Apache Hadoop – A course for undergraduates

Lecture 7

Common MapReduce Algorithms

Chapter 7.1

Common MapReduce Algorithms

- **How to sort and search large data sets**
- **How to perform a secondary sort**
- **How to index data**
- **How to compute term frequency – inverse document frequency (TF-IDF)**
- **How to calculate word co-occurrence**

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**
  - Often using Oozie (see later)
- **You are likely to find that many of your MapReduce jobs use very similar code**
- **In this chapter we present some very common MapReduce algorithms**
  - These algorithms are frequently the basis for more complex MapReduce jobs

Chapter Topics

Common MapReduce Algorithms

- **Sorting and Searching Large Data Sets**
- Indexing Data
- Computing Term Frequency – Inverse Document Frequency (TF-IDF)
- Calculating Word Co-Occurrence
- Performing a Secondary Sort

Sorting (1)

- **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, '')`

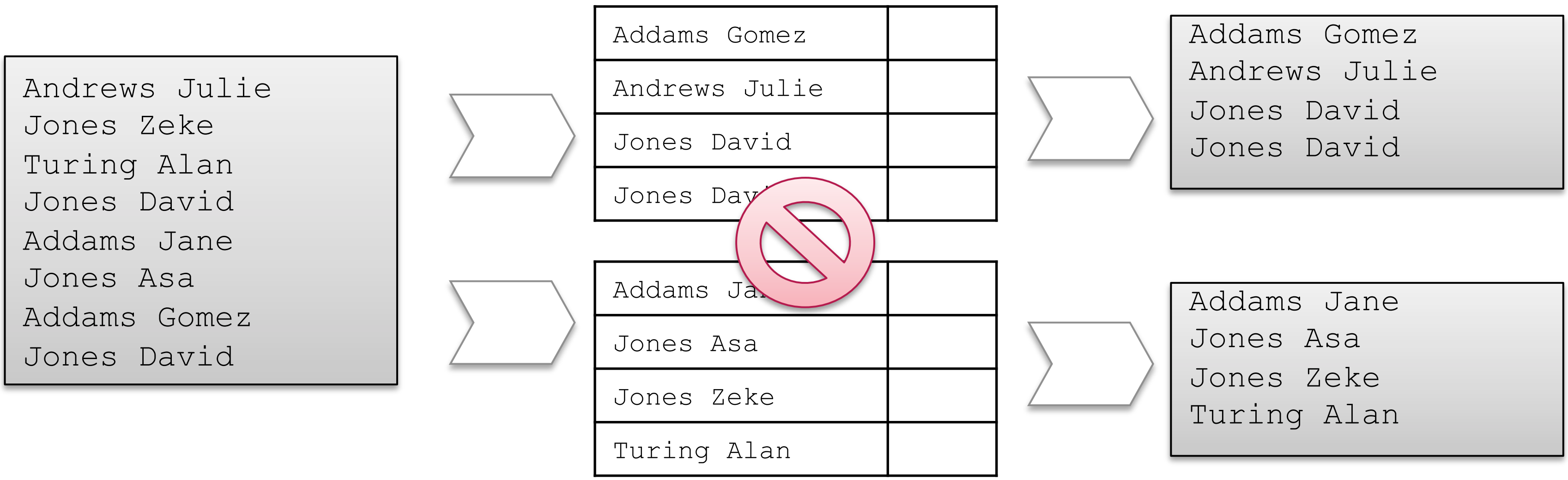
```
Andrews Julie
Jones Zeke
Turing Alan
Jones David
Addams Jane
Jones Asa
Addams Gomez
Jones David
```

Addams Gomez	
Addams Jane	
Andrews Julie	
Jones Asa	
Jones David	
Jones David	
Jones Zeke	
Turing Alan	

```
Addams Gomez
Addams Jane
Andrews Julie
Jones Asa
Jones David
Jones David
Jones Zeke
Turing Alan
```

Sorting (2)

- **Trivial with a single Reducer**
- **Harder for multiple Reducers**



- **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

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**
  - We saw how to pass parameters to a Mapper in a previous chapter
- **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
    - Just outputs each intermediate key

Chapter Topics

Common MapReduce Algorithms

- Sorting and Searching Large Data Sets
- **Indexing Data**
- Computing Term Frequency – Inverse Document Frequency (TF-IDF)
- Calculating Word Co-Occurrence
- Performing a Secondary Sort

