# Processing and Aggregation

Module 6

# Introduction: Processing

- All operations on data are processing
  - Processing is done on **records** in a **dataset**
- Examine and Understand
    - *Types* of processing
    - *Methods* of processing
    - *Stages* in a processing job
- Processing can be optimized by considering *pipelines*
  - Learn about and understand processing pipelines
    - **Map-Reduce** patterns
    - **Directed Acyclic Graphs (DAGs)**
  - Understand how DAGs can be optimized

# Types of Processing Operations

What kind of processing can we perform?

Filter

Exclude some records from the set

Mutate

Transform records into different records

Aggregate

Produce a new set of records which aggregates features

Combine or Merge

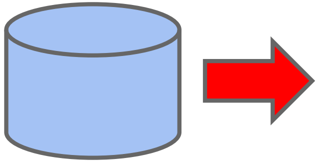
Merge records from multiple sets into a new data sets

# Methods of Processing

- There are broadly two ways we can process data
  - Serially (one-at-a-time)
  - Function
- Serial processing is perhaps easier to reason about
  - Analogous to procedural programming
- Some approaches scale more readily
  - Declarative
  - Functional

# Serialized Processing (One-at-a-Time)

**While *r* != EOF:**



**1) I/O**
read one record, *r*

**2) Filter**
if *f(r)*, emit *r*

**3) Mutate**
emit *t(r)*

**4) Aggregate**
update *a* = *agg(r)*

Basic serial processing
- Read and process a record
- Read the next record

I/O is **very** expensive
- Maximize the number of records per I/O

How do we scale this **out?**

# Serialized != Good Data Processing

| Work Done | Time Taken | Cost increae |
|---|---|---|
| 1 CPU instruction | <= 1ns | |
| Read 1MB from Memory | 0.25ms | 250k ns |
| Read 1MB from SSD | 1ms | 4x memory |
| Read 1MB From Disk | 20ms | 80x memory |
| Disk seek | 10ms | 40x memory |

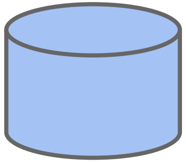
Disk I/O costs dwarf memory costs

Disk seeks may be required for any disk read

For large scale data processing **amortize** the cost of disk I/O

# Functionally-Driven Processing

**While *r* != EOF:**



**1) I/O**
read a large batch ***R***

**2) Filter**
emit
$\boldsymbol{R'} = \{r' | \forall r \in \boldsymbol{R},\ f(r) \wedge true\}$

**3) Mutate**
emit
$\boldsymbol{T} = \{t(r) | \forall r \in \boldsymbol{R'}\}$

**4) Aggregate**
emit $a = agg(\boldsymbol{T})$

Functional processing
- Logically consistent with serial
- Read **batches** of records
- Apply **functions** to batches

Allows us to distribute operations logically
- Between threads
- Between computers

# Common Operators for Functional Collection Processing

filter

Filter the collection such that only

map

Apply function *f(x)* to all members *x* in the collection

flatMap

Flatten the collection and apply *f(x)* to all resulting members

reduce

Apply aggregating function *f(x,y)* to all members in the collection

fold

Apply an aggregating function *f(x,y)* to all members

Store the result in a variable *g*

# Processing in Stages

- Complex processing can be broken down into smaller stages
- We can build a DAG which serves as a plan for processing
- DAGs allow
  - Optimization
  - Fault Tolerance
- DAGs can introduce new complexities
  - Merging datasets

# Map-Reduce

## "Classic" Word Counting

**map(***word => (word, 1)***)**

Prime minister Alexis Tsipras of Greece has resigned from his position as of Thursday, August 15.

(prime, 1) (Tsipras, 1) (of, 1),
(minister, 1), (Greece, 1), (has, 1)
(Alexis, 1), (resigned, 1), (from, 1), (as, 1),
(of, 1), (Thursday, 1), (August, 1), (20, 1)

(prime, 1) (Tsipras, 1) (of, 1),
(minister, 1), (Greece, 1), (has, 1)
(Alexis, 1), (resigned, 1), (from, 1), (as, 1),
(of, 1), (Thursday, 1), (August, 1), (20, 1)

**reduce(**$x, y \Rightarrow (x_1, x_2 + y_2)$**)**

(of, 2) (prime, 1) (minister, 1), ...

