

Map Reduce

History

- A *simple* paradigm that popped up several times as paradigm
- Observed by google as a software pattern:
 - Data gets filtered locally and filtered data is then reassembled elsewhere
 - Software pattern: Many engineers are re-engineering the same steps
- Map-reduce:
 - Engineer the common steps efficiently
 - Individual problems only need to be engineered for what makes them different

History

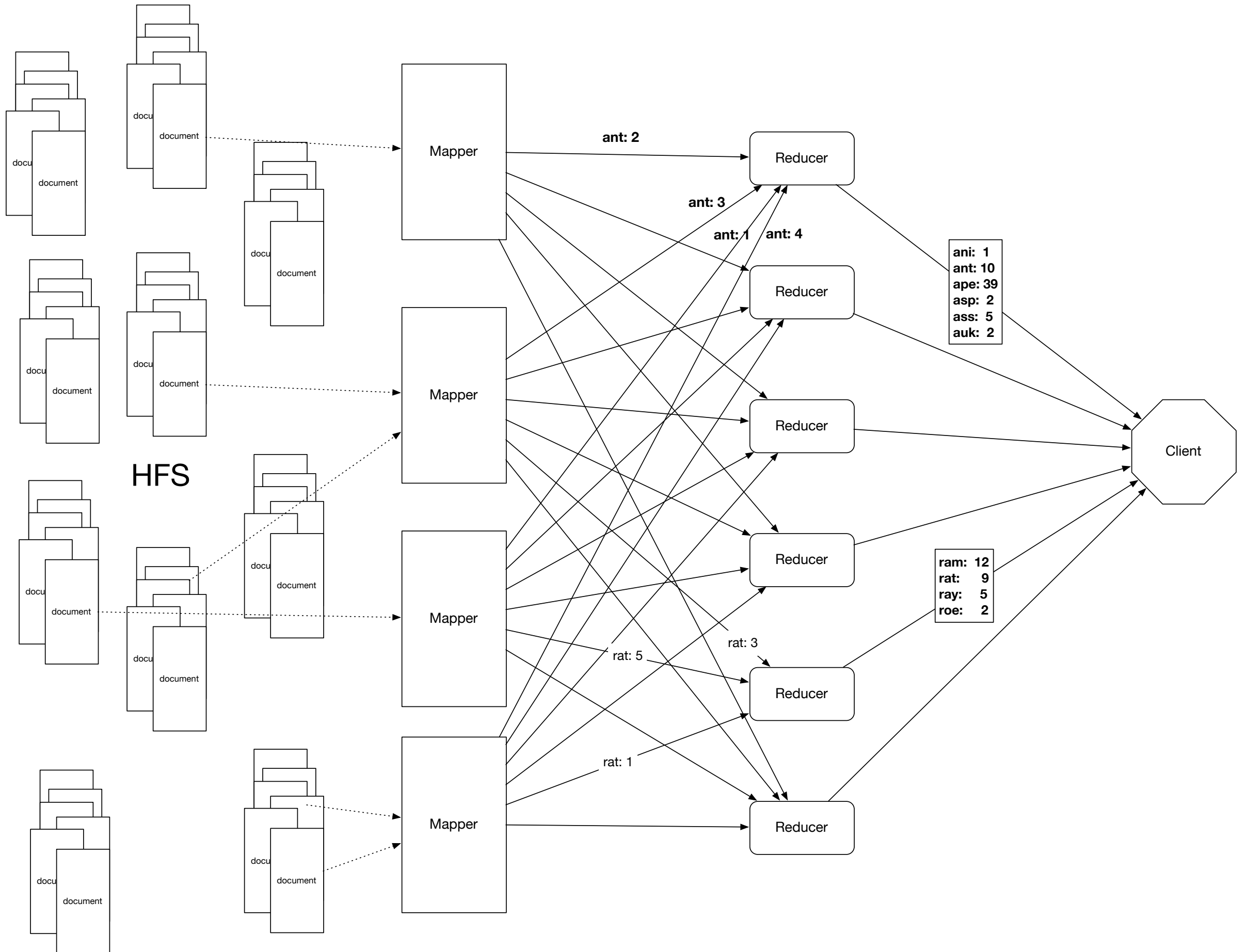
- Open source project (in part sponsored by Yahoo!)
 - Java-based Hadoop
 - Eventually a first tier Apache Foundation project
- Other projects at higher level: Pig, Hive, HBase, Mahout, Zookeeper
 - Use Hadoop as foundation
 - Hadoop is becoming a distributed OS

Map Reduce Paradigm

- Input: Large amount of data spread over many different nodes
- Output: A single file of results
- Two important phases:
 - **Mapper:** Records are processed into key-value pairs. Mapper sends key-value pairs to reducers
 - **Reducer:** Create final answer from mapper

Simple Example

- Hadoop Word Count
 - Given different documents on many different sites
 - Mapper:
 - Extract words from record
 - Combines words and generates key-value pairs of type word: key
 - Sends to the reducers based on hash of key
 - Reducer:
 - Receives key-value pairs
 - Adds values for each key
 - Sends accumulated results to aggregator - client



Map-reduce paradigm in detail

- The simple mapper-reducer paradigm can be expanded into several, typical components

Map Reduce in Detail

- **Mapper:**
 - **Record Reader**
 - Parses the data into records
 - Example: Stackoverflow comments.
 - `<row Id="5" PostId="5" Score="2" Text="Programming in Portland, cooking in Chippewa ; it makes sense that these would be unlocalized. But does bicycling.se need to follow only that path? I agree that route a to b in city x is not a good use of this site; but general resources would be." CreationDate="2010-08-25T21:21:03.233" UserId="21" />`
 - Record reader extract the "Text=" string
 - Passes record into a key-value format to rest of mapper

Map Reduce in Detail

- **Mapper**

- `map`

- Produces “intermediate” key-value pairs from the record

- Example:

- "Programming in Portland, cooking in Chippewa ; it makes sense that these would be unlocalized. But does bicycling.se need to follow only that path? I agree that route a to b in city x is not a good use of this site; but general resources would be."
 - Map produces: `<programming: 1> <in: 1>`
`<Portland: 1> <cooking: 1> <in: 1> ...`

Map Reduce in Detail

- **Mapper**

- Combiner — a local reducer

- Takes key-value pairs and processes them

- Example:

- Map produces: <programming: 1> <in: 1>
<Portland: 1> <cooking: 1> <in: 1> ...

- Combiner combines words: <programming: 1>
<in: 4> <Portland: 3> ...

Map Reduce in Detail

- Combiners allow us to reduce network traffic
 - By compacting the same information

Map Reduce in Detail

- **Mapper**
 - Partitioner
 - Partitioner creates shards of the key-value pairs produced
 - One for each reducer
 - Often uses a hash function or a range
 - Example:
 - $\text{md5}(\text{key}) \bmod (\#\text{reducers})$

Map Reduce in Detail

- **Reducer**
 - Shuffle and Sort
 - **Part of the map-reduce framework**
 - Incoming key-value pairs are sorted by key into one large data list
 - Groups keys together for easy agglomeration
 - Programmer can specify the comparator, but nothing else

Map Reduce in Detail

- **Reducer**
 - `reduce`
 - Written by programmer
 - Works on each key group
 - Data can be combined, filtered, aggregated
 - Output is prepared

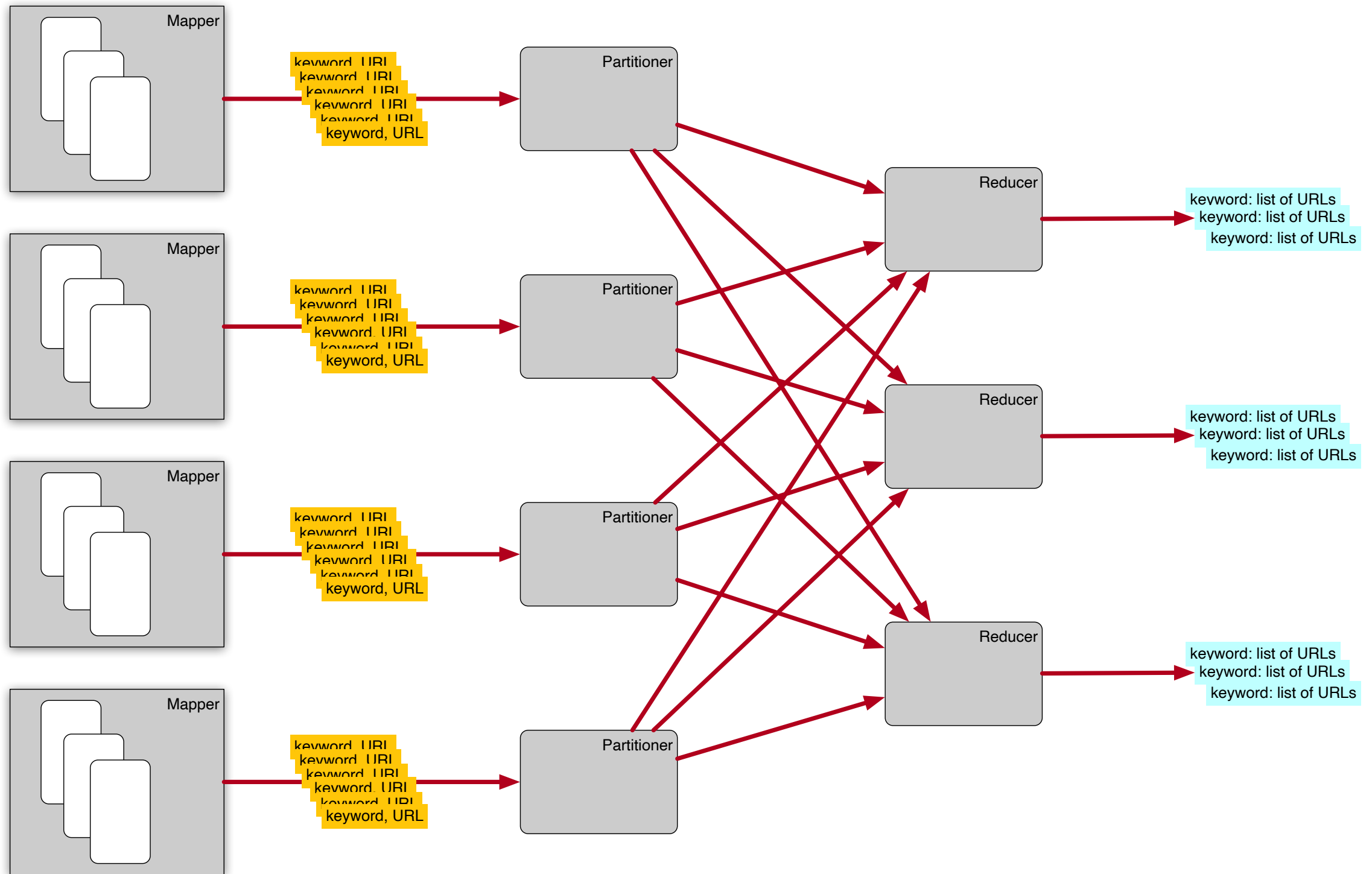
Map Reduce in Detail

- **Reducer**
 - Output format
 - Formats final key-value pair

Inverted Index Pattern

- Task: Create an inverted index for a large set of documents
- Desired output:
 - A database that gives all documents (as URL) where a certain word appears.