Indexing

- **Assume the input is a set of files containing lines of text**
- **Key is the byte offset of the line, value is the line itself**
- **We can retrieve the name of the file using the Context object**
  - More details on how to do this in the Exercise

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)
    - Emit `(word, filename_list)`

Inverted Index: Dataflow

File1

```
I never saw a
purple cow;
```

File2

```
I never hope to
see one.
```

i	File1
never	File1
saw	File1
a	File1
purple	File1
cow	File1

i	File2
never	File2
hope	File2
to	File2
see	File2
one	File2

a	File1
cow	File1
hope	File2
i	File1, File2
never	File1, File2
one	File2
purple	File1
saw	File1
see	File2
to	File2

Aside: Word Count

- **Recall the WordCount example we used earlier in the course**
  - For each word, Mapper emitted `(word, 1)`
  - Very similar to the inverted index
- **This is a common theme: reuse of existing Mappers, with minor modifications**

Chapter Topics

Common MapReduce Algorithms

- Sorting and Searching Large Data Sets
- Indexing Data
- **Computing Term Frequency – Inverse Document Frequency (TF-IDF)**
- Calculating Word Co-occurrence
- Performing a Secondary Sort

Term Frequency – Inverse Document Frequency

- **Term Frequency – Inverse Document Frequency (TF-IDF)**
  - Answers the question "How important is this term in a document?"
- **Known as a *term weighting function***
  - Assigns a score (weight) to each term (word) in a document
- **Very commonly used in text processing and search**
- **Has many applications in data mining**

TF-IDF: Motivation

- **Merely counting the number of occurrences of a word in a document is not a good enough measure of its relevance**
  - If the word appears in many other documents, it is probably less relevant
  - 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: Data Mining Example

- **Consider a music recommendation system**
  - Given many users' music libraries, provide "you may also like" suggestions
- **If user A and user B have similar libraries, user A may like an artist in user B's library**
  - But some artists will appear in almost everyone's library, and should therefore be ignored when making recommendations
    - Almost everyone has The Beatles in their record collection!

TF-IDF Formally Defined

- **Term Frequency (TF)**
  - Number of times a term appears in a document (i.e., the count)
- **Inverse Document Frequency (IDF)**

$$idf = \log\left(\frac{N}{n}\right)$$

  - N: total number of documents
  - *n*: number of documents that contain a term
- **TF-IDF**
  - TF × IDF

Computing TF-IDF

- **What we need:**
  - Number of times *t* appears in a document
    - Different value for each document
  - Number of documents that contains *t*
    - One value for each term
  - Total number of documents
    - One value

Computing TF-IDF With MapReduce

- **Overview of algorithm: 3 MapReduce jobs**
  - Job 1: compute term frequencies
  - Job 2: compute number of documents each word occurs in
  - Job 3: compute TF-IDF
- **Notation in following slides:**
  - *docid* = a unique ID for each document
  - *contents* = the complete text of each document
  - *N* = total number of documents
  - *term* = a term (word) found in the document
  - *tf* = term frequency
  - *n* = number of documents a term appears in
- **Note that real-world systems typically perform 'stemming' on terms**
  - Removal of plurals, tense, possessives etc

Computing TF-IDF: Job 1 – Compute *tf*

- **Mapper**
  - Input: (docid, contents)
  - For each term in the document, generate a (term, docid) pair
    - i.e., we have seen this term in this document once
  - Output: ((term, docid), 1)
- **Reducer**
  - Sums counts for word in document
  - Outputs ((term, docid), *tf*)
    - i.e., the term frequency of term in docid is *tf*
- **We can add a Combiner, which will use the same code as the Reducer**

Computing TF-IDF: Job 2 – Compute *n*

- **Mapper**
  - Input: ((term, docid), *tf*)
  - Output: (term, (docid, *tf*, 1))
- **Reducer**
  - Sums 1s to compute *n* (number of documents containing term)
  - Note: need to buffer (docid, *tf*) pairs while we are doing this (more later)
  - Outputs ((term, docid), (*tf*, *n*))

Computing TF-IDF: Job 3 – Compute TF-IDF

- **Mapper**
  - Input: ((term, docid), (*tf*, *n*))
  - Assume N is known (easy to find)
  - Output ((term, docid), TF × IDF)
- **Reducer**
  - The identity function

