[544] Spark Internals and Performance

Tyler Caraza-Harter

Learning Objectives

- select an appropriate caching level based on resources available

- identify cases where hash partitioning is necessary (instead of regular partitioning) to bring "related" data together

- describe three major Spark optimization related to groups/aggregates: partial aggregates, partition coalescing, and Parquet bucketing

- describe two major distributed join algorithms (BHJ and SMJ) and the tradeoffs between them

Outline

Schema Inference

Collecting Data

Caching

Grouping

Joining

With Schema Inference

```
df = (spark.read.format("csv")
      .option("header", True)
      .option("inferSchema", True)
      .load("hdfs://nn:9000/sf.csv"))
```

- 17 tasks, 33 seconds
- reads whole file to guess types

Without Schema Inference

```
df = (spark.read.format("csv")
      .option("header", True)
      .load("hdfs://nn:9000/sf.csv"))
```

- 1 task, 0.3 seconds
- only reads header
- everything is a string

```
df = (spark.read.format("csv")
      .schema("????")
      .load("hdfs://nn:9000/sf.csv"))
```

- 0 tasks, 0.04 seconds
- need to manually specify types

```
df = (spark.read.format("parquet")
      .load("hdfs://nn:9000/sf.parquet"))
```

- 1 tasks, 0.2 seconds
- only reads schema info

Outline
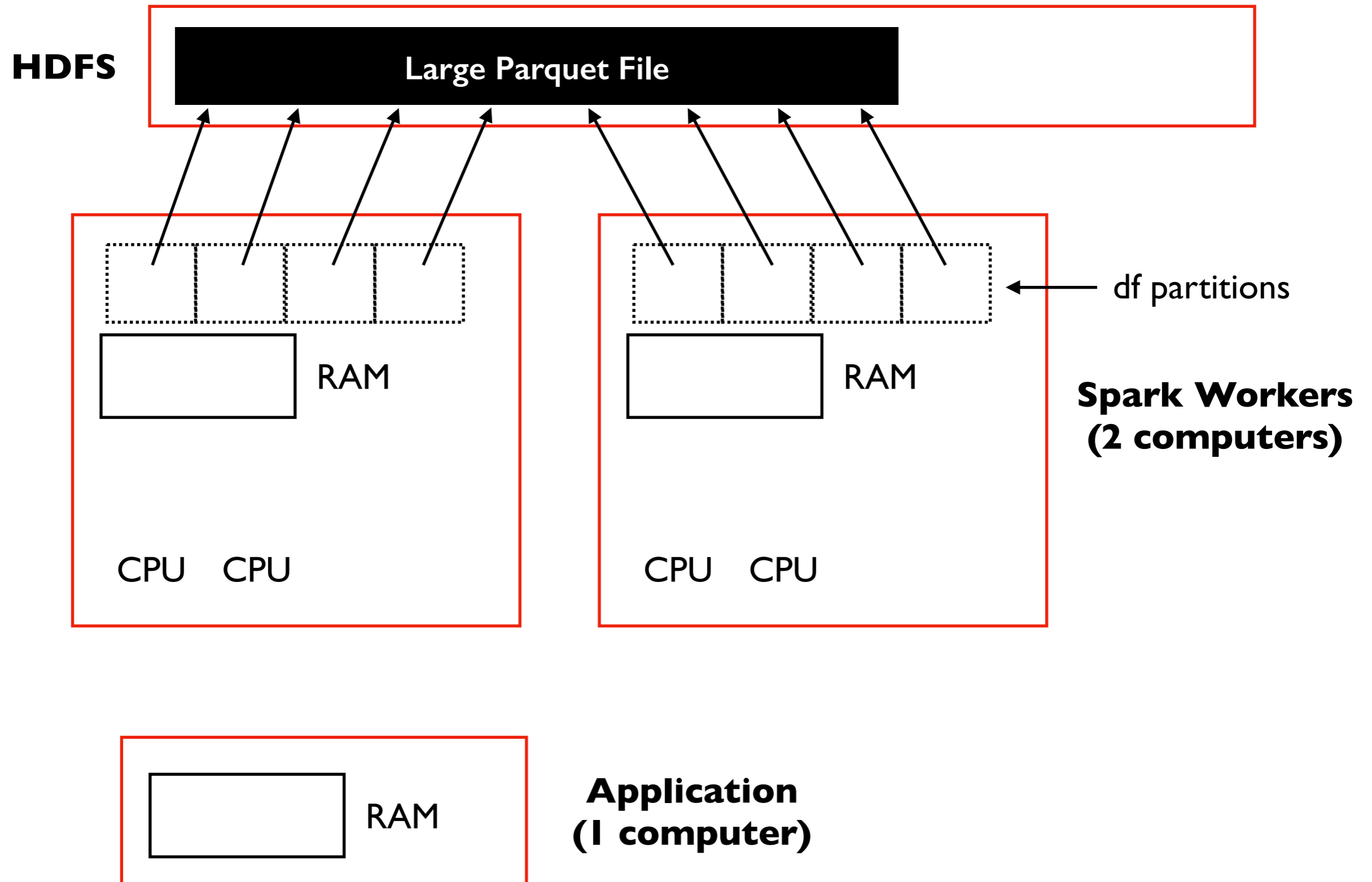
Schema Inference

Collecting Data

Caching

Grouping

Joining

Collecting Data (OK)

```
# df refers to parquet file
# results = df.where(???).collect()
results = df.where(???).toPandas()
```

HDFS

Large Parquet File

df partitions

RAM

RAM

**Spark Workers
(2 computers)**

CPU   CPU

CPU   CPU

RAM

**Application
(1 computer)**

Collecting Data (OK)

```
# df refers to parquet file
# results = df.where(???).collect()
results = df.where(???).toPandas()
```

HDFS

Large Parquet File

RAM

df partitions

task1   *task2*

CPU   CPU

RAM

task3   *task4*

CPU   CPU

**Spark Workers
(2 computers)**

RAM

**Application
(1 computer)**

Collecting Data (OK)

```
# df refers to parquet file
# results = df.where(???).collect()
results = df.where(???).toPandas()
```

HDFS

Large Parquet File

df partitions

RAM

RAM

**Spark Workers
(2 computers)**

task5 *task6*

CPU CPU

task7 *task8*

CPU CPU

RAM

**Application
(1 computer)**

Collecting Data (bad)

```
# df refers to parquet file
# results = df.where(???).collect()
results = df.where(???).toPandas()
```

HDFS

```
Large Parquet File
```

df partitions

RAM

RAM

**Spark Workers
(2 computers)**

CPU   CPU

CPU   CPU

RAM

**Application
(1 computer)**

Collecting Data (bad)

```
# df refers to parquet file
# results = df.where(???).collect()
results = df.where(???).toPandas()
```

HDFS

Large Parquet File

df partitions

RAM

RAM

**Spark Workers
(2 computers)**

task1   *task2*

CPU   CPU

task3   *task4*

CPU   CPU

out of memory! (only 2 of 8 partitions fit)

RAM

**Application
(1 computer)**

Outline

Schema Inference

Collecting Data

Caching
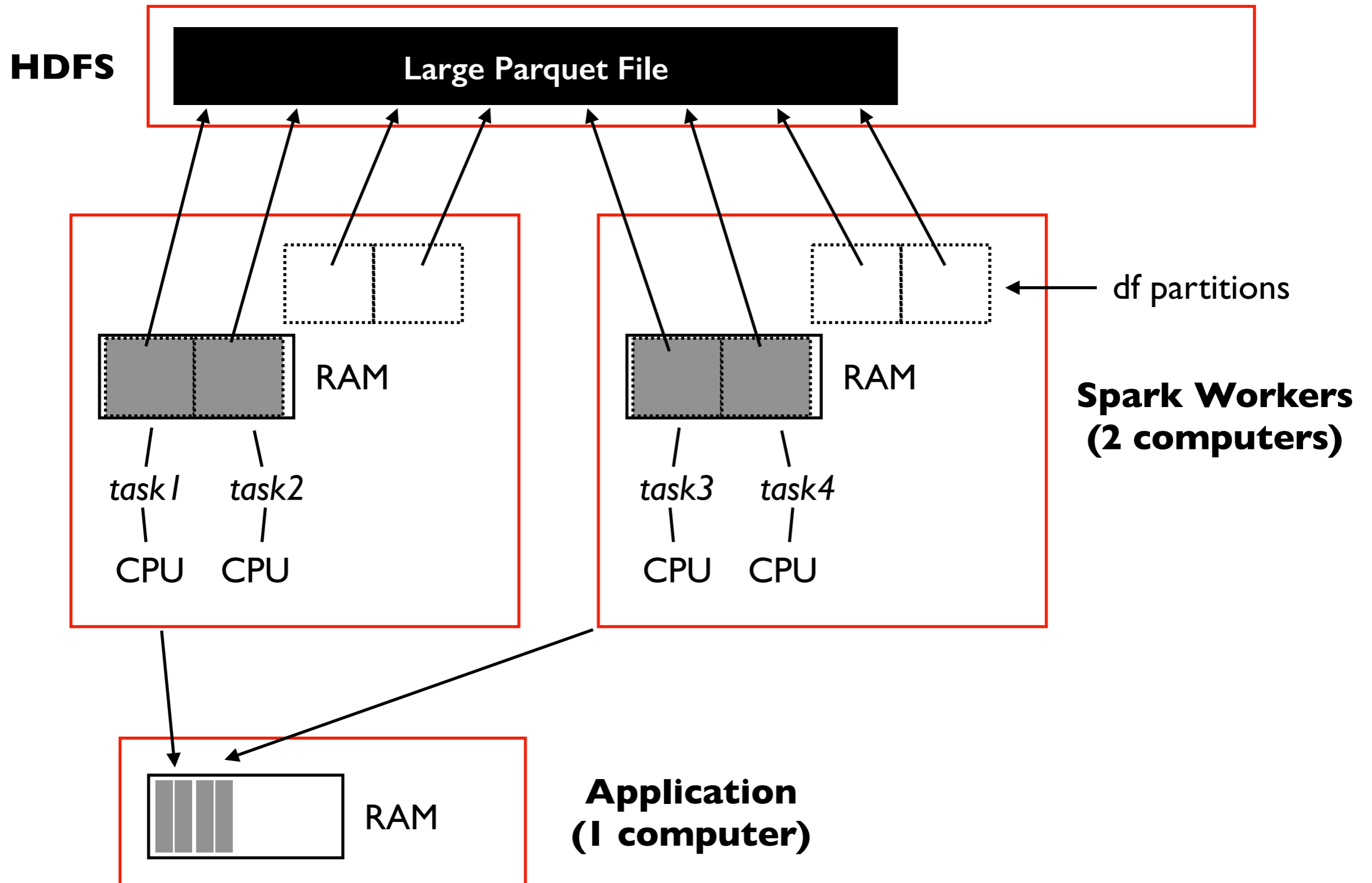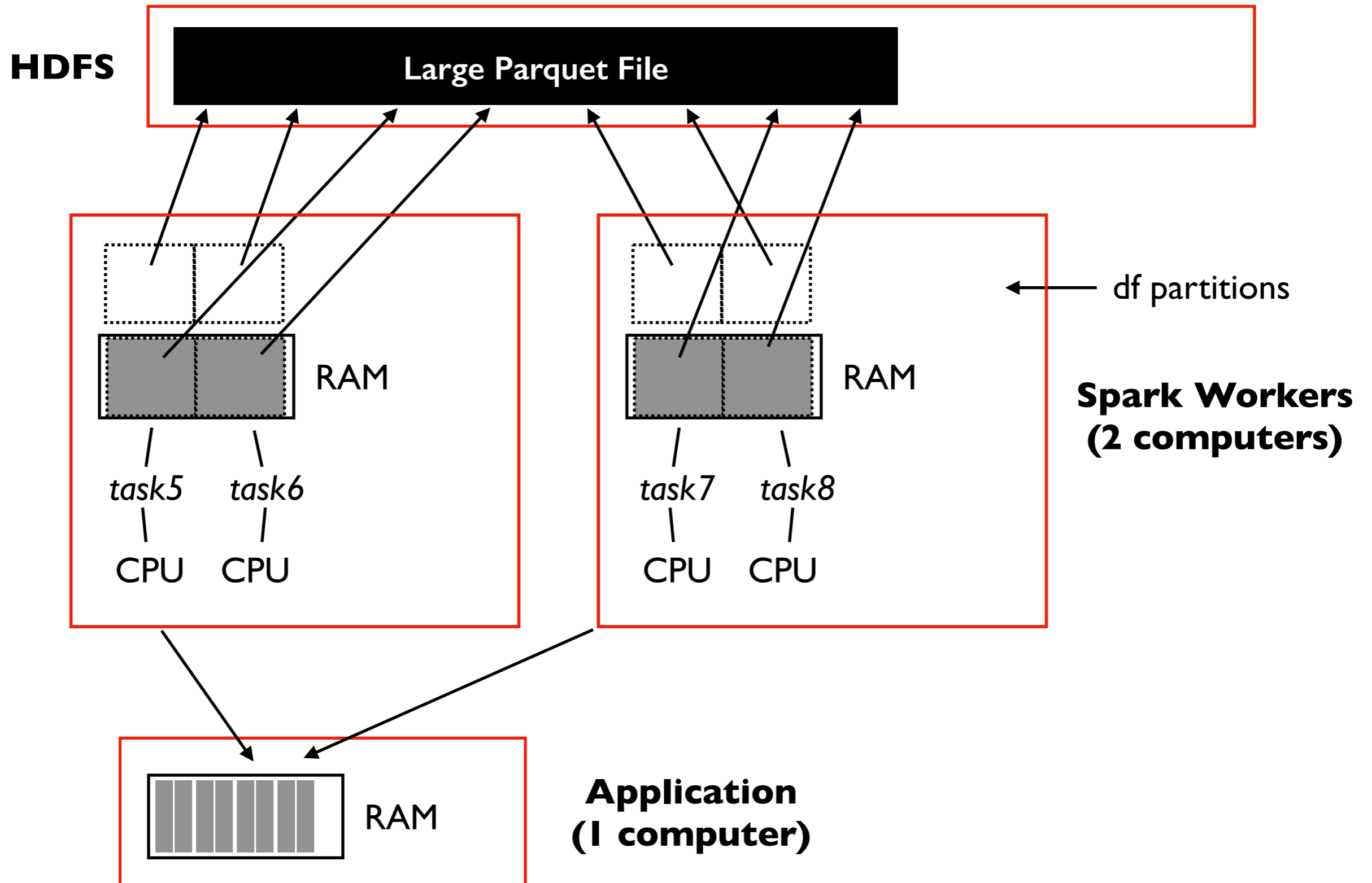
Grouping

