FAQs

- Q2 (MapReduce)
  - Your score has been posted in canvas
  - combiner

- How to improve your MapReduce programing skills
  - Programming assignment
  - https://github.com/adamjshook/mapreducepatterns
  - Download, read, and run the code!

Hadoop Combiner

- Minimizes the data transferred between map and reduce tasks
- Users can specify a combiner function
- To be run on the map output
- To replace the map output with the combiner output
- Hadoop does NOT guarantee how many times it will call combiner for a particular map output record
- The function should be **cumulative and associative**

Example: Find the maximum temperature

- First map produces (1950, 0) (1950, 20) (1950, 10) → (1950, 20)
- Input to the reduce function (1950, [0, 20, 10, 25, 15])
- Output (1950, 25)
Topics

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

Partitioner [1/4]

- Partitions the key-value pairs of intermediate Map-outputs
- Uses a user-defined condition
- e.g. Process the input dataset to find the highest salaried employee by gender in different age groups

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Gender</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>James</td>
<td>45</td>
<td>Male</td>
<td>50000</td>
</tr>
<tr>
<td>Matthew</td>
<td>30</td>
<td>Male</td>
<td>45000</td>
</tr>
<tr>
<td>Kevin</td>
<td>34</td>
<td>Male</td>
<td>43000</td>
</tr>
<tr>
<td>Maria</td>
<td>20</td>
<td>Female</td>
<td>23000</td>
</tr>
<tr>
<td>Julia</td>
<td>25</td>
<td>Female</td>
<td>42000</td>
</tr>
<tr>
<td>Lavanya</td>
<td>19</td>
<td>Female</td>
<td>19000</td>
</tr>
<tr>
<td>Joseph</td>
<td>22</td>
<td>Male</td>
<td>32000</td>
</tr>
<tr>
<td>Steve</td>
<td>24</td>
<td>Male</td>
<td>33000</td>
</tr>
<tr>
<td>Max</td>
<td>28</td>
<td>Male</td>
<td>37000</td>
</tr>
<tr>
<td>Jenifer</td>
<td>18</td>
<td>Female</td>
<td>19000</td>
</tr>
<tr>
<td>Nick</td>
<td>33</td>
<td>Male</td>
<td>56000</td>
</tr>
<tr>
<td>MaryAnn</td>
<td>39</td>
<td>Female</td>
<td>65000</td>
</tr>
</tbody>
</table>

Partitioner [2/4]

- Map Tasks
  - Input
    - Dummy key data
    - (Dummy_key, "1201 \ t James \ t 45 \ t male \ t 50000")
  - Functionality
    - Read the value and extract gender information
      ```java
      String str = value.toString();
      String gender = str.split(\"\t\", -3)[3];
      context.write(new Text(gender), value);
      ```
  - Output
    - Gender data and value

Partitioner [3/4]

- Partitioner Task
  - Dividing the data from the map task into segments
  - Uses a user-defined condition
  - Method
    - Read the age field and apply conditions
  - Output
    - The data of key-value pairs are segmented into three collections of key-value pairs

```java
int age = Integer.parseInt(str[2]);
if(age<=20) {
    return 0;
} else if(age>20 && age<=30) {
    return 1 % numReduceTasks;
} else {
    return 2 % numReduceTasks;
}
```

Partitioner [4/4]

- Reduce Task
  - The number of partitions (segments) is equal to the number of reduce tasks
  - The reducer will execute three times (in our example) with different collection of key-value pairs
Driver Code (last access date partitioner)

Sort the user activities based on the last access year using a partitioner
original data contains the timestamps of the last access
(e.g. YYYY:HH:MM:SS)

```java
// 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 (last access date partitioner)

```java
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 (last access date partitioner)

```java
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);
    }
}
```

Reducer Code (last access date partitioner)

```java
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());
        }
    }
}
```

Unevenly distributed partitions

- **Observation**
  - Recent years will have more users
  - Provide finer grained segmentations to the recent years
  - e.g. Monthly partitions for recent 3 years
MapReduce Design Patterns II: Filtering Patterns
3. Total Order Sorting Pattern

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
    - 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
  - 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

TeraSort Benchmark
- The most well-known Hadoop benchmark
  - In 2008, Yahoo! Set a record by sorting 1 TB of data in 209 seconds
    - Hadoop cluster with 910 nodes
    - Owen O’Malley of the Yahoo!
  - In 2009, Yahoo! Sorted 1PB of data in 16 hours
    - Hadoop cluster of 3800 nodes
      - For 1TB, it took 62 seconds.
TeraSort Benchmark APIs

- TeraGen
  - MR to generate the data
- TeraSort
  - Samples the input data and uses MR to sort the data into a total order
- TeraValidate
  - MR that validates the output
- TeraSort is a standard MapReduce with a custom partitioner that uses a sorted list of N-1 sorted sampled keys that define the key range for each reduce
- \( \text{sample}[i..j] \leq \text{key} < \text{sample}[i] \) are sent to reducer \( i \)
- Total 1,200 lines of java code

