Spark Internals and Performance

DS 5110/CS 5501: Big Data Systems

Spring 2024

Lecture 5b

Yue Cheng

University of Virginia

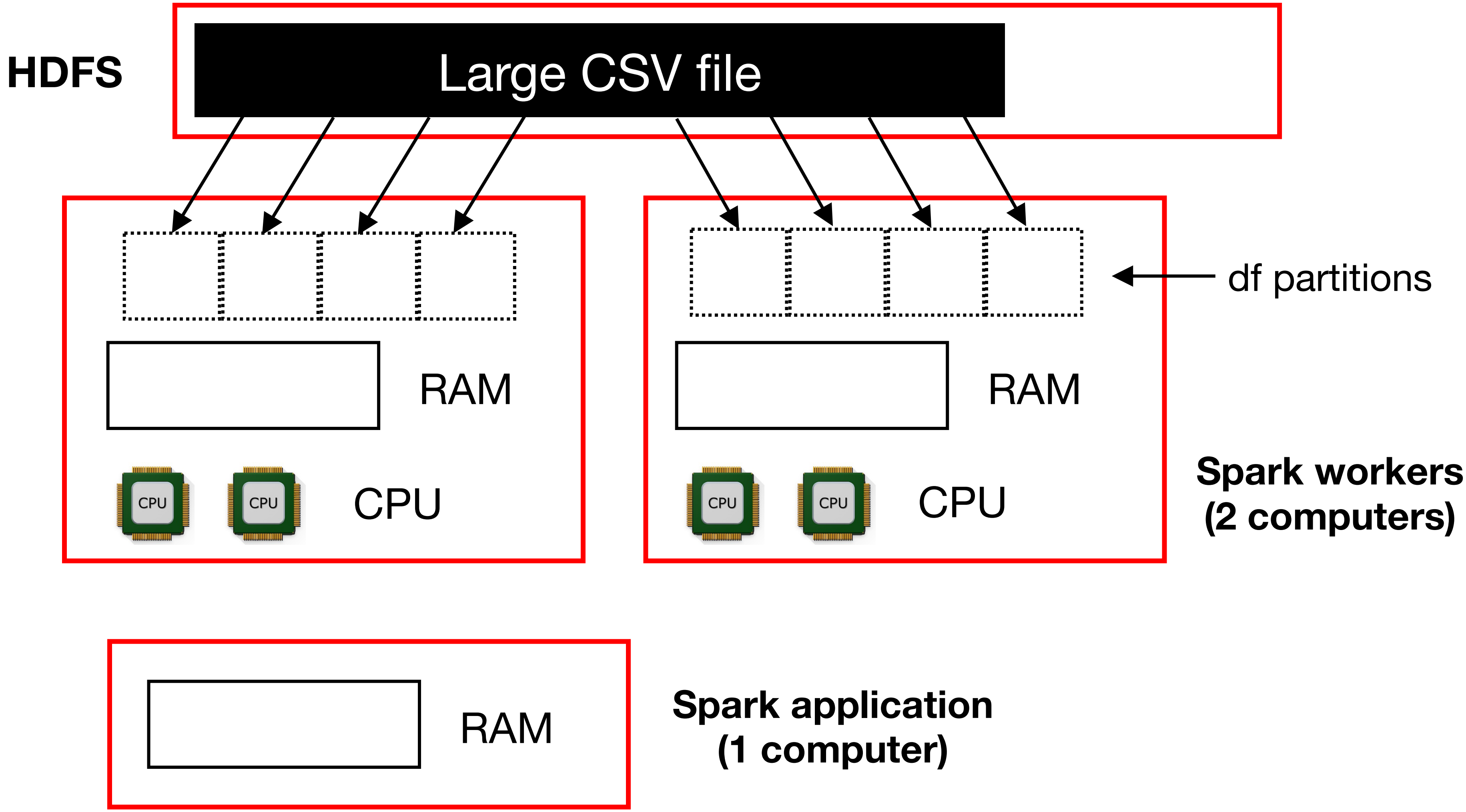
Learning objectives

- Know different storage and caching levels of Spark RDD
- Understand basic hash partitioning and join operations
- Describe the Spark implementation of the PageRank graph mining algorithm and roles various Spark optimizations play

