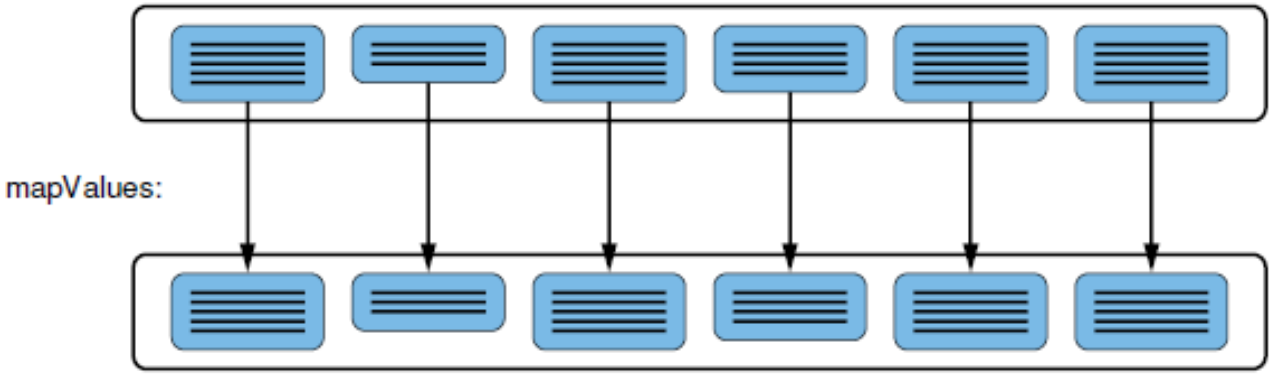
`flatMap`(*f*, *preservesPartitioning=False*) [source]

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

`flatMapValues`(*f*) [source]

Pass each value in the key-value pair RDD through a flatMap function without changing the keys; this also retains the original RDD's partitioning.

→ Neither imposes a partitioning, so locations need not change.



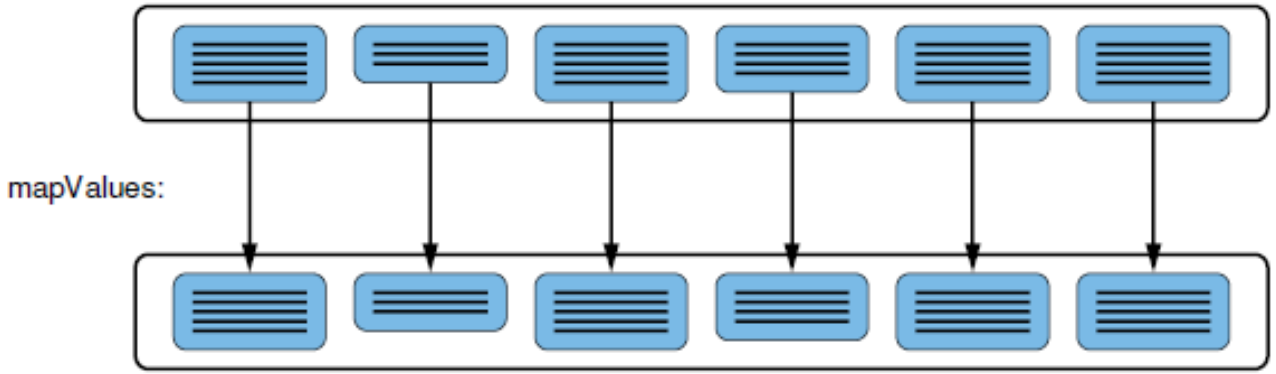
(narrow)

filter

filter(*f*) [source]

Return a new RDD containing only the elements that satisfy a predicate.

→ Preserves the parent partitioning, so locations need not change.



(narrow)

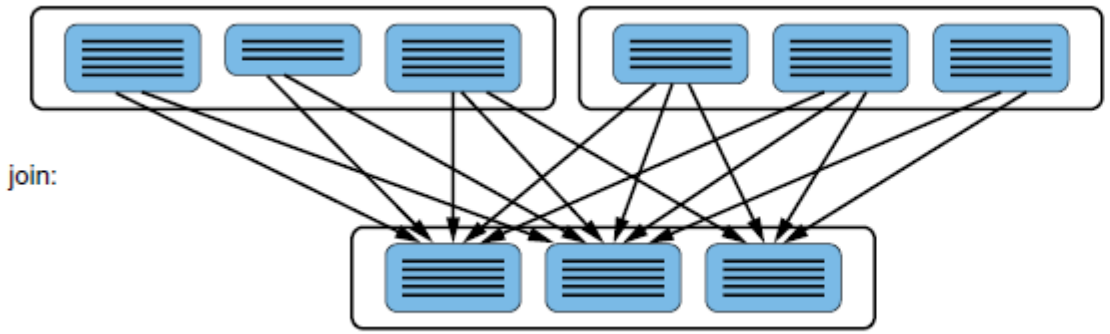
join

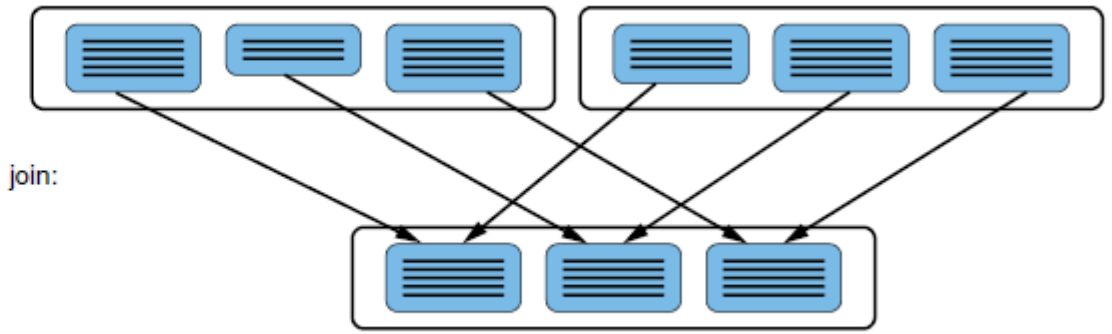
join(*other*, *numPartitions=None*) [source]

Return an RDD containing all pairs of elements with matching keys in **self** and **other**.

Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in **self** and (k, v2) is in **other**.

Performs a hash join across the cluster.





