

# Fundamental Techniques

Mirek Riedewald



This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.

# Key Learning Goals

- Write the Map and Reduce functions for sorting in MapReduce. Does the choice of Partitioner matter?
- Does secondary sort require a special type of Partitioner, or could even the simple default hash Partitioner work?
- Why do we not just implement secondary sort as a total sort on a composite key?
- What is the difference between sort and secondary sort? Explain, using an example.

# Key Learning Goals

- How do we implement a Combiner in Spark Scala?
- Which of the following pair-RDD operations performs per-partition aggregation during the grouping stage, before shuffling:  
`groupByKey().reduce()`, `reduceByKey`, `foldByKey`, `aggregateByKey`?
- Is `DataSet's groupBy(...).agg(...)` performing per-partition aggregation before shuffling? Is it removing data columns that are neither in the grouping list nor appear in the aggregation-column list?

# Introduction

- We discuss three fundamental techniques that appear in many big-data analysis problems:
  - Aggregation (with or without grouping)
  - Sort
  - Secondary sort
- Each of them highlights interesting aspects of the underlying system infrastructure.

# Aggregation

- For aggregation, we return to the Word Count program.
  - Recall that it determines the total number of occurrences of each word in a collection of documents.
- We will pay particular attention to the question of how to ensure that data will be aggregated **as early as possible**.
- First, recall the desired distributed execution.

# Refresh: Word Count

Input (a letter represents a word)

b a c c

a c d b

b b c b

Local counting  
task 0

Local counting  
task 1

Local counting  
task 2

(a,1), (b, 1), (c, 2)

(a,1), (b, 1), (c, 1), (d, 1)

(b, 3), (c, 1)

Shuffle

Summation  
task 0

Summation  
task 1

(a,2), (c, 4)

(b, 5), (d, 1)

$h(a) \bmod 2 = 0$   
 $h(b) \bmod 2 = 1$   
 $h(c) \bmod 2 = 0$   
 $h(d) \bmod 2 = 1$

Final output: (word, count) pairs

# Early-Aggregation Opportunities

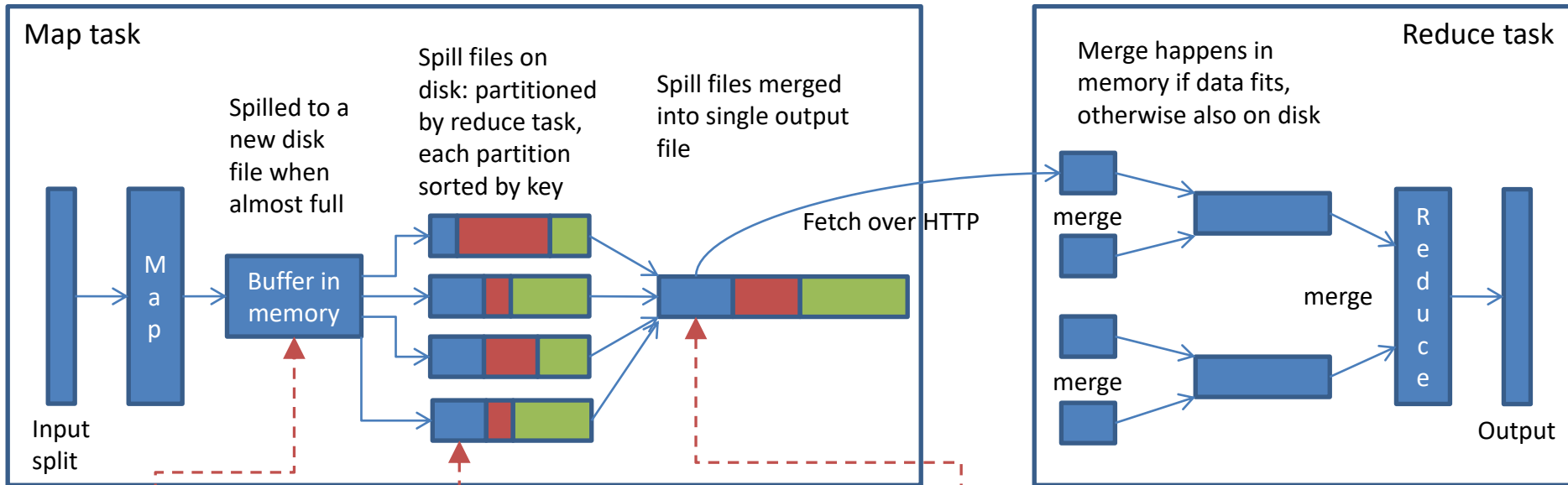
- Consider local counting task 2. It should send (b, 3), not 3 times (b, 1), to the summation task.
  - Replacing k times (word, 1) by a single record (word, k) will reduce network traffic k-fold. It also reduces the cost for the summation task for receiving the data, merging the data (it is more likely to fit in memory or require fewer passes), and final aggregation.
  - On the downside, aggregation increases CPU cost for the local counting task.
- What if local counting tasks 0 and 2 are executed on the same machine—should we aggregate across tasks?
  - No. This complicates system design, because the tasks would have to agree on the final counts. What if one task fails half-way and the other succeeds? What if one gets re-scheduled on a different machine, but not the other?
- In summary, we want to aggregate *within* each local counting task, but aggregation *across* tasks is left to the summation tasks.

