FAQs

- PA0 submission is open
  - Feb. 6, 5:00PM via Canvas
  - Individual submission (No team submission)
  - If you have not been assigned the "port range", please contact the GTA immediately

- PA1 has been posted
  - Feb. 21, 5:00PM via Canvas
  - Individual submission (No team submission)

- Total Order Sorting Pattern

Topics

- MapReduce Design Pattern III. Data Organization Patterns
- MapReduce Design Pattern IV. Join Patterns

Part 1. Large Scale Data Analytics

Design Pattern 3: Data Organization Patterns

Total Order Sorting Pattern

- Sorts your data
  - e.g. Sorting 1TB of numeric values
  - e.g. Sorting comments by userID and you have a million users
Structure of Total Order Sorting Pattern

- Two phases
  - Analysis phase
    - Determines the ranges
  - Sorting phase
    - Actually sorts the data

Structure of Total Order Sorting Pattern
- Analysis phase
  - Performs a simple random sampling
  - Generates outputs with the sort key as its output keys
  - Data will show up as sorted at the reducer
  - Sampling rate?
    - Assume that the number of records in the entire dataset is known (or can be estimated)
    - If you plan on running the order with a thousand reducers
      - Sampling about a hundred thousand records will be enough
    - Only one reducer will be used
      - Collects the sort keys together into a sorted list
      - The list of sorted keys will be sliced into the data range boundaries

Structure of Total Order Sorting Pattern
- Sorting phase
  - Mapper extracts the sort key
    - Stores the sort key to the "value"
  - Custom partitioner
    - Use TotalOrderPartitioner (Hadoop API)
      - Takes the data ranges from the partition file and decides which reducer to send the data
      - Dynamic and load balanced
  - Reducer
    - The number of reducers needs to be equal to the number of partitions

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>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3738</td>
<td>New York, NY</td>
</tr>
<tr>
<td>4</td>
<td>12946</td>
<td>New York, NY</td>
</tr>
<tr>
<td>5</td>
<td>17556</td>
<td>San Diego, CA</td>
</tr>
<tr>
<td>9</td>
<td>3443</td>
<td>Oakland, CA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UserID</th>
<th>PostID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>35164</td>
<td>Not sure why this is getting downvoted.</td>
</tr>
<tr>
<td>3</td>
<td>48802</td>
<td>Hey, of course, it's all true!</td>
</tr>
<tr>
<td>5</td>
<td>48920</td>
<td>Please see my post below</td>
</tr>
<tr>
<td>8</td>
<td>48678</td>
<td>Thank you very much for your reply</td>
</tr>
</tbody>
</table>

### Full outer join

Records from a foreign key not present in both table will be also in the final table.

**Left Outer Join**

Unmatched records in the "left" table will be in the final table.
Null values in the columns of the right table that did not match.

**Right Outer Join**

The right table records are kept and the left table values are null where appropriate.

**Full outer join**
contains all unmatched records from both tables.

### Anti Join

<table>
<thead>
<tr>
<th>UserID</th>
<th>Reputation</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3738</td>
<td>New York, NY</td>
</tr>
<tr>
<td>4</td>
<td>12946</td>
<td>New York, NY</td>
</tr>
<tr>
<td>5</td>
<td>17556</td>
<td>San Diego, CA</td>
</tr>
<tr>
<td>9</td>
<td>3443</td>
<td>Oakland, CA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UserID</th>
<th>PostID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>35164</td>
<td>Not sure why this is getting downvoted.</td>
</tr>
<tr>
<td>3</td>
<td>48802</td>
<td>Hey, of course, it's all true!</td>
</tr>
<tr>
<td>5</td>
<td>48920</td>
<td>Please see my post below</td>
</tr>
<tr>
<td>8</td>
<td>48678</td>
<td>HTML is not a subset of XML!</td>
</tr>
</tbody>
</table>

