Optimizing Apache Spark SQL Joins

Vida Ha
Solutions Architect

About Me

2005 **Mobile Web & Voice Search** 

About Me

2005 **Mobile Web & Voice Search**

2012 **Reporting & Analytics**

About Me

2005 **Mobile Web & Voice Search**

2012 **Reporting & Analytics**

2014 **Solutions Architect**

Evolution of Spark…

2014:

- **Spark 1.x**
- **RDD based API's.**
- **Everyday I'm Shufflin'**

2017:

- **Spark 2.x**
- **Dataframes & Datasets**
- **Adv SQL Catalyst**
- **Optimizing Joins**

Spark SQL Joins

```
SELECT …
FROM TABLE A
JOIN TABLE B
ON A.KEY1 = B.KEY2
```

Topics Covered Today

Basic Joins:

- **Shuffle Hash Join**
  - **Troubleshooting**
- **Broadcast Hash Join**
- **Cartesian Join**

Special Cases:

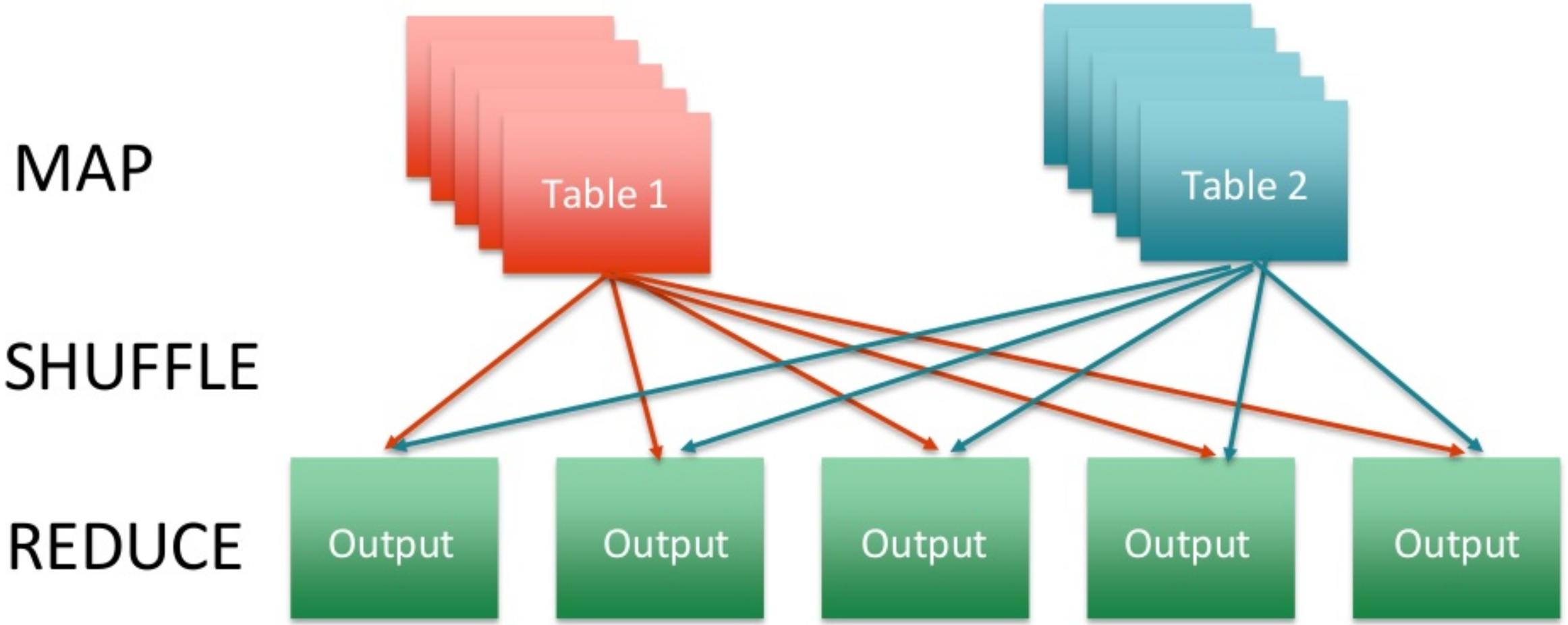
- **Theta Join**
- **One to Many Join**

Shuffle Hash Join

A Shuffle Hash Join is the most basic type of join, and goes back to Map Reduce Fundamentals.

