# MapReduce Data Organization Patterns



- In contrast to the previous lecture on filtering, this lecture is all about reorganizing data.
- The value of individual records is often multipled by the way they are partitioned, sharded, or sorted.
- This is especially true in distributed systems, where partitioning, sharding, and sorting can be exploited for performance.
- In many organizations, Hadoop and other MapReduce solutions are only a piece in the larger data analysis platform.
- Data will typically have to be transformed to interface nicely with the other systems.
- Likewise, data might have to be transformed from its original state to a new state to make analysis in MapReduce easier.

# Data Organization Patterns



- This lecture discusses several pattern subcategories as you will see in each pattern description:
  - The structured to hierarchical pattern
  - The partitioning and binning patterns
  - The total order sorting and shuffling patterns
  - The generating data pattern

- The patterns in this lecture are often used together to solve data organization problems.
  - For example, you may want to restructure your data to be hierarchical, bin the data, and then have the bins be sorted.

# Structured to Hierarchical Pattern



- **Pattern Description:**
  - The structured to hierarchical pattern creates new records from data that started in a very different structure.

- **Intent:**
  - Transform your row-based data to a hierarchical format, such as JSON or XML.

# Motivation



- When migrating data from an RDBMS to a Hadoop system, one of the first things you should consider doing is reformatting your data into a more conducive structure.
- Since Hadoop doesn't care what format your data is in, you should take advantage of hierarchical data to avoid doing joins.

- For example, our StackOverflow data contains a table about comments, a table about posts, etc.
  - It is pretty obvious that the data is stored in an normalized SQL database.
  - When you visit a post on StackOverflow, all the different pieces need to be coalesced into one view.
  - This gets even more complicated when you are trying to do analytics at the level of individual posts.

# Motivation Cont'd



- Imagine trying to correlate the length of the post with the length of the comments.
- This requires you to first do a join, an expensive operation, then extract the data that allows you to do your real work.
- If instead you group the data by post so that the comments are co-located with the posts and the edit revisions (i.e., denormalizing the tables), this type of analysis will be much easier and more intuitive.

# Motivation Cont'd

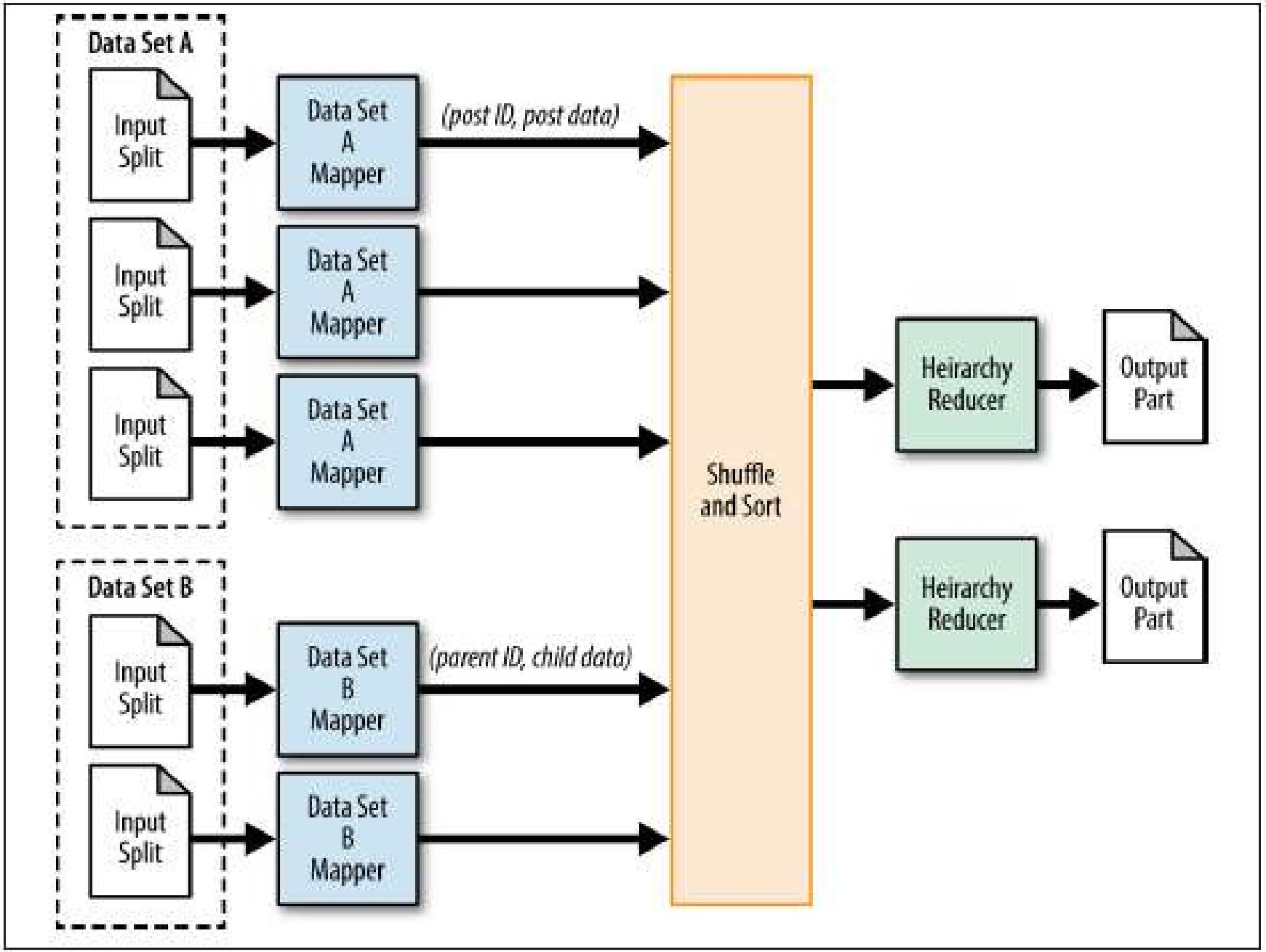


- Keeping the data in a normalized form in this case serves little purpose.
- Unfortunately, data doesn't always come grouped together.
- When someone posts an answer to a StackOverflow question, Hadoop can't insert that record into the hierarchy immediately.
- Therefore, creating the denormalized records for MapReduce has to be done in a batch fashion periodically.
- Another way to deal with a steady stream of updates is HBase.
- HBase is able to store data in a semi-structured and hierarchical fashion well.
- MongoDB would also be a good candidate for storing this type of data.

# Applicability



- The following should be true for this pattern to be appropriate:
  - You have data sources that are linked by some set of foreign keys.
  - Your data is structured and row-based.






*The structure of the structured to hierarchical pattern*

- If you wish to combine multiple data sources into a hierarchical data structure, a Hadoop class called `MultipleInputs` from `org.apache.hadoop.mapreduce.lib.input` is extremely valuable. `MultipleInputs` allows you to specify different input paths and different mapper classes for each input. The configuration is done in the driver. If you are loading data from only one source in this pattern, you don't need this step.
- The mappers load the data and parse the records into one cohesive format so that your work in the reducers is easier. The output key should reflect how you want to identify the root of each hierarchical record. For example, in our StackOverflow example, the root would be the post ID. You also need to give each piece of data some context about its source. You need to identify whether this output record is a post or a comment. To do this, you can simply concatenate some sort of label to the output value text.
- In general, a combiner isn't going to help you too much here. You could hypothetically group items with the same key and send them over together, but this has no major compression gains since all you would be doing is concatenating strings, so the size of the resulting string would be the same as the inputs.
- The reducer receives the data from all the different sources key by key. All of the data for a particular grouping is going to be provided for you in one iterator, so all that is left for you to do is build the hierarchical data structure from the list of data items. With XML or JSON, you'll build a single object and then write it out as output. The examples in this section show XML, which provides several convenient methods for constructing data structures. If you are using some other format, such as a custom format, you'll just need to use the proper object building and serialization methods.

