MapReduce Algorithms

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

Source: Parsian, Mahmoud (2015): Data algorithms. Recipes for scaling up with Hadoop and Spark. O'Reilly Media.

Motivation

- In Hadoop you have one Map and one Reduce function
- It is typical to combine multiple small MapReduce jobs together in a single workflow
- MapReduce jobs tend to be relatively short in terms of lines of code
- We present some examples of common MapReduce algorithms

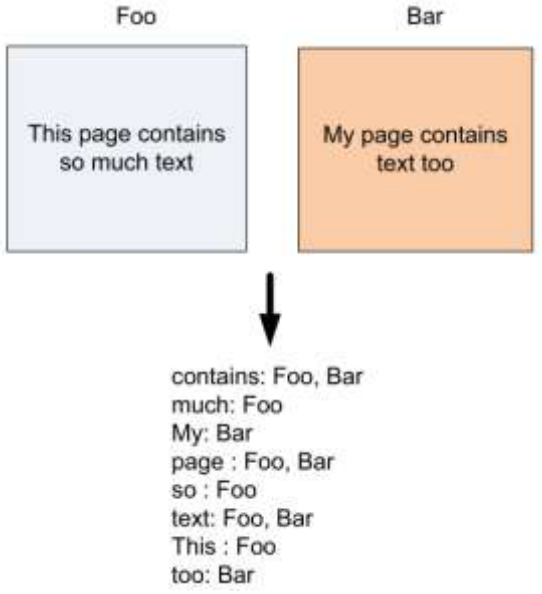
Outline

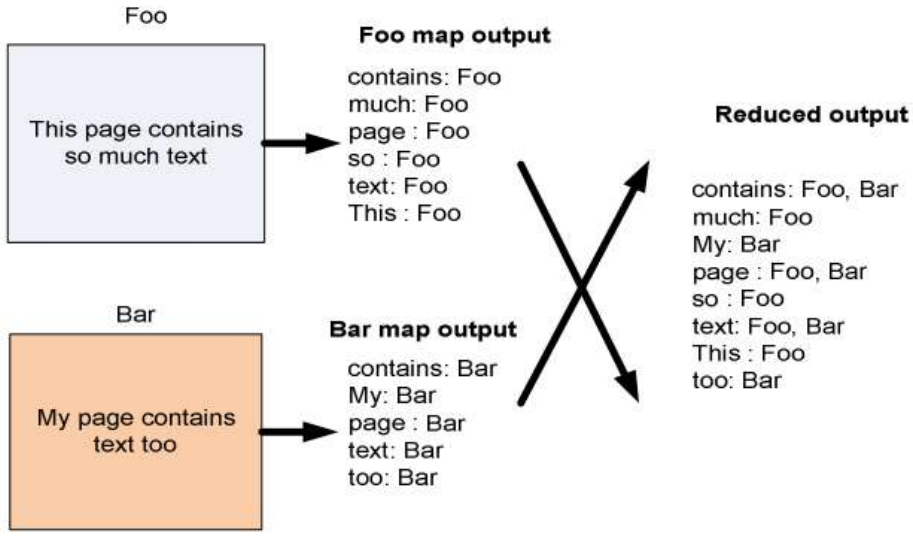
- Indexing Data
- Secondary Sort
- Top N List
- Left Outer Join

Indexing

Inverted Index for a Search Engine

- 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





Inverted Index: MapReduce

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: Data

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

Outline

- Indexing Data
- Secondary Sort
- Top N List
- Left Outer Join

Secondary Sort

A secondary sort problem relates to sorting values associated with a key in the reduce phase.

MapReduce framework automatically sorts the keys generated by mappers.

But values passed to each reducer are not sorted at all; they can be in any order.

$\texttt{map}(\texttt{key}_1, \texttt{value}_1) \rightarrow \texttt{list}(\texttt{key}_2, \texttt{value}_2)$

$\texttt{reduce}(\texttt{key}_2, \texttt{list}(\texttt{value}_2)) \rightarrow \texttt{list}(\texttt{key}_3, \texttt{value}_3)$

Consider key-value pair ($\texttt{key}_2$, $\texttt{list}(\texttt{value}_2)$) as an input for a reducer

$\texttt{list}(\texttt{value}_2) = (V_1, V_2, \ldots, V_n)$

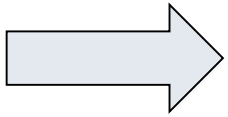
Secondary Sort sorts the values received by a reducer in some order.

