CPSC 4330: Big Data Analytics

Common MapReduce Algorithms

Common MapReduce Algorithms

- Sort and search large data sets
- Build Inverted Index
- Calculate word co-occurrence
- Compute term frequency – inverse document frequency (TF-IDF)
- Perform a secondary sort

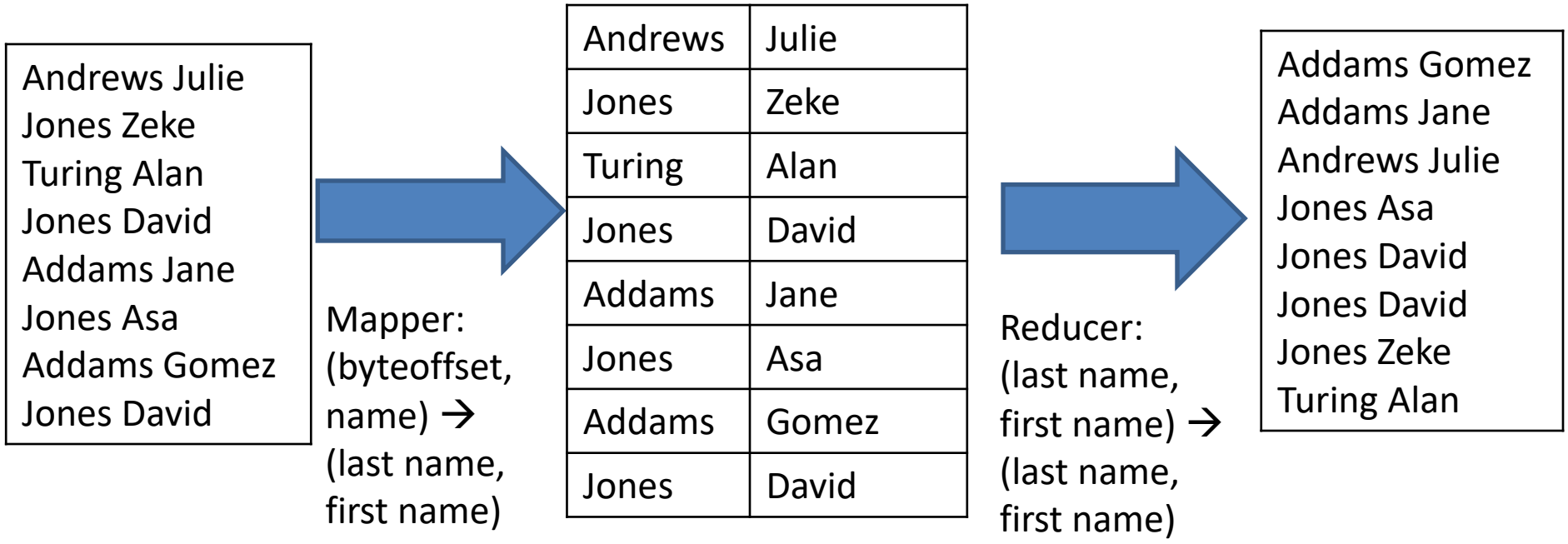
Introduction

- MapReduce jobs tend to be relatively short in terms of lines of code
- It is typical to combine multiple small MapReduce jobs together in a single workflow
- You are likely to find that many of your MapReduce jobs use very similar code
- The common MapReduce algorithms are frequently the basis for more complex MapReduce jobs

Sorting Large Data Sets (1)

- MapReduce is suited to sorting large data sets

Example: How to sort alphabetically by Last Name?

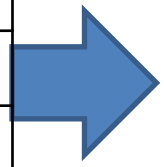


Sorting Large Data Sets (2)

- Example: Sort by salary of employee

Andrews Julie, 120,000
Jones Zeke, 150,000
Turing Alan, 200,000
Jones David, 160,000
Addams Jane, 90,000
Jones Asa, 85,000
Addams Gomez, 250,000
Jones David, 180,000

120,000	Andrews Julie
150,000	Jones Zeke
200,000	Turing Alan
160,000	Jones David
90,000	Addams Jane
85,000	Jones Asa
250,000	Addams Gomez
180,000	Jones David



85,000	Jones Asa
90,000	Addams Jane
120,000	Andrews Julie
150,000	Jones Zeke
160,000	Jones David
180,000	Jones David
200,000	Turing Alan
250,000	Addams Gomez

Mapper:
(name, salary) → (salary, name)

Reducer:
(salary, name) → (salary, name)

Sorting Large Data Sets (3)

- With a single Reducer, a job outputs a single file where all keys are listed in sorted order (by default is in ascending order).
- With multiple Reducers, each output file is sorted. But globally, output files are not totally sorted.

