Big Data
- MR Advanced
- Development

Dr. Qing "Matt" Zhang
ITU

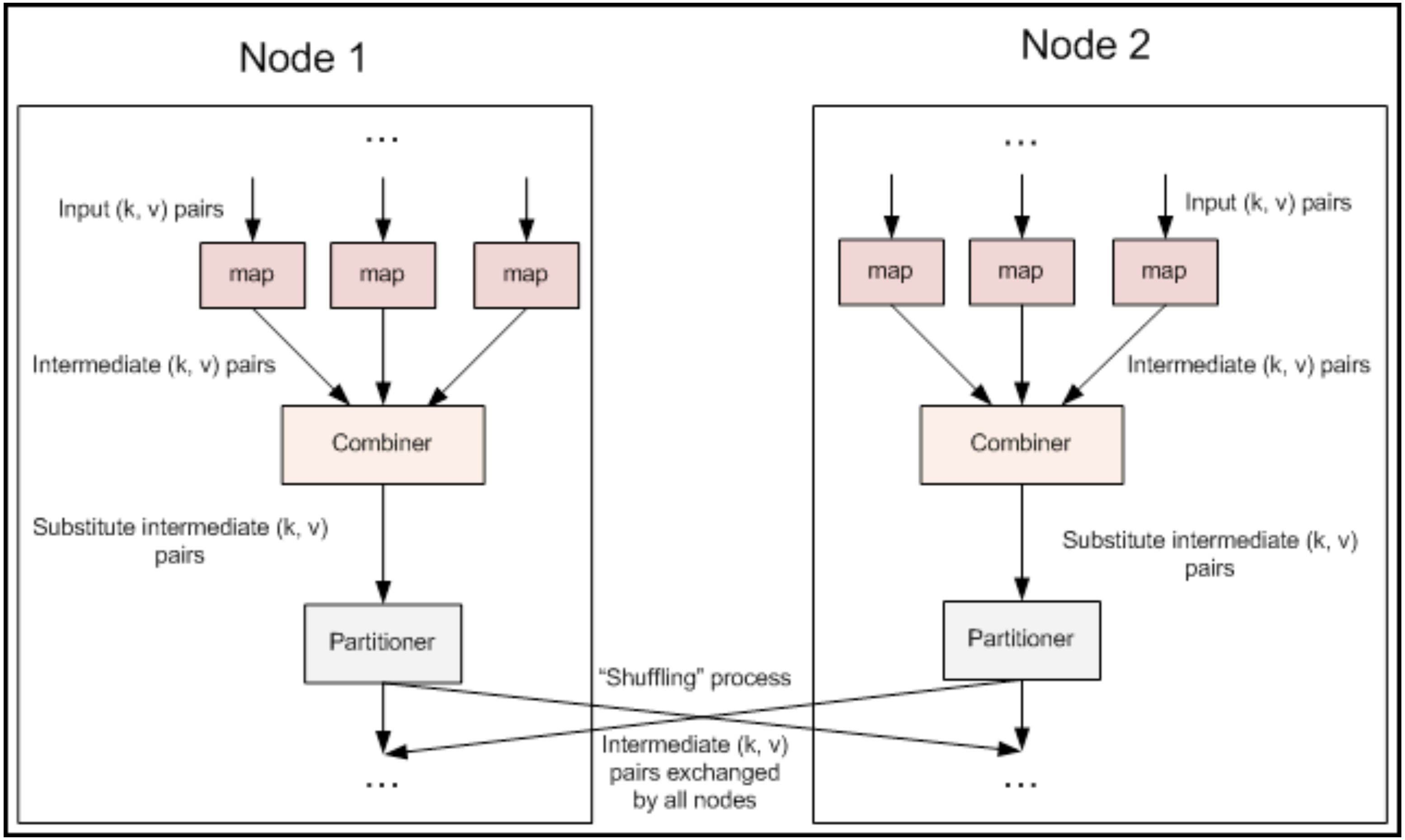
Outline

- **Combiner**
- Custom Partitioner
- Accessing HDFS
- Distributed Cache
- Sorting
- Searching
- Inverted Index
- Word Co-Occurrence
- Join

Combiner

- Mappers often produce large amounts of intermediate data
  - That data must be passed to the Reducers
  - This can result in a lot of network traffic
- Add a Combiner!
  - Like a ‘mini/Reducer’
  - Runs locally on a single Mapper’s output
  - Output from the Combiner is sent to the Reducers
  - Input and output data types for the Combiner/Reducer must be identical
- Combiner and Reducer code are often identical
  - Technically, this is possible if the operation performed is commutative and associative