- **Map** through two different data frames/tables.
- Use the fields in the join condition as the **output key**.
- **Shuffle** both datasets by the output key.
- In the **reduce** phase, join the two datasets now any rows of both tables with the same keys are on the same machine and are sorted.

Shuffle Hash Join



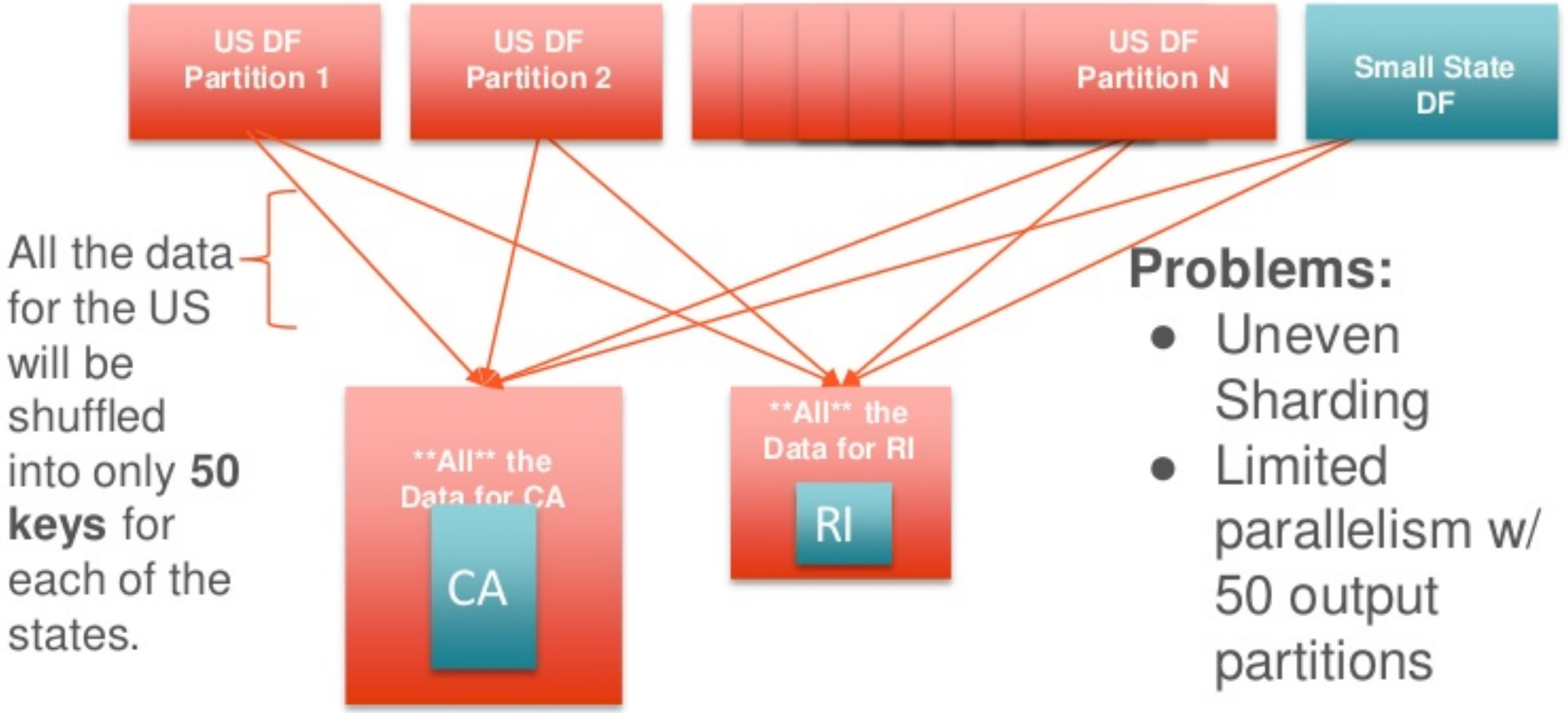
Shuffle Hash Join Performance

Works best when the DF's:

- Distribute evenly with the key you are joining on.
- Have an adequate number of keys for parallelism.

```
join_rdd = sqlContext.sql("select *
  FROM people_in_the_us
  JOIN states
  ON people_in_the_us.state = states.name")
```

Uneven Sharding & Limited Parallelism,



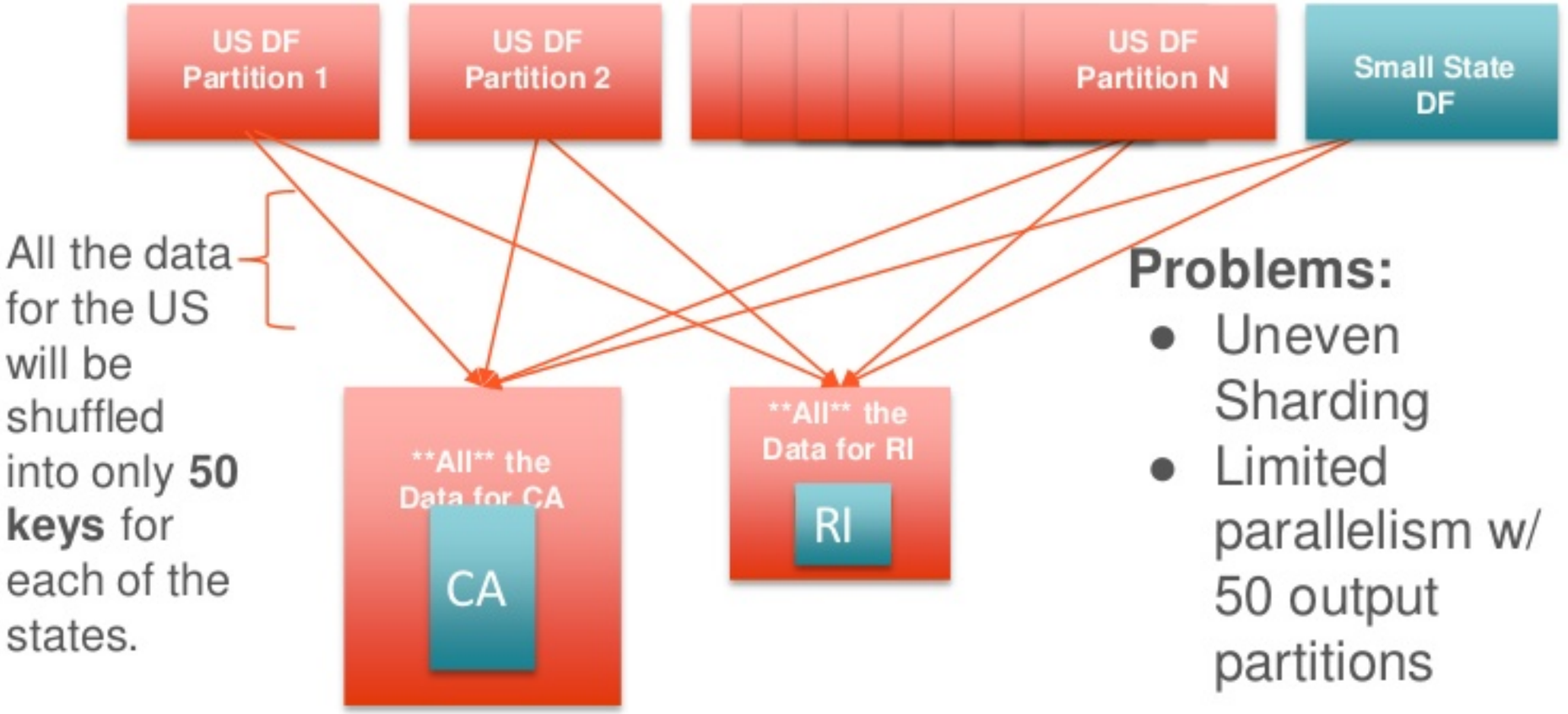
Problems:

- Uneven Sharding
- Limited parallelism w/ 50 output partitions

A larger Spark Cluster will not solve these problems!

databricks

Uneven Sharding & Limited Parallelism,



Broadcast Hash Join can address this problem if one DF is small enough to fit in memory.

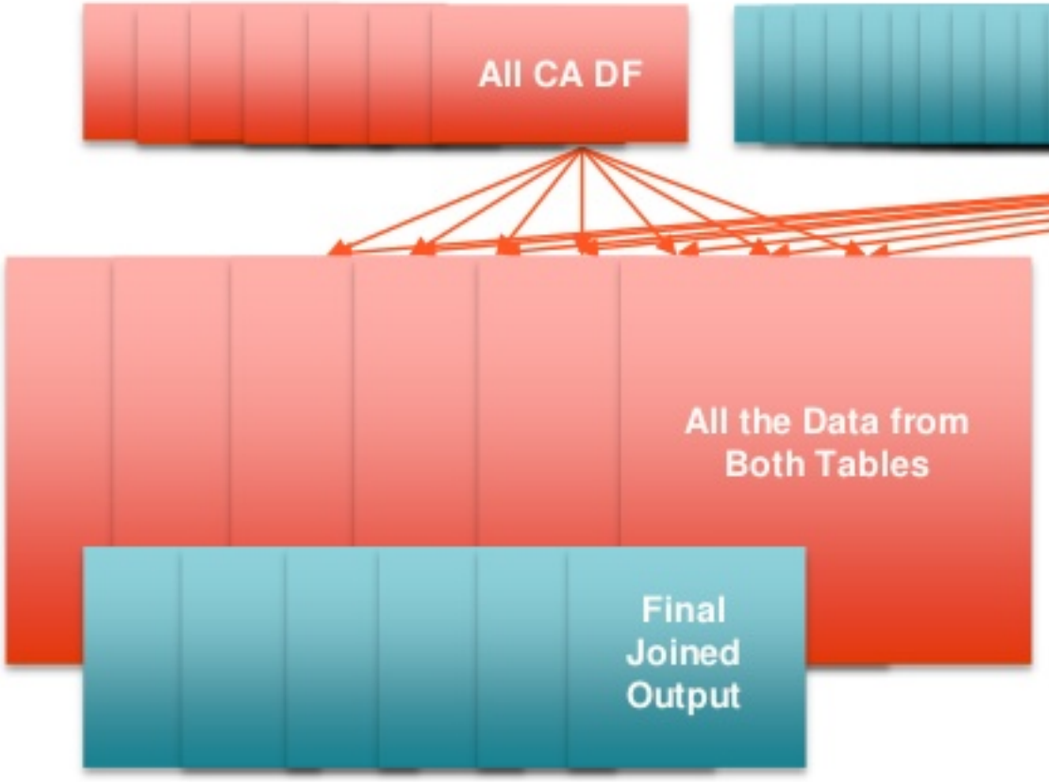
databricks

More Performance Considerations

```
join_rdd = sqlContext.sql(“select *
  FROM people_in_california
  LEFT JOIN all_the_people_in_the_world
  ON people_in_california.id =
     all_the_people_in_the_world.id”)
```

Final output keys = # of people in CA, so don’t need a huge Spark cluster, right?

