# MapReduce
# Design Patterns

(Based on "MapReduce Design Patterns" by Donald Miner and Adam Shook)

Linh B. Ngo

# CONTENTS

- Motivation
- Basic Patterns
- Summarization
- Filtering
- Data Organization
- Join
- Metapatterns

# MOTIVATION

- MapReduce is designed as a framework
  - The solution has to fit into the framework
  - Clear boundaries on what can and cannot be done
  - Creating a solution within boundaries is a challenge
- MapReduce Design Patterns
  - A template for solving a common and general data manipulation problem with MapReduce
  - Is not specific to a domain, but a general approach to solving a problem

# BASIC PATTERNS

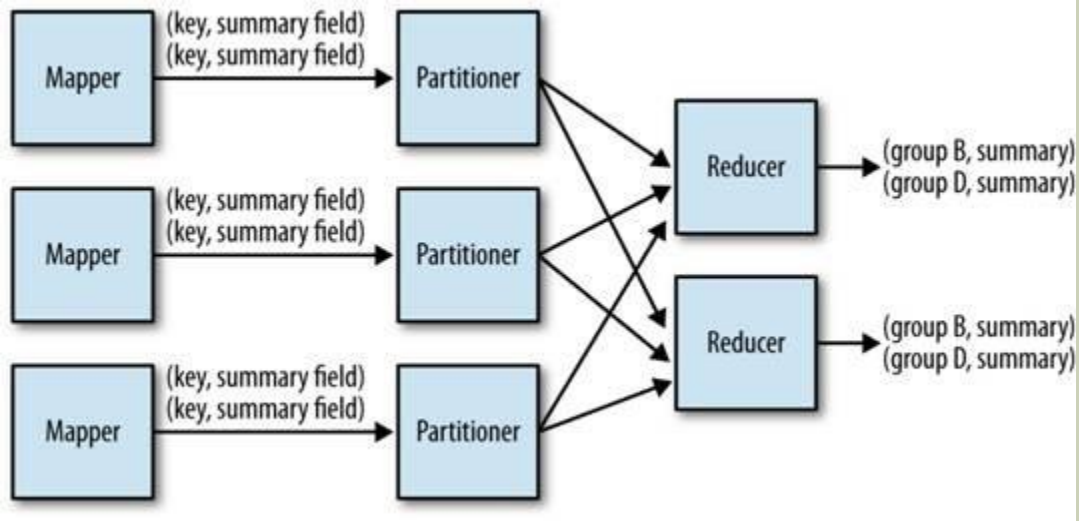
- Summarization
- Filtering
- Data Organization
- Join
- Metapatterns

# SUMMARIZATION

- Grouping similar data together and performance an operation such as calculating a statistic, building an index, or just simply counting
- Examples
  - Numerical Summarizations
  - Inverted Index
  - Counting with Counters

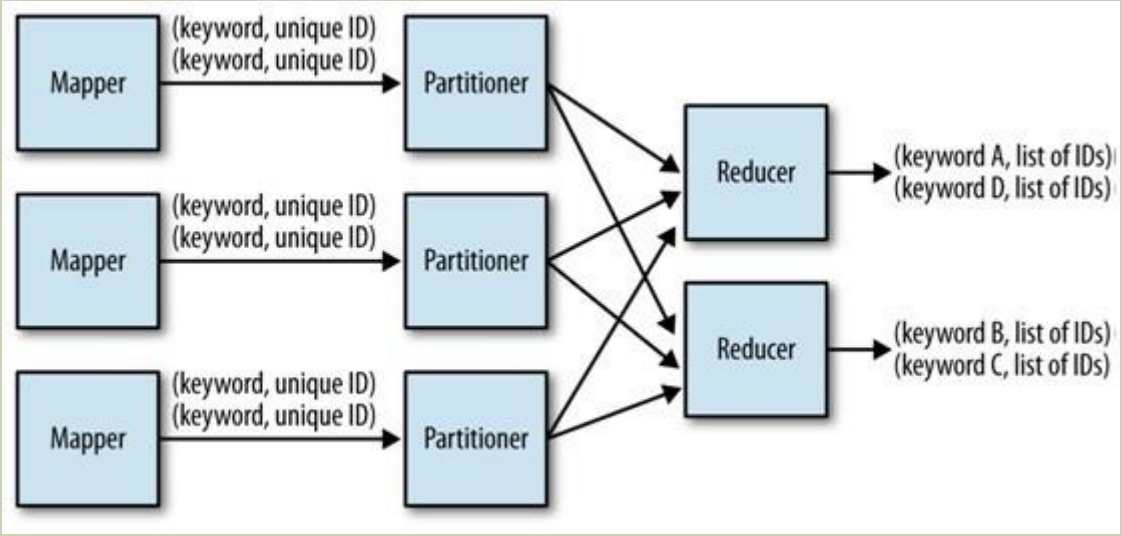
# SUMMARIZATION
## NUMERICAL SUMMARIZATIONS

- Calculate aggregate statistical values over a large data set
- Benefit from a properly-designed combiner (How?)
- Benefit from a properly-designed partitioner (How?)