(*left*: wide, if parent RDDs are not co-partitioned;
right: narrow, if co-partitioned)

The graph of dependencies over RDDs

→Transformations over RDD partitions:

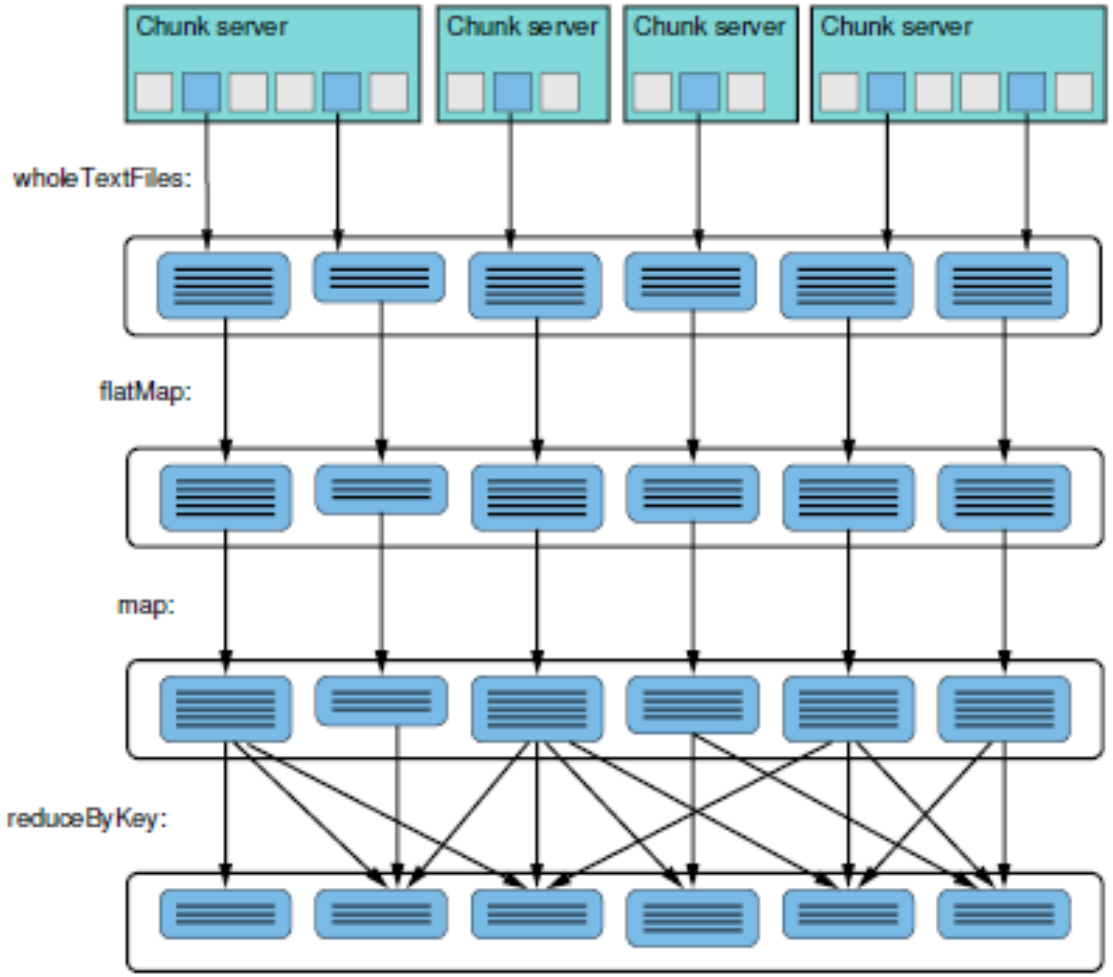
- build up into a **directed graph without cycles** (DAG)
- a **path** ending in a partition

  = a log of transformations

  = the “lineage” of that data

RDD dependencies: word count

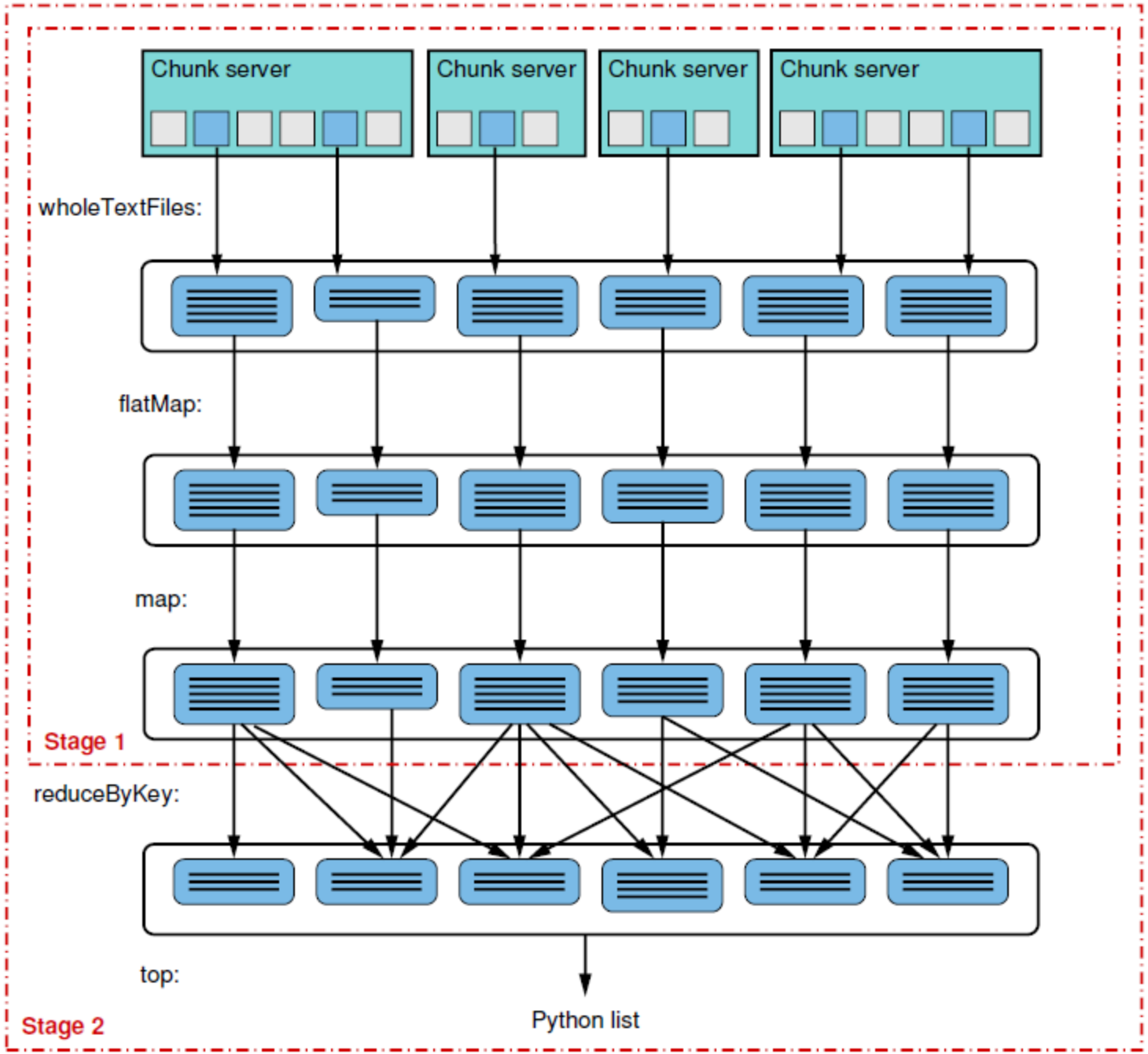
```
rdd = sc.wholeTextFiles("/data/doina/
    Gutenberg-EBooks")

words = rdd
    .flatMap(lambda (file, contents):
                contents.lower().split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)

print words.toDebugString()

top_words = words
    .top(50, key=lambda record:
                    record[1])
```



Task scheduling

→When your program calls an **action** (take, count, top) on an RDD:

- the scheduler examines that RDD's lineage graph, and
- builds a **sequence of stages** to execute;
- each stage "groups" pipelined transformations with **narrow**
- **dependencies**,
- so end at operations with **wide dependencies** (shuffle).