Computing TF-IDF: Working At Scale

- **Job 2: We need to buffer (docid, *tf*) pairs counts while summing 1's (to compute *n*)**
  - Possible problem: pairs may not fit in memory!
    - In how many documents does the word "the" occur?
- **Possible solutions**
  - Ignore very-high-frequency words
  - Write out intermediate data to a file
  - Use another MapReduce pass

TF-IDF: Final Thoughts

- **Several small jobs add up to full algorithm**
  - Thinking in MapReduce often means decomposing a complex algorithm into a sequence of smaller jobs
- **Beware of memory usage for large amounts of data!**
  - Any time when you need to buffer data, there's a potential scalability bottleneck

Chapter Topics

Common MapReduce Algorithms

- Sorting and Searching Large Data Sets
- Indexing Data
- Computing Term Frequency – Inverse Document Frequency (TF-IDF)
- **Calculating Word Co-Occurrence**
- Performing a Secondary Sort

Word Co-Occurrence: Motivation

- **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'

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)
}
```

Chapter Topics

Common MapReduce Algorithms

- Sorting and Searching Large Data Sets
- Indexing Data
- Computing Term Frequency – Inverse Document Frequency (TF-IDF)
- Calculating Word Co-occurrence
- **Performing a Secondary Sort**

Secondary Sort: Motivation (1)

- **Recall that keys are passed to the Reducer in sorted order**
- **The list of values for a particular key is not sorted**
  - Order may well change between different runs of the MapReduce job

```
Andrews 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 1964-09-18
Addams	Jane 1860-Sep-06
Andrews	Julie 1935-Oct-01
Jones	Zeke 2001-Dec-12
Jones	David 1947-Jan-08
Jones	Asa 1901-Aug-08
Jones	David 1957-Jan-08
Turing	Alan 1912-Jun-23

Secondary Sort: Motivation (2)

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

- **Example: Sort by Last Name, then First Name**

Addams	**Jane** 1860-Sep-06
Addams	**Gomez** 1964-Sep-18
Andrews	**Julie** 1935-Oct-01
Jones	**Zeke** 2001-Dec-12
Jones	**David** 1957-Jan-08
Jones	**Asa** 1901-Aug-08
Jones	**David** 1945-Dec-30
Turing	**Alan** 1912-Jun-23

Addams	**Gomez** 1964-Sep-18
Addams	**Jane** 1860-Sep-06
Andrews	**Julie** 1935-Oct-01
Jones	**Asa** 1901-Aug-08
Jones	**David** 1957-Jan-08
Jones	**David** 1945-Dec-30
Jones	**Zeke** 2001-Dec-12
Turing	**Alan** 1912-Jun-23

Secondary Sort: Motivation (3)

- **Example: Find the latest birth year for each surname in a list**
- **Naïve solution**
  - Reducer loops through all values, keeping track of the latest year
  - Finally, emit the latest year
- **Better solution**
  - Pass the values sorted by year in descending order to the Reducer, which can then just emit the first value

Addams	Gomez **1964**-09-18
Addams	Jane **1860**-Sep-06
Andrews	Julie **1935**-Oct-01
Jones	Zeke **2001**-Dec-12
Jones	David **1947**-Jan-08
Jones	Asa **1901**-Aug-08
Jones	David **1945**-Dec-30
Turing	Alan **1912**-Jun-23

Addams	Gomez **1964**-09-18
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**

Implementing Secondary Sort: Composite Keys

- **To implement a secondary sort, the intermediate key should be a composite of the 'actual' (natural) key and the value**
- **Implement a mapper to construct composite keys**

```
let map(k, v) =
    emit(new Pair(v.getPrimaryKey(), v.getSecondaryKey)), v)
```

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

Mapper

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

Implementing Secondary Sort: Partitioning Composite Keys

- **Create a custom partitioner**
  - Use natural key to determine which Reducer to send the key to

```
let getPartition(Pair k, Text v, int numReducers) =
    return(k.getPrimaryKey().hashCode() % numReducers)
```

Jones#1947	Jones David 1947-Jan-08
Addams#1860	Addams Jane 1860-Sep-06
Jones#1901	Jones Asa 1901-Aug-08
Addams#1964	Addams Gomez 1964-Sep-18
Jones#1945	Jones David 1945-Dec-30

Partitioner

Partition 0	
Jones#1947	Jones David 1947-Jan-08
Jones#1901	Jones Asa 1901-Aug-08
Jones#1945	Jones David 1945-Dec-30