# Map-Reduce vs. Hadoop MapReduce

- Logical concept
  - Chaining of 2 functions
- Exists in many languages
  - Python
  - Ruby
  - Scala

- Data Processing Framework
  - Distributed system
  - Designed for massive data processing
- A MapReduce program does exactly
  - Read Data
  - Map
  - Reduce
  - Write Data

# Overcoming MapReduce Limitations

- A MapReduce program can only Map & Reduce between I/Os
- How do we build anything complex?
- Build a DAG with I/Os in-between
  - Read → Map → Reduce → Write temp result
  - Read temp → Map → Reduce → Write result
- Allows us to do
  - Iterative operations
  - Merge-and-Aggregate
  - etc.

# Chaining Map-Reduce Pairs

**Map #1:**

**Reduce #1:**

**Map #2:**

**Reduce #2:**

# What problems does this have?

**Map #1:**

**Reduce #1:**

**Checkpoint**

**Map #2:**

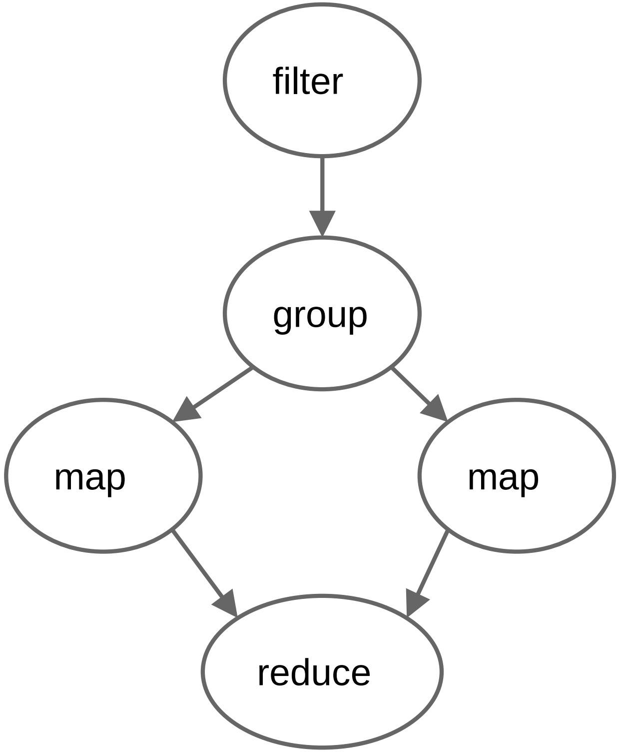
**Reduce #2:**

Are we doing as much work as possible in each step?

Forced checkpointing:
Must write and re-read

Good for fault-tolerance
Bad for performance

# Extending to Generic DAGs




A **directed acyclic graph**

- Enables dataflow programming

DAGs

- form a logical plan for data processing
- can be optimized
- can be distributed (shuffling)