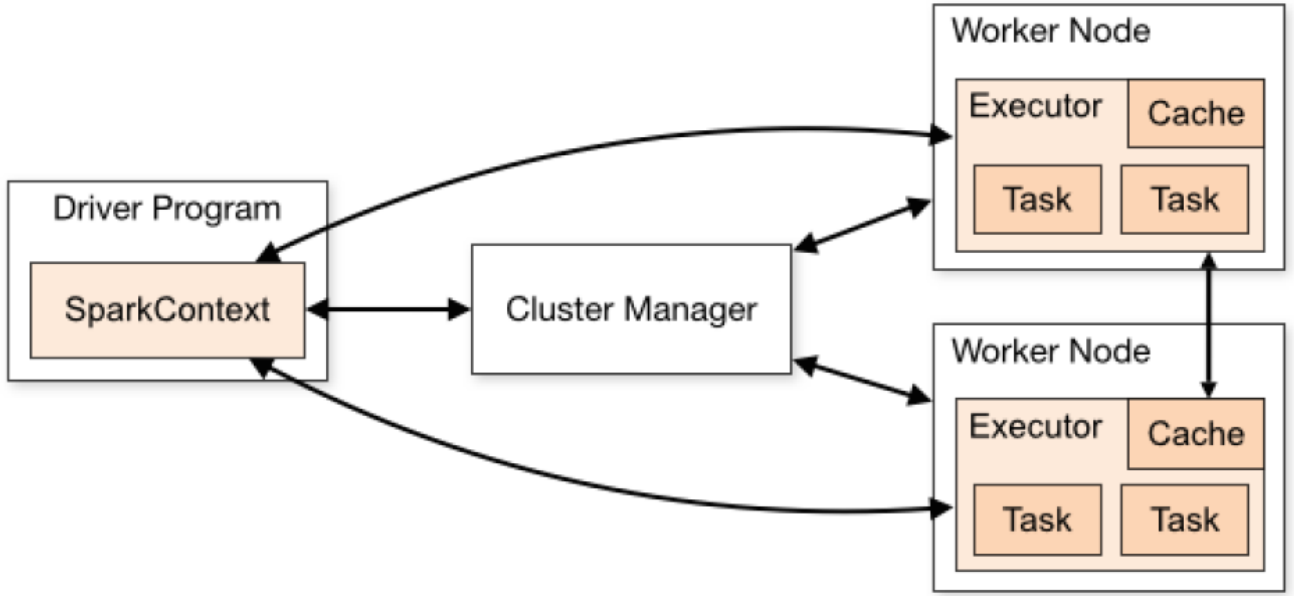

Chunk server
Chunk server
Chunk server
Chunk server
wholeTextFiles:
flatMap:
map:
Stage 1
reduceByKey:
top:
Python list
Stage 2

SDU

→ Then, the Spark **driver program**:

- launches a group of **tasks** to compute missing partitions from each stage,
- until it has computed the target RDD and its action;
- assigns tasks to machines based on **data locality**:

  if the input partition is in memory/on disk on a node, the task runs on that node.



Spark with Yarn

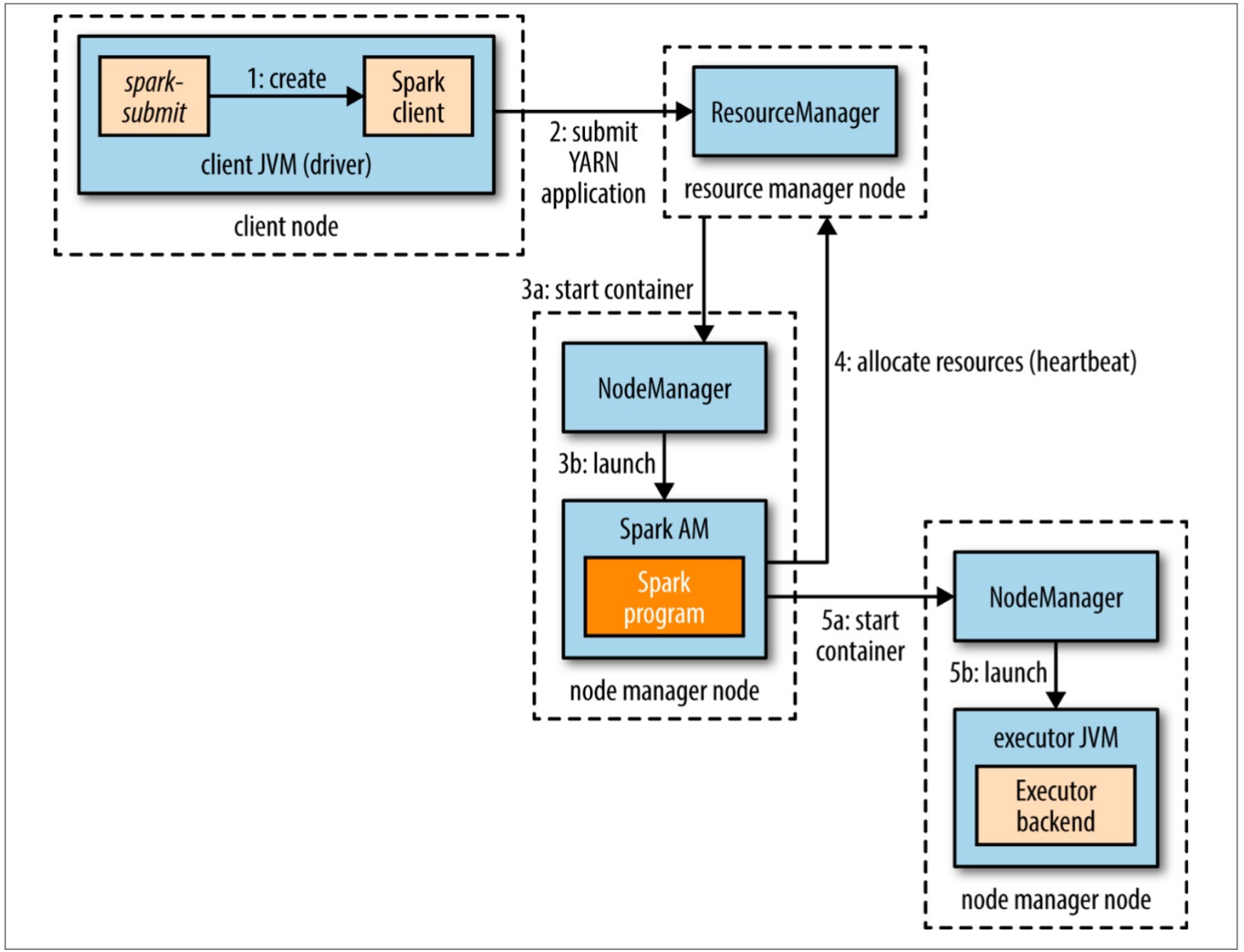


Figure 19-4. How Spark executors are started in YARN cluster mode

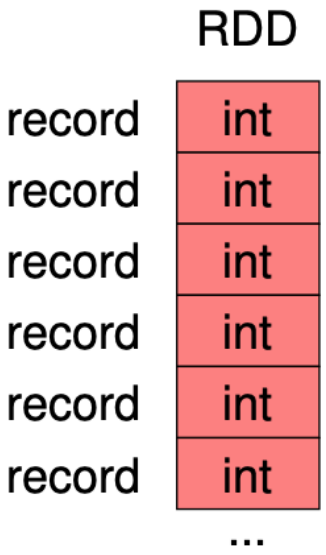
Spark SQL

The Spark core data model: RDD typing

→ RDD: arbitrary types of objects (in Python, Scala, Java, R).