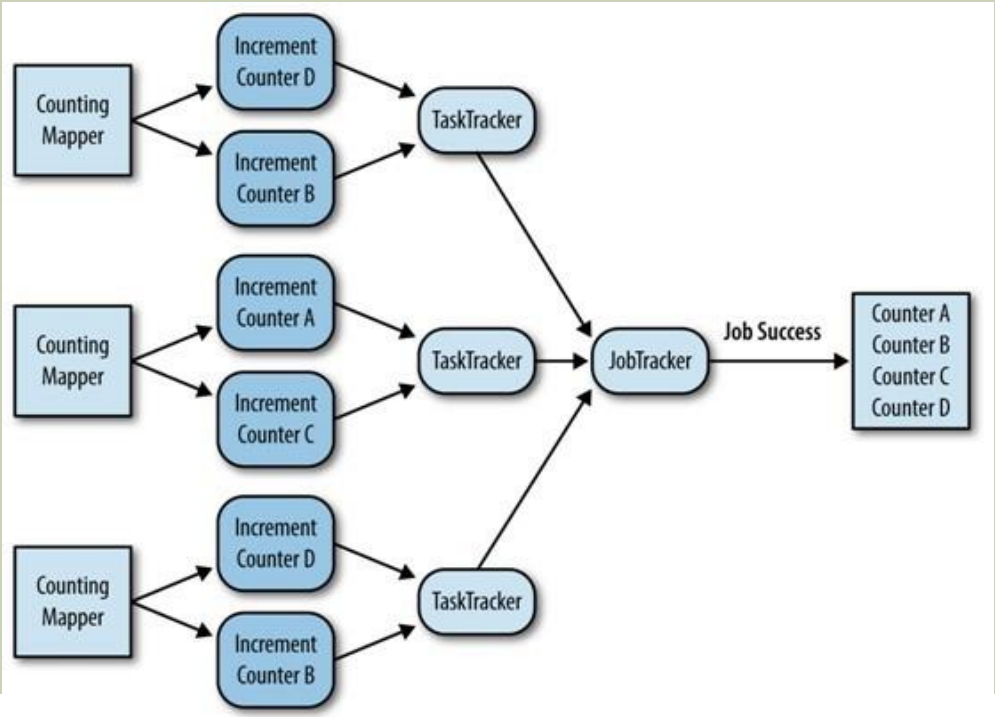
# SUMMARIZATION
## INVERTED INDEX

- Generating an index from a data set to allow for faster searches or data enrichment capabilities
- Benefit from the combiner (How?)



# SUMMARIZATION
## COUNTING WITH COUNTERS

- Utilizing MapReduce's internal counter to calculate global sums on the entire data set
- Done entirely in the mapping phase
- Useful if number of keys is small



# SUMMARIZATION
## COUNTING WITH COUNTERS

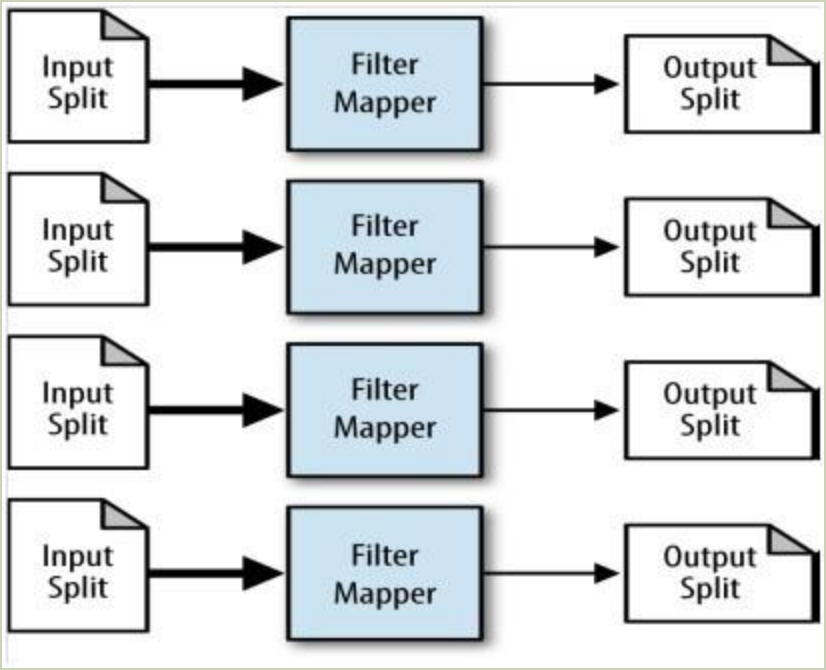
```
public static classCountNumUsersByStateMapper extendsMapper<Object,Text, NullWritable,
NullWritable> {
public static finalString STATE_COUNTER_GROUP ="State";
public static finalString UNKNOWN_COUNTER ="Unknown";
public static finalString NULL_OR_EMPTY_COUNTER ="Null or Empty";
privateString[]statesArray =new String[] {"AL", "AK", … };
privateHashSet<String>states = new HashSet<String>(Arrays.asList(statesArray));

publicvoid map(Object key,Text value,Context context) throwsIOException,InterruptedException {
// Parsed data from value, get the value for the Location attribute, look for a state abbreviation
code if the location is not null or empty
// For each token, check if it is a state. If so, increment the state's counter by I and flag it as
not unknown
context.getCounter(STATE_COUNTER_GROUP,state).increment(1);
…
// If the state is unknown, increment the UNKNOWN_COUNTER counter
context.getCounter(STATE_COUNTER_GROUP,UNKNOWN_COUNTER).increment(1);

// If it is empty or null, increment the NULL_OR_EMPTY_COUNTER counter by 1
context.getCounter(STATE_COUNTER_GROUP,NULL_OR_EMPTY_COUNTER).increment(1);
}
```