Examples

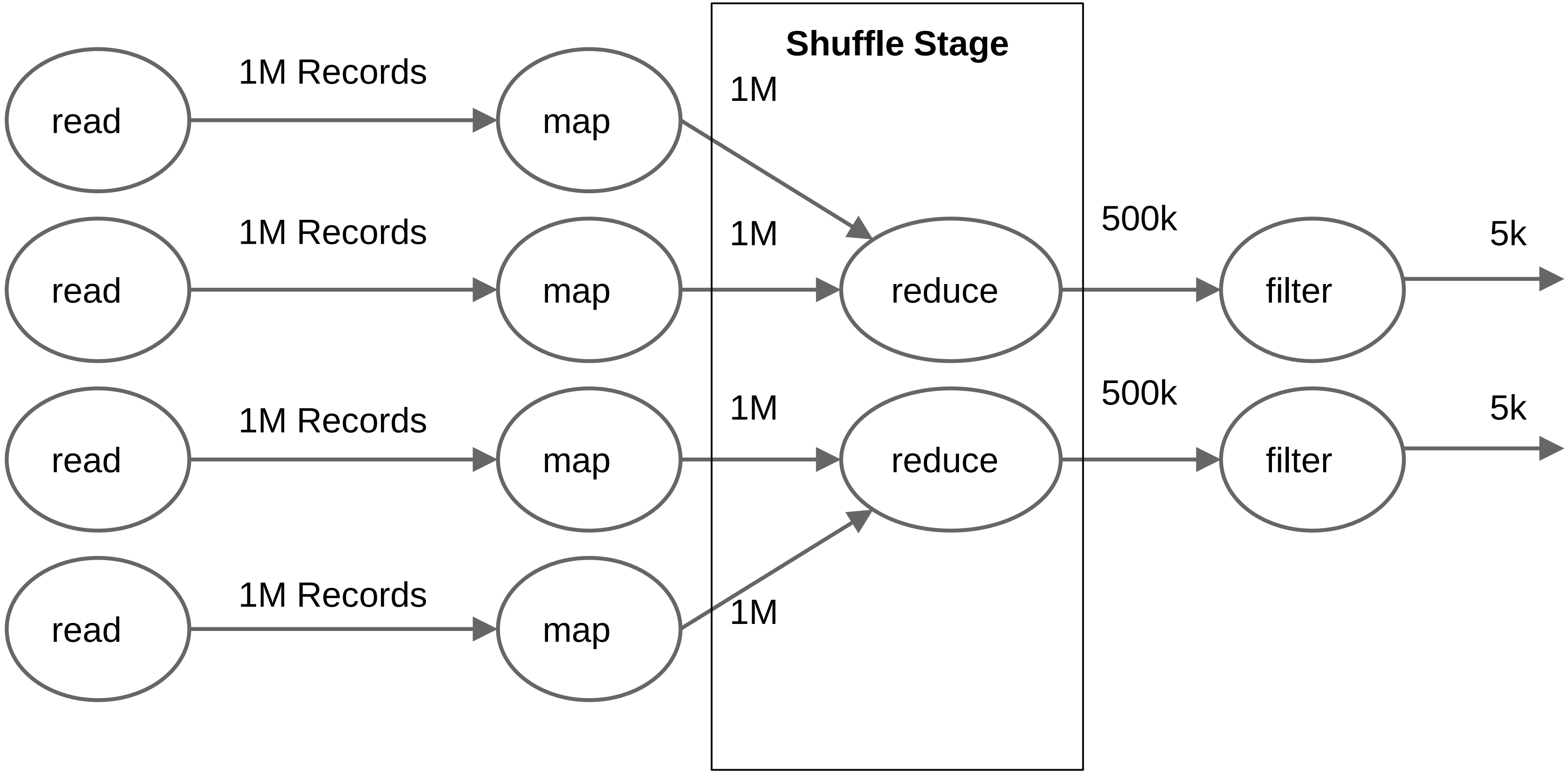
- Apache Pig, Tez, Apache Spark, Parallel RDBMS