# Outcomes



- The output will be in a hierarchical form, grouped by the key that you specified.
- However, be careful that many formats such as XML and JSON have some sort of top level root element that encompasses all of the records.
- If you actually need the document to be well-formed top-to-bottom, it's usually easier to add this header and footer text as some post-processing step.

# Known uses



- **Pre-joining data**
  - Data arrives in disjointed structured data sets, and for analytical purposes it would be easier to bring the data together into more complex objects.
  - By doing this, you are setting up your data to take advantage of the NoSQL model of analysis.

- **Preparing data for HBase or MongoDB**
  - HBase is a natural way to store this data, so you can use this method to bring the data together in preparation for loading into HBase or MongoDB.
  - Creating a new table and then executing a bulk import via MapReduce is particularly effective.
  - The alternative is to do several rounds of inserts, which might be less efficient.

### Resemblances

*SQL*

It's rare that you would want to do something like this in a relational database, since storing data in this way is not conducive to analysis with SQL. However, the way you would solve a similar problem in an RDBMS is to join the data and then perform analysis on the result.

*Pig*

Pig has reasonable support for hierarchical data structures. You can have hierarchical bags and tuples, which make it easy to represent hierarchical structures and lists of objects in a single record. The `COGROUP` method in Pig does a great job of bringing data together while preserving the original structure. However, using the predefined keywords to do any sort of real analysis on a complex record is more challenging out of the box. For this, a user-defined function is the right way to go. Basically, you would use Pig to build and group the records, then a UDF to make sense of the data.

```
data_a = LOAD '/data/comments/' AS PigStorage('|');
data_b = LOAD '/data/posts/' AS PigStorage(',');

grouped = COGROUP data_a BY $2, data_b BY $1;

analyzed = FOREACH grouped GENERATE udfs.analyze(group, $1, $2);

---
```

# Performance Analysis



- There are 2 performance concerns you should pay attention to when using this pattern.
  - You need to be aware of how much data is being sent to reducers from mappers.
  - You need to be aware of the memory footprint of the object that the reducer builds.
- Since records with the grouping key can be scattered anywhere in the data set, pretty much all of data is going to move across the network.
- For this reason, you will need to pay particular attention to having an adequate number of reducers.
- The same strategies apply here that are employed in other patterns that shuffle everything over the network.

# Performance Analysis Cont'd



- The next major concern is the possibility of hot spots in the data that could result in an obscenely large record.
- With large data sets, it is conceivable that a particular output record is going to have a lot of data associated with it.

- Imagine that for some reason a post on StackOverflow has a million comments associated with it.
  - That would be extremely rare and unlikely, but not in the realm of the impossible.
  - If you are building some sort of XML object, all of those comments at one point might be stored in memory before writing the object out.
  - This can cause you to blow out the heap of the Java Virtual Machine, which obviously should be avoided.

# Performance Analysis Cont'd



- Another problem with hot spots is a skew in how much data each reducer is handling.
- This is going to be a similar problem in just about any MapReduce job.
- In many cases the skew can be ignored, but if it really matters you can write a custom partitioner to split the data up more evenly.

# Structured to Hierarchical Examples



- **Post/comment building on StackOverflow**
  - In this example, we will take the posts and comments of the StackOverflow data and group them together. A hierarchy will look something like:

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

- **Problem:**
  - Given a list of posts and comments, create a structured XML hierarchy to nest comments with their related post.

# Driver Code



- We don't usually describe the code for the driver, but in this case we are doing something exotic with MultipleInputs.
- All we do differently is create a MultipleInputs object and add the comments path and the posts path with their respective mappers.
- The paths for the posts and comments data are provided via the command line, and the program retrieves them from the args array.

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "PostCommentHierarchy");
    job.setJarByClass(PostCommentBuildingDriver.class);

    MultipleInputs.addInputPath(job, new Path(args[0]),
            TextInputFormat.class, PostMapper.class);

    MultipleInputs.addInputPath(job, new Path(args[1]),
            TextInputFormat.class, CommentMapper.class);

    job.setReducerClass(UserJoinReducer.class);

    job.setOutputFormatClass(TextOutputFormat.class);
    TextOutputFormat.setOutputPath(job, new Path(args[2]));

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(Text.class);

    System.exit(job.waitForCompletion(true) ? 0 : 2);
}
```

# Mapper Code



- In this case, there are 2 mappers, one for comments and one for posts.
- In both, we extract the post ID to use it as the output key.
- We output the input value prepended with a character
  ("P" for a post or "C" for a comment)
  so we know which dataset the record came from during the reduce phase

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

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

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

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

        // The foreign join key is the post ID
        outkey.set(parsed.get("Id"));

        // Flag this record for the reducer and then output
        outvalue.set("P" + value.toString());
        context.write(outkey, outvalue);
    }
}
```

```
public static class CommentMapper extends Mapper<Object, Text, Text, Text> {
    private Text outkey = new Text();
    private Text outvalue = new Text();

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

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

        // The foreign join key is the post ID
        outkey.set(parsed.get("PostId"));

        // Flag this record for the reducer and then output
        outvalue.set("C" + value.toString());
        context.write(outkey, outvalue);
    }
}
```

# Reducer Code



- The reducer builds the hierarchical XML object.
- All the values are iterated to get the post record and collect a list of comments.
- We know which record is which by the flag we added to the value.
- These flags are removed when assigning post or adding the list.
- Then, if the post is not null, an XML record is constructed with the post as the parent and comments as the children.
- The implementation of the nestElements follows.
  - We chose to use an XML library to build the final record, but please feel free to use whatever means you deem necessary.

```
public static class PostCommentHierarchyReducer extends
        Reducer<Text, Text, Text, NullWritable> {

    private ArrayList<String> comments = new ArrayList<String>();
    private DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    private String post = null;

    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        // Reset variables
        post = null;
        comments.clear();

        // For each input value
        for (Text t : values) {
            // If this is the post record, store it, minus the flag
            if (t.charAt(0) == 'P') {
                post = t.toString().substring(1, t.toString().length())
                        .trim();
            } else {
                // Else, it is a comment record. Add it to the list, minus
                // the flag
                comments.add(t.toString()
                        .substring(1, t.toString().length()).trim());
            }
        }
        // If there are no comments, the comments list will simply be empty.

        // If post is not null, combine post with its comments.
        if (post != null) {
            // nest the comments underneath the post element
            String postWithCommentChildren = nestElements(post, comments);

            // write out the XML
            context.write(new Text(postWithCommentChildren),
                    NullWritable.get());
        }
    }
```

