

[544] Spark Internals and Performance

Meenakshi Syamkumar

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

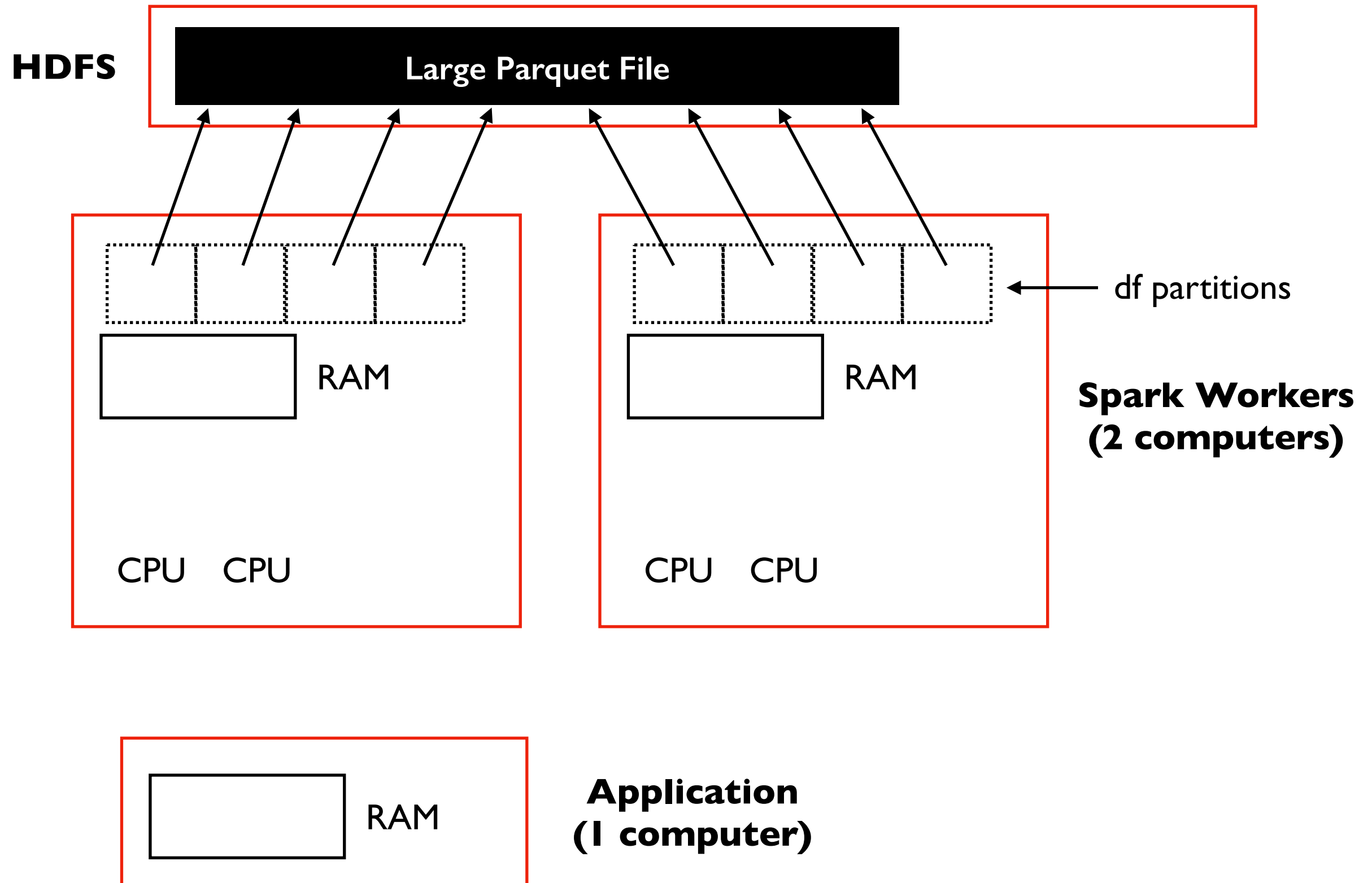
Joining

Collecting Data (OK)

df refers to parquet file

results = df.where(???).collect()

results = df.where(???).toPandas()

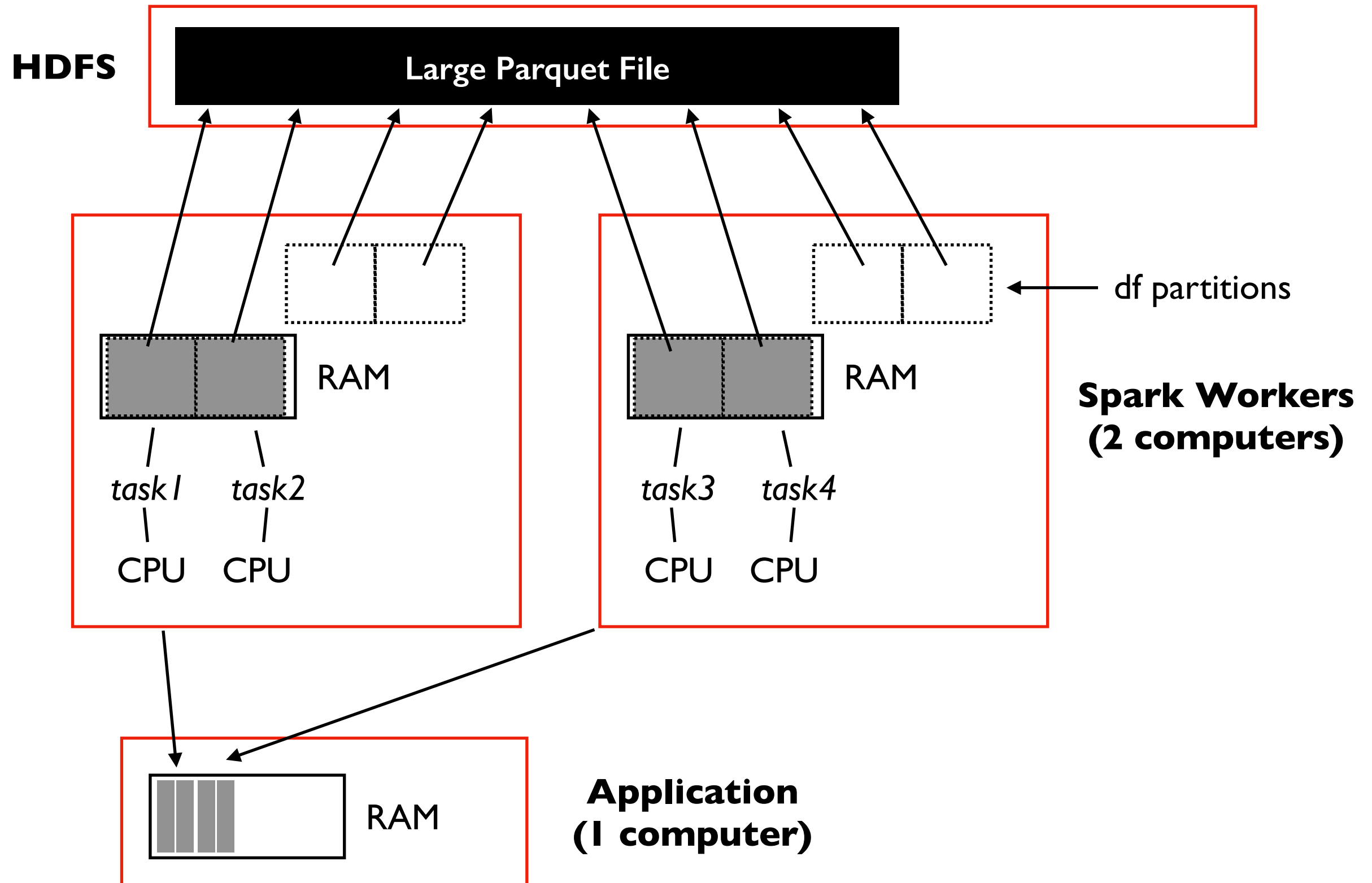


Collecting Data (OK)

df refers to parquet file

results = df.where(???).collect()

results = df.where(???).toPandas()

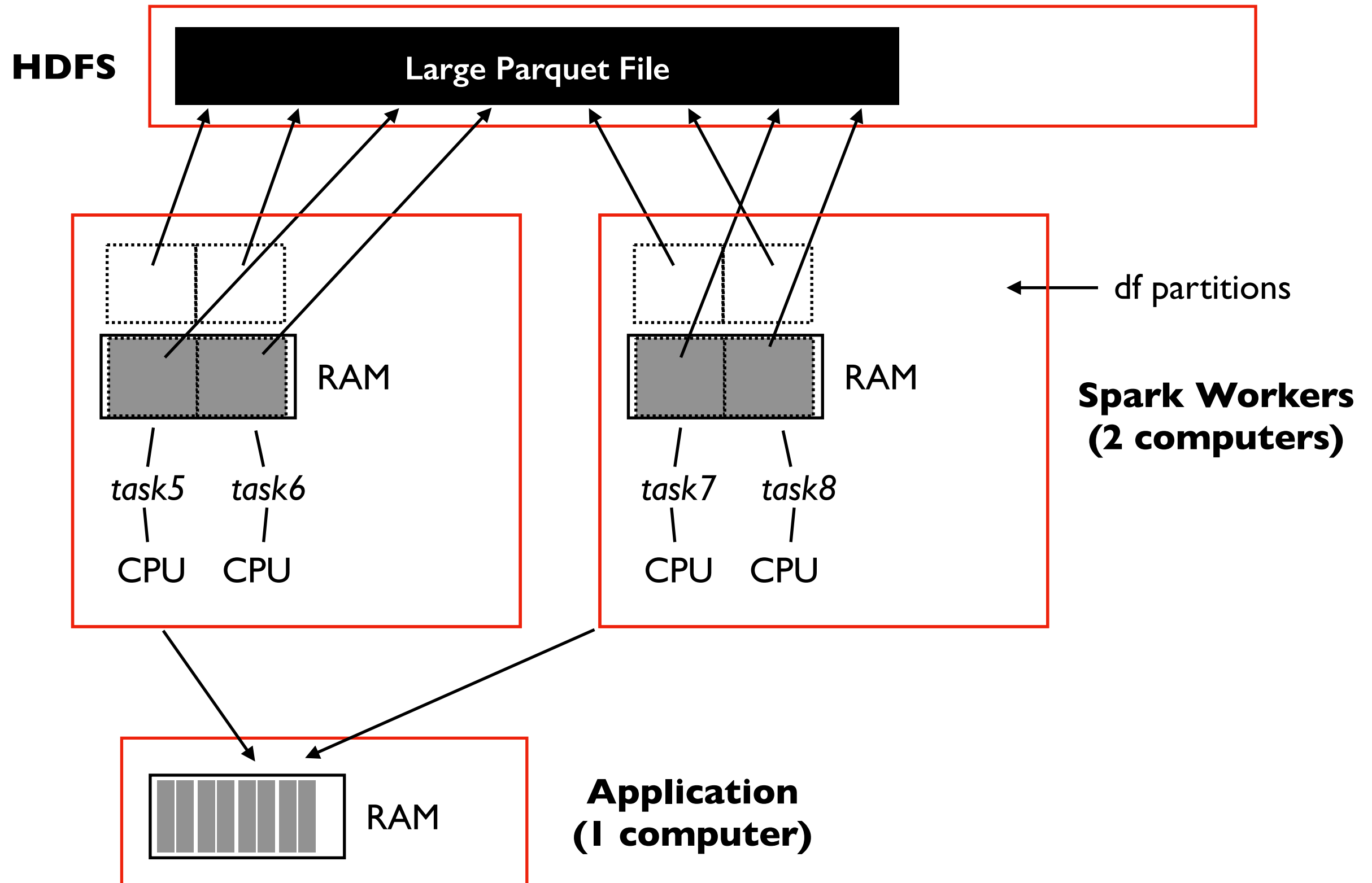


Collecting Data (OK)

df refers to parquet file

results = df.where(???).collect()

results = df.where(???).toPandas()