Partition 1	
Addams#1860	Addams Jane 1860-Sep-06
Addams#1964	Addams Gomez 1964-Sep-18

Implementing Secondary Sort: Sorting Composite Keys

- **Comparator classes are classes that compare objects**
  - `compare(A,B)` returns:
    - 1 if A>B
    - 0 if A=B
    - -1 if A<B
- **Custom comparators can be used to sort composite keys**
  - extend **`WritableComparator`**
  - override **`int compare()`**
- **Two comparators are required:**
  - Sort Comparator
  - Group Comparator

Implementing Secondary Sort: Sort Comparator

- **Sort Comparator**
  - Sorts the input to the Reducer
  - Uses the full composite key: compares natural key first; if equal, compares secondary key

```
let compare(Pair k1, Pair k2) =
    compare k1.getPrimaryKey(), k2.getPrimaryKey()
    if equal
        compare k1.getSecondaryKey(), k2.getSecondaryKey()
```

```
Addams#1860 > Addams#1964
Addams#1860 < Jones#1965
```

Implementing Secondary Sort: Grouping Comparator

- **Grouping Comparator**
  - Uses 'natural' key only
  - Determines which keys and values are passed in a single call to the Reducer

```
let compare(Pair k1, Pair k2) =
    compare k1.getPrimaryKey(), k2.getPrimaryKey()
```

```
Addams#1860 = Addams#1964
Addams#1860 < Jones#1945
```

Implementing Secondary Sort: Setting Comparators

- **Configure the job to use both comparators**

```
public class MyDriver extends Configured implements Tool {

    public int run(String[] args) throws Exception {
        …
        job.setSortComparatorClass(NameYearComparator.class);
        job.setGroupingComparatorClass(NameComparator.class);
        …
    }
}
```

Secondary Sort: Summary

1. Mapper emits composite keys

Turing#1912	Turing Alan 1912-Jun-23
Jones#1947	Jones David 1947-Jan-08
Addams#1960	Addams Jane 1860-Sep-06
Jones#1901	Jones Asa 1901-Aug-08
Addams#1964	Addams Gomez 1964-Sep-18
Jones#1945	Jones David 1945-Dec-30

2. Custom Partitioner partitions by natural key

Partition 0	
Jones#1947	Jones David 1947-Jan-08
Turing#1912	Turing Alan 1912-Jun-23
Jones#1901	Jones Asa 1901-Aug-08
Jones#1945	Jones David 1945-Dec-30

Partition 1	
Addams#1860	Addams Jane 1860-Sep-06
Addams#1964	Addams Gomez 1964-Sep-18

3. Sort Comparator sorts composite key

Partition 0	
Jones#1947	Jones David 1947-Jan-08
Jones#1945	Jones David 1945-Dec-30
Jones#1901	Jones Asa 1901-Aug-08
Turing#1912	Turing Alan 1912-Jun-23

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

Jones#1947	Jones David 1947-Jan-08
Jones#1945	Jones David 1945-Dec-30
Jones#1901	Jones Asa 1901-Aug-08

Turing#1912	Turing Alan 1912-Jun-23

Key Points (1)

Common MapReduce Algorithms

- **Sorting**
  - simple for single reduce jobs, more complex for multiple reduces
- **Searching**
  - Pass a match string parameter to a search mapper
  - Emit matching records, ignore non-matching records
- **Indexing**
  - Inverse Mapper: emit (term, file)
  - Identity Reducer
- **Term frequency – inverse document frequency (TF-IDF)**
  - Often used for recommendation engines and text analysis
  - Three sequential MapReduce jobs

Key Points (2)

- **Word co-occurrence**
  - Mapper: emits pairs of “close” words as keys, their frequencies as values
  - Reducer: sum frequencies for each pair
- **Secondary Sort**
  - Define a composite key type with natural key and secondary key
  - Partition by natural key
  - Define comparators for sorting (by both keys) and grouping (by natural key)

Bibliography

The following offer more information on topics discussed in this chapter

- **For more information on TF-IDF, see**
  - http://marcellodesales.wordpress.com/2009/12/31/tf-idf-in-hadoop-part-1-word-frequency-in-doc/
- **The secondary sort is described in TDG 3e on pages 277-283.**

