Lesson 2

MapReduce Basics

Divide and Conquer

For example, the following are just some of the issues that need to be addressed:

- How do we break up a large problem into smaller tasks? More specifically, how do we decompose the problem so that the smaller tasks can be executed in parallel?

- How do we assign tasks to workers distributed across a potentially large number of machines (while keeping in mind that some workers are better suited to running some tasks than others, e.g., due to available resources, locality constraints, etc.)?

- How do we ensure that the workers get the data they need?

- How do we coordinate synchronization among the different workers?

- How do we share partial results from one worker that is needed by another?

- How do we accomplish all of the above in the face of software errors and hardware faults?

Divide and Conquer

- One of the most significant advantages of MapReduce is that it provides an abstraction that hides many system-level details from the programmer. Therefore, a developer can focus on what computations need to be performed, as opposed to how those computations are actually carried out or how to get the data to the processes that depend on them.

- MapReduce provides a means to distribute computation without burdening the programmer with the details of distributed computing.

- Large-data processing by definition requires bringing data and code together for computation to occur. MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a scalable, robust, and efficient manner.

- Instead of moving large amounts of data around, it is far more efficient, if possible, to move the code to the data. This is operationally realized by spreading data across the local disks of nodes in a cluster and running processes on nodes that hold the data. The complex task of managing storage in such a processing environment is typically handled by a distributed file system that sits underneath MapReduce.

Functional Programming

MapReduce has its roots in functional programming (eg. ML, LISP)

A key feature of functional languages is the concept of higher order functions, or functions that can accept other functions as arguments.

Two common built-in higher order functions are map and fold.

Given a list, map takes as an argument a function f (that takes a single argument) and applies it to all elements in a list.

Functional Programming

Given a list, fold takes as arguments a function g (that takes two arguments) and an initial value: g is first applied to the initial value and the first item in the list, the result of which is stored in an intermediate variable. This intermediate variable and the next item in the list serve as the arguments to a second application of g, the results of which are stored in the intermediate variable. This process repeats until all items in the list have been consumed; fold then returns the final value of the intermediate variable.

Typically, map and fold are used in combination.

Sum of squares

For example, to compute the sum of squares of a list of integers, one could map a function that squares its argument (that is, $f(x) = x^2$) over the input list,

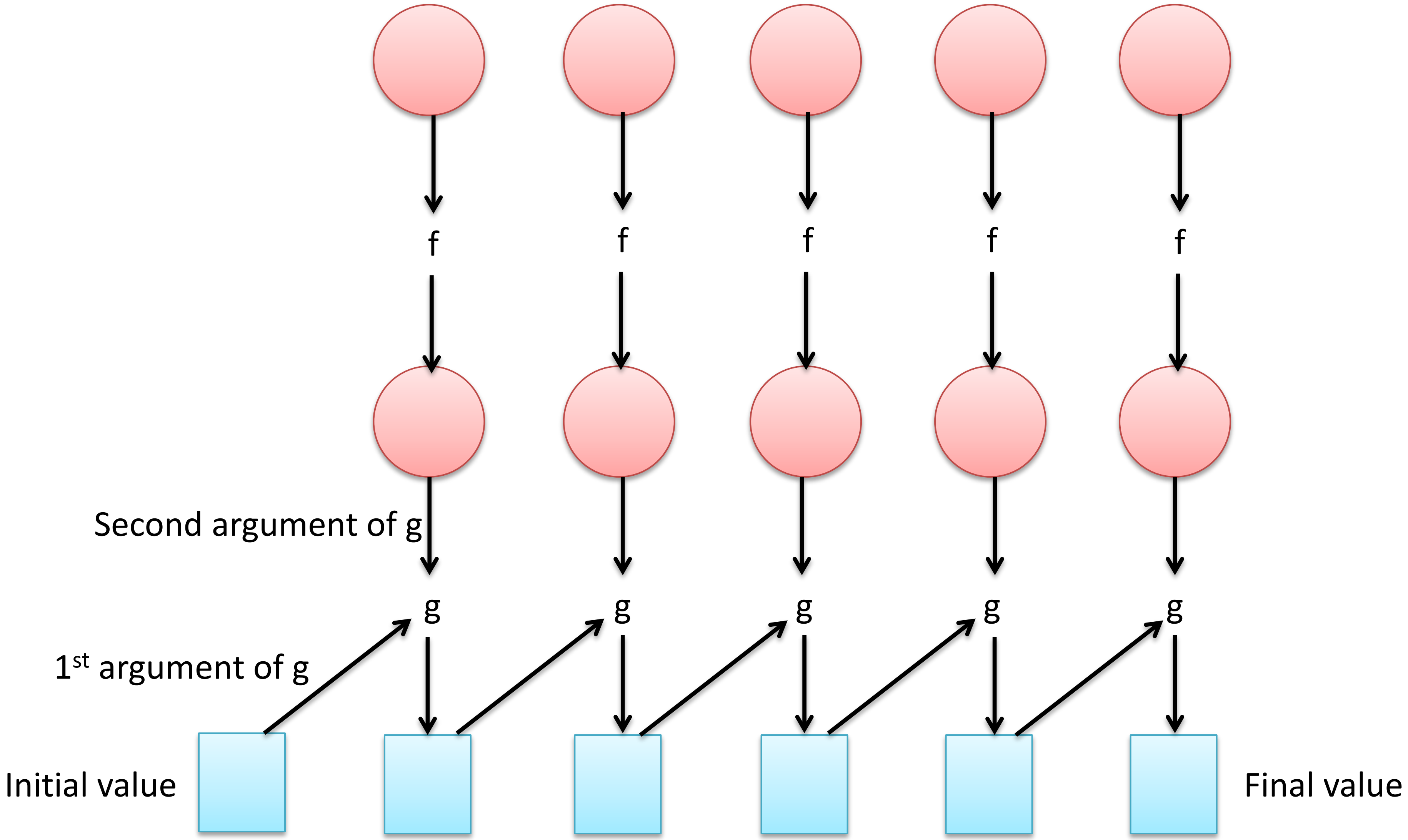
$$\{2, 7, 9\} \xrightarrow{\text{Map(f)}} \{4, 49, 81\}$$

