MapReduce Summarization Patterns

- Your data is large and vast, with more data coming into the system every day.
- This lecture focuses on design patterns that produce a top-level, summarized view of your data so you can glean insights not available from looking at a localized set of records alone.
- Summarization analytics are all about grouping similar data together and then performing an operation such as calculating a statistic, building an index, or just simply counting.
- Calculating some sort of aggregate over groups in your data set is a great way to easily extract value right away.
  - For example, you might want to calculate the total amount of money your stores have made by state or the average amount of time someone spends logged into your website by demographic.
- Typically, with a new data set, you will start with these types of analyses to help you gauge what is interesting or unique in your data and what needs a closer look.

Summarization Patterns

- The patterns in this lecture are:
  - numerical summarizations
  - inverted index
  - counting with counters

- They are more straightforward applications of MapReduce than some of the other patterns to be discussed later.

- This is because grouping data together by a key is the core function of the MapReduce paradigm:
  - all of the keys are grouped together and collected in the reducers.

- If you emit the fields in the mapper you want to group on as your key, the grouping is all handled by the MapReduce framework for free.

Pattern Description

- The numerical summarizations pattern is a general pattern for calculating aggregate statistical values over your data is discussed in detail.
- Be careful of how deceptively simple this pattern is.
- It is extremely important to use the combiner properly and to understand the calculation you are performing.

Intent

- Group records together by a key field and calculate a numerical aggregate per group to get a top-level view of the larger data set.
- Consider θ to be a generic numerical summarization function we wish to execute over some list of values *(v1, v2, v3, …, vn)* to find a value λ, i.e. $\lambda = \theta(v1, v2, v3, \ldots, vn)$.
- Examples of θ include a minimum, maximum, average, median, and standard deviation.

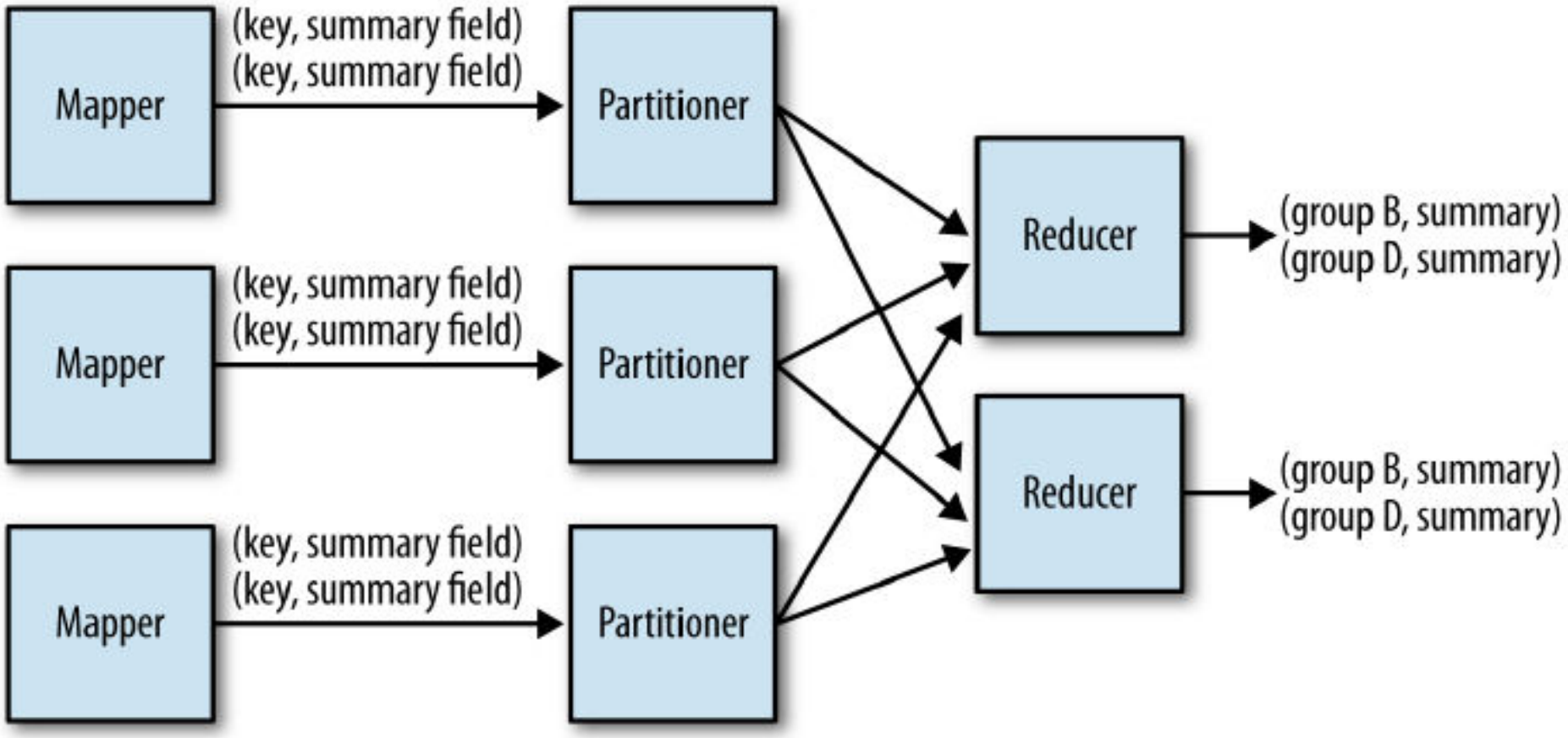
Motivation

- Many data sets these days are too large for a human to get any real meaning out it by reading through it manually.
- For example, if your website logs each time a user logs onto the website, enters a query, or performs any other notable action, it would be extremely difficult to notice any real usage patterns just by reading through terabytes of log files with a text reader.
- If you group logins by the hour of the day and perform a count of the number of records in each group, you can plot these counts on a histogram and recognize times when your website is more active.
- Similarly, if you group advertisements by types, you can determine how affective your ads are for better targeting.
- Maybe you want to cycle ads based on how effective they are at the time of day
- All of these types of questions can be answered through numerical summarizations to get a top-level view of your data.

Applicability

- Numerical summarizations should be used when both of the following are true:
  - You are dealing with numerical data or counting.
  - The data can be grouped by specific fields.

Structure of the Numerical Summarizations Pattern



MAPPER

- The mapper outputs keys that consist of each field to group by, and values consisting of any pertinent numerical items.
- Imagine the mapper setting up a relational table, where the columns relate to the fields which the function θ will be executed over and each row contains an individual record output from the mapper.
- The output value of the mapper contains the values of each column and the output key determines the table as a whole, as each table is created by MapReduce's grouping functionality.

- *Grouping typically involves sending a large subset of the input data down to finally be reduced.*
  - *Each input record is most likely going to be output from the map phase.*
  - *Make sure to reduce the amount of data being sent to the reducers by choosing only the fields that are necessary to the analytic and handling any bad input conditions properly.*

COMBINER

- The combiner can greatly reduce the number of intermediate key/value pairs to be sent across the network to the reducers for some numerical summarization functions.
- If the function θ is an associative and commutative operation, it can be used for this purpose.
- That is, if you can arbitrarily change the order of the values and you can group the computation arbitrarily, you can use a combiner here.

PARTITIONER

- Numerical summaries can benefit from a custom partitioner to better distribute key/value pairs across n number of reduce tasks.
- The need for this is rare, but can be done if job execution time is critical, the amount of data is huge, and there is severe data skew.
- *A custom partitioner is often overlooked, but taking the time to understand the distribution of output keys and partitioning based on this distribution will improve performance when grouping (and everything else, for that matter).*
- *Starting a hundred reduce tasks, only to have eighty of them complete in thirty seconds and the others in twenty-five minutes, is not efficient.*

REDUCER

- The reducer receives a set of numerical values (v1, v2, v3, …, vn) associated with a group-by key records to perform the function $\lambda = \theta(v1, v2, v3, \ldots, vn)$.

- The value of λ is output with the given input key.

OUTCOMES

- The output of the job will be a set of part files containing a single record per reducer input group.
- Each record will consist of the key and all aggregate values.