  - need to choose a partitioning function such that if k1 < k2, partition(k1) <= partition(k2)

  - use TotalOrderPartitioner

Searching

- Input of job:
  - A set of files containing lines of text (including line number)
  - A search pattern (e.g. a given word)
- Mapper:
  - Input:
    - ✓ Key: line number
    - ✓ Value: text of line
    - ✓ Search pattern
  - Processing
    - ✓ Compares the line against the pattern
  - Output:
    - ✓ If the pattern matches, outputs (filename+line #,)
    - ✓ If the pattern does not match, outputs nothing

We can retrieve the name of the file using the Context object:

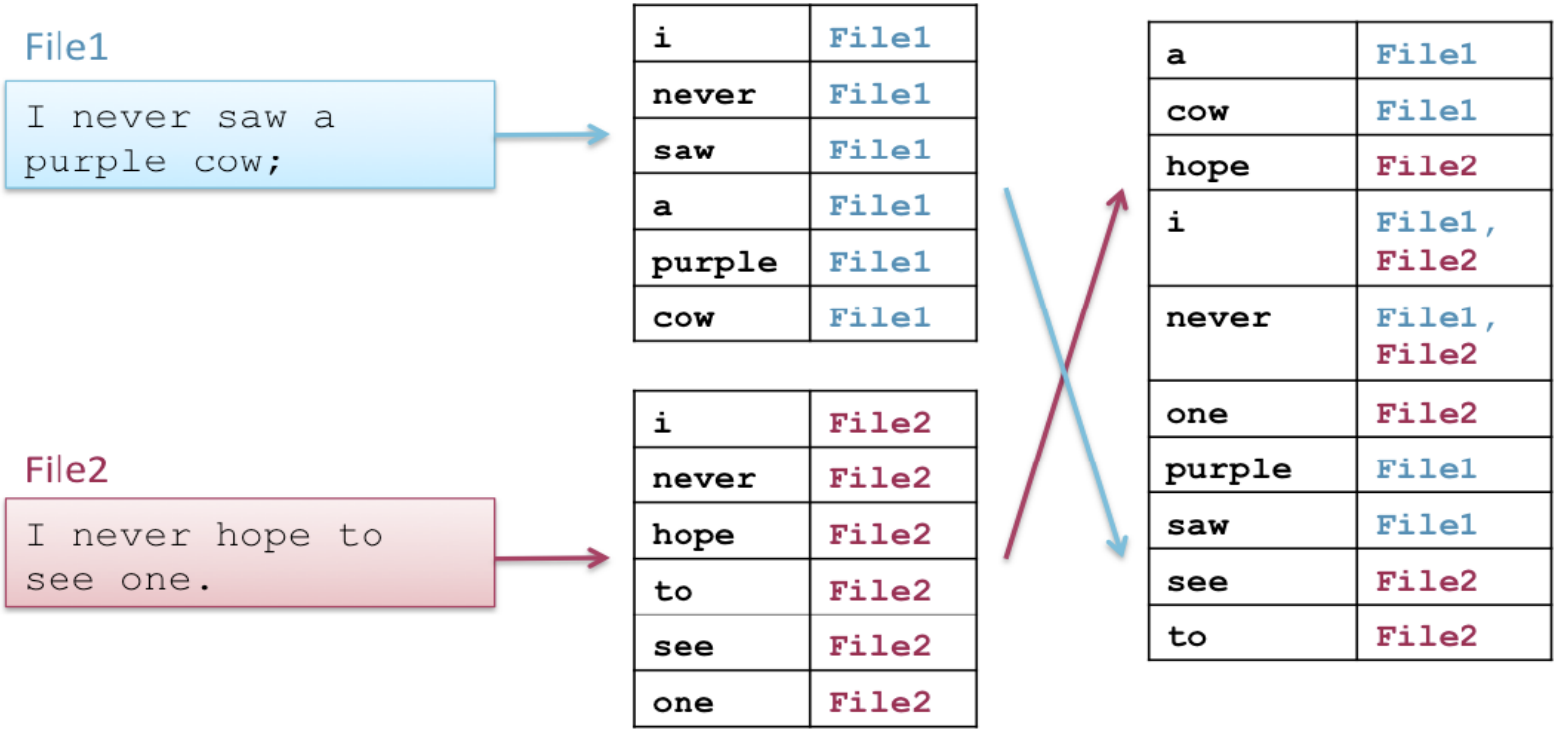
```
FileSplit  fs = (FileSplit) context.getInputSplit();
String  fileName = fs.getPath().getName();
```