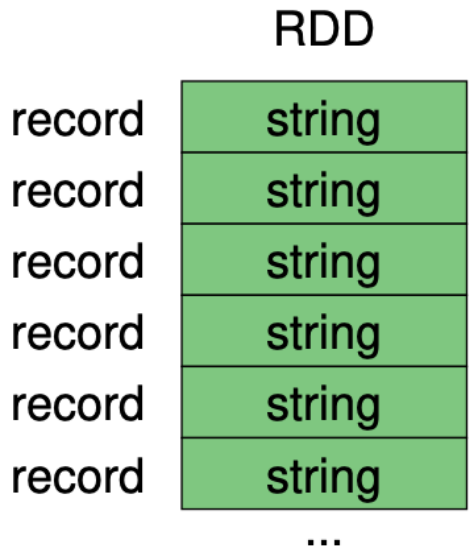
Python RDDs are untyped (can contain multiple record types). The programmer bears responsibility when coding.

```
1 sc.parallelize(xrange(0, 1000), 2)
```



Scala RDDs are statically typed: RDD[int]. The type can be inferred automatically.

```
1 sc.textFile("file.txt")
```



```
1 sc.wholeTextFiles("folder")
```

RDD

record	string	string
record	string	string
record	string	string
record	string	string
record	string	string
record	string	string

...

```
1 sc.wholeTextFiles("folder") \
2     .map(lambda (file, contents):
       contents.split())
```

RDD

record	string	string	string	string	string	string
record	string	string				
record	string					
record	string	string	string	string	string	
record	string	string	string	string		
record	string	string	string	string	string	

...

```
1 sc.wholeTextFiles("folder") \
2     .flatMap(lambda (file, contents): contents.split()) \
3     .map(lambda word: (word, 1))
```

RDD

record	string	int
record	string	int
record	string	int
record	string	int
record	string	int
record	string	int

...

SDL

A big-data database in an RDD?

→ It is possible to create a **table-like RDD**, but naming the columns and enforcing the types in the table schema is the responsibility of your own program.

→ For example, using Python dictionaries in RDD records:

```
1 {'species': 'Elephant', 'num': 1}
2 {'species': 'Snake',    'num': 7}
3 {'species': 'Unicorn',  'num': 13}
4 ...
```

You then need to write your own SQL language, using map and reduce RDD methods

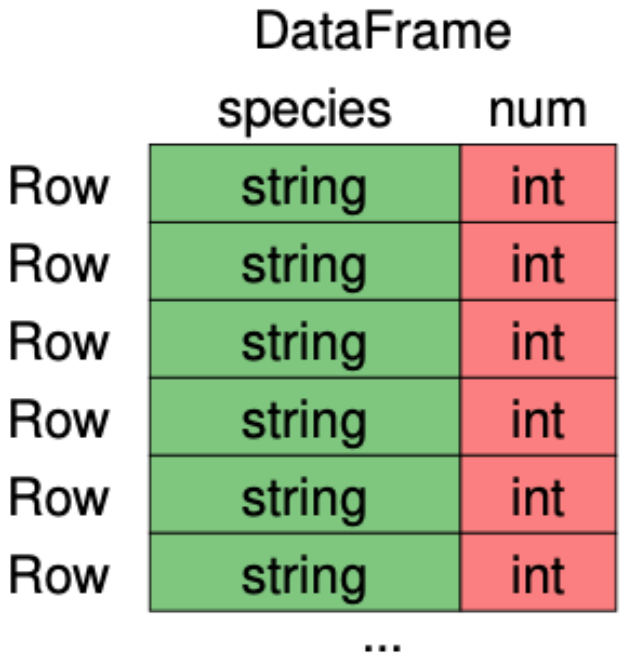
The Spark SQL data model: the DataFrame

- Like an RDD, data is still a distributed, in-memory collection of records

- but the record always has the type `Row()`: `Row(species='Elephant', num=1)`.

- **Dataframe is utalizes RDDs underneath!**

- Unlike an RDD, it keeps a statically typed schema, like a table in a relational database:
  - all Rows have the same schema,
  - each Column has a type,
  - the types are standard SQL data types: boolean, integer, double, decimal, string, struct, array, map, etc.



Creating a DataFrame of Rows (see [API])

```
>>> from pyspark.sql import Row

>>> species = ['Elephant', 'Snake', 'Unicorn']
>>> counts  = [1, 7, 13]

>>> df1 = spark.createDataFrame(
    [Row(species=species[i], num=counts[i]) for i in range(3)])

>>> df1.show()
+---+--------+
|num| species|
+---+--------+
|  1|Elephant|
|  7|   Snake|
| 13| Unicorn|
+---+--------+

>>> df1.printSchema()
root                                    # inferred automatically
 |-- num: long (nullable = true)
 |-- species: string (nullable = true)
```

Column access

Columns are named.

Access to a Column by name (in either a Row or a DataFrame):

- attribute-like r.num, or
- dictionary-like r[’num’]

```
>>> r = df1.take(3)[2]
>>> r
Row(num=13, species=u’Unicorn’)

>>> r.num, r.species
(13, u’Unicorn’)

>>> r[’num’], r[’species’]
(13, u’Unicorn’)
```

Column selection

select(**cols*)

Projects a set of expressions and returns a new **DataFrame**.

Parameters: **cols** – list of column names (string) or expressions (**Column**). If one of the column names is '*', that column is expanded to include all columns in the current DataFrame.

```
>>> df2 = df1.select('species')
>>> df2 = df1.select(df1.species)
>>> df2 = df1.select(df1['species'])

>>> df2.show()
+--------+
| species|
+--------+
|Elephant|
|   Snake|
| Unicorn|
+--------+
```

SDU

... also takes arithmetic **expressions**, in an own domain-specific language (DSL):
+, -, *, /, <, >, ==, !=

```
>>> df2 = df1.select(df1.num * 2)
>>> df2.show()
+----------+
|(num * 2)|
+----------+
|         2|
|        14|
|        26|
+----------+
```

Column renaming

alias(**alias*, ***kwargs*)

Returns this column aliased with a new name or names (in the case of expressions that return more than one column, such as explode).

Parameters: • **alias** – strings of desired column names (collects all positional arguments passed)

```
>>> df2 = df1.select((df1.num * 2).alias('mult'))
>>> df2.show()
+----+
|mult|
+----+
|   2|
|  14|
|  26|
+----+
```

SDU

Row filtering

`filter`(*condition*)

Filters rows using the given condition.