# NestElements Method



- The nestElements method takes the post and the list of comments to create a new string of XML to output.
- It uses a DocumentBuilder and some additional helper methods to copy the Element objects into new ones, in addition to their attributes.
- This copying occurs to rename the element tags from row to either post or comment.
- The final Document is then transformed into an XML string.

```
    private String nestElements(String post, List<String> comments) {
        // Create the new document to build the XML
        DocumentBuilder bldr = dbf.newDocumentBuilder();
        Document doc = bldr.newDocument();

        // Copy parent node to document
        Element postEl = getXmlElementFromString(post);
        Element toAddPostEl = doc.createElement("post");

        // Copy the attributes of the original post element to the new one
        copyAttributesToElement(postEl.getAttributes(), toAddPostEl);

        // For each comment, copy it to the "post" node
        for (String commentXml : comments) {
            Element commentEl = getXmlElementFromString(commentXml);
            Element toAddCommentEl = doc.createElement("comments");

            // Copy the attributes of the original comment element to
            // the new one
            copyAttributesToElement(commentEl.getAttributes(),
                    toAddCommentEl);

            // Add the copied comment to the post element
            toAddPostEl.appendChild(toAddCommentEl);
        }

        // Add the post element to the document
        doc.appendChild(toAddPostEl);

        // Transform the document into a String of XML and return
        return transformDocumentToString(doc);
    }

    private Element getXmlElementFromString(String xml) {
        // Create a new document builder
        DocumentBuilder bldr = dbf.newDocumentBuilder();

        return bldr.parse(new InputSource(new StringReader(xml)))
                .getDocumentElement();
    }

    private void copyAttributesToElement(NamedNodeMap attributes,
            Element element) {

        // For each attribute, copy it to the element
        for (int i = 0; i < attributes.getLength(); ++i) {
            Attr toCopy = (Attr) attributes.item(i);
            element.setAttribute(toCopy.getName(), toCopy.getValue());
        }
    }

    private String transformDocumentToString(Document doc) {

        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer();
        transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION,
                "yes");
        StringWriter writer = new StringWriter();
        transformer.transform(new DOMSource(doc), new StreamResult(
                writer));
        // Replace all new line characters with an empty string to have
        // one record per line.
        return writer.getBuffer().toString().replaceAll("\n|\r", "");
    }
}
```

# Question/Answer Building on StackOverflow



- This is a continuation of the previous example and will use the previous analytic's output as the input to this analytic.
- Now that we have the comments associated with the posts, we are going to associate the post answers with the post questions.
- This needs to be done because posts consist of both answers and questions and are differentiated only by their PostTypeId.
- We'll group them together by Id in questions and ParentId in answers.
- The main difference between the two applications of this pattern is that in this one we are dealing only with one data set.
- Effectively, we are using a self-join here to correlate the different records from the same data set.

# Problem



- Given the output of the previous example, perform a self-join operation to create a question, answer, and comment hierarchy.

# Mapper Code



- The first thing the mapper code does is determine whether the record is a question or an answer, because the behavior for each will be different.
- For a question, we will extract Id as the key and label it as a question.
- For an answer, we will extract ParentId as the key and label it as an answer.

```
public class QuestionAnswerBuildingDriver {

    public static class PostCommentMapper extends
            Mapper<Object, Text, Text, Text> {

        private DocumentBuilderFactory dbf = DocumentBuilderFactory
                .newInstance();
        private Text outkey = new Text();

        private Text outvalue = new Text();

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

            // Parse the post/comment XML hierarchy into an Element
            Element post = getXmlElementFromString(value.toString());

            int postType = Integer.parseInt(post.getAttribute("PostTypeId"));

            // If postType is 1, it is a question
            if (postType == 1) {
                outkey.set(post.getAttribute("Id"));
                outvalue.set("Q" + value.toString());
            } else {
                // Else, it is an answer
                outkey.set(post.getAttribute("ParentId"));
                outvalue.set("A" + value.toString());
            }

            context.write(outkey, outvalue);
        }

        private Element getXmlElementFromString(String xml) {
            // same as previous example, "Mapper code" (page 80)
        }
    }
```

# Reducer Code



- The reducer code is very similar to the that in the previous example.
- It iterates through the input values and grabs the question and answer, being sure to remove the flag.
- It then nests the answers inside the question in the same fashion as the previous example.
- The difference is that tags are “question” instead of the “post” and “answer” instead of “comment.”
- The helper functions are omitted here for brevity.
- They can be viewed in the previous example.

```
public static class QuestionAnswerReducer extends
        Reducer<Text, Text, Text, NullWritable> {

    private ArrayList<String> answers = new ArrayList<String>();
    private DocumentBuilderFactory dbf = DocumentBuilderFactory
            .newInstance();
    private String question = null;

    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        // Reset variables
        question = null;
        answers.clear();

        // For each input value
        for (Text t : values) {
            // If this is the post record, store it, minus the flag
            if (t.charAt(0) == 'Q') {
                question = t.toString().substring(1, t.toString().length())
                        .trim();
            } else {
                // Else, it is a comment record. Add it to the list, minus
                // the flag
                answers.add(t.toString()
                        .substring(1, t.toString().length()).trim());
            }
        }

        // If post is not null
        if (question != null) {
            // nest the comments underneath the post element
            String postWithCommentChildren = nestElements(question, answers);

            // write out the XML
            context.write(new Text(postWithCommentChildren),
                    NullWritable.get());
        }
    }

    ... // ommitted helper functions
}
```

# Partitioning Pattern



- **Pattern Description**
- The partitioning pattern moves the records into categories (i.e., shards, partitions, or bins) but it doesn't really care about the order of records.

- **Intent:**
- The intent is to take similar records in a data set and partition them into distinct, smaller data sets.

# Motivation



- If you want to look at a particular set of data—such as postings made on a particular date—the data items are normally spread out across the entire data set.

- So looking at just one of these subsets requires an entire scan of all of the data.
- Partitioning means breaking a large set of data into smaller subsets, which can be chosen by some criterion relevant to your analysis.

- To improve performance, you can run a job that takes the data set and breaks the partitions out into separate files.

- Then, when a particular subset for the data is to be analyzed, the job needs only to look at that data.

# Motivation Cont'd



- Partitioning by date is one of the most common schemes.
- This helps when we want to analyze a certain span of time, because the data is already grouped by that criterion.
- For instance, suppose you have event data that spans three years in your Hadoop cluster, but for whatever reason the records are not ordered at all by date.
  - If you only care about data from Jan. 27 to February 3 of the current year, you must scan all of the data since those events could be anywhere in the data set.
  - If instead you had the events partitioned into months (i.e., you have a file with January data, a file with February data, etc.), you would only need to run your MapReduce job over the January and February partitions.
- It would be even better if they were partitioned by day!

# Motivation Cont'd



- Partitioning can also help out when you have several different types of records in the same data set, which is increasingly common in NoSQL.
  - For example, in a HTTP server logs, you'll have GET and POST requests, internal system messages, and error messages.
- Analysis may care about only one category of this data, so partitioning it into these categories will help narrow down the data the job runs over before it even runs.
- In an RDBMS, a typical criterion for partitioning is what you normally filter by in the WHERE clause.
  - So, for example, if you are typically filtering down records by country, perhaps you should partition by country.
- This applies in MapReduce as well.
  - If you find yourself filtering out a bunch of records in the mapper due to the same criteria over and over, you should consider partitioning your data set.
  - There is no downside to partitioning other than having to build the partitions.
  - A Map-Reduce job can still run over all the partitions at once if necessary.