Known Uses

- **Word count**
  - The “Hello World” of MapReduce. The application outputs each word of a document as the key and “1” as the value, thus grouping by words.
  - The reduce phase then adds up the integers and outputs each unique word with the sum.
- **Record count**
  - A very common analytic to get a heartbeat of your data flow rate on a particular interval
    - weekly, daily, hourly, etc.
- **Min/Max/Count**
  - An analytic to determine the minimum, maximum, and count of a particular event,
    - such as the first time a user posted
    - the last time a user posted
    - the number of times they posted in between that time period.
  - You don’t have to collect all three of these aggregates at the same time, or any of the other use cases listed here if you are only interested in one of them.
- **Average/Median/Standard deviation**
  - Similar to Min/Max/Count, but not as straightforward of an implementation because these operations are not associative.
  - A combiner can be used for all three, but requires a more complex approach than just reusing the reducer implementation.

Resemblances

SQL

The Numerical Aggregation pattern is analogous to using aggregates after a `GROUP BY` in SQL:

```
SELECT MIN(numericalcol1), MAX(numericalcol1),
       COUNT(*) FROM table GROUP BY groupcol2;
```

Pig

The `GROUP ... BY` expression, followed by a `FOREACH ... GENERATE`:

```
b = GROUP a BY groupcol2;
c = FOREACH b GENERATE group, MIN(a.numericalcol1),
       MAX(a.numericalcol1), COUNT_STAR(a);
```

Performance Analysis

- Aggregations performed by jobs using this pattern typically perform well when the combiner is properly used.
- These types of operations are what MapReduce was built for.
- Like most of the patterns, developers need to be concerned about
  - the appropriate number of reducers
  - any data skew that may be present in the reduce groups.
- *That is, if there are going to be many more intermediate key/value pairs with a specific key than other keys, one reducer is going to have a lot more work to do than others.*

Numerical Summarization Examples

- **Minimum, Maximum, and Count Example**
- Calculating the minimum, maximum, and count of a given field are all excellent applications of the numerical summarization pattern.

- After a grouping operation, the reducer simply iterates through all the values associated with the group and finds the min and max, as well as counts the number of members in the key grouping.

- Due to the associative and commutative properties, a combiner can be used to vastly cut down on the number of intermediate key/value pairs that need to be shuffled to the reducers.

- If implemented correctly, the code used for your reducer can be identical to that of a combiner.

Problem:

- Given a list of user's comments, determine the first and last time a user commented and the total number of comments from that user.

MinMaxCountTuple Code

- The MinMaxCountTuple is a Writable object that stores three values.
- This class is used as the output value from the mapper.
- While these values can be crammed into a Text object with some delimiter, it is typically a better practice to create a custom Writable.
- Not only is it cleaner, but you won't have to worry about any string parsing when it comes time to grab these values from the reduce phase.

```java
public class MinMaxCountTuple implements Writable {
    private Date min = new Date();
    private Date max = new Date();
    private long count = 0;

    private final static SimpleDateFormat frmt = new SimpleDateFormat(
            "yyyy-MM-dd'T'HH:mm:ss.SSS");

    public Date getMin() {
        return min;
    }

    public void setMin(Date min) {
        this.min = min;
    }

    public Date getMax() {
        return max;
    }

    public void setMax(Date max) {
        this.max = max;
    }

    public long getCount() {
        return count;
    }

    public void setMax(Date max) {
        this.max = max;
    }

    public long getCount() {
        return count;
    }

    public void setCount(long count) {
        this.count = count;
    }

    public void readFields(DataInput in) throws IOException {
        // Read the data out in the order it is written,
        // creating new Date objects from the UNIX timestamp
        min = new Date(in.readLong());
        max = new Date(in.readLong());
        count = in.readLong();
    }

    public void write(DataOutput out) throws IOException {
         // Write the data out in the order it is read,
         // using the UNIX timestamp to represent the Date
         out.writeLong(min.getTime());
         out.writeLong(max.getTime());
         out.writeLong(count);
     }

     public String toString() {
         return frmt.format(min) + "\t" + frmt.format(max) + "\t" + count;
     }
}
```

Mapper Code

- The mapper will preprocess our input values by extracting the attributes from each input record: the creation data and the user identifier.
- The input key is ignored.
- The creation date is parsed into a Java Date object for ease of comparison in the combiner and reducer.
- The output key is the user ID and the value is three columns of our future output:
  - the minimum date
  - the maximum date
  - the number of comments this user has created.
- These three fields are stored in a custom Writable object of type MinMaxCountTuple, which stores the first two columns as Date objects and the final column as a long.
- These names are accurate for the reducer but don't really reflect how the fields are used in the mapper, but we wanted to use the same data type for both the mapper and the reducer.

Mapper Cont'd

- In the mapper, we'll set both min and max to the comment creation date.
- The date is output twice so that we can take advantage of the combiner optimization that is described later.
- The 3rd column will be a count of 1, to indicate that we know this user posted one comment.
- Eventually, all of these counts are going to be summed together and the minimum and maximum date will be determined in the reducer.

```
public static class MinMaxCountMapper extends
 Mapper<Object, Text, Text, MinMaxCountTuple> {

  // Our output key and value Writables
  private Text outUserId = new Text();
  private MinMaxCountTuple outTuple = new MinMaxCountTuple();

  // This object will format the creation date string into a Date object
  private final static SimpleDateFormat frmt =
                      new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS");

  public void map(Object key, Text value, Context context)
          throws IOException, InterruptedException {

      Map<String, String> parsed = transformXmlToMap(value.toString());

      // Grab the "CreationDate" field since it is what we are finding
      // the min and max value of
      String strDate = parsed.get("CreationDate");

      // Grab the “UserID” since it is what we are grouping by
      String userId = parsed.get("UserId");
      // Parse the string into a Date object
      Date creationDate = frmt.parse(strDate);

      // Set the minimum and maximum date values to the creationDate
      outTuple.setMin(creationDate);
      outTuple.setMax(creationDate);

      // Set the comment count to 1
      outTuple.setCount(1);

      // Set our user ID as the output key
      outUserId.set(userId);

      // Write out the hour and the average comment length
      context.write(outUserId, outTuple);
  }
```

Reducer Code

- The reducer iterates through the values to find the minimum and maximum dates, and sums the counts.
- We start by initializing the output result for each input group.
- For each value in this group, if the output result's minimum is not yet set, or the value's minimum is less than result's current minimum, we set the result's minimum to the input value.
- The same logic applies to the maximum, except using a greater than operator.
- Each value's count is added to a running sum, similar to the word count example.
- After determining the minimum and maximum dates from all input values, the final count is set to our output value.
- The input key is then written to the file system along with the output value.