and then fold the resulting list with the addition function (more precisely, $g(x, y) = x + y$) using an initial value of zero.

$$\{4, 49, 81\} \xrightarrow{\text{Fold(g, 0)}} 134$$

$(0 + 4 = 4, 4 + 49 = 53, 53 + 81 = 134)$

Map and Fold


f
f
f
f
f
Second argument of g
g
g
g
g
g
1st argument of g
Initial value
Final value

Another Example of Map and Fold

Write a Map and Fold Algorithm to classify an array into three different category.

A) Has more even numbers
B) Has more odd numbers
C) Has equal number of even and odd numbers

Solution

Part 1: pseudo code for f.

```
int f(x)
{          return (x % 2 == 0) ? 1 : -1;     }
```

Part 2: pseudo code for g.

```
int g(x, y)
{          return x + y;     }
```

Part 3: Specification of initial value.

Initial value = 0

Part 4: Interpretation of final value.

+ integer : more evens, - integer: more odds, 0: equal

Another Example of Map and Fold

A **wave array** is defined to an array which does **not** contain two even numbers or two odd numbers in adjacent locations.

{7, 2, 9, 10, 5}, {4, 11, 12, 1, 6}, {1, 0, 5},{2} are all **wave arrays**.

{**2, 6**, 3, 4} is not a wave array because the even numbers 2 and 6 are adjacent to each other.