$SORT(V_1, V_2, ..., V_n) = (S_1, S_2, ..., S_n)$

$list(value_2) = (S_1, S_2, ..., S_n)$

where: $S_1 < S_2 < ... < S_n$, or $S_1 > S_2 > ... > S_n$

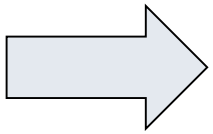
```
2012, 01, 01, 5
2012, 01, 02, 45
2012, 01, 03, 35
2012, 01, 04, 10
...
2001, 11, 01, 46
2001, 11, 02, 47
2001, 11, 03, 48
2001, 11, 04, 40
...
2005, 08, 20, 50
2005, 08, 21, 52
2005, 08, 22, 38
2005, 08, 23, 70
```



```
2012-01: 5, 10, 35, 45, ...
2001-11: 40, 46, 47, 48, ...
2005-08: 38, 50, 52, 70, ...
```

Example

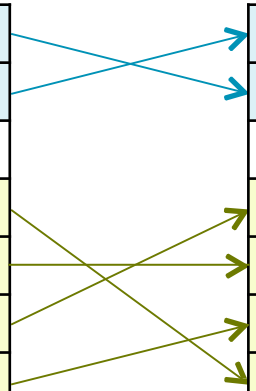
```
Andrews Julie 1935-Oct-01
Jones Zeke 2001-Dec-12
Turing Alan 1912-Jun-23
Jones David 1947-Jan-08
Addams Jane 1960-Sep-06
Jones Asa 1901-Aug-08
Addams Gomez 1964-Sep-18
Jones David 1945-Dec-30
```



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

With Secondary Sort: 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

Naïve Solution to the Secondary Sort Problem

The reducer reads and buffers all of the values for a given key (in an array data structure, for example), then doing an in-reducer sort on the values.

Problems:

- This approach will not scale: since the reducer will be receiving all values for a given key, this approach might cause the reducer to run out of memory.
- On the other hand, this approach can work well if the number of values is small enough that it will not cause an out-of-memory error

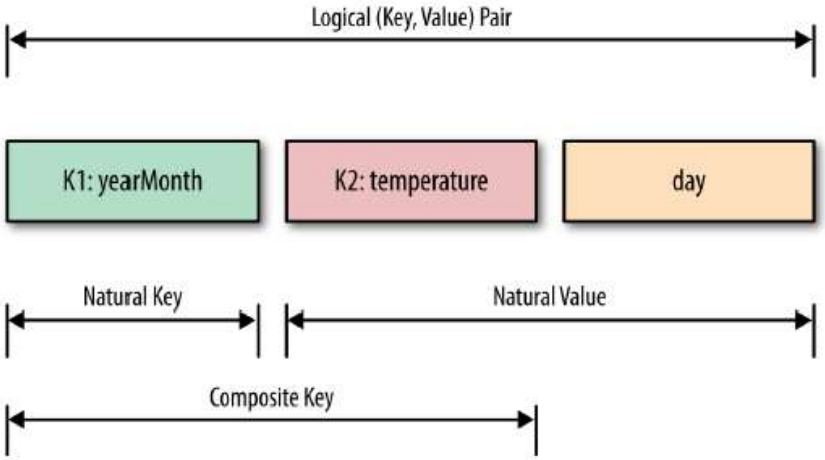
Solution to the Secondary Sort Problem

- Use the MapReduce framework for sorting the reducer values
- This does not require in-reducer sorting of values passed to the reduce.
- Idea: “create a composite key by adding a part of, or the entire value to, the natural key to achieve your sorting objectives.”

- This option is scalable and will not generate out-of-memory errors.
- We basically offload the sorting to the MapReduce framework
- Sorting is a paramount feature of the MapReduce Hadoop framework.