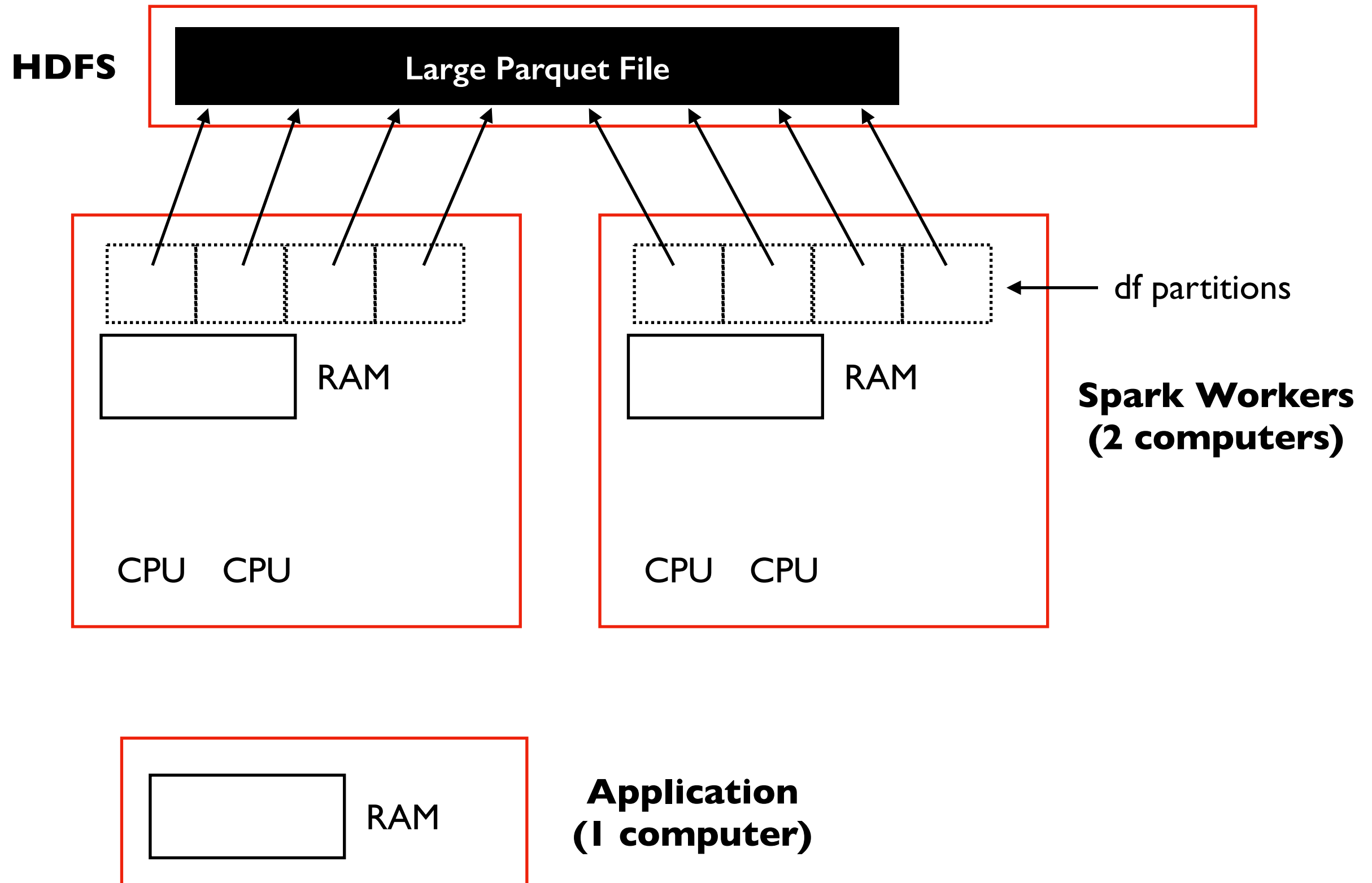


Collecting Data (bad)

df refers to parquet file

results = df.where(???).collect()

results = df.where(???).toPandas()

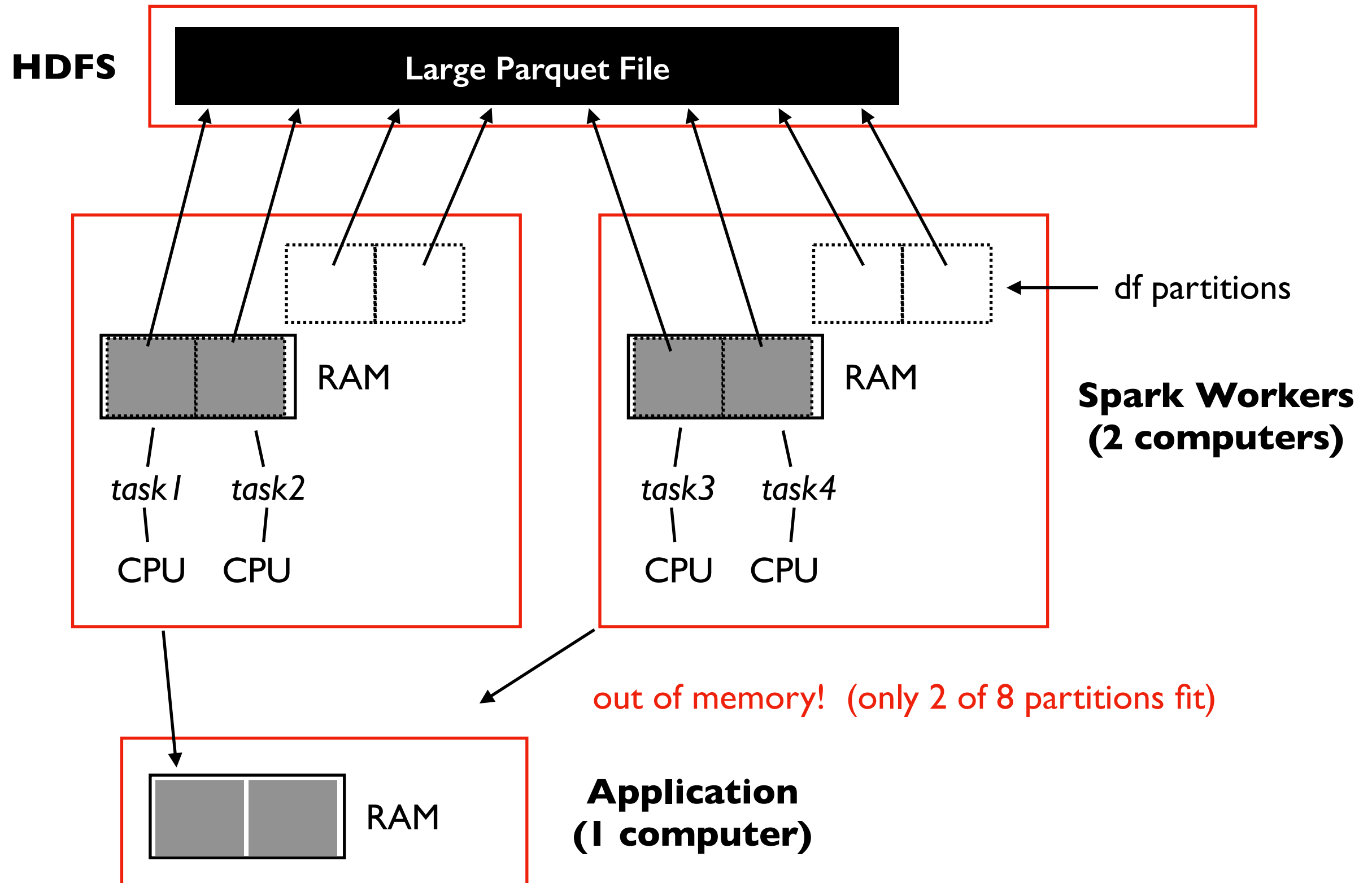


Collecting Data (bad)

df refers to parquet file

results = df.**where(???)**.collect()

results = df.**where(???)**.toPandas()