Collecting data

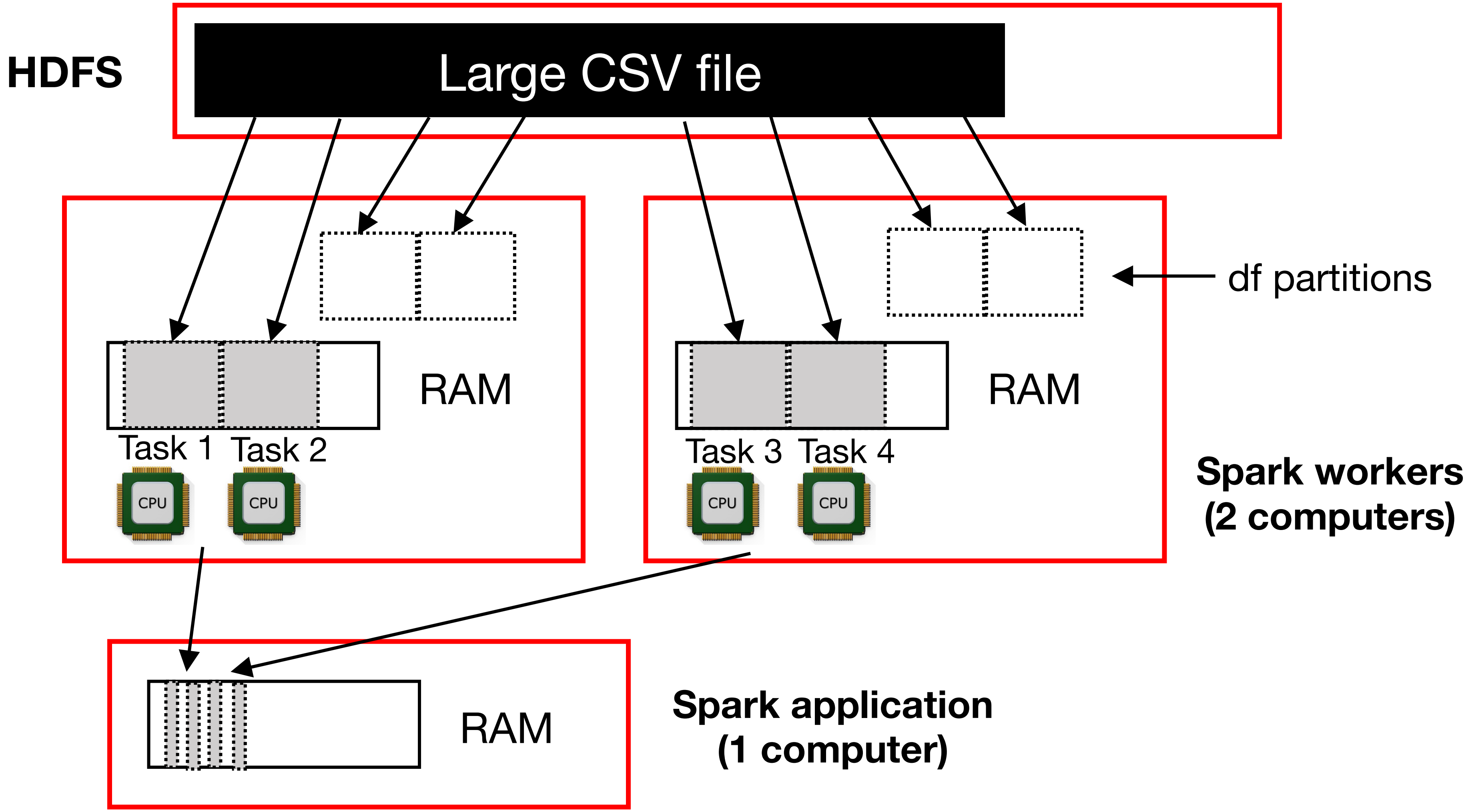
Collecting data (OK)

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



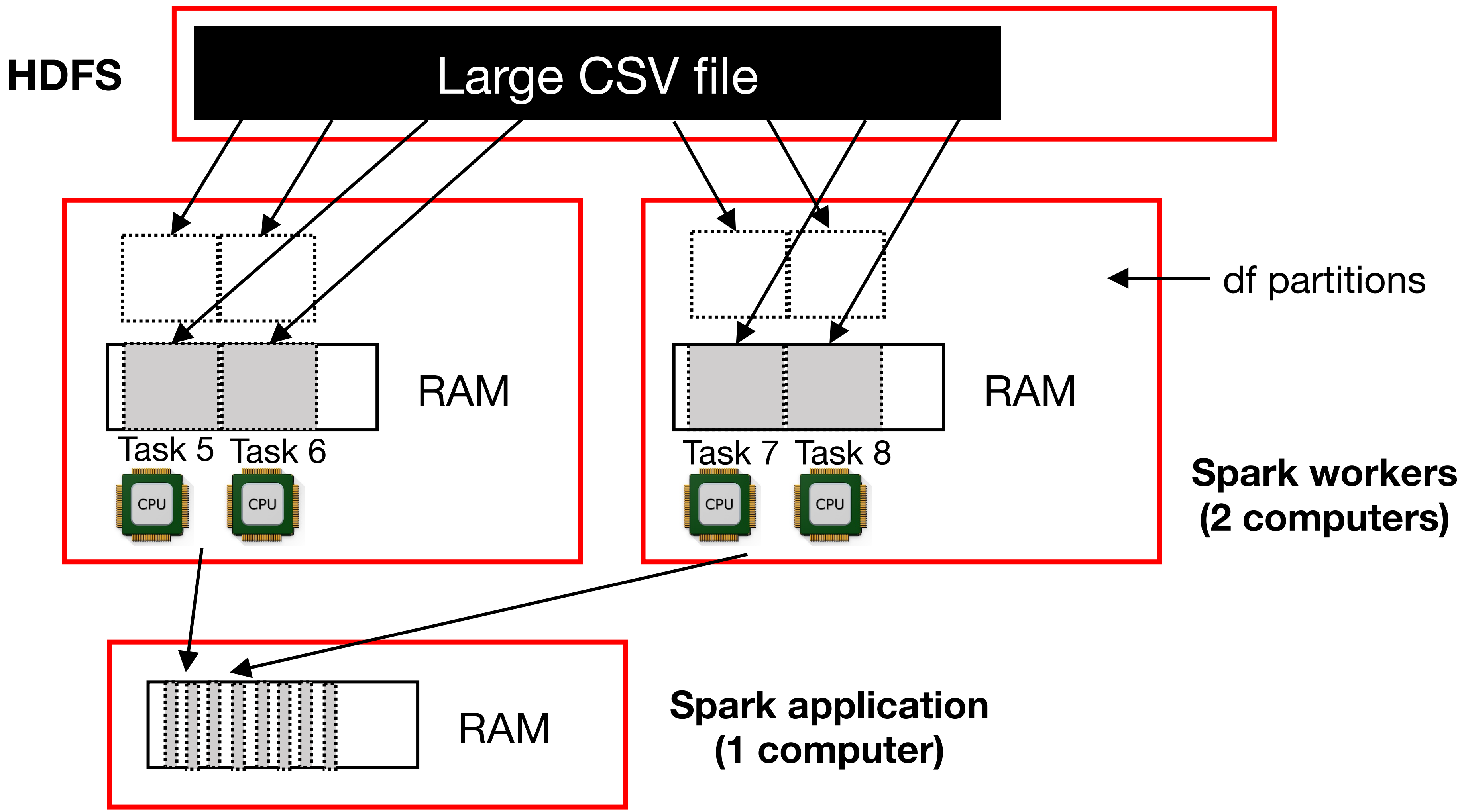
Collecting data (OK)