Inverted Index Pattern



Inverted Index Pattern

- Mapper Code:
 - Input: A wikipedia article
 - Procedure:
 - Create a bag of words, stem words, decide whether words are proper names or non-frequent words
 - Output: Create a list of key-value pairs:
 - Keys: important words or proper names
 - Value: URL of article

Inverted Index Pattern

- Combiner Optimization:
 - Weed out duplicate results

Inverted Index Pattern

- Reducer:
 - Input: A list of **ordered** key value pairs
 - <word, URL>
 - Output: Agglomerate to:
 - <word, list of URLs>

Shuffle and Sort

- Each reduce job gets sorted input
- System part that sorts input and transfers to outputs of mappers to reducers is *shuffle*

Shuffle and Sort

- Map side
 - Output is not simply written to disk
 - For better performance:
 - Output is put into buffers
 - Buffers are partitioned and sorted when flushed to disk as *spill files*
 - When mapper finishes:
 - Spill files are combined
- Output is made available to reducers using HDFS

Shuffle and Sort

- Reduce side:
 - Reducers start copying mapper output as soon as they are available (*copy phase*)
 - If mapper outputs at reducer reach critical size, they are placed into spill files on disk
 - Spill files are sorted and combined in batches — typically 10 spill files
 - Final combination feeds directly to reducer

Map Reduce Patterns

- Summarizations
 - Input: A large data set that can be grouped according to various criteria
 - Output: A numerical summary
 - Example:
 - Calculate minimum, maximum, total of certain fields in documents in xml format ordered by user-id

Summarization

- Example:
 - Given a database in xml-document format

```
<row Id="193" PostTypeId="1" AcceptedAnswerId="194"
CreationDate="2010-10-23T20:08:39.740" Score="3" ViewCount="30"
Body="<p>Do you lose one point of reputation when you
down vote community wiki? Meta? </p>&#xA;&#xA;<p>I
know that you do for "regular questions". </
p>&#xA;" OwnerUserId="134"
LastActivityDate="2010-10-24T05:41:48.760" Title="Do you lose
one point of reputation when you down vote community wiki?
Meta?" Tags="<discussion>" AnswerCount="1"
CommentCount="0" />
```

- Determine the earliest, latest, and number of posts for each user

Summarization

- Mapper:
 - Step 1: Preprocess document by extracting the user ID and the date of the post
 - Step 2: map:
 - User ID becomes the key.
 - Value stores the date twice in Java-date format and adds a long value of 1

“134”: (2010-10-23T20:08:39.740, 2010-10-23T20:08:39.740, 1)

Summarization

- Mapper:
 - Step 3: Combiner
 - Take intermediate User-ID — value pairs
 - Combine the value pairs
 - Combination of two values:
 - first item is minimum of the dates
 - second item is maximum of the dates
 - third item is sum of third items

Summarization

- The map reduce framework is given the number of reducers
 - Autonomously maps combiner results to reducers
 - Each reducer gets key-value parts for a range of user-IDs grouped by user-ID

Summarization

- Reducer:
 - Passes through each group combining key-value pairs
 - End-result:
 - Key-value pair with key = user-id
 - Value is a triple with
 - minimum posting date
 - maximum posting date
 - number of posts

Summarization

- Reducer:
 - Each summary key— value pair is sent to client

Summarization

- Example (cont.)

Mapper 1

UserID 12345	01.02.2010	01.02.2010	1
UserID 12345	02.02.2010	02.02.2010	1
UserID 12345	04.02.2010	04.02.2010	1
UserID 98765	12.02.2010	12.02.2010	1
UserID 98765	02.02.2010	02.02.2010	1
UserID 98765	05.02.2010	05.02.2010	1
UserID 56565	02.02.2010	02.02.2010	1
UserID 56565	03.02.2010	03.02.2010	1

Combiner

UserID 12345	01.02.2010	04.02.2010	3
UserID 98765	02.02.2010	12.02.2010	3
UserID 56565	02.02.2010	03.02.2010	2

Mapper 2

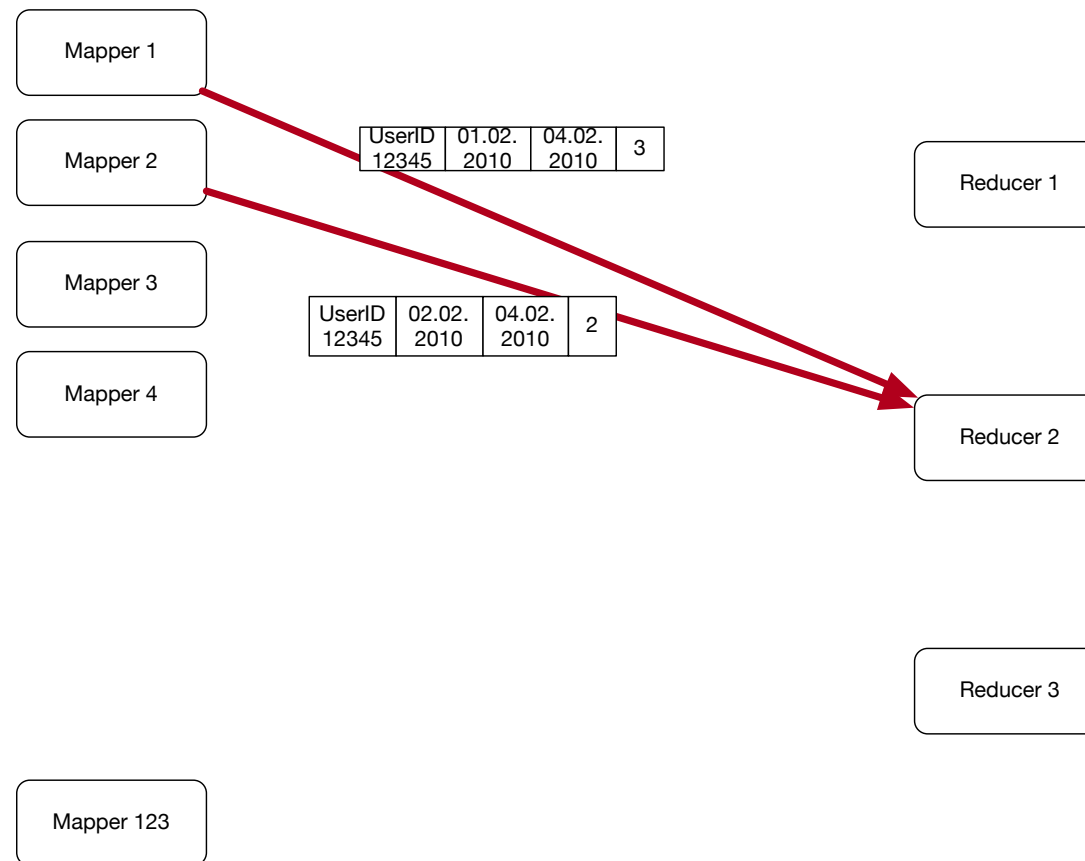
UserID 12345	02.02.2010	02.02.2010	1
UserID 12345	04.02.2010	04.02.2010	1
UserID 77444	12.02.2010	12.02.2010	1
UserID 77444	02.02.2010	02.02.2010	1
UserID 98765	05.02.2010	05.02.2010	1

Combiner

UserID 12345	02.02.2010	04.02.2010	2
UserID 77444	02.02.2010	12.02.2010	2
UserID 98765	05.02.2010	05.02.2010	1

Summarization

- Example (cont.) Automatic Shuffle and Sort
 - Records with the same key are sent to the same reducer



Summarization

- Example (cont.)
 - Reducer receives records already ordered by user-ID
 - Combines records with same key

UserID 12345	01.02.2010	04.02.2010	3
UserID 12345	02.02.2010	04.02.2010	2
UserID 12345	26.03.2010	30.04.2010	5
UserID 12345	19.01.2010	01.04.2010	3
UserID 16542	02.02.2010	04.02.2010	6
UserID 16542	26.03.2010	29.05.2010	5
UserID 16542	19.01.2010	19.01.2010	1



UserID 12345	01.02.2010	30.02.2010	13
UserID 16542	19.01.2010	29.05.2010	12

Summarization

- In (pseudo-)pig:
 - Load data

```
posts = LOAD '/stackexchange/posts.tsv.gz'  
USING PigStorage('\t') AS (  
post_id : long,  
user_id : int,  
text : chararray,  
...  
post : date  
)
```

Summarization

- In (pseudo-)pig:

- Group by user-id

```
post_group = GROUP posts BY user_id;
```

- Obtain min, max, count:

```
result = FOREACH post_group GENERATE group,  
MIN(posts.date), MAX(posts.date),  
COUNT_STAR(post_group)
```

Summarization

- In (pseudo-)pig:
 - Load data

```
orders = LOAD '/stackexchange/posts.tsv.gz'  
USING PigStorage('\t') AS (  
post_id : long,  
user_id : int,  
text : chararray,  
...  
post : date  
)
```

Summarization

- Your turn:
 - Calculate the average score per user
 - The score is kept in the “score”-field

Summarization

- Solution:
 - Need to aggregate sum of score and number of posts
 - Mapper: for each user-id, create a record with score

```
userid: score, 1
```
 - Combiner adds scores and counts

```
userid: sum_score, count
```
 - Reducer combines as well
 - Generates output key-value pair and sends it to the user
 - ```
userid: sum_score/count
```

# Summarization

- Finding the median of a numerical variable
  - Mapper aggregates all values in a list
  - Reducer aggregates all values in a list
  - Reducer then determines median of the list
- Can easily run into memory problems

# Summarization

- Median calculation:
  - Can compress lists by using counts
    - `2, 3, 3, 3, 2, 4, 5, 2, 1, 2` becomes  
`(1, 1), (2, 4), (3, 3), (4, 1) (5, 1)`
  - Combiner creates compressed lists
  - Reducer code directly calculates median
    - An instance where combiner and reducer use different code



# Summarization

- Standard Deviation
  - Square-root of variance
  - Variance — Average square deviation from average

- $$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$$

- Leads to a two pass solution, calculate average first

# Summarization

- Standard Deviation
  - Numerically dangerous one-path solution

- $$\begin{aligned}\sigma_x^2 &= \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2 \\ &= \frac{1}{N} \sum_{i=1}^N (x_i^2 - 2\bar{x}x_i + \bar{x}^2) \\ &= \frac{1}{N} \sum_{i=1}^N x_i^2 - 2\bar{x} \frac{1}{N} \sum_{i=1}^N x_i + \bar{x}^2 \\ &= \frac{1}{N} \sum_{i=1}^N x_i^2 - 2\bar{x}^2 + \bar{x}^2 = \frac{1}{N} \sum_{i=1}^N x_i^2 + \bar{x}^2\end{aligned}$$

# Summarization

- Chan's adaptation of Welford's online algorithm
  - Using the counts of elements, can calculate the variance in parallel from any number of partitions

```
def parallel_variance(avg_a, count_a, var_a, avg_b, count_b, var_b):
 delta = avg_b - avg_a
 m_a = var_a * (count_a - 1)
 m_b = var_b * (count_b - 1)
 M2 = m_a + m_b + delta ** 2 * count_a * count_b / (count_a + count_b)
 return M2 / (count_a + count_b - 1)
```

- Unfortunately, can still be numerically instable

# Summarization

- Standard Deviation:
  - Schubert & Gertz: Numerically Stable Parallel Computation of (Co)-Variance
  - SSDBM '18 Proceedings of the 30th International Conference on Scientific and Statistical Database Management

# Summarization

- Inverted Index
  - Analyze each comment in StackOverflow to find hyperlinks to Wikipedia
  - Create an index of wikipedia pages pointing to StackOverflow comments that link to them

# Summarization

- Inverted Index is a group-by problem solved almost entirely in the map-reduce framework

# Summarization / Inverted Index

- **Mapper**
  - Parser:
    - Processes posts
    - Checks for right type of post, extracts a list of wikipedia urls (or Null if there are none)
    - Outputs key-value pairs :
      - Keys: wikipedia url
      - Value: row-ID of post
  - Optional combiner:
    - Aggregates values for a wikipedia url in a single list