# Early Aggregation in MapReduce

- In MapReduce, local aggregation happens in Map tasks, before transferring the data to the Reducers. Hence we want to aggregate counts for all words that appear in the **same input split**.
- In Hadoop MapReduce, each map function call sees only a single line of text. Hence it cannot aggregate across different lines.
- We saw the **Combiner** feature, which elegantly supports the desired per-split aggregation.
  - Unfortunately, the Combiner cannot be controlled by the user: the MapReduce system decides when and on which Map output records the Combiner is executed. It might not be executed at all. Or it might be executed only on some subset of the output records, e.g., only the records currently in memory. And it might be executed multiple times, possibly reading records output by an earlier Combiner execution together with “fresh,” recently emitted records.



# Combiner Execution Options



The Combiner could be executed on some or all records in any of these locations, at any time, as often as desired.

Illustration based on White's book

How can the programmer take more control of combining?

# Original MapReduce Program

- This version of the program does not aggregate any counts in the Mapper. It would need a Combiner.

```
map(offset B, line L )  
  for each word x in L do  
    emit(x, 1)
```

```
reduce(word x, [c1, c2,...])  
  total = 0  
  for each c in input list do  
    total += c  
  emit(x, total)
```

# Tally Counts Per Line

```
map(offset B, line L )  
  h = new hashMap // stores count for each word in line L  
  for each word x in L do  
    h[x]++  
  for each word x in h do  
    emit(x, h[x])
```

- Our first attempt combines counts inside a single Map function call. To do so, we need a data structure in Map that tracks the word occurrences in the input line. The Reduce function does not change.
- While simple, this modification will typically not be very effective, because it combines the counts only within a single line of text.
- To increase combining opportunities, we would like to aggregate counts across the **entire Map input split**, not just a single line.

# Tally Counts Per Map Task

- To aggregate counts across the entire Map task input split, we must work at the **task level**, i.e., above individual Map calls. We need a data structure H that is a private member of the Mapper class and can be updated by each Map call in the same task.
  - Notice that H is local to a single task and only accessed by a single thread, i.e., it does not introduce any inter- or intra-task synchronization issues.
- H must be initialized before the first Map function call in the task. This is done in the `setup()` function, which is guaranteed to be executed when the task starts, before any of the Map calls.
- Each Map function call then updates the counts for the words it finds in its input line. However, it does not emit any output!
- To emit the final tally for the entire task, the counts in H must be emitted **after the last Map call** in the task has completed. This is achieved by “emptying out” the contents of H in the `cleanup()` function of the Mapper class. `Cleanup()` is guaranteed to be executed after the last Map call.
- The Reducer remains unchanged.

```
Class Mapper {
    hashMap H

    setup() {
        H = new hashMap
    }

    map(offset B, line L ) {
        for each word x in L do
            H[x]++
    }

    cleanup() {
        for each word x in H do
            emit(x, H[x])
    }
}
```

# Summary of the Design Pattern for Local Aggregation

- The tally-per-task version of Word Count is an example for the **in-mapper combining** design pattern. Its main idea is to preserve state at the task level, across all Map calls in the same task. This pattern can also be applied to Reduce tasks.
- Advantages over using Combiners:
  - The Combiner does not guarantee if, when, or how often it will be executed.
  - A Combiner combines data *after* it was generated. In-mapper combining avoids generating large amounts of intermediate data by immediately aggregating it as it is produced by a Map call. This often reduces local CPU and disk I/O cost for the Mappers.
- Drawbacks compared to Combiners:
  - In-mapper-combining code needs to be integrated with the Mapper code, increasing code complexity and hence the probability of introducing errors.
  - It needs more memory for managing state, e.g., to hold the hashMap H in memory. If the data structure used for in-mapper combining exceeds the amount of available memory, the programmer is forced to write non-trivial memory-management code to page it to disk.

What is the equivalent of in-mapper combining in Spark?

# Refresher: Word Count in Spark

- In the Spark program, it is not obvious if the equivalent of a Combiner or in-mapper combining is used.
- This means that we have to (1) read the documentation of `reduceByKey` or (2) find out by asking Spark to explain the execution to us.