Outline

Schema Inference

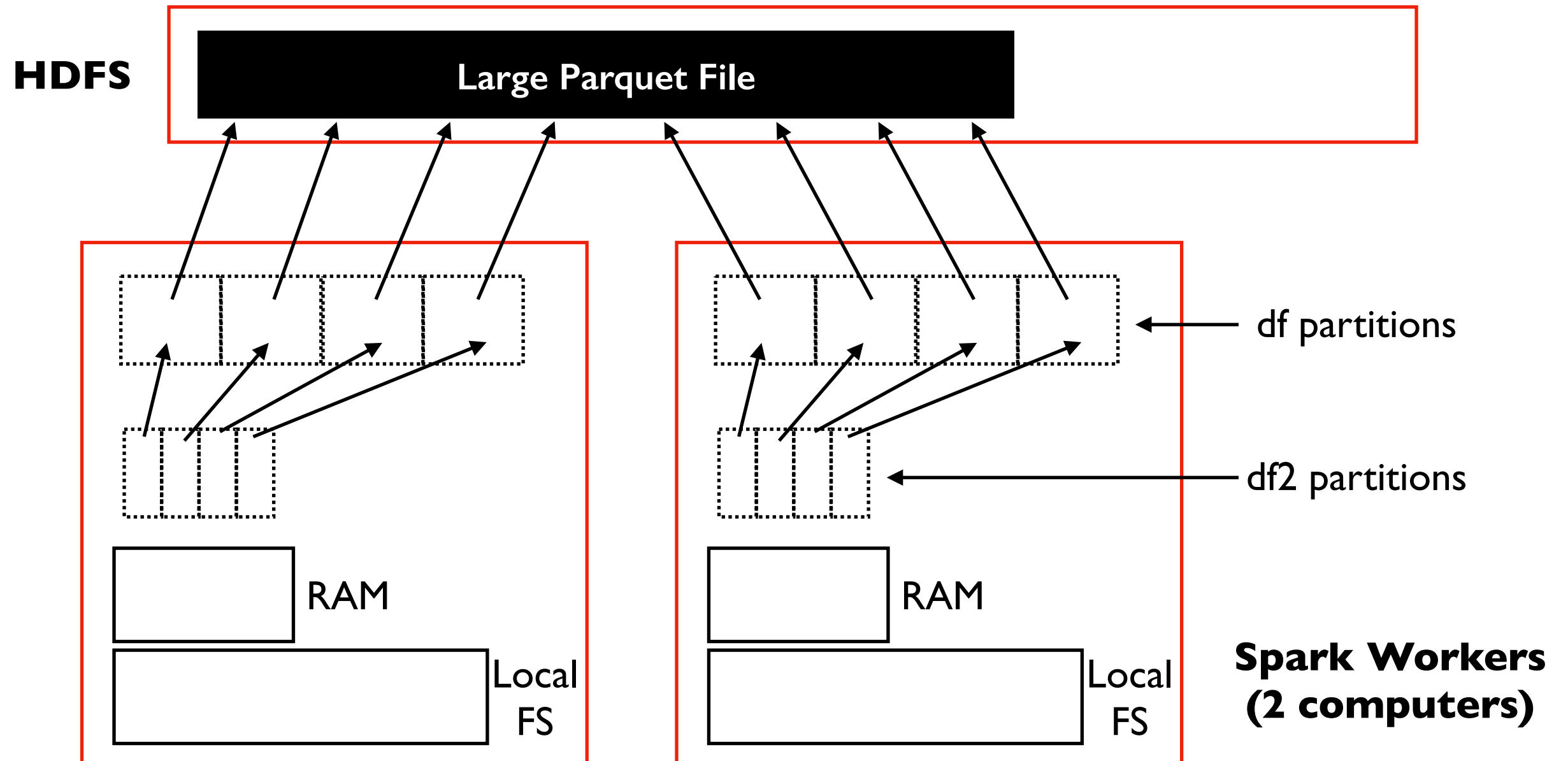
Collecting Data

Caching

Grouping

Joining

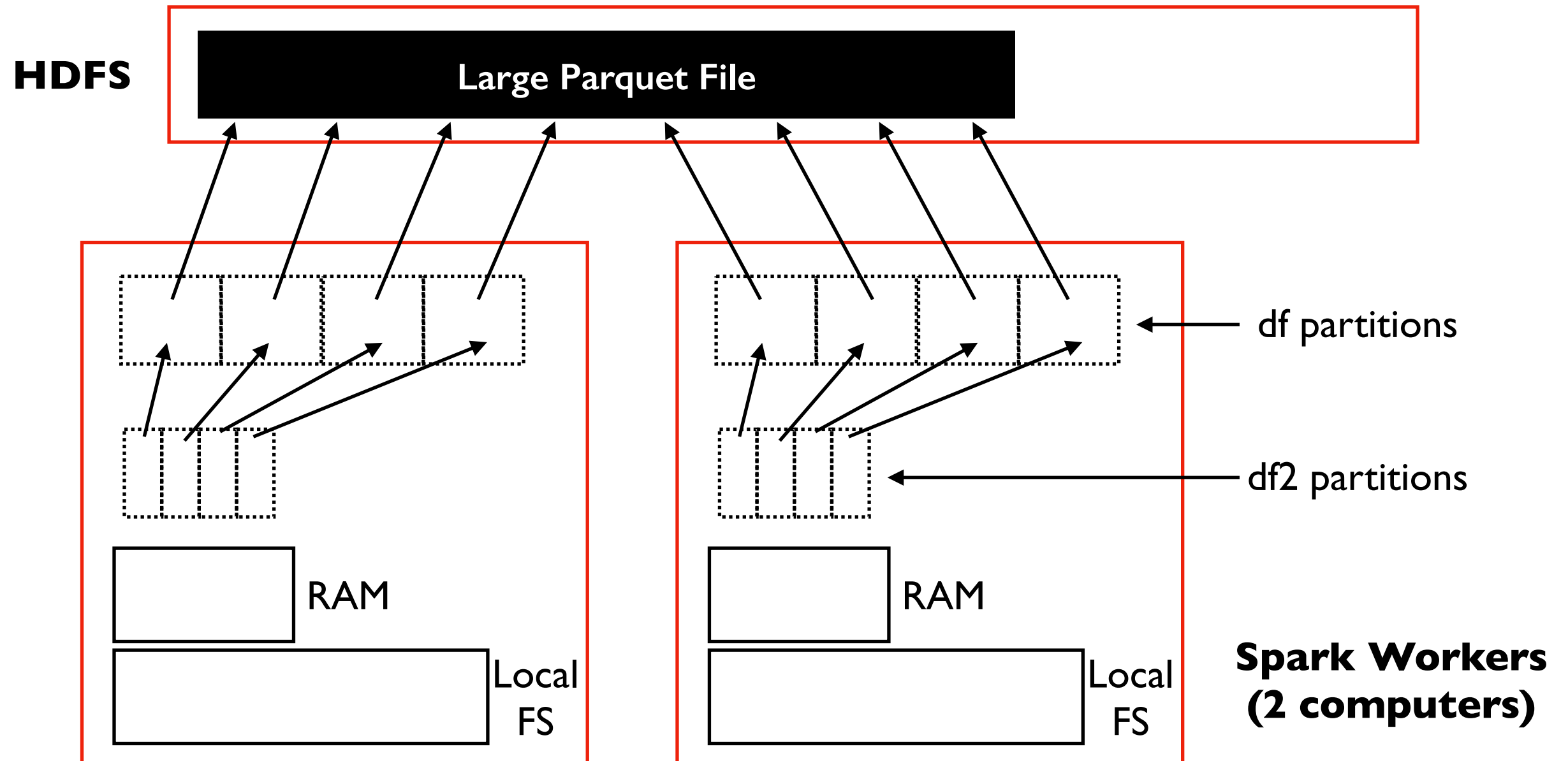
Persisting/Caching



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



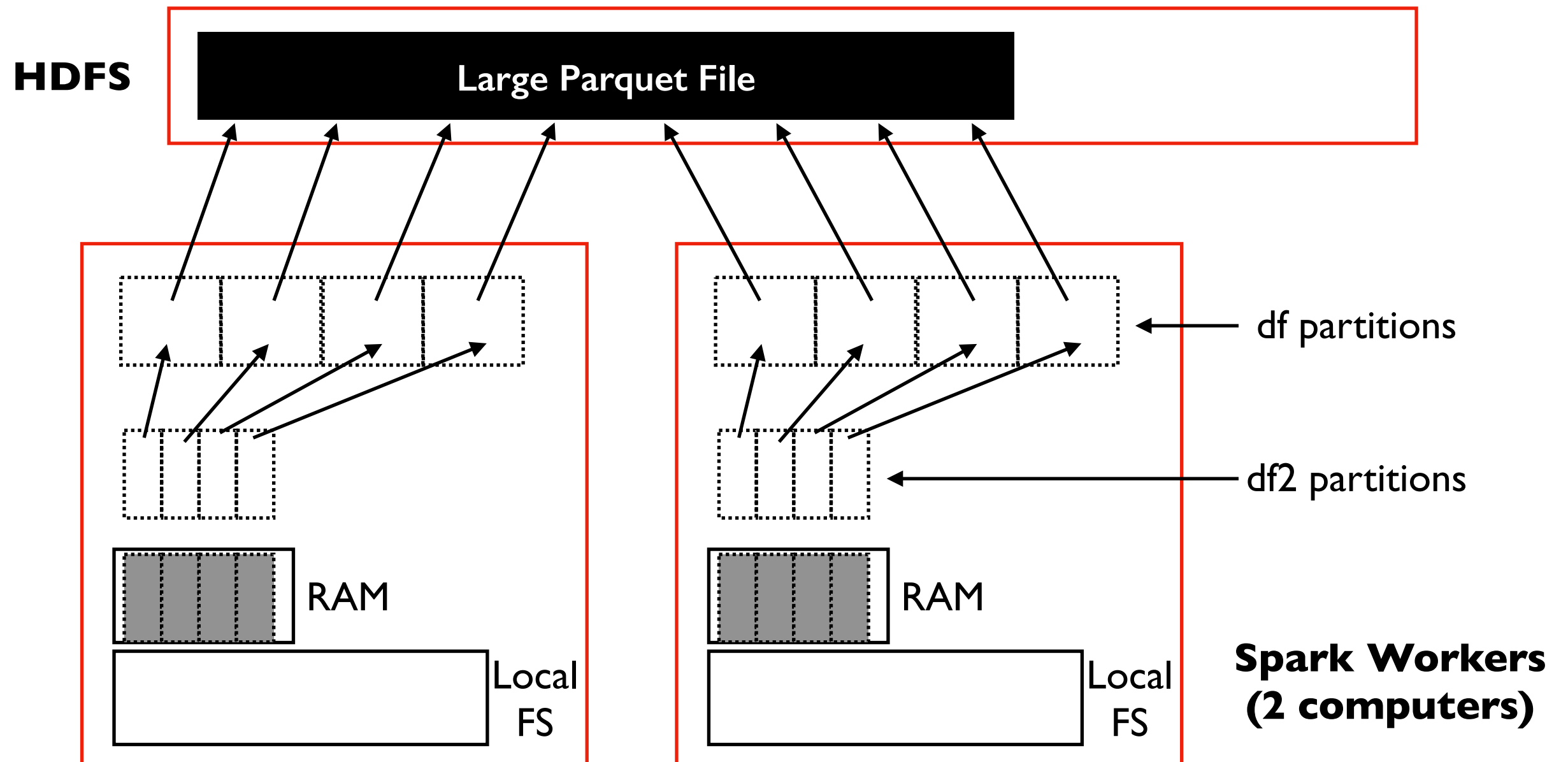
```
from pyspark.storagelevel import StorageLevel  
df2 = df.where(???)
```

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

Persist levels

- MEMORY_ONLY
- MEMORY_ONLY_SER
- DISK_ONLY
- others...