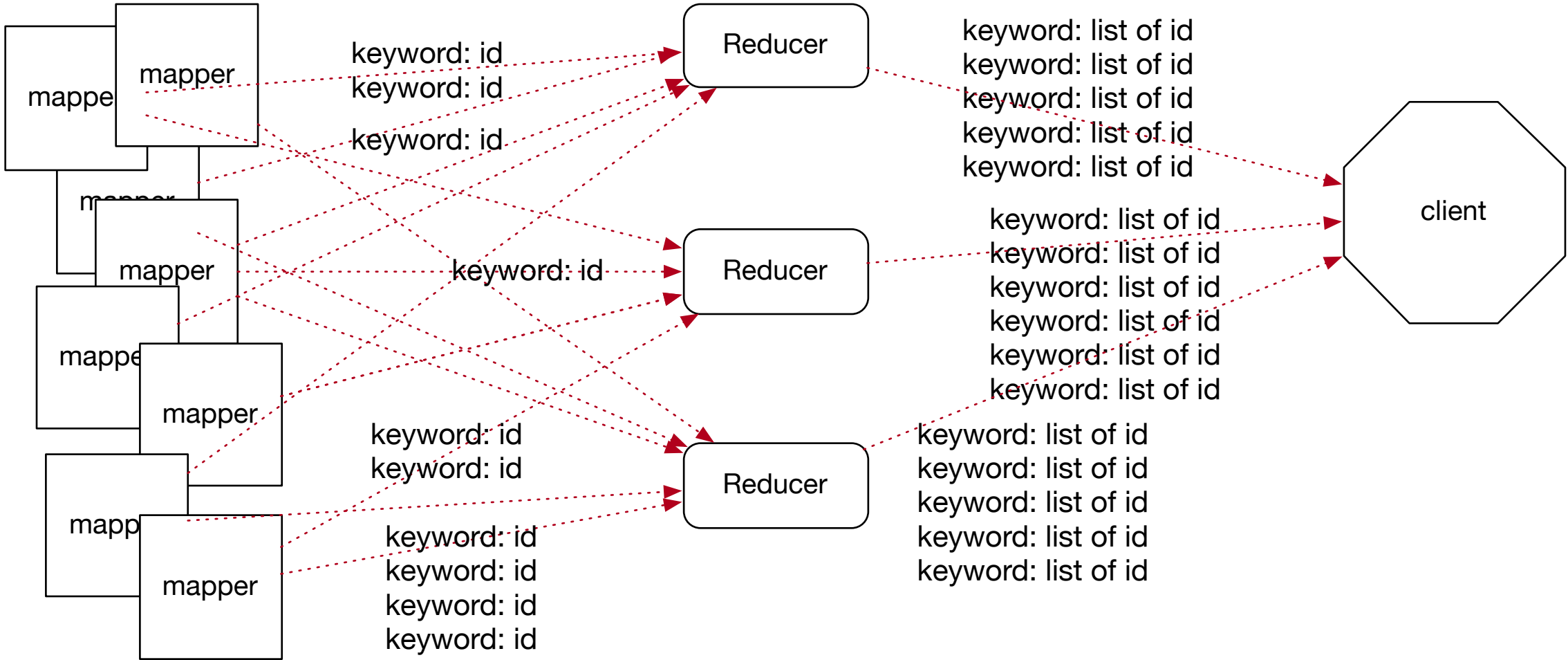
# Summarization / Inverted Index

- **Reducer**
  - Aggregates values belonging to the same key in a list



# Summarization / Inverted Index

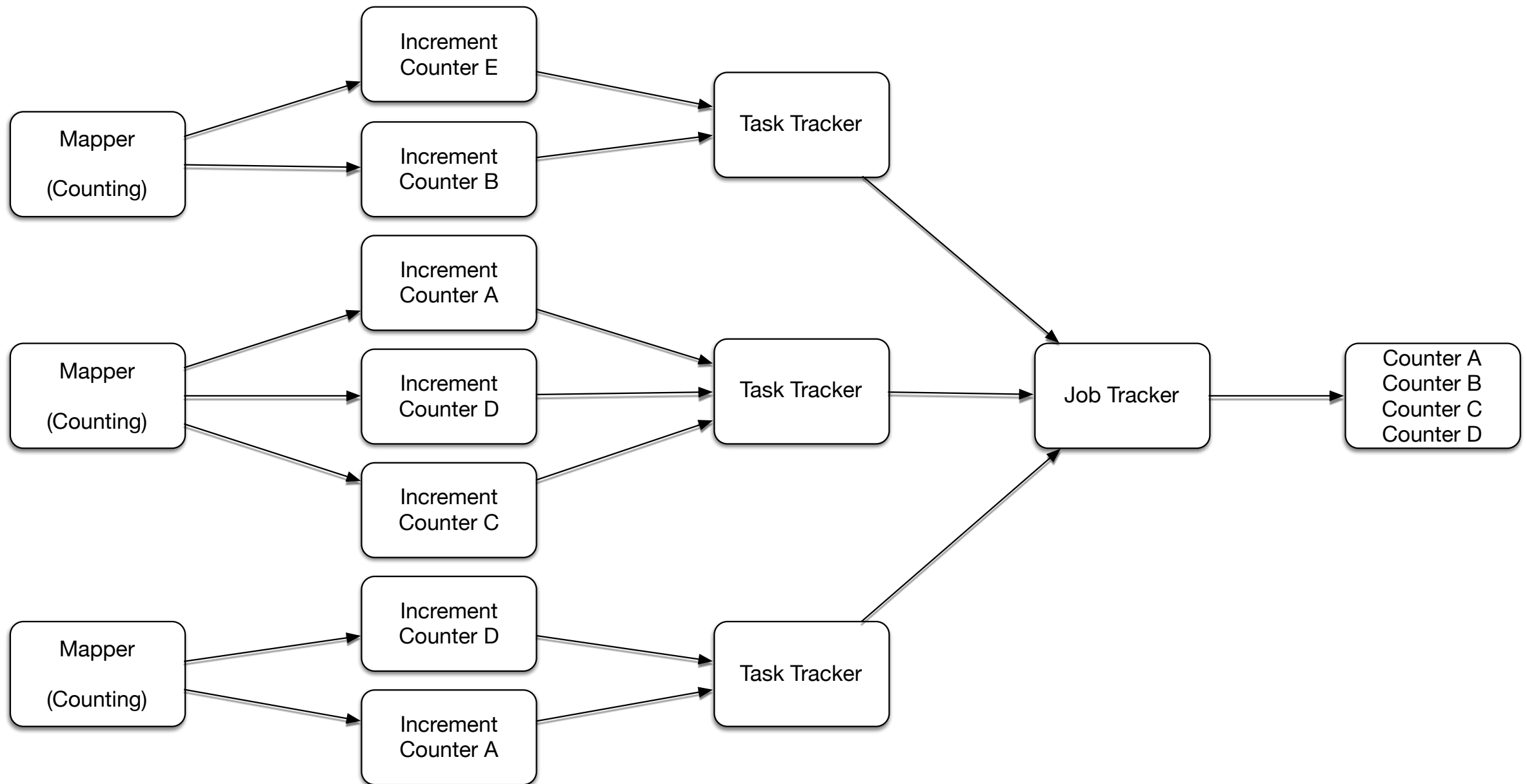
- Generic Inverted Index diagram



# Counter Pattern

- Used to gather stats on an Hadoop job
  - Create various counters (but not too many)
  - Counters work exclusively in the Map-Reduce Paradigm

# Counter Pattern



# Counter Pattern

- Mapper processes each input
  - Increments counter for each record
- Counters are aggregated by Task Trackers
- Task Trackers report counts to Job Tracker
- Job Tracker aggregates counts (unless task tracker failed)

# Counter Pattern

```
public static class CountNrUsersByState extends Mapper<Object,
Text, NullWritable, NullWritable> {

public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
Map<String, String> parsed =
MRDPUtils.transformXmlToMap(value.toString())

String location = parsed.get("Location");
if (location !=null && !location.isEmpty()) {
 if (states.contains(state)) {
 context.getCounter(STATE_COUNTER_GROUP, state).increment(1);
 break;
 }
}
...
}
```

# Counter Pattern

- To get the counts, just

```
int code = job.waitForCompletion(true)? 0 : 1;
if (code == 0) {
 for (Counter counter : job.getCounters().getGroup(
 CountNumUsersByStateMapper.STATECOUNTERGROUP)) {
 System.out.println(counter.getDisplayName() +
 "\t" + counter.getValue());
 }
}
```

# Filtering Patterns

- Extract data from records without changing them
  - Sampling:
    - get a few random records
    - get records with very high or low values in a field

# Filtering Patterns

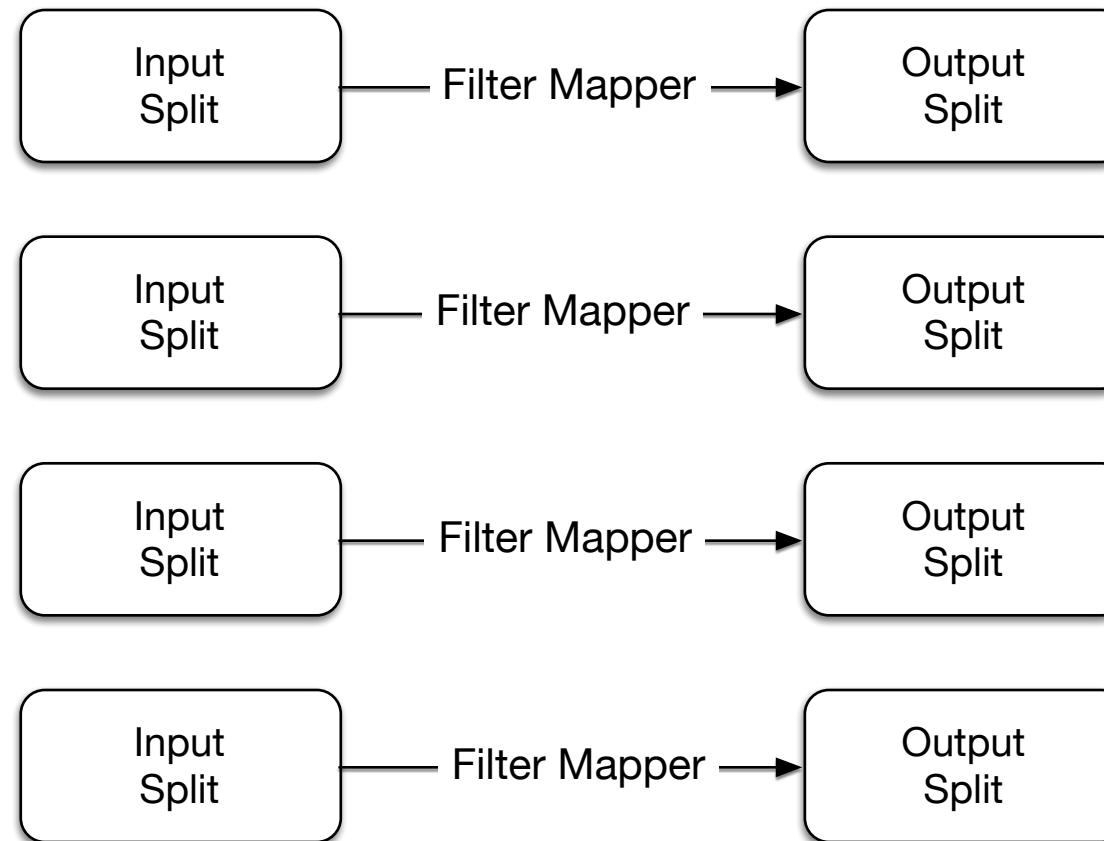
- Simple filtering:
  - User defined function or condition is a boolean
    - Decides whether record is to be kept or not
  - Very generic pattern

```
map(key, record) :
 if(user_condition(record) {
 emit key, value
```



# Filtering Patterns

- Simple Filtering



# Filtering Patterns

- Simple Filtering
  - There is no “reduce” operation because there is no aggregation
  - If output splits are saved, they can serve as new inputs

# Filtering Patterns

- Simple Filtering in Pig
  - Uses the **FILTER** keyword
  - `b = FILTER a BY value < 3`

# Filtering Patterns

- Because there are no reducers
  - Data never has to be transmitted
  - There is no sort phase and no reduce phase

# Filtering Patterns

- Filtering pattern:
  - Grep: filtering for a regular expression

# Filtering Patterns

- Getting a random sample
  - Simple random sampling (SRS)
    - Grab a random subset of data
  - Can get `filter_percentage` property:
    - `context.getConfiguration().get("filter_percentage")`
  - Mapper writes objects with a given probability
  - There neither combiner nor reducer

# Excuse: Bloom Filters

- Bloom filters (1970 Burton Howard Bloom)
  - Use to test membership in a set
  - Idea:
    - A data structure that can quickly decide whether an element does **not** belong to a large set
    - And probabilistically whether an element is present
      - With a low probability of error

# Excuse: Bloom Filters

- Idea: A bloom filter is a large bit array
  - (How large: Deduplication proposes sizes of several GB)
  - Uses a good hash function
  - For each element in the set:
    - Calculate  $h(ele,0)$   $h(ele,1)$   $h(ele,2)$
    - Change the resulting bits in the bit array