Joining Data Sets in MapReduce Jobs

Chapter 7.2

Joining Data Sets in MapReduce Jobs

- **Writing a Map-side join**
- **Writing a Reduce-side join**

Introduction

- **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 and Reduce-side joins**
- **Map-side joins are easier to write, but have potential scaling issues**
- **We will investigate both types of joins in this chapter**

But First…

- **But first…**
- **Avoid writing joins in Java MapReduce if you can!**
- **Tools such as Impala, Hive, and Pig 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 Impala, Hive, or Pig**

Chapter Topics

Joining Data Sets in MapReduce Jobs

- **Writing a Map-side Join**
- Writing a Reduce-side Join

Map-Side Joins: The Algorithm

- **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

Map-Side Joins: Problems, Possible Solutions

- **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

Chapter Topics

Joining Data Sets in MapReduce Jobs

- Writing a Map-side Join
- **Writing a Reduce-side Join**

Reduce-Side Joins: The Basic Concept

- **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

Join

Employees

empid	empname	locid
001	Elizabeth Windsor	4
002	Peter Parker	5
003	Levi Strauss	2
004	Francis Bacon	4
…	…	…

Locations

locid	location
1	Chicago
2	San Francisco
3	Amsterdam
4	London
5	New York

```
003   Levi Strauss       San Francisco
001   Elizabeth Windsor London
004   Francis Bacon      London
002   Peter Parker       New York
…
```

Example Record Data Structure

- **A data structure to hold a record could look like this:**

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

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

- **Example records**

```
   type: emp
  empId: 002
empName: Levi Strauss
  locId: 2
locName: <null>
```

```
   type: loc
  empId: <null>
empName: <null>
  locId: 4
locName: London
```

Reduce-Side Join: Mapper

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

```
001  Elizabeth Windsor 4
002  Levi Strauss      2
004  Francis Bacon     4
```

```
1    Chicago
2    San Francisco
3    Amsterdam
4    London
```

Map

4	emp 001 Elizabeth Windsor 4 <null>
2	emp 003 Levi Strauss 2 <null>
4	emp 004 Francis Bacon 4 <null>
1	loc <null> <null> 1 Chicago
2	loc <null> <null> 2 San Francisco
3	loc <null> <null> 3 Amsterdam
4	loc <null> <null> 4 London

Reduce-Side Join: Shuffle and Sort

4	emp 001 Elizabeth Windsor 4 <null>
2	emp 003 Levi Strauss 2 <null>
4	emp 004 Francis Bacon 4 <null>
1	loc <null> <null> 1 Chicago
2	loc <null> <null> 2 San Francisco
3	loc <null> <null> 3 Amsterdam
4	loc <null> <null> 4 London

Shuffle and Sort

1	loc <null> <null> 1 Chicago
2	emp 003 Levi Strauss 2 <null>
2	loc <null> <null> 2 San Francisco
3	loc <null> <null> 3 Amsterdam
4	emp 001 Elizabeth Windsor 4 <null>
4	loc <null> <null> 4 London
4	emp 004 Francis Bacon 4 <null>

Reduce-Side Join: Reducer

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

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

Reduce-Side Join: Reducer Grouping

1	loc <null> <null> 1 Chicago
2	emp 003 Levi Strauss 2 <null>
2	loc <null> <null> 2 San Francisco
3	loc <null> <null> 3 Amsterdam
4	emp 001 Elizabeth Windsor 4 <null>
4	loc <null> <null> 4 London
4	emp 004 Francis Bacon 4 <null>

Reduce

emp	003	Levi Strauss	2	San Francisco
emp	001	Elizabeth Windsor	4	London
emp	004	Francis Bacon	4	London

Scalability Problems With Our Reducer

- **All employees for a given location are buffered in the Reducer**
  - Could result in out-of-memory errors for large data sets

```
…
   for (Record v in values) {
      if (v.type == RecType.loc) {
         thisLocation = v;
      } else {
         employees.add(v);
      }
   }
…
```

- **Solution: Ensure the location record is the first one to arrive at the Reducer**
  - Using a Secondary Sort

A Better Intermediate Key (1)

```
class LocKey {
  int locId;
  boolean isLocation;

  public int compareTo(LocKey k) {
    if (locId != k.locId) {
      return Integer.compare(locId, k.locId);
    } else {
      return Boolean.compare(k.isLocation, isLocation);
    }
  }

  public int hashCode() {
    return locId;
  }
}
```