# Applicability

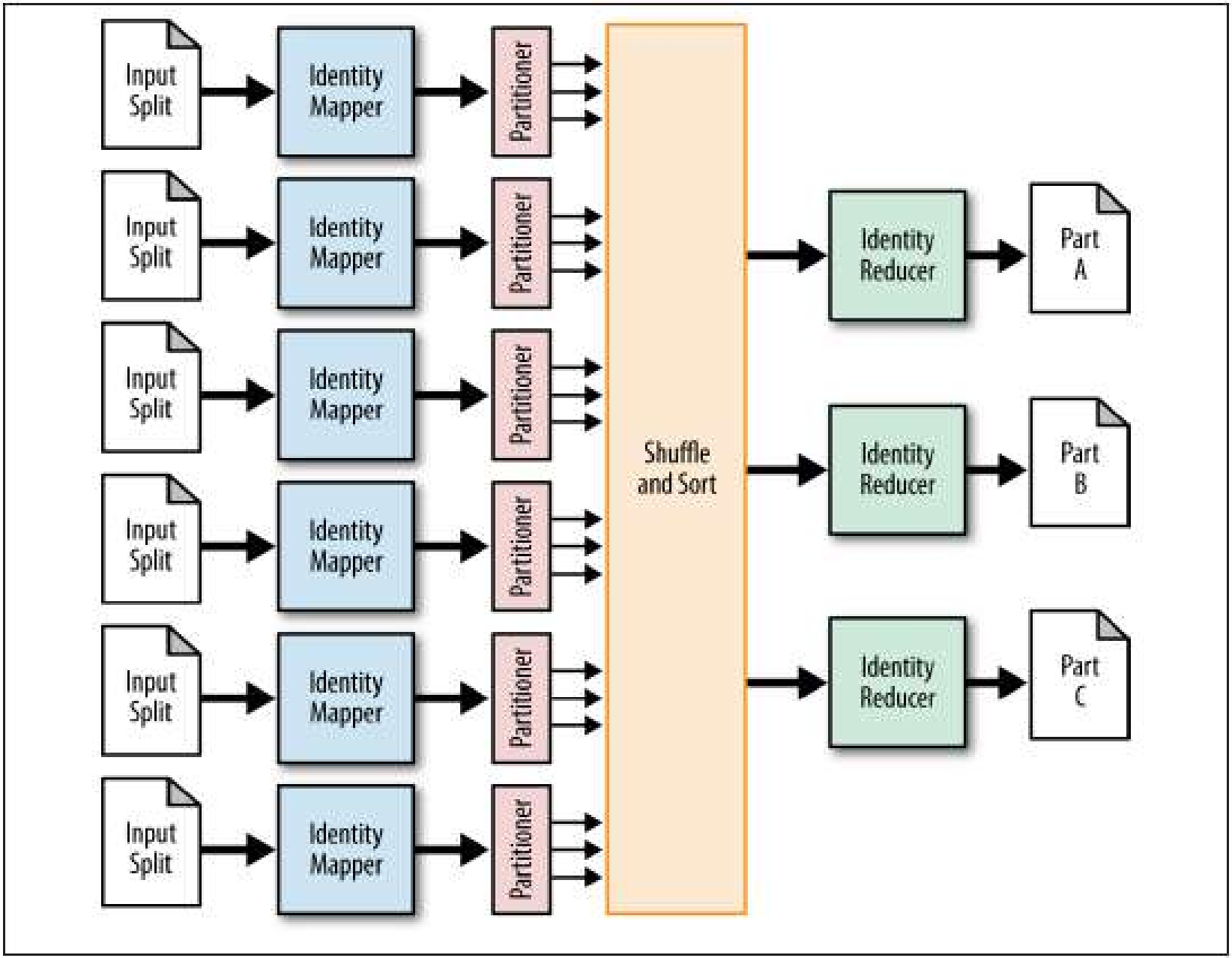


- The one major requirement to apply this pattern is knowing how many partitions you are going to have ahead of time.
- For example, if you know you are going to partition by day of the week, you know that you will have seven partitions.
- You can get around this requirement by running an analytic that determines the number of partitions.
- For example, if you have a bunch of timestamped data, but you don't know how far back it spans, run a job that figures out the date range for you.

# Structure



- This pattern is interesting in that it exploits the fact that the partitioner partitions data.
- There is no actual partitioning logic; all you have to do is define the function that determines what partition a record is going to go to in a custom partitioner.
- Figure in the next slide shows the structure of this pattern.
  - In most cases, the identity mapper can be used.
  - The custom partitioner is the meat of this pattern.
    - The custom partitioner will determine which reducer to send each record to; each reducer corresponds to particular partitions.
  - In most cases, the identity reducer can be used.
    - But this pattern can do additional processing in the reducer if needed.
    - Data is still going to get grouped and sorted, so data can be deduplicated, aggregated, or summarized, per partition.




*The structure of the partitioning pattern*

# Outcomes



- The output folder of the job will have one part file for each partition.
- Since each category will be written out to one large file, this is a great place to store the data in block-compressed SequenceFiles, which are arguably the most efficient and easy-to-use data format in Hadoop.

# Known Uses



- **Partition pruning by continuous value**
  - You have some sort of continuous variable, such as a date or numerical value, and at any one time you care about only a certain subset of that data.
  - Partitioning the data into bins will allow your jobs to load only pertinent data.

- **Partition pruning by category**
  - Instead of having some sort of continuous variable, the records fit into one of several defined categories, such as country, phone area code, or language.

- **Sharding**
  - A system in your architecture has divisions of data—such as different disks—and you need to partition the data into these existing shards.

# Resemblances



- **SQL**
  - Some SQL databases allow for automatically partitioned tables.
  - This allows “partition pruning” which allows the database to exclude large portions of irrelevant data before running the SQL.

- **Other patterns**
  - This pattern is similar to the binning pattern in this lecture.
  - In most cases, binning can perform the same partitioning behavior as this pattern.

# Performance Analysis



- The main performance concern with this pattern is that the resulting partitions will likely not have similar number of records.
- Perhaps one partition turns out to hold 50% of the data of a very large data set.
- If implemented naively, all of this data will get sent to one reducer and will slow down processing significantly.
- It's pretty easy to get around this, though.
- Split very large partitions into several smaller partitions, even if just randomly.
- Assign multiple reducers to one partition and then randomly assign records into each to spread it out a bit better.

# Performance Analysis Cont'd



- For example, consider the "last access date" field for a user in StackOverflow.
- If we partitioned on this property equally over months, the most recent month will very likely be much larger than any other month.
- To prevent skew, it may make sense to partition the most recent month into days, or perhaps just randomly.
- This method doesn't affect processing over partitions, since you know that these set of files represent one larger partition.
- Just include all of them as input.

# Partitioning Examples



- **Partitioning users by last access date**
  - In the StackOverflow data set, users are stored in the order in which they registered.
  - Instead, we want to organize the data into partitions based on the year of the last access date.
  - This is done by creating a custom partitioner to assign record to a particular partition based on that date.

- **Problem:**
  - Given a set of user information, partition the records based on the year of last access date, one partition per year.

# Driver Code



- This driver is a little different than the norm.
- The job needs to be configured to use the custom built partitioner, and this partitioner needs to be configured.
- The minimum last access year needs to be configured, which is 2008.
- The reason for this is explained in the partitioner code section.
- Also, the number of reducers is important to make sure the full range of partitions is accounted for.
- Given that the authors are running this example in 2012, the maximum last access year was in 2011, spanning 4 years from 2008 to 2011.
- Users can fall into these dates as well as those in between, meaning the job is configured to have exactly 4 reducers.

```
...
// Set custom partitioner and min last access date
job.setPartitionerClass(LastAccessDatePartitioner.class);
LastAccessDatePartitioner.setMinLastAccessDate(job, 2008);

// Last access dates span between 2008-2011, or 4 years
job.setNumReduceTasks(4);
...
```

# Mapper code



- The mapper pulls the last access date out of each input record.
- This date is output as the key, and the full input record is output as the value.
- This is so the partitioner can do the work of putting each record into its appropriate partition.
- This key is later ignored during output from the reduce phase.