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                    .map(word => (word, 1))
                    .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```



# Dissecting ReduceByKey

- `ReduceByKey(_ + _)` is shorthand for `reduceByKey((x,y) => (x+y))`. It iterates through a list of values, adding them one-by-one, as illustrated below.
- Spark understands the semantics of this aggregation and applies it, like in-mapper combining, to the RDD partition before shuffling.
  - The same holds for `foldByKey` and `aggregateByKey`.
- In contrast, `groupByKey` only performs grouping but no combining. Hence `myRDD.groupByKey().reduce(...)` or similar will perform aggregation only after shuffling, i.e., does not do the equivalent of in-mapper combining!

Input list: 

1	1	1	1
---	---	---	---

1	1	1	1
---	---	---	---

Reduce(1, 1) = 2

1	1	1	1
---	---	---	---

Reduce(2, 1) = 3

1	1	1	1
---	---	---	---

Reduce(3, 1) = 4

# Aggregation with DataSets

- The semantics of DataSet functions are a little more complex. To fully appreciate this, let us use a slightly more complicated example and compute annual sales using DataSet myDS with columns (year, month, day, sale).
  - Looking at the Spark Scala API, we find various versions of function `agg`. While powerful and flexible, it only supports aggregation across the *entire* dataset, not for separate groups.
  - Function `groupBy` creates groups based on any column(s).
  - Putting both together, we find the solution:  
`myDS.groupBy("year").sum("sale")`
- Challenge question: Will `myDS.groupBy("year").sum("sale")` perform combining or not? (Recall that pair RDD's `groupByKey` did not perform combining.) What happens to columns month and day? Are they in the output? Are they removed before shuffling?

# DataSet Aggregation in Depth

- Try the code below in the Spark shell and look at the explanation. Read it from bottom to top—it should say something like this:
  - ...SerializeFromObject turns the sequence into a DataSet.

```
val myDS = sc.parallelize(Seq( ("x1", "y1", 1), ("x1", "y2", 2), ("x2", "y3", 3), ("x1", "y4", 4) )).toDF("x", "y", "z")
myDS.groupBy("x").agg(sum($"z")).explain
```

Stage 1:

Partition 0: (x1, y1, 1), (x1, y2, 2)

Partition 1: (x2, y3, 3), (x1, y4, 4)

# DataSet Aggregation in Depth

- Try the code below in the Spark shell and look at the explanation. Read it from bottom to top—it should say something like this:
  - ...SerializeFromObject turns the sequence into a DataSet.
  - Project[\_1 ..., \_3...] projects away all columns except for 1 (x) and 3 (z), which are needed for the aggregate.

```
val myDS = sc.parallelize(Seq( ("x1", "y1", 1), ("x1", "y2", 2), ("x2", "y3", 3), ("x1", "y4", 4) )).toDF("x", "y", "z")
myDS.groupBy("x").agg(sum($"z")).explain
```

Stage 1:

Partition 0: (x1, y1, 1), (x1, y2, 2)

1. Remove y-column: (x1, 1), (x1, 2)

Partition 1: (x2, y3, 3), (x1, y4, 4)

1. Remove y-column: (x2, 3), (x1, 4)

# DataSet Aggregation in Depth

- Try the code below in the Spark shell and look at the explanation. Read it from bottom to top—it should say something like this:
  - ...SerializeFromObject turns the sequence into a DataSet.
  - Project[\_1 ..., \_3...] projects away all columns except for 1 (x) and 3 (z), which are needed for the aggregate.
  - HashAggregate(keys=..., functions=...) indicates grouping key and aggregation function and that aggregation happens locally (like a Combiner).

```
val myDS = sc.parallelize(Seq( ("x1", "y1", 1), ("x1", "y2", 2), ("x2", "y3", 3), ("x1", "y4", 4) )).toDF("x", "y", "z")
myDS.groupBy("x").agg(sum($"z")).explain
```

Stage 1:

Partition 0: (x1, y1, 1), (x1, y2, 2)

1. Remove y-column: (x1, 1), (x1, 2)
2. Group by x-column and sum the z-values in each group: (x1, 3)

Partition 1: (x2, y3, 3), (x1, y4, 4)

1. Remove y-column: (x2, 3), (x1, 4)
2. Group by x-column and sum the z-values in each group: (x2, 3), (x1, 4)

# DataSet Aggregation in Depth

- Try the code below in the Spark shell and look at the explanation. Read it from bottom to top—it should say something like this:
  - ...SerializeFromObject turns the sequence into a DataSet.
  - Project[\_1 ..., \_3...] projects away all columns except for 1 (x) and 3 (z), which are needed for the aggregate.
  - HashAggregate(keys=..., functions=...) indicates grouping key and aggregation function and that aggregation happens locally (like a Combiner).
  - Exchange hashpartitioning(...) indicates shuffling using hash partitioning.

```
val myDS = sc.parallelize(Seq( ("x1", "y1", 1), ("x1", "y2", 2), ("x2", "y3", 3), ("x1", "y4", 4) )).toDF("x", "y", "z")
myDS.groupBy("x").agg(sum($"z")).explain
```