Persisting/Caching



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

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

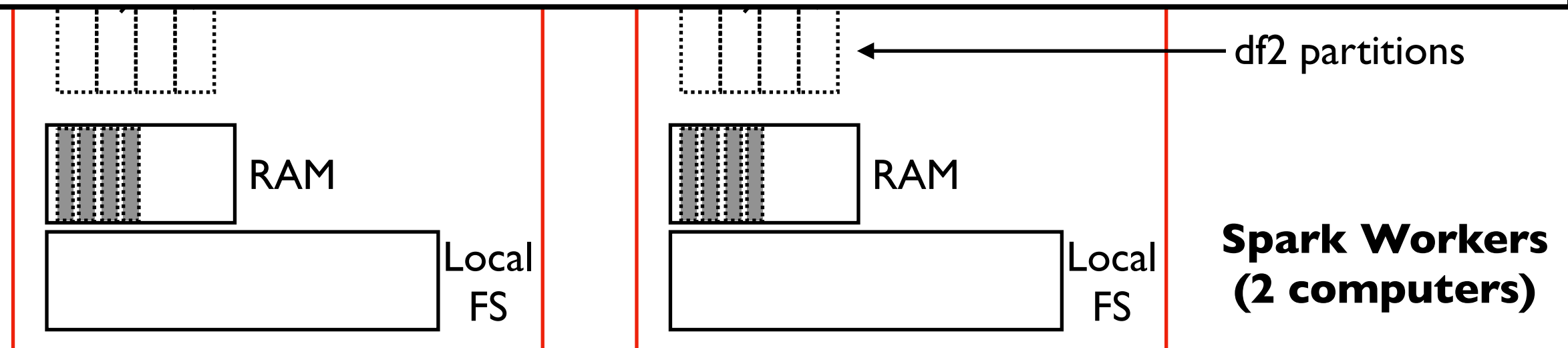
Persist levels

- **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 `Strings` have about 40 bytes of overhead over the raw string data (since they store it in an array of `Chars` 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`.



```
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:

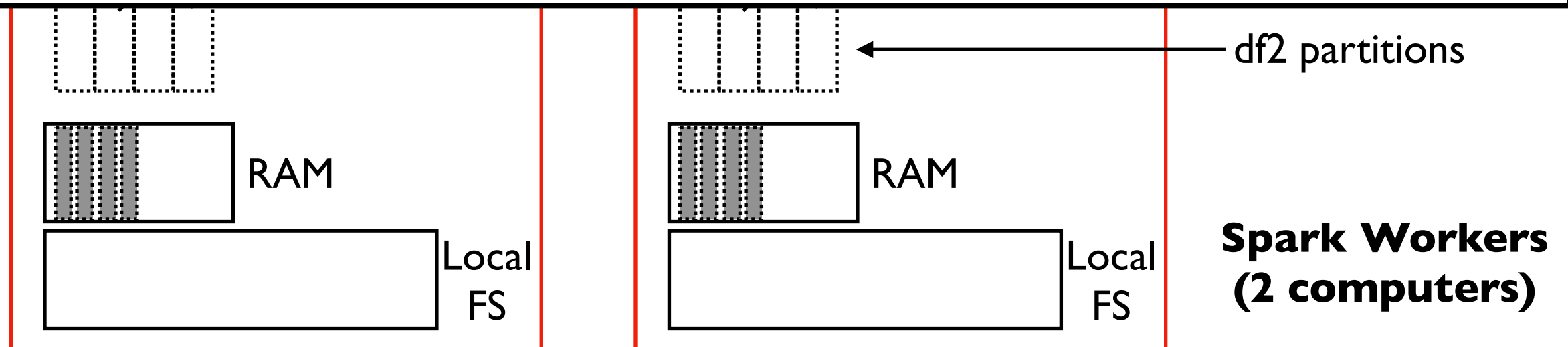
- Each distinct Java object has an “object header”, which is about 16 bytes and contains information such as a pointer to its class. For an object with very little data in it (say one Int field), this can be bigger than the data.

- Java Storage Levels
array of objects
to String
bytes.
• Common Storage Levels 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.**
• Collections

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

Storage Levels 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.**



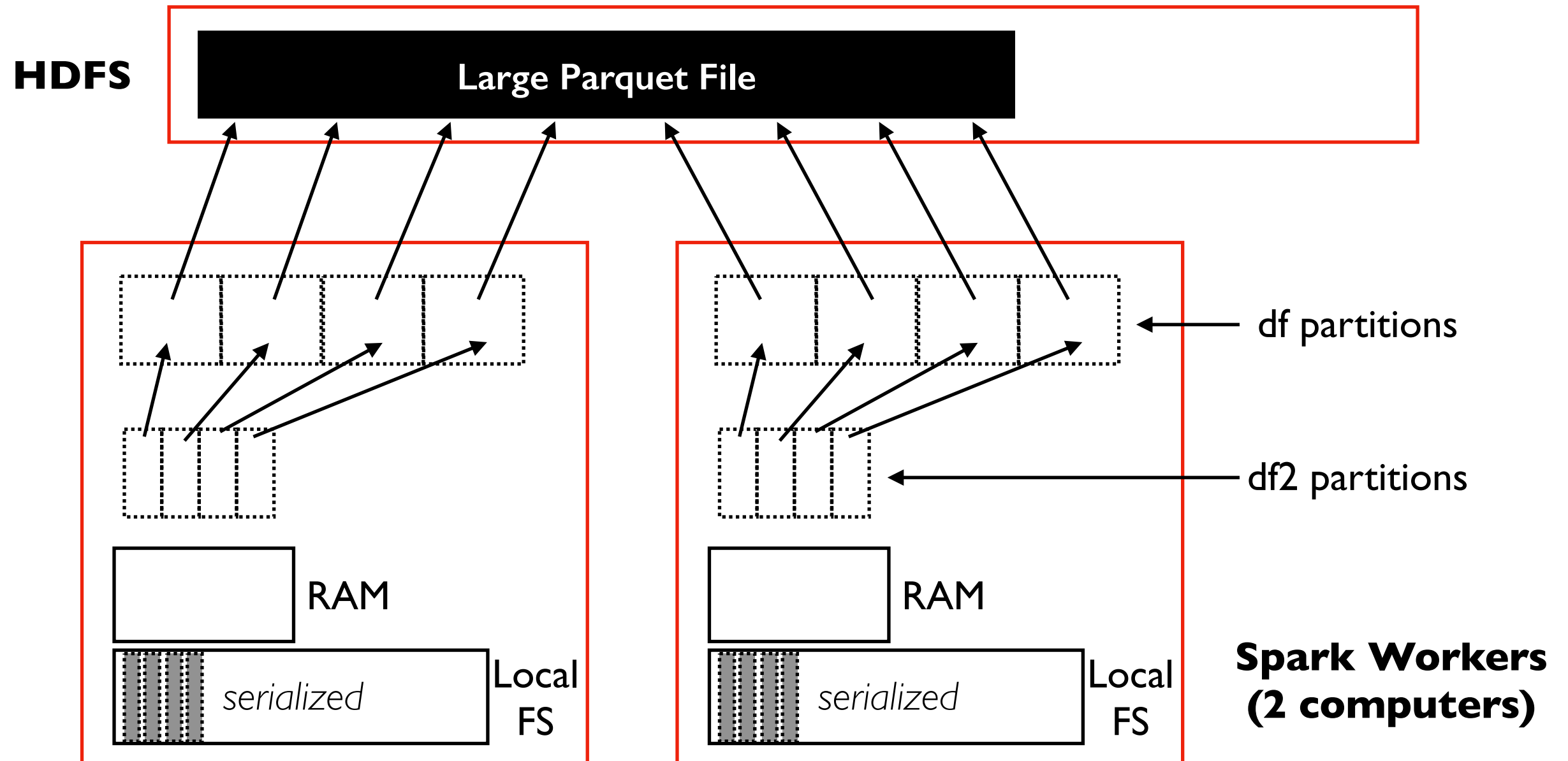
```
from pyspark.storagelevel import StorageLevel  
df2 = df.where(???)
```

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

Persist levels

- MEMORY_ONLY
- **MEMORY_ONLY_SER**
- DISK_ONLY
- others...

Persisting/Caching



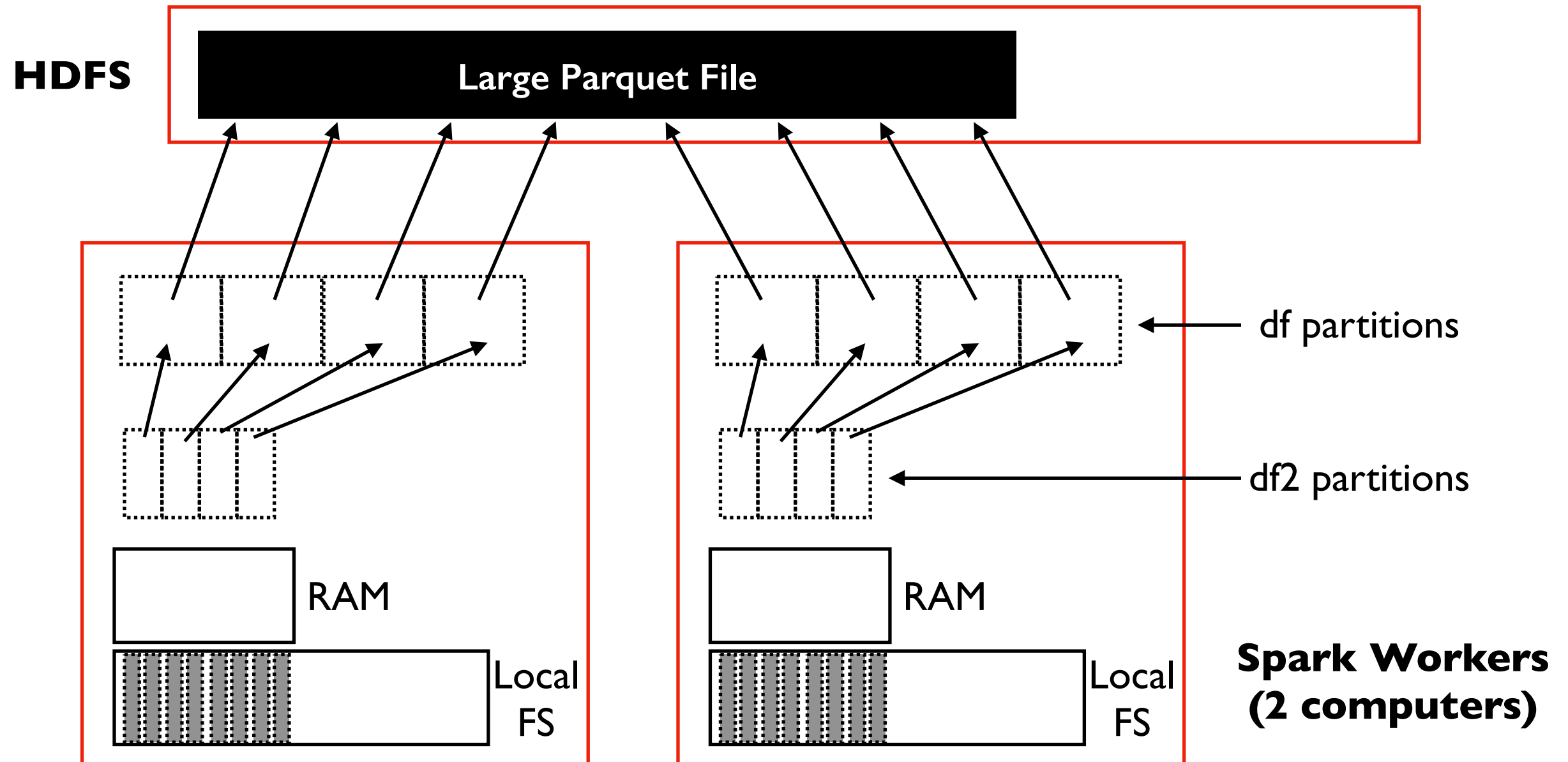
```
from pyspark.storagelevel import StorageLevel  
df2 = df.where(???)
```

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

Persist levels

- MEMORY_ONLY
- MEMORY_ONLY_SER
- **DISK_ONLY**
- others...

Persisting/Caching



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

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

Persist levels (2x replication)

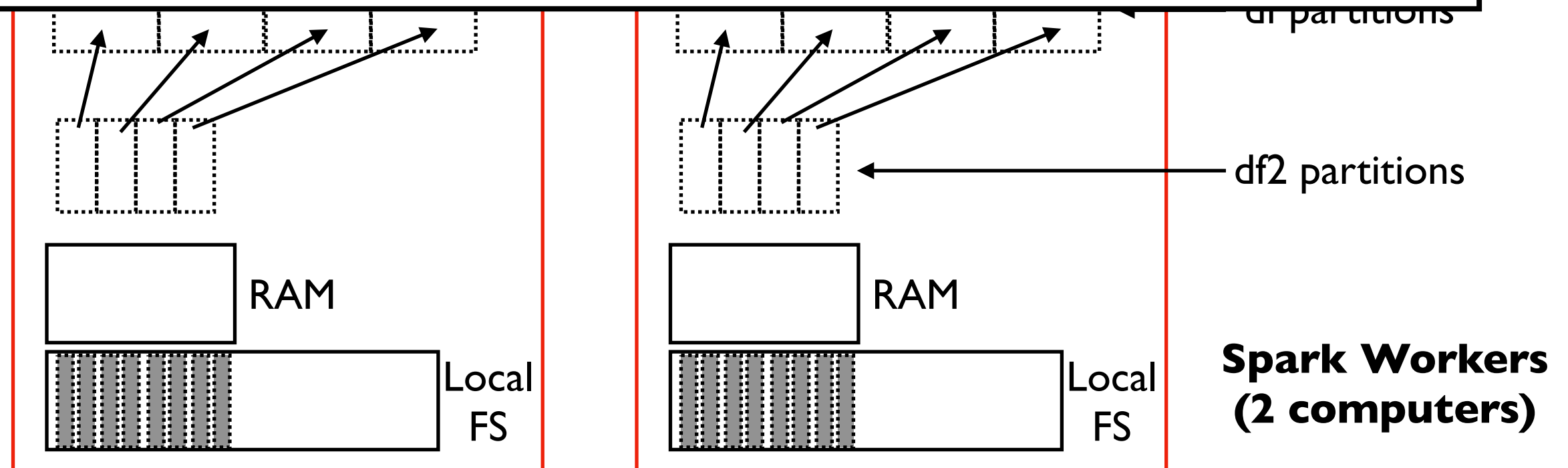
- MEMORY_ONLY_2
- MEMORY_ONLY_SER_2
- **DISK_ONLY_2**
- others...

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

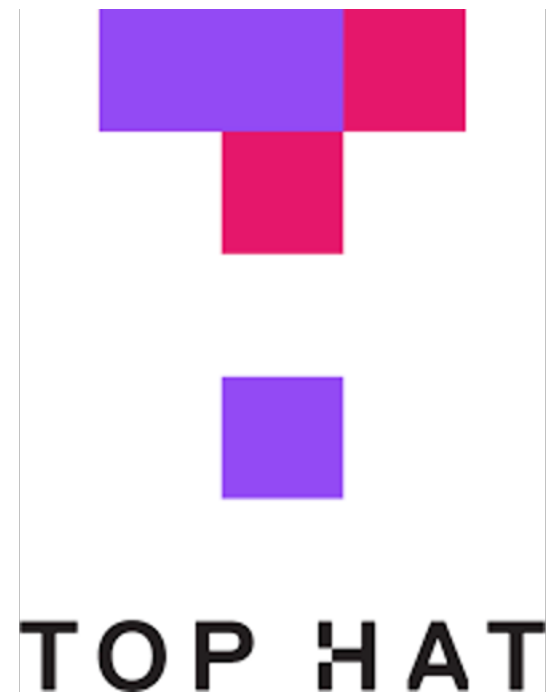