### Inner Join

**Dataset A**

<table>
<thead>
<tr>
<th>UserID</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>New York, NY</td>
</tr>
<tr>
<td>4</td>
<td>New York, NY</td>
</tr>
<tr>
<td>5</td>
<td>San Diego, CA</td>
</tr>
<tr>
<td>9</td>
<td>Oakland, CA</td>
</tr>
</tbody>
</table>

**Dataset B**

<table>
<thead>
<tr>
<th>UserID</th>
<th>PostID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>35164</td>
<td>Not sure why this is getting downvoted.</td>
</tr>
<tr>
<td>3</td>
<td>48802</td>
<td>Hey, of course, it's all true!</td>
</tr>
<tr>
<td>5</td>
<td>48920</td>
<td>Please see my post below</td>
</tr>
<tr>
<td>8</td>
<td>48678</td>
<td>HTML is not a subset of XML!</td>
</tr>
</tbody>
</table>

### 1. Reduce Side Join Pattern

- **MapReduce Design Patterns IV: Join Patterns**

1. **Join Pattern**

   - **Inner Join**
     - Records from both A and B that contain identical values for the given foreign key are brought together.
     - **Anti Join**
       - Full outer join minus the inner join.
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

Structure of the reduce side join pattern

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

Driver Code

```java
    // Use MultipleInputs to set which input uses what 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.getConfiguration();
```

User Mapper Code

```java
    public static class UserJoinMapper 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());
            String userId = parsed.get("Id");
            outkey.set(userId);
            // Flag this record for the reducer and then output
            outvalue.set("A" + value.toString());
            context.write(outkey, outvalue);
        }
    }
```

Comment mapper code

```java
    public static class CommentJoinMapper 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());
            String userId = parsed.get("UserId");
            outkey.set(userId);
            // Flag this record for the reducer and then output
            outvalue.set("B" + value.toString());
            context.write(outkey, outvalue);
        }
    }
```
Reducer Code

```java
public static class UserJoinReducer extends Reducer<Text, Text, Text, Text> {
    private static final Text EMPTY_TEXT = Text.create("");
    private Text tmp = new Text();
    private ArrayList<Text> listA = new ArrayList<Text>();
    private ArrayList<Text> listB = new ArrayList<Text>();
    private String joinType = null;

    public void setup(Context context) {
        // Get the type of join from our configuration
        joinType = context.getConfiguration().get("join.type");
    }

    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        // Clear our lists
        listA.clear();
        listB.clear();
        // Iterate through all our values, binning each record based on what
        // it was tagged with. Make sure to remove the tag!
        while (values.hasNext()) {
            tmp = values.next();
            if (tmp.charAt(0) == 'A') {
                listA.add(new Text(tmp.toString().substring(1)));
            } else if (tmp.charAt(0) == 'B') {
                listB.add(new Text(tmp.toString().substring(1)));
            }
            } // Execute our join logic now that the lists are filled
    executeJoinLogic(context);
    }

    private void executeJoinLogic(Context context) throws IOException, InterruptedException {
        ...
    }
}
```

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);
        }
    }
}
```

MapReduce Design Patterns IV: Join Patterns

2. Replicated Join
Replicated Join

- Special type of join operation between one 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

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

Working with DistributedCache

- Make sure
  - Your file is available and accessible via http:// or hdfs://
- Setup the application’s JobConf in your Driver class
  - `DistributeCache.addFileToClasspath(new Path("/usr/datafile/XYZ"))`

Size of DistributedCache in Hadoop

- Size
  - Default size of the Hadoop distributed cache is 10GB
- 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
    - (UserID, record) pairs are stored in a HashMap for data retrieval during the map process
  - Map phase
    - For each input record (from the large dataset), the user information is retrieved from the HashMap
    - Assemble a joined record
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

MapReduce Design Patterns IV: Join Patterns

3. Composite Join

Structure of the composite join pattern

- Partitioned and sorted datasets
  - Each input dataset must be partitioned and sorted in a specific way and divided into the same number of partitions
Questions?