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



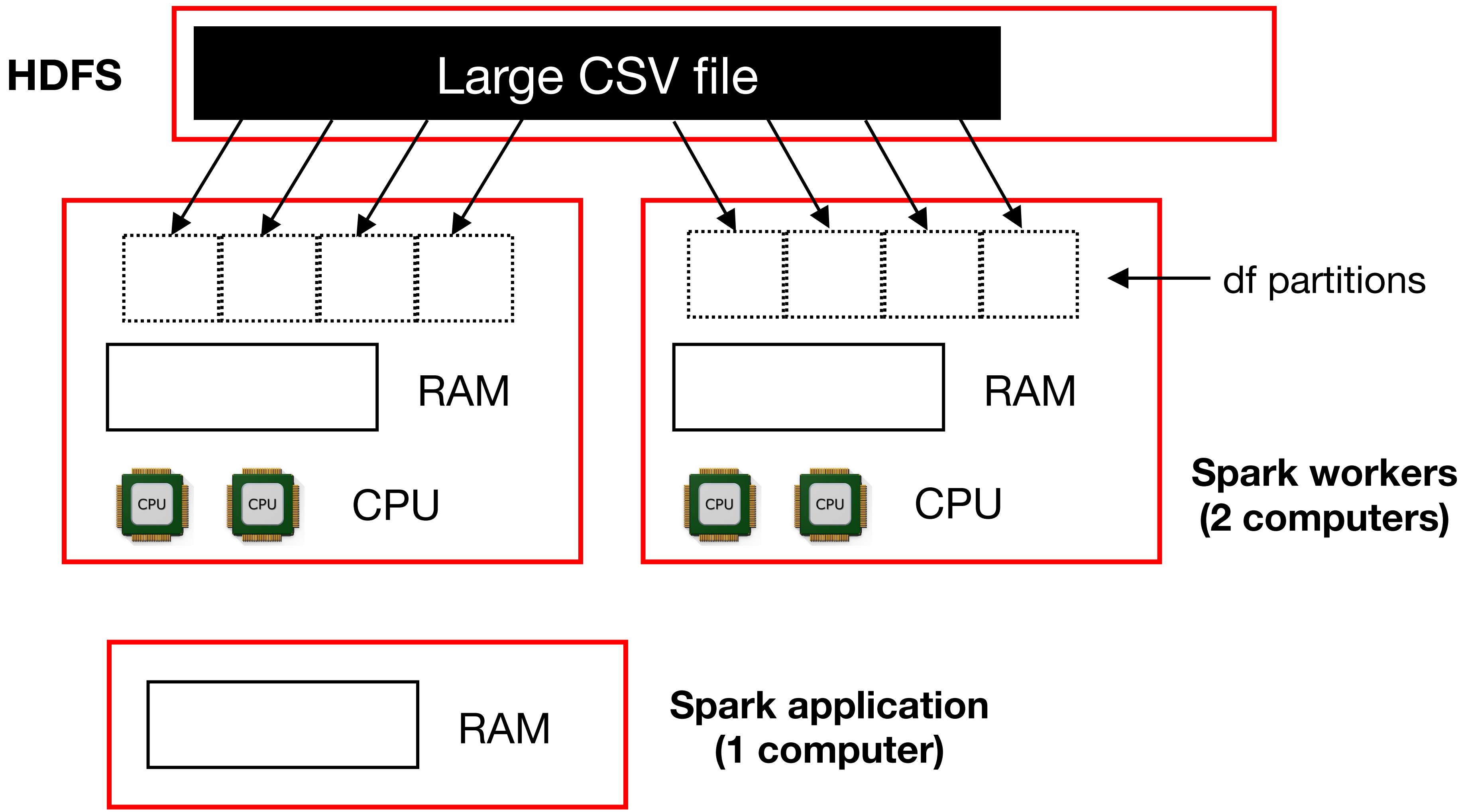
Collecting data (OK)

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



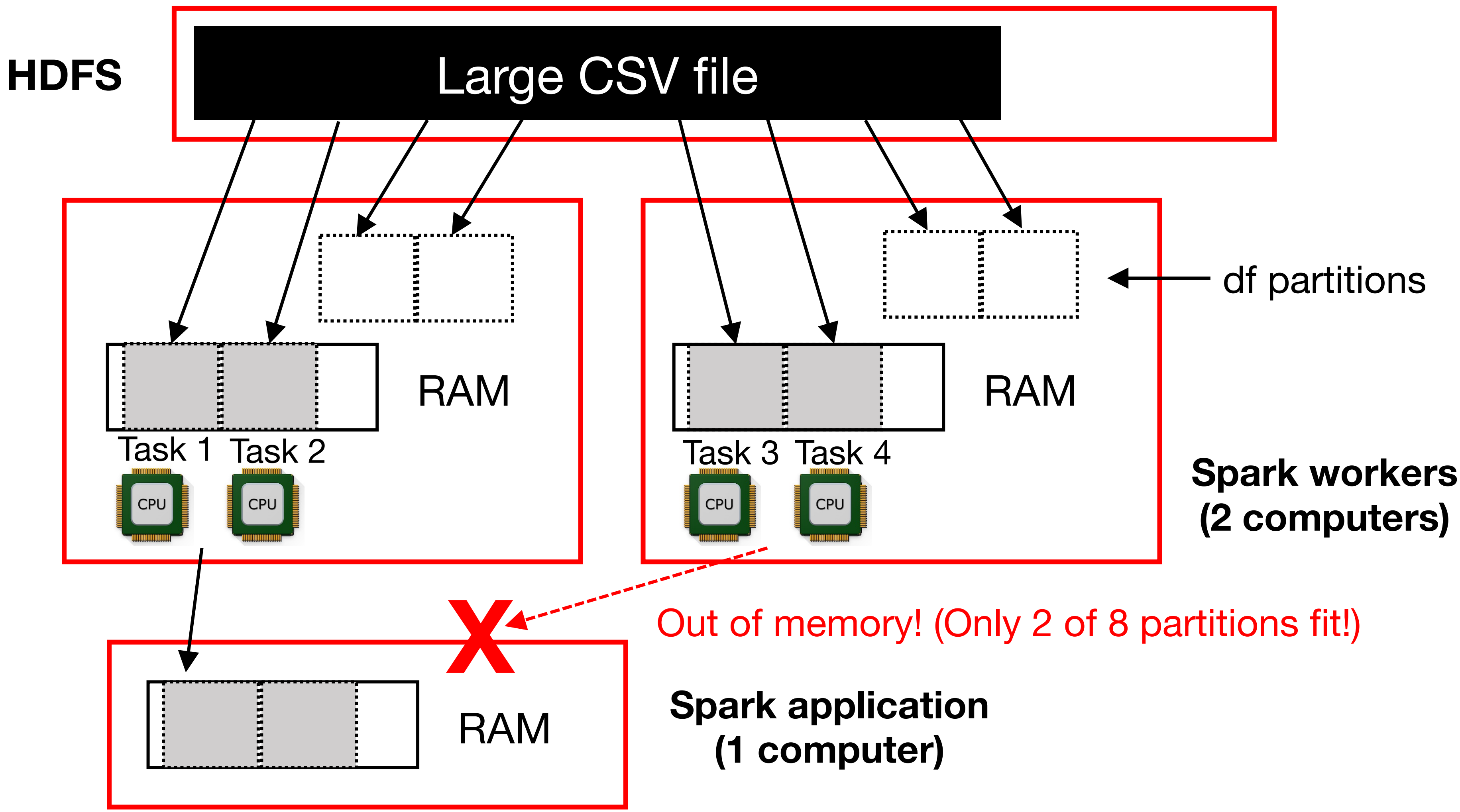
Collecting data (bad)