Joining

Persisting/Caching

HDFS

Large Parquet File

df partitions

df2 partitions

RAM

Local FS

RAM

Local FS

Spark Workers (2 computers)
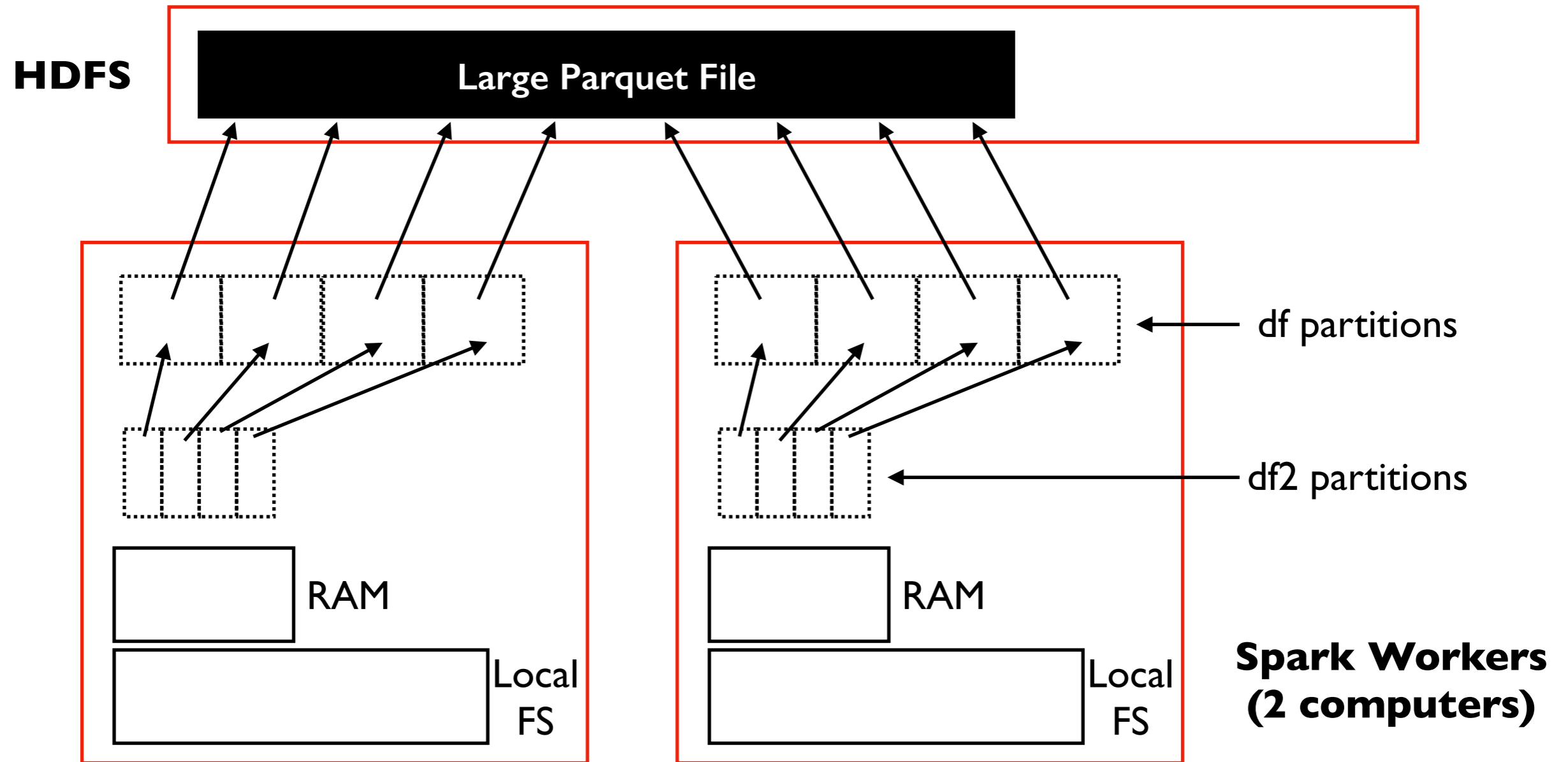
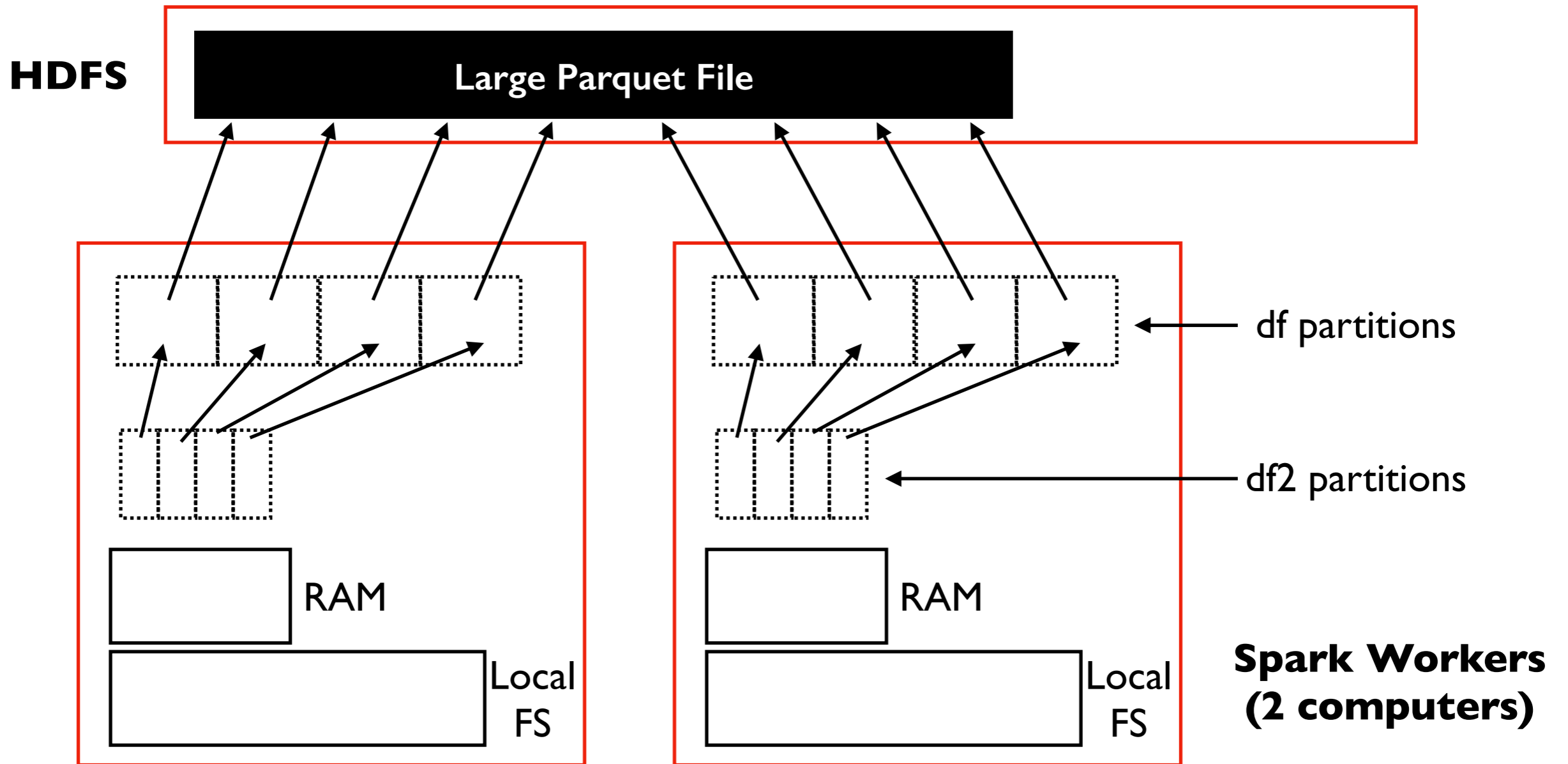# df refers to parquet file
df2 = df.where(???)

