FAQs

• PA1 submission is open
  • September 18, 5:00PM via Canvas
  • Individual submission (No team submission)

• Partitioner VS. Shuffling?
  • What if the end of input to the mapper is not included in the current data chunk?

FAQs: Quiz 2

Find the 100 clients who spent the most electricity (kilowatt) for each zip code for the last month. Files contain information about the last month only. The data is formatted as follows:

{customerID, TAB, address, TAB, zipcode, TAB, electricity usage, LINEFEED}. Assume that each line will be used as the input to a Map function.

Output of map: <zipcode, [a list of local top 100 info (info includes userID and electricity usage)]>

Input to the reducer: <zipcode, [a list of all of the local top 100 info (info includes userID and electricity usage)]>

Reducer: Select global top 100 within the zipcode and sort them based on the electricity usage

Output of reduce: <zipcode, [a list of global top 100 userID]>
Topics covered in this lecture

- MapReduce Design Pattern IV: Join Patterns

Join Patterns
- Data is all over the place
- "Joins" allow users to create a smaller reference set or filter out or select dataset to discover interesting relationships across datasets
- Joining a terabyte of data onto another terabyte dataset could require up to two terabytes of bandwidth!
  - That’s before any actual join logic can be done!

1. Reduce Side Join Pattern
2. Replicated Join Pattern
3. Composite Join Pattern
4. Cartesian Product Pattern

A Refresher on Joins
- A Join is an operation that combines records from two or more datasets based on a field or set of fields
- Foreign key
- The foreign key is the field in a relational table that matches the column of another table
- Used as a means to cross-reference between tables

Example

<table>
<thead>
<tr>
<th>UserID</th>
<th>Reputation</th>
<th>Location</th>
<th>Text</th>
<th>UserID</th>
<th>PostID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3738</td>
<td>New York, NY</td>
<td>Not sure why this is getting downvoted</td>
<td>3</td>
<td>48002</td>
<td>Hey, of course, it's atrocious!</td>
</tr>
<tr>
<td>5</td>
<td>17556</td>
<td>San Diego, CA</td>
<td>Please see my post below</td>
<td>5</td>
<td>48678</td>
<td>Thank you very much for your reply</td>
</tr>
<tr>
<td>9</td>
<td>3443</td>
<td>Oakland, CA</td>
<td>HTML is not a subset of XML!</td>
<td>9</td>
<td>59276</td>
<td>HTML is not a subset of XML!</td>
</tr>
</tbody>
</table>
Join Pattern

1. Reduce Side Join Pattern

- Most straightforward implementation of a join in MapReduce
- Requires a large amount of network bandwidth
  - Bulk of the data is sent to the reduce phase
  - If you have resources available, this will be a possible solution

Example:
Dataset A: User information (user_id, first_name, last_name, ...)
Dataset B: Comment information (comment_id, user_id, ...)
Task: Joining datasets A and B based on user_id
Reducer Code

```java
public static class UserJoinReducer extends Reducer<Text, Text, Text, Text> {
    private static final Text EMPTY_TEXT = Text.empty;
    private Text outputKey = new Text();
    private Text outputValue = new Text();
    private ArrayList<Text> listA = new ArrayList<Text>();
    private ArrayList<Text> listB = new ArrayList<Text>();

    public void setup(Context context) throws IOException, InterruptedException {
        // Set the type of Join from our configuration
        String joinType = context.getConfiguration().get("joinType");
        // Initialize our lists
        listA.clear();
    }

    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        // Execute our join logic now that the lists are filled
        executeJoinLogic(context);
    }
}
```

User Mapper Code

```java
public static class UserJoinMapper extends Mapper<Object, Text, Text, Text> {
    private static final Text EMPTY_TEXT = Text.empty;
    private Text outputKey = new Text();
    private Text outputValue = new Text();
    private ArrayList<Text> listA = new ArrayList<Text>();
    private ArrayList<Text> listB = new ArrayList<Text>();

    public void setup(Context context) throws IOException, InterruptedException {
        // Set the type of Join from our configuration
        String joinType = context.getConfiguration().get("joinType");
        // Initialize our lists
        listA.clear();
    }

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        // Parse the input string into a nice map
        Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());
        // The foreign join key is the user ID
        String userId = parsed.get("userId");
        // If this record is for the reducer and then output
        context.write(outputKey, outputValue);
    }
}
```

Reducer Code

```java
public static class UserJoinReducer extends Reducer<Text, Text, Text> {
    private static final Text EMPTY_TEXT = Text.empty;
    private Text outputKey = new Text();
    private Text outputValue = new Text();
    private ArrayList<Text> listA = new ArrayList<Text>();
    private ArrayList<Text> listB = new ArrayList<Text>();

    public void setup(Context context) throws IOException, InterruptedException {
        // Set the type of Join from our configuration
        String joinType = context.getConfiguration().get("joinType");
        // Initialize our lists
        listA.clear();
    }

    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        // Execute our join logic now that the lists are filled
        executeJoinLogic(context);
    }
}
```

Driver Code

```java
... // Use MultipleInputs to set which input uses which mapper
... // This will keep parsing of each data set separate from a logical standpoint
... MultipleInputs.addInputPath(job, new Path(args[0]), TextInputFormat.class, UserJoinMapper.class);
... MultipleInputs.addInputPath(job, new Path(args[1]), TextInputFormat.class, CommentJoinMapper.class);
... job.setConfiguration(...);
... ...
```
Inner Join Code

```java
if (joinType.equalsIgnoreCase("inner")) {  
    // If both lists are not empty, join A with B  
    if (!listA.isEmpty() && !listB.isEmpty()) {  
        for (Text A : listA) {  
            for (Text B : listB) {  
                context.write(A, B);  
            }  
        }  
    }  
} ...
```