Stage 1:

Partition 0: (x1, y1, 1), (x1, y2, 2)

1. Remove y-column: (x1, 1), (x1, 2)
2. Group by x-column and sum the z-values in each group: (x1, 3)

Partition 1: (x2, y3, 3), (x1, y4, 4)

1. Remove y-column: (x2, 3), (x1, 4)
2. Group by x-column and sum the z-values in each group: (x2, 3), (x1, 4)

Partition(x1) = 1

Partition(x2) = 0

(x1, 3)

(x2, 3)

(x1, 4)

Stage 2:

Partition 0: (x2, 3)

Partition 1: (x1, 3), (x1, 4)

# DataSet Aggregation in Depth

- Try the code below in the Spark shell and look at the explanation. Read it from bottom to top—it should say something like this:
  - ...SerializeFromObject turns the sequence into a DataSet.
  - Project[\_1 ..., \_3...] projects away all columns except for 1 (x) and 3 (z), which are needed for the aggregate.
  - HashAggregate(keys=..., functions=...) indicates grouping key and aggregation function and that aggregation happens locally (like a Combiner).
  - Exchange hashpartitioning(...) indicates shuffling using hash partitioning.
  - HashAggregate(keys=..., functions=...) is the final aggregation per group (like Reduce).

```
val myDS = sc.parallelize(Seq( ("x1", "y1", 1), ("x1", "y2", 2), ("x2", "y3", 3), ("x1", "y4", 4) )).toDF("x", "y", "z")
myDS.groupBy("x").agg(sum($"z")).explain
```

Stage 1:

Partition 0: (x1, y1, 1), (x1, y2, 2)

1. Remove y-column: (x1, 1), (x1, 2)
2. Group by x-column and sum the z-values in each group: (x1, 3)

Partition 1: (x2, y3, 3), (x1, y4, 4)

1. Remove y-column: (x2, 3), (x1, 4)
2. Group by x-column and sum the z-values in each group: (x2, 3), (x1, 4)

Partition(x1) = 1

Partition(x2) = 0

(x1, 3)

(x2, 3)

(x1, 4)

Stage 2:

Partition 0: (x2, 3)

1. Group by x-column and sum the z-values in each group: (x2, 3)

Partition 1: (x1, 3), (x1, 4)

1. Group by x-column and sum the z-values in each group: (x1, 7)

# More DataSet Subtleties

- You might think that applying a function to a DataSet will create another DataSet—like relational operators applied to relations will always produce relations. And you will often be right...
- ...but not always. For example, `groupBy(...)` returns a result of type `RelationalGroupedDataset`, representing a DataSet that is grouped by some column(s). Why do we need this?
  - Consider `myDS.sum(...)` and `myDS.groupBy(...).sum(...)`. If `myDS` is a “simple” DataSet, then the former computes the sum over the entire DataSet, while the latter computes a sum for each group. If `myDS` is of type `RelationalGroupedDataset`, then `myDS.sum(...)` computes the sum for each group.
- In short, you must gain a deep understanding of Spark in order to know what really happens internally. Become familiar with the `explain` function—it provides useful information about the execution of your program.



Let us move on to distributed sorting.

# Distributed Sort

- Sorting is one of the most widely used data-processing operations. How can we sort big data in parallel? First attempt:
  - Since data is already managed as file splits, each task could sort a split locally.
  - Then the sorted splits (called “runs”) are merged, maybe in multiple passes.
- It is easy to see that the local sorting parallelizes nicely. **How about the merge phase?**
  - No matter how many rounds of merging we consider, at the end a *single task* will have to perform the final merge on the entire dataset.
  - This results in poor speedup and does not use multiple workers effectively.
- Can we avoid the merge phase?
  - Yes, as long as we partition the data in a clever way.

# Distributed Sorting Through Range-Partitioning

- Consider a simple scenario with 2 tasks sorting [5, 2, 4, 6, 1, 3]. If all **small** numbers, i.e., [2, 1, 3], are assigned to task 0 and all **large** numbers, i.e., [5, 4, 6] to task 1, then sorting is easy:
  - Task 0 sorts locally and writes [1, 2, 3] to output0.
  - Task 1 sorts locally and writes [4, 5, 6] to output1.
  - Sorted output [1, 2, 3, 4, 5, 6] is obtained by “concatenating” output0 with output1. This is a constant-time operation and requires no merging.
- The challenge is to separate the given data into “small” and “large” numbers, which is called **range partitioning**. Range partitioning ensures that if records  $i$  and  $k$ ,  $i \leq k$ , are assigned to a partition, then all records  $j$  between them, i.e.,  $i \leq j \leq k$ , will be assigned to that same partition.
- We also want the different partitions to be of the same size, so that the local sort work is evenly distributed.