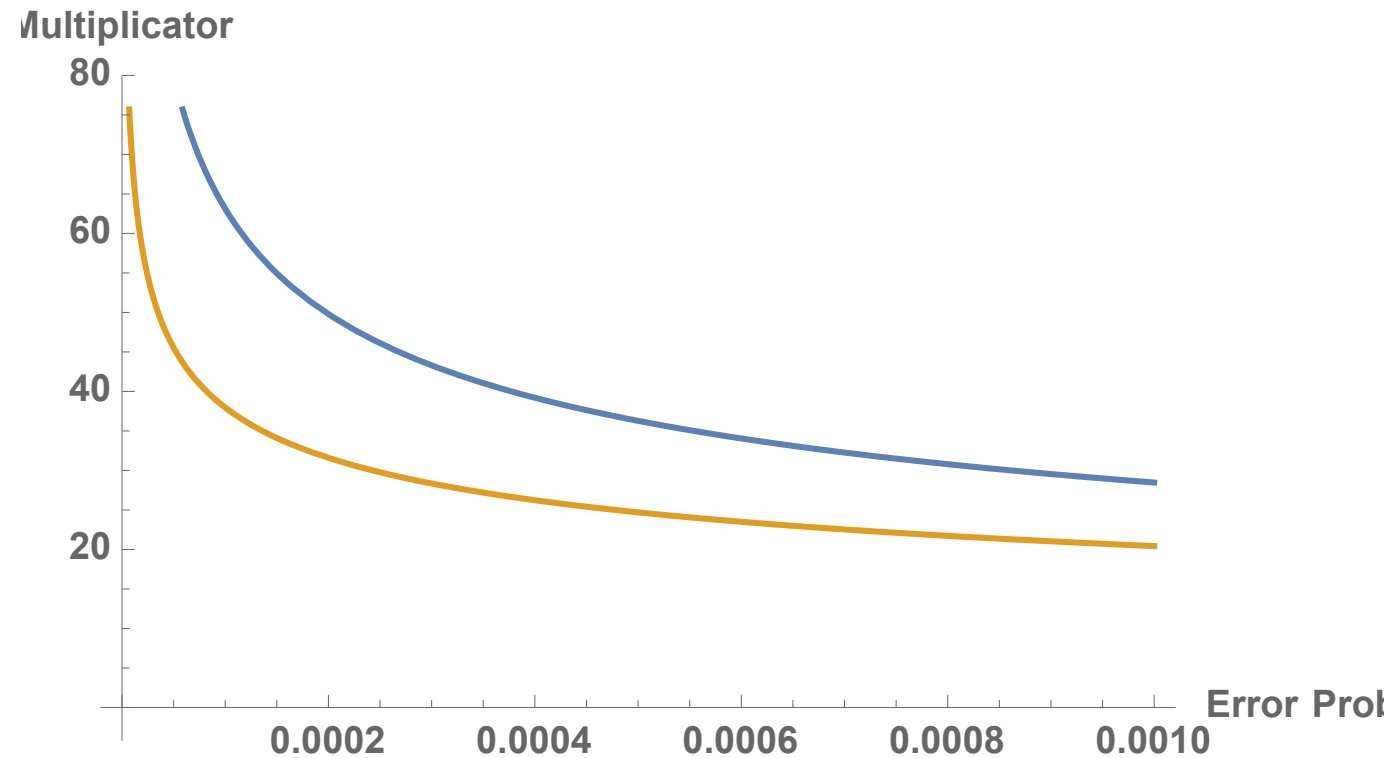


# Excuse: Bloom Filters

- To test for the presence of  $x$  in the set  $S$ 
  - Calculate  $h(x,0)$   $h(x,1)$   $h(x,2)$
  - Check whether the corresponding bits are set.
    - Bits are set, but  $x \notin S$ 
      - Then bits were set by other elements.
    - If  $|S| = n$  and there are  $N$  elements in the bit array, this happens with probability

$$\left(1 - \left(1 - \frac{1}{N}\right)^{3n}\right)^3 \approx \left(1 - \exp\left(\frac{-3n}{N}\right)\right)^3$$

# Excuse: Bloom Filters



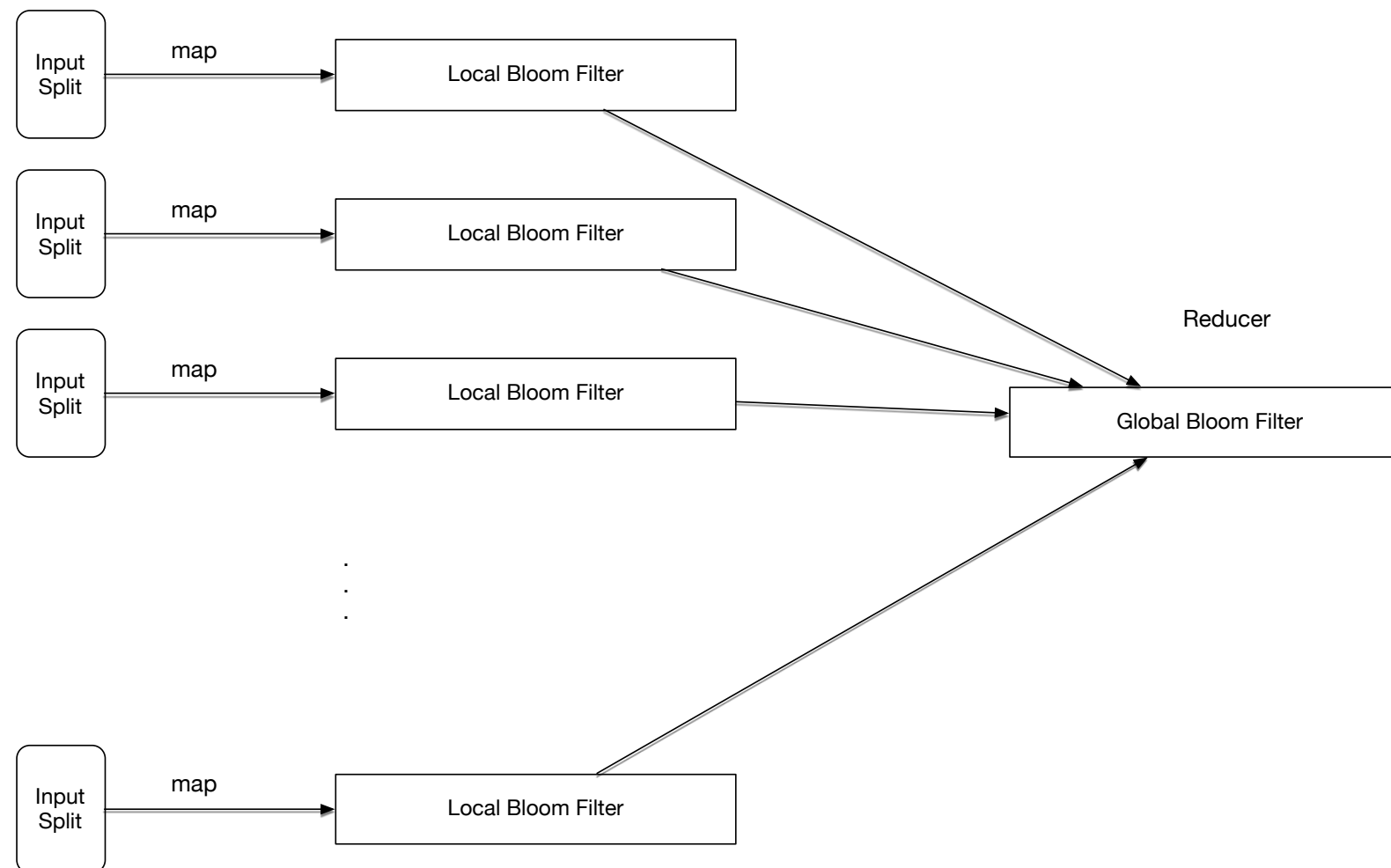
Number of bits = number of set elements times multiplier  
needed to achieve a certain error probability with 3 and 4 hashes

# Filtering Patterns

- Bloom Filtering pattern:
  - Need to accept a few false positives
- Creating Bloom Filter
- Using Bloom Filter