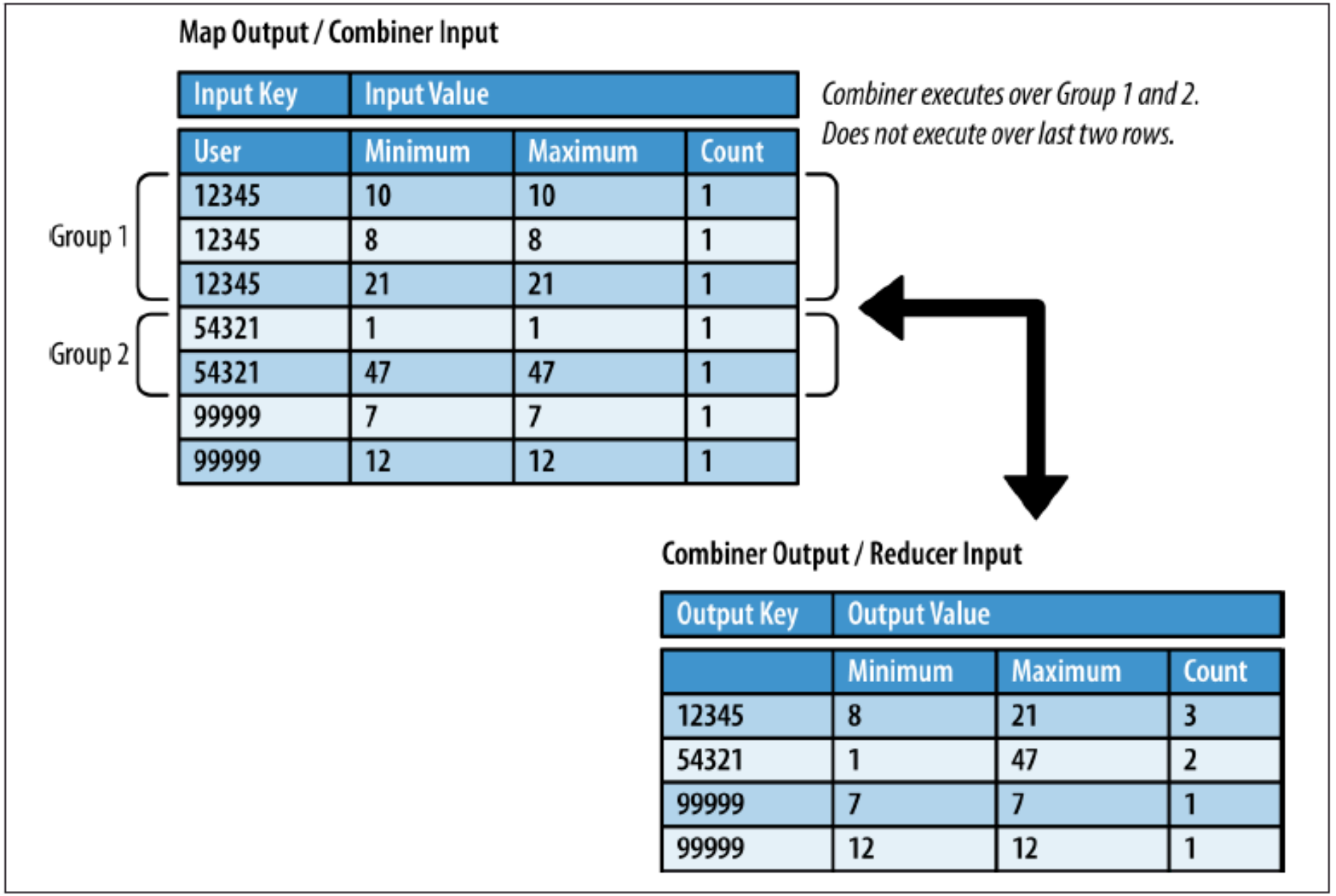
```java
public static class MinMaxCountReducer extends
Reducer<Text, MinMaxCountTuple, Text, MinMaxCountTuple> {

        // Our output value Writable
        private MinMaxCountTuple result = new MinMaxCountTuple();

        public void reduce(Text key, Iterable<MinMaxCountTuple> values,
                Context context) throws IOException, InterruptedException {

            // Initialize our result
            result.setMin(null);
            result.setMax(null);
            result.setCount(0);
            int sum = 0;

            // Iterate through all input values for this key
            for (MinMaxCountTuple val : values) {
                // If the value's min is less than the result's min
                // Set the result's min to value's
                if (result.getMin() == null ||
                    val.getMin().compareTo(result.getMin()) < 0) {
                result.setMin(val.getMin());
                }

                // If the value's max is more than the result's max
                // Set the result's max to value's
                if (result.getMax() == null  ||
                        val.getMax().compareTo(result.getMax()) > 0) {
                result.setMax(val.getMax());
                }

                // Add to our sum the count for value
                sum += val.getCount();
            }

            // Set our count to the number of input values
            result.setCount(sum);
            context.write(key, result);
        }
    }
```

Combiner Optimization

- The reducer implementation just shown can be used as the job's combiner.
- As we are only interested in the count, minimum date, and maximum date, multiple comments from the same user do not have to be sent to the reducer.
- The minimum and maximum comment dates can be calculated for each local map task without having an effect on the final minimum and maximum.
- The counting operation is an associative and commutative operation and won't be harmed by using a combiner.

Data Flow Diagram

- Figure in the next slide shows the flow between the mapper, combiner, and reducer to help describe their interactions.
- Numbers are used rather than dates for simplicity, but the concept is the same.
- A combiner possibly executes over each of the highlighted output groups from a mapper, determining the minimum and maximum values in the first two columns and adding up the number of rows in the “table” (group).
- The combiner then outputs the minimum and maximum along with the new count.
- If a combiner does not execute over any rows, they will still be accounted for in the reduce phase.



The Min/Max/Count MapReduce data flow through the combiner

Average Example

- To calculate an average, we need two values for each group: the sum of the values that we want to average and the number of values that went into the sum.
- These two values can be calculated on the reduce side very trivially, by iterating through each value in the set and adding to a running sum while keeping a count.
- After the iteration, simply divide the sum by the count and output the average.
- However, if we do it this way we cannot use this same reducer implementation as a combiner, because calculating an average is not an associative operation.

- Instead, our mapper will output two "columns" of data, count and average.
- For each input record, this will simply be "1" and the value of the field.
- The reducer will multiply the "count" field by the "average" field to add to a running sum, and add the "count" field to a running count.
- It will then divide the running sum with the running count and output the count with the calculated average.
- With this more round-about algorithm, the reducer code can be used as a combiner as associativity is preserved.

Problem:

- Given a list of user's comments, determine the average comment length per hour of day.

Mapper Code

- The mapper will process each input record to calculate the average comment length based on the time of day.
- The output key is the hour of day, which is parsed from the creation date XML attribute.
- The output value is two columns, the comment count and the average length of the comments for that hour.
- Because the mapper operates on one record at a time, the count is simply 1 and the average length is equivalent to the comment length.
- These two values are output in a custom Writable, a CountAverageTuple.
- This type contains two float values, a count, and an average.

```java
public static class AverageMapper extends
        Mapper<Object, Text, IntWritable, CountAverageTuple> {

    private IntWritable outHour = new IntWritable();
    private CountAverageTuple outCountAverage = new CountAverageTuple();
    private final static SimpleDateFormat frmt = new SimpleDateFormat(
            "yyyy-MM-dd'T'HH:mm:ss.SSS");

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {

        Map<String, String> parsed = transformXmlToMap(value.toString());

        // Grab the "CreationDate" field,
        // since it is what we are grouping by
        String strDate = parsed.get("CreationDate");

        // Grab the comment to find the length
        String text = parsed.get("Text");

        // get the hour this comment was posted in
        Date creationDate = frmt.parse(strDate);
        outHour.set(creationDate.getHours());

        // get the comment length
        outCountAverage.setCount(1);
        outCountAverage.setAverage(text.length());

        // write out the hour with the comment length
        context.write(outHour, outCountAverage);
    }
}
```

Reducer Code

- The reducer code iterates through all given values for the hour and keeps two local variables:
  - a running count
  - running sum
- For each value, the count is multiplied by the average and added to the running sum.
- The count is simply added to the running count.
- After iteration, the input key is written to the file system with the count and average, calculated by dividing the running sum by the running count.