Left Join - Shuffle Step



Shuffles everything before dropping keys

Not a Problem:

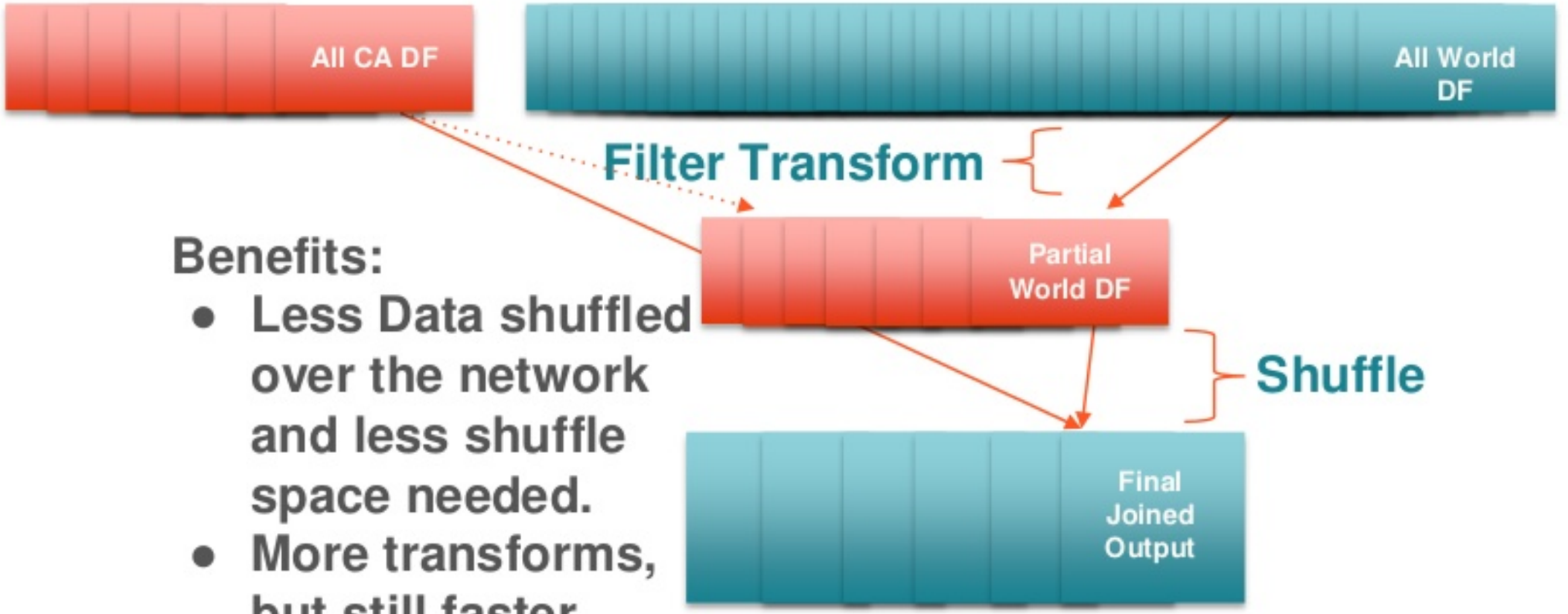
- Even Sharding
- Good Parallelism

The Size of the Spark Cluster to run this job is limited by the Large table rather than the Medium Sized Table.

databricks

A Better Solution

Filter the World DF for only entries that match the CA ID



Benefits:

- **Less Data shuffled over the network and less shuffle space needed.**
- **More transforms, but still faster.**

databricks

What's the Tipping Point for Huge?

- Can't tell you.
- There aren't always strict rules for optimizing.
- If you were only considering two small columns from the World RDD in Parquet format, the filtering step may not be worth it.

You should understand your data and it's unique properties in order to best optimize your Spark Job.

In Practice: Detecting Shuffle Problems

Check the Spark UI pages for task level detail about your Spark job.