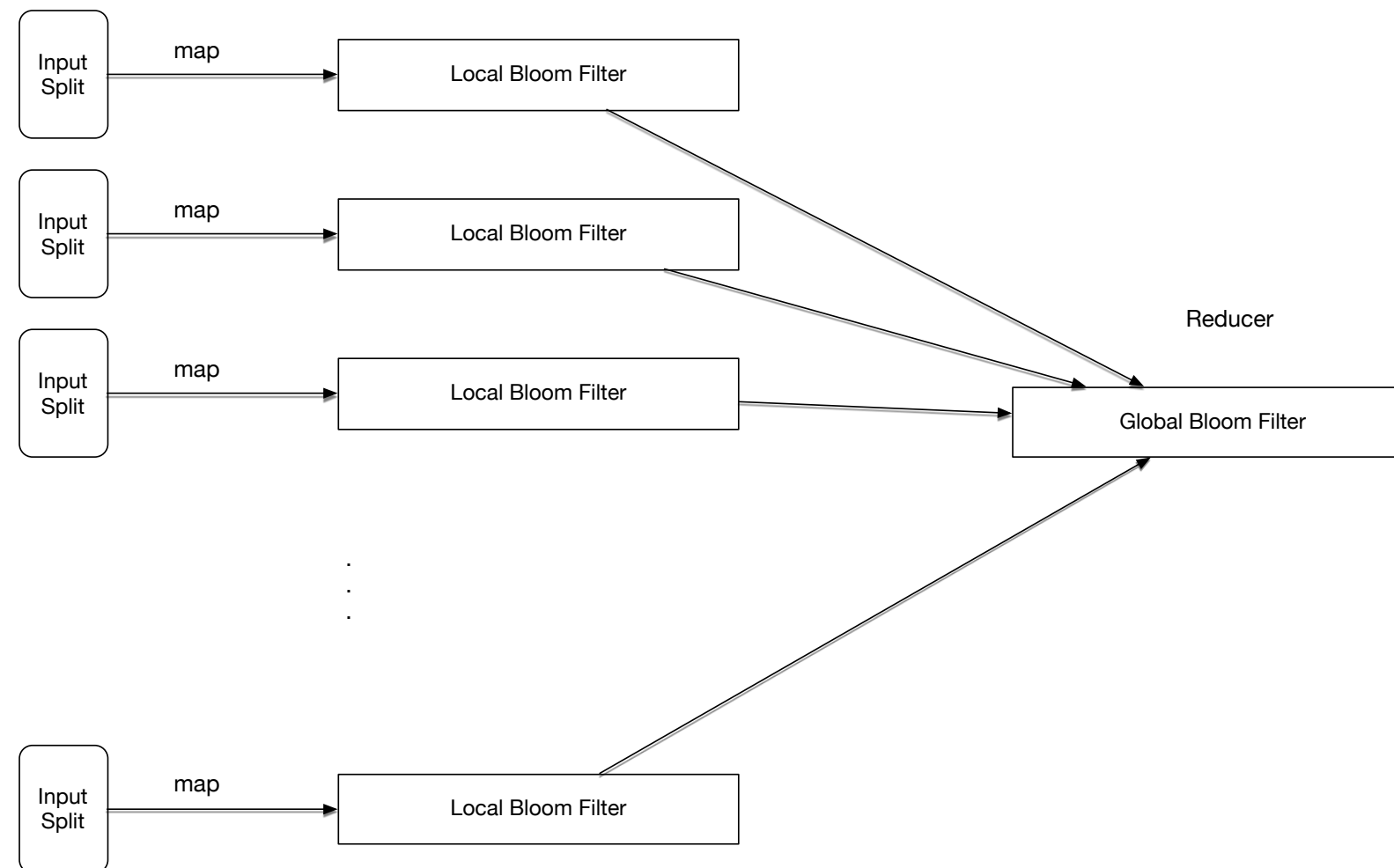
# Filtering Patterns

- Mappers create local bloom filter
- Reducer combines them



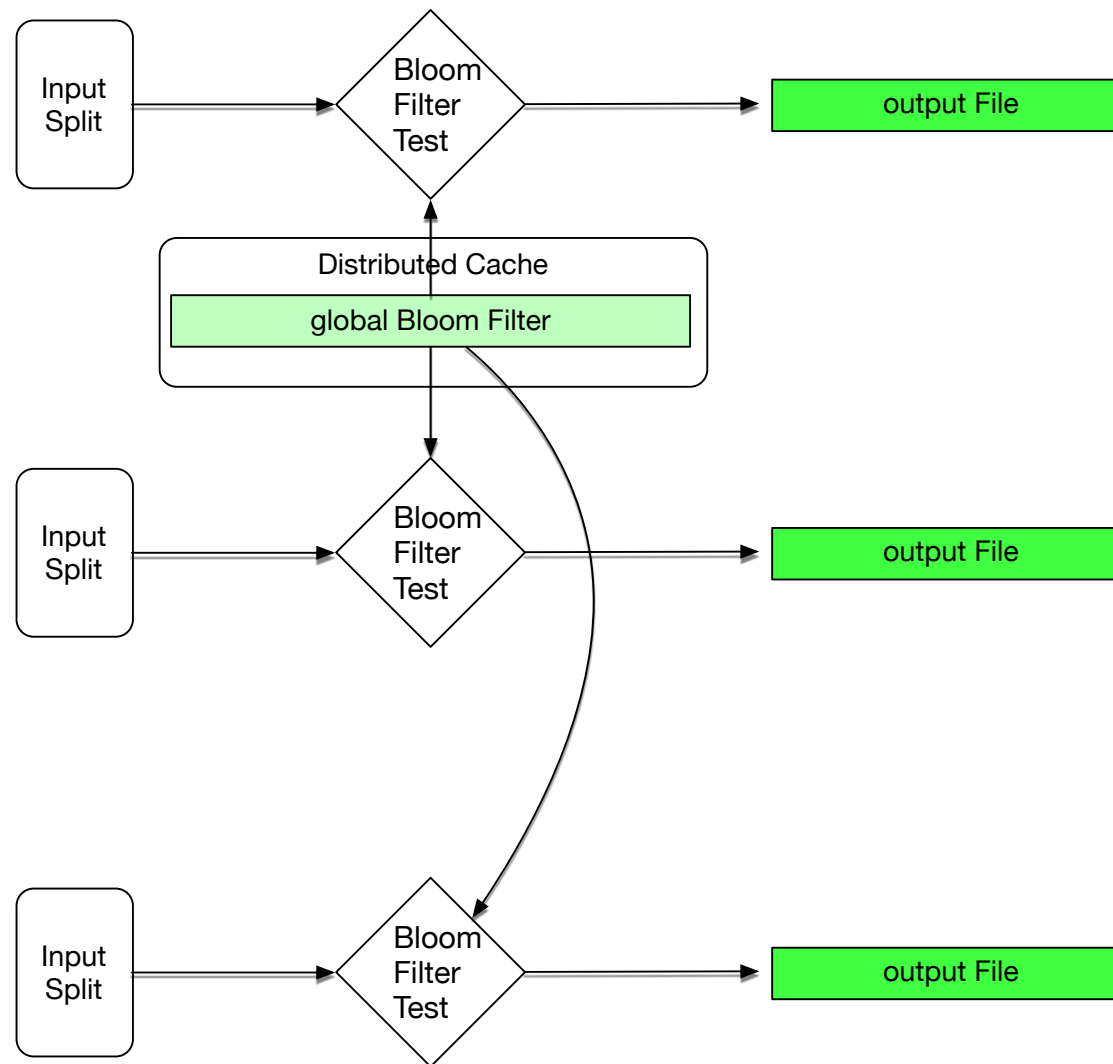
# Filtering Patterns

- Could use more than one reducer by breaking up local bloom filter into ranges



# Filtering Patterns

- Bloom Filtering



# Filtering Patterns

- Bloom Filtering is used for
  - Removing (almost all of) unwatched items
  - Prefiltering data
- Pig: Need to implement Bloom filtering as user-defined functions

# Filtering Patterns

- Top-ten
  - Retrieve the records that have the  $k$  largest values in a certain attribute
- Group Exercise



# Filtering Patterns

- A set of distinct records
  - Example: Web page log
    - Want to have records where user-name, device, or browser are different, but we don't care about time stamps

# Filtering Patterns

- Unique records:
  - Use the map-reduce grouping properties
    - Mapper group by the attributes we are interested in
    - Combiners emit one value for each group
    - Reducer only emits one value for each group

```
map(key, record) :
 emit record, null
```

```
reduce(key, records) :
 emit key
```

# Filtering Patterns

- Unique records
  - Pig:

```
b = distinct a;
```

# Data Organization Patterns

- Problem:
  - Transform data to a different format
    - Row-based data to hierarchical format such as JSON or XML

# Data Organization Patterns

- Example
  - StackOverflow data
    - Posts and comments are separated
      - Lines in an XML document
    - Hierarchy combines posts and comments
  - Hierarchical data model allows us to correlate length of posts with number of comments, etc.

```
Posts
 Post
 Comment
 Comment
 Post
 Post
 Comment
 Comment
 Comment
```

# Data Organization Patterns

- Often, the data to be combined comes from different data sets
  - Hadoop class `MultipleInputs` from `org.apache.hadoop.mapreduce.lib.input`
  - Allows to specify different input paths and different mappers for each input
  - Configuration is done in the driver

# Data Organization Patterns

- Multiple sources to hierarchical Pattern
  - Mappers load data and parse it into a cohesive format
  - Output key corresponds to root of hierarchical record
    - E.g. StackOverflow: root is post\_id
  - Need to identify the source for each mapper output
    - E.g. StackOverflow: is this a post or a comment
- Combiners are pretty useless because we create large strings

# Data Organization Patterns

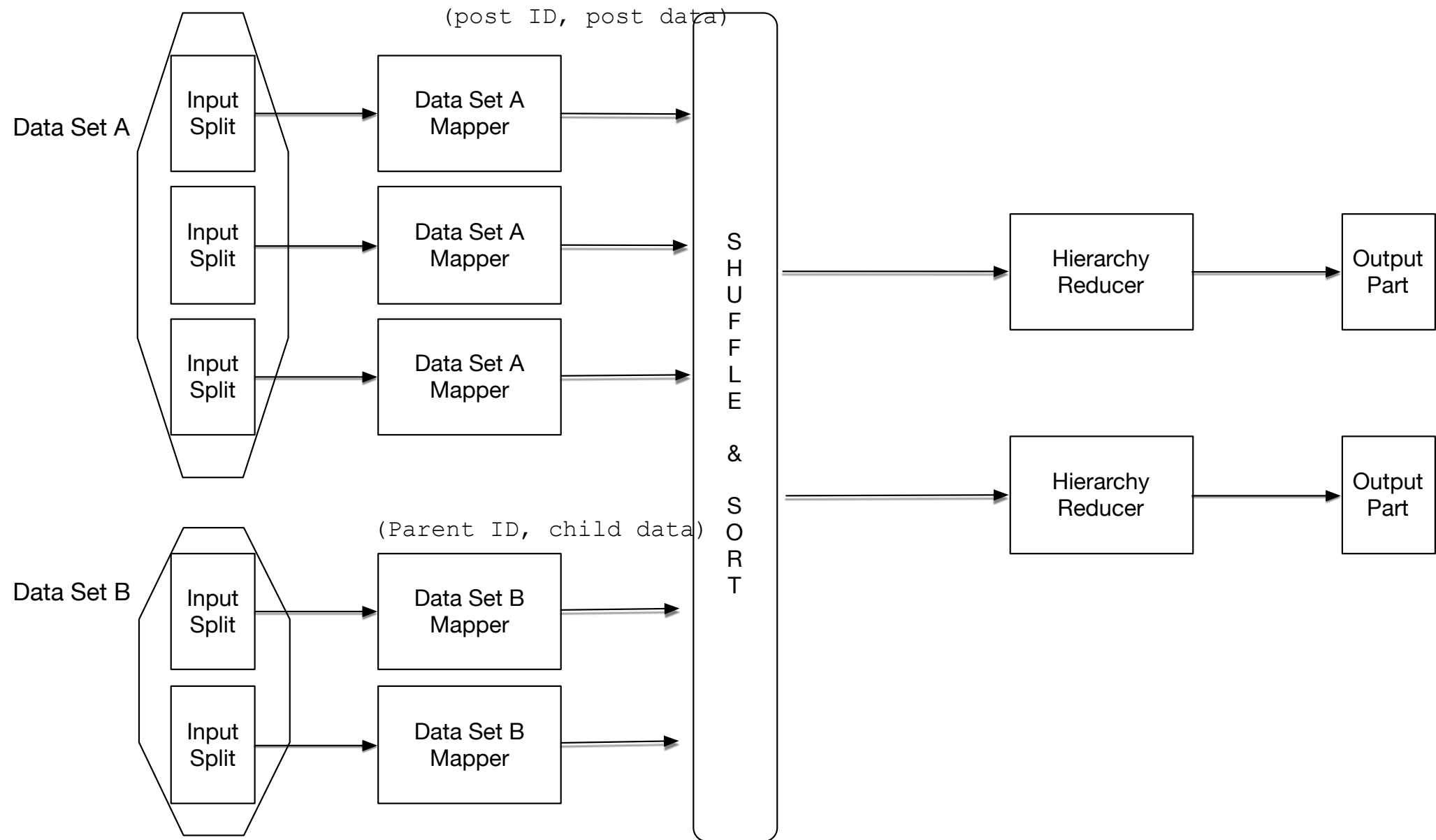
- Multiple sources to hierarchical Pattern
  - Reducers receive data from different sources key by key
  - For each key, can now build hierarchical data structure



# Data Organization Patterns

- Multiple sources to hierarchical Pattern
  - Result is in hierarchical form
  - Probably need to add header and footer so that it is well-formed

# Data Organization Patterns



# Data Organization Patterns

- Multiple sources to Hierarchical Pattern
  - Performance problems
    - Need enough reducers
    - Reducers might see a lot of skew:
      - Some are busy, others are not
    - Hot spots can result in humongous strings moved between mappers and reducers
      - Could take up the heap of a Java Virtual Machine

# Data Organization Patterns

- Needs two mappers: one for comments, one for posts
- Both: extract post-id to use as output key
- Append “P” or “C” to distinguish between sources

# Data Organization Patterns

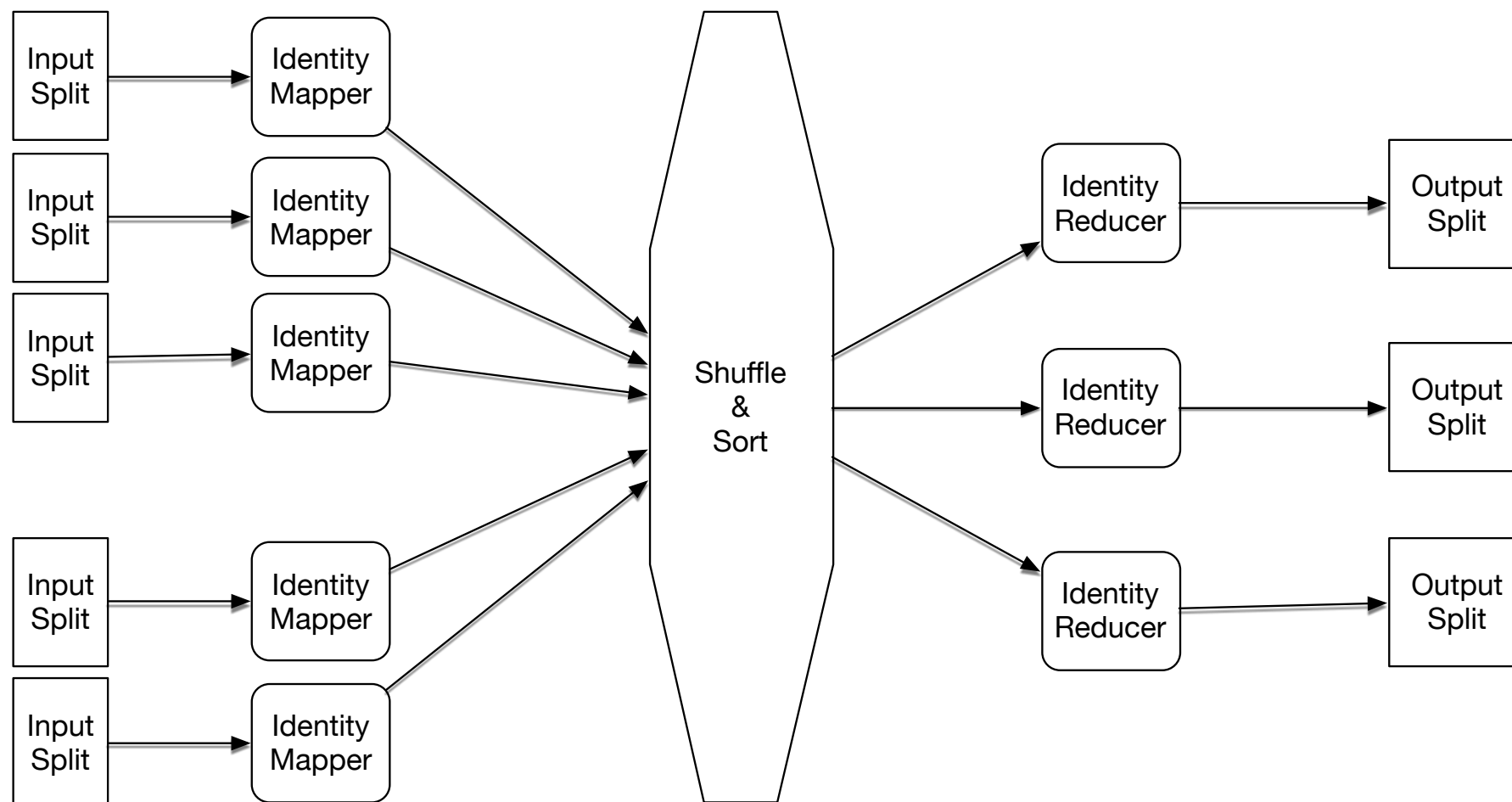
- Reducer:
  - Reducers receive post-id + marker as key and text as value
    - For each post-id with “P” marker:
      - Create a post entry in the XML
    - For each post-id with “C” marker:
      - Create child