Scenario: want to do lots of computations on df2
Goal: avoid repeatedly reading HDFS and filtering df

Persisting/Caching

HDFS

Large Parquet File

df partitions

df2 partitions

RAM

Local FS

RAM

Local FS

Spark Workers (2 computers)

```
from pyspark.storagelevel import StorageLevel
df2 = df.where(???)

df2.persist(StorageLevel.????)
```
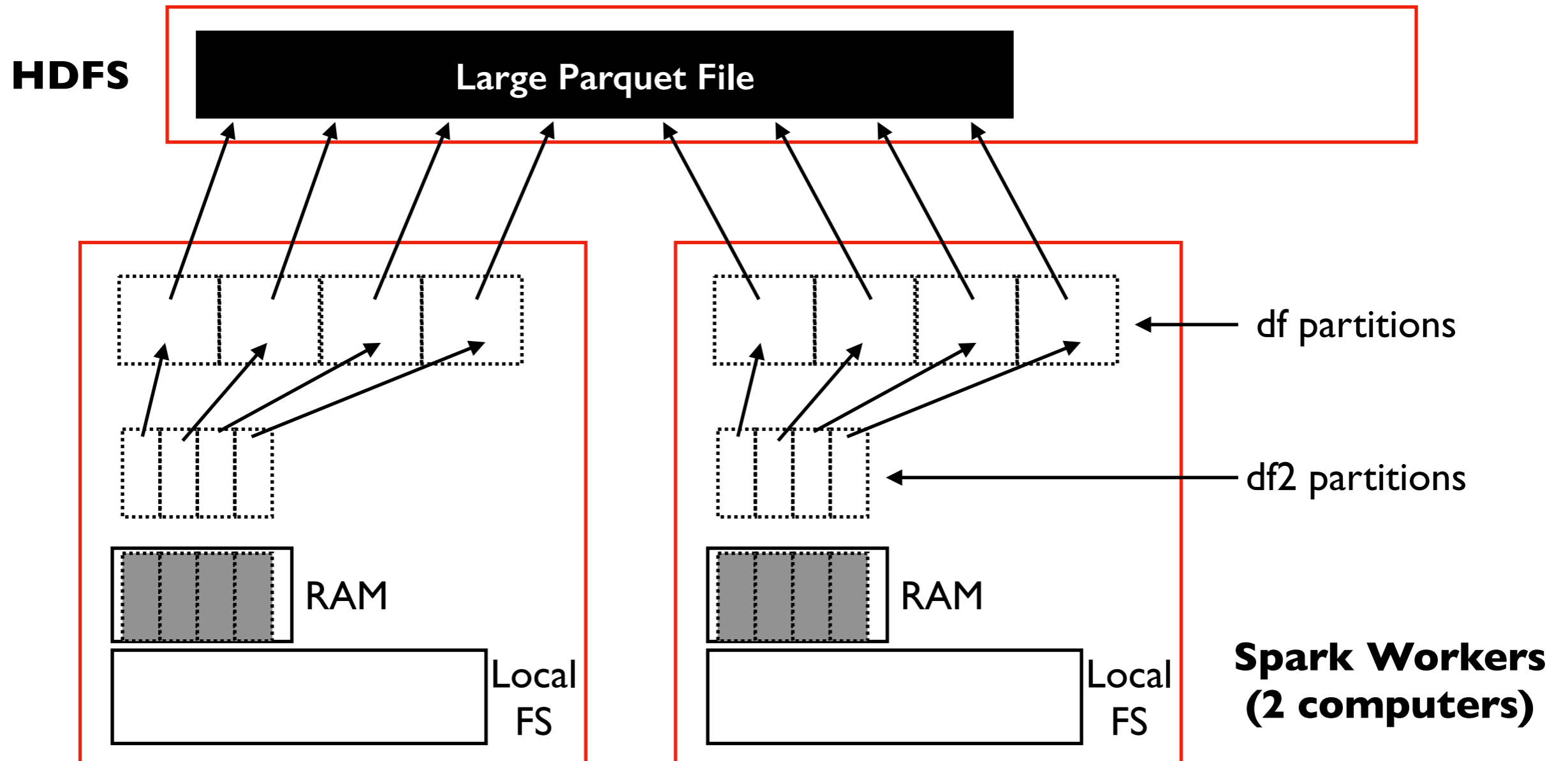
Persist levels
- MEMORY_ONLY
- MEMORY_ONLY_SER
- DISK_ONLY
- *others...*

Persisting/Caching

HDFS

Large Parquet File

df partitions

df2 partitions

RAM

Local FS

RAM

Local FS

Spark Workers (2 computers)

```
from pyspark.storagelevel import StorageLevel
df2 = df.where(???)

df2.persist(StorageLevel.????) # df.cache() is a shortcut
```

Persist levels
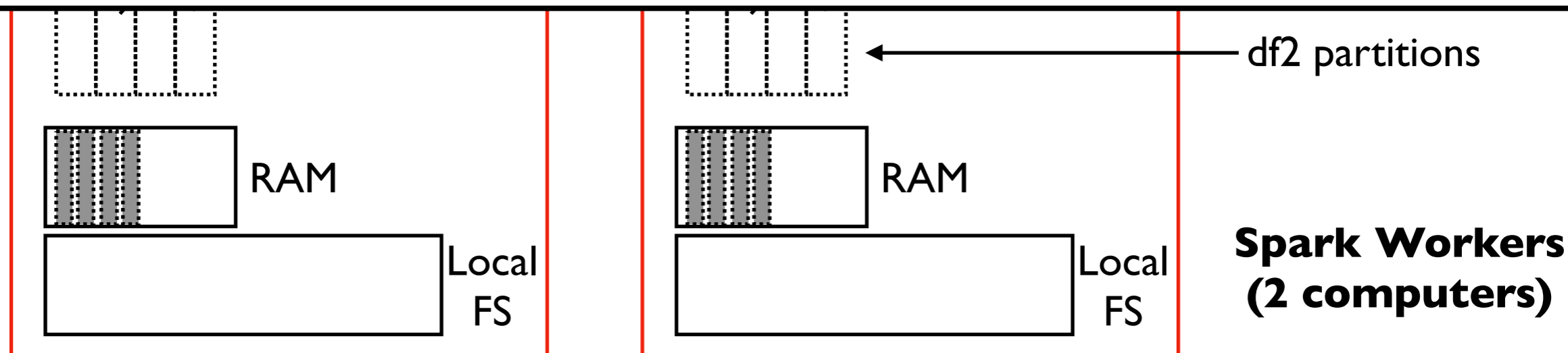- **MEMORY_ONLY**
- MEMORY_ONLY_SER
- DISK_ONLY
- *others...*

Documentation Snippet (https://spark.apache.org/docs/2.2.2/tuning.html#memory-tuning)

By default, Java objects are fast to access, but can easily consume a factor of 2-5x more space than the "raw" data inside their fields. This is due to several reasons:

- Each distinct Java object has an "object header", which is about 16 bytes and contains information such as a pointer to its class. For an object with very little data in it (say one `Int` field), this can be bigger than the data.
- Java `String`s have about 40 bytes of overhead over the raw string data (since they store it in an array of `Char`s and keep extra data such as the length), and store each character as *two* bytes due to `String`'s internal usage of UTF-16 encoding. Thus a 10-character string can easily consume 60 bytes.
- Common collection classes, such as `HashMap` and `LinkedList`, use linked data structures, where there is a "wrapper" object for each entry (e.g. `Map.Entry`). This object not only has a header, but also pointers (typically 8 bytes each) to the next object in the list.
- Collections of primitive types often store them as "boxed" objects such as `java.lang.Integer`.

df2 partitions

RAM

Local FS

RAM

Local FS

Spark Workers (2 computers)

from pyspark.storagelevel import StorageLevel
df2 = df.where(???)

df2.persist(StorageLevel.????)

Persist levels
- MEMORY_ONLY
- MEMORY_ONLY_SER
- DISK_ONLY
- *others...*

Documentation Snippet (https://spark.apache.org/docs/2.2.2/tuning.html#memory-tuning)

By default, Java objects are fast to access, but can easily consume a factor of 2-5x more space than the "raw" data inside their fields. This is due to several reasons:

- Each distinct Java object has an "object header", which is about 16 bytes and contains information such as a pointer to its class. For an object with very little data in it (say one `Int` field), this can be bigger than the data.

- Java S_____
  array c
  to `Str`
  bytes.

- Comm
  there i
  also p

- Collec

More documentation (https://spark.apache.org/docs/2.2.2/tuning.html#memory-tuning)

When your objects are still too large to efficiently store despite this tuning, a much simpler way to reduce memory usage is to store them in serialized form, using the serialized StorageLevels in the RDD persistence API, such as MEMORY_ONLY_SER. Spark will then store each RDD partition as one large byte array. The only downside of storing data in serialized form is slower access times, due to having to deserialize each object on the fly.

df2 partitions

RAM

Local FS

RAM

Local FS

**Spark Workers
(2 computers)**

```
from pyspark.storagelevel import StorageLevel
df2 = df.where(???)


df2.persist(StorageLevel.????)
```

Persist levels
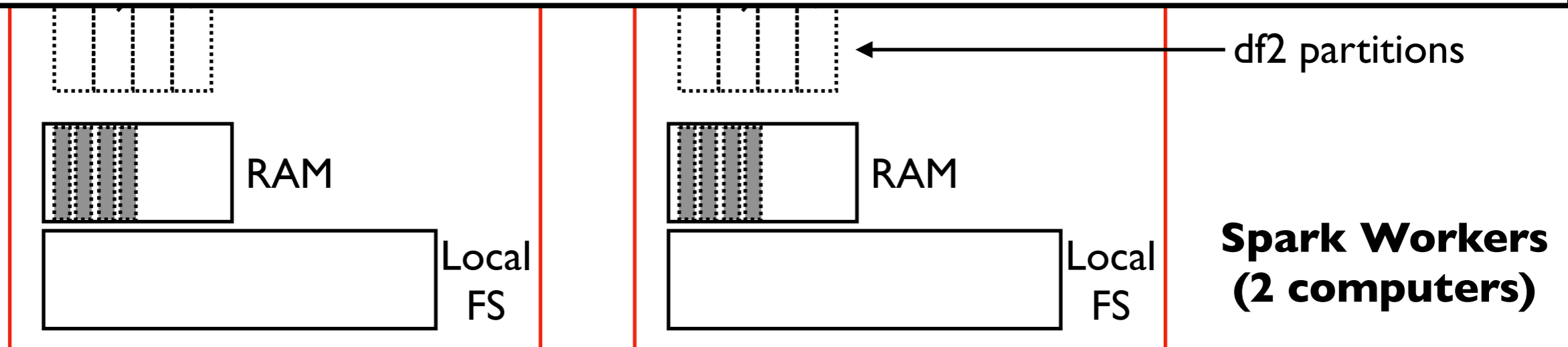- MEMORY_ONLY
- **MEMORY_ONLY_SER**
- DISK_ONLY
- *others...*

Persisting/Caching

HDFS

Large Parquet File

df partitions

df2 partitions

RAM

serialized

Local FS

RAM

serialized

Local FS

Spark Workers (2 computers)

```
from pyspark.storagelevel import StorageLevel
df2 = df.where(???)

df2.persist(StorageLevel.????)
```