```
from pyspark.storagelevel import StorageLevel  
df2 = df.where(???)
```

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

Persist levels (2x replication)

- MEMORY_ONLY_2
- MEMORY_ONLY_SER_2
- **DISK_ONLY_2**
- others...



Demos...

Outline

Schema Inference

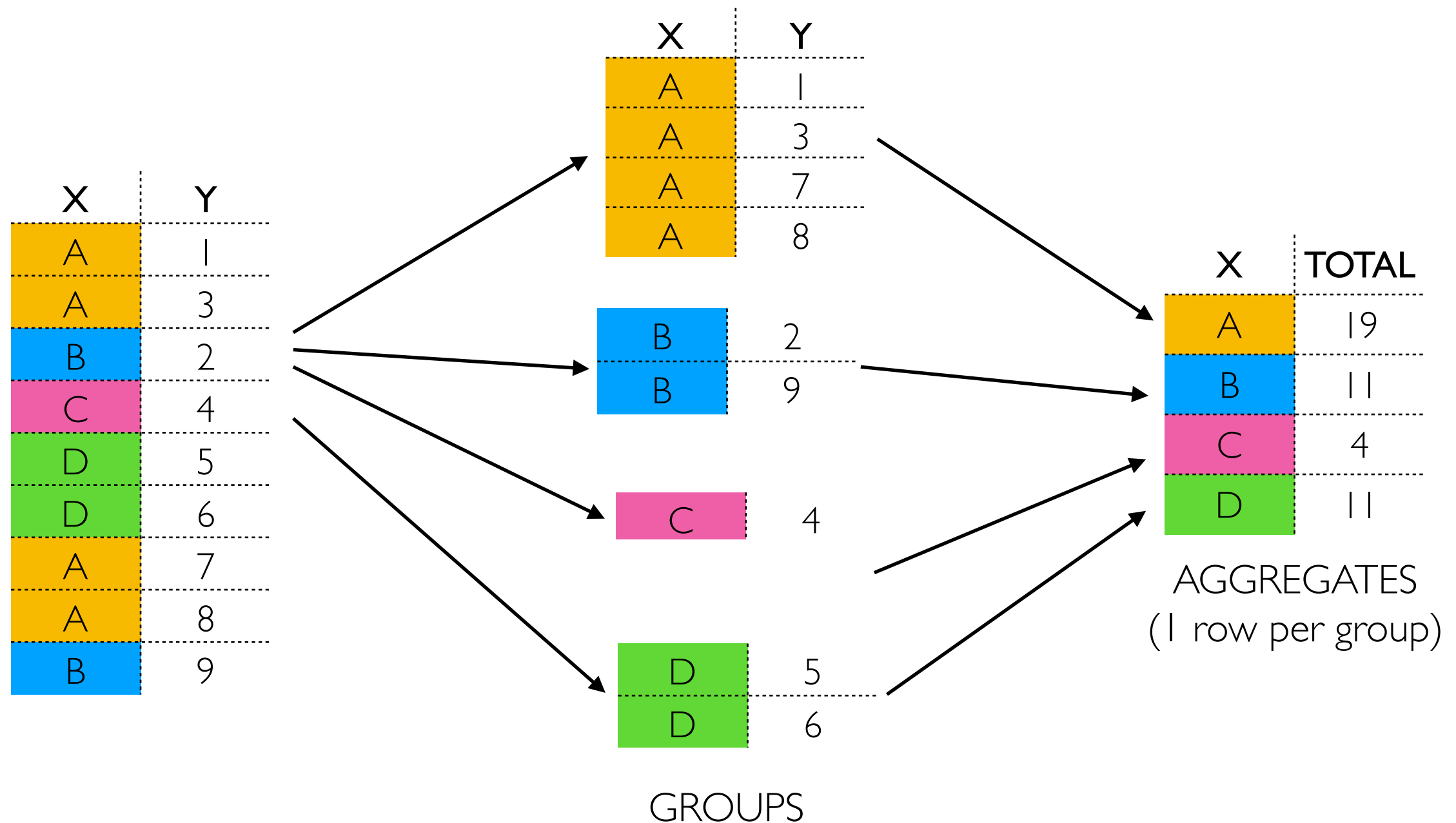
Collecting Data

Caching

Grouping

Joining

GROUPS, AGGREGATES

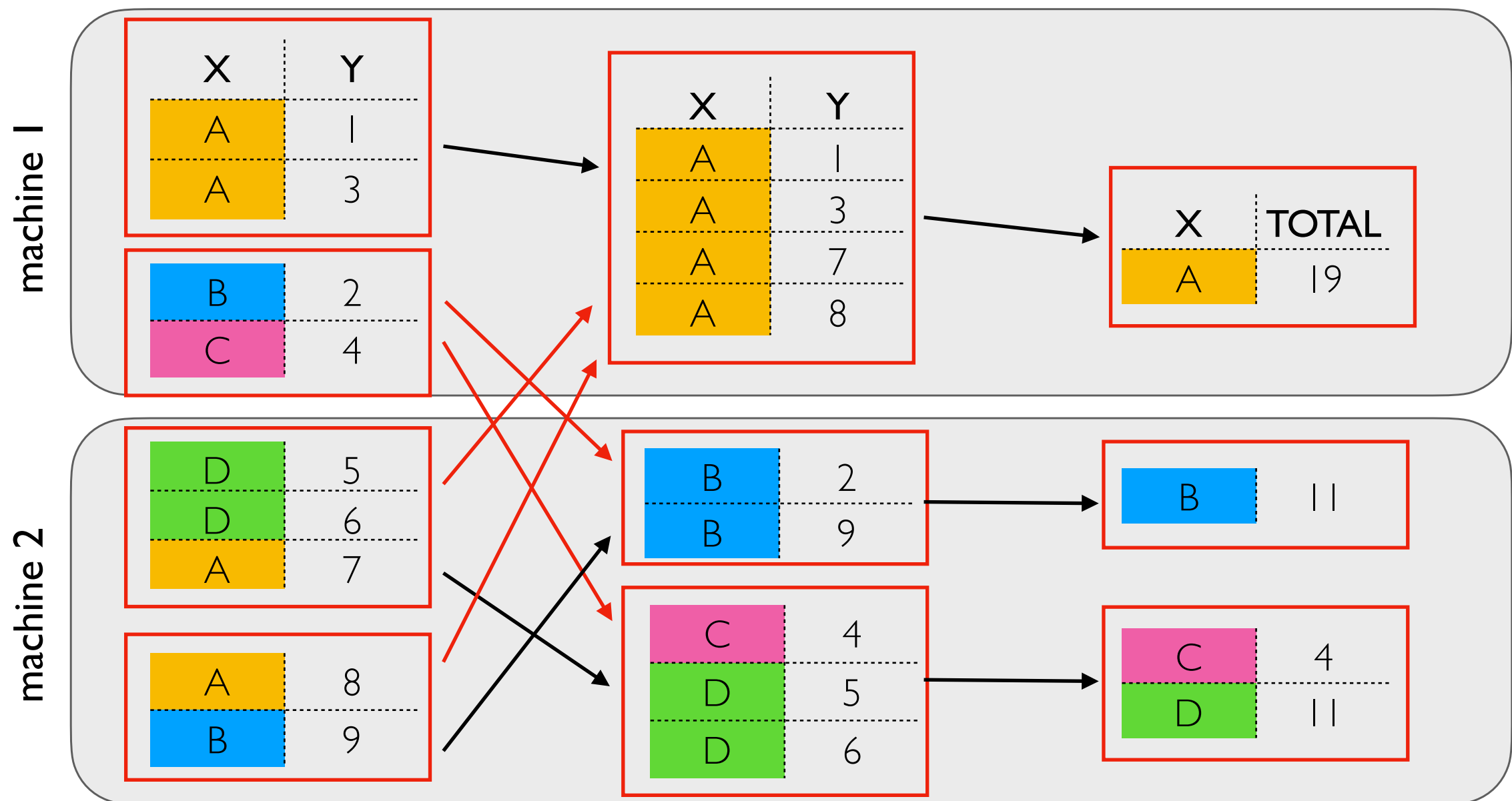


Logically

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

Spark: Physical Execution on Partitions

□ partition



Logically

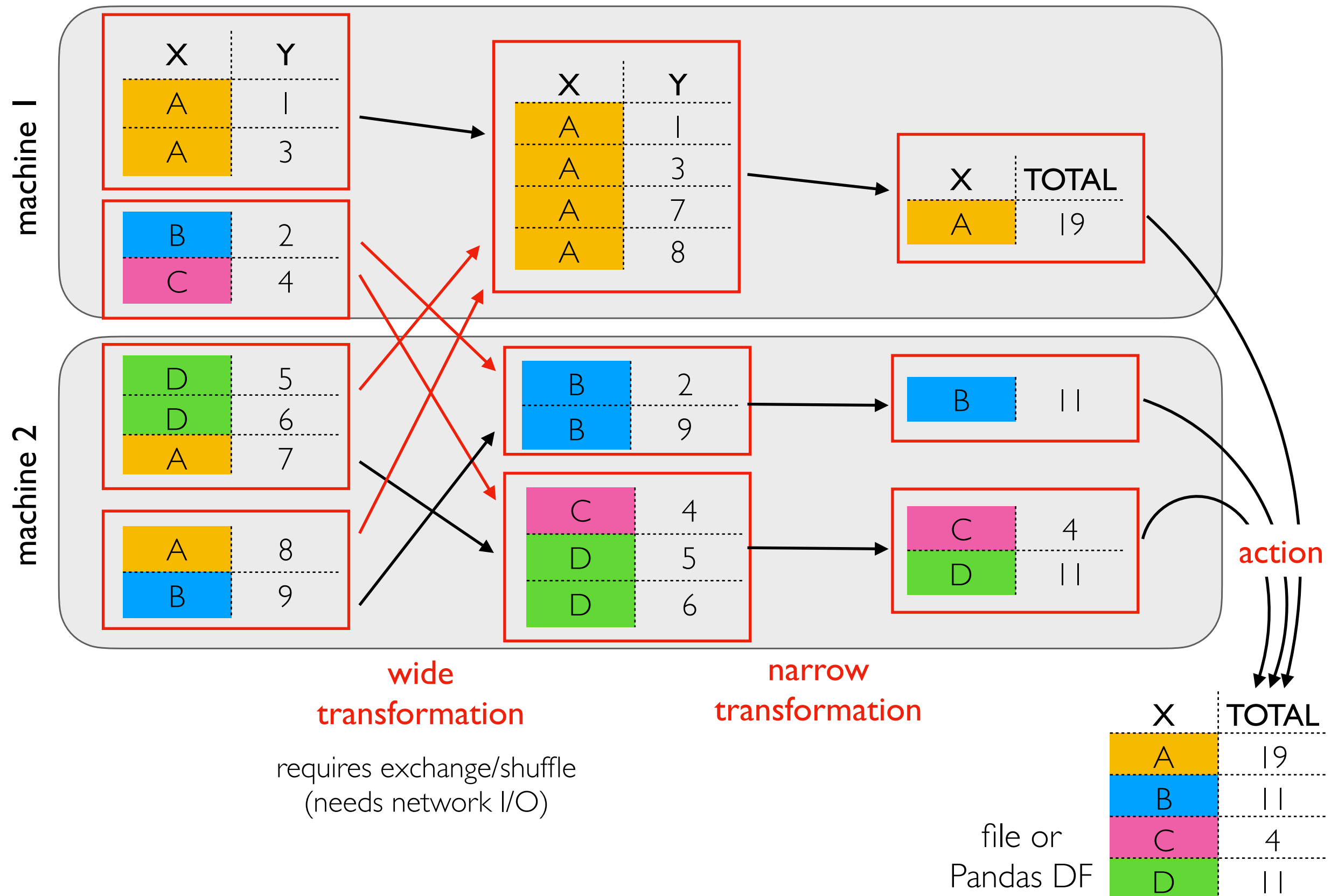
- lots of groups