# FILTERING

- Don’t change the actual records but find a subset of data in the entire data set
- Examples
  - Filtering
  - Top-ten
  - Distinct

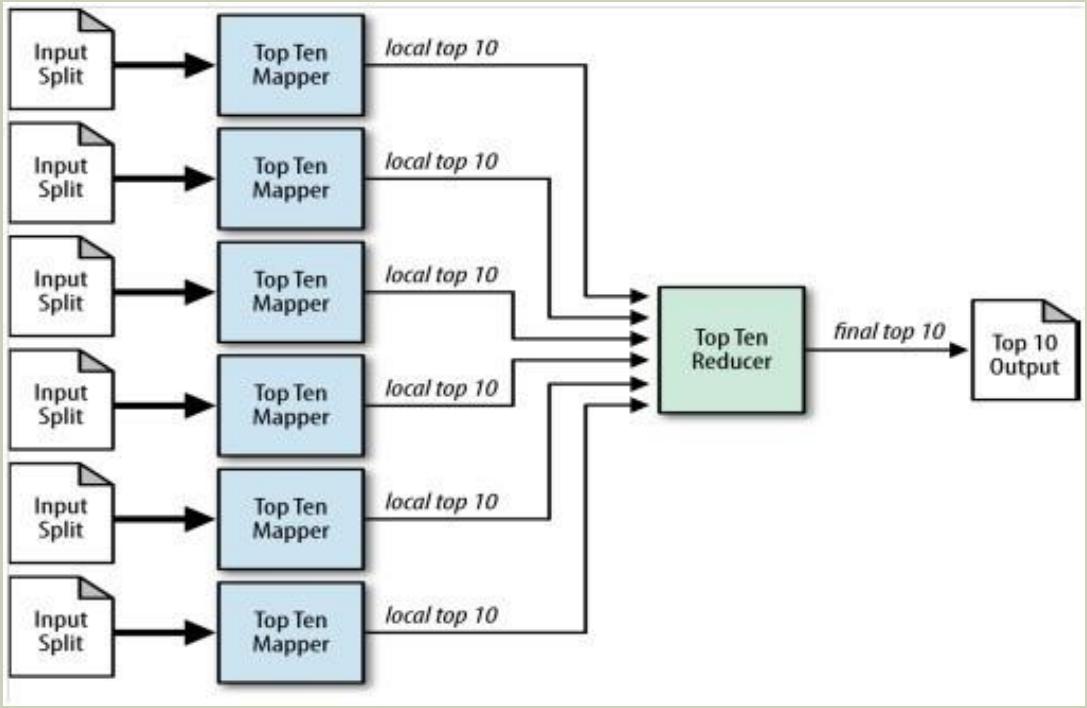
# FILTERING
## FILTERING

- Evaluates each record and decides, based on some condition, whether it should stay or go
- Not require the “reduce part”
- Can a combiner helps?



# FILTERING
## TOP-TEN

- Retrieve a relatively small number of top K records according to a ranking scheme in the data set
- Important: Unique entries