Searching (Cont.)

- Reducer
  - Input
    - ✓ (filename+line #,)
  - Processing
    - ✓ Identity Reducer
  - Output:
    - ✓ (filename+line #,)

Inverted Index

- The indexing technique that is normally used in MapReduce is Inverted Index (which is used in search engines)



Inverted Index (Cont.)

- Input of Job: A set of files containing lines of text
- Mapper:
  - Input:
    - ✓ Key: byte offset of the line
    - ✓ Value: text of the line
  - Processing:
    - ✓ For each word in the line, emit (word, filename)
  - Output: (word, filename)
- Reducer:
  - Input:
    - ✓ Key: Word
    - ✓ Value: a list of filename of files that contain the word
  - Processing:
    - ✓ Add a separator (e.g. ",") between each filename
  - Output: (word, filename list)

In-Class Exercise

- Write a MapReduce job that produces an inverted index.

Input: a directory of files; each is a Shakespeare play

```
1    HAMLET
2
3
4    DRAMATIS  PERSONAE
5
6
7    CLAUDIUS    king of Denmark. (KING CLAUDIUS: )
8
9    HAMLET      son to the late, and nephew to the present king.
10
11   POLONIUS    lord chamberlain.  (LORD  POLONIUS:

…
```

Each line contains:
Line number, a tab character, and the line of text

Your program should produce an index of all the words in the text. For each word, the index should have a list of all the locations where the word appears.

honeysuckle 2kinghenryiv@1038, midsummernightdream@2175, …

Discussion

- Input of Mapper?
  - Key: line number; Value: the line of words
- Output of Mapper?
  - key: each word of the line
  - value: filename and line number
- Input of Reducer?
  - key: word
  - value: a list of filenames and line number that contain the word
- Output of Reducer?
  - key: word
  - value: a list of filename and line number that contain the word, separated by the separator (e.g. “,”)

Calculating Word Co-Occurrence

- Word Co-Occurrence

  - measures the frequency with which words appear close to each other in a corpus of documents

Example:

Consider the text “I never saw a purple cow”. With a value of 1, the co-occurrence for the word “saw” would be “never” and “a”. With a value of 2, the co-occurrence for the word “saw” would be “I never”, “a purple”.

Word Co-Occurrence (Cont.)

- Input of Job: A set of files containing lines of text
- Mapper:

  - Input:
    - ✓ Key: byte offset of the line
    - ✓ Value: text of the line
  - Processing:
    - ✓ Split the line to create a string array
    - ✓ Construct a nested loop
      - The outer loop iterates over each word in the array
      - The inner loop iterates over the "neighbors" of the current word and emits a word tuple as key (e.g. consisting of the current word on the left and the neighbor word on the right) and 1 as the value
  - Output: (word tuple, 1)

How many iterations of the inner loop?

- Up to the developers (e.g. look at word immediately adjacent to the current word, or look at 2 words…)

To the left or to the right?

- Up to the developers

Word Co-Occurrence (Cont.)

- Reducer

  - Input:
    - ✓ Key: word tuple
    - ✓ Value: a list of IntWritable (i.e. 1, 1, 1, …)
  - Processing:
    - ✓ Sum up the ones
  - Output:
    - ✓ Key: word tuple
    - ✓ Value: the frequency of the word tuple

Term Frequency – Inverse Document Frequency (TF - IDF)

- Term Frequency – Inverse Document Frequency

- a numerical statistic that reflects how important a word is to a document in a corpus

- assigns a score(weight) to each term(word) in a document

- very commonly used in text-based recommender systems, scoring and ranking a document's relevance given a user query, etc.