- need to bring related (grouped) data together
- stats per group

Physically (Spark)

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

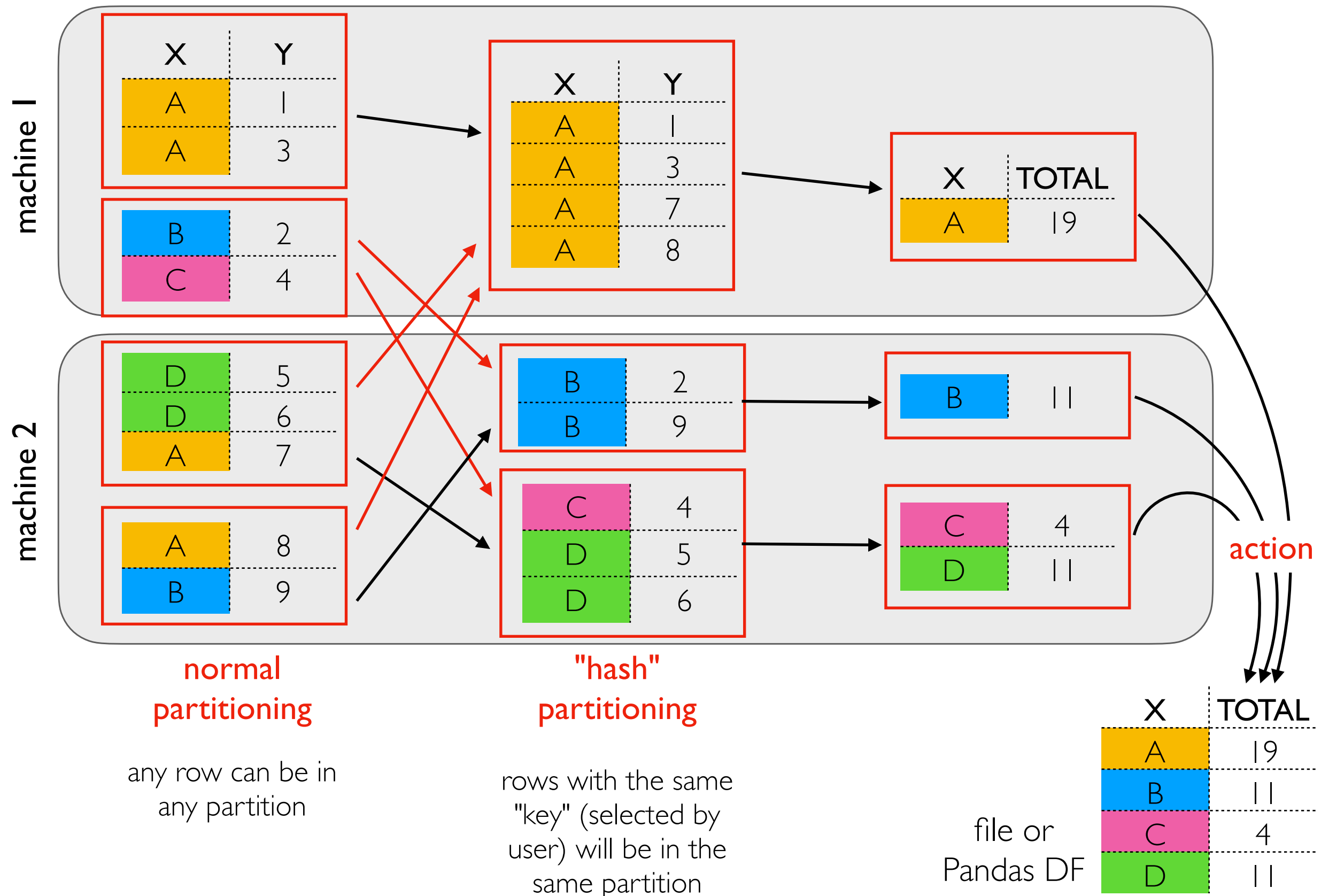
Spark: Physical Execution on Partitions

□ partition



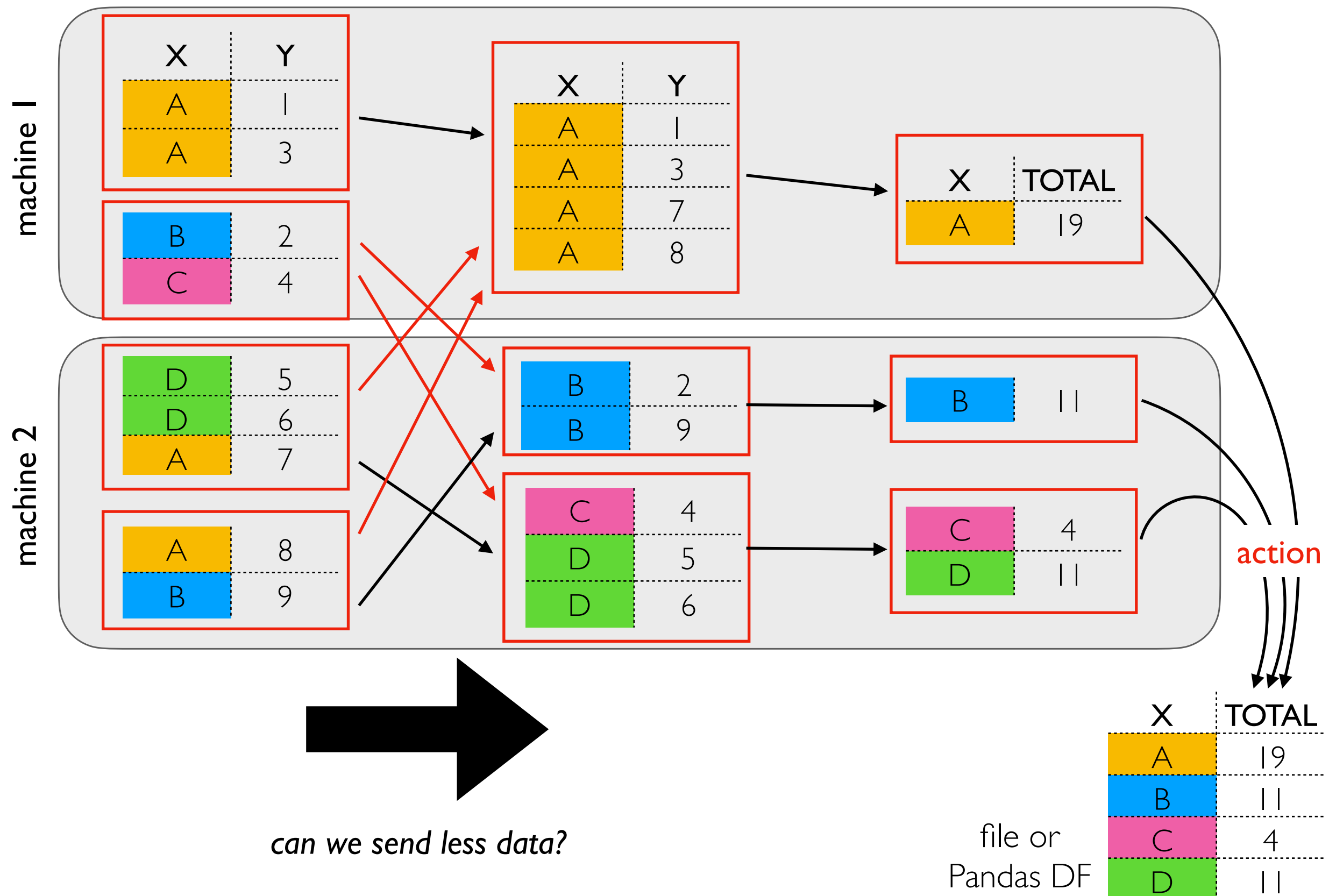
Spark: Physical Execution on Partitions

□ partition



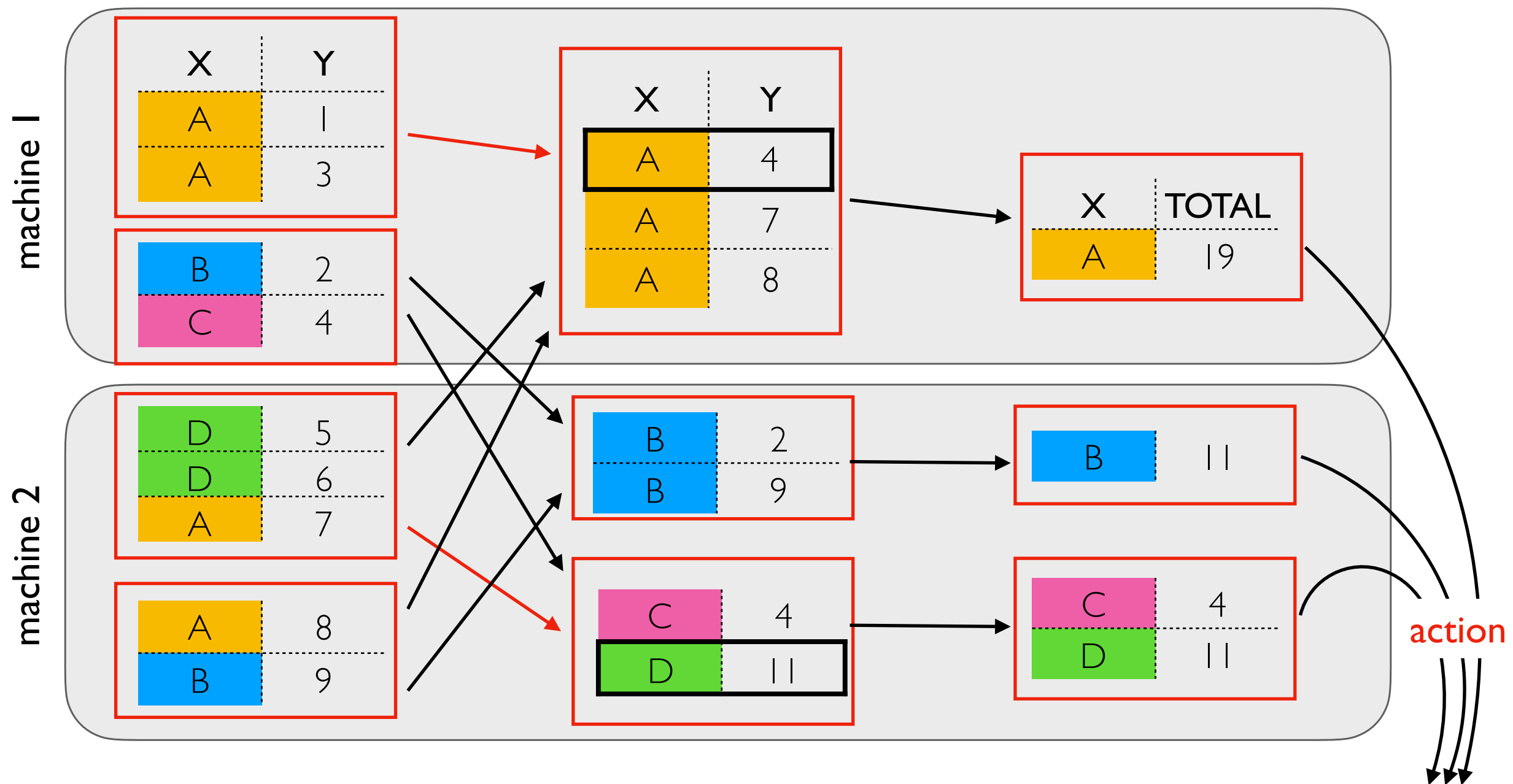
Spark: Physical Execution on Partitions

□ partition



Spark: Physical Execution on Partitions

□ partition



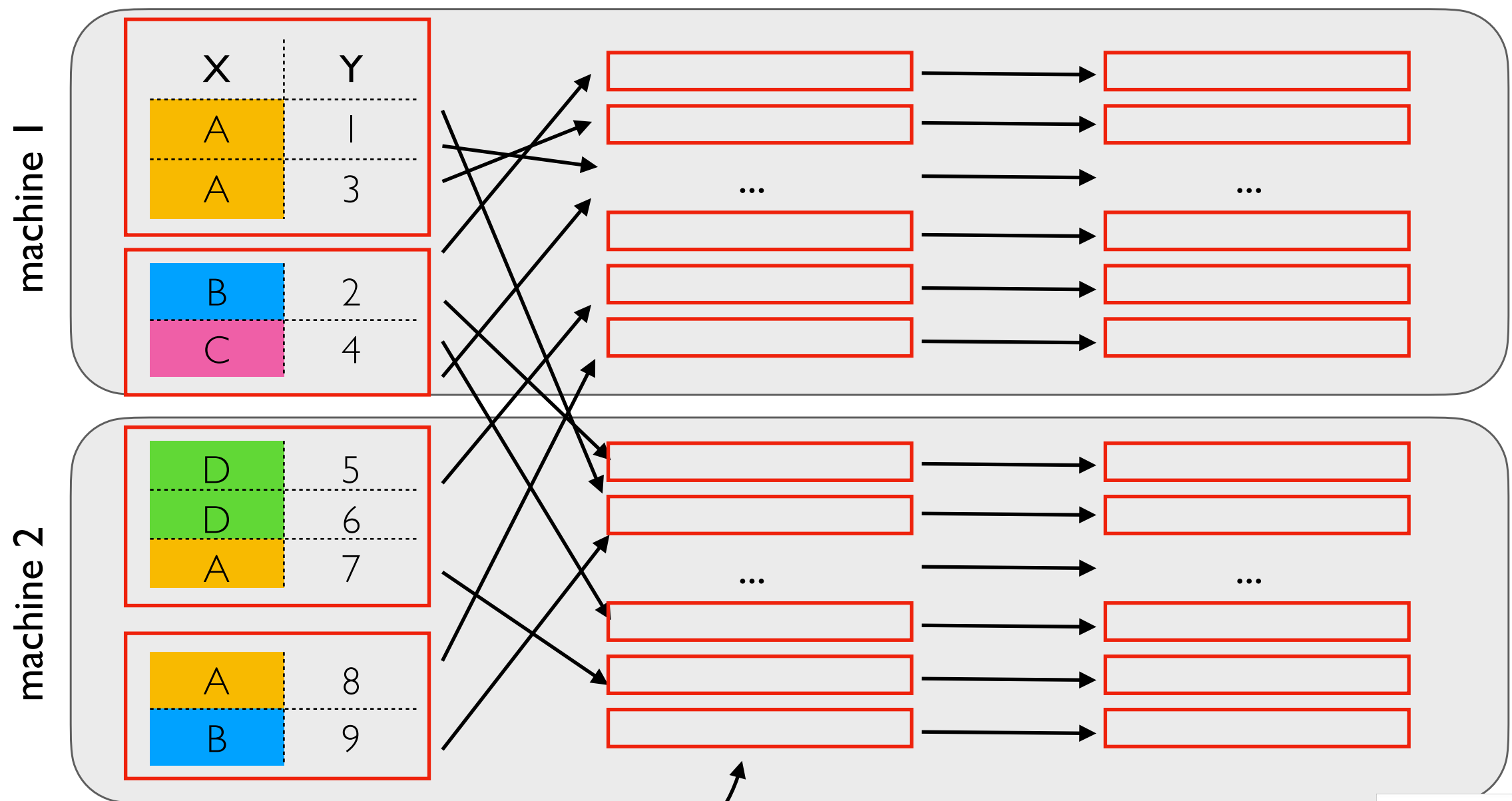
Optimization I: partial aggregates

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