`where()` is an alias for **`filter()`**.

Parameters: **condition** – a **`Column`** of **`types.BooleanType`** or a string of SQL expression.

```
>>> df3 = df1.filter(df1.num > 2)
>>> df3.show()
+---+-------+
|num|species|
+---+-------+
|  7|  Snake|
| 13|Unicorn|
+---+-------+
```

SDU

rlike(*other*)

Return a Boolean **Column** based on a regex match.

Parameters: **other** – an extended regex expression

```
>>> df3 = df1.filter(df1.species.rlike('.*[ch]orn.*'))
>>> df3.show()
+---+-------+
|num|species|
+---+-------+
| 13|Unicorn|
+---+-------+
```

Reading JSON as a DataFrame

json(*path*, *schema=None*, *primitivesAsString=None*, *prefersDecimal=None*, *allowComments=None*, *allowUnquotedFieldNames=None*, *allowSingleQuotes=None*, *allowNumericLeadingZero=None*, *allowBackslashEscapingAnyCharacter=None*, *mode=None*, *columnNameOfCorruptRecord=None*, *dateFormat=None*, *timestampFormat=None*, *multiLine=None*)

Loads JSON files and returns the results as a **DataFrame**.

JSON Lines (newline-delimited JSON) is supported by default. For JSON (one record per file), set the `multiLine` parameter to `true`.

If the `schema` parameter is not specified, this function goes through the input once to determine the input schema.

```
>>> df1 = spark.read.json("/data/doina/UCSD-Amazon-Data/
    meta_Books_sample.json.gz", mode="DROPMALFORMED")
>>> df1.printSchema()                          # inferred automatically!

>>> df2 = df1.select('asin', 'title', 'salesRank')
>>> df2.count()
>>> df3 = df2.filter(df2.salesRank.Books < 1000)
>>> df3.count()
>>> df4 = df2.filter(df2.title.rlike('.*Unicorn.*'))
>>> rdd = df2.rdd.map(lambda r: r.title.lower())
```

SDU

The Spark SQL programming model is:

→ Relational

You can write declarative queries:

```
1 df.select().distinct().where().where()
```

→ equivalent to the SQL

→ `SELECT .. DISTINCT FROM .. WHERE .. WHERE ..`

→ Procedural

You can split a query into steps, name and debug

→ intermediate results:

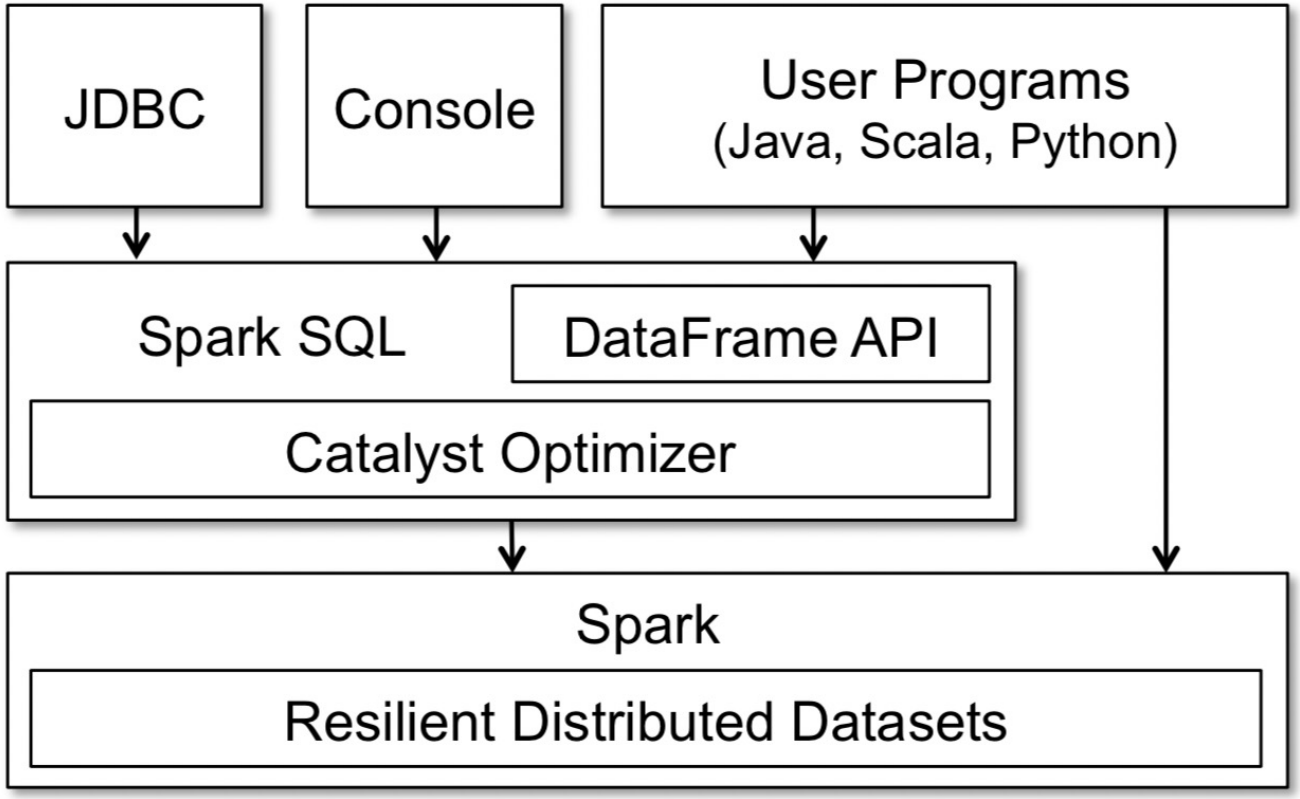
```
1 df1 = spark.read.json()
2 df2 = df1.select().cache()
3 df3 = df2.filter()
4 df4 = df2.filter()
```

→ Functional

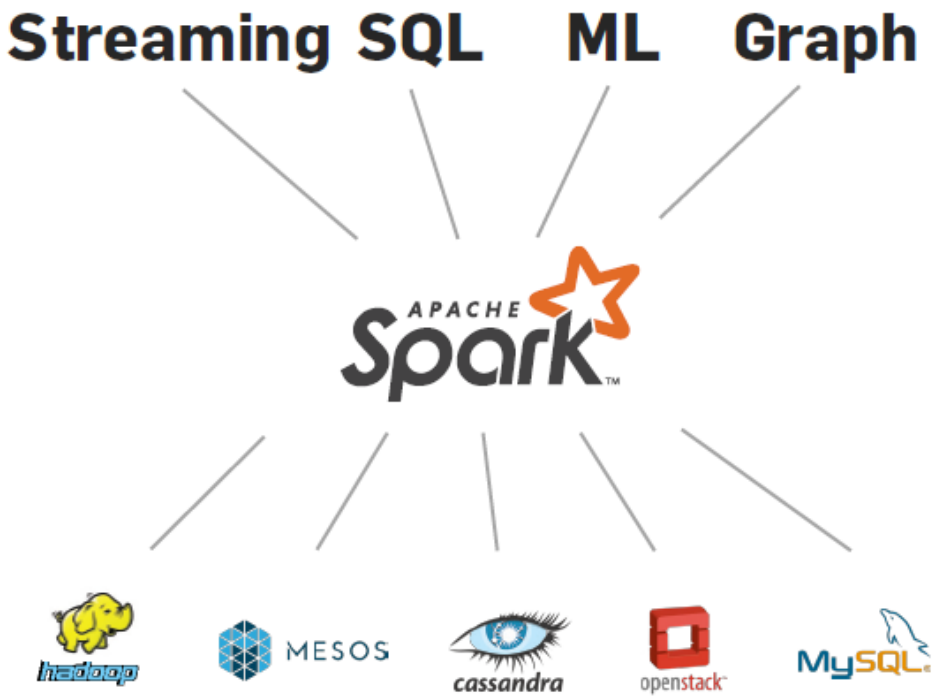
(Python is in a grey area)

The functional look is inherited from the core Spark API, which in turn inherited it from the MapReduce framework.