Combiner



Node 1
Node 2
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes

WordCount Re-Visit

```
map(String input_key, String input_value)
    foreach word w in input_value:
        emit(w, 1)
```

```
reduce(String output_key,
                    Iterator<int> intermediate_vals)
    set count = 0
    foreach v in intermediate_vals:
        count += v
    emit(output_key, count)
```

WordCount Re-Visit

- Input to the Mapper

```
(3414, 'the cat sat on the mat')
(3437, 'the aardvark sat on the sofa')
```

- Output from the Mapper

```
('the', 1), ('cat', 1), ('sat', 1), ('on', 1),
('the', 1), ('mat', 1), ('the', 1), ('aardvark', 1),
('sat', 1), ('on', 1), ('the', 1), ('sofa', 1)
```

WordCount Re-Visit

- Intermediate data sent to reducer

```
('aardvark', [1])
('cat', [1])
('mat', [1])
('on', [1, 1])
('sat', [1, 1])
('sofa', [1])
('the', [1, 1, 1, 1])
```

- Reducer output

```
('aardvark', 1)
('cat', 1)
('mat', 1)
('on', 2)
('sat', 2)
('sofa', 1)
('the', 4)
```

WordCount with Combiner

- Intermediate data sent to Reducer after a Combiner, using the same code as Reducer:

```
('aardvark', [1])
('cat', [1])
('mat', [1])
('on', [2])
('sat', [2])
('sofa', [1])
('the', [4])
```

Combiner

- Aggregate intermediate map output locally on individual mapper outputs
  - Decrease the amount of data sent to reducers
  - Decrease the amount of network traffic
  - Decrease the amount of work at Reducer
  - Often use the same code as Reducer

Use a Combiner

- To specify the Combiner class to be used in your MapReduce code, put the following line in your driver code:

  job.setCombinerClass(MyCombiner.class);

- The Combiner uses the same interface as the Reducer
  - Takes in a key and a list of values
  - Outputs zero or more (key, value) pairs
  - The actual method called is the reduce method in the class

Combiner call

- The Combiner may run once, or more than once, on the output from any given Mapper
  - Do not put code in the Combiner which could influence your results if it runs more than once
  - Won't run if map output is empty, or is a single pair

Outline

- Combiner
- **Custom Partitioner**
- Accessing HDFS
- Distributed Cache
- Sorting
- Searching
- Inverted Index
- Word Co-Occurrence

Partitioner

- Partitioner controls what intermediate data is sent to which reducer
  - After the map phase and before the reduce phase
- Number of partitions equals the number of reducers
  - All data in the same partition is processed by a single reducer

Default Partitioner

- Default partitioning function is the HashPartitioner
  - Hash data based on the key field

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {

    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

Custom Partitioner

- Sometimes you will need to write your own Partitioner
- Example: your key is a custom *WritableComparable* which contains a pair of values (a, b)
  - You may decide that all keys with the same value for a need to go to the same Reducer
  - The default Partitioner is not sufficient in this case

More Partitioner Scenarios

- Custom partitioners are needed for secondary sort
- Custom partitioners can help performance
  - Balance reducer loads
    - Example: in WordCount job, we wouldn't want a single Reducer dealing with all the three- and four-letter words, while another only had to handle 10- and 11- letter words

Write Custom Partitioner

- Steps to create a custom Partitioner:
  - Create a class for the Partitioner, which extends Partitioner
  - Create a method in the class, called getPartition
    - Receives the key, the value, and the number of Reducers
    - Should return an integer number in the range of

      [0, number of Reducers -1]
  - Specify the custom Partitioner in your driver code

    job.setPartitionerClass(MyPartitioner.class);

Setting up variables for Partitioner

- If you need to set up variables for use in your Partitioner, it should implement *Configurable*

```
class MyPartitioner extends Partitioner<K, V> implements Configurable {

    private Configuration configuration;
    // Define your own variables here

    @Override
    public void setConf(Configuration configuration) {
        this.configuration = configuration;
        // Set up your variables here
    }
    @Override
    public Configuration getConf() {
        return configuration;
    }
    ...
}
```

Setting up variables for Partitioner

- If a Hadoop object implements Configurable, its setConf() method will be called once, when it is instantiated
- You can therefore set up variables in the setConf() method which your getPartition() method will then be able to access