# Challenge Question

- Where have you seen this sorting idea before?
  1. Bubble sort
  2. Merge sort
  3. Quicksort
  4. Heapsort

# Challenge Question

- Where have you seen this sorting idea before?
  1. Bubble sort
  2. Merge sort
  3. Quicksort
  4. Heapsort

In Quicksort, a pivot element is selected and then the data array is re-shuffled so that all elements less than or equal to the pivot come before all those greater than the pivot. Then each of these “halves” of the array is recursively sorted the same way. The ideal choice of pivot element is the data median, i.e., the element with the property that as many other elements are smaller than it as there are elements larger than it.

# Range-Partitioning in MapReduce:

## “Explicit” Approach

- Assume we want to sort key-value pairs by key, using two tasks. First, we find the median of the keys. Exact median computation is expensive, but it can be estimated reliably and cheaply through sampling.
  - To ensure uniform random sampling, all splits of the input data must be accessed. We can reduce cost by sampling from only a few splits, but this might result in a sample that does not represent the entire distribution well.
  - After obtaining a sample that fits in memory, this sample can be sorted on a single worker. The median of the sample is an approximation for the true median.
- Once we know the (approximate) median, we can use the program below to sort the input. All small keys are processed by the Reduce call for intermediate key 0, all larger ones by the one for key 1. There is more to this program:
  - We also need a Partitioner that assigns the smaller input keys to lower task numbers to ensure ordering across tasks.
  - The approach generalizes to more partitions by using more quantiles, not just the median.
- The Reduce function below assumes that its input fits in memory. If not, then smaller partitions need to be created by using more quantiles; or a disk-based sort implementation is needed.

```
map( key k, value v )  
  if (k < median)  
    emit(0, (k, v))  
  else  
    emit(1, (k, v))
```

```
reduce(partitionNumber, [(r1.key, r1.value), (r2.key, r2.value),...])  
  load the value list into memory and sort it on r.key  
  for each record r in the sorted list  
    emit(r)
```

# Challenge Question

- What could go wrong if the median is determined from a few file splits only?

# Challenge Question

- What could go wrong if the median is determined from a few file splits only?
  - Consider a file that is already sorted and assume we only access the first out of a thousand splits. Instead of sampling uniformly from the entire key distribution, we would only sample from the smallest  $1/1000$ -th. This sample would provide a poor representation of the entire distribution, because it does not contain any of the larger keys.



# Range-Partitioning in MapReduce: “Minimalist” Approach

- This approach is more elegant and exploits that the MapReduce system guarantees that for each Reduce task, the assigned set of intermediate keys is processed in key order. We leverage this guarantee for sorting, taking advantage of the fact that MapReduce is already optimized for dealing with big data.
- The program below shows how this is done. Map and Reduce both emit their input 1-to-1, i.e., they are **identity functions**. The “magic” of this program lies in the Partitioner’s `getPartition` function, which assigns the small keys to Reduce task 0 and the large ones to Reduce task 1. Since Reduce function calls in a task are processed in key order, all records are correctly sorted by input key.
  - We need an appropriate key comparator. Since Hadoop MapReduce keys must implement `WritableComparable`, they have a `compareTo` function.
- This idea generalizes to arbitrary range partitions, e.g., quantiles. Instead of an explicit if-then-else or case statement in `getPartition`, Hadoop already offers the `TotalOrderPartitioner` class to assign key ranges to Reduce tasks.

```
map( key k, value v )  
emit(k, v)
```

```
reduce(key k, [v1, v2,...])  
for each value v in the input list  
emit( k, v )
```

```
getPartition( key k )  
if (k < median) return 0  
else return 1
```

# Challenge Question

- Look at the sort program code in the Hadoop distribution and find the following:
  - How does it determine the approximate quantiles?
  - How are they passed to the TotalOrderPartitioner?
  - Where are Mapper and Reducer class defined?
- You can find a copy of the file from Hadoop 3.1.1 at <http://khoury.northeastern.edu/home/mirek/code/Sort.java>

# Sort in a DBMS

- In SQL, sorting is done with the ORDER BY clause. Given a relation R with schema (key, value), the query is
  - `SELECT * FROM R ORDER BY key.`
- The implementation will be chosen automatically by the optimizer. If the data is already range-partitioned, the optimizer will ideally determine that no shuffling is needed.