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

    // 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");

    private IntWritable outkey = new IntWritable();

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

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

        // Grab the last access date
        String strDate = parsed.get("LastAccessDate");

        // Parse the string into a Calendar object
        Calendar cal = Calendar.getInstance();
        cal.setTime(frmt.parse(strDate));
        outkey.set(cal.get(Calendar.YEAR));

        // Write out the year with the input value
        context.write(outkey, value);
    }
}
```

# Partitioner Code



- The partitioner examines each key/value pair output by the mapper to determine which partition the key/value pair will be written.
- Each numbered partition will be copied by its associated reduce task during the reduce phase.
- The partitioner implements the Configurable interface.
- The setConf method is called during task construction to configure the partitioner.
- Here, the minimum value of the last access date is pulled from the configuration.
- The driver is responsible for calling LastAccess DatePartitioner.setMinLastAccessDate during job configuration.
- This date is used to subtract from each key (last access date) to determine what partition it goes to.
- The minimum last access date is 2008, so all users who last logged into StackOverflow in 2008 will be assigned to partition zero.

```
public static class LastAccessDatePartitioner extends
        Partitioner<IntWritable, Text> implements Configurable {

    private static final String MIN_LAST_ACCESS_DATE_YEAR =
            "min.last.access.date.year";

    private Configuration conf = null;
    private int minLastAccessDateYear = 0;

    public int getPartition(IntWritable key, Text value, int numPartitions) {
        return key.get() - minLastAccessDateYear;
    }

    public Configuration getConf() {
        return conf;
    }

    public void setConf(Configuration conf) {
        this.conf = conf;
        minLastAccessDateYear = conf.getInt(MIN_LAST_ACCESS_DATE_YEAR, 0);
    }

    public static void setMinLastAccessDate(Job job,
            int minLastAccessDateYear) {
        job.getConfiguration().setInt(MIN_LAST_ACCESS_DATE_YEAR,
                minLastAccessDateYear);
    }
}
```

**Reducer code.** The reducer code is very simple since we simply want to output the values. The work of partitioning has been done at this point.

```
public static class ValueReducer extends
        Reducer<IntWritable, Text, Text, NullWritable> {

    protected void reduce(IntWritable key, Iterable<Text> values,
            Context context) throws IOException, InterruptedException {
        for (Text t : values) {
            context.write(t, NullWritable.get());
        }
    }
}
```

# Binning Pattern



- **Pattern Description:**
- The binning pattern, much like the previous pattern, moves the records into categories irrespective of the order of records.

- **Intent:**
- For each record in the data set, file each one into one or more categories.

# Motivation of Binning Pattern

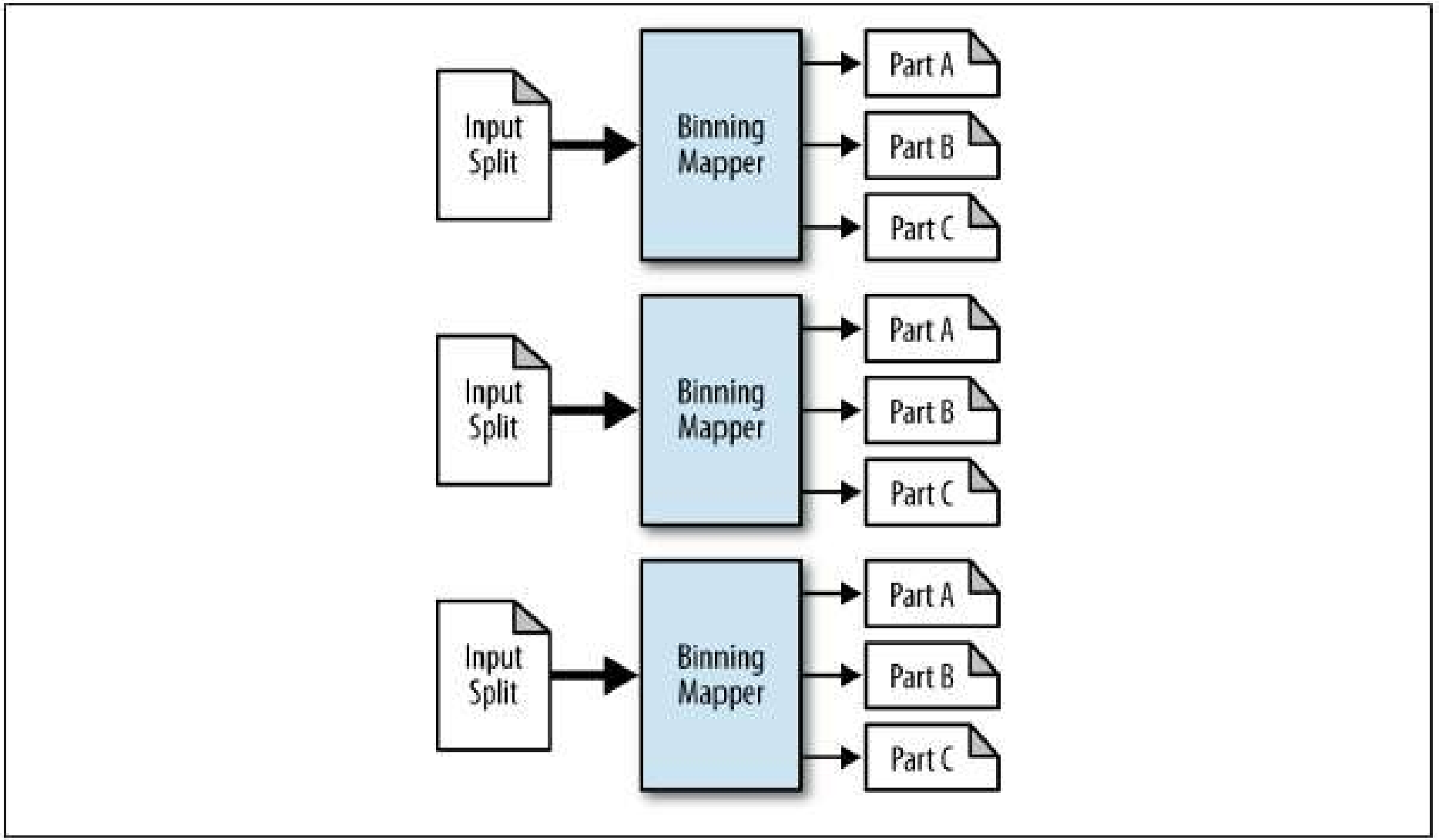


- Binning is very similar to partitioning and often can be used to solve the same problem.
- The major difference is in how the bins or partitions are built using the MapReduce framework. In some situations, one solution works better than the other.
- Binning splits data up in the map phase instead of in the partitioner.
- This has the major advantage of eliminating the need for a reduce phase, usually leading to more efficient resource allocation.
- The downside is that each mapper will now have one file per possible output bin.
  - This means that, if you have a thousand bins and a thousand mappers, you are going to output a total of one million files.
  - This is bad for NameNode scalability and follow-on analytics.
- The partitioning pattern will have one output file per category and does not have this problem.

# Structure of Binning Pattern



- This pattern's driver is unique in using the MultipleOutputs class, which sets up the job's output to write multiple distinct files.
- The mapper looks at each line, then iterates through a list of criteria for each bin.
  - If the record meets the criteria, it is sent to that bin.
- No combiner, partitioner, or reducer is used in this pattern.




*The structure of the binning pattern*

# Outcomes of Binning Pattern



- Each mapper outputs one small file per bin.
- Data should not be left as a bunch of tiny files.
  - At some point, you should run some post-processing that collects the outputs into larger files.

# Resemblances of Binning Pattern



*Pig*

The SPLIT operation in Pig implements this pattern.