# Data Organization Patterns

- Partitioning
  - Moves records into shards, but does not care about ordering
    - Example: partitioning by date
  - Need to know number of partitions ahead of time

# Data Organization Patterns

- Partitioning: Let the partitioner do the job



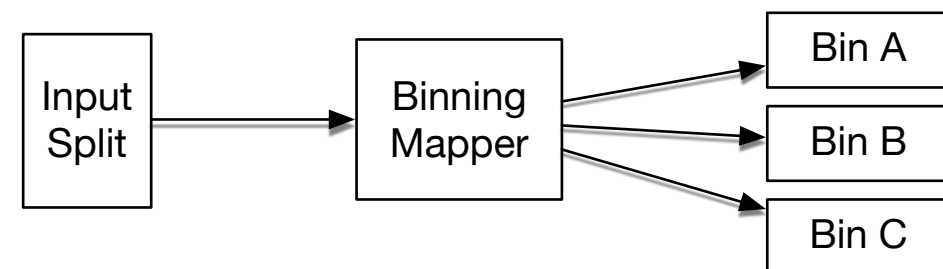
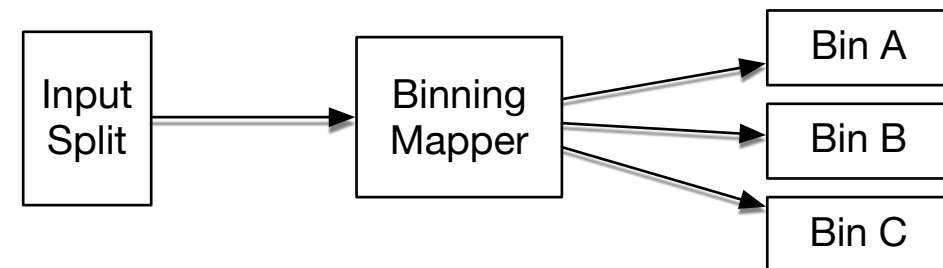
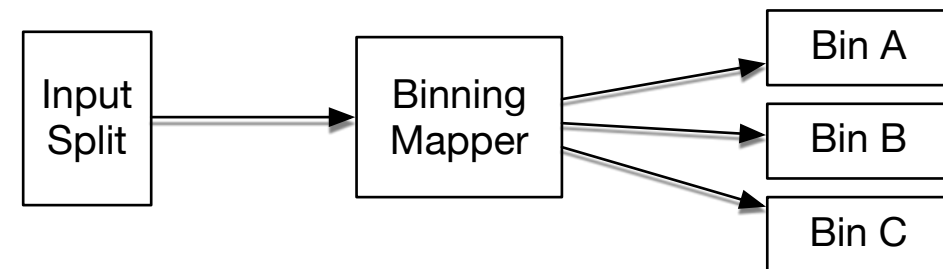
# Data Organization Patterns

- Binning:
  - Moves records into categories irrespective of order
  - Related to partitioning
    - Which one works better depends on the system
    - Bidders do not use reducers



# Data Organization Patterns

- Binning
  - Number of outputs =  
Number of mappers  
times Number of bins



# Data Organization Patterns

- Binning:
  - Pig
    - Split data INTO eights IF  $col1 == 8$ , bigs IF  $col1 > 8$ ,  
smalls IF  $col1 < 8$ ;

# Data Organization Patterns

- Total Order Sorting
  - Data needs to be sorted by a given comparator
  - Result is a set of shards that are ordered
    - Need to know the distribution of data first
      - Run an analyze phase first

# Data Organization Patterns

- Total order sorting:
  - Analyze phase
    - Mapper does random sampling
    - Only outputs the key after which we sort
    - Use only one reducer which will give us the sort keys in order

# Data Organization Patterns

- Total order sorting:
  - Order phase
    - Mapper extracts the sort key and stores the record as a value
    - Custom partitioner is loaded based on the results of the analysis phase
      - TotalOrderPartitioner in Hadoop
      - Takes the data ranges prescribed and uses them to partition
    - Reducer simply outputs the values
    - Shuffle and sort has already done all the work

# Data Organization Patterns

- Total Ordering in Pig
  - `c = order b by col1;`

# Data Organization Patterns

- Shuffling
  - Randomizes the order of a set of records

# Data Organization Patterns

- Shuffling
  - Mapper maintains the records, but creates a random key
  - Reducer sort according to random keys
  - Only record is printed out



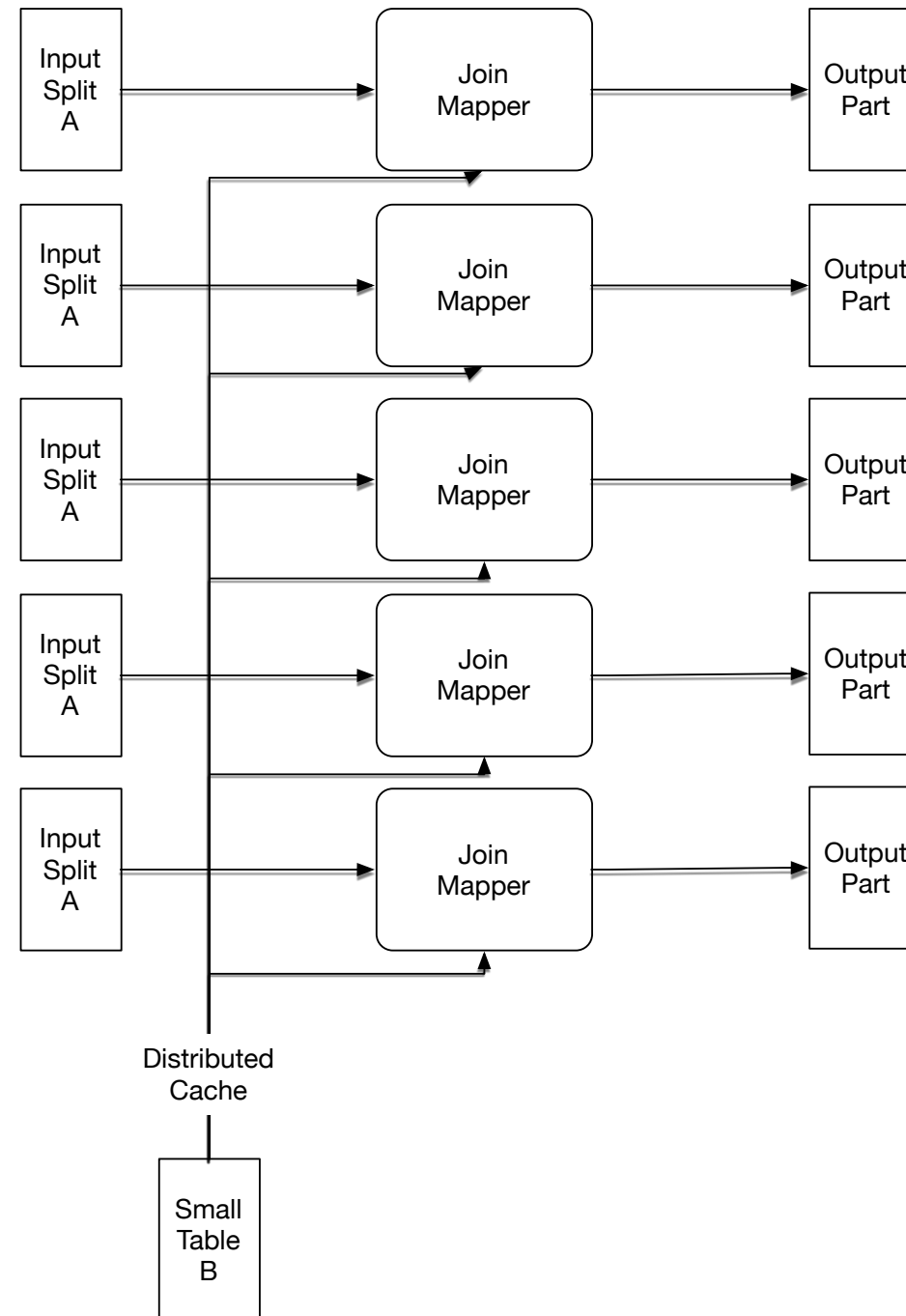
# Data Organization Patterns

- Shuffling
  - Pig:
    - `c = Group b by Random( );`
    - `d = FOREACH c generate Flatten(b);`

# Join Patterns

- Join Pattern
  - Different cases
    - Simplest Case: Join with a small table
      - Send small table to all mappers
      - Mappers calculate local join
      - Reducers

# Join Patterns



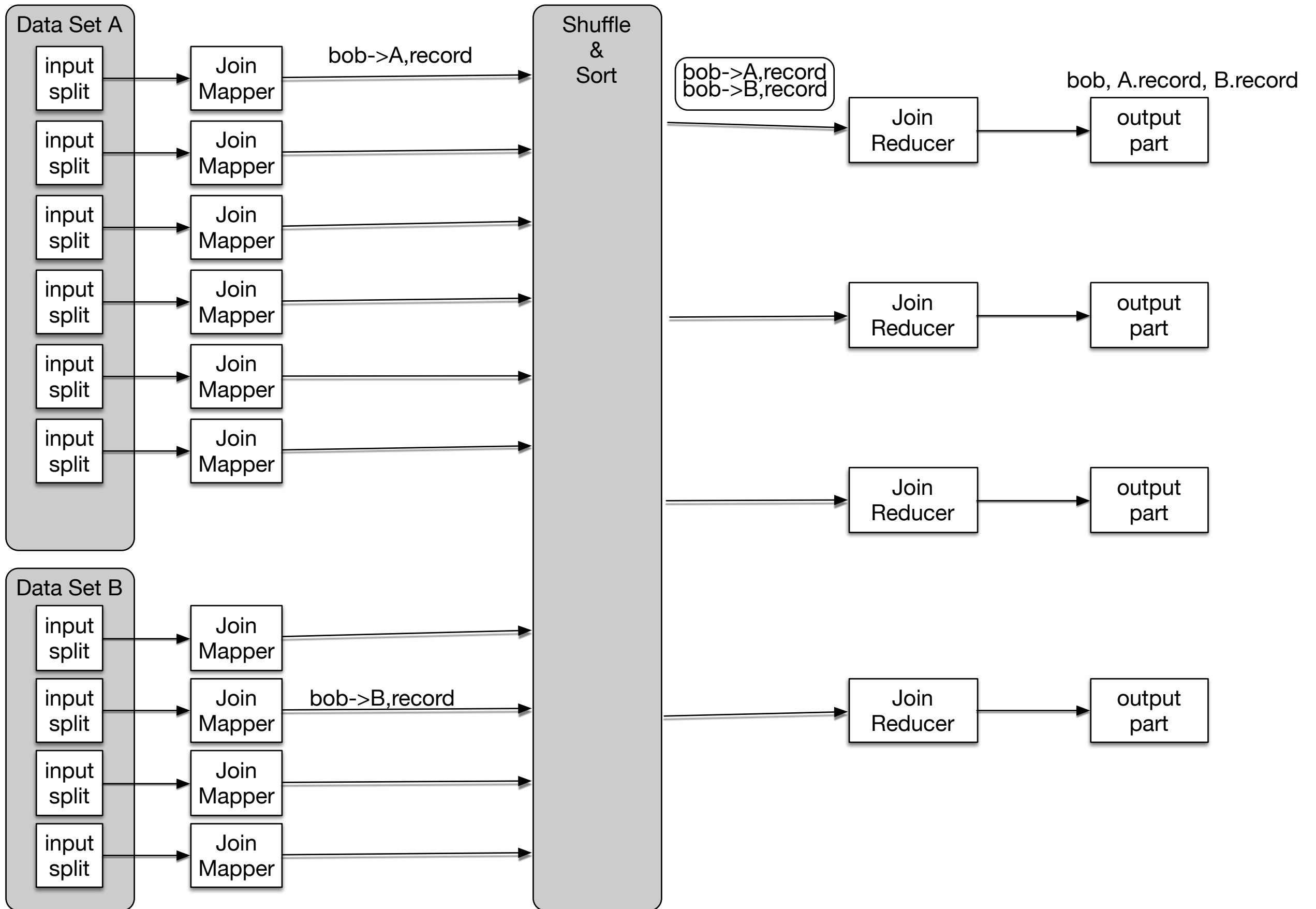
# Join Patterns

- Pig allows you to give hints for joins

```
huge = LOAD 'huge.txt' AS (h1,h2);
smallest = LOAD 'smallest.txt' AS (ss1, ss2);
small = LOAD 'small.txt' AS (s1,s2);
A = JOIN huge BY h1, small BY s1, smallest BY ss1 USING
'replicated';
```