# Sort in Spark Scala

- The `sortBy(elementFunction)` transformation sorts an RDD by whatever value `elementFunction` returns for an element of the RDD. Typically `elementFunction` extracts a field embedded in a string or non-scalar data type such as list or list of lists etc.
- On a pair RDD, transformation `sortByKey()` will efficiently sort the data by key.
- For `DataSet`, the corresponding transformation is `sort("key")`, assuming the `DataSet` has schema (key, value).
- Sorting is implemented using range partitioning, with range boundaries obtained from a sample, as discussed for MapReduce.

# Comparators in Scala

- For a class to be orderable, it needs to extend Scala's `Ordered` trait (like Java's `Comparable` interface).
  - This means that it must implement the `compare` function.
  - Or it needs to use Scala's `Ordering` trait (like Java's `Comparator` interface). To do so, one defines an object of type `Ordering[myKeyClass]` in the scope of the function calling `sortByKey`.
- Recommendation: If you can change the class, make it extend `Ordered`. If you cannot change it, use `Ordering`.

From sort to secondary sort...

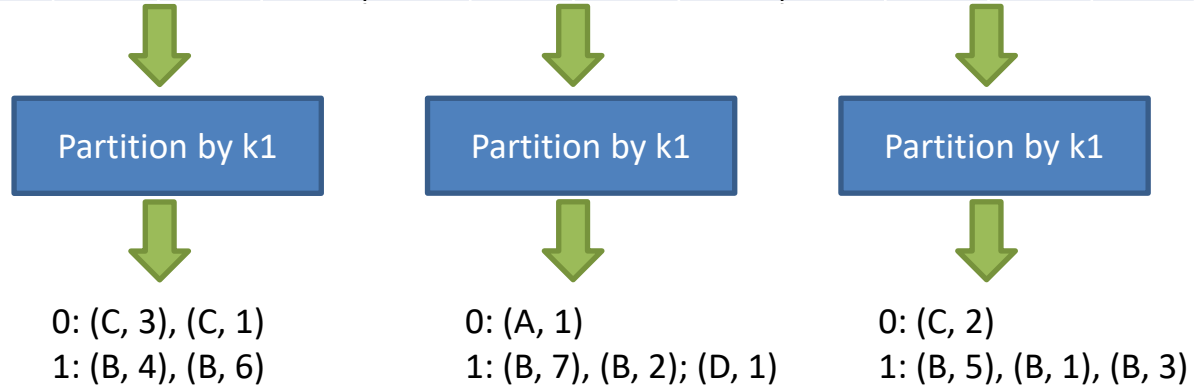
# Secondary Sort

- Secondary sort requires two keys: one for **grouping** the data, and the other to locally **sort each group**. Hence the input conceptually is a set of  $(k_1, k_2, v)$  triples. How can secondary sort be implemented?
- We could perform a **full sort** on composite key  $(k_1, k_2)$ .
  - The drawbacks of this approach are (1) that we waste resources to sort on  $k_1$ , even though we only need to group by it, and (2) that the range partitions might separate keys belonging to the same group. The latter is particularly undesirable, because secondary sort is generally applied with the goal of performing some operation on the (entire) sorted group.
- Both drawbacks can be addressed by partitioning the data on  $k_1$  and sorting each partition on  $k_2$ .
  - A simple hash Partitioner suffices, avoiding the cost of finding approximate quantiles.
  - The sorting part is more subtle as we discuss next.

# Where to Sort?

- This is the obvious approach: the first round of tasks partitions the data, while sorting happens in the next round. As the example illustrates, this can result in high sort cost for tasks receiving bigger groups, e.g., when the input is skewed on key k1.

Input (only keys shown)



$h(A) \bmod 2 = 0$   
 $h(B) \bmod 2 = 1$   
 $h(C) \bmod 2 = 0$   
 $h(D) \bmod 2 = 1$

Sort groups by k2,  
task 0

Groups:  
 [(A, 1)]  
 [(C, 1), (C, 2), (C, 3)]

Sort groups by k2,  
task 1

Groups:  
 [(B, 1), (B, 2), (B, 3), (B, 4), (B, 5), (B, 6), (B, 7)]  
 [(D, 1)]



# Where to Sort?

- This version already sorts each output partition in the Mappers by  $k_2$ . Now the cost for sorting  $k_1=B$  is spread over multiple tasks in round 1. Task 1 in round 2 simply merges the pre-sorted B-records. In addition to better load distribution, this also simplifies the implementation in MapReduce.