Persist levels
- MEMORY_ONLY
- MEMORY_ONLY_SER
- DISK_ONLY
- *others...*

Persisting/Caching

HDFS

Large Parquet File

df partitions

df2 partitions

RAM

RAM

Local FS

Local FS

Spark Workers (2 computers)

```
from pyspark.storagelevel import StorageLevel
df2 = df.where(???)

df2.persist(StorageLevel.????)
```
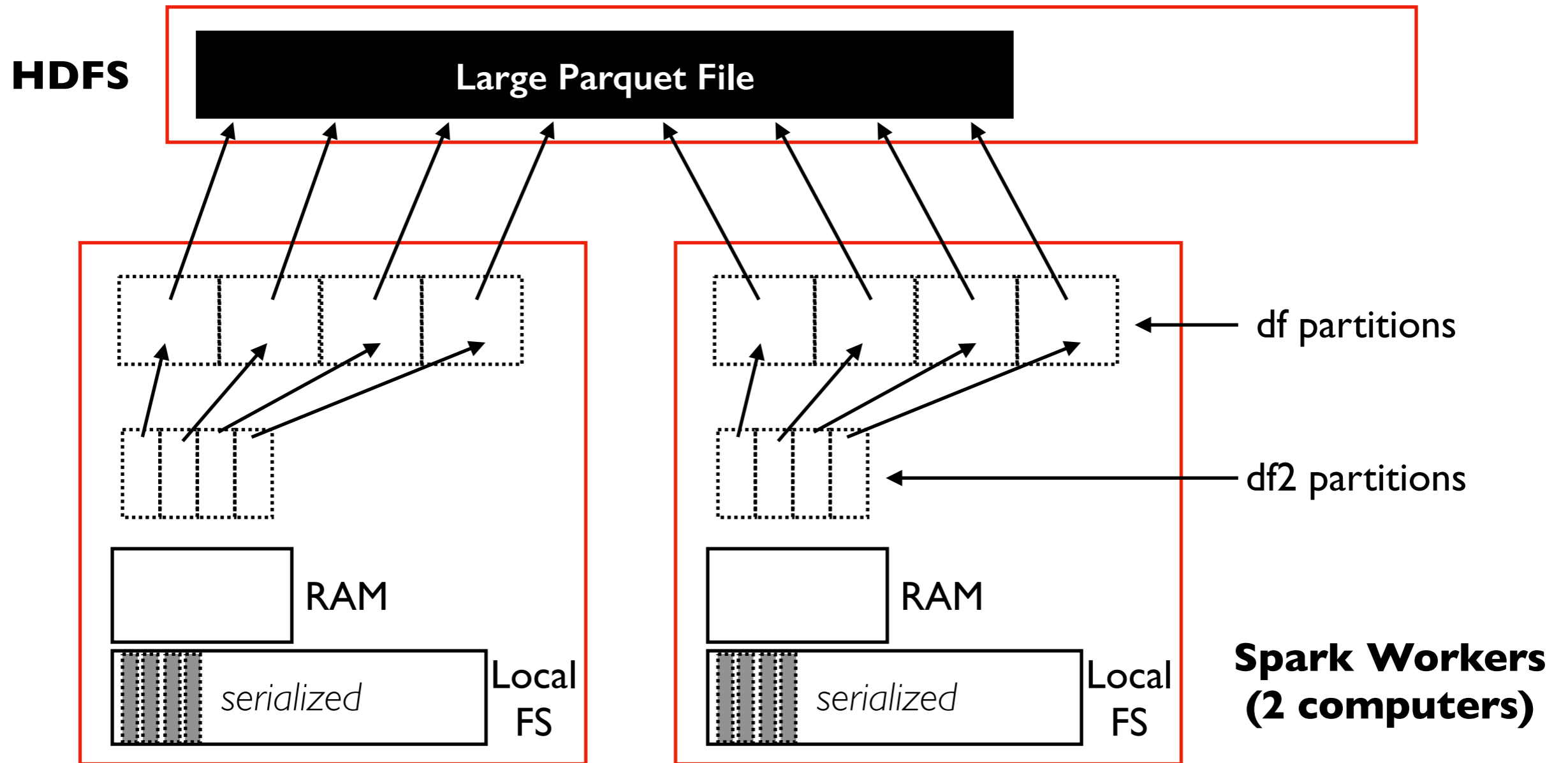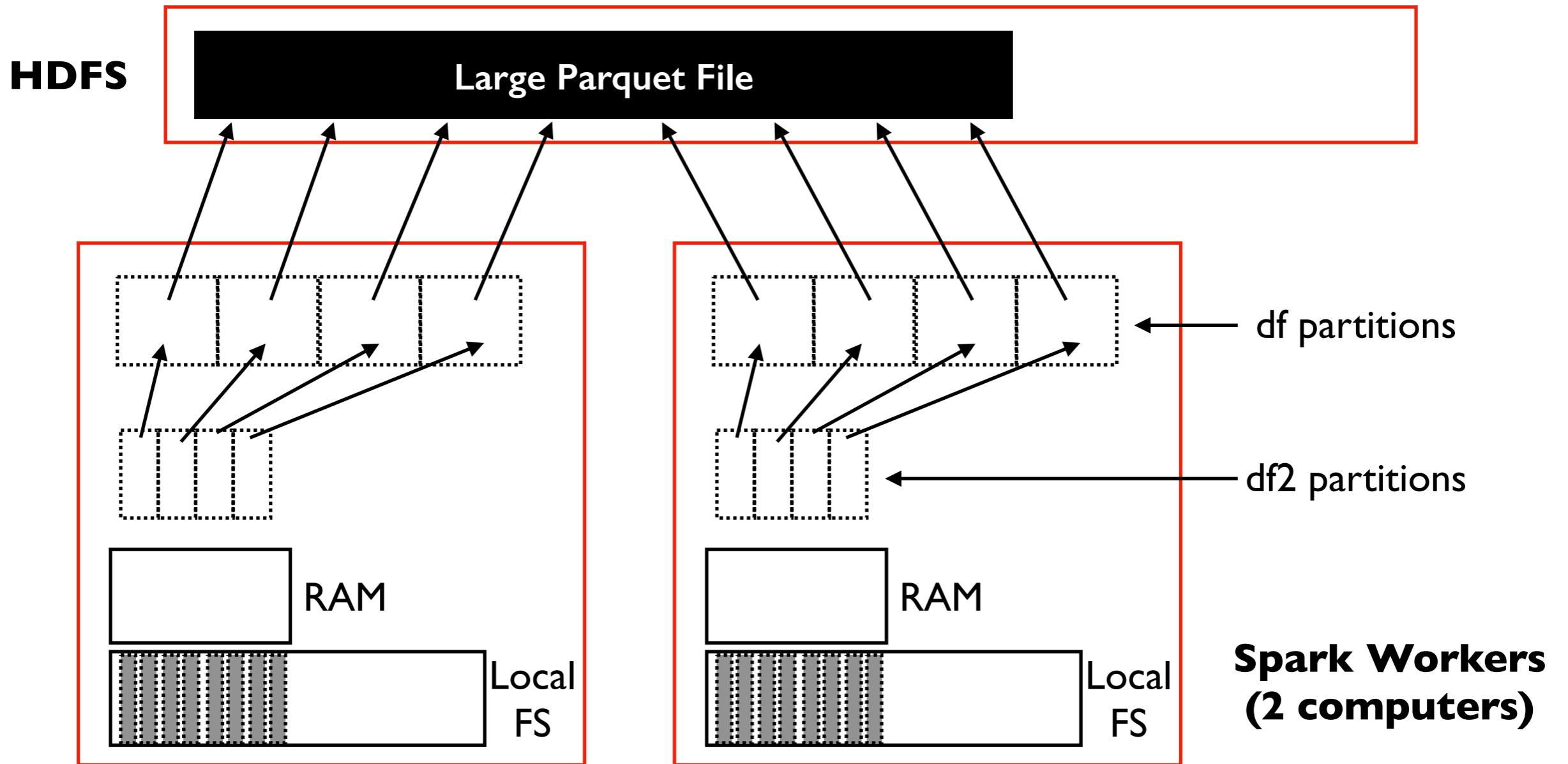
Persist levels (2x replication)
- MEMORY_ONLY_2
- MEMORY_ONLY_SER_2
- DISK_ONLY_2
- *others...*

Replication benefits
- two choices for where to run task without needing network transfer
- if a worker dies, no need to re-compute cached data

Replication downside
- uses twice as much space
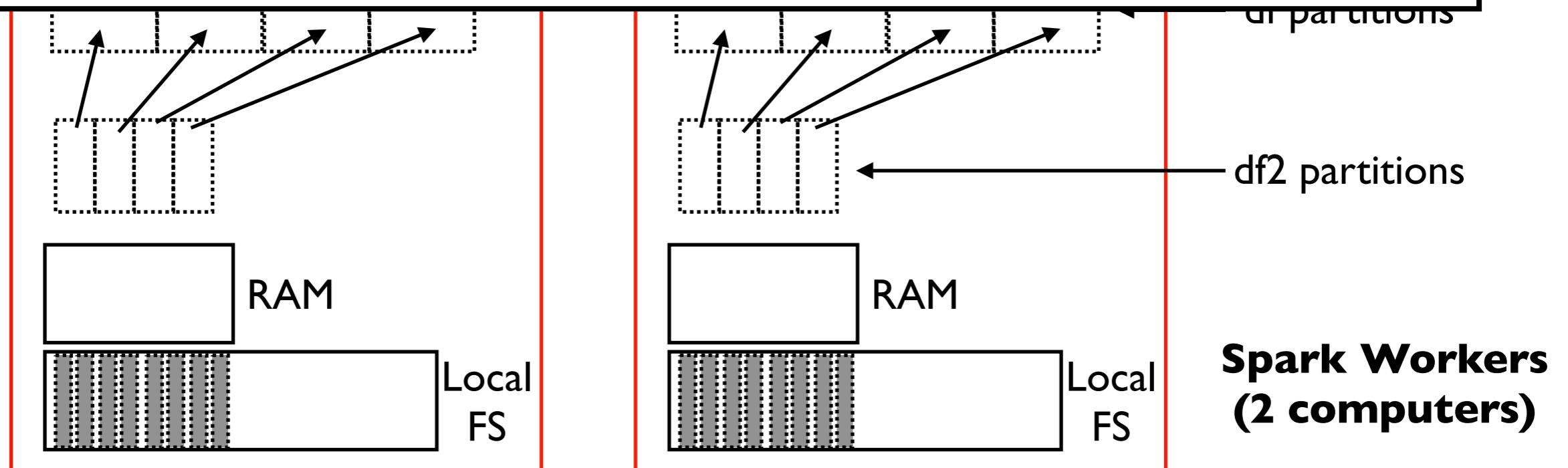
df partitions

df2 partitions

RAM

Local
FS

RAM

Local
FS

**Spark Workers
(2 computers)**

```
from pyspark.storagelevel import StorageLevel
df2 = df.where(???)

df2.persist(StorageLevel.????)
```

Persist levels (2x replication)
- MEMORY_ONLY_2
- MEMORY_ONLY_SER_2
- DISK_ONLY_2
- *others...*

TopHat, Demos...