# FILTERING
## DISTINCT

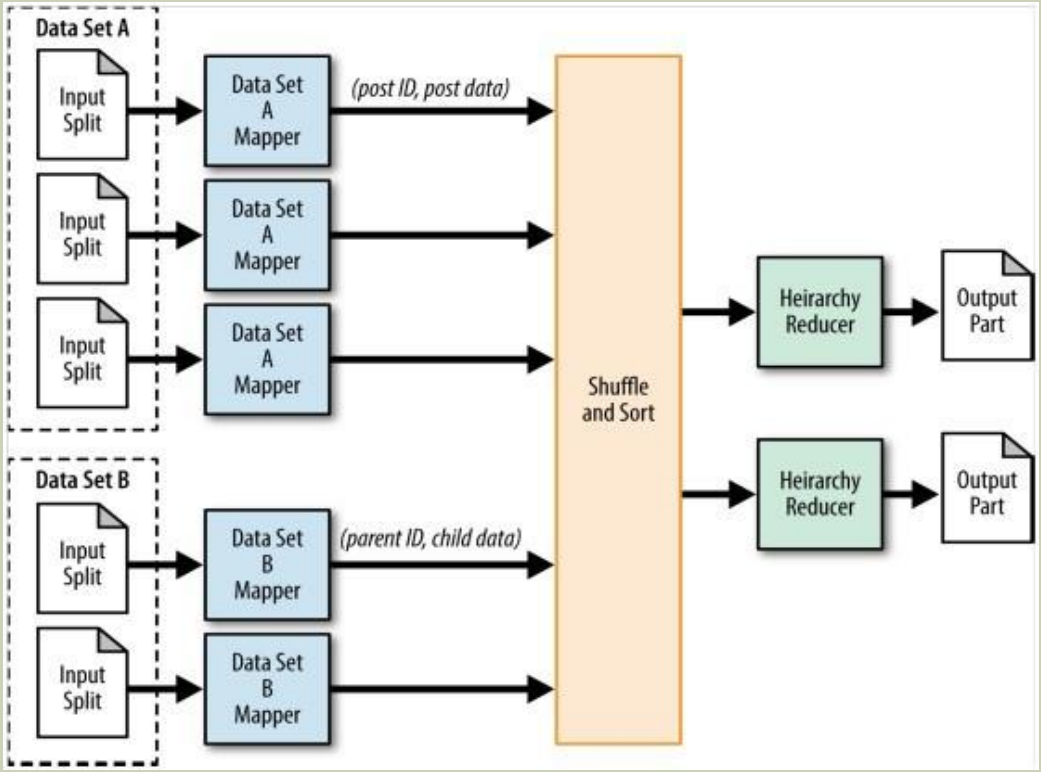
- Find a unique set of values
- Key process: deduplication
- Benefit from deduplication processes implemented inside mappers and combiners
- Benefit from a large number of reducers

# DATA ORGANIZATION

- Reorganizing data to help improving value, performance, and usability.
- Examples:
  - Structured to Hierarchical
  - Partitioning and Binning

# DATA ORGANIZATION
## STRUCTURED TO HIERARCHICAL

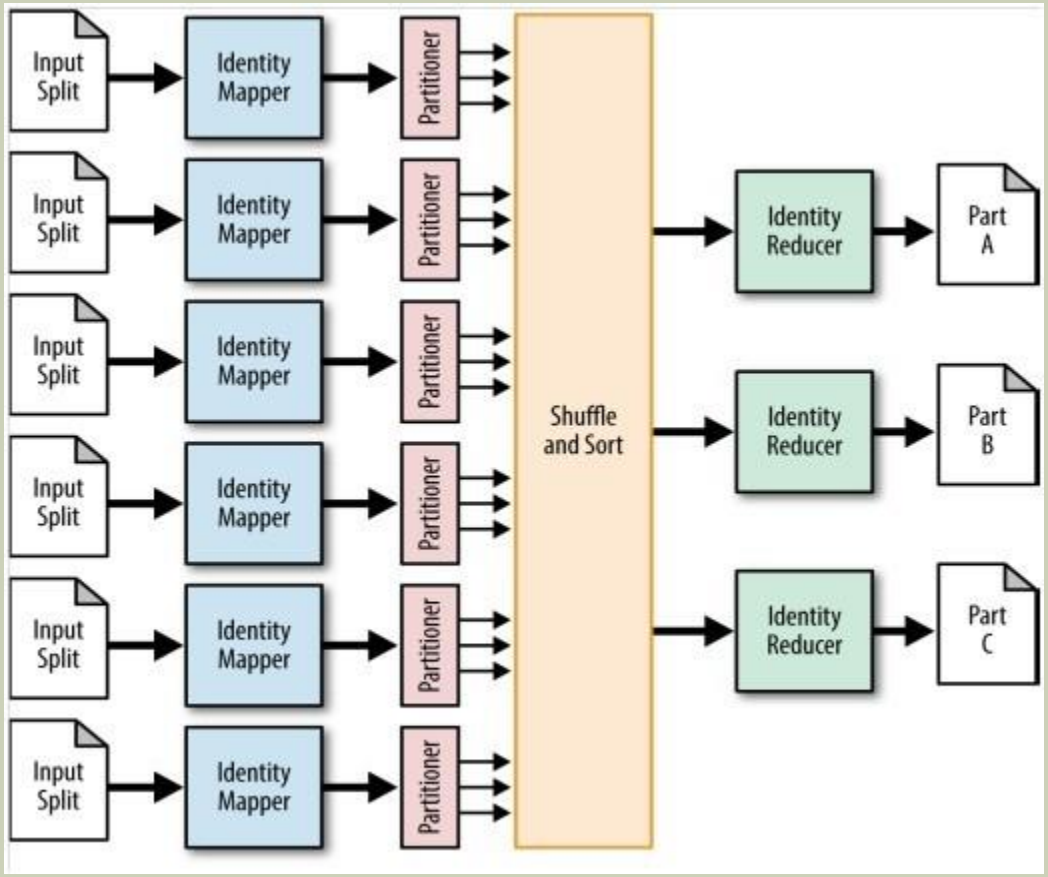
- Create new records with a different structure from the input data (e.g.: transform row-based data into a hierarchical format such as XML or JSON)