Use the *Value-to-Key Conversion* design pattern

1. Form a composite intermediate key, (K, V_1), where V_1 is the secondary key. Here, K is called a *natural key*. To inject a value (i.e., V_1) into a reducer key, simply create a composite key.
2. Let the MapReduce execution framework do the sorting
3. Preserve state across multiple key-value pairs to handle processing; you can achieve this by having proper mapper output partitioners



Source: Data Algorithms

Example: Address

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
```

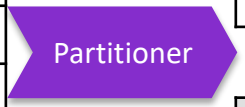
Jones#2001	Jones Zeke 2001-Dec-12
Turing#1912	Turing Alan 1912-Jun-23
Jones#1947	Jones David 1947-Jan-08
Addams#1860	Addams Jane 1860-Sep-06
Jones#1901	Jones Asa 1901-Aug-08
Addams#1964	Addams Gomez 1964-Sep-18
Jones#1945	Jones David 1945-Dec-30

Example: Address - 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



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

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

Example: Address - 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
```

Example: Address - 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
```

Example: Adress-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 Partioner partions by natural key

ParJJon 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

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

3. Sort Comparator sorts composite key

ParJJon 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

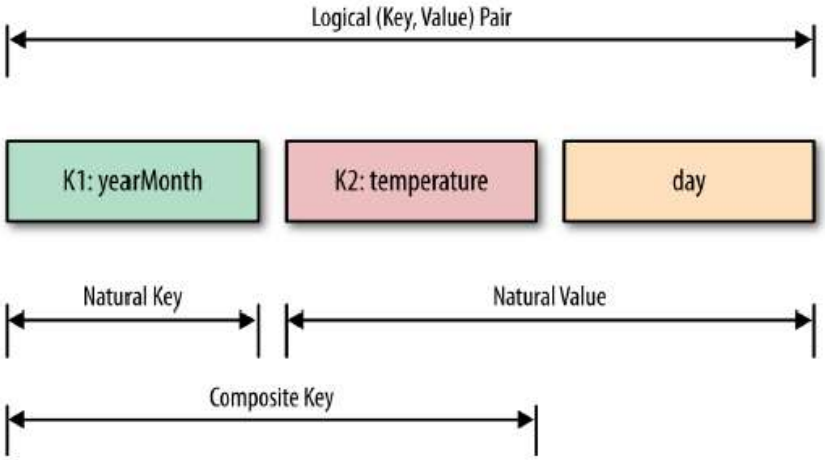
Jones#1947	Jones David 1947-Jan-08
Jones#1945	Jones David 1945-Dec-30
Jones#1901	Jones Asa 1901-Aug-08

Turing#1912	Turing Alan 1912-Jun-23

Source: Cloudera

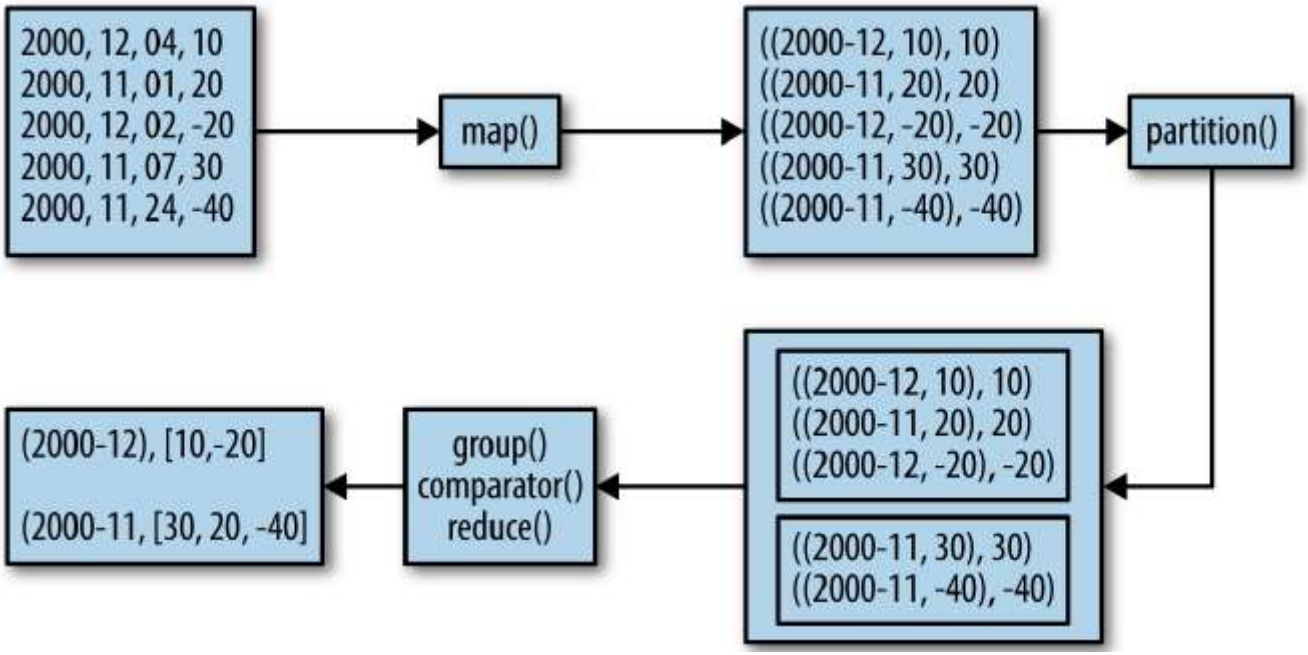
Use the *Value-to-Key Conversion* design pattern

1. Form a composite intermediate key, (K, V_1), where V_1 is the secondary key. Here, K is called a *natural key*. To inject a value (i.e., V_1) into a reducer key, simply create a composite key.
2. Let the MapReduce execution framework do the sorting
3. Preserve state across multiple key-value pairs to handle processing; you can achieve this by having proper mapper output partitioners



Source: Data Algorithms

Temperature: Data Flow



Temperature: Mapper