TF-IDF: Motivation

- Merely counting the number of occurrences of a word in a document is not a good enough measure of its relevance

  - Some words appear too frequently in all documents to be relevant
    - ✓ known as ‘stopwords’ e.g. a, the, this, to, from, etc

- TF-IDF considers both the frequency of a word in a given document and the number of documents which contain the word

TF-IDF Formally Defined

- Term Frequency (TF)

  - Measures how frequently a particular term occurs in a document

  TF(term) = number of times the term appears in a document/total number of words in the document

- Inverse Document Frequency (IDF)

  - measures the importance of a term

  IDF(term) = log(N/n)
  - N: total number of documents in the corpus
  - n: number of documents that contain the term

- TF-IDF

  TF-IDF = TF × IDF

Example

- Consider a document containing 1000 words, where the term “computer” appears 50 times.
- Assume there are 10 million documents and the word “computer” appears in 1000 of those documents.

TF(“computer”) = 50/1000 = 0.05
IDF(“computer”) = log(10,000,000/1,000) = 4
TF-IDF(“computer”) = 0.05 × 4 = 0.2

Computing TF-IDF

- What we need:

  - Number of times a term T appears in a given document

  - Number of words in a document

  - Number of documents that contain the term T

  - Total number of documents

Computing TF-IDF with MapReduce

- 4 MapReduce jobs

  - Job 1: compute number of times a term T appears in a given document

  - Job 2: compute number of words in a document

  - Job 3: compute number of documents that contain the term T

  - Job 4: compute TF-IDF

chain jobs together such that the output of one job is used as input for the next job

Computing TF-IDF: Job 1

- The input has docid and document contents
- The 'docid' uniquely identifies the document. It will typically be an absolute file path or a URL.

- Mapper

  - Input: (byteoffset, line of text)

  - For each term in the line, generate a (term, docid) pair

    - i.e. we have seen this term in this document once

  - Output: ((term, docid), 1)

- Reducer

  - Input: ((term, docid), a list of 1s)

  - Sums counts for word in document

  - Outputs ((term, docid), n) where n is the count of the term in the document

Computing TF-IDF: Job 2

- Mapper

  - Input: ((term, docid), n)

  - Output: (docid, (term,n))

- Reducer

  - Input:

    ✓ key:docid

    ✓ Value: a list of (term, n), e.g. (term1, n1), (term2, n2), …

  - Sums counts of all words in the document (i.e. n1+n2+…)

  - Outputs ((term, docid), (n, N)) where n is the count of the term in the document and N is the total number of words in the document

Computing TF-IDF: Job 3

- Mapper

  - Input: ((term, docid), (n, N))

  - Output: (term, (docid, n, N, 1))

    - ✓ One document contains the term

- Reducer

  - Input:

    - ✓ Key: term
    - ✓ Value: a list of (docid, n, N, 1), e.g. (docid1, n1, N1, 1), (docid2, n2, N2, 1), …

  - Sums the number of documents that contain the term

  - Outputs ((term, docid), (n, N, m)) where n is the count of the term in the document, N is the total number of words in the document, m is the number of documents that contain the term

Computing TF-IDF: Job 4

- Mapper

  - Input: ((term, docid), (n, N, m))

  - Assume D (the total number of documents in the corpus) is known

  - Output: ((term, docid), TF × IDF) where TF = n/N, IDF = log(D/m)

- Reducer

  - Identity Reducer

How to Chain Jobs?

- Your driver code needs to configure two jobs
- The command line to run the mapreduce program takes three arguments:
  - the input directory
  - the output directory of the 1st job
  - the final output directory (i.e. the output directory of the 2nd job)

How to Chain Jobs?(Code)

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
//You may need to add other import statements here for key/value types, etc.

public class ChainJobsDriver {
      public static void main(String[] args) throws Exception {

            if (args.length != 3) {
                System.out.printf("Usage: ChainJobs <input dir>  <temp output dir>  <final output dir>\n");
                System.exit(-1);
             }

            Configuration conf = new Configuration();
```

How to Chain Jobs?(Cont.)