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



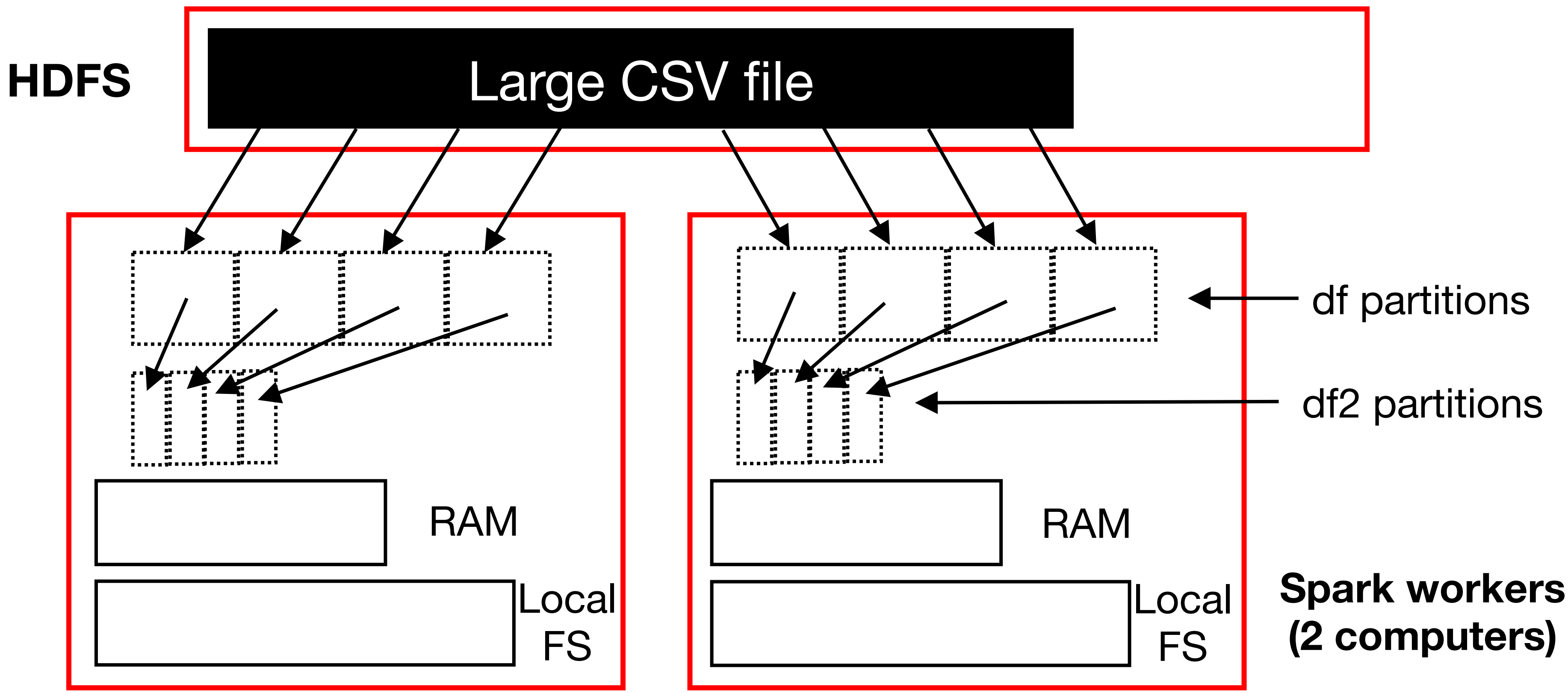
Collecting data (bad)