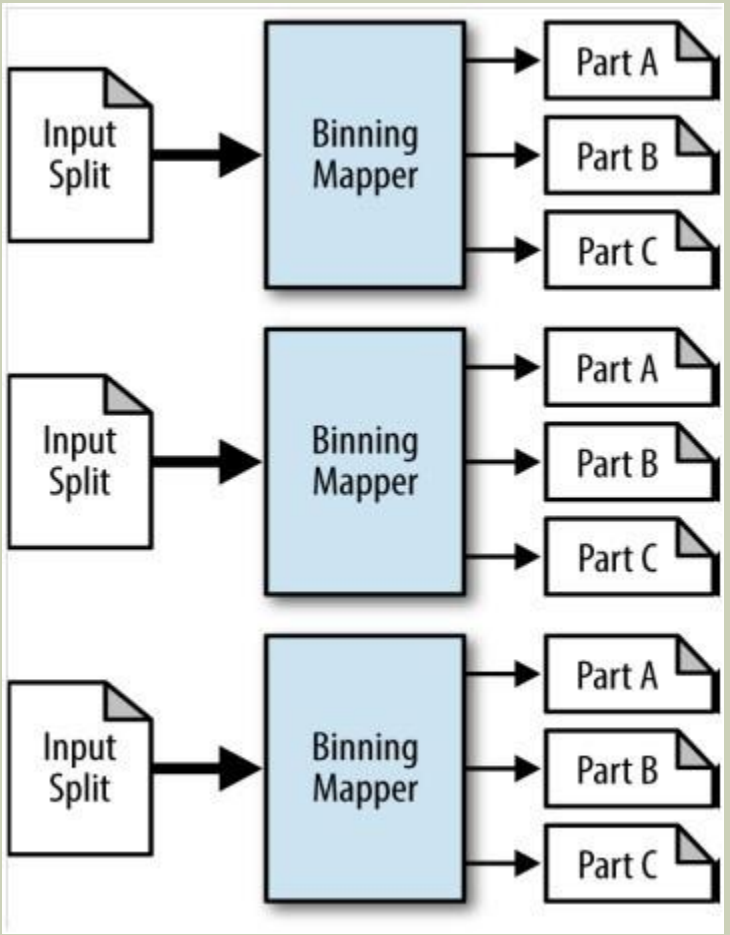


# DATA ORGANIZATION
## PARTITIONING AND BINNING

- Relies on Hadoop's Partitioner and Binning capabilities to categorize data based on key ranges.
- Binning: Mapping phase
- Partitioning: After mapping

# DATA ORGANIZATION
## PARTITIONING AND BINNING



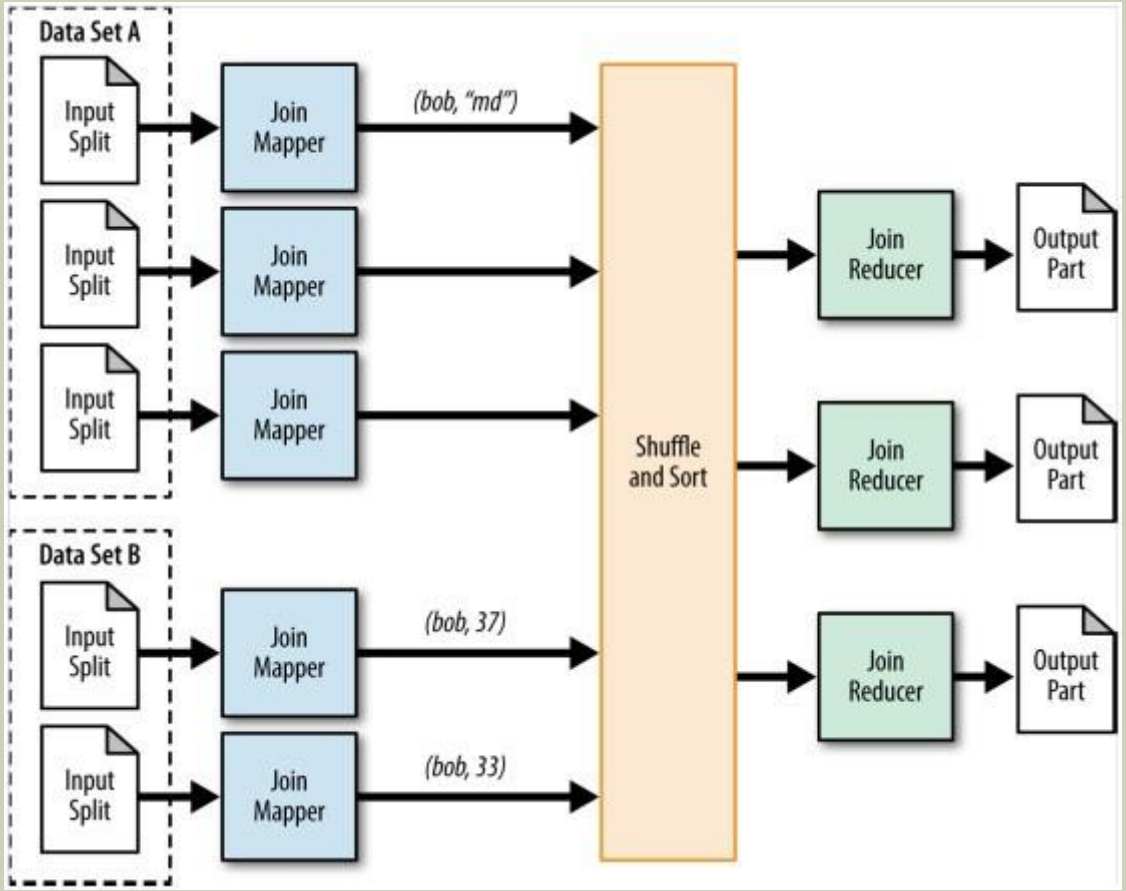
# JOIN

- Merging similar data (with or without aggregation) from multiple data sets
- Examples:
  - Reduce-side join
  - Map-side join
  - Cartesian product