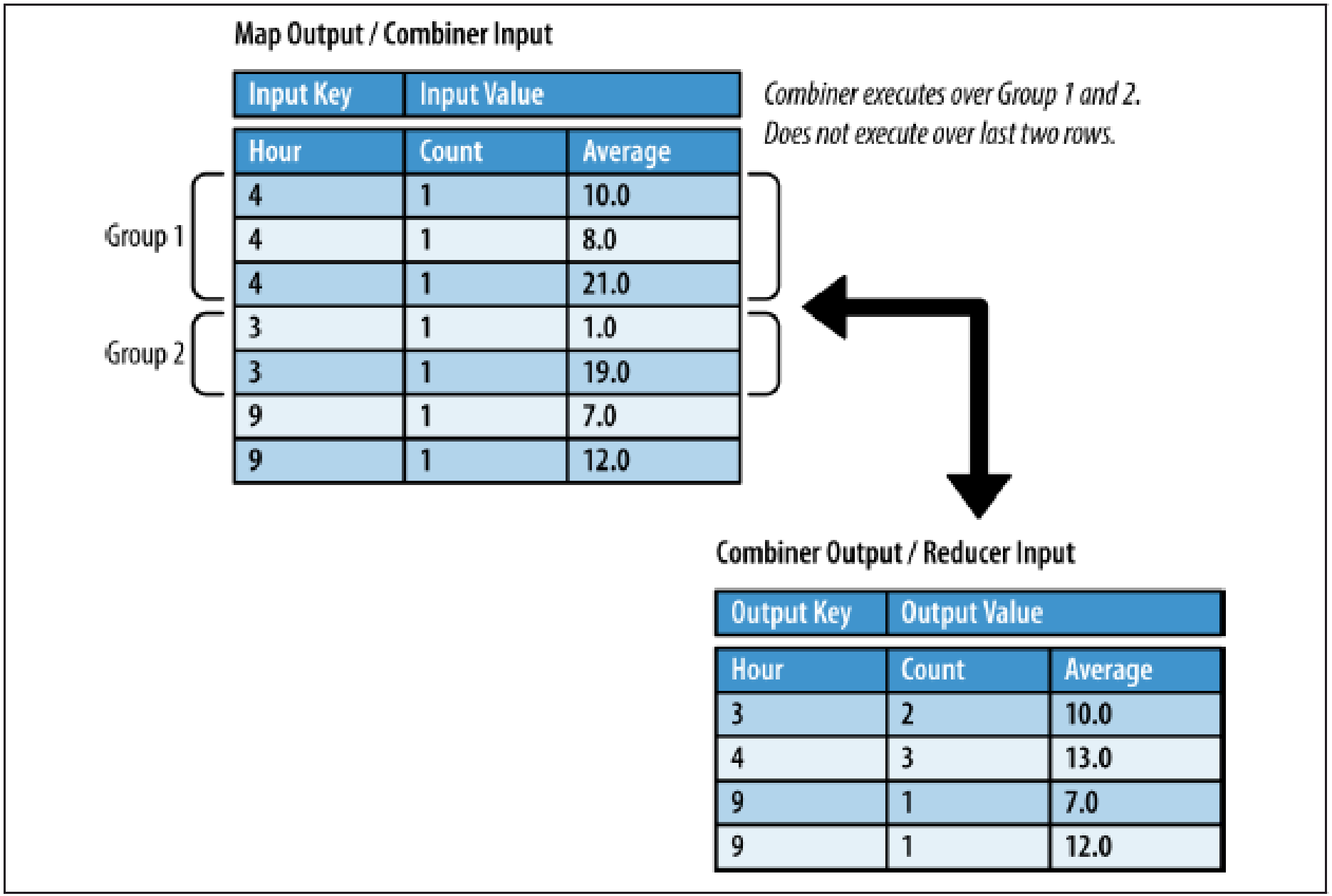
```java
public static class AverageReducer extends
        Reducer<IntWritable, CountAverageTuple,
            IntWritable, CountAverageTuple> {

    private CountAverageTuple result = new CountAverageTuple();

    public void reduce(IntWritable key, Iterable<CountAverageTuple> values,
            Context context) throws IOException, InterruptedException {

        float sum = 0;
        float count = 0;

        // Iterate through all input values for this key
        for (CountAverageTuple val : values) {
            sum += val.getCount() * val.getAverage();
            count += val.getCount();
        }

        result.setCount(count);
        result.setAverage(sum / count);

        context.write(key, result);
    }
}
```

Combiner Optimization

- When determining an average, the reducer code can be used as a combiner when outputting the count along with the average.
- An average is not an associative operation, but if the count is output from the reducer with the count, these two values can be multiplied to preserve the sum for the final reduce phase.
- Without outputting the count, a combiner cannot be used because taking an average of averages is not equivalent to the true average.
- Typically, writing the count along with the average to the file system is not an issue.
- However, if the count is impeding the analysis at hand, it can be omitted by making a combiner implementation nearly identical to the reducer implementation just shown
- The only differentiation between the two classes is that the reducer does not write the count with the average.

Data Flow Diagram

- Figure in the next slide shows the flow between the mapper, combiner, and reducer to help describe their interactions.
- A combiner possibly executes over each of the highlighted output groups from a mapper, determining the average and outputting it with the count, which is the number of rows corresponding to the group.
- If a combiner does not execute over any rows, they will still be accounted for in the reduce phase.



Data flow for the average example

Median and Standard Deviation

- Finding the median and standard deviation is a little more complex than the previous examples.
- Because these operations are not associative, they cannot benefit from a combiner as easily as their counterparts.
- A median is the numerical value separating the lower & higher halves of a data set.
- This requires the data set to be complete, which in turn requires it to be shuffled.
- The data must also be sorted, which can present a barrier because MapReduce does not sort values.
- A standard deviation shows how much variation exists in the data from the average, thus requiring the average to be discovered prior to reduction.

An Easy Solution

- The easiest way to perform these operations involves copying the list of values into a temporary list in order to find the median or iterating over the set again to determine the standard deviation.

- With large data sets, this implementation may result in Java heap space issues, because each value is copied into memory for every input group.

Problem:

- Given a list of user's comments, determine the median and standard deviation of comment lengths per hour of day.

Mapper Code

- The mapper will process each input record to calculate the median comment length within each hour of the day.
- The output key is the hour of day, which is parsed from the CreationDate XML attribute.
- The output value is a single value:
  - the comment length.

```
public static class MedianStdDevMapper extends
        Mapper<Object, Text, IntWritable, IntWritable> {

    private IntWritable outHour = new IntWritable();
    private IntWritable outCommentLength = new IntWritable();

    private final static SimpleDateFormat frmt = new SimpleDateFormat(
            "yyyy-MM-dd'T'HH:mm:ss.SSS");

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {

        Map<String, String> parsed = transformXmlToMap(value.toString());

        // Grab the "CreationDate" field,
        // since it is what we are grouping by
        String strDate = parsed.get("CreationDate");

        // Grab the comment to find the length
        String text = parsed.get("Text");

        // get the hour this comment was posted in
        Date creationDate = frmt.parse(strDate);
        outHour.set(creationDate.getHours());

        // set the comment length
        outCommentLength.set(text.length());

        // write out the user ID with min max dates and count
        context.write(outHour, outCommentLength);
    }
}
```

Reducer Code

- The reducer code iterates through the given set of values and adds each value to an in-memory list.
- The iteration also calculates a running sum and count.
- After iteration, the comment lengths are sorted to find the median value.
- If the list has an odd number of entries, the median value is set to the middle value.
- If the number is even, the middle two values are averaged.
- Next, the standard deviation is calculated by iterating through our sorted list after finding the mean from our running sum and count.
- A running sum of deviations is calculated by squaring the difference between each comment length and the mean.
- The standard deviation is then calculated from this sum.
- Finally, the median and standard deviation are output along with the input key.

```java
public static class MedianStdDevReducer extends
        Reducer<IntWritable, IntWritable,
            IntWritable, MedianStdDevTuple> {

    private MedianStdDevTuple result = new MedianStdDevTuple();
    private ArrayList<Float> commentLengths = new ArrayList<Float>();

    public void reduce(IntWritable key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {

        float sum = 0;
        float count = 0;
        commentLengths.clear();
        result.setStdDev(0);

        // Iterate through all input values for this key
        for (IntWritable val : values) {
            commentLengths.add((float) val.get());
            sum += val.get();
            ++count;
        }

        // sort commentLengths to calculate median
        Collections.sort(commentLengths);

        // if commentLengths is an even value, average middle two elements
        if (count % 2 == 0) {
            result.setMedian((commentLengths.get((int) count / 2 - 1) +
                    commentLengths.get((int) count / 2)) / 2.0f);
        } else {
            // else, set median to middle value
            result.setMedian(commentLengths.get((int) count / 2));
        }

        // calculate standard deviation
        float mean = sum / count;
        float sumOfSquares = 0.0f;
        for (Float f : commentLengths) {
            sumOfSquares += (f - mean) * (f - mean);
        }

        result.setStdDev((float) Math.sqrt(sumOfSquares / (count - 1)));
        context.write(key, result);
    }
}
```

Combiner Optimization

- A combiner cannot be used in this implementation.
- The reducer requires all the values associated with a key in order to find the median and standard deviation.
- Because a combiner runs only over a map's locally output intermediate key/value pairs, being able to calculate the full median and standard deviation is impossible.
- However, the next example describes a more complex implementation that uses a custom combiner.