Outline

- Combiner
- Custom Partitioner
- **Accessing HDFS**
- Distributed Cache
- Sorting
- Searching
- Inverted Index
- Word Co-Occurrence

Accessing HDFS Programmatically

- In addition to using the command line shell, you can access HDFS programmatically
  - Useful if your code needs to read or write 'side data' in addition to the standard MapReduce inputs and outputs
  - Or for programs outside of Hadoop which need to read the results of MapReduce jobs
- HDFS is not a general purpose file system!
  - Files cannot be modified once they have been written
- Hadoop provides the FileSystem abstract base class
  - Provides an API to generic file systems
    - Could be HDFS, or your local file system
    - Or others like Amazon S3

FileSystem API

- Create an instance of the FileSystem API:

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
```

- The conf object has read in the Hadoop configuration files, and therefore knows the address of the NameNode etc.
- A file in HDFS is represented by a Path object

```
Path p = new Path("/path/to/my/file");
```

FileSystem API(cont'd)

- Useful API methods:
  - FSDataOutputStream create(...)
    - Extends java.io.DataOutputStream
    - Provides methods for writing primitives, raw bytes etc
  - FSDataInputStream open(...)
    - Extends java.io.DataInputStream
    - Provides methods for reading primitives, raw bytes, etc
  - boolean delete(...)
  - boolean mkdirs(...)
  - void copyFromLocalFile(...)
  - void copyToLocalFile(...)
  - FileStatus[] listStatus(...)

Example: list directory

```
Path p = new Path("/my/path");

Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
FileStatus[] fileStats = fs.listStatus(p);

for (int i = 0; i < fileStats.length; i++) {
    Path f = fileStats[i].getPath();

    // do something interesting
}
```

Example: Writing data to file

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);

Path p = new Path("/my/path/foo");

FSDataOutputStream out = fs.create(path, false);

// write some raw bytes
out.write(getBytes());

// write an int
out.writeInt(getInt());

...

out.close();
```

Outline

- Combiner
- Custom Partitioner
- Accessing HDFS
- **Distributed Cache**
- Sorting
- Searching
- Inverted Index
- Word Co-Occurrence

Application Requirement

- A common requirement is for a Mapper or Reducer to need access to some 'side data'
  - Lookup tables
  - Dictionaries
  - Standard configuration values

Distributed Cache

- The Distributed Cache provides an API to push data to all slave nodes
  - Transfer happens behind the scenes before any task is executed
  - Files are only copied once per job
  - It can cache archives which are un-archived on the slaves.
  - Distributed Cache is read-only
    - It tracks modification timestamps of the cache files.
    - The cache files should not be modified by the application or externally while the job is executing.
  - Files in the Distributed Cache are automatically deleted from slave nodes when the job finishes

Using the Distributed Cache

- Place the files in HDFS first

```
$ bin/hadoop fs -copyFromLocal lookup.dat /myapp/lookup.dat
$ bin/hadoop fs -copyFromLocal map.zip /myapp/map.zip
$ bin/hadoop fs -copyFromLocal mylib.jar /myapp/mylib.jar
$ bin/hadoop fs -copyFromLocal mytar.tar /myapp/mytar.tar
$ bin/hadoop fs -copyFromLocal mytgz.tgz /myapp/mytgz.tgz
$ bin/hadoop fs -copyFromLocal mytargz.tar.gz /myapp/mytargz.tar.gz
```

Setup the Distributed Cache

- Setup Application's JobConf in the driver code:

```
JobConf job = new JobConf();
DistributedCache.addCacheFile(new URI("/myapp/lookup.dat#lookup.dat"),
                              job);
DistributedCache.addCacheArchive(new URI("/myapp/map.zip", job);
DistributedCache.addFileToClassPath(new Path("/myapp/mylib.jar"), job);
DistributedCache.addCacheArchive(new URI("/myapp/mytar.tar", job);
DistributedCache.addCacheArchive(new URI("/myapp/mytgz.tgz", job);
DistributedCache.addCacheArchive(new URI("/myapp/mytargz.tar.gz", job);
```

  - *.jar* files added with *addFileToClassPath* will be added to your Mapper or Reducer's classpath
  - Files added with *addCacheArchive* will automatically be dearchived/decompressed

Access the Cached Files

- Files added to the Distributed Cache are made available in your task's local working directory
  - Access them from your Mapper or Reducer the way you would read any ordinary local file