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



Persisting/Caching

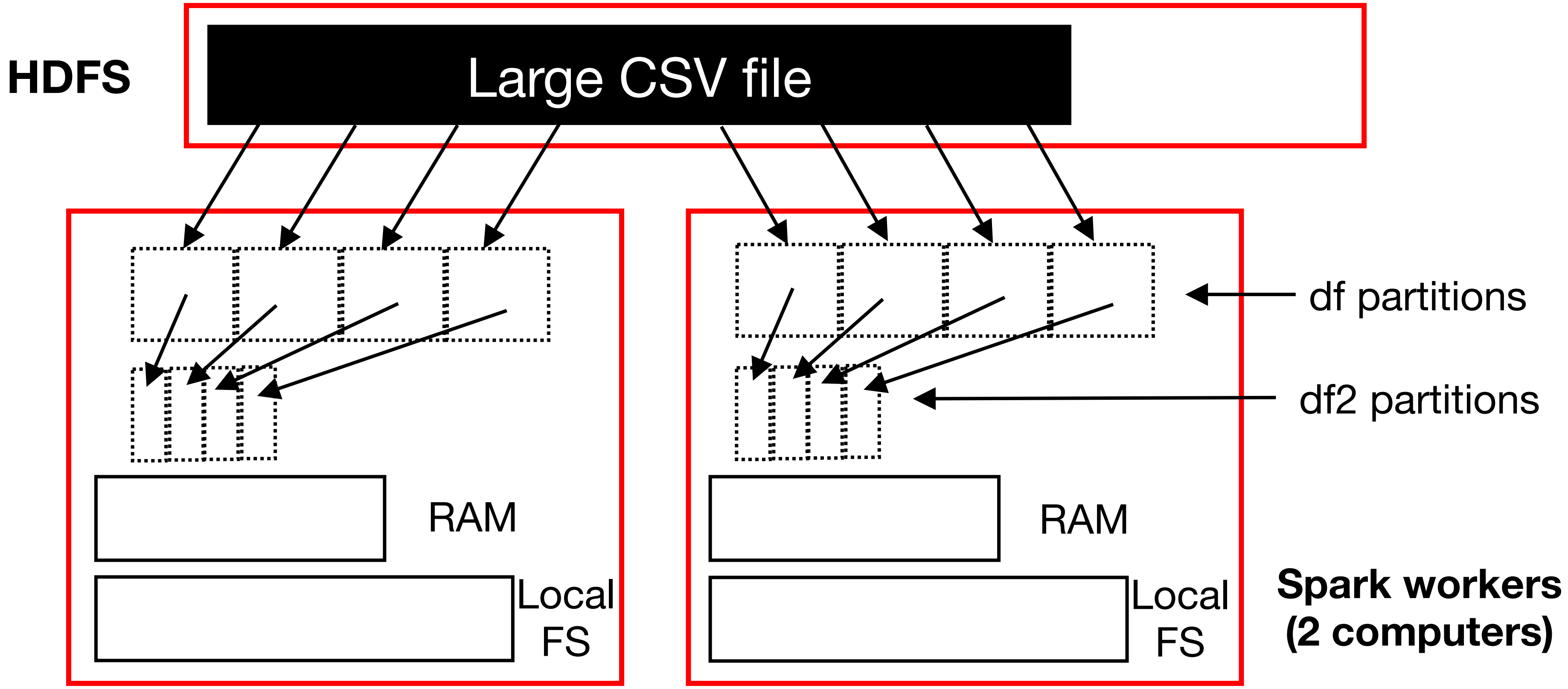
Persisting/Caching




```
# df refers to CSV file
df2 = df.where(???)
```