Memory-Conscious Median and Standard Deviation

- The following implementation is differentiated from the previous median and standard deviation example by reducing the memory footprint.
- Inserting every value into a list will result in many duplicate elements.
- One way to get around this duplication is to keep a count of elements instead.
- For instance, instead of keeping a list of < 1, 1, 1, 1, 2, 2, 3, 4, 5, 5, 5 >, a sorted map of values to counts is kept: (1→4, 2→2, 3→1, 4→1, 5→3).
- The core concept is the same:
  - all the values are iterated through in the reduce phase and stored in an in-memory data structure.
- The data structure and how it is searched are all that has changed.

A Map Reduces the Memory Footprint Drastically

- Instead of having a list whose scaling is ***O(n)*** *where n = number of comments*, the number of key/value pairs in our map is ***O(max(m))***, *m = maximum comment length*

- As an added bonus, a combiner can be used to help aggregate counts of comment lengths and output the map in a Writable object to be used later by the reducer.

Problem:

- Given a list of user's comments, determine the median and standard deviation of comment lengths per hour of day.

Mapper Code

- The mapper processes each input record to calculate the median comment length based on the hour of the day during which the comment was posted.
- The output key is the hour of day, which is parsed from the creation date XML attribute.
- The output value is a SortedMapWritable object that contains one element:
  - the comment length and a count of “1”.
- This map is used more heavily in the combiner and reducer.

```java
public static class MedianStdDevMapper extends
        Mapper<lObject, Text, IntWritable, SortedMapWritable> {

    private IntWritable commentLength = new IntWritable();
    private static final LongWritable ONE = new LongWritable(1);
    private IntWritable outHour = new IntWritable();

    private final static SimpleDateFormat frmt = new SimpleDateFormat(
        "yyyy-MM-dd'T'HH:mm:ss.SSS");

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {

        Map<String, String> parsed = transformXmlToMap(value.toString());

        // Grab the "CreationDate" field,
        // since it is what we are grouping by
        String strDate = parsed.get("CreationDate");

        // Grab the comment to find the length
        String text = parsed.get("Text");

        // Get the hour this comment was posted in
        Date creationDate = frmt.parse(strDate);
        outHour.set(creationDate.getHours());

        commentLength.set(text.length());
        SortedMapWritable outCommentLength = new SortedMapWritable();
        outCommentLength.put(commentLength, ONE);

        // Write out the user ID with min max dates and count
        context.write(outHour, outCommentLength);
    }
}
```

Reducer Code

- The reducer code iterates through the given set of SortedMapWritable to aggregate all the maps together into a single TreeMap, which is a implementation of SortedMap.
- The key is the comment length and the value is the total count associated with the comment length.
- After iteration, the median is calculated.
- The code finds the list index where the median would be by dividing the total number of comments by two.
- The entry set of the TreeMap is then iterated to find the keys that satisfy the condition $previousCommentCount \leq medianIndex < commentCount$ adding the value of the tree map to comments at each step of the iteration.
- Once this condition is met, if there is an even number of comments and medianIndex is equivalent to previousComment, the median is reset to the average of the previous length and current length.
- Otherwise, the median is simply the current comment length.

Standard Deviation

- Next, the standard deviation is calculated by iterating through the TreeMap again and finding the sum of squares, making sure to multiply by the count associated with the comment length.
- The standard deviation is then calculated from this sum.
- The median and standard deviation are output with the input key, the hour during which these comments were posted.

```
public static class MedianStdDevReducer extends
        Reducer<IntWritable, SortedMapWritable,
            IntWritable, MedianStdDevTuple> {

    private MedianStdDevTuple result = new MedianStdDevTuple();
    private TreeMap<Integer, Long> commentLengthCounts =
            new TreeMap<Integer, Long>();

    public void reduce(IntWritable key, Iterable<SortedMapWritable> values,
            Context context) throws IOException, InterruptedException {
        float sum = 0;
        long totalComments = 0;
        commentLengthCounts.clear();
        result.setMedian(0);
        result.setStdDev(0);

        for (SortedMapWritable v : values) {
            for (Entry<WritableComparable, Writable> entry : v.entrySet()) {
                int length = ((IntWritable) entry.getKey()).get();
                long count = ((LongWritable) entry.getValue()).get();

                totalComments += count;
                sum += length * count;

                Long storedCount = commentLengthCounts.get(length);
                if (storedCount == null) {
                    commentLengthCounts.put(length, count);
                } else {
                    commentLengthCounts.put(length, storedCount + count);
                }
            }
        }
        long medianIndex = totalComments / 2L;
        long previousComments = 0;
        long comments = 0;
        int prevKey = 0;
        for (Entry<Integer, Long> entry : commentLengthCounts.entrySet()) {
            comments = previousComments + entry.getValue();

            if (previousComments ≤ medianIndex && medianIndex < comments) {
                if (totalComments % 2 == 0 && previousComments == medianIndex) {
                    result.setMedian((float) (entry.getKey() + prevKey) / 2.0f);
                } else {
                    result.setMedian(entry.getKey());
                }
                break;
            }

            previousComments = comments;
            prevKey = entry.getKey();
        }

        // calculate standard deviation
        float mean = sum / totalComments;

        float sumOfSquares = 0.0f;
        for (Entry<Integer, Long> entry : commentLengthCounts.entrySet()) {
            sumOfSquares += (entry.getKey() - mean) * (entry.getKey() - mean) *
                    entry.getValue();
        }
        result.setStdDev((float) Math.sqrt(sumOfSquares / (totalComments - 1)));
        context.write(key, result);
    }
}
```