Left outer Join Code

```java
...  
else if (joinType.equalsIgnoreCase("leftouter")) {  
    // For each entry in A,  
    for (Text A : lista) {  
        // If list B is not empty, join A and B  
        if (!listB.isEmpty()) {  
            for (Text B : listB) {  
                context.write(A, B);  
            }  
        } else {  
            // Else, output A by itself  
            context.write(A, EMPTY_TEXT);  
        }  
    }  
} ...
```

Right outer Join Code

```java
...  
else if (joinType.equalsIgnoreCase("rightouter")) {  
    // For each entry in B,  
    for (Text B : listb) {  
        // If list A is not empty, join A and B  
        if (!lista.isEmpty()) {  
            for (Text A : lista) {  
                context.write(A, B);  
            }  
        } else {  
            // Else, output B by itself  
            context.write(EMPTY_TEXT, B);  
        }  
    }  
} ...
```

Performance analysis

- The reducer side join puts a lot of strain on the cluster’s network
- The foreign key and output record of each input record are extracted
  - No data can be filtered ahead of time
  - Almost all of the data will be sent to the shuffle and sort step
- Reduce side joins will typically utilize relatively more reducers than your typical analytics

Replicated Join

- Special type of join operation between one reasonably large and (many) small data set(s) that can be performed on the map-side
- Mapper
  - Reads all files from the distributed cache during the setup phase
  - Sorting them in in-memory lookup tables
  - Performs mapper process
  - Joining data
  - If the foreign key is not found in the in-memory structure?
    - The record is either omitted or output (based on the join type)
- No combiner/partitioner/reducer needed
Structure of the replicated join pattern

![Diagram of replicated join pattern]

Hadoop DistributedCache

- Provided by the Hadoop MapReduce Framework
- Caches read only text files, archives, jar files etc.
- Once a file is cached for a job using Distributed cache
  - Data will be available on each data node where map/reduce tasks are running

Hadoop DistributedCache

- Default size of the Hadoop distributed cache is 10GB
- Configurable in mapred-site.xml
- Data consistency
  - Hadoop Distributed Cache tracks the modification of timestamps of the cache file
- Overhead
  - Object serialization

Using DistributedCache for replicated join

- A small file is pushed to all map tasks using DistributedCache
- Useful for join between a small set and a large set of data
  - e.g. user information vs. transaction records, user information vs. comment history
- Mapper Code
  - Setup phase
    - User data is read from the DistributedCache and stored in memory
  - Map phase
    - For each input record (from the large dataset), the user information is retrieved from the
      HashMap
    - assemble a joined record

Mapper Code

```
public static class ReplicatedJoinMapper extends Mapper < Object, Text, Text, Text > {
  private static final Text EMPTY_TEXT = new Text();
  private HashMap < String, String > userIdToInfo = new HashMap < String, String >();
  private Text outvalue = new Text();
  private String joinType = null;

  public void setup(Context context) throws IOException,InterruptedException {
    Path[] files = DistributedCache.getLocalCacheFiles(context.getConfiguration());
  }
```
MapReduce Design Patterns IV: Join Patterns

3. Composite Join

- Joins very large datasets together
- And if the datasets are sorted by foreign key
- No shuffle and sort needed
- Each input dataset must be partitioned and sorted in a specific way and divided into the same number of partitions

Example:
Dataset A: User information (user_id, first_name, last_name, …)
Dataset B: Comment information (comment_id, user_id, …)
Task: Joining datasets A and B based on user_id
Joining process (Inner Join)- Within a group

UserID, 11 ~ 20 were grouped together

Joining process (Inner Join)-Between groups

Example: Composite user comment join

- Preconditions
  - User and comment datasets should be preprocessed by MapReduce
  - CompositeInputFormat will be used
  - Output should use the TextOutputFormat
  - Key: userXML
  - Value: userXML or comment XML
  - Hadoop Key/Value/TextOutputFormat can parse these information

- Note: Since CompositeInputFormat uses Text objects as the key for comparisons
  - Align your keys as Text objects (rather than LongWritable)
  - e.g. "12345" comes before "2"

Driver Code

```java
public static void main(String[] args) throws Exception {
    Path userPath = new Path(args[0]);
    Path commentPath = new Path(args[1]);
    String outputDir = new Path(args[2]);
    String joinType = args[3];
    JobConf conf = new JobConf();
    conf.setJobClass("CompositeJoin");
    conf.setMapperClass(CompositeMapper.class);
    conf.setReducerClass(CompositeReducer.class);

    // Set the input format class to a CompositeInputFormat class.
    // The CompositeInputFormat will parse all of input class
    // and output records to our mapper.
    conf.setInputFormatClass(CompositeInputFormat.class);

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

    // set the path to the output directory
    conf.setOutputPath(outputDir);

    // set the number of reduce tasks
    conf.setNumReduceTasks(2);

    // set the input format class to a CompositeInputFormat class.
    // The CompositeInputFormat will parse all of input class
    // and output records to our mapper.
    conf.setInputFormatClass(CompositeInputFormat.class);

    // will parse all of our input files
    // and output records to our mapper.
    conf.setMapperClass(CompositeMapper.class);

    Job job = new Job(conf);
    job.setJobName("CompositeJoin");
    job.setJarByClass(CompositeJoin.class);

    TextInputFormat.setInputPaths(job, userPath, commentPath);
    TextInputFormat.setInputPaths(job, commentPath, userPath);

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

    job.waitForCompletion(true);
}
```
Mapper Code

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

All of the matching has been done in CompositeInputFormat already.

Questions?