# Join Patterns

- Reduce Side Join
  - Join large multiple data sets together by some foreign key
  - Structure:
    - Mapper goes through all records in both data sets
    - Mapper creates pairs
      - foreign key  $\rightarrow$  source, rest of record
      - source is the name of the table
    - Can use hash partitioner or a customized partitioner
    - Reducer combines values of each input group into two lists



# Join Patterns

- The reducer is given within the input group all records with a foreign key
- This allows the reducer to create many types of joins based on equality

# Join Patterns

- Inner join:
  - Records from Table A and Table B are joined if they share the same foreign key





# Join Patterns

- Outer Join
  - If the foreign key is not present in one table than the lacking values are made into Null values



# Join Patterns

- Optimization with Bloom Filters
  - If we calculate an inner join with map-reduce, a mapper does not have to create a key-value pair if the foreign key value is not present in the other table

**Table A**

| foreign key | attribute 1 | attribute 2 |
|-------------|-------------|-------------|
|             |             |             |
| value       |             |             |
|             |             |             |
|             |             |             |
|             |             |             |
|             |             |             |
|             |             |             |
|             |             |             |

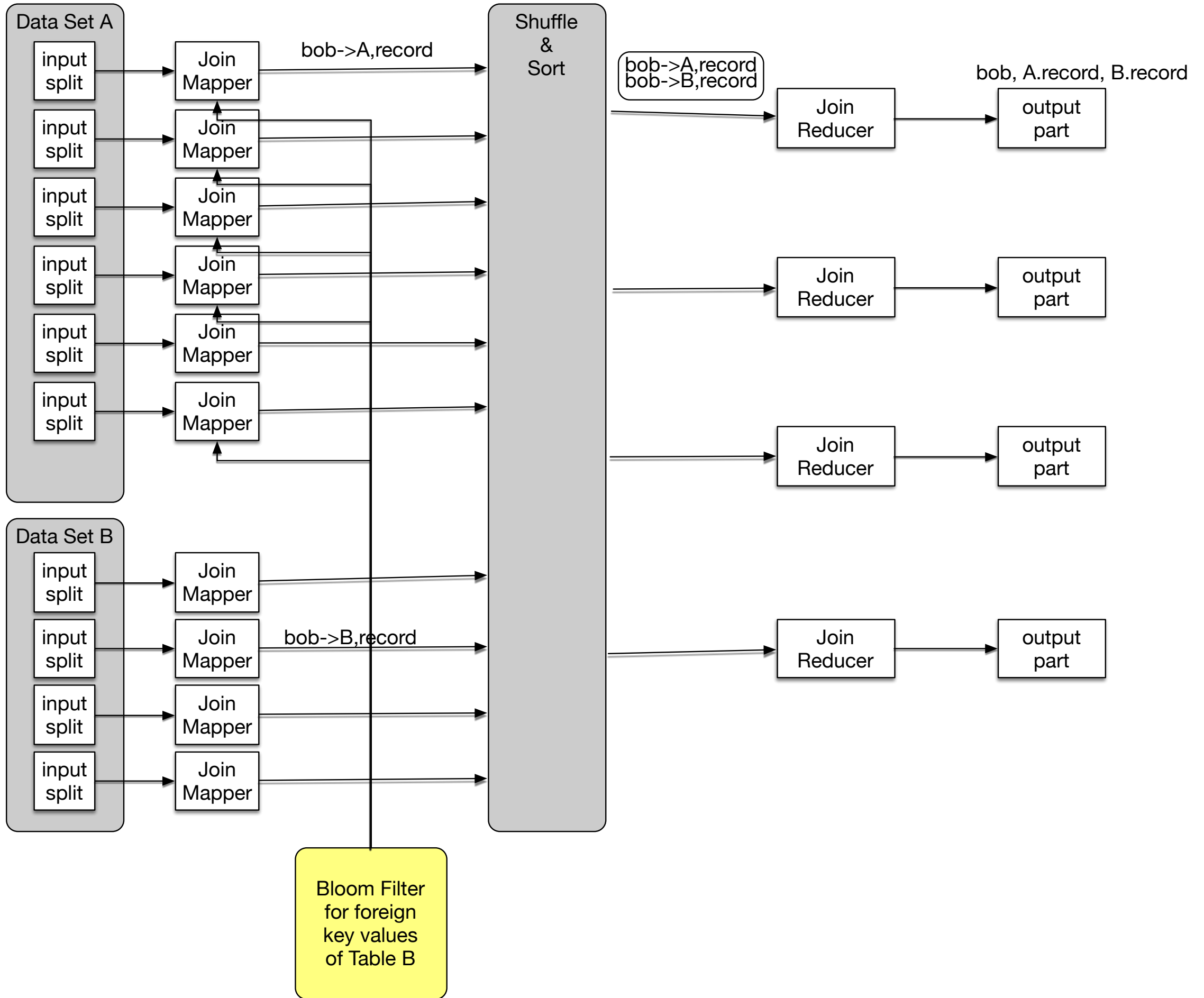
**Table B**

| foreign key | attribute 1 | attribute 2 | attribute 3 |
|-------------|-------------|-------------|-------------|
|             |             |             |             |
|             |             |             |             |
|             |             |             |             |
|             |             |             |             |
|             |             |             |             |
|             |             |             |             |
|             |             |             |             |

value not present

# Join Patterns

- Create a Bloom Filter for both sets (or for only one)
- Put Bloom filter in distributed cache or send it to all mappers for the other table
- Then only send records to the reducer when we know that the foreign key value is also present in the other table



# Join Patterns

- Bloom filter does not need to be very good to be efficient
  - Bloom filter does not have false negatives, only false positives
- Bloom filter needs to be created before it can be used, making this into a two phase job

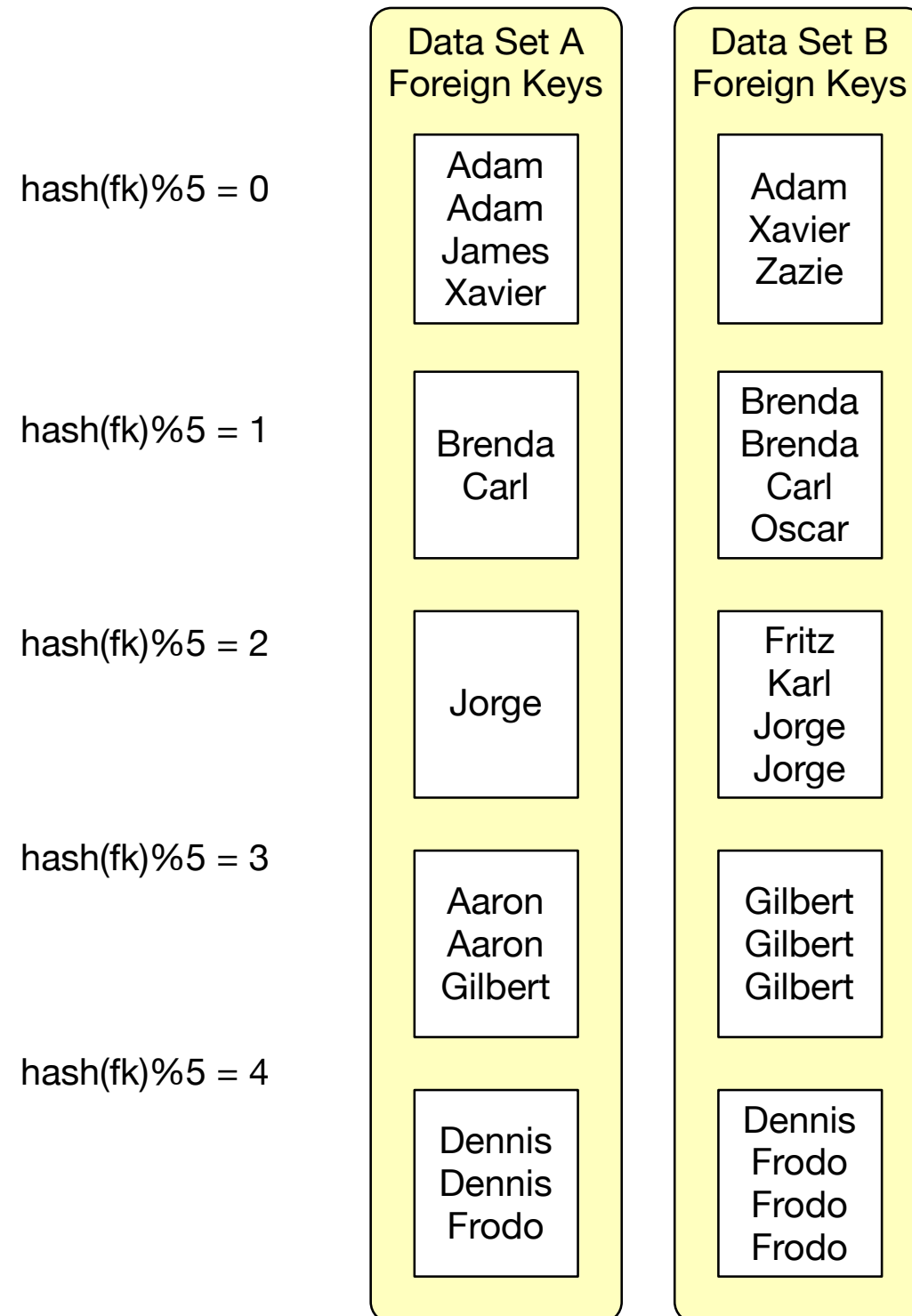
# Join Patterns

- Composite Join
  - Supported by Hadoop: CompositeInputFormat
- Idea:
  - Preprocess all table contents to create input shards that are sorted and partitioned by foreign key.
  - This is somewhat similar to the idea of the hash-join
- Two-phase job again