```
public class SecondarySortMapper
    extends Mapper<LongWritable, Text, DateTemperaturePair, Text> {
    private final Text theTemperature = new Text();
    private final DateTemperaturePair pair = new DateTemperaturePair();
    @Override
    /**
     * @param key is generated by Hadoop (ignored here)
     * @param value has this format: "YYYY,MM,DD,temperature"
     */
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String line = value.toString();
        String[] tokens = line.split(",");
        // YYYY = tokens[0]
        // MM = tokens[1]
        // DD = tokens[2]
        // temperature = tokens[3]
        String yearMonth = tokens[0] + tokens[1];
        String day = tokens[2];
        int temperature = Integer.parseInt(tokens[3]);

        pair.setYearMonth(yearMonth);
        pair.setDay(day);
        pair.setTemperature(temperature);
        theTemperature.set(tokens[3]);

        context.write(pair, theTemperature);
    }}
```

Temperature: Reducer

```
public class SecondarySortReducer
    extends Reducer<DateTemperaturePair, Text, Text, Text> {

    @Override
    protected void reduce(DateTemperaturePair key, Iterable<Text> values,
Context context)
        throws IOException, InterruptedException {
        StringBuilder builder = new StringBuilder();
        for (Text value : values) {
                builder.append(value.toString());
                builder.append(",");
                }
        context.write(key.getYearMonth(), new Text(builder.toString()));
    }
}
```

Temperature: Pair (excerpt)

```
public class DateTemperaturePair
    implements Writable, WritableComparable<DateTemperaturePair> {
    private final Text yearMonth = new Text();
    private final Text day = new Text();
    private final IntWritable temperature = new IntWritable();

    public DateTemperaturePair() {
    }
    public DateTemperaturePair(String yearMonth, String day, int temperature) {
        this.yearMonth.set(yearMonth);
        this.day.set(day);
        this.temperature.set(temperature);
    }
    public static DateTemperaturePair read(DataInput in) throws IOException {
        DateTemperaturePair pair = new DateTemperaturePair();
        pair.readFields(in);
        return pair;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        yearMonth.write(out);
        day.write(out);
        temperature.write(out);
…
}
```

Temperature: Pair compareTo

```
…
/**
* This comparator controls the sort order of the keys.
*/
public int compareTo(DateTemperaturePair pair) {
  int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
  if (compareValue == 0) {
      compareValue = temperature.compareTo(pair.getTemperature());
  }
  //return compareValue; // sort ascending
  return -1*compareValue; // sort descending
}
```

Temperature: Comparator

```
public class DateTemperatureGroupingComparator
    extends WritableComparator {

     public DateTemperatureGroupingComparator() {
         super(DateTemperaturePair.class, true);
     }