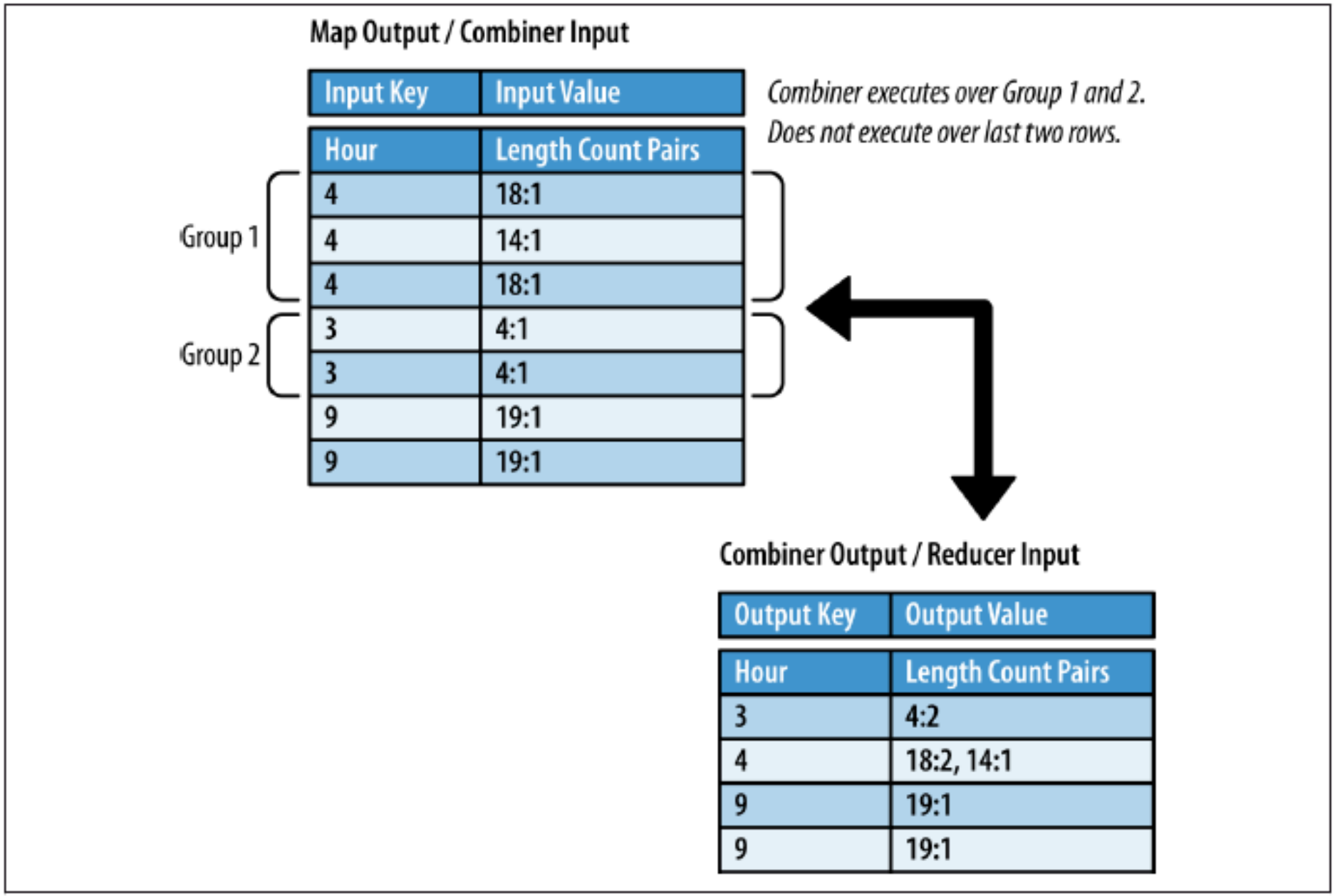
Combiner Optimization

- Unlike the previous examples, the combiner for this algorithm is different from the reducer.
- While the reducer actually calculates the median and standard deviation, the combiner aggregates the SortedMapWritable entries for each local map's intermediate key/value pairs.
- The code to parse through the entries and aggregate them in a local map is identical to the reducer code in the previous section.
- Here, a HashMap is used instead of a TreeMap, because sorting is unnecessary and a HashMap is typically faster.
- While the reducer uses this map to calculate the median and standard deviation, the combiner uses a SortedMapWritable in order to serialize it for the reduce phase

```
public static class MedianStdDevCombiner extends
        Reducer<IntWritable, SortedMapWritable, IntWritable, SortedMapWritable> {

    protected void reduce(IntWritable key,
            Iterable<SortedMapWritable> values, Context context)
            throws IOException, InterruptedException {

        SortedMapWritable outValue = new SortedMapWritable();

        for (SortedMapWritable v : values) {
            for (Entry<WritableComparable, Writable> entry : v.entrySet()) {
                LongWritable count = (LongWritable) outValue.get(entry.getKey());

                if (count != null) {
                    count.set(count.get()
                            + ((LongWritable) entry.getValue()).get());
                } else {
                    outValue.put(entry.getKey(), new LongWritable(
                            ((LongWritable) entry.getValue()).get()));
                }
            }
            v.clear();
        }

        context.write(key, outValue);
    }
}
```

Data Flow Diagram

- Figure in the next slide shows the flow between the mapper, combiner, and reducer to help describe their interactions.
- A combiner possibly executes over each of the highlighted output groups from a mapper.
- For each group, it builds the internal map of comment length to the count of comment lengths.
- The combiner then outputs the input key and the SortedMapWritable of length/count pairs, which it serializes from the map.



Data flow for the standard deviation example

Inverted Index Summarizations

- Pattern Description
  - The inverted index pattern is commonly used as an example for MapReduce analytics.
  - We're going to discuss the general case where we want to build a map of some term to a list of identifiers.

Intent

- Generate an index from a data set to allow for faster searches or data enrichment capabilities.

Motivation

- It is often convenient to index large data sets on keywords, so that searches can trace terms back to records that contain specific values.

- While building an inverted index does require extra processing up front, taking the time to do so can greatly reduce the amount of time it takes to find something.

Search Engines Build Indexes to Improve Search Performance

- Imagine entering a keyword and letting the engine crawl the Internet and build a list of pages to return to you.
- Such a query would take an extremely long amount of time to complete.
- By building an inverted index, the search engine knows all the web pages related to a keyword ahead of time and these results are simply displayed to the user.
- These indexes are often ingested into a database for fast query responses.
- Building an inverted index is a fairly straightforward application of MapReduce because the framework handles a majority of the work.

Applicability

- Inverted indexes should be used when quick search query responses are required.
- The results of such a query can be preprocessed and ingested into a database.

Structure

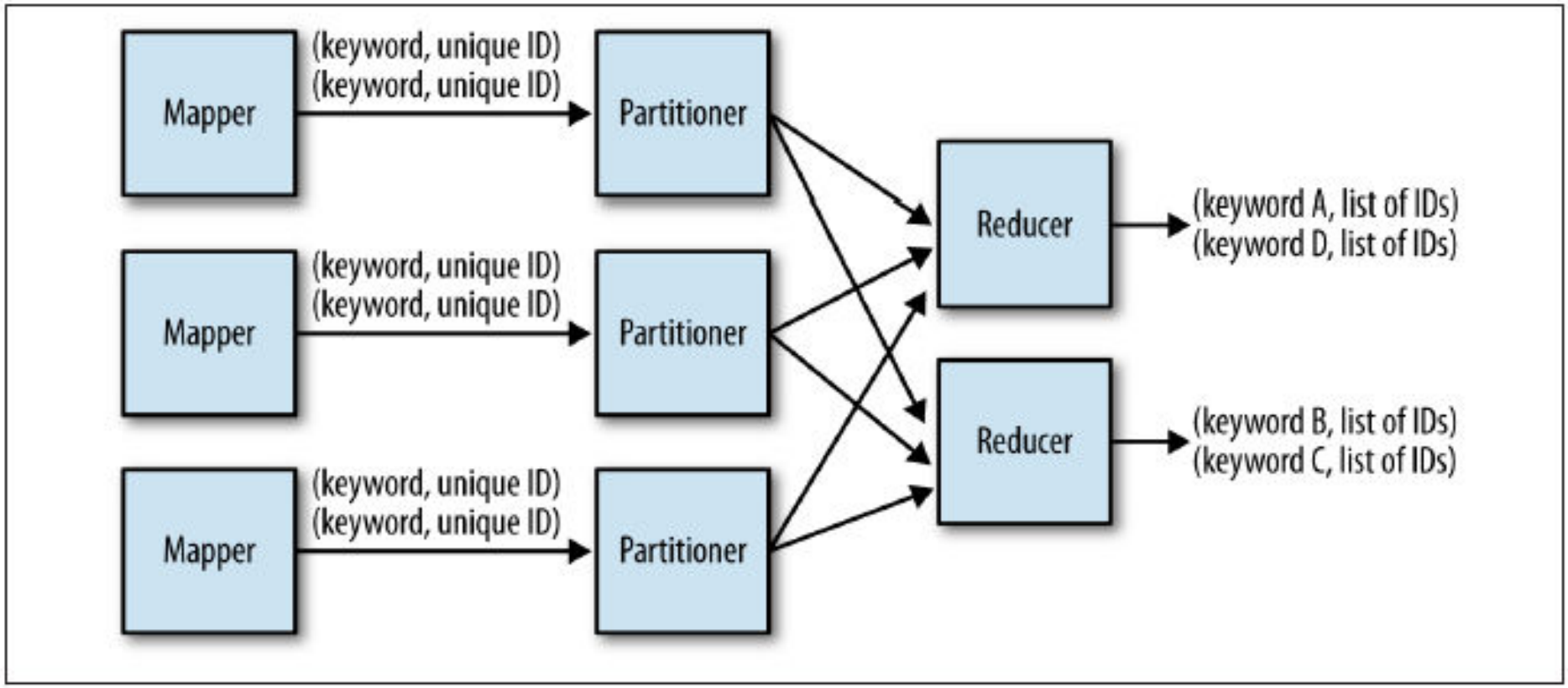
- The mapper outputs the desired fields for the index as the key and the unique identifier as the value.