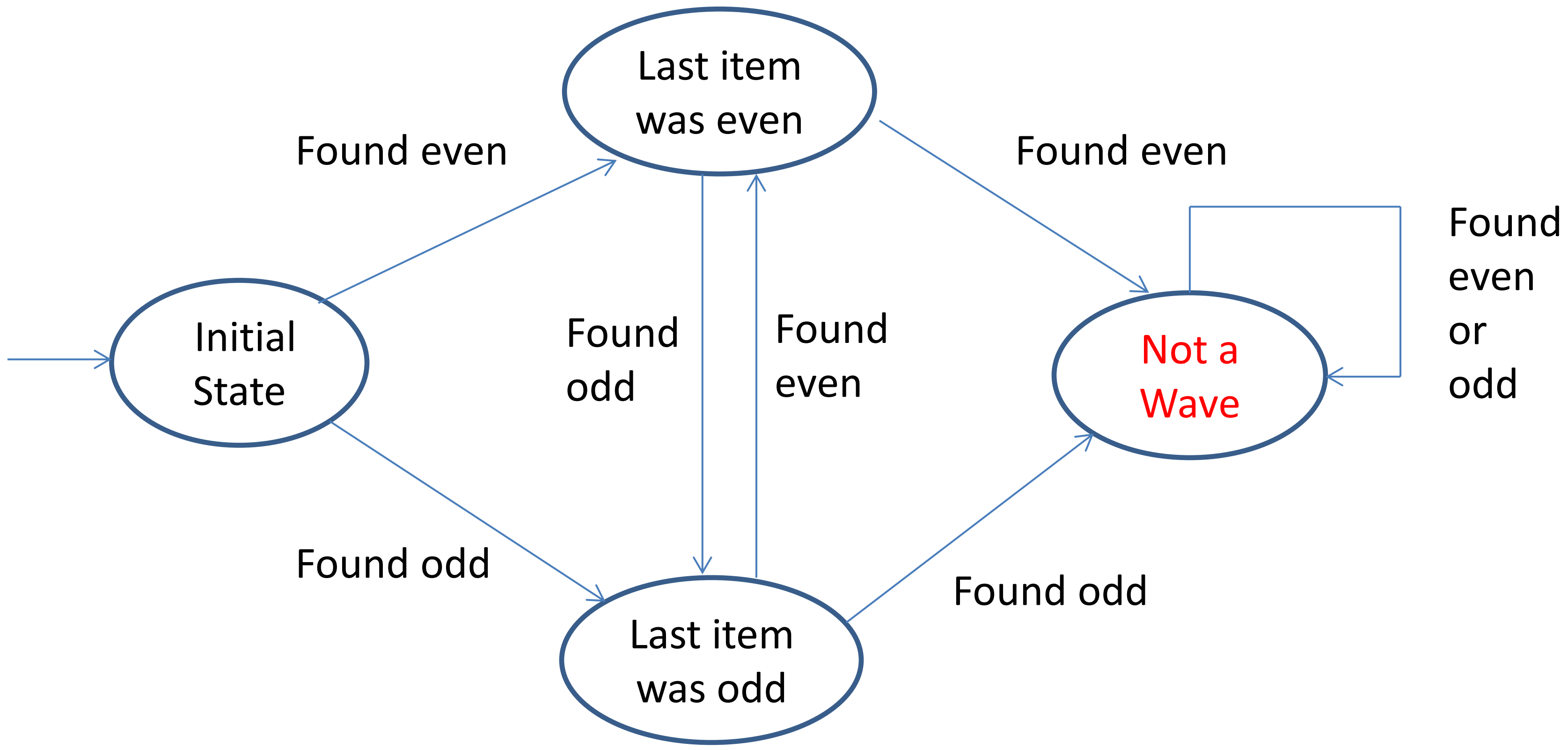
{3, 4, 9, 11, 8} is not a wave array because the odd numbers 9 and 11 are adjacent to each other.

You can assume array has at least one element.

State Diagram : isWave

All states other than initial state is final.