# How Do We Optimize DAGs?

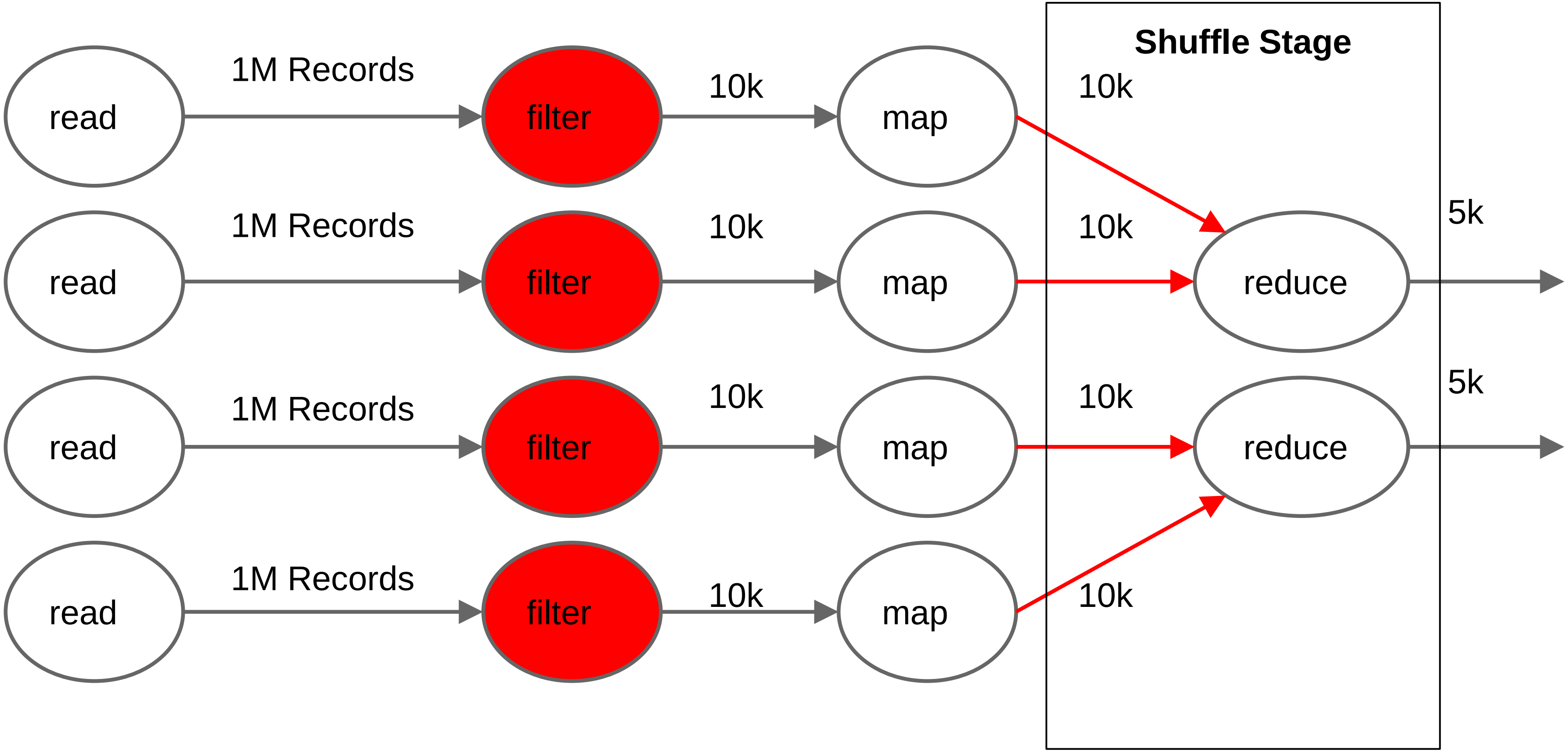
- Minimize materialization
  - Read the minimum amount of data necessary
  - Filter data as early as possible
  - Write data as late possible

- Minimize data shipping, particularly in distributed systems
  - Passing data between processes is expensive
    - Shared memory
    - Network transfer
  - Minimize the number and size of shuffles