     @Override
    /**
     * Compare two objects
     *
     * @param wc1 a WritableComparable object, which represents a DateTemperaturePair
     * @param wc2 a WritableComparable object, which represents a DateTemperaturePair
     * @return 0, 1, or -1 (depending on the comparison of two DateTemperaturePair objects).
      */
     public int compare(WritableComparable wc1, WritableComparable wc2) {
         DateTemperaturePair pair = (DateTemperaturePair) wc1;
         DateTemperaturePair pair2 = (DateTemperaturePair) wc2;
         return pair.getYearMonth().compareTo(pair2.getYearMonth());
     }
}
```

Temperature: Partitioner

```
public class DateTemperaturePartitioner
   extends Partitioner<DateTemperaturePair, Text> {

    @Override
    public int getPartition(DateTemperaturePair pair,
                            Text text,
                            int numberOfPartitions) {
        // make sure that partitions are non-negative
        return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
    }
}
```

Hadoop provides a plug-in architecture for injecting the custom partitioner code into the framework. This is how we do so inside the driver class:

```
import org.apache.hadoop.mapreduce.Job;
...
Job job = ...;
...
job.setPartitionerClass(TemperaturePartitioner.class);
```

Outline

- Indexing Data
- Secondary Sort
- Top N List
- Left Outer Join

Top N List

Problem:

Given a set of (key-as-string, value-as-integer) pairs, we want to create a top N (where N > 0) list.

Examples for this type of problem:

TopN formalized

Let L be a List<Tuple2<T,Integer>>, where T can be any type (such as a string or URL); L.size() = S and S > N with N > 0.

Elements of L: $\{(K_i, V_i), 1 \leq i \leq S\}$

- where K_i has a type of T and
- V_i is an Integer type (this is the frequency of K_i).

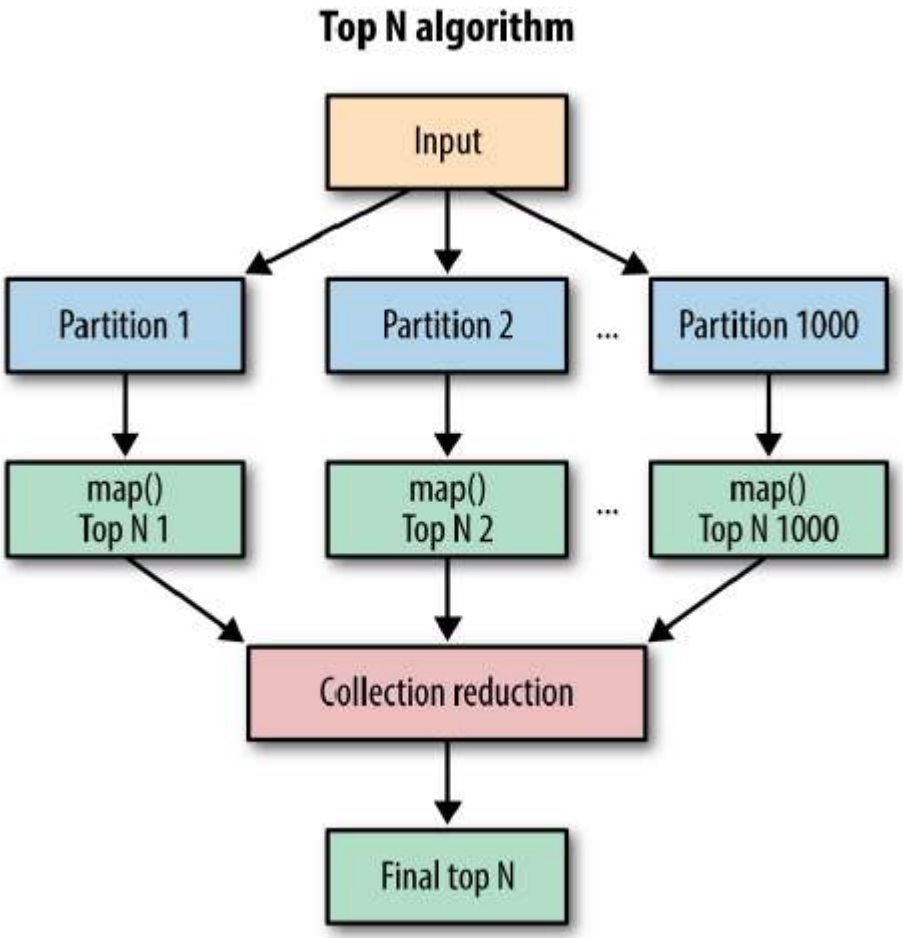
sort(L) returns sorted values of L by using the frequency as a key:

$\{(A_j, B_j), 1 \leq j \leq S; B_1 \geq B_2 \geq \ldots \geq B_S\}$ where $(A_j, B_j) \in L$.

The top N of L is defined as:

$\text{topN}(L) = \{(A_j, B_j), 1 \leq j \leq N, B_1 \geq B_2 \geq \ldots \geq B_N \geq B_{N+1} \geq \ldots \geq B_S\}$

TopN: Overview MapReduce



TopN: Mapper