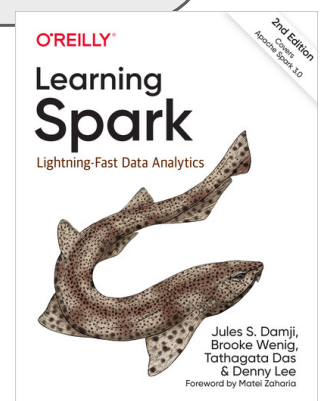
X	TOTAL
A	19
B	11
C	4
D	11

Shuffle Partitions



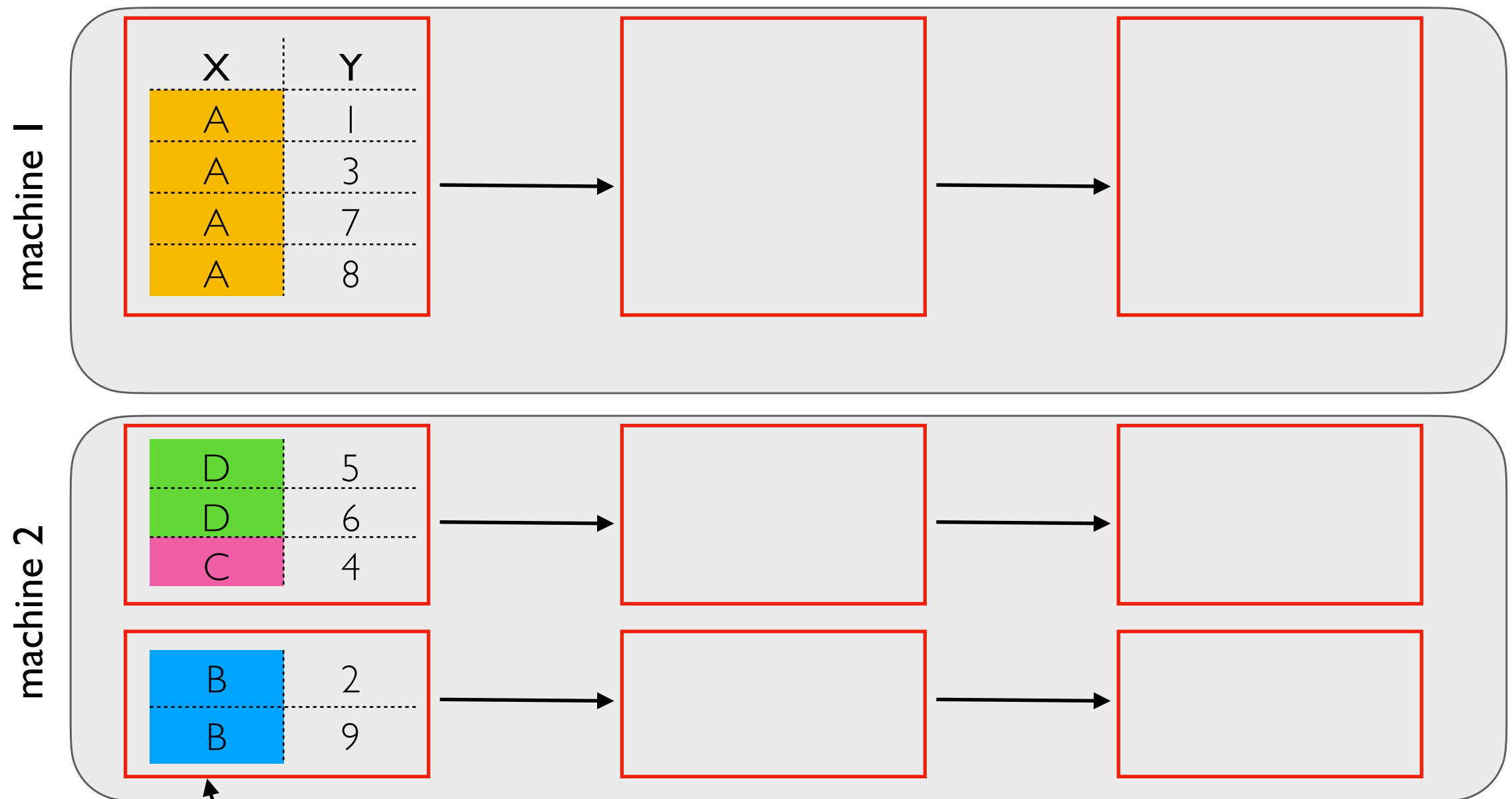
How many partitions will we have?

- `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 coalescing not available for Spark streaming (later lecture)



see Epilogue:
Apache Spark 3.0

Parquet: Bucketed Data



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

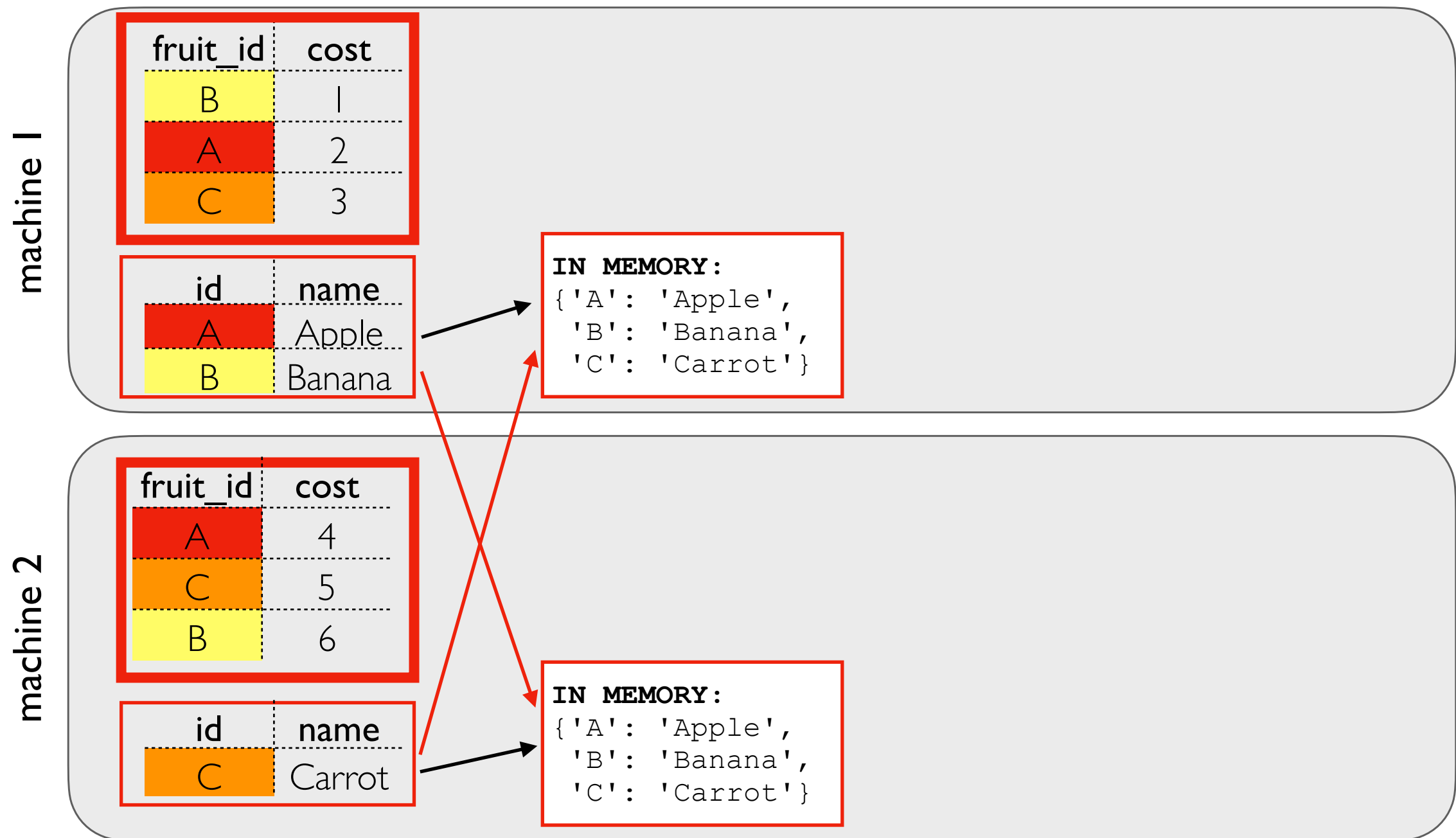
fruit_id	cost
A	4
C	5
B	6

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

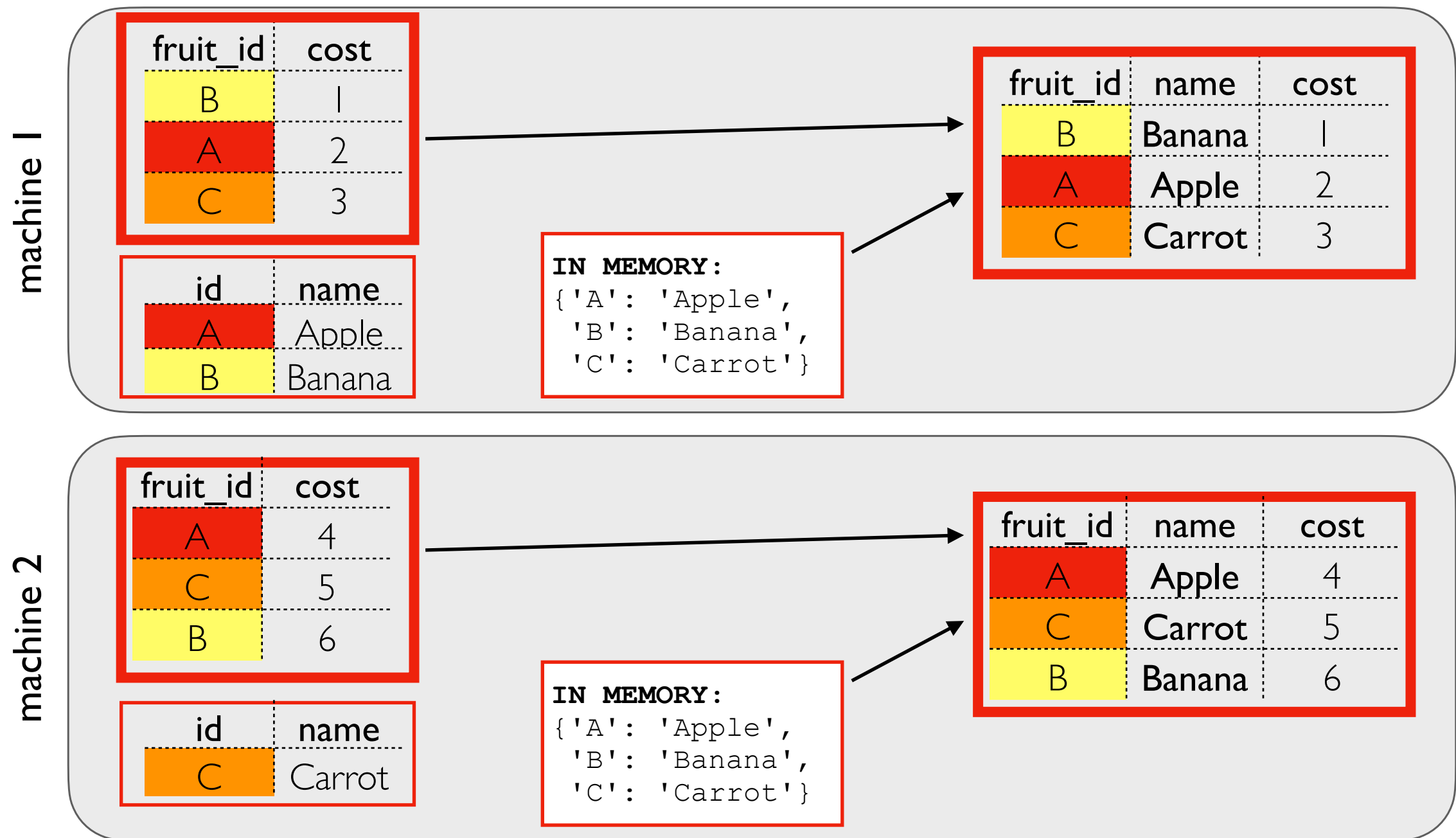


Broadcast step

- 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