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

 ejemplo

<table>
<thead>
<tr>
<th>UserID</th>
<th>Reputation</th>
<th>Location</th>
<th>Text</th>
<th>PostID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3748</td>
<td>New York, NY</td>
<td>Not sure why this is getting downvoted</td>
<td>35314</td>
</tr>
<tr>
<td>3</td>
<td>99990</td>
<td>New York, NY</td>
<td>Hey, of course, it’s all true</td>
<td>48002</td>
</tr>
<tr>
<td>4</td>
<td>12946</td>
<td>New York, NY</td>
<td>Please see my post below</td>
<td>48002</td>
</tr>
<tr>
<td>5</td>
<td>5443</td>
<td>San Diego, CA</td>
<td>Thank you very much for your reply</td>
<td>44921</td>
</tr>
<tr>
<td>6</td>
<td>99676</td>
<td>Oakland, CA</td>
<td>HTML is not a subset of XML</td>
<td>44920</td>
</tr>
</tbody>
</table>

Inner Join

<table>
<thead>
<tr>
<th>UserID</th>
<th>Reputation</th>
<th>Location</th>
<th>Text</th>
<th>PostID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3748</td>
<td>New York, NY</td>
<td>Not sure why this is getting downvoted</td>
<td>35314</td>
</tr>
<tr>
<td>3</td>
<td>99990</td>
<td>New York, NY</td>
<td>Hey, of course, it’s all true</td>
<td>48002</td>
</tr>
<tr>
<td>4</td>
<td>12946</td>
<td>New York, NY</td>
<td>Please see my post below</td>
<td>48002</td>
</tr>
<tr>
<td>5</td>
<td>5443</td>
<td>San Diego, CA</td>
<td>Thank you very much for your reply</td>
<td>44921</td>
</tr>
<tr>
<td>6</td>
<td>99676</td>
<td>Oakland, CA</td>
<td>HTML is not a subset of XML</td>
<td>44920</td>
</tr>
</tbody>
</table>

Default join style

Records from both A and B that contain identical values for the given foreign key are brought together

<table>
<thead>
<tr>
<th>A userid</th>
<th>A.Reputation</th>
<th>A.Location</th>
<th>B.userid</th>
<th>B.PostID</th>
<th>B.Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3748</td>
<td>New York, NY</td>
<td>3</td>
<td>48002</td>
<td>Hey, of course, it’s all true</td>
</tr>
<tr>
<td>3</td>
<td>99990</td>
<td>New York, NY</td>
<td>4</td>
<td>48002</td>
<td>Please see my post below</td>
</tr>
<tr>
<td>4</td>
<td>12946</td>
<td>New York, NY</td>
<td>5</td>
<td>44921</td>
<td>Thank you very much for your reply</td>
</tr>
</tbody>
</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

Anti join of A+B on UserID

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
Performance analysis

- The reduce 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
// The first two elements of the args array are the two inputs
MultipleInputs.addInputPath(job, new Path(args[0]));
TextInputFormat.class, UserJoinMapper.class);
MultipleInputs.addInputPath( job, new Path(args[1]));
TextInputFormat.class, CommentJoinMapper.class);
job.setConfiguration();
...
```

User Mapper Code

```java
public static class UserJoinMapper extends Mapper < Object, Text, Text, Text > {
    public void map(Object key, Text value, Context context) throws IOException,
                  InterruptedException {
        private Text outkey = new Text();
        private Text outvalue = new Text();
        private ArrayList< Text > listA = new ArrayList< >();
        private ArrayList< Text > listB = new ArrayList< >();
        String joinType = null;
        try {
            Map < String, String > parsed = MRDPUtils.transformXmlToMap(value.toString());
            userId = parsed.get("UserId");
            outvalue.set(userId);
            outkey.set(userId + "Id");
        } catch (IOException e) {
            return;
        }        
        if (tmp.charAt(0) == 'A') {
            listA.add(tmp.toString());
        } else if (tmp.charAt(0) == 'B') {
            listB.add(tmp.toString());
        }
    }
    public void setup(Context context) {
        private String joinType = context.getConfiguration().get("join.type");
    }
    public void reduce(Text key, Text value, Context context) throws IOException,
                      InterruptedException {
        private Text outkey = new Text();
        private Text outvalue = new Text();
        String joinType = context.getConfiguration().get("join.type");
        if (joinType == "inner") {
            for (String sA : listA) {
                for (String sB : listB) {
                    if (parsed.get("A" + value.toString()).equals(sA) && parsed.get("B" + value.toString()).equals(sB)) {
                        outkey.set(sA + sB);
                        outvalue.set(sA + sB);
                        context.write(outkey, outvalue);
                    }
                }
            }
        } else if (joinType == "left") {
            for (String sA : listA) {
                outkey.set(sA + userId);
                outvalue.set(userId);
                context.write(outkey, outvalue);
            }
        } else if (joinType == "right") {
            for (String sB : listB) {
                outkey.set(userId + sB);
                outvalue.set(userId);
                context.write(outkey, outvalue);
            }
        }
    }
}
```

Reducer Code