```
public class TopNMapper extends
   Mapper<Text, IntWritable, NullWritable, Text> {
   private int N = 10; // default
   private SortedMap<Integer, String> top = new TreeMap<Integer, String>();

   @Override
   public void map(Text key, IntWritable value, Context context)
         throws IOException, InterruptedException {

      String keyAsString = key.toString();
      int frequency =  value.get();
      String compositeValue = keyAsString + "," + frequency;
      top.put(frequency, compositeValue);
      // keep only top N
      if (top.size() > N) {
         top.remove(top.firstKey());
      }
   }
   protected void cleanup(Context context) throws IOException,
         InterruptedException {
      for (String str : top.values()) {
         context.write(NullWritable.get(), new Text(str));
   }
}
}
```

TopN: Reducer

```
public class TopNReducer  extends
   Reducer<NullWritable, Text, IntWritable, Text> {
   private int N = 10; // default
   private SortedMap<Integer, String> top = new TreeMap<Integer, String>();

   @Override
   public void reduce(NullWritable key, Iterable<Text> values, Context context)
      throws IOException, InterruptedException {
      for (Text value : values) {
         String valueAsString = value.toString().trim();
         String[] tokens = valueAsString.split(",");
         String url = tokens[0];
         int frequency =  Integer.parseInt(tokens[1]);
         top.put(frequency, url);
         // keep only top N
         if (top.size() > N) {
            top.remove(top.firstKey());
      }}
      // emit final top N
        List<Integer> keys = new ArrayList<Integer>(top.keySet());
        for(int i=keys.size()-1; i>=0; i--){
         context.write(new IntWritable(keys.get(i)), new
Text(top.get(keys.get(i)))); }}
```

Outline

- Indexing Data
- Secondary Sort
- Top N List
- Left Outer Join
- Summary

Motivation - eCommerce

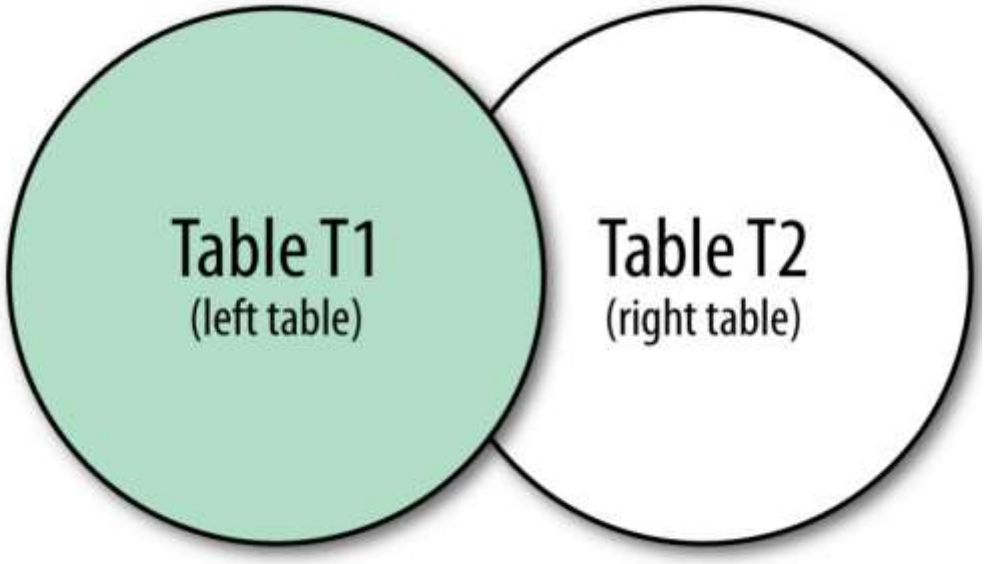
The users data consists of users' location information and the transactions data includes user identity information, but no direct information about a user's location.