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

Persisting/Caching



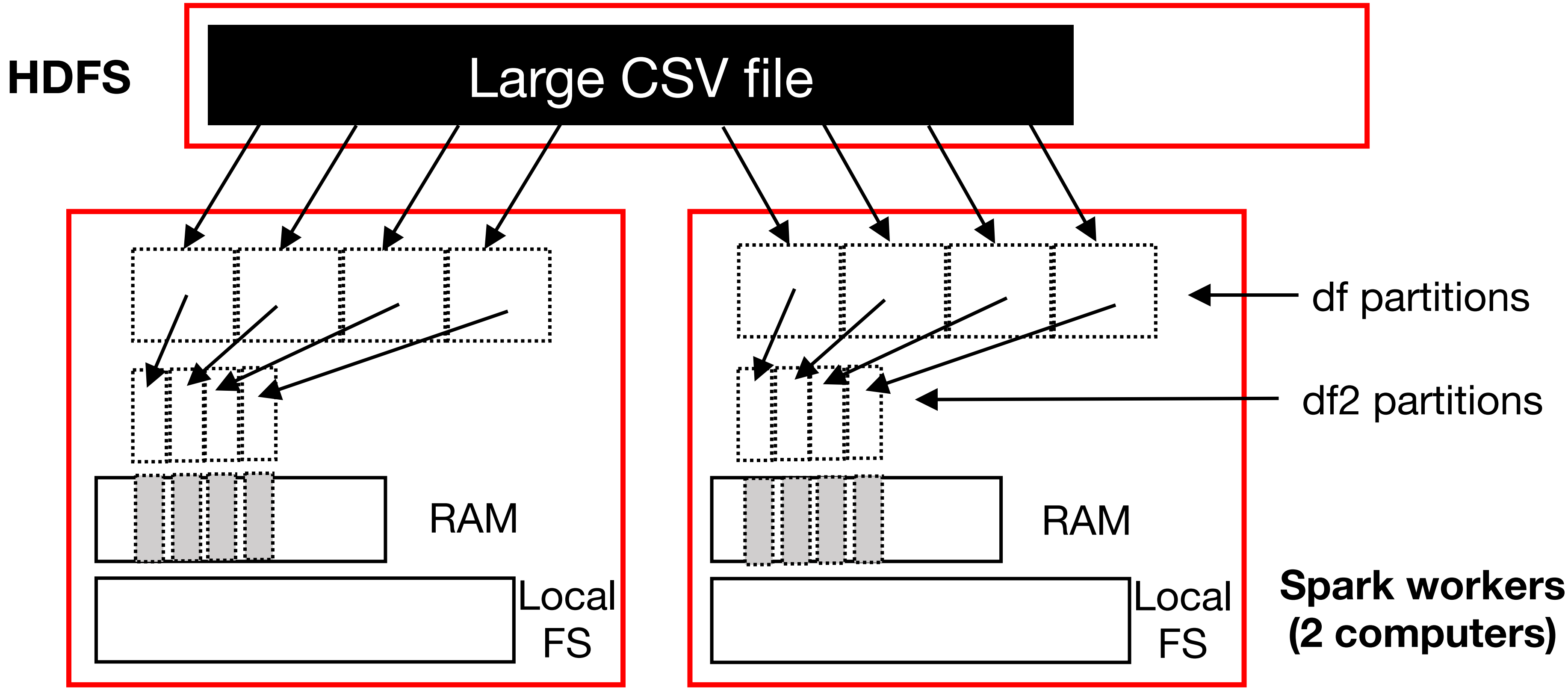

```
from pyspark.storagelevel import StorageLevel
df2 = df.where(???)

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

Persist level

- MEMORY_ONLY
- MEMORY_ONLY_SER
- DISK_ONLY

Persisting/Caching



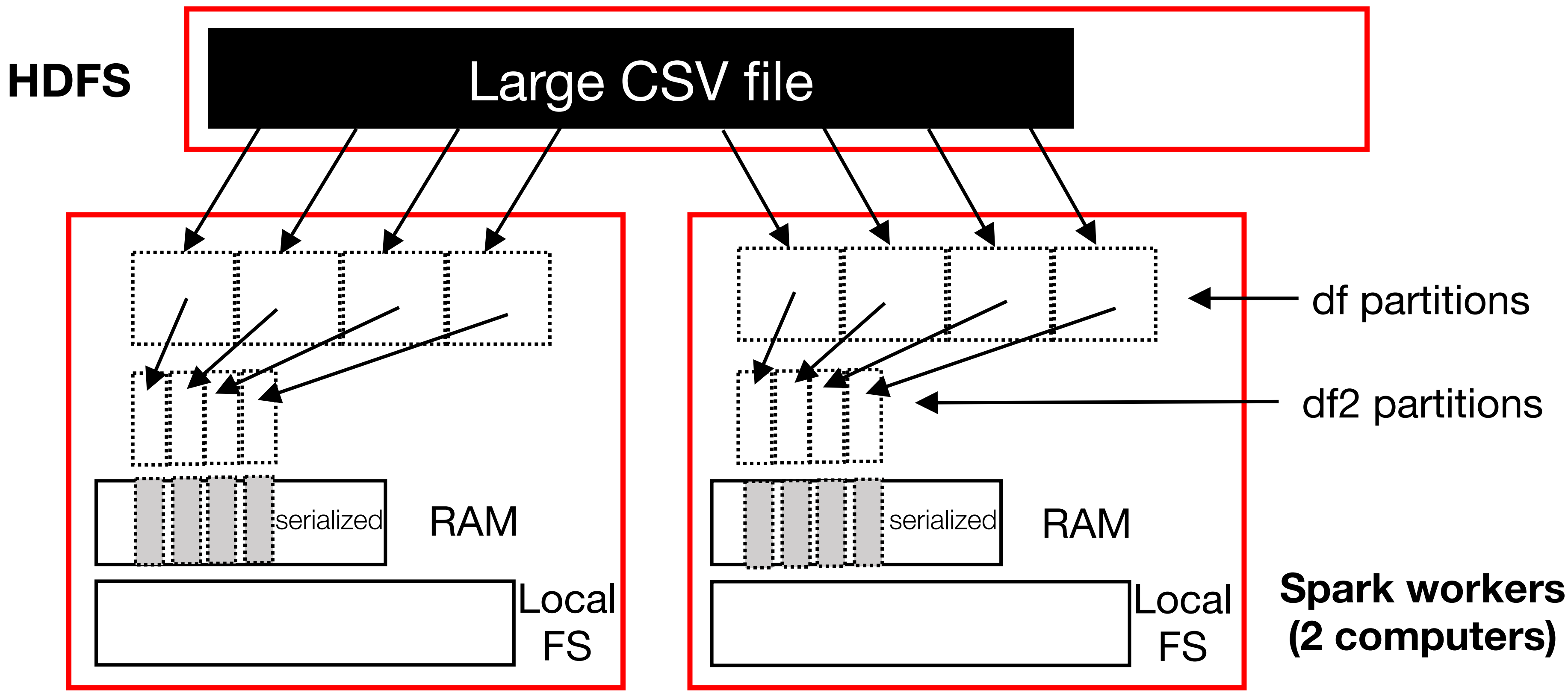

```
from pyspark.storagelevel import StorageLevel
df2 = df.where(???)