- The combiner can be omitted if you are just using the identity reducer, because under those circumstances a combiner would just create unnecessary processing.
  - Some implementations concatenate the values associated with a group before outputting them to the file system.
  - In this case, a combiner can be used.
  - It won't have as beneficial an impact on byte count as the combiners in other patterns, but there will be an improvement.

Structure Cont'd

- The partitioner is responsible for determining where values with the same key will eventually be copied by a reducer for final output.
  - It can be customized for more efficient load balancing if the intermediate keys are not evenly distributed.

- The reducer will receive a set of unique record identifiers to map back to the input key.
  - The identifiers can either be concatenated by some unique delimiter, leading to the output of one key/value pair per group, or each input value can be written with the input key, known as the identity reducer.



The structure of the inverted index pattern

Outcomes

- The final output of is a set of part files that contain a mapping of field value to a set of unique IDs of records containing the associated field value.

Performance Analysis

- The performance of building an inverted index depends mostly on the computational cost of parsing
  - the content in the mapper
  - the cardinality of the index keys
  - the number of content identifiers per key

- Parsing text or other types of content in the mapper can sometimes be the most computationally intense operation in a MapReduce job.

- This is especially true for semistructured data, such as XML or JSON, since these typically require parsing arbitrary quantities of information into usable objects.

- It's important to parse the incoming records as efficiently as possible to improve your overall job performance.

Performance Analysis Cont'd

- If the number of unique keys and the number of identifiers is large, more data will be sent to the reducers.
- If more data is going to the reducers, you should increase the number of reducers to increase parallelism during the reduce phase.
- Inverted indexes are particularly susceptible to hot spots in the index keys, since the index keys are rarely evenly distributed.
- For example, the reducer that handles the word "the" in a text search application is going to be particularly busy since "the" is seen in so much text.
- This can slow down your entire job since a few reducers will take much longer than the others.
- To avoid this problem, you might need to implement a custom partitioner, or omit common index keys that add no value to your end goal.

Inverted Index Example - Wikipedia Reference Inverted Index

- Building an inverted index is a straightforward MapReduce application and is often the second example newcomers to MapReduce experience after the word count application.
- Much like the word count application, the bulk of the operation is a group and is therefore handled entirely by the MapReduce framework.
- Suppose we want to add StackOverflow links to each Wikipedia page that is referenced in a StackOverflow comment.
- The following example analyzes each comment in Stack-Overflow to find hyperlinks to Wikipedia.
- If there is one, the link is output with the comment ID to generate the inverted index.
- When it comes to the reduce phase, all the comment IDs that reference the same hyperlink will be grouped together.
- These groups are then concatenated together into a white space delimited String and directly output to the file system.
- From here, this data file can be used to update the Wikipedia page with all the comments that reference it.

Problem:

- Given a set of user's comments, build an inverted index of Wikipedia URLs to a set of answer post IDs .

Mapper Code

- The mapper parses the posts from StackOverflow to output the row IDs of all answer posts that contain a particular Wikipedia URL.
- First, the XML attributes for the text, post type, and row ID are extracted.
- If the post type is not an answer, identified by a post type of “2”, we parse the text to find a Wikipedia URL.
- This is done using the getWikipediaURL method, which takes in a String of unescaped HTML and returns a Wikipedia URL if found, or null otherwise.
- The method is omitted for brevity.
- If a URL is found, the URL is output as the key and the rowID is output as the value

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

    private Text link = new Text();
    private Text outkey = new Text();

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {

        Map<String, String> parsed = MRDPUtils.transformXmlToMap(value
                .toString());

        // Grab the necessary XML attributes
        String txt = parsed.get("Body");
        String posttype = parsed.get("PostTypeId");
        String row_id = parsed.get("Id");

        // if the body is null, or the post is a question (1), skip
        if (txt == null || (posttype != null && posttype.equals("1"))) {
            return;
        }

        // Unescape the HTML because the SO data is escaped.
        txt = StringEscapeUtils.unescapeHtml(txt.toLowerCase());

        link.set(getWikipediaURL(txt));
        outkey.set(row_id);
        context.write(link, outkey);
    }
}
```

Reducer Code

- The reducer iterates through the set of input values and appends each row ID to a String, delimited by a space character.
- The input key is output along with this concatenation.

```java
public static class Concatenator extends Reducer<Text,Text,Text,Text> {
    private Text result = new Text();

    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {

        StringBuilder sb = new StringBuilder();
        boolean first = true;
        for (Text id : values) {
            if (first) {
                first = false;
            } else {
                sb.append(" ");
            }
            sb.append(id.toString());
        }