Outline

Schema Inference

Collecting Data

Caching

Grouping

Joining

GROUPS, AGGREGATES

GROUPS

AGGREGATES
(1 row per group)
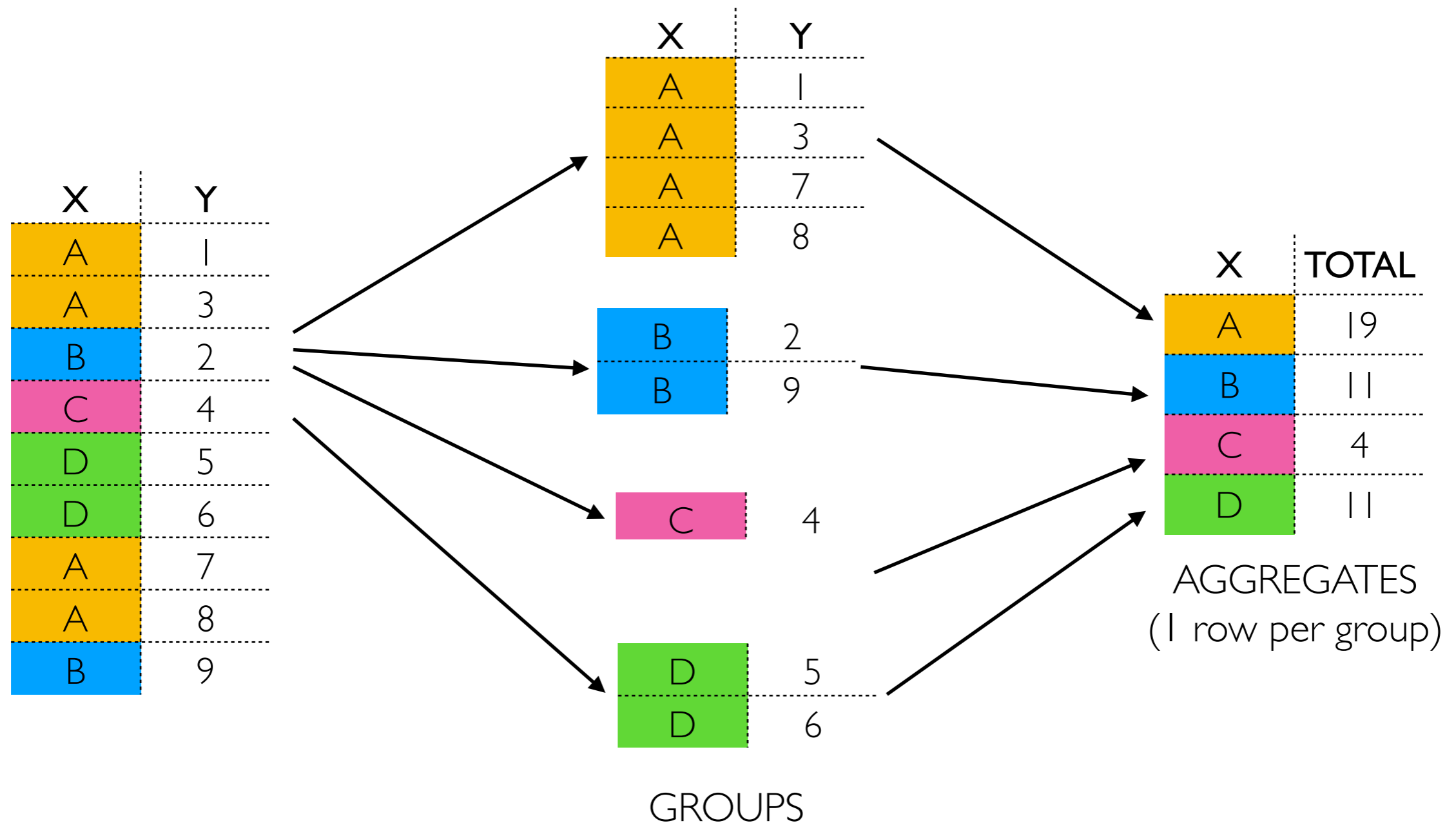
Logically
- lots of groups
- need to bring related (grouped) data together
- stats per group

Spark: Physical Execution on Partitions

□ partition

machine 1

X	Y
A	1
A	3

X	Y
B	2
C	4

X	Y
A	1
A	3
A	7
A	8

X	TOTAL
A	19

machine 2

X	Y
D	5
D	6
A	7

X	Y
A	8
B	9

B	2
B	9

B	11

C	4
D	5
D	6

C	4
D	11
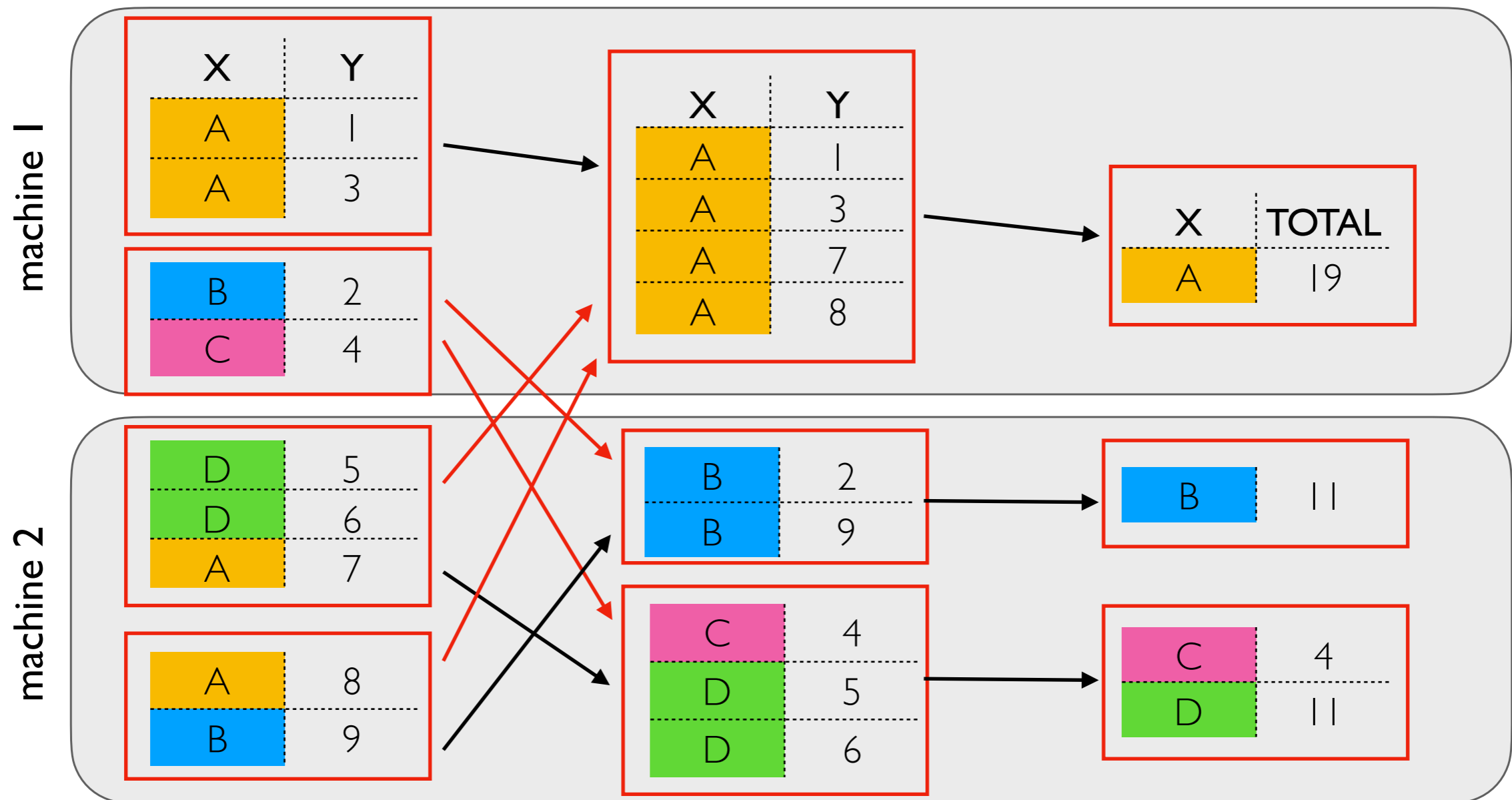
Logically
- lots of groups
- need to bring related (grouped) data together
- stats per group

Physically (Spark)
- RDDs broken into partitions
- generally many groups per partition
- tasks processing partitions run on specific machines
- generally multiple partitions per machine

Spark: Physical Execution on Partitions

□ partition

machine 1

X	Y
A	1
A	3

X	Y
B	2
C	4

X	Y
A	1
A	3
A	7
A	8

X	TOTAL
A	19

machine 2

X	Y
D	5
D	6
A	7

X	Y
A	8
B	9

X	Y
B	2
B	9

X	Y
C	4
D	5
D	6

X	
B	11

X	
C	4
D	11

wide transformation

requires exchange/shuffle
(needs network I/O)