df2.persist(StorageLevel.???) # df.cache()
```

Persist level

- **MEMORY_ONLY**
- MEMORY_ONLY_SER
- DISK_ONLY

Persisting/Caching




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

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

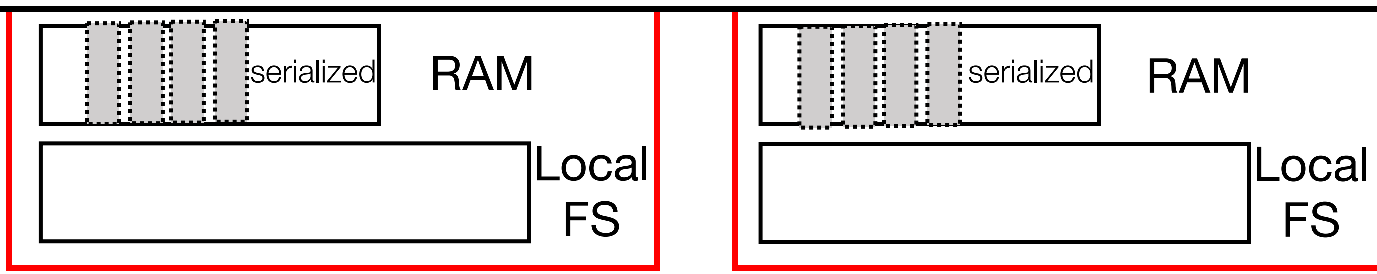
Persist level

- MEMORY_ONLY
- **MEMORY_ONLY_SER**
- DISK_ONLY

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`.



Spark workers (2 computers)

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

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

Persist level

- MEMORY_ONLY
- **MEMORY_ONLY_SER**
- DISK_ONLY

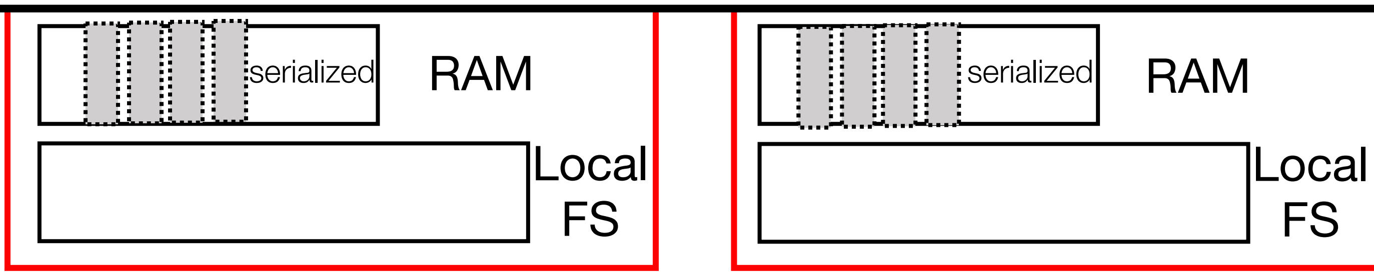
Documentation snippet: https://spark.apache.org/docs/2.2.2/tuning.html#memory-tuning

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

Documentation snippet: https://spark.apache.org/docs/2.2.2/tuning.html#serialized-rdd-storage

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.



Spark workers (2 computers)

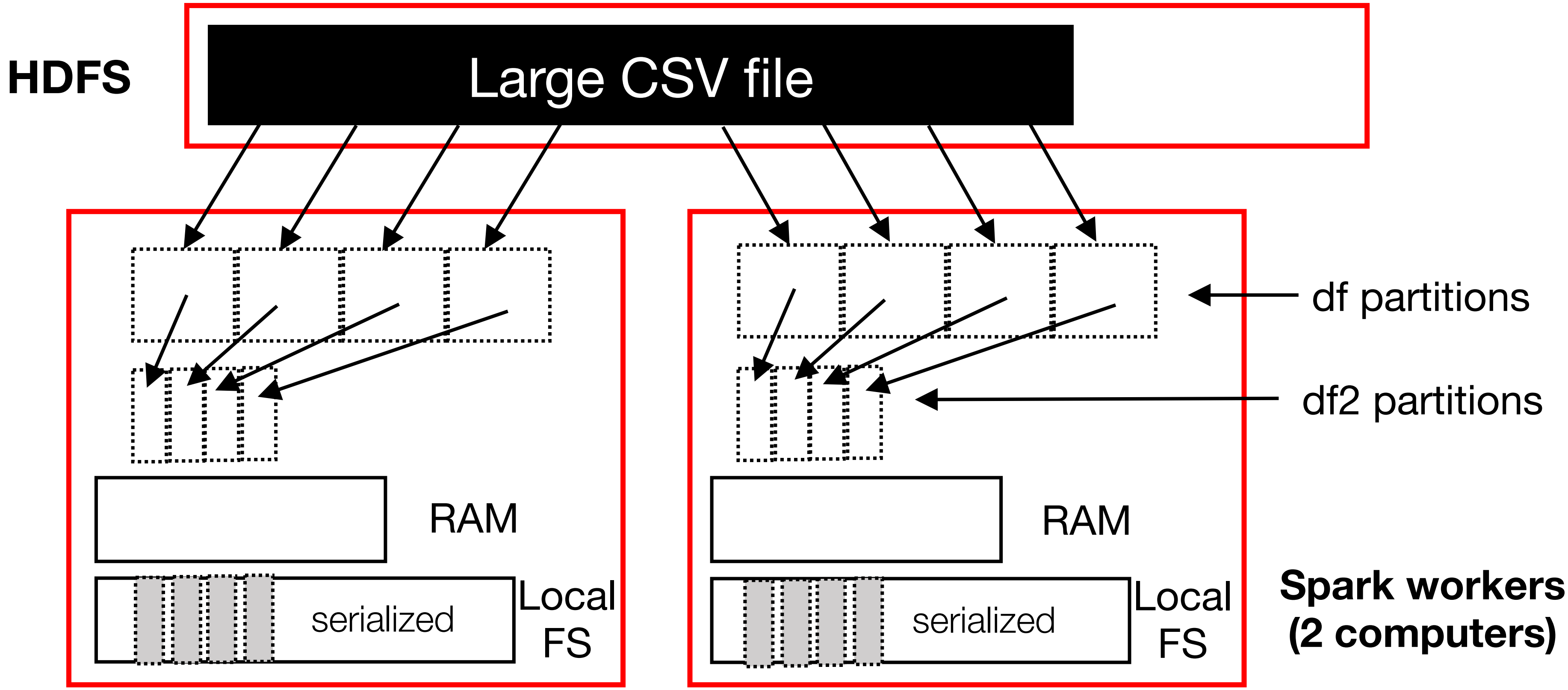
```
from pyspark.storagelevel import StorageLevel
df2 = df.where(???)

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

Persist level

- MEMORY_ONLY
- **MEMORY_ONLY_SER**
- DISK_ONLY

Persisting/Caching




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

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

Persist level

- MEMORY_ONLY
- MEMORY_ONLY_SER
- **DISK_ONLY**

Join and partitioning

Join and partitioning (best case)

Computer 1	Alice	5		Alice	F		Alice	5	F
	Bob	6	$\bowtie$	Bob	M	=	Bob	6	M
Computer 2	Claire	4		Claire	F		Claire	4	F

Join and partitioning (worst case)

Computer 1	Alice	5
	Bob	6
Computer 2	Claire	4

⋈

Alice	F
Bob	M
Claire	F

=

Alice	5	F
Bob	6	M
Claire	4	F

Computer 1	A	5
	A	2
	A	3
Computer 2	B	4
	B	1
	C	6
	C	8

⋈

C	5
B	2
A	3
B	4
A	1
B	6
C	8

If partitioning doesn't match, then need to shuffle (**via network**) to match pairs.

Join and partitioning (optimization)

Computer 1	Alice	5
	Bob	6
Computer 2	Claire	4

⋈

Alice	F
Bob	M
Claire	F

=

Alice	5	F
Bob	6	M
Claire	4	F

Computer 1	A	5
	A	2
	A	3
Computer 2	B	4
	B	1
	C	6
	C	8

⋈

A	3
A	1
B	2
B	4
B	6
C	5
C	8

`partitionBy()` is specific to key-value pair RDDs. It is used to partition RDDs based on keys, by default using a **hash partitioner**.