        result.set(sb.toString());
        context.write(key, result);
    }
}
```

Combiner Optimization

- The combiner can be used to do some concatenation prior to the reduce phase.
- Because all row IDs are simply concatenated together, the number of bytes that need to be copied by the reducer is more than in a numerical summarization pattern.
- The same code for the reducer class is used as the combiner.

Counting with Counters

- Pattern Description
  - This pattern utilizes the MapReduce framework's counters utility to calculate a global sum entirely on the map side without producing any output.

Intent

- An efficient means to retrieve count summarizations of large data sets.

Motivation

- A count or summation can tell you a lot about particular fields of data, or your data as a whole.
- Hourly ingest record counts can be post processed to generate helpful histograms.
- This can be executed in a simple “word count” manner, in that for each input record, you output the same key, say the hour of data being processed, and a count of 1.
- The single reduce will sum all the input values and output the final record count with the hour.
- This works very well, but it can be done more efficiently using counters. Instead of writing any key value pairs at all, simply use the framework’s counting mechanism to keep track of the number of input records.
- This requires no reduce phase and no summation!
- The framework handles monitoring the names of the counters and their associated values, aggregating them across all tasks, as well as taking into account any failed task attempts.

Motivation Cont'd

- Say you want to find the number of times your employees log into your heavily used public website every day.
- Assuming you have a few dozen employees, you can apply filter conditions while parsing through your web logs.
- Rather than outputting the employee's user name with a count of '1', you can simply create a counter with the employee's ID and increment it by 1.
- At the end of the job, simply grab the counters from the framework and save them wherever your heart desires—the log, local file system, HDFS, etc.

Motivation Cont'd

- Some counters come built into the framework, such as number of input/output records and bytes.
- Hadoop allows for programmers to create their own custom counters for whatever their needs may be.
- This pattern describes how to utilize these custom counters to gather count or summation metrics from your data sets.
- The major benefit of using counters is all the counting can be done during the map phase.

The Caveat to Using Counters

- The caveat to using counters is they are all stored in-memory by the JobTracker.
- The counters are serialized by each map task and sent with status updates.
- In order to play nice and not bog down the JobTracker, the number of counters should be in the tens -- a hundred at most... And thats a big “at most”!
- Counters are definitely not meant to aggregate lots of statistics about your MapReduce job!
- Newer versions of Hadoop actually limit the number of counters a job can create to prevent any permanent damage to the JobTracker.
- The last thing you want is to have your analytic take down the JobTracker because you created a few hundred custom counters!

Applicability

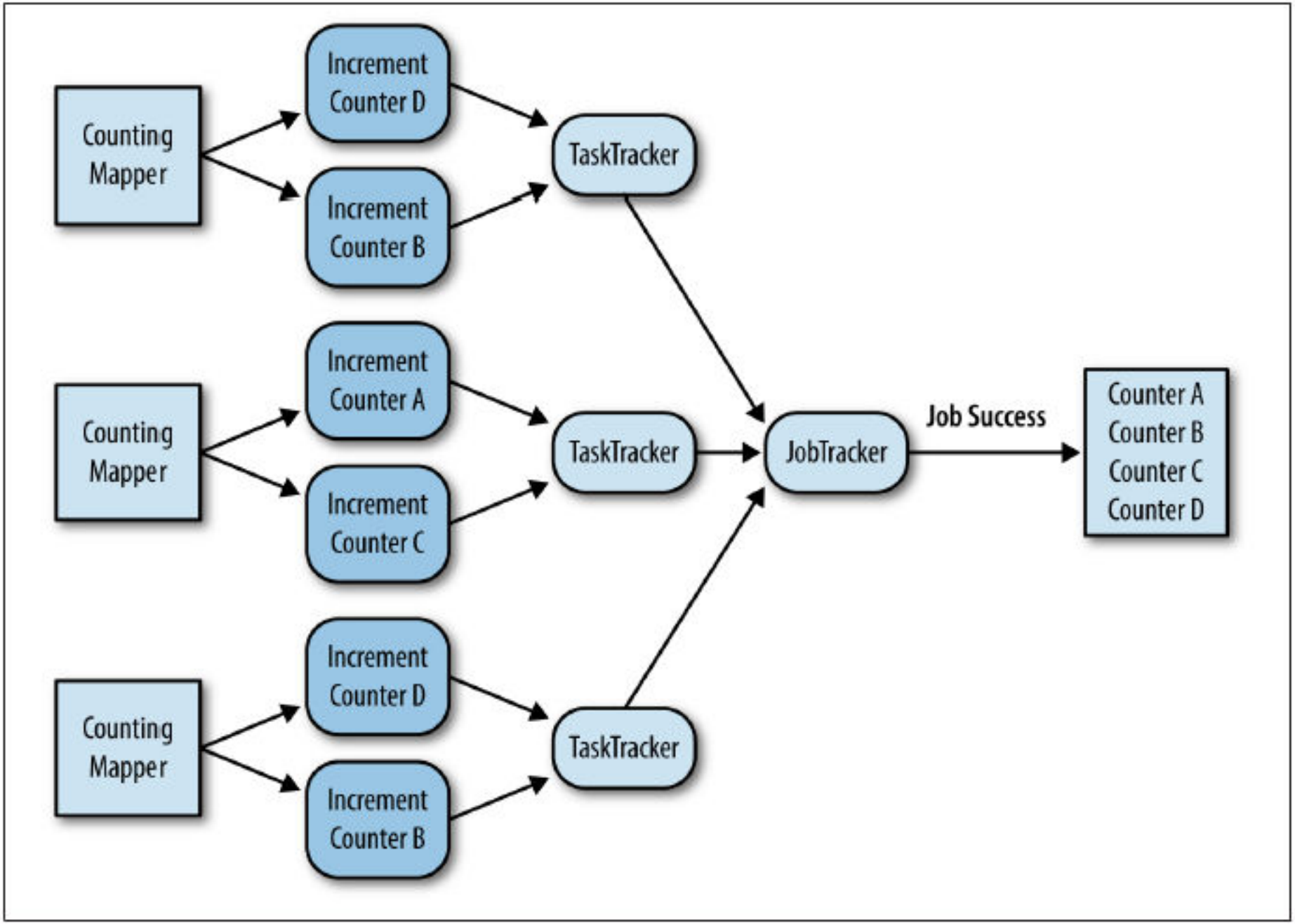
- Counting with counters should be used when:
  - You have a desire to gather counts or summations over large data sets.
  - The number of counters you are going to create is small—in the double digits.

Structure

- Figure in the next slide shows the general structure of how this pattern works in MapReduce.
- The mapper processes each input record at a time to increment counters based on certain criteria.
- The counter is either incremented by one if counting a single instance, or incremented by some number if executing a summation.
  - These counters are then aggregated by the TaskTrackers running the tasks and incrementally reported to the JobTracker for overall aggregation upon job success.
  - The counters from any failed tasks are disregarded by the JobTracker in the final summation.
- As this job is map only, there is no combiner, partitioner, or reducer required.

Outcomes

- The final output is a set of counters grabbed from the job framework.
- There is no actual output from the analytic itself. However, the job requires an output directory to execute.
- This directory will exist and contain a number of empty part files equivalent to the number of map tasks.
- This directory should be deleted on job completion.



The structure of the counting with counters pattern

Known Uses

- Count number of records
  - Simply counting the number of records over a given time period is very common.
  - It's typically a counter provided by the framework, among other common things.
- Count a small number of unique instances
  - Counters can also be created on the fly by using a string variable.
  - You might now know what the value is, but the counters don't have to be created ahead of time.
  - Simply creating a counter using the value of a field and incrementing it is enough to solve this use case.
  - Just be sure the number of counters you are creating is a small number!
- Summations
  - Counters can be used to sum fields of data together. Rather than performing the sum on the reduce side, simply create a new counter and use it to sum the field values.

Performance Analysis

- Using counters is very fast, as data is simply read in through the mapper and no output is written.
- Performance depends largely on the number of map tasks being executed and how much time it takes to process each record.

Counting with Counters Example - Number of Users per State

- For this example, we use a map-only job to count the number of users in each state
- The Location attribute is a user-entered value and doesn't have any concrete inputs
- Because of this, there are a lot of null or empty fields, as well as made up locations.
- We need to account for this when processing each record to ensure we don't create a large number of counters.
- We verify each location contains a state abbreviation code prior to creating a counter
- This will create at most 52 counters - 50 for the states and two for NullOr Empty and Unknown.
- This is a manageable number of custom counters for the JobTracker, but your job should not have many more than this!

Problem:

- Count the number of users from each state using Hadoop custom counters.

Mapper Code

- The mapper reads each user record & gets his or her location.
- The location is split on white space & searched for something that resembles a state
- We keep a set of all the state abbreviations in-memory to prevent creating an excessive amount of counters, as the location is simply a string set by the user and nothing structured.
- If a state is recognized, the counter for the state is incremented by one and the loop is broken.
- Counters are identified by both a group and a name.
- Here, the group is “State” (identified by a public String variable) and the counter name is the state abbreviation code.

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

    public static final String STATE_COUNTER_GROUP = "State";
    public static final String UNKNOWN_COUNTER = "Unknown";
    public static final String NULL_OR_EMPTY_COUNTER = "Null or Empty";

    private String[] statesArray = new String[] { "AL", "AK", "AZ", "AR",
            "CA", "CO", "CT", "DE", "FL", "GA", "HI", "ID", "IL", "IN",
            "IA", "KS", "KY", "LA", "ME", "MD", "MA", "MI", "MN", "MS",
            "MO", "MT", "NE", "NV", "NH", "NJ", "NM", "NY", "NC", "ND",
            "OH", "OK", "OR", "PA", "RI", "SC", "SF", "TN", "TX", "UT",
            "VT", "VA", "WA", "WV", "WI", "WY" };

    private HashSet<String> states = new HashSet<String>(
            Arrays.asList(statesArray));

    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {

        Map<String, String> parsed = MRDPUtils.transformXmlToMap(value
                .toString());

        // Get the value for the Location attribute
        String location = parsed.get("Location");

        // Look for a state abbreviation code if the
        // location is not null or empty
        if (location != null && !location.isEmpty()) {

            // Make location uppercase and split on white space
            String[] tokens = location.toUpperCase().split("\\s");

            // For each token
            boolean unknown = true;
            for (String state : tokens) {

                // Check if it is a state
                if (states.contains(state)) {
                    // If so, increment the state's counter by 1
                    // and flag it as not unknown
                    context.getCounter(STATE_COUNTER_GROUP, state)
                            .increment(1);
                    unknown = false;
                    break;
                }
            }

            // If the state is unknown, increment the UNKNOWN_COUNTER counter
            if (unknown) {
                context.getCounter(STATE_COUNTER_GROUP, UNKNOWN_COUNTER)
                        .increment(1);
            }
        } else {
            // 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);
        }
    }
}
```

Driver Code

- The driver code is mostly boilerplate, with the exception of grabbing the counters after the job completes.
- If the job completed succesfully, we get the “States” counter group and write out the counter name and value to stdout.
- These counter values are also output when the job completes, so writing to stdout may be redundant if you are obtaining these values by scraping log files.
- The output directory is then deleted, success or otherwise, as this job doesn’t create any tangible output.

```
...

int code = job.waitForCompletion(true) ? 0 : 1;

if (code == 0) {
    for (Counter counter : job.getCounters().getGroup(
            CountNumUsersByStateMapper.STATE_COUNTER_GROUP)) {
        System.out.println(counter.getDisplayName() + "\t"
                + counter.getValue());
    }
}

// Clean up empty output directory
FileSystem.get(conf).delete(outputDir, true);

System.exit(code);
```