File f = new File(" cached_file_name");

Using the Distributed Cache (cont'd)

- Use the cached file in Mapper or Reducer

```
public static class MapClass extends MapReduceBase
implements Mapper<K, V, K, V> {

  private Path[] localArchives;
  private Path[] localFiles;

  public void configure(JobConf job) {
    // Get the cached archives/files
    File f = new File("./map.zip/some/file/in/zip.txt");
  }

  public void map(K key, V value,
                  OutputCollector<K, V> output, Reporter reporter)
  throws IOException {
    // Use data from the cached archives/files here
    // ...
    // ...
    output.collect(k, v);
  }
}
```

Easy Way: Use ToolRunner

- You can add files to the Distributed Cache directly from the command line when you run the job
  - No need to copy the files to HDFS first
  - Use the -files option to add files

  hadoop jar myjar.jar MyDriver -files file1, file2, file3, ...

- The *-archives* flag adds archived files, and automatically unarchives them on the destination machines
- The *-libjars* flag adds jar files to the classpath

Outline

- Combiner
- Custom Partitioner
- Accessing HDFS
- Distributed Cache
- **Sorting**
- Searching
- Inverted Index
- Word Co-Occurrence

Sorting

- MapReduce is very well suited to sorting large data sets
  - Recall: keys are passed to the Reducer in sorted order
- Assuming the file to be sorted contains lines with a single value:
  - Mapper is merely the identity function for the value

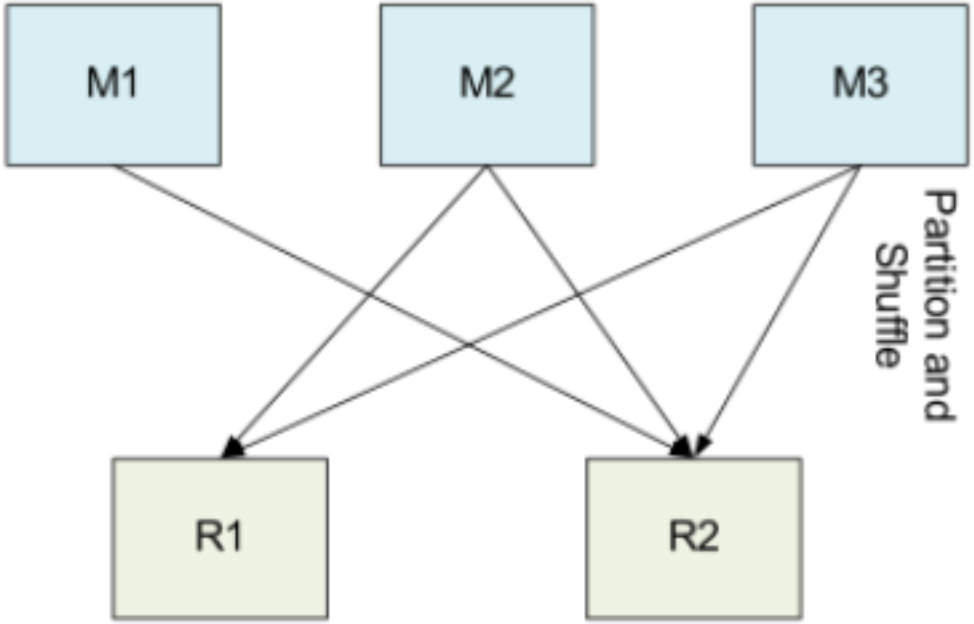
    (k, v) -> (v, _)
  - Reducer is the identity function

    (k, _) -> (k, '')

Sorting (cont'd)

- Trivial with a single Reducer
  - For multiple Reducers, need to choose a partitioning function such that if

$$k1 < k2,\ partition(k1) <= partition(k2)$$



Sorting as a Speed Test of Hadoop

- Sorting is frequently used as a speed test for a Hadoop cluster
  - Mapper and Reducer are trivial
    - Therefore sorting is effectively testing the Hadoop framework's I/O
- Good way to measure the increase in performance if you enlarge your cluster
  - Run and time a sort job before and after you add more nodes
  - terasort is one of the sample jobs provided with Hadoop
    - Creates and sorts very large files

Outline

- Combiner
- Custom Partitioner
- Accessing HDFS
- Distributed Cache
- Sorting
- **Searching**
- Inverted Index
- Word Co-Occurrence