# Unoptimized DAG Example



# Optimized DAG Example

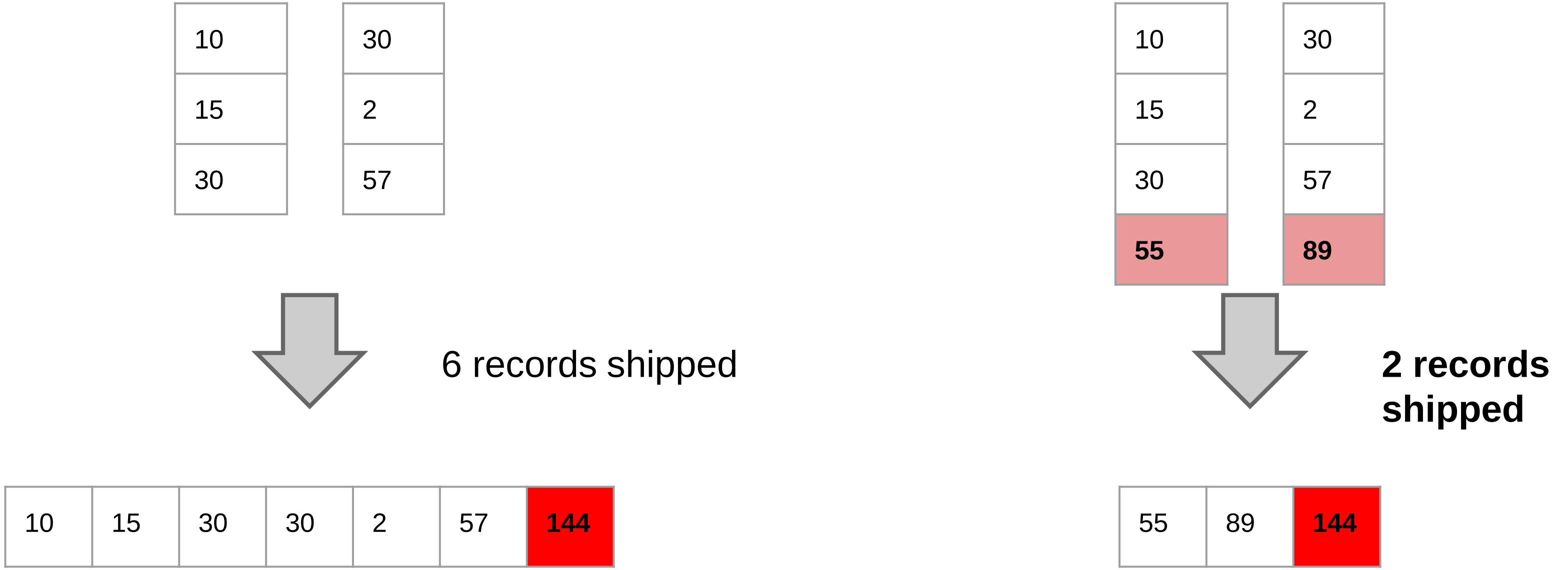


# Optimizing Shuffle Stages

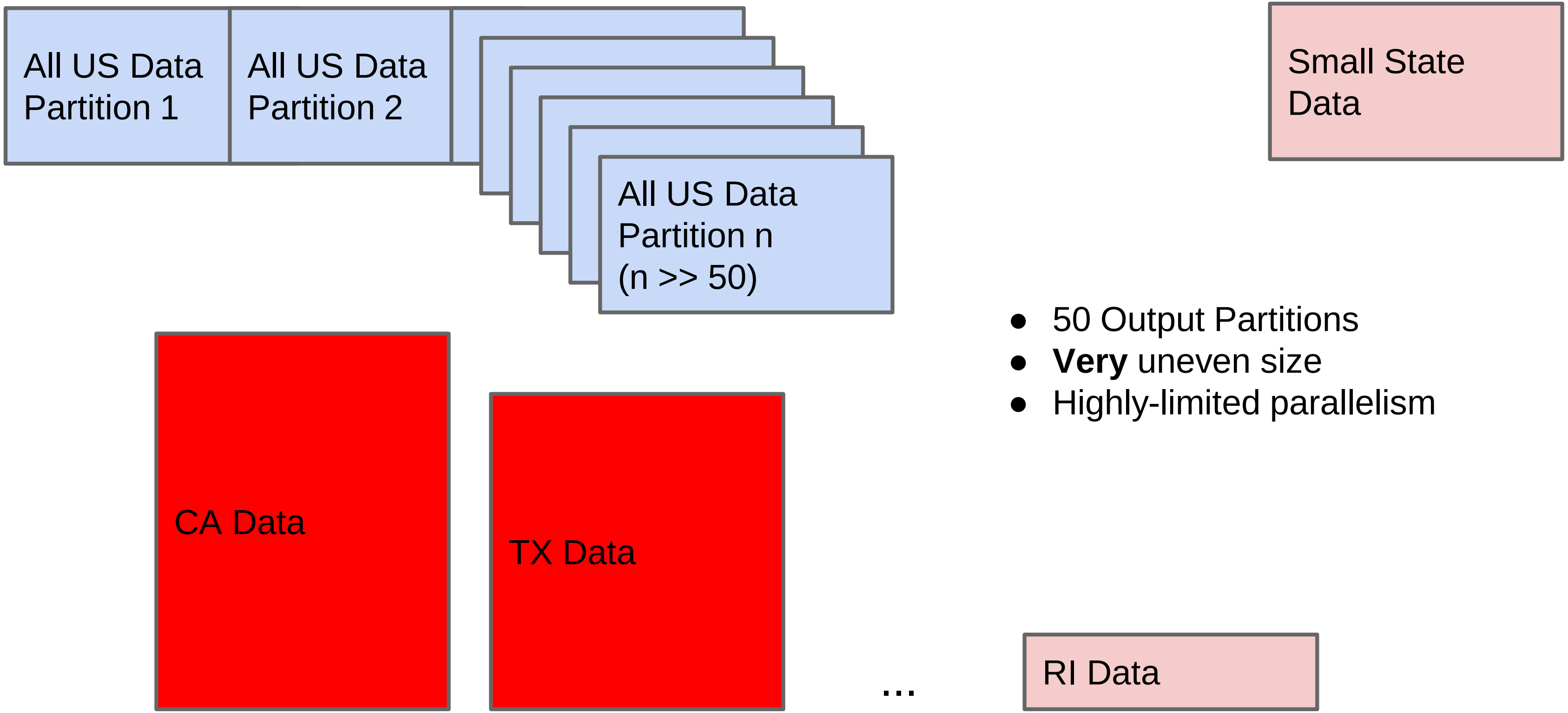
- Shuffle the smallest amount of data possible
  - Filter data early in the graph
  - Utilize *combiners* when possible (partial aggregation)
  - Choose specific implementations of operators
    - Broadcast Joins for merged small data with large
    - Hash Joins for merging two sets of data local to the process