narrow transformation

action

X	TOTAL
A	19
B	11
C	4
D	11

file or
Pandas DF

Spark: Physical Execution on Partitions

□ partition

machine 1

X	Y
A	1
A	3

X	Y
B	2
C	4

X	Y
A	1
A	3
A	7
A	8

X	TOTAL
A	19

machine 2

X	Y
D	5
D	6
A	7

X	Y
A	8
B	9

X	Y
B	2
B	9

X	Y
C	4
D	5
D	6

X		
B		11

X		
C	4	
D	11	
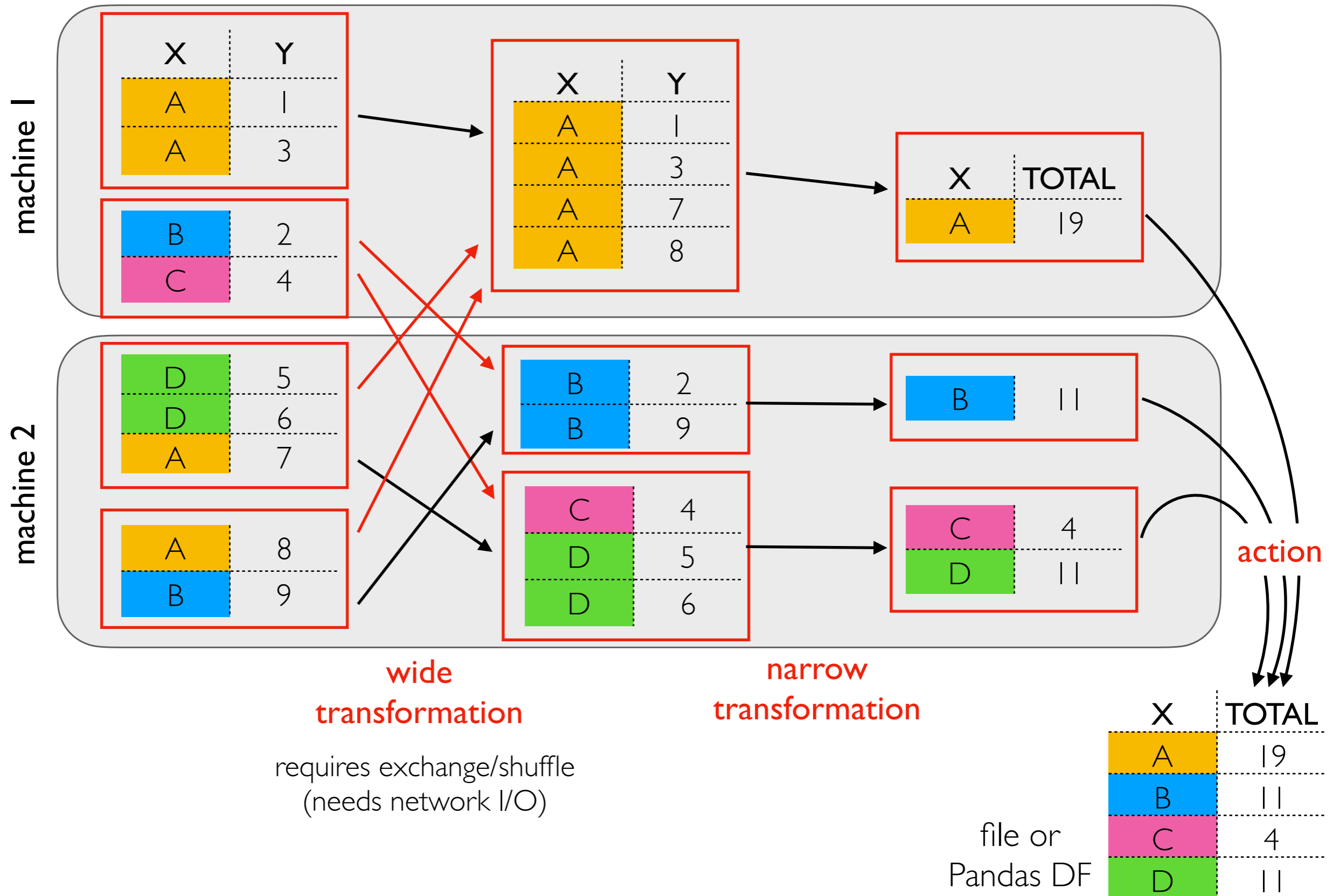
action
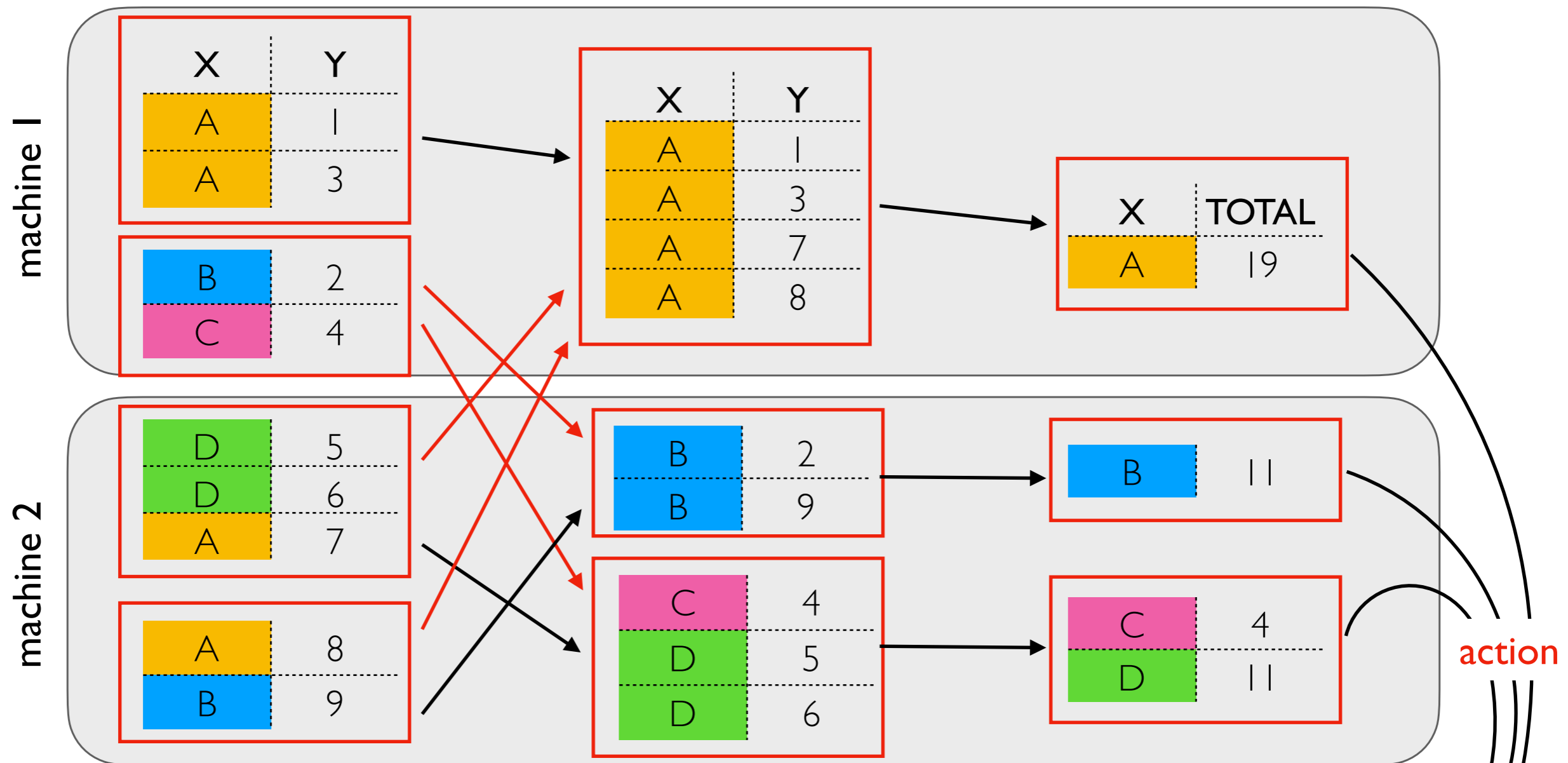
normal partitioning

any row can be in any partition

"hash" partitioning

rows with the same "key" (selected by user) will be in the same partition

X	TOTAL
A	19
B	11
C	4
D	11

file or Pandas DF

Spark: Physical Execution on Partitions

Spark: Physical Execution on Partitions

□ partition

machine 1

X	Y
A	1
A	3

| B | 2 |
| C | 4 |

X	Y
A	4
A	7
A	8

X	TOTAL
A	19

machine 2

X	Y
D	5
D	6
A	7

| A | 8 |
| B | 9 |

| B | 2 |
| B | 9 |

| B | 11 |

| C | 4 |
| D | 11 |

| C | 4 |
| D | 11 |

action

Optimization 1: partial aggregates
For some aggregates (e.g., sum, count, avg), we can compute
partial results *prior* to the exchange, often saving network I/O

file or
Pandas DF

X	TOTAL
A	19
B	11
C	4
D	11

Shuffle Partitions
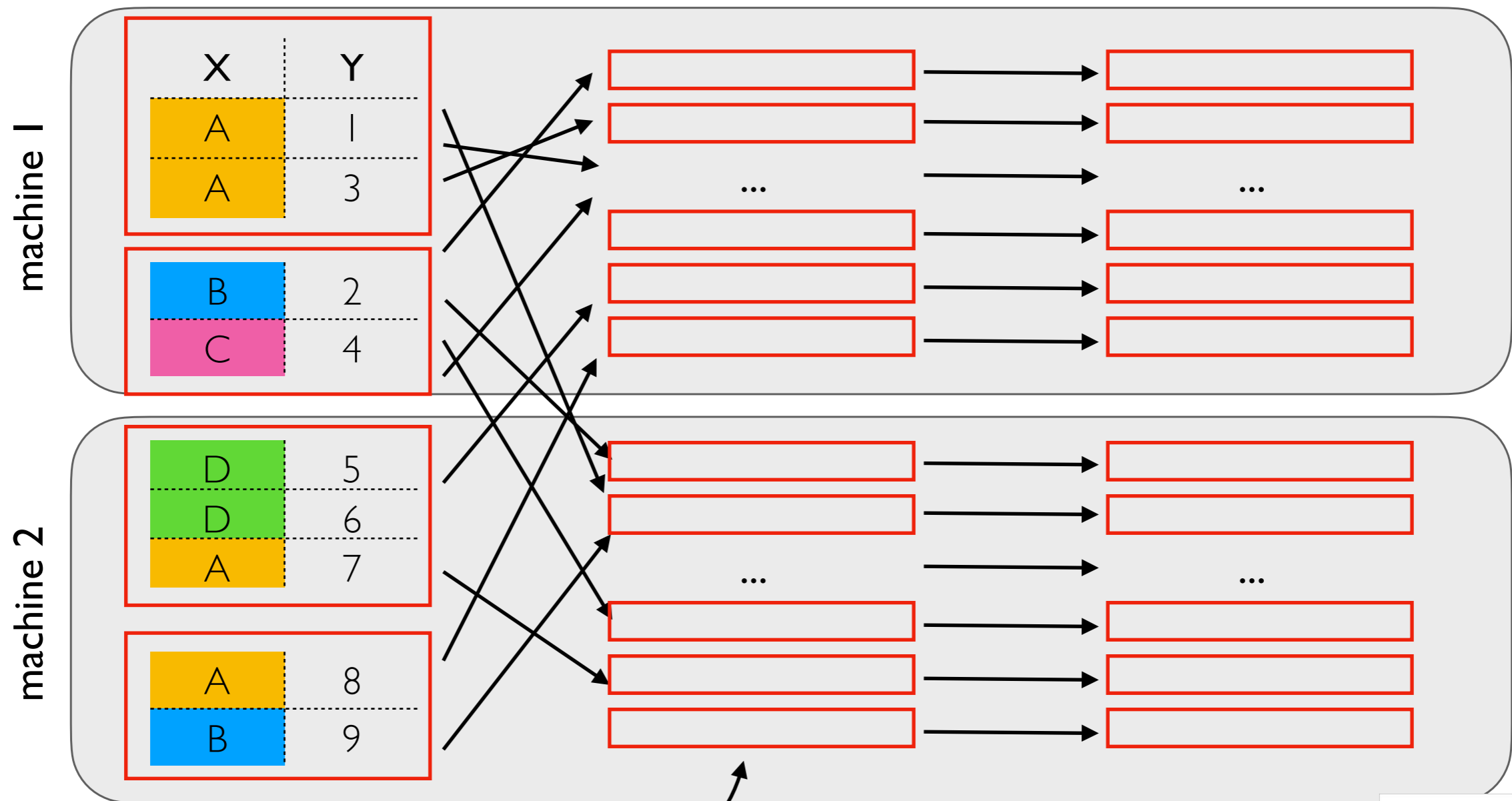
How many partitions will we have?

- spark.sql.shuffle.partitions (default 200) sets this -- fixed for whole application
- 200 is often too much given dataset and cluster size
- **Optimization 2:** spark.sql.adaptive.coalescePartitions.enabled
  (combine small partitions into few bigger ones)
- partition coelescing not available for Spark streaming (later lecture)
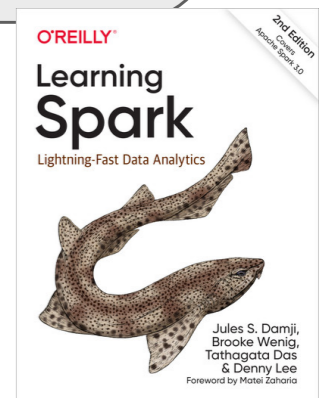
see Epilogue:
Apache Spark 3.0