Searching

- Assume the input is a set of files containing lines of text
- Assume the Mapper has been passed the pattern for which to search as a special parameter
  - pattern may be saved in distributed cache files

Searching (cont'd)

- Algorithm:
  - Mapper compares the line against the pattern
  - If the pattern matches, Mapper outputs (line, _)
    - Or (filename + line#, _), or …
  - If the pattern does not match, Mapper outputs nothing
  - Reducer is the Identity Reducer
    - The intermediate result from the Mapper is already the result

Outline

- Combiner
- Custom Partitioner
- Accessing HDFS
- Distributed Cache
- Sorting
- Searching
- **Inverted Index**
- Word Co-Occurrence

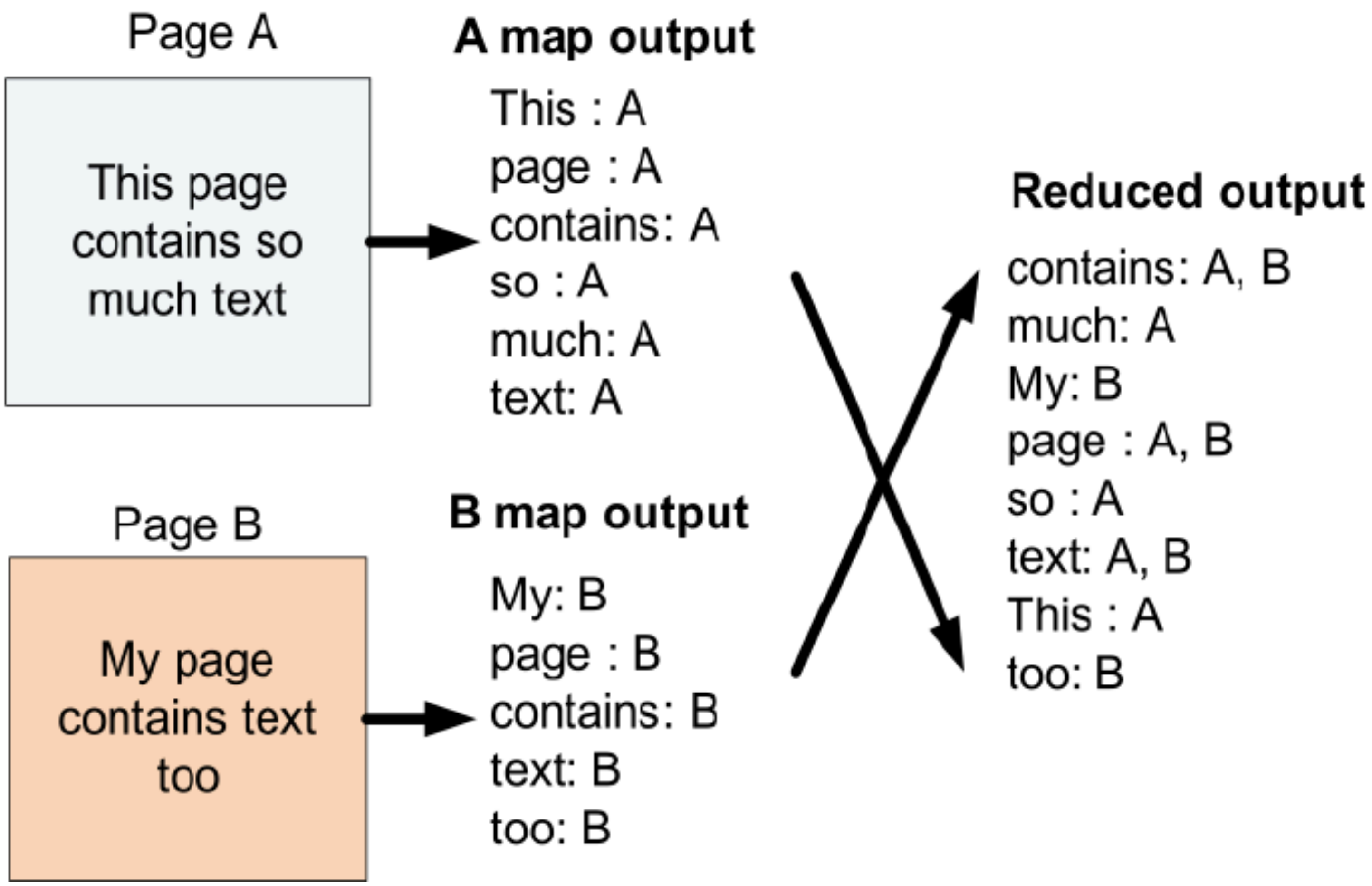
Inverted Index

- Very useful for almost all information retrieval applications
- Input is a set of files, containing lines of text
- Output is each word, with the corresponding files it's in
  - You may also include line # etc.