# Partial Aggregation

Pre-aggregate results → reduced shuffle overhead

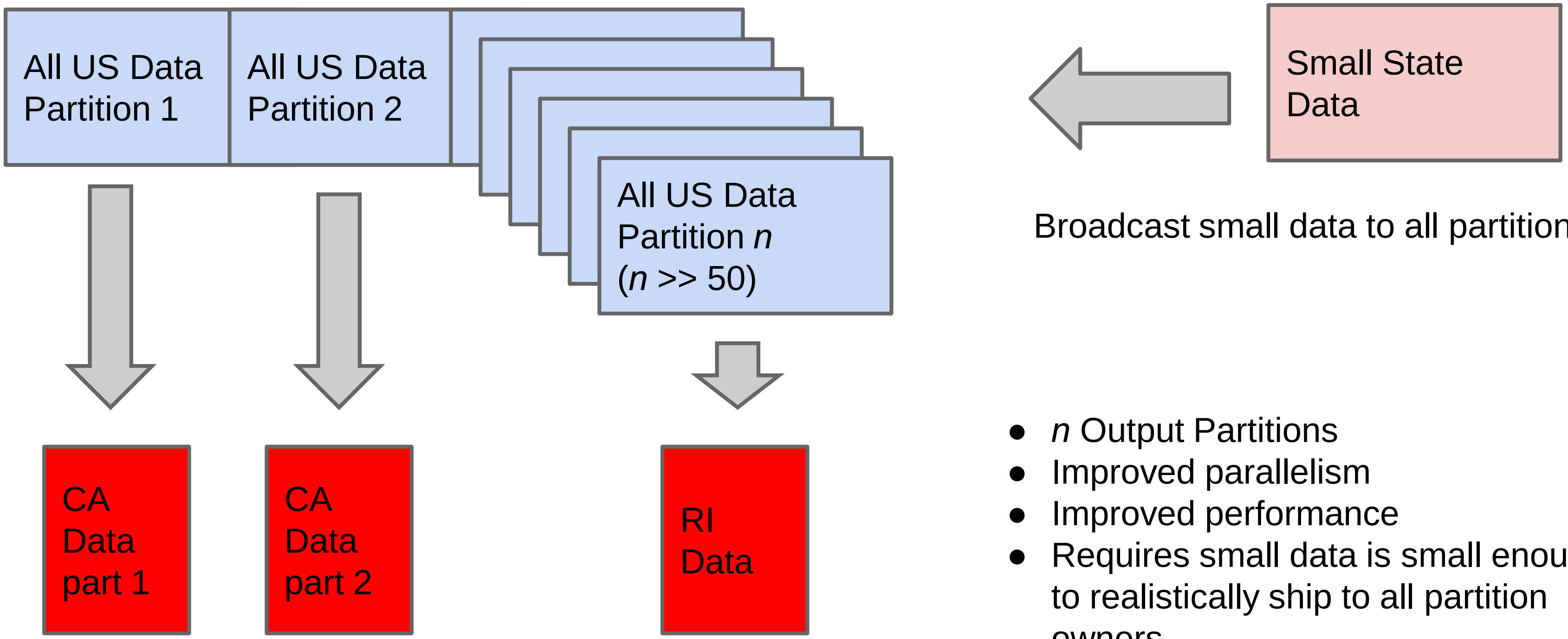


# Merging and Data Skew




- 50 Output Partitions
- **Very** uneven size
- Highly-limited parallelism

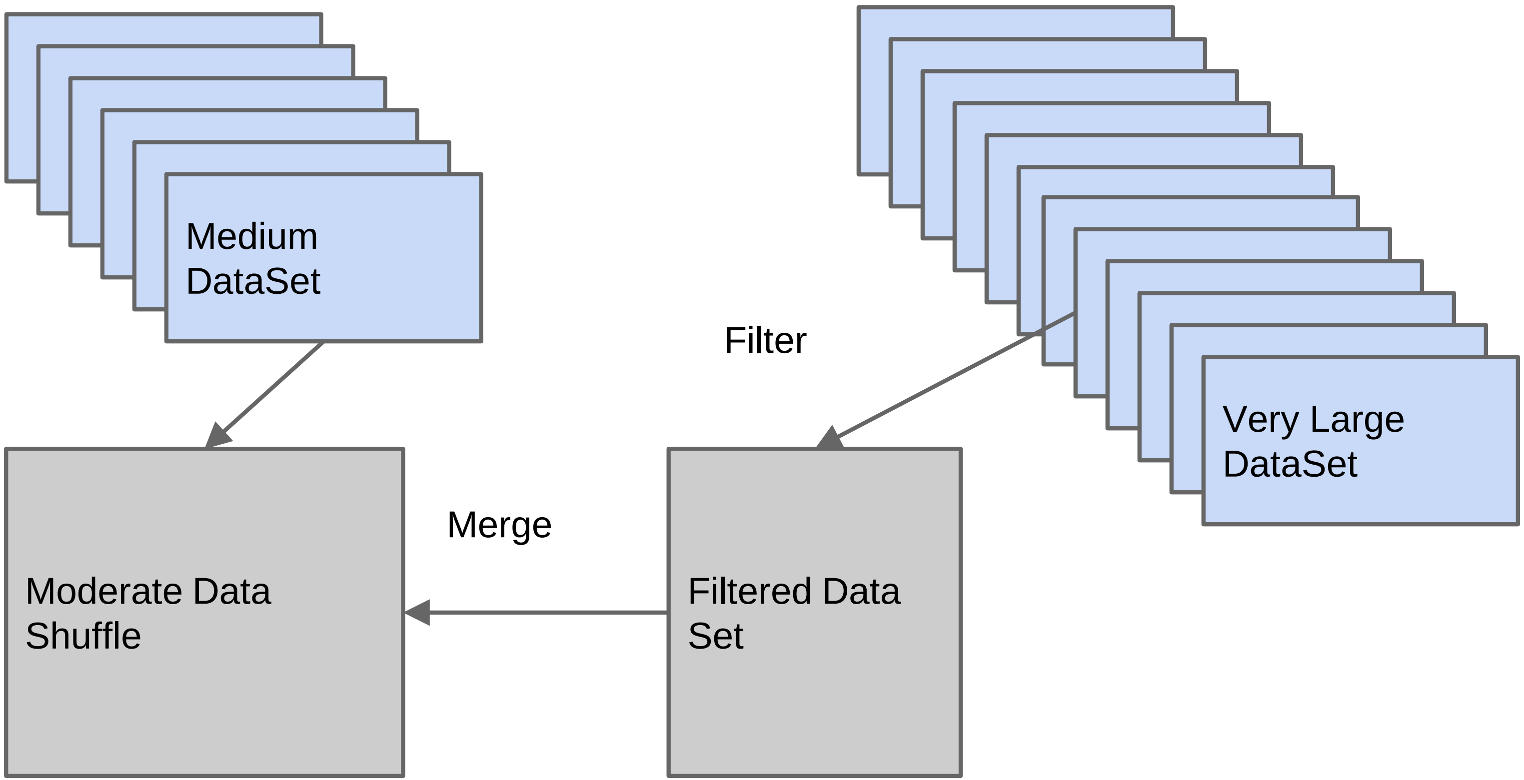
# Skew Handling: Broadcast Join




Broadcast small data to all partition owners

- *n* Output Partitions
- Improved parallelism
- Improved performance
- Requires small data is small enough to realistically ship to all partition owners

# Skew Handling: Pre-Filtering



# Summary

- Data processing can be broken down into fundamental operations
  - Filter, mutate, aggregation, merge
- Disk and Network I/O dominate processing costs
- Functional approaches allow
  - Distribution of operations
  - Amortization of I/O
- Acyclic Graphs of fundamental operations allow
  - Process planning
  - Optimization
- Aggregation and Merge require extra care