```
SPLIT data INTO
    eights IF col1 == 8,
    bigs IF col1 > 8,
    smalls IF (col1 < 8 AND col1 > 0);
```

# Performance Analysis of Binning Pattern



- This pattern has the same scalability and performance properties as other map-only jobs.
- No sort, shuffle, or reduce needs to be performed, and most of the processing is going to be done on data that is local.

# Binning Examples



- Binning by Hadoop-Related Tags
  - We want to filter data by tag into different bins so that we can run follow-on analysis without having to run over all of the data.
  - We care only about the Hadoop-related tags, specifically hadoop, pig, hive, and hbase.
  - Also, if the post mentions Hadoop anywhere in the text or title, we'll put that into its own bin.

- **Problem:**
  - Given a set of StackOverflow posts, bin the posts into four bins based on the tags hadoop, pig, hive, and hbase.
  - Also, create a separate bin for posts mentioning hadoop in the text or title.

# Driver Code



- The driver is pretty much the same boiler plate code, except that we use MultipleOutputs for the different bins.
- MultipleOutputs takes in a name, bins, that is used in the mapper to write different output.
- The name is essentially the output directory of the job.
- Output counters are disabled by default, so be sure to turn those on if you don't expect a large number of named outputs.
- We also set the number of reduce tasks to zero, as this is a map-only job.

```
...
// Configure the MultipleOutputs by adding an output called "bins"
// With the proper output format and mapper key/value pairs
MultipleOutputs.addNamedOutput(job, "bins", TextOutputFormat.class,
        Text.class, NullWritable.class);

// Enable the counters for the job
// If there are a significant number of different named outputs, this
// should be disabled
MultipleOutputs.setCountersEnabled(job, true);

// Map-only job
job.setNumReduceTasks(0);
...
```

# Mapper Code



- The setup phase creates an instance of MultipleOutputs using the context.
- The mapper consists of several if statements to check each of the tags of a post.
- Each tag is checked against one of our tags of interest.
- If the post contains the tag, it is written to the bin.
- Posts with multiple interesting tags will essentially be duplicated as they are written to the appropriate bins.

- Finally, we check whether the body of the post contains the word “hadoop”.
  - If it does, we output it to a separate bin.

- Be sure to close the MultipleOutputs during cleanup!
  - Otherwise, you may not have much output at all.

The typical file names, `part-mnnnnn`, will be in the final output directory. These files will be empty unless the `Context` object is used to write key/value pairs. Instead, files will be named *`bin_name-mnnnnn`*. In the following example, *`bin_name`* will be, `hadoop-tag`, `pig-tag`, `hive-tag`, `hbase-tag`, or `hadoop-post`.

Note that setting the output format of the job to a `NullOutputFormat` will remove these empty output files when using the `mapred` package. In the newer API, the output files are not committed from their `_temporary` directory into the configured output directory in HDFS. This may be fixed in a newer version of Hadoop.

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

    private MultipleOutputs<Text, NullWritable> mos = null;

    protected void setup(Context context) {
        // Create a new MultipleOutputs using the context object
        mos = new MultipleOutputs(context);
    }

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

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

        String rawtags = parsed.get("Tags");

        // Tags are delimited by ><. i.e. <tag1><tag2><tag3>
        String[] tagTokens = StringEscapeUtils.unescapeHtml(rawtags).split(
                "><");

        // For each tag
        for (String tag : tagTokens) {
            // Remove any > or < from the token
            String groomed = tag.replaceAll(">|<", "").toLowerCase();

            // If this tag is one of the following, write to the named bin
            if (groomed.equalsIgnoreCase("hadoop")) {
                mos.write("bins", value, NullWritable.get(), "hadoop-tag");
            }
            if (groomed.equalsIgnoreCase("pig")) {
                mos.write("bins", value, NullWritable.get(), "pig-tag");
            }
            if (groomed.equalsIgnoreCase("hive")) {
                mos.write("bins", value, NullWritable.get(), "hive-tag");
            }
            if (groomed.equalsIgnoreCase("hbase")) {
                mos.write("bins", value, NullWritable.get(), "hbase-tag");
            }
        }

        // Get the body of the post
        String post = parsed.get("Body");

        // If the post contains the word "hadoop", write it to its own bin
        if (post.toLowerCase().contains("hadoop")) {
            mos.write("bins", value, NullWritable.get(), "hadoop-post");
        }
    }

    protected void cleanup(Context context) throws IOException,
            InterruptedException {
        // Close multiple outputs!
        mos.close();
    }
}
```

# Total Order Sorting Pattern



- **Pattern Description:**
- The total order sorting pattern is concerned with the order of the data from record to record.

- **Intent:**
- You want to sort your data in parallel on a sort key.

# Total Order Sorting in MapReduce



We saw in the previous part that when using multiple reducers, each reducer receives (key,value) pairs assigned to them by the Partitioner. When a reducer receives those pairs they are sorted by key, so generally the output of a reducer is also sorted by key. However, the outputs of different reducers are not ordered between each other, so they cannot be concatenated or read sequentially in the correct order.

For example with 2 reducers, sorting on simple Text keys, you can have :

– Reducer 1 output : (**a**,5), (**d**,6), (**w**,5)

– Reducer 2 output : (**b**,2), (**c**,5), (**e**,7)

The keys are only sorted if you look at each output individually, but if you read one after the other, the ordering is broken.

The objective of *Total Order Sorting* is to have all outputs sorted **across all reducers** :

– Reducer 1 output : (**a**,5), (**b**,2), (**c**,5)

– Reducer 2 output : (**d**,6), (**e**,7), (**w**,5)

This way the outputs can be read/searched/concatenated sequentially as a single ordered output.

# Motivation



- Sorting is easy in sequential programming.

- Sorting in MapReduce, or more generally in parallel, is not easy.
  - This is because the typical “divide and conquer” approach is a bit harder to apply here.

- Each individual reducer will sort its data by key, but unfortunately, this sorting is not global across all data.
  - What we want to do here is a total order sorting where, if you concatenate the output files, the records are sorted.
  - If we just concatenate the output of a simple MapReduce job, segments of the data will be sorted, but the whole set will not be.

# Motivation Cont'd



- Sorted data has a number of useful properties.
  - Sorted by time, it can provide a timeline view on the data.
  - Finding things in a sorted data set can be done with binary search instead of linear search.

- In the case of MapReduce, we know the upper and lower boundaries of each file by looking at the last and first records, respectively.
- This can be useful for finding records, as well, and is one of the primary characteristics of HBase.
- Some databases can bulk load data faster if the data is sorted on the primary key or index column.
- There are countless more reasons to have sorted data from an application standpoint or follow-on system standpoint.
- However, having data sorted for use in MapReduce serves little purpose, so hopefully this expensive operation only has to be done sparingly.

# Applicability



- The main requirement here is pretty obvious:
  - your sort key has to be comparable so the data can be ordered.

# Structure



- Total order sorting may be one of the more complicated patterns you'll see.
- The reason this is that you first have to determine a set of partitions divided by ranges of values that will produce equal-sized subsets of data.
- These ranges will determine which reducer will sort which range of data.
- Then something similar to the partitioning pattern is run:
  - a custom partitioner is used to partition data by the sort key.
  - The lowest range of data goes to the 1st reducer, the next range goes to the 2nd reducer, so on.