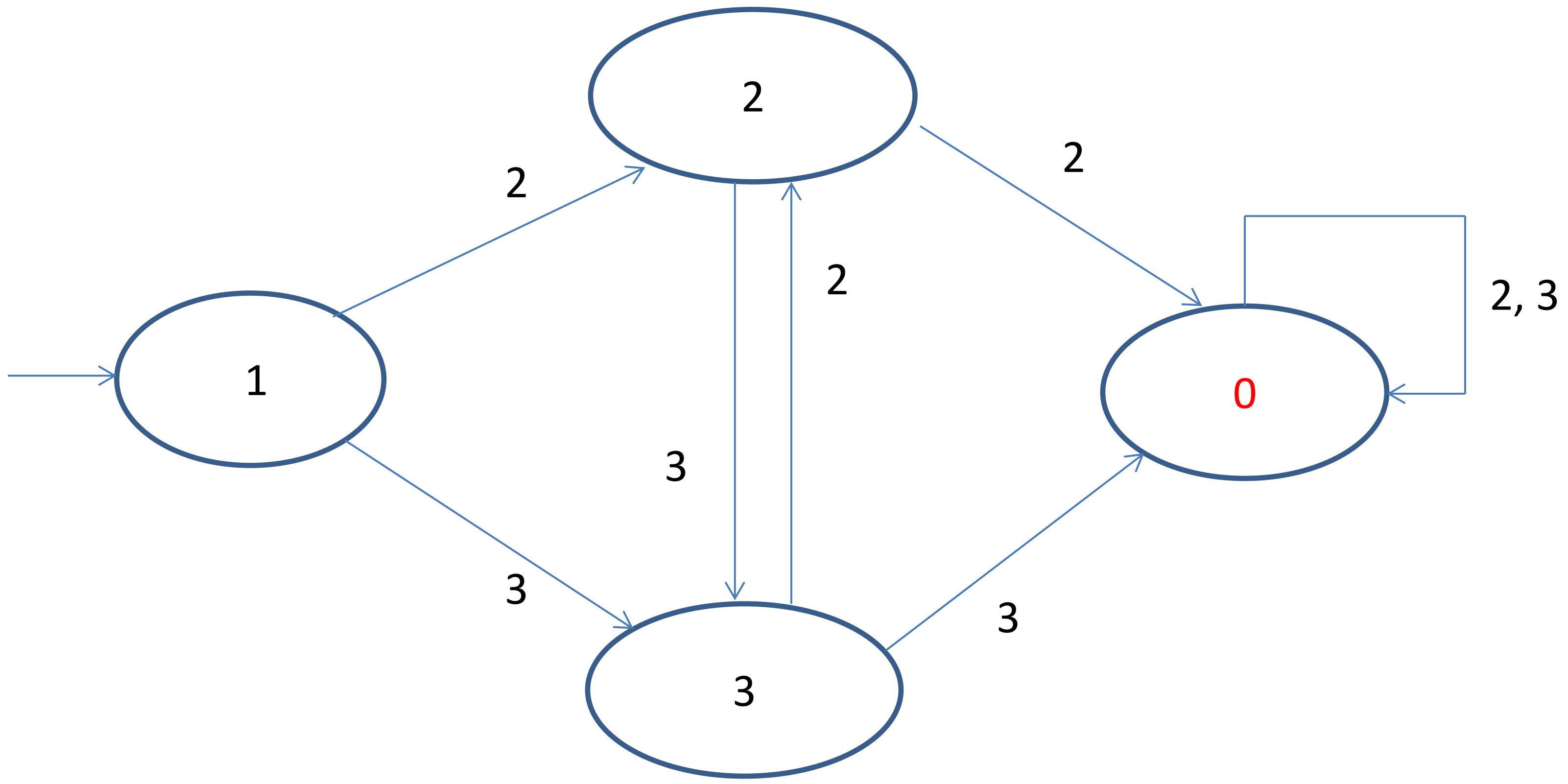
Last item odd (even) : Wave



State Diagram : isWave

All states other than initial state is final.

Last item odd (even) : Wave



$$\begin{tikzcd}
 & 2 \arrow[d, shift right, "3"'] \arrow[dr, "2"] & \\
\arrow[r] 1 \arrow[ur, "2"] \arrow[dr, "3"'] & & 0 \arrow[loop right, "{2, 3}"] \\
 & 3 \arrow[uu, shift right, "2"'] \arrow[ur, "3"'] &
\end{tikzcd}$$

Solution

Part 1: pseudo code for f.

```
int f(x)
{         return (x % 2) + 2 ;      }
```

Part 2: pseudo code for g.

```
int g(x, y)
{         if (x == 0 || x == y) return 0;
          return y;          }
```

Part 3: Specification of initial value.

Initial value = 1

Part 4: Interpretation of final value.

0: not a wave   **2,3 : wave**

Observations

We can view map as a concise way to represent the transformation of a dataset (as defined by the function f).

We can view fold as an aggregation operation, as defined by the function g.

One immediate observation is that the application of f to each item in a list (or more generally, to elements in a large dataset) can be parallelized in a straightforward manner, since each functional application happens in isolation. In a cluster, these operations can be distributed across many different machines.

The fold operation, on the other hand, has more restrictions on data locality. Elements in the list must be brought together before the function g can be applied.

Mappers and Reducers

Key-value pairs form the basic data structure in MapReduce. Keys and values may be primitives such as integers, floating point values, strings, and raw bytes, or they may be arbitrarily complex structures (lists, tuples, associative arrays, etc.). Programmers typically need to define their own custom data types.

In MapReduce, the programmer defines a mapper and a reducer with the following signatures:

- map: (k1; v1) $\longrightarrow$ [(k2; v2)]
- reduce: (k2; [v2]) $\longrightarrow$ [(k3; v3)]

Mappers and Reducers

The input to a MapReduce job starts as data stored on the underlying distributed file system. The mapper is applied to every input key-value pair (split across an arbitrary number of files) to generate an arbitrary number of intermediate key-value pairs.

The reducer is applied to **all values** associated with the same intermediate key to generate output key-value pairs.