```
users(user_id, location_id)
transactions(transaction_id, product_id, user_id,
quantity, amount)
```

Our goal is to find the number of unique locations in which each product has been sold.

SQL

```
SELECT field_1, field_2, ...
   FROM T1 LEFT OUTER JOIN T2
      ON T1.K = T2.K;
```



The result of a left outer join for relations T1 and T2 on the join key of K contains all records of the left table (T1), even if the join condition does not find any matching record in the right table (T2).

Query 1: find all products sold and their associated locations

```
SELECT product_id, location_id
FROM transactions LEFT OUTER JOIN users
ON transactions.user_id = users.user_id;
```

Query 2: find all products sold and their associated location counts

```
SELECT product_id, count(location_id)
FROM transactions LEFT OUTER JOIN users
ON transactions.user_id = users.user_id group by product_id;
```

Query 3: find all products sold and their unique location counts

```
SELECT product_id, count(distinct location_id)
FROM transactions LEFT OUTER JOIN users
ON transactions.user_id = users.user_id group by product_id;
```

MapReduce

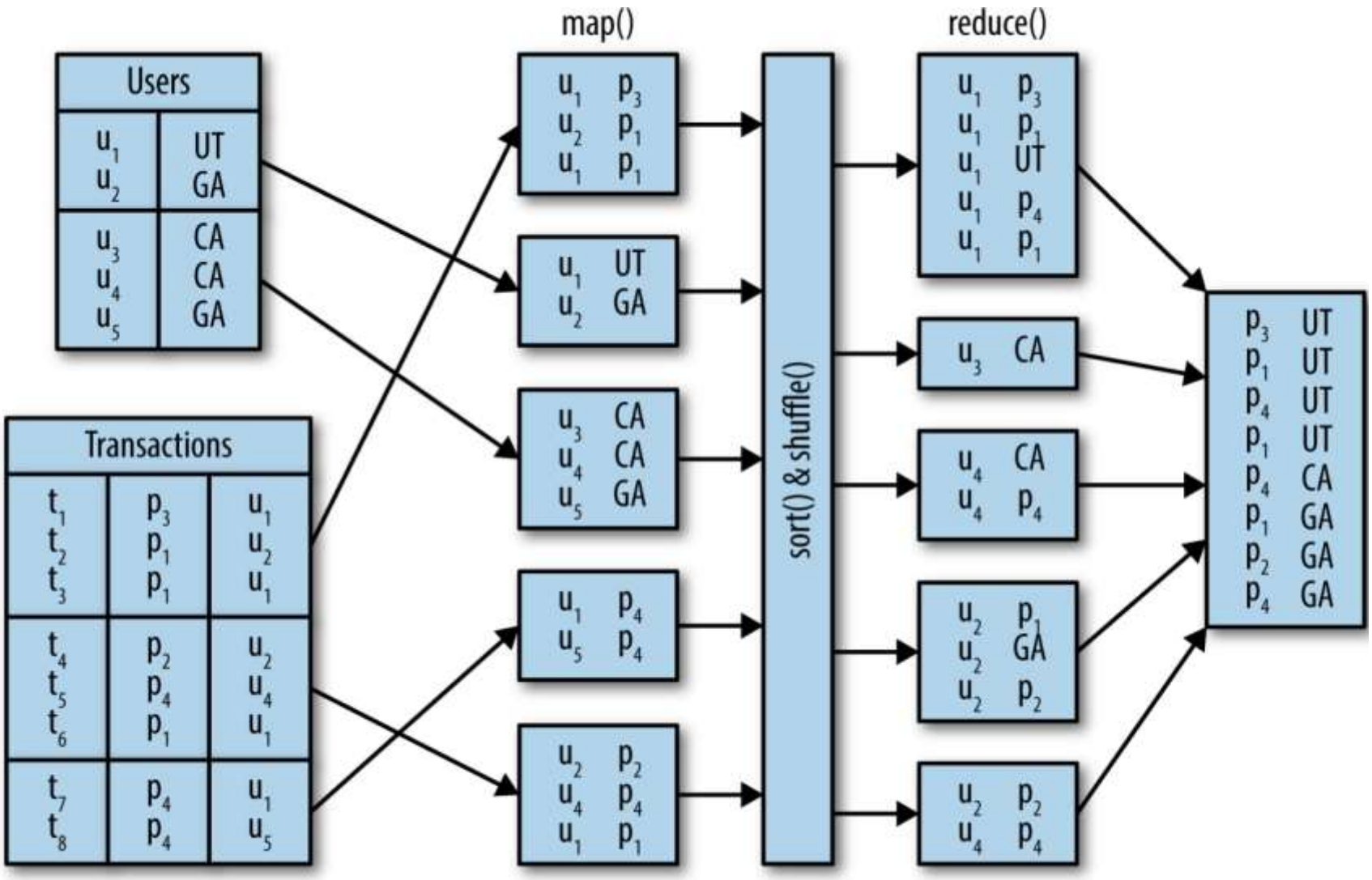
MapReduce phase 1:

find all products sold (and their associated locations). We accomplish this using SQL query 1 from the previous section.

MapReduce phase 2: find all products sold (and their associated unique location counts). We accomplish this using SQL query 3 from the previous section.