Spark SQL is still powered by RDDs:



Integration with data-storage systems. Higher-level libraries



Spark is agnostic towards the **storage system** (can read from many),
Mlib implements 50+ algorithms for **distributed model training.**

Recommended practical walkthrough

→ https://www.youtube.com/watch?v=3-pnWVWyH-s
  → Contains explanations of:
    → Inference
    → Querying (where, select, etc.)
    → Aggregation
    → Group By
    → Order By

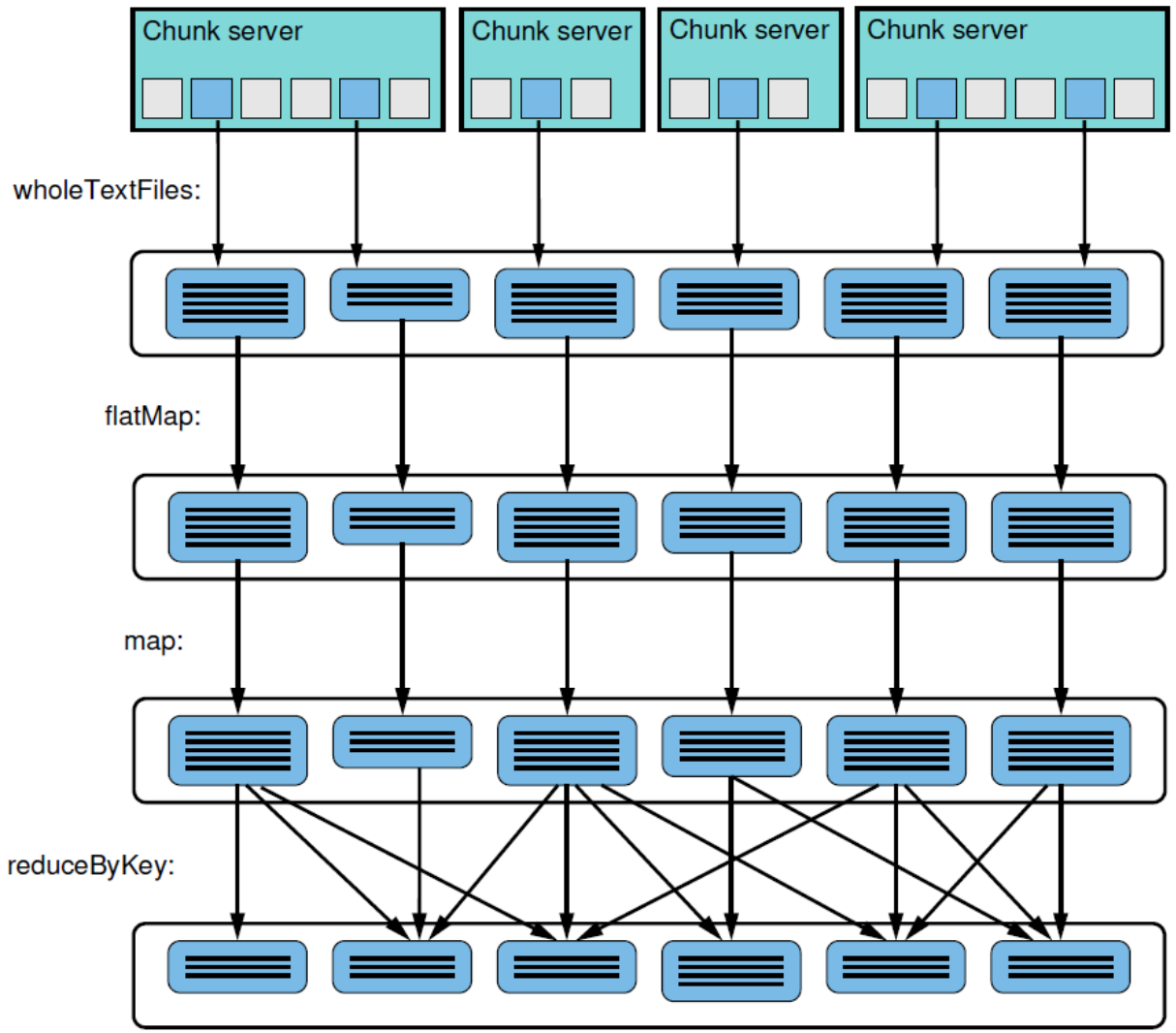
Apache Spark Streaming

SDU

Apache Spark Streaming:

- → **Spark put into a streaming context**
- → **Spark Streaming can read data from**
  - → **HDFS**
  - → **Flume**
  - → **Kafka**
  - → **Twitter**
  - → **ZeroMQ**
  - → **Custom data sources**

Traditional spark example



(Word count)

Spark Streaming:

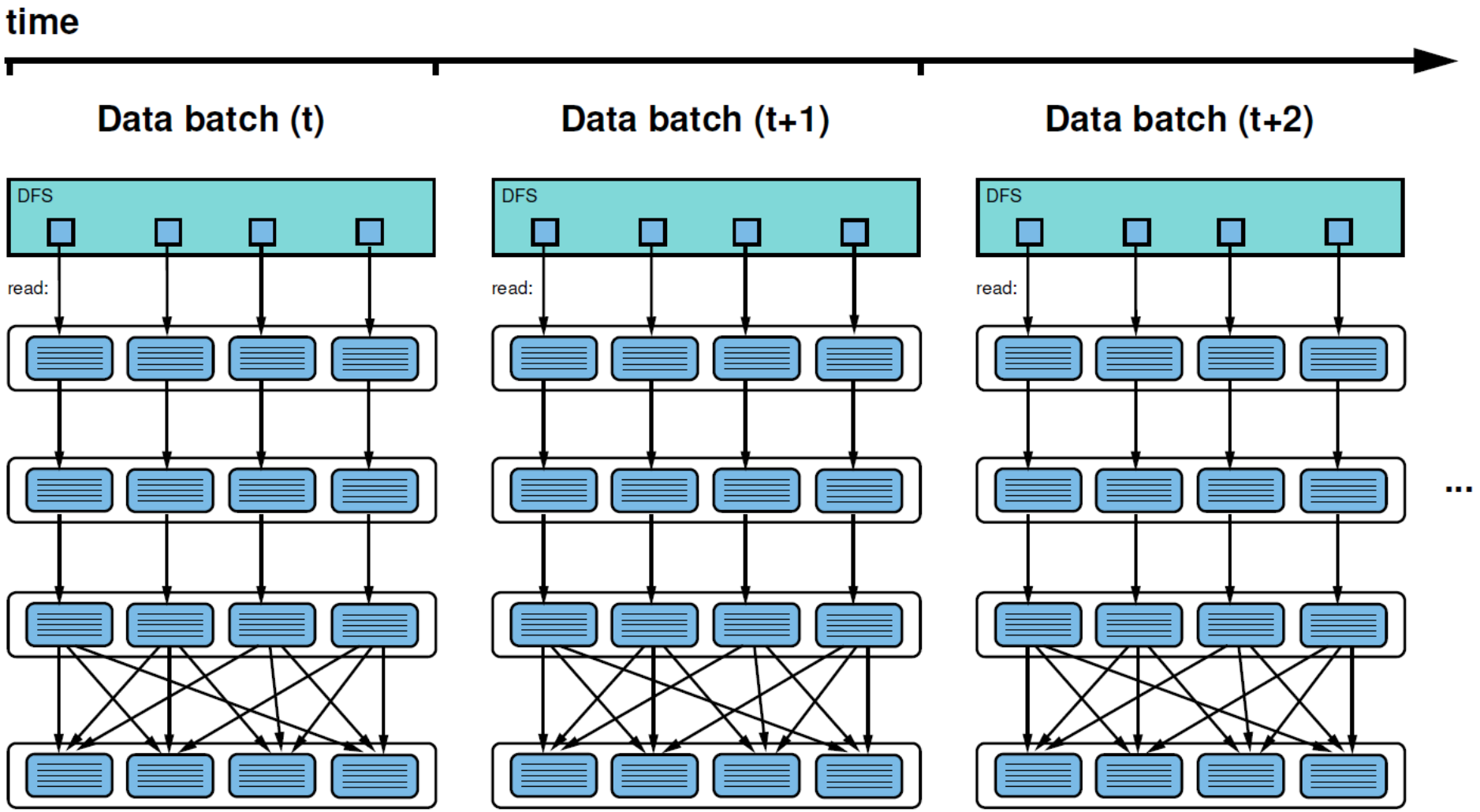
→Streaming input data is **batched** (discretized):

→At small time interval (every 1 second);

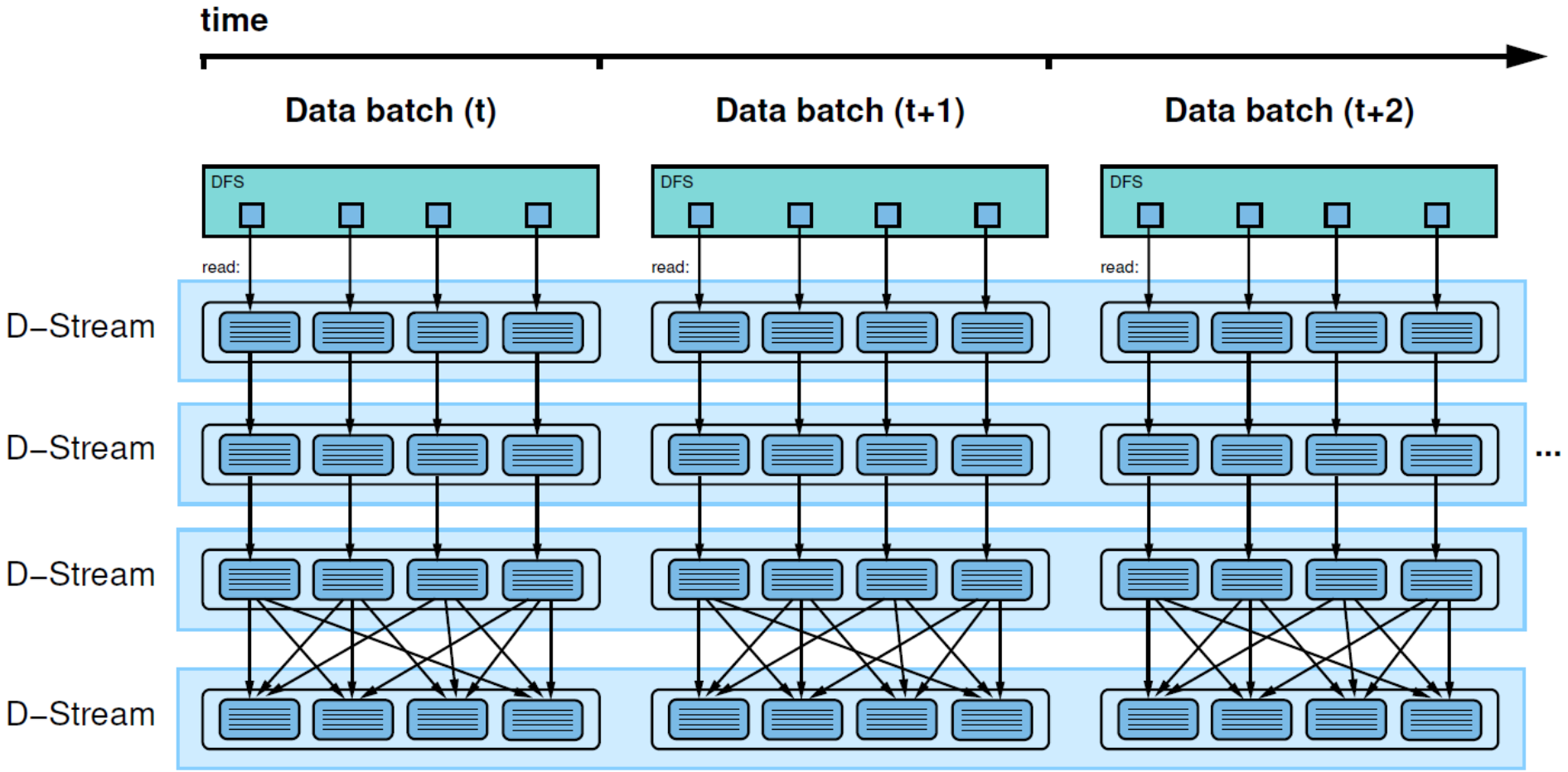
→The smaller this interval, the lower the latency;

→Each batch is processed with Spark.

Spark Streaming Batches



The D-Stream



The discretized stream (D-Stream): a sequence of immutable, partitioned datasets (RDDs) of the same type.