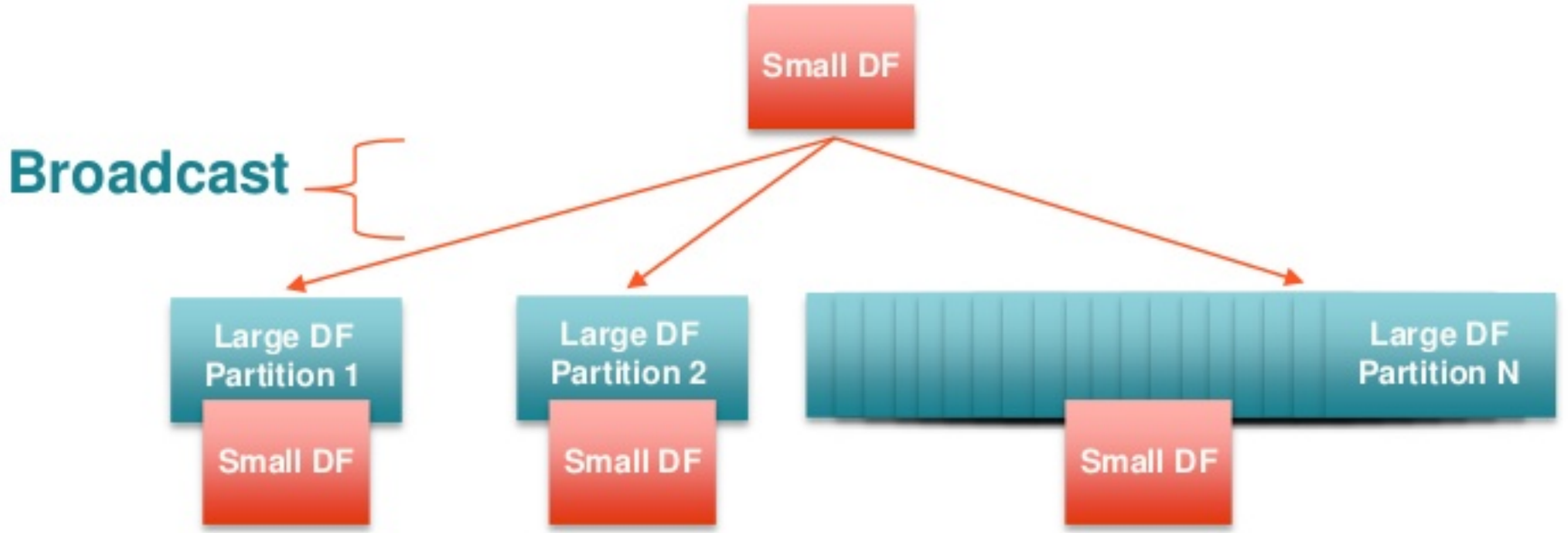
Things to Look for:

- Tasks that take much longer to run than others.
- Speculative tasks that are launching.
- Shards that have a lot more input or shuffle output.

databricks

Broadcast Hash Join

Optimization: When one of the DF's is small enough to fit in memory on a single machine.



Parallelism of the large DF is maintained (n output partitions), and shuffle is not even needed.

Broadcast Hash Join

- Often optimal over Shuffle Hash Join.
- Use “**explain**” to determine if the Spark SQL catalyst hash chosen Broadcast Hash Join.
- Should be automatic for many Spark SQL tables, may need to provide hints for other types.

Cartesian Join

- A cartesian join can easily explode the number of output rows.

  100,000 X 100,000 = 10 Billion

- Alternative to a full blown cartesian join:
  - Create an RDD of UID by UID.
  - Force a Broadcast of the rows of the table .
  - Call a UDF given the UID by UID to look up the table rows and perform your calculation.
- Time your calculation on a sample set to size your cluster.

One To Many Join

- A single row on one table can map to many rows on the 2nd table.
- Can explode the number of output rows.
- Not a problem if you use parquet - the size of the output files is not that much since the duplicate data encodes well.

Theta Join

```
join_rdd = sqlContext.sql(“select *
  FROM tableA
  JOIN tableB
  ON (keyA < keyB + 10)”)
```

- Spark SQL consider each keyA against each keyB in the example above and loop to see if the theta condition is met.
- Better Solution - create buckets for keyA and KeyB can be matched on.

Thank you

Questions?