# JOIN
## REDUCE-SIDE JOIN

- Join large multiple data sets together by some foreign keys
- Simplest and most straight forward
- Should be the last solution to look at

# JOIN
## REDUCE-SIDE JOIN

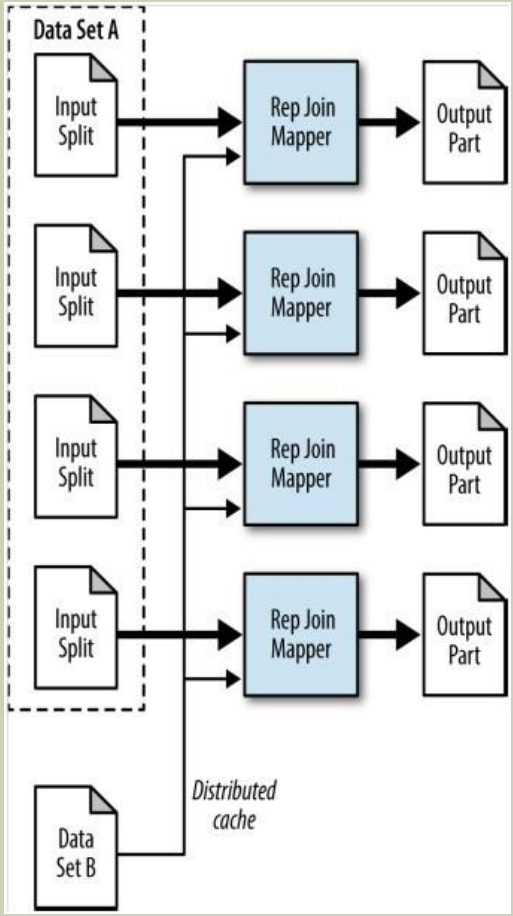


# JOIN
## MAP-SIDE JOIN

- Joining happens in the map phase
- Replicated Join
- Composite Join

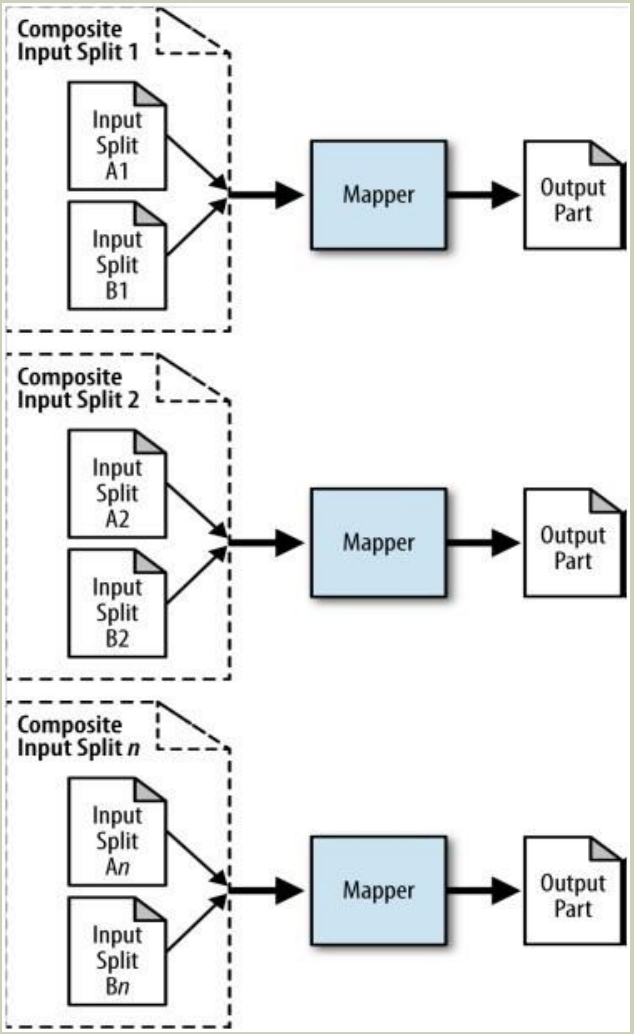
# JOIN
## MAP-SIDE JOIN: REPLICATED JOIN