# Structure Cont'd



- This pattern has two phases:
  - an analyze phase that determines the ranges
  - the order phase that actually sorts the data.

- The analyze phase is optional in some ways.
- You need to run it only once if the distribution of your data does not change quickly over time, because the value ranges it produces will continue to perform well.
- Also, in some cases, you may be able to guess the partitions yourself, if the data is evenly distributed

- For example, if you are sorting comments by user ID, and you have a million users, you can assume that with a thousand reducers, each range is going to have a range of a thousand users.
  - This is because comments by user ID should be spread out evenly and since you know the number of total users, you can divide that number by the number of reducers you want to use.

# Structure Cont'd



- The analyze phase is a random sampling of the data.
- The partitions are then based on that random sample.
- The principle is that partitions that evenly split the random sample should evenly split the larger data set well.

# The structure of the analyze step is as follows



- **The mapper does a simple random sampling.**
  - When dividing records, it outputs the sort key as its output key so that the data will show up sorted at the reducer.
  - We don't care at all about the actual record, so we'll just use a null value to save on space.
- **Ahead of time, determine the number of records in the total data set and figure out what percentage of records you'll need to analyze to make a reasonable sample.**
  - For example, if you plan on running the order with a thousand reducers, sampling about a hundred thousand records should give nice, even partitions.
  - Assuming you have a billion records, divide 100,000 by 1,000,000,000.
  - This gives 0.0001, meaning .01% of the records should be run through the analyze phase.
- **Only one reducer will be used here.**
  - This will collect the sort keys together into a sorted list
    - they come in sorted, so that will be easy.
  - Then, when all of them have been collected, the list of keys will be sliced into the data range boundaries.

# The structure of the order step is as follows



- The order phase is a relatively straightforward application of MapReduce that uses a custom partitioner.
- **The mapper extracts the sort key in the same way as the analyze step.**
  - However, this time the record itself is stored as the value instead of being ignored.
- **A custom partitioner is used that loads up the partition file.**
  - In Hadoop, the TotalOrderPartitioner is built specifically for this purpose.
  - It takes the data ranges from the partition file produced in the previous step and decides which reducer to send the data to.
- **The reducer's job here is simple.**
  - The shuffle and sort take care of the heavy lifting.
  - The reduce function simply takes the values that have come in and outputs them.
  - The number of reducers needs to be equal to the number of partitions for the TotalOrderPartitioner to work properly.

# Remarks



- Note that the number of ranges in the intermediate partition needs to be equal to the number of reducers in the order step.
  - If you decide to change the number of reducers and you've been reusing the same file, you'll need to rebuild it.
- If you want to have a primary sort key and a secondary sort key, concatenate the keys, delimited by something.
  - For example, if you want to sort by last name first, and city second, use a key that looks like Smith^Baltimore.
- Using Text for nearly everything in Hadoop is very natural since that's the format in which data is coming in.
  - Be careful when sorting on numerical data, though!
  - The string "10000" is less than than "9" if they are compared as strings, which is not what we want.
  - Either pad the numbers with zeros or use a numerical data type.

# Outcomes



- The output files will contain sorted data, and the output file names will be sorted such that the data is in a total sorting.

- In Hadoop, you'll be able to issue

```
hadoop fs -cat output/part-r-*
```

and retrieve the data in a sorted manner.

# Resemblances



*SQL*

Ordering in SQL is pretty easy!

```
SELECT * FROM data ORDER BY col1;
```

*Pig*

Ordering in Pig is syntactically pretty easy, but it's a very expensive operation. Behind the scenes, it will run a multi-stage MapReduce job to first find the partitions, and then perform the actual sort.

```
c = ORDER b BY col1;
```

# Performance Analysis



- This operation is expensive because you need to load and parse the data twice:
  - first to build the partition ranges
  - then to actually sort the data.
- The job that builds the partitions is straightforward and efficient since it has only one reducer and sends a minimal amount of data over the network.
  - The output file is small, so writing it out is trivial.
  - Also, you may only have to run this now and then, which will amortize the cost of building it over time.
- The order step of the job has performance characteristics similar to the other data organization patterns, because it has to move all of the data over the network and write all of the data back out.
  - Therefore, you should use a relatively large number of reducers.

# Total Order Sorting Examples



- **Sort users by last visit**
  - The user data in our StackOverflow data set is in the order of the account's creation.
  - Instead, we'd like to have the data ordered by the last time they have visited the site.

- For this example, we have a special driver that runs both the analyze and order steps.

- Also, there are two sets of MapReduce jobs, one for analyze and one for order.

# Driver code



- Let's break the driver down into two sections:
  - building the partition list via sampling
  - then performing the sort.
- The 1st section parses the input command line arguments to create input/output variables from them.
  - It creates path files to the partition list and the staging directory.
  - The partition list is used by the TotalOrderPartitioner top properly sort the key/value pairs.
  - The staging directory is used to store intermediate output between the two jobs.
  - There is nothing too special with the first job configuration.
  - The main thing to note is that the first job is a map-only only job that uses a SequenceFileOutputFormat.
- The 2nd job uses the identity mapper and our reducer implementation.
  - The input is the output from the first job, so we'll use the identity mapper to output the key/value pairs as they are stored from the output.
  - The job is configured to 10 reducers, but any reasonable number can be used.
  - Next, the partition file is configured, even though we have not created it yet.

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Path inputPath = new Path(args[0]);
    Path partitionFile = new Path(args[1] + "_partitions.lst");
    Path outputStage = new Path(args[1] + "_staging");
    Path outputOrder = new Path(args[1]);

    // Configure job to prepare for sampling
    Job sampleJob = new Job(conf, "TotalOrderSortingStage");
    sampleJob.setJarByClass(TotalOrderSorting.class);

    // Use the mapper implementation with zero reduce tasks
    sampleJob.setMapperClass(LastAccessDateMapper.class);
    sampleJob.setNumReduceTasks(0);

    sampleJob.setOutputKeyClass(Text.class);
    sampleJob.setOutputValueClass(Text.class);

    TextInputFormat.setInputPaths(sampleJob, inputPath);

    // Set the output format to a sequence file
    sampleJob.setOutputFormatClass(SequenceFileOutputFormat.class);
    SequenceFileOutputFormat.setOutputPath(sampleJob, outputStage);

    // Submit the job and get completion code.
    int code = sampleJob.waitForCompletion(true) ? 0 : 1;

    ...
```

# InputSampler Utility



- The next important line uses the InputSampler utility.
  - This sampler writes the partition file by reading through the configured input directory of the job.

- Using the RandomSampler, it takes a configurable number of samples of the previous job's output.
  - This can be an expensive operation, as the entire output is read using this constructor.
  - Another constructor of RandomSampler allows you to set the number of input splits that will be sampled.
  - This will increase execution time, but you might not get as good a distribution.

- After the partition file is written, the job is executed.
- The partition file and staging directory are then deleted, as they are no longer needed for this example.