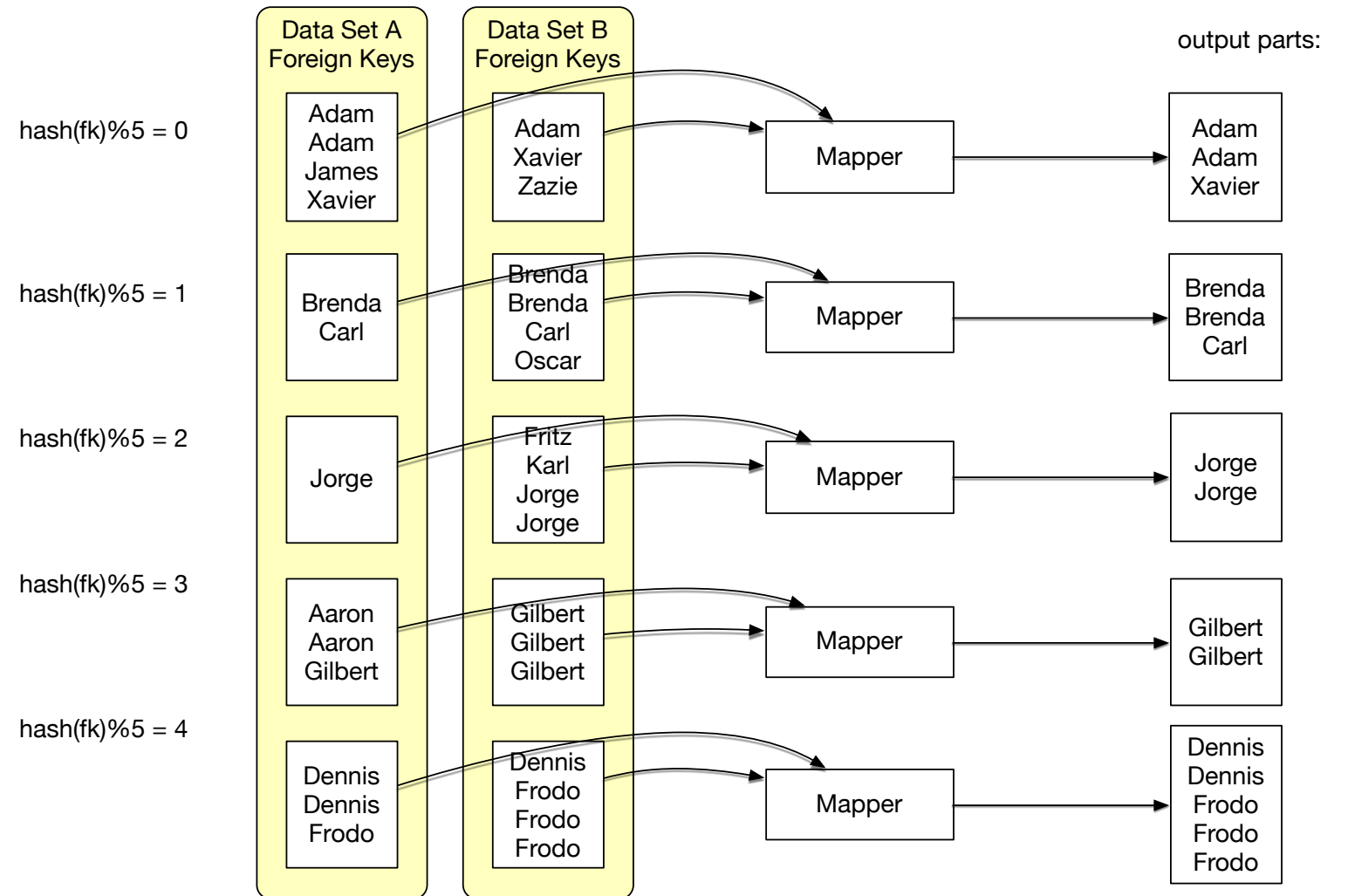
# Join Patterns

- Create hash buckets for records based on foreign keys
- Buckets are sorted



# Join Patterns

- Send all buckets in the hash to the same mapper
- Mappers then combine
- There are no reducers involved

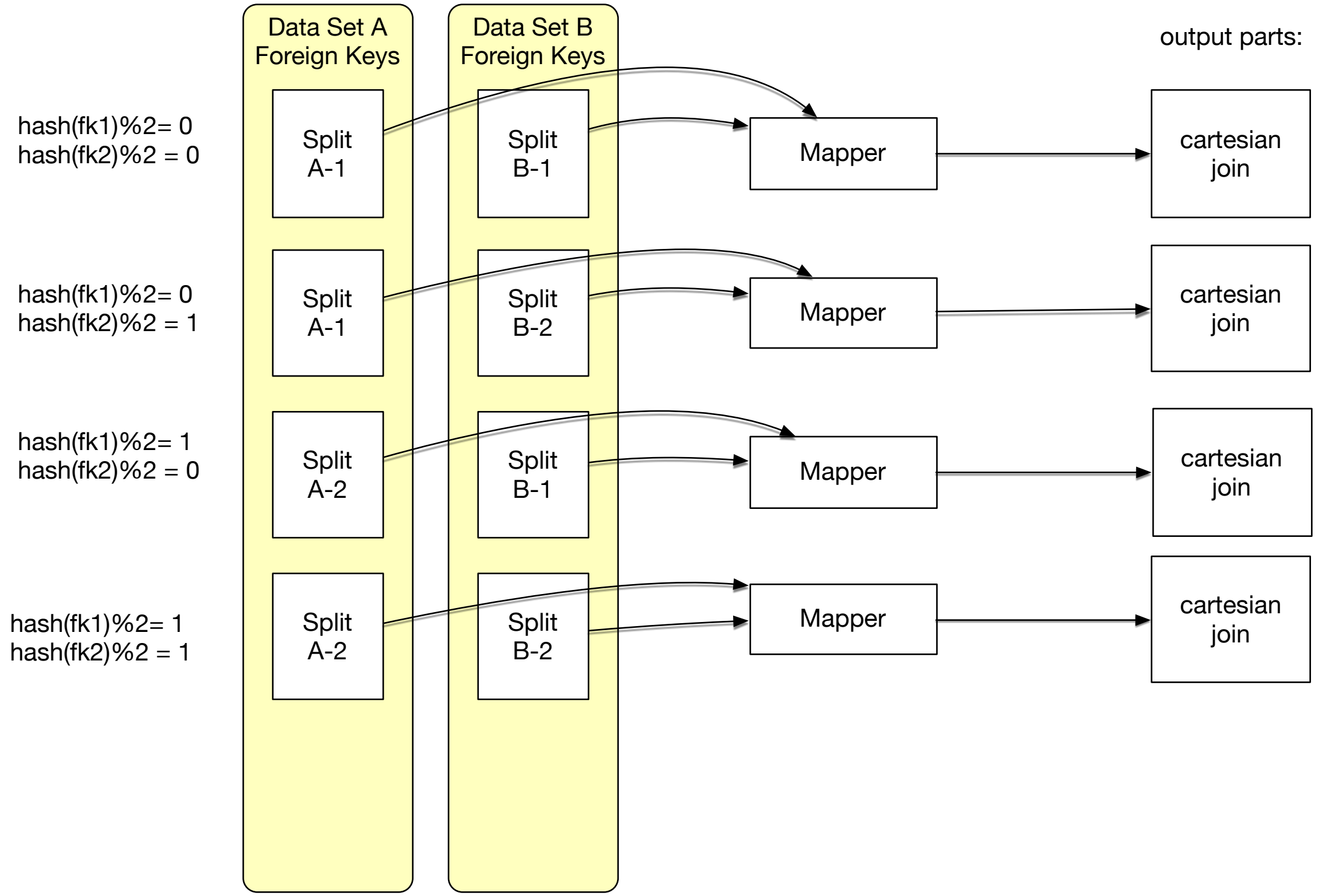


# Join Patterns

- Composite Join Performance
  - Most of the work in the creation of the Composite Join Splits

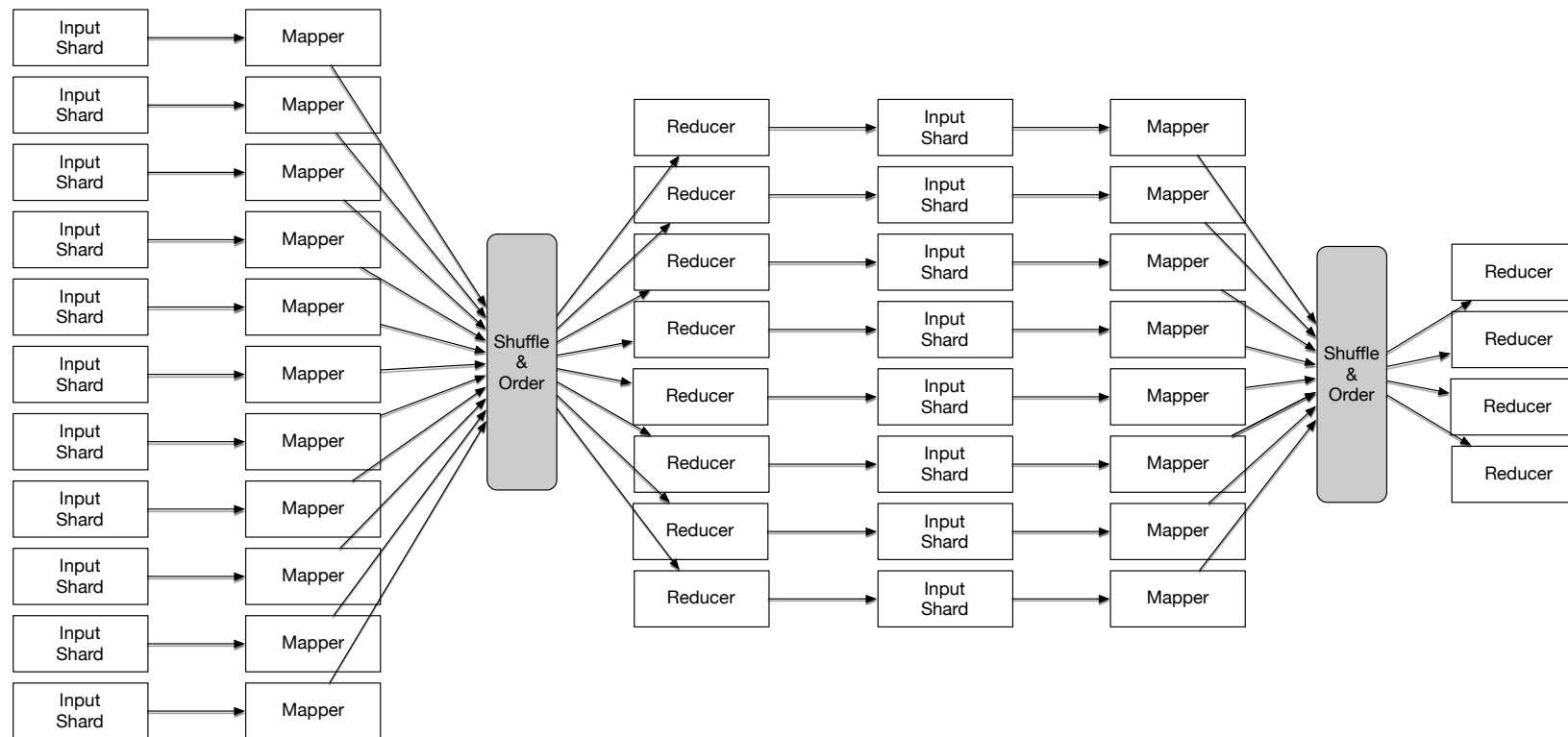
# Join Patterns

- Cartesian Product
  - Cartesian product combines all values in one table with all values in one other table
  - Easily create gigantic results
  - With huge execution times
- Uses essentially the composite join pattern
  - Each mapper receives an input split from Table A and an input split from Table B
  - If we break Table A into  $n$  pieces and Table B into  $m$  pieces, then we need  $n \times m$  mappers



# Meta Patterns

- Job Chaining
  - Run two or more map-reduce jobs
  - Output of the first is input to the second



# Meta Patterns

- Map-Reduce framework is not very good at this
  - Special frameworks exist
    - Oozie – Apache Project Workflow Engine
      - Java web application
      - Workflow - collection of actions
        - Map-reduce jobs, Pig jobs arranged in a DAG
        - Uses XML-based Hadoop Process Definition Language for workflow specifications
    - Oozie coordinates jobs

# Meta Patterns

- Doing it yourself
  - Using Java
  - Map-reduce drivers are simple Java classes
  - Take the drivers of the individual Map-reduce jobs and call them in sequence
    - Output and input paths need to match
  - Use `Job.Submit()`, `Job.waitForCompletion()`, and `Job.waitForCompletion()`



# Meta Patterns

- Doing it yourself
  - Scripting
  - Using JobControl

# Chain Folding

- Opportunities for optimization
  - Mappers work in isolation
  - A record can be submitted to multiple mappers or to a reducer-mapper combination
- Avoids transfer of files

# Chain Folding

- Chain folding patterns
  - Multiple mapping phases are adjacent
    - Fold them into single mappers
  - If a job ends with a map phase
    - Push into the preceding reducer phase
  - Split mappers that reduce data and mappers that increase data
  - Filter data as early as possible

# Job Merging

- If two unrelated map-reduce jobs use the same input set
  - Can combine the mappers and reducers
  - Data is loaded and parsed now only once