- Use distributed cache on all mappers



# JOIN

## MAP-SIDE JOIN: COMPOSITE JOIN



# JOIN
## COMPOSITE JOIN

```
public static void main(String[] args) throws Exception {
   Path userPath = new Path(args[0]);
   Path commentPath = new Path(args[1]);
   Path outputDir = new Path(args[2]);
   String joinType = args[3];
   JobConf conf = new JobConf("CompositeJoin");
   conf.setJarByClass(CompositeJoinDriver.class);
   conf.setMapperClass(CompositeMapper.class);
   conf.setNumReduceTasks(0);
   // Set the input format class to a CompositeInputFormat class.  The CompositeInputFormat will
parse      all of our input files and output records to our mapper.
   conf.setInputFormat(CompositeInputFormat.class);
   // The composite input format join expression will set how the records are going to be read in,
and in what input format.
   conf.set("mapred.join.expr", CompositeInputFormat.compose(joinType,
         KeyValueTextInputFormat.class, userPath, commentPath));
   TextOutputFormat.setOutputPath(conf, outputDir);
   conf.setOutputKeyClass(Text.class);
   conf.setOutputValueClass(Text.class);
   RunningJob job = JobClient.runJob(conf);
}
```

# JOIN
## COMPOSITE JOIN

```
public static class CompositeMapper extends MapReduceBase
implements
      Mapper<Text, TupleWritable, Text, Text> {
   publicvoid map(Text key, TupleWritable value,
         OutputCollector<Text, Text> output, Reporter reporter)
         throws IOException {
      // Get the first two elements in the tuple and output them
      output.collect((Text) value.get(0), (Text) value.get(1));
   }
}
```

# METAPATTERNS

- Patterns that deal with patterns
- Examples:
  - Job chaining
  - Job merging

# METAPATTERNS
## JOB CHAINING

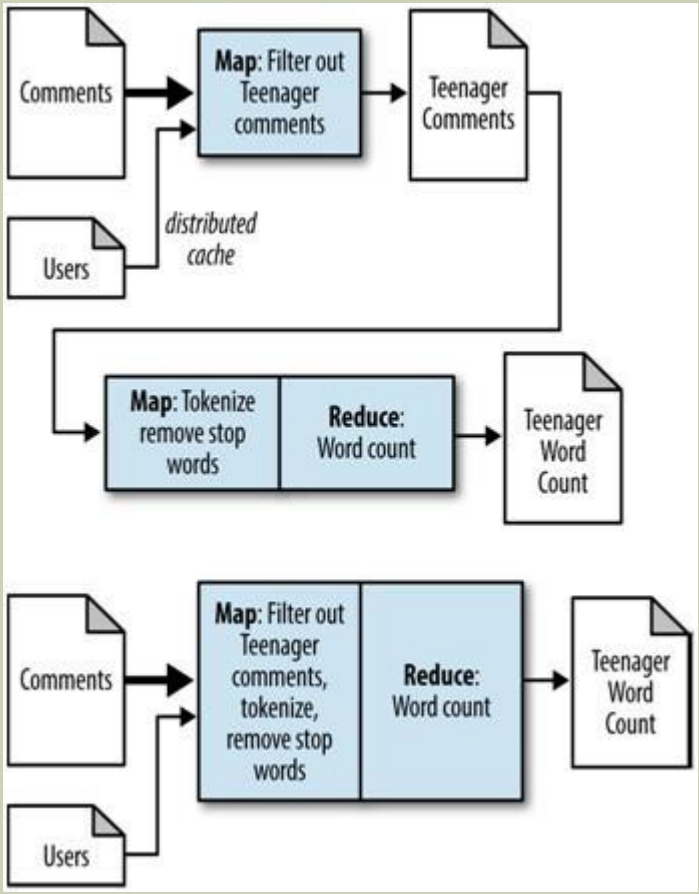
- Many problems can't be solved with a single MapReduce job
- The default MapReduce framework requires a lot of manual coding to handle multistage jobs
  - Clean up intermediate outputs
  - Handle failures
- Can be chained though the main Java program or external scripting
- Possible support tools: Oozie, Cascade, Tez …
- Serial chaining
- Parallel chaining

# METAPATTERNS
## JOB CHAINING: CHAIN FOLDING

- Optimization that is applied to MapReduce job chain
- Reduce the amount of data movement in the MapReduce pipe line