```java
public static class UserJoinReducer extends Reducer < Text, Text, Text, Text > {
    public void reduce(Text key, Text value, Context context) throws IOException,
                      InterruptedException {
        private Text outkey = new Text();
        private Text outvalue = new Text();
        final String joinType = context.getConfiguration().get("join.type");
        if (joinType == "inner") {
            String[] pairs = value.toString().split(" ");
            int count = 0;
            for (String s : pairs) {
                if (parsed.get("A" + s).equals(userId)) {
                    count++;
                }
            }
            if (count == 0) {
                return;
            }
            outvalue.set(userId);
            outkey.set(userId + key.toString());
            context.write(outkey, outvalue);
        } else if (joinType == "left") {
            String[] pairs = value.toString().split(" ");
            for (String s : pairs) {
                try {
                    outvalue.set(s);
                    outkey.set(userId + s);
                    context.write(outkey, outvalue);
                } catch (IOException e) {
                    return;
                }
            }
        } else if (joinType == "right") {
            String[] pairs = value.toString().split(" ");
            for (String s : pairs) {
                try {
                    outvalue.set(s);
                    outkey.set(s + userId);
                    context.write(outkey, outvalue);
                } catch (IOException e) {
                    return;
                }
            }
        }
    }
    public void setup(Context context) {
        private String joinType = context.getConfiguration().get("join.type");
        if (joinType == "inner") {
            context.getConfiguration().set("join.type", "left");
        }
    }
}
```

Comment mapper code

```java
public static class CommentJoinMapper extends Mapper < Object, Text, Text, Text > {
    public void map(Object key, Text value, Context context) throws IOException,
                  InterruptedException {
        private Text outkey = new Text();
        private Text outvalue = new Text();
        String joinType = null;
        try {
            Map < String, String > parsed = MRDPUtils.transformXmlToMap(value.toString());
            userId = parsed.get("UserId");
            outvalue.set(userId);
            outkey.set(userId + "Id");
        } catch (IOException e) {
            return;
        }        
        if (tmp.charAt(0) == 'A') {
            listA.add(tmp.toString());
        } else if (tmp.charAt(0) == 'B') {
            listB.add(tmp.toString());
        }
    }
    public void setup(Context context) {
        private String joinType = context.getConfiguration().get("join.type");
        if (joinType == "inner") {
            context.getConfiguration().set("join.type", "left");
        }
    }
    public void reduce(Text key, Text value, Context context) throws IOException,
                      InterruptedException {
        private Text outkey = new Text();
        private Text outvalue = new Text();
        String joinType = context.getConfiguration().get("join.type");
        if (joinType == "inner") {
            String[] pairs = value.toString().split(" ");
            int count = 0;
            for (String s : pairs) {
                if (parsed.get("A" + s).equals(userId)) {
                    count++;
                }
            }
            if (count == 0) {
                return;
            }
            outvalue.set(userId);
            outkey.set(userId + key.toString());
            context.write(outkey, outvalue);
        } else if (joinType == "left") {
            String[] pairs = value.toString().split(" ");
            for (String s : pairs) {
                try {
                    outvalue.set(s);
                    outkey.set(userId + s);
                    context.write(outkey, outvalue);
                } catch (IOException e) {
                    return;
                }
            }
        } else if (joinType == "right") {
            String[] pairs = value.toString().split(" ");
            for (String s : pairs) {
                try {
                    outvalue.set(s);
                    outkey.set(s + userId);
                    context.write(outkey, outvalue);
                } catch (IOException e) {
                    return;
                }
            }
        }
    }
}
```
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 to 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

Inner Join Code

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

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

... 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);
    }
  }
} ...
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

Size of DistributedCache in Hadoop

- Size
  - 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

Working with DistributedCache

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

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
    - (user ID, records) 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

Mapper Code

```java
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());
    }
}
```

Mapper Code

```java
public static class ReplicatedJoinMapper extends Mapper < Object, Text, Text, Text > {
    private static final Text EMPTY_TEXT = new Text(""");
    private static final Text ERROR_TEXT = new Text("ERROR");
    private static final Text USER_ID = new Text("User ID");
    private static final Text RECORD = new Text("Record");
    private static final Text JOIN_TYPE = new Text("Join Type");
    private static final Text ERROR = new Text("Error");
    private static final Text SUCCESS = new Text("Success");
    private static final Text EXIT = new Text("Exit");
    private static final Text FILE_NAME = new Text("FileName");
    private static final Text LINE_NUMBER = new Text("Line Number");
    private static final Text FILE_LINE = new Text("File Line");
    private static final Text RECORD_COUNT = new Text("Record Count");
    private static final Text COMPLETED_RECORDS = new Text("Completed Records");
    private static final Text IDENTITY = new Text("Identity");
    private static final Text IDENTITY_TYPE = new Text("Identity Type");
    private static final Text IDENTITY_VALUE = new Text("Identity Value");
    private static final Text IDENTITY_COUNT = new Text("Identity Count");
    private static final Text IDENTITY_RECORDS = new Text("Identity Records");
    private static final Text IDENTITY_GROUP = new Text("Identity Group");
    private static final Text IDENTITY_SUBGROUP = new Text("Identity Subgroup");
    private static final Text IDENTITY_SUBSUBGROUP =
```
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