Parquet: Bucketed Data

X	Y
A	1
A	3
A	7
A	8

X	Y
D	5
D	6
C	4

X	Y
B	2
B	9

machine 1

machine 2
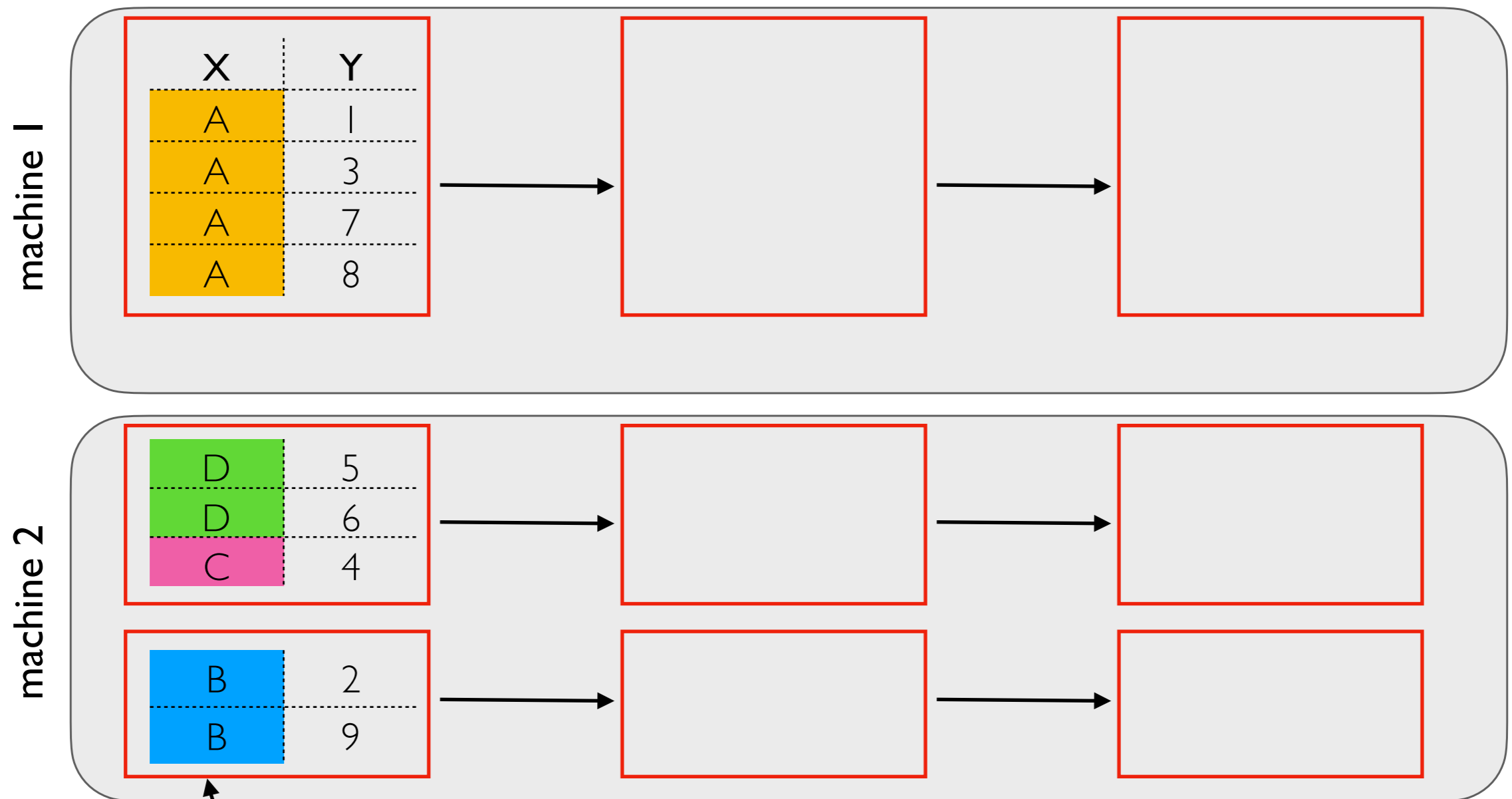
Wouldn't it be fantastic if the data came pre-partitioned?

- Parquet-formatted Spark tables can be written this way
- Decide carefully which column to use based on future calculations
- You can only choose one per table! (though you could have copies)
- **Optimization 3:** bucketBy (when table was previously created)

Grouping Demos

Single-Machine Join Demos

Outline

Schema Inference

Collecting Data

Caching

Grouping

Joining

BHJ: Broadcast Hash Join

☐ partition

machine 1

fruit_id	cost
B	1
A	2
C	3

id	name
A	Apple
B	Banana

machine 2

fruit_id	cost
A	4
C	5
B	6

id	name
C	Carrot
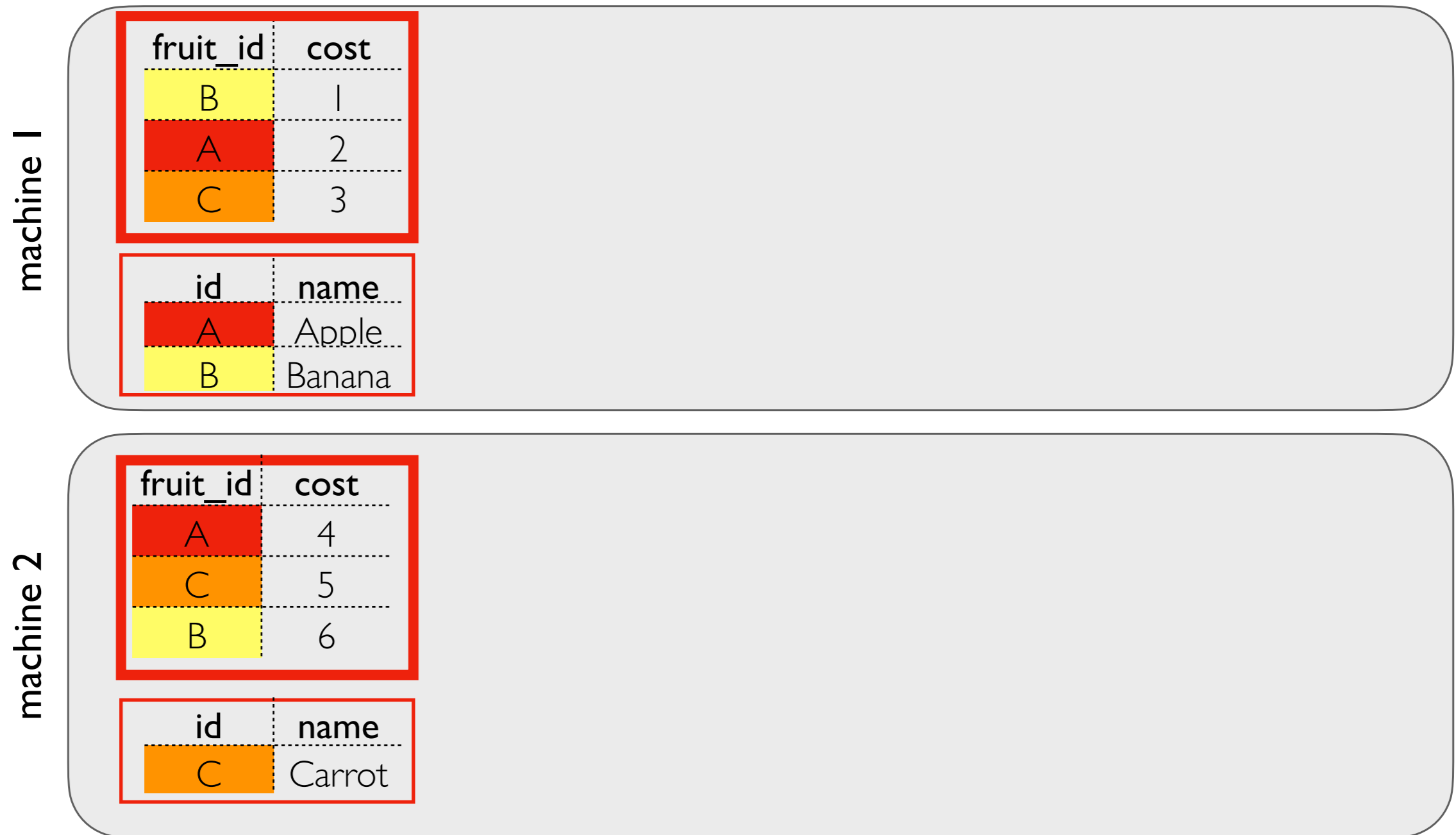
we can apply the strategy from the coding demo to each partition of the bigger table

BHJ: Broadcast Hash Join

☐ partition

machine 1

fruit_id	cost
B	1
A	2
C	3

id	name
A	Apple
B	Banana

IN MEMORY:
```
{'A': 'Apple',
 'B': 'Banana',
 'C': 'Carrot'}
```

machine 2

fruit_id	cost
A	4
C	5
B	6

id	name
C	Carrot

IN MEMORY:
```
{'A': 'Apple',
 'B': 'Banana',
 'C': 'Carrot'}
```
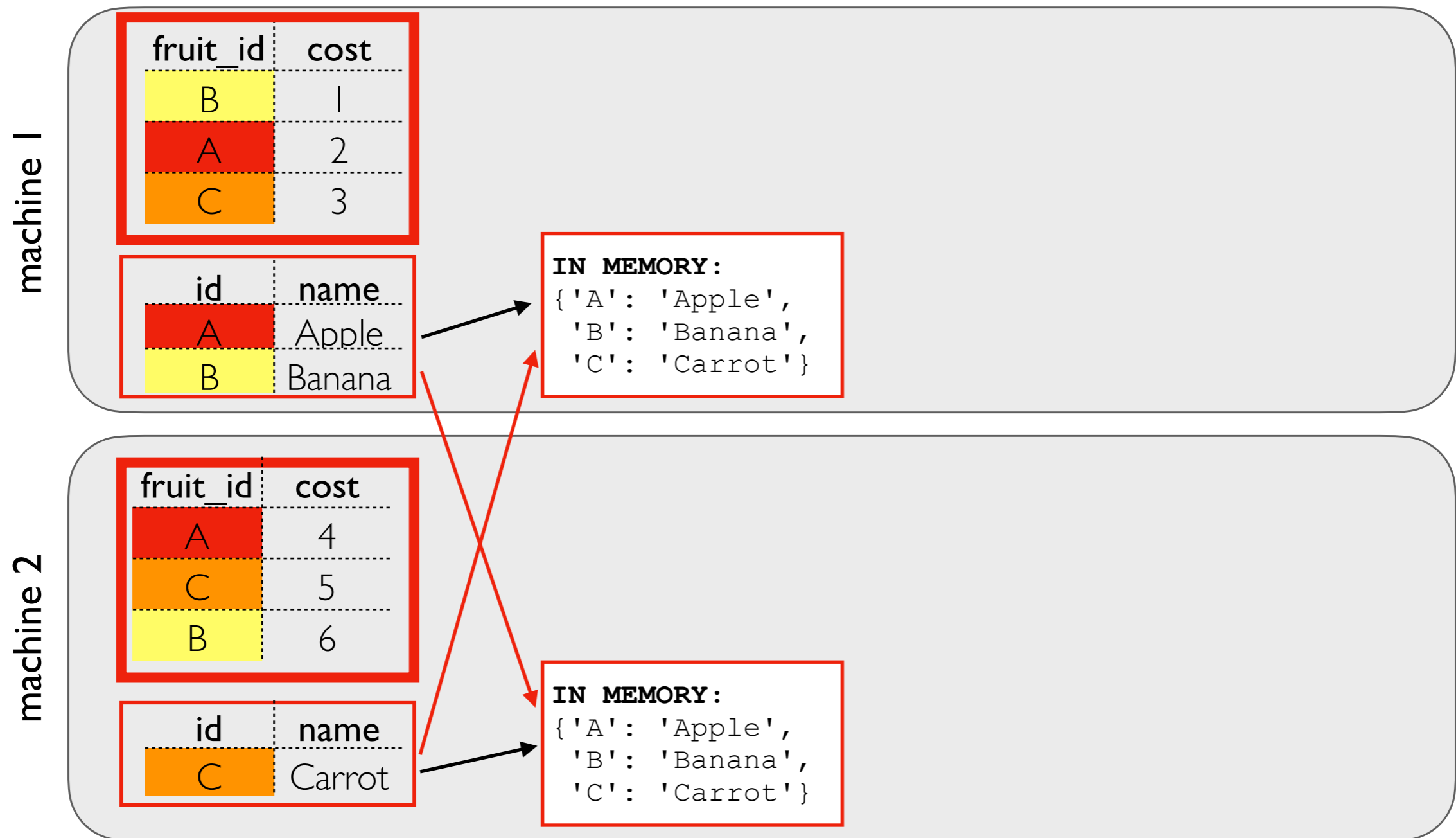
Broadcast step
- a copy of the smaller table is sent to EVERY machine involved
- it is loaded to an in-memory hash table (dict) for quick lookup