Implicit between the map and reduce phases is a distributed “group by" operation on intermediate keys.

Mappers and Reducers

Intermediate data arrive at each reducer in order, sorted by the key.

However, no ordering relationship is guaranteed for keys across different reducers.

Output key-value pairs from each reducer are written persistently back onto the distributed file system (whereas intermediate key-value pairs are transient and not preserved).

The output ends up in r files on the distributed file system, where r is the number of reducers.

For the most part, there is no need to consolidate reducer output, since the r files often serve as input to yet another MapReduce job.

Word count algorithm

```
class Mapper
    method Map(docid a; doc d)
        for all term t in doc d do
            Emit(term t; count 1)
class Reducer
    method Reduce(term t; counts [c1; c2; …])
        sum  = 0
        for all count c in counts [c1; c2; …] do
            sum =  sum + c
        Emit(term t; count sum)
```

Word count algorithm

This algorithm counts the number of occurrences of every word in a text collection, which may be the first step in, for example, building a unigram language model (i.e., probability distribution over words in a collection).

Input key-values pairs take the form of (docid, doc) pairs stored on the distributed file system, where the former is a unique identifier for the document, and the latter is the text of the document itself.

Word count algorithm

The mapper takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for every word: the word itself serves as the key, and the integer one serves as the value (denoting that we've seen the word once).

The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Therefore, in our word count algorithm, we simply need to sum up all counts (ones) associated with each word.

Word count algorithm

The reducer does exactly this, and emits final key-value pairs with the word as the key, and the count as the value. Final output is written to the distributed file system, one file per reducer.

Words within each file will be sorted by alphabetical order, and each file will contain roughly the same number of words.

The partitioner controls the assignment of words to reducers.

The output can be examined by the programmer or used as input to another MapReduce program.

Hadoop vs. Google

In Hadoop, the reducer is presented with a key and an iterator over all values associated with the particular key. The **values are arbitrarily ordered.**

Google's implementation allows the programmer to specify a secondary sort key for ordering the values (if desired) in which case values associated with each key would be presented to the developer's reduce code **in sorted order**.

In Google's implementation the programmer is **not allowed to change the key in the reducer**. That is, the reducer output key must be exactly the same as the reducer input key.

In Hadoop, there is no such restriction, and the **reducer can emit an arbitrary number of output key-value pairs** (with different keys).

Mappers and Reducers

Mappers and reducers are objects that implement the Map and Reduce methods, respectively. In Hadoop, a mapper object is initialized for each map task (associated with a particular sequence of key-value pairs called an input split) and the Map method is called on each key-value pair by the execution framework. In a MapReduce job, the programmer provides

a hint on the number of map tasks to run, but the execution framework makes the final determination based on the physical layout of the data. The situation is similar for the reduce phase:

a reducer object is initialized for each reduce task, and the Reduce method is called once per intermediate key. In contrast with the number of map tasks, the programmer can precisely specify the number of reduce tasks.

Restrictions on Mappers and Reducers

Mappers and reducers can express arbitrary computations over their inputs. However, one must generally be careful about use of external resources since multiple mappers or reducers may be contending for those resources. For example, it may be unwise for a mapper to query an external SQL database.

Mappers can emit an arbitrary number of intermediate key-value pairs, and they need not be of the same type as the input key-value pairs. Similarly, reducers can emit an arbitrary number of final key-value pairs, and they can differ in type from the intermediate key-value pairs.

Side effects

Mappers and reducers can have side effects. It may be useful for mappers or reducers to have external side effects, such as writing files to the distributed file system. Since many mappers and reducers are run in parallel, and the distributed file system is a shared global resource, special care must be taken to ensure that such operations avoid synchronization conflicts. One strategy is to write a temporary file that is renamed upon successful completion of the mapper or reducer.

Special cases

Case 1. MapReduce programs can contain no reducers, in which case mapper output is directly written to disk (one file per mapper). Example problems: parse a large text collection or independently analyze a large number of images.

Case 2. MapReduce program with no mappers is not possible, although in some cases it is useful for the mapper to implement the identity function and simply pass input key-value pairs to the reducers. This has the effect of sorting and regrouping the input for reduce-side processing.

Special cases

Case 3. Similarly, in some cases it is useful for the reducer to implement the identity function, in which case the program simply sorts and groups mapper output.

Case 4. Running identity mappers and reducers has the effect of regrouping and resorting the input data (which is sometimes useful).