```
...

if (code == 0) {
    Job orderJob = new Job(conf, "TotalOrderSortingStage");
    orderJob.setJarByClass(TotalOrderSorting.class);

    // Here, use the identity mapper to output the key/value pairs in
    // the SequenceFile
    orderJob.setMapperClass(Mapper.class);
    orderJob.setReducerClass(ValueReducer.class);

    // Set the number of reduce tasks to an appropriate number for the
    // amount of data being sorted
    orderJob.setNumReduceTasks(10);

    // Use Hadoop's TotalOrderPartitioner class
    orderJob.setPartitionerClass(TotalOrderPartitioner.class);

    // Set the partition file
    TotalOrderPartitioner.setPartitionFile(orderJob.getConfiguration(),
            partitionFile);

    orderJob.setOutputKeyClass(Text.class);
    orderJob.setOutputValueClass(Text.class);

    // Set the input to the previous job's output
    orderJob.setInputFormatClass(SequenceFileInputFormat.class);
    SequenceFileInputFormat.setInputPaths(orderJob, outputStage);

    // Set the output path to the command line parameter
    TextOutputFormat.setOutputPath(orderJob, outputOrder);

    // Set the separator to an empty string
    orderJob.getConfiguration().set(
            "mapred.textoutputformat.separator", "");

    // Use the InputSampler to go through the output of the previous
    // job, sample it, and create the partition file
    InputSampler.writePartitionFile(orderJob,
            new InputSampler.RandomSampler(.001, 10000));

    // Submit the job
    code = orderJob.waitForCompletion(true) ? 0 : 2;
}

// Clean up the partition file and the staging directory
FileSystem.get(new Configuration()).delete(partitionFile, false);
FileSystem.get(new Configuration()).delete(outputStage, true);

System.exit(code);
}
```

**Analyze mapper code.** This mapper simply pulls the last access date for each user and sets it as the sort key for the record. The input value is output along with it. These key/value pairs, per our job configuration, are written to a `SequenceFile` that is used to create the partition list for the `TotalOrderPartitioner`. There is no reducer for this job.

```
public static class LastAccessDateMapper extends
        Mapper<Object, Text, Text, 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());

        outkey.set(parsed.get("LastAccessDate"));
        context.write(outkey, value);
    }
}
```

# Order Mapper and Reducer Codes



- **Order Mapper**
  - This job simply uses the identity mapper to take each input key/value pair and output them.
  - No special configuration or implementation is needed.

- **Order Reducer**
  - Because the TotalOrderPartitioner took care of all the sorting, all the reducer needs to do is output the values with a NullWritable object.
  - This will produce a part file for this reducer that is sorted by last access date.
  - The partitioner ensures that the concatenation of all these part files (in order) produces a totally ordered data set.

```
public static class ValueReducer extends
        Reducer<Text, Text, Text, NullWritable> {

    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        for (Text t : values) {
            context.write(t, NullWritable.get());
        }
    }
}
```

# Shuffling Pattern



- **Pattern Description:**
- The total order sorting and shuffling patterns are opposites in terms of effect, but the latter is also concerned with the order of data in records.

- **Intent:**
- You have a set of records that you want to completely randomize.

# Motivation



- This whole lecture has been about applying some sort of order to your data set except for this pattern which is instead about completely destroying the order.

- The use cases for doing such a thing are definitely few and far between, but two stand out.
  - One is shuffling the data for the purposes of anonymizing it.
  - Another is randomizing the data set for repeatable random sampling.

# Motivation Cont'd



- **Anonymizing data has recently become important for organizations that want to maintain their users' privacy, but still run analytics.**
  - The order of the data can provide some information that might lead to the identity of a user.
  - By shuffling the entire data set, organization is taking an extra step to anonymize the data.

- **Another reason for shuffling data is to perform some sort of repeatable random sampling.**
  - For example, the first hundred records will be a simple random sampling.
  - Every time we pull the first hundred records, we'll get the same sample.
  - This allows analytics that run over a random sample to have a repeatable result.
  - Also, a separate job won't have to be run to produce a simple random sampling every time you need a new sample.

# Structure



- All the mapper does is output the record as the value along with a random key.
- The reducer sorts the random keys, further randomizing the data.
- In other words, each record is sent to a random reducer.
- Then, each reducer sorts on the random keys in the records, producing a random order in that reducer.
- The mapper in the shuffle pattern is barely doing anything.
  - This would be a good time to anonymize the data further by transforming the records into an anonymized form.

# Outcomes



- Each reducer outputs a file containing random records.

# Resemblances



*SQL*

The SQL equivalent to this is to order the data set by a random value, instead of some column in the table. This makes it so each record is compared on the basis of two random numbers, which will produce a random ordering. We don't have to go all the way and do a total ordering in MapReduce, as in the previous pattern. This is because sending data to a random reducer is sufficient.

```
SELECT * FROM data ORDER BY RAND()
```

*Pig*

Shuffling in Pig can be done as we did it in SQL: performing an `ORDER BY` on a random column. In this case, doing a total ordering is unnecessary. Instead, we can `GROUP BY` a random key, and then `FLATTEN` the grouping. This effectively implements the shuffle pattern we proposed behind the scenes.

```
c = GROUP b BY RANDOM();
d = FOREACH c GENERATE FLATTEN(b);
```

# Performance Analysis



- The shuffle has some very nice performance properties.
  - Since the reducer each record goes to is completely random, the data distribution across reducers will be completely balanced.
  - With more reducers, the data will be more spread out.
  - The size of the files will also be very predictable:
    - each is the size of the data set divided by the number of reducers.
    - This makes it easy to get a specific desired file size as output.
- Other than that, the typical performance properties for the other patterns in this lecture apply.
- The pattern shuffles all of the data over the network and writes all of the data back to HDFS, so a relatively high number of reducers should be used.

# Shuffle Examples



- **Anonymizing StackOverflow comments**
  - To anonymize the StackOverflow comments, this example strips out the user ID and row ID, and truncates the date and time to just the date.
  - Then the data is shuffled.

- **Problem:**
  - Given a large data set of StackOverflow comments, anonymize each comment by removing IDs, removing the time from the record, and then randomly shuffling the records within the data set.

# Mapper Code



- The mapper transforms the data using our utility function that parses the data.
- Each XML attribute is looked at, and an action is taken based on the attribute to create a new line of XML.
- If it is a user ID or row ID, it is ignored.
- If it is a creation date, the characters following the ***T*** are removed to ignore the time
- Otherwise, just write out the XML attribute and value.
- A random key is generated and output along with the newly constructed record.

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

private IntWritable outkey = new IntWritable();
private Random rndm = new Random();
private Text outvalue = new Text();

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

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

    if (parsed.size() > 0) {
        StringBuilder bldr = new StringBuilder();
        // Create the start of the record
        bldr.append("<row ");

        // For each XML attribute
        for (Entry<String, String> entry : parsed.entrySet()) {

            // If it is a user ID or row ID, ignore it
            if (entry.getKey().equals("UserId")
                    || entry.getKey().equals("Id")) {
            } else if (entry.getKey().equals("CreationDate")) {

                // If it is a CreationDate, remove the time from the date
                // i.e., anything after the 'T' in the value
                bldr.append(entry.getKey()
                        + "=\""
                        + entry.getValue().substring(0,
                                entry.getValue().indexOf('T')) + "\" ");
            } else {
                // Otherwise, output the attribute and value as is
                bldr.append(entry.getKey() + "=\"" + entry.getValue()
                        + "\" ");
            }

        }
        // Add the /> to finish the record
        bldr.append("/>");

        // Set the sort key to a random value and output
        outkey.set(rndm.nextInt());
        outvalue.set(bldr.toString());
        context.write(outkey, outvalue);
    }
}
}
```

**Reducer code.** This reducer class just outputs the values in order to strip out the random key.

```
public static class ValueReducer extends
        Reducer<IntWritable, Text, Text, NullWritable> {

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

        for (Text t : values) {
            context.write(t, NullWritable.get());
        }
    }
}
```