BHJ: Broadcast Hash Join

☐ partition

machine 1

fruit_id	cost
B	1
A	2
C	3

id	name
A	Apple
B	Banana

```
IN MEMORY:
{'A': 'Apple',
 'B': 'Banana',
 'C': 'Carrot'}
```

fruit_id	name	cost
B	Banana	1
A	Apple	2
C	Carrot	3

machine 2

fruit_id	cost
A	4
C	5
B	6

id	name
C	Carrot

```
IN MEMORY:
{'A': 'Apple',
 'B': 'Banana',
 'C': 'Carrot'}
```

fruit_id	name	cost
A	Apple	4
C	Carrot	5
B	Banana	6
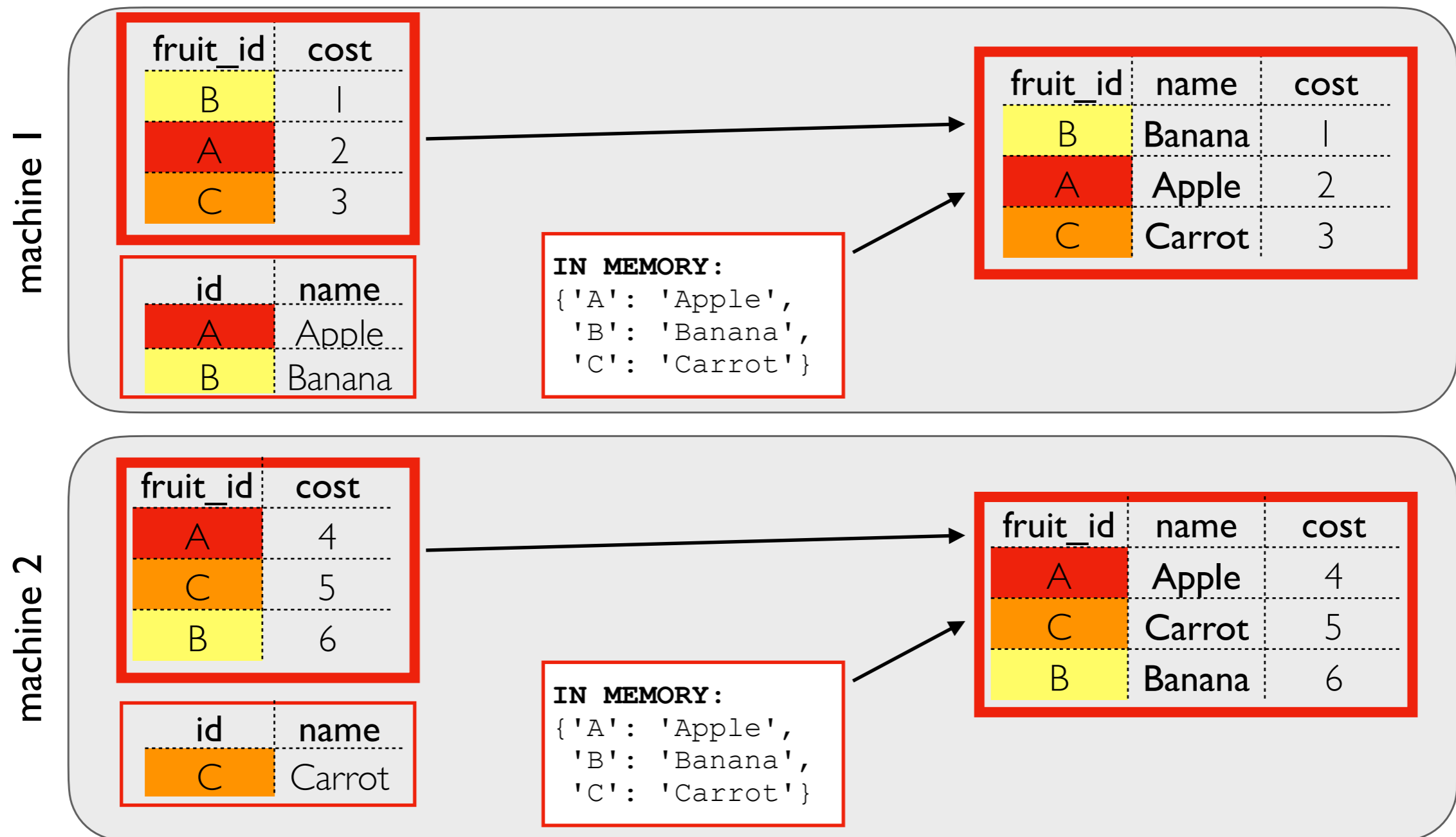
Hash Join Step
- don't transfer bigger table over network
- loop over it
- lookup keys in in-memory hash table (dict)

SMJ: Shuffle Sort Merge Join

□ partition

machine 1

fruit_id	cost
B	1
A	2
C	3

id	name
A	Apple
B	Banana

machine 2

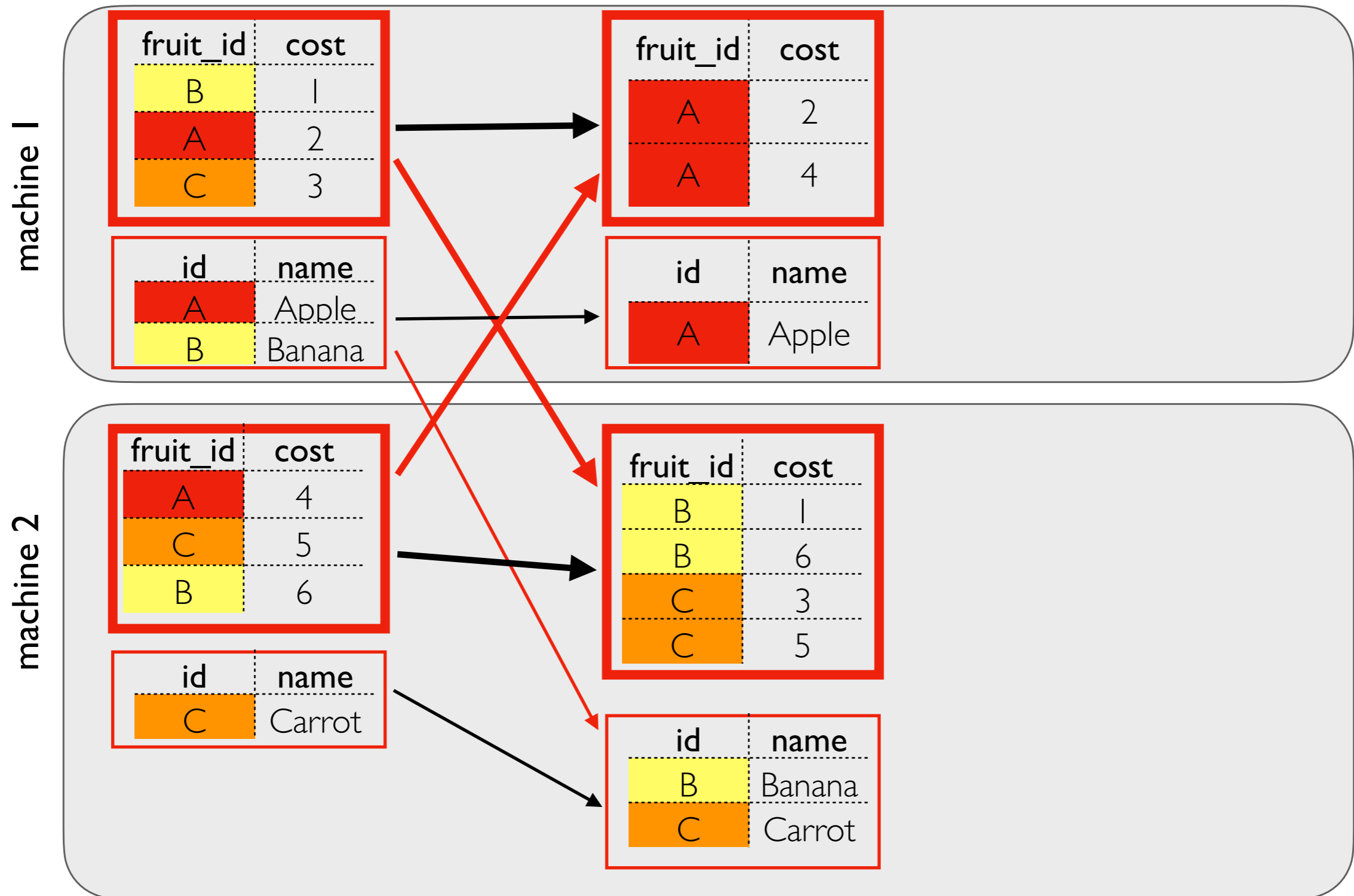| fruit_id	cost
A	4
C	5
B	6

id	name
C	Carrot

need to pull related data (same fruit_id) from both tables together to the same place

SMJ: Shuffle Sort Merge Join

sorted within partitions

☐ partition

machine 1

fruit_id	cost
B	1
A	2
C	3

id	name
A	Apple
B	Banana

fruit_id	cost
A	2
A	4

id	name
A	Apple

machine 2

fruit_id	cost
A	4
C	5
B	6

id	name
C	Carrot

fruit_id	cost
B	1
B	6
C	3
C	5

id	name
B	Banana
C	Carrot

Shuffle+Sort Step

SMJ: Shuffle Sort Merge Join

☐ partition

machine 1

fruit_id	cost
B	1
A	2
C	3

id	name
A	Apple
B	Banana

fruit_id	cost
A	2
A	4

id	name
A	Apple

fruit_id	name	cost
A	Apple	2
A	Apple	4

machine 2

fruit_id	cost
A	4
C	5
B	6

id	name
C	Carrot

fruit_id	cost
B	1
B	6
C	3
C	5

id	name
B	Banana
C	Carrot

fruit_id	name	cost
B	Banana	1
B	Banana	6
C	Carrot	3
C	Carrot	5

Merge Join Step

Network I/O: SMJ vs. BHJ

SMJ
- each table goes over the network about once

BHJ
- only the small table goes over the network
- but it goes about N times! (where N is the number of nodes involved)

When does BHJ tend to do well?
- when one table is much smaller than the other
- when the smaller table fits entirely into memory as a hash table
- when the smaller table does not need to be sent to too many nodes

Seeing Join Type with Explain

very large table

tiny table

```
(calls
 .join(holidays, calls["CallDate"] == holidays["date"],
       how="inner")
 .groupby("date", "holiday").count()).explain()
```

Simplified Output:

```
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate - count
   +- Exchange hashpartitioning
      +- HashAggregate - partial count
         +- Project
            +- BroadcastHashJoin                    using BHJ
               :- Filter isnotnull(CallDate#230)
               :  ...
               +- BroadcastExchange
                  +- Filter isnotnull(date#339)
                     +- FileScan csv (holidays2.csv)
```

send contents of holidays2.csv to every worker involved in the JOIN

Join Hints

```
(calls
 .join(holidays.hint("merge"),
       calls["CallDate"] == holidays["date"],
       how="inner")
 .groupby("date", "holiday").count()).explain()
```

Simplified Output:

```
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate - count
   +- Exchange hashpartitioning
      +- HashAggregate - partial count
         +- Project
            +- SortMergeJoin        <--- using SMJ
               ...
```