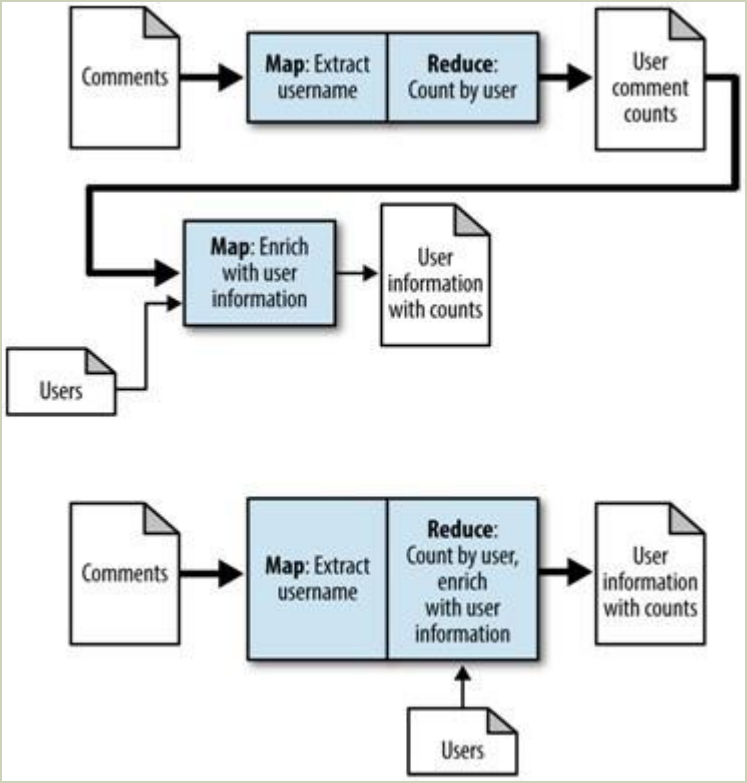
# METAPATTERNS
## JOB CHAINING: CHAIN FOLDING

- If multiple map phases are adjacent, merge them



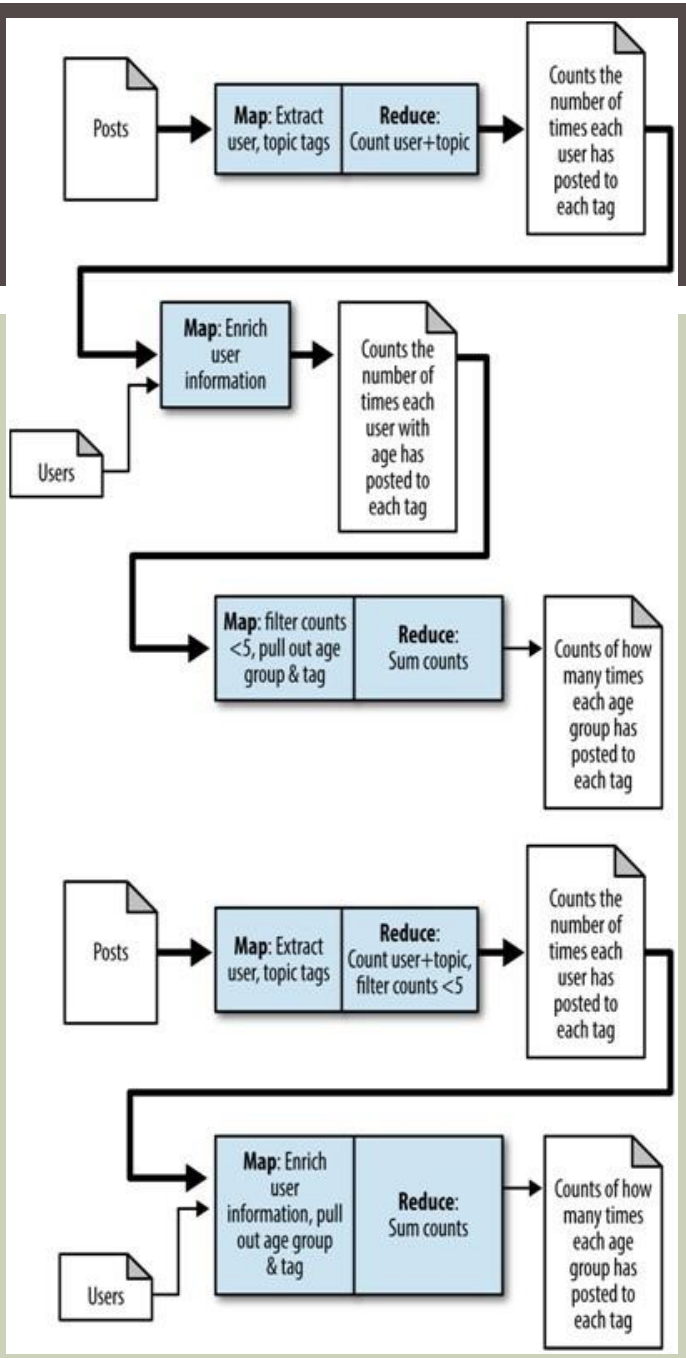
# METAPATTERNS
## JOB CHAINING: CHAIN FOLDING

- If a job ends with a map phase, push that phase into the previous reducer



# METAPATTERNS
## JOB CHAINING: CHAIN FOLDING

- Split up map operations that *decrease* the amount of data from those that *increase* the amount of data


Posts
Map: Extract user, topic tags
Reduce: Count user+topic
Counts the number of times each user has posted to each tag
Map: Enrich user information
Users
Counts the number of times each user with age has posted to each tag
Map: filter counts <5, pull out age group & tag
Reduce: Sum counts
Counts of how many times each age group has posted to each tag
Posts
Map: Extract user, topic tags
Reduce: Count user+topic, filter counts <5
Counts the number of times each user has posted to each tag
Map: Enrich user information, pull out age group & tag
Users
Reduce: Sum counts
Counts of how many times each age group has posted to each tag

# METAPATTERNS
## JOB MERGING

- Allows unrelated jobs loading the same data to share the MapReduce pipeline