```
//set job1
Job job1 = Job.getInstance(conf, “job1”);
job1.setJarByClass(ChainJobsDriver.class);
//set the input path
FileInputFormat.setInputPaths(job1, new  Path(args[0]));
//set the output path
FileOutputFormat.setOutputPath(job1, new Path(args[1]));
// add the code to set Mapper, Reducer of job1
// add the code to set output key/value types of Mapper, Reducer of job1
…….
//wait for job1 to complete
int flag = job1.waitForCompletion(true)?0 : 1;
if (flag != 0) {
        System.out.println("Job1 failed, exiting");
        System.exit(flag) ;
}

//set job2
Job job2 = Job.getInstance(conf, “job2”);
job2.setJarByClass(ChainJobsDriver.class);
//set the input path of job2 to the directory which has output of job1
//set the output path of job2
FileInputFormat.setInputPaths(job2, new Path(args[1]));
FileOutputFormat.setOutputPath(job2, new Path(args[2]));
// add the code to set Mapper, Reducer of job2
// add the code to set output key/value types of Mapper, Reducer of job2
…….
//wait for job2 to complete
System.exit(job2.waitForCompletion(true) ? 0 : 1);
```

How to Sort in Descending Order?

- Provide a custom comparator (class that compares objects)
- Configure the job to use the custom comparator

Custom Comparator

```
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.IntWritable.Comparator;

public class DescendingIntComparator extends Comparator {

  public DescendingIntComparator() {
        super();
   }

   @Override
   public int compare(WritableComparable v1, WritableComparable v2){
        return -1 * super.compare(v1, v2);
    }

   @Override
  public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
         return -1 * super.compare(b1,s1,l1,b2,s2,l2);
   }
}
```

When extending Comparator, you need to provide your own constructor.

Custom Comparator

```
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.IntWritable.Comparator

public class DescendingIntComparator extends Comparator {

  public DescendingIntComparator() {
        super();
   }

   @Override
   public int compare(WritableComparable v1, WritableComparable v2){
        return -1 * super.compare(v1, v2);
   }

   @Override
  public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
         return -1 * super.compare(b1,s1,l1,b2,s2,l2);
   }
}
```

- public int compare(WritableComparable a, WritableComparable b)
  - ✓ Compare two WritableComparables
  - ✓ The default implementation uses the ascending order
- Overrides the default compare() method and reverses the comparison

Custom Comparator

```
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.IntWritable.Comparator

public class DescendingIntComparator extends Comparator {

  public DescendingIntComparator() {
        super();
   }

   @Override
   public int compare(WritableComparable v1, WritableComparable v2){
        return -1 * super.compare(v1, v2);
   }

   @Override
  public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        return -1 * super.compare(b1,s1,l1,b2,s2,l2);
   }
}
```

```
public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        return -1 * super.compare(b1,s1,l1,b2,s2,l2);
 }
```

- public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2)
  - ✓ compares byte strings rather than objects because it is much more efficient
  - ✓ b1 – the first byte array
  - ✓ s1 – the position index in b1. The object 1's starting index
  - ✓ l1 – the length of the object in b1
  - ✓ b2 – the second byte array
  - ✓ s2 – the position index in b2. The object 2's starting index
  - ✓ l2 – the length of the object in b2
  - ✓ returns an integer result of the comparison
- Your custom comparator needs to override this compare method to reverse the comparison

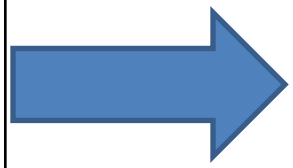
How to configure the job to use the custom comparator?
In Driver code:

```
job.setSortComparatorClass(DescendingIntComparator.class);
```

Secondary Sort

- Sometimes a job needs to receive the values for a particular key in a sorted order
  - this is known as a secondary sort

Addams Julie 1935-Oct-01
Jones Zeke 2001-Dec-12
Turing Alan 1912-Jun-23
Jones David 1947-Jan-08
Addams Jane 1960-Sep-06
Jones Asa 1901-Aug-08
Addams Gomez 1964-Sep-18
Jones David 1945-Dec-30



Addams	**Gomez**	1964-Sep-18
Addams	**Jane**	1860-Sep-06
Andrews	**Julie**	1935-Oct-01
Jones	**Asa**	1901-Aug-08
Jones	**David**	1947-Jan-08
Jones	**David**	1945-Dec-30
Jones	**Zeke**	2001-Dec-12
Turing	**Alan**	1912-Jun-23

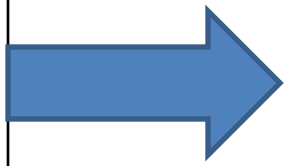
How to perform secondary sort?

Performing a Secondary Sort

- Can we use last name as the key?

- Records are sorted by last name. But the list of values for a particular key might not be sorted.

Addams Julie 1935-Oct-01
Jones Zeke 2001-Dec-12
Turing Alan 1912-Jun-23
Jones David 1947-Jan-08
Addams Jane 1960-Sep-06
Jones Asa 1901-Aug-08
Addams Gomez 1964-Sep-18
Jones David 1945-Dec-30



Addams	**Gomez**	1964-Sep-18
Addams	**Jane**	1860-Sep-06
Andrews	**Julie**	1935-Oct-01
Jones	**Zeke**	2001-Dec-12
Jones	**David**	1945-Dec-30
Jones	**Asa**	1901-Aug-08
Jones	**David**	1947-Jan-08
Turing	**Alan**	1912-Jun-23

For each key, the order of values might change between different runs of the MapReduce job.

Performing a Secondary Sort (Cont.)

- Can we use both last name and first name as the key?

  - the sorting of key happens automatically

  - Persons with the same last name might go to different Reducers (because they have different key)

  - Even if persons with the same last name go to the same Reducer, they won't be grouped into the same call of reduce() method (again because they have different key)

does not work for problems such as finding the latest birth year for each last name

Why Need Secondary Sort?

- Example: Find the latest birth year for each last name in a list
- Solution 1:

  - use the last name as the key

  - Reducer loops through all values, keeping track of the latest year and finally emits the latest year

How can we take advantage of the sorting of MapReduce instead of writing much code in Reducer to find the latest year?

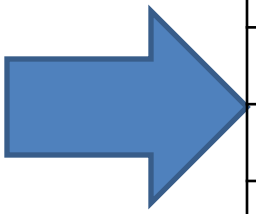
- Use Secondary Sort

Why Need Secondary Sort? (Cont.)

- Solution 2:

- Pass the values sorted by birth year in descending order to the Reducer, which can then just emit the first value.

Addams	Gomez	1964-Sep-18
Addams	Jane	1860-Sep-06
Andrews	Julie	1935-Oct-01
Jones	Zeke	2001-Dec-12
Jones	David	1947-Jan-08
Jones	David	1945-Dec-30
Jones	Asa	1901-Aug-08
Turing	Alan	1912-Jun-23



Reducer

Addams	1964
Andrews	1935
Jones	2001
Turing	1912

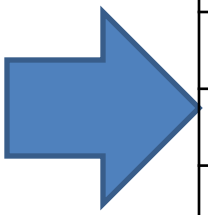
How do we make each partition sorted by last name, and then sorted by birth year in descending order?

Implementing Secondary Sort: Composite Keys

- To implement a secondary sort, the intermediate key should be a composite of the "primary" key and the "secondary" key

  - implement a mapper to emit composite keys

Addams Julie	1935-Oct-01
Jones Zeke	2001-Dec-12
Turing Alan	1912-Jun-23
Jones David	1947-Jan-08
Addams Jane	1960-Sep-06
Jones Asa	1901-Aug-08
Addams Gomez	1964-Sep-18
Jones David	1945-Dec-30



Mapper

Addams#1935	Addams Julie 1935-Oct-01
Jones#2001	Jones Zeke 2001-Dec-12
Turing#1912	Turing Alan 1912-Jun-23
Jones#1947	Jones David 1947-Jan-08
Addams#1960	Addams Jane 1960-Sep-06
Jones#1901	Jones Asa 1901-Aug-08
Addams#1946	Addams Gomez 1964-Sep-18
Jones#1945	Jones David 1945-Dec-30

Primary key: Last name
Secondary key: birth year

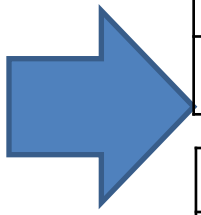
Addams#1935, Addams#1960 and Addams#1946 are three different keys. How to make the three records go to the same partition?

Implementing Secondary Sort: Partitioning Composite Keys

- Create a custom Partitioner

  - Use “Primary” key (e.g. last name in this example) to determine which Reducer to send the record to

Addams#1935	Addams Julie 1935-Oct-01
Jones#2001	Jones Zeke 2001-Dec-12
Turing#1912	Turing Alan 1912-Jun-23
Jones#1947	Jones David 1947-Jan-08
Addams#1960	Addams Jane 1960-Sep-06
Jones#1901	Jones Asa 1901-Aug-08
Addams#1946	Addams Gomez 1964-Sep-18
Jones#1945	Jones David 1945-Dec-30



Partitioner

Partition 0	
Addams#1935	Addams Julie 1935-Oct-01
Addams#1946	Addams Gomez 1964-Sep-18
Addams#1960	Addams Jane 1960-Sep-06
Turing#1912	Turing Alan 1912-Jun-23

Partition 1	
Jones#2001	Jones Zeke 2001-Dec-12
Jones#1947	Jones David 1947-Jan-08
Jones#1901	Jones Asa 1901-Aug-08
Jones#1945	Jones David 1945-Dec-30

Implementing Secondary Sort: Sorting Composite Keys

- Custom comparators can be used to sort composite keys
  - must extend Comparator
  - must override the method “int compare( )”
  - compare(A, B) returns (for ascending order)
    - ✓ 1 if A > B
    - ✓ 0 if A = B
    - ✓ -1 if A < B

What about for descending order?
- Reverse the comparison

- Two comparators are required
  - Sort Comparator
  - Group Comparator

Implementing Secondary Sort: Sort Comparator and Grouping Comparator

- Sort Comparator

  - sorts the input (i.e. partition) to the Reducer

  - compares “primary” key first; if equal, compares “secondary” key

e.g. for Sort Comparator
Addams#1860 > Addams#1964
Addams#1860 < Jones#1965

- Grouping Comparator

  - Determines which key and value pairs are passed in a single call to the Reduce() method

  - compares “primary” key only; if equal, passed to a single call to the Reduce() method

e.g. for Grouping Comparator
Addams#1860 = Addams#1964
Addams#1860 < Jones#1965

Implementing Secondary Sort: Setting Comparators

- Configure the job to use both comparators

In the Driver code:

```
job.setSortComparatorClass(NameYearComparator.class);
job.setGroupingComparatorClass(NameComparator.class);
```

Secondary Sort: Summary

Addams#1935	Addams Julie 1935-Oct-01
Jones#2001	Jones Zeke 2001-Dec-12
Turing#1912	Turing Alan 1912-Jun-23
Jones#1947	Jones David 1947-Jan-08
Addams#1960	Addams Jane 1960-Sep-06
Jones#1901	Jones Asa 1901-Aug-08
Addams#1946	Addams Gomez 1964-Sep-18
Jones#1945	Jones David 1945-Dec-30

1. Mapper emits composite keys

Partition 0	
Addams#1935	Addams Julie 1935-Oct-01
Addams#1946	Addams Gomez 1964-Sep-18
Addams#1960	Addams Jane 1960-Sep-06
Turing#1912	Turing Alan 1912-Jun-23

Partition 1	
Jones#2001	Jones Zeke 2001-Dec-12
Jones#1947	Jones David 1947-Jan-08
Jones#1901	Jones Asa 1901-Aug-08
Jones#1945	Jones David 1945-Dec-30

2. Custom Partitioner partitions by "primary" key

Secondary Sort: Summary (Cont.)

Partition 0	
Addams#1960	Addams Jane 1960-Sep-06
Addams#1946	Addams Gomez 1964-Sep-18
Addams#1935	Addams Julie 1935-Oct-01
Turing#1912	Turing Alan 1912-Jun-23

Partition 1	
Jones#2001	Jones Zeke 2001-Dec-12
Jones#1947	Jones David 1947-Jan-08
Jones#1945	Jones David 1945-Dec-30
Jones#1901	Jones Asa 1901-Aug-08

3. Sort Comparator sorts composite key

Partition 0	
Addams#1960	Addams Jane 1960-Sep-06
Addams#1946	Addams Gomez 1964-Sep-18
Addams#1935	Addams Julie 1935-Oct-01
Turing#1912	Turing Alan 1912-Jun-23

Partition 1	
Jones#2001	Jones Zeke 2001-Dec-12
Jones#1947	Jones David 1947-Jan-08
Jones#1945	Jones David 1945-Dec-30
Jones#1901	Jones Asa 1901-Aug-08

4. Grouping Comparator groups by "primary" key for reduce() calls