Example: PageRank

Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each dest page's rank to $\Sigma_{i \in neighbors}$ $rank_{neighbor\_i}$ / $|neighbors_i|$

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each dest page's rank to $\Sigma_{i \in neighbors}$ $rank_{neighbor\_i}$ / $|neighbors_i|$

RDD[(URL, Seq[URL])]

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

RDD[(URL, Rank)]

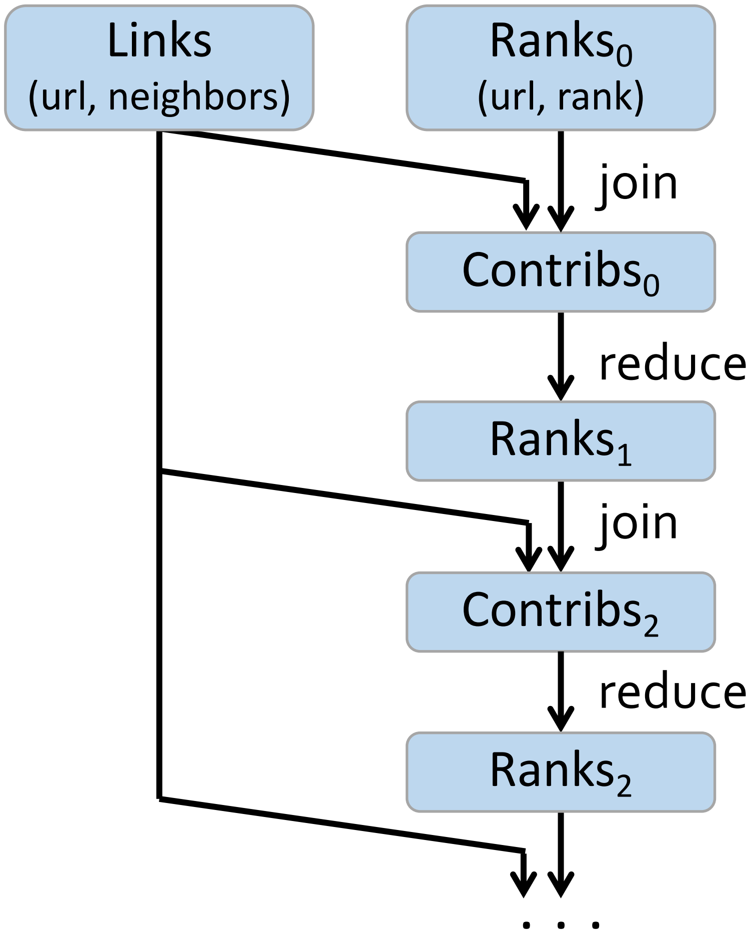
RDD[(URL, (Seq[URL],Rank))]

For each neighbor in links emits (URL, RankContrib)

Reduce to RDD[(URL, Rank)]

Demo …

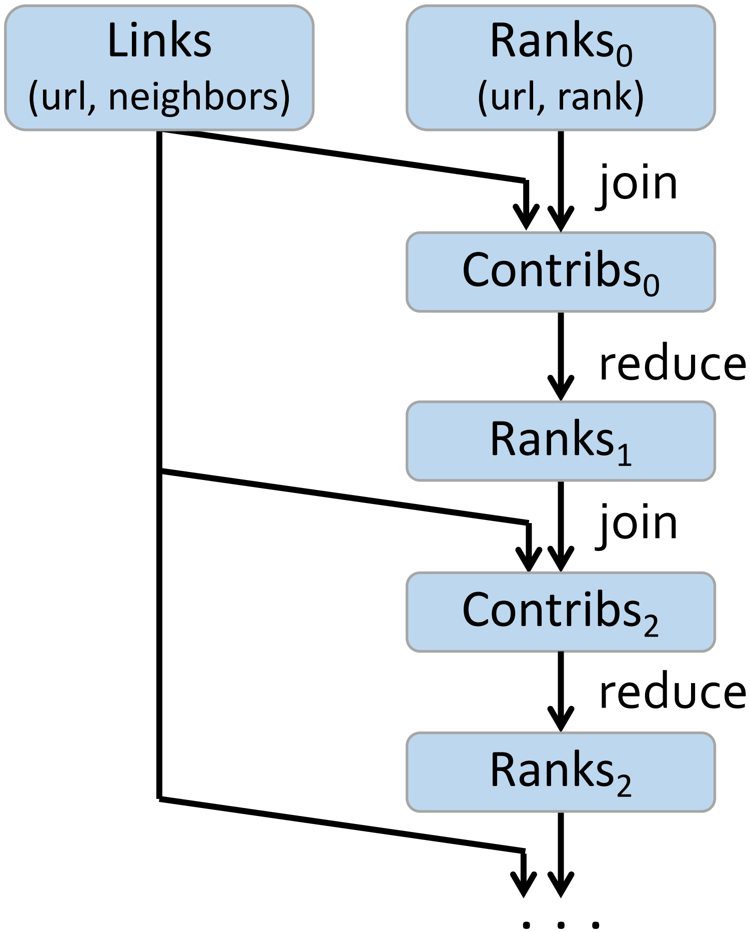
Optimizing placement



- `links` & `ranks` repeated joined
- Can co-partition them (e.g., hash both on source URLs) to avoid shuffles

```
links = links.partitionBy(N)
ranks = ranks.partitionBy(N)
```

Optimizing placement



- `links` & `ranks` repeated joined
- Can co-partition them (e.g., hash both on source URLs) to avoid shuffles

```
links = links.partitionBy(N)
ranks = ranks.partitionBy(N)
```

Q1: Should we apply `.persist(DISK_ONLY)` to `links` or `ranks`?

Q2: Where might we have placed `.persist(DISK_ONLY)`?

Discussion: Spark perf (paper)