```
import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;
...
Job job = new Job(...);
...
Path transactions = <hdfs-path-to-transactions-data>;
Path users = <hdfs-path-to-users-data>;
MultipleInputs.addInputPath(job,
                            transactions,
                            TextInputFormat.class,
                            TransactionMapper.class);
MultipleInputs.addInputPath(job,
                            users,
                            TextInputFormat.class,
                            UserMapper.class);
```

Left outer join data flow, phase 1



The core pieces of the left outer join data flow are

- Transaction mapper
  The transaction map() reads (transaction_id, product_id, user_id, quantity, amount) and emits a key-value pair composed of (user_id, product_id) .
- User mapper
  The user map() reads (user_id, location_id) and emits a key-value pair composed of (user_id, location_id) .

Transaction mapper (version 2)

```
1 /**
2  * @param key is framework generated, ignored here
3  * @param value is the
4  *     transaction_id<TAB>product_id<TAB>user_id<TAB>quantity<TAB>amount
5  */
6 map(key, value) {
7     String[] tokens = StringUtil.split(value, "\t");
8     String productID = tokens[1];
9     String userID = tokens[2];
10    outputKey = Pair(userID, 2);
11    outputValue = Pair("P", productID);
12    emit(outputKey, outputValue);
13 }
```

User mapper (version 2)

```
1  /**
2   * @param key is framework generated, ignored here
3   * @param value is the user_id<TAB>location_id
4   */
5  map(key, value) {
6     String[] tokens = StringUtil.split(value, "\t");
7     String userID = tokens[0];
8     String locationID = tokens[1];
9     outputKey = Pair(userID, 1); // make sure location shows before products
10    outputValue = Pair("L", locationID);
11    emit(outputKey, outputValue);
12 }
```

The reducer for phase 1 (version 2)

```
10  * NOTE that the Pair<"L", locationID> arrives
11  * before all product pairs. The first value is location;
12  * if it's not, then we don't have a user record, so we'll
13  * set the locationID as "undefined".
14  */
15 reduce(key, values) {
16    locationID = "undefined";
17    for (Pair<left, right> value: values) {
18        // the following if-stmt will be true
19        // once at the first iteration
20        if (value.left.equals("L")) {
21            locationID = value.right;
22            continue;
23        }
24
25        // here we have a product: value.left.equals("P")
26        productID = value.right;
27        emit(productID, locationID);
28    }
29 }
```

MapReduce Phase 2: Counting Unique Locations

Mapper phase 2: counting unique locations

```
/**
  * @param key is product_id
  * @param value is location_id
  */
 map(key, value) {
    emit(key, value);
}
```

Reducer phase 2: counting unique locations

```
reduce(key, values) {
    Set<String> set = new HashSet<String>();
    for (String locationID : values) {
       set.add(locationID);
    }

    int uniqueLocationsCount = set.size();
    emit(key, uniqueLocationsCount);
 }
```

Outline

- Indexing Data
- Secondary Sort
- Top N List
- Left Outer Join
- Summary

Key Points

Inverted Index

- Inverse Mapper: emit (term, file) and Identity Reducer
- One of the first applications for MapReduce with TF-IDF, PageRank

TopN

- Typical question in analysis

Secondary Sort

- Define a composite key type with natural key and secondary key
- Partion by natural key
- Define comparators for Sorting (by both keys) and grouping (by natural key)