Input (only keys shown)



Partition by  $k_1$ ,  
sort by  $k_2$

Partition by  $k_1$ ,  
sort by  $k_2$

Partition by  $k_1$ ,  
sort by  $k_2$

$h(A) \bmod 2 = 0$   
 $h(B) \bmod 2 = 1$   
 $h(C) \bmod 2 = 0$   
 $h(D) \bmod 2 = 1$

0: (C, 1), (C, 3)  
1: (B, 4), (B, 6)

0: (A, 1)  
1: (B, 2), (B, 7); (D, 1)

0: (C, 2)  
1: (B, 1), (B, 3), (B, 5)

Merge groups,  
task 0

Groups:  
 [(A, 1)]  
 [(C, 1), (C, 2), (C, 3)]

Merge groups,  
task 1

Groups:  
 [(B, 1), (B, 2), (B, 3), (B, 4), (B, 5), (B, 6), (B, 7)]  
 [(D, 1)]

# Secondary Sort in MapReduce

- How can we implement this idea in MapReduce?
- We illustrate this with weather observations of type (station, date, temperature). (In practice there will be more fields). Our goal is to compute the *temperature change* between all consecutive measurements by the same station. For simplicity, assume there is at most one temperature reported per station and date.
- Recall that secondary sort uses two keys, but MapReduce allows only one key per record. We want to implement this in a single MapReduce job.
  - What should be the intermediate key of this job?

# First Attempt: Station as Key

- Since we want to group the data by station, **station** would be a natural choice for the intermediate key. This way all weather reports from the same station end up in the same Reduce call. Unfortunately, they are not sorted by date.
  - Sorting then must happen in user code, requiring non-trivial logic if the Reduce input list does not fit in memory.

```
map( station S, date D, temperature T )  
  emit( S, (D, T) )
```

```
reduce( station S, [(D1, T1), (D2, T2),...] )  
  L = load all (D, T) pairs into memory  
  Sort L on the D field  
  Iterate through L and emit all (S, T[i] - T[i-1])
```

# Second Attempt: (Station, Date) as Key

- When using composite key (station, date), Mappers automatically sort and partition on the two fields. The sorting on date is exactly what we wanted, when data for the same station is assigned to the same Reduce task.
  - Recall that Mappers sort all data going to the same Reduce task by key.
  - The Reducers then simply merge the pre-sorted runs, which is cheaper than sorting.
- Unfortunately, there are two drawbacks:
  - Each Reduce call gets only reports from a single date, i.e., it cannot produce temperature differences. In-reducer combining could fix this.
  - The Partitioner might assign records of the same station to different Reduce tasks. Then even a task-level data structure cannot put reports from the same station together across different Reducers.

```
map( station S, date D, temperature T )  
  emit( (S, D), T )
```

```
reduce( (station S, date D), [T] )  
  // We only have the temperature from a  
  // single date  
  // How can we access the other dates for  
  // the same station?
```

# Second Attempt—Problem Solved?

- We can use **in-reducer combining**, like in-mapper combining, to keep track of “state” across the different Reduce calls.
- For this to work, we need the right **Partitioner** that ensures that all dates for a given station are assigned to the same Reduce task. Any Partitioner that only considers the station field for partition assignment will have this property.
- Since all Reduce calls in the same task are executed in key order, we can collect the data in the right order from the Reduce calls.
- This looks a bit clumsy and incurs unnecessary overhead for executing many Reduce calls, each doing trivial work.

```
Class Reducer {
  currentStation
  lastTemp

  setup() {
    currentStation = NULL
    lastTemp = NULL
  }

  reduce( (station S, date D), [T] ) {
    if (S <> currentStation)
      // New station found.
      currentStation = S
      lastTemp = T
    else { // Another call for current station
      emit( S, T - lastTemp)
      lastTemp = T
    }
  }

  cleanup() {}
}
```

```
map( station S, date D, temperature T )
  emit( (S,D), T )
```

```
getPartition( (station S, date D) )
  return myPartitionFct( S )
```

```
keyComparator( (station S, date D) )
  // Sorts on station first.
  // If the station is equal, sorts on date
```

# Third Attempt: Best of Both Worlds

- We use the same Mapper, Partitioner, and key comparator as for the second attempt, but will essentially use the Reducer from the first attempt (without the need for sorting in user code).
- To make this work, we need a special **grouping comparator**. It determines which key-value pairs are passed to the same Reduce function call.

# Pseudo-Code for Third Attempt

```
map( station S, date D, temperature T )  
  emit( (S, D), T )
```

```
getPartition( (station S, date D) )  
  return myPartitionFct( S )
```

```
keyComparator( (station S, date D) )  
  // Sorts on station first.  
  // If the station is equal, sorts on date
```

```
groupingComparator( (station S, date D) )  
  // Considers station only.  
  // Hence two keys with the same station  
  // are considered identical, no matter the  
  // date.
```

```
reduce( station S, [(D1, T1), (D2, T2),...] )  
  for each (Di, Ti) in input list  
    emit( S, Ti - Ti-1 )
```

# How Does this Actually Work?

Assume Reduce task 0 received the example records below from the Mappers. Notice that these records are sorted based on the key comparator (sorted first by station, then by date).