A Better Intermediate Key (2)

```
class LocKey {
  int locId;
  boolean isLocation;

  public i
    if (lo
      retu
    } else
      retu
    }
  }

  public int hashCode() {
    return locId;
  }
}
```

Example Keys:

```
    locId: 4
isLocation: true
```

```
    locId: 4
isLocation: false
```

A Better Intermediate Key (3)

```
class LocKey {
  int locId;
  boolean isLocation;

  public int compareTo(LocKey k) {
    if (locId != k.locId) {
      return Integer.compare(locId, k.locId);
    } else {
      return Boolean.compare(k.isLocation, isLocation);
    }
  }

  public i
    return
  }
}
```

The `compareTo` method ensures that location keys will sort earlier than employee keys for the same location.

```
    locId: 4                      locId: 4
isLocation: true      >      isLocation: false
```

A Better Intermediate Key (4)

```
class LocKey {
  int locI
  boolean
  
  public i
    if (lo
      retu
    } else
      retu
    }
  }

  public int hashCode() {
    return locId;
  }
}
```

The `hashCode` method only looks at the location ID portion of the record. This ensures that all records with the same key will go to the same Reducer. This is an alternative to providing a custom Partitioner.

```
locId: 4                     locId: 4
isLocation: true     ==     isLocation: false
```

A Better Mapper

```
void map(k, v) {
  Record r = parse(v);
  LocKey newkey = new LocKey;
  newkey.locId = r.locId;

  if (r.type == RecordType.emp) {
    newkey.isLocation = false;
  } else {
    newkey.isLocation = true;
  }
  emit (newkey, r);
}
```

```
001  Elizabeth Windsor   4
002  Levi Strauss        2
004  Francis Bacon       4
```

```
1    Chicago
2    San Francisco
3    Amsterdam
```

Map

4#false	001 Elizabeth Windsor
2#false	003 Levi Strauss
4#false	004 Francis Bacon
1#true	Chicago
2#true	San Francisco
3#true	Amsterdam
4#true	London

Create a Sort Comparator…

- **Create a sort comparator to ensure that the location record is the first one in the list of records passed in each Reducer call**

```
class LocKeySortComparator

  boolean compare (k1,k2) {
    return (k1.compareTo(k2));
  }
}
```

…And a Grouping Comparator…

- **Create a Grouping Comparator to ensure that all records for a given location are passed in a single call to the `reduce()` method**

```
class LocKeyGroupingComparator

  boolean compare (k1,k2) {
    return (Integer.compare(k1.locId, k2.locId));
  }
}
```

…And Configure Hadoop To Use It In The Driver

```
job.setSortComparatorClass(LocKeySortComparator.class);
job.setGroupingComparatorClass(LocKeyGroupingComparator.class);
```

4#false	001 Elizabeth Windsor
2#false	003 Levi Strauss
4#false	004 Francis Bacon
1#true	Chicago
2#true	San Francisco
3#true	Amsterdam
4#true	London

Shuffle and Sort

1#true	Chicago
2#true	San Francisco
2#false	003 Levi Strauss
3#true	Amsterdam
4#true	London
4#false	001 Elizabeth Windsor
4#false	004 Francis Bacon

A Better Reducer

```
Record thisLoc;

void reduce(k, values) {
  for (Record v in values) {
    if (v.type == RecordType.loc) {
      thisLoc = v;
    } else {
      v.locationName = thisLoc.locationName;
      emit(v);
    }
  }
}
```

A Better Reducer: Output with Correct Sorting and Grouping

1#true	Chicago
2#true	San Francisco
2#false	003 Levi Strauss
3#true	Amsterdam
4#true	London
4#false	001 Elizabeth Windsor
4#false	004 Francis Bacon

reduce()

reduce()

reduce()

reduce()

```
002   Levi Strauss     San Francisco
001   Elizabeth Windsor    London
004   Francis Bacon    London
```

Key Points

- **Joins are usually best done using Impala, Hive, or Pig**
- **Map-side joins are simple but don't scale well**
- **Use reduce-side joins when both datasets are large**
  - Mapper:
    - Merges both data sets into a common record type
    - Use a composite key (custom `WritableComparable`) with join key/record type
  - Shuffle and sort:
    - Secondary sort so that 'primary' records are processed first
    - Custom Partitioner to ensure records are sent to the correct Reducer (or hack the hashCode of the composite key)
  - Reducer:
    - Group by join key (custom grouping comparator)
    - Write out 'secondary' records joined with 'primary' record data