Inverted Index Algorithm

- Mapper:
  - For each word in the line, emit (word, filename)
- Reducer:
  - Identity function
    - Collect together all values for a given key (i.e., all filenames for a particular word)

Inverted Index



Outline

- Combiner
- Custom Partitioner
- Accessing HDFS
- Distributed Cache
- Sorting
- Searching
- Inverted Index
- **Word Co-Occurrence**

Word Co-Occurrence Algorithm

- Mapper

```
map(docid a, doc d) {
    foreach w in d do
        foreach u near w do
            emit(pair(w, u), 1)
}
```

- Reducer

```
reduce(pair p, Iterator counts) {
    s = 0
    foreach c in counts do
        s += c
    emit(p, s)
}
```

Word Co-Occurrence

- Word Co-Occurrence measures the frequency with which two words appear close to each other in a corpus of documents
  - For some definition of ‘close’
- This is at the heart of many data mining techniques
  - Provides results for “people who did this, also do that”
  - Examples:
    - Shopping recommendations
    - Credit risk analysis
    - Identifying ‘people of interest’

Join Data Sets in MR Jobs

- We frequently need to join data together from two sources as part of a MapReduce job, such as
  - Lookup tables
  - Data from database tables
- There are two fundamental approaches:
  - Map-side joins
  - Reduce- side joins
- Map-side joins are easier to write, but have potential scaling issues

MR Joins

- Avoid writing joins in Java MapReduce if you can!
- Abstractions such as Pig and Hive are much easier to use
  - Save hours of programming
- If you are dealing with text based data, there really is no reason not to use Pig or Hive

Map-Side Joins

- Basic idea for Map-side joins:
  - Load one set of data into memory, stored in a hash table
  - Key of the hash table is the join key
  - Map over the other set of data, and perform a lookup on the hash table using the join key
- If the join key is found, you have a successful join
  - Otherwise, do nothing

Problems with Map-Side Joins

- Map-side joins have scalability issues
  - The associative array may become too large to fit in memory
- Possible solution: break one data set into smaller pieces
  - Load each piece into memory individually, mapping over the second data set each time
  - Then combine the result sets together

Reduce-Side Joins

- For a Reduce-side join, the basic concept is:
  - Map over both data sets
  - Emit a (key, value) pair for each record
    - Key is the join key, value is the entire record
  - In the Reducer, do the actual join
  - Because of the Shuffle and Sort, values with the same key are brought together

Reduce-Side Joins: Example

- Example input data:
  - we may have two data sets, one of them contains employee information:
    - EMP: 42, Aaron, loc(13)
  - The other contains location coding:
    - LOC: 13, New York City
- Required output (join on loc_id)
  - EMP: 42, Aaron, loc(13), New York City

Example Record Data Structure

- This is data structure constructed by mapper, created for each input data set:

```
class Record {
   enum Typ { emp, loc };
   Typ type;

   String empName;
   int empId;
   int locId;
   String locationName;
}
```

Reduce-Side Join: Mapper

- Mapper will transform each input data set into the Record data type defined earlier

```
void map(k, v) {
  Record r = parse(v);
  emit (r.locId, r);
}
```

Reduce-Side Join: Reducer

```
void reduce(k, values) {
  Record thisLocation;
  List<Record> employees;

  for (Record v in values) {
    if (v.type == Typ.loc) {
      thisLocation = v;
    } else {
      employees.add(v);
    }
  }
  for (Record e in employees) {
    e.locationName = thisLocation.locationName;
    emit(e);
  }
}
```

Reduce-Side Join: Reducer

- For each loc_id, we'll have only one record for that
- If the incoming data is a location data, we'll update it to *thisLocation*
- If the incoming data is an employee data, we'll save it to a list of Record
- We then update all employees in the above Record list, and set the location to *thisLocation*

Scalability Problems

- All employees for a given location must potentially be buffered in the Reducer
  - The employee Record list in previous example
  - Could result in out-of-memory errors for large data sets
- Solution: Ensure the location record is the first one to arrive at the Reducer
  - Using a Secondary Sort (not covered in this course)