Key: station	Key: date	Value: temperature
S1	2011	80
S1	2013	65
S1	2013	70
S2	2010	75
S2	2010	80
S2	2012	71



# Example (Cont.)

Without the special grouping comparator, there is one Reduce call per distinct (station, date) pair.

	Key: station	Key: date	Value: temperature	
Reduce call for key (S1, 2011)	S1	2011	80	
	S1	2013	65	Reduce call for key (S1, 2013)
	S1	2013	70	
Reduce call for key (S2, 2010)	S2	2010	75	
	S2	2010	80	
	S2	2012	71	Reduce call for key (S2, 2012)

# Example (Cont.)

With the special grouping comparator, keys such as (S1, 2011) and (S1, 2013) are considered identical. Hence, they are processed in the same Reduce function call. In general, there is only a single Reduce function call per station.

Reduce call for key  
(S1, \*)

Key: station	Key: date	Value: temperature
S1	2011	80
S1	2013	65
S1	2013	70
S2	2010	75
S2	2010	80
S2	2012	71

Reduce call for key  
(S2, \*)

# Real Code

- Take a look at the secondary sort program example from Tom White's book at <http://khoury.northeastern.edu/home/mirek/code/MaxTemperatureUsingSecondarySort.java>  
[a](#)

# Secondary Sort in MapReduce Summary

- The general **design pattern** for secondary sort is as follows:
  - To partition by key  $k_1$  and sort each  $k_1$ -group by another key  $k_2$ , make  $(k_1, k_2)$  the intermediate key.
  - Define a key comparator to order by composite key  $(k_1, k_2)$ .
  - Define Partitioner and grouping comparator for key  $(k_1, k_2)$  to consider only  $k_1$  for partitioning and grouping: Their `getPartition` and `compareTo` functions should ignore  $k_2$ .
- Secondary sort is also useful for finding the greatest or smallest value of key  $k_2$  for each key  $k_1$ . However, since MIN and MAX can be found trivially in linear time in the Reduce function input list, using only constant space to hold the current MIN/MAX, secondary sort is not necessarily faster.

# Secondary Sort in a DBMS

- In SQL, GROUP BY cannot return individual records in a group, but only a single record representing an aggregate for the group. Given a relation R with schema (k1, k2, value), the closest to secondary sort would be `SELECT * FROM R ORDER BY k1, k2`. However, this would not compute the desired temperature differences.
- With the introduction of window functions in SQL:2003, one can now write a query like `SELECT k1, value - lag(value) OVER (PARTITION BY k1 ORDER BY k2) FROM D`. The lag() function returns the value of the previous entry in the same partition.

# Secondary Sort in Spark Scala

- For pair RDDs, secondary sort can be achieved using the `repartitionAndSortWithinPartitions(partitionerObject)` transformation. It partitions the RDD using the specified Partitioner and sorts each partition by the key.
  - From the Spark 2.3.0 API (as of September 2018): “This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.”
- Interestingly, when using `DataSet`, one can write window functions like in SQL:2003, using `org.apache.spark.sql.expressions.Window`
  - Define the `DataSet` with schema `(k1, k2, value)`
  - `val groupedSorted = Window.partitionBy(k1).orderBy(k2)`
  - `val diff = lag(value, 1).over(groupedSorted)`

Time to wrap up this module.

# Module Summary: Local Aggregation

- Opportunities to perform **local aggregation** vary depending on problem type and data distribution.
- Combiners can significantly reduce cost by decreasing the amount of data sent from Mappers to Reducers. Unfortunately, the programmer cannot control if and when a Combiner will be executed by Hadoop.
- In-mapper combining can be used instead of a Combiner. It gives the programmer explicit control but requires changes to the Mappers.
- Both Combiners and in-mapper combining are only applicable for certain types of aggregate functions. And they are only effective if many Map output records in the same Map task have the same key.
- In **Spark**, when using pair RDDs, use specialized aggregation operations such as `reduceByKey`, `foldByKey`, and `aggregateByKey`. They perform the equivalent of in-mapper combining—in contrast to using `groupByKey` followed by aggregation.
- When using `DataSets`, `DataSet.groupBy.agg` is the right approach. Here the optimizer automatically determines that “in-mapper combining” can be applied.



# Module Summary: Sorting

- In **MapReduce**, it is best to let the system take care of all sorting, using its highly optimized key-sorting algorithm during the shuffle phase. This eliminates the need for sorting in user code.
  - To do this, sort uses the identity function for both Map and Reduce, relying on range partitioning for correctness.
  - Secondary sort requires the secondary key to be part of the intermediate key. For correctness, Partitioner and grouping comparator need to ignore the secondary key.
- In **Spark**, one should choose the corresponding built-in functionality for sort and secondary sort.