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

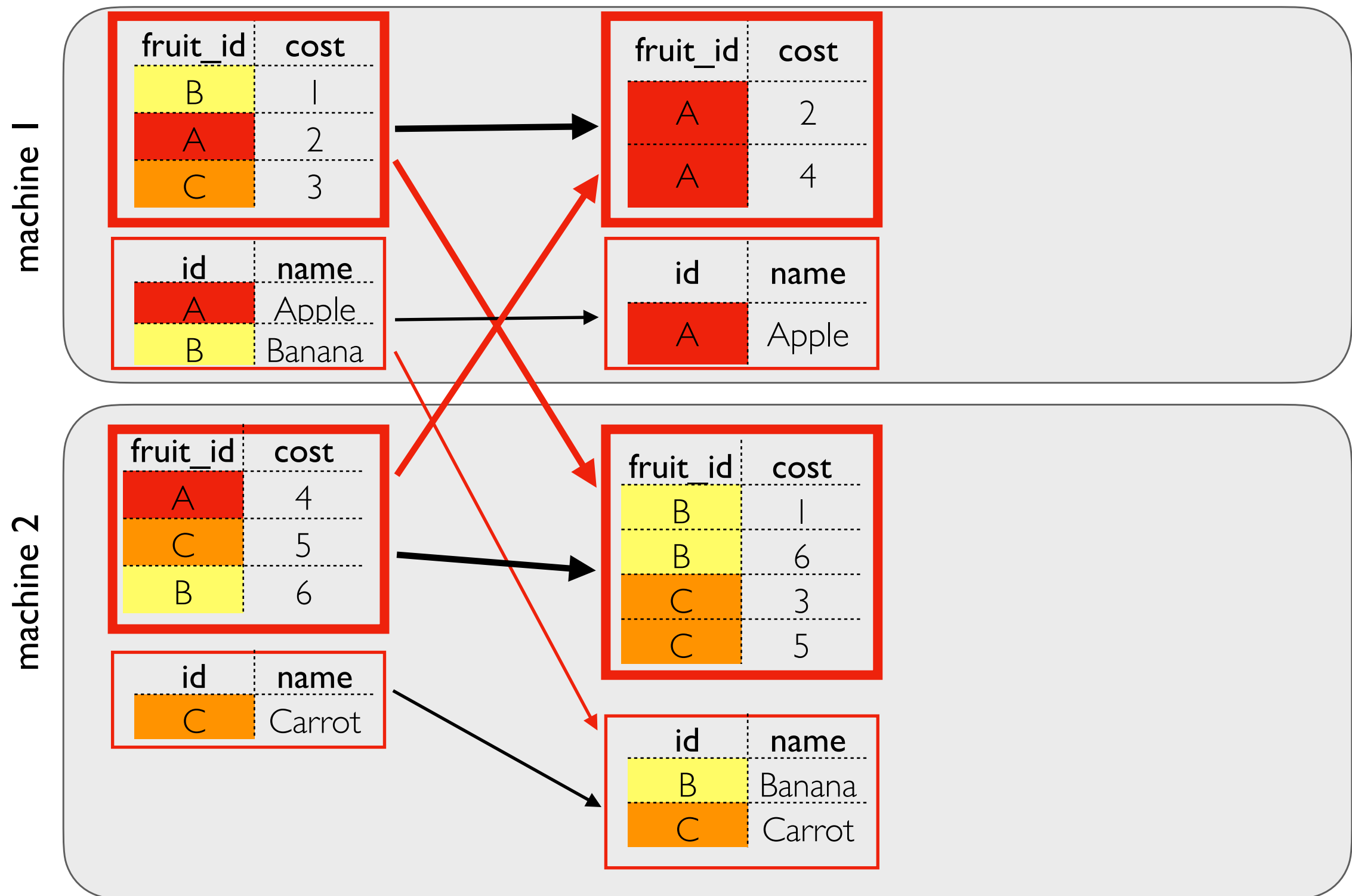
id	name
C	Carrot

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

SMJ: Shuffle Sort Merge Join

sorted within partitions

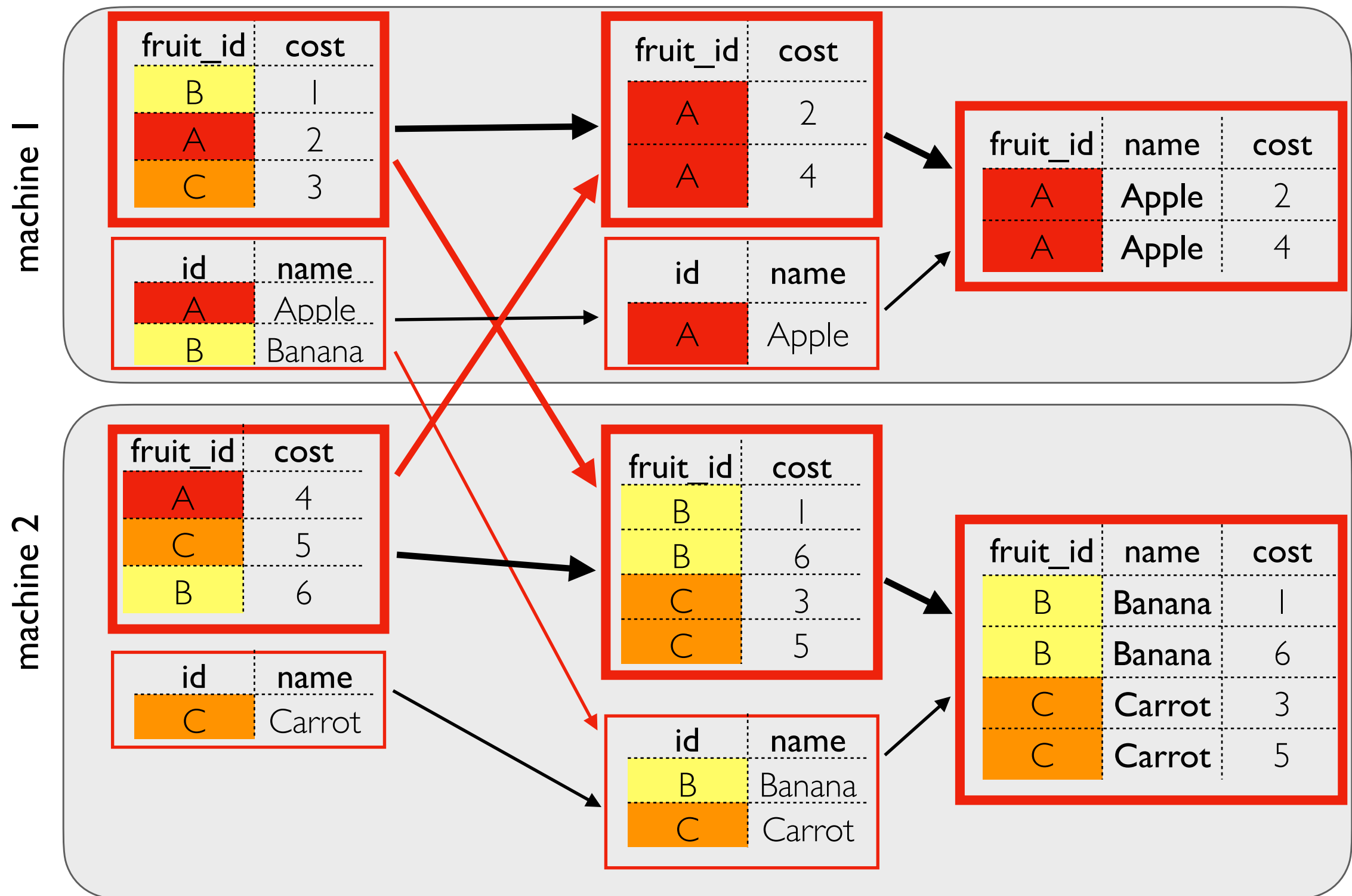
□ partition



Shuffle+Sort Step

SMJ: Shuffle Sort Merge Join

□ partition



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

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

+ Filter isnotnull(date#339)

+ FileScan csv (holidays2.csv)

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
          ...
```