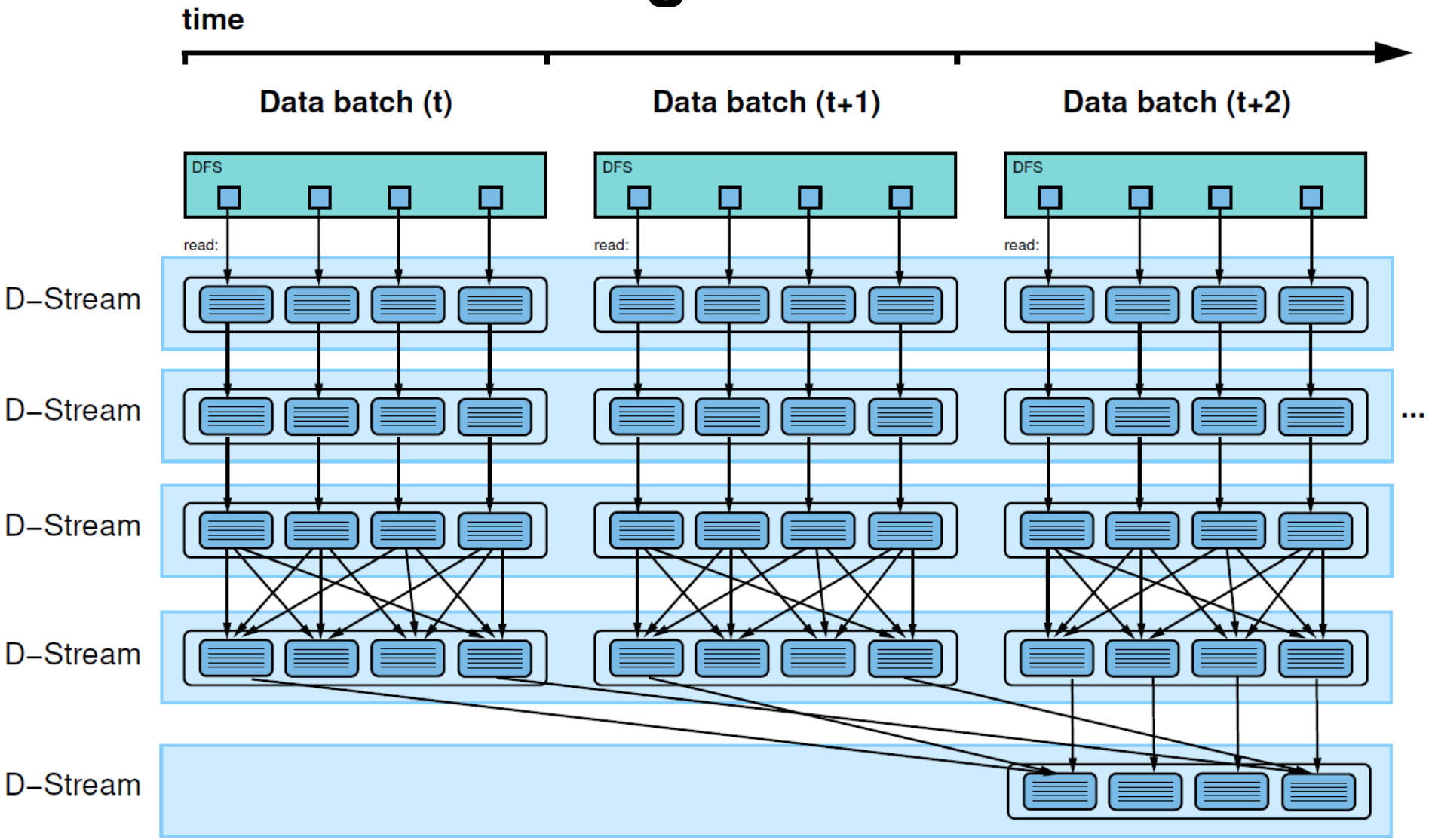
SDU

D-Stream and RDD

→Each D-Steam periodically generates an RDD, either from live data or by transforming the RDD generated by a parent D-Stream

→Then groups all records from past sliding windows into the RDD

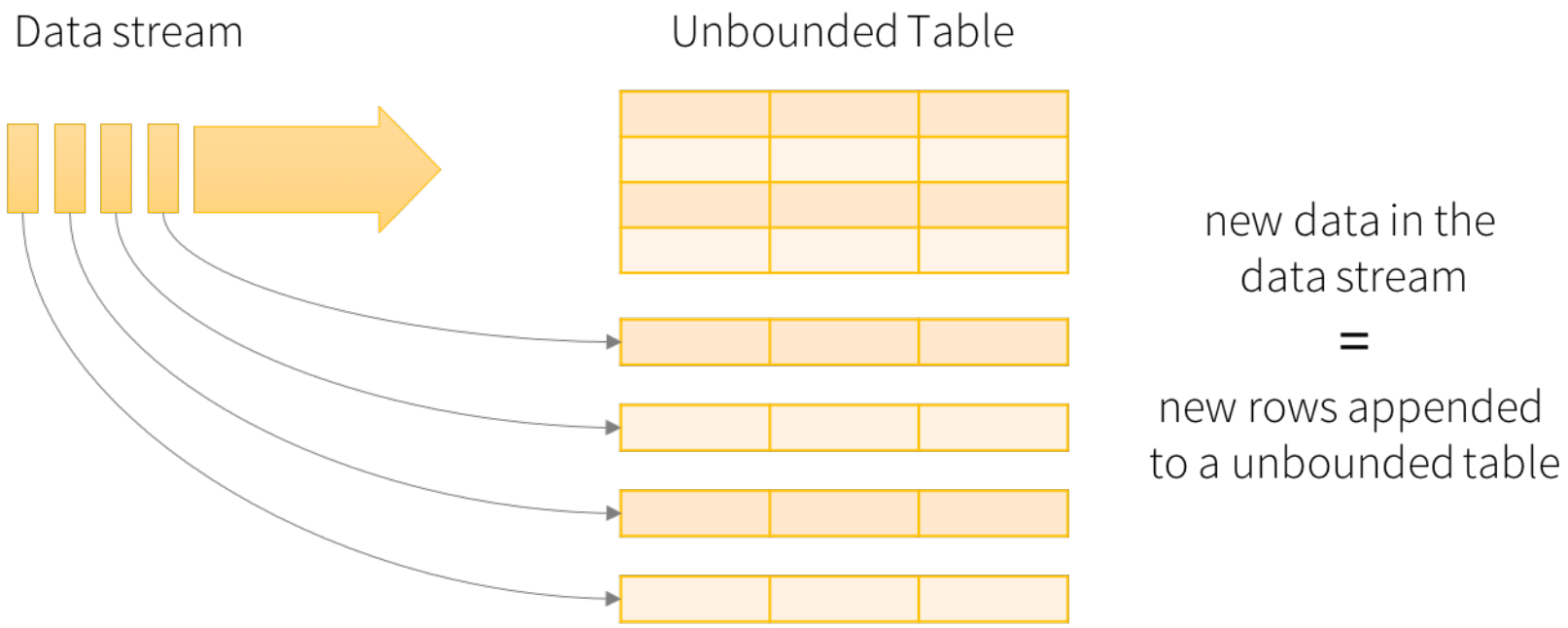
The Sliding Window



Spark Streaming: Guarantees

→D-Streams (and RDDs) track their **lineage** (the graph of transformations)
  →at the level of partitions within each RDD
  →when a node fails, its RDD partitions are rebuilt on other machines from the original input data stored in the cluster.
→D-Streams provide consistent, exactly-once processing.

Spark Structured Streaming vs Spark Streaming



Data stream as an unbounded table

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
Recommendation: Use Structed Streaming!!!!

Exercises!